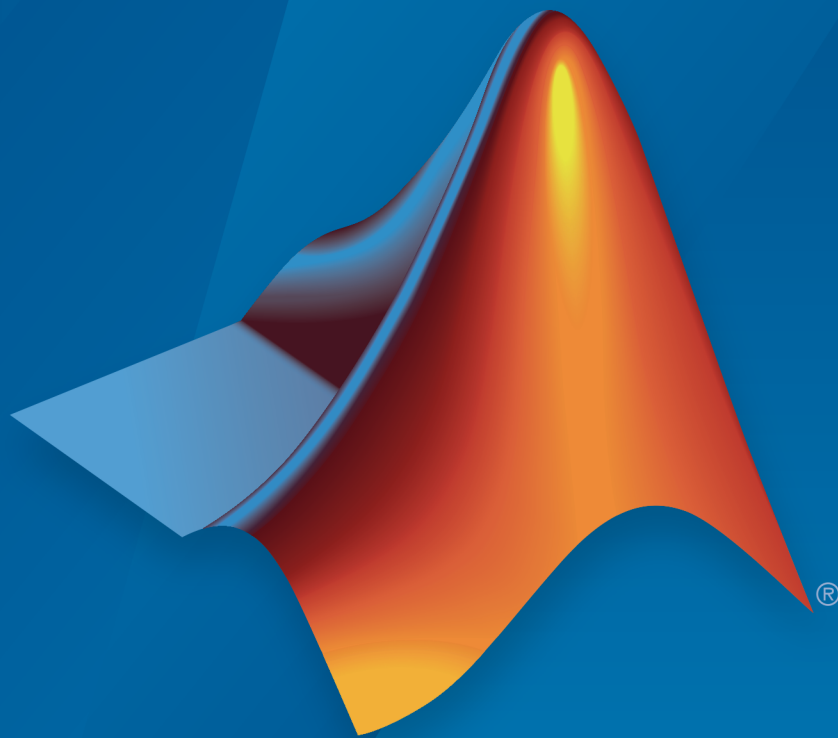


Audio System Toolbox™

Reference



MATLAB® & SIMULINK®

R2016a



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Audio System Toolbox™ Reference Guide

© COPYRIGHT 2016 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 2016 Online only New for Version 1.0 (Release 2016a)

1	<u>Apps in Audio System Toolbox</u>
2	<u>Functions in Audio System Toolbox</u>
3	<u>System objects in Audio System Toolbox</u>
4	<u>Classes in Audio System Toolbox</u>
5	<u>Blocks in Audio System Toolbox</u>

Apps in Audio System Toolbox

Audio Test Bench

Develop, debug, test, and tune audio plugin




Description










The Audio Test Bench provides a graphical interface through which you can develop, debug, test, and tune your audio plugin in real time. You can interact with properties of your audio plugin using associated parameter graphical widgets. See `audioPluginParameter` for more information.

Using the Audio Test Bench, you can:

- Debug your audio plugin.
- Simulate your audio plugin as generated in a digital audio workstation (DAW).
- Visualize your processing with time-domain and frequency-domain scopes.
- Interactively synchronize MIDI controls to plugin properties.

Develop and Test Features

Button	Description
 Run	Run your audio plugin in an audio stream loop using the specified input and output configuration. You can tune parameters of your audio processing algorithm in real time. The MATLAB [®] command line and objects used by the test bench are locked while the test bench is running.
 Pause (appears while audio test bench is running)	Pause audio stream loop. The MATLAB command line is released. Objects used by the test bench remain locked.
 Step Forward	Call the processing function of your audio plugin one time in an audio stream loop, with input and output specified by your input and output configuration.

Button	Description
 Stop	Stop the audio stream loop. The MATLAB command line and objects used by the test bench are released.
 Reset	Reset internal states of your audio plugin.
 View Source Code	Open the source file of your audio plugin.
 Time Scope	Open an instance of <code>dsp.TimeScope</code> , which provides a time-domain visualization of the output from your audio stream loop.
 Spectrum Analyzer	Open an instance of <code>dsp.SpectrumAnalyzer</code> , which provides a frequency-domain visualization of the output from your audio stream loop.
 Synchronize to MIDI Controls	Start the <code>configureMIDI</code> user interface (UI) for your plugin object.
 Audio Test Bench Help	Open MATLAB documentation for Audio Test Bench .
 Configure Input	Open input configuration UI. The UI options depend on whether you choose Audio File Reader or Audio Device Reader for the input to your audio stream loop. For more information, see <code>audioDeviceReader</code> and <code>dsp.AudioFileReader</code> .
 Configure Output	Open output configuration UI. The UI options depend on whether you choose Audio File Writer or Audio Device Writer for the output from your audio stream loop. If you choose to output Both , two dialog boxes open: one for the Audio File Writer and one for the Audio Device Writer . For more information, see <code>dsp.AudioFileWriter</code> and <code>audioDeviceWriter</code> .

Open the Audio Test Bench App

At the MATLAB command prompt, enter:

- `audioTestBench pluginClass`

Opens an audio plugin test bench user interface (UI) for an instance of `pluginClass`.

- `audioTestBench(pluginClassInstance)`

Opens an audio plugin test bench UI for `pluginClassInstance`, where `pluginClassInstance` is an instance of an audio plugin class.

Note: The input to `audioTestBench` must derive from the `audioPlugin` class, not the `audioPluginSource` class.

- `audioTestBench ASTSystemObject`

Opens an audio plugin test bench UI for an instance of a compatible Audio System Toolbox™ System object™.

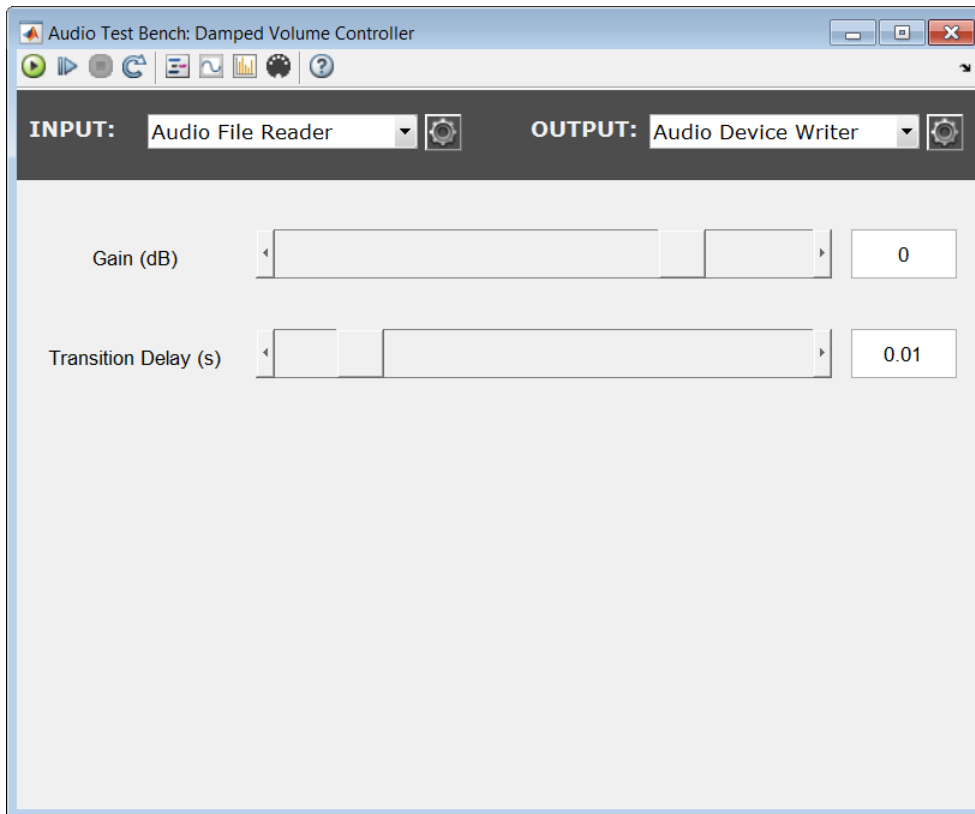
- `audioTestBench(ASTSystemObjectInstance)`

Opens an audio plugin test bench UI for `ASTSystemObjectInstance`, where `ASTSystemObjectInstance` is an instance of a compatible Audio System Toolbox System object.

Examples

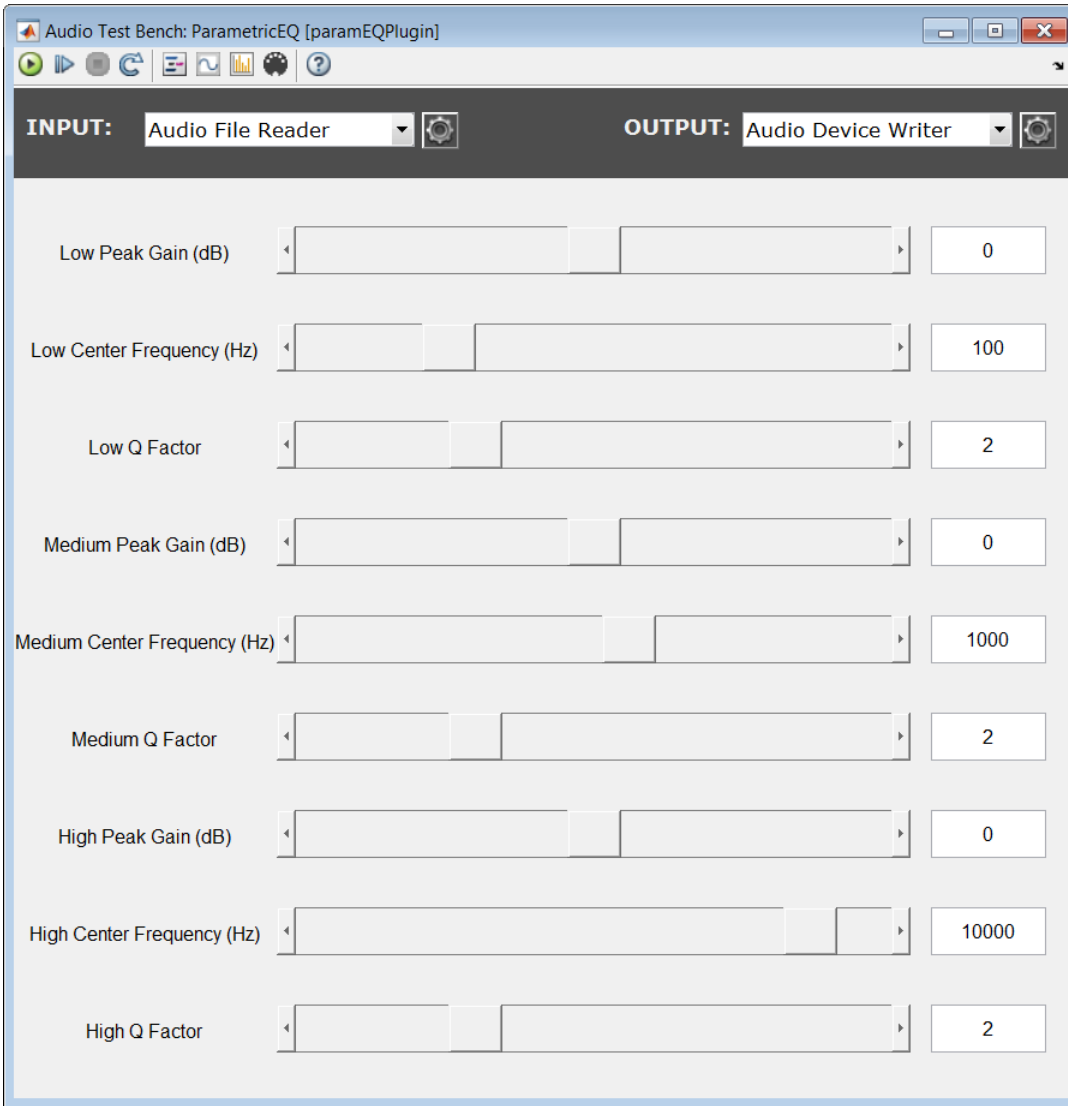
Open Audio Test Bench from Plugin Class

```
audioTestBench audiopluginexample.DampedVolumeController;
```

Open Audio Test Bench from Plugin Object

```
paramEQPlugin = audiopluginexample.ParametricEqualizer;  
audioTestBench(paramEQPlugin);
```



Open Audio Test Bench from Instance of Audio System Toolbox System Object

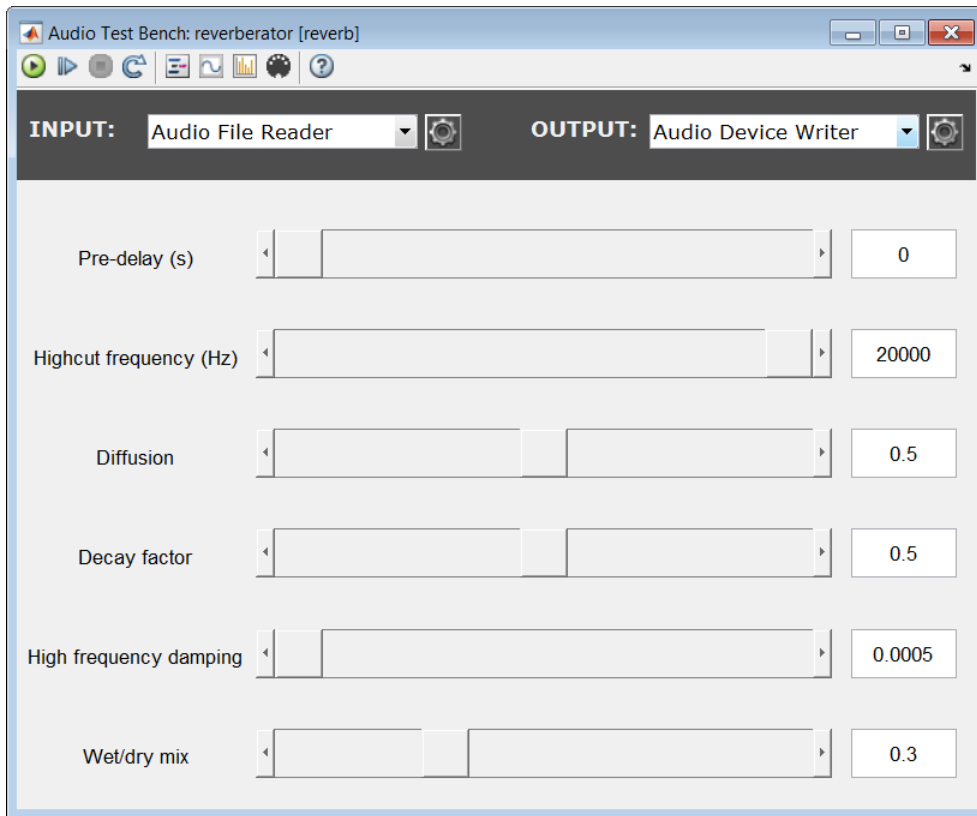
Construct an object from the reverberator System object.

reverb = reverberator

```
reverb =  
    reverberator with properties:  
        PreDelay: 0  
        HighCutFrequency: 20000  
        Diffusion: 0.5000  
        DecayFactor: 0.5000  
        HighFrequencyDamping: 5.0000e-04  
        WetDryMix: 0.3000  
        SampleRate: 44100
```

Open the audio test bench to interact with your System object.

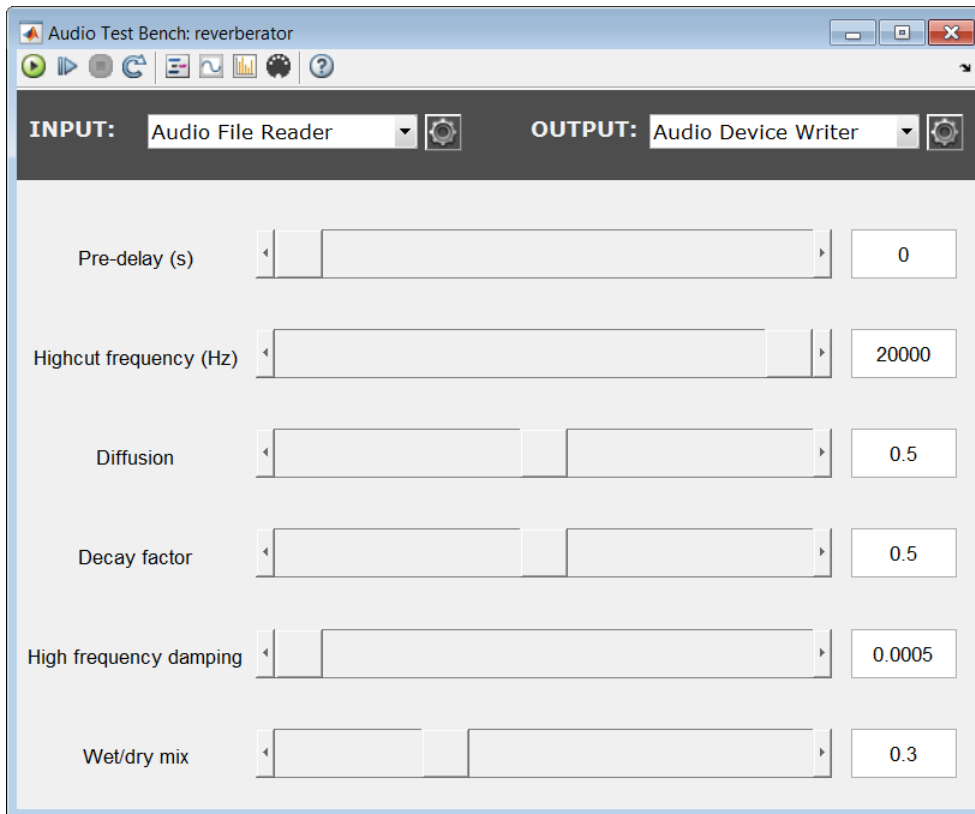
```
audioTestBench(reverb);
```



Open Audio Test Bench from Audio System Toolbox System Object

Open the audio test bench to interact with your plugin.

```
audioTestBench reverberator;
```



- “Use the Audio Test Bench”

More About

- “What Are DAWs, Audio Plugins, and MIDI Controllers?”
- “Design an Audio Plugin”
- “Audio Plugin Example Gallery”

See Also

Functions

`audioPluginInterface` | `audioPluginParameter` | `generateAudioPlugin` | `validateAudioPlugin`

Classes

`audioPluginSource` | `audioPlugin`

Introduced in R2016a

Functions in Audio System Toolbox

audioPluginInterface

Specify audio plugin interface

Syntax

```
PluginInterface = audioPluginInterface  
PluginInterface = audioPluginInterface(pluginParameters)  
PluginInterface = audioPluginInterface(Name,Value)
```

Description

`PluginInterface = audioPluginInterface` returns an object, `PluginInterface`, that specifies the interface of an audio plugin in a digital audio workstation (DAW) environment. It also specifies interface attributes, such as naming for identification.

`PluginInterface = audioPluginInterface(pluginParameters)` specifies audio plugin parameters, which are user-facing variables associated with audio plugin properties. See `audioPluginParameter` for more details.

`PluginInterface = audioPluginInterface(Name,Value)` specifies `audioPluginInterface` properties using one or more `Name,Value` pair arguments.

Examples

Specify Default Audio Plugin Interface

Create a basic audio plugin class definition file.

```
classdef myAudioPlugin < audioPlugin  
    methods  
        function out = process(~,in)  
            out = in;  
        end  
    end  
end
```


Add a constant property, `PluginInterface`, which is specified as an `audioPluginInterface` object.

```
classdef myAudioPlugin < audioPlugin
    properties (Constant)
        PluginInterface = audioPluginInterface;
    end
    methods
        function out = process(~,in)
            out = in;
        end
    end
end
```

Associate Property with Parameter

Create a basic audio plugin class definition file. Specify a property, `Gain`, and a processing function that multiplies input by `Gain`.

```
classdef myAudioPlugin < audioPlugin
    properties
        Gain = 1;
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end
```

Add a constant property, `PluginInterface`, which is specified as an `audioPluginInterface` object.

```
classdef myAudioPlugin < audioPlugin
    properties
        Gain = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface;
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end
```

end

Pass `audioPluginParameter` to `audioPluginInterface`. To associate the plugin property, `Gain`, to a plugin parameter, specify the first argument of `audioPluginParameter` as the property name, `'Gain'`.

```
classdef myAudioPlugin < audioPlugin
    properties
        Gain = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain'));
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end
```

If you generate and deploy `myAudioPlugin` to a digital audio workstation (DAW) environment, the plugin property, `Gain`, synchronizes with a user-facing plugin parameter.

Specify Interface Properties

Create a basic audio plugin class definition file. Specify the plugin name, vendor name, vendor version, unique identification, number of input channels, and number of output channels.

```
classdef monoGain < audioPlugin
    properties
        Gain = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain'),...
            'PluginName','Simple Gain',...
            'VendorName','Cool Company',...
            'VendorVersion','1.0.0',...
            'UniqueId','1a1Z',...
            'InputChannels',1,...
            'OutputChannels',1);
    end
end
```

```

    end
  methods
    function out = process(plugin,in)
      out = in*plugin.Gain;
    end
  end
end
end

```

Input Arguments

pluginParameters — Audio plugin parameters

none (default) | one or more `audioPluginParameter` objects

Audio plugin parameters, specified as one or more `audioPluginParameter` objects.

To create an audio plugin parameter, use the `audioPluginParameter` function. In a digital audio workstation (DAW) environment, they synchronize plugin class properties with user-facing parameters.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'PluginName', 'cool effect', 'VendorVersion', '1.0.2'` specifies the name of the generated audio plugin as `'cool effect'` and the vendor version as `'1.0.2'`.

'PluginName' — Name of generated plugin

name of plugin class (default) | string

Name of your generated plugin, as seen by a host audio application, specified as a comma-separated pair consisting of `'PluginName'` and a string of up to 127 characters. If `'PluginName'` is not specified, the generated plugin is given the name of the audio plugin class it is generated from.

'VendorName' — Vendor name of the plugin creator

'' (default) | string

Vendor name of the plugin creator, specified as the comma-separated pair 'VendorName' and a string of up to 127 characters.

'VendorVersion' — Vendor version

'1.0.0' (default) | dot-separated string

Vendor version used to track plugin releases, specified as a comma-separated pair consisting of 'VendorVersion' and a dot-separated string of 1–3 integers in the range 0 to 9.

Example: '1'

Example: '1.4'

Example: '1.3.5'

'UniqueId' — Unique identifier of plugin

'MwAp' (default) | four-character string

Unique identifier for your plugin, specified as a comma-separated pair consisting of 'UniqueID' and a four-character string, used for recognition in certain digital audio workstation (DAW) environments.

'InputChannels' — Input channels

2 (default) | integer | vector of integers

Input channels, specified as a comma-separated pair consisting of 'InputChannels' and an integer or vector of integers. The *input channels* are the number of input data arguments and associated channels (columns) passed to the processing function of your audio plugin.

Example: 'InputChannels', 3 calls the processing function with one data argument containing 3 channels.

Example: 'InputChannels', [2, 4, 1, 5] calls the processing function with 4 data arguments. The first argument contains 2 channels, the second contains 4 channels, the third contains 1 channel, and the fourth contains 5 channels.

Note: This property is not applicable for audio source plugins, and must be omitted.

'OutputChannels' — Output channels

2 (default) | integer | vector of integers

Output channels, specified a comma-separated pair consisting of 'OutputChannels' and an integer or vector of integers. The *output channels* are the number of input data arguments and associated channels (columns) passed from the processing function of your audio plugin.

Example: 'OutputChannels', 3 specifies the processing function to output one data argument containing 3 channels.

Example: 'OutputChannels', [2,4,1,5] specifies the processing function to output 4 data arguments. The first argument contains 2 channels, the second contains 4 channels, the third contains 1 channel, and the fourth contains 5 channels.

See Also

audioPlugin | audioPluginSource | audioPluginParameter | generateAudioPlugin
| validateAudioPlugin

Introduced in R2016a

audioPluginParameter

Specify audio plugin parameters

Syntax

```
pluginParameter = audioPluginParameter(propertyName)  
pluginParameter = audioPluginParameter(propertyName,Name,Value)
```

Description

`pluginParameter = audioPluginParameter(propertyName)` returns an object, `pluginParameter`, that associates an audio plugin parameter to the audio plugin property specified by `propertyName`. Use the plugin parameter object, `pluginParameter`, as an argument to an `audioPluginInterface` function in your plugin class definition.

In a digital audio workstation (DAW) environment, or when using Audio Test Bench in the MATLAB environment, plugin parameters are tunable, user-facing variables with defined ranges mapped to controls. When you modify a parameter value using a control, the associated plugin property is also modified. If the audio processing algorithm of the plugin depends on properties, the algorithm is also modified.

To visualize the relationship between plugin properties, parameters, and the environment in which a plugin is run, see “Implementation of Audio Plugin Parameters” on page 2-13.

`pluginParameter = audioPluginParameter(propertyName,Name,Value)` specifies `audioPluginParameter` properties using one or more `Name,Value` pair arguments.

Examples

Associate Property with Parameter

Create a basic audio plugin class definition file. Specify a property, `Gain`, and a processing function that multiplies input by `Gain`.

```

classdef myAudioPlugin < audioPlugin
    properties
        Gain = 1;
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end

```

Add a constant property, `PluginInterface`, which is specified as an `audioPluginInterface` object.

```

classdef myAudioPlugin < audioPlugin
    properties
        Gain = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface;
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end

```

Pass `audioPluginParameter` to `audioPluginInterface`. To associate the plugin property, `Gain`, to a plugin parameter, specify the first argument of `audioPluginParameter` as the property name, `'Gain'`.

```

classdef myAudioPlugin < audioPlugin
    properties
        Gain = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain'));
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end

```

```
end  
end
```

Specify Parameter Information

Create a basic plugin class definition file. Specify 'DisplayName' as 'Awesome Gain', 'Label' as '(linear)', and 'Mapping' as {'lin',0,20}.

```
classdef myAudioPlugin < audioPlugin  
    properties  
        Gain = 1;  
    end  
    properties (Constant)  
        PluginInterface = audioPluginInterface(...  
            audioPluginParameter('Gain',...  
                'DisplayName', 'Awesome Gain',...  
                'Label', '(linear)',...  
                'Mapping', {'lin',0,20}));  
    end  
    methods  
        function out = process(plugin,in)  
            out = in*plugin.gain;  
        end  
    end  
end
```

Input Arguments

propertyName — Name of audio plugin property

string

Name of the audio plugin property that you want to associate with a parameter, specified as a string. Enter the property name exactly as it is defined in the property section of your audio plugin class.

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example: 'DisplayName', 'Gain', 'Label', 'dB' specifies the display name of your parameter as 'Gain' and the display label for parameter value units as 'dB'.

'DisplayName' — Display name of parameter

associated property name (default) | string

Display name of your parameter, specified as a comma-separated pair consisting of 'DisplayName' and a string. If 'DisplayName' is not specified, the name of the associated property is used.

The display name of your parameter is used in a digital audio workstation (DAW) environment, and when using Audio Test Bench in the MATLAB environment.

'Label' — Display label for parameter value units

' ' (default) | string

Display label for parameter value units, specified as a comma-separated pair consisting of 'Label' and a string.

The display label for parameter value units is used in a digital audio workstation (DAW) environment, and when using Audio Test Bench in the MATLAB environment.

The 'Label' name-value pair is ignored for nonnumeric parameters.

'Mapping' — Mapping between property and parameter range

cell array

Mapping between property and parameter range, specified as a comma-separated pair consisting of 'Mapping' and a cell array.

Parameter range mapping specifies a mapping between a property and the associated parameter range.

The first element of the cell array is a string specifying the kind of mapping. The valid values are 'lin', 'log', 'pow', 'int' and 'enum'. The subsequent elements of the cell array depend on the kind of mapping. The valid mappings depend on the property data type.

Property Data Type	Valid Mappings	Default
double	'lin', 'log', 'pow', 'int'	{'lin', 0, 1}

Property Data Type	Valid Mappings	Default
logical	'enum'	{'enum', 'off', 'on'}
enumeration class	'enum'	enumeration name

'lin' — Specifies a linear relationship with given minimum and maximum values.

Example: {'lin', 0, 24} specifies a linear relationship with a minimum of 0 and maximum of 24.

'log' — Specifies a logarithmic relationship with given minimum and maximum values, where the control position maps to the logarithm of the property value. The minimum value must be greater than 0.

Example: {'log', 1, 22050} specifies a logarithmic relationship with a minimum of 1 and a maximum of 22050.

'pow' — Specifies a power law relationship with given exponent, minimum, and maximum values. The property value is related to the control position raised to the exponent: $(property\ value) = \min + (\max - \min) \times (parameter\ value)^{exp}$.

Example: {'pow', 2, 0, 10} specifies a power law relationship with an exponent of 2, a minimum of 0, and a maximum of 10.

'int' — Quantizes the control position and maps it to the range of consecutive integers with given minimum and maximum values.

Example: {'int', -100, 3} specifies a linear, quantized, relationship with a minimum of -100 and maximum of 3. The property value is mapped as an integer in the range -100 to 3.

'enum' (logical) — Optionally provides strings for display on the plugin dialog box.

Example: {'enum', 'unvoiced speech', 'voiced speech'} specifies the string 'unvoiced speech' if the parameter value is false and 'voiced speech' if the parameter value is true.

'enum' (enumeration class) — Optionally provides strings for the members of the enumeration class.

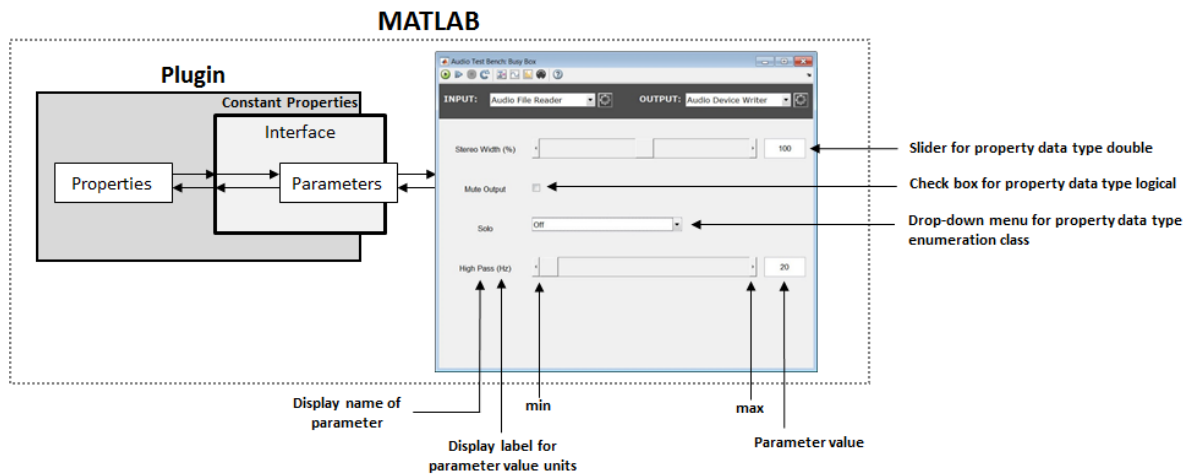
Example: {'enum', 'white noise', 'pink noise', 'brown noise'} specifies the strings 'white noise', 'pink noise', 'brown noise'.

More About

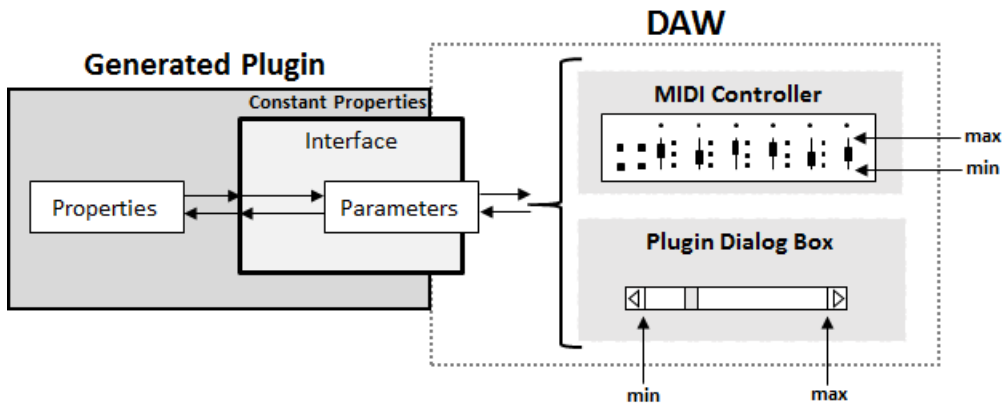
Implementation of Audio Plugin Parameters

Audio plugin parameters are visible and tunable in both the MATLAB and digital audio workstation (DAW) environments.

MATLAB Environment. Use Audio Test Bench to interact with plugin parameters in the MATLAB environment.



DAW Environment. Use `generateAudioPlugin` to deploy your audio plugin to a DAW environment. The DAW environment determines the exact layout of plugin parameters as seen by the plugin user.



See Also

`audioPluginSource` | `audioPlugin` | `audioPluginInterface` | `generateAudioPlugin`
| `validateAudioPlugin`

Introduced in R2016a

configureMIDI

Configure MIDI connections between audio plugin and MIDI controller

Syntax

```
configureMIDI(myAudioPlugin)
configureMIDI(myAudioPlugin,propertyName)
configureMIDI(myAudioPlugin,propertyName,controlNumber)
configureMIDI(myAudioPlugin,propertyName,controlNumber,'DeviceName',
deviceNameValue)
```

Description

`configureMIDI(myAudioPlugin)` opens a MIDI configuration user interface (UI). Use the UI to synchronize parameters of the plugin, `myAudioPlugin`, to MIDI controls on your default MIDI device. You can also generate MATLAB code corresponding to the MIDI configuration developed using the `configureMIDI` UI.

To set your default device, type this syntax in the command line:

```
setpref midi DefaultDevice deviceNameValue
```

`deviceNameValue` is the MIDI device name, assigned by the device manufacturer or host operating system.

`configureMIDI(myAudioPlugin,propertyName)` makes the plugin property, `propertyName`, respond to any control on the default MIDI device.

`configureMIDI(myAudioPlugin,propertyName,controlNumber)` makes the plugin property respond to the MIDI control specified by `controlNumber`.

`configureMIDI(myAudioPlugin,propertyName,controlNumber,'DeviceName',deviceNameValue)` makes the plugin property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceNameValue`.

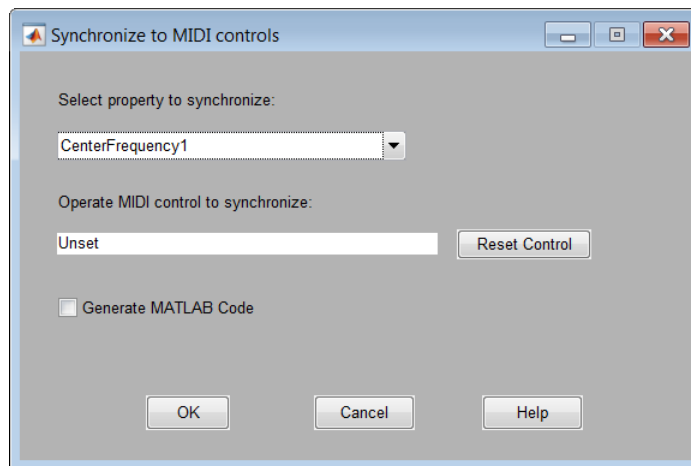
Examples

Synchronize Plugin Parameters to MIDI Controls

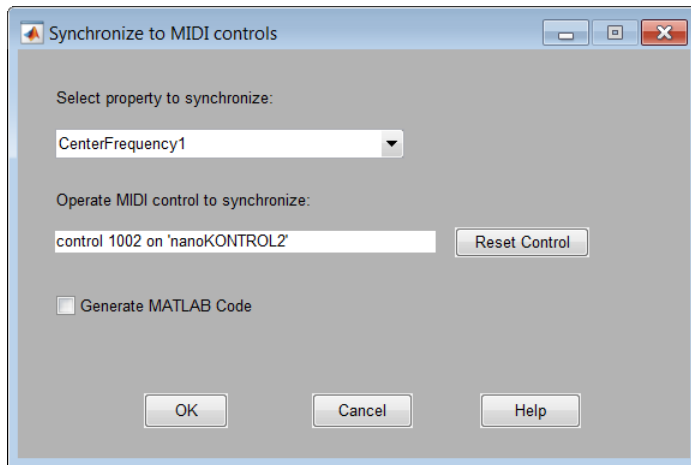
- 1 Open the MIDI configuration UI for a parametric equalizer plugin object.

```
parametricEQPlugin = audiopluginexample.ParametricEqualizer;  
configureMIDI(parametricEQPlugin);
```

- 2 In the UI, select a property to synchronize with your default MIDI device.



- 3 On your MIDI device, operate the control that you want to synchronize to the selected plugin property. The control appears in the **Operate MIDI control to synchronize** box.



- 4 Repeat steps 2 and 3 as needed to synchronize multiple properties to multiple MIDI controls.

To disconnect the property and control currently displayed on your `configureMIDI` UI, click **Reset Control** at any time.

- 5 Click **OK**.

The specified MIDI controls and properties are now synchronized.

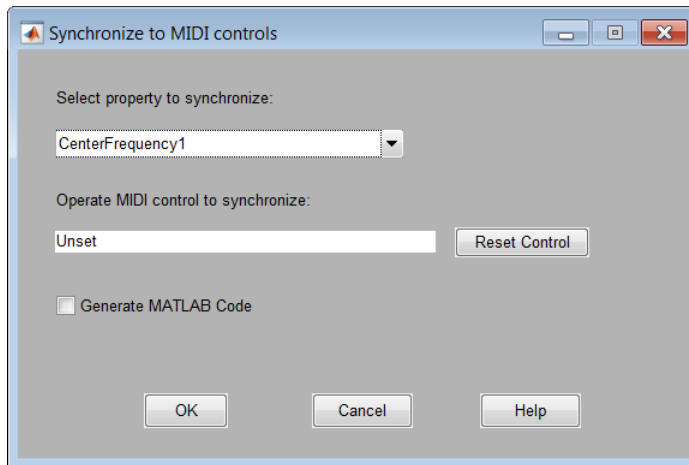
Generate MATLAB Code from `configureMIDI` UI

Generate MATLAB code corresponding to the MIDI configuration developed using the `configureMIDI` UI. You can embed the MATLAB code in your simulation so that you do not need to reopen the UI to restore your chosen MIDI connections.

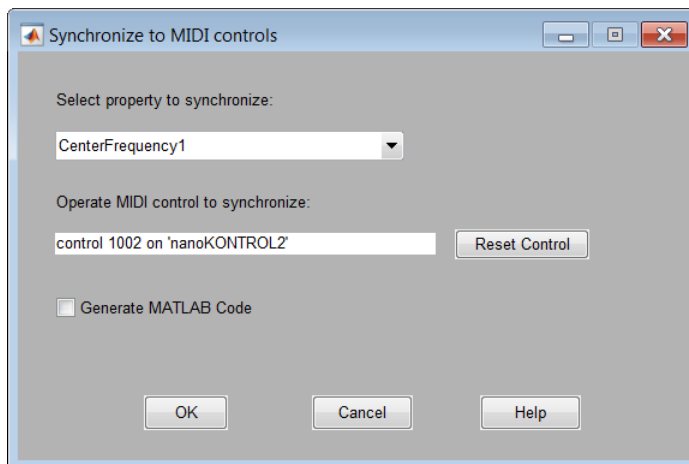
- 1 Open the MIDI configuration UI for a parametric equalizer plugin object.

```
parametricEQPlugin = audiopluginexample.ParametricEqualizer;
configureMIDI(parametricEQPlugin);
```

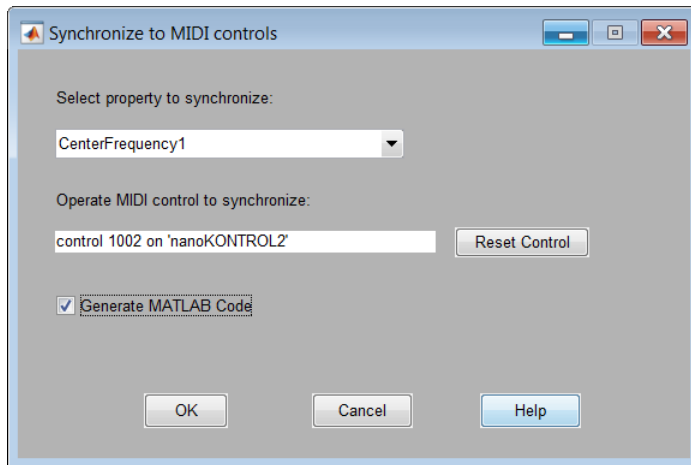
- 2 In the UI, select a property to synchronize with your default MIDI device.



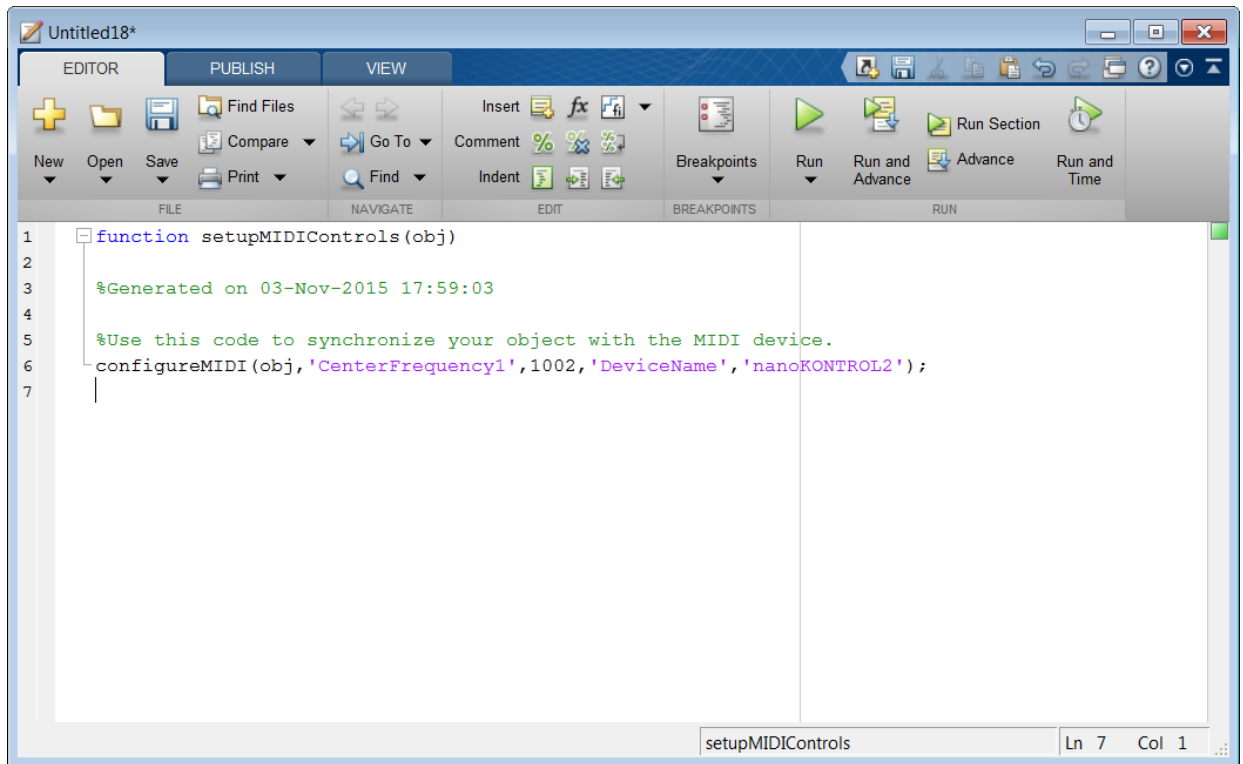
- 3 On your MIDI device, operate the control that you want to synchronize to the selected plugin property. The control appears in the **Operate MIDI control to synchronize** box.



- 4 Select the **Generate MATLAB Code** check box.



- 5 Click **OK**. The generated MATLAB code corresponds to the MIDI configuration that you developed.



Make Plugin Property Respond to Any MIDI Control

Make a plugin property respond to any control on your default MIDI device.

```
parametricEQPlugin = audiopluginexample.ParametricEqualizer;
configureMIDI(parametricEQPlugin, 'CenterFrequency1');
```

Make Plugin Property Respond to Specific MIDI Control on Default MIDI Device

Make a plugin property respond to a specific MIDI control on your default MIDI device.

Create an object of the audio plugin example
audiopluginexample.ParametricEqualizer.

```
parametricEQPlugin = audiopluginexample.ParametricEqualizer;
```

Use `midiid` to identify a MIDI control to synchronize with your property.

```
[controlNumber,device] = midiid
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
```

```
controlNumber =
```

```
    1003
```

```
device =
```

```
nanoKONTROL2
```

Use `configureMIDI` to synchronize your chosen MIDI control, specified by `controlNumber`, with a property.

```
configureMIDI(parametricEQPlugin, 'CenterFrequency1', controlNumber);
```

Make Plugin Property Respond to Specific MIDI Control on a Specific MIDI Device

Make a plugin property respond to any control on your default MIDI device.

Create an object of the audio plugin example,
`audiopluginexample.ParametricEqualizer`.

```
parametricEQPlugin = audiopluginexample.ParametricEqualizer;
```

Use `midiid` to identify a specific MIDI control on a specific MIDI device.

```
[controlNumber,device] = midiid
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
```

```
controlNumber =
```

```
    1003
```

```
device =
```

```
nanoKONTROL2
```

Use `configureMIDI` to synchronize a property with your chosen MIDI control, specified by `controlNumber`, on your chosen MIDI device, specified by `device`.

```
configureMIDI(parametricEQPlugin, 'CenterFrequency1', controlNumber, 'DeviceName', device)
```

Input Arguments

myAudioPlugin — Audio plugin

object

Audio plugin, specified as an object that inherits from the `audioPlugin` class.

propertyName — Name of audio plugin property

string

Name of the audio plugin property, specified as a string. Enter the property name exactly as it is defined in the property section of your audio plugin class.

controlNumber — MIDI device control number

integer values

MIDI device control number, specified as an integer. The value is assigned to the control by the device manufacturer. It is used for identification purposes.

deviceNameValue — MIDI device name

string

MIDI device name, assigned by the device manufacturer or host operating system, specified as a string. If you do not specify a MIDI device name, the default MIDI device is used.

Limitations

For MIDI connections established by `configureMIDI`, moving a MIDI control sends a callback to update the associated plugin property values. To synchronize your MIDI device in an audio stream loop, you might need to use the `drawnow` command for the callback to process immediately. For efficiency, use the `drawnow limitrate` syntax.

For example, to synchronize your MIDI device and audio plugin, uncomment the `drawnow limitrate` command from this code :

```
fileReader = dsp.AudioFileReader(...  
    'Filename', 'RockDrums-44p1-stereo-11secs.mp3');  
deviceWriter = audioDeviceWriter;  
compressor = dynamicRangeCompressor;  
  
configureMIDI(compressor, 'Threshold');  
  
while ~isDone(fileReader)  
    input = step(fileReader);  
    output = step(compressor, input);  
    step(deviceWriter, output);  
    % drawnow limitrate;  
end  
  
release(fileReader);  
release(deviceWriter);
```

If your audio stream loop includes visualizing data on a scope, such as `dsp.SpectrumAnalyzer`, `dsp.TimeScope`, or `dsp.ArrayPlot`, the `drawnow` command is not required.

More About

- “Musical Instrument Digital Interface (MIDI)”

See Also

`audioPlugin` | `audioPluginSource` | `disconnectMIDI` | `getMIDIConnections` | `midicallback` | `midicontrols` | `midiid` | `midiread` | `midisync`

Introduced in R2016a

designParamEQ

Design parametric equalizer

Syntax

```
[B,A] = designParamEQ(N,gain,centerFreq,bandwidth)  
[B,A] = designParamEQ(N,gain,centerFreq,bandwidth,mode)
```

Description

`[B,A] = designParamEQ(N,gain,centerFreq,bandwidth)` designs an Nth-order parametric equalizer with specified gain, center frequency, and bandwidth. **B** and **A** are matrices of numerator and denominator coefficients, with columns corresponding to cascaded second-order section (SOS) filters.

`[B,A] = designParamEQ(N,gain,centerFreq,bandwidth,mode)` specifies whether the parametric equalizer is implemented with second-order sections or fourth-order sections (FOS).

Examples

Design Two-Band Parametric Equalizer

Specify the filter order, peak gain in dB, normalized center frequencies, and normalized bandwidth of the bands of your parametric equalizer.

```
N = [2,4];  
gain = [6,-4];  
centerFreq = [0.25,0.75];  
bandwidth = [0.12,0.10];
```

Generate the filter coefficients using the specified parameters.

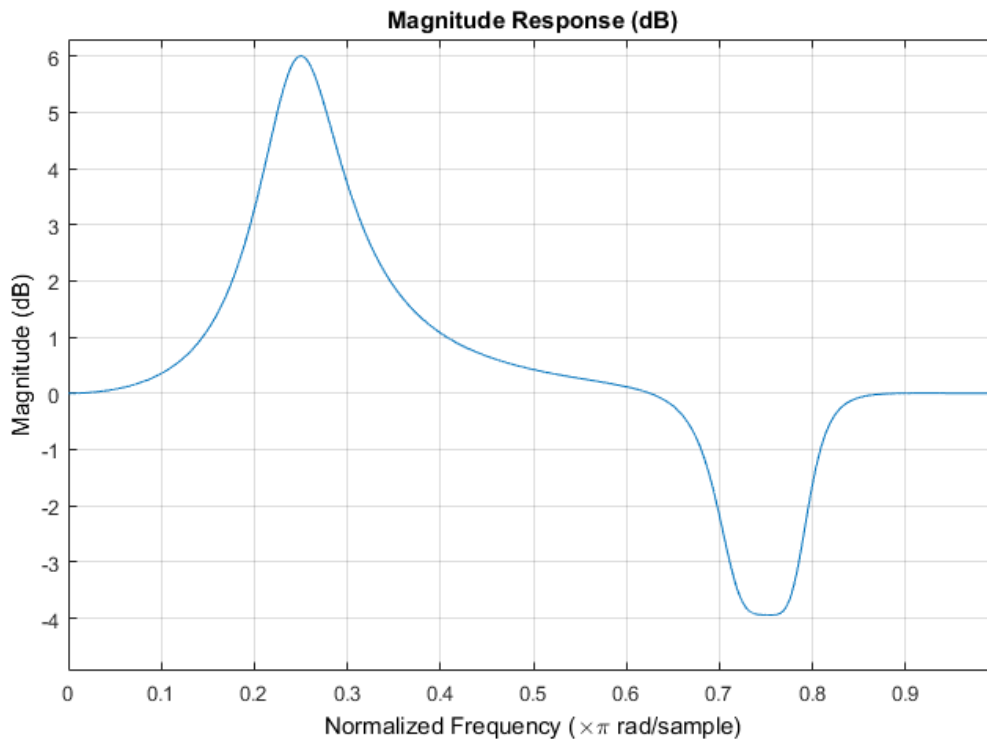
```
[B,A] = designParamEQ(N,gain,centerFreq,bandwidth);
```

Create a filter matrix compatible with `fvtool`.

```
SOS = [B', [ones(sum(N)/2,1), A' ]];
```

Visualize your filter design.

```
fvtool(SOS)
```



Filter Audio Using SOS Parametric Equalizer

Design a second-order sections (SOS) parametric equalizer using `designParamEQ`, and filter an audio stream.

Construct audio file reader and audio device writer System objects. Use the sample rate of the reader as the sample rate of the writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
frameSize = 256;

fileReader = dsp.AudioFileReader(...
    'RockGuitar-16-44p1-stereo-72secs.wav',...
    'SamplesPerFrame',frameSize);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);

setup(fileReader);
setup(deviceWriter,ones(frameSize,2));
```

Play the audio signal through your device.

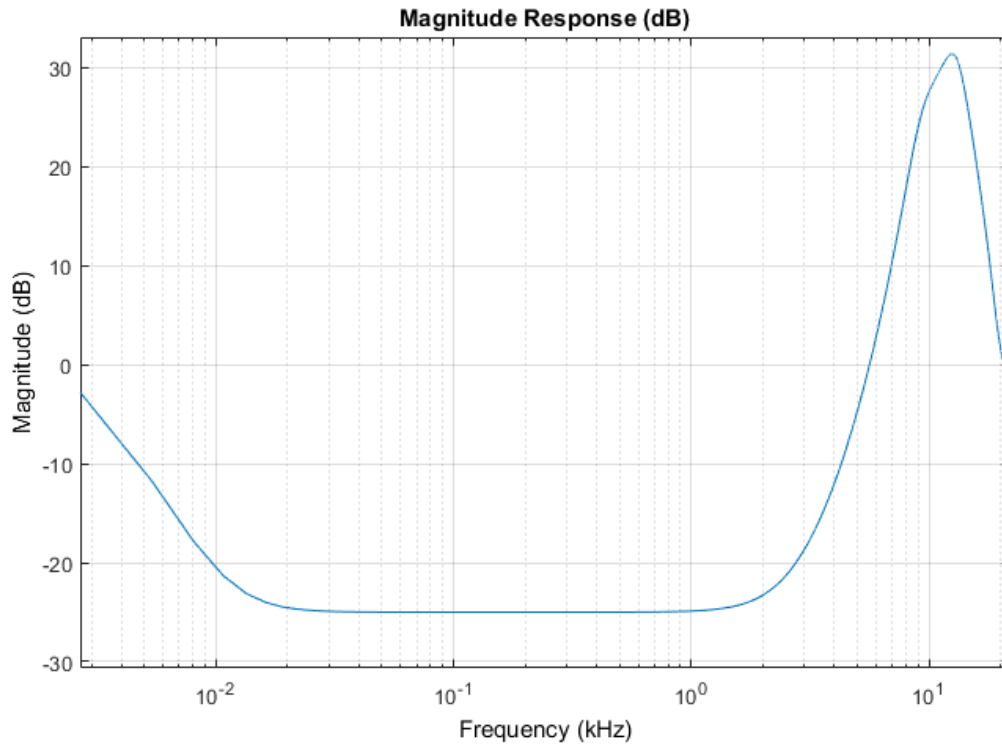
```
count = 0;
while count < 2500
    audio = step(fileReader);
    play(deviceWriter,audio);
    count = count+1;
end
reset(fileReader);
```

Design a SOS parametric equalizer.

```
N = [4,4];
gain = [-25,35];
centerFreq = [0.01,0.5];
bandwidth = [0.35,0.5];
[B,A] = designParamEQ(N,gain,centerFreq,bandwidth);
```

Visualize your filter design.

```
SOS = [B',[ones(4,1),A']];
fvtool(SOS,...
    'Fs',fileReader.SampleRate,...
    'FrequencyScale','Log');
```

Construct a biquad filter System object.

```
myFilter = dsp.BiquadFilter(...
    'SOSMatrixSource', 'Input port',...
    'ScaleValuesInputPort', false);
```

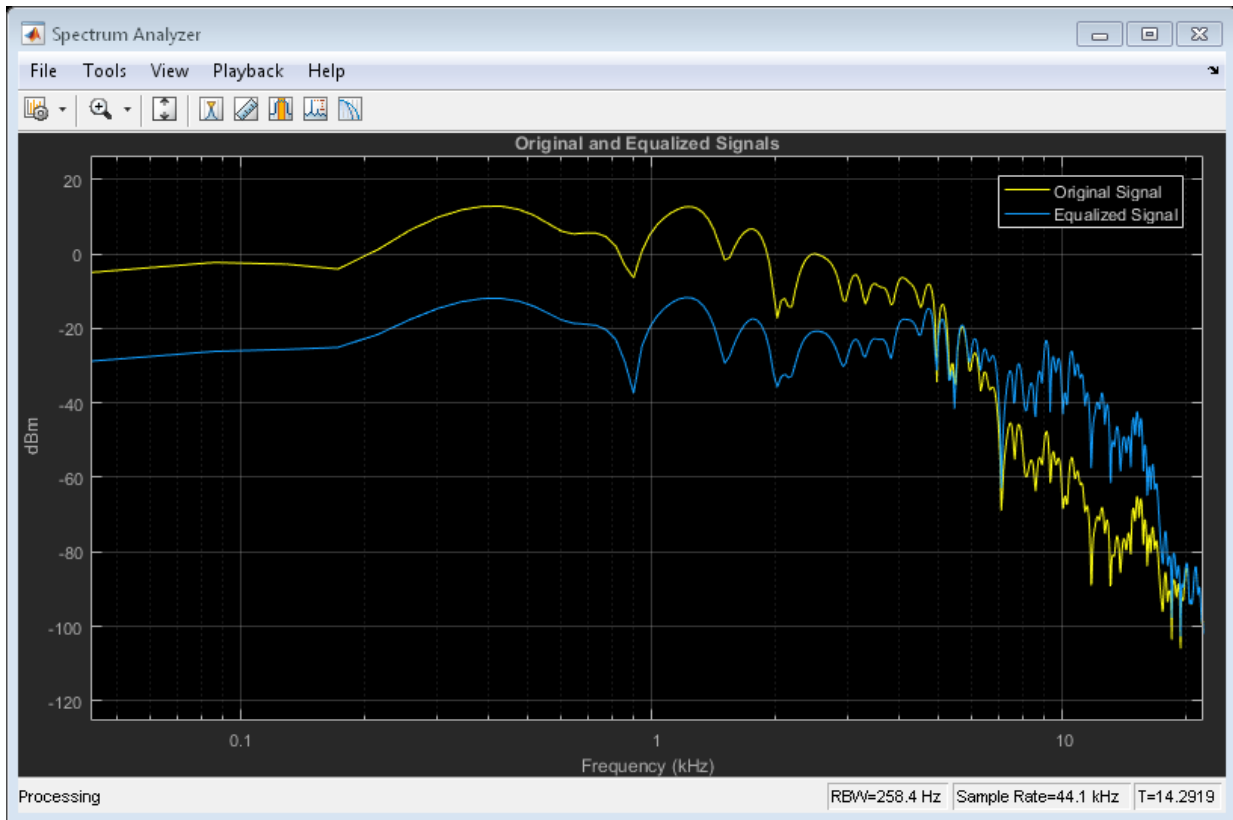
Construct a spectrum analyzer to visualize the original audio signal and the audio signal passed through your parametric equalizer.

```
scope = dsp.SpectrumAnalyzer(...
    'SampleRate', fileReader.SampleRate, ...
    'PlotAsTwoSidedSpectrum', false, ...
    'FrequencyScale', 'Log', ...
    'FrequencyResolutionMethod', 'WindowLength', ...
    'WindowLength', frameSize, ...
    'Title', 'Original and Equalized Signals', ...
```

```
'ShowLegend',true,...  
'ChannelNames',{'Original Signal','Equalized Signal'});
```

Play the filtered audio signal and visualize the original and filtered spectrums.

```
setup(scope,ones(frameSize,2));  
count = 0;  
while count < 2500  
    originalSignal = step(fileReader);  
    equalizedSignal = step(myFilter,originalSignal,B,A);  
    step(scope,[originalSignal(:,1),equalizedSignal(:,1)]);  
    play(deviceWriter,equalizedSignal);  
    count = count+1;  
end
```



Filter Audio Using FOS Parametric Equalizer

Design a fourth-order sections (FOS) parametric equalizer using `designParamEQ`, and filter an audio stream.

Construct audio file reader and audio device writer `System` objects. Use the sample rate of the reader as the sample rate of the writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
frameSize = 256;
```

```
fileReader = dsp.AudioFileReader(...
    'RockGuitar-16-44p1-stereo-72secs.wav',...
    'SamplesPerFrame',frameSize);
```

```
deviceWriter = audioDeviceWriter(...  
    'SampleRate',fileReader.SampleRate);
```

```
setup(fileReader);  
setup(deviceWriter,ones(frameSize,2));
```

Play the audio signal through your device.

```
count = 0;  
while count < 2500  
    audio = step(fileReader);  
    play(deviceWriter,audio);  
    count = count+1;  
end  
reset(fileReader);
```

Design FOS parametric equalizer coefficients.

```
N = [2,4];  
gain = [5,10];  
centerFreq = [0.025,0.65];  
bandwidth = [0.025,0.35];  
mode = 'fos';
```

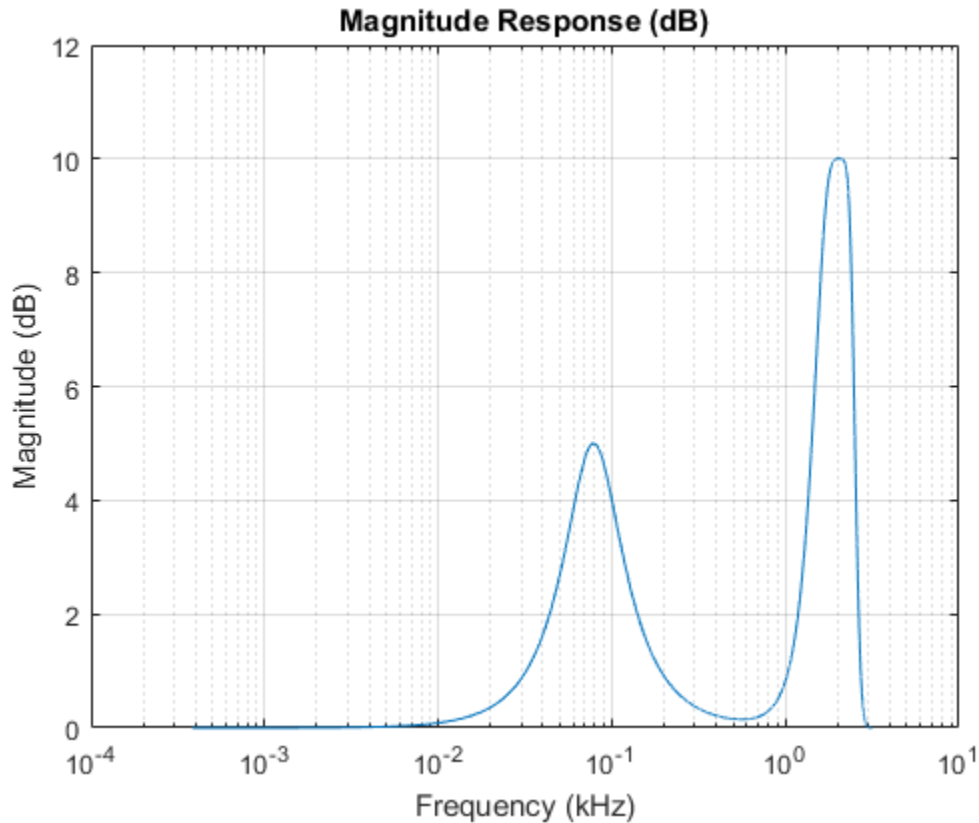
```
[B,A] = designParamEQ(N,gain,centerFreq,bandwidth,mode);
```

Construct FOS IIR filters.

```
section1 = dsp.IIRFilter('Numerator',B(:,1)', 'Denominator',[1,A(:,1)']);  
section2 = dsp.IIRFilter('Numerator',B(:,2)', 'Denominator',[1,A(:,2)']);
```

Visualize the frequency response of your parametric equalizer.

```
[H1,w] = freqz(section1);  
H2 = freqz(section2);  
  
H = 20.*log10(abs(H1.*H2));  
  
semilogx(w,H);  
title('Magnitude Response (dB)')  
xlabel('Frequency (kHz)')  
ylabel('Magnitude (dB)')  
grid on
```

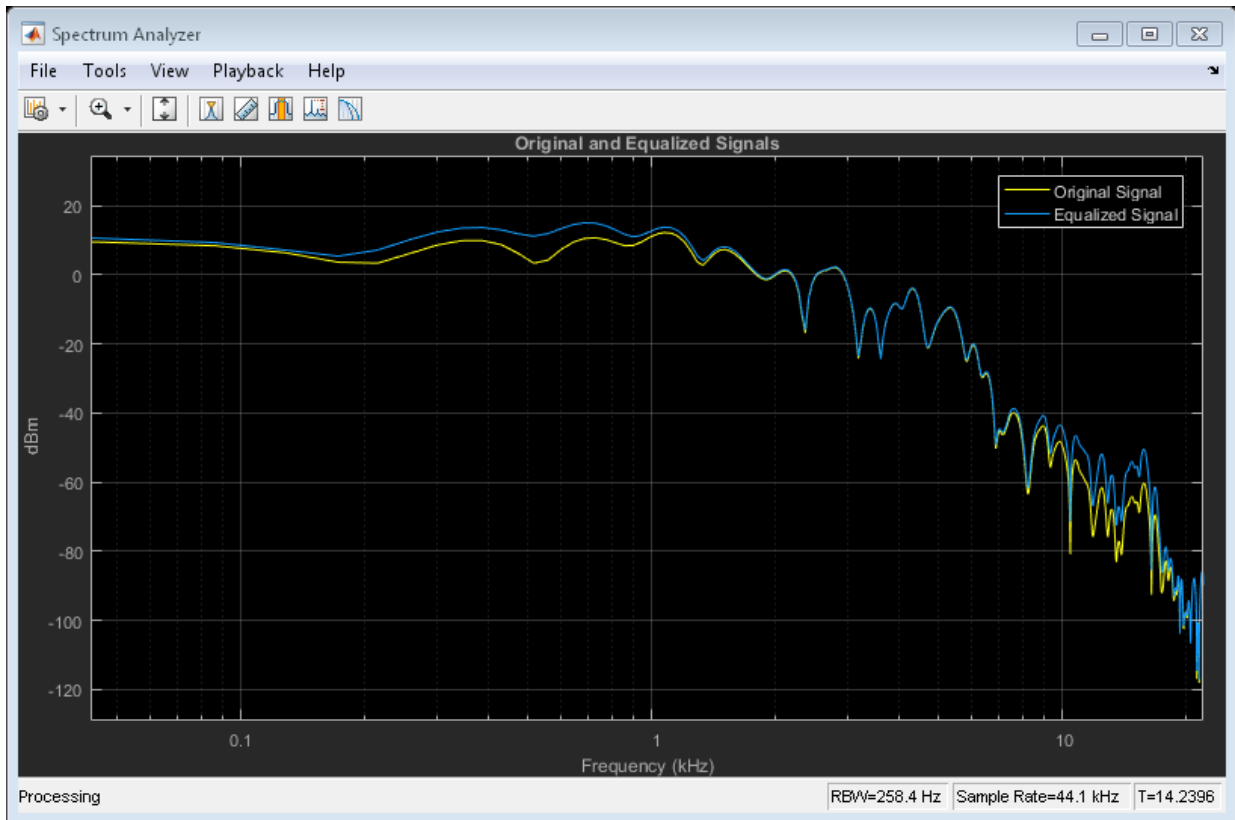


Construct a spectrum analyzer to visualize the original audio signal and the audio signal passed through your parametric equalizer.

```
scope = dsp.SpectrumAnalyzer(...
    'SampleRate',fileReader.SampleRate,...
    'PlotAsTwoSidedSpectrum',false,...
    'FrequencyScale','Log',...
    'FrequencyResolutionMethod','WindowLength',...
    'WindowLength',frameSize,...
    'Title','Original and Equalized Signals',...
    'ShowLegend',true,...
    'ChannelNames',{'Original Signal','Equalized Signal'});
```

Play the filtered audio signal, and visualize the original and filtered spectrums.

```
setup(scope,ones(frameSize,2));  
count = 0;  
while count < 2500  
    originalSignal = step(fileReader);  
    x = step(section1,originalSignal);  
    y = step(section2,x);  
    step(scope,[originalSignal(:,1),y(:,1)]);  
    play(deviceWriter,y);  
    count = count + 1;  
end
```



Input Arguments

N — Filter order

scalar | row vector

Filter order, specified as a scalar or row vector the same length as `centerFreq`. Elements of the vector must be even integers.

gain — Peak gain (dB)

scalar | row vector

Peak gain in dB, specified as a scalar or row vector the same length as `centerFreq`. Elements of the vector must be real-valued.

centerFreq — Normalized center frequency of equalizer bands

scalar | row vector

Normalized center frequency of equalizer bands, specified as a scalar or row vector of real values in the range 0 to 1, where 1 corresponds to the Nyquist frequency (π rad/sample). If `centerFreq` is specified as a row vector, separate equalizers are designed for each element of `centerFreq`.

bandwidth — Normalized bandwidth

scalar | row vector

Normalized bandwidth, specified as a scalar or row vector the same length as `centerFreq`. Elements of the vector are specified as real values in the range 0 to 1, where 1 corresponds to the Nyquist frequency (π rad/sample).

Normalized bandwidth is measured at gain/2 dB. If gain is set to `-Inf` (notch filter), normalized bandwidth is measured at the 3 dB attenuation point: $10 \times \log_{10}(0.5)$.

To convert octave bandwidth to normalized bandwidth, calculate the associated Q -factor as

$$Q = \frac{\sqrt{2^{(\text{octave bandwidth})}}}{2^{(\text{octave bandwidth})} - 1}.$$

Then convert to bandwidth

$$\text{bandwidth} = \frac{\text{centerFreq}}{Q}.$$

mode — Design mode

'sos' (default) | 'fos'

Design mode, specified as 'sos' or 'fos'.

- 'sos' — Implements your equalizer as cascaded second-order filters.
- 'fos' — Implements your equalizer as cascaded fourth-order filters. Because fourth-order sections do not require the computation of roots, they are generally more computationally efficient.

Output Arguments

B — Numerator filter coefficients

matrix

Numerator filter coefficients, returned as a matrix. Each column of **B** corresponds to the numerator coefficients of a different second-order or fourth-order section of your cascaded equalizer.

A — Denominator filter coefficients

matrix

Denominator filter coefficients, returned as a matrix. Each column of **A** corresponds to the denominator coefficients of a different second-order or fourth-order section of your cascaded equalizer.

A does not include the leading unity coefficient for each section.

See Also

[multibandParametricEQ](#) | [dsp.biquadFilter](#) | [designShelvingEQ](#) | [designVarSlopeFilter](#)

Introduced in R2016a

designShelvingEQ

Design shelving equalizer

Syntax

```
[B,A] = designShelvingEQ(gain,slope,Fc)  
[B,A] = designShelvingEQ(gain,slope,Fc,type)
```

Description

`[B,A] = designShelvingEQ(gain,slope,Fc)` designs a low-shelf equalizer with the specified gain, slope, and cutoff frequency, `Fc`. The equalizer is returned as cascaded second-order section (SOS) IIR filters.

`[B,A] = designShelvingEQ(gain,slope,Fc,type)` specifies the design type as a low-shelving or high-shelving equalizer.

Examples

Design Low-Shelf Equalizer

Design three second-order IIR low-shelf equalizers using `designShelvingEQ`. The three shelving equalizers use three separate slope specifications.

Specify sampling frequency, peak gain, slope coefficient, and normalized cutoff frequency for three shelving equalizers. The sampling frequency is in Hz. The peak gain is in dB.

```
Fs = 44.1e3;
```

```
gain = 5;
```

```
slope1 = 0.5;
```

```
slope2 = 0.75;
```

```
slope3 = 1;
```

```
Fc = 1000/(Fs/2);
```

Design the filter coefficients using the specified parameters.

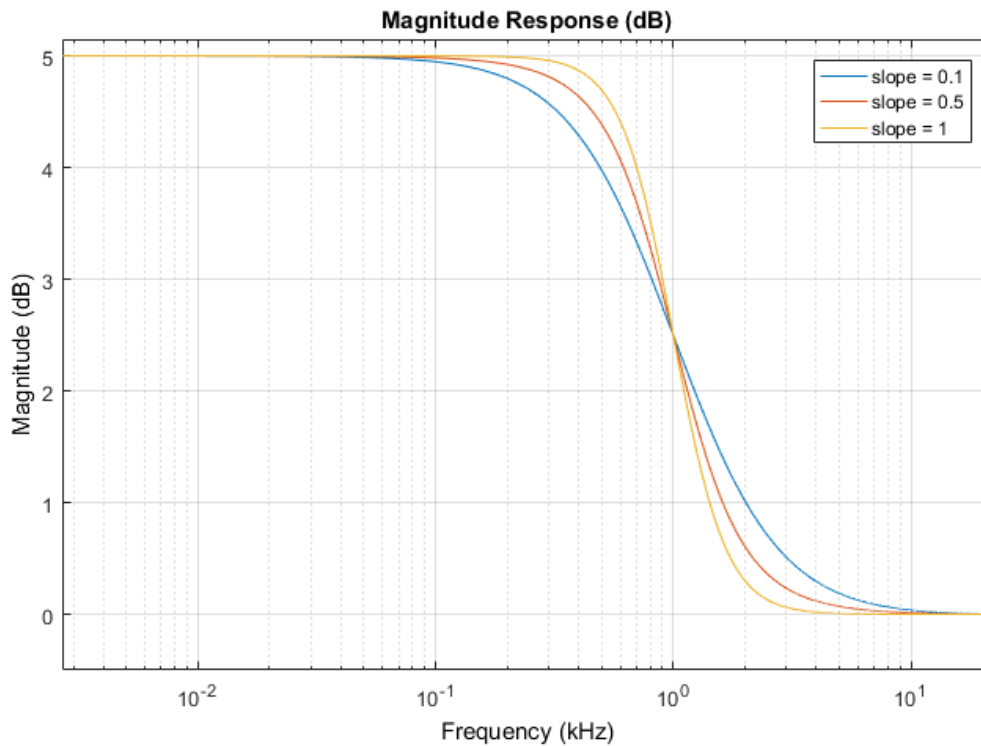
```
[B1,A1] = designShelvingEQ(gain,slope1,Fc);  
[B2,A2] = designShelvingEQ(gain,slope2,Fc);  
[B3,A3] = designShelvingEQ(gain,slope3,Fc);
```

Create filter matrices compatible with `fvtool`.

```
SOS1 = [B1',[1,A1']];  
SOS2 = [B2',[1,A2']];  
SOS3 = [B3',[1,A3']];
```

Visualize your filter design.

```
fvtool(...  
    dsp.BiquadFilter('SOSMatrix',SOS1),...  
    dsp.BiquadFilter('SOSMatrix',SOS2),...  
    dsp.BiquadFilter('SOSMatrix',SOS3),...  
    'Fs',Fs,...  
    'FrequencyScale','Log');  
  
legend('slope = 0.1',...  
    'slope = 0.5',...  
    'slope = 1');
```



Filter Audio Using Low-Shelf Equalizer

Design a low-shelf equalizer, and then use it to filter an audio signal.

Construct audio file reader and audio device writer objects. Use the sample rate of the reader as the sample rate of the writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
frameSize = 256;

fileReader = dsp.AudioFileReader(...
    'RockGuitar-16-44p1-stereo-72secs.wav',...
    'SamplesPerFrame',frameSize);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
```

```
setup(fileReader);  
setup(deviceWriter,ones(frameSize,2));
```

Play the audio signal through your device.

```
count = 0;  
while count < 2500  
    audio = step(fileReader);  
    play(deviceWriter,audio);  
    count = count+1;  
end  
reset(fileReader)
```

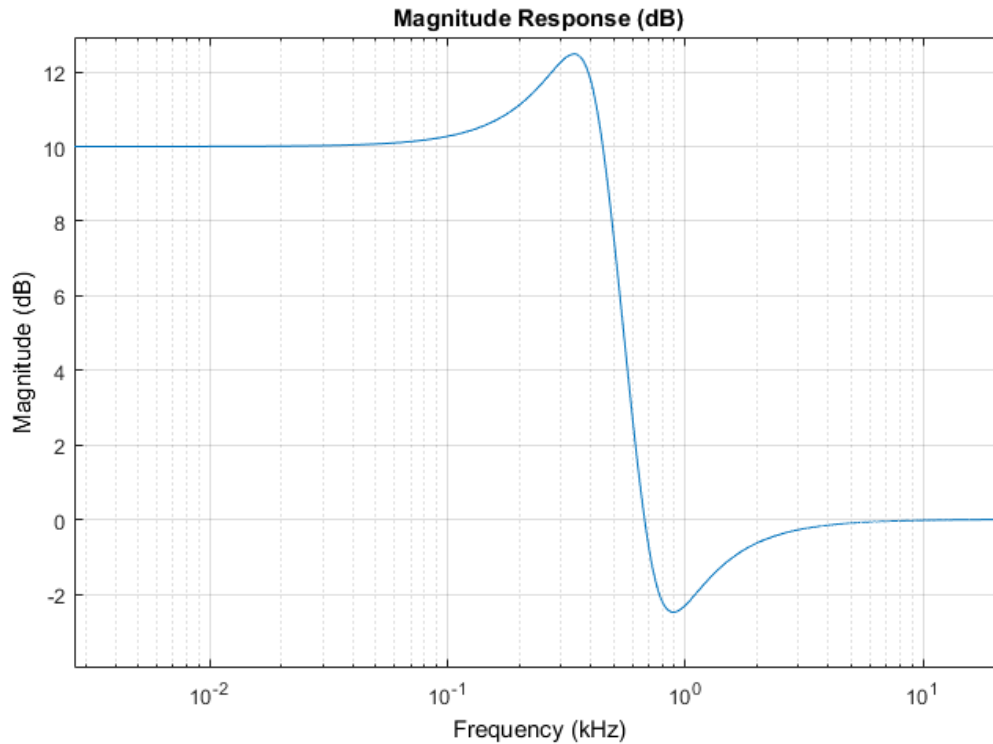
Design a second-order sections (SOS) low-shelf equalizer.

```
gain = 10;  
slope = 3;  
Fc = 0.025;
```

```
[B,A] = designShelvingEQ(gain,slope,Fc);
```

Visualize your equalizer design.

```
SOS = [B',[1,A']];  
fvtool(dsp.BiquadFilter('SOSMatrix',SOS),...  
    'Fs',fileReader.SampleRate,...  
    'FrequencyScale','Log');
```



Construct a biquad filter object.

```
myFilter = dsp.BiquadFilter(...  
    'SOSMatrixSource', 'Input port',...  
    'ScaleValuesInputPort', false);
```

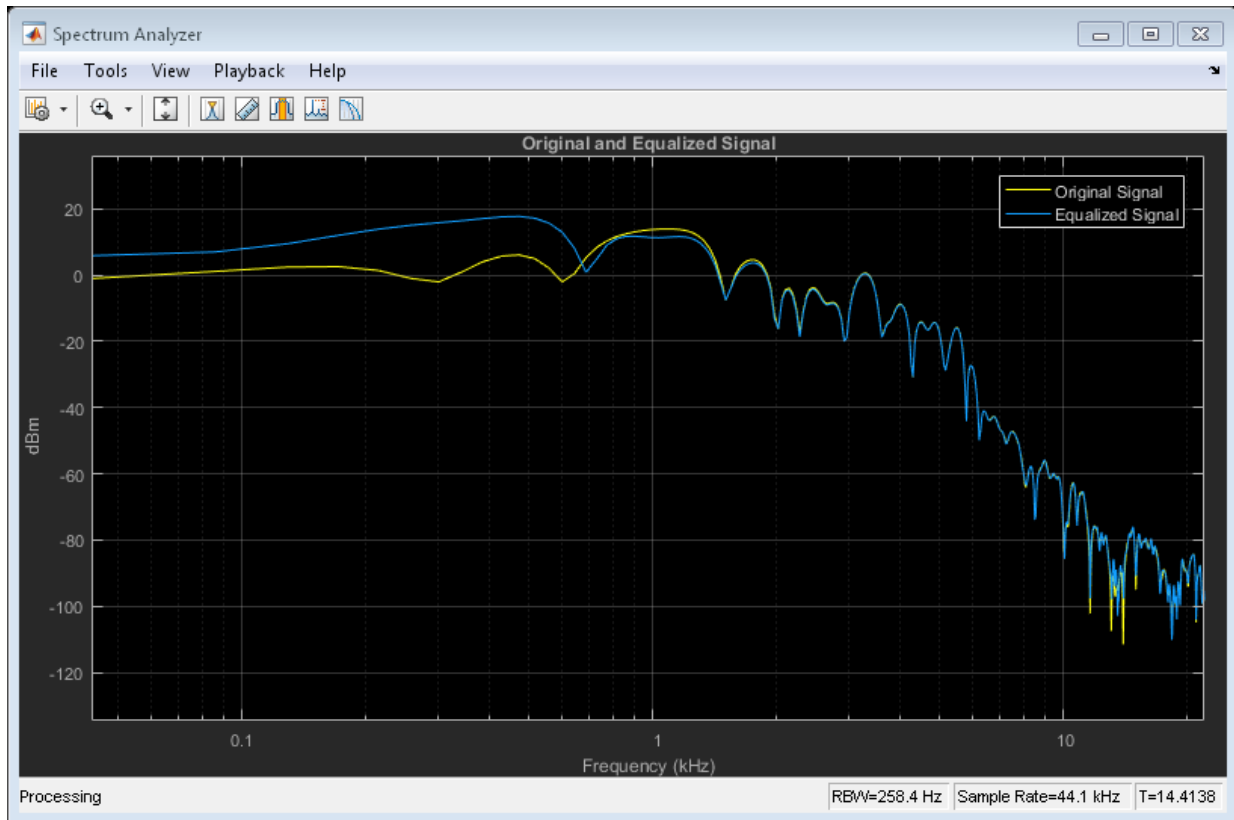
Construct a spectrum analyzer object to visualize the original audio signal and the audio signal passed through your low-shelf equalizer.

```
scope = dsp.SpectrumAnalyzer(...  
    'SampleRate', fileReader.SampleRate,...  
    'PlotAsTwoSidedSpectrum', false,...  
    'FrequencyScale', 'Log',...  
    'FrequencyResolutionMethod', 'WindowLength',...  
    'WindowLength', frameSize,...  
    'Title', 'Original and Equalized Signal',...)
```

```
'ShowLegend',true,...  
'ChannelNames',{'Original Signal','Equalized Signal'});
```

Play the equalized audio signal and visualize the original and equalized spectrums.

```
setup(scope,ones(frameSize,2));  
count = 0;  
while count < 2500  
    originalSignal = step(fileReader);  
    equalizedSignal = step(myFilter,originalSignal,B,A);  
    step(scope,[originalSignal(:,1),equalizedSignal(:,1)]);  
    play(deviceWriter,equalizedSignal);  
    count = count+1;  
end
```



Design High-Shelf Equalizer

Design three second-order IIR high shelf equalizers using `designShelvingEQ`. The three shelving equalizers use three separate gain specifications.

Specify sampling frequency, peak gain, slope coefficient, and normalized cutoff frequency for three shelving equalizers. The sampling frequency is in Hz. The peak gain is in dB

```
Fs = 44.1e3;
```

```
gain1 = -6;
```

```
gain2 = 6;
```

```
gain3 = 12;
```



```
slope = 0.8;
```

```
Fc = 18000/(Fs/2);
```

Design the filter coefficients using the specified parameters.

```
[B1,A1] = designShelvingEQ(gain1,slope,Fc,'hi');
```

```
[B2,A2] = designShelvingEQ(gain2,slope,Fc,'hi');
```

```
[B3,A3] = designShelvingEQ(gain3,slope,Fc,'hi');
```

Create filter matrices compatible with fvtool.

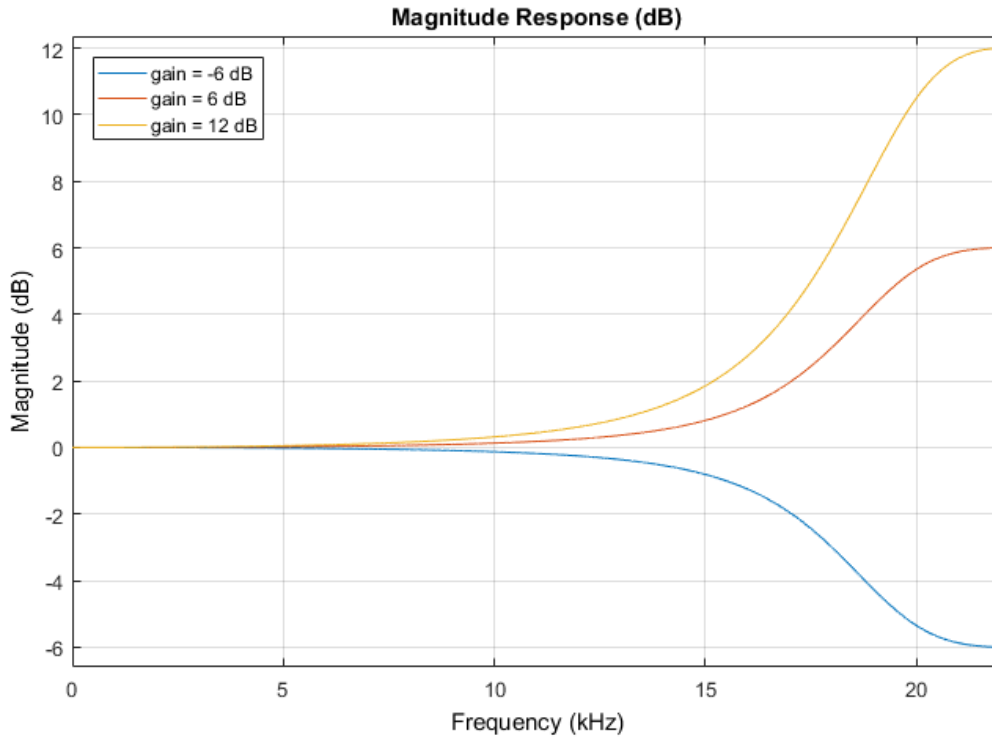
```
SOS1 = [B1',[1,A1']];
```

```
SOS2 = [B2',[1,A2']];
```

```
SOS3 = [B3',[1,A3']];
```

Visualize your filter design.

```
fvtool(dsp.BiquadFilter('SOSMatrix',SOS1),...  
       dsp.BiquadFilter('SOSMatrix',SOS2),...  
       dsp.BiquadFilter('SOSMatrix',SOS3),...  
       'Fs',Fs);  
legend('gain = -6 dB',...  
       'gain = 6 dB',...  
       'gain = 12 dB',...  
       'Location','NorthWest')
```



Input Arguments

gain — Peak gain (dB)

real scalar in the range -12 to 12

Peak gain in dB, specified as a real scalar in the range -12 to 12.

slope — Slope coefficient

real scalar in the range 0 to 5

Slope coefficient, specified as a real scalar in the range 0 to 5.

Fc — Normalized cutoff frequency

real scalar in the range 0 to 1

Normalized cutoff frequency, specified as a real scalar in the range 0 to 1, where 1 corresponds to the Nyquist frequency (π rad/sample).

Normalized cutoff frequency is implemented as half the shelving filter gain, or $\text{gain}/2$ dB.

type — Filter type

'lo' (default) | 'hi'

Filter type, specified as 'lo' or 'hi'.

- 'lo' — Low shelving equalizer
- 'hi' — High shelving equalizer

Output Arguments

B — Numerator filter coefficients

three-element column vector

Numerator filter coefficients of the designed second-order IIR filter, returned as a three-element column vector.

A — Denominator filter coefficients

two-element column vector.

Denominator filter coefficients of the designed second-order IIR filter, returned as a two-element column vector. A does not include the leading unity coefficient.

See Also

multibandParametricEQ | designParamEQ | designVarSlopeFilter

Introduced in R2016a

designVarSlopeFilter

Design variable slope lowpass or highpass IIR filter

Syntax

```
[B,A] = designVarSlopeFilter(slope,Fc)
[B,A] = designVarSlopeFilter(slope,Fc,type)
```

Description

`[B,A] = designVarSlopeFilter(slope,Fc)` designs a lowpass filter with the specified slope and cutoff frequency. **B** and **A** are matrices of numerator and denominator coefficients, with columns corresponding to cascaded second-order sections (SOS).

`[B,A] = designVarSlopeFilter(slope,Fc,type)` specifies the design type as a lowpass or highpass filter.

Examples

Design Lowpass IIR Filter

Design two second-order section (SOS) lowpass IIR filters using `designVarSlopeFilter`.

Specify the sampling frequency, slope, and normalized cutoff frequency for two lowpass IIR filters. The sampling frequency is in Hz. The slope is in dB/octave.

```
Fs = 48e3;
```

```
slope = 18;
```

```
Fc1 = 10000/(Fs/2);
```

```
Fc2 = 16000/(Fs/2);
```

Design the filter coefficients using the specified parameters.

```
[B1,A1] = designVarSlopeFilter(slope,Fc1);
[B2,A2] = designVarSlopeFilter(slope,Fc2);
```

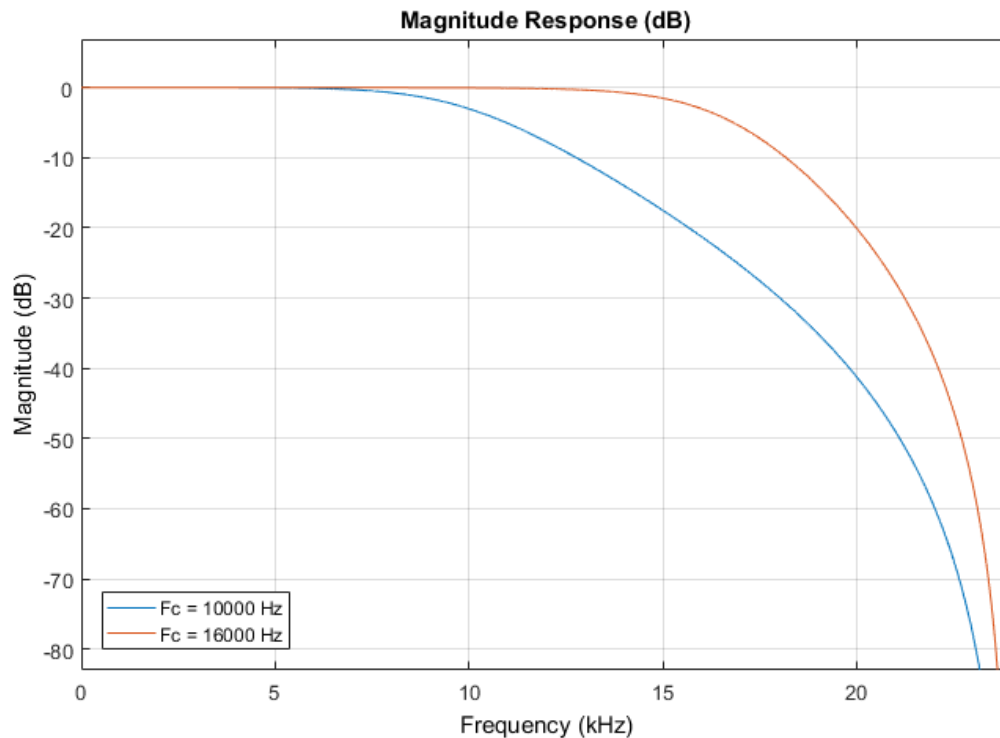
Create filter matrices compatible with `fvtool`.

```
SOS1 = [B1', [ones(4,1), A1' ]];
SOS2 = [B2', [ones(4,1), A2' ]];
```

Visualize your filter design.

```
fvtool(SOS1, SOS2, 'Fs', Fs);
```

```
legend('Fc = 10000 Hz', ...
       'Fc = 16000 Hz', ...
       'Location', 'SouthWest');
```



Process Audio Using Lowpass Filter

Design a second-order section (SOS) lowpass IIR filter using `designVarSlopeFilter`. Use your lowpass filter to process an audio signal.

Construct audio file reader and audio device writer System objects. Use the sample rate of the reader as the sample rate of the writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
frameSize = 256;

fileReader = dsp.AudioFileReader(...
    'RockGuitar-16-44p1-stereo-72secs.wav',...
    'SamplesPerFrame',frameSize);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);

setup(fileReader);
setup(deviceWriter,ones(frameSize,2));
```

Play the audio signal through your device.

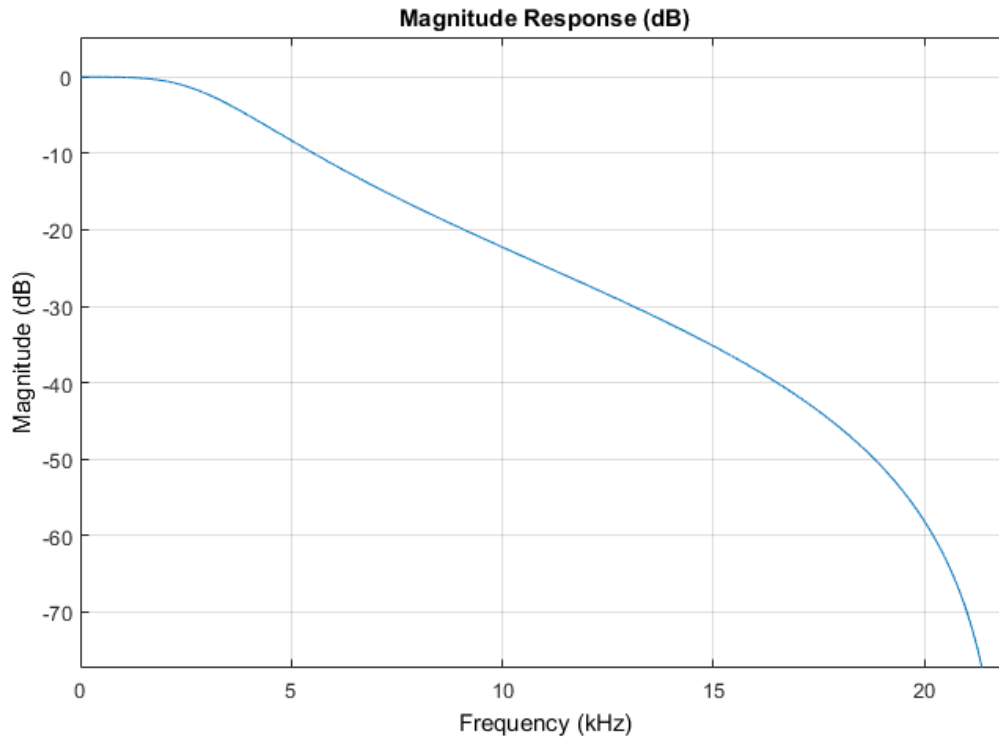
```
count = 0;
while count < 2500
    audio = step(fileReader);
    play(deviceWriter,audio);
    count = count+1;
end
reset(fileReader);
```

Design a lowpass filter with a 12 dB/octave slope and a 0.15 normalized frequency cutoff.

```
[B,A] = designVarSlopeFilter(12,0.15);
```

Visualize your filter design.

```
SOS = [B',[ones(4,1),A' ]];
fvtool(SOS,...
    'Fs',fileReader.SampleRate);
```



Construct a biquad filter System object.

```
myFilter = dsp.BiquadFilter(...
    'SOSMatrixSource', 'Input port',...
    'ScaleValuesInputPort', false);
```

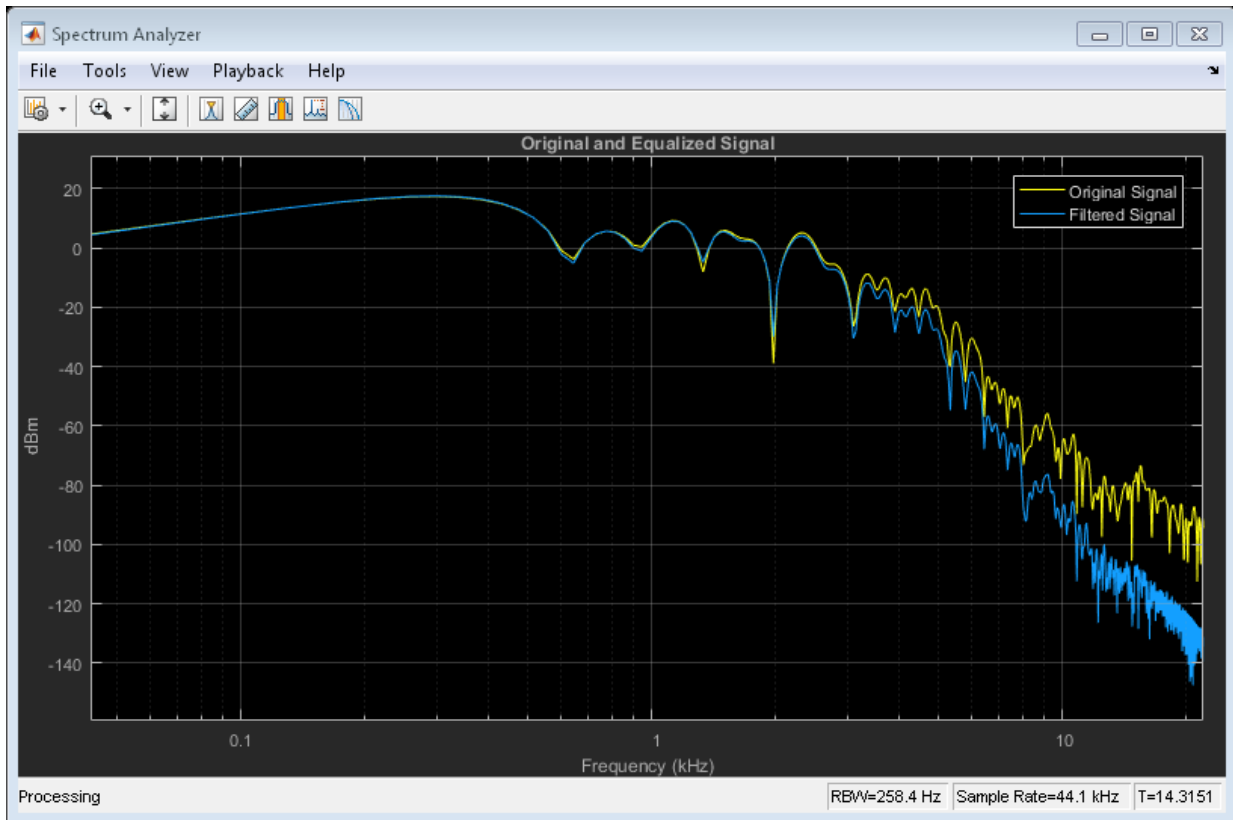
Construct a spectrum analyzer System object to visualize the original audio signal and the audio signal passed through your lowpass filter.

```
scope = dsp.SpectrumAnalyzer(...
    'SampleRate', fileReader.SampleRate, ...
    'PlotAsTwoSidedSpectrum', false, ...
    'FrequencyScale', 'Log', ...
    'FrequencyResolutionMethod', 'WindowLength', ...
    'WindowLength', frameSize, ...
    'Title', 'Original and Equalized Signal', ...
```

```
'ShowLegend',true,...  
'ChannelNames',{'Original Signal','Filtered Signal'});
```

Play the filtered audio signal and visualize the original and filtered spectrums.

```
setup(scope,ones(frameSize,2));  
count = 0;  
while count < 2500  
    originalSignal = step(fileReader);  
    filteredSignal = step(myFilter,originalSignal,B,A);  
    step(scope,[originalSignal(:,1),filteredSignal(:,1)]);  
    play(deviceWriter,filteredSignal);  
    count = count+1;  
end
```

Design Highpass IIR Filter

Design two second-order section (SOS) highpass IIR filters using `designVarSlopeFilter`.

Specify the sampling frequency in Hz, the slope in dB/octave, and the normalized cutoff frequency.

```
Fs = 48e3;
slope1 = 18;
slope2 = 36;
Fc = 4000/(Fs/2);
```

Design the filter coefficients using the specified parameters.

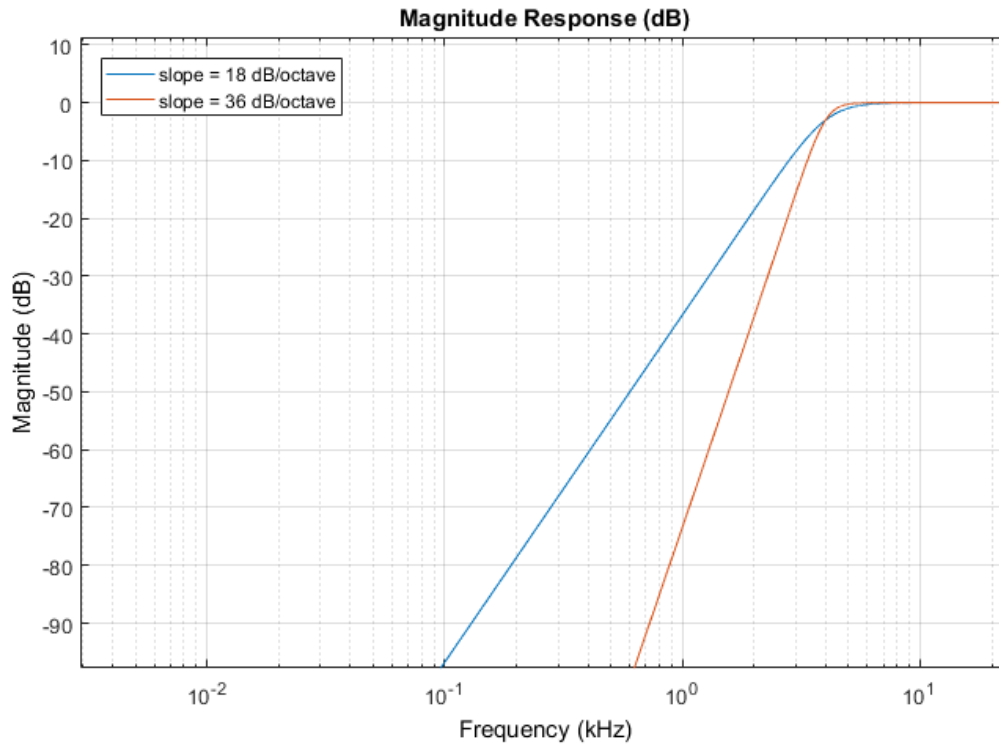
```
[B1,A1] = designVarSlopeFilter(slope1,Fc,'hi');  
[B2,A2] = designVarSlopeFilter(slope2,Fc,'hi');
```

Create filter matrices compatible with fvtool.

```
SOS1 = [B1',[ones(4,1),A1']];  
SOS2 = [B2',[ones(4,1),A2']];
```

Visualize your filter design.

```
fvtool(SOS1,SOS2,...  
    'Fs',Fs,...  
    'FrequencyScale','Log');  
legend('slope = 18 dB/octave',...  
    'slope = 36 dB/octave',...  
    'Location','NorthWest')
```



Input Arguments

slope — Filter slope (dB/octave)

real scalar in the range [0:6:48]

Filter slope in dB/octave, specified as a real scalar in the range [0:6:48]. Values that are not multiples of 6 are rounded.

Fc — Normalized cutoff frequency

real scalar in the range 0 to 1

Normalized cutoff frequency, specified as a real scalar in the range 0 to 1, where 1 corresponds to the Nyquist frequency (π rad/sample).

type — Filter type

'lo' (default) | 'hi'

Filter type, specified as 'lo' or 'hi'.

- 'lo' — Lowpass filter
- 'hi' — Highpass filter

Output Arguments

B — Numerator filter coefficients

3-by-4 matrix

Numerator filter coefficients, returned as a 3-by-4 matrix. Each column of **B** corresponds to the numerator coefficients of a different second-order section of your cascaded IIR filter.

A — Denominator filter coefficients

2-by-4 matrix

Denominator filter coefficients, returned as a 2-by-4 matrix. Each column of **A** corresponds to the denominator coefficients of a different second-order section of your cascaded IIR filter.

A does not include the leading unity coefficient for each section.

See Also

multibandParametricEQ | designParameEQ | designShelvingEQ

Introduced in R2016a

disconnectMIDI

Disconnect MIDI controls from audio plugin

Syntax

```
disconnectMIDI(myAudioPlugin)
```

Description

`disconnectMIDI(myAudioPlugin)` disconnects MIDI controls from your audio plugin object, `myAudioPlugin`. Only those MIDI connections established using `configureMIDI` are disconnected.

Examples

Disconnect MIDI Controls from Audio Plugin

Create an object of the audio plugin example `audiopluginexample.Echo`.

```
echoPlugin = audiopluginexample.Echo;
```

Get the MIDI connections of `echoPlugin` and verify that it has no MIDI connections.

```
myMIDIConnections = getMIDIConnections(echoPlugin);  
isempty(myMIDIConnections)
```

```
ans =
```

```
1
```

Add MIDI connections using `configureMIDI`.

```
configureMIDI(echoPlugin, 'Delay1');
```

Get the MIDI connections of `echoPlugin` using `getMIDIConnections`. The MIDI connections you configured are saved as a structure. View details of the MIDI connections using dot notation.

```
myMIDIConnections = getMIDIConnections(echoPlugin);
myMIDIConnections.Delay1

ans =

    Law: 'lin'
    Min: 0
    Max: 1
MIDIControl: 'any control on 'BCF2000''
```

Use `disconnectMIDI` to remove MIDI connections between your `echoPlugin` object and your MIDI device.

```
disconnectMIDI(echoPlugin);
```

Get MIDI connections of `echoPlugin` and verify that you have successfully disconnected MIDI controls from your plugin.

```
myMIDIConnections = getMIDIConnections(echoPlugin);
isempty(myMIDIConnections)

ans =

    1
```

Input Arguments

myAudioPlugin — Audio plugin
object

Audio plugin, specified as an object that inherits from the `audioPlugin` class or the `audioPluginSource` class.

More About

- “Musical Instrument Digital Interface (MIDI)”

See Also

`audioPlugin` | `audioPluginSource` | `configureMIDI` | `getMIDIConnections` | `midicallback` | `midicontrols` | `midiid` | `midiread` | `midisync`

Introduced in R2016a

fdesign.audioweighting

Audio weighting filter specification object

Syntax

```
HAWf = fdesign.audioweighting  
HAWf = fdesign.audioweighting(spec)  
HAWf = fdesign.audioweighting(spec,specvalue1,specvalue2)  
HAWf = fdesign.audioweighting(specvalue1,specvalue2)  
HAWf = fdesign.audioweighting(...,Fs)
```

Description

Supported audio weighting filter types are: A weighting, C weighting, C-message, ITU-T 0.41, and ITU-R 468–4 weighting.

HAWf = fdesign.audioweighting constructs an audio weighting filter specification object *HAWf* with a weighting type of A and a filter class of 1. Use the **design** method to instantiate a **dfilt** object based on the specifications in *HAWf*. Use **designmethods** to find valid filter design methods. Because the standards for audio weighting filters are in Hz, normalized frequency specifications are not supported for **fdesign.audioweighting** objects. The default sampling frequency for A weighting, C weighting, C-message, and ITU-T 0.41 filters is 48 kHz. The default sampling frequency for the ITU-R 468–4 filter is 80 kHz. If you invoke the **normalizefreq** method, a warning is issued when you instantiate the **dfilt** object and the default sampling frequencies in Hz are used.

HAWf = fdesign.audioweighting(*spec*) returns an audio weighting filter specification object using default values for the specification string in *spec*. The following are valid entries for *spec*. The entries are not case sensitive.

- 'WT,Class' (default *spec*)

The 'WT,Class' specification is valid for A weighting and C weighting filters of class 1 or 2.

The weighting type is specified by the string: 'A' or 'C'. The class is the scalar 1 or 2.

The default values for 'WT,Class' are 'A',1.

- 'WT'

The 'WT' specification is valid for C-message (default), ITU-T 0.41, and ITU-R 468–4 weighting filters.

The weighting type is specified by the string: 'Cmessage', 'ITUT041', or 'ITUR4684'.

HAWf = fdesign.audioweighting(*spec*,*specvalue1*,*specvalue2*) constructs an audio weighting filter specification object *HAWf* and sets its specifications at construction time.

HAWf = fdesign.audioweighting(*specvalue1*,*specvalue2*) constructs an audio weighting filter specification object *HAWf* with the specification 'WT,Class' using the values you provide. The valid weighting types are 'A' or 'C'.

HAWf = fdesign.audioweighting(...,Fs) specifies the sampling frequency in Hz. The sampling frequency is a scalar trailing all other input arguments.

Input Arguments

Parameter Name/Value Pairs

'WT'

Weighting type

The weighting type defines the frequency response of the filter. The valid weighting types are: A weighting, C weighting, C-message, ITU-T 0.41, and ITU-R 468–4 weighting. The weighting types are described in “Definitions” on page 2-63.

'Class'

Filter Class

Filter class is only applicable for A weighting and C weighting filters. The filter class describes the frequency-dependent tolerances specified in the relevant standards [1],

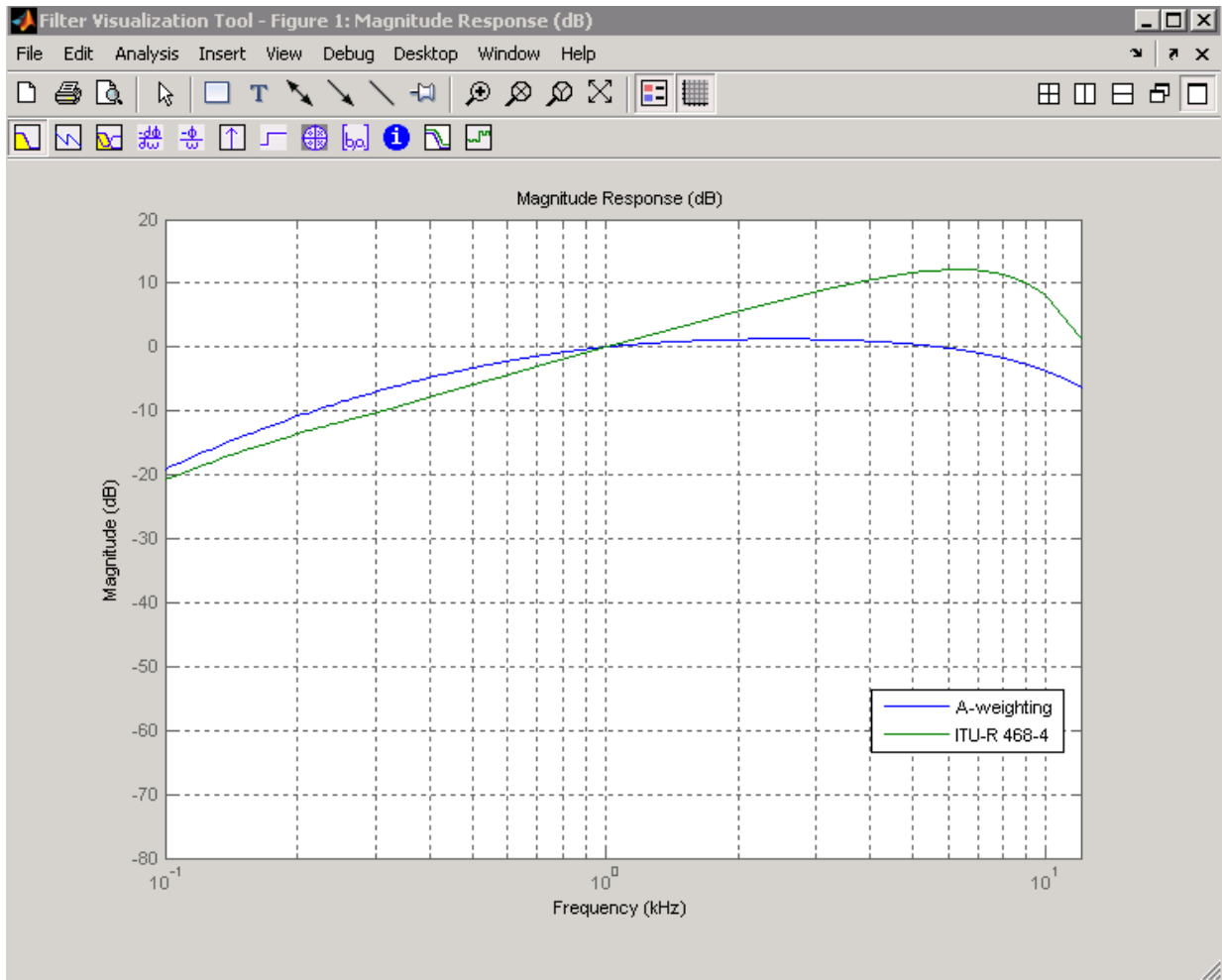
[2]. There are two possible class values: 1 and 2. Class 1 weighting filters have stricter tolerances than class 2 filters. The filter class value does not affect the design. The class value is only used to provide a specification mask in `fvtool` for the analysis of the filter design.

Default: 1

Examples

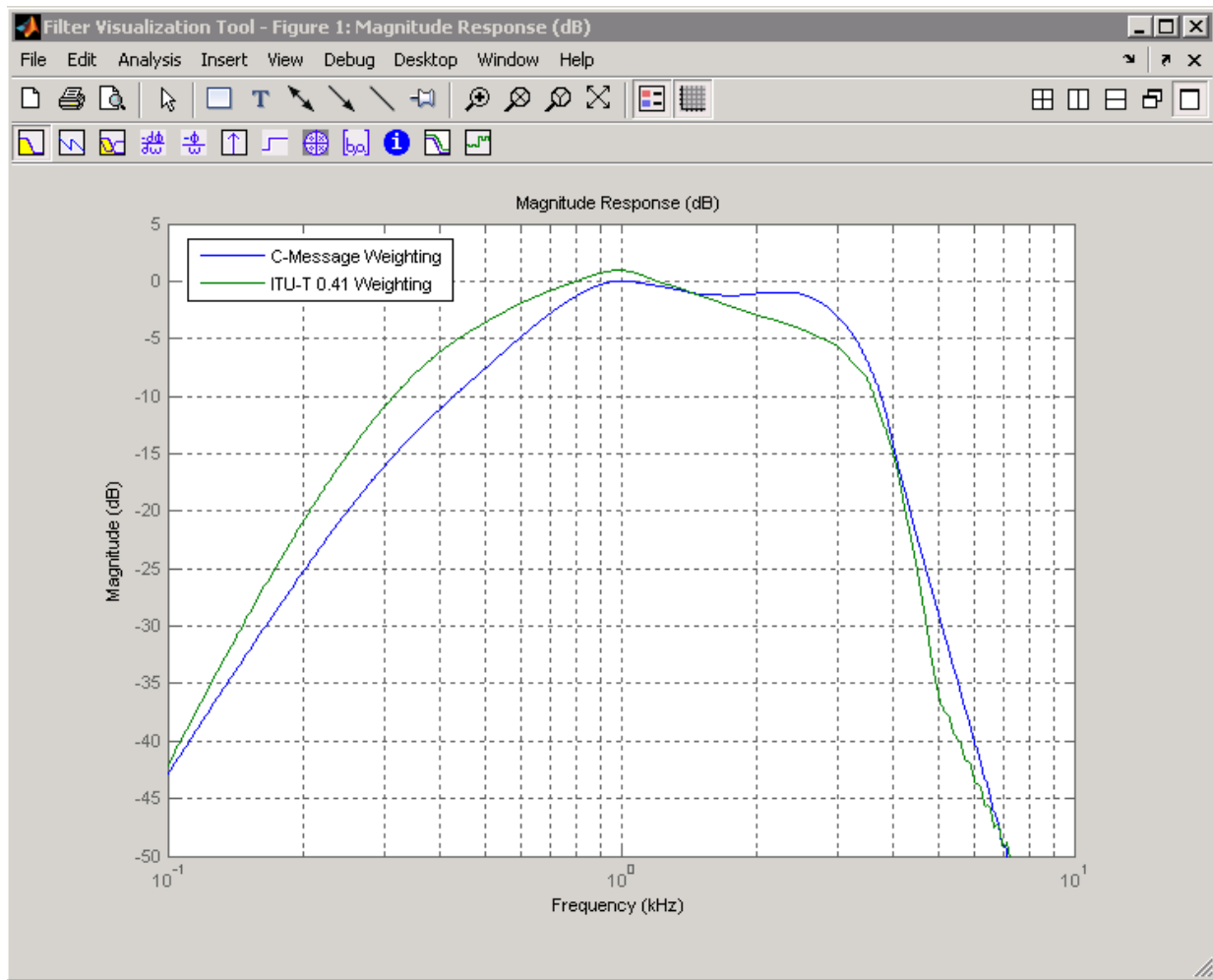
Compare class 1 A weighting and ITU-R 468–4 filters between 0.1 and 12 kHz:

```
HawfA = fdesign.audioweighting('WT,Class','A',1,44.1e3);  
% Sampling frequency is 44.1 kHz  
HawfITUR = fdesign.audioweighting('WT','ITUR4684',44.1e3);  
Afilter = design(HawfA);  
ITURfilter = design(HawfITUR);  
hfvtool([Afilter ITURfilter]);  
axis([0.1 12 -80 20]);  
legend(hfvtool,'A-weighting','ITU-R 468-4');
```



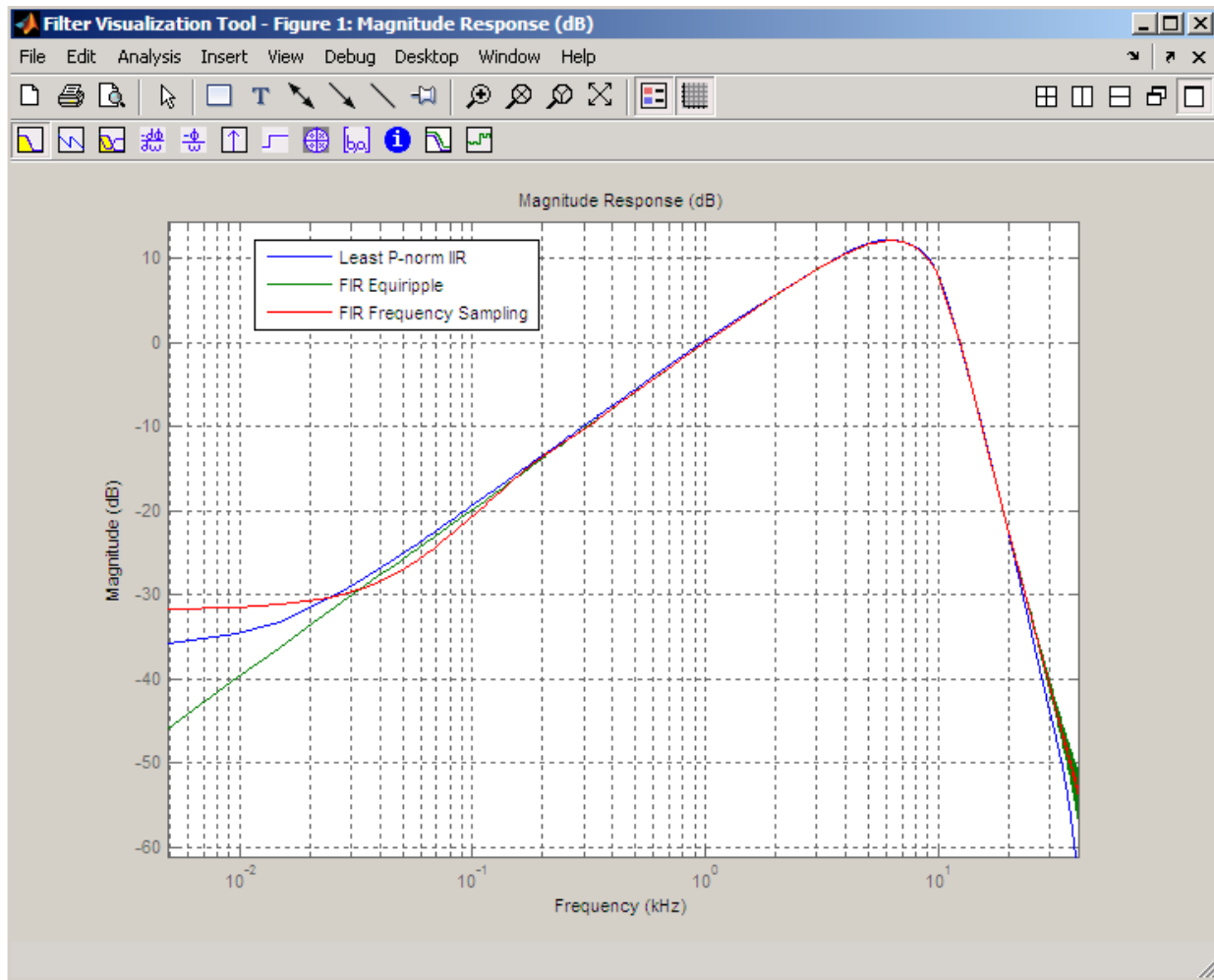
Compare C-message and ITU-T 0.41 filters:

```
hCmessage = fdesign.audioweighting('WT','Cmessage',24e3);
hITUT = fdesign.audioweighting('WT','ITUT041',24e3);
dCmessage = design(hCmessage);
dITUT = design(hITUT);
hfvt = fvtool([dCmessage dITUT]);
legend(hfvt,'C-Message Weighting','ITU-T 0.41 Weighting');
axis([0.1 10 -50 5]);
```



Construct an ITU-R 468–4 filter using all available design methods:

```
HAwf = fdesign.audioweighting('WT','ITUR4684');
designmethods(HAwf)
% returns iirlpnorm,equiripple,freqsamp
D = design(HAwf,'all'); % returns all designs
hfvtool(D);
legend(hfvtool,'Least P-norm IIR','FIR Equiripple',...,
'FIR Frequency Sampling')
```



More About

A weighting

The specifications for the A weighting filter are found in ANSI standard S1.42-2001. The A weighting filter is based on the 40-phon Fletcher-Munson equal loudness contour [3]. One phon is equal to one dB sound pressure level (SPL) at one kHz. The Fletcher-

Munson equal loudness contours are designed to account for frequency and level dependent differences in the perceived loudness of tonal stimuli. The 40-phon contour reflects the fact that the human auditory system is more sensitive to frequencies around 1–2 kHz than lower and higher frequencies. The filter roll off is more pronounced at lower frequencies and more modest at higher frequencies. While A weighting is based on the equal loudness contour for low-level (40-phon) tonal stimuli, it is commonly used in the United States for assessing potential health risks associated with noise exposure to narrowband and broadband stimuli at high levels.

C weighting

The specifications for the C weighting filter are found in ANSI standard S1.42-2001. The C weighting filter approximates the 100-phon Fletcher-Munson equal loudness contour for tonal stimuli. The C weighting magnitude response is essentially flat with 3-dB frequencies at 31.5 Hz and 8000 Hz. While C weighting is not as common as A weighting, sound level meters frequently have a C weighting filter option.

C-message

The specifications for the C-message weighting filter are found in Bell System Technical Reference, PUB 41009. C-message weighting filters are designed for measuring the impact of noise on telecommunications circuits used in speech transmission [6]. C-message weighting filters are commonly used in North America, while the ITU-T 0.41 filter is more commonly used outside of North America.

ITU-R 468-4

The specifications for the ITU-R 468-4 weighting filter are found in the International Telecommunication Union Recommendation ITU-R BS.468-4. ITU-R 468-4 is an equal loudness contour weighting filter. Unlike the A weighting filter, the ITU-R 468-4 filter describes subjective loudness judgements for broadband stimuli [4]. A common criticism of the A weighting filter is that it underestimates the loudness judgement of real-world stimuli particularly in the frequency band from about 1–9 kHz. A comparison of A weighting and ITU-R 468-4 weighting curves shows that the ITU-R 468-4 curve applies more gain between 1 and 10 kHz with a peak difference of approximately 12 dB around 6–7 kHz.

ITU-T 0.41

The specifications for the ITU-T 0.41 filter are found in the ITU-T Recommendation 0.41. ITU-T 0.41 weighting filters are designed for measuring the impact of noise on telecommunications circuits used in speech transmission [5]. ITU-T 0.41 weighting filters

are commonly used outside of North America, while the C-message weighting filter is more common in North America.

- “Audio Weighting Filters”
- “Design a Filter in Fdesign — Process Overview”

References

- [1] *American National Standard Design Response of Weighting Networks for Acoustical Measurements*, ANSI S1.42-2001, Acoustical Society of America, New York, NY, 2001.
- [2] *Electroacoustics Sound Level Meters Part 1: Specifications*, IEC 61672-1, First Edition 2002-05.
- [3] Fletcher, H. and W.A. Munson. “Loudness, its definition, measurement and calculation.” *Journal of the Acoustical Society of America*, Vol. 5, 1933, pp. 82–108.
- [4] *Measurement of Audio-Frequency Noise Voltage Level in Sound Broadcasting*, International Telecommunication Union Recommendation ITU-R BS.468-4, 1986.
- [5] *Psophometer for Use on Telephone-Type Circuits*, ITU-T Recommendation 0.41.
- [6] *Transmission Parameters Affecting Voiceband Data Transmission-Measuring Techniques*, Bell System Technical Reference, PUB 41009, 1972.

See Also

[design](#) | [designmethods](#) | [fdesign](#) | [fvtool](#)

fdesign.octave

Octave filter specification

Syntax

```
d = fdesign.octave(l)
d = fdesign.octave(l, MASK)
d = fdesign.octave(l, MASK, spec)
d = fdesign.octave(..., Fs)
```

Description

`d = fdesign.octave(l)` constructs an octave filter specification object `d`, with `l` bands per octave. The default value for `l` is one.

Note: The filters created by `fdesign.octave` comply with the ANSI[®] S1.11-2004 and IEC 61260:1995 standards.

`d = fdesign.octave(l, MASK)` constructs an octave filter specification object `d` with `l` bands per octave and `MASK` specification for the FVTool. The available values for `mask` are:

- 'class 0'
- 'class 1'
- 'class 2'

`d = fdesign.octave(l, MASK, spec)` constructs an octave filter specification object `d` with `l` bands per octave, `MASK` specification for the FVTool, and the `spec` specification string. The specification strings available are:

- 'N, F0'

(not case sensitive), where:

- N is the filter order
- F0 is the center frequency. The center frequency is specified in normalized frequency units assuming a sampling frequency of 48 kHz, unless a sampling frequency in Hz is included in the specification: `d = fdesign.octave(..., Fs)`. If you specify an invalid center frequency, a warning is issued and the center frequency is rounded to the nearest valid value. You can determine the valid center frequencies for your design by using `validfrequencies` with your octave filter specification object. For example:

```
d = fdesign.octave(1, 'Class 1', 'N,F0',6,1000,44.1e3);
validcenterfreq = validfrequencies(d);
Valid center frequencies:
```

- Must be greater than 20 Hz and less than 20 kHz if you specify a sampling frequency. The range 20 Hz to 20 kHz is the standard range of human hearing.
- Are calculated according to the following algorithm if the number of bands per octave, L, is even

```
G = 10^(3/10);
x = -1000:1350;
validcenterfreq = 1000*(G.^((2*x-59)/(2*L)));
validcenterfreq = validcenterfreq(validcenterfreq>20 & validcenterfreq<2e4);
```

Only center frequencies greater than 20 and less than 20000 are retained. Choosing a center frequency greater than your Nyquist frequency (1/2 the sampling rate) results in an error when you design the filter. If you do not specify a sampling frequency, the remaining center frequencies are divided by 24000 to obtain valid normalized center frequencies. For `fdesign.octave`, normalized frequency assumes a sampling frequency of 48 kHz.

```
validcenterfreq = validcenterfreq/24000;
```

- Are calculated according to the following algorithm if the number of bands per octave, L, is odd

```
G = 10^(3/10);
x = -1000:1350;
validcenterfreq = 1000*(G.^((x-30)/L));
validcenterfreq = validcenterfreq(validcenterfreq>20 & validcenterfreq<2e4);
```

Only center frequencies greater than 20 and less than 20000 are retained. Choosing a center frequency greater than your Nyquist frequency (1/2 the sampling rate) results in an error when you design the filter. If you do not specify a sampling frequency, the remaining center frequencies are divided by

24000 to obtain valid normalized center frequencies. For `fdesign.octave`, normalized frequency assumes a sampling frequency of 48 kHz.

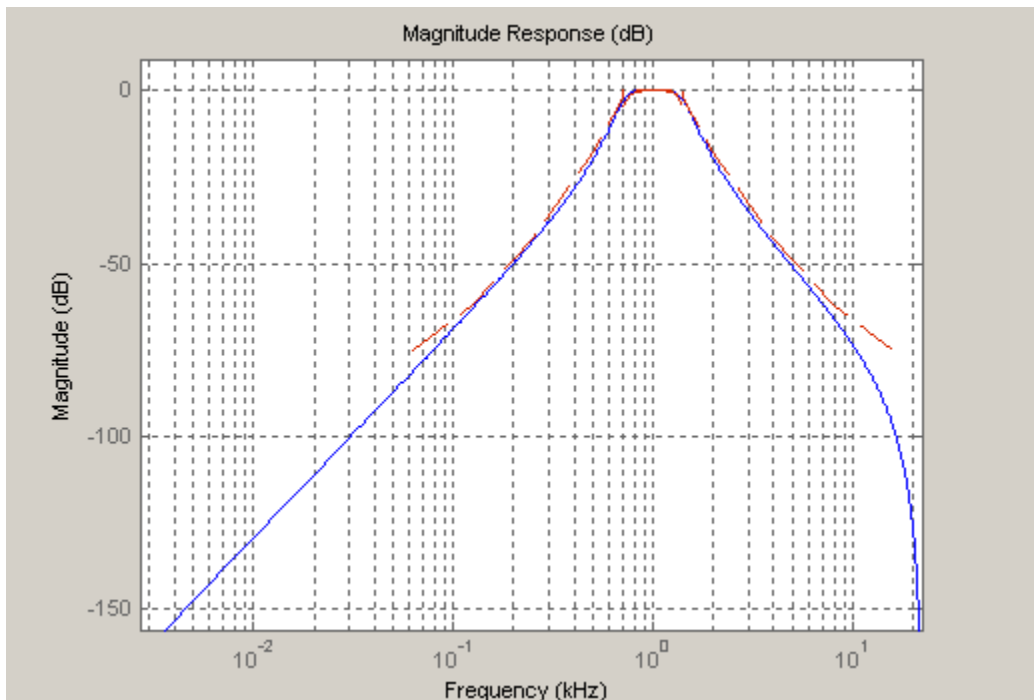
```
validcenterfreq = validcenterfreq/24000;
```

Examples

Design an sixth order, octave-band class 0 filter with a center frequency of 1000 Hz and, a sampling frequency of 44.1 kHz.

```
d = fdesign.octave(1, 'Class 0', 'N,F0', 6, 1000, 44100);  
Hd = design(d);  
fvtool(Hd)
```

The following figure shows the magnitude response plot of the filter. The logarithmic scale for frequency is automatically set by FVTool for the octave filters.



See Also
fdesign

fdesign.parmeq

Parametric equalizer filter specification

Syntax

```
d = fdesign.parmeq(spec, specvalue1, specvalue2, ...)  
d = fdesign.parmeq(... fs)
```

Description

`d = fdesign.parmeq(spec, specvalue1, specvalue2, ...)` constructs a parametric equalizer filter design object, where `spec` is a non-case sensitive specification string. The choices for `spec` are as follows:

- 'FO, BW, BWp, Gref, GO, GBW, Gp' (minimum order default)
- 'FO, BW, BWst, Gref, GO, GBW, Gst'
- 'FO, BW, BWp, Gref, GO, GBW, Gp, Gst'
- 'N, FO, BW, Gref, GO, GBW'
- 'N, FO, BW, Gref, GO, GBW, Gp'
- 'N, FO, Fc, Qa, GO'
- 'N, FO, Fc, S, GO'
- 'N, FO, BW, Gref, GO, GBW, Gst'
- 'N, FO, BW, Gref, GO, GBW, Gp, Gst'
- 'N, Flow, Fhigh, Gref, GO, GBW'
- 'N, Flow, Fhigh, Gref, GO, GBW, Gp'
- 'N, Flow, Fhigh, Gref, GO, GBW, Gst'
- 'N, Flow, Fhigh, Gref, GO, GBW, Gp, Gst'

where the parameters are defined as follows:

- BW — Bandwidth
- BWp — Passband Bandwidth

- BWst — Stopband Bandwidth
- Gref — Reference Gain (decibels)
- G0 — Center Frequency Gain (decibels)
- GBW — Gain at which Bandwidth (BW) is measured (decibels)
- Gp — Passband Gain (decibels)
- Gst — Stopband Gain (decibels)
- N — Filter Order
- F0 — Center Frequency
- Fc — Cutoff frequency
- Fhigh - Higher Frequency at Gain GBW
- Flow - Lower Frequency at Gain GBW
- Qa-Quality Factor
- S-Slope Parameter for Shelving Filters

Regardless of the specification string chosen, there are some conditions that apply to the specification parameters. These are as follows:

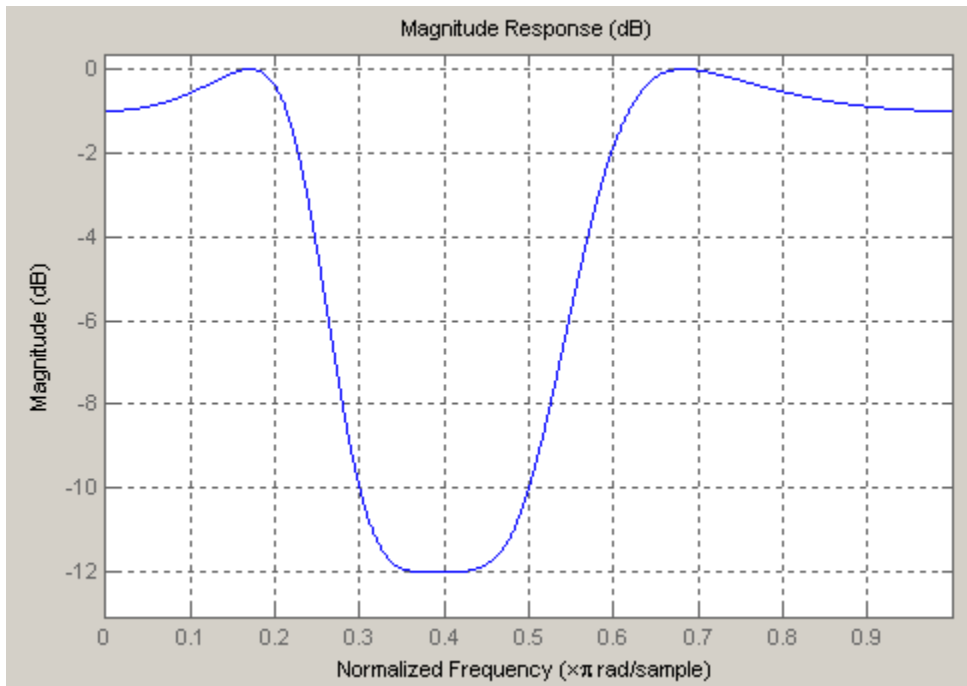
- Specifications for parametric equalizers must be given in decibels
- To boost the input signal, set $G0 > Gref$; to cut, set $Gref > G0$
- For boost: $G0 > Gp > GBW > Gst > Gref$; For cut: $G0 < Gp < GBW < Gst < Gref$
- Bandwidth must satisfy: $BWst > BW > BWp$

`d = fdesign.parmeq(... fs)` adds the input sampling frequency. Fs must be specified as a scalar trailing the other numerical values provided, and is assumed to be in Hz.

Examples

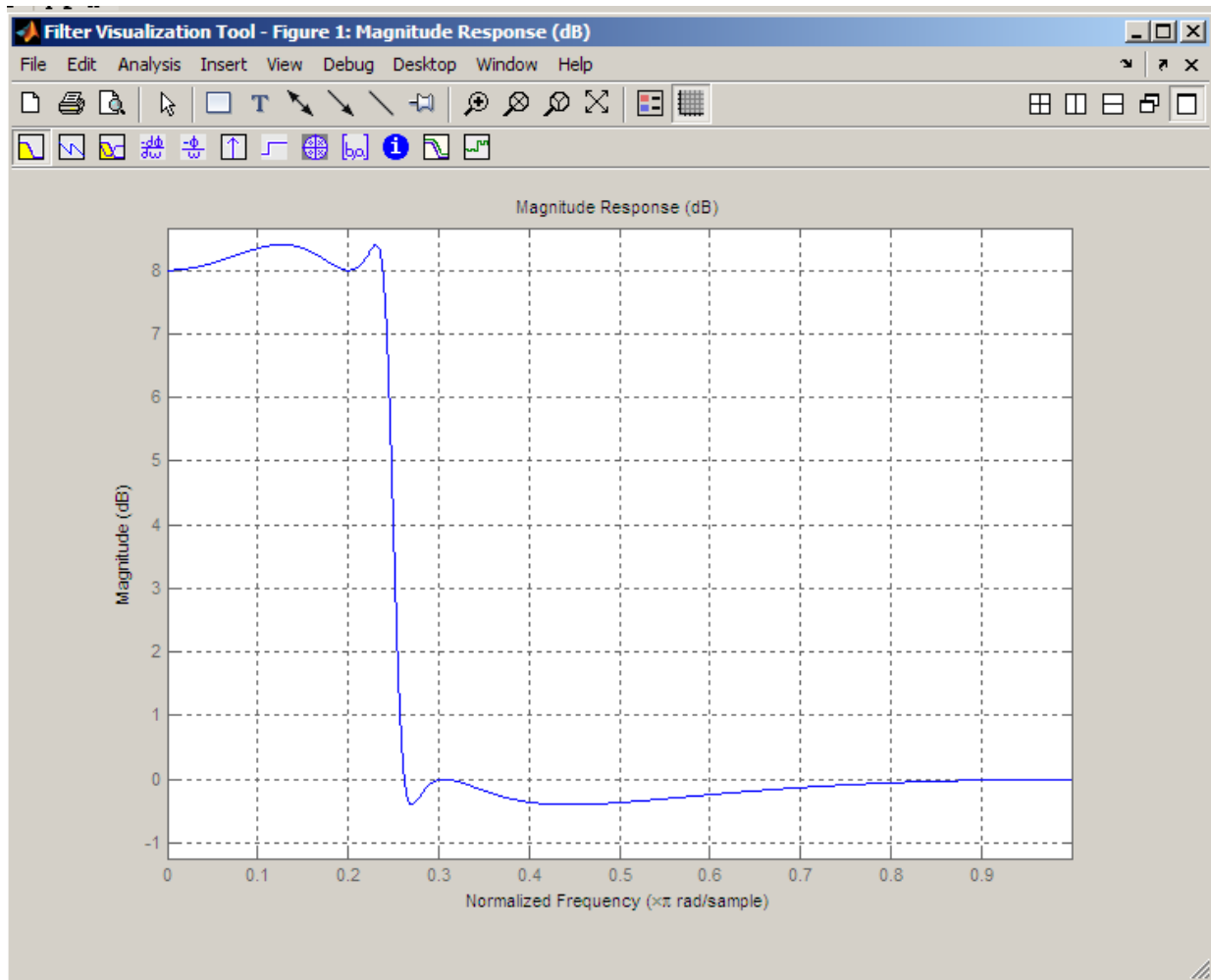
Design a Chebyshev Type II parametric equalizer filter that cuts by 12 dB:

```
d = fdesign.parmeq('N,Flow,Fhigh,Gref,G0,GBW,Gst',...
    4,.3,.5,0,-12,-10,-1);
Hd = design(d,'cheby2');
fvtool(Hd)
```



Design a 4th order audio lowpass ($F_0 = 0$) shelving filter with cutoff frequency of $F_c = 0.25$, quality factor $Q_a = 10$, and boost gain of $G_0 = 8$ dB:

```
d = fdesign.parmeq('N,F0,Fc,Qa,G0',4,0,0.25,10,8);  
Hd = design(d);  
fvtool(Hd)
```



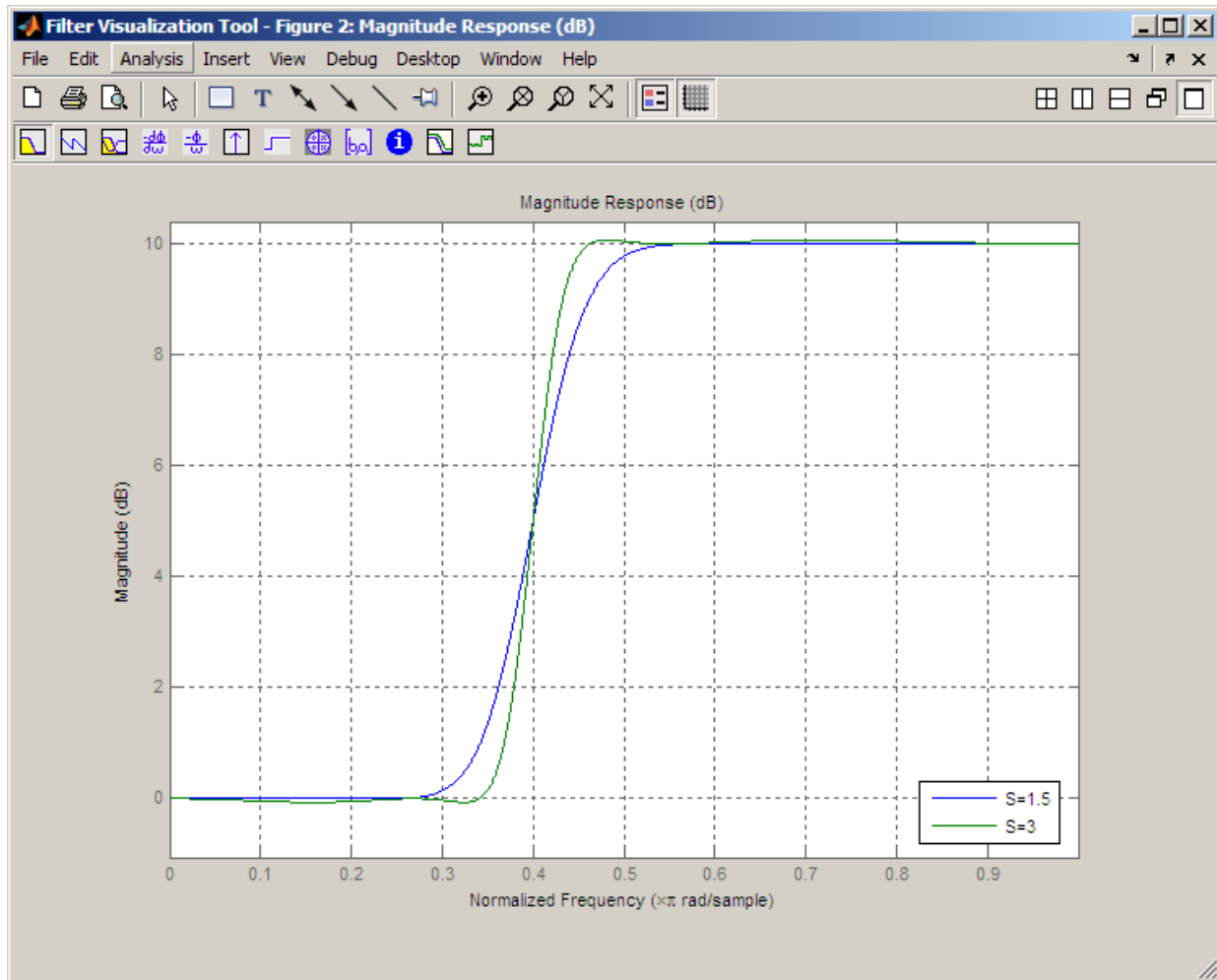
Design 4th-order highpass shelving filters with $S=1.5$ and $S=3$:

```

N=4;
FO = 1;
Fc = .4; % Cutoff Frequency
GO = 10;
S = 1.5;
S2=3;
f = fdesign.parmeq('N,FO,Fc,S,GO',N,FO,Fc,S,GO);

```

```
h1 = design(f);  
f.S=3;  
h2=design(f);  
hfvtool([h1 h2]);  
set(hfvtool,'Filters',[h1 h2]);  
legend(hfvtool,'S=1.5','S=3');
```



More About

- “Parametric Equalizer Design”

See Also

fdesign

generateAudioPlugin

Generate audio plugin from MATLAB class

Syntax

```
generateAudioPlugin className  
generateAudioPlugin options className
```

Description

`generateAudioPlugin className` generates a VST 2 audio plugin from a MATLAB class specified by `className`. See “Supported Compilers” on page 2-78 for a list of compilers supported by `generateAudioPlugin`.

`generateAudioPlugin options className` specifies nondefault output folder, plugin name, or file type. Options can be specified in any grouping, and in any order.

The extension of your generated plugin depends on your operating system.

Operating System	File Extension
Windows	.dll
OSX	.vst

Examples

Generate Audio Plugin

```
generateAudioPlugin audiopluginexample.Echo
```

A VST 2 plugin named `Echo` is saved to your current folder. The extension of your plugin depends on your operating system.

Specify Output Folder for Generated Plugin

```
generateAudioPlugin -outdir myPluginFolder audiopluginexample.Echo
```

A VST 2 plugin named Echo is saved to your specified folder. The extension of your plugin depends on your operating system.

Specify File Name of Generated Plugin

```
generateAudioPlugin -output awesomeEffect audiopluginexample.Echo
```

A VST 2 plugin named awesomeEffect is saved to your current folder. The extension of your plugin depends on your operating system.

Specify Output Folder and File Name of Generated Plugin

```
generateAudioPlugin -output awesomeEffect -outdir myPluginFolder audiopluginexample.Echo
```

A VST 2 plugin named awesomeEffect is saved to your specified folder. The extension of your plugin depends on your operating system.

Generate win32 Plugin from win64 System

```
generateAudioPlugin -win32 audiopluginexample.Echo
```

A 32-bit VST 2 plugin named Echo.dll is saved to your current folder.

Input Arguments

options — Options to specify output folder, plugin name, and file type

```
-outdir folder | -output pluginName | -win32
```

Options can be specified in any grouping, and in any order.

Option	Description
-outdir <i>folder</i>	Generates a plugin to a specific folder. By default, the generated plugin is placed in the current folder. If <i>folder</i> is not in the current directory, specify the exact path.
-output <i>pluginName</i>	Specifies the file name of the generated plugin. The appropriate extension is appended to the <i>pluginName</i> based on the platform on which the plugin is generated. By default, the plugin is named after the class.
-win32	Creates a 32-bit audio plugin. Valid only on win64.

className — Name of the plugin class to generate

plugin class

Name of the plugin class to generate. The plugin class must be on the MATLAB path. It must derive from either the `audioPlugin` class or the `audioPluginSource` class.

Note: `className` is not the name of a file. Arguments such as `'myPlugin.m'` issue an error.

More About

Supported Compilers

Compilers supported by `generateAudioPlugin`.

Operating System	Supported Compilers
win64	Microsoft Visual C++ 2013 Professional Microsoft Visual C++ 2012 Professional Microsoft Visual C++ 2010 Professional SP1
maci64	Xcode 6.2

- “Export a MATLAB Plugin to a DAW”

See Also

`audioPlugin` | `audioPluginSource` | Audio Test Bench | `validateAudioPlugin`

Introduced in R2016a

getMIDIConnections

Get MIDI connections of audio plugin

Syntax

```
connectionInfo = getMIDIConnections(myAudioPlugin)
```

Description

`connectionInfo = getMIDIConnections(myAudioPlugin)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your audio plugin, `myAudioPlugin`. Only those MIDI connections established using `configureMIDI` are returned.

The `connectionInfo` structure contains a substructure for each tunable property of `myAudioPlugin` that has established MIDI connections. Each substructure contains the control number, the device name of the corresponding MIDI control, and the property mapping information (mapping rule, minimum value, and maximum value).

Examples

Get MIDI Connections of Plugin

Create an object of the audio plugin `example audiopluginexample.Echo`.

```
echoEffect = audiopluginexample.Echo;
```

Use `configureMIDI` to synchronize `echoEffect` properties with specific MIDI controls on the default MIDI device.

```
configureMIDI(echoEffect, 'Delay1', 1001);  
configureMIDI(echoEffect, 'Gain1', 1002);  
configureMIDI(echoEffect, 'Delay2', 1003);  
configureMIDI(echoEffect, 'Gain2', 1004);
```

Use `getMIDIConnections` to view the MIDI connections you established.

```
connectionInfo = getMIDIConnections(echoEffect)
```

```
connectionInfo =  
  
    Delay1: [1x1 struct]  
    Gain1: [1x1 struct]  
    Delay2: [1x1 struct]  
    Gain2: [1x1 struct]
```

View details of the `Delay1` MIDI connection using dot notation.

```
connectionInfo.Delay1  
  
ans =  
  
    Law: 'lin'  
    Min: 0  
    Max: 1  
MIDIControl: 'control 1001 on 'nanoKONTROL2''
```

Input Arguments

myAudioPlugin — Audio plugin

object

Audio plugin, specified as an object that inherits from the `audioPlugin` class.

Output Arguments

connectionInfo — Information about MIDI connection

structure

Information about MIDI connection between the specified audio plugin object and MIDI devices, returned as a structure. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each established MIDI connection. Each substructure contains the control number, the device name of the corresponding MIDI control, and the property mapping information (mapping rule, minimum value, and maximum value).

More About

- “Musical Instrument Digital Interface (MIDI)”

See Also

`audioPlugin` | `audioPluginSource` | `configureMIDI` | `disconnectMIDI` |
`midicallback` | `midicontrols` | `midiid` | `midiread` | `midisync`

Introduced in R2016a

midicallback

Call function handle when MIDI controls change value

Syntax

```
oldFunctionHandle = midicallback(midicontrolsObject,functionHandle)
oldFunctionHandle = midicallback(midicontrolsObject,[])
currentFunctionHandle = midicallback(midicontrolsObject)
```

Description

`oldFunctionHandle = midicallback(midicontrolsObject,functionHandle)` sets `functionHandle` as the function handle called when `midicontrolsObject` changes value, and returns the previous function handle, `oldFunctionHandle`.

`oldFunctionHandle = midicallback(midicontrolsObject,[])` clears the function handle.

`currentFunctionHandle = midicallback(midicontrolsObject)` returns the current function handle.

Examples

Interactively Read MIDI Controls

Create a default MIDI controls object. Use `midicallback` to associate an anonymous function with your MIDI controls object, `mc`.

```
mc = midicontrols;
midicallback(mc,@(x)disp(midiread(x)));
```

Move any control on your default MIDI device to display its current normalized value on the command line.

```
0.5079
```



```
0.5000
0.4921
0.4841
0.4762
0.4683
0.4603
0.4683
```

Use midicallback to Update Plot

Use `midiid` to identify the name of your MIDI device and a specified control. Move the MIDI control you want to identify.

```
[controlNumber,deviceName] = midiid;
```

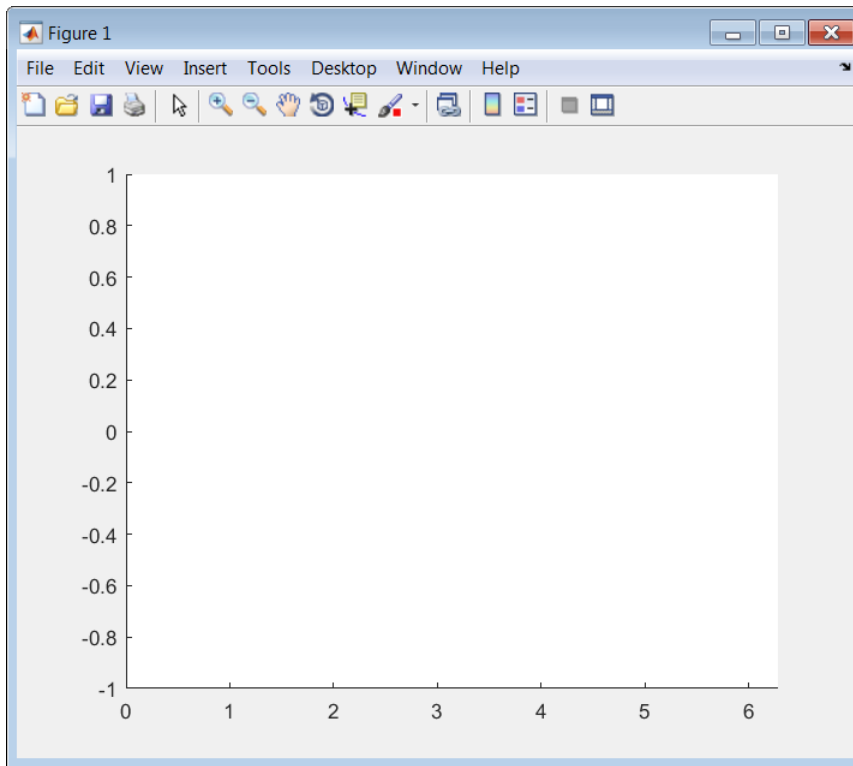
```
Move the control you wish to identify; type ^C to abort.
Waiting for control message...
```

Create an object that responds to the control you specified.

```
midicontrolsObject = midicontrols(controlNumber);
```

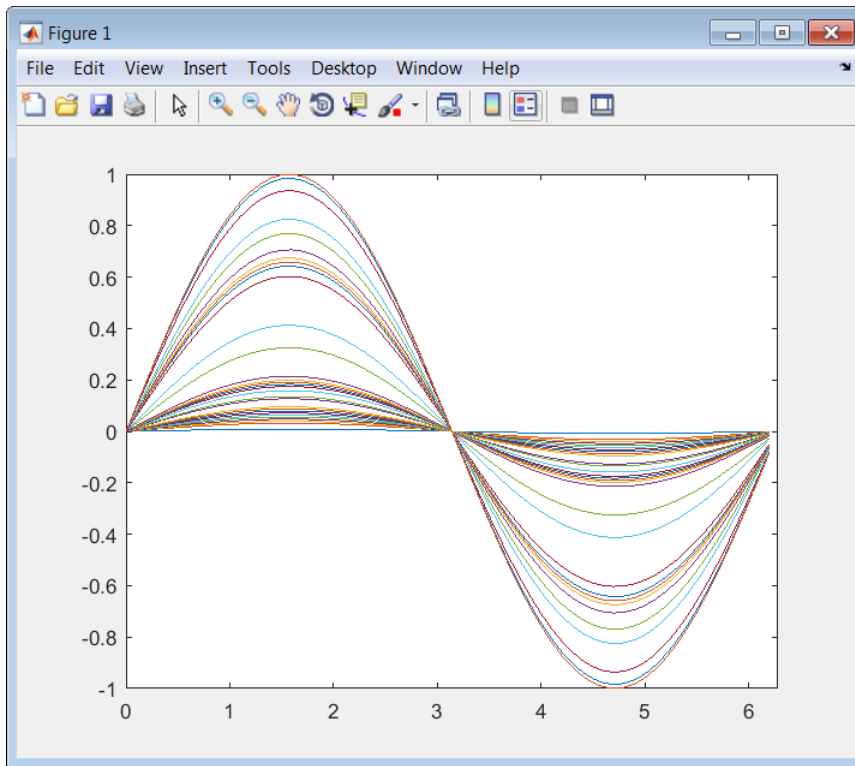
Define a function that plots a sinusoid with the amplitude set by your MIDI control. Make the axis constant.

```
axis([0,2*pi,-1,1]);
axis manual
hold on
sinePlotter = @(obj) plot(0:0.1:2*pi,midiread(obj).*sin(0:0.1:2*pi));
```



Use the `midicallback` function to associate your `sinePlotter` function with the control specified by your `midicontrolsObject`. Move your specified MIDI control. The plot updates automatically with the sinusoid amplitude specified by your MIDI control.

```
midicallback(midicontrolsObject,sinePlotter)
```



Change Function Handle Associated with MIDI Control

Create an object that responds to any control on the default MIDI device.

```
midicontrolsObject = midicontrols;
```

Define an anonymous function to display the current value of the MIDI control. Use `midicallback` to associate your MIDI control object with the function you created. Verify that your object is associated with your function.

```
displayControlValue = @(object) disp(midiread(object));
midicallback(midicontrolsObject, displayControlValue);
currentFunctionHandle = midicallback(midicontrolsObject)
```

```
currentFunctionHandle =
```

```
@(object)disp(midiread(object))
```

Move any control on your default MIDI device to display its current normalized value on the command line.

```
0.3095
0.4603
0.6746
0.7381
0.8175
0.8571
0.9048
```

Define an anonymous function to print the current value of the MIDI control rounded to two significant digits. Use `midicallback` to associate your MIDI controls object with the function you created. Return the old function handle.

```
displayRoundedControlValue = @(object) fprintf('%0.2f\n',midiread(object));
oldFunctionHandle = midicallback(midicontrolsObject,displayRoundedControlValue)

oldFunctionHandle =

    @(object)disp(midiread(object))
```

Move a control to display its current normalized value rounded to two significant digits.

```
0.91
0.83
0.67
0.49
0.29
0.18
0.05
```

Remove the association between the object and the function. Return the old function handle.

```
oldFunctionHandle = midicallback(midicontrolsObject,[])
```

```
oldFunctionHandle =
    @(object)fprintf('%.2f\n',midiread(object))
```

Verify that no function is associated with your MIDI controls object.

```
currentFunctionHandle = midicallback(midicontrolsObject)
currentFunctionHandle =
    []
```

Associate a Function with MIDI Controls

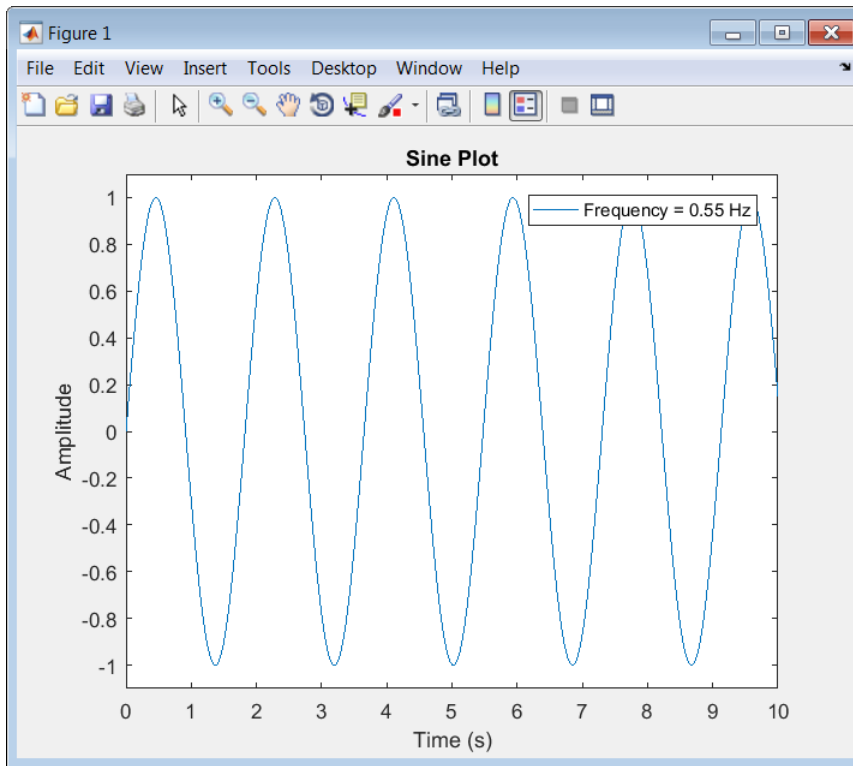
Define this function and save it to your current folder.

```
function plotSine(midicontrolsObject)
frequency = midiread(midicontrolsObject);
x = 0:0.01:10;
sinusoid = sin(2*pi*frequency.*x);
plot(x,sinusoid)
axis([0,10,-1.1,1.1]);
ylabel('Amplitude');
xlabel('Time (s)');
title('Sine Plot')
legend(sprintf('Frequency = %0.2f Hz',frequency));
end
```

Create a `midicontrols` object. Create a function handle for your `plotSine` function. Use `midicallback` to associate your `midicontrolsObject` with `plotSineHandle`.

Move any controller on your MIDI device to plot a sinusoid. The sinusoid frequency updates when you move MIDI controls.

```
midicontrolsObject = midicontrols;
plotSineHandle = @plotSine;
midicallback(midicontrolsObject,plotSineHandle);
```



Input Arguments

midcontrolsObject — Object that listens to the controls on a MIDI device
object

Object that listens to the controls on a MIDI device, specified as an object created by `midcontrols`.

functionHandle — New function handle
function handle

New function handle, specified as a function handle that contains one input argument. The new function handle is called when `midcontrolsObject` changes value. For information on what function handles are, see “Function Handles”.

Output Arguments

oldFunctionHandle — Old function handle

function handle

Old function handle set by the previous call to `midicallback`, returned as a function handle.

currentFunctionHandle — Current function handle

function handle

The function handle set by the most recent call to `midicallback`, returned as a function handle.

More About

- “Musical Instrument Digital Interface (MIDI)”

See Also

`configureMIDI` | `disconnectMIDI` | `getMIDIConnections` | `midicontrols` | `midiid` | `midiread` | `midisync` | `setpref`

midicontrols

Open group of MIDI controls for reading

Syntax

```
midicontrolsObject = midicontrols
midicontrolsObject = midicontrols(controlNumbers)
midicontrolsObject = midicontrols(controlNumbers,initialValues)
midicontrolsObject = midicontrols( ____, 'MIDIDevice',deviceName)
midicontrolsObject = midicontrols( ____, 'OutputMode',mode)
```

Description

`midicontrolsObject = midicontrols` returns an object that listens to all controls on your default MIDI device.

Call `midiread` with the object to return the values of controls on your MIDI device. If you call `midiread` before a control is moved, `midiread` returns the initial value of your `midicontrols` object.

`midicontrolsObject = midicontrols(controlNumbers)` listens to controls specified by `controlNumbers` on your default MIDI device.

`midicontrolsObject = midicontrols(controlNumbers,initialValues)` specifies `initialValues` associated with `controlNumbers`.

`midicontrolsObject = midicontrols(____, 'MIDIDevice',deviceName)` specifies the MIDI device your `midicontrols` object listens to, using any of the previous syntaxes.

`midicontrolsObject = midicontrols(____, 'OutputMode',mode)` specifies the range of values returned by `midiread` and accepted as `initialValues` for `midicontrols` and as `controlValues` for `midisync`.

Examples

Listen to Any Control on Default Device

Create a `midicontrols` object and read the default control value.

```
midicontrolsObject = midicontrols
midiread(midicontrolsObject)

midicontrolsObject =
midicontrols object: any control on 'BCF2000'

ans =
    0
```

Move any control on your MIDI device. Use `midiread` to return the most recent value of the last control moved.

```
midiread(midicontrolsObject)

ans =
    0.3810
```

Listen to Specific Control

Use `midiid` to identify the name of your MIDI device and a specified control. Move the MIDI control you want to identify.

```
[controlNumber,deviceName] = midiid;
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message...
```

Create an object that responds to the control you specified.

```
midicontrolsObject = midicontrols(controlNumber);
```

Move your selected MIDI control, and then use `midiread` to return its most recent value.

```
midicontrolsObject = midiread(midicontrolsObject);
```

```
ans =  
  
    0.4048
```

Specify Control Numbers and Initial Value

Determine the control numbers of four different controls on your MIDI device.

```
[controlNumber1,~] = midiid;  
[controlNumber2,~] = midiid;  
[controlNumber3,~] = midiid;  
[controlNumber4,~] = midiid;
```

```
controlNumbers = [controlNumber1,controlNumber3;...  
                  controlNumber2,controlNumber4]
```

```
Move the control you wish to identify; type ^C to abort.  
Waiting for control message... done  
Move the control you wish to identify; type ^C to abort.  
Waiting for control message... done  
Move the control you wish to identify; type ^C to abort.  
Waiting for control message... done  
Move the control you wish to identify; type ^C to abort.  
Waiting for control message... done
```

```
controlNumbers =  
  
    1081    1085  
    1082    1087
```

Create a `midicontrols` object that listens to your specified controls. Specify an initial value for all controls.

```
initialValue = 0.5;  
midicontrolsObject = midicontrols(controlNumbers,initialValue);
```

Move one of your specified controls, and then read the latest value of all your specified controls.

```
midiread(midicontrolsObject)
```

```
ans =  
  
    0.0873    0.5000
```

```
0.5000    0.5000
```

Specify Controls Numbers, Initial Value, and Output Mode

Determine the control numbers of two different controls on your MIDI device.

```
[controlNumber1,~] = midiid;
[controlNumber2,~] = midiid;
```

```
controlNumbers = [controlNumber1,controlNumber2];
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
```

Create a `midicontrols` object that listens to your specified controls. Specify an initial value for all controls.

```
initialValue = 12;
midicontrolsObject = midicontrols(controlNumbers,initialValue,'OutputMode','rawmidi');
```

Move one of your specified controls, and then read the latest value of all your specified controls.

```
midiread(midicontrolsObject)
```

```
ans =
```

```
63    12
```

Set the Default MIDI Device

Assume that your MIDI device is a Behringer BCF2000. Enter this syntax at the MATLAB command line:

```
setpref midi DefaultDevice BCF2000
```

This preference persists across MATLAB sessions. You do not need to set it again unless you want to change your default device.

Specify Control Numbers and MIDI Device Name

Assume that your MIDI device is a Behringer BCF2000 and has a control with identification number 1001. Create a `midicontrols` object, which listens to control number 1001 on your Behringer BCF2000 device.

```
midicontrolsObject = midicontrols(1001, 'MIDIDevice', 'BCF2000');
```

Input Arguments

controlNumbers — MIDI device control numbers

integer | array of integers

MIDI device control numbers, specified as an integer or array of integers. Use `mididid` to interactively identify the control numbers of your device. See “MIDI Device Control Numbers” on page 2-96 for an advanced explanation of how `controlNumbers` are determined.

If you specify `controlNumbers` as an empty vector, `[]`, then the `midicontrols` object responds to any control on your MIDI device.

Example: 1081

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

initialValues — Initial values of MIDI controls

0 (default) | scalar | array the same size as `controlNumbers`

Initial values of MIDI controls, specified as a scalar or an array the same size as `controlNumbers`. If you specify `initialValues` as a scalar, all controls specified by `controlNumbers` are assigned that value.

The value associated with your MIDI controls cannot be determined until you move a MIDI control. If you specify an initial value associated with your MIDI control, the initial value is returned by the `midiread` function until the MIDI control is moved.

- If `OutputMode` is specified as `'normalized'`, then initial values must be in the range `[0,1]`. Actual initial values are quantized and can be slightly different from initial values specified when your `midicontrols` object is created.
- If `OutputMode` is specified as `'rawmidi'`, then initial values must be integers in the range `[0,127]`

Example: 0.3

Example: `[0,0.3,0.6]`

Example: 5

Example: [5;15;20]

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

deviceName — MIDI device name

string

MIDI device name, assigned by the device manufacturer or host operating system, specified as a string. The specified **deviceName** can be a substring of the exact name of your device. If you do not specify **deviceName**, the default MIDI device is used. See “Set the Default MIDI Device” on page 2-93 for an example of specifying a default MIDI device.

If you do not set a default MIDI device, the host operating system chooses the default device in an unspecified way. As a best practice, use `midid` to identify the name of the device you want.

Example: 'MIDIdevice', 'BCF2000 MIDI 1'

Data Types: char

mode — Output mode for MIDI control value

'normalized' (default) | 'rawmidi'

Output mode for MIDI control value, specified as 'normalized' or 'rawmidi'.

- 'normalized' — Values of your MIDI control are normalized. If your `midicontrols` object is called by `midiread`, then values in the range [0,1] are returned.
- 'rawmidi' — Values of your MIDI control are not normalized. If your `midicontrols` object is called by `midiread`, then integer values in the range [0,127] are returned.

Example: 'OutputMode', 'normalized'

Example: 'OutputMode', 'rawmidi'

Data Types: char

Output Arguments

midicontrolsObject — Object that listens to the controls on a MIDI device
object

Object that listens to the controls on a MIDI device.

More About

MIDI Device Control Numbers

MATLAB defines *MIDI device control numbers* as $(MIDI\ Channel\ Number) \times 1000 + (MIDI\ Controller\ Number)$.

- *MIDI Channel Number* is the transmission channel that your device uses to send messages. This value is in the range 1–16.
- *MIDI Controller Number* is a number assigned to an individual control on your MIDI device. This value is in the range 1–127.

Your MIDI device determines the values of *MIDI Channel Number* and *MIDI Controller Number*.

- “Musical Instrument Digital Interface (MIDI)”

See Also

`configureMIDI` | `disconnectMIDI` | `getMIDIConnections` | `midicallback` | `midiid` | `midiread` | `midisync` | `setpref`

midiid

Interactively identify MIDI control

Syntax

```
[controlNumber,deviceName] = midiid
```

Description

`[controlNumber,deviceName] = midiid` returns the control number and device name of the MIDI control you move. Call the function and then move the control you want to identify. The function detects which control you move and returns the control number and device name that specify that control.

Examples

Identify Control Number and Device Name

Call `midiid` and then move the control you want to identify on the MIDI device you want to identify.

```
[ctl,dev] = midiid;  
Move the control you wish to identify; type ^C to abort.  
Waiting for control message...
```

```
ctl =  
1002  
dev =  
nanoKONTROL
```

Output Arguments

controlNumber — MIDI device control number
integer

MIDI device control number, specified as an integer. The device manufacturer assigns the value to the control for identification purposes.

deviceName — MIDI device name

string

MIDI device name assigned by the device manufacturer or host operating system, specified as a string.

See Also

`configureMIDI` | `disconnectMIDI` | `getMIDIConnections` | `midicallback` | `midicontrols` | `midiread` | `midisync` | `setpref`

midiread

Return most recent value of MIDI controls

Syntax

```
controlValues = midiread(midicontrolsObject)
```

Description

`controlValues = midiread(midicontrolsObject)` returns the most recent value of the MIDI controls associated with the specified `midicontrolsObject`. To create this object, use the `midicontrols` function.

Examples

Read Control Values of MIDI Device

```
midicontrolsObject = midicontrols;  
controlValue = midiread(midicontrolsObject);
```

Read Multiple Control Values of MIDI Device

Identify two MIDI controls on your MIDI device.

```
[controlOne,~] = midiid  
[controlTwo,~] = midiid
```

```
Move the control you wish to identify; type ^C to abort.  
Waiting for control message... done
```

```
controlOne =  
  
1081
```

```
Move the control you wish to identify; type ^C to abort.  
Waiting for control message... done
```

```
controlTwo =  
    1082
```

Create a MIDI controls object that listens to both controls you identified.

```
controlNumbers = [controlOne,controlTwo];  
midicontrolsObject = midicontrols(controlNumbers);
```

Move your specified MIDI controls and return their values. The values are returned as a vector that corresponds to your control numbers vector, `controlNumbers`.

```
tic  
while toc < 5  
    controlValues = midiread(midicontrolsObject)  
end  
  
controlValues =  
    0.0397    0.0556
```

Read Control Values in an Audio Stream Loop

Use `midid` to identify the name of your MIDI device and a specified control. Move the MIDI control you want to identify.

```
[controlNumber, deviceName] = midid;
```

```
Move the control you wish to identify; type ^C to abort.  
Waiting for control message... done
```

Create a MIDI controls object. The value associated with your MIDI controls object cannot be determined until you move the MIDI control. Specify an initial value associated with your MIDI control. The `midiread` function returns the initial value until the MIDI control is moved.

```
initialControlValue = 1;  
midicontrolsObject = midicontrols(controlNumber,initialControlValue);
```

Create a `dsp.AudioFileReader` System object with default settings. Create an `audioDeviceWriter` System object and specify the sample rate.

```
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3');  
deviceWriter = audioDeviceWriter(...
```

```
'SampleRate',fileReader.SampleRate);
```

In an audio stream loop, read an audio signal frame from the file, apply gain specified by the control on your MIDI device, and then write the frame to your audio output device. By default, the control value returned by `midiread` is normalized.

```
while ~isDone(fileReader)
    audioData = step(fileReader);

    controlValue = midiread(midicontrolsObject);

    gain = controlValue*2;
    audioDataWithGain = audioData*gain;

    play(deviceWriter,audioDataWithGain);
end
```

Close the input file and release your output device.

```
release(fileReader);
release(deviceWriter);
```

Input Arguments

midicontrolsObject — Object that listens to the controls on a MIDI device

object

Object that listens to the controls on a MIDI device, specified as an object created by `midicontrols`.

Output Arguments

controlValues — Most recent values of MIDI controls

[0, 1] (default) | integer values in the range [0, 127]

Most recent values of MIDI controls, returned as normalized values in the range [0, 1], or as integer values in the range [0, 127]. The output values depend on the `OutputMode` specified when your `midicontrols` object is created.

- If `OutputMode` was specified as 'normalized', then `midiread` returns values in the range [0, 1]. The default `OutputMode` is 'normalized'.

- If `OutputMode` was specified as `'rawmidi'`, then `midiread` returns integer values in the range `[0,127]`, and no quantization is required.

More About

- “Musical Instrument Digital Interface (MIDI)”

See Also

`configureMIDI` | `disconnectMIDI` | `getMIDIConnections` | `midicallback` | `midicontrols` | `midiid` | `midisync` | `setpref`

midisync

Send values to MIDI controls for synchronization

Syntax

```
midisync(midicontrolsObject)
midisync(midicontrolsObject,controlValues)
```

Description

`midisync(midicontrolsObject)` sends the initial values of controls to your MIDI device, as specified by your MIDI controls object. To create this object, use the `midicontrols` function. If your MIDI device can receive and respond to messages, it adjusts its controls as specified.

Note: Many MIDI devices are not bidirectional. Calling `midisync` with a unidirectional device has no effect. `midisync` cannot tell whether a value is successfully sent to a device or even whether the device is bidirectional. If sending a value fails, no errors or warnings are generated.

`midisync(midicontrolsObject,controlValues)` sends `controlValues` to the MIDI controls associated with the specified `midicontrolsObject`.

Examples

Synchronize MIDI Control to Initial Value

Use `midiid` to identify a control on your default MIDI device.

```
[controlNumber,~] = midiid;
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
```

Create a MIDI controls object. Specify an initial value for your control. Call `midisync` to set the specified control on your device to the initial value.

```
initialValue = 0.5;
midicontrolsObject = midicontrols(controlNumber,initialValue);
midisync(midicontrolsObject);
```

Synchronize MIDI Control to Specified Value

Use `midiid` to identify three controls on your default MIDI device.

```
[controlNumber1,~] = midiid;
[controlNumber2,~] = midiid;
[controlNumber3,~] = midiid;
controlNumbers = [controlNumber1,controlNumber2,controlNumber3];
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
```

Create a MIDI controls object. Specify initial values for your controls. Call `midisync` to set the specified control on your device to the initial value.

```
controlValues = [0,0,1];
midicontrolsObject = midicontrols(controlNumbers,controlValues);
midisync(midicontrolsObject);
```

Create a loop that updates your control values and synchronizes those values to the physical controls on your device.

```
for i = 1:100
    controlValues = controlValues + [0.006,0.008,-0.008];
    midisync(midicontrolsObject,controlValues);
    pause(0.1)
end
```

Create UI Slider and Synchronize with MIDI Control

Define this function and save it to your current folder.

```
function trivialmidigui(controlNumber,deviceName)
```

```

slider = uicontrol('Style','slider');
mc = midicontrols(controlNumber,'MIDIDevice',deviceName);
midisync(mc);
set(slider,'Callback',@slidercb);
midicallback(mc, @mccb);

function slidercb(slider,~)
    val = get(slider,'Value');
    midisync(mc, val);
    disp(val);
end

function mccb(mc)
    val = midiread(mc);
    set(slider,'Value',val);
    disp(val);
end

end

```

Use `midiid` to identify a control number and device name. Call the function you created, specifying the control number and device name as inputs.

```

[controlNumber,deviceName] = midiid;
trivialmidigui(controlNumber,deviceName)

```

The slider on the user interface is synchronized with the specified control on your device. Move one to see the other respond.

Input Arguments

midicontrolsObject — Object that listens to the controls on a MIDI device

object

Object that listens to the controls on a MIDI device, specified as an object created by `midicontrols`.

controlValues — Values sent to MIDI device

initial values specified by `midicontrolsObject` (default) | scalar | array

Values sent to MIDI device, specified as a scalar or an array the same size as `controlNumbers` of the associated `midicontrols` object. If you do not specify

`controlValues`, the default value is the `initialValues` of the associated `midicontrols` object.

The possible range for `controlValues` depends on the `OutputMode` of the associated `midicontrols` object.

- If `OutputMode` is specified as `'normalized'`, then `controlValues` must consist of values in the range `[0,1]`. The default `OutputMode` is `'normalized'`.
- If `OutputMode` is specified as `'rawmidi'`, then `controlValues` must consist of integer values in the range `[0,127]`.

Example: `0.3`

Example: `[0,0.3,0.6]`

Example: `5`

Example: `[5;15;20]`

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

More About

- “Musical Instrument Digital Interface (MIDI)”

See Also

`configureMIDI` | `disconnectMIDI` | `getMIDIConnections` | `midicallback` | `midicontrols` | `midiid` | `midiread` | `setpref`

validateAudioPlugin

Test MATLAB source code for audio plugin

Syntax

```
validateAudioPlugin pluginClass
validateAudioPlugin options pluginClass
```

Description

`validateAudioPlugin pluginClass` generates and runs a “Test Bench Procedure” on page 2-109 that exercises your audio plugin class.

`validateAudioPlugin options pluginClass` specifies options to modify the default “Test Bench Procedure” on page 2-109.

If you use the `-keeptestbench` option, or if an error occurs during validation, the test bench files are saved to your current folder.

Output File Name	Output File Type	Output File Name With File Extension
testbench_ <i>myClassName</i>	MATLAB	testbench_ <i>myClassName</i> .m
testbench_ <i>myClassName</i>	MEX	testbench_ <i>myClassName</i> _mex.mexw64 The MEX-file extension depends on your operating system.

Examples

Validate Audio Plugin

```
validateAudioPlugin audiopluginexample.Echo
```

```
Checking plug-in class 'audiopluginexample.Echo'... passed.
Generating testbench file 'testbench_Echo.m'... done.
```

```
Running testbench... passed.  
Generating mex file 'testbench_Echo_mex.mexw64'... done.  
Running mex testbench... passed.  
Deleting testbench.  
Ready to generate audio plug-in.
```

Skip MEX Version of Test Bench

```
validateAudioPlugin -nomex audiopluginexample.Echo
```

```
Checking plug-in class 'audiopluginexample.Echo'... passed.  
Generating testbench file 'testbench_Echo.m'... done.  
Running testbench... passed.  
Skipping mex.  
Deleting testbench.
```

Keep Test Benches After Validation

```
validateAudioPlugin -keeptestbench audiopluginexample.Echo
```

```
Checking plug-in class 'audiopluginexample.Echo'... passed.  
Generating testbench file 'testbench_Echo.m'... done.  
Running testbench... passed.  
Generating mex file 'testbench_Echo_mex.mexw64'... done.  
Running mex testbench... passed.  
Keeping testbench.  
Ready to generate audio plug-in.
```

Two test benches are saved to your current folder:

- testbench_Echo.m
- testbench_Echo_mex.mexw64

Skip MEX Version and Keep Test Bench

```
validateAudioPlugin -keeptestbench -nomex audiopluginexample.Echo
```

```
Checking plug-in class 'audiopluginexample.Echo'... passed.  
Generating testbench file 'testbench_Echo.m'... done.  
Running testbench... passed.  
Skipping mex.  
Keeping testbench.
```

One test bench is saved to your current folder:

- testbench_Echo.m

Input Arguments

options — Options to modify test bench procedure

-nomex | -keeptestbench

Options to modify test bench procedure, specified as `-nomex` or `-keeptestbench`. Options can be specified together or separately, and in any order.

- `-nomex` — `validateAudioPlugin` does not generate and run a MEX version of the test bench file. This option significantly reduces run time of the test bench procedure.
- `-keeptestbench` — `validateAudioPlugin` saves the generated test benches to the current folder.

pluginClass — Name of the plugin class to validate

plugin class

Name of the plugin class to validate. The plugin class must derive from either the `audioPlugin` class or the `audioPluginSource` class. The `validateAudioPlugin` function exercises an instance of the specified plugin class.

Limitations

The `valdiateAudioPlugin` function is compatible with Windows and Mac operating systems. It is not compatible with Linux.

More About

Test Bench Procedure

The `valudateAudioPlugin` function uses dynamic testing to find common audio plugin programming mistakes not found by the static checks performed by `generateAudioPlugin`. The function:

- 1 Runs a subset of error checks performed by `generateAudioPlugin`.
- 2 Generates and runs a MATLAB test bench to exercise the class.

- 3 Generates and runs a MEX version of the test bench.
- 4 Removes the generated test benches.

If the plugin class fails testing, step 4 is automatically omitted. To debug your plugin, step through the saved generated test bench.

See Also

Functions

`generateAudioPlugin`

Classes

`audioPlugin` | `audioPluginSource`

Introduced in R2016a

System objects in Audio System Toolbox

audioDeviceReader System object

Record from sound card

Description

The `audioDeviceReader` System object reads audio samples using your computer's audio device. See “Audio Device Reader System Interaction” on page 3-12 for a visualization of how the `audioDeviceReader` acquires data.

To stream data from an audio device:

- 1 Define and set up your audio device reader. See “Construction” on page 3-2.
- 2 Call `step` or `record` to stream data from your audio device.

Construction

`aDR = audioDeviceReader` returns a System object, `aDR`, that reads audio samples using an audio input device in real time.

`aDR = audioDeviceReader(sampleRateValue)` sets the `SampleRate` property to `sampleRateValue`.

`aDR = audioDeviceReader(sampleRateValue,samplesPerFrameValue)` sets the `SamplesPerFrame` property to `samplesPerFrameValue`.

`aDR = audioDeviceReader(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `aDR = audioDeviceReader(12000,'BitDepth','8-bit integer')` creates a System object, `aDR`, with the `SampleRate` property set to 12000 and the `BitDepth` property set to '8-bit integer'.

Properties

Driver — Driver used to access audio device (Windows only)

'DirectSound' (default) | 'ASIO'

Driver used to access your audio device, specified as 'DirectSound' or 'ASIO'. ASIO drivers do not come pre-installed on Windows machines. You must install an ASIO driver outside of MATLAB to use the 'ASIO' driver option.

Note: If `Driver` is specified as 'ASIO', open the ASIO UI outside of MATLAB to set the sound card buffer size to the `SamplesPerFrame` value of your `audioDeviceReader` System object. See your ASIO driver documentation for more information.

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

Device — Device used to acquire audio samples

default audio device (default) | string

Device used to acquire audio samples, specified as a string. Use the `getAudioDevices` method to list available devices.

NumChannels — Number of input channels acquired by audio device

1 (default) | integer

Number of input channels acquired by audio device, specified as an integer. The range of `NumChannels` depends on your audio hardware.

This property is available when you set `ChannelMappingSource` to 'Auto'.

SamplesPerFrame — Frame size read from audio device

1024 (default) | integer

Frame size read from audio device, specified as a positive integer. `SamplesPerFrame` is also the size of your device buffer, and the number of columns of the output matrix returned when calling `step` or `record` on `audioDeviceReader`.

SampleRate — Sample rate used by device to acquire audio data (Hz)

44100 (default) | positive integer

Sample rate used by device to acquire audio data, in Hz, specified as a positive integer. The range of `SampleRate` depends on your audio hardware.

BitDepth — Data type used by device to acquire audio data

'16-bit integer' (default) | '8-bit integer' | '32-bit float' | '24-bit float'

Data type used by device to acquire audio data, specified as a string.

ChannelMappingSource — Source of mapping between channels of input device and columns of output matrix

'Auto' (default) | 'Property'

Source of mapping between the channels of your audio input device and columns of the output matrix, specified as 'Auto' or 'Property'.

- 'Auto' — The default settings determine the mapping between device channels and output matrix. For example, suppose that your audio device has six channels available, and you set `NumChannels` to 6. The output from a call to `step` or `record` is a six-column matrix. Column 1 corresponds with channel 1, column 2 corresponds with channel 2, and so on.
- 'Property' — The `ChannelMapping` property determines mapping between channels of your audio device and columns of the output matrix.

ChannelMapping — Nondefault mapping between channels of input device and columns of output matrix

[1:MaximumInputChannels] (default) | scalar | vector

Nondefault mapping between channels of your audio input device and columns of the output matrix, specified as a vector of valid channel indices. See the “Specify Channel Mapping for `audioDeviceReader`” on page 3-9 example for more information.

This property is available when you set `ChannelMappingSource` to 'Property'.

OutputDataType — Data type of the output

'double' (default) | 'single' | 'int32' | 'int16' | 'uint8'

Data type of the output, specified as a string.

Note: If `OutputDataType` is specified as 'double' or 'single', the audio device reader outputs data in the range $[-1, 1]$. For other data types, the range is $[\min, \max]$ of the specified data type.

Methods

`clone`

Create copy of System object with same property values

getAudioDevices	List available audio input devices
info	Get information about selected device
isLocked	Locked status for input attributes and nontunable properties
record	Stream audio data from input device
release	Enable property values and input characteristics to change
step	Stream audio data from input device

Examples

Read from Microphone and Write to Audio File

Record ten seconds of speech with a microphone and send the output to a .wav file.

Create an `audioDeviceReader` System object™ with default settings. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
deviceReader = audioDeviceReader;
setup(deviceReader);
```

Create a `dsp.AudioFileWriter` System object. Specify the file name and type to write.

```
fileWriter = dsp.AudioFileWriter(...
    'mySpeech.wav',...
    'FileFormat','WAV');
```

Record 10 seconds of speech. In an audio stream loop, read an audio signal frame from the device, and write the audio signal frame to a specified file. The file saves to your current folder.

```
disp('Speak into microphone now. ');
tic;
while toc < 10
    acquiredAudio = record(deviceReader);
    step(fileWriter, acquiredAudio);
end
disp('Recording complete.');
```

Speak into microphone now.

Recording complete.

Release the audio device and close the output file.

```
release(deviceReader);  
release(fileWriter);
```

Reduce Latency Due to Input Device Buffer

Latency due to the input device buffer is the time delay of acquiring one frame of data. In this example, you modify default properties of your `audioDeviceReader` System object™ to reduce latency.

Create an `audioDeviceReader` System object with default settings.

```
deviceReader = audioDeviceReader
```

```
deviceReader =
```

```
System: audioDeviceReader
```

```
Properties:
```

```
          Driver: 'DirectSound'  
          Device: 'Default'  
    NumChannels: 1  
SamplesPerFrame: 1024  
      SampleRate: 44100
```

```
Advanced properties:
```

```
          BitDepth: '16-bit integer'  
ChannelMappingSource: 'Auto'  
          OutputDataType: 'double'
```

Calculate the latency due to your device buffer.

```
fprintf('Latency due to device buffer: %f seconds.\n',...  
        deviceReader.SamplesPerFrame/deviceReader.SampleRate);
```

```
Latency due to device buffer: 0.023220 seconds.
```

Set the `SamplesPerFrame` property of your `audioDeviceReader` System object to 64. Calculate the latency.

```
deviceReader.SamplesPerFrame = 64;
fprintf('Latency due to device buffer: %f seconds.\n',...
    deviceReader.SamplesPerFrame/deviceReader.SampleRate);
```

Latency due to device buffer: 0.001451 seconds.

Set the `SampleRate` property of your `audioDeviceReader` System object to 96,000. Calculate the latency.

```
deviceReader.SampleRate = 96000;
fprintf('Latency due to device buffer: %f seconds.\n',...
    deviceReader.SamplesPerFrame/deviceReader.SampleRate);
```

Latency due to device buffer: 0.000667 seconds.

Determine and Decrease Overrun

Overrun refers to input signal drops, which occur when the audio stream loop does not keep pace with the device. Determine overrun of an audio stream loop, add an artificial computational load to the audio stream loop, and then modify properties of your `audioDeviceReader` System object™ to decrease overrun. Your results depend on your computer.

Create an `audioDeviceReader` System object with `SamplesPerFrame` set to 256 and `SampleRate` set to 44,100. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
deviceReader = audioDeviceReader(...
    'SamplesPerFrame',256,...
    'SampleRate',44100);
setup(deviceReader);
```

Create a `dsp.AudioFileWriter` System object. Specify the file name and type to write.

```
fileWriter = dsp.AudioFileWriter(...
    'mySpeech.wav',...
    'FileFormat','WAV');
```

Record 5 seconds of speech. In an audio stream loop, read an audio signal frame from your device, and write the audio signal frame to a specified file.

```
totalOverrun = 0;
disp('Speak into microphone now.');
```

```
while toc < 5
    [input,numOverrun] = record(deviceReader);
    totalOverrun = totalOverrun + numOverrun;
    step(fileWriter, input);
end
fprintf('Recording complete.\n')
fprintf('Total number of samples overrun: %d.\n',...
    totalOverrun);
fprintf('Total seconds overrun: %d.\n',...
    double(totalOverrun)/double(deviceReader.SampleRate));
```

```
Speak into microphone now.
Recording complete.
Total number of samples overrun: 0.
Total seconds overrun: 0.
```

Release your `audioDeviceReader` and `dsp.AudioDeviceWriter` System objects and zero your counter variable.

```
release(fileWriter);
release(deviceReader);
totalOverrun = 0;
```

Add an artificial computational load to your audio stream loop. The computational load causes the audio stream loop to go slower than the device, which causes acquired samples to be dropped.

```
disp('Speak into microphone now.');
```

```
tic;
while toc < 5
    [input,numOverrun] = record(deviceReader);
    totalOverrun = totalOverrun + numOverrun;
    step(fileWriter, input);
    for m=1:200000
        n = sqrt(m);
    end
end
fprintf('Recording complete.\n')
fprintf('Total number of samples overrun: %d.\n',...
    totalOverrun);
fprintf('Total seconds overrun: %d.\n',...
    double(totalOverrun)/double(deviceReader.SampleRate));
```

```
Speak into microphone now.
Recording complete.
```

```
Total number of samples overrun: 0.
Total seconds overrun: 0.
```

Release your `audioDeviceReader` and `dsp.AudioFileWriter` System objects, and set the `SamplePerFrame` property to 512. The device buffer size increases so that the device now takes longer to acquire a frame of data. Set your counter variable to zero.

```
release(fileWriter);
release(deviceReader);
deviceReader.SamplesPerFrame = 512;
totalOverrun = 0;
```

Calculate the total overrun of the audio stream loop using your modified `SamplesPerFrame` property.

```
disp('Speak into microphone now. ');
tic;
while toc < 5
    [input,numOverrun] = record(deviceReader);
    totalOverrun = totalOverrun + numOverrun;
    step(fileWriter, input);
    for m=1:200000
        n = sqrt(m);
    end
end
fprintf('Recording complete.\n')
fprintf('Total number of samples overrun: %d.\n',...
    totalOverrun);
fprintf('Total seconds overrun: %f.\n',...
    totalOverrun/deviceReader.SampleRate);
```

```
Speak into microphone now.
Recording complete.
Total number of samples overrun: 0.
Total seconds overrun: 0.000000.
```

Specify Channel Mapping for audioDeviceReader

Specify non-default channel mapping for an `audioDeviceReader` System object™. This example is hardware specific. It assumes that your computer has a default audio input device with two available channels.

Create an `audioDeviceReader` System object with default settings.

```
deviceReader = audioDeviceReader;
```

The default number of channels is 1. Call the `step` method or `record` method of your `audioDeviceReader` System object to read one frame of data from your audio device. Verify that the output data matrix has one column.

```
x = step(deviceReader);  
[frameLength,numChannels] = size(x)
```

```
frameLength =  
            1024
```

```
numChannels =  
            1
```

Use `info` to determine the maximum number of input channels available with your specified `Driver` and `Device` configuration.

```
info(deviceReader)
```

```
ans =  
            Driver: 'DirectSound'  
            DeviceName: 'Primary Sound Capture Driver'  
            MaximumInputChannels: 2
```

Set `ChannelMappingSource` to 'Property'. The `audioDeviceReader` System object must be unlocked to change this property.

```
release(deviceReader);  
deviceReader.ChannelMappingSource = 'Property'
```

```
deviceReader =  
            System: audioDeviceReader  
            Properties:  
                Driver: 'DirectSound'  
                Device: 'Default'
```

```
SamplesPerFrame: 1024
SampleRate: 44100
```

```
Advanced properties:
    BitDepth: '16-bit integer'
    ChannelMappingSource: 'Property'
    ChannelMapping: [1 2]
    OutputDataType: 'double'
```

By default, if `ChannelMappingSource` is set to 'Property', all available channels are mapped to the output. Call the `step` method or `record` method of your `audioDeviceReader` System object to read one frame of data from your audio device. Verify that the output data matrix has two columns.

```
x = step(deviceReader);
[frameLength,numChannels] = size(x)
```

```
frameLength =
    1024
```

```
numChannels =
    2
```

Use the `ChannelMapping` property to specify an alternative mapping between channels of your device and columns of the output matrix. Indicate the input channel number at an index corresponding to the output column. To change this property, first unlock the `audioDeviceReader` System object.

```
release(deviceReader);
deviceReader.ChannelMapping = [2,1];
```

If you call `step` or `record`:

- Input channel 1 of your device maps to the second column of your output matrix.
- Input channel 2 of your device maps to the first column of your output matrix.

Acquire a specific channel from your input device.

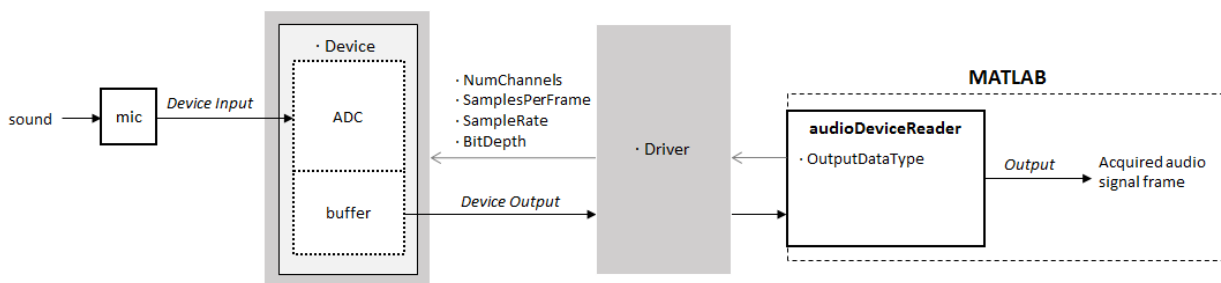
```
deviceReader.ChannelMapping = 2;
```

If you call `step` or `record`, input channel 2 of your device maps to an output vector.

More About

Audio Device Reader System Interaction

The audio device reader specifies the driver, the device and its attributes, and the data type and size output from your System object.



Run an Executable Outside MATLAB

The generated code for the `audioDeviceReader` System object relies on prebuilt dynamic library files that ship with MATLAB. You must account for these extra library files when you run `audioDeviceReader` outside the MATLAB environment. To run a standalone executable generated from code containing the `audioDeviceReader` System object, set your system environment using the commands specific to your platform.

Platform	Command
Mac	<pre>setenv DYLD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/maci64 (csh/tcsh) export DYLD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/maci64 (Bash)</pre>

Platform	Command
Linux	<pre>setenv LD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/glnxa64 (csh/tcsh) export LD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/glnxa64 (Bash)</pre>
Windows	<pre>set PATH = \$MATLABROOT\bin\win64; %PATH%</pre>

See Also

audioDeviceWriter | dsp.AudioFileReader | Audio Device Reader

More About

- “Audio I/O: Buffering, Latency, and Throughput”
- “Real-Time Audio in MATLAB”

Introduced in R2016a

clone

System object: audioDeviceReader

Create copy of System object with same property values

Syntax

```
aDRclone = clone(aDR)
```

Description

`aDRclone = clone(aDR)` creates an audio device reader System object, `aDRclone`, with the same property values as `aDR`. If the original object is locked, then `clone` creates a copy that is also locked. This copy has states initialized to the same values as the original. If the original object is not locked, then `clone` creates a new unlocked object with uninitialized states.

Introduced in R2016a

getAudioDevices

System object: audioDeviceReader

List available audio input devices

Syntax

```
devices = getAudioDevices(aDR)
```

Description

`devices = getAudioDevices(aDR)` returns a cell array listing available audio input devices. The list of available input devices depends on the specified `Driver` property of your `audioDeviceReader` object.

Introduced in R2016a

info

System object: `audioDeviceReader`

Get information about selected device

Syntax

```
aDRInfo = info(aDR)
```

Description

`aDRInfo = info(aDR)` returns a structure containing information about your `audioDeviceReader` System object. The structure contains information about the driver, device, and maximum number of input channels for your `audioDeviceReader` System object.

Introduced in R2016a

isLocked

System object: audioDeviceReader

Locked status for input attributes and nontunable properties

Syntax

L = isLocked(aDR)

Description

L = isLocked(aDR) returns a logical value, L, that indicates whether input attributes and nontunable properties are locked for the audio device reader, aDR.

The aDR object performs an internal initialization the first time you execute `step` or `record`. The initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After the object is locked, the `isLocked` method returns a `true` value.

Introduced in R2016a

record

System object: `audioDeviceReader`

Stream audio data from input device

Syntax

```
[x,numOverrun] = record(aDR)
```

Description

`[x,numOverrun] = record(aDR)` reads one frame of audio samples from the selected audio input device and returns the number of samples by which the audio device reader's queue was overrun since the last call to `record`.

When you call the `record` method of an `audioDeviceReader` System object, the audio device specified by the `Device` property is locked. An audio device can be locked by only one `audioDeviceReader` at a time. Call the `release` method of the `audioDeviceReader` System object to release the audio device.

Note: The System object performs an internal initialization the first time you execute `record`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

release

System object: audioDeviceReader

Enable property values and input characteristics to change

Syntax

```
release(aDR)
```

Description

`release(aDR)` releases system resources, such as memory, from your audio device reader, `aDR`. After you call **release**, all properties and input characteristics of `aDR` can change.

Note: Once you call **release** on a System object, subsequent calls to **setup**, **step**, **record**, **reset**, or **release** do not support code generation.

Introduced in R2016a

step

System object: `audioDeviceReader`

Stream audio data from input device

Syntax

```
[x,numOverrun] = step(aDR)
```

Description

`[x,numOverrun] = step(aDR)` reads one frame of audio samples from the selected audio input device and returns the number of samples by which the audio device reader's queue was overrun since the last call to `step`.

When you call the `step` method of an `audioDeviceReader` System object, the audio device specified by the `Device` property is locked. An audio device can be locked by only one `audioDeviceReader` at a time. Call the `release` method of the `audioDeviceReader` System object to release the audio device.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

audioDeviceWriter System object

Play to sound card

Description

The `audioDeviceWriter` System object writes audio samples to an audio output device. See “Audio Device Writer System Interaction” on page 3-30 for a visualization of how the `audioDeviceWriter` System object plays audio samples.

To stream data to an audio device:

- 1 Define and set up your audio device writer. See “Construction” on page 3-21.
- 2 Call `step` or `play` to stream data to an audio device.

Construction

`aDW = audioDeviceWriter` returns a System object, `aDW` that writes audio samples to an audio output device in real time.

`aDW = audioDeviceWriter(sampleRateValue)` sets the `SampleRate` property to `sampleRateValue`.

`aDW = audioDeviceWriter(Name, Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `aDW = audioDeviceWriter(12000, 'BitDepth', '8-bit integer')` creates a System object, `aDW`, with the `SampleRate` property set to 12000 and the `BitDepth` property set to '8-bit integer'.

Properties

Driver — Driver used to access audio device (Windows only)

'DirectSound' (default) | 'ASIO'

Driver used to access your audio device, specified as 'DirectSound' or 'ASIO'. ASIO drivers do not come pre-installed on Windows machines. Install an ASIO driver outside of MATLAB to use the 'ASIO' driver option.

Note: If `Driver` is specified as 'ASIO', open the ASIO UI outside of MATLAB to set the sound card buffer size to the `BufferSize` value of your `audioDeviceWriter` System object. See the documentation of your ASIO driver for more information.

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

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

Device — Device used to play audio samples

default audio device (default) | string

Device used to play audio samples, specified as a string. Use the `getAudioDevices` method to list available devices.

SampleRate — Sample rate of signal sent to audio device (Hz)

44100 (default) | positive integer

Sample rate of signal sent to audio device, in Hz, specified as a positive integer. The range of `SampleRate` depends on your audio hardware.

BitDepth — Data type used by the device

'16-bit integer' (default) | '8-bit integer' | '24-bit integer' | '32-bit float'

Data type used by the device, specified as a string. Before performing digital-to-analog conversion, the input data is cast to a data type specified by `BitDepth`.

To specify a nondefault `BitDepth`, you must install Audio System Toolbox. If the toolbox is not installed, specifying a nondefault `BitDepth` returns an error.

SupportVariableSizeInput — Option to support variable frame size

false (default) | true

Option to support variable frame size, specified as `true` or `false`.

- `false` — If the `audioDeviceWriter` object is locked, the input must have the same frame size at each call to `step` or `play`. The buffer size of your audio device is the same as the input frame size.
- `true` — If the `audioDeviceWriter` object is locked, the input frame size can change at each call to `step` or `play`. The buffer size of your audio device is specified through the `BufferSize` property.

BufferSize — Buffer size of audio device

4096 (default) | positive integer

Buffer size of audio device, specified as a positive integer.

Note: If `Driver` is specified as `'ASIO'`, open the ASIO UI to set the sound card buffer size to the `BufferSize` value of your `audioDeviceWriter` System object.

This property is available when you set `SupportVariableSizeInput` to `true`.

ChannelMappingSource — Source of mapping between columns of input matrix and channels of output device

'Auto' (default) | 'Property'

Source of mapping between columns of input matrix and channels of audio output device, specified as `'Auto'` or `'Property'`.

- `'Auto'` — Default settings determine the mapping between columns of input matrix and channels of audio output device. For example, suppose your input is a matrix with four columns, and your audio device has four channels available. Column 1 of your input data writes to channel 1 of your device, column 2 of your input data writes to channel 2 of your device, and so on.
- `'Property'` — The `ChannelMapping` property determines the mapping between columns of input matrix and channels of audio output device.

ChannelMapping — Nondefault mapping between columns of input matrix and channels of output device

[1:MaximumOutputChannels] (default) | scalar | vector

Nondefault mapping between columns of input matrix and channels of output device, specified as a scalar or vector of valid channel indices. See the “Specify Channel Mapping for `audioDeviceWriter`” on page 3-28 example for more information.

This property is available when you set `ChannelMappingSource` to `'Property'`.

To selectively map between columns of the input matrix and your sound card's output channels, you must install Audio System Toolbox. If the toolbox is not installed, specifying a nondefault `ChannelMapping` returns an error.

Methods

<code>clone</code>	Create copy of System object with same property values
<code>getAudioDevices</code>	List available audio input devices
<code>info</code>	Get information about selected device
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>play</code>	Stream audio data to output device
<code>release</code>	Enable property values and input characteristics to change
<code>step</code>	Stream audio data to output device

Examples

Read from File and Write to Audio Device

Read an MP3 audio file and play it through your default audio output device.

Create a `dsp.AudioFileReader` System object™ with default settings. Use the `audioinfo` function to return a structure containing information about the audio file.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');  
fileInfo = audioinfo('speech_dft.mp3');
```

Create an `audioDeviceWriter` System object and specify the sample rate. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
deviceWriter = audioDeviceWriter(...  
    'SampleRate',fileInfo.SampleRate);  
setup(deviceWriter,...
```

```
zeros(fileReader.SamplesPerFrame,fileInfo.NumChannels));
```

In an audio stream loop, read an audio signal frame from the file, and write the frame to your device.

```
while ~isDone(fileReader)
    audioData = step(fileReader);
    play(deviceWriter,audioData);
end
```

Close the input file and release the device.

```
release(fileReader);
release(deviceWriter);
```

Reduce Latency due to Output Device Buffer

Modify default properties of your `audioDeviceWriter` System object™ to reduce latency due to device buffer size.

Create a `dsp.AudioFileReader` System object to read an audio file with default settings.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
```

Create an `audioDeviceWriter` System object and specify the sample rate to match that of the audio file reader.

```
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
```

Calculate the latency due to your device buffer, in seconds.

```
bufferLatency = fileReader.SamplesPerFrame/deviceWriter.SampleRate
```

```
bufferLatency =
```

```
0.0464
```

Set the `SamplesPerFrame` property of your `dsp.AudioFileReader` System object to 256. Calculate the buffer latency in seconds.

```
fileReader.SamplesPerFrame = 256;
bufferLatency = fileReader.SamplesPerFrame/deviceWriter.SampleRate
```

```
bufferLatency =  
    0.0116
```

Determine and Decrease Underrun

Underrun refers to output signal silence, which occurs when the audio stream loop does not keep pace with the output device. Determine the underrun of an audio stream loop, add artificial computational load to the audio stream loop, and then modify properties of your `audioDeviceWriter` System object (™) to decrease underrun. Your results depend on your computer.

Create a `dsp.AudioFileReader` System object, and specify the file to read. Use the `audioinfo` function to return a structure containing information about the audio file.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');  
fileInfo = audioinfo('speech_dft.mp3');
```

Create an `audioDeviceWriter` System object. Use the `SampleRate` of the file reader as the `SampleRate` of the device writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
deviceWriter = audioDeviceWriter(...  
    'SampleRate',fileReader.SampleRate);  
setup(deviceWriter,...  
    zeros(fileReader.SamplesPerFrame,fileInfo.NumChannels));
```

Run your audio stream loop with input from file and output to device. Print the total samples underrun and the underrun in seconds.

```
totalUnderrun = 0;  
while ~isDone(fileReader)  
    input = step(fileReader);  
    numUnderrun = play(deviceWriter,input);  
    totalUnderrun = totalUnderrun + numUnderrun;  
end  
fprintf('Total samples underrun: %d.\n',...  
    totalUnderrun);  
fprintf('Total seconds underrun: %d.\n',...  
    double(totalUnderrun)/double(deviceWriter.SampleRate));
```

```
Total samples underrun: 0.
```

Total seconds underrun: 0.

Release your `dsp.AudioFileReader` and `audioDeviceWriter` System objects and set your counter variable to zero.

```
release(fileReader);
release(deviceWriter);
totalUnderrun = 0;
```

Add an artificial computational load to your audio stream loop. The computational load causes the audio stream loop to go slower than the device, which results in periods of silence in the output audio signal.

```
while ~isDone(fileReader)
    input = step(fileReader);
    numUnderrun = play(deviceWriter,input);
    totalUnderrun = totalUnderrun + numUnderrun;
    for m = 1:1500000
        n = sqrt(m);
    end
end
fprintf('Total samples underrun: %d.\n',...
        totalUnderrun);
fprintf('Total seconds underrun: %d.\n',...
        double(totalUnderrun)/double(deviceWriter.SampleRate));
```

Total samples underrun: 0.
Total seconds underrun: 0.

Release your `audioDeviceReader` and `dsp.AudioFileWriter` and set the counter variable to zero.

```
release(fileReader);
release(deviceWriter);
totalUnderrun = 0;
```

Set the frame size of your audio stream loop to 2048. Because the `SupportVariableSizeInput` property of your `audioDeviceWriter` System object is set to `false`, the buffer size of your audio device is the same size as the input frame size. Increasing your device buffer size decreases underrun.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
fileReader.SamplesPerFrame = 2048;
fileInfo = audioinfo('speech_dft.mp3');
```

```
deviceWriter = audioDeviceWriter(...  
    'SampleRate',fileReader.SampleRate);  
setup(deviceWriter,...  
    zeros(fileReader.SamplesPerFrame,fileInfo.NumChannels));
```

Calculate the total underrun.

```
while ~isDone(fileReader)  
    input = step(fileReader);  
    numUnderrun = play(deviceWriter,input);  
    totalUnderrun = totalUnderrun + numUnderrun;  
    for m = 1:1500000  
        n = sqrt(m);  
    end  
end  
fprintf('Total samples underrun: %d.\n',...  
    totalUnderrun);  
fprintf('Total seconds underrun: %d.\n',...  
    double(totalUnderrun)/double(deviceWriter.SampleRate));
```

```
Total samples underrun: 0.
```

```
Total seconds underrun: 0.
```

The increased frame size reduces the total underrun of your audio stream loop. However, increasing the frame size also increases latency. Other approaches to reduce underrun include:

- Increase the buffer size independent of input frame size. To increase buffer size independent of input frame size, you must first set `SupportVariableSizeInput` to `true`. This approach also increases latency.
- Decrease the sample rate. Decreasing the sample rate reduces both latency and underrun at the cost of signal resolution.
- Choose an optimal driver and device for your system.

Specify Channel Mapping for audioDeviceWriter

Specify non-default channel mapping for an `audioDeviceWriter` System object™. This example is hardware specific. It assumes that your computer has a default audio output device with two available channels.

Create an `audioDeviceWriter` System object™ with default settings.

```
deviceWriter = audioDeviceWriter;
```


By default, the `audioDeviceWriter` System object writes the max number of channels available, corresponding to the columns of the input matrix. Use `info` to get the max number of channels of your device.

```
info(deviceWriter)
```

```
ans =
```

```

                Driver: 'DirectSound'
            DeviceName: 'Primary Sound Driver'
MaximumOutputChannels: 2

```

If `deviceWriter` is called with one column of data, two channels are written to your audio output device. Both channels correspond to the one column of data.

Use the `audioOscillator` System object to output a tone to your `audioDeviceWriter` System object. Your object, `sineGenerator`, returns a vector when called by `step`.

```
sineGenerator = audioOscillator;
```

Write the sine tone to your audio device. If you are using headphones, you can hear the tone from both channels.

```
count = 0;
while count < 500
    sine = step(sineGenerator);
    play(deviceWriter,sine);
    count = count + 1;
end
```

If your `audioDeviceWriter` System object is called with two columns of data, two channels are written to your audio output device. The first column corresponds to channel 1 of your audio output device, and the second column corresponds to channel 2 of your audio output device.

Write a two-column matrix to your audio output device. Column one corresponds to the sine tone and column two corresponds to a static signal. If you are using headphones, you can hear the tone from one speaker and the static from the other speaker.

```
count = 0;
while count < 500
    sine = step(sineGenerator);
    static = randn(length(sine),1);
```

```
play(deviceWriter,[sine,static]);  
count = count + 1;  
end
```

Specify alternative mappings between channels of your device and columns of the output matrix by indicating the output channel number at an index corresponding to the input column. Set `ChannelMappingSource` to 'Property'. Indicate that the first column of your input data writes to channel 2 of your output device, and that the second column of your input data writes to channel 1 of your output device. To modify the channel mapping, you must first unlock the `audioDeviceReader` System object.

```
release(deviceWriter);  
deviceWriter.ChannelMappingSource = 'Property';  
deviceWriter.ChannelMapping = [2,1];
```

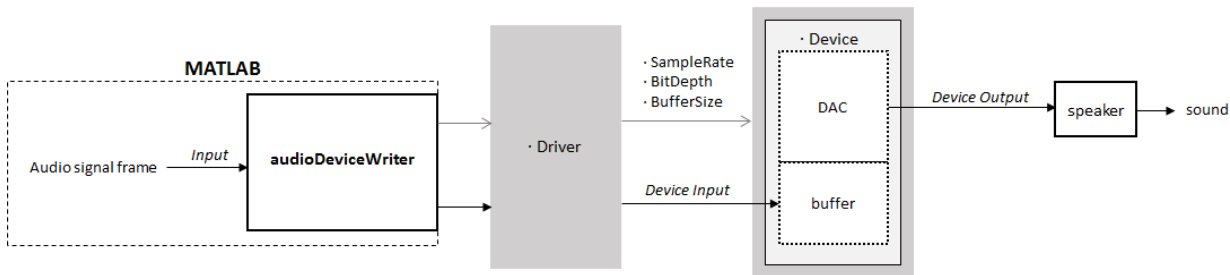
Play your audio signals with reversed mapping. If you are using headphones, notice that the tone and static have switched speakers.

```
count = 0;  
while count < 500  
    sine = step(sineGenerator);  
    static = randn(length(sine),1);  
    play(deviceWriter,[sine,static]);  
    count = count + 1;  
end
```

More About

Audio Device Writer System Interaction

Properties of the audio device writer specify the driver, the device, and device attributes such as sample rate, bit depth, and buffer size.



See “Audio I/O: Buffering, Latency, and Throughput” for a detailed explanation of the audio device writer data flow.

Run an Executable Outside MATLAB

The generated code for the `audioDeviceWriter` System object relies on prebuilt dynamic library files that ship with MATLAB. You must account for these extra library files when you run `audioDeviceWriter` outside the MATLAB environment. To run a standalone executable generated from code containing the `audioDeviceWriter` System object, set your system environment using the commands specific to your platform.

Platform	Command
Mac	<pre>setenv DYLD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/maci64 (csh/tcsh) export DYLD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/maci64 (Bash)</pre>
Linux	<pre>setenv LD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/glnxa64 (csh/tcsh) export LD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/glnxa64 (Bash)</pre>
Windows	<pre>set PATH = \$MATLABROOT\bin\win64; %PATH%</pre>

See Also

`audioDeviceReader` | `dsp.AudioFileWriter` | `dsp.AudioFileReader` | `Audio Device Writer`

More About

- “Audio I/O: Buffering, Latency, and Throughput”
- Measure Audio Latency

- “Real-Time Audio in MATLAB”

Introduced in R2016a

clone

System object: audioDeviceWriter

Create copy of System object with same property values

Syntax

```
aDWclone = clone(aDW)
```

Description

`aDWclone = clone(aDW)` creates an audio device writer System object, `aDWclone`, with the same property values as `aDW`. If the original object is locked, then `clone` creates a copy that is also locked. This copy has states initialized to the same values as the original. If the original object is not locked, then `clone` creates a new unlocked object with uninitialized states.

Introduced in R2016a

getAudioDevices

System object: audioDeviceWriter

List available audio input devices

Syntax

```
devices = getAudioDevices(aDW)
```

Description

`devices = getAudioDevices(aDW)` returns a cell array listing available audio output devices. The list of available output devices depends on the specified **Driver** property of your `audioDeviceWriter` object.

Introduced in R2016a

info

System object: audioDeviceWriter

Get information about selected device

Syntax

```
aDWInfo = info(aDW)
```

Description

`aDWInfo = info(aDW)` returns a structure containing information about your `audioDeviceWriter` System object. The structure contains information about the driver, device, and maximum number of input channels for your `audioDeviceWriter` System object.

Introduced in R2016a

isLocked

System object: audioDeviceWriter

Locked status for input attributes and nontunable properties

Syntax

`L = isLocked(aDW)`

Description

`L = isLocked(aDW)` returns a logical value, `L`, that indicates whether input attributes and nontunable properties are locked for the audio device writer, `aDW`.

The `aDW` object performs an internal initialization the first time you execute `step` or `play`. The initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After the object is locked, the `isLocked` method returns a `true` value.

Introduced in R2016a

play

System object: audioDeviceWriter

Stream audio data to output device

Syntax

```
numUnderrun = play(aDW,x)
```

Description

`numUnderrun = play(aDW,x)` writes one frame of audio samples, `x`, to the audio output device specified by the audio device writer System object, `aDW`. The number of samples underrun since the last call to `play` is returned.

If `x` is of data type 'double' or 'single', the audio device writer clips values outside the range $[-1, 1]$. For other data types, the allowed input range is $[\text{min}, \text{max}]$ of the specified data type.

When you call the `play` method of an `audioDeviceWriter` System object, the audio device specified by the `Device` property is locked. An audio device can be locked by only one `audioDeviceWriter` at a time. Call the `release` method of the `audioDeviceWriter` System object to release the audio device.

Note: The System object performs an internal initialization the first time you execute `play`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

release

System object: audioDeviceWriter

Enable property values and input characteristics to change

Syntax

```
release(aDW)
```

Description

`release(aDW)` releases system resources, such as memory, from your audio device writer, `aDW`. After you call `release`, all properties and input characteristics of `aDW` can change.

Note: Once you call `release` on a System object, subsequent calls to `setup`, `step`, `play`, `reset`, or `release` do not support code generation.

Introduced in R2016a

step

System object: audioDeviceWriter

Stream audio data to output device

Syntax

```
numUnderrun = step(aDW,x)
```

Description

`numUnderrun = step(aDW,x)` writes one frame of audio samples, `x`, to the audio output device specified by the audio device writer System object, `aDW`. The number of samples underrun since the last call to `step` is returned.

If `x` is of data type 'double' or 'single', the audio device writer clips values outside the range $[-1, 1]$. For other data types, the allowed input range is $[\min, \max]$ of the specified data type.

When you call the `step` method of an `audioDeviceWriter` System object, the audio device specified by the `Device` property is locked. An audio device can be locked by only one `audioDeviceWriter` at a time. Call the `release` method of the `audioDeviceWriter` System object to release the audio device.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

audioOscillator System object

Generate sine, square, and sawtooth waveforms

Description

The `audioOscillator` System object generates tunable waveforms. Typical uses include the generation of test signals for test benches, and the generation of control signals for audio effects. Properties of the `audioOscillator` System object specify the type of waveform generated.

To generate tunable waveforms:

- 1 Define and set up your audio oscillator. See “Construction” on page 3-40.
- 2 Call step to generate a waveform according to the properties of your `audioOscillator` object. The object has internal memory suited to frame-based processing.

Construction

`osc = audioOscillator` creates an audio oscillator System object, `osc`, with default property values.

`osc = audioOscillator(signalTypeValue)` sets the `SignalType` property to `signalTypeValue`.

`osc = audioOscillator(signalTypeValue, frequencyValue)` sets the `Frequency` property to `frequencyValue`.

`osc = audioOscillator(Name, Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `osc = audioOscillator('SignalType','sine','Frequency',8000,'DCOffset',1)` creates a System object, `osc`, which generates sine waveforms with `Frequency` set to 8000 and `DCOffset` set to 1.

Properties

SignalType — Type of generated waveform

'sine' (default) | 'square' | 'sawtooth'

Type of waveform generated by your `audioOscillator` object, specified as 'sine', 'square', or 'sawtooth'.

The waveforms are generated using the algorithms specified by the `sin`, `square`, and `sawtooth` functions.

This property is not tunable. You cannot change the value of this property when the object is locked.

Frequency — Frequency of generated waveform (Hz)

100 (default) | real scalar | vector of real scalars

Frequency of generated waveform in Hz, specified as a real scalar or vector of real scalars greater than or equal to 0.

- For sine waveforms, specify `Frequency` as a scalar or as a vector of length `NumTones`.
- For square waveforms, specify `Frequency` as a scalar.
- For sawtooth waveforms, specify `Frequency` as a scalar.

This property is tunable. You can change the value of this property even when the object is locked.

Amplitude — Amplitude of generated waveform

1 (default) | real scalar | vector of real scalars

Amplitude of generated waveform, specified as a real scalar or vector of real scalars greater than or equal to 0.

- For sine waveforms, specify `Amplitude` as a vector of length `NumTones`.
- For square waveforms, specify `Amplitude` as a scalar.
- For sawtooth waveforms, specify `Amplitude` as a scalar.

The generated waveform is multiplied by the value specified by `Amplitude` at the output, before DC offset is applied.

This property is tunable. You can change the value of this property even when the object is locked.

PhaseOffset — Normalized phase offset of generated waveform

0 (default) | real scalar | vector of real scalars

Normalized phase offset of generated waveform, specified as a real scalar or vector of real scalars with values in the range 0 to 1. The range is a normalized 2π radians interval.

- For sine waveforms, specify `PhaseOffset` as a vector of length `NumTones`.
- For square waveforms, specify `PhaseOffset` as a scalar.
- For sawtooth waveforms, specify `PhaseOffset` as a scalar.

This property is not tunable. You cannot change the value of this property when the object is locked.

DCOffset — Value added to each element of generated waveform

0 (default) | real scalar | vector of real scalars

Value added to each element of generated waveform, specified as a real scalar or vector of real scalars.

- For sine waveforms, specify `DCOffset` as a vector of length `NumTones`.
- For square waveforms, specify `DCOffset` as a scalar.
- For sawtooth waveforms, specify `DCOffset` as a scalar.

This property is tunable. You can change the value of this property even when the object is locked.

NumTones — Number of pure sine waveform tones

1 (default) | positive integer

Number of pure sine waveform tones summed and then generated by the audio oscillator, specified as a positive integer. This property applies when you set the `SignalType` property to `'sine'`.

Individual tones are generated based on values specified by `Frequency`, `Amplitude`, `PhaseOffset`, and `DCOffset`.

This property is tunable. You can change the value of this property even when the object is locked.

DutyCycle — Square waveform duty cycle

0.5 (default) | scalar in the range 0 to 1

Square waveform duty cycle, specified as a scalar in the range 0 to 1. This property applies when you set the `SignalType` property to 'square'.

Square waveform duty cycle is the percentage of one period in which the waveform is above the median amplitude. A `DutyCycle` of 1 or 0 is equivalent to a DC offset.

This property is tunable. You can change the value of this property even when the object is locked.

Width — Sawtooth width

1 (default) | real positive scalar

Sawtooth width, specified as a scalar in the range 0 to 1. This property applies when you set the `SignalType` property to 'sawtooth'.

Sawtooth width determines the point in a sawtooth waveform period at which the maximum occurs.

This property is tunable. You can change the value of this property even when the object is locked.

SamplesPerFrame — Number of samples per frame

512 (default) | positive integer

Number of samples per frame, specified as a positive integer in the range 1 to 192,000.

This property determines the vector length that the `step` method of your `audioOscillator` object outputs.

This property is tunable. You can change the value of this property even when the object is locked.

SampleRate — Sample rate of generated waveform (Hz)

44100 (default) | positive scalar

Sample rate of generated waveform in Hz, specified as a positive scalar greater than twice the value specified by `Frequency`.

This property is tunable. You can change the value of this property even when the object is locked.

OutputDataType — Data type of generated waveform

'double' (default) | 'single'

Data type of generated waveform, specify as 'double' or 'single'.

This property is not tunable. You cannot change the value of this property when the object is locked.

Methods

clone	Create copy of System object with same property values
configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
isLocked	Locked status for input attributes and nontunable properties
release	Enable property values and input characteristics to change
reset	Reset internal states of System object
step	Generate tunable waveforms

Examples

Generate Variable-Frequency Sine Wave

Use the `audioOscillator` System object™ to generate a variable-frequency sine wave.

Create an audio oscillator to generate a sine wave. Use the default settings.

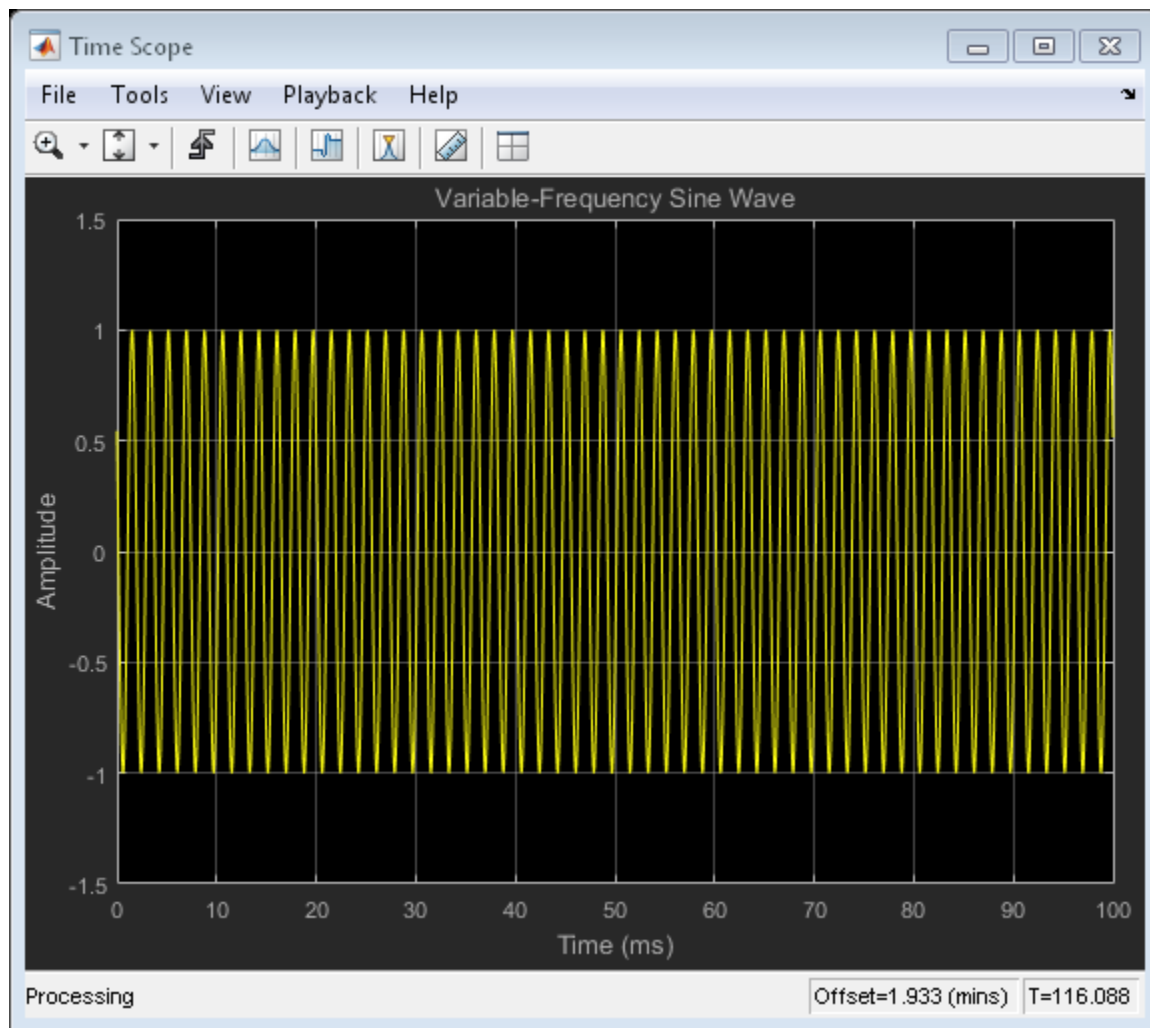
```
osc = audioOscillator;
```


Create a time scope to visualize the variable-frequency sine wave generated by the audio oscillator.

```
scope = dsp.TimeScope(...  
    'SampleRate',osc.SampleRate,...  
    'TimeSpan',0.1,...  
    'YLimits',[-1.5,1.5],...  
    'TimeSpanOverrunAction', 'Scroll', ...  
    'ShowGrid',true,...  
    'Title','Variable-Frequency Sine Wave');
```

Place the audio oscillator in an audio stream loop. Increase the frequency of your sinewave in 50 Hz increments.

```
counter = 0;  
while (counter < 1e4)  
    counter = counter + 1;  
    sineWave = step(osc);  
    step(scope,sineWave);  
    if mod(counter,1000)==0  
        osc.Frequency = osc.Frequency + 50;  
    end  
end
```



See Also

wavetableSynthesizer

Introduced in R2016a

clone

System object: audioOscillator

Create copy of System object with same property values

Syntax

```
oscClone = clone(osc)
```

Description

`oscClone = clone(osc)` creates an audio oscillator System object, `oscClone`, with the same property values as `osc`. If the original object is locked, then `clone` creates a copy that is also locked. This copy has states initialized to the same values as the original. If the original object is not locked, then `clone` creates a new unlocked object with uninitialized states.

Introduced in R2016a

configureMIDI

System object: audioOscillator

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(osc)  
configureMIDI(osc,propName)  
configureMIDI(osc,propName,controlNumber)  
configureMIDI(osc,propName,controlNumber,'DeviceName',deviceName)
```

Description

`configureMIDI(osc)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the audio oscillator System object, `osc`, to MIDI controls of your choice.

`configureMIDI(osc,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

`configureMIDI(osc,propName,controlNumber)` makes the property respond to the MIDI control specified by `controlNumber`.

`configureMIDI(osc,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `audioOscillator` System object maps to MIDI controls with a specified range.

Property	Range	Mapping
Frequency	0.1 Hz to 20 kHz	log
Amplitude	0 to 10	linear
DCoffset	-10 to 10	linear

Property	Range	Mapping
DutyCycle (available when you set SignalType to 'square')	0 to 1	linear
Width (available when you set SignalType to 'sawtooth')	0 to 1	linear

Introduced in R2016a

createAudioPluginClass

System object: audioOscillator

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(osc)  
createAudioPluginClass(osc,pluginName)
```

Description

`createAudioPluginClass(osc)` creates a System object source plugin that implements the functionality of the `audioOscillator` System object, `osc`. The name of the created class is the `audioOscillator` System object variable name followed by 'Plugin', for example, `oscPlugin`. By default, the created class outputs a one-channel (column) matrix.

`createAudioPluginClass(osc,pluginName)` specifies the name of your created System object source plugin class.

Example: `createAudioPluginClass(osc, 'myOscillator')` creates a System object source plugin with class name `myOscillator`.

Each tunable property of the `audioOscillator` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Mapping
Frequency	0.1 Hz to 20 kHz	log
Amplitude	0 to 10	linear
DCOffset	-10 to 10	linear
DutyCycle (available when you set <code>SignalType</code> to 'square')	0 to 1	linear

Property	Plugin Parameter Range	Mapping
Width (available when you set <code>SignalType</code> to 'sawtooth')	0 to 1	linear

Introduced in R2016a

disconnectMIDI

System object: audioOscillator

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(osc)
```

Description

`disconnectMIDI(osc)` disconnects MIDI controls from your audio oscillator, `osc`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: audioOscillator

Get MIDI connection information

Syntax

```
connectionInfo = getMIDIConnections(osc)
```

Description

`connectionInfo = getMIDIConnections(osc)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your audio oscillator, `osc`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `osc` that has established MIDI connections. Each substructure contains the control number, the device name of the corresponding MIDI control, and the property mapping information (mapping rule, minimum value, and maximum value).

Introduced in R2016a

isLocked

System object: audioOscillator

Locked status for input attributes and nontunable properties

Syntax

`L = isLocked(osc)`

Description

`L = isLocked(osc)` returns a logical value, `L`, that indicates whether input attributes and nontunable properties are locked for the audio oscillator, `osc`.

The `osc` object performs an internal initialization the first time you execute `step`. The initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After the object is locked, the `isLocked` method returns a `true` value.

Introduced in R2016a

release

System object: audioOscillator

Enable property values and input characteristics to change

Syntax

```
release(osc)
```

Description

`release(osc)` releases system resources, such as memory, from your audio oscillator, `osc`. After you call `release`, all properties and input characteristics of `OSC` can change.

Note: Once you call `release` on a System object, subsequent calls to `setup`, `step`, `reset`, or `release` do not support code generation.

Introduced in R2016a

reset

System object: audioOscillator

Reset internal states of System object

Syntax

`reset(osc)`

Description

`reset(osc)` resets internal states of the audio oscillator, `osc`, to their initial values.

Introduced in R2016a

step

System object: audioOscillator

Generate tunable waveforms

Syntax

```
y = step(osc)
```

Description

`y = step(osc)` generates a waveform output, `y`. The type of waveform is specified by the algorithm and properties of the System object, `osc`.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

crossoverFilter System object

Audio crossover filter

Description

The `crossoverFilter` System object implements an audio crossover filter, which is used to split an audio signal into two or more frequency bands. Crossover filters are multiband filters whose overall magnitude frequency response is flat.

To implement an audio crossover filter:

- 1 Define and set up your crossover filter. See “Construction” on page 3-58.
- 2 Call step to implement a crossover filter on each channel of the input signal according to the properties of your `crossoverFilter` object. The input must be a real-valued, double-precision or single-precision matrix. The `crossoverFilter` object treats each column of the input as an independent channel.

Construction

`crossFilt = crossoverFilter` creates a System object, `crossFilt`, that implements an audio crossover filter.

`crossFilt = crossoverFilter(numCrossoversValue)` sets the `NumCrossovers` property to `numCrossoversValue`.

`crossFilt = crossoverFilter(numCrossoversValue, crossoverFrequenciesValue)` sets the `CrossoverFrequencies` property to `crossoverFrequenciesValue`.

`crossFilt = crossoverFilter(numCrossoversValue, crossoverFrequenciesValue, crossoverSlopesValue)` sets the `CrossoverSlopes` property to `crossoverSlopesValue`.

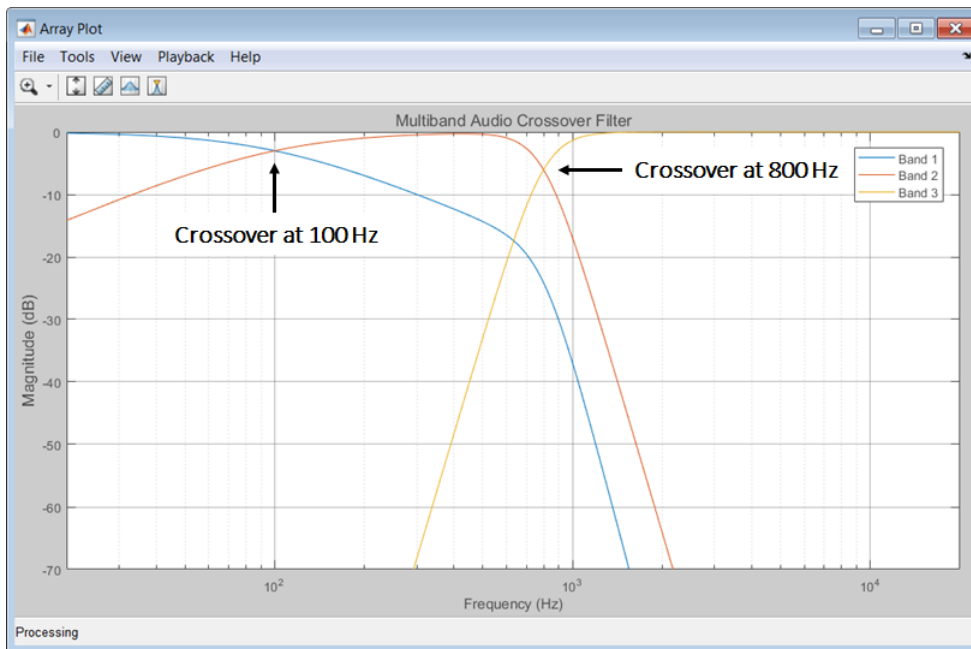
`crossFilt = crossoverFilter(numCrossoversValue, crossoverFrequenciesValue, crossoverSlopesValue, Fs)` sets the `SampleRate` property to `Fs`.

`crossFilt = crossoverFilter(___,Name,Value)` sets each property Name to the specified Value. Unspecified properties have default values.

Example: `crossFilt = crossoverFilter(2,'CrossoverFrequencies',[100,800],'CrossoverSlopes',[6,48])` creates a System object, `crossFilt`, with the `NumCrossovers` property set to 2, the `CrossoverFrequencies` property set to `[100,800]`, and the `CrossoverSlopes` property set to `[6,48]`.

To visualize the crossover bands of the `crossFilt` System object, use the `visualize` method of the object.

```
visualize(crossFilt)
```



Properties

NumCrossovers — Number of magnitude response band crossings
1 (default) | 2 | 3 | 4

Number of magnitude response band crossings, specified as a scalar integer in the range 1 to 4.

The number of bands output when `crossoverFilter` is called by `step` is one more than the `NumCrossovers` value.

Number of magnitude response band crossings	Number of bands output
1	two-band
2	three-band
3	four-band
4	five-band

This property is not tunable. You cannot change the value of this property when the object is locked.

CrossoverFrequencies — Crossover frequencies (Hz)

100 (default) | scalar | vector

Crossover frequencies in Hz, specified as a scalar or vector of real values of length `NumCrossovers`.

Crossover frequencies are the intersections of magnitude response bands of the individual two-band crossover filters used in the multiband crossover filter.

This property is tunable. You can change the value of this property even when the object is locked.

CrossoverSlopes — Crossover slopes (dB/octave)

12 (default) | scalar | vector

Crossover slopes in dB/octave, specified as a scalar or vector of real values in the range [6:6:48]. If a crossover slope is not specified inside the range, it is rounded to the nearest allowed value.

- If `CrossoverSlopes` is a scalar, all two-band component crossovers slopes take that value.
- If `CrossoverSlopes` is a vector of length `NumCrossovers`, the respective two-band component crossover slopes take those values.

Crossover slopes are the slopes of individual bands at the associated crossover frequency, as specified in the two-band component crossover.

This property is tunable. You can change the value of this property even when the object is locked.

SampleRate — Input sample rate (Hz)

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

This property is tunable. You can change the value of this property even when the object is locked.

Methods

clone	Create copy of System object with same property values
configureMIDI	Configure MIDI connections between System object and MIDI controller
cost	Implementation cost of System object
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
isLocked	Locked status for input attributes and nontunable properties
release	Enable property values and input characteristics to change
reset	Reset internal states of System object
step	Implement audio crossover filter
visualize	Visualize magnitude response of System object

Examples

Pass Noise Signal Through Crossover Filter

Use the `crossoverFilter` System object™ to split Gaussian noise into three separate frequency bands.

Create a 5 second noise signal that assumes a 12,000 Hz sample rate.

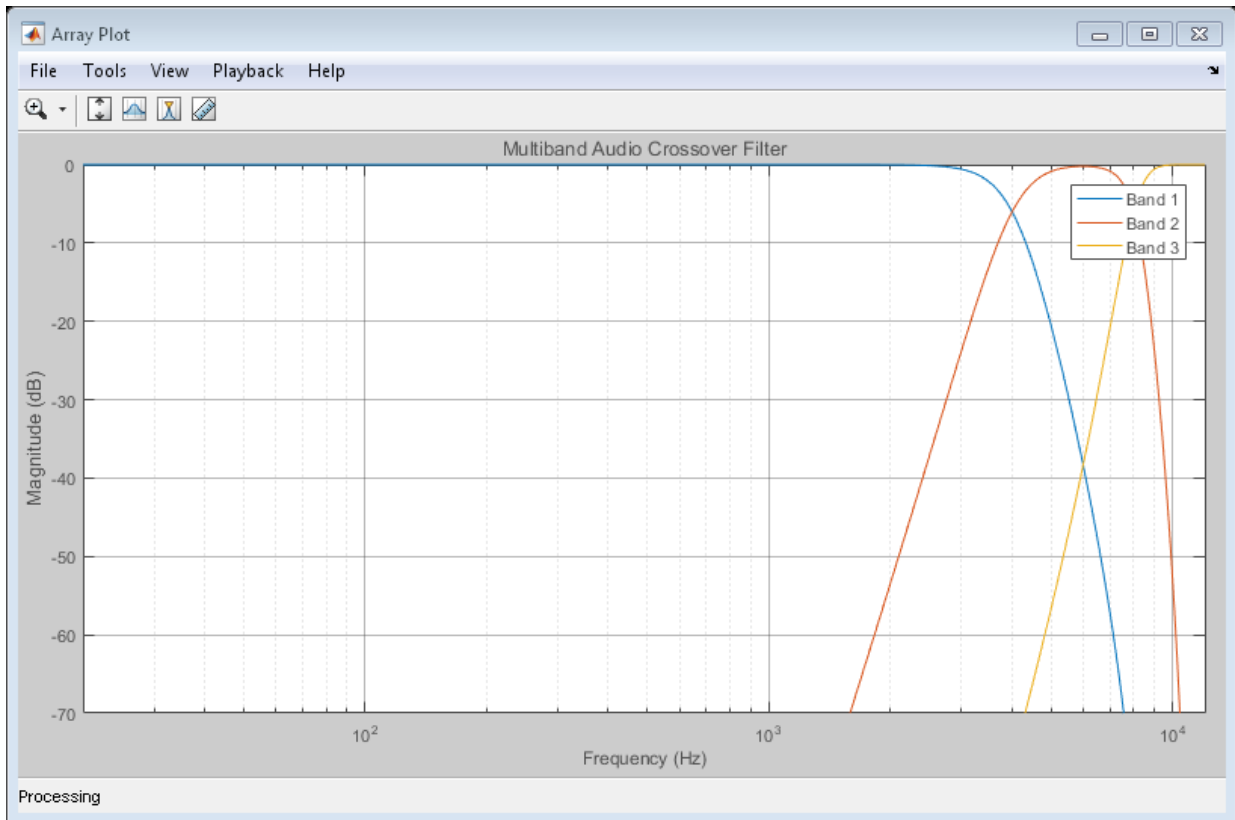
```
noise = randn(12000*5,1);
```

Create a `crossoverFilter` System object with 2 crossovers (3 bands), crossover frequencies at 4 kHz and 8 kHz, a slope of 48 dB/octave, and a sample rate of 24 kHz.

```
crossFilt = crossoverFilter(...  
    'NumCrossovers',2,...  
    'CrossoverFrequencies',[4000,8000],...  
    'CrossoverSlopes',48,...  
    'SampleRate',24000);
```

Visualize the magnitude response of your crossover filter object.

```
visualize(crossFilt);
```



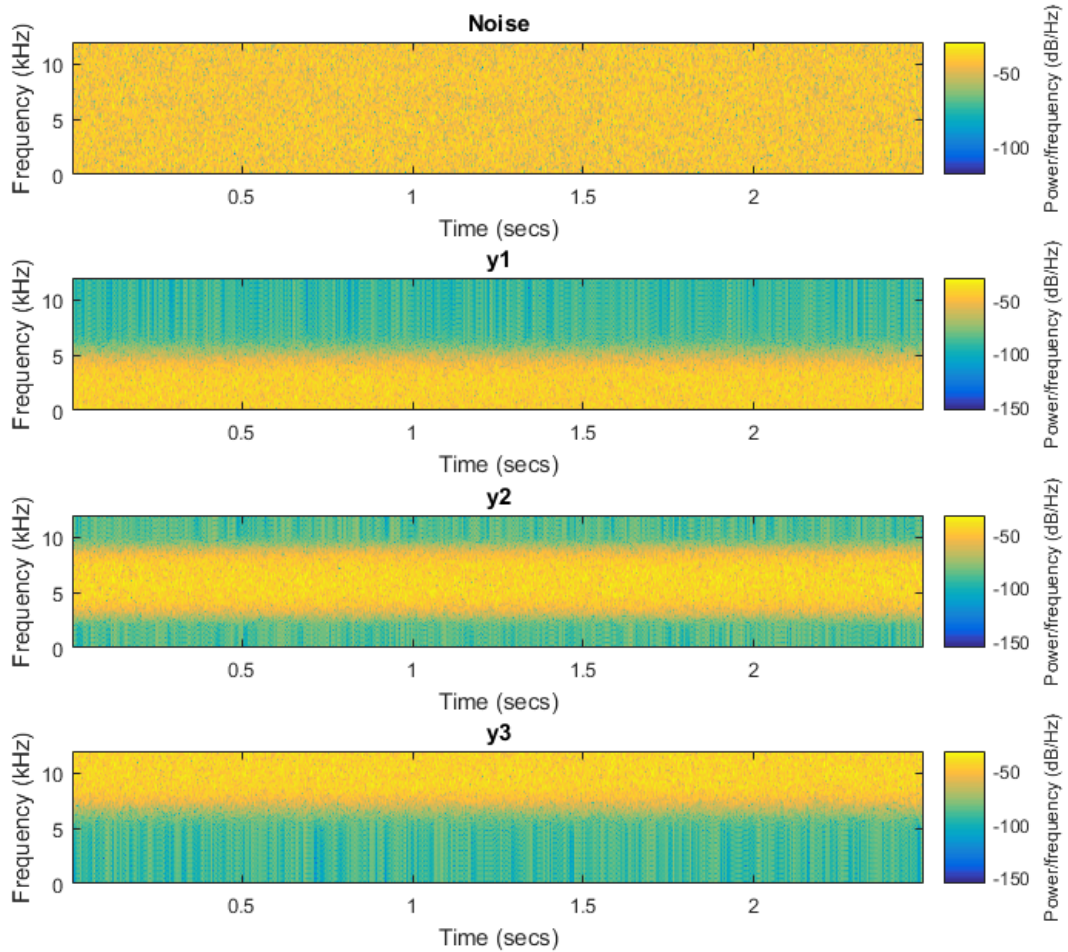
Call `step` to pass the noise signal through your crossover filter.

```
[y1,y2,y3] = step(crossFilt,noise);
```

Visualize the results using a spectrogram.

```
figure('Position',[100,100,800,700]);
subplot(4,1,1)
    spectrogram(noise,120,100,6000,24000,'yaxis');
    title('Noise');
subplot(4,1,2)
    spectrogram(y1,120,100,6000,24000,'yaxis');
    title('y1');
subplot(4,1,3)
    spectrogram(y2,120,100,6000,24000,'yaxis');
```

```
        title('y2');  
subplot(4,1,4)  
    spectrogram(y3,120,100,6000,24000,'yaxis');  
        title('y3');
```



Split Audio Signal into Three Bands

Use the `crossoverFilter` System object™ to split an audio signal into three frequency bands.

Construct the audio file reader and audio device writer System objects. Use the sample rate of the reader as the sample rate of the writer. Call `setup` to reduce the computation load of initialization in an audio stream loop.

```
samplesPerFrame = 256;

fileReader = dsp.AudioFileReader(...
    'RockGuitar-16-44p1-stereo-72secs.wav',...
    'SamplesPerFrame',samplesPerFrame);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);

setup(fileReader);
setup(deviceWriter,ones(samplesPerFrame,2));
```

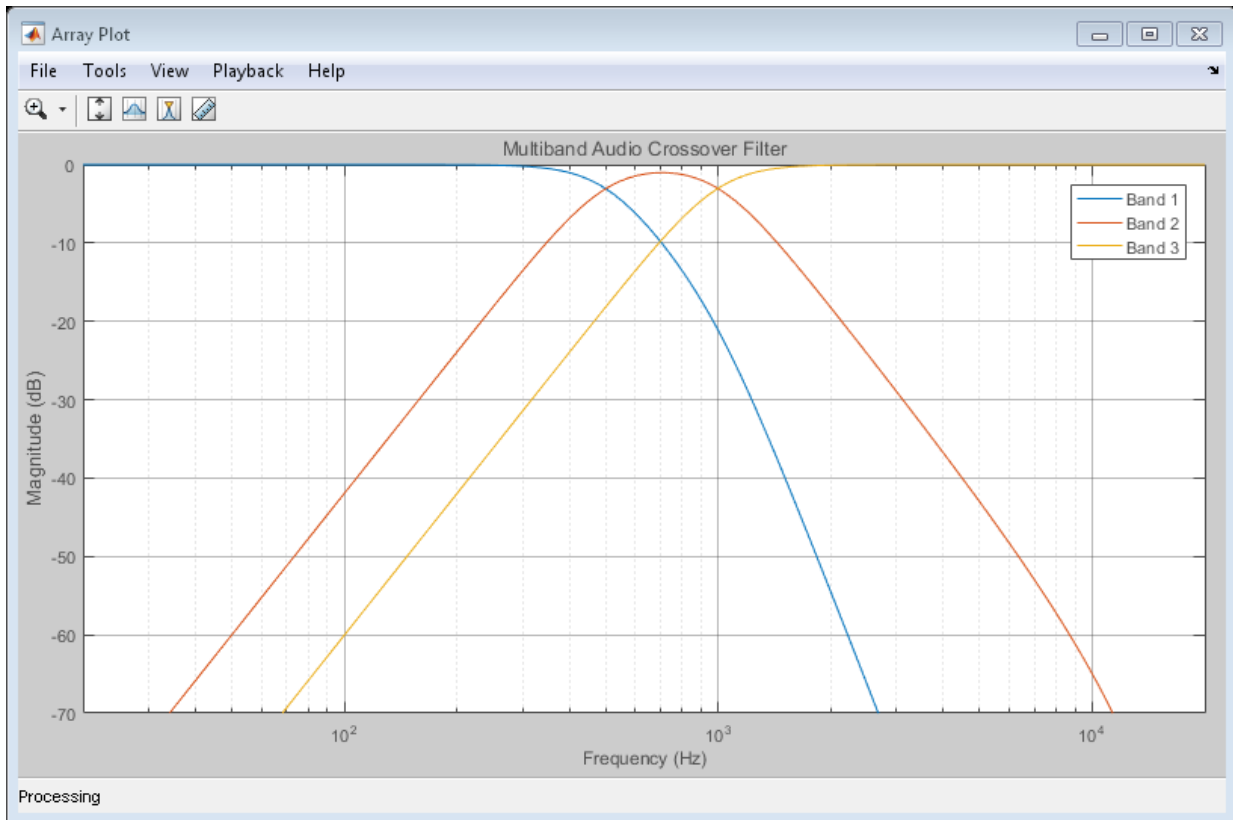
Create a crossover filter System object with 2 crossovers (3 bands), crossover frequencies at 500 Hz and 1 kHz, and a slope of 18 dB/octave. Use the sample rate of the reader as the sample rate of the crossover filter.

```
crossFilt = crossoverFilter(...
    'NumCrossovers',2,...
    'CrossoverFrequencies',[500,1000],...
    'CrossoverSlopes',18,...
    'SampleRate',fileReader.SampleRate);

setup(crossFilt,ones(samplesPerFrame,2));
```

Visualize the bands of the crossover filter.

```
visualize(crossFilt);
```



Get the cost of the crossover filter.

```
cost(crossFilt)
```

```
ans =
```

```

      NumCoefficients: 48
      NumStates: 18
 MultiplicationsPerInputSample: 48
      AdditionsPerInputSample: 37

```

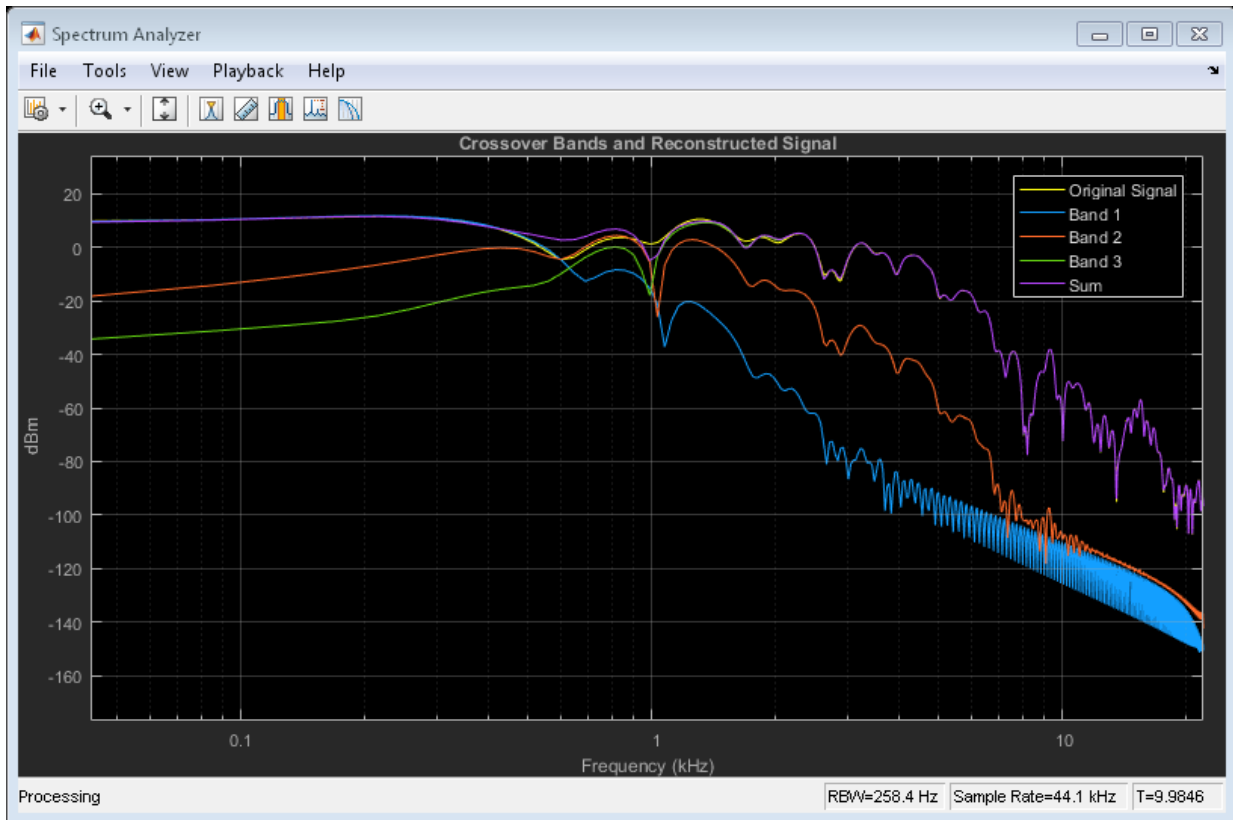
Create a spectrum analyzer to visualize the effect of the crossover filter.

```
scope = dsp.SpectrumAnalyzer(...
```

```
'SampleRate',fileReader.SampleRate,...  
'PlotAsTwoSidedSpectrum',false,...  
'FrequencyScale','Log',...  
'FrequencyResolutionMethod','WindowLength',...  
'WindowLength',samplesPerFrame,...  
'Title',...  
'Crossover Bands and Reconstructed Signal',...  
'ShowLegend',true,...  
'ChannelNames',{'Original Signal','Band 1',...  
'Band 2','Band 3','Sum'});
```

Play 10 seconds of the audio signal. Visualize the spectrum of the original audio, the crossover bands, and the reconstructed signal (sum of bands).

```
setup(scope,ones(samplesPerFrame,5));  
count = 0;  
while count < (fileReader.SampleRate/samplesPerFrame)*10  
    originalSignal = step(fileReader);  
    [band1,band2,band3] = step(crossFilt,originalSignal);  
    sumOfBands = band1 + band2 + band3;  
    step(scope,...  
        [originalSignal(:,1),...  
        band1(:,1),...  
        band2(:,1),...  
        band3(:,1),...  
        sumOfBands(:,1)]);  
    step(deviceWriter,sumOfBands);  
    count = count+1;  
end
```

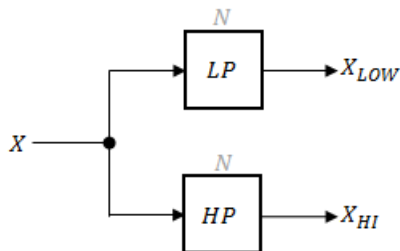



Algorithms

The crossover System object is implemented as a binary tree of crossover pairs with additional phase-compensating sections [1]. Odd-order crossovers are implemented with Butterworth filters, while even-order crossovers are implemented with cascaded Butterworth filters (Linkwitz-Riley filters).

Odd-Order Crossover Pair

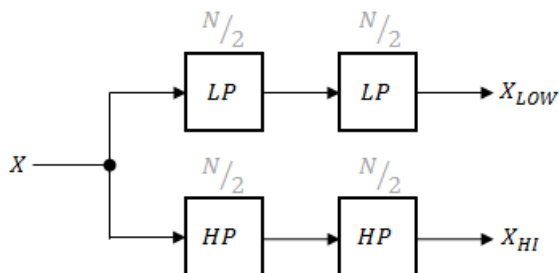
Odd-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.



LP and *HP* are Butterworth filters of order N , implemented as direct-form II transposed second-order sections. The shared cutoff frequency used in their design corresponds to the crossover of the resulting bands.

Even-Order Crossover Pair

Even-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.

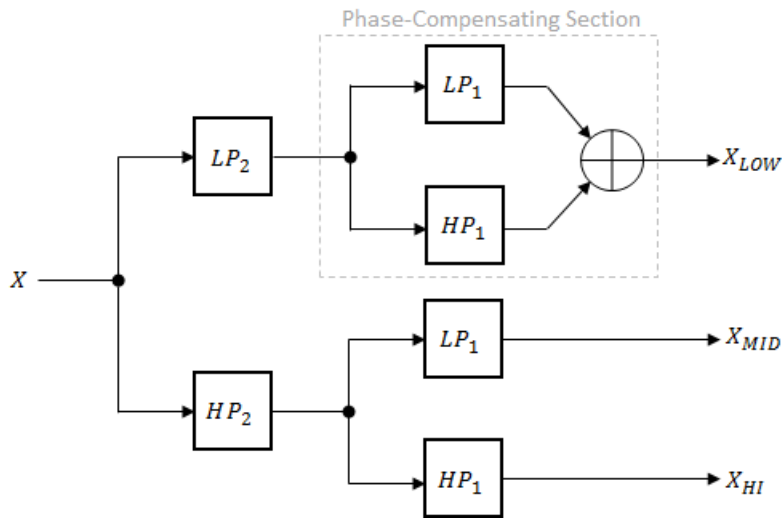


LP and *HP* are Butterworth filters of order $N/2$, where N is the order of the overall filter. The filters are implemented as direct-form II transposed second-order sections.

For overall filters of orders 2 and 6, X_{HI} is multiplied by -1 internally so that the branches of your crossover pair are in-phase.

Even-Order Three-Band Filter

Even-order three-band (two crossovers) filters are implemented as parallel complementary highpass and lowpass filters organized in a tree structure.



The phase-compensating section is equivalent to an allpass filter.

The design of four-band and five-band filters (three and four crossovers) are extensions of the pattern developed for even-order and odd-order crossovers and the tree structure specified for three-band (two crossover) filters.

References

- [1] D'Appolito, Joseph A. "Active Realization of Multiway All-Pass Crossover Systems". *Journal of Audio Engineering Society*. Vol. 35, Issue 4, pp. 239–245.

See Also

multibandParametricEQ | Crossover Filter

Introduced in R2016a

clone

System object: crossoverFilter

Create copy of System object with same property values

Syntax

```
crossFiltClone = clone(crossFilt)
```

Description

`crossFiltClone = clone(crossFilt)` creates a crossover filter System object, `crossFiltClone`, with the same property values as `CROSSFILT`. If the original object is locked, then `clone` creates a copy that is also locked. This copy has states initialized to the same values as the original. If the original object is not locked, then `clone` creates a new unlocked object with uninitialized states.

Introduced in R2016a

configureMIDI

System object: crossoverFilter

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(crossFilt)
configureMIDI(crossFilt,propName)
configureMIDI(crossFilt,propName,controlNumber)
configureMIDI(crossFilt,propName,controlNumber,'DeviceName',
deviceName)
```

Description

`configureMIDI(crossFilt)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the crossover filter System object, `crossFilt`, to MIDI controls of your choice.

`configureMIDI(crossFilt,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

`configureMIDI(crossFilt,propName,controlNumber)` makes the property respond to the MIDI control specified by `controlNumber`.

`configureMIDI(crossFilt,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `crossoverFilter` System object maps to MIDI controls with a specified range.

Property	Range	Unit
CrossoverFrequencies	20 to 20,000	Hz
CrossoverSlopes	6 to 48	dB/octave

Introduced in R2016a

cost

System object: crossoverFilter

Implementation cost of System object

Syntax

```
C = cost(crossFilt)
```

Description

`C = cost(crossFilt)` returns a structure, `C`, whose fields contain information about the computation cost of implementing the crossover filter, `CROSSFILT`.

Structure Field	Description
<code>NumCoefficients</code>	Number of filter coefficients (excluding coefficients with values 0, 1, or -1)
<code>NumStates</code>	Number of states
<code>MultiplicationsPerInputSample</code>	Number of multiplications per input sample
<code>AdditionsPerInputSample</code>	Number of additions per input sample

Introduced in R2016a

createAudioPluginClass

System object: crossoverFilter

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(crossFilt)
createAudioPluginClass(crossFilt,pluginName)
```

Description

`createAudioPluginClass(crossFilt)` creates a System object plugin that implements the functionality of the `crossoverFilter` System object, `crossFilt`. The name of the created class is the `crossoverFilter` System object variable name followed by 'Plugin', for example, `crossFiltPlugin`.

Note: If the object is locked, the number of input and output channels of the plugin is equal to the number of channels of the object. Otherwise, the number of channels is equal to 2.

`createAudioPluginClass(crossFilt,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(crossFilt, 'xOverFilter')` creates a System object plugin with class name `xOverFilter`.

Each tunable property of the `crossoverFilter` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Unit
CrossoverFrequencies	20 to 20,000	Hz
CrossoverSlopes	6 to 48	dB/octave

Introduced in R2016a

disconnectMIDI

System object: `crossoverFilter`

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(crossFilt)
```

Description

`disconnectMIDI(crossFilt)` disconnects MIDI controls from your crossover filter, `crossFilt`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: crossoverFilter

Get MIDI connection information

Syntax

```
connectionInfo= getMIDIConnections(crossFilt)
```

Description

`connectionInfo= getMIDIConnections(crossFilt)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your crossover filter, `crossFilt`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `crossFilt` that has established MIDI connections. Each substructure contains the control number, the device name of the corresponding MIDI control, and the property mapping information (mapping rule, minimum value, and maximum value).

Introduced in R2016a

isLocked

System object: `crossoverFilter`

Locked status for input attributes and nontunable properties

Syntax

`L = isLocked(crossFilt)`

Description

`L = isLocked(crossFilt)` returns a logical value, `L`, that indicates whether input attributes and nontunable properties are locked for the crossover filter, `crossFilt`.

The `crossFilter` object performs an internal initialization the first time you execute `step`. The initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After the object is locked, the `isLocked` method returns a `true` value.

Introduced in R2016a

release

System object: crossoverFilter

Enable property values and input characteristics to change

Syntax

```
release(crossFilt)
```

Description

`release(crossFilt)` releases system resources, such as memory, of your crossover filter, `crossFilt`. After you call **release**, all properties and input characteristics of `crossFilt` can change.

Note: Once you call **release** on a System object, subsequent calls to **setup**, **step**, **reset**, or **release** do not support code generation.

Introduced in R2016a

reset

System object: `crossoverFilter`

Reset internal states of System object

Syntax

```
reset(crossFilter)
```

Description

`reset(crossFilter)` resets internal states of the crossover filter, `crossFilt`, to their initial values.

Introduced in R2016a

step

System object: crossoverFilter

Implement audio crossover filter

Syntax

```
[band1,band2,...,bandN] = step(crossFilt,x)
```

Description

[band1,band2,...,bandN] = step(crossFilt,x) applies a crossover filter on the input, x, and returns the filtered output bands, [band1,band2,...,bandN], where N = NumCrossovers + 1.

x must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute step. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

Introduced in R2016a

visualize

System object: `crossoverFilter`

Visualize magnitude response of System object

Syntax

```
visualize(crossFilt)  
visualize(crossFilt,N)
```

Description

`visualize(crossFilt)` plots the magnitude response of each individual filter band.

`visualize(crossFilt,N)` specifies an N-point FFT used to calculate the magnitude response. The default is 2048.

Introduced in R2016a

compressor System object

Dynamic range compressor

Description

The `compressor` System object performs dynamic range compression independently across each input channel. Dynamic range compression attenuates the volume of loud sounds that cross a given threshold. It uses specified attack and release times to achieve a smooth applied gain curve. Properties of the `compressor` System object specify the type of dynamic range compression.

To perform dynamic range compression on your input:

- 1 Define and set up your dynamic range compressor. See “Construction” on page 3-85.
- 2 Call step to perform dynamic range compression on each channel of the input signal according to the properties of your `compressor` object. The input must be a real-valued, double-precision or single-precision matrix. The `compressor` object treats each column of the input as an independent channel.

Construction

`dRC = compressor` creates a System object, `dRC`, that performs dynamic range compression independently across each input channel over time.

`dRC = compressor(thresholdValue)` sets the `Threshold` property to `thresholdValue`.

`dRC = compressor(thresholdValue, ratioValue)` sets the `Ratio` property to `ratioValue`.

`dRC = compressor(Name, Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `dRC = compressor('AttackTime',0.01,'SampleRate',16000)` creates a System object, `dRC`, with the `AttackTime` property set to 0.01 and the `SampleRate` property set to 16000.

Properties

Threshold — Operation threshold (dB)

-10 (default) | real scalar

Operation threshold in dB, specified as a real scalar.

Operation threshold is the level above which gain is applied to the input signal.

This property is tunable. You can change the value of this property even when the object is locked.

Ratio — Compression ratio

5 (default) | real scalar

Compression ratio, specified as a real scalar greater than or equal to 1.

Compression ratio is the input/output ratio for signals that overshoot the operation threshold.

Assuming a hard knee characteristic and a steady-state input such that $x[n]$ dB >

`thresholdValue`, the compression ratio is defined as
$$R = \frac{(x[n] - T)}{(y[n] - T)}$$
.

- R is the compression ratio.
- $x[n]$ is the input signal in dB.
- $y[n]$ is the output signal in dB.
- T is the threshold in dB.

This property is tunable. You can change the value of this property even when the object is locked.

KneeWidth — Knee width (dB)

0 (default) | real scalar

Knee width in dB, specified as a real scalar greater than or equal to 0.

Knee width is the transition area in the compression characteristic.

For soft knee characteristics, the transition area is defined by the relation

$$y = x + \frac{\left(\frac{1}{R} - 1\right) \times \left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $(2 \times |x - T|) \leq W$.

- y is the output level in dB.
- x is the input level in dB.
- R is the compression ratio.
- T is the threshold in dB.
- W is the knee width in dB.

This property is tunable. You can change the value of this property even when the object is locked.

AttackTime — Attack time (s)

0.05 (default) | real scalar

Attack time in seconds, specified as a real scalar greater than or equal to 0.

Attack time is the time it takes the compressor gain to rise from 10% to 90% of its final value when the input goes above the threshold.

This property is tunable. You can change the value of this property even when the object is locked.

ReleaseTime — Release time (s)

0.2 (default) | real scalar

Release time in seconds, specified as a real scalar greater than or equal to 0.

Release time is the time it takes the compressor gain to drop from 90% to 10% of its final value when the input goes below the threshold.

This property is tunable. You can change the value of this property even when the object is locked.

MakeUpGainMode — Make-up gain mode

'Auto' (default) | 'Property'

Make-up gain mode, specified as 'Auto' or 'Property'.

- 'Auto' — Make-up gain is applied at the output of the dynamic range compressor such that a steady-state 0 dB input has a 0 dB output.
- 'Property' — Make-up gain is set to the value specified in the `MakeUpGain` property.

MakeUpGain — Make-up gain (dB)

0 (default) | real scalar

Make-up gain in dB, specified as a real scalar.

Make-up gain compensates for gain lost during compression. It is applied at the output of the dynamic range compressor. This property is available when you set `MakeUpGainMode` to 'Property'.

This property is tunable. You can change the value of this property even when the object is locked.

SampleRate — Input sample rate (Hz)

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

This property is tunable. You can change the value of this property even when the object is locked.

Methods

`clone`

Create copy of System object with same property values

`configureMIDI`

Configure MIDI connections between System object and MIDI controller

<code>createAudioPluginClass</code>	Create audio plugin class that implements functionality of System object
<code>disconnectMIDI</code>	Disconnect MIDI controls from System object
<code>getMIDIConnections</code>	Get MIDI connection information
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Enable property values and input characteristics to change
<code>reset</code>	Reset internal states of System object
<code>step</code>	Perform dynamic range compression
<code>visualize</code>	Visualize static compression characteristics of System object

Examples

Compress Audio Signal

Use dynamic range compression to attenuate the volume of loud sounds.

Set up the audio file reader and audio device writer System objects.

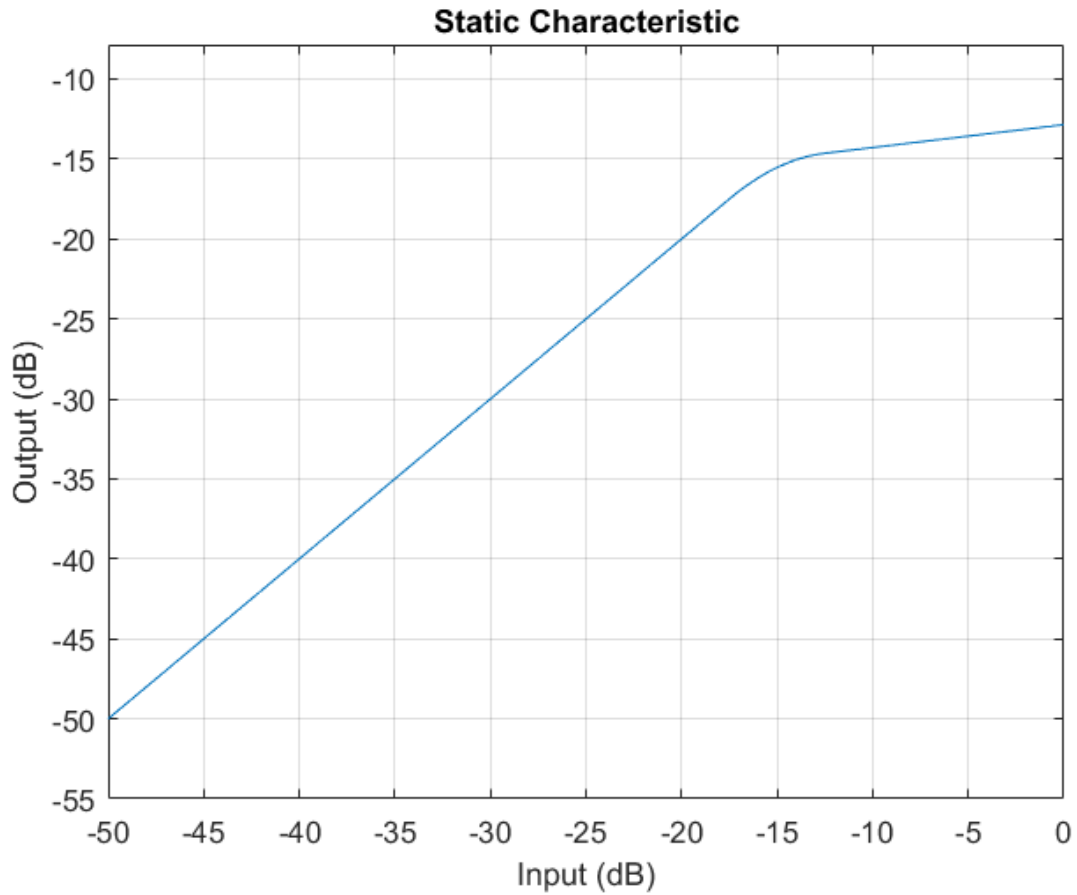
```
frameLength = 1024;
fileReader = dsp.AudioFileReader(...
    'Filename', 'RockDrums-44p1-stereo-11secs.mp3', ...
    'SamplesPerFrame', frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate', fileReader.SampleRate);
```

Set up the compressor to have a threshold of -15 dB, a ratio of 7, and a knee width of 5. Use the sample rate of your audio file reader.

```
dRC = compressor(-15, 7, ...
    'KneeWidth', 5, ...
    'SampleRate', fileReader.SampleRate);
```

Visualize the compression static characteristic.

```
visualize(dRC);
```



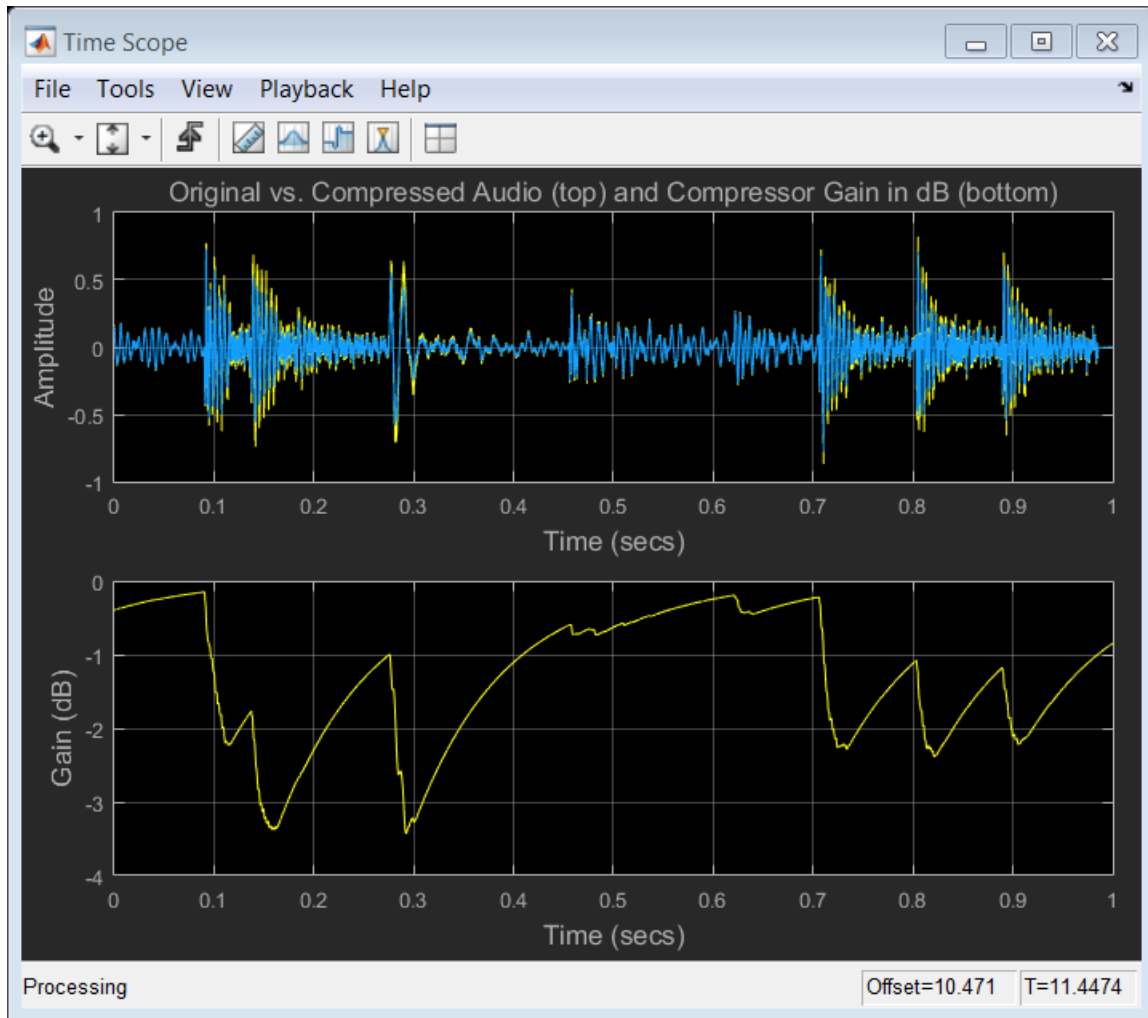
Set up the scope to visualize the original audio signal, the compressed audio signal, and the applied compressor gain.

```
scope = dsp.TimeScope(...  
    'SampleRate',fileReader.SampleRate,...  
    'TimeSpan',1,...  
    'BufferLength',44100*4,...  
    'YLimits',[-1,1],...  
    'TimeSpanOverrunAction','Scroll',...  
    'ShowGrid',true,...  
    'LayoutDimensions',[2,1],...
```

```
    'NumInputPorts',2,...  
    'Title',...  
    ['Original vs. Compressed Audio (top)'...  
     ' and Compressor Gain in dB (bottom)']);  
scope.ActiveDisplay = 2;  
scope.YLimits = [-4,0];  
scope.YLabel = 'Gain (dB)';
```

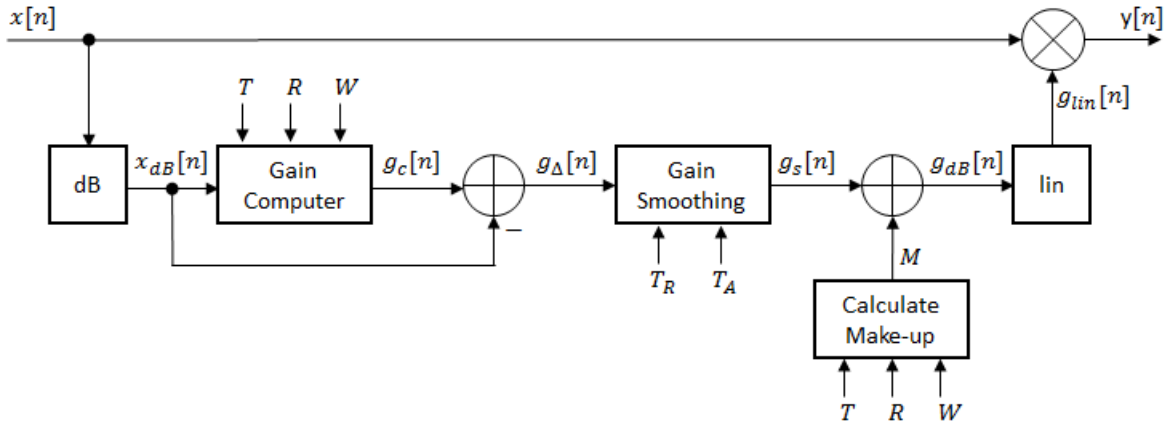
Play the processed audio and visualize it on the scope.

```
while ~isDone(fileReader)  
    x = step(fileReader);  
    [y,g] = step(dRC,x);  
    step(deviceWriter,y);  
    x1 = x(:,1);  
    y1 = y(:,1);  
    step(scope,[x1,y1],g(:,1))  
end
```



Algorithms

The `compressor` System object processes a signal frame by frame and element by element.



- 1 The N -point signal, $x[n]$, is converted to decibels:

$$x_{dB}[n] = 20 \times \log_{10} |x[n]|.$$

- 2 $x_{dB}[n]$ passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range compressor to attenuate gain that is above the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:

$$g_c(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{\left(\frac{1}{R} - 1\right) \left(x_{dB} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \leq x_{dB} \leq \left(T + \frac{W}{2}\right) \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases},$$

where T is the threshold, R is the ratio, and W is the knee width.

If you specified a hard knee, the gain computer has the following static characteristic:

$$g_c(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < T \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} \geq T \end{cases} .$$

- 3 The gain modification, $g_\Delta[n]$, is calculated as

$$g_\Delta[n] = g_c[n] - x_{dB}[n].$$

- 4 $g_\Delta[n]$ is smoothed using specified attack and release time properties,

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) |g_\Delta[n]|, & |g_\Delta[n]| > g_s[n-1] \\ \alpha_R g_s[n-1] + (1 - \alpha_R) |g_\Delta[n]|, & |g_\Delta[n]| \leq g_s[n-1] \end{cases} ,$$

where α_A , the attack time coefficient, is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{F_s \times T_A}\right),$$

and α_R , the release time coefficient, is calculated as

$$\alpha_R = \exp\left(\frac{-\log(9)}{F_s \times T_R}\right).$$

T_A is the attack time period, specified by the `AttackTime` property. T_R is the release time period, specified by the `ReleaseTime` property. F_s is the input sampling rate, specified by the `SampleRate` property.

- 5 If `MakeUpGainMode` is set to the default 'Auto', the make-up gain is calculated as the negative of the computed gain for a 0 dB input,

$$M = -g_c(x_{dB} = 0).$$

Given a steady-state input of 0 dB, this configuration achieves a steady-state output of 0 dB. The make-up gain is determined by the `Threshold`, `Ratio`, and `KneeWidth` properties. It does not depend on the input signal.

- 6 The make-up gain, M , is added to the smoothed gain, $g_s[n]$:

$$g_{dB}[n] = g_s[n] + M.$$

- 7 The calculated gain in dB, $g_{dB}[n]$, is translated to a linear domain:

$$g_{lin}[n] = 10^{\left(\frac{g_{dB}[n]}{20}\right)}.$$

- 8 The output of the dynamic range compressor is given as

$$y[n] = x[n] \times g_{lin}[n].$$

References

- [1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. “Digital Dynamic Range Compressor Design—A Tutorial And Analysis”. *Journal of Audio Engineering Society*. Vol. 60, Issue 6, pp. 399–408.

See Also

expander | noiseGate | limiter | Compressor

Introduced in R2016a

clone

System object: compressor

Create copy of System object with same property values

Syntax

```
dRCclone = clone(dRC)
```

Description

`dRCclone = clone(dRC)` creates a dynamic range compressor System object, `dRCclone`, with the same property values as `dRC`. If the original object is locked, then `clone` creates a copy that is also locked. This copy has states initialized to the same values as the original. If the original object is not locked, then `clone` creates a new unlocked object with uninitialized states.

Introduced in R2016a

configureMIDI

System object: compressor

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(dRC)
configureMIDI(dRC,propName)
configureMIDI(dRC,propName,controlNumber)
configureMIDI(dRC,propName,controlNumber,'DeviceName',deviceName)
```

Description

`configureMIDI(dRC)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the dynamic range compressor System object, `dRC`, to MIDI controls of your choice.

`configureMIDI(dRC,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

`configureMIDI(dRC,propName,controlNumber)` makes the property respond to the MIDI control specified by `controlNumber`.

`configureMIDI(dRC,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `compressor` System object maps to MIDI controls with a specified range.

Property	Range	Unit
Threshold	-50 to 0	dB
Ratio	1 to 50	none
KneeWidth	0 to 20	dB

Property	Range	Unit
AttackTime	0 to 4	seconds
ReleaseTime	0 to 4	seconds
MakeUpGain (available when you set MakeUpGainMode to 'Property')	-10 to 24	dB

Introduced in R2016a

createAudioPluginClass

System object: compressor

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(dRC)
createAudioPluginClass(dRC,pluginName)
```

Description

`createAudioPluginClass(dRC)` creates a System object plugin that implements the functionality of the `compressor` System object, `dRC`. The name of the created class is the `compressor` System object variable name followed by 'Plugin', for example, `dRCPlugin`.

Note: If the object is locked, the number of input and output channels of the plugin is equal to the number of channels of the object. Otherwise, the number of channels is equal to 2.

`createAudioPluginClass(dRC,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(dRC,'myCompressor')` creates a System object plugin with class name `myCompressor`.

Each tunable property of the `compressor` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Unit
Threshold	-50 to 0	dB
Ratio	1 to 50	none
KneeWidth	0 to 20	dB

Property	Plugin Parameter Range	Unit
AttackTime	0 to 4	seconds
ReleaseTime	0 to 4	seconds
MakeUpGain (available when you set MakeUpGainMode to 'Property')	-10 to 24	dB

Introduced in R2016a

disconnectMIDI

System object: compressor

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(dRC)
```

Description

`disconnectMIDI(dRC)` disconnects MIDI controls from your dynamic range compressor, `dRC`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: compressor

Get MIDI connection information

Syntax

```
connectionInfo = getMIDIConnections(dRC)
```

Description

`connectionInfo = getMIDIConnections(dRC)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your dynamic range compressor, `dRC`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `dRC` that has established MIDI connections. Each substructure contains the control number, the device name of the corresponding MIDI control, and the property mapping information (mapping rule, minimum value, and maximum value).

Introduced in R2016a

isLocked

System object: compressor

Locked status for input attributes and nontunable properties

Syntax

`L = isLocked(dRC)`

Description

`L = isLocked(dRC)` returns a logical value, `L`, that indicates whether input attributes and nontunable properties are locked for the dynamic range compressor, `dRC`.

The `dRC` object performs an internal initialization the first time you execute `step`. The initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After the object is locked, the `isLocked` method returns a `true` value.

Introduced in R2016a

release

System object: compressor

Enable property values and input characteristics to change

Syntax

```
release(dRC)
```

Description

`release(dRC)` releases system resources, such as memory, from your dynamic range compressor, `dRC`. After you call `release`, all properties and input characteristics of `dRC` can change.

Note: Once you call `release` on a System object, subsequent calls to `setup`, `step`, `reset`, or `release` do not support code generation.

Introduced in R2016a

reset

System object: compressor

Reset internal states of System object

Syntax

reset(dRC)

Description

reset(dRC) resets internal states of the dynamic range compressor, dRC, to their initial values.

Introduced in R2016a

step

System object: compressor

Perform dynamic range compression

Syntax

```
y = step(dRC,x)
[y,g] = step(dRC,x)
```

Description

`y = step(dRC,x)` performs dynamic range compression on the input signal, `x`, and returns the compressed signal, `y`. The type of dynamic range compression is specified by the algorithm and properties of the `compressor` System object, `dRC`.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

`[y,g] = step(dRC,x)` also returns the gain, in dB, applied at each input sample.

Introduced in R2016a

visualize

System object: compressor

Visualize static compression characteristics of System object

Syntax

```
visualize(dRC)
visualize(dRC,myInputRange)
Y = visualize( ___ )
```

Description

`visualize(dRC)` plots the static compression characteristic of the dynamic range compressor, `dRC`. The method computes the dB output level for the input range `[-50:0.01:0]` dB.

`visualize(dRC,myInputRange)` enables you to specify the input range in dB. Specify `myInputRange` as a vector of ascending values.

`Y = visualize(___)` returns the dB output level, `Y`, corresponding to the input range. You can use any of the input arguments from previous syntaxes.

Introduced in R2016a

expander System object

Dynamic range expander

Description

The `expander` System object performs dynamic range expansion independently across each input channel. Dynamic range expansion attenuates the volume of quiet sounds below a given threshold. It uses specified attack, release, and hold times to achieve a smooth applied gain curve. Properties of the `expander` System object specify the type of dynamic range expansion.

To perform dynamic range expansion on your input:

- 1 Define and set up your dynamic range expander. See “Construction” on page 3-108.
- 2 Call step to perform dynamic range expansion on each channel of the input signal according to the properties of your `expander` object. The input must be a real-valued, double-precision or single-precision matrix. The `expander` object treats each column of the input as an independent channel.

Construction

`dRE = expander` creates a System object, `dRE`, that performs dynamic range expansion independently across each input channel.

`dRE = expander(thresholdValue)` sets the `Threshold` property to `thresholdValue`.

`dRE = expander(thresholdValue, ratioValue)` sets the `Ratio` property to `ratioValue`.

`dRE = expander(Name, Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `dRE = expander('AttackTime', 0.01, 'SampleRate', 16000)` creates a System object, `dRE`, with the `AttackTime` property set to 0.01, and the `SampleRate` property set to 16000.

Properties

Threshold — Operation threshold (dB)

-10 (default) | real scalar

Operation threshold in dB, specified as a real scalar.

Operation threshold is the level below which gain is applied to the input signal.

This property is tunable. You can change the value of this property even when the object is locked.

Ratio — Expansion ratio

5 (default) | real scalar

Expansion ratio, specified as a real scalar greater than or equal to 1.

Expansion ratio is the input/output ratio for signals that undershoot the operation threshold.

Assuming a hard knee characteristic and a steady-state input such that $x[n]$ dB <

thresholdValue, the expansion ratio is defined as
$$R = \frac{(y[n]-T)}{(x[n]-T)}$$
.

- R is the expansion ratio.
- $y[n]$ is the output signal in dB.
- $x[n]$ is the input signal in dB.
- T is the threshold in dB.

This property is tunable. You can change the value of this property even when the object is locked.

KneeWidth — Knee width (dB)

0 (default) | real scalar

Knee width in dB, specified as a real scalar greater than or equal to 0.

Knee width is the transition area in the expansion characteristic.

For soft knee characteristics, the transition area is defined by the relation

$$y = x + \frac{(1 - R) \times \left(x - T - \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $(2 \times |x - T|) \leq W$.

- y is the output level in dB.
- x is the input level in dB.
- R is the expansion ratio.
- T is the threshold in dB.
- W is the knee width in dB.

This property is tunable. You can change the value of this property even when the object is locked.

AttackTime — Attack time (s)

0.05 (default) | real scalar

Attack time in seconds, specified as a real scalar greater than or equal to 0.

Attack time is the time it takes the expander gain to rise from 10% to 90% of its final value when the input goes below the threshold.

This property is tunable. You can change the value of this property even when the object is locked.

ReleaseTime — Release time (s)

0.2 (default) | real scalar

Release time in seconds, specified as a real scalar greater than or equal to 0.

Release time is the time it takes the expander gain to drop from 90% to 10% of its final value when the input goes above the threshold.

This property is tunable. You can change the value of this property even when the object is locked.

HoldTime — Hold time (s)

0.05 (default) | real scalar

Hold time in seconds, specified as a real scalar greater than or equal to 0.

Hold time is the period in which the applied gain is held constant before it starts moving toward its steady-state value. Hold time begins when the input level crosses the operation threshold.

This property is tunable. You can change the value of this property even when the object is locked.

SampleRate — Input sample rate (Hz)

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

This property is tunable. You can change the value of this property even when the object is locked.

Methods

clone	Create copy of System object with same property values
configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
isLocked	Locked status for input attributes and nontunable properties
release	Enable property values and input characteristics to change
reset	Reset internal states of System object

step	Perform dynamic range expansion
visualize	Visualize static expander characteristics of System object

Examples

Expand Audio Signal

Use dynamic range expansion to attenuate background noise from an audio signal.

Set up the audio file reader and audio device writer System objects.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader(...
    'Filename','Counting-16-44p1-mono-15secs.wav',...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
```

Corrupt the audio signal with Gaussian noise. Play the audio.

```
while ~isDone(fileReader)
    x = step(fileReader);
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);
    play(deviceWriter,xCorrupted);
end

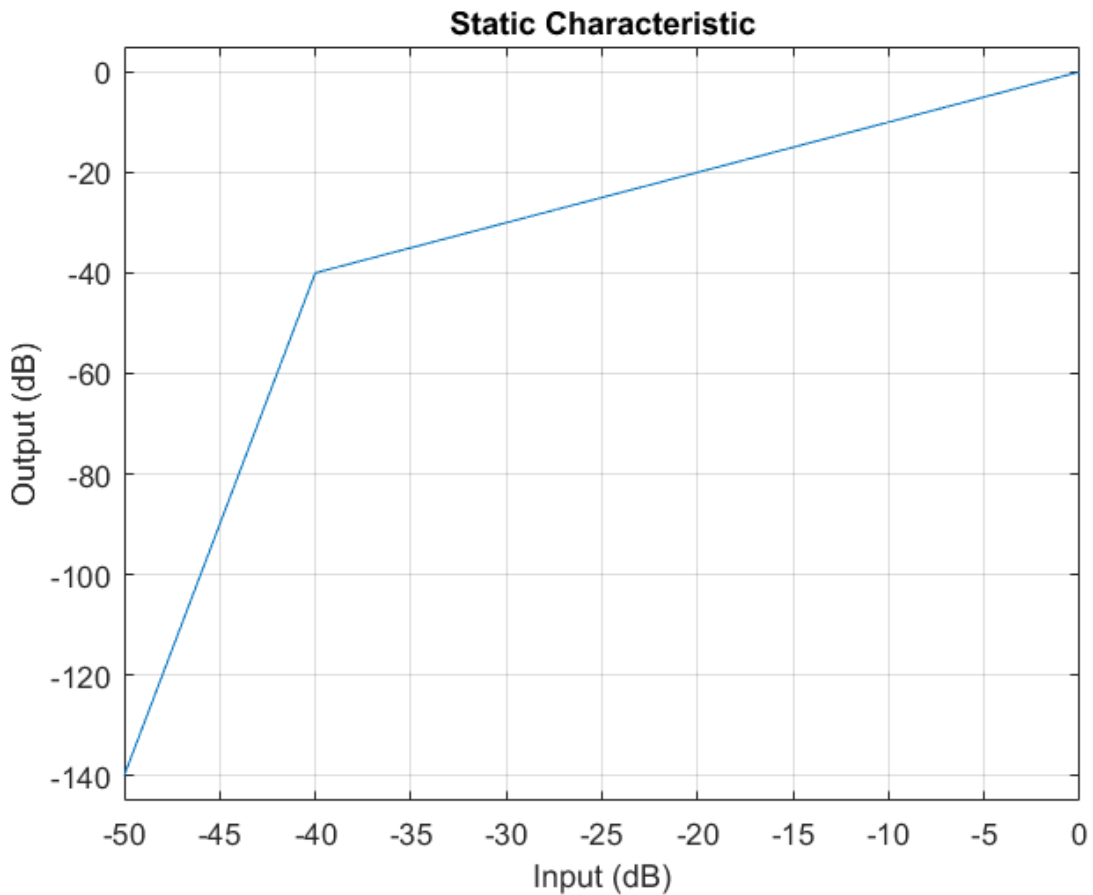
release(fileReader);
```

Set up the expander with a threshold of -40 dB, a ratio of 10, an attack time of 0.01 seconds, a release time of 0.02 seconds, and a hold time of 0 seconds. Use the sample rate of your audio file reader.

```
dRE = expander(-40,10,...
    'AttackTime',0.01,...
    'ReleaseTime',0.02,...
    'HoldTime',0,...
    'SampleRate',fileReader.SampleRate);
```

Visualize the expansion static characteristic.

```
visualize(dRE);
```

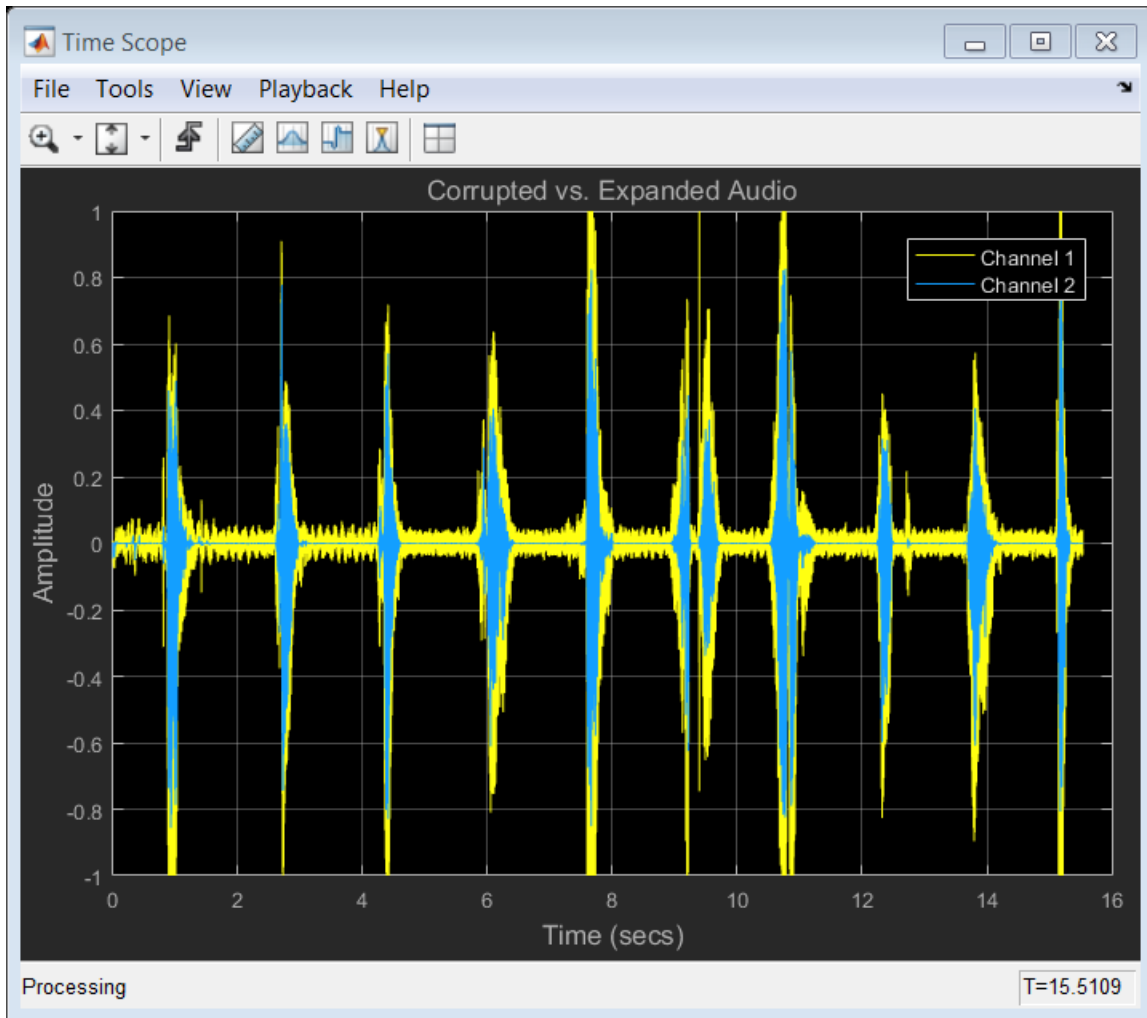


Set up the scope to visualize the signal before and after dynamic range expansion.

```
scope = dsp.TimeScope(...  
    'SampleRate',fileReader.SampleRate,...  
    'TimeSpanOverrunAction','Scroll',...  
    'TimeSpan',16,...  
    'BufferLength',1.5e6,...  
    'YLimits',[-1 1],...  
    'ShowGrid',true,...  
    'ShowLegend',true,...  
    'Title','Corrupted vs. Expanded Audio');
```

Play the processed audio and visualize it on the scope.

```
while ~isDone(fileReader)
    x = step(fileReader);
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);
    y = step(dRE,xCorrupted);
    play(deviceWriter,y);
    step(scope,[xCorrupted,y])
end
```



See Also

noiseGate | compressor | limiter | Expander

Introduced in R2016a

clone

System object: expander

Create copy of System object with same property values

Syntax

```
dREclone = clone(dRE)
```

Description

`dREclone = clone(dRE)` creates a dynamic range expander System object, `dREclone`, with the same property values as `dRE`. If the original object is locked, then `clone` creates a copy that is also locked. This copy has states initialized to the same values as the original. If the original object is not locked, then `clone` creates a new unlocked object with uninitialized states.

Introduced in R2016a

configureMIDI

System object: expander

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(dRE)
configureMIDI(dRE,propName)
configureMIDI(dRE,propName,controlNumber)
configureMIDI(dRE,propName,controlNumber,'DeviceName',deviceName)
```

Description

`configureMIDI(dRE)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the dynamic range expander System object, `dRE`, to MIDI controls of your choice.

`configureMIDI(dRE,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

`configureMIDI(dRE,propName,controlNumber)` makes the property respond to the MIDI control specified by `controlNumber`.

`configureMIDI(dRE,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `expander` System object maps to MIDI controls with a specified range.

Property	Range	Unit
Threshold	-140 to 0	dB
Ratio	1 to 50	none
KneeWidth	0 to 20	dB

Property	Range	Unit
AttackTime	0 to 4	seconds
ReleaseTime	0 to 4	seconds
HoldTime	0 to 4	seconds

Introduced in R2016a

createAudioPluginClass

System object: expander

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(dRE)
createAudioPluginClass(dRE,pluginName)
```

Description

`createAudioPluginClass(dRE)` creates a System object plugin that implements the functionality of the expander System object, `dRE`. The name of the created class is the expander System object variable name followed by 'Plugin', for example, `dREPlugin`.

Note: If the object is locked, the number of input and output channels of the plugin is equal to the number of channels of the object. Otherwise, the number of channels is equal to 2.

`createAudioPluginClass(dRE,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(dRE, 'myExpander')` creates a System object plugin with class name `myExpander`.

Each tunable property of the expander System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Unit
Threshold	-140 to 0	dB
Ratio	1 to 50	none
KneeWidth	0 to 20	dB

Property	Plugin Parameter Range	Unit
AttackTime	0 to 4	seconds
ReleaseTime	0 to 4	seconds
HoldTime	0 to 4	seconds

Introduced in R2016a

disconnectMIDI

System object: expander

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(dRE)
```

Description

`disconnectMIDI(dRE)` disconnects MIDI controls from your dynamic range expander, `dRE`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: expander

Get MIDI connection information

Syntax

```
connectionInfo = getMIDIConnections(dRE)
```

Description

`connectionInfo = getMIDIConnections(dRE)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your dynamic range expander, `dRE`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `dRE` that has established MIDI connections. Each substructure contains the control number, the device name of the corresponding MIDI control, and the property mapping information (mapping rule, minimum value, and maximum value).

Introduced in R2016a

isLocked

System object: expander

Locked status for input attributes and nontunable properties

Syntax

`L = isLocked(dRE)`

Description

`L = isLocked(dRE)` returns a logical value, `L`, that indicates whether input attributes and nontunable properties are locked for the dynamic range expander, `dRE`.

The `dRE` object performs an internal initialization the first time you execute `step`. The initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After the object is locked, the `isLocked` method returns a `true` value.

Introduced in R2016a

release

System object: expander

Enable property values and input characteristics to change

Syntax

```
release(dRE)
```

Description

`release(dRE)` releases system resources, such as memory, from your dynamic range expander, `dRE`. After you call `release`, all properties and input characteristics of `dRE` can change.

Note: Once you call `release` on a System object, subsequent calls to `setup`, `step`, `reset`, or `release` do not support code generation.

Introduced in R2016a

reset

System object: expander

Reset internal states of System object

Syntax

reset(dRE)

Description

reset(dRE) resets internal states of the dynamic range expander, dRE, to their initial values.

Introduced in R2016a

step

System object: expander

Perform dynamic range expansion

Syntax

```
y = step(dRE,x)
[y,g] = step(dRE,x)
```

Description

`y = step(dRE,x)` performs dynamic range expansion on the input signal, `x`, and returns the expanded signal, `y`. The type of dynamic range expansion is specified by the algorithm and properties of the `expander` System object, `dRE`.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

`[y,g] = step(dRE,x)` also returns the gain, in dB, applied at each input sample.

Introduced in R2016a

visualize

System object: expander

Visualize static expander characteristics of System object

Syntax

```
visualize(dRE)  
visualize(dRE,myInputRange)  
Y = visualize( ___ )
```

Description

`visualize(dRE)` plots the static expansion characteristic of the dynamic range expander, `dRE`. The method computes the dB output level for the input range `[-20:0.01:0]` dB.

`visualize(dRE,myInputRange)` enables you to specify the input range in dB. Specify `myInputRange` as a vector of ascending values.

`Y = visualize(___)` returns the dB output level, `Y`, corresponding to the input range. You can use any of the input arguments from previous syntaxes.

Introduced in R2016a

limiter System object

Dynamic range limiter

Description

The `limiter` System object performs brick-wall dynamic range limiting independently across each input channel. Dynamic range limiting suppresses the volume of loud sounds that cross a given threshold. It uses specified attack and release times to achieve a smooth applied gain curve. Properties of the `limiter` System object specify the type of dynamic range limiting.

To perform dynamic range limiting on your input:

- 1 Define and set up your dynamic range limiter. See “Construction” on page 3-128.
- 2 Call step to perform dynamic range limiting on each channel of the input signal according to the properties of your `limiter` object. The input must be a real-valued, double-precision or single-precision matrix. The `limiter` object treats each column of the input as an independent channel.

Construction

`dRL = limiter` creates a System object, `dRL`, that performs brick-wall dynamic range limiting independently across each input channel.

`dRL = limiter(thresholdValue)` sets the `Threshold` property to `thresholdValue`.

`dRL = limiter(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `dRL = limiter('AttackTime',0.01,'SampleRate',16000)` creates a System object, `dRL`, with `AttackTime` property set to 0.01 and `SampleRate` property set to 16000.

Properties

Threshold — Operation threshold (dB)

–10 (default) | real scalar

Operation threshold in dB, specified as a real scalar.

Operation threshold is the level above which gain is applied to the input signal.

This property is tunable. You can change the value of this property even when the object is locked.

KneeWidth — Knee width (dB)

0 (default) | real scalar

Knee width in dB, specified as a real scalar greater than or equal to 0.

Knee width is the transition area in the limiter characteristic.

For soft knee characteristics, the transition area is defined by the relation

$$y = x - \frac{\left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $(2 \times |x - T|) \leq W$.

- y is the output level in dB.
- x is the input level in dB.
- T is the threshold in dB.
- W is the knee width in dB.

This property is tunable. You can change the value of this property even when the object is locked.

AttackTime — Attack time (s)

0 (default) | real scalar

Attack time in seconds, specified as a real scalar greater than or equal to 0.

Attack time is the time it takes the limiter gain to rise from 10% to 90% of its final value when the input goes above the threshold.

This property is tunable. You can change the value of this property even when the object is locked.

ReleaseTime — Release time (s)

0.2 (default) | real scalar

Release time in seconds, specified as a real scalar greater than or equal to 0.

Release time is the time it takes the limiter gain to drop from 90% to 10% of its final value when the input goes below the threshold.

This property is tunable. You can change the value of this property even when the object is locked.

MakeUpGainMode — Make-up gain mode

'Auto' (default) | 'Property'

Make-up gain mode, specified as 'Auto' or 'Property'.

- 'Auto' — Make-up gain is applied at the output of the dynamic range limiter such that a steady-state 0 dB input has a 0 dB output.
- 'Property' — Make-up gain is set to the value specified in the `MakeUpGain` property.

MakeUpGain — Make-up gain (dB)

0 (default) | real scalar

Make-up gain in dB, specified as a real scalar.

Make-up gain compensates for gain lost during limiting. It is applied at the output of the dynamic range limiter. This property is available when you set `MakeUpGainMode` to 'Property'.

This property is tunable. You can change the value of this property even when the object is locked.

SampleRate — Input sample rate (Hz)

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

This property is tunable. You can change the value of this property even when the object is locked.

Methods

clone	Create copy of System object with same property values
configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
isLocked	Locked status for input attributes and nontunable properties
release	Enable property values and input characteristics to change
reset	Reset internal states of System object
step	Perform dynamic range limiting
visualize	Visualize static limiter characteristics of System object

Examples

Limit Audio Signal

Use dynamic range limiting to suppress the volume of loud sounds.

Set up the audio file reader and audio device writer System objects.

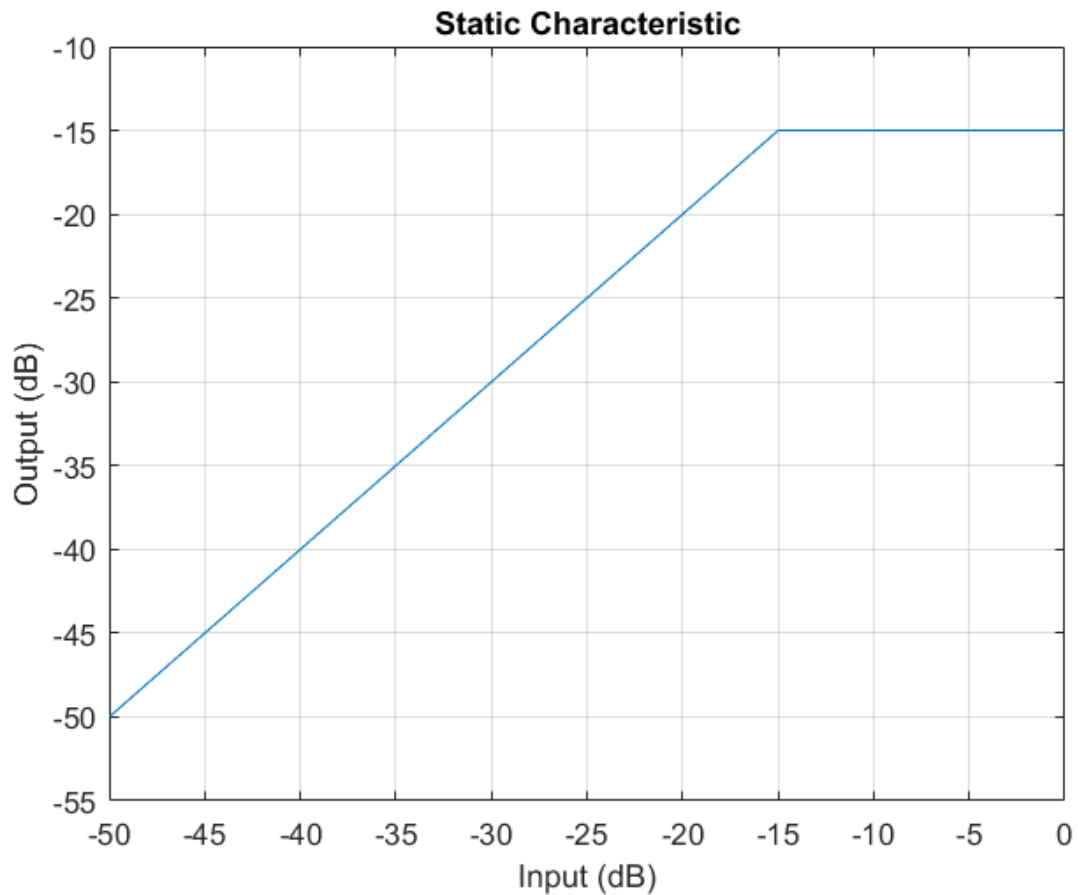
```
frameLength = 1024;
fileReader = dsp.AudioFileReader(...
    'Filename', 'RockDrums-44p1-stereo-11secs.mp3', ...
    'SamplesPerFrame', frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate', fileReader.SampleRate);
```

Set up the `limiter` to have a threshold of -15 dB, an attack time of 0.005 seconds, and a release time of 0.1 seconds. Set make-up gain to 0 dB (default). To specify this value, set the make-up gain mode to 'Property' but do not specify the `MakeUpGain` property. Use the sample rate of your audio file reader.

```
dRL = limiter(-15, ...
    'AttackTime', 0.005, ...
    'ReleaseTime', 0.1, ...
    'MakeUpGainMode', 'Property', ...
    'SampleRate', fileReader.SampleRate);
```

Visualize the static characteristic of the `limiter`.

```
visualize(dRL);
```

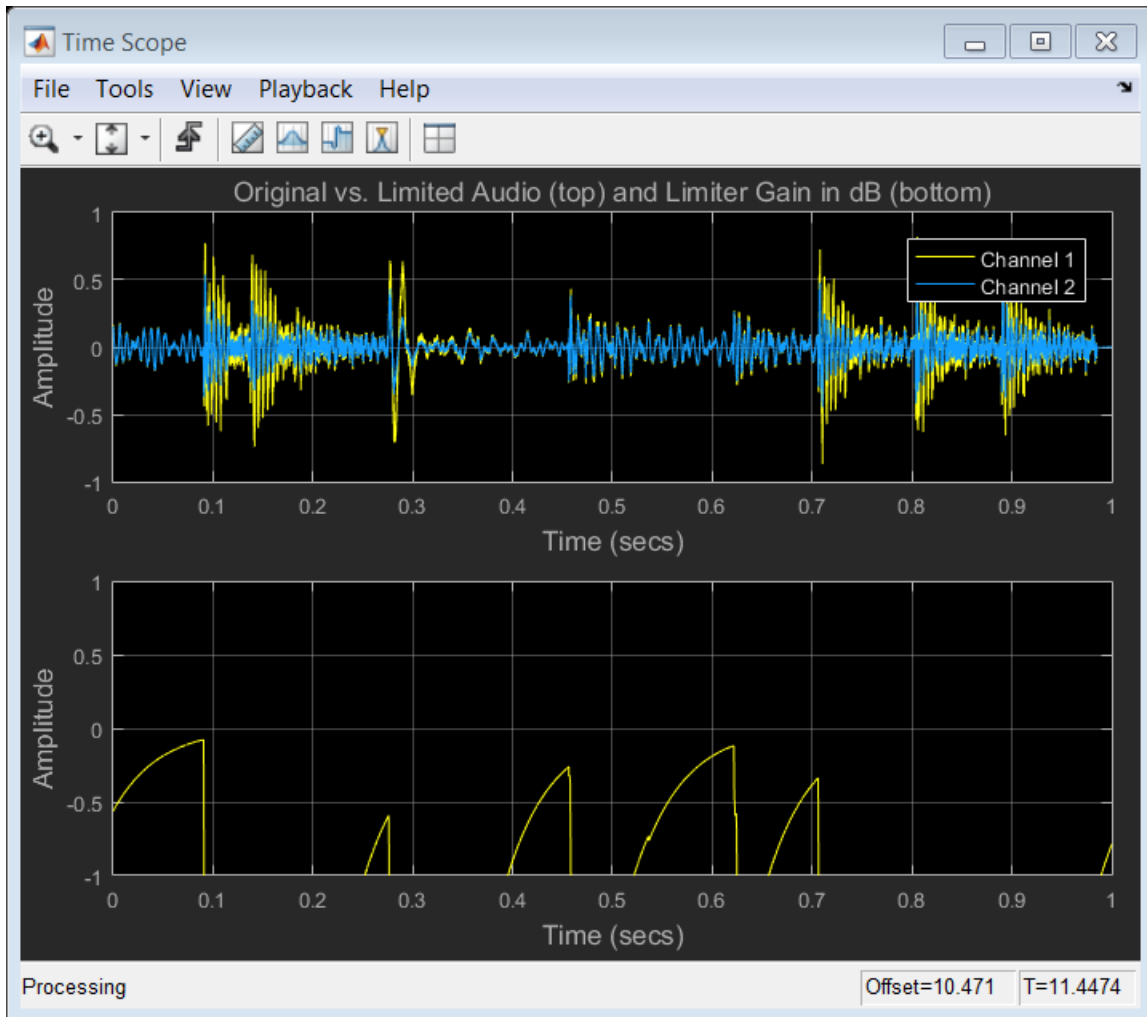
Set up a time scope to visualize the original signal and the limited signal.

```
scope = dsp.TimeScope(...  
    'SampleRate',fileReader.SampleRate,...  
    'TimeSpanOverrunAction','Scroll',...  
    'TimeSpan',1,...  
    'BufferLength',44100*4,...  
    'YLimits',[-1 1],...  
    'ShowGrid',true,...  
    'LayoutDimensions',[2,1],...  
    'NumInputPorts',2,...
```

```
'ShowLegend',true,...  
'Title',['Original vs. Limited Audio (top)'...  
' and Limiter Gain in dB (bottom)']];
```

Play the processed audio and visualize it on the scope.

```
while ~isDone(fileReader)  
    x = step(fileReader);  
    [y,g] = step(dRL,x);  
    step(deviceWriter,y);  
    x1 = x(:,1);  
    y1 = y(:,1);  
    g1 = g(:,1);  
    step(scope,[x1,y1],g1);  
end
```



See Also

noiseGate | compressor | expander | Limiter

Introduced in R2016a

clone

System object: limiter

Create copy of System object with same property values

Syntax

```
dRLclone = clone(dRL)
```

Description

`dRLclone = clone(dRL)` creates a dynamic range limiter System object, `dRLclone`, with the same property values as `dRL`. If the original object is locked, then `clone` creates a copy that is also locked. This copy has states initialized to the same values as the original. If the original object is not locked, then `clone` creates a new unlocked object with uninitialized states.

Introduced in R2016a

configureMIDI

System object: limiter

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(dRL)
configureMIDI(dRL,propName)
configureMIDI(dRL,propName,controlNumber)
configureMIDI(dRL,propName,controlNumber,'DeviceName',deviceName)
```

Description

`configureMIDI(dRL)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the dynamic range limiter System object, `dRL`, to MIDI controls of your choice.

`configureMIDI(dRL,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

`configureMIDI(dRL,propName,controlNumber)` makes the property respond to the MIDI control specified by `controlNumber`.

`configureMIDI(dRL,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `lLimiter` System object maps to a MIDI control with a specified range.

Property	Range	Unit
Threshold	-50 to 0	dB
KneeWidth	0 to 20	dB
AttackTime	0 to 4	seconds

Property	Range	Unit
ReleaseTime	0 to 4	seconds
MakeUpGain (available when you set MakeUpGainMode to 'Property')	-10 to 24	dB

Introduced in R2016a

createAudioPluginClass

System object: limiter

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(dRL)
createAudioPluginClass(dRL,pluginName)
```

Description

`createAudioPluginClass(dRL)` creates a System object plugin that implements the functionality of the `dynamicRangeLimiter` System object, `dRL`. The name of the created class is the `limiter` System object variable name followed by 'Plugin', for example, `dRLPlugin`.

Note: If the object is locked, the number of input and output channels of the plugin is equal to the number of channels of the object. Otherwise, the number of channels is equal to 2.

`createAudioPluginClass(dRL,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(dRL,'myLimiter')` creates a System object plugin with class name `myLimiter`.

Each tunable property of the `limiter` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Unit
Threshold	-50 to 0	dB
KneeWidth	0 to 20	dB
AttackTime	0 to 4	s

Property	Plugin Parameter Range	Unit
ReleaseTime	0 to 4	s
MakeUpGain (available when you set MakeUpGainMode to 'Property')	-10 to 24	dB

Introduced in R2016a

disconnectMIDI

System object: limiter

Disconnect MIDI controls from System object

Syntax

disconnectMIDI(dRL)

Description

disconnectMIDI(dRL) disconnects MIDI controls from your dynamic range limiter, dRL. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: limiter

Get MIDI connection information

Syntax

```
connectionInfo = getMIDIConnections(dRL)
```

Description

`connectionInfo = getMIDIConnections(dRL)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your dynamic range limiter, `dRL`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `dRL` that has established MIDI connections. Each substructure contains the control number, the device name of the corresponding MIDI control, and the property mapping information (mapping rule, minimum value, and maximum value).

Introduced in R2016a

isLocked

System object: limiter

Locked status for input attributes and nontunable properties

Syntax

`L = isLocked(dRL)`

Description

`L = isLocked(dRL)` returns a logical value, `L`, that indicates whether input attributes and nontunable properties are locked for the dynamic range limiter, `dRL`.

The `dRL` object performs an internal initialization the first time you execute `step`. The initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After the object is locked, the `isLocked` method returns a `true` value.

Introduced in R2016a

release

System object: limiter

Enable property values and input characteristics to change

Syntax

```
release(dRL)
```

Description

`release(dRL)` releases system resources, such as memory, from your dynamic range limiter, `dRL`. After you call `release`, all properties and input characteristics of `dRL` can change.

Note: Once you call `release` on a System object, subsequent calls to `setup`, `step`, `reset`, or `release` do not support code generation.

Introduced in R2016a

reset

System object: limiter

Reset internal states of System object

Syntax

reset(dRL)

Description

reset(dRL) resets internal states of the dynamic range limiter, dRL, to their initial values.

Introduced in R2016a

step

System object: limiter

Perform dynamic range limiting

Syntax

```
y = step(dRL,x)
[y,g] = step(dRL,x)
```

Description

`y = step(dRL,x)` performs dynamic range limiting on the input signal, `x`, and returns the limited signal, `y`. The type of dynamic range limiting is specified by the algorithm and properties of the `limiter` System object, `dRL`.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

`[y,g] = step(dRL,x)` also returns the gain, in dB, applied at each input sample.

Introduced in R2016a

visualize

System object: limiter

Visualize static limiter characteristics of System object

Syntax

```
visualize(dRL)  
visualize(dRL,myInputRange)  
Y = visualize( ___ )
```

Description

`visualize(dRL)` plots the static compression characteristic of the dynamic range limiter, `dRL`. The method computes the dB output level for the input range `[-50:0.01:0]` dB.

`visualize(dRL,myInputRange)` enables you to specify the input range in dB. Specify `myInputRange` as a vector of ascending values.

`Y = visualize(___)` returns the dB output level, `Y`, corresponding to the input range. You can use any of the input arguments from previous syntaxes.

Introduced in R2016a

multibandParametricEQ System object

Multiband parametric equalizer

Description

The `multibandParametricEQ` System object performs multiband parametric equalization independently across each channel of input using specified center frequencies, gains, and quality factors. You can configure the System object with up to 10 bands. You can add low-shelf and high-shelf filters, as well as highpass (low-cut) and lowpass (high-cut) filters.

To implement a multiband parametric equalizer:

- 1 Define and set up your multiband parametric equalizer. See “Construction” on page 3-148.
- 2 Call step to perform multiband parametric equalization on each channel of the input signal according to the properties of your `multibandParametricEQ` object. The input must be a real-valued, double-precision or single-precision matrix. The `multibandParametricEQ` object treats each column of the input as an independent channel.

Construction

`mPEQ = multibandParametricEQ` creates a System object, `mPEQ`, that performs multiband parametric equalization.

`mPEQ = multibandParametricEQ(Name, Value)` sets each construction argument or property `Name` to the specified `Value`. Unspecified properties and construction arguments have default values.

Example: `mPEQ = multibandParametricEQ('NumEQBands', 3, 'Frequencies', [300, 1200, 5000])` creates a multiband parametric equalizer System object, `mPEQ`, with `NumEQBands` set to 3 and the `Frequencies` property set to [300, 1200, 5000].

Note: The value specified by `NumEQBands` must be the length of the row vectors specified by `Frequencies`, `QualityFactors`, and `PeakGains`. During construction, the first property you specify locks the value.

Construction Arguments

NumEQBands — Number of equalizer bands

3 (default) | integer in the range 1 to 10

Number of equalizer bands, specified as an integer in the range 1 to 10. The number of equalizer bands does not include shelving filters, highpass filters, or lowpass filters.

`NumEQBands` must be set during construction. It cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('NumEQBands',5)` constructs a multiband parametric equalizer with 5 bands.

EQOrder — Order of individual equalizer bands

2 (default) | even integer

Order of individual equalizer bands, specified as an even integer. All equalizer bands have the same order.

`EQOrder` must be set during construction. It cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('EQOrder',6)` constructs a multiband parametric equalizer with the default 3 bands, all of order 6.

HasLowShelfFilter — Low-shelf filter toggle

false (default) | true

Low-shelf filter toggle, specified as `false` or `true`.

- `false` — Do not include low-shelf filter in multiband parametric equalizer implementation.
- `true` — Include low-shelf filter in multiband parametric equalizer implementation.

`HasLowpassFilter` must be set during construction. It cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('HasLowShelfFilter',true)` constructs a default multiband parametric equalizer with low-shelf filtering enabled.

HasHighShelfFilter — High-shelf filter toggle

false (default) | true

High-shelf filter toggle, specified as `false` or `true`.

- `false` — Do not include high-shelf filter in multiband parametric equalizer implementation.
- `true` — Include high-shelf filter in multiband parametric equalizer implementation.

`HasHighShelfFilter` must be set during construction. It cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('HasHighShelfFilter',true)` constructs a default multiband parametric equalizer with high-shelf filtering enabled.

HasLowpassFilter — Lowpass filter toggle

false (default) | true

Lowpass filter toggle, specified as `false` or `true`.

- `false` — Do not include lowpass filter in multiband parametric equalizer implementation.
- `true` — Include lowpass filter in multiband parametric equalizer implementation.

`HasLowpassFilter` must be set during construction. It cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('HasLowpassFilter',true)` constructs a default multiband parametric equalizer with lowpass filtering enabled.

HasHighpassFilter — Highpass filter toggle

false (default) | true

Highpass filter toggle, specified as `false` or `true`.

- `false` — Do not include highpass filter in multiband parametric equalizer implementation.
- `true` — Include highpass filter in multiband parametric equalizer implementation.

`HasHighpassFilter` must be set during construction. It cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('HasHighpassFilter',true)` constructs a default multiband parametric equalizer with highpass filtering enabled.

Oversample — Oversample toggle

`false` (default) | `true`

Oversample toggle, specified as `false` or `true`.

- `false` — Runs the multiband parametric equalizer at the input sample rate.
- `true` — Runs the multiband parametric equalizer at two times the input sample rate. Oversampling minimizes the frequency warping effects introduced by the bilinear transformation.

A halfband interpolator implements oversampling before equalization. A halfband decimator reduces the sample rate back to the input sampling rate after equalization.

Oversample must be set during construction. It cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('Oversample',true)` constructs a default multiband parametric equalizer with oversampling enabled.

Properties

Multiband Equalizer

Frequencies — Center frequencies of equalizer bands (Hz)

`[100,181,325]` (default) | row vector of length `NumEQBands`

Center frequencies of equalizer bands in Hz, specified as a row vector of length `NumEQBands`. The vector consists of real scalars in the range 0 to `SampleRate/2`.

This property is tunable. You can change the value of this property even when the object is locked.

QualityFactors — Quality factors of equalizer bands

`[1.6,1.6,1.6]` (default) | row vector of length `NumEQBands`

Quality factors of equalizer bands, specified as a row vector of length `NumEQBands`. The vector consists of real scalars in the range 0.2 to 700. Any values outside the range are saturated.

This property is tunable. You can change the value of this property even when the object is locked.

PeakGains — Peak or dip filter gains (dB)

[0,0,0] (default) | row vector of length NumEQBands

Peak or dip filter gains in dB, specified as a row vector of length NumEQBands. The vector consists of real scalars in the range $-\text{Inf}$ to 20. Values above 20 are saturated.

This property is tunable. You can change the value of this property even when the object is locked.

Low-shelf Filter

LowShelfCutoff — Low-shelf filter cutoff (Hz)

200 (default) | scalar

Low-shelf filter cutoff in Hz, specified as a scalar greater than or equal to 0.

This property is available when you set HasLowShelfFilter to true during construction.

This property is tunable. You can change the value of this property even when the object is locked.

LowShelfSlope — Low-shelf filter slope coefficient

1.5 (default) | real scalar in the range 0.1 to 5

Low-shelf filter slope coefficient, specified as a real scalar in the range 0.1 to 5. Values outside the range are saturated.

This property is available when you set HasLowShelfFilter to true during construction.

This property is tunable. You can change the value of this property even when the object is locked.

LowShelfGain — Low-shelf filter gain (dB)

0 (default) | real scalar in the range -12 to 12

Low-shelf filter gain in dB, specified as a real scalar in the range -12 to 12. Values outside the range are saturated.

This property is available when you set `HasLowShelfFilter` to `true` during construction.

This property is tunable. You can change the value of this property even when the object is locked.

High-Shelf Filter

HighShelfCutoff — High-shelf filter cutoff (Hz)

15000 (default) | nonnegative real scalar

High-shelf filter cutoff in Hz, specified as a real scalar greater than or equal to 0.

This property is available when you set `HasHighShelfFilter` to `true` during construction.

This property is tunable. You can change the value of this property even when the object is locked.

HighShelfSlope — High-shelf slope coefficient

1.5 (default) | real scalar in the range 0.1 to 5

High-shelf filter slope coefficient, specified as a real scalar in the range 0.1 to 5. Values outside the range are saturated.

This property is available when you set `HasHighShelfFilter` to `true` during construction.

This property is tunable. You can change the value of this property even when the object is locked.

HighShelfGain — High-shelf filter gain (dB)

0 (default) | real scalar in the range -12 to 12

High-shelf filter gain in dB, specified as a real scalar in the range -12 to 12. Values outside the range are saturated.

This property is available when you set `HasHighShelfFilter` to `true` during construction.

This property is tunable. You can change the value of this property even when the object is locked.

Lowpass Filter

LowpassCutoff — Lowpass filter cutoff frequency (Hz)

18000 (default) | nonnegative real scalar

Lowpass filter cutoff frequency in Hz, specified as a real scalar greater than or equal to 0.

This property is available when you set `HasLowpassFilter` to `true` during construction.

This property is tunable. You can change the value of this property even when the object is locked.

LowpassSlope — Lowpass filter slope (dB/octave)

12 (default) | real scalar in the range [0:6:48]

Lowpass filter slope in dB/octave, specified as a real scalar in the range [0:6:48]. Values that are not multiples of 6 are rounded.

This property is available when you set `HasLowpassFilter` to `true` during construction.

This property is tunable. You can change the value of this property even when the object is locked.

Highpass Filter

HighpassCutoff — Highpass filter cutoff frequency (Hz)

20 (default) | nonnegative real scalar

Highpass filter cutoff in Hz, specified as a real scalar greater than or equal to 0.

This property is available when you set `HasHighpassFilter` to `true` during construction.

This property is tunable. You can change the value of this property even when the object is locked.

HighpassSlope — Highpass filter slope (dB/octave)

30 (default) | real scalar in the range [0:6:48]

Highpass filter slope in dB/octave, specified as a real scalar in the range [0:6:48]. Values that are not multiples of 6 are rounded.

This property is available when you set `HasHighpassFilter` to `true` during construction.

This property is tunable. You can change the value of this property even when the object is locked.

Sampling

SampleRate — Input sample rate (Hz)

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

This property is tunable. You can change the value of this property even when the object is locked.

Methods

<code>clone</code>	Create copy of System object with same property values
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Enable property values and input characteristics to change
<code>reset</code>	Reset internal states of System object
<code>step</code>	Perform multiband parametric equalization
<code>visualize</code>	Visualize magnitude response of System object

Examples

Multiband Parametric Equalization

Create audio file reader and audio device writer System objects™. Use the sample rate of the reader as the sample rate of the writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
frameLength = 512;

fileReader = dsp.AudioFileReader(...
    'Filename','RockDrums-48-stereo-11secs.mp3',...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);

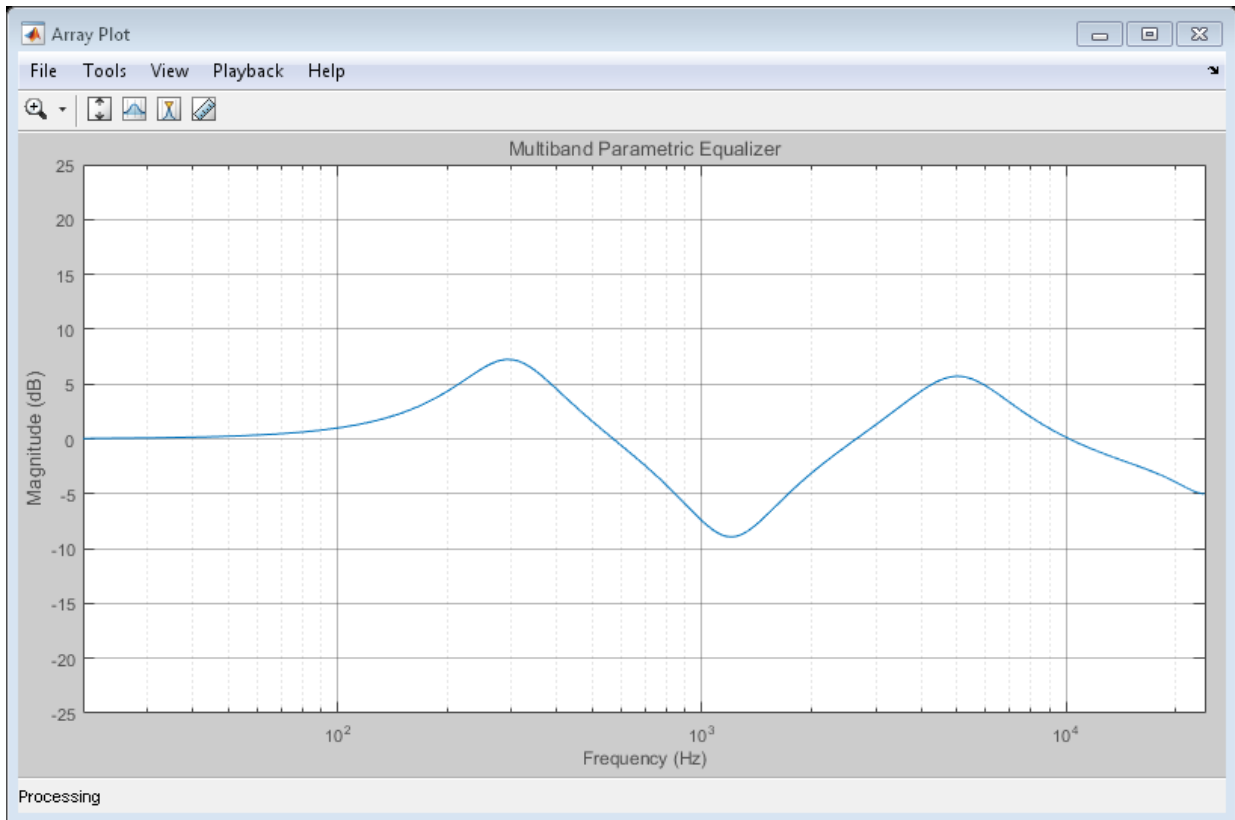
setup(deviceWriter,ones(frameLength,2));
```

Construct a three-band parametric equalizer with a high-shelf filter.

```
mPEQ = multibandParametricEQ(...
    'NumEQBands',3,...
    'Frequencies',[300,1200,5000],...
    'QualityFactors',[1,1,1],...
    'PeakGains',[8,-10,7],...
    'HasHighShelfFilter',true,...
    'HighShelfCutoff',14000,...
    'HighShelfSlope',0.3,...
    'HighShelfGain',-5,...
    'SampleRate',fileReader.SampleRate);
```

Visualize the magnitude frequency response of your multiband parametric equalizer.

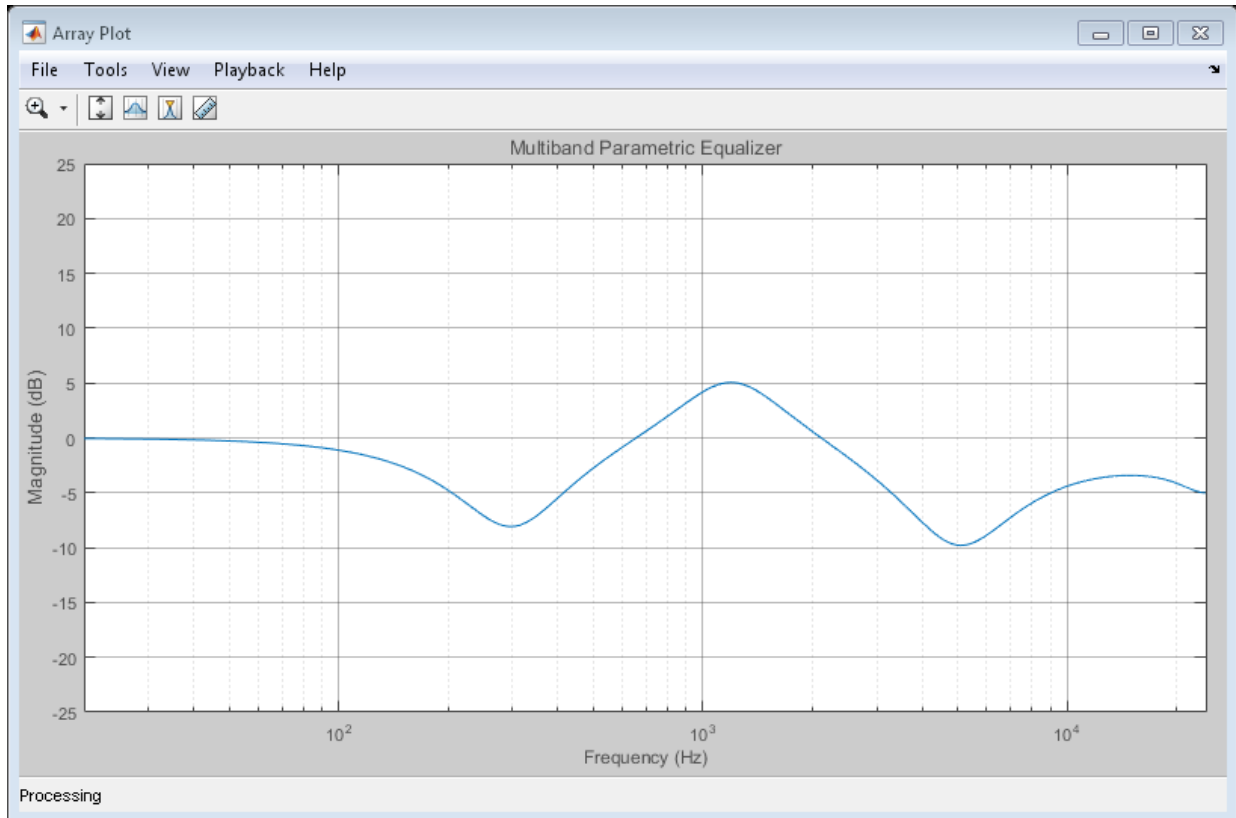
```
visualize(mPEQ);
```

Play the equalized audio signal. Update the peak gains of your equalizer band to hear the effect of the equalizer and visualize the changing magnitude response.

```
count = 0;
while ~isDone(fileReader)
    originalSignal = step(fileReader);
    equalizedSignal = step(mPEQ,originalSignal);
    play(deviceWriter,equalizedSignal);
    if mod(count,100) == 0
        mPEQ.PeakGains(1) = mPEQ.PeakGains(1) - 1.5;
        mPEQ.PeakGains(2) = mPEQ.PeakGains(2) + 1.5;
        mPEQ.PeakGains(3) = mPEQ.PeakGains(3) - 1.5;
    end
    count = count + 1;
    visualize(mPEQ)
```

end



Oversample Audio Signal

Reduce warping by specifying your `multibandParametricEQ` System object™ to perform oversampling before equalization.

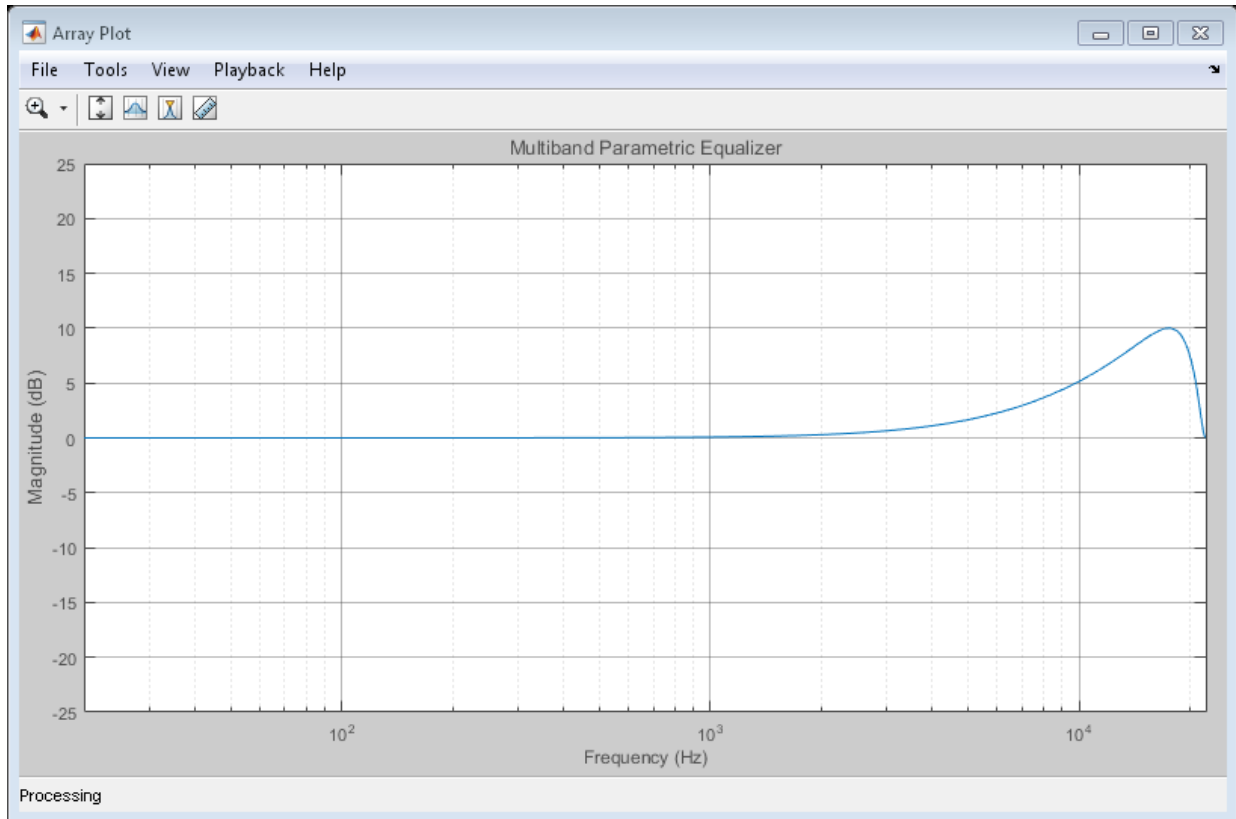
Create a one-band equalizer. Visualize the equalizer band as its center frequency approaches the Nyquist rate.

```
mPEQ = multibandParametricEQ(...  
    'NumEQBands',1,...  
    'Frequencies',9.5e3,...  
    'PeakGains',10);  
visualize(mPEQ)
```

```

for i = 1:1000
    mPEQ.Frequencies = mPEQ.Frequencies + 8;
end

```



The equalizer band is warped.

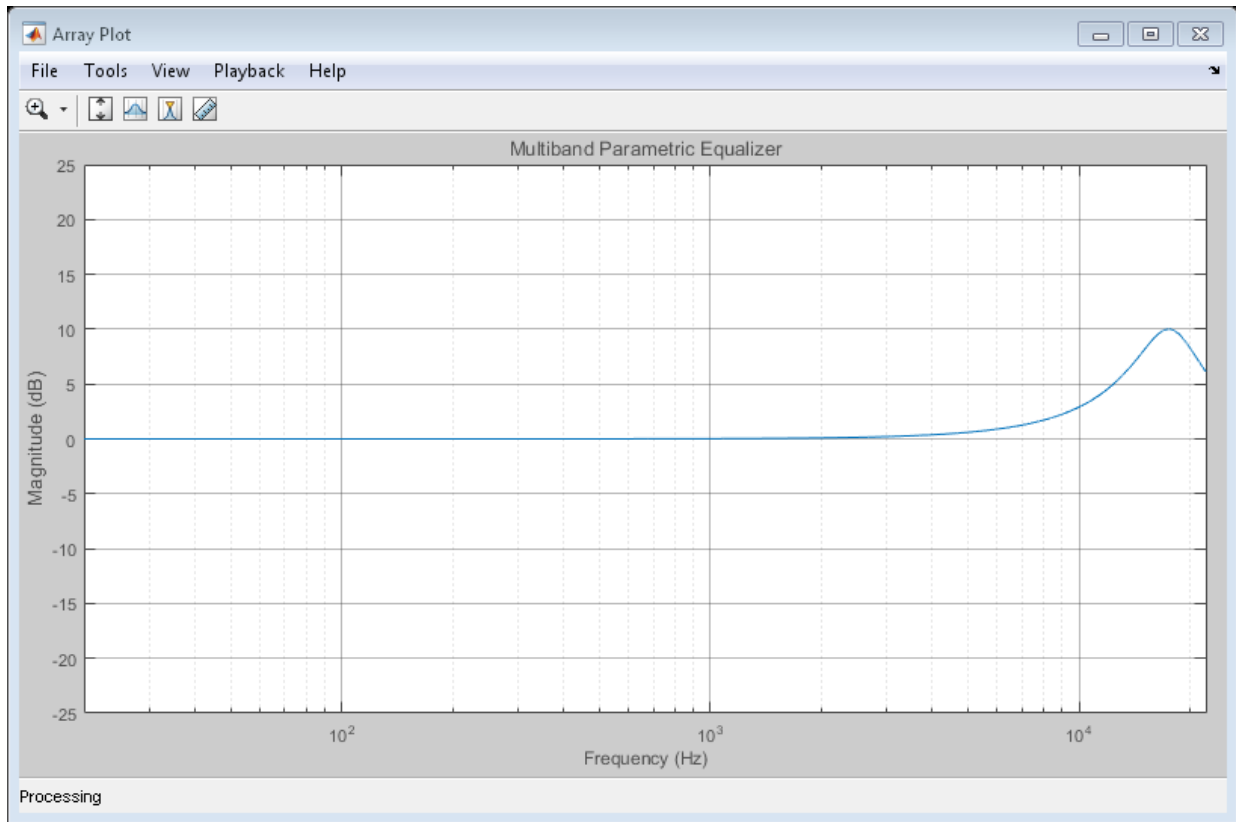
Create a one-band equalizer with `Oversample` set to `true`. Visualize the equalizer band as its center frequency approaches the Nyquist rate.

```

mPEQoversampled = multibandParametricEQ(...
    'NumEQBands', 1, ...
    'Frequencies', 9.5e3, ...
    'PeakGains', 10, ...
    'Oversample', true);
visualize(mPEQoversampled)

```

```
for i = 1:1000
    mPEQoversampled.Frequencies = mPEQoversampled.Frequencies + 8;
end
```



Warping is reduced.

See Also

[designParamEQ](#) | [designShelvingEQ](#) | [designVarSlopeFilter](#) | [Parametric EQ Filter](#)

Introduced in R2016a

clone

System object: multibandParametricEQ

Create copy of System object with same property values

Syntax

```
mPEQClone = clone(mPEQ)
```

Description

`mPEQClone = clone(mPEQ)` creates a `multibandParametricEQ` System object, `mPEQClone`, with the same property values as `mPEQ`. If the original object is locked, then `clone` creates a copy that is also locked. This copy has states initialized to the same values as the original. If the original object is not locked, then `clone` creates a new unlocked object with uninitialized states.

Introduced in R2016a

isLocked

System object: multibandParametricEQ

Locked status for input attributes and nontunable properties

Syntax

`L = isLocked(mPEQ)`

Description

`L = isLocked(mPEQ)` returns a logical value, `L`, that indicates whether input attributes and nontunable properties are locked for the multiband parametric equalizer, `mPEQ`.

The `mPEQ` object performs an internal initialization the first time you execute `step`. The initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After the object is locked, the `isLocked` method returns a `true` value.

Introduced in R2016a

release

System object: multibandParametricEQ

Enable property values and input characteristics to change

Syntax

```
release(mPEQ)
```

Description

`release(mPEQ)` releases system resources, such as memory, from your multiband parametric equalizer, `mPEQ`. After you call **release**, all properties and input characteristics of `mPEQ` can change.

Note: Once you call **release** on a System object, subsequent calls to **setup**, **step**, **reset**, or **release** do not support code generation.

Introduced in R2016a

reset

System object: multibandParametricEQ

Reset internal states of System object

Syntax

`reset(mPEQ)`

Description

`reset(mPEQ)` resets internal states of the multiband parametric equalizer, `mPEQ`, to their initial values.

Introduced in R2016a

step

System object: multibandParametricEQ

Perform multiband parametric equalization

Syntax

```
y = step(mPEQ,x)
```

Description

`y = step(mPEQ,x)` performs multiband parametric equalization on the input signal, `x`, and returns the filtered signal, `y`. The type of equalization is specified by the algorithm and properties of the `multibandParametricEQ` System object, `mPEQ`.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

visualize

System object: multibandParametricEQ

Visualize magnitude response of System object

Syntax

```
visualize(mPEQ)  
visualize(mPEQ,N)
```

Description

`visualize(mPEQ)` plots the magnitude response of the multiband parametric equalizer. The plot includes any enabled shelving filters, lowpass filters, or highpass filters.

`visualize(mPEQ,N)` specifies an N-point FFT used to calculate the magnitude response. The default is 2048.

Introduced in R2016a

noiseGate System object

Dynamic range gate

Description

The `noiseGate` System object performs dynamic range gating independently across each input channel. Dynamic range gating suppresses signals below a given threshold. It uses specified attack, release, and hold times to achieve a smooth applied gain curve. Properties of the `noiseGate` System object specify the type of dynamic range gating.

To perform dynamic range gating on your input:

- 1 Define and set up your dynamic range gate. See “Construction” on page 3-167.
- 2 Call step to perform dynamic range gating on each channel of the input signal according to the properties of your `noiseGate` object. The input must be a real-valued, double-precision or single-precision matrix. The `noiseGate` object treats each column of the input as an independent channel.

Construction

`dRG = noiseGate` creates a System object, `dRG`, that performs dynamic range gating independently across each input channel.

`dRG = noiseGate(thresholdValue)` sets the `Threshold` property to `thresholdValue`.

`dRG = noiseGate(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `dRG = noiseGate('AttackTime',0.01,'SampleRate',16000)` creates a System object, `dRG`, with the `AttackTime` property set to `0.01`, and the `SampleRate` property set to `16000`.

Properties

Threshold — Operation threshold (dB)

-10 (default) | real scalar

Operation threshold in dB, specified as a real scalar.

Operation threshold is the level below which gain is applied to the input signal.

This property is tunable. You can change the value of this property even when the object is locked.

AttackTime — Attack time (s)

0.05 (default) | real scalar

Attack time in seconds, specified as a real scalar greater than or equal to 0.

Attack time is the time it takes the applied gain to rise from 10% to 90% of its final value when the input goes below the threshold.

This property is tunable. You can change the value of this property even when the object is locked.

ReleaseTime — Release time (s)

0.02 (default) | real scalar

Release time in seconds, specified as a real scalar greater than or equal to 0.

Release time is the time it takes the applied gain to drop from 90% to 10% of its final value when the input goes above the threshold.

This property is tunable. You can change the value of this property even when the object is locked.

HoldTime — Hold time (s)

0.05 (default) | real finite scalar

Hold time in seconds, specified as a real scalar greater than or equal to 0.

Hold time is the period in which the applied gain is held constant before it starts moving toward its steady-state value. Hold time begins when the input level crosses the operation threshold.

This property is tunable. You can change the value of this property even when the object is locked.

SampleRate — Input sample rate (Hz)

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

This property is tunable. You can change the value of this property even when the object is locked.

Methods

clone	Create copy of System object with same property values
configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
isLocked	Locked status for input attributes and nontunable properties
release	Enable property values and input characteristics to change
reset	Reset internal states of System object
step	Perform dynamic range gating
visualize	Visualize static gate characteristics of System object

Examples

Gate Audio Signal

Use dynamic range gating to attenuate background noise from an audio signal.

Set up the audio file reader and audio device writer System objects.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader(...
    'Filename','Counting-16-44p1-mono-15secs.wav',...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
```

Corrupt the audio signal with Gaussian noise. Play the audio.

```
while ~isDone(fileReader)
    x = step(fileReader);
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);
    step(deviceWriter,xCorrupted);
end

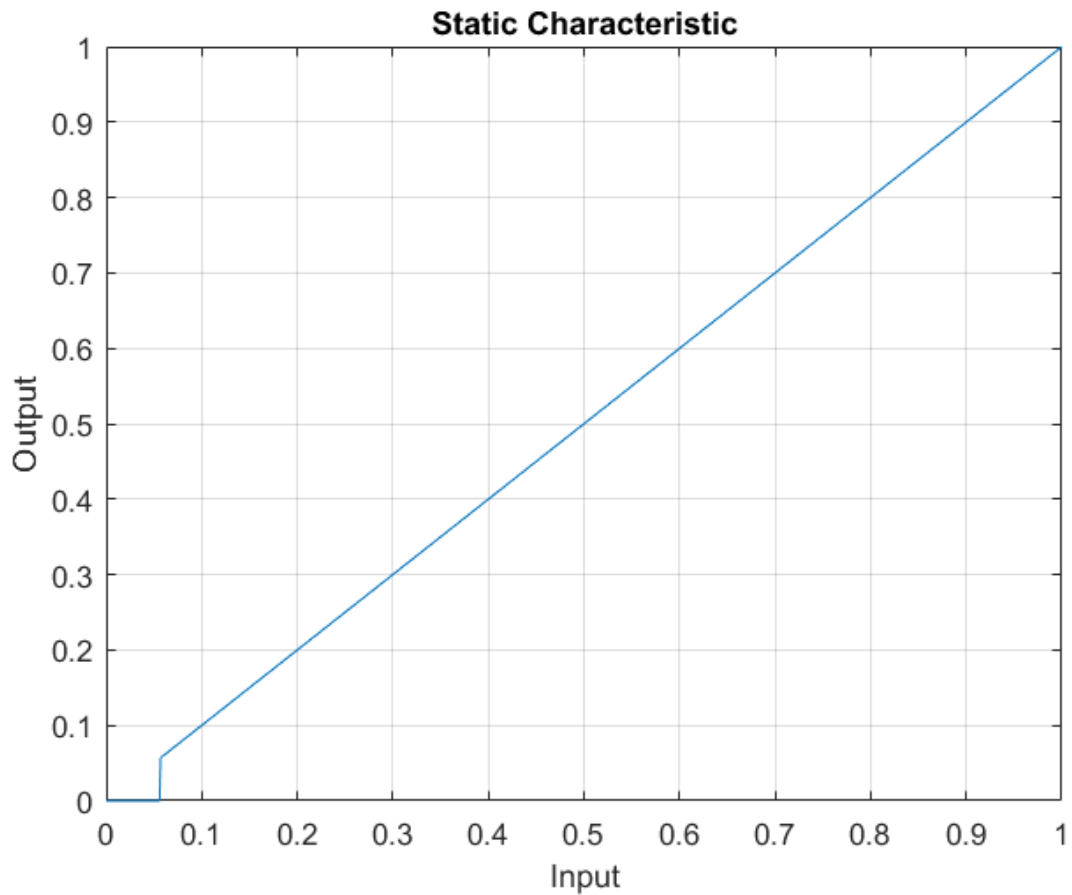
release(fileReader);
```

Set up a dynamic range gate with a threshold of -25 dB, an attack time of 0.01 seconds, a release time of 0.02 seconds, and a hold time of 0 seconds. Use the sample rate of your audio file reader.

```
gate = noiseGate(-25,...
    'AttackTime',0.01,...
    'ReleaseTime',0.02,...
    'HoldTime',0,...
    'SampleRate',fileReader.SampleRate);
```

Visualize the static characteristic of the gate.

```
visualize(gate);
```

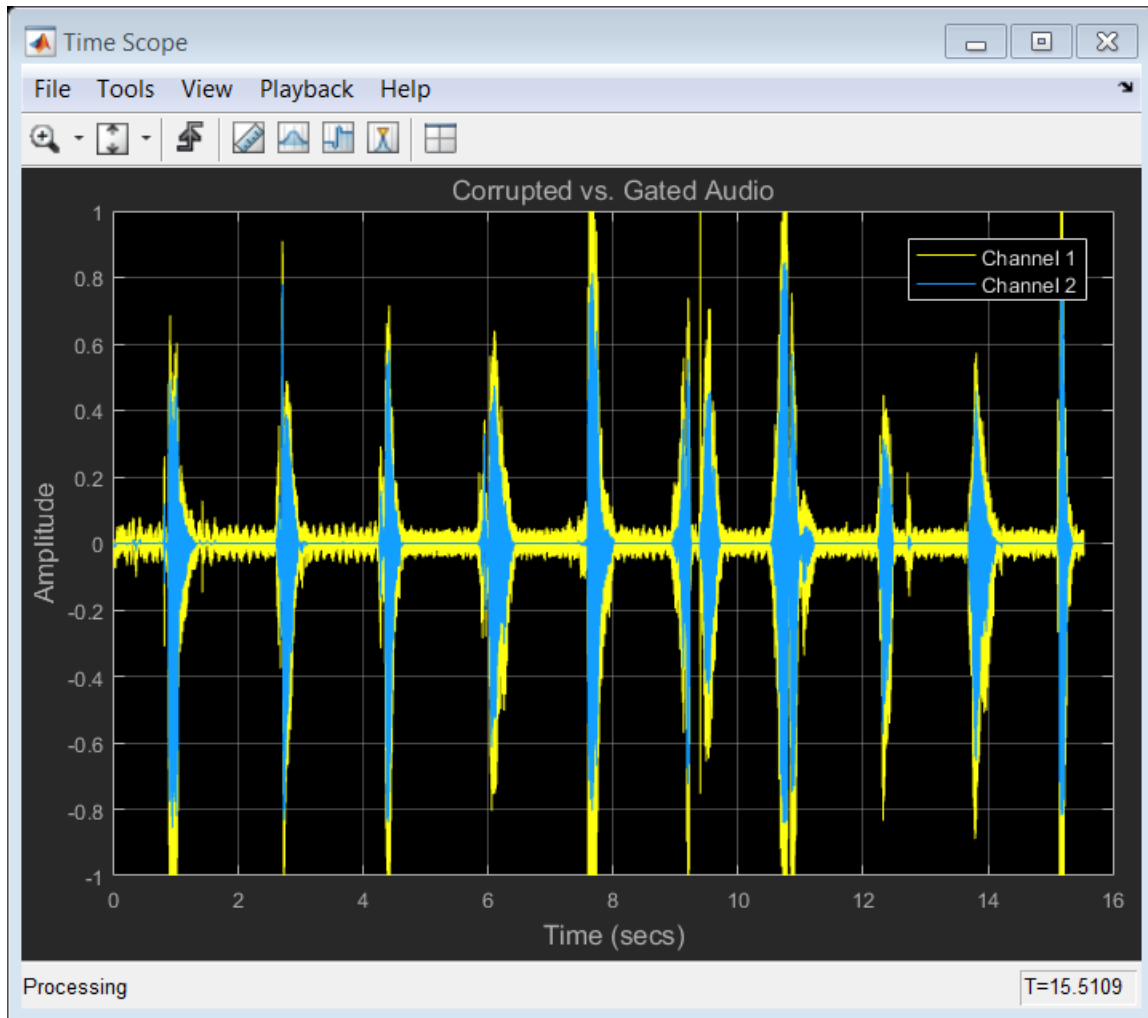


Set up a time scope to visualize the signal before and after dynamic range gating.

```
scope = dsp.TimeScope(...  
    'SampleRate',fileReader.SampleRate,...  
    'TimeSpanOverrunAction','Scroll',...  
    'TimeSpan',16,...  
    'BufferLength',1.5e6,...  
    'YLimits',[-1 1],...  
    'ShowGrid',true,...  
    'ShowLegend',true,...  
    'Title','Corrupted vs. Gated Audio');
```

Play the processed audio and visualize it on scope.

```
while ~isDone(fileReader)
    x = step(fileReader) + (1e-2/4)*randn(frameLength,1);
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);
    y = step(gate,xCorrupted);
    step(deviceWriter,y);
    step(scope,[xCorrupted,y]);
end
```

See Also

expander | compressor | limiter | Noise Gate

Introduced in R2016a

clone

System object: noiseGate

Create copy of System object with same property values

Syntax

```
dRGclone = clone(dRG)
```

Description

`dRGclone = clone(dRG)` creates a dynamic range gate System object, `dRGclone`, with the same property values as `dRG`. If the original object is locked, then `clone` creates a copy that is also locked. This copy has states initialized to the same values as the original. If the original object is not locked, then `clone` creates a new unlocked object with uninitialized states.

Introduced in R2016a

configureMIDI

System object: noiseGate

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(dRG)
configureMIDI(dRG,propName)
configureMIDI(dRG,propName,controlNumber)
configureMIDI(dRG,propName,controlNumber,'DeviceName',deviceName)
```

Description

`configureMIDI(dRG)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the dynamic range gate System object, `dRG`, to MIDI controls of your choice.

`configureMIDI(dRG,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

`configureMIDI(dRG,propName,controlNumber)` makes the property respond to the MIDI control specified by `controlNumber`.

`configureMIDI(dRG,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `noiseGate` System object maps to MIDI controls with a specified range.

Property	Range	Unit
Threshold	-140 to 0	dB
AttackTime	0 to 4	seconds
ReleaseTime	0 to 4	seconds

Property	Range	Unit
HoldTime	0 to 4	seconds

Introduced in R2016a

createAudioPluginClass

System object: noiseGate

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(dRG)
createAudioPluginClass(dRG,pluginName)
```

Description

`createAudioPluginClass(dRG)` creates a System object plugin that implements the functionality of the `noiseGate` System object, `dRG`. The name of the created class is the `noiseGate` System object variable name followed by 'Plugin', for example, `dRGPlugin`.

Note: If the object is locked, the number of input and output channels of the plugin is equal to the number of channels of the object. Otherwise, the number of channels is equal to 2.

`createAudioPluginClass(dRG,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(dRG,'myGate')` creates a System object plugin with class name `myGate`.

Each tunable property of the `noiseGate` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Unit
Threshold	-140 to 0	dB
AttackTime	0 to 4	s
ReleaseTime	0 to 4	s

Property	Plugin Parameter Range	Unit
HoldTime	0 to 4	s

Introduced in R2016a

disconnectMIDI

System object: noiseGate

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(dRG)
```

Description

`disconnectMIDI(dRG)` disconnects MIDI controls from your dynamic range gate, `dRG`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: noiseGate

Get MIDI connection information

Syntax

```
connectionInfo = getMIDIConnections(dRG)
```

Description

`connectionInfo = getMIDIConnections(dRG)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your dynamic range gate, `dRG`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `dRG` that has established MIDI connections. Each substructure contains the control number, the device name of the corresponding MIDI control, and the property mapping information (mapping rule, minimum value, and maximum value).

Introduced in R2016a

isLocked

Locked status for input attributes and nontunable properties

Syntax

```
L = isLocked(dRG)
```

Description

`L = isLocked(dRG)` returns a logical value, `L`, that indicates whether input attributes and nontunable properties are locked for the dynamic range gate, `dRG`.

The `dRG` object performs an internal initialization the first time you execute `step`. The initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After the object is locked, the `isLocked` method returns a `true` value.

Introduced in R2016a

release

System object: noiseGate

Enable property values and input characteristics to change

Syntax

```
release(dRG)
```

Description

`release(dRG)` releases system resources, such as memory, from your dynamic range gate, `dRG`. After you call `release`, all properties and input characteristics of `dRG` can change.

Note: Once you call `release` on a System object, subsequent calls to `setup`, `step`, `reset`, or `release` do not support code generation.

Introduced in R2016a

reset

System object: noiseGate

Reset internal states of System object

Syntax

reset(dRG)

Description

reset(dRG) resets internal states of the dynamic range gate, dRG, to their initial values.

Introduced in R2016a

step

System object: noiseGate

Perform dynamic range gating

Syntax

```
y = step(dRG,x)
[y,g] = step(dRG,x)
```

Description

`y = step(dRG,x)` performs dynamic range gating on the input signal, `x`, and returns the gated signal, `y`. The type of dynamic range gating is specified by the algorithm and properties of the `noiseGate` System object, `dRG`.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

`[y,g] = step(dRG,x)` also returns the gain, in dB, applied at each input sample.

Introduced in R2016a

visualize

System object: noiseGate

Visualize static gate characteristics of System object

Syntax

```
visualize(dRG)  
visualize(dRG,myInputRange)  
Y = visualize( ___ )
```

Description

`visualize(dRG)` plots the static gate characteristics of the dynamic range gate, `dRG`. The method computes the dB output level for the input range `[0:0.001:1]` dB.

`visualize(dRG,myInputRange)` enables you to specify the input range in dB. Specify `myInputRange` as a vector of ascending values.

`Y = visualize(___)` returns the dB output level, `Y`, corresponding to the input range. You can use any of the input arguments from previous syntaxes.

Introduced in R2016a

reverberator System object

Add reverberation to audio signal

Description

The `reverberator` System object adds reverberation to mono or stereo audio signals. Properties of the `reverberator` System object specify the “Reverberation Model” on page 3-192 used.

To add reverberation to your input:

- 1 Define and set up your reverberator. See “Construction” on page 3-186.
- 2 Call step to add reverberation to the input signal according to the properties of your `reverberator` object. The input must be a real-valued, double-precision or single-precision matrix. The input matrix must have one or two columns, corresponding to a mono or stereo signal, respectively. The output matrix always has two columns (stereo).

Construction

`reverb = reverberator` creates a System object, `reverb`, that adds artificial reverberation to an audio signal.

`reverb = reverberator(Name, Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `reverb = reverberator('PreDelay', 0.5, 'WetDryMix', 1)` creates a System object, `reverb`, with the `PreDelay` property set to 0.5 and the `WetDryMix` property set to 1.

Properties

PreDelay — Pre-delay for reverberation (s)

0 (default) | real positive scalar

Pre-delay for reverberation in seconds, specified as a real scalar in the range 0 to 1.

Pre-delay for reverberation is the time between hearing direct sound and the first early reflection. The value of `PreDelay` is proportional to the size of the room being modeled.

This property is tunable. You can change the value of this property even when the object is locked.

HighCutFrequency — Lowpass filter cutoff (Hz)

20000 (default) | real positive scalar

Lowpass filter cutoff in Hz, specified as a real positive scalar in the range 0 to

$$\left(\frac{\text{SampleRate}}{2} \right).$$

Lowpass filter cutoff is the -3 dB cutoff frequency for the single-pole lowpass filter at the front of the reverberator structure. It prevents the application of reverberation to high-frequency components of the input.

This property is tunable. You can change the value of this property even when the object is locked.

Diffusion — Density of reverb tail

0.5 (default) | real positive scalar

Density of reverb tail, specified as a real positive scalar in the range 0 to 1.

`Diffusion` is proportional to the rate at which the reverb tail builds in density. Increasing `Diffusion` pushes the reflections closer together, thickening the sound. Reducing `Diffusion` creates more discrete echoes.

This property is tunable. You can change the value of this property even when the object is locked.

DecayFactor — Decay factor of reverb tail

0.5 (default) | real positive scalar

Decay factor of reverb tail, specified as a real positive scalar in the range 0 to 1.

`DecayFactor` is proportional to the time it takes for reflections to run out of energy. To model a large room, use a long reverb tail (low decay factor). To model a small room, use a short reverb tail (high decay factor).

This property is tunable. You can change the value of this property even when the object is locked.

HighFrequencyDamping — High-frequency damping

0.0005 (default) | real scalar

High-frequency damping, specified as a real positive scalar in the range 0 to 1.

`HighFrequencyDamping` is proportional to the attenuation of high frequencies in the reverberation output. Setting `HighFrequencyDamping` to a large value makes high-frequency reflections decay faster than low-frequency reflections.

This property is tunable. You can change the value of this property even when the object is locked.

WetDryMix — Wet-dry mix

0.3 (default) | real scalar

Wet-dry mix, specified as a real positive scalar in the range 0 to 1.

Wet-dry mix is the ratio of wet (reverberated) to dry (original) signal that your reverberator System object outputs.

This property is tunable. You can change the value of this property even when the object is locked.

SampleRate — Input sample rate (Hz)

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

This property is tunable. You can change the value of this property even when the object is locked.

Methods

clone

Create copy of System object with same property values

configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
isLocked	Locked status for input attributes and nontunable properties
release	Enable property values and input characteristics to change
reset	Reset internal states of System object
step	Add artificial reverberation

Examples

Add Reverberation to Audio Signal

Use the `reverberator` System object™ to add artificial reverberation to an audio signal read from a file.

Construct the audio file reader and audio device writer System objects. Use the sample rate of the reader as the sample rate of the writer.

```
fileReader = dsp.AudioFileReader(...
    'FunkyDrums-44p1-stereo-25secs.mp3',...
    'SamplesPerFrame',1024);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
```

Play 10 seconds of the audio signal through your device.

```
tic
while toc < 10
    audio = step(fileReader);
    play(deviceWriter,audio);
end
```

```
release(fileReader);
```

Construct a `reverberator` System object with default settings.

```
reverb = reverberator
```

```
reverb =
```

```
    reverberator with properties:
```

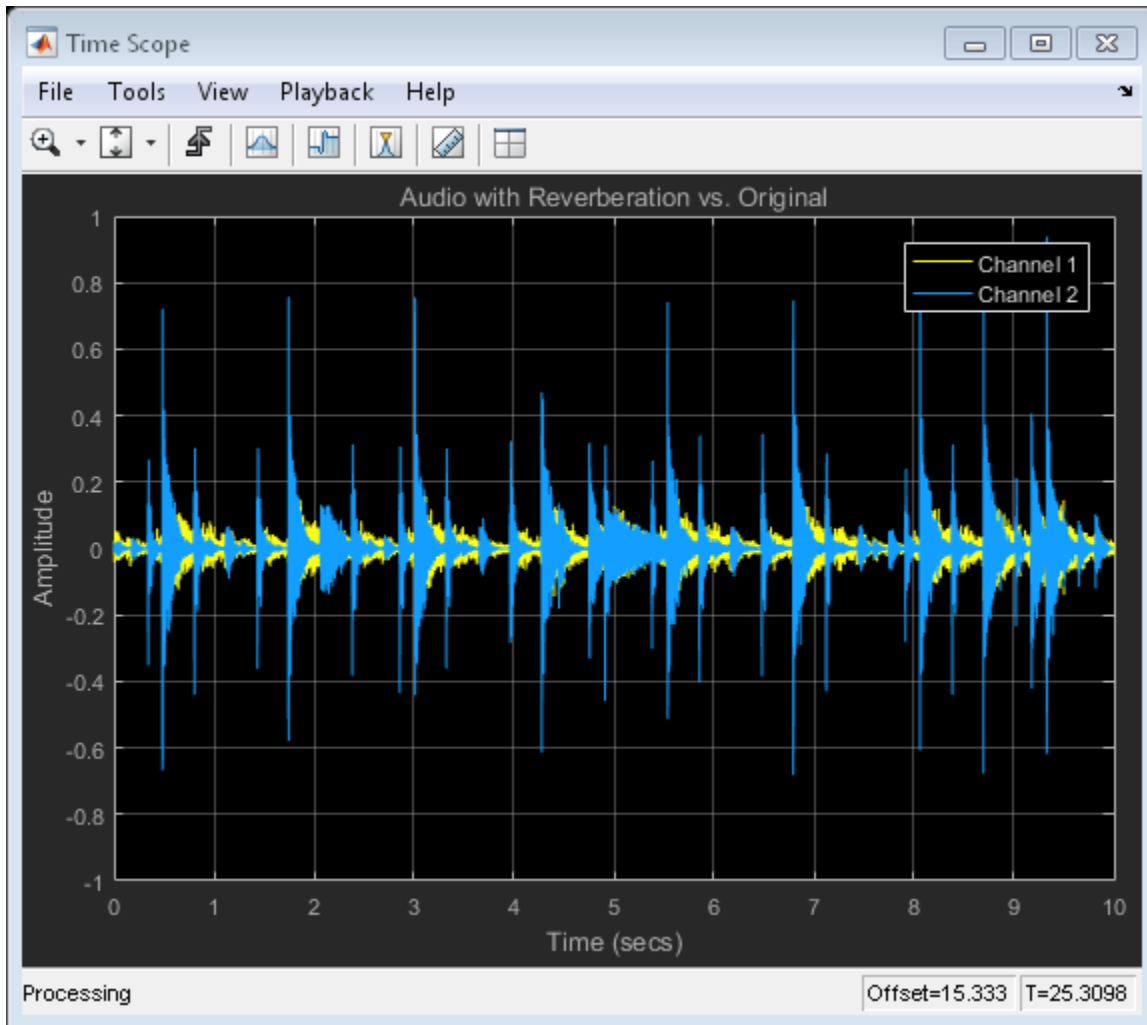
```
        PreDelay: 0
    HighCutFrequency: 20000
        Diffusion: 0.5000
        DecayFactor: 0.5000
    HighFrequencyDamping: 5.0000e-04
        WetDryMix: 0.3000
        SampleRate: 44100
```

Construct a time scope to visualize the original audio signal and the audio signal with added artificial reverberation.

```
scope = dsp.TimeScope(...
    'SampleRate',fileReader.SampleRate,...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpan',10,...
    'BufferLength',1.5e6,...
    'YLimits',[-1,1],...
    'ShowGrid',true,...
    'ShowLegend',true,...
    'Title','Audio with Reverberation vs. Original');
```

Play the audio signal with artificial reverberation. Visualize the audio with reverberation and the original audio.

```
while ~isDone(fileReader)
    audio = step(fileReader);
    audioWithReverb = step(reverb,audio);
    step(deviceWriter,audioWithReverb);
    step(scope,[audioWithReverb(:,1),audio(:,1)]);
end
```



Algorithms

The algorithm to add reverberation is based on the plate-class reverberation topology described in [1].

Definitions

Reverberation Model

Reverberation refers to the buildup and decay of reflected audio waves in a given space. Reverberation models are used in digital environments to mimic the physical effect of reverberation.

References

- [1] Dattorro, Jon. "Effect Design, Part 1: Reverberator and Other Filters." *Journal of the Audio Engineering Society*. Vol. 45, Issue 9, pp. 660–684.

See Also

`generateAudioPlugin` | `Reverberator` | `validateAudioPlugin`

Introduced in R2016a

clone

System object: reverberator

Create copy of System object with same property values

Syntax

```
reverbClone = clone(reverb)
```

Description

`reverbClone = clone(reverb)` creates a reverberator System object, `reverbClone`, with the same property values as `reverb`. If the original object is locked, then `clone` creates a copy that is also locked. This copy has states initialized to the same values as the original. If the original object is not locked, then `clone` creates a new unlocked object with uninitialized states.

Introduced in R2016a

configureMIDI

System object: reverberator

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(reverb)
configureMIDI(reverb,propName)
configureMIDI(reverb,propName,controlNumber)
configureMIDI(reverb,propName,controlNumber,'DeviceName',deviceName)
```

Description

`configureMIDI(reverb)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the `reverberator` System object, `reverb`, to MIDI controls of your choice.

`configureMIDI(reverb,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

`configureMIDI(reverb,propName,controlNumber)` makes the property respond to the MIDI control specified by `controlNumber`.

`configureMIDI(reverb,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `reverberator` System object maps to MIDI controls with a specified range.

Property	Range	Unit
PreDelay	0 to 1	s
HighCutFrequency	20 to 20,000 (log scale)	Hz
Diffusion	0 to 1	none

Property	Range	Unit
DecayFactor	0 to 1	none
HighFrequencyDamping	0 to 1	none
WetDryMix	0 to 1	none

Introduced in R2016a

createAudioPluginClass

System object: reverberator

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(reverb)  
createAudioPluginClass(reverb,pluginName)
```

Description

`createAudioPluginClass(reverb)` creates a System object plugin that implements the functionality of the reverberator System object, `reverb`. The name of the created class is the reverberator System object variable name followed by 'Plugin', for example, `reverbPlugin`.

Note: If the object is locked, the number of input channels of the plugin is equal to the number of channels of the object. Otherwise, the number of channels is equal to 2. The number of output channels of the plugin is always equal to 2.

`createAudioPluginClass(reverb,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(reverb, 'concertHall')` creates a System object plugin with class name `concertHall`.

Each tunable property of the reverberator System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Unit
PreDelay	0 to 1	s
HighCutFrequency	20 to 20,000 (log scale)	Hz
Diffusion	0 to 1	none

Property	Plugin Parameter Range	Unit
DecayFactor	0 to 1	none
HighFrequencyDamping	0 to 1	none
WetDryMix	0 to 1	none

Introduced in R2016a

disconnectMIDI

System object: reverberator

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(reverb)
```

Description

`disconnectMIDI(reverb)` disconnects MIDI controls from your reverberator, `reverb`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: reverberator

Get MIDI connection information

Syntax

```
connectionInfo = getMIDIConnections(reverb)
```

Description

`connectionInfo = getMIDIConnections(reverb)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your reverberator, `reverb`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `reverb` that has established MIDI connections. Each substructure contains the control number, the device name of the corresponding MIDI control, and the property mapping information (mapping rule, minimum value, and maximum value).

Introduced in R2016a

isLocked

System object: reverberator

Locked status for input attributes and nontunable properties

Syntax

```
L = isLocked(reverb)
```

Description

`L = isLocked(reverb)` returns a logical value, `L`, that indicates whether input attributes and nontunable properties are locked for the reverberator, `reverb`.

The `reverb` object performs an internal initialization the first time you execute `step`. The initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After the object is locked, the `isLocked` method returns a `true` value.

Introduced in R2016a

release

System object: reverberator

Enable property values and input characteristics to change

Syntax

```
release(reverb)
```

Description

`release(reverb)` releases system resources, such as memory, from your reverberator, `reverb`. After you call `release`, all properties and input characteristics of `reverb` can change.

Note: Once you call `release` on a System object, subsequent calls to `setup`, `step`, `reset`, or `release` do not support code generation.

Introduced in R2016a

reset

System object: reverberator

Reset internal states of System object

Syntax

`reset(reverb)`

Description

`reset(reverb)` resets internal states of the reverberator, `reverb`, to their initial values.

Introduced in R2016a

step

System object: reverberator

Add artificial reverberation

Syntax

```
y = step(reverb,x)
```

Description

`y = step(reverb,x)` adds artificial reverberation to the input signal, `x`, and returns the mixed signal, `y`. The type of reverberation is specified by the algorithm and properties of the reverberator System object, `reverb`.

`x` must be a real-valued, double-precision or single-precision matrix with one or two columns. The output is always a stereo signal (two columns).

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

wavetableSynthesizer System object

Generate periodic signals from single-cycle waveforms

Description

The `wavetableSynthesizer` System object generates a periodic signal with tunable properties. The periodic signal is defined by a single-cycle waveform cached as the `Wavetable` property of your `wavetableSynthesizer` object.

To generate a periodic signal:

- 1 Define and set up your wavetable synthesizer. See “Construction” on page 3-204.
- 2 Call step to generate a signal according to the properties of your `wavetableSynthesizer` object. The object has internal memory suited to frame-based processing.

Construction

`waveSynth = wavetableSynthesizer` creates a wavetable synthesizer System object, `waveSynth`, with default property values.

`waveSynth = wavetableSynthesizer(wavetableValue)` sets the `Wavetable` property to `wavetableValue`.

`waveSynth = wavetableSynthesizer(wavetableValue, frequencyValue)` sets the `Frequency` property to `frequencyValue`.

`waveSynth = wavetableSynthesizer(Name, Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `waveSynth = wavetableSynthesizer('Amplitude',2,'DCOffset',2.5)` creates a System object, `waveSynth`, that generates sine waveforms using the default `Wavetable` (a single-cycle sinusoid), with `Amplitude` set to 2 and `DCOffset` set to 2.5.

Properties

Wavetable — Single-cycle waveform

`sin(2*pi*(0:511)/512)` (default) | vector of real values

Single-cycle waveform, specified as a vector of real values. The algorithm of the `wavetableSynthesizer` indexes into the single-cycle waveform to synthesize a periodic wave.

This property is semi-tunable. You can tune the values of the wavetable when the object is locked. However, you cannot tune the length of the wavetable when the object is locked.

Frequency — Frequency of generated signal (Hz)

100 (default) | real scalar

Frequency of generated signal in Hz, specified as a real scalar greater than or equal to 0.

This property is tunable. You can change the value of this property even when the object is locked.

Amplitude — Amplitude of generated signal

1 (default) | real scalar

Amplitude of generated signal, specified as a real scalar greater than or equal to 0.

The generated signal is multiplied by the value specified by `Amplitude` at the output, before DC offset is applied.

This property is tunable. You can change the value of this property even when the object is locked.

PhaseOffset — Normalized phase offset of generated signal

0 (default) | real scalar

Normalized phase offset of generated signal, specified as a real scalar with values in the range 0 to 1. The range is a normalized 2π radians interval.

This property is not tunable. You cannot change the value of this property when the object is locked.

DCOffset — Value added to each element of generated signal

0 (default) | real scalar

Value added to each element of the generated signal, specified as a real scalar.

This property is tunable. You can change the value of this property even when the object is locked.

SamplesPerFrame — Number of samples per frame

512 (default) | positive integer

Number of samples per frame, specified as a positive integer in the range 1 to 192,000.

This property determines the vector length that the `step` method of your `wavetableSynthesizer` object outputs.

This property is tunable. You can change the value of this property even when the object is locked.

SampleRate — Sample rate of generated signal (Hz)

44100 (default) | real positive scalar

Sample rate of generated signal in Hz, specified as a real positive scalar.

This property is tunable. You can change the value of this property even when the object is locked.

OutputDataType — Data type of generated signal

'double' (default) | 'single'

Data type of generated signal, specified as 'double' or 'single'.

This property is not tunable. You cannot change the value of this property when the object is locked.

Methods

`clone`

Create copy of System object with same property values

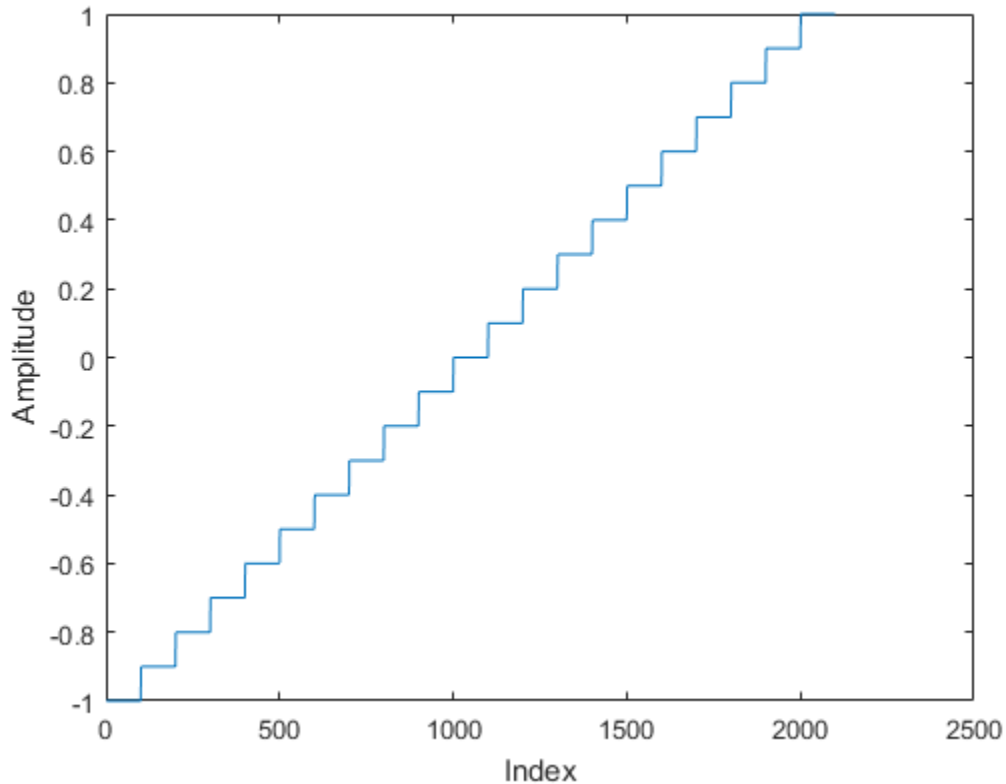
configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
isLocked	Locked status for input attributes and nontunable properties
release	Enable property values and input characteristics to change
reset	Reset internal states of System object
step	Generate periodic signals from single-cycle waveforms

Examples

Generate Variable-Frequency Staircase Wave

Define and plot a single-cycle waveform.

```
values = -1:.1:1;  
singleCycleWave = ones(100,1) * values;  
singleCycleWave = reshape(singleCycleWave,numel(singleCycleWave),1);  
  
plot(singleCycleWave);  
xlabel('Index');  
ylabel('Amplitude');
```



Create a wavetable synthesizer, `waveSynth`, to generate a staircase wave using the single-cycle waveform. Specify a frequency of 10 Hz.

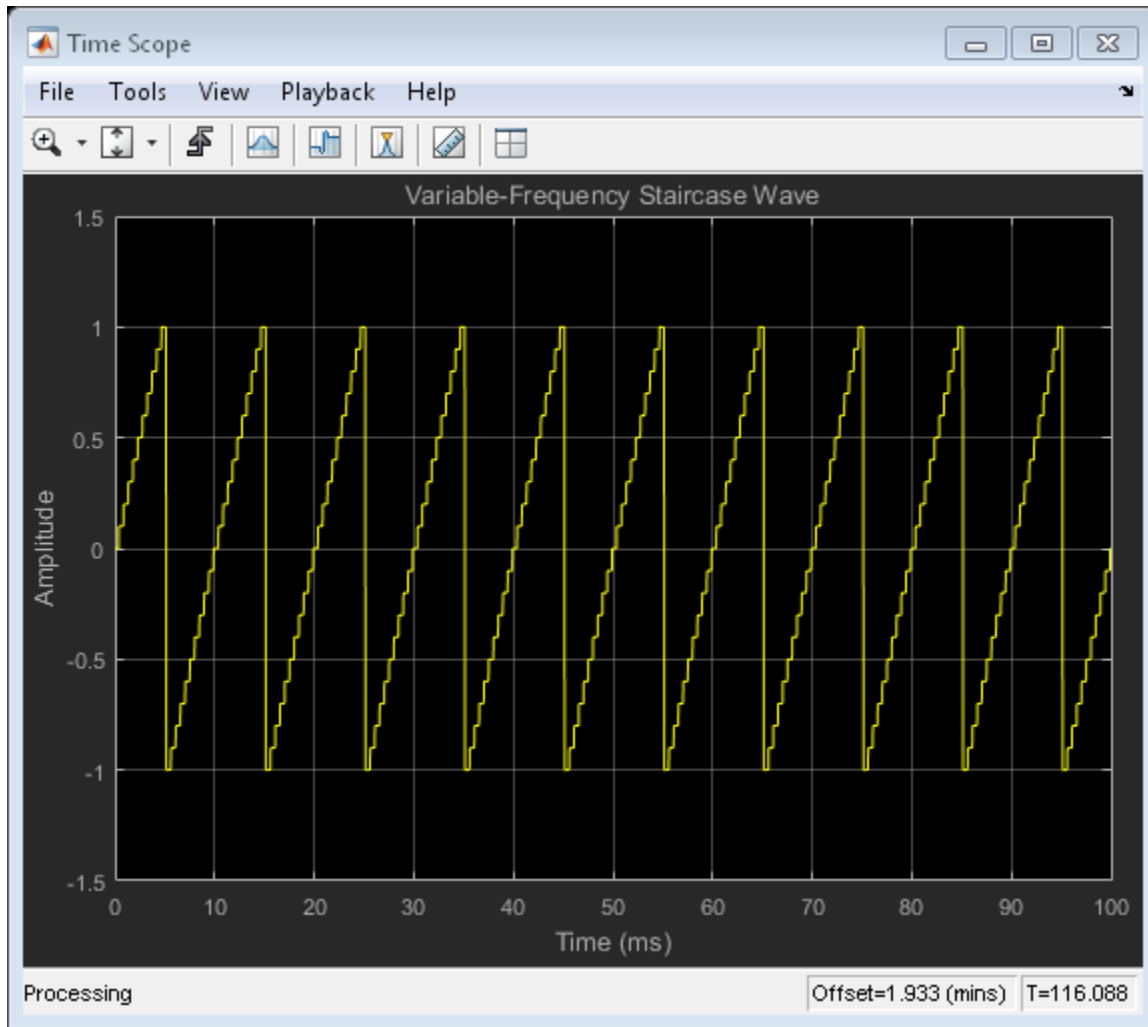
```
waveSynth = wavetableSynthesizer(singleCycleWave,10);
```

Create a time scope to visualize the staircase wave generated by `waveSynth`.

```
scope = dsp.TimeScope(...  
    'SampleRate',waveSynth.SampleRate,...  
    'TimeSpan',.1,...  
    'YLimits',[-1.5,1.5],...  
    'TimeSpanOverrunAction','Scroll',...  
    'ShowGrid',true,...  
    'Title','Variable-Frequency Staircase Wave');
```

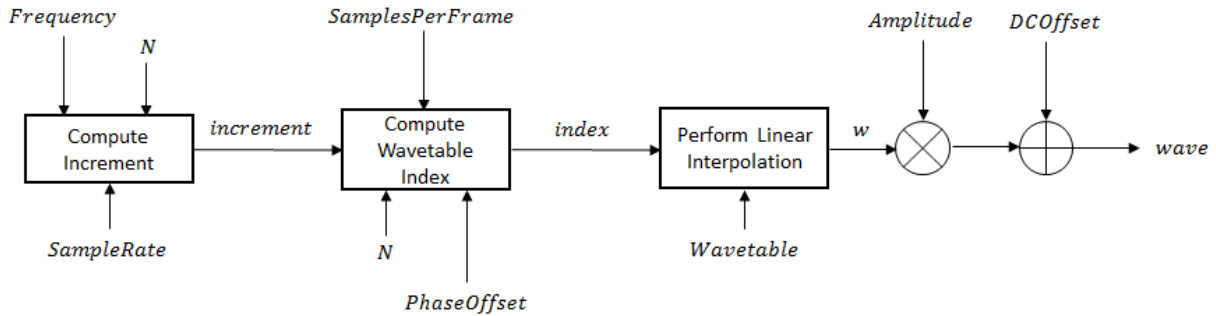
Place the wavetable synthesizer in an audio stream loop. Increase the frequency of your staircase wave in 10 Hz increments.

```
counter = 0;
while (counter < 1e4)
    counter = counter + 1;
    staircaseWave = step(waveSynth);
    step(scope, staircaseWave);
    if mod(counter, 1000) == 0
        waveSynth.Frequency = waveSynth.Frequency + 10;
    end
end
```



Algorithms

The wavetable synthesizer System object synthesizes periodic signals using a cached single-cycle waveform, specified waveform properties, and phase memory.



- 1 Compute the increment step size,

$$increment = \frac{Frequency}{SampleRate} \times N,$$

where N is the number of elements in your wavetable.

- 2 Compute Wavetable index,

$$index[n] = \begin{cases} index[n-1] + increment & \text{if } index[n-1] < N \\ index[n-1] + increment - N & \text{else} \end{cases},$$

for $2 \leq n \leq SamplesPerFrame$. The PhaseOffset property determines $index[n=1]$.

- 3 Index into the Wavetable and perform linear interpolation:

$$w = \begin{cases} (Wavetable[1] - Wavetable[index_{low}]) \times fraction + Wavetable[index_{low}] & \text{if } index_{high} \\ (Wavetable[index_{high}] - Wavetable[index_{low}]) \times fraction + Wavetable[index_{low}] & \text{else} \end{cases}$$

- $index_{low} = \text{floor}(index[n] + 1)$
- $index_{high} = index_{low} + 1$
- $fraction = index - \text{floor}(index)$

- 4 Multiply by Amplitude and add DCOffset.

$$wave = w \times Amplitude + DCOffset$$

See Also

audioOscillator

Introduced in R2016a

clone

System object: wavetableSynthesizer

Create copy of System object with same property values

Syntax

```
waveSynthClone = clone(waveSynth)
```

Description

`waveSynthClone = clone(waveSynth)` creates a `wavetableSynthesizer` System object, `waveSynthClone`, with the same property values as `waveSynth`. If the original object is locked, then `clone` creates a copy that is also locked. This copy has states initialized to the same values as the original. If the original object is not locked, then `clone` creates a new unlocked object with uninitialized states.

Introduced in R2016a

configureMIDI

System object: wavetableSynthesizer

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI (waveSynth)
configureMIDI (waveSynth,propName)
configureMIDI (waveSynth,propName,controlNumber)
configureMIDI (waveSynth,propName,controlNumber,'DeviceName',
deviceName)
```

Description

`configureMIDI (waveSynth)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the `wavetableSynthesizer` System object, `waveSynth`, to MIDI controls of your choice.

`configureMIDI (waveSynth,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

`configureMIDI (waveSynth,propName,controlNumber)` makes the property respond to the MIDI control specified by `controlNumber`.

`configureMIDI (waveSynth,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `wavetableSynthesizer` System object maps to MIDI controls with a specified range.

Property	Range	Mapping
Frequency	0.1 Hz to 20 kHz	log
Amplitude	0 to 10	linear
DCoffset	-10 to 10	linear

Introduced in R2016a

createAudioPluginClass

System object: wavetableSynthesizer

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(waveSynth)
createAudioPluginClass(waveSynth,pluginName)
```

Description

`createAudioPluginClass(waveSynth)` creates a System object source plugin that implements the functionality of the `wavetableSynthesizer` System object, `waveSynth`. The name of the created class is the `wavetableSynthesizer` System object variable name followed by 'Plugin', for example, `waveSynthPlugin`. By default, the created class outputs a one-channel (column) matrix.

`createAudioPluginClass(waveSynth,pluginName)` specifies the name of your created System object source plugin class.

Example: `createAudioPluginClass(waveSynth, 'myWavetableSynthesizer')` creates a System object source plugin with class name `myWavetableSynthesizer`.

Each tunable property of the `wavetableSynthesizer` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Mapping
Frequency	0.1 Hz to 20 kHz	log
Amplitude	0 to 10	linear
DCOffset	-10 to 10	linear

Introduced in R2016a

disconnectMIDI

System object: wavetableSynthesizer

Disconnect MIDI controls from System object

Syntax

`disconnectMIDI(waveSynth)`

Description

`disconnectMIDI(waveSynth)` disconnects MIDI controls from your wavetable synthesizer, `waveSynth`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: wavetableSynthesizer

Get MIDI connection information

Syntax

```
connectionInfo= getMIDIConnections(waveSynth)
```

Description

`connectionInfo= getMIDIConnections(waveSynth)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your wavetable synthesizer, `waveSynth`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `waveSynth` that has established MIDI connections. Each substructure contains the control number, the device name of the corresponding MIDI control, and the property mapping information (mapping rule, minimum value, and maximum value).

Introduced in R2016a

isLocked

System object: wavetableSynthesizer

Locked status for input attributes and nontunable properties

Syntax

`L = isLocked(waveSynth)`

Description

`L = isLocked(waveSynth)` returns a logical value, `L`, that indicates whether input attributes and nontunable properties are locked for the wavetable synthesizer, `waveSynth`.

The `waveSynth` object performs an internal initialization the first time you execute `step`. The initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After the object is locked, the `isLocked` method returns a `true` value.

Introduced in R2016a

release

System object: wavetableSynthesizer

Enable property values and input characteristics to change

Syntax

```
release(waveSynth)
```

Description

`release(waveSynth)` releases system resources of your wavetable synthesizer, `waveSynth`. Resources include memory, file handles, and hardware connections. After you call `release`, all properties and input characteristics of `waveSynth` can change.

Note: Once you call `release` on a System object, subsequent calls to `setup`, `step`, `reset`, or `release` do not support code generation.

Introduced in R2016a

reset

System object: wavetableSynthesizer

Reset internal states of System object

Syntax

reset(waveSynth)

Description

reset(waveSynth) resets internal states of the wavetable synthesizer, waveSynth, to their initial values.

Introduced in R2016a

step

System object: wavetableSynthesizer

Generate periodic signals from single-cycle waveforms

Syntax

```
y = step(waveSynth)
```

Description

`y = step(waveSynth)` generates a periodic signal, `y`. The type of signal is specified by the algorithm and properties of the `wavetableSynthesizer` System object, `waveSynth`.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

Classes in Audio System Toolbox

audioPlugin class

Base class for audio plugins

Description

`audioPlugin` is the base class for audio plugins. In your class definition file, you must subclass your object from this base class or from the `audioPluginSource` class, which inherits from `audioPlugin`. Subclassing enables you to inherit the attributes necessary to generate plugins and access Audio System Toolbox functionality.

To inherit from the `audioPlugin` base class directly, type this syntax as the first line of your class definition file:

```
classdef myAudioPlugin < audioPlugin  
myAudioPlugin is the name of your object.
```

For a tutorial on designing audio plugins, see “Design an Audio Plugin”.

Methods

`getSampleRate`

Get sample rate at which the plugin is run

`setSampleRate`

Set sample rate at which the plugin is run
(MATLAB environment only)

Copy Semantics

Handle. To learn how handle classes affect copy operations, see “Copying Objects” in the MATLAB documentation.

Examples

Design Valid Audio Plugin

Design a valid basic audio plugin class

Terminology:

- A valid audio plugin is one that can be deployed in a digital audio workstation (DAW) environment. To validate it, use the `validateAudioPlugin` function. To generate it, use the `generateAudioPlugin` function.
- A basic audio plugin inherits from the `audioPlugin` class but not the `matlab.System` class.

Define a basic audio plugin class that inherits from `audioPlugin`.

```
classdef myAudioPlugin < audioPlugin
end
```

Add a processing function to your plugin class.

All valid audio plugins include a processing function. For basic audio plugins, the processing function is named `process`. The processing function is where audio processing occurs. It always has an output.

```
classdef myAudioPlugin < audioPlugin
    methods
        function out = process(~,in)
            out = in;
        end
    end
end
```

Design Valid Audio Plugin That Uses `getSampleRate`

Design an `audioPlugin` class that uses the `getSampleRate` method to get the sample rate at which the plugin is run. The plugin in this example, `simpleStrobe`, uses the sample rate to determine a constant 50 ms strobe period.

```
classdef simpleStrobe < audioPlugin
    % simpleStrobe Add audio strobe effect
    % Add a strobe effect by gain switching between 0 and 1 in
    % 50 ms increments. Although the input sample rate can change,
    % the strobe period remains constant.
    %
    % simpleStrobe properties:
    % period - Number of samples between gain switches
    % gain - Gain multiplier, one or zero
    % count - Number of samples since last gain switch
    %
```

```
%
% simpleStrobe methods:
% process - Multiply input frame by gain, element by element
% reset - Reset count and gain to initial conditions
%           and get sample rate

properties
    Period = 44100*0.05;
    Gain = 1;
end
properties (Access = private)
    Count = 1;
end
methods
    function out = process(plugin,in)
        for i = 1:size(in,1)
            if plugin.Count == plugin.Period
                plugin.Gain = 1 - plugin.Gain;
                plugin.Count = 1;
            end
            in(i,:) = in(i,)*plugin.Gain;
            plugin.Count = plugin.Count + 1;
        end
        out = in;
    end
    function reset(plugin)
        plugin.Period = floor( getSampleRate(plugin)*0.05 );
        plugin.Count = 1;
        plugin.Gain = 1;
    end
end
end
```

See Also

[audioPluginSource](#) | [audioPluginInterface](#) | [audioPluginParameter](#) | [generateAudioPlugin](#) | [validateAudioPlugin](#)

More About

- “Hierarchies of Classes — Concepts”

Introduced in R2016a

getSampleRate

Class: audioPlugin

Get sample rate at which the plugin is run

Syntax

```
sampleRate = getSampleRate(myAudioPlugin)
```

Description

`sampleRate = getSampleRate(myAudioPlugin)` returns the sample rate in Hz at which the plugin is being run.

- In a digital audio workstation (DAW) environment, the DAW user sets the sample rate. `getSampleRate` interacts with the DAW to determine the sample rate.
- In the MATLAB environment, `getSampleRate` returns the value set by a previous call to `setSampleRate`. If `setSampleRate` has not been called, `getSampleRate` returns the default value, 44100.

Introduced in R2016a

setSampleRate

Class: audioPlugin

Set sample rate at which the plugin is run (MATLAB environment only)

Syntax

```
setSampleRate(myAudioPlugin,sampleRate)
```

Description

`setSampleRate(myAudioPlugin,sampleRate)` sets the sample rate of the plugin, `myAudioPlugin`, to the value specified by `sampleRate`. Specify `sampleRate` as a positive real integer. `setSampleRate` enables the MATLAB environment to mimic behavior in a digital audio workstation (DAW) environment.

Note: Do not use `setSampleRate` in a generated plugin. If you call `setSampleRate` in your plugin, `generateAudioPlugin` throws an error.

Introduced in R2016a

audioPluginSource class

Base class for audio source plugins

Description

`audioPluginSource` is the base class for audio source plugins. Use audio source plugins to produce audio signals.

To create a valid audio source plugin, in your class definition file, subclass your object from the `audioPluginSource` class. Subclassing enables you to inherit the attributes necessary to generate audio source plugins and access Audio System Toolbox functionality. To inherit from the `audioPluginSource` base class directly, type this syntax as the first line of your class definition file:

```
classdef myAudioSourcePlugin < audioPluginSource  
myAudioSourcePlugin is the name of your object.
```

Methods

<code>getSamplesPerFrame</code>	Get frame size at which the plugin is run
<code>setSamplesPerFrame</code>	Set frame size at which the plugin is run (MATLAB environment only)

Inherited Methods

<code>getSampleRate</code>	Get sample rate at which the plugin is run
<code>setSampleRate</code>	Set sample rate at which the plugin is run (MATLAB environment only)

Copy Semantics

Handle. To learn how handle classes affect copy operations, see “Copying Objects” in the MATLAB documentation.

Examples

Design Valid Audio Plugin

Design a valid basic audio source plugin class

Terminology:

- A valid audio source plugin is one that can be deployed in a digital audio workstation (DAW) environment. To validate it, use the `validateAudioPlugin` function. To generate it, use the `generateAudioPlugin` function.
- A basic audio source plugin inherits from the `audioPluginSource` class but not the `matlab.System` class.

Define a basic audio source plugin class that inherits from `audioPluginSource`.

```
classdef myAudioSourcePlugin < audioPluginSource
end
```

Add a processing function to your audio source plugin class.

All valid audio source plugins include a processing function. For basic audio source plugins, the processing function is named `process`. The processing function defines the audio signal that your plugin outputs. Audio source plugins do not accept audio signals as input to the processing function.

The default audio plugin interface assumes a stereo output. Specify the processing output as a matrix with two columns. These columns correspond to the left and right channels of a stereo signal. The number of rows in the output matrix correspond to the frame size.

The output frame size must match the frame size of the environment in which the plugin is run. A DAW environment has variable frame size. To determine the current environment frame size, call `getSamplesPerFrame` in the `process` function.

```
classdef myAudioSourcePlugin < audioPluginSource
    methods
        function out = process(plugin)
            out = 0.5*randn(getSamplesPerFrame(plugin),2);
        end
    end
end
```

`myAudioSourcePlugin` generates a Gaussian white noise audio signal with 0.5 standard deviation.

See Also

`audioPlugin` | `audioPluginInterface` | `audioPluginParameter` |
`generateAudioPlugin` | `validateAudioPlugin`

More About

- “Hierarchies of Classes — Concepts”

Introduced in R2016a

getSamplesPerFrame

Class: audioPluginSource

Get frame size at which the plugin is run

Syntax

```
frameSize = getSamplesPerFrame(myAudioSourcePlugin)
```

Description

`frameSize = getSamplesPerFrame(myAudioSourcePlugin)` returns the frame size at which the plugin is run. It is valid only when called in the processing function of an audio source plugin. `frameSize` is the number of output samples (rows) that the current call to the processing function of `myAudioSourcePlugin` must return.

- In a digital audio workstation (DAW) environment, `getSamplesPerFrame` interacts with the DAW to determine the frame size. Frame size can vary from call to call, as determined by the DAW environment.
- In the MATLAB environment, `getSamplesPerFrame` returns the value set by a previous call to the `setSamplesPerFrame` method. If `setSamplesPerFrame` has not been called, then `getSamplesPerFrame` returns the default value, 256.

Introduced in R2016a

setSamplesPerFrame

Class: audioPluginSource

Set frame size at which the plugin is run (MATLAB environment only)

Syntax

```
setSamplesPerFrame(myAudioSourcePlugin, frameSize)
```

Description

`setSamplesPerFrame(myAudioSourcePlugin, frameSize)` sets the frame size (rows) that the source plugin, `myAudioSourcePlugin`, must return in subsequent calls to its processing function. Specify `frameSize` as a real integer greater than or equal to 0. `setSamplesPerFrame` enables the MATLAB environment to mimic behavior in a digital audio workstation (DAW) environment.

Note: Do not use `setSamplesPerFrame` in a generated plugin. If you call `setSamplesPerFrame` in your plugin, `generateAudioPlugin` throws an error.

Introduced in R2016a

Blocks in Audio System Toolbox

Audio Device Reader

Record from sound card

Library

Sources

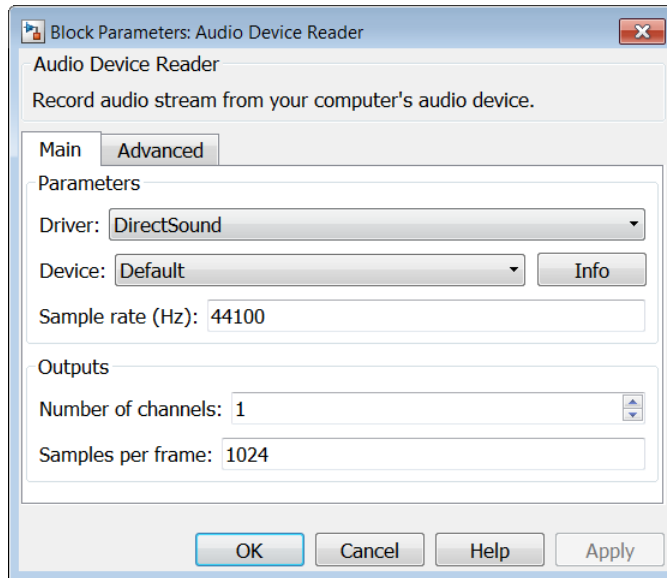
audiosources

Description



The **Audio Device Reader** block reads audio samples using your computer's audio device. See “System Interaction of Audio Device Reader Block” on page 5-6 for a visualization of how the **Audio Device Reader** block acquires data.

Dialog Box



Main Tab

Driver

Driver used to access your audio device, specified as `DirectSound` or `ASIO`. The default is `DirectSound`. ASIO drivers do not come pre-installed on Windows machines. You must install an ASIO driver outside of MATLAB to use the ASIO driver option.

Note: If `Driver` is specified as `ASIO`, open the ASIO UI outside of MATLAB to set the sound card buffer size to the value specified by the **Samples per frame** parameter. See the documentation of your ASIO driver for more information.

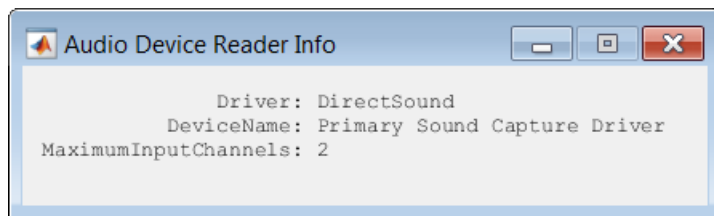
This parameter applies only on Windows machines. Linux machines always use the ALSA driver. Mac machines always use the CoreAudio driver.

Device

Device used to acquire audio samples. The default for this parameter is the default audio input device of your computer. The device list is populated with devices available on your computer.

Info

Opens a dialog box with information about your configured driver and device, and the maximum number of channels you can acquire given your configuration.



Sample rate (Hz)

Sample rate used by your device to acquire audio data, in Hz, specified as a positive integer. The possible range of **Sample rate (Hz)** depends on your audio hardware.

Number of channels

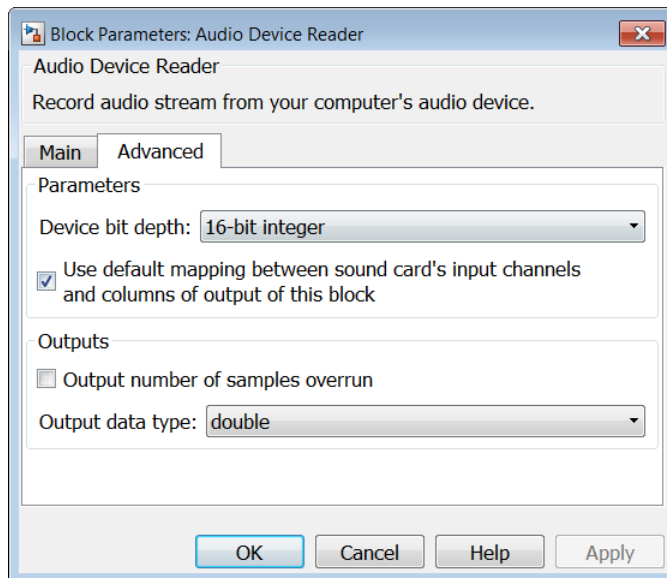
Number of input channels acquired by your audio device, specified as an integer. The number of input channels is also the number of channels (matrix columns) output by the `Audio Device Reader` block. The range of **Number of channels** depends on your audio hardware.

To specify the number of input channels acquired by your audio device, select the **Use default mapping between sound card's input channels and columns of output of this block** check box.

Samples per frame

Frame size read from audio device, specified as a positive integer. **Samples per frame** is also the device buffer size, and the frame size (number of matrix rows) output by the `Audio Device Reader` block.

Advanced Tab



Device bit depth

Data type used by device to acquire audio data. You can set this parameter to **8-bit integer**, **16-bit integer**, **24-bit integer**, or **32-bit integer**. The default is **16-bit integer**.

Use default mapping between sound card's input channels and columns of output of this block

When you select this check box, the block uses the default mapping between the sound card's input channels and the matrix columns output by this block. When you clear this check box, you specify the mapping in **Device input channels**. By default, this check box is selected.

Device input channels

Nondefault map of device channels and matrix output by the **Audio Device Reader** block, specified as a MATLAB vector. For example:

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

- Channel 1 maps to the first column of the output matrix.

- Channel 2 maps to the second column of the output matrix.
- Channel 3 maps to the third column of the output matrix.

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

- Channel 3 maps to the first column of the output matrix.
- Channel 1 maps to the second column of the output matrix.
- Channel 2 maps to the third column of the output matrix.

To specify a nondefault mapping, clear the **Use default mapping between sound card's input channels and columns of output of this block** check box.

Output number of samples overrun

When you select this check box, an additional output port, O, is added to the block. The O port outputs the number of samples overrun while acquiring a frame of data (one output matrix). By default, this check box is cleared.

Output data type

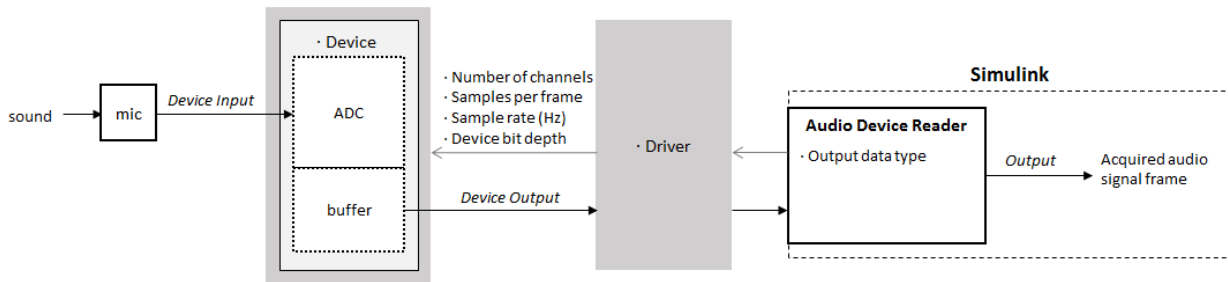
You can set this parameter to `uint8`, `int16`, `int32`, `single`, or `double`. The default is `double`.

Note: If **Output data type** is set to `double` or `single`, the Audio Device Reader block outputs data in the range [-1, 1]. For other data types, the range is [min, max] of the specified data type.

More About

System Interaction of Audio Device Reader Block

The Audio Device Reader block specifies the driver, the device and its attributes, and the data type and size output from your Audio Device Reader block.



Run an Executable Outside MATLAB

The generated code for the `Audio Device Reader` block relies on prebuilt dynamic library files that ship with MATLAB. You must account for these extra library files when you run `Audio Device Reader` outside the MATLAB environment. To run a standalone executable generated from a model containing the `Audio Device Reader` block, set your system environment using commands specific to your platform.

Platform	Command
Mac	<pre>setenv DYLD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/maci64 (csh/tcsh) export DYLD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/maci64 (Bash)</pre>
Linux	<pre>setenv LD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/glnxa64 (csh/tcsh) export LD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/glnxa64 (Bash)</pre>
Windows	<pre>set PATH = \$MATLABROOT\bin\win64; %PATH%</pre>

Alternatively, the `packNGO` function creates a single zip file containing all pieces required to run or rebuild this code. See `packNGO` for more information.

See Also

`audioDeviceReader` | `audioDeviceWriter` | `Audio Device Writer`

Introduced in R2016a

Audio Device Writer

Play to sound card

Library

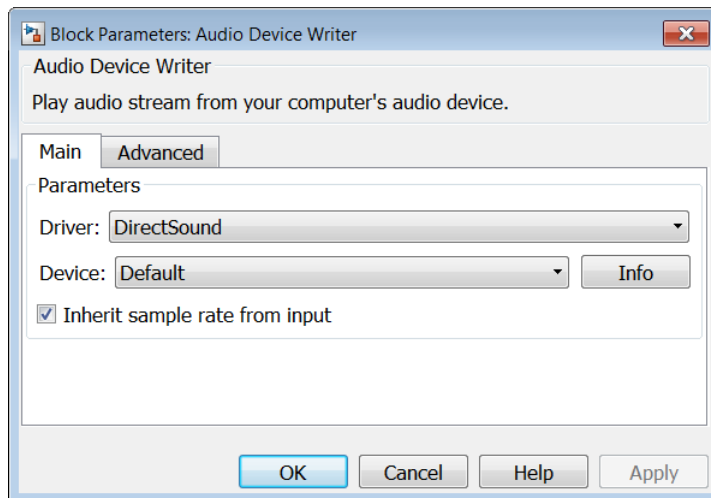
Sinks

audiosinks



The Audio Device Writer block writes audio samples to an audio output device. See “System Interaction of Audio Device Writer Block” on page 5-13 for a visualization of how the Audio Device Writer block plays audio data.

Dialog Box



Main Tab

Driver

Driver used to access your audio device, specified as `DirectSound` or `ASIO`. The default is `DirectSound`. ASIO drivers do not come pre-installed on Windows machines. You must install an ASIO driver outside of MATLAB to use the ASIO driver option.

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

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

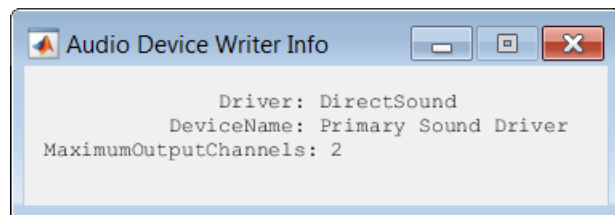
To specify nondefault **Driver** values, you must install Audio System Toolbox. If the toolbox is not installed, specifying nondefault **Driver** values returns an error.

Device

Device used to play audio samples. The default for this parameter is the default audio output device of your computer. The device list is populated with devices available on your computer.

Info

Opens a dialog box with information about your configured driver and device, and the maximum number of channels you can output from your device given your configuration.



Inherit sample rate from input

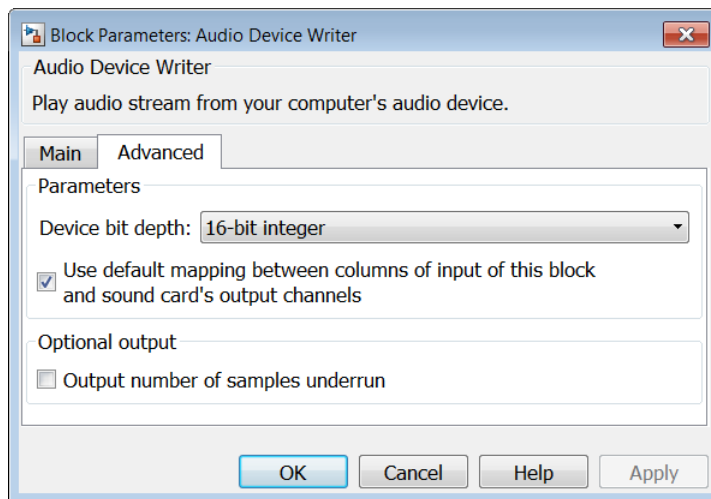
When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**. By default, this check box is selected.

Sample rate (Hz)

Sample rate used by device to play audio data, in Hz, specified as a positive integer. The default is 44100 Hz. The possible range of **Sample rate (Hz)** depends on your audio hardware.

You can specify an input sample rate when the **Inherit sample rate from input** check box is cleared.

Advanced Tab



Device bit depth

Data type used by device to perform digital-to-analog conversion. Before performing digital-to-analog conversion, the input data is cast to a data type specified by the **Device bit depth** parameter. You can set this parameter to **8-bit integer**, **16-bit integer**, **24-bit integer**, or **32-bit integer**. The default is **16-bit integer**.

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

Use default mapping between columns of input of this block and sound card's output channels

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

Device output channels

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

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

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

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

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

To specify a nondefault mapping, clear the **Use default mapping between columns of input of this block and sound card's output channels** check box.

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

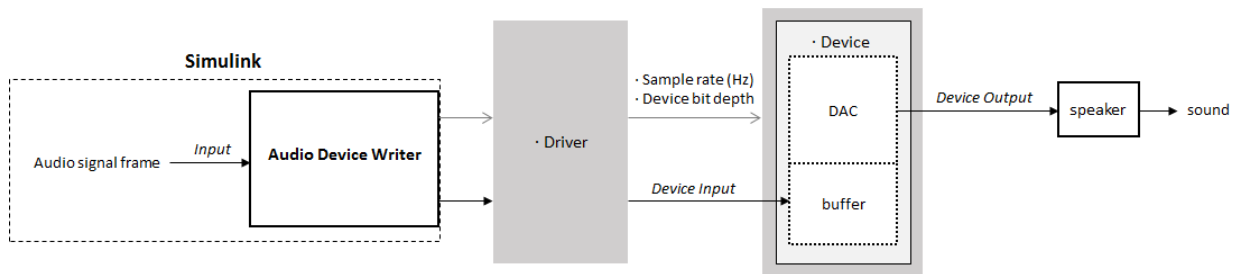
Output number of samples underrun

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

More About

System Interaction of Audio Device Writer Block

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



Run an Executable Outside MATLAB

The generated code for the Audio Device Writer block relies on prebuilt dynamic library files that ship with MATLAB. You must account for these extra library files when you run Audio Device Writer outside the MATLAB environment. To run a standalone executable generated from a model containing the Audio Device Writer block, set your system environment using commands specific to your platform.

Platform	Command
Mac	<pre> setenv DYLD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/maci64 (csh/tcsh) export DYLD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/maci64 (Bash) </pre>
Linux	<pre> setenv LD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/glnxa64 (csh/tcsh) </pre>

Platform	Command
	<pre>export LD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/ bin/glnxa64 (Bash)</pre>
Windows	<pre>set PATH = \$MATLABROOT\bin\win64; %PATH%</pre>

Alternatively, the `packNGO` function creates a single zip file containing all pieces required to run or rebuild this code. See `packNGO` for more information.

Supported Data Types

Port	Supported Data Types
Input	<ul style="list-style-type: none">• Double-precision floating point• Single-precision floating point• 32-bit signed integers• 16-bit signed integers• 8-bit unsigned integers
Underrun	32-bit unsigned integer

Note: If input to the `Audio Device Writer` block is of data type `double` or `single`, the `Audio Device Writer` block clips values outside the range $[-1, 1]$. For other data types, the allowed input range is $[\text{min}, \text{max}]$ of the specified data type.

See Also

`audioDeviceWriter` | `audioDeviceReader` | `Audio Device Reader`

Introduced in R2016a

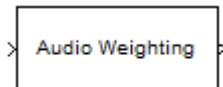
Audio Weighting Filter

Design audio weighting filter

Library

Filters

audiofilters

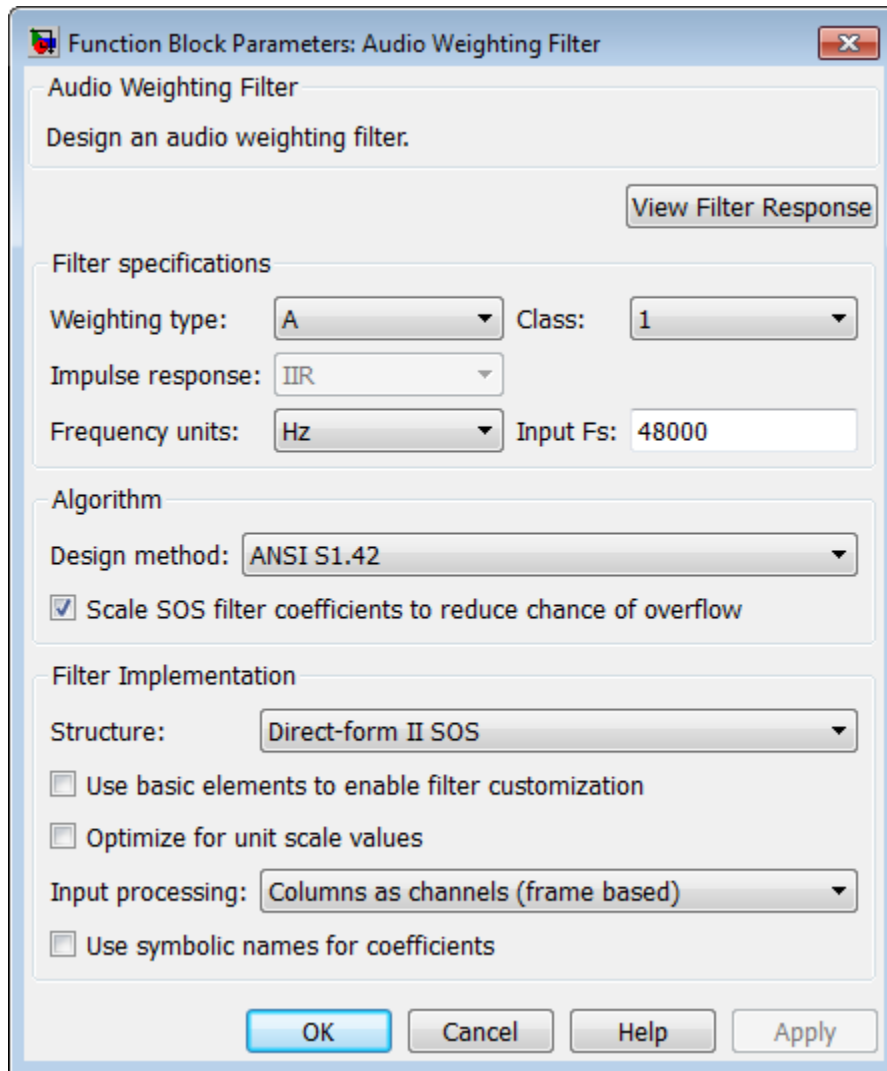


Description Audio Weighting Filter

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

Dialog Box

See “Audio Weighting Filter Design Dialog Box — Main Pane” for more information about the parameters of this block.



View Filter Response

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

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

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

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

Filter Specifications

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

Weighting type

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

Class

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

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

Impulse response

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

Frequency units

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

Input Fs

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

Algorithm

Design Method

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

Scale SOS filter coefficients to reduce chance of overflow

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

Filter Implementation

Structure

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

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

Use basic elements to enable filter customization

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

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

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

Optimize for unit-scale values

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

Input processing

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

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

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

Use symbolic names for coefficients

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

References

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

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

Supported Data Types

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

See Also

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

How To

- [Audio Weighting Filters Example](#)

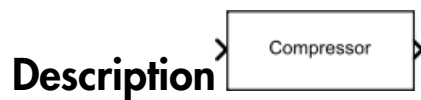
Compressor

Dynamic range compressor

Library

Dynamic Range Control

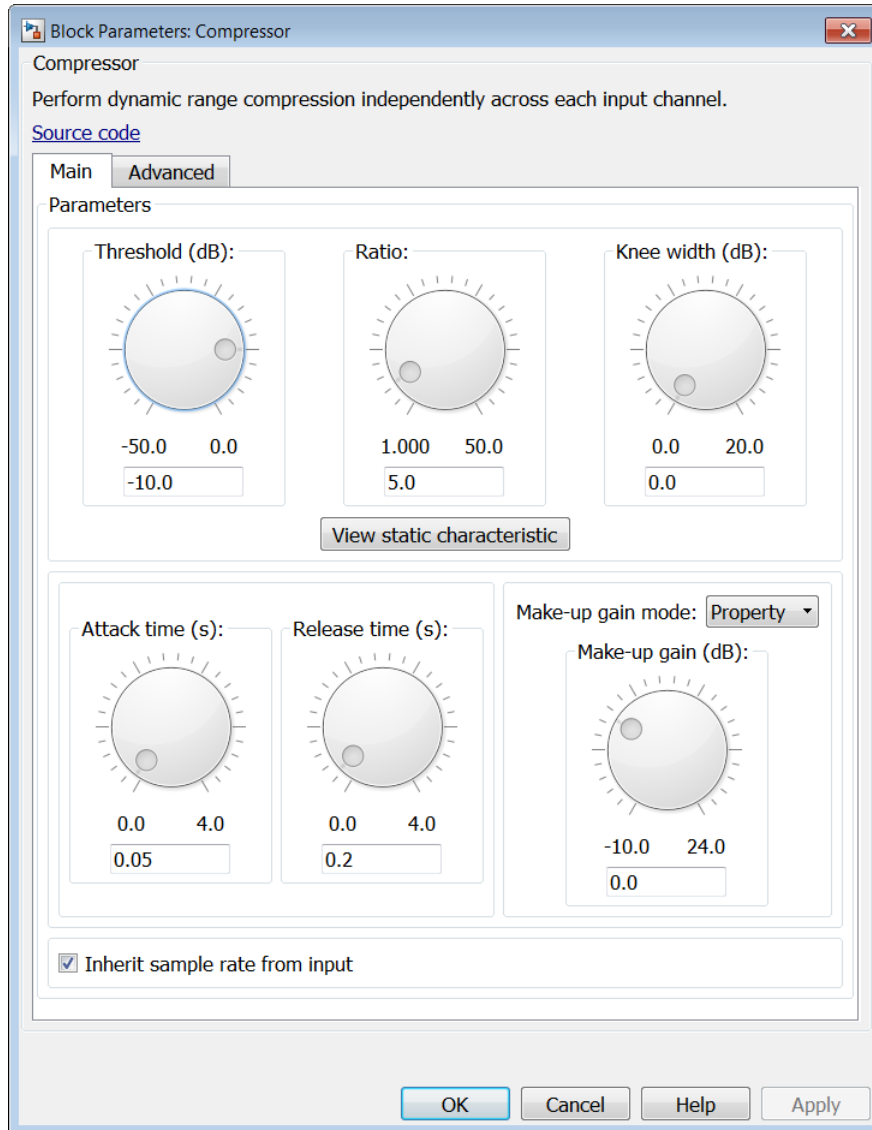
audiodynamicrange



The **Compressor** block performs dynamic range compression independently across each input channel. Dynamic range compression attenuates the volume of loud sounds that cross a given threshold. It uses specified attack and release times to achieve a smooth applied gain curve. You can tune parameters of the **Compressor** block to meet your processing needs.

The input must be a real-valued, double-precision or single-precision matrix. The **Compressor** block treats each column of the input as an independent channel.

Dialog Box



Main Tab

Threshold (dB)

Operation threshold in dB, specified as a real scalar in the range -50 to 0 . The default is -10 dB.

Operation threshold is the level above which gain is applied to the input signal.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Ratio

Compression ratio, specified as a real scalar in the range 1 to 50 . The default is 5 .

Compression ratio is the input/output ratio for signals that overshoot the operation threshold.

Assuming a hard knee characteristic and a steady-state input such that $x[n]$ dB $>$

Threshold (dB), the compression ratio is defined as
$$R = \frac{(x[n] - T)}{(y[n] - T)}$$
.

- R is the compression ratio.
- $x[n]$ is the input signal in dB.
- $y[n]$ is the output signal in dB.
- T is the threshold in dB.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Knee width (dB)

Knee width in dB, specified as a real scalar in the range 0 to 20 dB. The default is 0 dB.

Knee width is the transition area in the compression characteristic.

For soft knee characteristics, the transition area is defined by the relation

$$y = x + \frac{\left(\frac{1}{R} - 1\right) \times \left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

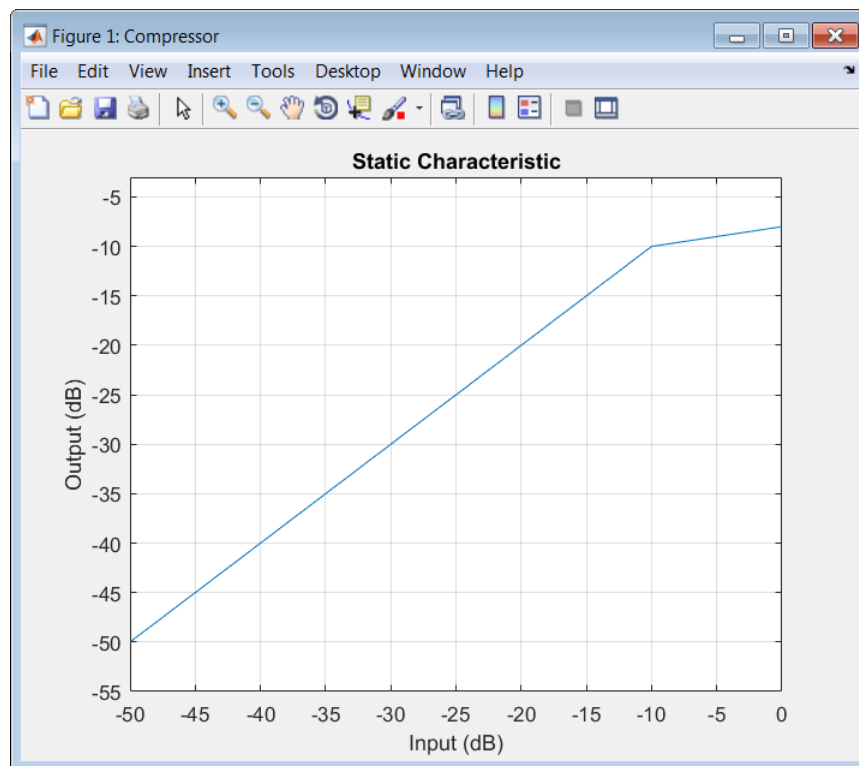
for the range $(2 \times |x - T|) \leq W$.

- y is the output level in dB.
- x is the input level in dB.
- R is the compression ratio.
- T is the threshold in dB.
- W is the knee width in dB.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

View static characteristic

Plots the static compression characteristic of the **Compressor** block, as defined by the parameters in the block dialog box. The plot updates when you tune parameters.



Attack time (s)

Attack time in seconds, specified as a real scalar in the range 0 to 4. The default is 0.05 seconds.

Attack time is the time it takes the compressor gain to rise from 10% to 90% of its final value when the input goes above the threshold.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Release time (s)

Release time in seconds, specified as a real scalar in the range 0 to 4. The default is 0.2 seconds.

Release time is the time it takes the compressor gain to drop from 90% to 10% of its final value when the input goes below the threshold.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Make-up gain mode

Make-up gain mode, specified as **Auto** or **Property**. The default is **Property**.

- **Auto** — Make-up gain is applied at the output of the **Compressor** block such that a steady-state 0 dB input has a 0 dB output.
- **Property** — Make-up gain is set to the value specified by **Make-up gain (dB)**.

Make-up gain (dB)

Make-up gain in dB, specified as a real scalar in the range -10 to 24. The default is 0 dB.

Make-up gain compensates for gain lost during compression. It is applied at the output of the **Compressor** block. This parameter is available when you set **Make-up gain mode** to **Property**.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

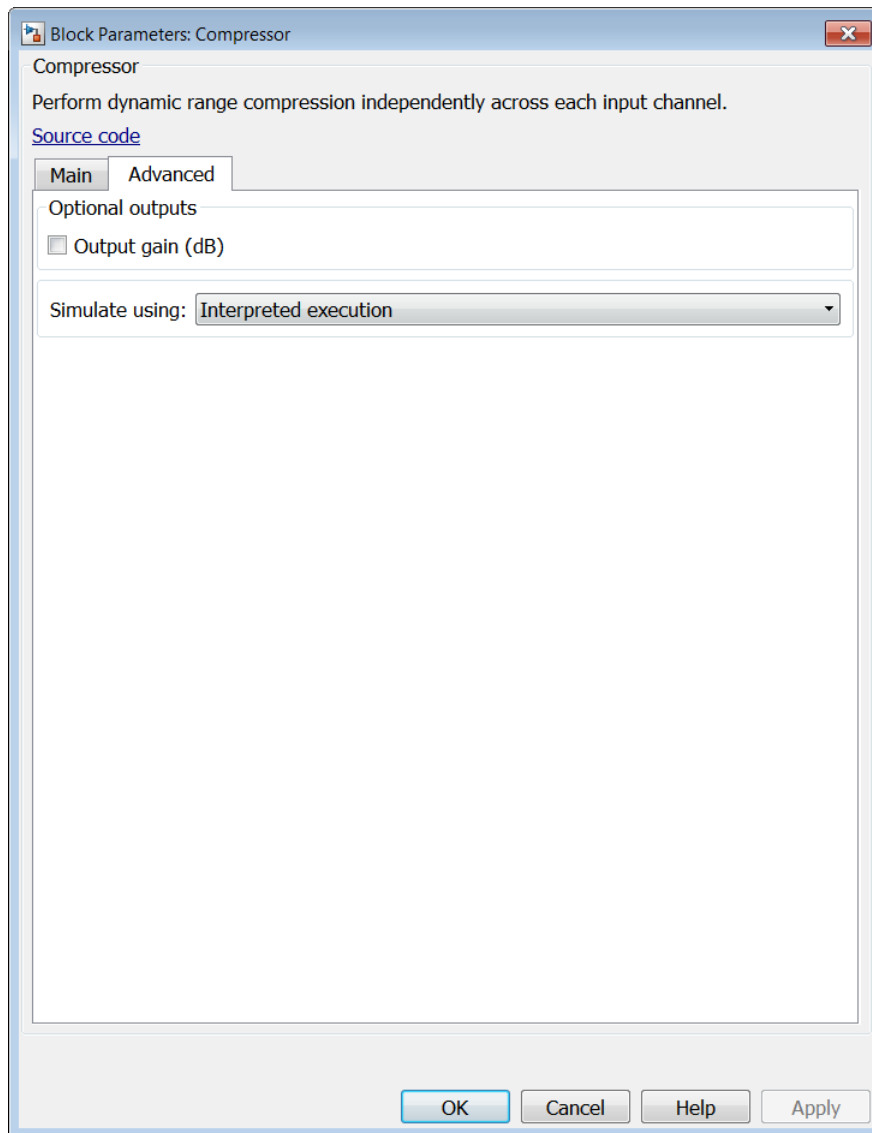
Inherit sample rate from input

When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**. By default, this check box is selected.

Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default is 44100 Hz. You can specify an input sample rate when the **Inherit sample rate from input** check box is cleared.

Advanced Tab



Output gain (dB)

When you select this check box, an additional output port, G, is added to the block. The G port outputs the gain applied on each input channel in dB. By default, this check box is cleared.

Simulate using

Type of simulation to run. You can set this parameter to:

- Interpreted execution (default)

Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to **Code generation**. In this mode, you can debug the source code of the block.

- Code generation

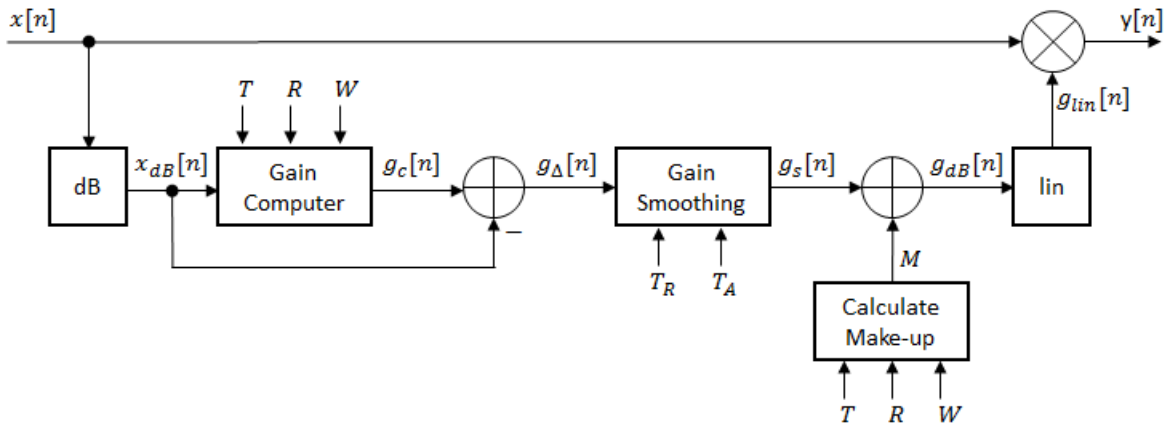
Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is comparable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

Supported Data Types

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

Algorithm

The **Compressor** block processes a signal frame by frame and element by element.



- 1 The N -point signal, $x[n]$, is converted to decibels:

$$x_{dB}[n] = 20 \times \log_{10} |x[n]|.$$

- 2 $x_{dB}[n]$ passes through the gain computer. The gain computer uses the static compression characteristic of the **COMPRESSOR** block to attenuate gain that is above the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:

$$g_c(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{\left(\frac{1}{R} - 1\right) \left(x_{dB} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \leq x_{dB} \leq \left(T + \frac{W}{2}\right) \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases},$$

where T is the threshold, R is the ratio, and W is the knee width.

If you specified a hard knee, the gain computer has the following static characteristic:

$$g_c(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < T \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} \geq T \end{cases} .$$

- 3 The gain modification, $g_\Delta[n]$, is calculated as

$$g_\Delta[n] = g_c[n] - x_{dB}[n].$$

- 4 $g_\Delta[n]$ is smoothed using specified attack and release time parameters,

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) |g_\Delta[n]|, & |g_\Delta[n]| > g_s[n-1] \\ \alpha_R g_s[n-1] + (1 - \alpha_R) |g_\Delta[n]|, & |g_\Delta[n]| \leq g_s[n-1] \end{cases} ,$$

where α_A , the attack time coefficient, is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{Fs \times T_A}\right),$$

and α_R , the release time coefficient, is calculated as

$$\alpha_R = \exp\left(\frac{-\log(9)}{Fs \times T_R}\right).$$

T_A is the attack time period, specified by the **Attack time (s)** parameter. T_R is the release time period, specified by the **Release time (s)** parameter. Fs is the input sampling rate, specified by the **Inherit sample rate from input** or the **Input sample rate (Hz)** parameter.

- 5 If **Make-up gain (dB)** is set to **Auto**, the make-up gain is calculated as the negative of the computed gain for a 0 dB input,

$$M = -g_c(x_{dB} = 0).$$

Given a steady-state input of 0 dB, this configuration achieves a steady-state output of 0 dB. The make-up gain is determined by the **Threshold (dB)**, **Ratio**, and **Knee width (dB)** parameters. It does not depend on the input signal.

- 6 The make-up gain, M , is added to the smoothed gain, $g_s[n]$:

$$g_{dB}[n] = g_s[n] + M.$$

- 7 The calculated gain in dB, $g_{dB}[n]$, is translated to a linear domain:

$$g_{lin}[n] = 10^{\left(\frac{g_{dB}[n]}{20}\right)}.$$

- 8 The output of the Compressor block is given as

$$y[n] = x[n] \times g_{lin}[n].$$

References

- [1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. “Digital Dynamic Range Compressor Design—A Tutorial And Analysis”. *Journal of Audio Engineering Society*. Vol. 60, Issue 6, pp. 399–408.

See Also

compressor | Expander | Limiter | Noise Gate

Introduced in R2016a

Crossover Filter

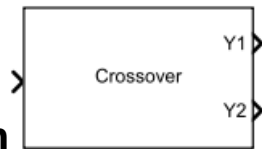
Audio crossover filter

Library

Filters

audiofilters

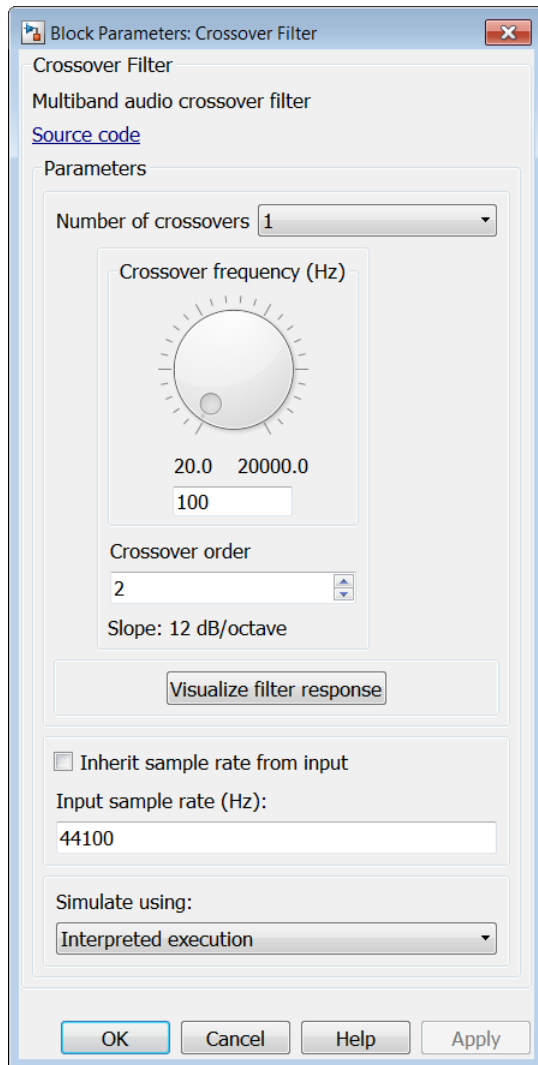
Description



The `Crossover Filter` block implements an audio crossover filter, which is used to split an audio signal into two or more frequency bands. Crossover filters are multiband filters whose overall magnitude frequency response is flat.

The input must be a real-valued, double-precision or single-precision matrix. The `Crossover Filter` block treats each column of the input as an independent channel.

Dialog Box



Number of crossovers

Number of magnitude response band crossings, specified as 1, 2, 3, or 4. The default is 1. If multiple crossovers are specified, your dialog box populates with the

corresponding parameters for **Crossover frequency (Hz)** and **Crossover order** automatically.

The number of bands output by the **Crossover Filter** block is one more than the **Number of crossovers**.

Number of crossovers	Number of bands output
1	two-band
2	three-band
3	four-band
4	five-band

This parameter is not tunable. You cannot change the value of this parameter when the simulation is running.

Crossover frequency (Hz)

Crossover frequencies in Hz, specified as a real scalar in the range 20 to 20,000 Hz. The default is 100 Hz.

Crossover frequencies are the intersections of magnitude response bands of the individual two-band crossover filters used in the multiband crossover filter.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Crossover order

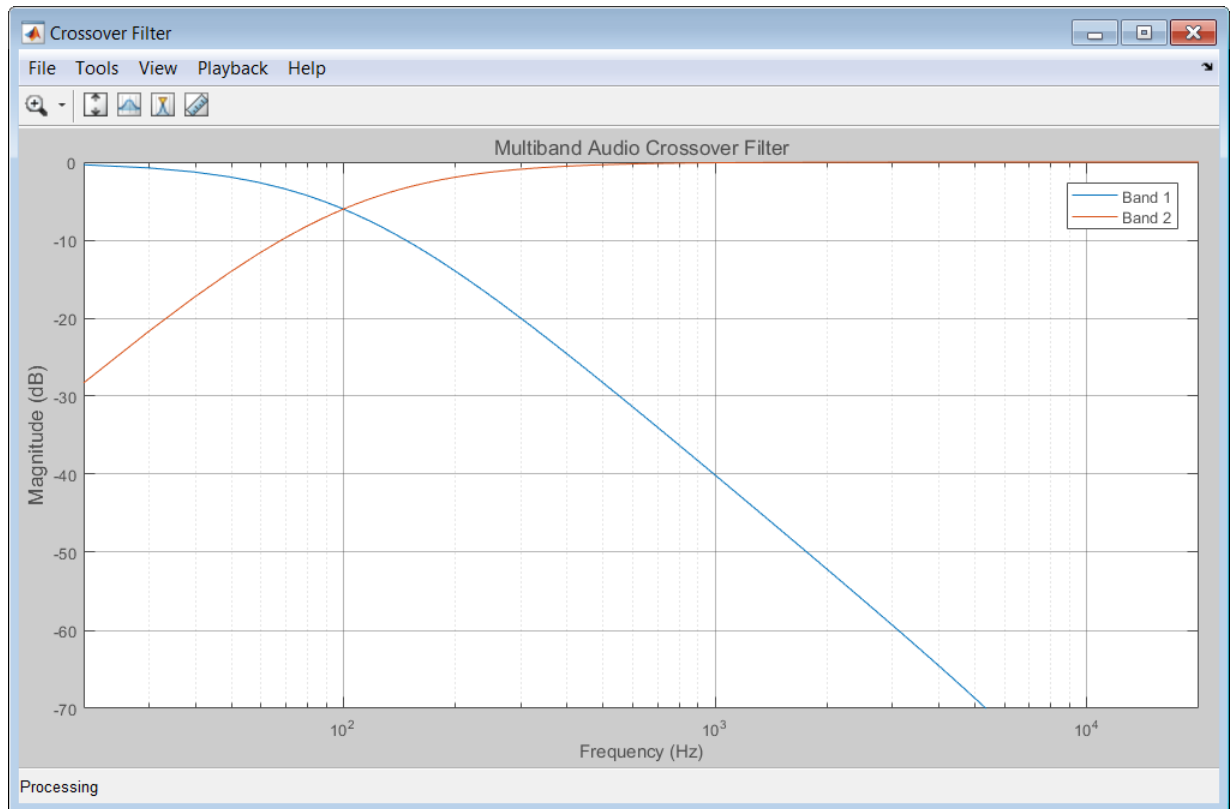
Crossover filter order, specified as 1, 2, 3, 4, 5, 6, 7, or 8. The default is 2.

Crossover filter order relates to crossover filter slope in dB/octave: $slope = N \times 6$, where N is the crossover order.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

View filter response

Plots the magnitude response of the **Crossover Filter** block, as defined by the parameters in the block dialog box. The plot updates when you tune parameters.



Inherit sample rate from input

When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**. By default, this check box is cleared.

Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default is 44100. You can specify an input sample rate when the **Inherit sample rate from input** check box is cleared.

Simulate using

Type of simulation to run. You can set this parameter to:

- Interpreted execution (default)

Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to **Code generation**. In this mode, you can debug the source code of the block.

- **Code generation**

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

Supported Data Types

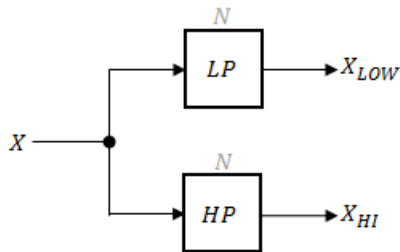
Port	Supported Data Types
Input	<ul style="list-style-type: none">• Double-precision floating point• Single-precision floating point
Output	<ul style="list-style-type: none">• Double-precision floating point• Single-precision floating point

Algorithms

The **Crossover Filter** block is implemented as a binary tree of crossover pairs with additional phase-compensating sections [1]. Odd-order crossovers are implemented with Butterworth filters, while even-order crossovers are implemented with cascaded Butterworth filters (Linkwitz-Riley filters).

Odd-Order Crossover Pair

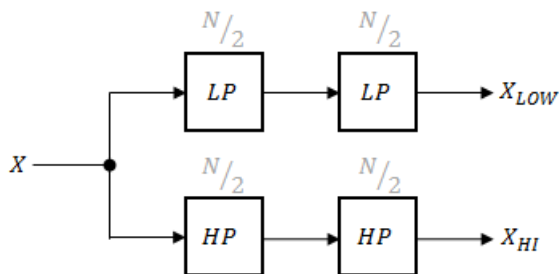
Odd-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.



LP and *HP* are Butterworth filters of order N , implemented as direct-form II transposed second-order sections. The shared cutoff frequency used in their design corresponds to the crossover of the resulting bands.

Even-Order Crossover Pair

Even-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.

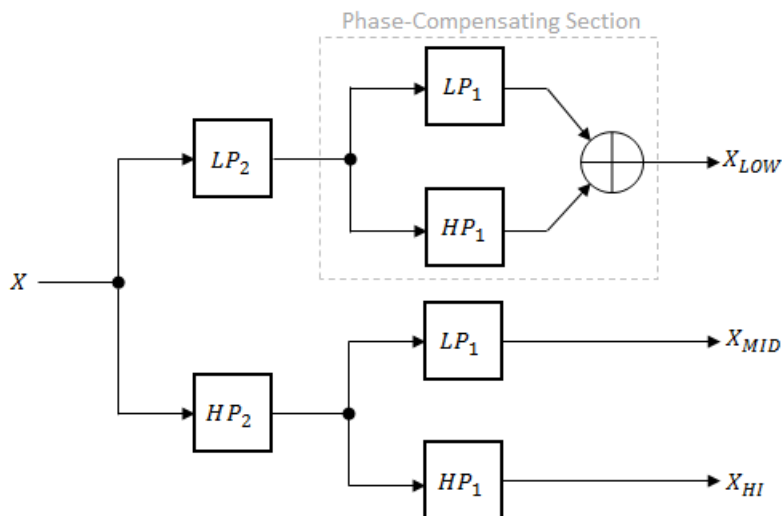


LP and *HP* are Butterworth filters of order $N/2$, where N is the order of the overall filter. The filters are implemented as direct-form II transposed second-order sections.

For overall filters of orders 2 and 6, X_{HI} is multiplied by -1 internally so that the branches of your crossover pair are in-phase.

Even-Order Three-Band Filter

Even-order three-band (two crossovers) filters are implemented as parallel complementary highpass and lowpass filters organized in a tree structure.



The phase-compensating section is equivalent to an allpass filter.

The design of four-band and five-band filters (three and four crossovers) are extensions of the pattern developed for even-order and odd-order crossovers and the tree structure specified for three-band (two crossover) filters.

References

- [1] D'Appolito, Joseph A. "Active Realization of Multiway All-Pass Crossover Systems". *Journal of Audio Engineering Society*. Vol. 35, Issue 4, pp. 239–245.

See Also

crossoverFilter

Related Examples

- "Multiband Dynamic Range Compression"

Introduced in R2016a

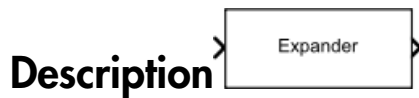
Expander

Dynamic range expander

Library

Dynamic Range Control

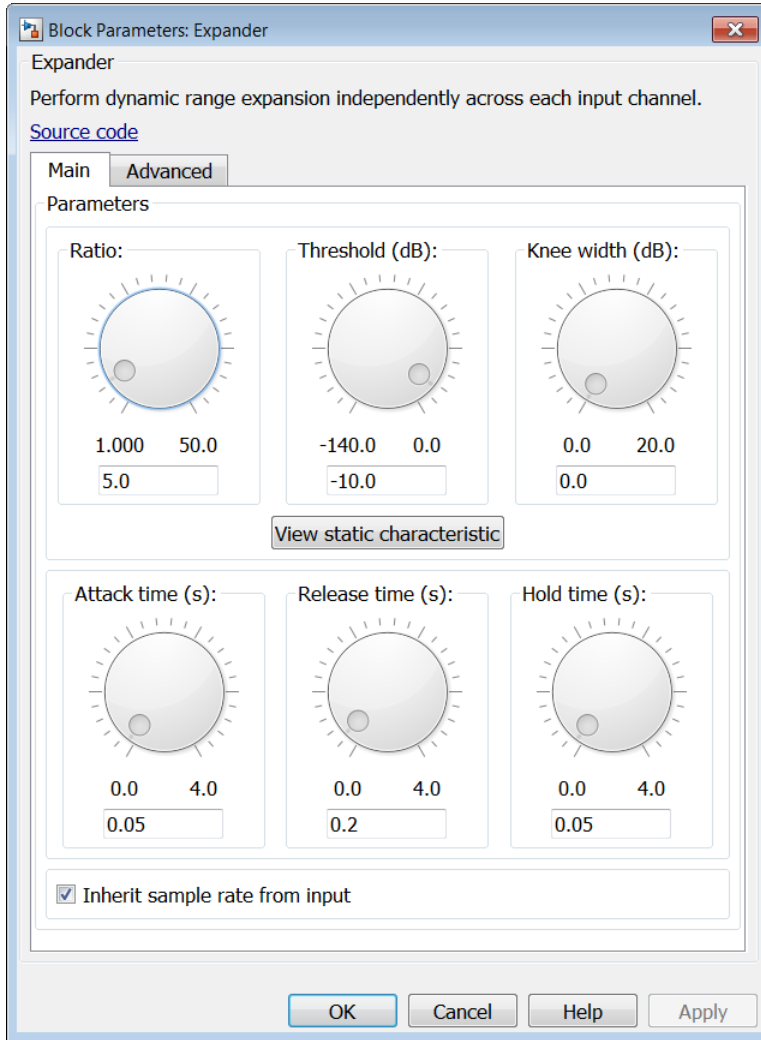
audiodynamicrange



The Dynamic Range Expander block performs dynamic range expansion independently across each input channel. Dynamic range expansion attenuates the volume of quiet sounds below a given threshold. It uses specified attack, release, and hold times to achieve a smooth applied gain curve. You can tune parameters of the Dynamic Range Expander block to meet your processing needs.

The input must be a real-valued, double-precision or single-precision matrix. The Dynamic Range Expander block treats each column of the input as an independent channel.

Dialog Box



Main Tab

Ratio

Expansion ratio, specified as a real scalar in the range 1 to 50. The default is 5.

Expansion ratio is the input/output ratio for signals that undershoot the operation threshold.

Assuming a hard knee characteristic and a steady-state input such that $x[n]$ dB <

Threshold (dB), the expansion ratio is defined as
$$R = \frac{(y[n]-T)}{(x[n]-T)}$$
.

- R is the expansion ratio.
- $y[n]$ is the output signal in dB.
- $x[n]$ is the input signal in dB.
- T is the threshold in dB.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Threshold (dB)

Operation threshold in dB, specified as a real scalar in the range -140 to 0 . The default is -10 dB.

Operation threshold is the level below which gain is applied to the input signal.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Knee width (dB)

Knee width in dB, specified as a real scalar in the range 0 to 20 . The default is 0 dB.

Knee width is the transition area in the expansion characteristic.

For soft knee characteristics, the transition area is defined by the relation

$$y = x + \frac{(1-R) \times \left(x - T - \frac{W}{2}\right)^2}{(2 \times W)}$$

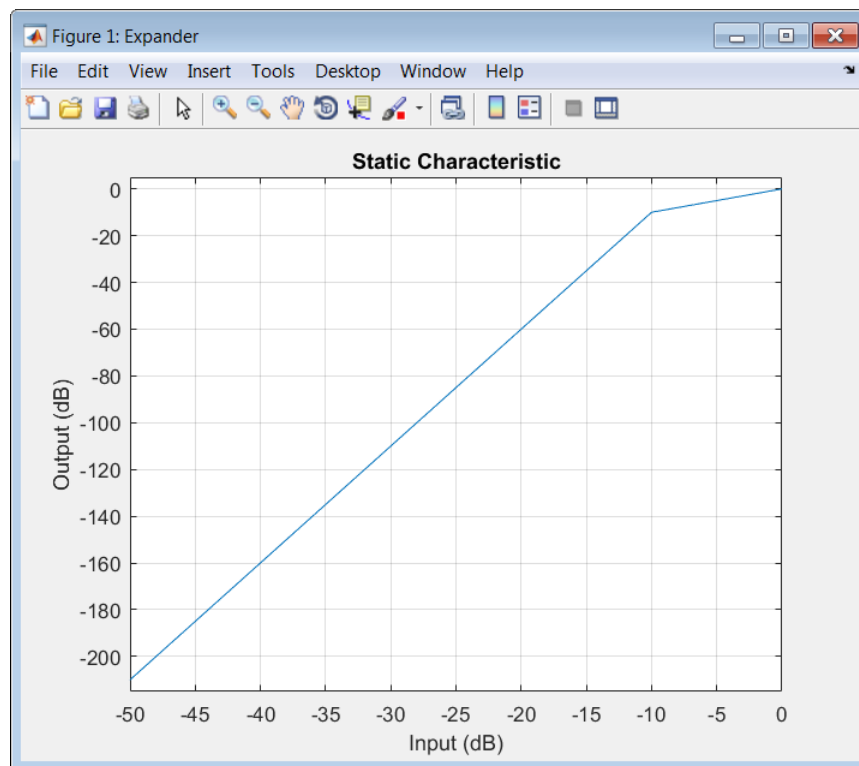
for the range $(2 \times |x - T|) \leq W$.

- y is the output level in dB.
- x is the input level in dB.
- R is the expansion ratio.
- T is the threshold in dB.
- W is the knee width in dB.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

View static characteristic

Plots the static expansion characteristic of the Dynamic Range Expander block, as defined by the parameters in the block dialog box. The plot updates when you tune parameters.



Attack time (s)

Attack time in seconds, specified as a real scalar in the range 0 to 4. The default is 0.05 seconds.

Attack time is the time it takes the expander gain to rise from 10% to 90% of its final value when the input goes below the threshold.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Release time (s)

Release time in seconds, specified as a real scalar in the range 0 to 4. The default is 0.2 seconds.

Release time is the time it takes the expander gain to drop from 90% to 10% of its final value when the input goes above the threshold.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Hold time (s)

Hold time in seconds, specified as a real scalar in the range 0 to 4. The default is 0.05 seconds.

Hold time is the period in which the applied gain is held constant before it starts moving toward its steady-state value. Hold time begins when the input level crosses the operation threshold.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

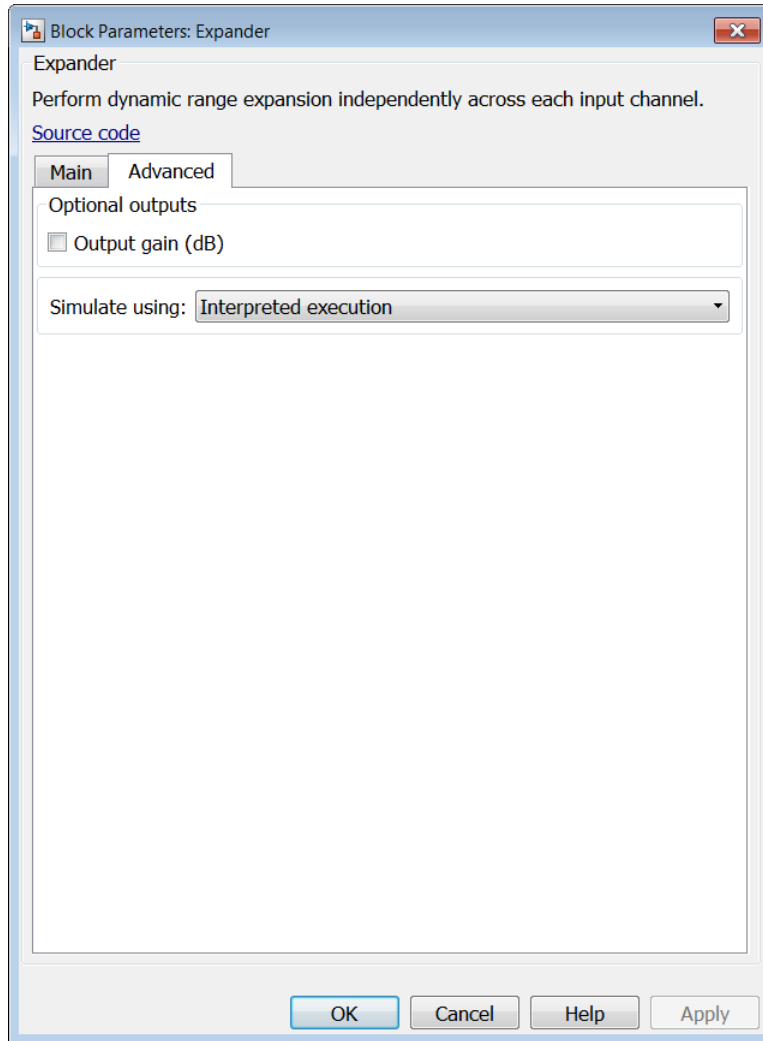
Inherit sample rate from input

When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**. By default, this check box is selected.

Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default is 44100 Hz. You can specify an input sample rate when the **Inherit sample rate from input** check box is cleared.

Advanced Tab



Output gain (dB)

When you select this check box, an additional output port, G, is added to the block. The G port outputs the gain applied on each input channel in dB. By default, this check box is cleared.

Simulate using

Type of simulation to run. You can set this parameter to:

- **Interpreted execution** (default)

Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to **Code generation**. In this mode, you can debug the source code of the block.

- **Code generation**

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

Supported Data Types

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

See Also

expander | Compressor | Limiter | Noise Gate

Introduced in R2016a

Limiter

Dynamic range limiter

Library

Dynamic Range Control

`audiodynamicrange`

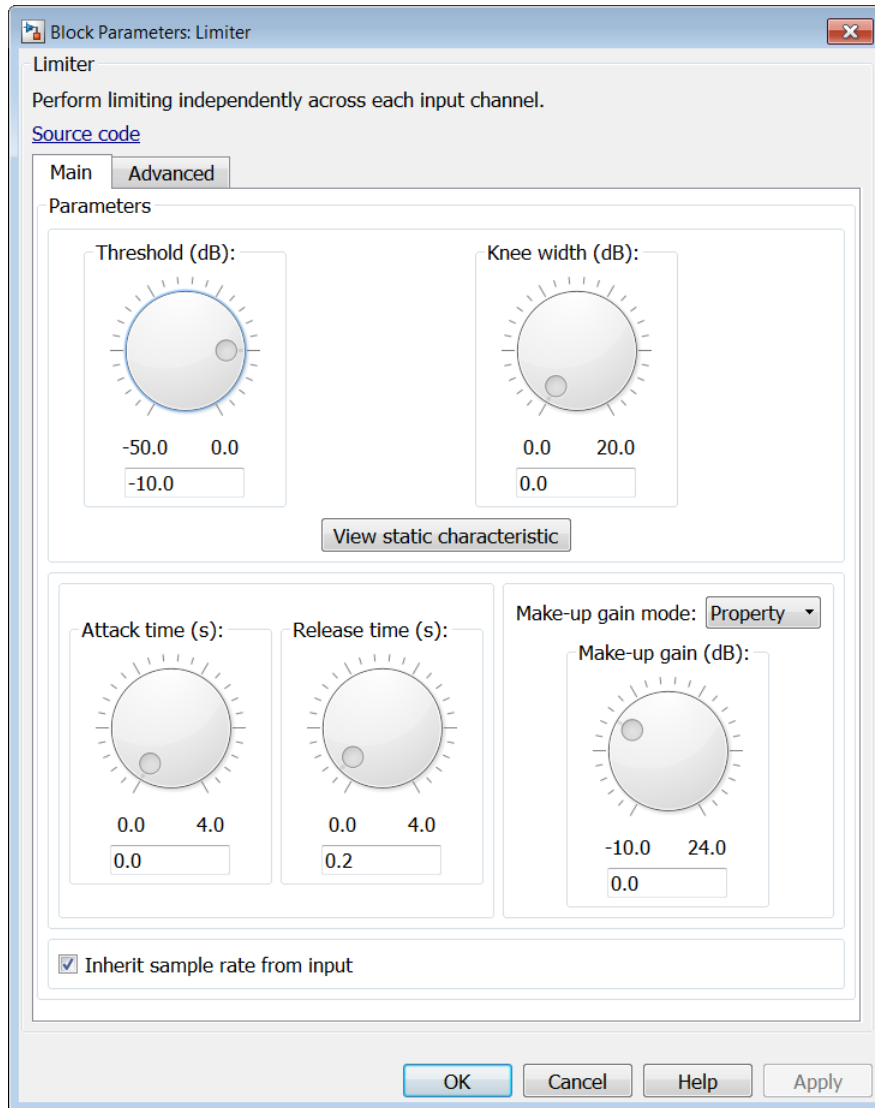
Description



The `Limiter` block performs dynamic range limiting independently across each input channel. Dynamic range limiting suppresses the volume of loud sounds that cross a given threshold. It uses specified attack and release times to achieve a smooth applied gain curve. You can tune parameters of the `Limiter` block to meet your processing needs.

The input must be a real-valued, double-precision or single-precision matrix. The `Limiter` block treats each column of the input as an independent channel.

Dialog Box



Main Tab

Threshold (dB)

Operation threshold in dB, specified as a real scalar in the range -50 to 0 . The default is -10 dB.

Operation threshold is the level above which gain is applied to the input signal.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Knee width (dB)

Knee width in dB, specified as a real scalar in the range 0 to 20 . The default is 0 dB.

Knee width is the transition area in the limiter characteristic.

For soft knee characteristics, the transition area is defined by the relation

$$y = x - \frac{\left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

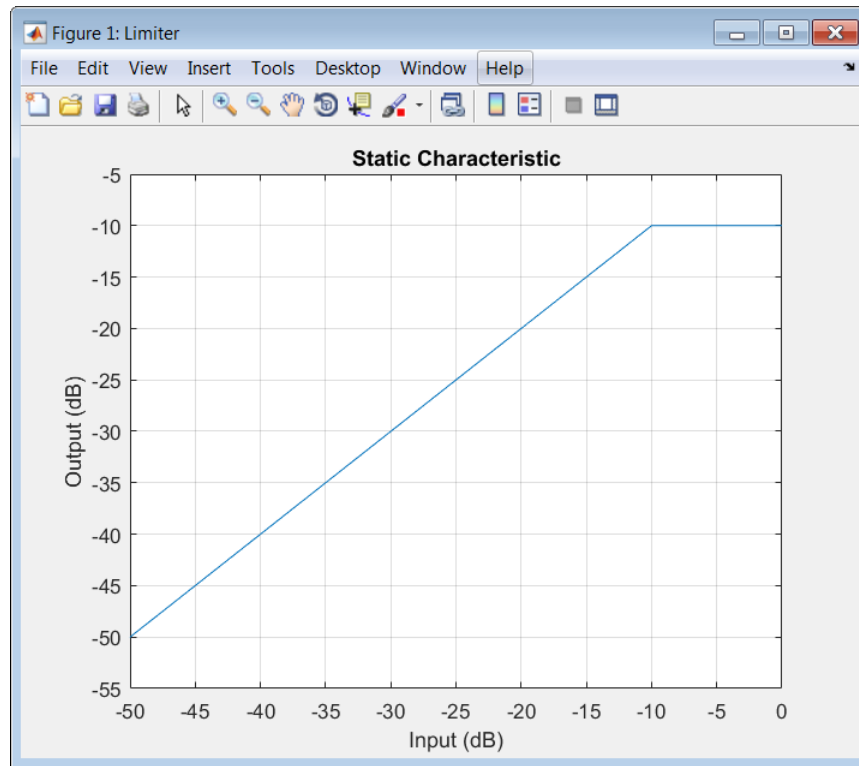
for the range $(2 \times |x - T|) \leq W$.

- y is the output level in dB.
- x is the input level in dB.
- T is the threshold in dB.
- W is the knee width in dB.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

View static characteristic

Plots the static limiter characteristic of the `Limiter` block, as defined by the parameters in the block dialog box. The plot updates when you tune parameters.



Attack time (s)

Attack time in seconds, specified as a real scalar in the range 0 to 4. The default is 0 seconds.

Attack time is the time it takes the limiter gain to rise from 10% to 90% of its final value when the input goes above the threshold.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Release time (s)

Release time in seconds, specified as a real scalar in the range 0 to 4. The default is 0.2 seconds.

Release time is the time it takes the limiter gain to drop from 90% to 10% of its final value when the input goes below the threshold.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Make-up gain mode

Make-up gain mode, specified as **Auto** or **Property**. The default is **Property**.

- **Auto** — Make-up gain is applied at the output of the **Limiters** block such that a steady-state 0 dB input has a 0 dB output.
- **Property** — Make-up gain is set to the value specified by **Make-up gain (dB)**.

Make-up gain (dB)

Make-up gain in dB, specified as a real scalar in the range -10 to 24 . The default is 0 dB.

Make-up gain compensates for gain lost during limiting. It is applied at the output of the **Limiters** block. This parameter is available when you set **Make-up gain mode** to **Property**.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

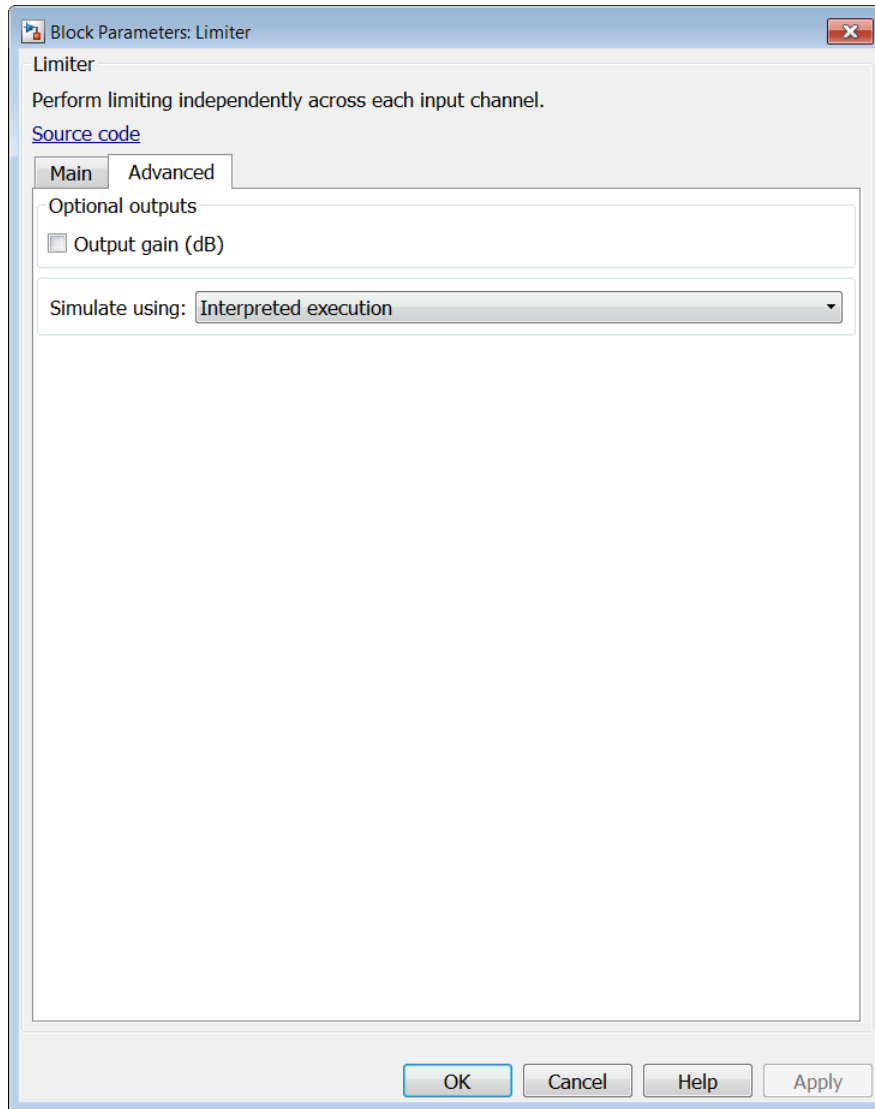
Inherit sample rate from input

When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**. By default, this check box is selected.

Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default is 44100 Hz. You can specify an input sample rate when the **Inherit sample rate from input** check box is cleared.

Advanced Tab



Output gain (dB)

When you select this check box, an additional output port, G, is added to the block. The G port outputs the gain applied on each input channel in dB. By default, this check box is cleared.

Simulate using

Type of simulation to run. You can set this parameter to:

- **Interpreted execution** (default)

Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to **Code generation**. In this mode, you can debug the source code of the block.

- **Code generation**

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

Supported Data Types

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

See Also

limiter | Compressor | Expander | Noise Gate

Introduced in R2016a

MIDI Controls

Output values from controls on MIDI control surface

Library

Sources

audiosources

Description



The MIDI Controls block outputs values from controls on a MIDI control surface in real time.

Use the **MIDI device** parameter to specify the name of the MIDI control surface device from which to receive control values. You can choose:

- Default
- Specify other

If you choose **Default**, the block looks for a MATLAB preference with a group named **midi** and preference named **DefaultDevice**. You can set this preference using the MATLAB `setpref` function. For example, if the desired device is named **BCF2000**, you can type the following command at the MATLAB command line:

```
>> setpref('midi', 'DefaultDevice', 'BCF2000');
```

If the block does not find this preference, it then attempts to choose a device using an algorithm that is unspecified and platform dependent.

If you choose **Specify other**, then a **MIDI device name** edit box appears for you to enter a MATLAB expression for the device name. Enter any MATLAB expression

that can evaluate to a string. Literal names must be enclosed in quotes, (for example, 'BCF2000').

You can determine the name of your MIDI device using the MATLAB function `midid`, discussed in “Identifying MIDI Device Names and Control Numbers” on page 5-55.

Use the **MIDI controls** parameter to specify the controls on the MIDI device to which the block should respond. This parameter also determines the size of the block output port. You can choose:

- Respond to any control
- Respond to specified controls

If you choose **Respond to any control**, then the block output will be a scalar. This scalar outputs the value from any and all controls that are manipulated on the MIDI device. Use this option in simple cases when you need only a single control value and the control to which it responds is unimportant.

If you choose **Respond to specified controls**, then a **MIDI control numbers** edit box opens. In this box, enter a MATLAB expression for the device control numbers. Enter any MATLAB expression that can evaluate to a row vector of real double-precision values. The block outputs a 1-D vector with one element corresponding to the output of each specified control.

Use the **Initial values** parameter to specify the value of the block output when simulation starts. The MIDI protocol transmits control values only when a control changes. This protocol provides no means for the block to query the current value of a control. Thus, the block must have some initial value to output until it receives a control change from the device.

Use the **Send initial values to device at start** check box to synchronize the device controls with the block outputs when simulation starts. Some MIDI control surfaces are bidirectional, meaning that they not only send control values but can also receive them. For example, some devices have motorized controls that move to the appropriate position when they receive a control value. If you have such a bidirectional device, select this check box. The block attempts to send the initial values to the device when the simulation starts. No diagnostic message appears if the attempt fails.

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy. However you must account for these extra .dll files when doing so. The `packNGo` function creates a single .zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

Output Port

The MIDI Controls block output is a vector whose width is determined by the **MIDI controls** and **MIDI control numbers** parameters previously described. The output data type can be either real double-precision floating point, or uint8 integer if the output mode is 'Raw MIDI'. The output values range from 0.0 to 1.0, inclusively, and in the raw mode, they range from 0 to 127, inclusively. The output port back inherits its sample time.

Identifying MIDI Device Names and Control Numbers

To specify a particular control on a particular MIDI device, you must know the name assigned to the device by the operating system. In addition, a number is always associated with the control. You can interactively discover this information using the MATLAB function, `midiid`. Follow these steps to identify device names and control numbers:

- 1 Verify that MIDI control surface device is correctly connected to the host computer running MATLAB.

Note: For the most consistent behavior, MathWorks recommends that you connect your MIDI control surface device to your computer before starting MATLAB. In some circumstances MATLAB may not be able to find your device if you connect it after starting your MATLAB session. Also, it may not find your device if you disconnect it and reconnect it during your MATLAB session.

- 2 Type the following command at the MATLAB command line.

```
>> [ctlnum devname] = midiid
```

You are prompted to move the control in which you are interested.

```
>> [ctlnum devname] = midiid
Move the control you wish to identify; type ^C to abort.
Waiting for control message ...
```

- 3 Move the control. `midiid` detects the movement and returns the device name and control number.

```
>> [ctlnum devname] = midiid
Move the control you wish to identify; type ^C to abort.
Waiting for control message ... done
```

```

ctlnum =
    1081
devname =
    BCF2000
>>

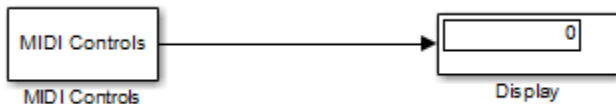
```

- 4 Use the device name in the block dialog, or set it as the default device using `setpref`. Then, enter the control number in the block dialog. Concatenate the number with other control numbers as needed.

Examples

How to Output Values from Controls

Use this example to familiarize yourself with how to set controls in the MIDI Controls block as it interacts with the MIDI control surface.



Open the `ex_simplemidi` model, and follow these steps:

Connect a MIDI device to the computer.

Use `midid` to determine the name of the device, and set it on the MIDI Controls block.

Verify that any control changes the display value.

Use `midid` to determine the number of a particular control, and set that on the MIDI Controls block.

Verify that a particular control changes the display value and that other controls do not.

Use `midid` to determine the number of a few more controls, and set those on the MIDI Controls block.

Verify that the display block shows the correct number of values. Also verify that the controls you specified change the appropriate display values and that the other controls do not change the values.

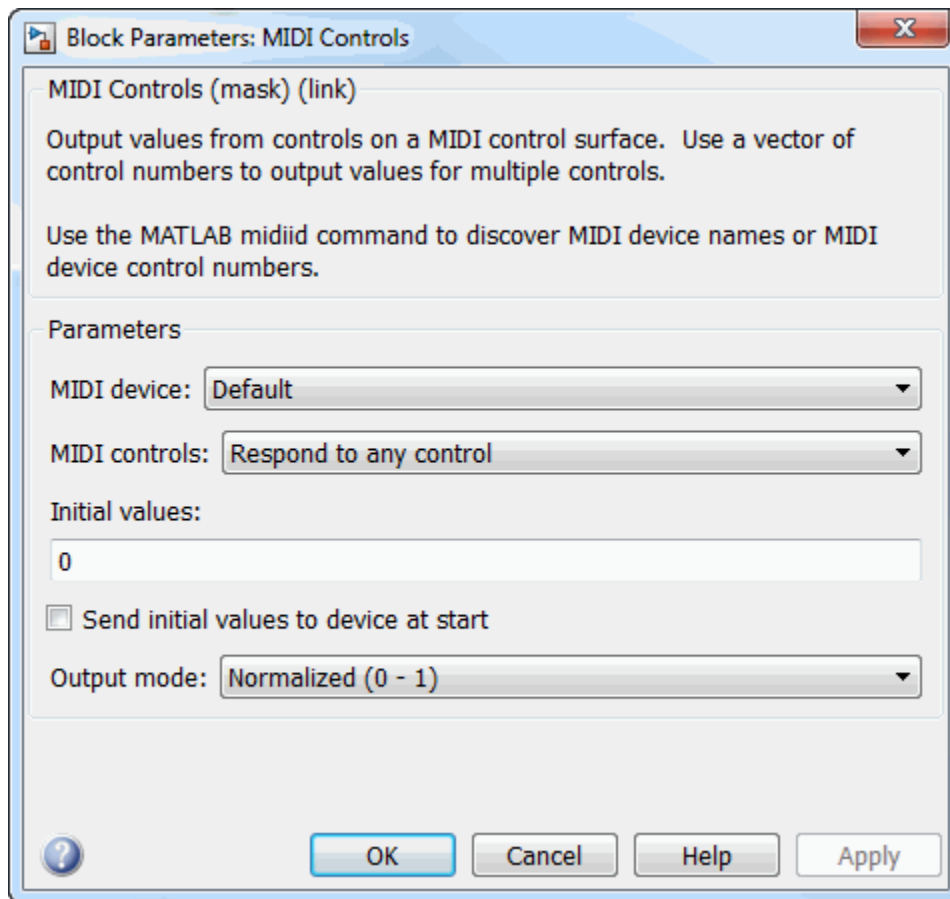
Set each control to have a unique initial value.

Verify that the correct initial values appear on the display when the model starts.

If your MIDI device is bidirectional, on the MIDI Controls block, select the **Send initial values to device at start** check box.

Verify that the controls are set to the correct initial values when the model starts.

Dialog Box



MIDI device

Specify whether to use a default MIDI device, or specify a particular device by name.

MIDI device name

Specify the name of a particular MIDI control surface device from which to receive control values.

MIDI controls

Specify whether to respond to any control on the MIDI device or respond to particular specified controls.

MIDI control numbers

Specify particular controls to which the block should respond.

Initial values

Specify initial values to output when simulation starts.

Send initial values to device at start

Select this check box to attempt to synchronize a bidirectional MIDI device with block initial values when simulation starts.

Output mode

Specify the mode in which the control values are generated. When you set **Output mode** to **Normalized (0-1)**, the block generates control values in the range [0 1]. In this mode, control values are represented as a fraction of a full-scale. Hence, you can easily scale this range to your particular application. When you set **Output mode** to **RAW MIDI (0-127)**, the block generates byte-oriented MIDI control values in the range [0 127]. By default, this parameter is set to **Normalized (0-1)**.

Supported Data Types

Port	Supported Data Types
Output	• Double-precision floating point, uint8 integer

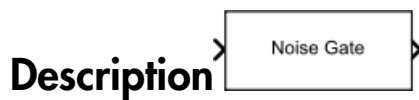
Noise Gate

Dynamic range gate

Library

Dynamic Range Control

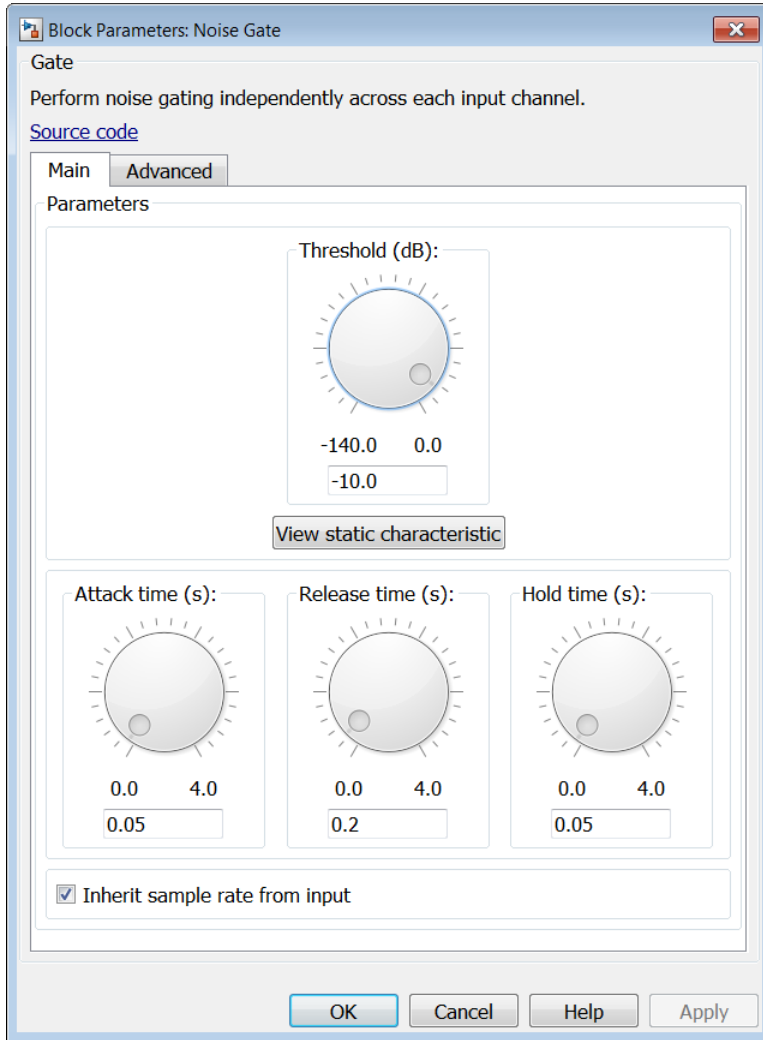
audiodynamicrange



The `Noise Gate` block performs dynamic range gating independently across each input channel. Dynamic range gating suppresses signals below a given threshold. It uses specified attack, release, and hold times to achieve a smooth applied gain curve. You can tune parameters of the `Noise Gate` block to meet your processing needs.

The input must be a real-valued, double-precision or single-precision matrix. The `Noise Gate` block treats each column of the input as an independent channel.

Dialog Box



Main Tab

Threshold (dB)

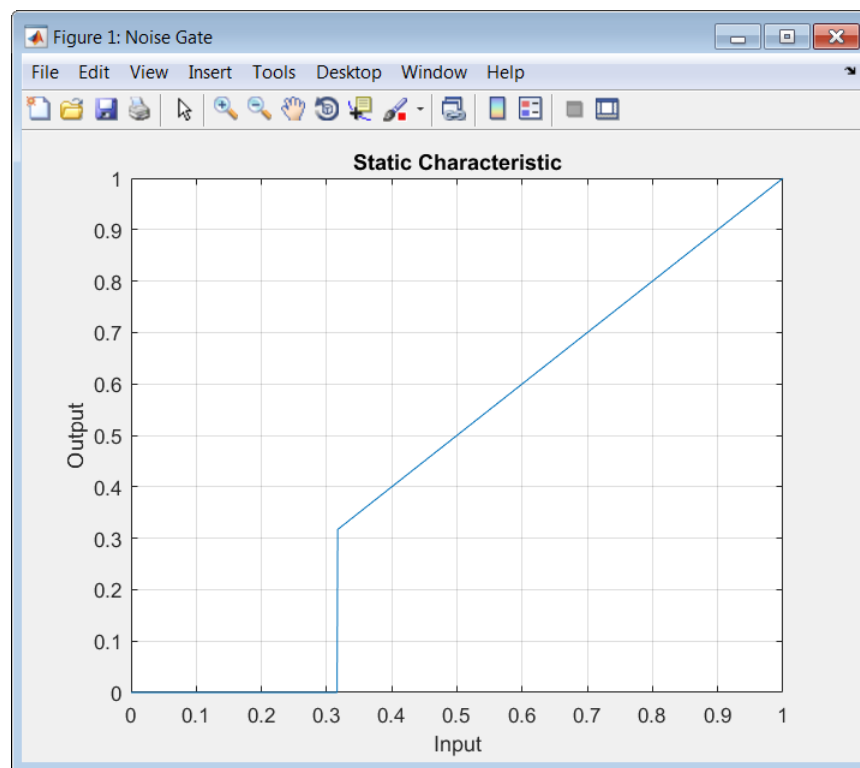
Operation threshold in dB, specified as a real scalar in the range -140 to 0 . The default is -10 dB.

Operation threshold is the level below which gain is applied to the input signal.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

View static characteristic

Plots the static gate characteristic of the **Noise Gate** block, as defined by the parameters in the block dialog box. The plot updates when you tune parameters.



Attack time (s)

Attack time in seconds, specified as a real scalar in the range 0 to 4 . The default is 0.05 seconds.

Attack time is the time it takes the applied gain to rise from 10% to 90% of its final value when the input goes below the threshold.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Release time (s)

Release time in seconds, specified as a real scalar in the range 0 to 4. The default is 0.2 seconds.

Release time is the time it takes the applied gain to drop from 90% to 10% of its final value when the input goes above the threshold.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Hold time (s)

Hold time in seconds, specified as a real scalar in the range 0 to 4. The default is 0.05 seconds.

Hold time is the period in which the applied gain is held constant before it starts moving toward its steady-state value. Hold time begins when the input level crosses the operation threshold.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

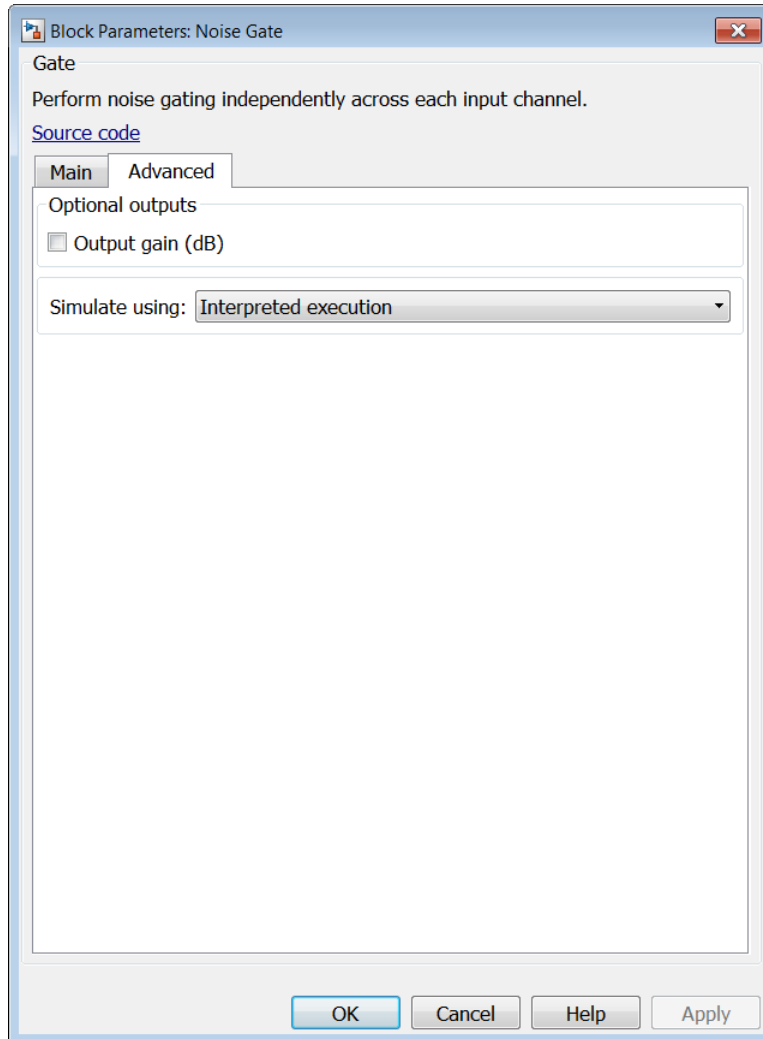
Inherit sample rate from input

When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**. By default, this check box is selected.

Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default is 44100 Hz. You can specify an input sample rate when the **Inherit sample rate from input** check box is cleared.

Advanced Tab



Output gain (dB)

When you select this check box, an additional output port, G, is added to the block. The G port outputs the gain applied on each input channel in dB. By default, this check box is cleared.

Simulate using

Type of simulation to run. You can set this parameter to:

- Interpreted execution (default)

Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to Code generation. In this mode, you can debug the source code of the block.

- Code generation

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

Supported Data Types

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

See Also

noiseGate | Compressor | Expander | Limiter

Introduced in R2016a

Octave Filter

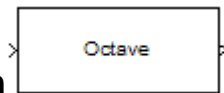
Design octave filter

Library

Filters

audiofilters

Description

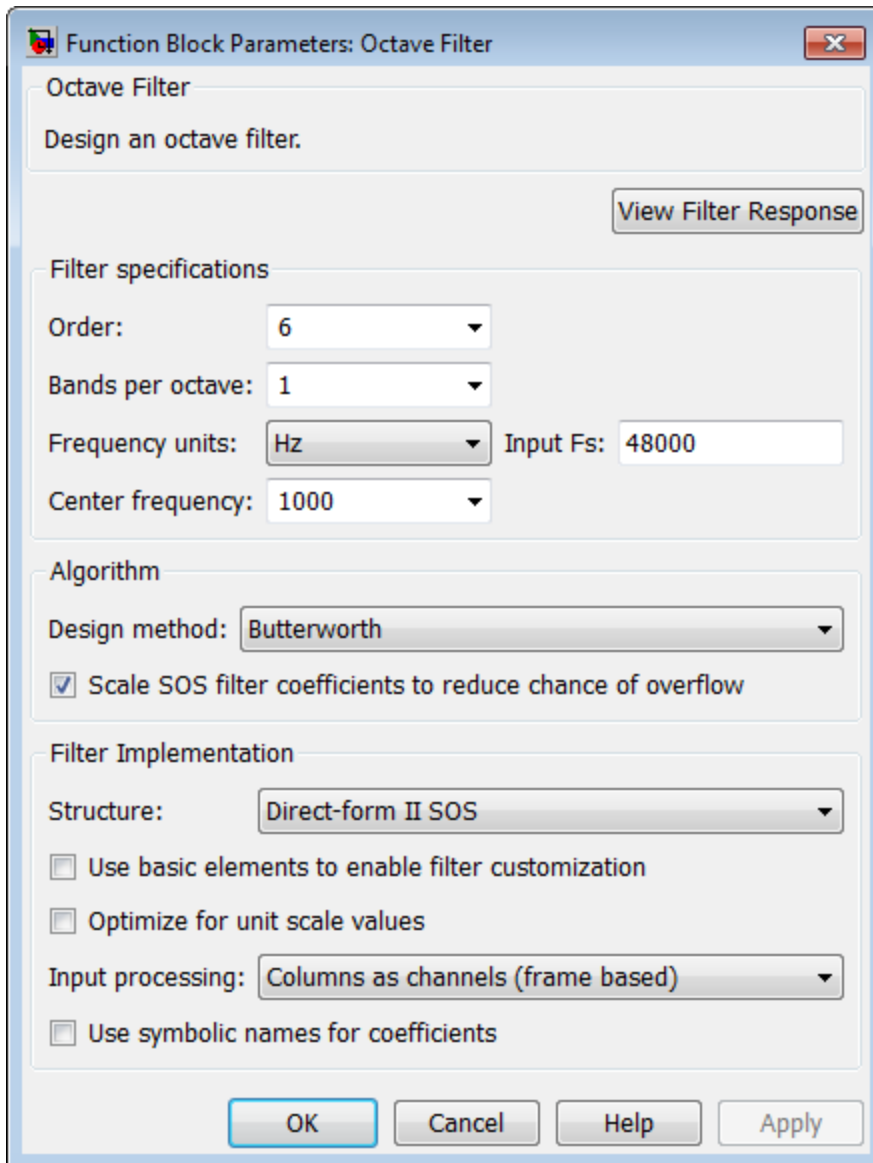


This block brings the filter design capabilities of the `filterbuilder` function to the Simulink environment.

Dialog Box

See “Octave Filter Design Dialog Box — Main Pane” for more information about the parameters of this block.

Parameters of this block that do not change filter order or structure are tunable.



View filter response

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

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

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

Filter Specifications

Order

Specify filter order. Possible values are: 4, 6, 8, 10.

Bands per octave

Specify the number of bands per octave. Possible values are: 1, 3, 6, 12, 24.

Frequency units

Specify frequency units as HZ or KHZ.

Input Fs

Specify the input sampling frequency in the frequency units specified previously.

Center Frequency

Select from the drop-down list of available center frequency values.

Algorithm

Design Method

Butterworth is the design method used for this type of filter.

Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

Filter Implementation

Structure

Specify filter structure. Choose from:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

Use basic elements to enable filter customization

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

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

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

Optimize for unit-scale values

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

Rate options

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

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

Use symbolic names for coefficients

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

Supported Data Types

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

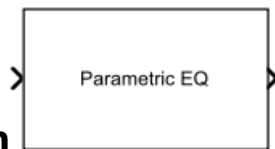
Parametric EQ Filter

Model second-order parametric equalizer filter

Library

Filters

audiofilters



Description

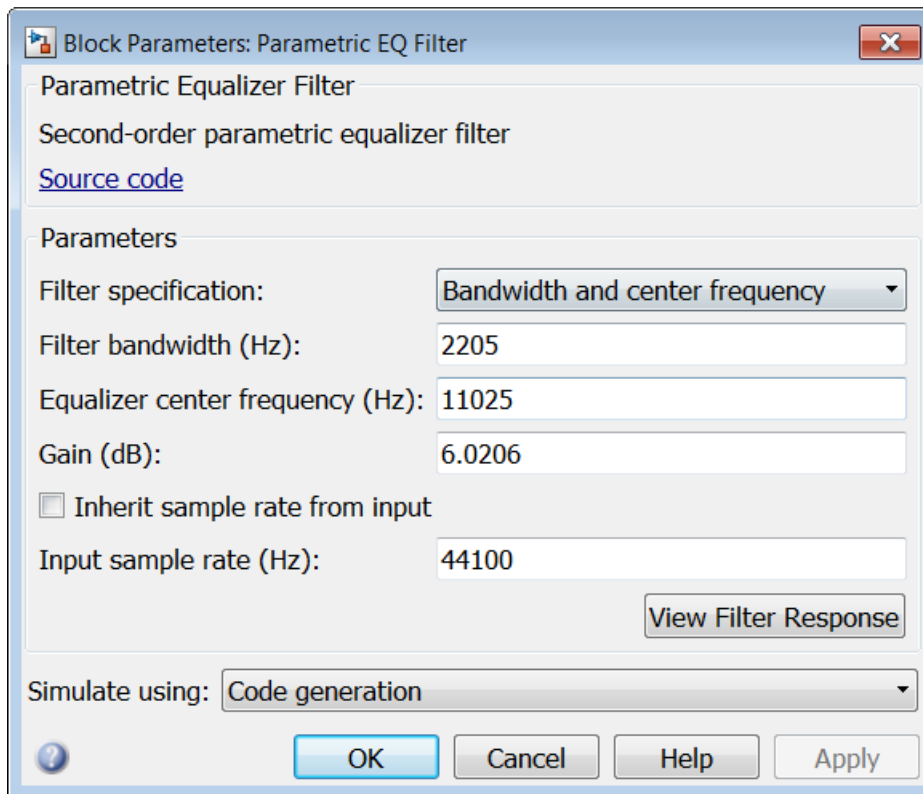
The Parametric EQ Filter block filters each channel of the input signal over time using a specified center frequency, bandwidth, and peak (dip) gain. This block offers tunable filter design parameters, which enable you to tune the filter characteristics while the simulation is running.

The block designs the filter according to the filter parameters set in the block dialog box. The output port properties, such as datatype, complexity, and dimension, are identical to the input port properties.

Each column of the input signal is treated as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then it is interpreted as a single channel.

This block supports variable-size input, enabling you to change the channel length during simulation. To enable variable-size input, clear the **Inherit sample rate from input** check box. The number of channels must remain constant.

Dialog Box



Filter specification

Parameters or coefficients used to design the filter, specified as one of the following:

- **Bandwidth and center frequency (default)** — Design the filter using **Filter bandwidth (Hz)**, **Equalizer center frequency (Hz)**, and **Gain (dB)**.
- **Coefficients** — Design the filter using **Bandwidth coefficient**, **Center frequency coefficient**, and **Gain (Linear Units)**.
- **Quality factor and center frequency** — Design the filter using **Equalizer center frequency (Hz)**, **Gain (dB)**, and **Quality factor**.

This parameter is nontunable.

Filter bandwidth (Hz)

Bandwidth of the filter, specified as a finite positive numeric scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter specification** to **Bandwidth** and **center frequency**. The default is 2205. This parameter is tunable.

Equalizer center frequency (Hz)

Center frequency of the filter, specified as a finite positive scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter specification** to **Bandwidth** and **center frequency** or **Quality factor** and **center frequency**. The default is 11025. This parameter is tunable.

Gain (dB)

Peak or dip gain of the filter, specified as a real scalar in dB. A value greater than zero corresponds to a peak. A value less than zero corresponds to a dip. This parameter applies when you set **Filter specification** to **Bandwidth** and **center frequency** or **Quality factor** and **center frequency**. The default is 6.0206. This parameter is tunable.

Bandwidth coefficient

Coefficient that determines the filter bandwidth, specified as a finite numeric scalar in the range $[-1 \ 1]$.

- -1 corresponds to the maximum bandwidth (one-fourth the sample rate of the input signal).
- 1 corresponds to the minimum bandwidth (0 Hz, that is, an allpass filter).

This parameter applies when you set **Filter specification** to **Coefficients**. The default is 0.72654. This parameter is tunable.

Center frequency coefficient

Coefficient that determines the center frequency of the filter, specified as a finite numeric scalar in the range $[-1 \ 1]$.

- -1 corresponds to the minimum center frequency (0 Hz).
- 1 corresponds to the maximum center frequency (half the sample rate of the input signal).

This parameter applies when you set **Filter specification** to **Coefficients**. The default is 0, which corresponds to one-fourth the sample rate of the input signal. This parameter is tunable.

Gain (Linear Units)

Peak or dip gain of the filter, specified as a real positive scalar in linear units. A value greater than one boosts the input signal. A value less than one attenuates the input signal. This parameter applies when you set **Filter specification** to **Coefficients**. The default is 2. This parameter is tunable.

Quality factor

Quality factor of the filter, specified as a real positive scalar. The quality factor is defined as **Equalizer center frequency (Hz) / Filter bandwidth (Hz)**. A higher quality factor corresponds to a narrower peak or dip. This parameter applies when you set **Filter specification** to **Quality factor and center frequency**. The default is 5. This parameter is tunable.

Inherit sample rate from input

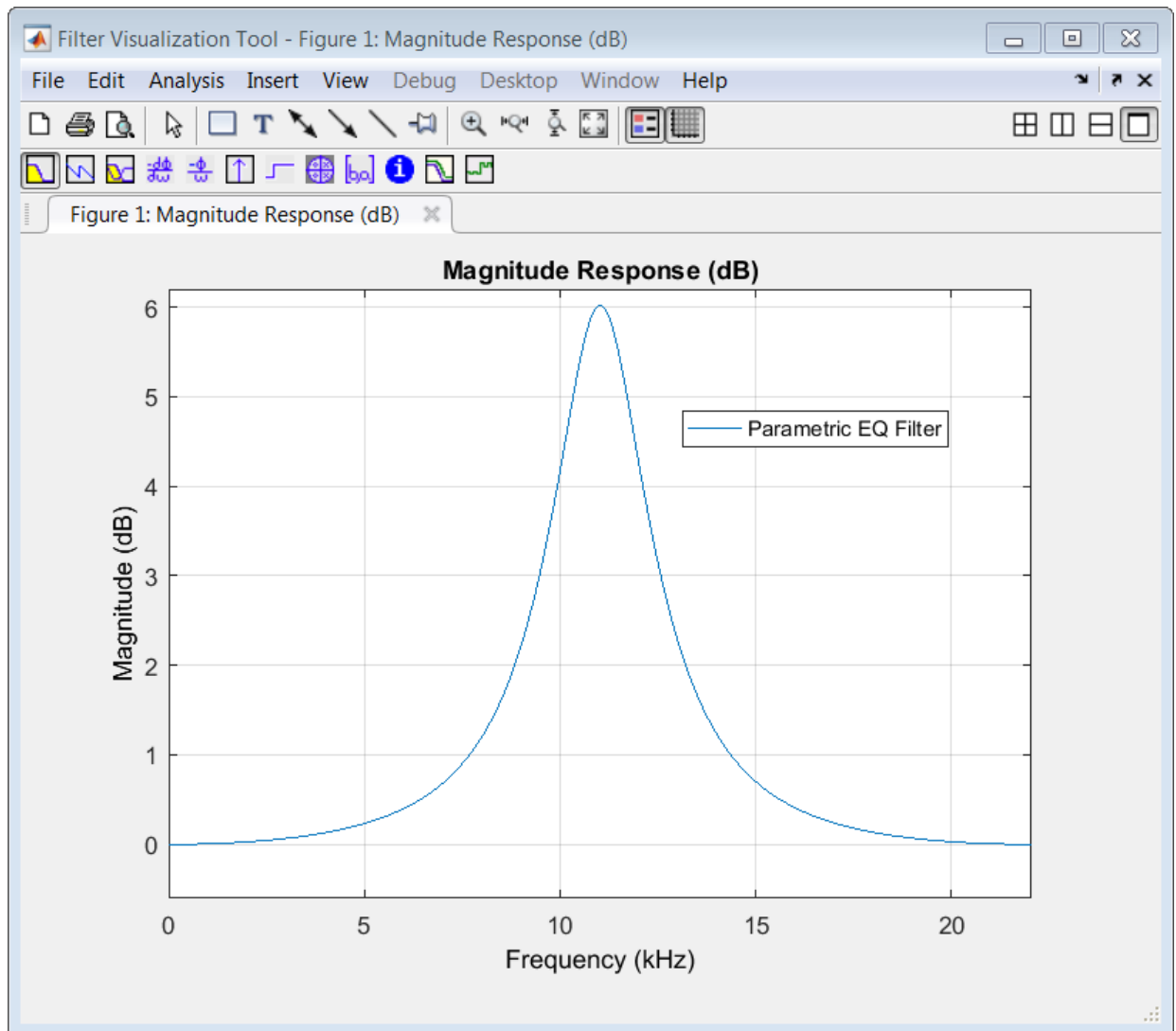
When you select this check box, the block's sample rate is computed as N/T_s , where N is the frame size of the input signal, and T_s is the sample time of the input signal. When you clear this check box, the block sample rate is the value specified in **Input sample rate (Hz)**. By default, this check box is selected.

Input sample rate (Hz)

Sample rate of the input signal, specified as a positive scalar value. The default is 44100. This parameter applies when you clear the **Inherit sample rate from input** check box. This parameter is nontunable.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the **Parametric EQ Filter**. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

- Code generation (default)

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

- Interpreted execution

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

Supported Data Types

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

Algorithms

This block brings the capabilities of the `dsp.ParametricEQFilter` System object to the Simulink environment.

The filter uses a coupled allpass structure to optimize joint computation of the peak and notch response. For information on the algorithms used by the Parametric EQ Filter block, see the “Algorithm” section of `dsp.ParametricEQFilter`.

References

- [1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

See Also

`multibandParametricEQ`

Reverberator

Add reverberation to audio signal

Library

Effects

audioeffects

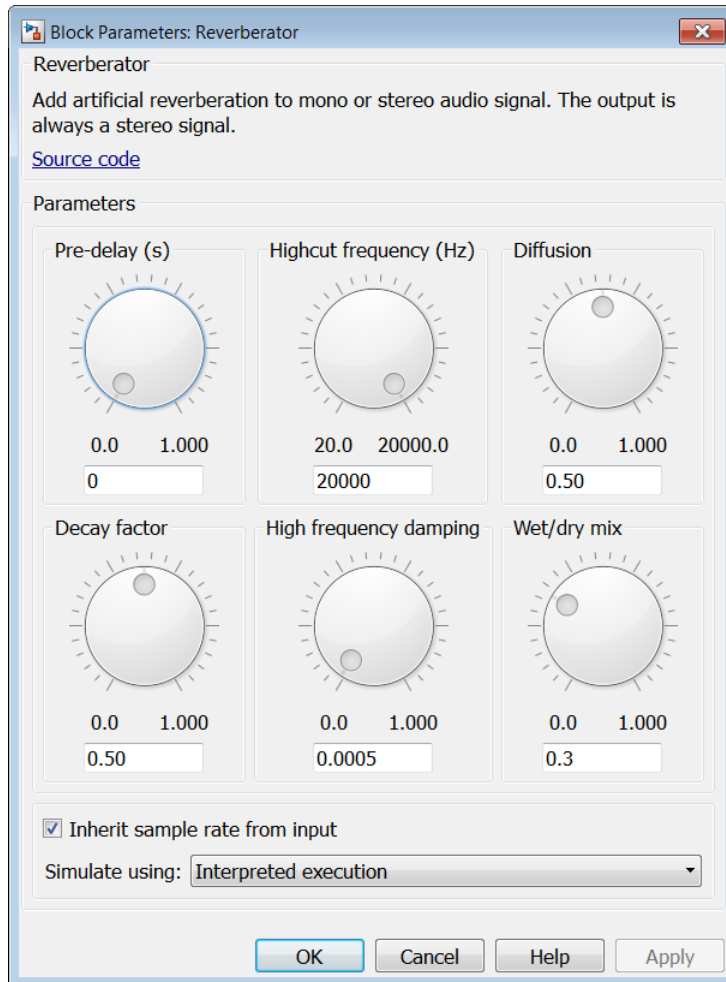
Description



The **Reverberator** block adds reverberation to mono or stereo audio signals. You can tune parameters of the **Reverberator** block to mimic different acoustic environments.

The input must be a real-valued, double-precision or single-precision matrix. The input matrix must have one or two columns, corresponding to a mono or stereo signal, respectively. The **Reverberator** block treats each column of the input as an independent channel. The output is always stereo.

Dialog Box



Pre-delay (s)

Pre-delay for reverberation in seconds, specified as a real scalar in the range 0 to 1. The default is 0 seconds.

Pre-delay for reverberation is the time between hearing direct sound and the first early reflection. The value of **Pre-delay (s)** is proportional to the size of the room being modeled.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Highcut frequency (Hz)

Lowpass filter cutoff in Hz, specified as a real positive scalar in the range 0 to

$\left(\frac{\text{SampleRate}}{2}\right)$. The default is 20000 Hz.

Lowpass filter cutoff is the -3 dB cutoff frequency for the single-pole lowpass filter at the front of the reverberator structure. It prevents the application of reverberation to high-frequency components of the input.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Diffusion

Density of reverb tail, specified as a real positive scalar in the range 0 to 1. The default is 0.50.

Diffusion is proportional to the rate at which the reverb tail builds in density. Increasing **Diffusion** pushes the reflections closer together, thickening the sound. Reducing **Diffusion** creates more discrete echoes.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Decay factor

Decay factor of reverb tail, specified as a real positive scalar in the range 0 to 1. The default is 0.50.

Decay factor is proportional to the time it takes for reflections to run out of energy. To model a large room, use a long reverb tail (low decay factor). To model a small room, use a short reverb tail (high decay factor).

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

High frequency damping

High-frequency damping, specified as a real positive scalar in the range 0 to 1. The default is 0.0005.

High frequency damping is proportional to the attenuation of high frequencies in the reverberation output. Setting **High frequency damping** to a large value makes high-frequency reflections decay faster than low-frequency reflections.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Wet/dry mix

Wet-dry mix, specified as a real positive scalar in the range 0 to 1. The default is 0.3.

Wet-dry mix is the ratio of wet (reverberated) signal to dry (original) signal that your **Reverberator** block outputs.

This parameter is tunable. You can change the value of this parameter even when the simulation is running.

Inherit sample rate from input

When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**. By default, this check box is selected.

Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default is 44100 Hz. You can specify an input sample rate when the **Inherit sample rate from input** check box is cleared.

Simulate using

Type of simulation to run. You can set this parameter to:

- **Interpreted execution (default)**

Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to **Code generation**. In this mode, you can debug the source code of the block.

- **Code generation**

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional

startup time but the speed of the subsequent simulations is comparable to Interpreted execution.

Supported Data Types

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

Algorithm

The algorithm to add reverberation is based on the plate-class reverberation topology described in [1].

References

- [1] Dattorro, Jon. "Effect Design, Part 1: Reverberator and Other Filters." *Journal of the Audio Engineering Society*. Vol. 45, Issue 9, pp. 660–684.

See Also

reverberator

Introduced in R2016a

