

**Audio Toolbox™**

Reference



**MATLAB® & SIMULINK®**

R2019b



# How to Contact MathWorks



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



Phone: 508-647-7000



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

## *Audio Toolbox™ Reference Guide*

© COPYRIGHT 2016 - 2019 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](http://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](http://www.mathworks.com/patents) for more information.

### **Revision History**

March 2016	Online only	New for Version 1.0 (Release 2016a)
September 2016	Online only	Revised for Version 1.1 (Release 2016b)
March 2017	Online only	Revised for Version 1.2 (Release 2017a)
September 2017	Online only	Revised for Version 1.3 (Release 2017b)
March 2018	Online only	Revised for Version 1.4 (Release 2018a)
September 2018	Online only	Revised for Version 1.5 (Release 2018b)
March 2019	Online only	Revised for Version 2.0 (Release 2019a)
September 2019	Online only	Revised for Version 2.1 (Release 2019b)

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



# Apps in Audio Toolbox

---

## Audio Labeler

Define and visualize ground-truth labels

### Description

The **Audio Labeler** app enables you to label ground-truth data at both the region level and file level.

Using the app, you can:

- Create label definitions for consistent and fast labeling.
- Visualize the time-domain waveform during playback.
- Interactively specify labels at the file level and region level. You can specify regions by drawing directly on the time-domain waveform.
- Record new audio to add to your dataset.
- Apply automatic labeling of detected speech regions.
- Apply automatic word labeling using third-party speech-to-text transcription services. See “Speech-to-Text Transcription” for more information.

The app exports data as a `labeledSignalSet` object. You can use `labeledSignalSet` to train a network, classifier, or analyze data and report statistics.

### Open the Audio Labeler App

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

### Examples

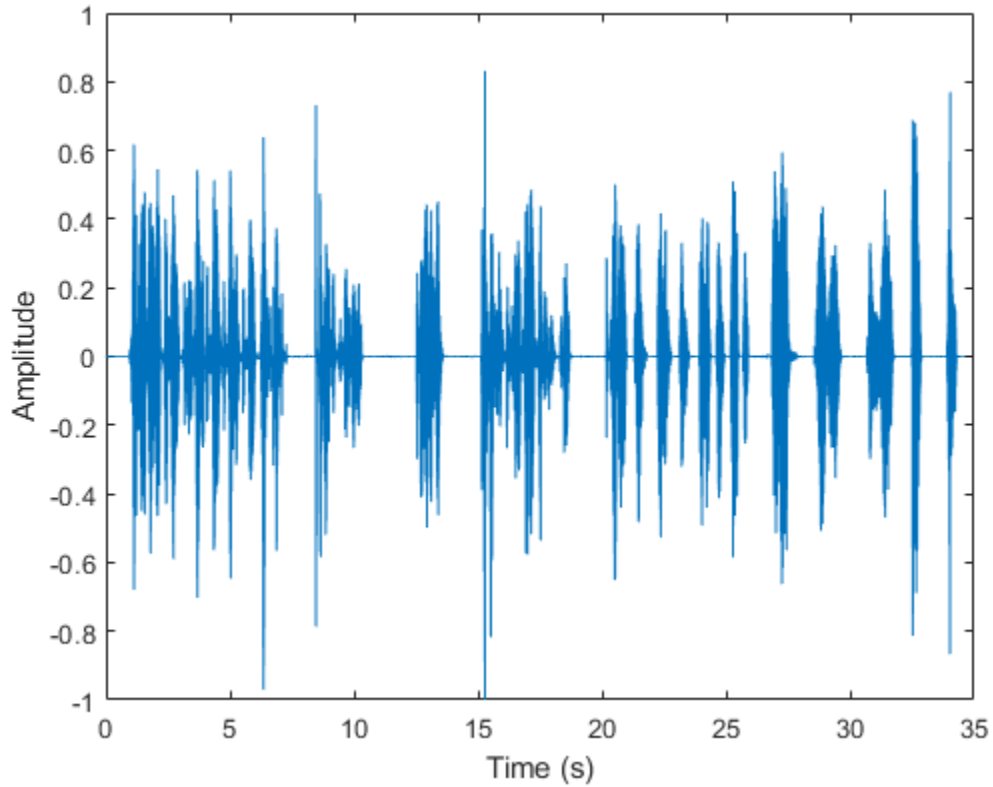
## Create Keyword Spotting Mask Using Audio Labeler

In this example, you create a logical mask for an audio signal where ones correspond to the utterance "yes" and zeros correspond to the absence of the utterance "yes". To create the mask, you use the IBM™ speech-to-text API through the **Audio Labeler** app.

This example requires that you install the "Speech-to-Text Transcription" functionality.

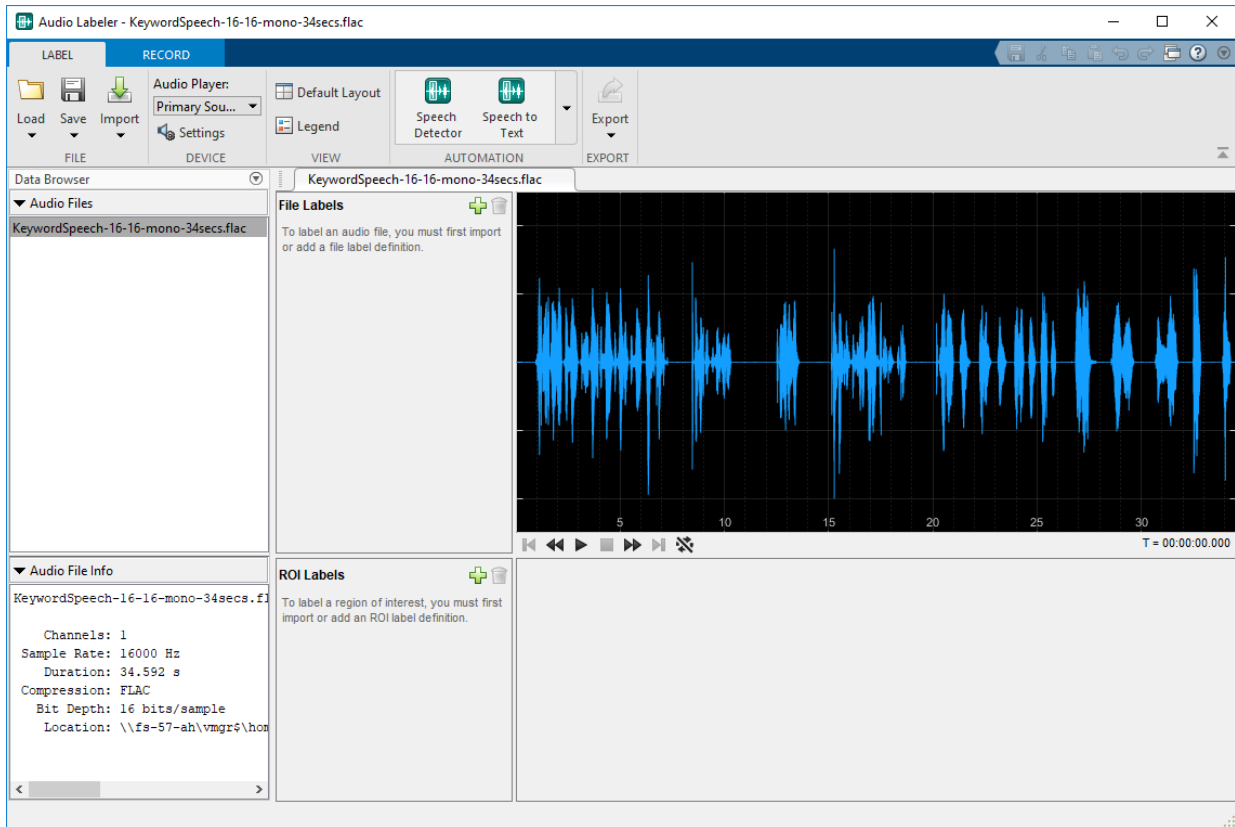
Listen to the audio file that you want to label and then visualize it in the time domain.

```
[audioIn,fs] = audioread("KeywordSpeech-16-16-mono-34secs.flac");  
  
sound(audioIn,fs)  
  
t = (0:numel(audioIn)-1)/fs;  
plot(t,audioIn)  
xlabel('Time (s)')  
ylabel('Amplitude')
```



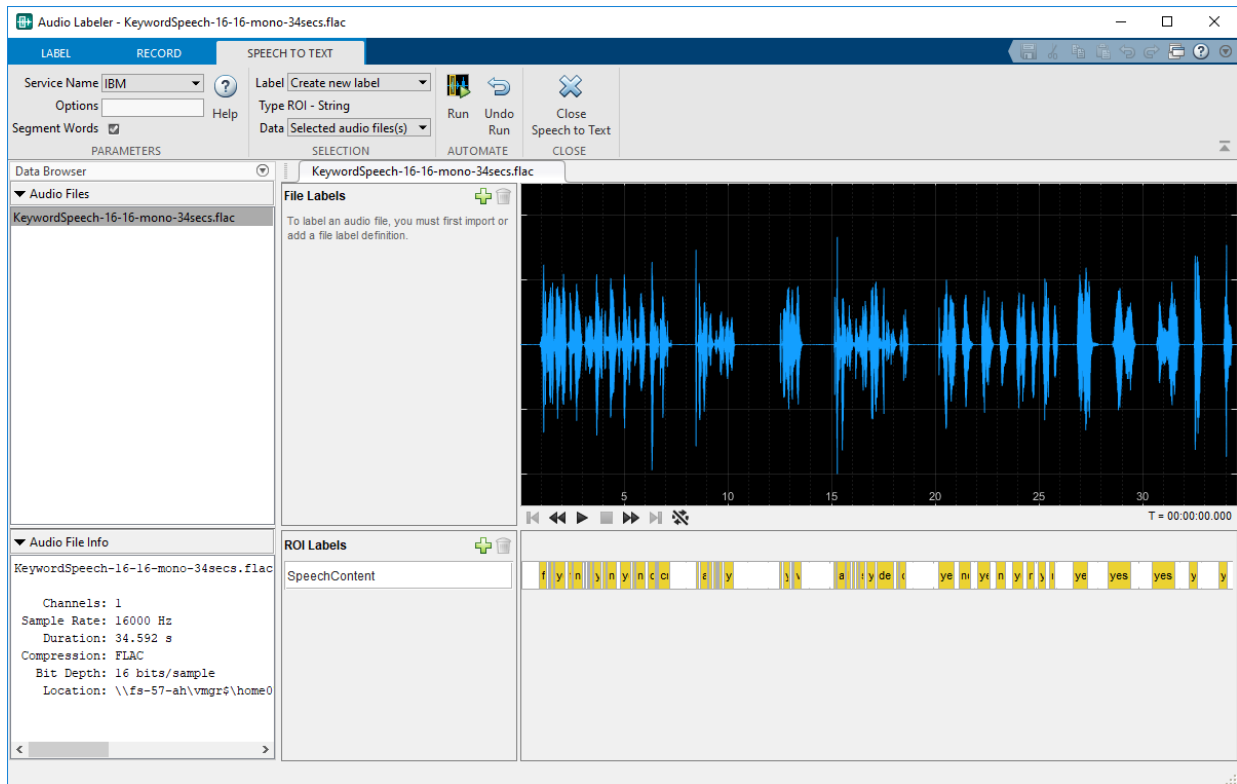
Open the **Audio Labeler** app and load the KeywordSpeech-16-16-mono-34secs.flac file into the **Data Browser**.



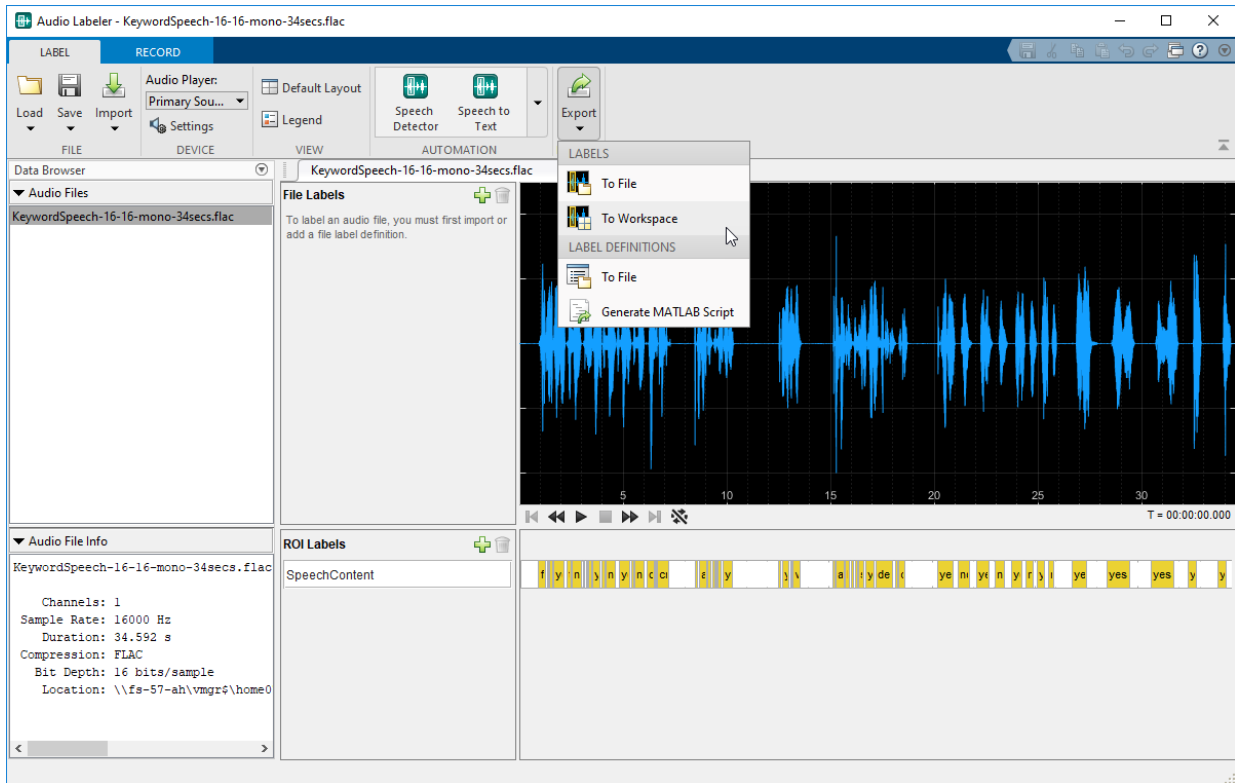


Under **Automation**, click **Speech to Text**. On the **Speech to Text** tab, select your preferred speech-to-text API. This example uses the IBM speech-to-text API. Select **Segment Words** so that the text labels are divided into individual words instead of sentences. Click **Run** to interface with the speech-to-text API and create a new region of interest (ROI) label. The ROI label contains words detected and labeled by IBM's speech-to-text API.

# 1 Apps in Audio Toolbox



Close the **Speech to Text** tab and then export the labeled signal set to the workspace.



The labels are exported to the workspace as `labeledSignalSet` object with a time stamp. Set the variable `labeledSet` to the time-stamped `labeledSignalSet` object.

```
labeledSet = myLabeledSet;
```

Inspect the `SpeechContent` label.

```
speechContent = labeledSet.Labels.SpeechContent{1}
```

```
speechContent=52x2 table
    ROIlimits      Value
    _____    _____
    0.87    1.31    "first"
    1.31    1.41    "you"
    1.41    1.63    "said"
```

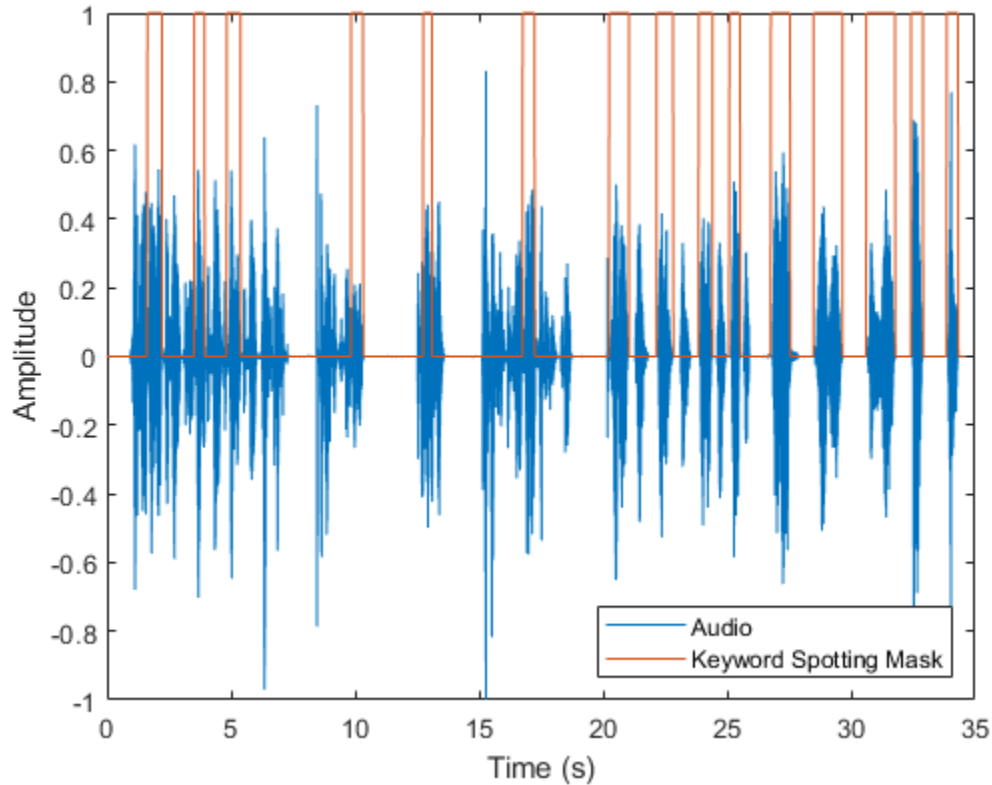
```
1.63  2.22  "yes"  
2.25  2.52  "then"  
2.52  3.03  "no"  
3.09  3.22  "and"  
3.22  3.32  "you"  
3.32  3.52  "said"  
3.52  3.94  "yes"  
3.94  4.16  "then"  
4.16  4.66  "no"  
4.83  5.39  "yes"  
5.42  5.57  "the"  
5.57  6.07  "no"  
6.15  6.56  "driving"  
:
```

The speech-to-text API returns the limits of the ROI labels in seconds. Use the `SpeechContent` table to create a logical vector.

```
keywordLabels = speechContent(speechContent.Value == "yes",:);  
keywordROIlimitsInSamples = round(keywordLabels.ROIlimits*fs);  
  
mask = zeros(size(audioIn),"logical");  
for i = 1:size(keywordROIlimitsInSamples)  
    mask(keywordROIlimitsInSamples(i,1):keywordROIlimitsInSamples(i,2)) = true;  
end
```

Plot the speech signal and the keyword spotting mask.

```
plot(t, audioIn, ...  
     t, mask)  
xlabel('Time (s)')  
ylabel('Amplitude')  
legend('Audio', 'Keyword Spotting Mask', 'Location', 'southeast')
```



- “Label Audio Using Audio Labeler”

## Programmatic Use

audioLabeler opens the app, enabling you to label ground-truth data about audio.

## See Also

[audioDatastore](#) | [audioDeviceReader](#) | [audioDeviceWriter](#) | [labeledSignalSet](#) | [signalLabelDefinition](#)

## **Topics**

“Label Audio Using Audio Labeler”

**Introduced in R2018b**

# Impulse Response Measurer

Measure impulse response of audio system

## Description

The **Impulse Response Measurer** app enables you to acquire, analyze, and export impulse response and frequency response measurements through a user interface.

Using this app, you can:

- Acquire impulse responses to create filters and generate models for offline simulations.
- Determine whether audio devices (loudspeakers, for example) meet time and frequency specifications.
- Optimize audio systems, such as automotive-acoustic systems, to match goal specifications.
- Acquire accurate impulse response measurements for use in acoustic reporting.

## Open the Impulse Response Measurer App


MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.

MATLAB Command prompt: Enter `impulseResponseMeasurer`.

## Examples

### Verify Input/Output Configuration

For large systems with multiple audio devices and multiple input and output channels, tracking how reported devices and channels correspond to physical devices can be difficult. The **Impulse Response Measurer** provides a level monitor so that you can verify your audio I/O configuration.

To open the level monitor, click **Level Monitor**, .



Choose a player and recorder channel, the test signal, and the output level. Verify that the level reported by the recorder reacts appropriately to level changes output by the player. Once you are satisfied that your system is configured correctly, close the level monitor and begin the impulse response capture.

- “Impulse Response Measurer Walkthrough”



## Parameters

### Method — Select excitation signal as MLS or swept sine wave

MLS (default) | Exponential Swept Sine

Select the excitation signal algorithm used to generate an impulse response measurement:

- **MLS** -- The maximum length sequence (MLS) technique is based on the excitation of the acoustical space by a periodic pseudorandom signal. The impulse response is obtained by circular cross-correlation between the measured output and the test tone. For more details, see [2].
- **Exponential Swept Sine** -- The swept sine measurement technique uses an exponential time-growing frequency sweep as an output signal. The output signal is recorded, and deconvolution is used to recover the impulse response from the swept sine tone. For more details, see [1]. The swept sine technique enables you to modify additional **Advanced Settings** to control the excitation signal. The advanced settings apply per run:
  - **Sweep start frequency**
  - **Sweep stop frequency**
  - **Sweep duration**
  - **End silence duration**

The value of the **End silence duration** is read-only and depends on the **Sweep duration** and **Duration per Run (s)**: End silence duration = Duration per Run – Sweep duration

## References

- [1] Farino, Angelo. "Advancements in Impulse Response Measurements by Sine Sweeps." Presented at the *Audio Engineering Society 122nd Convention*, Vienna, Austria, 2007.
- [2] Guy-Bart, Stan, Jean-Jacques Embrachts, and Dominique Archambeau. "Comparison of Different Impulse Response Measurement Techniques." *Journal of Audio Engineering Society*. Vol. 50, Issue 4, 2002, pp. 246-262.

[3] Armelloni, Enrico, Christian Giottoli, and Angelo Farina. "Implementation of Real-Time Partitioned Convolution on a DSP Board." *Application of Signal Processing to Audio and Acoustics, 2003 IEEE Workshop*, pp. 71–74. IEEE, 2003.

## **See Also**

audioPlayerRecorder | reverberator | splMeter

## **Topics**

"Impulse Response Measurer Walkthrough"

**Introduced in R2018a**

# Audio Test Bench

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.
- Run validation checks and generate VST plugins.

## 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 test bench runs)	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 and set parameters to their initial values.
	View Source Code	Open the source file of your audio plugin.
	Synchronize to MIDI Controls	Start the <code>configureMIDI</code> user interface (UI) for your plugin object.
	Open the visualizer of the object under test	Call the <code>visualize</code> function of the object under test with no input arguments. If your object under test does not define a <code>visualize</code> function, then the  button does not appear.  See the <code>audiopluginexample.VarSlopeBandpassFilter</code> plugin for an example of how to define the <code>visualize</code> function.
	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.
	Generate VST 2 Audio Plugin	Open a UI to validate and generate your plugin object. For Audio Toolbox System objects, the <b>Audio Test Bench</b> creates an <code>audioPlugin</code> class using the <code>createAudioPluginClass</code> method of the object. The created plugin class is used to generate a plugin object. For more information, see <code>validateAudioPlugin</code> , <code>generateAudioPlugin</code> , and the <code>createAudioPluginClass</code> method of your System object™.
	Generate MATLAB Script	Generate a MATLAB script implementation of your audio test bench.
	Help	Open MATLAB documentation for <b>Audio Test Bench</b> .

Button		Description
	Configure Input	<p>Open the input configuration UI. The UI options depend on your choice of input to the audio stream loop. See the corresponding documentation for your input choice:</p> <ul style="list-style-type: none"> <li>• Audio File Reader -- <code>dsp.AudioFileReader</code></li> <li>• Audio Device Reader -- <code>audioDeviceReader</code></li> <li>• Audio Oscillator -- <code>audioOscillator</code></li> <li>• Wavetable Synthesizer -- <code>wavetableSynthesizer</code></li> <li>• Chirp Signal -- <code>dsp.Chirp</code></li> <li>• Colored Noise -- <code>dsp.ColoredNoise</code></li> </ul>
	Configure Output	<p>Open the 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>.</p>

## Open the Audio Test Bench App

MATLAB command prompt: Enter `audioTestBench`.

## Examples

- “Audio Test Bench Walkthrough”

## Programmatic Use

`audioTestBench pluginClass` opens the **Audio Test Bench** for an instance of `pluginClass`. The input to `audioTestBench` must derive from the `audioPlugin` class, not the `audioPluginSource` class.

`audioTestBench(pluginClassInstance)` opens the **Audio Test Bench** for `pluginClassInstance`, where `pluginClassInstance` is an instance of an audio plugin class. The input to `audioTestBench` must derive from the `audioPlugin` class, not the `audioPluginSource` class.

`audioTestBench ASTSystemObject` opens the **Audio Test Bench** for an instance of a compatible Audio Toolbox System object.

`audioTestBench(ASTSystemObjectInstance)` opens the **Audio Test Bench** for `ASTSystemObjectInstance`, where `ASTSystemObjectInstance` is an instance of a compatible Audio Toolbox System object.

`audioTestBench(hostedPlugin)` opens the **Audio Test Bench** for `hostedPlugin`, where `hostedPlugin` is an object returned by the `loadAudioPlugin` function.

`audioTestBench(pluginPath)` opens the **Audio Test Bench** for `pluginPath`, where `pluginPath` is the location of an external plugin. Use the full path to specify the audio plugin you want to host. If the plugin is located in the current folder, specify it by its name.

## Tips

- The **Audio Test Bench** provides persistent input and output settings across sessions.

## See Also

### Functions

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

### Classes

`audioPlugin` | `audioPluginSource`

## Topics

“Audio Test Bench Walkthrough”

“What Are DAWs, Audio Plugins, and MIDI Controllers?”

“Audio Plugins in MATLAB”

“Audio Plugin Example Gallery”

**Introduced in R2016a**





# Functions in Audio Toolbox

---

# showaudioblockdatatypetable

Simulink block data type support table

## Syntax

```
showaudioblockdatatypetable
```

## Description

`showaudioblockdatatypetable` shows a table of characteristics for Audio Toolbox blocks. The table lists capabilities and limitations about blocks, such as support for code generation and variable-sized input.

## Examples

### Show Block Characteristics for Audio Toolbox™

Show a table of Audio Toolbox™ block characteristics. The table opens in a separate window.

```
showaudioblockdatatypetable
```

```
Loading Audio Toolbox Library.
```

## See Also

### Topics

“Real-Time Audio in Simulink”

**Introduced in R2016a**

# audioPluginGridLayout

Specify layout for audio plugin UI

## Syntax

```
gridLayout = audioPluginGridLayout  
gridLayout = audioPluginGridLayout(Name,Value)
```

## Description

`gridLayout = audioPluginGridLayout` creates an object that specifies the layout grid for an audio plugin graphical user interface. Use the plugin grid layout object, `gridLayout`, as an argument to `audioPluginInterface` in your plugin class definition. `audioPluginGridLayout` specifies only the grid. The placement of individual graphical elements is specified using `audioPluginParameter`.

To learn how to design a graphic user interface, see “Design User Interface for Audio Plugin”.

For example plugins, see “Audio Plugin Example Gallery”.

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

## Examples

### Use Default Audio Plugin Grid Layout

The default audio plugin grid layout specifies a 2-by-2 grid. Call `audioPluginGridLayout` with no arguments to view the default settings.

```
audioPluginGridLayout
```

```
ans =
```

audioPluginGridLayout with properties:

```
    RowHeight: [100 100]
    ColumnWidth: [100 100]
    RowSpacing: 10
    ColumnSpacing: 10
    Padding: [10 10 10 10]
```

noisifyClassic uses a default grid layout by passing audioPluginGridLayout, without any arguments, to audioPluginInterface. When you use audioPluginGridLayout, you must specify the position of each audioPluginParameter on the grid using Layout. Display names corresponding to parameters occupy cells on the grid also. The default grid contains only four cells and noisifyClassic has four parameters, so you must set DisplayNameLocation to none to fit all elements on the grid. audioPluginGridLayout is passed to the audioPluginInterface. Save noisifyClassic to your current folder.

```
classdef noisifyClassic < audioPlugin
    properties
        DropoutLeft = false
        DropoutRight = false
        NoiseLeftGain = 0
        NoiseRightGain = 0
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('DropoutLeft', ...
                'Layout',[2,1], ...
                'DisplayNameLocation','none'), ...
            audioPluginParameter('DropoutRight', ...
                'Layout',[2,2], ...
                'DisplayNameLocation','none'), ...
            audioPluginParameter('NoiseLeftGain', ...
                'Layout',[1,1], ...
                'DisplayNameLocation','none'), ...
            audioPluginParameter('NoiseRightGain', ...
                'Layout',[1,2], ...
                'DisplayNameLocation','none'), ...
            ...
            audioPluginGridLayout)
    end
```

```
methods
function out = process(plugin,in)
    r = size(in,1);
    dropRate = 0.1;

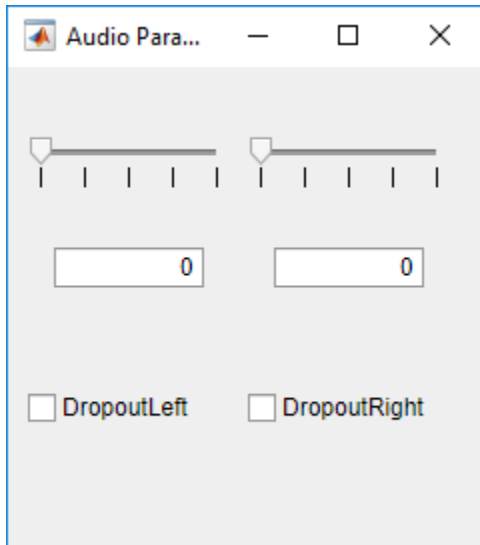
    if plugin.DropoutLeft
        idx = randperm(r,round(r*dropRate));
        in(idx,1) = 0;
    end
    if plugin.DropoutRight
        idx = randperm(r,round(r*dropRate));
        in(idx,2) = 0;
    end

    in(:,1) = in(:,1) + plugin.NoiseLeftGain*(2*rand(r,1,'like',in)-1);
    in(:,2) = in(:,2) + plugin.NoiseRightGain*(2*rand(r,1,'like',in)-1);

    out = in;
end
end
end
```

You can quickly iterate on your UI design by using `parameterTuner` to visualize the plugin UI. Call `parameterTuner` on `noisifyClassic`.

```
parameterTuner(noisifyClassic)
```



### Design Audio Plugin Grid Layout

The example plugin, `noisify`, adds noise to your audio signal channel-wise at a specified gain (per channel) and dropout rate.

```
classdef noisifyOriginal < audioPlugin
    properties
        DropoutLeft = false;
        DropoutRight = false;
        NoiseLeftGain = 0;
        NoiseRightGain = 0;
        DropoutRate = 0.1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('DropoutLeft'), ...
            audioPluginParameter('DropoutRight'), ...
            audioPluginParameter('NoiseLeftGain'), ...
            audioPluginParameter('NoiseRightGain'), ...
            audioPluginParameter('DropoutRate'))
    end
end
```

```
methods
function out = process(plugin,in)
    r = size(in,1);

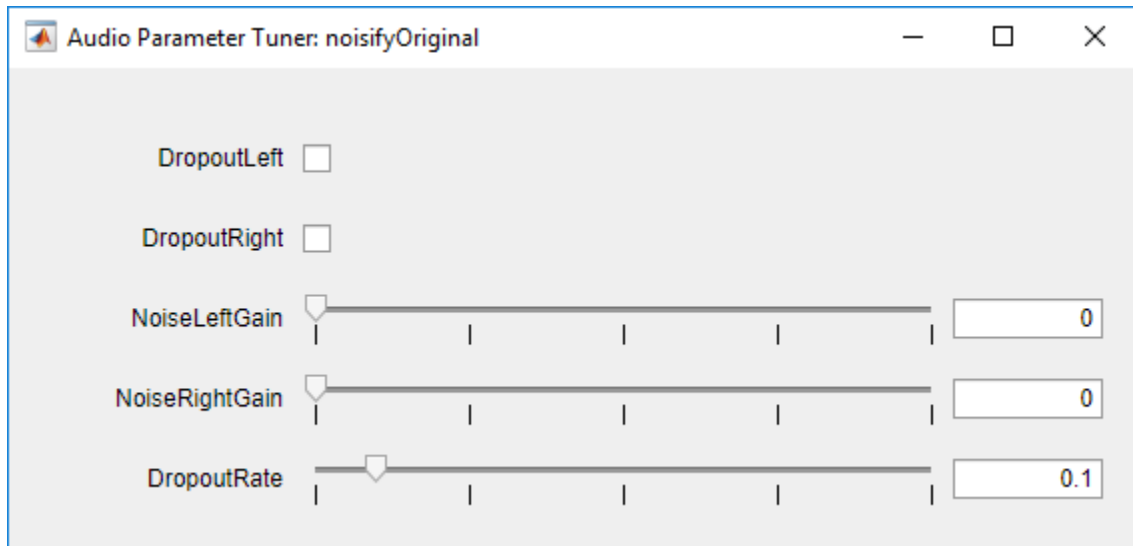
    if plugin.DropoutLeft
        idx = randperm(r,round(r*plugin.DropoutRate));
        in(idx,1) = 0;
    end
    if plugin.DropoutRight
        idx = randperm(r,round(r*plugin.DropoutRate));
        in(idx,2) = 0;
    end

    in(:,1) = in(:,1) + plugin.NoiseLeftGain*randn(r,1,'like',in);
    in(:,2) = in(:,2) + plugin.NoiseRightGain*randn(r,1,'like',in);

    out = in;
end
end
end
```

To see the corresponding UI for the plugin, call `parameterTuner` with the plugin. When you generate an audio plugin and deploy it to a DAW, the DAW uses a default UI that is similar to the default UI of `parameterTuner`.

```
parameterTuner(noisifyOriginal)
```



You can create a more intuitive and visually pleasing UI using `audioPluginInterface`, `audioPluginGridLayout`, and `audioPluginParameter`. For example, to create a more intuitive UI for `noisyOriginal`, you could update the `audioPluginInterface` as follows:

```
classdef noisy < audioPlugin
    properties
        DropoutLeft = false;
        DropoutRight = false;
        NoiseLeftGain = 0;
        NoiseRightGain = 0;
        DropoutRate = 0.1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('DropoutLeft', ...
                'Layout',[4,1], ...
                'DisplayName','Dropout (L)', ...
                'DisplayNameLocation','above', ...
                'Style','vrocker'), ...
            audioPluginParameter('DropoutRight', ...
                'Layout',[4,4], ...
                'DisplayName','Dropout (R)', ...
```



```

        'DisplayNameLocation','above', ...
        'Style','vrockert'), ...
    audioPluginParameter('NoiseLeftGain', ...
        'DisplayName','Noise Gain (L)', ...
        'Layout',[2,1;2,2], ...
        'DisplayNameLocation','above', ...
        'Style','rotaryknob'), ...
    audioPluginParameter('NoiseRightGain', ...
        'Layout',[2,3;2,4], ...
        'DisplayName','Noise Gain (R)', ...
        'DisplayNameLocation','above', ...
        'Style','rotaryknob'), ...
    audioPluginParameter('DropoutRate', ...
        'Layout',[4,2;4,3], ...
        'DisplayName','Dropout Rate', ...
        'DisplayNameLocation','below', ...
        'Style','vslider'), ...
    ...
    audioPluginGridLayout( ...
        'RowHeight',[15,150,15,150,15], ...
        'ColumnWidth',[100,40,40,100], ...
        'RowSpacing',30))
end
methods
function out = process(plugin,in)
    r = size(in,1);

    if plugin.DropoutLeft
        idx = randperm(r,round(r*plugin.DropoutRate));
        in(idx,1) = 0;
    end
    if plugin.DropoutRight
        idx = randperm(r,round(r*plugin.DropoutRate));
        in(idx,2) = 0;
    end

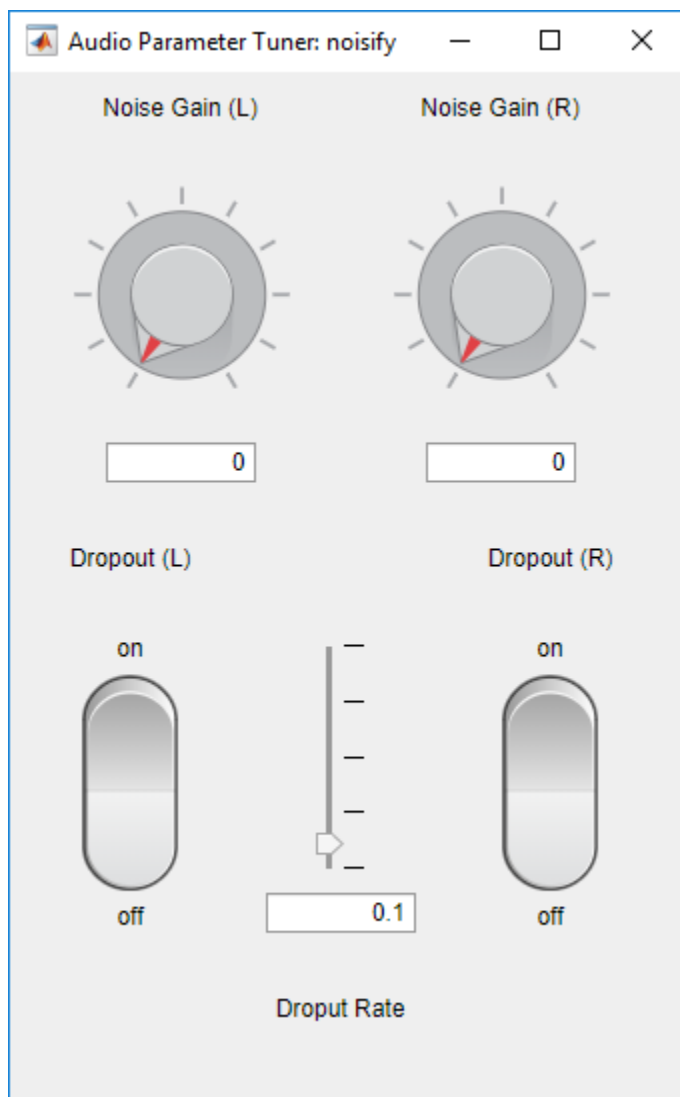
    in(:,1) = in(:,1) + plugin.NoiseLeftGain*randn(r,1,'like',in);
    in(:,2) = in(:,2) + plugin.NoiseRightGain*randn(r,1,'like',in);

    out = in;
end
end
end

```

You can quickly iterate on your UI design by using `parameterTuner` to visualize incremental changes. Call `parameterTuner` on `noisify`. When you generate an audio plugin and deploy it to a DAW, the DAW uses the enhanced UI.

```
parameterTuner(noisify)
```



## Input Arguments

### 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'RowHeight', [50,200,150]` species a grid with three rows. The first row is 50 pixels high, the second row is 200 pixels high, and the third row is 150 pixels high.

### **RowHeight — Height of each row (pixels)**

`[100, 100]` (default) | row vector of positive integers

Height in pixels of each row in the grid, specified as a comma-separated pair consisting of `'RowHeight'` and a row vector of positive integers.

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

### **ColumnWidth — Width of each column (pixels)**

`[100, 100]` (default) | row vector of positive integers

Width in pixels of each column in the grid, specified as a comma-separated pair consisting of `'ColumnWidth'` and a row vector of positive integers.

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

### **RowSpacing — Distance between rows (pixels)**

`10` (default) | nonnegative integer

Distance between rows in pixels, specified as a comma-separated pair consisting of `'RowSpacing'` and a nonnegative integer.

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

### **ColumnSpacing — Distance between columns (pixels)**

`10` (default) | nonnegative integer

Distance between columns in pixels, specified as a comma-separated pair consisting of `'ColumnSpacing'` and a nonnegative integer.

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

### **Padding — Padding around the outer perimeter of grid (pixels)**

`[10, 10, 10, 10]` (default) | `[left, bottom, right, top]`

Padding around the outer perimeter of the grid in pixels, specified as a comma-separated pair consisting of 'Padding' and a four-element row vector of nonnegative integers. The elements of the vector are interpreted as [left, bottom, right, top], where:

- left -- Distance in pixels from the left edge of the grid to the left edge of the parent container.
- bottom -- Distance in pixels from the bottom edge of the grid to the bottom edge of the parent container.
- right -- Distance in pixels from the right edge of the grid to the right edge of the parent container.
- top -- Distance in pixels from the top edge of the grid to the top edge of the parent container.

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

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`Audio Test Bench` | `audioPlugin` | `audioPluginInterface` | `audioPluginParameter` | `audioPluginSource` | `generateAudioPlugin` | `parameterTuner` | `validateAudioPlugin`

### Topics

“Design User Interface for Audio Plugin”

“Audio Plugin Example Gallery”

“Audio Plugins in MATLAB”

“Export a MATLAB Plugin to a DAW”

**Introduced in R2019b**

## **pinknoise**

Generate pink noise

### **Syntax**

```
X = pinknoise(n)
X = pinknoise(sz1,sz2)
X = pinknoise(sz)
X = pinknoise( ____, typename)
X = pinknoise( ____, 'like', p)
```

### **Description**

`X = pinknoise(n)` returns a pink noise column vector of length `n`.

`X = pinknoise(sz1,sz2)` returns a `sz1`-by-`sz2` matrix. Each channel (column) of the output `X` is an independent pink noise signal.

`X = pinknoise(sz)` returns a vector or matrix with dimensions defined by the elements of vector `sz`. `sz` must be a one- or two-element row vector of positive integers. Each channel (column) of the output `X` is an independent pink noise signal.

`X = pinknoise( ____, typename)` returns an array of pink noise of data type `typename`. The `typename` input can be either `'single'` or `'double'`. You can combine `typename` with any of the input arguments in the previous syntaxes.

`X = pinknoise( ____, 'like', p)` returns an array of pink noise like `p`. You can specify either `typename` or `'like'`, but not both.

### **Examples**

#### **Generate Pink Noise**

Generate 100 seconds of pink noise with a sample rate of 44.1 kHz.

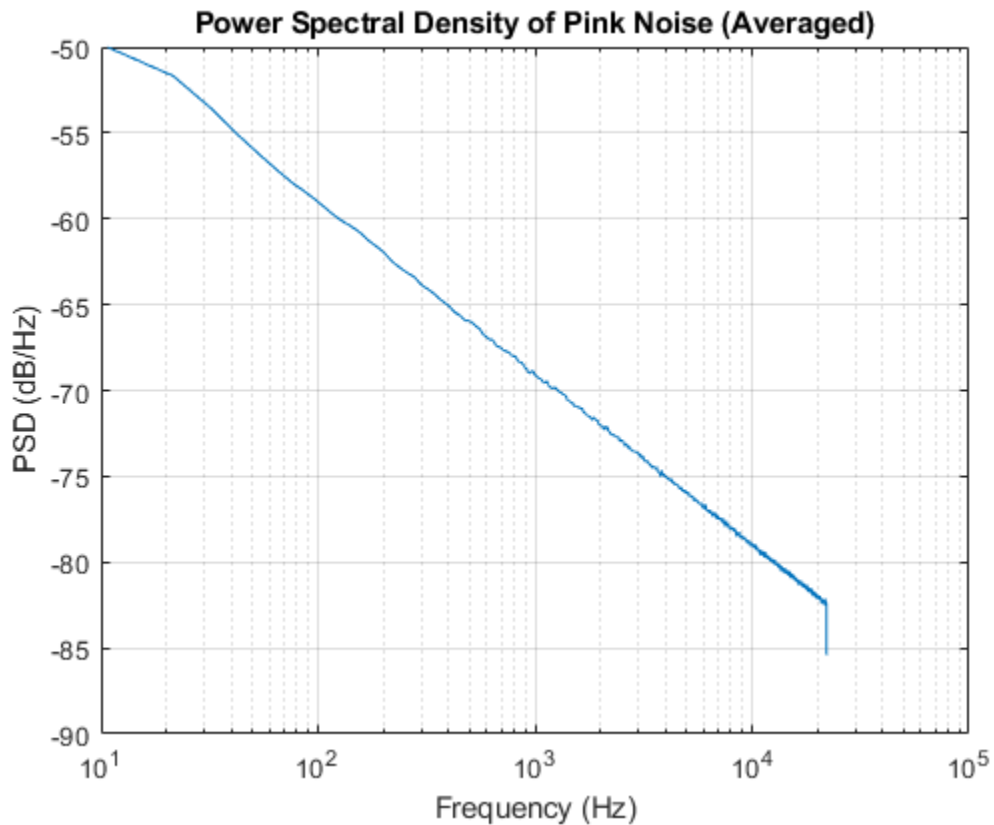
```
fs = 44.1e3;  
duration = 100;
```

```
y = pinknoise(duration*fs);
```

Plot the average power spectral density (PSD) of the generated pink noise.

```
[~,freqVec,~,psd] = spectrogram(y,round(0.05*fs),[],[],fs);  
meanPSD = mean(psd,2);
```

```
semilogx(freqVec,db(meanPSD,"power"))  
xlabel('Frequency (Hz)')  
ylabel('PSD (dB/Hz)')  
title('Power Spectral Density of Pink Noise (Averaged)')  
grid on
```



### Amplitude Distribution of Pink Noise

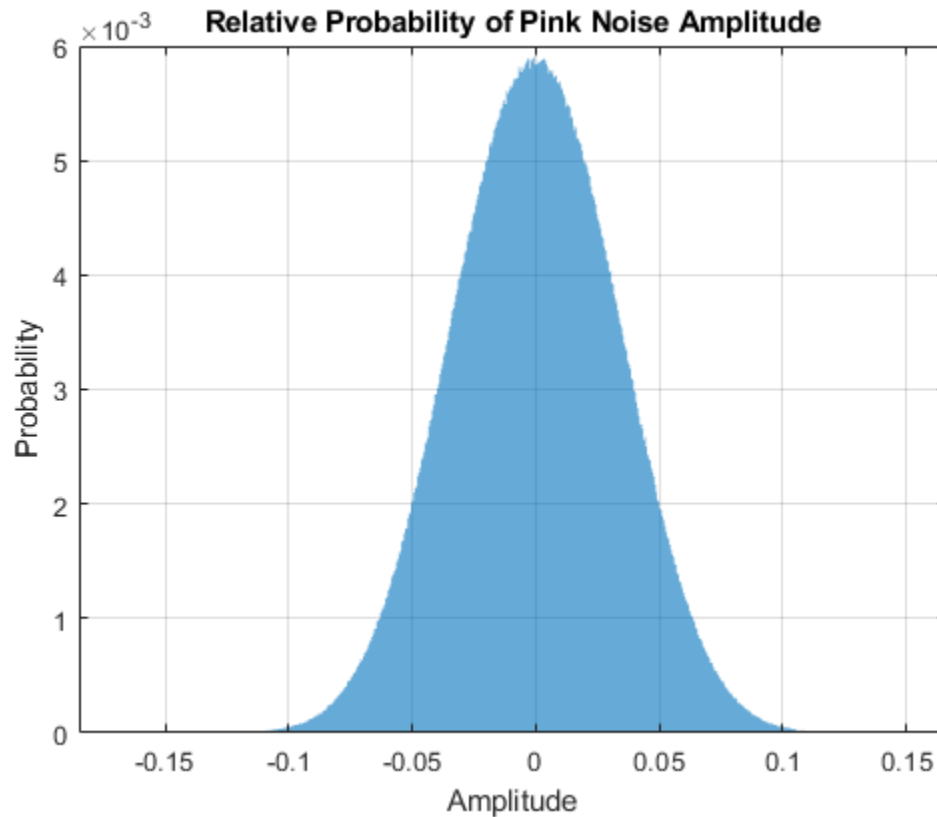
Generate 500 seconds of pink noise with a sample rate of 16 kHz.

```
fs = 16e3;  
duration = 500;  
  
y = pinknoise(duration*fs);
```

Plot the relative probability of the pink noise amplitude. The amplitude is always bounded between  $-1$  and  $1$ .



```
histogram(y,"Normalization","probability","EdgeColor","none")  
xlabel("Amplitude")  
ylabel("Probability")  
title("Relative Probability of Pink Noise Amplitude")  
grid on
```



### Generate Multiple Independent Channels of Pink Noise

Create a 5 second stereo pink noise signal with a 48 kHz sample rate.

```
fs = 48e3;  
duration = 5;
```

```
numChan = 2;  
pn = pinknoise(duration*fs,numChan);
```

Listen to the stereo pink noise signal.

```
sound(pn, fs)
```

Channels of the pink noise function are generated independently. Note that the off-diagonal correlation coefficients are close to zero (uncorrelated).

```
R = corrcoef(pn(:,1),pn(:,2))
```

```
R = 2×2
```

```
    1.0000    -0.0190  
   -0.0190    1.0000
```

Correlated and uncorrelated pink noise have different psychoacoustic effects. When the noise is correlated, the sound is less ambient and more centralized. To listen to correlated pink noise, send a single channel of the pink noise signal to your stereo device. The effect is most pronounced when using headphones.

```
sound([pn(:,1),pn(:,1)], fs)
```

### Add Pink Noise to Audio Signal

Read in an audio file.

```
[audioIn,fs] = audioread("MainStreet0ne-24-96-stereo-63secs.wav");
```

Create a pink noise signal of the same size and data type as `audioIn`.

```
noise = pinknoise(size(audioIn),'like',audioIn);
```

Add the pink noise to the audio signal and then listen to the first 5 seconds.

```
noisyMainStreet = noise + audioIn;  
sound(noisyMainStreet(1:fs*5,:), fs)
```

The `pinknoise` function generates an approximate  $-29.5$  dB signal level, which is close to the power of the audio signal.

```
noisePower = sum(noise.^2,1)/size(noise,1);  
signalPower = sum(audioIn.^2,1)/size(audioIn,1);  
snr = 10*log10(signalPower./noisePower)
```

```
snr = 1x2  
    -0.3505    -1.6718
```

```
noisePowerdB = 10*log10(noisePower)
```

```
noisePowerdB = 1x2  
    -29.6072    -29.5546
```

```
signalPowerdB = 10*log10(signalPower)
```

```
signalPowerdB = 1x2  
    -29.9577    -31.2264
```

Mix the input audio with the generated pink noise at an 8 dB SNR.

```
desiredSNR = 8;  
scaleFactor = sqrt(signalPower./(noisePower*(10^(desiredSNR/10))));  
noise = noise.*scaleFactor;
```

Verify the resulting SNR is 8 dB and then listen to the first 5 seconds.

```
noisePower = sum(noise.^2,1)/size(noise,1);  
snr = 10*log10(signalPower./noisePower)
```

```
snr = 1x2  
    8.0000    8.0000
```

```
noisyMainStreet = noise + audioIn;  
sound(noisyMainStreet(1:fs*5,:), fs)
```

## Input Arguments

### **n** — Number of rows of pink noise

nonnegative integer

Number of rows of pink noise, specified as a nonnegative integer.

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

### **sz1, sz2** — Size of each dimension (as separate arguments)

nonnegative integers

Size of each dimension, specified as a nonnegative integer or two separate arguments of nonnegative integers.

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

### **sz** — Size of each dimension (as a row vector)

one- or two-element row vector of nonnegative integers

Size of each dimension, specified as a one- or two-element row vector of nonnegative integers. Each element of this vector indicates the size of the corresponding dimension.

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

### **typename** — Data type to create

'double' (default) | 'single'

Data type to create, specified as 'double' or 'single'.

Data Types: `char` | `string`

### **p** — Prototype of array to create

numeric array

Prototype of array to create, specified as a numeric array. The generated pink noise is the same data type as `p`.

Data Types: `single` | `double`

## Output Arguments

### **X** — Pink noise

column vector | matrix

Pink noise, returned as a column vector or matrix of independent channels.

Data Types: `single` | `double`

## Tips

- The concatenation of multiple pink noise vectors does not result in pink noise. For streaming applications, use `dsp.ColoredNoise`.

## Algorithms

Pink noise is generated by passing uniformly distributed random numbers through a series of randomly initiated SOS filters. The resulting pink noise amplitude distribution is quasi-Gaussian and bounded between  $-1$  and  $1$ . The resulting pink noise power spectral density (PSD) is inversely proportional to frequency:

$$S(f) \propto \frac{1}{f}$$

## Extended Capabilities

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

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

### **See Also**

`dsp.ColoredNoise` | `rand` | `rng`

**Introduced in R2019b**

# stretchAudio

Time-stretch audio

## Syntax

```
audioOut = stretchAudio(audioIn,alpha)
audioOut = stretchAudio(audioIn,alpha,Name,Value)
```

## Description

`audioOut = stretchAudio(audioIn,alpha)` applies time scale modification (TSM) on the input audio by the TSM factor `alpha`.

`audioOut = stretchAudio(audioIn,alpha,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

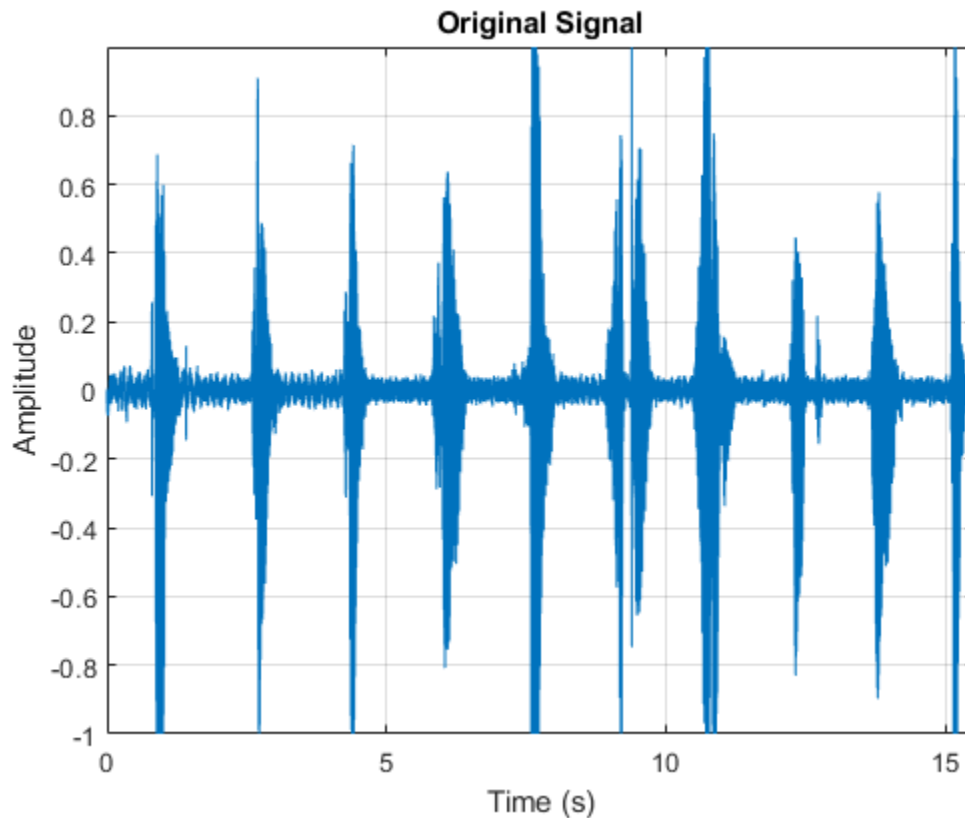
## Examples

### Apply TSM

Read in an audio signal. Listen to the audio signal and plot it over time.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");

t = (0:size(audioIn,1)-1)/fs;
plot(t,audioIn)
xlabel('Time (s)')
ylabel('Amplitude')
title('Original Signal')
axis tight
grid on
```



```
sound(audioIn, fs)
```

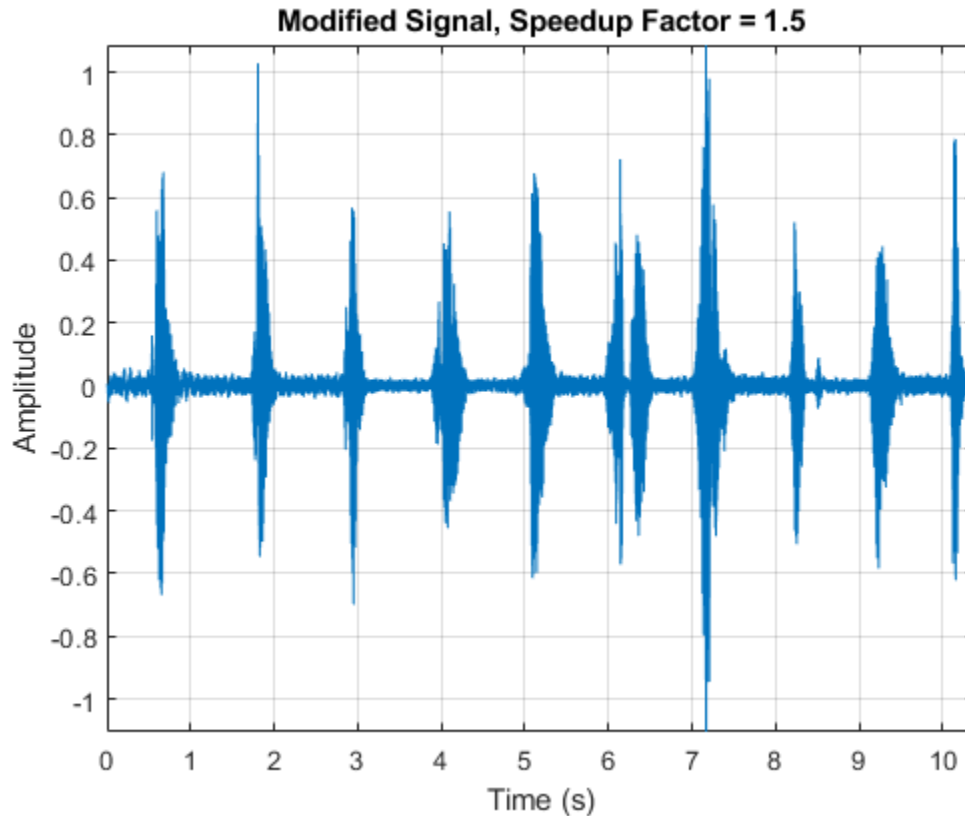
Use `stretchAudio` to apply a 1.5 speedup factor. Listen to the modified audio signal and plot it over time. The sample rate remains the same, but the duration of the signal has decreased.

```
audioOut = stretchAudio(audioIn, 1.5);
```

```
t = (0:size(audioOut,1)-1)/fs;  
plot(t, audioOut)  
xlabel('Time (s)')  
ylabel('Amplitude')  
title('Modified Signal, Speedup Factor = 1.5')
```



```
axis tight  
grid on
```

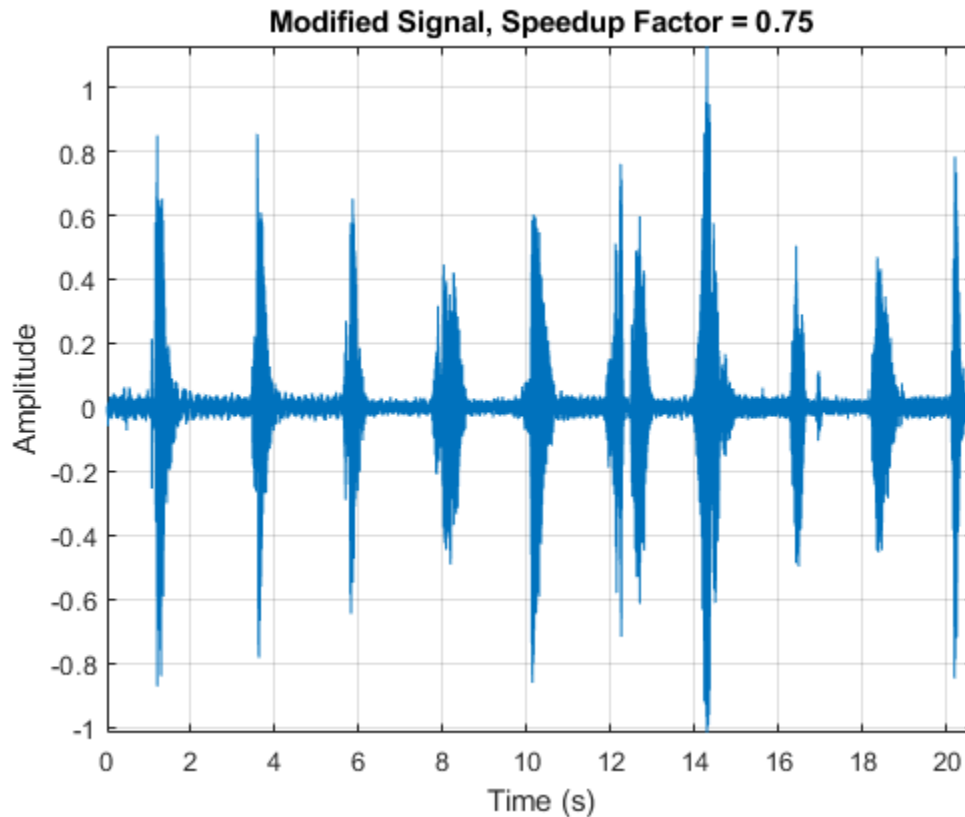


```
sound(audioOut, fs)
```

Slow down the original audio signal by a 0.75 factor. Listen to the modified audio signal and plot it over time. The sample rate remains the same as the original audio, but the duration of the signal has increased.

```
audioOut = stretchAudio(audioIn,0.75);  
  
t = (0:size(audioOut,1)-1)/fs;  
plot(t,audioOut)  
xlabel('Time (s)')
```

```
ylabel('Amplitude')
title('Modified Signal, Speedup Factor = 0.75')
axis tight
grid on
```



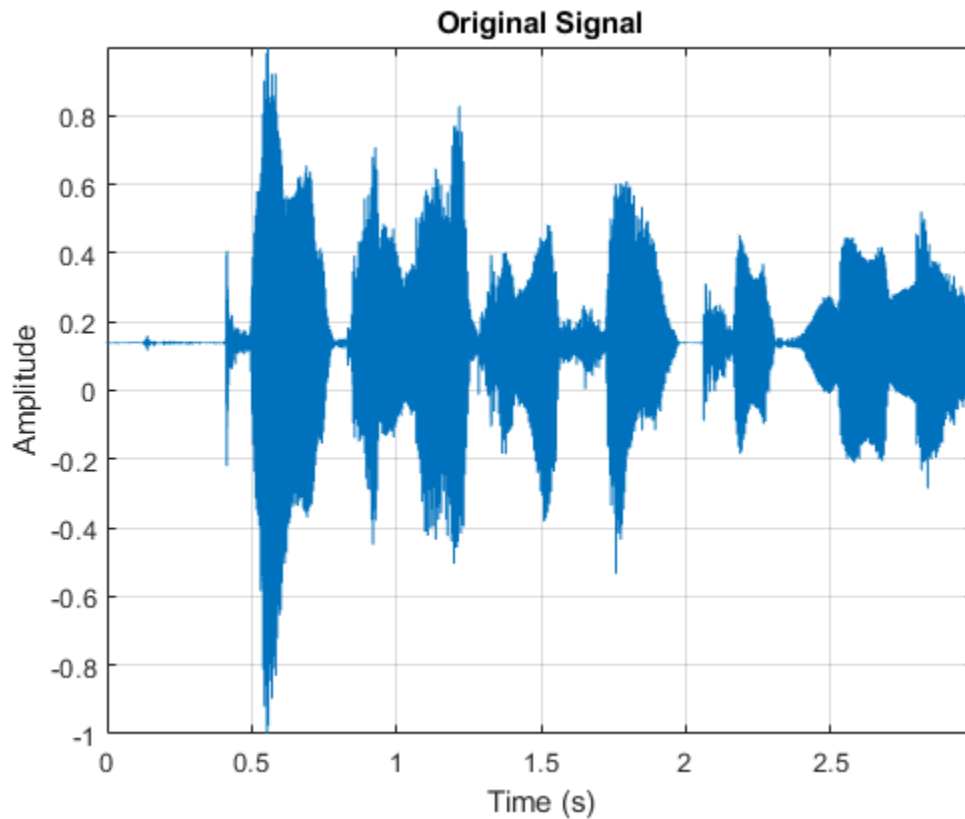
```
sound(audioOut, fs)
```

### Apply TSM to Frequency-Domain Audio

`stretchAudio` supports TSM on frequency-domain audio when using the default vocoder method. Applying TSM to frequency-domain audio enables you to reuse your STFT computation for multiple TSM factors.

Read in an audio signal. Listen to the audio signal and plot it over time.

```
[audioIn,fs] = audioread('FemaleSpeech-16-8-mono-3secs.wav');  
sound(audioIn,fs)  
  
t = (0:size(audioIn,1)-1)/fs;  
plot(t,audioIn)  
xlabel('Time (s)')  
ylabel('Amplitude')  
title('Original Signal')  
axis tight  
grid on
```



Convert the audio signal to the frequency domain.

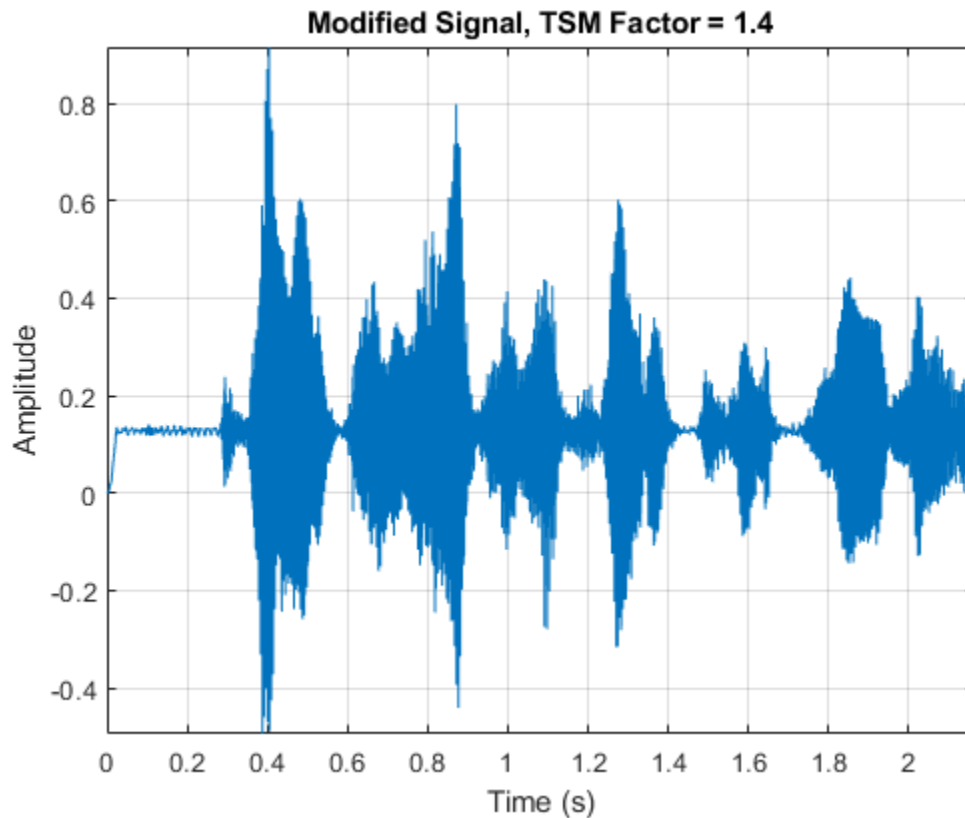
```
win = sqrt(hann(256, 'periodic'));  
ovrlp = 192;  
S = stft(audioIn, 'Window', win, 'OverlapLength', ovrlp, 'Centered', false);
```

Speed up the audio signal by a factor of 1.4. Specify the window and overlap length used to create the frequency-domain representation.

```
alpha = 1.4;  
audioOut = stretchAudio(S, alpha, 'Window', win, 'OverlapLength', ovrlp);
```

```
sound(audioOut, fs)
```

```
t = (0:size(audioOut,1)-1)/fs;  
plot(t, audioOut)  
xlabel('Time (s)')  
ylabel('Amplitude')  
title('Modified Signal, TSM Factor = 1.4')  
axis tight  
grid on
```



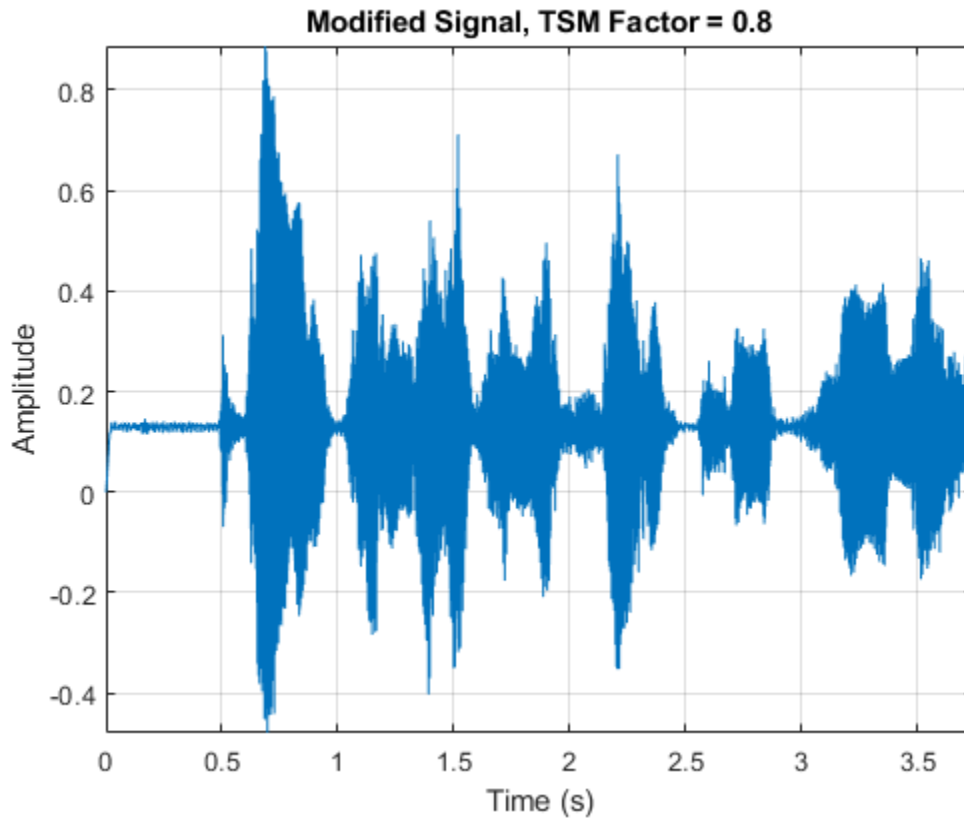
Slow down the audio signal by a factor of 0.8. Specify the window and overlap length used to create the frequency-domain representation.

```
alpha = 0.8;
audioOut = stretchAudio(S,alpha,'Window',win,'OverlapLength',ovrlp);

sound(audioOut,fs)

t = (0:size(audioOut,1)-1)/fs;
plot(t,audioOut)
xlabel('Time (s)')
ylabel('Amplitude')
title('Modified Signal, TSM Factor = 0.8')
```

```
axis tight  
grid on
```



### Increase Fidelity Using Phase-Locking

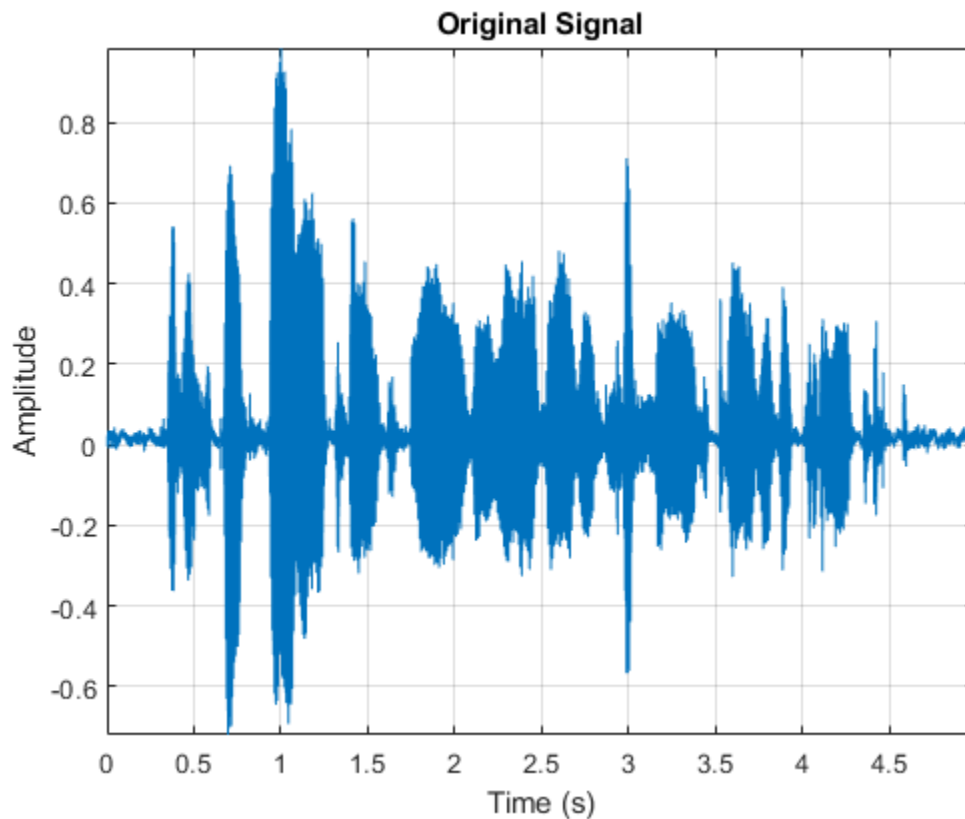
The default TSM method (vocoder) enables you to additionally apply phase-locking to increase the fidelity to the original audio.

Read in an audio signal. Listen to the audio signal and plot it over time.

```
[audioIn,fs] = audioread("SpeechDFT-16-8-mono-5secs.wav");
```

```
sound(audioIn, fs)

t = (0:size(audioIn,1)-1)/fs;
plot(t, audioIn)
xlabel('Time (s)')
ylabel('Amplitude')
title('Original Signal')
axis tight
grid on
```



Phase-locking adds a nontrivial computational load to TSM and is not always required. By default, phase-locking is disabled. Apply a speedup factor of 1.8 to the input audio signal. Listen to the audio signal and plot it over time.

```
alpha = 1.8;

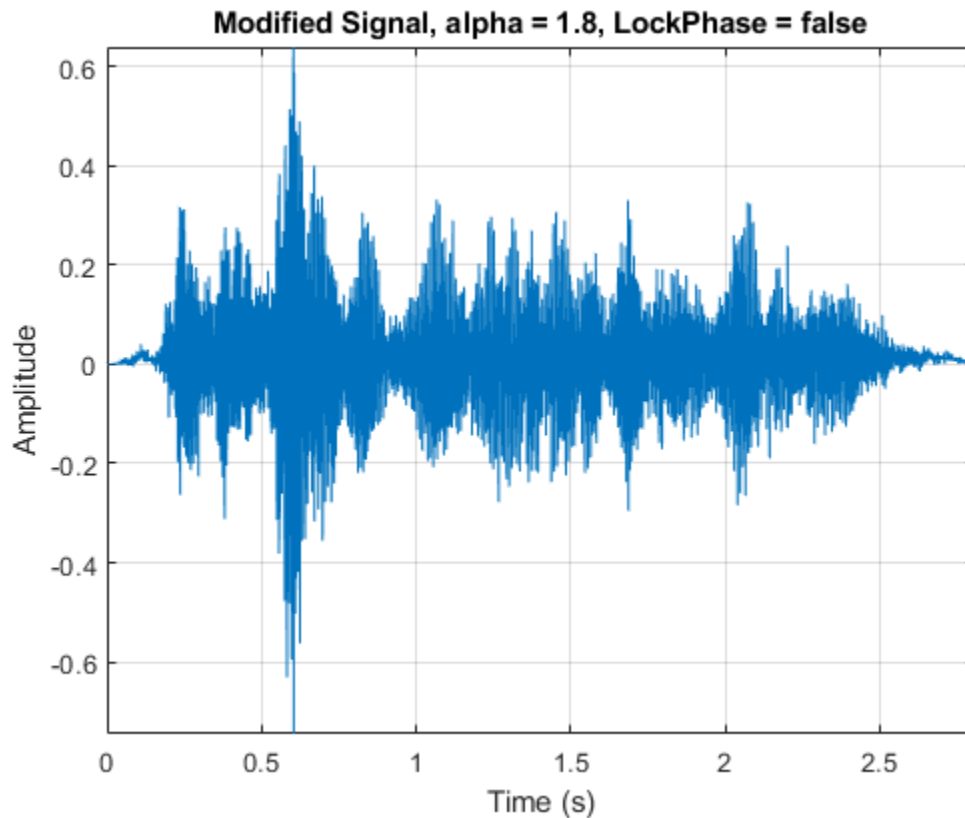
tic
audioOut = stretchAudio(audioIn,alpha);
processingTimeWithoutPhaseLocking = toc

processingTimeWithoutPhaseLocking = 0.0798

sound(audioOut,fs)

t = (0:size(audioOut,1)-1)/fs;
plot(t,audioOut)
xlabel('Time (s)')
ylabel('Amplitude')
title('Modified Signal, alpha = 1.8, LockPhase = false')
axis tight
grid on
```





Apply the same 1.8 speedup factor to the input audio signal, this time enabling phase-locking. Listen to the audio signal and plot it over time.

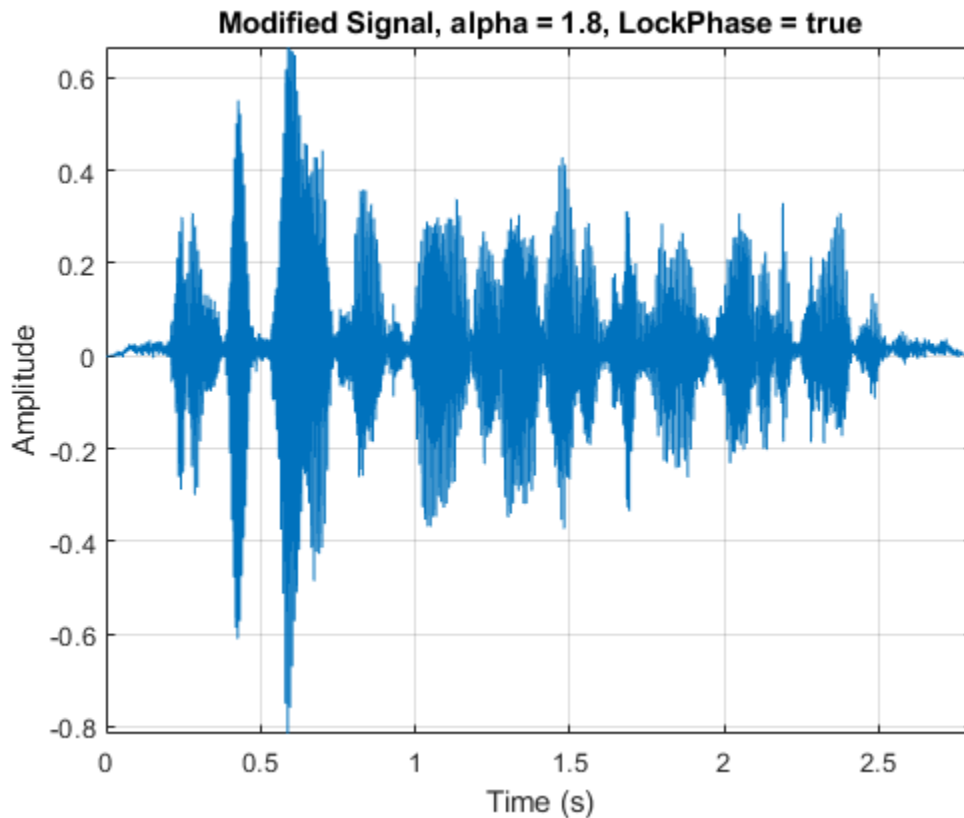
```
tic
audioOut = stretchAudio(audioIn,alpha,"LockPhase",true);
processingTimeWithPhaseLocking = toc

processingTimeWithPhaseLocking = 0.1154

sound(audioOut,fs)

t = (0:size(audioOut,1)-1)/fs;
plot(t,audioOut)
xlabel('Time (s)')
```

```
ylabel('Amplitude')
title('Modified Signal, alpha = 1.8, LockPhase = true')
axis tight
grid on
```



### Increase Fidelity Using WSOLA Delta

The waveform similarity overlap-add (WSOLA) TSM method enables you to specify the maximum number of samples to search for the best signal alignment. By default, WSOLA delta is the number of samples in the analysis window minus the number of samples

overlapped between adjacent analysis windows. Increasing the WSOLA delta increases the computational load but might also increase fidelity.

Read in an audio signal. Listen to the first 10 seconds of the audio signal.

```
[audioIn,fs] = audioread('RockGuitar-16-96-stereo-72secs.flac');
sound(audioIn(1:10*fs,:),fs)
```

Apply a TSM factor of 0.75 to the input audio signal using the WSOLA method. Listen to the first 10 seconds of the resulting audio signal.

```
alpha = 0.75;
tic
audioOut = stretchAudio(audioIn,alpha,"Method","wsola");
processingTimeWithDefaultWSOLADelta = toc

processingTimeWithDefaultWSOLADelta = 19.4403

sound(audioOut(1:10*fs,:),fs)
```

Apply a TSM factor of 0.75 to the input audio signal, this time increasing the WSOLA delta to 1024. Listen to the first 10 seconds of the resulting audio signal.

```
tic
audioOut = stretchAudio(audioIn,alpha,"Method","wsola","WSOLADelta",1024);
processingTimeWithIncreasedWSOLADelta = toc

processingTimeWithIncreasedWSOLADelta = 25.5306

sound(audioOut(1:10*fs,:),fs)
```

## Input Arguments

### **audioIn** — Input signal

column vector | matrix | 3-D array

Input signal, specified as a column vector, matrix, or 3-D array. How the function interprets `audioIn` depends on the complexity of `audioIn` and the value of `Method`:

- If `audioIn` is real, `audioIn` is interpreted as a time-domain signal. In this case, `audioIn` must be a column vector or matrix. Columns are interpreted as individual channels.

This syntax applies when `Method` is set to `'vocoder'` or `'wsola'`.

- If `audioIn` is complex, `audioIn` is interpreted as a frequency-domain signal. In this case, `audioIn` must be an  $L$ -by- $M$ -by- $N$  array, where  $L$  is the FFT length,  $M$  is the number of individual spectrums, and  $N$  is the number of channels.

This syntax only applies when `Method` is set to `'vocoder'`.

Data Types: `single` | `double`

Complex Number Support: Yes

### **`alpha` — TSM factor**

positive scalar

TSM factor, specified as a positive scalar.

Data Types: `single` | `double`

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Window', kbdwin(512)`

### **Method — Method used to time-scale audio**

`'vocoder'` (default) | `'wsola'`

Method used to time-scale audio, specified as the comma-separated pair consisting of `'Method'` and `'vocoder'` or `'wsola'`. Set `'Method'` to `'vocoder'` to use the phase vocoder method. Set `'Method'` to `'wsola'` to use the WSOLA method.

If `'Method'` is set to `'vocoder'`, `audioIn` can be real or complex. If `'Method'` is set to `'wsola'`, `audioIn` must be real.

Data Types: `single` | `double`

### **Window — Window applied in time domain**

`sqrt(hann(1024, 'periodic'))` (default) | real vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range

[1, size(audioIn,1)]. The number of elements in the vector must also be greater than OverlapLength.

---

**Note** If using stretchAudio with frequency-domain input, you must specify Window as the same window used to transform audioIn to the frequency domain.

---

Data Types: single | double

**OverlapLength — Number of samples overlapped between adjacent windows**

round(0.75\*numel(Window)) (default) | scalar in the range [0 numel(Window))

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of 'OverlapLength' and an integer in the range [0, numel(Window)).

---

**Note** If using stretchAudio with frequency-domain input, you must specify OverlapLength as the same overlap length used to transform audioIn to a time-frequency representation.

---

Data Types: single | double

**LockPhase — Apply identity phase-locking**

false (default) | true

Apply identity phase-locking, specified as the comma-separated pair consisting of 'LockPhase' and false or true.

**Dependencies**

To enable this name-value pair argument, set Method to 'vocoder'.

Data Types: logical

**WSOLADelta — Maximum samples used to search for best signal alignment**

numel(Window) - OverlapLength (default) | nonnegative scalar

Maximum number of samples used to search for the best signal alignment, specified as the comma-separated pair consisting of 'WSOLADelta' and a nonnegative scalar.

### Dependencies

To enable this name-value pair argument, set `Method` to `'wsola'`.

Data Types: `single` | `double`

## Output Arguments

### **audioOut** — Time-scale modified audio

column vector | matrix

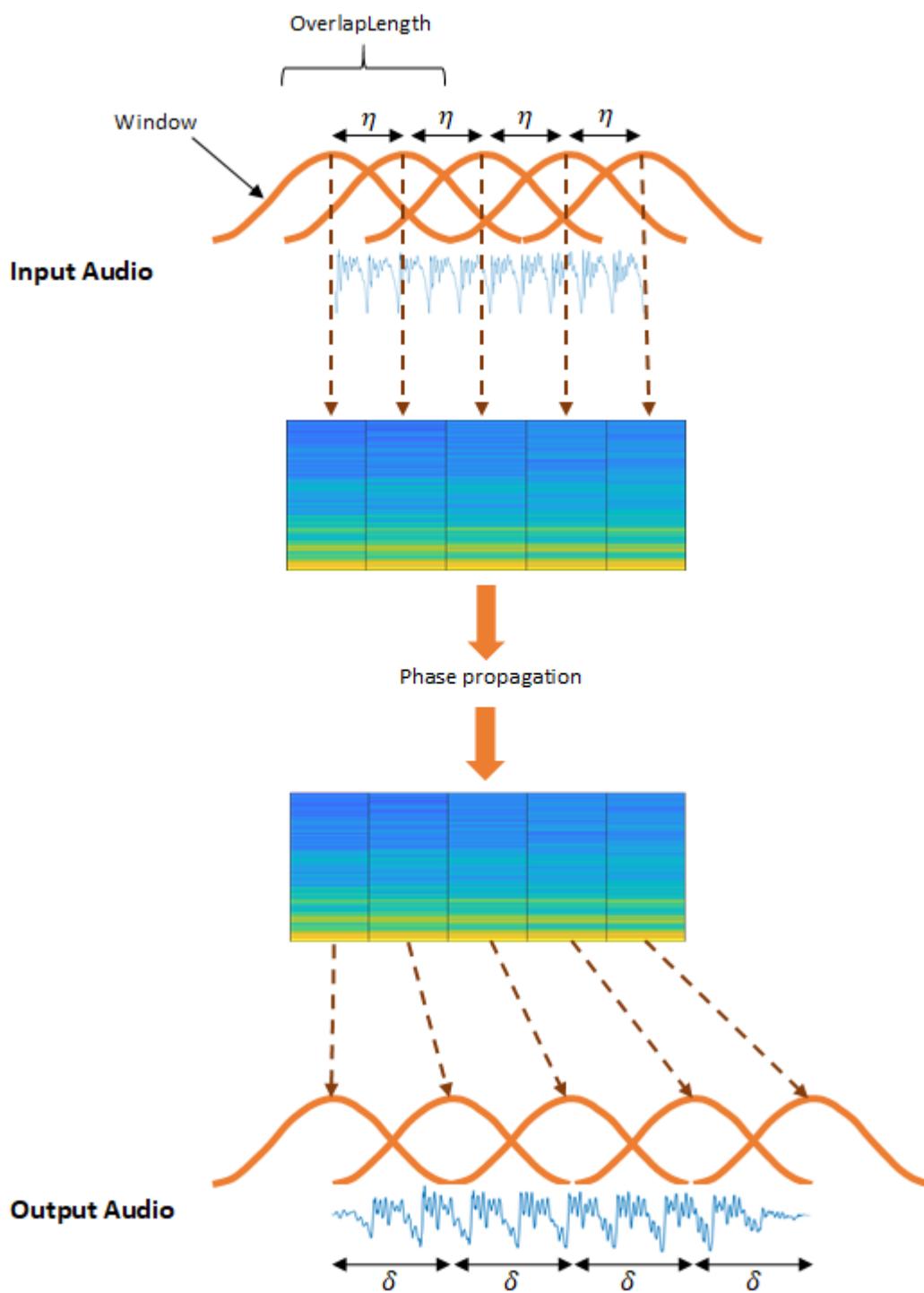
Time-scale modified audio, returned as a column vector or matrix of independent channels.

## Algorithms

### Phase Vocoder

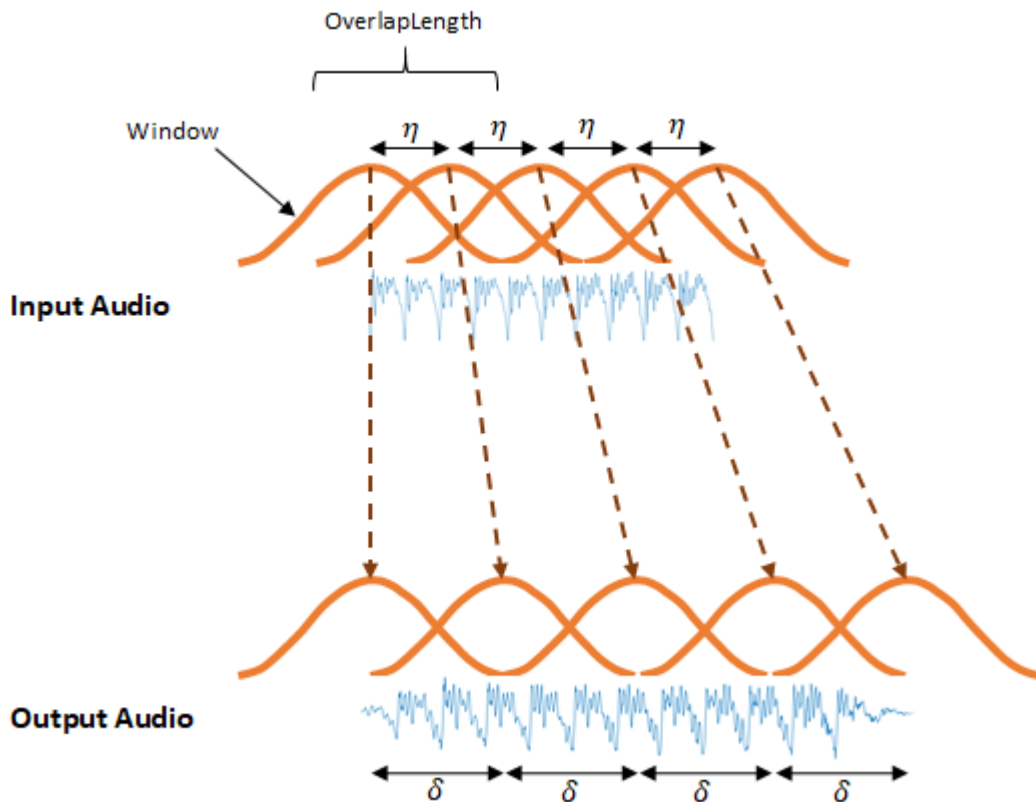
The phase vocoder algorithm is a frequency-domain approach to TSM [1][2]. The basic steps of the phase vocoder algorithm are:

- 1 The algorithm windows a time-domain signal at interval  $\eta$ , where  $\eta = \text{numel}(\text{Window}) - \text{OverlapLength}$ . The windows are then converted to the frequency domain.
- 2 To preserve horizontal (across time) phase coherence, the algorithm treats each bin as an independent sinusoid whose phase is computed by accumulating the estimates of its instantaneous frequency.
- 3 To preserve vertical (across an individual spectrum) phase coherence, the algorithm locks the phase advance of groups of bins to the phase advance of local peaks. This step only applies if `LockPhase` is set to `true`.
- 4 The algorithm returns the modified spectrogram to the time domain, with windows spaced at intervals of  $\delta$ , where  $\delta \approx \eta/\alpha$ .  $\alpha$  is the speedup factor specified by the `alpha` input argument.



## WSOLA

The WSOLA algorithm is a time-domain approach to TSM [1][2]. WSOLA is an extension of the overlap and add (OLA) algorithm. In the OLA algorithm, a time-domain signal is windowed at interval  $\eta$ , where  $\eta = \text{numel}(\text{Window}) - \text{OverlapLength}$ . To construct the time-scale modified output audio, the windows are spaced at interval  $\delta$ , where  $\delta \approx \eta/\alpha$ .  $\alpha$  is the TSM factor specified by the `alpha` input argument.

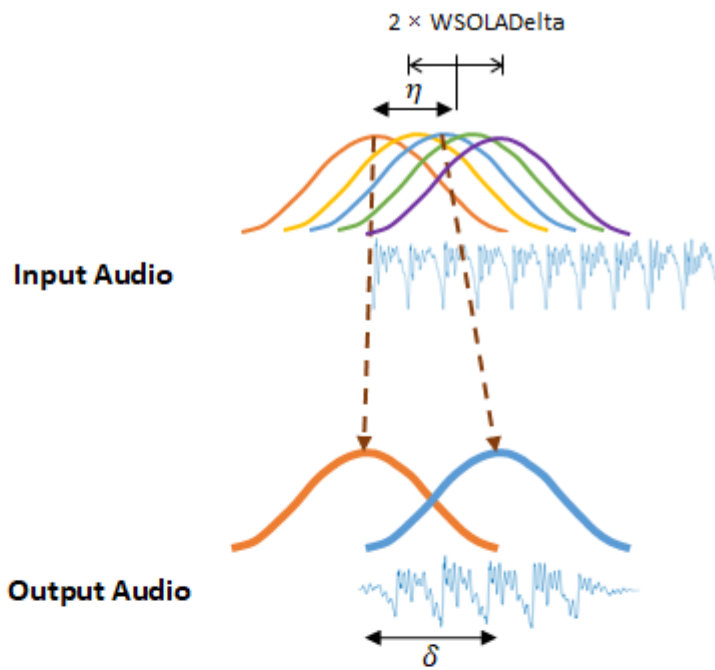


The OLA algorithm does a good job of recreating the magnitude spectra but can introduce phase jumps between windows. The WSOLA algorithm attempts to smooth the phase jumps by searching `WSOLA``Delta` samples around the  $\eta$  interval for a window that

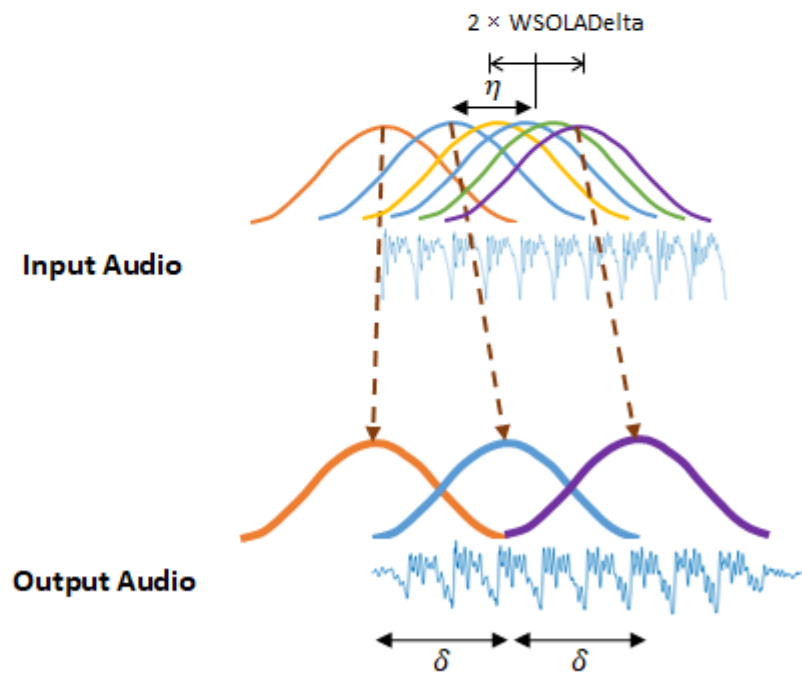


minimizes phase jumps. The algorithm searches for the best window iteratively, so that each successive window is chosen relative to the previously selected window.

Iteration 1



Iteration 2



If `WSOLADelta` is set to  $\theta$ , then the algorithm reduces to OLA.

## References

- [1] Driedger, Johnathan, and Meinard Müller. "A Review of Time-Scale Modification of Music Signals." *Applied Sciences*. Vol. 6, Issue 2, 2016.
- [2] Driedger, Johnathan. "Time-Scale Modification Algorithms for Music Audio Signals", Master's thesis, Saarland University, Saarbrücken, Germany, 2011.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`audioDataAugmenter` | `audioTimeScaler` | `reverberator` | `shiftPitch`

### Topics

"Time-Frequency Masking for Harmonic-Percussive Source Separation"

**Introduced in R2019b**

# shiftPitch

Shift audio pitch

## Syntax

```
audioOut = shiftPitch(audioIn,nsemitones)
audioOut = shiftPitch(audioIn,nsemitones,Name,Value)
```

## Description

`audioOut = shiftPitch(audioIn,nsemitones)` shifts the pitch of the audio input by the specified number of semitones, `nsemitones`.

`audioOut = shiftPitch(audioIn,nsemitones,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

## Examples

### Apply Pitch-Shifting to Time-Domain Audio

Read in an audio file and listen to it.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
sound(audioIn,fs)
```

Increase the pitch by 3 semitones and listen to the result.

```
nsemitones = 3;
audioOut = shiftPitch(audioIn,nsemitones);
sound(audioOut,fs)
```

Decrease the pitch of the original audio by 3 semitones and listen to the result.

```
nsemitones = -3;
audioOut = shiftPitch(audioIn,nsemitones);
sound(audioOut,fs)
```

## Apply Pitch-Shifting to Frequency-Domain Audio

Read in an audio file and listen to it.

```
[audioIn,fs] = audioread("SpeechDFT-16-8-mono-5secs.wav");
sound(audioIn,fs)
```

Convert the audio signal to a time-frequency representation using `stft`. Use a 512-point `kbdwin` with 75% overlap.

```
win = kbdwin(512);
overlapLength = 0.75*numel(win);

S = stft(audioIn, ...
    "Window",win, ...
    "OverlapLength",overlapLength, ...
    "Centered",false);
```

Increase the pitch by 8 semitones and listen to the result. Specify the window and overlap length you used to compute the STFT.

```
nsemitones = ;
lockPhase = ;
audioOut = shiftPitch(S,nsemitones, ...
    "Window",win, ...
    "OverlapLength",overlapLength, ...
    "LockPhase",lockPhase);

sound(audioOut,fs)
```

Decrease the pitch of the original audio by 8 semitones and listen to the result. Specify the window and overlap length you used to compute the STFT.

```
nsemitones = ;
lockPhase = ;
audioOut = shiftPitch(S,nsemitones, ...
    "Window",win, ...
    "OverlapLength",overlapLength, ...
```

```
        "LockPhase",lockPhase);  
sound(audioOut,fs)
```

### Increase Fidelity Using Phase Locking

Read in an audio file and listen to it.

```
[audioIn,fs] = audioread('FemaleSpeech-16-8-mono-3secs.wav');  
sound(audioIn,fs)
```

Increase the pitch by 6 semitones and listen to the result.

```
nsemitones = 6;  
lockPhase = false;  
audioOut = shiftPitch(audioIn,nsemitones, ...  
    "LockPhase",lockPhase);  
sound(audioOut,fs)
```

To increase fidelity, set LockPhase to true. Apply pitch shifting, and listen to the results.

```
lockPhase = true;  
audioOut = shiftPitch(audioIn,nsemitones, ...  
    "LockPhase",lockPhase);  
sound(audioOut,fs)
```

### Increase Fidelity Using Formant Preservation

Read in the first 11.5 seconds of an audio file and listen to it.

```
[audioIn,fs] = audioread('Rainbow-16-8-mono-114secs.wav',[1,8e3*11.5]);  
sound(audioIn,fs)
```

Increase the pitch by 4 semitones and apply phase locking. Listen to the results. The resulting audio has a "chipmunk effect" that sounds unnatural.

```
nsemitones = ;  
lockPhase = ;  
audioOut = shiftPitch(audioIn,nsemitones, ...  
    "LockPhase",lockPhase);
```

```
sound(audioOut, fs)
```

To increase fidelity, set `PreserveFormants` to `true`. Use the default cepstral order of 30. Listen to the result.

```
cepstralOrder = ;  
audioOut = shiftPitch(audioIn, nsemitones, ...  
    "LockPhase", lockPhase, ...  
    "PreserveFormants", true, ...  
    "CepstralOrder", cepstralOrder);  
  
sound(audioOut, fs)
```

## Input Arguments

### **audioIn** — Input signal

column vector | matrix | 3-D array

Input signal, specified as a column vector, matrix, or 3-D array. How the function interprets `audioIn` depends on the complexity of `audioIn`:

- If `audioIn` is real, `audioIn` is interpreted as a time-domain signal. In this case, `audioIn` must be a column vector or matrix. Columns are interpreted as individual channels.
- If `audioIn` is complex, `audioIn` is interpreted as a frequency-domain signal. In this case, `audioIn` must be an  $L$ -by- $M$ -by- $N$  array, where  $L$  is the FFT length,  $M$  is the number of individual spectrums, and  $N$  is the number of channels.

Data Types: `single` | `double`

Complex Number Support: Yes

### **nsemitones** — Number of semitones to shift audio by

real scalar

Number of semitones to shift the audio by, specified as a real scalar.

The range of `nsemitones` depends on the window length (`numel(Window)`) and the overlap length (`OverlapLength`):

$$-12 \cdot \log_2(\text{numel}(\text{Window}) - \text{OverlapLength}) \leq \text{nsemitones} \leq -12 \cdot \log_2((\text{numel}(\text{Window}) - \text{OverlapLength}) / \text{numel}(\text{Window}))$$

Data Types: single | double

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Window', kbdwin(512)`

### **Window** — Window applied in time domain

`sqrt(hann(1024, 'periodic'))` (default) | real vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range `[1, size(audioIn, 1)]`. The number of elements in the vector must also be greater than `OverlapLength`.

---

**Note** If using `shiftPitch` with frequency-domain input, you must specify `Window` as the same window used to transform `audioIn` to the frequency domain.

---

Data Types: single | double

### **OverlapLength** — Number of samples overlapped between adjacent windows

`round(0.75 * numel(Window))` (default) | scalar in the range `[0, numel(Window))`

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of `'OverlapLength'` and an integer in the range `[0, numel(Window))`.

---

**Note** If using `shiftPitch` with frequency-domain input, you must specify `OverlapLength` as the same overlap length used to transform `audioIn` to a time-frequency representation.

---

Data Types: single | double



**LockPhase — Apply identity phase locking**`false (default) | true`

Apply identity phase locking, specified as the comma-separated pair consisting of 'LockPhase' and `false` or `true`.

Data Types: `logical`

**PreserveFormants — Preserve formants**`false (default) | true`

Preserves formants, specified as the comma-separated pair consisting of 'PreserveFormants' and `true` or `false`. Formant preservation is attempted using spectral envelope estimation with cepstral analysis.

Data Types: `logical`

**CepstralOrder — Cepstral order used for formant preservation**`30 (default) | nonnegative integer`

Cepstral order used for formant preservation, specified as the comma-separated pair consisting of 'CepstralOrder' and a nonnegative integer.

**Dependencies**

To enable this name-value pair argument, set `PreserveFormants` to `true`.

Data Types: `single` | `double`

## Output Arguments

**audioOut — Pitch-shifted audio**`column vector | matrix`

Pitch-shifted audio, returned as a column vector or matrix of independent channels.

## Algorithms

To apply pitch shifting, `shiftPitch` modifies the time-scale of audio using a phase vocoder and then resamples the modified audio. The time scale modification algorithm is based on [1] and [2] and is implemented as in `stretchAudio`.

After time-scale modification, `shiftPitch` performs sample rate conversion using an interpolation factor equal to the analysis hop length and a decimation factor equal to the synthesis hop length. The interpolation and decimation factors of the resampling stage are selected as follows: The analysis hop length is determined as  $\text{analysisHopLength} = \text{numel}(\text{Window}) - \text{OverlapLength}$ . The `shiftPitch` function assumes that there are 12 semitones in an octave, so the speedup factor used to stretch the audio is  $\text{speedupFactor} = 2^{(-\text{nsemitones}/12)}$ . The speedup factor and analysis hop length determine the synthesis hop length for time-scale modification as  $\text{synthesisHopLength} = \text{round}((1/\text{SpeedupFactor}) * \text{analysisHopLength})$ .

The achievable pitch shift is determined by the window length ( $\text{numel}(\text{Window})$ ) and `OverlapLength`. To see the relationship, note that the equation for speedup factor can be rewritten as:  $\text{nsemitones} = -12 * \log_2(\text{speedupFactor})$ , and the equation for synthesis hop length can be rewritten as  $\text{speedupFactor} = \text{analysisHopLength} / \text{synthesisHopLength}$ . Using simple substitution,  $\text{nsemitones} = -12 * \log_2(\text{analysisHopLength} / \text{synthesisHopLength})$ . The practical range of a synthesis hop length is  $[1, \text{numel}(\text{Window})]$ . The range of achievable pitch shifts is:

- Max number of semitones lowered:  $-12 * \log_2(\text{numel}(\text{Window}) - \text{OverlapLength})$
- Max number of semitones raised:  $-12 * \log_2((\text{numel}(\text{Window}) - \text{OverlapLength}) / \text{numel}(\text{Window}))$

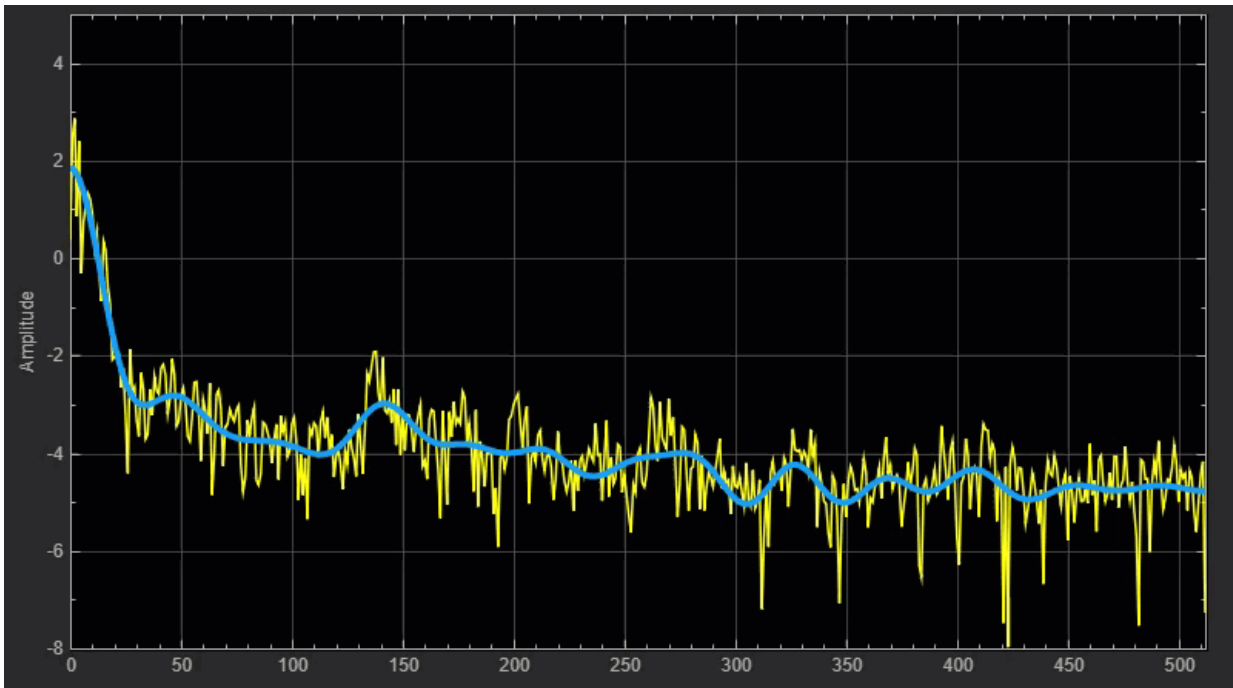
## Formant Preservation

Pitch shifting can alter the spectral envelope of the pitch-shifted signal. To diminish this effect, you can set `PreserveFormants` to `true`. If `PreserveFormants` is set to `true`, the algorithm attempts to estimate the spectral envelope using an iterative procedure in the cepstral domain, as described in [3] and [4]. For both the original spectrum,  $X$ , and the pitch-shifted spectrum,  $Y$ , the algorithm estimates the spectral envelope as follows.

For the first iteration,  $\text{Env}X_a$  is set to  $X$ . Then, the algorithm repeats these two steps in a loop:

- 1 Lowpass filters the cepstral representation of  $\text{Env}X_a$  to get a new estimate,  $\text{Env}X_b$ . The `CepstralOrder` parameter controls the quefreny bandwidth.
- 2 To update the current best fit, the algorithm takes the element-by-element maximum of the current spectral envelope estimate and the previous spectral envelope estimate:

$$\text{Env}X_a = \max(\text{Env}X_a, \text{Env}X_b).$$



The loop ends if either a maximum number of iterations (100) is reached, or if all bins of the estimated log envelope are within a given tolerance of the original log spectrum. The tolerance is set to  $\log(10^{(1/20)})$ .

Finally, the algorithm scales the spectrum of the pitch-shifted audio by the ratio of estimated envelopes, element-wise:

$$Y = Y \times \left( \frac{EnvX_b}{EnvY_b} \right).$$

## References

- [1] Driedger, Johnathan, and Meinard Müller. "A Review of Time-Scale Modification of Music Signals." *Applied Sciences*. Vol. 6, Issue 2, 2016.
- [2] Driedger, Johnathan. "Time-Scale Modification Algorithms for Music Audio Signals." Master's Thesis. Saarland University, Saarbrücken, Germany, 2011.

- [3] Axel Roebel, and Xavier Rodet. "Efficient Spectral Envelope Estimation and its application to pitch shifting and envelope preservation." International Conference on Digital Audio Effects, pp. 30-35. Madrid, Spain, September 2005. hal-01161334
- [4] S. Imai, and Y. Abe. "Spectral envelope extraction by improved cepstral method." *Electron. and Commun. in Japan*. Vol. 62-A, Issue 4, 1997, pp. 10-17.

## **Extended Capabilities**

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

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

### **See Also**

`audioDataAugmenter` | `audioTimeScaler` | `reverberator` | `stretchAudio`

**Introduced in R2019b**

# designAuditoryFilterBank

Design auditory filter bank

## Syntax

```
filterBank = designAuditoryFilterBank(fs)
filterBank = designAuditoryFilterBank(fs,Name,Value)
[filterBank,Fc,BW] = designAuditoryFilterBank( ___ )
```

## Description

`filterBank = designAuditoryFilterBank(fs)` returns a frequency-domain auditory filter bank, `filterBank`.

`filterBank = designAuditoryFilterBank(fs,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

`[filterBank,Fc,BW] = designAuditoryFilterBank( ___ )` returns the center frequency and bandwidth of each filter in the filter bank. You can use this output syntax with any of the previous input syntaxes.

## Examples

### Create Default Auditory Filter Bank

Call `designAuditoryFilterBank` with a specified sample rate to design the default auditory filter bank.

```
fs = 44.1e3;
fb = designAuditoryFilterBank(fs);
```

The default filter bank consists of 32 triangular bandpass filters spaced evenly on the mel scale between 0 and  $fs/2$  Hz.

```
numBands = size(fb,1)
```

```
numBands = 32
```

`designAuditoryFilterBank` is intended for use in frequency-domain filtering for a one-sided spectrum. By default, `designAuditoryFilterBank` assumes a 1024-point DFT, so it returns a half-sided frequency-domain filter bank with 513 points.

```
numPoints = size(fb,2)
```

```
numPoints = 513
```

### Design Mel-Based Auditory Filter Bank

Read in audio and convert it to a one-sided power spectrum.

```
[audioIn,fs] = audioread("Laughter-16-8-mono-4secs.wav");
```

```
win = hamming(1024,"periodic");
```

```
noverlap = 512;
```

```
fftLength = 1024;
```

```
[~,F,t,PowerSpectrum] = spectrogram(audioIn,win,noverlap,fftLength,"power","onesided",t);
```

Design a mel-based auditory filter bank. Plot the filter bank.

```
numBands = ;
```

```
range = [,];
```

```
normalization = ;
```

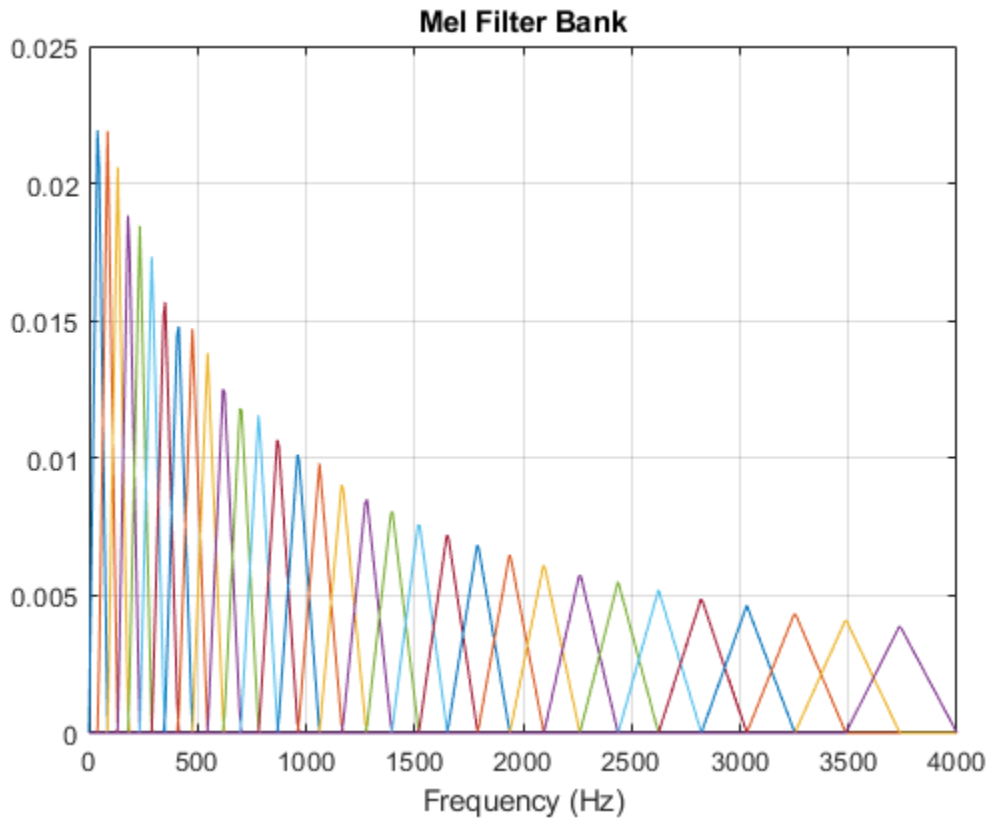
```
[fb,cf] = designAuditoryFilterBank(fs, ...  
    "FFTLength",fftLength, ...  
    "NumBands",numBands, ...  
    "FrequencyRange",range, ...  
    "Normalization",normalization);
```

```
plot(F,fb.')
```

```
grid on
```

```
title("Mel Filter Bank")
```

```
xlabel("Frequency (Hz)")
```



To apply frequency domain filtering, perform a matrix multiplication of the filter bank and the power spectrogram.

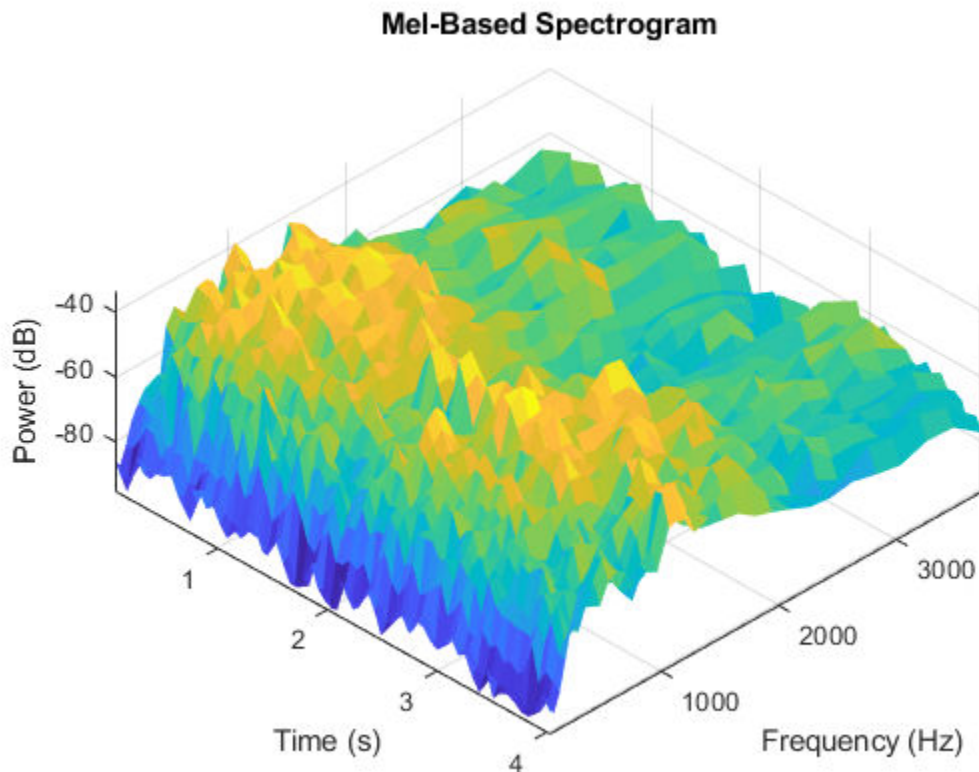
```
X = fb*PowerSpectrum;
```

Visualize the power-per-band in dB.

```
XdB = 10*log10(X);
```

```
surf(t,cf,XdB,"EdgeColor","none");
xlabel("Time (s)")
ylabel("Frequency (Hz)")
zlabel("Power (dB)")
view([45,60])
```

```
title('Mel-Based Spectrogram')  
axis tight
```



### Design Bark-Based Auditory Filter Bank

Read in audio and convert it to a one-sided power spectrum.

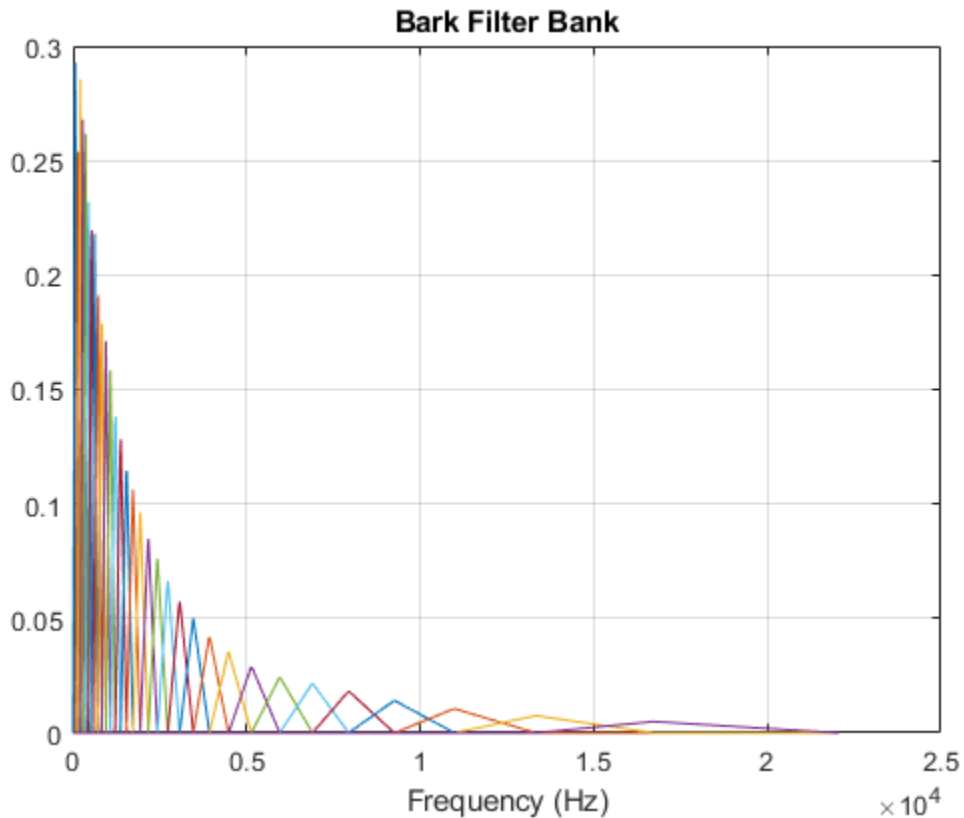
```
[audioIn,fs] = audioread("RockDrums-44p1-stereo-11secs.mp3");  
  
win = hann(round(0.03*fs),"periodic");  
noverlap = round(0.02*fs);  
fftLength = 2048;
```



```
[~,F,t,PowerSpectrumLeft] = spectrogram(audioIn(:,1),win,noverlap,fftLength,"power","c");  
[~,~,~,PowerSpectrumRight] = spectrogram(audioIn(:,2),win,noverlap,fftLength,"power","c");
```

Design a Bark-based auditory filter bank. Plot the filter bank.

```
numBands = ;  
range = [,];  
normalization = ;  
  
[fb,cf] = designAuditoryFilterBank(fs, ...  
    "FrequencyScale","bark", ...  
    "FFTLength",fftLength, ...  
    "NumBands",numBands, ...  
    "FrequencyRange",range, ...  
    "Normalization",normalization);  
  
plot(F,fb. ');  
grid on  
title("Bark Filter Bank")  
xlabel("Frequency (Hz)")
```



To apply frequency domain filtering, perform a matrix multiplication of the filter bank and the left and right power spectrograms.

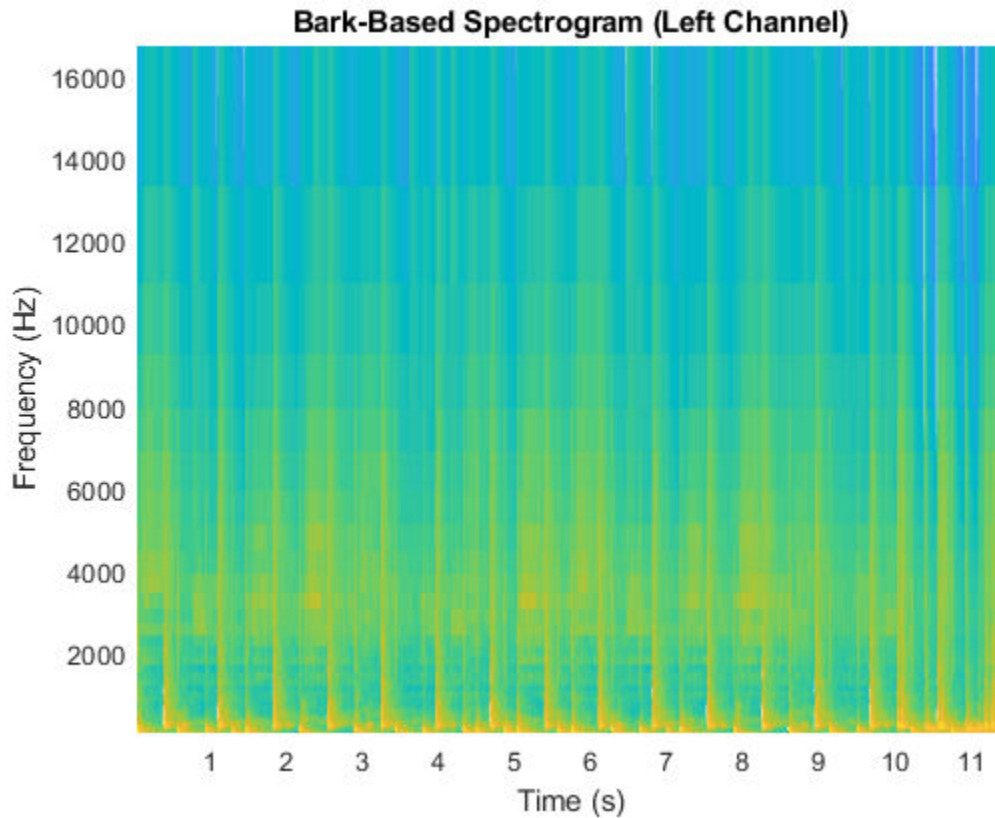
```
XL = fb*PowerSpectrumLeft;
XR = fb*PowerSpectrumRight;
```

Visualize the power-per-band in dB.

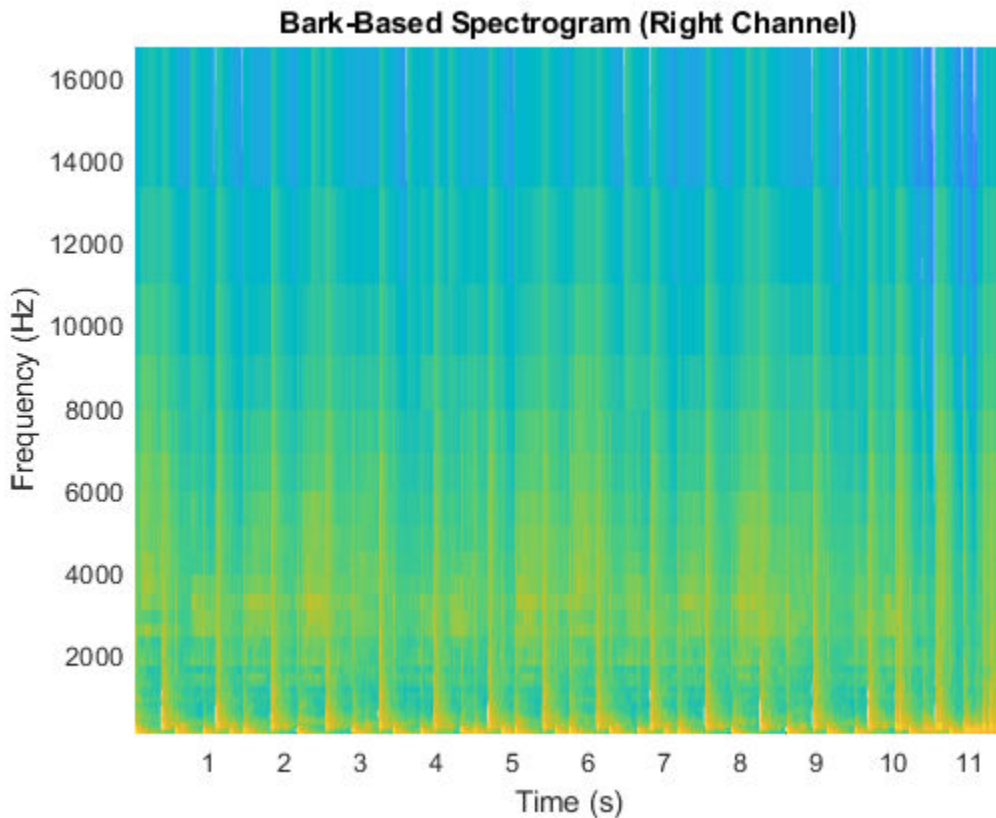
```
XLdB = 10*log10(XL);
XRdB = 10*log10(XR);

surf(t,cf,XLdB,"EdgeColor","none");
xlabel("Time (s)")
ylabel("Frequency (Hz)")
```

```
view([0,90])  
title("Bark-Based Spectrogram (Left Channel)")  
axis tight
```



```
surf(t,cf,XRdB,"EdgeColor","none");  
xlabel("Time (s)")  
ylabel("Frequency (Hz)")  
view([0,90])  
title("Bark-Based Spectrogram (Right Channel)")  
axis tight
```



### Design ERB-Based Auditory Filter Bank

Read in audio and convert it to a one-sided power spectrum.

```
[audioIn,fs] = audioread("NoisySpeech-16-22p5-mono-5secs.wav");
```

```
win = hann(round(0.04*fs),"periodic");
```

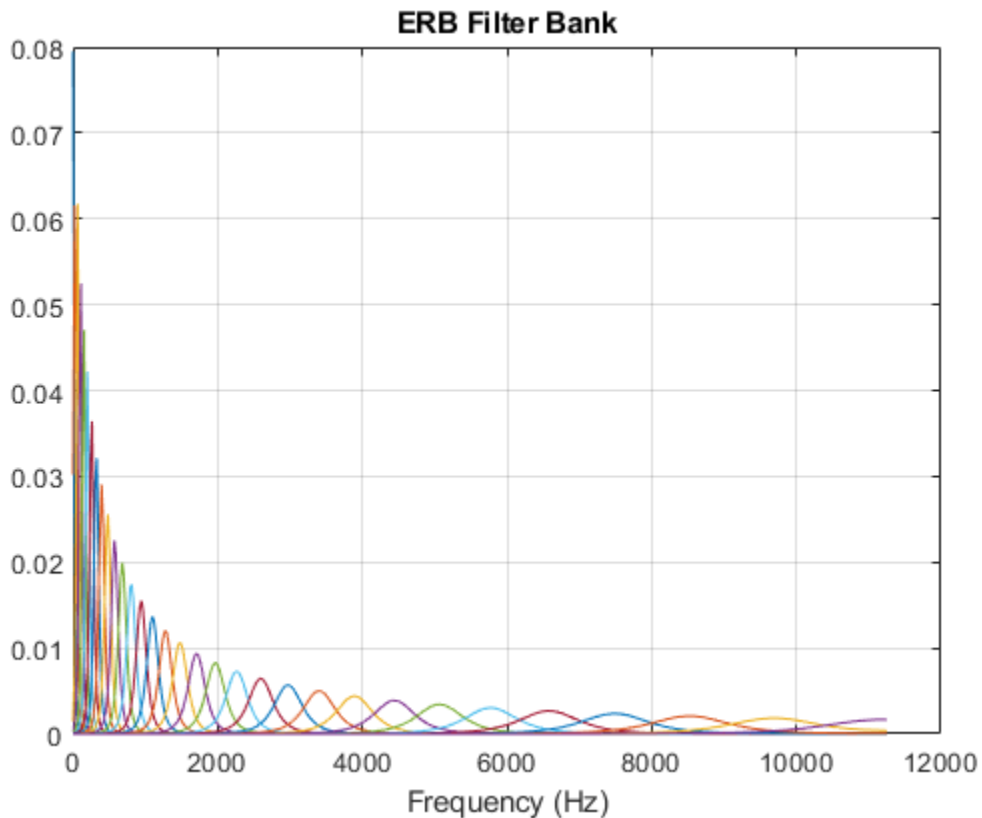
```
noverlap = round(0.02*fs);
```

```
fftLength = 1024;
```

```
[~,F,t,PowerSpectrum] = spectrogram(audioIn,win,noverlap,fftLength,"power","onesided");
```

Design an ERB-based auditory filter bank. Plot the filter bank.

```
numBands = ;  
range = [,];  
normalization = ;  
  
[fb,cf] = designAuditoryFilterBank(fs, ...  
    "FrequencyScale","erb", ...  
    "FFTLength",fftLength, ...  
    "NumBands",numBands, ...  
    "FrequencyRange",range, ...  
    "Normalization",normalization);  
  
plot(F,fb. ');  
grid on  
title("ERB Filter Bank")  
xlabel("Frequency (Hz)")
```

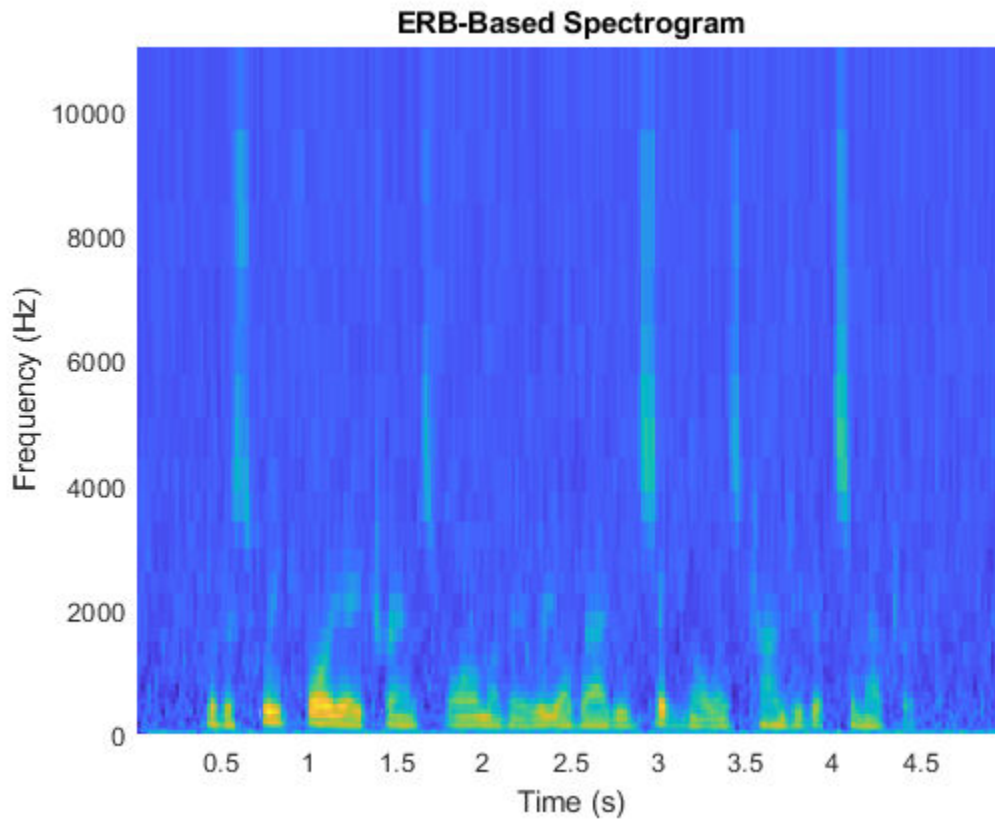


To apply frequency-domain filtering, perform a matrix multiplication of the filter bank and the power spectrogram.

```
X = fb*PowerSpectrum;
```

Visualize the power-per-band in dB.

```
XdB = 10*log10(X);  
surf(t,cf,XdB,"EdgeColor","none");  
xlabel("Time (s)")  
ylabel("Frequency (Hz)")  
view([0,90])  
title("ERB-Based Spectrogram")  
axis tight
```



## Input Arguments

**fs** — Sample rate of filter design (Hz)

positive scalar

Sample rate of filter design in Hz, specified as a positive scalar.

Data Types: `single` | `double`

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `"FrequencyScale", "mel"`

### **FrequencyScale** — Frequency scale

`"mel"` (default) | `"bark"` | `"erb"`

Frequency scale used to design the auditory filter bank, specified as the comma-separated pair consisting of `'FrequencyScale'` and `"mel"`, `"bark"`, or `"erb"`.

Data Types: `char` | `string`

### **FFTLength** — Number of DFT points

1024 (default) | positive integer

Number of points used to calculate the DFT, specified as the comma-separated pair consisting of `'FFTLength'` and a positive integer.

Data Types: `single` | `double`

### **NumBands** — Number of bandpass filters

positive integer

Number of bandpass filters, specified as the comma-separated pair consisting of `'NumBands'` and a positive integer. The default number of bandpass filters depends on the `FrequencyScale`:

- If `FrequencyScale` is set to `"bark"` or `"mel"`, then `NumBands` defaults to 32.
- If `FrequencyScale` is set to `"erb"`, then `NumBands` defaults to `ceil(hz2erb(FrequencyRange(2)) - hz2erb(FrequencyRange(1)))`.

Data Types: `single` | `double`

### **FrequencyRange** — Frequency range over which to design auditory filter bank (Hz)

`[0 fs/2]` (default) | two-element row vector



Frequency range over which to design auditory filter bank in Hz, specified as the comma-separated pair consisting of 'FrequencyRange' and a two-element row vector of monotonically increasing values in the range  $[0, fs/2]$ .

Data Types: `single` | `double`

### **Normalization — Normalize filter bank**

`"bandwidth"` (default) | `"area"` | `"none"`

Normalization technique used on the weights of the filter bank:

- `"bandwidth"` -- The weights of each bandpass filter are normalized by the corresponding bandwidth of the filter.
- `"area"` -- The weights of each bandpass filter are normalized by the corresponding area of the bandpass filter.
- `"none"` -- The weights of the filters are not normalized.

Data Types: `char` | `string`

## **Output Arguments**

### **filterBank — Auditory filter bank**

column vector | matrix

Auditory filter bank, returned as an  $M$ -by- $N$  matrix, where  $M$  is the number of bands (`NumBands`), and  $N$  is the number of frequency points of a one-sided spectrum (`ceil(FFTLength/2)`).

Data Types: `double`

### **Fc — Center frequencies of bandpass filters (Hz)**

row vector

Center frequencies of bandpass filters in Hz, returned as a row vector with `NumBands` elements.

Data Types: `double`

### **BW — Bandwidth of bandpass filters (Hz)**

row vector

Bandwidth of bandpass filters in Hz, returned as a row vector with `NumBands` elements.

Data Types: `double`

## Algorithms

The mel filter bank is designed as half-overlapped triangles equally spaced on the mel scale. [1]

The Bark filter bank is designed as half-overlapped triangles equally spaced on the Bark scale. [2]

The ERB filter bank is designed as gammatone filters [4] whose center frequencies are equally spaced on the ERB scale. [3]

## References

- [1] O'Shaghnessy, Douglas. *Speech Communication: Human and Machine*. Reading, MA: Addison-Wesley Publishing Company, 1987.
- [2] Traunmüller, Hartmut. "Analytical Expressions for the Tonotopic Sensory Scale." *Journal of the Acoustical Society of America*. Vol. 88, Issue 1, 1990, pp. 97-100.
- [3] Glasberg, Brian R., and Brian C. J. Moore. "Derivation of Auditory Filter Shapes from Notched-Noise Data." *Hearing Research*. Vol. 47, Issues 1-2, 1990, pp. 103-138.
- [4] Slaney, Malcolm. "An Efficient Implementation of the Patterson-Holdworth Auditory Filter Bank." Apple Computer Technical Report 35, 1993.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`bark2hz` | `erb2hz` | `gammatoneFilterBank` | `hz2bark` | `hz2erb` | `hz2mel` | `mel2hz` | `melSpectrogram`

**Introduced in R2019b**

# melSpectrogram

Mel spectrogram

## Syntax

```
S = melSpectrogram(audioIn,fs)
S = melSpectrogram(audioIn,fs,Name,Value)
[S,F,T] = melSpectrogram( ___ )
melSpectrogram( ___ )
```

## Description

`S = melSpectrogram(audioIn,fs)` returns the mel spectrogram of the audio input at sample rate `fs`. The function treats columns of the input as individual channels.

`S = melSpectrogram(audioIn,fs,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

`[S,F,T] = melSpectrogram( ___ )` returns the center frequencies of the bands in Hz and the location of each window of data in seconds. The location corresponds to the center of each window. You can use this output syntax with any of the previous input syntaxes.

`melSpectrogram( ___ )` plots the mel spectrogram on a surface in the current figure.

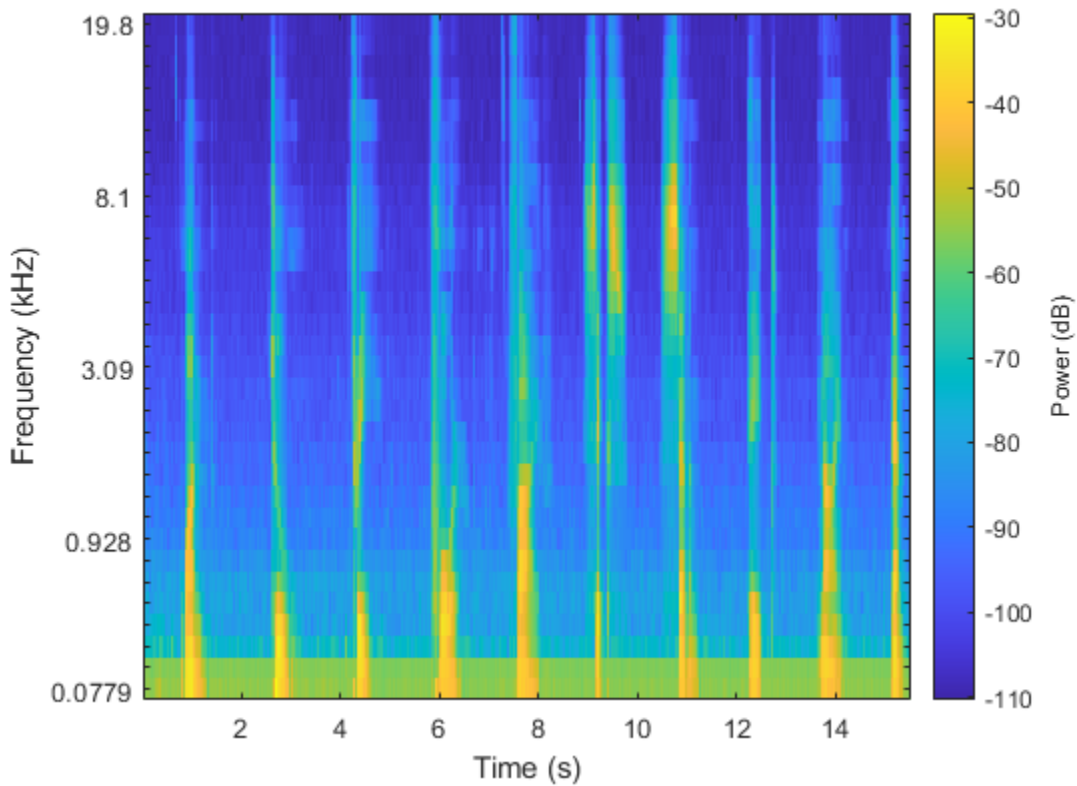
## Examples

### Calculate Mel Spectrogram

Use the default settings to calculate the mel spectrogram for an entire audio file. Print the number of bandpass filters in the filter bank and the number of frames in the mel spectrogram.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

```
S = melSpectrogram(audioIn,fs);  
  
[numBands,numFrames] = size(S);  
fprintf("Number of bandpass filters in filterbank: %d\n",numBands)  
  
Number of bandpass filters in filterbank: 32  
  
fprintf("Number of frames in spectrogram: %d\n",numFrames)  
  
Number of frames in spectrogram: 1551  
  
Plot the mel spectrogram.  
  
melSpectrogram(audioIn,fs)
```



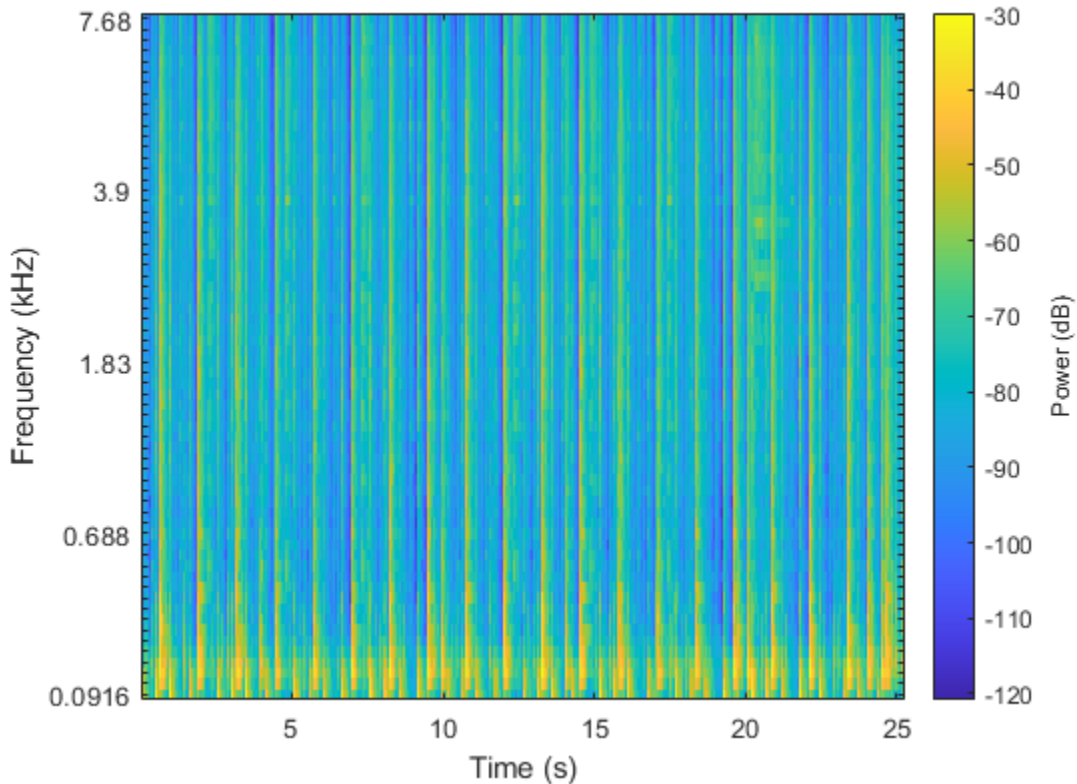
### Calculate Mel Spectrums of 2048-Point Windows

Calculate the mel spectrums of 2048-point windows with 1024-point overlap. Convert to the frequency domain using a 4096-point FFT. Pass the frequency-domain representation through 64 half-overlapped triangular bandpass filters that span the range 62.5 Hz to 8 kHz.

```
[audioIn,fs] = audioread('FunkyDrums-44p1-stereo-25secs.mp3');  
  
S = melSpectrogram(audioIn,fs, ...  
    'WindowLength',2048,...  
    'OverlapLength',1024, ...  
    'FFTLength',4096, ...  
    'NumBands',64, ...  
    'FrequencyRange',[62.5,8e3]);
```

Call `melSpectrogram` again, this time with no output arguments so that you can visualize the mel spectrogram. The input audio is a multichannel signal. If you call `melSpectrogram` with a multichannel input and with no output arguments, only the first channel is plotted.

```
melSpectrogram(audioIn,fs, ...  
    'WindowLength',2048,...  
    'OverlapLength',1024, ...  
    'FFTLength',4096, ...  
    'NumBands',64, ...  
    'FrequencyRange',[62.5,8e3])
```



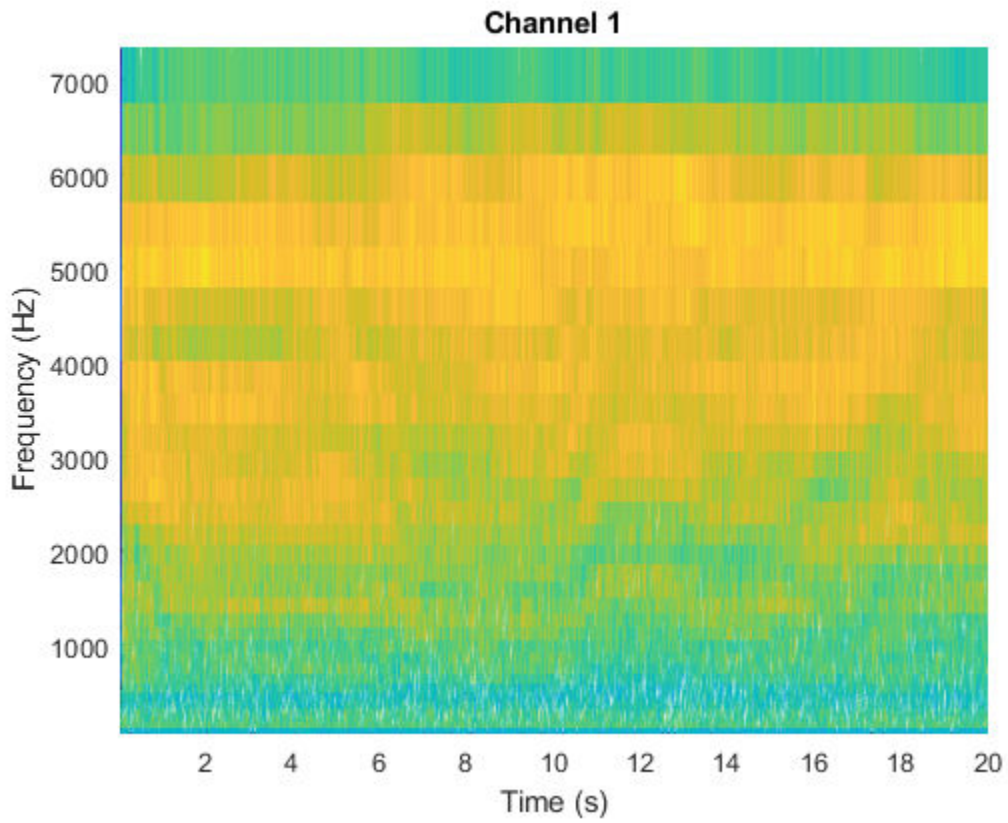
### Get Filter Bank Center Frequencies and Analysis Window Time Instants

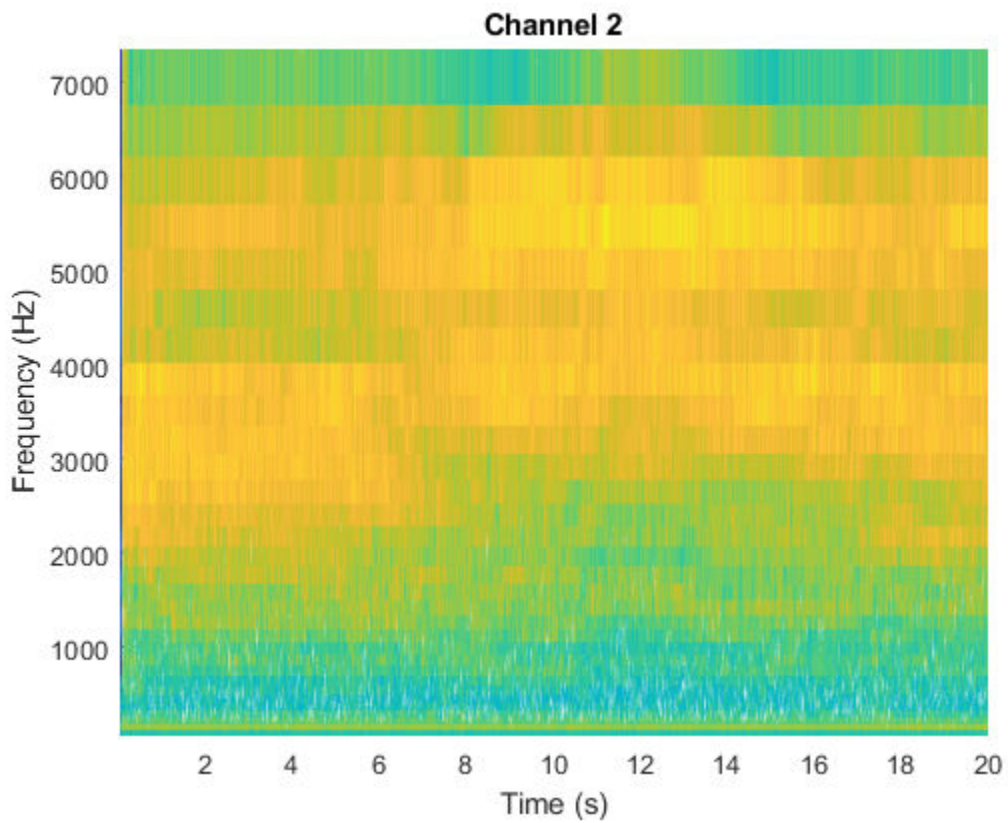
`melSpectrogram` applies a frequency-domain filter bank to audio signals that are windowed in time. You can get the center frequencies of the filters and the time instants corresponding to the analysis windows as the second and third output arguments from `melSpectrogram`.

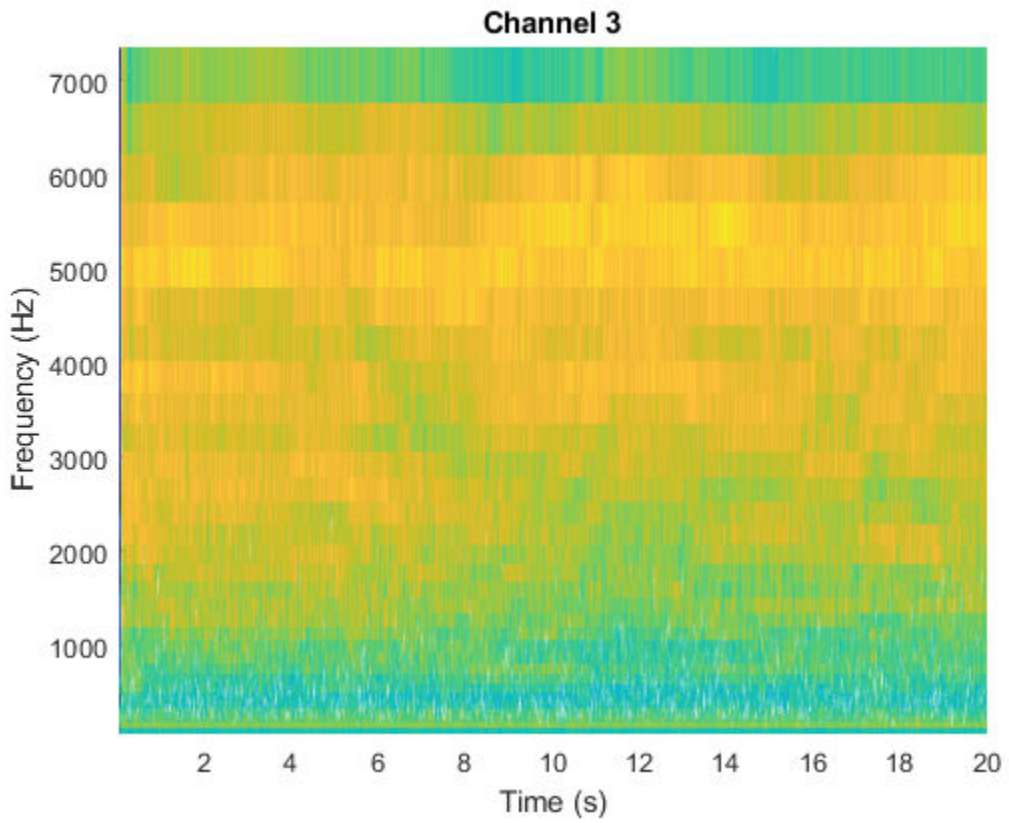
Get the mel spectrogram, filter bank center frequencies, and analysis window time instants of a multichannel audio signal. Use the center frequencies and time instants to plot the mel spectrogram for each channel.

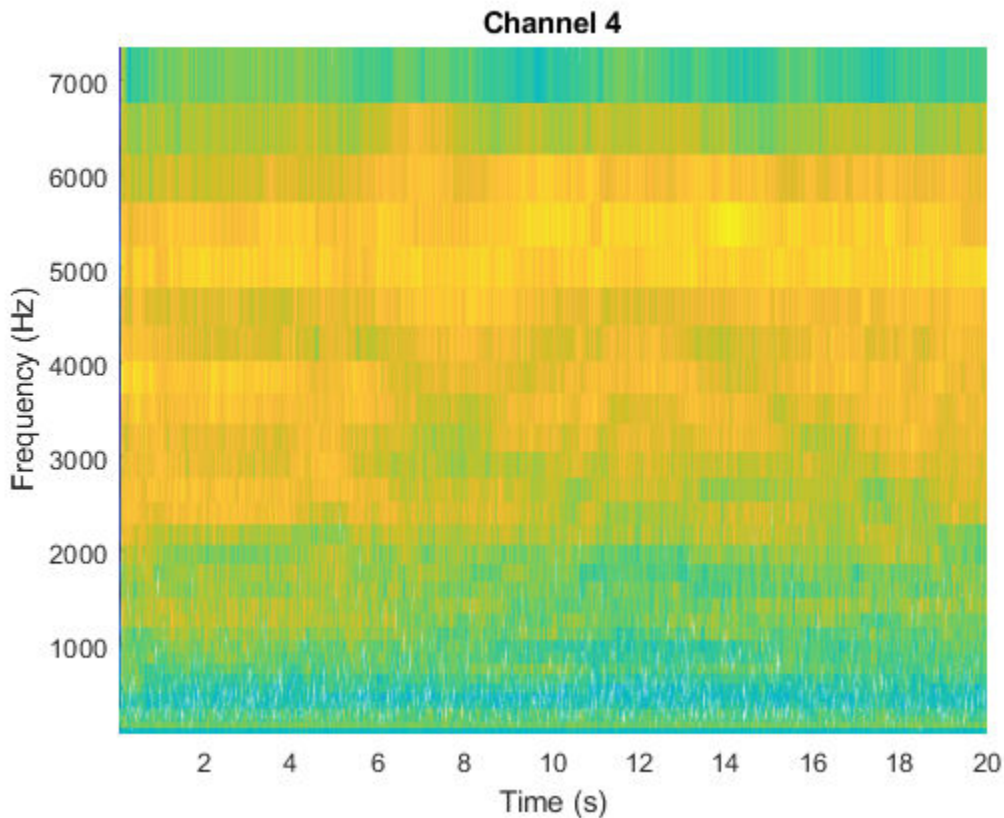
```
[audioIn,fs] = audioread('AudioArray-16-16-4channels-20secs.wav');  
[S,cF,t] = melSpectrogram(audioIn,fs);  
S = 10*log10(S+eps); % Convert to dB for plotting  
for i = 1:size(S,3)  
    figure(i)  
    surf(t,cF,S(:,:,i),'EdgeColor','none');  
    xlabel('Time (s)')  
    ylabel('Frequency (Hz)')  
    view([0,90])  
    title(sprintf('Channel %d',i))  
    axis([t(1) t(end) cF(1) cF(end)])  
end
```











## Input Arguments

### **audioIn** — Audio input

column vector | matrix

Audio input, specified as a column vector or matrix. If specified as a matrix, the function treats columns as independent audio channels.

Data Types: `single` | `double`

### **fs** — Input sample rate (Hz)

positive scalar

Input sample rate in Hz, specified as a positive scalar.

Data Types: `single` | `double`

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'WindowLength', 1024`

### **WindowLength — Analysis window length (samples)**

`round(0.03*fs)` (default) | integer in the range `[2, size(audioIn,1)]`

Analysis window length in samples, specified as the comma-separated pair consisting of `'WindowLength'` and an integer in the range `[2, size(audioIn,1)]`.

Data Types: `single` | `double`

### **OverlapLength — Analysis window overlap length (samples)**

`round(0.02*fs)` (default) | integer in the range `[0, (WindowLength - 1)]`

Analysis window overlap length in samples, specified as the comma-separated pair consisting of `'OverlapLength'` and an integer in the range `[0, (WindowLength - 1)]`.

Data Types: `single` | `double`

### **FFTLength — Number of DFT points**

`WindowLength` (default) | positive integer

Number of points used to calculate the DFT, specified as the comma-separated pair consisting of `'FFTLength'` and a positive integer greater than or equal to `WindowLength`. If unspecified, `FFTLength` defaults to `WindowLength`.

Data Types: `single` | `double`

### **NumBands — Number of mel bandpass filters**

`32` (default) | positive integer

Number of mel bandpass filters, specified as the comma-separated pair consisting of `'NumBands'` and a positive integer.

Data Types: `single` | `double`

**FrequencyRange — Frequency range over which to compute mel spectrogram (Hz)**

`[0 fs/2]` (default) | two-element row vector

Frequency range over which to compute the mel spectrogram in Hz, specified as the comma-separated pair consisting of 'FrequencyRange' and a two-element row vector of monotonically increasing values in the range `[0, fs/2]`.

Data Types: `single` | `double`

**SpectrumType — Type of mel spectrogram**

'power' (default) | 'magnitude'

Type of mel spectrogram, specified as the comma-separated pair consisting of 'SpectrumType' and 'power' or 'magnitude'.

Data Types: `char` | `string`

## Output Arguments

**S — Mel spectrogram**

column vector | matrix | 3-D array

Mel spectrogram, returned as a column vector, matrix, or 3-D array. The dimensions of S are *L*-by-*M*-by-*N*, where:

- *L* is the number of frequency bins in each mel spectrum. NumBands and fs determine *L*.
- *M* is the number of frames the audio signal is partitioned into. `size(audioIn,1)`, WindowLength, and OverlapLength determine *M*.
- *N* is the number of channels such that `N = size(audioIn,2)`.

Trailing singleton dimensions are removed from the output S.

Data Types: `single` | `double`

**F — Center frequencies of mel bandpass filters (Hz)**

row vector

Center frequencies of mel bandpass filters in Hz, returned as a row vector with length `size(S,1)`.

Data Types: `single` | `double`

### T — Location of each window of audio (s)

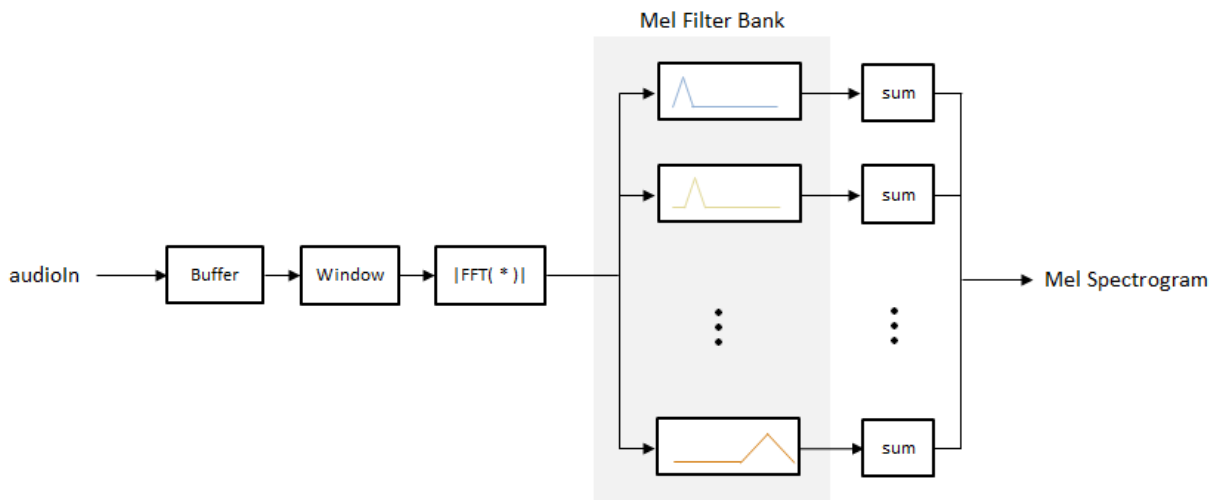
row vector

Location of each analysis window of audio in seconds, returned as a row vector length `size(S,2)`. The location corresponds to the center of each window.

Data Types: `single` | `double`

## Algorithms

The `melSpectrogram` function follows the general algorithm to compute a mel spectrogram as described in [1].

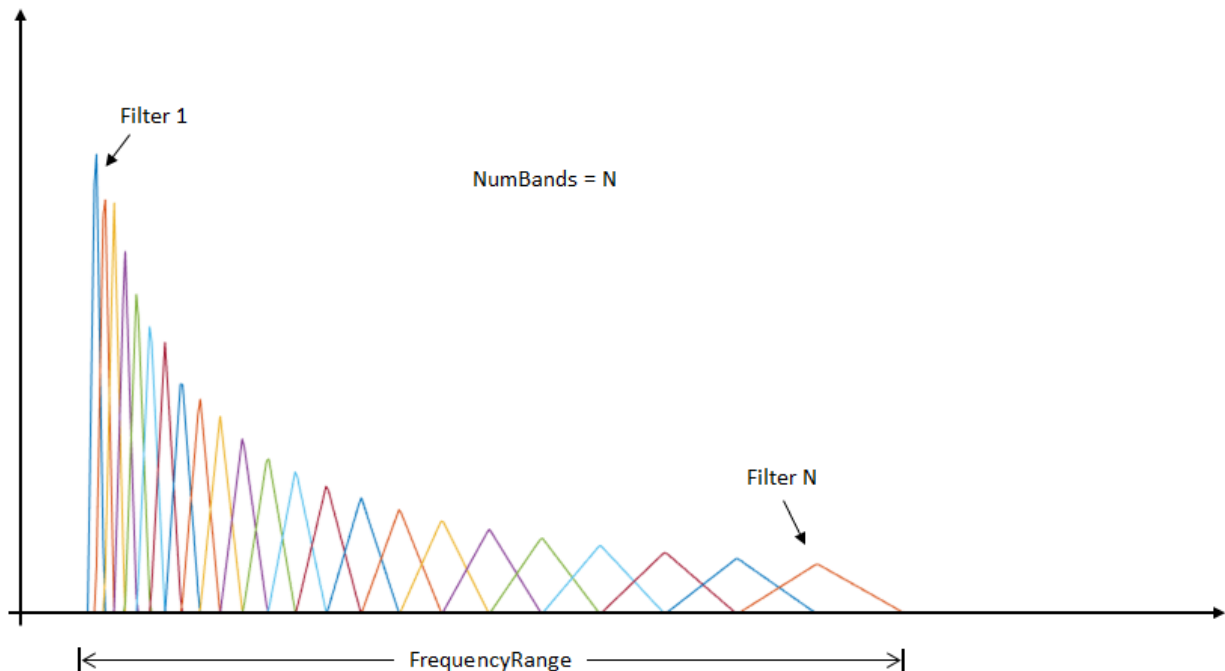


In this algorithm, the audio input is first buffered into frames of `WindowLength` number of samples. The frames are overlapped by `OverlapLength` number of samples. A periodic hamming window is applied to each frame, and then the frame is converted to frequency-domain representation with `FFTLenght` number of points. The frequency-domain representation can be either magnitude or power, specified by `SpectrumType`.

Each frame of the frequency-domain representation passes through a mel filter bank. The spectral values output from the mel filter bank are summed, and then the channels are concatenated so that each frame is transformed to a NumBands-element column vector.

### Filter Bank Design

The mel filter bank is designed as half-overlapped triangular filters equally spaced on the mel scale. NumBands controls the number of mel bandpass filters. FrequencyRange controls the band edges of the first and last filters in the mel filter bank. The filters are normalized by their bandwidths, so that if white noise is input to the system, each filter outputs an equal amount of energy.



### References

- [1] Rabiner, Lawrence R., and Ronald W. Schafer. *Theory and Applications of Digital Speech Processing*. Upper Saddle River, NJ: Pearson, 2010.



## Extended Capabilities

### C/C++ Code Generation

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

### See Also

gtcc | mdct | mfcc | spectrogram

### Topics

“Speech Command Recognition Using Deep Learning”

**Introduced in R2019a**

# kbdwin

Kaiser-Bessel-derived window

## Syntax

```
wdw = kbdwin(N)  
wdw = kbdwin(N,Beta)
```

## Description

`wdw = kbdwin(N)` returns an N-point Kaiser-Bessel-derived (KBD) window.

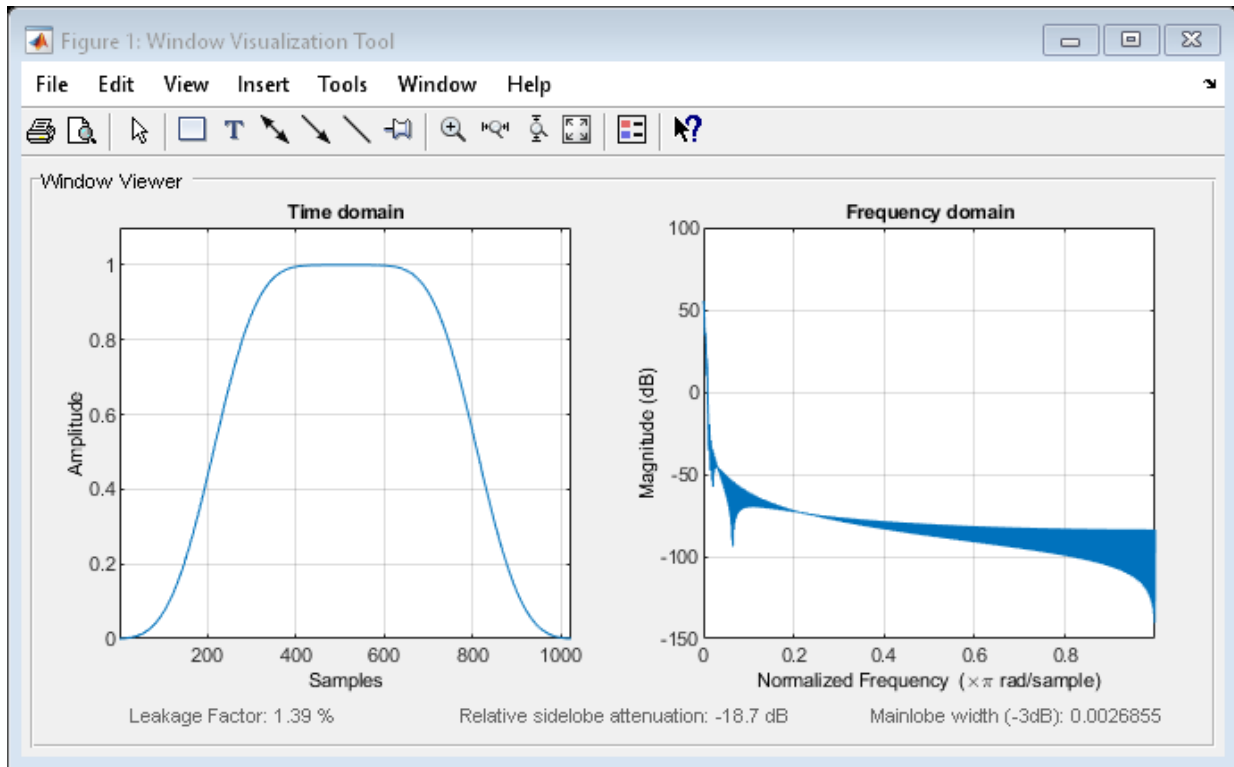
`wdw = kbdwin(N,Beta)` specifies the tuning parameter, `Beta`.

## Examples

### Create Kaiser-Bessel-Derived Window

Create a 1024-point Kaiser-Bessel-derived (KBD) window. Visualize the KBD window in the time and frequency domains using `wvtool`.

```
wdw = kbdwin(1024);  
wvtool(wdw)
```

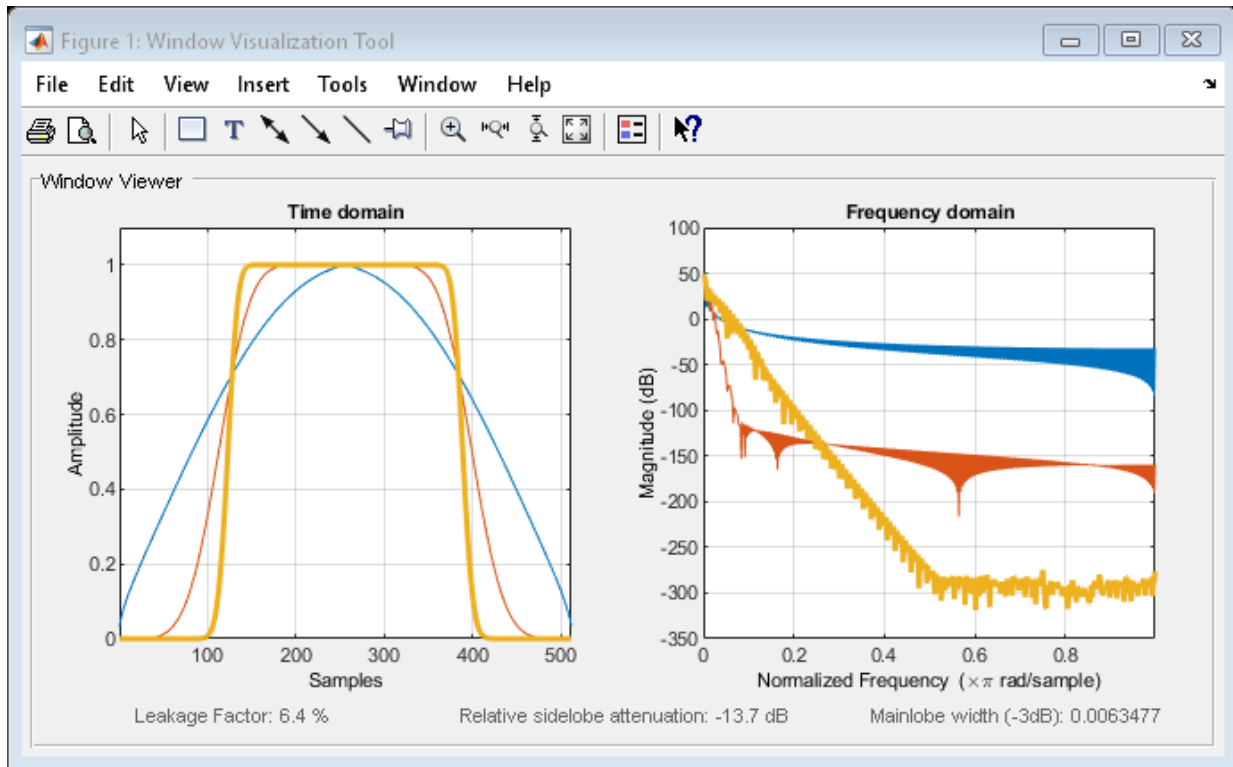


### Effect of Tuning Parameter Beta

Create three 512-point KBD windows, with Beta set to 1, 10, and 100. Display the windows for comparison using `wvtool`.

```
N = 512;
beta1 = kbdwin(N,1);
beta10 = kbdwin(N,10);
beta100 = kbdwin(N,100);

wvtool(beta1,beta10,beta100)
```



## Input Arguments

### **N** — Number of points in KBD window

even positive integer scalar

Number of points in the KBD window, specified as an even positive integer scalar.

Data Types: `single` | `double`

### **Beta** — Tuning parameter

5 (default) | nonnegative real scalar

Tuning parameter, specified as a nonnegative real scalar. If unspecified, Beta defaults to 5.

Data Types: `single` | `double`

## Output Arguments

### **wdw** — Kaiser-Bessel-derived window

N-point column vector

Kaiser-Bessel-derived window, returned as an N-point column vector.

## Algorithms

The coefficients of a Kaiser-Bessel-derived window are computed using the equation:

$$wdw[n] = \begin{cases} \sqrt{\frac{\sum_{i=1}^n w[i]}{\sum_{i=1}^{N/2+1} w[i]}} & \text{if } 1 \leq n < (N/2) \\ \sqrt{\frac{\sum_{i=1}^{N-n} w[i]}{\sum_{i=1}^{N/2+1} w[i]}} & \text{if } (N/2 + 1) \leq n < N \end{cases}$$

where  $w$  is a Kaiser window designed using the `kaiser` function:

```
w = kaiser(N/2+1,Beta*pi)
```

where  $N$  is the number of points in the KBD window and  $Beta$  is the tuning parameter.

## References

- [1] Bosi, Marina, and Richard E. Goldberg. *Introduction to Digital Audio Coding and Standards*. Dordrecht: Kluwer, 2003.

## **Extended Capabilities**

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

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

### **See Also**

kaiser | mdct | window

**Introduced in R2019a**

# mdct

Modified discrete cosine transform

## Syntax

```
Y = mdct(X,win)
Y = mdct(X,win,Name,Value)
[Y,S,Z] = mdct( ___ )
```

## Description

`Y = mdct(X,win)` returns the modified discrete cosine transform (MDCT) of `X`. Before the MDCT is calculated, `X` is buffered into 50% overlapping frames that are each multiplied by the time window `win`. The function treats each column of `X` as an independent channel.

`Y = mdct(X,win,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

`[Y,S,Z] = mdct( ___ )` returns the modified discrete sine transform (MDST), `S`, and the odd discrete Fourier transform (ODFT), `Z`.

## Examples

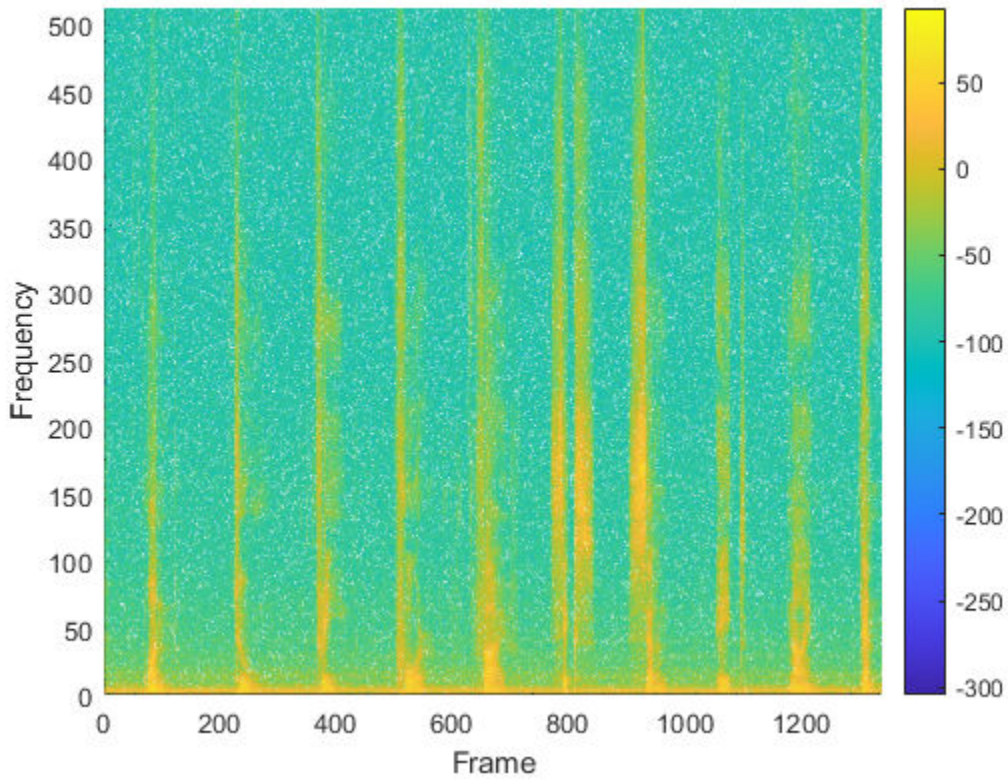
### Calculate MDCT

Read in an audio file and then calculate the MDCT using a 1024-point Kaiser-Bessel-derived window.

```
audioIn = audioread('Counting-16-44p1-mono-15secs.wav');
coef = mdct(audioIn,kbdwin(1024));
```

Plot the power of the MDCT coefficients over time.

```
surf(20*log10(coef.^2), 'EdgeColor', 'none');  
view([0 90])  
xlabel('Frame')  
ylabel('Frequency')  
axis([0 size(coef,2) 0 size(coef,1)])  
colorbar
```





## Effect of Input Padding on Perfect Reconstruction

To enable perfect reconstruction, the `mdct` function zero-pads the front and back of the audio input signal. The signal returned from `imdct` removes the zero padding added for perfect reconstruction.

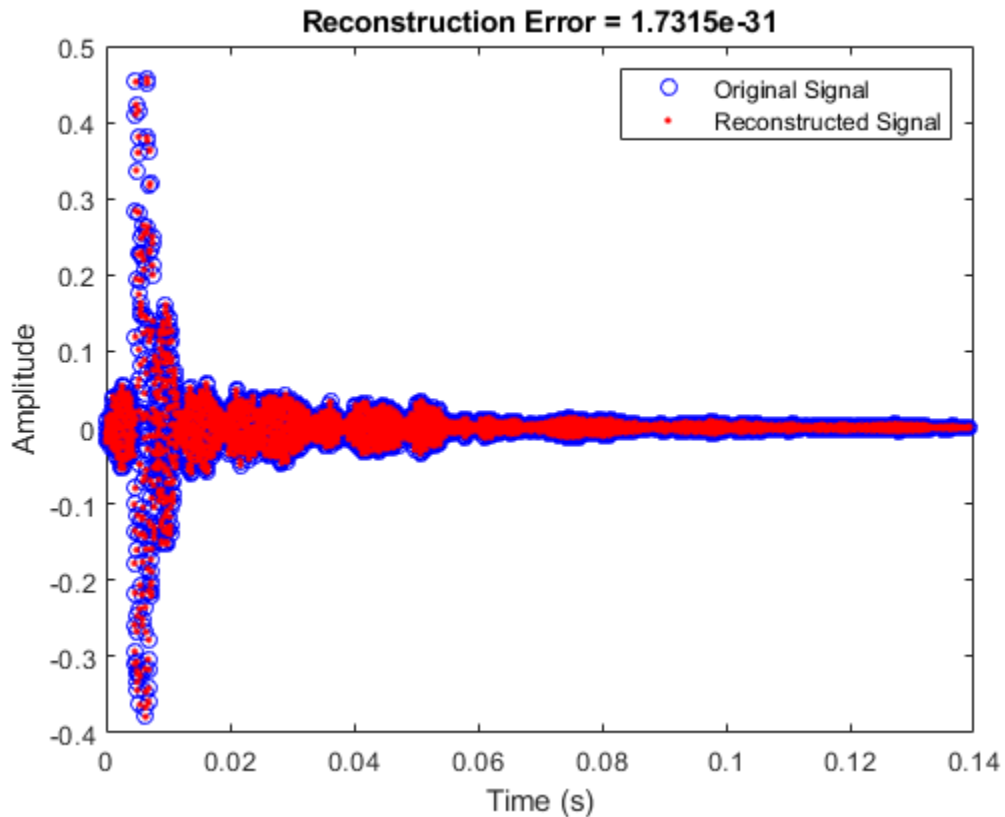
Read in an audio file, create a 2048-point Kaiser-Bessel-derived window, and then clip the audio signal so that its length is a multiple of 2048.

```
[x,fs] = audioread('Click-16-44p1-mono-0.2secs.wav');  
win = kbdwin(2048);
```

```
xClipped = x(1:end - rem(size(x,1),numel(win)));
```

Convert the signal to the frequency domain, and then reconstruct it back in the time domain. Plot the original and reconstructed signals and display the reconstruction error.

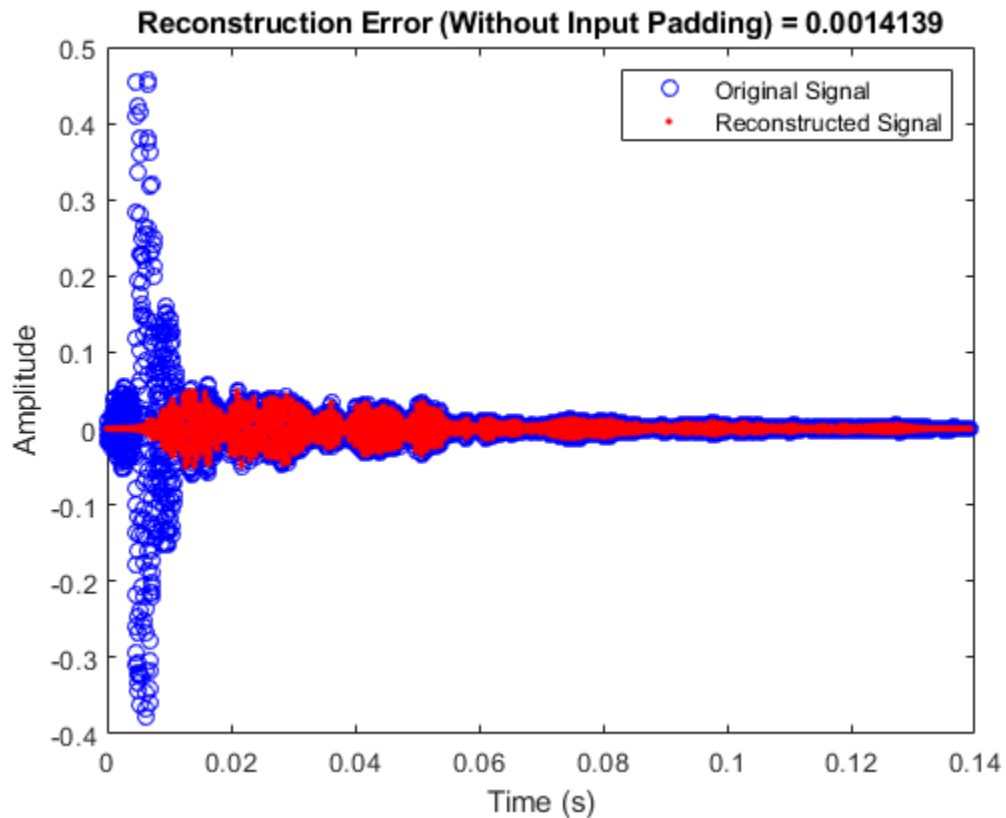
```
C = mdct(xClipped,win);  
y = imdct(C,win);  
  
figure(1)  
t = (0:size(xClipped,1)-1)/fs;  
plot(t,xClipped,'bo',t,y,'r.')  
legend('Original Signal','Reconstructed Signal')  
title(strcat("Reconstruction Error = ",num2str(mean((xClipped-y).^2))))  
xlabel('Time (s)')  
ylabel('Amplitude')
```



You can perform the MDCT and IMDCT without input padding using the `PadInput` name-value pair. However, there will be a reconstruction error in the first half-frame and last half-frame of the signal.

```
C = mdct(xClipped,win,'PadInput',false);
y = imdct(C,win,'PadInput',false);
```

```
figure(2)
t = (0:size(xClipped,1)-1)/fs;
plot(t,xClipped,'bo',t,y,'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error (Without Input Padding) = ",num2str(mean((xClipped-y)
xlabel('Time (s)')
ylabel('Amplitude')
```



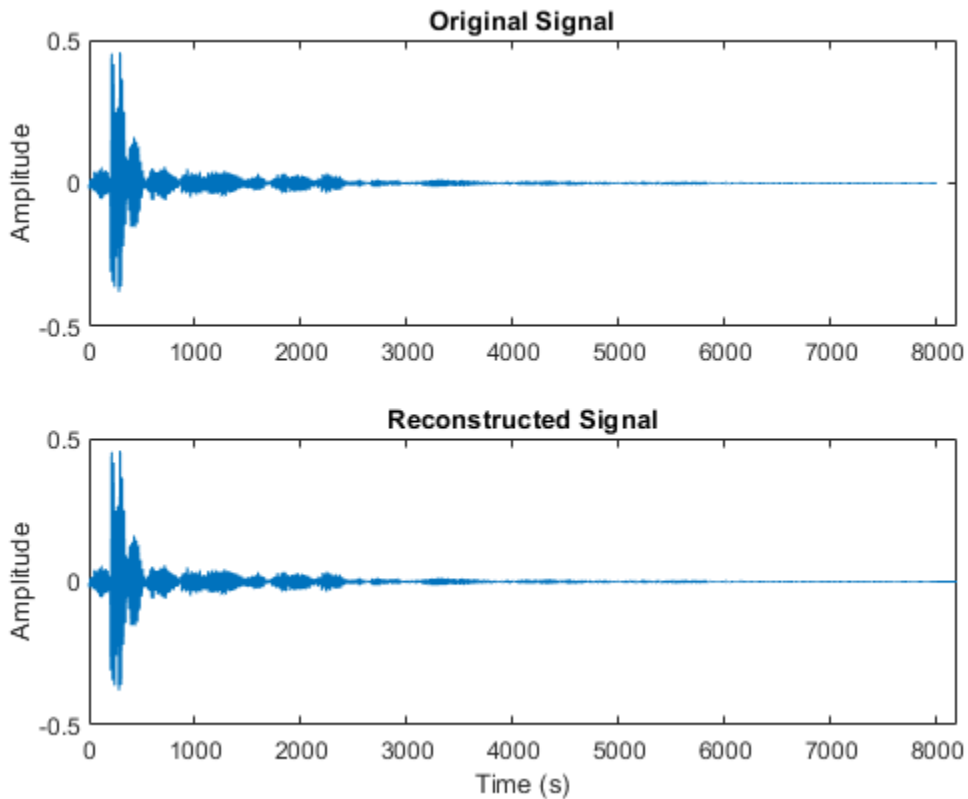
If you specify an input signal to the `mdct` that is not a multiple of the window length, then the input signal is padded with zeros. Pass the original unclipped signal through the transform pair and compare the original signal and the reconstructed signal.

```
C = mdct(x,win);
y = imdct(C,win);
```

```
figure(3)
```

```
subplot(2,1,1)
plot(x)
title('Original Signal')
ylabel('Amplitude')
axis([0,max(size(y,1),size(x,1)),-0.5,0.5])
```

```
subplot(2,1,2)
plot(y)
title('Reconstructed Signal')
xlabel('Time (s)')
ylabel('Amplitude')
axis([0,max(size(y,1),size(x,1)),-0.5,0.5])
```



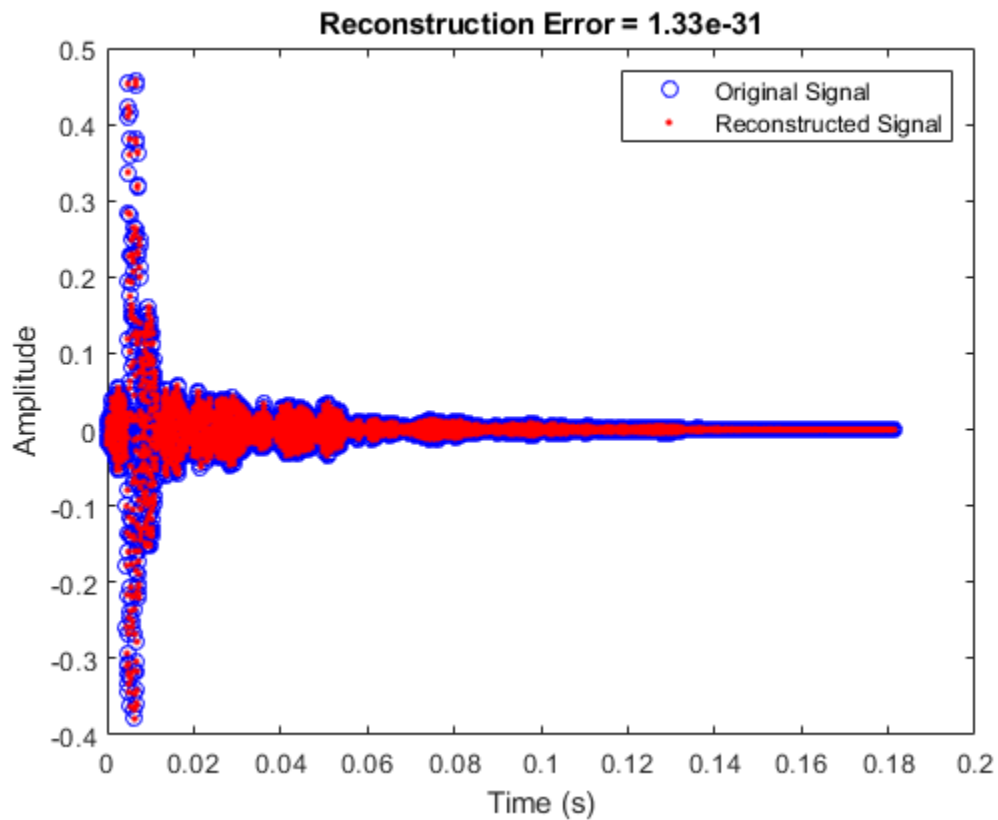
The reconstructed signal is padded with zeros at the back end. Remove the zero-padding from the reconstructed signal, plot the original and reconstructed signal, and then display the reconstruction error.

```
figure(4)
y = y(1:size(x,1));
```

```

t = (0:size(x,1)-1)/fs;
plot(t,x,'bo',t,y,'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error = ",num2str(mean((x-y).^2))))
xlabel('Time (s)')
ylabel('Amplitude')

```



### MDCT and IMDCT for Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the reconstructed signal for comparison. Create a `dsp.AsyncBuffer` to buffer the input stream.

```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3');  
logger = dsp.SignalSink;  
buff = dsp.AsyncBuffer;
```

Create a 512-point Kaiser-Bessel-derived window.

```
N = 512;  
win = kbdwin(N);
```

In an audio stream loop:

- 1 Read a frame of data from the file.
- 2 Write the frame of data to the async buffer.
- 3 If half a frame of data is present, read from the buffer and then perform the transform pair. Overlap-add the current output from `imdct` with the previous output, and log the results. Update the memory.

```
mem = zeros(N/2,2); % initialize an empty memory
```

```
while ~isDone(fileReader)  
    audioIn = fileReader();  
    write(buff,audioIn);
```

```
    while buff.NumUnreadSamples >= N/2  
        x = read(buff,N,N/2);  
        C = mdct(x,win,'PadInput',false);  
        y = imdct(C,win,'PadInput',false);
```

```
        logger(y(1:N/2,)+mem)  
        mem = y(N/2+1:end,:);
```

```
    end
```

```
end
```

```
% Perform the transform pair one last time with a zero-padded final signal.
```

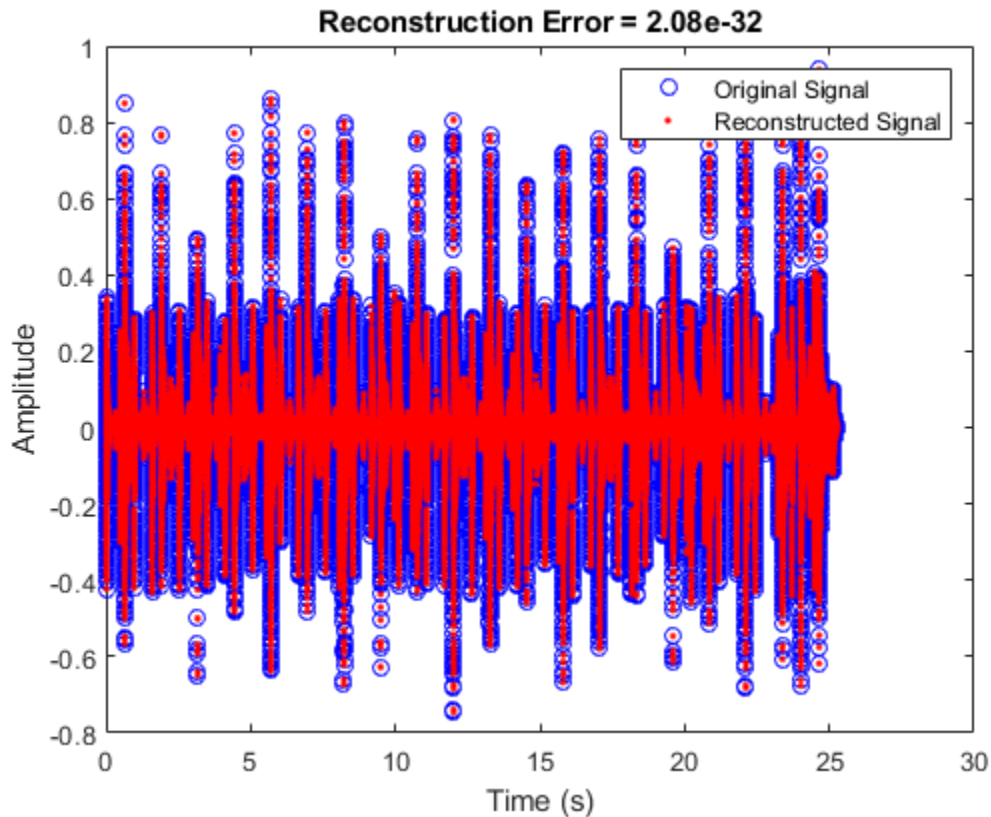
```
x = read(buff,N,N/2);  
C = mdct(x,win,'PadInput',false);  
y = imdct(C,win,'PadInput',false);  
logger(y(1:N/2,)+mem)
```

```
reconstructedSignal = logger.Buffer;
```

Read in the entire original audio signal. Trim the front and back zero padding from the reconstructed signal for comparison. Plot one channel of the original and reconstructed signals and display the reconstruction error.

```
[originalSignal,fs] = audioread(fileReader.FileName);
signalLength = size(originalSignal,1);
reconstructedSignal = reconstructedSignal((N/2+1):(N/2+1)+signalLength-1,:);

t = (0:size(originalSignal,1)-1)/fs;
plot(t,originalSignal(:,1),'bo',t,reconstructedSignal(:,1),'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error = ", ...
            num2str(mean((originalSignal-reconstructedSignal).^2,'all'))))
xlabel('Time (s)')
ylabel('Amplitude')
```



## Input Arguments

### **X** — Input array

column vector | matrix

Input array, specified as a column vector or matrix. If specified as a matrix, the columns are treated as independent audio channels.

Data Types: `single` | `double`

### **win** — Window applied in time domain

even-length vector



Window applied in the time domain, specified as an even-length vector. The transform performed by `mdct` has the same number of points as `win`. To enable perfect reconstruction, use a window that satisfies the Princen-Bradley condition ( $w_n^2 + w_{n+N}^2 = 1$ ), such as a sine window or `kbdwin`.

Data Types: `single` | `double`

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'PadInput', false`

### **PadInput** — Flag to pad input array

`true` (default) | `false`

Flag to pad input array, specified as the comma-separated pair consisting of `'PadInput'` and `true` or `false`. If set to `true`, zero-padding is added to the input `X` at both ends to enable perfect reconstruction. The number of zeros at each end is `numel(win)/2`.

Data Types: `logical`

## Output Arguments

### **Y** — Modified discrete cosine transform

vector | matrix | 3-D array

Modified discrete cosine transform (MDCT), returned as a vector, matrix, or 3-D array. The dimensions of `Y` are `L-by-M-by-N`, where:

- `L` -- Number of points in the frequency-domain representation of each frame, equal to `numel(win)/2`.
- `M` -- Number of frames the input array is partitioned into.
  - If `PadInput` is set to `true`, `M = ceil(2*size(X,1)/numel(win))+1`.
  - If `PadInput` is set to `false`, `M = ceil(2*size(X,1)/numel(win))-1`.

- $N$  -- Number of channels, equal to `size(X,2)`.

Trailing singleton dimensions are removed from the output  $Y$ .

Data Types: `single` | `double`

### **S — Modified discrete sine transform**

vector | matrix | 3-D array

Modified discrete sine transform (MDST), returned as a vector, matrix, or 3-D array. The dimensions of  $S$  are the same as the MDCT output,  $Y$ .

Data Types: `single` | `double`

### **Z — Half-sided odd discrete Fourier transform**

vector | matrix | 3-D array

Half-sided odd discrete Fourier transform (ODFT), returned as a vector, matrix, or 3-D array of complex numbers. The dimensions of  $Z$  are the same as the MDCT output,  $Y$ .

To construct the complete (two-sided) ODFT, mirror the half-sided ODFT:

```
cat(1,Z,conj(flip(Z,1))).
```

Data Types: `single` | `double`

Complex Number Support: Yes

## Algorithms

The modified discrete cosine transform is a time-frequency transform. Given an input signal  $X$  and window  $win$ , the `mdct` function performs the following steps for each independent channel:

- 1 The frame size is the number of elements in the specified window,  $N = \text{numel}(win)$ . By default, `PadInput` is set to `true`, so the input signal  $X$  is padded with  $N/2$  zeros on the front and back. If the input signal is not divisible by  $N$ , additional padding is added on the back. After padding, the input signal is buffered into 50% overlapped frames.
- 2 Each frame of the buffered and padded input signal is multiplied by the window,  $win$ .
- 3 The input is converted into a frequency representation using the modified discrete cosine transform:

$$Y(k) = \sum_{n=0}^{N-1} X(n) \cos \left[ \frac{\pi}{(N/2)} \left( n + \frac{(N/2) + 1}{2} \right) \left( k + \frac{1}{2} \right) \right], \quad k = 0, 1, \dots, (N/2) - 1$$

To take advantage of the FFT algorithm, the MDCT is calculated by first calculating the odd DFT:

$$Y_o(k) = \sum_{n=0}^{N-1} X(n) e^{-j \frac{\pi n}{N} (2k+1)}, \quad k = 0, 1, \dots, N-1$$

and then calculating the MDCT:

$$Y(k) = \Re \left\{ Y_o(k) \right\} \cos \left( \frac{\pi}{N} \left( k + \frac{1}{2} \right) \left( 1 + \frac{N}{2} \right) \right), \quad k = 0, 1, \dots, (N/2) - 1$$

If a second argument is requested from the `mdct` function, the modified discrete sine transform (MDST) is also computed and returned:

$$X(k) = \Im \left\{ X_o(k) \right\} \sin \left( \frac{\pi}{N} \left( k + \frac{1}{2} \right) \left( 1 + \frac{N}{2} \right) \right), \quad k = 0, 1, \dots, (N/2) - 1$$

## References

- [1] Princen, J., A. Johnson, and A. Bradley. "Subband/Transform Coding Using Filter Bank Designs Based on Time Domain Aliasing Cancellation." *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 1987, pp. 2161-2164.
- [2] Princen, J., and A. Bradley. "Analysis/Synthesis Filter Bank Design Based on Time Domain Aliasing Cancellation." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 34, Issue 5, 1986, pp. 1153-1161.

## Extended Capabilities

### C/C++ Code Generation

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

## **See Also**

`imdct` | `kbdwin` | `spectrogram`

**Introduced in R2019a**

# imdct

Inverse modified discrete cosine transform

## Syntax

```
X = imdct(Y,win)
X = imdct(Y,win,Name,Value)
```

## Description

`X = imdct(Y,win)` returns the inverse modified discrete cosine transform (IMDCT) of `Y`, followed by multiplication with time window `win` and overlap-addition of the frames with 50% overlap.

`X = imdct(Y,win,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

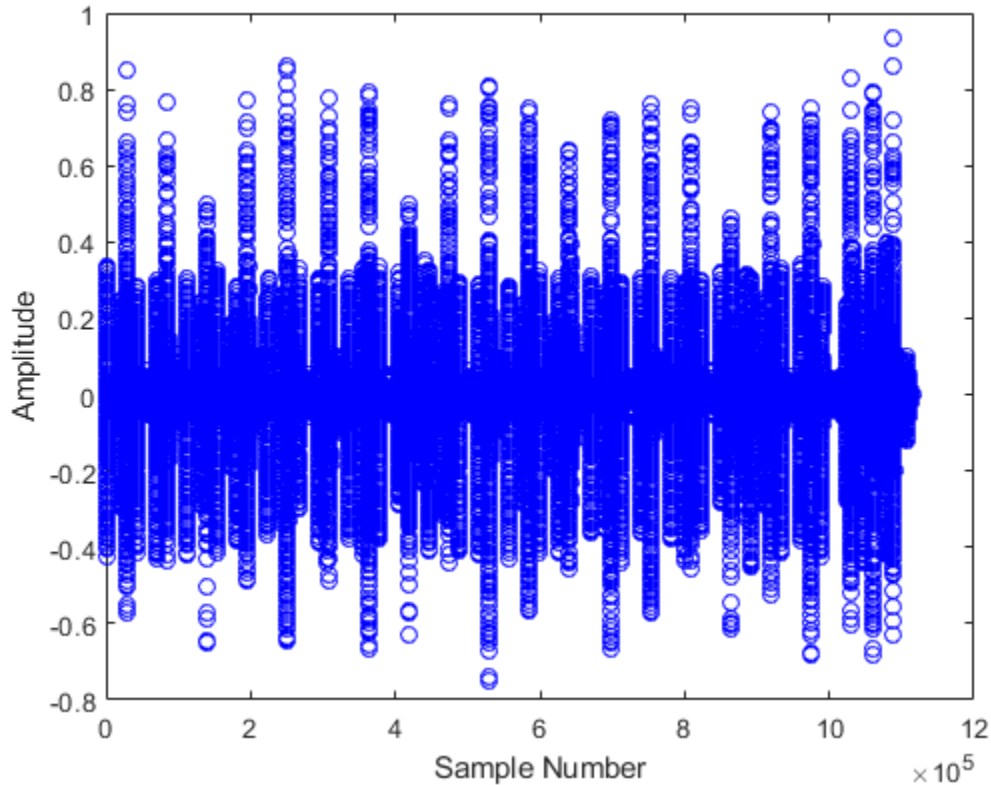
## Examples

### Calculate IMDCT

Read in an audio file, convert it to mono, and then plot it.

```
audioIn = audioread('FunkyDrums-44p1-stereo-25secs.mp3');
audioIn = mean(audioIn,2);

figure(1)
plot(audioIn,'bo')
ylabel('Amplitude')
xlabel('Sample Number')
```



Calculate the MDCT using a 4096-point sine window. Plot the power of the MDCT coefficients over time.

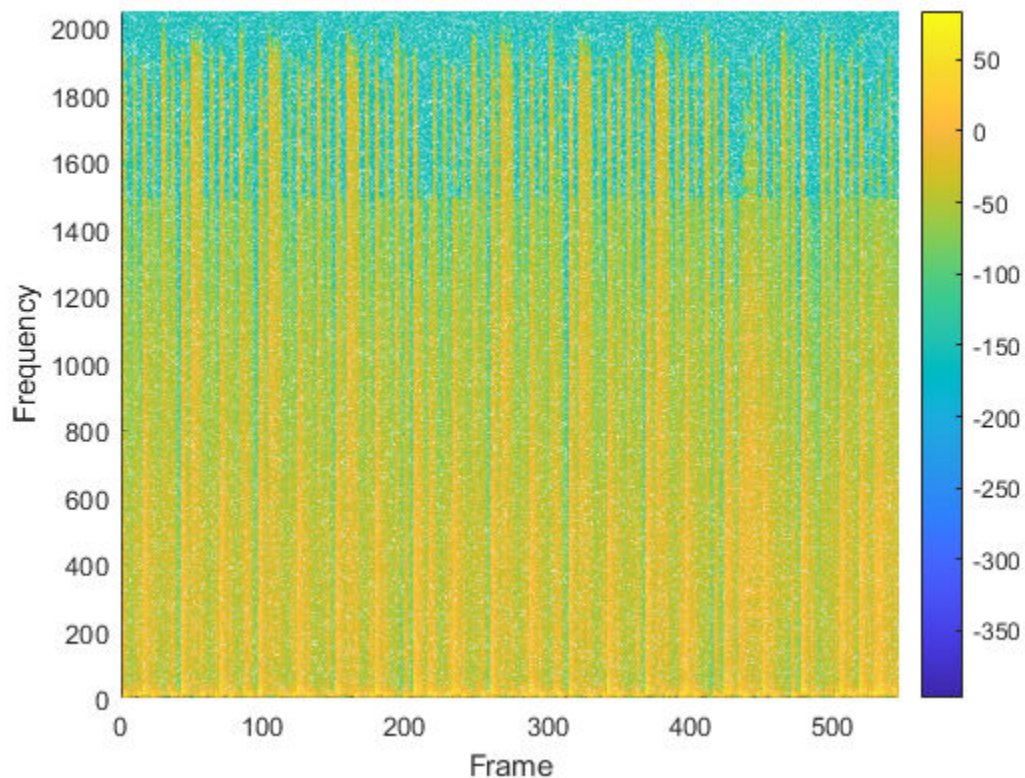
```

N = 4096;
wdw = sin(pi*((1:N)-0.5)/N);

C = mdct(audioIn,wdw);

figure(2)
surf(20*log10(C.*conj(C)), 'EdgeColor', 'none');
view([0 90])
xlabel('Frame')
ylabel('Frequency')
    
```

```
axis([0 size(C,2) 0 size(C,1)])  
colorbar
```

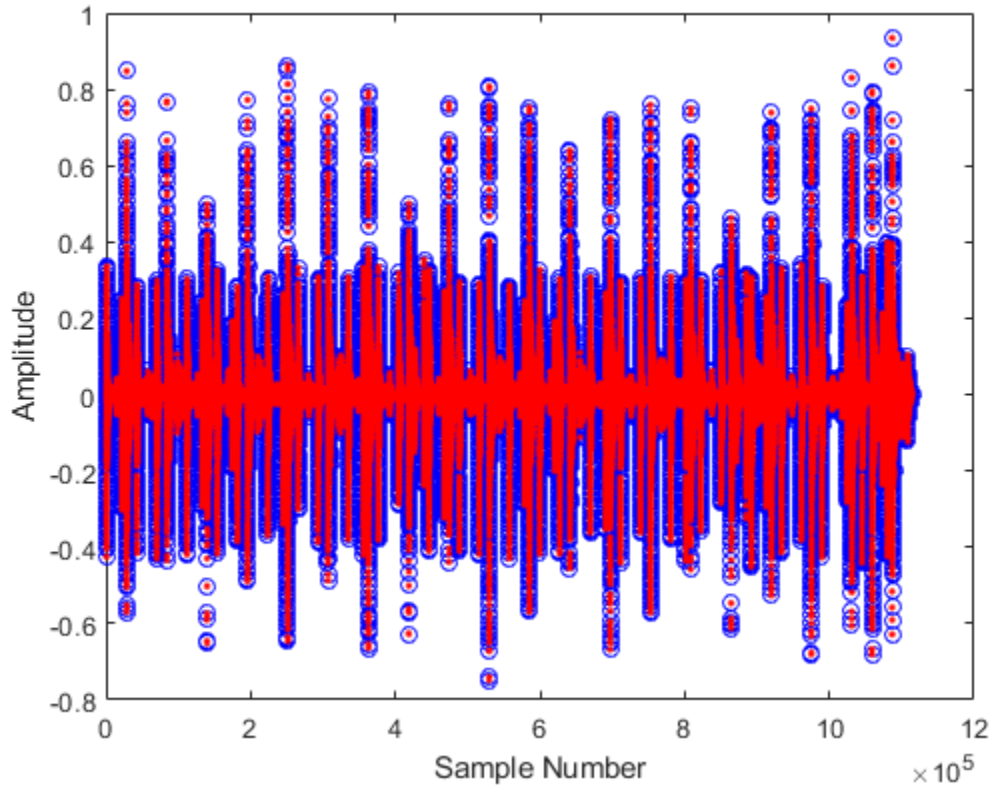


Transform the representation back to the time domain. Verify the perfect reconstruction property by computing the mean squared error. Plot the reconstructed signal over the original signal.

```
audioReconstructed = imdct(C,wdw);  
err = mean((audioIn-audioReconstructed(1:size(audioIn,1),:)).^2)  
  
err = 9.5889e-31
```

```
figure(1)  
hold on
```

```
plot(audioReconstructed,'r.')  
ylabel('Amplitude')  
xlabel('Sample Number')
```



### Effect of Input Padding on Perfect Reconstruction

To enable perfect reconstruction, the `mdct` function zero-pads the front and back of the audio input signal. The signal returned from `imdct` removes the zero padding added for perfect reconstruction.



Read in an audio file, create a 2048-point Kaiser-Bessel-derived window, and then clip the audio signal so that its length is a multiple of 2048.

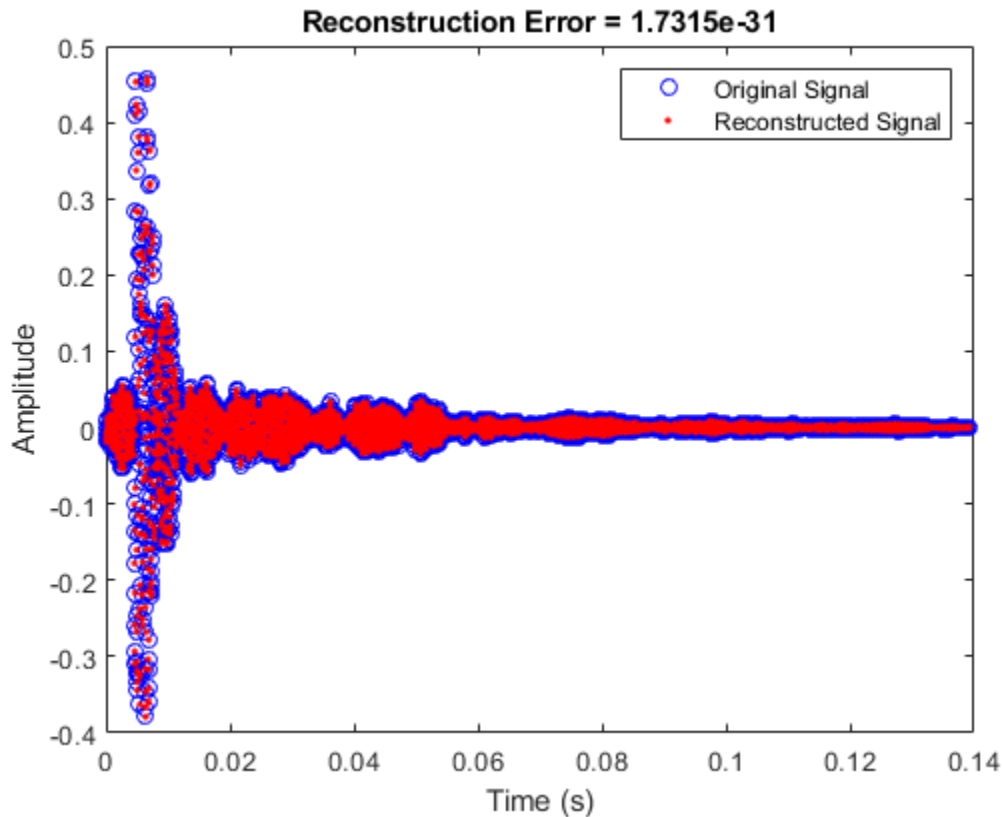
```
[x,fs] = audioread('Click-16-44p1-mono-0.2secs.wav');  
win = kbdwin(2048);
```

```
xClipped = x(1:end - rem(size(x,1),numel(win)));
```

Convert the signal to the frequency domain, and then reconstruct it back in the time domain. Plot the original and reconstructed signals and display the reconstruction error.

```
C = mdct(xClipped,win);  
y = imdct(C,win);
```

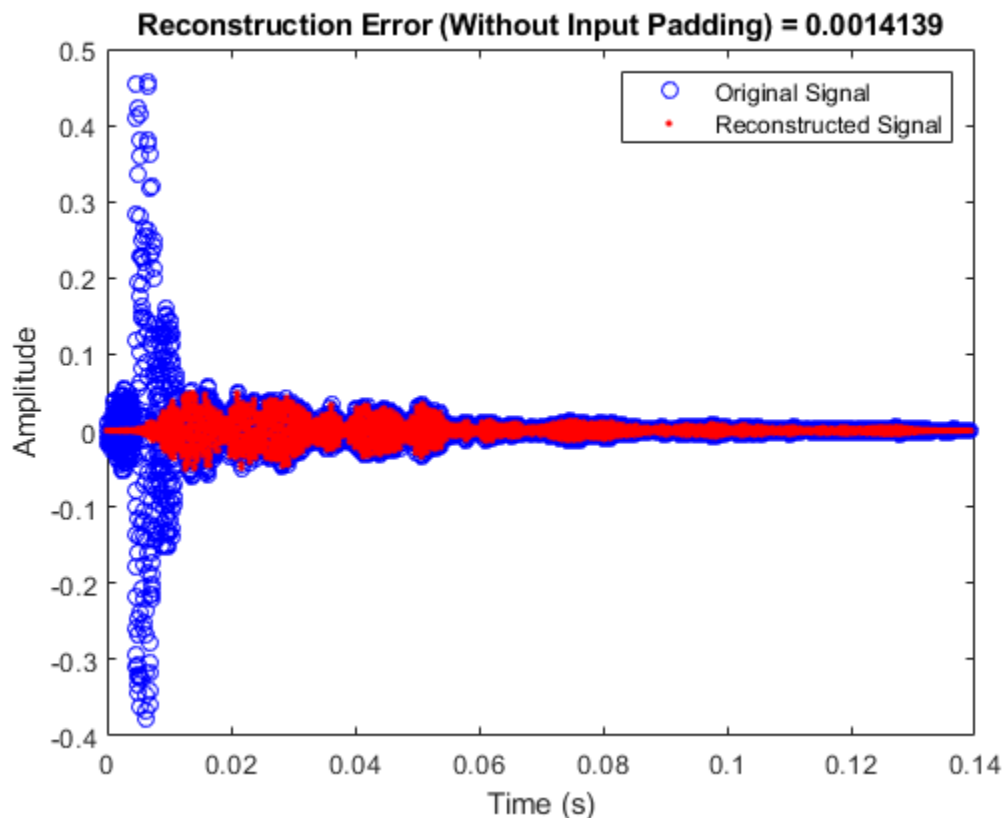
```
figure(1)  
t = (0:size(xClipped,1)-1)/fs;  
plot(t,xClipped,'bo',t,y,'r.')  
legend('Original Signal','Reconstructed Signal')  
title(strcat("Reconstruction Error = ",num2str(mean((xClipped-y).^2))))  
xlabel('Time (s)')  
ylabel('Amplitude')
```



You can perform the MDCT and IMDCT without input padding using the `PadInput` name-value pair. However, there will be a reconstruction error in the first half-frame and last half-frame of the signal.

```
C = mdct(xClipped,win,'PadInput',false);
y = imdct(C,win,'PadInput',false);
```

```
figure(2)
t = (0:size(xClipped,1)-1)/fs;
plot(t,xClipped,'bo',t,y,'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error (Without Input Padding) = ",num2str(mean((xClipped-y)
xlabel('Time (s)')
ylabel('Amplitude')
```



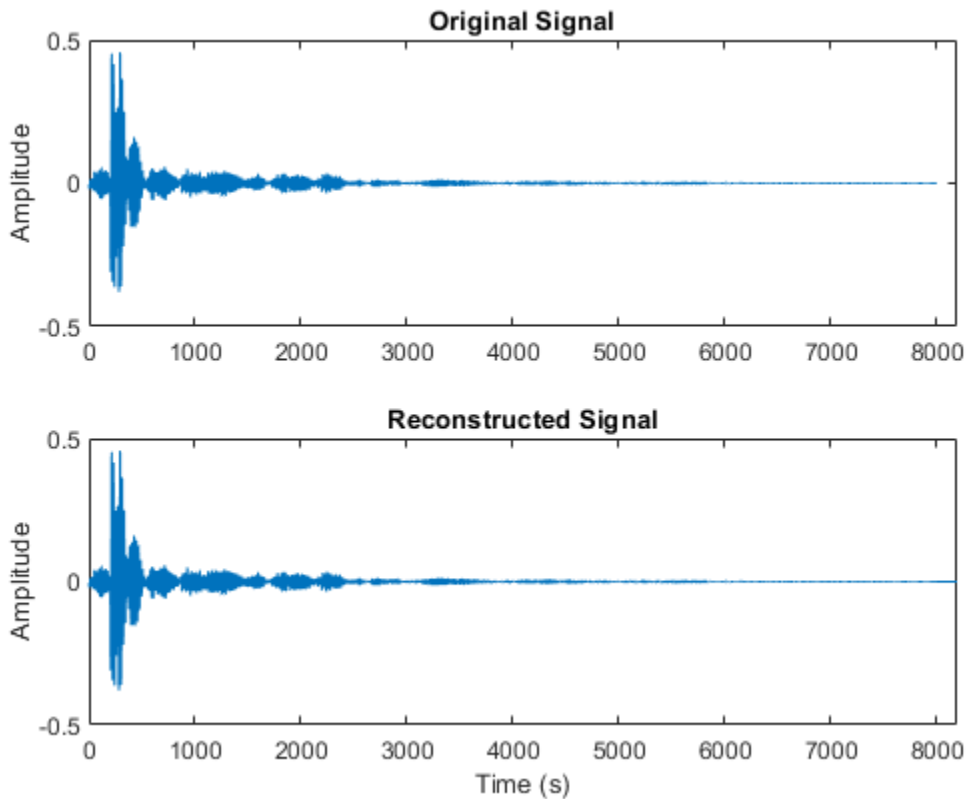
If you specify an input signal to the `mdct` that is not a multiple of the window length, then the input signal is padded with zeros. Pass the original unclipped signal through the transform pair and compare the original signal and the reconstructed signal.

```
C = mdct(x,win);
y = imdct(C,win);
```

```
figure(3)
```

```
subplot(2,1,1)
plot(x)
title('Original Signal')
ylabel('Amplitude')
axis([0,max(size(y,1),size(x,1)),-0.5,0.5])
```

```
subplot(2,1,2)
plot(y)
title('Reconstructed Signal')
xlabel('Time (s)')
ylabel('Amplitude')
axis([0,max(size(y,1),size(x,1)),-0.5,0.5])
```



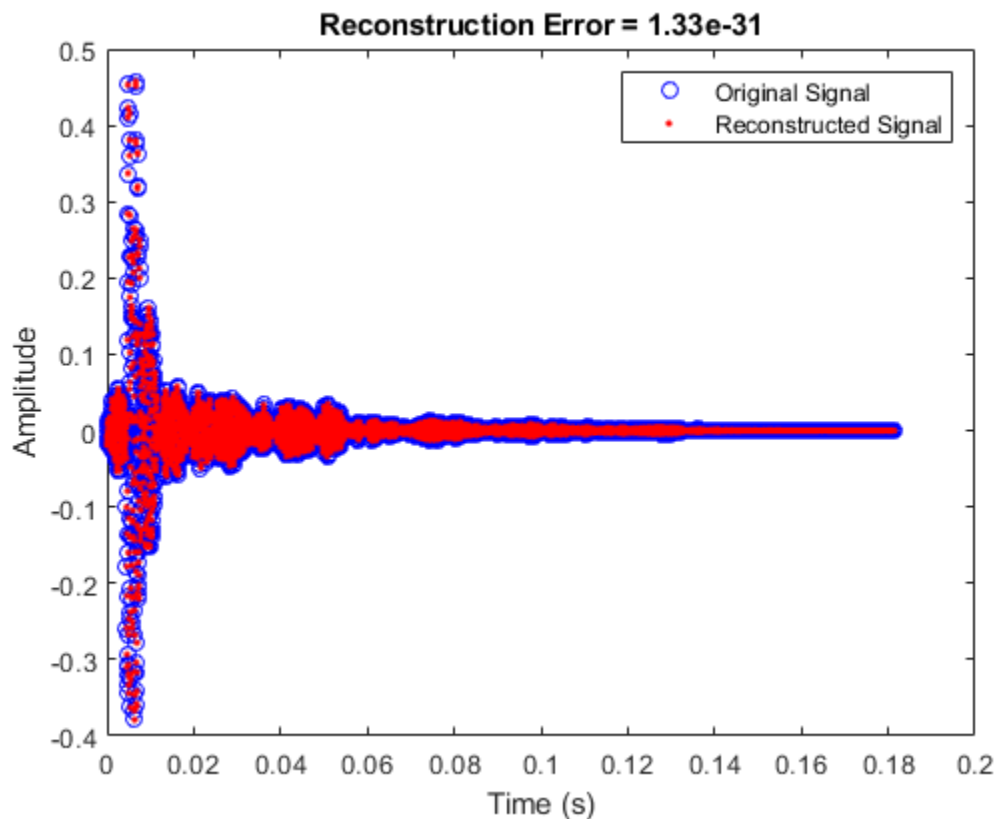
The reconstructed signal is padded with zeros at the back end. Remove the zero-padding from the reconstructed signal, plot the original and reconstructed signal, and then display the reconstruction error.

```
figure(4)
y = y(1:size(x,1));
```

```

t = (0:size(x,1)-1)/fs;
plot(t,x,'bo',t,y,'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error = ",num2str(mean((x-y).^2))))
xlabel('Time (s)')
ylabel('Amplitude')

```



### MDCT and IMDCT for Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the reconstructed signal for comparison. Create a `dsp.AsyncBuffer` to buffer the input stream.

```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3');  
logger = dsp.SignalSink;  
buff = dsp.AsyncBuffer;
```

Create a 512-point Kaiser-Bessel-derived window.

```
N = 512;  
win = kbdwin(N);
```

In an audio stream loop:

- 1 Read a frame of data from the file.
- 2 Write the frame of data to the async buffer.
- 3 If half a frame of data is present, read from the buffer and then perform the transform pair. Overlap-add the current output from `imdct` with the previous output, and log the results. Update the memory.

```
mem = zeros(N/2,2); % initialize an empty memory
```

```
while ~isDone(fileReader)  
    audioIn = fileReader();  
    write(buff,audioIn);
```

```
    while buff.NumUnreadSamples >= N/2  
        x = read(buff,N,N/2);  
        C = mdct(x,win,'PadInput',false);  
        y = imdct(C,win,'PadInput',false);
```

```
        logger(y(1:N/2,)+mem)  
        mem = y(N/2+1:end,:);
```

```
    end
```

```
end
```

```
% Perform the transform pair one last time with a zero-padded final signal.
```

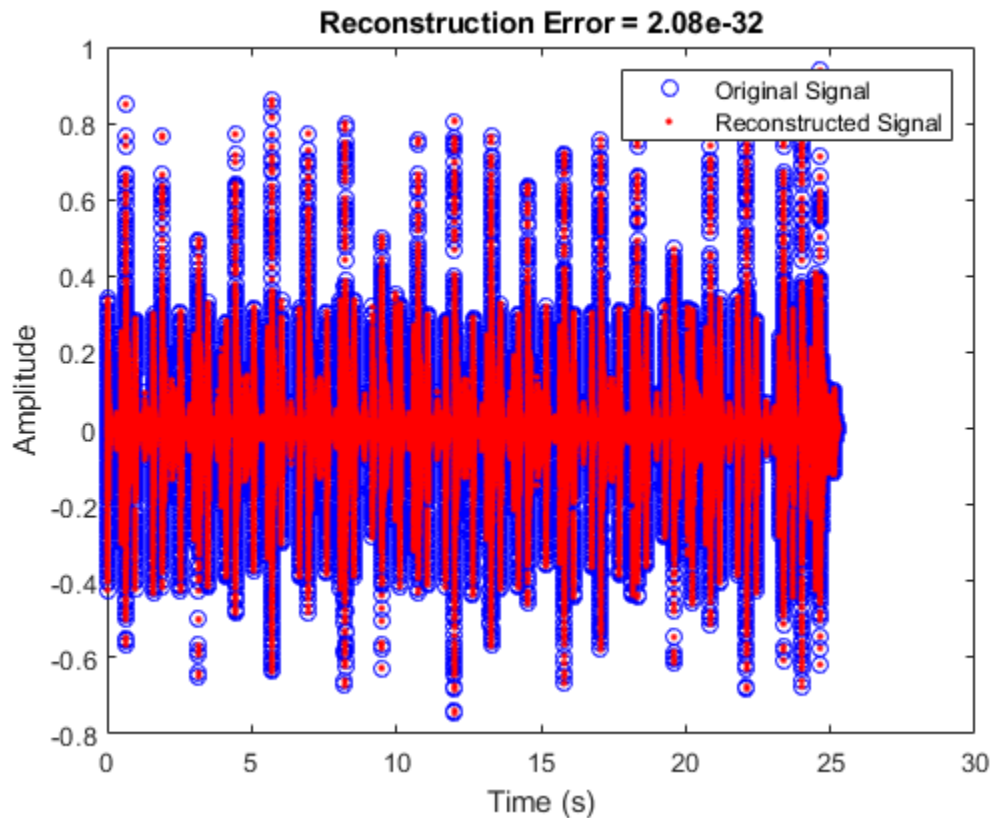
```
x = read(buff,N,N/2);  
C = mdct(x,win,'PadInput',false);  
y = imdct(C,win,'PadInput',false);  
logger(y(1:N/2,)+mem)
```

```
reconstructedSignal = logger.Buffer;
```

Read in the entire original audio signal. Trim the front and back zero padding from the reconstructed signal for comparison. Plot one channel of the original and reconstructed signals and display the reconstruction error.

```
[originalSignal,fs] = audioread(fileReader.FileName);
signalLength = size(originalSignal,1);
reconstructedSignal = reconstructedSignal((N/2+1):(N/2+1)+signalLength-1,:);

t = (0:size(originalSignal,1)-1)/fs;
plot(t,originalSignal(:,1),'bo',t,reconstructedSignal(:,1),'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error = ", ...
            num2str(mean((originalSignal-reconstructedSignal).^2,'all'))))
xlabel('Time (s)')
ylabel('Amplitude')
```



## Input Arguments

### **Y** — Modified discrete cosine transform

vector | matrix | 3-D array

Modified discrete cosine transform (MDCT), specified as a vector, matrix, or 3-D array. The dimensions of **Y** are interpreted as output from the `mdct` function. If **Y** is an  $L$ -by- $M$ -by- $N$  array, the dimensions are interpreted as:

- $L$  — Number of points in the frequency-domain representation of each frame.  $L$  must be half the number of points in the window, `win`.



- *M* -- Number of frames.
- *N* -- Number of channels.

Data Types: `single` | `double`

### **win** — Window applied in time domain

vector

Window applied in the time domain, specified as vector. The length of `win` must be twice the number of rows of `Y`: `numel(win)==2*size(Y,1)`. To enable perfect reconstruction, use the same window used in the forward transformation `mdct`.

Data Types: `single` | `double`

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'PadInput',false`

### **PadInput** — Flag if input was padded

`true` (default) | `false`

Flag if input to the forward `mdct` was padded. If set to `true`, the output is truncated at both ends to remove the zero-padding that the forward `mdct` added.

Data Types: `logical`

## **Output Arguments**

### **X** — Inverse modified discrete cosine transform

column vector | matrix

Inverse modified discrete cosine transform (IMDCT) of input array `Y`, returned as a column vector or matrix of independent channels.

Data Types: `single` | `double`

## Algorithms

The inverse modified discrete cosine transform is a time-frequency transform. Given a frequency domain input signal  $Y$  and window  $\text{win}$ , the `imdct` function performs the follows steps for each independent channel:

- 1 Each frame of the input is converted into a time-domain representation:

$$X(n) = \sum_{k=0}^{\frac{N}{2}-1} Y(k) \cos \left[ \frac{\pi}{(N/2)} \left( n + \frac{(N/2)+1}{2} \right) \left( k + \frac{1}{2} \right) \right], \quad n = 0, 1, \dots, N-1$$

where  $N$  is the number of elements in  $\text{win}$ .

- 2 Each frame of the time-domain signal is multiplied by the window,  $\text{win}$ .
- 3 The frames are overlap-added with 50% overlap to construct a contiguous time-domain signal. If `PadInput` is set to `true`, the `imdct` function assumes the original input signal in the forward transform (`mdct`) was padded with  $N/2$  zeros on the front and back and removes the padding. By default, `PadInput` is set to `true`.

## References

- [1] Princen, J., A. Johnson, and A. Bradley. "Subband/Transform Coding Using Filter Bank Designs Based on Time Domain Aliasing Cancellation." *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 1987, pp. 2161-2164.
- [2] Princen, J., and A. Bradley. "Analysis/Synthesis Filter Bank Design Based on Time Domain Aliasing Cancellation." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 34, Issue 5, 1986, pp. 1153-1161.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

kbdwin | mdct | spectrogram

**Introduced in R2019a**

## harmonicRatio

Harmonic ratio

### Syntax

```
hr = harmonicRatio(audioIn,fs)
hr = harmonicRatio(audioIn,fs,Name,Value)
```

### Description

`hr = harmonicRatio(audioIn,fs)` returns the harmonic ratio of the signal, `audioIn`, over time. Columns of the input are treated as individual channels.

`hr = harmonicRatio(audioIn,fs,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

Example: `hr = harmonicRatio(audioIn,fs,'Window',rectwin(round(fs*0.1)),'OverlapLength',round(fs*0.05))` returns the harmonic ratio for the audio input signal sampled at `fs` Hz. The harmonic ratio is calculated for 100 ms rectangular windows with 50 ms overlap.

### Examples

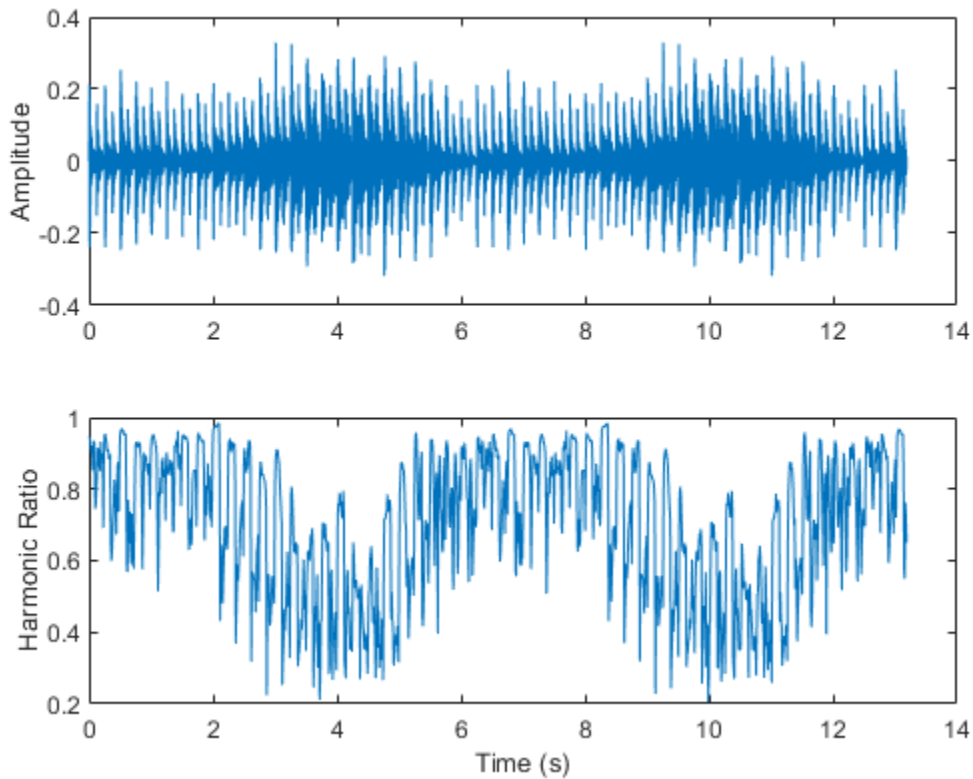
#### Calculate Harmonic Ratio

Read in an audio file, calculate the harmonic ratio using default parameters, and then plot the results.

```
[audioIn,fs] = audioread('RandomOscThree-24-96-stereo-13secs.aif');
audioInMono = mean(audioIn,2);
```

```
hr = harmonicRatio(audioInMono,fs);
```

```
t = (0:length(audioInMono)-1)/fs;  
subplot(2,1,1)  
plot(t, audioInMono)  
ylabel('Amplitude')  
  
t = linspace(0, size(audioInMono,1)/fs, size(hr,1));  
subplot(2,1,2)  
plot(t, hr)  
xlabel('Time (s)')  
ylabel('Harmonic Ratio')
```



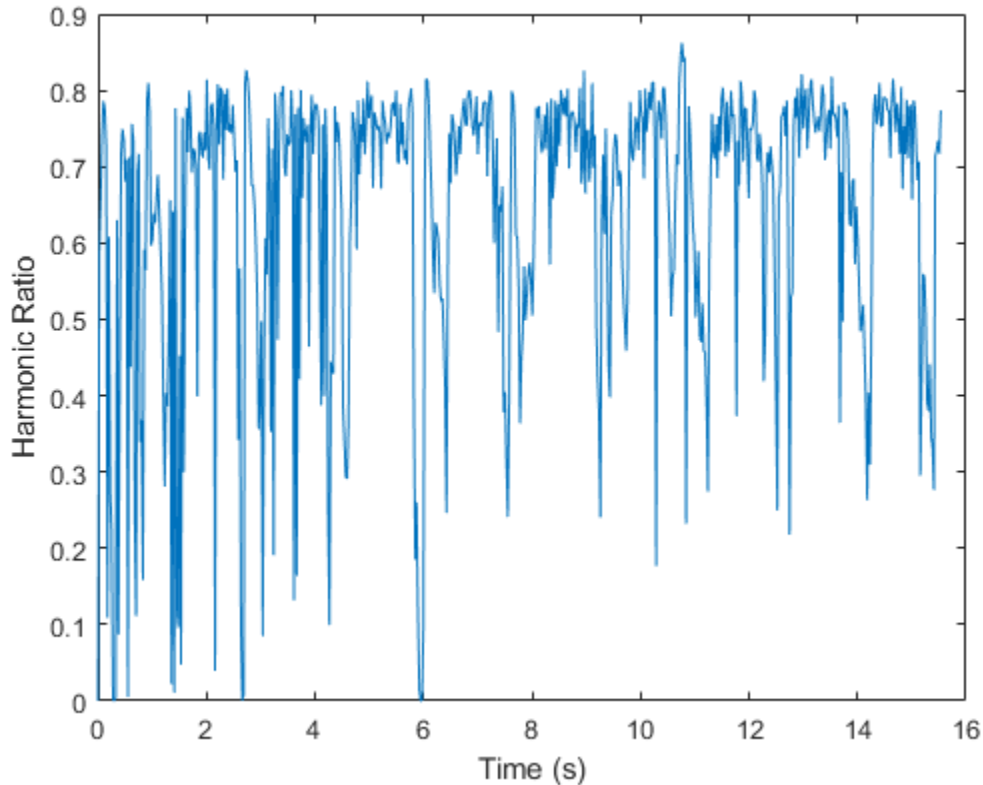
### Specify Nondefault Parameters

Read in an audio file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Calculate the harmonic ratio of the audio file using 50 ms Hann windows with 25 ms overlap. Plot the results.

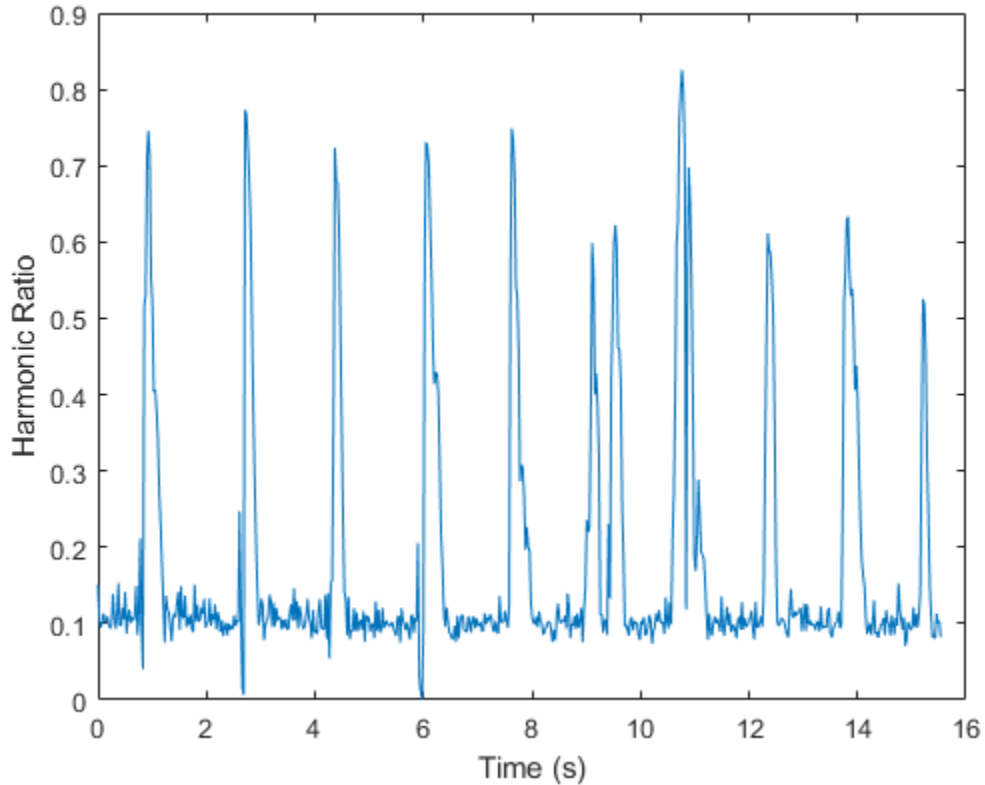
```
hr = harmonicRatio(audioIn,fs, ...  
                  'Window',hann(round(fs.*0.05),'periodic'), ...  
                  'OverlapLength',round(fs.*0.025));  
  
t = linspace(0,size(audioIn,1)/fs,size(hr,1));  
plot(t,hr)  
xlabel('Time (s)')  
ylabel('Harmonic Ratio')
```



The harmonic ratio indicates the ratio of energy in the harmonic portion of audio to the total energy of the audio. Because the audio signal in this example has regions of near silence, where the total energy is very low, the harmonic ratio does a poor job discriminating between regions of speech and regions of silence. Add white noise to the audio signal and then calculate and plot the harmonic ratio.

```
audioIn = audioIn + 0.1*randn(size(audioIn));  
hr = harmonicRatio(audioIn,fs, ...  
                  'Window',hann(round(fs.*0.05),'periodic'), ...  
                  'OverlapLength',round(fs.*0.025));  
  
t = linspace(0,size(audioIn,1)/fs,size(hr,1));  
plot(t,hr)
```

```
xlabel('Time (s)')  
ylabel('Harmonic Ratio')
```



### Calculate Harmonic Ratio of Streaming Audio

Create a `dsp.AudioFileReader` object to read in stereo audio data frame-by-frame.  
Create a `dsp.SignalSink` object to log the harmonic ratio calculation.

```
fileReader = dsp.AudioFileReader('RandomOscThree-24-96-stereo-13secs.aif');  
logger = dsp.SignalSink;
```

In an audio stream loop:



- 1 Read in a frame of audio data.
- 2 Calculate the harmonic ratio for each channel of the frame of audio.
- 3 Log the harmonic ratio for later plotting.

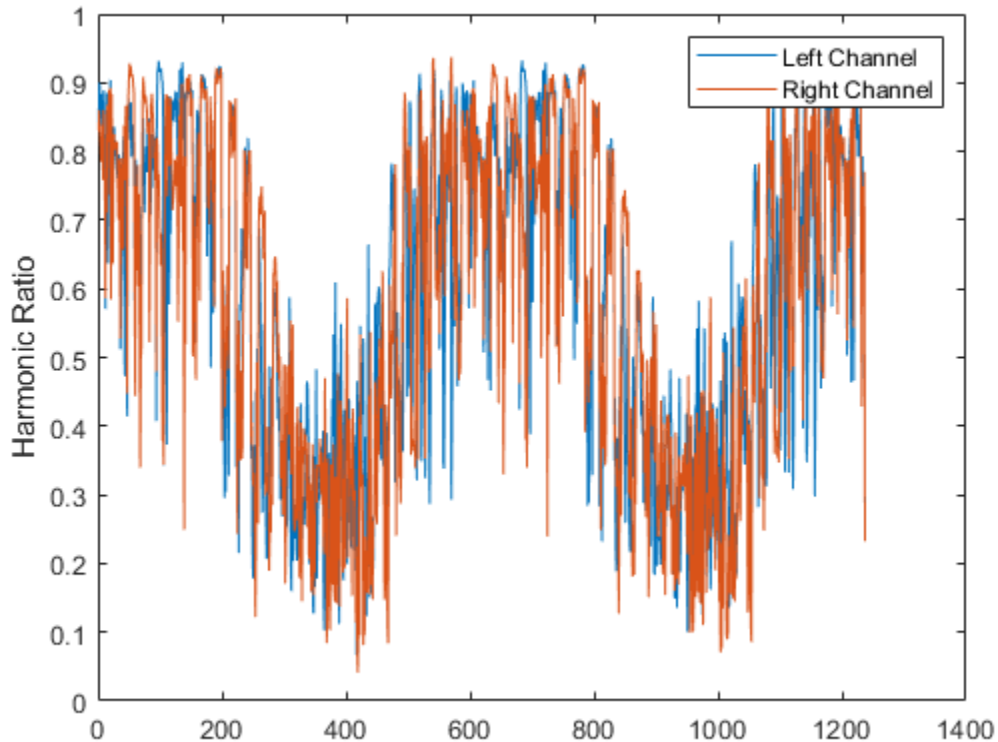
To calculate the harmonic ratio for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame, 'periodic');
while ~isDone(fileReader)
    audioIn = fileReader();

    hr = harmonicRatio(audioIn, fileReader.SampleRate, ...
        'Window', win, ...
        'OverlapLength', 0);

    logger(hr)
end

plot(logger.Buffer)
ylabel('Harmonic Ratio')
legend('Left Channel', 'Right Channel')
```



If the input to your audio stream loop has a variable samples-per-frame, an inconsistent samples-per-frame with the analysis window size of `harmonicRatio`, or if you want to calculate the harmonic ratio of overlapped data, use `dsp.AsyncBuffer`.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;  
reset(logger)  
release(fileReader)
```

Calculate the harmonic ratio using 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;
```

```
samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff, audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff, samplesPerFrame, samplesOverlap);

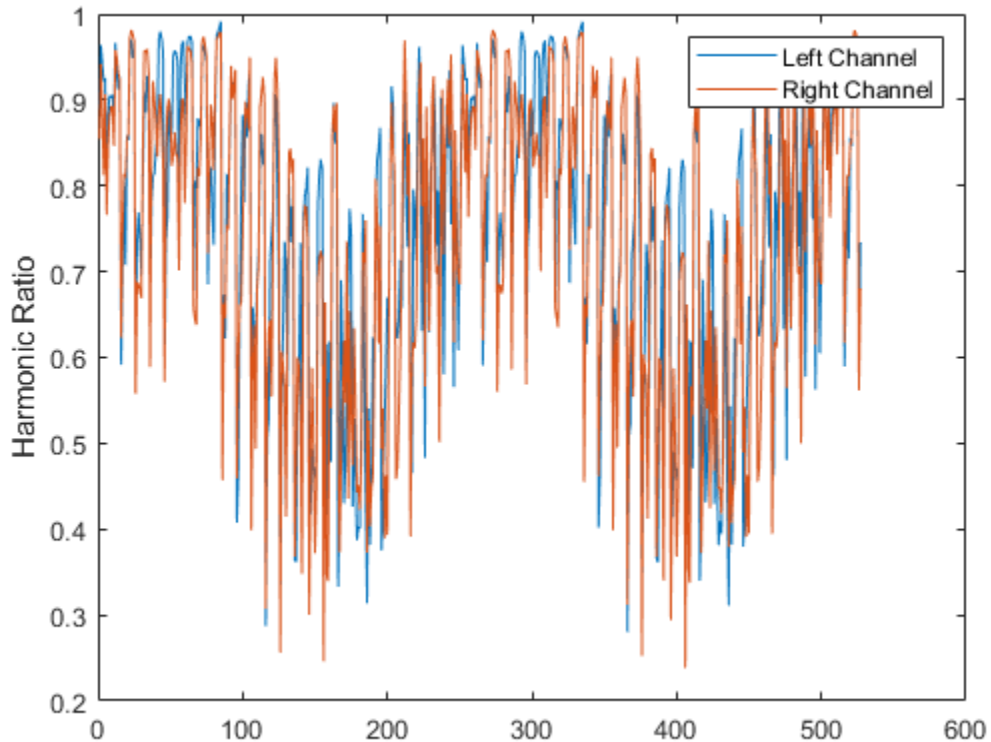
        hr = harmonicRatio(audioBuffered, fs, ...
                           'Window', win, ...
                           'OverlapLength', 0);

        logger(hr)
    end
end

release(fileReader)

Plot the logged data.

plot(logger.Buffer)
ylabel('Harmonic Ratio')
legend('Left Channel', 'Right Channel')
```



### **Harmonic Ratio of Tones and White Noise**

The harmonic ratio measures the amount of energy in the tonal part of the signal compared to the amount of energy in the total signal.

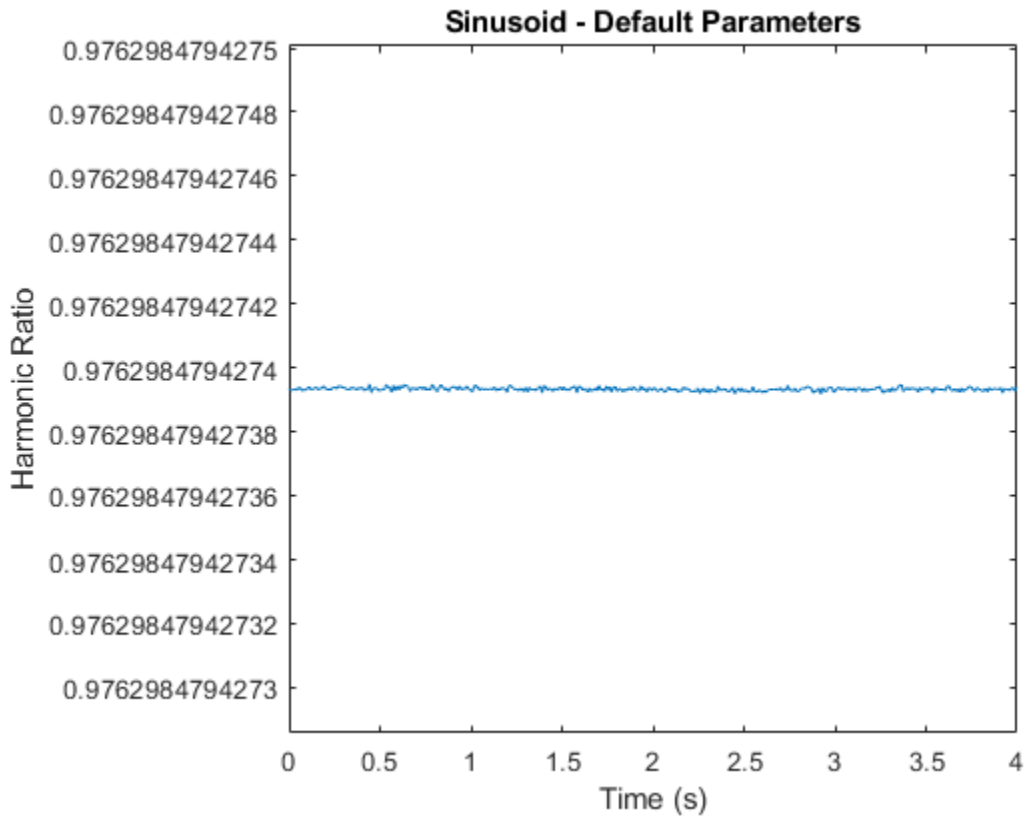
### **Harmonic Ratio of Pure Tone**

Create a pure tone and then calculate the harmonic ratio using default parameters. By default, the harmonic ratio is calculated for 30 ms Hamming windows with 10 ms hops. Plot the results. The harmonic ratio is near 1, which is the theoretical maximum.

```
fs = 48e3;
osc = audioOscillator('Frequency',500, ...
    'SamplesPerFrame',192e3,'SampleRate',fs);
sinewave = osc();

hr = harmonicRatio(sinewave,fs);

t = linspace(0,size(sinewave,1)/fs,size(hr,1));
plot(t,hr)
xlabel('Time (s)')
ylabel('Harmonic Ratio')
title('Sinusoid - Default Parameters')
```

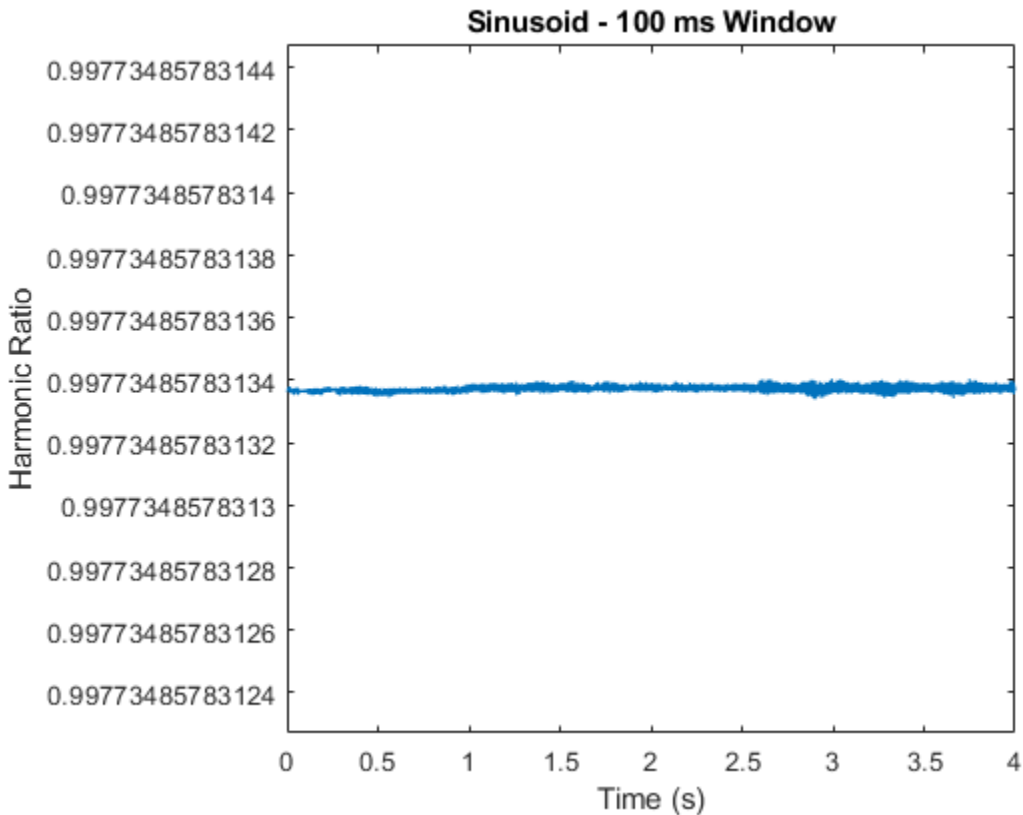


The short-time analysis required for windowing lowers the harmonic ratio from the theoretical value of 1. To diminish the effect of windowing, you can increase the window size. Use a 100 ms Hamming window and a 10 ms hop, and observe that the harmonic ratio is closer to one than when using the default window length.

```
win = hamming(round(fs*0.1), 'periodic');
overlap = round(fs*0.099);

hr = harmonicRatio(sinewave, fs, 'Window', win, 'OverlapLength', overlap);

t = linspace(0, size(sinewave, 1)/fs, size(hr, 1));
plot(t, hr)
xlabel('Time (s)')
ylabel('Harmonic Ratio')
title('Sinusoid - 100 ms Window')
```

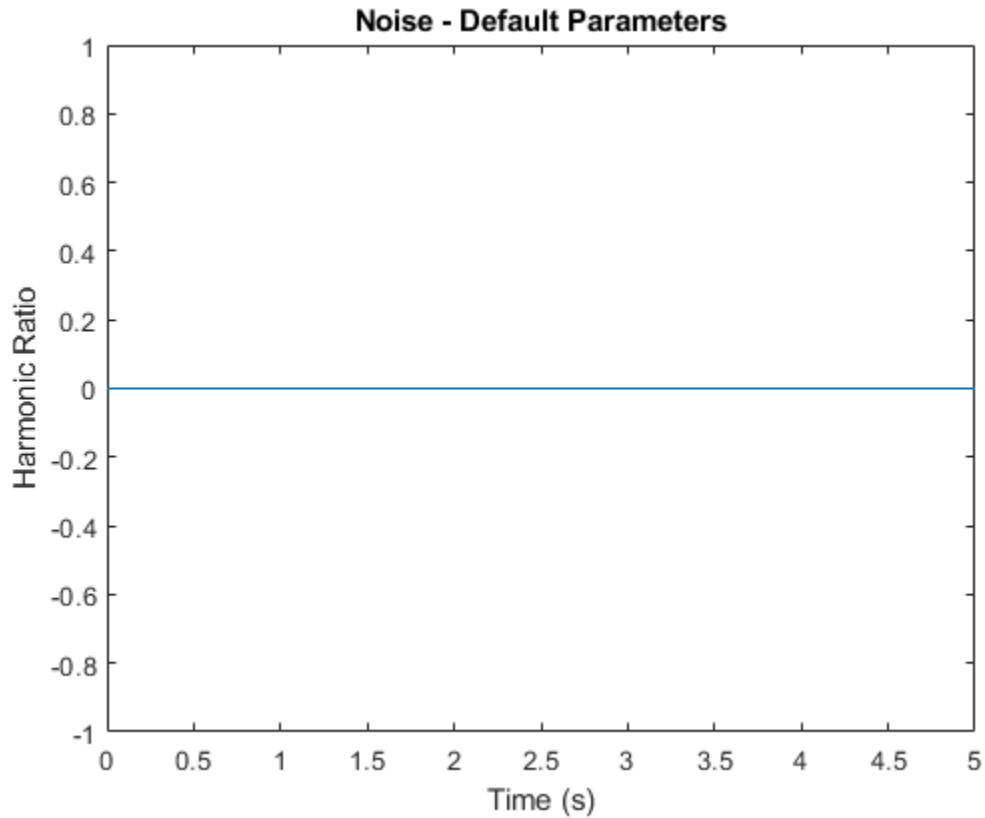


### Harmonic Ratio of White Noise

Create 5 seconds of white noise and then calculate the harmonic ratio using default parameters. By default, the harmonic ratio is calculated for 30 ms Hamming windows with 10 ms hops. Plot the results. The harmonic ratio is 0.

```
fs = 48e3;  
noise = rand(fs*5,1);  
  
hr = harmonicRatio(noise,fs);  
  
t = linspace(0,size(noise,1)/fs,size(hr,1));  
plot(t,hr)  
xlabel('Time (s)')
```

```
ylabel('Harmonic Ratio')  
title('Noise - Default Parameters')
```



## Input Arguments

### **audioIn** — Input signal

column vector | matrix

Input signal, specified as a column vector or matrix. If specified as a matrix, `harmonicRatio` treats the columns of the matrix as individual audio channels.

Data Types: `single` | `double`



**fs — Sample rate (Hz)**

positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Data Types: `single` | `double`**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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Window', hamming(256)`**Window — Window applied in time domain**`hamming(round(fs*0.03), 'periodic')` (default) | vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range `[1, size(audioIn, 1)]`. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`**OverlapLength — Number of samples overlapped between adjacent windows**`round(fs*0.02)` (default) | nonnegative integer scalar

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of `'OverlapLength'` and an integer in the range `[0, size(Window, 1)]`.

Data Types: `single` | `double`**Output Arguments****hr — Harmonic ratio**

scalar | vector | matrix

Harmonic ratio, returned as a scalar, vector, or matrix. Each row of `hr` corresponds to the harmonic ratio of a window of `audioIn`. The harmonic ratio is returned with values in the

range [0,1]. A value of 0 represents low harmonicity, and a value of 1 represents high harmonicity.

Data Types: `single` | `double`

## Algorithms

The harmonic ratio is calculated as described in [1]. The following algorithm is applied independently to each window of audio data. The normalized autocorrelation of the signal is determined as:

$$\Gamma(m) = \frac{\sum_{n=1}^N s(n)s(n-m)}{\sqrt{\sum_{n=1}^N s(n)^2 \sum_{n=0}^N s(n-m)^2}} \text{ for } (1 \leq m \leq M)$$

where

- $s$  is a single frame of audio data with  $N$  elements.
- $M$  is the maximum lag in the calculation. The maximum lag is 40 ms, which corresponds to a minimum fundamental frequency of 25 Hz.

A first estimate of the harmonic ratio is determined as the maximum of the normalized autocorrelation, within a given range:

$$\beta HR = \max_{M_0 \leq m \leq M} \{\Gamma(m)\}$$

where  $M_0$  is the lower edge of the search range, determined as the first zero crossing of the normalized autocorrelation.

Finally, the harmonic ratio estimate is improved using parabolic interpolation, as described in [2].

## References

- [1] Kim, Hyoung-Gook, Nicholas Moreau, and Thomas Sikora. *MPEG-7 Audio and Beyond: Audio Content Indexing and Retrieval*. John Wiley & Sons, 2005.

[2] Quadratic Interpolation of Spectral Peaks. Accessed October 11, 2018. [https://ccrma.stanford.edu/~jos/sasp/Quadratic\\_Interpolation\\_Spectral\\_Peaks.html](https://ccrma.stanford.edu/~jos/sasp/Quadratic_Interpolation_Spectral_Peaks.html)

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`pitch` | `spectralCentroid` | `voiceActivityDetector`

**Introduced in R2019a**

## gtcc

Extract gammatone cepstral coefficients, log-energy, delta, and delta-delta

### Syntax

```
coeffs = gtcc(audioIn,fs)
coeffs = gtcc( ____,Name,Value)
[coeffs,delta,deltaDelta,loc] = gtcc( ____ )
```

### Description

`coeffs = gtcc(audioIn,fs)` returns the gammatone cepstral coefficients (GTCCs) for the audio input, sampled at a frequency of `fs` Hz.

`coeffs = gtcc( ____,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

`[coeffs,delta,deltaDelta,loc] = gtcc( ____ )` returns the delta, delta-delta, and location in samples corresponding to each window of data. This output syntax can be used with any of the previous input syntaxes.

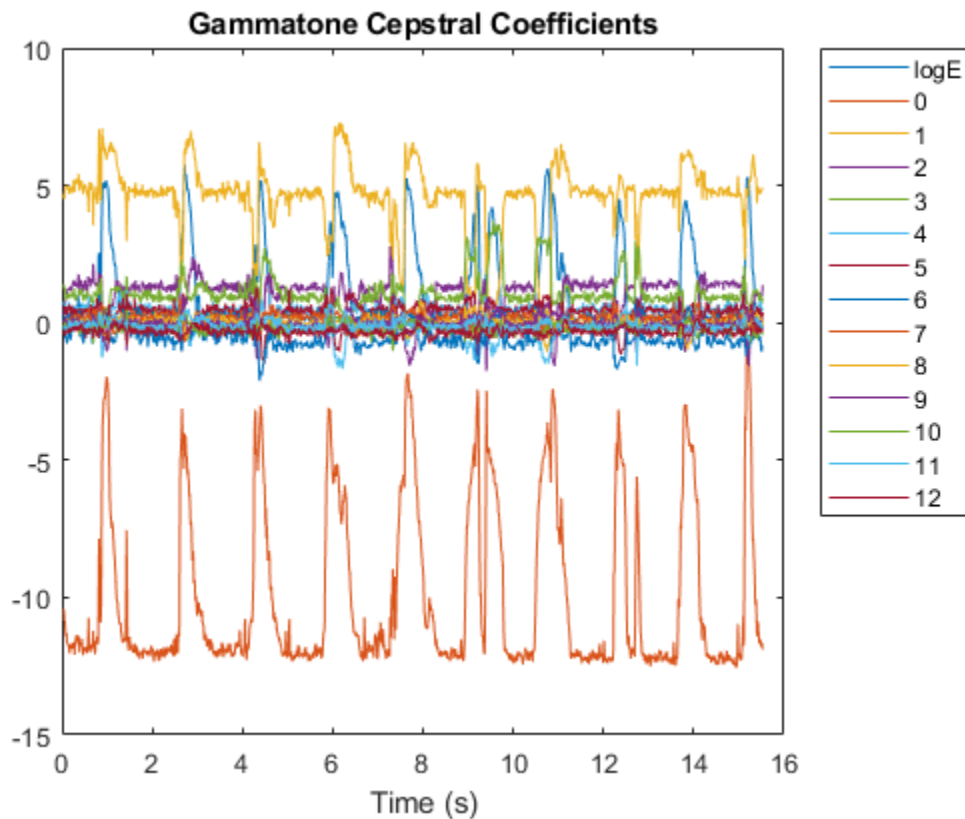
### Examples

#### Extract GTCC from Audio Signal

Get the gammatone cepstral coefficients for an audio file using default settings. Plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
[coeffs,~,~,loc] = gtcc(audioIn,fs);
t = loc./fs;
```

```
plot(t,coeffs)
xlabel('Time (s)')
title('Gammatone Cepstral Coefficients')
legend('logE','0','1','2','3','4','5','6','7','8','9','10','11','12', ...
       'Location','northeastoutside')
```



### Specify Nondefault Parameters

Read in an audio file.

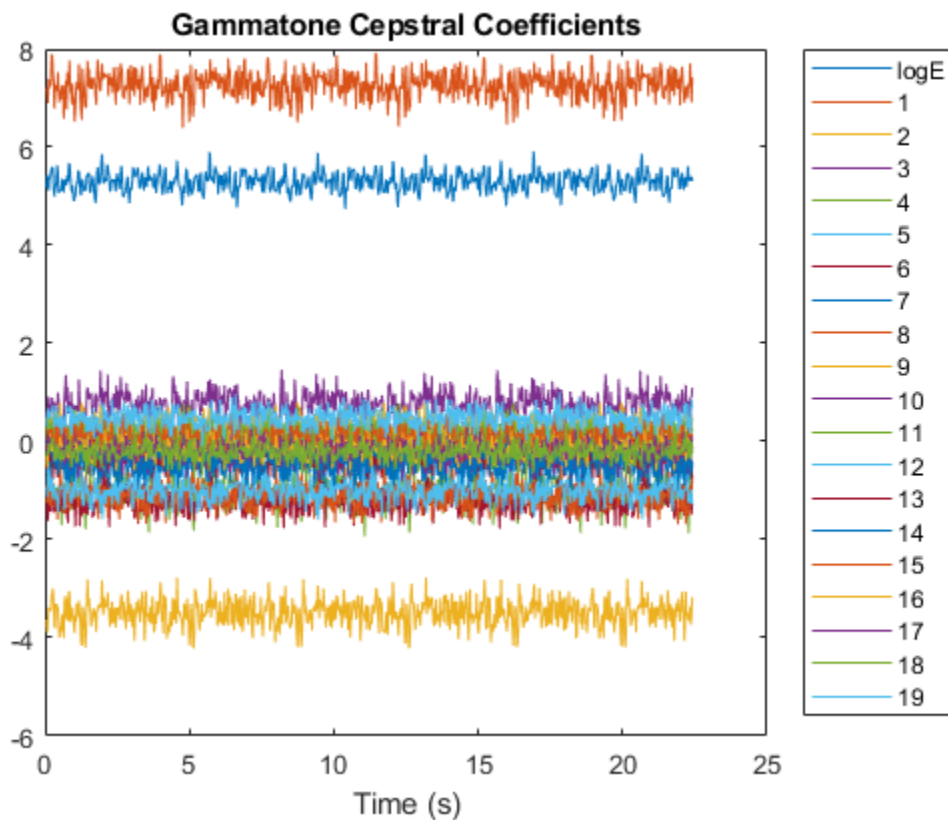
```
[audioIn,fs] = audioread('Turbine-16-44p1-mono-22secs.wav');
```

Calculate 20 GTCC using filters equally spaced on the ERB scale between `hz2erb(62.5)` and `hz2erb(12000)`. Calculate the coefficients using 50 ms windows with 25 ms overlap. Replace the 0th coefficient with the log-energy. Use time-domain filtering.

```
[coeffs,~,~,loc] = gtcc(audioIn,fs, ...  
    'NumCoeffs',20, ...  
    'FrequencyRange',[62.5,12000], ...  
    'WindowLength',round(0.05*fs), ...  
    'OverlapLength',round(0.025*fs), ...  
    'LogEnergy','Replace', ...  
    'FilterDomain','Time');
```

Plot the results.

```
t = loc./fs;  
  
plot(t,coeffs)  
xlabel('Time (s)')  
title('Gammatone Cepstral Coefficients')  
legend('logE','1','2','3','4','5','6','7','8','9','10','11','12','13', ...  
    '14','15','16','17','18','19','Location','northeastoutside');
```



### Extract GTCC from Frequency-Domain Audio

Read in an audio file and convert it to a frequency representation.

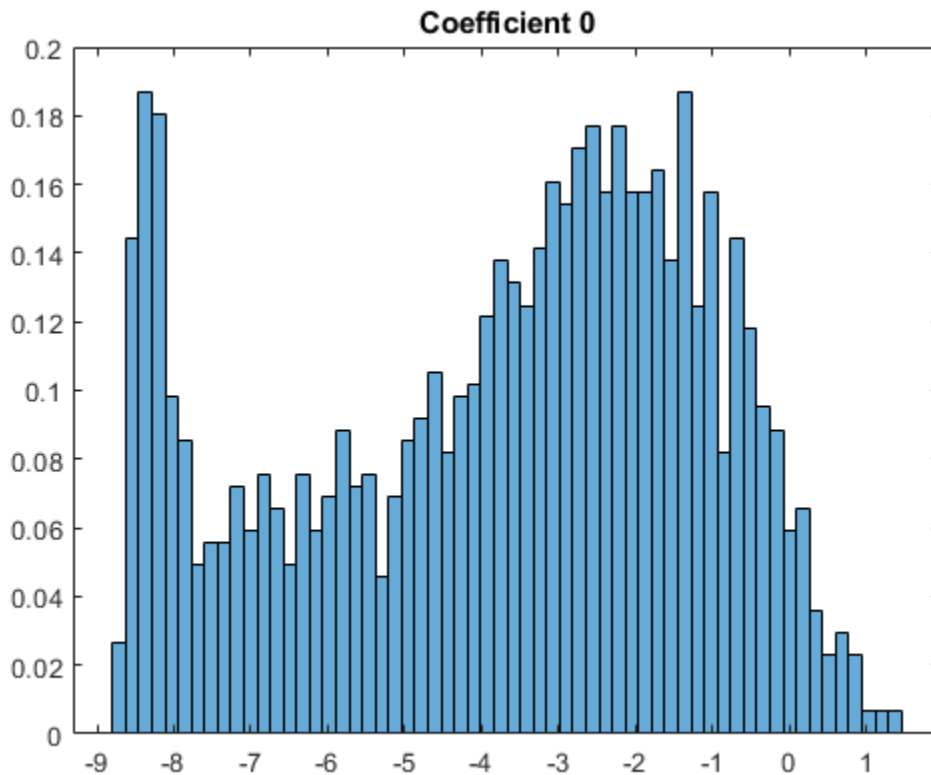
```
[audioIn,fs] = audioread("Rainbow-16-8-mono-114secs.wav");
win = hann(1024,"periodic");
S = stft(audioIn,"Window",win,"OverlapLength",512,"Centered",false);
```

To extract the gammatone cepstral coefficients, call `gtcc` with the frequency-domain audio. Ignore the log-energy.

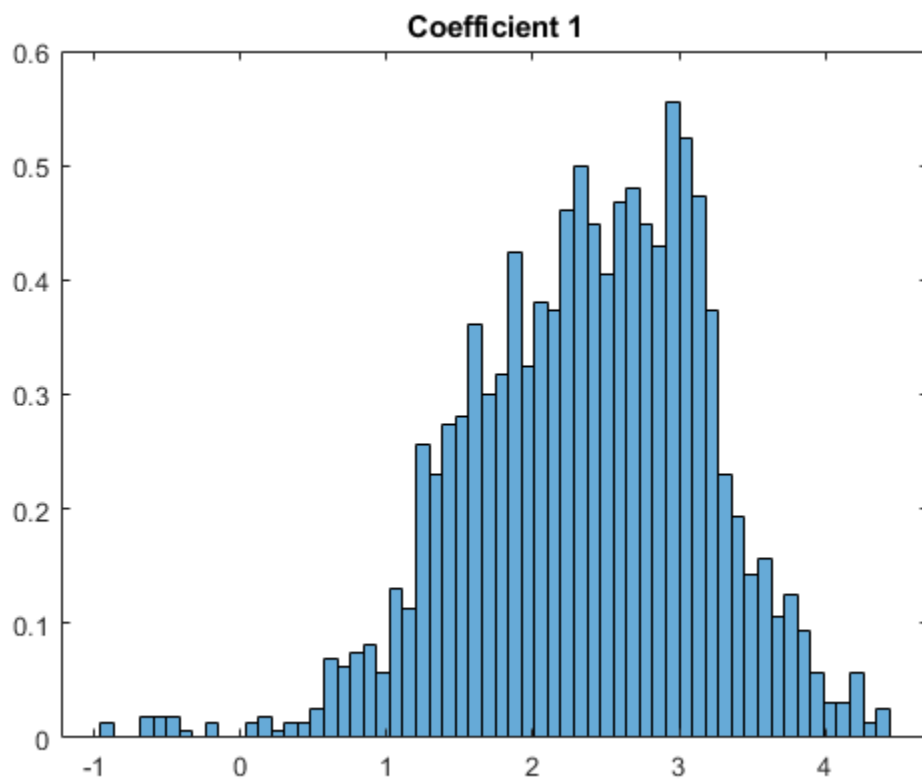
```
coeffs = gtcc(S,fs,"LogEnergy","Ignore");
```

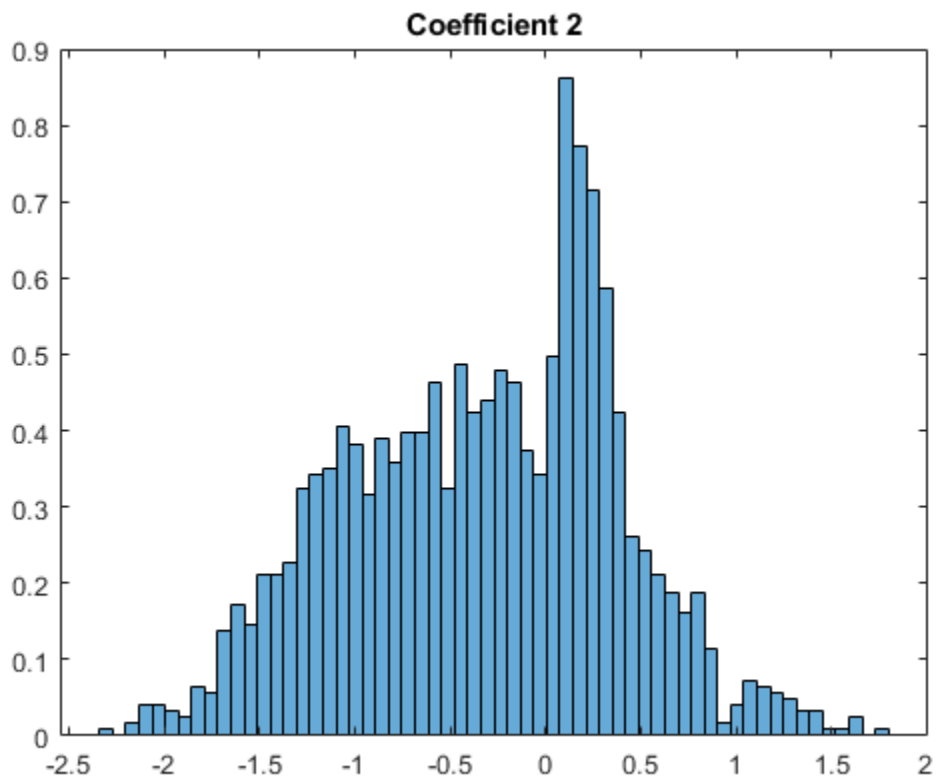
In many applications, GTCC observations are converted to summary statistics for use in classification tasks. Plot probability density functions of each of the gammatone cepstral coefficients to observe their distributions.

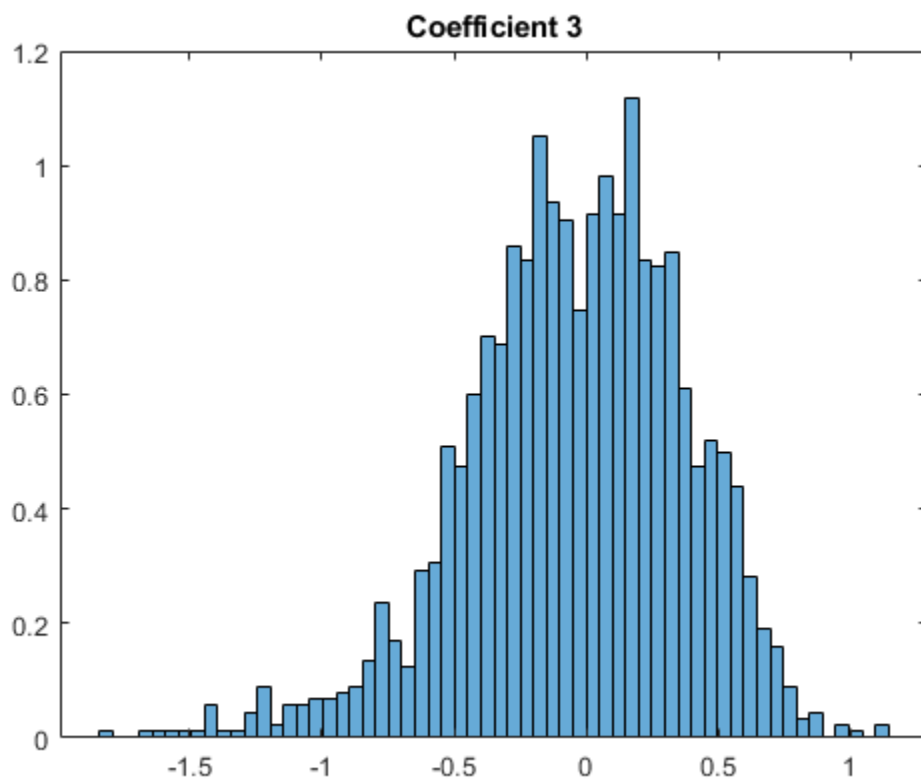
```
nbins = 60;  
for i = 1:size(coeffs,2)  
    figure  
    histogram(coeffs(:,i),nbins,'Normalization','pdf')  
    title(sprintf("Coefficient %d",i-1))  
end
```

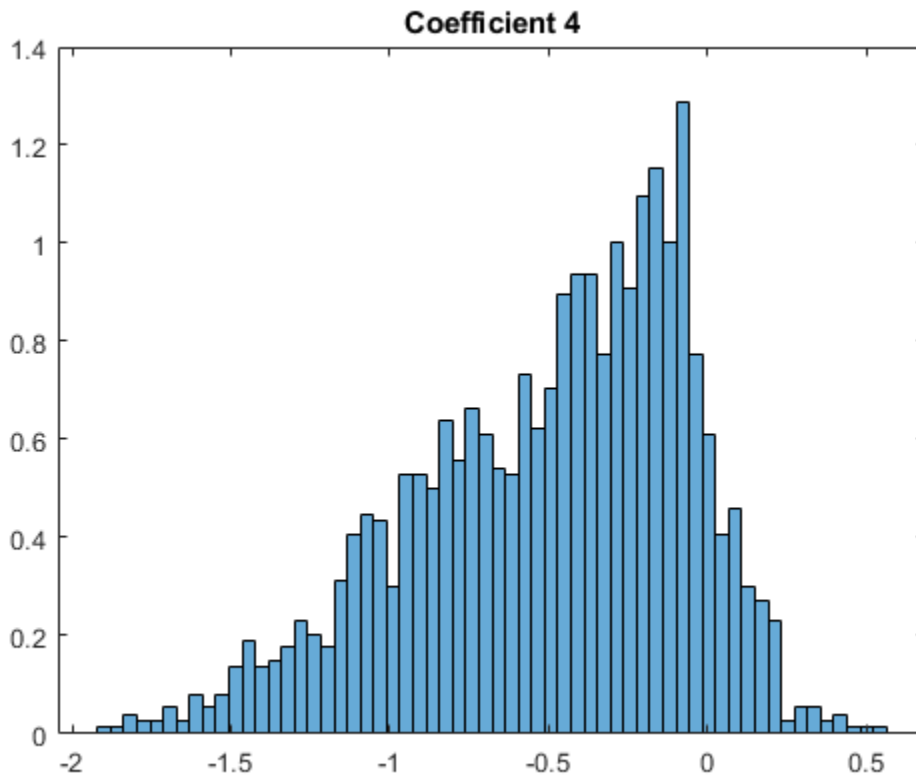


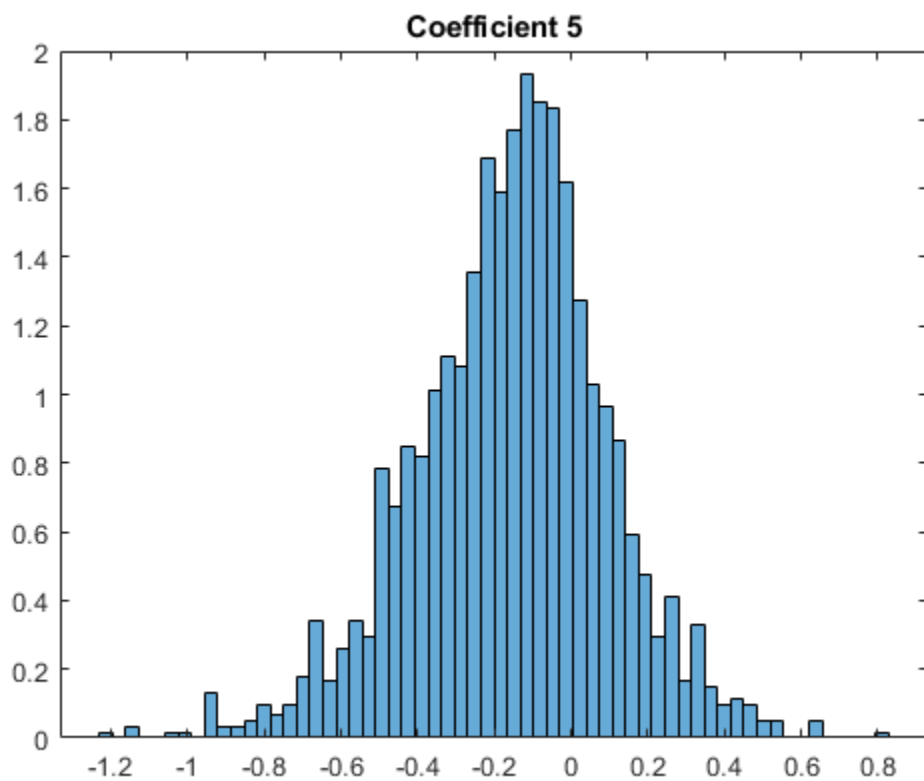


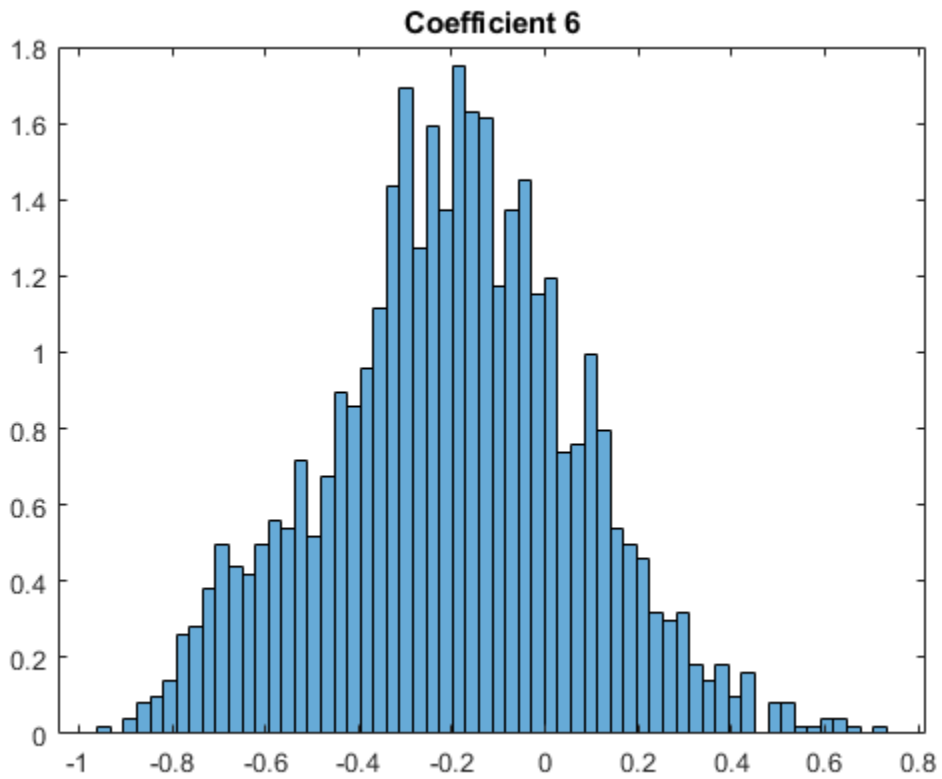


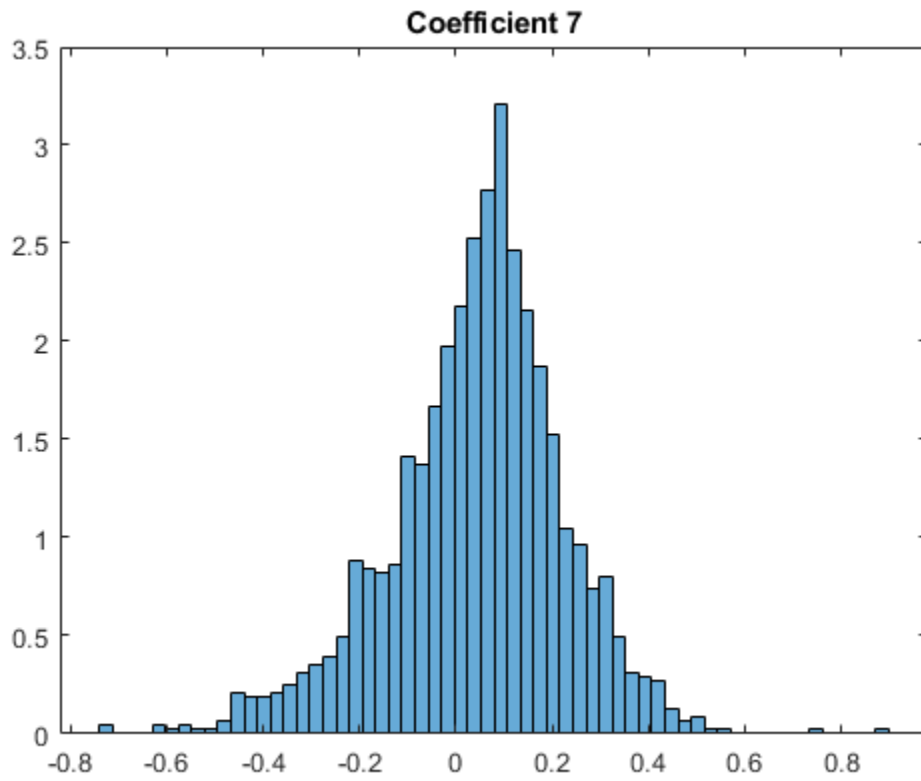


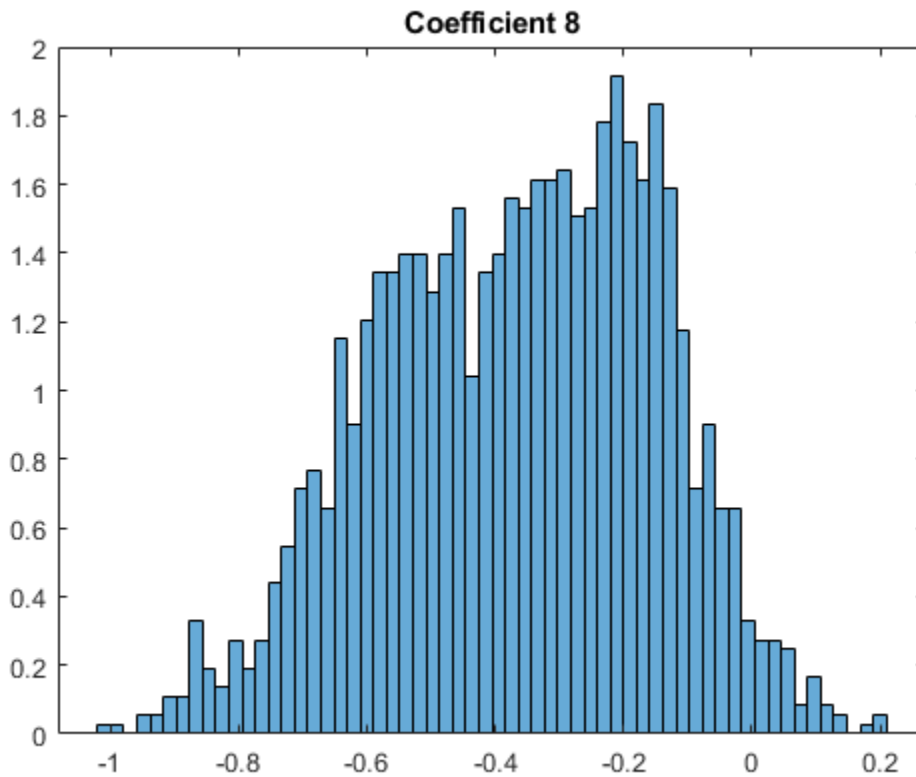




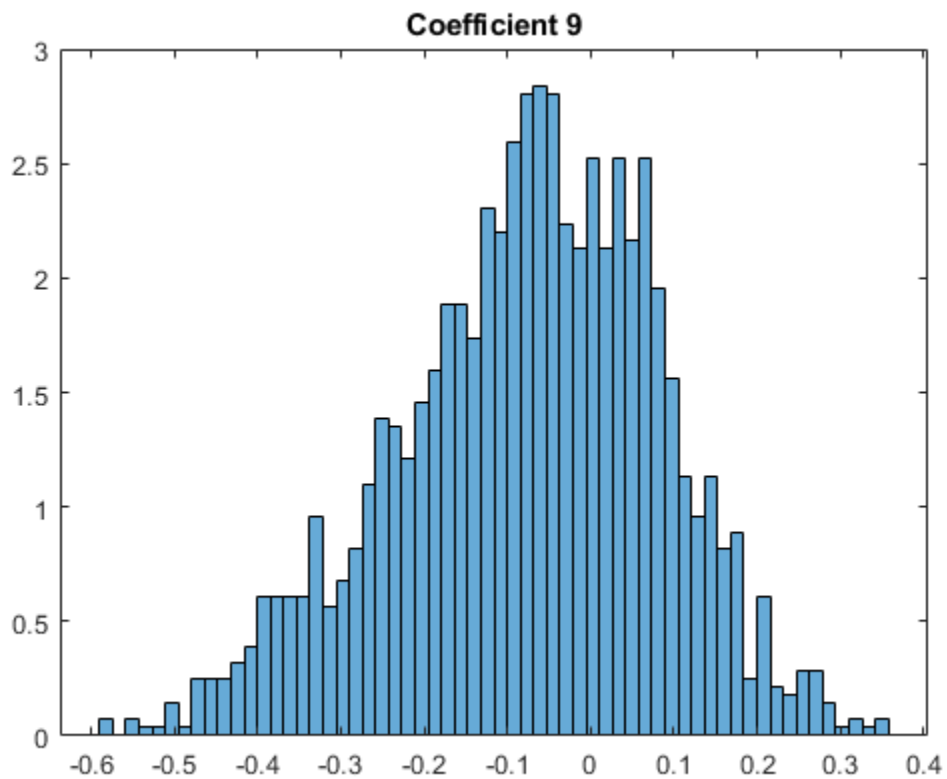


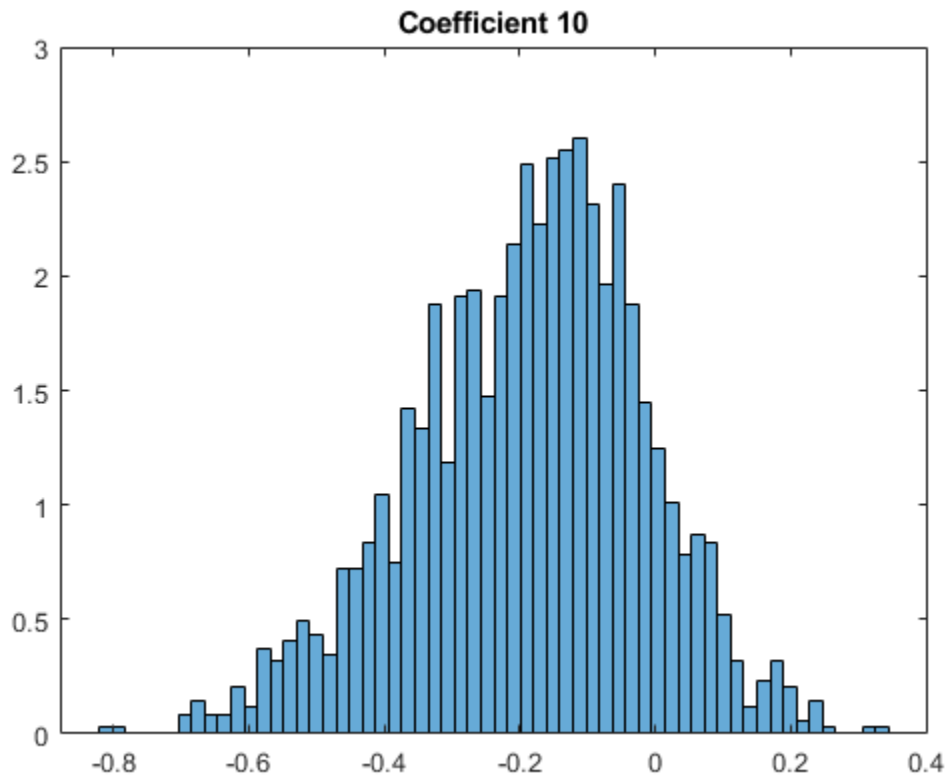


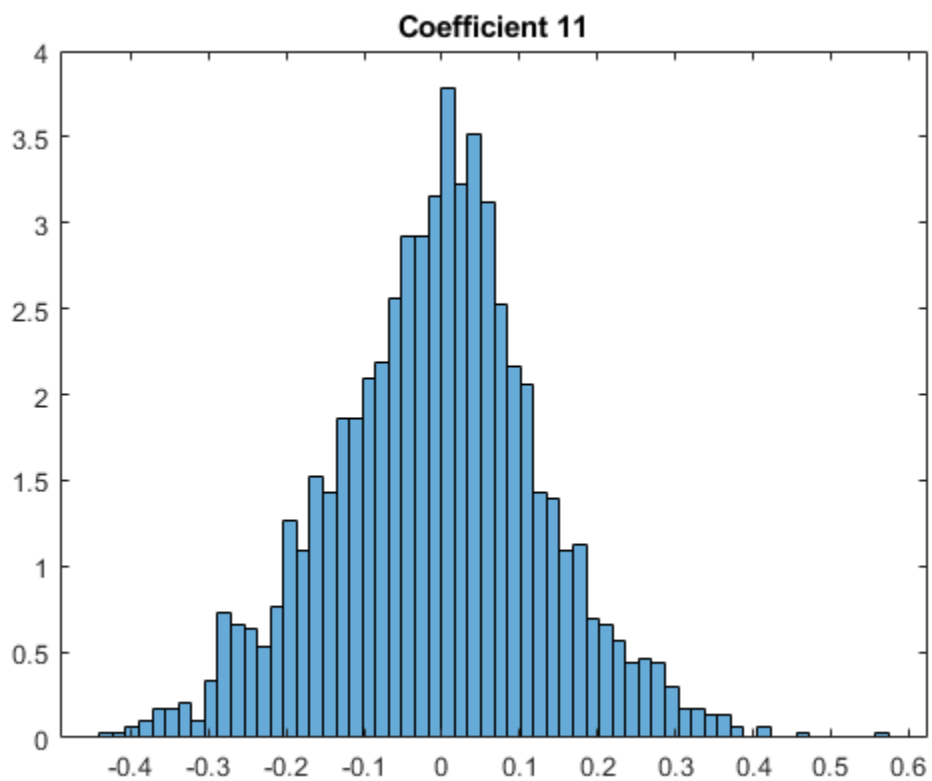


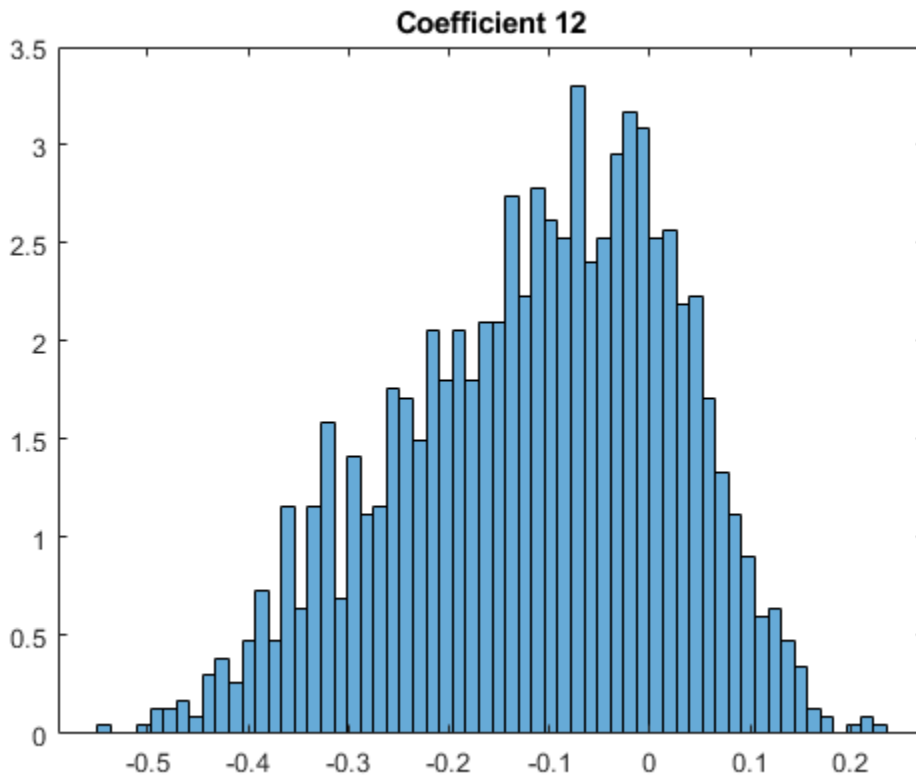












## Input Arguments

### **audioIn** — Input signal

vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array.

If 'FilterDomain' is set to 'Frequency' (default), then `audioIn` can be real or complex.

- If `audioIn` is real, it is interpreted as a time-domain signal and must be a column vector or a matrix. Columns of the matrix are treated as independent audio channels.

- If `audioIn` is complex, it is interpreted as a frequency-domain signal. In this case, `audioIn` must be an  $L$ -by- $M$ -by- $N$  array, where  $L$  is the number of DFT points,  $M$  is the number of individual spectrums, and  $N$  is the number of individual channels.

If `FilterDomain` is set to `'Time'`, then `audioIn` must be a real column vector or matrix. Columns of the matrix are treated as independent audio channels.

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

### **fs — Sample rate (Hz)**

positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Data Types: `single` | `double`

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `coeffs = gtcc(audioIn, fs, 'LogEnergy', 'Replace')` returns gammatone cepstral coefficients for the audio input signal sampled at `fs` Hz. For each analysis window, the first coefficient in the `coeffs` vector is replaced with the log energy of the input signal.

### **WindowLength — Number of samples in analysis window**

`round(0.03*fs)` (default) | positive scalar integer

Number of samples in analysis window used to calculate the coefficients, specified as the comma-separated pair consisting of `'WindowLength'` and an integer in the range `[2, size(audioIn, 1)]`. If unspecified, `WindowLength` defaults to `round(0.03*fs)`.

Data Types: `single` | `double`

### **OverlapLength — Number of samples overlapped between adjacent windows**

`round(0.02*fs)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of 'OverlapLength' and an integer in the range [0, WindowLength). If unspecified, OverlapLength defaults to  $\text{round}(0.02*fs)$ .

Data Types: single | double

**NumCoeffs — Number of coefficients returned**

13 (default) | positive scalar integer

Number of coefficients returned for each window of data, specified as the comma-separated pair consisting of 'NumCoeffs' and an integer in the range [2, v]. v is the number of valid passbands. If unspecified, NumCoeffs defaults to 13.

The number of valid passbands is defined as the number of ERB steps ( $\text{ERB}_N$ ) in the frequency range of the filter bank. The frequency range of the filter bank is specified by FrequencyRange.

Data Types: single | double

**FilterDomain — Domain in which to apply filtering**

'Frequency' (default) | 'Time'

Domain in which to apply filtering, specified as the comma-separated pair consisting of 'FilterDomain' and 'Frequency' or 'Time'. If unspecified, FilterDomain defaults to Frequency.

Data Types: string | char

**FrequencyRange — Frequency range of gammatone filter bank (Hz)**

[50 fs/2] (default) | two-element row vector

Frequency range of gammatone filter bank in Hz, specified as the comma-separated pair consisting of 'FrequencyRange' and a two-element row vector of increasing values in the range [0, fs/2]. If unspecified, FrequencyRange defaults to [50, fs/2]

Data Types: single | double

**FFTLength — Number of bins in DFT**

WindowLength (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as the comma-separated pair consisting of 'FFTLength' and a positive scalar integer. If unspecified, FFTLength defaults to WindowLength.

Data Types: `single` | `double`

### **DeltaWindowLength** — Number of coefficients used to calculate delta and delta-delta

2 (default) | odd integer greater than two

Number of coefficients used to calculate the delta and the delta-delta values, specified as the comma-separated pair consisting of `'DeltaWindowLength'` and two or an odd integer greater than two. If unspecified, `DeltaWindowLength` defaults to 2.

If `DeltaWindowLength` is set to 2, the `delta` is given by the difference between the current coefficients and the previous coefficients.

If `DeltaWindowLength` is set to an odd integer greater than 2, the following equation defines their values:

$$\mathit{delta} = \frac{\sum_{k=-M}^M k \cdot \mathit{coeffs}(k,:)}{\sum_{k=-M}^M k^2}$$

The function uses a least-squares approximation of the local slope over a region around the coefficients of the current analysis window. The delta cepstral values are computed by fitting the cepstral coefficients of neighboring analysis windows ( $M$  analysis windows before the current analysis window and  $M$  analysis windows after the current analysis window) to a straight line. For details, see [3].

Data Types: `single` | `double`

### **LogEnergy** — Log energy usage

`'Append'` (default) | `'Replace'` | `'Ignore'`

Log energy usage, specified as the comma-separated pair consisting of `'LogEnergy'` and `'Append'`, `'Replace'`, or `'Ignore'`. If unspecified, `LogEnergy` defaults to `Append`.

- `'Append'` -- The function prepends the log energy to the coefficients vector. The length of the coefficients vector is  $1 + \mathit{NumCoeffs}$ .
- `'Replace'` -- The function replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is  $\mathit{NumCoeffs}$ .

- 'Ignore' -- The function does not calculate or return the log energy.

Data Types: char | string

## Output Arguments

### **coeffs** — Gammatone cepstral coefficients

matrix | array

Gammatone cepstral coefficients, returned as an  $L$ -by- $M$  matrix or an  $L$ -by- $M$ -by- $N$  array, where:

- $L$  -- Number of analysis windows the audio signal is partitioned into. The input size, `WindowLength`, and `OverlapLength` control this dimension:  $L = \text{floor}((\text{size}(\text{audioIn},1) - \text{WindowLength})/(\text{WindowLength} - \text{OverlapLength}) + 1)$ .
- $M$  -- Number of coefficients returned per frame. This value is determined by `NumCoeffs` and `LogEnergy`.

When `LogEnergy` is set to:

- 'Append' -- The object prepends the log energy value to the coefficients vector. The length of the coefficients vector is  $1 + \text{NumCoeffs}$ .
- 'Replace' -- The object replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is `NumCoeffs`.
- 'Ignore' -- The object does not calculate or return the log energy. The length of the coefficients vector is `NumCoeffs`.
- $N$  -- Number of input channels (columns). This value is `size(audioIn,2)`.

Data Types: single | double

### **delta** — Change in coefficients

matrix | array

Change in coefficients from one analysis window to another, returned as an  $L$ -by- $M$  matrix or an  $L$ -by- $M$ -by- $N$  array. The `delta` array is the same size and data type as the `coeffs` array. See `coeffs` for the definitions of  $L$ ,  $M$ , and  $N$ .

The function uses a least-squares approximation of the local slope over a region around the current time sample. For details, see [3].



Data Types: `single` | `double`

### **deltaDelta — Change in delta values**

`matrix` | `array`

Change in `delta` values, returned as an  $L$ -by- $M$  matrix or an  $L$ -by- $M$ -by- $N$  array. The `deltaDelta` array is the same size and data type as the `coeffs` and `delta` arrays. See `coeffs` for the definitions of  $L$ ,  $M$ , and  $N$ .

The function uses a least-squares approximation of the local slope over a region around the current time sample. For details, see [3].

Data Types: `single` | `double`

### **loc — Location of the last sample in each analysis window**

`column vector`

Location of last sample in each analysis window, returned as a column vector with the same number of rows as `coeffs`.

Data Types: `single` | `double`

## **Algorithms**

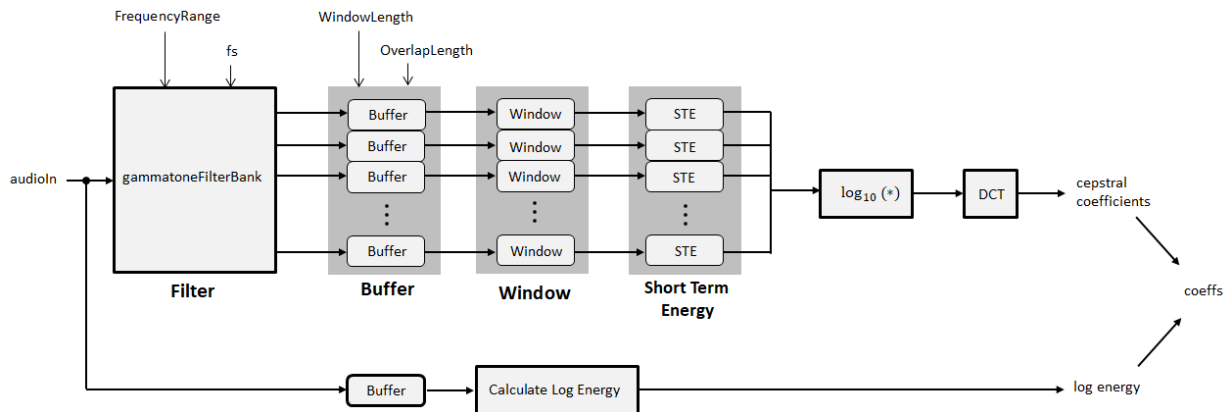
The `gtcc` function splits the entire data into overlapping segments. The length of each analysis window is determined by `WindowLength`. The length of overlap between analysis windows is determined by `OverlapLength`. The algorithm to determine the gammatone cepstral coefficients depends on the filter domain, specified by `FilterDomain`. The default filter domain is frequency.

## **Frequency-Domain Filtering**

`gtcc` computes the gammatone cepstral coefficients, log energy values, `delta`, and `delta-delta` values for each analysis window as per the algorithm described in `cepstralFeatureExtractor`.

## Time-Domain Filtering

If `FilterDomain` is specified as 'Time', the `gtcc` function uses the `gammatoneFilterBank` to apply time-domain filtering. The basic steps of the `gtcc` algorithm are outlined by the diagram.



The `FrequencyRange` and sample rate (`fs`) parameters are set on the filter bank using the name-value pairs input to the `gtcc` function. The number of filters in the gammatone filter bank is defined as  $\text{hz2erfb}(\text{FrequencyRange}(2)) - \text{hz2erfb}(\text{FrequencyRange}(1))$ . This roughly corresponds to placing a gammatone filter every 0.9 mm in the cochlea.

The output from the gammatone filter bank is a multichannel signal. Each channel output from the gammatone filter bank is buffered into overlapped analysis windows, as specified by `WindowLength` and `OverlapLength`. Then a periodic Hamming window is applied to each analysis window of data. The energy for each analysis window of data is calculated. The STE of the channels are concatenated. The concatenated signal is then passed through a logarithm function and transformed to the cepstral domain using a discrete cosine transform (DCT).

The log-energy is calculated on the original audio signal using the same buffering scheme applied to the gammatone filter bank output.

## References

- [1] Shao, Yang, Zhaozhang Jin, Deliang Wang, and Soundararajan Srinivasan. "An Auditory-Based Feature for Robust Speech Recognition." *IEEE International Conference on Acoustics, Speech and Signal Processing*. 2009.
- [2] Valero, X., and F. Alias. "Gammatone Cepstral Coefficients: Biologically Inspired Features for Non-Speech Audio Classification." *IEEE Transactions on Multimedia*. Vol. 14, Issue 6, 2012, pp. 1684-1689.
- [3] Rabiner, Lawrence R., and Ronald W. Schafer. *Theory and Applications of Digital Speech Processing*. Upper Saddle River, NJ: Pearson, 2010.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`cepstralFeatureExtractor` | `mfcc` | `pitch` | `voiceActivityDetector`

**Introduced in R2019a**

## spectralSpread

Spectral spread for audio signals and auditory spectrograms

### Syntax

```
spread = spectralSpread(x,f)
spread = spectralSpread(x,f,Name,Value)
[spread,centroid] = spectralSpread( ___ )
```

### Description

`spread = spectralSpread(x,f)` returns the spectral spread of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`spread = spectralSpread(x,f,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

`[spread,centroid] = spectralSpread( ___ )` returns the spectral centroid.

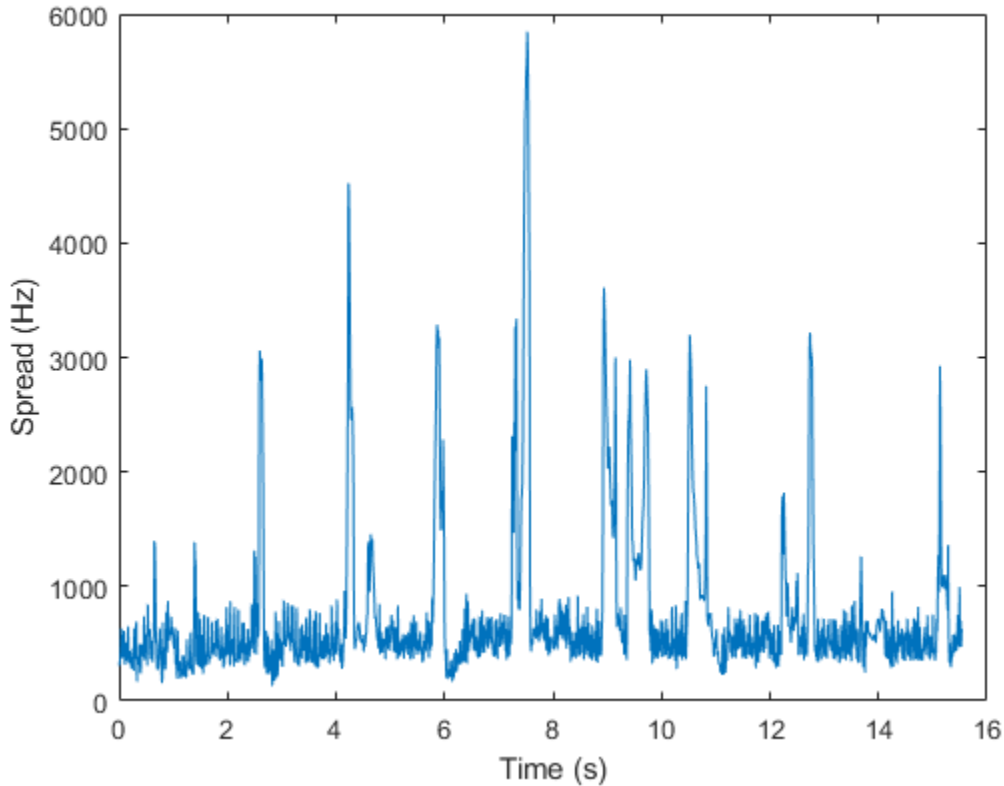
### Examples

#### Spectral Spread of Time-Domain Audio

Read in an audio file, calculate the spread using default parameters, and then plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
spread = spectralSpread(audioIn,fs);

t = linspace(0,size(audioIn,1)/fs,size(spread,1));
plot(t,spread)
xlabel('Time (s)')
ylabel('Spread (Hz)')
```



### Spectral Spread of Frequency-Domain Audio Data

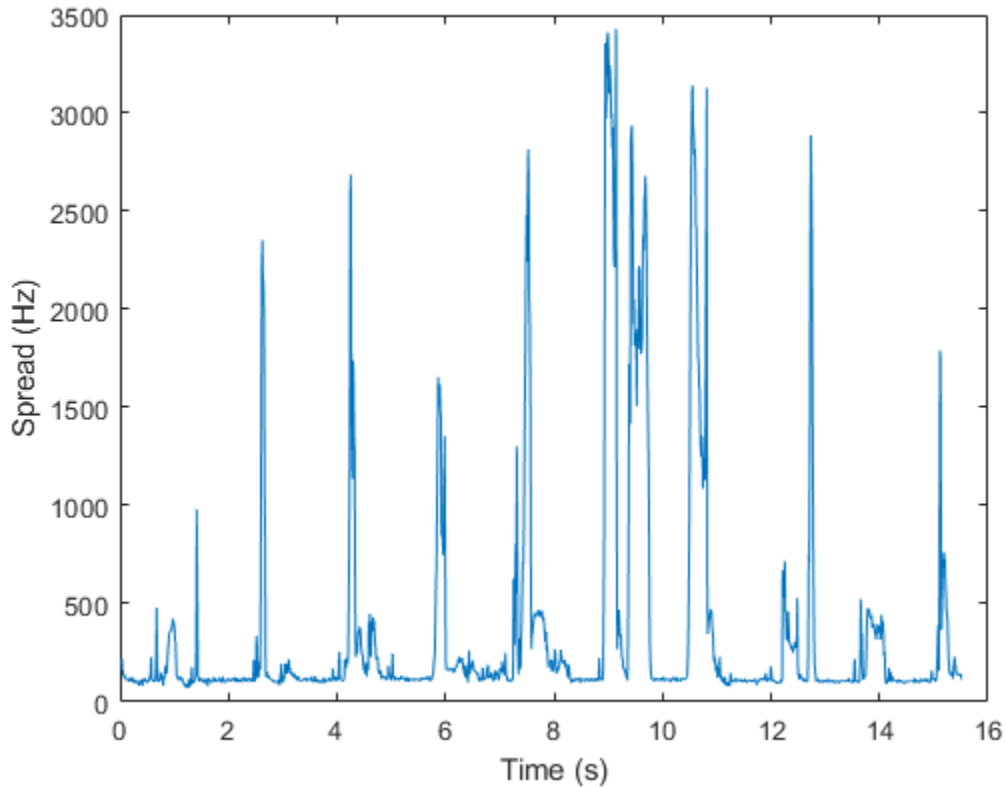
Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function. Calculate the spread of the mel spectrums over time. Plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

```
[s,cf,t] = melSpectrogram(audioIn,fs);
```

```
spread = spectralSpread(s,cf);
```

```
plot(t,spread)
xlabel('Time (s)')
ylabel('Spread (Hz)')
```



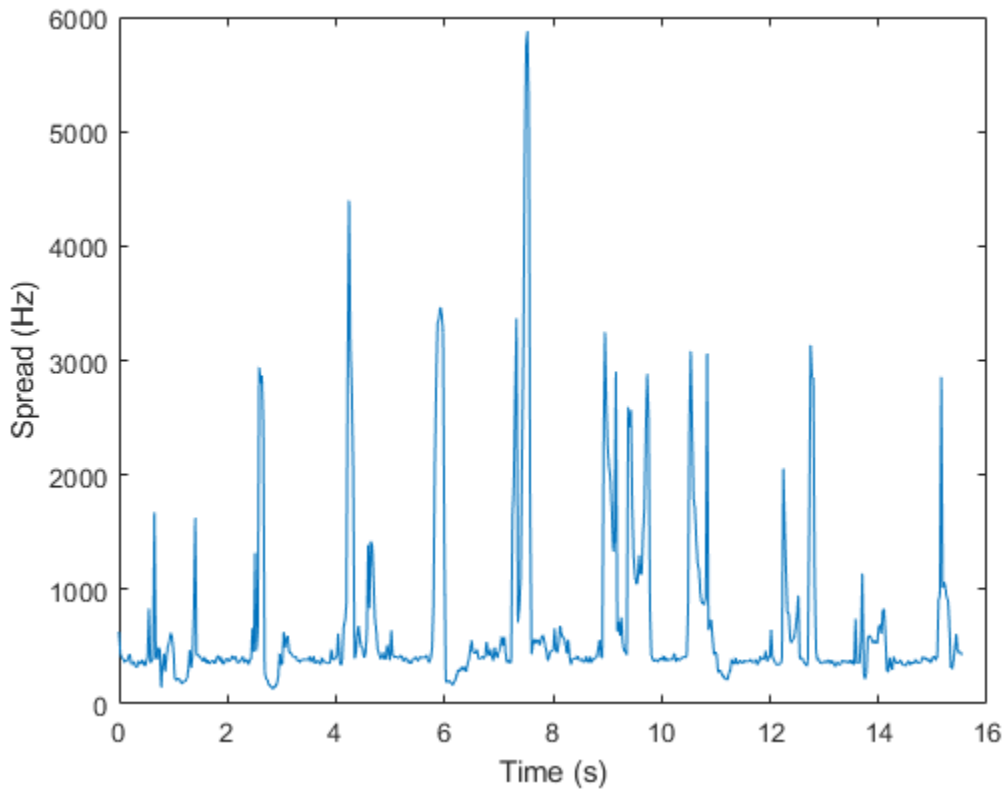
### Specify Nondefault Parameters

Read in an audio file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Calculate the spread of the power spectrum over time. Calculate the spread for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to  $f_s/2$  for the spread calculation. Plot the results.

```
spread = spectralSpread(audioIn,fs, ...  
                        'Window',hamming(round(0.05*fs)), ...  
                        'OverlapLength',round(0.025*fs), ...  
                        'Range',[62.5,fs/2]);  
  
t = linspace(0,size(audioIn,1)/fs,size(spread,1));  
plot(t,spread)  
xlabel('Time (s)')  
ylabel('Spread (Hz)')
```



### Calculate Spectral Spread of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral spread calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');  
logger = dsp.SignalSink;
```

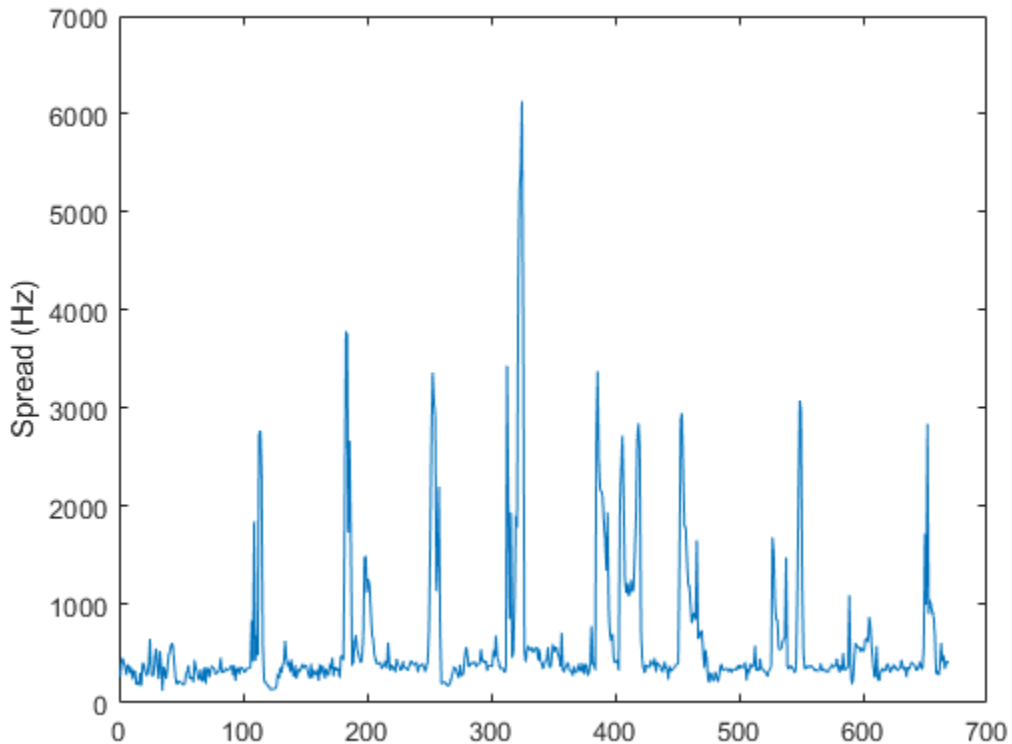
In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Calculate the spectral spread for the frame of audio.
- 3 Log the spectral spread for later plotting.

To calculate the spectral spread for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame);  
while ~isDone(fileReader)  
    audioIn = fileReader();  
    spread = spectralSpread(audioIn,fileReader.SampleRate, ...  
                           'Window',win, ...  
                           'OverlapLength',0);  
    logger(spread)  
end  
  
plot(logger.Buffer)  
ylabel('Spread (Hz)')
```





Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralSpread`.
- You want to calculate the spectral spread for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;  
reset(logger)  
release(fileReader)
```

Specify that the spectral spread is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

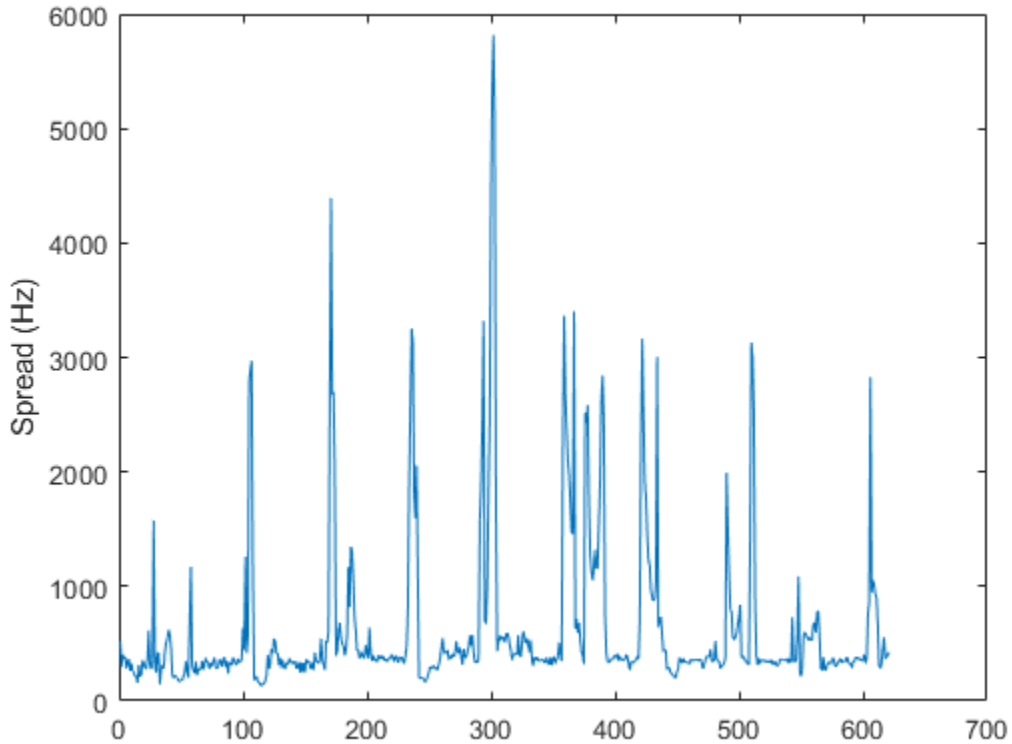
    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        spread = spectralSpread(audioBuffered,fs, ...
                                'Window',win, ...
                                'OverlapLength',0);

        logger(spread)
    end
end
release(fileReader)

Plot the logged data.

plot(logger.Buffer)
ylabel('Spread (Hz)')
```



## Input Arguments

### **x** — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets  $x$  depends on the shape of  $f$ .

Data Types: `single` | `double`

### **f** — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets  $x$  depends on the shape of  $f$ :

- If  $f$  is a scalar,  $x$  is interpreted as a time-domain signal, and  $f$  is interpreted as the sample rate. In this case,  $x$  must be a real vector or matrix. If  $x$  is specified as a matrix, the columns are interpreted as individual channels.
- If  $f$  is a vector,  $x$  is interpreted as a frequency-domain signal, and  $f$  is interpreted as the frequencies, in Hz, corresponding to the rows of  $x$ . In this case,  $x$  must be a real  $L$ -by- $M$ -by- $N$  array, where  $L$  is the number of spectral values at given frequencies of  $f$ ,  $M$  is the number of individual spectrums, and  $N$  is the number of channels.
- The number of rows of  $x$ ,  $L$ , must be equal to the number of elements of  $f$ .

Data Types: `single` | `double`

### Name-Value Pair Arguments

---

**Note** The following name-value pair arguments apply if  $x$  is a time-domain signal. If  $x$  is a frequency-domain signal, name-value pair arguments are ignored.

---

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

Example: `'Window', hamming(256)`

#### **Window — Window applied in time domain**

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range `[1, size(x,1)]`. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

#### **OverlapLength — Number of samples overlapped between adjacent windows**

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of 'OverlapLength' and an integer in the range [0, size(Window,1)].

Data Types: single | double

### **FFTLength — Number of bins in DFT**

numel(Window) (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as the comma-separated pair consisting of 'FFTLength' and a positive scalar integer. If unspecified, FFTLength defaults to the number of elements in the Window.

Data Types: single | double

### **Range — Frequency range (Hz)**

[0, f/2] (default) | two-element row vector

Frequency range in Hz, specified as the comma-separated pair consisting of 'Range' and a two-element row vector of increasing real values in the range [0, f/2].

Data Types: single | double

### **SpectrumType — Spectrum type**

'power' (default) | 'magnitude'

Spectrum type, specified as the comma-separated pair consisting of 'SpectrumType' and 'power' or 'magnitude':

- 'power' -- The spectral spread is calculated for the one-sided power spectrum.
- 'magnitude' -- The spectral spread is calculated for the one-sided magnitude spectrum.

Data Types: char | string

## **Output Arguments**

### **spread — Spectral spread (Hz)**

scalar | vector | matrix

Spectral spread in Hz, returned as a scalar, vector, or matrix. Each row of `centroid` corresponds to the spectral spread of a window of `x`. Each column of `spread` corresponds to an independent channel.

**centroid — Spectral centroid (Hz)**

scalar | vector | matrix

Spectral centroid in Hz, returned as a scalar, vector, or matrix. Each row of `centroid` corresponds to the spectral centroid of a window of `x`. Each column of `centroid` corresponds to an independent channel.

## Algorithms

The spectral spread is calculated as described in [1]:

$$\text{spread} = \sqrt{\frac{\sum_{k=b_1}^{b_2} (f_k - \mu_1)^2 s_k}{\sum_{k=b_1}^{b_2} s_k}}$$

where

- $f_k$  is the frequency in Hz corresponding to bin  $k$ .
- $s_k$  is the spectral value at bin  $k$ .
- $b_1$  and  $b_2$  are the band edges, in bins, over which to calculate the spectral spread.
- $\mu_1$  is the spectral centroid, calculated as described by the `spectralCentroid` function.

## References

- [1] Peeters, G. "A Large Set of Audio Features for Sound Description (Similarity and Classification) in the CUIDADO Project." Technical Report; IRCAM: Paris, France, 2004.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`spectralCentroid` | `spectralKurtosis` | `spectralSkewness`

### Topics

“Spectral Descriptors”

**Introduced in R2019a**

## spectralSlope

Spectral slope for audio signals and auditory spectrograms

### Syntax

```
slope = spectralSlope(x,f)  
slope = spectralSlope(x,f,Name,Value)
```

### Description

`slope = spectralSlope(x,f)` returns the spectral slope of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`slope = spectralSlope(x,f,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

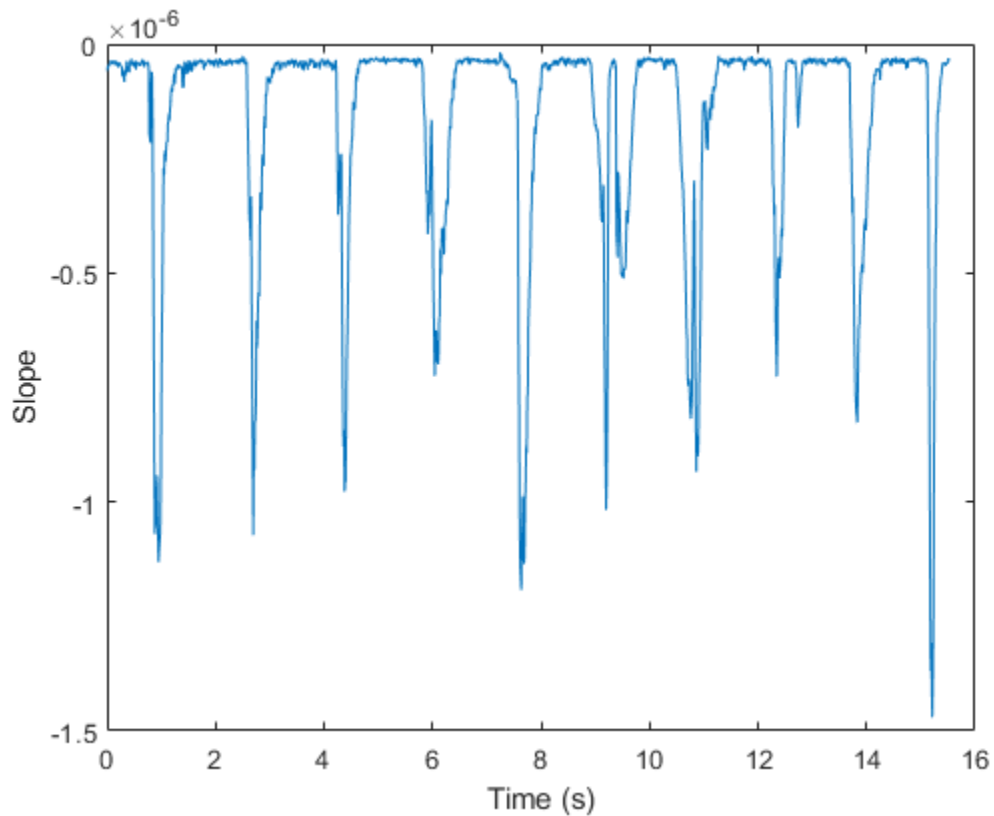
### Examples

#### Spectral Slope of Time-Domain Audio

Read in an audio file, calculate the slope using default parameters, and then plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
slope = spectralSlope(audioIn,fs);  
  
t = linspace(0,size(audioIn,1)/fs,size(slope,1));  
plot(t,slope)  
xlabel('Time (s)')  
ylabel('Slope')
```



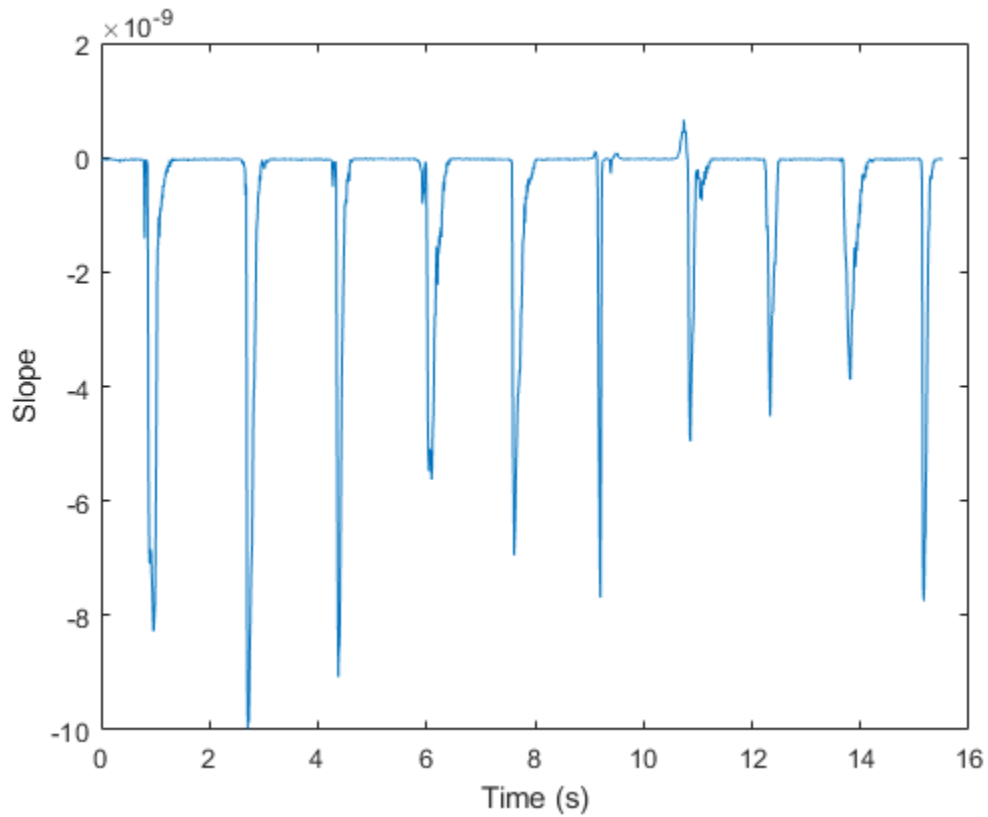


### Spectral Slope of Frequency-Domain Audio Data

Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function. Calculate the slope of the mel spectrogram over time. Plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
[s,cf,t] = melSpectrogram(audioIn,fs);  
slope = spectralSlope(s,cf);
```

```
plot(t,slope)
xlabel('Time (s)')
ylabel('Slope')
```



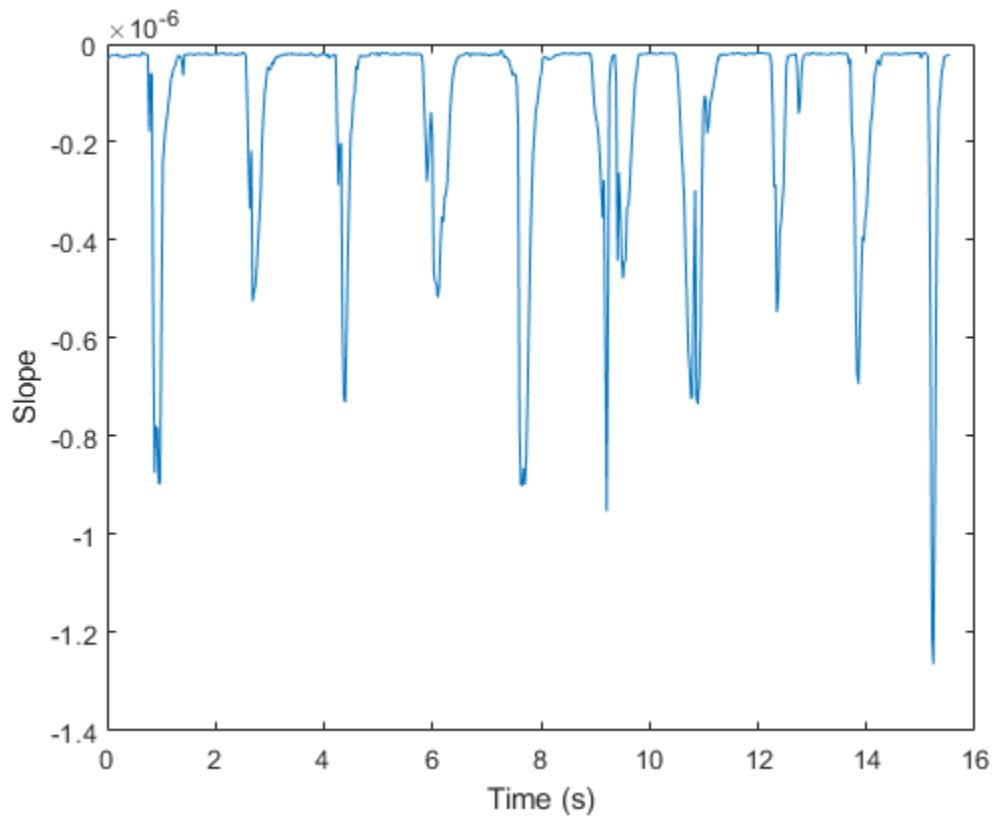
### Specify Nondefault Parameters

Read in an audio file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Calculate the slope of the magnitude spectrum over time. Calculate the slope for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to  $f_s/2$  for the slope calculation. Plot the results.

```
slope = spectralSlope(audioIn,fs, ...  
                    'Window',hamming(round(0.05*fs)), ...  
                    'OverlapLength',round(0.025*fs), ...  
                    'Range',[62.5,fs/2]);  
  
t = linspace(0,size(audioIn,1)/fs,size(slope,1));  
plot(t,slope)  
xlabel('Time (s)')  
ylabel('Slope')
```



### Calculate Spectral Slope of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral slope calculation.

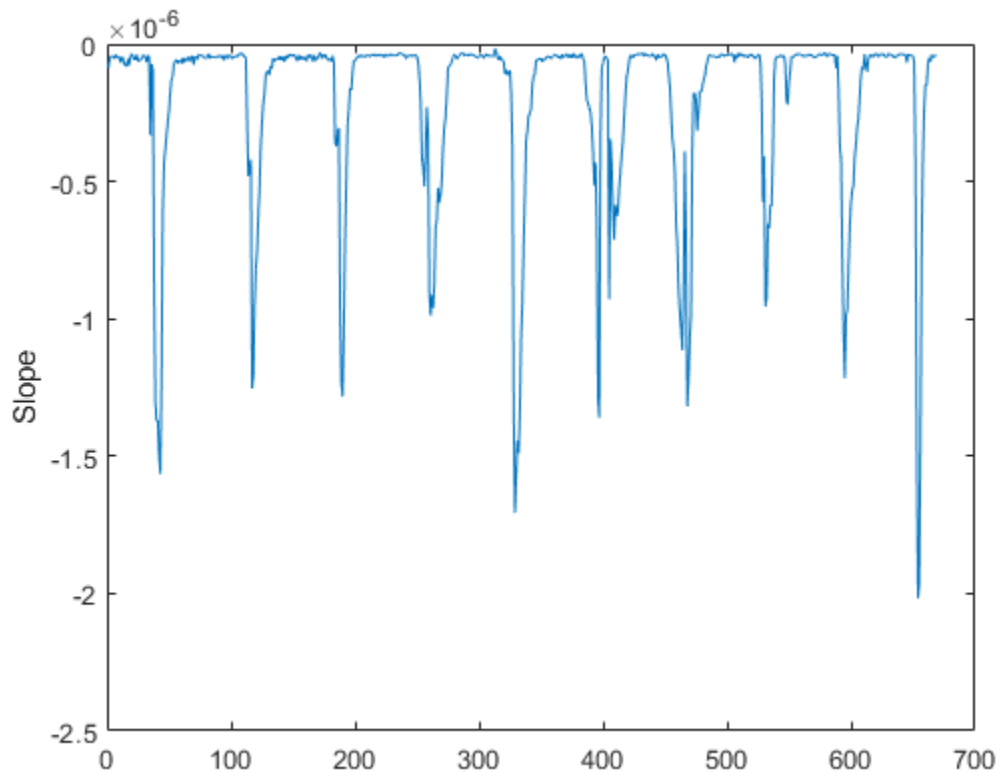
```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');  
logger = dsp.SignalSink;
```

In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Calculate the spectral slope for the frame of audio.
- 3 Log the spectral slope for later plotting.

To calculate the spectral slope for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame);  
while ~isDone(fileReader)  
    audioIn = fileReader();  
    slope = spectralSlope(audioIn,fileReader.SampleRate, ...  
                        'Window',win, ...  
                        'OverlapLength',0);  
    logger(slope)  
end  
  
plot(logger.Buffer)  
ylabel('Slope')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralSlope`.
- You want to calculate the spectral slope for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;  
reset(logger)  
release(fileReader)
```

Specify that the spectral slope is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

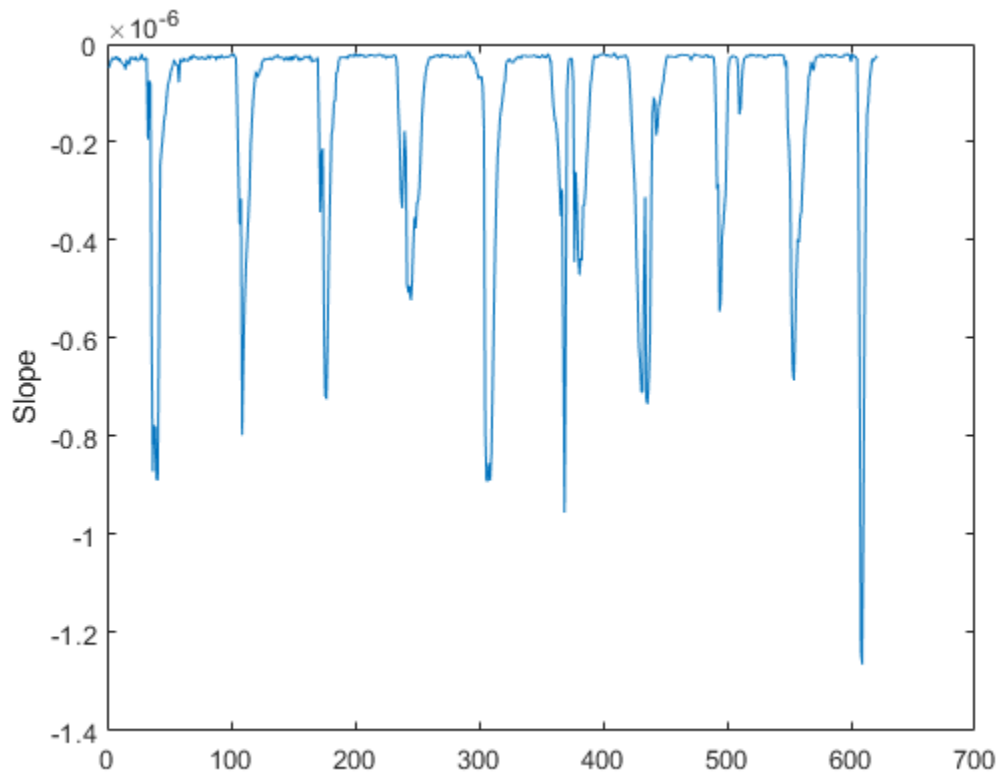
        slope = spectralSlope(audioBuffered,fs, ...
                               'Window',win, ...
                               'OverlapLength',0);

        logger(slope)
    end
end

release(fileReader)

Plot the logged data.

plot(logger.Buffer)
ylabel('Slope')
```



## Input Arguments

### **x** — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets **x** depends on the shape of **f**.

Data Types: `single` | `double`

### **f** — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets  $x$  depends on the shape of  $f$ :

- If  $f$  is a scalar,  $x$  is interpreted as a time-domain signal, and  $f$  is interpreted as the sample rate. In this case,  $x$  must be a real vector or matrix. If  $x$  is specified as a matrix, the columns are interpreted as individual channels.
- If  $f$  is a vector,  $x$  is interpreted as a frequency-domain signal, and  $f$  is interpreted as the frequencies, in Hz, corresponding to the rows of  $x$ . In this case,  $x$  must be a real  $L$ -by- $M$ -by- $N$  array, where  $L$  is the number of spectral values at given frequencies of  $f$ ,  $M$  is the number of individual spectrums, and  $N$  is the number of channels.
- The number of rows of  $x$ ,  $L$ , must be equal to the number of elements of  $f$ .

Data Types: `single` | `double`

### Name-Value Pair Arguments

---

**Note** The following name-value pair arguments apply if  $x$  is a time-domain signal. If  $x$  is a frequency-domain signal, name-value pair arguments are ignored.

---

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

Example: `'Window', hamming(256)`

#### **Window — Window applied in time domain**

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range  $[1, \text{size}(x, 1)]$ . The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

#### **OverlapLength — Number of samples overlapped between adjacent windows**

`round(f*0.02)` (default) | non-negative scalar



Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of 'OverlapLength' and an integer in the range [0, size(Window,1)].

Data Types: single | double

### **FFTLength — Number of bins in DFT**

numel(Window) (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as the comma-separated pair consisting of 'FFTLength' and a positive scalar integer. If unspecified, FFTLength defaults to the number of elements in the Window.

Data Types: single | double

### **Range — Frequency range (Hz)**

[0, f/2] (default) | two-element row vector

Frequency range in Hz, specified as the comma-separated pair consisting of 'Range' and a two-element row vector of increasing real values in the range [0, f/2].

Data Types: single | double

### **SpectrumType — Spectrum type**

'magnitude' (default) | 'power'

Spectrum type, specified as the comma-separated pair consisting of 'SpectrumType' and 'power' or 'magnitude':

- 'power' -- The spectral slope is calculated for the one-sided power spectrum.
- 'magnitude' -- The spectral slope is calculated for the one-sided magnitude spectrum.

Data Types: char | string

## **Output Arguments**

### **slope — Spectral slope**

scalar | vector | matrix

Spectral slope in Hz, returned as a scalar, vector, or matrix. Each row of `slope` corresponds to the spectral slope of a window of `x`. Each column of `slope` corresponds to an independent channel.

## Algorithms

The spectral slope is calculated as described in [1]:

$$\text{slope} = \frac{\sum_{k=b_1}^{b_2} (f_k - \mu_f)(s_k - \mu_s)}{\sum_{k=b_1}^{b_2} (f_k - \mu_f)^2}$$

where

- $f_k$  is the frequency in Hz corresponding to bin  $k$ .
- $\mu_f$  is the mean frequency.
- $s_k$  is the spectral value at bin  $k$ .
- $\mu_s$  is the mean spectral value.
- $b_1$  and  $b_2$  are the band edges, in bins, over which to calculate the spectral slope.

## References

- [1] Lerch, Alexander. *An Introduction to Audio Content Analysis Applications in Signal Processing and Music Informatics*. Piscataway, NJ: IEEE Press, 2012.

## Extended Capabilities

### C/C++ Code Generation

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

## **See Also**

spectralCrest | spectralDecrease

## **Topics**

“Spectral Descriptors”

**Introduced in R2019a**

## spectralSkewness

Spectral skewness for audio signals and auditory spectrograms

### Syntax

```
skewness = spectralSkewness(x,f)  
skewness = spectralSkewness(x,f,Name,Value)  
[skewness,spread,centroid] = spectralSkewness( ___ )
```

### Description

`skewness = spectralSkewness(x,f)` returns the spectral skewness of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`skewness = spectralSkewness(x,f,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

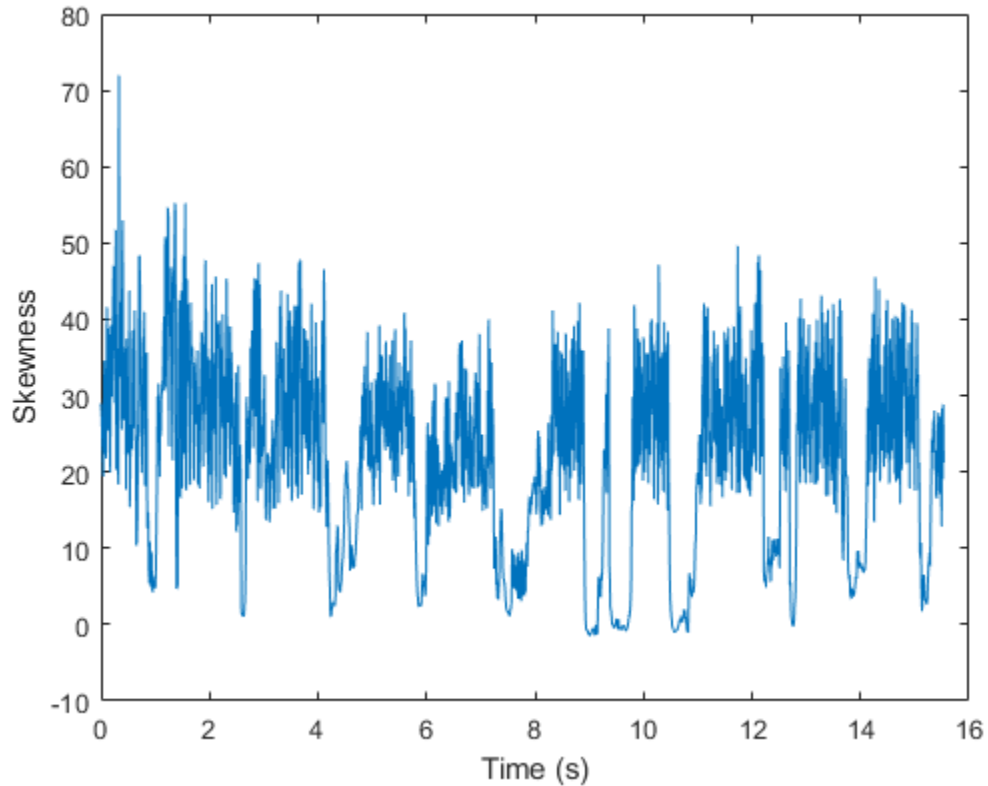
`[skewness,spread,centroid] = spectralSkewness( ___ )` returns the spectral spread and spectral centroid.

### Examples

#### Spectral Skewness of Time-Domain Audio

Read in an audio file, calculate the skewness using default parameters, and then plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
skewness = spectralSkewness(audioIn,fs);  
  
t = linspace(0,size(audioIn,1)/fs,size(skewness,1));  
plot(t,skewness)  
xlabel('Time (s)')  
ylabel('Skewness')
```



### **Spectral Skewness of Frequency-Domain Audio Data**

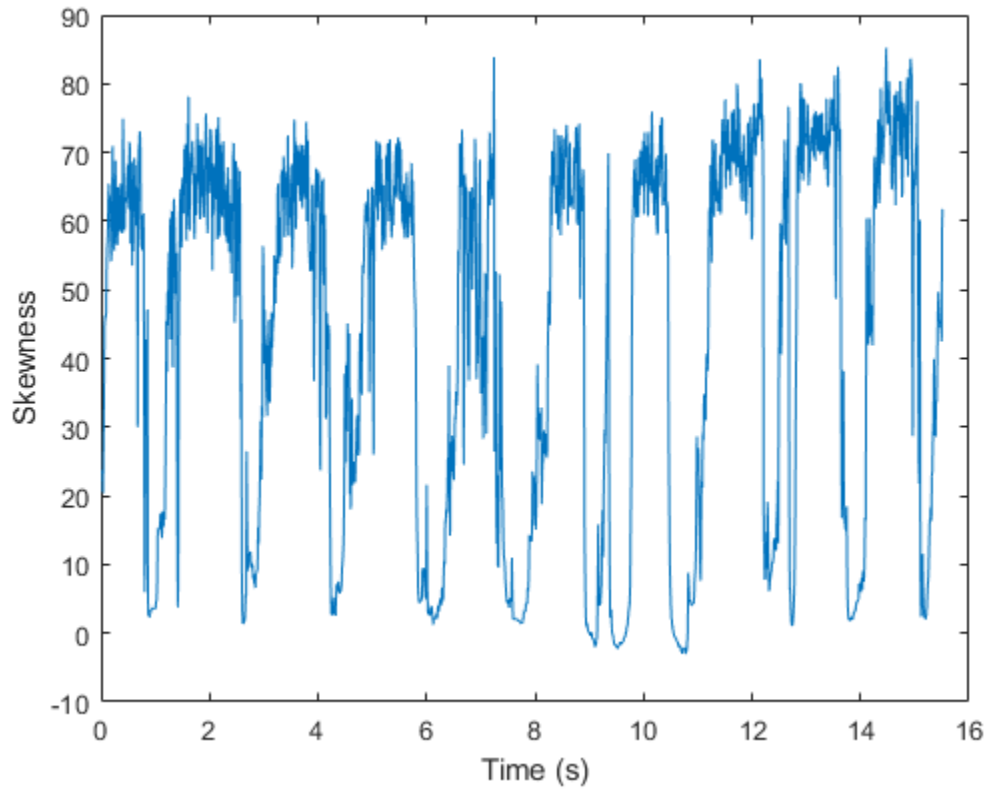
Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function. Calculate the skewness of the mel spectrogram over time. Plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

```
[s,cf,t] = melSpectrogram(audioIn,fs);
```

```
skewness = spectralSkewness(s,cf);
```

```
plot(t,skewness)
xlabel('Time (s)')
ylabel('Skewness')
```



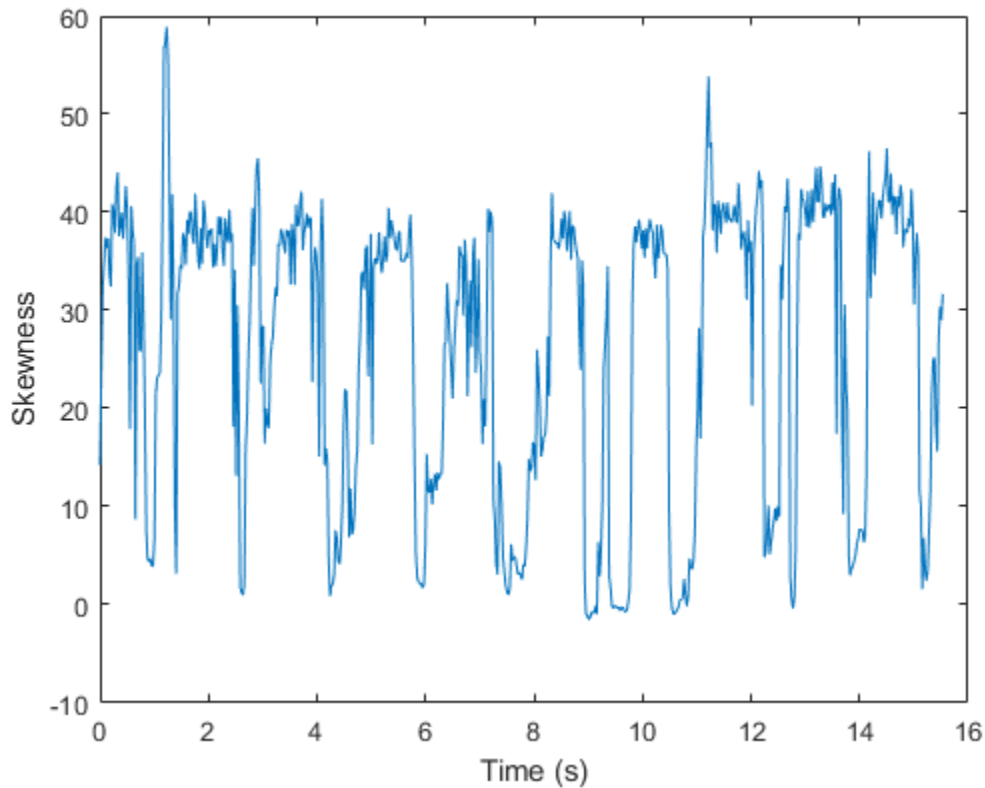
### Specify Nondefault Parameters

Read in an audio file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Calculate the skewness of the power spectrum over time. Calculate the skewness for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to  $f_s/2$  for the skewness calculation. Plot the results.

```
skewness = spectralSkewness(audioIn,fs, ...  
    'Window',hamming(round(0.05*fs)), ...  
    'OverlapLength',round(0.025*fs), ...  
    'Range',[62.5,fs/2]);  
  
t = linspace(0,size(audioIn,1)/fs,size(skewness,1));  
plot(t,skewness)  
xlabel('Time (s)')  
ylabel('Skewness')
```



### Calculate Spectral Skewness of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral skewness calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44pl-mono-15secs.wav');  
logger = dsp.SignalSink;
```

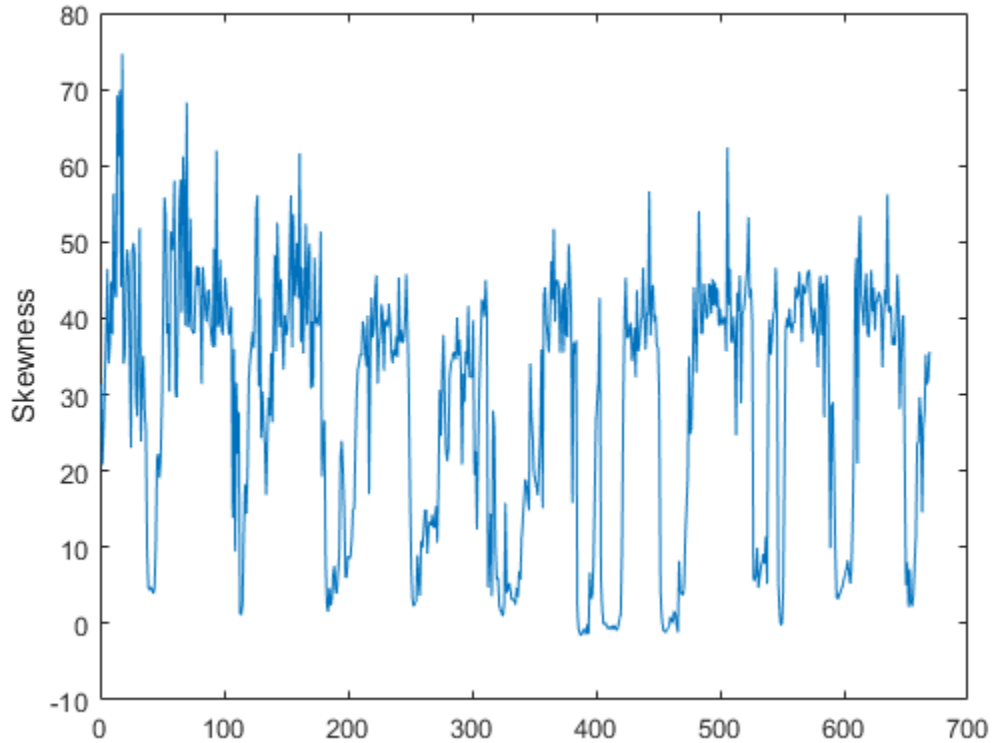
In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Calculate the spectral skewness for the frame of audio.
- 3 Log the spectral skewness for later plotting.

To calculate the spectral skewness for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame);  
while ~isDone(fileReader)  
    audioIn = fileReader();  
    skewness = spectralSkewness(audioIn,fileReader.SampleRate, ...  
                               'Window',win, ...  
                               'OverlapLength',0);  
    logger(skewness)  
end  
  
plot(logger.Buffer)  
ylabel('Skewness')
```





Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralSkewness`.
- You want to calculate the spectral skewness for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;  
reset(logger)  
release(fileReader)
```

Specify that the spectral skewness is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

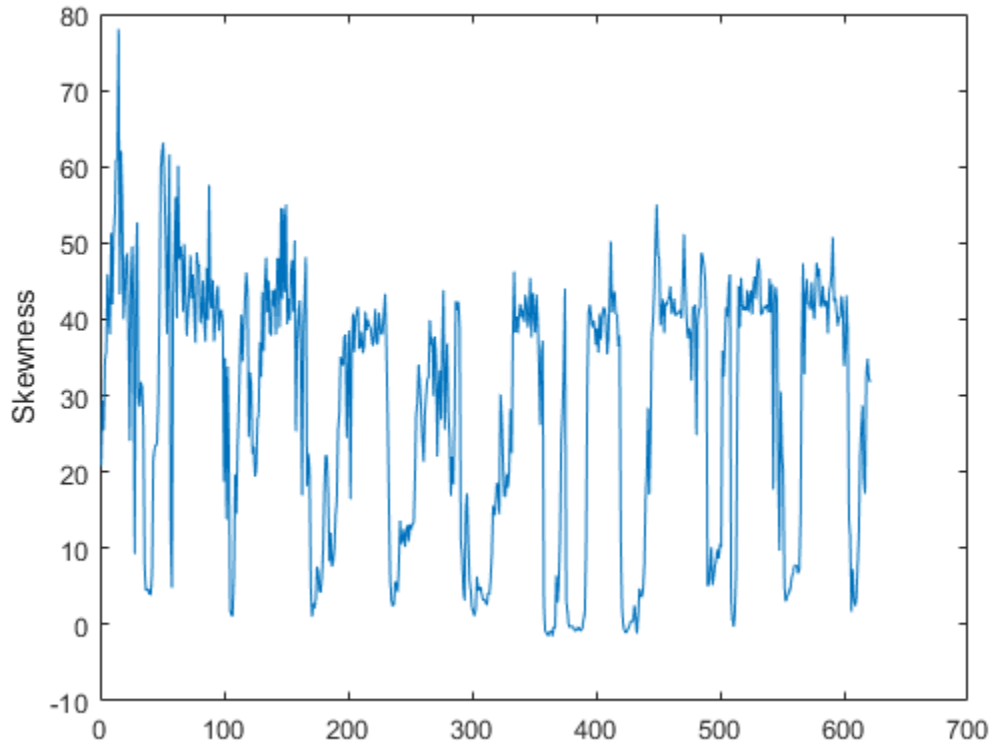
        skewness = spectralSkewness(audioBuffered,fs, ...
                                    'Window',win, ...
                                    'OverlapLength',0);

        logger(skewness)
    end

end
release(fileReader)

Plot the logged data.

plot(logger.Buffer)
ylabel('Skewness')
```



## Input Arguments

### **x** — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets **x** depends on the shape of **f**.

Data Types: `single` | `double`

### **f** — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets  $x$  depends on the shape of  $f$ :

- If  $f$  is a scalar,  $x$  is interpreted as a time-domain signal, and  $f$  is interpreted as the sample rate. In this case,  $x$  must be a real vector or matrix. If  $x$  is specified as a matrix, the columns are interpreted as individual channels.
- If  $f$  is a vector,  $x$  is interpreted as a frequency-domain signal, and  $f$  is interpreted as the frequencies, in Hz, corresponding to the rows of  $x$ . In this case,  $x$  must be a real  $L$ -by- $M$ -by- $N$  array, where  $L$  is the number of spectral values at given frequencies of  $f$ ,  $M$  is the number of individual spectrums, and  $N$  is the number of channels.
- The number of rows of  $x$ ,  $L$ , must be equal to the number of elements of  $f$ .

Data Types: `single` | `double`

### Name-Value Pair Arguments

---

**Note** The following name-value pair arguments apply if  $x$  is a time-domain signal. If  $x$  is a frequency-domain signal, name-value pair arguments are ignored.

---

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

Example: `'Window', hamming(256)`

#### **Window — Window applied in time domain**

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range `[1, size(x,1)]`. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

#### **OverlapLength — Number of samples overlapped between adjacent windows**

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of 'OverlapLength' and an integer in the range [0, size(Window,1)).

Data Types: single | double

### **FFTLength — Number of bins in DFT**

numel(Window) (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as the comma-separated pair consisting of 'FFTLength' and a positive scalar integer. If unspecified, FFTLength defaults to the number of elements in the Window.

Data Types: single | double

### **Range — Frequency range (Hz)**

[0, f/2] (default) | two-element row vector

Frequency range in Hz, specified as the comma-separated pair consisting of 'Range' and a two-element row vector of increasing real values in the range [0, f/2].

Data Types: single | double

### **SpectrumType — Spectrum type**

'power' (default) | 'magnitude'

Spectrum type, specified as the comma-separated pair consisting of 'SpectrumType' and 'power' or 'magnitude':

- 'power' -- The spectral skewness is calculated for the one-sided power spectrum.
- 'magnitude' -- The spectral skewness is calculated for the one-sided magnitude spectrum.

Data Types: char | string

## **Output Arguments**

### **skewness — Spectral skewness**

scalar | vector | matrix

Spectral skewness, returned as a scalar, vector, or matrix. Each row of `skewness` corresponds to the spectral skewness of a window of `x`. Each column of `skewness` corresponds to an independent channel.

**spread — Spectral spread**

scalar | vector | matrix

Spectral spread, returned as a scalar, vector, or matrix. Each row of `spread` corresponds to the spectral spread of a window of `x`. Each column of `spread` corresponds to an independent channel.

**centroid — Spectral centroid (Hz)**

scalar | vector | matrix

Spectral centroid in Hz, returned as a scalar, vector, or matrix. Each row of `centroid` corresponds to the spectral centroid of a window of `x`. Each column of `centroid` corresponds to an independent channel.

## Algorithms

The spectral skewness is calculated as described in [1]:

$$\text{skewness} = \frac{\sum_{k=b_1}^{b_2} (f_k - \mu_1)^3 s_k}{(\mu_2)^3 \sum_{k=b_1}^{b_2} s_k}$$

where

- $f_k$  is the frequency in Hz corresponding to bin  $k$ .
- $s_k$  is the spectral value at bin  $k$ .
- $b_1$  and  $b_2$  are the band edges, in bins, over which to calculate the spectral skewness.
- $\mu_1$  is the spectral centroid, calculated as described by the `spectralCentroid` function.
- $\mu_2$  is the spectral spread, calculated as described by the `spectralSpread` function.

## References

- [1] Peeters, G. "A Large Set of Audio Features for Sound Description (Similarity and Classification) in the CUIDADO Project." Technical Report; IRCAM: Paris, France, 2004.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[spectralCentroid](#) | [spectralKurtosis](#) | [spectralSpread](#)

### Topics

"Spectral Descriptors"

**Introduced in R2019a**

## spectralRolloffPoint

Spectral rolloff point for audio signals and auditory spectrograms

### Syntax

```
rolloffPoint = spectralRolloffPoint(x,f)  
rolloffPoint = spectralRolloffPoint(x,f,Name,Value)
```

### Description

`rolloffPoint = spectralRolloffPoint(x,f)` returns the spectral rolloff point of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`rolloffPoint = spectralRolloffPoint(x,f,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

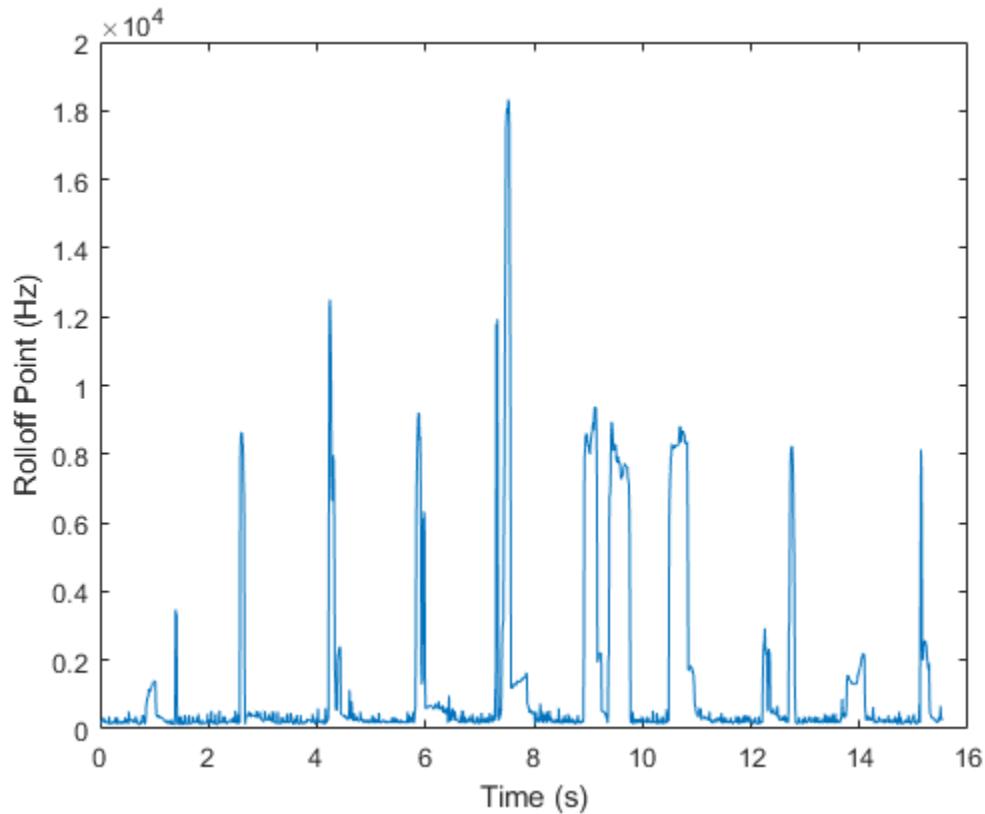
### Examples

#### Spectral Rolloff Point of Time-Domain Audio

Read in an audio file, calculate the rolloff point using default parameters, and then plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
rolloffPoint = spectralRolloffPoint(audioIn,fs);  
  
t = linspace(0,size(audioIn,1)/fs,size(rolloffPoint,1));  
plot(t,rolloffPoint)  
xlabel('Time (s)')  
ylabel('Rolloff Point (Hz)')
```



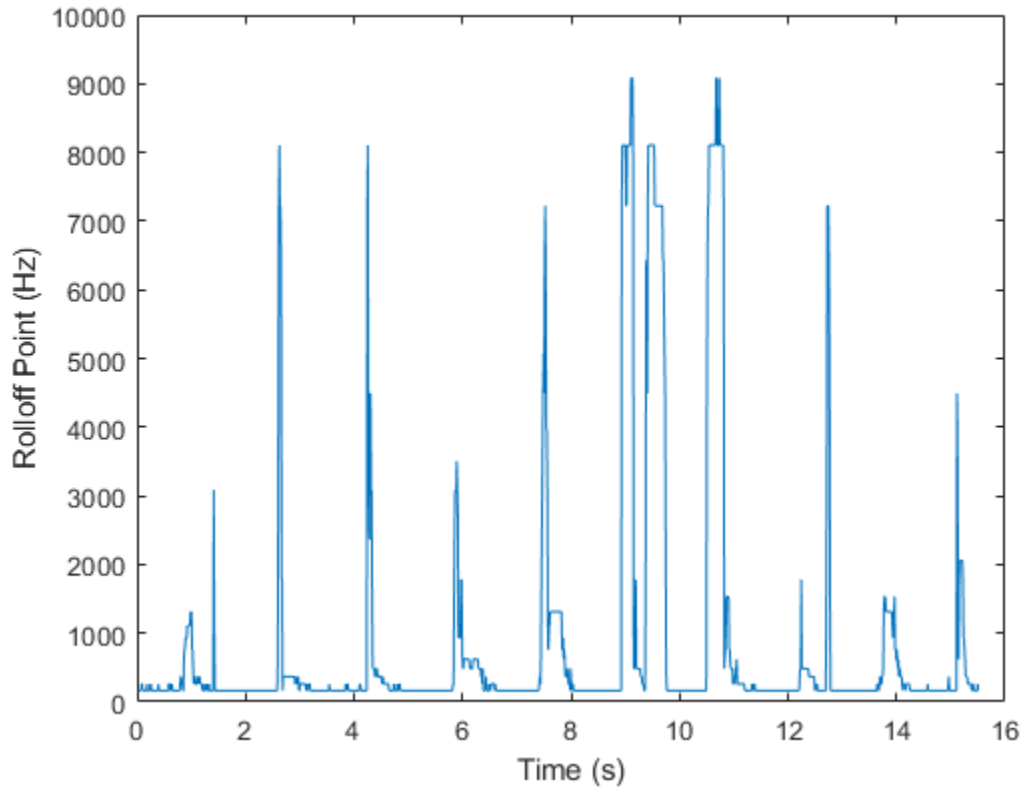


### Spectral Rolloff Point of Frequency-Domain Audio Data

Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function. Calculate the rolloff point of the mel spectrogram over time. Plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
[s,cf,t] = melSpectrogram(audioIn,fs);  
rolloffPoint = spectralRolloffPoint(s,cf);
```

```
plot(t,rolloffPoint)
xlabel('Time (s)')
ylabel('Rolloff Point (Hz)')
```



### Specify Nondefault Parameters

Read in an audio file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

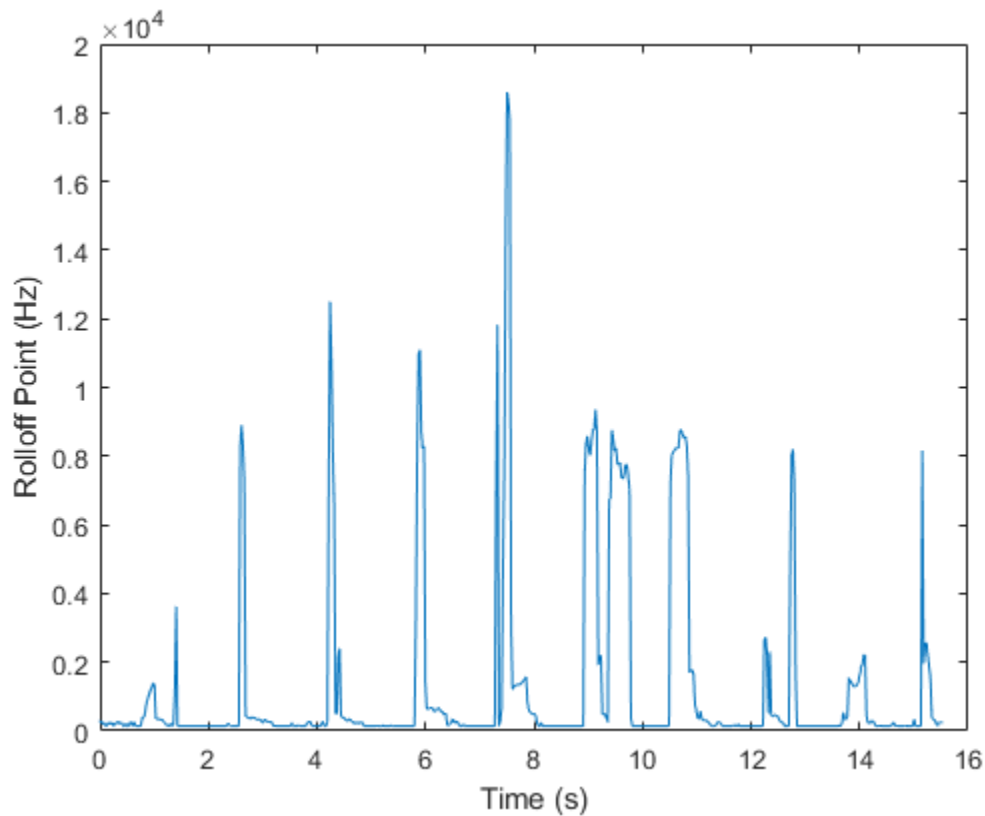
Calculate the rolloff point of the power spectrum over time. Calculate the rolloff point for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to  $f_s/2$  for the rolloff point calculation. Plot the results.

```

rolloffPoint = spectralRolloffPoint(audioIn,fs, ...
    'Window',hamming(round(0.05*fs)), ...
    'OverlapLength',round(0.025*fs), ...
    'Range',[62.5,fs/2]);

t = linspace(0,size(audioIn,1)/fs,size(rolloffPoint,1));
plot(t,rolloffPoint)
xlabel('Time (s)')
ylabel('Rolloff Point (Hz)')

```



### Calculate Spectral Rolloff Point of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral rolloff point calculation.

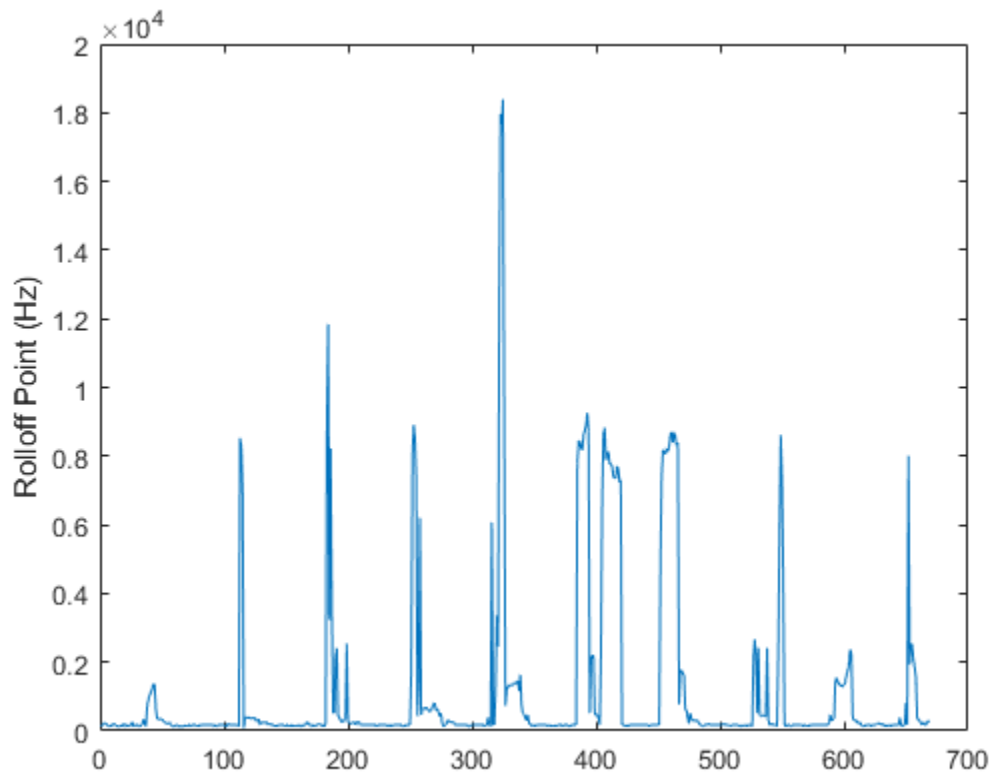
```
fileReader = dsp.AudioFileReader('Counting-16-44pl-mono-15secs.wav');  
logger = dsp.SignalSink;
```

In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Calculate the spectral rolloff point for the frame of audio.
- 3 Log the spectral rolloff point for later plotting.

To calculate the spectral rolloff point for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame);  
while ~isDone(fileReader)  
    audioIn = fileReader();  
    rolloffPoint = spectralRolloffPoint(audioIn,fileReader.SampleRate, ...  
                                       'Window',win, ...  
                                       'OverlapLength',0);  
    logger(rolloffPoint)  
end  
  
plot(logger.Buffer)  
ylabel('Rolloff Point (Hz)')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralRolloffPoint`.
- You want to calculate the spectral rolloff point for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;
reset(logger)
release(fileReader)
```

Specify that the spectral rolloff point is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

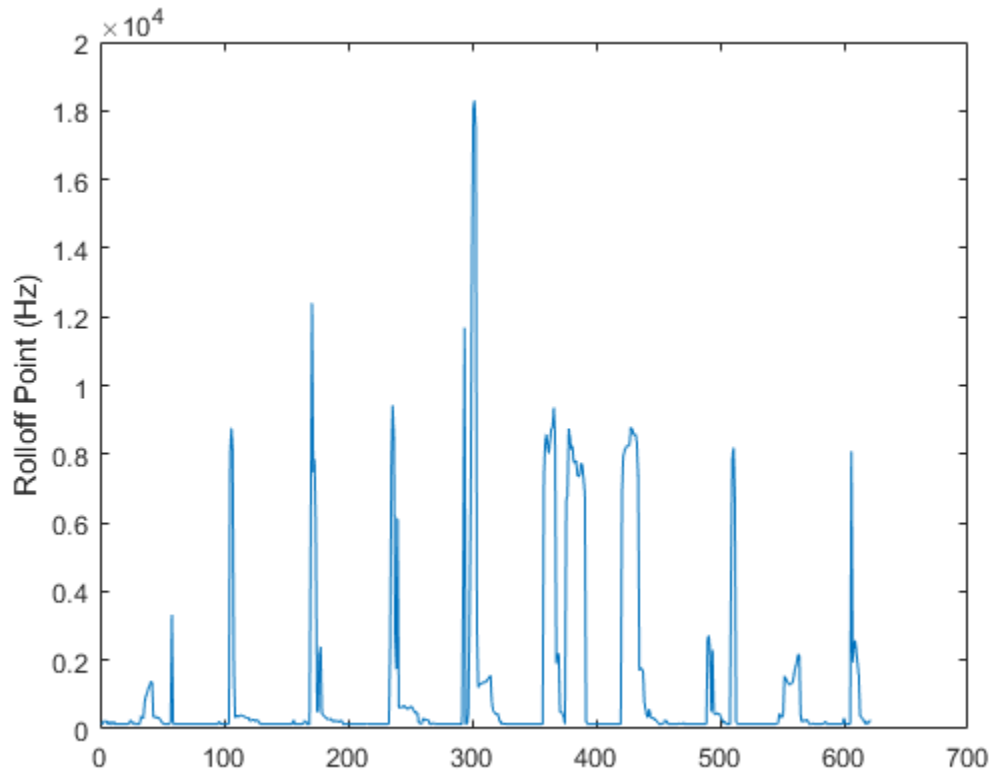
        rolloffPoint = spectralRolloffPoint(audioBuffered,fs, ...
            'Window',win, ...
            'OverlapLength',0);

        logger(rolloffPoint)
    end

end
release(fileReader)

Plot the logged data.

plot(logger.Buffer)
ylabel('Rolloff Point (Hz)')
```



## Input Arguments

### **x** — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets **x** depends on the shape of **f**.

Data Types: `single` | `double`

### **f** — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets `x` depends on the shape of `f`:

- If `f` is a scalar, `x` is interpreted as a time-domain signal, and `f` is interpreted as the sample rate. In this case, `x` must be a real vector or matrix. If `x` is specified as a matrix, the columns are interpreted as individual channels.
- If `f` is a vector, `x` is interpreted as a frequency-domain signal, and `f` is interpreted as the frequencies, in Hz, corresponding to the rows of `x`. In this case, `x` must be a real  $L$ -by- $M$ -by- $N$  array, where  $L$  is the number of spectral values at given frequencies of `f`,  $M$  is the number of individual spectrums, and  $N$  is the number of channels.
- The number of rows of `x`,  $L$ , must be equal to the number of elements of `f`.

Data Types: `single` | `double`

### 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Window', hamming(256)`

#### **Threshold** — Threshold of rolloff point

`0.95` (default) | scalar in the range (0,1)

Threshold of rolloff point, specified as the comma-separated pair consisting of `'Threshold'` and a scalar between zero and one, exclusive.

Data Types: `single` | `double`

---

**Note** The following name-value pair arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, name-value pair arguments are ignored.

---

#### **Window** — Window applied in time domain

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range



[1, size(x,1)]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

### **OverlapLength — Number of samples overlapped between adjacent windows**

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of `'OverlapLength'` and an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

### **FFTLength — Number of bins in DFT**

`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as the comma-separated pair consisting of `'FFTLength'` and a positive scalar integer. If unspecified, `FFTLength` defaults to the number of elements in the `Window`.

Data Types: `single` | `double`

### **Range — Frequency range (Hz)**

`[0, f/2]` (default) | two-element row vector

Frequency range in Hz, specified as the comma-separated pair consisting of `'Range'` and a two-element row vector of increasing real values in the range [0, `f/2`].

Data Types: `single` | `double`

### **SpectrumType — Spectrum type**

`'power'` (default) | `'magnitude'`

Spectrum type, specified as the comma-separated pair consisting of `'SpectrumType'` and `'power'` or `'magnitude'`:

- `'power'` -- The spectral rolloff point is calculated for the one-sided power spectrum.
- `'magnitude'` -- The spectral rolloff point is calculated for the one-sided magnitude spectrum.

Data Types: `char` | `string`

## Output Arguments

### **rolloffPoint** — Spectral rolloff point (Hz)

scalar | vector | matrix

Spectral rolloff point in Hz, returned as a scalar, vector, or matrix. Each row of `rolloffPoint` corresponds to the spectral rolloff point of a window of `x`. Each column of `rolloffPoint` corresponds to an independent channel.

## Algorithms

The spectral rolloff point is calculated as described in [1]:

$$\text{rolloffPoint} = i$$

such that

$$\sum_{k=b_1}^i s_k = \kappa \sum_{k=b_1}^{b_2} s_k$$

where

- $s_k$  is the spectral value at bin  $k$ .
- $b_1$  and  $b_2$  are the band edges, in bins, over which to calculate the spectral spread.
- $\kappa$  is the percentage of total energy contained between  $b_1$  and  $i$ . You can set  $\kappa$  using `Threshold`.

## References

- [1] Scheirer, E., and M. Slaney, "Construction and Evaluation of a Robust Multifeature Speech/Music Discriminator," *IEEE International Conference on Acoustics, Speech, and Signal Processing*. Volume 2, 1997, pp. 1221-1224.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[spectralKurtosis](#) | [spectralSkewness](#) | [spectralSpread](#)

### Topics

“Spectral Descriptors”

**Introduced in R2019a**

## spectralKurtosis

Spectral kurtosis for audio signals and auditory spectrograms

### Syntax

```
kurtosis = spectralKurtosis(x,f)  
kurtosis = spectralKurtosis(x,f,Name,Value)  
[kurtosis,spread,centroid] = spectralKurtosis( ___ )
```

### Description

`kurtosis = spectralKurtosis(x,f)` returns the spectral kurtosis of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`kurtosis = spectralKurtosis(x,f,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

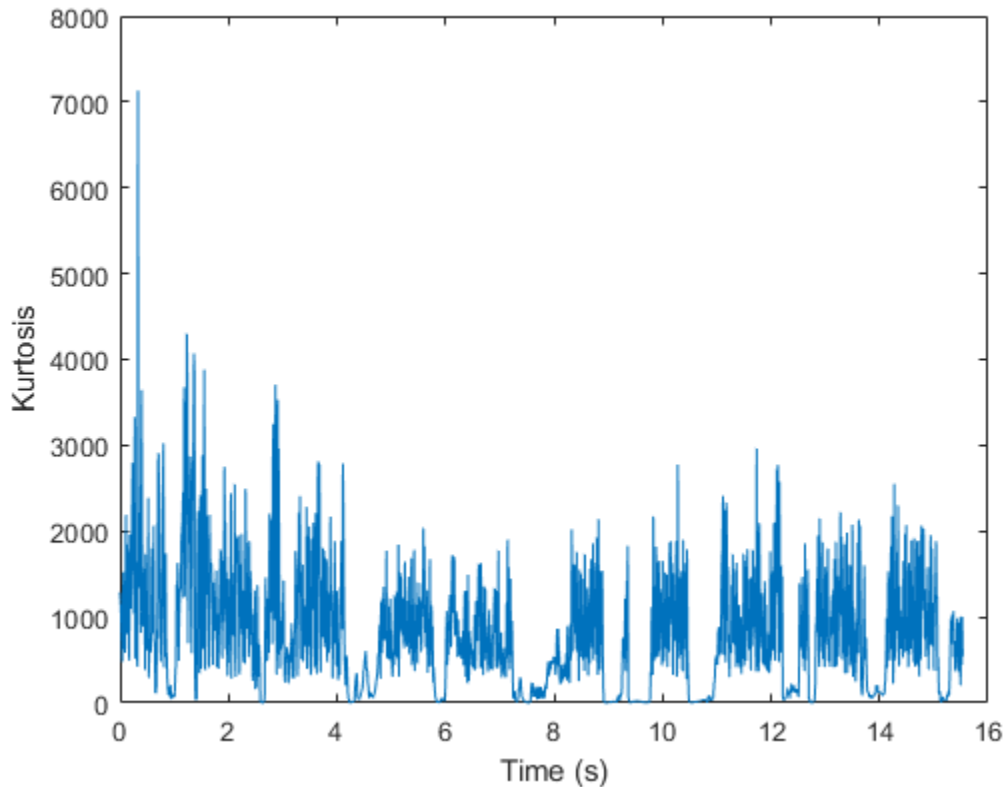
`[kurtosis,spread,centroid] = spectralKurtosis( ___ )` returns the spectral spread and spectral centroid.

### Examples

#### Spectral Kurtosis of Time-Domain Audio

Read in an audio file, calculate the kurtosis using default parameters, and then plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
kurtosis = spectralKurtosis(audioIn,fs);  
  
t = linspace(0,size(audioIn,1)/fs,size(kurtosis,1));  
plot(t,kurtosis)  
xlabel('Time (s)')  
ylabel('Kurtosis')
```



### Spectral Kurtosis of Frequency-Domain Audio Data

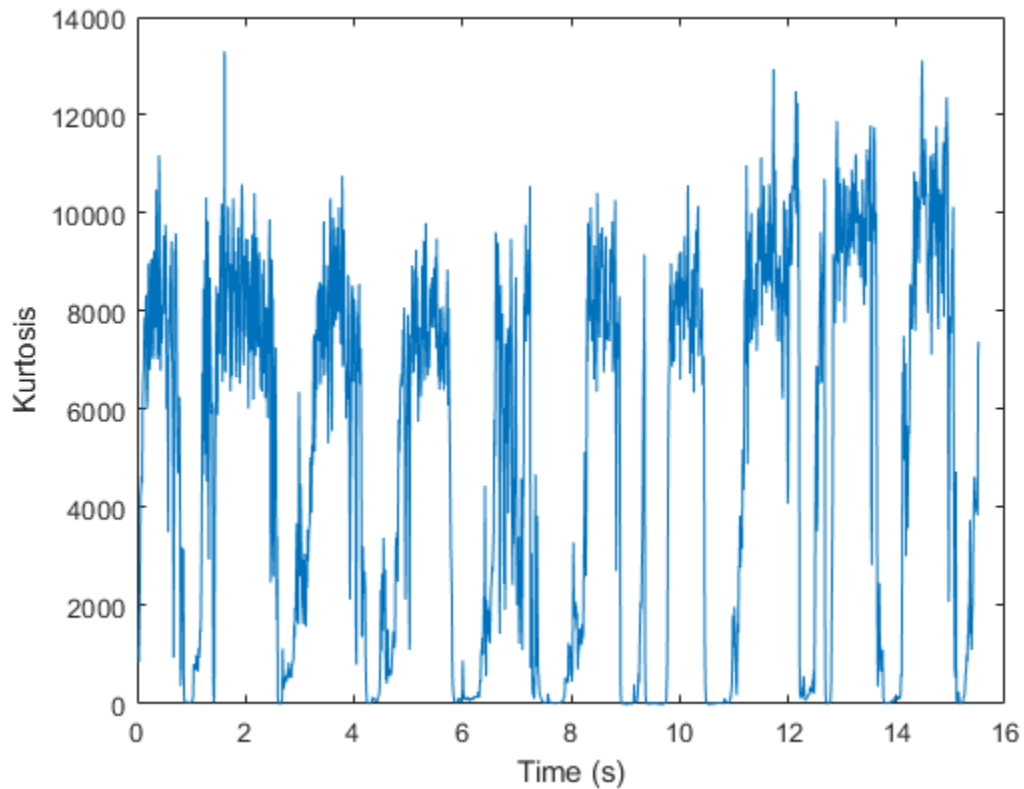
Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function. Calculate the kurtosis of the mel spectrogram over time. Plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

```
[s,cf,t] = melSpectrogram(audioIn,fs);
```

```
kurtosis = spectralKurtosis(s,cf);
```

```
plot(t,kurtosis)
xlabel('Time (s)')
ylabel('Kurtosis')
```



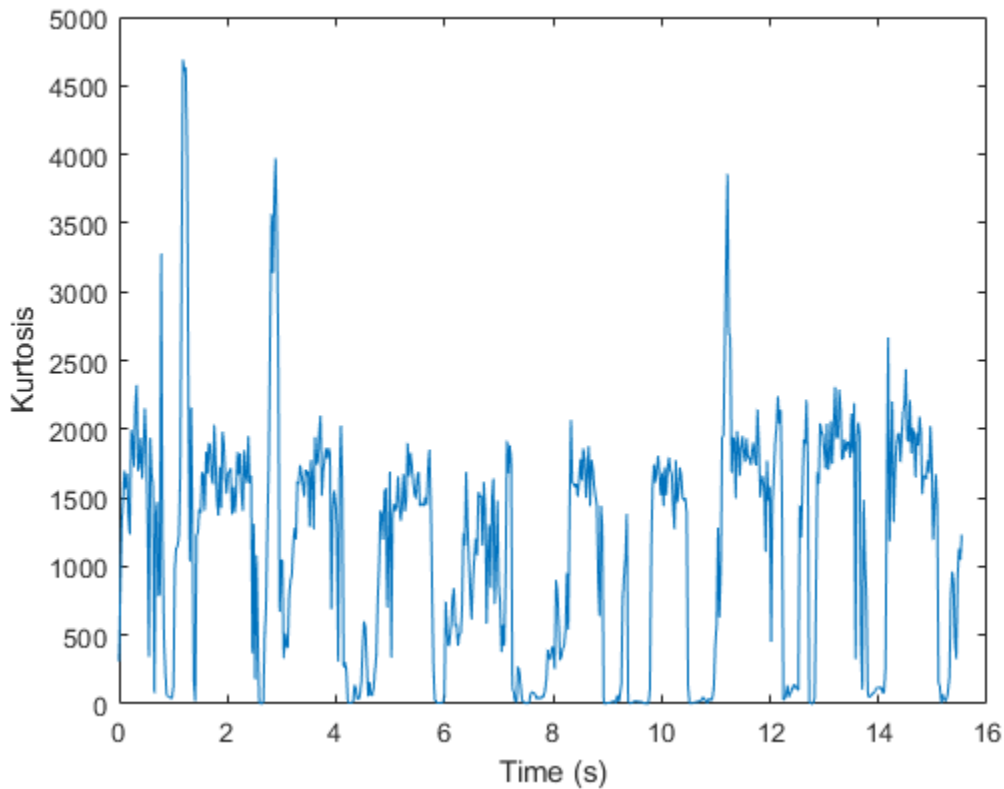
### Specify Nondefault Parameters

Read in an audio file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Calculate the kurtosis of the power spectrum over time. Calculate the kurtosis for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to  $f_s/2$  for the kurtosis calculation. Plot the results.

```
kurtosis = spectralKurtosis(audioIn,fs, ...  
                           'Window',hamming(round(0.05*fs)), ...  
                           'OverlapLength',round(0.025*fs), ...  
                           'Range',[62.5,fs/2]);  
  
t = linspace(0,size(audioIn,1)/fs,size(kurtosis,1));  
plot(t,kurtosis)  
xlabel('Time (s)')  
ylabel('Kurtosis')
```



### Calculate Spectral Kurtosis of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral kurtosis calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44pl-mono-15secs.wav');  
logger = dsp.SignalSink;
```

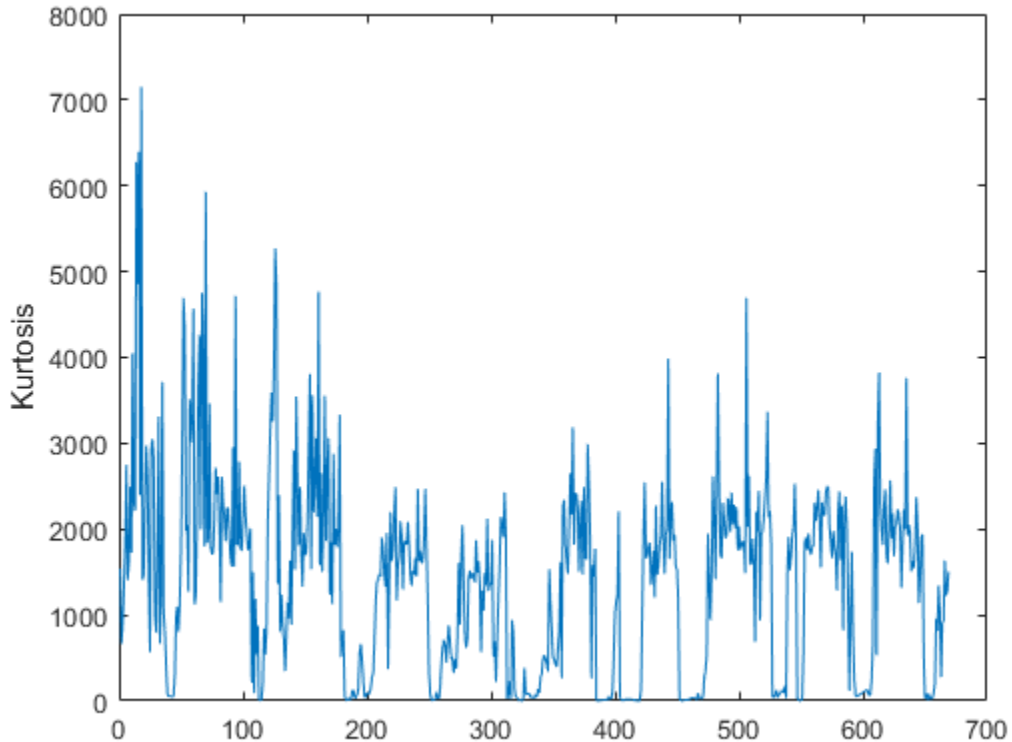
In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Calculate the spectral kurtosis for the frame of audio.
- 3 Log the spectral kurtosis for later plotting.

To calculate the spectral kurtosis for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame);  
while ~isDone(fileReader)  
    audioIn = fileReader();  
    kurtosis = spectralKurtosis(audioIn,fileReader.SampleRate, ...  
                              'Window',win, ...  
                              'OverlapLength',0);  
    logger(kurtosis)  
end  
  
plot(logger.Buffer)  
ylabel('Kurtosis')
```





Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralKurtosis`.
- You want to calculate the spectral kurtosis for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;  
reset(logger)  
release(fileReader)
```

Specify that the spectral kurtosis is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

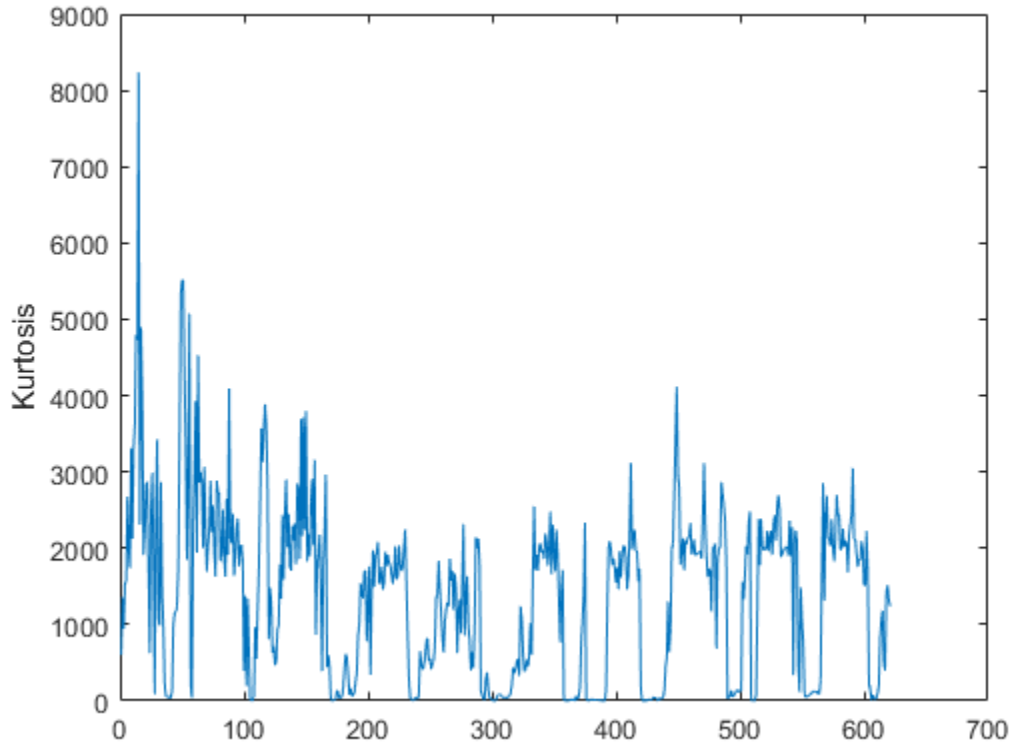
        kurtosis = spectralKurtosis(audioBuffered,fs, ...
                                   'Window',win, ...
                                   'OverlapLength',0);

        logger(kurtosis)
    end

end
release(fileReader)

Plot the logged data.

plot(logger.Buffer)
ylabel('Kurtosis')
```



## Input Arguments

### **x** — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets  $x$  depends on the shape of  $f$ .

Data Types: `single` | `double`

### **f** — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets  $x$  depends on the shape of  $f$ :

- If  $f$  is a scalar,  $x$  is interpreted as a time-domain signal, and  $f$  is interpreted as the sample rate. In this case,  $x$  must be a real vector or matrix. If  $x$  is specified as a matrix, the columns are interpreted as individual channels.
- If  $f$  is a vector,  $x$  is interpreted as a frequency-domain signal, and  $f$  is interpreted as the frequencies, in Hz, corresponding to the rows of  $x$ . In this case,  $x$  must be a real  $L$ -by- $M$ -by- $N$  array, where  $L$  is the number of spectral values at given frequencies of  $f$ ,  $M$  is the number of individual spectrums, and  $N$  is the number of channels.
- The number of rows of  $x$ ,  $L$ , must be equal to the number of elements of  $f$ .

Data Types: `single` | `double`

### Name-Value Pair Arguments

---

**Note** The following name-value pair arguments apply if  $x$  is a time-domain signal. If  $x$  is a frequency-domain signal, name-value pair arguments are ignored.

---

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

Example: `'Window', hamming(256)`

#### **Window — Window applied in time domain**

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range `[1, size(x,1)]`. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

#### **OverlapLength — Number of samples overlapped between adjacent windows**

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of 'OverlapLength' and an integer in the range [0, size(Window,1)).

Data Types: single | double

### **FFTLength — Number of bins in DFT**

numel(Window) (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as the comma-separated pair consisting of 'FFTLength' and a positive scalar integer. If unspecified, FFTLength defaults to the number of elements in the Window.

Data Types: single | double

### **Range — Frequency range (Hz)**

[0, f/2] (default) | two-element row vector

Frequency range in Hz, specified as the comma-separated pair consisting of 'Range' and a two-element row vector of increasing real values in the range [0, f/2].

Data Types: single | double

### **SpectrumType — Spectrum type**

'power' (default) | 'magnitude'

Spectrum type, specified as the comma-separated pair consisting of 'SpectrumType' and 'power' or 'magnitude':

- 'power' -- The spectral kurtosis is calculated for the one-sided power spectrum.
- 'magnitude' -- The spectral kurtosis is calculated for the one-sided magnitude spectrum.

Data Types: char | string

## **Output Arguments**

### **kurtosis — Spectral kurtosis**

scalar | vector | matrix

Spectral kurtosis, returned as a scalar, vector, or matrix. Each row of `kurtosis` corresponds to the spectral kurtosis of a window of `x`. Each column of `kurtosis` corresponds to an independent channel.

**spread — Spectral spread**

scalar | vector | matrix

Spectral spread, returned as a scalar, vector, or matrix. Each row of `spread` corresponds to the spectral spread of a window of `x`. Each column of `spread` corresponds to an independent channel.

**centroid — Spectral centroid (Hz)**

scalar | vector | matrix

Spectral centroid in Hz, returned as a scalar, vector, or matrix. Each row of `centroid` corresponds to the spectral centroid of a window of `x`. Each column of `centroid` corresponds to an independent channel.

## Algorithms

The spectral kurtosis is calculated as described in [1]:

$$\text{kurtosis} = \frac{\sum_{k=b_1}^{b_2} (f_k - \mu_1)^4 s_k}{(\mu_2)^4 \sum_{k=b_1}^{b_2} s_k}$$

where

- $f_k$  is the frequency in Hz corresponding to bin  $k$ .
- $s_k$  is the spectral value at bin  $k$ .
- $b_1$  and  $b_2$  are the band edges, in bins, over which to calculate the spectral skewness.
- $\mu_1$  is the spectral centroid, calculated as described by the `spectralCentroid` function.
- $\mu_2$  is the spectral spread, calculated as described by the `spectralSpread` function.

## References

- [1] Peeters, G. "A Large Set of Audio Features for Sound Description (Similarity and Classification) in the CUIDADO Project." Technical Report; IRCAM: Paris, France, 2004.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`spectralCentroid` | `spectralSkewness` | `spectralSpread`

### Topics

"Spectral Descriptors"

**Introduced in R2019a**

## spectralFlux

Spectral flux for audio signals and auditory spectrograms

### Syntax

```
flux = spectralFlux(x,f)  
flux = spectralFlux(x,f,Name,Value)
```

### Description

`flux = spectralFlux(x,f)` returns the spectral flux of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`flux = spectralFlux(x,f,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

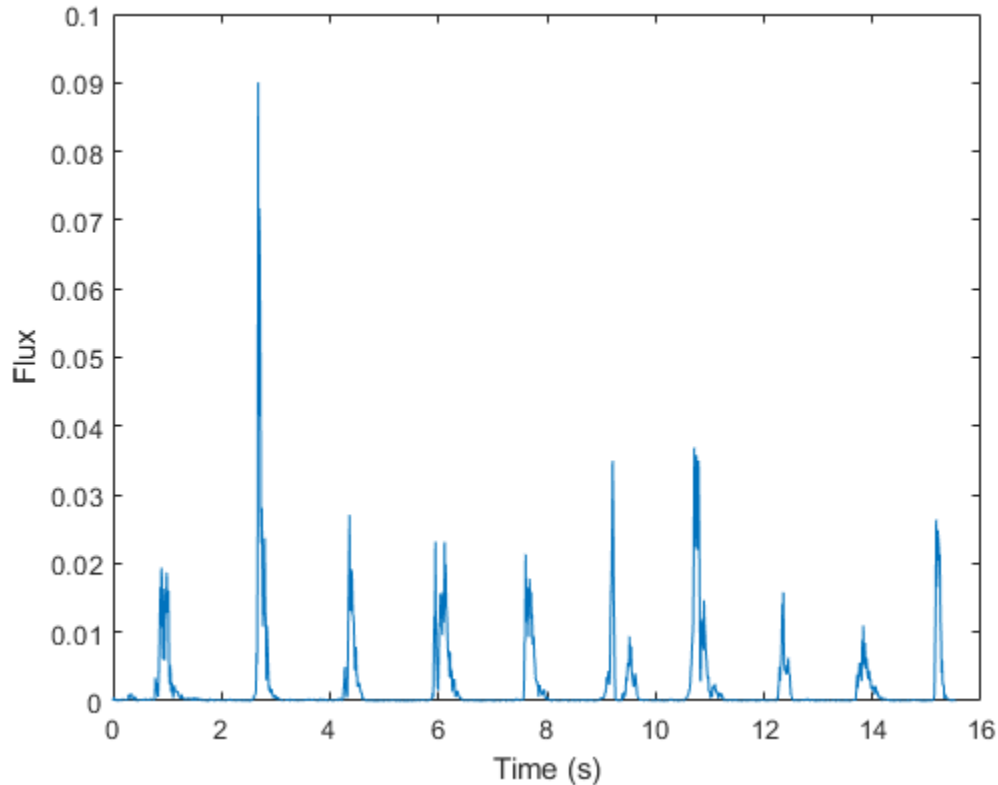
### Examples

#### Spectral Flux of Time-Domain Audio

Read in an audio file, calculate the flux using default parameters, and then plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
flux = spectralFlux(audioIn,fs);  
  
t = linspace(0,size(audioIn,1)/fs,size(flux,1));  
plot(t,flux)  
xlabel('Time (s)')  
ylabel('Flux')
```





### Spectral Flux of Frequency-Domain Audio Data

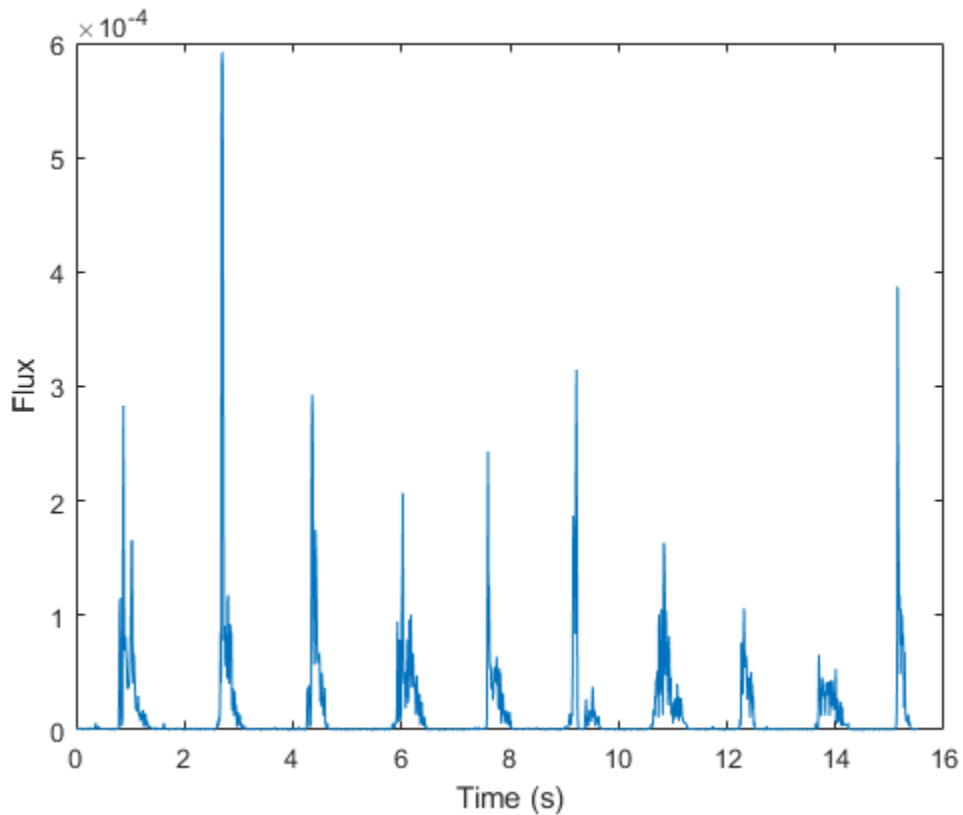
Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function. Calculate the flux of the mel spectrogram over time. Plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

```
[s,cf,t] = melSpectrogram(audioIn,fs);
```

```
flux = spectralFlux(s,cf);
```

```
plot(t,flux)
xlabel('Time (s)')
ylabel('Flux')
```



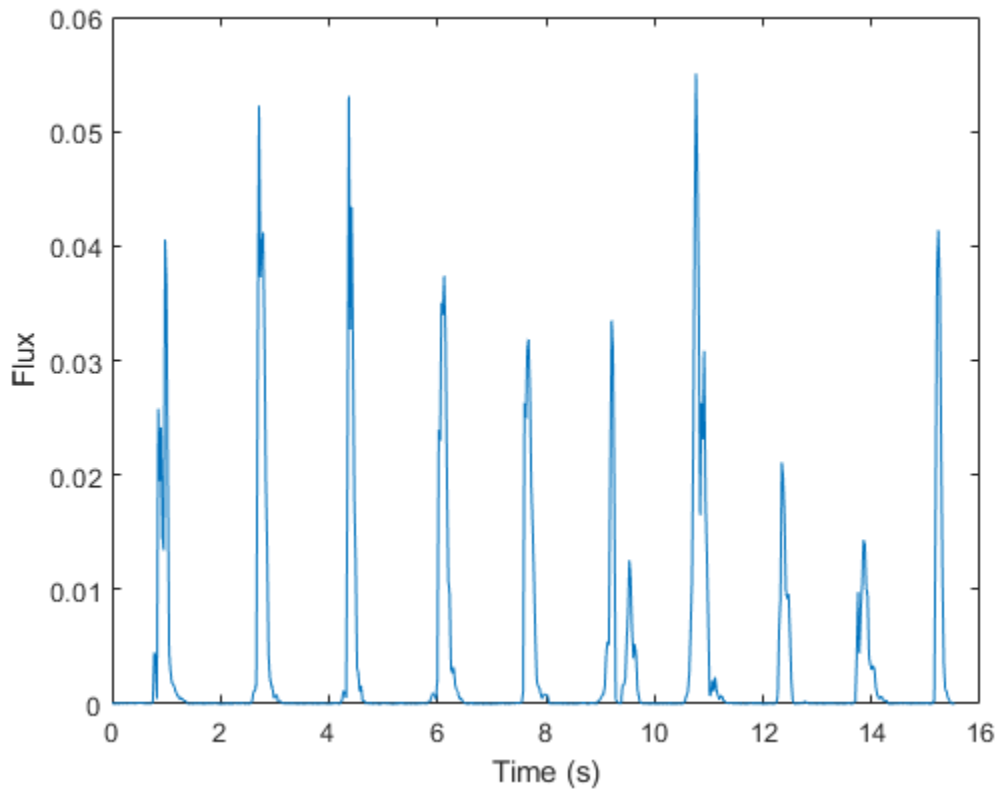
### Specify Nondefault Parameters

Read in an audio file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Calculate the flux of the power spectrum over time. Calculate the flux for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to  $f_s/2$  for the flux calculation. Plot the results.

```
flux = spectralFlux(audioIn,fs, ...  
                   'Window',hamming(round(0.05*fs)), ...  
                   'OverlapLength',round(0.025*fs), ...  
                   'Range',[62.5,fs/2]);  
  
t = linspace(0,size(audioIn,1)/fs,size(flux,1));  
plot(t,flux)  
xlabel('Time (s)')  
ylabel('Flux')
```



### Calculate Spectral Flux of Streaming Audio

Spectral flux measures the change in consecutive spectrums. To calculate spectral flux for a streaming audio signal, you must input at least two frames of audio data.

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.AsyncBuffer` to buffer audio into overlapped frames. Create a `dsp.SignalSink` to log the spectral flux calculation.

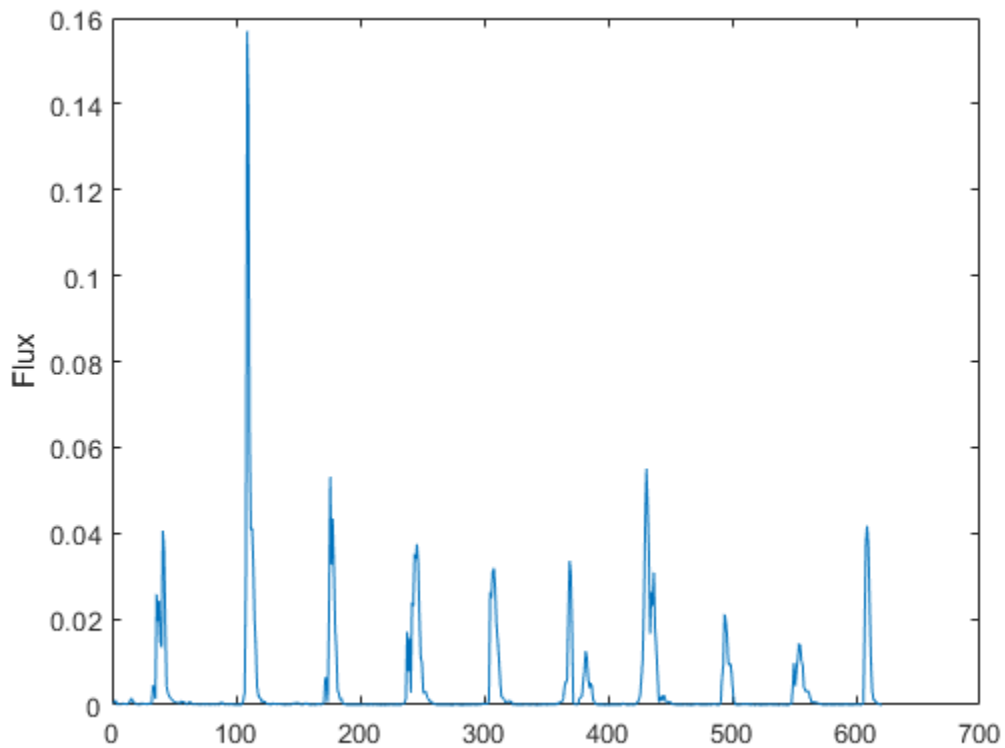
```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');  
buff = dsp.AsyncBuffer;  
logger = dsp.SignalSink;
```

In an audio stream loop:

- 1 Read in a frame of audio data from your source.
- 2 Write the audio data to a `dsp.AsyncBuffer`
- 3 If a frame of data is available from the buffer, read a frame and one hop of data, with overlap equal to samples per frame. This represents the two most recent audio frames.
- 4 Calculate the spectral flux for the two most recent audio frames.
- 5 Log the spectral flux for later plotting. Because flux is defined by a current frame and a previous frame, and because the condition before the first frame is unknown to the function, spectral flux outputs a flux of zero for the first frame. Log only the second value output from `spectralFlux`.

```
fs = fileReader.SampleRate;  
  
samplesPerFrame = round(fs*0.05);  
samplesOverlap = round(fs*0.025);  
  
samplesPerHop = samplesPerFrame - samplesOverlap;  
  
win = hamming(samplesPerFrame);  
  
while ~isDone(fileReader)  
    audioIn = fileReader();  
    write(buff,audioIn);  
  
    while buff.NumUnreadSamples >= samplesPerHop
```

```
audioBuffered = read(buff,samplesPerFrame+samplesPerHop,samplesPerFrame);  
flux = spectralFlux(audioBuffered,fs, ...  
    'Window',win, ...  
    'OverlapLength',samplesOverlap);  
    logger(flux(end))  
end  
  
end  
release(fileReader)  
  
Plot the logged data.  
plot(logger.Buffer)  
ylabel('Flux')
```



## Input Arguments

### **x** — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets **x** depends on the shape of **f**.

Data Types: `single` | `double`

### **f** — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets **x** depends on the shape of **f**:

- If **f** is a scalar, **x** is interpreted as a time-domain signal, and **f** is interpreted as the sample rate. In this case, **x** must be a real vector or matrix. If **x** is specified as a matrix, the columns are interpreted as individual channels.
- If **f** is a vector, **x** is interpreted as a frequency-domain signal, and **f** is interpreted as the frequencies, in Hz, corresponding to the rows of **x**. In this case, **x** must be a real  $L$ -by- $M$ -by- $N$  array, where  $L$  is the number of spectral values at given frequencies of **f**,  $M$  is the number of individual spectrums, and  $N$  is the number of channels.
- The number of rows of **x**,  $L$ , must be equal to the number of elements of **f**.

Data Types: `single` | `double`

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Window', hamming(256)`

### **NormType** — Norm type

2 (default) | 1

Norm type used to calculate, specified as the comma-separated pair consisting of `'NormType'` and 2 or 1.

Data Types: `single` | `double`

---

**Note** The following name-value pair arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, name-value pair arguments are ignored.

---

### **Window — Window applied in time domain**

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as the comma-separated pair consisting of 'Window' and a real vector. The number of elements in the vector must be in the range  $[1, \text{size}(x,1)]$ . The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

### **OverlapLength — Number of samples overlapped between adjacent windows**

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of 'OverlapLength' and an integer in the range  $[0, \text{size}(\text{Window},1)]$ .

Data Types: `single` | `double`

### **FFTLength — Number of bins in DFT**

`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as the comma-separated pair consisting of 'FFTLength' and a positive scalar integer. If unspecified, `FFTLength` defaults to the number of elements in the `Window`.

Data Types: `single` | `double`

### **Range — Frequency range (Hz)**

`[0, f/2]` (default) | two-element row vector

Frequency range in Hz, specified as the comma-separated pair consisting of 'Range' and a two-element row vector of increasing real values in the range  $[0, f/2]$ .

Data Types: `single` | `double`

### **SpectrumType — Spectrum type**

'power' (default) | 'magnitude'

Spectrum type, specified as the comma-separated pair consisting of 'SpectrumType' and 'power' or 'magnitude':

- 'power' -- The spectral flux is calculated for the one-sided power spectrum.
- 'magnitude' -- The spectral flux is calculated for the one-sided magnitude spectrum.

Data Types: `char` | `string`

## Output Arguments

### **flux** — Spectral flux (Hz)

`scalar` | `vector` | `matrix`

Spectral flux in Hz, returned as a scalar, vector, or matrix. Each row of `flux` corresponds to the spectral flux of a window of `x`. Each column of `flux` corresponds to an independent channel.

## Algorithms

The spectral flux is calculated as described in [1]:

$$\text{flux}(t) = \left( \sum_{k=b_1}^{b_2} |s_k(t) - s_k(t-1)|^P \right)^{1/P}$$

where

- $s_k$  is the spectral value at bin  $k$ .
- $b_1$  and  $b_2$  are the band edges, in bins, over which to calculate the spectral flux.
- $P$  is the norm type. You can specify the norm type using `NormType`.

## References

- [1] Scheirer, E., and M. Slaney. "Construction and Evaluation of a Robust Multifeature Speech/Music Discriminator." *IEEE International Conference on Acoustics, Speech, and Signal Processing*. Volume 2, 1997, pp. 1221-1224.



## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[integratedLoudness](#) | [spectralCentroid](#) | [splMeter](#)

### Topics

“Spectral Descriptors”

**Introduced in R2019a**

## spectralFlatness

Spectral flatness for audio signals and auditory spectrograms

### Syntax

```
flatness = spectralFlatness(x,f)  
flatness = spectralFlatness(x,f,Name,Value)  
[flatness,arithmeticMean,geometricMean] = spectralFlatness(____)
```

### Description

`flatness = spectralFlatness(x,f)` returns the spectral flatness of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`flatness = spectralFlatness(x,f,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

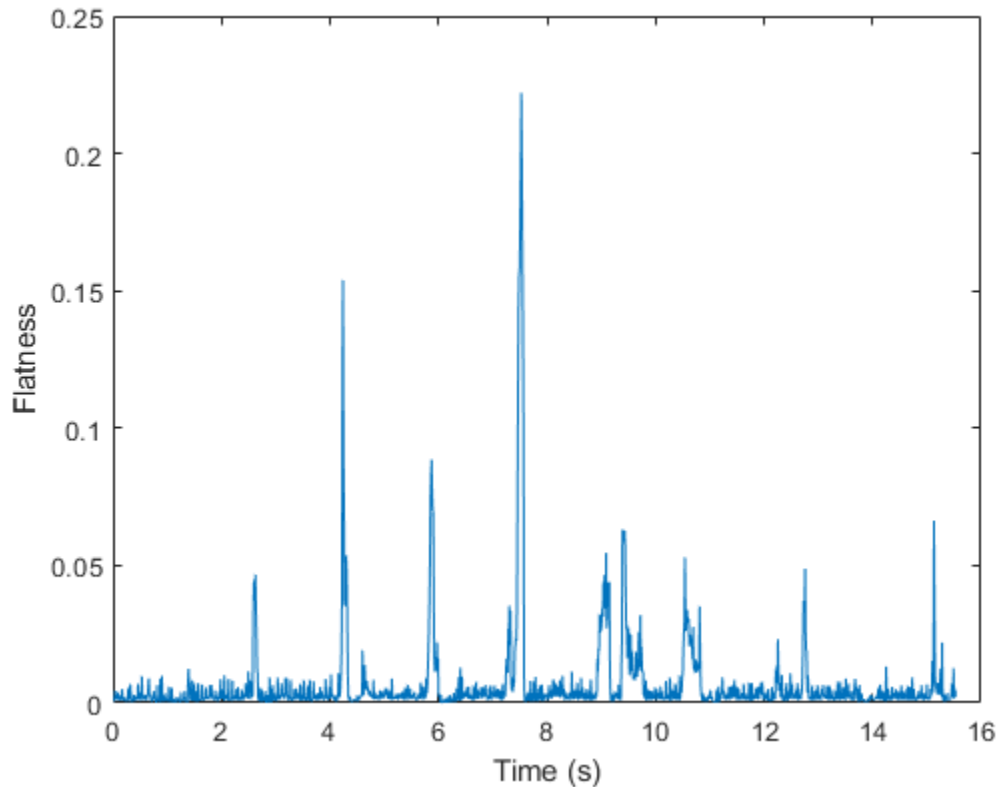
`[flatness,arithmeticMean,geometricMean] = spectralFlatness(____)` returns the spectral arithmetic mean and spectral geometric mean.

### Examples

#### Spectral Flatness of Time-Domain Audio

Read in an audio file, calculate the flatness using default parameters, and then plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
flatness = spectralFlatness(audioIn,fs);  
  
t = linspace(0,size(audioIn,1)/fs,size(flatness,1));  
plot(t,flatness)  
xlabel('Time (s)')  
ylabel('Flatness')
```



### Spectral Flatness of Frequency-Domain Audio Data

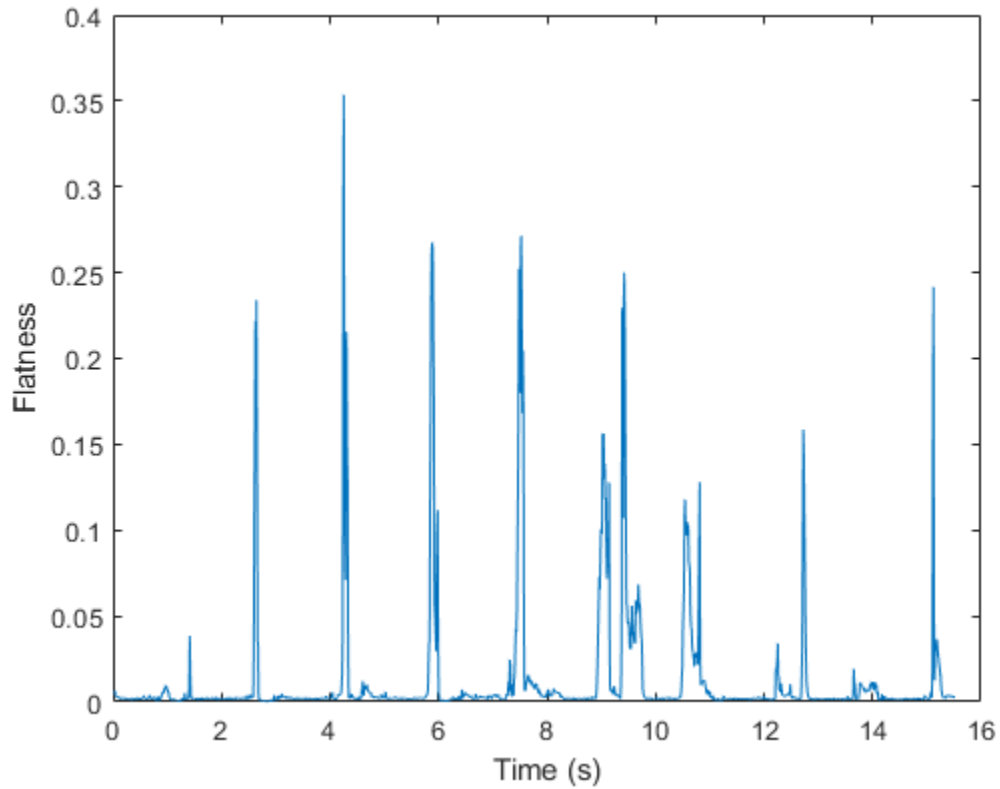
Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
[s,cf,t] = melSpectrogram(audioIn,fs);
```

Calculate the flatness of the mel spectrogram over time. Plot the results.

```
flatness = spectralFlatness(s,cf);
```

```
plot(t,flatness)
xlabel('Time (s)')
ylabel('Flatness')
```



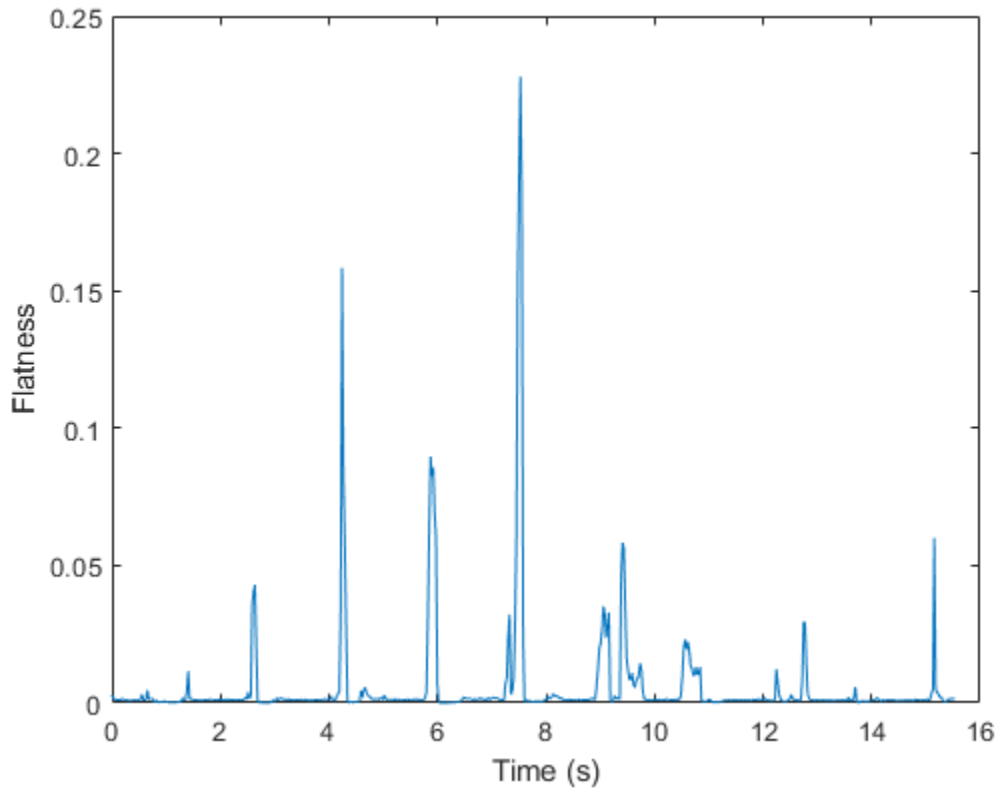
### Specify Nondefault Parameters

Read in an audio file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Calculate the flatness of the power spectrum over time. Calculate the flatness for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to  $f_s/2$  for the flatness calculation. Plot the results.

```
flatness = spectralFlatness(audioIn,fs, ...  
                            'Window',hamming(round(0.05*fs)), ...  
                            'OverlapLength',round(0.025*fs), ...  
                            'Range',[62.5,fs/2]);  
  
t = linspace(0,size(audioIn,1)/fs,size(flatness,1));  
plot(t,flatness)  
xlabel('Time (s)')  
ylabel('Flatness')
```



### Calculate Spectral Flatness of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral flatness calculation.

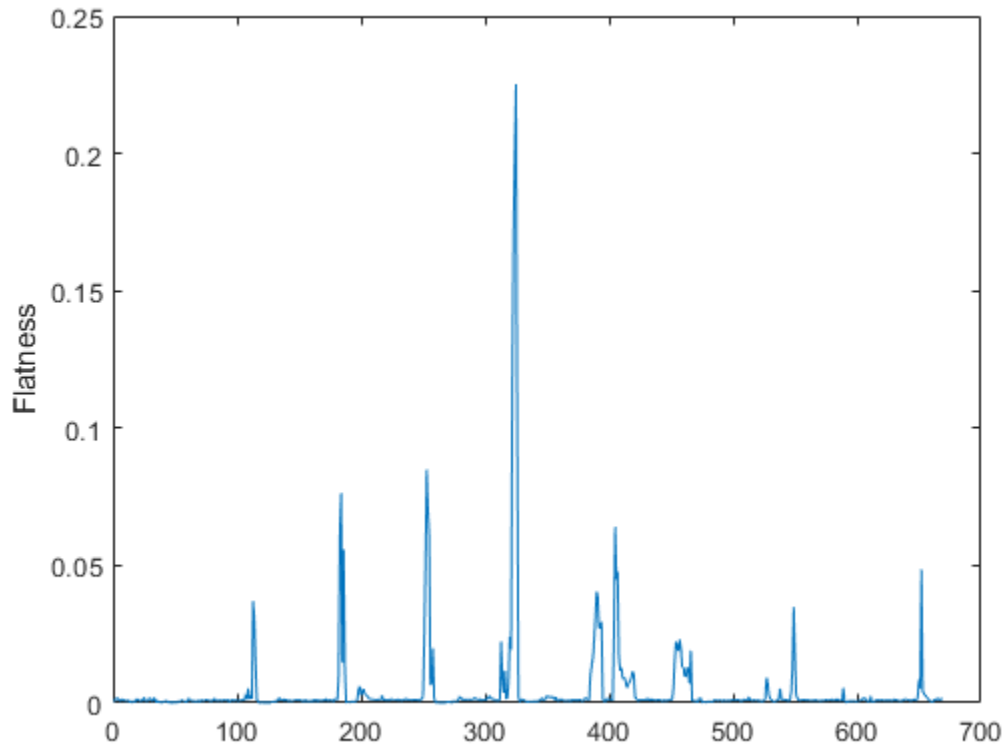
```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');  
logger = dsp.SignalSink;
```

In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Calculate the spectral flatness for the frame of audio.
- 3 Log the spectral flatness for later plotting.

To calculate the spectral flatness for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame);  
while ~isDone(fileReader)  
    audioIn = fileReader();  
    flatness = spectralFlatness(audioIn,fileReader.SampleRate, ...  
                               'Window',win, ...  
                               'OverlapLength',0);  
    logger(flatness)  
end  
  
plot(logger.Buffer)  
ylabel('Flatness')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralFlatness`.
- You want to calculate the spectral flatness for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;  
reset(logger)  
release(fileReader)
```

Specify that the spectral flatness is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        flatness = spectralFlatness(audioBuffered,fs, ...
                                    'Window',win, ...
                                    'OverlapLength',0);

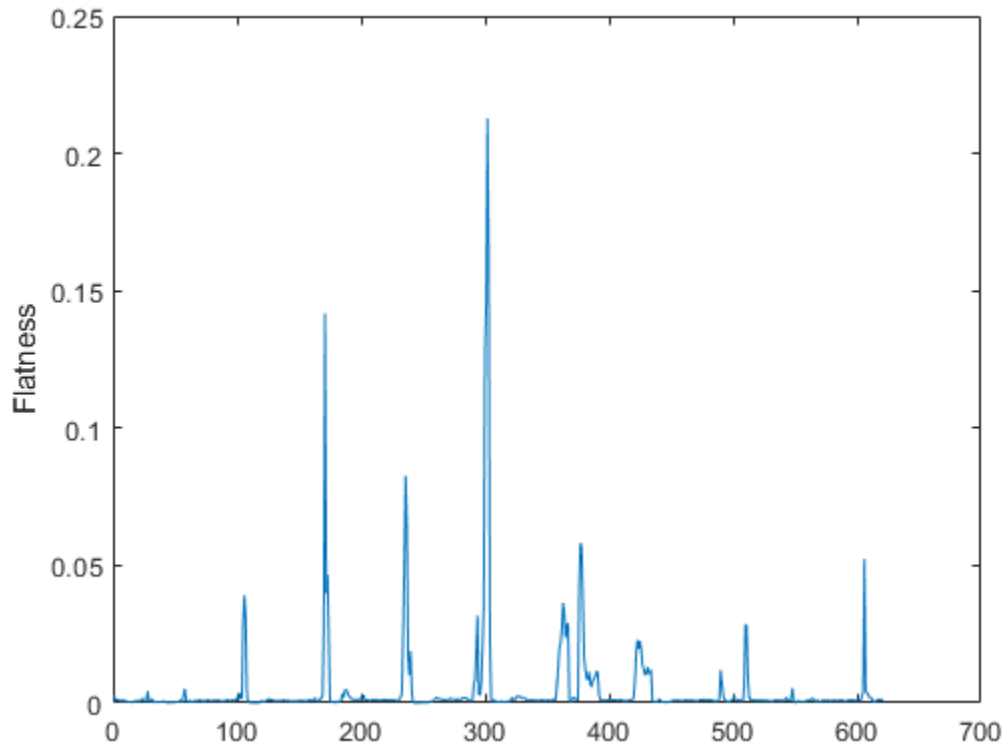
        logger(flatness)
    end

end
release(fileReader)

Plot the logged data.

plot(logger.Buffer)
ylabel('Flatness')
```





## Input Arguments

### **x** — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets  $x$  depends on the shape of  $f$ .

Data Types: `single` | `double`

### **f** — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets  $x$  depends on the shape of  $f$ :

- If  $f$  is a scalar,  $x$  is interpreted as a time-domain signal, and  $f$  is interpreted as the sample rate. In this case,  $x$  must be a real vector or matrix. If  $x$  is specified as a matrix, the columns are interpreted as individual channels.
- If  $f$  is a vector,  $x$  is interpreted as a frequency-domain signal, and  $f$  is interpreted as the frequencies, in Hz, corresponding to the rows of  $x$ . In this case,  $x$  must be a real  $L$ -by- $M$ -by- $N$  array, where  $L$  is the number of spectral values at given frequencies of  $f$ ,  $M$  is the number of individual spectrums, and  $N$  is the number of channels.
- The number of rows of  $x$ ,  $L$ , must be equal to the number of elements of  $f$ .

Data Types: `single` | `double`

### Name-Value Pair Arguments

---

**Note** The following name-value pair arguments apply if  $x$  is a time-domain signal. If  $x$  is a frequency-domain signal, name-value pair arguments are ignored.

---

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

Example: `'Window', hamming(256)`

#### **Window — Window applied in time domain**

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range `[1, size(x,1)]`. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

#### **OverlapLength — Number of samples overlapped between adjacent windows**

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of 'OverlapLength' and an integer in the range [0, size(Window,1)).

Data Types: single | double

### **FFTLength — Number of bins in DFT**

numel(Window) (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as the comma-separated pair consisting of 'FFTLength' and a positive scalar integer. If unspecified, FFTLength defaults to the number of elements in the Window.

Data Types: single | double

### **Range — Frequency range (Hz)**

[0, f/2] (default) | two-element row vector

Frequency range in Hz, specified as the comma-separated pair consisting of 'Range' and a two-element row vector of increasing real values in the range [0, f/2].

Data Types: single | double

### **SpectrumType — Spectrum type**

'power' (default) | 'magnitude'

Spectrum type, specified as the comma-separated pair consisting of 'SpectrumType' and 'power' or 'magnitude':

- 'power' -- The spectral flatness is calculated for the one-sided power spectrum.
- 'magnitude' -- The spectral flatness is calculated for the one-sided magnitude spectrum.

Data Types: char | string

## **Output Arguments**

### **flatness — Spectral flatness**

scalar | vector | matrix

Spectral flatness, returned as a scalar, vector, or matrix. Each row of `flatness` corresponds to the spectral flatness of a window of `x`. Each column of `flatness` corresponds to an independent channel.

### **arithmeticMean — Spectral arithmetic mean**

scalar | vector | matrix

Spectral arithmetic mean, returned as a scalar, vector, or matrix. Each row of `arithmeticMean` corresponds to the arithmetic mean of the spectrum of a window of `x`. Each column of `arithmeticMean` corresponds to an independent channel.

### **geometricMean — Spectral geometric mean**

scalar | vector | matrix

Spectral geometric mean, returned as a scalar, vector, or matrix. Each row of `geometricMean` corresponds to the geometric mean of the spectrum of a window of `x`. Each column of `geometricMean` corresponds to an independent channel.

## Algorithms

The spectral flatness is calculated as described in [1]:

$$\text{flatness} = \frac{\left( \prod_{k=b_1}^{b_2} s_k \right)^{\frac{1}{b_2 - b_1}}}{\frac{1}{b_2 - b_1} \sum_{k=b_1}^{b_2} s_k}$$

where

- $s_k$  is the spectral value at bin  $k$ .
- $b_1$  and  $b_2$  are the band edges, in bins, over which to calculate the spectral spread.

## References

- [1] Johnston, J.d. "Transform Coding of Audio Signals Using Perceptual Noise Criteria." *IEEE Journal on Selected Areas in Communications*. Vol. 6, Number 2, 1988, pp. 314-323.

## **Extended Capabilities**

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

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

### **See Also**

spectralCrest

### **Topics**

“Spectral Descriptors”

**Introduced in R2019a**

## spectralEntropy

Spectral entropy for audio signals and auditory spectrograms

### Syntax

```
entropy = spectralEntropy(x, f)  
entropy = spectralEntropy(x, f, Name, Value)
```

### Description

`entropy = spectralEntropy(x, f)` returns the spectral entropy of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

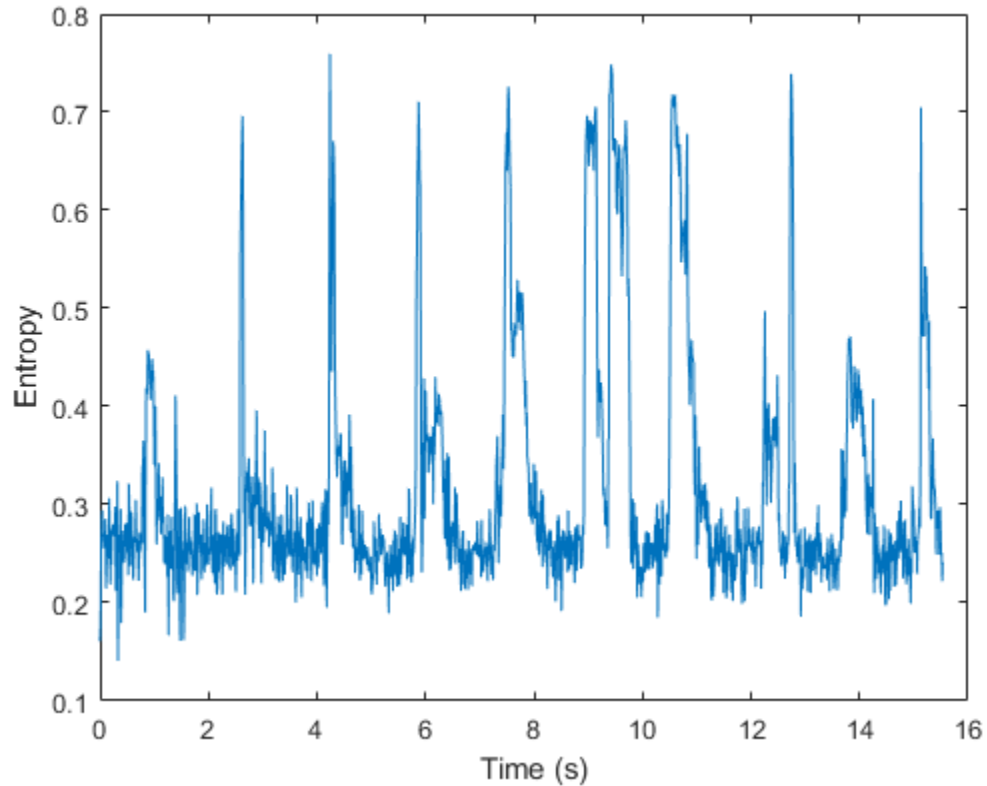
`entropy = spectralEntropy(x, f, Name, Value)` specifies options using one or more `Name, Value` pair arguments.

### Examples

#### Spectral Entropy of Time-Domain Audio

Read in an audio file, calculate the entropy using default parameters, and then plot the results.

```
[audioIn, fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
entropy = spectralEntropy(audioIn, fs);  
  
t = linspace(0, size(audioIn, 1)/fs, size(entropy, 1));  
plot(t, entropy)  
xlabel('Time (s)')  
ylabel('Entropy')
```



### Spectral Entropy of Frequency-Domain Audio Data

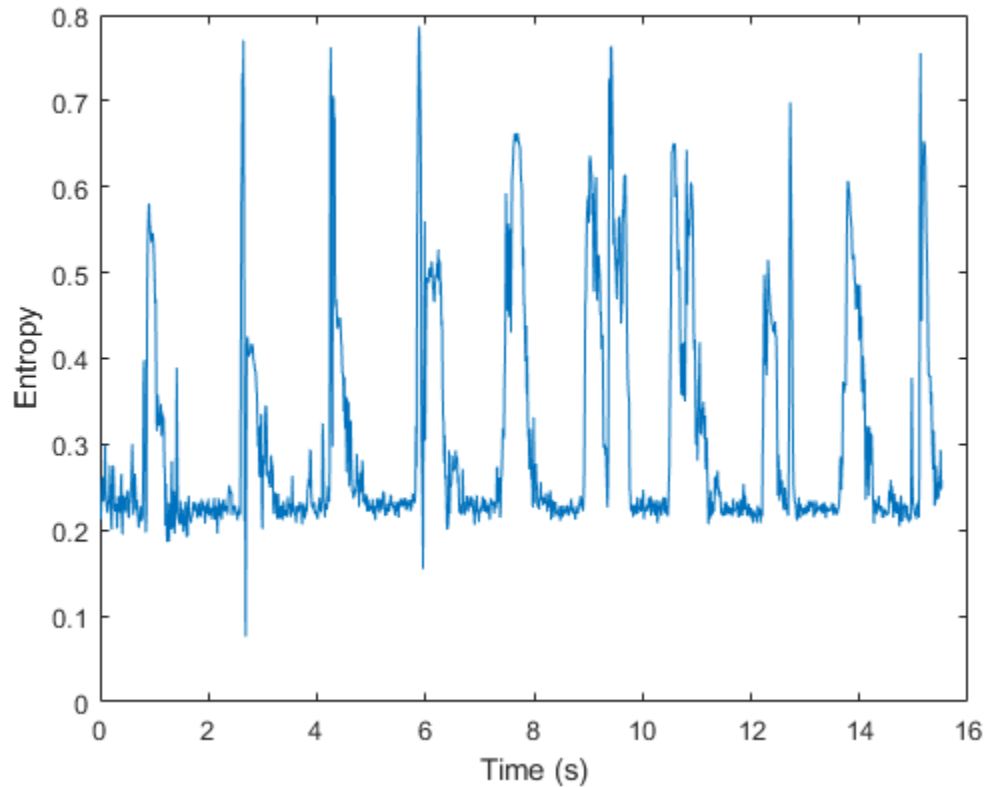
Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
[s,cf,t] = melSpectrogram(audioIn,fs);
```

Calculate the entropy of the mel spectrogram over time. Plot the results.

```
entropy = spectralEntropy(s,cf);
```

```
plot(t,entropy)
xlabel('Time (s)')
ylabel('Entropy')
```



### Specify Nondefault Parameters

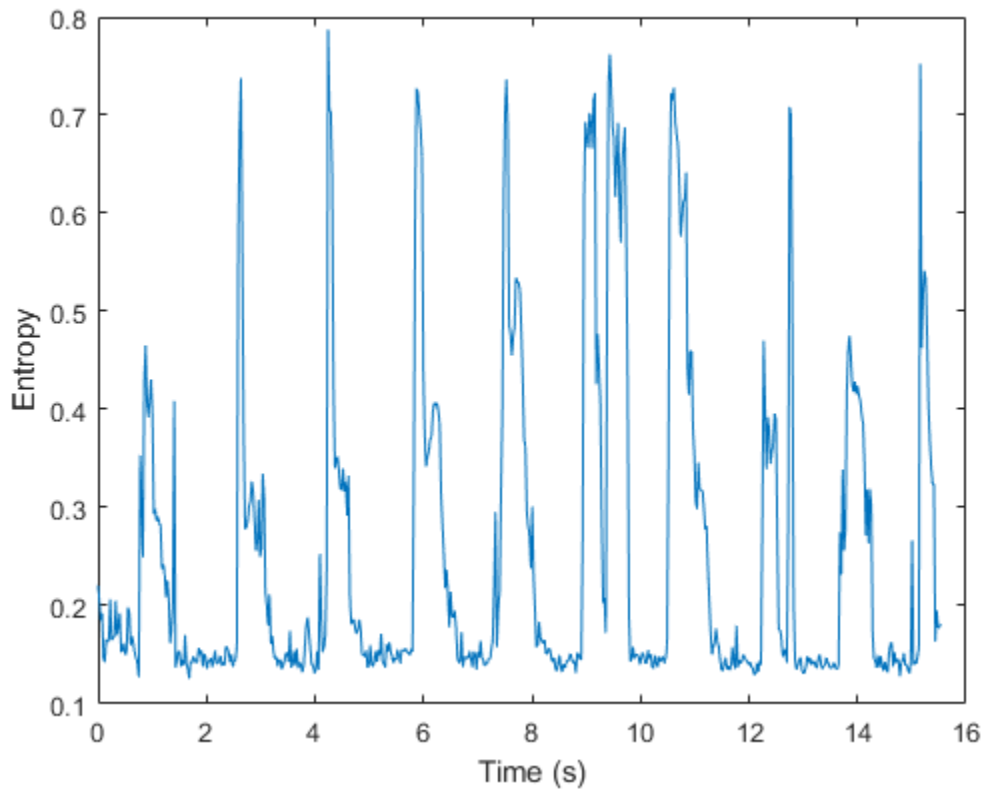
Read in an audio file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```



Calculate the entropy of the power spectrum over time. Calculate the entropy for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to  $f_s/2$  for the entropy calculation. Plot the results.

```
entropy = spectralEntropy(audioIn,fs, ...  
                          'Window',hamming(round(0.05*fs)), ...  
                          'OverlapLength',round(0.025*fs), ...  
                          'Range',[62.5,fs/2]);  
  
t = linspace(0,size(audioIn,1)/fs,size(entropy,1));  
plot(t,entropy)  
xlabel('Time (s)')  
ylabel('Entropy')
```



### Calculate Spectral Entropy of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral entropy calculation.

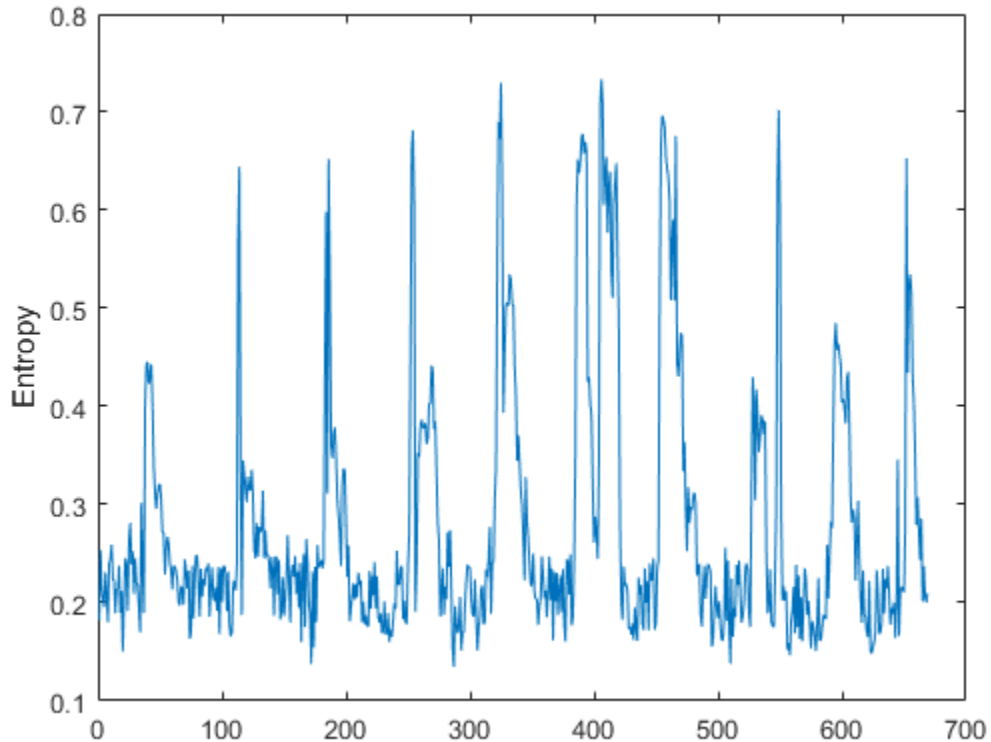
```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');  
logger = dsp.SignalSink;
```

In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Calculate the spectral entropy for the frame of audio.
- 3 Log the spectral entropy for later plotting.

To calculate the spectral entropy for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
while ~isDone(fileReader)  
    audioIn = fileReader();  
    entropy = spectralEntropy(audioIn,fileReader.SampleRate, ...  
                             'Window',hamming(size(audioIn,1)), ...  
                             'OverlapLength',0);  
    logger(entropy)  
end  
  
plot(logger.Buffer)  
ylabel('Entropy')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralEntropy`.
- You want to calculate the spectral entropy for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;  
reset(logger)  
release(fileReader)
```

Specify that the spectral entropy is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

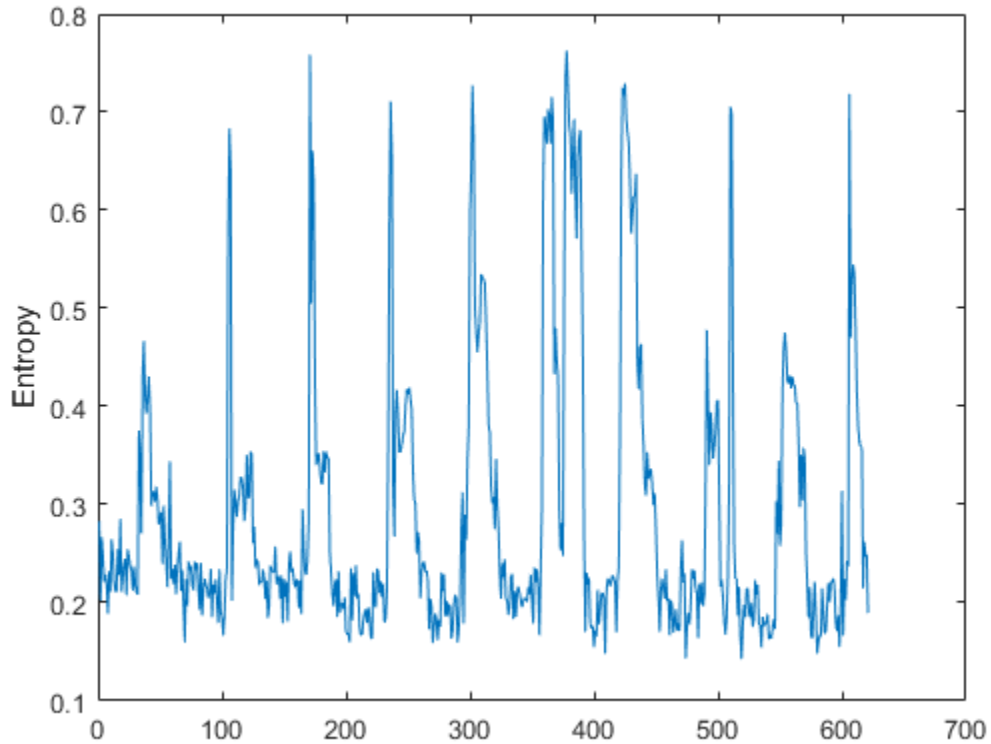
        entropy = spectralEntropy(audioBuffered,fs, ...
                                  'Window',win, ...
                                  'OverlapLength',0);

        logger(entropy)
    end

end
release(fileReader)

Plot the logged data.

plot(logger.Buffer)
ylabel('Entropy')
```



## Input Arguments

### **x** — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets  $x$  depends on the shape of  $f$ .

Data Types: `single` | `double`

### **f** — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets  $x$  depends on the shape of  $f$ :

- If  $f$  is a scalar,  $x$  is interpreted as a time-domain signal, and  $f$  is interpreted as the sample rate. In this case,  $x$  must be a real vector or matrix. If  $x$  is specified as a matrix, the columns are interpreted as individual channels.
- If  $f$  is a vector,  $x$  is interpreted as a frequency-domain signal, and  $f$  is interpreted as the frequencies, in Hz, corresponding to the rows of  $x$ . In this case,  $x$  must be a real  $L$ -by- $M$ -by- $N$  array, where  $L$  is the number of spectral values at given frequencies of  $f$ ,  $M$  is the number of individual spectrums, and  $N$  is the number of channels.
- The number of rows of  $x$ ,  $L$ , must be equal to the number of elements of  $f$ .

Data Types: `single` | `double`

### Name-Value Pair Arguments

---

**Note** The following name-value pair arguments apply if  $x$  is a time-domain signal. If  $x$  is a frequency-domain signal, name-value pair arguments are ignored.

---

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

Example: `'Window', hamming(256)`

#### **Window — Window applied in time domain**

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range `[1, size(x,1)]`. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

#### **OverlapLength — Number of samples overlapped between adjacent windows**

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of 'OverlapLength' and an integer in the range [0, size(Window,1)].

Data Types: single | double

### **FFTLength — Number of bins in DFT**

numel(Window) (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as the comma-separated pair consisting of 'FFTLength' and a positive scalar integer. If unspecified, FFTLength defaults to the number of elements in the Window.

Data Types: single | double

### **Range — Frequency range (Hz)**

[0, f/2] (default) | two-element row vector

Frequency range in Hz, specified as the comma-separated pair consisting of 'Range' and a two-element row vector of increasing real values in the range [0, f/2].

Data Types: single | double

### **SpectrumType — Spectrum type**

'power' (default) | 'magnitude'

Spectrum type, specified as the comma-separated pair consisting of 'SpectrumType' and 'power' or 'magnitude':

- 'power' -- The spectral entropy is calculated for the one-sided power spectrum.
- 'magnitude' -- The spectral entropy is calculated for the one-sided magnitude spectrum.

Data Types: char | string

## **Output Arguments**

### **entropy — Spectral entropy**

scalar | vector | matrix

Spectral entropy, returned as a scalar, vector, or matrix. Each row of entropy corresponds to the spectral entropy of a window of  $x$ . Each column of entropy corresponds to an independent channel.

## Algorithms

The spectral entropy is calculated as described in [1]:

$$\text{entropy} = \frac{- \sum_{k=b_1}^{b_2} s_k \log(s_k)}{\log(b_2 - b_1)}$$

where

- $s_k$  is the spectral value at bin  $k$ .
- $b_1$  and  $b_2$  are the band edges, in bins, over which to calculate the spectral entropy.

## References

- [1] Misra, H., S. Ikbal, H. Bourlard, and H. Hermansky. "Spectral Entropy Based Feature for Robust ASR." *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

spectralKurtosis | spectralSkewness | spectralSpread

### Topics

"Spectral Descriptors"



**Introduced in R2019a**

## spectralDecrease

Spectral decrease for audio signals and auditory spectrograms

### Syntax

```
decrease = spectralDecrease(x, f)  
decrease = spectralDecrease(x, f, Name, Value)
```

### Description

`decrease = spectralDecrease(x, f)` returns the spectral decrease of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

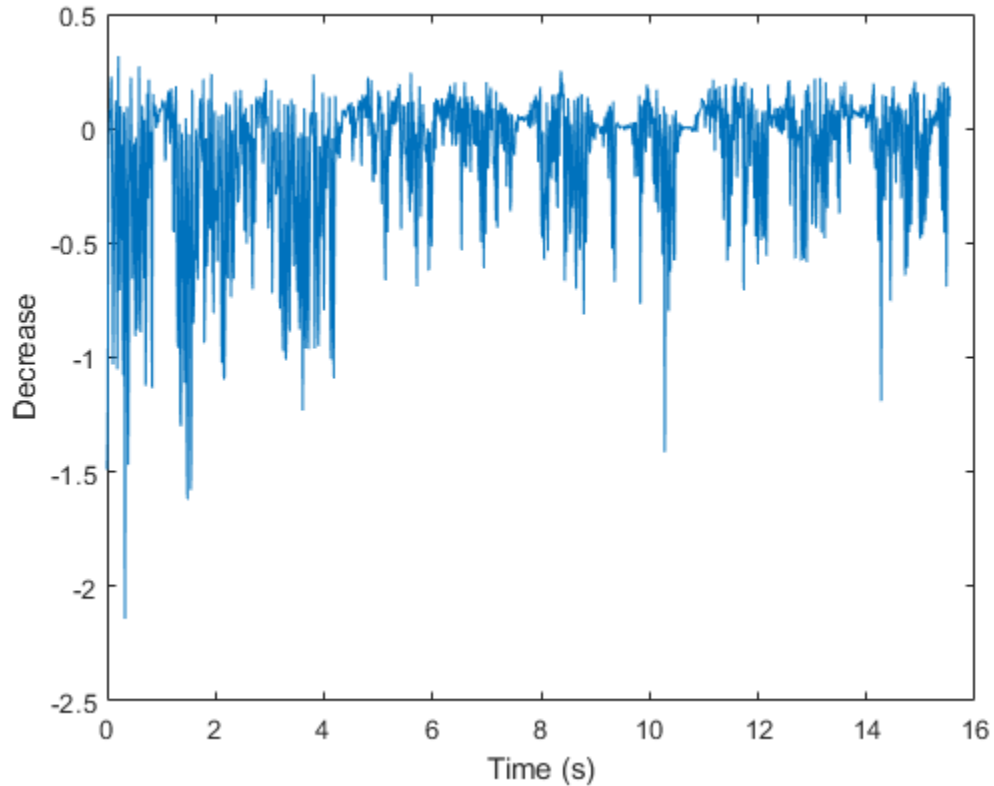
`decrease = spectralDecrease(x, f, Name, Value)` specifies options using one or more `Name, Value` pair arguments.

### Examples

#### Spectral Decrease of Time-Domain Audio

Read in an audio file, calculate the decrease using default parameters, and then plot the results.

```
[audioIn, fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
decrease = spectralDecrease(audioIn, fs);  
  
t = linspace(0, size(audioIn, 1)/fs, size(decrease, 1));  
plot(t, decrease)  
xlabel('Time (s)')  
ylabel('Decrease')
```



### Spectral Decrease of Frequency-Domain Audio Data

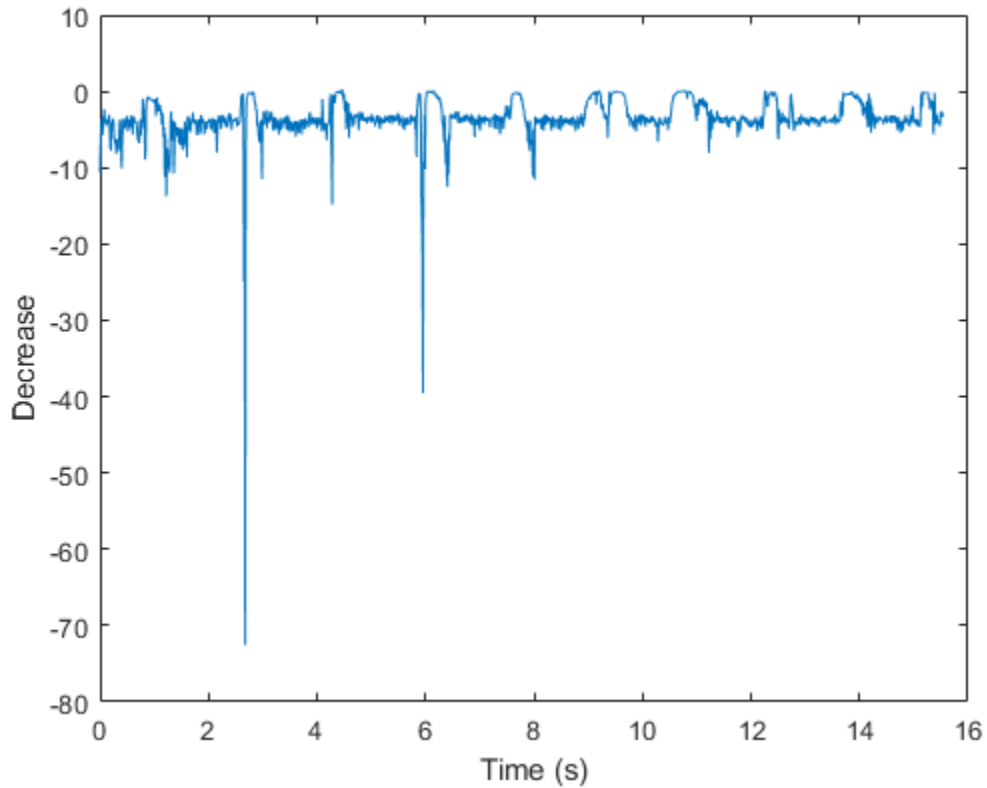
Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
[s,cf] = melSpectrogram(audioIn,fs);
```

Calculate the decrease of the mel spectrogram over time. Plot the results.

```
decrease = spectralDecrease(s,cf);
```

```
t = linspace(0,size(audioIn,1)/fs,size(decrease,1));  
plot(t,decrease)  
xlabel('Time (s)')  
ylabel('Decrease')
```



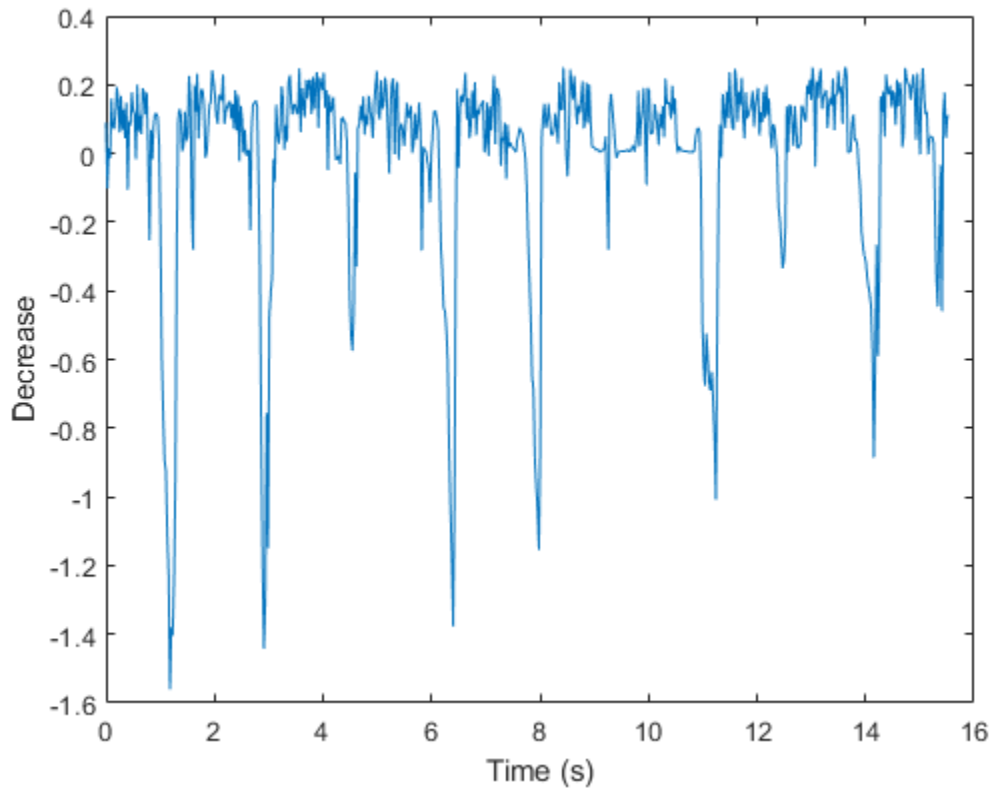
### Specify Nondefault Parameters

Read in an audio file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Calculate the decrease of the magnitude spectrum over time. Calculate the decrease for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to  $f_s/2$  for the decrease calculation. Plot the results.

```
decrease = spectralDecrease(audioIn,fs, ...  
    'Window',hamming(round(0.05*fs)), ...  
    'OverlapLength',round(0.025*fs), ...  
    'Range',[62.5,fs/2]);  
  
t = linspace(0,size(audioIn,1)/fs,size(decrease,1));  
plot(t,decrease)  
xlabel('Time (s)')  
ylabel('Decrease')
```



### Calculate Spectral Decrease of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral decrease calculation.

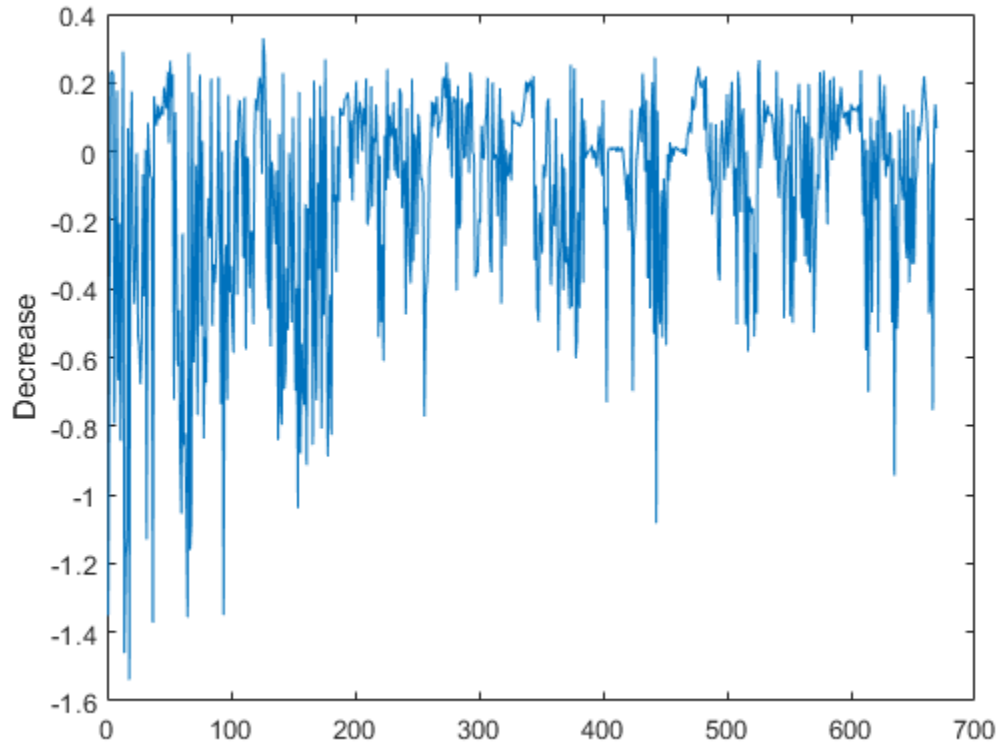
```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');  
logger = dsp.SignalSink;
```

In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Calculate the spectral decrease for the frame of audio.
- 3 Log the spectral decrease for later plotting.

To calculate the spectral decrease for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
while ~isDone(fileReader)  
    audioIn = fileReader();  
    decrease = spectralDecrease(audioIn,fileReader.SampleRate, ...  
                               'Window',hamming(size(audioIn,1)), ...  
                               'OverlapLength',0);  
  
    logger(decrease)  
end  
  
plot(logger.Buffer)  
ylabel('Decrease')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralDecrease`.
- You want to calculate the spectral decrease for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;  
reset(logger)  
release(fileReader)
```

Specify that the spectral decrease is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

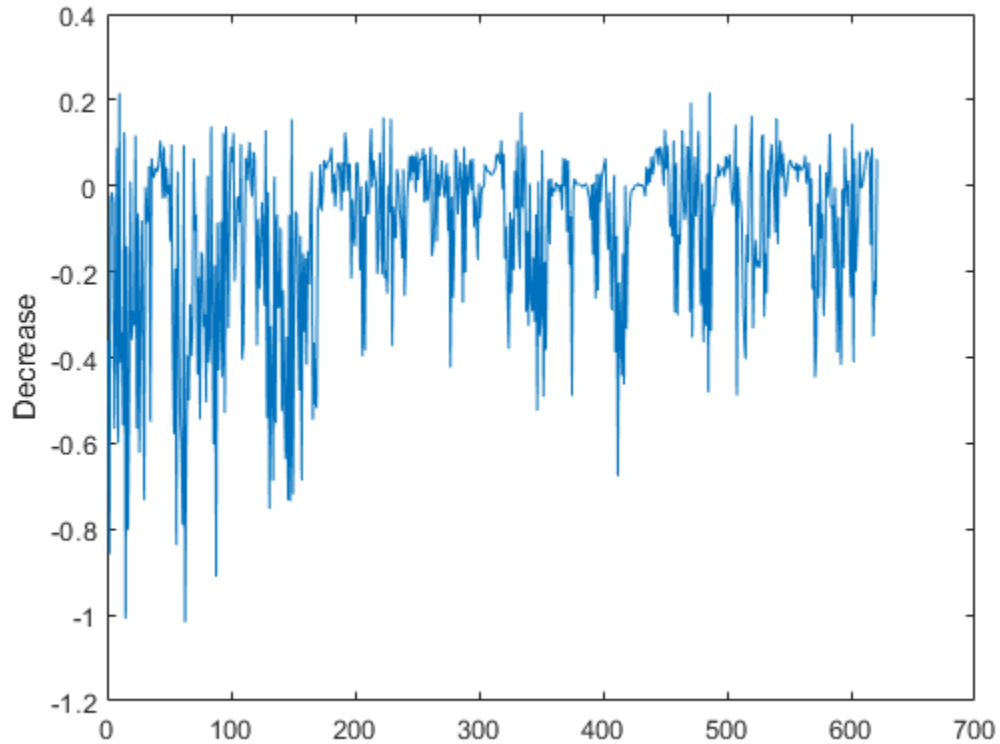
        decrease = spectralDecrease(audioBuffered,fs, ...
                                    'Window',win, ...
                                    'OverlapLength',0);

        logger(decrease)
    end
end
release(fileReader)

Plot the logged data.

plot(logger.Buffer)
ylabel('Decrease')
```





## Input Arguments

### **x** — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets  $x$  depends on the shape of  $f$ .

Data Types: `single` | `double`

### **f** — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets  $x$  depends on the shape of  $f$ :

- If  $f$  is a scalar,  $x$  is interpreted as a time-domain signal, and  $f$  is interpreted as the sample rate. In this case,  $x$  must be a real vector or matrix. If  $x$  is specified as a matrix, the columns are interpreted as individual channels.
- If  $f$  is a vector,  $x$  is interpreted as a frequency-domain signal, and  $f$  is interpreted as the frequencies, in Hz, corresponding to the rows of  $x$ . In this case,  $x$  must be a real  $L$ -by- $M$ -by- $N$  array, where  $L$  is the number of spectral values at given frequencies of  $f$ ,  $M$  is the number of individual spectrums, and  $N$  is the number of channels.
- The number of rows of  $x$ ,  $L$ , must be equal to the number of elements of  $f$ .

Data Types: `single` | `double`

### Name-Value Pair Arguments

---

**Note** The following name-value pair arguments apply if  $x$  is a time-domain signal. If  $x$  is a frequency-domain signal, name-value pair arguments are ignored.

---

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

Example: `'Window', hamming(256)`

#### **Window — Window applied in time domain**

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range `[1, size(x,1)]`. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

#### **OverlapLength — Number of samples overlapped between adjacent windows**

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of 'OverlapLength' and an integer in the range [0, size(Window,1)).

Data Types: single | double

### **FFTLength — Number of bins in DFT**

numel(Window) (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as the comma-separated pair consisting of 'FFTLength' and a positive scalar integer. If unspecified, FFTLength defaults to the number of elements in the Window.

Data Types: single | double

### **Range — Frequency range (Hz)**

[0, f/2] (default) | two-element row vector

Frequency range in Hz, specified as the comma-separated pair consisting of 'Range' and a two-element row vector of increasing real values in the range [0, f/2].

Data Types: single | double

### **SpectrumType — Spectrum type**

'magnitude' (default) | 'power'

Spectrum type, specified as the comma-separated pair consisting of 'SpectrumType' and 'power' or 'magnitude':

- 'power' -- The spectral decrease is calculated for the one-sided power spectrum.
- 'magnitude' -- The spectral decrease is calculated for the one-sided magnitude spectrum.

Data Types: char | string

## **Output Arguments**

### **decrease — Spectral decrease**

scalar | vector | matrix

Spectral decrease in Hz, returned as a scalar, vector, or matrix. Each row of `decrease` corresponds to the spectral centroid of a window of `x`. Each column of `decrease` corresponds to an independent channel.

## Algorithms

The spectral decrease is calculated as described in [1]:

$$\text{decrease} = \frac{\sum_{k=b_1+1}^{b_2} \frac{s_k - s_{b_1}}{k-1}}{\sum_{k=b_1+1}^{b_2} s_k}$$

where

- $s_k$  is the spectral value at bin  $k$ .
- $b_1$  and  $b_2$  are the band edges, in bins, over which to calculate the spectral decrease.

## References

- [1] Peeters, G. "A Large Set of Audio Features for Sound Description (Similarity and Classification) in the CUIDADO Project." Technical Report; IRCAM: Paris, France, 2004.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`spectralCrest` | `spectralSlope`

## **Topics**

“Spectral Descriptors”

**Introduced in R2019a**

## spectralCrest

Spectral crest for audio signals and auditory spectrograms

### Syntax

```
crest = spectralCrest(x,f)
crest = spectralCrest(x,f,Name,Value)
[crest,spectralPeak,spectralMean] = spectralCrest( ___ )
```

### Description

`crest = spectralCrest(x,f)` returns the spectral crest of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`crest = spectralCrest(x,f,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

`[crest,spectralPeak,spectralMean] = spectralCrest( ___ )` returns the spectral peak and spectral mean.

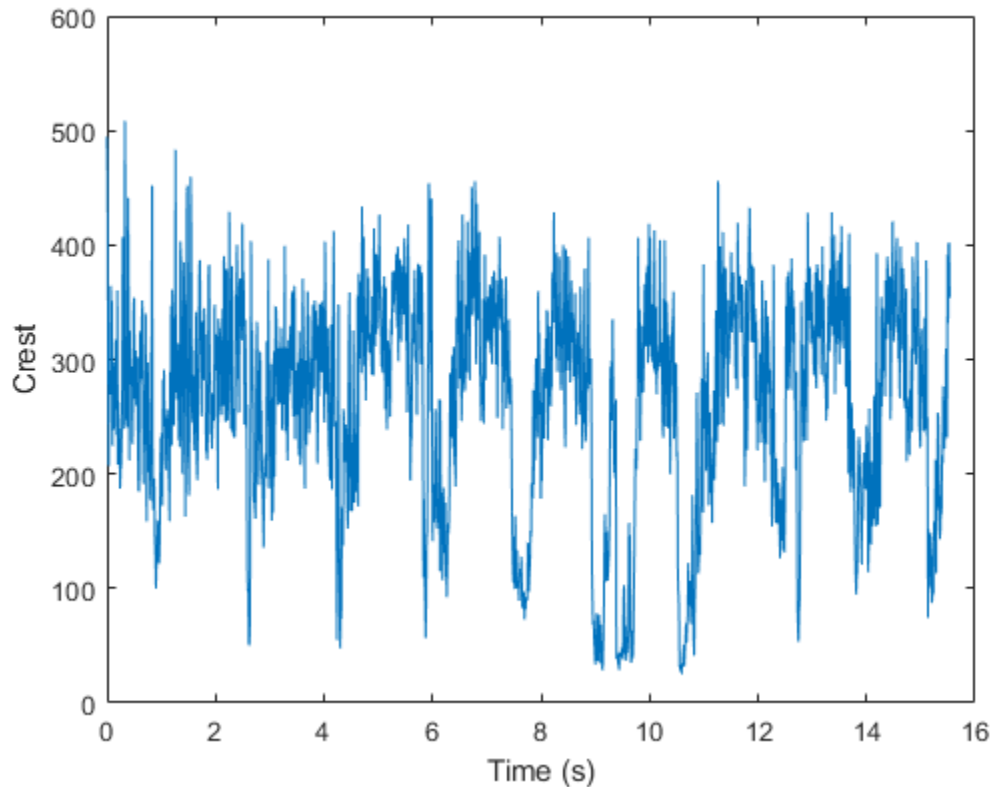
### Examples

#### Spectral Crest of Time-Domain Audio

Read in an audio file, calculate the crest using default parameters, and then plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
crest = spectralCrest(audioIn,fs);

t = linspace(0,size(audioIn,1)/fs,size(crest,1));
plot(t,crest)
xlabel('Time (s)')
ylabel('Crest')
```



### Spectral Crest of Frequency-Domain Audio Data

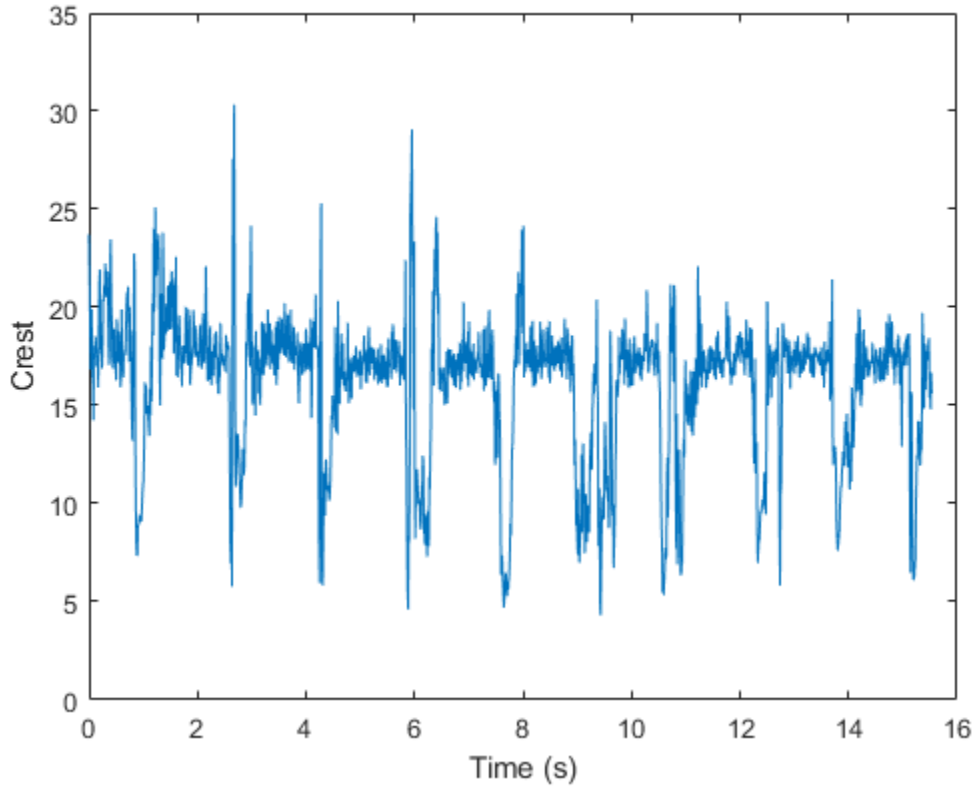
Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
[s,cf] = melSpectrogram(audioIn,fs);
```

Calculate the crest of the mel spectrogram over time. Plot the results.

```
crest = spectralCrest(s,cf);
```

```
t = linspace(0,size(audioIn,1)/fs,size(crest,1));  
plot(t,crest)  
xlabel('Time (s)')  
ylabel('Crest')
```



### Specify Nondefault Parameters

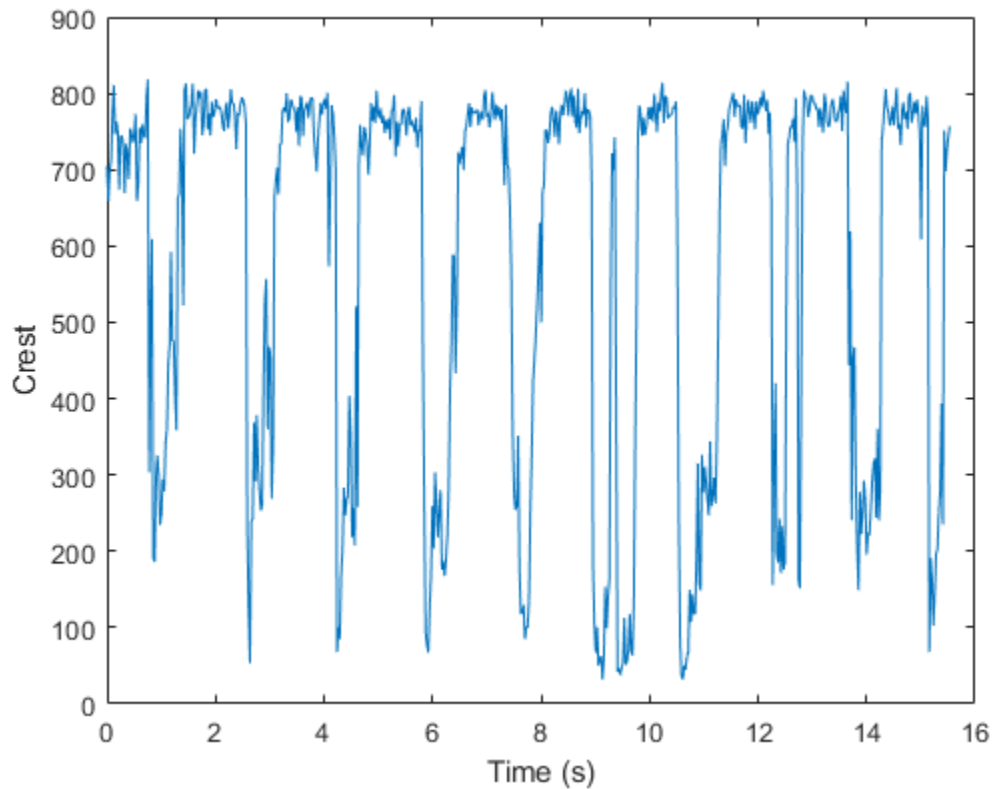
Read in an audio file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```



Calculate the crest of the power spectrum over time. Calculate the crest for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to  $f_s/2$  for the crest calculation. Plot the results.

```
crest = spectralCrest(audioIn,fs, ...  
                    'Window',hamming(round(0.05*fs)), ...  
                    'OverlapLength',round(0.025*fs), ...  
                    'Range',[62.5,fs/2]);  
  
t = linspace(0,size(audioIn,1)/fs,size(crest,1));  
plot(t,crest)  
xlabel('Time (s)')  
ylabel('Crest')
```



### Calculate Spectral Crest of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral crest calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44pl-mono-15secs.wav');  
logger = dsp.SignalSink;
```

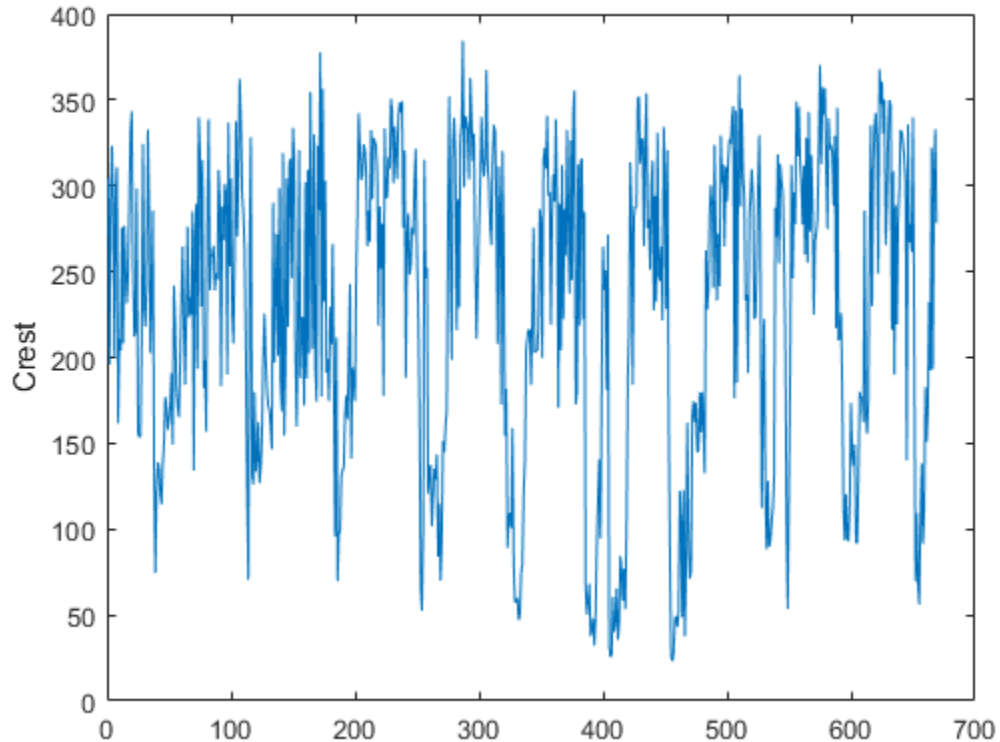
In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Calculate the spectral crest for the frame of audio.
- 3 Log the spectral crest for later plotting.

To calculate the spectral crest for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero.

Plot the logged data.

```
while ~isDone(fileReader)  
    audioIn = fileReader();  
    crest = spectralCrest(audioIn,fileReader.SampleRate, ...  
                        'Window',hamming(size(audioIn,1)), ...  
                        'OverlapLength',0);  
    logger(crest)  
end  
  
plot(logger.Buffer)  
ylabel('Crest')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralCrest`.
- You want to calculate the spectral crest for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;  
reset(logger)  
release(fileReader)
```

Specify that the spectral crest is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);

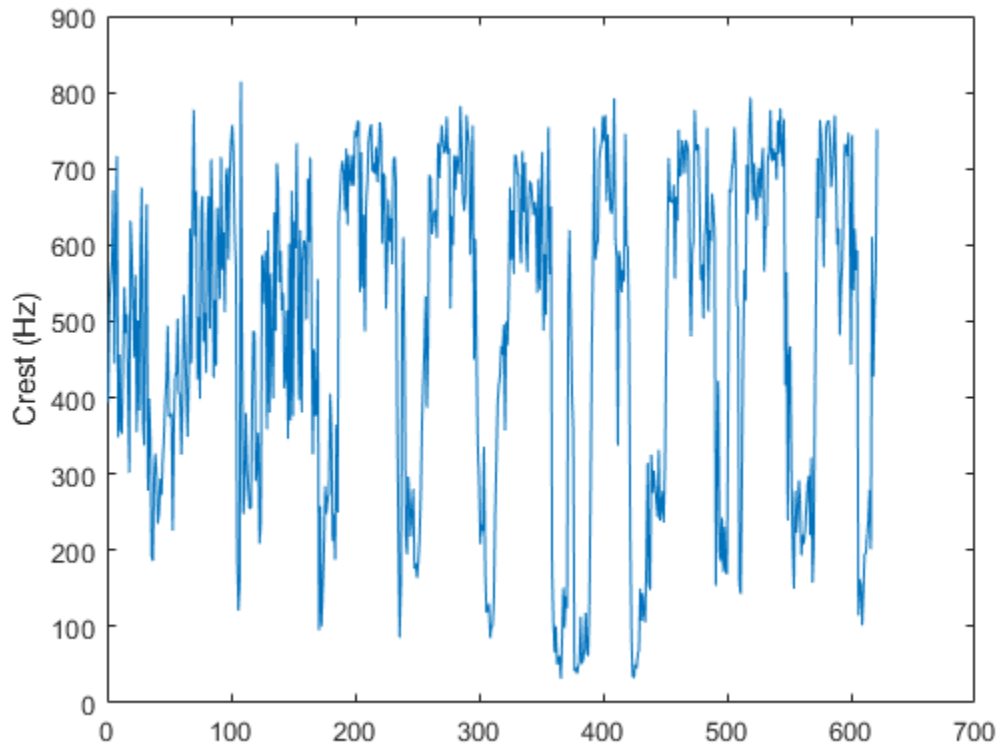
while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        crest = spectralCrest(audioBuffered,fs, ...
                               'Window',win, ...
                               'OverlapLength',0);
        logger(crest)
    end
end
release(fileReader)

Plot the logged data.

plot(logger.Buffer)
ylabel('Crest (Hz)')
```



## Input Arguments

### **x** — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets **x** depends on the shape of **f**.

Data Types: `single` | `double`

### **f** — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets  $x$  depends on the shape of  $f$ :

- If  $f$  is a scalar,  $x$  is interpreted as a time-domain signal, and  $f$  is interpreted as the sample rate. In this case,  $x$  must be a real vector or matrix. If  $x$  is specified as a matrix, the columns are interpreted as individual channels.
- If  $f$  is a vector,  $x$  is interpreted as a frequency-domain signal, and  $f$  is interpreted as the frequencies, in Hz, corresponding to the rows of  $x$ . In this case,  $x$  must be a real  $L$ -by- $M$ -by- $N$  array, where  $L$  is the number of spectral values at given frequencies of  $f$ ,  $M$  is the number of individual spectrums, and  $N$  is the number of channels.
- The number of rows of  $x$ ,  $L$ , must be equal to the number of elements of  $f$ .

Data Types: `single` | `double`

### Name-Value Pair Arguments

---

**Note** The following name-value pair arguments apply if  $x$  is a time-domain signal. If  $x$  is a frequency-domain signal, name-value pair arguments are ignored.

---

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

Example: `'Window', hamming(256)`

#### **Window — Window applied in time domain**

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range  $[1, \text{size}(x, 1)]$ . The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

#### **OverlapLength — Number of samples overlapped between adjacent windows**

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of 'OverlapLength' and an integer in the range [0, size(Window,1)].

Data Types: single | double

### **FFTLength — Number of bins in DFT**

numel(Window) (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as the comma-separated pair consisting of 'FFTLength' and a positive scalar integer. If unspecified, FFTLength defaults to the number of elements in the Window.

Data Types: single | double

### **Range — Frequency range (Hz)**

[0, f/2] (default) | two-element row vector

Frequency range in Hz, specified as the comma-separated pair consisting of 'Range' and a two-element row vector of increasing real values in the range [0, f/2].

Data Types: single | double

### **SpectrumType — Spectrum type**

'power' (default) | 'magnitude'

Spectrum type, specified as the comma-separated pair consisting of 'SpectrumType' and 'power' or 'magnitude':

- 'power' -- The spectral crest is calculated for the one-sided power spectrum.
- 'magnitude' -- The spectral crest is calculated for the one-sided magnitude spectrum.

Data Types: char | string

## **Output Arguments**

### **crest — Spectral crest**

scalar | vector | matrix

Spectral crest, returned as a scalar, vector, or matrix. Each row of `crest` corresponds to the spectral crest of a window of `x`. Each column of `crest` corresponds to an independent channel.

### **spectralPeak — Spectral peak**

scalar | vector | matrix

Spectral peak, returned as a scalar, vector, or matrix. Each row of `spectralPeak` corresponds to the spectral crest of a window of `x`. Each column of `spectralPeak` corresponds to an independent channel.

### **spectralMean — Spectral mean**

scalar | vector | matrix

Spectral mean, returned as a scalar, vector, or matrix. Each row of `spectralMean` corresponds to the spectral crest of a window of `x`. Each column of `spectralMean` corresponds to an independent channel.

## Algorithms

The spectral crest is calculated as described in [1]:

$$\text{crest} = \frac{\max(s_k \in [b_1, b_2])}{\frac{1}{b_2 - b_1} \sum_{k=b_1}^{b_2} s_k}$$

where

- $s_k$  is the spectral value at bin  $k$ .
- $b_1$  and  $b_2$  are the band edges, in bins, over which to calculate the spectral crest.

## References

- [1] Peeters, G. "A Large Set of Audio Features for Sound Description (Similarity and Classification) in the CUIDADO Project." Technical Report; IRCAM: Paris, France, 2004.



## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`spectralFlatness` | `spectralSkewness` | `spectralSpread`

### Topics

“Spectral Descriptors”

**Introduced in R2019a**

## spectralCentroid

Spectral centroid for audio signals and auditory spectrograms

### Syntax

```
centroid = spectralCentroid(x,f)  
centroid = spectralCentroid(x,f,Name,Value)
```

### Description

`centroid = spectralCentroid(x,f)` returns the spectral centroid of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

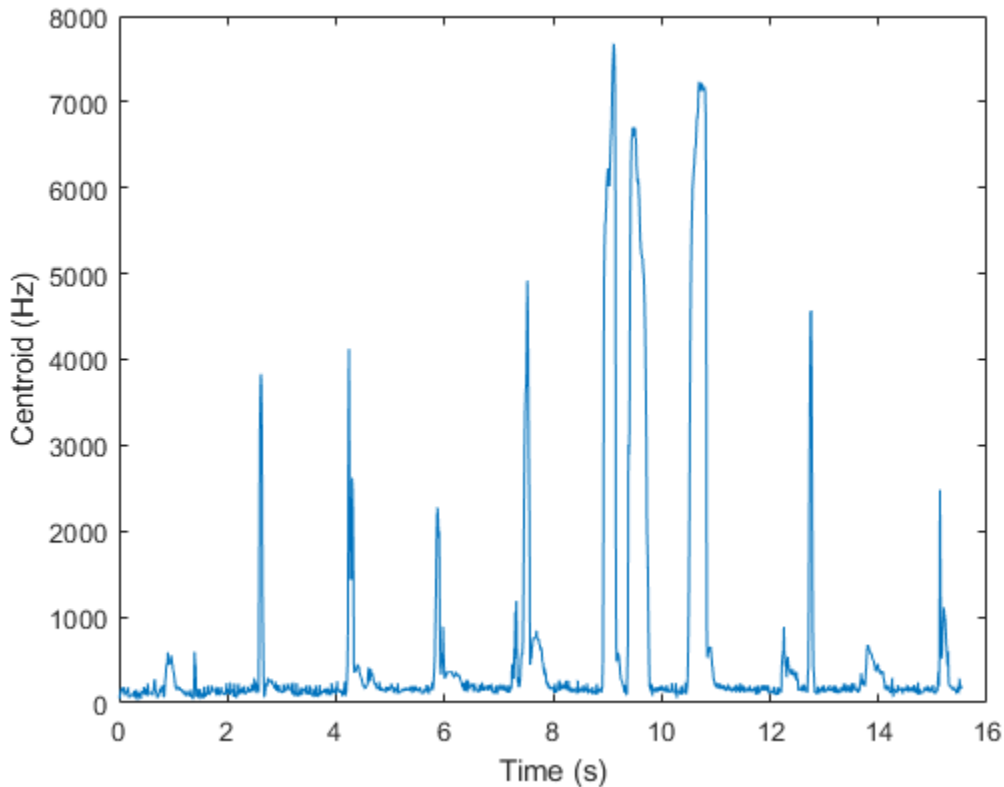
`centroid = spectralCentroid(x,f,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

### Examples

#### Spectral Centroid of Time-Domain Audio

Read in an audio file, calculate the centroid using default parameters, and then plot the results.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
centroid = spectralCentroid(audioIn,fs);  
  
t = linspace(0,size(audioIn,1)/fs,size centroid,1));  
plot(t,centroid)  
xlabel('Time (s)')  
ylabel('Centroid (Hz)')
```



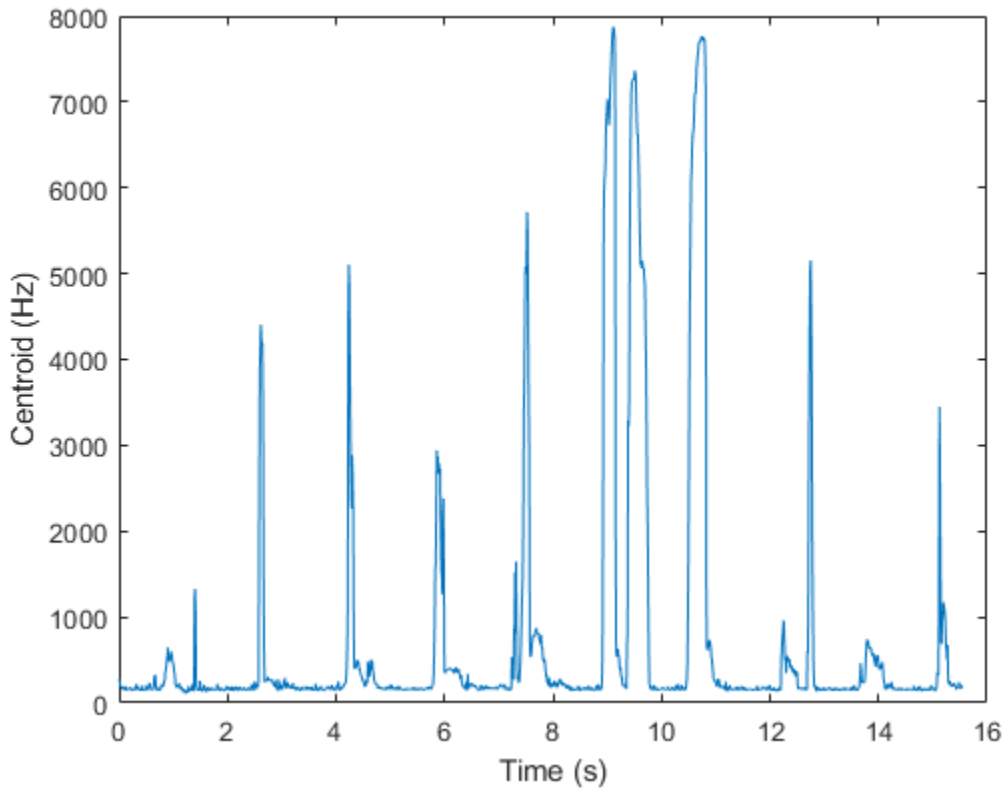
### Spectral Centroid of Frequency-Domain Audio Data

Read in an audio file and then buffer the signal into 30 ms frames with 20 ms overlap. Calculate the octave power spectrum using the `p octave` function.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
audioBuffered = buffer(audioIn,round(fs*0.03),round(fs*0.02),'nodelay');  
[p,cf] = p octave(audioBuffered,fs);
```

Calculate the centroid of the octave power spectrum over time. Plot the results.

```
centroid = spectralCentroid(p,cf);  
  
t = linspace(0,size(audioIn,1)/fs,size centroid,1);  
plot(t,centroid)  
xlabel('Time (s)')  
ylabel('Centroid (Hz)')
```



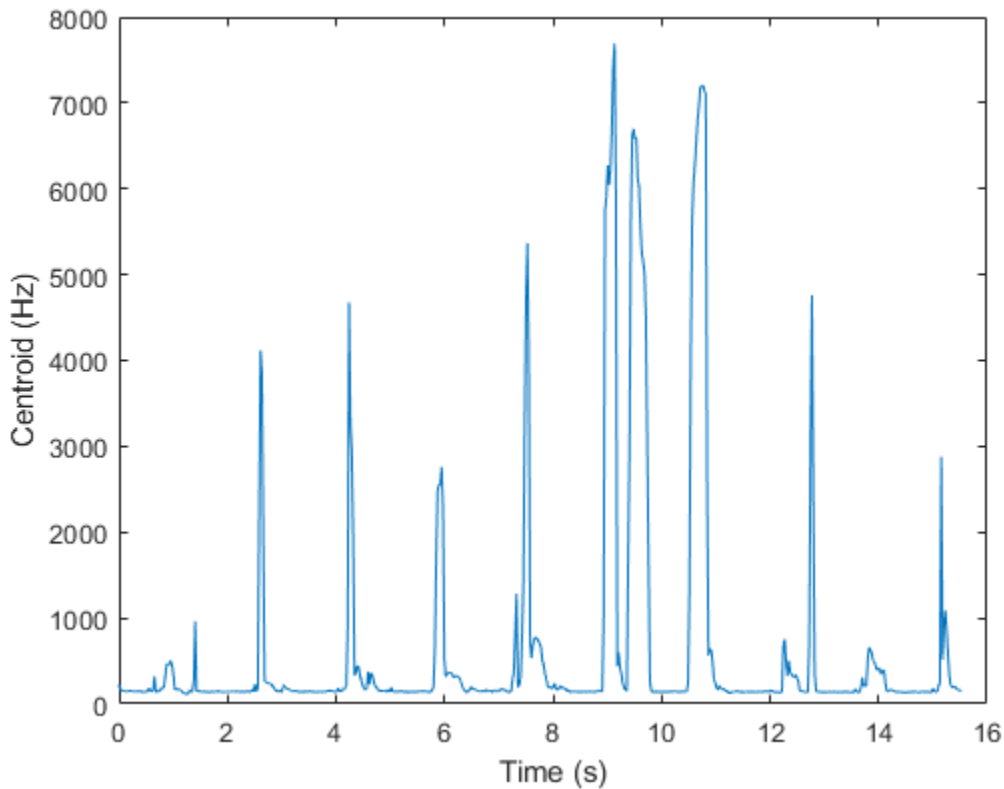
### Specify Nondefault Parameters

Read in an audio file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Calculate the centroid of the power spectrum over time. Calculate the centroid for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to  $f_s/2$  for the centroid calculation. Plot the results.

```
centroid = spectralCentroid(audioIn,fs, ...  
                            'Window',hamming(round(0.05*fs)), ...  
                            'OverlapLength',round(0.025*fs), ...  
                            'Range',[62.5,fs/2]);  
  
t = linspace(0,size(audioIn,1)/fs,size(centroid,1));  
plot(t,centroid)  
xlabel('Time (s)')  
ylabel('Centroid (Hz)')
```



### Calculate Spectral Centroid of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral centroid calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');  
logger = dsp.SignalSink;
```

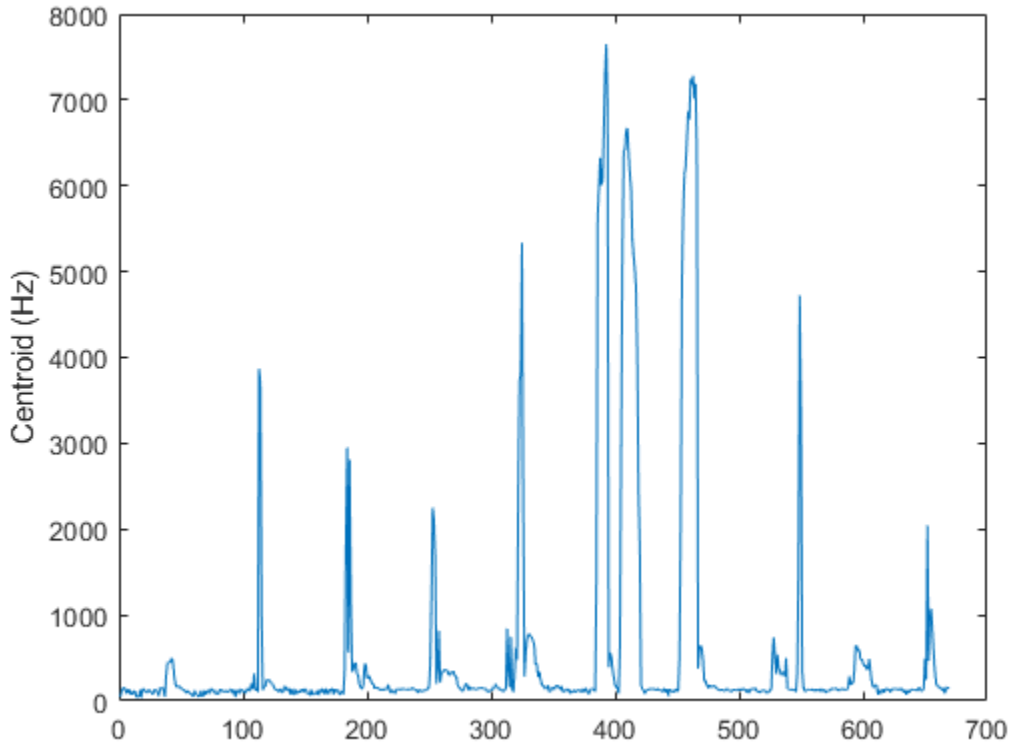
In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Calculate the spectral centroid for the frame of audio.
- 3 Log the spectral centroid for later plotting.

To calculate the spectral centroid for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero.

Plot the logged data.

```
while ~isDone(fileReader)  
    audioIn = fileReader();  
    centroid = spectralCentroid(audioIn,fileReader.SampleRate, ...  
                               'Window',hamming(size(audioIn,1)), ...  
                               'OverlapLength',0);  
    logger(centroid)  
end  
  
plot(logger.Buffer)  
ylabel('Centroid (Hz)')
```



If the input to your audio stream loop has a variable samples-per-frame, an inconsistent samples-per-frame with the analysis window size of `spectralCentroid`, or if you want to calculate the spectral centroid for overlapped data, use `dsp.AsyncBuffer`.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;  
reset(logger)  
release(fileReader)
```

Specify that the spectral centroid is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;
```

```
samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        centroid = spectralCentroid(audioBuffered,fs, ...
                                    'Window',win, ...
                                    'OverlapLength',0);

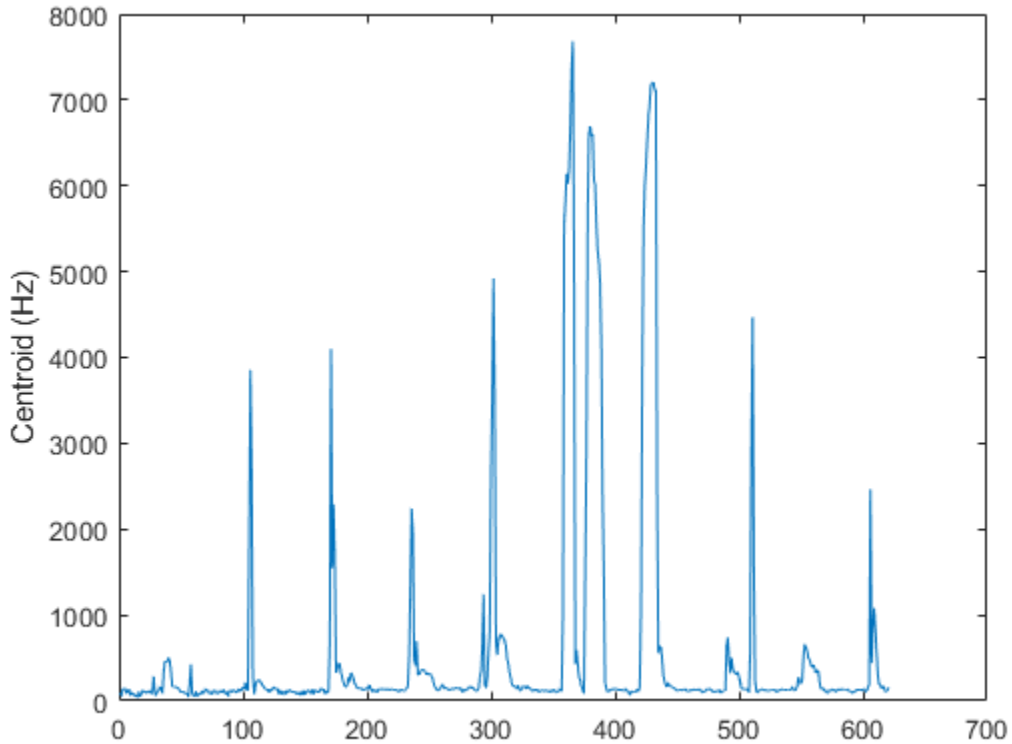
        logger(centroid)
    end
end

release(fileReader)

Plot the logged data.

plot(logger.Buffer)
ylabel('Centroid (Hz)')
```





## Input Arguments

### **x** — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets  $x$  depends on the shape of  $f$ .

Data Types: `single` | `double`

### **f** — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets `x` depends on the shape of `f`:

- If `f` is a scalar, `x` is interpreted as a time-domain signal, and `f` is interpreted as the sample rate. In this case, `x` must be a real vector or matrix. If `x` is specified as a matrix, the columns are interpreted as individual channels.
- If `f` is a vector, `x` is interpreted as a frequency-domain signal, and `f` is interpreted as the frequencies, in Hz, corresponding to the rows of `x`. In this case, `x` must be a real  $L$ -by- $M$ -by- $N$  array, where  $L$  is the number of spectral values at given frequencies of `f`,  $M$  is the number of individual spectrums, and  $N$  is the number of channels.
- The number of rows of `x`,  $L$ , must be equal to the number of elements of `f`.

Data Types: `single` | `double`

### Name-Value Pair Arguments

---

**Note** The following name-value pair arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, name-value pair arguments are ignored.

---

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

Example: `'Window', hamming(256)`

#### **Window — Window applied in time domain**

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range  $[1, \text{size}(x, 1)]$ . The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

#### **OverlapLength — Number of samples overlapped between adjacent windows**

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of 'OverlapLength' and an integer in the range [0, size(Window,1)).

Data Types: single | double

### **FFTLength — Number of bins in DFT**

numel(Window) (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as the comma-separated pair consisting of 'FFTLength' and a positive scalar integer. If unspecified, FFTLength defaults to the number of elements in the Window.

Data Types: single | double

### **Range — Frequency range (Hz)**

[0, f/2] (default) | two-element row vector

Frequency range in Hz, specified as the comma-separated pair consisting of 'Range' and a two-element row vector of increasing real values in the range [0, f/2].

Data Types: single | double

### **SpectrumType — Spectrum type**

'power' (default) | 'magnitude'

Spectrum type, specified as the comma-separated pair consisting of 'SpectrumType' and 'power' or 'magnitude':

- 'power' -- The spectral centroid is calculated for the one-sided power spectrum.
- 'magnitude' -- The spectral centroid is calculated for the one-sided magnitude spectrum.

Data Types: char | string

## **Output Arguments**

### **centroid — Spectral centroid (Hz)**

scalar | vector | matrix

Spectral centroid in Hz, returned as a scalar, vector, or matrix. Each row of `centroid` corresponds to the spectral centroid of a window of `x`. Each column of `centroid` corresponds to an independent channel.

### Algorithms

The spectral centroid is calculated as described in [1]:

$$\text{centroid} = \frac{\sum_{k=b_1}^{b_2} f_k s_k}{\sum_{k=b_1}^{b_2} s_k}$$

where

- $f_k$  is the frequency in Hz corresponding to bin  $k$ .
- $s_k$  is the spectral value at bin  $k$ .
- $b_1$  and  $b_2$  are the band edges, in bins, over which to calculate the spectral centroid.

### References

- [1] Peeters, G. "A Large Set of Audio Features for Sound Description (Similarity and Classification) in the CUIDADO Project." Technical Report; IRCAM: Paris, France, 2004.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`spectralKurtosis` | `spectralSkewness` | `spectralSpread`

## **Topics**

“Spectral Descriptors”

**Introduced in R2019a**

## hz2mel

Convert from hertz to mel scale

### Syntax

```
mel = hz2erb(hz)
```

### Description

`mel = hz2erb(hz)` converts values in hertz to values on the mel frequency scale.

### Examples

#### Convert Between Mel Scale and Hz

Set two bounding frequencies in Hz and then convert them to the mel scale.

```
b = hz2mel([20,8000]);
```

Generate a row vector of 32 values uniformly spaced on the mel scale.

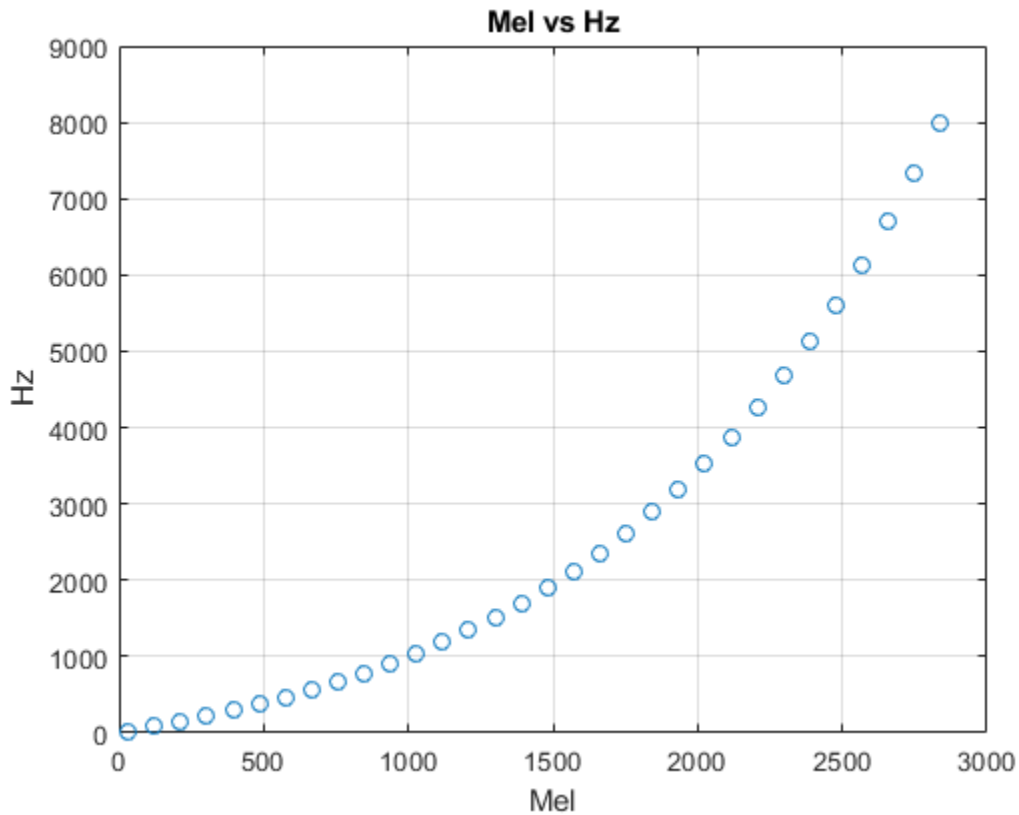
```
melVect = linspace(b(1),b(2),32);
```

Convert the row vector of values into equivalent frequencies in Hz.

```
hzVect = mel2hz(melVect);
```

Plot the two vectors for comparison. As mel values increase linearly, Hz values increase exponentially.

```
plot(melVect,hzVect,'o')  
title('Mel vs Hz')  
xlabel('Mel')  
ylabel('Hz')  
grid on
```



## Input Arguments

### **hz** — Input frequency in Hz

scalar | vector | matrix | multidimensional array

Input frequency in Hz, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: single | double

## Output Arguments

### **mel** — Output frequency on mel scale

scalar | vector | matrix | multidimensional array

Output frequency on the mel scale, returned as a scalar, vector, matrix, or multidimensional array the same size as `hz`.

Data Types: `single` | `double`

## Algorithms

The frequency conversion from Hz to the mel scale uses the following formula:

$$mel = 2595 \log_{10} \left( 1 + \frac{hz}{700} \right)$$

## References

- [1] O'Shaghnessy, Douglas. *Speech Communication: Human and Machine*. Reading, MA: Addison-Wesley Publishing Company, 1987.

## Extended Capabilities

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

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

### **See Also**

`bark2hz` | `erb2hz` | `hz2bark` | `hz2erb` | `mel2hz`

**Introduced in R2019a**



## hz2bark

Convert from hertz to Bark scale

### Syntax

```
bark = hz2erb(hz)
```

### Description

`bark = hz2erb(hz)` converts values in hertz to values on the Bark frequency scale.

### Examples

#### Convert Between Bark Scale and Hz

Set two bounding frequencies in Hz and then convert them to the Bark scale.

```
b = hz2bark([20,8000]);
```

Generate a row vector of 32 values uniformly spaced on the Bark scale.

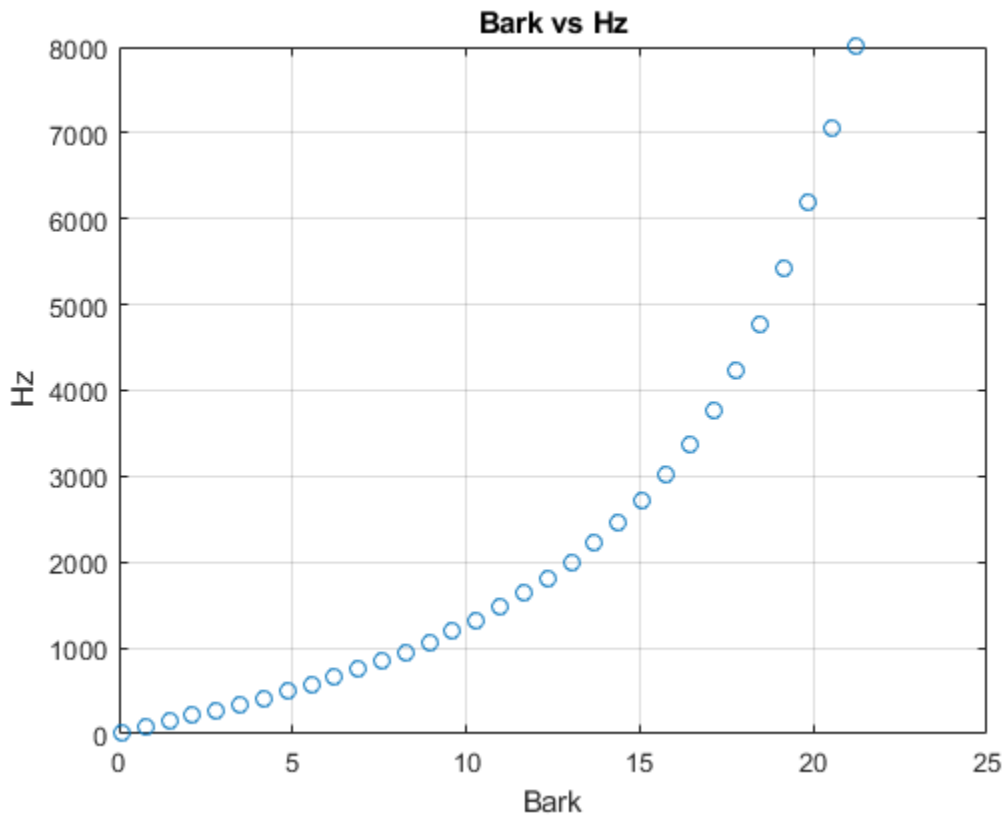
```
barkVect = linspace(b(1),b(2),32);
```

Convert the row vector of values into equivalent frequencies in Hz.

```
hzVect = bark2hz(barkVect);
```

Plot the two vectors for comparison. As Bark values increase linearly, Hz values increase exponentially.

```
plot(barkVect,hzVect,'o')  
title('Bark vs Hz')  
xlabel('Bark')  
ylabel('Hz')  
grid on
```



## Input Arguments

### **hz** — Input frequency in Hz

scalar | vector | matrix | multidimensional array

Input frequency in Hz, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: single | double

## Output Arguments

### **bark** — Output frequency on Bark scale

scalar | vector | matrix | multidimensional array

Output frequency on the Bark scale, returned as a scalar, vector, matrix, or multidimensional array the same size as `hz`.

Data Types: `single` | `double`

## Algorithms

The frequency conversion from Hz to the Bark scale uses the following formula:

$$bark = \frac{(26.81)(hz)}{1960 + hz} - 0.53$$

$$if: bark < 2 \rightarrow bark = bark + (0.15)(2 - bark)$$

$$if: bark > 20.1 \rightarrow bark = bark + (0.22)(bark - 20.1)$$

The Bark value correction occurs after the conversion from Hz to the Bark scale.

## References

- [1] Traunmüller, Hartmut. "Analytical Expressions for the Tonotopic Sensory Scale." *Journal of the Acoustical Society of America*. Vol. 88, Issue 1, 1990, pp. 97-100.

## Extended Capabilities

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

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

### **See Also**

`bark2hz` | `erb2hz` | `hz2erb` | `hz2mel` | `mel2hz`

**Introduced in R2019a**

## hz2erb

Convert from hertz to equivalent rectangular bandwidth (ERB) scale

### Syntax

```
erb = hz2erb(hz)
```

### Description

`erb = hz2erb(hz)` converts values in hertz to values on the ERB frequency scale.

### Examples

#### Convert Between ERB Scale and Hz

Set two bounding frequencies in Hz and then convert them to the ERB scale.

```
b = hz2erb([20,8000]);
```

Generate a row vector of 32 values uniformly spaced on the ERB scale.

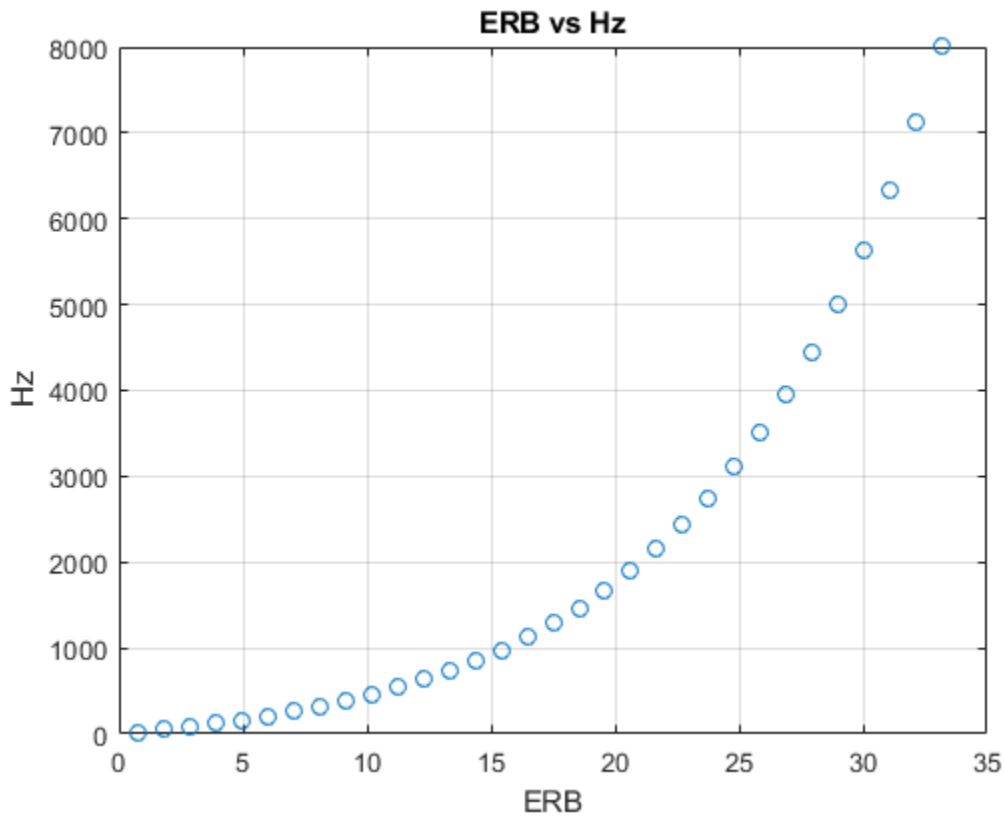
```
erbVect = linspace(b(1),b(2),32);
```

Convert the row vector of values into equivalent frequencies in Hz.

```
hzVect = erb2hz(erbVect);
```

Plot the two vectors for comparison. As ERB values increase linearly, Hz values increase exponentially.

```
plot(erbVect,hzVect,'o')  
title('ERB vs Hz')  
xlabel('ERB')  
ylabel('Hz')  
grid on
```



## Input Arguments

### hz — Input frequency in Hz

scalar | vector | matrix | multidimensional array

Input frequency in Hz, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: single | double

## Output Arguments

### **erb** — Output frequency on ERB scale

scalar | vector | matrix | multidimensional array

Output frequency on the ERB scale, returned as a scalar, vector, matrix, or multidimensional array the same size as `hz`.

Data Types: `single` | `double`

## Algorithms

The frequency conversion from Hz to the ERB scale uses the following formula:

$$erb = A \log_{10}(1 + hz(0.00437))$$

where

$$A = \frac{1000 \log_e(10)}{(24.7)(4.37)}$$

## References

- [1] Glasberg, Brian R., and Brian C. J. Moore. "Derivation of Auditory Filter Shapes from Notched-Noise Data." *Hearing Research*. Vol. 47, Issues 1-2, 1990, pp. 103-138.

## Extended Capabilities

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

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

### **See Also**

`bark2hz` | `erb2hz` | `hz2bark` | `hz2mel` | `mel2hz`

**Introduced in R2019a**



## mel2hz

Convert from mel scale to hertz

### Syntax

```
hz = mel2hz(mel)
```

### Description

`hz = mel2hz(mel)` converts values on the mel frequency scale to values in hertz.

### Examples

#### Convert Between Mel Scale and Hz

Set two bounding frequencies in Hz and then convert them to the mel scale.

```
b = hz2mel([20,8000]);
```

Generate a row vector of 32 values uniformly spaced on the mel scale.

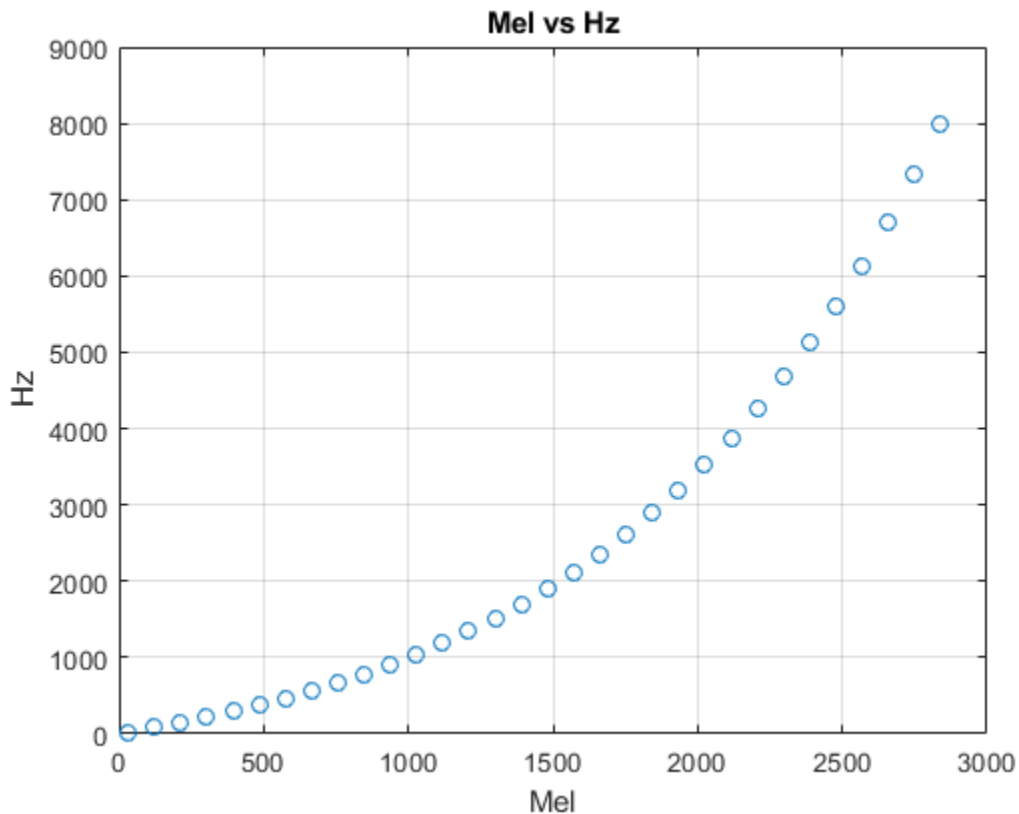
```
melVect = linspace(b(1),b(2),32);
```

Convert the row vector of values into equivalent frequencies in Hz.

```
hzVect = mel2hz(melVect);
```

Plot the two vectors for comparison. As mel values increase linearly, Hz values increase exponentially.

```
plot(melVect,hzVect,'o')  
title('Mel vs Hz')  
xlabel('Mel')  
ylabel('Hz')  
grid on
```



## Input Arguments

### **mel** — Input frequency on mel scale

scalar | vector | matrix | multidimensional array

Input frequency on the mel scale, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: single | double

## Output Arguments

### hz — Output frequency in Hz

scalar | vector | matrix | multidimensional array

Output frequency in Hz, returned as a scalar, vector, matrix, or multidimensional array the same size as `mel`.

Data Types: `single` | `double`

## Algorithms

The frequency conversion from the mel scale to Hz uses the following formula:

$$hz = 700 \left( 10^{\frac{mel}{2595}} - 1 \right)$$

## References

- [1] O'Shaghnessy, Douglas. *Speech Communication: Human and Machine*. Reading, MA: Addison-Wesley Publishing Company, 1987.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`bark2hz` | `erb2hz` | `hz2bark` | `hz2erb` | `hz2mel`

**Introduced in R2019a**

## bark2hz

Convert from Bark scale to hertz

### Syntax

```
hz = bark2hz(bark)
```

### Description

`hz = bark2hz(bark)` converts values on the Bark frequency scale to values in hertz.

### Examples

#### Convert Between Bark Scale and Hz

Set two bounding frequencies in Hz and then convert them to the Bark scale.

```
b = hz2bark([20,8000]);
```

Generate a row vector of 32 values uniformly spaced on the Bark scale.

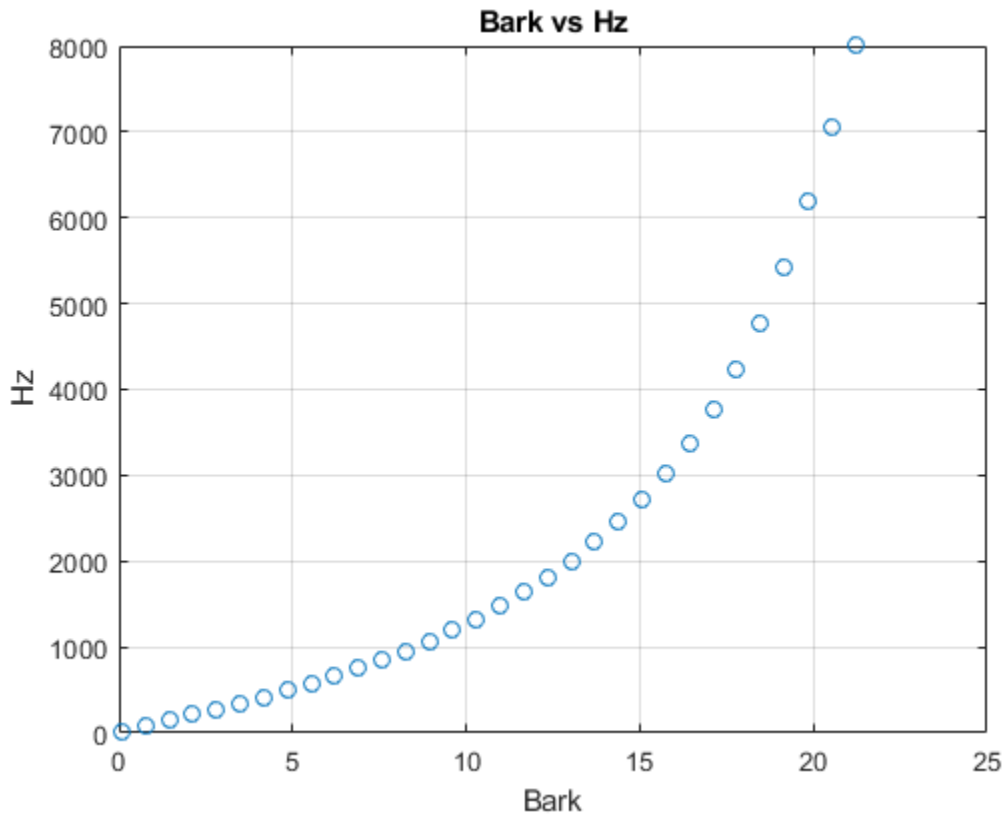
```
barkVect = linspace(b(1),b(2),32);
```

Convert the row vector of values into equivalent frequencies in Hz.

```
hzVect = bark2hz(barkVect);
```

Plot the two vectors for comparison. As Bark values increase linearly, Hz values increase exponentially.

```
plot(barkVect,hzVect,'o')  
title('Bark vs Hz')  
xlabel('Bark')  
ylabel('Hz')  
grid on
```



## Input Arguments

### **bark** — Input frequency on Bark scale

scalar | vector | matrix | multidimensional array

Input frequency on the Bark scale, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: single | double

## Output Arguments

### **hz** — Output frequency in Hz

scalar | vector | matrix | multidimensional array

Output frequency in Hz, returned as a scalar, vector, matrix, or multidimensional array the same size as `bark`.

Data Types: `single` | `double`

## Algorithms

The frequency conversion from the Bark scale to Hz uses the following formula:

$$\text{if: } bark < 2 \rightarrow bark = \frac{(bark - 0.3)}{0.85}$$

$$\text{if: } bark > 20.1 \rightarrow bark = \frac{(bark + 4.422)}{1.22}$$

$$hz = 1960 \left( \frac{bark + 0.53}{26.28 - bark} \right)$$

The Bark value correction occurs before the conversion from the Bark scale to Hz.

## References

- [1] Traunmüller, Hartmut. "Analytical Expressions for the Tonotopic Sensory Scale." *Journal of the Acoustical Society of America*. Vol. 88, Issue 1, 1990, pp. 97-100.

## Extended Capabilities

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

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

## **See Also**

erb2hz | hz2bark | hz2erb | hz2mel | mel2hz

**Introduced in R2019a**

## erb2hz

Convert from equivalent rectangular bandwidth (ERB) scale to hertz

### Syntax

```
hz = erb2hz(erb)
```

### Description

`hz = erb2hz(erb)` converts values on the ERB frequency scale to values in hertz.

### Examples

#### Convert Between ERB Scale and Hz

Set two bounding frequencies in Hz and then convert them to the ERB scale.

```
b = hz2erb([20,8000]);
```

Generate a row vector of 32 values uniformly spaced on the ERB scale.

```
erbVect = linspace(b(1),b(2),32);
```

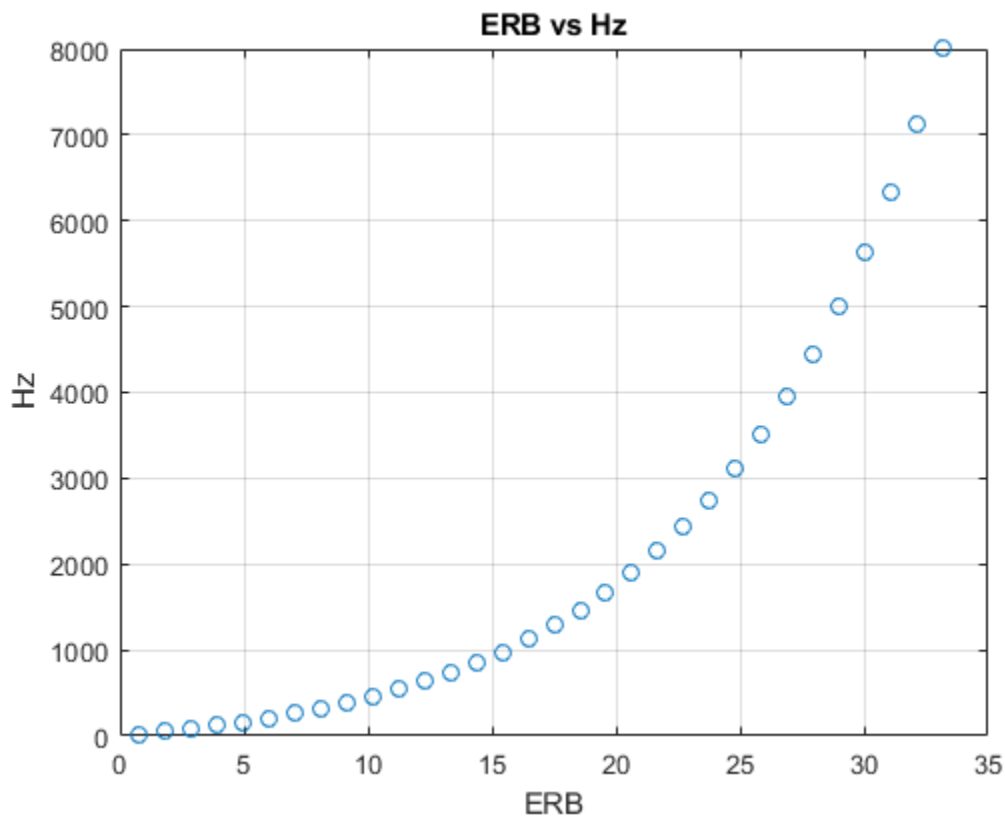
Convert the row vector of values into equivalent frequencies in Hz.

```
hzVect = erb2hz(erbVect);
```

Plot the two vectors for comparison. As ERB values increase linearly, Hz values increase exponentially.

```
plot(erbVect,hzVect,'o')  
title('ERB vs Hz')  
xlabel('ERB')  
ylabel('Hz')  
grid on
```





## Input Arguments

### **erb** — Input frequency on ERB scale

scalar | vector | matrix | multidimensional array

Input frequency on the equivalent rectangular band (ERB) scale, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: single | double

## Output Arguments

### **hz** — Output frequency in Hz

scalar | vector | matrix | multidimensional array

Output frequency in Hz, returned as a scalar, vector, matrix, or multidimensional array the same size as `erb`.

Data Types: `single` | `double`

## Algorithms

The frequency conversion from the ERB scale to Hz uses the following formula:

$$hz = \frac{10^{\frac{erb}{A}} - 1}{0.00437}$$

where

$$A = \frac{1000 \log_e(10)}{(24.7)(4.37)}$$

## References

- [1] Glasberg, Brian R., and Brian C. J. Moore. "Derivation of Auditory Filter Shapes from Notched-Noise Data." *Hearing Research*. Vol. 47, Issues 1-2, 1990, pp. 103-138.

## Extended Capabilities

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

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

### **See Also**

`bark2hz` | `hz2bark` | `hz2erb` | `hz2mel` | `mel2hz`

**Introduced in R2019a**

## mls

Maximum length sequence

### Syntax

```
excitation = mls  
excitation = mls(L)  
excitation = mls(L,Name,Value)
```

### Description

`excitation = mls` returns an excitation signal generated using the maximum length sequence (MLS) technique. This type of sequence is a pseudo-random binary sequence.

`excitation = mls(L)` specifies the output length `L` of the excitation signal.

`excitation = mls(L,Name,Value)` specifies options using one or more `Name,Value` pair arguments, in addition to the input arguments in the previous syntaxes.

### Examples

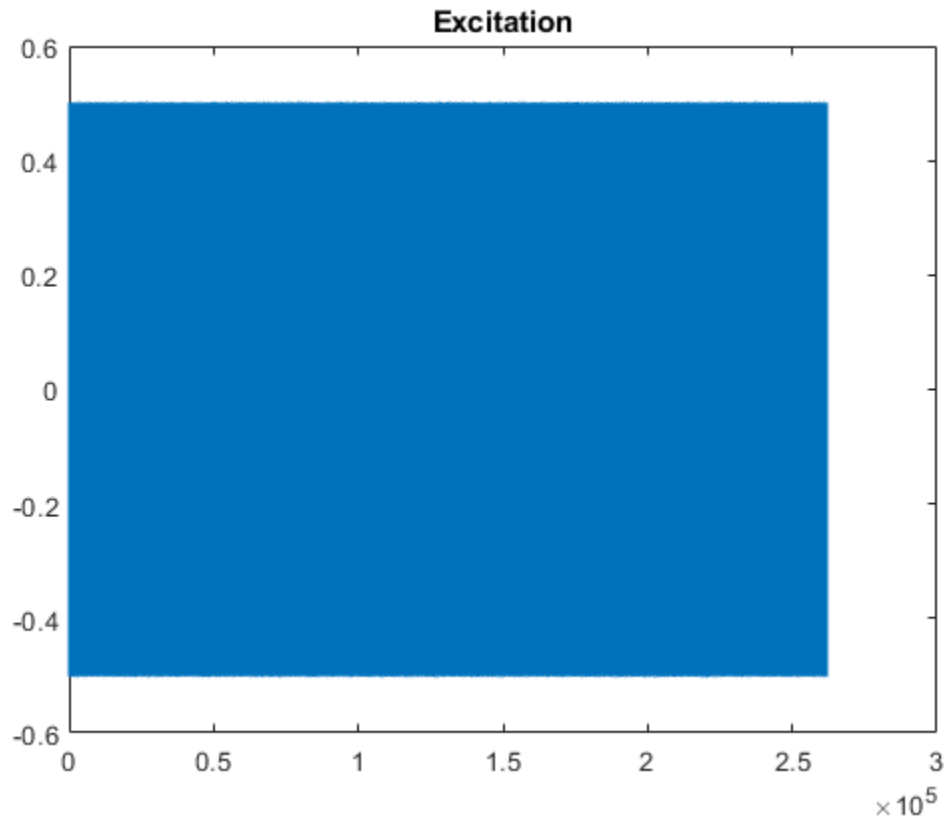
#### Estimate Impulse Response Using MLS Excitation

Use `audioread` to read in an impulse response recording. Create a `dsp.FrequencyDomainFIRFilter` object to perform frequency domain filtering using the known impulse response.

```
[irKnown,fs] = audioread('ChurchImpulseResponse-16-44p1-mono-5secs.wav');  
systemModel = dsp.FrequencyDomainFIRFilter(irKnown');
```

Create an MLS excitation signal by using the `mls` function. The MLS excitation signal must be longer than the impulse response. Note that the length of the MLS excitation is extended to the next power of two minus one.

```
excitation = mls(numel(irKnown)+1);  
  
plot(excitation)  
title('Excitation')
```

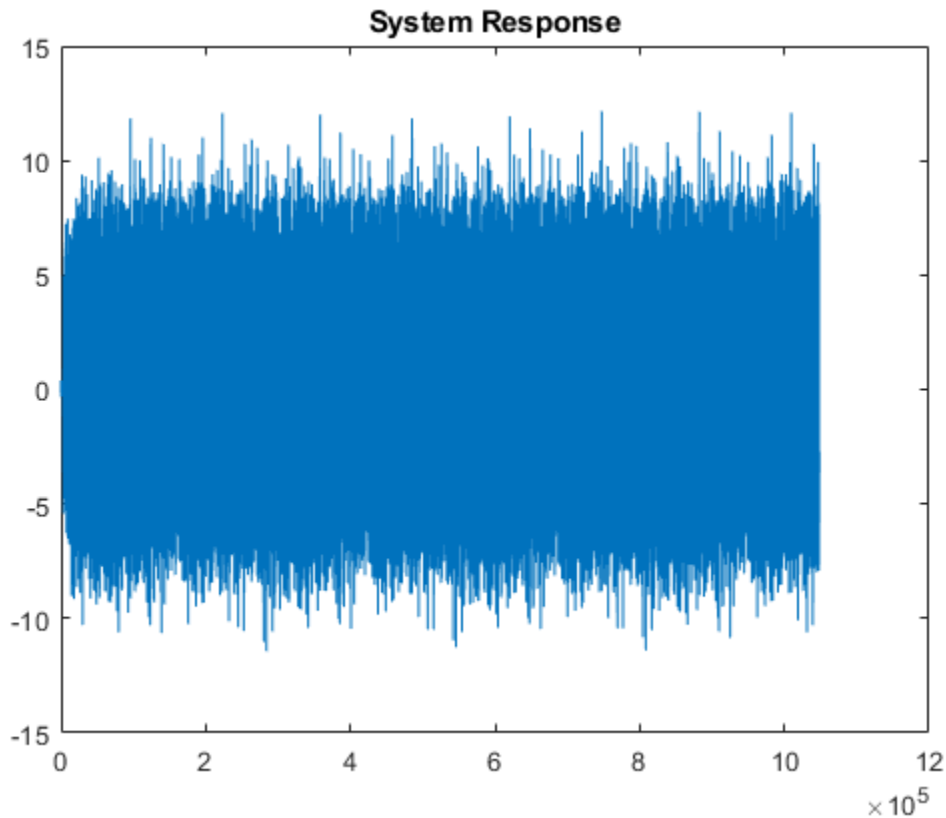


Replicate the excitation signal four times to measure the average of three measurements. The recording of the first MLS sequence does include all the impulse response information, so `impzest` discards it as a warmup run. Pad the excitation signal with zeros to account for the filter latency.

```
numRuns = 4;  
excrep = repmat(excitation,numRuns,1);  
excrep = [excrep;zeros(numel(irKnown)+1,1)];
```

Pass the excitation signal through the known filter and then add noise to model a real-world recording (system response). Cut the delay introduced at the beginning by the filter.

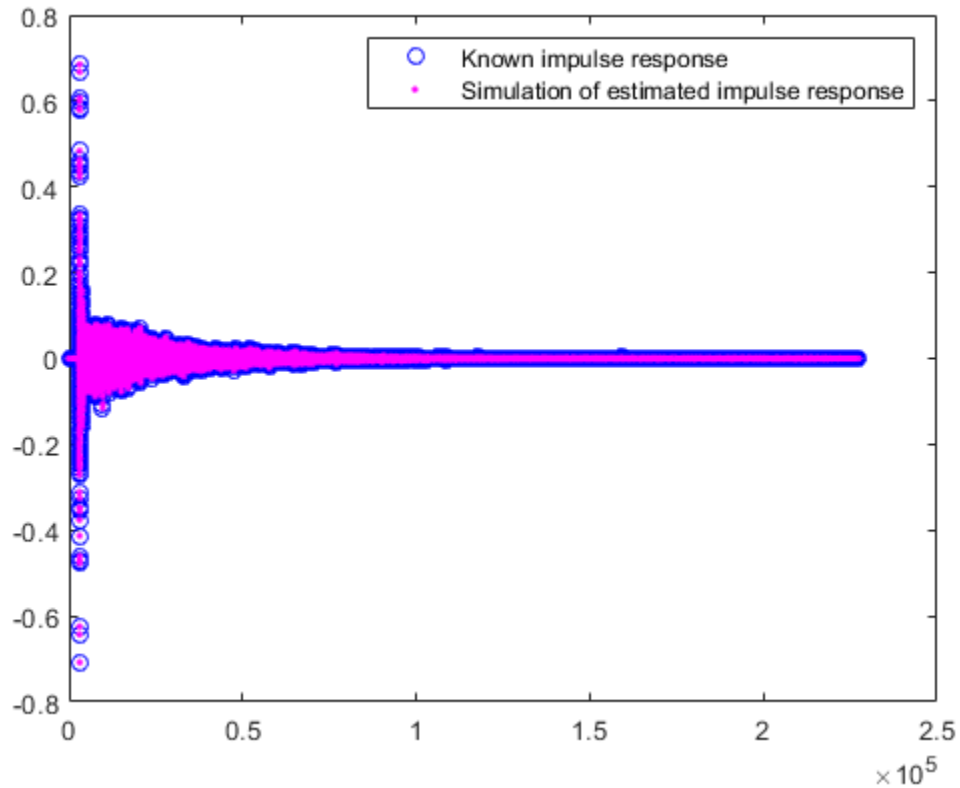
```
rec = systemModel(excprep);  
rec = rec + 0.1*randn(size(rec));  
  
rec = rec(numel(irKnown)+2:end,:);  
  
plot(rec)  
title('System Response')
```



In a real-world scenario, the MLS sequence is played back in the system under test while recording. The recording would be cut so that it begins at the moment the MLS sequence is picked-up and truncated to last the duration of the repeated sequence.

Pass the excitation signal and the system response to the `impzest` function to estimate the impulse response. Plot the known impulse response and the simulation of the estimated impulse response for comparison.

```
irEstimate = impzest(excitation,rec);  
  
samples = 1:numel(irKnown);  
plot(samples,irEstimate(samples),'bo', ...  
      samples,irKnown(samples),'m.')  
  
legend('Known impulse response','Simulation of estimated impulse response')
```



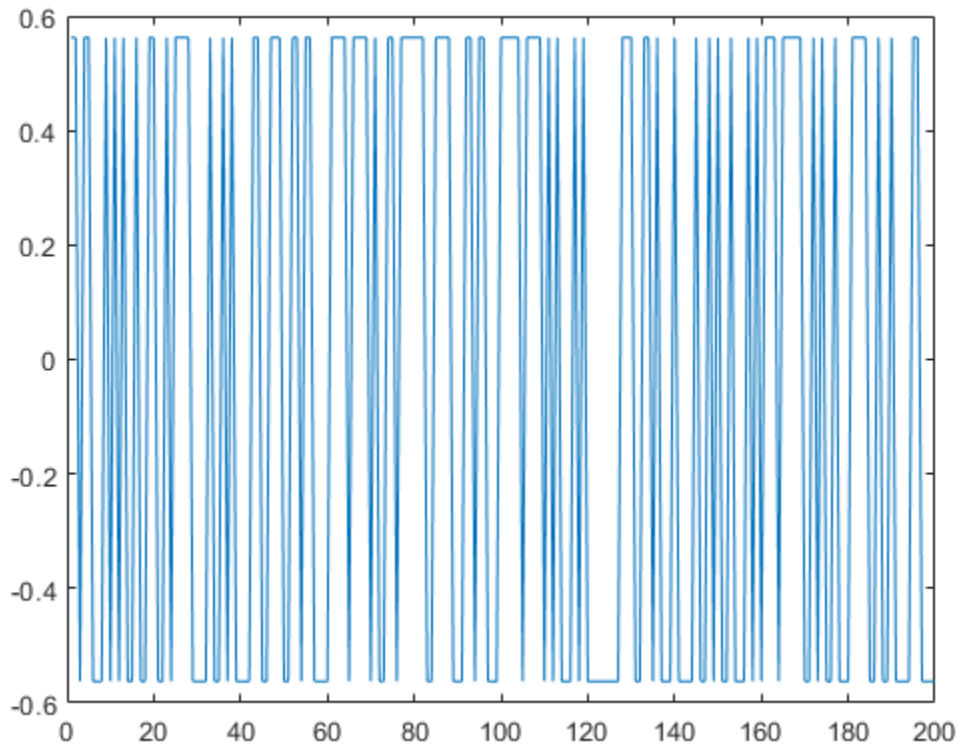
### Generate MLS Signal

Generate an MLS signal that is  $2^{14}-1$  samples long and has a level of -5 dB.

```
L = 2^14-1;  
level = -5;  
excitation = mls(L, 'ExcitationLevel', level);
```

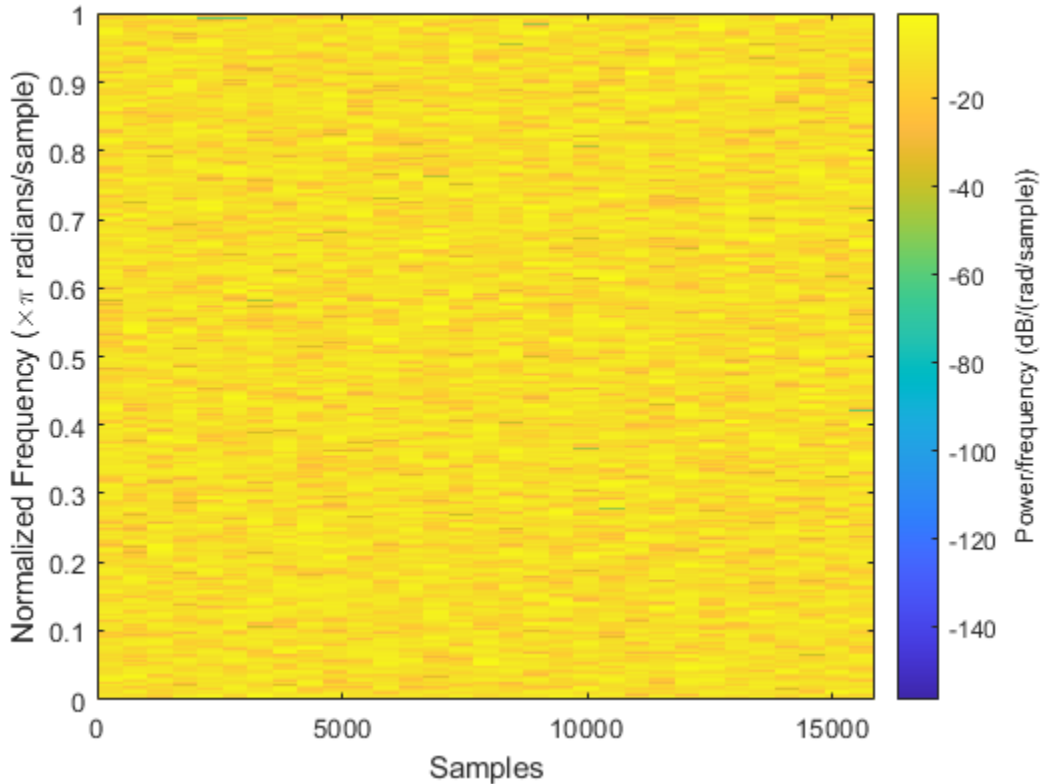
Visualize the excitation in time and time-frequency. For the time-domain plot, plot only the first 200 samples for visibility. The pattern is constant.

```
plot(excitation(1:200))
```



```
spectrogram(excitation, 512, 0, 1024, 'yaxis')
```





## Input Arguments

### **L** — Length of excitation signal

32767 (default) | scalar in the range  $[3, 2^{29}]$

Length of excitation signal to generate, specified as a scalar in the range  $[3, 2^{29}]$ .

The requested output length  $L$  must be a power of two minus one. Otherwise, the output length increases to the next valid length.

---

**Note** If you use the excitation signal generated by the `mls` function to record and estimate the impulse response of a system, then the length of the excitation signal must be at least as long as the impulse response that you want to estimate.

---

Data Types: `single` | `double`

### 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'ExcitationLevel', -5`

#### **ExcitationLevel** — Level of the excitation signal to generate (dB)

scalar in the range `[-42, 0]`

Level of the excitation signal to generate in dB, specified as a scalar in the range `[-42, 0]`.

Data Types: `single` | `double`

### Output Arguments

#### **excitation** — Excitation signal

column vector

Excitation signal generated using the maximum length sequence (MLS) technique, returned as a column vector.

Data Types: `single` | `double`

### References

- [1] Guy-Bart, Stan, Jean-Jacques Embrachts, and Dominique Archambeau. "Comparison of Different Impulse Response Measurement Techniques." *Journal of Audio Engineering Society*. Vol. 50, Issue 4, 2002, pp. 246-262.

## **See Also**

**Impulse Response Measurer** | `impzest` | `sweeptone`

**Introduced in R2018b**

## sweeptone

Exponential swept sine

### Syntax

```
excitation = sweeptone()  
excitation = sweeptone(swDur)  
excitation = sweeptone(swDur,silDur)  
excitation = sweeptone(swDur,silDur,fs)  
excitation = sweeptone( ____,Name,Value)
```

### Description

`excitation = sweeptone()` returns an excitation signal generated using the exponential swept sine (ESS) technique. By default, the signal has a 6-second duration, followed by 4 seconds of silence, for a sample rate of 44100 Hz.

`excitation = sweeptone(swDur)` specifies the duration of the exponential swept sine signal.

`excitation = sweeptone(swDur,silDur)` specifies the duration of the silence following the exponential swept sine signal.

`excitation = sweeptone(swDur,silDur,fs)` specifies the sample rate of the sweep tone as `fs` Hz.

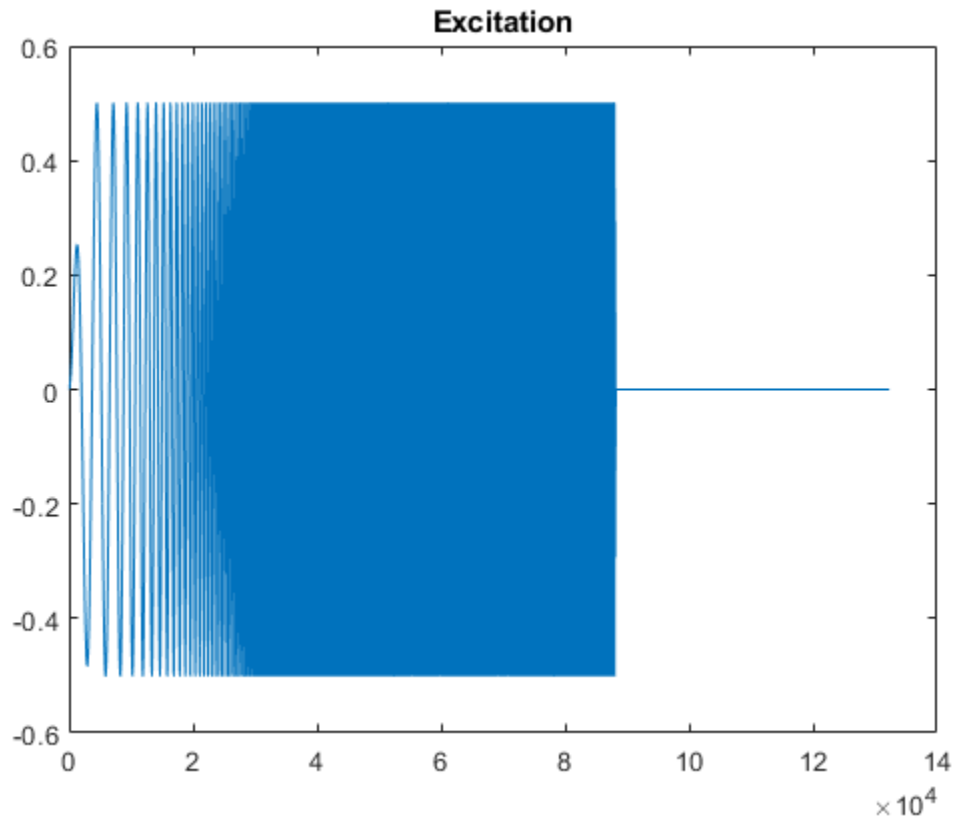
`excitation = sweeptone( ____,Name,Value)` specifies options using one or more `Name,Value` pair arguments, in addition to the input arguments in the previous syntaxes.

### Examples

#### Estimate Impulse Response Using Sweep Tone Excitation

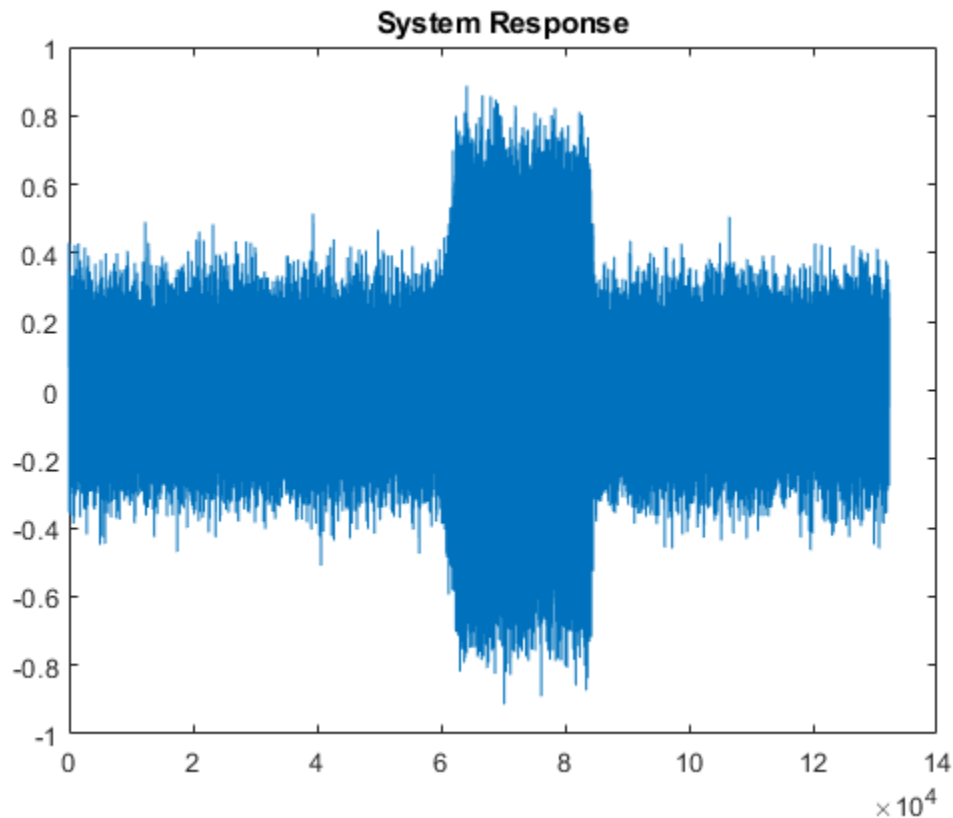
Create a sweep tone excitation signal by using the `sweeptone` function.

```
excitation = sweeptone(2,1,44100);  
  
plot(excitation)  
title('Excitation')
```



Pass the excitation signal through an infinite impulse response (IIR) filter and add noise to model a real-world recording (system response).

```
[B,A] = butter(10,[.1 .7]);  
rec = filter(B,A,excitation);  
nrec = rec + 0.12*randn(size(rec));  
  
plot(nrec)  
title('System Response')
```

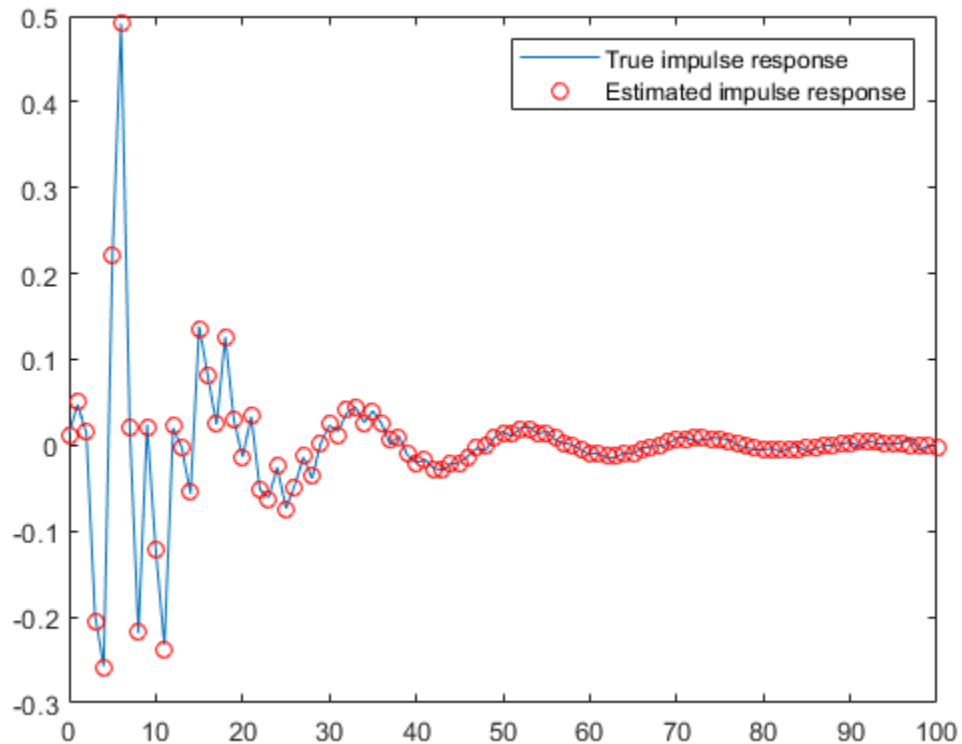


Pass the excitation signal and the system response to the `impzest` function to estimate the impulse response. Truncate the estimate to 100 points. Use `impz` to determine the true impulse response of the system. Plot the true impulse response and the estimated impulse response for comparison.

```
irEstimate = impzest(excitation,nrec);
irEstimate = irEstimate(1:101);

irTrue = impz(B,A,101);
plot(0:100,irEstimate, ...
     0:100,irTrue,'ro')

legend('True impulse response','Estimated impulse response')
```



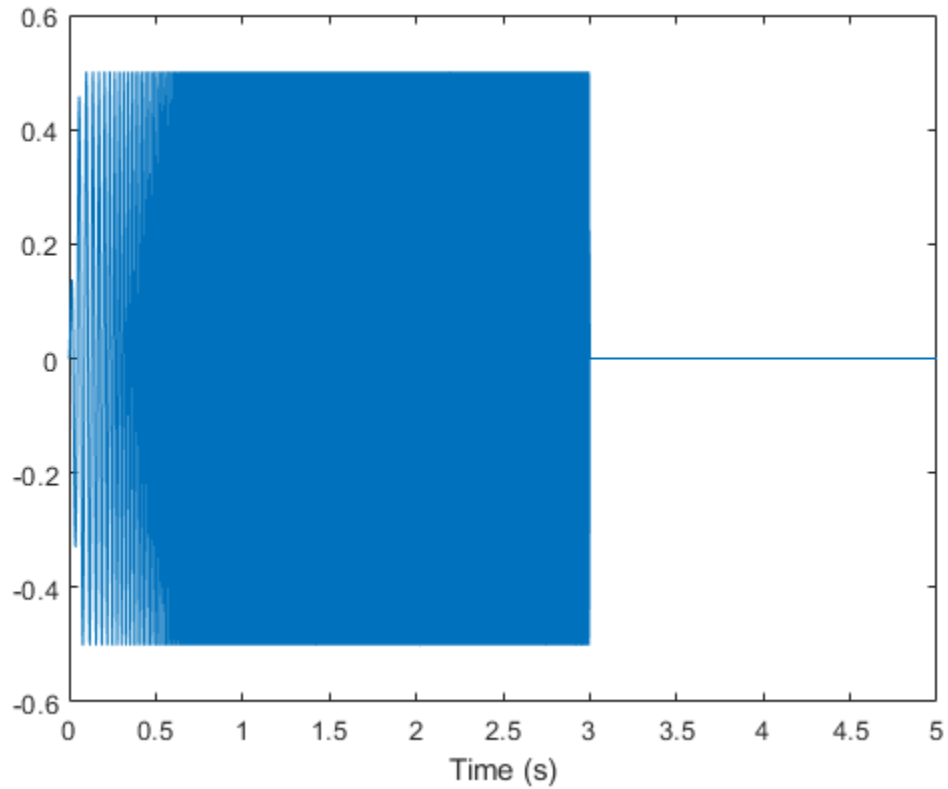
### Generate ESS Signal

Generate an exponential swept sine (ESS) signal with a 3-second sweep that goes from 20 Hz to 20 kHz, and ends with a 2-second silence. Specify the sample rate as 48 kHz.

```
fs = 48e3;  
excitation = sweeptone(3,2,fs, 'SweepFrequencyRange', [20 20e3]);
```

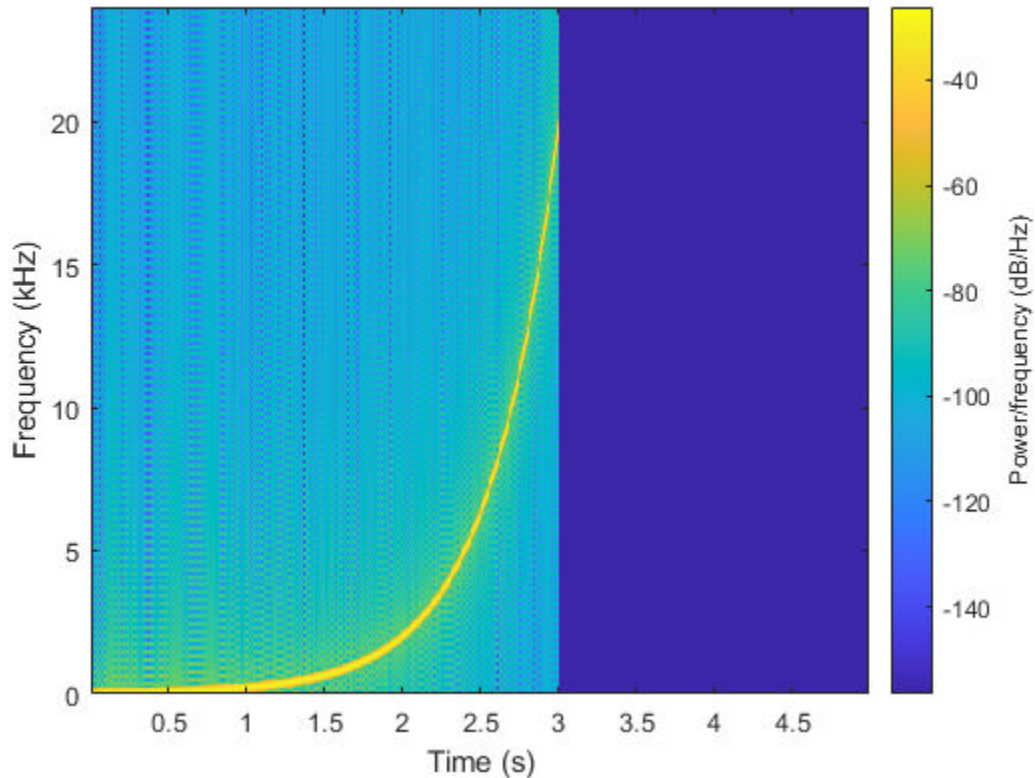
Visualize the excitation in time and time-frequency.

```
t = (0:numel(excitation)-1)/fs;  
plot(t,excitation)  
xlabel('Time (s)')
```



```
spectrogram(excitation,512,0,1024,fs,'yaxis')
```





## Input Arguments

### **swDur — Duration of exponential swept sine signal (s)**

6 (default) | scalar in the range [0.5,15]

Duration of exponential swept sine signal in seconds, specified as a scalar in the range [0.5,15].

The total duration of the excitation signal must be less than or equal to 15 seconds:  
 $swDur + silDur \leq 15$ .

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

### **silDur — Duration of silence after exponential swept sine signal (s)**

4 (default) | scalar in the range (0,14.5]

Duration of silence after exponential swept sine, specified as a scalar in the range (0,14.5].

The total duration of the excitation signal must be less than or equal to 15 seconds:  $swDur + silDur \leq 15$ .

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

### **fs — Sample rate (Hz)**

44100 (default) | positive scalar

Sample rate in Hz, specified as a positive scalar.

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

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'ExcitationLevel', -5`

### **ExcitationLevel — Level of excitation signal to generate (dB)**

-6 (default) | scalar in the range [-42, 0]

Level of the excitation signal to generate in dB, specified as a scalar in the range [-42, 0].

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

### **SweepFrequencyRange — Range of sweep frequency (Hz)**

[10 22000] | two-element positive row vector

Range of sweep frequency in Hz, specified as a two-element row vector. The sweep frequency range can be specified low to high or high to low. That is, [10 22000] and [22000 10] are both valid inputs. The largest value of the sweep frequency range must be less than or equal to  $f_s/2$ .

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

## Output Arguments

### **excitation** — Excitation signal

column vector

Excitation signal generated using the ESS technique, returned as a column vector. The length of the column vector is approximately  $(swDur+silDur)*fs$  samples.

Data Types: `double`

## References

- [1] Farino, Angelo. "Advancements in Impulse Response Measurements by Sine Sweeps." Presented at the *Audio Engineering Society 122nd Convention*, Vienna, Austria, 2007.

## See Also

**Impulse Response Measurer** | `impzest` | `mls`

**Introduced in R2018b**

## interpolateHRTF

3-D head-related transfer function (HRTF) interpolation

### Syntax

```
interpolatedHRTF = interpolateHRTF(HRTF,sourcePositions,  
desiredSourcePositions)  
interpolatedHRTF = interpolateHRTF( ____,Name,Value)
```

### Description

`interpolatedHRTF = interpolateHRTF(HRTF,sourcePositions,desiredSourcePositions)` returns the interpolated head-related transfer function (HRTF) at the desired position.

`interpolatedHRTF = interpolateHRTF( ____,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

### Examples

#### Render 3-D Audio on Headphones

Modify the 3-D audio image of a sound file by filtering it through a head-related transfer function (HRTF). Set the location of the sound source by specifying the desired azimuth and elevation.

```
load 'ReferenceHRTF.mat' hrtfData sourcePosition  
hrtfData = permute(double(hrtfData),[2,3,1]);  
sourcePosition = sourcePosition(:,[1,2]);
```

Calculate the head-related impulse response (HRIR) using the VBAP algorithm at a desired source position. Separate the output, `interpolatedIR`, into the impulse responses for the left and right ears.

```
desiredAz = 110;
desiredEl = -45;
desiredPosition = [desiredAz desiredEl];

interpolatedIR = interpolateHRTF(hrtfData,sourcePosition,desiredPosition, ...
                                "Algorithm","VBAP");

leftIR = squeeze(interpolatedIR(:,1,:));
rightIR = squeeze(interpolatedIR(:,2,:));
```

Create a `dsp.AudioFileReader` object to read in a file frame by frame. Create an `audioDeviceWriter` object to play audio to your sound card frame by frame. Create two `dsp.FIRFilter` objects and specify the filter coefficients using the head-related transfer function interpolated impulse responses.

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

leftFilter = dsp.FIRFilter('Numerator',leftIR);
rightFilter = dsp.FIRFilter('Numerator',rightIR);
```

In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Feed the stereo audio data through the left and right HRIR filters, respectively.
- 3 Concatenate the left and right channels and write the audio to your output device.

```
while ~isDone(fileReader)
    audioIn = fileReader();

    leftChannel = leftFilter(audioIn(:,1));
    rightChannel = rightFilter(audioIn(:,2));

    deviceWriter([leftChannel,rightChannel]);
end
```

As a best practice, release your System objects when complete.

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

### Model Moving Source Using HRIR Filtering

Create arrays of head-related impulse responses corresponding to desired source positions. Filter mono input to model a moving source.

Load the ARI HRTF dataset. Cast the `hrtfData` to type double, and reshape it to the required dimensions: (number of source positions)-by-2-by-(number of HRTF samples). Use the first two columns of the `sourcePosition` matrix only, which correspond to the azimuth and elevation of the source in degrees.

```
load 'ReferenceHRTF.mat' hrtfData sourcePosition  
  
hrtfData = permute(double(hrtfData),[2,3,1]);  
  
sourcePosition = sourcePosition(:,[1,2]);
```

Specify the desired source positions and then calculate the HRTF at these locations using the `interpolateHRTF` function. Separate the output, `interpolatedIR`, into the impulse responses for the left and right ears.

```
desiredAz = [-120;-60;0;60;120;0;-120;120];  
desiredEl = [-90;90;45;0;-45;0;45;45];  
desiredPosition = [desiredAz desiredEl];  
  
interpolatedIR = interpolateHRTF(hrtfData,sourcePosition,desiredPosition);  
  
leftIR = squeeze(interpolatedIR(:,1,:));  
rightIR = squeeze(interpolatedIR(:,2,:));
```

Create a `dsp.AudioFileReader` object to read in a file frame by frame. Create an `audioDeviceWriter` object to play audio to your sound card frame by frame. Create two `dsp.FIRFilter` objects with `NumeratorSource` set to `Input port`. Setting `NumeratorSource` to `Input port` enables you to modify the filter coefficients while streaming.

```
leftFilter = dsp.FIRFilter('NumeratorSource','Input port');  
rightFilter = dsp.FIRFilter('NumeratorSource','Input port');  
  
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');  
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Feed the audio data through the left and right HRIR filters.
- 3 Concatenate the left and right channels and write the audio to your output device. If you have a stereo output hardware, such as headphones, you can hear the source shifting position over time.
- 4 Modify the desired source position in 2-second intervals by updating the filter coefficients.

```

durationPerPosition = 2;
samplesPerPosition = durationPerPosition*fileReader.SampleRate;
samplesPerPosition = samplesPerPosition - rem(samplesPerPosition,fileReader.SamplesPerFrame);

sourcePositionIndex = 1;
samplesRead = 0;
while ~isDone(fileReader)
    audioIn = fileReader();
    samplesRead = samplesRead + fileReader.SamplesPerFrame;

    leftChannel = leftFilter(audioIn,leftIR(sourcePositionIndex,:));
    rightChannel = rightFilter(audioIn,rightIR(sourcePositionIndex,:));

    deviceWriter([leftChannel,rightChannel]);

    if mod(samplesRead,samplesPerPosition) == 0
        sourcePositionIndex = sourcePositionIndex + 1;
    end
end

```

As a best practice, release your System objects when complete.

```

release(deviceWriter)
release(fileReader)

```

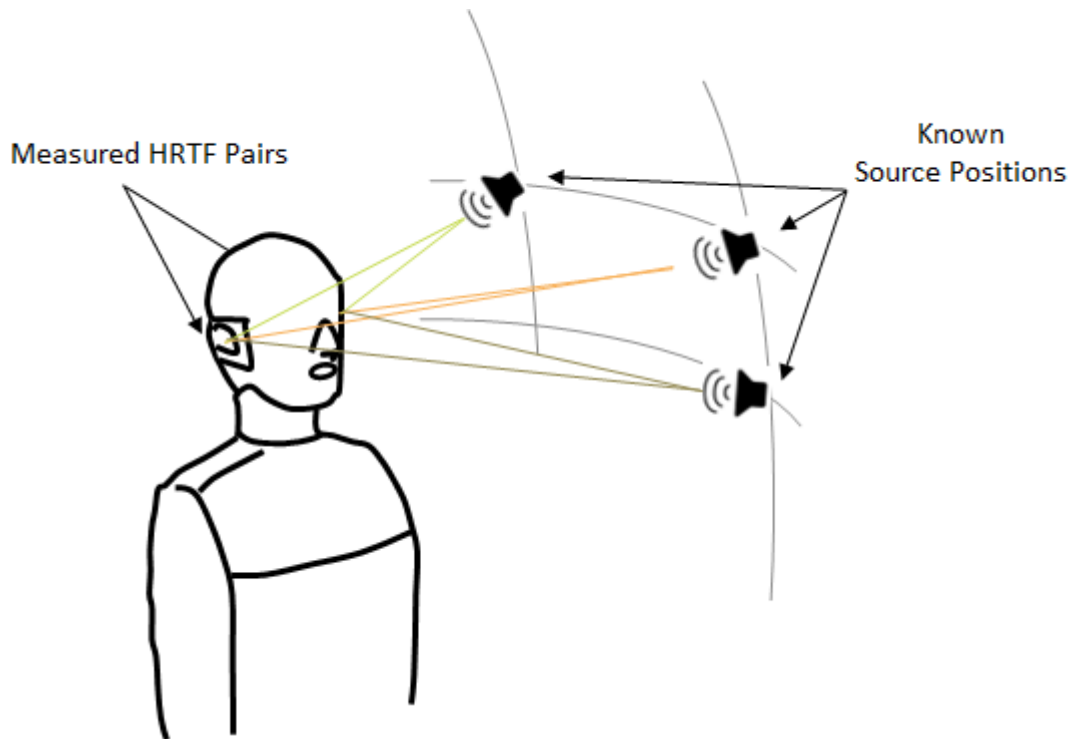
## Input Arguments

### HRTF — HRTF values measured at source positions

*N*-by-2-by-*M* array

HRTF values measured at the source positions, specified as a *N*-by-2-by-*M* array.

- $N$  -- Number of known HRTF pairs
- $M$  -- Number of samples in each known HRTF



If you specify HRTF with real numbers, the function assumes that the input represents an impulse response, and  $M$  corresponds to the length of the impulse response. If you specify HRTF with complex numbers, the function assumes that the input represents a transfer function, and  $M$  corresponds to the number of bins in the frequency response. The output of the `interpolateHRTF` function has the same complexity and interpretation as the input.

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

**sourcePositions** — Source positions corresponding to measured HRTF values  
 $N$ -by-2 matrix



Source positions corresponding to measured HRTF values, specified as a  $N$ -by-2 matrix.  $N$  is the number of known HRTF pairs. The two columns correspond to the azimuth and elevation of the source in degrees, respectively.

Azimuth must be in the range  $[-180,360]$ . You can use the  $-180$  to  $180$  convention or the  $0$  to  $360$  convention.

Elevation must be in the range  $[-90,180]$ . You can use the  $-90$  to  $90$  convention or the  $0$  to  $180$  convention.

Data Types: `single` | `double`

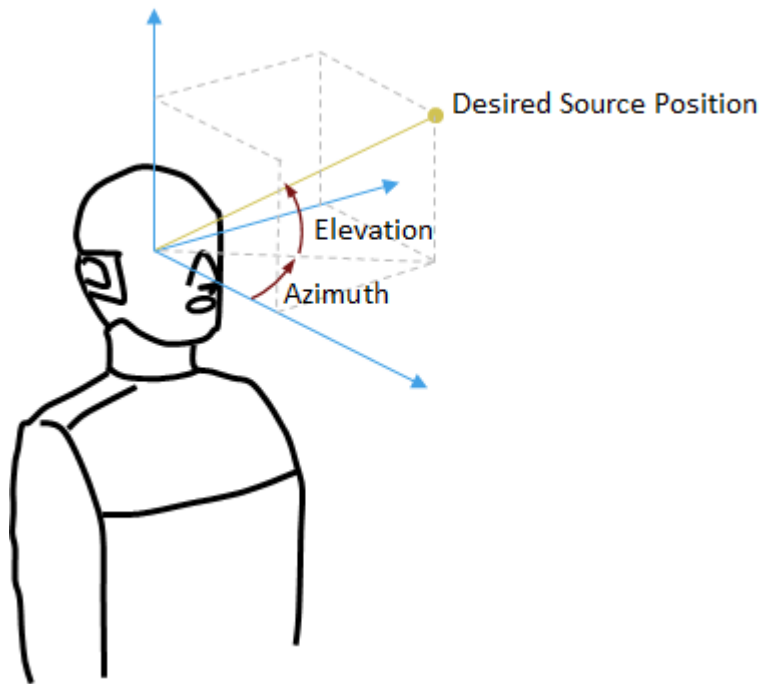
### **desiredSourcePositions** — Desired source positions for HRTF interpolation

$P$ -by-2 matrix

Desired source position for HRTF interpolation, specified as a  $P$ -by-2 matrix.  $P$  is the number of desired source positions. The columns correspond to the desired azimuth and elevation of the source in degrees, respectively.

Azimuth must be in the range  $[-180,360]$ . You can use the  $-180$  to  $180$  convention or the  $0$  to  $360$  convention.

Elevation must be in the range  $[-90,180]$ . You can use the  $-90$  to  $90$  convention or the  $0$  to  $180$  convention.



Data Types: `single` | `double`

### 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Algorithm', 'VBAP'`

#### **Algorithm** – Interpolation algorithm

`'Bilinear'` (default) | `'VBAP'`

Interpolation algorithm, specified as `"Bilinear"` or `"VBAP"`.

- `Bilinear` -- 3-D bilinear interpolation, as specified by [1].

- VBAP -- Vector base amplitude panning interpolation, as specified by [2].

Data Types: char | string

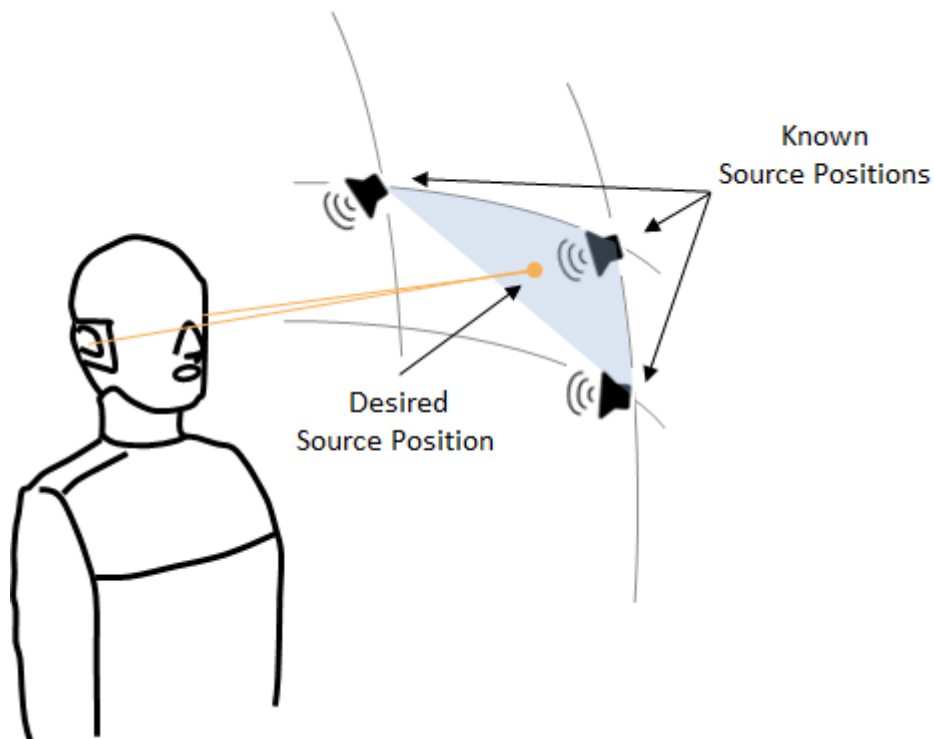
## Output Arguments

### **interpolatedHRTF** — Interpolated HRTF

*P*-by-2-by-*M*

Interpolated HRTF, returned as a *P*-by-2-by-*M* array.

- *P* -- Number of desired source positions, specified by the number of rows in the `desiredSourcePositions` input argument.
- *M* -- Number of samples in each known HRTF, specified by the number of pages in the HRTF input argument.



`interpolatedHRTF` has the same complexity and interpretation as the input. If you specify the input, `HRTF`, with real numbers, the function assumes that the input represents an impulse response. If you specify the input with complex numbers, the function assumes that the input represents a transfer function.

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

### References

- [1] F.P. Freeland, L.W.P. Biscainho and P.S.R. Diniz, "Interpolation of Head-Related Transfer Functions (HRTFS): A multi-source approach." *2004 12th European Signal Processing Conference*. Vienna, 2004, pp. 1761-1764.
- [2] Pulkki, Ville. "Virtual Sound Source Positioning Using Vector Base Amplitude Panning." *Journal of Audio Engineering Society*. Vol. 45. Issue 6, pp. 456-466.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`dsp.FIRFilter` | `dsp.FrequencyDomainFIRFilter`

### External Websites

Acoustics Research Institute HRTF Database

### Introduced in R2018b

# impzest

Estimate impulse response of audio system

## Syntax

```
ir = impzest(excitation, response)
ir = impzest(excitation, response, Name, Value)
```

## Description

`ir = impzest(excitation, response)` returns an estimate of the impulse response (IR) based on the excitation and response.

`ir = impzest(excitation, response, Name, Value)` specifies options using one or more `Name, Value` pair arguments.

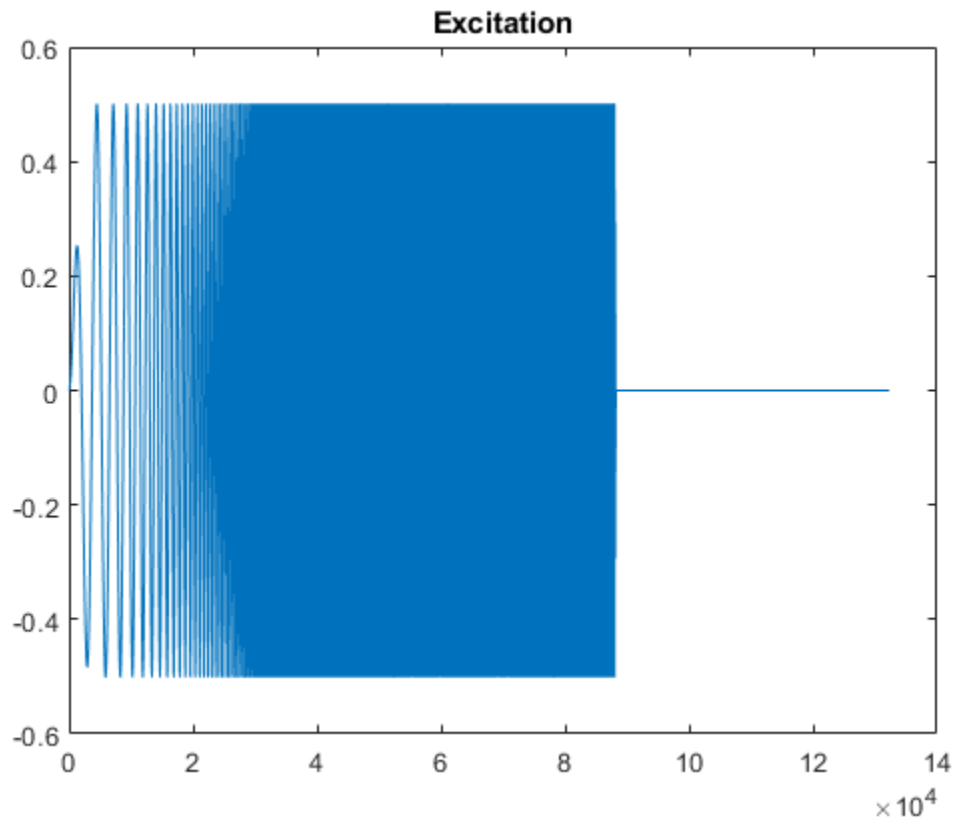
## Examples

### Estimate Impulse Response Using Sweep Tone Excitation

Create a sweep tone excitation signal by using the `sweeptone` function.

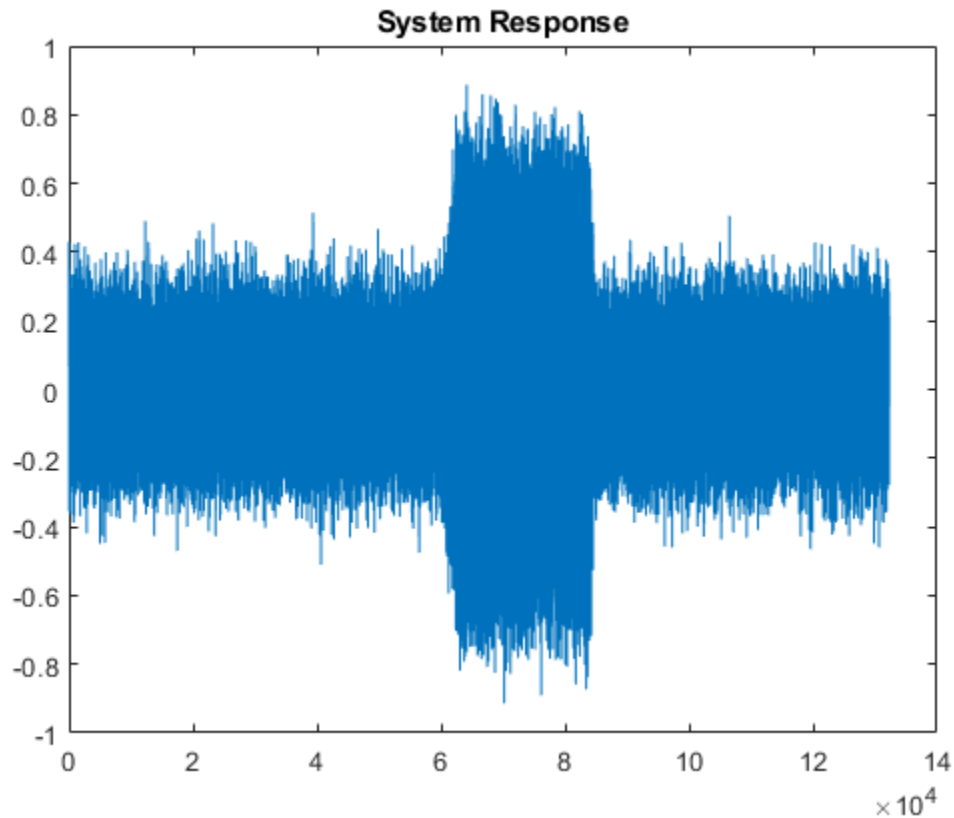
```
excitation = sweeptone(2,1,44100);
```

```
plot(excitation)
title('Excitation')
```



Pass the excitation signal through an infinite impulse response (IIR) filter and add noise to model a real-world recording (system response).

```
[B,A] = butter(10,[.1 .7]);  
rec = filter(B,A,excitation);  
nrec = rec + 0.12*randn(size(rec));  
  
plot(nrec)  
title('System Response')
```

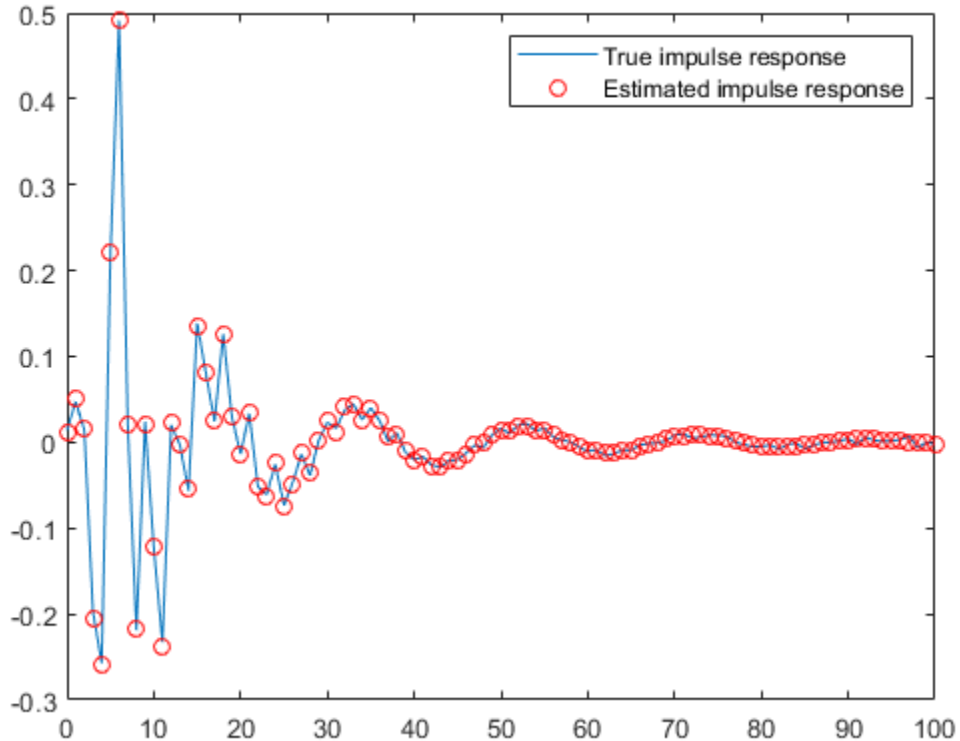


Pass the excitation signal and the system response to the `impzest` function to estimate the impulse response. Truncate the estimate to 100 points. Use `impz` to determine the true impulse response of the system. Plot the true impulse response and the estimated impulse response for comparison.

```
irEstimate = impzest(excitation,nrec);
irEstimate = irEstimate(1:101);
```

```
irTrue = impz(B,A,101);
plot(0:100,irEstimate, ...
     0:100,irTrue,'ro')
```

```
legend('True impulse response','Estimated impulse response')
```



### Estimate Impulse Response Using MLS Excitation

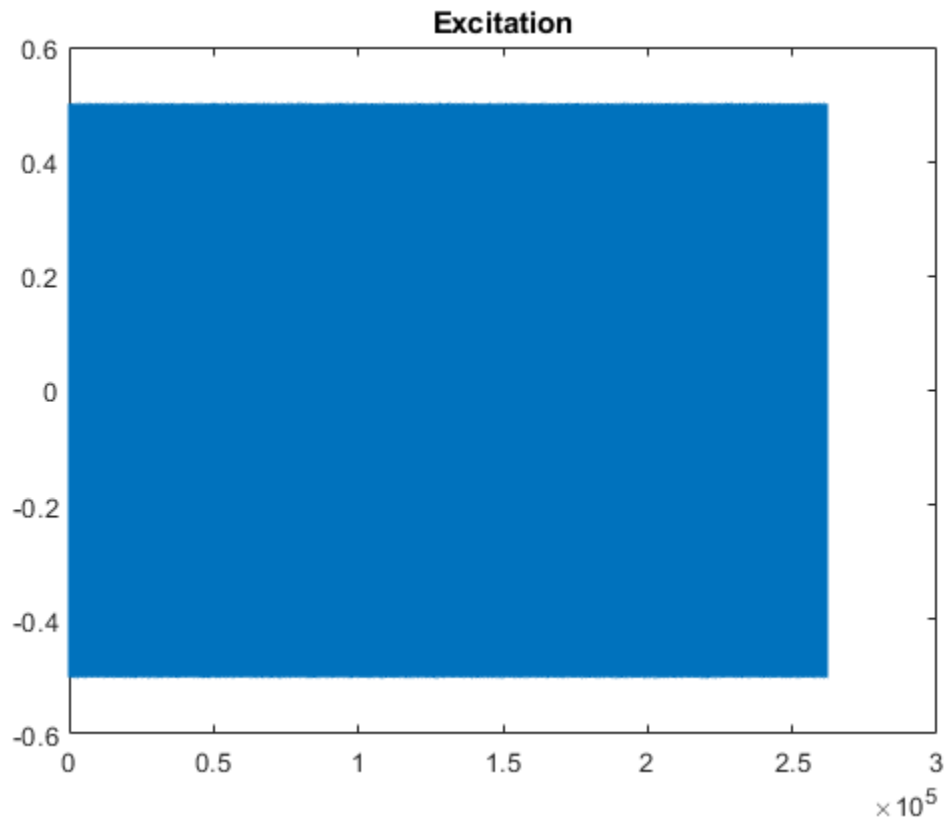
Use `audioread` to read in an impulse response recording. Create a `dsp.FrequencyDomainFIRFilter` object to perform frequency domain filtering using the known impulse response.

```
[irKnown,fs] = audioread('ChurchImpulseResponse-16-44p1-mono-5secs.wav');  
systemModel = dsp.FrequencyDomainFIRFilter(irKnown');
```



Create an MLS excitation signal by using the `mls` function. The MLS excitation signal must be longer than the impulse response. Note that the length of the MLS excitation is extended to the next power of two minus one.

```
excitation = mls(numel(irKnown)+1);  
  
plot(excitation)  
title('Excitation')
```

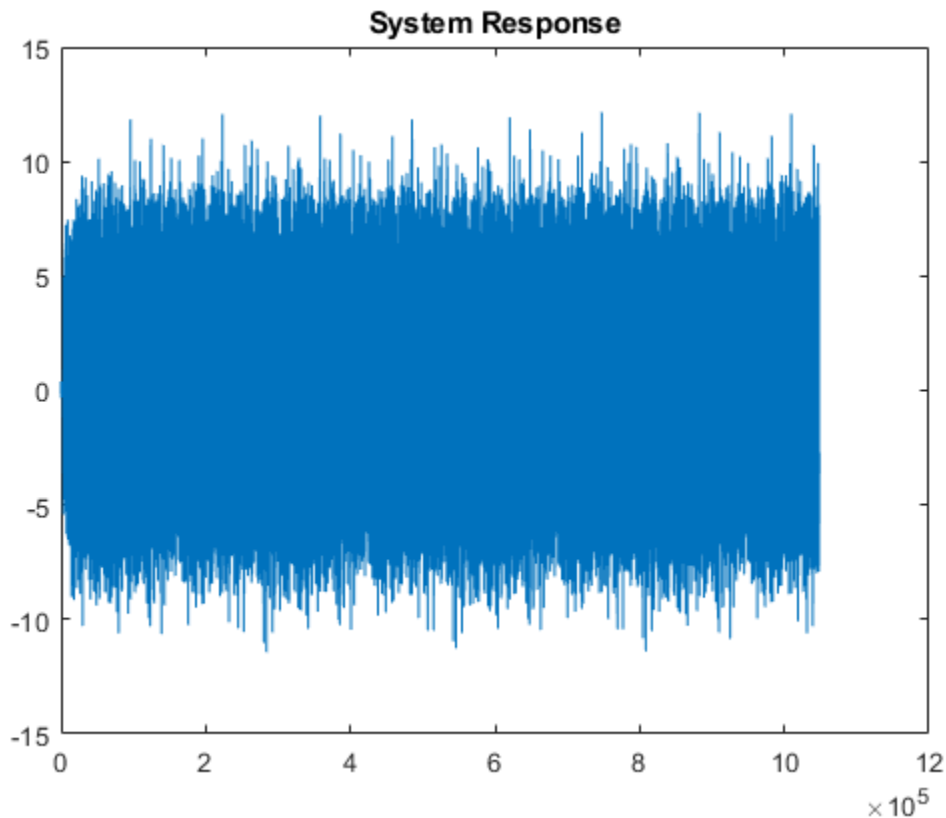


Replicate the excitation signal four times to measure the average of three measurements. The recording of the first MLS sequence does include all the impulse response information, so `impzest` discards it as a warmup run. Pad the excitation signal with zeros to account for the filter latency.

```
numRuns = 4;  
excrep = repmat(excitation,numRuns,1);  
excrep = [excrep;zeros(numel(irKnown)+1,1)];
```

Pass the excitation signal through the known filter and then add noise to model a real-world recording (system response). Cut the delay introduced at the beginning by the filter.

```
rec = systemModel(excrep);  
rec = rec + 0.1*randn(size(rec));  
  
rec = rec(numel(irKnown)+2:end,:);  
  
plot(rec)  
title('System Response')
```



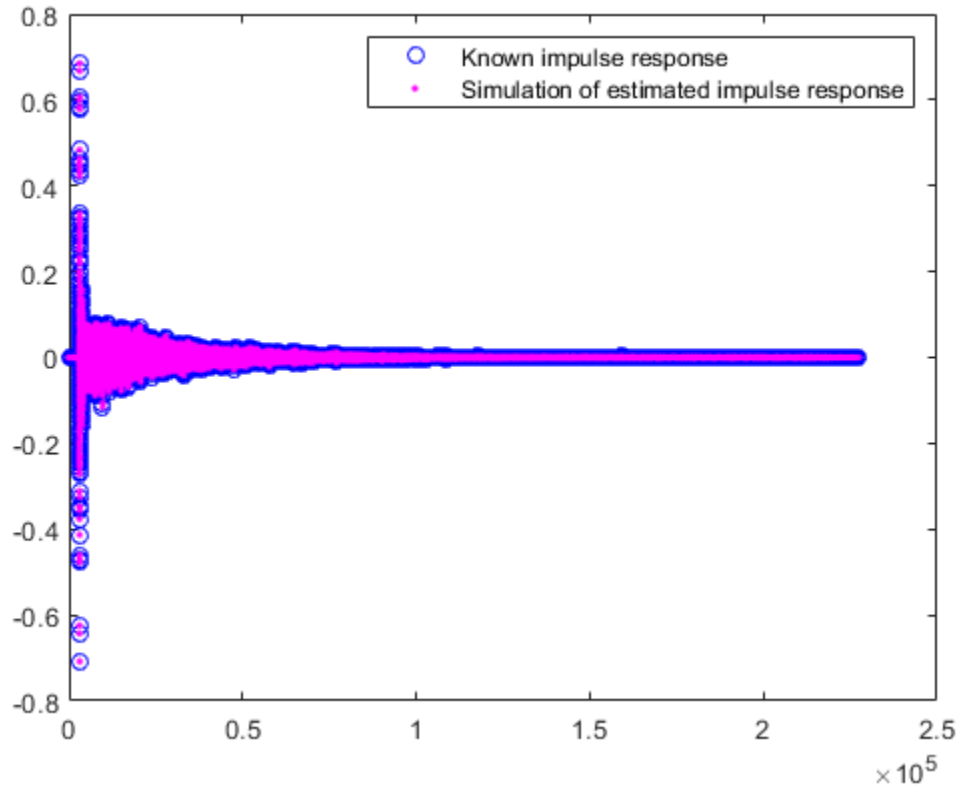
In a real-world scenario, the MLS sequence is played back in the system under test while recording. The recording would be cut so that it begins at the moment the MLS sequence is picked-up and truncated to last the duration of the repeated sequence.

Pass the excitation signal and the system response to the `impzest` function to estimate the impulse response. Plot the known impulse response and the simulation of the estimated impulse response for comparison.

```
irEstimate = impzest(excitation,rec);

samples = 1:numel(irKnown);
plot(samples,irEstimate(samples),'bo', ...
      samples,irKnown(samples),'m.')

legend('Known impulse response','Simulation of estimated impulse response')
```



## Input Arguments

**excitation** — Single period of excitation signal input to audio system  
column vector

Single period of excitation signal input to audio system, specified as a column vector.

You can generate excitation signals by using `m`ls (maximum length sequence) or `sweptone` (exponential sine sweep).

Data Types: `single` | `double`

**response — Recorded signal output from audio system**

column vector | matrix

Recorded signal output from audio system, specified as a column vector or matrix. If specified as a matrix, each column of the matrix is treated as an independent channel.

Data Types: single | double

**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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'WarmupRuns',2

**WarmupRuns — Number of warmup runs in response**

nonnegative integer

Number of warmup runs in the response, specified as a nonnegative integer. The `impzest` function estimates the impulse response after discarding the specified number of warmup runs from the response.

The default number of warmup runs depends on whether the excitation signal was generated using the `m1s` or `sweeptone` function:

- `m1s` -- 1
- `sweeptone` -- 0

Data Types: single | double

**Output Arguments****ir — Estimate of the impulse response of an audio system**

column vector | matrix

Estimate of the impulse response of an audio system, returned as a column vector or matrix. The size of `ir` is  $L$ -by- $C$ , where:

- *L* -- MLS length or duration of sweep tone silence
- *C* -- Number of columns (channels) in the response signal

Data Types: `single` | `double`

### References

- [1] Farino, Angelo. "Advancements in Impulse Response Measurements by Sine Sweeps." Presented at the *Audio Engineering Society 122nd Convention*, Vienna, Austria, 2007.
- [2] Guy-Bart, Stan, Jean-Jacques Embrachts, and Dominique Archambeau. "Comparison of Different Impulse Response Measurement Techniques." *Journal of Audio Engineering Society*. Vol. 50, Issue 4, 2002, pp. 246-262.
- [3] Armelloni, Enrico, Christian Giottoli, and Angelo Farina. "Implementation of Real-Time Partitioned Convolution on a DSP Board." *Application of Signal Processing to Audio and Acoustics, 2003 IEEE Workshop*, pp. 71-74. IEEE, 2003.

### See Also

**Impulse Response Measurer** | `mls` | `sweeptone`

**Introduced in R2018b**

# mididevinfo

MIDI device information

## Syntax

```
mididevinfo  
deviceInformation = mididevinfo
```

## Description

`mididevinfo` displays a table containing information about the MIDI devices attached to the system.

`deviceInformation = mididevinfo` returns a structure, `deviceInformation`, containing information about the MIDI devices attached to the system.

---

**Note** Before starting MATLAB, connect your MIDI device to your computer and turn on the device. For connection instructions, see the instructions for your MIDI device. If you start MATLAB before connecting your device, MATLAB might not recognize your device when you connect it. To correct the problem, restart MATLAB with the device already connected.

---

## Examples

### Display MIDI Device Connections

Call `mididevinfo` to display a table containing information about the MIDI devices attached to your system.

```
mididevinfo  
  
MIDI devices available:  
ID Direction Interface Name
```

```
0 output MMSystem 'Microsoft MIDI Mapper'  
1 input MMSystem 'BCF2000'  
2 input MMSystem 'MIDIIN2 (BCF2000)'  
3 output MMSystem 'Microsoft GS Wavetable Synth'  
4 output MMSystem 'BCF2000'  
5 output MMSystem 'MIDIOUT2 (BCF2000)'  
6 output MMSystem 'MIDIOUT3 (BCF2000)'
```

### Return Structure of MIDI Device Connections

Call `mididevinfo` with an output argument to return a structure containing MIDI device information.

```
deviceInformation = mididevinfo  
  
deviceInformation = struct with fields:  
    input: [0×0 struct]  
    output: [1×2 struct]
```

The `deviceInformation` structure has two fields: `input` and `output`. Both `input` and `output` contain arrays of structures. Each member has three fields: `Name`, `Interface`, and `ID`. Get the device information for the output Microsoft GS Wavetable Synth device.

```
deviceInformation.output(2)  
  
ans = struct with fields:  
    Name: 'Microsoft GS Wavetable Synth'  
    Interface: 'MMSystem'  
    ID: 1
```

## Output Arguments

### `deviceInformation` — Description of available devices

struct

Description of available devices, returned as nested structures. The outer structure has two fields: `input` and `output`. The `input` and `output` values are arrays of structures, and each member has three fields: `Name`, `Interface`, and `ID`.



Data Types: struct

## **See Also**

mididevice | midimsg | midireceive | midisend

## **Topics**

“MIDI Device Interface”

## **External Websites**

MIDI Manufacturers Association

**Introduced in R2018a**

## pitch

Estimate fundamental frequency of audio signal

### Syntax

```
f0 = pitch(audioIn,fs)
f0 = pitch(audioIn,fs,Name,Value)
[f0,loc] = pitch( ___ )
```

### Description

`f0 = pitch(audioIn,fs)` returns estimates of the fundamental frequency over time for the audio input, `audioIn`, with sample rate `fs`. Columns of the input are treated as individual channels.

`f0 = pitch(audioIn,fs,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

`[f0,loc] = pitch( ___ )` returns the locations, `loc`, associated with fundamental frequency estimates.

### Examples

#### Estimate Pitch of Speech Signal Using Default Parameters

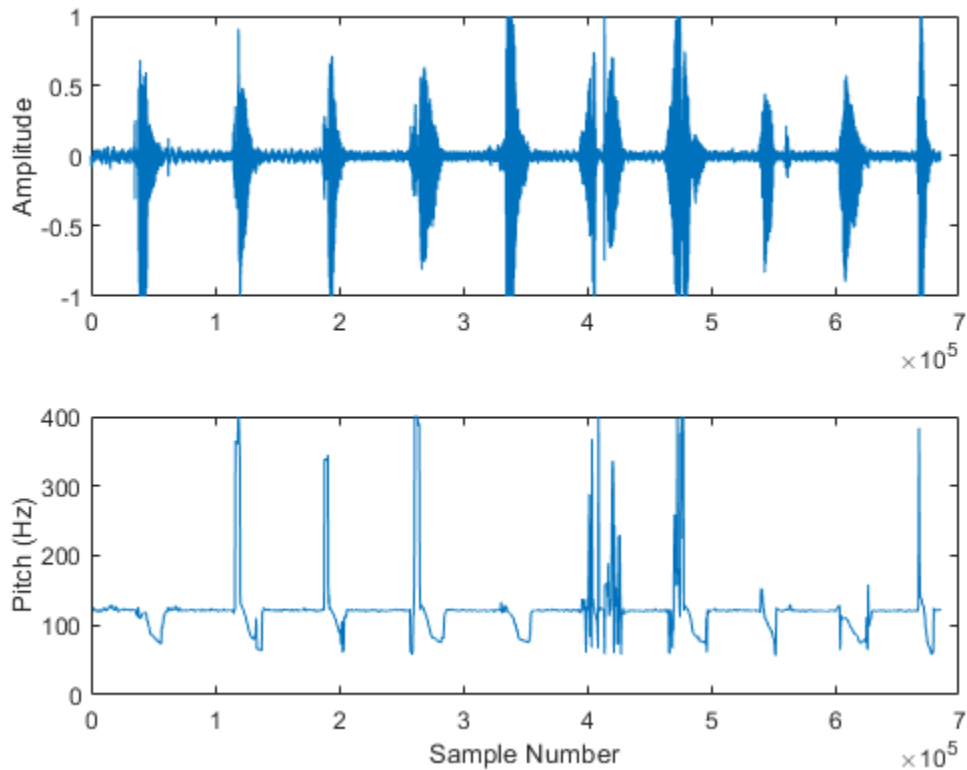
Read in an audio file and then call the `pitch` function with default parameters.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
[f0,idx] = pitch(audioIn,fs);
```

Plot the audio signal and pitch contour.

```
subplot(2,1,1)
plot(audioIn)
```

```
ylabel('Amplitude')  
  
subplot(2,1,2)  
plot(idx,f0)  
ylabel('Pitch (Hz)')  
xlabel('Sample Number')
```



The `pitch` function estimates the fundamental frequency of the input signal at locations determined by the `WindowLength` and `OverlapLength` name-value pairs.

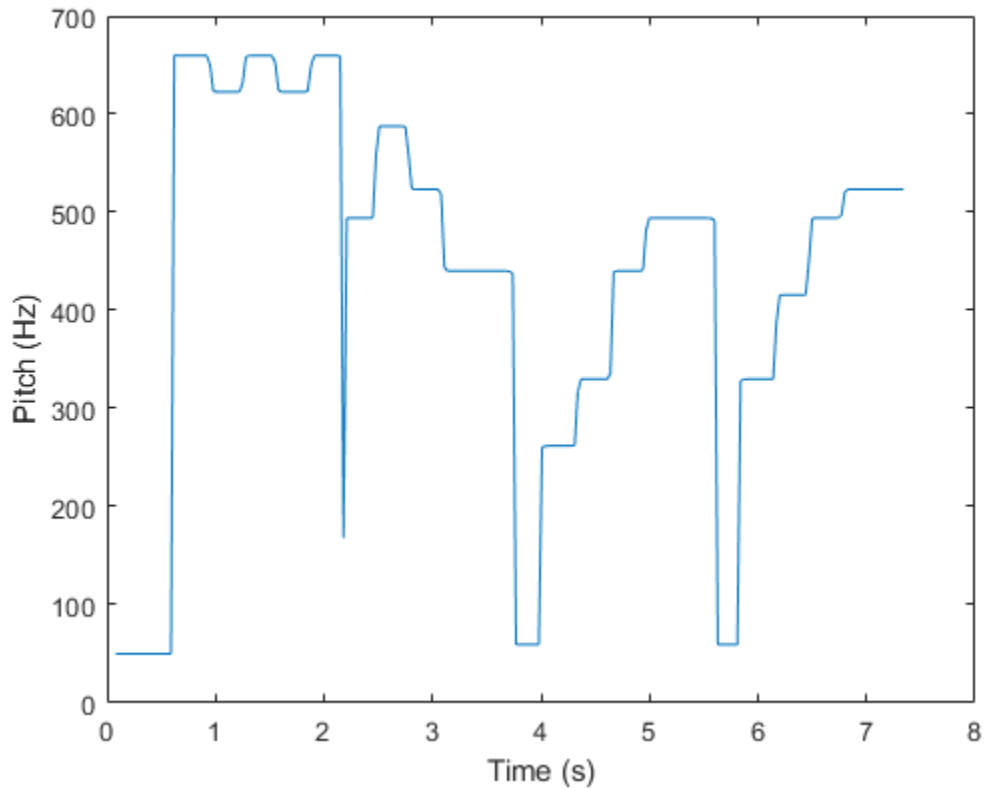
### Estimate Pitch of Musical Signal Using Nondefault Parameters

Load an audio file of the introduction to Für Elise and the sample rate of the audio. Call the `pitch` function using the pitch estimate filter (PEF), a search range from 50 Hz to 800 Hz, a window length of 80 ms, and an overlap of 50 ms. Plot the results and listen to the song to verify the fundamental frequency estimates returned by the `pitch` function.

```
load FurElise.mat song fs

[f0,loc] = pitch(song,fs, ...
    'Method','PEF', ...
    'Range',[50 800], ...
    'WindowLength',round(fs*0.08), ...
    'OverlapLength',round(fs*0.05));

t = loc/fs;
plot(t,f0)
ylabel('Pitch (Hz)')
xlabel('Time (s)')
```



```
sound(song, fs)
```

### Compare Pitch Detection Algorithms

The different methods of estimating pitch provide trade-offs in terms of noise robustness, accuracy, optimal lag, and computation expense. In this example, you compare the performance of different pitch detection algorithms in terms of gross pitch error (GPE) and computation time under different noise conditions.

### Prepare Test Signals

Load an audio file and determine the number of samples it has. Also load the true pitch corresponding to the audio file. The true pitch was determined as an average of several third-party algorithms on the clean speech file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
numSamples = size(audioIn,1);  
load TruePitch.mat truePitch
```

Create test signals by adding noise to the audio signal at given SNRs. The `mixSNR` function is a convenience function local to this example, which takes a signal, noise, and requested SNR and returns a noisy signal at the request SNR.

```
testSignals = zeros(numSamples,4);  
  
turbine = audioread('Turbine-16-44p1-mono-22secs.wav');  
testSignals(:,1) = mixSNR(audioIn,turbine,20);  
testSignals(:,2) = mixSNR(audioIn,turbine,0);  
  
whiteNoiseMaker = dsp.ColoredNoise('Color','white','SamplesPerFrame',size(audioIn,1));  
testSignals(:,3) = mixSNR(audioIn,whiteNoiseMaker(),20);  
testSignals(:,4) = mixSNR(audioIn,whiteNoiseMaker(),0);
```

Save the noise conditions and algorithm names as cell arrays for labeling and indexing.

```
noiseConditions = {'Turbine (20 dB)','Turbine (0 dB)','WhiteNoise (20 dB)','WhiteNoise'};  
algorithms = {'NCF','PEF','CEP','LHS','SRH'};
```

### Run Pitch Detection Algorithms

Preallocate arrays to hold pitch decisions for each algorithm and noise condition pair, and the timing information. In a loop, call the `pitch` function on each combination of algorithm and noise condition. Each algorithm has an optimal window length associated with it. In this example, for simplicity, you use the default window length for all algorithms. Use a 3-element median filter to smooth the pitch decisions.

```
f0 = zeros(numel(truePitch),numel(algorithms),numel(noiseConditions));  
algorithmTimer = zeros(numel(noiseConditions),numel(algorithms));  
  
for k = 1:numel(noiseConditions)  
    x = testSignals(:,k);  
    for i = 1:numel(algorithms)  
        tic
```

```

        f0temp = pitch(x,fs, ...
            'Range',[50 300], ...
            'Method',algorithms{i}, ...
            'MedianFilterLength',3);
        algorithmTimer(k,i) = toc;
        f0(1:max(numel(f0temp),numel(truePitch)),i,k) = f0temp;
    end
end

```

### Compare Gross Pitch Error

Gross pitch error (GPE) is a popular metric when comparing pitch detection algorithms. GPE is defined as the proportion of pitch decisions for which the relative error is higher than a given threshold, traditionally 20% in speech studies. Calculate the GPE and print it to the Command Window.

```

idxToCompare = ~isnan(truePitch);
truePitch = truePitch(idxToCompare);
f0 = f0(idxToCompare,:,:);

p = 0.20;
GPE = mean( abs(f0(1:numel(truePitch),:,:)) - truePitch) > truePitch.*p).*100;

for ik = 1:numel(noiseConditions)
    fprintf('\nGPE (p = %0.2f), Noise = %s.\n',p,noiseConditions{ik});
    for i = 1:size(GPE,2)
        fprintf('- %s : %0.1f %%\n',algorithms{i},GPE(1,i,ik))
    end
end

```

GPE (p = 0.20), Noise = Turbine (20 dB).

```

- NCF : 0.9 %
- PEF : 0.4 %
- CEP : 8.2 %
- LHS : 8.2 %
- SRH : 6.0 %

```

GPE (p = 0.20), Noise = Turbine (0 dB).

```

- NCF : 5.6 %
- PEF : 24.5 %
- CEP : 11.6 %
- LHS : 9.4 %
- SRH : 46.8 %

```

GPE (p = 0.20), Noise = WhiteNoise (20 dB).

```
- NCF : 0.9 %  
- PEF : 0.0 %  
- CEP : 12.9 %  
- LHS : 6.9 %  
- SRH : 2.6 %
```

```
GPE (p = 0.20), Noise = WhiteNoise (0 dB).
```

```
- NCF : 0.4 %  
- PEF : 0.0 %  
- CEP : 23.6 %  
- LHS : 7.3 %  
- SRH : 1.7 %
```

Calculate the average time it takes to process one second of data for each of the algorithms and print the results.

```
aT = sum(algorithmTimer)./((numSamples/fs)*numel(noiseConditions));  
for ik = 1:numel(algorithms)  
    fprintf('- %s : %0.3f (s)\n',algorithms{ik},aT(ik))  
end
```

```
- NCF : 0.058 (s)  
- PEF : 0.289 (s)  
- CEP : 0.066 (s)  
- LHS : 0.182 (s)  
- SRH : 0.212 (s)
```

### Determine Pitch Contour using `pitch` and `voiceActivityDetector`

Read in an entire speech file and determine the fundamental frequency of the audio using the `pitch` function. Then use the `voiceActivityDetector` to remove irrelevant pitch information that does not correspond to the speaker.

Read in the audio file and associated sample rate.

```
[audio,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Specify pitch detection using a 50 ms window length and 40 ms overlap (10 ms hop). Specify that the `pitch` function searches for the fundamental frequency over the range 50-150 Hz and postprocesses the results with a median filter. Plot the results.

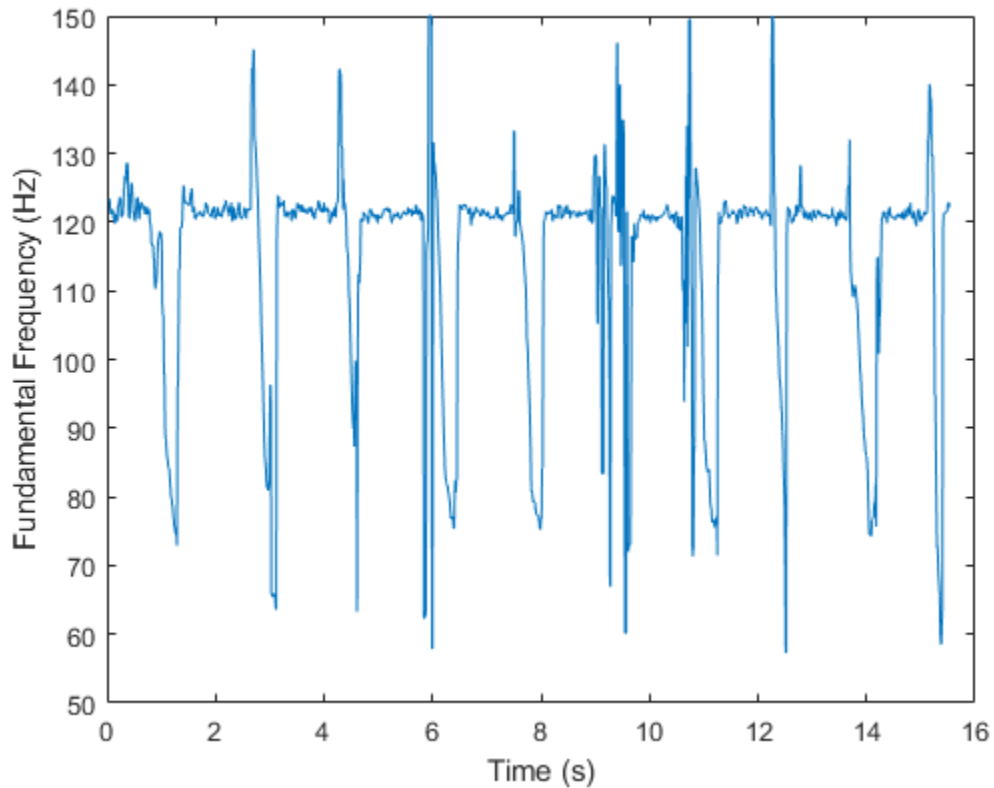
```
windowLength = round(0.05*fs);  
overlapLength = round(0.04*fs);
```



```
hopLength = windowLength - overlapLength;
```

```
[f0,loc] = pitch(audio,fs, ...  
    'WindowLength',windowLength, ...  
    'OverlapLength',overlapLength, ...  
    'Range',[50 150], ...  
    'MedianFilterLength',3);
```

```
plot(loc/fs,f0)  
ylabel('Fundamental Frequency (Hz)')  
xlabel('Time (s)')
```



Create a `dsp.AsyncBuffer` System object™ to chunk the audio signal into overlapped frames. Also create a `voiceActivityDetector` System object™ to determine if the frames contain speech.

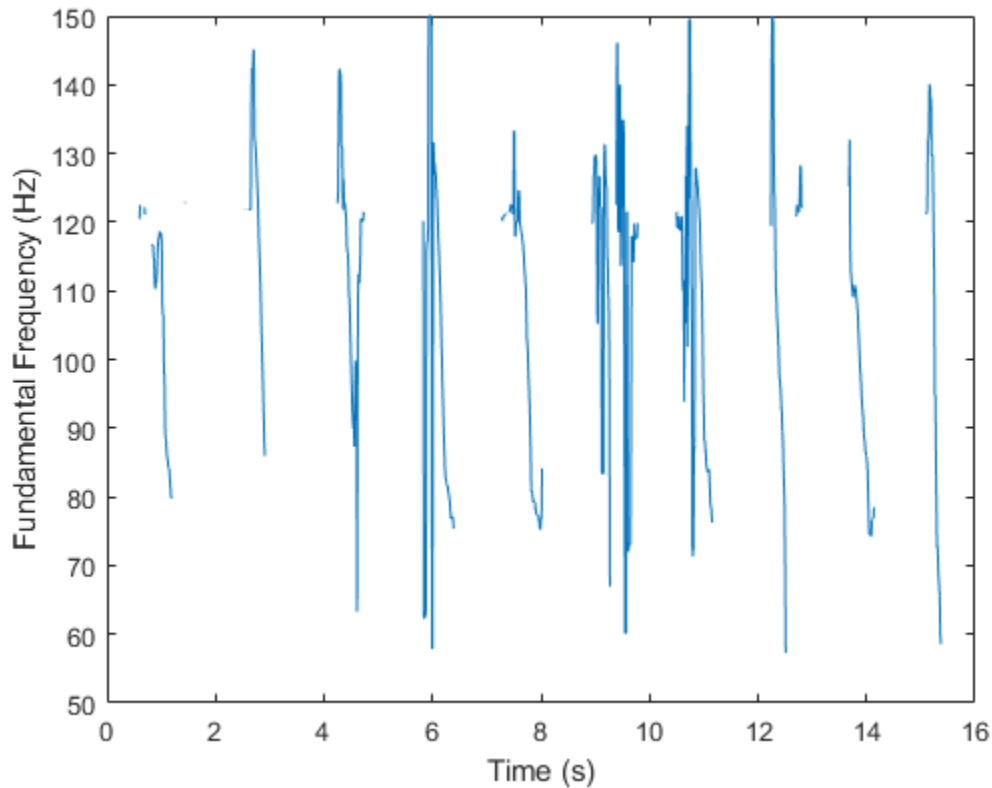
```
buffer = dsp.AsyncBuffer(numel(audio));  
write(buffer, audio);  
VAD = voiceActivityDetector;
```

While there are enough samples to `hop`, read from the buffer and determine the probability that the frame contains speech. To mimic the decision spacing in time of the pitch function, the first frame read from the buffer has no overlap.

```
n = 1;  
probabilityVector = zeros(numel(loc),1);  
while buffer.NumUnreadSamples >= hopLength  
    if n==1  
        x = read(buffer,windowLength);  
    else  
        x = read(buffer,windowLength,overlapLength);  
    end  
    probabilityVector(n) = VAD(x);  
    n = n+1;  
end
```

Use the probability vector determined by the `voiceActivityDetector` to plot a pitch contour for the speech file that corresponds to regions of speech.

```
validIdx = probabilityVector>0.99;  
loc(~validIdx) = nan;  
f0(~validIdx) = nan;  
plot(loc/fs,f0)  
ylabel('Fundamental Frequency (Hz)')  
xlabel('Time (s)')
```



## Input Arguments

### **audioIn** — Audio input signal

vector | matrix

Audio input signal, specified as a vector or matrix. The columns of the matrix are treated as individual audio channels.

Data Types: `single` | `double`

### **fs** — Sample rate (Hz)

positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

The sample rate must be greater than or equal to twice the upper bound of the search range. Specify the search range using the `Range` name-value pair.

Data Types: `single` | `double`

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `pitch(audioIn, fs, 'Range', [50, 150], 'Method', 'PEF')`

### Range — Search range for pitch estimates

`[50, 400]` (default) | two-element row vector with increasing positive integer values

Search range for pitch estimates, specified as the comma-separated pair consisting of 'Range' and a two-element row vector with increasing positive integer values. The function searches for a best estimate of the fundamental frequency within the upper and lower band edges specified by the vector, according to the algorithm specified by `Method`. The range is inclusive and units are in Hz.

Valid values for the search range depend on the sample rate, `fs`, and on the values of `WindowLength` and `Method`:

Method	Minimum Range	Maximum Range
'NCF'	$fs/WindowLength < Range(1)$	$Range(2) < fs/2$
'PEF'	$10 < Range(1)$	$Range(2) < \min(4000, fs/2)$
'CEP'	$fs / (2^{\text{nextpow2}(2*WindowLength-1)} < Range(1)$	$Range(2) < fs/2$
'LHS'	$1 < Range(1)$	$Range(2) < fs/5 - 1$
'SRH'	$1 < Range(1)$	$Range(2) < fs/5 - 1$

Data Types: `single` | `double`

**WindowLength — Number of samples in analysis window**`round(fs*0.052) (default) | integer`

Number of samples in the analysis window, specified as the comma-separated pair consisting of 'WindowLength' and an integer in the range [1, min(size(audioIn,1), 192000)]. Typical analysis windows are in the range 20–100 ms. The default window length is 52 ms.

Data Types: `single` | `double`

**OverlapLength — Number of samples of overlap between adjacent analysis windows**`round(fs*0.042) (default) | integer`

Number of samples of overlap between adjacent analysis windows, specified as the comma-separated pair consisting of 'OverlapLength' and an integer in the range (-inf,WindowLength). A negative overlap length indicates non-overlapping analysis windows.

Data Types: `single` | `double`

**Method — Method used to estimate pitch**`'NCF' (default) | 'PEF' | 'CEP' | 'LHS' | 'SRH'`

Method used to estimate pitch, specified as the comma-separated pair consisting of 'Method' and 'NCF', 'PEF', 'CEP', 'LHS', or 'SRH'. The different methods of calculating pitch provide trade-offs in terms of noise robustness, accuracy, and computation expense. The algorithms used to calculate pitch are based on the following papers:

- 'NCF' -- Normalized Correlation Function [1]
- 'PEF' -- Pitch Estimation Filter [2]. The function does not use the amplitude compression described by the paper.
- 'CEP' -- Cepstrum Pitch Determination [3]
- 'LHS' -- Log-Harmonic Summation [4]
- 'SRH' -- Summation of Residual Harmonics [5]

Data Types: `char` | `string`

**MedianFilterLength — Median filter length used to smooth pitch estimates over time**`1 (default) | positive integer`

Median filter length used to smooth pitch estimates over time, specified as the comma-separated pair consisting of 'MedianFilterLength' and a positive integer. The default, 1, corresponds to no median filtering. Median filtering is a postprocessing technique used to remove outliers while estimating pitch. The function uses `movmedian` after estimating the pitch using the specified `Method`.

Data Types: `single` | `double`

## Output Arguments

### **f0** — Estimated fundamental frequency (Hz)

`scalar` | `vector` | `matrix`

Estimated fundamental frequency, in Hz, returned as a scalar, vector, or matrix. The number of rows returned depends on the values of the `WindowLength` and `OverlapLength` name-value pairs, and on the input signal size. The number of columns (channels) returned depends on the number of columns of the input signal size.

Data Types: `single` | `double`

### **loc** — Locations associated with fundamental frequency estimations

`scalar` | `vector` | `matrix`

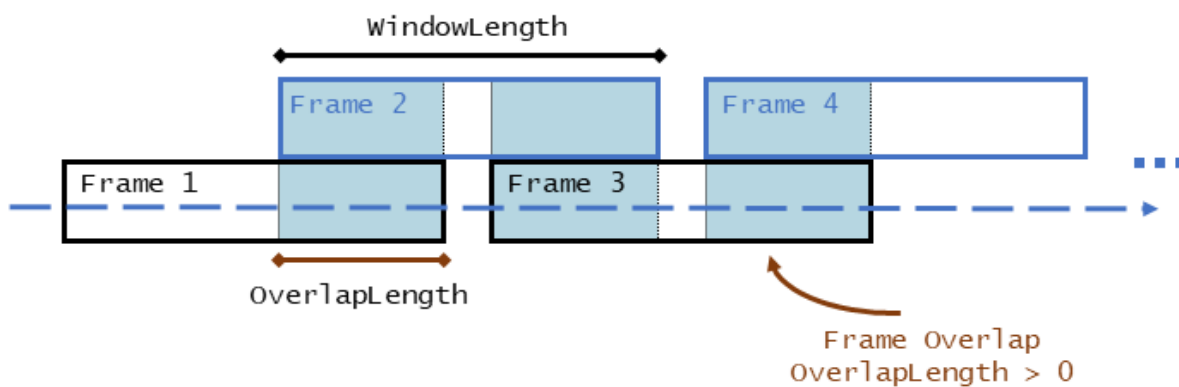
Locations associated with fundamental frequency estimations, returned as a scalar, vector, or matrix the same size as `f0`.

Fundamental frequency is estimated locally over a region of `WindowLength` samples. The values of `loc` correspond to the most recent sample (largest sample number) used to estimate fundamental frequency.

Data Types: `single` | `double`

## Algorithms

The `pitch` function segments the audio input according to the `WindowLength` and `OverlapLength` arguments. The fundamental frequency is estimated for each frame. The locations output, `loc` contains the most recent samples (largest sample numbers) of the corresponding frame.



For a description of the algorithms used to estimate the fundamental frequency, consult the corresponding references:

- 'NCF' -- Normalized Correlation Function [1]
- 'PEF' -- Pitch Estimation Filter [2]. The function does not use the amplitude compression described by the paper.
- 'CEP' -- Cepstrum Pitch Determination [3]
- 'LHS' -- Log-Harmonic Summation [4]
- 'SRH' -- Summation of Residual Harmonics [5]

## References

- [1] Atal, B.S. "Automatic Speaker Recognition Based on Pitch Contours." *The Journal of the Acoustical Society of America*. Vol. 52, No. 6B, 1972, pp. 1687-1697.
- [2] Gonzalez, Sira, and Mike Brookes. "A Pitch Estimation Filter robust to high levels of noise (PEFAC)." 19th European Signal Processing Conference. Barcelona, 2011, pp. 451-455.
- [3] Noll, Michael A. "Cepstrum Pitch Determination." *The Journal of the Acoustical Society of America*. Vol. 31, No. 2, 1967, pp. 293-309.
- [4] Hermes, Dik J. "Measurement of Pitch by Subharmonic Summation." *The Journal of the Acoustical Society of America*. Vol. 83, No. 1, 1988, pp. 257-264.

- [5] Drugman, Thomas, and Abeer Alwan. "Joint Robust Voicing Detection and Pitch Estimation Based on Residual Harmonics." Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH. 2011, pp. 1973-1976.

## **Extended Capabilities**

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

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

### **See Also**

`audioFeatureExtractor` | `harmonicRatio` | `mfcc` | `voiceActivityDetector`

### **Topics**

"Speaker Identification Using Pitch and MFCC"

**Introduced in R2018a**



# mfcc

Extract mfcc, log energy, delta, and delta-delta of audio signal

## Syntax

```
coeffs = mfcc(audioIn,fs)
coeffs = mfcc( ____,Name,Value)
[coeffs,delta,deltaDelta,loc] = mfcc( ____ )
```

## Description

`coeffs = mfcc(audioIn,fs)` returns the mel frequency cepstral coefficients (MFCCs) for the audio input, sampled at a frequency of `fs` Hz.

`coeffs = mfcc( ____,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values..

Example: `[coeffs] = mfcc(audioIn,fs,'LogEnergy','Replace')` returns mel frequency cepstral coefficients for the audio input signal sampled at `fs` Hz. The first coefficient in the `coeffs` vector is replaced with the log energy value.

`[coeffs,delta,deltaDelta,loc] = mfcc( ____ )` returns the delta, delta-delta, and location of samples corresponding to each window of data.

## Examples

### Compute Mel Frequency Cepstral Coefficients

Compute the mel frequency cepstral coefficients of a speech signal using the `mfcc` function. The function returns `delta`, the change in coefficients, and `deltaDelta`, the change in delta values. The log energy value that the function computes can prepend the coefficients vector or replace the first element of the coefficients vector. This is done based on whether you set the `'LogEnergy'` argument to `'Append'` or `'Replace'`.

Read an audio signal from the 'Counting-16-44p1-mono-15secs.wav' file using the `audioread` function. The `mfcc` function processes the entire speech data in a batch. The default `DeltaWindowLength` is 2. Therefore, `delta` is computed as the difference between the current coefficients and the previous coefficients. `deltaDelta` is computed as the difference between the current and the previous `delta` values. Based on the number of input rows, the window length, and the hop length, `mfcc` partitions the speech into 1551 frames and computes the cepstral features for each frame. Each row in the `coeffs` matrix corresponds to the log-energy value followed by the 13 mel-frequency cepstral coefficients for the corresponding frame of the speech file. The function also computes `loc`, the location of the last sample in each input frame.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
[coeffs,delta,deltaDelta,loc] = mfcc(audioIn,fs);
```

### Extract MFCC from Frequency-Domain Audio

Read in an audio file and convert it to a frequency representation.

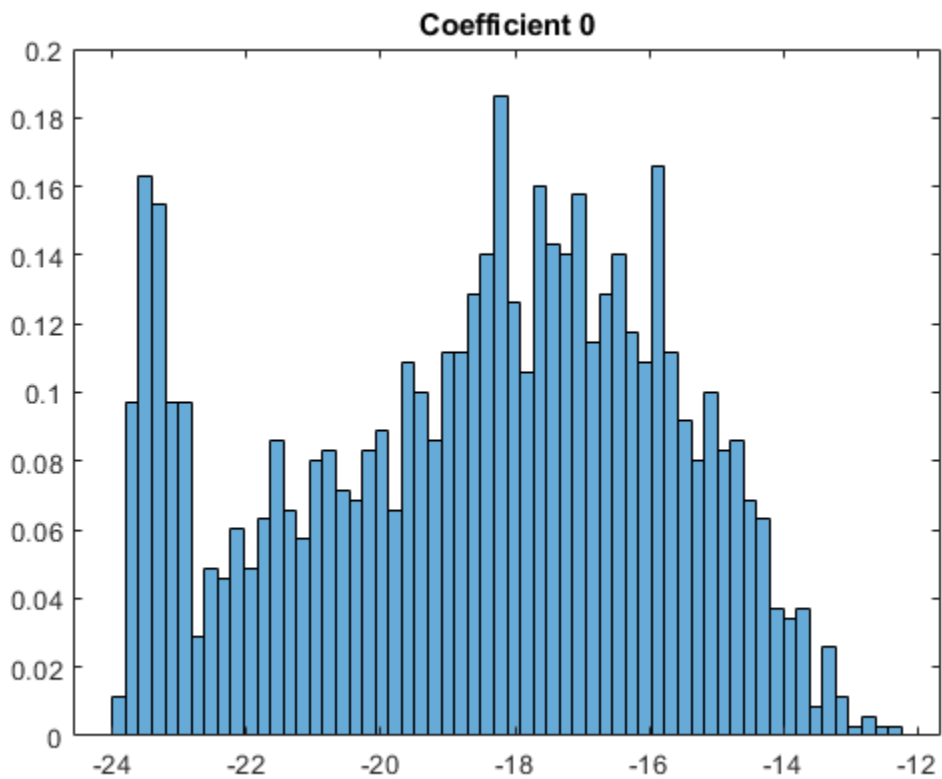
```
[audioIn,fs] = audioread("Rainbow-16-8-mono-114secs.wav");  
  
win = hann(1024,"periodic");  
S = stft(audioIn,"Window",win,"OverlapLength",512,"Centered",false);
```

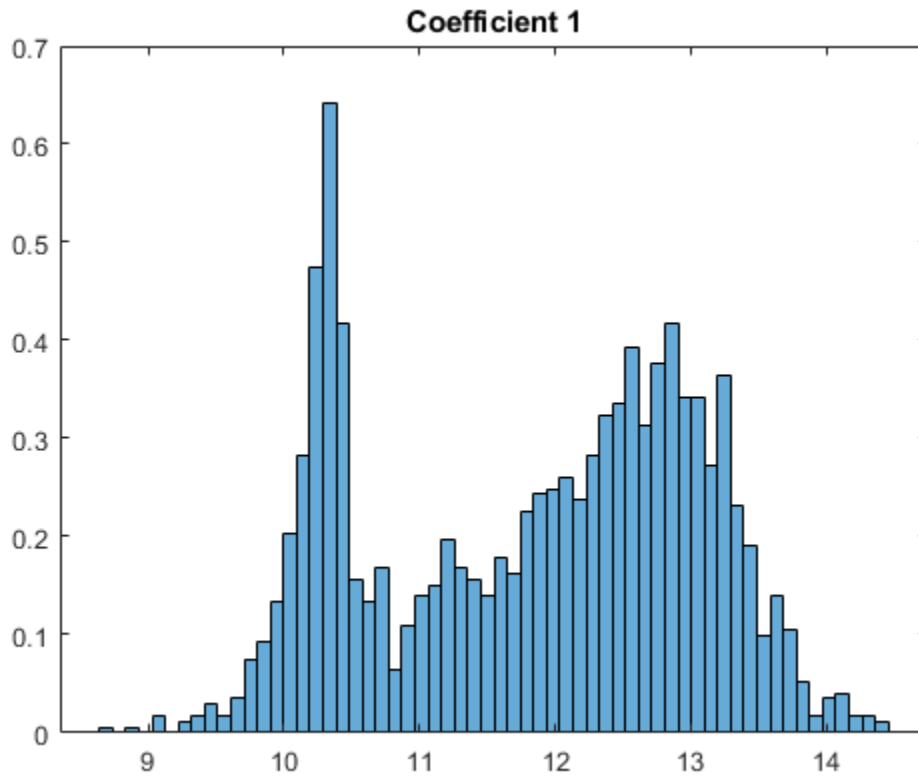
To extract the mel-frequency cepstral coefficients, call `mfcc` with the frequency-domain audio. Ignore the log-energy.

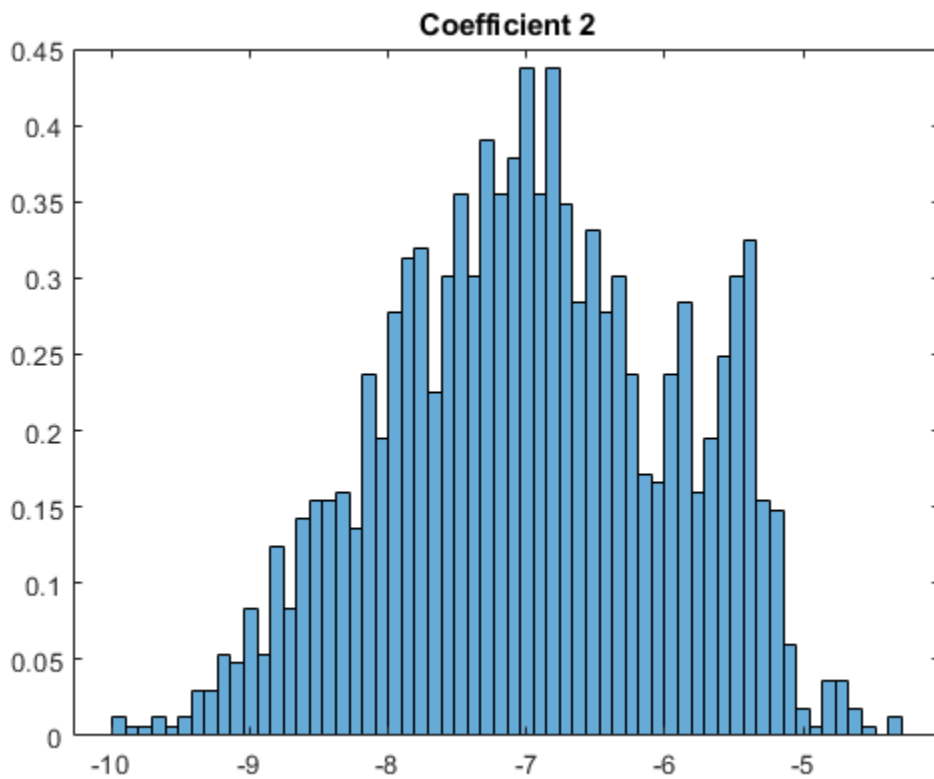
```
coeffs = mfcc(S,fs,"LogEnergy","Ignore");
```

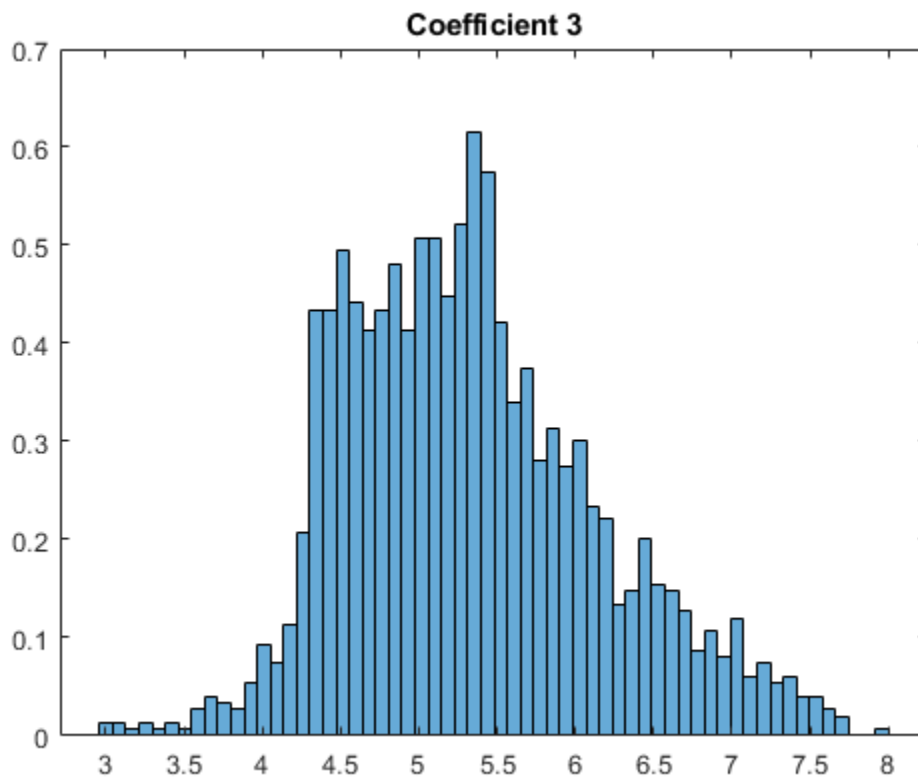
In many applications, MFCC observations are converted to summary statistics for use in classification tasks. Plot probability density functions of each of the mel-frequency cepstral coefficients to observe their distributions.

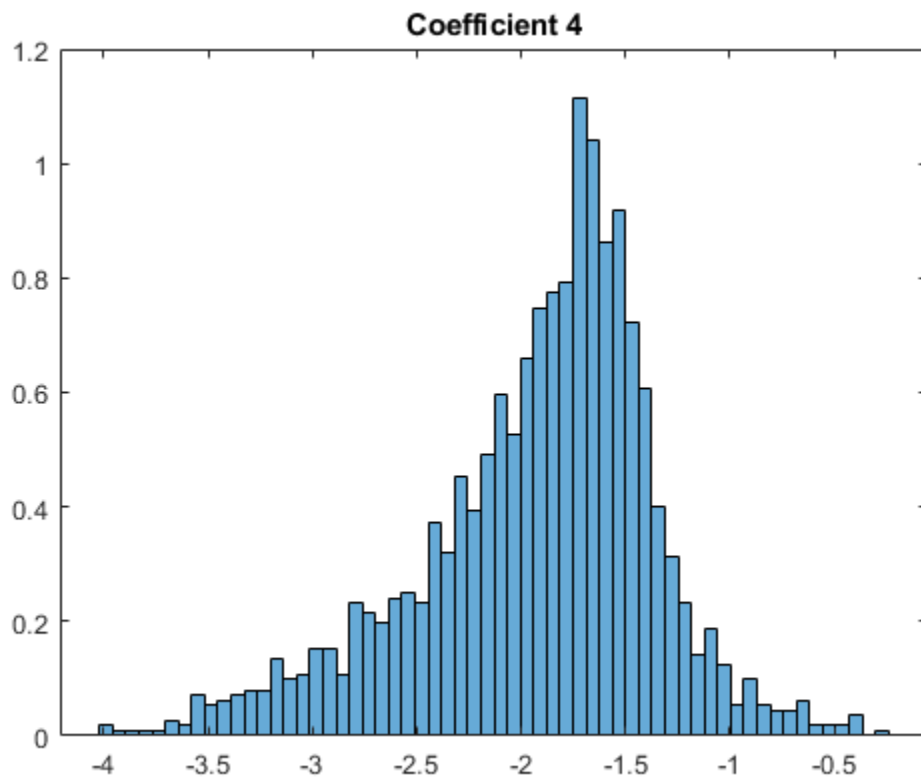
```
nbins = 60;  
for i = 1:size(coeffs,2)  
    figure  
    histogram(coeffs(:,i),nbins,"Normalization","pdf")  
    title(sprintf("Coefficient %d",i-1))  
end
```

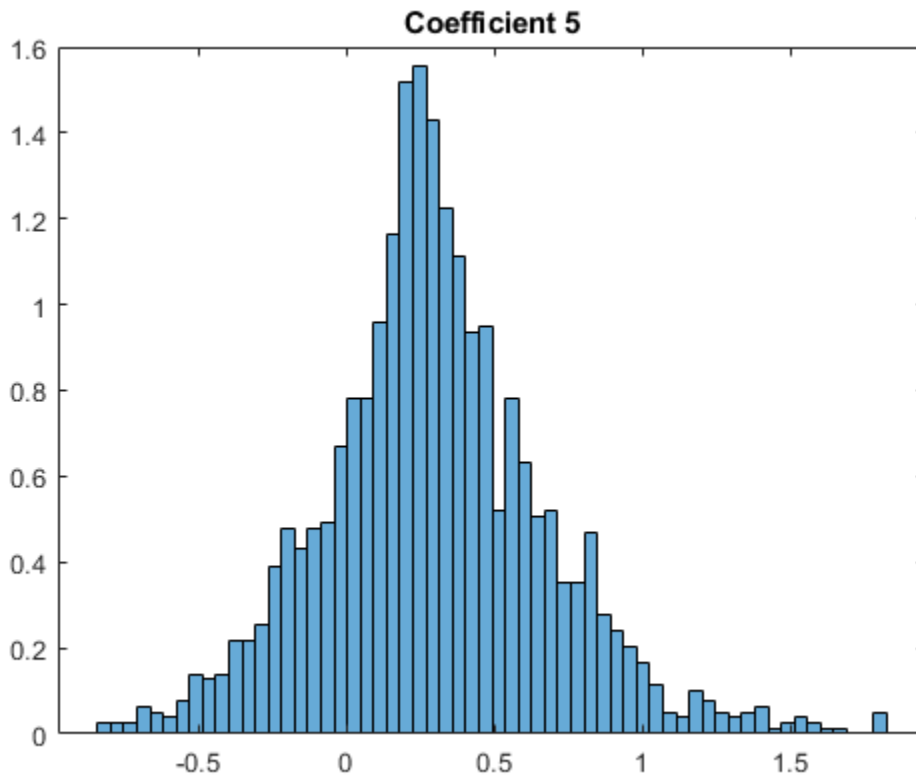




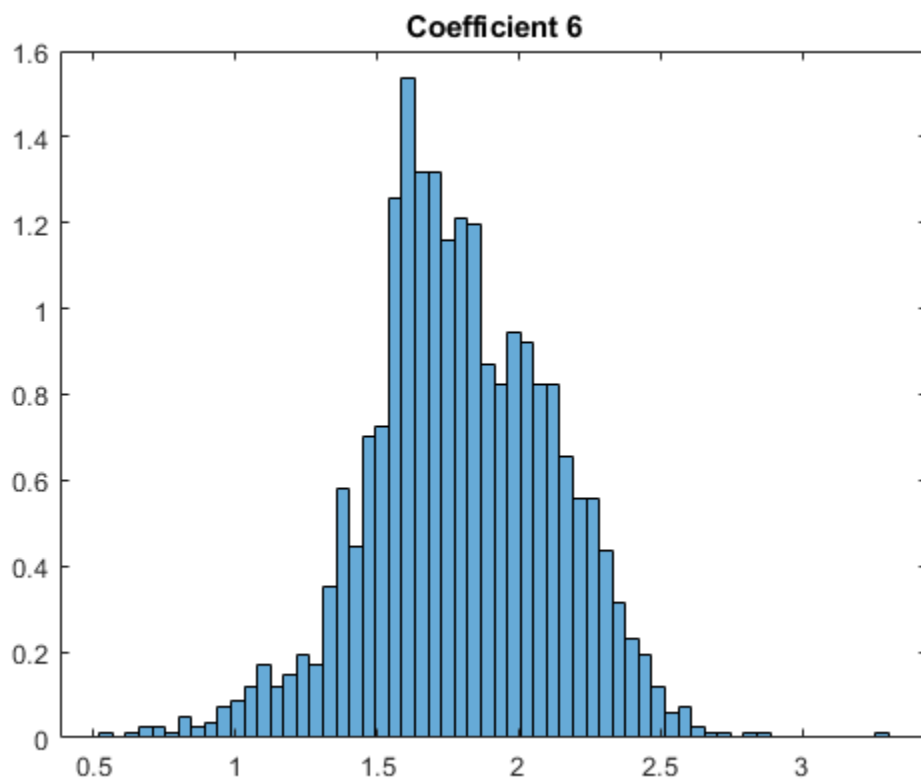


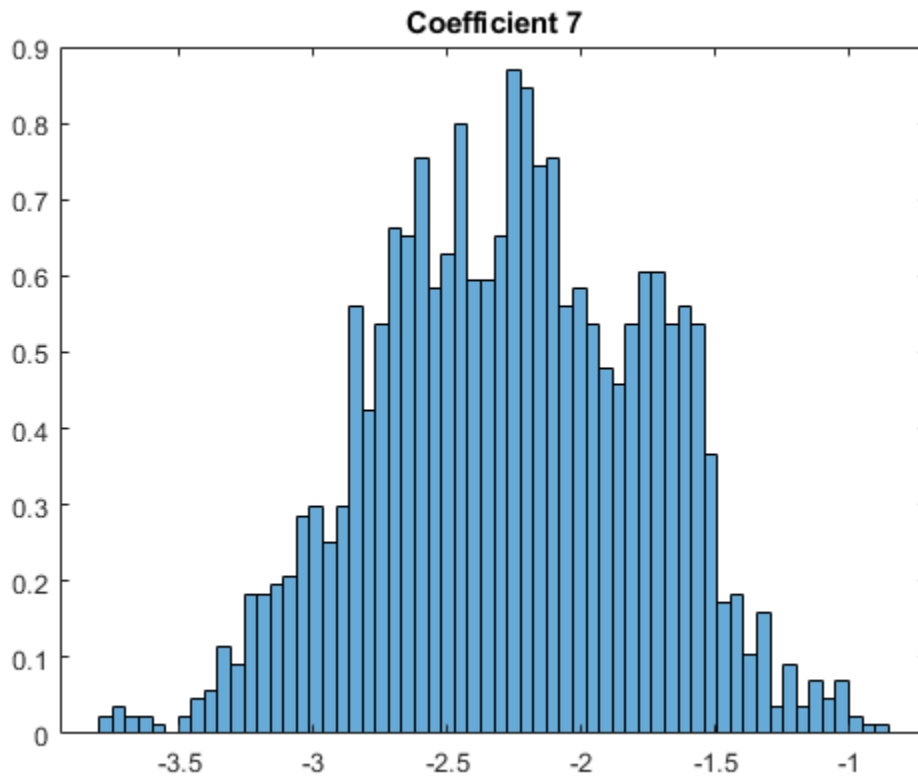


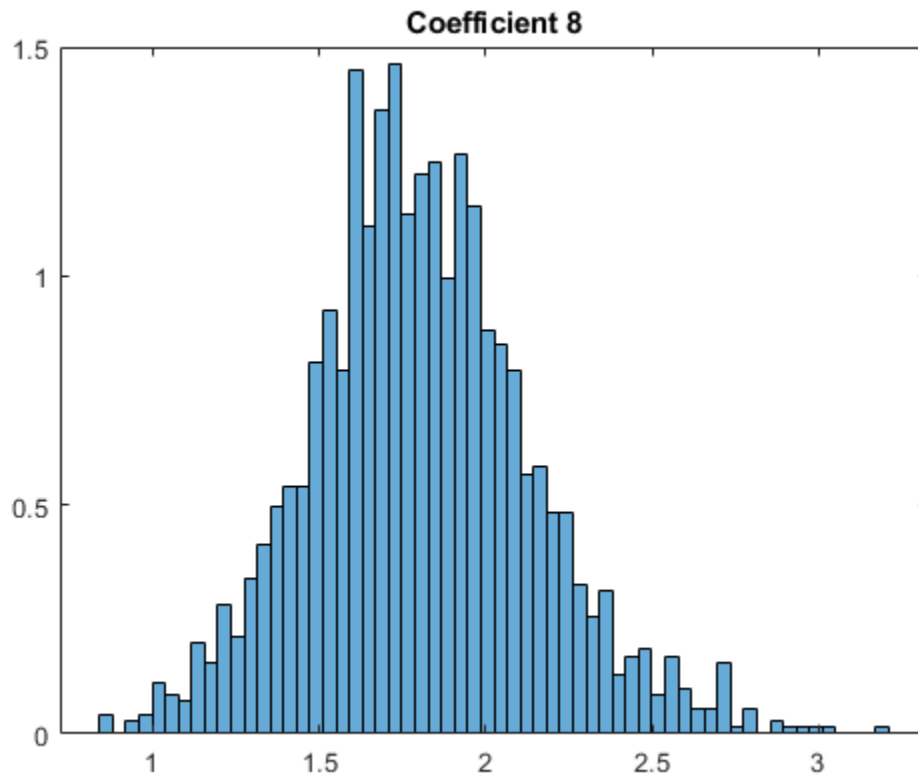


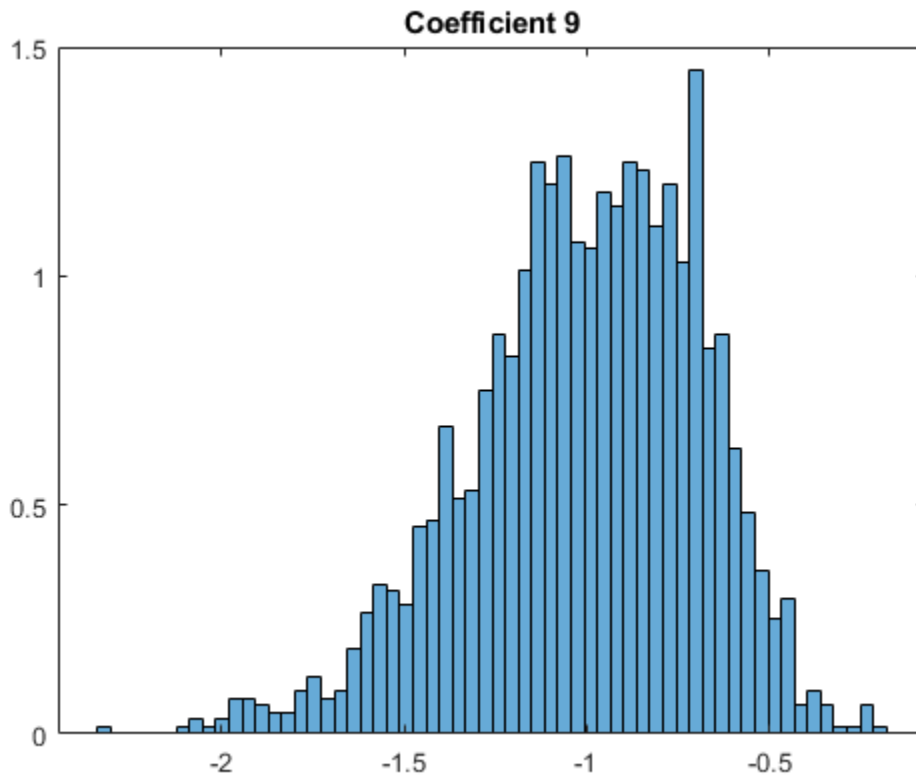


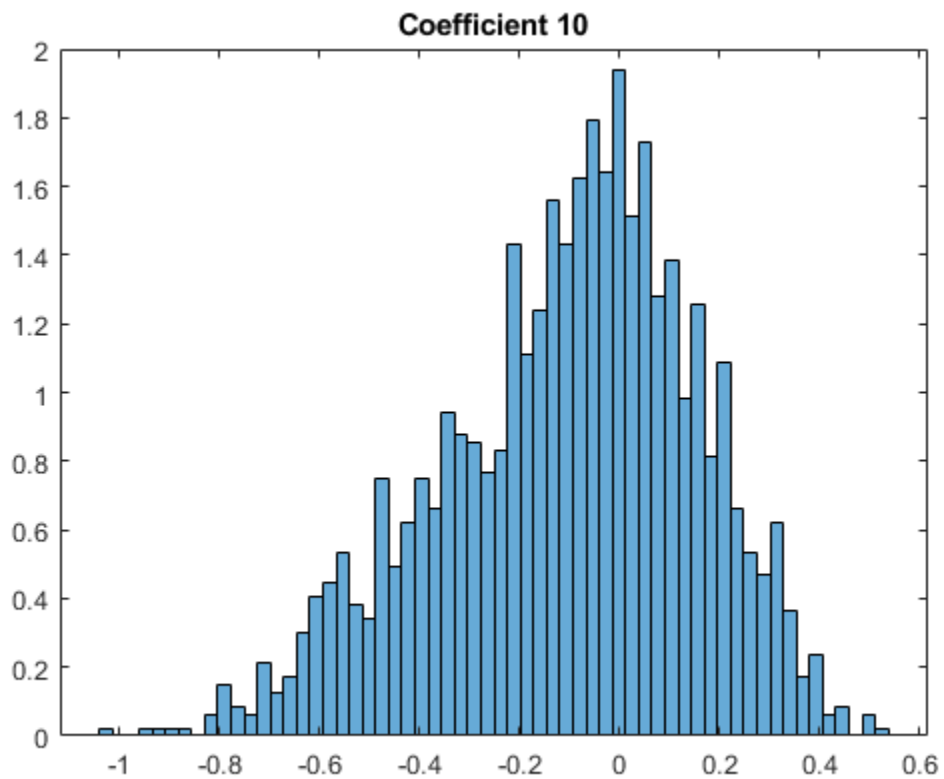


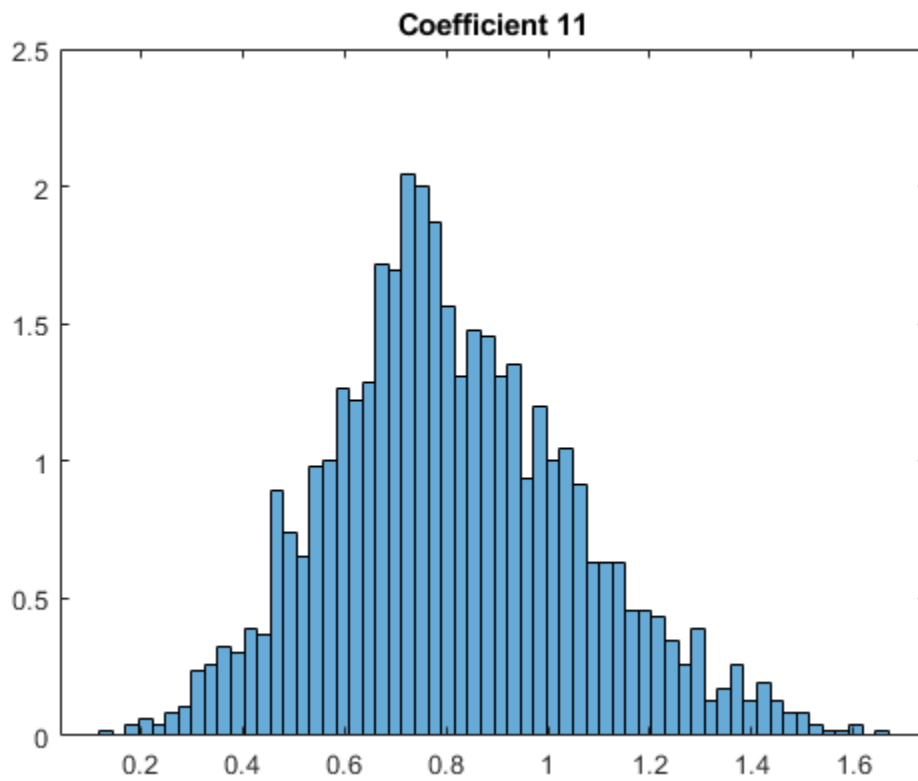


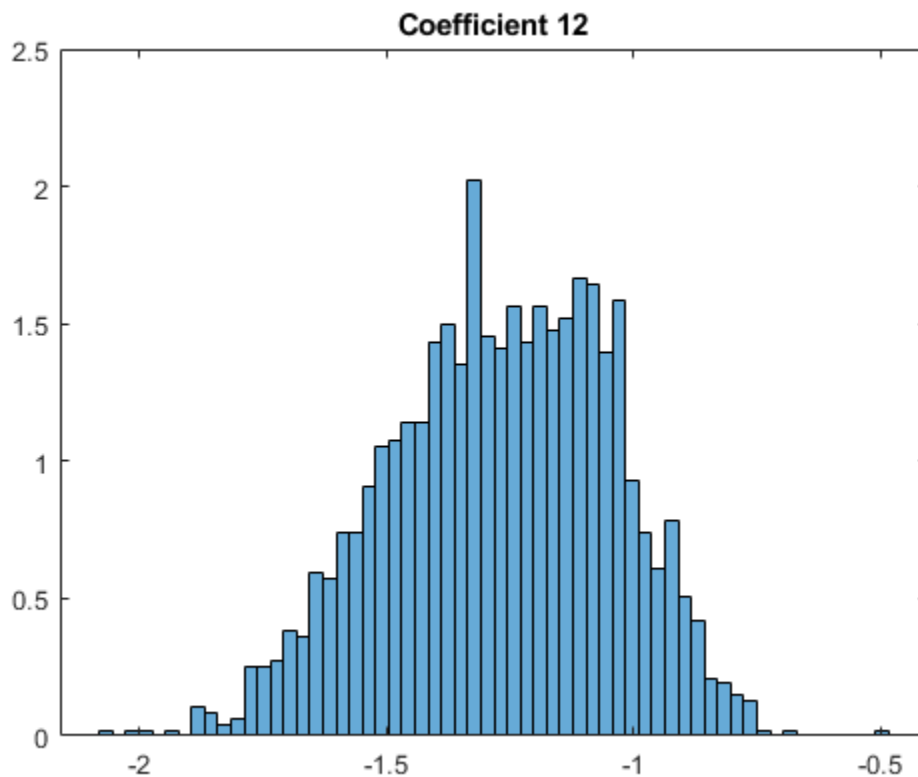












## Input Arguments

### **audioIn** — Input signal

vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array.

- If **audioIn** is real, it is interpreted as a time-domain signal and must be a column vector or a matrix. Columns of the matrix are treated as independent audio channels.

- If `audioIn` is complex, it is interpreted as a frequency-domain signal. In this case, `audioIn` must be an  $L$ -by- $M$ -by- $N$  array, where  $L$  is the number of DFT points,  $M$  is the number of individual spectrums, and  $N$  is the number of individual channels.

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

### **fs — Sample rate in Hz**

positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Data Types: `single` | `double`

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `[coeffs,delta,deltaDelta,loc] = mfcc(audioIn,fs,'LogEnergy','Replace','DeltaWindowLength',5)` returns mel frequency cepstral coefficients for the audio input signal sampled at `fs` Hz. The first coefficient in the `coeffs` vector is replaced with the log energy value. A set of 5 cepstral coefficients is used to compute the delta and the delta-delta values.

### **WindowLength — Number of samples in analysis window**

`round(fs*0.03)` (default) | positive scalar integer

Number of samples in analysis window used to calculate the coefficients, specified as an integer greater than or equal to 2. If unspecified, the `'WindowLength'` value defaults to `round(fs*0.03)`. Window length must be in the range `[2, size(audioIn,1)]`.

Data Types: `single` | `double`

### **OverlapLength — Number of overlapping samples between adjacent windows**

`round(fs*0.02)` (default) | integer

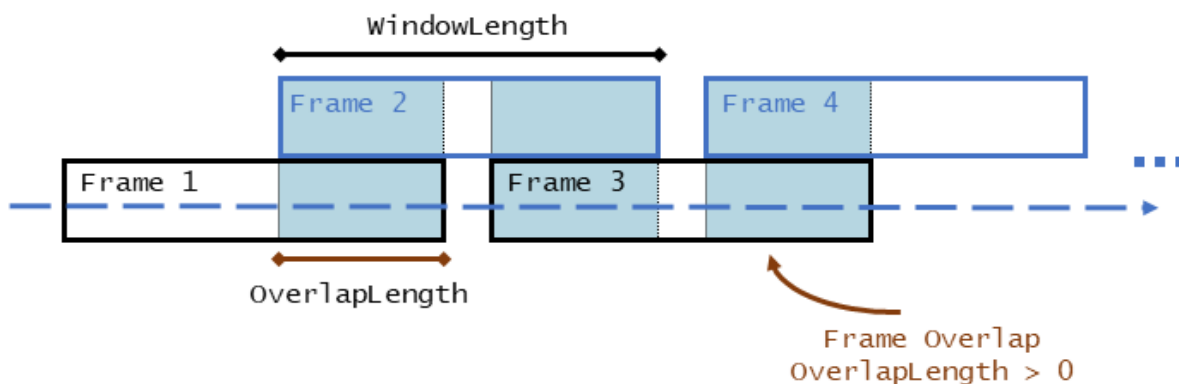
Number of samples which overlap or underlap between the adjacent windows. An `'OverlapLength'` value that is:



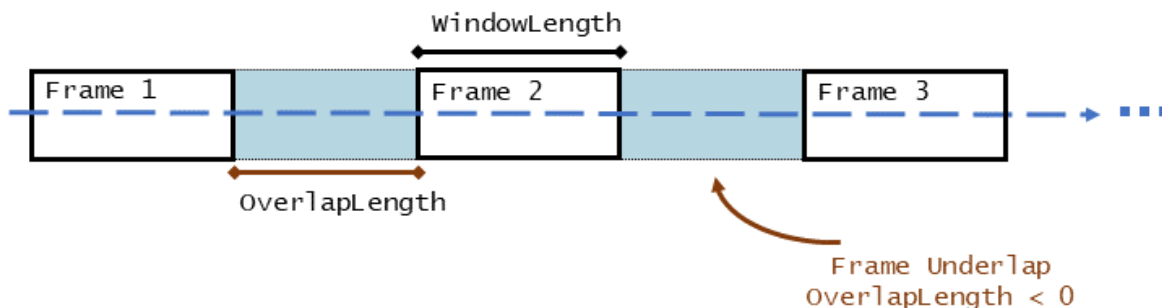
- Positive indicates an overlap between adjacent windows.
- Negative indicates an underlap between adjacent windows.
- Zero indicates no overlap between adjacent windows.

The 'OverlapLength' value must be set to less than the 'WindowLength'.

Here is how the overlapping frames look:



Here is how the underlapping frames look:



Data Types: single | double

**NumCoeffs** – Number of coefficients returned

13 (default) | positive scalar integer

Number of coefficients returned for each window of data, specified as an integer in the range  $[2 v]$ , where  $v$  is the number of valid passbands.

The number of valid passbands is defined as  $\text{sum}(\text{BandEdges} \leq \text{floor}(fs/2)) - 2$ . A passband is valid if its edges fall below  $fs/2$ , where  $fs$  is the sample rate of the input audio signal, specified as the second argument,  $fs$ .

Data Types: `single` | `double`

**BandEdges — Band edges of filter bank (Hz)**

row vector

Band edges of the filter bank in Hz, specified as a nonnegative monotonically increasing row vector in the range  $[0, fs/2]$ . The number of band edges must be in the range  $[4, 160]$ . The `mfcc` function designs half-overlapped triangular filters based on `BandEdges`. This means that all band edges, except for the first and last, are also center frequencies of the designed bandpass filters.

By default, `BandEdges` is a 42-element vector, which results in a 40-band filter bank that spans approximately 133 Hz to 6864 Hz:

Filters	Passband Edges (Hz)
Filter 1	[133 267]
Filter 2	[200 333]
Filter 3	[267 400]
Filter 4	[333 467]
Filter 5	[400 533]
Filter 6	[467 600]
Filter 7	[533 667]
Filter 8	[600 733]
Filter 9	[667 800]
Filter 10	[733 867]
Filter 11	[800 933]
Filter 12	[867 999]
Filter 13	[933 1071]

<b>Filters</b>	<b>Passband Edges (Hz)</b>
Filter 14	[999 1147]
Filter 15	[1071 1229]
Filter 16	[1147 1316]
Filter 17	[1229 1410]
Filter 18	[1316 1510]
Filter 19	[1410 1618]
Filter 20	[1510 1733]
Filter 21	[1618 1856]
Filter 22	[1733 1988]
Filter 23	[1856 2130]
Filter 24	[1988 2281]
Filter 25	[2130 2444]
Filter 26	[2281 2618]
Filter 27	[2444 2804]
Filter 28	[2618 3004]
Filter 29	[2804 3217]
Filter 30	[3004 3446]
Filter 31	[3217 3692]
Filter 32	[3446 3954]
Filter 33	[3692 4236]
Filter 34	[3954 4537]
Filter 35	[4236 4860]
Filter 36	[4537 5206]
Filter 37	[4860 5577]
Filter 38	[5206 5973]
Filter 39	[5577 6399]
Filter 40	[5973 6854]

The passband edges in the table are rounded for readability. For exact edges, see the default settings of the `cepstralFeatureExtractor`.

Data Types: `single` | `double`

### **FFTLength — Number of bins for calculating DFT**

`WindowLength` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples. The FFT length value must be greater than or equal to the 'WindowLength' value. The 'WindowLength' argument specifies the number of rows in the windowed input. By default, the FFT length value is set to the 'WindowLength'.

Data Types: `single` | `double`

### **DeltaWindowLength — Number of coefficients for calculating delta and delta-delta**

2 (default) | odd integer greater than 2

Number of coefficients used to calculate the delta and the delta-delta values, specified as 2 or an odd integer greater than 2.

If 'DeltaWindowLength' is set to 2, the delta is given by the difference between the current coefficients and the previous coefficients,  $delta = currCoeffs - prevCoeffs$

If 'DeltaWindowLength' is set to an odd integer greater than 2, the delta values are given by the following equation:

$$delta = \frac{\sum_{k=-M}^M k \cdot coeffs(k,:)}{\sum_{k=-M}^M k^2}$$

The function uses a least-squares approximation of the local slope over a region around the current time sample. The delta cepstral values are computed by fitting the cepstral coefficients of neighboring frames ( $M$  frames before the current frame and  $M$  frames after the current frame) by a straight line. For details, see [1].

Data Types: `single` | `double`

### LogEnergy — Specify how the log energy is shown

'Append' (default) | 'Replace' | 'Ignore'

Specify how the log energy is shown in the coefficients vector output, specified as:

- 'Append' -- The function prepends the log energy to the coefficients vector. The length of the coefficients vector is  $1 + \text{NumCoeffs}$ .
- 'Replace' -- The function replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is  $\text{NumCoeffs}$ .
- 'Ignore' -- The object does not calculate or return the log energy.

Data Types: char | string

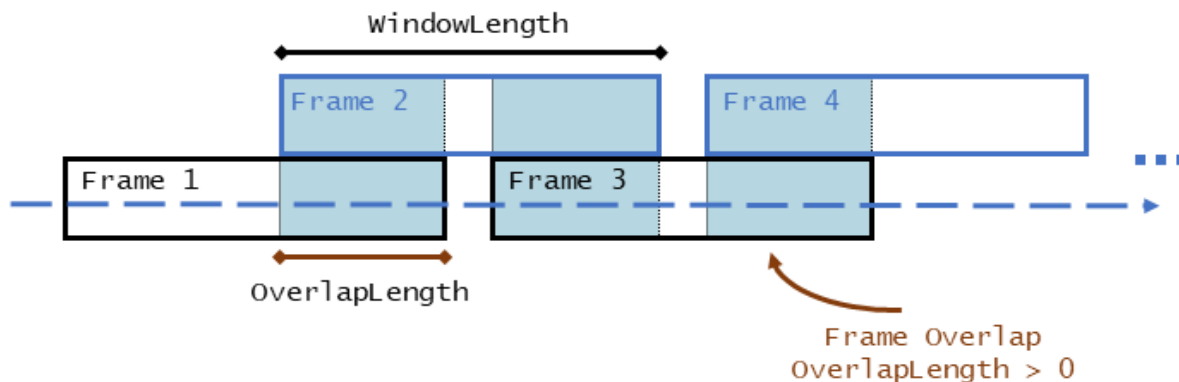
## Output Arguments

### coeffs — Mel frequency cepstral coefficients (MFCCs)

matrix | array

Mel frequency cepstral coefficients, returned as an  $L$ -by- $M$  matrix or an  $L$ -by- $M$ -by- $N$  array, where,

- $L$  -- Number of frames the audio signal is partitioned into. The 'WindowLength' and 'OverlapLength' properties control this dimension.



The number of audio frames,  $L$ , is computed using the following equation:

$$L = \text{floor}((nRows - winLen) / hopLen) + 1$$

- *nRows* -- Number of input rows.
- *winLen* -- Number of samples in the analysis window, specified by the 'WindowLength' argument. If not specified, the window length is  $\text{round}(\text{fs} * 0.03)$ .
- *hopLen* -- Number of samples in the current frame before the start of the next frame. Hop length is given by  $\text{hopLen} = \text{WindowLength} - \text{OverlapLength}$ .
- *M* -- Number of coefficients returned per frame. This value is determined by the NumCoeffs and LogEnergy properties.

When the LogEnergy property is set to:

- 'Append' -- The object prepends the log energy value to the coefficients vector. The length of the coefficients vector is  $1 + \text{NumCoeffs}$ .
- 'Replace' -- The object replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is NumCoeffs.
- 'Ignore' -- The object does not calculate or return the log energy.
- *N* -- Number of input channels (columns).

Data Types: `single` | `double`

### **delta** — Change in coefficients

`matrix` | `array`

Change in coefficients from one frame of data to another, returned as an *L*-by-*M* matrix or an *L*-by-*M*-by-*N* array. The `delta` array is the same size and data type as the `coeffs` array.

If 'DeltaWindowLength' is set to 2, the `delta` is given by the difference between the current coefficients and the previous coefficients,  $\text{delta} = \text{currCoeffs} - \text{prevCoeffs}$ .

Consider the example below which computes the mel frequency coefficients for the entire speech file. The 'DeltaWindowLength' value is 2. The `mfcc` function partitions the speech into 1551 frames. Each row in the `coeffs` matrix corresponds to the log energy value followed by the 13 mel frequency cepstral coefficients for the corresponding segment of the speech file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
[coeffs,delta,deltaDelta,loc] = mfcc(audioIn,fs);
```

The first row of the delta matrix, `delta(1, :)` is zeros. The second row, `delta(2, :)` equals the difference in coefficients for the current frame, `coeffs(2, :)` and the previous frame, `coeffs(1, :)`.

If 'DeltaWindowLength' is set to an odd integer greater than 2, the delta values are given by the following equation:

$$\mathit{delta} = \frac{\sum_{k=-M}^M k \cdot \mathit{coeffs}(k, :)}{\sum_{k=-M}^M k^2}$$

The function uses a least-squares approximation of the local slope over a region around the current time sample. For details, see [1].

Data Types: `single` | `double`

### **deltaDelta — Change in delta values**

matrix | array

Change in delta values from one frame of data to another, returned as an  $L$ -by- $M$  matrix or an  $L$ -by- $M$ -by- $N$  array. The `deltaDelta` array is the same size and data type as the `coeffs` and `delta` arrays.

If 'DeltaWindowLength' is set to 2, the `deltaDelta` is given by the difference between the current delta values and the previous delta values,

$$\mathit{deltaDelta} = \mathit{currdelta} - \mathit{prevdelta}$$

Consider the example below which computes the mel frequency coefficients for the entire speech file. The 'DeltaWindowLength' value is 2.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
[coeffs,delta,deltaDelta,loc] = mfcc(audioIn,fs);
```

The first row of the `deltaDelta` matrix, `deltaDelta(1, :)` is zeros. The second row, `deltaDelta(2, :)` equals the difference in delta values for the current frame, `delta(2, :)` and the previous frame, `delta(1, :)`.

If 'DeltaWindowLength' is set to an odd integer greater than 2, the `deltaDelta` values are given by the following equation:

$$\text{deltaDelta} = \frac{\sum_{k=-M}^M k \cdot \text{delta}(k,:)}{\sum_{k=-M}^M k^2}$$

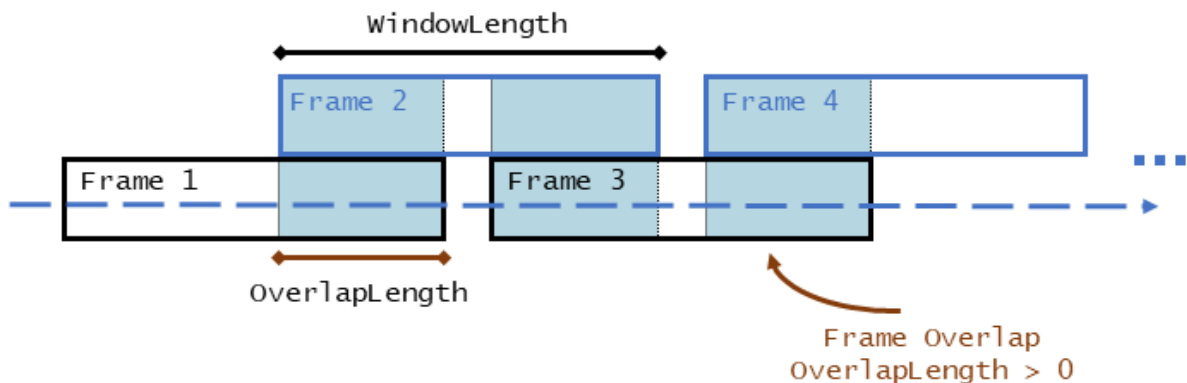
The function uses a least-squares approximation of the local slope over a region around the current time sample. For details, see [1].

Data Types: `single` | `double`

### **loc** — Location of the last sample in each input frame

vector

Location of last sample in each input frame, returned as a vector. The `loc` vector is given by the  $[t_1, t_2, t_3, \dots, t_n]$  elements in the following diagram, where  $n$  corresponds to the number of frames the input is partitioned into, and  $t_n$  is the last sample of the last frame.

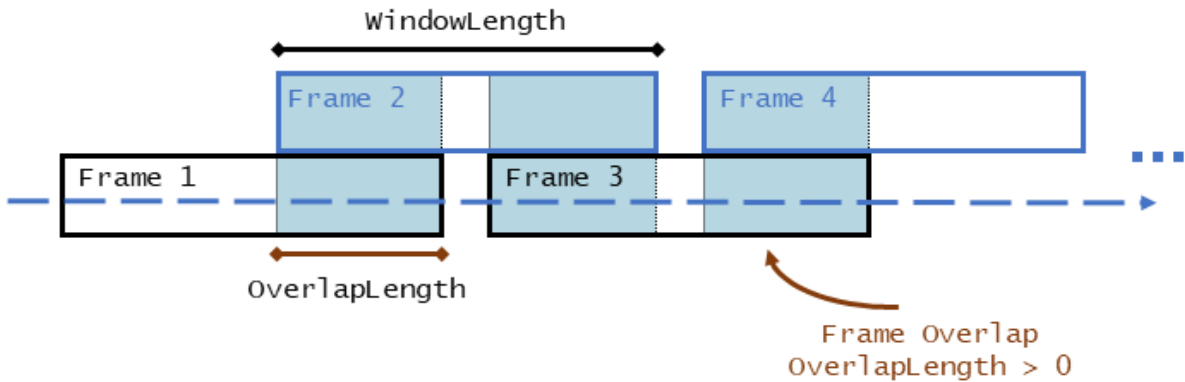


Data Types: `single` | `double`

## **Algorithms**

The `mfcc` function splits the entire data into overlapping segments. The length of each rolloff segment is determined by the 'WindowLength' argument. The length of overlap between segments is determined by the 'OverlapLength' argument.





The function computes the mel frequency cepstral coefficients, log energy values, cepstral delta, and the cepstral delta-delta values for each segment as per the algorithm described in `cepstralFeatureExtractor`.

## References

- [1] Rabiner, Lawrence R., and Ronald W. Schafer. *Theory and Applications of Digital Speech Processing*. Upper Saddle River, NJ: Pearson, 2010.
- [2] Auditory Toolbox. <https://engineering.purdue.edu/~malcolm/interval/1998-010/AuditoryToolboxTechReport.pdf>

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`Cepstral Feature Extractor` | `Voice Activity Detector` | `audioFeatureExtractor` | `cepstralFeatureExtractor` | `pitch` | `voiceActivityDetector`

## **Topics**

“Keyword Spotting in Noise Using MFCC and LSTM Networks”

“Speaker Identification Using Pitch and MFCC”

**Introduced in R2018a**

# asiosettings

Open settings panel for ASIO driver

## Syntax

```
asiosettings  
asiosettings(deviceName)
```

## Description

`asiosettings` opens the settings panel for the ASIO driver associated with the default audio device.

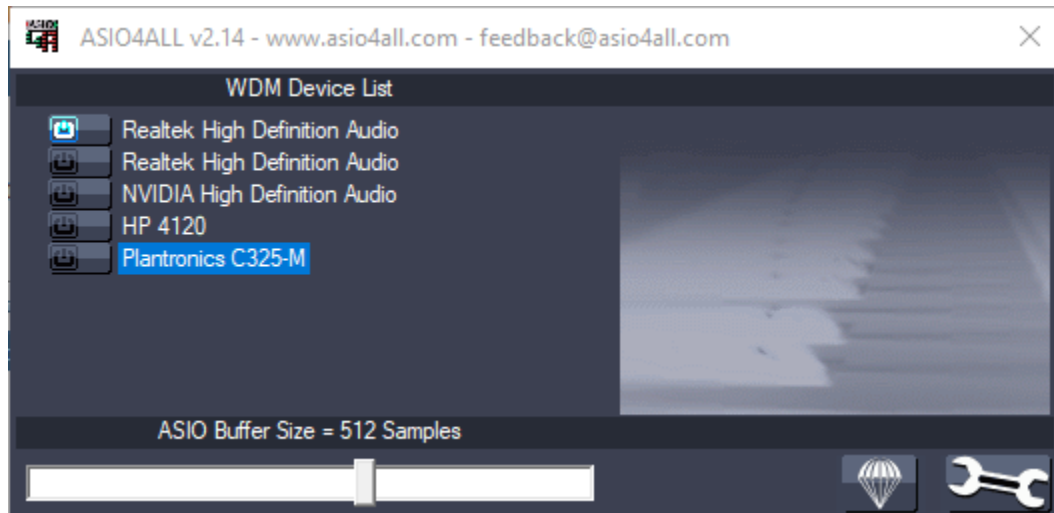
`asiosettings(deviceName)` opens the settings panel for the ASIO driver associated with the audio device, `deviceName`.

## Examples

### Open ASIO Settings Panel for Specified Device

Create an audio I/O object, `audioPlayerRecorder`. Call `asiosettings` with the device associated with `audioPlayerRecorder` as the argument.

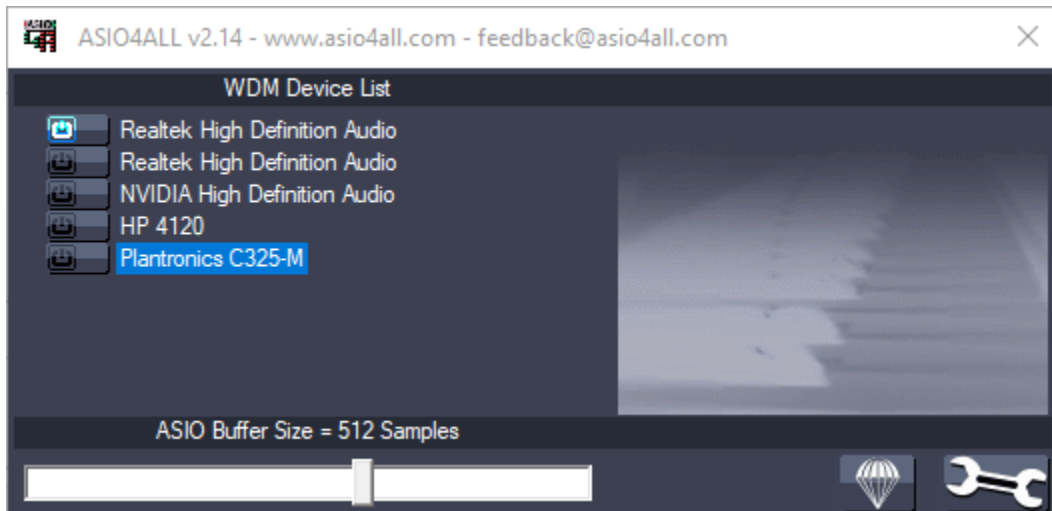
```
playRec = audioPlayerRecorder;  
asiosettings(playRec.Device)
```



### Open ASIO Settings Panel for Default Device

Call the `asiosettings` function with no arguments.

```
asiosettings()
```



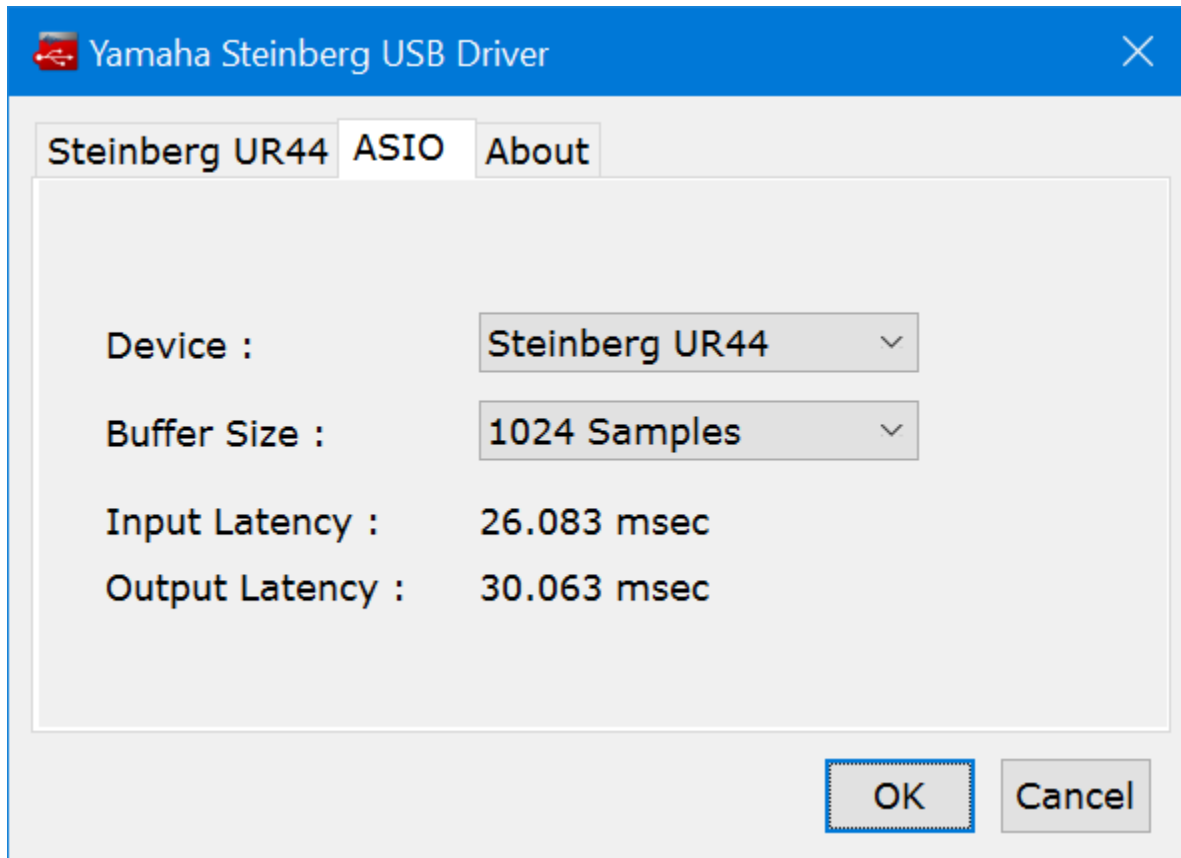
## Optimize Latency

To optimize latency when using an ASIO driver, set the buffer size of the ASIO driver to the buffer size of your audio I/O object. In this example, assume the input to your audio device writer is 64 samples per frame. This example requires a Windows machine and an ASIO driver.

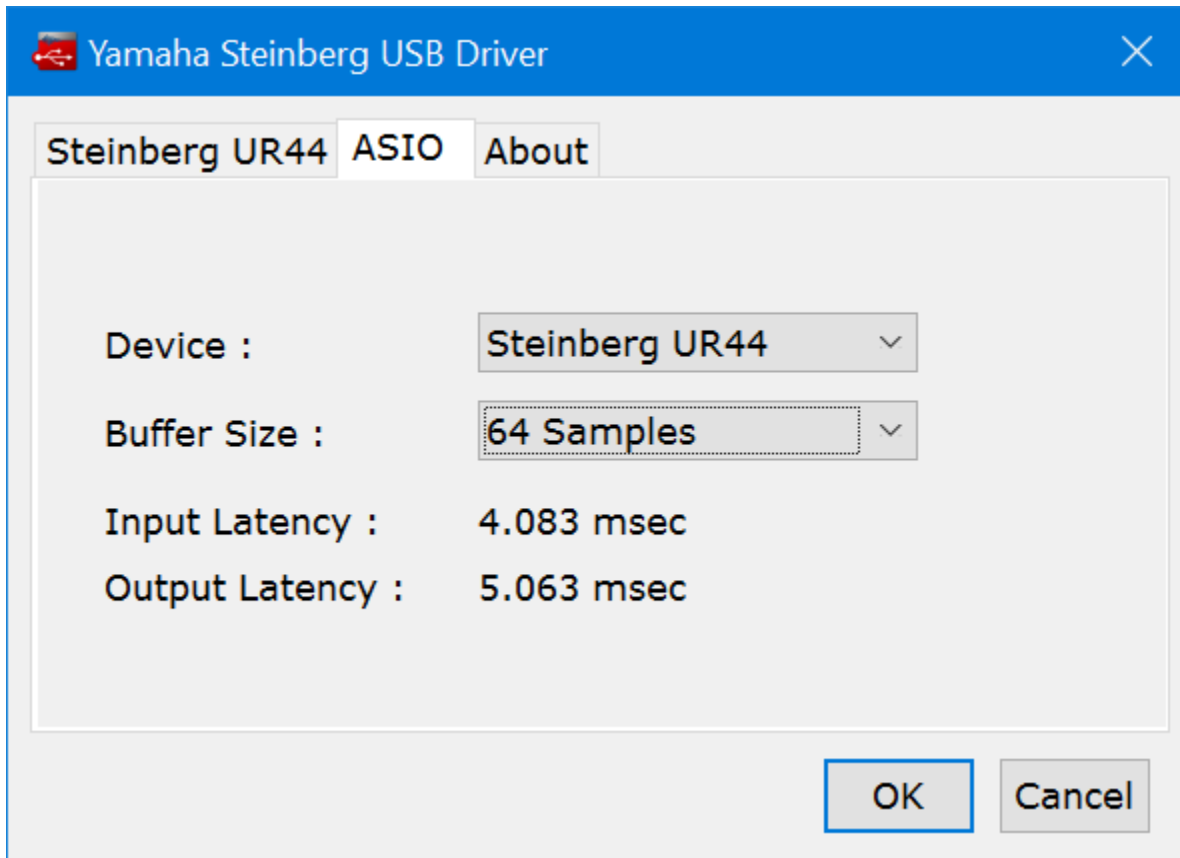
Create an `audioDeviceWriter` System object™. Open the ASIO settings panel for an ASIO-compatible device associated with your device writer.

```
deviceWriter = audioDeviceWriter('Driver', 'ASIO');  
asio4all(deviceWriter.Device)
```

On the machine in this example, the following dialog opens:



The dialog that opens is specific to your ASIO driver. Set the ASIO buffer size to the desired size, 64.



The latency is now minimized for the frame size of 64 samples. If you want to measure the reduction in latency specific to your system, follow the steps in the Measure Audio Latency example.

## Input Arguments

### **deviceName** — Name of ASIO-compatible device

default ASIO-compatible device (default) | character vector | string

Name of ASIO-compatible device, specified as a character vector or string. If deviceName is not specified, the default ASIO-compatible device is used.

To view a list of valid ASIO device names on your machine, use `getAudioDevices` on an `audioPlayerRecorder`, `audioDeviceReader('Driver', 'ASIO')`, or `audioDeviceWriter('Driver', 'ASIO')` object.

Data Types: `char` | `string`

### Tips

- `asioSettings` is compatible only on Windows machines with ASIO drivers. ASIO drivers do not come pre-installed with Windows.
- `asioSettings` returns an error if called with a locked audio device. For example:

```
aDR = audioDeviceReader('Driver', 'ASIO');  
aDR();  
asioSettings(aDR.Device)
```

```
Error using audio_asioSettings  
PortAudio Error: Device unavailable
```

```
Error in asioSettings (line 77)  
    audio_asioSettings(ID);
```

### See Also

`audioDeviceReader` | `audioDeviceWriter` | `audioPlayerRecorder`

### Topics

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

**Introduced in R2017b**



# getAudioDevices

List available audio devices

## Syntax

```
devices = getAudioDevices(obj)
```

## Description

`devices = getAudioDevices(obj)` returns a list of audio devices that are available and compatible with your audio I/O object, `obj`.

## Examples

### List Audio Devices Available to `audioDeviceReader`

Create an `audioDeviceReader` object and then call `getAudioDevices` on your object.

```
deviceReader = audioDeviceReader;  
devices = getAudioDevices(deviceReader)  
  
devices = 1x4 cell array  
    {'Default'}    {'Primary Sound Capture Driver'}    {'Headset Microphone (Plantronics)'}  
    {}
```

### List Audio Devices Available to `audioDeviceWriter`

Create an `audioDeviceWriter` object, and then call `getAudioDevices` on your object.

```
deviceWriter = audioDeviceWriter;  
devices = getAudioDevices(deviceWriter)
```

```
devices = 1x6 cell array
    {'Default'}    {'Primary Sound Driver'}    {'Headset Earphone (Plantronics C325-M)'}
    {'Headset Earphone (Plantronics C325-M)'}
    {'Headset Earphone (Plantronics C325-M)'}
    {'Headset Earphone (Plantronics C325-M)'}
    {'Headset Earphone (Plantronics C325-M)'}
    {'Headset Earphone (Plantronics C325-M)'}
```

### List Audio Devices Available to `audioPlayerRecorder`

Create an `audioPlayerRecorder` object, and then call `getAudioDevices` on your object.

```
playRec = audioPlayerRecorder;
devices = getAudioDevices(playRec)
```

```
devices = 1x2 cell array
    {'Default'}    {'ASIO4ALL v2'}
```

## Input Arguments

### **obj** — Audio I/O object

object of `audioDeviceReader` | object of `audioDeviceWriter` | object of `audioPlayerRecorder`

Audio I/O object, specified as an object of `audioDeviceReader`, `audioDeviceWriter`, or `audioPlayerRecorder`.

Data Types: `object`

## Output Arguments

### **devices** — List of available and compatible devices

array

List of available and compatible devices.

For `audioDeviceReader` and `audioDeviceWriter`, the list of audio devices depends on the specified `Driver` property of your object.

For `audioPlayerRecorder`, the audio devices listed support full-duplex mode and have a platform-appropriate driver:

- Windows® -- ASIO™
- Mac -- CoreAudio
- Linux® -- ALSA

Data Types: cell

## Tips

Devices are persistent within a MATLAB session. To recognize new devices within your MATLAB session, clear device data within your session using the command line. As an example, if you have created an `audioDeviceReader` System object, you can type the following into your command line:

```
>> deviceReader = audioDeviceReader;
>> devices = getAudioDevices(deviceReader)
```

```
devices =
```

```
    1×1 cell array
```

```
    {'No audio input device detected'}
```

This displays a list of the devices connected to your computer. To add more devices, connect the additional devices to your computer. Then, type the following into your command line:

```
>> clear deviceReader dspAudioDeviceInfo
>> deviceReader = audioDeviceReader;
>> devices = getAudioDevices(deviceReader)
```

```
devices =
```

```
    1×3 cell array
```

```
    {'Default'}    {'Primary Sound Capture Driver'}    {'Headset Microphone (Planthro...}'
```

This displays an updated list of the devices connected to your computer, including the devices you added during your current session. This process also works with the `audioDeviceWriter` and `audioPlayerRecorder` System objects.

## **See Also**

`audioDeviceReader` | `audioDeviceWriter` | `audioPlayerRecorder`

## **Topics**

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

**Introduced in R2016a**

# audioPluginInterface

Specify audio plugin interface

## Syntax

```
PluginInterface = audioPluginInterface
PluginInterface = audioPluginInterface(pluginParameters)
PluginInterface = audioPluginInterface(pluginParameters,gridLayout)
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.

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

`PluginInterface = audioPluginInterface(pluginParameters,gridLayout)` specifies a grid layout for audio plugin parameter UI controls.

`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
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, number of output channels, and a yellow background.

```

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, ...
        'BackgroundColor','y');
end
methods
    function out = process(plugin,in)
        out = in*plugin.Gain;
    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, audio plugin parameters synchronize plugin class properties with user-facing parameters.

### **gridLayout** — Layout for plugin UI

none (default) | `audioPluginGridLayout` object

Audio plugin grid layout, specified as an `audioPluginGridLayout` object.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside 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) | character vector | string

Name of your generated plugin, as seen by a host audio application, specified as a comma-separated pair consisting of 'PluginName' and a character vector or 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 plugin creator**

' ' (default) | character vector

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

### **VendorVersion — Vendor version**

'1.0.0' (default) | dot-separated character vector or string

Vendor version used to track plugin releases, specified as a comma-separated pair consisting of 'VendorVersion' and a dot-separated character vector or 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-element character vector or string

Unique identifier for your plugin, specified as a comma-separated pair consisting of 'UniqueID' and a four-element character vector or 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.

### **BackgroundColor — Color used for GUI background**

RGB triplet | short name | long name

Color used for GUI background, specified as short or long color name string, or an RGB triplet. See `ColorSpec` ([Color Specification](#)) for details.

Example: `'BackgroundColor', [1 1 0]` specifies the GUI background to be yellow.

Example: `'BackgroundColor', 'y'` specifies the GUI background to be yellow.

Example: `'BackgroundColor', 'yellow'` specifies the GUI background to be yellow.

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

### **BackgroundImage — Image used for GUI background**

`char` | `string`

Image used for GUI background, specified by its file name using either a character vector or string. If the file is not on path, you must specify the full file path. Supported file types are PNG, GIF, and JPG.

The background image may include transparencies, in which case the `BackgroundColor` is used.

Example: `'BackgroundImage', 'Sunrise.png'` specifies the GUI background image to be the 'Sunrise' image.

Example: `'BackgroundImage', fullfile(matlabroot, "mySkins", "Sunset.jpg")` specifies the GUI background to be the 'Sunset' image.

Data Types: `char` | `string`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`audioPlugin` | `audioPluginGridLayout` | `audioPluginParameter` |  
`audioPluginSource` | `generateAudioPlugin` | `validateAudioPlugin`

### Topics

"Audio Plugins in MATLAB"

**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 `audioPluginInterface` in your plugin class definition.

In a digital audio workstation (DAW) environment, or when using **Audio Test Bench** or `parameterTuner` in the MATLAB environment, plugin parameters are tunable, user-facing values 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 2-423.

`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
end
```

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

### Integer Parameter Mapping

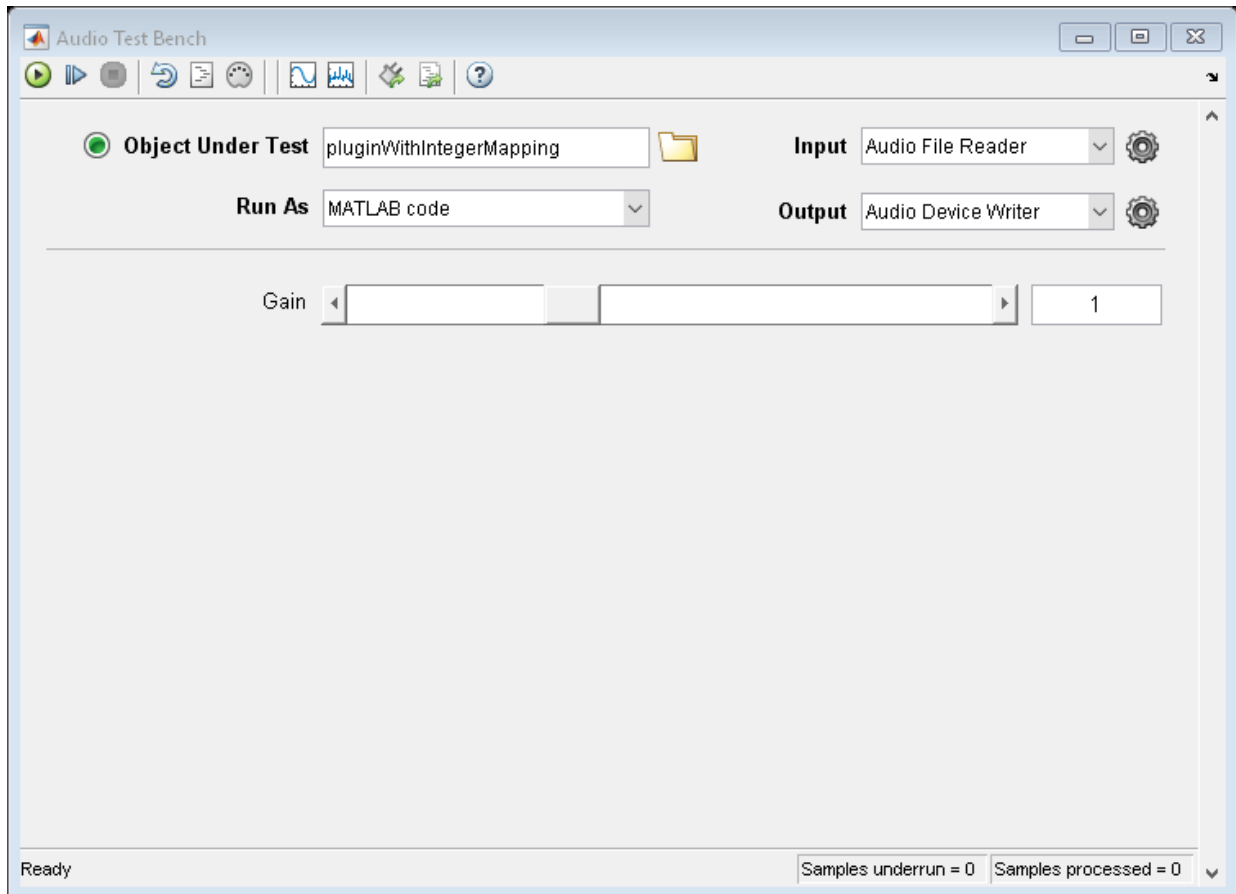
The following class definition uses integer parameter mapping to define the relationship between a property and a parameter. You can use the plugin created from this class to tune the linear gain of an audio signal in integer steps from 0 to 3.

```
classdef pluginWithIntegerMapping < audioPlugin
    properties
        Gain = 1;
    end
end
```

```
properties (Constant)
    PluginInterface = audioPluginInterface( ...
        audioPluginParameter('Gain', ...
            'Mapping', {'int',0,3}));
end
methods
    function out = process(plugin,in)
        out = in*plugin.Gain;
    end
end
end
```

To run the plugin, save the class definition to a local folder and then call the Audio Test Bench.

```
audioTestBench(pluginWithIntegerMapping)
```



### Power Parameter Mapping

The following class definition uses power parameter mapping to define the relationship between a property and a parameter. You can use the plugin created from this class to tune the gain of an audio signal in dB.

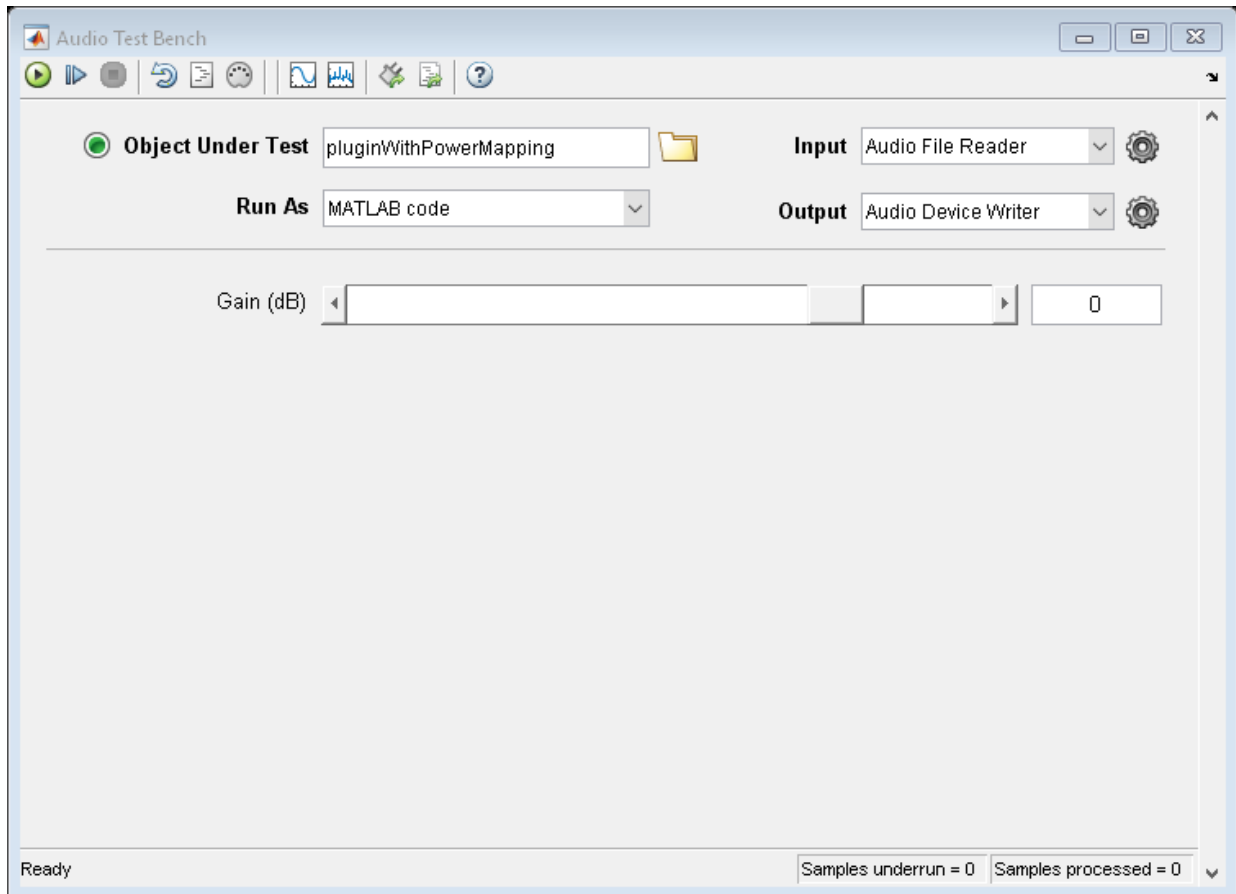
```
classdef pluginWithPowerMapping < audioPlugin
    properties
        Gain = 0;
```



```
end
properties (Constant)
    PluginInterface = audioPluginInterface( ...
        audioPluginParameter('Gain', ...
            'Label', 'dB', ...
            'Mapping', {'pow', 1/3, -140, 12}));
end
methods
    function out = process(plugin,in)
        dBGain = 10^(plugin.Gain/20);
        out = in*dBGain;
    end
end
end
```

To run the plugin, save the class definition to a local folder and then call the Audio Test Bench.

```
audioTestBench(pluginWithPowerMapping)
```



### Logarithmic Parameter Mapping

The following class definition uses logarithmic parameter mapping to define the relationship between a property and a parameter. You can use the plugin created from this class to tune the center frequency of a single-band EQ filter from 100 to 10000.

```
classdef pluginWithLogMapping < audioPlugin
    properties
        EQ
```

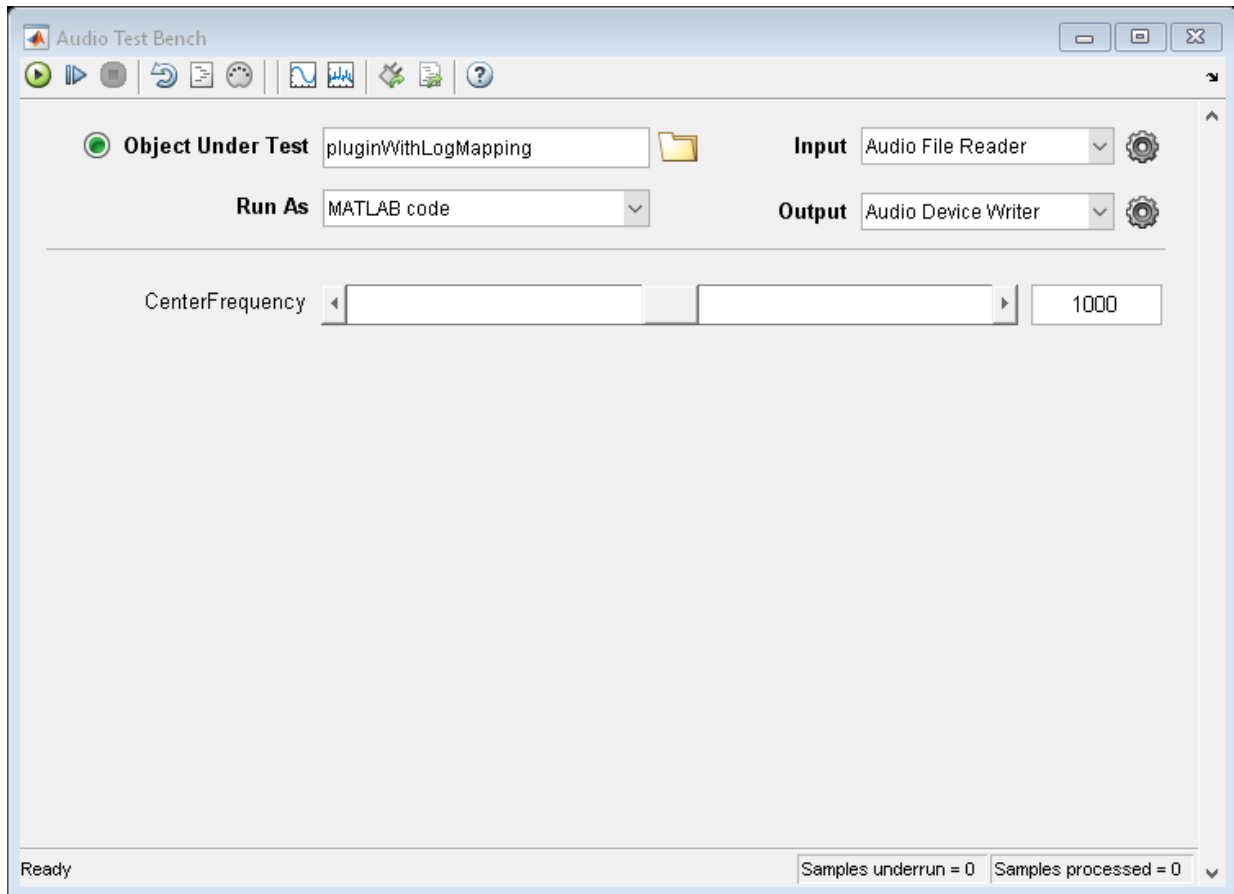
```

        CenterFrequency = 1000;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('CenterFrequency', ...
                'Mapping', {'log',100,10000}));
    end
    methods
        function plugin = pluginWithLogMapping
            plugin.EQ = multibandParametricEQ('NumEQBands',1, ...
                'PeakGains',20, ...
                'Frequencies',plugin.CenterFrequency);
        end
        function out = process(plugin,in)
            out = plugin.EQ(in);
        end
        function set.CenterFrequency(plugin,val)
            plugin.CenterFrequency = val;
            plugin.EQ.Frequencies = val;
        end
        function reset(plugin)
            plugin.EQ.SampleRate = getSampleRate(plugin);
        end
    end
end
end

```

To run the plugin, save the class definition to a local folder and then call the Audio Test Bench.

```
audioTestBench(pluginWithLogMapping)
```



### Enumeration for Logical Properties Parameter Mapping

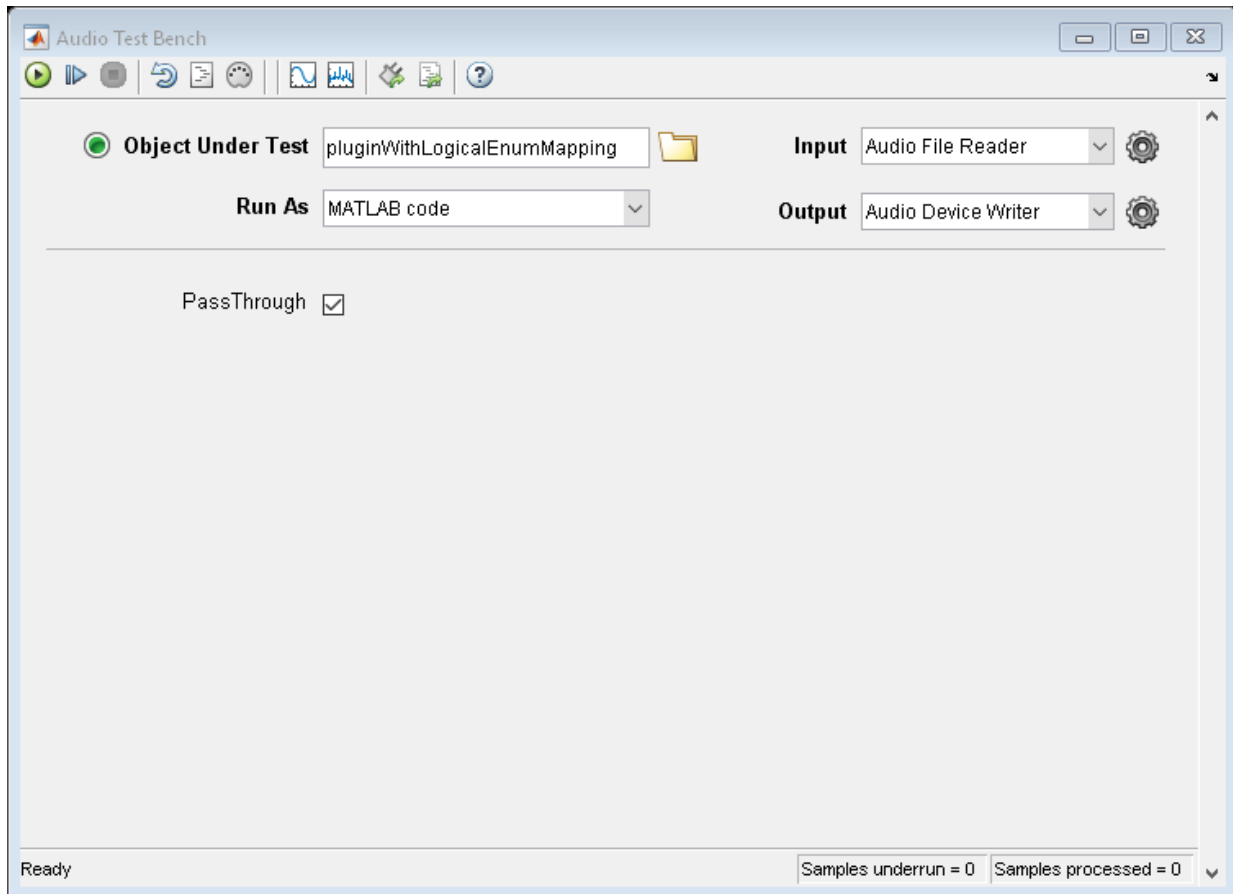
The following class definition uses enumeration parameter mapping to define the relationship between a property and a parameter. You can use the plugin created from this class to block or pass through the audio signal by tuning the `PassThrough` parameter.

```
classdef pluginWithLogicalEnumMapping < audioPlugin
    properties
```

```
        PassThrough = true;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('PassThrough', ...
                'Mapping', {'enum','Block signal','Pass through'}));
    end
    methods
        function out = process(plugin,in)
            if plugin.PassThrough
                out = in;
            else
                out = zeros(size(in));
            end
        end
    end
end
end
```

To run the plugin, save the class definition to a local folder and then call the Audio Test Bench.

```
audioTestBench(pluginWithLogicalEnumMapping)
```



### 'enum' for Enumeration Class Parameter Mapping

The following class definitions comprise a simple example of enumeration parameter mapping for properties defined by an enumeration class. You can specify the operating mode of the plugin created from this class by tuning the Mode parameter.

#### Plugin Class Definition

```
classdef pluginWithEnumMapping < audioPlugin
```

```

properties
    Mode = OperatingMode.boost;
end
properties (Constant)
    PluginInterface = audioPluginInterface(...
        audioPluginParameter('Mode',...
            'Mapping',{ 'enum', '+6 dB', '-6 dB', 'silence', 'white noise' }));
end
methods
    function out = process(plugin,in)
        switch (plugin.Mode)
            case OperatingMode.boost
                out = in * 2;
            case OperatingMode.cut
                out = in / 2;
            case OperatingMode.mute
                out = zeros(size(in));
            case OperatingMode.noise
                out = rand(size(in)) - 0.5;
            otherwise
                out = in;
        end
    end
end
end
end

```

### Enumeration Class Definition

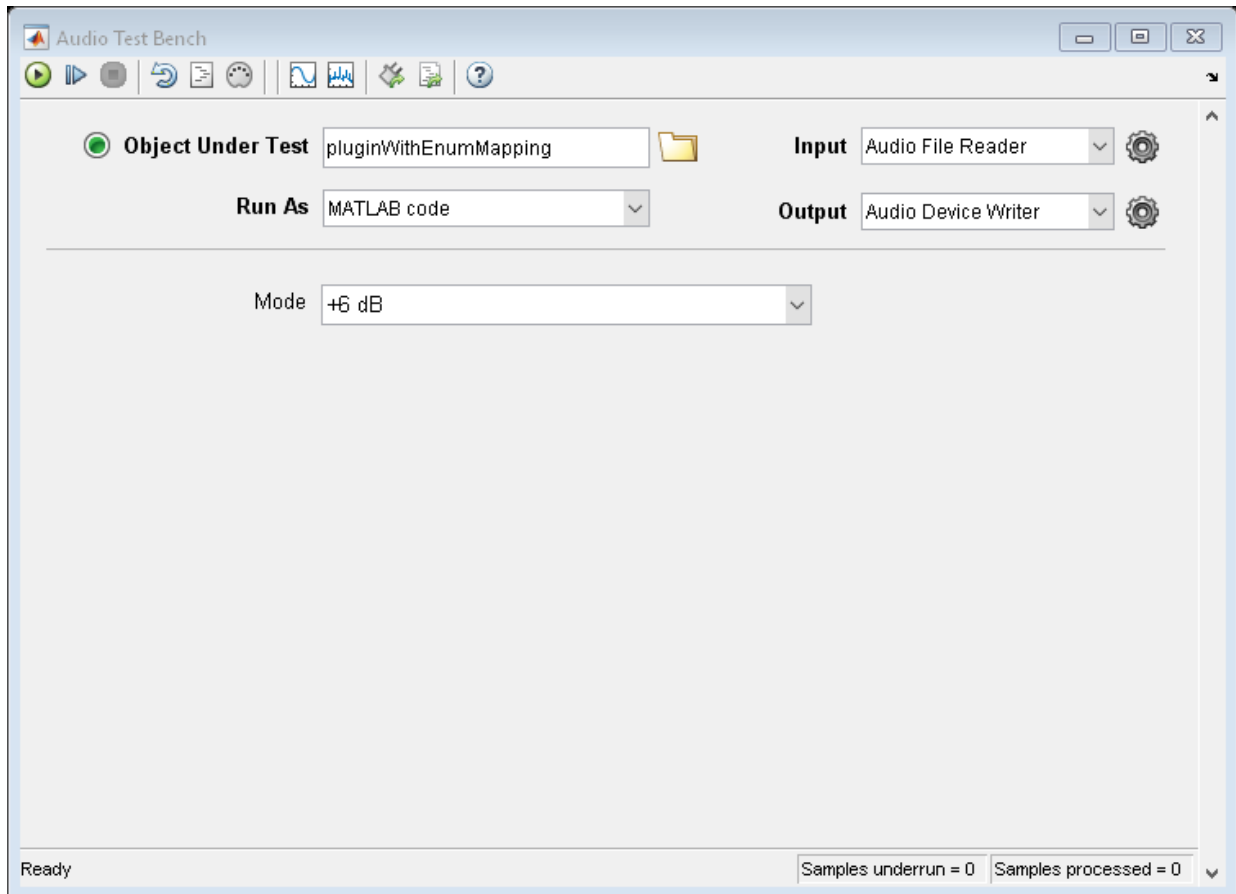
```

classdef OperatingMode < int8
    enumeration
        boost (0)
        cut (1)
        mute (2)
        noise (3)
    end
end

```

To run the plugin, save the plugin and enumeration class definition files to a local folder. Then call the Audio Test Bench on the plugin class.

```
audioTestBench(pluginWithEnumMapping)
```



## Input Arguments

### **propertyName** — Name of audio plugin property

character vector | string

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

Data Types: char | string



## 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 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'`.

### Mappings

#### Mapping — Mapping between property and parameter range

cell array

Mapping between property and parameter range, specified as the 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 character vector 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	<code>'lin'</code> , <code>'log'</code> , <code>'pow'</code> , <code>'int'</code>	<code>{'lin', 0, 1}</code>
logical	<code>'enum'</code>	<code>{'enum', 'off', 'on'}</code>
enumeration class	<code>'enum'</code>	enumeration names

Mapping	Description	Example
<code>'lin'</code>	Specifies a linear relationship with given minimum and maximum values.  $(\text{property value}) = \text{min} + (\text{max} - \text{min}) \times (\text{parameter value})$	<code>{'lin', 0, 24}</code> specifies a linear relationship with a minimum of 0 and maximum of 24.  <b>Example:</b> “Specify Parameter Information” on page 2-406

Mapping	Description	Example
'log'	<p>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.</p> $(property\ value) = \min \times (\max/\min)^{(parameter\ value)}$	<p>{'log', 1, 22050} specifies a logarithmic relationship with a minimum of 1 and a maximum of 22,050.</p> <p><b>Example:</b> "Logarithmic Parameter Mapping" on page 2-410</p>
'pow'	<p>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:</p> $(property\ value) = \min + (\max - \min) \times (parameter\ value)^{exp}$	<p>{'pow', 1/3, -140, 12} specifies a power law relationship with an exponent of 1/3, a minimum of -140, and a maximum of 12.</p> <p><b>Example:</b> "Power Parameter Mapping" on page 2-408</p>
'int'	<p>Quantizes the control position and maps it to the range of consecutive integers with given minimum and maximum values.</p> $(property\ value) = \text{floor}(0.5 + \min + (\max - \min) \times (parameter\ value))$	<p>{'int', 0, 3} specifies a linear, quantized relationship with a minimum of 0 and maximum of 3. The property value is mapped as an integer in the range [0, 3].</p> <p><b>Example:</b> "Integer Parameter Mapping" on page 2-406</p>
'enum' (logical )	<p>Optionally provides character vectors for display on the plugin dialog box.</p>	<p>{'enum', 'Block signal', 'Passthrough'} specifies the character vector 'Block signal' if the parameter value is false and 'Passthrough' if the parameter value is true.</p> <p><b>Example:</b> "Enumeration for Logical Properties Parameter Mapping" on page 2-412</p>

Mapping	Description	Example
'enum' (enumeration class)	Optionally provides character vectors for the members of the enumeration class.	<pre>{'enum', '+6 dB', '-6 dB', 'silence', 'white noise'}</pre> specifies the character vectors '+6 dB', '-6 dB', 'silence', and 'white noise'.  <b>Example:</b> “'enum' for Enumeration Class Parameter Mapping” on page 2-414

### Graphical User Interface

#### Layout — Grid cells occupied by parameter control

[row, column] (single-cell specification) | [upper, left; lower, right] (multi-cell specification)

Grid cells occupied by parameter control, specified as a comma-separated pair consisting of 'Layout' and a two-element row vector or 2-by-2 matrix. To use a single cell, specify [row, column] of the cell. To span multiple cells, specify the upper left and lower right cells as [upper, left; lower, right].

Example: 'Layout', [2,3]

Example: 'Layout', [2,3;3,6]

#### Dependencies

To enable this name-value pair, pass an audioPluginGridLayout object to audioPluginInterface.

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

#### DisplayName — Display name of parameter

associated property name (default) | character vector | string

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

Data Types: char | string

### **DisplayNameLocation — Location of display name**

'left' | 'right' | 'above' | 'below' | 'none'

Location of `DisplayName`, specified as a comma-separated pair consisting of `'DisplayNameLocation'` and `'left'`, `'right'`, `'above'`, `'below'`, or `'none'`. The location of the display name is defined relative to the `Layout`:

- 1 `'left'` -- The display name is located in the column to the left of `Layout` and spans the same rows as `Layout`.
- 2 `'right'` -- The display name is located in the column to the right of `Layout` and spans the same rows as `Layout`.
- 3 `'above'` -- The display name is located in the row above `Layout` and spans the same columns as `Layout`.
- 4 `'below'` -- The display name is located in the row below `Layout` and spans the same columns as `Layout`.
- 5 `'none'` -- `DisplayName` is suppressed.

Example: `'DisplayNameLocation','left'`

#### **Dependencies**

To enable this name-value pair, pass an `audioPluginGridLayout` object to `audioPluginInterface`.

Data Types: `char` | `string`

### **Label — Display label for parameter value units**

'' (default) | character vector | string

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

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

Data Types: `char` | `string`

### **Style — Visual control for plugin parameter**

'hslider' | 'vslider' | 'rotaryknob' | 'checkbox' | 'vrocker' | 'vtoggle' | 'dropdown'

Visual control for plugin parameter, specified as a comma-separated pair consisting of `'Style'` and a string or character vector:

Style	Description
'hslider'	Horizontal slider
'vslider'	Vertical slider
'rotaryknob'	Rotary knob
'checkbox'	Check box
'vrocker'	Vertical rocker switch
'vtoggle'	Vertical toggle switch
'dropdown'	Dropdown

Default and valid styles depends on the plugin parameter Mapping and corresponding property class:

Mapping	Property Class	Default Style	Additional
lin	single	hslider	vslider
log	double		rotaryknob
pow			
int			
enum	logical	checkbox	dropdown vrocker vtoggle
enum	enumeration with 2 values	vrocker	dropdown vtoggle
enum	enumeration	dropdown	

### Dependencies

To enable this name-value pair, pass an `audioPluginGridLayout` object to `audioPluginInterface`.

Data Types: `char` | `string`

### **Filmstrip — Name of PNG, GIF, or JPG graphics file**

character vector | string

Name of PNG, GIF, or JPG graphics file, specified as the comma-separated pair consisting of 'Filmstrip' and a character vector or string. The graphics file contains a sequence of images of controls.

Filmstrips enable you to replace default control graphics with your own custom images. Filmstrips support all control `Style` values except for dropdowns. A filmstrip is a single image created by concatenating smaller images called frames. Each frame is an image of a control in a particular position. For example, a filmstrip for a switch contains two frames: one depicting the "off" state and another depicting the "on" state. Frames can be concatenated vertically or horizontally. Suppose that the switch frames are 50 pixels wide by 100 pixels high. Then vertical concatenation produces a 50-by-200 pixel filmstrip image, with the top frame used for the switch "off" state. Horizontal concatenation produces a 100-by-100 pixel image, with the left frame used for the switch "off" state. Filmstrips for sliders and knobs typically contain many more frames. The top/left frame corresponds to the minimum control position, and the bottom/right frame corresponds to the maximum control position. The relative control position determines which frame is displayed for a given parameter value.

#### **Dependencies**

To enable this name-value pair, pass an `audioPluginGridLayout` object to `audioPluginInterface` and specify 'FilmstripFrameSize'.

Data Types: `char` | `string`

### **FilmstripFrameSize — Size of individual frames (pixels)**

[width, height]

Size of individual frames in the film strip in pixels, specified as the comma-separated pair consisting of 'FilmstripFrameSize' and a two-element row vector of integers that specify [width, height].

#### **Dependencies**

To enable this name-value pair, pass an `audioPluginGridLayout` object to `audioPluginInterface` and specify a 'Filmstrip'.

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

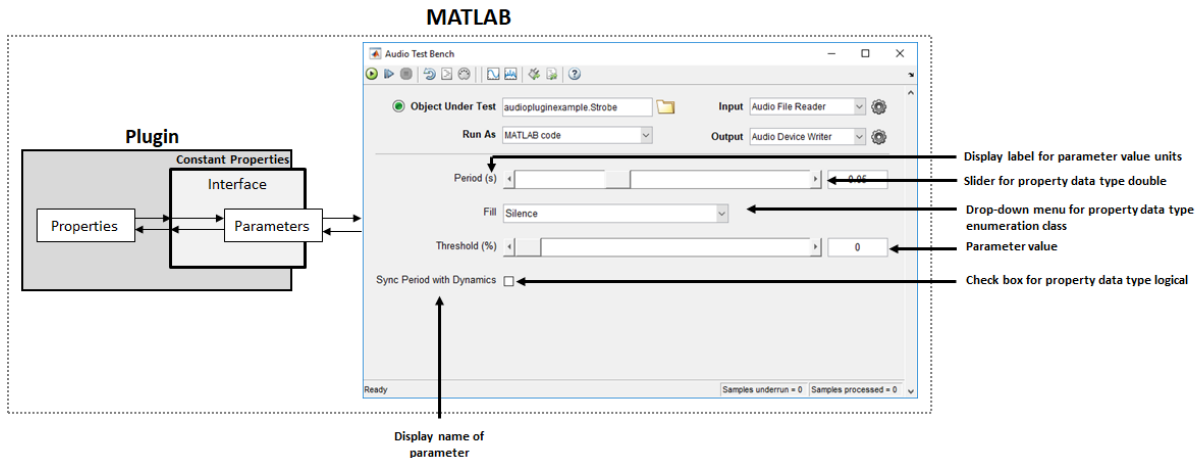
To learn how to design a graphic user interface, see “Design User Interface for Audio Plugin”.

## More About

### Implementation of Audio Plugin Parameters

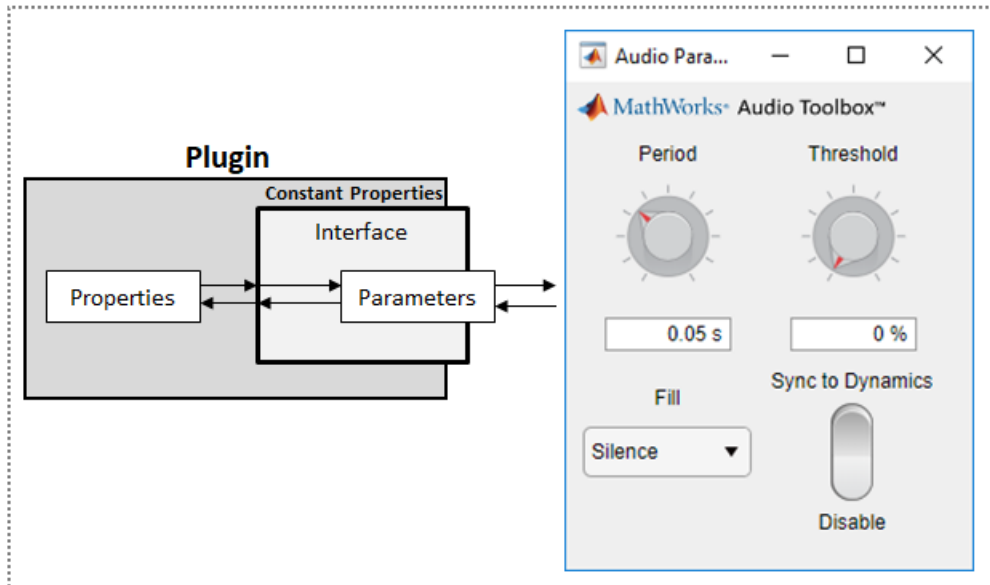
Audio plugin parameters are visible and tunable in both the MATLAB and digital audio workstation (DAW) environments.

**MATLAB Environment Using Audio Test Bench.** Use **Audio Test Bench** to interact with plugin parameters in the MATLAB environment in a complete GUI-based workflow. Using the **Audio Test Bench**, you can specify audio input and output, analyze your plugin using time- and frequency-domain scopes, connect to MIDI controls, and validate and generate your plugin.



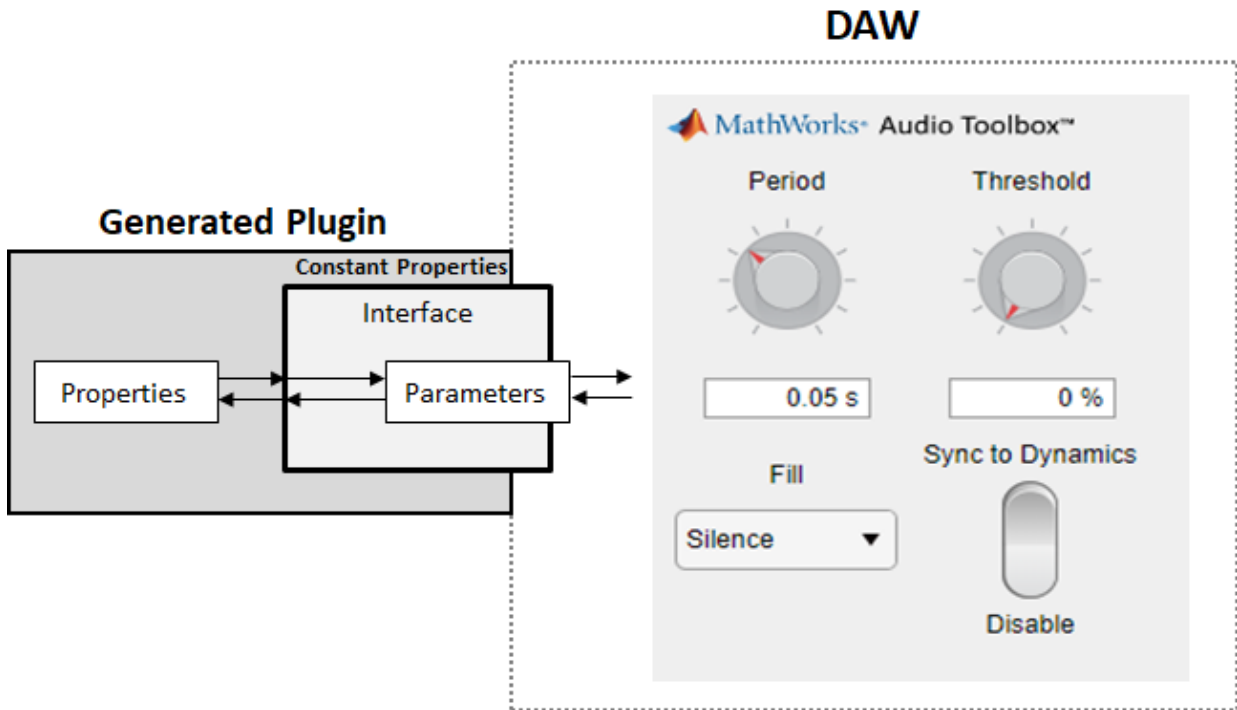
**MATLAB Environment Using parameterTuner.** Use `parameterTuner` to interact with plugin parameters in the MATLAB environment while developing, analyzing, or using your plugin in a programmatic workflow. The `parameterTuner` honors the graphical user interface you specified in `audioPluginParameter`, `audioPluginGridLayout`, and `audioPluginInterface` (except for filmstrips).

## MATLAB



**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.





## Extended Capabilities

### C/C++ Code Generation

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

### See Also

**Audio Test Bench** | `audioPlugin` | `audioPluginInterface` | `audioPluginSource` | `generateAudioPlugin` | `parameterTuner` | `validateAudioPlugin`

### Topics

“Audio Plugins in MATLAB”

“Export a MATLAB Plugin to a DAW”  
“Audio Plugin Example Gallery”

**Introduced in R2016a**

# configureMIDI

Configure MIDI connections between audio object and MIDI controller

## Syntax

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

## Description

`configureMIDI(audioObject)` opens a MIDI configuration user interface (UI). Use the UI to synchronize parameters of the plugin, `audioObject`, 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. Use `midid` to get the device name corresponding to your MIDI device.

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

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

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

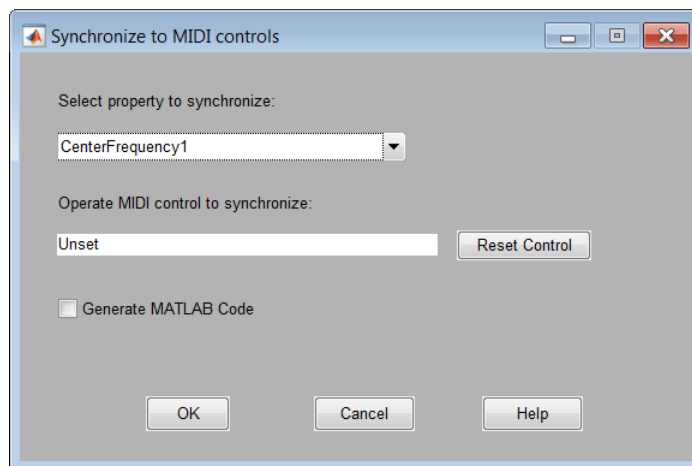
## Examples

### Synchronize Plugin Parameters to MIDI Controls

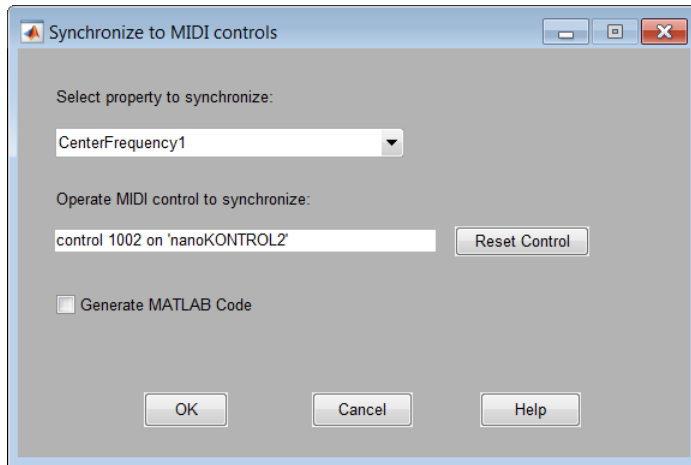
- 1 Open the MIDI configuration UI for a parametric equalizer plugin object.

```
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**.

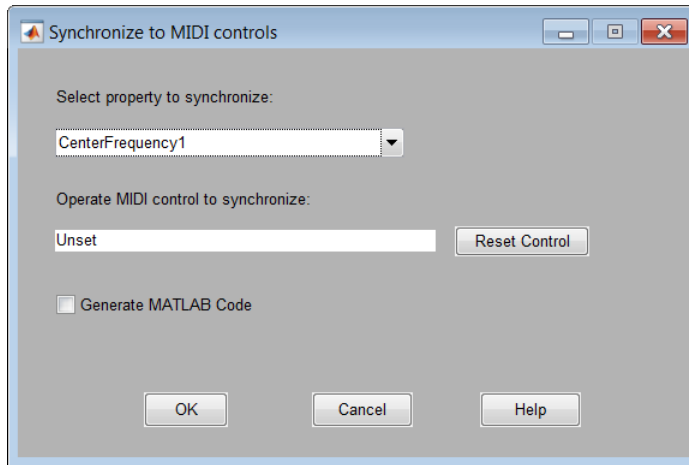
The specified MIDI controls and properties are now synchronized.

### Generate MATLAB Code from `configureMIDI` UI

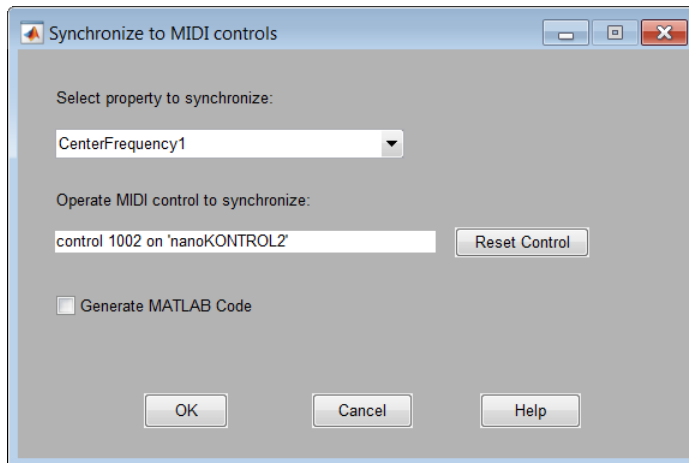
Generate MATLAB code corresponding to the MIDI configuration developed using the `configureMIDI` UI. You can embed the MATLAB code in your simulation so that you do not need to reopen the UI to restore your chosen MIDI connections.

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

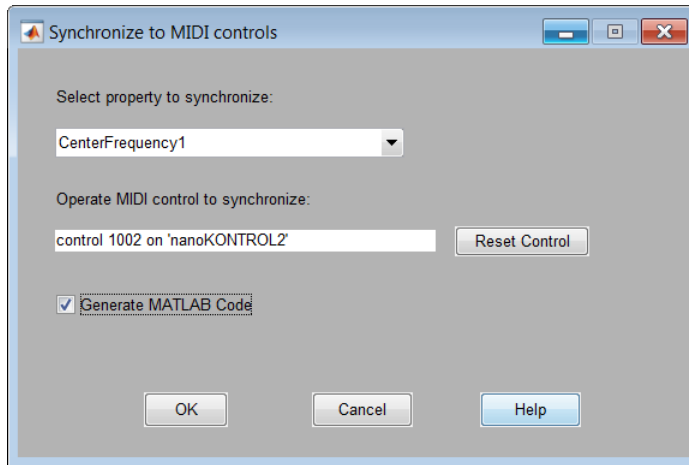
```
parametricEQPlugin = audiopluginexample.ParametricEqualizer;
configureMIDI(parametricEQPlugin);
```
- 2 In the UI, select a property to synchronize with your default MIDI device.



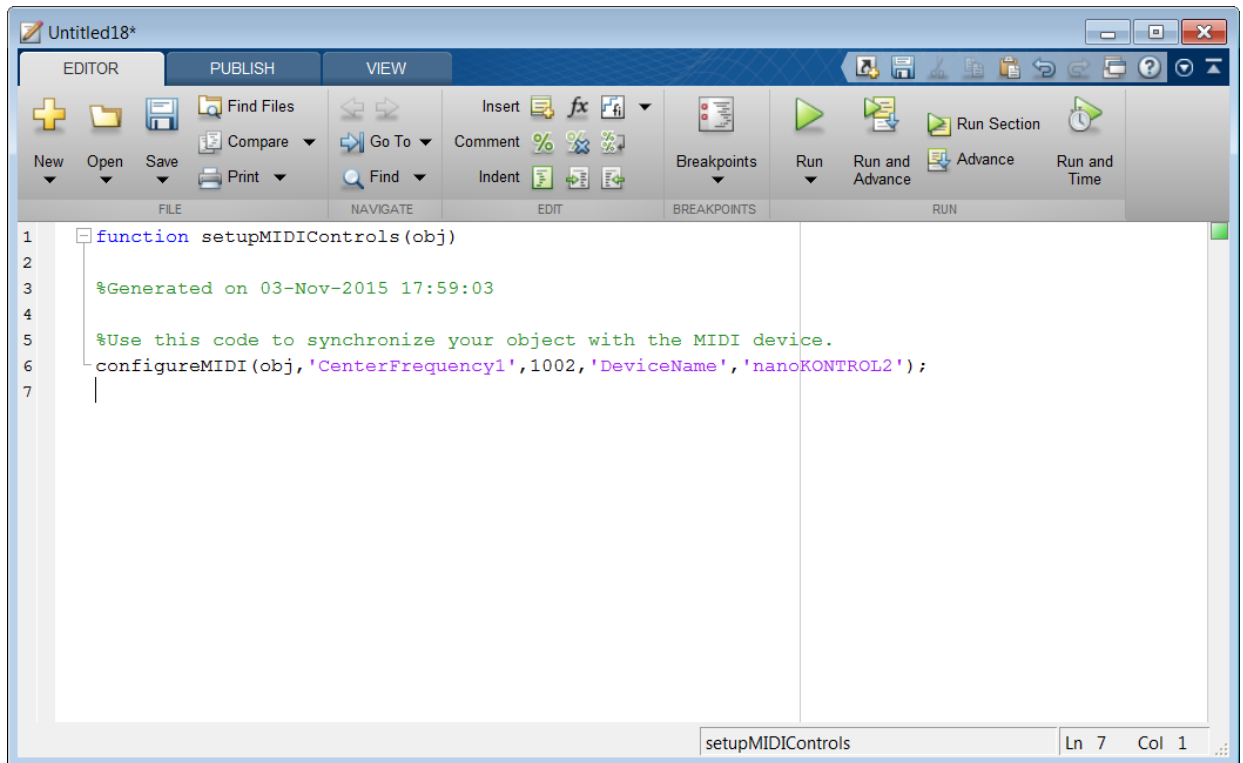
- 3 On your MIDI device, operate the control that you want to synchronize to the selected plugin property. The control appears in the **Operate MIDI control to synchronize** box.



- 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

### **audioObject** — Audio object

object

Audio plugin or compatible System object, specified as an object that inherits from the `audioPlugin` class or an object of a compatible Audio Toolbox System object.

### **propertyName** — Name of object property

character vector

Name of the object property, specified as a character vector. Enter the property name exactly as it is defined in the property section of your audio plugin or Audio Toolbox System object.

### **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

character vector

MIDI device name, assigned by the device manufacturer or host operating system, specified as a character vector. 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 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 object, uncomment the `drawnow limitrate` command from this code:

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

configureMIDI(compressor, 'Threshold');

while ~isDone(fileReader)
    input = fileReader();
    output = dRC(input);
    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.

## See Also

### Classes

[audioPlugin](#) | [audioPluginSource](#)

### Functions

[disconnectMIDI](#) | [getMIDIConnections](#) | [midicallback](#) | [midicontrols](#) | [midiid](#) | [midiread](#) | [midisync](#)

## Topics

“MIDI Control for Audio Plugins”

“MIDI Control Surface Interface”

**Introduced in R2016a**

# designParamEQ

Design parametric equalizer

## Syntax

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

## 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).

`[B,A] = designParamEQ( ____,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

## 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 = [ , ...
     ];
```

```
gain = [ , ...
```

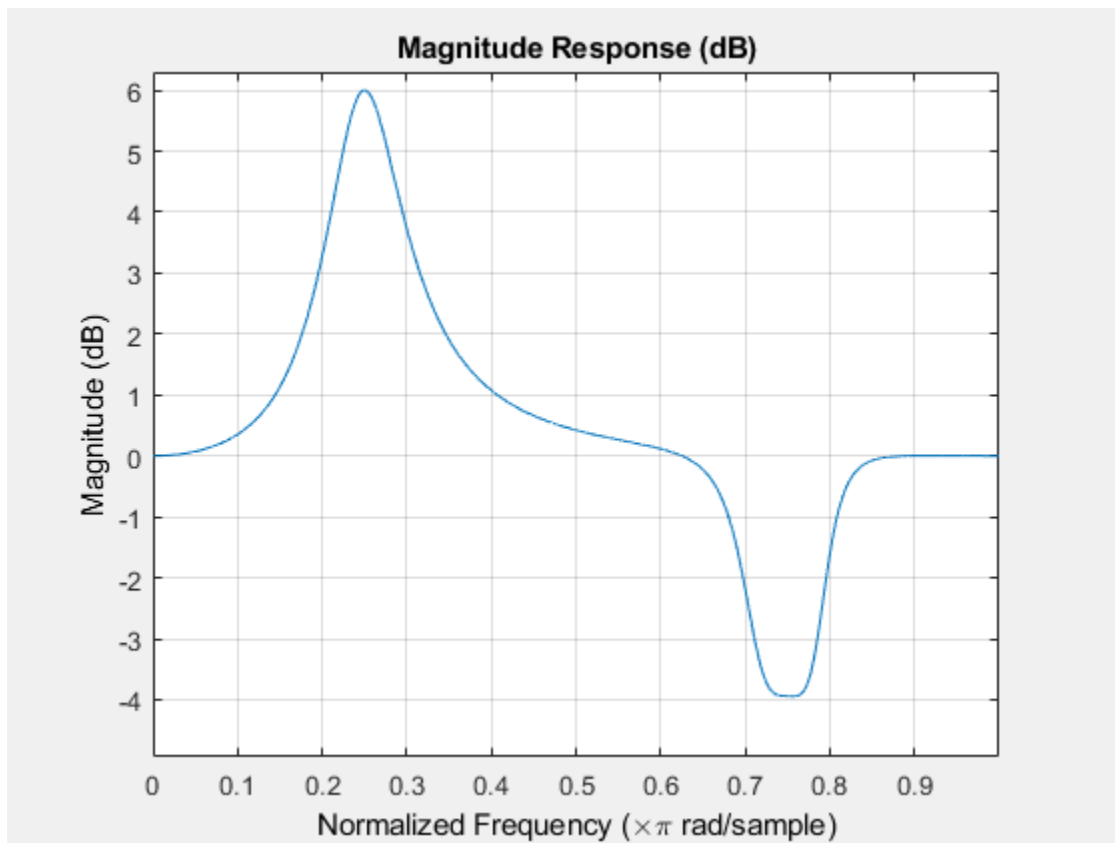
```
];  
centerFreq = [, ...  
              ];  
bandwidth = [, ...  
             ];
```

Generate the filter coefficients using the specified parameters.

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

Visualize your filter design.

```
fvtool([B,A]);
```



## 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);

sampleRate = fileReader.SampleRate;

deviceWriter = audioDeviceWriter("SampleRate", sampleRate);

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

Play the audio signal through your device.

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

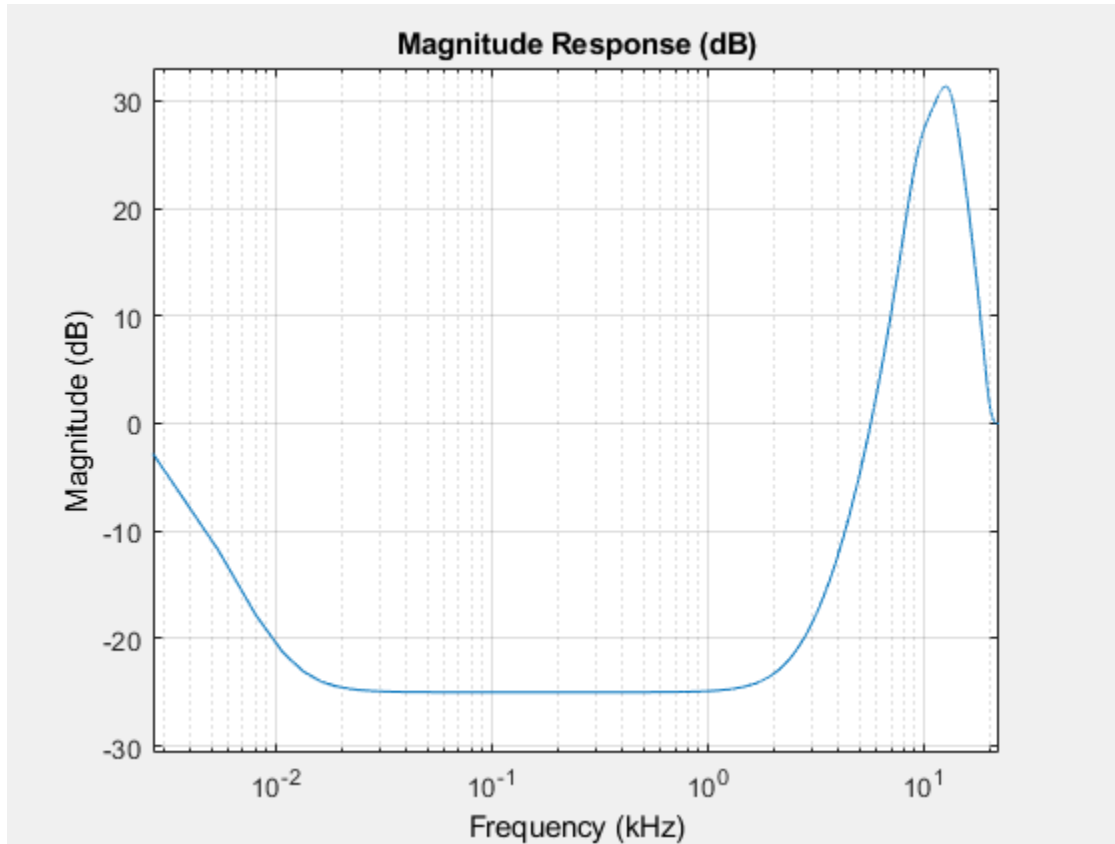
Design an SOS parametric equalizer suitable for use with `dsp.BiquadFilter`.

```
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. Call `designParamEQ` with the same design specifications. Specify the output orientation as "row" so that it is suitable for use with `fvtool`.

```
[Bvisualize, Avisualize] = designParamEQ(N, gain, centerFreq, bandwidth, "Orientation", "row");
fvtool([Bvisualize, Avisualize], ...
```

```
"Fs",fileReader.SampleRate, ...  
"FrequencyScale","Log");
```



Create a biquad filter.

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

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

```
scope = dsp.SpectrumAnalyzer( ...  
    "SampleRate",sampleRate, ...
```



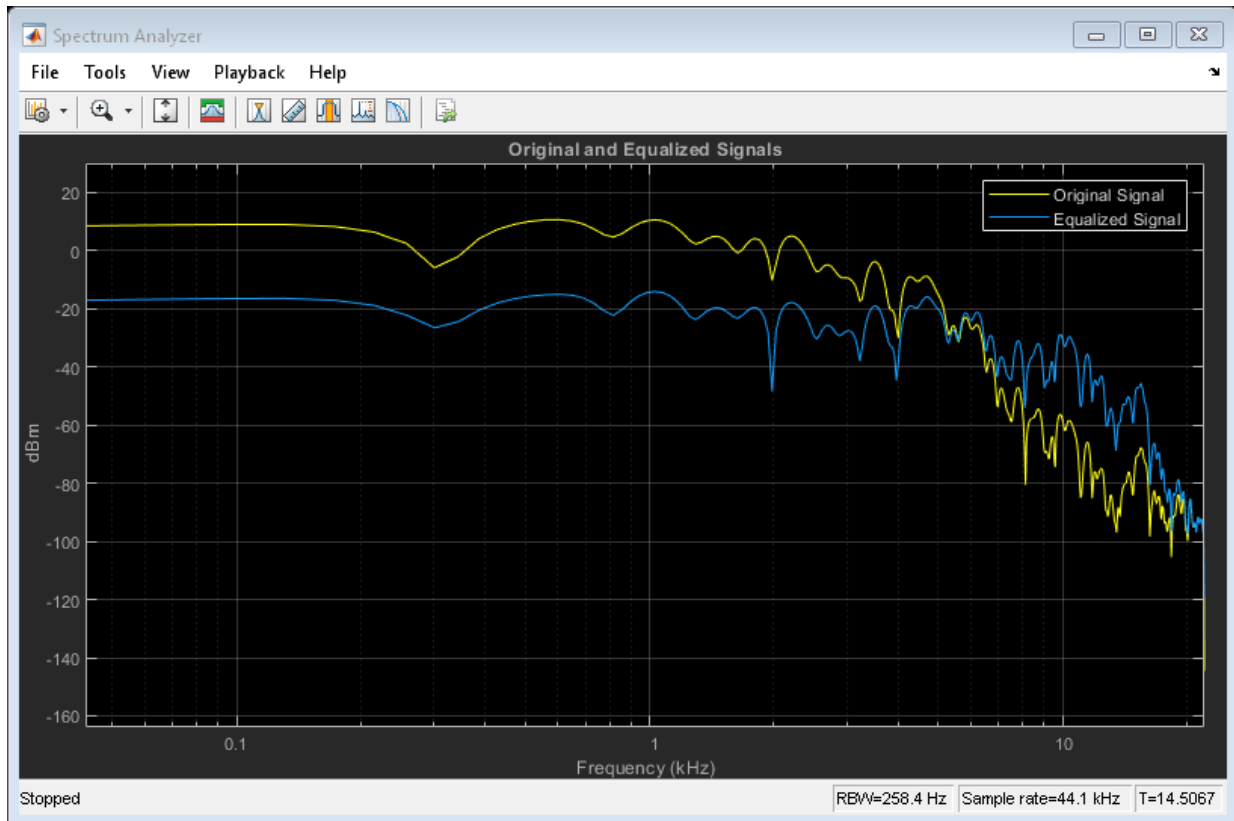
```
"PlotAsTwoSidedSpectrum",false, ...  
"FrequencyScale","Log", ...  
"FrequencyResolutionMethod","WindowLength", ...  
"WindowLength",frameSize, ...  
"Title","Original and Equalized Signals", ...  
"ShowLegend",true, ...  
"ChannelNames",{ 'Original Signal', 'Equalized Signal' });
```

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

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

As a best practice, release your objects once done.

```
release(deviceWriter)  
release(fileReader)  
release(scope)
```



### 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);

sampleRate = fileReader.SampleRate;

deviceWriter = audioDeviceWriter( ...
    "SampleRate", sampleRate);

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

```

Play the audio signal through your device.

```

count = 0;
while count < 2500
    x = fileReader();
    deviceWriter(x);
    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";

```

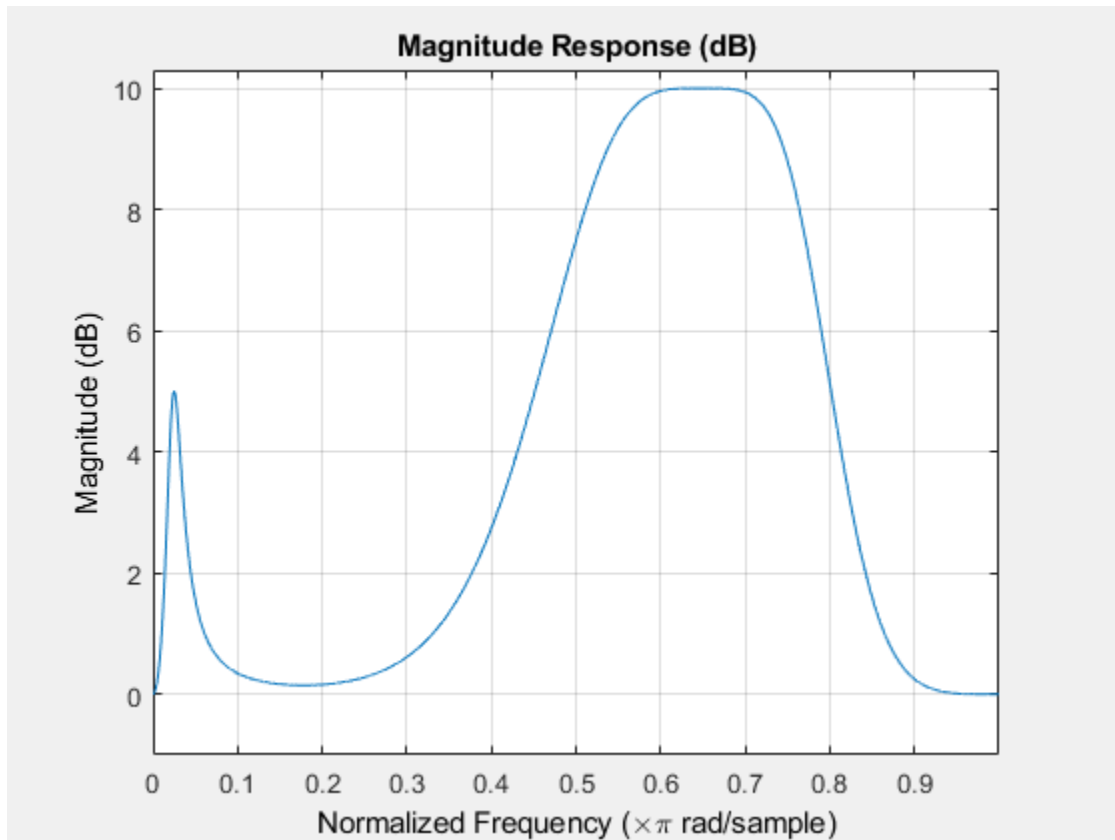
```
[B, A] = designParamEQ(N, gain, centerFreq, bandwidth, mode, "Orientation", "row");
```

Construct FOS IIR filters.

```
myFilter = dsp.FourthOrderSectionFilter(B, A);
```

Visualize the frequency response of your parametric equalizer.

```
fvtool(myFilter)
```



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

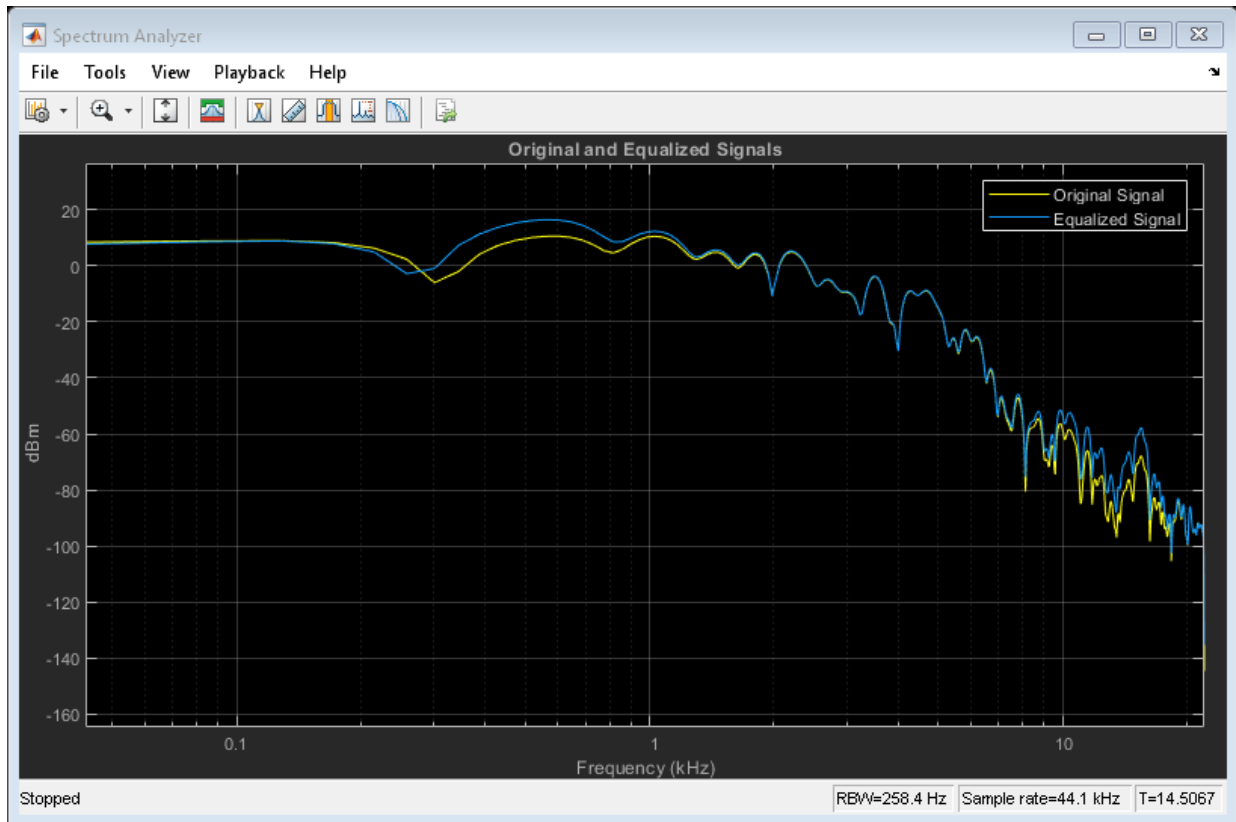
```
scope = dsp.SpectrumAnalyzer( ...
    "SampleRate",sampleRate, ...
    "PlotAsTwoSidedSpectrum",false, ...
    "FrequencyScale","Log", ...
    "FrequencyResolutionMethod","WindowLength", ...
    "WindowLength",frameSize, ...
    "Title","Original and Equalized Signals", ...
    "ShowLegend",true, ...
    "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  
    x = fileReader();  
    y = myFilter(x);  
  
    scope([x(:,1),y(:,1)]);  
  
    deviceWriter(y);  
  
    count = count + 1;  
end
```

As a best practice, release your objects once done.

```
release(fileReader)  
release(deviceWriter)  
release(scope)
```



## 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.

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

**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.

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

**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 ( $\pi$  rad/sample). If `centerFreq` is specified as a row vector, separate equalizers are designed for each element of `centerFreq`.

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

**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 ( $\pi$  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

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

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

### **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.

Data Types: char | string

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'Orientation', "row"

### **Orientation — Orientation of returned filter coefficients**

"column" (default) | "row"

Orientation of returned filter coefficients, specified as the comma-separated pair consisting of 'Orientation' and "column" or "row":

- Set 'Orientation' to "row" for interoperability with **FVTool**, `dsp.DynamicFilterVisualizer`, and `dsp.FourthOrderSectionFilter`.
- Set 'Orientation' to "column" for interoperability with `dsp.BiquadFilter`.

Data Types: char | string

## **Output Arguments**

### **B — Numerator filter coefficients**

matrix

Numerator filter coefficients, returned as a matrix. The size and interpretation of `B` depends on the `Orientation` and `mode`:



- If 'Orientation' is set to "column" and mode is set to "sos", then B is returned as an  $L$ -by-3 matrix. Each column corresponds to the numerator coefficients of your cascaded second-order sections.
- If 'Orientation' is set to "column" and mode is set to "fos", then B is returned as an  $L$ -by-5 matrix. Each column corresponds to the numerator coefficients of your cascaded fourth-order sections.
- If 'Orientation' is set to "row" and mode is set to "sos", then B is returned as a 3-by- $L$  matrix. Each row corresponds to the numerator coefficients of your cascaded second-order sections.
- If 'Orientation' is set to "row" and mode is set to "fos", then B is returned as a 5-by- $L$  matrix. Each row corresponds to the numerator coefficients of your cascaded fourth-order sections.

### **A – Denominator filter coefficients**

matrix

Denominator filter coefficients, returned as a matrix. The size and interpretation of A depends on the Orientation and mode:

- If 'Orientation' is set to "column" and mode is set to "sos", then A is returned as an  $L$ -by-2 matrix. Each column corresponds to the denominator coefficients of your cascaded second-order sections. A does not include the leading unity coefficients.
- If 'Orientation' is set to "column" and mode is set to "fos", then A is returned as an  $L$ -by-4 matrix. Each column corresponds to the denominator coefficients of your cascaded fourth-order sections. A does not include the leading unity coefficients.
- If 'Orientation' is set to "row" and mode is set to "sos", then A is returned as a 3-by- $L$  matrix. Each row corresponds to the denominator coefficients of your cascaded second-order sections.
- If 'Orientation' is set to "row" and mode is set to "fos", then A is returned as a 5-by- $L$  matrix. Each row corresponds to the denominator coefficients of your cascaded fourth-order sections.

## **References**

- [1] Orfanidis, Sophocles J. "High-Order Digital Parametric Equalizer Design." *Journal of the Audio Engineering Society*. Vol. 53, November 2005, pp. 1026-1046.

## **Extended Capabilities**

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

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

### **See Also**

`designShelvingEQ` | `designVarSlopeFilter` | `dsp.BiquadFilter` | `multibandParametricEQ`

### **Topics**

“Parametric Equalizer Design”  
“Equalization”

**Introduced in R2016a**

# designShelvingEQ

Design shelving equalizer

## Syntax

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

## 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.

`[B,A] = designShelvingEQ( ____,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

## 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;
```

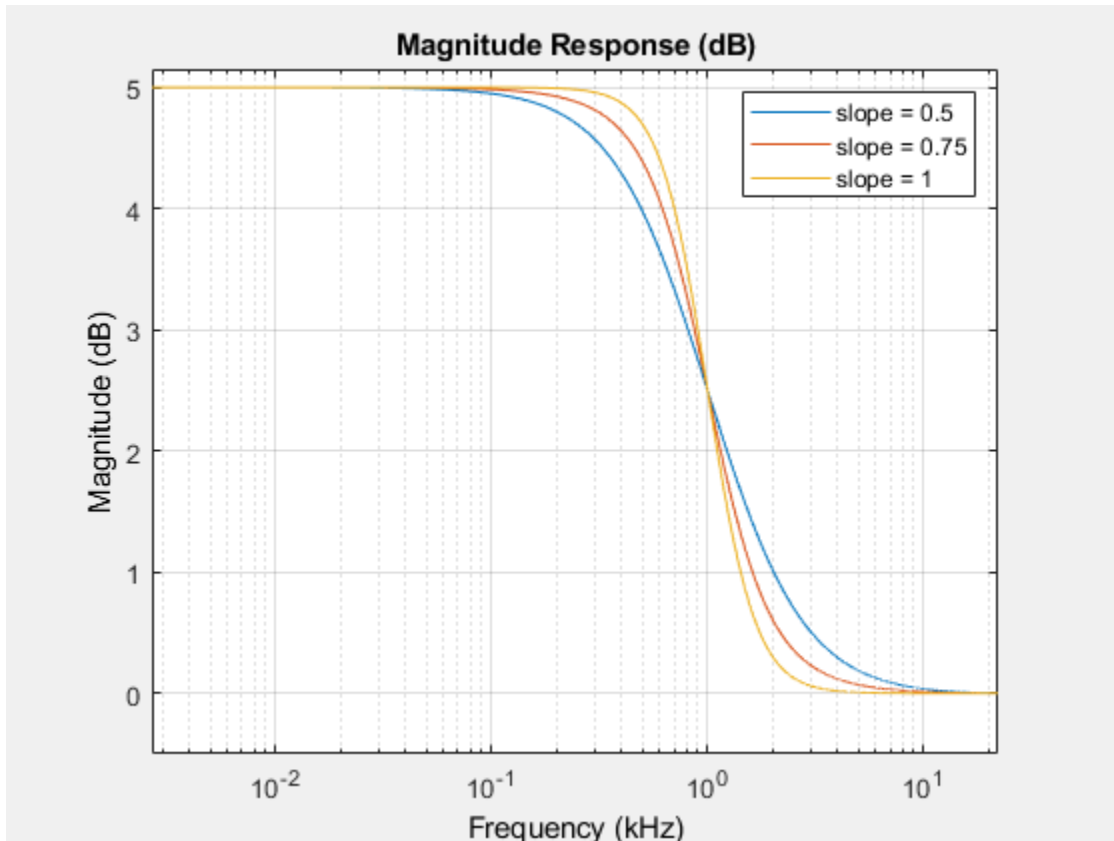
```
Fc = 1000/(Fs/2);
```

Design the filter coefficients using the specified parameters.

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

Visualize your filter design.

```
fvtool( ...  
    dsp.BiquadFilter([B1,A1]), ...  
    dsp.BiquadFilter([B2,A2]), ...  
    dsp.BiquadFilter([B3,A3]), ...  
    "Fs",Fs, ...  
    "FrequencyScale","Log");  
  
legend("slope = 0.5", ...  
    "slope = 0.75", ...  
    "slope = 1");
```



### Filter Audio Using Low-Shelf Equalizer

Create 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);
```

```
sampleRate = fileReader.SampleRate;

deviceWriter = audioDeviceWriter( ...
    "SampleRate",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)
```

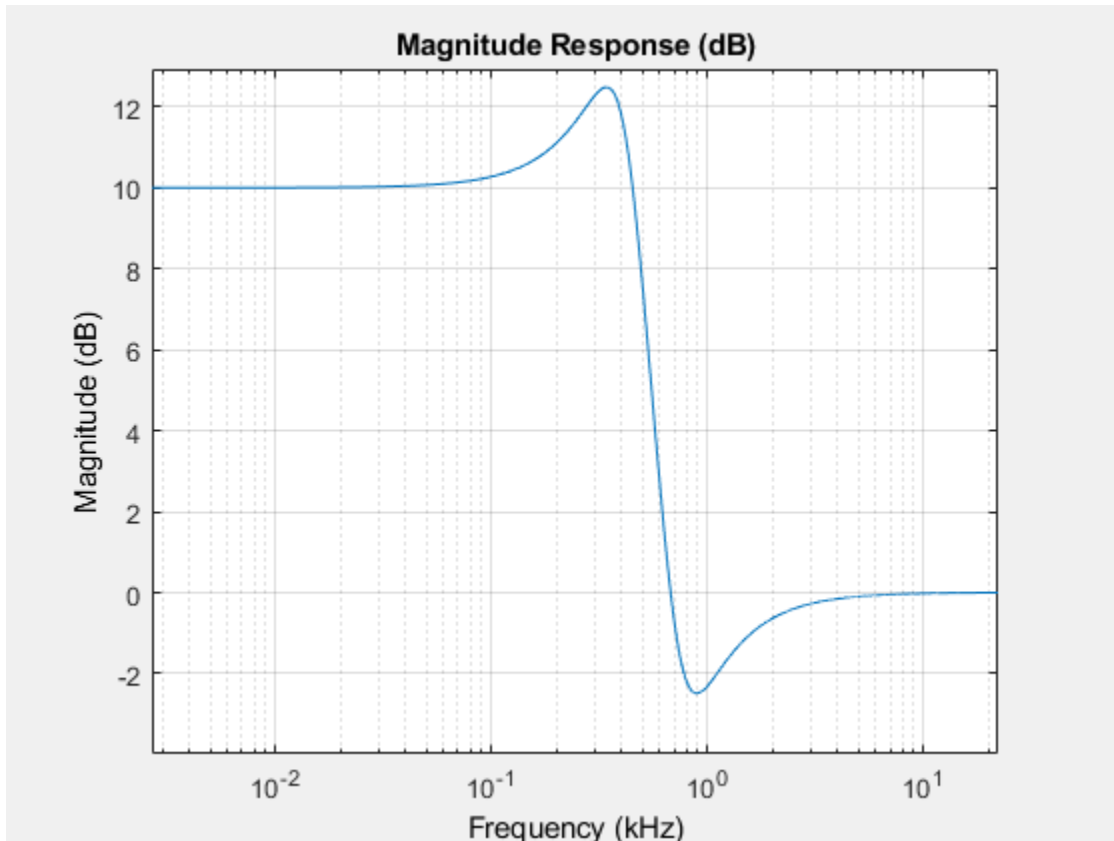
Design a second-order sections (SOS) low-shelf equalizer.

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

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

Visualize your shelving filter design.

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



Create a biquad filter object.

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

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

```
scope = dsp.SpectrumAnalyzer( ...  
    "SampleRate", sampleRate, ...  
    "PlotAsTwoSidedSpectrum", false, ...  
    "FrequencyScale", "Log", ...  
    "FrequencyResolutionMethod", "WindowLength", ...
```

```
"WindowLength",frameSize, ...  
"Title","Original and Equalized Signal", ...  
"ShowLegend",true, ...  
"ChannelNames",{ 'Original Signal', 'Equalized Signal' });
```

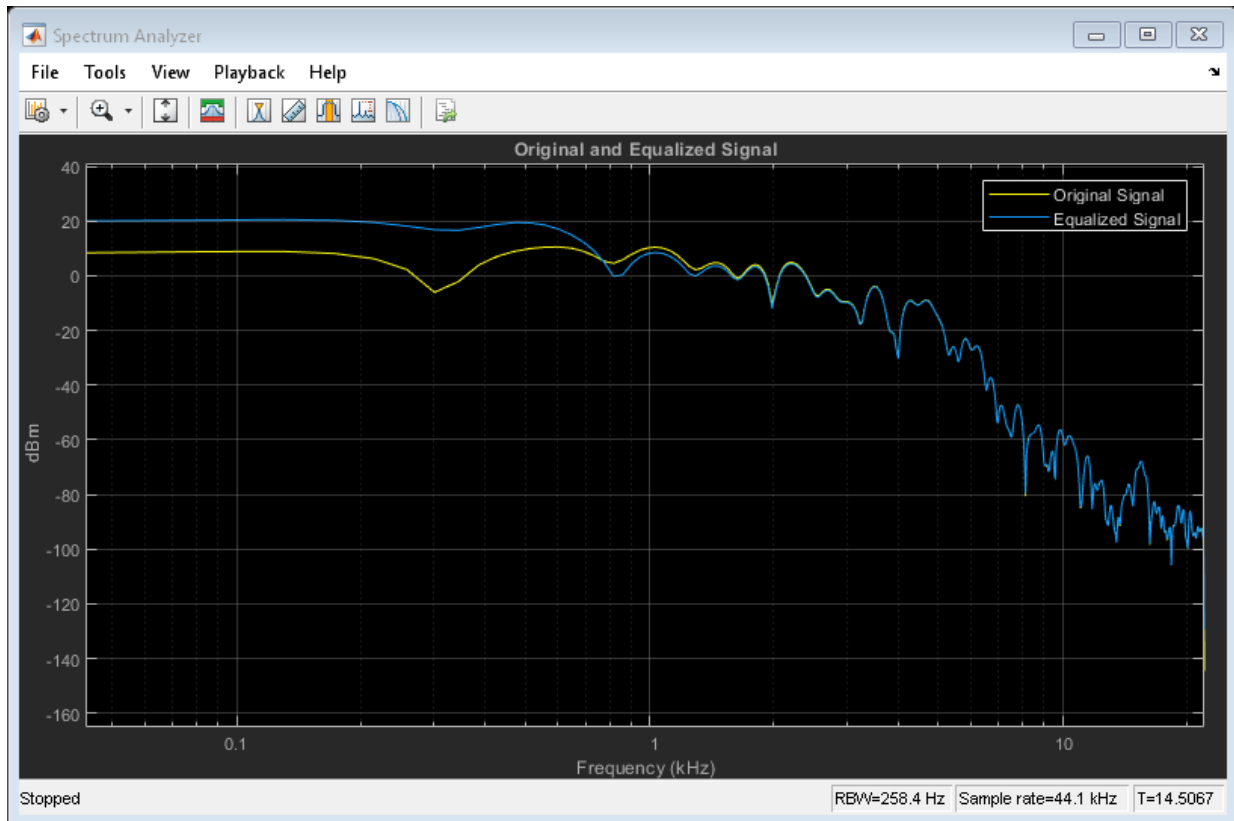
Play the equalized audio signal and visualize the original and equalized spectrums.

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

As a best practice, release your objects once done.

```
release(fileReader)  
release(deviceWriter)  
release(scope)
```





### 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","Orientation","row");
```

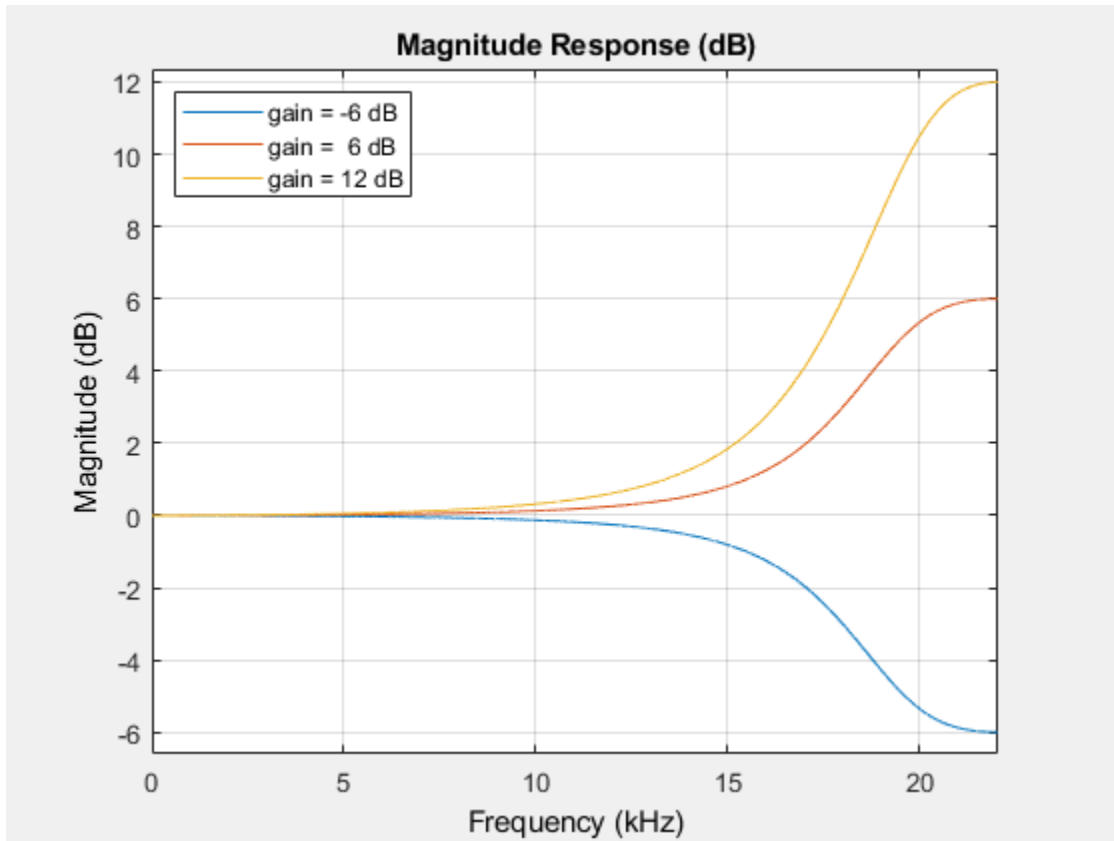
```
[B2,A2] = designShelvingEQ(gain2,slope,Fc,"hi","Orientation","row");
```

```
[B3,A3] = designShelvingEQ(gain3,slope,Fc,"hi","Orientation","row");
```

Visualize your filter design.

```
fvtool([B1,A1;[1 0 0 1 0 0]], ...  
       [B2,A2;[1 0 0 1 0 0]], ...  
       [B3,A3;[1 0 0 1 0 0]], ...  
       "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.

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

### **slope** — Slope coefficient

real scalar in the range 0 to 5

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

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

### **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 ( $\pi$  rad/sample).

Normalized cutoff frequency is implemented as half the shelving filter gain, or `gain/2` dB.

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

### **type — Filter type**

'lo' (default) | 'hi'

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

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

Data Types: `char` | `string`

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Orientation', "row"`

### **Orientation — Orientation of returned filter coefficients**

"column" (default) | "row"

Orientation of returned filter coefficients, specified as the comma-separated pair consisting of 'Orientation' and "column" or "row":

- Set 'Orientation' to "row" for interoperability with **FVTool**, `dsp.DynamicFilterVisualizer`, and `dsp.FourthOrderSectionFilter`.

- Set 'Orientation' to "column" for interoperability with `dsp.BiquadFilter`.

Data Types: char | string

## Output Arguments

### **B — Numerator filter coefficients**

three-element column vector | three-element row vector

Numerator filter coefficients, returned as a vector. The size and interpretation of B depends on the Orientation:

- If 'Orientation' is set to "column", then B is returned as a three-element column vector.
- If 'Orientation' is set to "row", then B is returned as a three-element row vector.

.

### **A — Denominator filter coefficients**

two-element column vector | three-element row vector

Denominator filter coefficients of the designed second-order IIR filter, returned as a vector. The size and interpretation of A depends on the Orientation:

- If 'Orientation' is set to "column", then A is returned as a two-element column vector. A does not include the leading unity coefficient.
- If 'Orientation' is set to "row", then A is returned as a three-element row vector.

## References

- [1] Bristow-Johnson, Robert. "Cookbook Formulae for Audio EQ Biquad Filter Coefficients." Accessed March 02, 2016. <http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt>.

## **Extended Capabilities**

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

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

### **See Also**

`designParamEQ` | `designVarSlopeFilter` | `multibandParametricEQ`

### **Topics**

“Parametric Equalizer Design”

“Equalization”

**Introduced in R2016a**

# designVarSlopeFilter

Design variable slope lowpass or highpass IIR filter

## Syntax

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

## 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.

`[B,A] = designVarSlopeFilter( ___,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

## 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 = 10e3/(Fs/2);  
Fc2 = 16e3/(Fs/2);
```

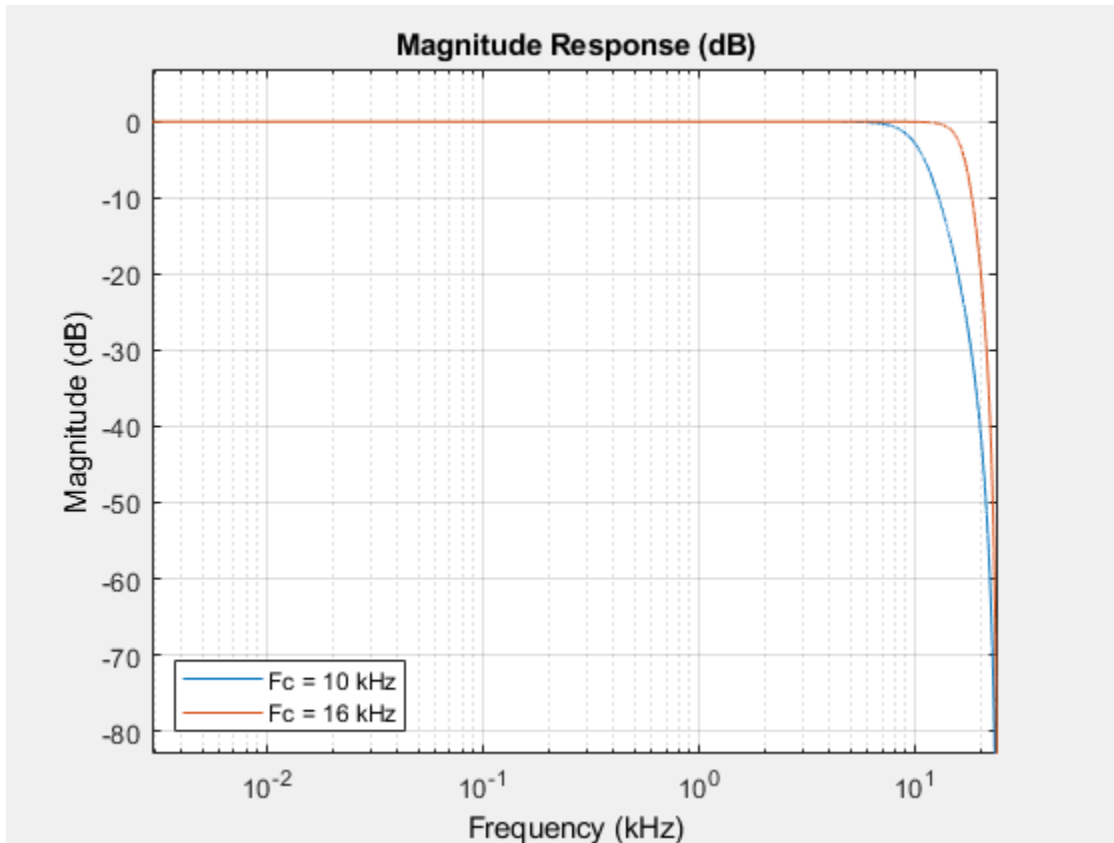
Design the filter coefficients using the specified parameters.

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

Visualize your filter design.

```
fvtool([B1,A1],[B2,A2],"Fs",Fs,"FrequencyScale","Log");  
legend("Fc = 10 kHz", ...  
       "Fc = 16 kHz", ...  
       "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.

Create 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);  
  
sampleRate = fileReader.SampleRate;  
  
deviceWriter = audioDeviceWriter( ...  
    "SampleRate", sampleRate);  
  
setup(fileReader)  
setup(deviceWriter, ones(frameSize, 2))
```

Play the audio signal through your device.

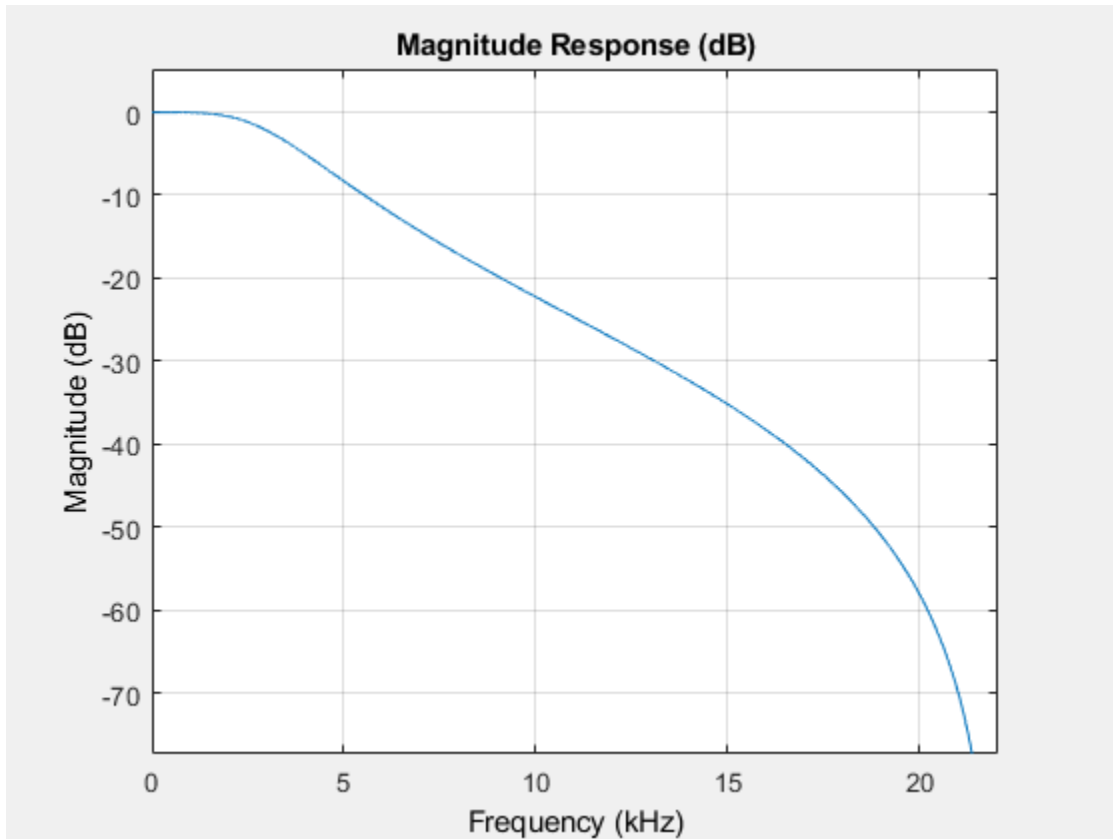
```
count = 0;  
while count < 2500  
    audio = fileReader();  
    deviceWriter(audio);  
    count = count + 1;  
end  
reset(fileReader)
```

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

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

Visualize your filter design. To output filter coefficients suitable for `fvtool`, call `designVarSlopeFilter` again with the same design specifications but with `Orientation` set to "row".

```
[Bvisualize, Avisualize] = designVarSlopeFilter(slope, cutoff, "Orientation", "row");  
fvtool([Bvisualize, Avisualize], "Fs", sampleRate);
```



Create a biquad filter.

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

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

```
scope = dsp.SpectrumAnalyzer( ...  
    "SampleRate", sampleRate, ...  
    "PlotAsTwoSidedSpectrum", false, ...  
    "FrequencyScale", "Log", ...  
    "FrequencyResolutionMethod", "WindowLength", ...
```

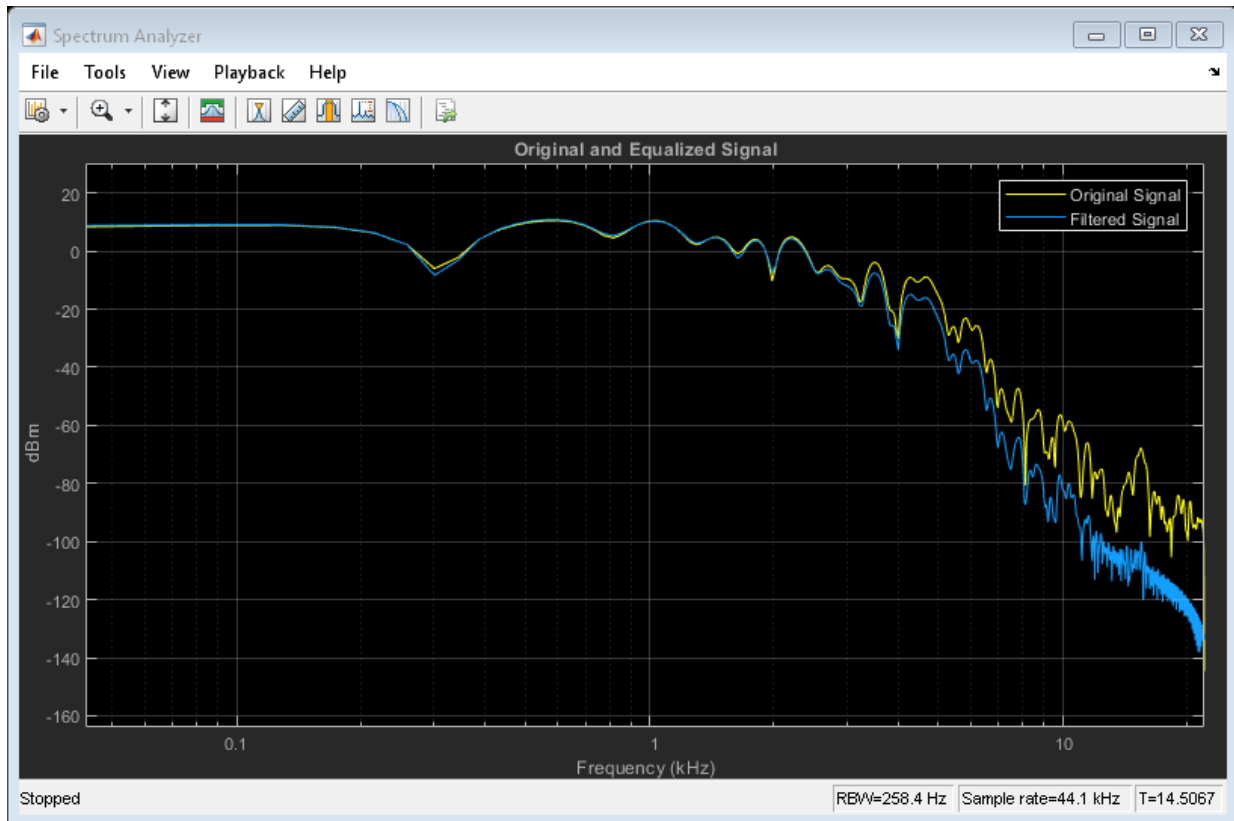
```
"WindowLength",frameSize, ...  
"Title","Original and Equalized Signal", ...  
"ShowLegend",true, ...  
"ChannelNames",{ 'Original Signal', 'Filtered Signal'});
```

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

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

As a best practice, release your objects once done.

```
release(deviceWriter)  
release(fileReader)  
release(scope)
```



### 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.

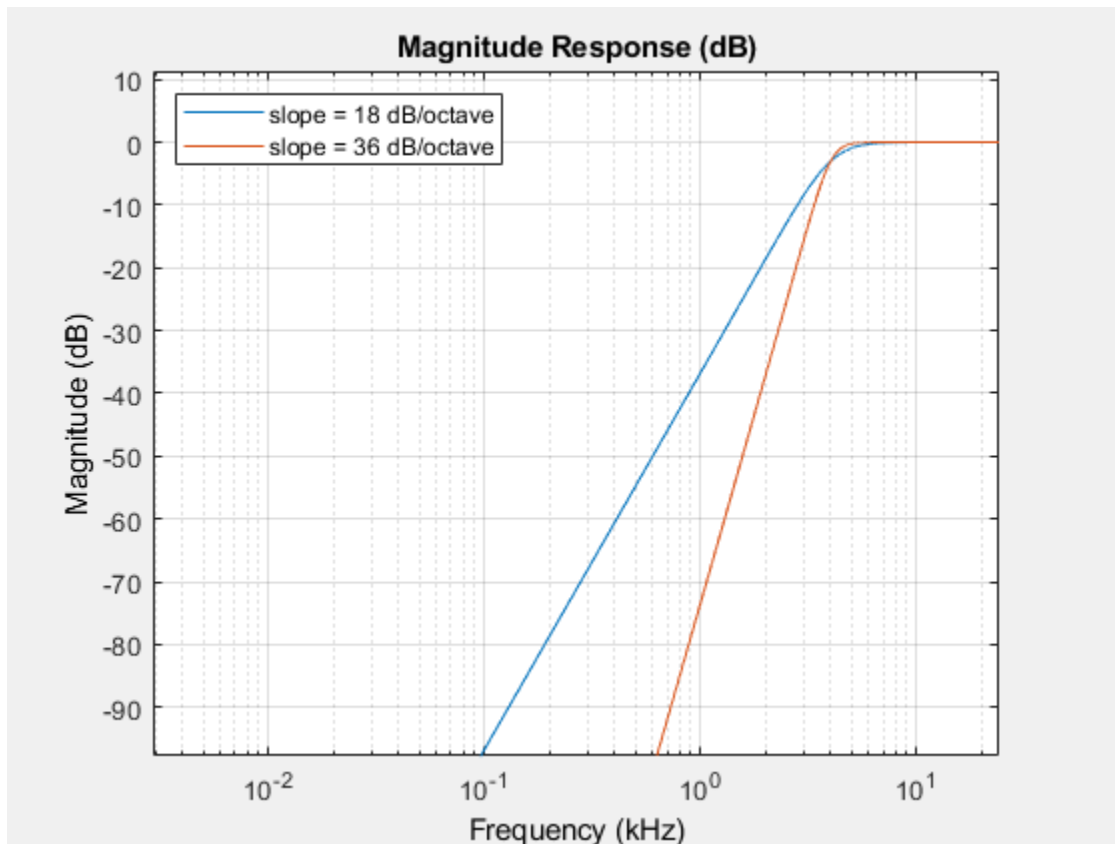
```
Fs = 48e3;
slope1 = 18;
slope2 = 36;
Fc = 4000/(Fs/2);
```

Design the filter coefficients using the specified parameters.

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

Visualize your filter design.

```
fvtool([B1,A1],[B2,A2],...  
       "Fs",Fs,...  
       "FrequencyScale","Log");  
legend("slope = 18 dB/octave", ...  
       "slope = 36 dB/octave", ...  
       "Location","NorthWest")
```



## Diminish Plosives from Speech Signal

Plosives are consonant sounds resulting from a sudden release of airflow. They are most pronounced in words beginning with *p*, *d*, and *g* sounds. Plosives can be emphasized by the recording process and are often displeasurable to hear. In this example, you minimize the plosives of a speech signal by applying highpass filtering and low-band compression.

Create a `dsp.AudioFileReader` object and a `audioDeviceWriter` object to read an audio signal from a file and write an audio signal to a device. Play the unprocessed signal. Then release the file reader and device writer.

```
fileReader = dsp.AudioFileReader('audioPlosives.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

while ~isDone(fileReader)
    audioIn = fileReader();
    deviceWriter(audioIn);
end
release(deviceWriter)
release(fileReader)
```

Design a highpass filter with a steep rolloff of all frequencies below 120 Hz. Use a `dsp.BiquadFilter` object to implement the highpass filter design. Create a crossover filter with one crossover at 250 Hz. The crossover filter enables you to separate the band of interest for processing. Create a dynamic range compressor to compress the dynamic range of plosive sounds. To apply no make-up gain, set the `MakeUpGainMode` to "Property" and use the default 0 dB `MakeUpGain` property value. Create a time scope to visualize the processed and unprocessed audio signal.

```
[B,A] = designVarSlopeFilter(48,120/(fileReader.SampleRate/2),"hi");
biquadFilter = dsp.BiquadFilter( ...
    "SOSMatrixSource","Input port", ...
    "ScaleValuesInputPort",false);

crossFilt = crossoverFilter( ...
    "SampleRate",fileReader.SampleRate, ...
    "NumCrossovers",1, ...
    "CrossoverFrequencies",250, ...
    "CrossoverSlopes",48);

dRCompressor = compressor( ...
    "Threshold",-35, ...
```

```
"Ratio",10, ...
"KneeWidth",20, ...
"AttackTime",1e-4, ...
"ReleaseTime",3e-1, ...
"MakeUpGainMode","Property", ...
"SampleRate",fileReader.SampleRate);

scope = dsp.TimeScope( ...
    "SampleRate",fileReader.SampleRate, ...
    "TimeSpan",3, ...
    "BufferLength",fileReader.SampleRate*3*2, ...
    "YLimits",[-1 1], ...
    "ShowGrid",true, ...
    "ShowLegend",true, ...
    "ChannelNames",{ 'Original', 'Processed' });
```

In an audio stream loop:

- 1 Read in a frame of the audio file.
- 2 Apply highpass filtering using your biquad filter.
- 3 Split the audio signal into two bands.
- 4 Apply dynamic range compression to the lower band.
- 5 Remix the channels.
- 6 Write the processed audio signal to your audio device for listening.
- 7 Visualize the processed and unprocessed signals on a time scope.

As a best practice, release your objects once done.

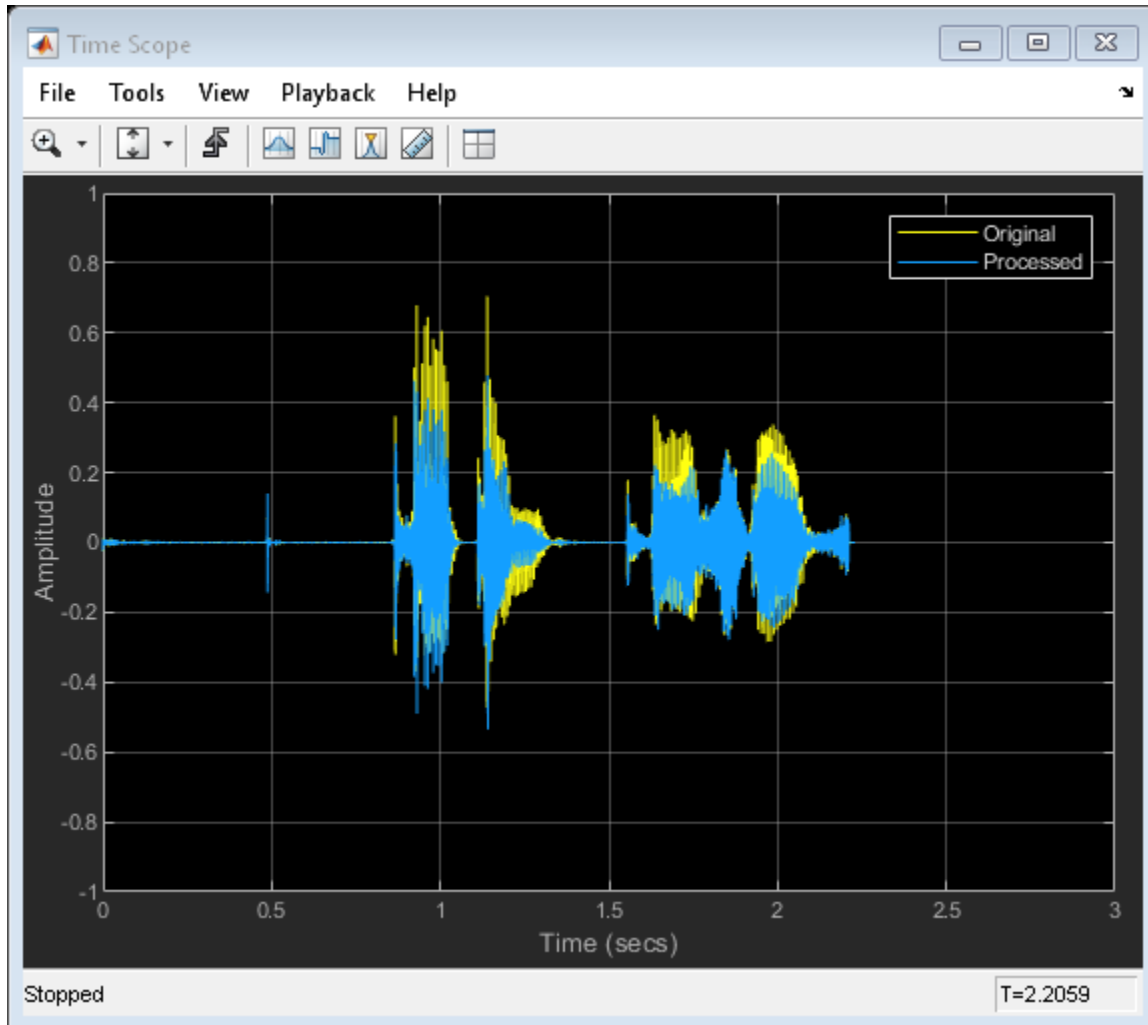
```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioIn = biquadFilter(audioIn,B,A);
    [band1,band2] = crossFilt(audioIn);
    band1compressed = dRCompressor(band1);
    audioOut = band1compressed + band2;
    deviceWriter(audioOut);
    scope([audioIn audioOut])
end
```

As a best practice, release your objects once done.

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



```
release(crossFilt)  
release(dRCompressor)  
release(scope)
```



## 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.

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

### **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 ( $\pi$  rad/sample).

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

### **type** — Filter type

'lo' (default) | 'hi'

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

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

Data Types: `char` | `string`

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Orientation', "row"`

### **Orientation** — Orientation of returned filter coefficients

"column" (default) | "row"

Orientation of returned filter coefficients, specified as the comma-separated pair consisting of 'Orientation' and "column" or "row":

- Set 'Orientation' to "row" for interoperability with **FVTool**, `dsp.DynamicFilterVisualizer`, and `dsp.FourthOrderSectionFilter`.
- Set 'Orientation' to "column" for interoperability with `dsp.BiquadFilter`.

Data Types: char | string

## Output Arguments

### **B — Numerator filter coefficients**

3-by-4 matrix | 4-by-3 matrix

Numerator filter coefficients, returned as a matrix. The size and interpretation of B depends on the Orientation:

- If 'Orientation' is set to "column", then B is 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.
- If 'Orientation' is set to "row", then B is returned as a 4-by-3 matrix. Each row 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 | 4-by-3 matrix

Denominator filter coefficients, returned as a matrix. The size and interpretation of A depends on the Orientation:

- If 'Orientation' is set to "column", then A is 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.
- If 'Orientation' is set to "row", then B is returned as a 4-by-3 matrix. Each row of B corresponds to the denominator coefficients of a different second-order section of your cascaded IIR filter.

## References

[1] Orfanidis, Sophocles J. "High-Order Digital Parametric Equalizer Design." *Journal of the Audio Engineering Society*. Vol. 53, November 2005, pp. 1026-1046.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`designParamEQ` | `designShelvingEQ` | `multibandParametricEQ`

### Topics

"Parametric Equalizer Design"

"Equalization"

**Introduced in R2016a**

# disconnectMIDI

Disconnect MIDI controls from audio object

## Syntax

```
disconnectMIDI(audioObject)
```

## Description

`disconnectMIDI(audioObject)` disconnects MIDI controls from your audio object, `audioObject`. 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

### **audioObject** — Audio object

object

Audio plugin or compatible System object, specified as an object that inherits from the `audioPlugin` class or an object of a compatible Audio Toolbox System object.

## See Also

### Classes

`audioPlugin` | `audioPluginSource`

### **Functions**

configureMIDI | getMIDIConnections | midicallback | midicontrols | midiid |  
midiread | midisync

### **Topics**

“MIDI Control for Audio Plugins”  
“MIDI Control Surface Interface”

### **Introduced in R2016a**

## **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 character vector. The choices for `spec` are as follows:

- 'F0, BW, BWp, Gref, G0, GBW, Gp' (minimum order default)
- 'F0, BW, BWst, Gref, G0, GBW, Gst'
- 'F0, BW, BWp, Gref, G0, GBW, Gp, Gst'
- 'N, F0, BW, Gref, G0, GBW'
- 'N, F0, BW, Gref, G0, GBW, Gp'
- 'N, F0, Fc, Qa, G0'
- 'N, F0, Fc, S, G0'
- 'N, F0, BW, Gref, G0, GBW, Gst'
- 'N, F0, BW, Gref, G0, GBW, Gp, Gst'
- 'N, Flow, Fhigh, Gref, G0, GBW'
- 'N, Flow, Fhigh, Gref, G0, GBW, Gp'
- 'N, Flow, Fhigh, Gref, G0, GBW, Gst'
- 'N, Flow, Fhigh, Gref, G0, GBW, Gp, Gst'

where the parameters are defined as follows:



Parameter	Definition	Unit
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 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  $G_0 > G_{ref}$ ; to cut, set  $G_{ref} > G_0$
- For boost:  $G_0 > G_p > GBW > G_{st} > G_{ref}$ ; For cut:  $G_0 < G_p < GBW < G_{st} < G_{ref}$
- Bandwidth must satisfy:  $BW_{st} > BW > BW_p$

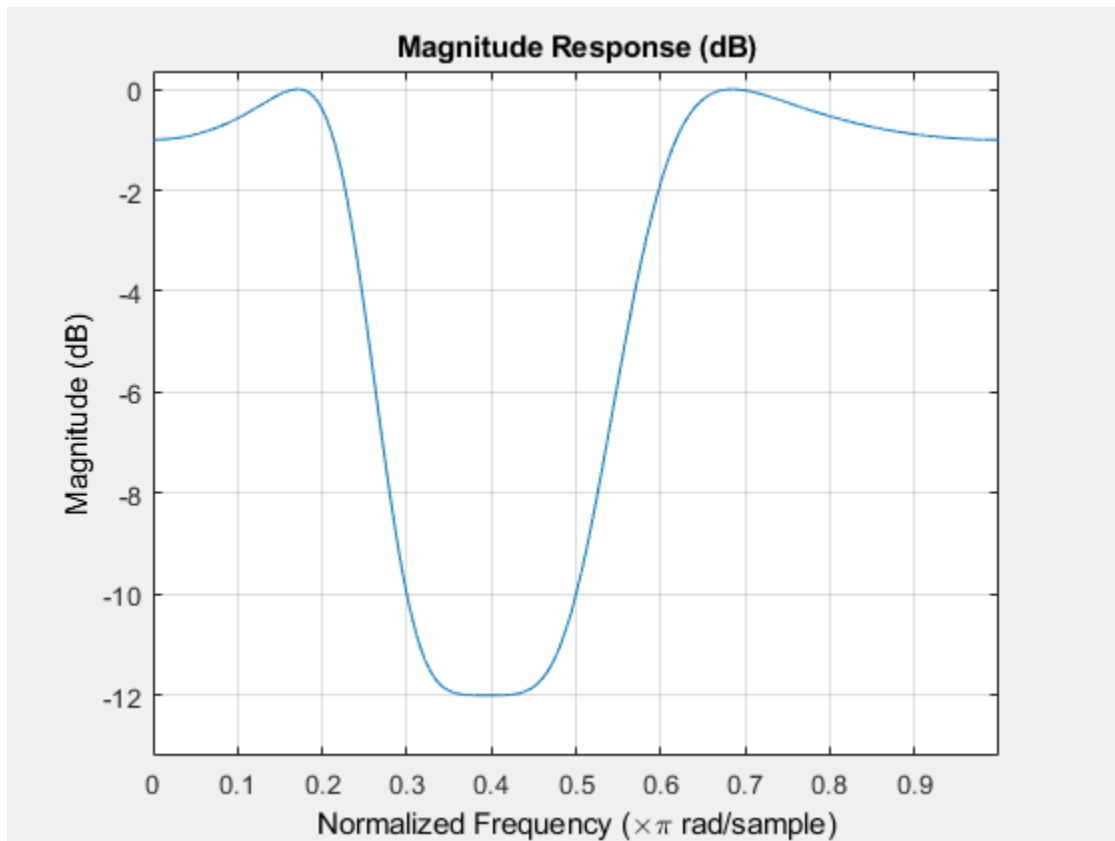
`d = fdesign.parmeq(... fs)` adds the input sampling frequency. `fs` must be specified as a scalar trailing the other numerical values provided, and is assumed to be in Hz.

## Examples

### Design Parametric Equalizers

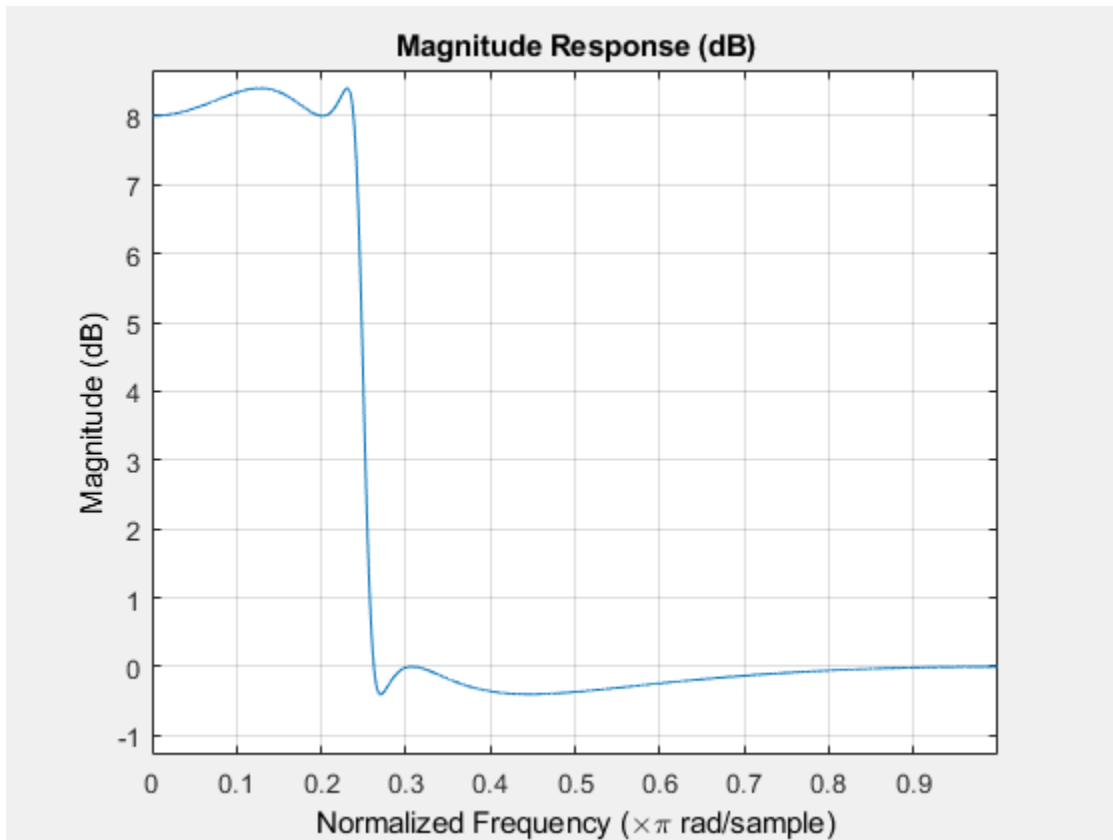
Design a Chebyshev Type II parametric equalizer filter that cuts by 12 dB.

```
parametricEQ = fdesign.parmeq('N,Flow,Fhigh,Gref,G0,GBW,Gst', ...  
    4,0.3,0.5,0,-12,-10,-1);  
  
parametricEQBiquad = design(parametricEQ,'cheby2','SystemObject',true);  
fvtool(parametricEQBiquad)
```



Design a 4th-order lowpass shelving filter with a normalized cutoff frequency of 0.25, a quality factor of 10, and an 8 dB boost gain.

```
parametricEQ = fdesign.parmeq('N,F0,Fc,Qa,G0',4,0,0.25,10,8);
parametricEQBiquad = design(parametricEQ,'SystemObject',true);
fvtool(parametricEQBiquad)
```



Design 4th-order highpass shelving filters with slopes of 1.5 and 3.

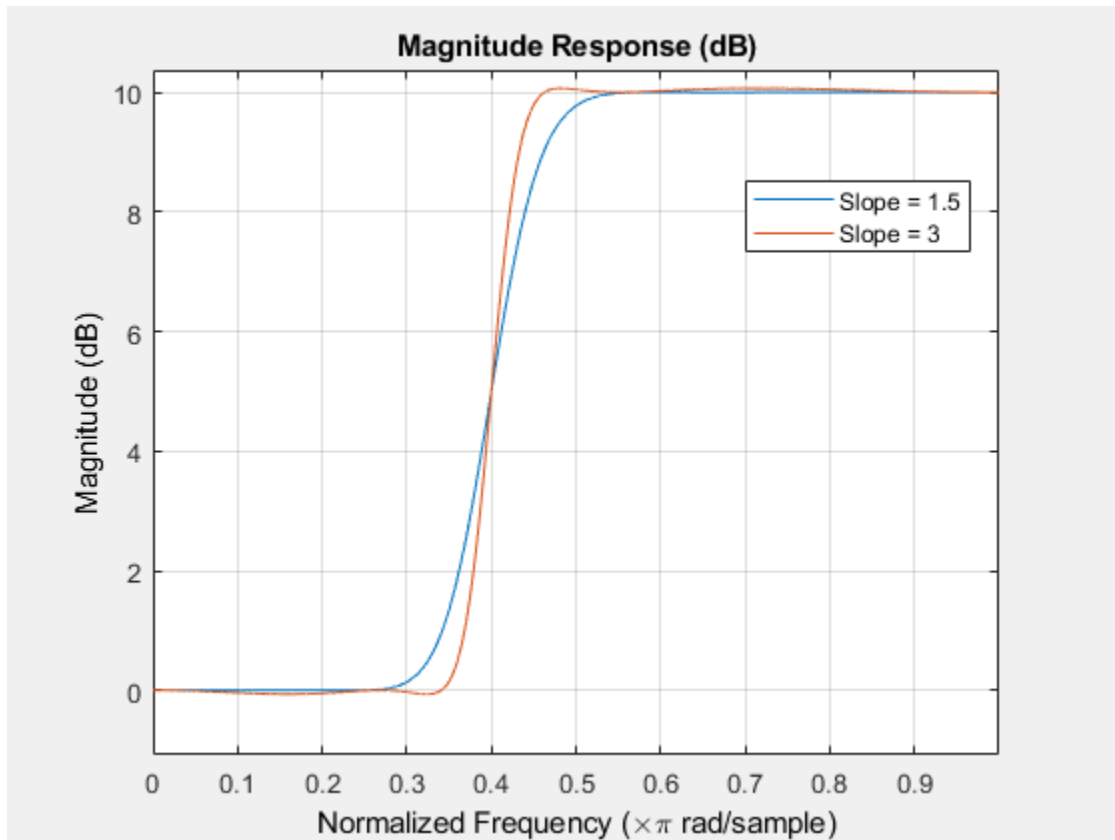
```
N = 4; % Filter order
F0 = 1; % Center Frequency (normalized)
Fc = 0.4; % Cutoff Frequency (normalized)
G0 = 10; % Center Frequency Gain (dB)
```

```
S1 = 1.5; % Slope for filter design 1
S2 = 3;  % Slope for filter design 2

filter = fdesign.parmeq('N,F0,Fc,S,G0',N,F0,Fc,S1,G0);
filterDesignS1 = design(filter,'SystemObject',true);

filter.S = S2;
filterDesignS2 = design(filter,'SystemObject',true);

filterVisualization = fvtool(filterDesignS1,filterDesignS2);
legend(filterVisualization,'Slope = 1.5','Slope = 3');
```



## See Also

[designParamEQ](#) | [designShelvingEQ](#) | [designVarSlopeFilter](#) | [fdesign](#) | [multibandParametricEQ](#)

## Topics

[“Parametric Equalizer Design”](#)  
[“Equalization”](#)

# 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 for a list of compilers supported by `generateAudioPlugin`.

`generateAudioPlugin options className` specifies a nondefault plugin type, output folder, file name, or file type. You can use the `-juceproject` option to create a zip file containing generated C/C++ code and a JUCER project. Options can be specified in any grouping, and in any order.

## 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, myPluginFolder. 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 coolEffect -outdir myPluginFolder audiopluginexample.Echo
```

A VST 2 plugin named coolEffect is saved to your specified folder, myPluginFolder. 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.

### Generate Zip File Compatible with JUCE 5.3.2

```
generateAudioPlugin -juceproject audiopluginexample.Echo
```

A zip file containing generated C/C++ code and a JUCER project file suitable for use with JUCE 5.3.2 is saved to your current folder.

## Input Arguments

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

```
-au | -vst | -outdir folder | -output fileName | -win32
```

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

Option	Description
-au	Generates an Audio Unit (AU) v2 audio plugin binary. This syntax is only valid on macOS.
-vst	Generates a VST 2 audio plugin binary. By default, generateAudioPlugin generates a VST 2 plugin.
-outdir <i>folder</i>	Generates a plugin or zip file to a specific folder. By default, the generated plugin is placed in the current folder. If <i>folder</i> is not in the current folder, specify the exact path.
-output <i>fileName</i>	Specifies the file name of the generated plugin or zip file. The appropriate extension is appended to the <i>fileName</i> based on the platform on which the plugin or zip file is generated. By default, the plugin or zip file is named after the class.
-win32	Creates a 32-bit audio plugin. Valid only on win64.
-juceproject	Creates a zip file containing generated C/C++ code and a JUCER project file suitable for use with JUCE 5.3.2. You can use the generated zip file to modify the generated plugin or compile it to a format other than VST 2.4. This option requires a MATLAB Coder™ license. To use the generated files with JUCE, you must obtain your own appropriately licensed copy of JUCE.

**className — Name of 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.

You can specify the plugin class to generate by specifying its class name or file name. For example, the following syntaxes perform equivalent operations:

- `generateAudioPlugin myPlugin`
- `generateAudioPlugin myPlugin.m`

If you want to specify the plugin class by file name, and your plugin class is inside a package, you must specify the package as a file path. For example, the following syntaxes perform equivalent operations:



- `generateAudioPlugin myPluginPackage.myPlugin`
- `generateAudioPlugin +myPluginPackage/myPlugin.m`

## Limitations

Build problems can occur when using folder names with spaces. For more information, see “Build Process Support for Folder Names with Spaces or Special Characters” (Simulink Coder) and Why is the build process failing for a shipped model in Simulink or for a model run in Accelerator mode?.

## More About

### Generated VST Plugin File Extension

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

Operating System	File Extension
Windows	.dll
macOS	.vst

## See Also

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

## Topics

“Audio Plugins in MATLAB”

“Export a MATLAB Plugin to a DAW”

**Introduced in R2016a**

## integratedLoudness

Measure integrated loudness and loudness range

### Syntax

```
loudness = integratedLoudness(audioIn,Fs)  
loudness = integratedLoudness(audioIn,Fs,channelWeights)  
[loudness,loudnessRange] = integratedLoudness(____)
```

### Description

`loudness = integratedLoudness(audioIn,Fs)` returns the integrated loudness of an audio signal, `audioIn`, with sample rate `Fs`. The ITU-R BS.1770-4 and EBU R 128 standards define the algorithms to calculate integrated loudness.

`loudness = integratedLoudness(audioIn,Fs,channelWeights)` specifies the channel weights used to compute the integrated loudness. `channelWeights` must be a row vector with the same number of elements as the number of channels in `audioIn`.

`[loudness,loudnessRange] = integratedLoudness(____)` returns the loudness range of the audio signal using either of the previous syntaxes. The EBU R 128 Tech 3342 standard defines the loudness range computation.

### Examples

#### Determine Integrated Loudness

Determine the integrated loudness of an audio signal.

Create a two-second sine wave with a 0 dB amplitude, a 1 kHz frequency, and a 48 kHz sample rate.

```
sampleRate = 48e3;  
increment = sampleRate*2;
```

```
amplitude = 10^(0/20);
frequency = 1e3;

sineGenerator = audioOscillator( ...
    'SampleRate',sampleRate, ...
    'SamplesPerFrame',increment, ...
    'Amplitude',amplitude, ...
    'Frequency',frequency);

signal = sineGenerator();
```

Calculate the integrated loudness of the audio signal at the specified sample rate.

```
loudness = integratedLoudness(signal,sampleRate)

loudness = -3.0036
```

### Specify Nondefault Channel Weights

Read in a four-channel audio signal. Specify a nondefault weighting vector with four elements.

```
[signal,fs] = audioread('AudioArray-16-16-4channels-20secs.wav');
weightingVector = [1,0.8,0.8,1.2];
```

Calculate the integrated loudness with the default channel weighting and the nondefault channel weighting vector.

```
standardLoudness = integratedLoudness(signal,fs,weightingVector)

standardLoudness = -11.6825

nonStandardLoudness = integratedLoudness(signal,fs)

nonStandardLoudness = -11.0121
```

### Determine Loudness Range

Read in an audio signal. Clip 3 five-second intervals out of the signal.

```
[x,fs] = audioread('FunkyDrums-44p1-stereo-25secs.mp3');
x1 = x(1:fs*5,:);
x2 = x(5e5:5e5+5*fs,:);
x3 = x(end-5*fs:end,:);
```

Calculate the loudness and loudness range of the total signal and of each interval.

```
[L,LRA] = integratedLoudness(x,fs);
[L1,LRA1] = integratedLoudness(x1,fs);
[L2,LRA2] = integratedLoudness(x2,fs);
[L3,LRA3] = integratedLoudness(x3,fs);

fprintf(['Loudness: %0.2f\n', ...
        'Loudness range: %0.2f\n\n', ...
        'Beginning loudness: %0.2f\n', ...
        'Beginning loudness range: %0.2f\n\n', ...
        'Middle loudness: %0.2f\n', ...
        'Middle loudness range: %0.2f\n\n', ...
        'End loudness: %0.2f\n', ...
        'End loudness range: %0.2f\n'], ...
        L,LRA,L1,LRA1,L2,LRA2,L3,LRA3);
```

```
Loudness: -22.98
Loudness range: 1.50
```

```
Beginning loudness: -23.38
Beginning loudness range: 1.18
```

```
Middle loudness: -22.97
Middle loudness range: 1.14
```

```
End loudness: -22.10
End loudness range: 1.82
```

## Input Arguments

### **audioIn** — Input signal

matrix

Input signal, specified as a matrix. The columns of the matrix are treated as audio channels.

The maximum number of columns of the input signal depends on your `channelWeights` specification:

- If you use the default `channelWeights`, the input signal has a maximum of five channels. Specify the channels in this order: [Left, Right, Center, Left surround, Right surround].
- If you specify nondefault `channelWeights`, the input signal must have the same number of columns as the number of elements in the `channelWeights` vector.

Data Types: `single` | `double`

### **Fs — Sample rate (Hz)**

positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Data Types: `single` | `double`

### **channelWeights — Linear weighting applied to each input channel**

[1.0, 1.0, 1.0, 1.41, 1.41] (default) | nonnegative row vector

Linear weighting applied to each input channel, specified as a row vector of nonnegative values. The number of elements in the row vector must be equal to or greater than the number of input channels. Excess values in the vector are ignored.

The default channel weights follow the ITU-R BS.1170-4 standard. To use the default channel weights, specify the channels of the `audioIn` matrix in this order: [Left, Right, Center, Left surround, Right surround].

It is a best practice to specify the `channelWeights` vector in order: [Left, Right, Center, Left surround, Right surround].

Data Types: `single` | `double`

## **Output Arguments**

### **Loudness — Integrated loudness (LUFS)**

scalar

Integrated loudness in loudness units relative to full scale (LUFS), returned as a scalar.

The ITU-R BS.1770-4 and EBU R 128 standards define the integrated loudness. The algorithm computes the loudness by breaking down the audio signal into 0.4-second segments with 75% overlap. If the input signal is less than 0.4 seconds, loudness is returned empty.

Data Types: `single` | `double`

**LoudnessRange — Loudness range (LU)**

scalar

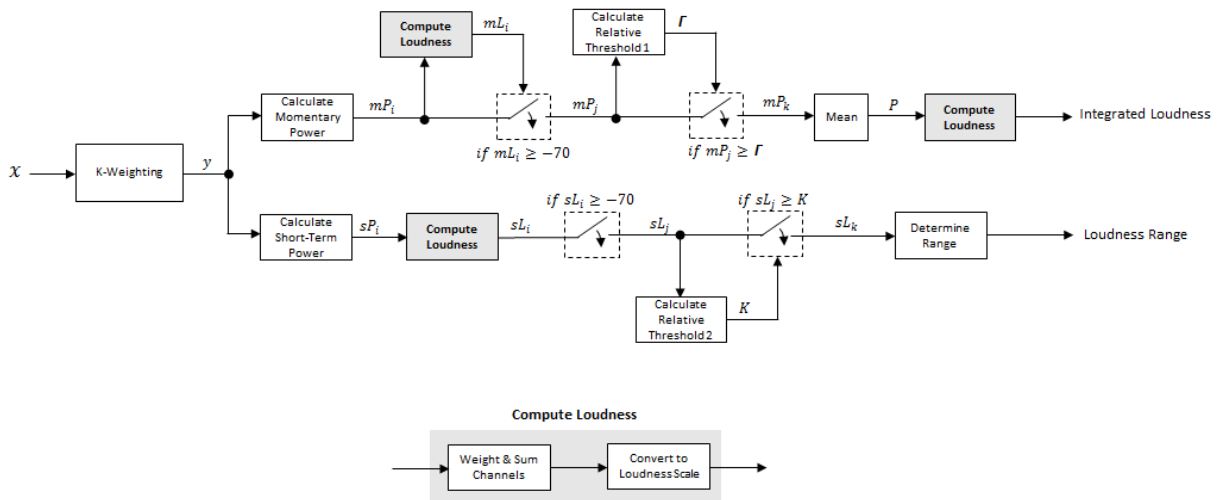
Loudness range in loudness units (LU), returned as a scalar.

The EBU R 128 Tech 3342 standard defines the loudness range. The algorithm computes the loudness range by breaking down the audio into 3-second segments with 2.9-second overlap. If the input signal is less than three seconds, loudnessRange is returned empty.

Data Types: `single` | `double`

## Algorithms

The `integratedLoudness` function returns the integrated loudness and loudness range (LRA) of an audio signal. You can specify any number of channels and nondefault channel weights used for loudness measurements. The `integratedLoudness` algorithm is described for the general case of  $n$  channels.



## Integrated Loudness and Loudness Range

The input channels,  $x$ , pass through a K-weighted weightingFilter. The K-weighted filter shapes the frequency spectrum to reflect perceived loudness.

### Integrated Loudness

- 1 The K-weighted channels,  $y$ , are divided into 0.4-second segments with 0.3-second overlap. The power (mean square) of each segment of the K-weighted channels is calculated:

$$mP_i = \frac{1}{w} \sum_{k=1}^w y_i^2[k]$$

- $mP_i$  is the momentary power of the  $i$ th segment of a channel.
- $w$  is the segment length in samples.

- 2 The momentary loudness,  $mL$ , is computed for each segment:

$$mL_i = -0.691 + 10 \log_{10} \left( \sum_{c=1}^n G_c \times mP_{(i,c)} \right) \quad LUFS$$

- $G_c$  is the weighting for channel  $c$ .

- 3 The momentary power is gated using the momentary loudness calculation:

$$mP_i \rightarrow mP_j$$

$$j = \{ i \mid mL_i \geq -70 \}$$

- 4 The relative threshold,  $\Gamma$ , is computed:

$$\Gamma = -0.691 + 10 \log_{10} \left( \sum_{c=1}^n G_c \times l_c \right) - 10$$

$l_c$  is the mean momentary power of channel  $c$ :

$$l_c = \frac{1}{|j|} \sum_j mP_{(j,c)}$$

- 5 The momentary power subset,  $mP_j$ , is gated using the relative threshold:

$$mP_j \rightarrow mP_k$$

$$k = \{ j \mid mP_j \geq \Gamma \}$$

- 6 The momentary power segments are averaged:

$$P = \frac{1}{|k|} \sum_k mP_k$$

- 7 The integrated loudness is computed by passing the mean momentary power subset,  $P$ , through the Compute Loudness system:

$$\text{Integrated Loudness} = -0.691 + 10 \log_{10} \left( \sum_{c=1}^n G_c \times P_c \right) \text{ LUFS}$$

### Loudness Range

- 1 The K-weighted channels,  $y$ , are divided into 3-second segments with 2.9-second overlap. The power (mean square) of each segment of the K-weighted channels is calculated:

$$sP_i = \frac{1}{w} \sum_{k=1}^w y_i^2[k]$$

- $sP_i$  is the short-term power of the  $i$ th segment of a channel.
- $w$  is the segment length in samples.

- 2 The short-term loudness,  $sL$ , is computed for each segment:

$$sL_i = -0.691 + 10 \log_{10} \left( \sum_{c=1}^n G_c \times sP_{(i,c)} \right)$$

- $G_c$  is the weighting for channel  $c$ .

- 3 The short-term loudness is gated using an absolute threshold:

$$sL_i \rightarrow sL_j$$

$$j = \{ i \mid sL_i \geq -70 \}$$

- 4 The gated short-term loudness is converted back to linear, and then the mean is taken:

$$sP_j = \frac{1}{|j|} \sum_j 10^{(sL_j/10)}$$



The relative threshold,  $K$ , is computed:

$$K = -20 + 10\log_{10}(sP_j)$$

- 5 The short-term loudness subset,  $sL_j$ , is gated using the relative threshold:

$$sL_j \rightarrow sL_k$$

$$k = \{ j \mid sL_j \geq K \}$$

- 6 The short-term loudness subset,  $sL_k$ , is sorted. The loudness range is calculated as between the 10th and 95th percentiles of the distribution, and is returned in loudness units (LU).

## References

- [1] International Telecommunication Union; Radiocommunication Sector. *Algorithms to Measure Audio Programme Loudness and True-Peak Audio Level*. ITU-R BS.1770-4. 2015.
- [2] European Broadcasting Union. *Loudness Normalisation and Permitted Maximum Level of Audio Signals*. EBU R 128. 2014.
- [3] European Broadcasting Union. *Loudness Metering: 'EBU Mode' Metering to Supplement EBU R 128 Loudness Normalization*. EBU R 128 Tech 3341. 2014.
- [4] European Broadcasting Union. *Loudness Range: A Measure to Supplement EBU R 128 Loudness Normalization*. EBU R 128 Tech 3342. 2016.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

Loudness Meter | loudnessMeter | weightingFilter

**Introduced in R2016b**

# getMIDIConnections

Get MIDI connections of audio object

## Syntax

```
connectionInfo = getMIDIConnections(audioObject)
```

## Description

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

The `connectionInfo` structure contains a substructure for each tunable property of `audioObject` 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 `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

**audioObject** — Audio object  
object

Audio plugin or compatible System object, specified as an object that inherits from the `audioPlugin` class or an object of a compatible Audio Toolbox System object.

## 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).

## See Also

### Classes

audioPlugin | audioPluginSource

### Functions

configureMIDI | disconnectMIDI | midicallback | midicontrols | midiid |  
midiread | midisync

### Topics

“MIDI Control for Audio Plugins”

“MIDI Control Surface Interface”

### Introduced in R2016a

## loadAudioPlugin

Load VST, VST3, and AU plugins into MATLAB environment

### Syntax

```
hostedPlugin = loadAudioPlugin(pluginpath)
```

### Description

`hostedPlugin = loadAudioPlugin(pluginpath)` loads the 64-bit VST, VST3, or AU audio plugin specified by `pluginpath`. On Windows, you can load VST and VST3 plugins. On macOS, you can load AU, VST, and VST3 plugins.

Your hosted plugin has two display modes: **Parameters** and **Properties**. The default display mode is **Properties**.

- **Parameters** -- Interact with normalized parameter values of the hosted plugin using `set` and `get` functions.
- **Properties** -- Interact with heuristically interpreted parameters with real-world values. You can use standard dot notation to set and get the values while using this mode.

You can specify the display mode of the hosted plugin using standard dot notation, for example:

```
hostedPlugin.DisplayMode = 'Parameters';
```

See “Host External Audio Plugins” for a discussion of display modes and a walkthrough of both modes of interaction.

You can interact with and exercise the hosted plugin using the following functions.

#### Process Audio

- `audioOut = process(hostedPlugin, audioIn)`

Returns an audio signal processed according to the algorithm and parameters of the hosted plugin. For source plugins, call `process` without an audio input.

### **Set and Get Normalized Parameter Values**

- `value = getParameter(hostedPlugin,parameter)`

Returns the normalized value of the specified hosted plugin parameter. Normalized values are in the range [0,1]. You can specify a parameter by its name or by its index. To specify the name, use a character vector.

- `setParameter(hostedPlugin,parameter,newValue)`

Sets the normalized value of the specified hosted plugin parameter to `newValue`. Normalized values are in the range [0,1].

### **Get High-Level Information About the Hosted Plugin**

- `dispParameter(hostedPlugin)`

Displays all parameters and associated indices, values, displayed values, and display labels of the hosted plugin.

- `pluginInfo = info(hostedPlugin)`

Returns a structure containing information about the hosted plugin.

### **Set the Environment in Which the Plugin Is Run**

- `frameSize = getSamplesPerFrame(hostedPlugin)`

Returns the frame size that the hosted plugin returns in subsequent calls to its processing function (source plugins only).

- `setSamplesPerFrame(hostedPlugin,frameSize)`

Sets the frame size that the hosted plugin must return in subsequent calls to its processing function (source plugins only).

- `setSampleRate(hostedPlugin,sampleRate)`

Sets the sample rate of the hosted plugin.

- `sampleRate = getSampleRate(hostedPlugin)`

Returns the sample rate in Hz at which the plugin is being run.

## Examples

### Host External Plugins in MATLAB

Use `loadAudioPlugin` to host a VST external plugin and a VST external source plugin in MATLAB®.

Use the `fullfile` command to determine the full path to the oscillator VST plugin and parametric equalizer VST plugin included with Audio Toolbox™. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
oscPluginPath = ...
    fullfile(matlabroot, 'toolbox/audio/samples/oscillator.dll');
EQPluginPath = ...
    fullfile(matlabroot, 'toolbox/audio/samples/ParametricEqualizer.dll');
```

Create external plugin objects by calling `loadAudioPlugin` for each of the plugin paths.

```
hostedSourcePlugin = loadAudioPlugin(oscPluginPath);
hostedPlugin = loadAudioPlugin(EQPluginPath);
```

Hosted plugins derive from either the `externalAudioPlugin` or `externalAudioSourcePlugin` class. Because `oscillator.dll` is a source audio plugin, the hosted object derives from `externalAudioSourcePlugin`. Use `class()` to verify the classes of the hosted plugins.

```
class(hostedPlugin)

class(hostedSourcePlugin)
```

Call the hosted plugins to display basic information about them. This information includes the format, the plugin name, the number of channels in and out, and the tunable properties of the plugin. Source plugins also display the frame size of the plugin.

```
hostedSourcePlugin
hostedPlugin
```



## Run External Plugin in MATLAB

Load a VST audio plugin into MATLAB™ by specifying its full path. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
pluginPath = ...
    fullfile(matlabroot, 'toolbox/audio/samples/ParametricEqualizer.dll');
hostedPlugin = loadAudioPlugin(pluginPath);
```

Create input and output objects for an audio stream loop that reads from a file and writes to your audio device. Set the sample rate of the hosted plugin to the sample rate of the input to the plugin.

```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3');
deviceWriter = audioDeviceWriter('SampleRate', fileReader.SampleRate);
setSampleRate(hostedPlugin, fileReader.SampleRate);
```

Set the `MediumPeakGain` property to -20 dB.

```
hostedPlugin.MediumPeakGain = -20;
```

Use the hosted plugin to process the audio file in an audio stream loop. Sweep the medium peak gain upward in the loop to hear the effect.

```
while hostedPlugin.MediumPeakGain < 19
    hostedPlugin.MediumPeakGain = hostedPlugin.MediumPeakGain + 0.04;
    x = fileReader();
    y = process(hostedPlugin, x);
    deviceWriter(y);
end

release(fileReader)
release(deviceWriter)
```

## Run External Source Plugin in MATLAB

Load a VST audio source plugin into MATLAB™ by specifying its full path. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
pluginPath = fullfile(matlabroot, 'toolbox', 'audio', 'samples', 'oscillator.dll');
hostedSourcePlugin = loadAudioPlugin(pluginPath);
```

Set the `Amplitude` property to 0.5. Set the `Frequency` property to 16 kHz.

```
hostedSourcePlugin.Amplitude = 0.5;  
hostedSourcePlugin.Frequency = 16000;
```

Set the sample rate at which to run the plugin. Create an output object to write to your audio device.

```
setSampleRate(hostedSourcePlugin,44100);  
deviceWriter = audioDeviceWriter('SampleRate',44100);
```

Use the hosted source plugin to output an audio stream. The processing in the audio stream loop ramps the frequency parameter down and then up.

```
k = 1;  
for i = 1:1000  
    hostedSourcePlugin.Frequency = hostedSourcePlugin.Frequency - 30*k;  
    y = process(hostedSourcePlugin);  
    deviceWriter(y);  
    if (hostedSourcePlugin.Frequency - 30 <= 0.1) || ...  
        (hostedSourcePlugin.Frequency + 30 >= 20e3)  
        k = -1*k;  
    end  
end  
release(deviceWriter)
```

## Input Arguments

### **pluginpath** — Location of external plugin

character vector | string

Location of the external plugin, specified as a character vector. Use the full path to specify the audio plugin you want to host in MATLAB. If the plugin is located in the current folder, specify it by its name.

Example: `loadAudioPlugin('coolPlugin.dll')`

Example: `loadAudioPlugin('C:\Program Files\VSTPlugins\coolPlugin.dll')`

### **Plugin Path for Mac**

For macOS, the plugin locations are predetermined depending on if the plugin was saved system wide or for a particular user.

This table shows the system-wide paths.

Plugin Type	Path
VST2	/Library/Audio/Plug-Ins/VST/coolPlugin.vst
VST3	/Library/Audio/Plug-Ins/VST3/coolPlugin.vst3
AU	/Library/Audio/Plug-Ins/Components/coolPlugin.component

This table shows the user-specific paths.

Plugin Type	Path
VST2	~/Library/Audio/Plug-Ins/VST/coolPlugin.vst
VST3	~/Library/Audio/Plug-Ins/VST3/coolPlugin.vst3
AU	~/Library/Audio/Plug-Ins/Components/coolPlugin.component

## Output Arguments

### hostedPlugin — Object of external plugin

externalAudioPlugin | externalAudioSourcePlugin

Object of an external plugin, derived from the externalAudioPlugin or externalAudioSourcePlugin class. You can interact with the hosted plugin as a DAW would, with the additional functionality of the MATLAB environment.

## Limitations

The loadAudioPlugin function supports 64-bit plugins only. You cannot load 32-bit plugins using the loadAudioPlugin function.

## See Also

audioPlugin | audioPluginSource | externalAudioPlugin | externalAudioPluginSource

## Topics

“Host External Audio Plugins”

**Introduced in R2016b**

# 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;
```

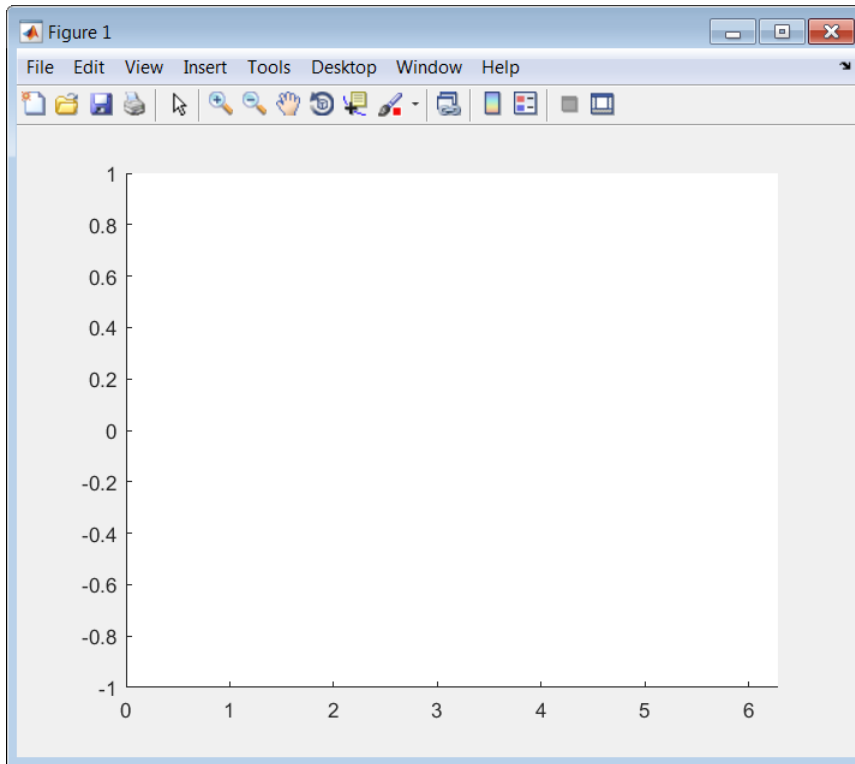
```
Move the control you wish to identify; type ^C to abort.
Waiting for control message...
```

Create an object that responds to the control you specified.

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

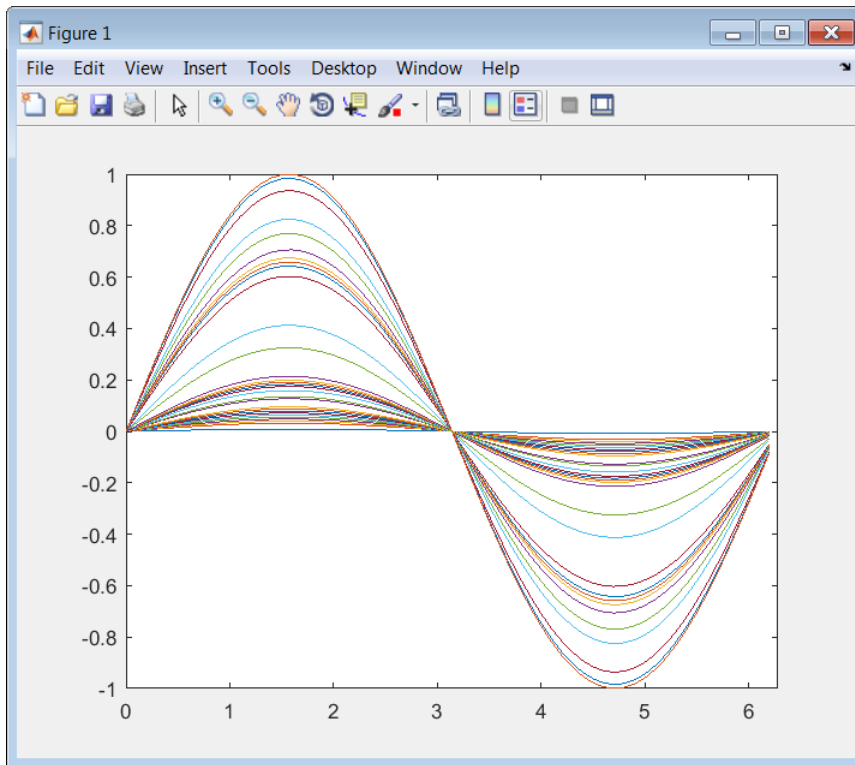
Define a function that plots a sinusoid with the amplitude set by your MIDI control. Make the axis constant.

```
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('%.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

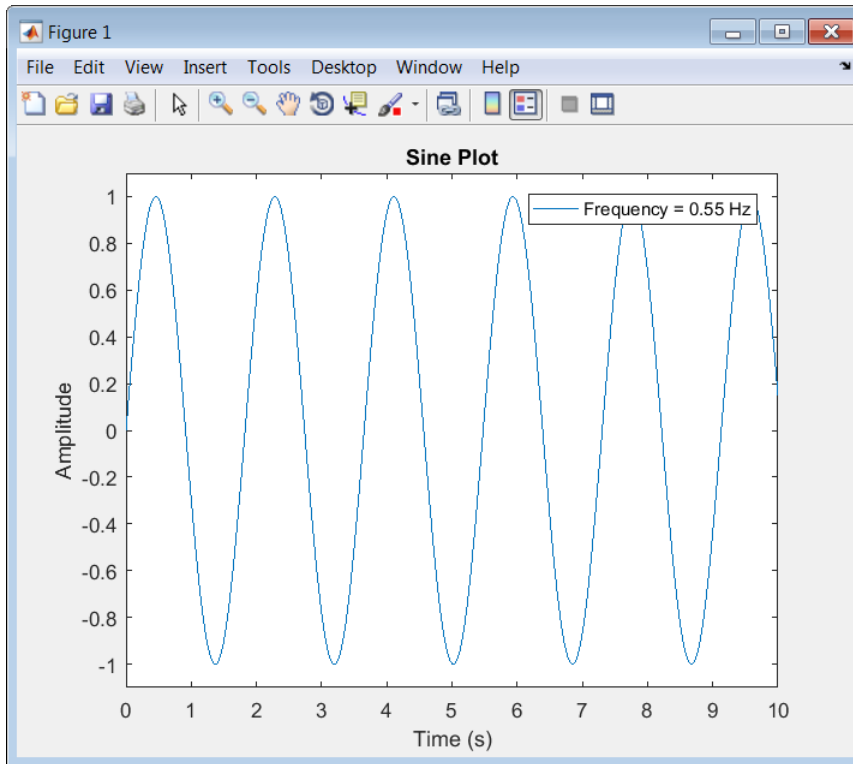
Define this function and save it to your current folder.

```
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

**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`.

**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 `midicontrolsObject` changes value. For information on what function handles are, see “Function Handles” (MATLAB).

## 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.

## See Also

### **Functions**

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

### **Topics**

“MIDI Control Surface Interface”

“MIDI Control for Audio Plugins”

### **Introduced in R2016a**

# 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 `midiiid` to interactively identify the control numbers of your device. See “MIDI Device Control Numbers” on page 2-523 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

character vector | 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 2-520 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 `mid iid` to identify the name of the device you want.

Example: `'MIDIDevice', 'BCF2000 MIDI 1'`

Data Types: `char` | `string`

### **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 | string

## 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*.

## See Also

### Functions

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

### Topics

“MIDI Control Surface Interface”

“MIDI Control for Audio Plugins”

**Introduced in R2016a**

# **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

### **Functions**

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

### **Topics**

“MIDI Control Surface Interface”

“MIDI Control for Audio Plugins”

### **Introduced in R2016a**

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

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

```
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.

## See Also

### Functions

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

### Topics

“MIDI Control Surface Interface”

“MIDI Control for Audio Plugins”

**Introduced in R2016a**

## 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`

## See Also

### Functions

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

### Topics

“MIDI Control Surface Interface”

“MIDI Control for Audio Plugins”

### Introduced in R2016a

# validateAudioPlugin

Test MATLAB source code for audio plugin

## Syntax

```
validateAudioPlugin classname  
validateAudioPlugin options classname
```

## Description

`validateAudioPlugin classname` generates and runs a “Test Bench Procedure” on page 2-538 that exercises your audio plugin class.

`validateAudioPlugin options classname` specifies options to modify the default “Test Bench Procedure” on page 2-538.

## 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.

### **classname** — 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.

You can specify the plugin class to validate by specifying its class name or file name. For example, the following syntaxes perform equivalent operations:

- `validateAudioPlugin myPlugin`
- `validateAudioPlugin myPlugin.m`

If you want to specify the plugin class by file name, and your plugin class is inside a package, you must specify the package as a file path. For example, the following syntaxes perform equivalent operations:

- `validateAudioPlugin myPluginPackage.myPlugin`
- `validateAudioPlugin +myPluginPackage/myPlugin.m`

## Limitations

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

## More About

### Test Bench Procedure

The `validateAudioPlugin` 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.

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

## See Also

### Functions

`generateAudioPlugin`

### Classes

`audioPlugin` | `audioPluginSource`

### Topics

“Audio Plugins in MATLAB”

**Introduced in R2016a**

# **System objects in Audio Toolbox**

---

# audioTimeScaler

Apply time scaling to streaming audio

## Description

The `audioTimeScaler` object performs audio time scale modification (TSM) independently across each input channel.

To modify the time scale of streaming audio:

- 1 Create the `audioTimeScaler` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

## Syntax

```
aTS = audioTimeScaler
aTS = audioTimeScaler(speedupFactor)
aTS = audioTimeScaler( ____, 'Name', Value)
```

## Description

`aTS = audioTimeScaler` creates an object, `aTS`, that performs audio time scale modification independently across each input channel over time.

`aTS = audioTimeScaler(speedupFactor)` sets the `SpeedupFactor` property to `speedupFactor`.

`aTS = audioTimeScaler( ____, 'Name', Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `aTS = audioTimeScaler(1.2, 'Window', sqrt(hann(1024, 'periodic')) , 'OverlapLength', 768)` creates an object, `aTS`, that increases the tempo of audio by 1.2 times its original speed using a periodic 1024-point Hann window and a 768-point overlap.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects (MATLAB)*.

### **SpeedupFactor — Speedup factor**

1.1 (default) | positive real scalar

Speedup factor, specified as a positive real scalar.

**Tunable:** Yes

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

### **InputDomain — Domain of input signal**

"Time" (default) | "Frequency"

Domain of the input signal, specified as "Time" or "Frequency".

Data Types: `char` | `string`

### **Window — Analysis window**

`sqrt(hann(512, 'periodic'))` (default) | real vector

Analysis window, specified as a real vector.

---

**Note** If using `audioTimeScaler` with frequency-domain input, you must specify `Window` as the same window used to transform `audioIn` to the frequency domain.

---

Data Types: `single` | `double`

### **OverlapLength** — Overlap length of adjacent analysis windows

384 (default) | nonnegative integer

Overlap length of adjacent analysis windows, specified as a nonnegative integer.

---

**Note** If using `audioTimeScaler` with frequency-domain input, you must specify `OverlapLength` as the same overlap length used to transform `audioIn` to a time-frequency representation.

---

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

### **FFTLength** — FFT length

[ ] (default) | positive scalar integer

FFT length, specified as a positive integer. The default, [ ], means that the FFT length is equal to the number of rows in the input signal.

#### **Dependencies**

To enable this property, set `InputDomain` to `'Time'`.

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

### **LockPhase** — Apply identity phase locking

false (default) | true

Apply identity phase locking, specified as `true` or `false`.

Data Types: `logical`

## Usage

## Syntax

```
audioOut = aTS(audioIn)
```

## Description

`audioOut = aTS(audioIn)` applies time-scale modification to the input, `audioIn`, and returns the time-scaled output, `audioOut`.

## Input Arguments

### **audioIn** — Input audio

column vector | matrix

Input audio, specified as a column vector or matrix. How `audioTimeScaler` interprets `audioIn` depends on the `InputDomain` property.

- If `InputDomain` is set to "Time", `audioIn` must be a real  $N$ -by-1 column vector or  $N$ -by- $C$  matrix. The number of rows,  $N$ , must be equal to or less than the hop length (`size(audioIn,1) <= numel(Window) - OverlapLength`). Columns of a matrix are interpreted as individual channels.
- If `InputDomain` is set to "Frequency", specify `audioIn` as a real or complex  $NFFT$ -by-1 column vector or  $NFFT$ -by- $C$  matrix. The number of rows,  $NFFT$ , is the number of points in the DFT calculation, and is set on the first call to the audio time scaler.  $NFFT$  must be greater than or equal to the window length (`size(audioIn,1) >= numel(Window)`). Columns of a matrix are interpreted as individual channels.

Data Types: `single` | `double`

Complex Number Support: Yes

## Output Arguments

### **audioOut** — Time-stretched audio

column vector | matrix

Time-stretched audio, returned as a column vector or matrix.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the `System` object as the first input argument. For example, to release system resources of a `System` object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

## Examples

### Apply Time Scale Modification to Streaming Audio

To minimize artifacts caused by windowing, create a square root Hann window capable of perfect reconstruction. Use `iscola` to verify the design.

```
win = sqrt(hann(1024,'periodic'));  
overlapLength = 896;  
iscola(win,overlapLength)
```

```
ans = logical  
     1
```

Create an `audioTimeScaler` with a speedup factor of 1.5. Change the value of `alpha` to hear the effect of the speedup factor.

```
alpha = ;  
aTS = audioTimeScaler( ...  
    'SpeedupFactor',alpha, ...  
    'Window',win, ...  
    'OverlapLength',overlapLength);
```

Create a `dsp.AudioFileReader` object to read frames from an audio file. The length of frames input to the audio time scaler must be less than or equal to the analysis hop length defined in `audioTimeScaler`. To minimize buffering, set the samples per frame of the file reader to the analysis hop length.

```
hopLength = numel(aTS.Window) - overlapLength;  
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav', ...  
    'SamplesPerFrame',hopLength);
```



Create an `audioDeviceWriter` to write frames to your audio device. Use the same sample rate as the file reader.

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

In an audio stream loop, read a frame the file, apply time scale modification, and then write a frame to the device.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = aTS(audioIn);
    deviceWriter(audioOut);
end
```

As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(aTS)
```

## Process Frequency-Domain Input

Create a window capable of perfect reconstruction. Use `iscola` to verify the design.

```
win = kbdwin(512);
overlapLength = 256;
iscola(win,overlapLength)

ans = logical
     1
```

Create an `audioTimeScaler` with a speedup factor of 0.8. Set `InputDomain` to "Frequency" and specify the window and overlap length used to transform time-domain audio to the frequency domain. Set `LockPhase` to `true` to increase the fidelity in the time-scaled output.

```
alpha = 0.8;
timeScaleModification = audioTimeScaler( ...
    "SpeedupFactor",alpha, ...
    "InputDomain","Frequency", ...
    "Window",win, ...
```

```
"OverlapLength",overlapLength, ...  
"LockPhase",true);
```

Create a `dsp.AudioFileReader` object to read frames from an audio file. Create a `dsp.STFT` object to perform a short-time Fourier transform on streaming audio. Specify the same window and overlap length you used to create the `audioTimeScaler`. Create an `audioDeviceWriter` object to write frames to your audio device.

```
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3','SamplesPerFrame',fs);  
shortTimeFourierTransform = dsp.STFT('Window',win,'OverlapLength',overlapLength,'FTLength',fftLength);  
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

In an audio stream loop:

- 1 Read a frame from the file.
- 2 Input the frame to the STFT. The `dsp.STFT` object performs buffering.
- 3 Apply time scale modification.
- 4 Write the modified audio to your audio device.

```
while ~isDone(fileReader)  
    x = fileReader();  
    X = shortTimeFourierTransform(x);  
    y = timeScaleModification(X);  
    deviceWriter(y);  
end
```

As a best practice, release your objects once done.

```
release(fileReader)  
release(shortTimeFourierTransform)  
release(timeScaleModification)  
release(deviceWriter)
```

## Algorithms

`audioTimeScaler` uses the same phase vocoder algorithm as `stretchAudio` and is based on the descriptions in [1] and [2].

## References

- [1] Driedger, Johnathan, and Meinard Müller. "A Review of Time-Scale Modification of Music Signals." *Applied Sciences*. Vol. 6, Issue 2, 2016.
- [2] Driedger, Johnathan. "Time-Scale Modification Algorithms for Music Audio Signals." Master's thesis, Saarland University, 2011.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

`audioDataAugmenter` | `shiftPitch` | `stretchAudio`

**Introduced in R2019b**

## parameterTuner

Tune object parameters while streaming

### Syntax

```
H = parameterTuner(obj)
```

### Description

`H = parameterTuner(obj)` creates a parameter tuning UI and returns a figure handle, `H`.

### Examples

#### Tune Parameters of Multiple Objects

`parameterTuner` enables you to graphically tune parameters of multiple objects. In this example, you use a crossover filter to split a signal into multiple subbands and then apply different effects to the subbands.

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card.

```
fileReader = dsp.AudioFileReader('FunkyDrums-48-stereo-25secs.mp3', ...  
                                'PlayCount',2);  
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

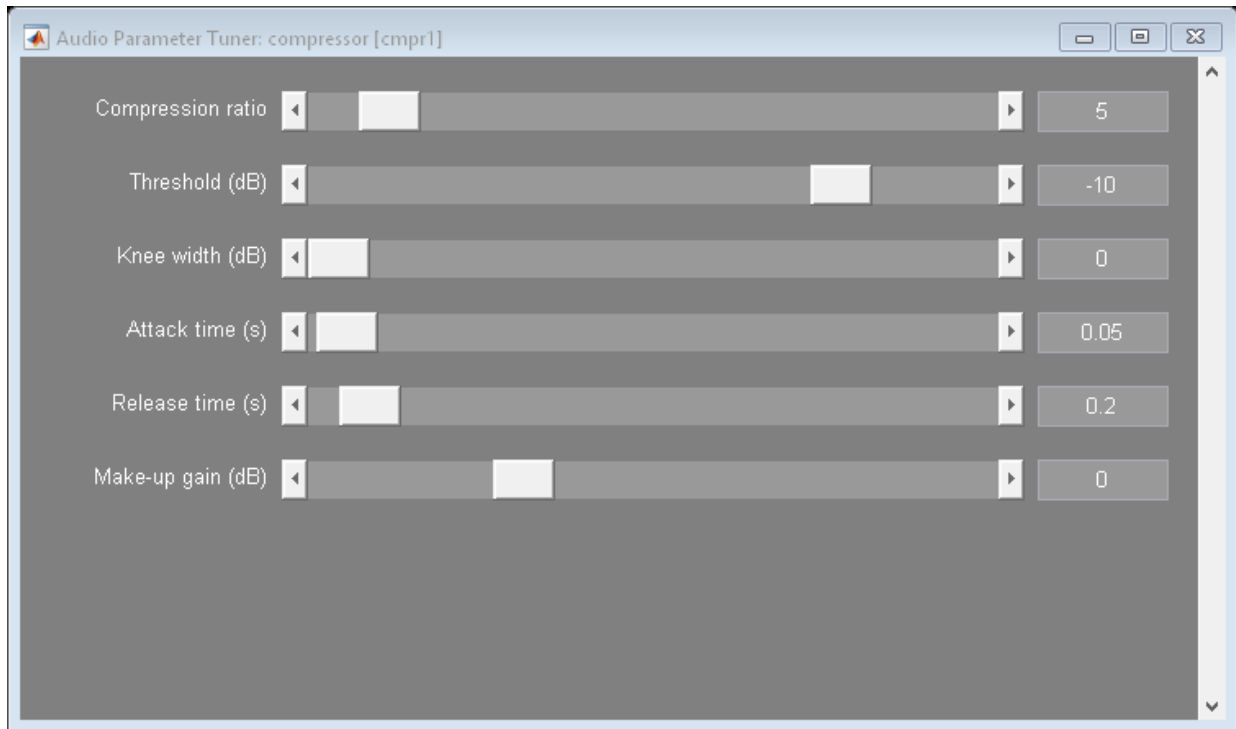
Create a `crossoverFilter` with two crossovers to split the audio into three bands. Call `visualize` to plot the frequency responses of the filters. Call `parameterTuner` to open a UI to tune the crossover frequencies while streaming.

```
xFilt = crossoverFilter('SampleRate',fileReader.SampleRate,'NumCrossovers',2);  
visualize(xFilt)  
parameterTuner(xFilt)
```

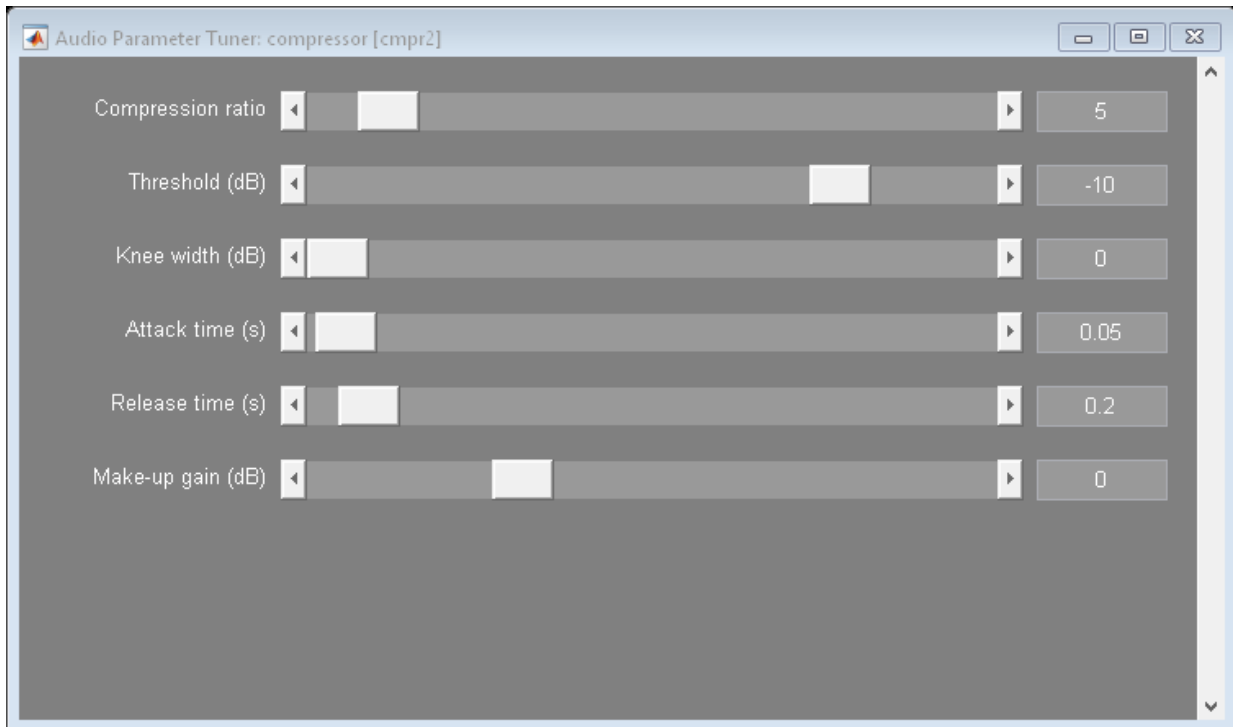


Create two compressor objects to apply dynamic range compression on two of the subbands. Call `visualize` to plot the static characteristic of both of the compressors. Call `parameterTuner` to open UIs to tune the static characteristics.

```
cmpr1 = compressor('SampleRate',fileReader.SampleRate);  
visualize(cmpr1)  
parameterTuner(cmpr1)
```



```
cmpr2 = compressor('SampleRate',fileReader.SampleRate);  
visualize(cmpr2)  
parameterTuner(cmpr2)
```



Create an `audiopluginexample.Chorus` to apply a chorus effect to one of the bands. Call `parameterTuner` to open a UI to tune the chorus plugin parameters.

```
chorus = audiopluginexample.Chorus;  
setSampleRate(chorus, fileReader.SampleRate);  
parameterTuner(chorus)
```

In an audio stream loop:

- 1 Read in a frame of audio from the file.
- 2 Split the audio into three bands using the crossover filter.
- 3 Apply dynamic range compression to the first and second bands.
- 4 Apply a chorus effect to the third band.
- 5 Sum the audio bands.
- 6 Write the frame of audio to your audio device for listening.

```
while ~isDone(fileReader)
    audioIn = fileReader();

    [b1,b2,b3] = xFilt(audioIn);

    b1 = cmpr1(b1);
    b2 = cmpr2(b2);
    b3 = process(chorus,b3);

    audioOut = b1+b2+b3;

    deviceWriter(audioOut);

    drawnow limitrate % Process parameterTuner callbacks
end
```

As a best practice, release your objects once done.

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

#### Tune Hosted Audio Plugin Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card. Use `loadAudioPlugin` to load an equalizer plugin. If you are using a Mac, replace the `.dll` file extension with `.vst`.

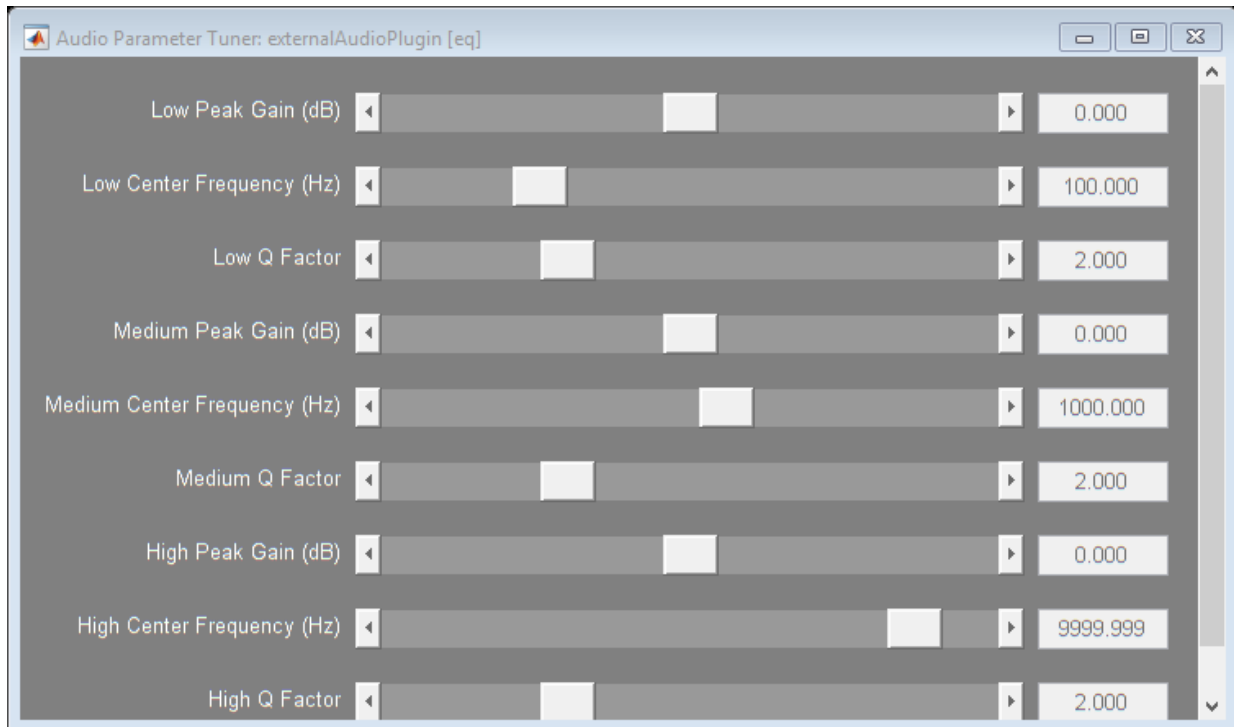
```
fileReader = dsp.AudioFileReader('FunkyDrums-48-stereo-25secs.mp3');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

pluginPath = fullfile(matlabroot,'toolbox/audio/samples/ParametricEqualizer.dll');
eq = loadAudioPlugin(pluginPath);
setSampleRate(eq,fileReader.SampleRate);
```

Call `parameterTuner` to open a UI to tune parameters of the equalizer while streaming.

```
parameterTuner(eq)
```





In an audio stream loop:

- 1 Read in a frame of audio from the file.
- 2 Apply equalization.
- 3 Write the frame of audio to your audio device for listening.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = process(eq,audioIn);
    deviceWriter(audioOut);

    drawnow limitrate % Process parameterTuner callbacks
end
```

As a best practice, release your objects once done.

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

#### Tune MATLAB Audio Plugin Parameters

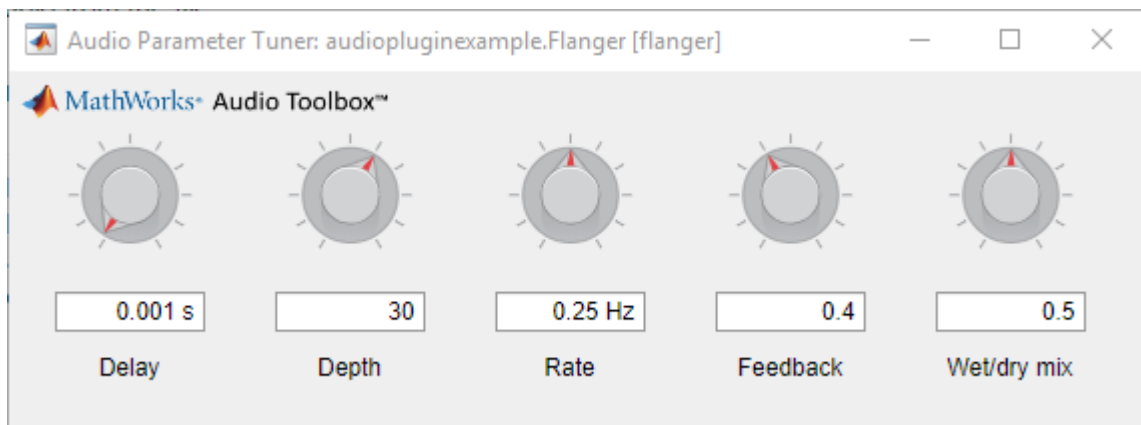
Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card. Create an `audiopluginexample.Flanger` to process the audio data and set the sample rate.

```
fileReader = dsp.AudioFileReader('RockGuitar-16-96-stereo-72secs.flac');  
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

```
flanger = audiopluginexample.Flanger;  
setSampleRate(flanger,fileReader.SampleRate);
```

Call `parameterTuner` to open a UI to tune parameters of the flanger while streaming.

```
parameterTuner(flanger)
```



In an audio stream loop:

- 1 Read in a frame of audio from the file.
- 2 Apply flanging.
- 3 Write the frame of audio to your audio device for listening.

```
while ~isDone(fileReader)  
    audioIn = fileReader();  
    audioOut = process(flanger,audioIn);
```

```
        deviceWriter(audioOut);  
  
        drawnow limitrate % Process parameterTuner callbacks  
end
```

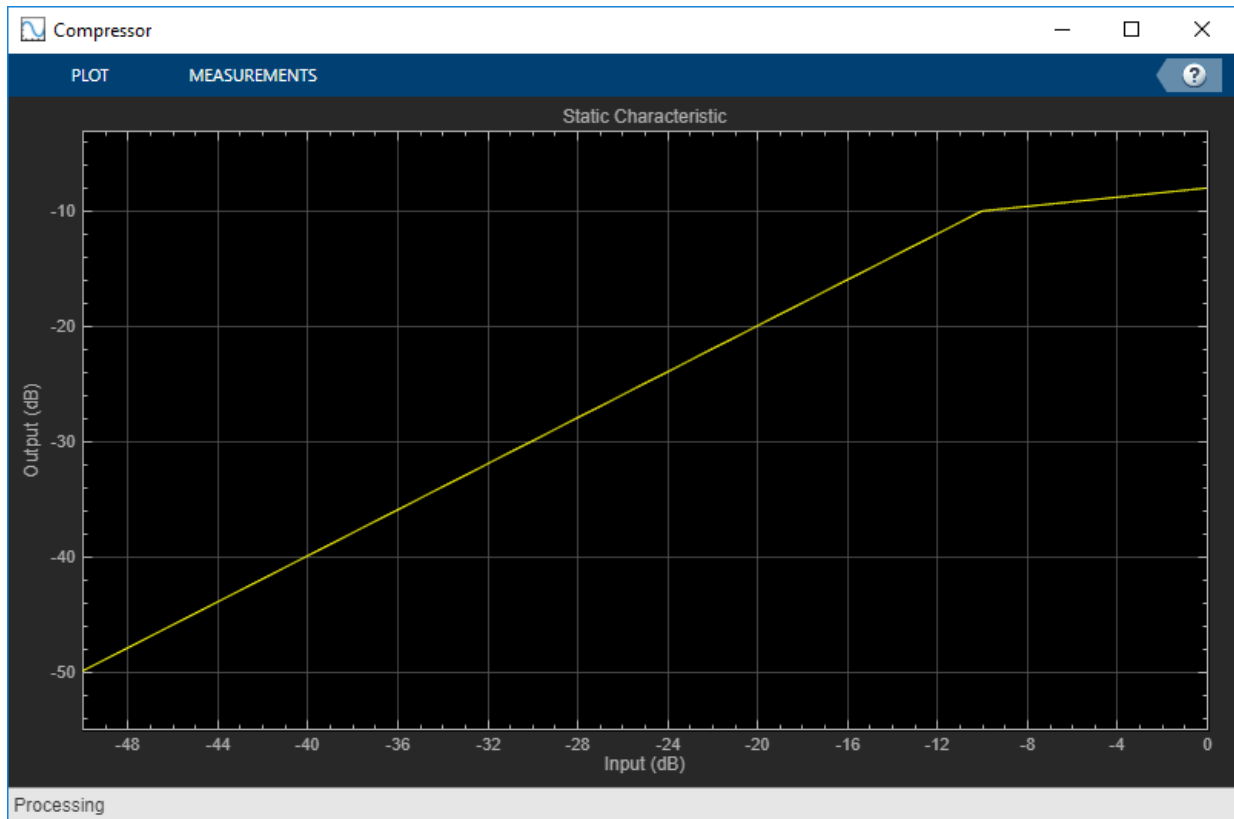
As a best practice, release your objects once done.

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

### **Tune Compressor Parameters**

Create an `dsp.AudioFileReader` to read in audio frame-by-frame. Create a `audioDeviceWriter` to write audio to your sound card. Create a `compressor` to process the audio data. Call `visualize` to plot the static characteristic of the compressor.

```
frameLength = 1024;  
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3', ...  
    'SamplesPerFrame', frameLength);  
deviceWriter = audioDeviceWriter('SampleRate', fileReader.SampleRate);  
  
dRC = compressor('SampleRate', fileReader.SampleRate);  
visualize(dRC)
```

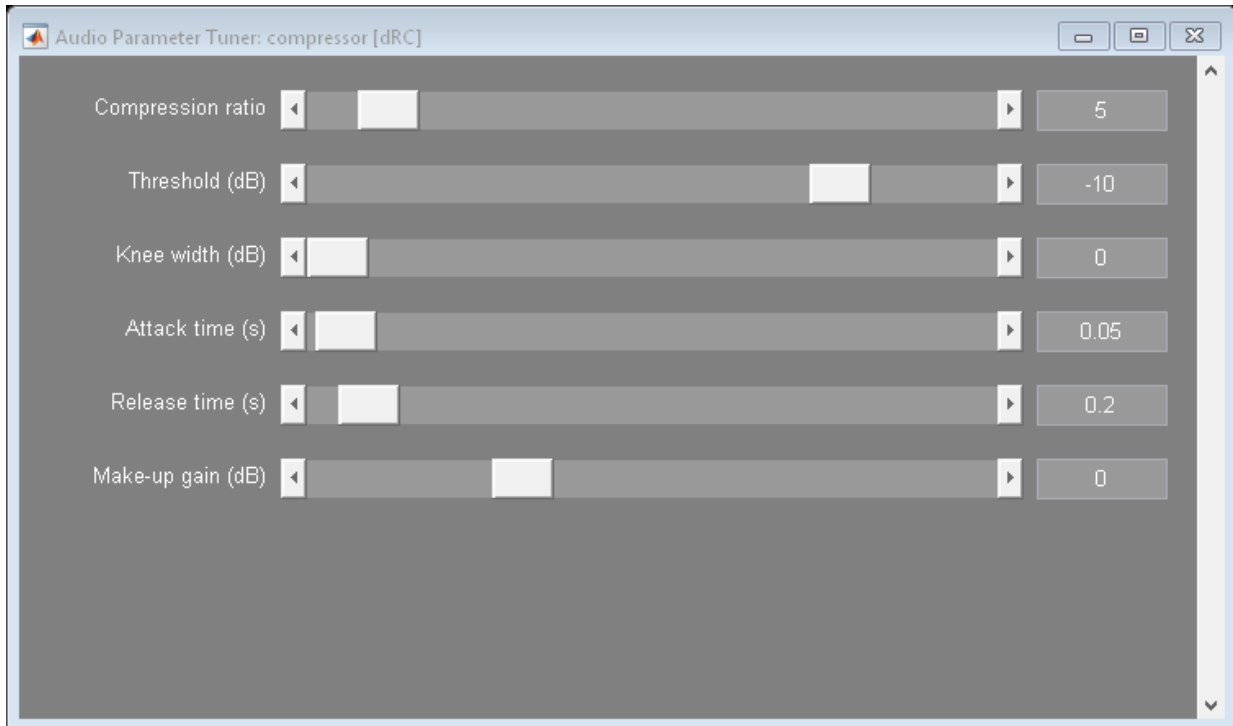


Create a `dsp.TimeScope` to visualize the original and processed audio.

```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpan',1, ...
    'BufferLength',fileReader.SampleRate*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)';
```

Call `parameterTuner` to open a UI to tune parameters of the compressor while streaming.

```
parameterTuner(dRC)
```



In an audio stream loop:

- 1 Read in a frame of audio from the file.
- 2 Apply dynamic range compression.
- 3 Write the frame of audio to your audio device for listening.
- 4 Visualize the original audio, the processed audio, and the gain applied.

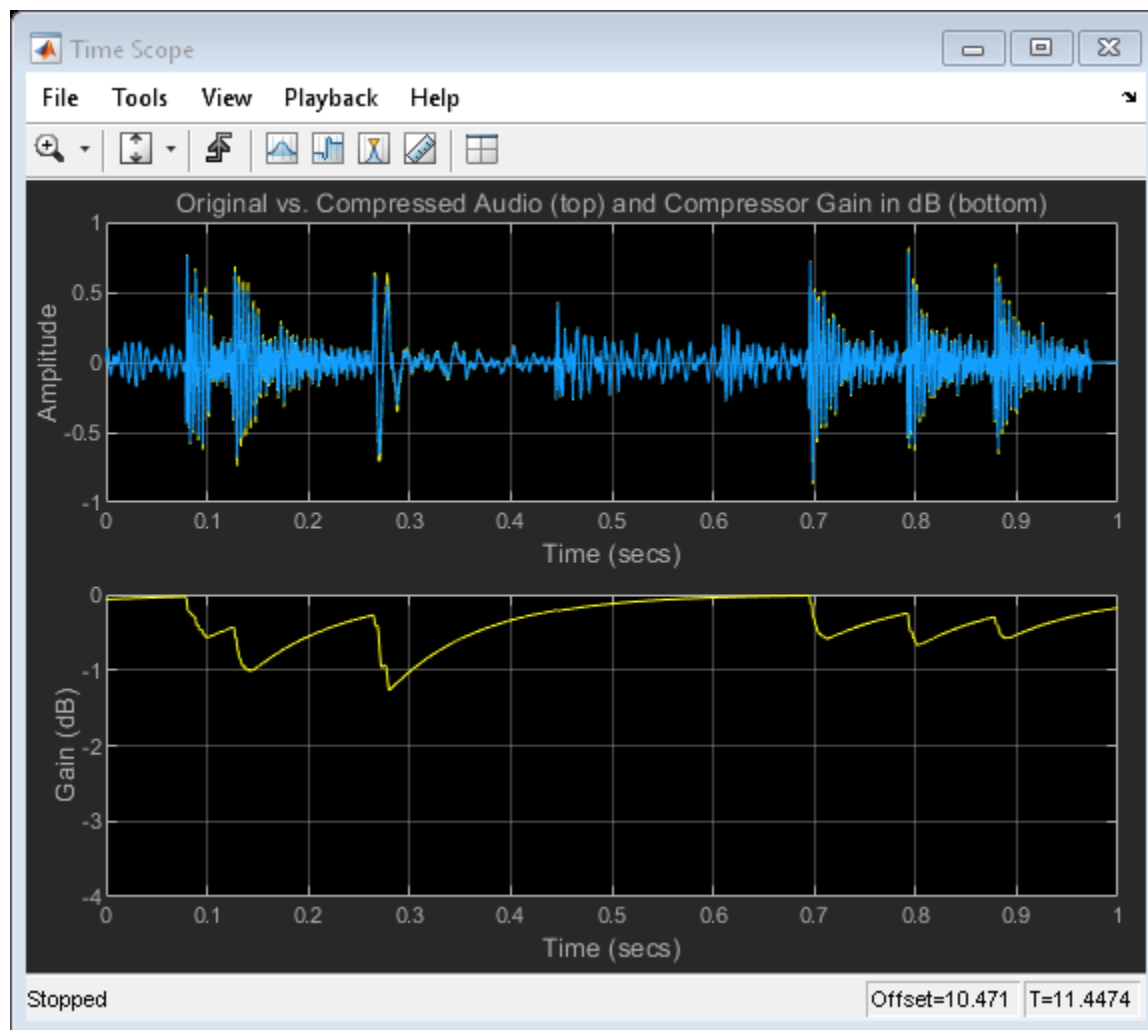
While streaming, tune parameters of the dynamic range compressor and listen to the effect.

```
while ~isDone(fileReader)
    audioIn = fileReader();
```

```
[audioOut,g] = dRC(audioIn);  
deviceWriter(audioOut);  
scope([audioIn(:,1),audioOut(:,1)],g(:,1));  
drawnow limitrate % required to update parameter  
end
```

As a best practice, release your objects once done.

```
release(deviceWriter)  
release(fileReader)  
release(dRC)  
release(scope)
```



## Input Arguments

### **obj** — Object to tune

audioPlugin object | compressor | expander | limiter | noiseGate | octaveFilter | crossoverFilter | multibandParametricEQ | graphicEQ | audioOscillator | wavetableSynthesizer | reverberator

Object to tune, specified as an object that inherits from `audioPlugin` or one of the following Audio Toolbox objects:

- `compressor`
- `expander`
- `limiter`
- `noiseGate`
- `octaveFilter`
- `crossoverFilter`
- `multibandParametricEQ`
- `graphicEQ`
- `audioOscillator`
- `wavetableSynthesizer`
- `reverberator`

## Output Arguments

### **H** — Target figure

Figure object

Target figure, returned as a Figure object.

## See Also

**Audio Test Bench** | `audioPlugin`

**Introduced in R2019a**

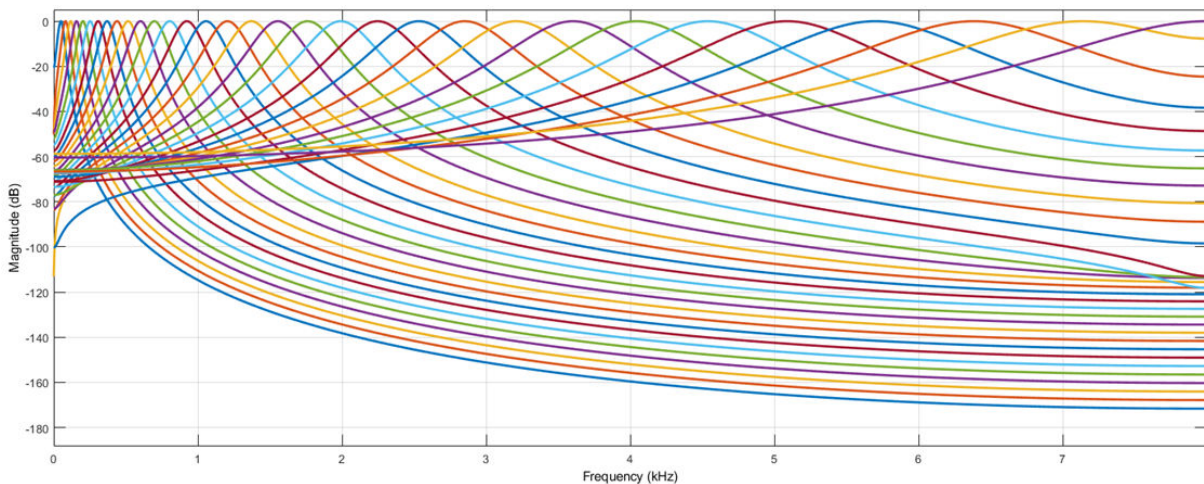


# gammatoneFilterBank

Gammatone filter bank

## Description

`gammatoneFilterBank` decomposes a signal by passing it through a bank of gammatone filters equally spaced on the ERB scale. Gammatone filter banks were designed to model the human auditory system.



To model the human auditory system:

- 1 Create the `gammatoneFilterBank` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

# Creation

## Syntax

```
gammaFiltBank = gammatoneFilterBank  
gammaFiltBank = gammatoneFilterBank(range)  
gammaFiltBank = gammatoneFilterBank(range,numFilts)  
gammaFiltBank = gammatoneFilterBank(range,numFilts,fs)  
gammaFiltBank = gammatoneFilterBank( ____,Name,Value)
```

## Description

`gammaFiltBank = gammatoneFilterBank` returns a gammatone filter bank. The object filters data independently across each input channel over time.

`gammaFiltBank = gammatoneFilterBank(range)` sets the `Range` property to `range`.

`gammaFiltBank = gammatoneFilterBank(range,numFilts)` sets the `NumFilters` property to `numFilts`.

`gammaFiltBank = gammatoneFilterBank(range,numFilts,fs)` sets the `SampleRate` property to `fs`.

`gammaFiltBank = gammatoneFilterBank( ____,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `gammaFiltBank = gammatoneFilterBank([62.5,12e3], 'SampleRate',24e3)` creates a gammatone filter bank, `gammaFiltBank`, with bandpass filters placed between 62.5 Hz and 12 kHz. `gammaFiltBank` operates at a sample rate of 24 kHz.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects \(MATLAB\)](#).

**FrequencyRange — Frequency range of filter bank (Hz)**

[50 8000] (default) | two-element row vector of monotonically increasing values

Frequency range of the filter bank in Hz, specified as a two-element row vector of monotonically increasing values.

**Tunable:** No

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

**NumFilters — Number of filters**

32 (default) | positive integer scalar

Number of filters in the filter bank, specified as a positive integer scalar.

**Tunable:** No

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

**SampleRate — Input sample rate (Hz)**

16000 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: single | double

## Usage

## Syntax

```
audioOut = gammaFiltBank(audioIn)
```

### Description

`audioOut = gammaFiltBank(audioIn)` applies the gammatone filter bank on the input and returns the filtered output.

### Input Arguments

#### **audioIn** — Audio input to filter bank

scalar | vector | matrix

Audio input to the filter bank, specified as a scalar, vector, or matrix. If specified as a matrix, the columns are treated as independent audio channels.

Data Types: `single` | `double`

### Output Arguments

#### **audioOut** — Audio output from filter bank

scalar | vector | matrix | 3-D array

Audio output from the filter bank, returned as a scalar, vector, matrix, or 3-D array. The shape of `audioOut` depends on the shape of `audioIn` and `NumFilters`. If `audioIn` is an  $M$ -by- $N$  matrix, then `audioOut` is returned as an  $M$ -by-`NumFilters`-by- $N$  array. If  $N$  is 1, then `audioOut` is returned as a matrix.

Data Types: `single` | `double`

### Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

#### **Specific to `gammatoneFilterBank`**

<code>fvtool</code>	Visualize filter bank
<code>freqz</code>	Compute frequency response
<code>getCenterFrequencies</code>	Center frequencies of filters

getBandwidths	Get filter bandwidths
coeffs	Get filter coefficients
info	Get filter information

## Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

## Examples

### Apply Gammatone Filterbank

Create a default gammatone filter bank for a 16 kHz sample rate.

```
fs = 16e3;  
gammaFiltBank = gammatoneFilterBank('SampleRate', fs)
```

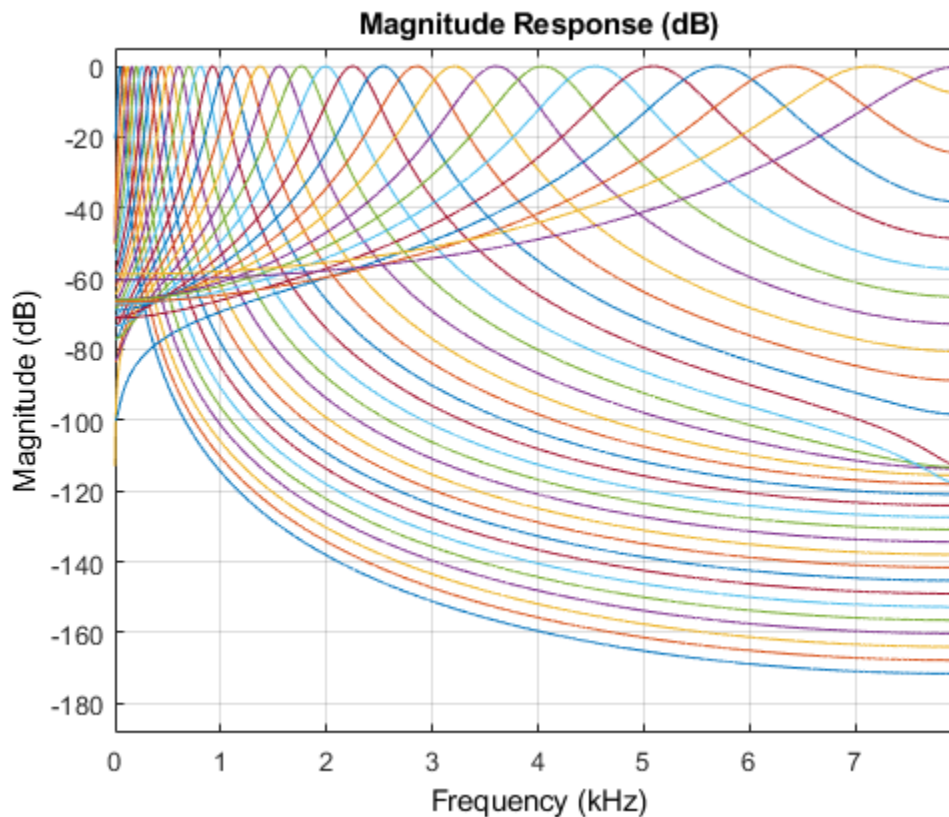
```
gammaFiltBank =
```

```
gammatoneFilterBank with properties:
```

```
FrequencyRange: [50 8000]  
NumFilters: 32  
SampleRate: 16000
```

Use `fvtool` to visualize the response of the filter bank.

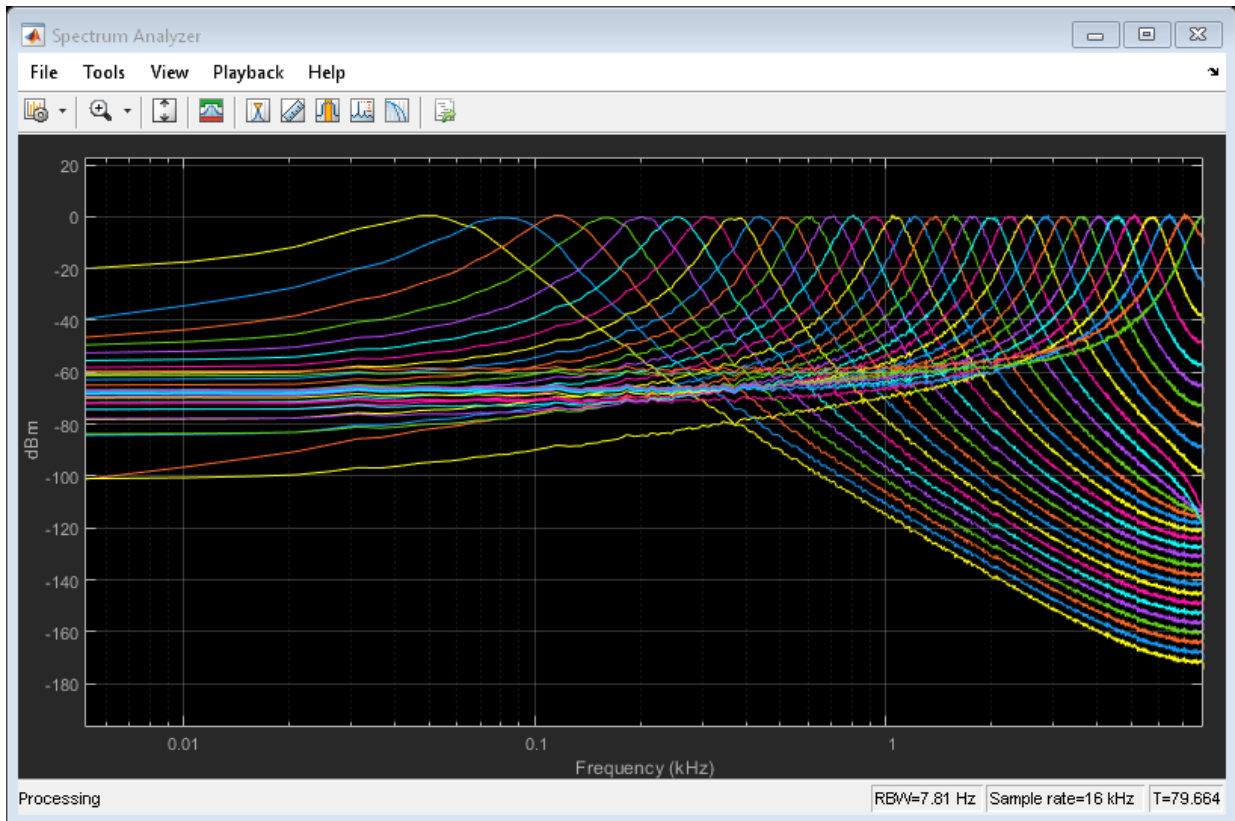
```
fvtool(gammaFiltBank)
```



Process white Gaussian noise through the filter bank. Use a spectrum analyzer to view the spectrum of the filter outputs.

```
sa = dsp.SpectrumAnalyzer('SampleRate',16e3,...
    'PlotAsTwoSidedSpectrum',false,...
    'FrequencyScale','log',...
    'SpectralAverages',100);

for i = 1:5000
    x = randn(256,1);
    y = gammaFiltBank(x);
    sa(y);
end
```



## Analysis and Synthesis

This example illustrates a nonoptimal but simple approach to analysis and synthesis using `gammatoneFilterBank`.

Read in an audio file and listen to its contents.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
sound(audioIn,fs)
```

Create a default `gammatoneFilterBank`. The default frequency range of the filter bank is 50 to 8000 Hz. Frequencies outside of this range are attenuated in the reconstructed signal.

```
gammaFiltBank = gammatoneFilterBank('SampleRate', fs)
gammaFiltBank =
    gammatoneFilterBank with properties:
        FrequencyRange: [50 8000]
        NumFilters: 32
        SampleRate: 44100
```

Pass the audio signal through the gammatone filter bank. The output is 32 channels, where the number of channels is set by the `NumFilters` property of the `gammatoneFilterBank`.

```
audioOut = gammaFiltBank(audioIn);
[N,numChannels] = size(audioOut)
N = 685056
numChannels = 32
```

To reconstruct the original signal, sum the channels. Listen to the result.

```
reconstructedSignal = sum(audioOut,2);
sound(reconstructedSignal,fs)
```

The gammatone filter bank introduced various group delays for the output channels, which results in poor reconstruction. To compensate for the group delay, remove the beginning delay from the individual channels and zero-pad the ends of the channels. Use `info` to get the group delays. Listen to the group delay-compensated reconstruction.

```
infoStruct = info(gammaFiltBank);
groupDelay = round(infoStruct.GroupDelays); % round for simplicity
audioPadded = [audioOut;zeros(max(groupDelay),gammaFiltBank.NumFilters)];
for i = 1:gammaFiltBank.NumFilters
    audioOut(:,i) = audioPadded(groupDelay(i)+1:N+groupDelay(i),i);
end
reconstructedSignal = sum(audioOut,2);
sound(reconstructedSignal,fs)
```



## Create Gammatone Spectrogram

Read in an audio signal and convert it to mono for easy visualization.

```
[audio,fs] = audioread('WaveGuideLoopOne-24-96-stereo-10secs.aif');
audio = mean(audio,2);
```

Create a gammatoneFilterBank with 64 filters that span the range 62.5 to 20,000 Hz. Pass the audio signal through the filter bank.

```
gammaFiltBank = gammatoneFilterBank('SampleRate',fs, ...
    'NumFilters',64, ...
    'FrequencyRange',[62.5,20e3]);
```

```
audioOut = gammaFiltBank(audio);
```

Calculate the energy-per-band using 50 ms windows with 25 ms overlap. Use dsp.AsyncBuffer to divide the signals into overlapped windows and then to log the RMS value of each window for each channel.

```
samplesPerFrame = round(0.05*fs);
samplesOverlap = round(0.025*fs);
```

```
buff = dsp.AsyncBuffer(numel(audio));
write(buff,audioOut.^2);
```

```
sink = dsp.AsyncBuffer(numel(audio));
```

```
while buff.NumUnreadSamples > 0
    currentFrame = read(buff,samplesPerFrame,samplesOverlap);
    write(sink,mean(currentFrame,1));
end
```

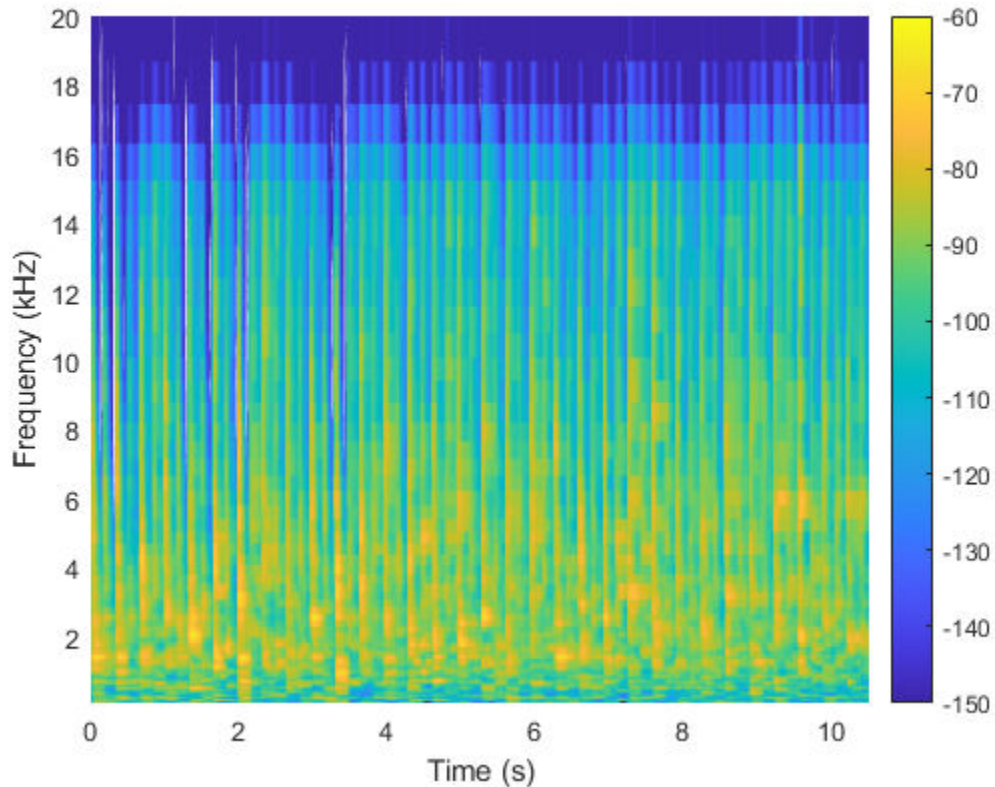
Convert the energy values to dB. Plot the energy-per-band over time.

```
gammatoneSpec = read(sink);
D = 20*log10(gammatoneSpec');
```

```
timeVector = ((samplesPerFrame-samplesOverlap)/fs)*(0:size(D,2)-1);
cf = getCenterFrequencies(gammaFiltBank)./1e3;
```

```
surf(timeVector,cf,D,'EdgeColor','none')
axis([timeVector(1) timeVector(end) cf(1) cf(end)])
view([0 90])
caxis([-150,-60])
```

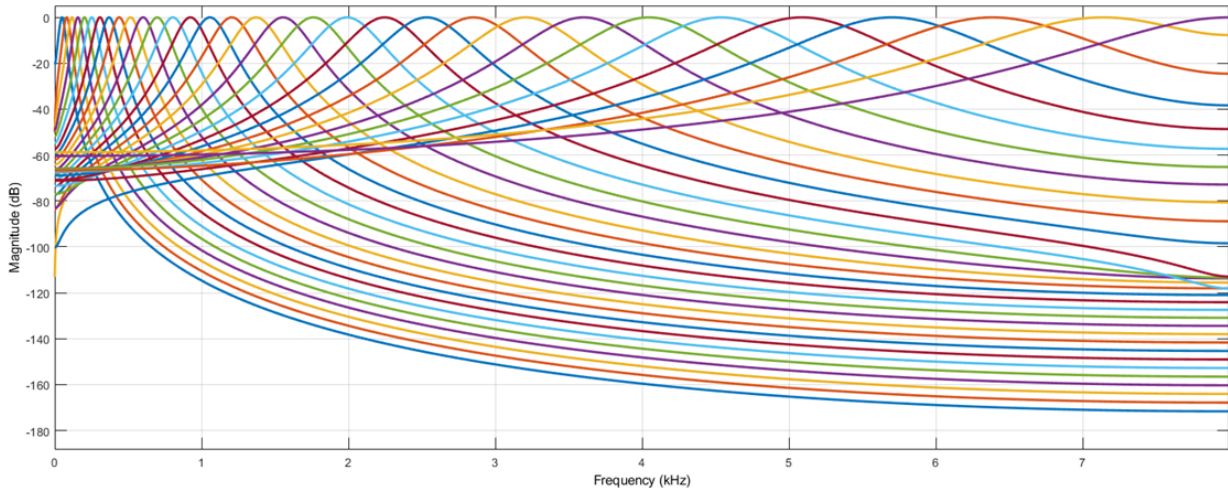
```
colorbar  
xlabel('Time (s)')  
ylabel('Frequency (kHz)')
```



## Algorithms

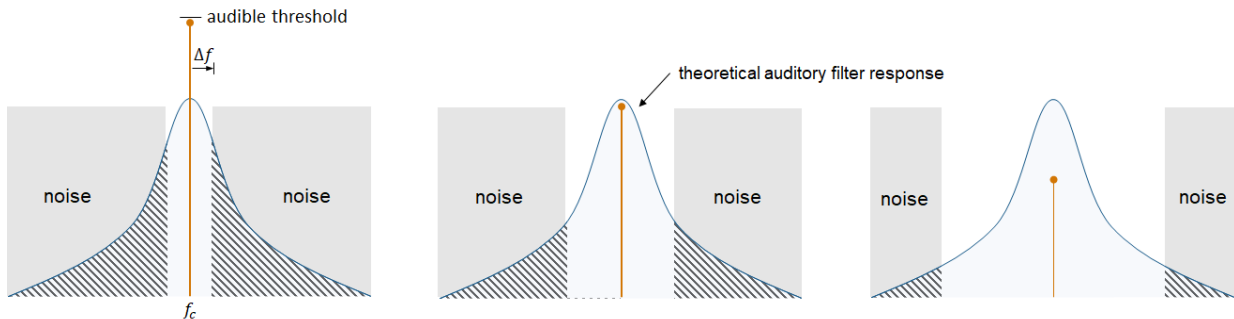
A gammatone filter bank is often used as the front end of a cochlea simulation, which transforms complex sounds into a multichannel activity pattern like that observed in the auditory nerve.[2] The `gammatoneFilterBank` follows the algorithm described in [1]. The algorithm is an implementation of an idea proposed in [2]. The design of the gammatone filter bank can be described in two parts: the filter shape (gammatone) and

the frequency scale. The equivalent rectangular bandwidth (ERB) scale defines the relative spacing and bandwidth of the gammatone filters. The derivation of the ERB scale also provides an estimate of the auditory filter response which closely resembles the gammatone filter.

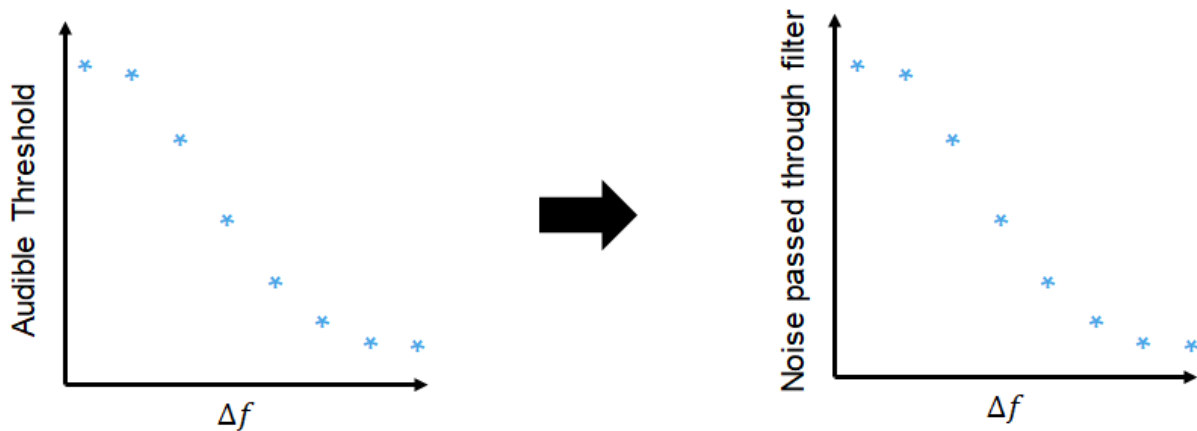


## Frequency Scale

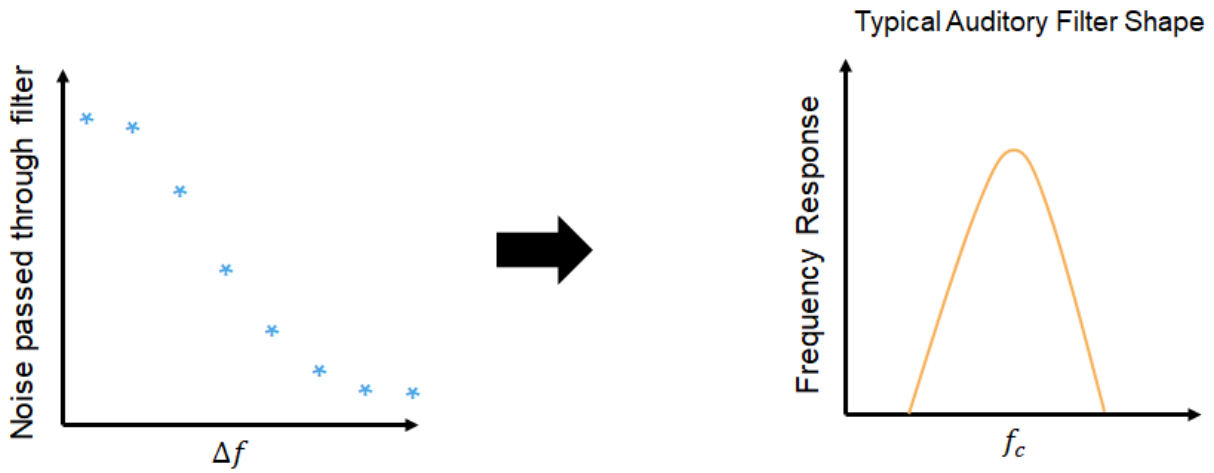
The ERB scale was determined using the notched-noise masking method. This method involves a listening test wherein notched noise is centered on a tone. The power of the tone is tuned, and the audible threshold (the power required for the tone to be heard) is recorded. The experiment is repeated for different notch widths and center frequencies.



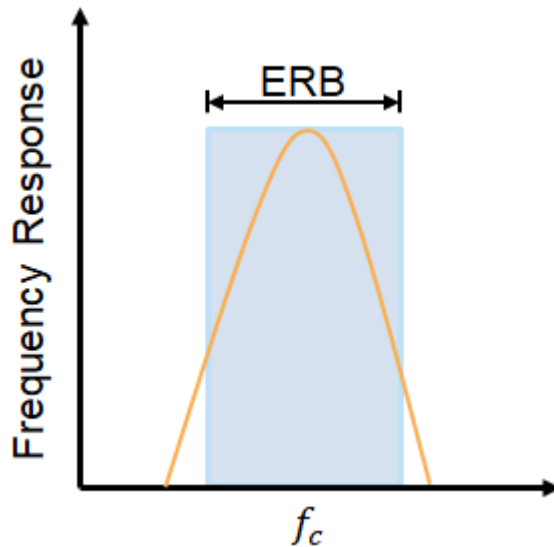
The notched-noise method assumes the audible threshold corresponds to a constant signal-to-masker ratio at the output of the theoretical auditory filter. That is, the ratio of the power of the  $f_c$  tone and the shaded area is constant. Therefore, the relationship between the audible threshold and  $2\Delta f$  (the notch bandwidth) is linearly related to the relationship between the noise passed through the filter and  $2\Delta f$ .



The derivative of the function relating  $\Delta f$  to the noise passed through the filter estimates the auditory filter shape. Because  $\Delta f$  has an inverse relationship with the noise power passed through the filter, the derivative of the function must be multiplied by  $-1$ . The resulting auditory filter shape is usually approximated as a roex filter.



The equivalent rectangular bandwidth of the auditory filter is defined as the width of a rectangular filter required to pass the same noise power as the auditory filter.



[4] defines ERB as a function of center frequency for young listeners with normal hearing and a moderate noise level:

$$\text{ERB} = 24.7(0.00437f_c + 1)$$

The ERB scale (ERBs) is an extension of the relationship between ERB and center frequency, derived by integrating the reciprocal of the ERB function:

$$\text{ERBs} = 21.4\log_{10}(0.00437f + 1)$$

To design a gammatone filter bank, [2] suggests distributing the center frequencies of the filters in proportion to their bandwidth. To accomplish this, `gammatoneFilterBank` defines the center frequencies as linearly spaced on the ERB scale, covering the specified frequency range with the desired number of filters. You can specify the frequency range and desired number of filters using the `FrequencyRange` and `NumFilters` properties.

## Gammatone Filter

The gammatone filter was introduced in [3]. The continuous impulse response is:

$$g(t) = at^{n-1}e^{-2\pi bt}\cos(2\pi f_c t + \phi)$$

where

- $a$  -- amplitude factor
- $t$  -- time in seconds
- $n$  -- filter order (set to four to model human hearing)
- $f_c$  -- center frequency
- $b$  -- bandwidth, set to  $1.019 \cdot \text{hz2erb}(f_c)$ .
- $\phi$  -- phase factor

The gammatone filter is similar to the roex filter derived from the notched-noise experiment. `gammatoneFilterBank` implements the digital filter as a cascade of four second-order sections, as described in [1].

## References

- [1] Slaney, Malcolm. "An Efficient Implementation of the Patterson-Holdworth Auditory Filter Bank." Apple Computer Technical Report 35, 1993.

- [2] Patterson, R.d., K. Robinson, J. Holdsworth, D. Mckeown, C. Zhang, and M. Allerhand. "Complex Sounds and Auditory Images." *Auditory Physiology and Perception*. 1992, pp. 429-446.
- [3] Aertsen, A. M. H. J., and P. I. M. Johannesma. "Spectro-temporal Receptive Fields of Auditory Neurons in the Grassfrog." *Biological Cybernetics*. Vol. 38, Issue 4, 1980, pp. 223-234.
- [4] Glasberg, Brian R., and Brian CJ Moore. "Derivation of Auditory Filter Shapes from Notched-Noise Data." *Hearing Research*. Vol. 47. Issue 1-2, 1990, pp. 103 -138.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

`crossoverFilter` | `octaveFilterBank`

**Introduced in R2019a**

## coeffs

Get filter coefficients

## Syntax

```
[B,A] = coeffs(obj)
```

## Description

`[B,A] = coeffs(obj)` returns the coefficients of the filters created by `obj`.

## Examples

### Get graphicEQ Coefficients

#### Cascade Structure

Create a `graphicEQ` and then call `coeffs` to get its coefficients. The coefficients are returned as second-order sections. The dimensions of `B` are 3-by-(`M * EQOrder / 2`), where `M` is the number of bandpass equalizers. The dimensions of `A` are 2-by-(`M * EQOrder / 2`). The leading unity coefficient is not returned.

```
fs = 44.1e3;  
x = 0.1*randn(fs*5,1);  
equalizer = graphicEQ('SampleRate',fs, ...  
                     'Gains',[-10,-10,10,10,-10,-10,10,10,-10,-10], ...  
                     'EQOrder',2);
```

```
[B,A] = coeffs(equalizer);
```

Compare using `filter` with coefficients `B` and `A` and the output of `graphicEQ`. For simplicity, compare output channel five only.

```
channelToCompare = 5;  
y = x;
```

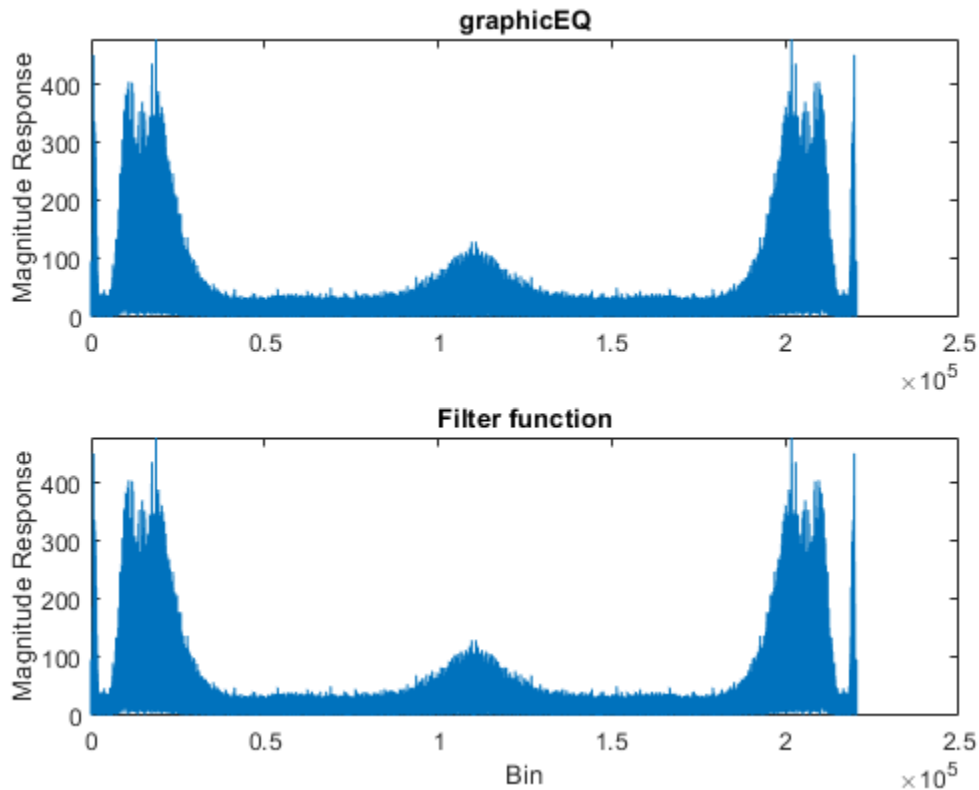


```
for section = 1:equalizer.EQOrder/2
    for i = 1:numel(equalizer.Gains)
        y = filter(B(:,i*section),[1;A(:,i*section)],y);
    end
end
audioOut_filter = y;

audioOut = equalizer(x);

subplot(2,1,1)
plot(abs(fft(audioOut)))
title('graphicEQ')
ylabel('Magnitude Response')

subplot(2,1,2)
plot(abs(fft(audioOut_filter)))
title('Filter function')
xlabel('Bin')
ylabel('Magnitude Response')
```



#### Get gammatoneFilterBank Coefficients

Create the default `gammatoneFilterBank`, and then call `coeffs` to get its coefficients. Each gammatone filter is an eighth-order IIR filter composed of a cascade of four second-order sections. The size of `B` is 4-by-3-by-`NumFilters`. The size of `A` is 4-by-2-by-`NumFilters`.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
gammaFiltBank = gammatoneFilterBank('SampleRate',fs);  
[B,A] = coeffs(gammaFiltBank);
```

Compare using `filter` with coefficients `B` and `A` and the output of `gammaFiltBank`. For simplicity, compare output channel eight only.

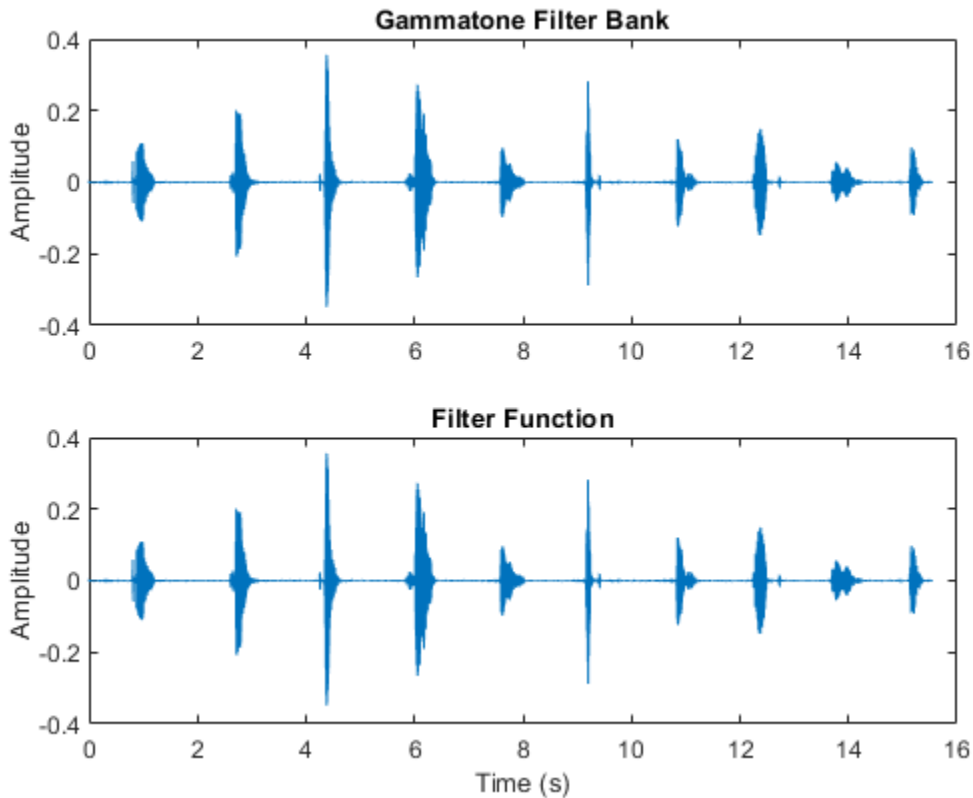
```
channelToCompare = 8;
y1 = filter(B(1,:,channelToCompare),[1,A(1,:,channelToCompare)],audioIn);
y2 = filter(B(2,:,channelToCompare),[1,A(2,:,channelToCompare)],y1);
y3 = filter(B(3,:,channelToCompare),[1,A(3,:,channelToCompare)],y2);
audioOut_filter = filter(B(4,:,channelToCompare),[1,A(4,:,channelToCompare)],y3);

audioOut = gammaFiltBank(audioIn);

t = (0:(size(audioOut,1)-1))/fs;

subplot(2,1,1)
plot(t,audioOut(:,channelToCompare))
title('Gammatone Filter Bank')
ylabel('Amplitude')

subplot(2,1,2)
plot(t,audioOut_filter)
title('Filter Function')
xlabel('Time (s)')
ylabel('Amplitude')
```



#### Get octaveFilterBank Coefficients

Create the default `octaveFilterBank`, and then call `coeffs` to get its coefficients. The coefficients are returned as fourth-order sections. The dimensions of `B` and `A` are  $T$ -by- $5$ -by- $M$ , where  $T$  is the number of sections and  $M$  is the number of filters.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');  
octFiltBank = octaveFilterBank('SampleRate',fs);  
[B,A] = coeffs(octFiltBank);
```

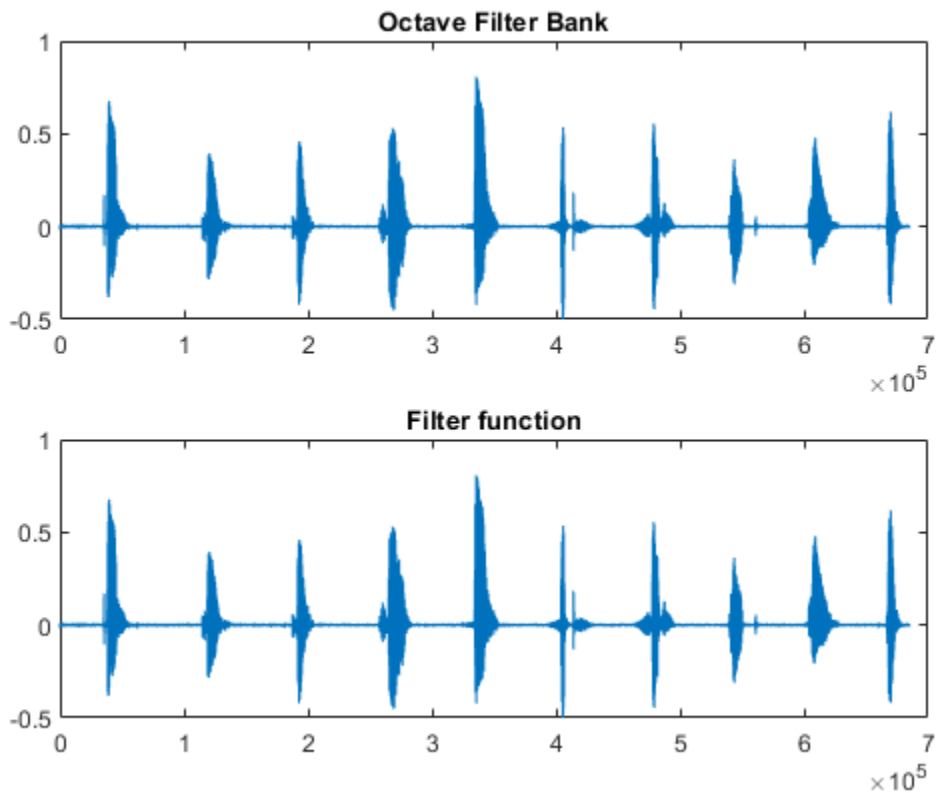
Compare using `filter` with coefficients `B` and `A` and the output of `octaveFilterBank`. For simplicity, compare output channel eight only.

```
channelToCompare = 5;
y1 = filter(B(1,:,channelToCompare),A(1,:,channelToCompare),audioIn);
audioOut_filter = y1;

audioOut = octaveFilterBank(audioIn);

subplot(2,1,1)
plot(audioOut(:,channelToCompare))
title('Octave Filter Bank')

subplot(2,1,2)
plot(audioOut_filter)
title('Filter function')
```



## Input Arguments

**obj** — Object to get filter coefficients from

gammatoneFilterBank | octaveFilterBank | graphicEQ

Object to get filter coefficients from, specified as an object of gammatoneFilterBank, octaveFilterBank, or graphicEQ.

## Output Arguments

### **B — Numerator filter coefficients**

matrix | 3-D array

Numerator filter coefficients, returned as a 2-D matrix or 3-D array, depending on `obj`.

Data Types: `single` | `double`

### **A — Denominator filter coefficients**

matrix | 3-D array

Numerator filter coefficients, returned as a 2-D matrix or 3-D array, depending on `obj`.

Data Types: `single` | `double`

## See Also

`gammatoneFilterBank` | `graphicEQ` | `octaveFilterBank`

**Introduced in R2019a**

## freqz

Compute frequency response

### Syntax

```
[H,f] = freqz(obj)
[H,f] = freqz(obj,ind)
[H,f] = freqz( ___,Name,Value)
freqz( ___ )
```

### Description

`[H,f] = freqz(obj)` returns a matrix of complex frequency responses for each filter designed by `obj`.

`[H,f] = freqz(obj,ind)` returns the frequency response of filters with indices corresponding to the elements in vector `ind`.

`[H,f] = freqz( ___,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

`freqz( ___ )` with no output arguments plots the frequency response of the filter bank.

### Examples

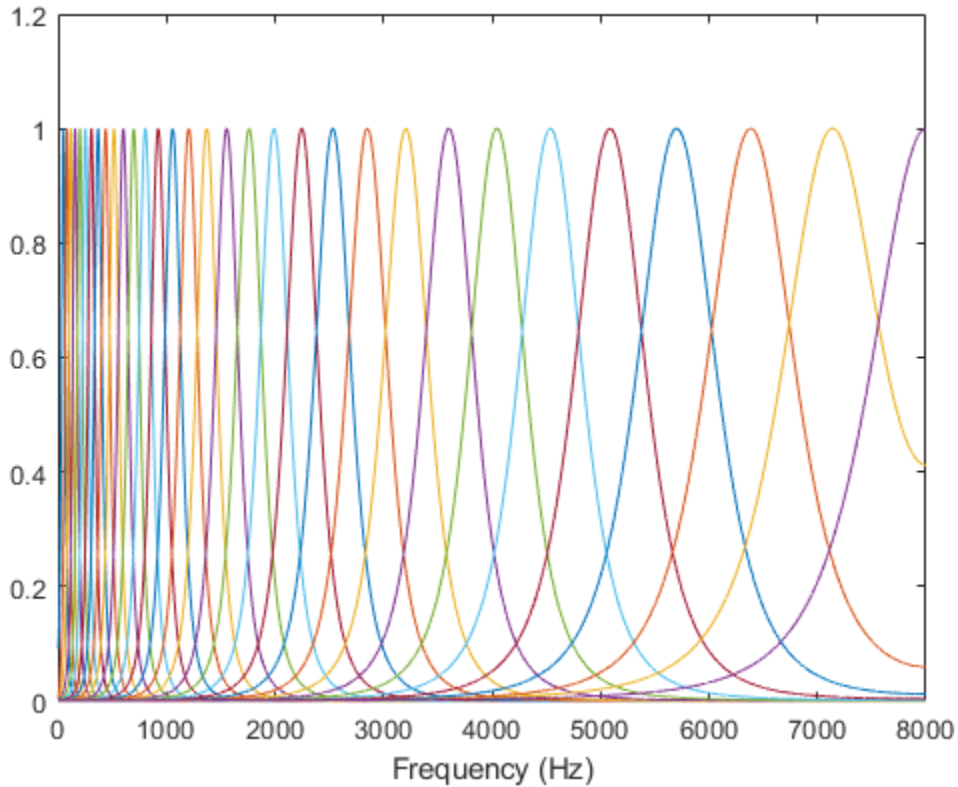
#### Frequency Response of `gammatoneFilterBank`

Create a `gammatoneFilterBank` object. Call `freqz` to get the complex frequency response, `H`, of the filter bank and a vector of frequencies, `f`, at which the response is calculated. Plot the magnitude frequency response of the filter bank.

```
gammaFiltBank = gammatoneFilterBank;
[H,f] = freqz(gammaFiltBank);
```



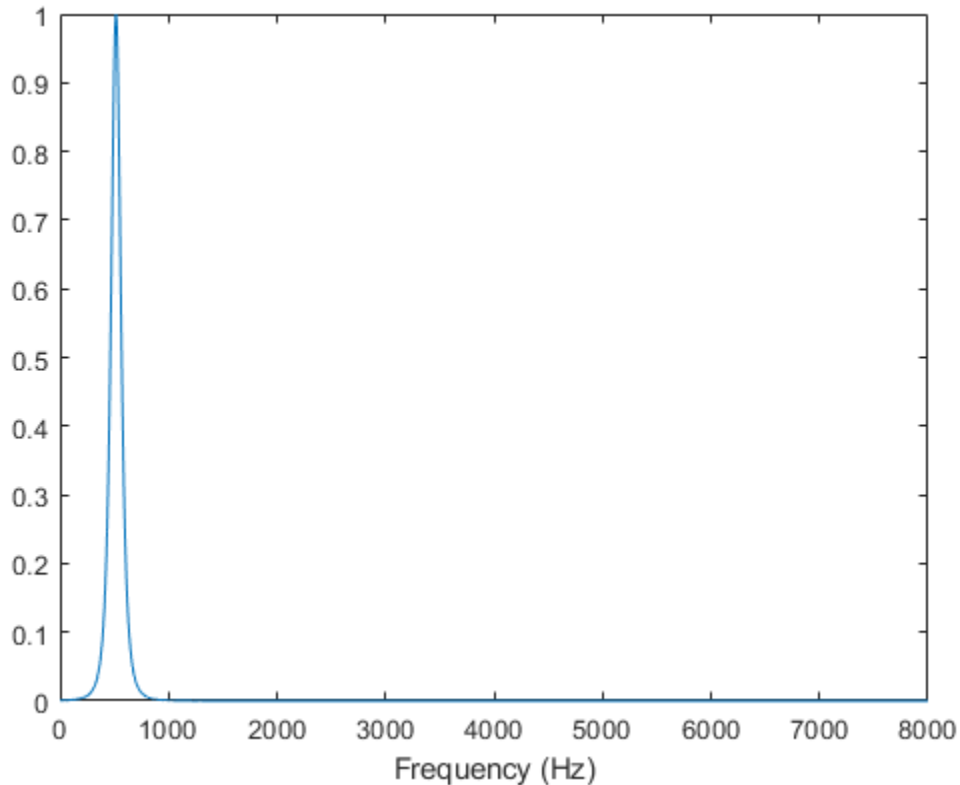
```
plot(f,abs(H))  
xlabel('Frequency (Hz)')
```



To get the frequency response of a subset of filters in the filter bank, specify the second argument as a row vector of indices between one and the number of filters in the filter bank. Get the frequency response of the 10th filter in the filter bank and plot the magnitude frequency response.

```
[H,f] = freqz(gammaFiltBank,10);
```

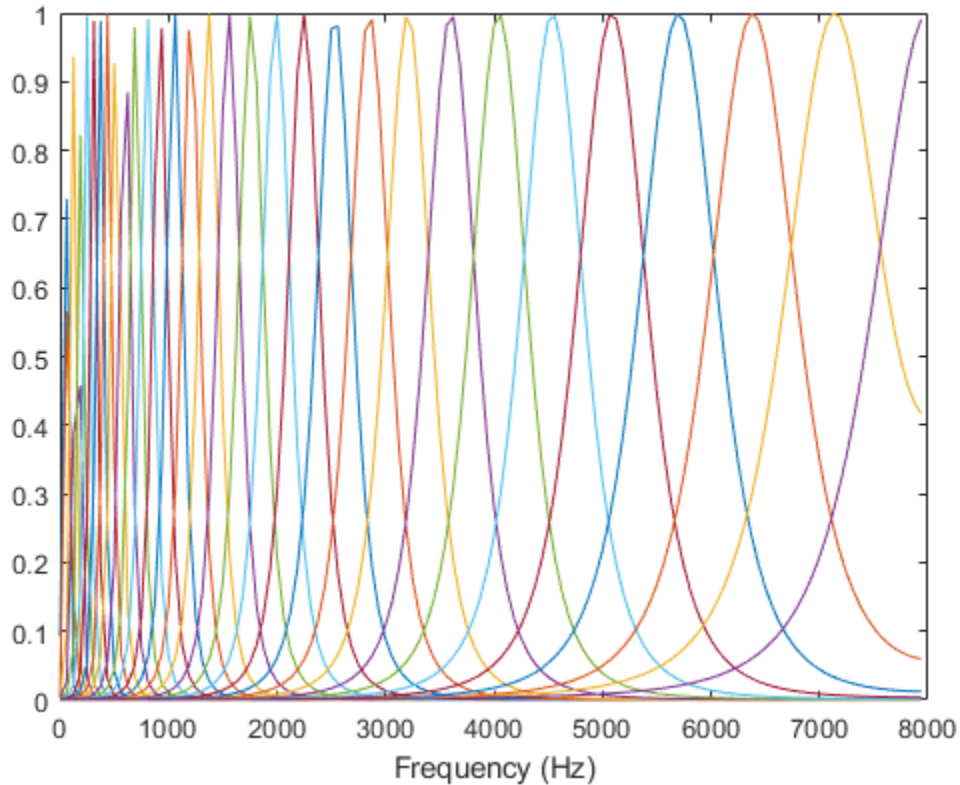
```
plot(f,abs(H))  
xlabel('Frequency (Hz)')
```



To specify the number of FFT points used to compute the frequency response, use the `NFFT` name-value pair. Specify that the frequency response is calculated using a 128-point FFT. Plot the magnitude frequency response.

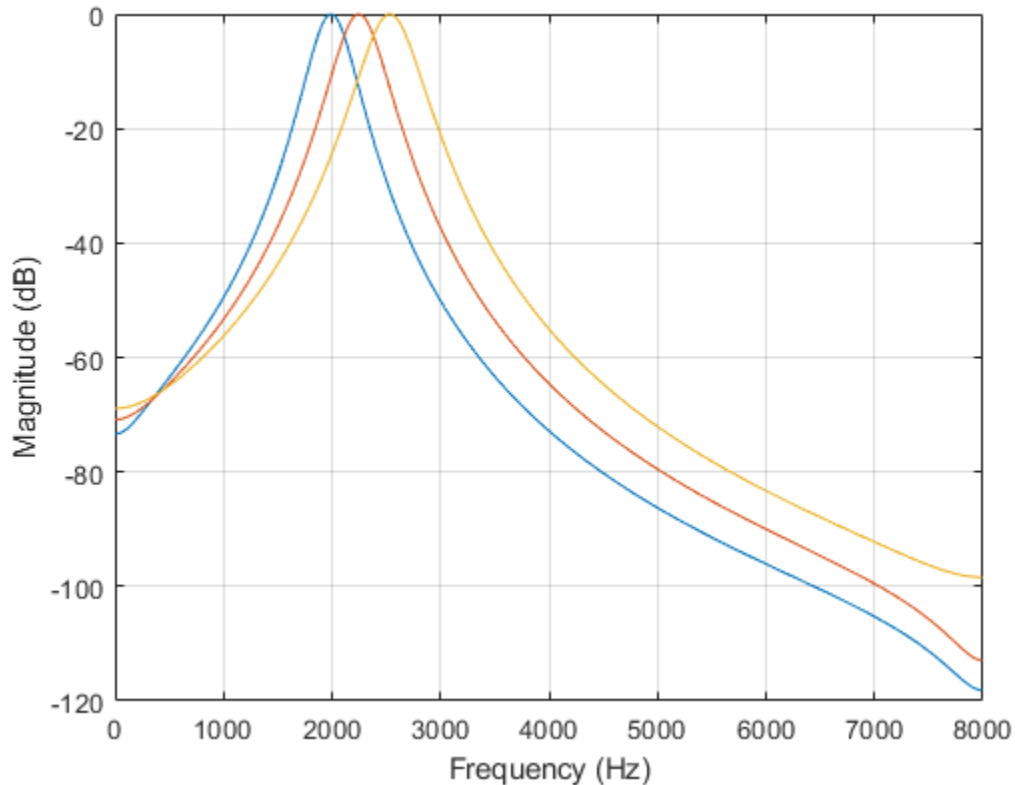
```
[H,f] = freqz(gammaFiltBank, 'NFFT',128);
```

```
plot(f,abs(H))  
xlabel('Frequency (Hz)')
```



To visualize the magnitude frequency response only, call `freqz` without any output arguments. Plot the magnitude frequency response, in dB, of filters 20, 21, and 22 using a 1024-point DFT.

```
freqz(gammaFiltBank,[20,21,22],'NFFT',1024)
```



#### Frequency Response of octaveFilterBank

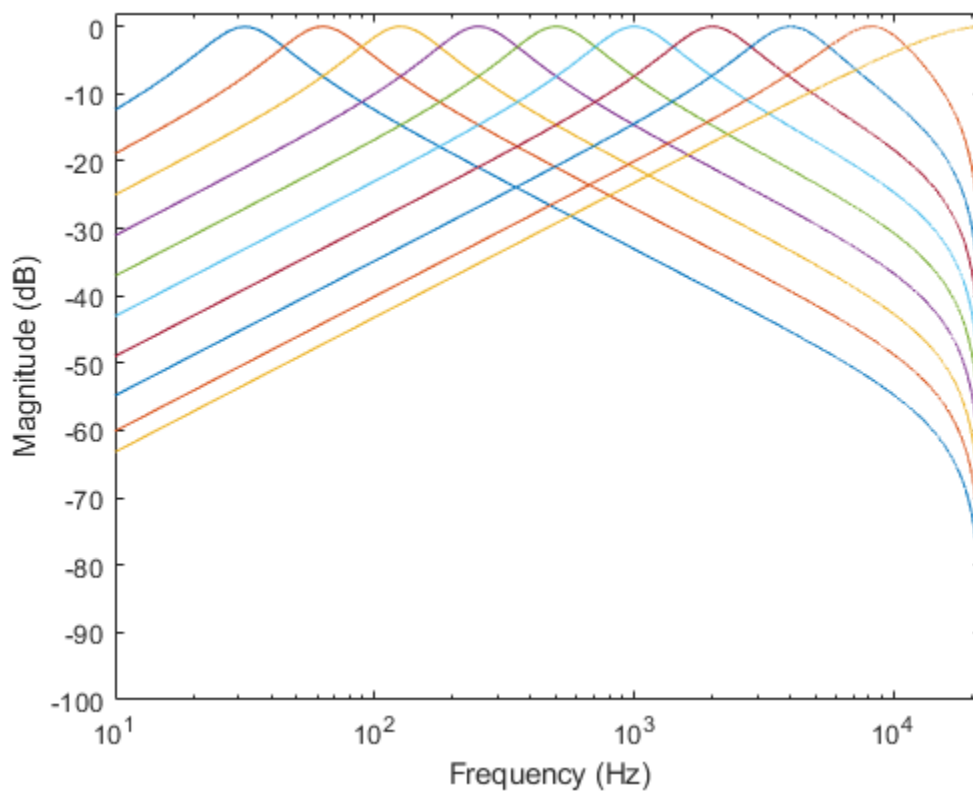
Create an `octaveFilterBank` object. Call `freqz` to get the complex frequency response, `H`, of the filter bank and a vector of frequencies, `f`, at which the response is calculated. Plot the magnitude frequency response in dB.

```
octFiltBank = octaveFilterBank;  
[H,f] = freqz(octFiltBank);  
  
plot(f,20*log10(abs(H)))  
xlabel('Frequency (Hz)')
```

```

ylabel('Magnitude (dB)')
set(gca,'XScale','log')
axis([10 octFiltBank.SampleRate/2 -100 2])

```



To get the frequency response of a subset of filters in the filter bank, specify the second argument as a row vector of indices between one and the number of filters in the filter bank. Get the frequency response of the 5th filter in the filter bank and plot the magnitude frequency response in dB.

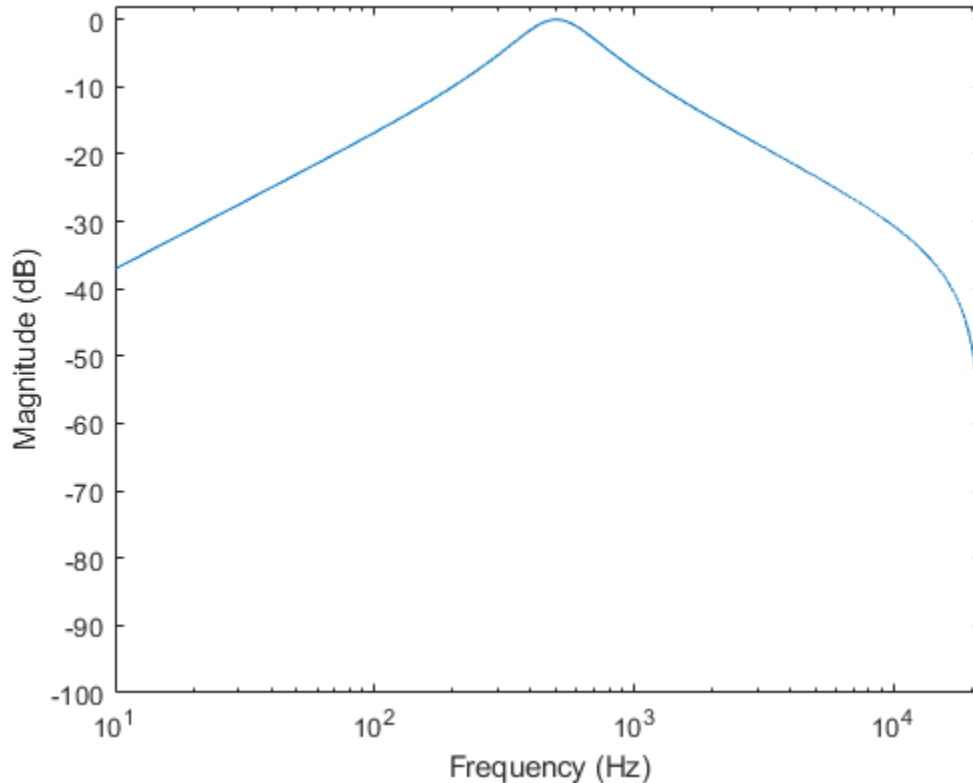
```

[H,f] = freqz(octFiltBank,5);

plot(f,20*log10(abs(H)))
xlabel('Frequency (Hz)')
ylabel('Magnitude (dB)')

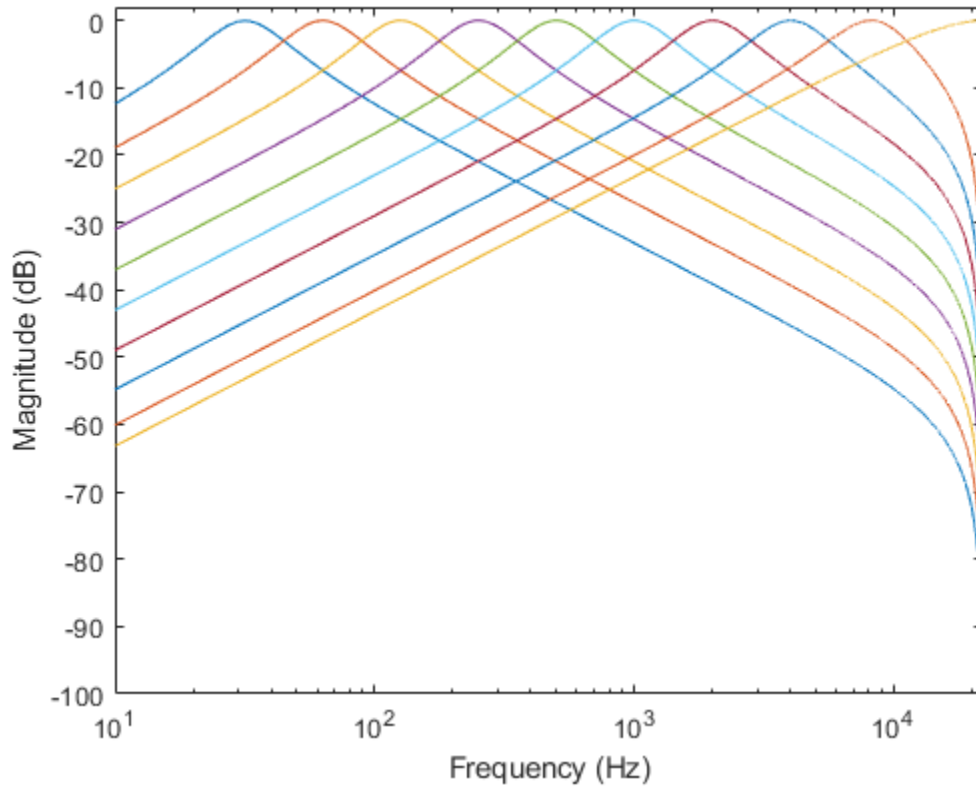
```

```
set(gca, 'XScale', 'log')  
axis([10 octFiltBank.SampleRate/2 -100 2])
```



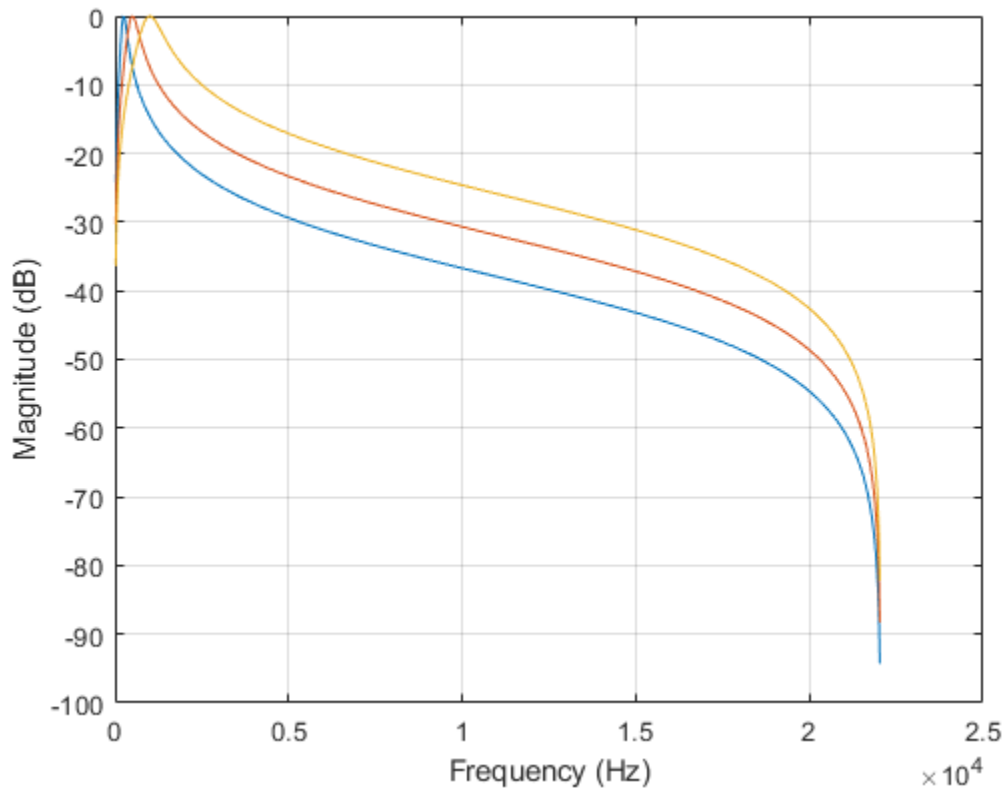
To specify the number of FFT points used to compute the frequency response, use the `NFFT` name-value pair. Specify that the frequency response is calculated using a 8192-point FFT. Plot the magnitude frequency response in dB.

```
[H,f] = freqz(octFiltBank, 'NFFT',8192);  
  
plot(f,20*log10(abs(H)))  
xlabel('Frequency (Hz)')  
ylabel('Magnititude (dB)')  
set(gca, 'XScale', 'log')  
axis([10 octFiltBank.SampleRate/2 -100 2])
```



To visualize the magnitude frequency response only, call `freqz` without any output arguments. Plot the magnitude frequency response, in dB, of filters 4, 5, and 6 using a 1024-point DFT.

```
freqz(octFiltBank, [4,5,6], 'NFFT', 1024)
```



## Input Arguments

**obj** — Object to get filter frequency responses from  
gammatoneFilterBank | octaveFilterBank

Object to get filter frequency responses from, specified as an object of gammatoneFilterBank or octaveFilterBank.

**ind** — Indices of filters to calculate frequency responses from  
1:N (default) | row vector of integers with values in the range [1, N]



Indices of filters to calculate frequency responses from, specified as a row vector of integers with values in the range  $[1, N]$ .  $N$  is the total number of filters designed by `obj`.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'NFFT', 2048`

### **NFFT** — Number of DFT bins

8192 (default) | positive integer

Number of DFT bins, specified as a positive integer.

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

## Output Arguments

### **H** — Complex frequency response of each filter

matrix

Complex frequency response of each filter, returned as an  $M$ -by- $N$  matrix.  $M$  is the number of DFT bins, specified by `NFFT`.  $N$  is the number of filters, which is either `length(ind)` or, if `ind` is not specified, the total number of filters in the filter bank.

Data Types: `double`

### **f** — Frequencies at which response is computed (Hz)

column vector

Frequencies at which the response is computed in Hz, returned as a column vector.

Data Types: `double`

## See Also

`fvtool` | `gammatoneFilterBank` | `octaveFilterBank`

**Introduced in R2019a**

## fvtool

Visualize filter bank

### Syntax

```
fvtool(obj)
fvtool(obj,ind)
fvtool( ____,Name,Value)
```

### Description

`fvtool(obj)` visualizes the filters in the filter bank using the Filter Visualization Tool (FVTool).

`fvtool(obj,ind)` visualizes the filters corresponding to the elements in the vector `ind`.

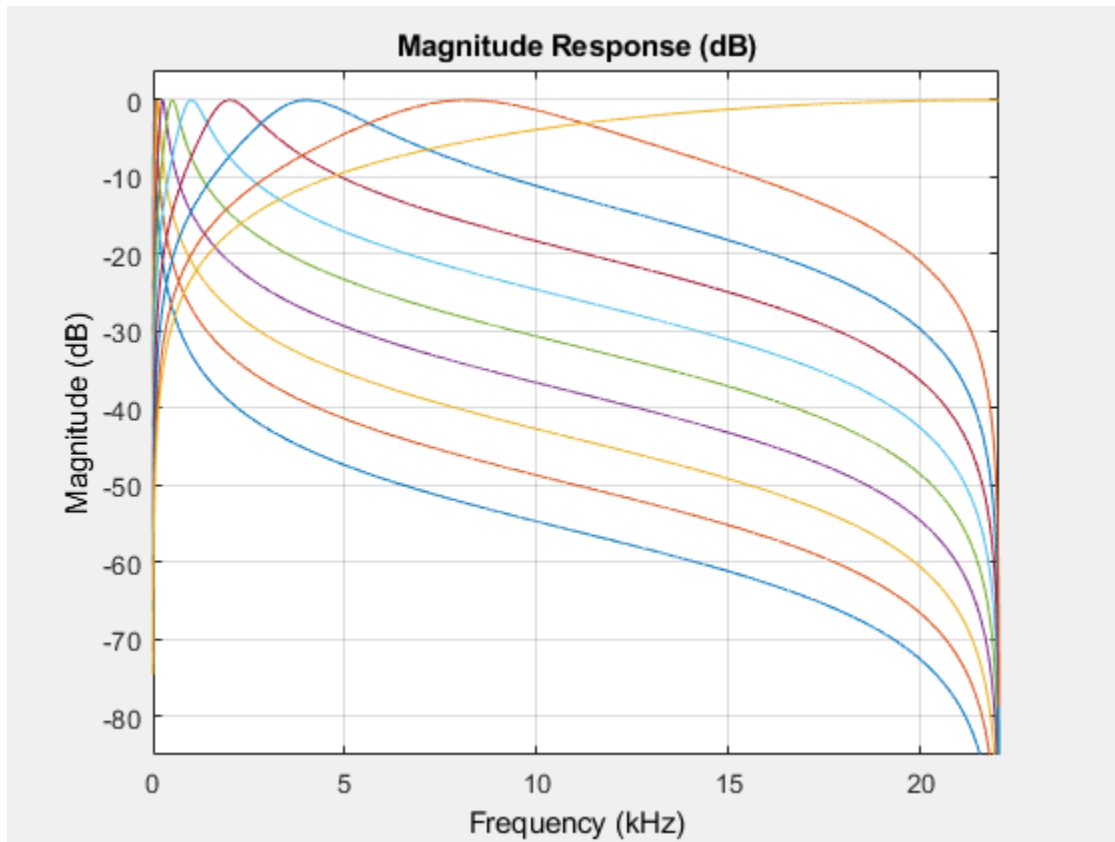
`fvtool( ____,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

### Examples

#### View octaveFilterBank in FVTool

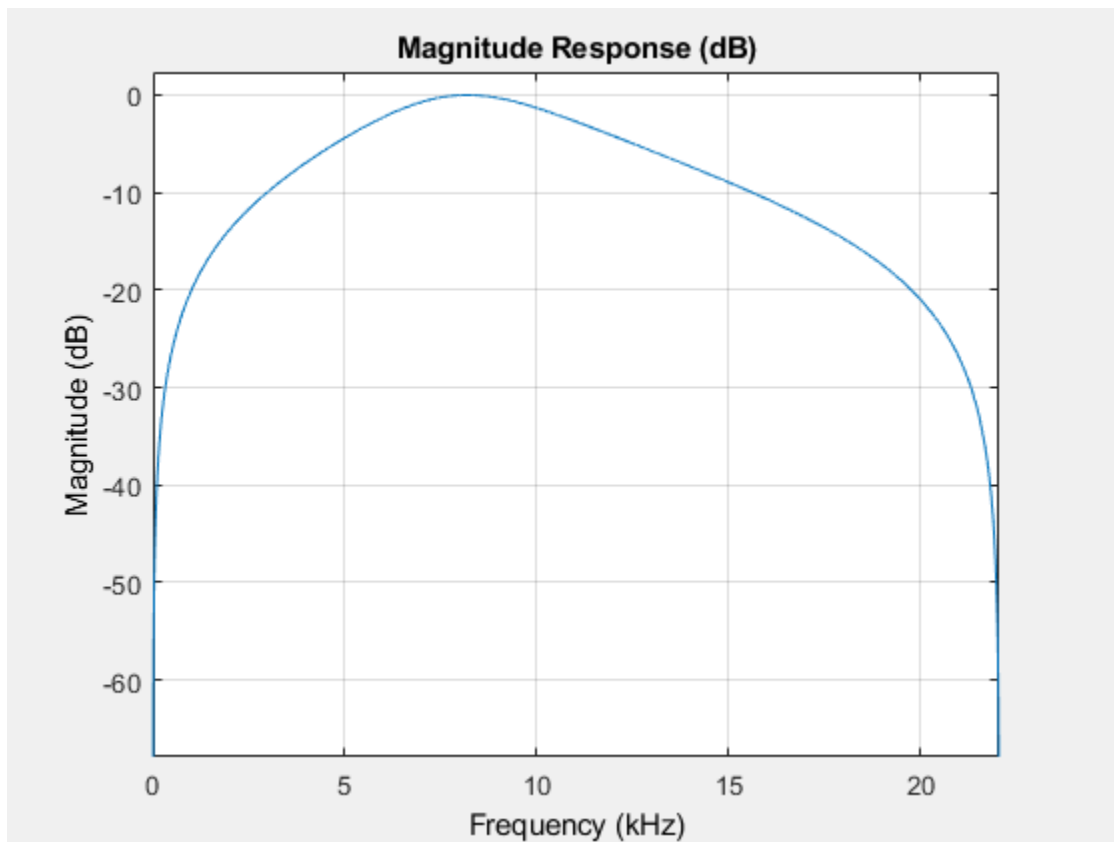
Create an `octaveFilterBank` object. Call `fvtool` to visualize the filter bank.

```
octFiltBank = octaveFilterBank;
fvtool(octFiltBank);
```



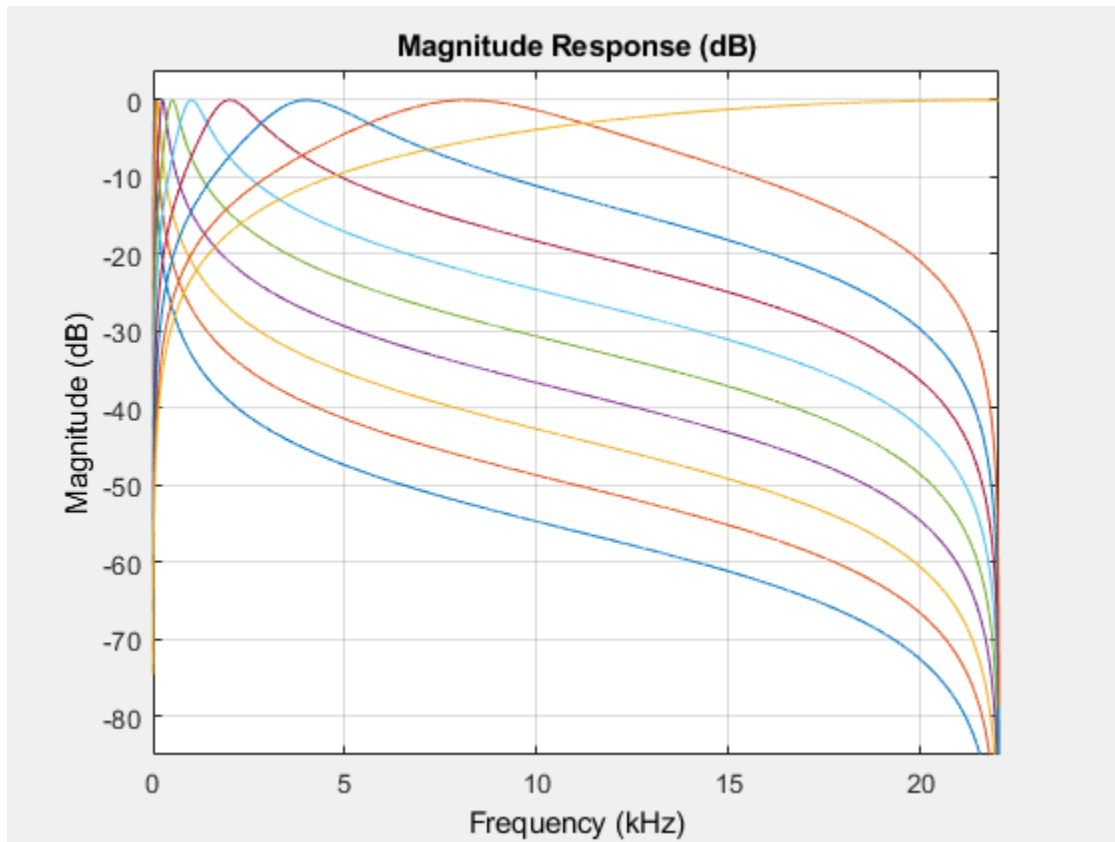
To visualize a subset of filters in the filter bank, specify the second argument as a row vector of indices between one and the number of filters in the filter bank. If not specified, `fvtool` visualizes 1 to  $N$  filters of the filter bank, where  $N$  is the smallest of `octFiltBank.NumFilters` and 64. Visualize the ninth filter.

```
fvtool(octFiltBank,9);
```



To specify the number of FFT points used to compute the frequency response, use the `NFFT` name-value pair. Specify that the frequency response is calculated using a 8192-point FFT.

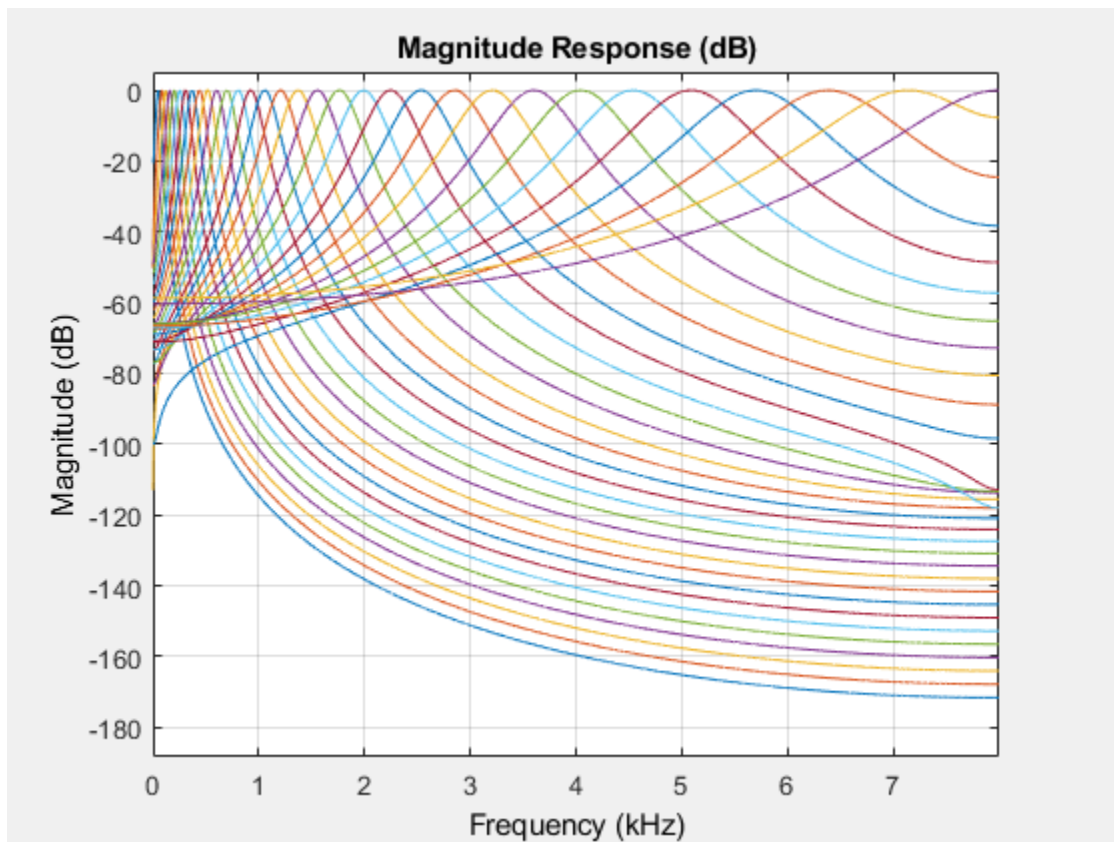
```
fvtool(octFiltBank, 'NFFT', 8192);
```



#### View `gammatoneFilterBank` in FVTool

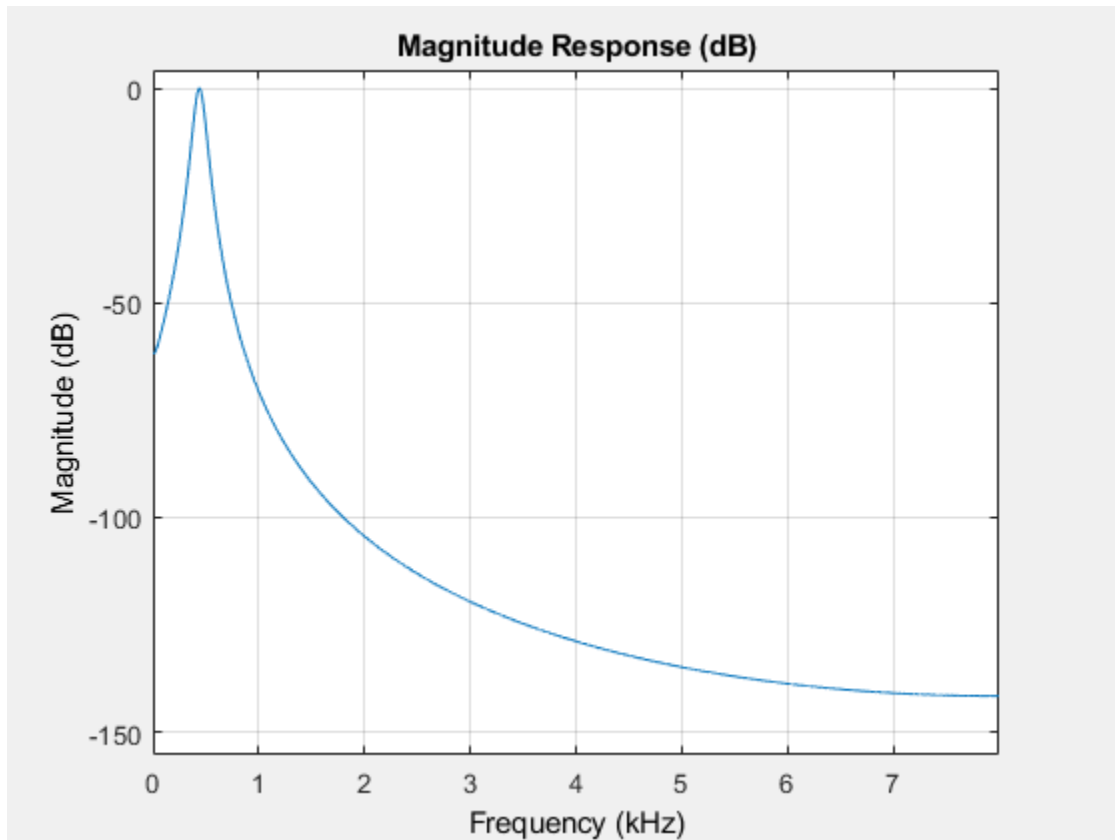
Create a `gammatoneFilterBank` object. Call `fvtool` to visualize the filter bank.

```
gammaFiltBank = gammatoneFilterBank;  
fvtool(gammaFiltBank);
```



To visualize a subset of filters in the filter bank, specify the second argument as a row vector of indices between one and the number of filters in the filter bank. If not specified, `fvtool` visualizes 1 to  $N$  filters of the filter bank, where  $N$  is the smallest of `gammaFiltBank.NumFilters` and 64. Visualize the ninth filter.

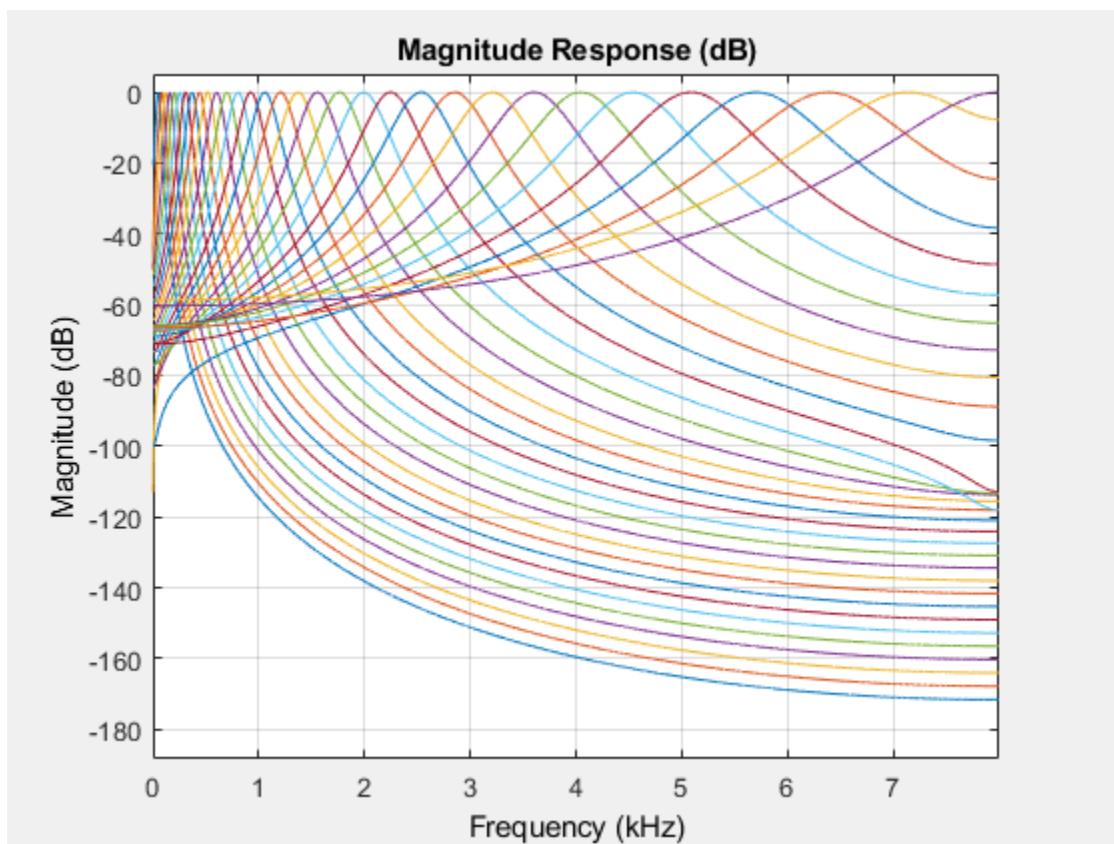
```
fvtool(gammaFiltBank,9);
```



To specify the number of FFT points used to compute the frequency response, use the `NFFT` name-value pair. Specify that the frequency response is calculated using a 8192-point FFT.

```
fvtool(gammaFiltBank, 'NFFT', 8192);
```





## Input Arguments

**obj** — Object to get filter frequency responses from  
gammatoneFilterBank | octaveFilterBank

Object to get filter frequency responses from, specified as an object of gammatoneFilterBank or octaveFilterBank.

**ind** — Indices of filters to calculate frequency responses from  
1:max(N, 64) (default) | row vector of integers with values in the range [1, N]

Indices of filters to calculate frequency responses from, specified as a row vector of integers with values in the range  $[1, N]$ .  $N$  is the total number of filters designed by `obj`.

### 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'NFFT', 2048`

#### **NFFT — Number of DFT bins**

8192 (default) | positive integer

Number of DFT bins, specified as a positive integer.

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

### See Also

`gammatoneFilterBank` | `octaveFilterBank`

**Introduced in R2019a**

# getBandedgeFrequencies

Get filter bandedges

## Syntax

```
bandEdges = getBandedgeFrequencies(obj)
[bandEdges,centerFrequencies] = getBandedgeFrequencies(obj)
```

## Description

`bandEdges = getBandedgeFrequencies(obj)` returns the bandedge frequencies of the filters designed by `obj`. If there are  $M$  filters, then there are  $M$  center frequencies and  $M+1$  band edge frequencies.

`[bandEdges,centerFrequencies] = getBandedgeFrequencies(obj)` returns the center frequencies of the filters designed by `obj`.

## Examples

### Get Bandedge Frequencies

Create a default `octaveFilterBank` object.

```
octFiltBank = octaveFilterBank;
```

Call `getBandedgeFrequencies` to return a vector of bandedge frequencies.

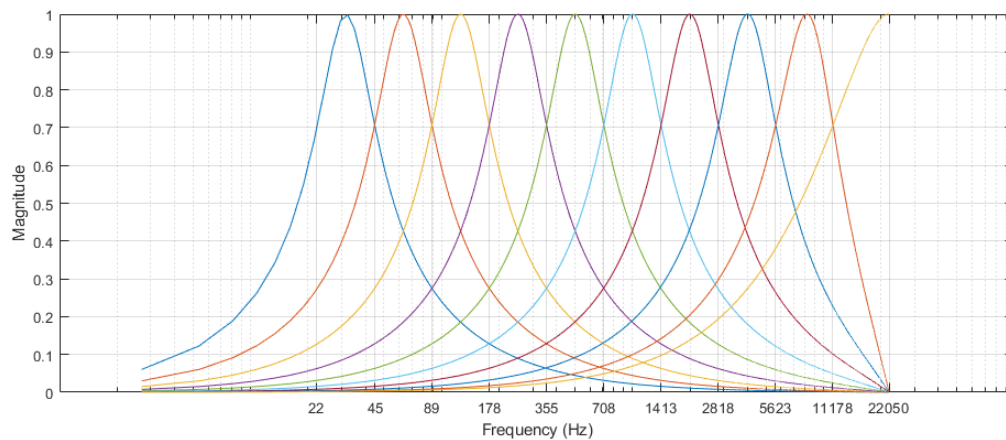
```
bE = getBandedgeFrequencies(octFiltBank)
```

```
bE = 1×11
104 ×
```

```
    0.0022    0.0045    0.0089    0.0178    0.0355    0.0708    0.1413    0.2818    0.5
```

Call `freqz` to get the frequency response of the filter bank. Plot the magnitude frequency response. Use the bandedge frequencies to label the frequency axis.

```
[H,f] = freqz(octFiltBank);  
semilogx(f,abs(H))  
xticks(round(bE))  
xlabel('Frequency (Hz)')  
ylabel('Magnitude')  
grid on  
h = gcf;  
set(h,'Position',[h.Position(1) h.Position(2) h.Position(3)*2 h.Position(4)])
```



## Input Arguments

**obj** — Object to get filter information from

octaveFilterBank object

Object to get filter information from, specified as an object of `octaveFilterBank`.

## Output Arguments

**bandEdges** — Bandedges of filters (Hz)

row vector

Bandedges of filters designed by `obj` in Hz, returned as a row vector.

Data Types: `double` | `single`

### **centerFrequencies** — Center frequencies of filters (Hz)

row vector

Center frequencies of filters designed by `obj` in Hz, returned as a row vector.

Data Types: `double` | `single`

## **See Also**

`octaveFilterBank`

**Introduced in R2019a**

## getCenterFrequencies

Center frequencies of filters

### Syntax

```
cf = getCenterFrequencies(obj)
```

### Description

`cf = getCenterFrequencies(obj)` returns the center frequencies of the filters created by `obj`, in Hz.

### Examples

#### Center Frequencies of `gammatoneFilterBank`

Create a `gammatoneFilterBank` and get the center frequencies of the filters in the filter bank.

```
gammaFiltBank = gammatoneFilterBank;
```

```
cf = getCenterFrequencies(gammaFiltBank)
```

```
cf = 1×32  
103 ×
```

```
    0.0500    0.0822    0.1180    0.1581    0.2027    0.2525    0.3080    0.3700    0.4
```

Center frequencies of a gammatone filter bank are spaced evenly on the ERB scale. Convert the center frequencies vector to the ERB scale and calculate the differences between center frequencies.

```
diff(hz2erb(cf))
```

```
ans = 1×31
    1.0116    1.0116    1.0116    1.0116    1.0116    1.0116    1.0116    1.0116    1.0116    1.0116
```

### Center Frequencies of octaveFilterBank

Create an octaveFilterBank and get the center frequencies of the filters in the filter bank.

```
octFiltBank = octaveFilterBank;
cf = getCenterFrequencies(octFiltBank)

cf = 1×10
    104 ×
    0.0032    0.0063    0.0126    0.0251    0.0501    0.1000    0.1995    0.3981    0.7959
```

Center frequencies of an octave filter bank are spaced evenly on a logarithmic scale. Convert the center frequencies vector to the log scale and calculate the differences between center frequencies.

```
diff(log10(cf))

ans = 1×9
    0.3000    0.3000    0.3000    0.3000    0.3000    0.3000    0.3000    0.3000    0.3000
```

### Get Center Frequencies of Octave Filter Bank Used in splMeter

Create an octave bandwidth splMeter and get the center frequencies of the octave filter bank. Round the center frequencies to two significant digits for display purposes.

```
SPL = splMeter('SampleRate',44100,'Bandwidth','1 octave');
cf = getCenterFrequencies(SPL);
round(cf,2,'significant')
```

```
ans = 1x10  
      32      63     130     250     500     1000     2000
```

### Input Arguments

**obj** — Object to get filter bank center frequencies from  
gammatoneFilterBank | octaveFilterBank | splMeter

Object to get filter bank center frequencies from, specified as an object of gammatoneFilterBank, octaveFilterBank, or splMeter.

### Output Arguments

**cf** — Filter bank center frequencies (Hz)  
scalar | vector

Filter bank center frequencies in Hz, returned a scalar or vector.

### See Also

gammatoneFilterBank | octaveFilterBank | splMeter

**Introduced in R2019a**



# getBandwidths

Get filter bandwidths

## Syntax

```
bw = getBandwidths(obj)
```

## Description

`bw = getBandwidths(obj)` returns the bandwidths of the filters created by `obj`, in Hz.

## Examples

### Get Filter Bandwidths of `gammatoneFilterBank`

Create a default `gammatoneFilterBank`. Call `getBandwidths` to get the bandwidths of the filters, in Hz.

```
gammaFiltBank = gammatoneFilterBank;
```

```
bw = getBandwidths(gammaFiltBank)
```

```
bw = 1×32
```

```
    30.6688    34.2071    38.1536    42.5554    47.4650    52.9410    59.0489    65.8614    73.4
```

## Input Arguments

**obj** — Object to get filter bandwidth from  
`gammatoneFilterBank`

Object to get filter bandwidth from, specified as an object of `gammatoneFilterBank`.

## Output Arguments

### **bw** — Filter bandwidths (Hz)

scalar | vector

Filter bandwidths in Hz, returned a scalar or vector.

## See Also

`gammatoneFilterBank`

**Introduced in R2019a**

# getGroupDelays

Get group delays

## Syntax

```
groupDelays = getGroupDelays(obj)  
[groupDelays,centerFrequencies] = getGroupDelays(obj)
```

## Description

`groupDelays = getGroupDelays(obj)` returns the group delay of each filter at its center frequency.

`[groupDelays,centerFrequencies] = getGroupDelays(obj)` returns the center frequency of each filter.

## Examples

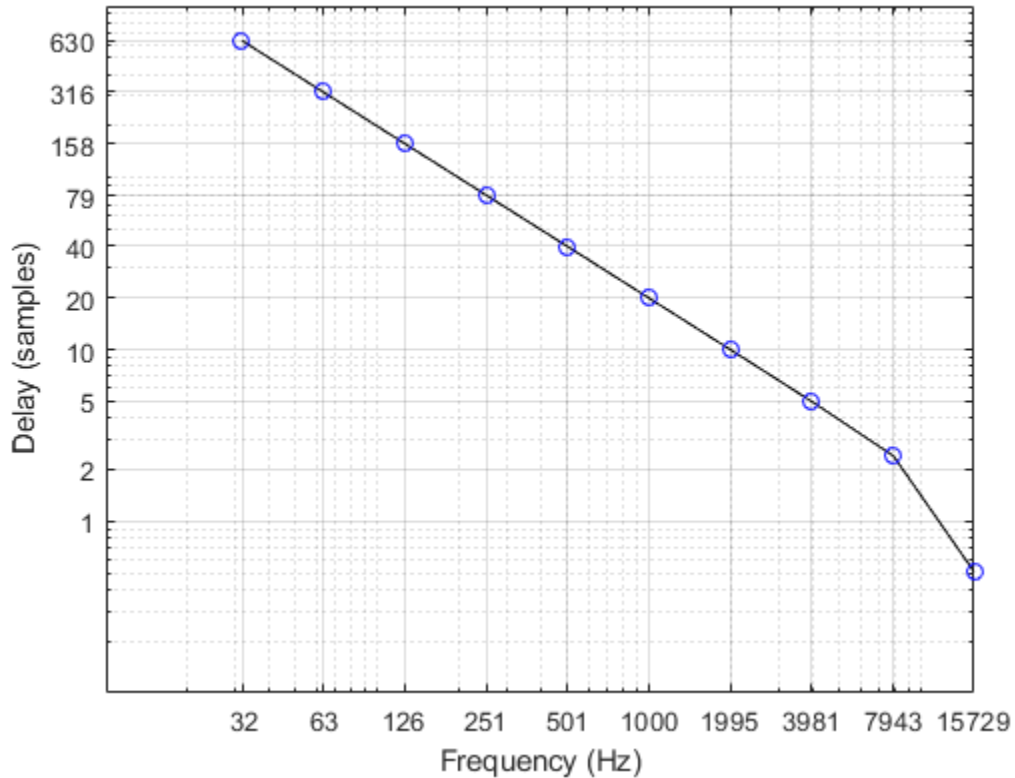
### Get Group Delays

Create a default `octaveFilterBank` object. Call `getGroupDelays` to get the group delay of each octave filter at its center frequency.

```
octFiltBank = octaveFilterBank;  
[gd,cf] = getGroupDelays(octFiltBank);
```

Plot the group delay as a function of filter center frequency.

```
loglog(cf,gd, 'k-',cf,gd, 'bo')  
grid on  
xlabel('Frequency (Hz)')  
ylabel('Delay (samples)')  
xticks(round(cf))  
yticks(round(fliplr(gd)))
```



## Input Arguments

**obj** — Object to get group delays from  
`octaveFilterBank`

Object to get group delays from, specified as an object of `octaveFilterBank`.

## Output Arguments

### **groupDelays** — Group delays (samples)

row vector

Group delay of each filter at its center frequency in samples, returned as a row vector.

### **centerFrequencies** — Center frequencies of filters (Hz)

row vector

Center frequencies of filters designed by `obj` in Hz, returned as a row vector.

Data Types: `double` | `single`

## See Also

`octaveFilterBank`

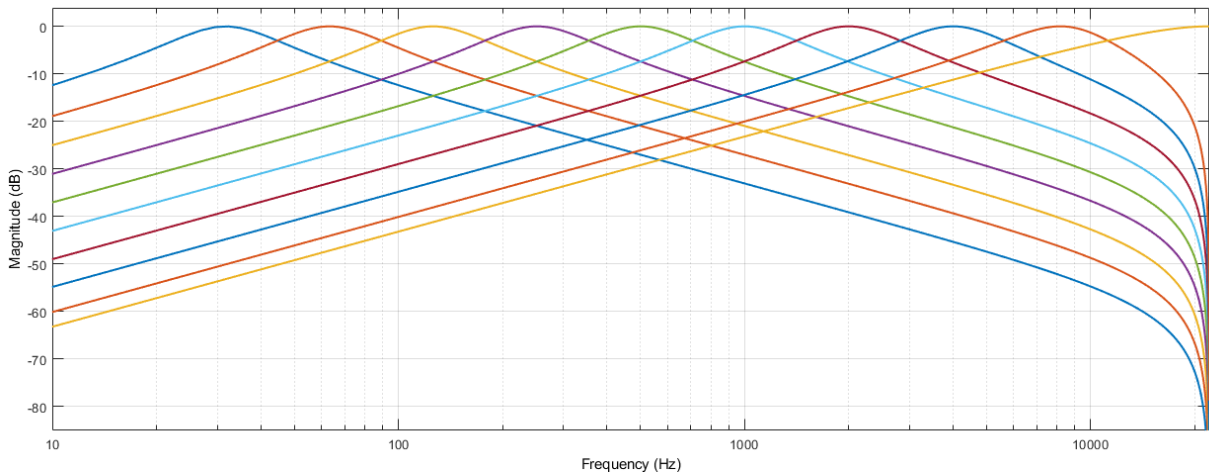
**Introduced in R2019a**

## octaveFilterBank

Octave and fractional-octave filter bank

### Description

`octaveFilterBank` decomposes a signal into octave or fractional-octave subbands. An octave-band is a frequency band where the highest frequency is twice the lowest frequency. Octave-band and fractional octave-band filters are commonly used to mimic how humans perceive loudness.



To apply a bank of octave-band or fractional octave-band filters:

- 1 Create the `octaveFilterBank` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

## Syntax

```
octFiltBank = octaveFilterBank
octFiltBank = octaveFilterBank(bandwidth)
octFiltBank = octaveFilterBank(bandwidth, fs)
octFiltBank = octaveFilterBank( ____, Name, Value)
```

## Description

`octFiltBank = octaveFilterBank` returns an octave filter bank. The object filters data independently across each input channel over time.

`octFiltBank = octaveFilterBank(bandwidth)` sets the `Bandwidth` property to `bandwidth`.

`octFiltBank = octaveFilterBank(bandwidth, fs)` sets the `SampleRate` property to `fs`.

`octFiltBank = octaveFilterBank( ____, Name, Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `octFiltBank = octaveFilterBank('1/2 octave', 'FrequencyRange', [62.5, 12000])` creates a  $\frac{1}{2}$  octave-band filter bank, `octFiltBank`, with bandpass filters placed between 62.5 Hz and 12,000 Hz.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects \(MATLAB\)](#).

**Bandwidth — Filter bandwidth (octave)**

'1 octave' (default) | '2/3 octave' | '1/2 octave' | '1/3 octave' | '1/6 octave' | '1/12 octave' | '1/24 octave' | '1/48 octave'

Filter bandwidth in octaves, specified as '1 octave', '2/3 octave', '1/2 octave', '1/3 octave', '1/6 octave', '1/12 octave', '1/24 octave', or '1/48 octave'.

**Tunable:** No

Data Types: char | string

**SampleRate — Input sample rate (Hz)**

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: single | double

**FrequencyRange — Frequency range of filter bank (Hz)**

[22 22050] (default) | two-element row vector of positive monotonically increasing values

Frequency range of the filter bank in Hz, specified as a two-element row vector of positive monotonically increasing values. The filter bank center frequencies are placed according to the Bandwidth, ReferenceFrequency, and OctaveRatioBase properties. Filters that have a center frequency outside FrequencyRange are ignored.

**Tunable:** No

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

**ReferenceFrequency — Reference frequency (Hz)**

1000 (default) | positive integer scalar

Reference frequency of the filter bank in Hz, specified as a positive integer scalar. The reference frequency defines one of the center frequencies. All other center frequencies are set relative to the reference frequency.

**Tunable:** No



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

### **FilterOrder — Order of octave filters**

2 (default) | even integer

Order of the octave filters, specified as an even integer. The filter order applies to each individual filter in the filter bank.

**Tunable:** No

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

### **OctaveRatioBase — Octave ratio base**

10 (default) | 2

Octave ratio base, specified as 10 or 2. The octave ratio base determines the distribution of the center frequencies of the octave filters. The ANSI S1.11 standard recommends base 10. Base 2 is popular for music applications. Base 2 defines an octave as a factor of 2, and base 10 defines an octave as a factor of  $10^{0.3}$ .

**Tunable:** No

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

## **Usage**

## **Syntax**

```
audioOut = octFiltBank(audioIn)
```

## **Description**

`audioOut = octFiltBank(audioIn)` applies the octave filter bank on the input and returns the filtered output.

### Input Arguments

#### **audioIn** — Audio input to octave filter bank

scalar | vector | matrix

Audio input to the octave filter bank, specified as a scalar, vector, or matrix. If specified as a matrix, the columns are treated as independent audio channels.

Data Types: `single` | `double`

### Output Arguments

#### **audioOut** — Audio output from octave filter bank

matrix | 3-D array

Audio output from octave filter bank, returned as a scalar, vector, matrix, or 3-D array. The shape of `audioOut` depends on the shape of `audioIn` and the number of filters in the filter bank. If  $M$  is the number of filters, and `audioIn` is an  $L$ -by- $N$  matrix, then `audioOut` is returned as an  $L$ -by- $M$ -by- $N$  array. If  $N$  is 1, then `audioOut` is a matrix.

Data Types: `single` | `double`

### Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

#### **Specific to octaveFilterBank**

<code>coeffs</code>	Get filter coefficients
<code>freqz</code>	Compute frequency response
<code>fvtool</code>	Visualize filter bank
<code>getBandedgeFrequencies</code>	Get filter bandedges
<code>getCenterFrequencies</code>	Center frequencies of filters
<code>getGroupDelays</code>	Get group delays
<code>info</code>	Get filter information

## Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

## Examples

### Apply Octave Filter Bank

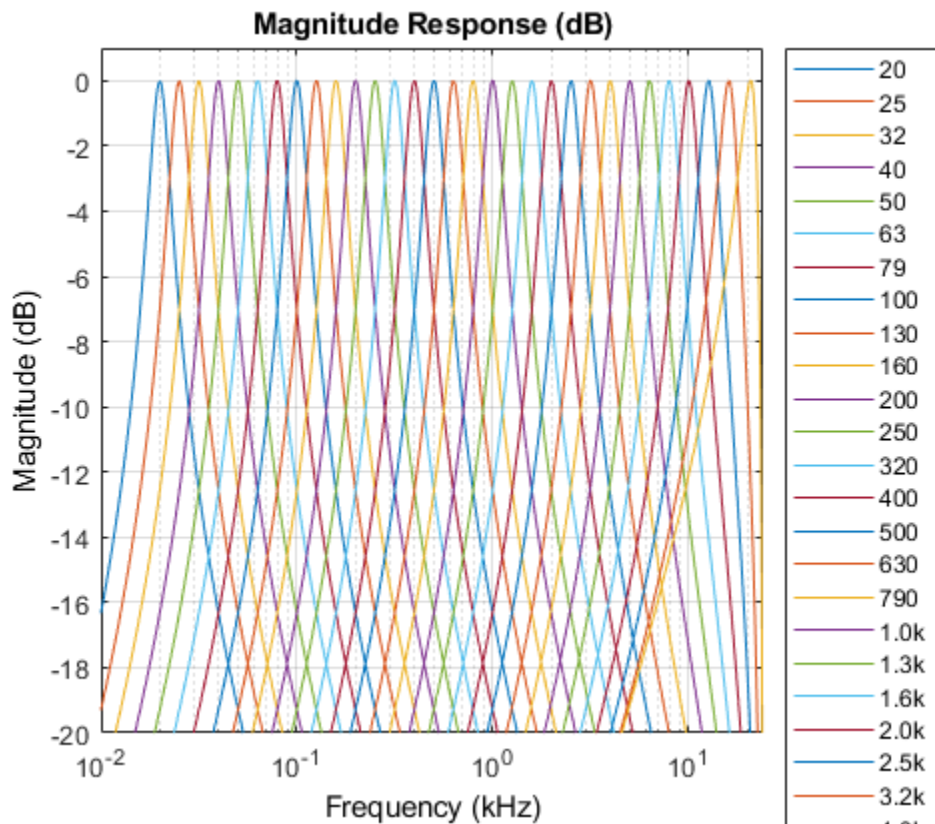
Create a 1/3-octave filter bank for a signal sampled at 48 kHz. Set the frequency range to [18 22000] Hz.

```
octFilBank = octaveFilterBank('1/3 octave',48000, ...
                              'FrequencyRange',[18 22000]);
```

Use `fvtool` to visualize the response of the filter bank. To get a high-resolution view on the lower frequencies, set the scale of the x-axis to `log` and `NFFT` to  $2^{16}$ . Add a legend indicating the filter bank center frequencies.

```
fvtool(octFilBank, 'NFFT', 2^16);
set(gca, 'XScale', 'log')
axis([.01 24 -20 1])

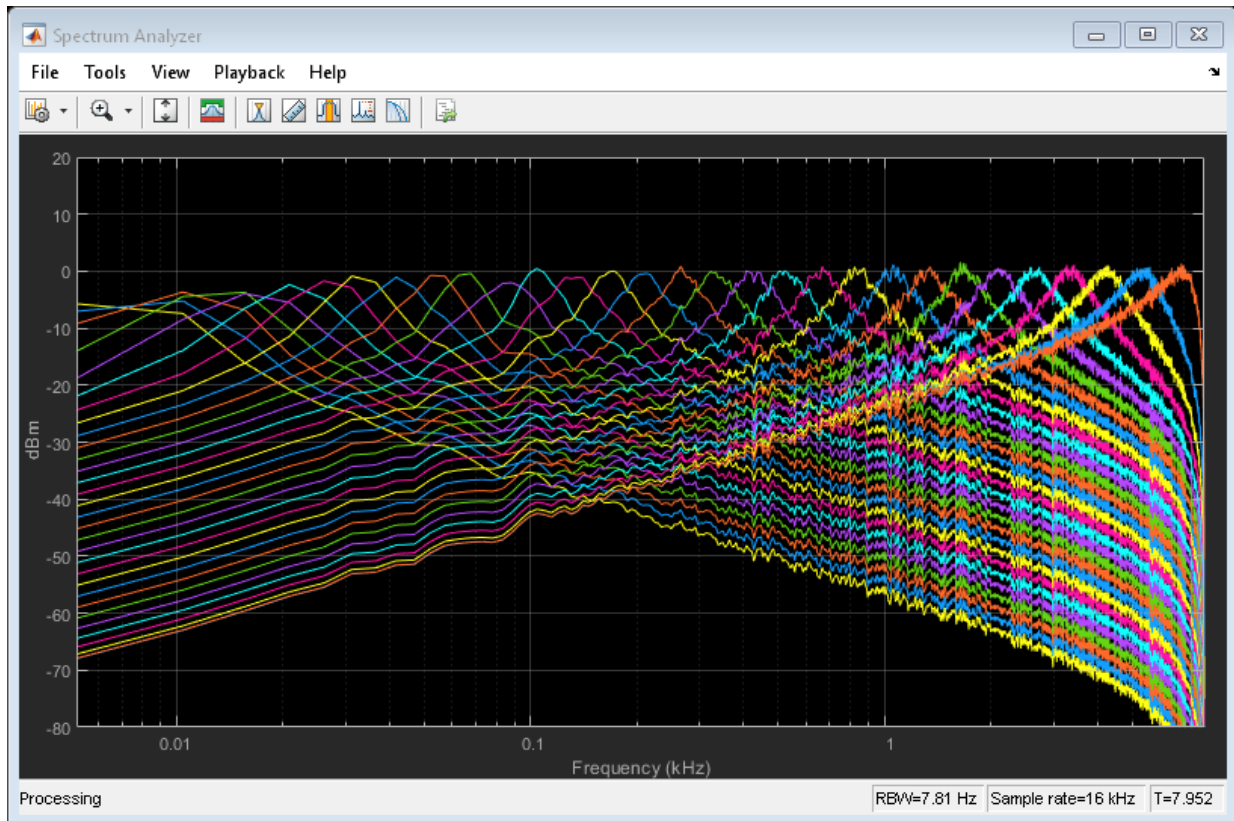
fc = getCenterFrequencies(octFilBank);
fcc = cell(size(fc));
for ii = find(fc < 1000)
    fcc{ii} = sprintf('%.0f', round(fc(ii), 2, 'significant'));
end
for ii = find(fc >= 1000)
    fcc{ii} = sprintf('%.1fk', fc(ii)/1000);
end
legend(fcc, 'Location', 'eastoutside')
```



Process white Gaussian noise through the filter bank. Use a spectrum analyzer to view the spectrum of the filter outputs.

```
sa = dsp.SpectrumAnalyzer('SampleRate',16e3,...
    'PlotAsTwoSidedSpectrum',false,...
    'FrequencyScale','log',...
    'SpectralAverages',100);

for index = 1:500
    x = randn(256,1);
    y = octFilBank(x);
    sa(y);
end
```



## Analysis and Synthesis

The `octaveFilterBank` enables good reconstruction of a signal after analyzing or modifying its subbands.

Read in an audio file and listen to its contents.

```
[audioIn,fs] = audioread('RandomOscThree-24-96-stereo-13secs.aif');
sound(audioIn,fs)
```

Create a default `octaveFilterBank`. The default frequency range of the filter bank is 22 to 22,050 Hz. Frequencies outside of this range are attenuated in the reconstructed signal.

```
octFiltBank = octaveFilterBank('SampleRate', fs);
```

Pass the audio signal through the octave filter bank. The number of outputs depends on the `FrequencyRange`, `ReferenceFrequency`, `OctaveRatioBase`, and `Bandwidth` properties of the octave filter bank. Each channel of the input is passed through a filter bank independently and is returned as a separate page in the output.

```
audioOut = octFiltBank(audioIn);
```

```
[N,numFilters,numChannels] = size(audioOut)
```

```
N = 1265935
```

```
numFilters = 10
```

```
numChannels = 2
```

The octave filter bank introduces various group delays. To compensate for the group delay, remove the beginning delay from the individual filter outputs and zero-pad the ends of the signals so that they are all the same size. Use `getGroupDelays` to get the group delays. Listen to the group delay-compensated reconstruction.

```
groupDelay = round(getGroupDelays(octFiltBank)); % round for simplicity
```

```
audioPadded = [audioOut;zeros(max(groupDelay),numFilters,numChannels)];
```

```
for i = 1:numFilters
```

```
    audioOut(:,i,:) = audioPadded(groupDelay(i)+1:N+groupDelay(i),i,:);
```

```
end
```

To reconstruct the original signal, sum the outputs of the filter banks for each channel. Use `squeeze` to move the second channel from the third dimension to the second in the reconstructed signal.

```
reconstructedSignal = squeeze(sum(audioOut,2));
```

```
sound(reconstructedSignal, fs)
```

## Algorithms

The `octaveFilterBank` is implemented as a parallel structure of octave filters. Individual octave filters are designed as described by `octaveFilter`. By default, the octave filter bank center frequencies are placed as specified by the ANSI S1.11-2004

standard. You can modify the filter placements using the `Bandwidth`, `FrequencyRange`, `ReferenceFrequency`, and `OctaveRatioBase` properties.

## References

- [1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 2010.
- [2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters*. ANSI S1.11-2004. Melville, NY: Acoustical Society of America, 2009.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

`gammatoneFilterBank` | `graphicEQ` | `octaveFilter` | `splMeter`

### Topics

“Octave-Band and Fractional Octave-Band Filters”

**Introduced in R2019a**

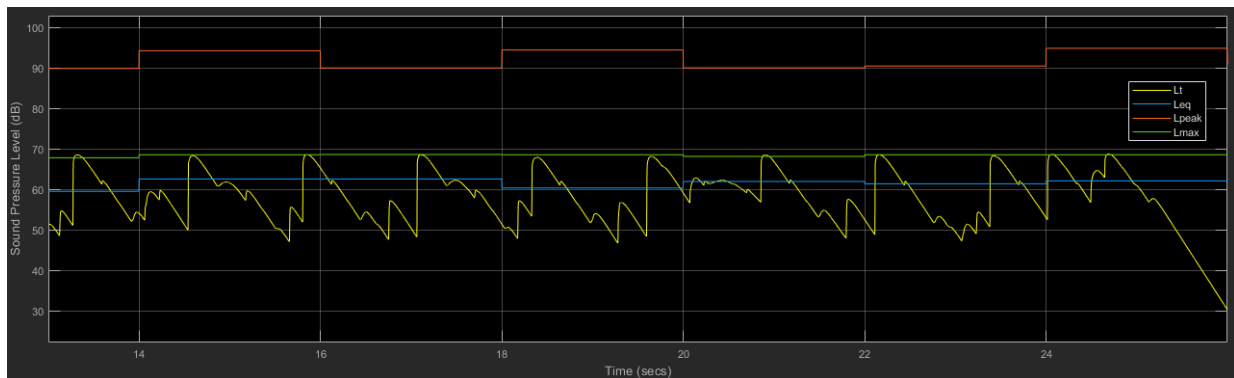
## splMeter

Measure sound pressure level of audio signal

### Description

The `splMeter` System object computes sound pressure level measurements. The object returns measurements for:

- frequency-weighted sound levels
- fast or slow time-weighted sound levels
- equivalent-continuous sound levels
- peak sound levels
- maximum sound levels



To implement SPL metering:

- 1 Create the `splMeter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).



## Creation

### Syntax

```
SPL = splMeter
SPL = splMeter(Name,Value)
```

### Description

`SPL = splMeter` creates a System object, `SPL`, that performs SPL metering.

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

Example: `SPL = splMeter('FrequencyWeighting','C-weighting','SampleRate',12000)` creates a System object, `SPL`, that performs C-weighting and operates at 12 kHz.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects (MATLAB)*.

#### **Bandwidth — Width of analysis bands**

'Full band' (default) | '1 octave' | '2/3 octave' | '1/3 octave'

Width of analysis bands, specified as 'Full band', '1 octave', '2/3 octave', or '1/3 octave'. If `Bandwidth` is specified as 'Full band', the SPL meter returns one set of measurements for the whole frequency band. If `Bandwidth` is specified as '1 octave', '2/3 octave', or '1/3 octave', the SPL meter returns one set of measurements per octave or fractional-octave band.

**Tunable:** No

Data Types: char | string

#### **OctaveFilterOrder — Order of octave filter**

2 (default) | even integer

Order of the octave filter, specified as an even integer.

**Tunable:** No

#### **Dependencies**

To enable this property, set Bandwidth to '1 octave', '2/3 octave', or '1/3 octave'.

Data Types: single | double

#### **FrequencyWeighting — Frequency weighting applied to input**

'A-weighting' (default) | 'C-weighting' | 'Z-weighting' (no weighting)

Frequency weighting applied to input, specified as 'A-weighting', 'C-weighting', or 'Z-weighting', where Z-weighting corresponds to no weighting. The frequency weighting is designed and implemented using the `weightingFilter` System object.

**Tunable:** No

Data Types: char | string

#### **TimeWeighting — Time weighting (s)**

'Fast' (default) | 'Slow'

Time weighting, in seconds, for calculation of time-weighted sound level and maximum time-weighted sound level, specified as 'Fast' or 'Slow'. The `TimeWeighting` property is used to specify the coefficient of a lowpass filter.

- 'Fast' - 1/8
- 'Slow' - 1

**Tunable:** Yes

Data Types: char | string

#### **PressureReference — Reference pressure for dB calculations (Pa)**

2e-5 (default) | positive scalar

Reference pressure for dB calculations in Pa, specified as a positive scalar.

**Tunable:** Yes

Data Types: single | double

**TimeInterval** — Time interval for reporting level measurements (s)

1 (default) | positive scalar

Time interval, in seconds, to report equivalent-continuous, peak, and maximum time-weighted sound levels, specified as a positive scalar integer.

**Tunable:** No

Data Types: single | double

**CalibrationFactor** — Scalar calibration factor multiplied by input

1 (default) | positive finite scalar

Scalar calibration factor multiplied by input.

To set the calibration factor using a reference tone, use `calibrate`.

**Tunable:** No

Data Types: single | double

**SampleRate** — Input sample rate (Hz)

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** No

Data Types: single | double

## Usage

## Syntax

[Lt,Leq,Lpeak,Lmax] = SPL(audioIn)

### Description

`[Lt,Leq,Lpeak,Lmax] = SPL(audioIn)` returns measurement values for the time-weighted (`Lt`) sound level of the current input frame, `audioIn`. The object also returns the equivalent-continuous (`Leq`), peak (`Lpeak`), and maximum time-weighted (`Lmax`) sound levels of the input to your SPL meter.

### Input Arguments

#### **audioIn** — Audio input to SPL meter

column vector | matrix

Audio input to the SPL meter, specified as a column vector or matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

### Output Arguments

#### **Lt** — Time-weighted sound level (dB)

column vector | matrix | 3-D array

Time-weighted sound level in dB, returned as a column vector, matrix, or 3-D array the same type as `audioIn`.

Size and interpretation of the outputs depend on what the `Bandwidth` property is set to:

- `'Full band'` (default) -- `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as column vectors or matrices the same size as `audioIn`.
- `'1 octave'`, `'2/3 octave'`, or `'1/3 octave'` -- `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as *L-by-B-by-C* arrays.
  - *L* -- Number of rows in `audioIn`
  - *B* -- Number of octave bands
  - *C* -- Number of columns in `audioIn`

Data Types: `single` | `double`

#### **Leq** — Equivalent-continuous sound level (dB)

column vector | matrix | 3-D array

Equivalent-continuous sound level in dB, returned as a column vector, matrix, or 3-D array the same type as `audioIn`.

Size and interpretation of the outputs depend on what the `Bandwidth` property is set to:

- 'Full band' (default) -- `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as column vectors or matrices the same size as `audioIn`.
- '1 octave', '2/3 octave', or '1/3 octave' -- `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as *L-by-B-by-C* arrays.
  - *L* -- Number of rows in `audioIn`
  - *B* -- Number of octave bands
  - *C* -- Number of columns in `audioIn`

Data Types: `single` | `double`

#### **Lpeak — Peak sound level (dB)**

column vector | matrix | 3-D array

Peak sound level in dB, returned as a column vector, matrix, or 3-D array the same type as `audioIn`.

Size and interpretation of the outputs depend on what the `Bandwidth` property is set to:

- 'Full band' (default) -- `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as column vectors or matrices the same size as `audioIn`.
- '1 octave', '2/3 octave', or '1/3 octave' -- `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as *L-by-B-by-C* arrays.
  - *L* -- Number of rows in `audioIn`
  - *B* -- Number of octave bands
  - *C* -- Number of columns in `audioIn`

Data Types: `single` | `double`

#### **Lmax — Maximum time-weighted sound level (dB)**

column vector | matrix | 3-D array

Maximum time-weighted sound level in dB, returned as a column vector, matrix, or 3-D array the same type as `audioIn`.

Size and interpretation of the outputs depend on what the Bandwidth property is set to:

- 'Full band' (default) --  $L_t$ ,  $L_{eq}$ ,  $L_{peak}$ , and  $L_{max}$  are returned as column vectors or matrices the same size as `audioIn`.
- '1 octave', '2/3 octave', or '1/3 octave' --  $L_t$ ,  $L_{eq}$ ,  $L_{peak}$ , and  $L_{max}$  are returned as *L-by-B-by-C* arrays.
  - $L$  -- Number of rows in `audioIn`
  - $B$  -- Number of octave bands
  - $C$  -- Number of columns in `audioIn`

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `splMeter`

`calibrate`                      Calibrate meter using calibration tone with known level  
`getCenterFrequencies`      Center frequencies of filters

### Common to All System Objects

`step`                      Run System object algorithm  
`release`                    Release resources and allow changes to System object property values and input characteristics  
`reset`                      Reset internal states of System object

## Examples

## Measure SPL of Audio Signal

Use the `splMeter` System object™ to measure the A-weighted sound pressure level of a streaming audio signal. Specify a two second time-interval for reporting and a fast time-weighting. Visualize the SPL measurements using the `dsp.TimeScope` System object.

Create a `dsp.AudioFileReader` object to read in an audio file frame by frame. Create an `audioDeviceWriter` object to listen to the audio signal. Create a `dsp.TimeScope` object to visualize SPL measurements. Create an `splMeter` to measure the sound pressure level of the audio file. Use the default calibration factor of 1.

```
source = dsp.AudioFileReader('Ambiance-16-44p1-mono-12secs.wav');
fs = source.SampleRate;

player = audioDeviceWriter('SampleRate',fs);

scope = dsp.TimeScope('SampleRate',fs, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',3,'ShowGrid',true, ...
    'YLimits',[20 110],'AxesScaling','Auto', ...
    'ShowLegend',true,'BufferLength',4*3*fs, ...
    'ChannelNames', ...
    {'Lt_AF','Leq_A','Lpeak_A','Lmax_AF'}, ...
    'Name','Sound Pressure Level Meter');

SPL = splMeter('TimeWeighting','Fast', ...
    'FrequencyWeighting','A-weighting', ...
    'SampleRate',fs, ...
    'TimeInterval',2);
```

In an audio stream loop:

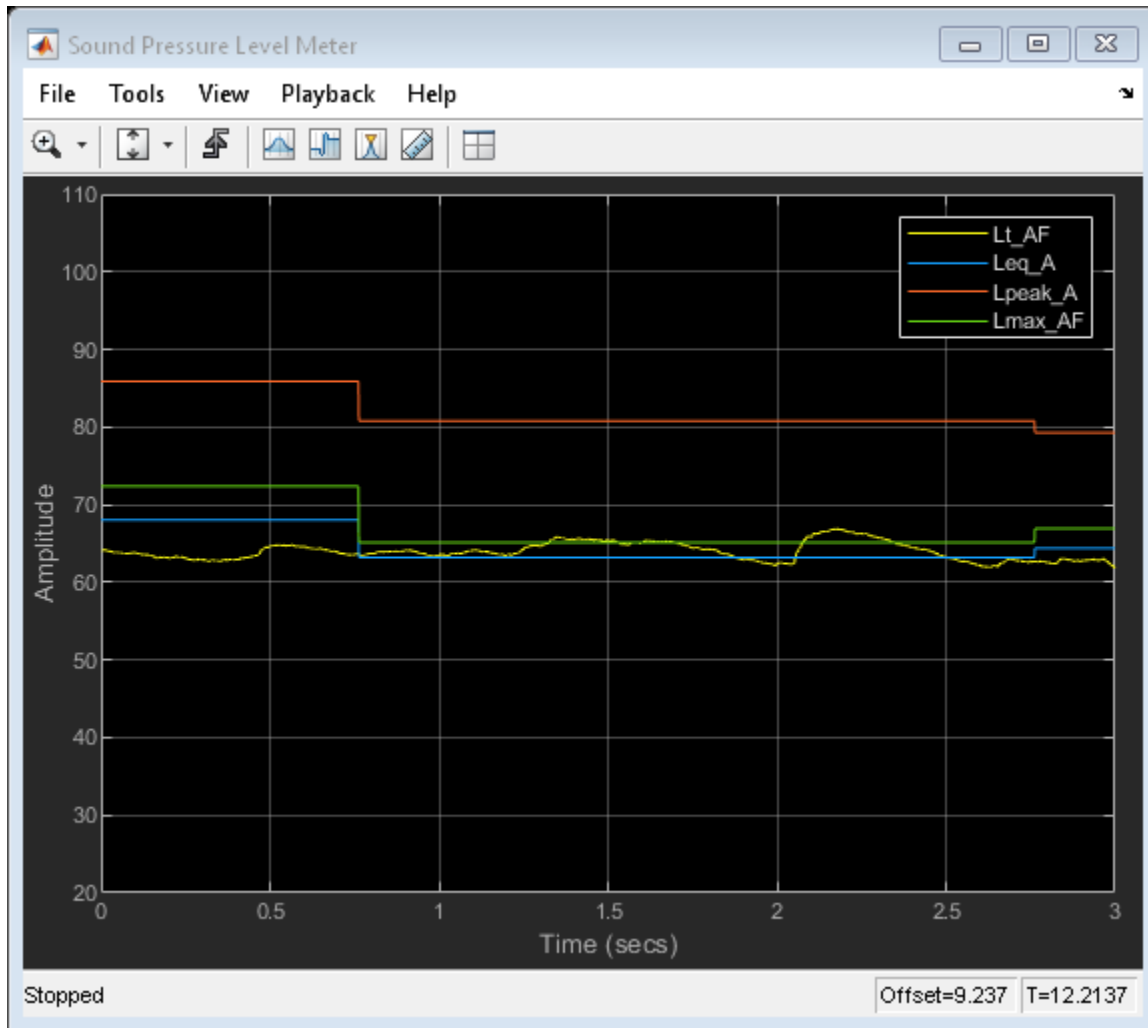
- 1 Read in the audio signal frame.
- 2 Play the audio signal to your output device.
- 3 Call the SPL meter to return the time-weighted, equivalent-continuous, peak, and maximum time-weighted sound levels in dB.
- 4 Display the sound levels using the scope.

As a best practice, release your objects once complete.

```
while ~isDone(source)
    x = source();
    player(x);
```

```
[Lt,Leq,Lpeak,Lmax] = SPL(x);  
scope([Lt,Leq,Lpeak,Lmax])  
end  
  
release(source)  
release(player)  
release(SPL)  
release(scope)
```





### Octave SPL Metering

The `splMeter` enables you to monitor sound pressure level for octave and fractional-octave bands. In this example, you monitor the equivalent-continuous sound pressure level of 1/3-octave bands.

Create a `dsp.AudioFileReader` object to read in an audio file frame by frame. Create an `audioDeviceWriter` object so you can listen to the audio signal. Create an `splMeter` to measure the octave sound pressure level of the audio file. Use the default calibration factor of 1. Create a `dsp.ArrayPlot` object to visualize the equivalent-continuous SPL for each octave band.

```
source = dsp.AudioFileReader('JetAirplane-16-11p025-mono-16secs.wav');
fs = source.SampleRate;

player = audioDeviceWriter('SampleRate',fs);

SPL = splMeter( ...
    'Bandwidth','1/3 octave', ...
    'SampleRate',fs);
centerFrequencies = getCenterFrequencies(SPL);

scope = dsp.ArrayPlot(...
    'XDataMode','Custom', ...
    'CustomXData',centerFrequencies, ...
    'XLabel','Octave Band Center Frequencies (Hz)', ...
    'YLabel','Equivalent-Continuous Sound Level (dB)', ...
    'YLimits',[20 90], ...
    'ShowGrid',true, ...
    'Name','Sound Pressure Level Meter');
```

In an audio stream loop:

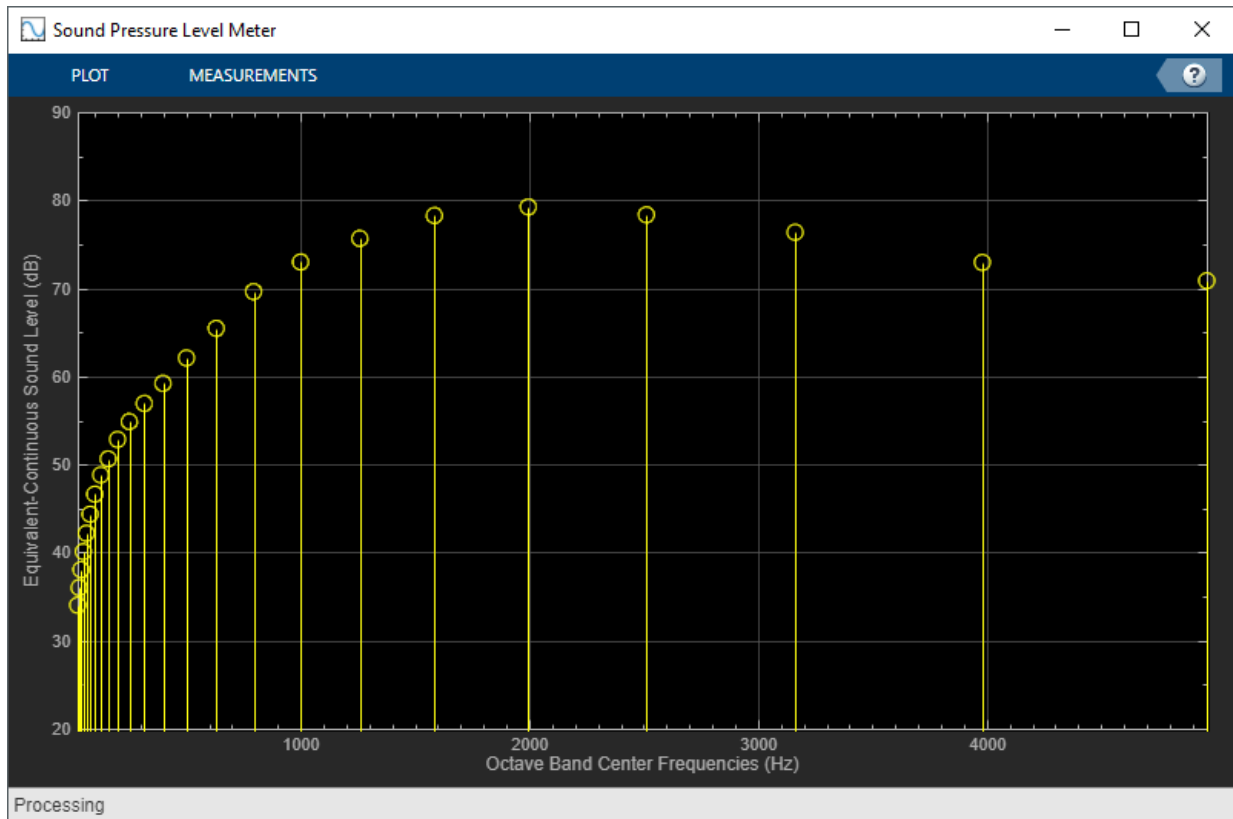
- 1 Read in the audio signal frame.
- 2 Play the audio signal to your output device.
- 3 Call the SPL meter to return the equivalent-continuous sound pressure level in dB.
- 4 Display the sound levels using the scope. Update the scope only when the equivalent-continuous sound pressure level has changed.

As a best practice, release your objects once complete.

```
LeqPrevious = zeros(size(centerFrequencies));
while ~isDone(source)
    x = source();
    player(x);
    [~,Leq] = SPL(x);

    for i = 1:size(Leq,1)
        if LeqPrevious ~= Leq(i,:)
            LeqPrevious(i,:) = Leq(i,:);
        end
    end
end
```

```
        scope(Leq(i,:))  
        LeqPrevious = Leq(i,:);  
    end  
end  
  
end  
  
release(source)  
release(player)  
release(SPL)  
release(scope)
```



## Algorithms

Sound pressure level calculations follow the algorithms described in [1]. You can specify property values to conform to standards [2] and [3].

## Calibration

To account for environmental and input device effects in SPL measurements, the audio input is multiplied by a calibration factor:

$$x = \text{audioIn} \times \text{CalibrationFactor}$$

The `CalibrationFactor` property can be set directly, or by using the `calibrate` function, which compares a known level with acquired data. The known level is determined using a physical calibrator.

## Frequency Weighting

A-, C-, or Z-frequency weighting is applied. The frequency weighting is implemented using the `weightingFilter` System object.

## Analysis Bands

If you specify the `Bandwidth` property as '1 octave', '2/3 octave' or '1/3 octave', then the SPL calculations are applied to each octave or fractional-octave band. These analysis bands are determined after frequency weighting.

## Time-Weighted Sound Level

Time-weighted sound level is defined as the ratio of the time-weighted root mean squared sound pressure to the reference sound pressure, converted to dB. That is,

$$L_t = 10 \log_{10} \left\{ \frac{\left( \frac{1}{\tau} \int_s^t y(\xi)^2 e^{-(t-\xi)/\tau} d\xi \right)}{p_0^2} \right\}$$

$$= 10 \log_{10} \left\{ \frac{h(y^2)}{p_0^2} \right\}$$

$h(y^2)$  can be interpreted as the convolution of  $y^2$  with a filter with impulse response  $(1/\tau)e^{-t/\tau}$ .  $y$  is the output of the frequency-weighting filter. The impulse response corresponds to a lowpass filter of the form  $H(s) = \frac{1/\tau}{s + 1/\tau}$ . Using impulse invariance, the discrete filter can be interpreted as,

$$H(z) = \frac{1/(\tau \times fs)}{1 - e^{-1/(\tau \times fs)} z^{-1}} .$$

- $\tau$  is specified by the time-weighting coefficient as 0.125 (if `TimeWeighting` is set to 'Fast') or 1 (if `TimeWeighting` is set to 'Slow').

- $fs$  is the sample rate specified by the `SampleRate` property.

## Equivalent-Continuous Sound Level

Equivalent-continuous sound level is also called time-average sound level. It is defined as the ratio of root mean squared sound pressure to the reference sound pressure, converted to dB. That is,

$$Leq = 10\log_{10}\left\{\frac{(1/T)\int_1^{t_2} y^2 dt}{p_o^2}\right\}$$
$$= 20\log_{10}(\text{rms}(y)/p_o)$$

where

- $y$  is the output of the frequency-weighting filter.
- $p_o$  is the reference sound pressure, specified by the `PressureReference` property.

## Peak Sound Level

Peak sound level is defined as the ratio of peak sound pressure to the reference sound pressure, converted to dB. That is,

$$L_{peak} = 20\log_{10}(\max(|y|)/p_o)$$

where

- $y$  is the output of the frequency-weighting filter.
- $p_o$  is the reference sound pressure, specified by the `PressureReference` property.

## Max Time-Weighted Sound Level

Maximum time-weighted sound level is defined as the greatest time-weighted sound level within a stated time interval.

## References

- [1] Harris, Cyril M. *Handbook of Acoustical Measurements and Noise Control*. 3rd ed. American Institute of Physics, 1998.
- [2] International Electrotechnical Commission. *Electroacoustics - Sound level meters - Part 1: Specifications*. IEC 61672-1:2013.
- [3] American National Standards Institute. *ANSI S1.4: Specification for Sound Level Meters*. 1983.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

Loudness Meter | `integratedLoudness` | `loudnessMeter`

### Topics

“Sound Pressure Measurement of Octave Frequency Bands”

**Introduced in R2018a**

# calibrate

Calibrate meter using calibration tone with known level

## Syntax

```
calibrate(SPL,calibrationTone,trueLevel)
```

## Description

`calibrate(SPL,calibrationTone,trueLevel)` sets the `CalibrationFactor` property based on the computed sound pressure level of `calibrationTone` and the known `trueLevel`. `trueLevel` refers to the physical calibrator level used to generate the `calibrationTone`.

## Input Arguments

### **SPL** — `splMeter` System object

object

`splMeter` System object to be calibrated.

### **calibrationTone** — Audio signal used to calibrate SPL meter

column vector

Audio signal used to calibrate the SPL meter, specified as a column vector.

Data Types: `single` | `double`

### **trueLevel** — True level of calibration tone (dB)

scalar

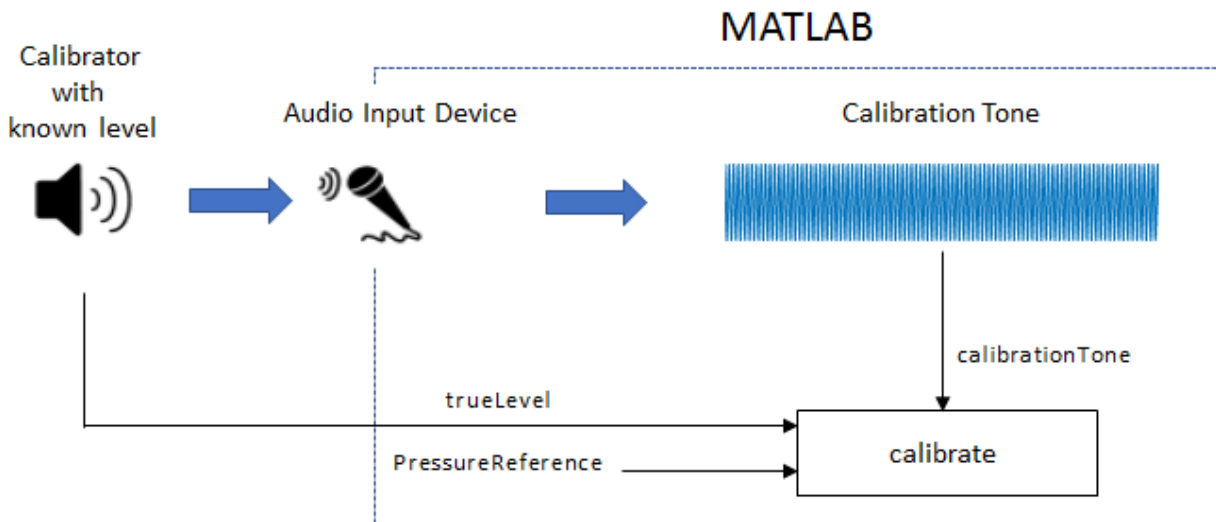
True level of calibration tone in dB, specified as a scalar. The true level is the known level of output by a physical calibrator.

Data Types: `single` | `double`



## Algorithms

To set the `CalibrationFactor` property on an `splMeter` object, the `calibrate` function uses a calibration tone, the known level output by the calibrator to produce the calibration tone, and the `PressureReference` property.



The `CalibrationFactor` property is set according to the equation:

$$\text{CalibrationFactor} = \frac{10^{(\text{trueLevel}-k)/20}}{\text{rms}(\text{calibrationTone})}$$

where  $k$  is 1 pascal relative to the reference pressure calculated in dB:

$$k = 20 \log_{10} \left( \frac{1}{\text{PressureReference}} \right).$$

## See Also

`splMeter`

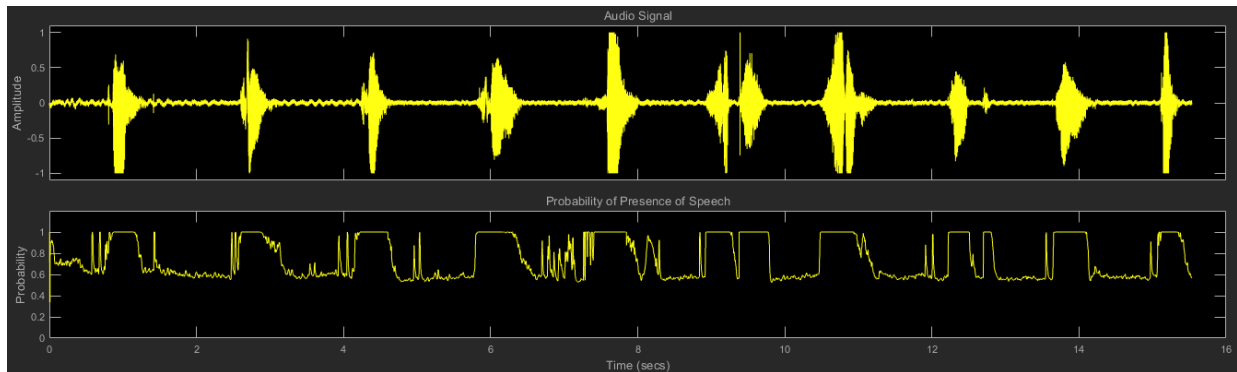
**Introduced in R2018a**

## voiceActivityDetector

Detect presence of speech in audio signal

### Description

The `voiceActivityDetector` System object detects the presence of speech in an audio segment. You can also use the `voiceActivityDetector` System object to output an estimate of the noise variance per frequency bin.



To detect the presence of speech:

- 1 Create the `voiceActivityDetector` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

### Creation

### Syntax

```
VAD = voiceActivityDetector
```

VAD = voiceActivityDetector(Name,Value)

## Description

VAD = voiceActivityDetector creates a System object, VAD, that detects the presence of speech independently across each input channel.

VAD = voiceActivityDetector(Name,Value) sets each property Name to the specified Value. Unspecified properties have default values.

Example: VAD = voiceActivityDetector('InputDomain','Frequency') creates a System object, VAD, that accepts frequency-domain input.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects (MATLAB)*.

### **InputDomain** — Domain of input signal

'Time' (default) | 'Frequency'

Domain of the input signal, specified as 'Time' or 'Frequency'.

**Tunable:** No

Data Types: char | string

### **FFTLength** — FFT length

[] (default) | positive scalar

FFT length, specified as a positive scalar. The default is [], which means that the `FFTLength` is equal to the number of rows of the input.

**Tunable:** No

#### **Dependencies**

To enable this property, set `InputDomain` to `'Time'`.

Data Types: `single` | `double`

#### **Window — Window function for FFT**

`'Hann'` (default) | `'Chebyshev'` | `'Flat Top'` | `'Hamming'` | `'Kaiser'` | `'Rectangular'`

Time-domain window function applied before calculating the discrete-time Fourier transform (DTFT), specified as `'Hann'`, `'Rectangular'`, `'Flat Top'`, `'Hamming'`, `'Chebyshev'`, or `'Kaiser'`.

The window function is designed using the algorithms of the following functions:

- `Hann` -- `hann`
- `Chebyshev` -- `chebwin`
- `Flat Top` -- `flattopwin`
- `Hamming` -- `hamming`
- `Kaiser` -- `kaiser`

**Tunable:** No

#### **Dependencies**

To enable this property, set `InputDomain` to `'Time'`.

Data Types: `char` | `string`

#### **SidelobeAttenuation — Sidelobe attenuation of window (dB)**

60 (default) | real positive scalar

Sidelobe attenuation of the window in dB, specified as a real positive scalar.

**Tunable:** No

#### **Dependencies**

To enable this property, set `InputDomain` to `'Time'` and `Window` to `'Chebyshev'` or `'Kaiser'`.

Data Types: `single` | `double`

**SilenceToSpeechProbability** — Probability of transition from a frame of silence to a frame of speech

0.2 (default) | scalar in the range [0,1]

Probability of transition from a frame of silence to a frame of speech, specified as a scalar in the range [0,1].

**Tunable:** Yes

Data Types: single | double

**SpeechToSilenceProbability** — Probability of transition from a frame of speech to a frame of silence

0.1 (default) | scalar in the range [0,1]

Probability of transition from a frame of speech to a frame of silence, specified as a scalar in the range [0,1].

**Tunable:** Yes

Data Types: single | double

## Usage

## Syntax

```
[probability,noiseEstimate] = VAD(audioIn)
```

## Description

[probability,noiseEstimate] = VAD(audioIn) applies a voice activity detector on the input, audioIn, and returns the probability that speech is present. It also returns the estimated noise variance per frequency bin.

## Input Arguments

**audioIn** — Audio input to voice activity detector

scalar | vector | matrix

Audio input to the voice activity detector, specified as a scalar, vector, or matrix. If `audioIn` is a matrix, the columns are treated as independent audio channels.

The size of the audio input is locked after the first call to the `voiceActivityDetector` object. To change the size of `audioIn`, call `release` on the object.

If `InputDomain` is set to `'Time'`, `audioIn` must be real-valued. If `InputDomain` is set to `'Frequency'`, `audioIn` can be real-valued or complex-valued.

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

### Output Arguments

#### **probability** — Probability that speech is present

scalar | row vector

Probability that speech is present, returned as a scalar or row vector with the same number of columns as `audioIn`.

Data Types: `single` | `double`

#### **noiseEstimate** — Estimate of noise variance per frequency bin

column vector | matrix

Estimate of the noise variance per frequency bin, returned as a column vector or matrix with the same number of columns as `audioIn`.

Data Types: `single` | `double`

### Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Common to All System Objects

`clone`      Create duplicate System object  
`isLocked`   Determine if System object is in use

release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

## Examples

### Detect Voice Activity

Use the default `voiceActivityDetector` System object? to detect the presence of speech in a streaming audio signal.

Create an audio file reader to stream an audio file for processing. Define parameters to chunk the audio signal into 10 ms non-overlapping frames.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
fs = fileReader.SampleRate;
fileReader.SamplesPerFrame = ceil(10e-3*fs);
```

Create a default `voiceActivityDetector` System object to detect the presence of speech in the audio file.

```
VAD = voiceActivityDetector;
```

Create a scope to plot the audio signal and corresponding probability of speech presence as detected by the voice activity detector. Create an audio device writer to play the audio through your sound card.

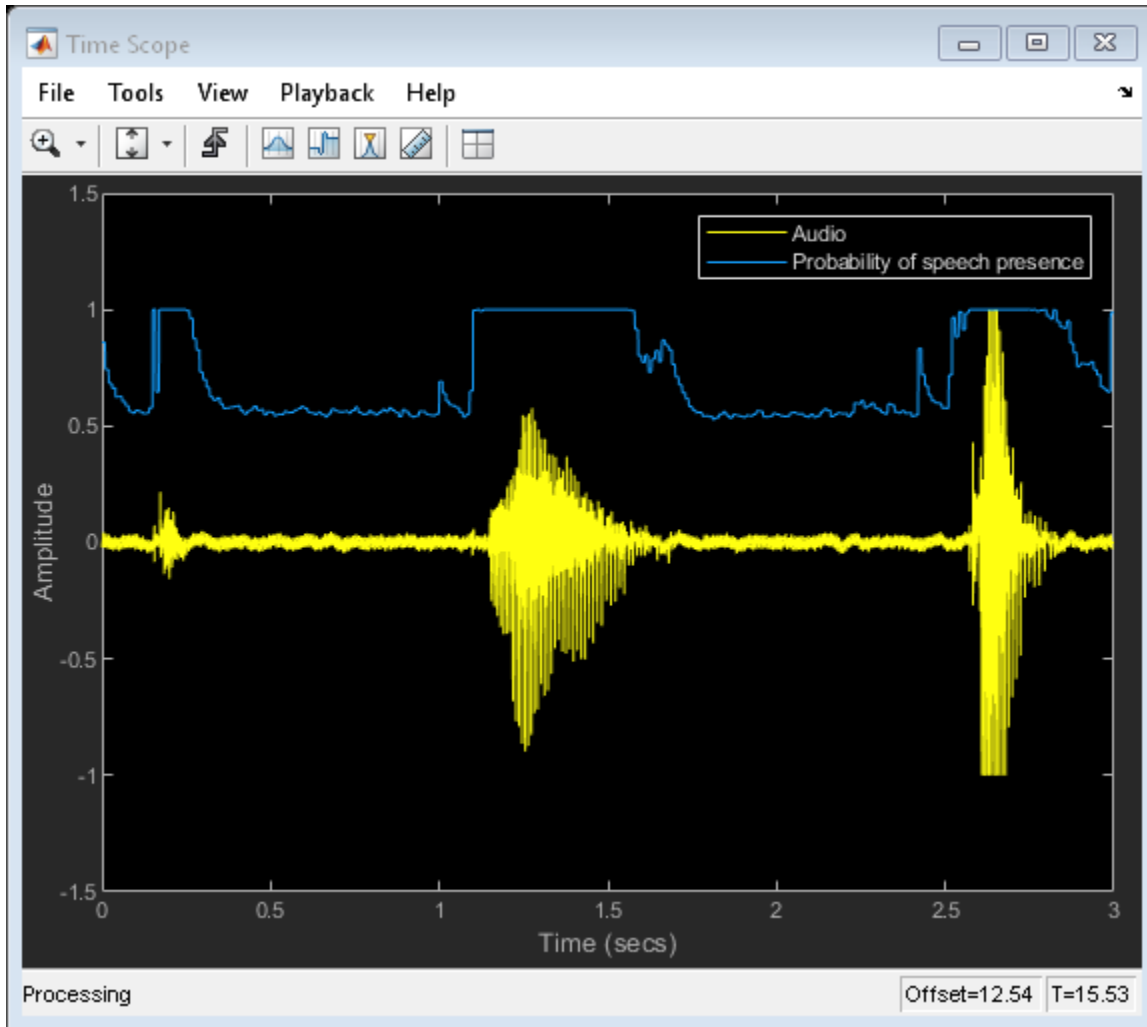
```
scope = dsp.TimeScope( ...
    'NumInputPorts',2, ...
    'SampleRate',fs, ...
    'TimeSpan',3, ...
    'BufferLength',3*fs, ...
    'YLimits',[-1.5 1.5], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowLegend',true, ...
    'ChannelNames',{'Audio','Probability of speech presence'});
deviceWriter = audioDeviceWriter('SampleRate',fs);
```

In an audio stream loop:

- 1 Read from the audio file.
- 2 Calculate the probability of speech presence.
- 3 Visualize the audio signal and speech presence probability.
- 4 Play the audio signal through your sound card.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    probability = VAD(audioIn);
    scope(audioIn,probability*ones(fileReader.SamplesPerFrame,1))
    deviceWriter(audioIn);
end
```





### Detect Voice Activity Using Overlapped Frames

Use a voice activity detector to detect the presence of speech in an audio signal. Plot the probability of speech presence along with the audio samples.

Create a `dsp.AudioFileReader` System object? to read a speech file.

```
afr = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');  
fs = afr.SampleRate;
```

Chunk the audio into 20 ms frames with 75% overlap between successive frames. Convert the frame time in seconds to samples. Determine the hop size (the increment of new samples). In the audio file reader, set the samples per frame to the hop size. Create a default `dsp.AsyncBuffer` object to manage overlapping between audio frames.

```
frameSize = ceil(20e-3*fs);  
overlapSize = ceil(0.75*frameSize);  
hopSize = frameSize - overlapSize;  
afr.SamplesPerFrame = hopSize;  
  
inputBuffer = dsp.AsyncBuffer('Capacity',frameSize);
```

Create a `voiceActivityDetector` System object. Specify an FFT length of 1024.

```
VAD = voiceActivityDetector('FFTLength',1024);
```

Create a scope to plot the audio signal and corresponding probability of speech presence as detected by the voice activity detector. Create an `audioDeviceWriter` System object to play audio through your sound card.

```
scope = dsp.TimeScope('NumInputPorts',2, ...  
    'SampleRate',fs, ...  
    'TimeSpan',3, ...  
    'BufferLength',3*fs, ...  
    'YLimits',[-1.5,1.5], ...  
    'TimeSpanOverrunAction','Scroll', ...  
    'ShowLegend',true, ...  
    'ChannelNames',{'Audio','Probability of speech presence'});
```

```
player = audioDeviceWriter('SampleRate',fs);
```

Initialize a vector to hold the probability values.

```
pHold = ones(hopSize,1);
```

In an audio stream loop:

- 1 Read a hop worth of samples from the audio file and save the samples into the buffer.
- 2 Read a frame from the buffer with specified overlap from the previous frame.
- 3 Call the voice activity detector to get the probability of speech for the frame under analysis.

- 4 Set the last element of the probability vector to the new probability decision. Visualize the audio and speech presence probability using the time scope.
- 5 Play the audio through your sound card.
- 6 Set the probability vector to the most recent result for plotting in the next loop.

```
while ~isDone(afr)
    x = afr();
    n = write(inputBuffer,x);

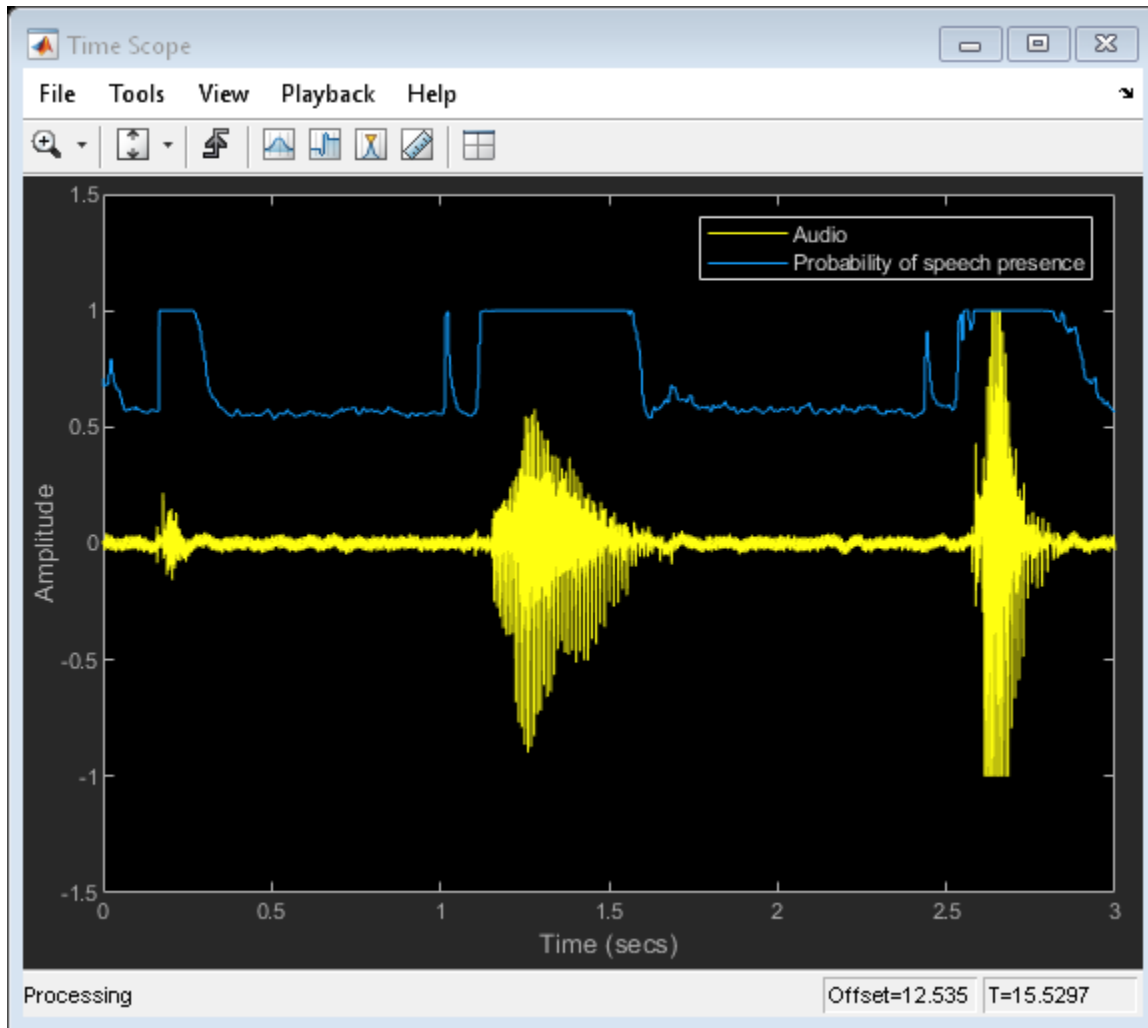
    overlappedInput = read(inputBuffer,frameSize,overlapSize);

    p = VAD(overlappedInput);

    pHold(end) = p;
    scope(x,pHold)

    player(x);

    pHold(:) = p;
end
```



Release the player once the audio finishes playing.

```
release(player)
```

## Frequency-Domain Voice Activity Detection and Cepstral Feature Extraction

Many feature extraction techniques operate on the frequency domain. Converting an audio signal to the frequency domain only once is efficient. In this example, you convert a streaming audio signal to the frequency domain and feed that signal into a voice activity detector. If speech is present, mel-frequency cepstral coefficients (MFCC) features are extracted from the frequency-domain signal using the `cepstralFeatureExtractor` System object™.

Create a `dsp.AudioFileReader` System object to read from an audio file.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
fs = fileReader.SampleRate;
```

Process the audio in 30 ms frames with a 10 ms hop. Create a default `dsp.AsyncBuffer` object to manage overlap between audio frames.

```
samplesPerFrame = ceil(0.03*fs);
samplesPerHop = ceil(0.01*fs);
samplesPerOverlap = samplesPerFrame - samplesPerHop;
```

```
fileReader.SamplesPerFrame = samplesPerHop;
buffer = dsp.AsyncBuffer;
```

Create a `voiceActivityDetector` System object and a `cepstralFeatureExtractor` System object. Specify that they operate in the frequency domain. Create a `dsp.SignalSink` to log the extracted cepstral features.

```
VAD = voiceActivityDetector('InputDomain', 'Frequency');
cepFeatures = cepstralFeatureExtractor('InputDomain', 'Frequency', 'SampleRate', fs, 'LogE');
sink = dsp.SignalSink;
```

In an audio stream loop:

- 1 Read one hop's of samples from the audio file and save the samples into the buffer.
- 2 Read a frame from the buffer with specified overlap from the previous frame.
- 3 Call the voice activity detector to get the probability of speech for the frame under analysis.
- 4 If the frame under analysis has a probability of speech greater than 0.75, extract cepstral features and log the features using the signal sink. If the frame under analysis has a probability of speech less than 0.75, write a vector of NaNs to the sink.

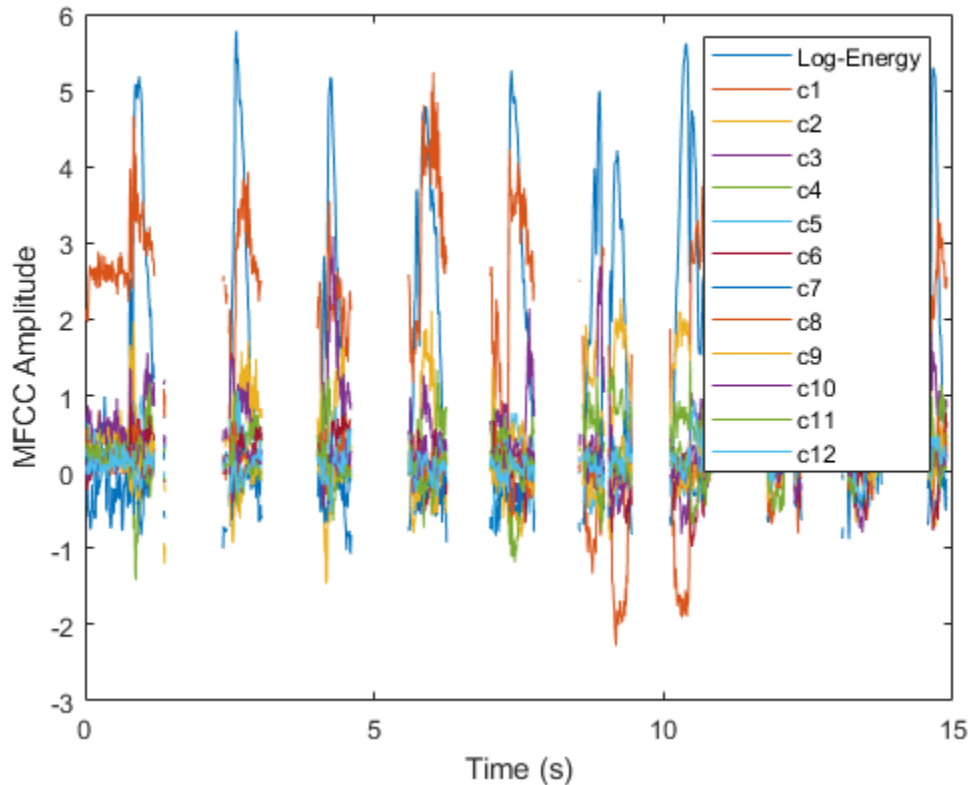
```
threshold = 0.75;
nanVector = nan(1,13);
while ~isDone(fileReader)
    audioIn = fileReader();
    write(buffer,audioIn);

    overlappedAudio = read(buffer,samplesPerFrame,samplesPerOverlap);
    X = fft(overlappedAudio,2048);

    probabilityOfSpeech = VAD(X);
    if probabilityOfSpeech > threshold
        xFeatures = cepFeatures(X);
        sink(xFeatures')
    else
        sink(nanVector)
    end
end
```

Visualize the cepstral coefficients over time.

```
timeVector = linspace(0,15,size(sink.Buffer,1));
plot(timeVector,sink.Buffer)
xlabel('Time (s)')
ylabel('MFCC Amplitude')
legend('Log-Energy','c1','c2','c3','c4','c5','c6','c7','c8','c9','c10','c11','c12')
```



### Determine Pitch Contour using `pitch` and `voiceActivityDetector`

Read in an entire speech file and determine the fundamental frequency of the audio using the `pitch` function. Then use the `voiceActivityDetector` to remove irrelevant pitch information that does not correspond to the speaker.

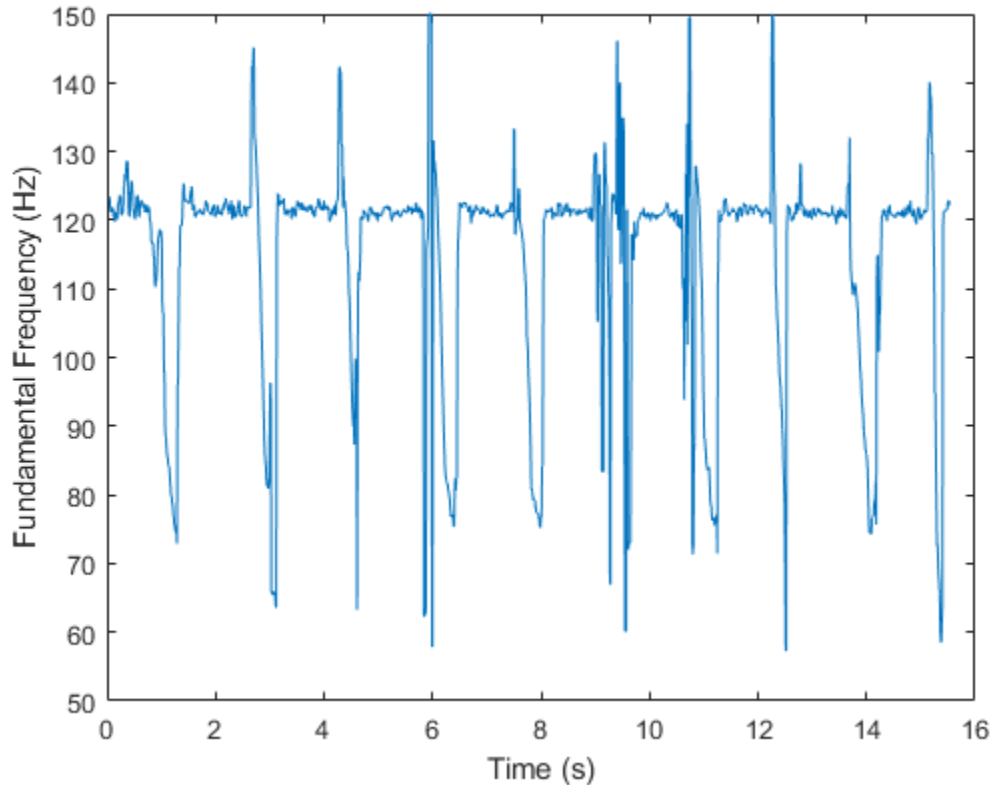
Read in the audio file and associated sample rate.

```
[audio,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Specify pitch detection using a 50 ms window length and 40 ms overlap (10 ms hop). Specify that the `pitch` function searches for the fundamental frequency over the range 50-150 Hz and postprocesses the results with a median filter. Plot the results.

```
windowLength = round(0.05*fs);  
overlapLength = round(0.04*fs);  
hopLength = windowLength - overlapLength;  
  
[f0,loc] = pitch(audio,fs, ...  
    'WindowLength',windowLength, ...  
    'OverlapLength',overlapLength, ...  
    'Range',[50 150], ...  
    'MedianFilterLength',3);  
  
plot(loc/fs,f0)  
ylabel('Fundamental Frequency (Hz)')  
xlabel('Time (s)')
```





Create a `dsp.AsyncBuffer` System object™ to chunk the audio signal into overlapped frames. Also create a `voiceActivityDetector` System object™ to determine if the frames contain speech.

```
buffer = dsp.AsyncBuffer(numel(audio));
write(buffer, audio);
VAD = voiceActivityDetector;
```

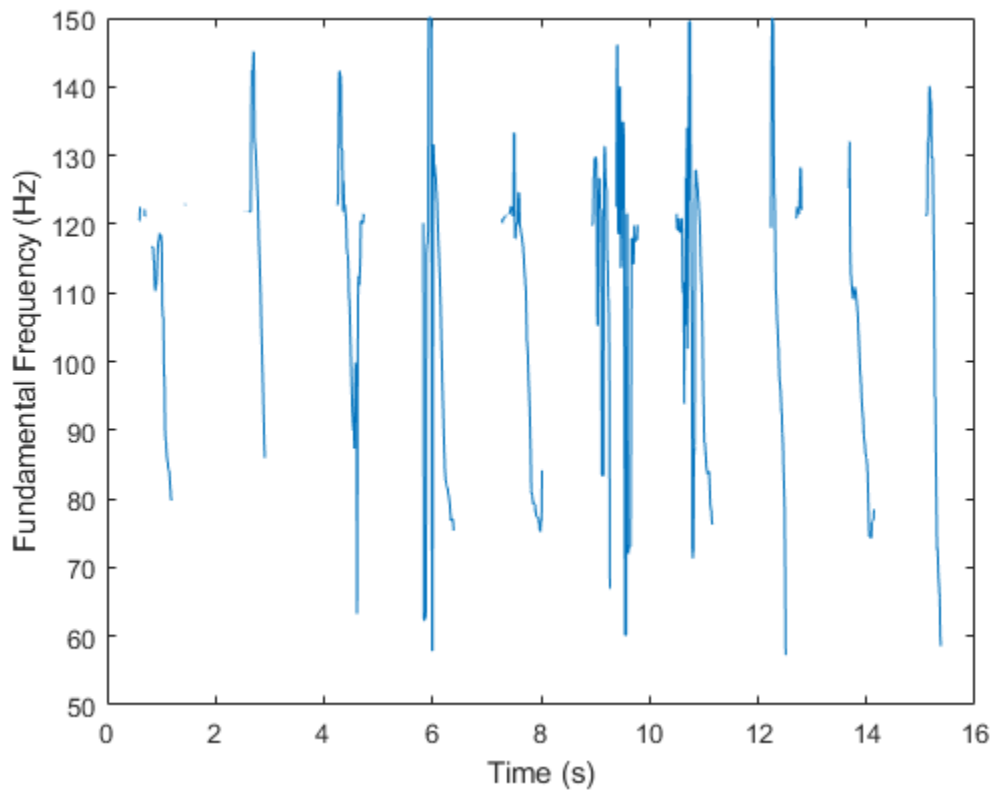
While there are enough samples to hop, read from the buffer and determine the probability that the frame contains speech. To mimic the decision spacing in time of the pitch function, the first frame read from the buffer has no overlap.

```
n = 1;
probabilityVector = zeros(numel(loc), 1);
```

```
while buffer.NumUnreadSamples >= hopLength
    if n==1
        x = read(buffer,windowLength);
    else
        x = read(buffer,windowLength,overlapLength);
    end
    probabilityVector(n) = VAD(x);
    n = n+1;
end
```

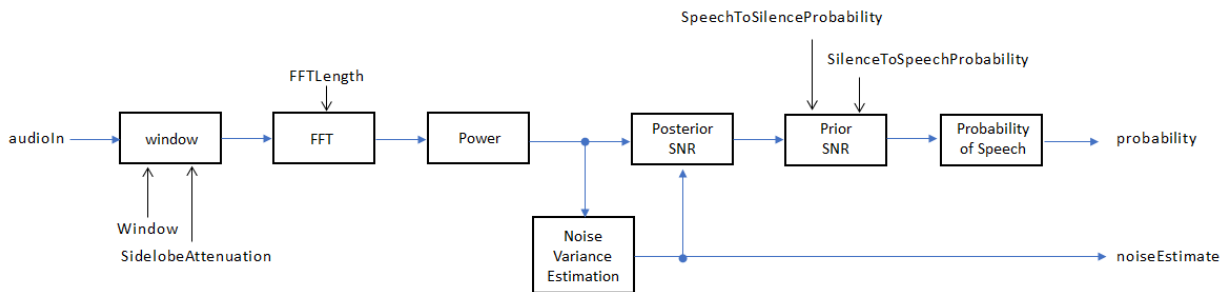
Use the probability vector determined by the `voiceActivityDetector` to plot a pitch contour for the speech file that corresponds to regions of speech.

```
validIdx = probabilityVector>0.99;
loc(~validIdx) = nan;
f0(~validIdx) = nan;
plot(loc/fs,f0)
ylabel('Fundamental Frequency (Hz)')
xlabel('Time (s)')
```



## Algorithms

The `voiceActivityDetector` implements the algorithm described in [1].



If `InputDomain` is specified as 'Time', the input signal is windowed and then converted to the frequency domain according to the `Window`, `SidelobeAttenuation`, and `FFTLength` properties. If `InputDomain` is specified as frequency, the input is assumed to be a windowed discrete time Fourier transform (DTFT) of an audio signal. The signal is then converted to the power domain. Noise variance is estimated according to [2]. The posterior and prior SNR are estimated according to the Minimum Mean-Square Error (MMSE) formula described in [3]. A log likelihood ratio test and Hidden Markov Model (HMM)-based hang-over scheme determine the probability that the current frame contains speech, according to [1].

## References

- [1] Sohn, Jongseo., Nam Soo Kim, and Wonyong Sung. "A Statistical Model-Based Voice Activity Detection." *Signal Processing Letters IEEE*. Vol. 6, No. 1, 1999.
- [2] Martin, R. "Noise Power Spectral Density Estimation Based on Optimal Smoothing and Minimum Statistics." *IEEE Transactions on Speech and Audio Processing*. Vol. 9, No. 5, 2001, pp. 504-512.
- [3] Ephraim, Y., and D. Malah. "Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 32, No. 6, 1984, pp. 1109-1121.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

## **See Also**

Voice Activity Detector | [audioFeatureExtractor](#) | [cepstralFeatureExtractor](#) | [mfcc](#) | [pitch](#)

**Introduced in R2018a**

# cepstralFeatureExtractor

Extract cepstral features from audio segment

## Description

The `cepstralFeatureExtractor` System object extracts cepstral features from an audio segment. Cepstral features are commonly used to characterize speech and music signals.

To extract cepstral features:

- 1 Create the `cepstralFeatureExtractor` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

## Syntax

```
cepFeatures = cepstralFeatureExtractor  
cepFeatures = cepstralFeatureExtractor(Name,Value)
```

## Description

`cepFeatures = cepstralFeatureExtractor` creates a System object, `cepFeatures`, that calculates cepstral features independently across each input channel. Columns of the input are treated as individual channels.

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

Example: `cepFeatures = cepstralFeatureExtractor('InputDomain','Frequency','SampleRate',fs,'`

`LogEnergy` , `'Replace'` ) accepts a signal in the frequency domain, sampled at `fs` Hz. The first element of the coefficients vector is replaced by the log energy value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects \(MATLAB\)](#).

### FilterBank — Type of filter bank

`'Mel'` (default) | `'Gammatone'`

Type of filter bank, specified as either `'Mel'` or `'Gammatone'`. When `FilterBank` is set to `Mel`, the object computes the mel frequency cepstral coefficients (MFCC). When `FilterBank` is set to `Gammatone`, the object computes the gammatone cepstral coefficients (GTCC).

Data Types: `char` | `string`

### InputDomain — Domain of input signal

`'Time'` (default) | `'Frequency'`

Domain of the input signal, specified as either `'Time'` or `'Frequency'`.

Data Types: `char` | `string`

### NumCoeffs — Number of coefficients to return

13 (default) | positive integer

Number of coefficients to return, specified as an integer in the range  $[2, v]$ , where  $v$  is the number of valid passbands. The number of valid passbands depends on the type of filter bank:

- `Mel` -- The number of valid passbands is defined as `sum(BandEdges <= floor(SampleRate/2)) - 2`.
- `Gammatone` -- The number of valid passbands is defined as `ceil(hz2erb(FrequencyRange(2)) - hz2erb(FrequencyRange(1)))`.

Data Types: `single` | `double`

#### **FFTLength** — FFT length

`[]` (default) | positive integer

FFT length, specified as a positive integer. The default, `[]`, means that the FFT length is equal to the number of rows in the input signal.

#### **Dependencies**

To enable this property, set `InputDomain` to `'Time'`.

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

#### **LogEnergy** — Specify how the log energy is shown

`'Append'` (default) | `'Replace'` | `'Ignore'`

Specify how the log energy is shown in the coefficients vector output, specified as:

- `'Append'` -- The object prepends the log energy to the coefficients vector. The length of the coefficients vector is `1 + NumCoeffs`.
- `'Replace'` -- The object replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is `NumCoeffs`.
- `'Ignore'` -- The object does not calculate or return the log energy.

Data Types: `char` | `string`

#### **SampleRate** — Input sample rate (Hz)

16000 (default) | positive scalar

Input sample rate in Hz, specified as a real positive scalar.

**Tunable:** Yes

Data Types: `single` | `double`

#### **Advanced properties**

#### **BandEdges** — Band edges of mel filter bank (Hz)

row vector



Band edges of the filter bank in Hz, specified as a nonnegative monotonically increasing row vector in the range  $[0, \infty)$ . The maximum bandedge frequency can be any finite number. The number of bandedges must be in the range  $[4, 80]$ .

The default band edges are spaced linearly for the first ten and then logarithmically after. The default band edges are set as recommended by [1].

### Dependencies

To enable this property, set `FilterBank` to `Mel`.

Data Types: `single` | `double`

### FrequencyRange — Frequency range of gammatone filter bank (Hz)

`[50 8000]` (default) | two-element row vector

Frequency range of the filter bank in Hz, specified as a positive, monotonically increasing two-element row vector. The maximum frequency can be any finite number. The center frequencies of the filter bank are equally spaced between `hz2erb(FrequencyRange(1))` and `hz2erb(FrequencyRange(2))` on the ERB scale.

### Dependencies

To enable this property, set `FilterBank` to `Gammatone`.

Data Types: `single` | `double`

### FilterBankDesignDomain — Domain for mel filter bank design

`'Hz'` (default) | `'Bin'`

Domain for filter bank design, specified as either `'Hz'` or `'Bin'`. The filter bank is designed as overlapped triangles with band edges specified by the `BandEdges` property.

The `BandEdges` property is specified in Hz. When you set the design domain to:

- `'Hz'` -- Filter bank triangles are drawn in Hz and are mapped onto bins.

Here is an example that plots the filter bank in bins when the `FilterBankDesignDomain` is set to `'Hz'`:

```
[audioFile, fs] = audioread('NoisySpeech-16-22p5-mono-5secs.wav');
duration = round(0.02*fs); % 20 ms audio segment
audioSegment = audioFile(5500:5500+duration-1);
cepFeatures = cepstralFeatureExtractor('SampleRate', fs)
```

```
cepFeatures =  
cepstralFeatureExtractor with properties:
```

```
Properties
```

```
InputDomain: 'Time'  
NumCoeffs: 13  
FFTLength: []  
LogEnergy: 'Append'  
SampleRate: 22500
```

```
Advanced Properties
```

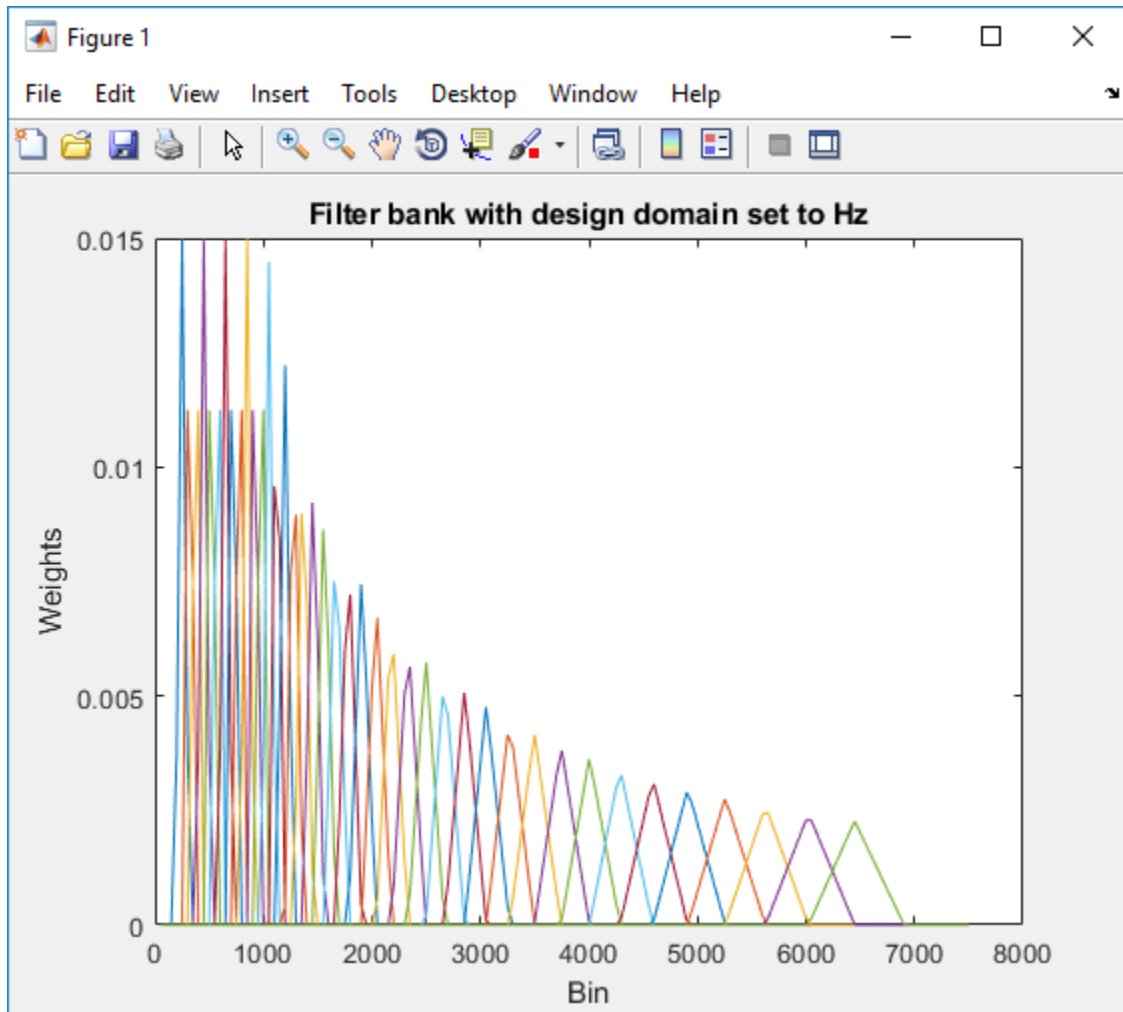
```
BandEdges: [1×42 double]  
FilterBankDesignDomain: 'Hz'  
FilterBankNormalization: 'Bandwidth'
```

Pass the audio segment as an input to the cepstral feature extractor algorithm to lock the object.

```
[coeffs,delta,deltaDelta] = cepFeatures(audioSegment);
```

Using the `getFilters` function, get the filter bank. Plot the filter bank.

```
[filterbank, freq] = getFilters(cepFeatures);  
plot(freq(1:150),filterbank(1:150,:))
```



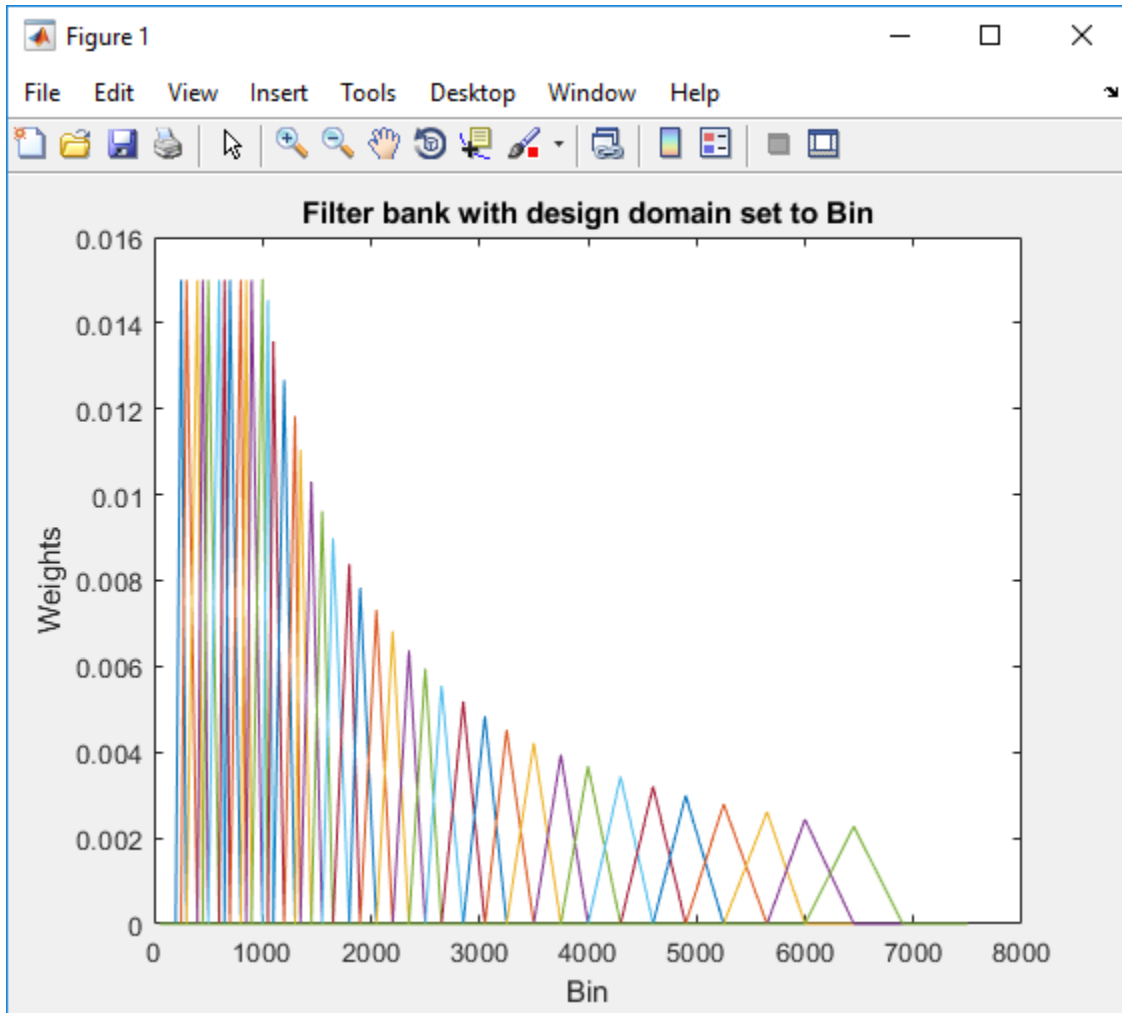
For details, see [1].

- 'Bin' -- The bandedge frequencies in 'Hz' are converted to bins. The filter bank triangles are drawn symmetrically in bins.

Change the `FilterBankDesignDomain` property to 'Bin':

```
release(cepFeatures);
cepFeatures.FilterBankDesignDomain = 'Bin';
```

```
[coeffs,delta,deltaDelta] = cepFeatures(audioSegment);  
[filterbank, freq] = getFilters(cepFeatures);  
plot(freq(1:150),filterbank(1:150,:))
```



For details, see [2].

## Dependencies

To enable this property, set `FilterBank` to `Mel`.

Data Types: `char` | `string`

## FilterBankNormalization — Normalize filter bank

'Bandwidth' (default) | 'Area' | 'None'

Normalization technique used on the weights of the filter bank, specified as:

- 'Bandwidth' -- The weights of each bandpass filter are normalized by the corresponding bandwidth of the filter.
- 'Area' -- The weights of each bandpass filter are normalized by the corresponding area of the bandpass filter.
- 'None' -- The weights of the filter are not normalized.

Data Types: `char` | `string`

## Usage

## Syntax

```
[coeffs,delta,deltaDelta] = cepFeatures(audioIn)
```

## Description

`[coeffs,delta,deltaDelta] = cepFeatures(audioIn)` returns the cepstral coefficients, the log energy, the delta, and the delta-delta.

The log energy value prepends the coefficient vector or replaces the first element of the coefficients vector based on whether you set the `LogEnergy` property to 'Append' or 'Replace'. For details, see “coeffs” on page 3-0 .

## Input Arguments

### audioIn — Input signal

column vector | matrix

Input signal, specified as a column vector or matrix. If `InputDomain` is set to `'Time'`, specify `audioIn` as a real-valued frame of audio data. If `InputDomain` is set to `'Frequency'`, specify `audioIn` as a real- or complex-valued discrete Fourier transform. If specified as a matrix, the columns are treated as independent audio channels.

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

## Output Arguments

### `coeffs` — Cepstral coefficients

column vector | matrix

Cepstral coefficients, returned as a column vector or a matrix. If the coefficients matrix is an  $N$ -by- $M$  matrix,  $N$  is determined by the values you specify in `NumCoeffs` and `LogEnergy` properties.  $M$  equals the number of input audio channels.

When the `LogEnergy` property is set to:

- `'Append'` -- The object prepends the log energy value to the coefficients vector. The length of the coefficients vector is  $1 + \text{NumCoeffs}$ . This is the default setting of the `LogEnergy` property.
- `'Replace'` -- The object replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is `NumCoeffs`.
- `'Ignore'` -- The object does not calculate or return the log energy.

Data Types: `single` | `double`

### `delta` — Change in coefficients

column vector | matrix

Change in coefficients over consecutive calls to the algorithm, returned as a vector or a matrix. The `delta` array is of the same size and data type as the `coeffs` array.

In this example, `cepFeatures` is the cepstral feature extractor that accepts audio input signal sampled at 12 kHz. Stream in three segments of audio signal on three consecutive calls to the object algorithm. Return the cepstral coefficients of the filter bank and the corresponding `delta` values.

```
cepFeatures = cepstralFeatureExtractor('SampleRate',12000);  
[coeff1,delta1] = cepFeatures(audioIn);  
[coeff2,delta2] = cepFeatures(audioIn);  
[coeff3,delta3] = cepFeatures(audioIn);
```

$\text{delta2}$  is computed as  $\text{coeff2} - \text{coeff1}$ , while  $\text{delta3}$  is computed as  $\text{coeff3} - \text{coeff2}$ . The initial array,  $\text{delta1}$ , is an array of zeros.

Data Types: `single` | `double`

### **deltaDelta — Change in delta values**

column vector | matrix

Change in  $\text{delta}$  values over consecutive calls to the algorithm, returned as a vector or a matrix. The  $\text{deltaDelta}$  array is the same size and data type as the  $\text{coeffs}$  and  $\text{delta}$  arrays.

In this example, consecutive calls to the cepstral feature extractor algorithm return the  $\text{deltaDelta}$  values in addition to the coefficients and the  $\text{delta}$  values.

```
cepFeatures = cepstralFeatureExtractor('SampleRate',12000);
[coeff1,delta1,deltaDelta1] = cepFeatures(audioIn);
[coeff2,delta2,deltaDelta2] = cepFeatures(audioIn);
[coeff3,delta3,deltaDelta3] = cepFeatures(audioIn);
```

$\text{deltaDelta2}$  is computed as  $\text{delta2} - \text{delta1}$ , while  $\text{deltaDelta3}$  is computed as  $\text{delta3} - \text{delta2}$ . The initial array,  $\text{deltaDelta1}$ , is an array of zeros.

Data Types: `single` | `double`

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### **Specific to cepstralFeatureExtractor**

`getFilters` Get auditory filter bank

### **Common to All System Objects**

<code>clone</code>	Create duplicate System object
<code>isLocked</code>	Determine if System object is in use
<code>release</code>	Release resources and allow changes to System object property values and input characteristics
<code>reset</code>	Reset internal states of System object

step          Run System object algorithm

## Examples

### Get MFCC Data for Speech Segment

Extract the mel frequency cepstral coefficients and the log energy values of segments in a speech file. Return `delta`, the difference between current and the previous cepstral coefficients, and `deltaDelta`, the difference between the current and the previous `delta` values. The log energy value the object computes can prepend the coefficients vector or replace the first element of the coefficients vector. This is done based on whether you set the `LogEnergy` property of the `cepstralFeatureExtractor` object to `'Replace'` or `'Append'`.

Read an audio signal from `'SpeechDFT-16-8-mono-5secs.wav'` file. Extract a 40 ms segment from the audio data. Create a `cepstralFeatureExtractor` object. The cepstral coefficients computed by the default object are the mel frequency coefficients. In addition, the object computes the log energy, `delta`, and `delta-delta` values of the audio segment.

```
[audioFile, fs] = audioread('SpeechDFT-16-8-mono-5secs.wav');
duration = round(0.04*fs); % 40 ms
audioSegment = audioFile(5500:5500+duration-1);
cepFeatures = cepstralFeatureExtractor('SampleRate', fs)
```

```
cepFeatures =
  cepstralFeatureExtractor with properties:
```

```
Properties
  FilterBank: 'Mel'
  InputDomain: 'Time'
  NumCoeffs: 13
  FFTLength: []
  LogEnergy: 'Append'
  SampleRate: 8000
```

```
Show all properties
```

The `LogEnergy` property is set to `'Append'`. The first element in the coefficients vector is the log energy value and the remaining elements are the 13 cepstral coefficients



computed by the object. The number of cepstral coefficients is determined by the value you specify in the NumCoeffs property.

```
[coeffs,delta,deltaDelta] = cepFeatures(audioSegment)
```

```
coeffs = 14×1
```

```
    3.8281  
   -19.4827  
    11.7649  
   -6.2989  
    5.8894  
   -0.3366  
    0.9583  
    0.8768  
   -2.0384  
    2.3678  
     :
```

```
delta = 14×1
```

```
    0  
    0  
    0  
    0  
    0  
    0  
    0  
    0  
    0  
    0  
     :
```

```
deltaDelta = 14×1
```

```
    0  
    0  
    0  
    0  
    0  
    0  
    0  
    0
```

```
0  
0  
⋮
```

The initial values for the `delta` and `deltaDelta` arrays are always zero. Consider another 40 ms audio segment in the file and extract the cepstral features from this segment.

```
audioSegmentTwo = audioFile(5820:5820+duration-1);  
[coeffsTwo,deltaTwo,deltaDeltaTwo] = cepFeatures(audioSegmentTwo)
```

```
coeffsTwo = 14×1
```

```
3.0899  
-20.4756  
10.4455  
-5.8759  
7.2215  
-1.2027  
-0.0236  
1.9183  
-1.2127  
2.0669  
⋮
```

```
deltaTwo = 14×1
```

```
-0.7382  
-0.9928  
-1.3194  
0.4230  
1.3321  
-0.8661  
-0.9819  
1.0415  
0.8257  
-0.3009  
⋮
```

```
deltaDeltaTwo = 14×1
```

```
-0.7382
```

```

-0.9928
-1.3194
 0.4230
 1.3321
-0.8661
-0.9819
 1.0415
 0.8257
-0.3009
  :
```

Verify that the difference between `coeffsTwo` and `coeffs` vectors equals `deltaTwo`.

```
isequal(coeffsTwo-coeffs,deltaTwo)
```

```
ans = logical
      1
```

Verify that the difference between `deltaTwo` and `delta` vectors equals `deltaDeltaTwo`.

```
isequal(deltaTwo-delta,deltaDeltaTwo)
```

```
ans = logical
      1
```

## Frequency-Domain Voice Activity Detection and Cepstral Feature Extraction

Many feature extraction techniques operate on the frequency domain. Converting an audio signal to the frequency domain only once is efficient. In this example, you convert a streaming audio signal to the frequency domain and feed that signal into a voice activity detector. If speech is present, mel-frequency cepstral coefficients (MFCC) features are extracted from the frequency-domain signal using the `cepstralFeatureExtractor` System object™.

Create a `dsp.AudioFileReader` System object to read from an audio file.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
fs = fileReader.SampleRate;
```

Process the audio in 30 ms frames with a 10 ms hop. Create a default `dsp.AsyncBuffer` object to manage overlap between audio frames.

```
samplesPerFrame = ceil(0.03*fs);
samplesPerHop = ceil(0.01*fs);
samplesPerOverlap = samplesPerFrame - samplesPerHop;

fileReader.SamplesPerFrame = samplesPerHop;
buffer = dsp.AsyncBuffer;
```

Create a `voiceActivityDetector` System object and a `cepstralFeatureExtractor` System object. Specify that they operate in the frequency domain. Create a `dsp.SignalSink` to log the extracted cepstral features.

```
VAD = voiceActivityDetector('InputDomain','Frequency');
cepFeatures = cepstralFeatureExtractor('InputDomain','Frequency','SampleRate',fs,'LogE');
sink = dsp.SignalSink;
```

In an audio stream loop:

- 1 Read one hop's of samples from the audio file and save the samples into the buffer.
- 2 Read a frame from the buffer with specified overlap from the previous frame.
- 3 Call the voice activity detector to get the probability of speech for the frame under analysis.
- 4 If the frame under analysis has a probability of speech greater than 0.75, extract cepstral features and log the features using the signal sink. If the frame under analysis has a probability of speech less than 0.75, write a vector of NaNs to the sink.

```
threshold = 0.75;
nanVector = nan(1,13);
while ~isDone(fileReader)
    audioIn = fileReader();
    write(buffer,audioIn);

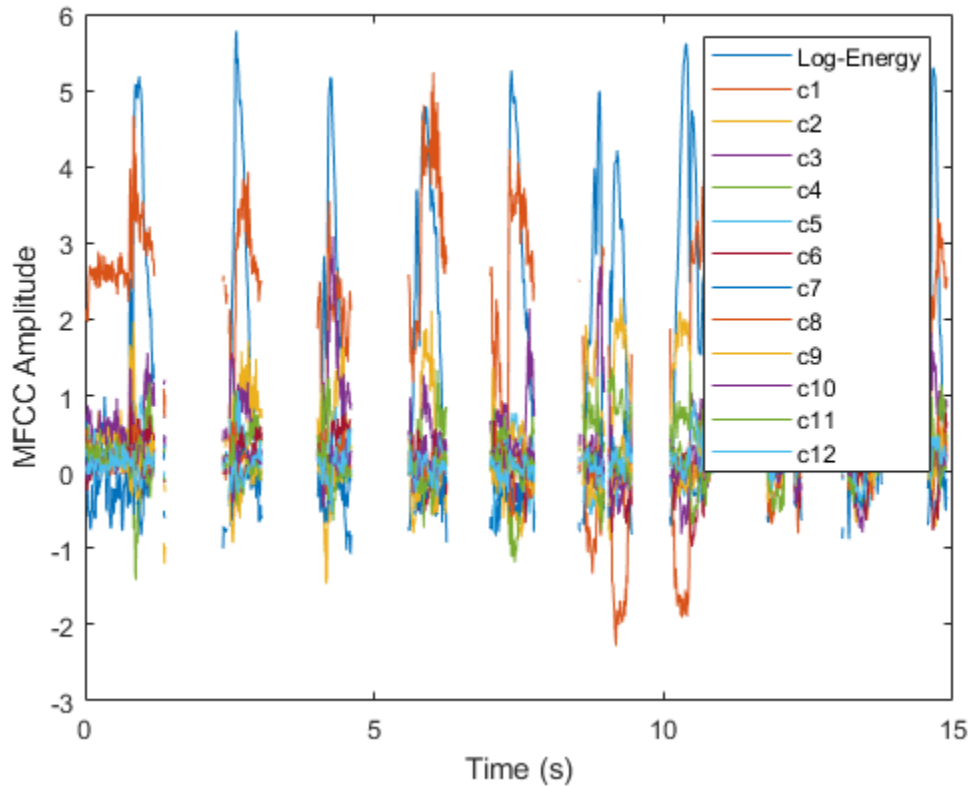
    overlappedAudio = read(buffer,samplesPerFrame,samplesPerOverlap);
    X = fft(overlappedAudio,2048);

    probabilityOfSpeech = VAD(X);
    if probabilityOfSpeech > threshold
        xFeatures = cepFeatures(X);
        sink(xFeatures')
    else
        sink(nanVector)
```

```
end  
end
```

Visualize the cepstral coefficients over time.

```
timeVector = linspace(0,15,size(sink.Buffer,1));  
plot(timeVector,sink.Buffer)  
xlabel('Time (s)')  
ylabel('MFCC Amplitude')  
legend('Log-Energy','c1','c2','c3','c4','c5','c6','c7','c8','c9','c10','c11','c12')
```



#### Extract GTCC from Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create an `audioDeviceWriter` to write the audio to your sound card. Create a `dsp.ArrayPlot` to visualize the GTCC over time.

```
fileReader = dsp.AudioFileReader('RandomOscThree-24-96-stereo-13secs.aif');  
deviceWriter = audioDeviceWriter(fileReader.SampleRate);  
scope = dsp.ArrayPlot;
```

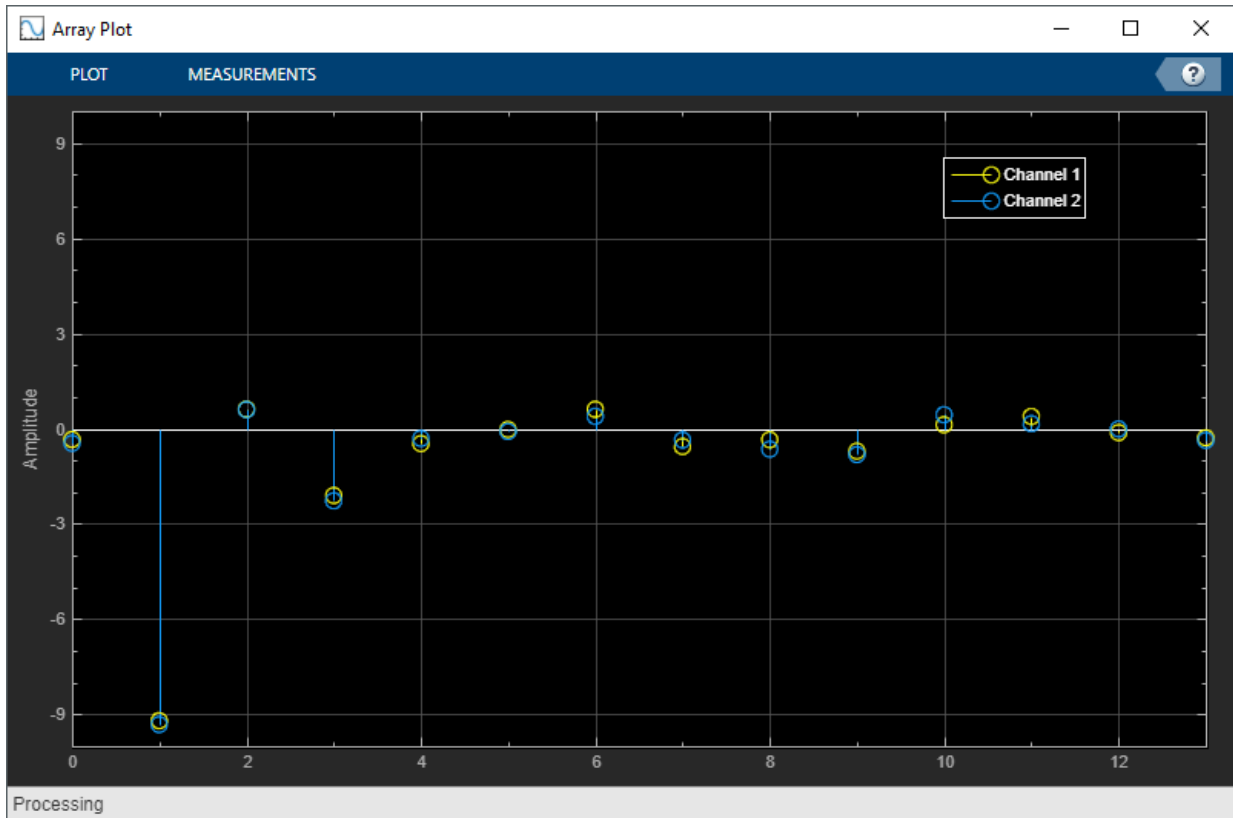
Create a `cepstralFeatureExtractor` that extracts GTCC.

```
cepFeatures = cepstralFeatureExtractor('FilterBank', 'Gammatone', ...  
                                       'SampleRate', fileReader.SampleRate);
```

In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Extract the GTCC from the frame of audio.
- 3 Visualize the GTCC.
- 4 Write the audio frame to your device.

```
while ~isDone(fileReader)  
    audioIn = fileReader();  
    coeffs = cepFeatures(audioIn);  
    scope(coeffs)  
    deviceWriter(audioIn);  
end  
  
release(cepFeatures)  
release(scope)  
release(fileReader)
```

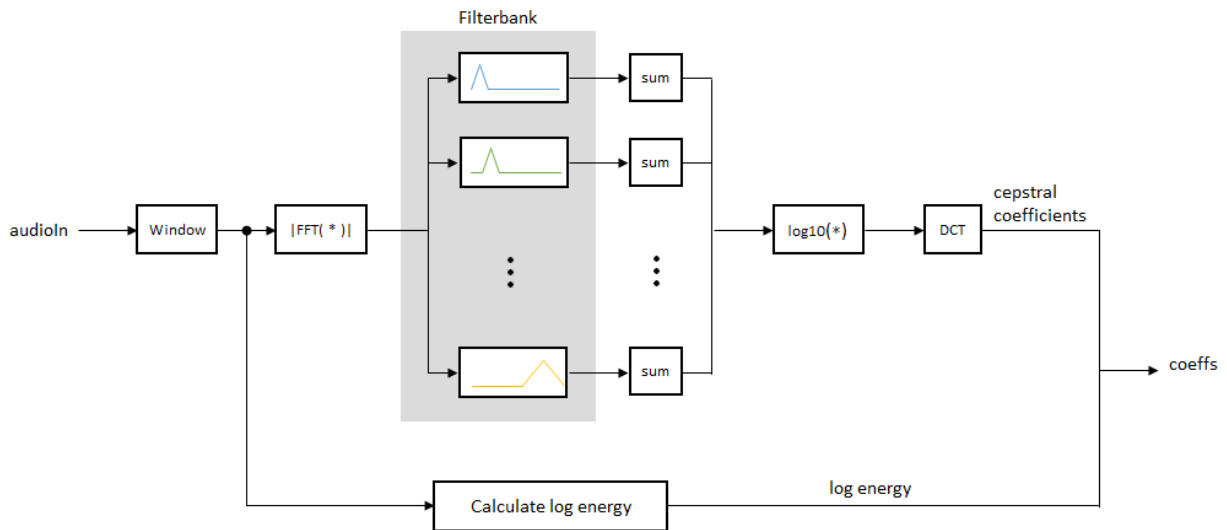


## Algorithms

### Auditory Cepstrum Coefficients

Auditory cepstrum coefficients are popular features extracted from speech signals for use in recognition tasks. In the source-filter model of speech, cepstral coefficients are understood to represent the filter (vocal tract). The vocal tract frequency response is relatively smooth, whereas the source of voiced speech can be modeled as an impulse train. As a result, the vocal tract can be estimated by the spectral envelope of a speech segment.

The motivating idea of cepstral coefficients is to compress information about the vocal tract (smoothed spectrum) into a small number of coefficients based on an understanding of the cochlea. Although there is no hard standard for calculating the coefficients, the basic steps are outlined by the diagram.

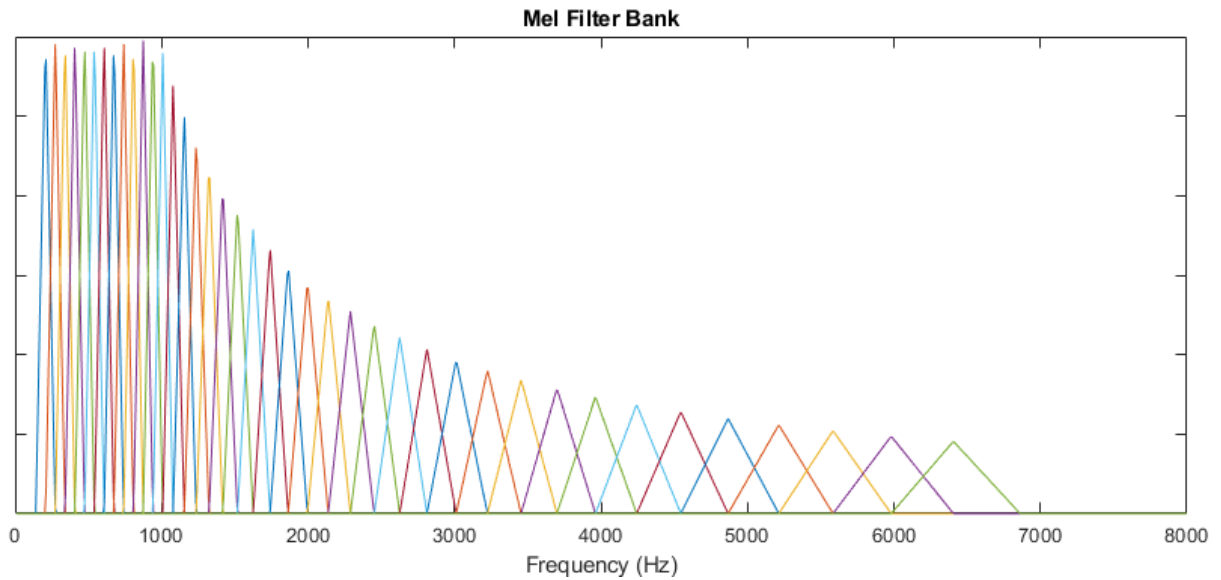


Two popular implementations of the filter bank are the mel filter bank and the gammatone filter bank.

#### Mel Filter Bank

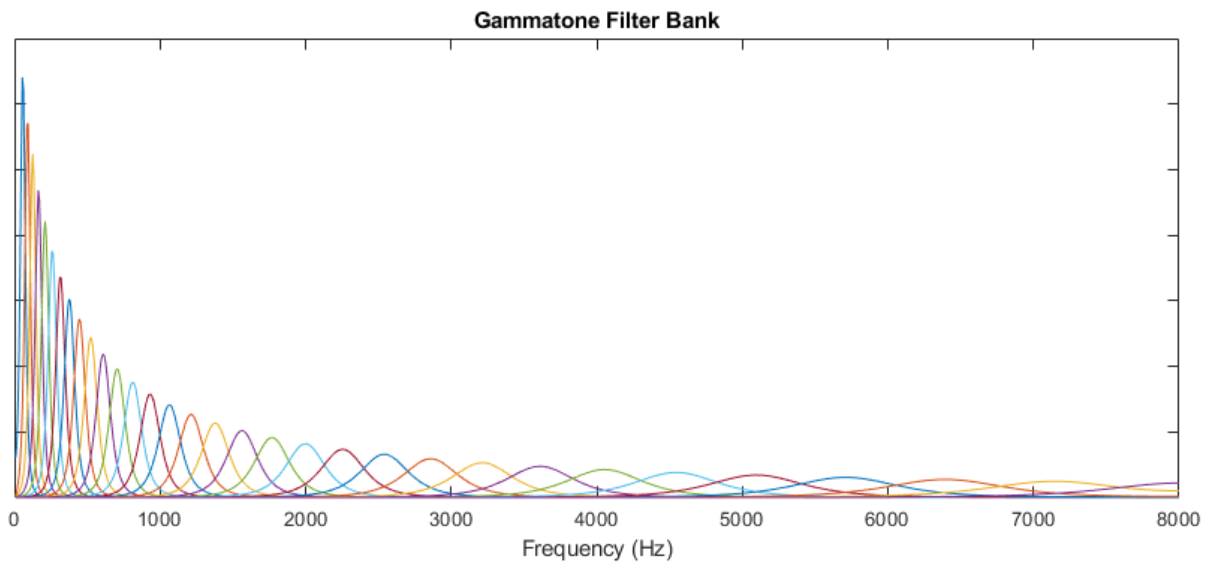
The default mel filter bank linearly spaces the first 10 triangular filters and logarithmically spaces the remaining filters.





### **Gammatone Filter Bank**

The default gammatone filter bank is composed of gammatone filters spaced linearly on the ERB scale between 50 and 8000 Hz. The filter bank is designed by `gammatoneFilterBank`.



## Log Energy

If the input ( $x$ ) is a time-domain signal, the log energy is computed using the following equation:

$$\log E = \log(\text{sum}(x^2))$$

If the input ( $x$ ) is a frequency-domain signal, the log energy is computed using the following equation:

$$\log E = \log(\text{sum}(|x|^2)/\text{FFTLength})$$

## References

- [1] Auditory Toolbox. <https://engineering.purdue.edu/~malcolm/interval/1998-010/AuditoryToolboxTechReport.pdf>
- [2] ETSI ES 201 108 V1.1.3 (2003-09). [https://www.etsi.org/deliver/etsi\\_es/201100\\_201199/201108/01.01.03\\_60/es\\_201108v010103p.pdf](https://www.etsi.org/deliver/etsi_es/201100_201199/201108/01.01.03_60/es_201108v010103p.pdf)

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

Cepstral Feature Extractor | Voice Activity Detector | `gammatoneFilterBank` | `gtcc` | `mfcc` | `pitch` | `voiceActivityDetector`

### Topics

“Speaker Identification Using Pitch and MFCC”

“Classify Gender Using LSTM Networks”

**Introduced in R2018a**

## getFilters

Get auditory filter bank

### Syntax

```
[filterbank, freq] = getFilters(cepFeatures)
```

### Description

`[filterbank, freq] = getFilters(cepFeatures)` returns the filter bank and the corresponding frequency bins in Hz. Each column of the filter bank corresponds to a single bandpass filter. The filterbank is undefined until the object is locked.

### Examples

#### Get Auditory Filter Bank

The auditory filter bank contains a set of bandpass filters that are used to extract the cepstral features from an audio signal. The cepstral features include cepstral coefficients (`coeffs`), the difference between the current and the previous cepstral coefficients (`delta`), and the difference between the current and the previous delta values, `deltaDelta`. The `getFilters` function returns the auditory filter bank and the corresponding frequency bins.

Read an audio signal from 'SpeechDFT-16-8-mono-5secs.wav' file. Extract a 40 ms segment from the audio data. Create a `cepstralFeatureExtractor` System object™ that accepts a time-domain audio input signal sampled at 8 kHz.

```
[audioFile, fs] = audioread('SpeechDFT-16-8-mono-5secs.wav');  
duration = round(0.04*fs); % 40 ms  
audioSegment = audioFile(5500:5500+duration-1);  
cepFeatures = cepstralFeatureExtractor('SampleRate', fs)  
  
cepFeatures =  
    cepstralFeatureExtractor with properties:
```

```
Properties
  FilterBank: 'Mel'
  InputDomain: 'Time'
  NumCoeffs: 13
  FFTLength: []
  LogEnergy: 'Append'
  SampleRate: 8000
```

Show all properties

Pass the 40 ms audio segment as an input to the `cepstralFeatureExtractor` algorithm. The algorithm computes the mel frequency coefficients, log energy, delta, and delta-delta values of the audio segment.

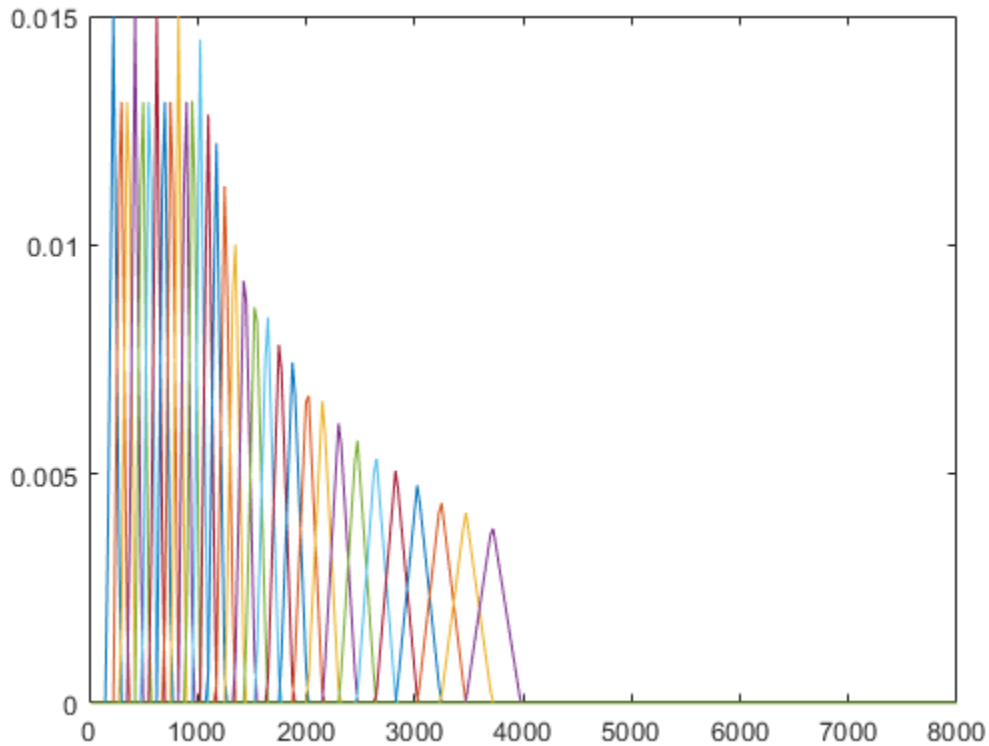
```
[coeffs,delta,deltaDelta] = cepFeatures(audioSegment);
```

Using the `getFilters` function, get the filter bank that computes the cepstral features. Each column in the filter bank contains a bandpass filter. The frequency bins corresponding to the bandpass filters are displayed in `Bins`. Note that the `getFilters` function requires the `cepstralFeatureExtractor` object to be locked.

```
[filterbank, freq] = getFilters(cepFeatures);
```

Plot the filter bank, and you can see that the filter bank columns 33 to 40 have zero elements. These columns correspond to filters whose band edges (one or both) are above the Nyquist frequency, ( $f_s/2$ ). In this example, the Nyquist frequency is 4 kHz.

```
plot(freq,filterbank)
```



```
cepFeatures.BandEdges(33:40)
```

```
ans = 1×8  
103 ×
```

```
3.6915 3.9543 4.2357 4.5371 4.8601 5.2059 5.5765 5.9733
```

```
nnz(filterbank(:,33:40))
```

```
ans = 0
```

Release the cepstral feature extractor object and pass a different audio signal sampled at 22.5 kHz.

```
release(cepFeatures)
[audioFileTwo, fsTwo] = audioread('NoisySpeech-16-22p5-mono-5secs.wav');
duration = round(0.04*fsTwo); % 40 ms
audioSegmentTwo = audioFileTwo(5500:5500+duration-1);
cepFeatures = cepstralFeatureExtractor('SampleRate',fsTwo)
```

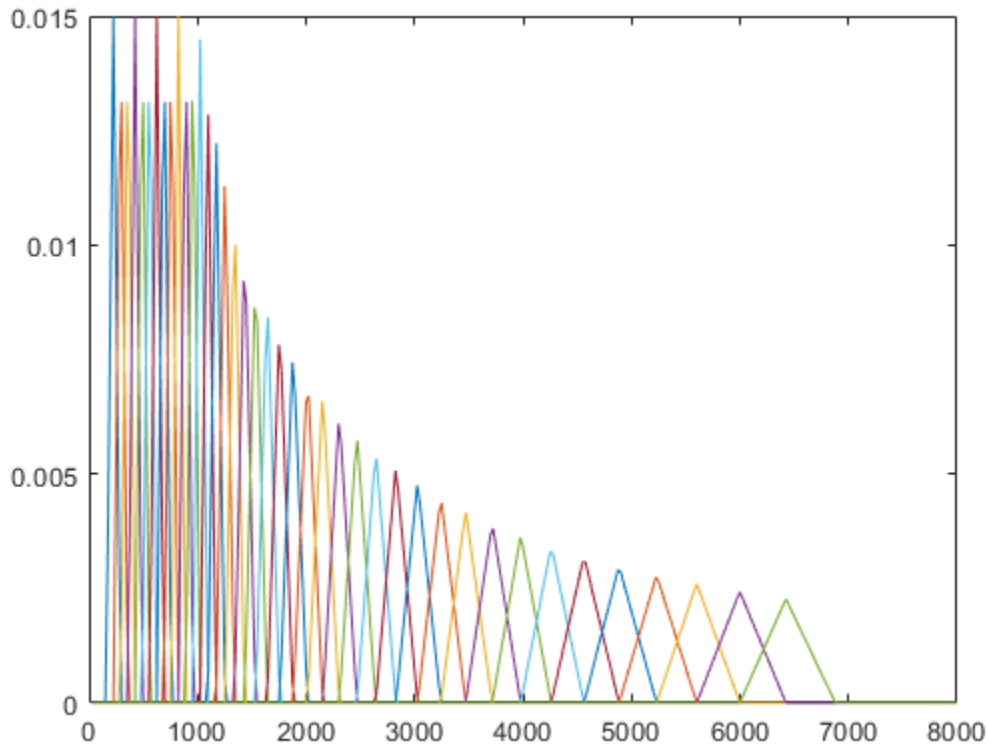
```
cepFeatures =
  cepstralFeatureExtractor with properties:
```

```
  Properties
    FilterBank: 'Mel'
    InputDomain: 'Time'
    NumCoeffs: 13
    FFTLength: []
    LogEnergy: 'Append'
    SampleRate: 22500
```

```
Show all properties
```

The Nyquist frequency is  $22,500/2$ , which is 11,250 Hz. Extract the cepstral features of the second audio segment. Plot the filter bank, which is used to compute the cepstral features. Zoom in on the axis for comparison.

```
[coeffsTwo,deltaTwo,deltaDeltaTwo] = cepFeatures(audioSegmentTwo);
[filterbankTwo, freqTwo] = getFilters(cepFeatures);
plot(freqTwo,filterbankTwo)
axis([0 8000 0 0.015])
```



All the band edges are below the Nyquist frequency, and the bandpass filters in the filter bank have nonzero coefficients.

## Input Arguments

**cepFeatures** — Input cepstral feature extractor System object

`cepstralFeatureExtractor` System object

Input cepstral feature extractor, specified as a `cepstralFeatureExtractor` System object. To use the `getFilters` function, the object must be locked. The filter bank is



defined only when the object is locked. The object is locked when you call the object algorithm.

## Output Arguments

### **filterbank** — Auditory filter bank

matrix

Filter bank used to calculate cepstral features, returned as a matrix. Each column of the matrix corresponds to a single bandpass filter in the filter bank. The number of columns in the matrix is given by  $m - 2$ , where  $m$  is the length of the vector you specify in the `BandEdges` property of the `System` object. The number of rows in the matrix corresponds to the FFT length. By default, the FFT length equals the number of rows in the input signal. You can also specify the FFT length through the `FFTLength` property of the `System` object.

If the Nyquist frequency,  $fs/2$ , is less than the band edge frequencies you specify in the `BandEdges` property, the coefficients of the bandpass filters that fall outside the Nyquist range are set to zero.  $fs$  is the sample rate you specify in the `SampleRate` property of the `System` object.

Data Types: `single` | `double`

### **freq** — Frequency bins corresponding to filter bank (Hz)

row vector

Frequency bins corresponding to the filter bank in Hz, returned as a row vector. The length of the vector equals the FFT length.

Data Types: `single` | `double`

## See Also

`cepstralFeatureExtractor`

**Introduced in R2018a**

## visualize

Visualize static characteristic of dynamic range controller

### Syntax

```
visualize(dynamicRangeController)
visualize(dynamicRangeController,inputRange)
outputLevel = visualize( ___ )
```

### Description

`visualize(dynamicRangeController)` plots the static characteristic of the dynamic range control object. The plot is updated automatically when properties of the object change.

`visualize(dynamicRangeController,inputRange)` enables you to specify the input range.

`outputLevel = visualize( ___ )` returns the dB output level corresponding to the input range. You can use any of the input arguments from previous syntaxes.

---

**Note** This syntax is only available for the `compressor`, `limiter`, and `expander` System objects. It is not available for the `noiseGate` System object.

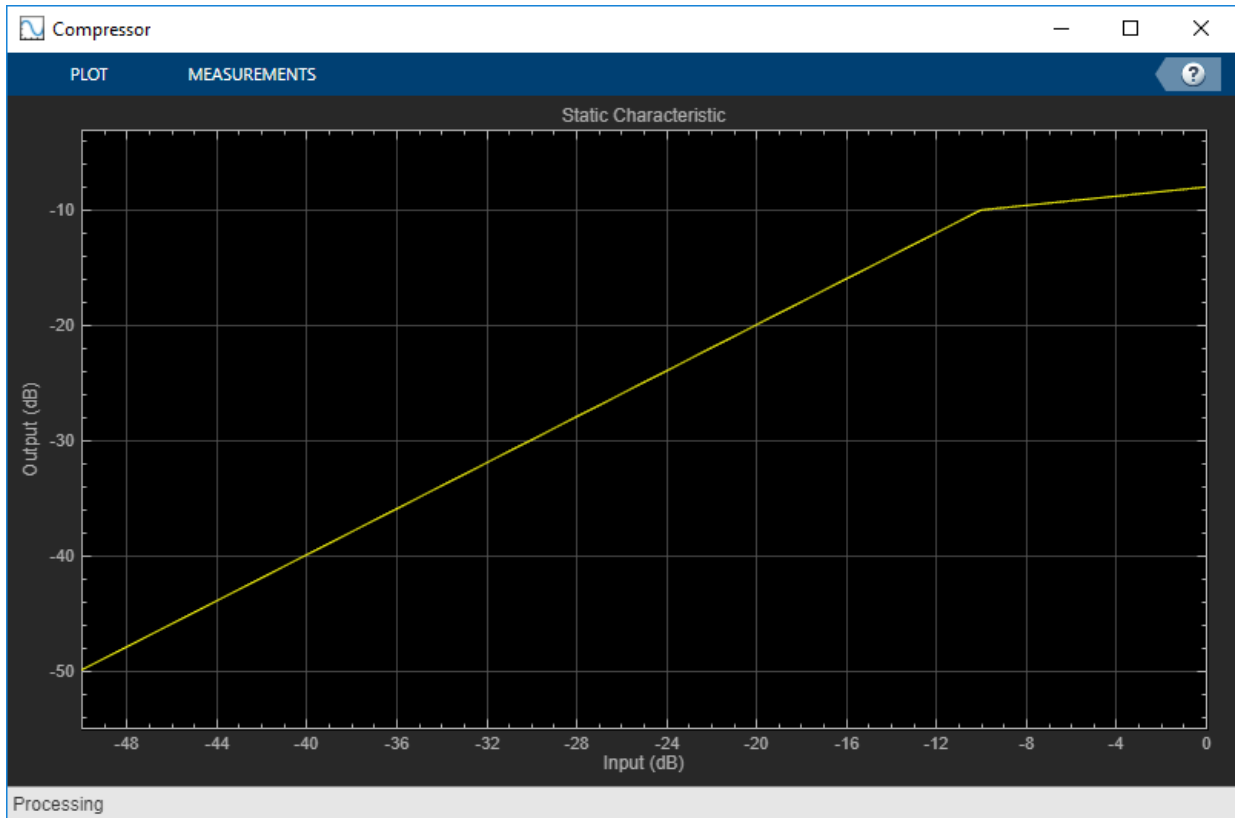
---

## Examples

### Plot Static Characteristic

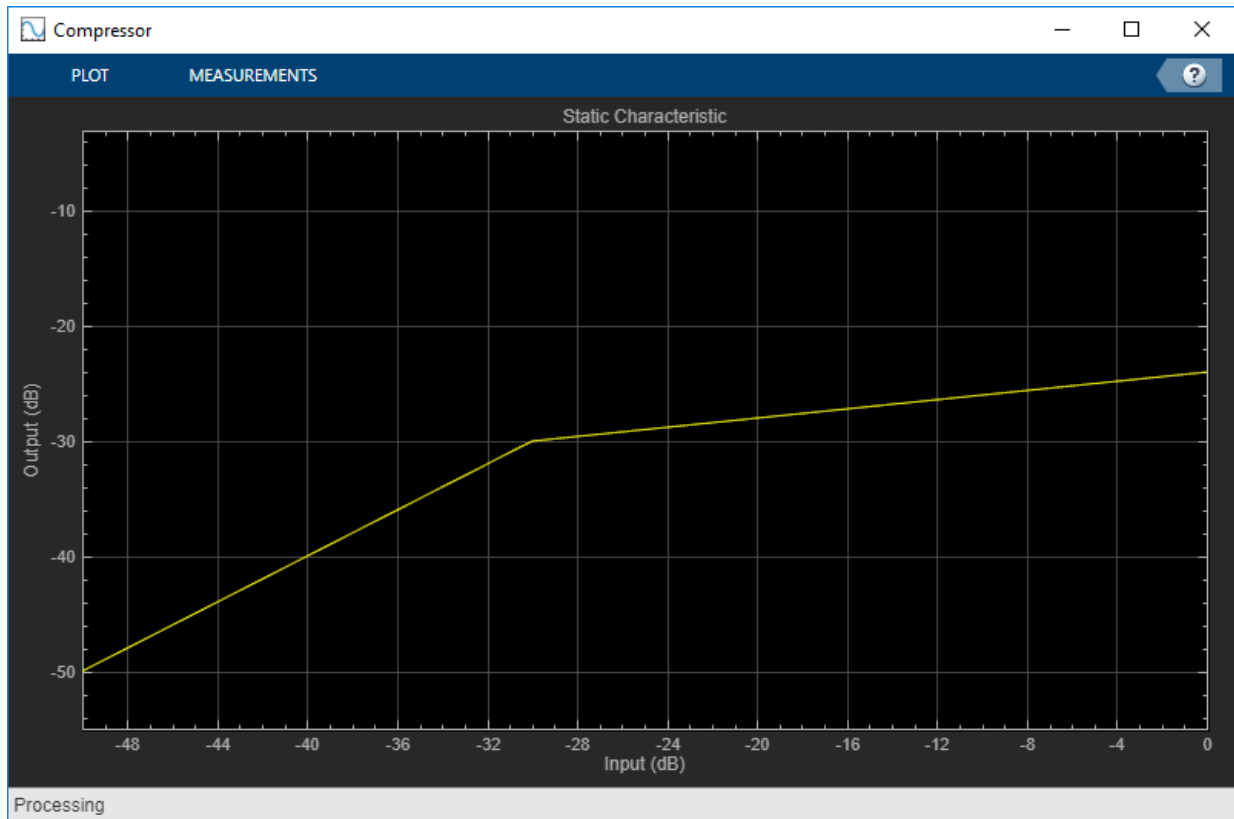
Create an object of the `compressor` System object™, and then plot the static characteristic.

```
dynamicRangeCompressor = compressor;
visualize(dynamicRangeCompressor)
```



The static characteristic plot updates automatically if you modify a property of the object.

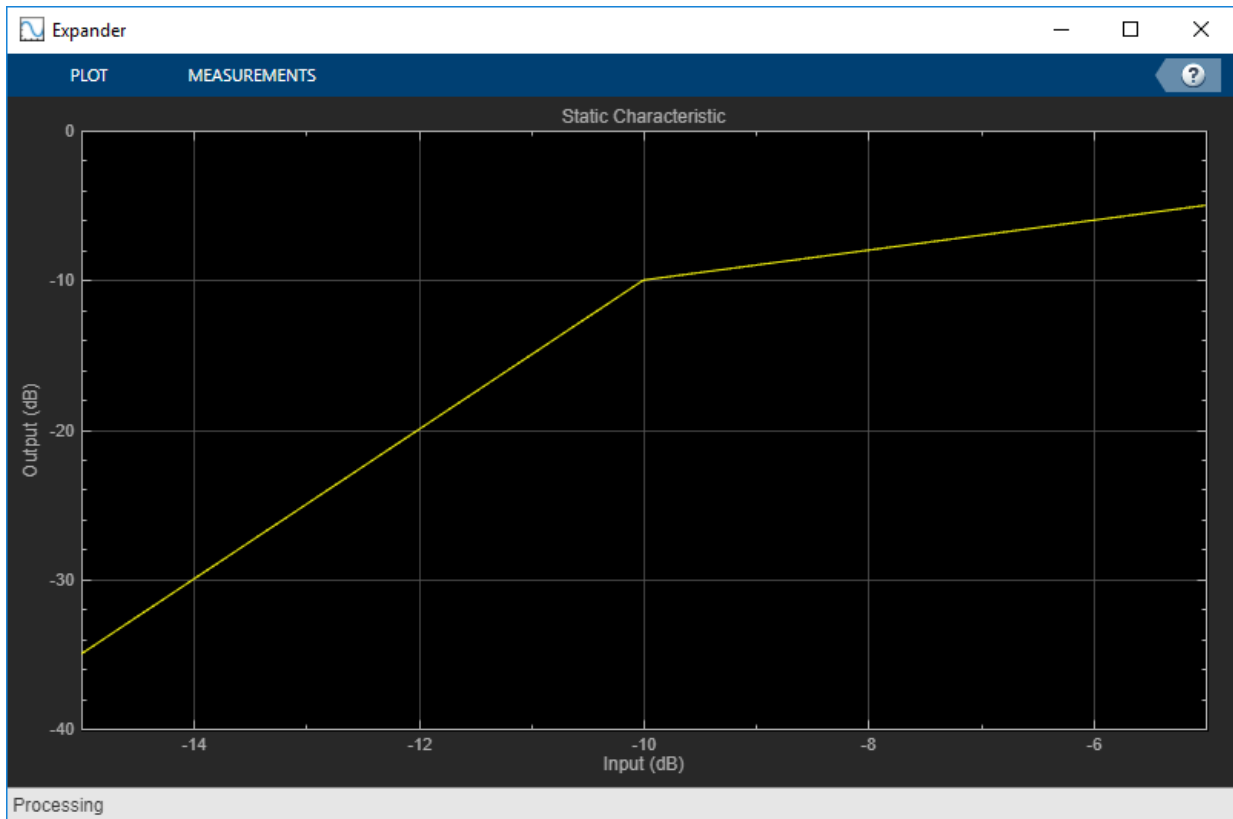
```
dynamicRangeCompressor.Threshold = -30;
```



#### Specify Range of Static Characteristic Plot

Create an object of the `dynamicRangeExpander` System object™. Plot the static characteristic over the range -15 to -5, in 0.001 dB increments.

```
dynamicRangeExpander = dynamicRangeExpander;  
visualize(dynamicRangeExpander, -15:0.001:-5)
```



### Get Output Level From Static Characteristic

Create an object of the `limiter System` object™. Get the output level of the static characteristic over a specified range.

```
dynamicRangeLimiter = limiter;
inputLevel = -15:1:-5
```

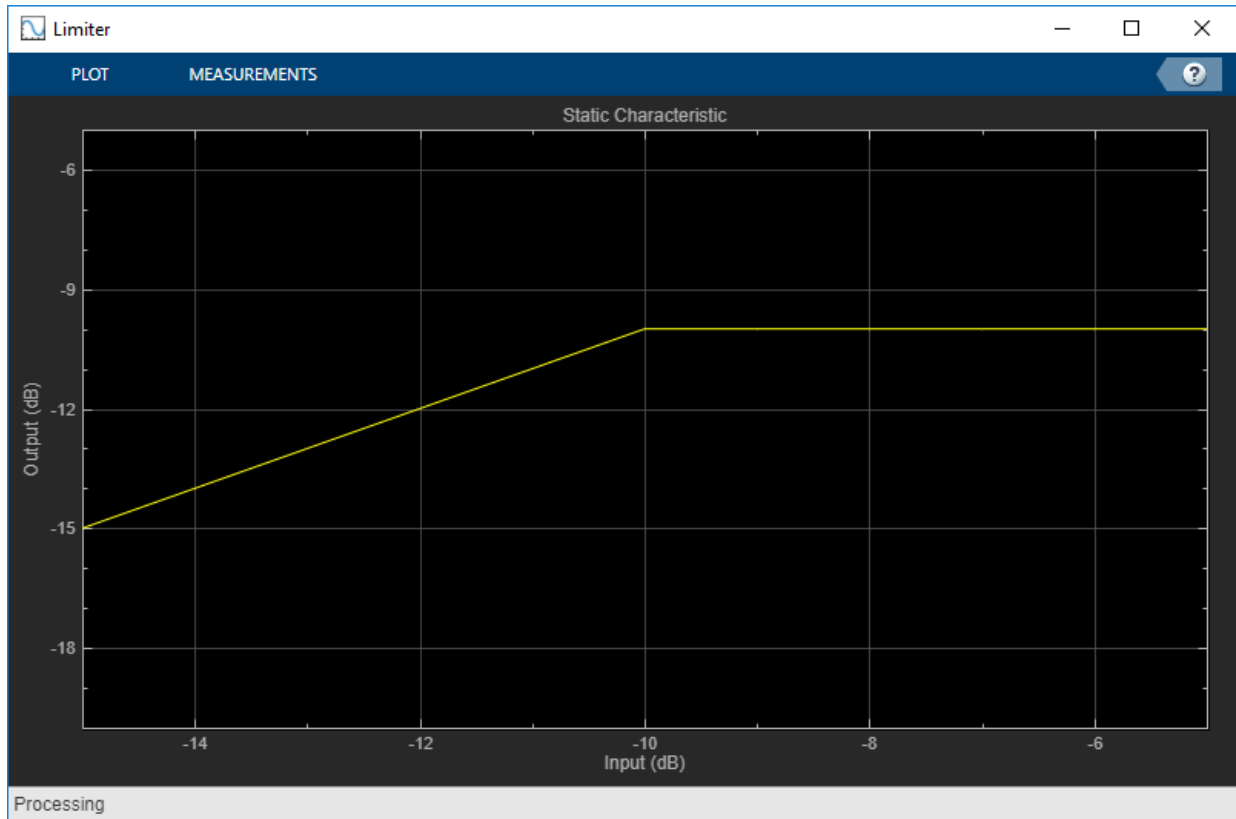
```
inputLevel = 1x11
```

```
-15 -14 -13 -12 -11 -10 -9 -8 -7 -6 -5
```

```
outputLevel = visualize(dynamicRangeLimiter,inputLevel)

outputLevel = 1x11

    -15    -14    -13    -12    -11    -10    -10    -10    -10    -10    -10
```



## Input Arguments

**dynamicRangeController** — Dynamic range control object  
object

Dynamic range control object, specified as an object of compressor, expander, limiter or noiseGate.

**inputRange — Range to calculate static characteristic output**

vector of monotonically increasing values

Range over which to calculate the output of the static characteristic.

The default input range depends on the dynamic range control object:

- compressor -- [-50:0.01:0] dB
- limiter -- [-50:0.01:0] dB
- expander -- [-50:0.01:0] dB
- noiseGate -- [0:0.001:1] linear

## Output Arguments

**outputLevel — Output level (dB)**

vector

Output level in dB, returned as a vector the same size as `inputRange`.

This output is only available for the compressor, limiter, and expander System objects. It is not available for the noiseGate System object.

## See Also

compressor | expander | limiter | noiseGate

## Topics

“Dynamic Range Control”

**Introduced in R2016a**

# createAudioPluginClass

Create audio plugin class that implements functionality of System object

## Syntax

```
createAudioPluginClass(obj)  
createAudioPluginClass(obj,pluginName)
```

## Description

`createAudioPluginClass(obj)` creates a System object plugin that implements the functionality of the Audio Toolbox System object, `obj`. The name of the created class is the System object variable name, `obj`, followed by 'Plugin', for example, `objPlugin`.

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(obj,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(obj,'coolEffect')` creates a System object plugin with class name 'coolEffect'.

## Examples

### Create an Audio Plugin Class From a System Object

Create a compressor object. Call `createAudioPluginClass` to create a System object™ plugin class that implements the functionality of the compressor object.

```
cmpr = compressor;  
createAudioPluginClass(cmpr)
```



## Specify Name of Created Plugin Class

Create an object of the reverberator System object™. Call `createAudioPluginClass` to create a System object™ plugin class that implements the functionality of the reverberator object, specifying the plugin class name as the second argument.

```
reverb = reverberator;  
createAudioPluginClass(reverb, 'Garage');
```

## Input Arguments

### **obj** — System object to create plugin class from

Audio Toolbox System object

System object from which to create a plugin class.

### **pluginName** — Name of created plugin class

character vector

Name of created plugin class, specified as a character vector with fewer than 64 elements.

Data Types: char

## See Also

`audioOscillator` | `compressor` | `crossoverFilter` | `expander` | `graphicEQ` | `limiter` | `multibandParametricEQ` | `noiseGate` | `octaveFilter` | `reverberator` | `wavetableSynthesizer` | `weightingFilter`

## Topics

“Audio Plugins in MATLAB”

“Export a MATLAB Plugin to a DAW”

## Introduced in R2016a

# getFilter

Return biquad filter object with design parameters set

## Syntax

```
biquad = getFilter(obj)
```

## Description

`biquad = getFilter(obj)` returns a `dsp.BiquadFilter` object, `biquad`. The `SOSMatrix` and `ScaleValues` properties of the biquad filter object are set as specified by the `obj` System object.

Use `getFilter` for the design capabilities of the `obj` System object and the processing capabilities of the `dsp.BiquadFilter` System object.

## Examples

### Get Biquad Filter for Octave Filter Design

Create an `octaveFilter` System object™. Call `getFilter` on your object to return a `dsp.BiquadFilter` object with design parameters specified by your `octaveFilter` System object.

```
octFilt = octaveFilter;  
biquad = getFilter(octFilt)
```

```
biquad =  
    FourthOrderSectionFilter with properties:
```

```
    Numerator: [2x5 double]  
    Denominator: [2x5 double]
```

## Get Biquad Filter for Weighting Filter Design

Create a weightingFilter System object™.

```
weightFilt = weightingFilter;
```

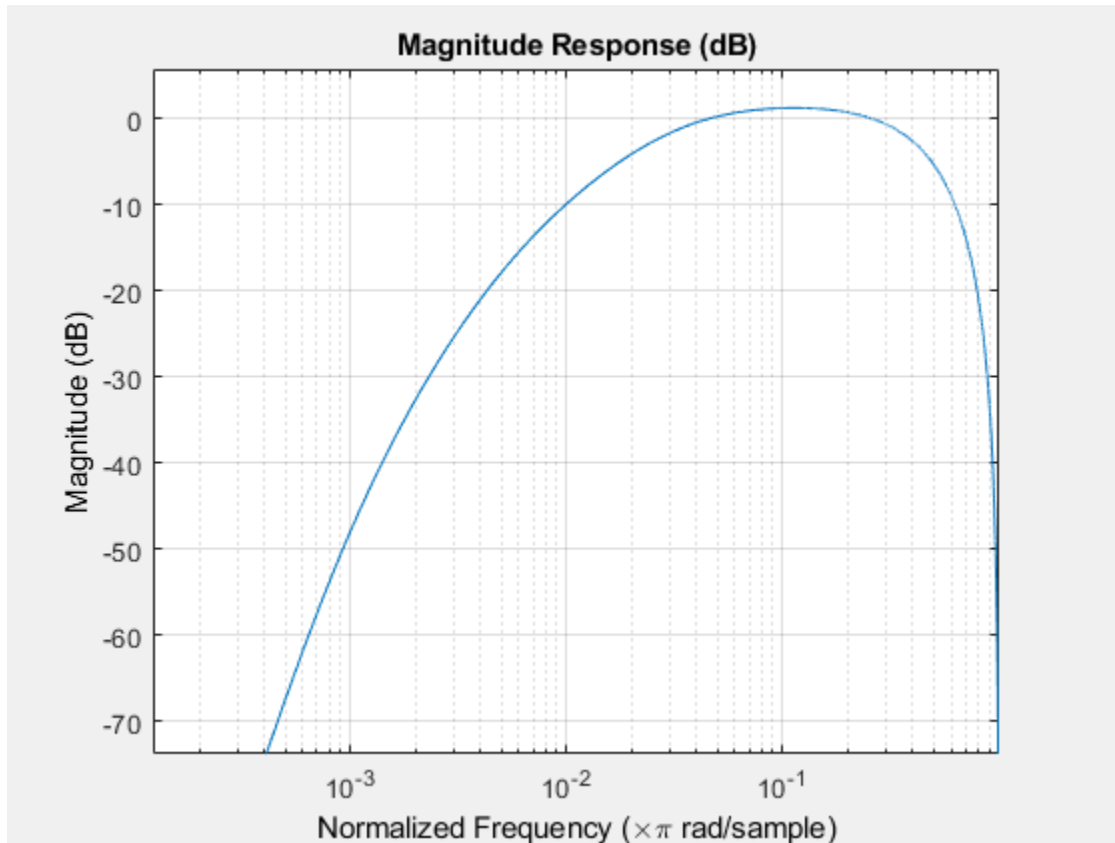
Call `getFilter` on your object to return a `dsp.BiquadFilter` object with design parameters specified by your `weightingFilter` System object. Use `fvtool` to visualize the biquad filter.

```
biquad = getFilter(weightFilt)
```

```
biquad =  
  dsp.BiquadFilter with properties:  
  
          Structure: 'Direct form II transposed'  
    SOSMatrixSource: 'Property'  
          SOSMatrix: [3x6 double]  
        ScaleValues: [4x1 double]  
    InitialConditions: 0  
  OptimizeUnityScaleValues: true
```

```
Show all properties
```

```
fvtool(biquad, 'FrequencyScale', 'log')
```



## Input Arguments

**obj** — System object to get filter from

System object

System object that you want to get a biquad filter object from.

## Output Arguments

**biquad** — Object of `dsp.BiquadFilter`

object

`dsp.BiquadFilter` object.

## See Also

`dsp.BiquadFilter` | `octaveFilter` | `weightingFilter`

## Topics

“Audio Weighting Filters”

“Octave-Band and Fractional Octave-Band Filters”

“Sound Pressure Measurement of Octave Frequency Bands”

**Introduced in R2016b**

# info

Get audio device information

## Syntax

```
infoStruct = info(obj)
```

## Description

`infoStruct = info(obj)` returns a structure, `infoStruct`, containing information about the System object, `obj`.

## Examples

### Get Input Audio Device Information

Create an object of the `audioDeviceReader` System object™ and then call `info` to return a structure containing information about the selected driver, device name, and the maximum number of input channels.

```
deviceReader = audioDeviceReader;  
info(deviceReader)
```

### Get Output Audio Device Information

Create an object of the `audioDeviceWriter` System object™ and then call `info` to return a structure containing information about the selected driver, device name, and the maximum number of output channels.

```
deviceWriter = audioDeviceWriter;  
info(deviceWriter)
```

## Get Audio I/O Device Information

Create an object of the `audioPlayerRecorder` System object™ and then call `info` to return a structure containing information about the selected driver, device name, and the maximum number of input and output channels.

```
playRec = audioPlayerRecorder;  
info(playRec)
```

## Input Arguments

**obj** — System object to get information from

System object

System object to get information from.

## Output Arguments

**infoStruct** — Struct containing object information

struct

Struct containing information about the System object, `obj`. Fields of the struct depend on the System object.

## See Also

`audioDeviceReader` | `audioDeviceWriter` | `audioPlayerRecorder`

**Introduced in R2016a**

## cost

Estimate implementation cost of audio System objects

## Syntax

```
implementationCost = cost(audioObj)
```

## Description

`implementationCost = cost(audioObj)` returns a structure, `implementationCost`, whose fields contain information about the computation cost of implementing the audio System object, `audioObj`.

## Examples

### Estimate Implementation Cost of Crossover Filter

Create a crossover filter with 2 crossovers with 48 dB/octave slopes. Call `cost` to get an estimate of the implementation cost.

```
crossFilt = crossoverFilter('NumCrossovers',2,'CrossoverSlopes',48);  
cost1 = cost(crossFilt)
```

```
cost1 = struct with fields:  
    NumCoefficients: 120  
    NumStates: 48  
    MultiplicationsPerInputSample: 120  
    AdditionsPerInputSample: 97
```

Reduce the crossover slopes for both crossovers to 12 dB/octave. Call `cost` to get an estimate of the new implementation cost.

```
crossFilt.CrossoverSlopes = 12;  
cost2 = cost(crossFilt)
```



```
cost2 = struct with fields:
    NumCoefficients: 36
    NumStates: 12
    MultiplicationsPerInputSample: 36
    AdditionsPerInputSample: 25
```

## Input Arguments

### **audioObj** — Audio System object

crossoverFilter object

Specify the input as a supported audio System object.

Data Types: object

## Output Arguments

### **implementationCost** — Estimate of implementation cost

struct

Estimate of the implementation cost of a filter, returned as struct:

Structure Field	Description
NumCoefficients	Number of filter coefficients (excluding coefficients with values 0, 1 or -1)
NumStates	Number of states
MultiplicationsPerInputSample	Number of multiplication per input sample
AdditionsPerInputSample	Number of additions per input sample

## See Also

crossoverFilter,

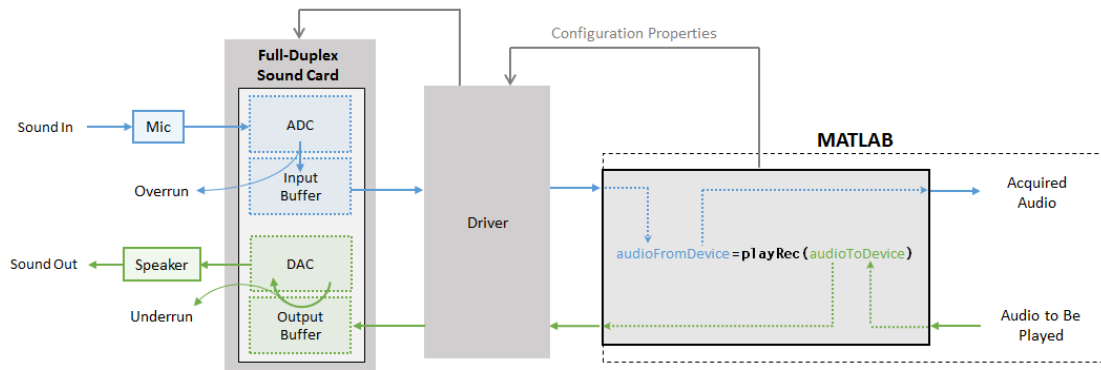
**Introduced in R2016a**

## audioPlayerRecorder

Simultaneously play and record using an audio device

### Description

The `audioPlayerRecorder` System object reads and writes audio samples using your computer's audio device. To use `audioPlayerRecorder`, you must have an audio device and driver capable of simultaneous playback and record.



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

To simultaneously play and record:

- 1 Create the `audioPlayerRecorder` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

## Syntax

```
playRec = audioPlayerRecorder
playRec = audioPlayerRecorder(sampleRateValue)
playRec = audioPlayerRecorder( ____,Name,Value)
```

## Description

`playRec = audioPlayerRecorder` returns a System object, `playRec`, that plays audio samples to an audio device and records samples from the same audio device, in real time.

`playRec = audioPlayerRecorder(sampleRateValue)` sets the `SampleRate` property to `sampleRateValue`.

`playRec = audioPlayerRecorder( ____,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `playRec = audioPlayerRecorder(48000,'BitDepth','8-bit integer')` creates a System object, `playRec`, that operates at a 48 kHz sample rate and an 8-bit integer bit depth.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects \(MATLAB\)](#).

### **Device — Device used to play and record audio data**

default audio device (default) | character vector | string

Device used to play and record audio data, specified as a character vector or string. The object supports only devices enabled for simultaneous playback and recording (full-duplex mode). Use `getAudioDevices` to list available devices.

Supported drivers for `audioPlayerRecorder` are platform-specific:

- Windows -- ASIO
- Mac -- CoreAudio
- Linux -- ALSA

---

**Note** The default audio device is the default device of your machine only if it supports full-duplex mode. If your machine's default audio device does not support full-duplex mode, `audioPlayerRecorder` specifies as the default device the first available device it detects that is capable of full-duplex mode. Use the `info` method to get the device name associated with your `audioPlayerRecorder` object.

---

Data Types: `char` | `string`

**SampleRate — Sample rate used by device to record and play audio data (Hz)**

44100 (default) | positive integer

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

Data Types: `single` | `double`

**BitDepth — Data type used by device**

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

Data type used by device, specified as a character vector or string.

Data Types: `char` | `string`

**SupportVariableSize — Support variable frame size**

false (default) | true

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

- `false` -- If the `audioPlayerRecorder` object is locked, the input must have the same frame size at each call. The buffer size of your audio device is the same as the

input frame size. If you are using the object on Windows, open the ASIO UI to set the sound card buffer to the frame size value.

- `true` -- If the `audioPlayerRecorder` object is locked, the input frame size can change at each call. The buffer size of your audio device is specified through the `BufferSize` property.

To minimize latency, set `SupportVariableSize` to `false`. If variable-size input is required by your audio system, set `SupportVariableSize` to `true`.

Data Types: `logical`

### **BufferSize** — Buffer size of audio device

1024 (default) | positive integer

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

---

**Note** If you are using the object on a Windows machine, use `asioSettings` to set the sound card buffer size to the `BufferSize` value of your `audioPlayerRecorder` System object.

---

### **Dependencies**

To enable this property, set `SupportVariableSize` to `true`.

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

### **PlayerChannelMapping** — Mapping between columns of played data and channels of device

`[]` (default) | scalar | vector

Mapping between columns of played data and channels of output device, specified as a scalar or as a vector of valid channel indices. The default value of this property is `[]`, which means that the default channel mapping is used.

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

### **RecorderChannelMapping** — Mapping between channels of device and columns of recorded data

1 (default) | scalar | vector

Mapping between channels of your audio device and columns of recorded data, specified as a scalar or as a vector of valid channel indices. The default value is 1, which means that the first recording channel on the device is used to acquire data and is mapped to a single-column matrix.

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

## Usage

## Syntax

```
audioFromDevice = playRec(audioToDevice)
[audioFromDevice,numUnderrun] = playRec(audioToDevice)
[audioFromDevice,numUnderrun,numOverrun] = playRec(audioToDevice)
```

## Description

`audioFromDevice = playRec(audioToDevice)` writes one frame of audio samples, `audioToDevice`, to the selected audio device, and returns one frame of audio, `audioFromDevice`.

`[audioFromDevice,numUnderrun] = playRec(audioToDevice)` returns the number of samples overrun since the last call to `playRec`.

`[audioFromDevice,numUnderrun,numOverrun] = playRec(audioToDevice)` returns the number of samples underrun since the last call to `playRec`.

**Note:** When you call the `audioPlayerRecorder` System object, the audio device specified by the `Device` property is locked. An audio device can be locked by only one `audioPlayerRecorder` at a time. To release the audio device, call `release` on the `audioPlayerRecorder` System object.

## Input Arguments

**audioToDevice** — Audio to device  
matrix

Audio signal to write to device, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8`

## Output Arguments

### **audioFromDevice** — Audio from device

matrix

Audio signal read from device, returned as a matrix the same size and data type as `audioToDevice`.

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

### **numUnderrun** — Number of samples underrun

scalar

Number of samples by which the player queue was underrun since the last call to `playRec`. Underrun refers to output signal silence. Output signal silence occurs if the device buffer is empty when it is time for digital-to-analog conversion. This results when the processing loop in MATLAB does not supply samples at the rate the sound card demands.

Data Types: `uint32`

### **numOverrun** — Number of samples overrun

scalar

Number of samples by which the recorder queue was overrun since the last call to `playRec`. Overrun refers to input signal drops. Input signal drops occur when the processing stage does not keep pace with the acquisition of samples.

Data Types: `uint32`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to audioPlayerRecorder

getAudioDevices List available audio devices  
info Get audio device information

## Common to All System Objects

clone Create duplicate System object  
isLocked Determine if System object is in use  
release Release resources and allow changes to System object property values and input characteristics  
reset Reset internal states of System object  
step Run System object algorithm

## Examples

### Synchronize Playback and Recording

Synchronize playback and recording using a single audio device. If synchronization is lost, print information about samples dropped.

Create objects to read from and write to an audio file. Create an `audioPlayerRecorder` object to play an audio signal to your device and simultaneously record audio from your device.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav', ...  
    'SamplesPerFrame',512);  
fs = fileReader.SampleRate;  
  
fileWriter = dsp.AudioFileWriter('Counting-PlaybackRecorded.wav', ...  
    'SampleRate',fs);  
  
aPR = audioPlayerRecorder('SampleRate',fs);
```

In a frame-based loop:

- 1 Read an audio signal from your file.
- 2 Play the audio signal to your device and simultaneously record audio from your device. Use the optional `nUnderruns` and `nOverruns` output arguments to track any loss of synchronization.



3 Write your recorded audio to a file.

Once the loop is completed, release the objects to free devices and resources.

```
while ~isDone(fileReader)
    audioToPlay = fileReader();

    [audioRecorded,nUnderruns,nOVERRUNS] = aPR(audioToPlay);

    fileWriter(audioRecorded)

    if nUnderruns > 0
        fprintf('Audio player queue was underrun by %d samples.\n',nUnderruns);
    end
    if nOVERRUNS > 0
        fprintf('Audio recorder queue was overrun by %d samples.\n',nOVERRUNS);
    end
end
```

```
Audio player queue was underrun by 512 samples.
```

```
release(fileReader)
release(fileWriter)
release(aPR)
```

### Specify Nondefault Channel Mapping

The `audioPlayerRecorder` System object™ enables you to specify a nondefault mapping between the channels of your audio device and the data sent to and received from your audio device. To run this example, your audio device must have at least two channels and be capable of full-duplex mode.

### Using Default Settings

Create an `audioPlayerRecorder` object with default settings. The `audioPlayerRecorder` is automatically configured to a compatible device and driver.

```
aPR = audioPlayerRecorder;
```

The `audioPlayerRecorder` combines reading from your device and writing to your device in a single call: `audioFromDevice = aPR(audioToDevice)`. Calling the `audioPlayerRecorder` with default settings:

- Maps columns of `audioToDevice` to output channels of your device
- Maps input channels of your device to columns of `audioFromDevice`

By default, `audioFromDevice` is a one-column matrix corresponding to channel 1 of your audio device. To view the maximum number of input and output channels of your device, use the `info` method.

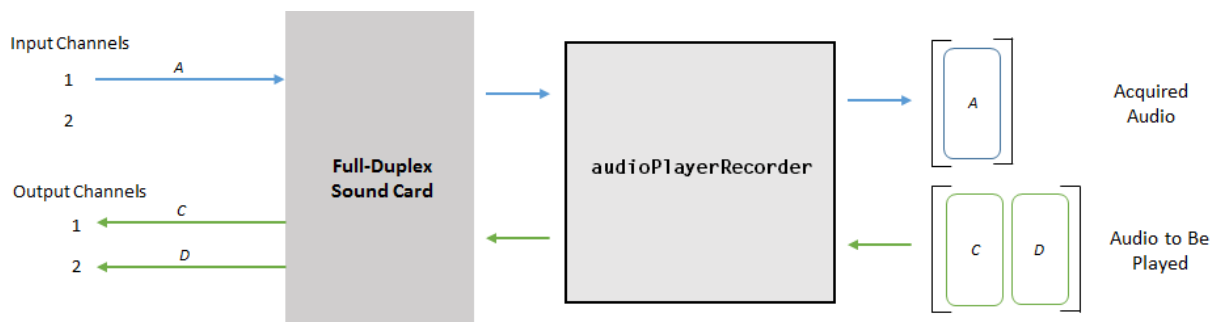
```
aPRInfo = info(aPR);
```

`aPRInfo` is returned as a structure with fields containing information about your selected driver, audio device, and the maximum number of input and output channels in your configuration.

Call the `audioPlayerRecorder` with a two-column matrix. By default, column 1 is mapped to output channel 1, and column 2 is mapped to output channel 2. The `audioPlayerRecorder` returns a one-column matrix with the same number of rows as the `audioToDevice` matrix.

```
highToneGenerator = audioOscillator('Frequency',600,'SamplesPerFrame',256);
lowToneGenerator = audioOscillator('Frequency',200,'SamplesPerFrame',256);
```

```
for i = 1:250
    C = highToneGenerator();
    D = lowToneGenerator();
    audioToDevice = [C,D];
    audioFromDevice = aPR(audioToDevice);
end
```



#### Nondefault Channel Mapping for Audio Output

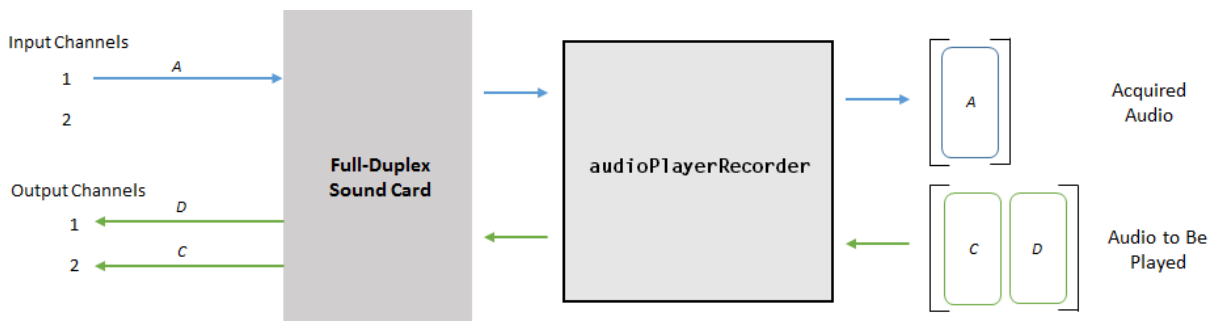
Specify a nondefault channel mapping for your audio output. Specify that column 1 of `audioToDevice` maps to channel 2, and that column 2 of `audioToDevice` maps to

channel 1. To modify the channel mapping, the `audioPlayerRecorder` object must be unlocked.

Run the `audioPlayerRecorder` object. If you are using headphones or stereo speakers, notice that the high frequency and low frequency tones have switched speakers.

```
release(aPR)
aPR.PlayerChannelMapping = [2,1];

for i = 1:250
    C = highToneGenerator();
    D = lowToneGenerator();
    audioToDevice = [C,D];
    audioFromDevice = aPR(audioToDevice);
end
```



### Nondefault Channel Mapping for Audio Input

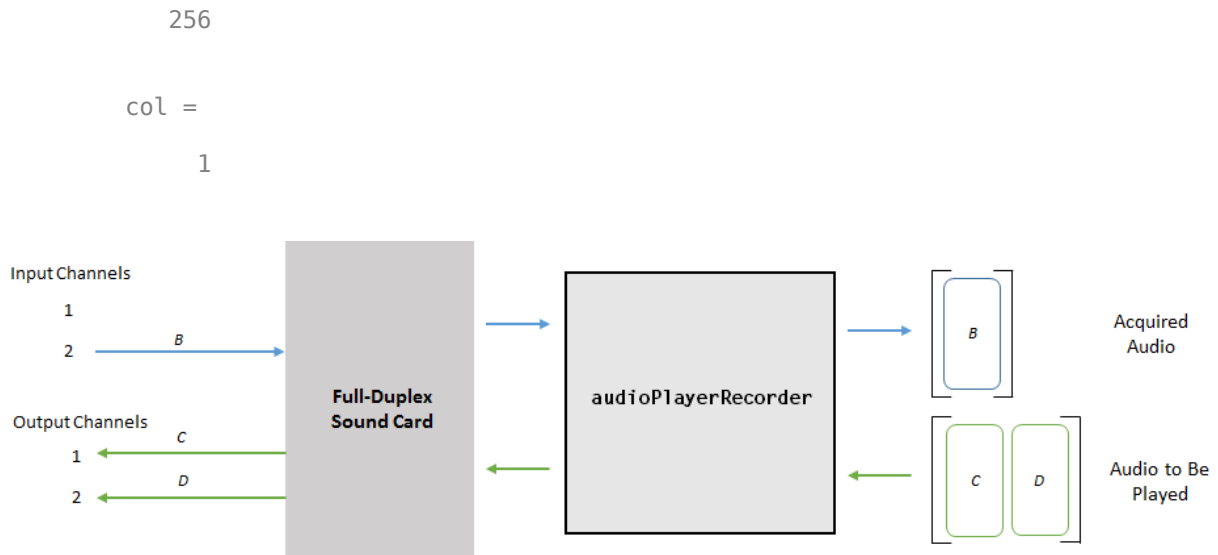
Specify a nondefault channel mapping for your audio input. Record data from only channel two of your device. In this case, channel 2 is mapped to a one-column matrix. Use `size` to verify that `audioFromDevice` is a 256-by-1 matrix.

```
release(aPR)
aPR.RecorderChannelMapping = 2;

audioFromDevice = aPR(audioToDevice);

[rows,col] = size(audioFromDevice)

rows =
```



As a best practice, release your audio device once complete.

```
release(aPR)
```

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGO` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “Run Audio I/O Features Outside MATLAB and Simulink”.

## See Also

### Functions

asioSettings | audioDeviceReader | audioDeviceWriter |  
dsp.AudioFileReader | getAudioDevices

### Blocks

Audio Device Reader | Audio Device Writer

### Topics

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

“Run Audio I/O Features Outside MATLAB and Simulink”

“Real-Time Audio in MATLAB”

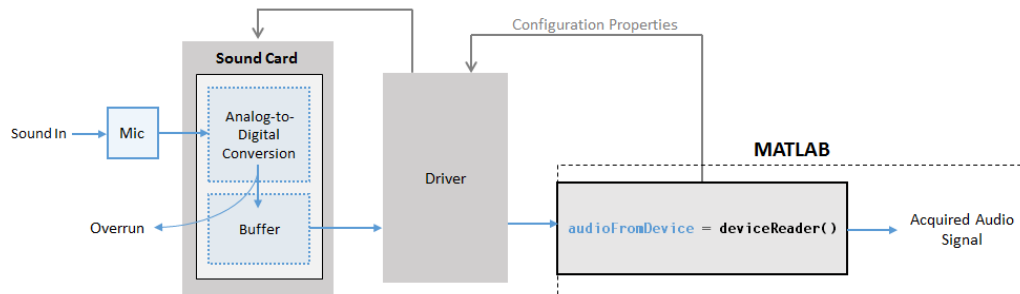
### Introduced in R2017a

# audioDeviceReader

Record from sound card

## Description

The `audioDeviceReader` System object reads audio samples using your computer's audio input device.



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

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

To stream data from an audio device:

- 1 Create the `audioDeviceReader` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

## Syntax

```
deviceReader = audioDeviceReader  
deviceReader = audioDeviceReader(sampleRateValue)  
deviceReader = audioDeviceReader(sampleRateValue, sampPerFrameValue)  
deviceReader = audioDeviceReader( ____, Name, Value)
```

## Description

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

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

`deviceReader = audioDeviceReader(sampleRateValue, sampPerFrameValue)` sets the `SamplesPerFrame` property to `sampPerFrameValue`.

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

Example: `deviceReader = audioDeviceReader(16000, 'BitDepth', '8-bit integer')` creates a System object, `deviceReader`, that operates at a 16 kHz sample rate and an 8-bit integer bit depth.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects \(MATLAB\)](#).

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

'DirectSound' (default) | 'ASIO' | 'WASAPI'

Driver used to access your audio device, specified as 'DirectSound', 'ASIO', or 'WASAPI'.

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

---

**Note** If Driver is specified as 'ASIO', use `asioSettings` to set the sound card buffer size to the `SamplesPerFrame` value of your `audioDeviceReader` System object.

---

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, set `SampleRate` to a sample rate supported by your audio device.

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

Data Types: `char` | `string`

#### **Device — Device used to acquire audio samples**

default audio device (default) | character vector | string

Device used to acquire audio samples, specified as a character vector or string. Use `getAudioDevices` to list available devices for the selected driver.

Data Types: `char` | `string`

#### **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.

#### **Dependencies**

To enable this property, set `ChannelMappingSource` to 'Auto'.

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



**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 by your `audioDeviceReader` object.

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

**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.

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

**BitDepth — Data type used by device to acquire audio data**

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

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

Data Types: `char` | `string`

**ChannelMappingSource — Source of mapping between device channels and 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 your audio device reader is a six-column matrix. Column 1 corresponds to channel 1, column 2 corresponds to channel 2, and so on.
- 'Property' -- The `ChannelMapping` property determines the mapping between channels of your audio device and columns of the output matrix.

Data Types: `char` | `string`

### **ChannelMapping — Nondefault mapping between device channels and 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 “Specify Channel Mapping for `audioDeviceReader`” on page 3-190 for more information.

#### **Dependencies**

To enable this property, set `ChannelMappingSource` to 'Property'.

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

#### **OutputDataType — Data type of the output**

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

Data type of the output, specified as a character vector or 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.

---

Data Types: `char` | `string`

## **Usage**

## **Syntax**

```
audioFromDevice = deviceReader()  
[audioFromDevice,numOverrun] = deviceReader()
```

## **Description**

`audioFromDevice = deviceReader()` returns one frame of audio samples from the selected audio input device.

[audioFromDevice,numOverrun] = deviceReader() returns the number of samples by which the audio reader's queue was overrun since the last call to deviceReader.

**Note:** When you call the 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. To release the audio device, call release on your audioDeviceReader object.

## Output Arguments

### audioFromDevice — Audio from device

matrix

Audio signal read from device, returned as a matrix. The specified number of channels and the SamplesPerFrame property determine the matrix size. The data type of the matrix depends on the OutputDataType property.

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

### numOverrun — Number of samples overrun

scalar

Number of samples by which the audio reader's queue was overrun since the last call to deviceReader.

Data Types: uint32

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named obj, use this syntax:

```
release(obj)
```

### Specific to audioDeviceReader

getAudioDevices	List available audio devices
info	Get audio device information

## Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

## Examples

### Read from Microphone and Write to Audio File

Record 10 seconds of speech with a microphone and send the output to a WAV file.

Create an `audioDeviceReader` 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.')
```

Speak into microphone now.

```
tic  
while toc < 10  
    acquiredAudio = deviceReader();  
    fileWriter(acquiredAudio);  
end  
disp('Recording complete.')
```

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` object to reduce latency.

Create an `audioDeviceReader` object with default settings.

```
deviceReader = audioDeviceReader

deviceReader =
    audioDeviceReader with properties:

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

    Show all properties
```

Calculate the latency due to your device buffer.

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

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

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

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

Latency due to device buffer: 0.001451 seconds.
```

Set the `SampleRate` property of your `audioDeviceReader` System object to 96000. Calculate the latency.

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

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` object to decrease overrun. Your results depend on your computer.

Create an `audioDeviceReader` System object with `SamplesPerFrame` set to 256 and `SampleRate` set to 44100. 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` object. Specify the file name and data 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.')

Speak into microphone now.

tic
while toc < 5
    [input,numOverrun] = deviceReader();
    totalOverrun = totalOverrun + numOverrun;
    fileWriter(input);
end
fprintf('Recording complete.\n')

Recording complete.
```

```
fprintf('Total number of samples overrun: %d.\n',totalOverrun)
```

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

```
fprintf('Total seconds overrun: %d.\n',double(totalOverrun)/double(deviceReader.SamplePerFrame))
```

```
Total seconds overrun: 0.
```

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

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

Use `pause` to 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.')
```

```
Speak into microphone now.
```

```
tic
while toc < 5
    [input,numOverrun] = deviceReader();
    totalOverrun = totalOverrun + numOverrun;
    fileWriter(input);
    pause(0.01)
end
fprintf('Recording complete.\n')
```

```
Recording complete.
```

```
fprintf('Total number of samples overrun: %d.\n',totalOverrun)
```

```
Total number of samples overrun: 97536.
```

```
fprintf('Total seconds overrun: %d.\n',double(totalOverrun)/double(deviceReader.SamplePerFrame))
```

```
Total seconds overrun: 2.211701e+00.
```

Release your `audioDeviceReader` and `dsp.AudioFileWriter` 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.')
```

```
Speak into microphone now.
```

```
tic
while toc < 5
    [input,numOverrun] = deviceReader();
    totalOverrun = totalOverrun + numOverrun;
    fileWriter(input);
    pause(0.01)
end
fprintf('Recording complete.\n')
```

```
Recording complete.
```

```
fprintf('Total number of samples overrun: %d.\n',totalOverrun)
```

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

```
fprintf('Total seconds overrun: %f.\n',totalOverrun/deviceReader.SampleRate)
```

```
Total seconds overrun: 0.000000.
```

### Specify Channel Mapping for `audioDeviceReader`

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

Create an `audioDeviceReader` object with default settings.

```
deviceReader = audioDeviceReader;
```



The default number of channels is 1. Call your `audioDeviceReader` object like a function with no arguments to read one frame of data from your audio device. Verify that the output data matrix has one column.

```
x = 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 = struct with fields:
    Driver: 'DirectSound'
    DeviceName: 'Primary Sound Capture Driver'
    MaximumInputChannels: 2
```

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

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

deviceReader =
  audioDeviceReader with properties:
    Driver: 'DirectSound'
    Device: 'Default'
    SamplesPerFrame: 1024
    SampleRate: 44100

Show all properties
```

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

```
x = 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` object.

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

Now when you call your `audioDeviceReader`:

- 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 your `audioDeviceReader`, input channel 2 of your device maps to an output vector.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGO` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “Run Audio I/O Features Outside MATLAB and Simulink”.

## See Also

### Functions

asioSettings | audioDeviceWriter | audioPlayerRecorder |  
dsp.AudioFileReader | getAudioDevices

### Blocks

Audio Device Reader

### Topics

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

“Run Audio I/O Features Outside MATLAB and Simulink”

“Real-Time Audio in MATLAB”

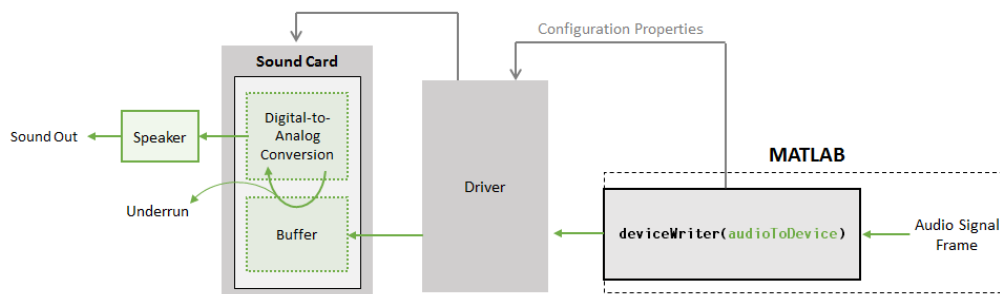
### Introduced in R2016a

## audioDeviceWriter

Play to sound card

### Description

The `audioDeviceWriter` System object writes audio samples to an audio output device. 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.

To stream data to an audio device:

- 1 Create the `audioDeviceWriter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

## Syntax

```
deviceWriter = audioDeviceWriter
deviceWriter = audioDeviceWriter(sampleRateValue)
deviceWriter = audioDeviceWriter( ____, Name, Value)
```

## Description

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

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

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

Example: `deviceWriter = audioDeviceWriter(48000, 'BitDepth', '8-bit integer')` creates a System object, `deviceWriter`, that operates at a 48 kHz sample rate and an 8-bit integer bit depth.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects \(MATLAB\)](#).

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

'DirectSound' (default) | 'ASIO' | 'WASAPI'

Driver used to access your audio device, specified as 'DirectSound', 'ASIO', or 'WASAPI'.

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

---

**Note** If `Driver` is specified as 'ASIO', use `asioSettings` to set the sound card buffer size to the buffer size of your `audioDeviceWriter` System object.

---

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, set `SampleRate` to a sample rate supported by your audio device.

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

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

Data Types: `char` | `string`

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

default audio device (default) | character vector | string scalar

Device used to play audio samples, specified as a character vector or string scalar. Use `getAudioDevices` to list available devices for the selected driver.

Data Types: `char` | `string`

#### **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.

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

#### **BitDepth — Data type used by the device**

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

Data type used by the device, specified as a character vector or string scalar. 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 have an Audio Toolbox licence. If the toolbox is not installed, specifying a nondefault `BitDepth` returns an error.

Data Types: `char` | `string`

### **SupportVariableSizeInput — 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. 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. The buffer size of your audio device is specified through the `BufferSize` property.

Data Types: `char`

### **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.

---

### **Dependencies**

To enable this property, set `SupportVariableSizeInput` to `true`.

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

### **ChannelMappingSource — Source of mapping between input matrix and device channels**

`'Auto'` (default) | `'Property'`

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

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

Data Types: `char` | `string`

### **ChannelMapping — Nondefault mapping between input matrix and device channels**

[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 3-204 example for more information.

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

#### **Dependencies**

To enable this property, set `ChannelMappingSource` to `'Property'`.

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

## **Usage**

## **Syntax**

```
numUnderrun = deviceWriter(audioToDevice)
```



## Description

`numUnderrun = deviceWriter(audioToDevice)` writes one frame of audio samples, `audioToDevice`, to the selected audio device and returns the number of audio samples underrun since the last call to `deviceWriter`.

**Note:** When you call the `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. To release the audio device, call `release` on your `audioDeviceWriter` System object.

## Input Arguments

### **audioToDevice** — Audio to device

matrix

Audio signal to write to device, specified as a matrix. The columns of the matrix are treated as independent audio channels.

If `audioToDevice` 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.

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

## Output Arguments

### **numUnderrun** — Number of samples underrun

scalar

Number of samples by which the audio device writer queue was underrun since the last call to `deviceWriter`.

Data Types: `uint32`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

#### Specific to audioDeviceWriter

```
getAudioDevices List available audio devices  
info            Get audio device information
```

#### Common to All System Objects

```
clone          Create duplicate System object  
isLocked       Determine if System object is in use  
release        Release resources and allow changes to System object property values and  
               input characteristics  
reset          Reset internal states of System object  
step           Run System object algorithm
```

## 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` 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')  
  
fileInfo = struct with fields:  
    Filename: 'B:\matlab\toolbox\dsp\dsp\speech_dft.mp3'  
    CompressionMethod: 'MP3'  
    NumChannels: 1  
    SampleRate: 22050  
    TotalSamples: 112893  
    Duration: 5.1199  
    Title: []  
    Comment: []  
    Artist: []  
    BitRate: 64
```

Create an `audioDeviceWriter` object and specify the sample rate.

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

Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
setup(deviceWriter,zeros(fileReader.SamplesPerFrame,fileInfo.NumChannels))
```

Use the `info` function to obtain the characteristic information about the device writer.

```
info(deviceWriter)
```

```
ans = struct with fields:
    Driver: 'DirectSound'
    DeviceName: 'Primary Sound Driver'
    MaximumOutputChannels: 2
```

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

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

Close the input file and release the device.

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

### Reduce Latency due to Output Device Buffer

*Latency* due to the output device buffer is the time delay of writing one frame of data. 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 %#ok  
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` object to decrease underrun. Your results depend on your computer.

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

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

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

```
totalUnderrun = 0;  
while ~isDone(fileReader)
```

```

        input = fileReader();
        numUnderrun = deviceWriter(input);
        totalUnderrun = totalUnderrun + numUnderrun;
    end
    fprintf('Total samples underrun: %d.\n',totalUnderrun)

```

Total samples underrun: 0.

```

    fprintf('Total seconds underrun: %d.\n',double(totalUnderrun)/double(deviceWriter.Samp

```

Total seconds underrun: 0.

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

```

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

```

Use `pause` to mimic an algorithm that takes 0.075 seconds to process. The pause causes the audio stream loop to go slower than the device, which results in periods of silence in the output audio signal.

```

while ~isDone(fileReader)
    input = fileReader();
    numUnderrun = deviceWriter(input);
    totalUnderrun = totalUnderrun + numUnderrun;
    pause(0.075)
end
fprintf('Total samples underrun: %d.\n',totalUnderrun)

```

Total samples underrun: 71680.

```

fprintf('Total seconds underrun: %d.\n',double(totalUnderrun)/double(deviceWriter.Samp

```

Total seconds underrun: 3.250794e+00.

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

```

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

```

Set the frame size of your audio stream loop to 2048. Because the `SupportVariableSizeInput` property of your `audioDeviceWriter` System object is

set to `false`, the buffer size of your audio device is the same size as the input frame size. Increasing your device buffer size decreases underrun.

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

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

Calculate the total underrun.

```
while ~isDone(fileReader)  
    input = fileReader();  
    numUnderrun = deviceWriter(input);  
    totalUnderrun = totalUnderrun + numUnderrun;  
    pause(0.075)  
end  
fprintf('Total samples underrun: %d.\n',totalUnderrun)
```

Total samples underrun: 0.

```
fprintf('Total seconds underrun: %d.\n',double(totalUnderrun)/double(deviceWriter.SampleRate))
```

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:

- Increasing 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.
- Decreasing the sample rate. Decreasing the sample rate reduces both latency and underrun at the cost of signal resolution.
- Choosing an optimal driver and device for your system.

## Specify Channel Mapping for audioDeviceWriter

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

Create an `audioDeviceWriter` object with default settings.

```
deviceWriter = audioDeviceWriter;
```

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

```
info(deviceWriter)
```

```
ans =
```

```
    struct with fields:
```

```
                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` object to output a tone to your `audioDeviceWriter` object. Your object, `sineGenerator`, returns a vector when called.

```
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 = sineGenerator();  
    deviceWriter(sine);  
    count = count + 1;  
end
```

If your `audioDeviceWriter` 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 1 corresponds to the sine tone, and column 2 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 = sineGenerator();
    static = randn(length(sine),1);
    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` 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 = sineGenerator();
    static = randn(length(sine),1);
    deviceWriter([sine,static]);
```



```
    count = count + 1;  
end
```

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “Run Audio I/O Features Outside MATLAB and Simulink”.

### See Also

[Audio Device Writer](#) | [asiosettings](#) | [audioDeviceReader](#) | [audioPlayerRecorder](#) | [dsp.AudioFileReader](#) | [dsp.AudioFileWriter](#) | [getAudioDevices](#)

### Topics

“Run Audio I/O Features Outside MATLAB and Simulink”

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

“Measure Audio Latency”

“Real-Time Audio in MATLAB”

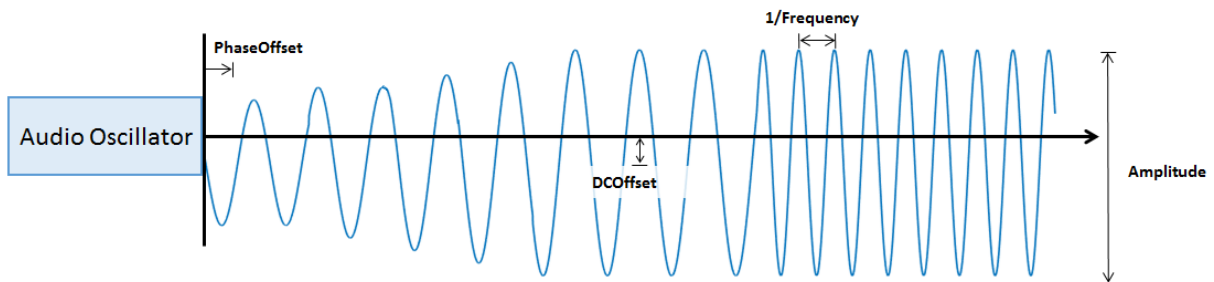
**Introduced in R2016a**

# audioOscillator

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 Create the `audioOscillator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

## Syntax

```
osc = audioOscillator  
osc = audioOscillator(signalTypeValue)
```

```
osc = audioOscillator(signalTypeValue, frequencyValue)
osc = audioOscillator( ____, Name, Value)
```

## Description

`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 8 kHz sinusoids with a DC offset of one.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects \(MATLAB\)](#).

### 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.

**Tunable:** No

Data Types: `char` | `string`

#### **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.

**Tunable:** Yes

Data Types: `single` | `double`

#### **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 `DCOffset` is applied.

**Tunable:** Yes

Data Types: `single` | `double`

#### **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, 1]. The range is a normalized  $2\pi$ -radian 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.

**Tunable:** No

Data Types: single | double

**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.

**Tunable:** Yes

Data Types: single | double

**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.

Individual tones are generated based on values specified by Frequency, Amplitude, PhaseOffset, and DCOffset.

**Tunable:** No

**Dependencies**

To enable this property, set SignalType to 'sine'.

Data Types: single | double

**DutyCycle — Square waveform duty cycle**

0.5 (default) | scalar in the range [0, 1]

Square waveform duty cycle, specified as a scalar in the range [0, 1].

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.

**Tunable:** Yes

**Dependencies**

To enable this property, set `SignalType` to 'square'.

Data Types: `single` | `double`

**Width — Sawtooth width**

1 (default) | scalar in the range [0, 1]

Sawtooth width, specified as a scalar in the range [0, 1].

Sawtooth width determines the point in a sawtooth waveform period at which the maximum occurs.

**Tunable:** Yes

**Dependencies**

To enable this property, set `SignalType` to 'sawtooth'.

Data Types: `single` | `double`

**SamplesPerFrame — Number of samples per frame**

512 (default) | positive integer

Number of samples per frame, specified as a positive integer in the range [1, 192000].

This property determines the vector length that your `audioOscillator` object outputs.

**Tunable:** Yes

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

**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`.

**Tunable:** Yes

Data Types: `single` | `double`

**OutputDataType — Data type of generated waveform**`'double' (default) | 'single'`

Data type of generated waveform, specified as `'double'` or `'single'`.

**Tunable:** Yes

Data Types: `char` | `string`

## Usage

## Syntax

```
waveform = osc()
```

## Description

`waveform = osc()` generates a waveform output, `waveform`. The type of waveform is specified by the algorithm and properties of the `System` object, `osc`.

## Output Arguments

**waveform — Waveform output from oscillator**`column vector`

Waveform output from the audio oscillator, returned as a column vector with length specified by the `SamplesPerFrame` property.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the `System` object as the first input argument. For example, to release system resources of a `System` object named `obj`, use this syntax:

```
release(obj)
```

### Specific to audioOscillator

`createAudioPluginClass` Create audio plugin class that implements functionality of System object  
`parameterTuner` Tune object parameters while streaming

### MIDI

`configureMIDI` Configure MIDI connections between audio object and MIDI controller  
`disconnectMIDI` Disconnect MIDI controls from audio object  
`getMIDIConnections` Get MIDI connections of audio object

### Common to All System Objects

`clone` Create duplicate System object  
`isLocked` Determine if System object is in use  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object  
`step` Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `audioOscillator` System object to user-facing parameters:

Property	Range	Mapping	Units
Frequency	[0.1, 20000]	log	Hz
Amplitude	[0, 10]	linear	no units
DCOffset	[-10, 10]	linear	no units
DutyCycle (available when you set <code>SignalType</code> to 'square')	[0, 1]	linear	no units
Width (available when you set <code>SignalType</code> to 'sawtooth')	[0, 1]	linear	no units



## Examples

### Generate Variable-Frequency Sine Wave

Use the `audioOscillator` to generate a variable-frequency sine wave.

Create an audio oscillator to generate a sine wave. Use the default settings.

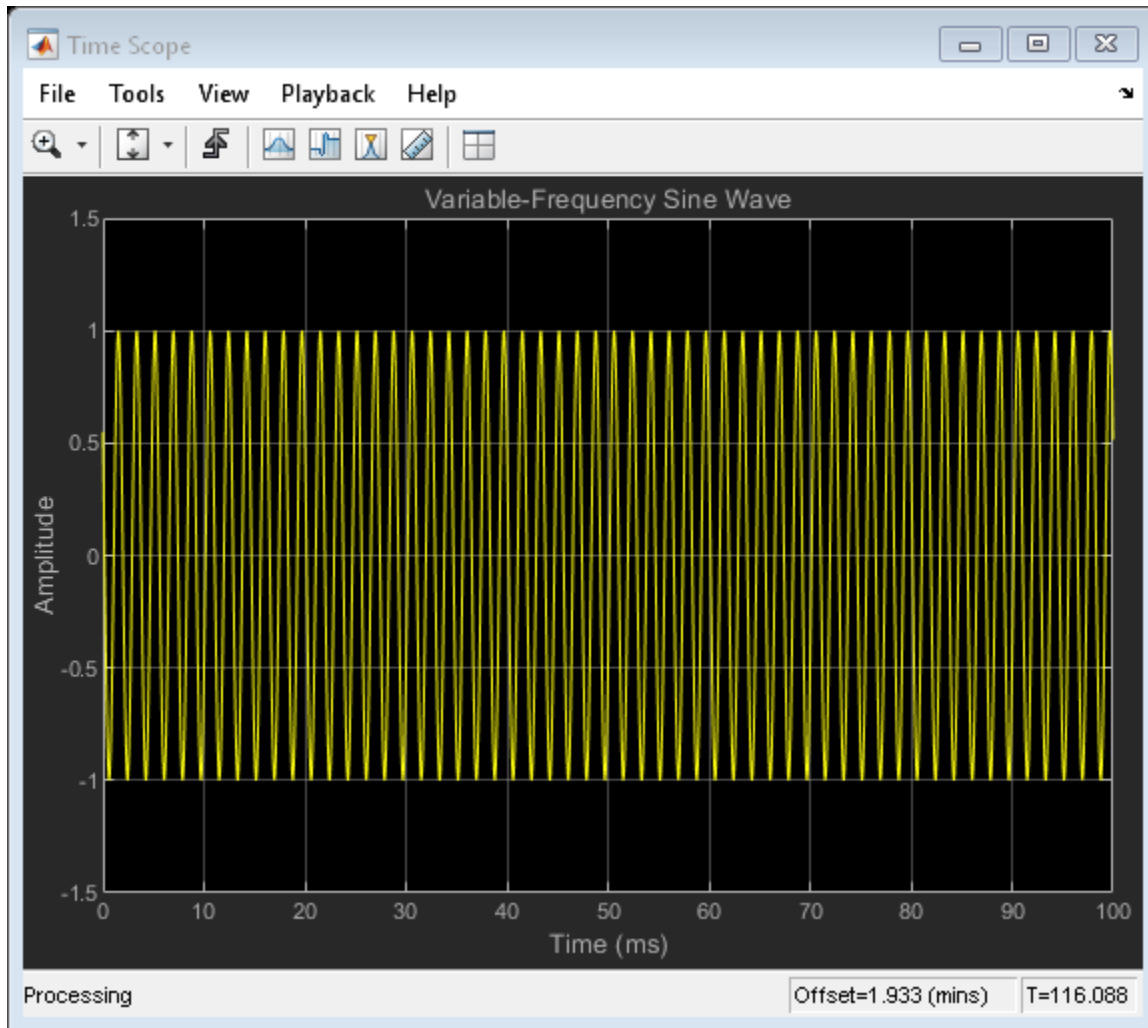
```
osc = audioOscillator;
```

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 sine wave in 50-Hz increments.

```
counter = 0;  
while (counter < 1e4)  
    counter = counter + 1;  
    sineWave = osc();  
    scope(sineWave);  
    if mod(counter,1000)==0  
        osc.Frequency = osc.Frequency + 50;  
    end  
end
```



#### Create a Melody by Tuning Oscillation Frequency

Tune the frequency of an audio oscillator at regularly spaced intervals to create a melody. Play the melody to your audio output device.

Create a structure to hold the frequency values of notes in a melody.

```
notes = struct('C4',261.63,'E4',329.63,'G4sharp',415.30,'A4',440,'B4',493.88, ...
    'C5',523.25,'D5',587.25,'D5sharp',622.25,'E5',659.25,'Silence',0);
```

Create `audioOscillator` and `audioDeviceWriter` objects. Use the default settings.

```
osc = audioOscillator;
aDW = audioDeviceWriter;
```

Create a vector with the initial melody of Fur Elise.

```
melody = [notes.Silence notes.Silence, ...
    notes.E5 notes.D5sharp notes.E5 notes.D5sharp notes.E5 notes.B4 ...
    notes.D5 notes.C5 notes.A4 notes.A4 notes.Silence ...
    notes.C4 notes.E4 notes.A4 notes.B4 notes.B4 notes.Silence ...
    notes.E4 notes.G4sharp notes.B4 notes.C5 notes.C5 notes.Silence];
```

Specify the note duration in seconds. In an audio stream loop, call your audio oscillator and write the sound to your audio device. Update the frequency of the audio oscillator in `noteDuration` time steps to follow the melody. As a best practice, release your objects once complete.

```
noteDuration = 0.3;

i = 1;
tic
while i < numel(melody)
    tone = osc();
    aDW(tone);
    if toc >= noteDuration
        i = i + 1;
        osc.Frequency = melody(i);
        tic
    end
end

release(osc)
release(aDW)
```

### Control Cutoff Frequency of Lowpass Filter

Create a low-frequency oscillator (LFO) lowpass filter, using the `audioOscillator` as a control signal.

Create `dsp.AudioFileReader` and `audioDeviceWriter` System objects to read from an audio file and write to your audio device. Create a biquad filter object to apply lowpass filtering to your audio signal.

```
fileReader = dsp.AudioFileReader('Filename','Engine-16-44p1-stereo-20sec.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
lowpassFilter = dsp.BiquadFilter( ...
    'SOSMatrixSource','Input port', ...
    'ScaleValuesInputPort',false);
```

Create an audio oscillator object. Your audio oscillator controls the cutoff frequency of the lowpass filter in an audio stream loop.

```
osc = audioOscillator('SignalType','sawtooth', ...
    'DCOffset',0.05, ...
    'Amplitude',0.03, ...
    'SamplesPerFrame',fileReader.SamplesPerFrame, ...
    'SampleRate',fileReader.SampleRate, ...
    'Frequency',5);
```

In a loop, filter the audio signal through the lowpass filter. Write the output signal to your audio device.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    ctrlSignal = osc();
    [B,A] = designVarSlopeFilter(48,ctrlSignal(end));
    audioOut = lowpassFilter(audioIn,B,A);
    deviceWriter(audioOut);
end
```

As a best practice, release objects once complete.

```
release(osc)
release(fileReader)
release(deviceWriter)
```

For a more complete implementation of an LFO Filter, see `audiopluginexample.LFOFilter` in the “Audio Plugin Example Gallery”.

## Extended Capabilities

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

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### **See Also**

wavetableSynthesizer

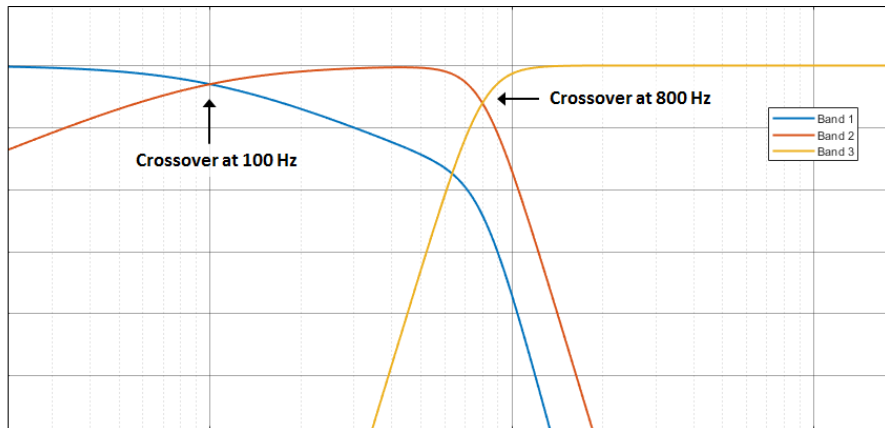
**Introduced in R2016a**

## crossoverFilter

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 Create the `crossoverFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

### Syntax

```
crossFilt = crossoverFilter
crossFilt = crossoverFilter(nCrossovers)
crossFilt = crossoverFilter(nCrossovers,xFrequencies)
crossFilt = crossoverFilter(nCrossovers,xFrequencies,xSlopes)
crossFilt = crossoverFilter(nCrossovers,xFrequencies,xSlopes,Fs)
crossFilt = crossoverFilter( ____,Name,Value)
```

### Description

`crossFilt = crossoverFilter` creates a System object, `crossFilt`, that implements an audio crossover filter.

`crossFilt = crossoverFilter(nCrossovers)` sets the `NumCrossovers` property to `nCrossovers`.

`crossFilt = crossoverFilter(nCrossovers,xFrequencies)` sets the `CrossoverFrequencies` property to `xFrequencies`.

`crossFilt = crossoverFilter(nCrossovers,xFrequencies,xSlopes)` sets the `CrossoverSlopes` property to `xSlopes`.

`crossFilt = crossoverFilter(nCrossovers,xFrequencies,xSlopes,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 two crossovers located at 100 Hz and 800 Hz, and crossover slopes of 6 dB/octave and 48 dB/octave, respectively.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects \(MATLAB\)](#).

### **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 implementing crossover filtering 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

**Tunable:** No

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

### **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.



**Tunable:** Yes

Data Types: `single` | `double`

### **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 specified crossover slope is not inside the range, the slope is rounded to the nearest allowed value.

- If `CrossoverSlopes` is a scalar, all two-band component crossover 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.

**Tunable:** Yes

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

### **SampleRate — Input sample rate (Hz)**

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: `single` | `double`

## **Usage**

## **Syntax**

```
[band1,...,bandN] = crossFilt(audioIn)
```

### Description

`[band1, ..., bandN] = crossFilt(audioIn)` applies a crossover filter on the input, `audioIn`, and returns the filtered output bands, `[band1, ..., bandN]`, where  $N = \text{NumCrossovers} + 1$ .

### Input Arguments

#### **audioIn** — Audio input to crossover filter

matrix

Audio input to the crossover filter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

### Output Arguments

#### **[band1, ..., bandN]** — Audio bands output from crossover filter

set of matrices

Audio bands output from the crossover filter, returned as a set of  $N$  bands. The `NumCrossovers` property determines the number of return arguments:  $N = \text{NumCrossovers} + 1$ . The size of each output argument is the same size as `audioIn`.

Data Types: `single` | `double`

### Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

#### **Specific to crossoverFilter**

`visualize`

Visualize magnitude response of crossover filter

`cost`

Estimate implementation cost of audio System objects

createAudioPluginClass Create audio plugin class that implements functionality of System object  
 parameterTuner Tune object parameters while streaming

## MIDI

configureMIDI Configure MIDI connections between audio object and MIDI controller  
 disconnectMIDI Disconnect MIDI controls from audio object  
 getMIDIConnections Get MIDI connections of audio object

## Common to All System Objects

clone Create duplicate System object  
 isLocked Determine if System object is in use  
 release Release resources and allow changes to System object property values and input characteristics  
 reset Reset internal states of System object  
 step Run System object algorithm

The createAudioPluginClass and configureMIDI functions map tunable properties of the crossoverFilter System object to user-facing parameters:

Property	Range	Mapping	Unit
CrossoverFrequencies	[20, 20000]	linear	Hz
CrossoverSlopes	[6, 48]	linear	dB/octave

## Examples

### Pass Noise Signal Through Crossover Filter

Use the crossoverFilter object to split Gaussian noise into three separate frequency bands.

Create a 5 second noise signal that assumes a 24 kHz sample rate.

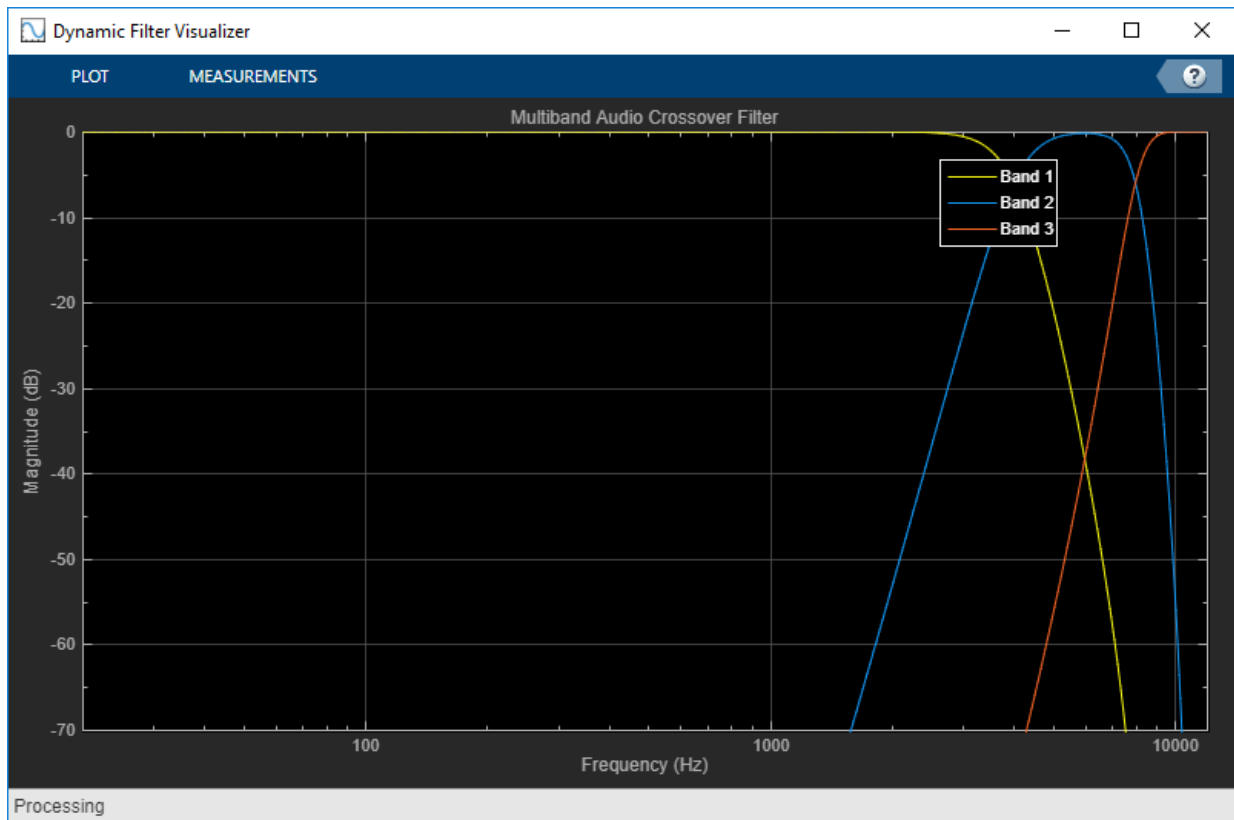
```
fs = 24e3;
noise = randn(fs*5,1);
```

Create a `crossoverFilter` 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',fs);
```

Visualize the magnitude response of your crossover filter object.

```
visualize(crossFilt)
```



Call your crossover filter like a function with the noise signal as the argument.

```
[y1,y2,y3] = crossFilt(noise);
```

Visualize the results using a spectrogram.

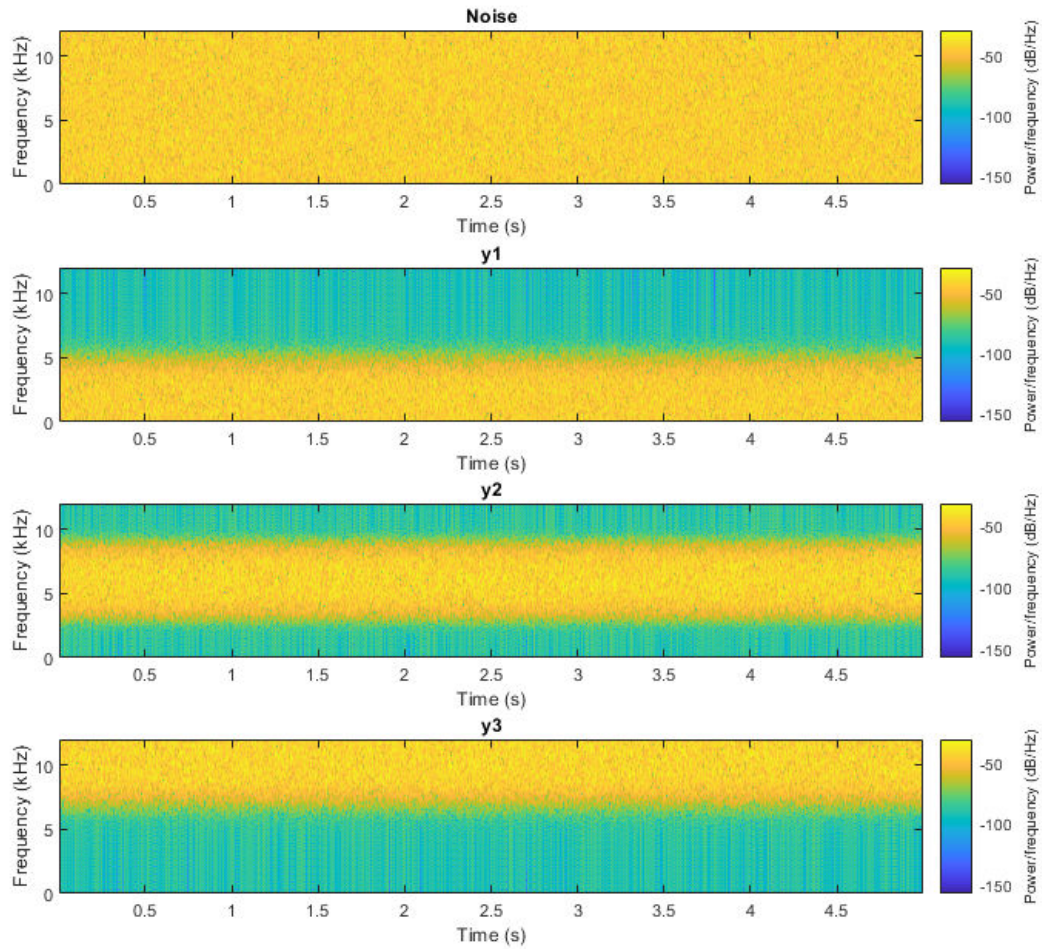
```
figure('Position',[100,100,800,700])

subplot(4,1,1)
spectrogram(noise,120,100,6000,fs,'yaxis')
title('Noise')

subplot(4,1,2)
spectrogram(y1,120,100,6000,fs,'yaxis')
title('y1')

subplot(4,1,3)
spectrogram(y2,120,100,6000,fs,'yaxis')
title('y2')

subplot(4,1,4)
spectrogram(y3,120,100,6000,fs,'yaxis')
title('y3')
```



#### Split Audio Signal into Three Bands

Use the `crossoverFilter` object to split an audio signal into three frequency bands.

Create the `dsp.AudioFileReader` and `audioDeviceWriter` 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))
```

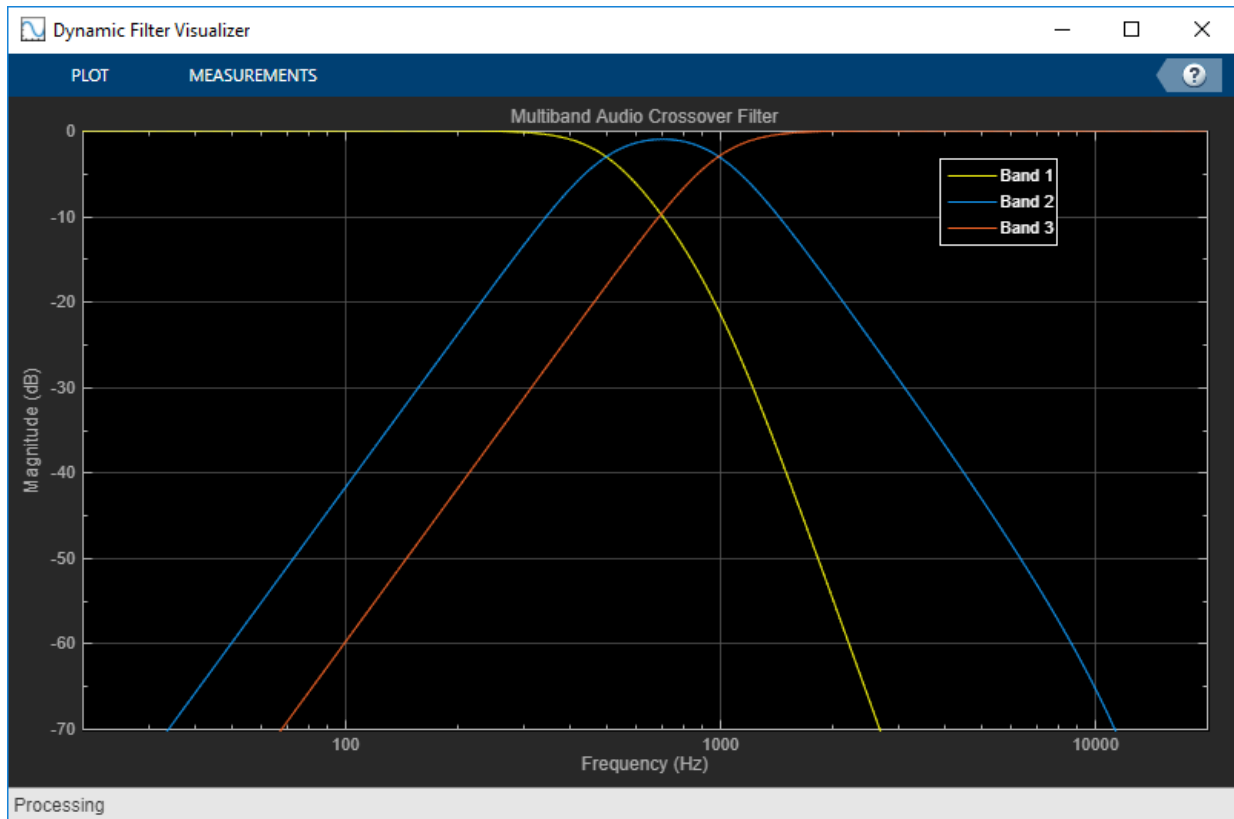
Create a `crossoverFilter` object with 2 crossovers (3 bands), crossover frequencies at 500 Hz and 1 kHz, and a slope of 18 dB/octave. Use the sample rate of the reader as the sample rate of the crossover filter.

```
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 = struct with fields:  
    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'});

```

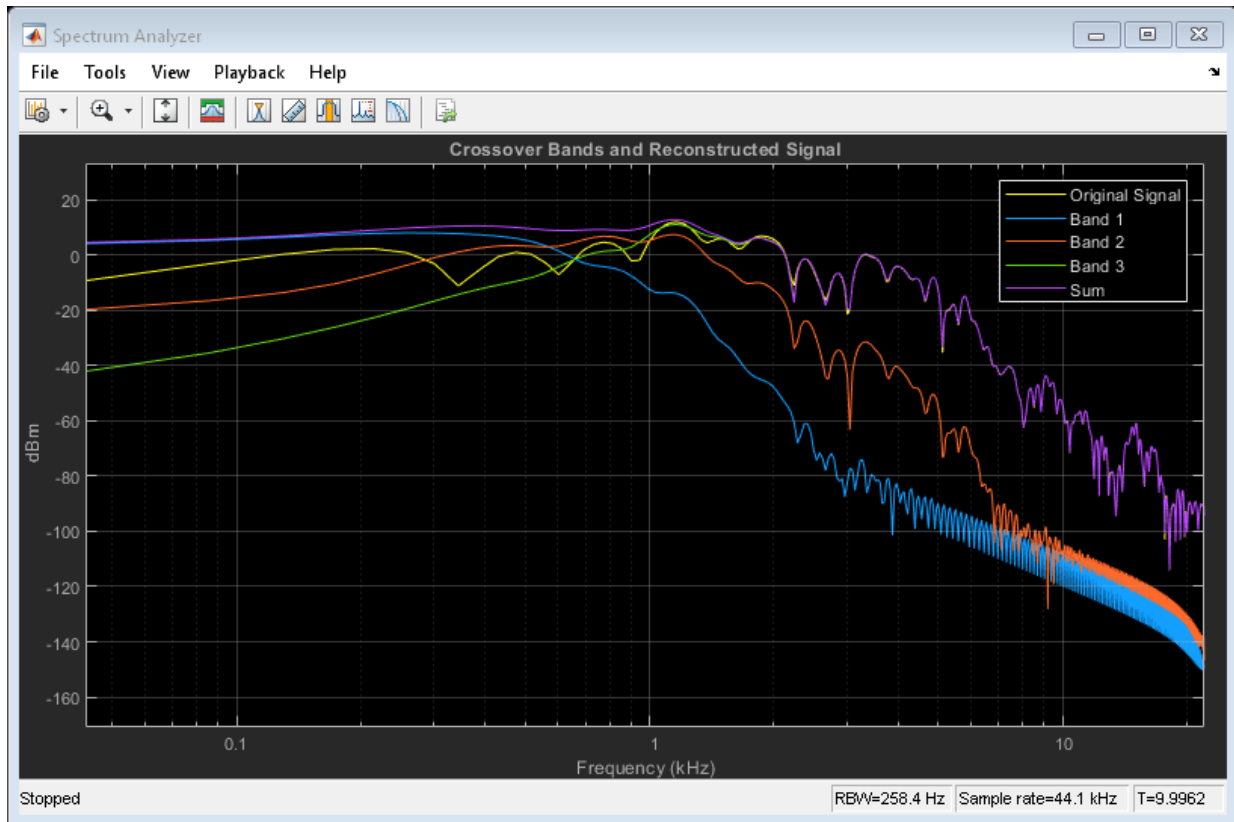
Play 10 seconds of the audio signal. Visualize the spectrum of the original audio, the crossover bands, and the reconstructed signal (sum of bands).

```

setup(scope,ones(samplesPerFrame,5))
count = 0;
while count < (fileReader.SampleRate/samplesPerFrame)*10
    originalSignal = fileReader();
    [band1,band2,band3] = crossFilt(originalSignal);
    sumOfBands = band1 + band2 + band3;
    scope([originalSignal(:,1), ...
           band1(:,1), ...
           band2(:,1), ...
           band3(:,1), ...
           sumOfBands(:,1)])
    deviceWriter(sumOfBands);
    count = count + 1;
end

release(fileReader)
release(crossFilt)
release(scope)

```



```
release(deviceWriter)
```

### Apply Split-Band De-Essing

De-essing is the process of diminishing sibilant sounds in an audio signal. Sibilance refers to the *s*, *z*, and *sh* sounds in speech, which can be disproportionately emphasized during recording. *es* sounds fall under the category of unvoiced speech with all consonants and have a higher frequency than voiced speech. In this example, you apply split-band de-essing to a speech signal by separating the signal into high and low frequencies, applying an expander to diminish the sibilant frequencies, and then remixing the channels.

Create a `dsp.AudioFileReader` object and an `audioDeviceWriter` object to read from a sound file and write to an audio device. Listen to the unprocessed signal. Then release the file reader and device writer.

```
fileReader = dsp.AudioFileReader( ...
    fullfile(matlabroot,'examples','audio','Sibilance.wav'));
deviceWriter = audioDeviceWriter;

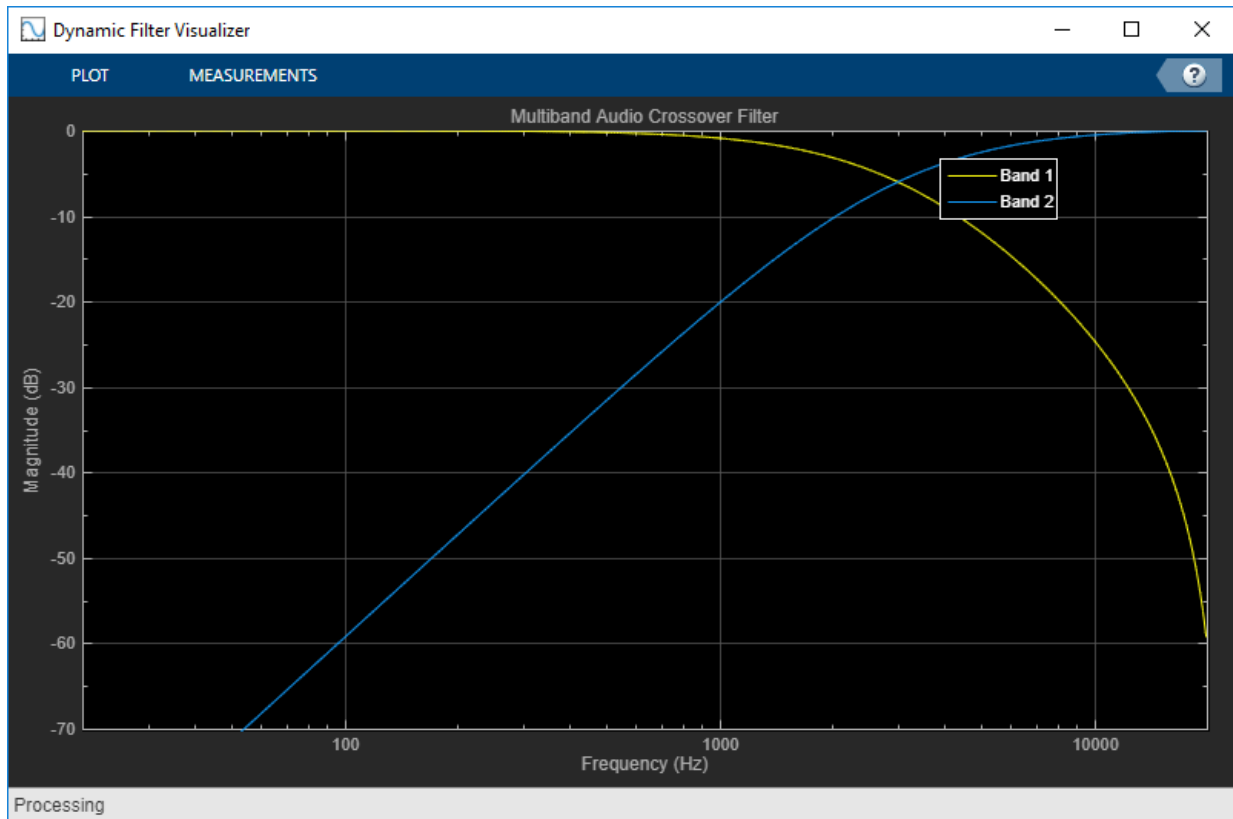
while ~isDone(fileReader)
    audioIn = fileReader();
    deviceWriter(audioIn);
end

release(deviceWriter)
release(fileReader)
```

Create an `expander` System object to de-ess the audio signal. Set the sample rate of the expander to the sample rate of the audio file. Create a two-band crossover filter with a crossover of 3000 Hz. Sibilance is usually found in this range. Set the crossover slope to 12. Plot the frequency response of the crossover filter to confirm your design visually.

```
dRExpander = expander( ...
    'Threshold',-50, ...
    'AttackTime', 0.05, ...
    'ReleaseTime',0.05, ...
    'HoldTime',0.005, ...
    'SampleRate',fileReader.SampleRate);

crossFilt = crossoverFilter( ...
    'NumCrossovers',1, ...
    'CrossoverFrequencies',3000, ...
    'CrossoverSlopes',12);
visualize(crossFilt)
```



Create a `dsp.TimeScope` System object to visualize the original and processed audio signals.

```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',4, ...
    'BufferLength',fileReader.SampleRate*8, ...
    'YLimits',[-1,1], ...
    'ShowGrid',true, ...
    'ShowLegend',true, ...
    'ChannelNames',{'Original','Processed'});
```

In an audio stream loop:

- 1 Read in a frame of the audio file.
- 2 Split the audio signal into two bands.
- 3 Apply dynamic range expansion to the upper band.
- 4 Remix the channels.
- 5 Write the processed audio signal to your audio device for listening.
- 6 Visualize the processed and unprocessed signals on a time scope.

As a best practice, release your objects once done.

```
while ~isDone(fileReader)
    audioIn = fileReader();

    [band1,band2] = crossFilt(audioIn);

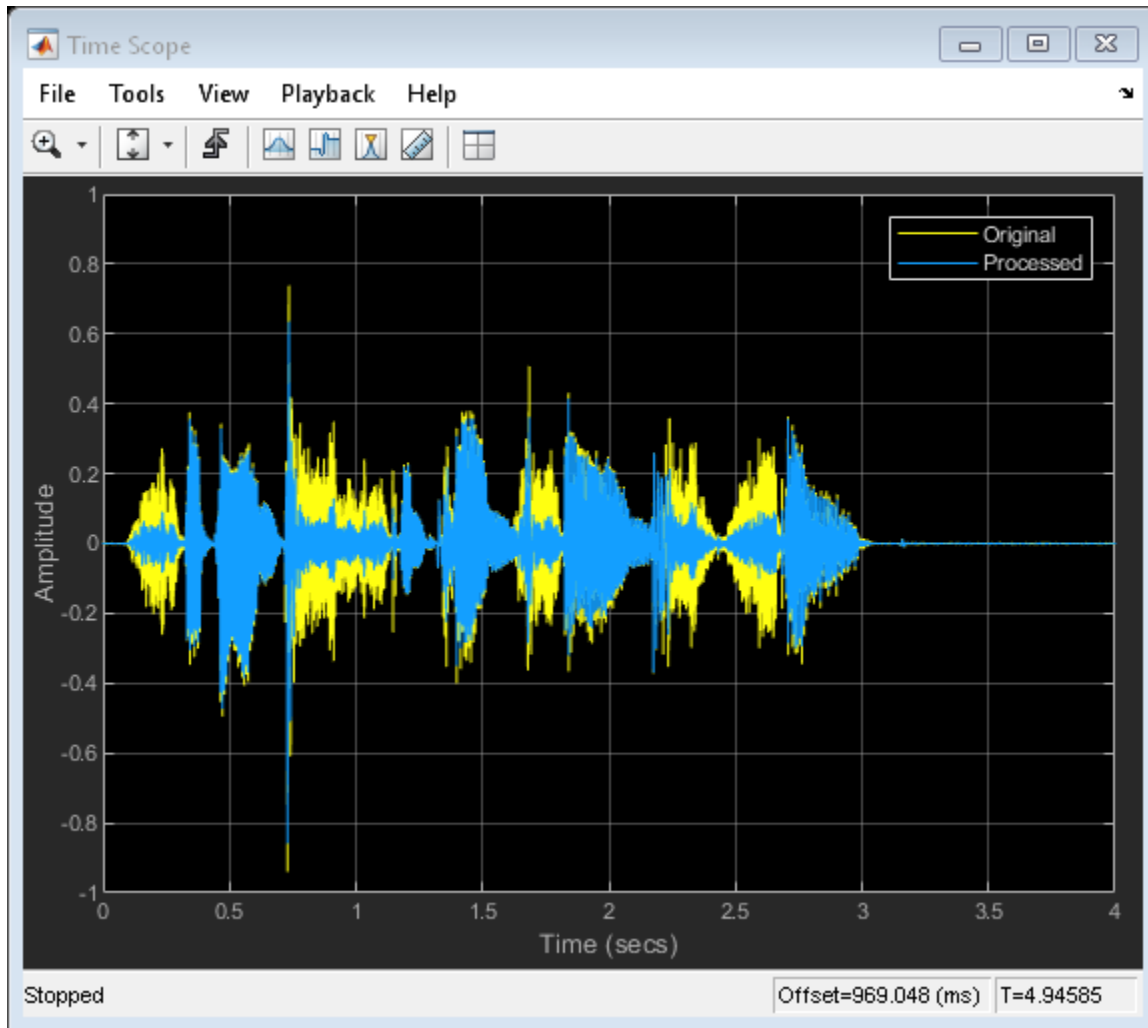
    band2processed = dRExpander(band2);

    procAudio = band1 + band2processed;

    deviceWriter(procAudio);

    scope([audioIn procAudio]);
end

release(deviceWriter)
release(fileReader)
release(scope)
```



```
release(crossFilt)  
release(dRExpander)
```

## Diminish Plosives from Speech Signal

Plosives are consonant sounds resulting from a sudden release of airflow. They are most pronounced in words beginning with *p*, *d*, and *g* sounds. Plosives can be emphasized by the recording process and are often displeasurable to hear. In this example, you minimize the plosives of a speech signal by applying highpass filtering and low-band compression.

Create a `dsp.AudioFileReader` object and a `audioDeviceWriter` object to read an audio signal from a file and write an audio signal to a device. Play the unprocessed signal. Then release the file reader and device writer.

```
fileReader = dsp.AudioFileReader('audioPlosives.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

while ~isDone(fileReader)
    audioIn = fileReader();
    deviceWriter(audioIn);
end
release(deviceWriter)
release(fileReader)
```

Design a highpass filter with a steep rolloff of all frequencies below 120 Hz. Use a `dsp.BiquadFilter` object to implement the highpass filter design. Create a crossover filter with one crossover at 250 Hz. The crossover filter enables you to separate the band of interest for processing. Create a dynamic range compressor to compress the dynamic range of plosive sounds. To apply no make-up gain, set the `MakeUpGainMode` to "Property" and use the default 0 dB `MakeUpGain` property value. Create a time scope to visualize the processed and unprocessed audio signal.

```
[B,A] = designVarSlopeFilter(48,120/(fileReader.SampleRate/2),"hi");
biquadFilter = dsp.BiquadFilter( ...
    "SOSMatrixSource","Input port", ...
    "ScaleValuesInputPort",false);

crossFilt = crossoverFilter( ...
    "SampleRate",fileReader.SampleRate, ...
    "NumCrossovers",1, ...
    "CrossoverFrequencies",250, ...
    "CrossoverSlopes",48);

dRCompressor = compressor( ...
    "Threshold",-35, ...
    "Ratio",10, ...
    "KneeWidth",20, ...
```

```
"AttackTime",1e-4, ...
"ReleaseTime",3e-1, ...
"MakeUpGainMode","Property", ...
"SampleRate",fileReader.SampleRate);

scope = dsp.TimeScope( ...
    "SampleRate",fileReader.SampleRate, ...
    "TimeSpan",3, ...
    "BufferLength",fileReader.SampleRate*3*2, ...
    "YLimits",[-1 1], ...
    "ShowGrid",true, ...
    "ShowLegend",true, ...
    "ChannelNames",{ 'Original', 'Processed' });
```

In an audio stream loop:

- 1 Read in a frame of the audio file.
- 2 Apply highpass filtering using your biquad filter.
- 3 Split the audio signal into two bands.
- 4 Apply dynamic range compression to the lower band.
- 5 Remix the channels.
- 6 Write the processed audio signal to your audio device for listening.
- 7 Visualize the processed and unprocessed signals on a time scope.

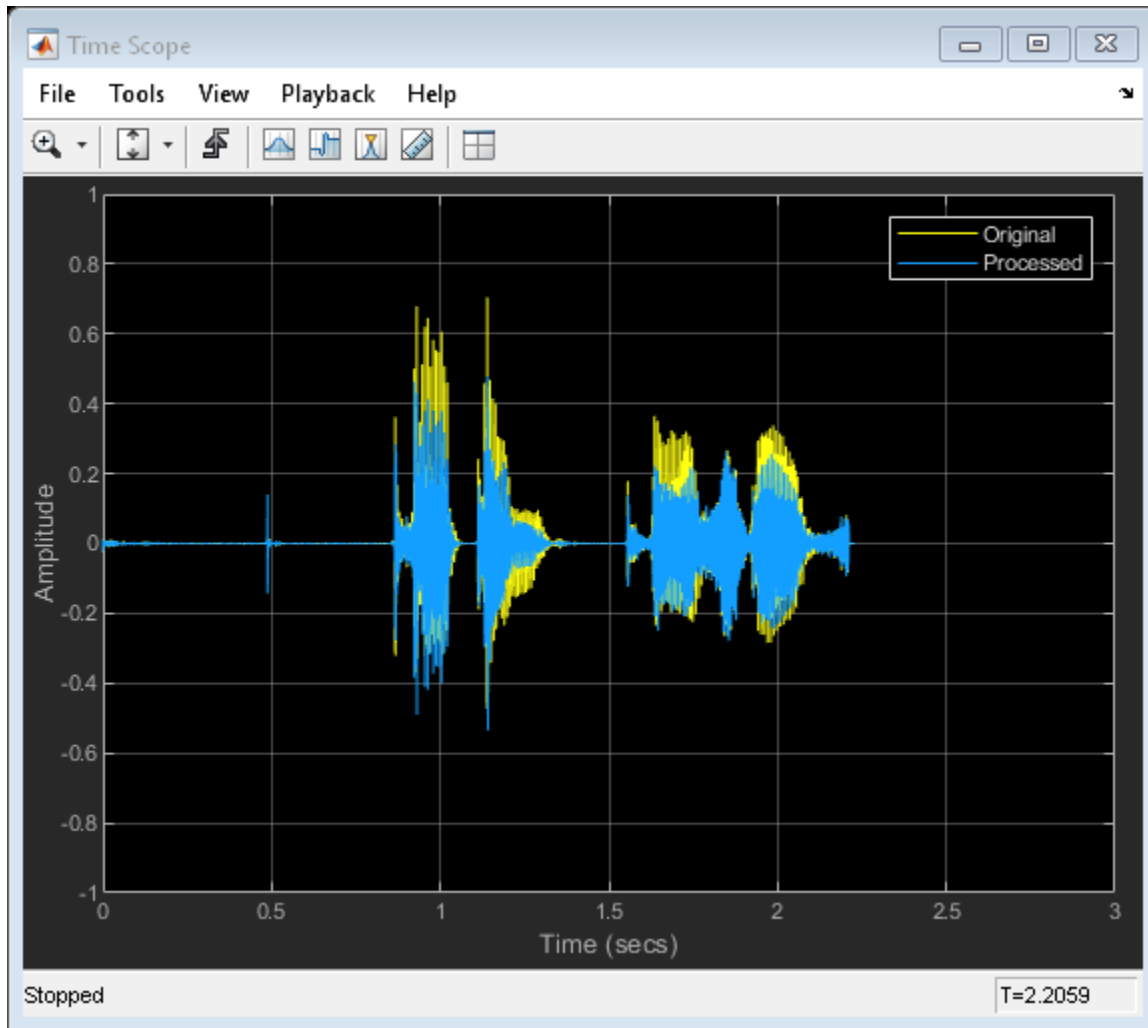
As a best practice, release your objects once done.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioIn = biquadFilter(audioIn,B,A);
    [band1,band2] = crossFilt(audioIn);
    band1compressed = dRCompressor(band1);
    audioOut = band1compressed + band2;
    deviceWriter(audioOut);
    scope([audioIn audioOut])
end
```

As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(crossFilt)
release(dRCompressor)
release(scope)
```



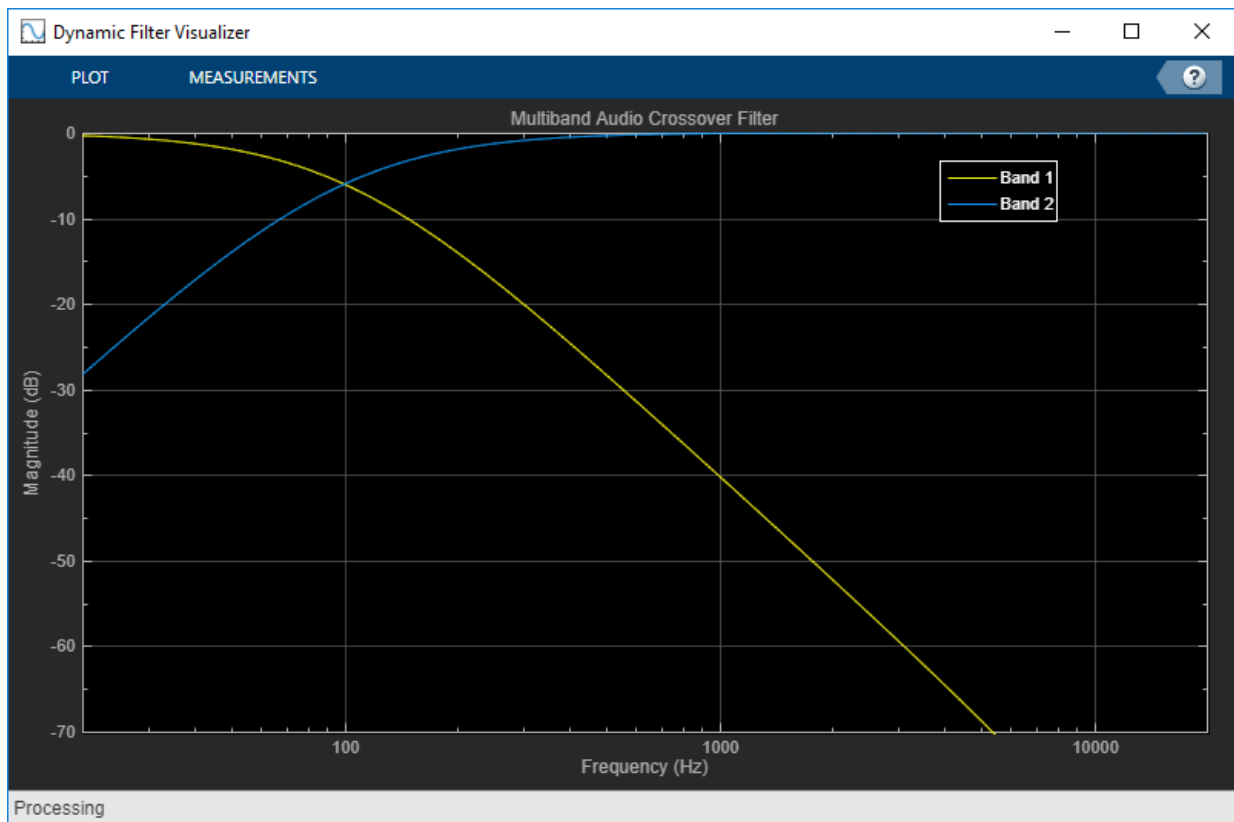


### Tune Crossover Filter Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create a `audioDeviceWriter` to write audio to your sound card. Create a `crossoverFilter` to process the audio data. Call `visualize` to plot the frequency responses of the filters.

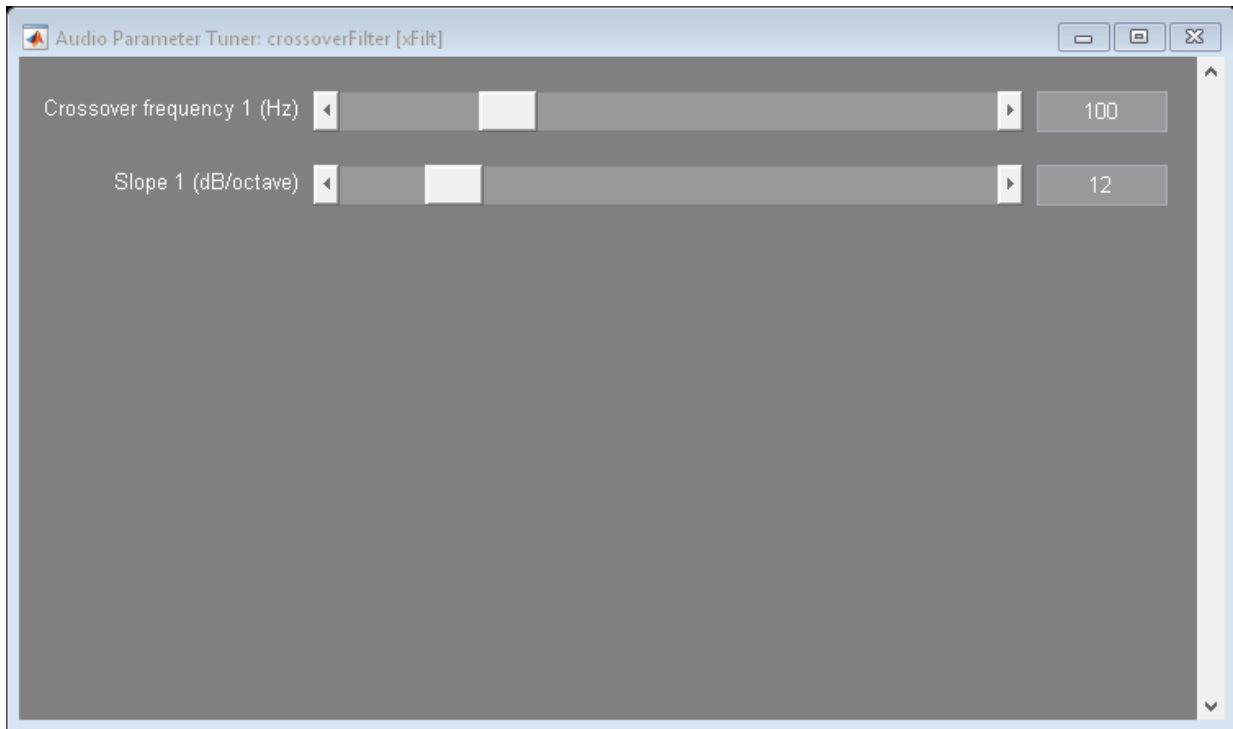
### 3 System objects in Audio Toolbox

```
frameLength = 1024;  
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3', ...  
    'SamplesPerFrame',frameLength);  
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);  
  
xFilt = crossoverFilter('SampleRate',fileReader.SampleRate);  
visualize(xFilt)
```



Call `parameterTuner` to open a UI to tune parameters of the crossover filter while streaming.

```
parameterTuner(xFilt)
```



In an audio stream loop:

- 1 Read in a frame of audio from the file.
- 2 Apply crossover filtering.
- 3 Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the crossover filter and listen to the effect.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    [low,high] = xFilt(audioIn);
    deviceWriter([low(:,1),high(:,1)]);
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

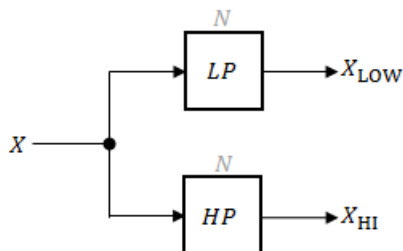
```
release(deviceWriter)
release(fileReader)
release(xFilt)
```

## 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

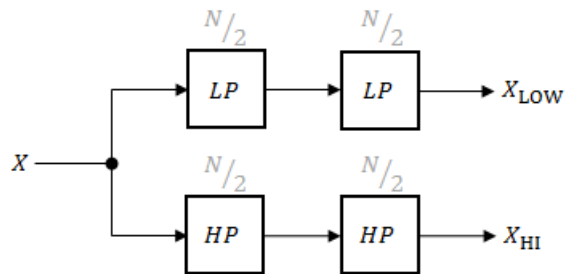
Odd-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.



*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.

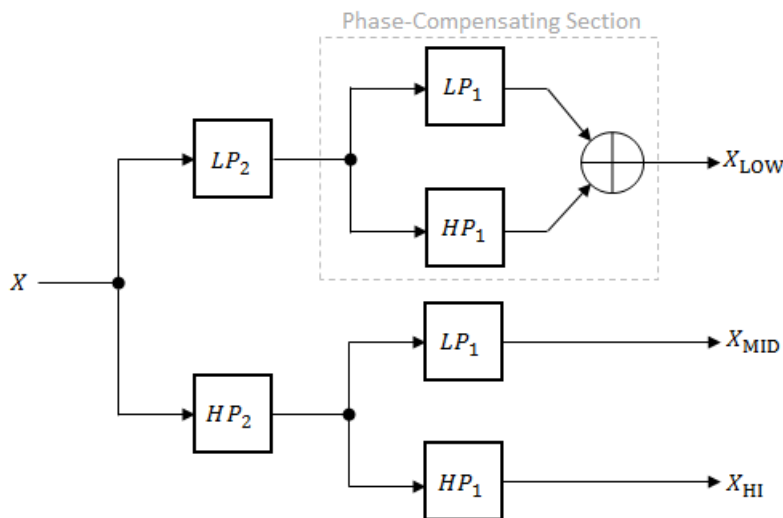


$LP$  and  $HP$  are Butterworth filters of order  $N/2$ , where  $N$  is the order of the overall filter. The filters are implemented as direct-form II transposed second-order sections.

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, 1987, pp. 239-245.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

#### Objects

`multibandParametricEQ`

#### Blocks

Crossover Filter

**Introduced in R2016a**

# visualize

Visualize magnitude response of crossover filter

## Syntax

```
visualize(crossFilt)  
visualize(crossFilt,NFFT)
```

## Description

`visualize(crossFilt)` plots the magnitude response of the `crossoverFilter`. The plot is updated automatically when properties of the object change.

`visualize(crossFilt,NFFT)` specifies an N-point FFT used to calculate the magnitude response.

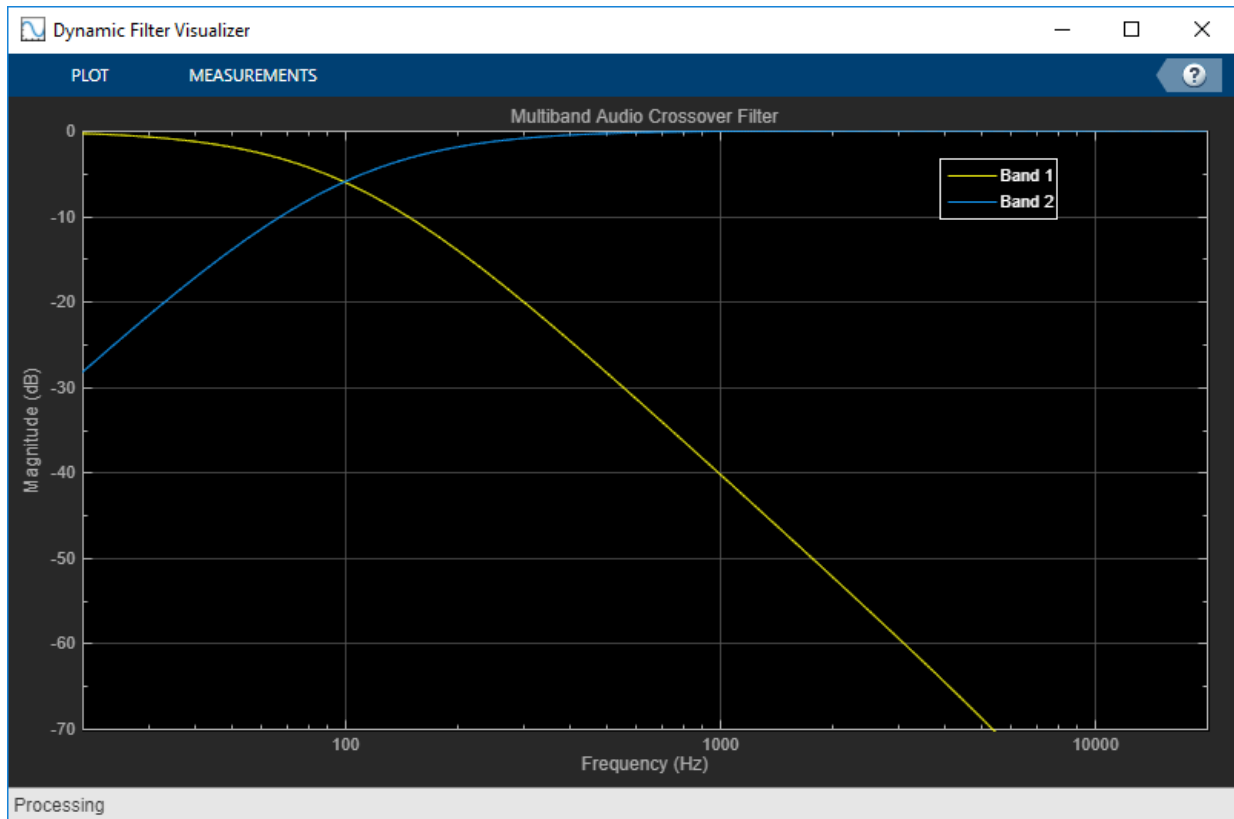
## Examples

### Visualize Magnitude Response of Crossover Filter

Create an object of the `crossoverFilter` object, and then call `visualize` to plot the magnitude response of the filter.

```
crossFilt = crossoverFilter;  
visualize(crossFilt)
```

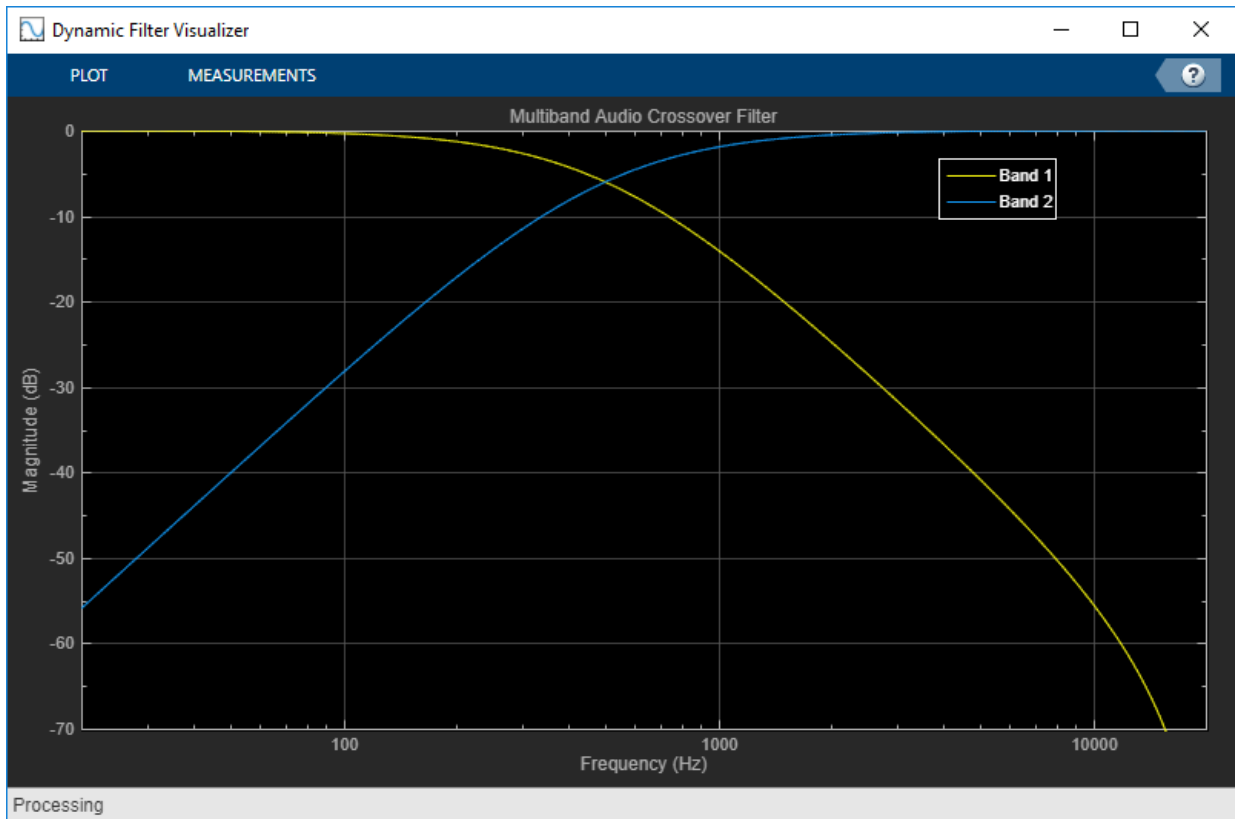
### 3 System objects in Audio Toolbox



Modify the crossover frequency and observe that the plot is updated automatically.

```
crossFilt.CrossoverFrequencies = 500;
```





## Input Arguments

### **crossFilt** — Crossover filter to visualize

object of `crossoverFilter` System object

Crossover filter whose magnitude response you want to plot.

### **NFFT** — N-point FFT

2048 (default) | positive scalar

Number of bins used to calculate the DFT, specified as a positive scalar.

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

## **See Also**

`crossoverFilter`

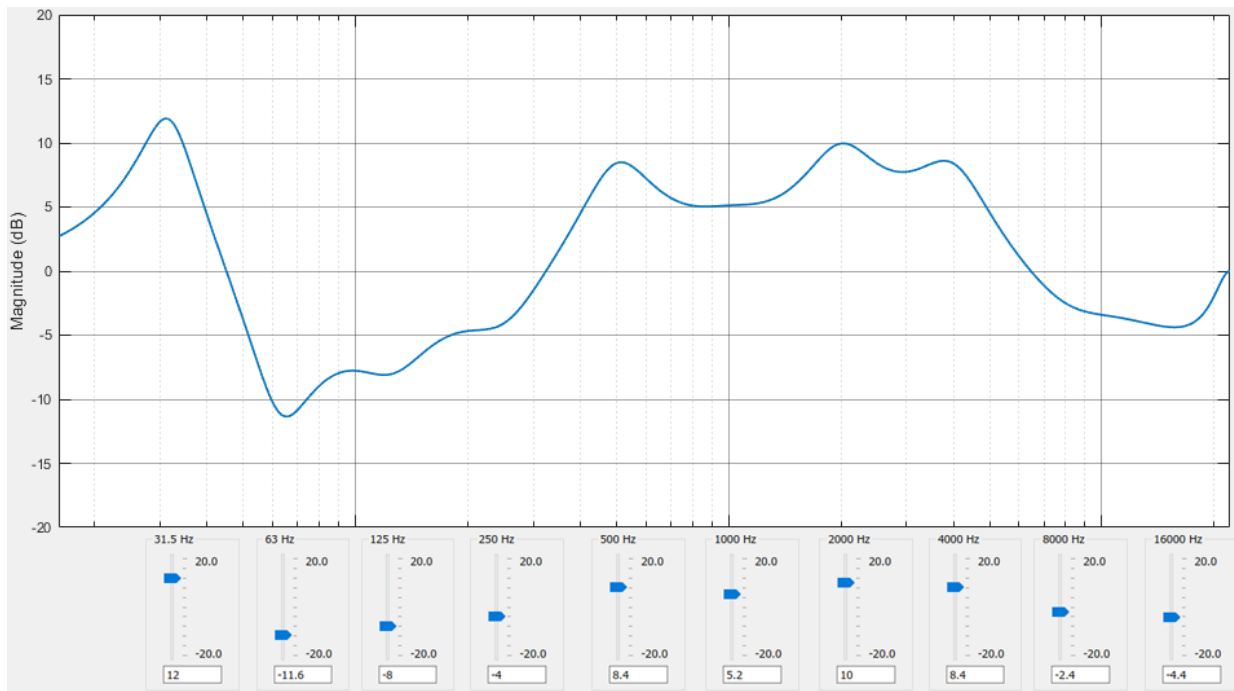
**Introduced in R2016a**

# graphicEQ

Standards-based graphic equalizer

## Description

The `graphicEQ` System object implements a graphic equalizer that can tune the gain on individual octave or fractional octave bands. The object filters the data independently across each input channel over time using the filter specifications. Center and edge frequencies of the bands are based on the ANSI S1.11-2004 standard.



To equalize an audio signal:

- 1 Create the `graphicEQ` object and set its properties.

2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

### Syntax

```
equalizer = graphicEQ  
equalizer = graphicEQ(Name,Value)
```

### Description

`equalizer = graphicEQ` creates a graphic equalizer with default values.

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

Example: `equalizer = graphicEQ('Structure','Parallel','EQOrder','1/3 octave')` creates a System object, `equalizer`, which implements filtering using a parallel structure and one-third octave filter bandwidth.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects \(MATLAB\)](#).

### Gains — Gain of each octave or fractional octave band (dB)

[0 0 0 0 0 0 0 0 0 0] (default) | 10-, 15-, or 30-element row vector

Gain of each octave of fractional octave band in dB, specified as a row vector with a length determined by the `Bandwidth` property:

- '1 octave' -- Specify gains as a 10-element row vector.
- '2/3 octave' -- Specify gains as a 15-element row vector.
- '1/3 octave' -- Specify gains as a 30-element row vector.

Example: `equalizer = graphicEQ('Bandwidth','2/3 octave','Gains',[5,5,5,5,5,0,0,0,0,0,-5,-5,-5,-5,-5])` creates a two-third octave graphic equalizer with specified gains.

You can tune the gains of your graphic equalizer when the object is locked. However, you cannot tune the length of the gains when the object is locked.

**Tunable:** Yes

Data Types: `single` | `double`

### **EQOrder — Order of individual equalizer bands**

2 (default) | positive even integer

Order of individual equalizer bands, specified as a positive even integer. All equalizer bands have the same order.

**Tunable:** No

Data Types: `single` | `double`

### **Bandwidth — Filter bandwidth (octaves)**

'1 octave' (default) | '2/3 octave' | '1/3 octave'

Filter bandwidth in octaves, specified as '1 octave', '2/3 octave', or '1/3 octave'.

The ANSI S1.11-2004 standard defines the center and edge frequencies of your equalizer. The ISO 266:1997(E) standard specifies corresponding preferred frequencies for labeling purposes.

### **1-Octave Bandwidth**

Center frequencies	32 63 126 251 501 1000 1995 3981 7943 15849
Edge frequencies	22 45 89 178 355 708 1413 2818 5623 1122 22387

Preferred frequencies	31.5 63 125 250 500 1000 2000 4000 8000 16000
-----------------------	--

**2/3-Octave Bandwidth**

Center frequencies	25 40 63 100 158 251 398 631 1000 1585 2512 3981 6310 10000 15849
Edge frequencies	20 32 50 79 126 200 316 501 794 1259 1995 3162 5012 7943 12589 19953
Preferred frequencies	25 40 63 100 160 250 400 630 1000 1600 2500 4000 6300 10000 16000

**1/3-Octave Bandwidth**

Center frequencies	25 32 40 50 63 79 100 126 158 200 251 316 398 501 631 794 1000 1259 1585 1995 2512 3162 3981 5012 6310 7943 10000 12589 15849 19953
Edge frequencies	22 28 35 45 56 71 89 112 141 178 224 282 355 447 562 708 891 1122 1413 1778 2239 2818 3548 4467 5623 7079 8913 11220 14125 17783 22387
Preferred frequencies	25 31.5 40 50 63 80 100 125 160 200 250 315 400 500 630 800 1000 1250 1600 2000 2500 3150 4000 5000 6300 8000 10000 12500 16000 20000

**Tunable:** No

Data Types: char | string

**Structure — Type of implementation**

'Cascade' (default) | 'Parallel'

Type of implementation, specified as 'Cascade' or 'Parallel'. See “Algorithms” on page 3-258 and “Graphic Equalization” for information about these implementation structures.

**Tunable:** No

Data Types: char | string

**SampleRate — Input sample rate (Hz)**

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: single | double

## Usage

## Syntax

```
audioOut = equalizer(audioIn)
```

## Description

`audioOut = equalizer(audioIn)` performs graphic equalization on the input signal, `audioIn`, and returns the equalized signal, `audioOut`. The type of equalization is specified by the algorithm and properties of the `graphicEQ` System object, `equalizer`.

## Input Arguments

**audioIn — Audio input to graphic equalizer**

matrix

Audio input to the graphic equalizer, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: single | double

## Output Arguments

**audioOut — Audio output from graphic equalizer**

matrix

Audio output from the graphic equalizer, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `graphicEQ`

<code>createAudioPluginClass</code>	Create audio plugin class that implements functionality of System object
<code>coeffs</code>	Get filter coefficients
<code>info</code>	Get filter information
<code>visualize</code>	Visualize magnitude response of graphic equalizer
<code>parameterTuner</code>	Tune object parameters while streaming

### MIDI

<code>configureMIDI</code>	Configure MIDI connections between audio object and MIDI controller
<code>disconnectMIDI</code>	Disconnect MIDI controls from audio object
<code>getMIDIConnections</code>	Get MIDI connections of audio object

### Common to All System Objects

<code>clone</code>	Create duplicate System object
<code>isLocked</code>	Determine if System object is in use
<code>release</code>	Release resources and allow changes to System object property values and input characteristics
<code>reset</code>	Reset internal states of System object
<code>step</code>	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `graphicEQ` System object to user-facing parameters:



Property	Range	Mapping	Unit
Gains	[-20, 20]	linear	dB

## Examples

### Perform Graphic Equalization

Create objects to read from an audio file and write to your audio device. Use the sample rate of the reader as the sample rate of the writer.

```
frameLength = 512;
reader = dsp.AudioFileReader('RockDrums-48-stereo-11secs.mp3', 'SamplesPerFrame', frameLength);
player = audioDeviceWriter('SampleRate', reader.SampleRate);
```

In an audio stream loop, read audio from a file and play the audio through your audio device.

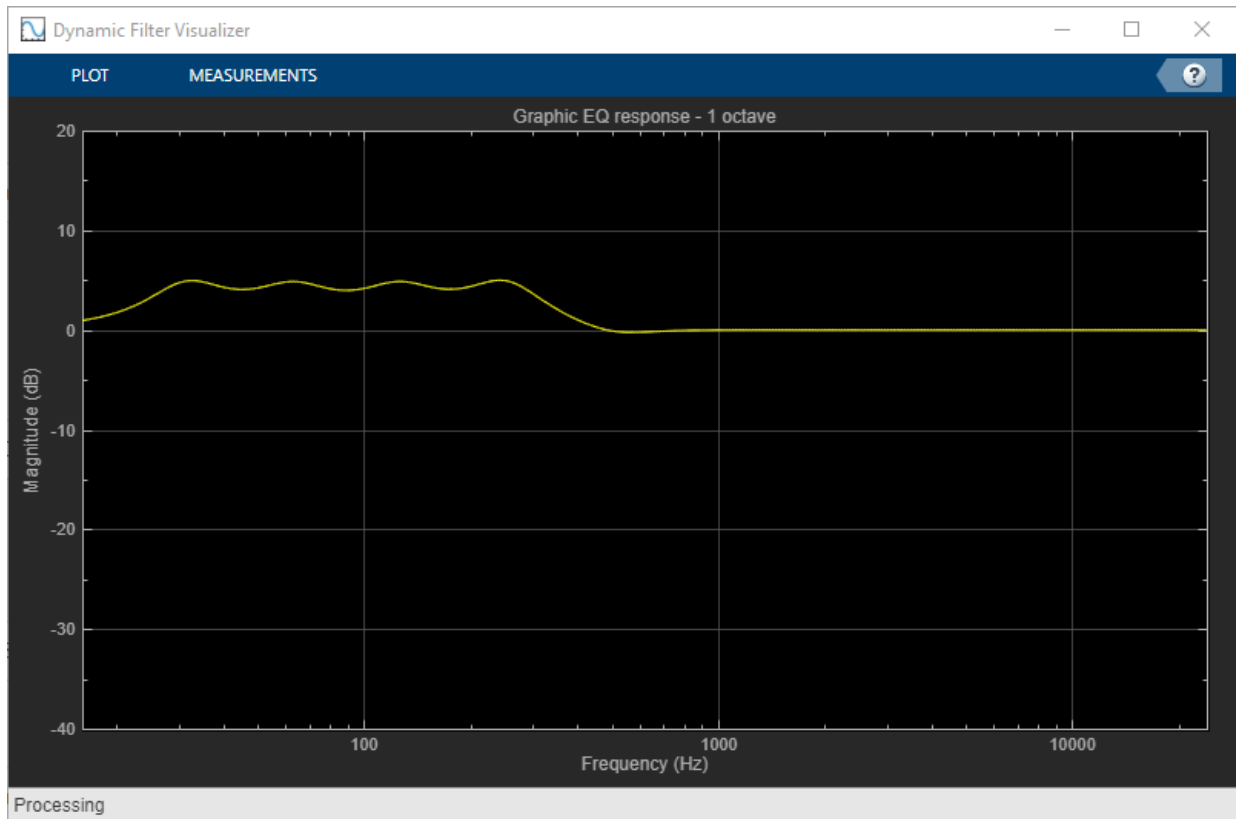
```
while ~isDone(reader)
    x = reader();
    player(x);
end
release(reader)
release(player)
```

Create a one-octave graphic equalizer implemented with a cascade structure. Use the sample rate of the reader as the sample rate of the equalizer.

```
equalizer = graphicEQ( ...
    'Bandwidth', '1 octave', ...
    'Structure', 'Cascade', ...
    'SampleRate', reader.SampleRate);
```

Specify to increase the gain on low frequencies and then visualize the equalizer.

```
equalizer.Gains = [5,5,5,5,0,0,0,0,0,0];
visualize(equalizer)
```



In an audio stream loop, read audio from a file, apply equalization, and then play the equalized audio through your audio device.

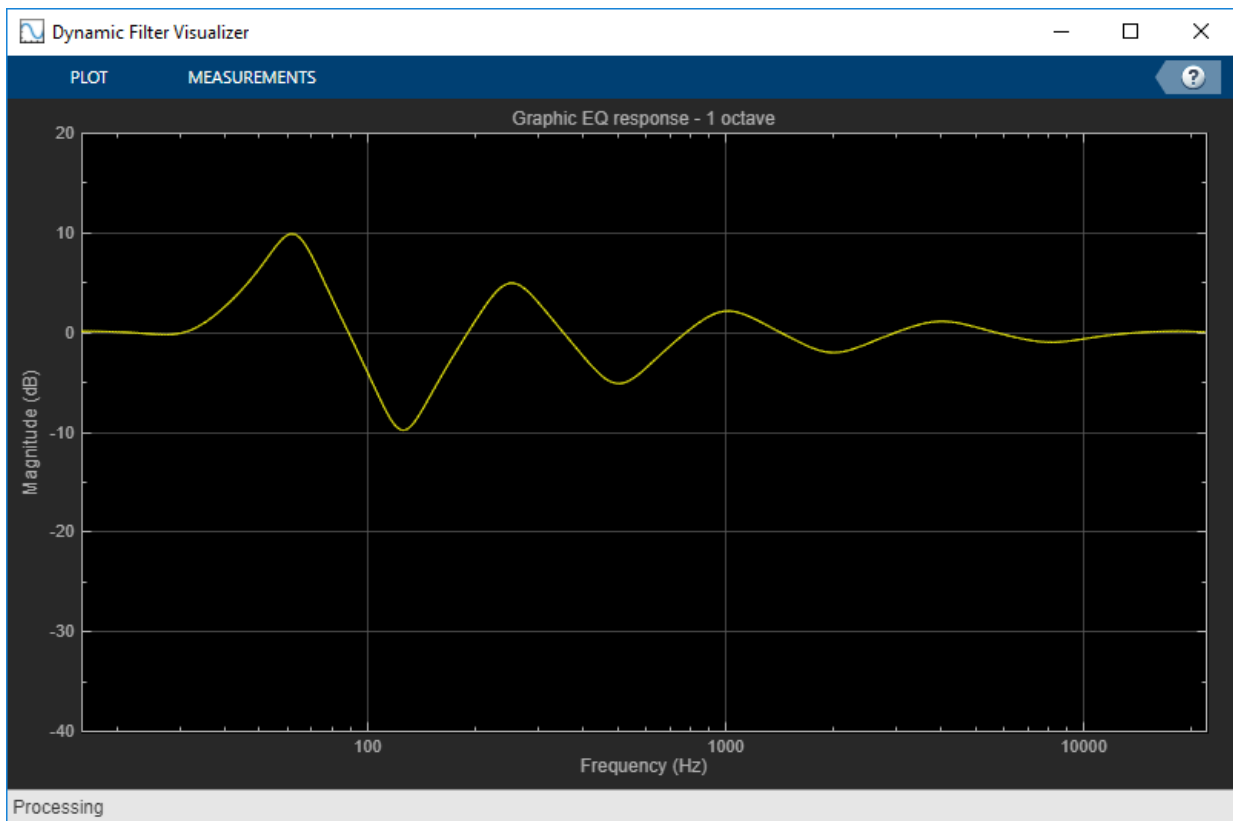
```
while ~isDone(reader)
    x = reader();
    y = equalizer(x);
    player(y);
end
release(reader)
release(player)
```

## Tune Graphic EQ Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card. Create a `graphicEQ` to process the audio data. Call `visualize` to plot the frequency response of the graphic equalizer.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3', 'SamplesPerFrame', ...
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

equalizer = graphicEQ('SampleRate',fileReader.SampleRate,'Gains',[0,10,-10,5,-5,2,-2,1,0]);
visualize(equalizer)
```



Call `parameterTuner` to open a UI to tune parameters of the equalizer while streaming.

```
parameterTuner(equalizer)
```

In an audio stream loop:

- 1 Read in a frame of audio from the file.
- 2 Apply equalization.
- 3 Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the equalizer and listen to the effect.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = equalizer(audioIn);
    deviceWriter(audioOut);
    drawnow limitrate % required to update parameter
end
```

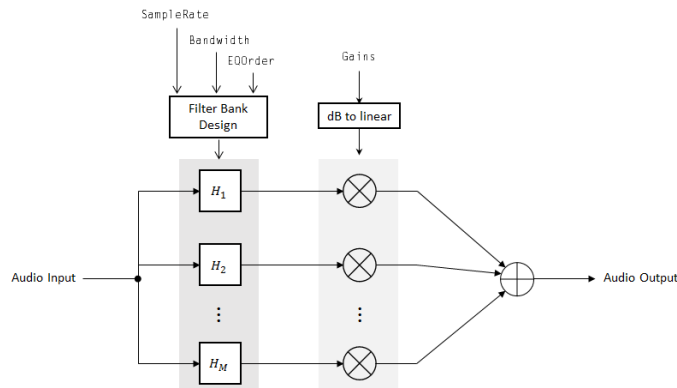
As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(equalizer)
```

## Algorithms

The implementation of your graphic equalizer depends on the `Structure` property. See “Graphic Equalization” for a discussion of the pros and cons of the parallel and cascade implementations. Refer to the following sections to understand how these algorithms are implemented in Audio Toolbox.

## Parallel Structure



### Filter Bank Design

The parallel implementation designs the individual equalizers using the `octaveFilter design` method and spaces them on the spectrum according to the ANSI S1.11-2004 standard.

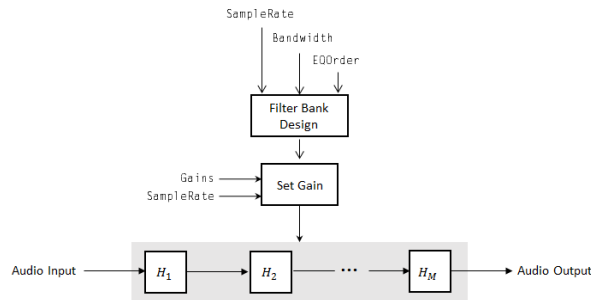
If you set the `SampleRate` property so that the Nyquist frequency ( $\text{SampleRate}/2$ ) is less than the final bandpass edge defined by the ANSI S1.11-2004 standard, then:

- The final bandpass filter is the one whose upper bandpass edge is less than the Nyquist frequency.
- The final filter is implemented as a highpass filter designed by the `designParamEQ` function.

### Real-Time Computation

- 1 The input signal is fed into a filterbank of  $M$  filters, where  $M$  depends on the specified `Bandwidth` and `SampleRate` properties.
- 2 Each branch of the filterbank is multiplied by the linear form of the corresponding element of the `Gains` property.
- 3 The branches are summed and the output signal is returned.

## Cascade Structure



### Filter Bank Design

The cascade implementation designs the graphic equalizer filter bank using the `multibandParametricEQ` System object.

### Gain Setting

If the `EQOrder` property is set to 2, then a gain correction is calculated according to [1]. The gain correction is independent of the requested gains. The gain correction is recomputed during the real-time processing only if the `SampleRate` property is modified.

If the `EQOrder` property is not set to 2, no gain correction is applied, and the requested gains are passed on to the `multibandParametricEQ` object.

### Real-Time Computation

The input signal is fed into a cascade of  $M$  biquad filters, where  $M$  depends on the specified `Bandwidth` and `SampleRate` properties.

## References

- [1] Oliver, Richard J., and Jean-Marc Jot. "Efficient Multi-Band Digital Audio Graphic Equalizer with Accurate Frequency Response Control." Presented at the 139th Convention of the AES, New York, October 2015.
- [2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters*. ANSI S1.11-2004. Melville, NY: Acoustical Society of America, 2009.

[3] International Organization for Standardization. *Acoustics -- Preferred frequencies*. ISO 266:1997(E). Second Edition. 1997.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

## See Also

### Blocks

Graphic EQ | Parametric EQ Filter

### Functions

designParamEQ | designShelvingEQ | designVarSlopeFilter |  
multibandParametricEQ

### Topics

“Graphic Equalization”  
“Equalization”

**Introduced in R2017b**

## info

Get filter information

## Syntax

```
infoStruct = info(obj)
```

## Description

`infoStruct = info(obj)` returns a structure, `infoStruct`, containing information about `obj`.

## Examples

### Get Graphic Equalizer Standards-Based Frequencies

Create a `graphicEQ` System object™. Call `info` to return a structure containing standards-based center, edge, and preferred frequencies.

```
equalizer = graphicEQ;  
info(equalizer)
```

```
ans = struct with fields:  
    CenterFrequencies: [1x10 double]  
    EdgeFrequencies: [1x11 double]  
    PreferredFrequencies: [31.5000 63 125 250 500 1000 2000 4000 8000 16000]
```

### octaveFilterBank Info

Create a default `octaveFilterBank`. Call `info` to return a struct containing information about the octave filter bank.



```
octFiltBank = octaveFilterBank;

infoStruct = info(octFiltBank)

infoStruct = struct with fields:
    CenterFrequencies: [1x10 double]
    BandedgeFrequencies: [1x11 double]
    GroupDelays: [1x10 double]
```

### gammatoneFilterBank Info

Create a default `gammatoneFilterBank`. Call `info` to return a struct containing information about the octave filter bank.

```
gammaFiltBank = gammatoneFilterBank;

infoStruct = info(gammaFiltBank)

infoStruct = struct with fields:
    CenterFrequencies: [1x32 double]
    Bandwidths: [1x32 double]
    GroupDelays: [1x32 double]
```

## Input Arguments

### **obj** — Object to get information from

`graphicEQ` | `gammatoneFilterBank` | `octaveFilterBank`

Object to get information from, specified as an object of `gammatoneFilterBank`, `octaveFilterBank`, or `graphicEQ`.

## Output Arguments

### **infoStruct** — Struct containing object information

struct

Struct containing information about the input obj.

## **See Also**

`gammatoneFilterBank` | `graphicEQ` | `octaveFilterBank`

**Introduced in R2017b**

# visualize

Visualize magnitude response of graphic equalizer

## Syntax

```
visualize(equalizer)  
visualize(equalizer,NFFT)
```

## Description

`visualize(equalizer)` plots the magnitude response of the `graphicEQ` object, `equalizer`. The plot is updated automatically when properties of the object change.

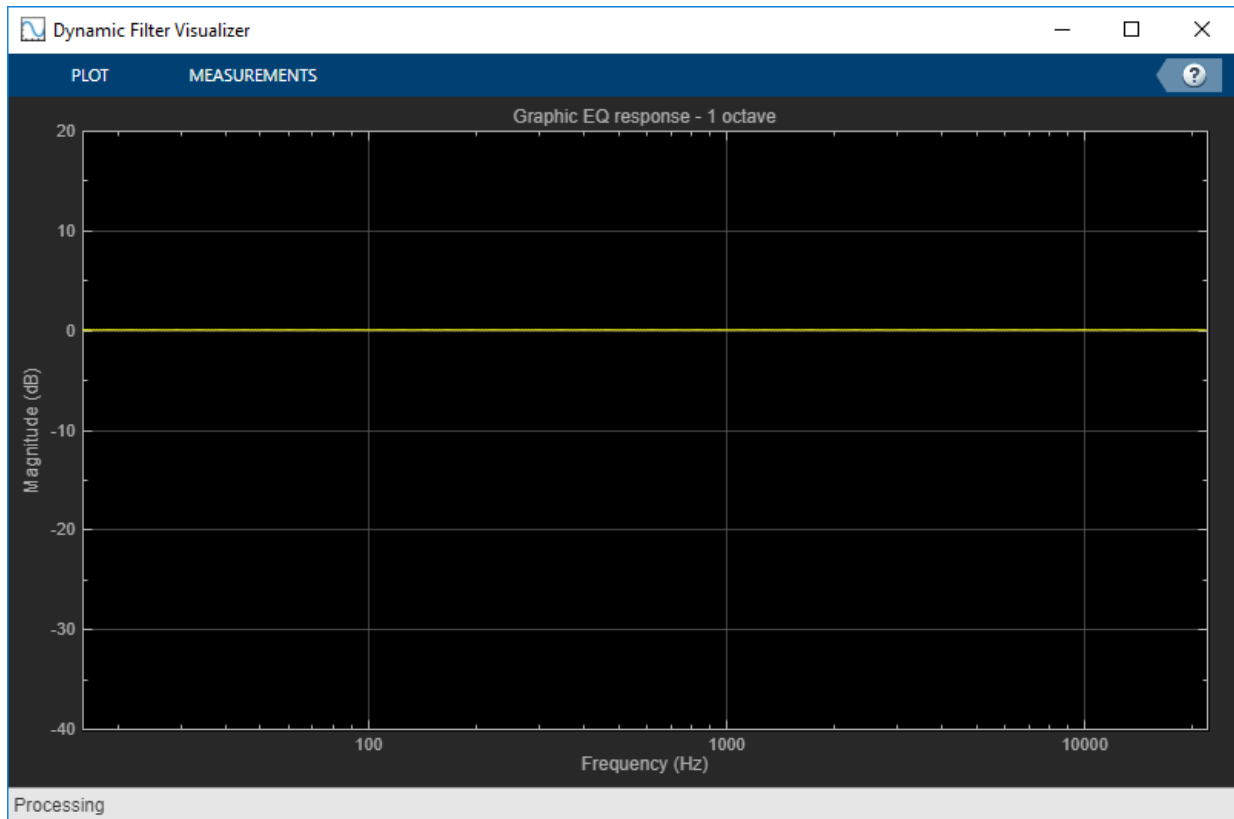
`visualize(equalizer,NFFT)` specifies an N-point FFT used to calculate the magnitude response.

## Examples

### Visualize Magnitude Response of Graphic Equalizer

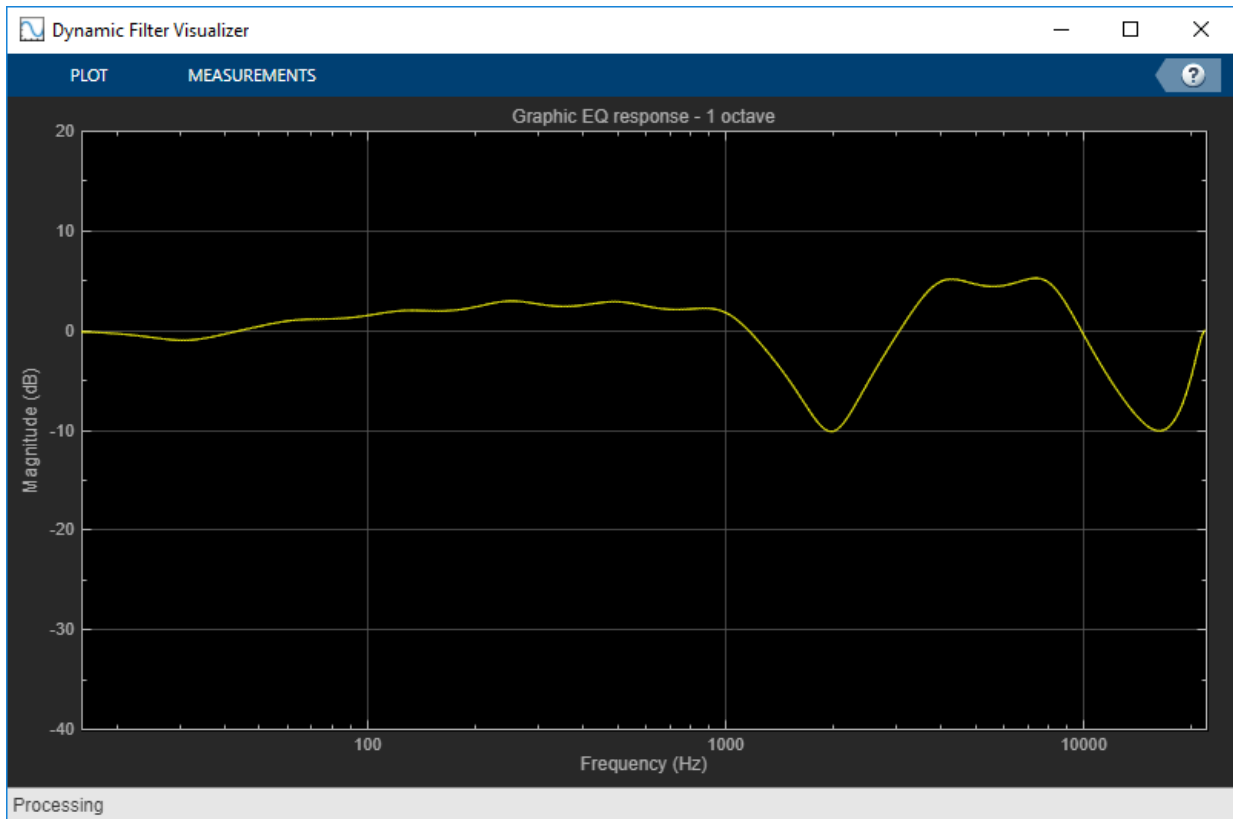
Create a default object of the `graphicEQ` System object™ and then call `visualize`.

```
equalizer = graphicEQ;  
visualize(equalizer)
```



Set the gains of the graphic equalizer to new values. The visualization of the magnitude response updates automatically.

```
equalizer.Gains = [-1,1,2,3,3,2,-10,5,5,-10];
```



## Input Arguments

### **equalizer** — Graphic equalizer to visualize

object of `graphicEQ` System object

Graphic equalizer whose magnitude response you want to plot.

### **NFFT** — N-point FFT

2048 (default) | positive scalar

Number of bins used to calculate the DFT, specified as a positive scalar.

Data Types: `single` | `double`

## **See Also**

`graphicEQ`

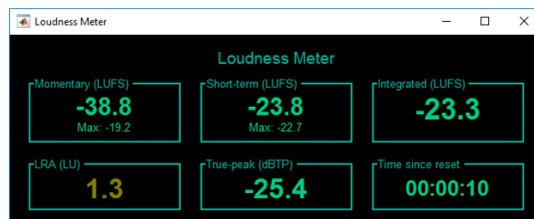
**Introduced in R2017b**

# loudnessMeter

Standard-compliant loudness measurements

## Description

The `loudnessMeter` System object computes the loudness, loudness range, and true-peak of an audio signal in accordance with EBU R 128 and ITU-R BS.1770-4 standards.



To implement loudness metering:

- 1 Create the `loudnessMeter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

## Syntax

```
loudMtr = loudnessMeter  
loudMtr = loudnessMeter(Name, Value)
```

### Description

`loudMtr = loudnessMeter` creates a System object, `loudMtr`, that performs loudness metering independently across each input channel.

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

Example: `loudMtr = loudnessMeter('ChannelWeights', [1.2, 0.8], 'SampleRate', 12000)` creates a System object, `loudMtr`, with channel weights of 1.2 and 0.8, and a sample rate of 12 kHz.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects (MATLAB)*.

#### **ChannelWeights — Linear weighting applied to each input channel**

[1, 1, 1, 1.41, 1.41] (default) | nonnegative row vector

Linear weighting applied to each input channel, specified as a row vector of nonnegative values. The number of elements in the row vector must be equal to or greater than the number of input channels. Excess values in the vector are ignored.

The default channel weights follow the ITU-R BS.1170-4 standard. To use the default channel weights, specify the input signal channels as a matrix in this order: [Left, Right, Center, Left surround, Right surround].

As a best practice, specify the `ChannelWeights` property in order: [Left, Right, Center, Left surround, Right surround].

**Tunable:** Yes

Data Types: `single` | `double`



**UseRelativeScale — Use relative scale for loudness measurements**`false (default) | true`

Use relative scale for loudness measurements, specified as a logical scalar.

- `false` -- The loudness measurements are absolute and returned in loudness units full scale (LUFS).
- `true` -- The loudness measurements are relative to the `TargetLoudness` value and returned in loudness units (LU).

**Tunable:** No

Data Types: `logical`

**TargetLoudness — Target loudness level for relative scale (LUFS)**`-23 (default) | real scalar`

Target loudness level for relative scale in LUFS, specified as a real scalar.

For example, if the `TargetLoudness` is `-23` LUFS, then a loudness value of `-23` LUFS is reported as `0` LU.

**Tunable:** Yes

**Dependencies**

To enable this property, set `UseRelativeScale` to `true`.

Data Types: `single` | `double`

**SampleRate — Input sample rate (Hz)**`44100 (default) | positive scalar`

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: `single` | `double`

# Usage

# Syntax

```
[momentary,shortTerm,integrated,range,peak] = loudMtr(audioIn)
```

# Description

[momentary,shortTerm,integrated,range,peak] = loudMtr(audioIn) returns measurement values for momentary and short-term loudness of the input to your loudness meter, and the true-peak value of the current input frame, `audioIn`. It also returns the integrated loudness and loudness range of the input to your loudness meter since the last time reset was called.

# Input Arguments

## **audioIn** — Audio input to loudness meter

matrix

Audio input to the loudness meter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

---

**Note** If you use the default `ChannelWeights` of the `LoudnessMeter`, as a best practice, specify the input channels in this order: [Left, Right, Center, Left surround, Right surround].

---

Data Types: `single` | `double`

# Output Arguments

## **momentary** — Momentary loudness (LUFS)

column vector

Momentary loudness in loudness units relative to full scale (LUFS), returned as a column vector with the same number of rows as `audioIn`.

By default, loudness measurements are returned in LUFS. If you set the `UseRelativeScale` property to `true`, loudness measurements are returned in loudness units (LU).

Data Types: `single` | `double`

### **shortTerm** — Short-term loudness (LUFS)

column vector

Short-term loudness in loudness units relative to full scale (LUFS), returned as a column vector with the same number of rows as `audioIn`.

By default, loudness measurements are returned in LUFS. If you set the `UseRelativeScale` property to `true`, loudness measurements are returned in loudness units (LU).

Data Types: `single` | `double`

### **integrated** — Integrated loudness (LUFS)

column vector

Integrated loudness in loudness units relative to full scale (LUFS), returned as a column vector with the same number of rows as `audioIn`.

By default, loudness measurements are returned in LUFS. If you set the `UseRelativeScale` property to `true`, loudness measurements are returned in loudness units (LU).

Data Types: `single` | `double`

### **range** — Loudness range (LU)

column vector

Loudness range in loudness units (LU), returned as a column vector with the same number of rows as `audioIn`.

Data Types: `single` | `double`

### **peak** — True-peak loudness (dB-TP)

scalar

True-peak loudness in dB-TP, returned as a column vector with the same number of rows as `audioIn`.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to LoudnessMeter

`visualize` Open 'EBU Mode' meter display

### Common to All System Objects

`clone` Create duplicate System object  
`isLocked` Determine if System object is in use  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object  
`step` Run System object algorithm

## Examples

### Loudness of Audio Signal

Create a `dsp.AudioFileReader` System object™ to read in an audio file. Create a `LoudnessMeter` System object. Use the sample rate of the audio file as the sample rate of the `LoudnessMeter`.

```
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3');  
loudMtr = loudnessMeter('SampleRate',fileReader.SampleRate);
```

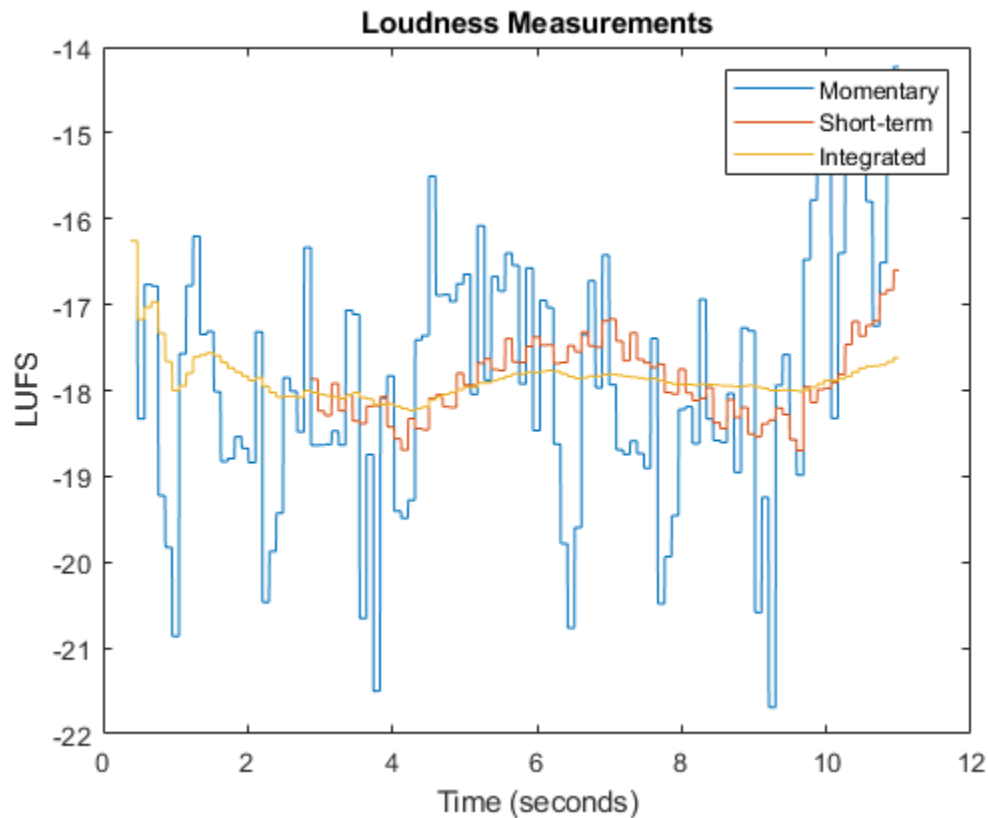
Read in the audio file in an audio stream loop. Use the loudness meter to determine the momentary, short-term, and integrated loudness of the audio signal. Cache the loudness measurements for analysis.

```
momentary = [];  
shortTerm = [];  
integrated = [];  
  
while ~isDone(fileReader)
```

```
x = fileReader();  
[m,s,i] = loudMtr(x);  
momentary = [momentary;m];  
shortTerm = [shortTerm;s];  
integrated = [integrated;i];  
end  
  
release(fileReader)
```

Plot the momentary, short-term, and integrated loudness of the audio signal.

```
t = linspace(0,11,length(momentary));  
plot(t,[momentary,shortTerm,integrated])  
title('Loudness Measurements')  
legend('Momentary','Short-term','Integrated')  
xlabel('Time (seconds)')  
ylabel('LUFS')
```



#### Plot Momentary Loudness and Loudness Range of Audio Stream

Create an audio file reader and an audio device writer.

```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3', ...  
    'SamplesPerFrame',1024);  
fs = fileReader.SampleRate;  
deviceWriter = audioDeviceWriter('SampleRate',fs);
```

Create a time scope to visualize your audio stream loop.

```

timeScope = dsp.TimeScope('NumInputPorts',2, ...
    'SampleRate',fs, ...
    'TimeSpanOvverrunAction','Scroll', ...
    'LayoutDimensions',[2,1], ...
    'TimeSpan',5, ...
    'BufferLength',5*fs);

% Top subplot of scope
timeScope.Title = 'Momentary Loudness';
timeScope.YLabel = 'LUFS';
timeScope.YLimits = [-40, 0];

% Bottom subplot of scope
timeScope.ActiveDisplay = 2;
timeScope.Title = 'Loudness Range';
timeScope.YLabel = 'LU';
timeScope.YLimits = [-1, 2];

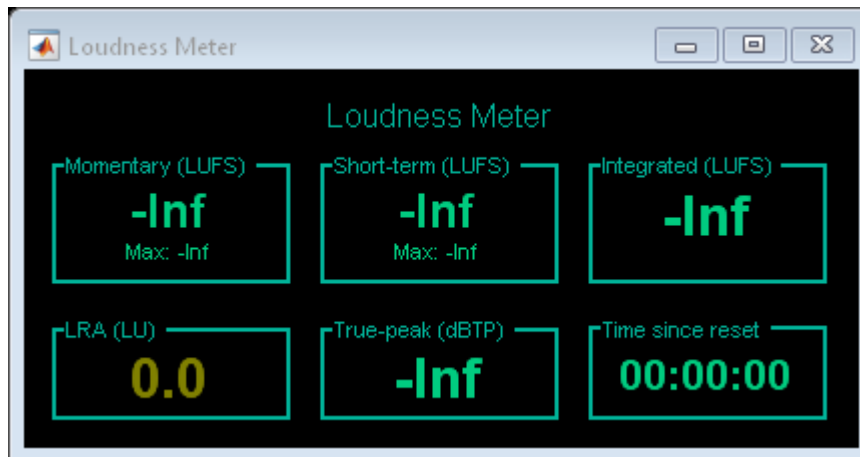
```

Create a loudness meter. Use the sample rate of your input file as the sample rate of your loudness meter. Call `visualize` to open an 'EBU-mode' visualization for your loudness meter.

```

loudMtr = loudnessMeter('SampleRate',fs);
visualize(loudMtr)

```



In an audio stream loop:

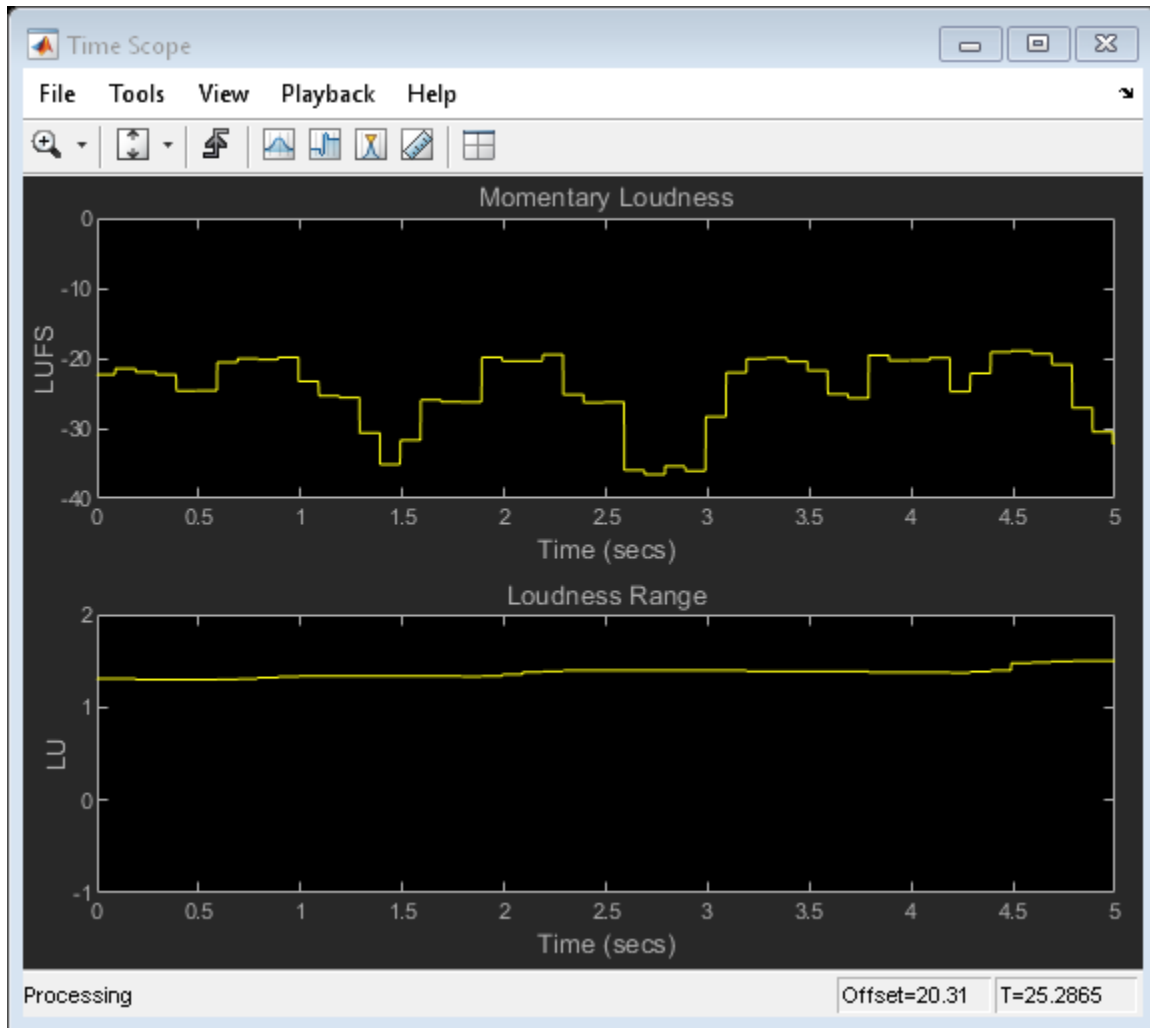
- Read in your audio file.
- Compute the momentary loudness and loudness range.
- Visualize the momentary loudness and loudness range on your time scope.
- Play the audio signal.

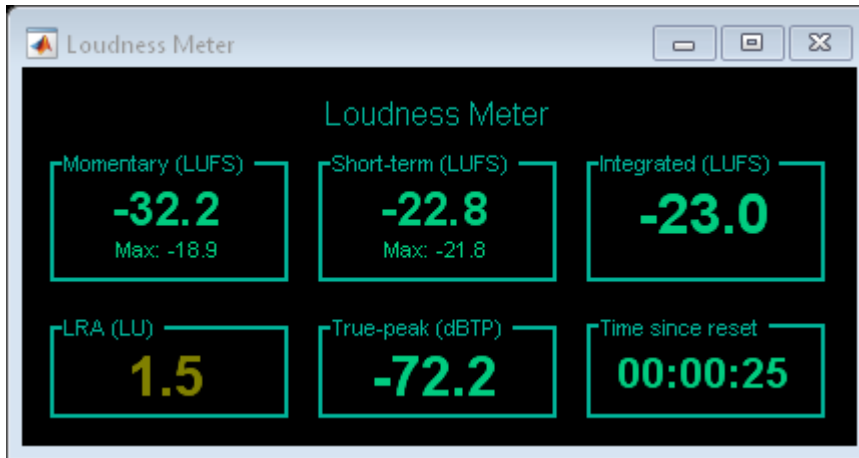
The 'EBU-mode' loudness meter visualization updates automatically while it is open. As a best practice, release your file reader and device writer once the loop is completed.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    [momentaryLoudness,~,~,LRA] = loudMtr(audioIn);
    timeScope(momentaryLoudness,LRA);
    deviceWriter(audioIn);
end

release(fileReader)
release(deviceWriter)
```







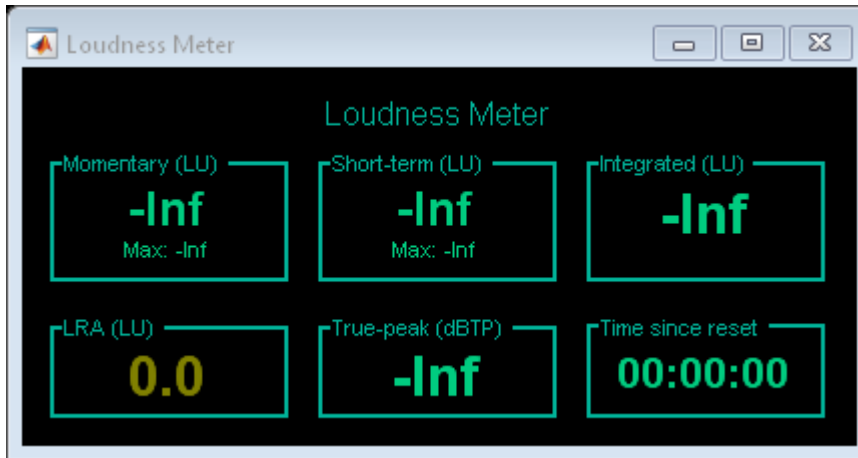
#### Relative Scale for Loudness Measurements

Create an audio file reader to read in an audio file. Create an audio device writer to write the audio file to your audio device. Use the sample rate of your file reader as the sample rate of your device writer.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav',...  
    'SamplesPerFrame',1024);  
fs = fileReader.SampleRate;  
deviceWriter = audioDeviceWriter('SampleRate',fs);
```

Create a loudness meter with the target loudness set to the default -23 LUFS. Open the 'EBU-mode' loudness meter visualization.

```
loudMtr = loudnessMeter('UseRelativeScale',true);  
visualize(loudMtr)
```



Create a time scope to visualize your audio signal and its measured relative momentary and short-term loudness.

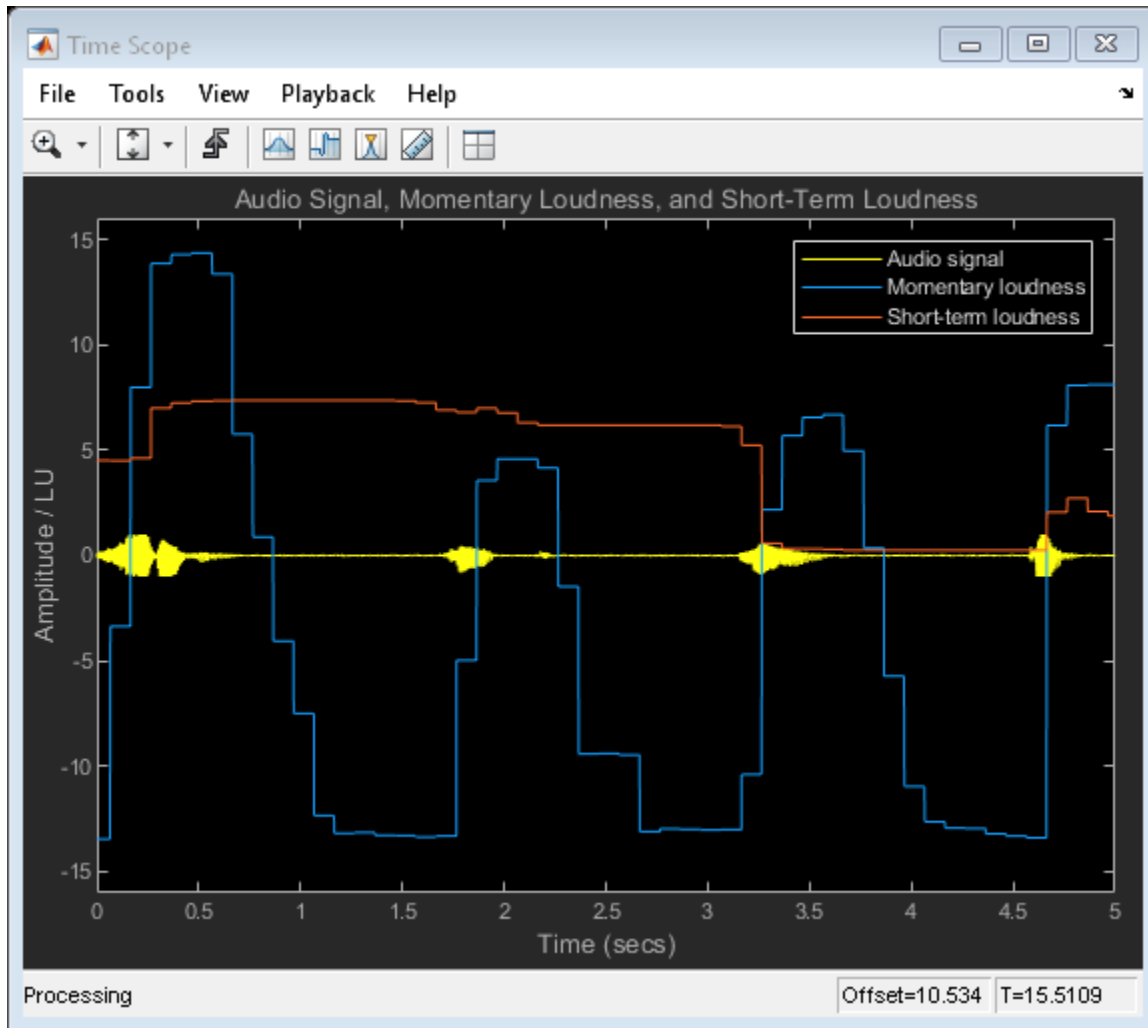
```
scope = dsp.TimeScope( ...
    'NumInputPorts',3, ...
    'SampleRate',fs, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',5, ...
    'BufferLength',5*fs, ...
    'Title','Audio Signal, Momentary Loudness, and Short-Term Loudness', ...
    'ChannelNames',{'Audio signal','Momentary loudness','Short-term loudness'}, ...
    'YLimits',[-16,16], ...
    'YLabel','Amplitude / LU', ...
    'ShowLegend',true);
```

In an audio stream loop, listen to and visualize the audio signal.

```
while ~isDone(fileReader)
    x = fileReader();
    [momentary,shortTerm] = loudMtr(x);
    scope(x,momentary,shortTerm)
    deviceWriter(x);
end

release(deviceWriter)
release(fileReader)
```

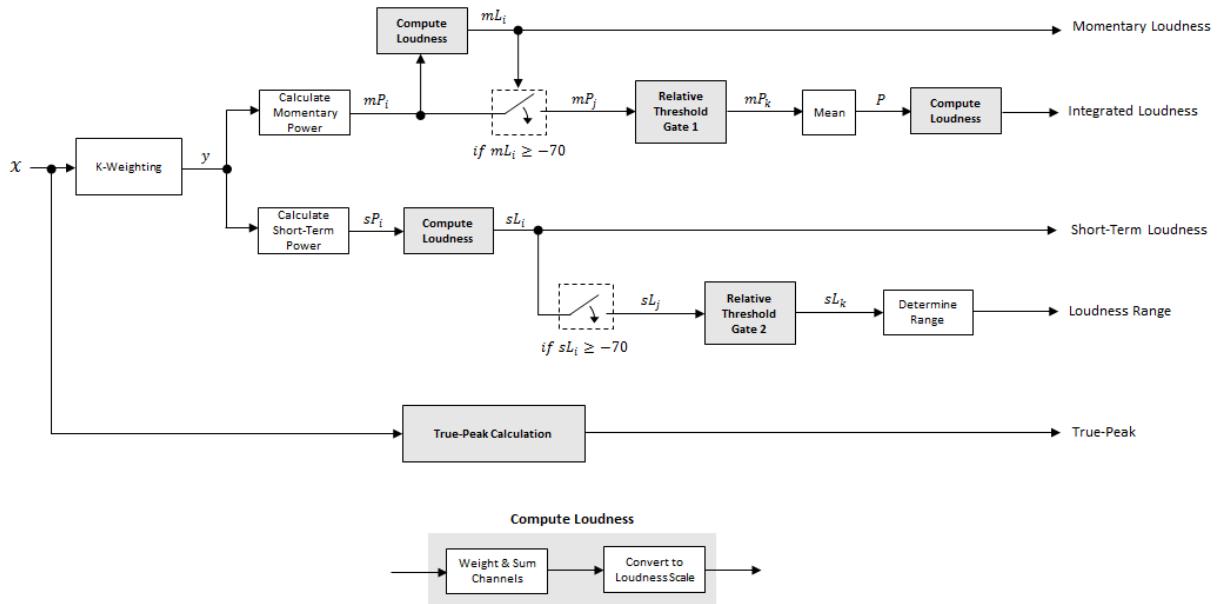




## Algorithms

The loudnessMeter System object calculates the momentary loudness, short-term loudness, integrated loudness, loudness range (LRA), and true-peak value of an audio signal. You can specify any number of channels and nondefault channel weights used for

loudness measurements. The `loudnessMeter` algorithm is described for the general case of  $n$  channels with default channel weights.



## Loudness Measurements

The input channels,  $x$ , pass through a K-weighted `weightingFilter`. The K-weighted filter shapes the frequency spectrum to reflect perceived loudness.

### Momentary Loudness and Integrated Loudness

- 1 The K-weighted channels,  $y$ , are divided into 0.4-second segments with 0.3-second overlap. If the required number of samples have not been collected yet, the `loudnessMeter` System object returns the last computed values for momentary and integrated loudness. If enough samples have been collected, then the power (mean square) of each segment of the K-weighted channels is calculated:

$$mP_i = \frac{1}{w} \sum_{k=1}^w y_i^2[k]$$

- $mP_i$  is the momentary power of the  $i$ th segment.
  - $w$  is the segment length in samples.
- 2 The momentary loudness,  $mL$ , is computed in LUFS for each segment:

$$mL_i = -0.691 + 10 \log_{10} \left( \sum_{c=1}^n G_c \times mP_{(i,c)} \right)$$

- $G_c$  is the weighting for channel  $c$ .

$mL$  is the momentary loudness returned by your `LoudnessMeter` System object. It is also used internally to calculate the integrated loudness (steps 3-6).

- 3 The *integrated loudness* measurement considers the audio signal since the last reset of your loudness meter. To calculate integrated loudness, the momentary power is passed through a gating system. The gate system pauses the measurement during periods of low sound, such as stretches of silence in a movie.

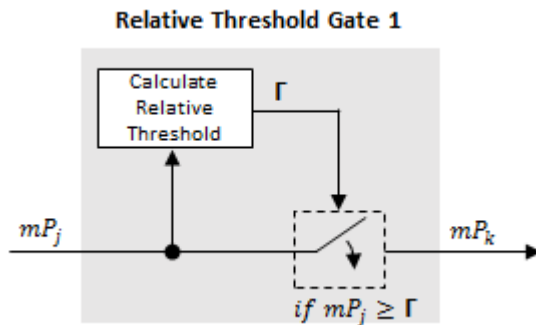
The momentary power segment is gated using the corresponding momentary loudness segment calculation:

$$mP_i \rightarrow mP_j$$

$$j = \{ i \mid mL_i \geq -70 \}$$

$mP_j$  is cached until your `LoudnessMeter` is reset.

- 4 The momentary power subset,  $mP_j$ , passes through a relative threshold gate.



- a The relative threshold,  $\Gamma$ , is computed:

$$\Gamma = -0.691 + 10\log_{10}\left(\sum_{c=1}^n G_c \times l_c\right) - 10$$

$l_c$  is the mean momentary power of channel  $c$ :

$$l_c = \frac{1}{|j|} \sum_j mP_{(j,c)}$$

**b** The momentary power subset,  $mP_j$ , is gated using relative threshold  $\Gamma$ :

$$mP_j \rightarrow mP_k$$

$$k = \{ j \mid mP_j \geq \Gamma \}$$

The relative threshold is recomputed during each call to your `loudnessMeter` object. The cached values of  $mP_j$  are gated again depending on the updated value of  $\Gamma$ .

**5** The momentary power segments are averaged:

$$P = \frac{1}{|k|} \sum_k mP_k$$

**6** The integrated loudness is computed in LUFS by passing the mean momentary power,  $P$ , through the Compute Loudness system:

$$\text{Integrated Loudness} = -0.691 + 10\log_{10}\left(\sum_{c=1}^n G_c \times P_c\right)$$

### Short-Term Loudness and Loudness Range

**1** The K-weighted channels,  $y$ , are divided into 3-second segments with 2.9-second overlap. If the required number of samples have not been collected yet, the `loudnessMeter` System object returns the last computed values for short-term loudness and loudness range. If enough samples have been collected, then the power (mean square) of each K-weighted channel is calculated:

$$sP_i = \frac{1}{w} \sum_{k=1}^w y_i^2[k]$$

- $sP_i$  is the short-term power of the  $i$ th segment of a channel.



- $w$  is the segment length in samples.
- 2 The short-term loudness,  $sL$ , is computed in LUFS for each segment:

$$sL_i = -0.691 + 10 \log_{10} \left( \sum_{c=1}^n G_c \times sP_{(i,c)} \right)$$

- $G_c$  is the weighting for channel  $c$ .

$sL$  is the short-term loudness returned by your `loudnessMeter` System object. It is also used internally to calculate the loudness range (steps 3-5).

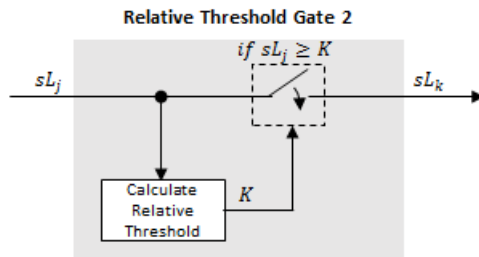
- 3 The short-term loudness is gated using an absolute threshold:

$$sL_i \rightarrow sL_j$$

$$j = \{ i \mid sL_i \geq -70 \}$$

$sL_j$  is cached until your `loudnessMeter` is reset.

- 4 The short-term loudness subset,  $sL_j$  passes through a relative threshold gate.



- a The gated short-term loudness is converted back to linear and then the mean is taken:

$$sP_j = \frac{1}{|j|} \sum_j 10^{(sL_j/10)}$$

The relative threshold,  $K$ , is computed:

$$K = -20 + 10 \log_{10}(sP_j)$$

- b The short-term loudness subset,  $sL_j$ , is gated using the relative threshold:

$$sL_j \rightarrow sL_k$$

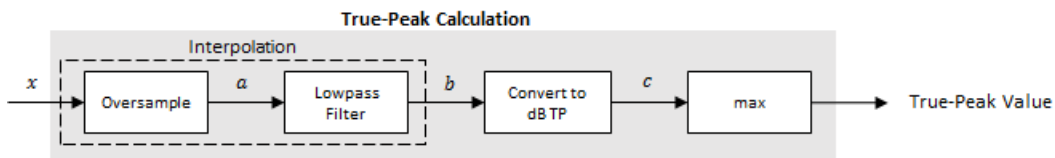
$$k = \{ j \mid sL_j \geq K \}$$

The relative threshold,  $K$ , is recomputed during each call to your `LoudnessMeter` object. The cached values of  $sL_j$  are gated again depending on the updated value of  $K$ .

- 5 The short-term loudness subset,  $sL_k$ , is sorted. The loudness range is calculated as between the 10th and 95th percentiles of the distribution and is returned in loudness units (LU).

## True-Peak

The *true-peak* measurement considers only the current input frame of a call to your loudness meter.



- 1 The signal is oversampled to at least 192 kHz. To optimize processing, the input sample rate determines the exact oversampling. An input sample rate below 750 Hz is not considered.

Input Sample Rate (kHz)	Upsample Factor
[0.75, 1.5)	256
[1.5, 3)	128
[3, 6)	64
[6, 12)	32
[12, 24)	16
[24, 48)	8
[48, 96)	4
[96, 192)	2
[192, ∞)	Not required

- 2 The oversampled signal,  $a$ , passes through a lowpass filter with a half-polyphase length of 12 and stopband attenuation of 80 dB. The filter design uses `designMultirateFIR`.
- 3 The filtered signal,  $b$ , is rectified and converted to the dB TP scale:  
$$c = 20 \times \log_{10}(|b|)$$
- 4 The true-peak is determined as the maximum of the converted signal,  $c$ .

## References

- [1] International Telecommunication Union; Radiocommunication Sector. *Algorithms to Measure Audio Programme Loudness and True-Peak Audio Level*. ITU-R BS.1770-4. 2015.
- [2] European Broadcasting Union. *Loudness Normalisation and Permitted Maximum Level of Audio Signals*. EBU R 128. 2014.
- [3] European Broadcasting Union. *Loudness Metering: 'EBU Mode' Metering to Supplement EBU R 128 Loudness Normalization*. EBU R 128 Tech 3341. 2014.
- [4] European Broadcasting Union. *Loudness Range: A Measure to Supplement EBU R 128 Loudness Normalization*. EBU R 128 Tech 3342. 2016.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

Supports MATLAB Function block: No

Dynamic Memory Allocation must not be turned off.

## See Also

### Blocks

Loudness Meter

### Functions

`integratedLoudness` | `octaveFilter` | `weightingFilter`

**Introduced in R2016b**

# visualize

Open 'EBU Mode' meter display

## Syntax

```
visualize(loudMtr)
```

## Description

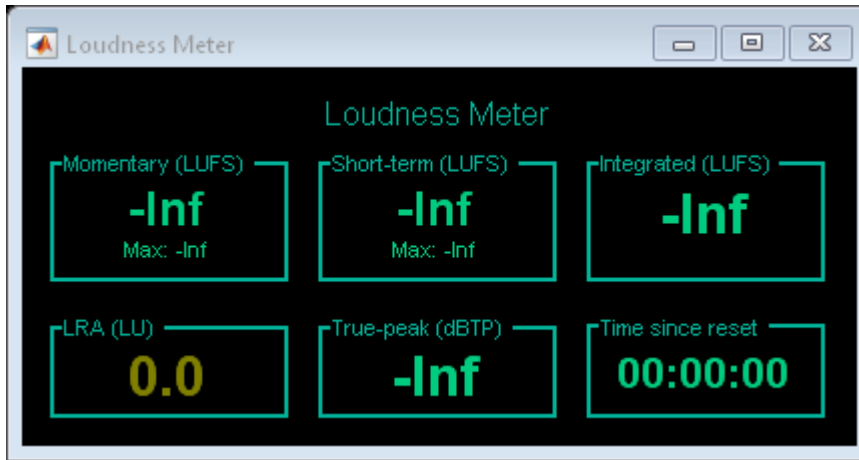
`visualize(loudMtr)` opens an 'EBU Mode' loudness meter display. The values of momentary loudness, short-term loudness, integrated loudness, loudness range, and true-peak are updated as the simulation progresses. The display also shows the maximum value of momentary and short-term loudness, and the time since the last call to `reset`.

## Examples

### Open an 'EBU Mode' Loudness Meter Display

Create an object of the `LoudnessMeter` System object™, and then call `visualize` to open an 'EBU Mode' loudness meter display.

```
loudMtr = LoudnessMeter;  
visualize(loudMtr)
```

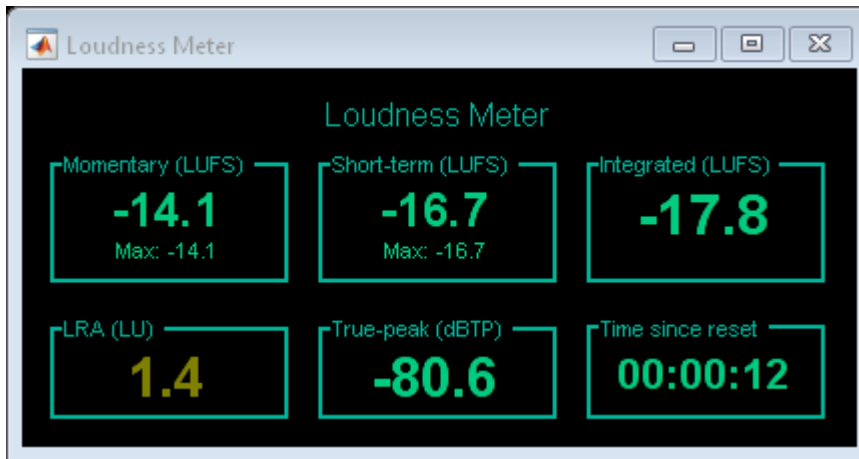


Create an audio file reader System object and specify the audio file to analyze. Create an audio device writer System object to play the audio to your output device.

```
fileReader = dsp.AudioFileReader('RockDrums-48-stereo-11secs.mp3');  
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

In an audio stream loop, read the audio from the file and play it to your device. The loudness meter visualization updates at each call.

```
while ~isDone(fileReader)  
    audioIn = fileReader();  
    loudMtr(audioIn);  
    deviceWriter(audioIn);  
end
```



## Input Arguments

### LoudMtr — Object of LoudnessMeter

object

Object of the loudnessMeter System object.

## See Also

### Blocks

Loudness Meter

### Functions

integratedLoudness

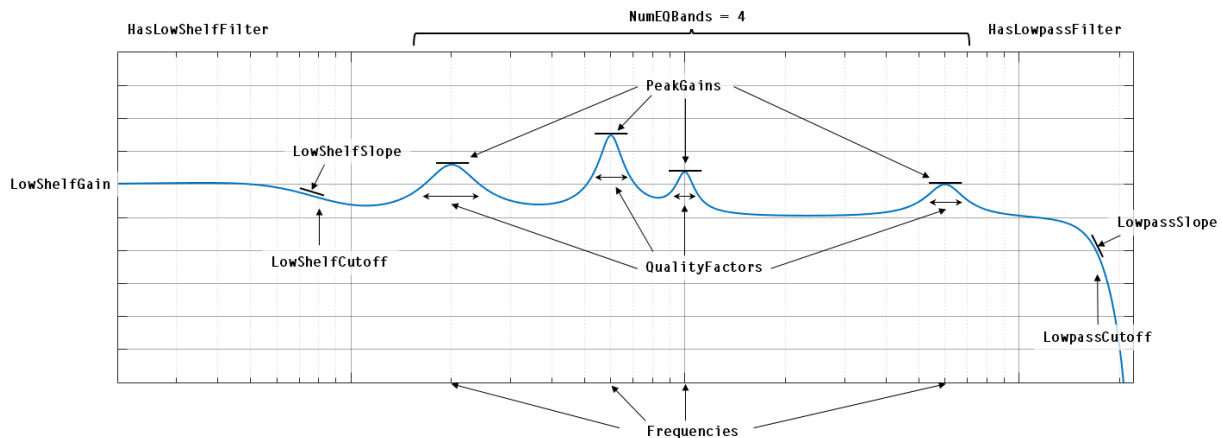
**Introduced in R2016b**

# multibandParametricEQ

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 Create the `multibandParametricEQ` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).



## Creation

## Syntax

```
mPEQ = multibandParametricEQ  
mPEQ = multibandParametricEQ(Name,Value)
```

## Description

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 creation 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 creation of the System object, the first property you specify locks the value.

---

## Creation Arguments

Creation arguments are properties which are set during creation of the System object and cannot be modified later. If you do not explicitly set a creation argument value, the property takes a default value.

### **NumEQBands — Number of equalizer bands**

3 (default) | integer in the range [1, 10]

Number of equalizer bands, specified as an integer in the range [1, 10]. The number of equalizer bands does not include shelving filters, highpass filters, or lowpass filters.

NumEQBands is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: `mPEQ = multibandParametricEQ('NumEQBands',5)` creates a multiband parametric equalizer with 5 bands.

Data Types: `single` | `double`

### **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` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: `mPEQ = multibandParametricEQ('EQOrder',6)` creates a multiband parametric equalizer with the default 3 bands, all of order 6.

Data Types: `single` | `double`

### **HasLowShelfFilter — Low-shelf filter toggle**

`false` (default) | `true`

Low-shelf filter toggle, specified as `false` or `true`.

- `false` -- Do not enable low-shelf filtering in multiband parametric equalizer implementation.
- `true` -- Enable low-shelf filtering in multiband parametric equalizer implementation.

`HasLowShelfFilter` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: `mPEQ = multibandParametricEQ('HasLowShelfFilter',true)` creates a default multiband parametric equalizer with low-shelf filtering enabled.

Data Types: `logical`

### **HasHighShelfFilter — High-shelf filter toggle**

`false` (default) | `true`

High-shelf filter toggle, specified as `false` or `true`.

- `false` -- Do not enable high-shelf filtering in multiband parametric equalizer implementation.

- `true` -- Enable high-shelf filtering in multiband parametric equalizer implementation.

`HasHighShelfFilter` is set during creation of the `System` object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: `mPEQ = multibandParametricEQ('HasHighShelfFilter',true)` creates a default multiband parametric equalizer with high-shelf filtering enabled.

Data Types: `logical`

### **HasLowpassFilter — Lowpass filter toggle**

`false` (default) | `true`

Lowpass filter toggle, specified as `false` or `true`.

- `false` -- Do not enable lowpass filtering in multiband parametric equalizer implementation.
- `true` -- Enable lowpass filtering in multiband parametric equalizer implementation.

`HasLowpassFilter` is set during creation of the `System` object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: `mPEQ = multibandParametricEQ('HasLowpassFilter',true)` creates a default multiband parametric equalizer with lowpass filtering enabled.

Data Types: `logical`

### **HasHighpassFilter — Highpass filter toggle**

`false` (default) | `true`

Highpass filter toggle, specified as `false` or `true`.

- `false` -- Do not enable highpass filtering in multiband parametric equalizer implementation.
- `true` -- Enable highpass filtering in multiband parametric equalizer implementation.

`HasHighpassFilter` is set during creation of the `System` object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: `mPEQ = multibandParametricEQ('HasHighpassFilter',true)` creates a default multiband parametric equalizer with highpass filtering enabled.

Data Types: `logical`

#### **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` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: `mPEQ = multibandParametricEQ('Oversample',true)` creates a default multiband parametric equalizer with oversampling enabled.

Data Types: `logical`

## **Properties**

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects* (MATLAB).

#### **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`.

**Tunable:** Yes

Data Types: `single` | `double`

**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, 700]. Any values outside the range are saturated.

**Tunable:** Yes

Data Types: `single` | `double`

**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 [-inf, 20]. Values above 20 are saturated.

**Tunable:** Yes

Data Types: `single` | `double`

**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.

**Tunable:** Yes

**Dependencies**

To enable this property, set HasLowShelfFilter to `true` during creation.

Data Types: `single` | `double`

**LowShelfSlope — Low-shelf filter slope coefficient**

1.5 (default) | real scalar in the range [0.1, 5]

Low-shelf filter slope coefficient, specified as a real scalar in the range [0.1, 5]. Values outside the range are saturated.

**Tunable:** Yes

### Dependencies

To enable this property, set `HasLowShelfFilter` to `true` during creation.

Data Types: `single` | `double`

### LowShelfGain — Low-shelf filter gain (dB)

0 (default) | real scalar in the range [-12, 12]

Low-shelf filter gain in dB, specified as a real scalar in the range [-12, 12]. Values outside the range are saturated.

**Tunable:** Yes

### Dependencies

To enable this property, set `HasLowShelfFilter` to `true` during creation.

Data Types: `single` | `double`

### 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.

**Tunable:** Yes

### Dependencies

To enable this property, set `HasHighShelfFilter` to `true` during creation.

Data Types: `single` | `double`

#### HighShelfSlope — High-shelf slope coefficient

1.5 (default) | real scalar in the range [0.1, 5]

High-shelf filter slope coefficient, specified as a real scalar in the range [0.1, 5]. Values outside the range are saturated.

**Tunable:** Yes

### Dependencies

To enable this property, set `HasHighShelfFilter` to `true` during creation.

Data Types: `single` | `double`

**HighShelfGain — High-shelf filter gain (dB)**

0 (default) | real scalar in the range [-12, 12]

High-shelf filter gain in dB, specified as a real scalar in the range [-12, 12]. Values outside the range are saturated.

**Tunable:** Yes

**Dependencies**

To enable this property, set `HasHighShelfFilter` to `true` during creation.

Data Types: `single` | `double`

**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.

**Tunable:** Yes

**Dependencies**

To enable this property, set `HasLowpassFilter` to `true` during creation.

Data Types: `single` | `double`

**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 to the nearest multiple of 6.

**Tunable:** Yes

**Dependencies**

To enable this property, set `HasLowpassFilter` to `true` during creation.

Data Types: `single` | `double`

### 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.

**Tunable:** Yes

#### **Dependencies**

To enable this property, set `HasHighpassFilter` to `true` during creation.

Data Types: `single` | `double`

#### **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 to the nearest multiple of 6.

**Tunable:** Yes

#### **Dependencies**

To enable this property, set `HasHighpassFilter` to `true` during creation.

Data Types: `single` | `double`

### Sampling

#### **SampleRate — Input sample rate (Hz)**

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: `single` | `double`



## Usage

## Syntax

```
audioOut = mPEQ(audioIn)
```

## Description

`audioOut = mPEQ(audioIn)` performs multiband parametric equalization on the input signal, `audioIn`, and returns the filtered signal, `audioOut`. The type of equalization is specified by the algorithm and properties of the `multibandParametricEQ` System object, `mPEQ`.

## Input Arguments

### **audioIn** — Audio input to equalizer

matrix

Audio input to the equalizer, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

## Output Arguments

### **audioOut** — Audio output from equalizer

matrix

Audio output from the equalizer, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to multibandParametricEQ

createAudioPluginClass	Create audio plugin class that implements functionality of System object
visualize	Visualize magnitude response of multiband parametric equalizer
parameterTuner	Tune object parameters while streaming

### MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

### Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The createAudioPluginClass and configureMIDI functions map tunable properties of the multibandParametricEQ System object to user-facing parameters:

Property	Range	Mapping	Unit
Frequencies	[20, 20000]	log	Hz
QualityFactors	[0.2, 700]	linear	none
PeakGains	[-50, 20]	linear	dB
LowShelfCutoff	[20, 20000]	log	Hz
LowShelfSlope	[0.1, 5]	linear	none
LowShelfGain	[-12, 12]	linear	dB
HighShelfCutoff	[20, 20000]	log	Hz
HighShelfSlope	[0.1, 5]	linear	none
HighShelfGain	[-12, 12]	linear	dB
LowpassCutoff	[20, 20000]	log	Hz

Property	Range	Mapping	Unit
LowpassSlope	[0, 48]	linear	dB/octave
HighpassCutoff	[20, 20000]	log	Hz
HighpassSlope	[0, 48]	linear	dB/octave

## Examples

### Multiband Parametric Equalization

Create `dsp.AudioFileReader` and `audioDeviceWriter` 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);

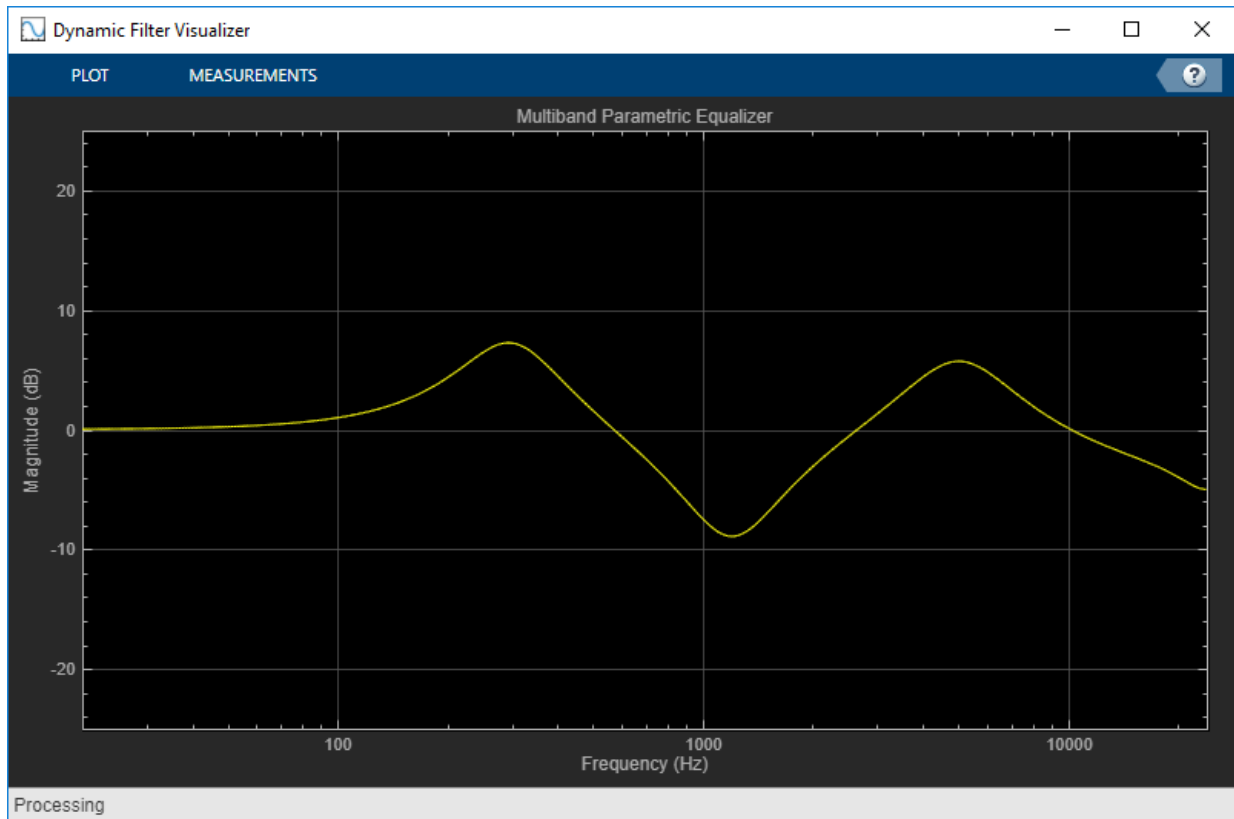
setup(deviceWriter,ones(frameLength,2))
```

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);
```

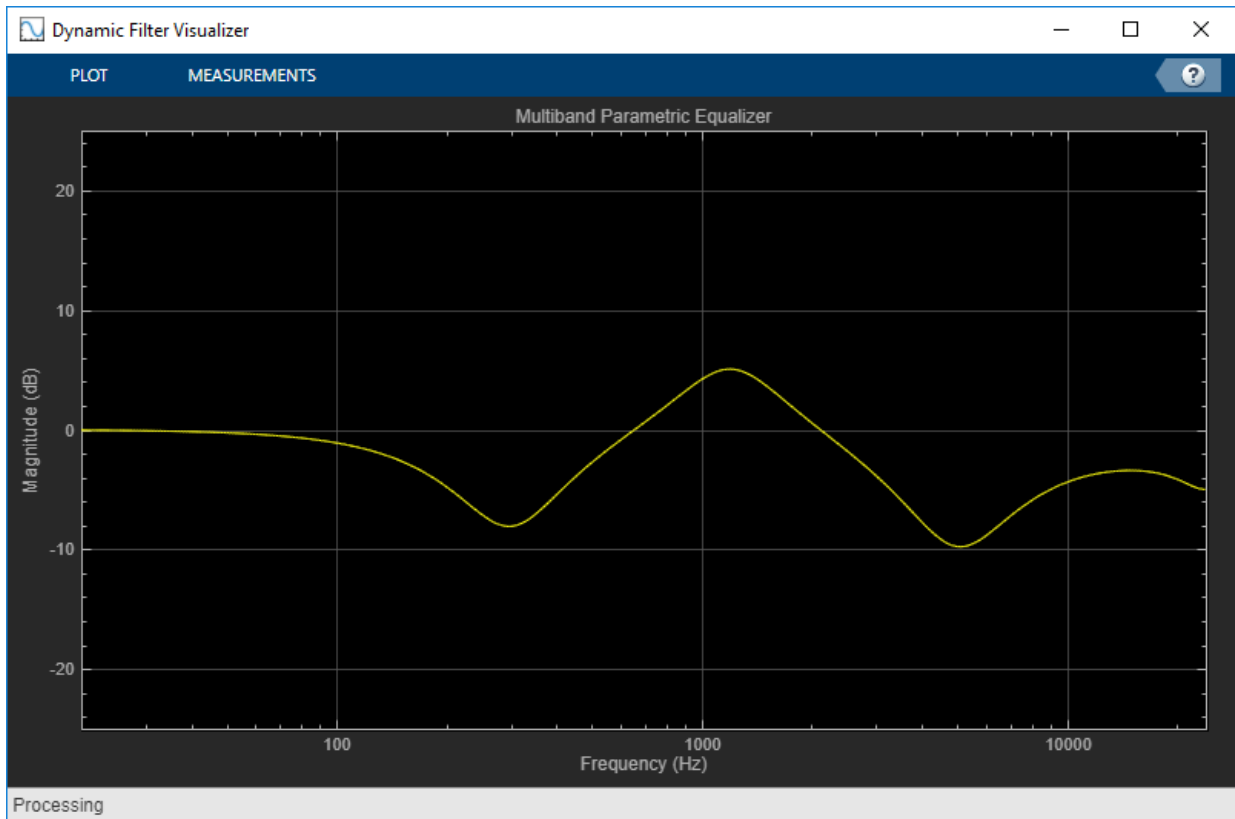
Visualize the magnitude frequency response of your multiband parametric equalizer.

```
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 = fileReader();
    equalizedSignal = mPEQ(originalSignal);
    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;
end
```



```
release(fileReader)
release(mPEQ)
release(deviceWriter)
```

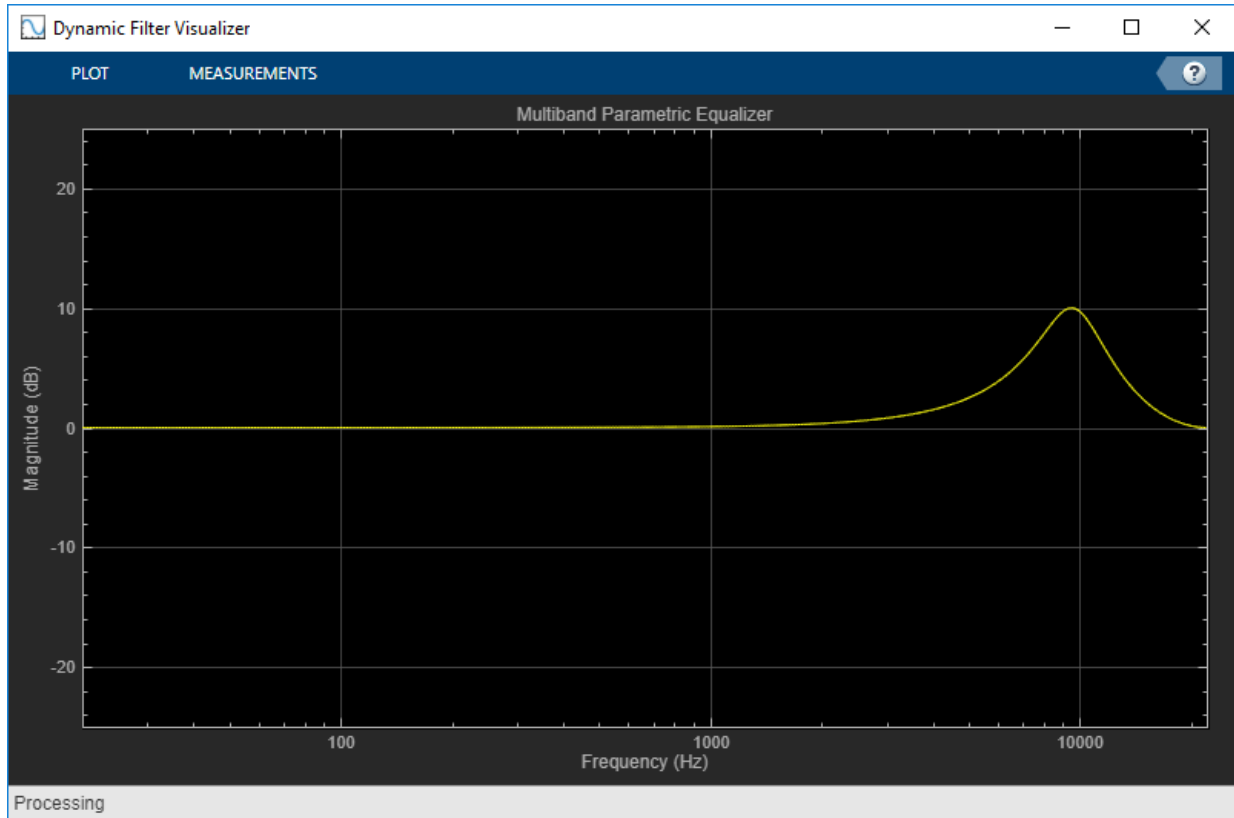
### Oversample Audio Signal

Reduce warping by specifying your `multibandParametricEQ` object to perform oversampling before equalization.

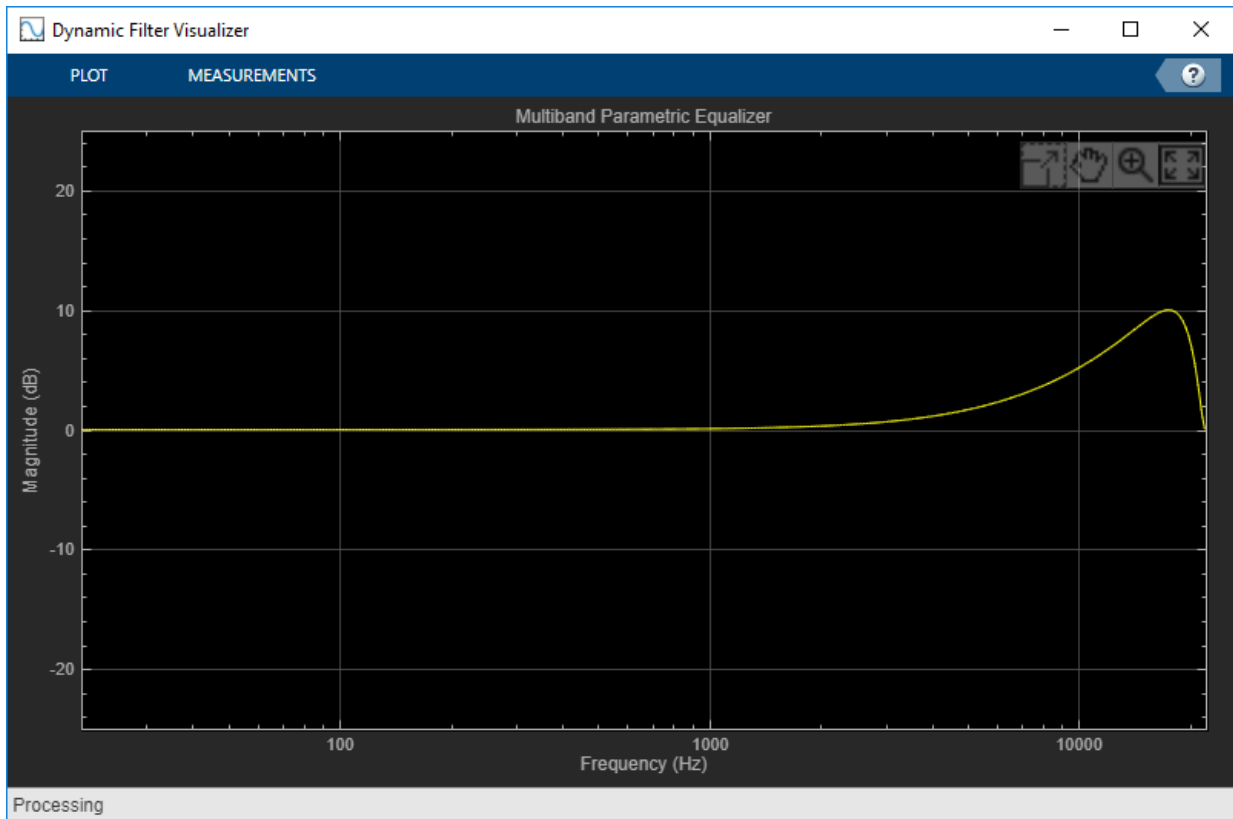
Create a one-band equalizer. Visualize the equalizer band as its center frequency approaches the Nyquist rate.

### 3 System objects in Audio Toolbox

```
mPEQ = multibandParametricEQ( ...  
    'NumEQBands',1, ...  
    'Frequencies',9.5e3, ...  
    'PeakGains',10);  
visualize(mPEQ)
```



```
for i = 1:1000  
    mPEQ.Frequencies = mPEQ.Frequencies + 8;  
end
```

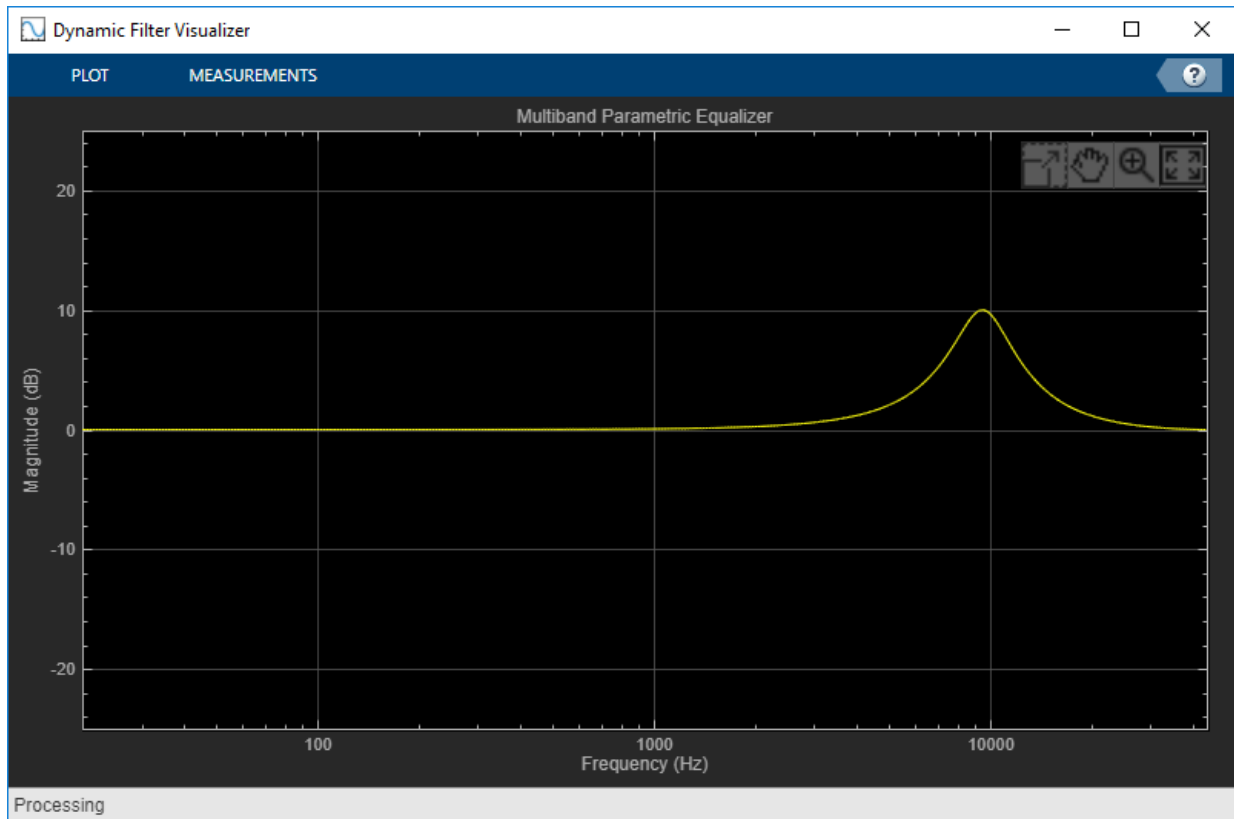


The equalizer band is warped.

Create a one-band equalizer with `Oversample` set to `true`. Visualize the equalizer band as its center frequency approaches the Nyquist rate.

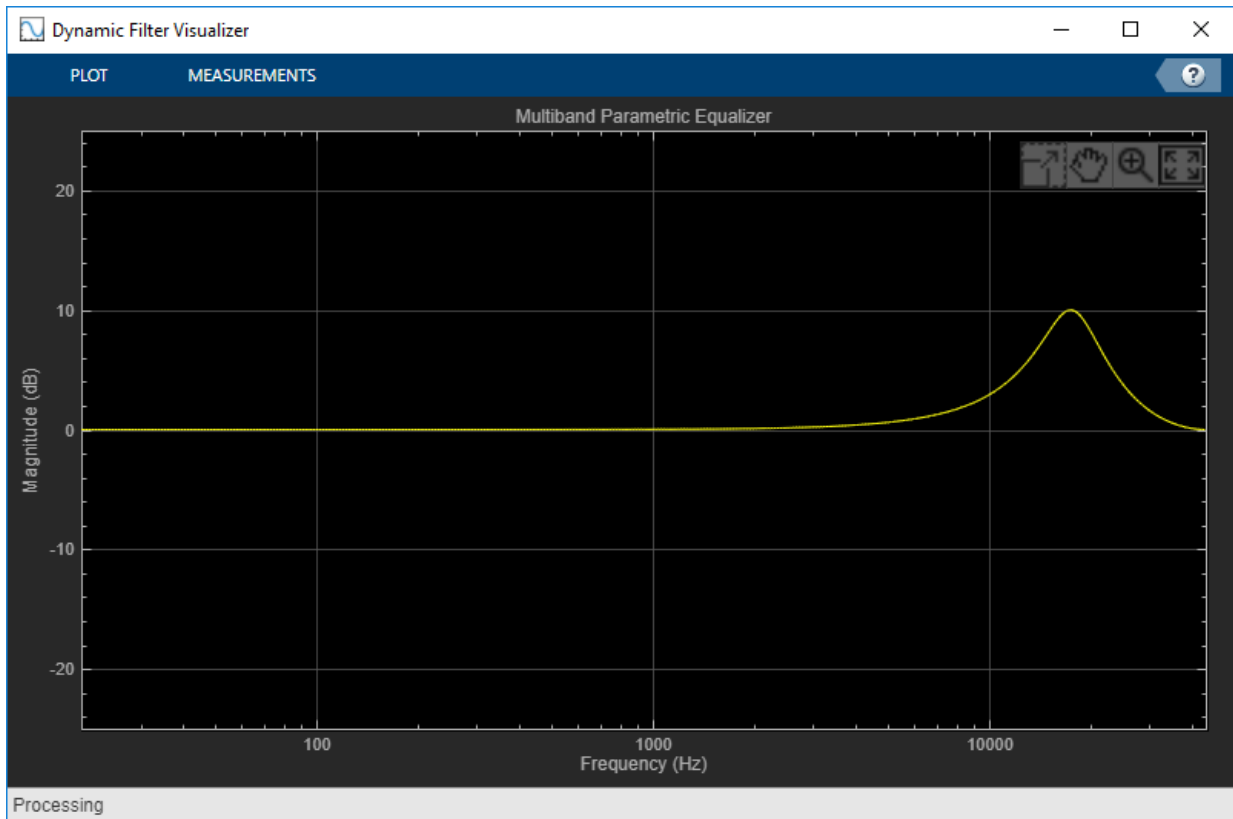
```
mPEQoversampled = multibandParametricEQ( ...
    'NumEQBands',1, ...
    'Frequencies',9.5e3, ...
    'PeakGains',10, ...
    'Oversample',true);
visualize(mPEQoversampled)
```

### 3 System objects in Audio Toolbox



```
for i = 1:1000
    mPEQoversampled.Frequencies = mPEQoversampled.Frequencies + 8;
end
```





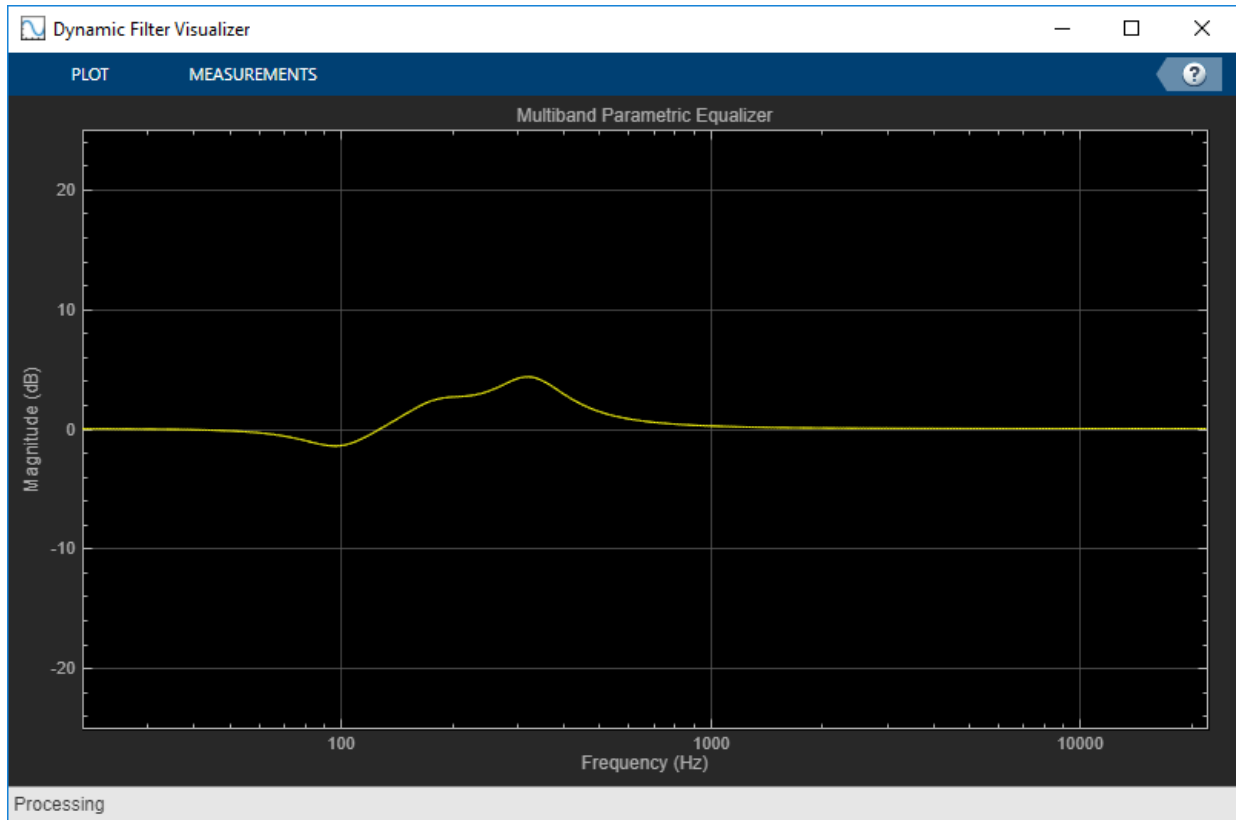
Warping is reduced.

### Tune Multiband Parametric EQ Parameters

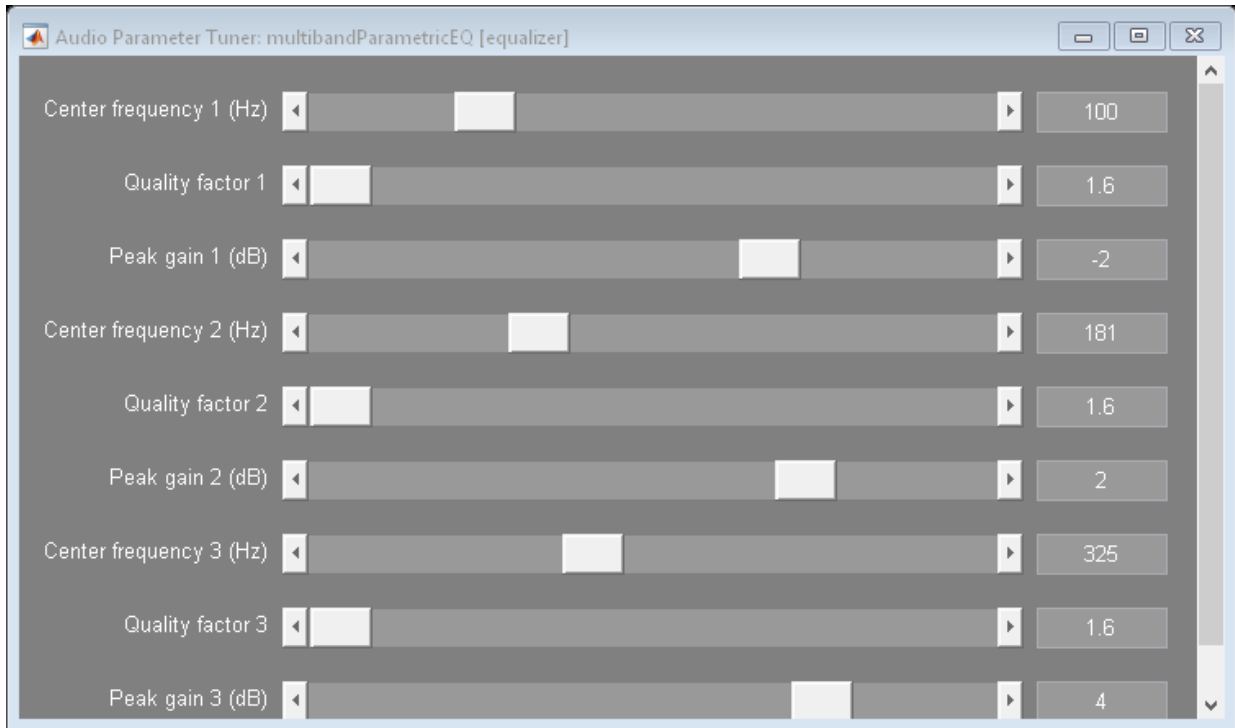
Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create a `audioDeviceWriter` to write audio to your sound card. Create a `multibandParametricEQ` to process the audio data. Call `visualize` to plot the frequency response of the equalizer.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3', 'SamplesPerFrame', ...
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

```
equalizer = multibandParametricEQ('SampleRate',fileReader.SampleRate, 'PeakGains',[-2,2  
visualize(equalizer)
```



Call `parameterTuner` to open a UI to tune parameters of the equalizer while streaming.  
`parameterTuner(equalizer)`



In an audio stream loop:

- 1 Read in a frame of audio from the file.
- 2 Apply equalization.
- 3 Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the equalizer and listen to the effect.

```
while ~isDone(fileReader)
  audioIn = fileReader();
  audioOut = equalizer(audioIn);
  deviceWriter(audioOut);
  drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(equalizer)
```

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

[Parametric EQ](#) | [designParamEQ](#) | [designShelvingEQ](#) | [designVarSlopeFilter](#)

### Topics

“Parametric Equalizer Design”

“Equalization”

**Introduced in R2016a**

# visualize

Visualize magnitude response of multiband parametric equalizer

## Syntax

```
visualize(mPEQ)  
visualize(mPEQ,NFFT)
```

## Description

`visualize(mPEQ)` plots the magnitude response of the `multibandParametricEQ` object, `mPEQ`. The plot is updated automatically when properties of the object change.

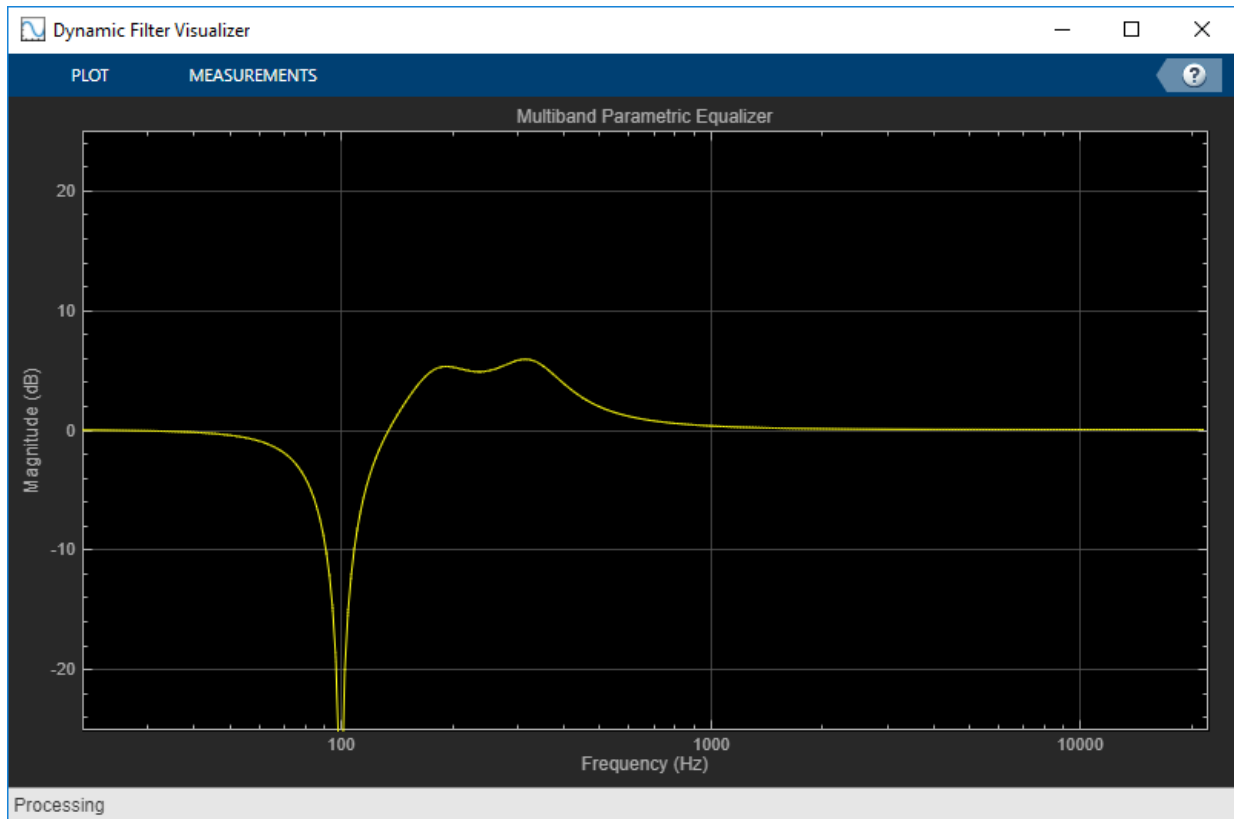
`visualize(mPEQ,NFFT)` specifies an N-point FFT used to calculate the magnitude response.

## Examples

### Specify a Nondefault Number of FFT Points

Create an object of the `multibandParametricEQ` System object™, and then call `visualize` to plot the magnitude response using a 5096-point FFT.

```
mPEQ = multibandParametricEQ('PeakGains',[-inf,5,5]);  
visualize(mPEQ,5096)
```



## Input Arguments

**mPEQ** — Multiband parametric equalizer to visualize  
object of `multibandParametricEQ` System object

Multiband parametric equalizer whose magnitude response you want to plot.

**NFFT** — N-point FFT  
2048 (default) | positive scalar

Number of bins used to calculate the DFT, specified as a positive scalar.

Data Types: single | double

## **See Also**

`multibandParametricEQ`

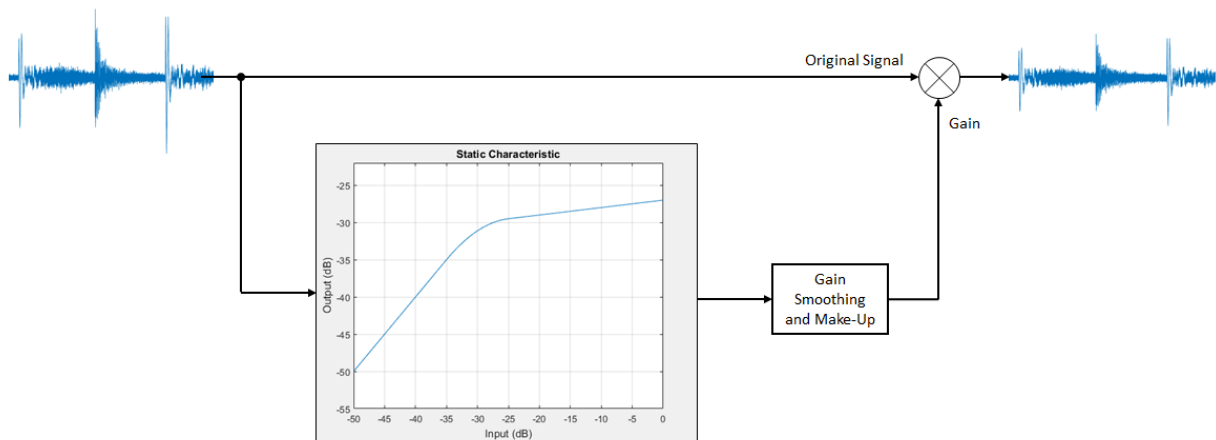
**Introduced in R2016a**

## compressor

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:

- 1 Create the `compressor` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).



## Creation

## Syntax

```
dRC = compressor
dRC = compressor(thresholdValue)
dRC = compressor(thresholdValue, ratioValue)
dRC = compressor( ____, Name, Value)
```

## Description

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 a 10 ms attack time operating at a 16 kHz sample rate.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the release function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

### 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.

**Tunable:** Yes

Data Types: `single` | `double`

### **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 > Threshold, 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.

**Tunable:** Yes

Data Types: `single` | `double`

### **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.

**Tunable:** Yes

Data Types: `single` | `double`

### **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.

**Tunable:** Yes

Data Types: `single` | `double`

### **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.

**Tunable:** Yes

Data Types: `single` | `double`

### **MakeUpGainMode — Make-up gain mode**

'Property' (default) | 'Auto'

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.

**Tunable:** No

Data Types: char | string

### **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.

**Tunable:** Yes

### **Dependencies**

To enable this property, set `MakeUpGainMode` to 'Property'.

Data Types: single | double

### **SampleRate — Input sample rate (Hz)**

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: single | double

## Usage

## Syntax

```
audioOut = dRC(audioIn)
[audioOut,gain] = dRC(audioIn)
```

## Description

`audioOut = dRC(audioIn)` performs dynamic range compression on the input signal, `audioIn`, and returns the compressed signal, `audioOut`. The type of dynamic range

compression is specified by the algorithm and properties of the compressor System object, `dRC`.

`[audioOut,gain] = dRC(audioIn)` also returns the applied gain, in dB, at each input sample.

## Input Arguments

### **audioIn** — Audio input to compressor

matrix

Audio input to the compressor, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

## Output Arguments

### **audioOut** — Audio output from compressor

matrix

Audio output from the compressor, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

### **gain** — Gain applied by compressor (dB)

matrix

Gain applied by compressor, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to compressor

visualize	Visualize static characteristic of dynamic range controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
parameterTuner	Tune object parameters while streaming

### MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

### Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the compressor System object to user-facing parameters:

Property	Range	Mapping	Unit
Threshold	[-50, 0]	linear	dB
Ratio	[1, 50]	linear	none
KneeWidth	[0, 20]	linear	dB
AttackTime	[0, 4]	linear	seconds
ReleaseTime	[0, 4]	linear	seconds
MakeUpGain (available when you set MakeUpGainMode to 'Property')	[-10, 24]	linear	dB

## Examples

### Compress Audio Signal

Use dynamic range compression to attenuate the volume of loud sounds.

Set up the `dsp.AudioFileReader` and `audioDeviceWriter` 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 dB. Use the sample rate of your audio file reader.

```
dRC = compressor(-15,7, ...
    'KneeWidth',5, ...
    'SampleRate',fileReader.SampleRate);
```

Set up the scope to visualize the original audio signal, the compressed audio signal, and the applied compressor gain.

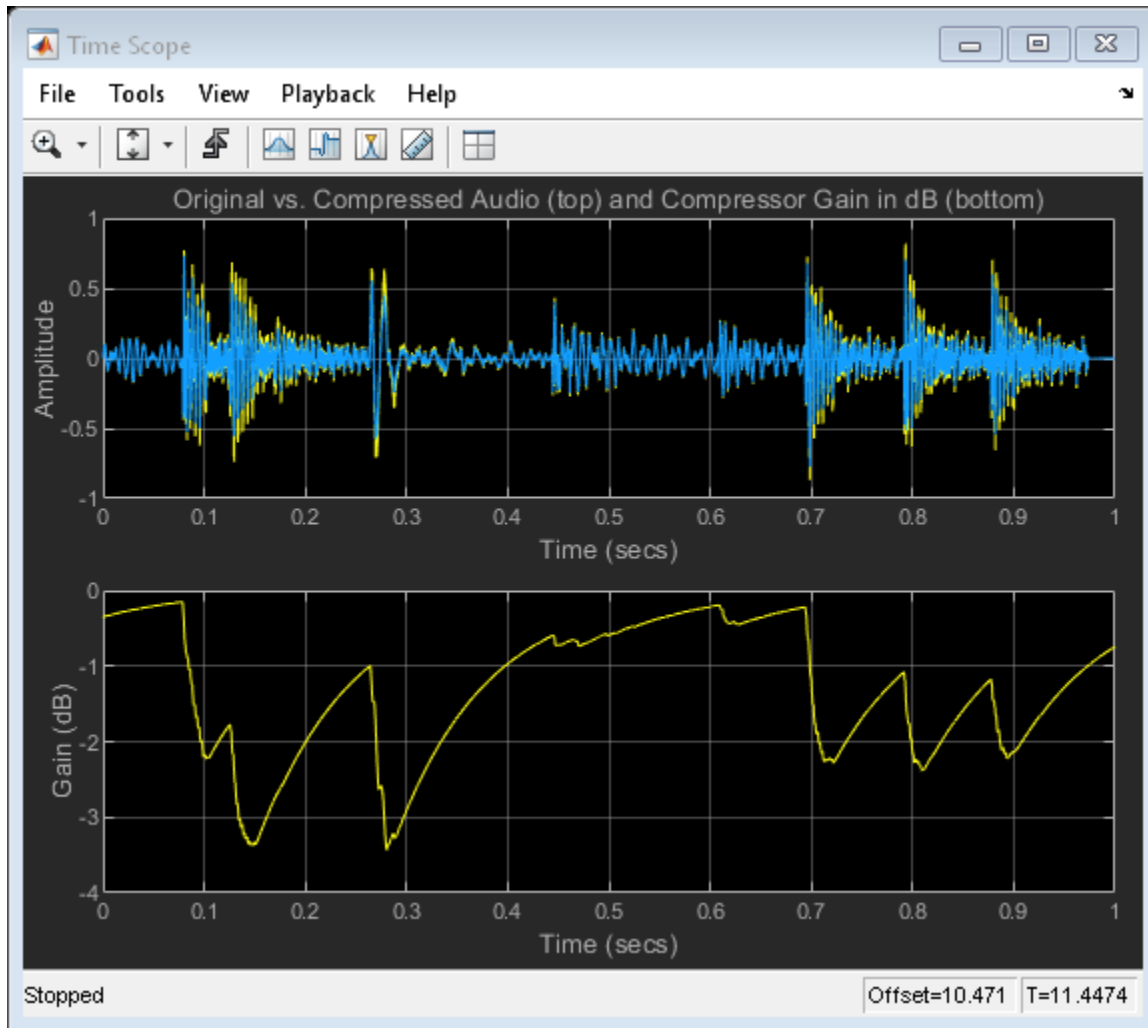
```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpan',1, ...
    'BufferLength',44100*4, ...
    'YLimits',[-1,1], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'LayoutDimensions',[2,1], ...
    'NumInputPorts',2, ...
    'Title', ...
    ['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 = fileReader();
    [y,g] = dRC(x);
    deviceWriter(y);
    scope([x(:,1),y(:,1)],g(:,1))
end

release(dRC)
release(deviceWriter)
release(scope)
```





### Compare Dynamic Range Limiter and Compressor

A dynamic range limiter is a special type of dynamic range compressor. In limiters, the level above an operational threshold is hard limited. In the simplest implementation of a limiter, the effect is equivalent to audio clipping. In compressors, the level above an

operational threshold is lowered using a specified compression ratio. Using a compression ratio results in a smoother processed signal.

### **Compare Limiter and Compressor Applied to Sinusoid**

Create a limiter System object™ and a compressor System object. Set the `AttackTime` and `ReleaseTime` properties of both objects to zero. Create an `audioOscillator` System object to generate a sinusoid with `Frequency` set to 5 and `Amplitude` set to 0.1.

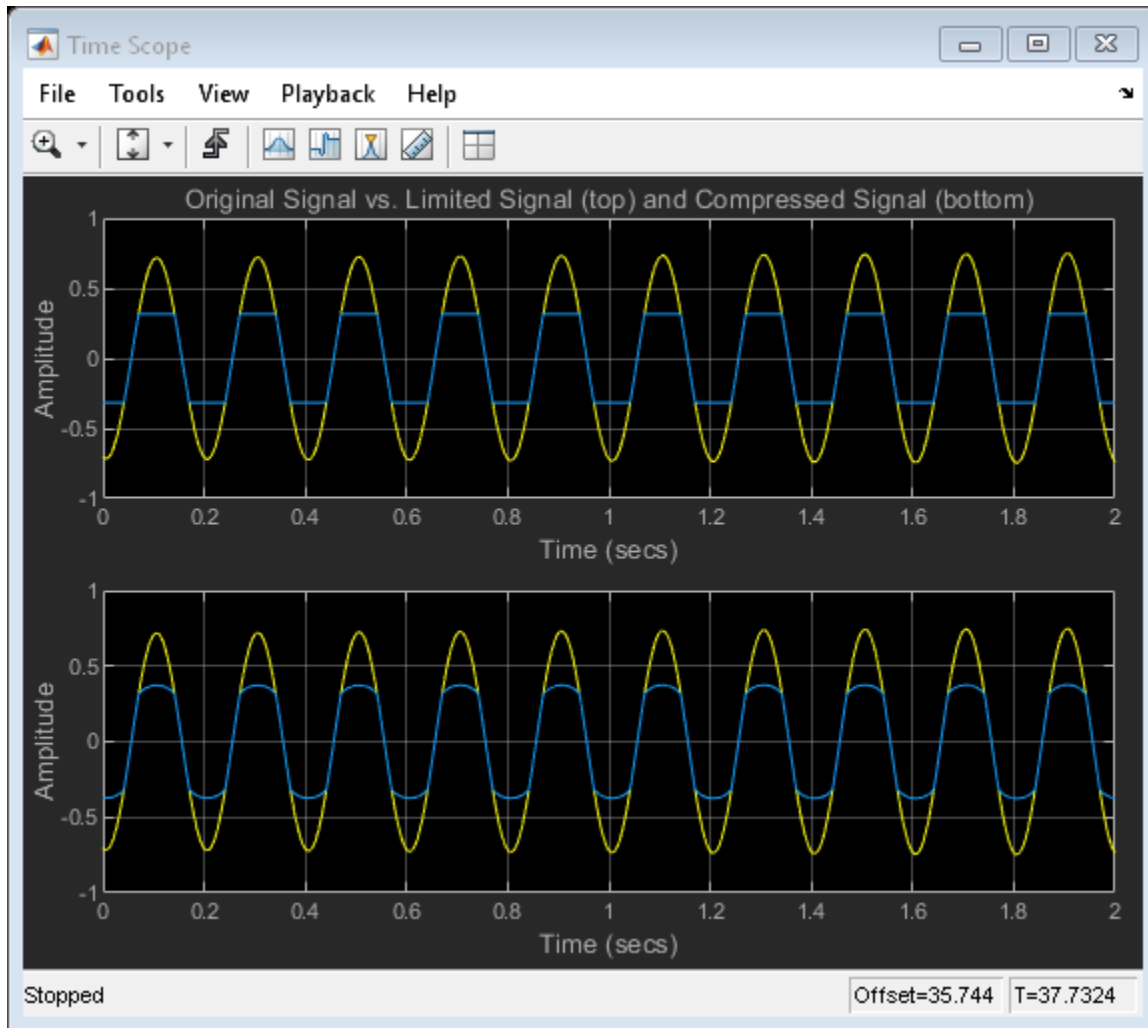
```
dRL = limiter('AttackTime',0,'ReleaseTime',0);  
dRC = compressor('AttackTime',0,'ReleaseTime',0);  
  
osc = audioOscillator('Frequency',5,'Amplitude',0.1);
```

Create a time scope to visualize the generated sinusoid and the processed sinusoid.

```
scope = dsp.TimeScope( ...  
    'SampleRate',osc.SampleRate, ...  
    'TimeSpan',2, ...  
    'BufferLength',osc.SampleRate*4, ...  
    'YLimits',[-1 1], ...  
    'TimeSpanOverrunAction','Scroll', ...  
    'ShowGrid',true, ...  
    'LayoutDimensions',[2 1], ...  
    'NumInputPorts',2, ...  
    'Title', ...  
    'Original Signal vs. Limited Signal (top) and Compressed Signal (bottom)');
```

In an audio stream loop, visualize the original sinusoid and the sinusoid processed by a limiter and a compressor. Increment the amplitude of the original sinusoid to illustrate the effect.

```
while osc.Amplitude < 0.75  
    x = osc();  
  
    xLimited = dRL(x);  
    xCompressed = dRC(x);  
  
    scope([x xLimited],[x xCompressed]);  
  
    osc.Amplitude = osc.Amplitude + 0.0002;  
end  
release(scope)
```



```
release(dRL)
release(dRC)
release(osc)
```

### Compare Limiter and Compressor Applied to Audio Signal

Compare the effect of dynamic range limiters and compressors on a drum track. Create a `dsp.AudioFileReader` System object and a `audioDeviceWriter` System object to

read audio from a file and write to your audio output device. To emphasize the effect of dynamic range control, set the operational threshold of the limiter and compressor to -20 dB.

```
dRL.Threshold = -20;  
dRC.Threshold = -20;
```

```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3');  
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

Read successive frames from an audio file in a loop. Listen to and compare the effect of dynamic range limiting and dynamic range compression on an audio signal.

```
numFrames = 300;
```

```
fprintf('Now playing original signal...\n')
```

```
Now playing original signal...
```

```
for i = 1:numFrames  
    x = fileReader();  
    deviceWriter(x);
```

```
end
```

```
reset(fileReader);
```

```
fprintf('Now playing limited signal...\n')
```

```
Now playing limited signal...
```

```
for i = 1:numFrames  
    x = fileReader();  
    xLimited = dRL(x);  
    deviceWriter(xLimited);
```

```
end
```

```
reset(fileReader);
```

```
fprintf('Now playing compressed signal...\n')
```

```
Now playing compressed signal...
```

```
for i = 1:numFrames  
    x = fileReader();  
    xCompressed = dRC(x);  
    deviceWriter(xCompressed);
```

```
end
```

```

release(fileReader)
release(deviceWriter)
release(dRC)
release(dRL)

```

## Diminish Plosives from Speech Signal

Plosives are consonant sounds resulting from a sudden release of airflow. They are most pronounced in words beginning with *p*, *d*, and *g* sounds. Plosives can be emphasized by the recording process and are often displeasurable to hear. In this example, you minimize the plosives of a speech signal by applying highpass filtering and low-band compression.

Create a `dsp.AudioFileReader` object and a `audioDeviceWriter` object to read an audio signal from a file and write an audio signal to a device. Play the unprocessed signal. Then release the file reader and device writer.

```

fileReader = dsp.AudioFileReader('audioPlosives.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

while ~isDone(fileReader)
    audioIn = fileReader();
    deviceWriter(audioIn);
end
release(deviceWriter)
release(fileReader)

```

Design a highpass filter with a steep rolloff of all frequencies below 120 Hz. Use a `dsp.BiquadFilter` object to implement the highpass filter design. Create a crossover filter with one crossover at 250 Hz. The crossover filter enables you to separate the band of interest for processing. Create a dynamic range compressor to compress the dynamic range of plosive sounds. To apply no make-up gain, set the `MakeUpGainMode` to "Property" and use the default 0 dB `MakeUpGain` property value. Create a time scope to visualize the processed and unprocessed audio signal.

```

[B,A] = designVarSlopeFilter(48,120/(fileReader.SampleRate/2),"hi");
biquadFilter = dsp.BiquadFilter( ...
    "SOSMatrixSource","Input port", ...
    "ScaleValuesInputPort",false);

crossFilt = crossoverFilter( ...
    "SampleRate",fileReader.SampleRate, ...
    "NumCrossovers",1, ...

```

```
"CrossoverFrequencies",250, ...
"CrossoverSlopes",48);

dRCompressor = compressor( ...
    "Threshold",-35, ...
    "Ratio",10, ...
    "KneeWidth",20, ...
    "AttackTime",1e-4, ...
    "ReleaseTime",3e-1, ...
    "MakeUpGainMode","Property", ...
    "SampleRate",fileReader.SampleRate);

scope = dsp.TimeScope( ...
    "SampleRate",fileReader.SampleRate, ...
    "TimeSpan",3, ...
    "BufferLength",fileReader.SampleRate*3*2, ...
    "YLimits",[-1 1], ...
    "ShowGrid",true, ...
    "ShowLegend",true, ...
    "ChannelNames",{ 'Original', 'Processed' });
```

In an audio stream loop:

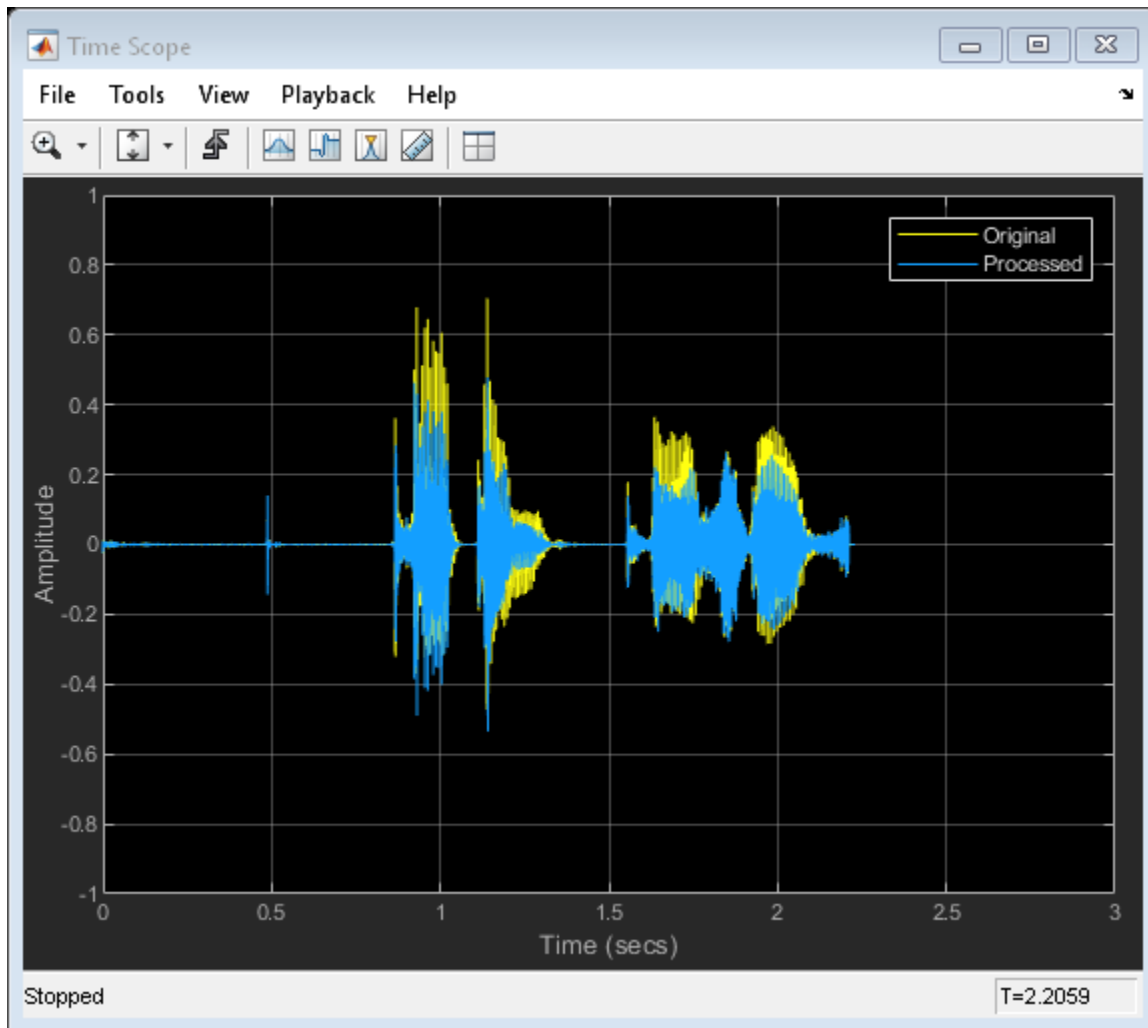
- 1 Read in a frame of the audio file.
- 2 Apply highpass filtering using your biquad filter.
- 3 Split the audio signal into two bands.
- 4 Apply dynamic range compression to the lower band.
- 5 Remix the channels.
- 6 Write the processed audio signal to your audio device for listening.
- 7 Visualize the processed and unprocessed signals on a time scope.

As a best practice, release your objects once done.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioIn = biquadFilter(audioIn,B,A);
    [band1,band2] = crossFilt(audioIn);
    band1compressed = dRCompressor(band1);
    audioOut = band1compressed + band2;
    deviceWriter(audioOut);
    scope([audioIn audioOut])
end
```

As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(crossFilt)
release(dRCompressor)
release(scope)
```



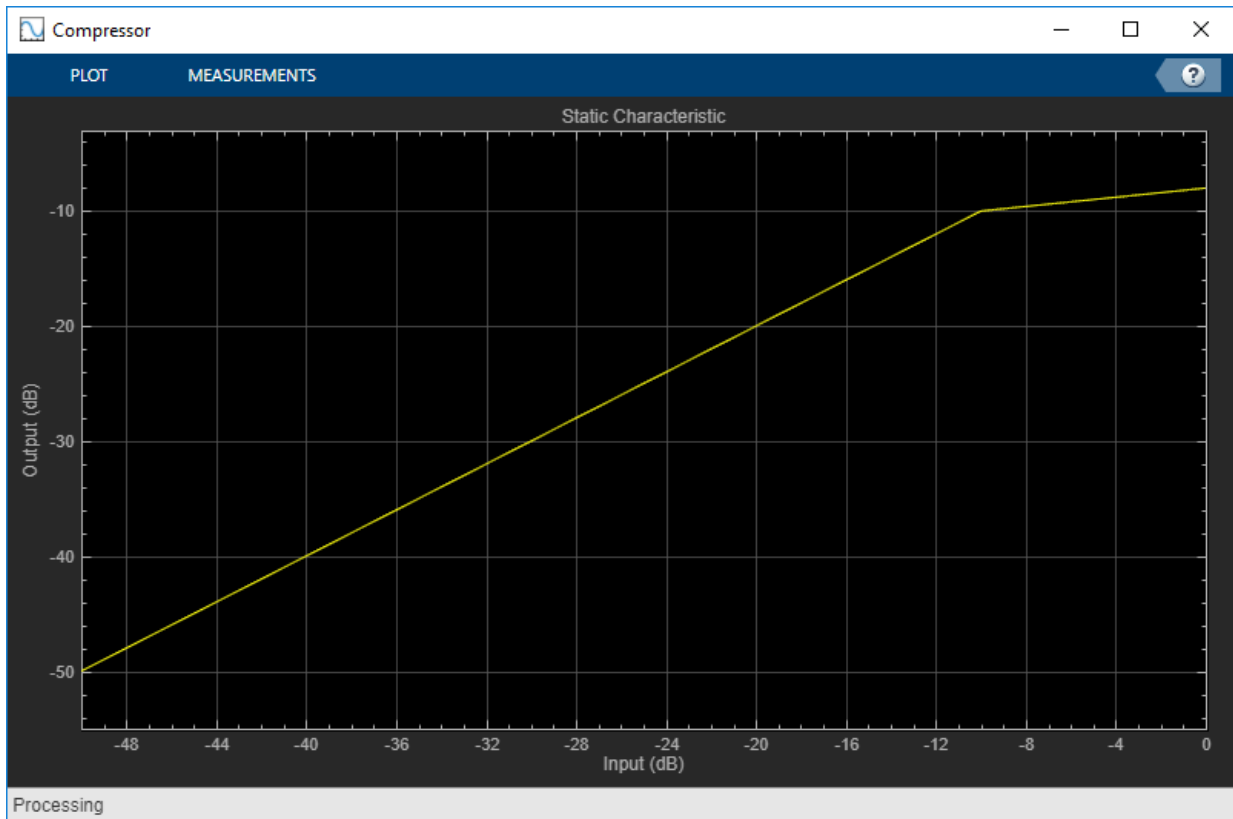
### Tune Compressor Parameters

Create an `dsp.AudioFileReader` to read in audio frame-by-frame. Create a `audioDeviceWriter` to write audio to your sound card. Create a `compressor` to process the audio data. Call `visualize` to plot the static characteristic of the compressor.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

dRC = compressor('SampleRate',fileReader.SampleRate);
visualize(dRC)
```



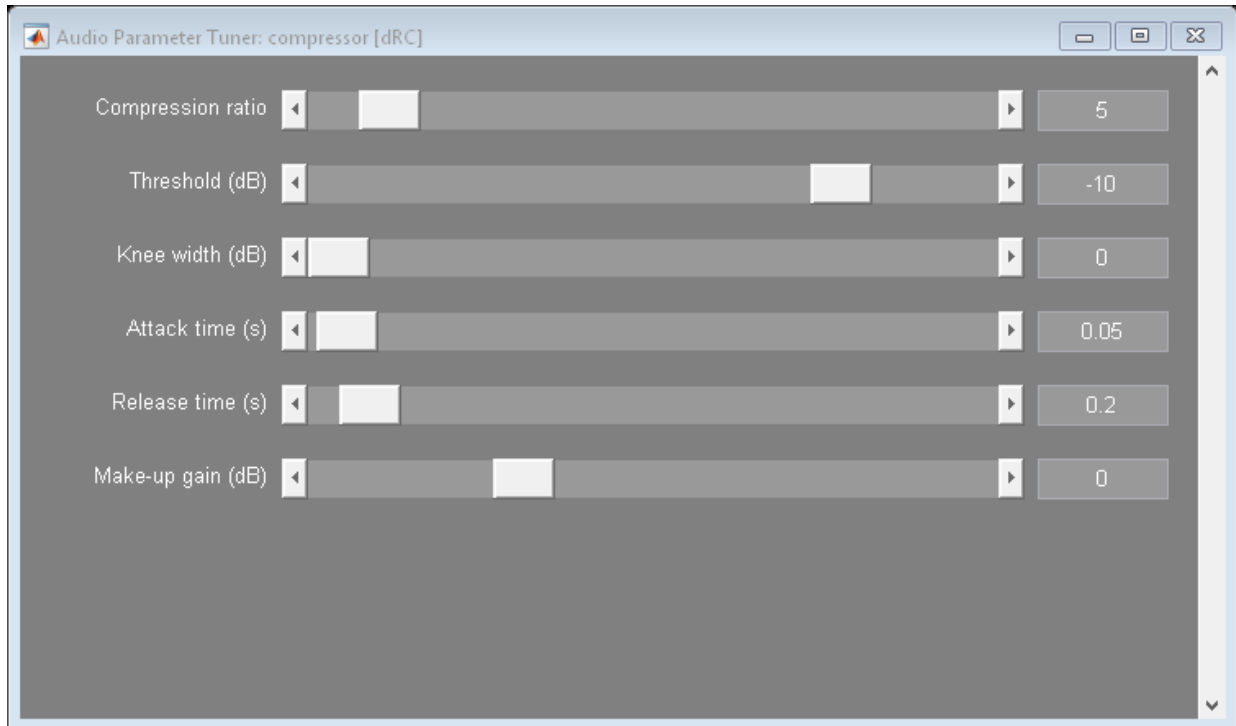


Create a `dsp.TimeScope` to visualize the original and processed audio.

```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpan',1, ...
    'BufferLength',fileReader.SampleRate*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)';
```

Call `parameterTuner` to open a UI to tune parameters of the compressor while streaming.

```
parameterTuner(dRC)
```



In an audio stream loop:

- 1 Read in a frame of audio from the file.
- 2 Apply dynamic range compression.
- 3 Write the frame of audio to your audio device for listening.
- 4 Visualize the original audio, the processed audio, and the gain applied.

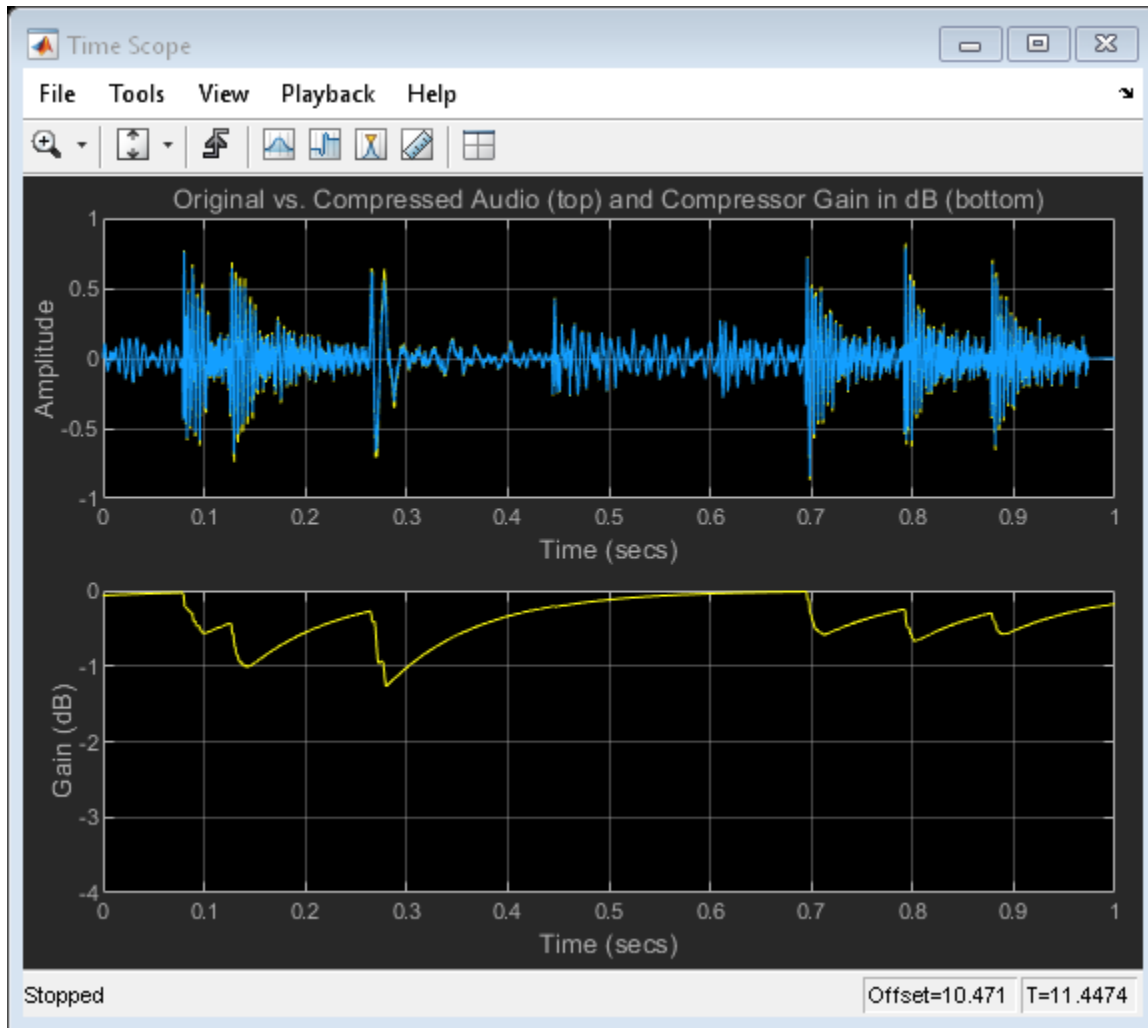
While streaming, tune parameters of the dynamic range compressor and listen to the effect.

```
while ~isDone(fileReader)  
    audioIn = fileReader();
```

```
[audioOut,g] = dRC(audioIn);  
deviceWriter(audioOut);  
scope([audioIn(:,1),audioOut(:,1)],g(:,1));  
drawnow limitrate % required to update parameter  
end
```

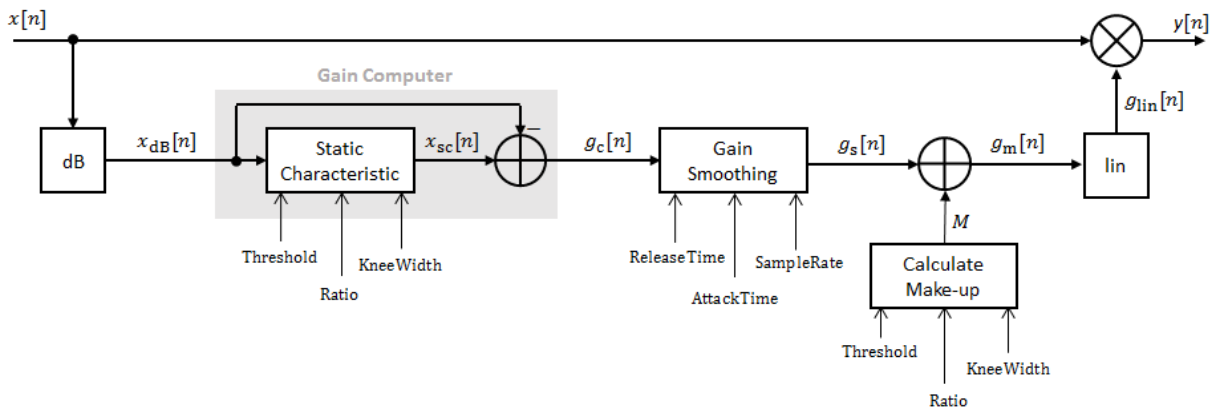
As a best practice, release your objects once done.

```
release(deviceWriter)  
release(fileReader)  
release(dRC)  
release(scope)
```



## Algorithms

The compressor System object processes a signal frame by frame and element by element.



## Convert Input Signal to dB

The  $N$ -point signal,  $x[n]$ , is converted to decibels:

$$x_{dB}[n] = 20 \times \log_{10}|x[n]|$$

## Gain Computer

$x_{dB}[n]$  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:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{\left(\frac{1}{R} - 1\right)\left(x_{dB} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \leq x_{dB} \leq \left(T + \frac{W}{2}\right) \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases},$$

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:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < T \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} \geq T \end{cases}$$

The computed gain,  $g_c[n]$ , is calculated as

$$g_c[n] = x_{sc}[n] - x_{dB}[n].$$

### Gain Smoothing

$g_c[n]$  is smoothed using specified attack and release time properties:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) g_c[n], & g_c[n] \leq g_s[n-1] \\ \alpha_R g_s[n-1] + (1 - \alpha_R) g_c[n], & g_c[n] > g_s[n-1] \end{cases}$$

The attack time coefficient,  $\alpha_A$ , is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{F_S \times T_A}\right).$$

The release time coefficient,  $\alpha_R$ , 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.

### Calculate and Apply Make-up Gain

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,

$$M = -x_{sc}|_{x_{dB} = 0}.$$

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.

The make-up gain,  $M$ , is added to the smoothed gain,  $g_s[n]$ :

$$g_m[n] = g_s[n] + M$$

## Calculate and Apply Linear Gain

The calculated gain in dB,  $g_m[n]$ , is translated to a linear domain:

$$g_{lin}[n] = 10\left(\frac{g_m[n]}{20}\right)$$

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, 2012, pp. 399-408.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

Compressor | expander | limiter | noiseGate

### Topics

“Dynamic Range Control”

**Introduced in R2016a**

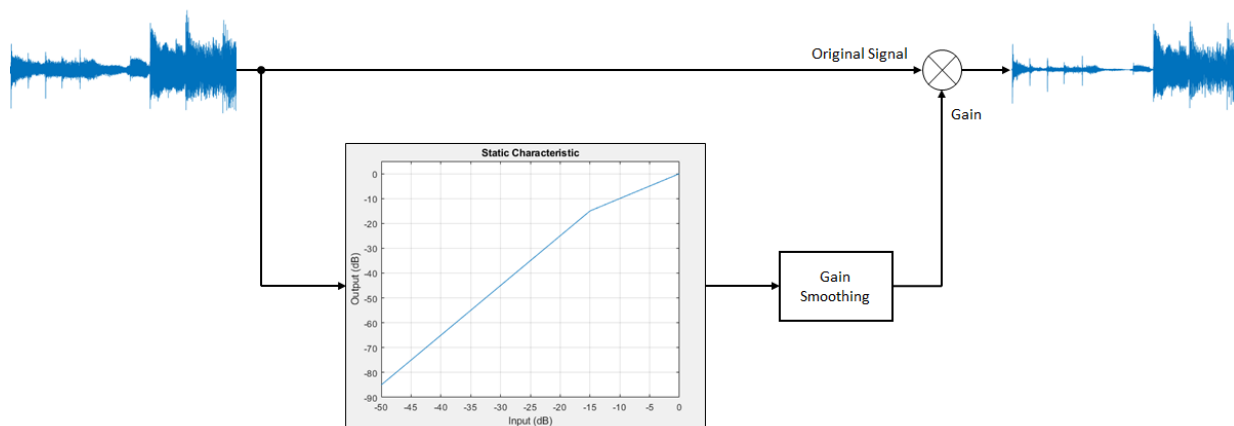


# expander

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:

- 1 Create the `expander` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

### Syntax

```
dRE = expander
dRE = expander(thresholdValue)
dRE = expander(thresholdValue, ratioValue)
dRE = expander( ___, Name, Value)
```

### Description

`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 a 0.01 second attack time and a 16 kHz sample rate.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects (MATLAB)*.

#### **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.

**Tunable:** Yes

Data Types: `single` | `double`

**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.

**Tunable:** Yes

Data Types: `single` | `double`

**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.

**Tunable:** Yes

Data Types: `single` | `double`

### **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.

**Tunable:** Yes

Data Types: `single` | `double`

### **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.

**Tunable:** Yes

Data Types: `single` | `double`

### **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.

**Tunable:** Yes

Data Types: `single` | `double`

**SampleRate — Input sample rate (Hz)**

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: `single` | `double`

## Usage

## Syntax

```
audioOut = dRE(audioIn)  
[audioOut,gain] = dRE(audioIn)
```

## Description

`audioOut = dRE(audioIn)` performs dynamic range expansion on the input signal, `audioIn`, and returns the expanded signal, `audioOut`. The type of dynamic range expansion is specified by the algorithm and properties of the `expander` System object, `dRE`.

`[audioOut,gain] = dRE(audioIn)` also returns the applied gain, in dB, at each input sample.

## Input Arguments

**audioIn — Audio input to expander**

matrix

Audio input to the expander, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

### Output Arguments

#### **audioOut** — Audio output from expander

matrix

Audio output from the expander, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

#### **gain** — Gain applied by expander (dB)

matrix

Gain applied by expander, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

### Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

#### Specific to expander

<code>visualize</code>	Visualize static characteristic of dynamic range controller
<code>createAudioPluginClass</code>	Create audio plugin class that implements functionality of System object
<code>parameterTuner</code>	Tune object parameters while streaming

#### MIDI

<code>configureMIDI</code>	Configure MIDI connections between audio object and MIDI controller
<code>disconnectMIDI</code>	Disconnect MIDI controls from audio object
<code>getMIDIConnections</code>	Get MIDI connections of audio object

#### Common to All System Objects

<code>clone</code>	Create duplicate System object
<code>isLocked</code>	Determine if System object is in use

release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the expander System object to user-facing parameters:

Property	Range	Mapping	Unit
Threshold	[-140, 0]	linear	dB
Ratio	[1, 50]	linear	none
KneeWidth	[0, 20]	linear	dB
AttackTime	[0, 4]	linear	seconds
ReleaseTime	[0, 4]	linear	seconds
HoldTime	[0, 4]	linear	seconds

## Examples

### Expand Audio Signal

Use dynamic range expansion to attenuate background noise from an audio signal.

Set up the `dsp.AudioFileReader` and `audioDeviceWriter` 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 = fileReader();
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);
    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);
```

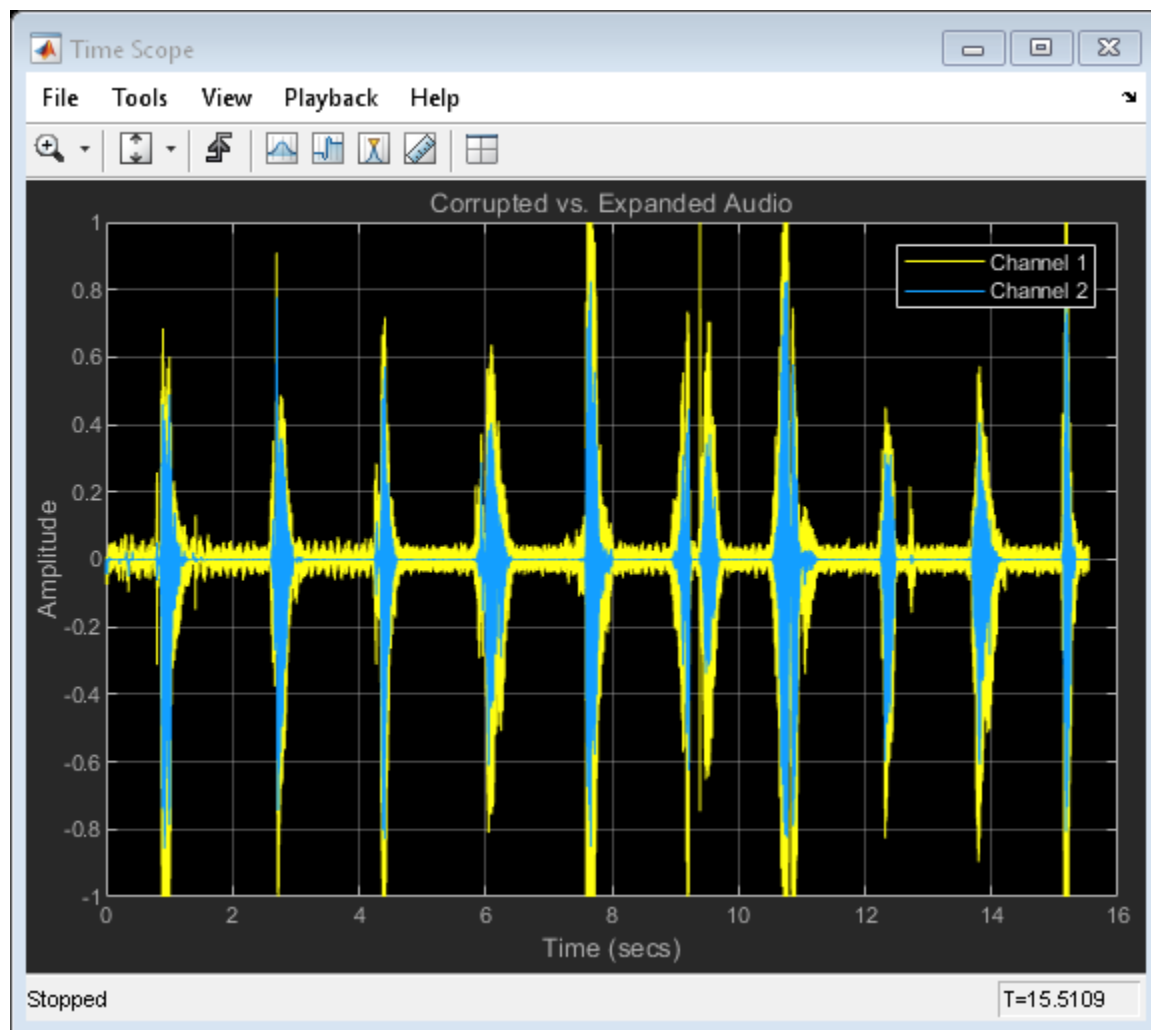
Set up the scope to visualize the signal before and after dynamic range expansion.

```
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 = fileReader();  
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);  
    y = dRE(xCorrupted);  
    deviceWriter(y);  
    scope([xCorrupted,y])  
end  
  
release(fileReader)  
release(dRE)  
release(deviceWriter)  
release(scope)
```





### Apply Split-Band De-Essing

De-essing is the process of diminishing sibilant sounds in an audio signal. Sibilance refers to the *s*, *z*, and *sh* sounds in speech, which can be disproportionately emphasized during recording. *es* sounds fall under the category of unvoiced speech with all consonants and

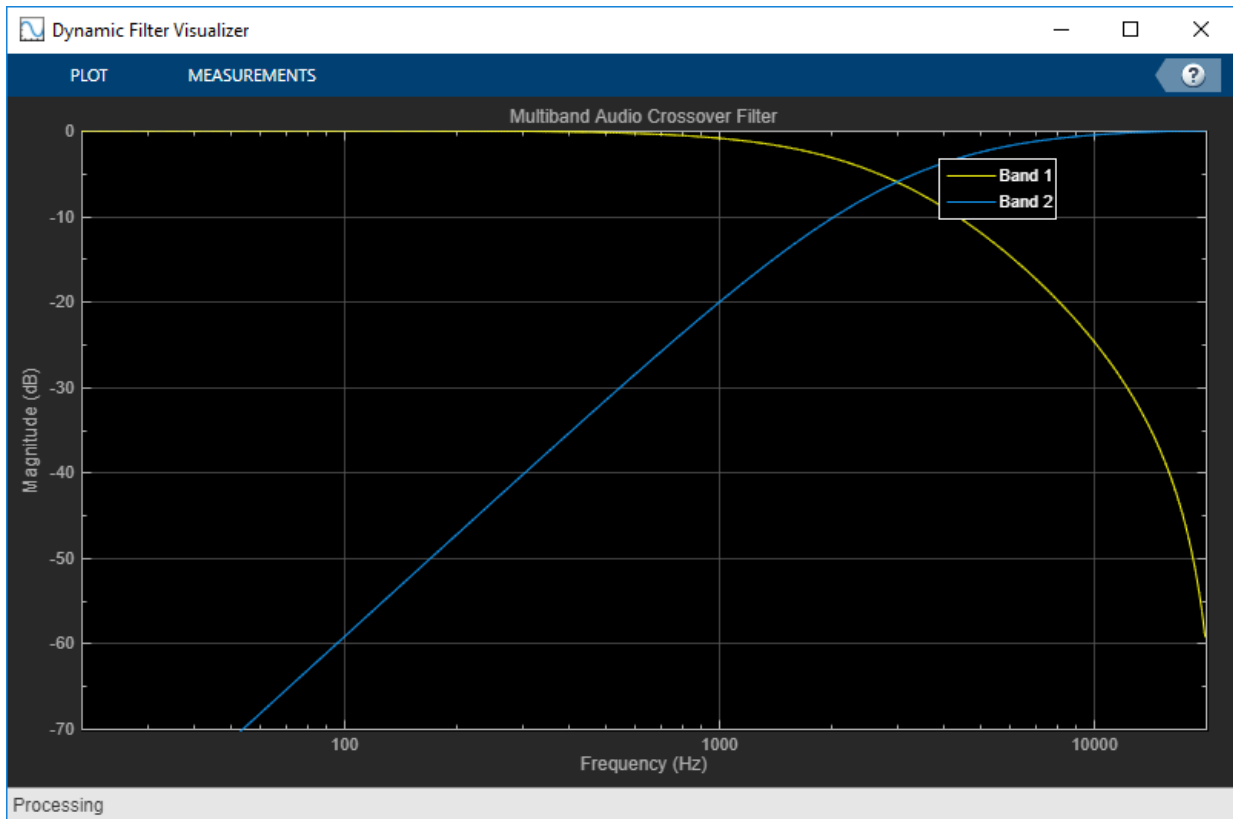
have a higher frequency than voiced speech. In this example, you apply split-band de-essing to a speech signal by separating the signal into high and low frequencies, applying an expander to diminish the sibilant frequencies, and then remixing the channels.

Create a `dsp.AudioFileReader` object and an `audioDeviceWriter` object to read from a sound file and write to an audio device. Listen to the unprocessed signal. Then release the file reader and device writer.

```
fileReader = dsp.AudioFileReader( ...  
    fullfile(matlabroot,'examples','audio','Sibilance.wav'));  
deviceWriter = audioDeviceWriter;  
  
while ~isDone(fileReader)  
    audioIn = fileReader();  
    deviceWriter(audioIn);  
end  
  
release(deviceWriter)  
release(fileReader)
```

Create an expander System object to de-ess the audio signal. Set the sample rate of the expander to the sample rate of the audio file. Create a two-band crossover filter with a crossover of 3000 Hz. Sibilance is usually found in this range. Set the crossover slope to 12. Plot the frequency response of the crossover filter to confirm your design visually.

```
dRExpander = expander( ...  
    'Threshold',-50, ...  
    'AttackTime', 0.05, ...  
    'ReleaseTime',0.05, ...  
    'HoldTime',0.005, ...  
    'SampleRate',fileReader.SampleRate);  
  
crossFilt = crossoverFilter( ...  
    'NumCrossovers',1, ...  
    'CrossoverFrequencies',3000, ...  
    'CrossoverSlopes',12);  
visualize(crossFilt)
```



Create a `dsp.TimeScope` System object to visualize the original and processed audio signals.

```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',4, ...
    'BufferLength',fileReader.SampleRate*8, ...
    'YLimits',[-1,1], ...
    'ShowGrid',true, ...
    'ShowLegend',true, ...
    'ChannelNames',{'Original','Processed'});
```

In an audio stream loop:

- 1 Read in a frame of the audio file.
- 2 Split the audio signal into two bands.
- 3 Apply dynamic range expansion to the upper band.
- 4 Remix the channels.
- 5 Write the processed audio signal to your audio device for listening.
- 6 Visualize the processed and unprocessed signals on a time scope.

As a best practice, release your objects once done.

```
while ~isDone(fileReader)
    audioIn = fileReader();

    [band1,band2] = crossFilt(audioIn);

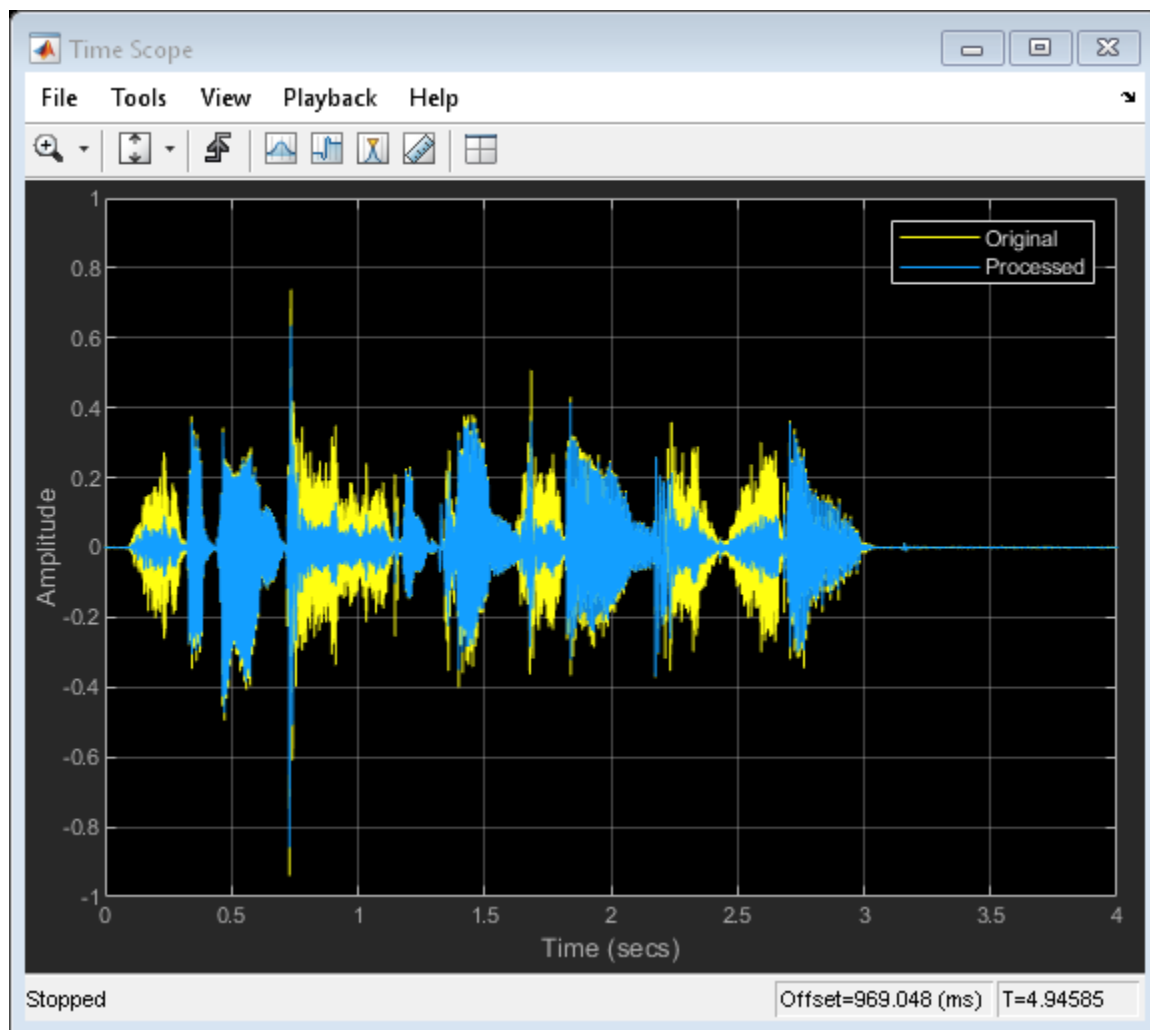
    band2processed = dRExpander(band2);

    procAudio = band1 + band2processed;

    deviceWriter(procAudio);

    scope([audioIn procAudio]);
end

release(deviceWriter)
release(fileReader)
release(scope)
```



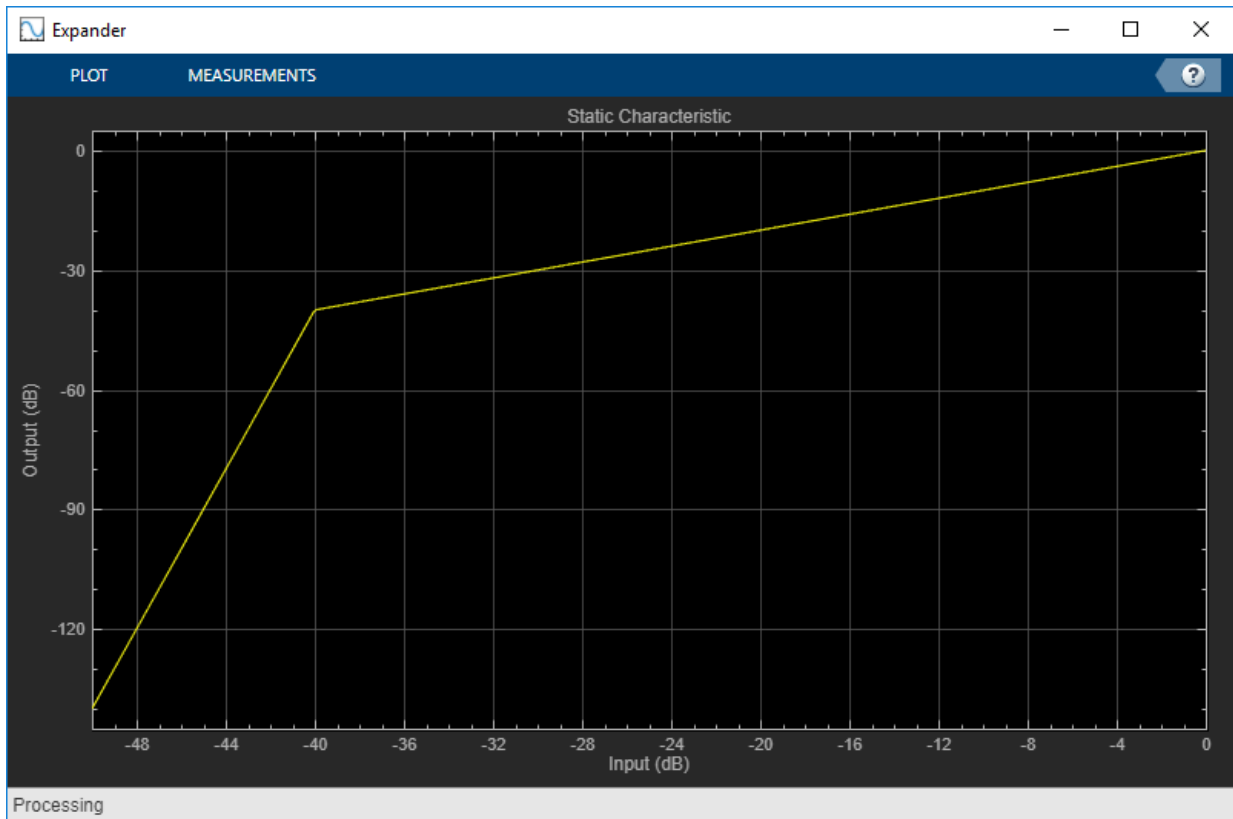
```
release(crossFilt)  
release(dRExpander)
```

### Tune Expander Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create a `audioDeviceWriter` to write audio to your sound card. Create an `expander` to process the audio data. Call `visualize` to plot the static characteristic of the expander.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

dRE = expander(-40,10, ...
    'AttackTime',0.01, ...
    'ReleaseTime',0.02, ...
    'HoldTime',0, ...
    'SampleRate',fileReader.SampleRate);
visualize(dRE)
```

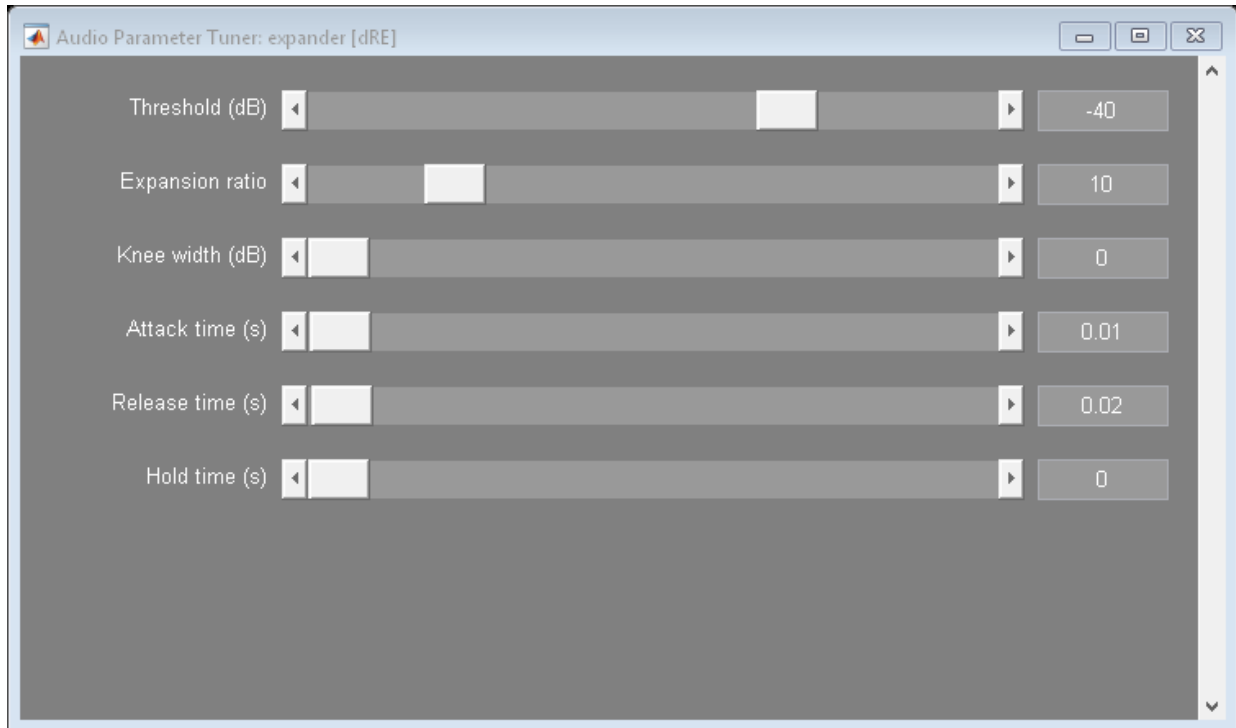


Create a `dsp.TimeScope` to visualize the original and processed audio.

```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpan',1, ...
    'BufferLength',fileReader.SampleRate*4, ...
    'YLimits',[-1,1], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'LayoutDimensions',[2,1], ...
    'NumInputPorts',2, ...
    'Title','Original vs. Processed Audio (top) and Applied Gain in dB (bottom)');
scope.ActiveDisplay = 2;
scope.YLimits = [-300,0];
scope.YLabel = 'Gain (dB)';
```

Call `parameterTuner` to open a UI to tune parameters of the expander while streaming.

`parameterTuner(dRE)`



In an audio stream loop:

- 1 Read in a frame of audio from the file.
- 2 Apply dynamic range expansion.
- 3 Write the frame of audio to your audio device for listening.
- 4 Visualize the original and processed audio, and the gain applied.

While streaming, tune parameters of the dynamic range expander and listen to the effect.

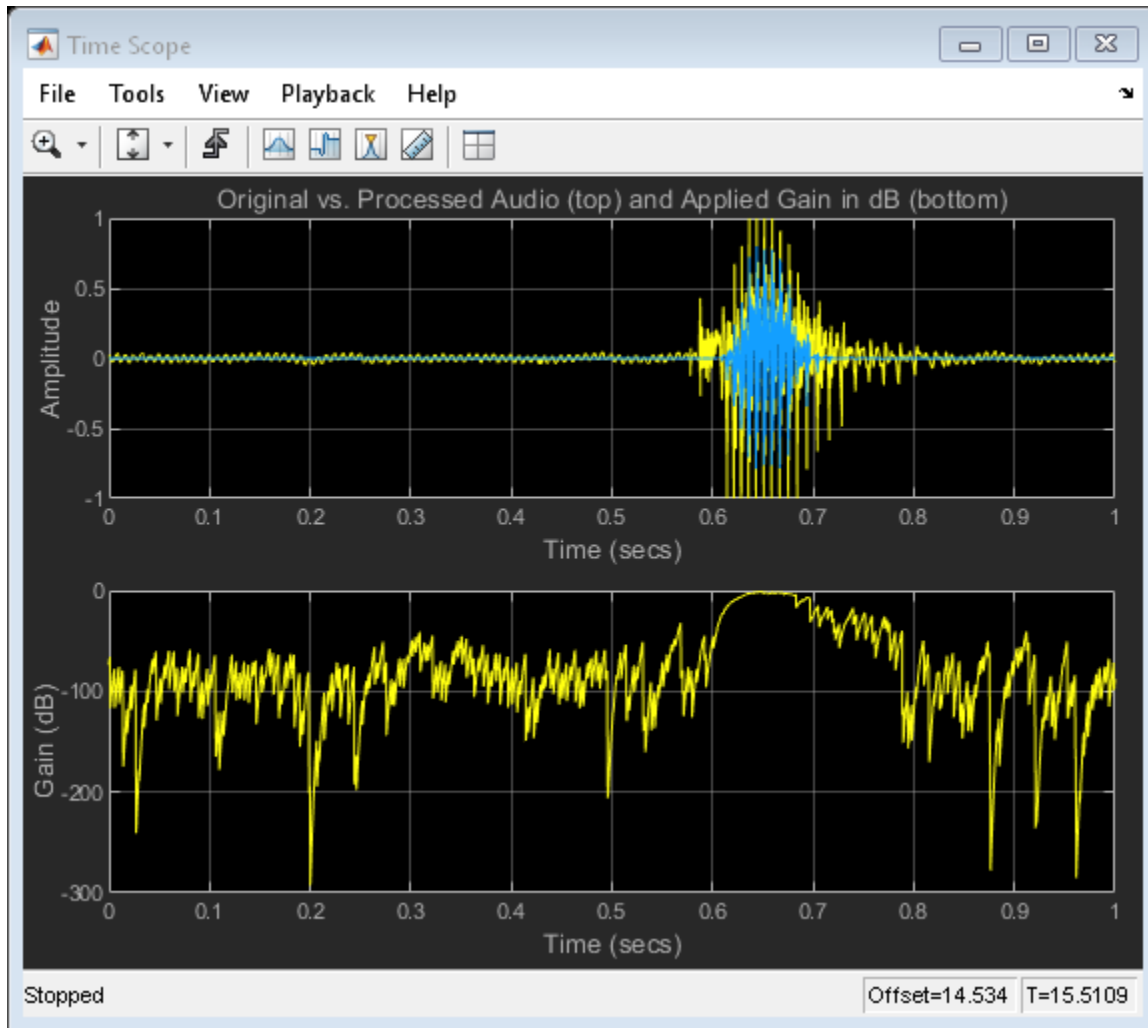
```
while ~isDone(fileReader)
    audioIn = fileReader();
    [audioOut,g] = dRE(audioIn);
    deviceWriter(audioOut);
```



```
    scope([audioIn(:,1),audioOut(:,1)],g(:,1));  
    drawnow limitrate % required to update parameter  
end
```

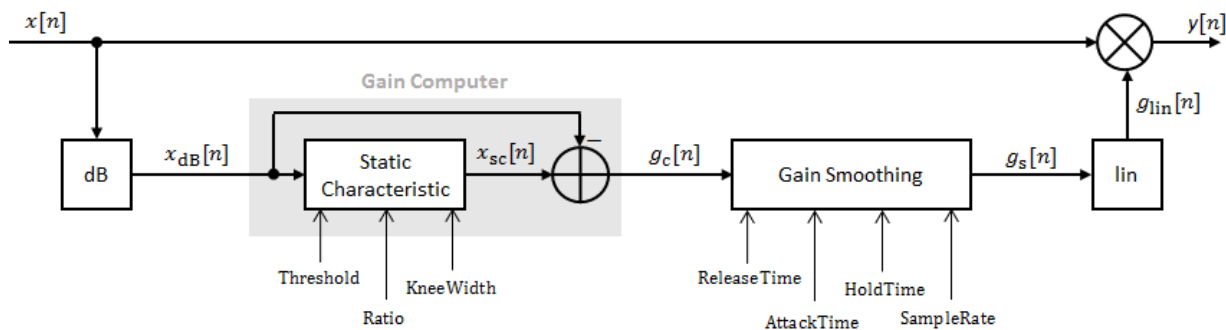
As a best practice, release your objects once done.

```
release(deviceWriter)  
release(fileReader)  
release(dRE)  
release(scope)
```



## Algorithms

The expander System object processes a signal frame by frame and element by element.



## Convert Input Signal to dB

The  $N$ -point signal,  $x[n]$ , is converted to decibels:

$$x_{dB}[n] = 20 \times \log_{10}|x[n]|$$

## Gain Computer

$x_{dB}[n]$  passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range expander to attenuate gain that is below the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} T + (x_{dB} - T) \times R & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{(1 - R)(x_{dB} - T - \frac{W}{2})^2}{2W} & \left(T - \frac{W}{2}\right) \leq x_{dB} \leq \left(T + \frac{W}{2}\right) \\ x_{dB} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases} ,$$

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:

$$x_{sc}(x_{dB}) = \begin{cases} T + (x_{dB} - T) \times R & x_{dB} < T \\ x_{dB} & x_{dB} \geq T \end{cases}$$

The computed gain,  $g_c[n]$ , is calculated as

$$g_c[n] = x_{sc}[n] - x_{dB}[n].$$

## Gain Smoothing

$g_c[n]$  is smoothed using specified attack, release, and hold time properties:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) g_c[n] & (C_A > T_H) \ \& \ (g_c[n] \leq g_s[n-1]) \\ g_s[n-1] & C_A \leq T_H \\ \alpha_R g_s[n-1] + (1 - \alpha_R) g_c[n] & (C_R > T_H) \ \& \ (g_c[n] > g_s[n-1]) \\ g_s[n-1] & C_R \leq T_H \end{cases}$$

The attack time coefficient,  $\alpha_A$ , is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{FS \times T_A}\right).$$

The release time coefficient,  $\alpha_R$ , 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 `AttackTime` property.  $T_R$  is the release time period, specified by the `ReleaseTime` property.  $FS$  is the input sampling rate, specified by the `SampleRate` property.

$C_A$  and  $C_R$  are hold counters for attack and release, respectively. The limit,  $T_H$ , is determined by the `HoldTime` property.

## Calculate and Apply Linear Gain

The smoothed gain in dB,  $g_s[n]$ , is translated to a linear domain:

$$g_{lin}[n] = 10\left(\frac{g_s[n]}{20}\right)$$

The output of the dynamic range expander 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, 2012, pp. 399-408.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

Expander | compressor | limiter | noiseGate

### Topics

“Dynamic Range Control”

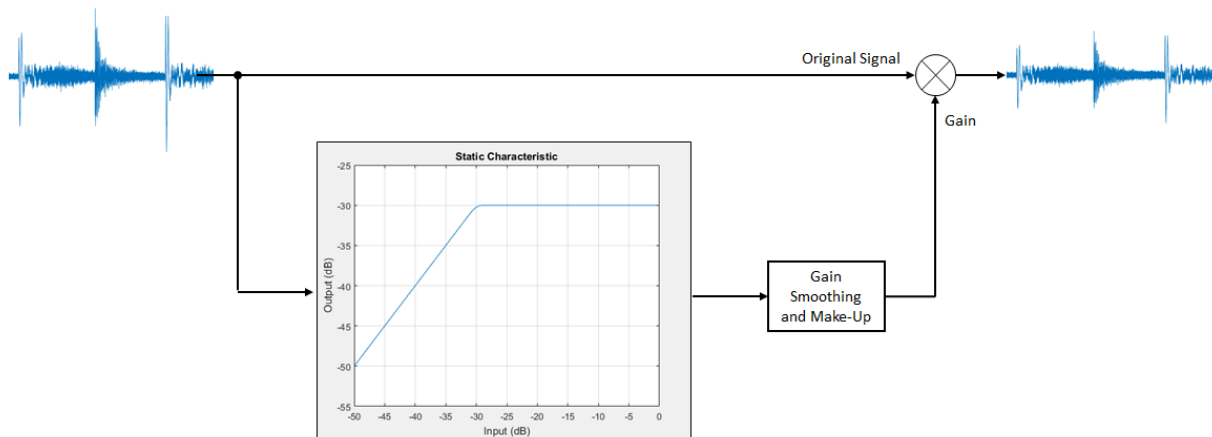
**Introduced in R2016a**

# limiter

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:

- 1 Create the `limiter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

## Syntax

```
dRL = limiter
dRL = limiter(thresholdValue)
dRL = limiter( ____,Name,Value)
```

## Description

`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 a 10 ms attack time and a sample rate of 16 kHz.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects \(MATLAB\)](#).

### **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.

**Tunable:** Yes

Data Types: `single` | `double`

**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.

**Tunable:** Yes

Data Types: `single` | `double`

**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.

**Tunable:** Yes

Data Types: `single` | `double`

**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.

**Tunable:** Yes

Data Types: `single` | `double`

### **MakeUpGainMode — Make-up gain mode**

'Property' (default) | 'Auto'

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.

**Tunable:** No

Data Types: `char` | `string`

### **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.

**Tunable:** Yes

### **Dependencies**

To enable this property, set MakeUpGainMode to 'Property'.

Data Types: `single` | `double`

### **SampleRate — Input sample rate (Hz)**

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: `single` | `double`

# Usage

## Syntax

```
audioOut = dRL(audioIn)  
[audioOut,gain] = dRL(audioIn)
```

## Description

`audioOut = dRL(audioIn)` performs dynamic range limiting on the input signal, `audioIn`, and returns the limited signal, `audioOut`. The type of dynamic range limiting is specified by the algorithm and properties of the `limiter` System object, `dRL`.

`[audioOut,gain] = dRL(audioIn)` also returns the applied gain, in dB, at each input sample.

## Input Arguments

### **audioIn** — Audio input to limiter

matrix

Audio input to the limiter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

## Output Arguments

### **audioOut** — Audio output from limiter

matrix

Audio output from the limiter, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

### **gain** — Gain applied by limiter (dB)

matrix

Gain applied by the limiter, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `limiter`

`visualize` Visualize static characteristic of dynamic range controller  
`createAudioPluginClass` Create audio plugin class that implements functionality of System object  
`parameterTuner` Tune object parameters while streaming

### MIDI

`configureMIDI` Configure MIDI connections between audio object and MIDI controller  
`disconnectMIDI` Disconnect MIDI controls from audio object  
`getMIDIConnections` Get MIDI connections of audio object

### Common to All System Objects

`clone` Create duplicate System object  
`isLocked` Determine if System object is in use  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object  
`step` Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `limiter` System object to user-facing parameters:

Property	Range	Mapping	Unit
Threshold	[-50, 0]	linear	dB
KneeWidth	[0, 20]	linear	dB
AttackTime	[0, 4]	linear	seconds

Property	Range	Mapping	Unit
ReleaseTime	[0, 4]	linear	seconds
MakeUpGain (available when you set MakeUpGainMode to 'Property')	[-10, 24]	linear	dB

## Examples

### Limit Audio Signal

Use dynamic range limiting to suppress the volume of loud sounds.

Set up the `dsp.AudioFileReader` and `audioDeviceWriter` 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);
```

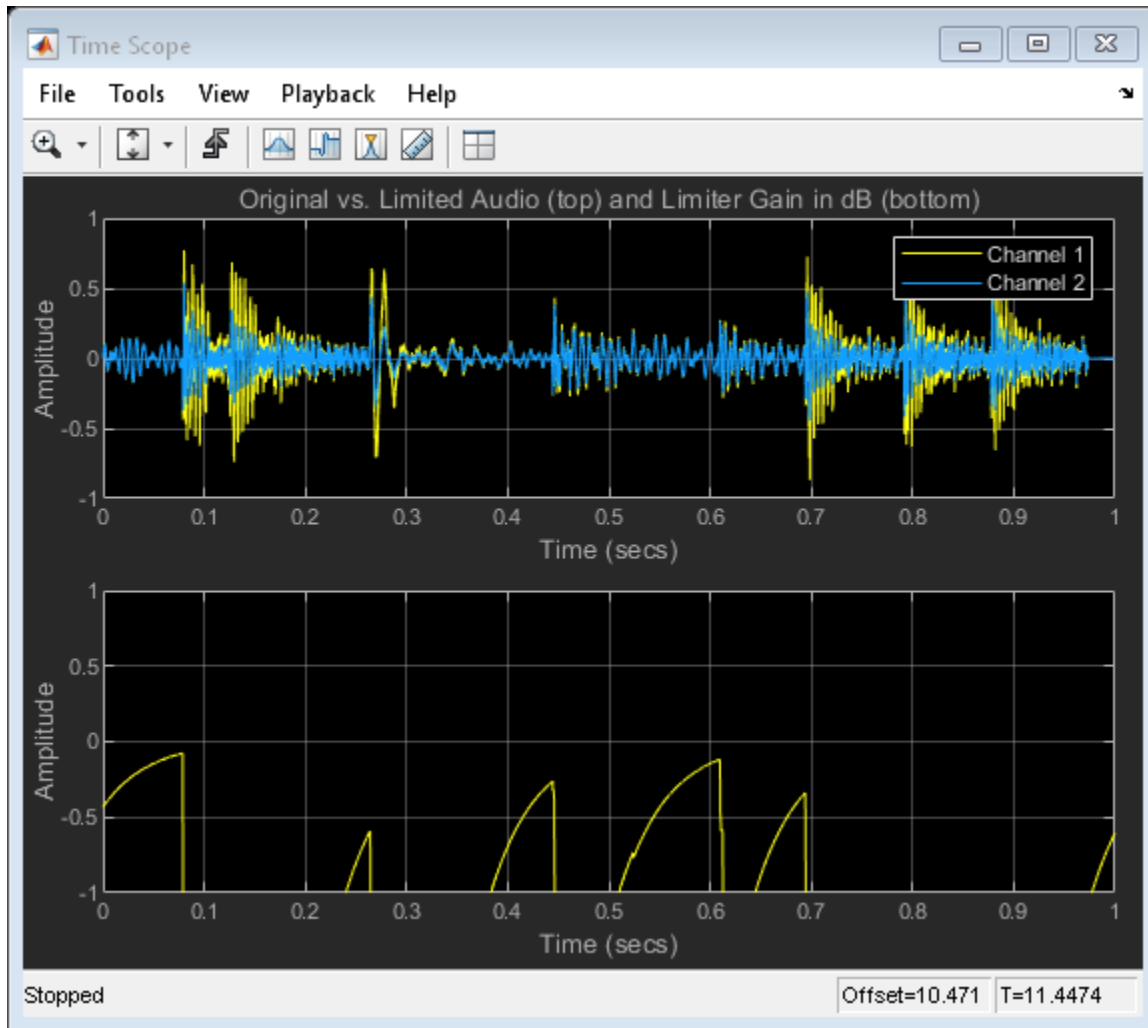
Set up a time scope to visualize the original signal and the limited signal.

```
scope = dsp.TimeScope( ...
    'SampleRate', fileReader.SampleRate, ...
    'TimeSpanOvverrunAction', '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 = fileReader();  
    [y,g] = dRL(x);  
    deviceWriter(y);  
    x1 = x(:,1);  
    y1 = y(:,1);  
    g1 = g(:,1);  
    scope([x1,y1],g1);  
end  
  
release(fileReader)  
release(dRL)  
release(deviceWriter)  
release(scope)
```



#### Compare Dynamic Range Limiter and Compressor

A dynamic range limiter is a special type of dynamic range compressor. In limiters, the level above an operational threshold is hard limited. In the simplest implementation of a limiter, the effect is equivalent to audio clipping. In compressors, the level above an

operational threshold is lowered using a specified compression ratio. Using a compression ratio results in a smoother processed signal.

### Compare Limiter and Compressor Applied to Sinusoid

Create a limiter System object™ and a compressor System object. Set the AttackTime and ReleaseTime properties of both objects to zero. Create an audioOscillator System object to generate a sinusoid with Frequency set to 5 and Amplitude set to 0.1.

```
dRL = limiter('AttackTime',0,'ReleaseTime',0);
dRC = compressor('AttackTime',0,'ReleaseTime',0);

osc = audioOscillator('Frequency',5,'Amplitude',0.1);
```

Create a time scope to visualize the generated sinusoid and the processed sinusoid.

```
scope = dsp.TimeScope( ...
    'SampleRate',osc.SampleRate, ...
    'TimeSpan',2, ...
    'BufferLength',osc.SampleRate*4, ...
    'YLimits',[-1 1], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'LayoutDimensions',[2 1], ...
    'NumInputPorts',2, ...
    'Title', ...
    'Original Signal vs. Limited Signal (top) and Compressed Signal (bottom)');
```

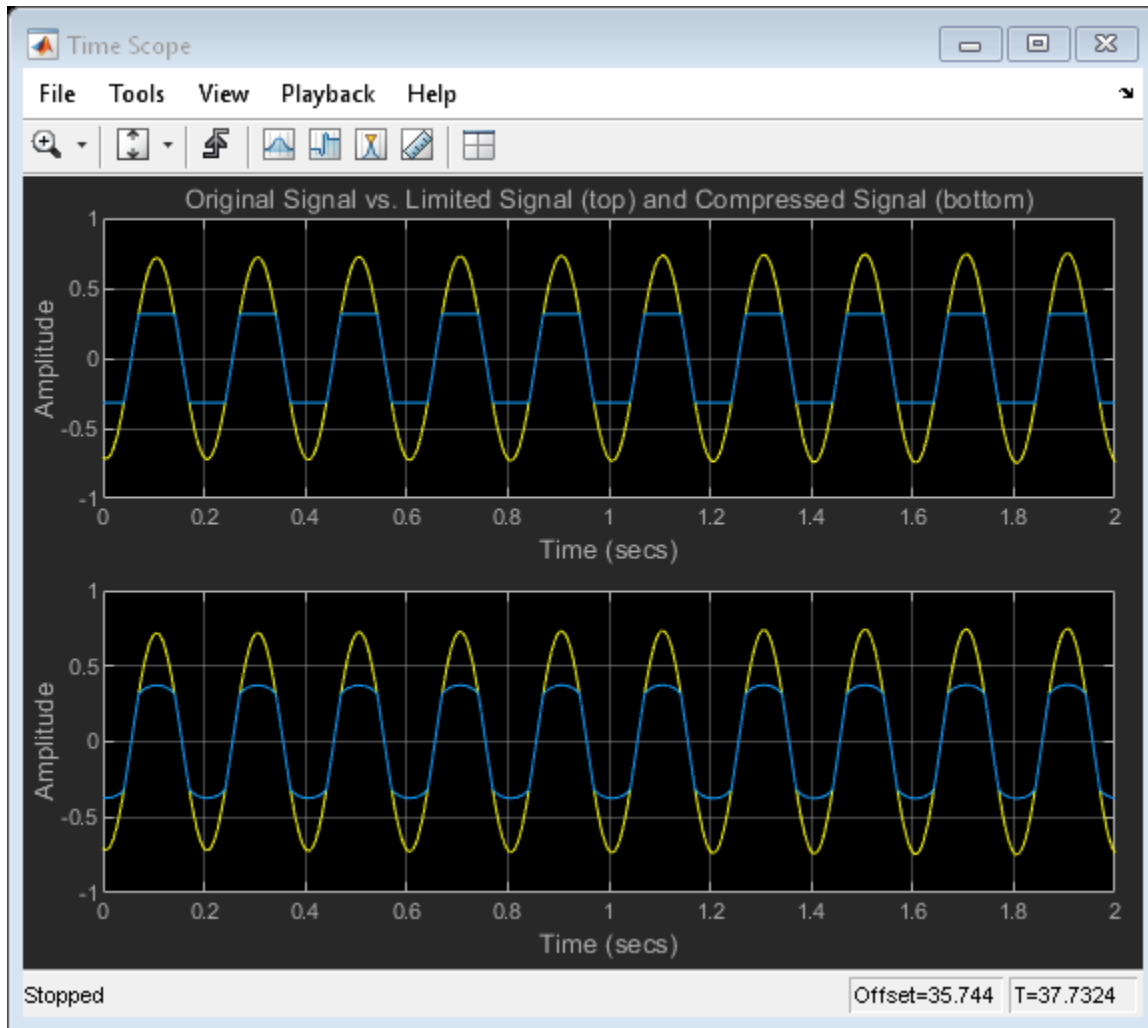
In an audio stream loop, visualize the original sinusoid and the sinusoid processed by a limiter and a compressor. Increment the amplitude of the original sinusoid to illustrate the effect.

```
while osc.Amplitude < 0.75
    x = osc();

    xLimited = dRL(x);
    xCompressed = dRC(x);

    scope([x xLimited],[x xCompressed]);

    osc.Amplitude = osc.Amplitude + 0.0002;
end
release(scope)
```



```
release(dRL)  
release(dRC)  
release(osc)
```

#### **Compare Limiter and Compressor Applied to Audio Signal**

Compare the effect of dynamic range limiters and compressors on a drum track. Create a `dsp.AudioFileReader` System object and a `audioDeviceWriter` System object to



read audio from a file and write to your audio output device. To emphasize the effect of dynamic range control, set the operational threshold of the limiter and compressor to -20 dB.

```
dRL.Threshold = -20;  
dRC.Threshold = -20;
```

```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3');  
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

Read successive frames from an audio file in a loop. Listen to and compare the effect of dynamic range limiting and dynamic range compression on an audio signal.

```
numFrames = 300;
```

```
fprintf('Now playing original signal...\n')
```

```
Now playing original signal...
```

```
for i = 1:numFrames  
    x = fileReader();  
    deviceWriter(x);
```

```
end
```

```
reset(fileReader);
```

```
fprintf('Now playing limited signal...\n')
```

```
Now playing limited signal...
```

```
for i = 1:numFrames  
    x = fileReader();  
    xLimited = dRL(x);  
    deviceWriter(xLimited);
```

```
end
```

```
reset(fileReader);
```

```
fprintf('Now playing compressed signal...\n')
```

```
Now playing compressed signal...
```

```
for i = 1:numFrames  
    x = fileReader();  
    xCompressed = dRC(x);  
    deviceWriter(xCompressed);
```

```
end
```

```
release(fileReader)
release(deviceWriter)
release(dRC)
release(dRL)
```

#### **Tune Limiter Parameters**

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create a `audioDeviceWriter` to write audio to your sound card. Create a `limiter` to process the audio data.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3', ...
    'SamplesPerFrame', frameLength);
deviceWriter = audioDeviceWriter('SampleRate', fileReader.SampleRate);

dRL = limiter('SampleRate', fileReader.SampleRate);
```

Call `parameterTuner` to open a UI to tune parameters of the limiter while streaming.

```
parameterTuner(dRL)
```



In an audio stream loop:

- 1 Read in a frame of audio from the file.
- 2 Apply dynamic range limiting.
- 3 Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the dynamic range limiter and listen to the effect.

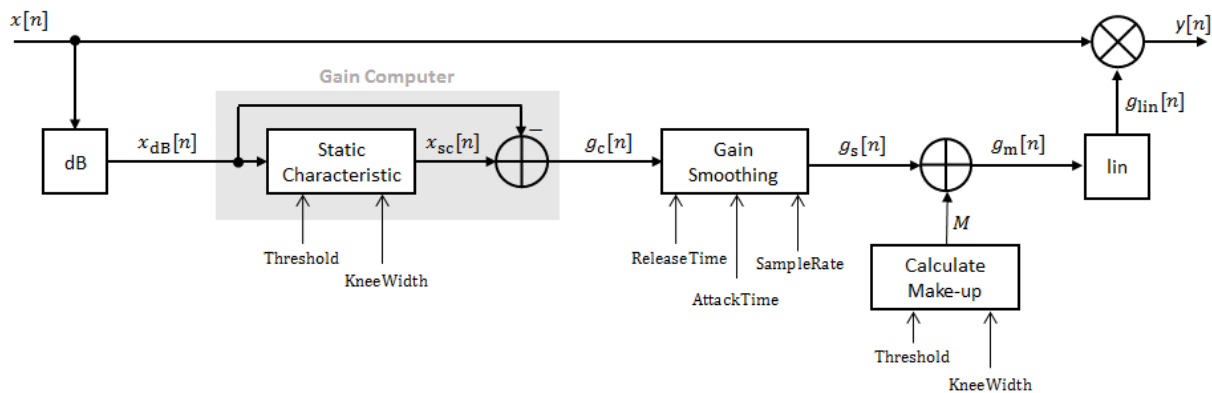
```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = dRL(audioIn);
    deviceWriter(audioOut);
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(dRL)
```

## Algorithms

The limiter System object processes a signal frame by frame and element by element.



### Convert Input Signal to dB

The  $N$ -point signal,  $x[n]$ , is converted to decibels:

$$x_{dB}[n] = 20 \times \log_{10}|x[n]|$$

### Gain Computer

$x_{dB}[n]$  passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range limiter to brick-wall gain that is above the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} - \frac{\left(x_{dB} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \leq x_{dB} \leq \left(T + \frac{W}{2}\right) \\ T & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases} ,$$

where  $T$  is the threshold and  $W$  is the knee width.

If you specified a hard knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < T \\ T & x_{dB} \geq T \end{cases}$$

The computed gain,  $g_c[n]$ , is calculated as

$$g_c[n] = x_{sc}[n] - x_{dB}[n].$$

## Gain Smoothing

$g_c[n]$  is smoothed using specified attack and release time:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) g_c[n], & g_c[n] \leq g_s[n-1] \\ \alpha_R g_s[n-1] + (1 - \alpha_R) g_c[n], & g_c[n] > g_s[n-1] \end{cases}$$

The attack time coefficient,  $\alpha_A$ , is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{FS \times T_A}\right).$$

The release time coefficient,  $\alpha_R$ , 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 `AttackTime` property.  $T_R$  is the release time period, specified by the `ReleaseTime` property.  $Fs$  is the input sampling rate, specified by the `SampleRate` property.

## Calculate and Apply Make-up Gain

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:

$$M = -x_{\text{sc}}|_{x_{\text{dB}} = 0}$$

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` and `KneeWidth` properties. It does not depend on the input signal.

The make-up gain,  $M$ , is added to the smoothed gain,  $g_s[n]$ :

$$g_m[n] = g_s[n] + M$$

## Calculate and Apply Linear Gain

The calculated gain in dB,  $g_m[n]$ , is translated to a linear domain:

$$g_{\text{lin}}[n] = 10^{\left(\frac{g_m[n]}{20}\right)}.$$

The output of the dynamic range limiter is given as

$$y[n] = x[n] \times g_{\text{lin}}[n].$$

## 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, 2012, pp. 399-408.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

## See Also

Limiter | compressor | expander | noiseGate

## Topics

“Dynamic Range Control”

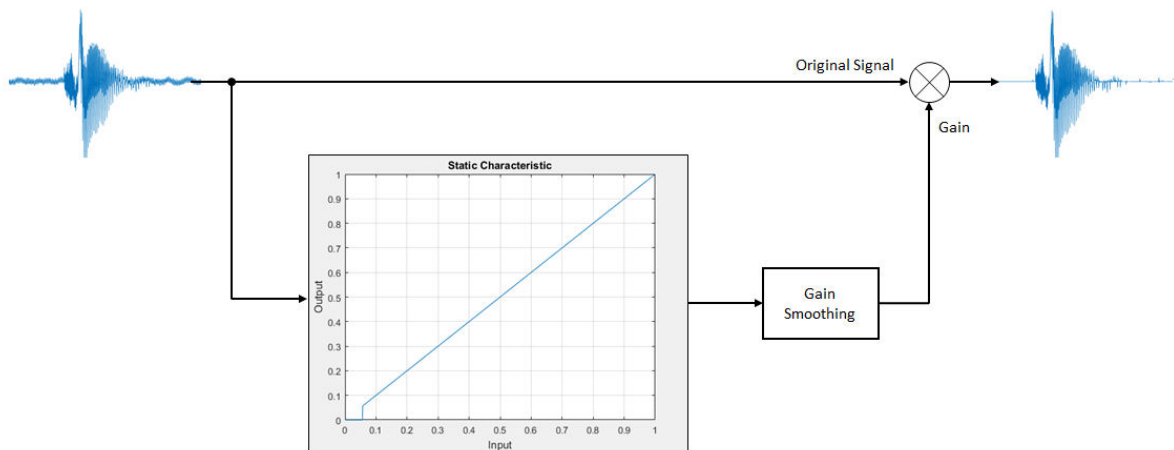
**Introduced in R2016a**

# noiseGate

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:

- 1 Create the `noiseGate` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).



## Creation

### Syntax

```
dRG = noiseGate
dRG = noiseGate(thresholdValue)
dRG = noiseGate( ____,Name,Value)
```

### Description

`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 a 10 ms attack time and a 16 kHz sample rate.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects (MATLAB)*.

### **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.

**Tunable:** Yes

Data Types: `single` | `double`

**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.

**Tunable:** Yes

Data Types: `single` | `double`

**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.

**Tunable:** Yes

Data Types: `single` | `double`

**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.

**Tunable:** Yes

Data Types: `single` | `double`

**SampleRate — Input sample rate (Hz)**

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: single | double

## Usage

## Syntax

```
audioOut = dRG(audioIn)  
[audioOut,gain] = dRG(audioIn)
```

## Description

`audioOut = dRG(audioIn)` performs dynamic range gating on the input signal, `audioIn`, and returns the gated signal, `audioOut`. The type of dynamic range gating is specified by the algorithm and properties of the `noiseGate` System object, `dRG`.

`[audioOut,gain] = dRG(audioIn)` also returns the applied gain, in dB, at each input sample.

## Input Arguments

### **audioIn** — Audio input to noise gate

matrix

Audio input to the noise gate, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: single | double

## Output Arguments

### **audioOut** — Audio output from noise gate

matrix

Audio output from the noise gate, returned as a matrix the same size as `audioIn`.

Data Types: single | double

#### **gain — Gain applied by noise gate (dB)**

matrix

Gain applied by noise gate, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### **Specific to noiseGate**

<code>visualize</code>	Visualize static characteristic of dynamic range controller
<code>createAudioPluginClass</code>	Create audio plugin class that implements functionality of System object
<code>parameterTuner</code>	Tune object parameters while streaming

### **MIDI**

<code>configureMIDI</code>	Configure MIDI connections between audio object and MIDI controller
<code>disconnectMIDI</code>	Disconnect MIDI controls from audio object
<code>getMIDIConnections</code>	Get MIDI connections of audio object

### **Common to All System Objects**

<code>clone</code>	Create duplicate System object
<code>isLocked</code>	Determine if System object is in use
<code>release</code>	Release resources and allow changes to System object property values and input characteristics
<code>reset</code>	Reset internal states of System object
<code>step</code>	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `noiseGate` System object to user-facing parameters:

Property	Range	Mapping	Unit
Threshold	[-140, 0]	linear	dB
AttackTime	[0, 4]	linear	seconds
ReleaseTime	[0, 4]	linear	seconds
HoldTime	[0, 4]	linear	seconds

## Examples

### Gate Audio Signal

Use dynamic range gating to attenuate background noise from an audio signal.

Set up the `dsp.AudioFileReader` and `audioDeviceWriter` 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 = fileReader();
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);
    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.

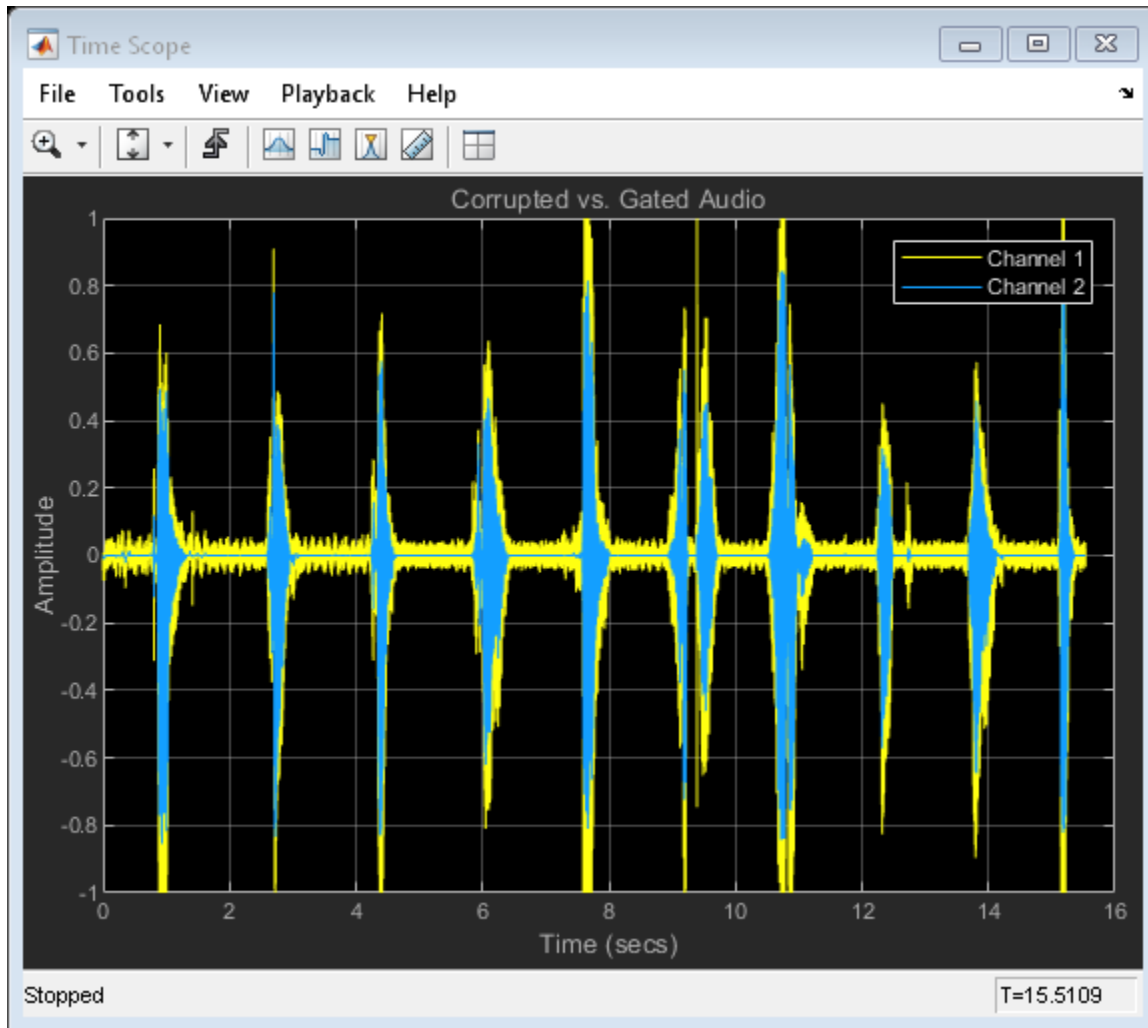
```
gate = noiseGate(-25, ...
    'AttackTime', 0.01, ...
    'ReleaseTime', 0.02, ...
    'HoldTime', 0, ...
    'SampleRate', fileReader.SampleRate);
```

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 = fileReader();  
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);  
    y = gate(xCorrupted);  
    deviceWriter(y);  
    scope([xCorrupted,y]);  
end  
  
release(fileReader)  
release(gate)  
release(deviceWriter)  
release(scope)
```



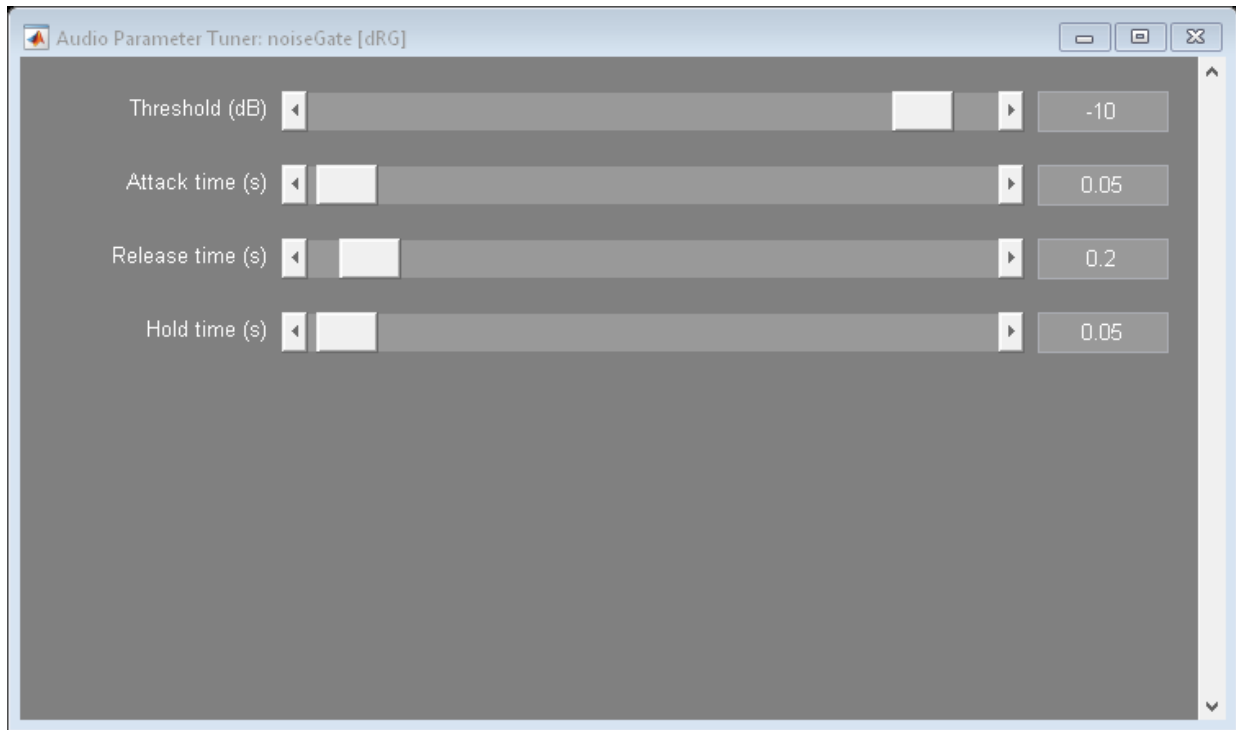
### Tune Noise Gate Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card. Create a `noiseGate` to process the audio data.

```
frameLength = 1024;  
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3', ...  
    'SamplesPerFrame',frameLength);  
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);  
  
dRG = noiseGate('SampleRate',fileReader.SampleRate);
```

Call `parameterTuner` to open a UI to tune parameters of the `noiseGate` while streaming.

```
parameterTuner(dRG)
```



In an audio stream loop:

- 1 Read in a frame of audio from the file.
- 2 Apply dynamic range gating.
- 3 Write the frame of audio to your audio device for listening.



While streaming, tune parameters of the dynamic range gate and listen to the effect.

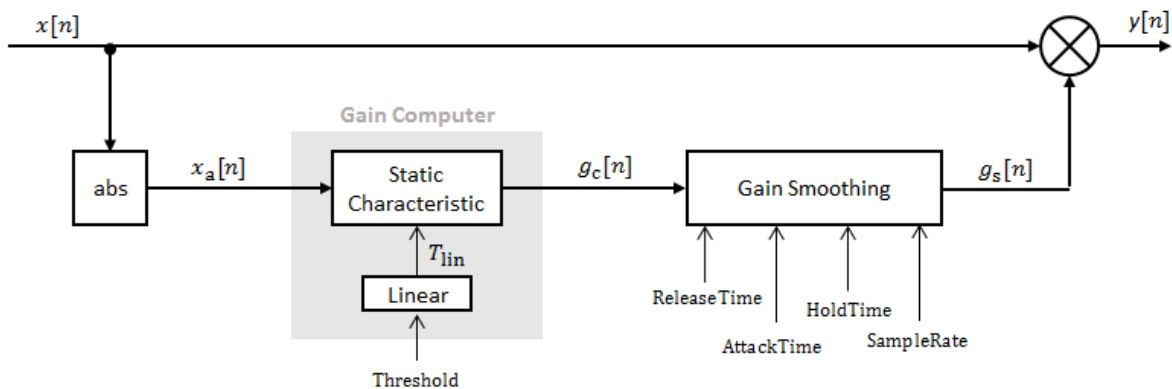
```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = dRG(audioIn);
    deviceWriter(audioOut);
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(dRG)
```

## Algorithms

The noiseGate System object processes a signal frame by frame and element by element.



### Convert Input Signal to Magnitude

The  $N$ -point signal,  $x[n]$ , is converted to magnitude:

$$x_a[n] = |x[n]|.$$

## Gain Computer

$x_a[n]$  passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range gate to determine a brick-wall gain for signal below the threshold:

$$g_c(x_a) = \begin{cases} 0 & x_a < T_{\text{lin}} \\ 1 & x_a \geq T_{\text{lin}} \end{cases}.$$

$T_{\text{lin}}$  is the threshold property converted to a linear domain:

$$T_{\text{lin}} = 10^{(T_{\text{dB}}/20)}.$$

## Gain Smoothing

The computed gain,  $g_c[n]$ , is smoothed using specified attack, release, and hold time properties:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) g_c[n] & (C_A > T_H) \ \& \ (g_c[n] \leq g_s[n-1]) \\ g_s[n-1] & C_A \leq T_H \\ \alpha_R g_s[n-1] + (1 - \alpha_R) g_c[n] & (C_R > T_H) \ \& \ (g_c[n] > g_s[n-1]) \\ g_s[n-1] & C_R \leq T_H \end{cases}$$

The attack time coefficient,  $\alpha_A$ , is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{F_S \times T_A}\right).$$

The release time coefficient,  $\alpha_R$ , 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.

$C_A$  and  $C_R$  are hold counters for attack and release, respectively. The limit,  $T_H$ , is determined by the `HoldTime` property.

## Apply Gain

The output of the dynamic range gate is given as

$$y[n] = x[n] \times g_s[n].$$

## 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, 2012, pp. 399-408.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

Noise Gate | compressor | expander | limiter

### Topics

“Dynamic Range Control”

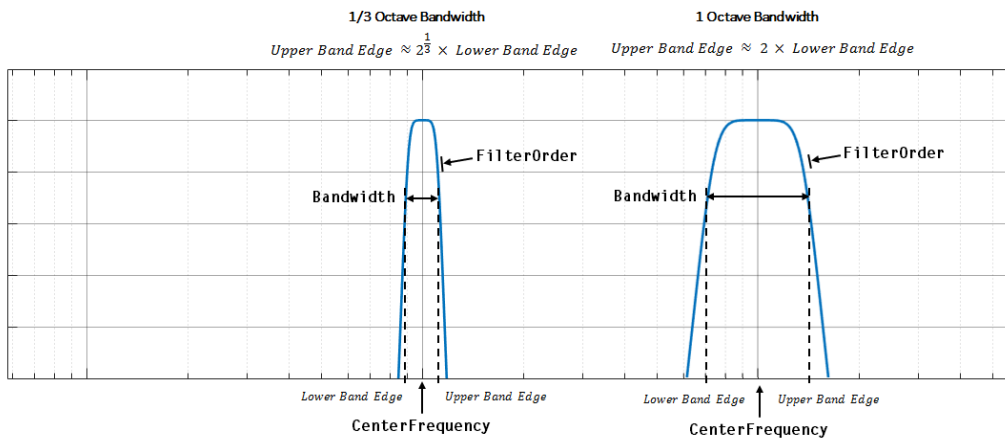
**Introduced in R2016a**

## octaveFilter

Octave-band and fractional octave-band filter

### Description

The `octaveFilter` System object performs octave-band or fractional octave-band filtering independently across each input channel. An octave-band is a frequency band where the highest frequency is twice the lowest frequency. Octave-band and fractional octave-band filters are commonly used to mimic how humans perceive loudness. Octave filters are best understood when viewed on a logarithmic scale, which models how the human ear weights the spectrum.



To perform octave-band or fractional octave-band filtering on your input:

- 1 Create the `octaveFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

## Syntax

```
octFilt = octaveFilter
octFilt = octaveFilter(centerFreq)
octFilt = octaveFilter(centerFreq,bw)
octFilt = octaveFilter( ____,Name,Value)
```

## Description

`octFilt = octaveFilter` creates a System object, `octFilt`, that performs octave-band filtering independently across each input channel.

`octFilt = octaveFilter(centerFreq)` sets the `CenterFrequency` property to `centerFreq`.

`octFilt = octaveFilter(centerFreq,bw)` sets the `Bandwidth` property to `bw`.

`octFilt = octaveFilter( ____,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `octFilt = octaveFilter(1000,'1/3 octave','SampleRate',96000)` creates a System object, `octFilt`, with a center frequency of 1000 Hz, a 1/3 octave filter bandwidth, and a sample rate of 96,000 Hz.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

### **FilterOrder — Order of octave filter**

6 (default) | even integer

Order of the octave filter, specified as an even integer.

**Tunable:** No

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

#### **CenterFrequency — Center frequency of octave filter (Hz)**

1000 (default) | positive scalar

Center frequency of the octave filter in Hz, specified as a positive scalar.

- The maximum center frequency is the value that causes the upper band edge to be equal to the Nyquist frequency,  $F_s/2$ . Frequencies above this value are saturated.
- The minimum center frequency is the value that causes the lower band edge to be equal to 1 Hz. Frequencies below this value are quantized to the value that corresponds to lower band edge equal to 1 Hz.

**Tunable:** Yes

Data Types: `single` | `double`

#### **Bandwidth — Filter bandwidth (octaves)**

'1 octave' (default) | '2/3 octave' | '1/2 octave' | '1/3 octave' | '1/6 octave' | '1/12 octave' | '1/24 octave' | '1/48 octave'

Filter bandwidth in octaves, specified as '1 octave', '2/3 octave', '1/2 octave', '1/3 octave', '1/6 octave', '1/12 octave', '1/24 octave', or '1/48 octave'.

**Tunable:** Yes

Data Types: `char` | `string`

#### **Oversample — Oversample toggle**

false (default) | true

Oversample toggle, specified as false or true.

- `false` -- The octave filter runs at the input sample rate.
- `true` -- The octave filter runs at two times the input sample rate. Oversampling minimizes the frequency warping effects introduced by the bilinear transformation. An FIR halfband interpolator implements oversampling before octave filtering. A halfband

decimator reduces the sample rate back to the input sampling rate after octave filtering.

**Tunable:** No

Data Types: `logical`

**SampleRate — Input sample rate (Hz)**

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: `single` | `double`

## Usage

## Syntax

```
audioOut = octFilt(audioIn)
```

## Description

`audioOut = octFilt(audioIn)` applies octave-band filtering to the input signal, `audioIn`, and returns the filtered signal, `audioOut`. The type of filtering is specified by the algorithm and properties of the `octaveFilter` System object, `octFilt`.

## Input Arguments

**audioIn — Audio input to octave filter**

matrix

Audio input to the octave filter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

## Output Arguments

### **audioOut** — Audio output from octave filter

matrix

Audio output from the octave filter, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to octaveFilter

<code>createAudioPluginClass</code>	Create audio plugin class that implements functionality of System object
<code>visualize</code>	Visualize and validate filter response
<code>isStandardCompliant</code>	Verify octave filter design is ANSI S1.11-2004 compliant
<code>getFilter</code>	Return biquad filter object with design parameters set
<code>getANSICenterFrequencies</code>	Get the list of valid ANSI S1.11-2004 center frequencies
<code>parameterTuner</code>	Tune object parameters while streaming

### MIDI

<code>configureMIDI</code>	Configure MIDI connections between audio object and MIDI controller
<code>disconnectMIDI</code>	Disconnect MIDI controls from audio object
<code>getMIDIConnections</code>	Get MIDI connections of audio object

### Common to All System Objects

<code>clone</code>	Create duplicate System object
<code>isLocked</code>	Determine if System object is in use
<code>release</code>	Release resources and allow changes to System object property values and input characteristics
<code>reset</code>	Reset internal states of System object



step      Run System object algorithm

---

**Note** octaveFilter supports additional filter analysis functions. See Analyze Octave Filter Design on page 3-401 for details.

---

## Examples

### Perform Fractional Octave-Band Filtering

Use octaveFilter to design a 1/3 octave-band filter centered at 1000 Hz. Process an audio signal using your octave filter design.

Create a dsp.AudioFileReader object.

```
samplesPerFrame = 1024;
reader = dsp.AudioFileReader('RockGuitar-16-44p1-stereo-72secs.wav', 'SamplesPerFrame', ...
```

Create an octaveFilter object. Use the sample rate of the reader as the sample rate of the octave filter.

```
centerFreq = 1000;
bw = '1/3 octave';
Fs = reader.SampleRate;
```

```
octFilt = octaveFilter(centerFreq,bw,'SampleRate',Fs);
```

Visualize the filter response and verify that it fits within the class 0 mask of the ANSI S1.11-2004 standard.

```
visualize(octFilt,'class 0')
```

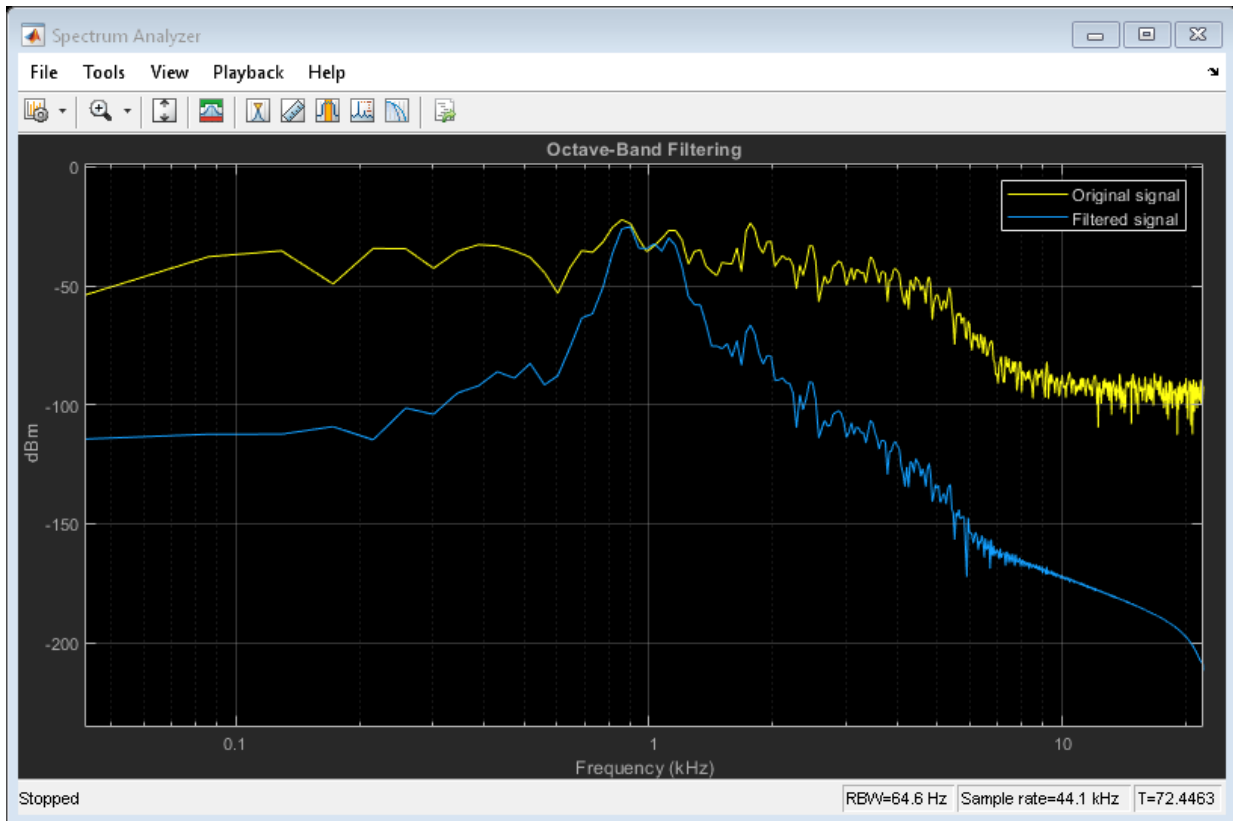
Create a spectrum analyzer to visualize the original audio signal and the audio signal after octave-band filtering.

```
scope = dsp.SpectrumAnalyzer( ...
    'SampleRate',Fs, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'FrequencyScale','Log', ...
    'FrequencyResolutionMethod','WindowLength', ...
    'WindowLength',samplesPerFrame, ...
```

```
'Title','Octave-Band Filtering', ...  
'ShowLegend',true, ...  
'ChannelNames',{'Original signal','Filtered signal'});
```

Process the audio signal in an audio stream loop. Visualize the filtered audio and the original audio. As a best practice, release the System objects when complete.

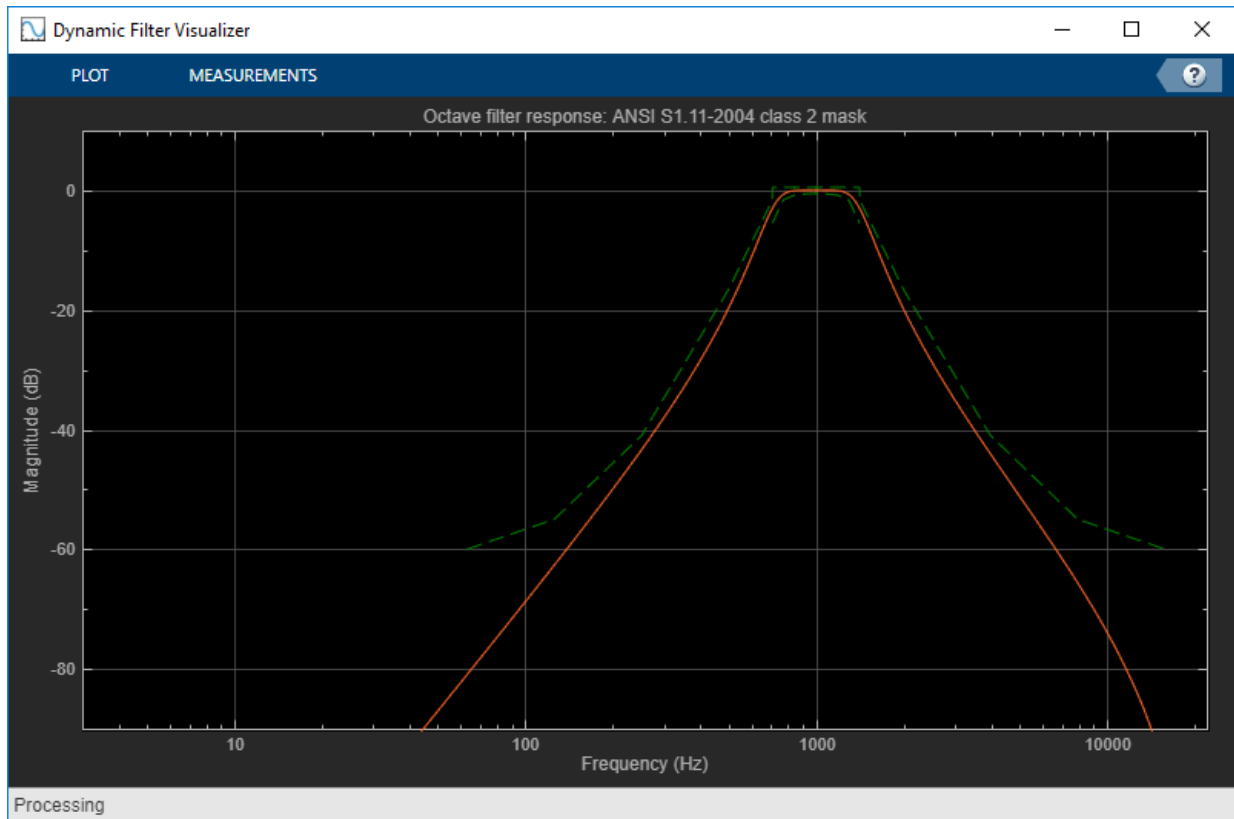
```
while ~isDone(reader)  
    x = reader();  
    y = octFilt(x);  
    scope([x(:,1),y(:,1)])  
end  
  
release(octFilt)  
release(reader)  
release(scope)
```



### Analyze Octave Filter Design

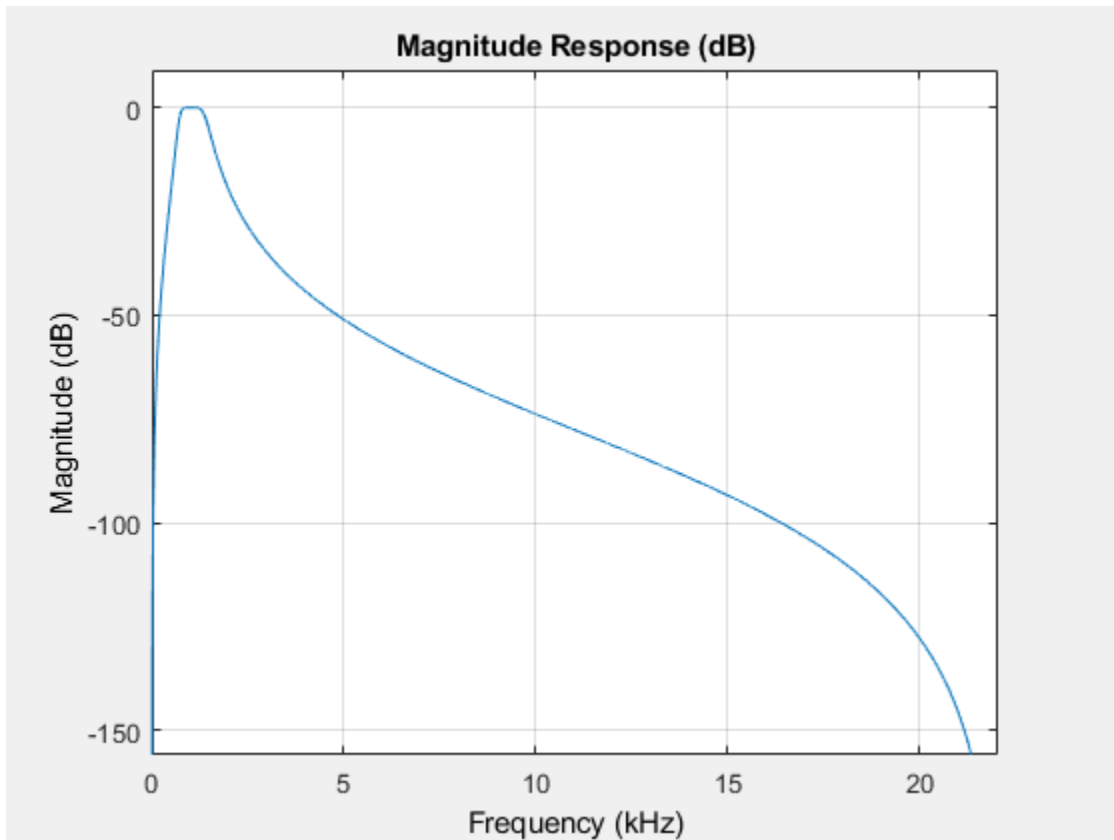
Create an octave filter. Visualize the filter response and validate that it is class 2 compliant.

```
octFilt = octaveFilter('CenterFrequency',1000);  
visualize(octFilt,'class 2')
```



Analyze the filter using `fvtool`.

```
fvtool(octFilt, 'Fs', octFilt.SampleRate)
```



The `octaveFilter` object supports several filter analysis methods. For more information, use `help` at the command line:

`help octaveFilter.helpFilterAnalysis`

The following analysis methods are available for discrete-time filter System objects:

<code>fvtool</code>	- Filter visualization tool
<code>info</code>	- Filter information
<code>freqz</code>	- Frequency response
<code>phasez</code>	- Phase response
<code>zerophase</code>	- Zero-phase response
<code>grpdelay</code>	- Group delay response
<code>phasedelay</code>	- Phase delay response

<code>impz</code>	- Impulse response
<code>impzlength</code>	- Length of impulse response
<code>stepz</code>	- Step response
<code>zplane</code>	- Pole/zero plot
<code>cost</code>	- Cost estimate for implementation of the filter System object
<code>measure</code>	- Measure characteristics of the frequency response
<code>order</code>	- Filter order
<code>coeffs</code>	- Filter coefficients in a structure
<code>firtype</code>	- Determine the type (1-4) of a linear phase FIR filter System object
<code>tf</code>	- Convert to transfer function
<code>zpk</code>	- Convert to zero-pole-gain
<code>ss</code>	- Convert to state space representation
<code>isallpass</code>	- Verify if filter System object is allpass
<code>isfir</code>	- Verify if filter System object is FIR
<code>islinphase</code>	- Verify if filter System object is linear phase
<code>ismaxphase</code>	- Verify if filter System object is maximum phase
<code>isminphase</code>	- Verify if filter System object is minimum phase
<code>isreal</code>	- Verify if filter System object is minimum real
<code>issos</code>	- Verify if filter System object is in second-order sections form
<code>isstable</code>	- Verify if filter System object is stable
<code>realizemdl</code>	- Filter realization (Simulink diagram)
<code>specifyall</code>	- Fully specify fixed-point filter System object settings
<code>cascade</code>	- Create a FilterCascade System object
Second-order sections:	
<code>scale</code>	- Scale second-order sections of BiquadFilter System object
<code>scalecheck</code>	- Check scaling of BiquadFilter System object
<code>reorder</code>	- Reorder second-order sections of BiquadFilter System object
<code>cumsec</code>	- Cumulative second-order section of BiquadFilter System object
<code>scaleopts</code>	- Create an options object for second-order section scaling
<code>sos</code>	- Convert to second-order-sections (for IIRFilter System objects only)
Fixed-Point (Fixed-Point Designer Required):	
<code>freqrespest</code>	- Frequency response estimate via filtering
<code>freqrespopts</code>	- Create an options object for frequency response estimate
<code>noisepsd</code>	- Power spectral density of filter output due to roundoff noise
<code>noisepsdopts</code>	- Create an options object for output noise PSD computation

Multirate Analysis:

```
polyphase           - Polyphase decomposition of multirate filter System object
gain (CIC decimator) - Gain of CIC decimator filter System object
gain (CIC interpolator) - Gain of CIC interpolator filter System object
```

For decimator, interpolator, or rate change filter System objects the analysis tools perform computations relative to the rate at which the filter is running. If a sampling frequency is specified, it is assumed that the filter is running at that rate.

Help for octaveFilter.helpFilterAnalysis is inherited from superclass DSP.PRIVATE.FILTER

### Create Octave-Band Filter Bank

Create an octave-band filter bank that conforms to ANSI S1.11-2004. Pass white noise through the filter bank and inspect the resulting power in each band.

Create an octave filter with default settings.

```
octFilt = octaveFilter;
```

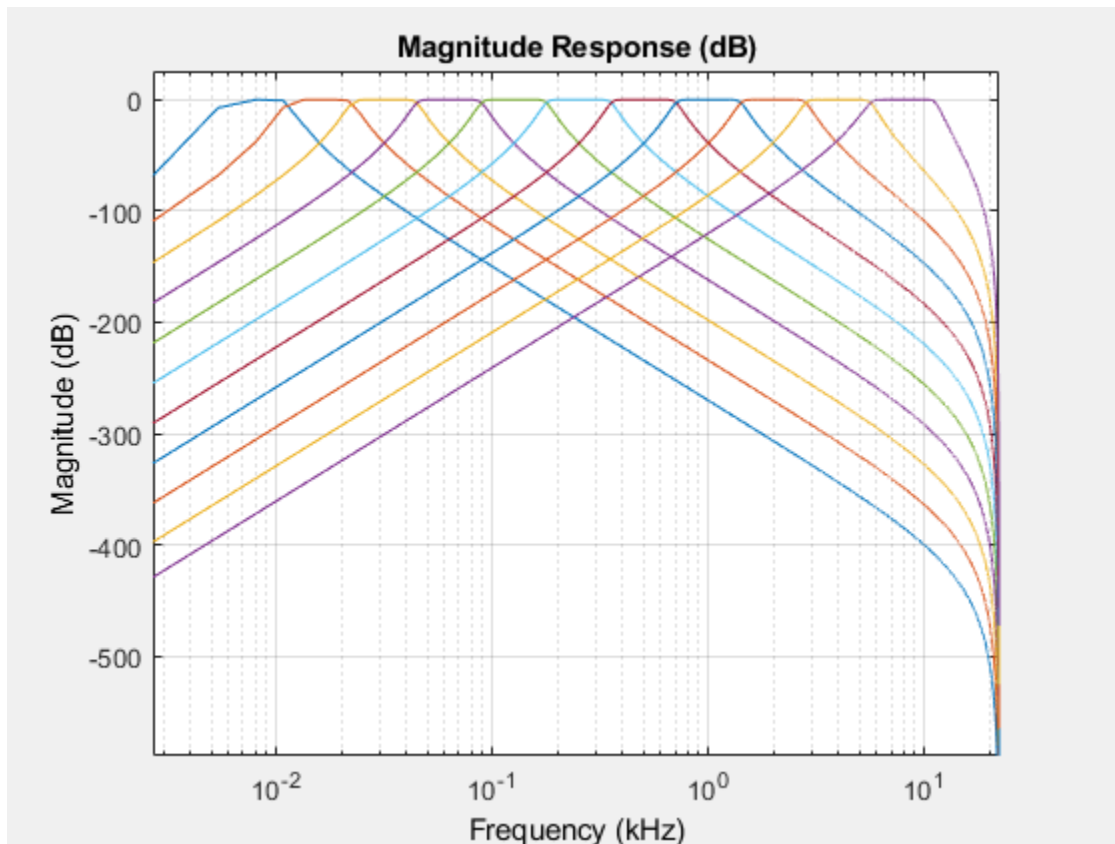
Get a vector of valid center frequencies, given the center frequency of octFilt. Create an octave filter bank using the valid center frequencies.

```
centerFrequencies = getANSICenterFrequencies(octFilt);
for i = 1:11
    octaveFilterBank{i} = octaveFilter(centerFrequencies(i), 'FilterOrder', 12);
end
```

Use `getFilter` to return biquad filter objects for each filter in your octave filter bank. Visualize the octave filter bank with a linear frequency scale. Visualize the octave filter bank with a logarithmic frequency scale. The logarithmic frequency scale makes the center frequencies appear evenly distributed.

```
fvtool(octaveFilterBank{1}, ...
       octaveFilterBank{2}, ...
       octaveFilterBank{3}, ...
       octaveFilterBank{4}, ...
       octaveFilterBank{5}, ...
       octaveFilterBank{6}, ...
```

```
octaveFilterBank{7}, ...  
octaveFilterBank{8}, ...  
octaveFilterBank{9}, ...  
octaveFilterBank{10}, ...  
octaveFilterBank{11}, ...  
'Fs', octaveFilterBank{1}.SampleRate, ...  
'FrequencyScale', 'Log');
```



Create a white noise signal. By definition, white noise has a flat power spectral density.

```
whiteNoiseGenerator = dsp.ColoredNoise(0,1024);  
whiteNoise = whiteNoiseGenerator();
```

Pass the white noise signal through the octave-band filter bank.

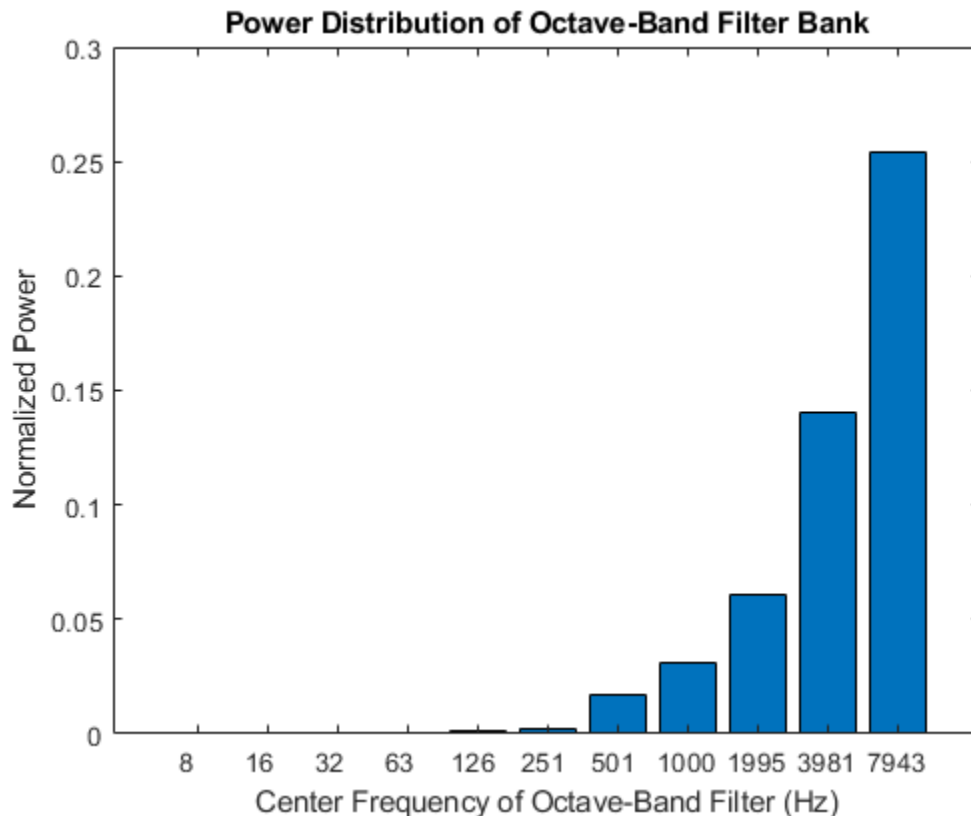


```
for i = 1:11
    filteredWhiteNoise(:,i) = octaveFilterBank{i}(whiteNoise);
end
```

Calculate and plot the power in each octave.

```
for i = 1:11
    powerPerBand(i) = bandpower(filteredWhiteNoise(:,i));
end
```

```
bar(powerPerBand)
title('Power Distribution of Octave-Band Filter Bank')
set(gca,'XTickLabel',{round(centerFrequencies)})
xlabel('Center Frequency of Octave-Band Filter (Hz)')
ylabel('Normalized Power')
```



The band power increases by a factor of approximately two because the octave bandwidth increases by a factor of two. The power distribution of an octave filter bank mimics how higher frequencies are perceived louder in white noise. You can use octave filter banks to weight a spectrum for perceived loudness.

### **Effect of Center Frequency on Octave-Band Filtering**

Process a speech signal using different octave bands from an octave-band filter bank.

Design a 1/2 octave filter with an estimated center frequency of 800 Hz. Use `isStandardCompliant` to find the nearest compliant center frequency.

```
octFilt = octaveFilter(800, '1/2 octave');
[complianceStatus, suggestedCenterFrequency] = isStandardCompliant(octFilt, 'class 0')
```

```
complianceStatus =
```

```
    logical
```

```
    0
```

```
suggestedCenterFrequency =
```

```
    841.3951
```

Change the center frequency of the `octFilt` object to the suggested center frequency returned by `isStandardCompliant`. Get a list of valid ANSI S1.11-2004 center frequencies, given your specified `octFilt` center frequency.

```
octFilt.CenterFrequency = suggestedCenterFrequency;
Fo = getANSICenterFrequencies(octFilt);
```

Create an audio file reader and audio device writer.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
deviceWriter = audioDeviceWriter('SampleRate', fileReader.SampleRate);
```

Create a scope to visualize the filtered and unfiltered signals.

```
scope = dsp.SpectrumAnalyzer(...
    'PlotAsTwoSidedSpectrum', false, ...
    'FrequencyScale', 'Log', ...
    'Title', 'Octave-Band Filtering', ...
    'ShowLegend', true, ...
    'ChannelNames', {'Original signal', 'Filtered signal'});
```

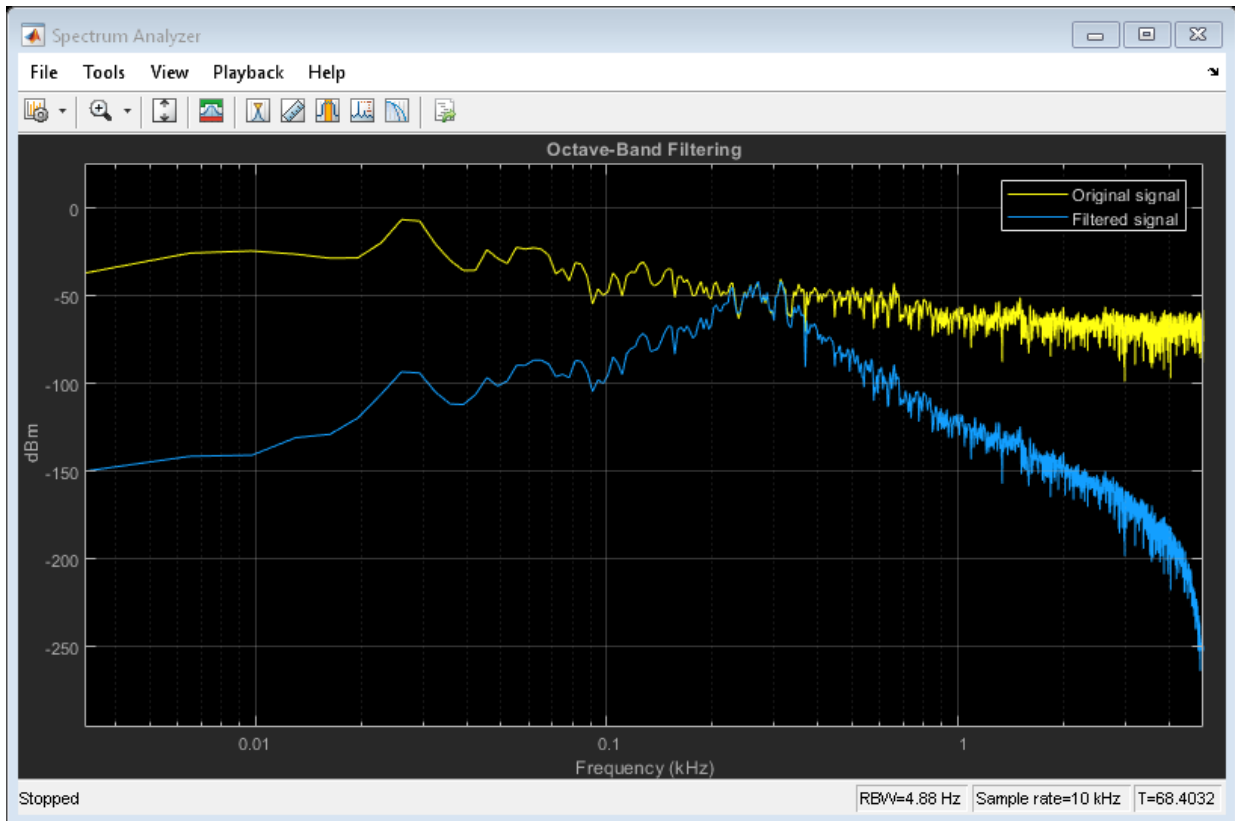
In an audio stream loop, process the audio signal using your octave-band filter. Vary the center frequency to hear the effect. As a best practice, release your objects after processing.

```
index = 12;
octFilt.CenterFrequency = Fo(index);
count = 1;
while ~isDone(fileReader)
```

```
x = fileReader();
y = octFilt(x);
scope([x,y])
deviceWriter(y);

if mod(count,100)==0
    octFilt.CenterFrequency = Fo(index);
    index = index+1;
end
count = count+1;
end

release(scope)
release(deviceWriter)
release(fileReader)
```



### Remove Noise from Tone Scale

Remove additive noise from an audio tone scale using an `octaveFilter`.

Create `audioOscillator` and `audioDeviceWriter` objects with default properties. Create an `octaveFilter` object with the center frequency set to 100 Hz.

```
osc = audioOscillator;
deviceWriter = audioDeviceWriter;
octFilt = octaveFilter(100);
```

In an audio stream loop, listen to a tone created by your audio oscillator. The tone contains additive Gaussian noise.

```
for i = 1:400
    x = osc();
    x1 = x + 0.1*randn(512,1);
    deviceWriter(x1);
    if rem(i,100)==0
        osc.Frequency = osc.Frequency*2;
    end
end
```

Create a spectrum analyzer to view your filtered and unfiltered signals.

```
scope = dsp.SpectrumAnalyzer( ...
    'PlotAsTwoSidedSpectrum',false, ...
    'FrequencyScale','Log', ...
    'FrequencyResolutionMethod','WindowLength', ...
    'Title','Octave-Band Filtering', ...
    'ShowLegend',true, ...
    'SpectralAverages',10, ...
    'ChannelNames',{'Original signal','Filtered signal'});
```

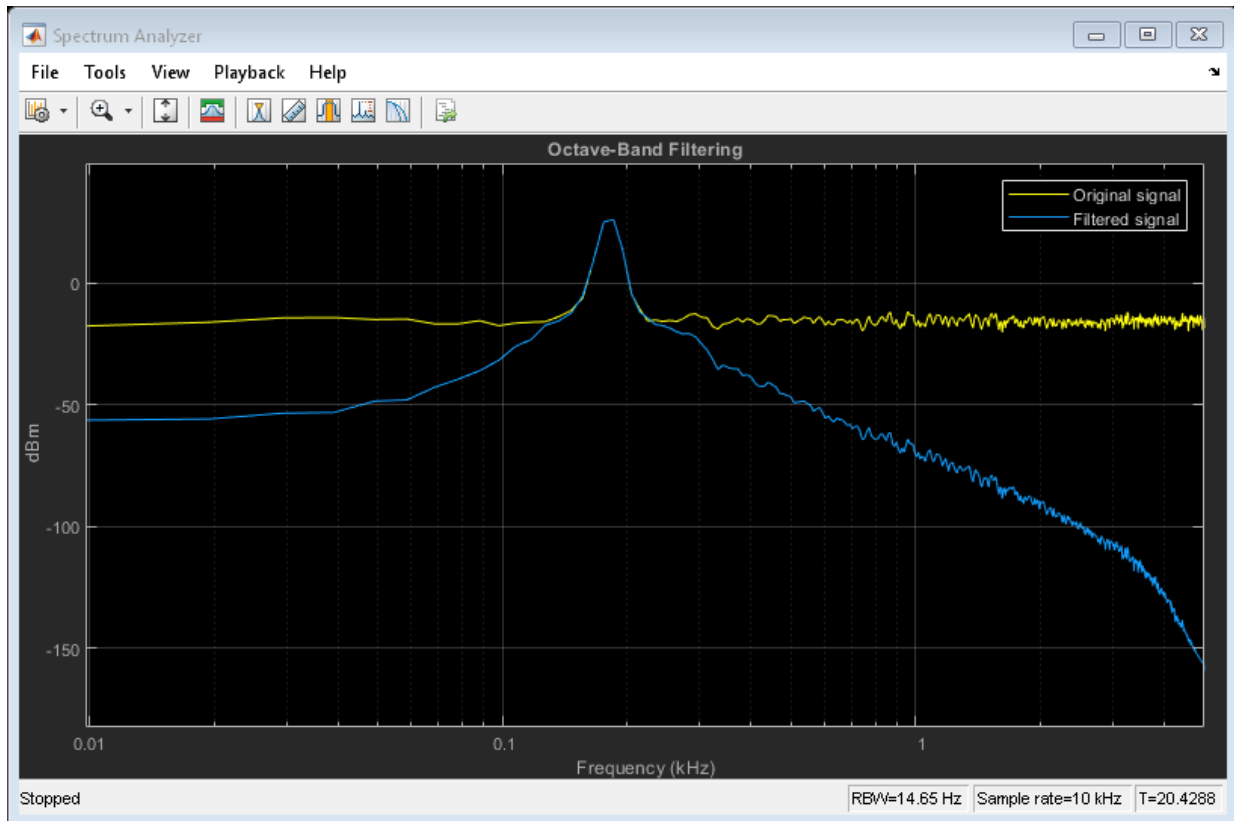
Reset the frequency of your audio oscillator to its default, 100 Hz.

```
osc.Frequency = 100;
```

In an audio stream loop, filter the corrupted tone using your octave-band filter. When the tone changes frequency in the loop, change the center frequency of your octave filter to match. As a best practice, release your audio device once done.

```
for i = 1:400
    x = osc();
    x1 = x + 0.1*randn(512,1);
    x2 = octFilt(x1);
    deviceWriter(x2);
    if rem(i,100)==0
        osc.Frequency = osc.Frequency*2;
        octFilt.CenterFrequency = octFilt.CenterFrequency*2;
    end
    scope([x1,x2])
end

release(deviceWriter)
release(scope)
```



## Design Compliant High-Frequency Filters

Design a sixth-order 1/3 octave filter with a sample rate of 96 kHz.

```
octFilt = octaveFilter('FilterOrder',6, ...
    'Bandwidth','1/3 octave', ...
    'SampleRate',96e3);
```

Get the center frequencies defined by the ANSI S1.11-2004 standard. The center frequencies defined by the standard depend on the Bandwidth and SampleRate properties.

```
centerFrequencies = getANSICenterFrequencies(octFilt)
```

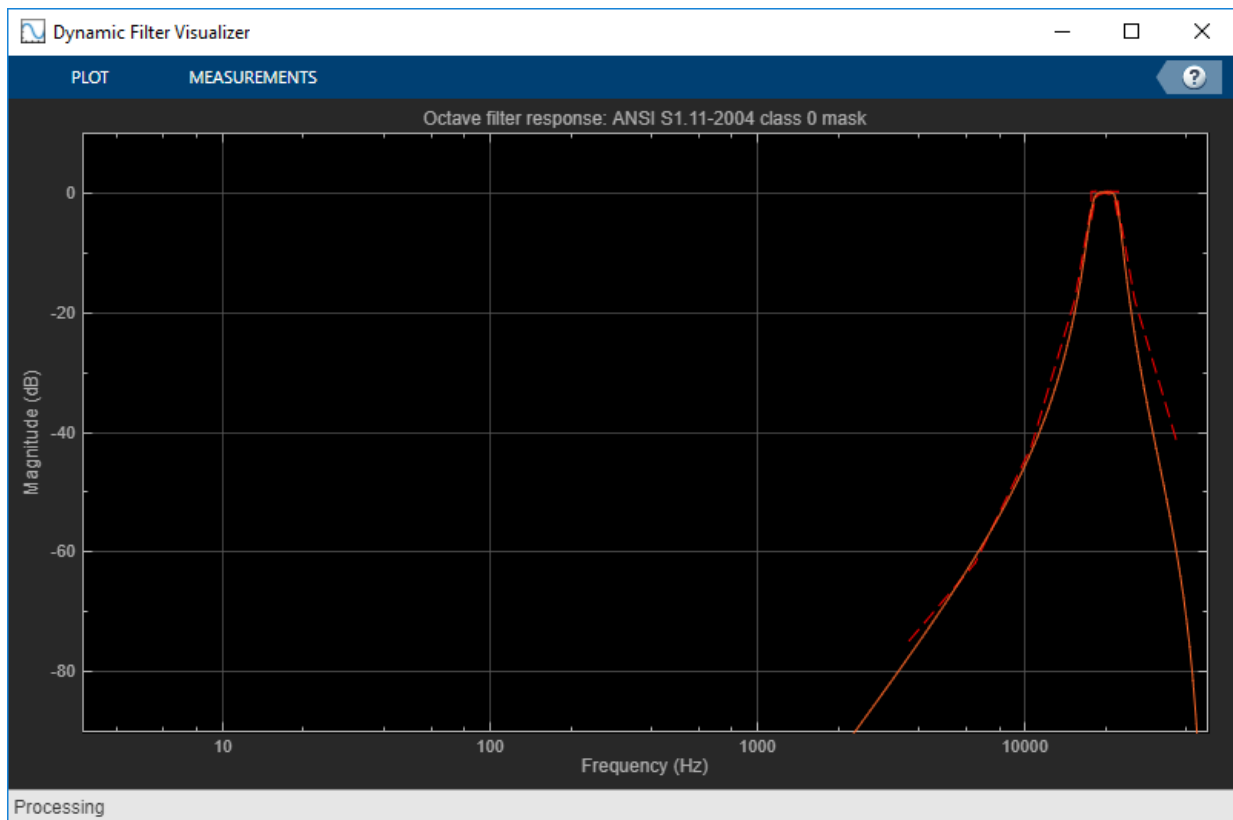
### 3 System objects in Audio Toolbox

```
centerFrequencies = 1x41  
104 ×
```

```
0.0004 0.0005 0.0006 0.0008 0.0010 0.0013 0.0016 0.0020 0.0
```

Set the center frequency of the octave filter to 19.953 kHz and visualize the response with a 'class 0' compliance mask.

```
octFilt.CenterFrequency = centerFrequencies(38);  
visualize(octFilt, 'class 0')
```



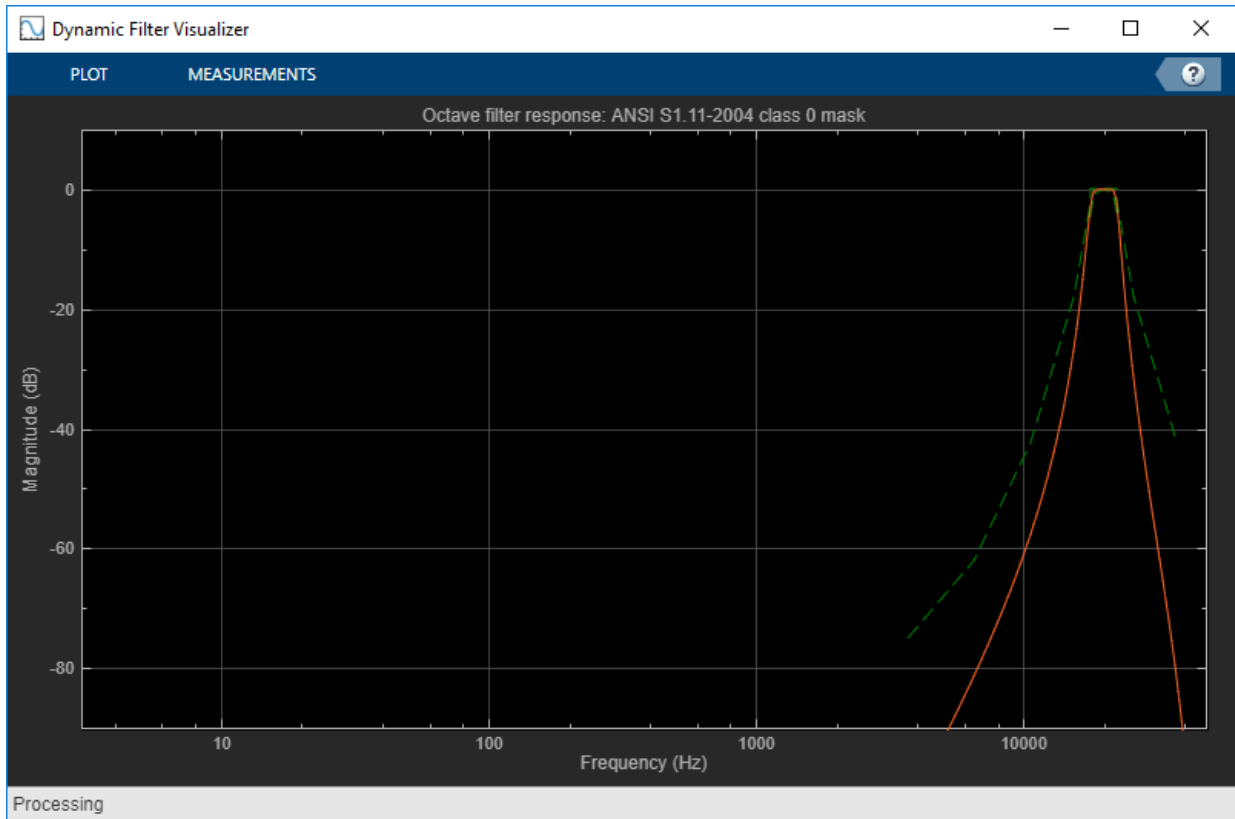
The red mask on the plot defines the bounds for the magnitude response of the filter. The magnitude response of this filter goes above the upper bound of the compliance mask



around 6.6 kHz. One way to counter this is to increase the filter order so that the filter's rolloff is steeper.

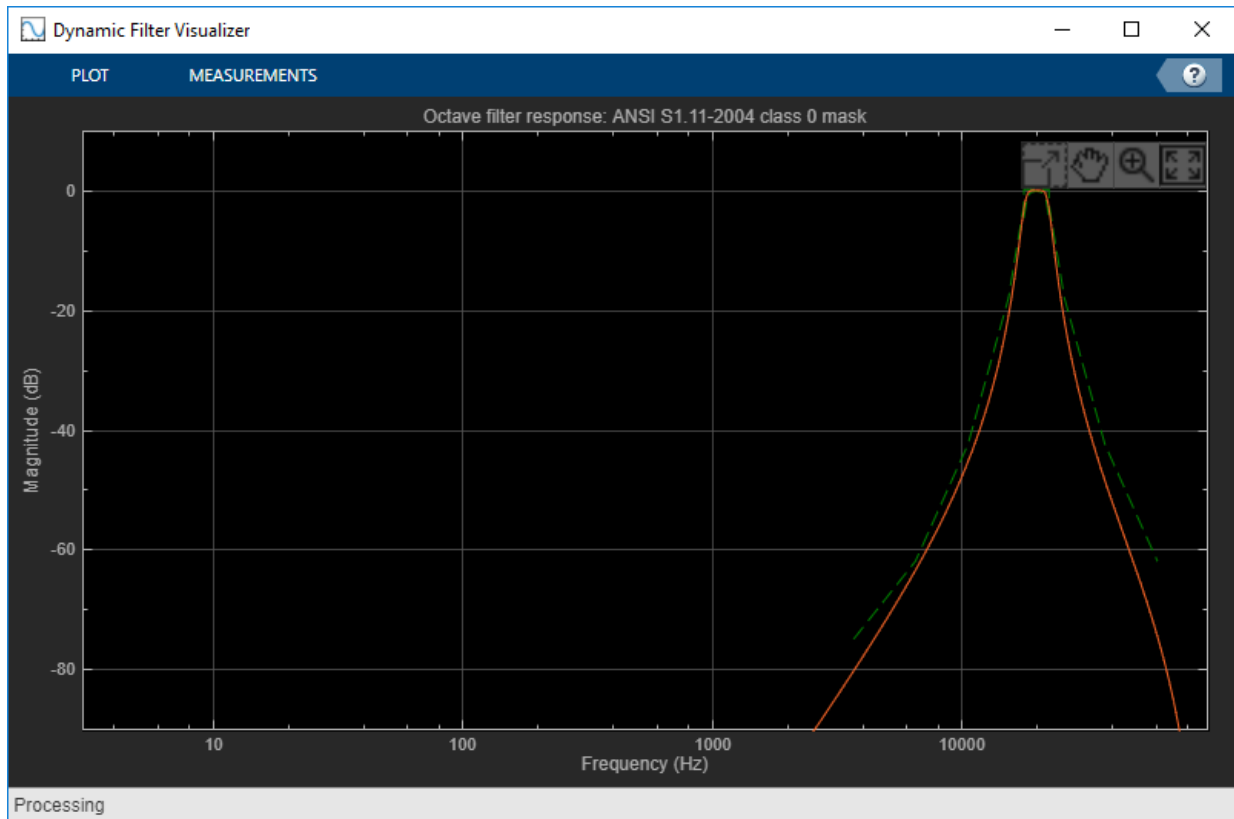
To bring the octave filter design into compliance, set the octave filter order to 8.

```
octFilt.FilterOrder = 8;
```



Another option to bring the octave filter design into compliance is to set the `Oversample` property to `true`. This designs and runs the filter at twice the specified `SampleRate` to reduce the effects of the bilinear transformation during the design stage.

```
octFilt.FilterOrder = 6;  
octFilt.Oversample = true;
```



#### Design Compliant Low-Frequency Filters

Design a sixth-order 2/3 octave filter with a 96 kHz sample rate.

```
octFilt = octaveFilter('FilterOrder',6, ...  
    'Bandwidth','2/3 octave', ...  
    'SampleRate',96e3);
```

Get the center frequencies defined by the ANSI S1.11-2004 standard. The center frequencies defined by the standard depend on the Bandwidth and SampleRate properties.

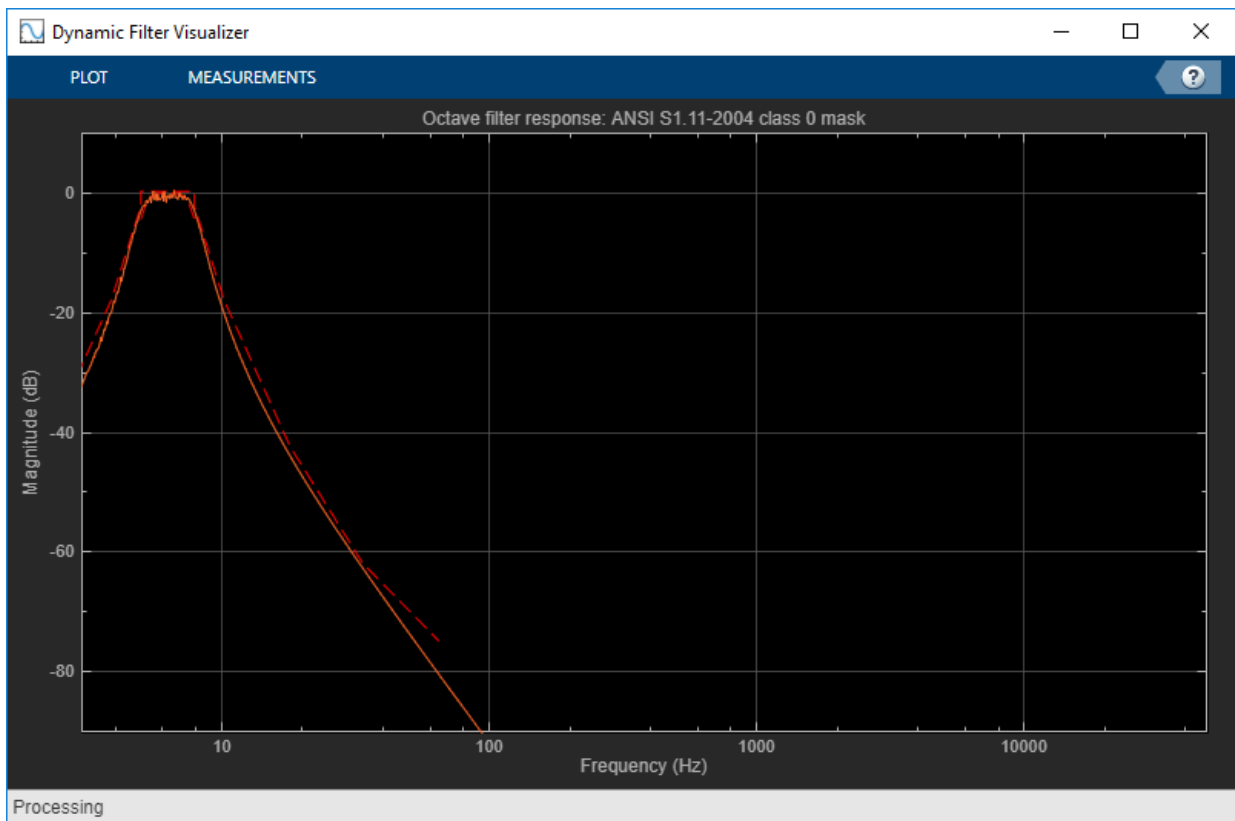
```
centerFrequencies = getANSICenterFrequencies(octFilt)
```

```
centerFrequencies = 1x20
10^4 x
```

```
0.0004 0.0006 0.0010 0.0016 0.0025 0.0040 0.0063 0.0100 0.0160 0.0250 0.0400 0.0630 0.1000 0.1600 0.2500 0.4000 0.6300 1.0000 1.6000 2.5000
```

Set the center frequency of the octave filter to ~6 Hz and visualize the response with a 'class 0' compliance mask.

```
octFilt.CenterFrequency = centerFrequencies(2);
visualize(octFilt, 'class 0')
```

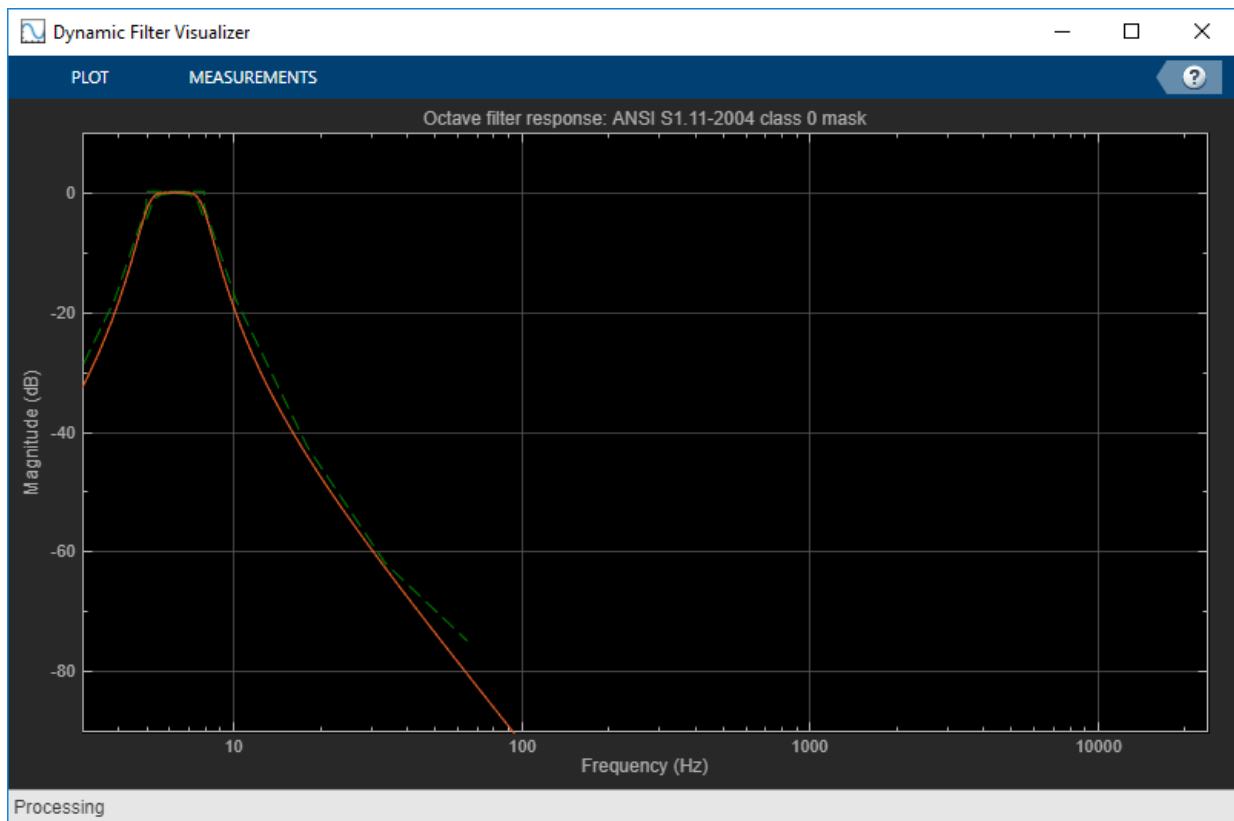


The red mask on the plot defines the bounds for the magnitude response of the filter. The magnitude response of this filter goes below the lower bound of the compliance mask between 5.5 and 7.5 Hz.

Low-frequency filters in an octave filter bank have very low normalized center frequencies, and the filters designed for them have poles that are almost on the unit circle. To make this filter ANSI compliant, it has to be designed and operated at a lower sample rate.

To bring the octave filter design into compliance, set the sample rate to 48 kHz.

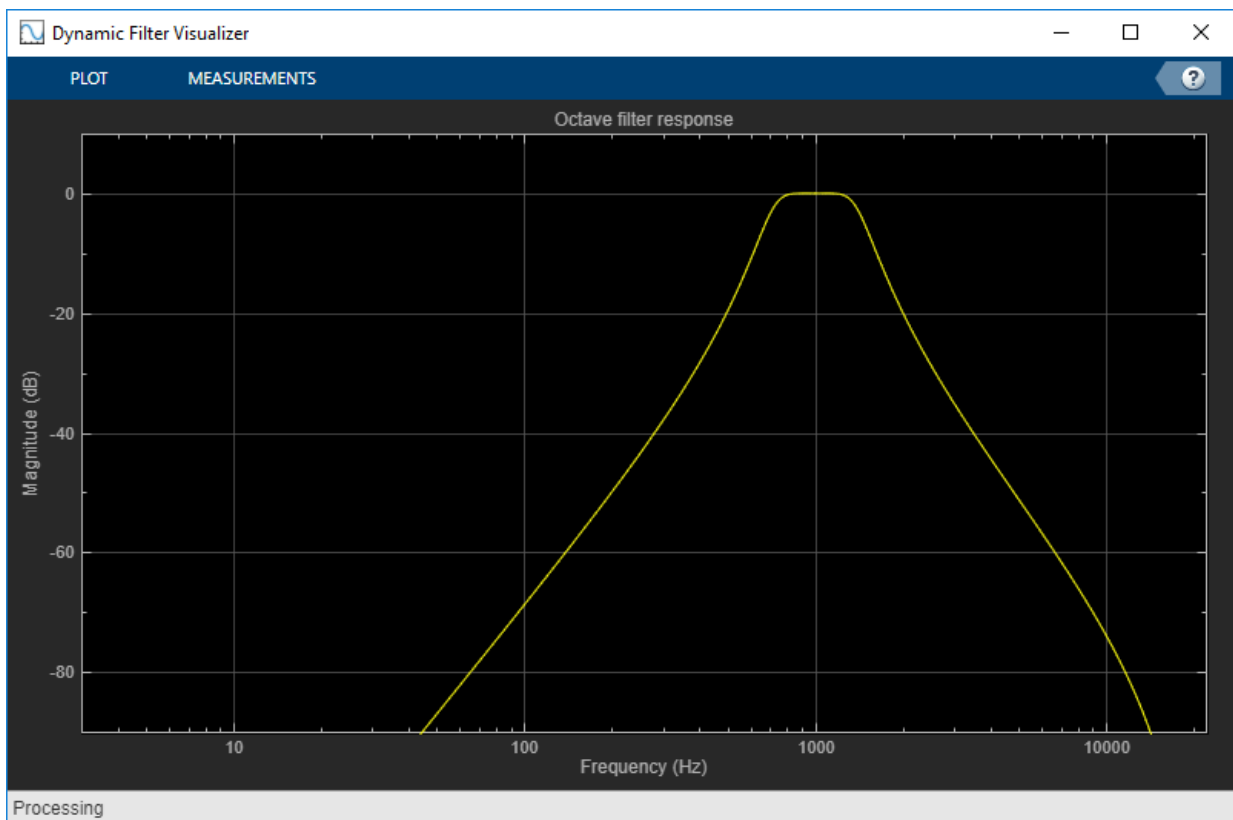
```
octFilt.SampleRate = 48e3;
```



## Tune Octave Filter Parameters

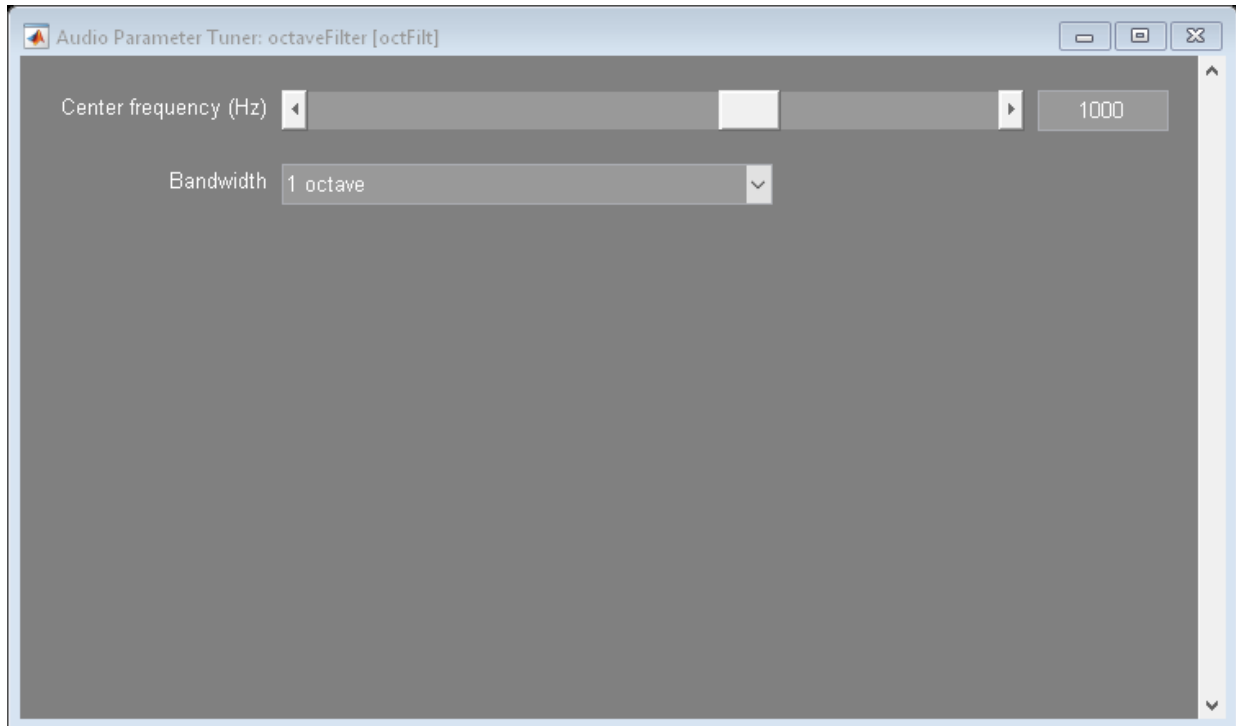
Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card. Create an `octaveFilter` to process the audio data. Call `visualize` to plot the frequency response of the octave filter.

```
frameLength = 1024;  
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3', ...  
    'SamplesPerFrame',frameLength);  
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);  
  
octFilt = octaveFilter('SampleRate',fileReader.SampleRate);  
visualize(octFilt)
```



Call `parameterTuner` to open a UI to tune parameters of the `octaveFilter` while streaming.

```
parameterTuner(octFilt)
```



In an audio stream loop:

- 1 Read in a frame of audio from the file.
- 2 Apply octave filtering.
- 3 Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the octave filter and listen to the effect.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = octFilt(audioIn);
    deviceWriter(audioOut);
```

```
drawnow limitrate % required to update parameter  
end
```

As a best practice, release your objects once done.

```
release(deviceWriter)  
release(fileReader)  
release(octFilt)
```

## More About

### Band Edge

A band edge frequency refers to the lower or upper edge of the passband of a bandpass filter.

### Center Frequency of Octave Filter

The center frequency of an octave filter is the geometric mean of the lower and upper band edge frequencies.

## Tips

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `octaveFilter` to user-facing parameters:

Property	Range	Mapping	Units
CenterFrequency	[3, 22000]	log	Hz

Property	Range	Mapping	Units
Bandwidth	'1 octave', '2/3 octave', '1/2 octave', '1/3 octave', '1/6 octave', '1/12 octave', '1/24 octave', or '1/48 octave'	Your MIDI controller range is discretized into seven levels, corresponding to the seven Bandwidth choices.	--

## Algorithms

### Octave Bandwidth to Band Edge Conversion

The `octaveFilter` System object uses the specified center frequency and filter bandwidth in octaves to determine the normalized band edges [2].

The object computes the upper and lower band edge frequencies:

$$f_{pa} = f_c \times G^{-1/2b}$$

$$f_{pb} = f_c \times G^{1/2b}$$

- $f_c$  is the normalized center frequency specified by the `CenterFrequency` property.
- $b$  is the octave bandwidth specified by the `Bandwidth` property. For example, if `Bandwidth` is specified as `'1/3 octave'`, the value of  $b$  is 3.
- $G$  is a conversion constant:

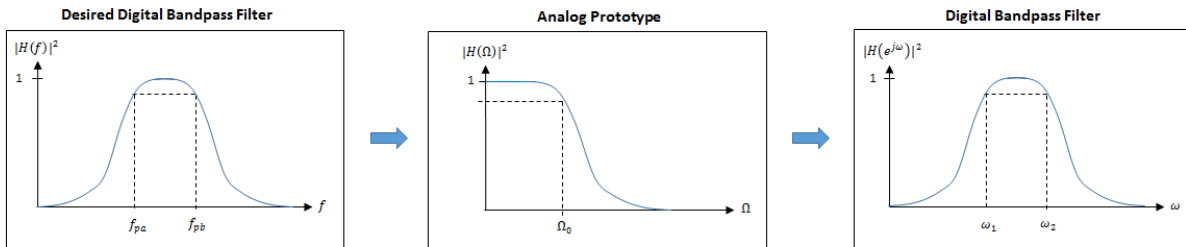
$$G = 10^{3/10}.$$

### Digital Filter Design

The `octaveFilter` System object implements a higher-order digital bandpass filter design method specified in [1].

In this design method, a desired digital bandpass filter maps to a Butterworth lowpass analog prototype, which is then mapped back to a digital bandpass filter:





- 1 The analog Butterworth filter is expressed as a cascade of second-order sections:

$$H(s) = H_1(s)H_2(s)\cdots H_{2N}(s),$$

where:

$$H_i(s) = \frac{1}{1 - 2\frac{s}{\Omega_0}\cos\theta_i + \frac{s^2}{\Omega_0^2}}, \quad i = 1, 2, \dots, 2N$$

$$\theta_i = \frac{\pi}{2N}(N - 1 + 2i), \quad i = 1, 2, \dots, 2N$$

$N$  is the filter order specified by the `FilterOrder` property.

- 2 The analog Butterworth filter is mapped to a digital filter using a bandpass version of the bilinear transformation:

$$s = \frac{1 - cz^{-1} + z^{-2}}{1 - z^{-2}},$$

where

$$c = \frac{\sin(\omega_{pa} + \omega_{pb})}{\sin\omega_{pa} + \sin\omega_{pb}}.$$

This mapping results in the following substitution:

$$\Omega_0 = \frac{c - \cos\omega_{pb}}{\sin\omega_{pb}}.$$

- 3 The analog prototype is evaluated:

$$H_i(z) = \frac{1}{1 - 2\frac{s}{\Omega_0}\cos\theta_i + \frac{s^2}{\Omega_0^2}} \bigg|_s = \frac{1 - 2cz^{-1} + z^{-2}}{1 - z^{-2}}$$

Because  $s$  is second-order in  $z$ , the bandpass version of the bilinear transformation is fourth-order in  $z$ .

## References

- [1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 2010.
- [2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters*. ANSI S1.11-2004. Melville, NY: Acoustical Society of America, 2009.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

Octave Filter | `dsp.BiquadFilter` | `multibandParametricEQ` | `octaveFilterBank` | `weightingFilter`

### Topics

“Octave-Band and Fractional Octave-Band Filters”

**Introduced in R2016b**

## getANSICenterFrequencies

Get the list of valid ANSI S1.11-2004 center frequencies

### Syntax

```
centerFrequencies = getANSICenterFrequencies(octFilt)
```

### Description

`centerFrequencies = getANSICenterFrequencies(octFilt)` returns a vector of valid center frequencies as specified by the ANSI S1.11-2004 standard.

### Examples

#### Get ANSI Center Frequencies

Create an object of the `octaveFilter` System object™. Call `getANSICenterFrequencies` to get a list of valid center frequencies.

```
octFilt = octaveFilter;  
centerFrequencies = getANSICenterFrequencies(octFilt)
```

```
centerFrequencies = 1×11  
103 ×
```

```
0.0079    0.0158    0.0316    0.0631    0.1259    0.2512    0.5012    1.0000    1.9953
```

### Input Arguments

**octFilt** — Object of `octaveFilter`  
object

Object of the octaveFilter System object.

## Output Arguments

### **centerFrequencies** — Center frequencies

vector

Center frequencies specified by the ANSI S1.11-2004 standard, returned as a vector.

The range for computing valid center frequencies is 3 Hz to  $(F_s/2)$  Hz, where the `SampleRate` property of your octave filter defines  $F_s$ .

Data Types: `single` | `double`

## See Also

### **Blocks**

Octave Filter

### **Topics**

“Octave-Band and Fractional Octave-Band Filters”

**Introduced in R2016b**

## isStandardCompliant

Verify octave filter design is ANSI S1.11-2004 compliant

### Syntax

```
complianceStatus = isStandardCompliant(octFilt, classType)
[complianceStatus, centerFreq] = isStandardCompliant(octFilt,
classType)
```

### Description

`complianceStatus = isStandardCompliant(octFilt, classType)` returns a logical scalar, `complianceStatus`, indicating whether the `complianceStatus` filter design is compliant with the ANSI S1.11-2004 standard for `classType`.

The mask used to determine compliance is centered on the nearest ANSI-compliant center frequency that ensures the center frequency of the object falls between the upper and lower band edges of the mask.

`[complianceStatus, centerFreq] = isStandardCompliant(octFilt, classType)` also returns the ANSI-compliant center frequency used to create the mask.

### Examples

#### Verify Standard Compliance

Create an object of the `octaveFilter` System object™. Call `isStandardCompliant`, specifying the compliance class type to check as the second argument.

```
octFilt = octaveFilter;
complianceStatus = isStandardCompliant(octFilt, 'class 2')

complianceStatus = logical
    1
```

## Get ANSI-Compliant Center Frequency

Create an object of the octaveFilter System object. Check the compliance to class 0 status of your object, and get the center frequency used to create the compliance mask.

```
octFilt = octaveFilter('CenterFrequency',1266);
[compliant, centerFreq] = isStandardCompliant(octFilt,'class 0')

compliant = logical
          0

centerFreq = 1000
```

## Input Arguments

**octFilt** — Object of octaveFilter  
object

Object of the octaveFilter System object.

**classType** — Compliance class type  
'class 0' | 'class 1' | 'class 2'

Compliance class type to verify, specified as 'class 0', 'class 1' or 'class 2'.

Data Types: char

## Output Arguments

**complianceStatus** — Compliance status of filter design  
scalar

Compliance status of filter design, returned as a logical scalar. The compliance status indicates whether the octFilt filter design is compliant with the ANSI S1.11-2004 standard for classType.

If your octave filter is noncompliant, try any of the following:

- Set the center frequency to one of the values returned by `getANSICenterFrequencies`
- Increase filter order
- Increase sample rate

Data Types: `logical`

### **centerFreq** — Center frequency of mask

scalar

Center frequency used to create the compliance mask, returned as a scalar.

Data Types: `single` | `double`

## **See Also**

`Octave Filter` | `dsp.BiquadFilter` | `multibandParametricEQ` | `weightingFilter`

## **Topics**

“Octave-Band and Fractional Octave-Band Filters”

**Introduced in R2016b**



# visualize

Visualize and validate filter response

## Syntax

```
visualize(octFilt)
visualize(octFilt,N)
visualize( ____,mType)
```

## Description

`visualize(octFilt)` plots the magnitude response of the octave-band filter, `octFilt`. The plot is updated automatically when properties of the object change.

`visualize(octFilt,N)` uses an N-point FFT to calculate the magnitude response.

`visualize( ____,mType)` creates a mask based on the class of filter specified by `mType`, using either of the previous syntaxes. Specify `mType` as 'class 0', 'class 1', or 'class 2'. The mask attenuation limits are defined in the ANSI S1.11-2004 standard. The mask center frequency is the ANSI standard center frequency, with band edge frequencies on either side of the `CenterFrequency` set in `octFilt`.

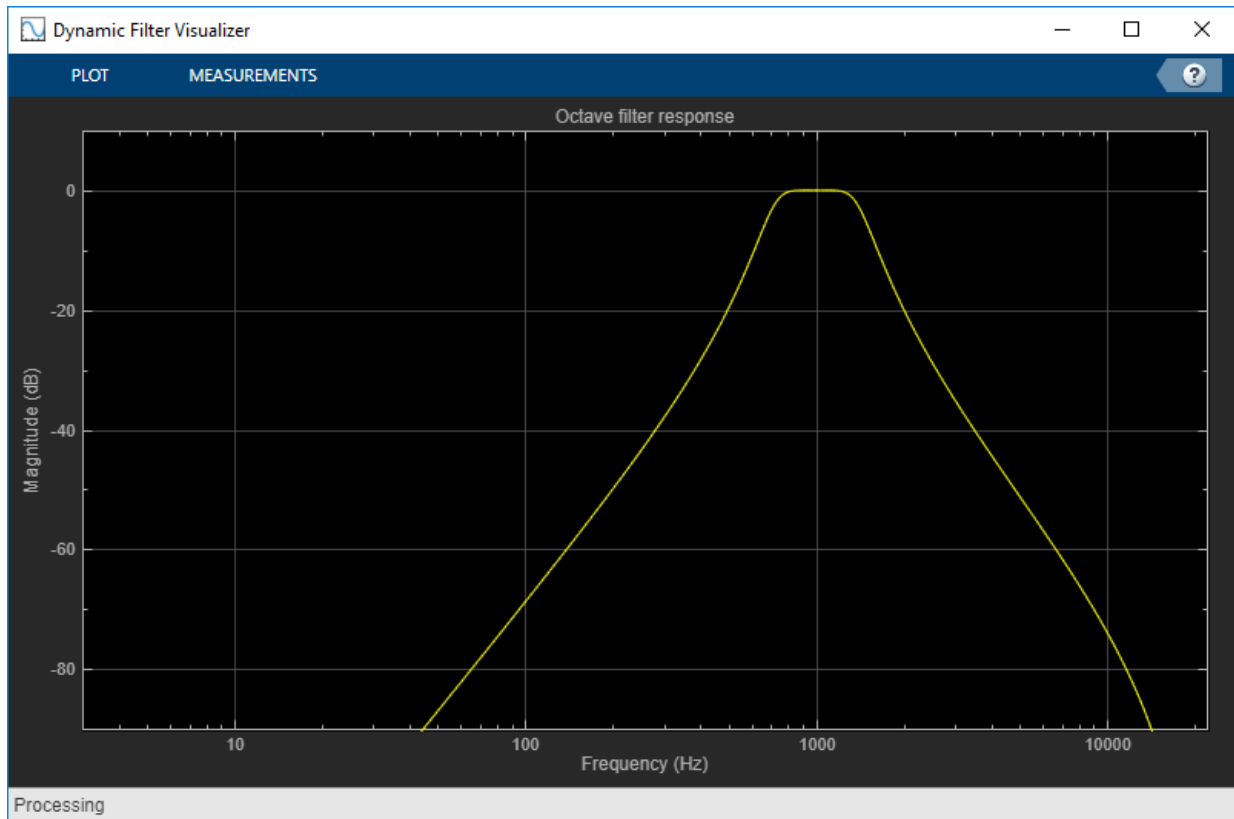
- If the mask is green, the design is compliant with the ANSI S1.11-2004 standard.
- If the mask is red, the design breaks compliance.

## Examples

### Plot Octave Filter Magnitude Response

Create an object of the `octaveFilter` System object™ and then plot the magnitude response of the filter.

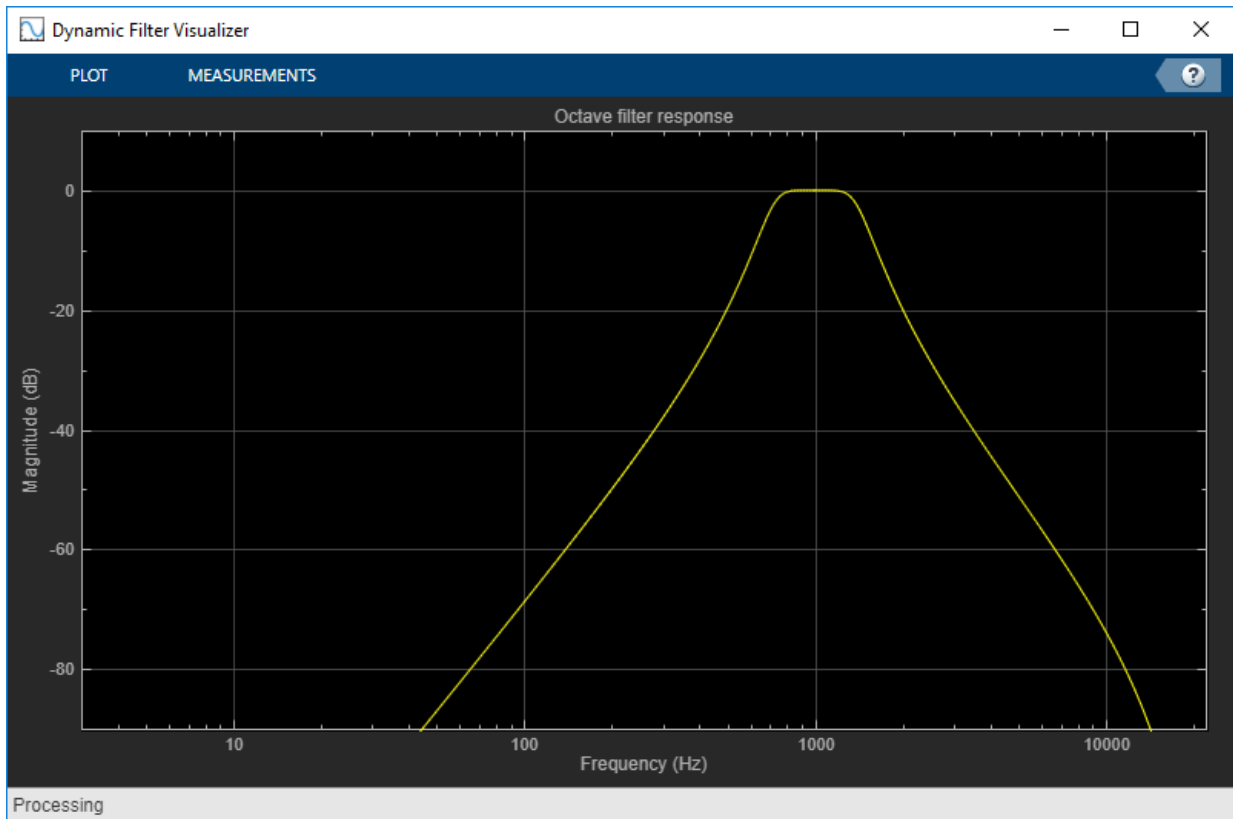
```
octFilt = octaveFilter;
visualize(octFilt)
```



#### Specify Number of Frequency Bins

Create an object of the `octaveFilter` System object™. Plot a 5096-point frequency representation.

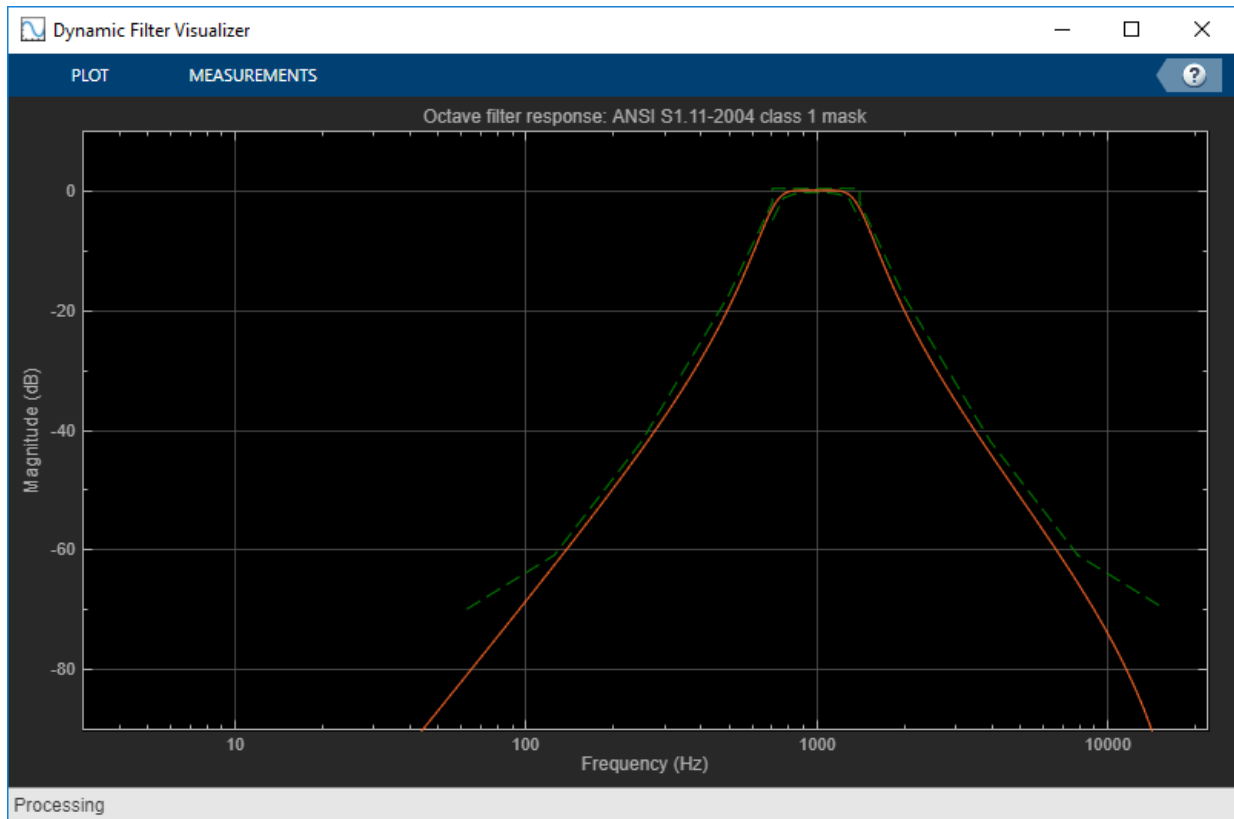
```
octFilt = octaveFilter;  
visualize(octFilt,5096)
```



### Visualize Standard-Compliance Mask

Create an object of the `octaveFilter` System object™. Visualize the class 1 compliance of the filter design.

```
octFilt = octaveFilter;  
visualize(octFilt, 'class 1')
```



## Input Arguments

**octFilt** — Object of octaveFilter object

Object of the octaveFilter System object.

**N** — Number of DFT bins  
2048 | positive scalar

Number of DFT bins in frequency-domain representation, specified as a positive scalar. The default is 2048.

---

Data Types: single | double

**mType — Type of mask**

'class 0' | 'class 1' | 'class 2'

Type of mask, specified as 'class 0', 'class 1', or 'class 2'.

The mask attenuation limits are defined in the ANSI S1.11-2004 standard. The mask center frequency is the ANSI standard center frequency, with band edge frequencies on either side of the CenterFrequency set in octFilt.

- If the mask is green, the design is compliant with the ANSI S1.11-2004 standard.
- If the mask is red, the design breaks compliance.

Data Types: char

## See Also

### Topics

“Octave-Band and Fractional Octave-Band Filters”

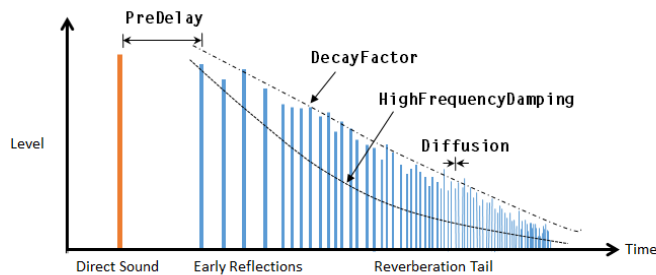
**Introduced in R2016b**

## reverberator

Add reverberation to audio signal

### Description

The reverberator System object adds reverberation to mono or stereo audio signals.



To add reverberation to your input:

- 1 Create the `reverberator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

### Creation

### Syntax

```
reverb = reverberator  
reverb = reverberator(Name,Value)
```

## Description

`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 a 0.5 second pre-delay and a wet-to-dry mix ratio of one.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects (MATLAB)*.

### **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, 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.

**Tunable:** Yes

Data Types: `single` | `double`

### **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.

**Tunable:** Yes

Data Types: `single` | `double`

### **Diffusion — Density of reverb tail**

0.5 (default) | real scalar

Density of reverb tail, specified as a real positive scalar in the range [0, 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.

**Tunable:** Yes

Data Types: `single` | `double`

### **DecayFactor — Decay factor of reverb tail**

0.5 (default) | real scalar

Decay factor of reverb tail, specified as a real positive scalar in the range [0, 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).

**Tunable:** Yes

Data Types: `single` | `double`

### **HighFrequencyDamping — High-frequency damping**

0.0005 (default) | real scalar

High-frequency damping, specified as a real positive scalar in the range [0, 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.

**Tunable:** Yes



Data Types: `single` | `double`

**WetDryMix — Wet-dry mix**

0.3 (default) | real scalar

Wet-dry mix, specified as a real positive scalar in the range [0, 1].

Wet-dry mix is the ratio of wet (reverberated) to dry (original) signal that your reverberator System object outputs.

**Tunable:** Yes

Data Types: `single` | `double`

**SampleRate — Input sample rate (Hz)**

44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: `single` | `double`

## Usage

## Syntax

```
audioOut = reverb(audioIn)
```

## Description

`audioOut = reverb(audioIn)` adds reverberation to the input signal, `audioIn`, and returns the mixed signal, `audioOut`. The type of reverberation is specified by the algorithm and properties of the reverberator System object, `reverb`.

## Input Arguments

**audioIn — Audio input to reverberator**

column vector |  $N$ -by-2 matrix

Audio input to the reverberator, specified as a column vector or two-column matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

## Output Arguments

### **audioOut** — Audio output from reverberator

*N*-by-2 matrix

Audio output from the reverberator, returned as a two-column matrix.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to reverberator

<code>createAudioPluginClass</code>	Create audio plugin class that implements functionality of System object
<code>parameterTuner</code>	Tune object parameters while streaming

### MIDI

<code>configureMIDI</code>	Configure MIDI connections between audio object and MIDI controller
<code>disconnectMIDI</code>	Disconnect MIDI controls from audio object
<code>getMIDIConnections</code>	Get MIDI connections of audio object

### Common to All System Objects

<code>clone</code>	Create duplicate System object
<code>isLocked</code>	Determine if System object is in use
<code>release</code>	Release resources and allow changes to System object property values and input characteristics
<code>reset</code>	Reset internal states of System object

step      Run System object algorithm

## Examples

### Add Reverberation to Audio Signal

Use the `reverberator` System object™ to add artificial reverberation to an audio signal read from a file.

Create the `dsp.AudioFileReader` and `audioDeviceWriter` 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',
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

Play 10 seconds of the audio signal through your device.

```
tic
while toc < 10
    audio = fileReader();
    deviceWriter(audio);
end
release(fileReader)
```

Construct a `reverberator` System object with default settings.

```
reverb = reverberator
```

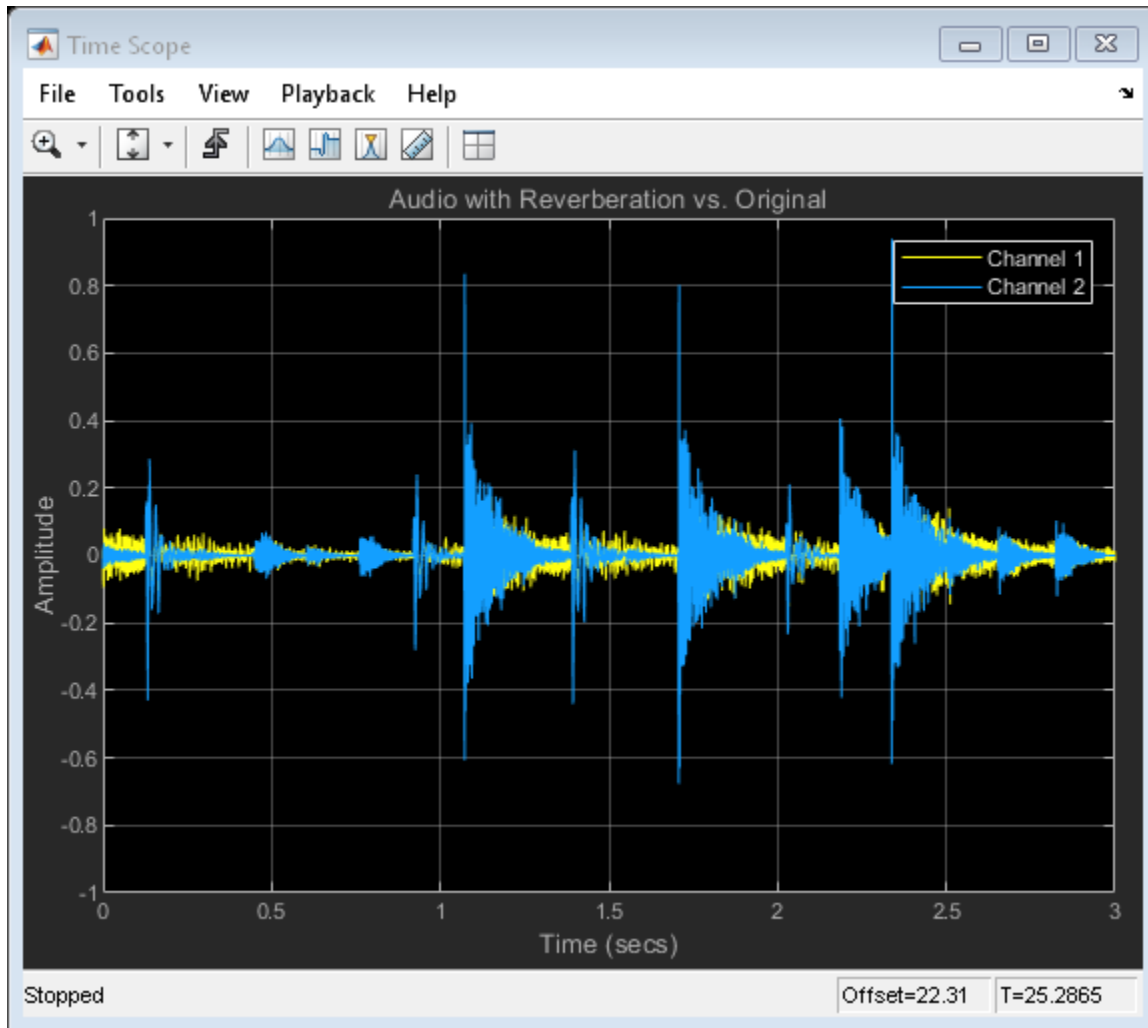
```
reverb =
    reverberator with properties:
        PreDelay: 0
        HighCutFrequency: 20000
        Diffusion: 0.5000
        DecayFactor: 0.5000
        HighFrequencyDamping: 5.0000e-04
        WetDryMix: 0.3000
        SampleRate: 44100
```

Construct a time scope to visualize the original audio signal and the audio signal with added artificial reverberation.

```
scope = dsp.TimeScope( ...  
    'SampleRate',fileReader.SampleRate, ...  
    'TimeSpanOvverrunAction','Scroll', ...  
    'TimeSpan',3, ...  
    'BufferLength',3*fileReader.SampleRate*2, ...  
    '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 = fileReader();  
    audioWithReverb = reverb(audio);  
    deviceWriter(audioWithReverb);  
    scope([audioWithReverb(:,1),audio(:,1)])  
end  
release(fileReader)  
release(deviceWriter)  
release(scope)
```



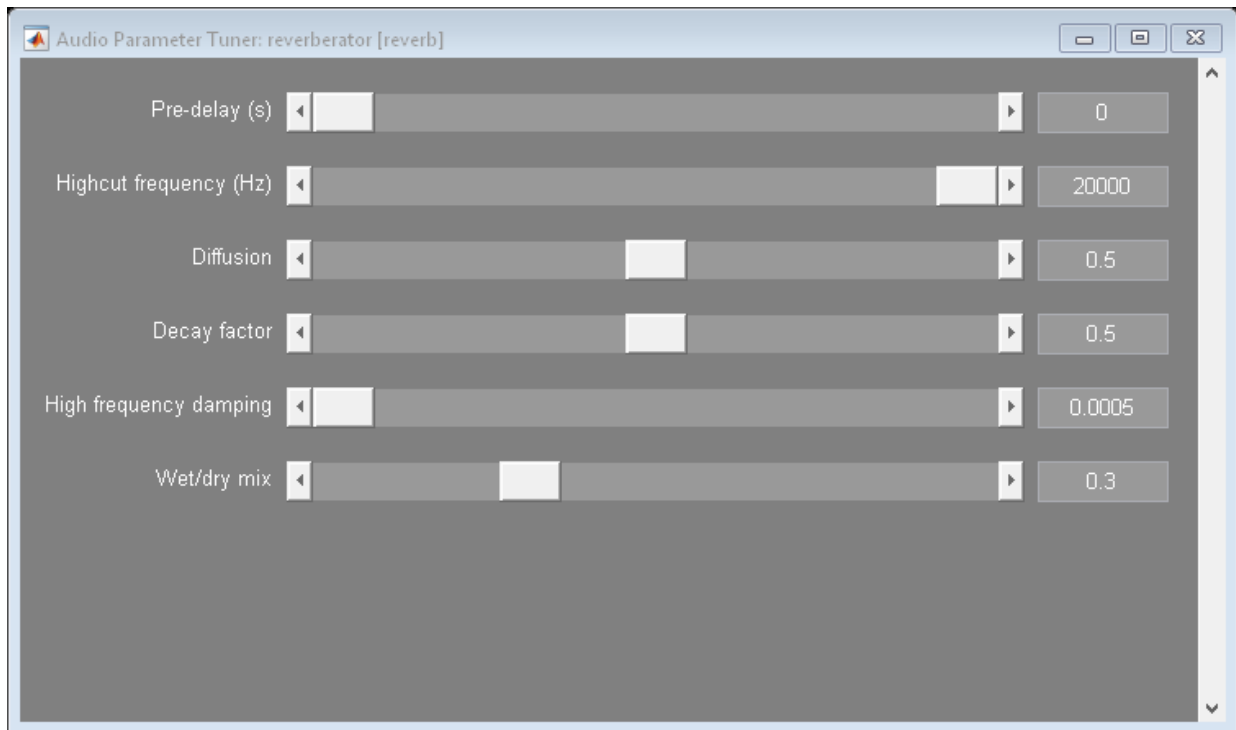
### Tune Reverberator Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card. Create a `reverberator` to process the audio data.

```
frameLength = 1024;  
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3', ...  
    'SamplesPerFrame',frameLength,'PlayCount',2);  
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);  
reverb = reverberator('SampleRate',fileReader.SampleRate);
```

Call `parameterTuner` to open a UI to tune parameters of the `octaveFilter` while streaming.

```
parameterTuner(reverb)
```



In an audio stream loop:

- 1 Read in a frame of audio from the file.
- 2 Apply reverberation.
- 3 Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the reverberator and listen to the effect.

```

while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = reverb(audioIn);
    deviceWriter(audioOut);
    drawnow limitrate % required to update parameter
end

```

As a best practice, release your objects once done.

```

release(deviceWriter)
release(fileReader)
release(reverb)

```

## Tips

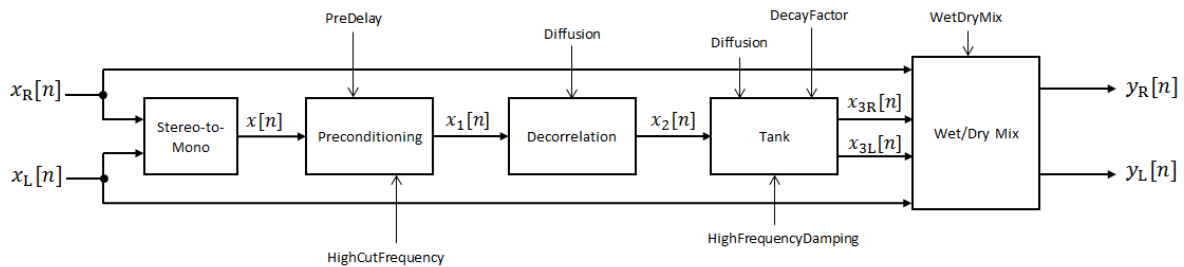
The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the compressor to user-facing parameters:

Property	Range	Mapping	Unit
PreDelay	[0, 1]	linear	s
HighCutFrequency	[20, 20000]	log	Hz
Diffusion	[0, 1]	linear	none
DecayFactor	[0, 1]	linear	none
HighFrequencyDamping	[0, 1]	linear	none
WetDryMix	[0, 1]	linear	none

## Algorithms

The algorithm to add reverberation follows the plate-class reverberation topology described in [1] and is based on a 29,761 Hz sample rate.

The algorithm has five stages.



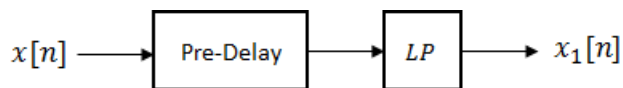
The description for the algorithm that follows is for a stereo input. A mono input is a simplified case.

### Stereo-to-Mono

A stereo signal is converted to a mono signal:  $x[n] = 0.5 \times (x_R[n] + x_L[n])$ .

### Preconditioning

A delay followed by a lowpass filter preconditions the mono signal.



- The pre-delay output is determined as  $x_p[n] = x[n - k]$ , where the `PreDelay` property determines the value of  $k$ .
- The signal is fed through a single-pole lowpass filter with transfer function

$$LP(z) = \frac{1 - \alpha}{1 - \alpha z^{-1}},$$

where

$$\alpha = \exp\left(-2\pi \times \frac{f_c}{f_s}\right).$$

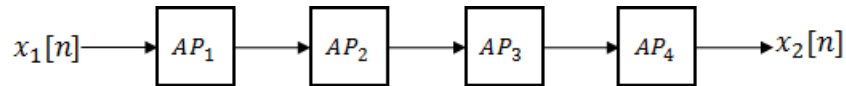
- $f_c$  is the cutoff frequency specified by the `HighCutFrequency` property.



- $f_s$  is the sampling frequency specified by the `SampleRate` property.

## Decorrelation

The signal is decorrelated by passing through a series of four allpass filters.



The allpass filters are of the form

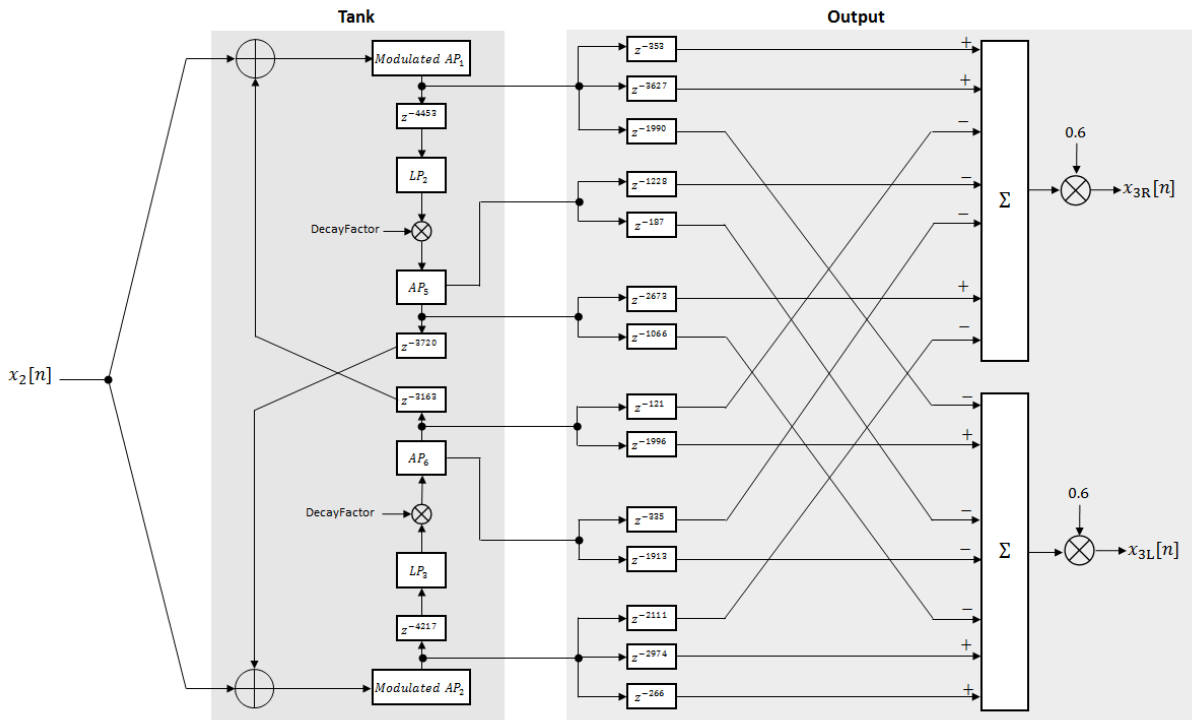
$$AP(z) = \frac{\beta + z^{-k}}{1 + \beta z^{-k}},$$

where  $\beta$  is the coefficient specified by the `Diffusion` property and  $k$  is the delay as follows:

- For  $AP_1$ ,  $k = 142$ .
- For  $AP_2$ ,  $k = 107$ .
- For  $AP_3$ ,  $k = 379$ .
- For  $AP_4$ ,  $k = 277$ .

## Tank

The signal is fed into the tank, where it circulates to simulate the decay of a reverberation tail.



The following description tracks the signal as it progresses through the top of the tank. The signal progression through the bottom of the tank follows the same pattern, with different delay specifications.

- 1 The new signal enters the top of the tank and is added to the circulated signal from the bottom of the tank.
- 2 The signal passes through a modulated allpass filter:

$$\text{Modulated } AP_1(z) = \frac{-\beta + z^{-k}}{1 - \beta z^{-k}}$$

- $\beta$  is the coefficient specified by the Diffusion property.
- $k$  is the variable delay specified by a 1 Hz sinusoid with amplitude =  $(8/29761) * \text{SampleRate}$ . To account for fractional delay resulting from the modulating  $k$ , allpass interpolation is used [2].

- 3 The signal is delayed again, and then passes through a lowpass filter:

$$LP_2(z) = \frac{1 - \varphi}{1 - \varphi z^{-1}}$$

- $\varphi$  is the coefficient specified by the `HighFrequencyDamping` property.

- 4 The signal is multiplied by a gain specified by the `DecayFactor` property. The signal then passes through an allpass filter:

$$AP_5(z) = \frac{\beta + z^{-k}}{1 + \beta z^{-k}}.$$

- $\beta$  is the coefficient specified by the `Diffusion` property.
- $k$  is set to 1800 for the top of the tank and 2656 for the bottom of the tank.

- 5 The signal is delayed again and then circulated to the bottom half of the tank for the next iteration.

A similar pattern is executed in parallel for the bottom half of the tank. The output of the tank is calculated as the signed sum of delay lines picked off at various points from the tank. The summed output is multiplied by  $\theta$ . 6.

## Wet/Dry Mix

The wet (processed) signal is then added to the dry (original) signal:

$$y_R[n] = (1 - \kappa)x_R[n] + \kappa x_{3R}[n],$$

$$y_L[n] = (1 - \kappa)x_L[n] + \kappa x_{3L}[n],$$

where the `WetDryMix` property determines  $\kappa$ .

## References

- [1] Dattorro, Jon. "Effect Design, Part 1: Reverberator and Other Filters." *Journal of the Audio Engineering Society*. Vol. 45, Issue 9, 1997, pp. 660-684.
- [2] Dattorro, Jon. "Effect Design, Part 2: Delay-Line Modulation and Chorus." *Journal of the Audio Engineering Society*. Vol. 45, Issue 10, 1997, pp. 764-788.

## **Extended Capabilities**

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

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### **See Also**

Reverberator

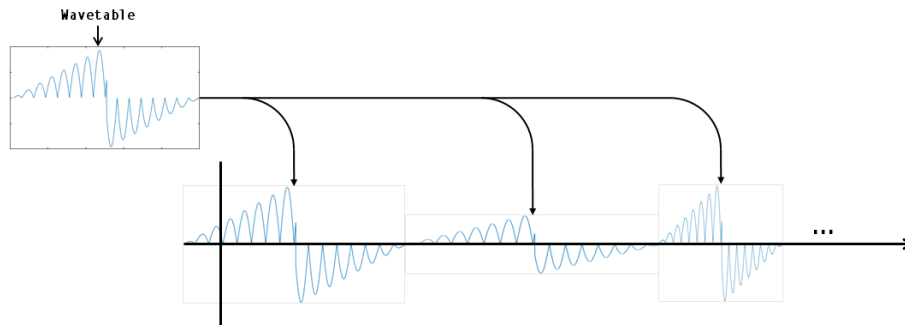
**Introduced in R2016a**

# wavetableSynthesizer

Generate periodic signal 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 Create the `wavetableSynthesizer` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

## Syntax

```
waveSynth = wavetableSynthesizer  
waveSynth = wavetableSynthesizer(wavetableValue)
```

```
waveSynth = wavetableSynthesizer(wavetableValue, frequencyValue)  
waveSynth = wavetableSynthesizer( ____, Name, Value)
```

## Description

`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 the default sine waveform with an amplitude of 2 and a DC offset of 2.5.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects \(MATLAB\)](#).

### 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.

**Tunable:** Yes

Data Types: single | double

**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.

**Tunable:** Yes

Data Types: single | double

**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 **DCOffset** is applied.

**Tunable:** Yes

Data Types: single | double

**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, 1]. The range is a normalized  $2\pi$  radians interval.

**Tunable:** No

Data Types: single | double

**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.

**Tunable:** Yes

Data Types: `single` | `double`

### **SamplesPerFrame — Number of samples per frame**

512 (default) | positive integer

Number of samples per frame, specified as a positive integer in the range [1, 192000].

This property determines the vector length that your `wavetableSynthesizer` object outputs.

**Tunable:** Yes

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

### **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.

**Tunable:** Yes

### **OutputDataType — Data type of generated signal**

'double' (default) | 'single'

Data type of generated signal, specified as 'double' or 'single'.

**Tunable:** No

Data Types: `char` | `string`

## Usage

## Syntax

```
waveform = waveSynth()
```



## Description

`waveform = waveSynth()` generates a periodic signal, `waveform`. The type of signal is specified by the algorithm and properties of the `wavetableSynthesizer` System object, `waveSynth`.

## Output Arguments

### **waveform** — Waveform output from wavetable synthesizer

column vector (default)

Waveform output from the wavetable synthesizer, returned as a column vector with length specified by the `SamplesPerFrame` property and data type specified by the `OutputDataType` property.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### **Specific to wavetableSynthesizer**

<code>createAudioPluginClass</code>	Create audio plugin class that implements functionality of System object
<code>parameterTuner</code>	Tune object parameters while streaming

### **MIDI**

<code>configureMIDI</code>	Configure MIDI connections between audio object and MIDI controller
<code>disconnectMIDI</code>	Disconnect MIDI controls from audio object
<code>getMIDIConnections</code>	Get MIDI connections of audio object

### **Common to All System Objects**

<code>clone</code>	Create duplicate System object
--------------------	--------------------------------

isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `wavetableSynthesizer` System object to user-facing parameters:

Property	Range	Mapping	Unit
Frequency	[0.1, 20000]	log	Hz
Amplitude	[0, 10]	linear	none
DCOffset	[-10, 10]	linear	none

## Examples

### Generate Variable-Frequency Staircase Wave

Define and plot a single-cycle waveform.

```
values = -1:0.1:1;
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',0.1, ...
    '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 = waveSynth();
    scope(staircaseWave)
    if mod(counter,1000)==0
        waveSynth.Frequency = waveSynth.Frequency + 10;
    end
end

```

### Manipulate Audio Samples Using Wavetable Synthesizer

Sample an audio file and save it to the `Wavetable` property of a `wavetableSynthesizer` System object™. Use the wavetable synthesizer to manipulate your audio sample.

Read in an entire audio file. Clip out an interesting sound from the audio and then play it.

```

[audio,fs] = audioread('MainStreet0ne-24-96-stereo-63secs.wav');

engine = audio(5.35e6:5.45e6);
sound(engine,fs)

```

Create a wavetable synthesizer using your audio clip. The duration of the engine audio clip is `numel(engine)/fs` seconds. In the `wavetableSynthesizer`, set the `Frequency` property to `1/(clip duration)`. The generated signal now plays back at the same rate it was recorded at.

```

duration = numel(engine)/fs;
waveSynth = wavetableSynthesizer('Wavetable',engine,'SampleRate',fs, ...
    'Frequency',1/duration);

```

Create an `audioDeviceWriter` to write to your audio device.

```

deviceWriter = audioDeviceWriter('SampleRate',fs);

```

In a loop, play the wavetable synthesizer to your device. After three seconds, begin increasing the frequency of the wavetable synthesizer. After six seconds, begin decreasing the frequency of the wavetable synthesizer.

```
timeElapsed = 0;
while timeElapsed < 9
    audioWave = waveSynth();
    deviceWriter(audioWave);

    if (timeElapsed > 3) && (timeElapsed < 6)
        waveSynth.Frequency = waveSynth.Frequency + 0.001;
    elseif timeElapsed > 6
        waveSynth.Frequency = waveSynth.Frequency - 0.002;
    end

    timeElapsed = timeElapsed + waveSynth.SamplesPerFrame*(1/fs);
end
```

#### Modify Wavetable While Stream Processing

Modify the `Wavetable` property of a `wavetableSynthesizer` object while stream processing. Visualize the wavetable and play the resulting audio.

Create a single-cycle waveform for the `wavetableSynthesizer` to index into. Create a `wavetableSynthesizer` object.

```
t = 0:0.001:1;
exponent = 5;
waveTable = [t.^exponent,fliplr(t.^exponent)] - 0.5;

waveSynth = wavetableSynthesizer('Wavetable',waveTable);
```

Create a `dsp.ArrayPlot` object to plot the wavetable as it is modified over time. Create an `audioDeviceWriter` object to listen to the signal output by your `wavetableSynthesizer`.

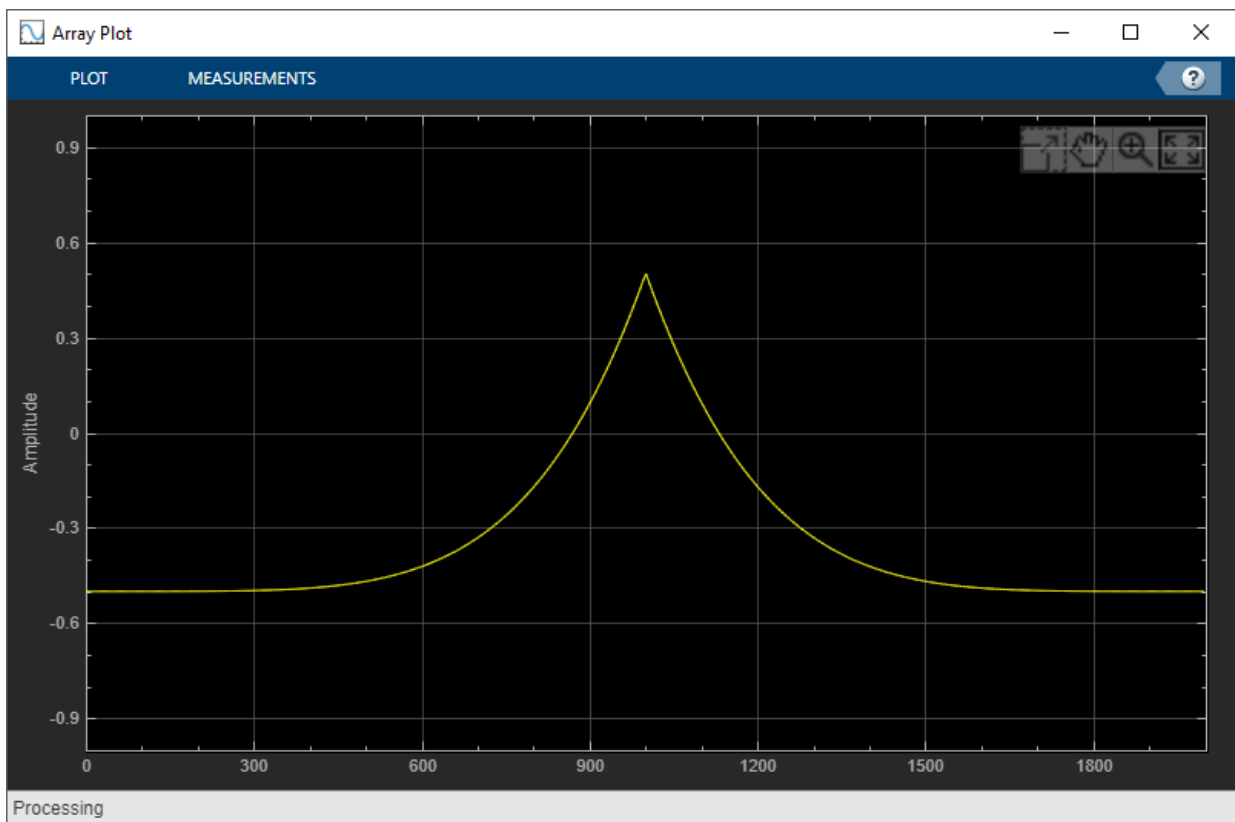
```
arrayPlotter = dsp.ArrayPlot('YLimits',[-1,1],'PlotType','Line');
deviceWriter = audioDeviceWriter;
```

In an audio stream loop, incrementally modify the `Wavetable` property of the `wavetableSynthesizer` and plot it. Call the synthesizer to output a waveform and play the waveform to your audio device.

```
tic
while toc < 10
    exponent = exponent - 0.01;
    waveSynth.Wavetable = [t.^abs(exponent),fliplr(t.^abs(exponent))] - 0.5;

    arrayPlotter(waveSynth.Wavetable')

    deviceWriter(waveSynth());
end
release(deviceWriter)
```



#### Tune Wavetable Synthesizer Parameters

Create a `wavetableSynthesizer` to generate a waveform. Create a `dsp.TimeScope` to visualize the waveform. Create an `audioDeviceWriter` to write audio to your sound card.

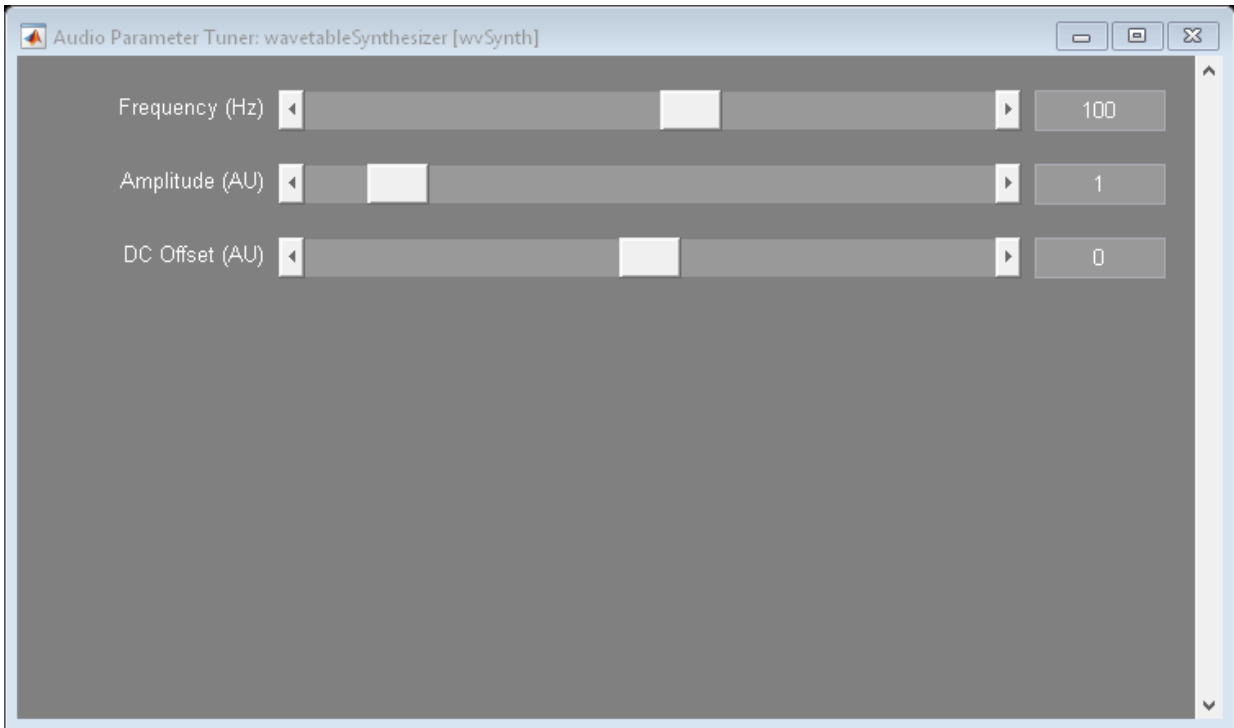
```
fs = 44.1e3;
wvSynth = wavetableSynthesizer('SampleRate',fs);

scope = dsp.TimeScope( ...
    'SampleRate',wvSynth.SampleRate, ...
    'TimeSpan',1, ...
    'YLimits',[-2,2], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true);

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

Call `parameterTuner` to open a UI to tune parameters of the wavetable synthesizer while streaming.

```
parameterTuner(wvSynth)
```



In an audio stream loop:

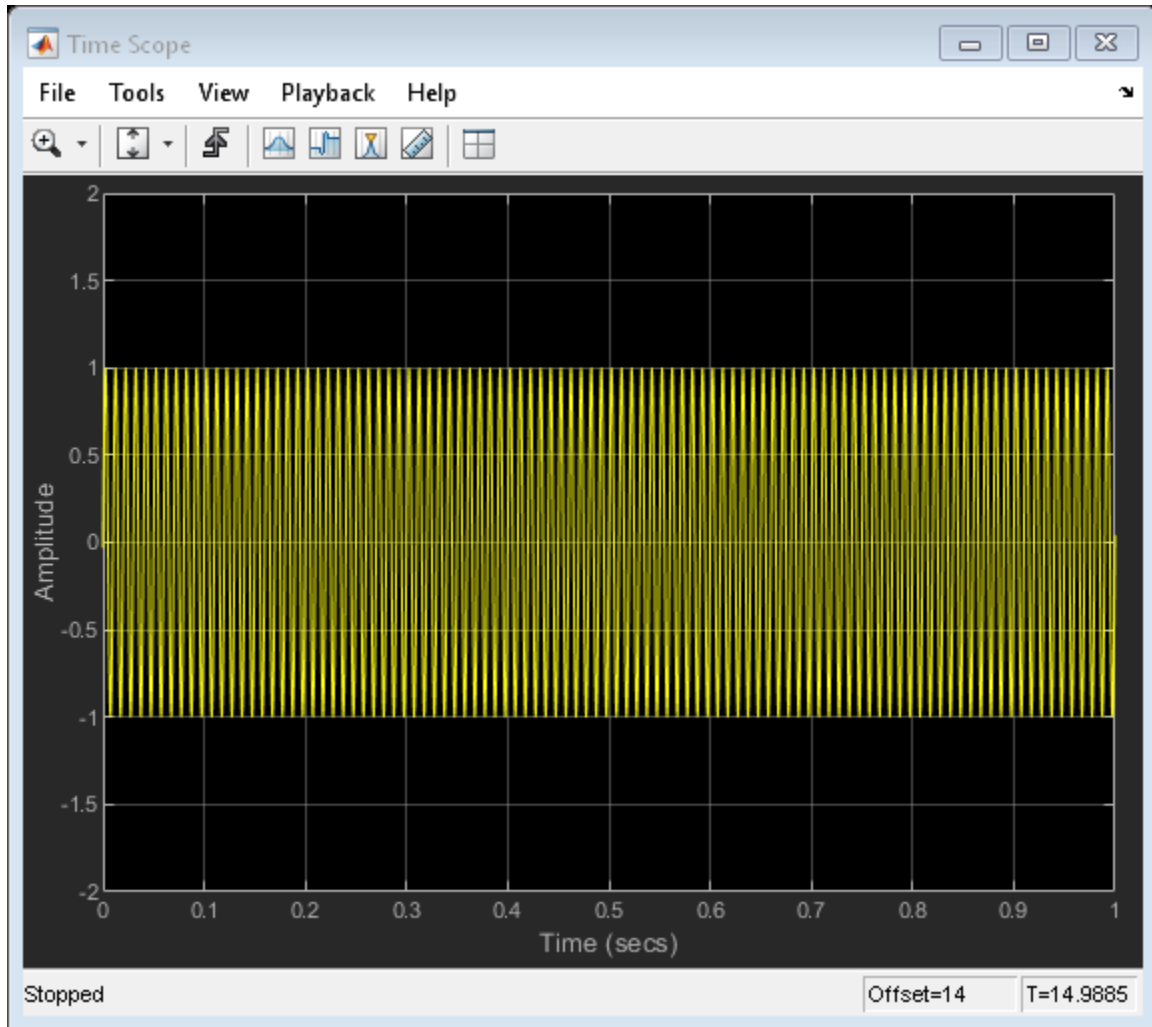
- 1 Call the wavetable synthesizer without arguments to output one frame of data.
- 2 Visualize the data using the time scope.
- 3 Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the wavetable synthesizer and listen to the effect.

```
duration = 15;
numIterations = round(wvSynth.SampleRate*duration/wvSynth.SamplesPerFrame);
for i = 1:numIterations
    audioOut = wvSynth();
    scope(audioOut)
    deviceWriter(audioOut);
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

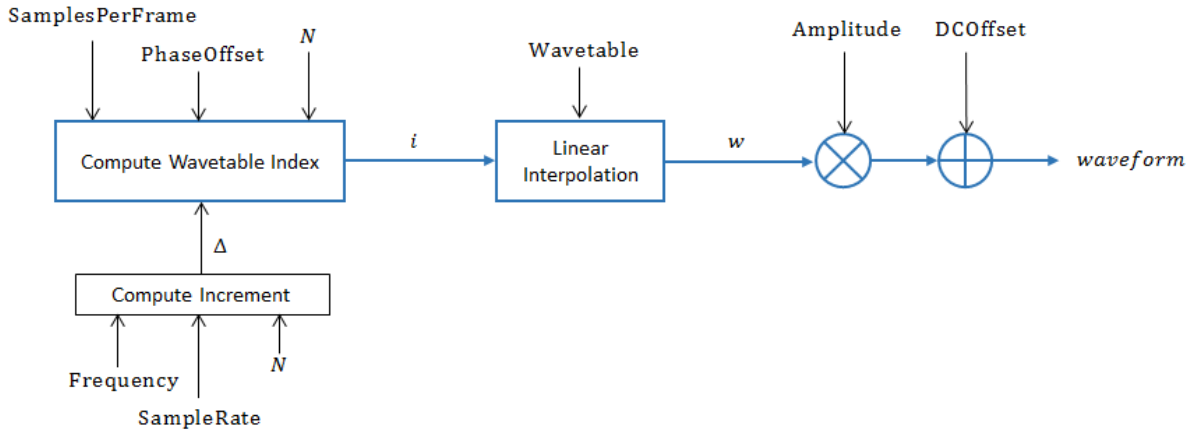
```
release(deviceWriter)  
release(wvSynth)  
release(scope)
```





## Algorithms

The `wavetableSynthesizer` System object synthesizes periodic signals using a cached single-cycle waveform, specified waveform properties, and phase memory.



### Compute Increment

Compute the increment step size:

$$\Delta = \frac{\text{Frequency}}{\text{SampleRate}} \times N,$$

where  $N$  is the number of elements in your wavetable.

### Compute Wavetable Index

Compute Wavetable index,

$$i[n] = \begin{cases} i[n-1] + \Delta & i[n-1] < N \\ i[n-1] + \Delta - N & i[n-1] \geq N \end{cases}$$

for  $2 \leq n \leq \text{SamplesPerFrame}$ . The `PhaseOffset` property determines  $i[n=1]$ .

## Linear Interpolation

Index into the `Wavetable` and perform linear interpolation:

$$w = \begin{cases} (\text{Wavetable}[1] - \text{Wavetable}[i_L]) \times \varepsilon + \text{Wavetable}[i_L] & i_H > N \\ (\text{Wavetable}[i_H] - \text{Wavetable}[i_L]) \times \varepsilon + \text{Wavetable}[i_L] & i_H \leq N \end{cases}$$

- $i_L = \text{floor}(i[n] + 1)$
- $i_H = i_L + 1$
- $\varepsilon = i - \text{floor}(i)$

## Apply Amplitude and DC Offset

Multiply by `Amplitude` and add `DCOffset`.

$$\text{waveform} = w \times \text{Amplitude} + \text{DCOffset}$$

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

`audioOscillator`

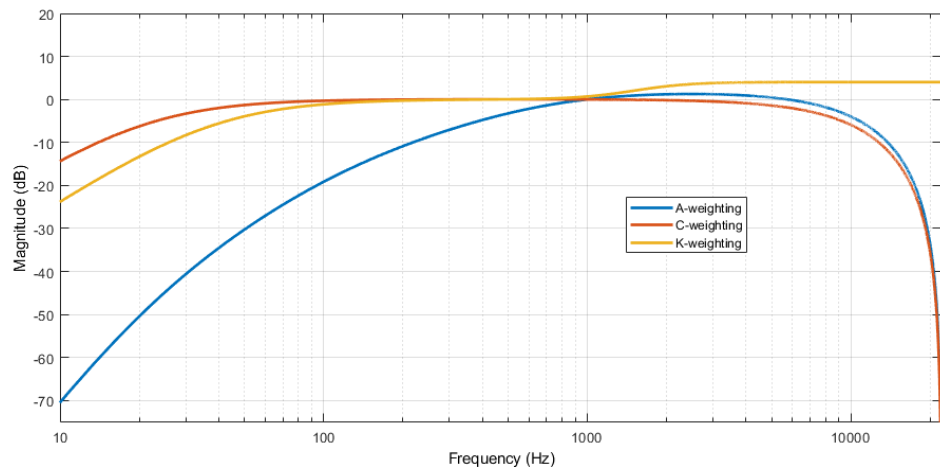
**Introduced in R2016a**

# weightingFilter

Frequency-weighted filter

## Description

The `weightingFilter` System object performs frequency-weighted filtering independently across each input channel.



To perform frequency-weighted filtering:

- 1 Create the `weightingFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

### Syntax

```
weightFilt = weightingFilter  
weightFilt = weightingFilter(weightType)  
weightFilt = weightingFilter(weightType,Fs)  
weightFilt = weightingFilter( ____,Name,Value)
```

### Description

`weightFilt = weightingFilter` creates a System object, `weightFilt`, that performs frequency-weighted filtering independently across each input channel.

`weightFilt = weightingFilter(weightType)` sets the `Method` property to `weightType`.

`weightFilt = weightingFilter(weightType,Fs)` sets the `SampleRate` property to `Fs`.

`weightFilt = weightingFilter( ____,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `weightFilt = weightingFilter('C-weighting','SampleRate',96000)` creates a C-weighting filter with a sample rate of 96,000 Hz.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects (MATLAB)*.

**Method — Type of weighting**`'A-weighting' (default) | 'C-weighting' | 'K-weighting'`

Type of weighting, specified as 'A-weighting', 'C-weighting', or 'K-weighting'. See “Algorithms” on page 3-479 for more information.

**Tunable:** No

Data Types: char | string

**SampleRate — Input sample rate (Hz)**`44100 (default) | positive scalar`

Input sample rate in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: single | double

## Usage

## Syntax

```
audioOut = weightFilt(audioIn)
```

## Description

`audioOut = weightFilt(audioIn)` applies frequency-weighted filtering to the input signal, `audioIn`, and returns the filtered signal, `audioOut`. The type of filtering is specified by the algorithm and properties of the `weightingFilter` System object, `weightFilt`.

## Input Arguments

**audioIn — Audio input to weighting filter**`matrix`

Audio input to the weighting filter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

## Output Arguments

### **audioOut** — Audio output from weighting filter

matrix

Audio output from the weighting filter, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `weightingFilter`

<code>visualize</code>	Visualize and validate filter response
<code>getFilter</code>	Return biquad filter object with design parameters set
<code>createAudioPluginClass</code>	Create audio plugin class that implements functionality of System object
<code>isStandardCompliant</code>	Verify filter design is IEC 61672-1:2002 compliant

### Common to All System Objects

<code>clone</code>	Create duplicate System object
<code>isLocked</code>	Determine if System object is in use
<code>release</code>	Release resources and allow changes to System object property values and input characteristics
<code>reset</code>	Reset internal states of System object
<code>step</code>	Run System object algorithm

---

**Note** `weightingFilter` supports additional filter analysis functions. See “Compare and Analyze Weighting Types” on page 3-473 for details.

---

## Examples

### Validate Filter Compliance

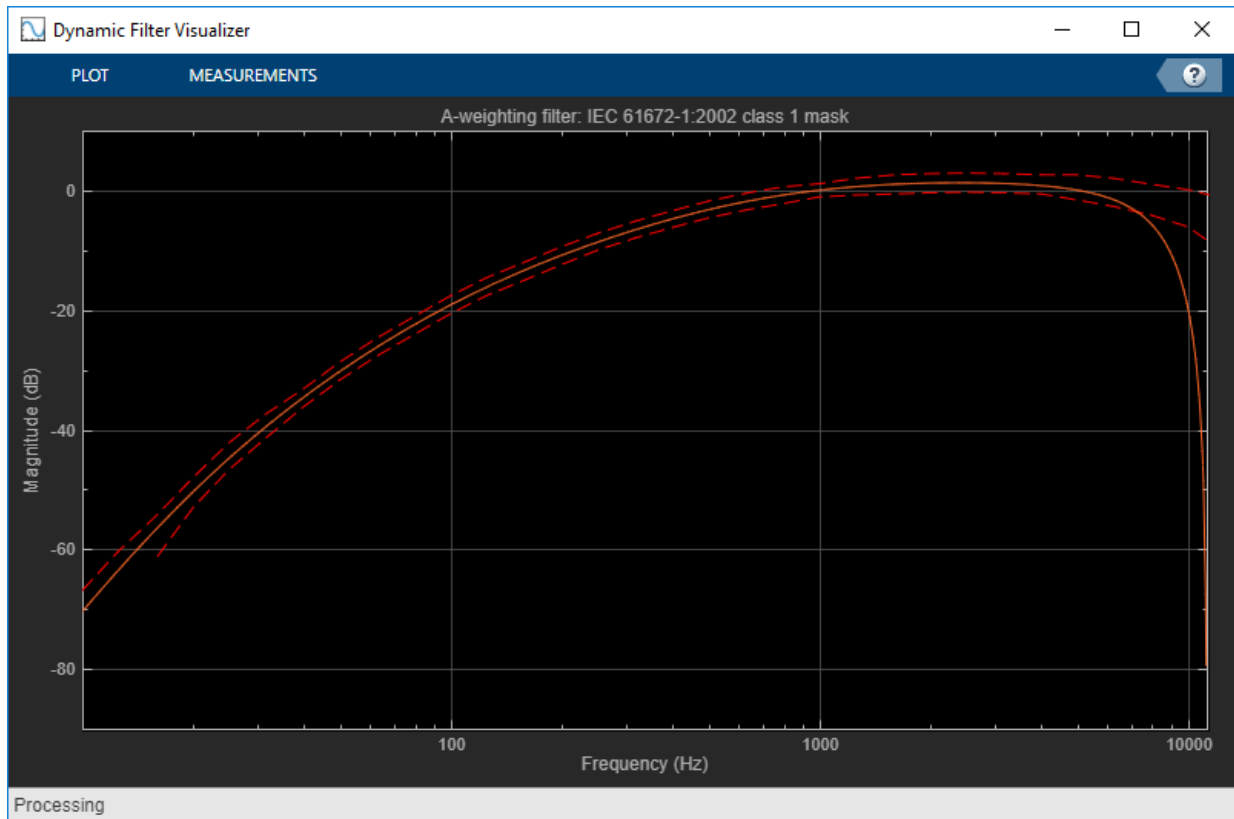
Check the compliance status of filter designs and visualize them.

Create an A-weighting filter with a 22.5 kHz sample rate. Verify that the filter is standard compliant and visualize the filter design.

```
aWeight = weightingFilter('A-weighting','SampleRate',22500);  
complianceStatus = isStandardCompliant(aWeight,'class 1')
```

```
complianceStatus = logical  
    0
```

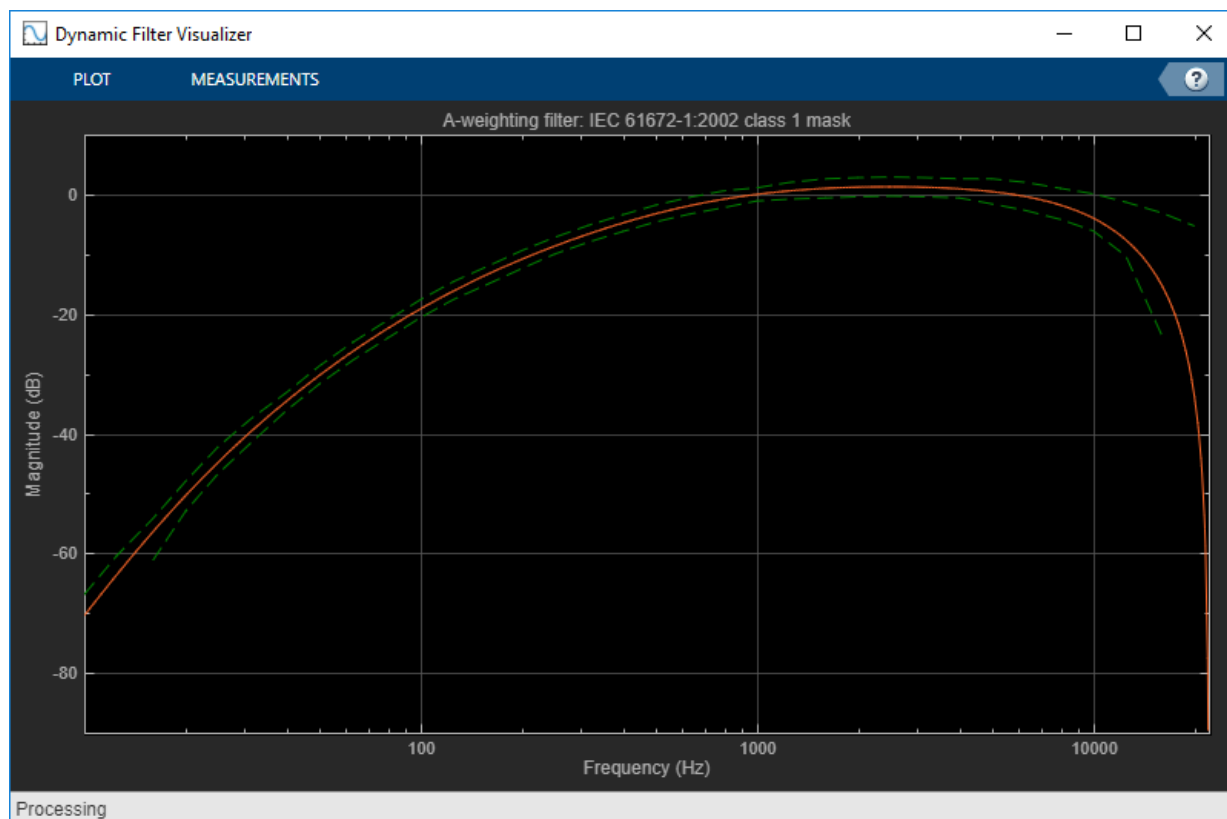
```
visualize(aWeight,'class 1')
```



Change your A-weighting filter sample rate to 44.1 kHz. Verify that the filter is standard compliant and visualize the filter design.

```
aWeight.SampleRate = 44100;  
complianceStatus = isStandardCompliant(aWeight, 'class 1')  
  
complianceStatus = logical  
1
```





## Perform A-Weighted Filtering

Use the `weightingFilter` System object™ to design an A-weighted filter, and then process an audio signal using your frequency-weighted filter design.

Create a `dsp.AudioFileReader` System object.

```
samplesPerFrame = 1024;
reader = dsp.AudioFileReader('Filename', ...
    'RockGuitar-16-44p1-stereo-72secs.wav', ...
    'SamplesPerFrame', samplesPerFrame, ...
    'PlayCount', Inf);
```

Create a `weightingFilter` System object. Use the sample rate of the reader as the sample rate of the weighting filter.

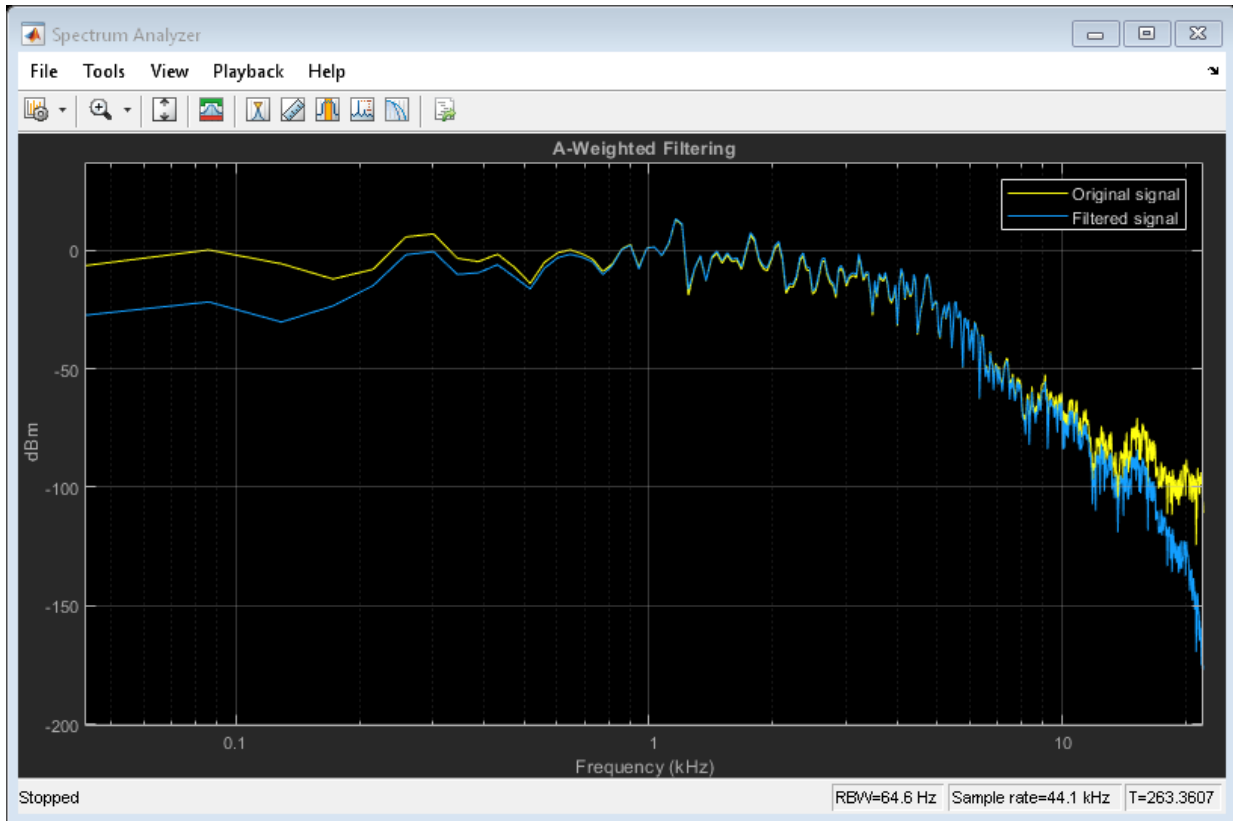
```
Fs = reader.SampleRate;  
weightFilt = weightingFilter('A-weighting',Fs);
```

Create a spectrum analyzer to visualize the original audio signal and the audio signal after frequency-weighted filtering.

```
scope = dsp.SpectrumAnalyzer( ...  
    'SampleRate',Fs, ...  
    'PlotAsTwoSidedSpectrum',false, ...  
    'FrequencyScale','Log', ...  
    'FrequencyResolutionMethod','WindowLength', ...  
    'WindowLength',samplesPerFrame, ...  
    'Title','A-Weighted Filtering', ...  
    'ShowLegend',true, ...  
    'ChannelNames',{'Original signal','Filtered signal'});
```

Process the audio signal in an audio stream loop. Visualize the filtered audio and the original audio. As a best practice, release the System objects when complete.

```
tic  
while toc < 20  
    x = reader();  
    y = weightFilt(x);  
    scope([x(:,1),y(:,1)])  
end  
  
release(weightFilt)  
release(reader)  
release(scope)
```



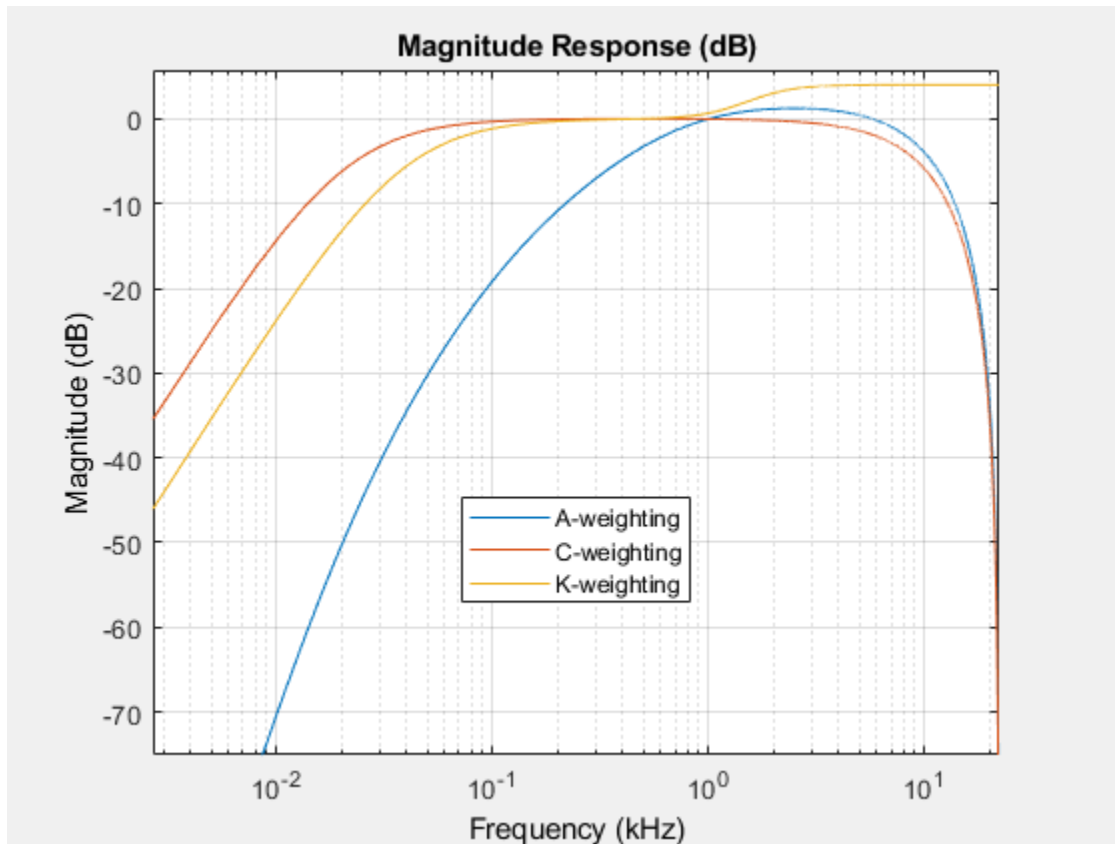
### Compare and Analyze Weighting Types

Compare the A-weighted, C-weighted, and K-weighted filtering of an engine sound.

Create an A-weighting filter, a C-weighting filter, and a K-weighting filter. Compare and analyze the filters using FVTool.

```
wF{1} = weightingFilter;
wF{2} = weightingFilter('C-weighting');
wF{3} = weightingFilter('K-weighting');
```

```
fvtool(wF{1},wF{2},wF{3}, 'FrequencyScale', 'Log', 'Fs', wF{1}.SampleRate)  
legend('A-weighting', 'C-weighting', 'K-weighting', 'location', 'best')
```



The `weightingFilter` object supports several filter analysis methods. For more information, use `help` at the command line:

```
help weightingFilter.helpFilterAnalysis
```

The following analysis methods are available for discrete-time filter System objects:

<code>fvtool</code>	- Filter visualization tool
<code>info</code>	- Filter information
<code>freqz</code>	- Frequency response
<code>phasez</code>	- Phase response

---

zerophase	- Zero-phase response
grpdelay	- Group delay response
phasedelay	- Phase delay response
impz	- Impulse response
impzlength	- Length of impulse response
stepz	- Step response
zplane	- Pole/zero plot
cost	- Cost estimate for implementation of the filter System object
measure	- Measure characteristics of the frequency response
order	- Filter order
coeffs	- Filter coefficients in a structure
firtype	- Determine the type (1-4) of a linear phase FIR filter System object
tf	- Convert to transfer function
zpk	- Convert to zero-pole-gain
ss	- Convert to state space representation
isallpass	- Verify if filter System object is allpass
isfir	- Verify if filter System object is FIR
islinphase	- Verify if filter System object is linear phase
ismaxphase	- Verify if filter System object is maximum phase
isminphase	- Verify if filter System object is minimum phase
isreal	- Verify if filter System object is minimum real
issos	- Verify if filter System object is in second-order sections form
isstable	- Verify if filter System object is stable
realizemdl	- Filter realization (Simulink diagram)
specifyall	- Fully specify fixed-point filter System object settings
cascade	- Create a FilterCascade System object
Second-order sections:	
scale	- Scale second-order sections of BiquadFilter System object
scalecheck	- Check scaling of BiquadFilter System object
reorder	- Reorder second-order sections of BiquadFilter System object
cumsec	- Cumulative second-order section of BiquadFilter System object
scaleopts	- Create an options object for second-order section scaling
sos	- Convert to second-order-sections (for IIRFilter System objects only)
Fixed-Point (Fixed-Point Designer Required):	
freqrespest	- Frequency response estimate via filtering

freqrespopts - Create an options object for frequency response estimate  
noisepsd - Power spectral density of filter output due to roundoff noise  
noisepsdopts - Create an options object for output noise PSD computation

Multirate Analysis:

polyphase - Polyphase decomposition of multirate filter System object  
gain (CIC decimator) - Gain of CIC decimator filter System object  
gain (CIC interpolator) - Gain of CIC interpolator filter System object

For decimator, interpolator, or rate change filter System objects the analysis tools perform computations relative to the rate at which the filter is running. If a sampling frequency is specified, it is assumed that the filter is running at that rate.

Help for `weightingFilter.helpFilterAnalysis` is inherited from superclass `DSP.PRIVATE.F`

Create a `dsp.AudioFileReader` and specify a sound file. Create an `audioDeviceWriter` with default properties. In an audio stream loop, play the white noise, and then listen to it filtered through the A-weighted, C-weighted, and K-weighted filters, successively.

```
fileReader = dsp.AudioFileReader('Engine-16-44p1-stereo-20sec.wav');  
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

```
fprintf('No filtering...')
```

```
No filtering...
```

```
for i = 1:400  
    x = fileReader();  
    if i==100  
        index = 1;  
        fprintf('A-weighted filtering...')  
    elseif i==200  
        index = 2;  
        fprintf('C-weighted filtering...')  
    elseif i==300  
        index = 3;  
        fprintf('K-weighted filtering...\n')  
    end  
    if i>99  
        y = wF{index}(x);  
    else  
        y = x;
```

```

    end
    deviceWriter(y);
end

```

A-weighted filtering...C-weighted filtering...K-weighted filtering...

As a best practice, release your objects once done.

```

release(deviceWriter)
release(fileReader)

```

### Use Weighting Filter Design with Biquad Filter

The `weightingFilter` object uses second-order sections (SOS) for filtering. To extract the weighting filter design, use `getFilter` to return a `dsp.BiquadFilter` object with the `SOSMatrix` and `ScaleValues` properties set.

Use `weightingFilter` to create C-weighted and A-weighted filter objects. Use `getFilter` to return corresponding `dsp.BiquadFilter` objects.

```

cFilt = weightingFilter('C-weighting');
aFilt = weightingFilter('A-weighting');
cSOSFilter = getFilter(cFilt);
aSOSFilter = getFilter(aFilt);

```

Create an audio file reader and audio device writer for audio input/output. Use the sample rate of your reader as the sample rate of your writer.

```

fileReader = dsp.AudioFileReader('JetAirplane-16-11p025-mono-16secs.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

```

In an audio stream loop, play the unfiltered signal. Release your file reader so that the next time you call it, it reads from the beginning of the file.

```

tic
while toc<8
    x = fileReader();
    deviceWriter(x);
end
release(fileReader)

```

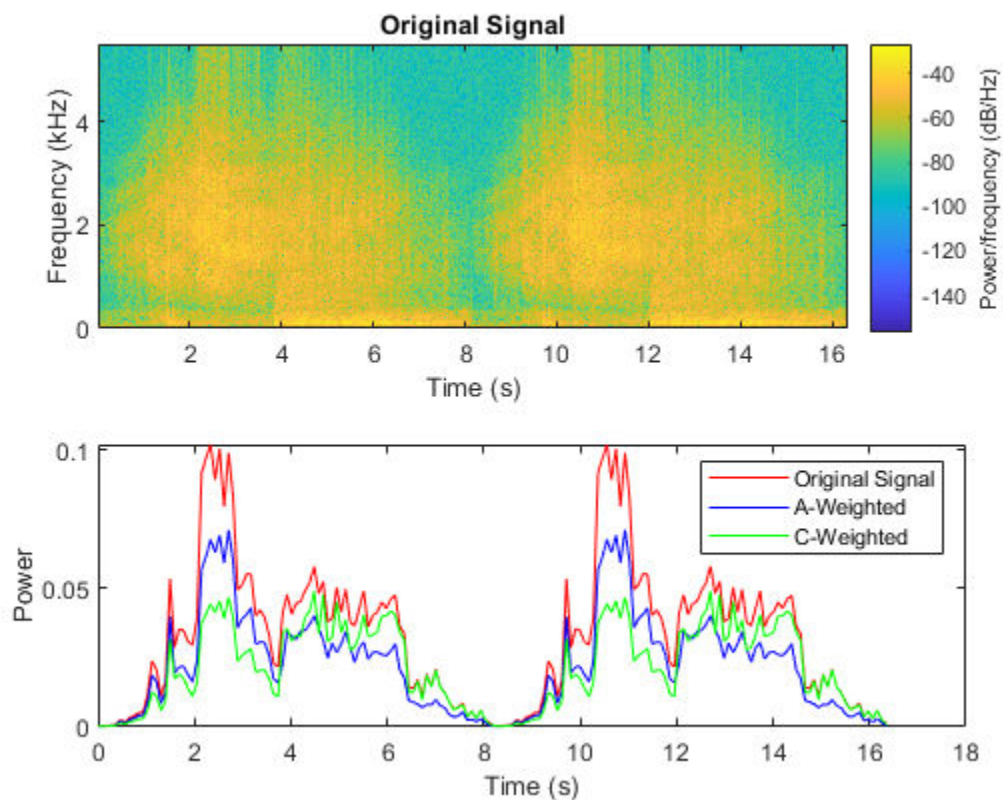
Play the signal processed by the A-weighted filter. Then play the signal processed by the C-weighted filter. Cache the power in each frame of the original and filtered signals for analysis. As a best practice, release your file reader and device writer once complete.

```
y = [];  
count = 1;  
tic  
while ~isDone(fileReader)  
    x = fileReader();  
    aFiltered = aSOSFilter(x);  
    cFiltered = cSOSFilter(x);  
    if toc>8  
        deviceWriter(cFiltered);  
    else  
        deviceWriter(aFiltered);  
    end  
    xPower(count) = var(x);  
    aPower(count) = var(aFiltered);  
    cPower(count) = var(cFiltered);  
    y = [y;x];  
    count = count+1;  
end  
  
release(fileReader)  
release(deviceWriter)
```

Plot the power of the original signal, the A-weighted signal, and the C-weighted signal over time.

```
subplot(2,1,1)  
spectrogram(y,512,256,4096,fileReader.SampleRate,'yaxis')  
title('Original Signal')  
  
subplot(2,1,2)  
t = linspace(0,16.3468,count-1);  
plot(t,xPower,'r',t,aPower,'b',t,cPower,'g')  
legend('Original Signal','A-Weighted','C-Weighted')  
xlabel('Time (s)')  
ylabel('Power')
```





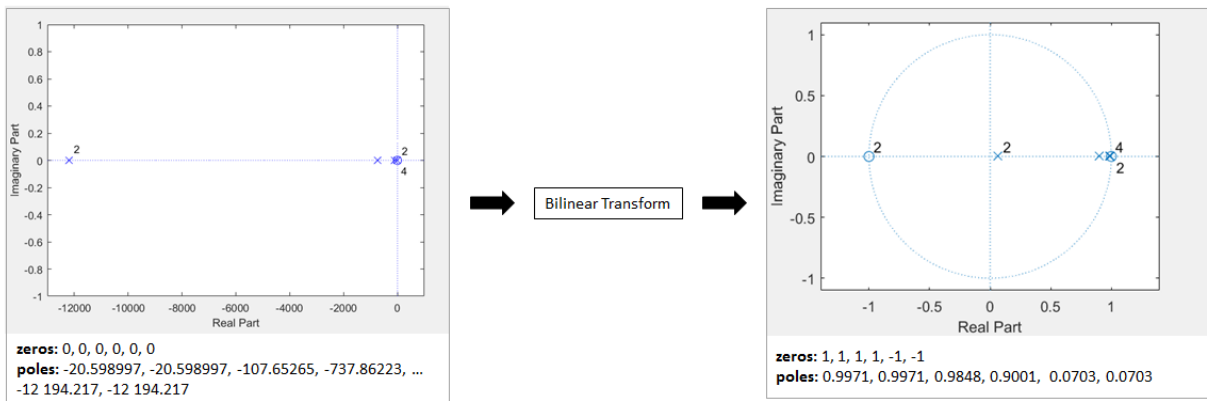
## Algorithms

### A-Weighting

The A-curve is a wide bandpass filter centered at 2.5 kHz, with approximately 20 dB attenuation at 100 Hz. A-weighted SPL measurements of noise level are increasingly found in sales literature for domestic appliances. In most countries, the use of A-weighting is mandated for the protection of workers against noise-induced deafness. The ISO and ICAO standards mandate A-weighting for all civil aircraft noise measurements.

The ANSI S1.42.2001 [1] defines this weighting curve. The IEC 61672-1:2002 [2] standard defines the minimum and maximum attenuation limits for an A-weighting filter.

ANSI S1.42.2001 defines the weighting curve by specifying analog poles and zeros. Audio Toolbox converts the specified poles and zeros to the digital domain using a bilinear transform:

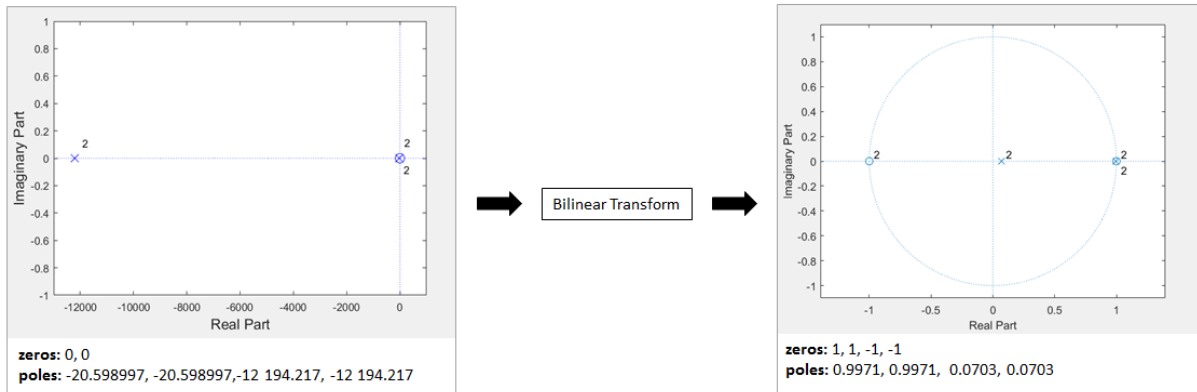


## C-Weighting

The C-curve is "flat," but with limited bandwidth: It has -3 dB corners at 31.5 Hz and 8 kHz. C-curves are used in sound level meters for sounds that are louder than those intended for A-weighting filters.

The ANSI S1.42-2001 [1] defines the C-weighting curve. The IEC 61672-1:2002 [2] standard defines the minimum and maximum attenuation limits for C-weighting filters.

ANSI S1.42.2001 defines the weighting curve by specifying analog poles and zeros. Audio Toolbox converts the specified poles and zeros to the digital domain using a bilinear transform:

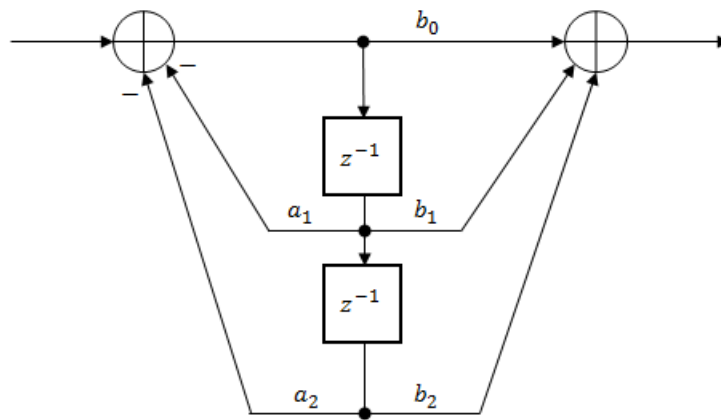


## K-Weighting

The K-weighting filter is used for loudness normalization in broadcast. It is composed of two stages of filtering: a first stage shelving filter and a second stage highpass filter.

The ITU-R BS.1770-4 [3] standard defines this curve.

Assume a second-order filter.



The table shows the coefficients for the filters.

<b>First Stage Shelving Coefficients</b>	<b>Second Stage Highpass Coefficients</b>
$a_1 = -1.69065929318241$	$a_1 = -1.99004745483398$
$a_2 = 0.73248077421585$	$a_2 = 0.99007225036621$
$b_0 = 1.53512485958697$	$b_0 = 1.0$
$b_1 = -2.6916918940638$	$b_1 = -2.0$
$b_2 = 1.19839281085285$	$b_2 = 1.0$

The coefficients presented by ITU-R BS.1770-4 are defined for 48 kHz. These coefficients are recomputed for nonstandard sample rates using the algorithm described in [4].

## References

- [1] Acoustical Society of America. *Design Response of Weighting Networks for Acoustical Measurements*. ANSI S1.42-2001. New York, NY: American National Standards Institute, 2001.
- [2] International Electrotechnical Commission. *Electroacoustics Sound Level Meters Part 1: Specifications*. First Edition. IEC 61672-1. 2002-2005.
- [3] International Telecommunication Union. *Algorithms to measure audio programme loudness and true-peak audio level*. ITU-R BS.1770-4. 2015.
- [4] Mansbridge, Stuart, Saoirse Finn, and Joshua D. Reiss. "Implementation and Evaluation of Autonomous Multi-track Fader Control." Paper presented at the 132nd Audio Engineering Society Convention, Budapest, Hungary, 2012.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

## See Also

Weighting Filter | `dsp.BiquadFilter` | `multibandParametricEQ` | `octaveFilter`

## Topics

“Audio Weighting Filters”

“Sound Pressure Measurement of Octave Frequency Bands”

**Introduced in R2016b**

## isStandardCompliant

Verify filter design is IEC 61672-1:2002 compliant

### Syntax

```
complianceStatus = isStandardCompliant(weightFilt,classType)
complianceStatus = isStandardCompliant( ____,freqRange)
```

### Description

`complianceStatus = isStandardCompliant(weightFilt,classType)` returns a logical scalar, `complianceStatus`, indicating whether the `weightFilt` filter design is compliant with the minimum and maximum attenuation specifications for the `classType` design specified in IEC 61672-1:2002. You can check compliance for A-weighting and C-weighting filters only.

`complianceStatus = isStandardCompliant( ____,freqRange)` specifies the range of frequencies checked for compliance.

### Examples

#### Verify Class 1 Standard Compliance

Create an object of the `weightingFilter` System object™. Call `isStandardCompliant`, specifying the compliance class type to check as the second argument.

```
weightFilt = weightingFilter;
complianceStatus = isStandardCompliant(weightFilt,'class 1')

complianceStatus = logical
    1
```

## Specify Frequency Range Checked for Compliance

Create an object of the `weightingFilter` System object™. Check the 'class 2' compliance status of the filter design over a specified frequency range.

```
weightFilt = weightingFilter;  
isStandardCompliant(weightFilt, 'class 2', [120,2000])  
  
ans = logical  
     1
```

## Input Arguments

### **weightFilt** — Object of `weightingFilter`

object

Object of the `weightingFilter` System object.

### **classType** — Compliance class type

'class 1' | 'class 2'

Compliance class type to verify, specified as 'class 1' or 'class 2'.

Data Types: char

### **freqRange** — Frequency range checked for compliance (Hz)

[minFreq,maxFreq] | two-element vector of increasing values

Specify the frequency range, in Hz, checked for compliance as a two-element vector of increasing values: [minFreq,maxFreq].

Data Types: single | double

## Output Arguments

### **complianceStatus** — Compliance status of filter design

scalar

Compliance status of filter design, returned as a logical scalar. The compliance status indicates whether the `weightFilt` filter design is compliant with the minimum and maximum attenuation specifications for the class type design specified by IEC 61672-1:2002 standard. Compliance can only be checked for A-weighting and C-weighting filters.

Data Types: `logical`

---

**Note** The pole-zero values defined in the ANSI S1.42-2001 standard are used for designing the A-weighted and C-weighted filters. The pole-zero values are based on analog filters, so the design can break compliance for lower sample rates.

---

## See Also

### Topics

“Audio Weighting Filters”

“Sound Pressure Measurement of Octave Frequency Bands”

**Introduced in R2016b**



# visualize

Visualize and validate filter response

## Syntax

```
visualize(weightFilt)
visualize(weightFilt,N)
visualize( ____,mType)
```

## Description

`visualize(weightFilt)` plots the magnitude response of the frequency-weighted filter, `weightFilt`. The plot is updated automatically when properties of the object change.

`visualize(weightFilt,N)` uses an N-point FFT to calculate the magnitude response.

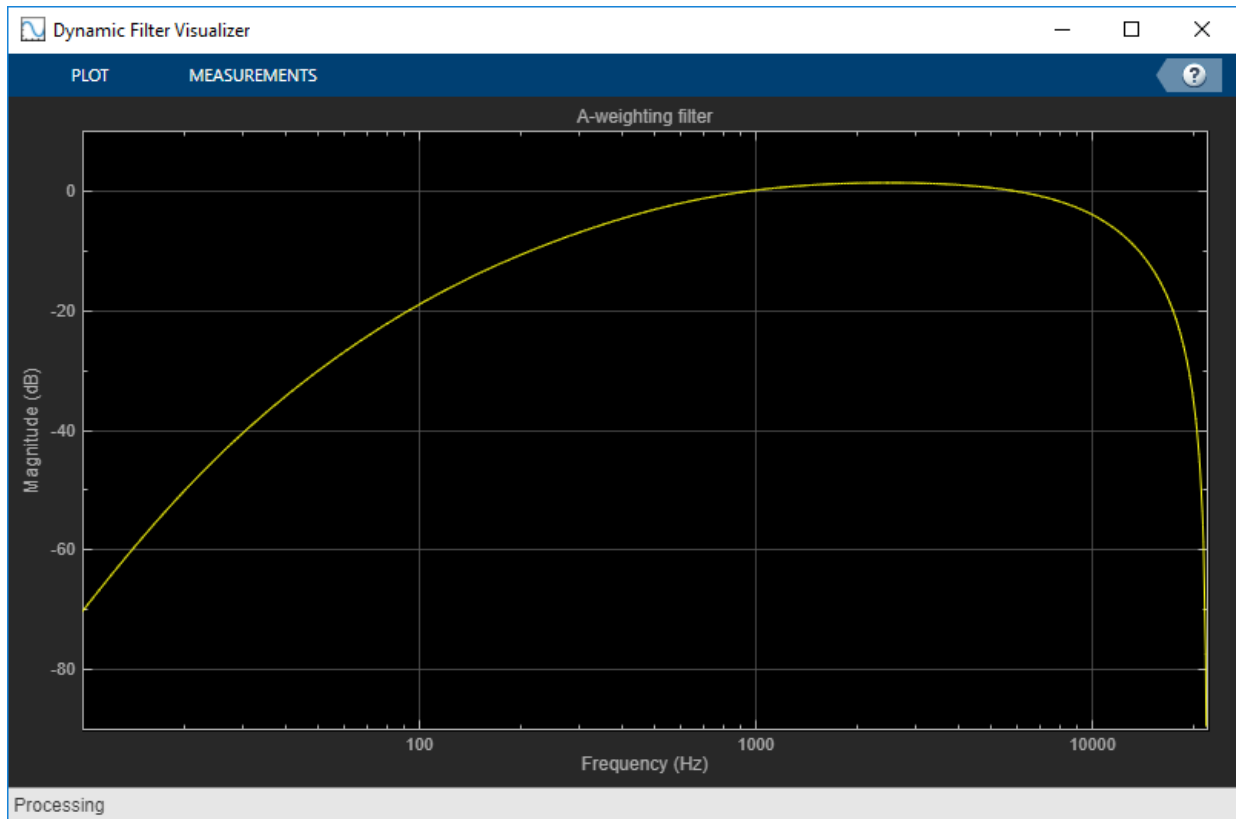
`visualize( ____,mType)` creates a mask based on the class of filter specified by `mType`, using either of the previous syntaxes.

## Examples

### Plot Weighting Filter Magnitude Response

Create an object of the `weightingFilter` System object™ and then plot the magnitude response of the filter.

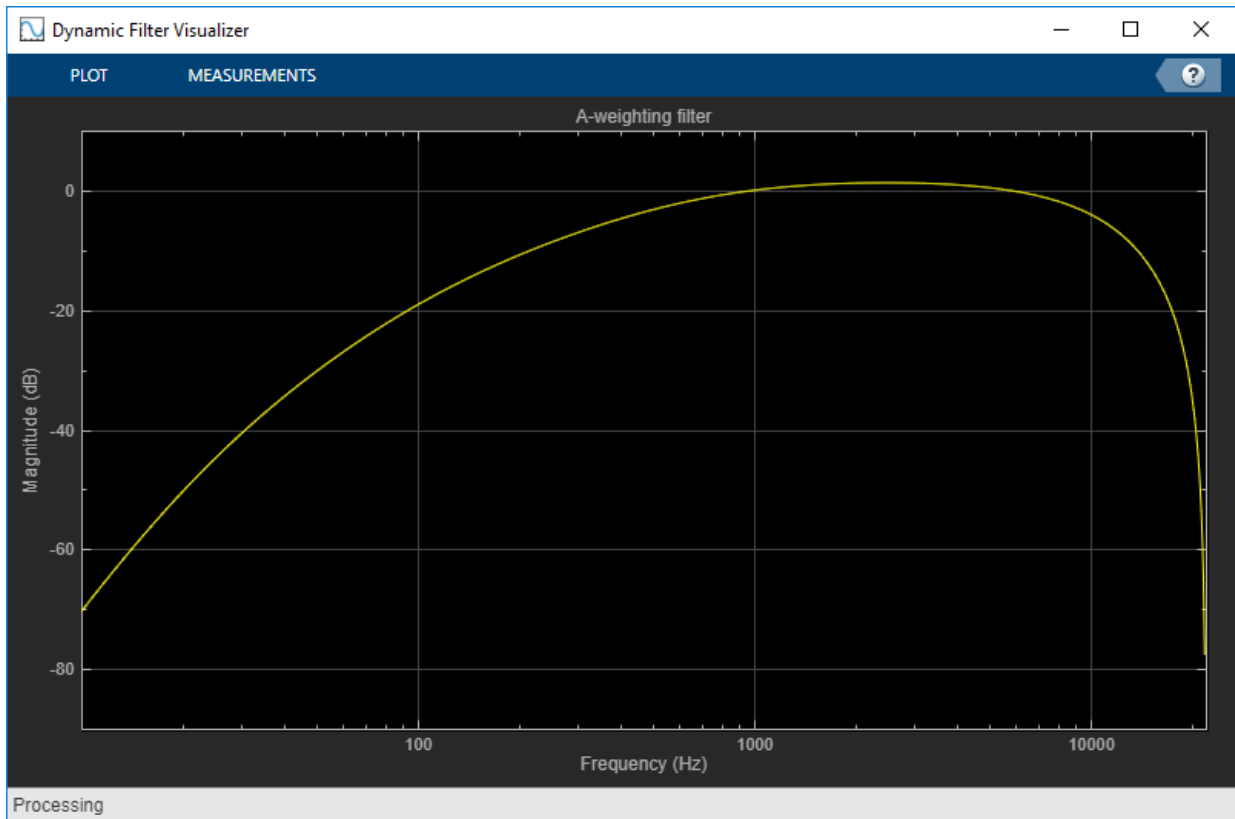
```
weightFilt = weightingFilter;
visualize(weightFilt)
```



#### Specify Number of Frequency Bins in FFT Calculation

Create an object of the `weightingFilter` System object™. Plot a 1024-point frequency representation.

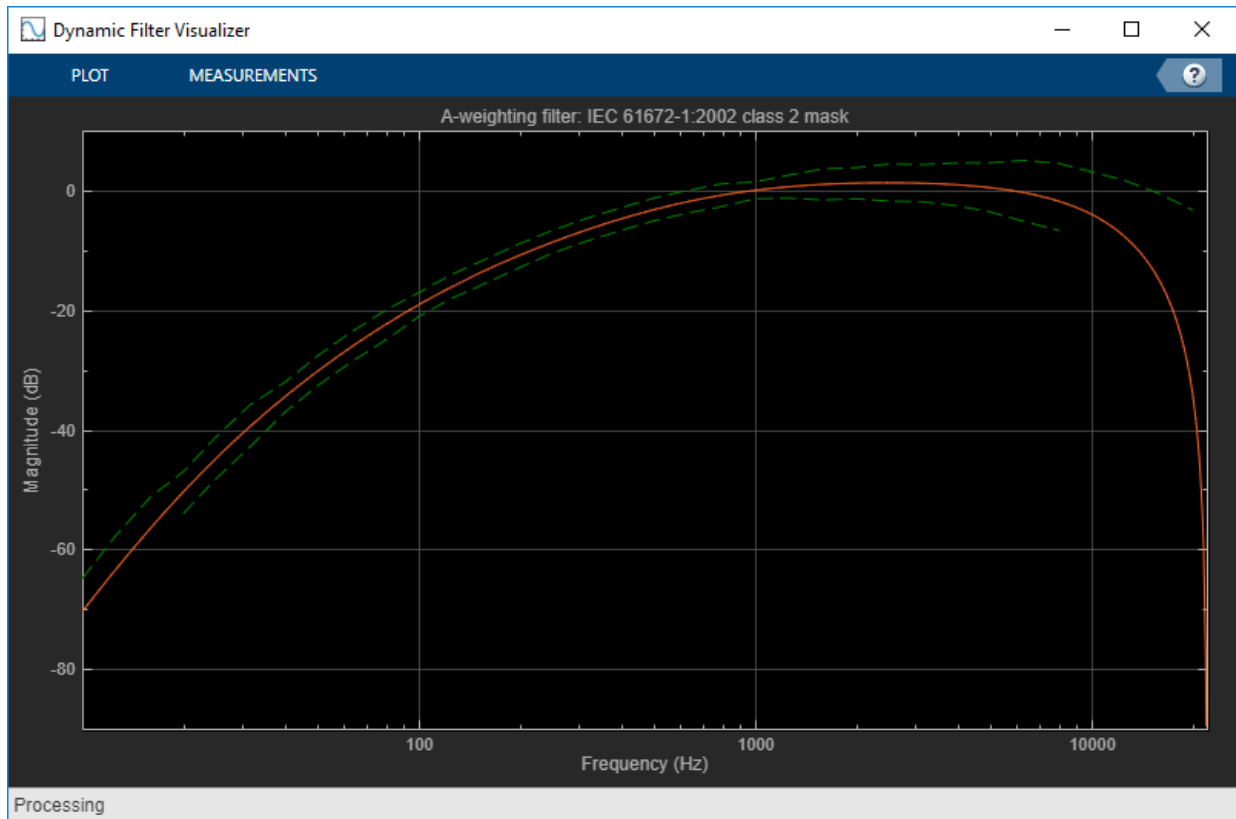
```
weightFilt = weightingFilter;  
visualize(weightFilt,1024)
```



### Visualize Class 2 Standard-Compliance Mask

Create an object of the `weightingFilter` System object™. Visualize the class 2 compliance of the filter design.

```
weightFilt = weightingFilter;  
visualize(weightFilt, 'class 2')
```



## Input Arguments

**weightFilt** — Object of `weightingFilter` object

Object of the `weightingFilter` System object.

**N** — Number of DFT bins

2048 | positive scalar

Number of DFT bins in frequency-domain representation, specified as a positive scalar. The default is 2048.

Data Types: single | double

### **mType — Type of mask**

'class 1' (default) | 'class 2'

Type of mask, specified as 'class 1' or 'class 2'.

The mask attenuation limits are defined in the IEC 61672-1:2002 standard. The mask is defined for A-weighting and C-weighting filters only.

- If the mask is green, the design is compliant with the IEC 61672-1:2002 standard.
- If the mask is red, the design breaks compliance.

---

**Note** The pole-zero values defined in the ANSI S1.42-2001 standard are used for designing the A-weighted and C-weighted filters. The pole-zero values are based on analog filters, so the design can break compliance for lower sample rates.

---

Data Types: char

## **See Also**

### **Topics**

“Audio Weighting Filters”

“Sound Pressure Measurement of Octave Frequency Bands”

**Introduced in R2016b**



# Classes in Audio Toolbox

---

## setExtractorParams

Set nondefault parameter values for individual feature extractors

### Syntax

```
setExtractorParams(aFE, featureName, params)  
setExtractorParams(aFE, featureName)
```

### Description

`setExtractorParams(aFE, featureName, params)` specifies parameters used to extract `featureName`.

`setExtractorParams(aFE, featureName)` returns the parameters used to extract `featureName` to default values.

### Examples

#### Extract Pitch Using the LHS Method

Read in an audio signal.

```
[audioIn, fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

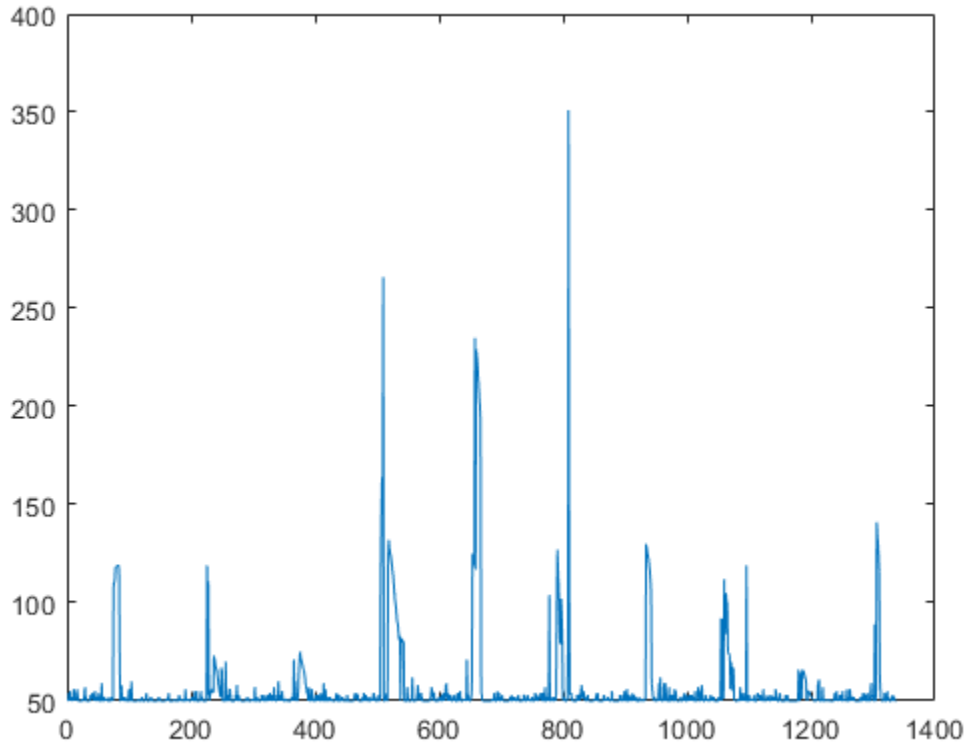
Create an `audioFeatureExtractor` object to extract pitch. Set the method of pitch extraction to "LHS".

```
aFE = audioFeatureExtractor("SampleRate", fs, "pitch", true);  
setExtractorParams(aFE, "pitch", "Method", "LHS")
```

Call `extract` and plot the results.

```
f0 = extract(aFE, audioIn);  
plot(f0)
```





### Modify Spectral Rolloff Threshold and Mel Spectrum Parameters

Read in an audio signal.

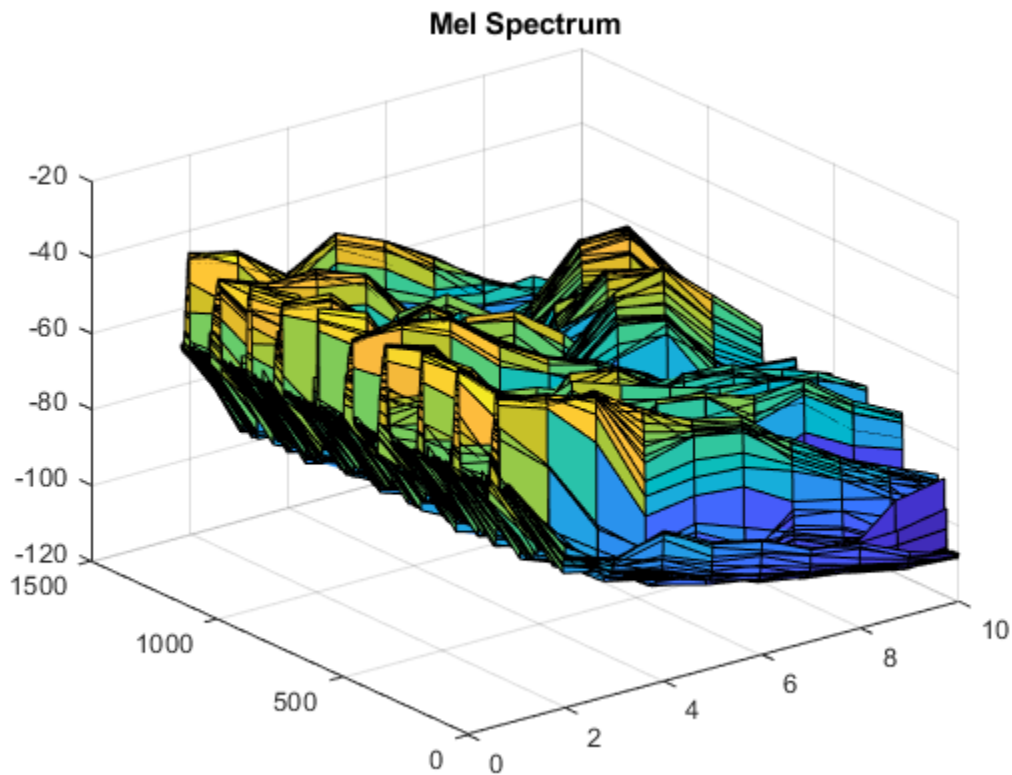
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioFeatureExtractor` object to extract the `melSpectrum` and `spectralRolloffPoint`. Specify ten bands for the mel spectrum and set the threshold for the rolloff point to 50% of the total energy.

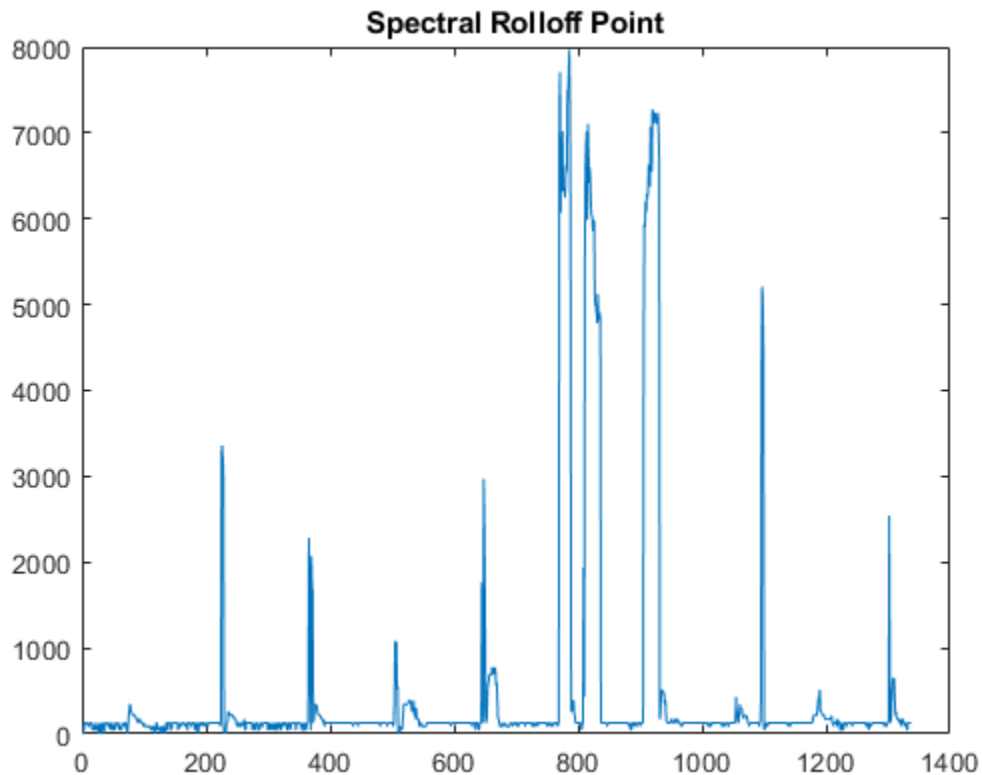
```
aFE = audioFeatureExtractor("SampleRate", fs, "melSpectrum", true, "spectralRolloffPoint", 1);  
setExtractorParams(aFE, "melSpectrum", "NumBands", 10)  
setExtractorParams(aFE, "spectralRolloffPoint", "Threshold", 0.5)
```

Call `extract` and plot the results.

```
features = extract(aFE, audioIn);  
idx = info(aFE);  
  
surf(10*log10(features(:, idx.melSpectrum)))  
title('Mel Spectrum')
```



```
plot(features(:, idx.spectralRolloffPoint))  
title('Spectral Rolloff Point')
```



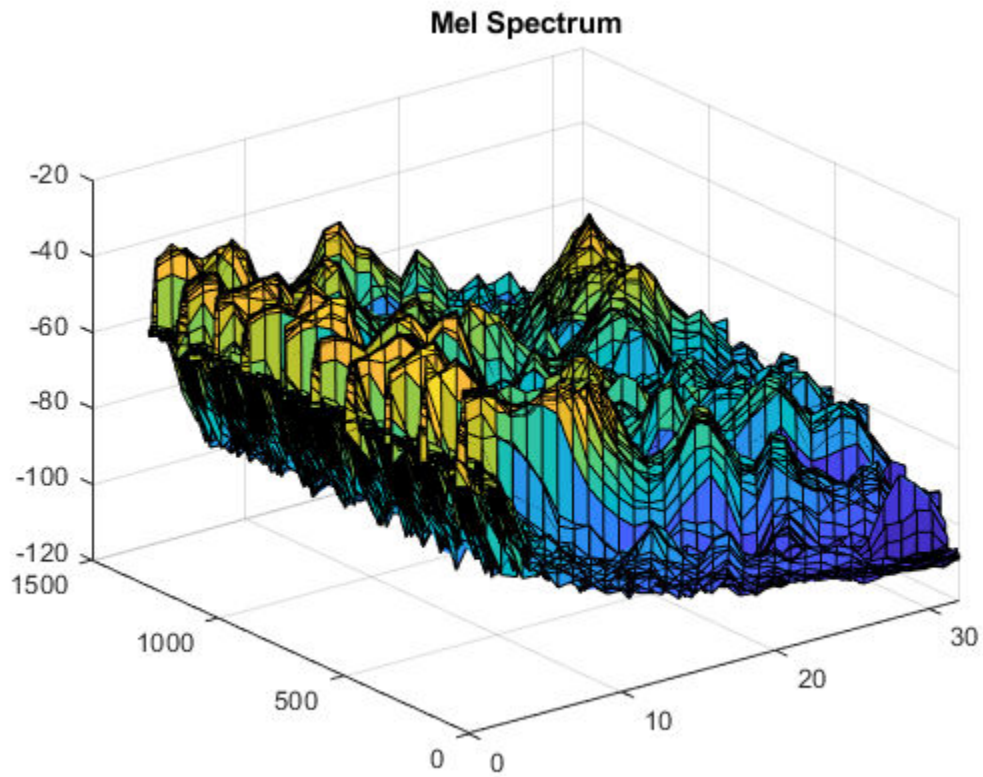
To return individual audio feature extractors to their default values, call `setExtractorParams` without specifying any parameters to set.

```
setExtractorParams(aFE, "melSpectrum")
setExtractorParams(aFE, "spectralRolloffPoint")
```

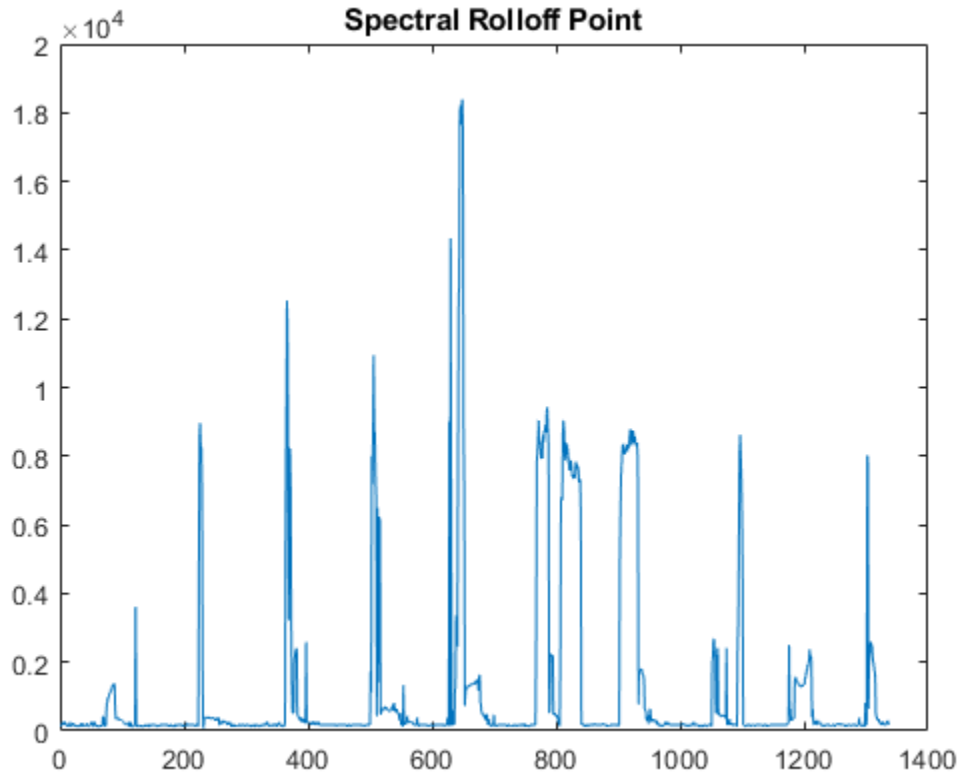
Call `extract` and plot the results.

```
features = extract(aFE, audioIn);
idx = info(aFE);

surf(10*log10(features(:, idx.melSpectrum)))
title('Mel Spectrum')
```



```
plot(features(:,idx.spectralRolloffPoint))  
title('Spectral Rolloff Point')
```



## Input Arguments

### **aFE** — Input object

audioFeatureExtractor object

audioFeatureExtractor object.

### **featureName** — Name of feature extractor

character array | string

Name of feature extractor, specified as a character array or string.

Data Types: `char` | `string`

**params — Parameters to set**

comma-separated name-value pairs | struct

Parameters to set, specified as comma-separated name-value pairs or as a struct.

### See Also

`audioFeatureExtractor`

**Introduced in R2019b**

# info

Output mapping and individual feature extractor parameters

## Syntax

```
idx = info(aFE)
idx = info(aFE,"all")
[idx,params] = info( ___ )
```

## Description

`idx = info(aFE)` returns a struct with field names corresponding to enabled feature extractors. The field values correspond to the column indices that the extracted features occupy in the output from `extract`.

`idx = info(aFE,"all")` returns a struct with field names corresponding to all available feature extractors. If the feature extractor is disabled, the field value is empty.

`[idx,params] = info( ___ )` returns a second struct, `params`. The field names of `params` correspond to the feature extractors with settable parameters. If the "all" flag is specified, `params` contains all feature extractors with settable parameters. If the "all" flag is not specified, `params` contains only the enabled feature extractors with settable parameters. You can set parameters using `setExtractorParams`.

## Examples

### Interpret Output from `extract`

Extract the mel spectrum, mel spectral centroid, and mel spectral skewness from concatenated white and pink noise.

```
fs = 48e3;
aFE = audioFeatureExtractor("SampleRate",fs, ...
    "melSpectrum",true, ...
```

```
"SpectralDescriptorInput", "melSpectrum", ...  
"spectralCentroid", true, ...  
"spectralSkewness", true);
```

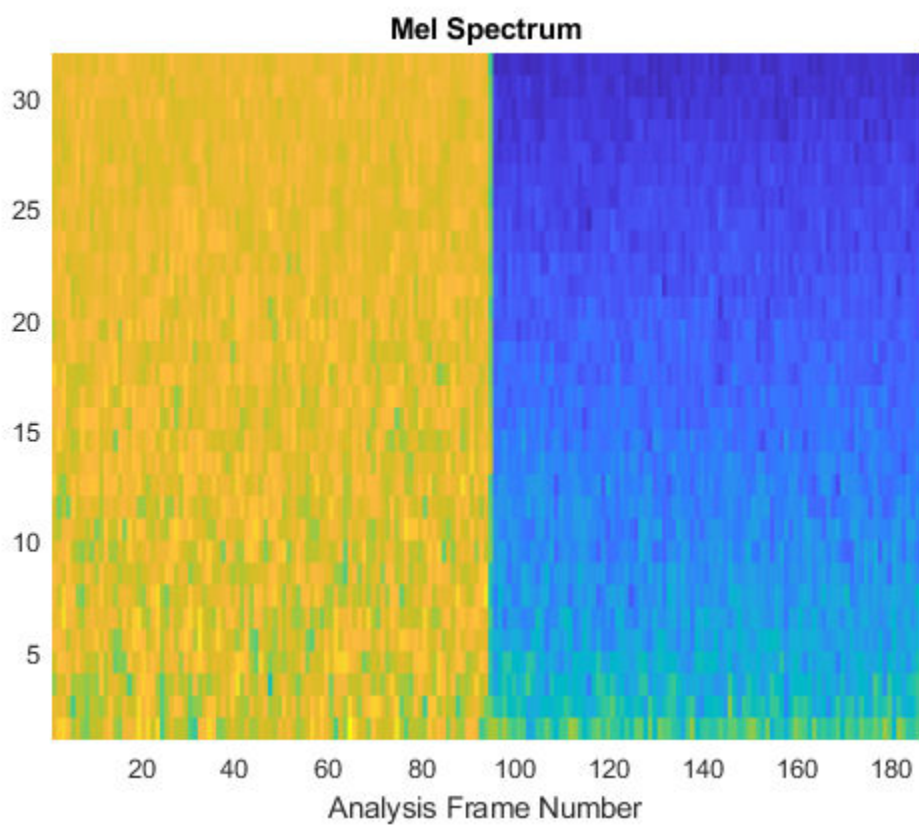
```
features = extract(aFE, [2*rand(fs,1)-1; pinknoise(fs,1)]);
```

Use `info` to determine which columns of the output correspond to which feature. Plot the features separately.

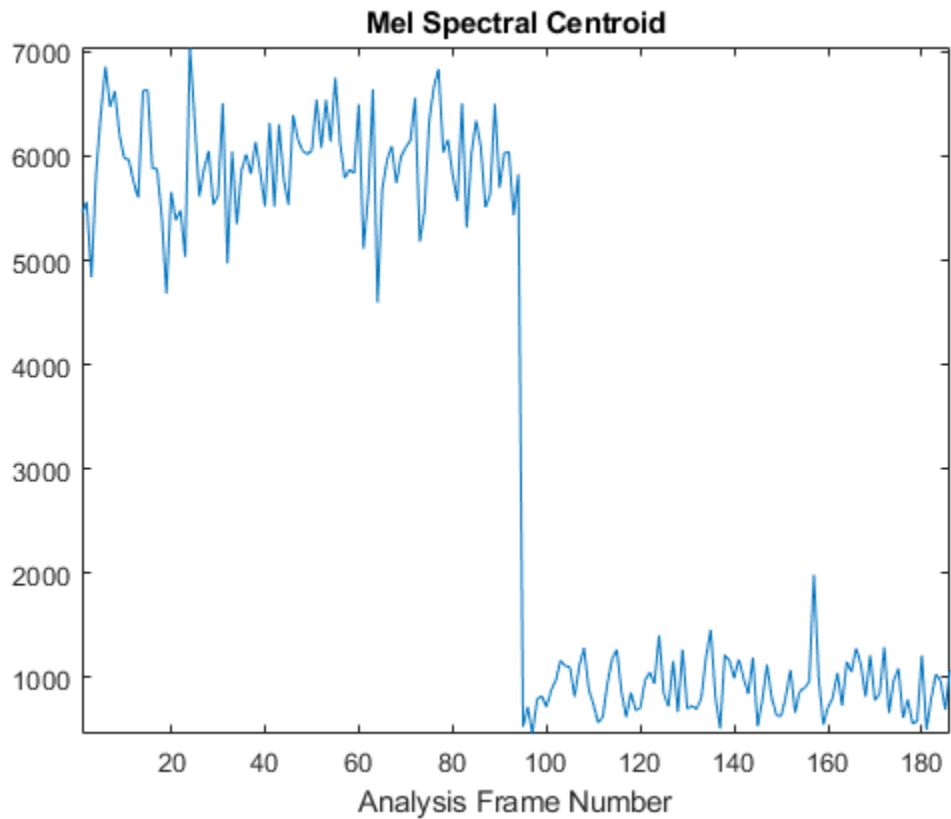
```
idx = info(aFE);
```

```
surf(log10(features(:, idx.melSpectrum)), "EdgeColor", "none");  
view([90, -90])  
axis tight  
title("Mel Spectrum")  
ylabel("Analysis Frame Number")
```

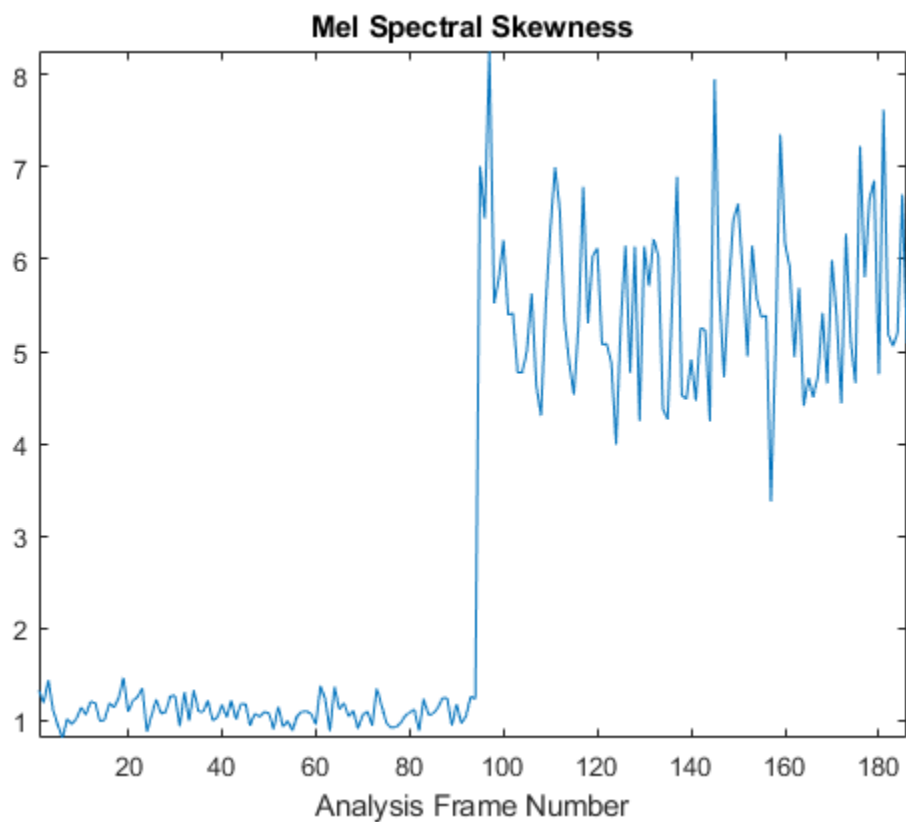




```
plot(features(:,idx.spectralCentroid))  
axis tight  
title("Mel Spectral Centroid")  
xlabel("Analysis Frame Number")
```



```
plot(features(:,idx.spectralSkewness))  
axis tight  
title("Mel Spectral Skewness")  
xlabel("Analysis Frame Number")
```



### Get List of All Features audioFeatureExtractor Provides

Create a default `audioFeatureExtractor` object. By default, all feature extractors are disabled.

```
aFE = audioFeatureExtractor
```

```
aFE =  
  audioFeatureExtractor with properties:
```

```
  Properties
```

```
        Window: [1024x1 double]
    OverlapLength: 512
        SampleRate: 44100
        FFTLength: []
SpectralDescriptorInput: 'linearSpectrum'
```

```
Enabled Features
    none
```

```
Disabled Features
```

```
linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
mfccDeltaDelta, gtcc, gtccDelta, gtccDeltaDelta, spectralCentroid, spectralCrest
spectralDecrease, spectralEntropy, spectralFlatness, spectralFlux, spectralKurtosis
spectralSkewness, spectralSlope, spectralSpread, pitch, harmonicRatio
```

To extract a feature, set the corresponding property to true.  
For example, `obj.mfcc = true`, adds mfcc to the list of enabled features.

The `info` function returns information about enabled feature extractors. To view information about all feature extractors, call `info` using the "all" flag.

```
[idx,params] = info(aFE,"all")
```

```
idx = struct with fields:
```

```
    linearSpectrum: [1x0 double]
    melSpectrum: [1x0 double]
    barkSpectrum: [1x0 double]
    erbSpectrum: [1x0 double]
    mfcc: [1x0 double]
    mfccDelta: [1x0 double]
    mfccDeltaDelta: [1x0 double]
    gtcc: [1x0 double]
    gtccDelta: [1x0 double]
    gtccDeltaDelta: [1x0 double]
    spectralCentroid: [1x0 double]
    spectralCrest: [1x0 double]
    spectralDecrease: [1x0 double]
    spectralEntropy: [1x0 double]
    spectralFlatness: [1x0 double]
    spectralFlux: [1x0 double]
    spectralKurtosis: [1x0 double]
    spectralRolloffPoint: [1x0 double]
    spectralSkewness: [1x0 double]
```

```

        spectralSlope: [1x0 double]
        spectralSpread: [1x0 double]
            pitch: [1x0 double]
        harmonicRatio: [1x0 double]

params = struct with fields:
    linearSpectrum: [1x1 struct]
    melSpectrum: [1x1 struct]
    barkSpectrum: [1x1 struct]
    erbSpectrum: [1x1 struct]
        mfcc: [1x1 struct]
        gtcc: [1x1 struct]
    spectralFlux: [1x1 struct]
    spectralRolloffPoint: [1x1 struct]
        pitch: [1x1 struct]

```

Use the `idx` struct to enable all feature extractors on the `audioFeatureExtractor` object.

```

features = fieldnames(idx);
for i = 1:numel(features)
    aFE.(features{i}) = true;
end
aFE

```

```

aFE =
    audioFeatureExtractor with properties:

    Properties
        Window: [1024x1 double]
        OverlapLength: 512
        SampleRate: 44100
        FFTLength: []
        SpectralDescriptorInput: 'linearSpectrum'

```

```

Enabled Features
    linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
    mfccDeltaDelta, gtcc, gtccDelta, gtccDeltaDelta, spectralCentroid, spectralCrest
    spectralDecrease, spectralEntropy, spectralFlatness, spectralFlux, spectralKurtosis
    spectralSkewness, spectralSlope, spectralSpread, pitch, harmonicRatio

```

```

Disabled Features
    none

```

To extract a feature, set the corresponding property to true.  
For example, `obj.mfcc = true`, adds `mfcc` to the list of enabled features.

### Determine Settable Parameters of Individual Feature Extractors

Create an `audioFeatureExtractor` to extract the ERB spectrum.

```
aFE = audioFeatureExtractor("erbSpectrum",true)
```

```
aFE =  
audioFeatureExtractor with properties:
```

```
Properties
```

```
Window: [1024x1 double]  
OverlapLength: 512  
SampleRate: 44100  
FFTLength: []
```

```
SpectralDescriptorInput: 'linearSpectrum'
```

```
Enabled Features
```

```
erbSpectrum
```

```
Disabled Features
```

```
linearSpectrum, melSpectrum, barkSpectrum, mfcc, mfccDelta, mfccDeltaDelta,  
gtcc, gtccDelta, gtccDeltaDelta, spectralCentroid, spectralCrest, spectralDecrease,  
spectralEntropy, spectralFlatness, spectralFlux, spectralKurtosis, spectralRolloff,  
spectralSlope, spectralSpread, pitch, harmonicRatio
```

To extract a feature, set the corresponding property to true.  
For example, `obj.mfcc = true`, adds `mfcc` to the list of enabled features.

The second output argument from `info` is a `struct` that contains the settable parameters of the individual feature extractors and their current value.

```
[~,params] = info(aFE)
```

```
params = struct with fields:
  erbSpectrum: [1x1 struct]
```

```
params.erbSpectrum
```

```
ans = struct with fields:
  NumBands: 43
  FrequencyRange: [0 22050]
  Normalization: "bandwidth"
  SpectrumType: "power"
```

If you are using the default parameter values, then the parameters are dynamic and updated when properties they depend on are updated. For example, the default frequency range of the ERB filter bank and the default number of bandpass filters in the ERB filter bank depends on the sample rate. Decrease the sample rate of the `audioFeatureExtractor` object and then call `info` again.

```
aFE.SampleRate = 16e3;
[~,params] = info(aFE);
params.erbSpectrum
```

```
ans = struct with fields:
  NumBands: 34
  FrequencyRange: [0 8000]
  Normalization: "bandwidth"
  SpectrumType: "power"
```

You can modify the individual feature extractor parameters using `setExtractorParams`. Set the number of bands to 40 and the spectrum type to "magnitude". Call `info` to confirm that the parameters are updated.

```
params.erbSpectrum.NumBands = 40;
params.erbSpectrum.SpectrumType = "magnitude";
setExtractorParams(aFE,"erbSpectrum",params.erbSpectrum)
[~,params] = info(aFE);
params.erbSpectrum
```

```
ans = struct with fields:
  NumBands: 40
  FrequencyRange: [0 8000]
  Normalization: "bandwidth"
```

```
SpectrumType: "magnitude"
```

When you set individual feature extractor parameters, they remain at the set value until you set them to another value or return them to defaults. Return the sample rate of the `audioFeatureExtractor` object to its initial value and then call `info`. The parameters remain at their set value.

```
aFE.SampleRate = 44.1e3;

[~,params] = info(aFE);
params.erbSpectrum

ans = struct with fields:
    NumBands: 40
    FrequencyRange: [0 8000]
    Normalization: "bandwidth"
    SpectrumType: "magnitude"
```

To return parameters to their default values, call `setExtractorParams` and specify no parameters.

```
setExtractorParams(aFE,"erbSpectrum")
[~,params] = info(aFE);
params.erbSpectrum

ans = struct with fields:
    NumBands: 43
    FrequencyRange: [0 22050]
    Normalization: "bandwidth"
    SpectrumType: "power"
```

## Input Arguments

### **aFE** — Input object

`audioFeatureExtractor` object

`audioFeatureExtractor` object.



---

## Output Arguments

### **idx — Mapping of requested features with output from extract**

struct

Mapping of requested features with output from `extract`, returned as a struct with field names corresponding to individual feature extractors and field values corresponding to column indices.

### **params — Settable parameters of individual feature extractors**

struct

Settable parameters of individual feature extractors, returned as a struct with field names corresponding to individual feature extractors and field values containing parameter specification structs. The parameter specification structs have field names corresponding to settable parameter names and field values corresponding to the current parameter setting.

## See Also

`audioFeatureExtractor`

**Introduced in R2019b**

## extract

Extract audio features

## Syntax

```
features = extract(aFE, audioIn)
```

## Description

`features = extract(aFE, audioIn)` returns an array containing features of the audio input.

## Examples

### Extract and Normalize Audio Features

Read in an audio signal.

```
[audioIn, fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioFeatureExtractor` to extract the centroid of the Bark spectrum, the kurtosis of the Bark spectrum, and the pitch of an audio signal.

```
aFE = audioFeatureExtractor("SampleRate", fs, ...  
    "SpectralDescriptorInput", "barkSpectrum", ...  
    "spectralCentroid", true, ...  
    "spectralKurtosis", true, ...  
    "pitch", true)
```

```
aFE =  
    audioFeatureExtractor with properties:
```

```
    Properties
```

```
        Window: [1024x1 double]  
    OverlapLength: 512
```

```

        SampleRate: 44100
        FFTLength: []
SpectralDescriptorInput: 'barkSpectrum'

```

```

Enabled Features
    spectralCentroid, spectralKurtosis, pitch

```

```

Disabled Features
    linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
    mfccDeltaDelta, gtcc, gtccDelta, gtccDeltaDelta, spectralCrest, spectralDecrease
    spectralEntropy, spectralFlatness, spectralFlux, spectralRolloffPoint, spectralSk
    spectralSpread, harmonicRatio

```

To extract a feature, set the corresponding property to true.  
For example, `obj.mfcc = true`, adds mfcc to the list of enabled features.

Call `extract` to extract the features from the audio signal. Normalize the features by their mean and standard deviation.

```

features = extract(aFE, audioIn);
features = (features - mean(features,1))./std(features,[],1);

```

Plot the normalized features over time.

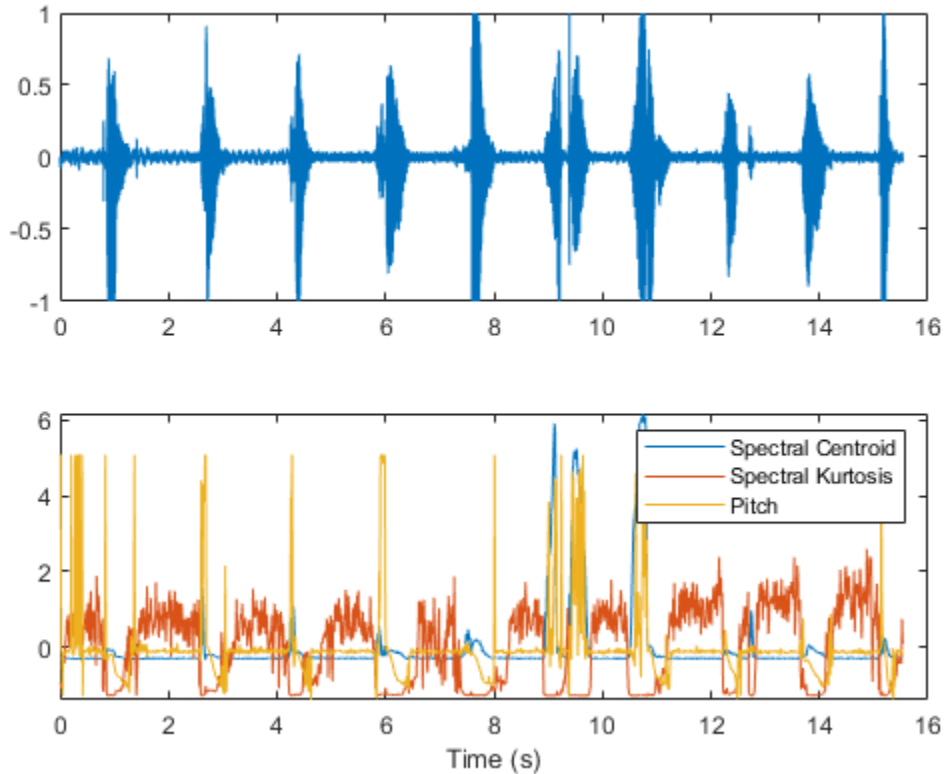
```

idx = info(aFE);
duration = size(audioIn,1)/fs;

subplot(2,1,1)
t = linspace(0,duration,size(audioIn,1));
plot(t,audioIn)

subplot(2,1,2)
t = linspace(0,duration,size(features,1));
plot(t,features(:,idx.spectralCentroid), ...
      t,features(:,idx.spectralKurtosis), ...
      t,features(:,idx.pitch));
legend("Spectral Centroid", "Spectral Kurtosis", "Pitch")
xlabel("Time (s)")

```



### Input Arguments

**aFE** — Input object

audioFeatureExtractor object

audioFeatureExtractor object.

**audioIn** — Input audio

column vector | matrix

Input audio, specified as a column vector or matrix of independent channels (columns).

Data Types: single | double

## Output Arguments

### **features** — Extracted audio features

vector | matrix | 3-D array

Extracted audio features, returned as an  $L$ -by- $M$ -by- $N$  array, where:

- $L$  -- Number of feature vectors (hops)
- $M$  -- Number of features extracted per analysis window
- $N$  -- Number of channels

Data Types: single | double

## See Also

audioFeatureExtractor

**Introduced in R2019b**

# audioFeatureExtractor

Streamline audio feature extraction

## Description

audioFeatureExtractor encapsulates multiple audio feature extractors into a streamlined and modular implementation.

## Creation

## Syntax

```
aFE = audioFeatureExtractor()  
aFE = audioFeatureExtractor(Name,Value)
```

## Description

aFE = audioFeatureExtractor() creates an audio feature extractor with default property values.

aFE = audioFeatureExtractor(Name,Value) specifies nondefault properties for aFE using one or more name-value pair arguments.

## Properties

### Main Properties

#### Window — Analysis window

hamming(1024,"periodic") (default) | real vector

Analysis window, specified as a real vector.

Data Types: `single` | `double`

**OverlapLength — Overlap length of adjacent analysis windows**

512 (default) | integer in the range [0, numel(Window))

Overlap length of adjacent analysis windows, specified as an integer in the range [0, numel(Window)).

Data Types: `single` | `double`

**FFTLength — FFT length**

[] (default) | positive integer

FFT length, specified as an integer. The default, [], means that the FFT length is equal to the window length, (numel(Window)).

Data Types: `single` | `double`

**SampleRate — Input sample rate (Hz)**

44100 (default) | nonnegative scalar

Input sample rate in Hz, specified as a nonnegative scalar.

Data Types: `single` | `double`

**SpectralDescriptorInput — Input to spectral descriptors**

"linearSpectrum" (default) | "melSpectrum" | "barkSpectrum" | "erbSpectrum"

Input to spectral descriptors, specified as "linearSpectrum", "melSpectrum", "barkSpectrum", or "erbSpectrum".

Spectral descriptors affected by this property are:

- spectralCentroid
- spectralCrest
- spectralDecrease
- spectralEntropy
- spectralFlatness
- spectralFlux
- spectralKurtosis
- spectralRolloffPoint

- spectralSkewness
- spectralSlope
- spectralSpread

The spectrum input to the spectral descriptors is the same as output from the corresponding feature:

- linearSpectrum
- melSpectrum
- barkSpectrum
- erbSpectrum

For example, if you set "SpectralDescriptorInput" to "barkSpectrum", and "spectralCentroid" to true, then aFE returns the centroid of the default Bark spectrum.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
aFE = audioFeatureExtractor("SampleRate",fs, ...
                           "SpectralDescriptorInput","barkSpectrum", ...
                           "spectralCentroid",true);
barkSpectralCentroid = extract(aFE,audioIn);
```

If you specify a nondefault barkSpectrum using setExtractorParams, then the nondefault Bark spectrum is the input to the spectral descriptors. For example, if you call setExtractorParams(aFE,"barkSpectrum","NumBands",40), then aFE returns the centroid of an 40-band Bark spectrum.

```
setExtractorParams(aFE,"barkSpectrum","NumBands",40)
bark40SpectralCentroid = extract(aFE,audioIn);
```

Data Types: char | string

## Features to Extract

### linearSpectrum — Extract linear spectrum

false (default) | true

Extract the one-sided linear spectrum, specified as true or false.

To set parameters of the linear spectrum extraction, use setExtractorParams:



```
setExtractorParams(aFE, "linearSpectrum", "Name", Value)
```

Settable parameters for the linear spectrum extraction are:

- "FrequencyRange" -- Frequency range of the extracted spectrum in Hz, specified as the comma-separated pair consisting of "FrequencyRange" and a two-element vector of increasing numbers in the range [0, SampleRate/2]. If unspecified, FrequencyRange defaults to [0, SampleRate/2].
- "SpectrumType" -- Spectrum type, specified as the comma-separated pair consisting of "SpectrumType" and "power" or "magnitude". If unspecified, SpectrumType defaults to "power".

Data Types: logical

### **melSpectrum — Extract mel spectrum**

false (default) | true

Extract the one-sided mel spectrum, specified as true or false.

To set parameters of the mel spectrum extraction, use `setExtractorParams`:

```
setExtractorParams(aFE, "melSpectrum", "Name", Value)
```

Settable parameters for the Bark spectrum extraction are:

- "FrequencyRange" -- Frequency range of the extracted spectrum in Hz, specified as the comma-separated pair consisting of "FrequencyRange" and a two-element vector of increasing numbers in the range [0, SampleRate/2]. If unspecified, FrequencyRange defaults to [0, SampleRate/2].
- "SpectrumType" -- Spectrum type, specified as the comma-separated pair consisting of "SpectrumType" and "power" or "magnitude". If unspecified, SpectrumType defaults to "power".
- "NumBands" -- Number of mel bands, specified as the comma-separated pair consisting of "NumBands" and an integer. If unspecified, NumBands defaults to 32.
- "Normalization" -- Normalization applied to bandpass filters, specified as the comma-separated pair consisting of "Normalization" and "bandwidth" or "area". If unspecified, Normalization defaults to "bandwidth".

Data Types: logical

### **barkSpectrum — Extract Bark spectrum**

false (default) | true

Extract the one-sided Bark spectrum, specified as `true` or `false`.

To set parameters of the Bark spectrum extraction, use `setExtractorParams`:

```
setExtractorParams(aFE, "barkSpectrum", "Name", Value)
```

Settable parameters for the Bark spectrum extraction are:

- "FrequencyRange" -- Frequency range of the extracted spectrum in Hz, specified as the comma-separated pair consisting of "FrequencyRange" and a two-element vector of increasing numbers in the range  $[0, \text{SampleRate}/2]$ . If unspecified, FrequencyRange defaults to  $[0, \text{SampleRate}/2]$ .
- "SpectrumType" -- Spectrum type, specified as the comma-separated pair consisting of "SpectrumType" and "power" or "magnitude". If unspecified, SpectrumType defaults to "power".
- "NumBands" -- Number of Bark bands, specified as the comma-separated pair consisting of "NumBands" and an integer. If unspecified, NumBands defaults to 32.
- "Normalization" -- Normalization applied to bandpass filters, specified as the comma-separated pair consisting of "Normalization" and "bandwidth" or "area". If unspecified, Normalization defaults to "bandwidth".

Data Types: `logical`

### **erbSpectrum** — Extract ERB spectrum

`false` (default) | `true`

Extract the one-sided ERB spectrum, specified as `true` or `false`.

To set parameters of the ERB spectrum extraction, use `setExtractorParams`:

```
setExtractorParams(aFE, "erbSpectrum", "Name", Value)
```

Settable parameters for the ERB spectrum extraction are:

- "FrequencyRange" -- Frequency range of the extracted spectrum in Hz, specified as the comma-separated pair consisting of "FrequencyRange" and a two-element vector of increasing numbers in the range  $[0, \text{SampleRate}/2]$ . If unspecified, FrequencyRange defaults to  $[0, \text{SampleRate}/2]$ .
- "SpectrumType" -- Spectrum type, specified as the comma-separated pair consisting of "SpectrumType" and "power" or "magnitude". If unspecified, SpectrumType defaults to "power".

- "NumBands" -- Number of ERB bands, specified as the comma-separated pair consisting of "NumBands" and an integer. If unspecified, NumBands defaults to  $\text{ceil}(\text{hz2erb}(\text{FrequencyRange}(2)) - \text{hz2erb}(\text{FrequencyRange}(1)))$ .
- "Normalization" -- Normalization applied to bandpass filters, specified as the comma-separated pair consisting of "Normalization" and "bandwidth" or "area". If unspecified, Normalization defaults to "bandwidth".

Data Types: logical

### **mfcc** — Extract mel-frequency cepstral coefficients (MFCC)

false (default) | true

Extract mel-frequency cepstral coefficients (MFCC), specified as true or false.

To set parameters of the MFCC extraction, use `setExtractorParams`:

```
setExtractorParams(aFE, "mfcc", "Name", Value)
```

Settable parameters for the MFCC extraction are:

- "NumCoeffs" -- Number of coefficients returned for each window, specified as a the comma-separated pair consisting of "NumCoeffs" and a positive integer. If unspecified, NumCoeffs defaults to 13.
- "DeltaWindowLength" -- Delta window length, specified as the comma-separated pair consisting of "DeltaWindowLength" and 2 or an odd integer. If unspecified, DeltaWindowLength defaults to 2. This parameter affects the `mfccDelta` and `mfccDeltaDelta` features.

The mel-frequency cepstral coefficients are calculated using the `melSpectrum`.

Data Types: logical

### **mfccDelta** — Extract delta of MFCC

false (default) | true

Extract delta of MFCC, specified as true or false.

The delta MFCC is calculated based on the extracted MFCC. Parameters set on `mfcc` affect `mfccDelta`.

Data Types: logical

### **mfccDeltaDelta — Extract delta-delta of MFCC**

false (default) | true

Extract delta-delta of MFCC, specified as true or false.

The delta-delta MFCC is calculated based on the extracted MFCC. Parameters set on `mfcc` affect `mfccDeltaDelta`.

Data Types: logical

### **gtcc — Extract gammatone cepstral coefficients (GTCC)**

false (default) | true

Extract gammatone cepstral coefficients (GTCC), specified as true or false.

To set parameters of the GTCC extraction, use `setExtractorParams`:

```
setExtractorParams(aFE, "gtcc", "Name", Value)
```

Settable parameters for the GTCC extraction are:

- "NumCoeffs" -- Number of coefficients returned for each window, specified as a the comma-separated pair consisting of "NumCoeffs" and a positive integer. If unspecified, NumCoeffs defaults to 13.
- "DeltaWindowLength" -- Delta window length, specified as the comma-separated pair consisting of "DeltaWindowLength" and 2 or an odd integer. If unspecified, DeltaWindowLength defaults to 2. This parameter affects the `gtccDelta` and `gtccDeltaDelta` features.

The gammatone cepstral coefficients are calculated using the `erbSpectrum`.

Data Types: logical

### **gtccDelta — Extract delta of GTCC**

false (default) | true

Extract delta of GTCC, specified as true or false.

The delta GTCC is calculated based on the extracted GTCC. Parameters set on `gtcc` affect `gtccDelta`.

Data Types: logical

**gtccDeltaDelta — Extract delta-delta of GTCC**`false (default) | true`

Extract delta-delta of GTCC, specified as `true` or `false`.

The delta-delta GTCC is calculated based on the extracted GTCC. Parameters set on `gtcc` affect `gtccDeltaDelta`.

Data Types: `logical`

**spectralCentroid — Extract spectral centroid**`false (default) | true`

Extract spectral centroid, specified as `true` or `false`.

The spectral centroid is calculated on one of the following spectral representations, as specified by the `SpectralDescriptorInput` property:

- `linearSpectrum`
- `melSpectrum`
- `barkSpectrum`
- `erbSpectrum`

Data Types: `logical`

**spectralCrest — Extract spectral crest**`false (default) | true`

Extract spectral crest, specified as `true` or `false`.

The spectral crest is calculated on one of the following spectral representations, as specified by the `SpectralDescriptorInput` property:

- `linearSpectrum`
- `melSpectrum`
- `barkSpectrum`
- `erbSpectrum`

Data Types: `logical`

**spectralDecrease — Extract spectral decrease**`false (default) | true`

Extract spectral decrease, specified as `true` or `false`.

The spectral decrease is calculated on one of the following spectral representations, as specified by the `SpectralDescriptorInput` property:

- `linearSpectrum`
- `melSpectrum`
- `barkSpectrum`
- `erbSpectrum`

Data Types: `logical`

### **spectralEntropy** — Extract spectral entropy

`false` (default) | `true`

Extract spectral entropy, specified as `true` or `false`.

The spectral entropy is calculated on one of the following spectral representations, as specified by the `SpectralDescriptorInput` property:

- `linearSpectrum`
- `melSpectrum`
- `barkSpectrum`
- `erbSpectrum`

Data Types: `logical`

### **spectralFlatness** — Extract spectral flatness

`false` (default) | `true`

Extract spectral flatness, specified as `true` or `false`.

The spectral flatness is calculated on one of the following spectral representations, as specified by the `SpectralDescriptorInput` property:

- `linearSpectrum`
- `melSpectrum`
- `barkSpectrum`
- `erbSpectrum`

Data Types: `logical`

### **spectralFlux** — Extract spectral flux

`false` (default) | `true`

Extract spectral flux, specified as `true` or `false`.

The spectral flux is calculated on one of the following spectral representations, as specified by the `SpectralDescriptorInput` property:

- `linearSpectrum`
- `melSpectrum`
- `barkSpectrum`
- `erbSpectrum`

To set parameters of the spectral flux extraction, use `setExtractorParams`:

```
setExtractorParams(aFE, "spectralFlux", "Name", Value)
```

Settable parameters for the spectral flux extraction are:

- `"NormType"` -- Norm type used to calculate the spectral flux, specified as the comma-separated pair consisting of `"NormType"` and a 1 or 2. If unspecified, `NormType` defaults to 2.

Data Types: `logical`

### **spectralKurtosis** — Extract spectral kurtosis

`false` (default) | `true`

Extract spectral kurtosis, specified as `true` or `false`.

The spectral kurtosis is calculated on one of the following spectral representations, as specified by the `SpectralDescriptorInput` property:

- `linearSpectrum`
- `melSpectrum`
- `barkSpectrum`
- `erbSpectrum`

Data Types: `logical`

### **spectralRolloffPoint** — Extract spectral rolloff point

false (default) | true

Extract spectral rolloff point, specified as true or false.

The spectral rolloff point is calculated on one of the following spectral representations, as specified by the SpectralDescriptorInput property:

- linearSpectrum
- melSpectrum
- barkSpectrum
- erbSpectrum

To set parameters of the spectral rolloff point extraction, use setExtractorParams:

```
setExtractorParams(aFE, "spectralRolloffPoint", "Name", Value)
```

Settable parameters for the spectral flux extraction are:

- "Threshold" -- Threshold of the rolloff point, specified as the comma-separated pair consisting of "Threshold" and a scalar in the range (0, 1). If unspecified, Threshold defaults to 0.95.

Data Types: logical

### **spectralSkewness** — Extract spectral skewness

false (default) | true

Extract spectral skewness, specified as true or false.

The spectral skewness is calculated on one of the following spectral representations, as specified by the SpectralDescriptorInput property:

- linearSpectrum
- melSpectrum
- barkSpectrum
- erbSpectrum

Data Types: logical

### **spectralSlope** — Extract spectral slope

false (default) | true



Extract spectral slope, specified as `true` or `false`.

The spectral slope is calculated on one of the following spectral representations, as specified by the `SpectralDescriptorInput` property:

- `linearSpectrum`
- `melSpectrum`
- `barkSpectrum`
- `erbSpectrum`

Data Types: `logical`

### **spectralSpread** — Extract spectral spread

`false` (default) | `true`

Extract spectral spread, specified as `true` or `false`.

The spectral spread is calculated on one of the following spectral representations, as specified by the `SpectralDescriptorInput` property:

- `linearSpectrum`
- `melSpectrum`
- `barkSpectrum`
- `erbSpectrum`

Data Types: `logical`

### **pitch** — Extract pitch

`false` (default) | `true`

Extract pitch, specified as `true` or `false`.

To set parameters of the pitch extraction, use `setExtractorParams`:

```
setExtractorParams(aFE, "pitch", "Name", Value)
```

Settable parameters for the pitch extraction are:

- `"Method"` -- Method used to calculate the pitch, specified as the comma-separated pair consisting of `"Method"` and `"PEF"`, `"NCF"`, `"CEP"`, `"LHS"`, or `"SRH"`. If unspecified, `Method` defaults to `"NCF"`. For a description of available pitch extraction methods, see `pitch`.

- "Range" -- Range within to search for the pitch in Hz, specified as the comma-separated pair consisting of "Range" and a two-element row vector of increasing values. If unspecified, Range defaults to [50, 400].
- "MedianFilterLength" -- Median filter length used to smooth pitch estimates over time, specified as the comma-separated pair consisting of "MedianFilterLength" and a positive integer. If unspecified, MedianFilterLength defaults to 1 (no median filtering).

Data Types: logical

### **harmonicRatio** — Extract harmonic ratio

false (default) | true

Extract harmonic ratio, specified as true or false.

Data Types: logical

## Object Functions

extract	Extract audio features
setExtractorParams	Set nondefault parameter values for individual feature extractors
info	Output mapping and individual feature extractor parameters

## Examples

### **Extract Multiple Audio Features**

Read in an audio signal.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioFeatureExtractor` object that extracts the MFCC, delta MFCC, delta-delta MFCC, pitch, and spectral centroid of an audio signal. Use a 30 ms analysis window with 20 ms overlap.

```
aFE = audioFeatureExtractor( ...  
    "SampleRate",fs, ...  
    "Window",hamming(round(0.03*fs),"periodic"), ...  
    "OverlapLength",round(0.02*fs), ...  
    "mfcc",true, ...
```

```
"mfccDelta",true, ...  
"mfccDeltaDelta",true, ...  
"pitch",true, ...  
"spectralCentroid",true);
```

Call `extract` to extract the audio features from the audio signal.

```
features = extract(aFE, audioIn);
```

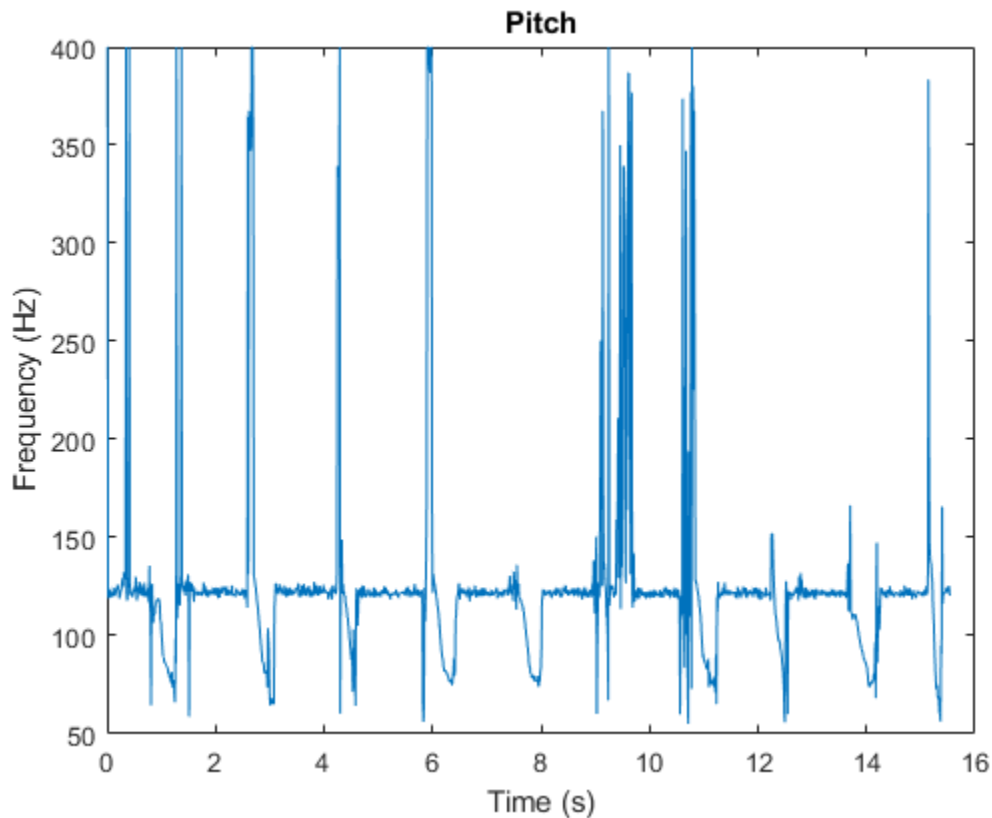
Use `info` to determine which column of the feature extraction matrix corresponds to the requested pitch extraction.

```
idx = info(aFE)
```

```
idx = struct with fields:  
    mfcc: [1 2 3 4 5 6 7 8 9 10 11 12 13]  
    mfccDelta: [14 15 16 17 18 19 20 21 22 23 24 25 26]  
    mfccDeltaDelta: [27 28 29 30 31 32 33 34 35 36 37 38 39]  
    spectralCentroid: 40  
    pitch: 41
```

Plot the detected pitch over time.

```
t = linspace(0, size(audioIn,1)/fs, size(features,1));  
plot(t, features(:, idx.pitch))  
title('Pitch')  
xlabel('Time (s)')  
ylabel('Frequency (Hz)')
```



### Extract Features from Dataset

Create an audio datastore that points to audio samples included with Audio Toolbox®.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');  
ads = audioDatastore(folder);
```

Find all files that correspond to a sample rate of 44.1 kHz and then subset the datastore.

```
keepFile = cellfun(@(x) contains(x, '44p1'), ads.Files);  
ads = subset(ads, keepFile);
```

Convert the data to a tall array. tall arrays are evaluated only when you request them explicitly using gather. MATLAB® automatically optimizes the queued calculations by minimizing the number of passes through the data. If you have Parallel Computing Toolbox™, you can spread the calculations across multiple machines. The audio data is represented as an  $M$ -by-1 tall cell array, where  $M$  is the number of files in the audio datastore.

```
adsTall = tall(ads)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 12).
```

```
adsTall =
```

```
Mx1 tall cell array

    { 539648x1 double}
    { 227497x1 double}
    {   8000x1 double}
    { 685056x1 double}
    { 882688x2 double}
    {1115760x2 double}
    { 505200x2 double}
    {3195904x2 double}
      :
      :
      :
```

Create an `audioFeatureExtractor` object to extract the mel spectrum, Bark spectrum, ERB spectrum, and linear spectrum from each audio file. Use the default analysis window and overlap length for the spectrum extraction.

```
aFE = audioFeatureExtractor('SampleRate',44.1e3, ...
    'melSpectrum',true, ...
    'barkSpectrum',true, ...
    'erbSpectrum',true, ...
    'linearSpectrum',true);
```

Define a cellfun function so that audio features are extracted from each cell of the tall array. Call `gather` to evaluate the tall array.

```
specsTall = cellfun(@(x)extract(aFE,x),adsTall,"UniformOutput",false);
specs = gather(specsTall);
```

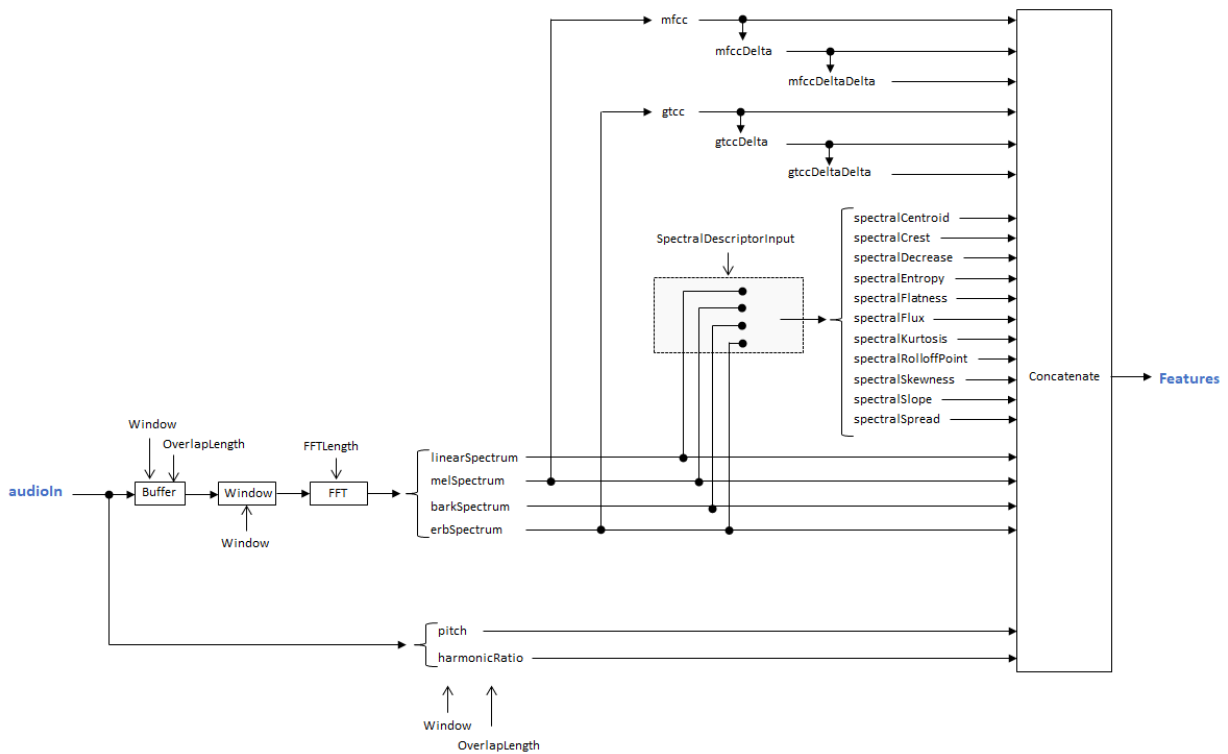
```
Evaluating tall expression using the Parallel Pool 'local':  
- Pass 1 of 1: Completed in 20 sec  
Evaluation completed in 20 sec
```

The `specs` variable returned from `gather` is an *numFiles*-by-1 cell array, where *numFiles* is the number of files in the datastore. Each element of the cell array is a *numHops*-by-*numFeatures*-by-*numChannels* array, where the number of hops and number of channels depends on the length and number of channels of the audio file, and the number of features is the requested number of features from the audio data.

```
numFiles = numel(specs)  
  
numFiles = 12  
  
[numHops1,numFeaturesFile1,numChannelsFile1] = size(specs{1})  
  
numHops1 = 1053  
  
numFeaturesFile1 = 620  
  
numChannelsFile1 = 1  
  
[numHops2,numFeaturesFile2,numChannelsFile2] = size(specs{2})  
  
numHops2 = 443  
  
numFeaturesFile2 = 620  
  
numChannelsFile2 = 1
```

## Algorithms

The `audioFeatureExtractor` creates a feature extraction pipeline based on your selected features. To reduce computations, `audioFeatureExtractor` reuses intermediary representations. Some intermediate representations can be output as features:



For example, to create an object that extracts the centroid of the Bark spectrum, the flux of the Bark spectrum, the pitch, the harmonic ratio, and the delta-delta of the MFCC, specify the `audioFeatureExtractor` as:

```
aFE = audioFeatureExtractor( ...
    "SpectralDescriptorInput", "barkSpectrum", ...
    "spectralCentroid", true, ...
    "spectralFlux", true, ...
    "pitch", true, ...
    "harmonicRatio", true, ...
    "mfccDeltaDelta", true)
```

aFE =

audioFeatureExtractor with properties:

Properties

Window: [1024x1 double]  
OverlapLength: 512

## 4 Classes in Audio Toolbox

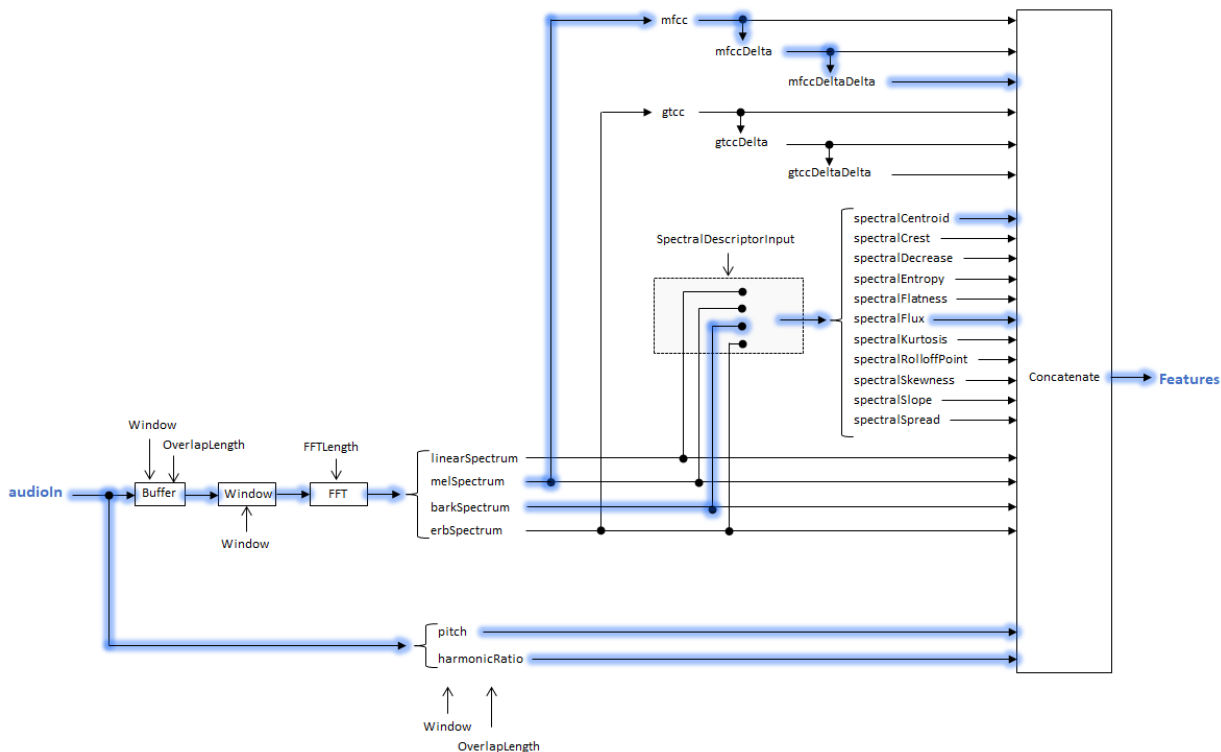
```
SampleRate: 44100  
FFTLength: []  
SpectralDescriptorInput: 'barkSpectrum'
```

Enabled Features  
mfccDeltaDelta, spectralCentroid, spectralFlux, pitch, harmonicRatio

Disabled Features  
linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta  
gtcc, gtccDelta, gtccDeltaDelta, spectralCrest, spectralDecrease, spectralEntropy  
spectralFlatness, spectralKurtosis, spectralRolloffPoint, spectralSkewness, spectralSlope, spectralSpread

To extract a feature, set the corresponding property to true.  
For example, `obj.mfcc = true`, adds `mfcc` to the list of enabled features.

This configuration corresponds to the highlighted feature extraction pipeline:





**Note** Because `audioFeatureExtractor` reuses intermediary representations, the features output from `audioFeatureExtractor` may not correspond with the default configuration of features output by corresponding individual feature extractors.

---

## See Also

**Audio Labeler** | `audioDataAugmenter` | `audioDatastore` | `cellfun` | `gather` | `subset` | `tall`

**Introduced in R2019b**

## removeAugmentationMethod

Remove custom augmentation method

### Syntax

```
removeAugmentationMethod(aug,algorithmName)
```

### Description

`removeAugmentationMethod(aug,algorithmName)` removes the custom augmentation algorithm from an `audioDataAugmenter` object.

### Examples

#### Remove Augmentation Method

Create a default `audioDataAugmenter` object.

```
aug = audioDataAugmenter
```

```
aug =  
    audioDataAugmenter with properties:
```

```
        AugmentationMode: 'sequential'  
AugmentationParameterSource: 'random'  
        NumAugmentations: 1  
    TimeStretchProbability: 0.5000  
        SpeedupFactorRange: [0.8000 1.2000]  
    PitchShiftProbability: 0.5000  
        SemitoneShiftRange: [-2 2]  
    VolumeControlProbability: 0.5000  
        VolumeGainRange: [-3 3]  
    AddNoiseProbability: 0.5000  
        SNRRange: [0 10]  
    TimeShiftProbability: 0.5000
```

```
TimeShiftRange: [-0.0050 0.0050]
```

Add a custom augmentation method that applies a random DC offset.

```
algorithmName = 'DCOffset';
algorithmHandle = @(x)x+rand(1,'like',x);
addAugmentationMethod(aug,algorithmName,algorithmHandle)
aug
```

```
aug =
  audioDataAugmenter with properties:

    AugmentationMode: 'sequential'
  AugmentationParameterSource: 'random'
    NumAugmentations: 1
  TimeStretchProbability: 0.5000
    SpeedupFactorRange: [0.8000 1.2000]
  PitchShiftProbability: 0.5000
    SemitoneShiftRange: [-2 2]
  VolumeControlProbability: 0.5000
    VolumeGainRange: [-3 3]
  AddNoiseProbability: 0.5000
    SNRRange: [0 10]
  TimeShiftProbability: 0.5000
    TimeShiftRange: [-0.0050 0.0050]
  DCOffsetProbability: 0.5000
```

Remove the custom augmentation method.

```
removeAugmentationMethod(aug,algorithmName)
aug
```

```
aug =
  audioDataAugmenter with properties:

    AugmentationMode: 'sequential'
  AugmentationParameterSource: 'random'
    NumAugmentations: 1
  TimeStretchProbability: 0.5000
    SpeedupFactorRange: [0.8000 1.2000]
  PitchShiftProbability: 0.5000
    SemitoneShiftRange: [-2 2]
  VolumeControlProbability: 0.5000
```

```
VolumeGainRange: [-3 3]
AddNoiseProbability: 0.5000
SNRRange: [0 10]
TimeShiftProbability: 0.5000
TimeShiftRange: [-0.0050 0.0050]
```

## Input Arguments

### **aug** — Audio data augments

audioDataAugmenter object

audioDataAugmenter object.

### **algorithmName** — Algorithm name

character vector | string

Algorithm name, specified as a character vector or string. `algorithmName` must match the algorithm name you used to add the algorithm using `addAugmentationMethod`.

Data Types: `char` | `string`

## See Also

`addAugmentationMethod` | `audioDataAugmenter`

**Introduced in R2019b**

# augment

Augment audio data

## Syntax

```
data = augment(aug, audioIn)
data = augment(aug, audioIn, fs)
```

## Description

`data = augment(aug, audioIn)` returns a table containing augmented audio data and information about the augmentation applied.

`data = augment(aug, audioIn, fs)` specifies the sample rate of the audio input.

## Examples

### Apply Random Sequential Augmentations

Read in an audio signal and listen to it.

```
[audioIn, fs] = audioread("Counting-16-44p1-mono-15secs.wav");
sound(audioIn, fs)
```

Create an `audioDataAugmenter` object that applies time stretching, volume control, and time shifting in cascade. Apply each of the augmentations with 80% probability. Set `NumAugmentations` to 5 to output five independently augmented signals. To skip pitch shifting and noise addition for each augmentation, set the respective probabilities to 0. Define parameter ranges for each relevant augmentation algorithm.

```
augmenter = audioDataAugmenter( ...
    "AugmentationMode", "sequential", ...
    "NumAugmentations", 5, ...
    ...
)
```

```

"TimeStretchProbability",0.8, ...
"SpeedupFactorRange", [1.3,1.4], ...
...
"PitchShiftProbability",0, ...
...
"VolumeControlProbability",0.8, ...
"VolumeGainRange", [-5,5], ...
...
"AddNoiseProbability",0, ...
...
"TimeShiftProbability",0.8, ...
"TimeShiftRange", [-500e-3,500e-3])

```

```

augmenter =

```

```

    audioDataAugmenter with properties:

```

```

        AugmentationMode: "sequential"
    AugmentationParameterSource: 'random'
        NumAugmentations: 5
        TimeStretchProbability: 0.8000
        SpeedupFactorRange: [1.3000 1.4000]
        PitchShiftProbability: 0
        VolumeControlProbability: 0.8000
        VolumeGainRange: [-5 5]
        AddNoiseProbability: 0
        TimeShiftProbability: 0.8000
        TimeShiftRange: [-0.5000 0.5000]

```

Call `augment` on the audio to create 5 augmentations. The augmented audio is returned in a table with variables `Audio` and `AugmentationInfo`. The number of rows in the table is defined by `NumAugmentations`.

```

data = augment(augmenter, audioIn, fs)

```

```

data=5x2 table
      Audio      AugmentationInfo
-----
{685056x1 double}  [1x1 struct]
{685056x1 double}  [1x1 struct]
{505183x1 double}  [1x1 struct]
{685056x1 double}  [1x1 struct]
{490728x1 double}  [1x1 struct]

```

In the current augmentation pipeline, augmentation parameters are assigned randomly from within the specified ranges. To determine the exact parameters used for an augmentation, inspect `AugmentationInfo`.

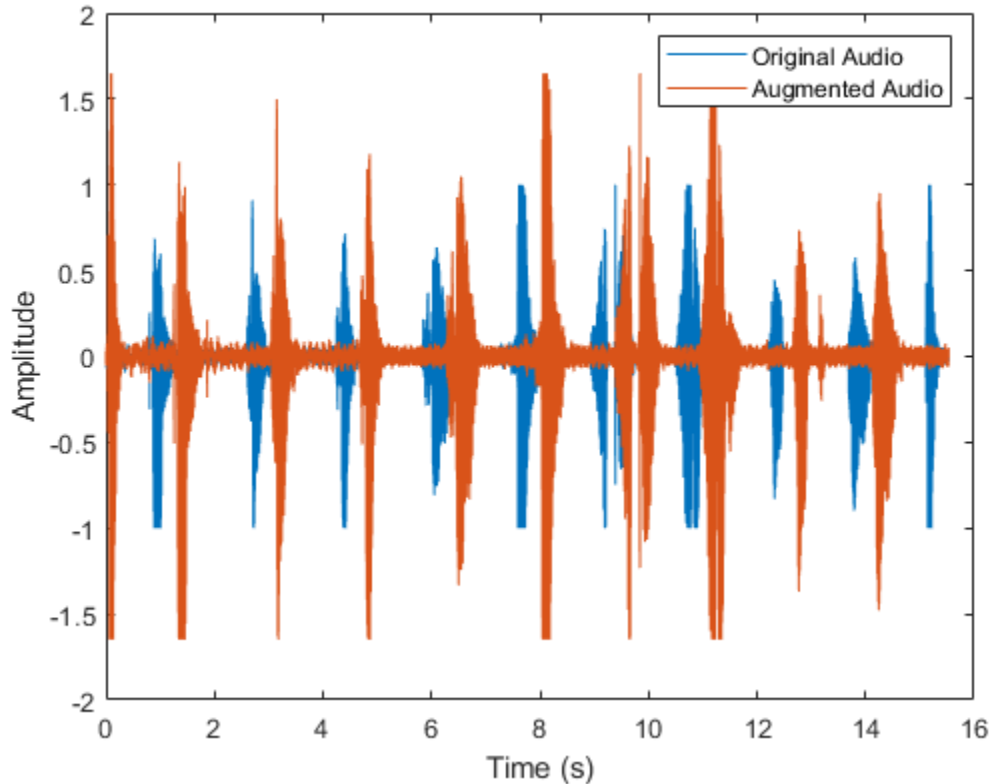
```
augmentationToInspect = ;  
data.AugmentationInfo(augmentationToInspect)
```

```
ans = struct with fields:  
  SpeedupFactor: 1  
  VolumeGain: 4.3399  
  TimeShift: 0.4502
```

Listen to the augmentation you are inspecting. Plot time representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};  
sound(augmentation, fs)
```

```
t = (0:(numel(audioIn)-1))/fs;  
taug = (0:(numel(augmentation)-1))/fs;  
plot(t, audioIn, taug, augmentation)  
legend("Original Audio", "Augmented Audio")  
ylabel("Amplitude")  
xlabel("Time (s)")
```



### Apply Specified Sequential Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");  
sound(audioIn,fs)
```

Create an `audioDataAugmenter` object that applies time stretching, pitch shifting, and noise corruption in cascade. Specify the time stretch speedup factors as 0.9, 1.1, and 1.2. Specify the pitch shifting in semitones as -2, -1, 1, and 2. Specify the noise corruption SNR as 10 dB and 15 dB.



```

augmenter = audioDataAugmenter( ...
    "AugmentationMode","sequential", ...
    "AugmentationParameterSource","specify", ...
    "SpeedupFactor",[0.9,1.1,1.2], ...
    "ApplyTimeStretch",true, ...
    "ApplyPitchShift",true, ...
    "SemitoneShift",[-2,-1,1,2], ...
    "SNR",[10,15], ...
    "ApplyVolumeControl",false, ...
    "ApplyTimeShift",false)

```

```

augmenter =
    audioDataAugmenter with properties:

```

```

        AugmentationMode: "sequential"
    AugmentationParameterSource: "specify"
        ApplyTimeStretch: 1
            SpeedupFactor: [0.9000 1.1000 1.2000]
        ApplyPitchShift: 1
            SemitoneShift: [-2 -1 1 2]
    ApplyVolumeControl: 0
        ApplyAddNoise: 1
            SNR: [10 15]
    ApplyTimeShift: 0

```

Call `augment` on the audio to create 24 augmentations. The augmentations represent every combination of the specified augmentation parameters ( $3 \times 4 \times 2 = 24$ ).

```

data = augment(augmenter, audioIn, fs)

```

```

data=24x2 table
      Audio          AugmentationInfo
-----
{761243x1 double}  [1x1 struct]
{622888x1 double}  [1x1 struct]
{571263x1 double}  [1x1 struct]
{761243x1 double}  [1x1 struct]
{622888x1 double}  [1x1 struct]
{571263x1 double}  [1x1 struct]
{761243x1 double}  [1x1 struct]
{622888x1 double}  [1x1 struct]
{571263x1 double}  [1x1 struct]
{761243x1 double}  [1x1 struct]

```

```
{622888x1 double}    [1x1 struct]
{571263x1 double}    [1x1 struct]
{761243x1 double}    [1x1 struct]
{622888x1 double}    [1x1 struct]
{571263x1 double}    [1x1 struct]
{761243x1 double}    [1x1 struct]
:
```

You can check the parameter configuration of each augmentation using the `AugmentationInfo` table variable.

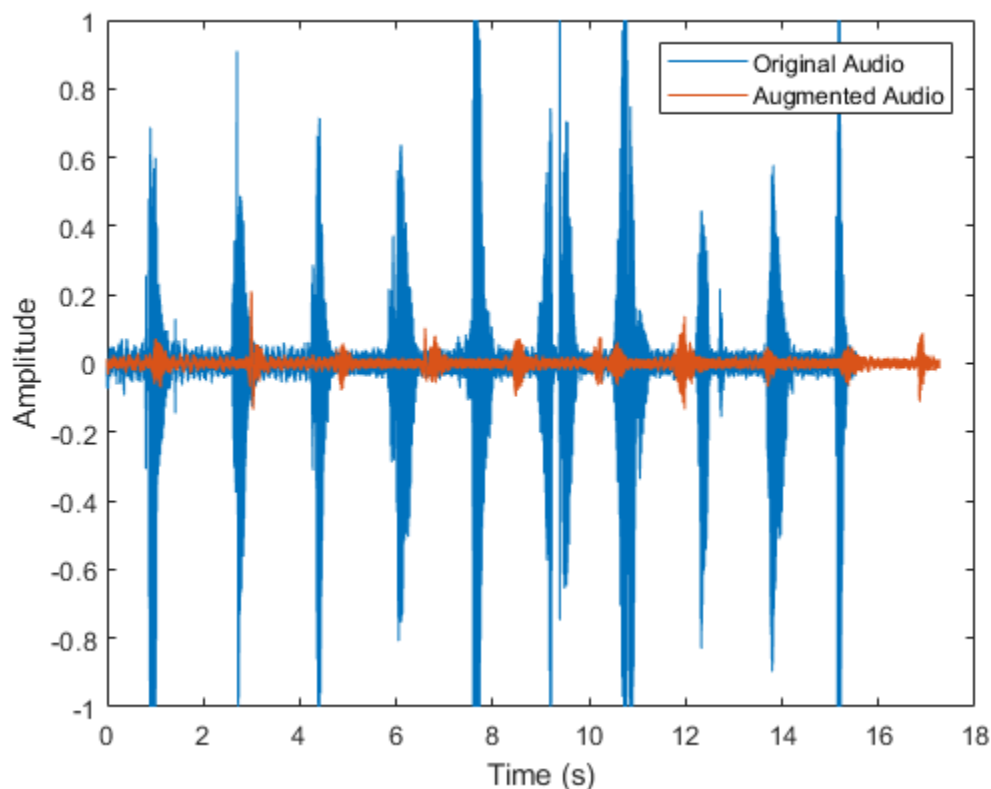
```
augmentationToInspect = ;
data.AugmentationInfo(augmentationToInspect)

ans = struct with fields:
  SpeedupFactor: 0.9000
  SemitoneShift: -2
  SNR: 10
```

Listen to the augmentation you are inspecting. Plot the time-domain representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation, fs)

t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t, audioIn, taug, augmentation)
legend("Original Audio", "Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



### Apply Random Independent Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioDataAugmenter` object that applies noise corruption, and time shifting in parallel branches. For the noise corruption branch, randomly apply noise with an SNR in the range 0 dB to 20 dB. For the time shifting branch, randomly apply time shifting in the range -300 ms to 300 ms. Apply augmentation 2 times for each branch, for 4 total augmentations.

```
augmenter = audioDataAugmenter( ...
    "AugmentationMode","independent", ...
    "AugmentationParameterSource","random", ...
    "NumAugmentations",2, ...
    "ApplyTimeStretch",false, ...
    "ApplyPitchShift",false, ...
    "ApplyVolumeControl",false, ...
    "SNRRange",[0,20], ...
    "TimeShiftRange",[-300e-3,300e-3])

augmenter =
    audioDataAugmenter with properties:

        AugmentationMode: "independent"
    AugmentationParameterSource: "random"
        NumAugmentations: 2
        ApplyTimeStretch: 0
        ApplyPitchShift: 0
    ApplyVolumeControl: 0
        ApplyAddNoise: 1
            SNRRange: [0 20]
        ApplyTimeShift: 1
        TimeShiftRange: [-0.3000 0.3000]
```

Call `augment` on the audio to create 3 augmentations.

```
data = augment(augmenter, audioIn, fs);
```

You can check the parameter configuration of each augmentation using the `AugmentationInfo` table variable.

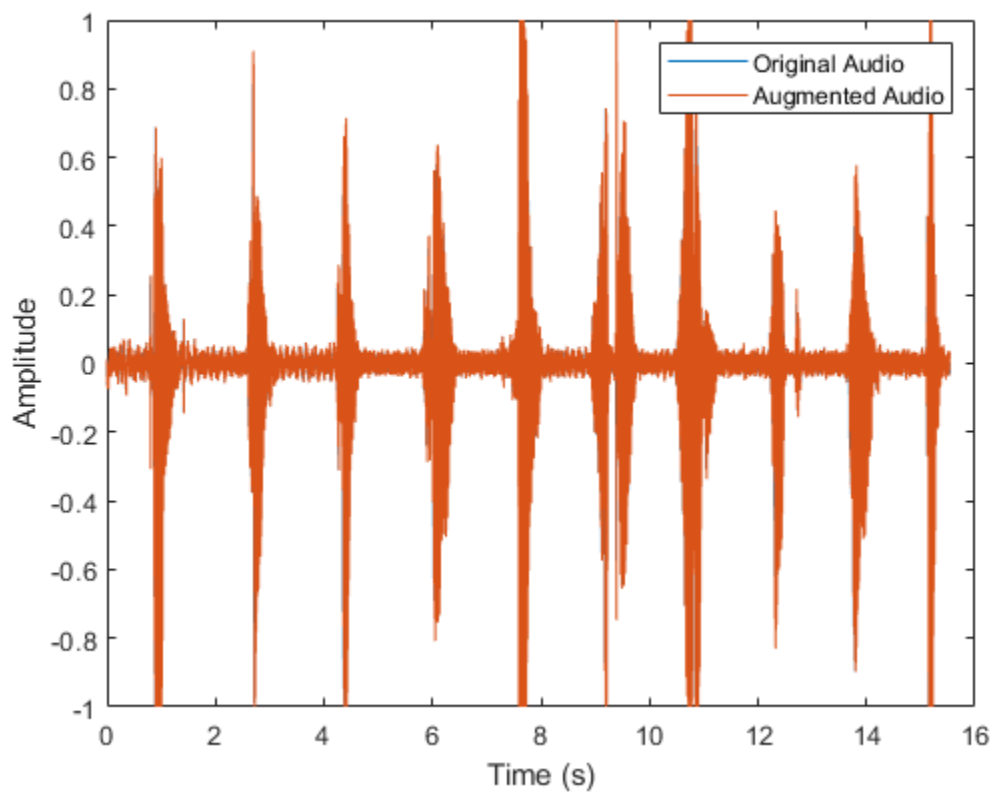
```
augmentationToInspect = ;
data.AugmentationInfo{augmentationToInspect}

ans = struct with fields:
    TimeShift: 0.0016
```

Listen to the audio you are inspecting. Plot the time-domain representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation, fs)
```

```
t = (0:(numel(audioIn)-1))/fs;  
taug = (0:(numel(augmentation)-1))/fs;  
plot(t,audioIn,taug,augmentation)  
legend("Original Audio","Augmented Audio")  
ylabel("Amplitude")  
xlabel("Time (s)")
```



### Apply Specified Independent Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioDataAugmenter` object that applies volume control, noise corruption, and time shifting in parallel branches.

```
augmenter = audioDataAugmenter( ...  
    "AugmentationMode","independent", ...  
    "AugmentationParameterSource","specify", ...  
    "ApplyTimeStretch",false, ...  
    "ApplyPitchShift",false, ...  
    "VolumeGain",2, ...  
    "SNR",0, ...  
    "TimeShift",2)
```

```
augmenter =  
    audioDataAugmenter with properties:
```

```
    AugmentationMode: "independent"  
    AugmentationParameterSource: "specify"  
    ApplyTimeStretch: 0  
    ApplyPitchShift: 0  
    ApplyVolumeControl: 1  
    VolumeGain: 2  
    ApplyAddNoise: 1  
    SNR: 0  
    ApplyTimeShift: 1  
    TimeShift: 2
```

Call `augment` on the audio to create 3 augmentations.

```
data = augment(augmenter,audioIn,fs)
```

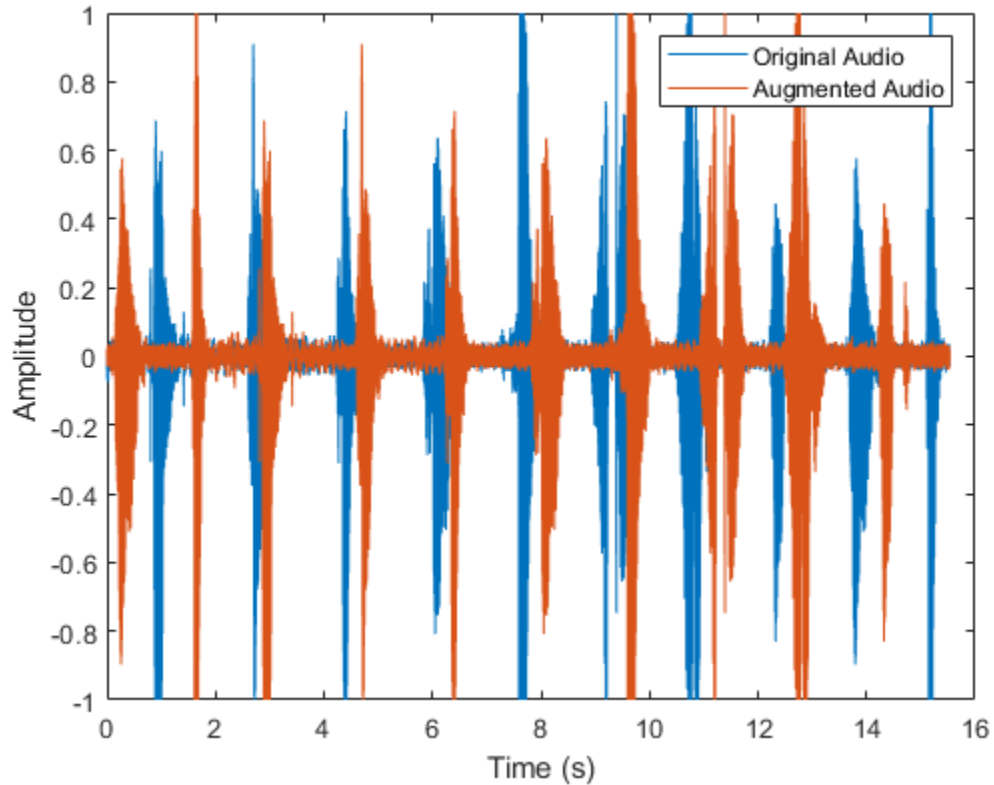
```
data=3x2 table  
    Audio      AugmentationInfo  
    _____  _____  
    {685056x1 double}  {1x1 struct}  
    {685056x1 double}  {1x1 struct}  
    {685056x1 double}  {1x1 struct}
```

You can check the parameter configuration of each augmentation using the `AugmentationInfo` table variable.

```
augmentationToInspect = ;  
data.AugmentationInfo{augmentationToInspect}  
  
ans = struct with fields:  
    TimeShift: 2
```

Listen to the audio you are inspecting. Plot the time-domain representations of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};  
sound(augmentation, fs)  
  
t = (0:(numel(audioIn)-1))/fs;  
taug = (0:(numel(augmentation)-1))/fs;  
plot(t, audioIn, taug, augmentation)  
legend("Original Audio", "Augmented Audio")  
ylabel("Amplitude")  
xlabel("Time (s)")
```



### Augment Audio Dataset

The `audioDataAugmenter` supports multiple workflows for augmenting your datastore, including:

- Offline augmentation
- Augmentation using tall arrays
- Augmentation using transform datastores



In each workflow, begin by creating an audio datastore to point to your audio data. In this example, you create an audio datastore that points to audio samples included with Audio Toolbox™. Count the number of files in the dataset.

```
folder = fullfile(matlabroot, "toolbox", "audio", "samples");
ADS = audioDatastore(folder)
```

```
ADS =
  audioDatastore with properties:

    Files: {
        ' ...\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono
        ' ...\matlab\toolbox\audio\samples\AudioArray-16-16-4char
        ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p
        ... and 26 more
    }
  AlternateFileSystemRoots: {}
  OutputDataType: 'double'
  Labels: {}
```

```
numFilesInDataset = numel(ADS.Files)
```

```
numFilesInDataset = 29
```

Create an `audioDataAugmenter` that applies random sequential augmentations. Set `NumAugmentations` to 2.

```
aug = audioDataAugmenter('NumAugmentations',2)
```

```
aug =
  audioDataAugmenter with properties:

    AugmentationMode: 'sequential'
  AugmentationParameterSource: 'random'
    NumAugmentations: 2
  TimeStretchProbability: 0.5000
  SpeedupFactorRange: [0.8000 1.2000]
  PitchShiftProbability: 0.5000
  SemitoneShiftRange: [-2 2]
  VolumeControlProbability: 0.5000
  VolumeGainRange: [-3 3]
  AddNoiseProbability: 0.5000
  SNRRange: [0 10]
  TimeShiftProbability: 0.5000
```

```
TimeShiftRange: [-0.0050 0.0050]
```

### Offline Augmentation

To augment the audio dataset, create two augmentations of each file and then write the augmentations as WAV files.

```
while hasdata(ADS)
    [audioIn,info] = read(ADS);

    data = augment(aug,audioIn,info.SampleRate);

    [~,fn] = fileparts(info.FileName);
    for i = 1:size(data,1)
        augmentedAudio = data.Audio{i};

        % If augmentation caused an audio signal to have values outside of -1 and 1,
        % normalize the audio signal to avoid clipping when writing.
        if max(abs(augmentedAudio),[],'all')>1
            augmentedAudio = augmentedAudio/max(abs(augmentedAudio),[],'all');
        end

        audiowrite(sprintf('%s_aug%d.wav',fn,i),augmentedAudio,info.SampleRate)
    end
end
```

Create an `audioDatastore` that points to the augmented dataset and confirm that the number of files in the dataset is double the original number of files.

```
augmentedADS = audioDatastore(pwd)

augmentedADS =
    audioDatastore with properties:

        Files: {
            ' ...\Examples\audio-ex28074079\Ambiance-16-44p1-mono-12
            ' ...\Examples\audio-ex28074079\Ambiance-16-44p1-mono-12
            ' ...\Examples\audio-ex28074079\AudioArray-16-16-4channel
            ... and 55 more
        }
        AlternateFileSystemRoots: {}
        OutputDataType: 'double'
        Labels: {}
```

```
numFilesInAugmentedDataset = numel(augmentedADS.Files)
```

```
numFilesInAugmentedDataset = 58
```

### Augment Using Tall Arrays

When augmenting a dataset using tall arrays, the input data to the augments should be sampled at a consistent rate. Subset the original audio dataset to only include files with a sample rate of 44.1 kHz. Most datasets are already cleaned to have a consistent sample rate.

```
keepFile = cellfun(@(x)contains(x,'44p1'),ADS.Files);
ads44p1 = subset(ADS,keepFile);
fs = 44.1e3;
```

Convert the audio datastore to a tall array. Tall arrays are evaluated only when you request them explicitly using `gather`. MATLAB® automatically optimizes the queued calculations by minimizing the number of passes through the data. If you have the Parallel Computing Toolbox™, you can spread the calculations across multiple machines. The audio data is represented as an  $M$ -by-1 tall cell array, where  $M$  is the number of files in the audio datastore.

```
adsTall = tall(ads44p1)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
adsTall =
```

```
M×1 tall cell array
```

```
{ 539648×1 double}
{ 227497×1 double}
{   8000×1 double}
{ 685056×1 double}
{ 882688×2 double}
{1115760×2 double}
{ 505200×2 double}
{3195904×2 double}
      :
      :
```

Define a `cellfun` function so that augmentation is applied to each cell of the tall array. Call `gather` to evaluate the tall array.

```
augTall = cellfun(@(x)augment(aug,x,fs),adsTall,"UniformOutput",false);  
augmentedDataset = gather(augTall)
```

```
Evaluating tall expression using the Parallel Pool 'local':  
- Pass 1 of 1: Completed in 1 min 34 sec  
Evaluation completed in 1 min 34 sec
```

```
augmentedDataset=12x1 cell  
    {2x2 table}  
    {2x2 table}  
    {2x2 table}  
    {2x2 table}  
    {2x2 table}  
    {2x2 table}  
    {2x2 table}  
    {2x2 table}  
    {2x2 table}  
    {2x2 table}  
    {2x2 table}  
    {2x2 table}
```

The augmented dataset is returned as a *numFiles*-by-1 cell array, where *numFiles* is the number of files in the datastore. Each element of the cell array is a *numAugmentationsPerFile*-by-2 table, where *numAugmentationsPerFile* is the number of augmentations returned per file.

```
numFiles = numel(augmentedDataset)
```

```
numFiles = 12
```

```
numAugmentationsPerFile = size(augmentedDataset{1},1)
```

```
numAugmentationsPerFile = 2
```

### Augment Using Transform Datastore

You can perform online data augmentation while you train your machine learning application using a transform datastore. Call `transform` to create a new datastore that applies data augmentation while reading.

```
transformADS = transform(ADS,@(x,info)augment(aug,x,info),'IncludeInfo',true)
```

```
transformADS =  
    TransformedDatastore with properties:
```

```

UnderlyingDatastore: [1x1 audioDatastore]
  Transforms: {@(x,info)augment(aug,x,info)}
  IncludeInfo: 1

```

Call `read` to return the augmented first file from the transform datastore.

```
augmentedRead = read(transformADS)
```

```

augmentedRead=2x2 table
      Audio      AugmentationInfo
      -----      -----
      {539648x1 double}      [1x1 struct]
      {586683x1 double}      [1x1 struct]

```

## Input Arguments

### **aug** — Audio data augments

audioDataAugmenter object

audioDataAugmenter object.

### **audioIn** — Audio input

vector | matrix

Audio input, specified as a column vector or matrix of independent channels (columns).

Data Types: single | double

### **fs** — Sample rate (Hz)

44100 (default) | positive scalar

Sample rate in Hz, specified as a positive scalar. The allowable range of `fs` depends on the properties of the audioDataAugmenter object.

Data Types: single | double

## Output Arguments

### **data** — Augmented audio and augmentation information

table

Augmented audio and augmentation information, returned as a two-column `table`. The first column holds the augmented audio signal. The second column holds information about the applied augmentation methods. The number of rows in `data` corresponds to the number of output augmented signals. The number of output augmented signals depends on the property values of the object.

## See Also

`addAugmentationMethod` | `audioDataAugmenter` | `removeAugmentationMethod` | `table`

**Introduced in R2019b**

# addAugmentationMethod

Add custom augmentation method

## Syntax

```
addAugmentationMethod(aug,algorithmName,algorithmHandle)
addAugmentationMethod(aug,algorithmName,algorithmHandle,Name,Value)
```

## Description

`addAugmentationMethod(aug,algorithmName,algorithmHandle)` adds a custom augmentation algorithm to an `audioDataAugmenter` object.

`addAugmentationMethod(aug,algorithmName,algorithmHandle,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

## Examples

### Add Custom Augmentation Method

You can expand the capabilities of `audioDataAugmenter` by adding custom augmentation methods.

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
sound(audioIn,fs)
```

Create an `audioDataAugmenter` object. Set the probability of applying white noise to 0.

```
augmenter = audioDataAugmenter('AddNoiseProbability',0)
```

```
augmenter =
    audioDataAugmenter with properties:
```

```
        AugmentationMode: 'sequential'  
AugmentationParameterSource: 'random'  
    NumAugmentations: 1  
    TimeStretchProbability: 0.5000  
        SpeedupFactorRange: [0.8000 1.2000]  
    PitchShiftProbability: 0.5000  
        SemitoneShiftRange: [-2 2]  
VolumeControlProbability: 0.5000  
        VolumeGainRange: [-3 3]  
    AddNoiseProbability: 0  
    TimeShiftProbability: 0.5000  
        TimeShiftRange: [-0.0050 0.0050]
```

Specify a custom augmentation algorithm that applies pink noise. The `AddPinkNoise` algorithm is added to the `augmenter` properties.

```
algorithmName = 'AddPinkNoise';  
algorithmHandle = @(x)x+pinknoise(size(x),'like',x);  
addAugmentationMethod(augmenter,algorithmName,algorithmHandle)
```

`augmenter`

`augmenter =`

`audioDataAugmenter` with properties:

```
        AugmentationMode: 'sequential'  
AugmentationParameterSource: 'random'  
    NumAugmentations: 1  
    TimeStretchProbability: 0.5000  
        SpeedupFactorRange: [0.8000 1.2000]  
    PitchShiftProbability: 0.5000  
        SemitoneShiftRange: [-2 2]  
VolumeControlProbability: 0.5000  
        VolumeGainRange: [-3 3]  
    AddNoiseProbability: 0  
    TimeShiftProbability: 0.5000  
        TimeShiftRange: [-0.0050 0.0050]  
    AddPinkNoiseProbability: 0.5000
```

Set the probability of adding pink noise to 1.

```
augmenter.AddPinkNoiseProbability = 1
```



```

augmenter =
  audioDataAugmenter with properties:

        AugmentationMode: 'sequential'
  AugmentationParameterSource: 'random'
        NumAugmentations: 1
    TimeStretchProbability: 0.5000
    SpeedupFactorRange: [0.8000 1.2000]
    PitchShiftProbability: 0.5000
    SemitoneShiftRange: [-2 2]
  VolumeControlProbability: 0.5000
    VolumeGainRange: [-3 3]
    AddNoiseProbability: 0
    TimeShiftProbability: 0.5000
        TimeShiftRange: [-0.0050 0.0050]
  AddPinkNoiseProbability: 1

```

Augment the original signal and listen to the result. Inspect parameters of the augmentation algorithms applied.

```

data = augment(augmenter, audioIn, fs);
sound(data.Audio{1}, fs)

```

```

data.AugmentationInfo(1)

```

```

ans = struct with fields:
  SpeedupFactor: 1
  SemitoneShift: 0
  VolumeGain: 2.4803
  TimeShift: -0.0022
  AddPinkNoise: 'Applied'

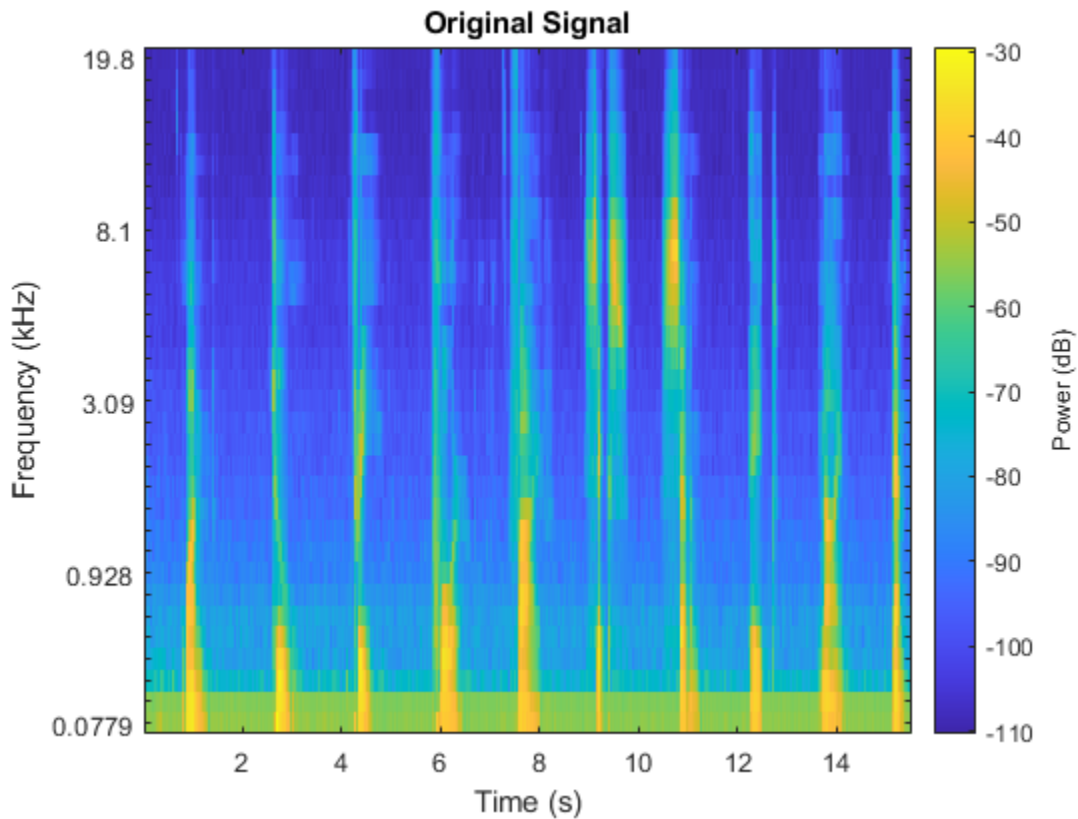
```

Plot the mel spectrograms of the original and augmented signals.

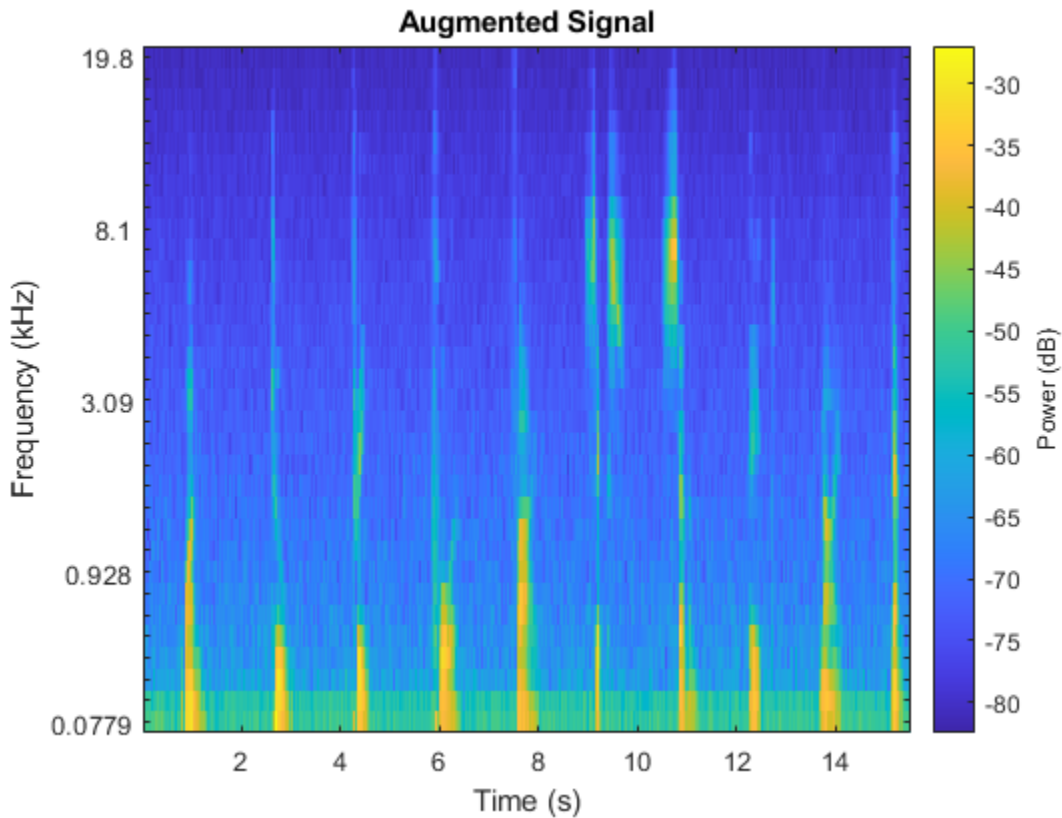
```

melSpectrogram(audioIn, fs)
title('Original Signal')

```



```
melSpectrogram(data.Audio{1}, fs)  
title('Augmented Signal')
```



### Specify Parameters of Custom Augmentation Method

In this example, you add a custom augmentation method that applies median filtering to your audio.

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");  
sound(audioIn,fs)
```

Create a random sequential augmentser that adds noise with an SNR range of 5 dB to 10 dB. Set the probability of applying volume control, time stretching, pitch shifting, and time shifting to 0. Set NumAugmentations to 4 to create 4 separate augmentations.

```
aug = audioDataAugmenter('NumAugmentations',4, ...  
    "AddNoiseProbability",1, ...  
    "SNRRange",[5,10], ...  
    "VolumeControlProbability",0, ...  
    "TimeStretchProbability",0, ...  
    "TimeShiftProbability",0, ...  
    "PitchShiftProbability",0)
```

```
aug =  
    audioDataAugmenter with properties:  
  
        AugmentationMode: 'sequential'  
        AugmentationParameterSource: 'random'  
        NumAugmentations: 4  
        TimeStretchProbability: 0  
        PitchShiftProbability: 0  
        VolumeControlProbability: 0  
        AddNoiseProbability: 1  
        SNRRange: [5 10]  
        TimeShiftProbability: 0
```

Call `addAugmentationMethod` with an algorithm name and function handle. Specify the algorithm name as `MedianFilter` and the function handle as `movmedian` with a 3-element window length. The augmentation is added to the properties of your `audioDataAugmenter` object.

```
algorithmName = 'MedianFilter';  
algorithmHandle = @(x)(movmedian(x,100));  
addAugmentationMethod(aug,algorithmName,algorithmHandle)
```

```
aug  
  
aug =  
    audioDataAugmenter with properties:  
  
        AugmentationMode: 'sequential'  
        AugmentationParameterSource: 'random'  
        NumAugmentations: 4  
        TimeStretchProbability: 0  
        PitchShiftProbability: 0
```

```

VolumeControlProbability: 0
  AddNoiseProbability: 1
    SNRRange: [5 10]
  TimeShiftProbability: 0
  MedianFilterProbability: 0.5000

```

Set the probability of applying median filtering to 80%.

```
aug.MedianFilterProbability = 0.8
```

```

aug =
  audioDataAugmenter with properties:

      AugmentationMode: 'sequential'
  AugmentationParameterSource: 'random'
      NumAugmentations: 4
      TimeStretchProbability: 0
      PitchShiftProbability: 0
      VolumeControlProbability: 0
      AddNoiseProbability: 1
        SNRRange: [5 10]
      TimeShiftProbability: 0
      MedianFilterProbability: 0.8000

```

Call `augment` on the audio to create 4 augmentations.

```
data = augment(aug, audioIn, fs);
```

You can check the parameter configuration of each augmentation using the `AugmentationInfo` table variable. If median filtering was applied for an augmentation, then `AugmentationInfo` lists the parameter as `'Applied'`. If median filtering was not applied for an augmentation, then `AugmentationInfo` lists the parameter as `'Bypassed'`.

```

augmentationToInspect = ;
data.AugmentationInfo(augmentationToInspect)

```

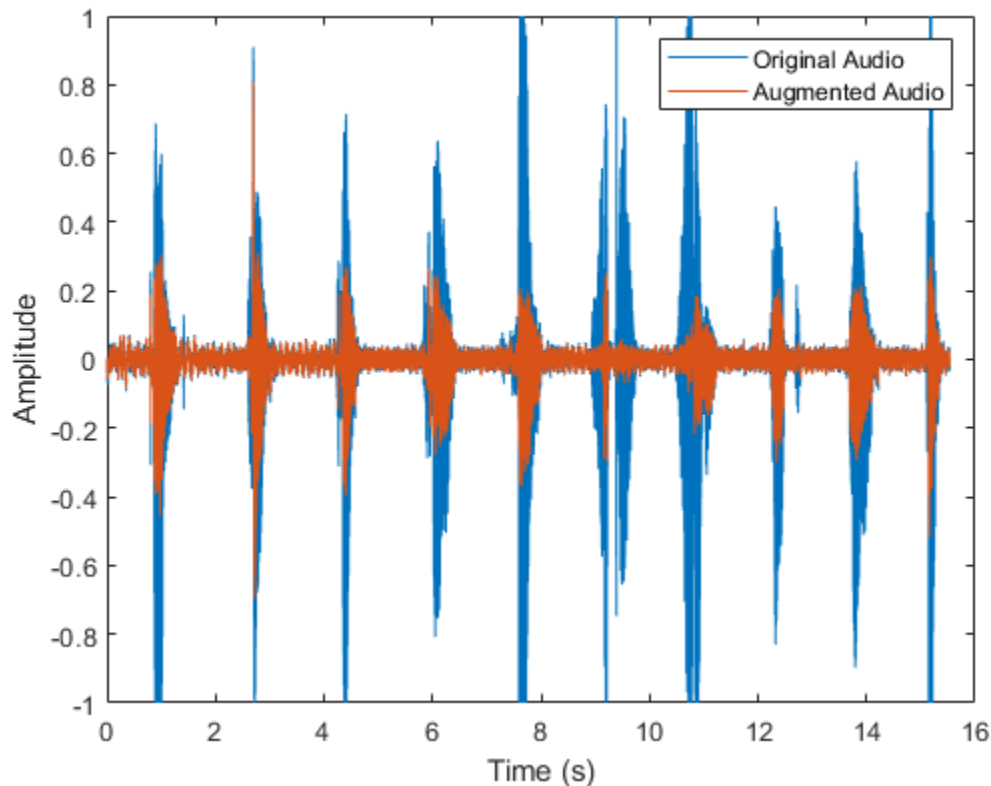
```

ans = struct with fields:
    SNR: 9.5787
  MedianFilter: 'Applied'

```

Listen to the audio you are inspecting. Plot the time-domain representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};  
sound(augmentation,fs)  
t = (0:(numel(audioIn)-1))/fs;  
taug = (0:(numel(augmentation)-1))/fs;  
plot(t,audioIn,taug,augmentation)  
legend("Original Audio","Augmented Audio")  
ylabel("Amplitude")  
xlabel("Time (s)")
```



You can specify additional parameters and corresponding parameter ranges (for use when `AugmentationParameterSource` is set to 'random') and parameter values (for use

when `AugmentationParameterSource` is set to `'specify'`). You must specify additional parameters, parameter ranges, and parameter values during your call to `addAugmentationMethod`.

Call `removeAugmentationMethod` to remove the `MedianFilter` augmentation method. Call `addAugmentationMethod` again, this time specifying an augmentation parameter, parameter range, and parameter value. The augmentation and parameter range is added to the properties of your `audioDataAugmenter` object.

```
removeAugmentationMethod(aug, 'MedianFilter')

algorithmName = 'MedianFilter';
augmentationParameter = 'MedianFilterWindowLength';
parameterRange = [1,200];
parameterValue = 100;

algorithmHandle = @(x,k)(movmedian(x,k));
addAugmentationMethod(aug,algorithmName,algorithmHandle, ...
    'AugmentationParameter',augmentationParameter, ...
    'ParameterRange',parameterRange, ...
    'ParameterValue',parameterValue)
```

`aug`

```
aug =
  audioDataAugmenter with properties:
      AugmentationMode: 'sequential'
      AugmentationParameterSource: 'random'
      NumAugmentations: 4
      TimeStretchProbability: 0
      PitchShiftProbability: 0
      VolumeControlProbability: 0
      AddNoiseProbability: 1
      SNRRange: [5 10]
      TimeShiftProbability: 0
      MedianFilterProbability: 0.5000
      MedianFilterWindowLengthRange: [1 200]
```

In the current augmentation pipeline configuration, the parameter value is not applicable. `ParameterValue` is applicable when `AugmentationParameterSource` is set to `'specify'`. Set `AugmentationParameterSource` to `'specify'` to enable the current parameter value.

```
aug.AugmentationParameterSource = 'specify'

aug =
  audioDataAugmenter with properties:

      AugmentationMode: 'sequential'
  AugmentationParameterSource: 'specify'
      ApplyTimeStretch: 1
          SpeedupFactor: 0.8000
      ApplyPitchShift: 1
          SemitoneShift: -3
  ApplyVolumeControl: 1
          VolumeGain: -3
      ApplyAddNoise: 1
          SNR: 5
      ApplyTimeShift: 1
          TimeShift: 0.0050
      ApplyMedianFilter: 1
  MedianFilterWindowLength: 100
```

Set `AugmentationParameterSource` to `random` and then call `augment`.

```
aug.AugmentationParameterSource = "random";
data = augment(aug, audioIn, fs);
```

If median filtering was applied for an augmentation, then `AugmentationInfo` lists the value applied.

```
augmentationToInspect = ;
data.AugmentationInfo(augmentationToInspect)
```

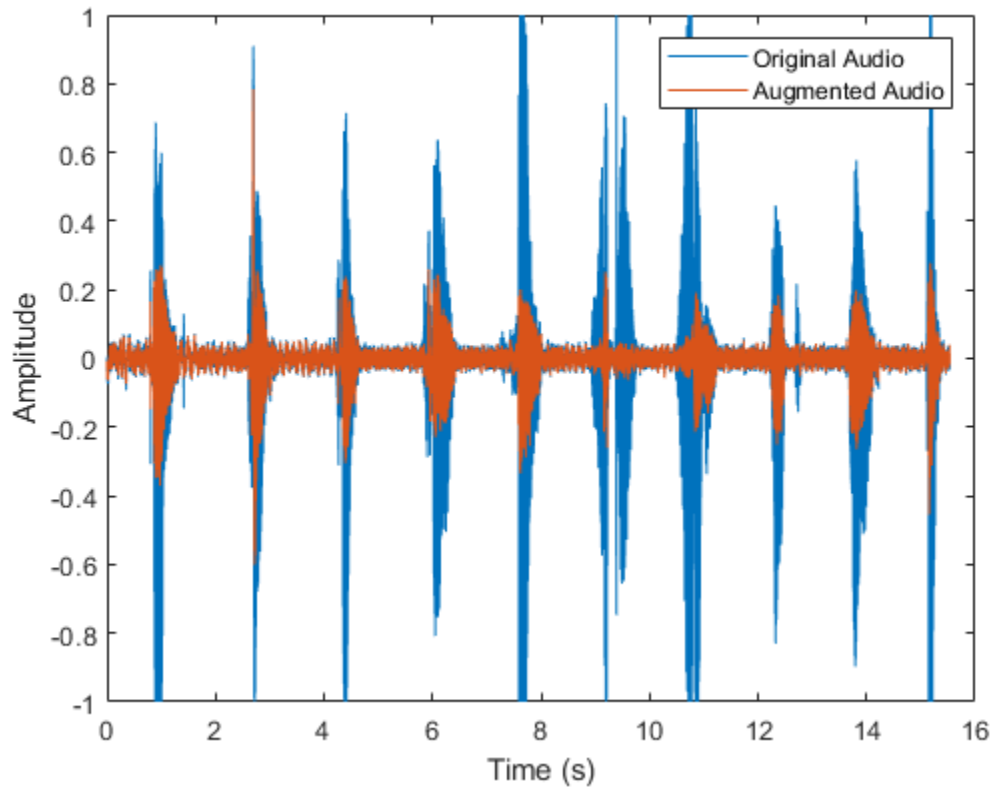
```
ans = struct with fields:
    SNR: 8.7701
    MedianFilter: 117.9847
```

Listen to the audio you are inspecting. Plot the time-domain representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation, fs)
t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t, audioIn, taug, augmentation)
```



```
legend("Original Audio","Augmented Audio")  
ylabel("Amplitude")  
xlabel("Time (s)")
```



### Specify Multiple Parameters of Custom Augmentation Method

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread('RockDrums-44p1-stereo-11secs.mp3');  
sound(audioIn,fs)
```

Create an `audioDataAugmenter` object that outputs 5 augmentations. Set the `AddNoiseProbability` to 0.

```
aug = audioDataAugmenter('NumAugmentations',5,'AddNoiseProbability',0);
```

Add reverberation as a custom augmentation algorithm. The `applyReverb` function creates a `reverberator` object, updates the sample rate, pre-delay, and wet/dry mix as indicated, and then applies reverberation. To minimize computational overhead, the `reverberator` object is persistent. The object is reset on every call to avoid mixing the reverberation tail between audio files.

type `applyReverb.m`

```
function audioOut = applyReverb(audio,preDelay,wetDryMix,sampleRate)
    persistent reverbObject
    if isempty(reverbObject)
        reverbObject = reverberator;
    end
    reverbObject.SampleRate = sampleRate;
    reverbObject.PreDelay = preDelay;
    reverbObject.WetDryMix = wetDryMix;

    audioOut = reverbObject(audio);
    reset(reverbObject)
end
```

Add `applyReverb` as a custom augmentation method. To specify multiple parameters for a custom method, specify the parameters, parameter ranges, and parameter values as cell arrays with the same number of cells. Set the probability of applying reverberation to 1.

```
algorithmName = 'Reverb';
algorithmHandle = @(x,preDelay,wetDryMix)applyReverb(x,preDelay,wetDryMix,fs);
parameters = {'PreDelay','WetDryMix'};
parameterRanges = {[0,1],[0,1]};
parameterValues = {0,0.3};

addAugmentationMethod(aug,algorithmName,algorithmHandle, ...
    'AugmentationParameter',parameters, ...
    'ParameterRange',parameterRanges, ...
    'ParameterValue',parameterValues)

aug.ReverbProbability = 1

aug =
    audioDataAugmenter with properties:
```

```

        AugmentationMode: 'sequential'
AugmentationParameterSource: 'random'
    NumAugmentations: 5
    TimeStretchProbability: 0.5000
    SpeedupFactorRange: [0.8000 1.2000]
    PitchShiftProbability: 0.5000
    SemitoneShiftRange: [-2 2]
    VolumeControlProbability: 0.5000
    VolumeGainRange: [-3 3]
    AddNoiseProbability: 0
    TimeShiftProbability: 0.5000
    TimeShiftRange: [-0.0050 0.0050]
    ReverbProbability: 1
    PreDelayRange: [0 1]
    WetDryMixRange: [0 1]

```

Call `augment` to create 5 augmentations.

```
data = augment(aug, audioIn, fs);
```

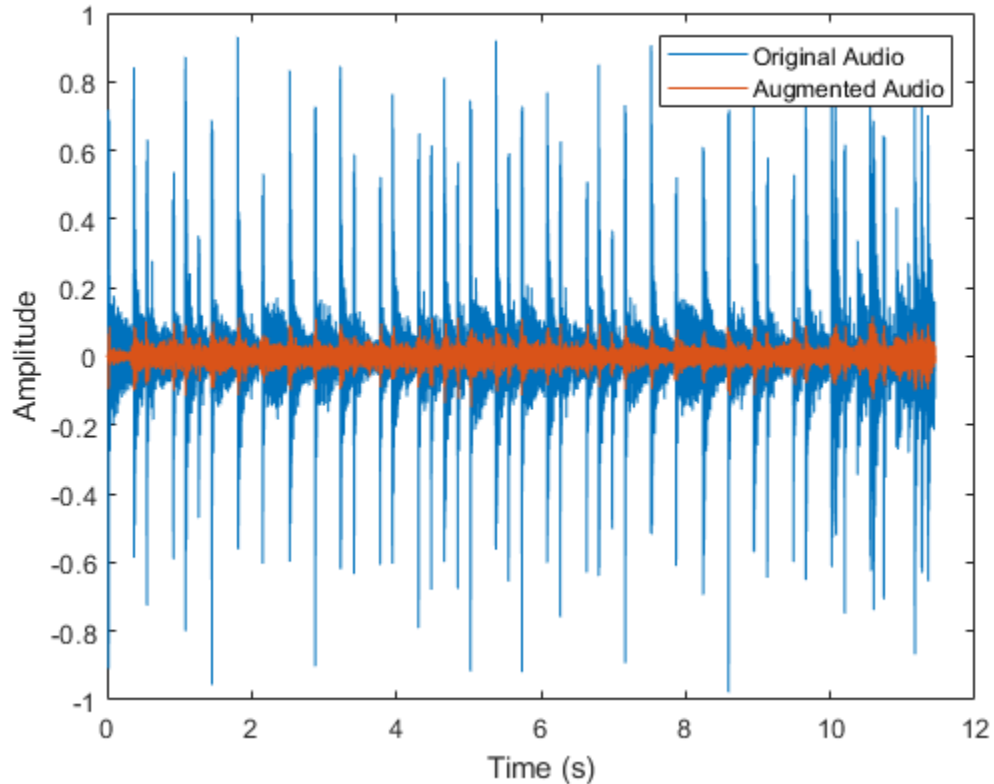
Check the configuration of each augmentation using `AugmentationInfo`.

```
augmentationToInspect = ;
data.AugmentationInfo(augmentationToInspect)
```

```
ans = struct with fields:
    SpeedupFactor: 1
    SemitoneShift: -1.4325
    VolumeGain: 0
    TimeShift: 0
    Reverb: [0.2760 0.4984]
```

Listen to the audio you are inspecting. Plot the time-domain representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation, fs)
t = (0:(size(audioIn,1)-1))/fs;
taug = (0:(size(augmentation,1)-1))/fs;
plot(t, audioIn(:,1), taug, augmentation(:,1))
legend("Original Audio", "Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



### Input Arguments

**aug** — Audio data augmenter

audioDataAugmenter object

audioDataAugmenter object.

**algorithmName** — Algorithm name

character vector | string

Algorithm name, specified as a character vector or string. `algorithmName` must be a unique property name on the `audioDataAugmenter`, `aug`.

Data Types: `char` | `string`

**algorithmHandle** — Handle to function that implements custom augmentation algorithm

`function_handle`

Handle to function that implements custom augmentation algorithm, specified as a `function_handle`.

Data Types: `function_handle`

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'AugmentationParameter', 'PreDelay'`

**AugmentationParameter** — Augmentation parameter

`character vector` | `string` | `cell array of character vectors` | `cell array of strings`

Augmentation parameter, specified as a character vector, string, cell array of character vectors, or cell array of strings.

Use cell arrays to create multiple augmentation parameters. If you create multiple augmentation parameters, you must also specify `ParameterRange` and `ParameterValue` as cell arrays containing information for each augmentation parameter.

Example: `'AugmentationParameter', 'PreDelay'`

Example: `'AugmentationParameter', {'PreDelay', 'HighCutFrequency'}`

Data Types: `char` | `string`

**ParameterRange** — Parameter range

`[0, 1]` (default) | `two-element vector of nondecreasing values` | `cell array of two-element vectors of nondecreasing values`

Parameter range, specified as a two-element vector of nondecreasing values (for a single parameter) or a cell array of two-element vectors of nondecreasing values (for multiple parameters).

Example: 'ParameterRange', [0,1]

Example: 'ParameterRange', {[0,1], [20,20000]}

### Dependencies

To enable this property, set the `AugmentationParameterSource` property of your `audioDataAugmenter` object to 'random'.

Data Types: single | double | cell

### ParameterValue — Parameter value

0 (default) | scalar | vector | cell array of scalars or vectors

Parameter value, specified as a scalar, vector, or cell array of scalars or vectors.

Example: 'ParameterValue', 0

Example: 'ParameterValue', [0,0.5,1]

Example: 'ParameterValue', {0,20000}

Example: 'ParameterValue', {[0,0.5,1],20000}

### Dependencies

To enable this property, set the `AugmentationParameterSource` property of your `audioDataAugmenter` to 'random'.

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

Complex Number Support: Yes

## See Also

`audioDataAugmenter` | `removeAugmentationMethod` | `reverberator`

### Introduced in R2019b

# audioDataAugmenter

Augment audio data

## Description

Enlarge your audio dataset using audio-specific augmentation techniques like pitch shifting, time-scale modification, time shifting, noise addition, and volume control. You can create cascaded or parallel augmentation pipelines to apply multiple algorithms deterministically or probabilistically.

## Creation

## Syntax

```
aug = audioDataAugmenter()  
aug = audioDataAugmenter(Name, Value)
```

## Description

`aug = audioDataAugmenter()` creates an audio data augmenter object with default property values.

`aug = audioDataAugmenter(Name, Value)` specifies nondefault properties for `aug` using one or more name-value pair arguments.

## Properties

### Augmentation Pipeline

**AugmentationMode** — Augmentation mode  
'sequential' (default) | 'independent'

Augmentation mode, specified as 'sequential' or 'independent'.

- 'sequential' -- Augmentation algorithms are applied sequentially (in series).
- 'independent' -- Augmentation algorithms are applied independently (in parallel).

Data Types: char | string

### **AugmentationParameterSource — Source of augmentation parameters**

'random' (default) | 'specify'

Source of augmentation parameters, specified as 'random' or 'specify'.

- 'random' -- Augmentation algorithms are applied probabilistically using a probability parameter and a range parameter.

For example, to create an `audioDataAugmenter` that applies time-stretching using a speedup factor between 0.5 and 1.5 with a 60% probability, enter the following in the Command Window:

```
aug = audioDataAugmenter('AugmentationParameterSource','random', ...  
                        'TimeStretchProbability',0.6, ...  
                        'SpeedupFactorRange',[0.5,1.5]);
```

When time-stretching is applied, the speedup factor is drawn from a uniform distribution centered at 1 (the mean of the range) with a minimum of 0.5 and a maximum of 1.5.

- 'specify' -- Augmentation algorithms are applied deterministically using a logical parameter and a specified parameter value. For example, to create an `audioDataAugmenter` that applies time-stretching using a 1.5 speedup factor with a 100% probability, enter the following in the Command Window:

```
aug = audioDataAugmenter('AugmentationParameterSource','specify', ...  
                        'ApplyTimeStretch',true, ...  
                        'SpeedupFactor',1.5);
```

Data Types: char | string

### **NumAugmentations — Number of augmented signals to output**

1 (default) | positive integer

Number of augmented signals to output, specified as a positive integer.



**Dependencies**

To enable this property, set `AugmentationParameterSource` to `'random'`.

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

**Stretch Time****TimeStretchProbability — Probability of applying time stretch**

`0.5` (default) | scalar in the range `[0, 1]`

Probability of applying time stretch, specified as a scalar in the range `[0, 1]`. Set the probability to `1` to apply time stretching every time you call `augment`. Set the probability to `0` to skip time stretching every time you call `augment`.

**Dependencies**

To enable this property, set `AugmentationParameterSource` to `'random'` and `AugmentationMode` to `'sequential'`.

Data Types: `single` | `double`

**SpeedupFactorRange — Range of time stretch speedup factor**

`[0.8 1.2]` (default) | two-element row vector of positive nondecreasing values

Range of time stretch speedup factor, specified as a two-element row vector of positive nondecreasing values.

**Dependencies**

To enable this property, set `AugmentationParameterSource` to `'random'`.

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

**ApplyTimeStretch — Apply time stretch**

`true` (default) | `false`

Apply time stretch, specified as `true` or `false`.

**Dependencies**

To enable this property, set `AugmentationParameterSource` to `'specify'`.

Data Types: `logical`

### **SpeedupFactor** — Time stretch speedup factor

0.8 (default) | real positive scalar | real positive vector

Time stretch speedup factor, specified as a scalar or vector of real positive values.

#### **Dependencies**

To enable this property, set `AugmentationParameterSource` to `'specify'`.

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

## **Shift Pitch**

### **PitchShiftProbability** — Probability of applying pitch shift

0.5 (default) | scalar in the range [0, 1]

Probability of applying pitch shift, specified as a scalar in the range [0, 1]. Set the probability to 1 to apply pitch shifting every time you call `augment`. Set the probability to 0 to skip pitch shifting every time you call `augment`.

#### **Dependencies**

To enable this property, set `AugmentationParameterSource` to `'random'` and `AugmentationMode` to `'sequential'`.

Data Types: `single` | `double`

### **SemitoneShiftRange** — Range of pitch shift (semitones)

[-2, 2] (default) | two-element row vector of nondecreasing values

Range of pitch shift in semitones, specified as a two-element row vector of nondecreasing values.

#### **Dependencies**

To enable this property, set `AugmentationParameterSource` to `'random'`.

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

### **ApplyPitchShift** — Apply pitch shift

true (default) | false

Apply pitch shift, specified as `true` or `false`.

### Dependencies

To enable this property, set `AugmentationParameterSource` to `'specify'`.

Data Types: `logical`

### SemitoneShift — Pitch shift (semitones)

-3 (default) | real scalar | real vector

Pitch shift in semitones, specified as a real scalar or vector.

### Dependencies

To enable this property, set `AugmentationParameterSource` to `'specify'`.

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

## Control Volume

### VolumeControlProbability — Probability of applying volume control

0.5 (default) | scalar in the range [0, 1]

Probability of applying volume control, specified as a scalar in the range [0, 1]. Set the probability to 1 to apply volume control every time you call `augment`. Set the probability to 0 to skip volume control every time you call `augment`.

### Dependencies

To enable this property, set `AugmentationParameterSource` to `'random'` and `AugmentationMode` to `'sequential'`.

Data Types: `single` | `double`

### VolumeGainRange — Range of volume gain (dB)

[-3, 3] (default) | two-element row vector of nondecreasing values

Range of volume gain in dB, specified as a two-element row vector of nondecreasing values.

### Dependencies

To enable this property, set `AugmentationParameterSource` to `'random'`.

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

### **ApplyVolumeControl** — Apply volume gain

`true` (default) | `false`

Apply volume gain, specified as `true` or `false`.

#### **Dependencies**

To enable this property, set `AugmentationParameterSource` to `'specify'`.

Data Types: `logical`

### **VolumeGain** — Volume gain (dB)

-3 (default) | scalar | vector

Volume gain in dB, specified as a scalar or vector.

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

## **Add Noise**

### **AddNoiseProbability** — Probability of applying noise addition

0.5 (default) | scalar in the range [0, 1]

Probability of applying Gaussian white noise addition, specified as a scalar in the range [0, 1]. Set the probability to 1 to add noise every time you call `augment`. Set the probability to 0 to skip adding noise every time you call `augment`.

#### **Dependencies**

To enable this property, set `AugmentationParameterSource` to `'random'` and `AugmentationMode` to `'sequential'`.

Data Types: `single` | `double`

### **SNRRange** — Range of noise addition SNR (dB)

[0, 10] (default) | two-element row vector of nondecreasing values

Range of noise addition SNR in dB, specified as a two-element row vector of nondecreasing values.

## Dependencies

To enable this property, set `AugmentationParameterSource` to `'range'`.

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

## ApplyAddNoise — Apply noise addition

`true` (default) | `false`

Apply Gaussian white noise addition, specified as `true` or `false`.

## Dependencies

To enable this property, set `AugmentationParameterSource` to `'specify'`.

Data Types: `logical`

## SNR — Noise addition SNR (dB)

`5` (default) | `scalar` | `vector`

Noise addition SNR in dB, specified as a scalar or vector.

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

## Shift Time

### TimeShiftProbability — Probability of applying time shift

`0.5` (default) | `scalar in the range [0, 1]`

Probability of applying time shift, specified as a scalar in the range `[0, 1]`. Set the probability to `1` to apply time shifting every time you call `augment`. Set the property to `0` to skip time shifting every time you call `augment`.

Time-shifting applies a circular shift on the time-domain audio data.

## Dependencies

To enable this property, set `AugmentationParameterSource` to `'random'` and `AugmentationMode` to `'sequential'`.

Data Types: `single` | `double`

### **TimeShiftRange — Range of time shift (s)**

`[-5e-3,5e3]` (default) | two-element row vector of nondecreasing values.

Range of time shift in seconds, specified as a two-element row vector of nondecreasing values.

#### **Dependencies**

To enable this property, set `AugmentationParameterSource` to `'random'`.

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

### **ApplyTimeShift — Apply time shift**

`true` (default) | `false`

Apply time shift, specified as `true` or `false`.

#### **Dependencies**

To enable this property, set `AugmentationParameterSource` to `'specify'`.

Time-shifting applies a circular shift on the time-domain audio data.

Data Types: `logical`

### **TimeShift — Time shift (s)**

`5e-3` (default) | `scalar` | `vector`

Time shift in seconds, specified as a scalar or vector.

#### **Dependencies**

To enable this property, set `AugmentationParameterSource` to `'specify'`.

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

## **Object Functions**

<code>addAugmentationMethod</code>	Add custom augmentation method
<code>removeAugmentationMethod</code>	Remove custom augmentation method
<code>augment</code>	Augment audio data

## Examples

### Apply Random Sequential Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
sound(audioIn,fs)
```

Create an `audioDataAugmenter` object that applies time stretching, volume control, and time shifting in cascade. Apply each of the augmentations with 80% probability. Set `NumAugmentations` to 5 to output five independently augmented signals. To skip pitch shifting and noise addition for each augmentation, set the respective probabilities to 0. Define parameter ranges for each relevant augmentation algorithm.

```
augmenter = audioDataAugmenter( ...
    "AugmentationMode","sequential", ...
    "NumAugmentations",5, ...
    ...
    "TimeStretchProbability",0.8, ...
    "SpeedupFactorRange", [1.3,1.4], ...
    ...
    "PitchShiftProbability",0, ...
    ...
    "VolumeControlProbability",0.8, ...
    "VolumeGainRange", [-5,5], ...
    ...
    "AddNoiseProbability",0, ...
    ...
    "TimeShiftProbability",0.8, ...
    "TimeShiftRange", [-500e-3,500e-3])
```

augmenter =

audioDataAugmenter with properties:

```
    AugmentationMode: "sequential"
    AugmentationParameterSource: 'random'
    NumAugmentations: 5
    TimeStretchProbability: 0.8000
    SpeedupFactorRange: [1.3000 1.4000]
    PitchShiftProbability: 0
    VolumeControlProbability: 0.8000
    VolumeGainRange: [-5 5]
```

```
AddNoiseProbability: 0
TimeShiftProbability: 0.8000
TimeShiftRange: [-0.5000 0.5000]
```

Call `augment` on the audio to create 5 augmentations. The augmented audio is returned in a table with variables `Audio` and `AugmentationInfo`. The number of rows in the table is defined by `NumAugmentations`.

```
data = augment(augmenter, audioIn, fs)
```

```
data=5x2 table
      Audio          AugmentationInfo
      -----          -----
      {685056x1 double}  [1x1 struct]
      {685056x1 double}  [1x1 struct]
      {505183x1 double}  [1x1 struct]
      {685056x1 double}  [1x1 struct]
      {490728x1 double}  [1x1 struct]
```

In the current augmentation pipeline, augmentation parameters are assigned randomly from within the specified ranges. To determine the exact parameters used for an augmentation, inspect `AugmentationInfo`.

```
augmentationToInspect = ;
data.AugmentationInfo(augmentationToInspect)
```

```
ans = struct with fields:
  SpeedupFactor: 1
  VolumeGain: 4.3399
  TimeShift: 0.4502
```

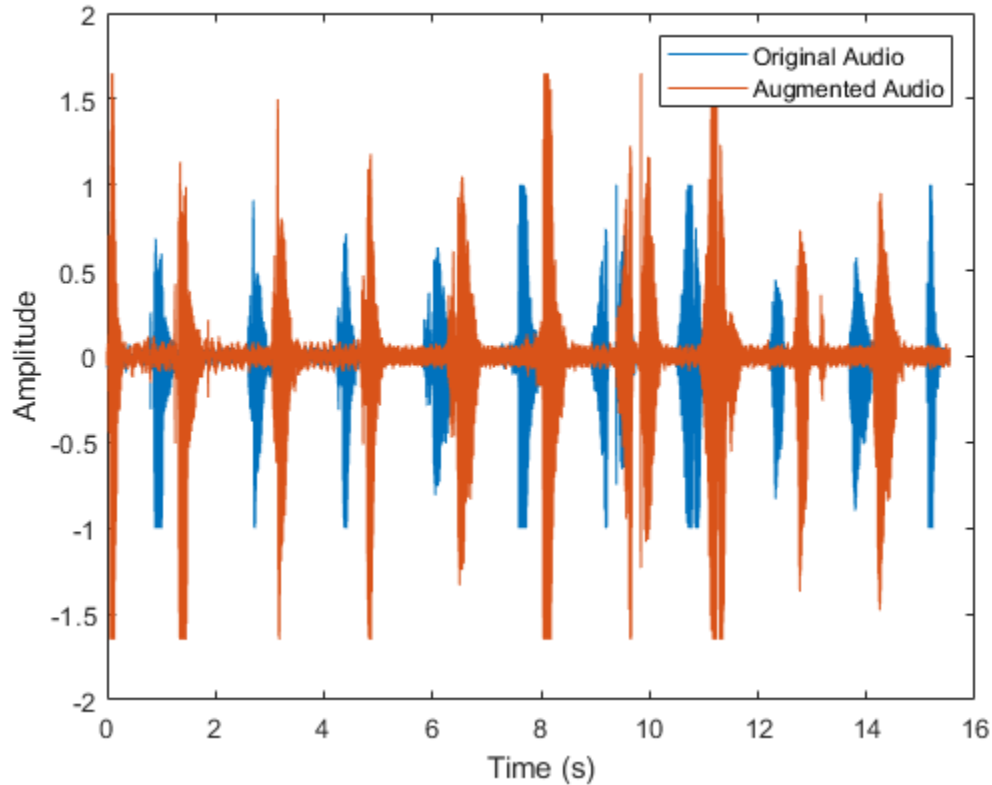
Listen to the augmentation you are inspecting. Plot time representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation, fs)
```

```
t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t, audioIn, taug, augmentation)
legend("Original Audio", "Augmented Audio")
```



```
ylabel("Amplitude")  
xlabel("Time (s)")
```



### Apply Specified Sequential Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");  
sound(audioIn,fs)
```

Create an `audioDataAugmenter` object that applies time stretching, pitch shifting, and noise corruption in cascade. Specify the time stretch speedup factors as 0.9, 1.1, and

1.2. Specify the pitch shifting in semitones as -2, -1, 1, and 2. Specify the noise corruption SNR as 10 dB and 15 dB.

```
augmenter = audioDataAugmenter( ...  
    "AugmentationMode","sequential", ...  
    "AugmentationParameterSource","specify", ...  
    "SpeedupFactor",[0.9,1.1,1.2], ...  
    "ApplyTimeStretch",true, ...  
    "ApplyPitchShift",true, ...  
    "SemitoneShift",[-2,-1,1,2], ...  
    "SNR",[10,15], ...  
    "ApplyVolumeControl",false, ...  
    "ApplyTimeShift",false)
```

```
augmenter =  
    audioDataAugmenter with properties:
```

```
        AugmentationMode: "sequential"  
    AugmentationParameterSource: "specify"  
        ApplyTimeStretch: 1  
        SpeedupFactor: [0.9000 1.1000 1.2000]  
        ApplyPitchShift: 1  
        SemitoneShift: [-2 -1 1 2]  
    ApplyVolumeControl: 0  
        ApplyAddNoise: 1  
                SNR: [10 15]  
    ApplyTimeShift: 0
```

Call `augment` on the audio to create 24 augmentations. The augmentations represent every combination of the specified augmentation parameters ( $3 \times 4 \times 2 = 24$ ).

```
data = augment(augmenter, audioIn, fs)
```

```
data=24x2 table  
        Audio          AugmentationInfo  
-----  
{761243x1 double}    [1x1 struct]  
{622888x1 double}    [1x1 struct]  
{571263x1 double}    [1x1 struct]  
{761243x1 double}    [1x1 struct]  
{622888x1 double}    [1x1 struct]  
{571263x1 double}    [1x1 struct]  
{761243x1 double}    [1x1 struct]
```

```

{622888x1 double}    [1x1 struct]
{571263x1 double}    [1x1 struct]
{761243x1 double}    [1x1 struct]
{622888x1 double}    [1x1 struct]
{571263x1 double}    [1x1 struct]
{761243x1 double}    [1x1 struct]
{622888x1 double}    [1x1 struct]
{571263x1 double}    [1x1 struct]
{761243x1 double}    [1x1 struct]
:

```

You can check the parameter configuration of each augmentation using the `AugmentationInfo` table variable.

```

augmentationToInspect = ;
data.AugmentationInfo(augmentationToInspect)

```

```

ans = struct with fields:
    SpeedupFactor: 0.9000
    SemitoneShift: -2
    SNR: 10

```

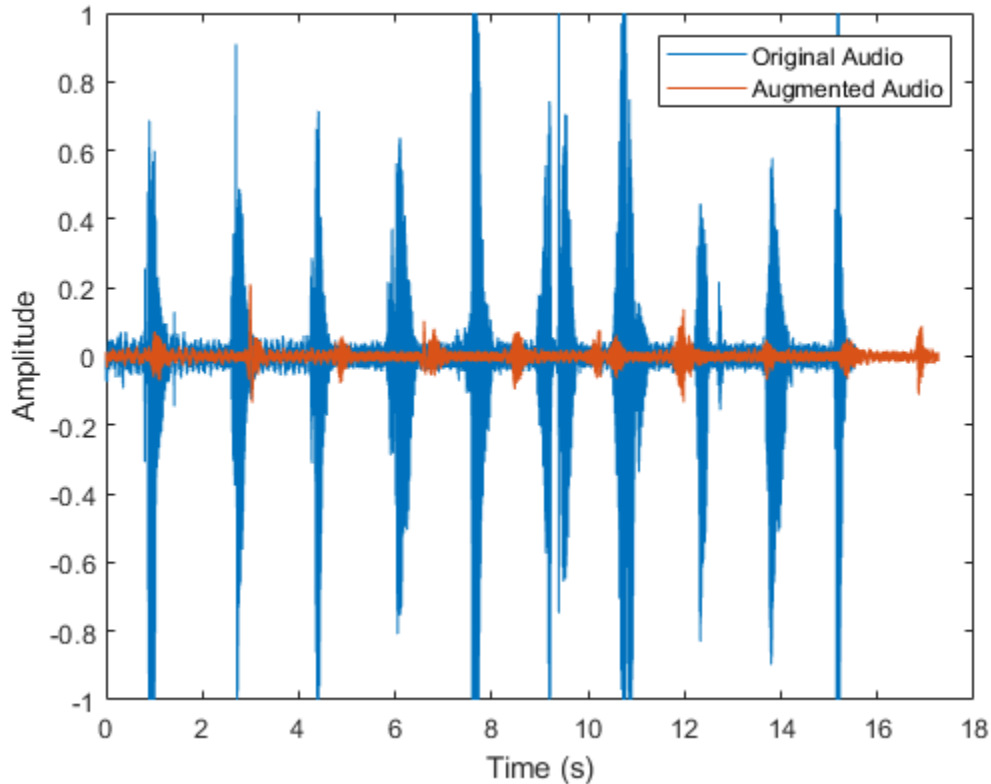
Listen to the augmentation you are inspecting. Plot the time-domain representation of the original and augmented signals.

```

augmentation = data.Audio{augmentationToInspect};
sound(augmentation, fs)

t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t, audioIn, taug, augmentation)
legend("Original Audio", "Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")

```



### Apply Random Independent Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioDataAugmenter` object that applies noise corruption, and time shifting in parallel branches. For the noise corruption branch, randomly apply noise with an SNR in the range 0 dB to 20 dB. For the time shifting branch, randomly apply time shifting in the range -300 ms to 300 ms. Apply augmentation 2 times for each branch, for 4 total augmentations.

```

augmenter = audioDataAugmenter( ...
    "AugmentationMode", "independent", ...
    "AugmentationParameterSource", "random", ...
    "NumAugmentations", 2, ...
    "ApplyTimeStretch", false, ...
    "ApplyPitchShift", false, ...
    "ApplyVolumeControl", false, ...
    "SNRRange", [0, 20], ...
    "TimeShiftRange", [-300e-3, 300e-3])

```

```

augmenter =
  audioDataAugmenter with properties:

      AugmentationMode: "independent"
  AugmentationParameterSource: "random"
      NumAugmentations: 2
      ApplyTimeStretch: 0
      ApplyPitchShift: 0
  ApplyVolumeControl: 0
      ApplyAddNoise: 1
          SNRRange: [0 20]
      ApplyTimeShift: 1
      TimeShiftRange: [-0.3000 0.3000]

```

Call `augment` on the audio to create 3 augmentations.

```
data = augment(augmenter, audioIn, fs);
```

You can check the parameter configuration of each augmentation using the `AugmentationInfo` table variable.

```

augmentationToInspect = ;
data.AugmentationInfo{augmentationToInspect}

```

```

ans = struct with fields:
  TimeShift: 0.0016

```

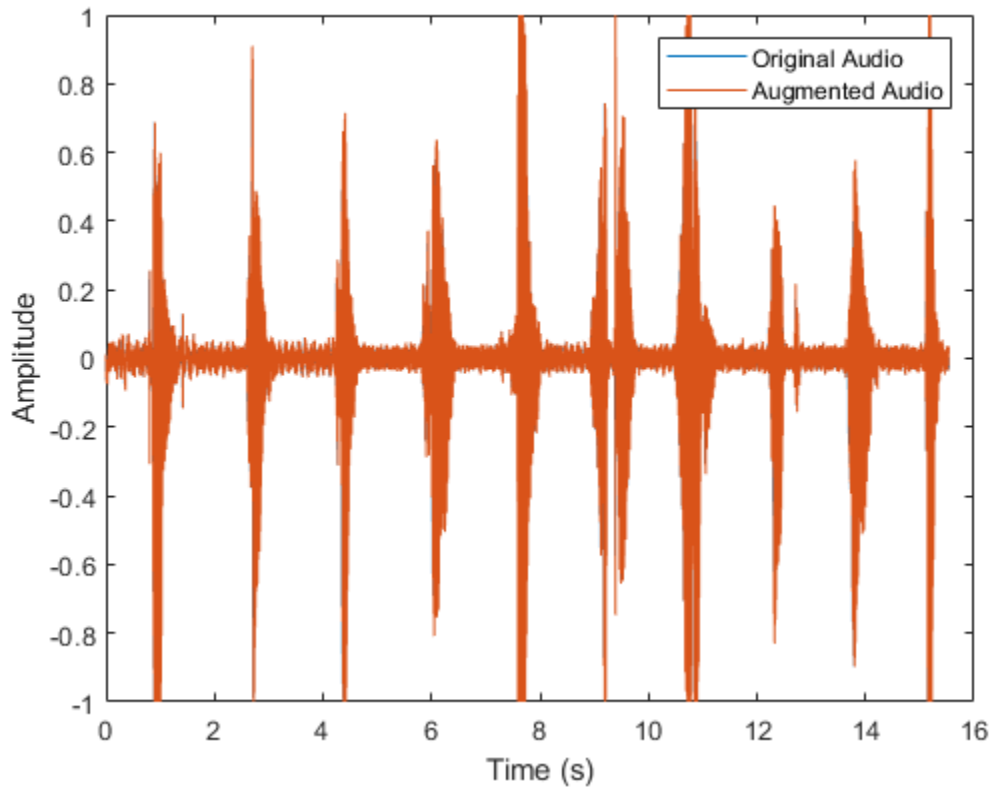
Listen to the audio you are inspecting. Plot the time-domain representation of the original and augmented signals.

```

augmentation = data.Audio{augmentationToInspect};
sound(augmentation, fs)

```

```
t = (0:(numel(audioIn)-1))/fs;  
taug = (0:(numel(augmentation)-1))/fs;  
plot(t,audioIn,taug,augmentation)  
legend("Original Audio","Augmented Audio")  
ylabel("Amplitude")  
xlabel("Time (s)")
```



### Apply Specified Independent Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioDataAugmenter` object that applies volume control, noise corruption, and time shifting in parallel branches.

```
augmenter = audioDataAugmenter( ...
    "AugmentationMode","independent", ...
    "AugmentationParameterSource","specify", ...
    "ApplyTimeStretch",false, ...
    "ApplyPitchShift",false, ...
    "VolumeGain",2, ...
    "SNR",0, ...
    "TimeShift",2)
```

```
augmenter =
    audioDataAugmenter with properties:
```

```
    AugmentationMode: "independent"
    AugmentationParameterSource: "specify"
    ApplyTimeStretch: 0
    ApplyPitchShift: 0
    ApplyVolumeControl: 1
    VolumeGain: 2
    ApplyAddNoise: 1
    SNR: 0
    ApplyTimeShift: 1
    TimeShift: 2
```

Call `augment` on the audio to create 3 augmentations.

```
data = augment(augmenter,audioIn,fs)
```

```
data=3x2 table
    Audio      AugmentationInfo
    _____  _____
    {685056x1 double}  {1x1 struct}
    {685056x1 double}  {1x1 struct}
    {685056x1 double}  {1x1 struct}
```

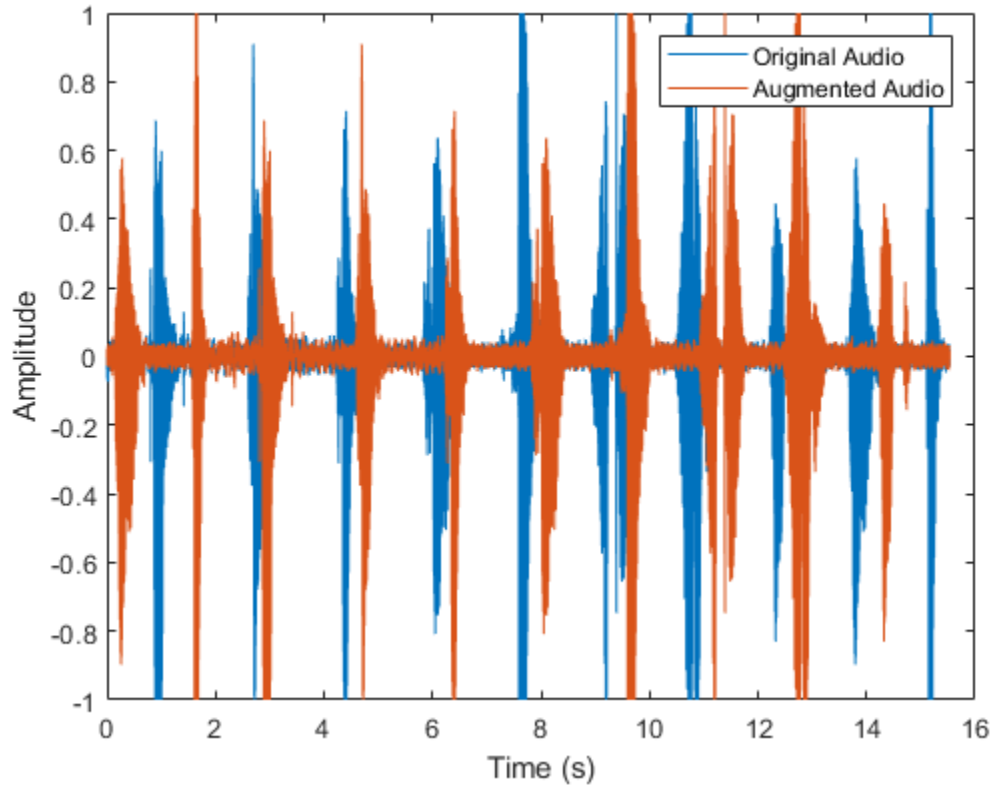
You can check the parameter configuration of each augmentation using the `AugmentationInfo` table variable.

```
augmentationToInspect = ;  
data.AugmentationInfo{augmentationToInspect}  
  
ans = struct with fields:  
    TimeShift: 2
```

Listen to the audio you are inspecting. Plot the time-domain representations of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};  
sound(augmentation, fs)  
  
t = (0:(numel(audioIn)-1))/fs;  
taug = (0:(numel(augmentation)-1))/fs;  
plot(t, audioIn, taug, augmentation)  
legend("Original Audio", "Augmented Audio")  
ylabel("Amplitude")  
xlabel("Time (s)")
```





### Augment Audio Dataset

The `audioDataAugmenter` supports multiple workflows for augmenting your datastore, including:

- Offline augmentation
- Augmentation using tall arrays
- Augmentation using transform datastores

In each workflow, begin by creating an audio datastore to point to your audio data. In this example, you create an audio datastore that points to audio samples included with Audio Toolbox™. Count the number of files in the dataset.

```
folder = fullfile(matlabroot, "toolbox", "audio", "samples");
ADS = audioDatastore(folder)
```

```
ADS =
  audioDatastore with properties:

    Files: {
        ' ...\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono
        ' ...\matlab\toolbox\audio\samples\AudioArray-16-16-4char
        ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p
        ... and 26 more
    }
  AlternateFileSystemRoots: {}
  OutputDataType: 'double'
  Labels: {}
```

```
numFilesInDataset = numel(ADS.Files)
```

```
numFilesInDataset = 29
```

Create an `audioDataAugmenter` that applies random sequential augmentations. Set `NumAugmentations` to 2.

```
aug = audioDataAugmenter('NumAugmentations',2)
```

```
aug =
  audioDataAugmenter with properties:

    AugmentationMode: 'sequential'
  AugmentationParameterSource: 'random'
    NumAugmentations: 2
  TimeStretchProbability: 0.5000
  SpeedupFactorRange: [0.8000 1.2000]
  PitchShiftProbability: 0.5000
  SemitoneShiftRange: [-2 2]
  VolumeControlProbability: 0.5000
  VolumeGainRange: [-3 3]
  AddNoiseProbability: 0.5000
  SNRRange: [0 10]
  TimeShiftProbability: 0.5000
```

```
TimeShiftRange: [-0.0050 0.0050]
```

## Offline Augmentation

To augment the audio dataset, create two augmentations of each file and then write the augmentations as WAV files.

```
while hasdata(ADS)
    [audioIn,info] = read(ADS);

    data = augment(aug,audioIn,info.SampleRate);

    [~,fn] = fileparts(info.FileName);
    for i = 1:size(data,1)
        augmentedAudio = data.Audio{i};

        % If augmentation caused an audio signal to have values outside of -1 and 1,
        % normalize the audio signal to avoid clipping when writing.
        if max(abs(augmentedAudio),[],'all')>1
            augmentedAudio = augmentedAudio/max(abs(augmentedAudio),[],'all');
        end

        audiowrite(sprintf('%s_aug%d.wav',fn,i),augmentedAudio,info.SampleRate)
    end
end
```

Create an `audioDatastore` that points to the augmented dataset and confirm that the number of files in the dataset is double the original number of files.

```
augmentedADS = audioDatastore(pwd)

augmentedADS =
    audioDatastore with properties:

        Files: {
            ' ...\Examples\audio-ex28074079\Ambiance-16-44p1-mono-12
            ' ...\Examples\audio-ex28074079\Ambiance-16-44p1-mono-12
            ' ...\Examples\audio-ex28074079\AudioArray-16-16-4channel
            ... and 55 more
        }
        AlternateFileSystemRoots: {}
        OutputDataType: 'double'
        Labels: {}
```

```
numFilesInAugmentedDataset = numel(augmentedADS.Files)

numFilesInAugmentedDataset = 58
```

### Augment Using Tall Arrays

When augmenting a dataset using tall arrays, the input data to the augmentser should be sampled at a consistent rate. Subset the original audio dataset to only include files with a sample rate of 44.1 kHz. Most datasets are already cleaned to have a consistent sample rate.

```
keepFile = cellfun(@(x)contains(x,'44p1'),ADS.Files);
ads44p1 = subset(ADS,keepFile);
fs = 44.1e3;
```

Convert the audio datastore to a tall array. Tall arrays are evaluated only when you request them explicitly using `gather`. MATLAB® automatically optimizes the queued calculations by minimizing the number of passes through the data. If you have the Parallel Computing Toolbox™, you can spread the calculations across multiple machines. The audio data is represented as an  $M$ -by-1 tall cell array, where  $M$  is the number of files in the audio datastore.

```
adsTall = tall(ads44p1)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
adsTall =
```

```
M×1 tall cell array

{ 539648×1 double}
{ 227497×1 double}
{   8000×1 double}
{ 685056×1 double}
{ 882688×2 double}
{1115760×2 double}
{ 505200×2 double}
{3195904×2 double}
      :
      :
```

Define a `cellfun` function so that augmentation is applied to each cell of the tall array. Call `gather` to evaluate the tall array.

```
augTall = cellfun(@(x)augment(aug,x,fs),adsTall,"UniformOutput",false);
augmentedDataset = gather(augTall)
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1 min 34 sec
Evaluation completed in 1 min 34 sec
```

```
augmentedDataset=12x1 cell
    {2x2 table}
    {2x2 table}
    {2x2 table}
    {2x2 table}
    {2x2 table}
    {2x2 table}
    {2x2 table}
    {2x2 table}
    {2x2 table}
    {2x2 table}
    {2x2 table}
    {2x2 table}
```

The augmented dataset is returned as a *numFiles*-by-1 cell array, where *numFiles* is the number of files in the datastore. Each element of the cell array is a *numAugmentationsPerFile*-by-2 table, where *numAugmentationsPerFile* is the number of augmentations returned per file.

```
numFiles = numel(augmentedDataset)
```

```
numFiles = 12
```

```
numAugmentationsPerFile = size(augmentedDataset{1},1)
```

```
numAugmentationsPerFile = 2
```

### Augment Using Transform Datastore

You can perform online data augmentation while you train your machine learning application using a transform datastore. Call `transform` to create a new datastore that applies data augmentation while reading.

```
transformADS = transform(ADS,@(x,info)augment(aug,x,info),'IncludeInfo',true)
```

```
transformADS =
    TransformedDatastore with properties:
```

```
UnderlyingDatastore: [1x1 audioDatastore]
  Transforms: {@(x,info)augment(aug,x,info)}
  IncludeInfo: 1
```

Call `read` to return the augmented first file from the transform datastore.

```
augmentedRead = read(transformADS)
```

```
augmentedRead=2x2 table
      Audio      AugmentationInfo
      -----      -----
      {539648x1 double}      [1x1 struct]
      {586683x1 double}      [1x1 struct]
```

### Add Custom Augmentation Method

You can expand the capabilities of `audioDataAugmenter` by adding custom augmentation methods.

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
sound(audioIn,fs)
```

Create an `audioDataAugmenter` object. Set the probability of applying white noise to 0.

```
augmenter = audioDataAugmenter('AddNoiseProbability',0)
```

```
augmenter =
  audioDataAugmenter with properties:

      AugmentationMode: 'sequential'
  AugmentationParameterSource: 'random'
      NumAugmentations: 1
  TimeStretchProbability: 0.5000
  SpeedupFactorRange: [0.8000 1.2000]
  PitchShiftProbability: 0.5000
  SemitoneShiftRange: [-2 2]
  VolumeControlProbability: 0.5000
```

```

        VolumeGainRange: [-3 3]
    AddNoiseProbability: 0
    TimeShiftProbability: 0.5000
        TimeShiftRange: [-0.0050 0.0050]

```

Specify a custom augmentation algorithm that applies pink noise. The `AddPinkNoise` algorithm is added to the `augmenter` properties.

```

algorithmName = 'AddPinkNoise';
algorithmHandle = @(x)x+pinknoise(size(x),'like',x);
addAugmentationMethod(augmenter,algorithmName,algorithmHandle)

```

`augmenter`

```

augmenter =
    audioDataAugmenter with properties:

        AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
        NumAugmentations: 1
    TimeStretchProbability: 0.5000
        SpeedupFactorRange: [0.8000 1.2000]
    PitchShiftProbability: 0.5000
        SemitoneShiftRange: [-2 2]
    VolumeControlProbability: 0.5000
        VolumeGainRange: [-3 3]
    AddNoiseProbability: 0
    TimeShiftProbability: 0.5000
        TimeShiftRange: [-0.0050 0.0050]
    AddPinkNoiseProbability: 0.5000

```

Set the probability of adding pink noise to 1.

```
augmenter.AddPinkNoiseProbability = 1
```

```

augmenter =
    audioDataAugmenter with properties:

        AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
        NumAugmentations: 1
    TimeStretchProbability: 0.5000
        SpeedupFactorRange: [0.8000 1.2000]

```

```
PitchShiftProbability: 0.5000
  SemitoneShiftRange: [-2 2]
VolumeControlProbability: 0.5000
  VolumeGainRange: [-3 3]
  AddNoiseProbability: 0
  TimeShiftProbability: 0.5000
    TimeShiftRange: [-0.0050 0.0050]
AddPinkNoiseProbability: 1
```

Augment the original signal and listen to the result. Inspect parameters of the augmentation algorithms applied.

```
data = augment(augmenter, audioIn, fs);
sound(data.Audio{1}, fs)
```

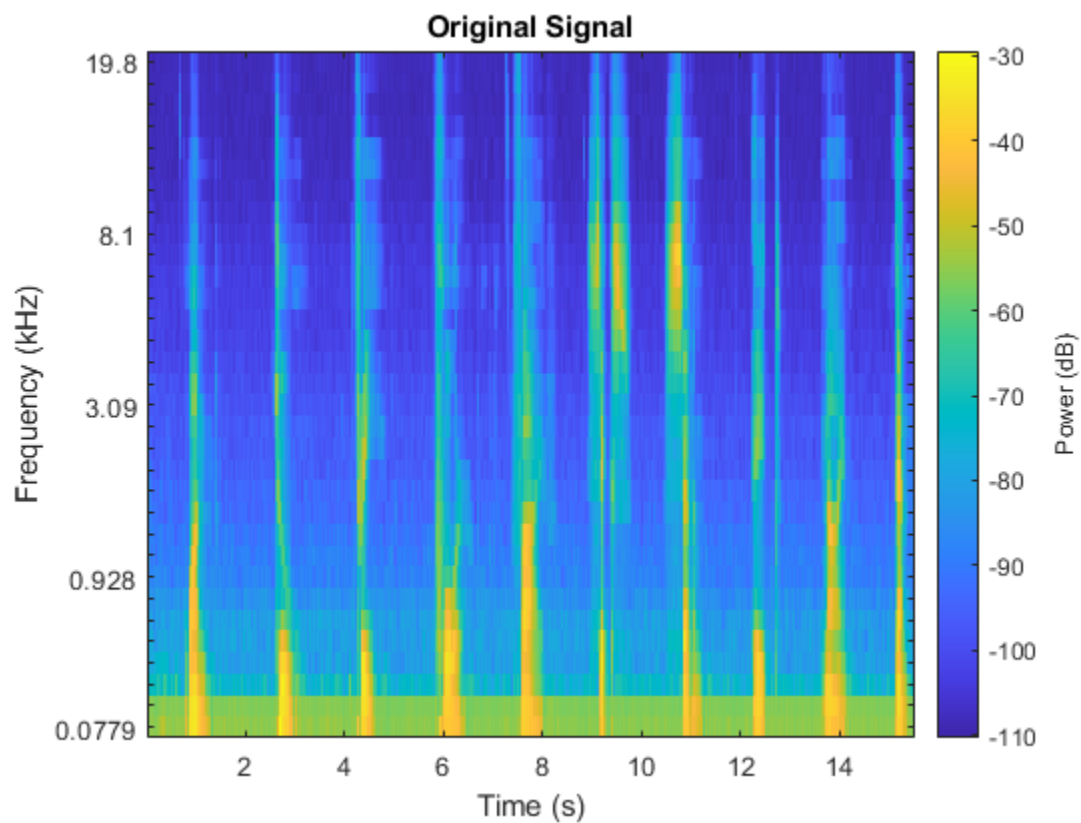
```
data.AugmentationInfo(1)
```

```
ans = struct with fields:
  SpeedupFactor: 1
  SemitoneShift: 0
  VolumeGain: 2.4803
  TimeShift: -0.0022
  AddPinkNoise: 'Applied'
```

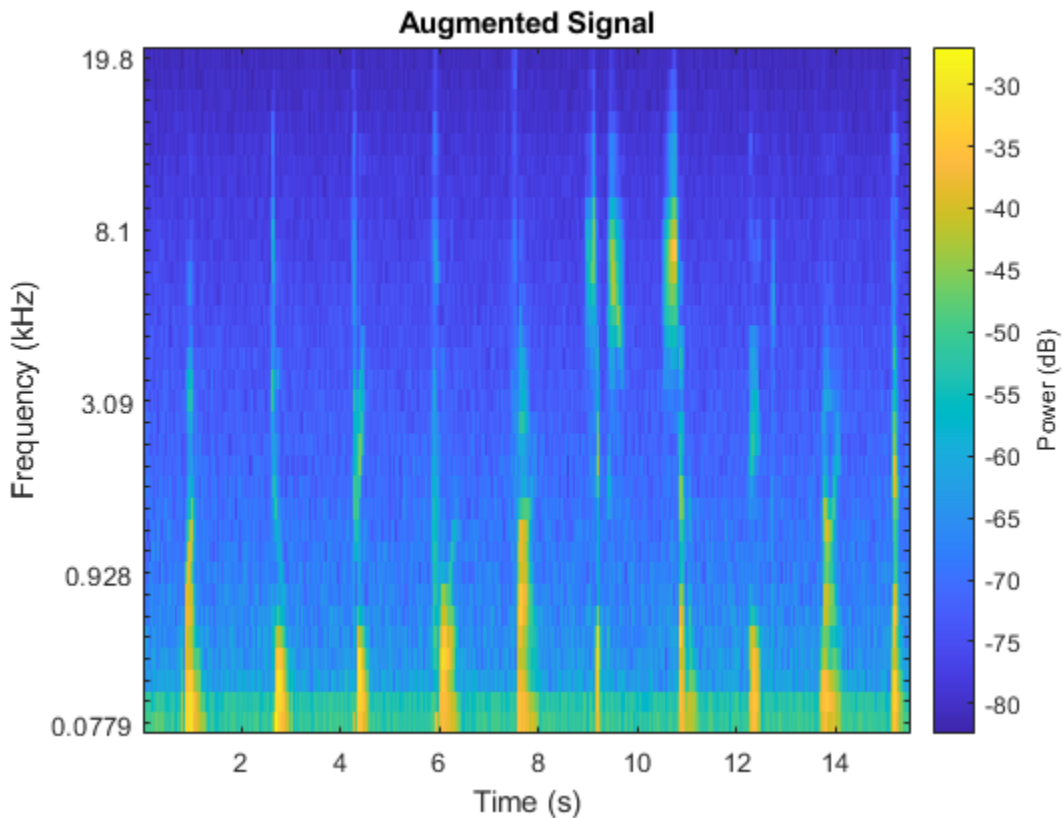
Plot the mel spectrograms of the original and augmented signals.

```
melSpectrogram(audioIn, fs)
title('Original Signal')
```





```
melSpectrogram(data.Audio{1}, fs)  
title('Augmented Signal')
```



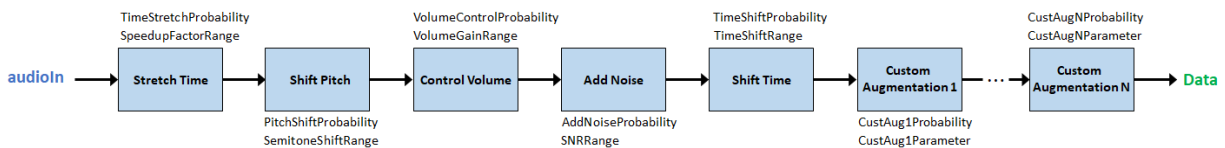
## Algorithms

The `audioDataAugmenter` object enables you to configure your augmentation pipeline as deterministic or probabilistic using the `AugmentationParameterSource` property. You can also choose to apply the augmentations in series or in parallel using the `AugmentationMode` property. The following sections describe the pipelines you can create and the applicable properties for each architecture.

## Random Sequential Augmentations

To define your augmentation as a sequence of probabilistically applied augmentations, set `AugmentationParameterSource` to `'random'` and `AugmentationMode` to `'sequential'`.

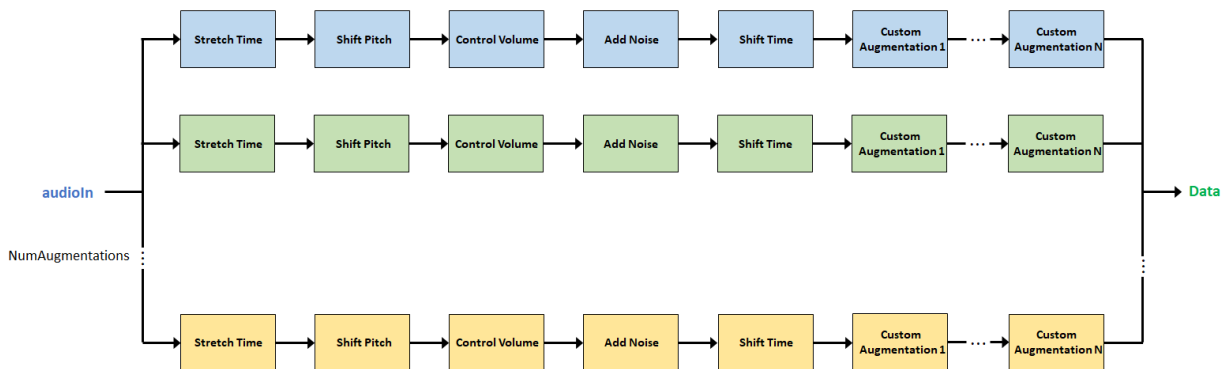
The order that augmentations are applied is always the same. If you specify custom algorithms, they are applied at the end of the sequence, in the order you specified them.



In this pipeline configuration, these parameters apply:

Augmentation Method	Parameters
Stretch Time	TimeStretchProbability SpeedupFactorRange
Shift Pitch	PitchShiftProbability SemitoneShiftRange
Control Volume	VolumeControlProbability VolumeGainRange
Add Noise	AddNoiseProbability SNRRange
Shift Time	TimeShiftProbability TimeShiftRange

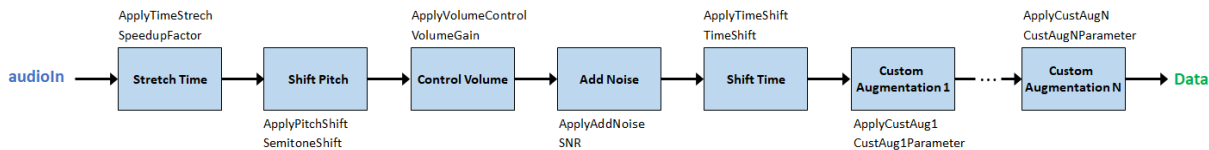
If you specify `NumAugmentations` as greater than 1, then the object applies `NumAugmentations` parallel random sequential augmentations. The probability of applying an augmentation, and the value of any parameters that are probabilistically determined, are independent.



## Specified Sequential Augmentations

To define your augmentation as a sequence of deterministically applied augmentations, set `AugmentationParameterSource` to 'specify' and `AugmentationMode` to 'sequential'.

The order that augmentations are applied is always the same. If you specify custom algorithms, they are applied at the end of the sequence, in the order you specified them.



In this pipeline configuration, these parameters apply:

Augmentation Method	Parameters
Stretch Time	ApplyTimeStretch SpeedupFactor

Augmentation Method	Parameters
Shift Pitch	ApplyPitchShift SemitoneShift
Control Volume	ApplyVolumeControl VolumeGain
Add Noise	ApplyAddNoise SNR
Shift Time	ApplyTimeShift TimeShift

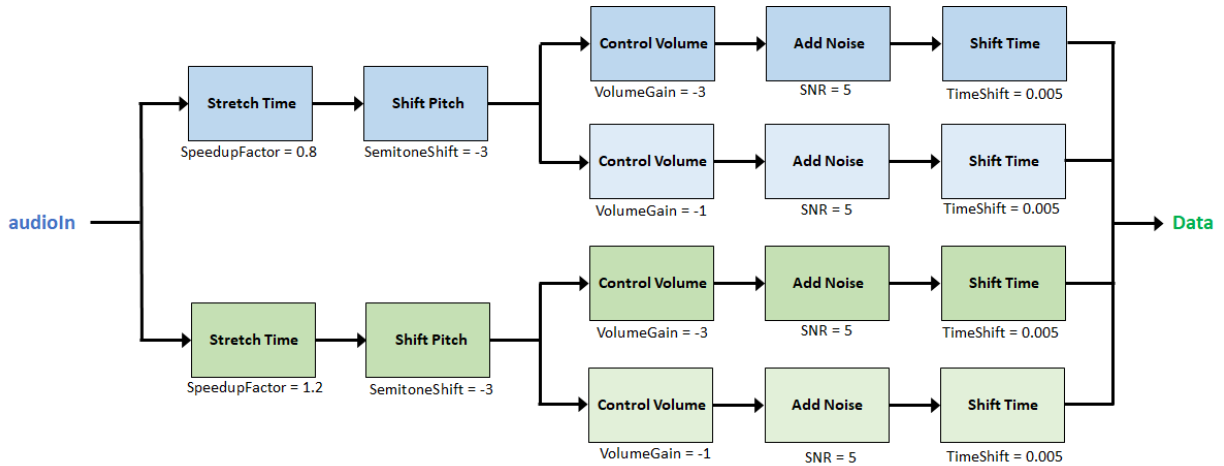
If you specify an augmentation method as a vector, then each element of the vector creates a separate branch in the augmentation pipeline. For example, the following object creates an augmentation pipeline that results in four separate augmentations:

```
aug = audioDataAugmenter("AugmentationMode","sequential", ...
    "AugmentationParameterSource","specify", ...
    "SpeedupFactor",[0.8,1.2], ...
    "VolumeGain",[-3,-1])
```

```
aug =
```

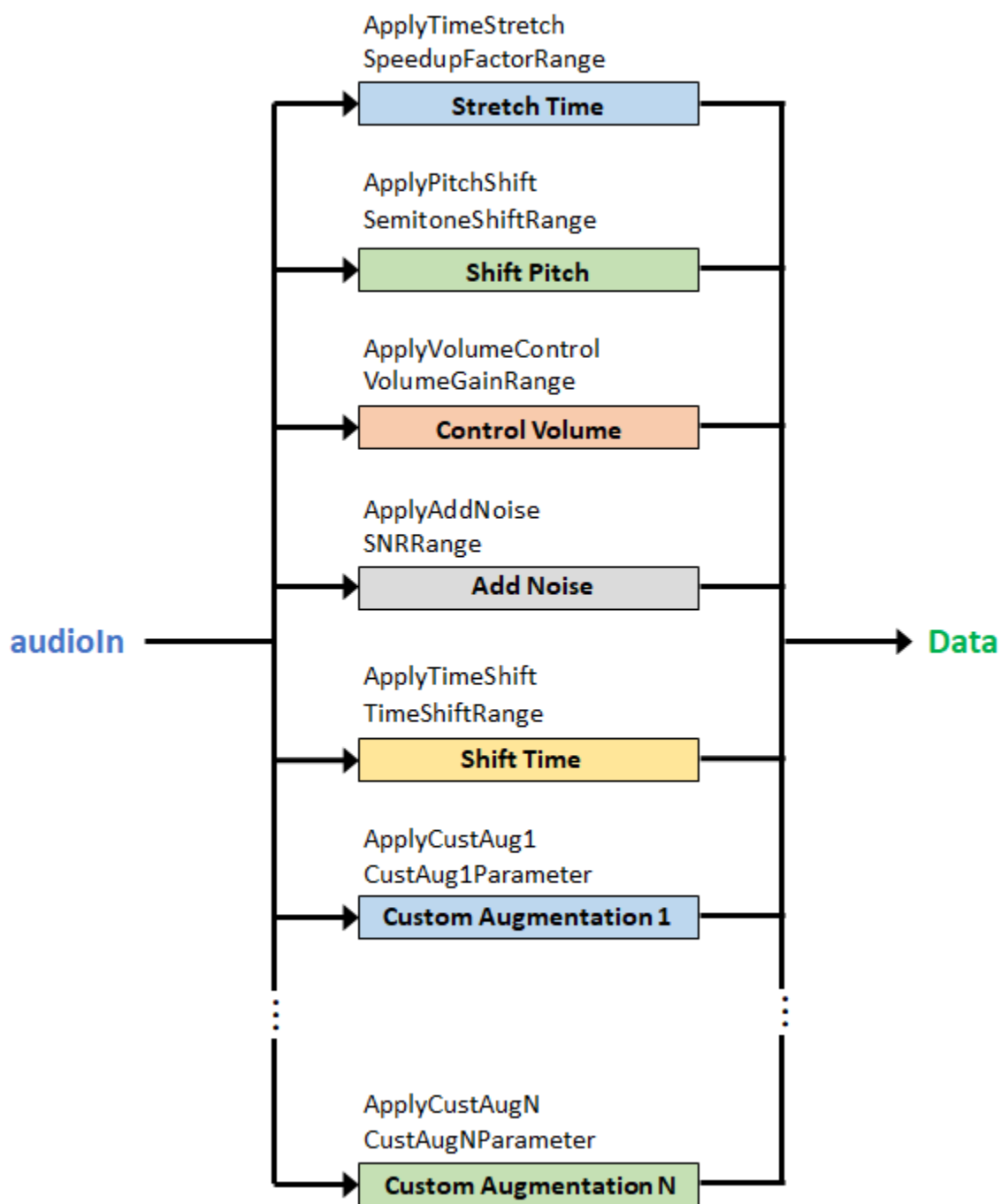
```
audioDataAugmenter with properties:
```

```
    AugmentationMode: "sequential"
    AugmentationParameterSource: "specify"
    ApplyTimeStretch: 1
    SpeedupFactor: [0.8000 1.2000]
    ApplyPitchShift: 1
    SemitoneShift: -3
    ApplyVolumeControl: 1
    VolumeGain: [-3 -1]
    ApplyAddNoise: 1
    SNR: 5
    ApplyTimeShift: 1
    TimeShift: 0.0050
```



## Random Independent Augmentations

To define your augmentation as independently applied augmentations with randomly determined parameters, set `AugmentationParameterSource` to `'random'` and `AugmentationMode` to `'independent'`.

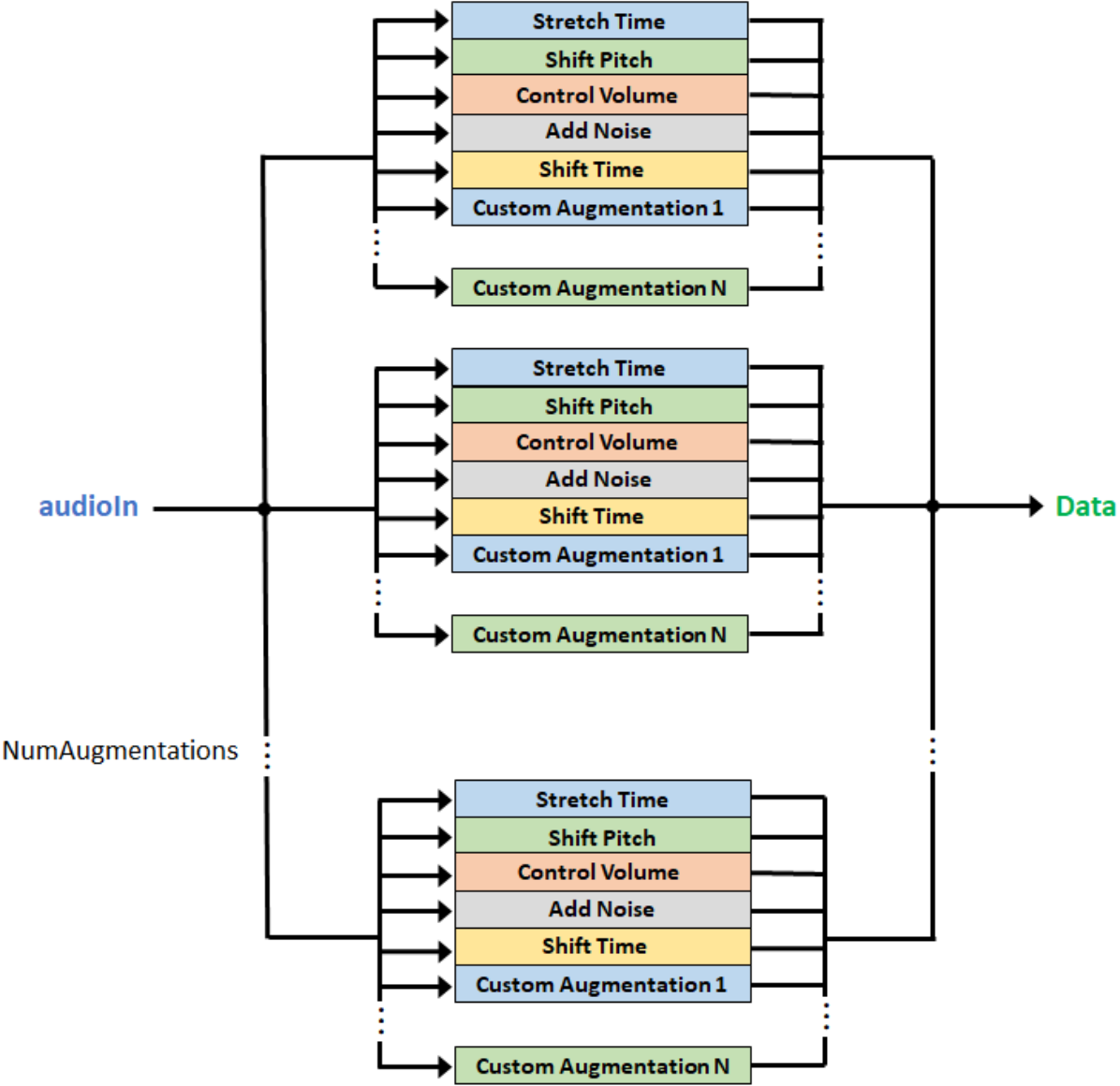


In this pipeline configuration, these parameters apply:

<b>Augmentation Method</b>	<b>Parameters</b>
Stretch Time	ApplyTimeStretch SpeedupFactorRange
Shift Pitch	ApplyPitchShift SemitoneShiftRange
Control Volume	ApplyVolumeControl VolumeGainRange
Add Noise	ApplyAddNoise SNRRange
Shift Time	ApplyTimeShift TimeShiftRange

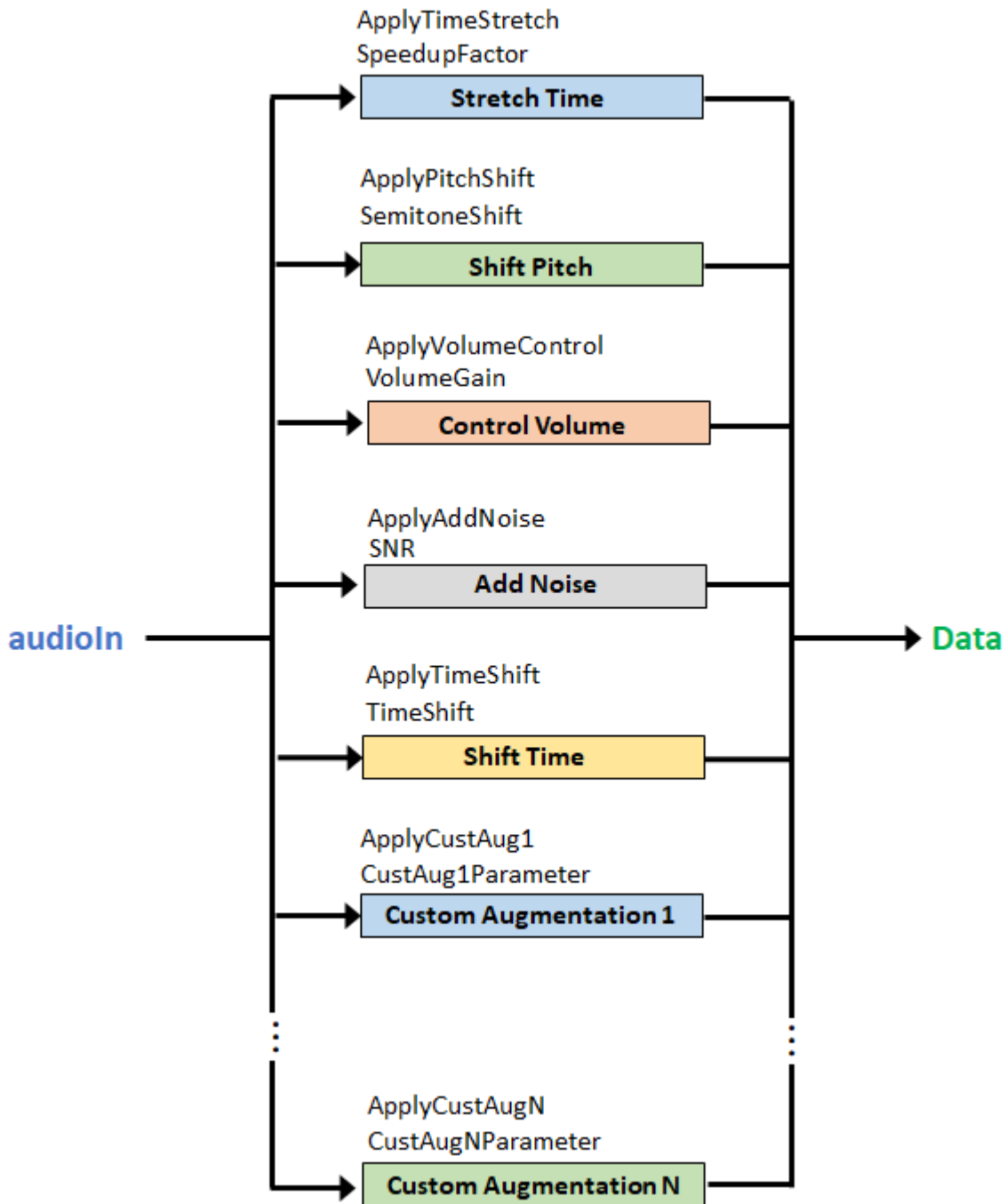
If you specify `NumAugmentations` as greater than 1, then the object applies `NumAugmentations` parallel random independent augmentations. The value of any parameters that are probabilistically determined are independent.





### **Specified Independent Augmentations**

To define your augmentation as deterministically applied independent augmentations with deterministic parameters, set `AugmentationParameterSource` to `'specify'` and `AugmentationMode` to `'independent'`.



In this pipeline configuration, these parameters apply:

Augmentation Method	Parameters
Stretch Time	ApplyTimeStretch SpeedupFactor
Shift Pitch	ApplyPitchShift SemitoneShift
Control Volume	ApplyVolumeControl VolumeGain
Add Noise	ApplyAddNoise SNR
Shift Time	ApplyTimeShift TimeShift

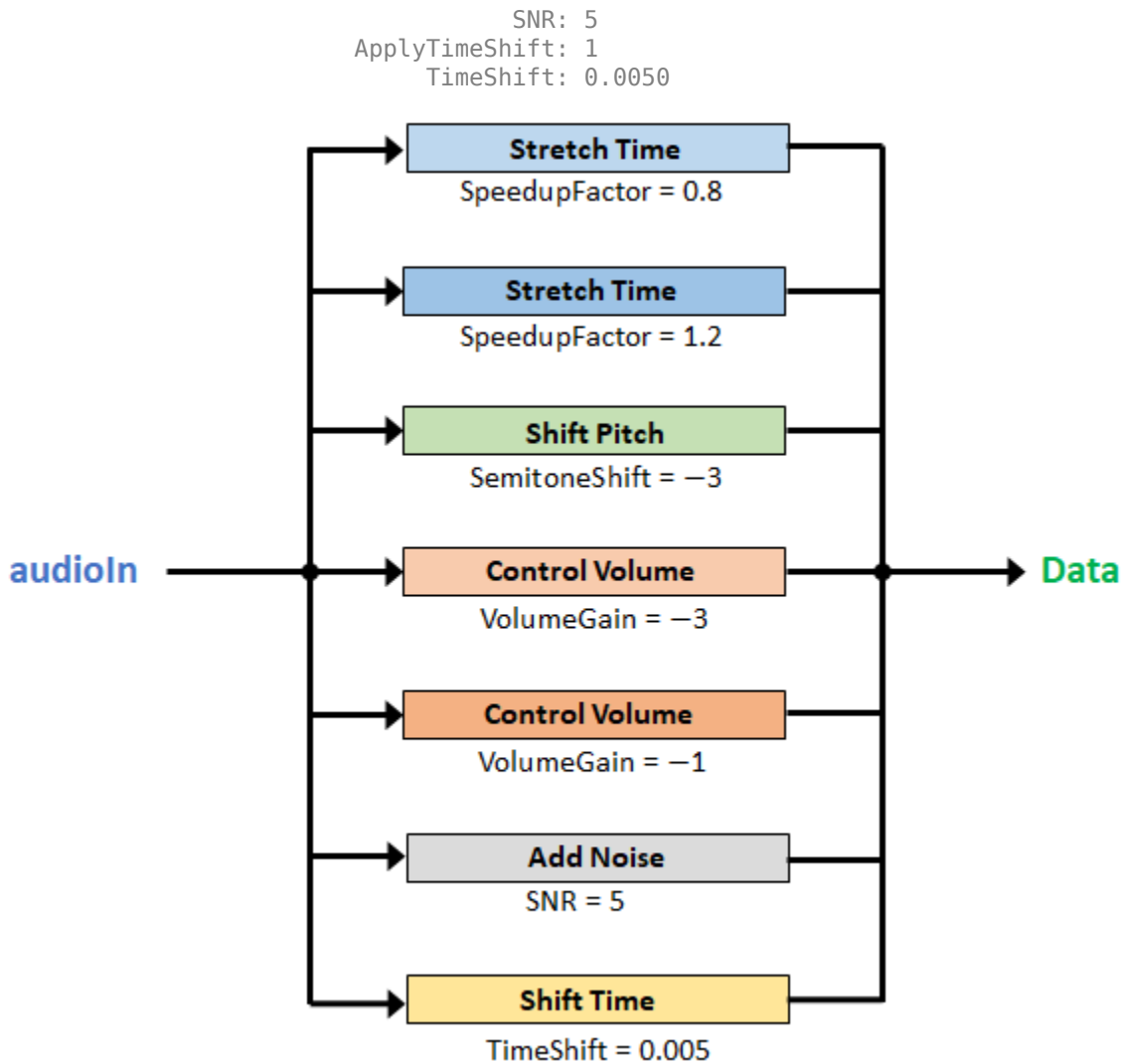
If you specify an augmentation method as a vector, then each element of the vector creates a separate branch in the augmentation pipeline. For example, the following object creates an augmentation pipeline that results in seven separate augmentations:

```
aug = audioDataAugmenter("AugmentationMode","independent", ...
    "AugmentationParameterSource","specify", ...
    "SpeedupFactor",[0.8,1.2], ...
    "VolumeGain",[-3,-1])
```

aug =

audioDataAugmenter with properties:

```
AugmentationMode: "independent"
AugmentationParameterSource: "specify"
ApplyTimeStretch: 1
SpeedupFactor: [0.8000 1.2000]
ApplyPitchShift: 1
SemitoneShift: -3
ApplyVolumeControl: 1
VolumeGain: [-3 -1]
ApplyAddNoise: 1
```



### References

- [1] Salamon, Justin, and Juan Pablo Bello. "Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification." *IEEE Signal Processing Letters*. Vol. 24, Issue 3, 2017.

### See Also

`audioFeatureExtractor` | `audioTimeScaler` | `shiftPitch` | `stretchAudio`

**Introduced in R2019b**

# transform

Transform audio datastore

## Syntax

```
transformDatastore = transform(ADS,@fcn)  
transformDatastore = transform(ADS,@fcn,Name,Value)
```

## Description

`transformDatastore = transform(ADS,@fcn)` creates a new datastore that transforms output from the read function.

`transformDatastore = transform(ADS,@fcn,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

## Examples

### Output Mono Audio from Datastore

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');  
ADS = audioDatastore(folder);
```

Call `transform` to create a new datastore that mixes multichannel signals to mono.

```
ADSnew = transform(ADS,@(x)mean(x,2));
```

Read from the new datastore and confirm that it only outputs mono signals.

```
while hasdata(ADSnew)  
    audio = read(ADSnew);
```





```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');
ADS = audioDatastore(folder);
```

Define a function to take as input the output of the `read` function. Make the function extract five seconds worth of data from the audio signal.

```
function [dataOut,info] = extractSegment(audioIn,info)
    [N,numChan] = size(audioIn);
    newN = round(info.SampleRate*5);
    if newN > N % signal length < 5 seconds
        numPad = newN - N + 1;
        dataOut = [audioIn;zeros(numPad,numChan,'like',audioIn)];
    elseif newN < N % signal length > 5 seconds
        start = randi(N - newN + 1);
        dataOut = audioIn(start:start+newN-1,:);
    else % signal length == 5 seconds
        dataOut = audioIn;
    end
end
```

Call `transform` to create a `TransformedDatastore` with `Transforms` set to the function you defined.

```
ADSnew = transform(ADS,@extractSegment,'IncludeInfo',true)
```

```
ADSnew =
    TransformedDatastore with properties:

        UnderlyingDatastore: [1x1 audioDatastore]
           Transforms: {@extractSegment}
        IncludeInfo: 1
```

Read the first three audio files and verify that the outputs are five second segments.

```
for i = 1:3
    [audio,info] = read(ADSnew);
    fprintf('Duration = %d seconds\n',size(audio,1)/info.SampleRate)
end
```

```
Duration = 5 seconds
Duration = 5 seconds
Duration = 5 seconds
```

### Output Mel Spectrogram

Use `transform` to create an audio datastore that returns a mel spectrogram representation from the `read` function.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');  
ADS = audioDatastore(folder);
```

Define a function that transforms audio data from a time-domain representation to a log mel spectrogram. The function adds the additional outputs from the `melSpectrogram` function to the `info` struct output from reading the audio datastore.

```
function [dataOut,infoOut] = extractMelSpectrogram(audioIn,info)  
  
    [S,F,T] = melSpectrogram(audioIn,info.SampleRate);  
  
    dataOut = 10*log10(S+eps);  
    infoOut = info;  
    infoOut.CenterFrequencies = F;  
    infoOut.TimeInstants = T;  
end
```

Call `transform` to create a `TransformedDatastore` with `Transforms` set to `extractMelSpectrogram`.

```
ADSnew = transform(ADS,@extractMelSpectrogram,'IncludeInfo',true)
```

```
ADSnew =
```

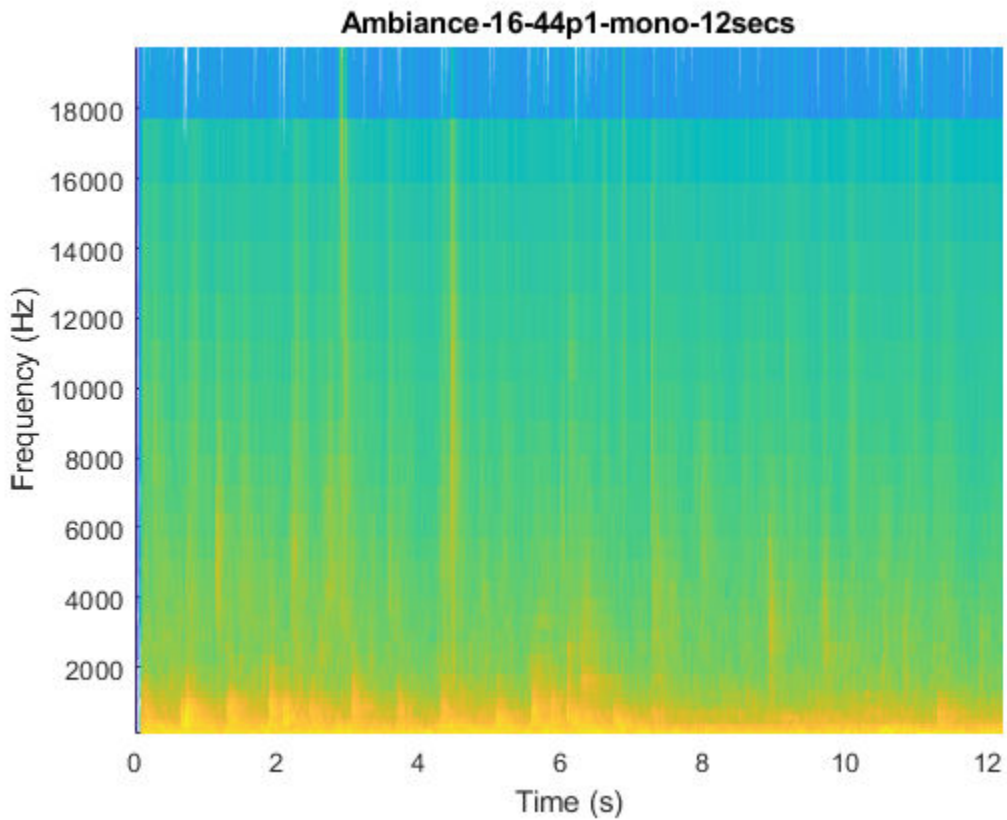
```
TransformedDatastore with properties:
```

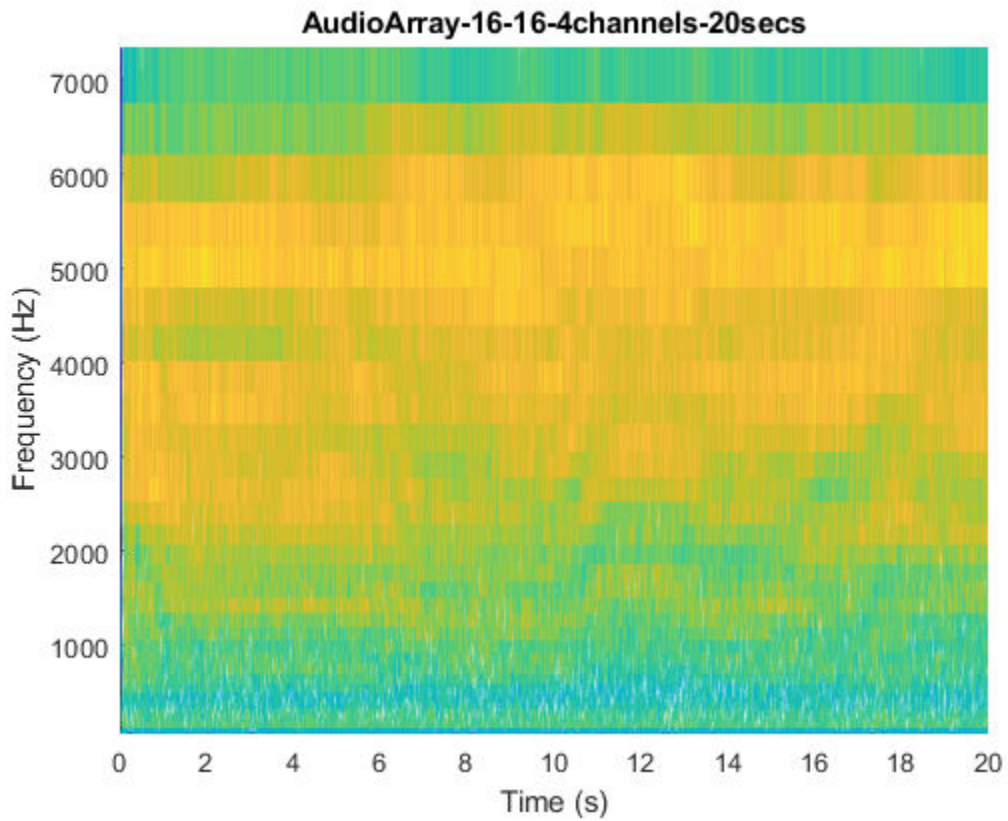
```
UnderlyingDatastore: [1x1 audioDatastore]  
Transforms: {@extractMelSpectrogram}  
IncludeInfo: 1
```

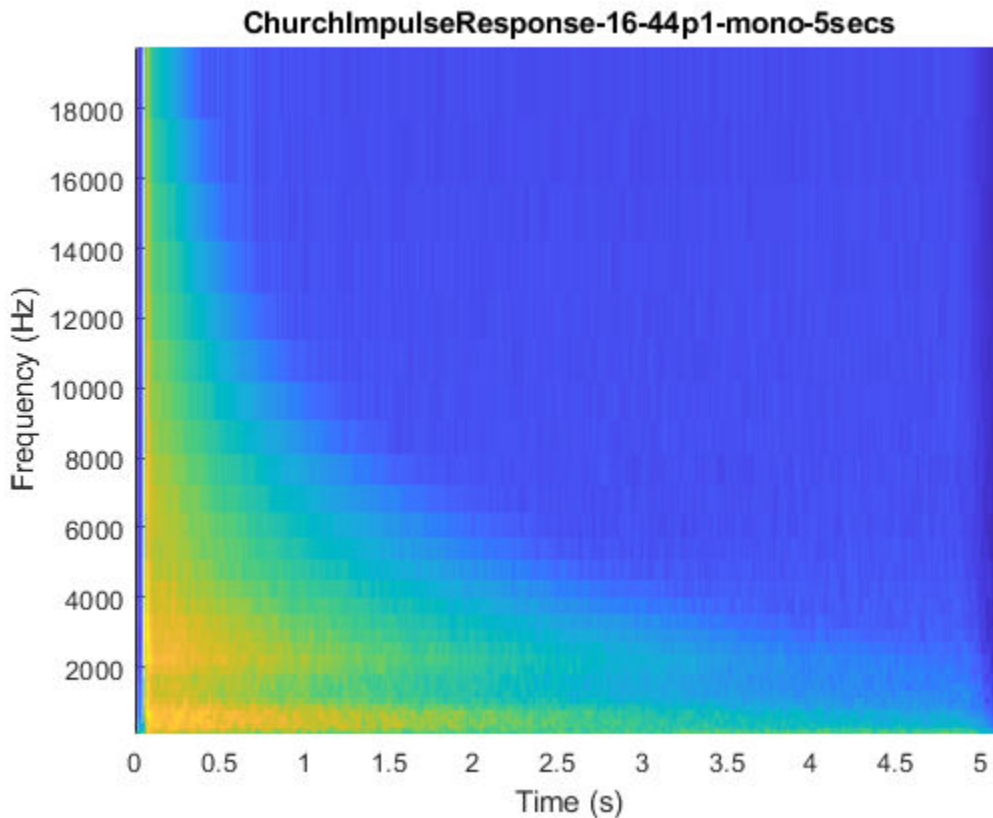
Read the first three audio files and plot the log mel spectrograms. If there are multiple channels, plot only the first channel.

```
for i = 1:3  
    [melSpec,info] = read(ADSnew);  
  
    figure(i)
```

```
surf(info.TimeInstants,info.CenterFrequencies,melSpec(:,:,1),'EdgeColor','none');  
xlabel('Time (s)')  
ylabel('Frequency (Hz)')  
[~,name] = fileparts(info.FileName);  
title(name)  
axis([0 info.TimeInstants(end) info.CenterFrequencies(1) info.CenterFrequencies(end)  
view([0,90])  
end
```







### Output Spectral Shape Features

Use `transform` to create an audio datastore that returns feature vectors.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');  
ADS = audioDatastore(folder);
```

Define a function, `extractFeatureVector`, that transforms the audio data from a time-domain representation to feature vectors.

```
function [dataOut,info] = extractFeatureVector(audioIn,info)

% Convert to frequency-domain representation
windowLength = 256;
overlapLength = 128;
[~,f,~,S] = spectrogram(mean(audioIn,2), ...
                        hann(windowLength,"Periodic"), ...
                        overlapLength, ...
                        windowLength, ...
                        info.SampleRate, ...
                        "power", ...
                        "onesided");

% Extract features
[kurtosis,spread,centroid] = spectralKurtosis(S,f);
skewness = spectralSkewness(S,f);
crest     = spectralCrest(S,f);
decrease = spectralDecrease(S,f);
entropy  = spectralEntropy(S,f);
flatness = spectralFlatness(S,f);
flux     = spectralFlux(S,f);
rolloff  = spectralRolloffPoint(S,f);
slope    = spectralSlope(S,f);

% Concatenate to create feature vectors
dataOut = [kurtosis,spread,centroid,skewness,crest,decrease,entropy,flatness,flux,

end
```

Call `transform` to create a `TransformedDatastore` with `Transforms` set to `extractFeatureVector`.

```
ADSnew = transform(ADS,@extractFeatureVector,'IncludeInfo',true)
```

```
ADSnew =
```

```
TransformedDatastore with properties:
```

```
UnderlyingDatastore: [1x1 audioDatastore]
Transforms: {@extractFeatureVector}
IncludeInfo: 1
```

Call `read` to return the feature vectors for the audio over time.

```
featureMatrix = read(ADSnew);  
[numFeatureVectors,numFeatures] = size(featureMatrix)
```

```
numFeatureVectors =
```

```
    4215
```

```
numFeatures =
```

```
    11
```

### Apply Bandpass Filtering

Use `transform` to create an audio datastore that applies bandpass filtering before returning audio from the `read` function.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');  
ADS = audioDatastore(folder);
```

Define a function, `applyBandpassFilter`, that applies a bandpass filter with a passband between 1 and 15 kHz.

```
function [audioOut,info] = applyBandpassFilter(audioIn,info)  
    audioOut = bandpass(audioIn,[1e3,15e3],info.SampleRate);  
end
```

Call `transform` to create a `TransformedDatastore` with `Transforms` set to `applyBandpassFilter`.

```
ADSnew = transform(ADS,@applyBandpassFilter,'IncludeInfo',true)
```

```
ADSnew =
```

```
TransformedDatastore with properties:
```

```
UnderlyingDatastore: [1x1 audioDatastore]  
Transforms: {@applyBandpassFilter}  
IncludeInfo: 1
```

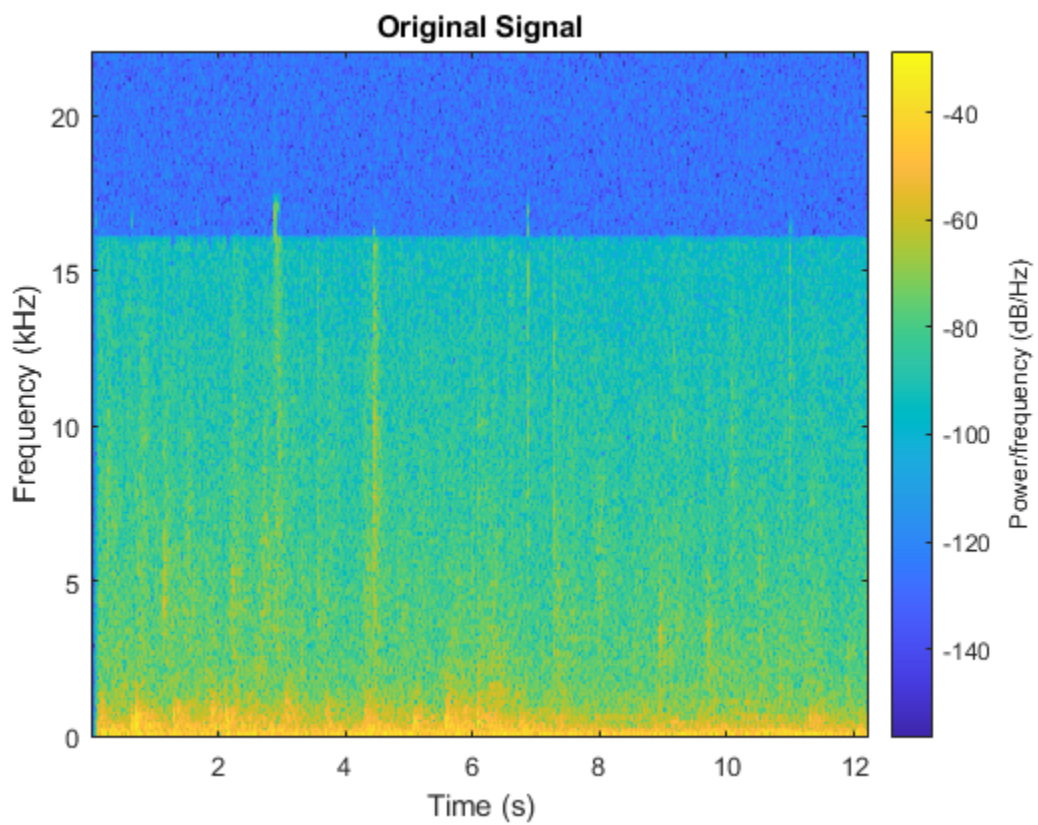
Call `read` to return the bandpass filtered audio from the transform datastore. Call `read` to return the bandpass filtered audio from the original datastore. Plot the spectrograms to visualize the difference.

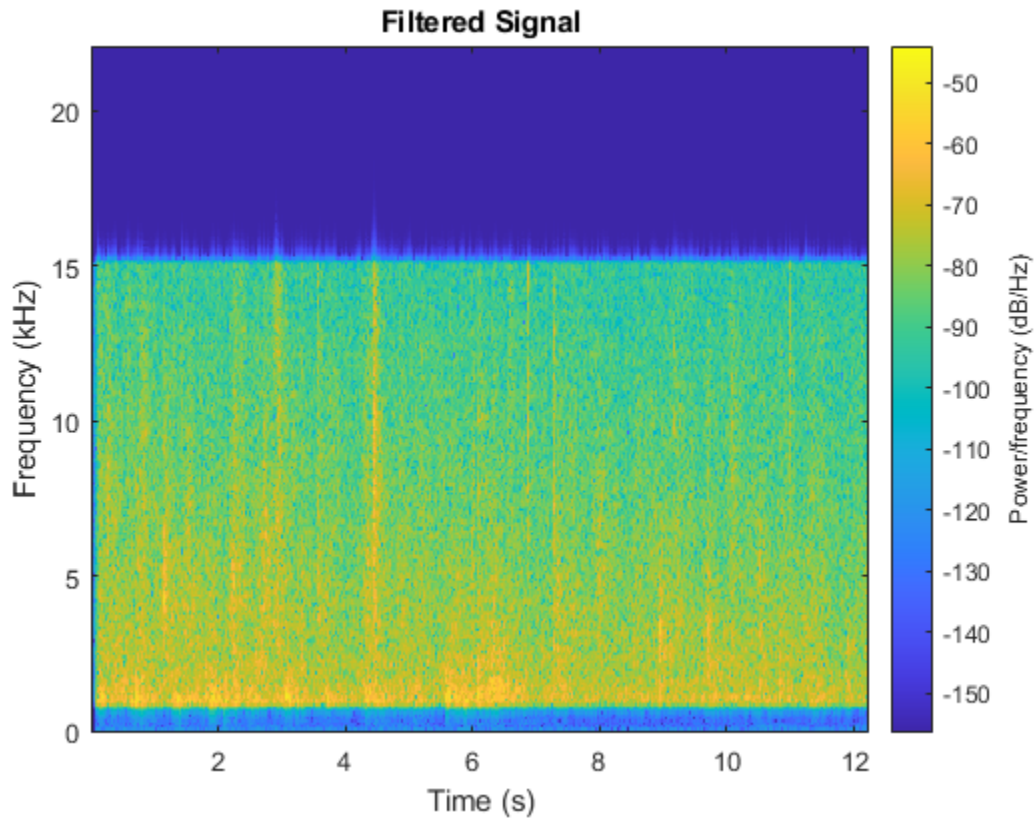
```
[audio1,info1] = read(ADS);  
[audio2,info2] = read(ADSnew);
```

```
figure(1)  
spectrogram(audio1,hann(512),256,512,info1.SampleRate,'yaxis')  
title('Original Signal')
```

```
figure(2)  
spectrogram(audio2,hann(512),256,512,info2.SampleRate,'yaxis')  
title('Filtered Signal')
```







## Input Arguments

### **ADS — Audio datastore**

audioDatastore object

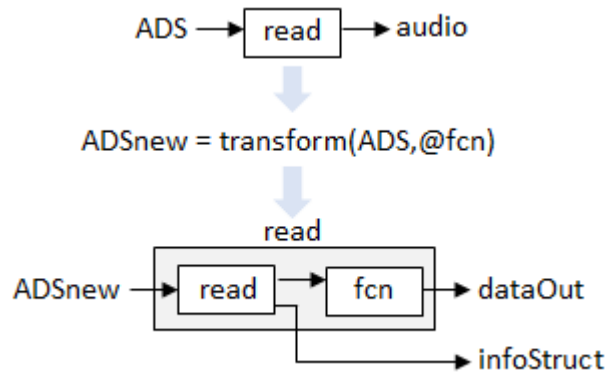
Audio datastore, specified as an audioDatastore object.

### **@fcn — Function that transforms data**

function handle

Function that transforms data, specified as a function handle. The signature of the function depends on the IncludeInfo parameter.

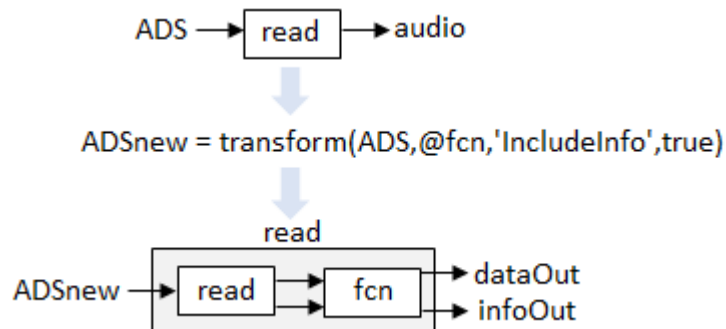
- If `IncludeInfo` is set to `false` (default), the function transforms the audio output from `read`. The info output from `read` is unaltered.



The transform function must have this signature:

```
function dataOut = fcn(audio)
...
end
```

- If `IncludeInfo` is set to `true`, the function transforms the audio output from `read`, and can use or modify the information returned from `read`.



The transform function must have this signature:

```
function [dataOut,infoOut] = fcn(audio,infoIn)
...
end
```

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'IncludeInfo', tf`

### **IncludeInfo** — Pass info through customized read function

`false` (default) | `true`

Pass info through the customized read function, specified as `true` or `false`. If `true`, the transform function can use or modify the information it gets from `read`. If unspecified, `IncludeInfo` defaults to `false`.

Data Types: `logical`

## Output Arguments

### **transformDatastore** — New datastore with customized read

`TransformedDatastore`

New datastore with customized read, returned as a `TransformedDatastore` with `UnderlyingDatastore` set to `ADS`, `Transforms` set to `fcn`, and `IncludeInfo` set to `true` or `false`.

## See Also

`audioDatastore` | `combine` | `hasdata` | `preview` | `read` | `readall` | `reset`

**Introduced in R2019a**

# combine

Combine data from multiple datastores

## Syntax

```
ADSnew = combine(ADS1,ADS2,...,ADSN)
```

## Description

`ADSnew = combine(ADS1,ADS2,...,ADSN)` combines two or more datastores by horizontally concatenating the data returned by `read` of the input datastores.

## Examples

### Combine Datastores

Create a datastore that maintains parity between the audio of the underlying datastores. Create two separate audio datastores, and then create a combined datastore representing the two underlying datastores.

Create a datastore `ads1` that points to the audio files included with Audio Toolbox.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');  
ads1 = audioDatastore(folder);
```

Create a second datastore `ads2` by adding noise to the audio in the `ads1`.

```
ads2 = transform(ads1,@(x) x + 0.01*randn(size(x)) );
```

Create a combined datastore from `ads1` and `ads2`.

```
adsCombined = combine(ads1,ads2);
```

Read the first pair of audio files from the combined datastore. Each read operation on this combined datastore returns a pair of audio signals in a 1-by-2 cell array and a pair of info structs in a 1-by-2 cell array.

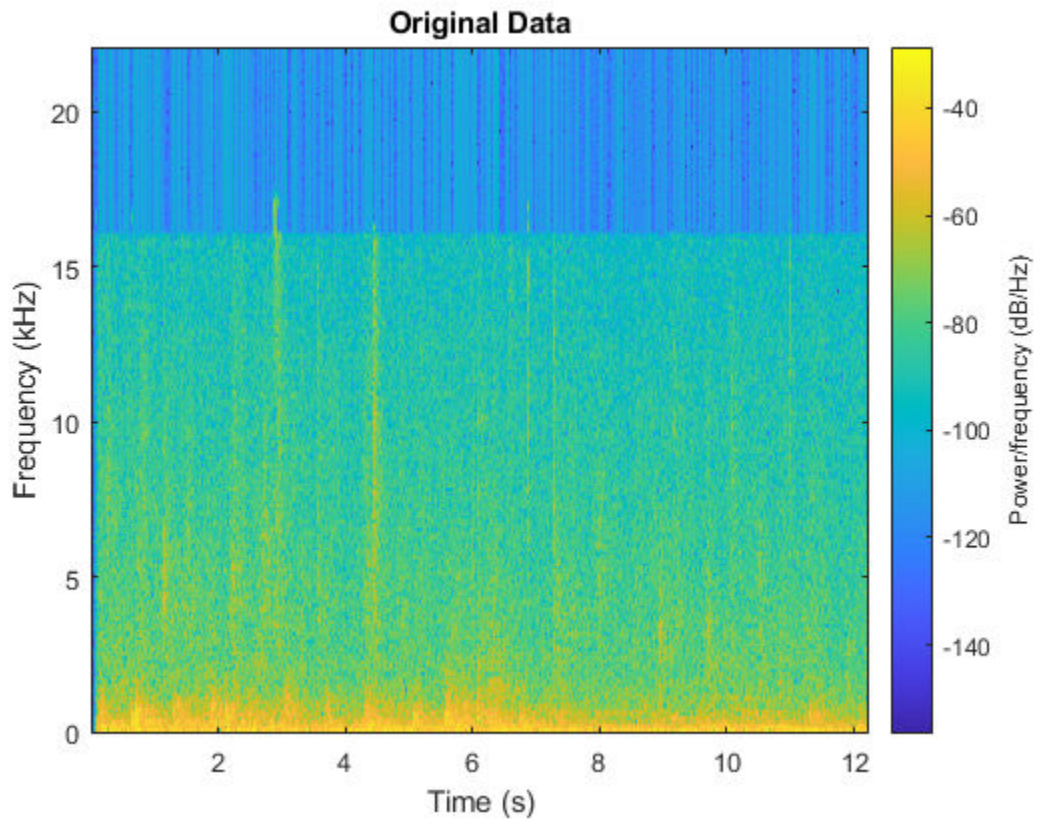
```
[dataOut,infoOut] = read(adsCombined)

dataOut=1×2 cell
    {539648×1 double}    {539648×1 double}

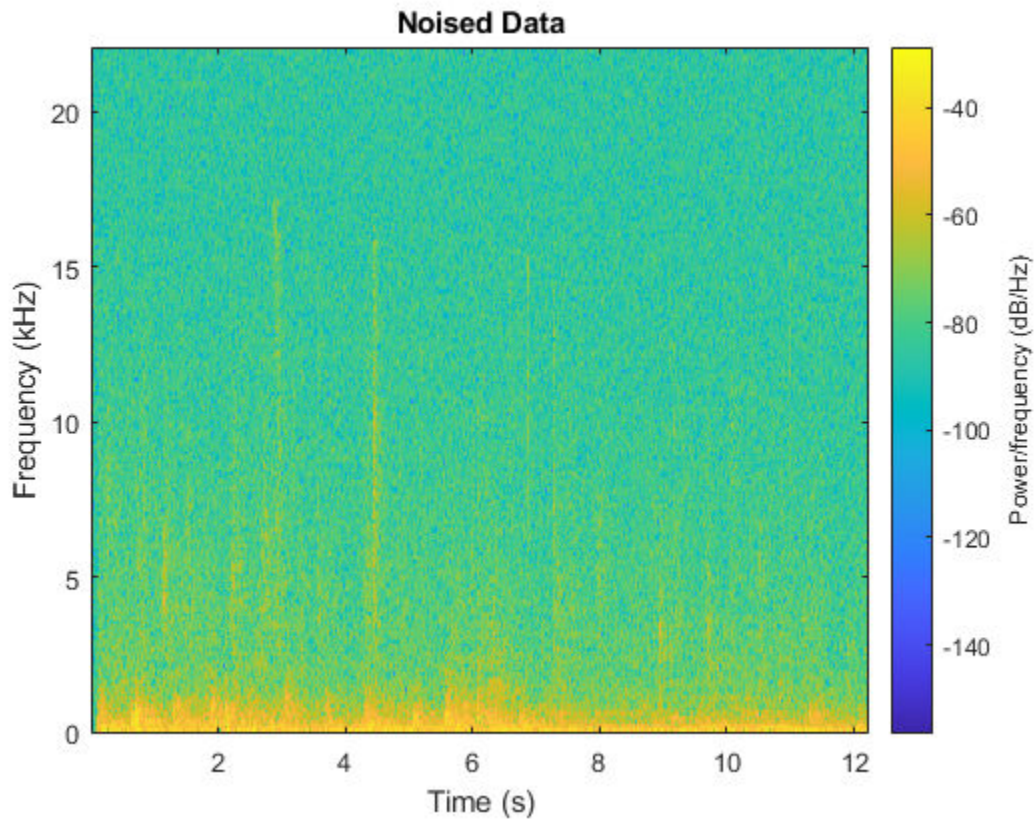
infoOut=1×2 cell
    {1×1 struct}    {1×1 struct}
```

Plot the spectrograms of the first channels from both audio signals.

```
figure(1)
spectrogram(dataOut{1},hamming(512),256,512,infoOut{1}.SampleRate,'yaxis')
title('Original Data')
```



```
figure(2)
idx = size(dataOut,2)/2+1;
spectrogram(dataOut{2},hamming(512),256,512,infoOut{2}.SampleRate,'yaxis')
title('Noised Data')
```



## Input Arguments

**ADS1, ADS2, . . . , ADSN** — Audio datastores to combine  
audioDatastore objects

Audio datastores to combine, specified as two or more comma separated audioDatastore objects.



## Output Arguments

### **ADSnew — New audio datastore with combined data**

audioDatastore object

New audio datastore with combined data, returned as a `matlab.io.datastore.CombinedDatastore` object.

Calling `read` on the combined datastore returns a cell array containing the output of calling `read` on the individual datastores.

## See Also

`audioDatastore` | `hasdata` | `preview` | `read` | `readall` | `reset` | `transform`

**Introduced in R2019a**

## progress

Fraction of files read

## Syntax

```
fractionRead = progress(ADS)
```

## Description

`fractionRead = progress(ADS)` returns the fraction of files read in the datastore as a normalized value in the range [0,1].

## Examples

### Return Fraction of Files Read

Create an `audioDatastore` object `ADS`. Read a file from the datastore and then call `progress` to return the fraction of files read.

```
ADS = audioDatastore(fullfile(matlabroot, 'toolbox', 'audio', 'samples'))
```

```
ADS =
```

```
  audioDatastore with properties:
```

```
      Files: {  
              'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12  
              'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channe  
              ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p  
              ... and 26 more  
            }  
  AlternateFileSystemRoots: {}  
      OutputDataType: 'double'  
      Labels: {}
```

```
fractionOfFilesRead = progress(ADS)
fractionOfFilesRead = 0
data = read(ADS);
fractionOfFilesRead = progress(ADS)
fractionOfFilesRead = 0.0345
```

## Input Arguments

### **ADS — Audio datastore**

audioDatastore object

Specify ADS as an audioDatastore object.

## Output Arguments

### **fractionRead — Fraction of files read**

normalized value in the range [0,1]

Fraction of files read, returned as a normalized value in the range [0,1].

Data Types: double

## See Also

audioDatastore | hasdata

## Topics

“Speech Command Recognition Using Deep Learning”

“Speaker Identification Using Pitch and MFCC”

“Denoise Speech Using Deep Learning Networks”

“Classify Gender Using LSTM Networks”

“Music Genre Classification Using Wavelet Time Scattering”

**Introduced in R2018b**

## numpartitions

Return estimate for reasonable number of partitions for parallel processing

### Syntax

```
n = numpartitions(ADS)
n = numpartitions(ADS,pool)
```

### Description

`n = numpartitions(ADS)` returns the default number of partitions for the datastore, ADS. The default number of partitions is the total number of files.

`n = numpartitions(ADS,pool)` returns a reasonable number of partitions to parallelize ADS over the parallel pool, based on the total number of files and the number of workers in pool. To parallelize datastore access, you must have Parallel Computing Toolbox™ installed.

### Examples

#### Estimate Reasonable Number of Partitions for Audio Datastore

`numpartitions` returns a reasonable number of partitions for an audio datastore. You can use `numpartitions` as input to the `partition` function.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

Use `numpartitions` to estimate a reasonable number of partitions for the audio datastore, ADS. By default, `numpartitions` returns the number of files the audio datastore points to.

```
n = numpartitions(ADS)
n = 29
```

### Number of Partitions for Parallel Datastore Access

Partition a datastore to facilitate parallel access over the available parallel pool of workers.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');
ADS = audioDatastore(folder);
```

Return an estimate for a reasonable number of partitions for parallel processing, given the current parallel pool.

```
pool = gcp;
n = numpartitions(ADS, pool);
```

Partition the audio datastore and read the data in each part.

```
parfor ii = 1:n
    subds = partition(ADS, n, ii);
    while hasdata(subds)
        data = read(subds);
    end
end
```

## Input Arguments

### ADS — Audio datastore

audioDatastore object

Specify ADS as an audioDatastore object.

**pool** — **Parallel pool**

parallel pool object

Parallel pool object.

## Output Arguments

**n** — **Number of partitions**

positive integer

Number of partitions to parallelize datastore access over.

## See Also

audioDatastore | partition

## Topics

“Speech Command Recognition Using Deep Learning”

“Speaker Identification Using Pitch and MFCC”

“Denoise Speech Using Deep Learning Networks”

“Classify Gender Using LSTM Networks”

“Music Genre Classification Using Wavelet Time Scattering”

**Introduced in R2018b**

# partition

Partition datastore and return on partitioned portion

## Syntax

```
subADS = partition(ADS,numPartitions,index)
subADS = partition(ADS,'Files',index)
subADS = partition(ADS,'Files',filename)
```

## Description

`subADS = partition(ADS,numPartitions,index)` partitions datastore ADS into the number of parts specified by `numPartitions` and returns the partition corresponding to the `index`.

`subADS = partition(ADS,'Files',index)` partitions the datastore by files and returns the partition corresponding to the file of `index` in the `Files` property.

`subADS = partition(ADS,'Files',filename)` partitions the datastore by files and returns the partition corresponding to the file specified by `filename`.

## Examples

### Partition Datastore into Specific Number of Parts

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder)
```

```
ADS =
    audioDatastore with properties:
```

```
Files: {
    'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-128k-16bit-10sec.wav'
    'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channel-10sec.wav'
    '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-10sec.wav'
    ... and 26 more
}
AlternateFileSystemRoots: {}
OutputDataType: 'double'
Labels: {}
```

Partition the datastore into three parts.

```
subADS1 = partition(ADS,3,1)
```

```
subADS1 =
    audioDatastore with properties:
```

```
Files: {
    'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-128k-16bit-10sec.wav'
    'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channel-10sec.wav'
    '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-10sec.wav'
    ... and 7 more
}
AlternateFileSystemRoots: {}
OutputDataType: 'double'
Labels: {}
```

```
subADS2 = partition(ADS,3,2)
```

```
subADS2 =
    audioDatastore with properties:
```

```
Files: {
    'B:\matlab\toolbox\audio\samples\JetAirplane-16-11p025-mono-10sec.wav'
    'B:\matlab\toolbox\audio\samples\Laughter-16-8-mono-4sec.wav'
    'B:\matlab\toolbox\audio\samples\MainStreetOne-24-96-ster-10sec.wav'
    ... and 7 more
}
AlternateFileSystemRoots: {}
OutputDataType: 'double'
Labels: {}
```

```
subADS3 = partition(ADS,3,3)
```



```
subADS3 =
    audioDatastore with properties:

        Files: {
            'B:\matlab\toolbox\audio\samples\RockGuitar-16-96-stereo-10s.wav'
            'B:\matlab\toolbox\audio\samples\SoftGuitar-44p1_mono-10s.wav'
            'B:\matlab\toolbox\audio\samples\SpeechDFT-16-8-mono-5sec.wav'
            ... and 6 more
        }
        AlternateFileSystemRoots: {}
        OutputDataType: 'double'
        Labels: {}
```

### Partition Datastore into Default Number of Parts

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');
ADS = audioDatastore(folder);
```

Get the default number of partitions for ADS.

```
n = numpartitions(ADS);
```

Partition the datastore into the default number of partitions and return the datastore corresponding to the first partition.

```
subADS = partition(ADS,n,1);
```

Read the data in subADS.

```
while hasdata(subADS)
    data = read(subADS);
end
```

### Partition Datastore by Files

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');  
ADS = audioDatastore(folder);
```

Partition the datastore by files and return the part corresponding to the second file. `subds` contains one file.

```
subds = partition(ADS, 'Files', 2)
```

```
subds =  
    audioDatastore with properties:
```

```
        Files: {  
                'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channe  
                }  
AlternateFileSystemRoots: {}  
        OutputDataType: 'double'  
        Labels: {}
```

### Number of Partitions for Parallel Datastore Access

Partition a datastore to facilitate parallel access over the available parallel pool of workers.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');  
ADS = audioDatastore(folder);
```

Return an estimate for a reasonable number of partitions for parallel processing, given the current parallel pool.

```
pool = gcp;  
n = numpartitions(ADS, pool);
```

Partition the audio datastore and read the data in each part.

```
parfor ii = 1:n
    subds = partition(ADS,n,ii);
    while hasdata(subds)
        data = read(subds);
    end
end
```

## Input Arguments

### **ADS — Audio datastore**

audioDatastore object

Audio datastore, specified as an audioDatastore object.

### **numPartitions — Number of partitions**

positive integer

Number of partitions, specified as a positive integer. Use numpartitions to estimate a reasonable value for numPartitions.

Data Types: double

### **index — Index of sub-datastore**

positive integer

Index of sub-datastore, specified as a positive integer in the range [1,numPartitions].

Data Types: double

### **filename — File name**

character vector

File name, specified as a character vector.

The value of filename must match exactly the file name contained in the Files property of the datastore.

Data Types: char

## Output Arguments

### **subADS — Output audio datastore**

audioDatastore object

Output audio datastore, returned as an audioDatastore object.

## See Also

audioDatastore | numpartitions

## Topics

“Speech Command Recognition Using Deep Learning”

“Speaker Identification Using Pitch and MFCC”

“Denoise Speech Using Deep Learning Networks”

“Classify Gender Using LSTM Networks”

“Music Genre Classification Using Wavelet Time Scattering”

**Introduced in R2018b**

# countEachLabel

Count number of unique labels

## Syntax

```
tbl = countEachLabel(ADS)
tbl = countEachLabel(ADS, 'TableVariable', VariableName)
```

## Description

`tbl = countEachLabel(ADS)` counts the number of times each unique label occurs in the datastore. In other words, it counts the number of files with each unique label. The output `tbl` is a table with variable names `Label` and `Count`.

`tbl = countEachLabel(ADS, 'TableVariable', VariableName)` counts the number of times each unique label occurs in the datastore. When the datastore `Labels` property is specified by a table, you must specify `VariableName`. `VariableName` is the table variable (column) name you want to count.

## Examples

### Label Count

Specify the file path to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');
```

Create an audio datastore that points to the specified folder. Specify the `LabelSource` property as `foldernames`, so that the label associated with each file is set to the folder name that contains the file.

```
ads = audioDatastore(folder, 'LabelSource', 'foldernames')
```

```
ads =  
    audioDatastore with properties:
```

```
Files: {
    'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-128k-16bit-32sec.wav'
    'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channel-128k-16bit-32sec.wav'
    '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-128k-16bit-32sec.wav'
    ... and 26 more
}
Labels: [samples; samples; samples ... and 26 more categorical]
AlternateFileSystemRoots: {}
OutputDataType: 'double'
```

Call `countEachLabel` to count the number of times each unique label occurs.

```
tbl = countEachLabel(ads)
```

```
tbl=1x2 table
  Label      Count
  _____  _____
  samples      29
```

### Label Count when Labels Is Specified by Table

If the `Labels` property of an audio datastore is specified as a table, you must specify the table variable name when counting labels.

Specify the file path to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');
```

Create an audio datastore that points to the specified folder.

```
ADS = audioDatastore(folder)
```

```
ADS =
  audioDatastore with properties:
```

```
Files: {
    'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-128k-16bit-32sec.wav'
    'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channel-128k-16bit-32sec.wav'
    ... and 26 more
}
```

```

        ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p
        ... and 26 more
    }
    AlternateFileSystemRoots: {}
    OutputDataType: 'double'
    Labels: {}

```

The file names contain information about the files. Parse the file names to collect information about whether a file is mono or stereo and whether a file is longer than thirty seconds. Create a table containing the parsed information and then set the `Labels` property of the audio datastore to the label table.

```

numFiles = numel(ADS.Files);

numChannels = cell(numFiles,1);
isLong = cell(numFiles,1);

for i = 1:numFiles
    if ~isempty(strfind(ADS.Files{i}, 'mono'))
        numChannels{i} = 'mono';
    elseif ~isempty(strfind(ADS.Files{i}, 'stereo'))
        numChannels{i} = 'stereo';
    else
        numChannels{i} = 'unknown';
    end

    secs = str2double(regexp(ADS.Files{i}, '-(\d+)secs', 'tokens', 'once'));
    if secs > 30
        isLong{i} = true;
    elseif secs <= 30
        isLong{i} = false;
    else
        isLong{i} = 'unknown';
    end
end
labelTable = table(numChannels,isLong, ...
    'VariableNames',{'NumberOfChannels','IsLongerThan30Seconds'});

ADS.Labels = labelTable;

```

Call `countEachLabel` on the audio datastore and specify the `TableVariable` as `NumberOfChannels`. Call `countEachLabel` and specify the `TableVariable` as `IsLongerThan30Seconds`.

```
countNumberOfChannelLabels = countEachLabel(ADS, 'TableVariable', 'NumberOfChannels')
```

```
countNumberOfChannelLabels=3x2 table
```

NumberOfChannels	Count
mono	16
stereo	11
unknown	2

```
countDurationLabels = countEachLabel(ADS, 'TableVariable', 'IsLongerThan30Seconds')
```

```
countDurationLabels=3x2 table
```

IsLongerThan30Seconds	Count
false	18
true	7
unknown	4

## Input Arguments

### **ADS — Audio datastore**

audioDatastore object

Specify ADS as an audioDatastore object.

### **VariableName — Label table variable name**

character vector | string

Label table variable name, specified as a character vector or string that corresponds to a table variable of the Label property.

This syntax is required if the Label property of audioDatastore is specified by a table.

Data Types: char | string



## Output Arguments

### **tbl** — Table of label counts

two-column table

Table of label counts, returned as a two-column table containing the name of each label in ADS and the number of files associated with each label.

Data Types: table

## See Also

audioDatastore | splitEachLabel

## Topics

“Speech Command Recognition Using Deep Learning”

“Speaker Identification Using Pitch and MFCC”

“Denoise Speech Using Deep Learning Networks”

“Classify Gender Using LSTM Networks”

“Music Genre Classification Using Wavelet Time Scattering”

**Introduced in R2018b**

## splitEachLabel

Splits datastore according to specified label proportions

### Syntax

```
[ADS1,ADS2] = splitEachLabel(ADS,p)
[ADS1,...,ADSM] = splitEachLabel(ADS,p1,...,pN)
___ = splitEachLabel(___,'randomized')
___ = splitEachLabel(___ ,Name,Value)
```

### Description

[ADS1,ADS2] = splitEachLabel(ADS,p) splits the audio files in ADS into two new datastores, ADS1 and ADS2. The new datastore ADS1 contains the first *p* files from each label ,and ADS2 contains the remaining files from each label. *p* can be either a number between 0 and 1, exclusive, indicating the percentage of the files from each label to assign to ADS1, or an integer indicating the absolute number of files from each label to assign to ADS1.

[ADS1,...,ADSM] = splitEachLabel(ADS,p1,...,pN) splits the datastore into *N* +1 new datastores. The new datastore ADS1 contains the first *p*<sub>1</sub> files from each label, the next new datastore ADS2 contains the next *p*<sub>2</sub> files, and so on. If *p*<sub>1</sub>, ..., *p*<sub>N</sub> represent numbers of files, then their sum must be no more than the number of files in the smallest label in the original datastore, ADS.

\_\_\_ = splitEachLabel( \_\_\_ , 'randomized') randomly assigns the specified proportion of files from each label to the new datastores.

\_\_\_ = splitEachLabel( \_\_\_ , Name, Value) specifies the properties of the new datastores using one or more name-value pair arguments. For example, you can specify which labels to split with 'Include', 'labelname'.

### Examples

## Split by Fractions

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');
ADS = audioDatastore(folder, 'FileExtensions', '.wav');
```

Add the label A to the first half of the files, and the label B to the second half. If there are an odd number of files, assign the extra file the label B. Call `countEachLabel` to confirm that half of the files are labeled A and half the files are labeled B.

```
labels = [repmat({'A'},1,floor(numel(ADS.Files)/2)), ...
          repmat({'B'},1,ceil(numel(ADS.Files)/2))];
ADS.Labels = labels;
```

```
countEachLabel(ADS)
```

```
ans=2x2 table
  Label  Count
  -----
      A     10
      B     11
```

Split ADS into two datastores, ADS1 and ADS2, specifying that each new datastore contains fifty percent of each label and the corresponding files. Call `countEachLabel` to confirm that half of the files are labeled A and half of the files are labeled B for each of the new datastores.

```
[ADS1,ADS2] = splitEachLabel(ADS,0.5)
```

```
ADS1 =
```

```
audioDatastore with properties:
```

```
Files: {
    'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-11...
    'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channe...
    '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p...
    ... and 8 more
}
Labels: {'A'; 'A'; 'A' ... and 8 more}
AlternateFileSystemRoots: {}
```

```
OutputDataType: 'double'
```

```
ADS2 =
```

```
  audioDatastore with properties:
```

```
      Files: {  
          'B:\matlab\toolbox\audio\samples\Engine-16-44p1-stereo-20  
          'B:\matlab\toolbox\audio\samples\FemaleSpeech-16-8-mono-3  
          'B:\matlab\toolbox\audio\samples\Heli_16ch_ACN_SN3D.wav'  
          ... and 7 more  
      }  
      Labels: {'A'; 'A'; 'A' ... and 7 more}  
  AlternateFileSystemRoots: {}  
      OutputDataType: 'double'
```

```
ADS1count = countEachLabel(ADS1)
```

```
ADS1count=2×2 table
```

Label	Count
A	5
B	6

```
ADS2count = countEachLabel(ADS2)
```

```
ADS2count=2×2 table
```

Label	Count
A	5
B	5

### Split by Number of Files

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder,'FileExtensions','.wav');
```

Add the label A to the first half of the files, and the label B to the second half. If there are an odd number of files, assign the extra file the label B. Call `countEachLabel` to confirm that half of the files are labeled A and half the files are labeled B.

```
labels = [repmat({'A'},1,floor(numel(ADS.Files)/2)), ...
          repmat({'B'},1,ceil(numel(ADS.Files)/2))];
ADS.Labels = labels;
```

```
countEachLabel(ADS)
```

```
ans=2x2 table
  Label    Count
  _____  _____
      A         10
      B         11
```

Split ADS into two datastores, ADS1 and ADS2. Specify that ADS1 contains four of each label and its corresponding file. ADS2 contains the remaining labels and corresponding files. Call `countEachLabel` to confirm that ADS1 contains four files labeled A and four files labeled B, and that ADS2 contains the remaining labels.

```
[ADS1,ADS2] = splitEachLabel(ADS,4)
```

```
ADS1 =
```

```
  audioDatastore with properties:
```

```

      Files: {
          'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-17
          'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4chann
          ' ... \toolbox\audio\samples\ChurchImpulseResponse-16-44p
          ... and 5 more
      }
      Labels: {'A'; 'A'; 'A' ... and 5 more}
  AlternateFileSystemRoots: {}
      OutputDataType: 'double'
```

```
ADS2 =
```

```
  audioDatastore with properties:
```

```

Files: {
    'B:\matlab\toolbox\audio\samples\Counting-16-44p1-mono-1...'
    'B:\matlab\toolbox\audio\samples\Engine-16-44p1-stereo-2...'
    'B:\matlab\toolbox\audio\samples\FemaleSpeech-16-8-mono-3...'
    ... and 10 more
}
Labels: {'A'; 'A'; 'A' ... and 10 more}
AlternateFileSystemRoots: {}
OutputDataType: 'double'

```

```
ADS1count = countEachLabel(ADS1)
```

```
ADS1count=2×2 table
Label      Count
-----
A          4
B          4
```

```
ADS2count = countEachLabel(ADS2)
```

```
ADS2count=2×2 table
Label      Count
-----
A          6
B          7
```

### Split Several Ways by Fractions

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder,'FileExtensions','.wav');
```

Add the label A to the first half of the files, and the label B to the second half. If there is an odd number of files, assign the extra file the label B. Call `countEachLabel` to confirm that half of the files are labeled A and half the files are labeled B.

```
labels = [repmat({'A'},1,floor(numel(ADS.Files)/2)), ...
          repmat({'B'},1,ceil(numel(ADS.Files)/2))];
ADS.Labels = labels;
```

```
countEachLabel(ADS)
```

```
ans=2x2 table
    Label    Count
    _____
         A         10
         B         11
```

Split ADS into three new datastores, ADS60, ADS10, and ADS30. The first datastore, ADS60, contains the first 60% of files with the A label and the first 60% of files with the B label. ADS10 contains the next 10% of files from each label. ADS30 contains the remaining 30% of files from each label. If the percentage applied to a label does not result in a whole number of files, `splitEachLabel` rounds down to the nearest whole number.

```
[ADS60,ADS10,ADS30] = splitEachLabel(ADS,0.6,0.1)
```

```
ADS60 =
```

```
audioDatastore with properties:
```

```
Files: {
    'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12'
    'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channe'
    '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p'
    ... and 10 more
}
Labels: {'A'; 'A'; 'A' ... and 10 more}
AlternateFileSystemRoots: {}
OutputDataType: 'double'
```

```
ADS10 =
```

```
audioDatastore with properties:
```

```
Files: {
    'B:\matlab\toolbox\audio\samples\FemaleSpeech-16-8-mono-3'
    'B:\matlab\toolbox\audio\samples\Turbine-16-44p1-mono-22'
}
Labels: {'A'; 'B'}
AlternateFileSystemRoots: {}
```

```
OutputDataType: 'double'
```

```
ADS30 =
```

```
  audioDatastore with properties:
```

```
      Files: {  
              'B:\matlab\toolbox\audio\samples\Heli_16ch_ACN_SN3D.wav'  
              'B:\matlab\toolbox\audio\samples\JetAirplane-16-11p025-m  
              'B:\matlab\toolbox\audio\samples\Laughter-16-8-mono-4sec  
              ... and 3 more  
            }  
      Labels: {'A'; 'A'; 'A' ... and 3 more}  
AlternateFileSystemRoots: {}  
      OutputDataType: 'double'
```

Call `countEachLabel` to confirm the correct distribution of labels for each datastore.

```
countEachLabel(ADS60)
```

```
ans=2x2 table
```

Label	Count
A	6
B	7

```
countEachLabel(ADS10)
```

```
ans=2x2 table
```

Label	Count
A	1
B	1

```
countEachLabel(ADS30)
```

```
ans=2x2 table
```

Label	Count
A	3



B 3

### Split Labels Several Ways by Number of Files

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder,'FileExtensions','.wav');
```

Add the label A to the first half of the files, and the label B to the second half. If there is an odd number of files, assign the extra file the label B. Call `countEachLabel` to confirm that half of the files are labeled A and half the files are labeled B.

```
labels = [repmat({'A'},1,floor(numel(ADS.Files)/2)), ...
          repmat({'B'},1,ceil(numel(ADS.Files)/2))];
ADS.Labels = labels;
```

```
countEachLabel(ADS)
```

```
ans=2x2 table
      Label      Count
      ----      -
      A          10
      B          11
```

Split ADS into three new datastores, ADS1, ADS2, and ADS3. The first datastore, ADS1, contains the first file with the A label and the first file with the B label. ADS2 contains the next file from each label. ADS3 contains the remaining files from each label. If the percentage applied to a label does not result in a whole number of files, `splitEachLabel` rounds down to the nearest whole number.

```
[ADS1,ADS2,ADS3] = splitEachLabel(ADS,1,1)
```

```
ADS1 =
  audioDatastore with properties:
```

```
Files: {
        'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-1
```

```
        'B:\matlab\toolbox\audio\samples\MainStreetOne-24-96-ste
    }
    Labels: {'A'; 'B'}
AlternateFileSystemRoots: {}
    OutputDataType: 'double'

ADS2 =
    audioDatastore with properties:

        Files: {
            'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channe
            'B:\matlab\toolbox\audio\samples\NoisySpeech-16-22p5-mono
        }
        Labels: {'A'; 'B'}
AlternateFileSystemRoots: {}
    OutputDataType: 'double'

ADS3 =
    audioDatastore with properties:

        Files: {
            '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p
            'B:\matlab\toolbox\audio\samples\Click-16-44p1-mono-0.2s
            'B:\matlab\toolbox\audio\samples\Counting-16-44p1-mono-1
            ... and 14 more
        }
        Labels: {'A'; 'A'; 'A' ... and 14 more}
AlternateFileSystemRoots: {}
    OutputDataType: 'double'
```

Call `countEachLabel` to confirm the correct distribution of labels for each datastore.

```
countEachLabel(ADS1)
```

```
ans=2x2 table
```

Label	Count
A	1
B	1

```
countEachLabel(ADS2)
```

```
ans=2x2 table
  Label    Count
  -----
     A         1
     B         1
```

```
countEachLabel(ADS3)
```

```
ans=2x2 table
  Label    Count
  -----
     A         8
     B         9
```

### Split Labels in Random Order

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');
ADS = audioDatastore(folder, 'FileExtensions', '.wav')
```

```
ADS =
  audioDatastore with properties:
```

```
Files: {
    'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-128k.wav'
    'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channel-128k.wav'
    '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-128k.wav'
    ... and 18 more
}
AlternateFileSystemRoots: {}
OutputDataType: 'double'
Labels: {}
```

Add the label A to the first half of the files, and the label B to the second half. If there is an odd number of files, assign the extra file the label B. Call `countEachLabel` to confirm that half of the files are labeled A and half the files are labeled B.

```
labels = [repmat({'A'},1,floor(numel(ADS.Files)/2)), ...
          repmat({'B'},1,ceil(numel(ADS.Files)/2))];
ADS.Labels = labels;
```

```
countEachLabel(ADS)
```

```
ans=2x2 table
    Label    Count
    -----
         A         10
         B         11
```

Create two new datastores from the files in **ADS** by randomly drawing from each label. The first datastore, **ADS1**, contains two random files with the A label and two random files with the B label. **ADS2** contains the remaining files from each label.

```
[ADS1,ADS2] = splitEachLabel(ADS,2,'randomized')
```

```
ADS1 =
```

```
audioDatastore with properties:
```

```
Files: {
    '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p
    'B:\matlab\toolbox\audio\samples\Engine-16-44p1-stereo-2
    'B:\matlab\toolbox\audio\samples\MainStreetOne-24-96-ste
    ... and 1 more
}
Labels: {'A'; 'A'; 'B' ... and 1 more}
AlternateFileSystemRoots: {}
OutputDataType: 'double'
```

```
ADS2 =
```

```
audioDatastore with properties:
```

```
Files: {
    'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-1
    'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channe
    'B:\matlab\toolbox\audio\samples\Click-16-44p1-mono-0.2s
    ... and 14 more
}
Labels: {'A'; 'A'; 'A' ... and 14 more}
AlternateFileSystemRoots: {}
```

```
OutputDataType: 'double'
```

### Include and Exclude Specified Labels

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');
ADS = audioDatastore(folder, 'FileExtensions', '.wav')
```

```
ADS =
```

```
audioDatastore with properties:
```

```
Files: {
    'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-128k-16bit.wav'
    'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channel-128k-16bit.wav'
    '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-128k-16bit.wav'
    ... and 18 more
}
AlternateFileSystemRoots: {}
OutputDataType: 'double'
Labels: {}
```

Add the label A to the first half of the files, and the label B to the second half. If there is an odd number of files, assign the extra file the label B. Call `countEachLabel` to confirm that half of the files are labeled A and half the files are labeled B.

```
labels = [repmat({'A'},1,floor(numel(ADS.Files)/2)), ...
    repmat({'B'},1,ceil(numel(ADS.Files)/2))];
ADS.Labels = labels;
```

```
countEachLabel(ADS)
```

```
ans =
```

```
2x2 table
```

Label	Count
A	10
B	11

Create two new datastores from the files in ADS, including only the files with the A label. ADS1 contains the first 70% of files with the A label, and ADS2 contains the remaining 30% of labels with the A label.

```
[ADS1,ADS2] = splitEachLabel(ADS,0.7,'Include','A')
```

```
ADS1 =
```

```
audioDatastore with properties:
```

```
Files: {
    'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-128k.wav'
    'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channel-128k.wav'
    '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-128k.wav'
    ... and 4 more
}
Labels: {'A'; 'A'; 'A' ... and 4 more}
AlternateFileSystemRoots: {}
OutputDataType: 'double'
```

```
ADS2 =
```

```
audioDatastore with properties:
```

```
Files: {
    'B:\matlab\toolbox\audio\samples\Heli_16ch_ACN_SN3D.wav'
    'B:\matlab\toolbox\audio\samples\JetAirplane-16-11p025-mono-128k.wav'
    'B:\matlab\toolbox\audio\samples\Laughter-16-8-mono-4sec.wav'
}
Labels: {'A'; 'A'; 'A'}
AlternateFileSystemRoots: {}
OutputDataType: 'double'
```

Equivalently, you can split only the A label by excluding the B label.

```
[ADS1,ADS2] = splitEachLabel(ADS,0.7,'Exclude','B')
```

```
ADS1 =
```

```
audioDatastore with properties:
```

```
    Files: {
        'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-128k.wav'
        'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channel-128k.wav'
        '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-128k.wav'
        ... and 4 more
    }
    Labels: {'A'; 'A'; 'A' ... and 4 more}
AlternateFileSystemRoots: {}
    OutputDataType: 'double'
```

```
ADS2 =
```

```
audioDatastore with properties:
```

```
    Files: {
        'B:\matlab\toolbox\audio\samples\Heli_16ch_ACN_SN3D.wav'
        'B:\matlab\toolbox\audio\samples\JetAirplane-16-11p025-mono-128k.wav'
        'B:\matlab\toolbox\audio\samples\Laughter-16-8-mono-4sec-128k.wav'
    }
    Labels: {'A'; 'A'; 'A'}
AlternateFileSystemRoots: {}
    OutputDataType: 'double'
```

### Split Using Fraction and Label Table

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder)
```

```
ADS =
```

```
audioDatastore with properties:
```

```
Files: {  
    'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-11-16-16-4channe  
    'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channe  
    '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p  
    ... and 26 more  
}  
AlternateFileSystemRoots: {}  
OutputDataType: 'double'  
Labels: {}
```

Create a label table with two variables:

- `containsMusic` -- Can be either `true` or `false`.
- `instrument` -- Can be `Guitar`, `Drums`, or `Unknown`.

```
containsGuitar = contains(ADS.Files, 'guitar', 'IgnoreCase', true);  
containsDrums = contains(ADS.Files, 'drum', 'IgnoreCase', true);  
containsMusic = or(containsGuitar, containsDrums);
```

```
instrument = strings(size(ADS.Files));  
instrument(:) = "Unknown";  
instrument(containsGuitar) = "Guitar";  
instrument(containsDrums) = "Drums";
```

Assign the label table to the `Labels` property of audio datastore to associate the rows of the label table with the rows of the datastore. Call `countEachLabel` to determine the incidences of `containsMusic` and `instrument`.

```
labels = table(containsMusic, instrument);  
ADS.Labels = labels;
```

```
containsMusicCount = countEachLabel(ADS, 'TableVariable', 'containsMusic')
```

```
containsMusicCount=2x2 table  
containsMusic    Count  
-----  
false            22  
true             7
```

```
instrumentCount = countEachLabel(ADS, 'TableVariable', 'instrument')
```

```
instrumentCount=3x2 table  
instrument    Count
```



Drums	4
Guitar	3
Unknown	22

Split the datastore ADS into two, based on whether the audio file contains music. ADS1 contains 70% of the audio files that contain music, and ADS2 contains the rest. Call `countEachLabel` to verify that the ratio of `containsMusic == true` to `containsMusic == false` is preserved for the new datastores, within rounding.

```
[ADS1,ADS2] = splitEachLabel(ADS,0.7,'TableVariable','containsMusic');
ADS1_containsMusicCount = countEachLabel(ADS1,'TableVariable','containsMusic')
```

```
ADS1_containsMusicCount=2x2 table
containsMusic    Count
```

false	15
true	5

```
ADS2_containsMusicCount = countEachLabel(ADS2,'TableVariable','containsMusic')
```

```
ADS2_containsMusicCount=2x2 table
containsMusic    Count
```

false	7
true	2

Split the datastore ADS into two, based on the type of instrument present in the audio file. ADS3 contains 25% of the audio files that have an instrument label, and ADS4 contains the rest. Call `countEachLabel` to verify that the ratio of `instrument == "drums"` to `instrument == "guitar"` is preserved for the new datastores, within rounding.

```
[ADS3,ADS4] = splitEachLabel(ADS,0.25,'TableVariable','instrument');
ADS3_instrumentCount = countEachLabel(ADS3,'TableVariable','instrument')
```

```
ADS3_instrumentCount=3x2 table
instrument    Count
```

Drums	1
Guitar	1
Unknown	6

```
ADS4_instrumentCount = countEachLabel(ADS4,'TableVariable','instrument')
```

```
ADS4_instrumentCount=3x2 table
```

instrument	Count
Drums	3
Guitar	2
Unknown	16

### Split by Number of Files and Label Table

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');  
ADS = audioDatastore(folder);
```

Create a label table with two variables:

- `containsMusic` - Can be either true or false.
- `instrument` - Can be Guitar, Drums, or Unknown.

```
containsGuitar = contains(ADS.Files,'guitar','IgnoreCase',true);  
containsDrums = contains(ADS.Files,'drum','IgnoreCase',true);  
containsMusic = or(containsGuitar,containsDrums);
```

```
instrument = strings(size(ADS.Files));  
instrument(:) = "Unknown";  
instrument(containsGuitar) = "Guitar";  
instrument(containsDrums) = "Drums";
```

Assign the label table to the `Labels` property of audio datastore to associate the rows of the label table with the rows of the datastore. Call `countEachLabel` to determine the incidences of `containsMusic` and `instrument`.

```
labels = table(containsMusic,instrument);
ADS.Labels = labels;
```

```
containsMusicCount = countEachLabel(ADS,'TableVariable','containsMusic')
```

```
containsMusicCount=2x2 table
  containsMusic    Count
  _____    _____
      false         22
      true          7
```

```
instrumentCount = countEachLabel(ADS,'TableVariable','instrument');
```

Split the datastore ADS into two, based on whether the audio file contains music. ADS1 contains 5 of each label under the table variable containsMusic, and ADS2 contains the rest. Call countEachLabel to verify.

```
[ADS1,ADS2] = splitEachLabel(ADS,5,'TableVariable','containsMusic');
ADS1_containsMusicCount = countEachLabel(ADS1,'TableVariable','containsMusic')
```

```
ADS1_containsMusicCount=2x2 table
  containsMusic    Count
  _____    _____
      false         5
      true          5
```

```
ADS2_containsMusicCount = countEachLabel(ADS2,'TableVariable','containsMusic')
```

```
ADS2_containsMusicCount=2x2 table
  containsMusic    Count
  _____    _____
      false        17
      true          2
```

Split the datastore ADS into two, based on the type of instrument present in the audio file. ADS3 contains 2 of each label under the table variable instrument, and ADS4 contains the rest. Call countEachLabel to verify.

```
[ADS3,ADS4] = splitEachLabel(ADS,2,'TableVariable','instrument');
ADS3_instrumentCount = countEachLabel(ADS3,'TableVariable','instrument')
```

```
ADS3_instrumentCount=3x2 table
  instrument      Count
  _____      _____
  Drums           2
  Guitar          2
  Unknown         2
```

```
ADS4_instrumentCount = countEachLabel(ADS4,'TableVariable','instrument')
```

```
ADS4_instrumentCount=3x2 table
  instrument      Count
  _____      _____
  Drums           2
  Guitar          1
  Unknown         20
```

## Input Arguments

### ADS — Input audio datastore

audioDatastore object

Input audio datastore, specified as an `audioDatastore` object.

### p — Proportion of files to split

scalar in interval (0,1) | positive integer scalar

Proportion of files to split, specified as a scalar in the interval (0,1), or a positive integer scalar.

If `p` is in the interval (0,1), it represents the percentage of the files from each label to assign to `ADS1`. If `p` represents a percentage, and it does not result in a whole number, then `splitEachLabel` rounds down to the nearest whole number.

If `p` is an integer, it represents the absolute number of files from each label to assign to `ADS1`. When `p` represents a number of files, there must be at least `p` files associated with each label.

Data Types: double

**p1, . . . , pN — List of proportions**

scalars in interval (0,1) | positive integer scalars

List of proportions, specified as scalars in the interval (0,1) or positive integer scalars.

If the proportions are in the interval (0,1), they represent the percentage of the files from each label to assign to the output datastores. When the proportions represent percentages, their sum must be no more than 1.

If the proportions are integers, they indicate the absolute number of files from each label to assign to the output datastores. When the proportions represent numbers of files, there must be enough files associated with each label to satisfy each proportion.

Data Types: double

**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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `[ADS1, ADS2] = splitEachLabel(ADS, 0.5, 'Exclude', 'noisy')`

**Include — Labels to include**

categorical, logical, or numeric vector | cell array of character vectors | string array

Labels to include, specified as the comma-separated pair consisting of `'Include'` and a vector, cell array, or string array of label names with the same type as the `Labels` property. Each name must match one of the labels in the `Labels` property of the datastore.

This option cannot be used with the `'Exclude'` option.

**Exclude — Labels to exclude**

categorical, logical, or numeric vector | cell array of character vectors | string array

Labels to exclude, specified as the comma-separated pair consisting of `'Exclude'` and a vector, cell array, or string array of label names with the same type as the `Labels` property. Each name must match one of the labels in the `Labels` property of the datastore.

This option cannot be used with the `'Include'` option.

**TableVariable** — Label table variable name

char | string

Table variable name, specified as the comma-separated pair consisting of 'TableVariable' and a character vector or string. When the Labels property of the audio datastore ADS is a table, you must use 'TableVariable' to specify which label you are using to split.

Data Types: char | string

## Output Arguments

**[ADS1, ADS2]** — Output audio datastores

audioDatastore objects

Output audio datastores, returned as audioDatastore objects. ADS1 contains the specified proportion of files from each label in ADS, and ADS2 contains the remaining files.

**[ADS1, ..., ADSM]** — List of output audio datastores

audioDatastore objects

List of output audio datastores, returned as audioDatastore objects. The number of elements in the list is one more than the number of listed proportions. Each of the new datastores contains the proportion of each label in ADS defined by  $p_1, \dots, p_N$ . Any files left over are assigned to the  $M$ th datastore.

## See Also

audioDatastore | countEachLabel | subset

## Topics

"Speech Command Recognition Using Deep Learning"

"Speaker Identification Using Pitch and MFCC"

"Denoise Speech Using Deep Learning Networks"

"Music Genre Classification Using Wavelet Time Scattering"

**Introduced in R2018b**

## preview

Read first file from datastore for preview

## Syntax

```
data = preview(ADS)
```

## Description

`data = preview(ADS)` always reads the first file from ADS. `preview` does not affect the state of ADS.

## Examples

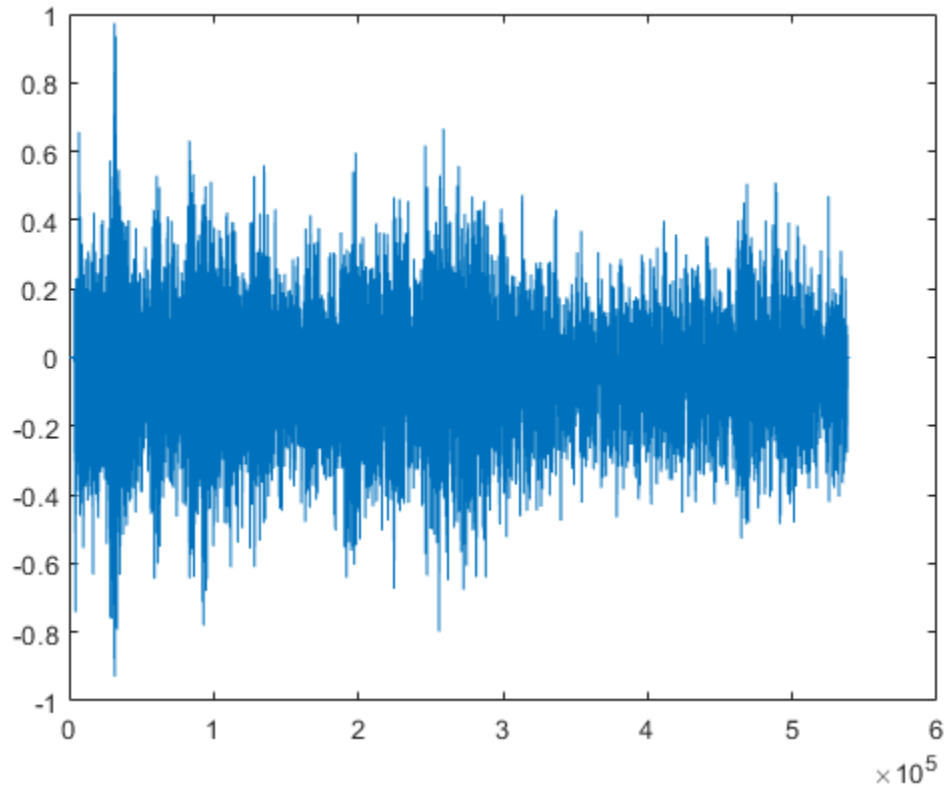
### Preview Data in Audio Datastore

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');  
ADS = audioDatastore(folder);
```

Preview the data in the audio datastore.

```
data = preview(ADS);  
plot(data)
```



### Input Arguments

**ADS — Audio datastore**  
audioDatastore object

Specify ADS as an audioDatastore object.



---

## Output Arguments

### **data** — Subset of data

array of audio samples

Subset of data, returned as an array of audio samples.

## See Also

`audioDatastore` | `hasdata`

## Topics

“Speech Command Recognition Using Deep Learning”

“Speaker Identification Using Pitch and MFCC”

“Denoise Speech Using Deep Learning Networks”

“Classify Gender Using LSTM Networks”

“Music Genre Classification Using Wavelet Time Scattering”

**Introduced in R2018b**

## subset

Create datastore with subset of files

## Syntax

```
ADSsubset = subset(ADS,indices)
```

## Description

`ADSsubset = subset(ADS,indices)` returns an audio datastore, `ADSsubset`, which contains a subset of the files in `ADS`.

## Examples

### Create Datastore with Subset Based on File Name

`subset` creates an audio datastore containing a subset of the files of the original datastore.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');  
ADS = audioDatastore(folder)
```

```
ADS =  
  audioDatastore with properties:
```

```
    Files: {  
            'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-1  
            'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channe  
            ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p  
            ... and 26 more  
          }
```

```
  AlternateFileSystemRoots: {}
```

```

OutputDataType: 'double'
Labels: {}

```

Create a logical vector indicating whether the file names in the audio datastore contain 'Guitar'.

```
fileContainsGuitar = cellfun(@(c)contains(c,'Guitar'),ADS.Files)
```

```
fileContainsGuitar = 29x1 logical array
```

```

0
0
0
0
0
0
0
0
0
0
:

```

Call `subset` with the audio datastore and the indices corresponding to the files you want create a new audio datastore from.

```
ADSsubset = subset(ADS,fileContainsGuitar)
```

```
ADSsubset =
audioDatastore with properties:
```

```

Files: {
    'B:\matlab\toolbox\audio\samples\RockGuitar-16-44p1-ster
    'B:\matlab\toolbox\audio\samples\RockGuitar-16-96-ster
    'B:\matlab\toolbox\audio\samples\SoftGuitar-44p1_mono-10
}
AlternateFileSystemRoots: {}
OutputDataType: 'double'
Labels: {}

```

### Create Datastore with Every Other File

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');  
ADS = audioDatastore(folder)
```

```
ADS =
```

```
  audioDatastore with properties:
```

```
      Files: {  
              'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12  
              'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channe  
              ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1  
              ... and 26 more  
            }  
  AlternateFileSystemRoots: {}  
      OutputDataType: 'double'  
      Labels: {}
```

Create an audio datastore containing every other file of the original datastore.

```
indices = 1:2:numel(ADS.Files);  
ADSsubset = subset(ADS,indices)
```

```
ADSsubset =
```

```
  audioDatastore with properties:
```

```
      Files: {  
              'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12  
              ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1  
              'B:\matlab\toolbox\audio\samples\Counting-16-44p1-mono-15  
              ... and 12 more  
            }  
  AlternateFileSystemRoots: {}  
      OutputDataType: 'double'  
      Labels: {}
```

## Input Arguments

### **ADS — Audio datastore**

audioDatastore object

Specify ADS as an audioDatastore object.

### **indices — Indices of files for subset**

vector of indices | logical vector

Specify indices as:

- A vector containing the indices of files to be included in ADSsubset.
- A logical vector the same length as the number of files in ADS. If specifying indices as a logical vector, true indicates that the corresponding files are included in ADSsubset.

Data Types: double | logical

## Output Arguments

### **ADSsubset — Subset of audio datastore**

audioDatastore object

Subset of audio datastore, returned as an audioDatastore object.

## See Also

audioDatastore | splitEachLabel

## Topics

“Speech Command Recognition Using Deep Learning”

“Speaker Identification Using Pitch and MFCC”

“Denoise Speech Using Deep Learning Networks”

“Classify Gender Using LSTM Networks”

“Music Genre Classification Using Wavelet Time Scattering”

**Introduced in R2018b**

# shuffle

Shuffle files in datastore

## Syntax

```
shuffledADS = shuffle(ADS)
```

## Description

`shuffledADS = shuffle(ADS)` creates a deep copy of the input datastore, `ADS`, and shuffles the files using `randperm`.

## Examples

### Shuffle Files

Create an `audioDatastore` object `ADS`. Shuffle the files to create a new datastore containing the same files in random order.

```
ADS = audioDatastore(fullfile(matlabroot, 'toolbox', 'audio', 'samples'))
```

```
ADS =
```

```
  audioDatastore with properties:
```

```
      Files: {  
              'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12  
              'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channe  
              ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p  
              ... and 26 more  
            }  
  AlternateFileSystemRoots: {}  
      OutputDataType: 'double'  
      Labels: {}
```

```
ADSshuffled = shuffle(ADS)

ADSshuffled =
    audioDatastore with properties:
        Files: {
            'B:\matlab\toolbox\audio\samples\SoftGuitar-44p1_mono-10r
            'B:\matlab\toolbox\audio\samples\Engine-16-44p1-stereo-20
            ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p
            ... and 26 more
        }
        AlternateFileSystemRoots: {}
        OutputDataType: 'double'
        Labels: {}
```

## Input Arguments

### ADS — Input audio datastore

audioDatastore object

Input audio datastore, specified as an audioDatastore object.

## Output Arguments

### shuffledADS — Shuffled audio datastore

audioDatastore object

Shuffled audio datastore, returned as an audioDatastore object containing randomly ordered files from ADS.

## See Also

audioDatastore

## Topics

“Speech Command Recognition Using Deep Learning”

“Speaker Identification Using Pitch and MFCC”



“Denoise Speech Using Deep Learning Networks”  
“Classify Gender Using LSTM Networks”  
“Music Genre Classification Using Wavelet Time Scattering”

**Introduced in R2018b**

## hasdata

Return true if there is more data in datastore

### Syntax

```
tf = hasdata(ADS)
```

### Description

`tf = hasdata(ADS)` returns logical 1 (`true`) if there is data available to read from the datastore specified by `ADS`. Otherwise, it returns logical 0 (`false`).

### Examples

#### Keep Reading While Datastore Has Data

`hasdata` returns a logical scalar indicating whether or not there is unread data in the datastore. You can use `audioDatastore` to read files sequentially until all data is read.

Specify the file path to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');
```

Create an audio datastore that points to the specified folder.

```
ADS = audioDatastore(folder);
```

While the datastore has unread data, read from the datastore.

```
while hasdata(ADS)
    data = read(ADS);
end
```

## Input Arguments

### **ADS — Audio datastore**

audioDatastore object

Specify ADS as an audioDatastore object.

## Output Arguments

### **tf — Indication if data is available to read**

true | false

Indication if data is available to read from the datastore, returned as true or false.

Data Types: logical

## See Also

audioDatastore | progress | read

## Topics

“Speech Command Recognition Using Deep Learning”

“Speaker Identification Using Pitch and MFCC”

“Denoise Speech Using Deep Learning Networks”

“Classify Gender Using LSTM Networks”

“Music Genre Classification Using Wavelet Time Scattering”

**Introduced in R2018b**

## reset

Reset datastore read pointer to start of data

## Syntax

```
reset(ADS)
```

## Description

`reset(ADS)` resets the datastore read pointer to the start of the data. Resetting allows re-reading from the same datastore.

## Examples

### Reset Audio Datastore to Initial State

Create an `audioDatastore` object `ADS`.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');  
ADS = audioDatastore(folder);
```

While the datastore has unread files, call `read` in a loop to read files sequentially.

```
while hasdata(ADS)  
    data = read(ADS);  
end
```

Reset the datastore to the state where no data has been read from it. Read the first file from the datastore.

```
reset(ADS)  
data = read(ADS);
```

---

## Input Arguments

### **ADS — Audio datastore**

audioDatastore object

Specify ADS as an audioDatastore object.

## See Also

audioDatastore

## Topics

“Speech Command Recognition Using Deep Learning”

“Speaker Identification Using Pitch and MFCC”

“Denoise Speech Using Deep Learning Networks”

“Classify Gender Using LSTM Networks”

“Music Genre Classification Using Wavelet Time Scattering”

**Introduced in R2018b**

## readall

Read all audio files from datastore

### Syntax

```
data = readall(ADS)
```

### Description

`data = readall(ADS)` reads all audio files from the datastore.

If all the data in the datastore does not fit in memory, then `readall` returns an error.

### Examples

#### Read All Data in Audio Datastore

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');  
ADS = audioDatastore(folder);
```

Read all the data in the datastore.

```
readall(ADS)
```

```
ans=29x1 cell  
  { 539648x1 double}  
  { 320512x4 double}  
  { 227497x1 double}  
  {   8000x1 double}  
  { 685056x1 double}  
  { 882688x2 double}  
  {  24000x1 double}
```

```
{1115760x2 double}
{1214832x2 double}
{ 263304x16 double}
{ 180224x1 double}
{ 32768x1 double}
{6076484x2 double}
{ 112893x1 double}
{ 913152x1 double}
{ 913152x1 double}
:
```

## Input Arguments

### **ADS — Audio datastore**

audioDatastore object

Specify ADS as an audioDatastore object.

## Output Arguments

### **data — All audio files in audio datastore**

cell array

All files in the audio datastore, returned as a cell array where each cell corresponds to a file.

## See Also

audioDatastore | read

## Topics

“Speech Command Recognition Using Deep Learning”

“Speaker Identification Using Pitch and MFCC”

“Denoise Speech Using Deep Learning Networks”

“Classify Gender Using LSTM Networks”

“Music Genre Classification Using Wavelet Time Scattering”

**Introduced in R2018b**



## read

Read next consecutive audio file

## Syntax

```
data = read(ADS)
[data,info] = read(ADS)
```

## Description

`data = read(ADS)` returns audio extracted from the datastore. Each subsequent call to the `read` function continues reading from the endpoint of the previous call.

`[data,info] = read(ADS)` also returns information about the extracted audio data.

## Examples

### Read Data in Audio Datastore

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');
ADS = audioDatastore(folder);
```

While the audio datastore has unread files, read consecutive files from the datastore. Use `progress` to monitor the fraction of files read.

```
while hasdata(ADS)
    data = read(ADS);
    fprintf('Fraction of files read: %.2f\n', progress(ADS))
end
```

```
Fraction of files read: 0.03
Fraction of files read: 0.07
```

```
Fraction of files read: 0.10
Fraction of files read: 0.14
Fraction of files read: 0.17
Fraction of files read: 0.21
Fraction of files read: 0.24
Fraction of files read: 0.28
Fraction of files read: 0.31
Fraction of files read: 0.34
Fraction of files read: 0.38
Fraction of files read: 0.41
Fraction of files read: 0.45
Fraction of files read: 0.48
Fraction of files read: 0.52
Fraction of files read: 0.55
Fraction of files read: 0.59
Fraction of files read: 0.62
Fraction of files read: 0.66
Fraction of files read: 0.69
Fraction of files read: 0.72
Fraction of files read: 0.76
Fraction of files read: 0.79
Fraction of files read: 0.83
Fraction of files read: 0.86
Fraction of files read: 0.90
Fraction of files read: 0.93
Fraction of files read: 0.97
Fraction of files read: 1.00
```

### Return Information About Data

Specify the file path to the audio samples you want to include in the audio datastore. In this example, the samples are located on a local desktop. Create an audio datastore that points to the specified folder.

```
folder = 'C:\Users\bhemmat\Desktop';
ADS = audioDatastore(folder, 'LabelSource', 'foldernames');
```

When you read data from the datastore, you can additionally return information about the data as a struct. The information struct contains the file name, any labels associated with the file, and the sample rate of the file.

```
[data,info] = read(ADS);
info

info =

    struct with fields:

        SampleRate: 44100
        FileName: 'C:\Users\bhemmat\Desktop\Turbine-16-44p1-mono-22secs.wav'
        Label: Desktop
```

## Input Arguments

### **ADS** — Audio datastore

audioDatastore object

Specify ADS as an audioDatastore object.

## Output Arguments

### **data** — Audio data

*M*-by-*N* matrix

Audio data, returned as a *M*-by-*N* matrix, where:

- *M* -- Total samples per channel in file.
- *N* -- Number of channels in file.

### **info** — Information about audio data

struct

Information about audio data, returned as a struct with the following fields:

- `FileName` -- Name of the current file.
- `Label` -- All labels of the file.
- `SampleRate` -- Sample rate of the file.

## See Also

audioDatastore | hasdata | readall

## Topics

“Speech Command Recognition Using Deep Learning”

“Speaker Identification Using Pitch and MFCC”

“Denoise Speech Using Deep Learning Networks”

“Classify Gender Using LSTM Networks”

“Music Genre Classification Using Wavelet Time Scattering”

**Introduced in R2018b**

# audioDatastore

Datastore for collection of audio files

## Description

Use an `audioDatastore` object to manage a collection of audio files, where each individual audio file fits in memory, but the entire collection of audio files does not necessarily fit.

## Creation

## Syntax

```
ADS = audioDatastore(location)
ADS = audioDatastore(location,Name,Value)
```

## Description

`ADS = audioDatastore(location)` creates a datastore `ADS` based on an audio file or collection of audio files in `location`.

`ADS = audioDatastore(location,Name,Value)` specifies additional parameters and properties for `ADS` using one or more name-value pair arguments.

## Input Arguments

### location — Files or folders to include in datastore

`path` | `DsFileSet` object

Files or folders included in the datastore, specified as a path or a `DsFileSet` object.

- `path` — Specify the path as a character vector, cell array of character vectors, string scalar, or a string array, containing the location of files or folders that are local or remote.

- Local files or folders — Specify `location` as a local path to files or folders. If the files are not in the current folder, then local path must specify full or relative paths. Files within subfolders of the specified folder are not automatically included in the datastore. You can use the wildcard character (\*) when specifying the local path. This character specifies that the datastore include all matching files or all files in the matching folders.
- Remote files or folders — Specify `location` to be the full paths of the files or folders as a uniform resource locator (URL) of the form `hdfs:///path_to_file`. For more information, see “Work with Remote Data” (MATLAB).
- `DsFileSet` object — You also can specify `location` as a `DsFileSet` object. For more information, see `matlab.io.datastore.DsFileSet`.

When `location` represents a folder, the datastore includes only supported file formats and ignores any other format. To specify a custom list of file extensions to include in your datastore, see the `FileExtensions` property.

Example: `'song.wav'`

Example: `'../dir/music/song.wav'`

Example: `{'C:\dir\music\song.wav', 'C:\dir\speech\english.mp3'}`

Example: `'C:\dir\music\*.ogg'`

Data Types: `char` | `string` | `cell`

### **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `ADS = audioDatastore('C:\dir\audiodata', 'FileExtensions', '.ogg')`

### **IncludeSubfolders — Subfolder inclusion flag**

`false` (default) | `true`

Subfolder inclusion flag, specified as the comma-separated pair consisting of `'IncludeSubfolders'` and `true` or `false`. Specify `true` to include all files and subfolders within each folder or `false` to include only the files within each folder.

If you do not specify `'IncludeSubfolders'`, then the default value is `false`.

Example: `'IncludeSubfolders',true`

Data Types: `logical | double`

### **LabelSource — Source providing label data**

`'none'` (default) | `'foldernames'`

Source providing label data, specified as the comma-separated pair consisting of `'LabelSource'` and `'none'` or `'foldernames'`. If `'none'` is specified, then the `Labels` property is empty. If `'foldernames'` is specified, then labels are assigned according to the folder names and stored in the `Labels` property. You can later modify the labels by accessing the `Labels` property directly.

Data Types: `char | string`

### **FileExtensions — Audio file extensions**

`character vector | cell array of character vectors | string scalar | string array`

Audio file extensions, specified as the comma-separated pair consisting of `'FileExtensions'` and a character vector, cell array of character vectors, string scalar, or string array. If you do not specify `'FileExtensions'`, then `audioDatastore` automatically includes all supported file types:

- `.wav`
- `.avi`
- `.aif`
- `.aifc`
- `.aiff`
- `.mp3`
- `.au`
- `.snd`
- `.mp4`
- `.m4a`
- `.flac`
- `.ogg`
- `.mov`

Example: `'FileExtensions','.wav'`

Example: `'FileExtensions',{'.mp3','.mp4'}`

Data Types: `char` | `cell` | `string`

In addition to these name-value pairs, you also can specify any of the properties on this page as name-value pairs, except for the `Files` property.

## Properties

### **Files** — Files included in datastore

`character vector` | `cell array of character vectors` | `string scalar` | `string array`

Files included in the datastore, specified as a character vector, cell array of character vectors, string scalar, or string array. Each character vector or string is a full path to a file. The `location` argument in the `audioDatastore` defines `Files` when the datastore is created.

Data Types: `char` | `cell` | `string`

### **Labels** — File labels

`categorical`, `logical`, or `numeric vector` | `cell array` | `string array` | `table`

File labels for the files in the datastore, specified as a vector, a cell array, a string array, or a table. The order of the labels in the array or table corresponds to the order of the associated files in the datastore.

If you specify `LabelSource` as `'foldernames'` when creating the `audioDatastore` object, then the label name for a file is the name of the folder containing it. If you do not specify `LabelSource` as `'foldernames'`, then `Labels` is an empty cell array or string array. If you change the `Files` property after the datastore is created, then the `Labels` property is not automatically updated to incorporate the added fields.

Data Types: `categorical` | `cell` | `logical` | `double` | `single` | `string` | `table`

### **OutputDataType** — Data type of output read

`'double'` (default) | `'native'`

Data type of the output, specified as `'double'` or `'native'`.

- `'double'` -- Double-precision normalized samples.
- `'native'` -- Native data type found in the file. Refer to `audioread` for more information about data types when `OutputDataType` is set to `native`.



The default value of this property is 'double'.

Data Types: char | string

### **AlternateFileSystemRoots — Alternate file system root paths**

string row vector | cell array of string vectors | cell array of character vectors

Alternate file system root paths, specified as a string row vector, a cell array of string vectors, or a cell array of character vectors. Use `AlternateFileSystemRoots` when you create a datastore on a local machine but must access and process data on another machine (possibly of a different operating system). Also, when processing data using Parallel Computing Toolbox and MATLAB Parallel Server™, and the data is stored on your local machines with a copy of the data available on different platform cloud or cluster machines, you must use `AlternateFileSystemRoots` to associate the root paths.

- To associate a set of root paths that are equivalent to one another, specify `AlternateFileSystemRoots` as a string vector. For example:

```
["Z:\datasets", "/mynetwork/datasets"]
```

- To associate multiple sets of root paths that are equivalent for the datastore, specify `AlternateFileSystemRoots` as a cell array containing multiple rows, where each row represents a set of equivalent root paths. Specify each row in the cell array as either a string vector or a cell array of character vectors. For example:

- Specify `AlternateFileSystemRoots` as a cell array of string vectors.

```
{["Z:\datasets", "/mynetwork/datasets"]; ...
 ["Y:\datasets", "/mynetwork2/datasets", "S:\datasets"]}
```

- Alternatively, specify `AlternateFileSystemRoots` as a cell array of cell arrays of character vectors.

```
{{'Z:\datasets', '/mynetwork/datasets'}; ...
 {'Y:\datasets', '/mynetwork2/datasets', 'S:\datasets'}}
```

The value of `AlternateFileSystemRoots` must satisfy these conditions:

- Contains one or more rows, where each row specifies a set of equivalent root paths.
- Each row specifies multiple root paths, and each root path must contain at least two characters.
- Root paths are unique and are not subfolders of one another.
- Contains at least one root path entry that points to the location of the files.

Data Types: char | cell | string

## Object Functions

read	Read next consecutive audio file
readall	Read all audio files from datastore
reset	Reset datastore read pointer to start of data
hasdata	Return true if there is more data in datastore
shuffle	Shuffle files in datastore
subset	Create datastore with subset of files
preview	Read first file from datastore for preview
progress	Fraction of files read
splitEachLabel	Splits datastore according to specified label proportions
countEachLabel	Count number of unique labels
partition	Partition datastore and return on partitioned portion
numpartitions	Return estimate for reasonable number of partitions for parallel processing
combine	Combine data from multiple datastores
transform	Transform audio datastore

## Examples

### Create Audio Datastore

Specify the file path to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');
```

Create an audio datastore that points to the specified folder.

```
ADS = audioDatastore(folder)
```

```
ADS =  
  audioDatastore with properties:
```

```
    Files: {  
        'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-128k.wav'  
        'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channel-128k.wav'  
        '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-128k.wav'  
    }
```

```

        ... and 26 more
    }
    AlternateFileSystemRoots: {}
    OutputDataType: 'double'
    Labels: {}

```

## Specify File Extensions to Include

Specify the file path to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');
```

Create an audio datastore that points to the .ogg files in the specified folder.

```
ADS = audioDatastore(folder, 'FileExtension', '.ogg')
```

```
ADS =
```

```
audioDatastore with properties:
```

```

        Files: {
            'B:\matlab\toolbox\audio\samples\SoftGuitar-44p1_mono-10
        }
    AlternateFileSystemRoots: {}
    OutputDataType: 'double'
    Labels: {}

```

## See Also

[datastore](#) | [mapreduce](#) | [tall](#)

## Topics

“Speech Command Recognition Using Deep Learning”

“Speaker Identification Using Pitch and MFCC”

“Denoise Speech Using Deep Learning Networks”

“Classify Gender Using LSTM Networks”

“Music Genre Classification Using Wavelet Time Scattering”

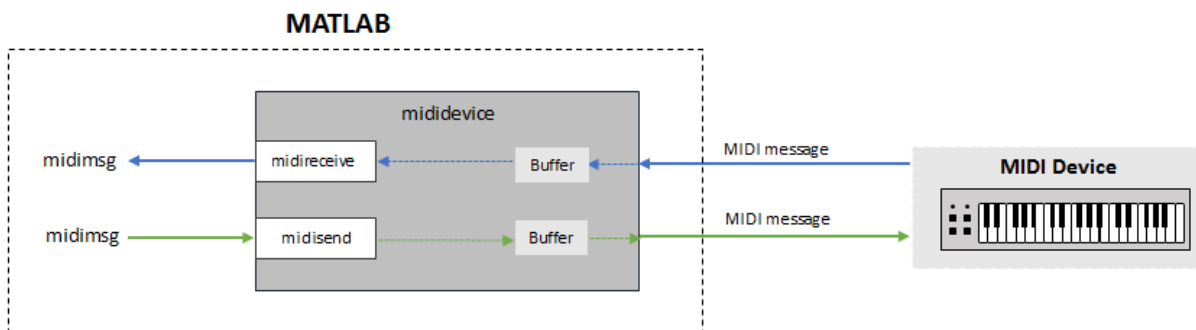
**Introduced in R2018b**

# midimsg

Create MIDI message

## Description

Create a MIDI message in MATLAB using `midimsg`. Create a MIDI device interface using `mididevice`. Send and receive messages using `midisend` and `midireceive`. When you create a MIDI message, you specify it as a MIDI message type.



For a tutorial on MIDI messages and interfacing with MIDI devices, see “MIDI Device Interface”.

## Creation

## Syntax

```
msg = midimsg('Note',channel,note,velocity,duration,timestamp)
msg = midimsg('NoteOn',channel,note,velocity,timestamp)
msg = midimsg('NoteOff',channel,note,velocity,timestamp)
msg = midimsg('ControlChange',channel,ccnumber,ccvalue,timestamp)
msg = midimsg('ProgramChange',channel,program,timestamp)
msg = midimsg('SystemExclusive',bytes,timestamp)
```

```
msg = midimsg('SystemExclusive',timestamp)
msg = midimsg('Data',bytes,timestamp)
msg = midimsg('EOX',timestamp)
msg = midimsg('TimingClock',timestamp)
msg = midimsg('Start',timestamp)
msg = midimsg('Continue',timestamp)
msg = midimsg('Stop',timestamp)
msg = midimsg('ActiveSensing',timestamp)
msg = midimsg('SystemReset',timestamp)
msg = midimsg('TuneRequest',timestamp)
msg = midimsg('MIDITimeCodeQuarterFrame',seq,value,timestamp)
msg = midimsg('SongPositionPointer',position,timestamp)
msg = midimsg('SongSelect',song,timestamp)
msg = midimsg('AllSoundOff',channel,timestamp)
msg = midimsg('ResetAllControllers',channel,timestamp)
msg = midimsg('LocalControl',channel,localcontrol,timestamp)
msg = midimsg('PolyOn',channel,timestamp)
msg = midimsg('MonoOn',channel,monoChannels,timestamp)
msg = midimsg('OmniOn',channel,timestamp)
msg = midimsg('OmniOff',channel,timestamp)
msg = midimsg('AllNotesOff',channel,timestamp)
msg = midimsg('PolyKeyPressure',channel,note,pressure,timestamp)
msg = midimsg('ChannelPressure',channel,pressure,timestamp)
msg = midimsg('PitchBend',channel,change,timestamp)
msg = midimsg
msg = midimsg(size)
msg = midimsg(0)
```

## Description

`msg = midimsg('Note',channel,note,velocity,duration,timestamp)` returns two MIDI messages: `NoteOn` and `NoteOff`, with specified `Channel`, `Note`, `Velocity`, and `Timestamp` properties. The `Timestamp` property of the `NoteOff` message is determined as the `Timestamp` property of the `NoteOn` message plus the duration.

`msg = midimsg('NoteOn',channel,note,velocity,timestamp)` returns a `NoteOn` `midimsg`, with specified `Channel`, `Note`, `Velocity`, and `Timestamp` properties.

`msg = midimsg('NoteOff', channel, note, velocity, timestamp)` returns a NoteOff midimsg, with specified Channel, Note, Velocity, and Timestamp properties.

`msg = midimsg('ControlChange', channel, ccnumber, ccvalue, timestamp)` returns a ControlChange midimsg, with specified Channel, CCNumber, CCValue, and Timestamp properties.

`msg = midimsg('ProgramChange', channel, program, timestamp)` returns a ProgramChange midimsg, with specified Channel, Program, and Timestamp properties.

`msg = midimsg('SystemExclusive', bytes, timestamp)` returns a complete SystemExclusive message sequence, with specified Timestamp property.

`msg = midimsg('SystemExclusive', timestamp)` returns a SystemExclusive midimsg, with specified Timestamp property.

`msg = midimsg('Data', bytes, timestamp)` returns a Data midimsg for use in a System Exclusive message, with specified MsgBytes and Timestamp properties. `bytes` is specified as a scalar, vector, or multi-dimensional array of elements. Each element of bytes must be in the range [0,127].

`msg = midimsg('EOX', timestamp)` returns an EOX midimsg, with specified Timestamp property.

`msg = midimsg('TimingClock', timestamp)` returns a TimingClock midimsg, with specified Timestamp property.

`msg = midimsg('Start', timestamp)` returns a Start midimsg, with specified Timestamp property.

`msg = midimsg('Continue', timestamp)` returns a Continue midimsg, with specified Timestamp property.

`msg = midimsg('Stop', timestamp)` returns a Stop midimsg, with specified Timestamp property.

`msg = midimsg('ActiveSensing', timestamp)` returns a ActiveSensing midimsg, with specified Timestamp property.

`msg = midimsg('SystemReset', timestamp)` returns a SystemReset midimsg, with specified Timestamp property.

`msg = midimsg('TuneRequest', timestamp)` returns a `TuneRequest` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('MIDITimeCodeQuarterFrame', seq, value, timestamp)` returns a `MIDITimeCodeQuarterFrame` `midimsg`, with specified `TimeCodeSequence`, `TimeCodeValue`, and `Timestamp` properties.

`msg = midimsg('SongPositionPointer', position, timestamp)` returns a `SongPositionPointer` `midimsg`, with specified `SongPosition` and `Timestamp` properties.

`msg = midimsg('SongSelect', song, timestamp)` returns a `SongSelect` `midimsg`, with specified `Song` and `Timestamp` properties.

`msg = midimsg('AllSoundOff', channel, timestamp)` returns a `AllSoundOff` `midimsg`, with specified `Channel` and `Timestamp` properties.

`msg = midimsg('ResetAllControllers', channel, timestamp)` returns a `ResetAllControllers` `midimsg`, with specified `Channel` and `Timestamp` properties.

`msg = midimsg('LocalControl', channel, localcontrol, timestamp)` returns a `LocalControl` `midimsg`, with specified `Channel`, `LocalControl`, and `Timestamp` properties.

`msg = midimsg('PolyOn', channel, timestamp)` returns a `PolyOn` `midimsg`, with specified `Channel` and `Timestamp` properties.

`msg = midimsg('MonoOn', channel, monoChannels, timestamp)` returns a `MonoOn` `midimsg`, with specified `Channel`, `MonoChannels`, and `Timestamp` properties.

`msg = midimsg('OmniOn', channel, timestamp)` returns an `OmniOn` `midimsg`, with specified `Channel` and `Timestamp` properties.

`msg = midimsg('OmniOff', channel, timestamp)` returns an `OmniOff` `midimsg`, with specified `Channel` and `Timestamp` properties.

`msg = midimsg('AllNotesOff', channel, timestamp)` returns an `AllNotesOff` `midimsg`, with specified `Channel` and `Timestamp` properties.

`msg = midimsg('PolyKeyPressure', channel, note, pressure, timestamp)` returns a `PolyKeyPressure` `midimsg`, with specified `Channel`, `Note`, `Pressure`, and `Timestamp` properties.



`msg = midimsg('ChannelPressure', channel, pressure, timestamp)` returns a ChannelPressure midimsg, with specified Channel, Pressure, and Timestamp properties.

`msg = midimsg('PitchBend', channel, change, timestamp)` returns a PitchBend midimsg, with specified Channel, PitchChange, and Timestamp properties.

`msg = midimsg` returns a scalar midimsg with all zero bytes. All zero bytes indicates a MIDI message with Type set to Data.

`msg = midimsg(size)` returns a midimsg array of size with all zero bytes.

`msg = midimsg(0)` returns an empty midimsg.

---

**Note** If `timestamp` is listed as an argument, it is optional and defaults to zero. The exception is the 'SystemExclusive', bytes, timestamp syntax, in which case the timestamp argument is required.

---

## Properties

### Type — Type of MIDI message

NoteOn | NoteOff | ControlChange | ProgramChange | SystemExclusive | Data | EOX | ...

This property is read-only.

Type of MIDI message, returned as one of the following midimsgtype enumeration values:

NoteOn	Data	Stop	SongPositionPointer	PolyOn	PolyKeyPressure
NoteOff	EOX	ActiveSensing	SongSelect	MonoOn	ChannelPressure
ControlChange	TimingClock	SystemReset	AllSoundOff	OmniOn	PitchBendChange
ProgramChange	Start	TuneRequest	ResetAllControllers	OmniOff	Undefined

SystemExclusive	Continue	MIDITimeCodeQuarterFrame	LocalControl	AllNotesOff	
-----------------	----------	--------------------------	--------------	-------------	--

You can specify the type of MIDI message during creation as a character vector, string, or member of the `midimsgtype` enumeration.

For example, the following create equivalent MIDI messages:

- `midimsg('SongPositionPointer',1)`
- `midimsg("SongPositionPointer",1)`
- `midimsg(midimsgtype.SongPositionPointer,1)`

**NumMsgBytes – Number of bytes in MIDI message**

scalar | vector | array

This property is read-only.

Number of bytes in the MIDI message, returned as a scalar, vector, or array the same size as `msg`.

Data Types: `double`

**MsgBytes – Actual bytes of constructed MIDI message (decimal)**

scalar | vector | array

This property is read-only.

Actual bytes of the constructed MIDI message in decimal, returned as a scalar, vector, or array the same size as `msg`.

Data Types: `uint8`

**Timestamp – Location in time for MIDI message**

scalar | vector | array

Location in time for the MIDI message, specified as a scalar, vector, or array the same size as `msg`.

You can specify the timestamp as any numeric value. However, the timestamp is always stored and returned as type `double`.

For more on how MIDI timestamps are implemented in Audio Toolbox, see “MIDI Message Timing”.

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

**Channel — MIDI channel to which message is addressed**

integer in the range [1,16]

MIDI channel to which message is addressed, specified as an integer in the range [1,16].

**Dependencies**

This property is valid only for `NoteOn`, `NoteOff`, `PolyKeyPressure`, `AllSoundOff`, `ResetAllControllers`, `LocalControl`, `AllNotesOff`, `OmniOn`, `OmniOff`, `MonoOn`, `PolyOn`, `ControlChange`, `ProgramChange`, `ChannelPressure`, and `PitchBend` `midimsg` objects.

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

**Note — MIDI note number**

integer in the range [0,127]

MIDI note number, specified as an integer in the range [0,127]. The MIDI specification defines note number 60 as Middle C, and all other notes are relative. MIDI devices and software define the mapping between a note and a MIDI note number. If Middle C is arbitrarily assumed to be C5 for the target MIDI hardware or software, the following table maps between MIDI note numbers and notes:

	Note Numbers											
Octave	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				

**Dependencies**

This property is valid only for NoteOn, NoteOff, and PolyKeyPressure midimsg objects.

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

**Velocity — Velocity of MIDI message**

integer in the range [0,127]

Velocity of MIDI message, specified as a scalar integer in the range [0,127]. Velocity describes how fast, or "hard," a note is played. A higher number corresponds to faster velocity.

**Dependencies**

This property is valid only for NoteOn and NoteOff midimsg objects.

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

**KeyPressure — Key pressure**

integer in the range [0,127]

Key pressure, specified as a scalar integer in the range [0,127]. Key pressure applies *aftertouch* to an individual note. For example, on a keyboard, key pressure describes the

pressure applied to a key after that key has been struck (after a `NoteOn` message is sent). You can use `KeyPressure` to add expression to held notes.

#### **Dependencies**

This property is valid only for `PolyKeyPressure` midimsg objects.

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

#### **LocalControl — Enable local control**

`true` | `false`

Enable local control, specified as `true` or `false`. When local control is set to `false`, all devices on a given channel respond only to data received over MIDI.

#### **Dependencies**

This property is valid only for `LocalControl` midimsg objects.

Data Types: `logical`

#### **MonoChannels — Channels for MonoOn messages**

integer in the range [0,16]

Channels for `MonoOn` messages, specified as a scalar integer in the range [0,16].

#### **Dependencies**

This property is valid only for `MonoOn` midimsg objects.

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

#### **CCNumber — Control change number**

integer in the range [0,119]

Control change number, specified as an integer in the range [0,119].

#### **Dependencies**

This property is valid only for `ControlChange` midimsg objects.

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

### **CCValue — Control change value**

integer in the range [0,127]

Control change value, specified as an integer in the range [0,127].

#### **Dependencies**

This property is valid only for `ControlChange` midimsg objects.

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

### **Program — Program number to switch to**

integer in the range [0,127]

Program number to switch to, specified as an integer in the range [0,127].

#### **Dependencies**

This property is valid only for `ProgramChange` midimsg objects.

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

### **ChannelPressure — Channel pressure**

integer in the range [0,127]

Channel pressure, specified as an integer in the range [0,127]. Key pressure applies *aftertouch* to all notes in a channel.

#### **Dependencies**

This property is valid only for `ChannelPressure` midimsg objects.

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

### **PitchChange — Amount of pitch change to apply**

integer in the range [0,16383]

Amount of pitch change to apply, specified as an integer in the range [0,16383]. The center position (no effect) is 8192. Sensitivity is a function of the receiver.

#### **Dependencies**

This property is valid only for `PitchBend` midimsg objects.

---

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

**TimeCodeSequence — Sequence number**

integer in the range [0,7]

Sequence number, specified as an integer in the range [0,7].

**Dependencies**

This property is valid only for MIDITimeCodeQuarterFrame midimsg objects.

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

**TimeCodeValue — Time code value**

integer in the range [0,15]

Time code value, specified as an integer in the range [0,15].

**Dependencies**

This property is valid only for MIDITimeCodeQuarterFrame midimsg objects.

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

**SongPosition — Position in song to go to**

integer in the range [0,16383]

Position in song to go to, specified as an integer in the range [0,16383].

**Dependencies**

This property is valid only for SongPositionPointer midimsg objects.

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

**Song — Song number to switch to**

integer in the range [0,127]

Song number to switch to, specified as an integer in the range [0,127].

### Dependencies

This property is valid only for `SongSelect` midimsg objects.

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

## Examples

### Create Note Messages

You can create MIDI note messages using the `NoteOn` and `NoteOff` midimsg objects. A `NoteOn` message indicates that a note should begin playing. A `NoteOff` message indicates that a note should stop playing. Alternatively, you can send a second `NoteOn` message with velocity set to zero to indicate that the note should stop playing. The Audio Toolbox® provides a convenience syntax to create pairs of note on and note off messages.

Create a pair of MIDI messages to indicate a Note On and Note Off sequence using the `Note` convenience syntax. Specify that the note starts after one second, and has a duration of two seconds.

```
channel = 1;
note = 60;
velocity = 64;
duration = 2;
timestamp = 1;
msgs = midimsg('Note',channel,note,velocity,duration,timestamp)

msgs=2×1 object
MIDI message:
  NoteOn          Channel: 1  Note: 60  Velocity: 64  Timestamp: 1  [ 90 3C 40 ]
  NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 3  [ 90 3C 00 ]
```

Two midimsg objects are created and returned as an array. The `Note` syntax returns the note off message as a `NoteOn` midimsg object with `Velocity` set to zero.

To create Note On and Note Off messages separately, create two `NoteOn` messages and concatenate them.

```
msgs = [midimsg('NoteOn',channel,note,velocity,timestamp), ...
        midimsg('NoteOn',channel,note,0,3)]
```



```

msgs=1x2 object
MIDI message:
  NoteOn          Channel: 1  Note: 60  Velocity: 64  Timestamp: 1  [ 90 3C 40 ]
  NoteOff         Channel: 1  Note: 60  Velocity: 0   Timestamp: 3  [ 90 3C 00 ]

```

You can also specify the Note Off using a `NoteOff` midimsg object. Using the `NoteOff` syntax enables you to specify a release velocity.

```

msgs = [midimsg('NoteOn',channel,note,velocity,timestamp), ...
        midimsg('NoteOff',channel,note,velocity,3)]

```

```

msgs=1x2 object
MIDI message:
  NoteOn          Channel: 1  Note: 60  Velocity: 64  Timestamp: 1  [ 90 3C 40 ]
  NoteOff         Channel: 1  Note: 60  Velocity: 64  Timestamp: 3  [ 80 3C 40 ]

```

## Control Change Messages for Control Surfaces

To create a control change message, specify the `midimsg` Type as `ControlChange` and set the required parameters: `Channel`, `CCNumber`, and `CCValue`. To determine the channel and control number assigned to your MIDI control surface, use `midid`. Enter `midid` at the Command Prompt and then move the control you want to identify.

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

```

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

```

`midid` returns the control change number and channel as a single number according to the following formula:  $ccInfo = (Channel * 1000 + CCNumber)$ . Define a MIDI Control Change message to move the identified controller. Your MIDI Control Surface must be bidirectional to receive Control Change messages.

```

channel = floor(ccInfo/1000);
ccnumber = ccInfo - channel*1000;
ccvalue = 1;
msg = midimsg('ControlChange',channel,ccnumber,ccvalue)

```

```
msg =  
    MIDI message:  
    ControlChange Channel: 1 CCNumber: 16 CCValue: 1 Timestamp: 0 [ B0 10 01 ]
```

Create a `mididevice` object using the `deviceName` identified using `midid`. Send the MIDI message to your device.

```
device = mididevice(deviceName);  
midisend(device,msg);
```

### Create a Program Change Message

Program Change messages, sometimes called "patch change" messages, specify how notes are interpreted. For example, a Program Change message can specify the instrument being played. To create a `ProgramChange` `midimsg` object, specify the `midimsg` type as `ProgramChange`, and the required property values: `Channel` and `Program`.

```
channel = 4;  
program = 7;  
msg = midimsg('ProgramChange',channel,program)
```

```
msg =  
    MIDI message:  
    ProgramChange Channel: 4 Program: 7 Timestamp: 0 [ C3 07 ]
```

### Create a System Exclusive Message

System Exclusive messages are defined by a sequence of `midimsg` objects: `SystemExclusive`, `Data`, and `E0X`. To create a System Exclusive sequence, specify the `SystemExclusive` `midimsg` type during creation and then specify the bytes of the message. This syntax requires a timestamp.

```
bytes = [0 1 2];  
timestamp = 0;  
msg = midimsg('SystemExclusive',bytes,timestamp)
```

```
msg=3x1 object  
    MIDI message:
```

```

SystemExclusive Timestamp: 0 [ F0 ]
Data            Timestamp: 0 [ 00 01 02 ]
EOX            Timestamp: 0 [ F7 ]

```

You can also create the `SystemExclusive`, `Data`, and `EOX` `midimsg` objects individually. For example, the following `midimsg` array is the same as the preceding.

```

msg = [midimsg('SystemExclusive',timestamp), ...
       midimsg('Data',bytes,timestamp), ...
       midimsg('EOX',timestamp)]

msg=1x3 object
MIDI message:
SystemExclusive Timestamp: 0 [ F0 ]
Data            Timestamp: 0 [ 00 01 02 ]
EOX            Timestamp: 0 [ F7 ]

```

### Create a Scalar Default MIDI Message

The default MIDI message is a scalar with all zero bytes, and Type is `Data`.

```

msg = midimsg

msg =
MIDI message:
Data            Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]

```

### Preallocate Array of MIDI Messages

You can create a MIDI message array by specifying the size by a scalar or row vector.

If you specify the size as a scalar  $M$ , `midimsg` returns an  $M$ -by- $M$  array with all zero bytes.

```

msg = midimsg(2)

msg=2x2 object
MIDI message:

```

```
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
```

An array of MIDI messages is always displayed vertically in order of their linear indexing. You can refer to individual elements of the array by specifying its position in each dimension, or by its linear index. For example, change the `Timestamp` of the third element from 0 to 2 using `linear indexing`, and then from 2 to 3 using `first dimensional indexing`.

```
msg(3).Timestamp = 2
```

```
msg=2x2 object
MIDI message:
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
Data          Timestamp: 2 [ 00 00 00 00 00 00 00 00 ]
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
```

```
msg(1,2).Timestamp = 3
```

```
msg=2x2 object
MIDI message:
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
Data          Timestamp: 3 [ 00 00 00 00 00 00 00 00 ]
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
```

You can also specify nonsymmetric arrays. If you specify the size as a row vector of two or more elements, `midimsg` returns an  $M$ -by- $N$ -by-...- $X$  multidimensional array. For example, to specify a three dimensional array with each dimension having a different number of elements, specify the size as a row vector of three elements.

```
msg = midimsg([2,1,3])
```

```
msg =
MIDI message:
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
```

```

Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]
Data          Timestamp: 0 [ 00 00 00 00 00 00 00 00 ]

size(msg)
ans = 1×3
     2     1     3

```

### Create Empty MIDI Message

```

msg = midimsg(0)
msg =
    empty MIDI message array

```

### Manipulate Array of MIDI Messages

In this example, you create an array of MIDI messages, and then index into the array in a loop to define a melody.

Create a 22-by-1 array of MIDI messages with all zero data.

```
msgArray = midimsg([22,1]);
```

To create a melody, create MIDI NoteOn and NoteOff messages by indexing in a loop. Display the result.

```

melody = [60,65,60,57,55,53,60,65,60,67,60];
for i = 1:numel(melody)
    idx = (2*i-1):(2*i);
    msgArray(idx) = midimsg('Note',1,melody(i),50,0.5,i);
end
msgArray

msgArray=22×1 object
MIDI message:
NoteOn          Channel: 1 Note: 60 Velocity: 50 Timestamp: 1 [ 90 3C 32 ]

```

```

NoteOn      Channel: 1 Note: 60 Velocity: 0   Timestamp: 1.5 [ 90 3C 00 ]
NoteOn      Channel: 1 Note: 65 Velocity: 50  Timestamp: 2   [ 90 41 32 ]
NoteOn      Channel: 1 Note: 65 Velocity: 0   Timestamp: 2.5 [ 90 41 00 ]
NoteOn      Channel: 1 Note: 60 Velocity: 50  Timestamp: 3   [ 90 3C 32 ]
NoteOn      Channel: 1 Note: 60 Velocity: 0   Timestamp: 3.5 [ 90 3C 00 ]
NoteOn      Channel: 1 Note: 57 Velocity: 50  Timestamp: 4   [ 90 39 32 ]
NoteOn      Channel: 1 Note: 57 Velocity: 0   Timestamp: 4.5 [ 90 39 00 ]
NoteOn      Channel: 1 Note: 55 Velocity: 50  Timestamp: 5   [ 90 37 32 ]
NoteOn      Channel: 1 Note: 55 Velocity: 0   Timestamp: 5.5 [ 90 37 00 ]
NoteOn      Channel: 1 Note: 53 Velocity: 50  Timestamp: 6   [ 90 35 32 ]
NoteOn      Channel: 1 Note: 53 Velocity: 0   Timestamp: 6.5 [ 90 35 00 ]
NoteOn      Channel: 1 Note: 60 Velocity: 50  Timestamp: 7   [ 90 3C 32 ]
NoteOn      Channel: 1 Note: 60 Velocity: 0   Timestamp: 7.5 [ 90 3C 00 ]
NoteOn      Channel: 1 Note: 65 Velocity: 50  Timestamp: 8   [ 90 41 32 ]
NoteOn      Channel: 1 Note: 65 Velocity: 0   Timestamp: 8.5 [ 90 41 00 ]
:

```

The order of the MIDI messages in the array is only important for readability. When you send MIDI messages using a `mididevice` object, the `mididevice` object reorders your MIDI messages according to their timestamps and sends them in chronological order. Create a `PitchBend` MIDI message to bend the fourth note downward and add it to the MIDI message array. For readability, sort the MIDI message array by `Timestamp`.

```

msg = midimsg('PitchBend',1,7192,4.01);
msgArray = [msgArray;msg]

```

```

msgArray=23x1 object

```

```

MIDI message:

```

```

NoteOn      Channel: 1 Note: 60 Velocity: 50  Timestamp: 1   [ 90 3C 32 ]
NoteOn      Channel: 1 Note: 60 Velocity: 0   Timestamp: 1.5 [ 90 3C 00 ]
NoteOn      Channel: 1 Note: 65 Velocity: 50  Timestamp: 2   [ 90 41 32 ]
NoteOn      Channel: 1 Note: 65 Velocity: 0   Timestamp: 2.5 [ 90 41 00 ]
NoteOn      Channel: 1 Note: 60 Velocity: 50  Timestamp: 3   [ 90 3C 32 ]
NoteOn      Channel: 1 Note: 60 Velocity: 0   Timestamp: 3.5 [ 90 3C 00 ]
NoteOn      Channel: 1 Note: 57 Velocity: 50  Timestamp: 4   [ 90 39 32 ]
NoteOn      Channel: 1 Note: 57 Velocity: 0   Timestamp: 4.5 [ 90 39 00 ]
NoteOn      Channel: 1 Note: 55 Velocity: 50  Timestamp: 5   [ 90 37 32 ]
NoteOn      Channel: 1 Note: 55 Velocity: 0   Timestamp: 5.5 [ 90 37 00 ]
NoteOn      Channel: 1 Note: 53 Velocity: 50  Timestamp: 6   [ 90 35 32 ]
NoteOn      Channel: 1 Note: 53 Velocity: 0   Timestamp: 6.5 [ 90 35 00 ]
NoteOn      Channel: 1 Note: 60 Velocity: 50  Timestamp: 7   [ 90 3C 32 ]
NoteOn      Channel: 1 Note: 60 Velocity: 0   Timestamp: 7.5 [ 90 3C 00 ]
NoteOn      Channel: 1 Note: 65 Velocity: 50  Timestamp: 8   [ 90 41 32 ]
NoteOn      Channel: 1 Note: 65 Velocity: 0   Timestamp: 8.5 [ 90 41 00 ]

```

```

:

timeStamps = [msgArray.Timestamp];
[~,idx] = sort(timeStamps);

msgArray = msgArray(idx)

msgArray=23x1 object
MIDI message:
NoteOn          Channel: 1 Note: 60 Velocity: 50 Timestamp: 1 [ 90 3C 32 ]
NoteOn          Channel: 1 Note: 60 Velocity: 0  Timestamp: 1.5 [ 90 3C 00 ]
NoteOn          Channel: 1 Note: 65 Velocity: 50 Timestamp: 2 [ 90 41 32 ]
NoteOn          Channel: 1 Note: 65 Velocity: 0  Timestamp: 2.5 [ 90 41 00 ]
NoteOn          Channel: 1 Note: 60 Velocity: 50 Timestamp: 3 [ 90 3C 32 ]
NoteOn          Channel: 1 Note: 60 Velocity: 0  Timestamp: 3.5 [ 90 3C 00 ]
NoteOn          Channel: 1 Note: 57 Velocity: 50 Timestamp: 4 [ 90 39 32 ]
PitchBend       Channel: 1 PitchChange: 7192 Timestamp: 4.01 [ E0 18 38 ]
NoteOn          Channel: 1 Note: 57 Velocity: 0  Timestamp: 4.5 [ 90 39 00 ]
NoteOn          Channel: 1 Note: 55 Velocity: 50 Timestamp: 5 [ 90 37 32 ]
NoteOn          Channel: 1 Note: 55 Velocity: 0  Timestamp: 5.5 [ 90 37 00 ]
NoteOn          Channel: 1 Note: 53 Velocity: 50 Timestamp: 6 [ 90 35 32 ]
NoteOn          Channel: 1 Note: 53 Velocity: 0  Timestamp: 6.5 [ 90 35 00 ]
NoteOn          Channel: 1 Note: 60 Velocity: 50 Timestamp: 7 [ 90 3C 32 ]
NoteOn          Channel: 1 Note: 60 Velocity: 0  Timestamp: 7.5 [ 90 3C 00 ]
NoteOn          Channel: 1 Note: 65 Velocity: 50 Timestamp: 8 [ 90 41 32 ]
:

```

## See Also

mididevice | midireceive | midisend

## Topics

“MIDI Device Interface”

## External Websites

MIDI Manufacturers Association

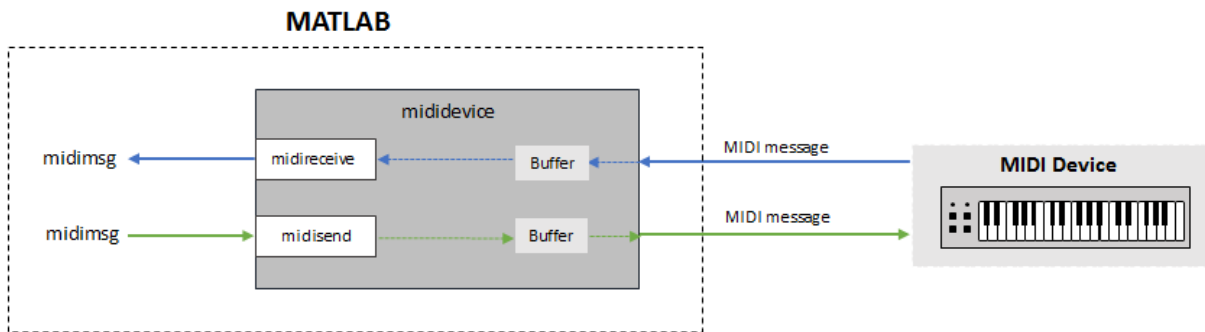
**Introduced in R2018a**

# mididevice

Send and receive MIDI messages

## Description

Interface to a MIDI device in MATLAB using `mididevice`. Package MIDI messages using `midimsg`. Send and receive messages using `midisend` and `midireceive`. Use `mididevinfo` to query your system for available MIDI devices.



For a tutorial on interfacing with MIDI devices, see “MIDI Device Interface”.

## Creation

## Syntax

```
device = mididevice(deviceNameOrID)
device = mididevice('Input',inDeviceNameOrID)
device = mididevice('Output',outDeviceNameOrID)
device = mididevice('Input',inDeviceNameOrID,
'Output',outDeviceNameOrID)
```



## Description

`device = mididevice(deviceNameOrID)` returns an interface to the MIDI device specified by `deviceNameOrID`. If the MIDI device supports MIDI in and MIDI out, then `device` also supports MIDI in and MIDI out.

`device = mididevice('Input', inDeviceNameOrID)` returns an input interface to the MIDI input device, `inDeviceNameOrID`.

`device = mididevice('Output', outDeviceNameOrID)` returns an output interface to the MIDI output device, `outDeviceNameOrID`.

`device = mididevice('Input', inDeviceNameOrID, 'Output', outDeviceNameOrID)` returns a MIDI I/O interface, where input is received from `inDeviceNameOrID` and output is sent to `outDeviceNameOrID`.

## Properties

### **Input** — Input device name associated with `mididevice`

empty char array (default)

This property is read-only.

Input device name attached to your `mididevice` object, returned as a character array.

Input is set during the creation of the `mididevice` object and cannot be modified later.

Data Types: char

### **Output** — Output device name associated with `mididevice`

empty char array (default)

This property is read-only.

Output device name attached to your `mididevice` object, returned as a character array

Output is set during the creation of the `mididevice` object and cannot be modified later.

Data Types: char

### **InputID** — Input device ID associated with `mididevice`

-1 (default)

This property is read-only.

Unique MIDI input device ID attached to your `mididevice` object, returned as a scalar double. If your system includes different MIDI devices with the same name, using the device ID removes ambiguity.

`InputID` is set during the creation of the `mididevice` object and cannot be modified later.

Data Types: `double`

### **OutputID — Output device name associated with `mididevice`**

-1 (default)

This property is read-only.

Unique MIDI output device ID attached to your `mididevice` object, returned as a scalar double. If your system includes different MIDI devices with the same name, using the device ID removes ambiguity.

`OutputID` is set during the creation of the `mididevice` object and cannot be modified later.

Data Types: `double`

## **Object Functions**

<code>midisend</code>	Send MIDI message to MIDI device
<code>midireceive</code>	Receive MIDI message from MIDI device
<code>hasdata</code>	Determine if data is available to read from MIDI device

## **Examples**

### **Connect Input and Output to Single MIDI Device**

Query your system for available MIDI devices.

```
mididevinfo
```

```
MIDI devices available:  
ID Direction Interface Name
```

```

0  output  MMSystem  'Microsoft MIDI Mapper'
1  input   MMSystem  'USB MIDI Interface '
2  output  MMSystem  'Microsoft GS Wavetable Synth'
3  output  MMSystem  'USB MIDI Interface '

```

Create a MIDI device object to interface with your selected device. If you specify a single MIDI device object, and it is capable of both input and output, `mididevice` connects to both the input and output.

```
device = mididevice('USB MIDI Interface ')
```

```

device =
mididevice connected to
  Input: 'USB MIDI Interface ' (1)
  Output: 'USB MIDI Interface ' (3)

```

### Connect Input to MIDI Device

Query your system for MIDI devices.

```
mididevinfo
```

```

MIDI devices available:
ID Direction Interface  Name
0  output  MMSystem  'Microsoft MIDI Mapper'
1  input   MMSystem  'USB MIDI Interface '
2  output  MMSystem  'Microsoft GS Wavetable Synth'
3  output  MMSystem  'USB MIDI Interface '

```

Create a MIDI device object to interface with your selected input device. As soon as you create the MIDI device object, it begins listening for MIDI messages and storing them in a buffer.

```
device = mididevice('Input','USB MIDI Interface ');
```

### Connect Output to MIDI Device

Query your system for available MIDI devices.

```
mididevinfo
```

```
MIDI devices available:
ID Direction Interface Name
0 output MMSystem 'Microsoft MIDI Mapper'
1 input MMSystem 'USB MIDI Interface '
2 output MMSystem 'Microsoft GS Wavetable Synth'
3 output MMSystem 'USB MIDI Interface '
```

Create a MIDI device object to interface with your selected output device.

```
device = mididevice('Output', 'USB MIDI Interface ')
```

```
device =
mididevice connected to
Output: 'USB MIDI Interface ' (3)
```

### Connect Input and Output to Different MIDI Devices

Query your system for available MIDI devices.

```
mididevinfo
```

```
MIDI devices available:
ID Direction Interface Name
0 output MMSystem 'Microsoft MIDI Mapper'
1 input MMSystem 'USB MIDI Interface '
2 output MMSystem 'Microsoft GS Wavetable Synth'
3 output MMSystem 'USB MIDI Interface '
```

Create a MIDI device object that receives data from one device and sends data to another device. In this example, the MIDI device object receives MIDI messages from the 'USB MIDI Interface ' device and sends data to the 'Microsoft GS Wavetable Synth' virtual output device. To avoid ambiguity, the MIDI devices are specified by the device IDs.

```
device = mididevice('Input',1, 'Output',2)
```

```
device =
mididevice connected to
Input: 'USB MIDI Interface ' (1)
Output: 'Microsoft GS Wavetable Synth' (2)
```

## **See Also**

mididevinfo | midimsg | midireceive | midisend

## **Topics**

“MIDI Device Interface”

## **External Websites**

MIDI Manufacturers Association

**Introduced in R2018a**

## hasdata

Determine if data is available to read from MIDI device

## Syntax

```
tf = hasdata(device)
```

## Description

`tf = hasdata(device)` returns logical 1 (true) if there is data available to read from the `mididevice` specified by `device`. Otherwise, it returns logical 0 (false).

## Examples

### Determine if Data Is Available to Receive

Create a `mididevice` object to interface with your MIDI device. Query your system for available MIDI devices.

```
mididevinfo
```

```
MIDI devices available:
ID Direction Interface Name
0 output MMSystem 'Microsoft MIDI Mapper'
1 input MMSystem 'nanoKONTROL2'
2 input MMSystem 'USB Uno MIDI Interface'
3 output MMSystem 'Microsoft GS Wavetable Synth'
4 output MMSystem 'nanoKONTROL2'
5 output MMSystem 'USB Uno MIDI Interface'
```

```
device = mididevice('USB Uno MIDI Interface')
```

```
device =
mididevice connected to
  Input: 'USB Uno MIDI Interface' (2)
  Output: 'USB Uno MIDI Interface' (5)
```

As soon as your `mididevice` object is created, it begins listening for MIDI messages and storing them in a buffer. When you call `midireceive`, MIDI messages are retrieved from the buffer and returned. You can use `hasdata` to query whether your `mididevice` object buffer contains unread MIDI messages.

```
hasdata(device)
```

```
ans = logical  
     0
```

## Input Arguments

**device** — **mididevice** object

`mididevice` object

Specify `device` as an object created by `mididevice`.

## See Also

`mididevice` | `mididevinfo` | `midimsg` | `midisend`

## Topics

“MIDI Device Interface”

## External Websites

MIDI Manufacturers Association

**Introduced in R2018a**

## midireceive

Receive MIDI message from MIDI device

### Syntax

```
msgs = midireceive(device)
msgs = midireceive(device,maxmsgs)
```

### Description

`msgs = midireceive(device)` returns the MIDI messages, `msgs`, received from a MIDI device using the MIDI device interface, `device`.

`msgs = midireceive(device,maxmsgs)` specifies the maximum number of MIDI messages to return as `maxmsgs`.

### Examples

#### Receive MIDI Messages

To determine what MIDI devices are attached to your MIDI input ports, call `mididevinfo`. Use the `availableDevices` struct to specify a valid MIDI device to create a `mididevice` object.

```
availableDevices = mididevinfo;
device = mididevice(availableDevices.input(1).ID);
```

Once your MIDI device object is created, it begins listening to MIDI messages from your specified device and storing them in a buffer. To get all MIDI messages in the buffer, call `midireceive`. In this example, several keys on a MIDI keyboard are played.

```
msgs = midireceive(device)
```

```
msgs =
```



```

MIDI message:
NoteOn      Channel: 1 Note: 52 Velocity: 64 Timestamp: 3.94 [ 90 34 40 ]
NoteOn      Channel: 1 Note: 52 Velocity: 0  Timestamp: 4.179 [ 90 34 00 ]
NoteOn      Channel: 1 Note: 48 Velocity: 64 Timestamp: 4.19  [ 90 30 40 ]
NoteOn      Channel: 1 Note: 47 Velocity: 64 Timestamp: 4.382 [ 90 2F 40 ]
NoteOn      Channel: 1 Note: 48 Velocity: 0  Timestamp: 4.459 [ 90 30 00 ]
NoteOn      Channel: 1 Note: 48 Velocity: 64 Timestamp: 4.59  [ 90 30 40 ]
NoteOn      Channel: 1 Note: 47 Velocity: 0  Timestamp: 4.776 [ 90 2F 00 ]
NoteOn      Channel: 1 Note: 50 Velocity: 64 Timestamp: 4.788 [ 90 32 40 ]
NoteOn      Channel: 1 Note: 47 Velocity: 64 Timestamp: 4.802 [ 90 2F 40 ]
NoteOn      Channel: 1 Note: 52 Velocity: 64 Timestamp: 4.831 [ 90 34 40 ]
NoteOn      Channel: 1 Note: 47 Velocity: 0  Timestamp: 4.84  [ 90 2F 00 ]
NoteOn      Channel: 1 Note: 48 Velocity: 0  Timestamp: 4.912 [ 90 30 00 ]
NoteOn      Channel: 1 Note: 52 Velocity: 0  Timestamp: 4.953 [ 90 34 00 ]
NoteOn      Channel: 1 Note: 50 Velocity: 0  Timestamp: 5.079 [ 90 32 00 ]

```

Reading from the buffer clears the data. For example, if no more MIDI messages are sent, and the buffer is reread, `midireceive` returns an empty MIDI message.

```
msgs = midireceive(device)
```

```
msgs =
```

```
empty MIDI message array
```

## Receive Limited Number of MIDI Messages

Query your system for available output from MIDI devices. Specify that the output of a MIDI device is connected to the input of your `mididevice` object.

```
mididevinfo
```

```

MIDI devices available:
ID  Direction  Interface  Name
0   output    MMSystem  'Microsoft MIDI Mapper'
1   input     MMSystem  'USB MIDI Interface '
2   output    MMSystem  'Microsoft GS Wavetable Synth'
3   output    MMSystem  'USB MIDI Interface '

```

```
device = mididevice('Input', 'USB MIDI Interface ');
```

Once your MIDI device object is created, it begins listening to MIDI messages from your specified device and storing them in a buffer. To get a limited number of MIDI messages from the buffer, call `midireceive` and specify the maximum number of messages to return. In this example, five keys are played on a MIDI device. A maximum of four MIDI messages are received at each call to `midireceive`.

```
midireceive(device,4)
```

```
ans =
```

```
MIDI message:
```

```
NoteOn      Channel: 1 Note: 36 Velocity: 64 Timestamp: 2929.71 [ 90 24 40 ]
NoteOn      Channel: 1 Note: 36 Velocity: 0  Timestamp: 2929.91 [ 90 24 00 ]
NoteOn      Channel: 1 Note: 37 Velocity: 64 Timestamp: 2930.43 [ 90 25 40 ]
NoteOn      Channel: 1 Note: 37 Velocity: 0  Timestamp: 2930.59 [ 90 25 00 ]
```

```
midireceive(device,4)
```

```
ans =
```

```
MIDI message:
```

```
NoteOn      Channel: 1 Note: 38 Velocity: 64 Timestamp: 2931.16 [ 90 26 40 ]
NoteOn      Channel: 1 Note: 38 Velocity: 0  Timestamp: 2931.32 [ 90 26 00 ]
NoteOn      Channel: 1 Note: 39 Velocity: 64 Timestamp: 2931.87 [ 90 27 40 ]
NoteOn      Channel: 1 Note: 39 Velocity: 0  Timestamp: 2932.01 [ 90 27 00 ]
```

```
midireceive(device,4)
```

```
ans =
```

```
MIDI message:
```

```
NoteOn      Channel: 1 Note: 40 Velocity: 64 Timestamp: 2932.52 [ 90 28 40 ]
NoteOn      Channel: 1 Note: 40 Velocity: 0  Timestamp: 2932.66 [ 90 28 00 ]
```

## Input Arguments

### **device** — Object of `mididevice`

object of `mididevice`

Specify `device` as an object created by `mididevice`.

### **maxmsgs** — Maximum number of messages to return

positive integer scalar

Maximum number of messages to return, specified as a positive integer scalar.

Data Types: `double`

## Output Arguments

### **msgs** — Object of `midimsg`

scalar | column vector

Object of `midimsg`, returned as a scalar or column vector. The number of MIDI messages in the `mididevice` buffer and `maxmsgs` determine the size of `msgs`.

## See Also

`mididevice` | `mididevinfo` | `midimsg` | `midisend`

## Topics

“MIDI Device Interface”

## External Websites

MIDI Manufacturers Association

## Introduced in R2018a

## midisend

Send MIDI message to MIDI device

### Syntax

```
midisend(device,msg)
midisend(device,varargin)
```

### Description

`midisend(device,msg)` sends the MIDI message, `msg`, to a MIDI device using the MIDI device interface, `device`.

`midisend(device,varargin)` creates MIDI messages using `varargin` and then sends the MIDI messages. The `varargin` syntax is for convenience and includes a call to `midimsg` with the call to `midisend`.

### Examples

#### Send MIDI Messages to Device

Query your system for available MIDI device output ports. Use the `availableDevices` struct to specify a valid MIDI device and create a `mididevice` object.

```
availableDevices = mididevinfo;
device = mididevice(availableDevices.output(2).ID);
```

Create a pair of `NoteOn` messages (to indicate Note On and Note Off) and send them to your selected MIDI device.

```
msgs = midimsg('Note',1,48,64,0.25);
midisend(device,msgs)
```

## Define and Send MIDI Messages to Device

midisend enables you to combine the definition and sending of a midimsg into a single function call. Send middle C on channel 3 with velocity 64.

```
mididevinfo
```

```
MIDI devices available:
ID Direction Interface Name
0 output MMSystem 'Microsoft MIDI Mapper'
1 input MMSystem 'nanoKONTROL2'
2 input MMSystem 'USB Uno MIDI Interface'
3 output MMSystem 'Microsoft GS Wavetable Synth'
4 output MMSystem 'nanoKONTROL2'
5 output MMSystem 'USB Uno MIDI Interface'
```

```
device = mididevice('USB Uno MIDI Interface')
```

```
device =
mididevice connected to
  Input: 'USB Uno MIDI Interface' (2)
  Output: 'USB Uno MIDI Interface' (5)
```

```
midisend(device, 'NoteOn', 3, 60, 64)
```

## Input Arguments

### device — Object of mididevice

scalar

Specify device as an object created by mididevice.

### msg — Object of midimsg

scalar | vector | array

Specify msg as an object created by midimsg.

### varargin — Variable number of arguments describing MIDI message

midimsg input arguments

Specify varargin as a valid combination of arguments that can construct a MIDI message. See midimsg for a description of valid arguments.

## **See Also**

`mididevice` | `mididevinfo` | `midimsg` | `midireceive`

## **Topics**

“MIDI Device Interface”

## **External Websites**

MIDI Manufacturers Association

**Introduced in R2018a**

# 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 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 “Audio Plugins in MATLAB”.

## 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

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see “Object Behavior” (MATLAB) 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.
end
```



```
%
% 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  
end
```

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

#### Classes

audioPluginSource

#### Functions

audioPluginInterface | audioPluginParameter | generateAudioPlugin |  
validateAudioPlugin

#### Topics

“Audio Plugins in MATLAB”

“Audio Plugin Example Gallery”

“Hierarchies of Classes — Concepts” (MATLAB)

**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

### 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** A plugin must not call `setSampleRate` on itself. If the plugin attempts to call `setSampleRate` on itself, `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 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

`getSamplesPerFrame` Get frame size returned by the plugin

`setSamplesPerFrame` Set frame size returned by the plugin (MATLAB environment only)

## Inherited Methods

`getSampleRate` Get sample rate at which the plugin is run

`setSampleRate` Set sample rate at which the plugin is run

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see “Object Behavior” (MATLAB) 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.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

#### Classes

audioPlugin

#### Functions

audioPluginInterface | audioPluginParameter | generateAudioPlugin |  
validateAudioPlugin

### Topics

“Audio Plugins in MATLAB”

“Audio Plugin Example Gallery”

“Hierarchies of Classes — Concepts” (MATLAB)

**Introduced in R2016a**

## getSamplesPerFrame

**Class:** audioPluginSource

Get frame size returned by the plugin

### Syntax

```
frameSize = getSamplesPerFrame(myAudioSourcePlugin)
```

### Description

`frameSize = getSamplesPerFrame(myAudioSourcePlugin)` returns the frame size at which the plugin is run. `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.

---

**Note** When authoring source plugins in MATLAB, `getSamplesPerFrame` is valid only when called in the processing function.

---

**Introduced in R2016a**



# setSamplesPerFrame

**Class:** audioPluginSource

Set frame size returned by the plugin (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 authored plugin, `generateAudioPlugin` throws an error.

---

**Introduced in R2016a**

## externalAudioPlugin class

Base class for external audio plugins

### Description

`externalAudioPlugin` is the base class for hosted audio plugins. When you load an external plugin using `loadAudioPlugin`, an object of that plugin is created having `externalAudioPlugin` or `externalAudioPluginSource` as a base class. The `externalAudioPluginSource` class is used when the external audio plugin is a source plugin.

For a tutorial on hosting audio plugins, see “Host External Audio Plugins”.

### Methods

<code>dispParameter</code>	Display information of single or multiple parameters
<code>getParameter</code>	Get normalized value and information about parameter
<code>info</code>	Get information about hosted plugin
<code>process</code>	Process audio stream
<code>setParameter</code>	Set normalized parameter value of hosted plugin

### 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

### Copy Semantics

Handle. To learn how handle classes affect copy operations, see “Object Behavior” (MATLAB) in the MATLAB documentation.

## Examples

### Specify Hosted Plugin Parameter Values

Load a VST audio plugin into MATLAB® by specifying its full path. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
pluginPath = fullfile(matlabroot, 'toolbox/audio/samples/ParametricEqualizer.dll');  
hostedPlugin = loadAudioPlugin(pluginPath)
```

Use `info` to return information about the hosted plugin.

```
info(hostedPlugin)
```

Use `setParameter` to change the normalized value of the Medium Center Frequency parameter to 0.75. Specify the parameter by its index.

```
setParameter(hostedPlugin, 5, 0.75)
```

When you set the normalized parameter value, the parameter display value is automatically updated. The normalized parameter value generally corresponds to the position of a UI widget or MIDI controller. The parameter display value typically reflects the value used internally for processing.

Use `dispParameter` to display the updated table of parameters.

```
dispParameter(hostedPlugin)
```

Alternatively, you can use `getParameter` to return the normalized value of a single parameter.

```
parameterIndex = 5;  
parameterValue = getParameter(hostedPlugin, parameterIndex)
```

### Run External Plugin in MATLAB

Load a VST audio plugin into MATLAB™ by specifying its full path. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
pluginPath = ...
    fullfile(matlabroot, 'toolbox/audio/samples/ParametricEqualizer.dll');
hostedPlugin = loadAudioPlugin(pluginPath);
```

Create input and output objects for an audio stream loop that reads from a file and writes to your audio device. Set the sample rate of the hosted plugin to the sample rate of the input to the plugin.

```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3');
deviceWriter = audioDeviceWriter('SampleRate', fileReader.SampleRate);
setSampleRate(hostedPlugin, fileReader.SampleRate);
```

Set the `MediumPeakGain` property to -20 dB.

```
hostedPlugin.MediumPeakGain = -20;
```

Use the hosted plugin to process the audio file in an audio stream loop. Sweep the medium peak gain upward in the loop to hear the effect.

```
while hostedPlugin.MediumPeakGain < 19
    hostedPlugin.MediumPeakGain = hostedPlugin.MediumPeakGain + 0.04;
    x = fileReader();
    y = process(hostedPlugin, x);
    deviceWriter(y);
end

release(fileReader)
release(deviceWriter)
```

## See Also

### Functions

`loadAudioPlugin`

### Classes

`audioPlugin` | `audioPluginSource` | `externalAudioPluginSource`

## Topics

“Host External Audio Plugins”

“Hierarchies of Classes — Concepts” (MATLAB)

**Introduced in R2016b**

## dispParameter

**Class:** externalAudioPlugin

Display information of single or multiple parameters

### Syntax

```
dispParameter(hostedPlugin)
dispParameter(hostedPlugin,parameter)
```

### Description

`dispParameter(hostedPlugin)` displays all parameters and associated indices, values, displayed values, and display labels. For example:

```
dispParameter(hostedPlugin)
```

	Parameter	Value	Display
1	Wet:	1.0000	+0.0 dB
2	Dry:	1.0000	+0.0 dB
3	1: Enabled:	1.0000	ON
4	1: Length:	0.0000	0.0 ms
5	1: Length:	0.0156	4.00 8N
6	1: Feedback:	0.0000	-inf dB
7	1: Lowpass:	1.0000	20000 Hz
8	1: Hipass:	0.0000	0 Hz
9	1: Resolution:	1.0000	24 bits
10	1: Stereo width:	1.0000	1.00
11	1: Volume:	1.0000	+0.0 dB
12	1: Pan:	0.5000	0.0 %

The `Value` column corresponds to the normalized parameter value. Generally, the normalized parameter value represents the position of a UI widget or MIDI controller. The `Display` column corresponds to an internal parameter value used for processing. The `Value` and `Display` are related by an unknown mapping that is internal to the hosted plugin.

`dispParameter(hostedPlugin,parameter)` displays a subset of parameters. You can specify a parameter by its name as a character vector, string, or as a vector of one or more parameter indices. For example:

- `dispParameter(hostedPlugin, 'Gain')` displays information about the 'Gain' parameter of `hostedPlugin`.
- `dispParameter(hostedPlugin, [1,3])` displays information about parameters specified by indices 1 and 3.

**Introduced in R2016b**

## getParameter

**Class:** externalAudioPlugin

Get normalized value and information about parameter

### Syntax

```
value = getParameter(hostedPlugin,parameter)
[value, parameterInformation] = getParameter(hostedPlugin,parameter)
```

### Description

`value = getParameter(hostedPlugin,parameter)` returns the normalized value of the parameter of `hostedPlugin`. You can specify a parameter by its name as a character vector, string, or by its index. For example:

- `getParameter(hostedPlugin, 'Gain')` returns the normalized value of the hosted plugin parameter named 'Gain'. If the parameter name is not unique, `getParameter` returns an error.
- `getParameter(hostedPlugin,2)` returns information about the parameter specified by index 2.

`[value, parameterInformation] = getParameter(hostedPlugin,parameter)` returns a structure containing additional information about the specified parameter of the hosted plugin.

Field	Description
DisplayName	Display name or prompt of the plugin parameter, returned as a character vector. The display name is intended for display on the plugin's user interface (UI).



---

Field	Description
DisplayValue	Display value of the plugin parameter, returned as a character vector. The parameter DisplayValue corresponds to the normalized parameter value by an unknown mapping internal to the hosted plugin. Generally, the display value reflects the value used internally by the plugin for processing, while the normalized parameter value corresponds to the position of a MIDI control or widget on a UI.
Label	Label intended for display with DisplayValue on the plugin's UI, returned as a character vector. Typical labels include dB and Hz.

**Introduced in R2016b**

## info

**Class:** externalAudioPlugin

Get information about hosted plugin

## Syntax

```
pluginInfo = info(hostedPlugin)
```

## Description

`pluginInfo = info(hostedPlugin)` returns a structure containing information about the hosted plugin.

Field	Description
PluginName	Display name of plugin.
Format	Software interface. Supported formats include VST, VST3, and AU.
InputChannels	Number of channels passed to the processing function of the plugin.
OutputChannels	Number of channels returned from the processing function of the plugin.
NumParams	Total number of plugin parameters.
PluginPath	Path specified when plugin is loaded using <code>loadAudioPlugin</code> .
VendorName	Name of the plugin creator.
VendorVersion	Version number. Typically used to track plugin releases.
UniqueID	Unique identifier of plugin used for recognition in certain digital audio workstation (DAW) environments.

**Introduced in R2016b**

## process

**Class:** externalAudioPlugin

Process audio stream

## Syntax

```
audioOut = process(hostedPlugin, audioIn)
```

## Description

`audioOut = process(hostedPlugin, audioIn)` returns an audio signal processed according to the algorithm and parameters of `hostedPlugin`. For source plugins, call `process` without an audio input. Use `info(hostedPlugin)` to determine the number of channels (columns) of the input and output audio signal.

Use `setSamplesPerFrame(hostedPlugin)` to specify the frame size returned by hosted source plugins.

**Introduced in R2016b**

## setParameter

**Class:** externalAudioPlugin

Set normalized parameter value of hosted plugin

### Syntax

```
setParameter(hostedPlugin,parameter,newValue)
```

### Description

`setParameter(hostedPlugin,parameter,newValue)` sets the normalized value corresponding to the `parameter` of `hostedPlugin` to `newValue`. Specify the parameter by its unique display name or its index. Specify the new normalized parameter value as a scalar in the range 0-1.

For example, assume `hostedPlugin` has a parameter with index 3 and a unique display name, 'Gain'. These commands are identical:

- `setParameter(hostedPlugin,'Gain',0.2)`
- `setParameter(hostedPlugin,3,0.2)`

---

**Note** A hosted plugin might quantize its parameters. The result of `setParameter` for quantized parameters depends on the type of quantization.

---

**Introduced in R2016b**

# externalAudioPluginSource class

Base class for external audio source plugins

## Description

`externalAudioPluginSource` is the base class for hosted audio source plugins. When you load an external plugin using `loadAudioPlugin`, an object of that plugin is created having `externalAudioPlugin` or `externalAudioPluginSource` as a base class. The `externalAudioPluginSource` class is used when the external audio plugin is a source plugin.

For a tutorial on hosting audio plugins, see “Host External Audio Plugins”.

## Methods

### Inherited Methods

<code>dispParameter</code>	Display information of single or multiple parameters
<code>getParameter</code>	Get normalized value and information about parameter
<code>info</code>	Get information about hosted plugin
<code>process</code>	Process audio stream
<code>setParameter</code>	Set normalized parameter value of hosted plugin

<code>getSampleRate</code>	Get sample rate at which the plugin is run
<code>setSampleRate</code>	Set sample rate at which the plugin is run

<code>getSamplesPerFrame</code>	Get frame size returned by the plugin
<code>setSamplesPerFrame</code>	Set frame size returned by the plugin (MATLAB environment only)

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see “Object Behavior” (MATLAB) in the MATLAB documentation.

## Examples

### Specify Hosted Source Plugin Parameter Values

Load a VST audio source plugin into MATLAB® by specifying its full path. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
pluginPath = fullfile(matlabroot, 'toolbox/audio/samples/oscillator.dll');  
hostedSourcePlugin = loadAudioPlugin(pluginPath)
```

Use `info` to return information about the hosted plugin.

```
info(hostedSourcePlugin)
```

Use `setParameter` to change the normalized value of the Frequency parameter to 0.8. Specify the parameter by its index.

```
setParameter(hostedSourcePlugin, 1, 0.8)
```

When you set the normalized parameter value, the parameter display value is automatically updated. Generally, the normalized parameter value corresponds to the position of a UI widget or MIDI controller. The parameter display value typically reflects the value used internally by the plugin for processing.

Use `dispParameter` to display the updated table of parameters.

```
dispParameter(hostedSourcePlugin)
```

Alternatively, you can use `getParameter` to return the normalized value of a single parameter.

```
getParameter(hostedSourcePlugin, 1)
```

## Run External Source Plugin in MATLAB

Load a VST audio source plugin into MATLAB™ by specifying its full path. If you are using a Mac, replace the .dll file extension with .vst.

```
pluginPath = fullfile(matlabroot, 'toolbox', 'audio', 'samples', 'oscillator.dll');  
hostedSourcePlugin = loadAudioPlugin(pluginPath);
```

Set the Amplitude property to 0.5. Set the Frequency property to 16 kHz.

```
hostedSourcePlugin.Amplitude = 0.5;  
hostedSourcePlugin.Frequency = 16000;
```

Set the sample rate at which to run the plugin. Create an output object to write to your audio device.

```
setSampleRate(hostedSourcePlugin, 44100);  
deviceWriter = audioDeviceWriter('SampleRate', 44100);
```

Use the hosted source plugin to output an audio stream. The processing in the audio stream loop ramps the frequency parameter down and then up.

```
k = 1;  
for i = 1:1000  
    hostedSourcePlugin.Frequency = hostedSourcePlugin.Frequency - 30*k;  
    y = process(hostedSourcePlugin);  
    deviceWriter(y);  
    if (hostedSourcePlugin.Frequency - 30 <= 0.1) || ...  
        (hostedSourcePlugin.Frequency + 30 >= 20e3)  
        k = -1*k;  
    end  
end  
release(deviceWriter)
```

## See Also

### Functions

loadAudioPlugin

### Classes

audioPlugin | audioPluginSource | externalAudioPlugin

## **Topics**

“Host External Audio Plugins”

“Hierarchies of Classes — Concepts” (MATLAB)

**Introduced in R2016b**



# Blocks in Audio Toolbox

---

## Voice Activity Detector

Detect presence of speech in audio signal

**Library:** Audio Toolbox / Measurements



### Description

The Voice Activity Detector block detects the presence of speech in an audio signal. You can also use the Voice Activity Detector block to output an estimate of the noise variance per frequency bin.

### Ports

#### Input

**Port\_1 — Input signal**

matrix | 1-D vector

- Matrix input -- Each column of the input is treated as an independent channel.
- 1-D vector input -- The input is treated as a single channel.

Data Types: `single` | `double`

#### Output

**P — Probability that speech is present**

scalar | row vector

The block outputs a scalar or row vector with the same number of columns as the input signal.

This port is unnamed until you select the **Output noise variance** parameter.

Data Types: single | double

### **N — Estimate of noise variance per frequency bin**

column vector | matrix

The block outputs a column vector or a matrix with the same number of columns as the input signal.

#### **Dependencies**

To enable this port, select the **Output noise variance** parameter.

Data Types: single | double

## **Parameters**

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

### **Domain of the input — Domain of the input**

Time (default) | Frequency

**Tunable:** No

### **Window — Windowing function applied before FFT**

Hann (default) | Chebyshev | Flat Top | Hamming | Kaiser | Rectangular

The window function is designed using the algorithms of the following functions:

- Hann -- hann
- Chebyshev -- chebwin
- Flat Top -- flattopwin
- Hamming -- hamming
- Kaiser -- kaiser

**Tunable:** No

#### **Dependencies**

To enable this parameter, set **Domain of the input** to Time.

**Sidelobe attenuation of the window (dB) — Sidelobe attenuation of the window (dB)**

60 (default) | positive finite scalar

**Tunable:** No

**Dependencies**

To enable this parameter, set **Domain of the input** to Time and **Window** to Chebyshev or Kaiser.

Data Types: single | double

**Inherit FFT length from input dimensions — Set FFT length to number of input samples**

on (default) | off

**Tunable:** No

**Dependencies**

To enable this parameter, set **Domain of the input** to Time.

**FFT length — Number of bins in frequency domain**

1024 (default) | positive finite integer

**Tunable:** No

**Dependencies**

To enable this parameter, set **Domain of the input** to Time and clear the **Inherit FFT length from input dimensions** parameter.

Data Types: single | double

**Probability of transition from a silence frame to a speech frame — Probability that a speech frame follows a silence frame**

0.2 (default) | value in the range [0,1]

**Tunable:** Yes

Data Types: single | double

**Probability of transition from a speech frame to a silence frame — Probability that a silence frame follows a speech frame**

0.1 (default) | value in the range [0,1]

**Tunable:** Yes

Data Types: `single` | `double`

**Output noise variance** — Output estimate of noise variance per frequency bin  
`off` (default) | `on`

When you select this parameter, an additional output port, **N**, is added to the block.

**Tunable:** No

**Simulate using** — Specify type of simulation to run

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

- `Code generation` -- Simulate model using generated C code. The first time you run a simulation, Simulink® generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is comparable to `Interpreted execution`.
- `Interpreted execution` -- Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.

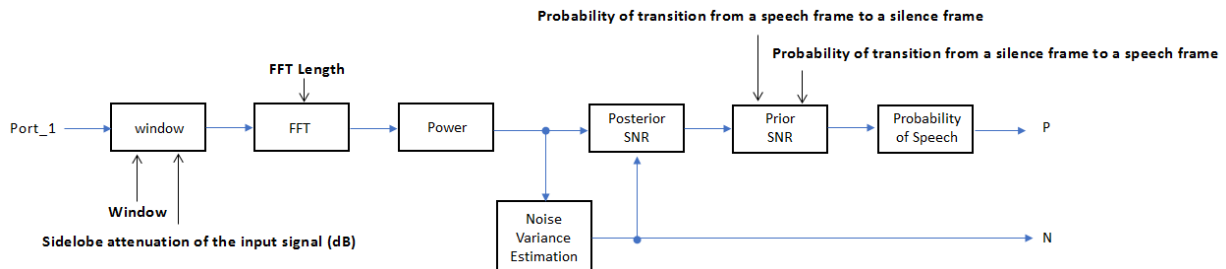
**Tunable:** No

## Block Characteristics

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

## Algorithms

The Voice Activity Detector implements the algorithm described in [1].



If **Domain of the input** is specified as Time, the input signal is windowed and then converted to the frequency domain according to the **Window**, **Sidelobe attenuation of the window (dB)**, and **FFT length** parameters. If **Domain of the input** is specified as Frequency, the input is assumed to be a windowed discrete time Fourier transform (DTFT) of an audio signal. The signal is then converted to the power domain. Noise variance is estimated according to [2]. The posterior and prior SNR are estimated according to the Minimum Mean-Square Error (MMSE) formula described in [3]. A log likelihood ratio test with a Hidden Markov Model (HMM)-based hang-over scheme is used, according to [1].

## References

- [1] Sohn, Jongseo., Nam Soo Kim, and Wonyong Sung. "A Statistical Model-Based Voice Activity Detection." *Signal Processing Letters IEEE*. Vol. 6, No. 1, 1999.
- [2] Martin, R. "Noise Power Spectral Density Estimation Based on Optimal Smoothing and Minimum Statistics." *IEEE Transactions on Speech and Audio Processing*. Vol. 9, No. 5, 2001, pp. 504-512.
- [3] Ephraim, Y., and D. Malah. "Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 32, No. 6, 1984, pp. 1109-1121.

## **Extended Capabilities**

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

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

### **See Also**

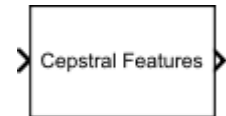
`voiceActivityDetector`

**Introduced in R2018a**

# Cepstral Feature Extractor

Extract cepstral features from audio segment

**Library:** Audio Toolbox / Measurements



## Description

The Cepstral Feature Extractor block extracts cepstral features from an audio segment. Cepstral features are commonly used to characterize speech and music signals.

## Ports

### Input

#### Port\_1 — Audio input to cepstral feature extractor

column vector | matrix

Audio input to the cepstral feature extractor, specified as a column vector or a matrix. If specified as a matrix, the columns are treated as independent audio channels.

Data Types: single | double

### Output

#### coeffs — Cepstral coefficients

column vector | matrix

Cepstral coefficients, returned as a column vector or a matrix. If the coefficients matrix is an  $N$ -by- $M$  matrix,  $N$  is determined by the values you specify in the **Number of coefficients to return** and **Log energy usage** parameters.  $M$  equals the number of input audio channels.

When the **Log energy usage** parameter is set to:



- **Append** -- The block prepends the log energy value to the coefficients vector. The length of the coefficients vector is  $1 + NumCoeffs$ , where *NumCoeffs* is the value specified in the **Number of coefficients to return** parameter.
- **Replace** -- The block replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is *NumCoeffs*.
- **Ignore** -- The block does not calculate or return the log energy.

This port is unnamed until you select **Output delta** parameter, the **Output delta-delta** parameter, or both.

Data Types: `single` | `double`

### **delta** — Change in coefficients

column vector | matrix

Change in coefficients over consecutive calls to the algorithm, returned as a column vector or a matrix. The **delta** array is of the same size and data type as the **coeffs** array.

#### **Dependencies**

To enable this port, select the **Output delta** parameter.

Data Types: `single` | `double`

### **deltaDelta** — Change in delta values

column vector | matrix

Change in **delta** values over consecutive calls to the algorithm, returned as a column vector or a matrix. The **deltaDelta** array is the same size and data type as the **coeffs** and **delta** arrays.

#### **Dependencies**

To enable this port, select the **Output delta-delta** parameter.

Data Types: `single` | `double`

## **Parameters**

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

### **Filter bank type** — Type of filter bank

Mel (default) | Gammatone

Type of filter bank, specified as either `Mel` or `Gammatone`:

- `Mel` -- The block computes the mel frequency cepstral coefficients (MFCC).
- `Gammatone` -- The block computes the gammatone cepstral coefficients (GTCC).

**Tunable:** No

**Domain of the input signal – Input signal domain**

`Time` (default) | `Frequency`

Input signal domain, specified as either `Time` or `Frequency`.

**Tunable:** No

**Number of coefficients to return – Number of coefficients to return**

`13` (default) | positive integer

Number of coefficients to return, specified as an integer in the range  $[2, v]$ , where  $v$  is the number of valid passbands. The number of valid passbands depends on the type of filter bank:

- `Mel` -- The number of valid passbands is defined as  $\text{sum}(\kappa \leq \text{floor}(fs/2)) - 2$ , where  $\kappa$  is the number of band edges in the mel filter bank and  $fs$  is the sample rate.
- `Gammatone` -- The number of valid passbands is defined as  $\text{ceil}(\text{hz2erb}(R(2)) - \text{hz2erb}(R(1)))$ , where  $R$  is the frequency range of the gammatone filter bank.

**Tunable:** No

Data Types: `single` | `double`

**Inherit FFT length from input dimensions – Inherit FFT length from input**

`on` (default) | `off`

When you select this parameter, the FFT length is equal to the number of rows in the input signal.

**Tunable:** No

**Dependencies**

To enable this parameter, set **Domain of the input signal** to `Time`.

**FFTLength – FFT length**

`[]` (default) | positive integer

FFT length, specified as a positive integer. The default, [], means that the FFT length is equal to the number of rows in the input signal.

**Tunable:** No

#### **Dependencies**

To enable this parameter, set **Domain of the input signal** to Time and select the **Inherit FFT length from input dimensions** parameter.

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

#### **Log energy usage — Specify how the log energy is shown**

Append (default) | Replace | Ignore

Specify how the log energy is shown in the coefficients vector output, specified as:

- **Append** -- The block prepends the log energy to the coefficients vector. The length of the coefficients vector is  $1 + NumCoeffs$ , where *NumCoeffs* is the value specified in the **Number of coefficients to return** parameter.
- **Replace** -- The block replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is *NumCoeffs*.
- **Ignore** -- The block does not calculate or return the log energy.

**Tunable:** No

#### **Output delta — Output delta values**

off (default) | on

When you select this parameter, an additional output port, **delta**, is added to the block. This port outputs the change in coefficients over consecutive calls to the algorithm.

**Tunable:** No

#### **Output delta-delta — Output delta-delta values**

off (default) | on

When you select this parameter, an additional output port, **deltaDelta**, is added to the block. This port outputs the change in delta values over consecutive calls to the algorithm.

**Tunable:** No

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

off (default) | on

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

**Tunable:** No**Input sample rate (Hz) — Sample rate of input**

16000 (default) | positive scalar

Input sample rate in Hz, specified as a real positive scalar.

**Tunable:** Yes**Dependencies**

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

**Simulate using — Specify type of simulation to run**

Code generation (default) | Interpreted execution

- **Code generation** -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to **Interpreted execution**.
- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than **Code generation**. In this mode, you can debug the source code of the block.

**Tunable:** No**Advanced Tab****Gammatone frequency range (Hz) — Frequency range of gammatone filter bank (Hz)**

[50 8000] (default) | two-element row vector

Frequency range of the gammatone filter bank in Hz, specified as a positive, monotonically increasing two-element row vector. The maximum frequency range can be

any finite number. The center frequencies of the filter bank are equally spaced across the frequency range on the ERB scale.

**Tunable:** No

**Dependencies**

To enable this parameter, set **Filter bank type** to Gammatone.

**Band edges of Mel filter bank (Hz) – Band edges of mel filter bank**

row vector

Band edges of the filter bank in Hz, specified as a nonnegative monotonically increasing row vector in the range  $[0, \infty)$ . The maximum bandedge frequency can be any finite number. The number of bandedges must be in the range  $[4, 80]$ .

The default band edges are spaced linearly for the first ten and then logarithmically thereafter. The default band edges are set as recommended by [1].

**Tunable:** No

**Dependencies**

To enable this parameter, set **Filter bank type** to Mel.

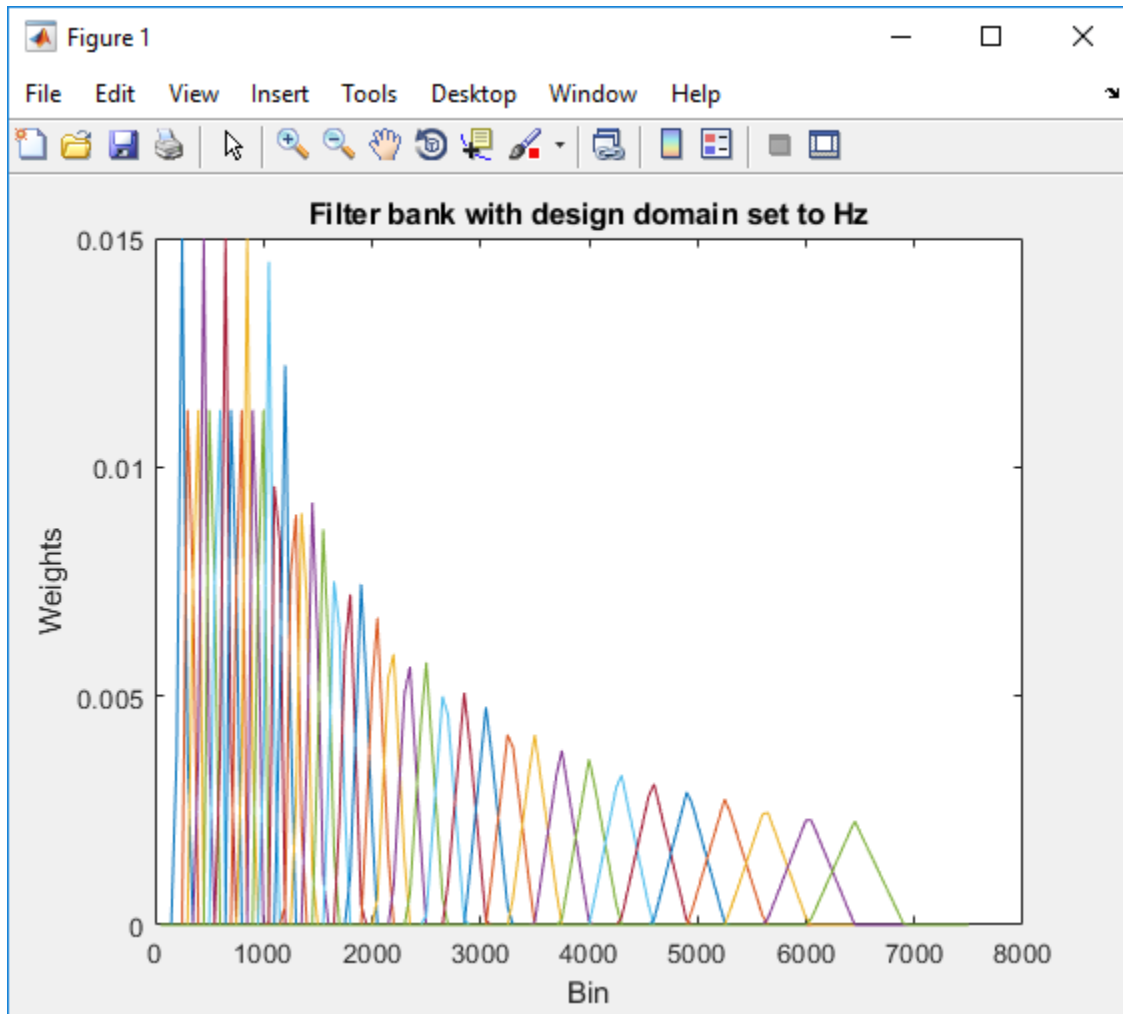
**Domain for Mel filter bank design – Mel filter bank design domain**

Hz (default) | Bin

Mel filter bank design domain, specified as either Hz or Bin. The filterbank is designed as overlapped triangles with band edges specified by the **Band edges of filter bank (Hz)** parameter.

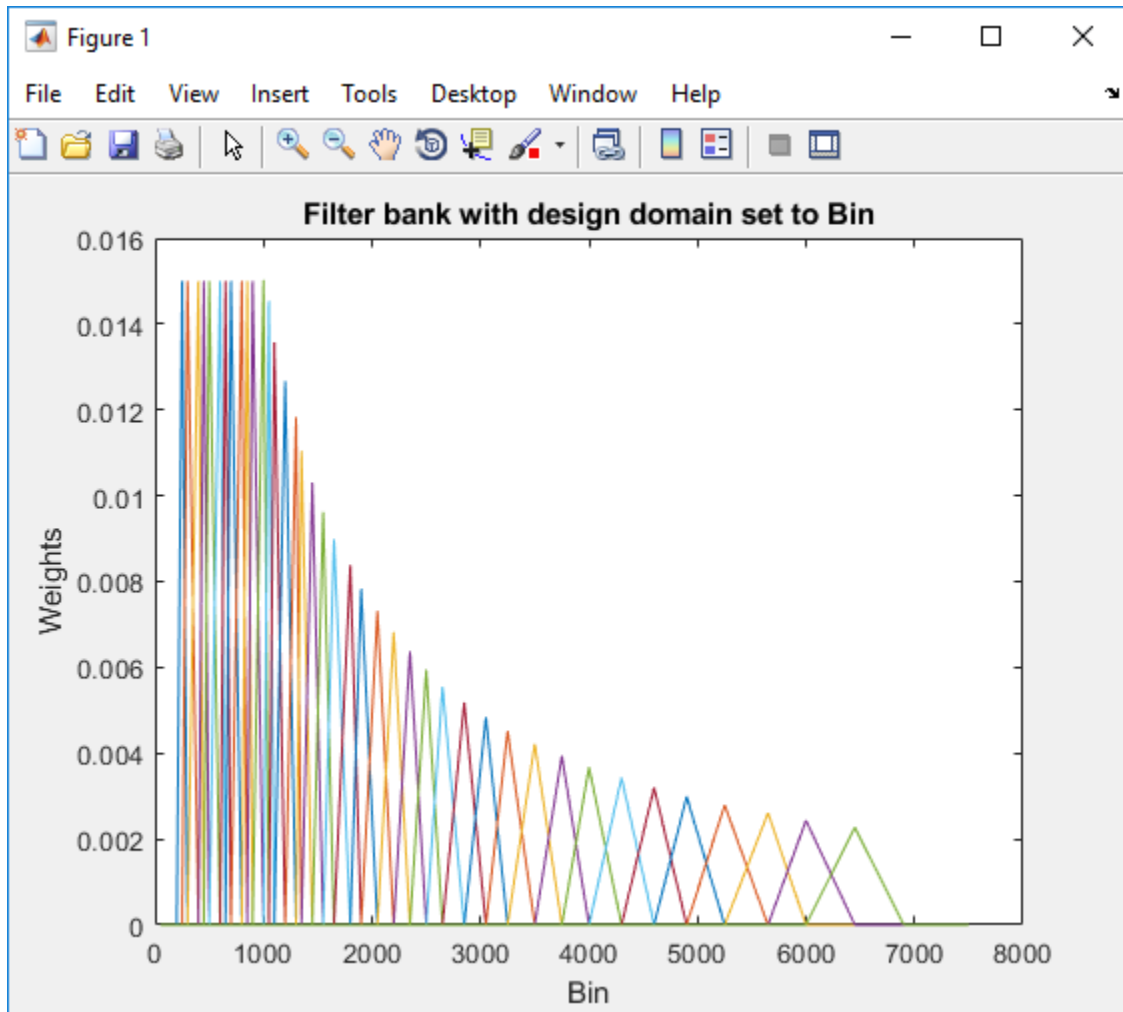
The band edges are specified in Hz. When you set the design domain to:

- Hz -- Filter bank triangles are drawn in Hz and are mapped onto bins.



For details, see [1].

- **Bin** -- The band edge frequencies in Hz are converted to bins. The filter bank triangles are drawn symmetrically in bins.



For details, see [2].

**Tunable:** No

**Dependencies**

To enable this parameter, set **Filter bank type** to Mel.

**Filter bank normalization — Normalize filter bank**

Bandwidth (default) | Area | None

Normalization technique used to normalize the weights of the filter bank, specified as:

- **Bandwidth** -- The weights of each bandpass filter are normalized by the corresponding bandwidth of the filter.
- **Area** -- The weights of each bandpass filter are normalized by the corresponding area of the bandpass filter.
- **None** -- The weights of the filter are not normalized.

**Tunable:** No

## Block Characteristics

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

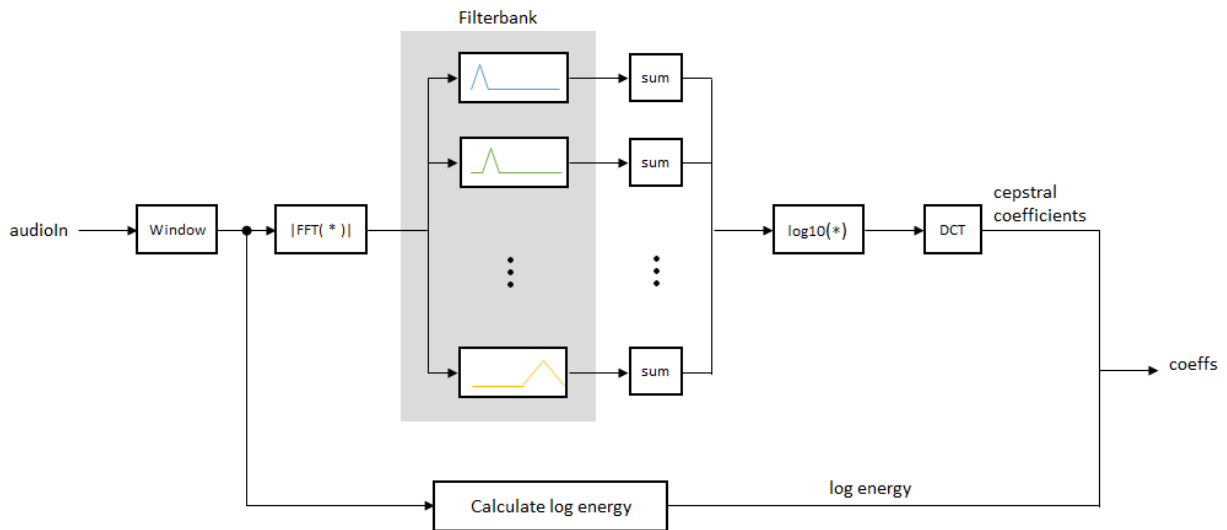
## Algorithms

### Auditory Cepstrum Coefficients

Auditory cepstrum coefficients are popular features extracted from speech signals for use in recognition tasks. In the source-filter model of speech, cepstral coefficients are understood to represent the filter (vocal tract). The vocal tract frequency response is relatively smooth, whereas the source of voiced speech can be modeled as an impulse train. As a result, the vocal tract can be estimated by the spectral envelope of a speech segment.



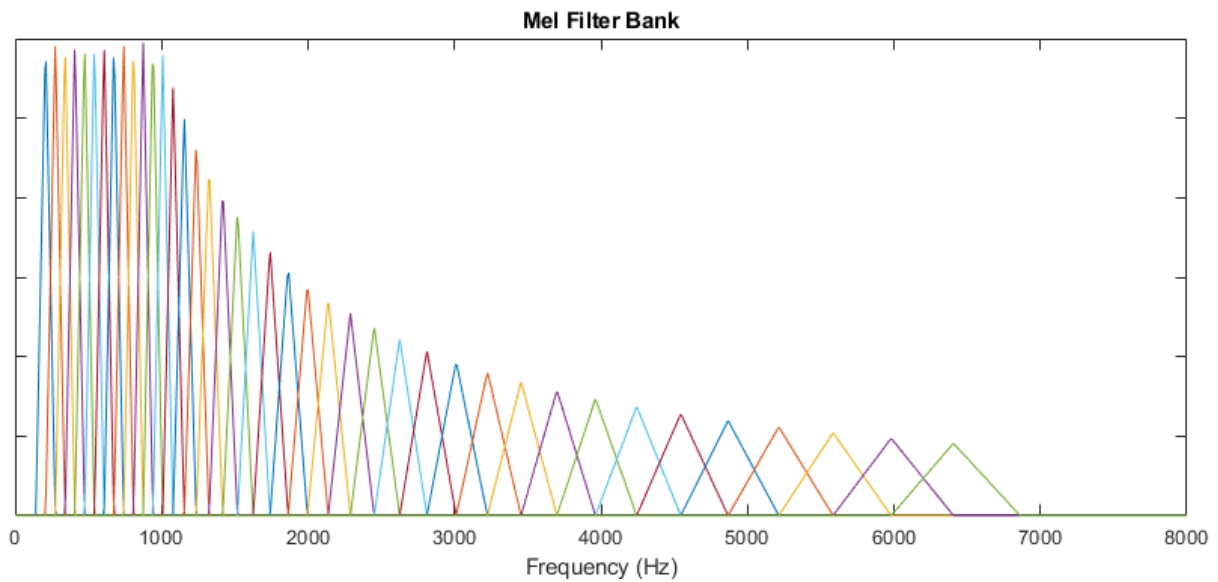
The motivating idea of cepstral coefficients is to compress information about the vocal tract (smoothed spectrum) into a small number of coefficients based on an understanding of the cochlea. Although there is no hard standard for calculating the coefficients, the basic steps are outlined by the diagram.



Two popular implementations of the filter bank are the mel filter bank and the gammatone filter bank.

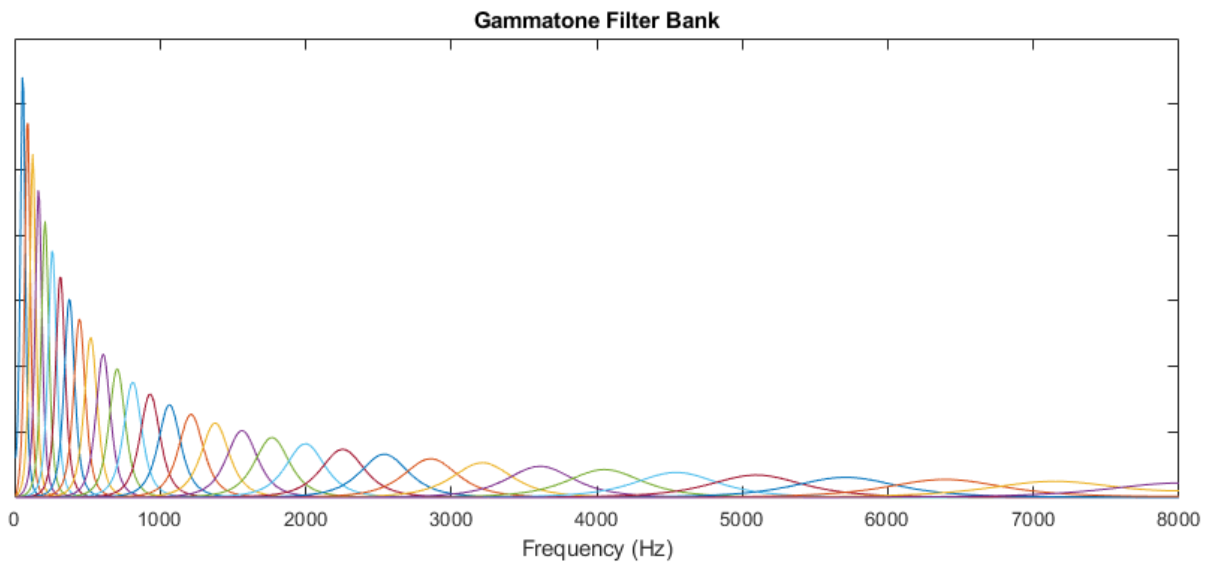
### Mel Filter Bank

The default mel filter bank linearly spaces the first 10 triangular filters and logarithmically spaces the remaining filters.



### **Gammatone Filter Bank**

The default gammatone filter bank is composed of gammatone filters spaced linearly on the ERB scale between 50 and 8000 Hz. The filter bank is designed by `gammatoneFilterBank`.



## Log Energy

If the input ( $x$ ) is a time-domain signal, the log energy is computed using the following equation:

$$\log E = \log(\text{sum}(x^2))$$

If the input ( $x$ ) is a frequency-domain signal, the log energy is computed using the following equation:

$$\log E = \log\left(\frac{\text{sum}(|x|^2)}{FFTL\text{Length}}\right)$$

## References

- [1] Auditory Toolbox. <https://engineering.purdue.edu/~malcolm/interval/1998-010/AuditoryToolboxTechReport.pdf>
- [2] ETSI ES 201 108 V1.1.3 (2003-09). [https://www.etsi.org/deliver/etsi\\_es/201100\\_201199/201108/01.01.03\\_60/es\\_201108v010103p.pdf](https://www.etsi.org/deliver/etsi_es/201100_201199/201108/01.01.03_60/es_201108v010103p.pdf)

## **Extended Capabilities**

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

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

### **See Also**

Voice Activity Detector | cepstralFeatureExtractor | mfcc | pitch | voiceActivityDetector

### **Topics**

“Speaker Identification Using Pitch and MFCC”

**Introduced in R2018a**

# Audio Device Reader

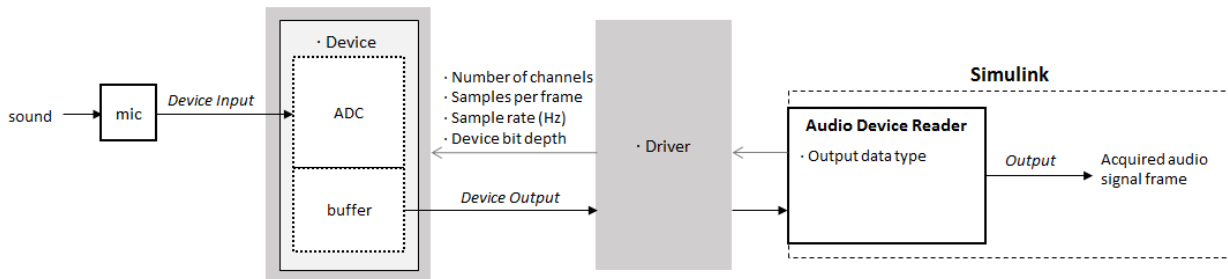
Record from sound card

**Library:** Audio Toolbox / Sources



## Description

The Audio Device Reader block reads audio samples using your computer's audio device. 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.



## Ports

## Output

### A — Output signal

scalar | vector | matrix

The output of the Audio Device Reader block is determined by the block's parameters. If the block output is a matrix, the columns correspond to independent channels.

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

### 0 — Number of samples overrun

scalar

This port outputs the number of samples overrun while acquiring a frame of data (one output matrix).

#### Dependencies

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

Data Types: uint32

## Parameters

### Main Tab

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

DirectSound (default) | ASIO | WASAPI

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

---

**Note** If **Driver** is set to ASIO, open the ASIO UI outside of MATLAB to set the sound card buffer size to the value specified by the **Samples per frame** parameter. See the documentation of your ASIO driver for more information.

---

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, set **Sample rate (Hz)** to a sample rate supported by your audio device.

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

#### Device — Device used to acquire audio samples

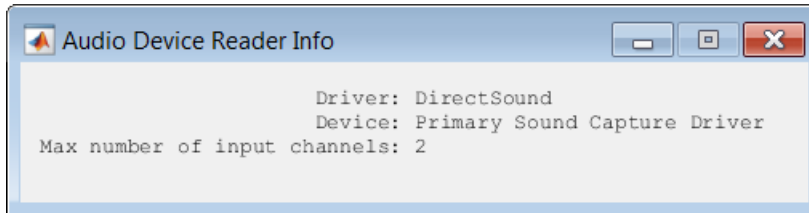
default audio device (default)

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

#### Info — View information about your audio input configuration

button

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



**Sample rate (Hz) — Sample rate your device uses to acquire audio data**

44100 (default) | integer

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

**Number of channels — Number of channels acquired by your audio device**

1 (default) | integer

The number of input channels is also the number of channels (matrix columns) output by the Audio Device Reader block.

**Dependencies**

To specify which input channels your audio device acquires, on the **Advanced** tab, select the **Use default channel mapping** parameter.

**Samples per frame — Frame size read from audio device**

1024 (default) | 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**

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

**Use default channel mapping — Toggle channel mapping source**

on (default) | off

When you select this parameter, 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 parameter, you specify the mapping in **Device input channels**.

**Device input channels — Specify nondefault channel mapping**

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

Nondefault map of device channels and matrix output by the Audio Device Reader block, specified as a scalar or 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.

**Dependencies**

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

**Output number of samples overrun — Specify additional output port for number of samples overrun**

off (default) | on

When you select this parameter, 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).

**Output data type — Data type output from block**

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



## Block Characteristics

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

a. Supports 16- and 32-bit signed and 8-bit unsigned integers.

## Extended Capabilities

### C/C++ Code Generation

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

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

## See Also

### System Objects

audioDeviceReader | audioDeviceWriter

### Blocks

Audio Device Writer

## **Topics**

“Run Audio I/O Features Outside MATLAB and Simulink”

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

**Introduced in R2016a**

# Audio Device Writer

Play to sound card

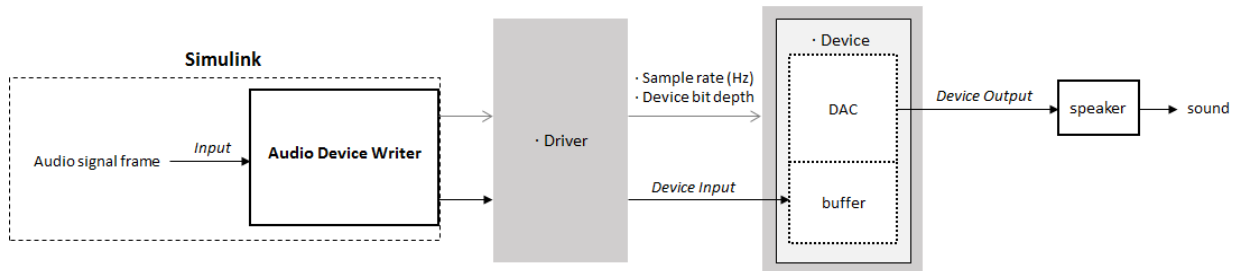
**Library:** Audio Toolbox / Sinks  
DSP System Toolbox / Sinks



## Description

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

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



## Ports

### Input

#### Port\_1 — Input signal

scalar | vector | matrix

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

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

## Output

### **Port\_1 – Number of samples underrun**

scalar

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

#### **Dependencies**

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

Data Types: uint32

## Parameters

### **Main Tab**

#### **Driver – Driver used to access your audio device**

DirectSound (default) | ASIO | WASAPI

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

---

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

---

- WASAPI drivers are supported for exclusive-mode only.

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

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

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

#### **Device – Device used to play audio samples**

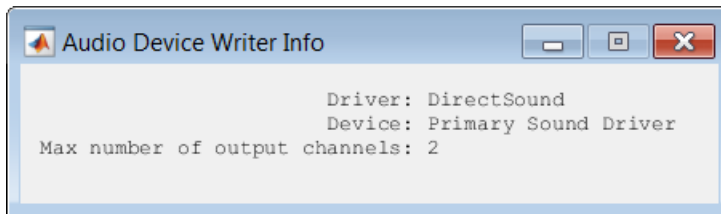
default audio device (default)

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

### **Info – View information about your audio output configuration**

button

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



### **Inherit sample rate from input – Specify source of input sample rate**

on (default) | off

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

### **Sample rate (Hz) – Sample rate used by device to play audio data**

44100 (default) | positive scalar

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

#### **Dependencies**

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

#### **Advanced Tab**

### **Device bit depth – Data type used by device to perform digital-to-analog conversion**

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

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

---

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

---

### Use default channel mapping — Toggle channel mapping source

on (default) | off

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

### Device output channels — Specify nondefault channel mapping

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

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

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

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

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

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

---

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

---

### Dependencies

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

### Output number of samples underrun — Specify output port for number of samples underrun

off (default) | on

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

## Block Characteristics

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

a. Supports 16- and 32-bit signed and 8-bit unsigned integers.

## Extended Capabilities

### C/C++ Code Generation

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

The following code generation limitations apply:

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

### See Also

Audio Device Reader | Binary File Reader | `audioDeviceReader` | `audioDeviceWriter`

### Topics

“Run Audio I/O Features Outside MATLAB and Simulink”

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

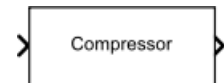
**Introduced in R2016a**



# Compressor

Dynamic range compressor

**Library:** Audio Toolbox / Dynamic Range Control



## Description

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. The block uses specified attack and release times to achieve a smooth applied gain curve.

## Ports

### Input

#### **x** — Input signal

matrix | 1-D vector

- Matrix input -- Each column of the input is treated as an independent channel.
- 1-D vector input -- The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

#### **T** — Threshold (dB)

scalar

### Dependencies

To enable this port, select **Specify from input port** for the “Threshold (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

### **R — Ratio**

scalar

#### **Dependencies**

To enable this port, select **Specify from input port** for the “Ratio” on page 5-0 parameter.

Data Types: `single` | `double`

### **K — Knee width (dB)**

scalar

#### **Dependencies**

To enable this port, select **Specify from input port** for the “Knee width (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

### **AT — Attack time (s)**

scalar

#### **Dependencies**

To enable this port, select **Specify from input port** for the “Attack time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

### **RT — Release time (s)**

scalar

#### **Dependencies**

To enable this port, select **Specify from input port** for the “Release time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

## **Output**

### **Y — Output signal**

matrix

The Compressor block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input -- The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input -- The block outputs an  $N$ -by-1 matrix (column vector), where  $N$  is the number of elements in the 1-D vector.

Data Types: single | double

### **G — Gain applied to each input sample**

matrix

#### **Dependencies**

To enable this port, select the **Output gain (dB)** parameter.

Data Types: single | double

## **Parameters**

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

#### **Main Tab**

#### **Threshold (dB) — Operation threshold**

-10 (default) | scalar in the range -50 to 0 inclusive

Operation threshold is the level above which gain is applied to the input signal.

To specify **Threshold (dB)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

#### **Ratio — Compression ratio**

5 (default) | scalar in the range 1 to 50 inclusive

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  $R = \frac{(x[n] - T)}{(y[n] - T)}$ , where

- $R$  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.

To specify **Ratio** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

#### **Knee width (dB) — Transition area in compression characteristic**

0 (default) | scalar in the range 0 to 20 inclusive

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  $(2 \times |x - T|) \leq W$ , where

- $y$  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.

To specify **Knee width (dB)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

#### **View static characteristic — Open static characteristic plot of dynamic range compressor**

button

The plot is updated automatically when parameters of the Compressor block change.

**Tunable:** Yes

**Attack time (s) — Time for applied gain to ramp up**

0.05 (default) | scalar in the range 0 to 4 inclusive

Attack time is the time the compressor gain takes to rise from 10% to 90% of its final value when the input goes above the threshold. The **Attack time (s)** parameter smooths the applied gain curve.

To specify **Attack time (s)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

**Release time (s) — Time for applied gain to ramp down**

0.2 (default) | scalar in the range 0 to 4 inclusive

Release time is the time the compressor gain takes to drop from 90% to 10% of its final value when the input goes below the threshold. The **Release time (s)** parameter smooths the applied gain curve.

To specify **Release time (s)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

**Make-up gain mode — Make-up gain mode**

Property (default) | Auto

- **Property** -- Make-up gain is set to the value specified by the **Make-up gain (dB)** parameter.
- **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.

**Tunable:** No

**Make-up gain (dB) — Applied make-up gain**

0 (default) | scalar in the range -10 to 24 inclusive

Make-up gain compensates for gain lost during compression. It is applied at the output of the Compressor block.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set the **Make-up gain mode** parameter to Property.

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

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

**Tunable:** No

**Input sample rate (Hz) — Sample rate of input**  
44100 (default) | positive scalar

**Tunable:** Yes

**Dependencies**

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

**Advanced Tab**

**Output gain (dB) — Gain applied on each input sample**  
off (default) | on

When you select this parameter, an additional output port, **G**, is added to the block. The **G** port outputs the gain applied on each input channel in dB.

**Tunable:** No

**Simulate using — Specify type of simulation to run**  
Interpreted execution (default) | Code generation

- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time and has a 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.

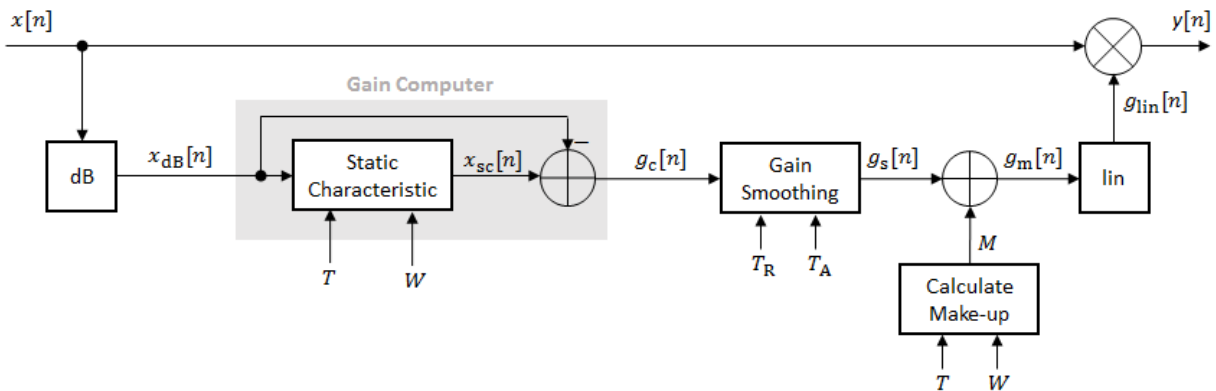
**Tunable:** No

## Block Characteristics

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

## Algorithms

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:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{\left(\frac{1}{R} - 1\right)\left(x_{dB} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \leq x_{dB} \leq \left(T + \frac{W}{2}\right) \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases} ,$$

where  $T$  is the threshold,  $R$  is the compression ratio, and  $W$  is the knee width.

If you specified a hard knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < T \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} \geq T \end{cases}$$

- 3 The computed gain,  $g_c[n]$ , is calculated as

$$g_c[n] = x_{sc}[n] - x_{dB}[n].$$

- 4  $g_c[n]$  is smoothed using specified attack and release time parameters:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) g_c[n], & g_c[n] \leq g_s[n-1] \\ \alpha_R g_s[n-1] + (1 - \alpha_R) g_c[n], & g_c[n] > g_s[n-1] \end{cases}$$

The attack time coefficient,  $\alpha_A$ , is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{FS \times T_A}\right).$$

The release time coefficient,  $\alpha_R$ , 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.  $F_s$  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:

$$M = -x_{sc}|_{x_{dB} = 0}.$$

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,  $g_s[n]$ :

$$g_m[n] = g_s[n] + M$$

- 7 The calculated gain in dB,  $g_{dB}[n]$ , is translated to a linear domain:

$$g_{lin}[n] = 10\left(\frac{g_m[n]}{20}\right)$$

- 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, 2012, pp. 399-408.

## Extended Capabilities

### C/C++ Code Generation

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

## **See Also**

Expander | Limiter | Noise Gate | compressor

## **Topics**

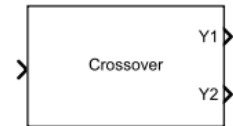
“Dynamic Range Control”

**Introduced in R2016a**

# Crossover Filter

Audio crossover filter

**Library:** Audio Toolbox / Filters



## 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.

## Ports

### Input

#### Port\_1 — Input signal

matrix | 1-D vector

- Matrix input -- Each column of the input is treated as an independent channel.
- 1-D vector input -- The input is treated as a single channel.

Data Types: single | double

### Output

#### Y1 — Output signal

matrix

Available if **Number of crossovers** is set to 1, 2, 3, or 4. Port **Y1** always corresponds to a lowpass filter.

Data Types: single | double

### Y2 — Output signal

matrix

Depending on the number of crossovers specified, port **Y2** outputs the original audio signal passed through a bandpass or highpass filter.

Available if **Number of crossovers** is set to 1, 2, 3, or 4.

Data Types: `single` | `double`

### Y3 — Output signal

matrix

Depending on the number of crossovers specified, port **Y3** corresponds to a bandpass or highpass filter of the original audio signal.

Available if **Number of crossovers** is set to 2, 3, or 4.

Data Types: `single` | `double`

### Y4 — Output signal

matrix

Available if **Number of crossovers** is set to 3 or 4.

Data Types: `single` | `double`

### Y5 — Output signal

matrix

Available if **Number of crossovers** is set to 4.

Data Types: `single` | `double`

## Parameters

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

### Number of crossovers — Number of magnitude response band crossings

1 (default) | 2 | 3 | 4

If you specify multiple crossovers, the corresponding **Crossover frequency (Hz)** and **Crossover order** parameters populate in the dialog box 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 bands
2	three bands
3	four bands
4	five bands

**Tunable:** No

**Crossover frequency (Hz) – Intersections of magnitude response bands**

100 (default) | real scalar in the range 20 to 20000

Crossover frequencies are the intersections of magnitude response bands of the individual two-band crossover filters used in the multiband crossover filter.

**Tunable:** Yes

**Crossover order – Order of individual crossover filters**

2 (default) | 1 | 3 | 4 | 5 | 6 | 7 | 8

The crossover filter order relates to the crossover filter slope in dB/octave:  $slope = N \times 6$ , where  $N$  is the crossover order.

**Tunable:** Yes

**View filter response – Open plot of magnitude response of each filter band**

button

The plot is updated automatically when parameters of the Crossover Filter block change.

**Tunable:** Yes

**Inherit sample rate from input – Specify source of input sample rate**

off (default) | on

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

**Tunable:** No

**Input sample rate (Hz) — Sample rate of input**

44100 (default) | positive scalar

**Tunable:** Yes**Dependencies**

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

**Simulate using — Specify type of simulation to run**

Interpreted execution (default) | Code generation

- **Interpreted execution** -- 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**.

**Tunable:** No

## Block Characteristics

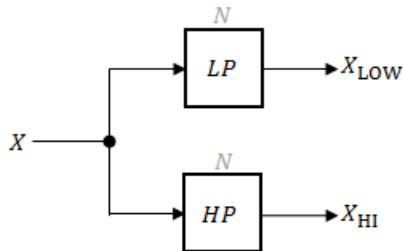
<b>Data Types</b>	double   single
<b>Direct Feedthrough</b>	no
<b>Multidimensional Signals</b>	no
<b>Variable-Size Signals</b>	yes
<b>Zero-Crossing Detection</b>	no

## 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

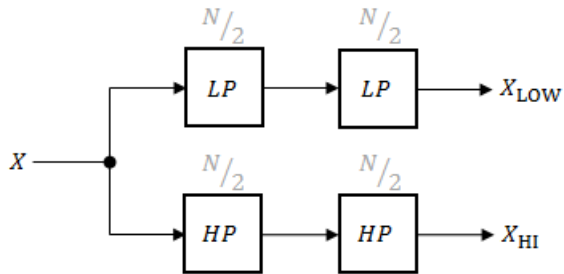
Odd-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.



*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.

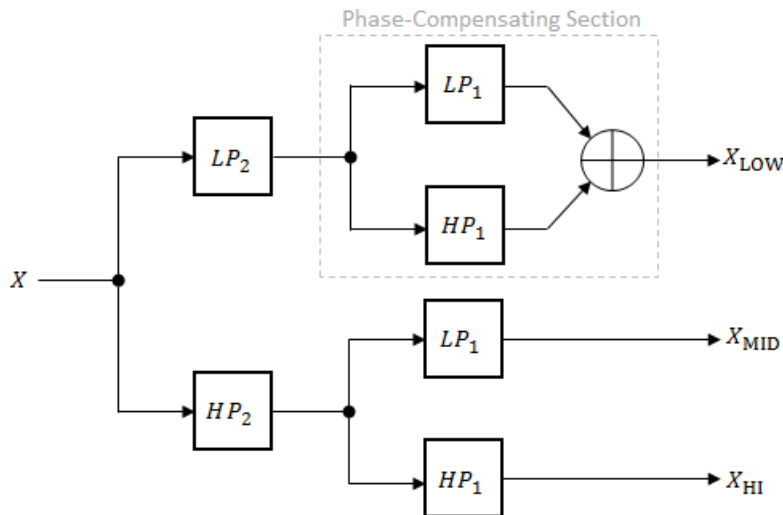


$LP$  and  $HP$  are Butterworth filters of order  $N/2$ , where  $N$  is the order of the overall filter. The filters are implemented as direct-form II transposed second-order sections.

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, 1987, pp. 239-245.



## **Extended Capabilities**

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

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

### **See Also**

`crossoverFilter`

### **Topics**

“Multiband Dynamic Range Compression”

**Introduced in R2016a**

# Expander

Dynamic range expander

**Library:** Audio Toolbox / Dynamic Range Control



## Description

The Expander block performs dynamic range expansion independently across each input channel. Dynamic range expansion attenuates the volume of quiet sounds below a given threshold. The block uses specified attack, release, and hold times to achieve a smooth applied gain curve.

## Ports

### Input

#### **x** — Input signal

matrix | 1-D vector

- Matrix input -- Each column of the input is treated as an independent channel.
- 1-D vector input -- The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

#### **R** — Ratio

scalar

### Dependencies

To enable this port, select **Specify from input port** for the “Ratio” on page 5-0 parameter.

Data Types: `single` | `double`

**T — Threshold (dB)**

scalar

**Dependencies**

To enable this port, select **Specify from input port** for the “Threshold (dB)” on page 5-0 parameter.

Data Types: single | double

**K — Knee width (dB)**

scalar

**Dependencies**

To enable this port, select **Specify from input port** for the “Knee width (dB)” on page 5-0 parameter.

Data Types: single | double

**AT — Attack time (s)**

scalar

**Dependencies**

To enable this port, select **Specify from input port** for the “Attack time (s)” on page 5-0 parameter.

Data Types: single | double

**RT — Release time (s)**

scalar

**Dependencies**

To enable this port, select **Specify from input port** for the “Release time (s)” on page 5-0 parameter.

Data Types: single | double

**HT — Hold time (s)**

scalar

**Dependencies**

To enable this port, select **Specify from input port** for the “Hold time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

## Output

### **Y — Output signal**

matrix

The Expander block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input -- The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input -- The block outputs an  $N$ -by-1 matrix (column vector), where  $N$  is the number of elements in the 1-D vector.

This port is unnamed until you select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

### **G — Gain applied to each input sample**

matrix

#### **Dependencies**

To enable this port, select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

## Parameters

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

#### **Main Tab**

### **Ratio — Expansion ratio**

5 (default) | scalar in the range 1 to 50 inclusive

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  $R = \frac{(y[n] - T)}{(x[n] - T)}$ , where

- $R$  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.

To specify **Ratio** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

#### **Threshold (dB) — Operation threshold**

-10 (default) | scalar in the range -140 to 0 inclusive

Operation threshold is the level below which gain is applied to the input signal.

To specify **Threshold (dB)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

#### **Knee width (dB) — Transition area in the compression characteristic**

0 (default) | scalar in the range 0 to 20

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  $(2 \times |x - T|) \leq W$ , where

- $y$  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.

To specify **Knee width (dB)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

**View static characteristic – Open static characteristic plot of dynamic range expander**

button

The plot is updated automatically when parameters of the Expander block change.

**Tunable:** Yes

**Attack time (s) – Time for applied gain to ramp up**

0.05 (default) | scalar in the range 0 to 4 inclusive

Attack time is the time the expander gain takes to rise from 10% to 90% of its final value when the input goes below the threshold. The **Attack time (s)** parameter smooths the applied gain curve.

To specify **Attack time (s)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

**Release time (s) – Time for applied gain to ramp down**

0.2 (default) | scalar in the range 0 to 4 inclusive

Release time is the time the expander gain takes to drop from 90% to 10% of its final value when the input goes above the threshold. The **Release time (s)** parameter smooths the applied gain curve.

To specify **Release time (s)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

**Hold time (s) – Time during which applied gain holds steady**

0.05 (default) | scalar in the range 0 to 4 inclusive

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.

To specify **Hold time (s)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

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

on (default) | off

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

**Tunable:** No**Input sample rate (Hz) — Sample rate of input**

44100 (default) | positive scalar

**Tunable:** Yes**Dependencies**

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

**Advanced Tab****Output gain (dB) — Gain applied on each input sample**

off (default) | on

When you select this parameter, an additional output port, **G**, is added to the block. The **G** port outputs the gain applied on each input channel in dB.

**Tunable:** No**Simulate using — Specify type of simulation to run**

Interpreted execution (default) | Code generation

- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time and has a 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**.

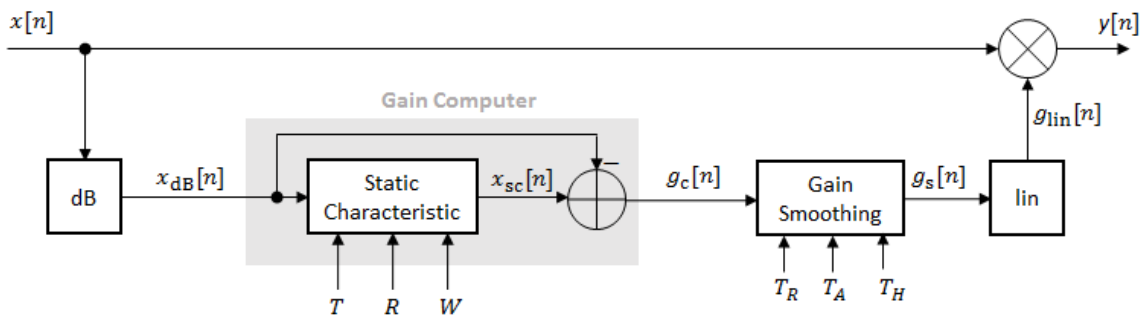
**Tunable:** No

## Block Characteristics

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

## Algorithms

The Expander 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 characteristic properties of the dynamic range expander to attenuate gain that is below the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:



$$x_{sc}(x_{dB}) = \begin{cases} T + (x_{dB} - T) \times R & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{(1-R)\left(x_{dB} - T - \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \leq x_{dB} \leq \left(T + \frac{W}{2}\right) \\ x_{dB} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases},$$

where  $T$  is the threshold,  $R$  is the expansion ratio, and  $W$  is the knee width.

If you specified a hard knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} T + (x_{dB} - T) \times R & x_{dB} < T \\ x_{dB} & x_{dB} \geq T \end{cases}$$

- 3 The computed gain,  $g_c[n]$ , is calculated as

$$g_c[n] = x_{sc}[n] - x_{dB}[n].$$

- 4  $g_c[n]$  is smoothed using specified attack, release, and hold time parameters:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) g_c[n] & (C_A > T_H) \text{ \& } (g_c[n] \leq g_s[n-1]) \\ g_s[n-1] & C_A \leq T_H \\ \alpha_R g_s[n-1] + (1 - \alpha_R) g_c[n] & (C_R > T_H) \text{ \& } (g_c[n] > g_s[n-1]) \\ g_s[n-1] & C_R \leq T_H \end{cases}$$

$C_A$  and  $C_R$  are hold counters for attack and release, respectively. The limit,  $T_H$ , is determined by the **Hold time (s)** parameter.

The attack time coefficient,  $\alpha_A$ , is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{FS \times T_A}\right).$$

The release time coefficient,  $\alpha_R$ , 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 **Input sample rate (Hz)** parameter.

- 5 The smoothed gain in dB,  $g_s[n]$ , is translated to a linear domain:

$$g_{\text{lin}}[n] = 10 \left( \frac{g_s[n]}{20} \right).$$

- 6 The output of the dynamic range expander is given as

$$y[n] = x[n] \times g_{\text{lin}}[n].$$

## 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, 2012, pp. 399-408.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

Compressor | Limiter | Noise Gate | expander

### Topics

"Dynamic Range Control"

**Introduced in R2016a**

# Graphic EQ

Standards-based graphic equalizer

**Library:** Audio Toolbox / Filters



## Description

The Graphic EQ block implements a graphic equalizer that can tune the gain on individual octave or fractional octave bands. The block filters the data independently across each input channel over time using the filter specifications. Center frequencies for bands in the graphic equalizer are based on the ANSI S1.11-2004 standard.

## Ports

### Input

#### Port\_1 — Input signal

matrix | 1-D vector

- Matrix input -- Each column of the input is treated as an independent channel.
- 1-D vector input -- The input is treated as a signal channel.

Data Types: `single` | `double`

### Output

#### Port\_1 — Output signal

matrix

The Graphic EQ block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input -- The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input -- The block outputs an  $N$ -by-1 matrix (column vector), where  $N$  is the number of elements in the 1-D vector input.

Data Types: `single` | `double`

## Parameters

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

### **EQ Order — Order of individual equalizer bands**

2 (default) | positive even integer

Specify the order of individual equalizer bands as a positive even integer. All equalizer bands have the same order.

**Tunable:** Yes

### **Bandwidth — Filter bandwidth (octaves)**

1 octave (default) | 2/3 octave | 1/3 octave

Specify the filter bandwidth as 1 octave, 2/3 octave, or 1/3 octave.

The ANSI S1.11-2004 standard defines the center and edge frequencies of your equalizer. The ISO 266:1997(E) standard specifies corresponding preferred frequencies for labeling purposes.

#### **1-Octave Bandwidth**

Center frequencies	32 63 126 251 501 1000 1995 3981 7943 15849
Edge frequencies	22 45 89 178 355 708 1413 2818 5623 1122 22387
Preferred frequencies	31.5 63 125 250 500 1000 2000 4000 8000 16000

#### **2/3-Octave Bandwidth**

Center frequencies	25 40 63 100 158 251 398 631 1000 1585 2512 3981 6310 10000 15849
Edge frequencies	20 32 50 79 126 200 316 501 794 1259 1995 3162 5012 7943 12589 19953
Preferred frequencies	25 40 63 100 160 250 400 630 1000 1600 2500 4000 6300 10000 16000

### 1/3-Octave Bandwidth

Center frequencies	25 32 40 50 63 79 100 126 158 200 251 316 398 501 631 794 1000 1259 1585 1995 2512 3162 3981 5012 6310 7943 10000 12589 15849 19953
Edge frequencies	22 28 35 45 56 71 89 112 141 178 224 282 355 447 562 708 891 1122 1413 1778 2239 2818 3548 4467 5623 7079 8913 11220 14125 17783 22387
Preferred frequencies	25 31.5 40 50 63 80 100 125 160 200 250 315 400 500 630 800 1000 1250 1600 2000 2500 3150 4000 5000 6300 8000 10000 12500 16000 20000

**Tunable:** Yes

### Structure — Type of implementation

Cascade (default) | Parallel

Specify the type of implementation as Cascade or Parallel. See “Algorithms” on page 5-63 and “Graphic Equalization” for information about these implementation structures.

**Tunable:** No

### Gains — Gain of each octave or fractional octave band (dB)

0 | scalar

Specify the gain of each octave or fractional octave band in dB. The number and position of filters in the graphic equalizer depends on the **Bandwidth** on page 5-0 parameter.

**Tunable:** Yes

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

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, specify the sample rate in **Input sample rate (Hz)** on page 5-0 .

**Tunable:** No

**Input sample rate (Hz) — Sample rate of input**  
44100 (default) | scalar

**Tunable:** Yes

#### **Dependencies**

To enable this parameter, clear the **Inherit sample rate from input** on page 5-0 parameter.

**Simulate using — Specify type of simulation to run**  
Code generation (default) | Interpreted execution

- **Interpreted execution** -- 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 faster than **Interpreted execution**.

**Tunable:** No

## **Block Characteristics**

<b>Data Types</b>	double   single
-------------------	-----------------

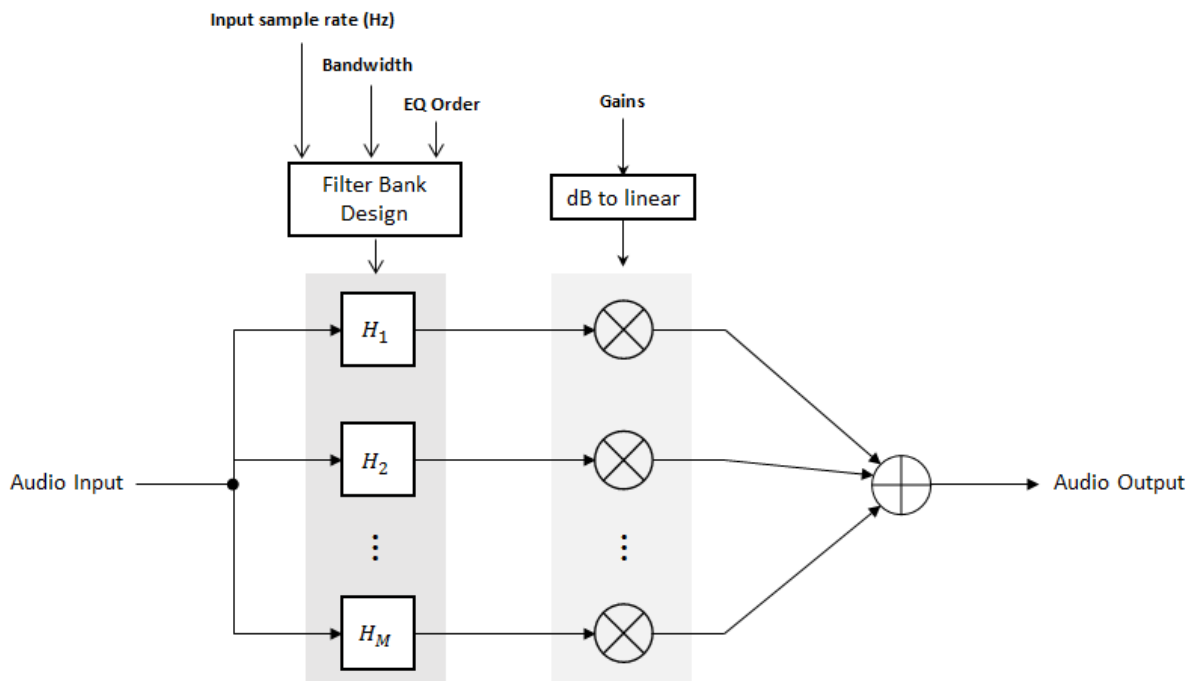
---

<b>Direct Feedthrough</b>	no
<b>Multidimensional Signals</b>	no
<b>Variable-Size Signals</b>	yes
<b>Zero-Crossing Detection</b>	no

## Algorithms

The implementation of your graphic equalizer depends on the **Structure** on page 5-0 parameter. See “Graphic Equalization” for a discussion of the pros and cons of the parallel and cascade implementations. Refer to the following sections to understand how these algorithms are implemented in Audio Toolbox.

## Parallel Structure



### Filter Bank Design

The parallel implementation designs the individual equalizers using the `octaveFilter design` method and spaces them on the spectrum according to the ANSI S1.11-2004 standard.

If you set the **Input sample rate (Hz)** parameter so that the Nyquist frequency (**Input sample rate (Hz)**/2) is less than the final bandpass edge defined by the ANSI S1.11-2004 standard, then:

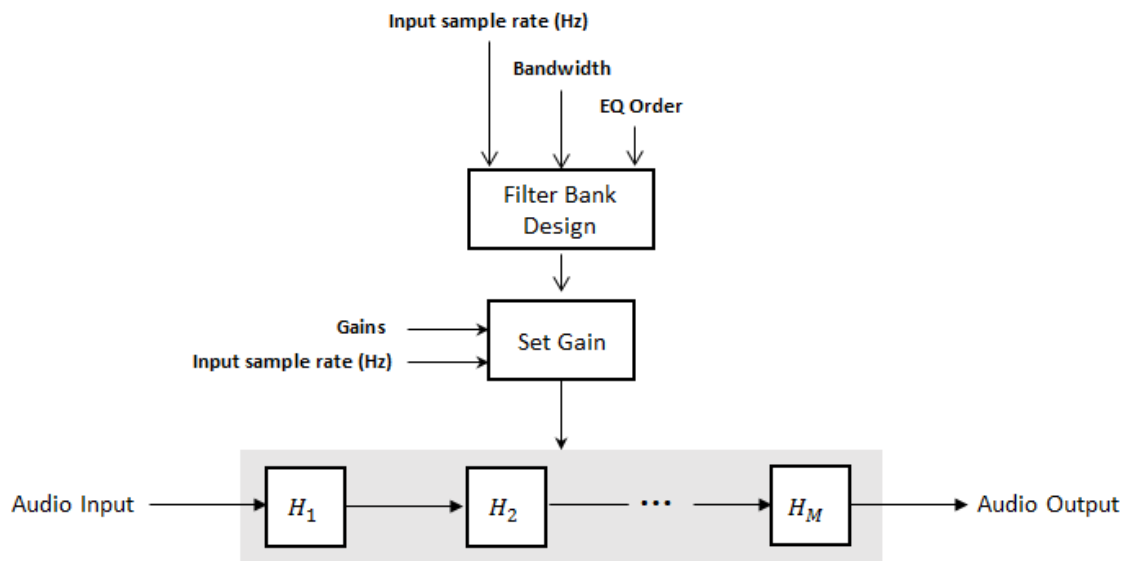
- The final bandpass filter is the one whose upper bandpass edge is less than the Nyquist frequency.
- The final filter is implemented as a highpass filter designed by the `designParamEQ` function.



## Real-Time Computation

- 1 The input signal is fed into a filterbank of  $M$  filters, where  $M$  depends on the specified **Bandwidth** and **Input sample rate (Hz)** parameters.
- 2 Each branch of the filterbank is multiplied by the linear form of the corresponding element of the **Gains** parameter.
- 3 The branches are summed and the output signal is returned.

## Cascade Structure



### Filter Bank Design

The cascade implementation designs the graphic equalizer filter bank using the `multibandParametricEQ` System object.

### Gain Setting

If the **EQ Order** on page 5-0 parameter is set to 2, then a gain correction is calculated according to [1]. The gain correction is independent of the requested gains. The gain

correction is recomputed during the real-time processing only if the **Input sample rate (Hz)** parameter is modified.

If the **EQ Order** parameter is not set to 2, no gain correction is applied and the requested gains are passed on to the `multibandParametricEQ` object.

### Real-Time Computation

The input signal is fed into a cascade of  $M$  biquad filters, where  $M$  depends on the specified **Bandwidth** and **Input sample rate (Hz)** parameters.

## References

- [1] Oliver, Richard J., and Jean-Marc Jot. "Efficient Multi-Band Digital Audio Graphic Equalizer with Accurate Frequency Response Control." Presented at the 139th Convention of the AES, New York, October 2015.
- [2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters*. ANSI S1.11-2004. Melville, NY: Acoustical Society of America, 2009.
- [3] International Organization for Standardization. *Acoustics -- Preferred frequencies*. ISO 266:1997(E). Second Edition. 1997.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`ParametricEQ` | `designParamEQ` | `designShelvingEQ` | `designVarSlopeFilter` | `graphicEQ` | `multibandParametricEQ`

### Topics

"Graphic Equalization"

“Equalization”

**Introduced in R2017b**

## Limiter

Dynamic range limiter

**Library:** Audio Toolbox / Dynamic Range Control



## 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. The block uses specified attack and release times to achieve a smooth applied gain curve.

## Ports

### Input

**x — Input signal**

1-D vector | matrix

- Matrix input -- Each column of the input is treated as an independent channel.
- 1-D vector input -- The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

**T — Threshold (dB)**

scalar

### Dependencies

To enable this port, select **Specify from input port** for the “Threshold (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

**K — Knee width (dB)**

scalar

**Dependencies**

To enable this port, select **Specify from input port** for the “Knee width (dB)” on page 5-0 parameter.

Data Types: single | double

**AT — Attack time (s)**

scalar

**Dependencies**

To enable this port, select **Specify from input port** for the “Attack time (s)” on page 5-0 parameter.

Data Types: single | double

**RT — Release time (s)**

scalar

**Dependencies**

To enable this port, select **Specify from input port** for the “Release time (s)” on page 5-0 parameter.

Data Types: single | double

**Output****Y — Output signal**

matrix

The Limiter block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input -- The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input -- The block outputs an  $N$ -by-1 matrix (column vector), where  $N$  is the number of elements in the 1-D vector.

This port is unnamed until you select the **Output gain (dB)** parameter.

Data Types: single | double

### **G — Gain applied to each input sample**

matrix

#### **Dependencies**

To enable this port, select the **Output gain (dB)** parameter.

Data Types: single | double

## **Parameters**

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

#### **Main Tab**

#### **Threshold (dB) — Operation threshold**

-10 (default) | scalar in the range -50 to 0 inclusive

Operation threshold is the level above which gain is applied to the input signal.

To specify **Threshold (dB)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

#### **Knee width (dB) — Transition area in the limiter characteristic**

0 (default) | scalar in the range 0 to 20 inclusive

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  $(2 \times |x - T|) \leq W$ , where

- $y$  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.

To specify **Knee width (dB)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

### **View static characteristic — Open static characteristic plot of dynamic range limiter**

button

The plot is updated automatically when parameters of the Limiter block change.

**Tunable:** Yes

### **Attack time (s) — Time for applied gain to ramp up**

0 (default) | scalar in the range 0 to 4 inclusive

Attack time is the time the limiter gain takes to rise from 10% to 90% of its final value when the input goes above the threshold. The **Attack time (s)** parameter smooths the applied gain curve.

To specify **Attack time (s)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

### **Release time (s) — Time for applied gain to ramp down**

0.2 (default) | scalar in the range 0 to 4 inclusive

Release time is the time the limiter gain takes to drop from 90% to 10% of its final value when the input goes below the threshold. The **Release time (s)** parameter smooths the applied gain curve.

To specify **Release time (s)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

### **Make-up gain mode — Make-up gain mode**

Property (default) | Auto

- **Property** -- Make-up gain is set to the value specified by the **Make-up gain (dB)** parameter.
- **Auto** -- Make-up gain is applied at the output of the Limiter block such that a steady-state 0 dB input has a 0 dB output.

**Tunable:** No

### **Make-up gain (dB) — Applied make-up gain**

0 (default) | scalar in the range -10 to 24 inclusive

Make-up gain compensates for gain lost during limiting. It is applied at the output of the Limiter block.

**Tunable:** Yes

### **Dependencies**

To enable this parameter, set the **Make-up gain mode** parameter to **Property**.

### **Inherit sample rate from input — Specify source of input sample rate**

on (default) | off

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

**Tunable:** No

### **Input sample rate (Hz) — Specify input sample rate**

44100 (default) | positive scalar

**Tunable:** Yes

### **Dependencies**

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

### **Advanced Tab**

### **Output gain (dB) — Gain applied on each input sample**

off (default) | on



When you select this parameter, an additional output port, **G**, is added to the block. The **G** port outputs the gain applied on each input channel in dB.

**Tunable:** No

### Simulate using — Specify type of simulation to run

Interpreted execution (default) | Code generation

- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time and has a 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**.

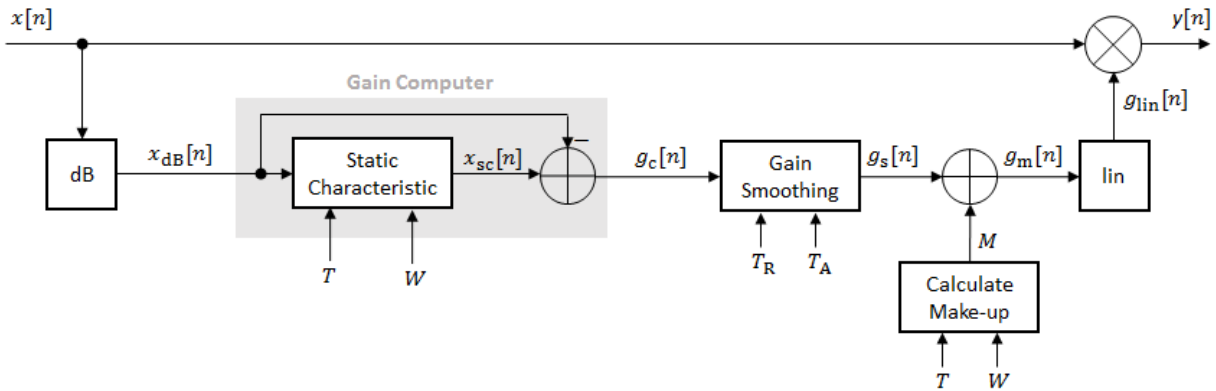
**Tunable:** No

## Block Characteristics

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

## Algorithms

The Limiter block processes a signal frame by frame and element by element.



- 1 The  $N$ -point signal,  $x[n]$ , is converted to decibels:
 
$$x_{\text{dB}}[n] = 20 \times \log_{10}|x[n]|$$
- 2  $x_{\text{dB}}[n]$  passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range limiter to brickwall gain that is above the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:

$$x_{\text{sc}}(x_{\text{dB}}) = \begin{cases} x_{\text{dB}} & x_{\text{dB}} < \left(T - \frac{W}{2}\right) \\ x_{\text{dB}} - \frac{\left(x_{\text{dB}} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \leq x_{\text{dB}} \leq \left(T + \frac{W}{2}\right) \\ T & x_{\text{dB}} > \left(T + \frac{W}{2}\right) \end{cases},$$

where  $T$  is the threshold and  $W$  is the knee width.

If you specified a hard knee, the gain computer has the following static characteristic:

$$x_{\text{sc}}(x_{\text{dB}}) = \begin{cases} x_{\text{dB}} & x_{\text{dB}} < T \\ T & x_{\text{dB}} \geq T \end{cases}$$

- 3 The computed gain,  $g_c[n]$ , is calculated as

$$g_c[n] = x_{\text{sc}}[n] - x_{\text{dB}}[n].$$

- 4  $g_c[n]$  is smoothed using specified attack and release time parameters:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) g_c[n], & g_c[n] \leq g_s[n-1] \\ \alpha_R g_s[n-1] + (1 - \alpha_R) g_c[n], & g_c[n] > g_s[n-1] \end{cases}$$

The attack time coefficient,  $\alpha_A$ , is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{F_S \times T_A}\right).$$

The release time coefficient,  $\alpha_R$ , 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 **Attack time (s)** parameter.  $T_R$  is the release time period, specified by the **Release time (s)** parameter.  $F_S$  is the input sampling rate, specified by the **Inherit sample rate from input** or **Input sample rate (Hz)** parameter.

- 5 If the **Make-up gain (dB)** parameter is set to **Auto**, the make-up gain is calculated as the negative of the computed gain for a 0 dB input:

$$M = -x_{sc}(x_{dB} = 0)$$

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)** 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,  $g_s[n]$ :

$$g_m[n] = g_s[n] + M$$

- 7 The calculated gain in dB,  $g_m[n]$ , is translated to a linear domain:

$$g_{lin}[n] = 10\left(\frac{g_m[n]}{20}\right)$$

- 8 The output of the dynamic range limiter 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, 2012, pp. 399-408.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

Compressor | Expander | Noise Gate | limiter

### Topics

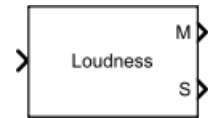
"Dynamic Range Control"

**Introduced in R2016a**

# Loudness Meter

Standard-compliant loudness measurements

**Library:** Audio Toolbox / Measurements



## Description

The Loudness Meter block measures the loudness and true-peak of an audio signal based on EBU R 128 and ITU-R BS.1770-4 standards.

## Ports

### Input

#### Port\_1 — Input signal

matrix | 1-D vector

- Matrix input -- Each column of the input is treated as an independent channel. If you use the default **Channel weights**, specify the input channels in order: [Left, Right, Center, Left surround, Right surround].
- 1-D vector input -- The input is treated as a single channel.

Data Types: `single` | `double`

### Output

#### M — Momentary loudness measurement

column vector

The block outputs a column vector with the same data type and number of rows as the input signal.

Data Types: `single` | `double`

**S — Short-term loudness measurement**

column vector

The block outputs a column vector with the same data type and number of rows as the input signal.

Data Types: `single` | `double`**TP — True-peak value**

real scalar

The block outputs a real scalar with the same data type as the input signal.

**Dependencies**

To enable this port, select the **Output true-peak value** parameter.

Data Types: `single` | `double`

## Parameters

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

**Channel weights — Linear weighting applied to each input channel**`[1, 1, 1, 1.41, 1.41]` (default) | nonnegative row vector

The number of elements of the row vector must be equal to or greater than the number of input channels. Excess values in the vector are ignored.

The default channel weights follow the ITU-R BS.1170-4 standard. To use the default channel weights, specify the input to the Loudness Meter block as a matrix whose columns correspond to channels in this order: [Left, Right, Center, Left surround, Right surround].

It is a best practice to specify the channel weights in order: [Left, Right, Center, Left surround, Right surround].

**Tunable:** Yes**Use relative scale for loudness measurements — Specify block to output loudness measurements relative to target level**

off (default) | on

- On — The loudness measurements are relative to the value specified by **Target loudness level (LUFS)**. The output of the block is returned in loudness units (LU).
- Off — The loudness measurements are absolute, and returned in loudness units full scale (LUFS).

**Tunable:** No

**Target loudness level (LUFS) — Reference level for relative loudness measurements**

-23 (default) | real scalar

For example, if the **Target loudness level (LUFS)** is -23, then a loudness value of -24 LUFS is reported as -1 LU.

**Tunable:** Yes

**Dependencies**

To enable this parameter, select the **Use relative scale for loudness measurements** parameter.

**Output true-peak value — Add output port for true-peak value**

off (default) | on

When you select this parameter, an additional output port, **TP**, is added to the block. The **TP** port outputs the true-peak value of the input frame.

**Tunable:** No

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

on (default) | off

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

**Tunable:** No

**Input sample rate (Hz) — Sample rate of input**

44100 (default) | scalar

**Tunable:** Yes

**Dependencies**

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

**Simulate using — Specify type of simulation to run**

Code generation (default) | Interpreted execution

- **Code generation** -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is comparable to **Interpreted execution**.
- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than **Code generation**. In this mode, you can debug the source code of the block.

**Tunable:** No

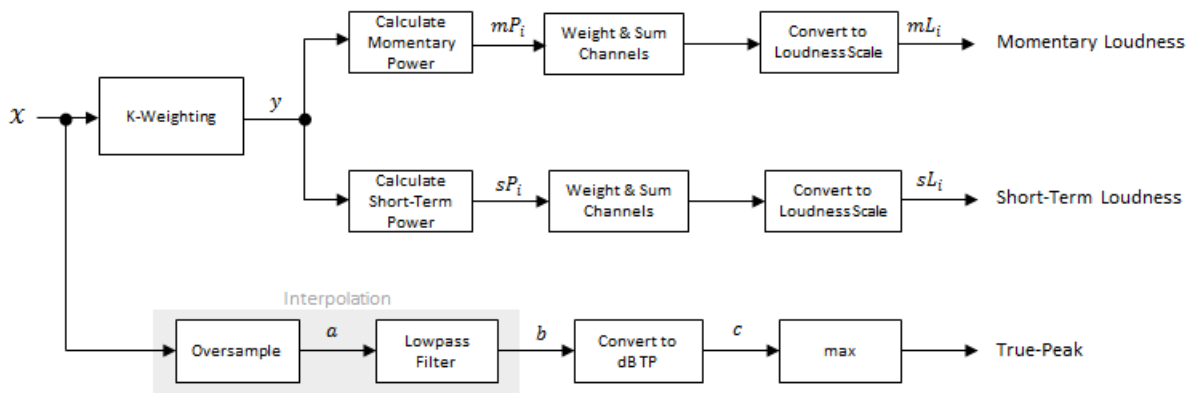
## Block Characteristics

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

## Algorithms

The Loudness Meter block calculates the momentary loudness, short-term loudness, and true-peak value of an audio signal. You can specify any number of channels and nondefault channel weights used for loudness measurements. The block algorithm is described for the general case of  $n$  channels and default channel weights.





## Loudness Measurements

The input channels,  $x$ , pass through a K-weighted filter implemented using the algorithm of the Weighting Filter block. The K-weighted filter shapes the frequency spectrum to reflect perceived loudness.

### Momentary Loudness

- 1 The K-weighted channels,  $y$ , are divided into 0.4-second segments with 0.3-second overlap. If the required number of samples have not been collected yet, the Loudness Meter block returns the last computed value for momentary loudness. If enough samples have been collected, then the power (mean square) of each segment of the K-weighted channels is calculated:

$$mP_i = \frac{1}{w} \sum_{k=1}^w y_i^2[k]$$

- $mP_i$  is the momentary power of the  $i$ th segment.
  - $w$  is the segment length in samples.
- 2 The momentary loudness,  $mL$ , is computed for each segment:

$$mL_i = -0.691 + 10 \log_{10} \left( \sum_{c=1}^n G_c \times mP_{(i,c)} \right) \text{ LUFS}$$

- $G_c$  is the weighting for channel  $c$ .

$mL$  is the momentary loudness returned by your Loudness Meter block.

### Short-Term Loudness

- 1 The K-weighted channels,  $y$ , are divided into 3-second segments with 2.9-second overlap. If the required number of samples have not been collected yet, the Loudness Meter block returns the last computed values for short-term loudness and loudness range. If enough samples have been collected, then the power (mean square) of each K-weighted channel is calculated:

$$sP_i = \frac{1}{w} \sum_{k=1}^w y_i^2[k]$$

- $sP_i$  is the short-term power of the  $i$ th segment of a channel.
  - $w$  is the segment length in samples.
- 2 The short-term loudness,  $sL$ , is computed for each segment:

$$sL_i = -0.691 + 10 \log_{10} \left( \sum_{c=1}^n G_c \times sP_{(i,c)} \right) \text{ LUFS}$$

- $G_c$  is the weighting for channel  $c$ .

$sL$  is the short-term loudness returned by your Loudness Meter block.

### True-Peak

The *true-peak* measurement considers only the current input frame of a call to your loudness meter.

- 1 The signal is oversampled to at least 192 kHz. To optimize processing, the input sample rate determines the exact oversampling. An input sample rate below 750 Hz is not considered.

Input Sample Rate (kHz)	Upsample Factor
[0.75,1.5)	256
[1.5,3)	128
[3,6)	64
[6,12)	32

Input Sample Rate (kHz)	Upsample Factor
[12,24)	16
[24,48)	8
[48,96)	4
[96,192)	2
[192,∞)	not required

- 2 The oversampled signal,  $a$ , passes through a lowpass filter with a half-polyphase length of 12 and stopband attenuation of 80 dB. The filter design uses `designMultirateFIR`.
- 3 The filtered signal,  $b$ , is rectified and converted to the dB TP scale:
 
$$c = 20 \times \log_{10}(|b|)$$
- 4 The true-peak is determined as the maximum of the converted signal,  $c$ .

## References

- [1] International Telecommunication Union; Radiocommunication Sector. *Algorithms to Measure Audio Programme Loudness and True-Peak Audio Level*. ITU-R BS.1770-4. 2015.
- [2] European Broadcasting Union. *Loudness Normalisation and Permitted Maximum Level of Audio Signals*. EBU R 128. 2014.
- [3] European Broadcasting Union. *Loudness Metering: 'EBU Mode' Metering to Supplement EBU R 128 Loudness Normalization*. EBU R 128 Tech 3341. 2014.

## Extended Capabilities

### C/C++ Code Generation

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

## **See Also**

`integratedLoudness` | `loudnessMeter`

**Introduced in R2016b**

# MIDI Controls

Output values from controls on MIDI control surface

**Library:** Audio Toolbox / Sources  
DSP System Toolbox / Sources



## Description

The MIDI Controls block outputs values from controls on a MIDI control surface in real time. Use the MIDI Controls block to interact with your audio processing model.

The MIDI Controls block combines the functionality of the general MIDI functions in MATLAB: `midicontrols`, `midiread`, `midisync`. Use the MATLAB `midiid` command to discover MIDI device names or MIDI device control numbers.

## Ports

### Output

#### Port\_1 — Output signal

matrix

The output size of the MIDI Controls block is determined by the **MIDI controls** and **MIDI control numbers** parameters.

The output data type is determined by the **Output mode** parameter.

Data Type	Output Mode
double	Normalized (0-1)
uint8	RAW MIDI (0-127)

Data Types: double | uint8

## Parameters

### **MIDI device — MIDI control surface your block listens to**

Default (default) | Specify other

To set the default MIDI device, use the `setpref` function. For example, if the device is named BCF2000, at the MATLAB command line, enter:

```
setpref('midi','DefaultDevice','BCF2000');
```

### **MIDI device name — Device name of MIDI control surface your block listens to**

character vector

The MIDI device name is assigned by the device manufacturer or host operating system, and specified as a character vector. Use `midiid` to interactively identify your MIDI device.

To enable this parameter, set **MIDI device** to Specify other.

### **MIDI controls — Specify if block responds to all controllers or specific controllers on MIDI surface**

Respond to any control (default) | Respond to specified controls

This parameter also determines the size of the block output port. If you choose Respond to any control, then the block output is a scalar corresponding to the value of the most recently manipulated control.

### **MIDI control numbers — Control numbers associated with MIDI surface controllers that your block responds to**

0 (default) | integer | array of integers

Use `midiid` to interactively identify the control numbers of your MIDI device. This parameter is available when you set **MIDI controls** to Respond to specified controls.

### **Initial values — Control numbers associated with MIDI surface controllers that your block responds to**

0 (default) | scalar | array

If you specify **Initial values** as a scalar, all controls specified by **MIDI control numbers** are assigned that value.

If you specify **Initial values** as an array, the array must be the same size as **MIDI control numbers**.

**Send initial values to device at start – Synchronize MIDI surface with values specified initial values**

off (default) | on

Select this parameter to synchronize a MIDI device with values specified by the **Initial values** when simulation starts. If your MIDI device can receive and respond to messages, it adjusts its controls as specified. This parameter is valid only when **MIDI controls** is set to Respond to specified controls.

Many MIDI devices are not bidirectional. Selecting this parameter with a unidirectional device has no effect. The MIDI Controls block 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.

**Output Mode – Output mode for MIDI control value**

Normalized (0-1) (default) | RAW MIDI (0-127)

## Block Characteristics

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

## Extended Capabilities

### C/C++ Code Generation

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

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

### See Also

#### Functions

`midicontrols` | `mididiid` | `midiread` | `midisync`

#### Topics

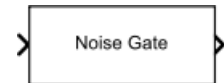
“MIDI Control Surface Interface”



# Noise Gate

Dynamic range gate

**Library:** Audio Toolbox / Dynamic Range Control



## Description

The Noise Gate block performs dynamic range gating independently across each input channel. Dynamic range gating suppresses signals below a given threshold. The block uses specified attack, release, and hold times to achieve a smooth applied gain curve.

## Ports

### Input

**x** — Input signal

matrix | 1-D vector

- Matrix input -- Each column of the input is treated as an independent channel.
- 1-D vector input -- The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: single | double

**T** — Threshold (dB)

scalar

### Dependencies

To enable this port, select **Specify from input port** for the “Threshold (dB)” on page 5-0 parameter.

Data Types: single | double

### **AT — Attack time (s)**

scalar

#### **Dependencies**

To enable this port, select **Specify from input port** for the “Attack time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

### **RT — Release time (s)**

scalar

#### **Dependencies**

To enable this port, select **Specify from input port** for the “Release time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

### **HT — Hold time (s)**

scalar

#### **Dependencies**

To enable this port, select **Specify from input port** for the “Hold time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

## **Output**

### **Y — Output signal**

matrix

The Noise Gate block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input -- The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input -- The block outputs an  $N$ -by-1 matrix (column vector), where  $N$  is the number of elements in the 1-D vector.

This port is unnamed until you select the **Output gain (dB)** parameter.

Data Types: single | double

### **G — Gain applied to each input sample**

matrix

#### **Dependencies**

To enable this port, select the **Output gain (dB)** parameter.

Data Types: single | double

## **Parameters**

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

#### **Main Tab**

### **Threshold (dB) — Operation threshold**

-10 (default) | scalar in the range -140 to 0 inclusive

Operation threshold is the level below which gain is applied to the input signal.

To specify **Threshold (dB)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

### **View static characteristic — Open static characteristic plot of dynamic range gate**

button

The plot is updated automatically when parameters of the Noise Gate block change.

**Tunable:** Yes

### **Attack time (s) — Time for applied gain to ramp up**

0.05 (default) | scalar in the range 0 to 4 inclusive

Attack time is the time the applied gain takes to rise from 10% to 90% of its final value when the input goes below the threshold. The **Attack time (s)** parameter smooths the applied gain curve.

To specify **Attack time (s)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

**Release time (s) — Time for applied gain to ramp down**

0.2 (default) | scalar in the range 0 to 4 inclusive

Release time is the time the applied gain takes to drop from 90% to 10% of its final value when the input goes above the threshold. The **Release time (s)** parameter smooths the applied gain curve.

To specify **Release time (s)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

**Hold time (s) — Time during which applied gain holds steady**

0.05 (default) | scalar in the range 0 to 4

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.

To specify **Hold time (s)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

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

on (default) | off

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

**Tunable:** No

**Input sample rate (Hz) — Specify input sample rate**

44100 (default) | scalar

**Tunable:** Yes

**Dependencies**

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

**Advanced Tab****Output gain (dB) – Gain applied on each input sample**

off (default) | on

When you select this parameter, an additional output port, **G**, is added to the block. The **G** port outputs the gain applied on each input channel in dB.

**Tunable:** No

**Simulate using – Specify type of simulation to run**

Interpreted execution (default) | Code generation

- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time and has a 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**.

**Tunable:** No

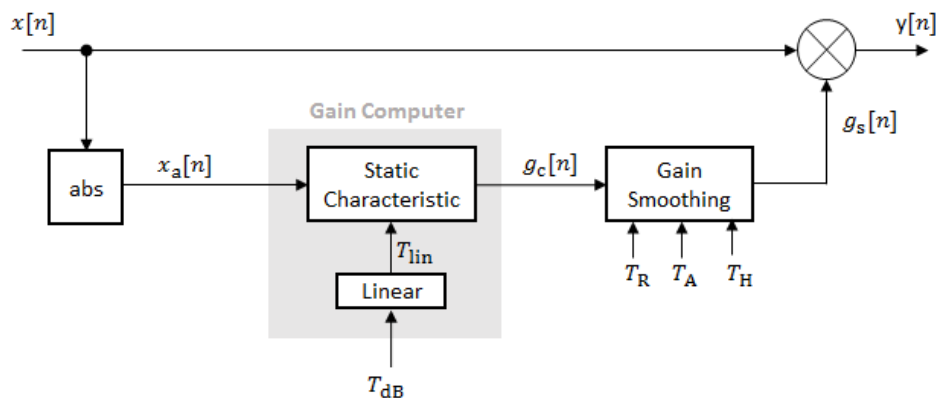
## Block Characteristics

<b>Data Types</b>	double   single
<b>Direct Feedthrough</b>	no
<b>Multidimensional Signals</b>	no
<b>Variable-Size Signals</b>	yes

<b>Zero-Crossing Detection</b>	no
--------------------------------	----

## Algorithms

The Noise Gate block processes a signal frame by frame and element by element.



- 1 The  $N$ -point signal,  $x[n]$ , is converted to magnitude:

$$x_a[n] = |x[n]|$$

- 2  $x_a[n]$  passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range gate to apply a brickwall gain for signal below the threshold:

$$g_c(x_a) = \begin{cases} 0 & x_a < T_{\text{lin}} \\ 1 & x_a \geq T_{\text{lin}} \end{cases}$$

$T_{\text{lin}}$  is the threshold property converted to a linear domain:

$$T_{\text{lin}} = 10^{(T_{\text{dB}}/20)} .$$

- 3 The computed gain,  $g_c[n]$ , is smoothed using specified attack, release, and hold time parameters:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) g_c[n] & \text{if } (C_A > T_H) \text{ \& } (g_c[n] \leq g_s[n-1]) \\ g_s[n-1] & \text{if } C_A \leq T_H \\ \alpha_R g_s[n-1] + (1 - \alpha_R) g_c[n] & \text{if } (C_R > T_H) \text{ \& } (g_c[n] > g_s[n-1]) \\ g_s[n-1] & \text{if } C_R \leq T_H \end{cases}$$

$C_A$  and  $C_R$  are hold counters for attack and release, respectively. The limit,  $T_H$ , is determined by the **Hold time (s)** parameter.

The attack time coefficient,  $\alpha_A$ , is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{F_S \times T_A}\right).$$

The release time coefficient,  $\alpha_R$ , 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 **Attack time (s)** parameter.  $T_R$  is the release time period, specified by the **Release time (s)** parameter.  $F_S$  is the input sampling rate, specified by the **Inherit sample rate from input** or **Input sample rate (Hz)** parameter.

- 4 The output of the dynamic range gate is given as

$$y[n] = x[n] \times g_s[n].$$

## 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, 2012, pp. 399-408.

## Extended Capabilities

### C/C++ Code Generation

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

## **See Also**

Compressor | Expander | Limiter | noiseGate

## **Topics**

“Dynamic Range Control”

**Introduced in R2016a**



# Octave Filter

Octave-band and fractional octave-band filter

**Library:** Audio Toolbox / Filters



## Description

The Octave Filter block performs octave-band or fractional octave-band filtering independently across each input channel. An octave-band is a frequency band where the highest frequency is twice the lowest frequency. Octave-band and fractional octave-band filters are commonly used to mimic how humans perceive loudness. Octave filters are best understood when viewed on a logarithmic scale, which models how the human ear weights the spectrum.

## Ports

### Input

#### **x** — Input signal

matrix | 1-D vector

- Matrix input -- Each column of the input is treated as an independent channel.
- 1-D vector input -- The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

#### **CF** — Center frequency (Hz)

scalar in the range 3 to 22,000 inclusive

### Dependencies

To enable this port, select **Specify from input port** for the “Center frequency (Hz)” on page 5-0 parameter.

Data Types: `single` | `double`

## Output

### Port\_1 — Output signal

matrix

The Octave Filter block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input -- The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input -- The block outputs an  $N$ -by-1 matrix (column vector), where  $N$  is the number of elements in the 1-D vector.

Data Types: `single` | `double`

## Parameters

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

### Filter order — Order of octave filter

6 (default) | even integer

**Tunable:** No

### Center frequency (Hz) — Center frequency of octave filter

1000 (default) | scalar in the range 3 to 22,000 inclusive

- The maximum center frequency is the value that causes the upper band edge to be equal to the Nyquist frequency,  $F_s/2$ . Frequencies above this value are saturated.
- The minimum center frequency is the value that causes the lower band edge to be equal to 1 Hz. Frequencies below this value are quantized to 1 Hz.

To specify **Center frequency (Hz)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

**Bandwidth — Filter bandwidth in octaves**

1 octave (default) | 2/3 octave | 1/2 octave | 1/3 octave | 1/6 octave | 1/12 octave | 1/24 octave | 1/48 octave

**Tunable:** Yes

**Oversample the input by 2 for this filter — Oversample toggle**

off (default) | on

- off -- The Octave Filter block runs at the input sample rate.
- on -- The Octave Filter block runs at two times the input sample rate. Oversampling minimizes the frequency warping effects introduced by the bilinear transformation. An FIR halfband interpolator implements oversampling before octave filtering. A halfband decimator reduces the sample rate back the input sampling rate after octave filtering.

**Tunable:** No

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

off (default) | on

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

**Tunable:** No

**Input sample rate (Hz) — Sample rate of input**

44100 (default) | scalar

**Tunable:** Yes

**Dependencies**

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

**Simulate using — Specify type of simulation to run**

Code generation (default) | Interpreted execution

- Code generation -- Simulate the 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.

- **Interpreted execution** -- Simulate the model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than Code generation. In this mode, you can debug the source code of the block.

**Tunable:** No

**Mask for attenuation limits — Create a mask for filter response visualization**

No mask (default) | Class 0 | Class 1 | Class 2

The mask attenuation limits are defined in the ANSI S1.11-2004 standard.

- If the mask is green, the design is compliant.
- If the mask is red, the design breaks compliance.

**Tunable:** Yes

**Visualize filter response — Open plot to visualize magnitude response and compliance mask**

button

A 2048-point FFT is used to calculate the magnitude response.

**Tunable:** Yes

## Block Characteristics

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

## More About

### Band Edge

A band edge frequency refers to the lower or upper edge of the passband of a bandpass filter.

### Center Frequency of Octave Filter

The center frequency of an octave filter is the geometric mean of the lower- and upper-band edge frequencies.

## Algorithms

### Octave Bandwidth to Band Edge Conversion

The Octave Filter block uses the specified center frequency and filter bandwidth in octaves to determine the normalized band edges [2].

First the block computes the upper and lower band edge frequencies:

$$f_{pa} = f_c \times G^{-1/2b}$$

$$f_{pb} = f_c \times G^{1/2b}$$

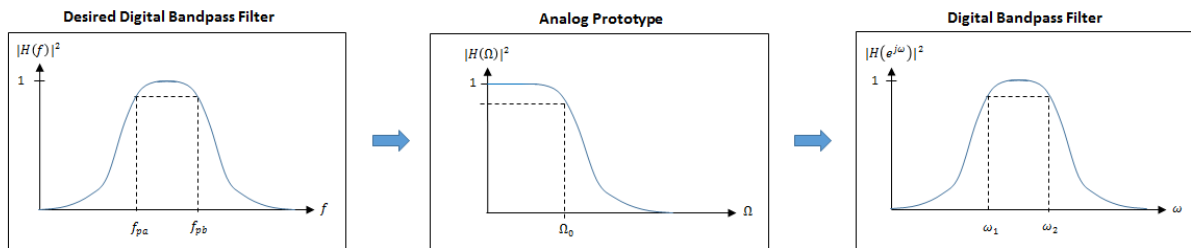
- $f_c$  is the normalized center frequency specified by the **Center frequency (Hz)** parameter.
- $b$  is the octave bandwidth specified by the **Bandwidth** parameter. For example, if **Bandwidth** is specified as 1/3 octave, the value of  $b$  is 3.
- $G$  is a conversion constant:

$$G = 10^{3/10}.$$

## Digital Filter Design

The Octave Filter block implements a higher-order digital bandpass filter design method as specified in [1].

In this design method, a desired digital bandpass filter maps to a Butterworth lowpass analog prototype, which is then mapped back to a digital bandpass filter:



- 1 The analog Butterworth filter is expressed as a cascade of second-order sections:

$H(s) = H_1(s)H_2(s)\dots H_{2N}(s)$ , where:

- $$H_i(s) = \frac{1}{1 - 2\frac{s}{\Omega_0}\cos\theta_i + \frac{s^2}{\Omega_0^2}}, \quad i = 1, 2, \dots, 2N$$
- $$\theta_i = \frac{\pi}{2N}(N - 1 + 2i), \quad i = 1, 2, \dots, N, \dots, 2N$$

$N$  is the filter order specified by the **Filter order** parameter.

- 2 The analog Butterworth filter is mapped to a digital filter using a bandpass version of the bilinear transformation:

$$s = \frac{1 - cz^{-1} + z^{-2}}{1 - z^{-2}},$$

where

$$c = \frac{\sin(\omega_{pa} + \omega_{pb})}{\sin\omega_{pa} + \sin\omega_{pb}}.$$

This mapping results in the following substitution:

$$\Omega_0 = \frac{c - \cos\omega_{pb}}{\sin\omega_{pb}}$$

- 3 The analog prototype is evaluated:

$$H_1(z) = \frac{1}{1 - 2\frac{s}{\Omega_0}\cos\theta_1 + \frac{s^2}{\Omega_0^2}} \Bigg|_s = \frac{1 - 2cz^{-1} + z^{-2}}{1 - z^{-2}}$$

Because  $s$  is second-order in  $z$ , the bandpass version of the bilinear transformation is fourth-order in  $z$ .

## References

- [1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 2010.
- [2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters: ANSI S1.11-2004*. Melville, NY: Acoustical Society of America, 2009.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

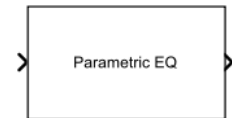
Weighting Filter | octaveFilter | weightingFilter

**Introduced in R2016b**

## Parametric EQ

Second-order parametric equalizer filter

**Library:** Audio Toolbox / Filters



### Description

The Parametric EQ 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 filter is designed using `designParamEQ` and implemented using `dsp.BiquadFilter`.

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** parameter. The number of channels must remain constant.

### Ports

#### Input

**x — Input signal**

matrix | 1-D vector

- Matrix input -- Each column of the input is treated as an independent channel.
- 1-D vector input -- The input is treated as a signal channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

**Fc — Center frequency (Hz)**

scalar



Specify the center frequency as a positive scalar that is less than half the sample rate of the input signal.

**Dependencies**

To enable this port, select **Specify from input port** for the **Center Frequency (Hz)** parameter.

Data Types: `single` | `double`

**BW — Bandwidth (Hz)**

scalar

Specify the filter bandwidth as a positive scalar that is less than or equal to half the sample rate of the input signal and 20 kHz.

**Dependencies**

To enable this port, select Bandwidth and Center Frequency for the **Filter specification** and **Specify from input port** for the **Filter Bandwidth (Hz)** parameter.

Data Types: `single` | `double`

**GdB — Peak or dip gain (dB)**

scalar

Specify the peak or dip gain in dB as a scalar.

**Dependencies**

To enable this port, select **Specify from input port** for the **Peak Gain (dB)** parameter.

Data Types: `single` | `double`

**Q — Quality factor**

scalar

Specify the quality factor as a positive scalar.

**Dependencies**

To enable this port, select Quality factor and center frequency for the **Filter Specification** and **Specify from input port** for the **Quality Factor** parameter.

Data Types: `single` | `double`

## Output

### Port\_1 — Output signal

matrix

The Parametric EQ block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input -- The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input -- The block outputs an  $N$ -by-1 matrix (column vector), where  $N$  is the number of elements in the 1-D vector.

Data Types: `single` | `double`

## Parameters

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

### Filter order — Order of filter

2 (default) | positive even scalar

**Tunable:** No

### Filter specification — Specify parameters used to design filter

Bandwidth and center frequency (default) | Quality factor and center frequency

- Bandwidth and center frequency -- Design the filter using **Filter Bandwidth (Hz)**, **Center Frequency (Hz)**, and **Peak Gain (dB)**.
- Quality factor and center frequency -- Design the filter using **Center Frequency (Hz)**, **Peak Gain (dB)**, and **Quality Factor**.

**Tunable:** No

### Center Frequency (Hz) — Center frequency of filter

11025 (default) | positive scalar

Specify the center frequency as a positive scalar that is less than half the sample rate of the input signal.

To specify **Center Frequency (Hz)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

**Filter Bandwidth (Hz) — Bandwidth of filter**

2205 (default) | positive scalar in the range [0, max(fs/2, 20,000)]

Specify the filter bandwidth as a positive scalar that is less than or equal to half the sample rate of the input signal or 20 kHz, whichever is larger.

To specify **Filter Bandwidth (Hz)** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set **Filter specification** to Bandwidth and center frequency.

**Quality Factor — Quality factor**

5 (default) | scalar in the range [0.1, 20]

Specify the quality factor as a scalar in the range [0.1, 20].

To specify **Quality Factor** from an input port, select **Specify from input port** for the parameter.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set **Filter specification** to Quality factor and center frequency.

**Peak Gain (dB) — Peak or dip gain of filter**

6.0206 (default) | scalar in the range [-30, 30]

Specify the peak gain in dB as a scalar in the range [-30, 30].

**Tunable:** Yes

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

on (default) | off

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

**Tunable:** No**Input sample rate (Hz) — Sample rate of input**

44100 (default) | scalar

**Tunable:** Yes**Dependencies**

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

**Simulate using — Specify type of simulation to run**

Code generation (default) | Interpreted execution

- **Interpreted execution** -- 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 faster compared to **Interpreted execution**.

**Tunable:** No**View filter response — Open plot to visualize magnitude response**

button

## Block Characteristics

<b>Data Types</b>	double   single
<b>Direct Feedthrough</b>	no

<b>Multidimensional Signals</b>	no
<b>Variable-Size Signals</b>	yes
<b>Zero-Crossing Detection</b>	no

## References

[1] Orfanidis, Sophocles J. "High-Order Digital Parametric Equalizer Design." *Journal of the Audio Engineering Society*. Vol. 53, November 2005, pp. 1026-1046.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`designParamEQ` | `designShelvingEQ` | `designVarSlopeFilter` | `multibandParametricEQ`

### Topics

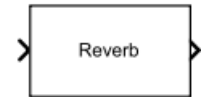
"Parametric Equalizer Design"  
"Equalization"

**Introduced in R2019a**

# Reverberator

Add reverberation to audio signal

**Library:** Audio Toolbox / Effects



## 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.

## Ports

### Input

#### Port\_1 — Input signal

matrix | 1-D vector

- Matrix input -- Each column of the input is treated as an independent channel.
- 1-D vector input -- The input is treated as a single channel.

Data Types: `single` | `double`

### Output

#### Port\_1 — Output signal

matrix

The Reverberator block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input -- The block outputs a matrix the same size and data type as the input signal.

- 1-D vector input -- The block outputs an  $N$ -by-1 matrix (column vector), where  $N$  is the number of elements in the 1-D vector.

Data Types: `single` | `double`

## Parameters

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

### **Pre-delay (s) — Pre-delay for reverberation**

0 (default) | 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 **Pre-delay (s)** is proportional to the size of the room being modeled.

**Tunable:** Yes

### **Highcut frequency (Hz) — Lowpass filter cutoff in the range 0 to (Sample Rate)/2**

20000 (default) | real positive scalar

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.

**Tunable:** Yes

### **Diffusion — Density of reverb tail**

0.50 (default) | 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.

**Tunable:** Yes

### **Decay factor — Decay factor of reverb tail**

0.50 (default) | scalar in the range 0 to 1

**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).

**Tunable:** Yes

**High frequency damping — High-frequency damping**

0.0005 (default) | scalar in the range 0 to 1

**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.

**Tunable:** Yes

**Wet/dry mix — Ratio of wet (reverberated) signal to dry (original) signal**

0.3 (default) | scalar in the range 0 to 1

Wet/dry mix is the ratio of wet (reverberated) signal to dry (original) signal that your Reverberator block outputs.

**Tunable:** Yes

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

on (default) | off

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

**Tunable:** No

**Input sample rate (Hz) — Sample rate of input**

44100 (default) | positive scalar

**Tunable:** Yes

**Dependencies**

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

**Simulate using — Specify type of simulation to run**

Interpreted execution (default) | Code generation



- **Interpreted execution** -- 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.

**Tunable:** No

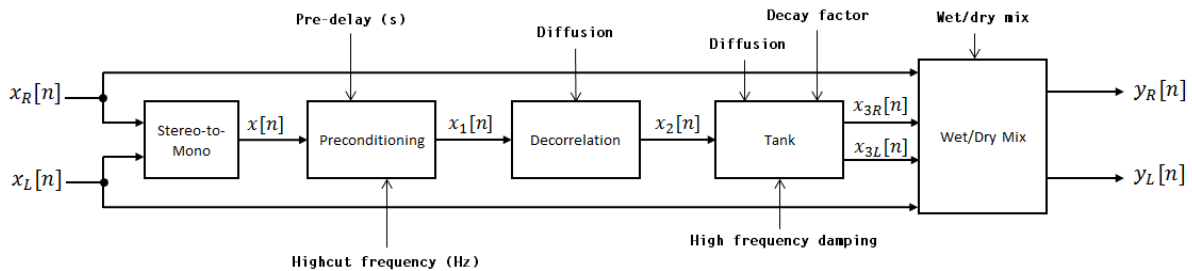
## Block Characteristics

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

## Algorithms

The algorithm to add reverberation follows the plate-class reverberation topology described in [1] and is based on a 29,761 Hz sample rate.

The algorithm has five stages.



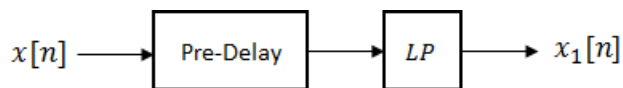
The description for the algorithm that follows is for a stereo input. A mono input is a simplified case.

## Stereo-to-Mono

A stereo signal is converted to a mono signal:  $x[n] = 0.5 \times (x_R[n] + x_L[n])$ .

## Preconditioning

A delay followed by a lowpass filter preconditions the mono signal.



- The pre-delay output is determined as  $x_p[n] = x[n - k]$ , where the **Pre-delay (s)** parameter determines the value of  $k$ .
- The signal is fed through a single-pole lowpass filter with transfer function

$$LP(z) = \frac{1 - \alpha}{1 - \alpha z^{-1}},$$

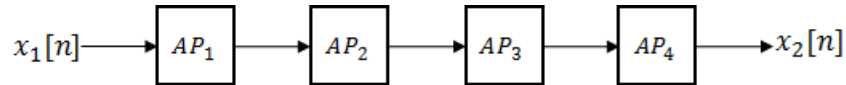
where

$$\alpha = \exp\left(-2\pi \times \frac{f_c}{f_s}\right).$$

- $f_c$  is the cutoff frequency specified by the **Pre-delay (s)** parameter.
- $f_s$  is the sampling frequency specified by the **Inherit sample rate from input** parameter or the **Input sample rate (Hz)** parameter.

## Decorrelation

The signal is decorrelated by passing through a series of four allpass filters.



The allpass filters are of the form

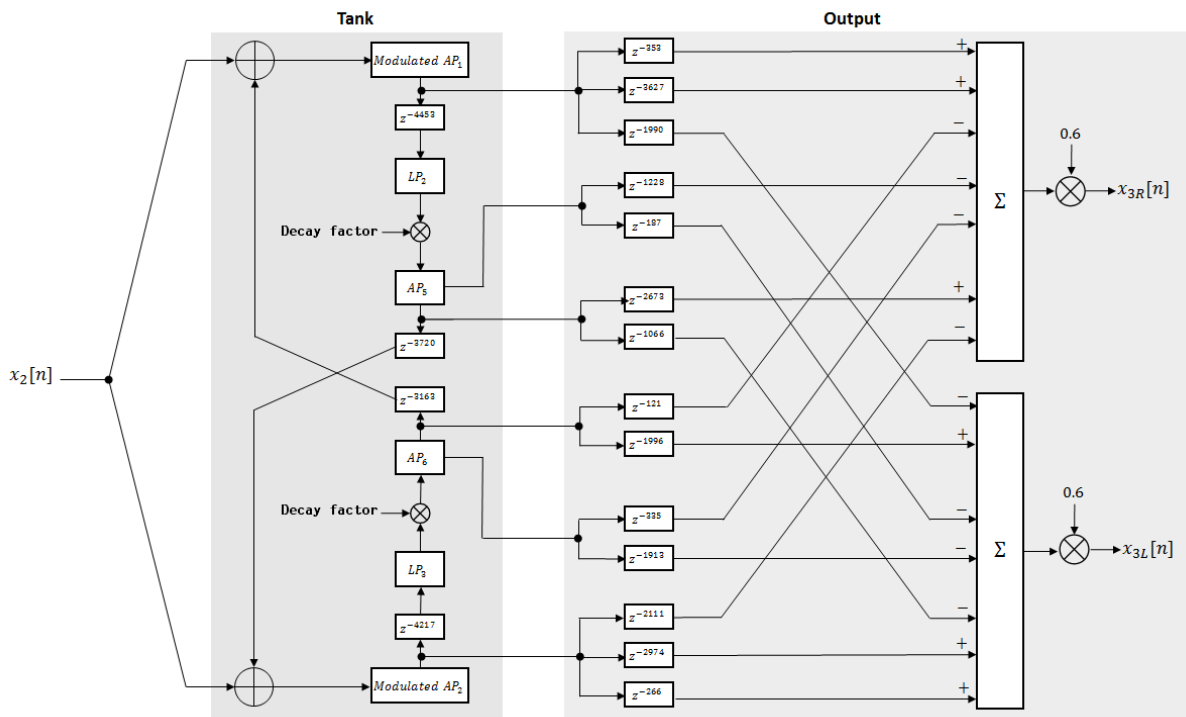
$$AP(z) = \frac{\beta + z^{-k}}{1 + \beta z^{-k}},$$

where  $\beta$  is the coefficient specified by the Diffusion property and  $k$  is the delay as follows:

- For  $AP_1$ ,  $k = 142$ .
- For  $AP_2$ ,  $k = 107$ .
- For  $AP_3$ ,  $k = 379$ .
- For  $AP_4$ ,  $k = 277$ .

## Tank

The signal is fed into the tank, where it circulates to simulate the decay of a reverberation tail.



The following description tracks the signal as it progresses through the top of the tank. The signal progression through the bottom of the tank follows the same pattern, with different delay specifications.

- 1 The new signal enters the top of the tank and is added to the circulated signal from the bottom of the tank.
- 2 The signal passes through a modulated allpass filter:

$$\text{Modulated } AP_1(z) = \frac{-\beta + z^{-k}}{1 - \beta z^{-k}}$$

- $\beta$  is the coefficient specified by the **Diffusion** parameter.
- $k$  is the variable delay specified by a 1 Hz sinusoid with amplitude =  $(8/29761) \times$  (sample rate). To account for fractional delay resulting from the modulating  $k$ , allpass interpolation is used [2].

- 3 The signal is delayed again, and then passes through a lowpass filter:

$$LP_2(z) = \frac{1 - \varphi}{1 - \varphi z^{-1}}$$

- $\varphi$  is the coefficient specified by the **High frequency damping** parameter.

- 4 The signal is multiplied by a gain specified by the **Decay factor** parameter. The signal then passes through an allpass filter:

$$AP_5(z) = \frac{\beta + z^{-k}}{1 + \beta z^{-k}} .$$

- $\beta$  is the coefficient specified by the **Diffusion** parameter.
- $k$  is set to 1800 for the top of the tank and 2656 for the bottom of the tank.

- 5 The signal is delayed again and then circulated to the bottom half of the tank for the next iteration.

A similar pattern is executed in parallel for the bottom half of the tank. The output of the tank is calculated as the signed sum of delay lines picked off at various points from the tank. The summed output is multiplied by  $\theta$ . 6.

## Wet/Dry Mix

The wet (processed) signal is then added to the dry (original) signal:

$$y_R[n] = (1 - \kappa)x_R[n] + \kappa x_{3R}[n],$$

$$y_L[n] = (1 - \kappa)x_L[n] + \kappa x_{3L}[n],$$

where the **Wet/dry mix** parameter determines  $\kappa$ .

## References

- [1] Dattorro, Jon. "Effect Design, Part 1: Reverberator and Other Filters." *Journal of the Audio Engineering Society*. Vol. 45, Issue 9, 1997, pp. 660-684.
- [2] Dattorro, Jon. "Effect Design, Part 2: Delay-Line Modulation and Chorus." *Journal of the Audio Engineering Society*. Vol. 45, Issue 10, 1997, pp. 764-788.

## **Extended Capabilities**

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

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

### **See Also**

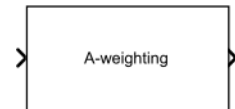
reverberator

**Introduced in R2016a**

# Weighting Filter

Weighted frequency response filter

**Library:** Audio Toolbox / Filters



## Description

The Weighting Filter block performs frequency-weighted filtering independently across each input channel.

## Ports

### Input

#### Port\_1 — Input signal

matrix | 1-D vector

- Matrix input -- Each column of the input is treated as an independent channel.
- 1-D vector input -- The input is treated as a single channel.

Data Types: `single` | `double`

### Output

#### Port\_1 — Output signal

matrix

The Weighting Filter block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input -- The block outputs a matrix the same size and data type as the input signal.

- 1-D vector input -- The block outputs an  $N$ -by-1 matrix (column vector), where  $N$  is the number of elements in the 1-D vector.

Data Types: `single` | `double`

## Parameters

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

### **Weighting method — Type of frequency weighting**

`A-weighting` (default) | `C-weighting` | `K-weighting`

See “A-Weighting” on page 5-122, “C-Weighting” on page 5-123, and “K-Weighting” on page 5-123 for the definition of the weighting curves.

**Tunable:** No

### **Inherit sample rate from input — Specify source of input sample rate**

`off` (default) | `on`

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

**Tunable:** No

### **Input sample rate (Hz) — Sample rate of input**

`44100` (default) | `positive scalar`

**Tunable:** Yes

### **Dependencies**

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

### **Simulate using — Specify type of simulation to run**

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

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



- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed compared to Code generation. In this mode, you can debug the source code of the block.

**Tunable:** No

### **Mask for attenuation limits – Creates a mask for filter response visualization**

No mask (default) | Class 1 | Class 2

The mask attenuation limits are defined in the IEC 61672-1:2002 standard.

- If the mask is green, the design is compliant.
- If the mask is red, the design breaks compliance.

**Tunable:** Yes

#### **Dependencies**

To enable this parameter, set **Weighting method** to A-weighting or C-weighting.

### **Visualize filter response – Open plot to visualize magnitude response and compliance mask**

button

A 2048-point FFT is used to calculate the magnitude response.

**Tunable:** Yes

## **Block Characteristics**

<b>Data Types</b>	double   single
<b>Direct Feedthrough</b>	no
<b>Multidimensional Signals</b>	no
<b>Variable-Size Signals</b>	yes

<b>Zero-Crossing Detection</b>	no
--------------------------------	----

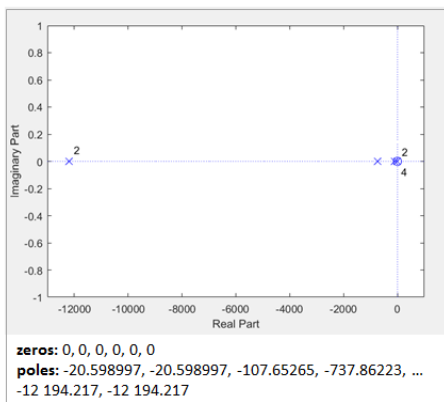
## More About

### A-Weighting

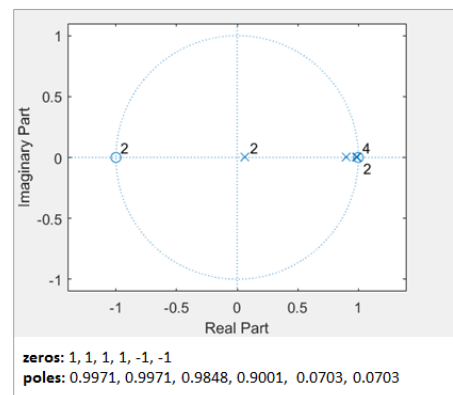
The A-curve is a wide bandpass filter centered at 2.5 kHz, with approximately 20 dB attenuation at 100 Hz and 10 dB attenuation at 20 kHz. A-weighted SPL measurements of noise level are increasingly found in sales literature for domestic appliances. In most countries, the use of A-weighting is mandated for the protection of workers against noise-induced deafness. The ISO and ICOA standards mandate A-weighting for all civil aircraft noise measurements.

The ANSI S1.42.2001 [1] defines this weighting curve. The IEC 61672-1:2002 [2] standard defines the minimum and maximum attenuation limits for an A-weighting filter.

ANSI S1.42.2001 defines the weighting curve by specifying analog poles and zeros. Audio Toolbox converts the specified poles and zeros to the digital domain using a bilinear transform:



→ Bilinear Transform →

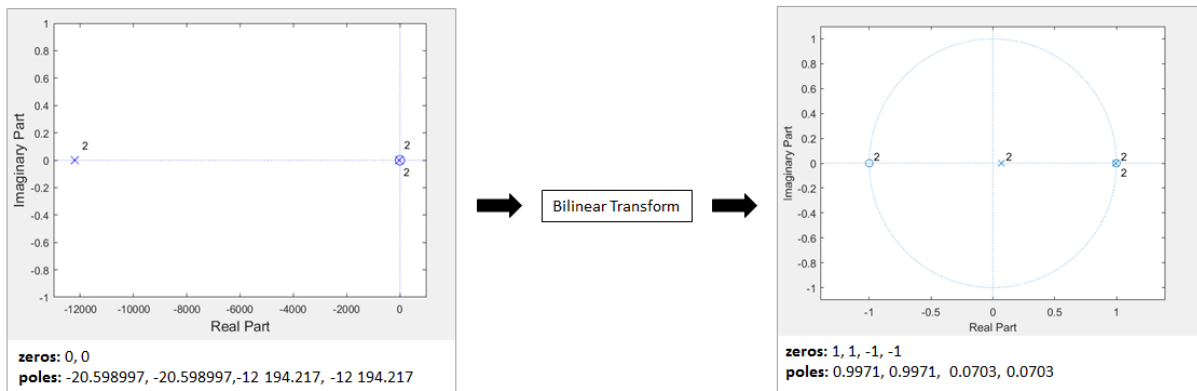


## C-Weighting

The C-curve is "flat," but with limited bandwidth: It has -3 dB corners at 31.5 Hz and 8 kHz. C-curves are used in sound level meters for sounds that are louder than sounds intended for A-weighting filters.

The ANSI S1.42-2001 [1] defines the C-weighting curve. The IEC 61672-1:2002 [2] standard defines the minimum and maximum attenuation limits for C-weighting filters.

ANSI S1.42.2001 defines the weighting curve by specifying analog poles and zeros. Audio Toolbox converts the specified poles and zeros to the digital domain using a bilinear transform:

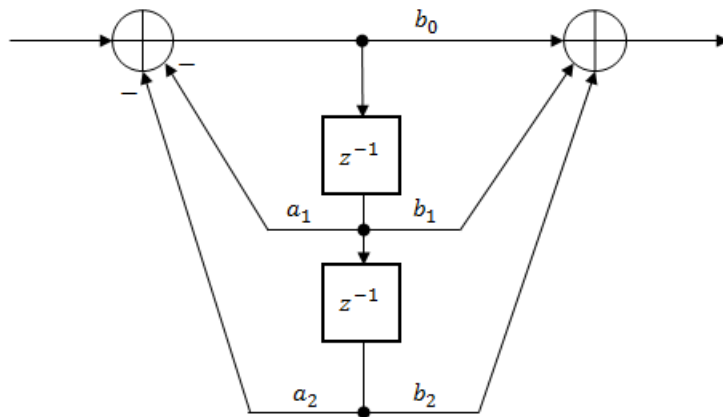


## K-Weighting

The K-weighting filter is used for loudness normalization in broadcast. It is composed of two stages of filtering: a first stage shelving filter and a second stage highpass filter.

The ITU-R BS.1770-4 [3] standard defines this curve.

Assume a second-order filter.



The table shows the coefficients for the filters.

First Stage Shelving Coefficients	Second Stage Highpass Coefficients
$a_1 = -1.69065929318241$	$a_1 = -1.99004745483398$
$a_2 = 0.73248077421585$	$a_2 = 0.99007225036621$
$b_0 = 1.53512485958697$	$b_0 = 1.0$
$b_1 = -2.6916918940638$	$b_1 = -2.0$
$b_2 = 1.19839281085285$	$b_2 = 1.0$

The coefficients presented by ITU-R BS.1770-4 are defined for 48 kHz. These coefficients are recomputed for nonstandard sample rates using the algorithm described in [4].

## References

- [1] Acoustical Society of America. *Design Response of Weighting Networks for Acoustical Measurements*. ANSI S1.42-2001. New York, NY: American National Standards Institute, 2001.
- [2] International Electrotechnical Commission. *Electroacoustics Sound Level Meters Part 1: Specifications*. First Edition. IEC 61672-1. 2002-2005.

- [3] International Telecommunication Union. *Algorithms to measure audio programme loudness and true-peak audio level*. ITU-R BS.1770-4. 2015.
- [4] Mansbridge, Stuart, Saoirse Finn, and Joshua D. Reiss. "Implementation and Evaluation of Autonomous Multi-track Fader Control." Paper presented at the 132nd Audio Engineering Society Convention, Budapest, Hungary, 2012.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

Loudness Meter | Octave Filter | `loudnessMeter` | `octaveFilter` | `weightingFilter`

**Introduced in R2016b**

