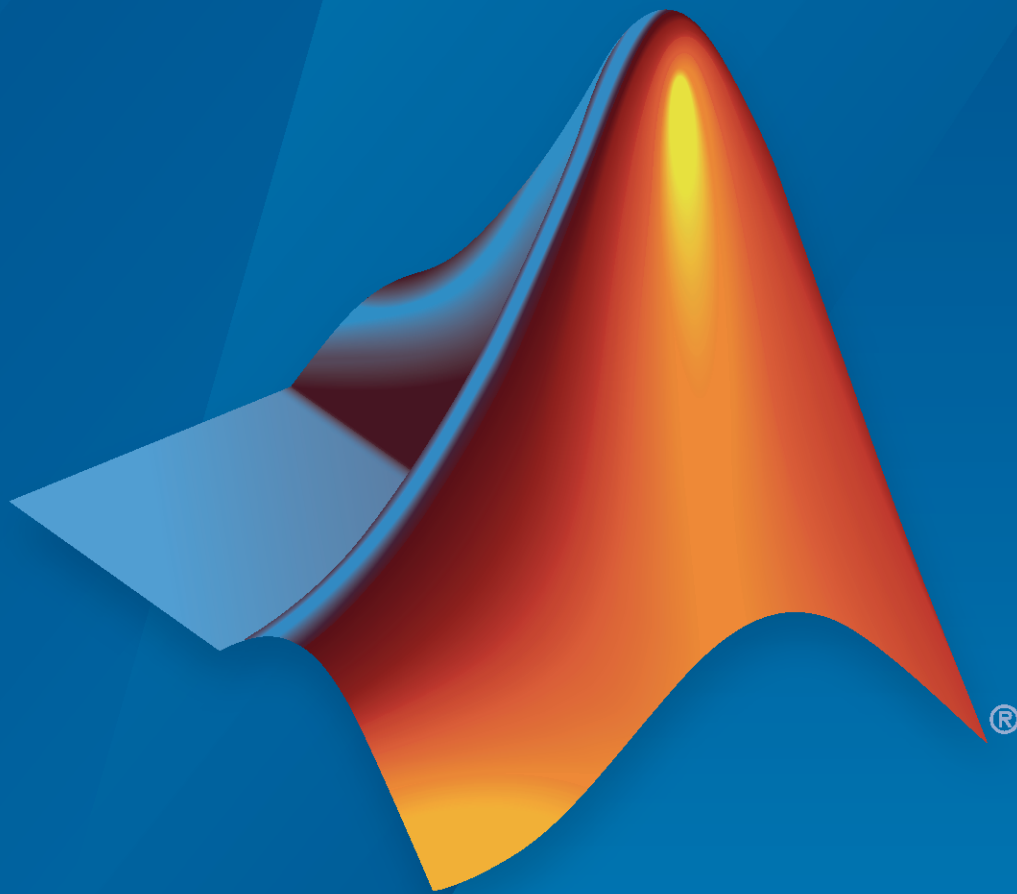


Audio Toolbox™

User's Guide



MATLAB® & SIMULINK®

R2020b



How to Contact MathWorks



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



Phone: 508-647-7000



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

Audio Toolbox™ User's Guide

© COPYRIGHT 2016 - 2020 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 2016	Online only	New for Version 1.0 (Release 2016a)
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)
March 2020	Online only	Revised for Version 2.2 (Release 2020a)
September 2020	Online only	Revised for Version 2.3 (Release 2020b)

Transfer Learning with Pretrained Audio Networks	1-2
Effect of Hearing Protection on Perceived Noise Levels	1-6
Speech Command Recognition Code Generation on Raspberry Pi	1-22
Speech Command Recognition Code Generation with Intel MKL-DNN	1-32
Time-Frequency Masking for Harmonic-Percussive Source Separation	1-39
Binaural Audio Rendering Using Head Tracking	1-62
Speech Emotion Recognition	1-67
Delay-Based Pitch Shifter	1-79
Psychoacoustic Bass Enhancement for Band-Limited Signals	1-83
Tunable Filtering and Visualization Using Audio Plugins	1-86
Communicate Between a DAW and MATLAB Using UDP	1-94
Acoustic Echo Cancellation (AEC)	1-98
Active Noise Control Using a Filtered-X LMS FIR Adaptive Filter	1-110
Acoustic Noise Cancellation Using LMS	1-117
Delay-Based Audio Effects	1-119
Add Reverberation Using Freeverb Algorithm	1-125
Multiband Dynamic Range Compression	1-128
Pitch Shifting and Time Dilation Using a Phase Vocoder in MATLAB	1-137
Pitch Shifting and Time Dilation Using a Phase Vocoder in Simulink	1-141
Remove Interfering Tone From Audio Stream	1-143
Vorbis Decoder	1-145

Dynamic Range Compression Using Overlap-Add Reconstruction	1-148
LPC Analysis and Synthesis of Speech	1-151
Simulation of a Plucked String	1-153
Audio Phaser Using Multiband Parametric Equalizer	1-154
Loudness Normalization in Accordance with EBU R 128 Standard . . .	1-158
Multistage Sample-Rate Conversion of Audio Signals	1-161
Graphic Equalization	1-168
Audio Weighting Filters	1-178
Sound Pressure Measurement of Octave Frequency Bands	1-187
Cochlear Implant Speech Processor	1-190
Acoustic Beamforming Using a Microphone Array	1-195
Identification and Separation of Panned Audio Sources in a Stereo Mix	1-204
Live Direction Of Arrival Estimation with a Linear Microphone Array	1-208
Positional Audio	1-212
Surround Sound Matrix Encoding and Decoding	1-215
Speaker Identification Using Pitch and MFCC	1-224
Measure Audio Latency	1-238
Measure Performance of Streaming Real-Time Audio Algorithms	1-243
THD+N Measurement with Tone-Tracking	1-246
Measure Impulse Response of an Audio System	1-249
Measure Frequency Response of an Audio Device	1-253
Generate Standalone Executable for Parametric Audio Equalizer	1-258
Deploy Audio Applications with MATLAB Compiler	1-261
Parametric Audio Equalizer for Android Devices	1-265
Parametric Audio Equalizer for iOS Devices	1-271
Audio Effects for iOS Devices	1-278
Multiband Dynamic Range Compression for iOS Devices	1-286

Denoise Speech Using Deep Learning Networks	1-297
Classify Gender Using LSTM Networks	1-317
Speech Command Recognition Using Deep Learning	1-331
Ambisonic Plugin Generation	1-350
Ambisonic Binaural Decoding	1-356
Music Genre Classification Using Wavelet Time Scattering	1-359
Multicore Simulation of Acoustic Beamforming Using a Microphone Array	1-368
Convert MIDI Files into MIDI Messages	1-374
Cocktail Party Source Separation Using Deep Learning Networks	1-384
Parametric Equalizer Design	1-406
Octave-Band and Fractional Octave-Band Filters	1-420
Pitch Tracking Using Multiple Pitch Estimations and HMM	1-427
Voice Activity Detection in Noise Using Deep Learning	1-449
Using a MIDI Control Surface to Interact with a Simulink Model	1-471
Spoken Digit Recognition with Wavelet Scattering and Deep Learning	1-474
Active Noise Control with Simulink Real-Time	1-492
Acoustic Scene Recognition Using Late Fusion	1-502
Keyword Spotting in Noise Using MFCC and LSTM Networks	1-522
Speaker Verification Using Gaussian Mixture Model	1-546
Sequential Feature Selection for Audio Features	1-564
Train Generative Adversarial Network (GAN) for Sound Synthesis ...	1-577
Speaker Verification Using i-Vectors	1-598

Plugin GUI Design

2

Design User Interface for Audio Plugin	2-2
---	------------

3

Label Audio Using Audio Labeler	3-2
Load Unlabeled Data	3-2
Define and Assign Labels	3-3
Export Label Definitions	3-9
Export Labeled Audio Data	3-10
Prepare Audio Datastore for Deep Learning Workflow	3-11

Speech2Text and Text2Speech Chapter

4

Speech-to-Text Transcription	4-2
Text-to-Speech Conversion	4-3

Measure Impulse Response of an Audio System

5

Impulse Response Measurer Walkthrough	5-2
Configure Audio I/O System	5-2
Configure IR Acquisition Method	5-2
Acquire IR Measurements	5-3
Analyze and Manage IR Measurements	5-4
Export IR Measurements	5-7

Design and Play a MIDI Synthesizer

6

Design and Play a MIDI Synthesizer	6-2
Convert MIDI Note Messages to Sound Waves	6-2
Synthesize MIDI Messages	6-3
Synthesize Real-Time Note Messages from MIDI Device	6-3

MIDI Device Interface

7

MIDI Device Interface	7-2
MIDI	7-2
MIDI Devices	7-2
MIDI Messages	7-3

Dynamic Range Control	8-2
Linear to dB Conversion	8-3
Gain Computer	8-3
Gain Smoothing	8-4
Make-Up Gain	8-6
dB to Linear Conversion	8-7
Apply Calculated Gain	8-7
Example: Dynamic Range Limiter	8-7

MIDI Control for Audio Plugins

MIDI Control for Audio Plugins	9-2
MIDI and Plugins	9-2
Use MIDI with MATLAB Plugins	9-2

MIDI Control Surface Interface

MIDI Control Surface Interface	10-2
About MIDI	10-2
MIDI Control Surfaces	10-2
Use MIDI Control Surfaces with MATLAB and Simulink	10-3

Use the Audio Test Bench

Audio Test Bench Walkthrough	11-2
Choose Object Under Test	11-2
Run Audio Test Bench	11-3
Debug Source Code of Audio Plugin	11-3
Open Scopes	11-5
Configure Input to Audio Test Bench	11-5
Configure Output from Audio Test Bench	11-6
Call Custom Visualization of Audio Plugin	11-7
Synchronize Plugin Property with MIDI Control	11-8
Play the Audio and Save the Output File	11-8
Validate and Generate Audio Plugin	11-8
Generate MATLAB Script	11-9

12

Audio Plugin Example Gallery	12-2
Audio Effects	12-2
Filters	12-2
Gain Control	12-2
Spatial Audio	12-2
Communicate Between MATLAB and DAW	12-2
Music Information Retrieval	12-2
Speech Processing	12-2
Audio Plugin Examples	12-2

Equalization

13

Equalization	13-2
Equalization Design Using Audio Toolbox	13-2
EQ Filter Design	13-2
Lowpass and Highpass Filter Design	13-5
Shelving Filter Design	13-6

Deployment

14

Desktop Real-Time Audio Acceleration with MATLAB Coder	14-2
---	-------------

Audio I/O User Guide

15

Run Audio I/O Features Outside MATLAB and Simulink	15-2
---	-------------

Block Example Repository

16

Decrease Underrun	16-2
--------------------------------	-------------

Extract Cepstral Coefficients	17-2
Tune Center Frequency Using Input Port	17-4
Gate Audio Signal Using VAD	17-6
Frequency-Domain Voice Activity Detection	17-8
Visualize Noise Power	17-9
Detect Presence of Speech	17-12
Perform Graphic Equalization	17-14
Split-Band De-Essing	17-16
Diminish Plosives from Speech	17-17
Suppress Loud Sounds	17-18
Attenuate Low-Level Noise	17-20
Suppress Volume of Loud Sounds	17-22
Gate Background Noise	17-24
Output Values from MIDI Control Surface	17-26
Apply Frequency Weighting	17-28
Compare Loudness Before and After Audio Processing	17-30
Two-Band Crossover Filtering for a Stereo Speaker System	17-32
Mimic Acoustic Environments	17-34
Perform Parametric Equalization	17-35
Perform Octave Filtering	17-37
Read from Microphone and Write to Speaker	17-39
Channel Mapping	17-41
Trigger Gain Control Based on Loudness Measurement	17-42
Generate Variable-Frequency Tones in Simulink	17-44
Trigger Reverberation Parameters	17-47

Model Engine Noise	17-48
---------------------------------	--------------

Real-Time Parameter Tuning

18

Real-Time Parameter Tuning	18-2
Programmatic Parameter Tuning	18-2

Tips and Tricks for Plugin Authoring

19

Tips and Tricks for Plugin Authoring	19-2
Avoid Disrupting the Event Queue in MATLAB	19-2
Separate Code for Features Not Supported for Plugin Generation	19-4
Implement Reset Correctly	19-6
Implement Plugin Composition Correctly	19-6
Address "A set method for a non-Dependent property should not access another property" Warning in Plugin	19-8
Use System Object That Does Not Support Variable-Size Signals	19-10
Using Enumeration Parameter Mapping	19-12

Spectral Descriptors Chapter

20

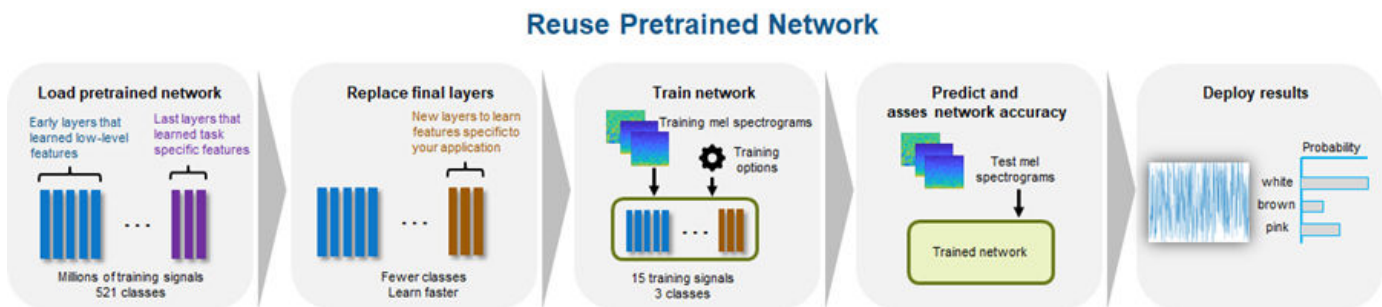
Spectral Descriptors	20-2
-----------------------------------	-------------

Audio Toolbox Examples

Transfer Learning with Pretrained Audio Networks

This example shows how to use transfer learning to retrain YAMNet, a pretrained convolutional neural network, to classify a new set of audio signals. To get started with audio deep learning from scratch, see “Classify Sound Using Deep Learning”.

Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features to a new task using a smaller number of training signals.



Audio Toolbox™ additionally provides the `classifySound` function, which implements necessary preprocessing for YAMNet and convenient postprocessing to interpret the results. Audio Toolbox also provides the pretrained VGGish network (`vggish`) as well as the `vggishFeatures` function, which implements preprocessing and postprocessing for the VGGish network.

Create Data

Generate 100 white noise signals, 100 brown noise signals, and 100 pink noise signals. Each signal represents a duration of 0.98 seconds assuming a 16 kHz sample rate.

```
fs = 16e3;
duration = 0.98;
N = duration*fs;
numSignals = 100;

wNoise = 2*rand([N,numSignals]) - 1;
wLabels = repelem(categorical("white"),numSignals,1);

bNoise = filter(1,[1,-0.999],wNoise);
bNoise = bNoise./max(abs(bNoise),[],'all');
bLabels = repelem(categorical("brown"),numSignals,1);

pNoise = pinknoise([N,numSignals]);
pLabels = repelem(categorical("pink"),numSignals,1);
```

Split the data into training and test sets. Normally, the training set consists of most of the data. However, to illustrate the power of transfer learning, you will use only a few samples for training and the majority for validation.

K = 5 ;


```

trainAudio = [wNoise(:,1:K),bNoise(:,1:K),pNoise(:,1:K)];
trainLabels = [wLabels(1:K);bLabels(1:K);pLabels(1:K)];

validationAudio = [wNoise(:,K+1:end),bNoise(:,K+1:end),pNoise(:,K+1:end)];
validationLabels = [wLabels(K+1:end);bLabels(K+1:end);pLabels(K+1:end)];

fprintf("Number of samples per noise color in train set = %d\n" + ...
        "Number of samples per noise color in validation set = %d\n",K,numSignals-K);

Number of samples per noise color in train set = 5
Number of samples per noise color in validation set = 95

```

Extract Features

Use `melSpectrogram` to extract log-mel spectrograms from both the training set and the validation set using the same parameters as the YAMNet model was trained on.

```

FFTLenght = 512;
numBands = 64;
frequencyRange = [125 7500];
windowLength = 0.025*fs;
overlapLength = 0.015*fs;

trainFeatures = melSpectrogram(trainAudio,fs, ...
    'Window',hann(windowLength,'periodic'), ...
    'OverlapLength',overlapLength, ...
    'FFTLenght',FFTLenght, ...
    'FrequencyRange',frequencyRange, ...
    'NumBands',numBands, ...
    'FilterBankNormalization','none', ...
    'WindowNormalization',false, ...
    'SpectrumType','magnitude', ...
    'FilterBankDesignDomain','warped');

trainFeatures = log(trainFeatures + single(0.001));
trainFeatures = permute(trainFeatures,[2,1,4,3]);

validationFeatures = melSpectrogram(validationAudio,fs, ...
    'Window',hann(windowLength,'periodic'), ...
    'OverlapLength',overlapLength, ...
    'FFTLenght',FFTLenght, ...
    'FrequencyRange',frequencyRange, ...
    'NumBands',numBands, ...
    'FilterBankNormalization','none', ...
    'WindowNormalization',false, ...
    'SpectrumType','magnitude', ...
    'FilterBankDesignDomain','warped');

validationFeatures = log(validationFeatures + single(0.001));
validationFeatures = permute(validationFeatures,[2,1,4,3]);

```

Transfer Learning

To load the pretrained network, call `yamnet`. If the Audio Toolbox model for YAMNet is not installed, then the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path. The YAMNet model can classify audio into one of 521 sound categories, including white noise and pink noise (but not brown noise).

```
net = yamnet;  
net.Layers(end).Classes  
  
ans = 521x1 categorical  
Speech  
Child speech, kid speaking  
Conversation  
Narration, monologue  
Babbling  
Speech synthesizer  
Shout  
Bellow  
Whoop  
Yell  
Children shouting  
Screaming  
Whispering  
Laughter  
Baby laughter  
Giggle  
Snicker  
Belly laugh  
Chuckle, chortle  
Crying, sobbing  
Baby cry, infant cry  
Whimper  
Wail, moan  
Sigh  
Singing  
Choir  
Yodeling  
Chant  
Mantra  
Child singing  
:
```

Prepare the model for transfer learning by first converting the network to a `layerGraph` (Deep Learning Toolbox). Use `replaceLayer` (Deep Learning Toolbox) to replace the fully-connected layer with an untrained fully-connected layer. Replace the classification layer with a classification layer that classifies the input as "white", "pink", or "brown". See "List of Deep Learning Layers" (Deep Learning Toolbox) for deep learning layers supported in MATLAB®.

```
uniqueLabels = unique(trainLabels);  
numLabels = numel(uniqueLabels);  
  
lgraph = layerGraph(net.Layers);  
  
lgraph = replaceLayer(lgraph,"dense",fullyConnectedLayer(numLabels,"Name","dense"));  
lgraph = replaceLayer(lgraph,"Sound",classificationLayer("Name","Sounds","Classes",uniqueLabels));
```

To define training options, use `trainingOptions` (Deep Learning Toolbox).

```
options = trainingOptions('adam','ValidationData',{single(validationFeatures),validationLabels});
```

To train the network, use `trainNetwork` (Deep Learning Toolbox). The network achieves a validation accuracy of 100% using only 5 signals per noise type.

```
trainNetwork(single(trainFeatures),trainLabels,lgraph,options);
```

Training on single CPU.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Validation Accuracy	Mini-batch Loss	Validation Loss
1	1	00:00:02	20.00%	88.77%	1.1922	0.0431
30	30	00:00:14	100.00%	100.00%	9.1076e-06	5.0431e-06

Effect of Hearing Protection on Perceived Noise Levels

In this example, you measure the noise of a helicopter and use psychoacoustic metrics to model its perceived loudness, sharpness, fluctuation strength, and relative annoyance level. You then model the effect of hearing protection to estimate the perceived improvement.

Recording Level Calibration

Measuring sound pressure levels (SPL) or perceptual metrics such as loudness requires you to first calibrate your microphone input level. You can use `calibrateMicrophone` to match your recording level to the SPL reading of a calibrated meter. All you need is a calibrated SPL meter and a 1 kHz tone being played back from your computer or even from an app on your phone. The 1 kHz calibration tone does not have to be at a known calibrated level, but it should be loud enough to dominate any background noise. For an example of how to calibrate by playing the tone with MATLAB, see the documentation for `calibrateMicrophone`.

Simulate the tone recording and include some background noise. Assume an SPL meter gave a reading of 90.1 dB (C-weighted).

```
FS = 48e3;  
t = (1:2*FS)/FS;  
s = rng('default');  
xtone = 0.046*sin(2*pi*t*1000).' + 1e-2*pinknoise(2*FS);  
rng(s)
```

```
splMeterReading = 90.1;
```

To compute the calibration level of a recording chain, call `calibrateMicrophone` and specify the test tone, the sample rate, the SPL reading, and the frequency weighting of the SPL meter. Even though the signal at 1 kHz is not affected by the frequency weighting, the background noise is, so you get a more precise calibration level by matching the meter's frequency weighting.

```
calib = calibrateMicrophone(xtone,FS,splMeterReading,"FrequencyWeighting","C-weighting");
```

Sound Pressure Levels (SPL)

Once you have a calibration factor for a recording chain, you can make acoustic measurements. When using a physical meter, you might be limited to whatever settings were selected at the time of the measurement. Here, using your recording and the `splMeter` object, you can select any settings. This makes it easy to experiment with different time and frequency weighting options.

Load a helicopter recording that contains spatial information. You only need to keep the first channel, which corresponds to the omnidirectional signal (similar to a mono recording with a normal omnidirectional microphone). Create the corresponding time vector.

```
[x,FS] = audioread("Heli_16ch_ACN_SN3D.wav");  
x = x(:,1);  
t = (1:size(x,1))/FS;
```

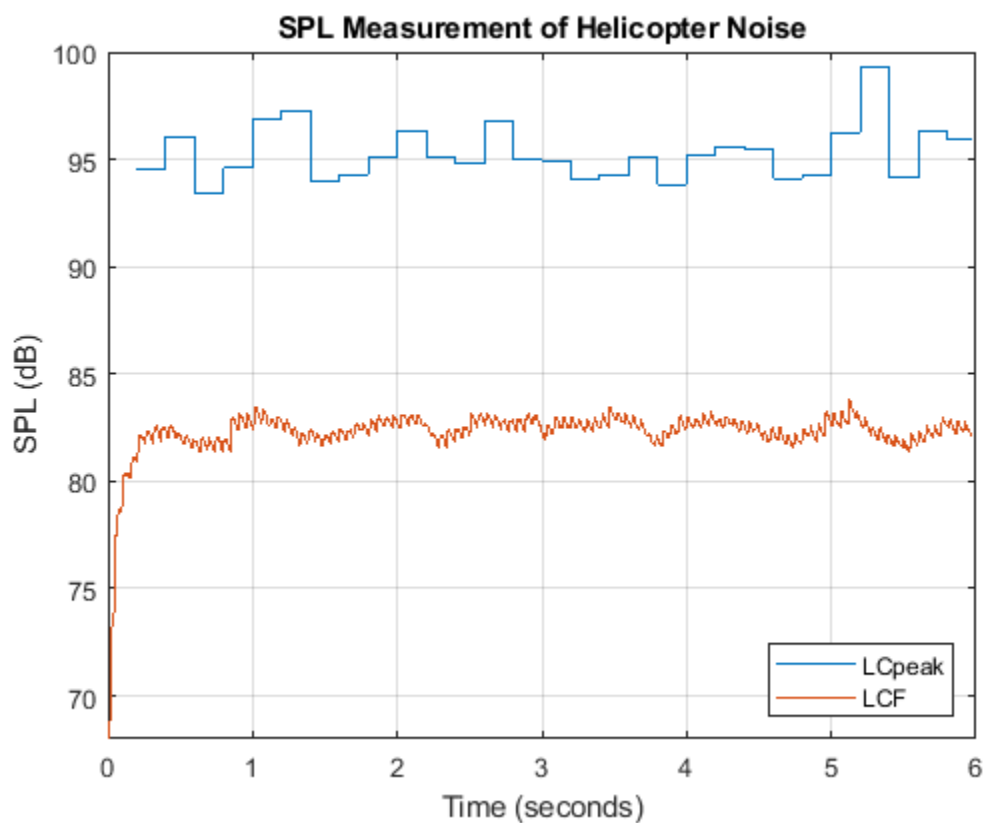
Create an `splMeter` object and select C-weighting, fast time weighting, and a 0.2 second interval for the peak SPL measurement.

```
spl = splMeter("CalibrationFactor",calib, ...  
              "FrequencyWeighting","C-weighting", ...  
              "TimeWeighting","Fast", ...
```

```
"TimeInterval",0.2, ...
"SampleRate",FS);
```

Plot SPL and peak SPL.

```
[LCF,~,LCpeak] = spl(x);
plot(t,LCpeak,t,LCF)
legend('LCpeak','LCF','Location','southeast')
title('SPL Measurement of Helicopter Noise')
xlabel('Time (seconds)')
ylabel('SPL (dB)')
ylim([68 100])
grid on
```



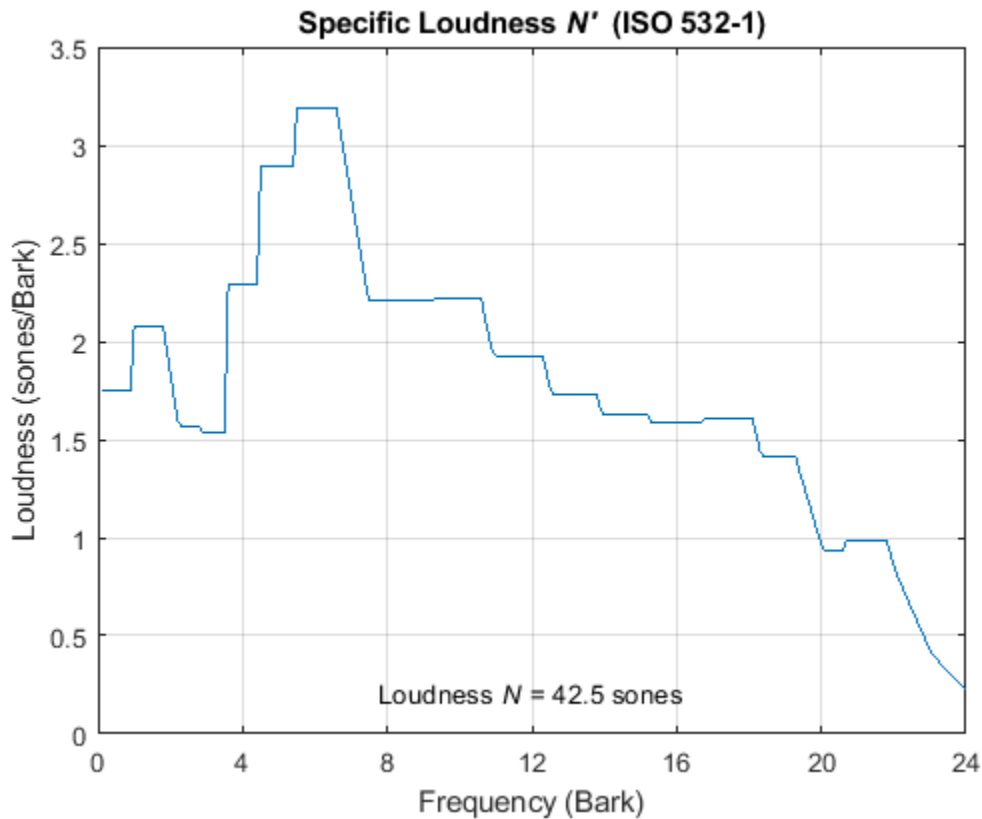
Psychoacoustic Metrics

Loudness level

Monitoring SPL is important for hearing safety, but `acousticLoudness` allows you to estimate loudness levels as they are actually perceived by people with normal hearing (that is, no large hearing impairments). You can also see what frequency bands contribute the most to the sensation of loudness.

Using the same calibration level as before, and assuming a free-field recording (the default), plot stationary loudness.

```
acousticLoudness(x,FS,calib)
```



The loudness is 42.5 sones, with a peak at 6 (Bark scale). Convert the peak loudness value to Hz using `bark2hz`

```
fprintf("Peak loudness frequency: %d Hz\n",round(bark2hz(6),-1));
```

Peak loudness frequency: 630 Hz

By default, the levels are in sones. It might be easier to understand this measurement if it was compared to an SPL (dB) reading. The phons unit of loudness is matched to a reference frequency of 1 kHz. For example, any signal with a loudness of 60 phons is perceived to be as loud as a 1 kHz tone that registers at 60 dB on an SPL meter (1 kHz is unaffected by the frequency weighting). Consequently, converting 42.5 sones to phons using `sone2phon` lets you know that the helicopter is perceived as loud as a 94 dB 1 kHz tone.

```
fprintf("Equivalent 1 kHz SPL: %d phons\n", round(sone2phon(42.5)));
```

Equivalent 1 kHz SPL: 94 phons

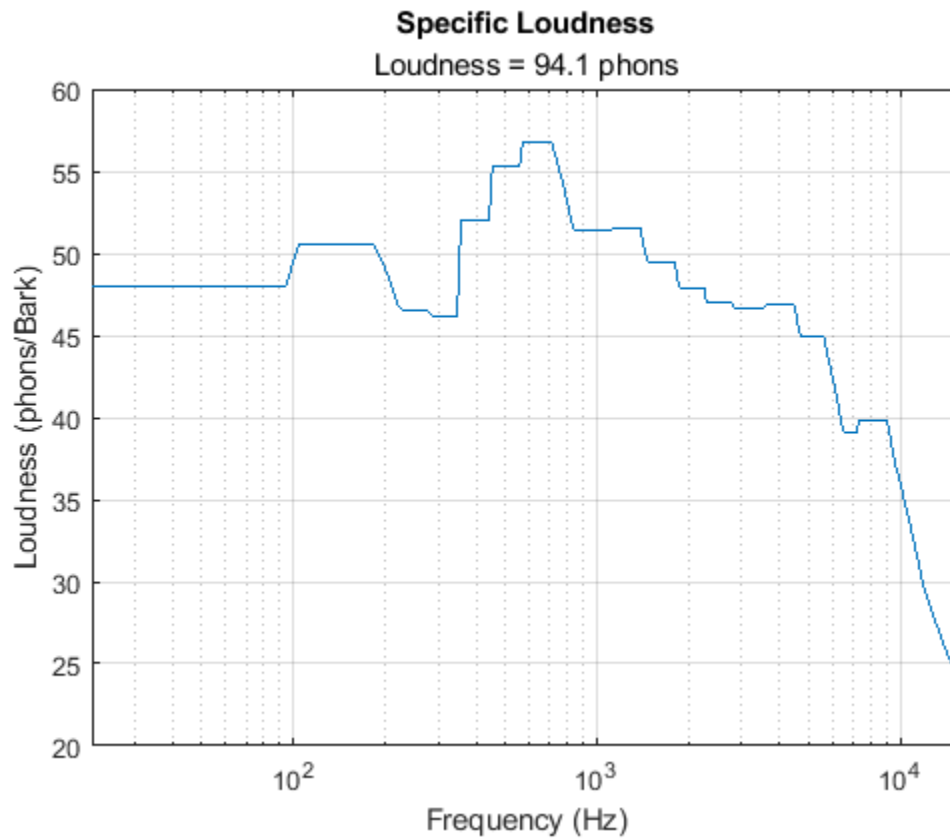
Make your own plot with units in phons and frequency in Hz on a log scale.

```
[sone,spec] = acousticLoudness(x,FS,calib);
barks = 0.1:0.1:24; % Bark scale for ISO 532-1 loudness
hz = bark2hz(barks);
specPhon = sone2phon(spec);
semilogx(hz,specPhon)
title('Specific Loudness')
subtitle(sprintf('Loudness = %.1f phons',sone2phon(sone)))
```

```

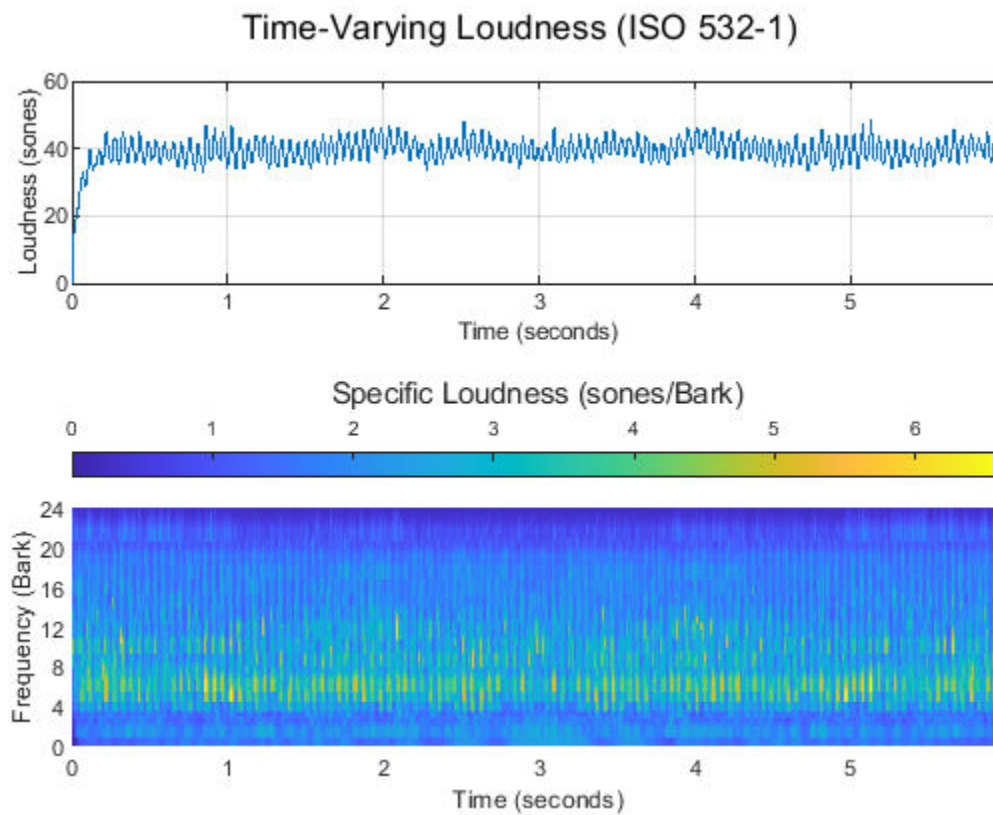
xlabel('Frequency (Hz)')
ylabel('Loudness (phons/Bark)')
xlim(hz([1,end]))
grid on

```



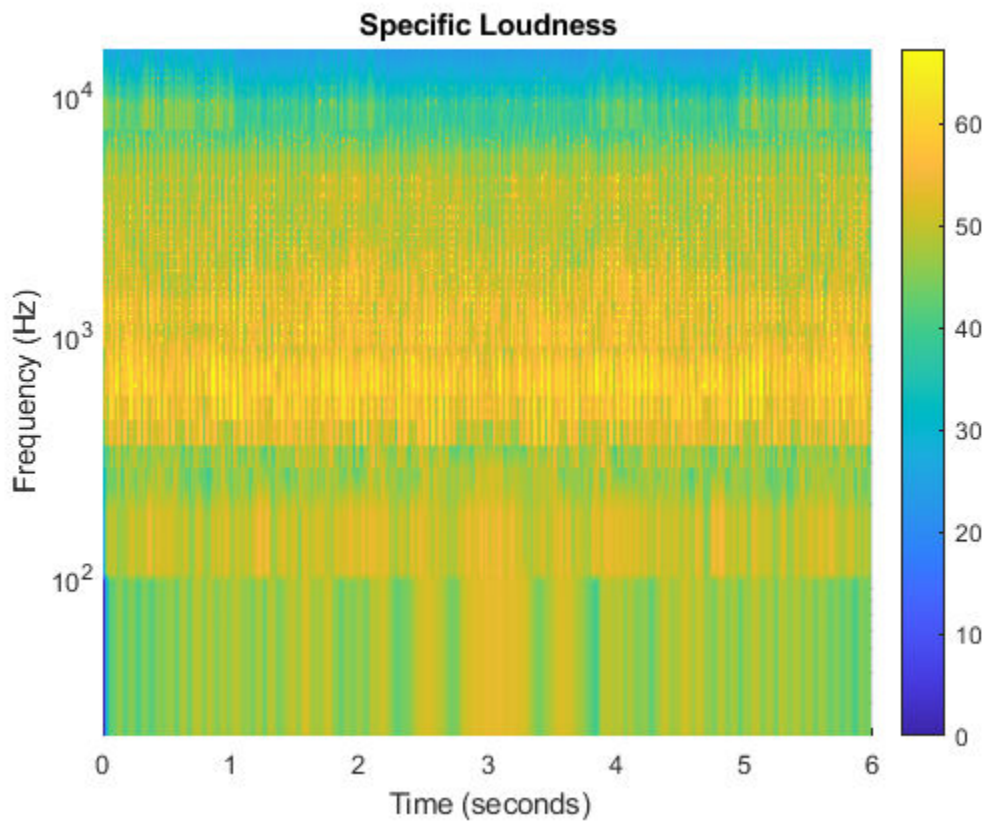
You can also plot time-varying loudness and specific loudness to analyze the sound of the moving helicopter. In this case, the noise is stationary, but you can observe the distinctive "impulsive" nature of a helicopter.

```
acousticLoudness(x,FS,calib,'TimeVarying',true)
```



Plot specific loudness with the frequency in Hz (log-scale).

```
[tvsone,tvspec,perc] = acousticLoudness(x,FS,calib,'TimeVarying',true);
spectime = 0:2e-3:2e-3*(size(tvspec,1)-1); % time axis for loudness output
clf; % do not reuse the previous subplot
surf(spectime,hz,sone2phon(tvspec).','EdgeColor','interp');
set(gca,'View',[0 90],'YScale','log','YLim',hz([1,end]));
title('Specific Loudness')
zlabel('Specific Loudness (phons/Bark)')
ylabel('Frequency (Hz)')
xlabel('Time (seconds)')
colorbar
```

Sharpness Level

The perceived sharpness of a sound might contribute significantly to how annoying it is to people. Estimate the perceived sharpness level using `acousticSharpness`.

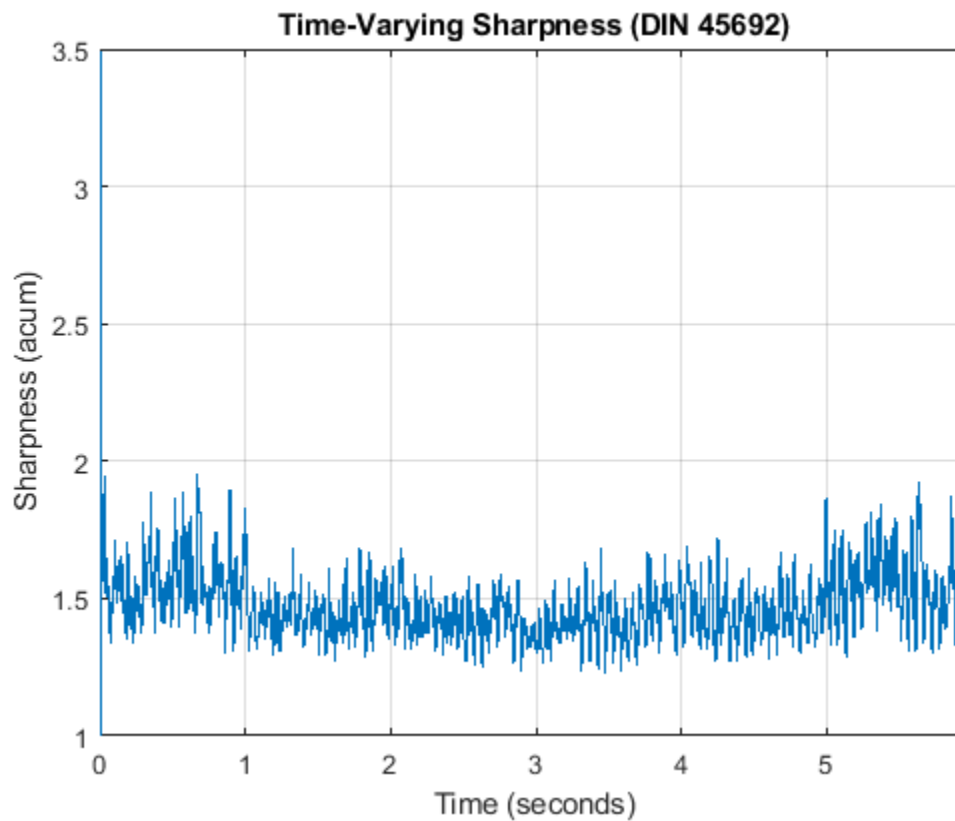
```
sharp = acousticSharpness(spec)
```

```
sharp = 1.4507
```

Pink noise has a sharpness of 2 acums, so you now know that the helicopter noise is biased towards low frequencies.

Plot time-varying sharpness.

```
acousticSharpness(x,FS,calib,'TimeVarying',true);
```

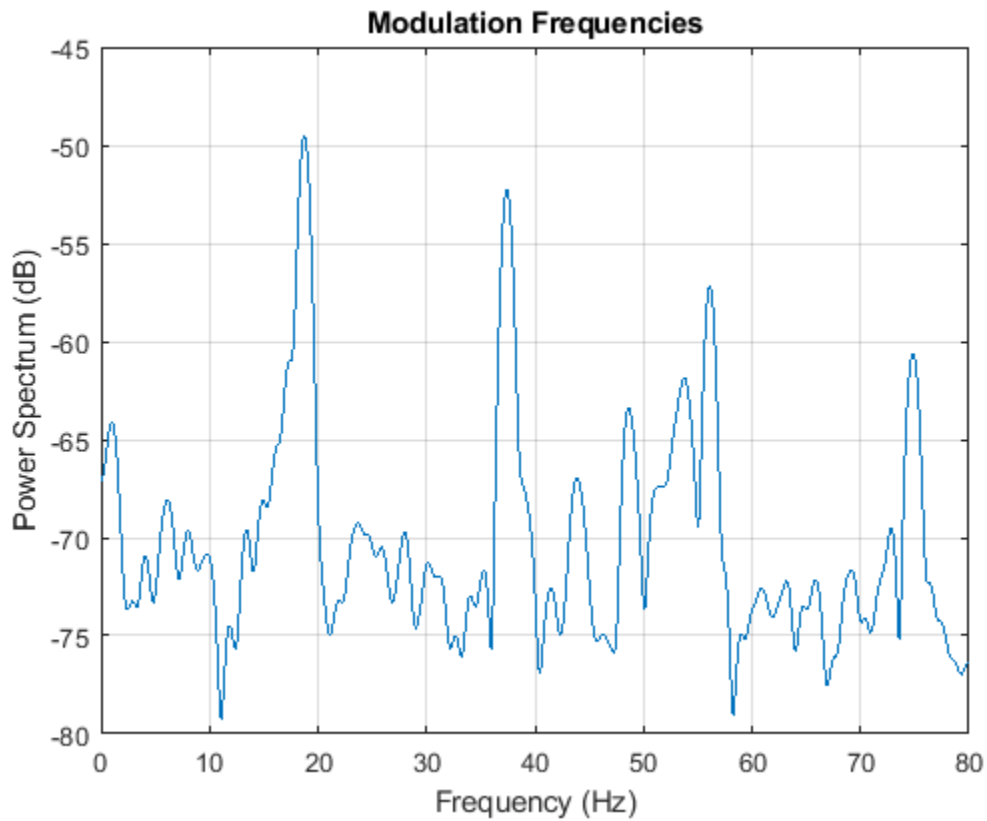


Fluctuation Strength

In the case of helicopter noise, the low-frequency modulation also contributes to how annoying the noise might be perceived.

First, look at what modulation frequencies are found in the signal.

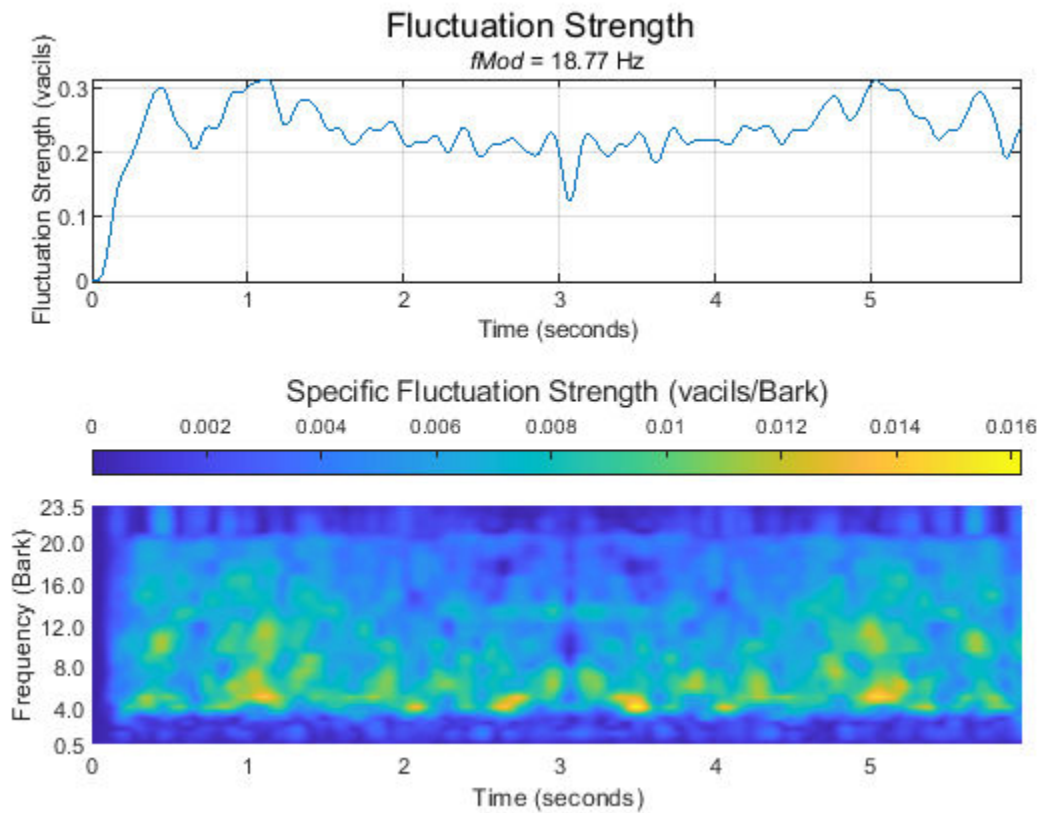
```
N = 2^nextpow2(size(x,1));  
xa = abs(x); % Use the rectified signal  
pspectrum(xa-mean(xa),FS,'FrequencyLimits',[0 80],'FrequencyResolution',1)  
title('Modulation Frequencies')
```



Observe that the modulation frequency peaks at 18.8 Hz. Below 30 Hz, modulation is perceived as fluctuation (as opposed to roughness).

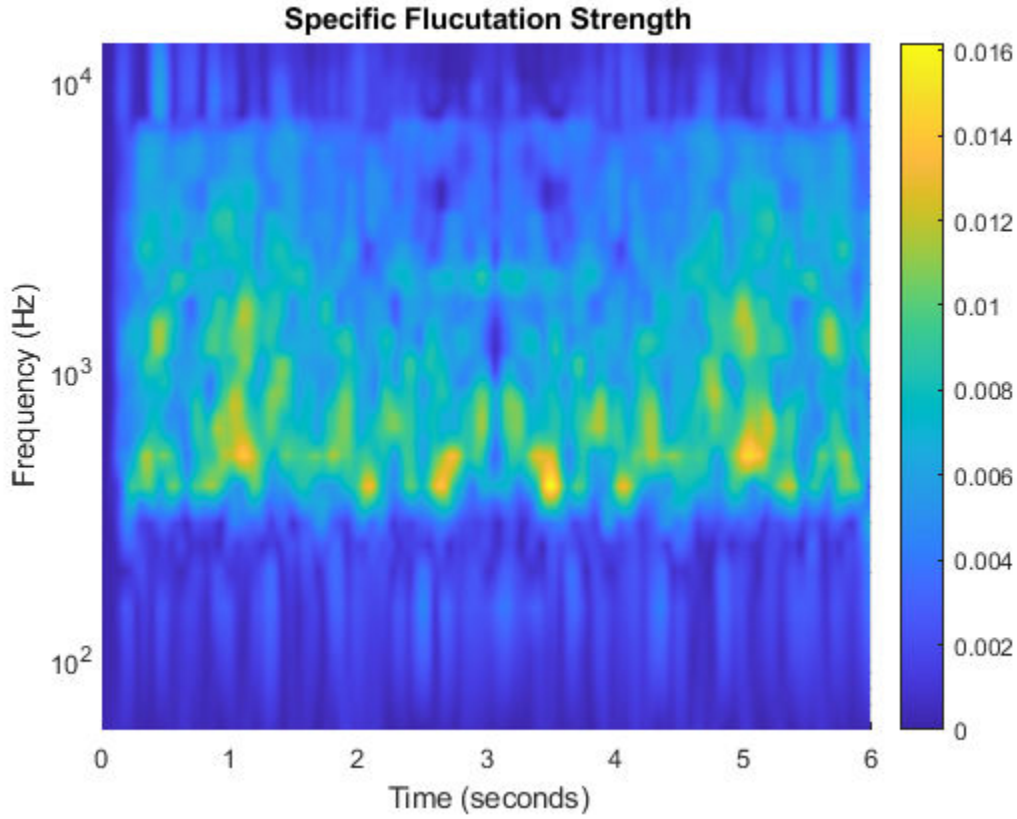
Use `acousticFluctuation` to compute the perceived fluctuation strength. Let the algorithm automatically detect the most audible fluctuation frequency (f_{Mod}).

```
acousticFluctuation(x,FS,calib)
```



You might want to use Hz instead of Bark. To save computations, reuse the time-varying specific loudness that you computed previously.

```
[vacil,spec,fMod] = acousticFluctuation(tvspec);
clf; % do not reuse previous subplot
flucHz = bark2hz(0.5:0.5:23.5);
surf(spectime,flucHz,spec.','EdgeColor','interp');
set(gca,'View',[0 90],'YScale','log','YLim',flucHz([1,end]));
title('Specific Fluctuation Strength')
zlabel('Specific Fluctuation Strength (vacils/Bark)')
ylabel('Frequency (Hz)')
xlabel('Time (seconds)')
colorbar
```



Sound Quality

For the evaluation of sound quality, the previous metrics can be combined to produce a *psychoacoustic annoyance* factor (Zwicker and Fastl). The relation is as follows:

$$PA \sim N \left(1 + \sqrt{[g_1(S)]^2 + [g_2(F, R)]^2} \right)$$

A quantitative description was developed using the results of psychoacoustic experiments:

$$PA = N_5 \left(1 + \sqrt{w_S^2 + w_{FR}^2} \right)$$

with:

- N_5 percentile loudness in sone (level that is exceeded only 5% of the time)
- $w_S = (S - 1.75) \cdot (0.25 \cdot \log_{10}(N_5 + 10))$ for $S > 1.75$, where S is the sharpness in acum
- $w_{FR} = \frac{2.18}{(N_5)^{0.4}} (0.4 \cdot F + 0.6 \cdot R)$, where F is the fluctuation strength in vacil and R is the roughness in asper

In this example, sharpness was less than 1.75, so it is not a contributing factor. The noise is mainly modulated by a frequency lower than 30 Hz, so consider roughness negligible for this example. Therefore, you can set R and w_S to zero.

Percentile loudness, N_5 , is the second value returned by the third output of `acousticLoudness` when "TimeVarying" is set to `true`.

```
N5 = perc(2);
```

Compute an average fluctuation strength ignoring the first second of the signal.

```
f = mean(vacil(501:end,1));
```

Compute the *psychoacoustic annoyance* factor.

```
pa = N5 * (1 + abs(2.18/(N5^.4)*(.4*f)))
```

```
pa = 46.7171
```

Effect of Ear Protection

Measure the impact of hearing protection on the measured SPL and the perceived noise. The attenuation characteristics of a 24-dB-rated hearing protector are:

Frequency (Hz) 125 250 500 1000 2000 3000 4000 6000 8000

Attenuation (dB) 14.4 21.6 27.0 31.9 36.1 39.5 41.9 39.8 37.0

Interpolate the hearing protection data.

```
att = [ 125 250 500 1000 2000 3000 4000 6000 8000 ; ...  
       14.4 21.6 27.0 31.9 36.1 39.5 41.9 39.8 37.0 ].';
```

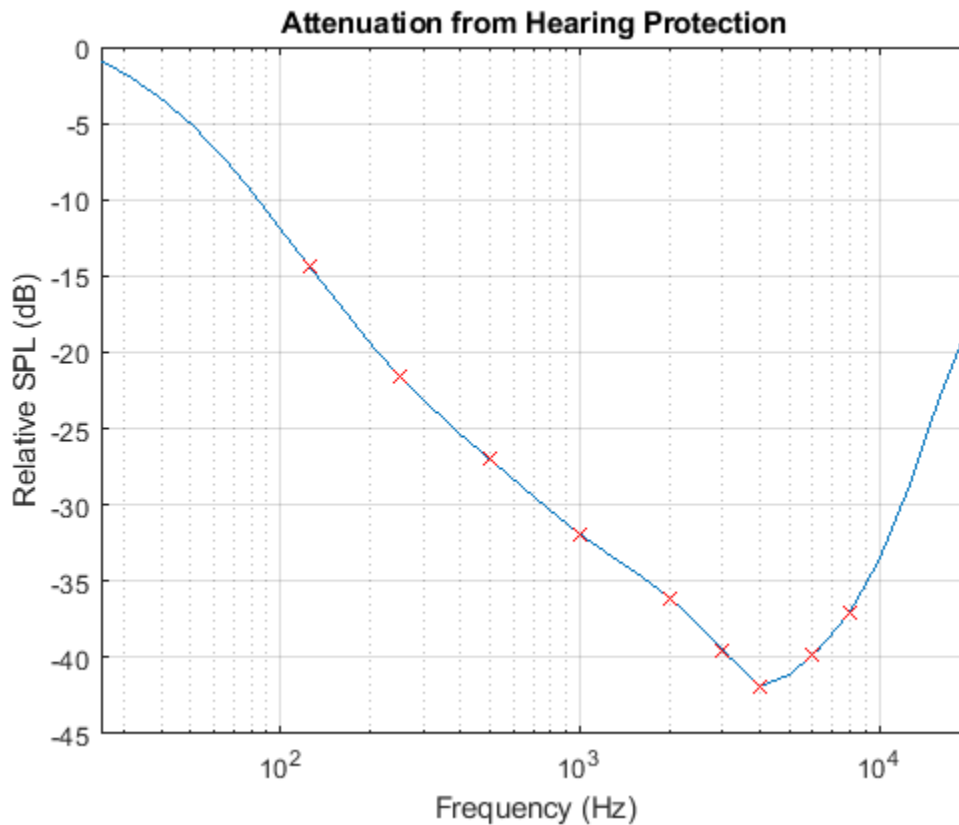
```
att = [ 20,0; att ]; % Assume zero attenuation at 20 Hz
```

```
cf = getCenterFrequencies(splMeter("Bandwidth","1/3 octave"));
```

```
p = interp1(att(:,1),att(:,2),cf,"pchip");
```

Display the interpolated attenuation.

```
semilogx(cf,-p,att(:,1),-att(:,2),'rx')  
title('Attenuation from Hearing Protection')  
ylabel('Relative SPL (dB)')  
xlabel('Frequency (Hz)')  
xlim(cf([1,end]))  
grid on
```



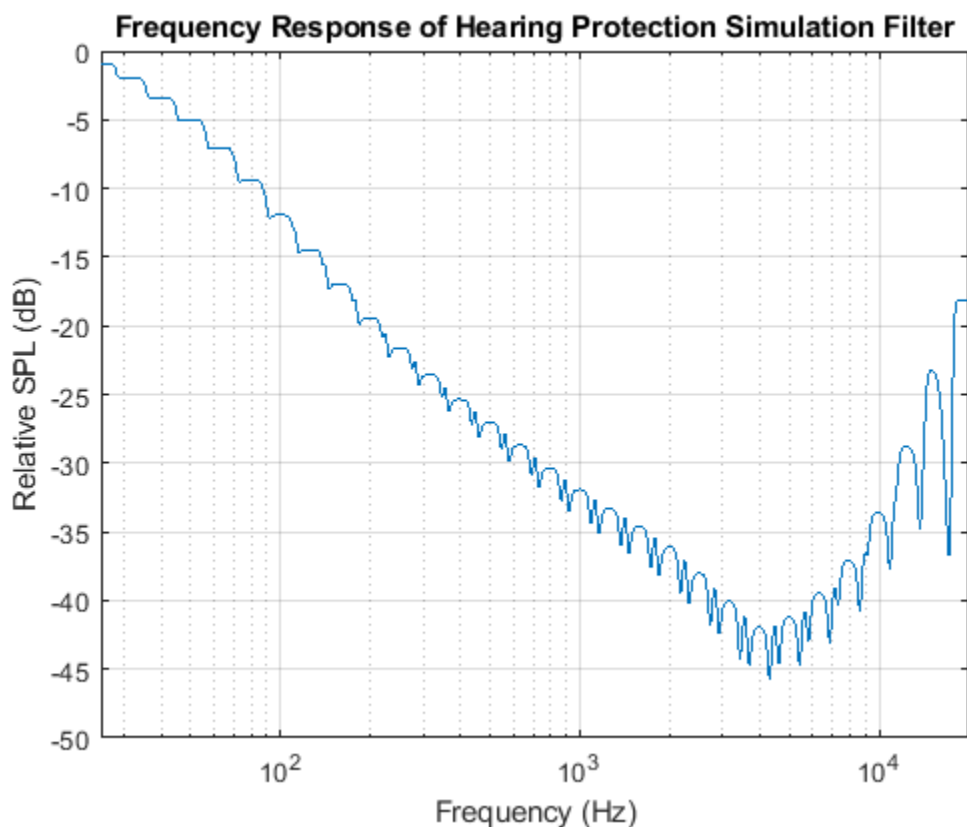
Simulation Using Graphic EQ Filter Bank

Design a graphicEQ object to simulate the effect of the hearing protection.

```
geq = graphicEQ("SampleRate",FS,"Bandwidth","1/3 octave","EQOrder",14,"Gains",-p);
```

Display the frequency response of the graphicEQ object.

```
[B,A] = coeffs(geq);
sos = [B;ones(1,size(A,2));A]].';
[H,w] = freqz(sos,2^16,FS);
semilogx(w,db(abs(H)))
title('Frequency Response of Hearing Protection Simulation Filter')
ylabel('Relative SPL (dB)')
xlabel('Frequency (Hz)')
xlim(cf([1,end]))
grid on
```

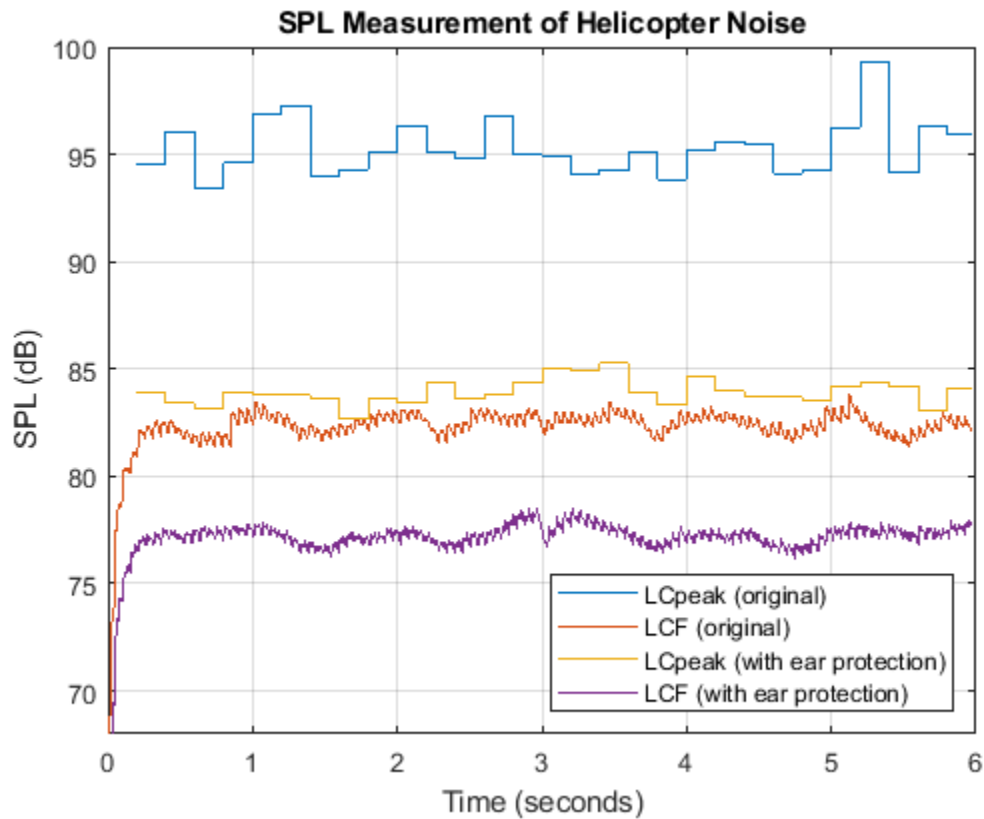


Filter the helicopter recording using the graphic EQ to simulate the hearing protection.

```
xhp = geq(x);
```

Compare the SPL with and without hearing protection. Reuse the same SPL meter settings, but use the filtered recording. Set the Y limits as before to make visual comparison easier with the earlier plot.

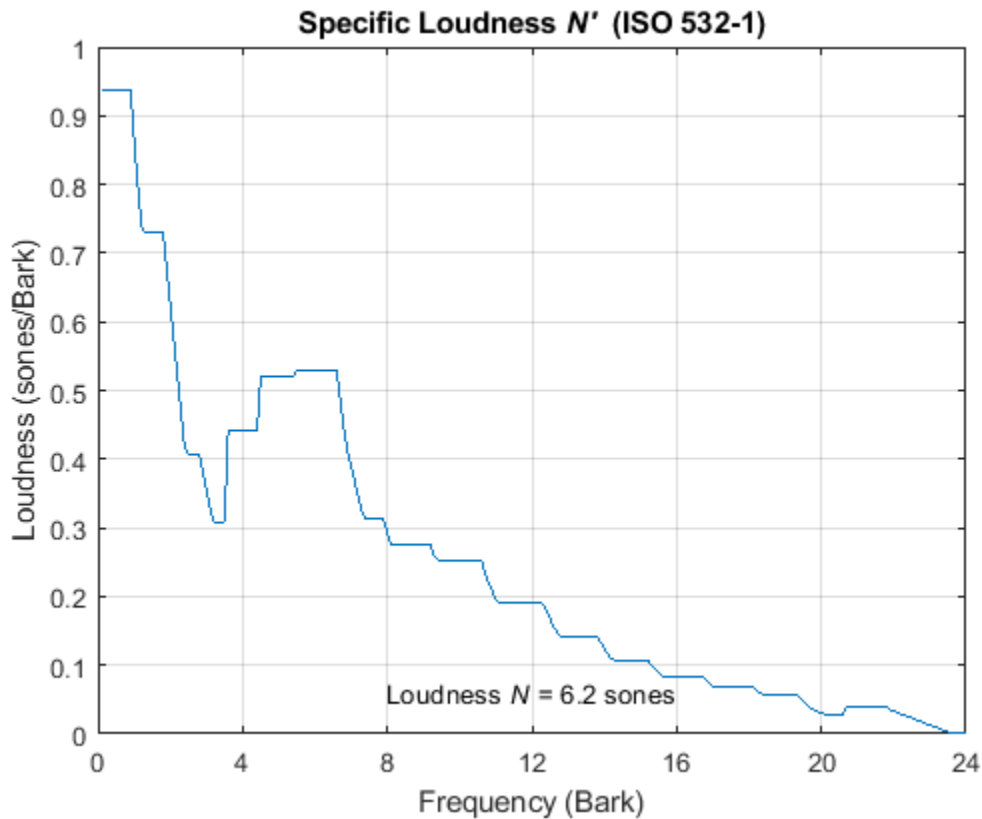
```
reset(spl)
[LCFnew,~,LCpeaknew] = spl(xhp);
plot(t,LCpeak,t,LCF,t,LCpeaknew,t,LCFnew)
legend('LCpeak (original)', 'LCF (original)', ...
       'LCpeak (with ear protection)', ...
       'LCF (with ear protection)', ...
       'Location','southeast')
title('SPL Measurement of Helicopter Noise')
xlabel('Time (seconds)')
ylabel('SPL (dB)')
ylim([68 100])
grid on
```

The attenuation might be less than expected by looking at the specification, but as you saw with the sharpness measurement, the helicopter noise is concentrated in the lower frequencies where any hearing protection is less efficient.

Look at the perceived loudness to get a better idea of how loud the helicopter now sounds.

```
acousticLoudness(xhp,FS,calib)
```



Loudness went from 42.5 to 6.2 sones. However, it might be easier to look at loudness in phons instead of sones, because phons are equivalent to the SPL of a 1 kHz tone. Convert the sone units to phons using `sone2phon`.

```
fprintf("Loudness without hearing protection:\t%.1f phons\n",sone2phon(42.5));
```

```
Loudness without hearing protection:    94.1 phons
```

```
fprintf("Loudness with hearing protection:\t%.1f phons\n",sone2phon(6.2));
```

```
Loudness with hearing protection:    66.3 phons
```

```
fprintf("Perceived noise reduction:\t\t%.1f phons (dB SPL at 1 kHz)\n",sone2phon(42.5)-sone2phon(6.2));
```

```
Perceived noise reduction:    27.8 phons (dB SPL at 1 kHz)
```

This hearing protection is rated as 24 dB, but the helicopter noise is perceived as approximately 28 dB quieter when wearing them.

Calculate the reduction in the psychoacoustic annoyance factor. Start by computing the mean sharpness.

```
[~,spechp,perchp] = acousticLoudness(xhp,FS,calib,"TimeVarying",true);
shp = acousticSharpness(spechp,'TimeVarying',true);
new_sharp = mean(shp(501:end))
```

```
new_sharp = 0.7258
```

Sharpness is below the threshold of 1.75, so it is ignored. Now, compute the mean fluctuation strength.

```
vacilhp = acousticFluctuation(spechp);  
fhp = mean(vacilhp(501:end,1));
```

Compute the new psychoacoustic annoyance factor. It has reduced from approximately 47 to 7.

```
N5hp = perchp(2); % N5 with hearing protection  
pahp = N5hp * (1 + abs(2.18/(N5hp^.4)*(.4*fhp)))  
  
pahp = 7.0230
```

Speech Command Recognition Code Generation on Raspberry Pi

This example shows how to deploy feature extraction and a convolutional neural network (CNN) for speech command recognition to Raspberry Pi™. To generate the feature extraction and network code, you use MATLAB Coder, MATLAB Support Package for Raspberry Pi Hardware, and the ARM® Compute Library. In this example, the generated code is an executable on your Raspberry Pi, which is called by a MATLAB script that displays the predicted speech command along with the signal and auditory spectrogram. Interaction between the MATLAB script and the executable on your Raspberry Pi is handled using the user datagram protocol (UDP). For details about audio preprocessing and network training, see “Speech Command Recognition Using Deep Learning” on page 1-331.

Prerequisites

- ARM processor that supports the NEON extension
- ARM Compute Library version 19.05 (on the target ARM hardware)
- Environment variables for the compilers and libraries

For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder).

Streaming Demonstration in MATLAB

Use the same parameters for the feature extraction pipeline and classification as developed in “Speech Command Recognition Using Deep Learning” on page 1-331.

Define the same sample rate the network was trained on (16 kHz). Define the classification rate and the number of audio samples input per frame. The feature input to the network is a Bark spectrogram that corresponds to 1 second of audio data. The Bark spectrogram is calculated for 25 ms windows with 10 ms hops. Calculate the number of individual spectrums in each spectrogram.

```
fs = 16000;
classificationRate = 20;
samplesPerCapture = fs/classificationRate;

segmentDuration = 1;
segmentSamples = round(segmentDuration*fs);

frameDuration = 0.025;
frameSamples = round(frameDuration*fs);

hopDuration = 0.010;
hopSamples = round(hopDuration*fs);

numSpectrumPerSpectrogram = floor((segmentSamples-frameSamples)/hopSamples) + 1;
```

Create an `audioFeatureExtractor` object to extract 50-band Bark spectrograms without window normalization. Calculate the number of elements in each spectrogram.

```
afe = audioFeatureExtractor( ...
    'SampleRate',fs, ...
    'FFTLength',512, ...
    'Window',hann(frameSamples,'periodic'), ...
    'OverlapLength',frameSamples - hopSamples, ...
```

```

    'barkSpectrum',true);

numBands = 50;
setExtractorParams(afe,'barkSpectrum','NumBands',numBands,'WindowNormalization',false);

```

```

numElementsPerSpectrogram = numSpectrumPerSpectrogram*numBands;

```

Load the pretrained CNN and labels.

```

load('commandNet.mat')
labels = trainedNet.Layers(end).Classes;
NumLabels = numel(labels);
BackGroundIdx = find(labels == 'background');

```

Define buffers and decision thresholds to post process network predictions.

```

probBuffer = single(zeros([NumLabels,classificationRate/2]));
YBuffer = single(NumLabels * ones(1, classificationRate/2));

countThreshold = ceil(classificationRate*0.2);
probThreshold = single(0.7);

```

Create an `audioDeviceReader` object to read audio from your device. Create a `dsp.AsyncBuffer` object to buffer the audio into chunks.

```

adr = audioDeviceReader('SampleRate',fs,'SamplesPerFrame',samplesPerCapture,'OutputDataType','single');
audioBuffer = dsp.AsyncBuffer(fs);

```

Create a `dsp.MatrixViewer` object and a `timescope` object to display the results.

```

matrixViewer = dsp.MatrixViewer("ColorBarLabel","Power per band (dB/Band)",...
    "XLabel","Frames",...
    "YLabel","Bark Bands", ...
    "Position",[400 100 600 250], ...
    "ColorLimits",[-4 2.6445], ...
    "AxisOrigin","Lower left corner", ...
    "Name","Speech Command Recognition using Deep Learning");

timeScope = timescope("SampleRate",fs, ...
    "YLimits",[-1 1], ...
    "Position",[400 380 600 250], ...
    "Name","Speech Command Recognition Using Deep Learning", ...
    "TimeSpanSource","Property", ...
    "TimeSpan",1, ...
    "BufferLength",fs, ...
    "YLabel","Amplitude", ...
    "ShowGrid",true);

```

Show the time scope and matrix viewer. Detect commands as long as both the time scope and matrix viewer are open or until the time limit is reached. To stop the live detection before the time limit is reached, close the time scope window or matrix viewer window.

```

show(timeScope)
show(matrixViewer)

```

```

timelimit = 10;

```

```

tic

```

```
while isVisible(timeScope) && isVisible(matrixViewer) && toc < timeLimit
    % Capture audio
    x = adr();
    write(audioBuffer,x);
    y = read(audioBuffer,fs,fs-samplesPerCapture);

    % Compute auditory features
    features = extract(afe,y);
    auditoryFeatures = log10(features + 1e-6);

    % Perform prediction
    probs = predict(trainedNet, auditoryFeatures);
    [~, YPredicted] = max(probs);

    % Perform statistical post processing
    YBuffer = [YBuffer(2:end),YPredicted];
    probBuffer = [probBuffer(:,2:end),probs(:)];

    [YModeIdx, count] = mode(YBuffer);
    maxProb = max(probBuffer(YModeIdx,:));

    if YModeIdx == single(BackGroundIdx) || single(count) < countThreshold || maxProb < probThresh
        speechCommandIdx = BackGroundIdx;
    else
        speechCommandIdx = YModeIdx;
    end

    % Update plots
    matrixViewer(auditoryFeatures');
    timeScope(x);

    if (speechCommandIdx == BackGroundIdx)
        timeScope.Title = ' ';
    else
        timeScope.Title = char(labels(speechCommandIdx));
    end
    drawnow limitrate
end
```

Hide the scopes.

```
hide(matrixViewer)
hide(timeScope)
```

Prepare MATLAB Code for Deployment

To create a function to perform feature extraction compatible with code generation, call `generateMATLABFunction` on the `audioFeatureExtractor` object. The `generateMATLABFunction` object function creates a standalone function that performs equivalent feature extraction and is compatible with code generation.

```
generateMATLABFunction(afe,'extractSpeechFeatures')
```

The `HelperSpeechCommandRecognitionRasPi` supporting function encapsulates the feature extraction and network prediction process demonstrated previously. So that the feature extraction is compatible with code generation, feature extraction is handled by the generated `extractSpeechFeatures` function. So that the network is compatible with code generation, the supporting function uses the `coder.loadDeepLearningNetwork` (MATLAB Coder) function to load

the network. The supporting function uses a `dsp.UDPReceiver` system object to send the auditory spectrogram and the index corresponding to the predicted speech command from Raspberry Pi to MATLAB. The supporting function uses the `dsp.UDPReceiver` system object to receive the audio captured by your microphone in MATLAB.

Generate Executable on Raspberry Pi

Replace the `hostIPAddress` with your machine's address. Your Raspberry Pi sends auditory spectrograms and the predicted speech command to this IP address.

```
hostIPAddress = coder.Constant('172.18.230.30');
```

Create a code generation configuration object to generate an executable program. Specify the target language as C++.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
```

Create a configuration object for deep learning code generation with the ARM compute library that is on your Raspberry Pi. Specify the architecture of the Raspberry Pi and attach the deep learning configuration object to the code generation configuration object.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmArchitecture = 'armv7';
dlcfg.ArmComputeVersion = '19.05';
cfg.DeepLearningConfig = dlcfg;
```

Use the Raspberry Pi Support Package function, `raspi`, to create a connection to your Raspberry Pi. In the following code, replace:

- `raspiname` with the name of your Raspberry Pi
- `pi` with your user name
- `password` with your password

```
r = raspi('raspiname','pi','password');
```

Create a `coder.hardware` (MATLAB Coder) object for Raspberry Pi and attach it to the code generation configuration object.

```
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Specify the build folder on the Raspberry Pi.

```
buildDir = '~/remoteBuildDir';
cfg.Hardware.BuildDir = buildDir;
```

Use an auto generated C++ main file for the generation of a standalone executable.

```
cfg.GenerateExampleMain = 'GenerateCodeAndCompile';
```

Call `codegen` (MATLAB Coder) to generate C++ code and the executable on your Raspberry Pi. By default, the Raspberry Pi application name is the same as the MATLAB function.

```
codegen -config cfg HelperSpeechCommandRecognitionRasPi -args {hostIPAddress} -report -v
```

Deploying code. This may take a few minutes.

Location of the generated elf : /home/pi/remoteBuildDir/MATLAB_ws/R2020b/C/Users/sporwal/OneDrive

```
### Using toolchain: GNU GCC Raspberry Pi
### 'C:\Users\sporwal\OneDrive - MathWorks\Documents\MATLAB\Examples\deeplearning_shared-ex00376
### Building 'HelperSpeechCommandRecognitionRasPi': make -f HelperSpeechCommandRecognitionRasPi.
```

Warning: Function 'HelperSpeechCommandRecognitionRasPi' does not terminate due to an infinite loop

Warning in ==> HelperSpeechCommandRecognitionRasPi Line: 86 Column: 1
Code generation successful (with warnings): View report

Initialize Application on Raspberry Pi

Create a command to open the HelperSpeechCommandRasPi application on Raspberry Pi.
Use system to send the command to your Raspberry Pi.

```
applicationName = 'HelperSpeechCommandRecognitionRasPi';

applicationDirPaths = raspi.utils.getRemoteBuildDirectory('applicationName',applicationName);
targetDirPath = applicationDirPaths{1}.directory;

exeName = strcat(applicationName,'.elf');
command = ['cd ' targetDirPath ' ; ./' exeName ' &> 1 &'];

system(r,command);
```

Create a dsp.UDPReceiver system object to send audio captured in MATLAB to your Raspberry Pi.
Update the targetIPAddress for your Raspberry Pi. Raspberry Pi receives the captured audio from the same port using the dsp.UDPReceiver system object.

```
targetIPAddress = '172.18.228.24';
UDPSend = dsp.UDPSender('RemoteIPPort',26000,'RemoteIPAddress',targetIPAddress);
```

Create a dsp.UDPReceiver system object to receive auditory features and the predicted speech command index from your Raspberry Pi. Each UDP packet received from the Raspberry Pi consists of auditory features in column-major order followed by the predicted speech command index. The maximum message length for the dsp.UDPReceiver object is 65507 bytes. Calculate the buffer size to accommodate the maximum number of UDP packets.

```
sizeofFloatInBytes = 4;
maxUDPMessageLength = floor(65507/sizeofFloatInBytes);
samplesPerPacket = 1 + numElementsPerSpectrogram;
numPackets = floor(maxUDPMessageLength/samplesPerPacket);
bufferSize = numPackets*samplesPerPacket*sizeofFloatInBytes;
```

```
UDPReceive = dsp.UDPReceiver("LocalIPPort",21000, ...
    "MessageDataType","single", ...
    "MaximumMessageLength",samplesPerPacket, ...
    "ReceiveBufferSize",bufferSize);
```

Reduce initialization overhead by sending a frame of zeros to the executable running on your Raspberry Pi.

```
UDPSend(zeros(samplesPerCapture,1,"single"));
```

Perform Speech Command Recognition Using Deployed Code

Detect commands as long as both the time scope and matrix viewer are open or until the time limit is reached. To stop the live detection before the time limit is reached, close the time scope or matrix viewer window.


```

show(timeScope)
show(matrixViewer)

timeLimit = 20;

tic
while isVisible(timeScope) && isVisible(matrixViewer) && toc < timeLimit
    % Capture audio and send that to RasPi
    x = adr();
    UDPSend(x);

    % Receive data packet from RasPi
    udpRec = UDPReceive();

    if ~isempty(udpRec)
        % Extract predicted index, the last sample of received UDP packet
        speechCommandIdx = udpRec(end);

        % Extract auditory spectrogram
        spec = reshape(udpRec(1:numElementsPerSpectrogram), [numBands, numSpectrumPerSpectrogram]);

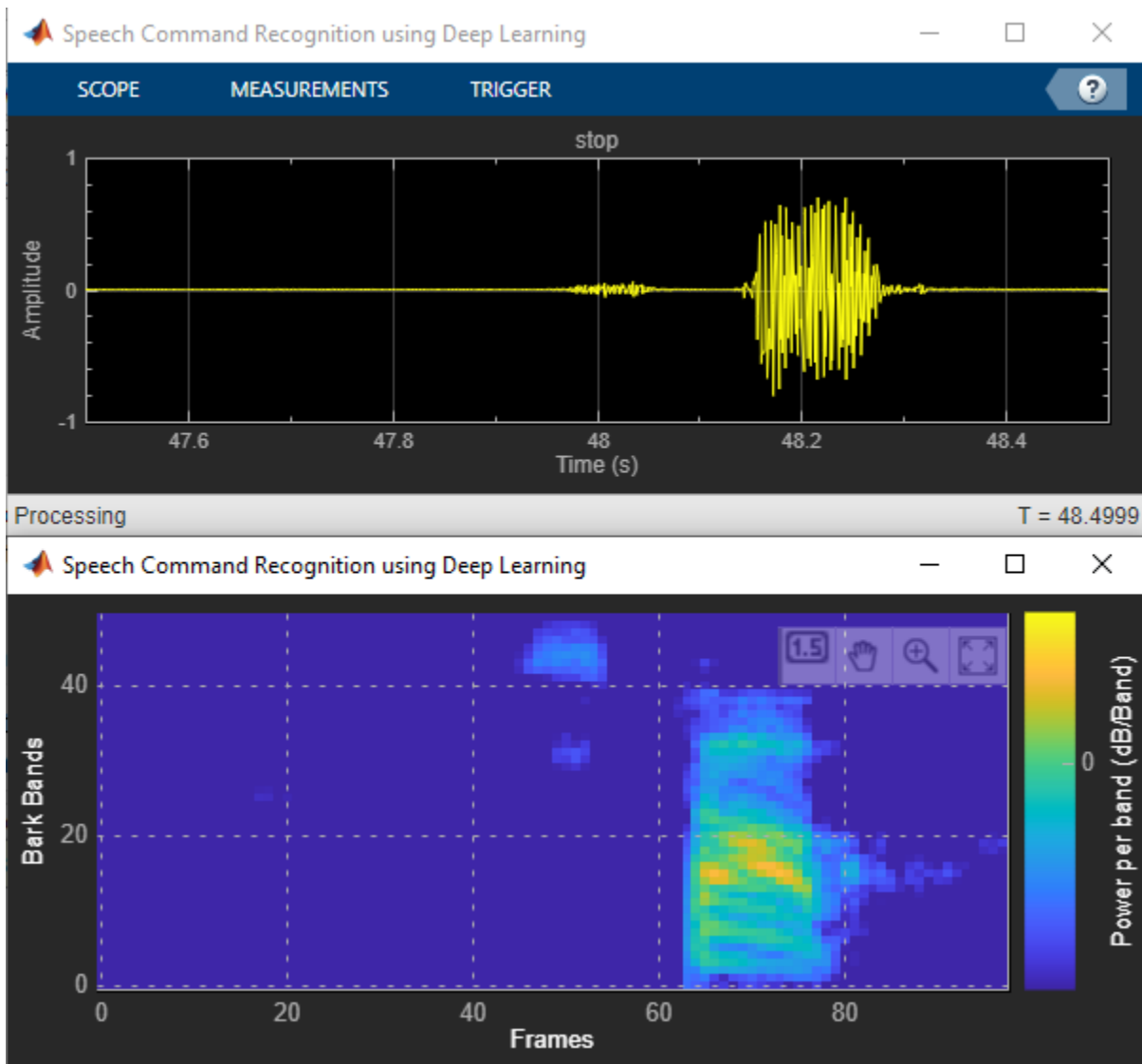
        % Display time domain signal and auditory spectrogram
        timeScope(x)
        matrixViewer(spec)

        if speechCommandIdx == BackgroundIdx
            timeScope.Title = ' ';
        else
            timeScope.Title = char(labels(speechCommandIdx));
        end

        drawnow limitrate
    end
end

hide(matrixViewer)
hide(timeScope)

```



To stop the executable on your Raspberry Pi, use `stopExecutable`. Release the UDP objects.

```
stopExecutable(codertarget.raspi.raspberrypi,exeName)
```

```
release(UDPSend)
release(UDPReceive)
```

Profile Using PIL Workflow

You can measure the execution time taken on the Raspberry Pi using a PIL (processor-in-loop) workflow. The `ProfileSpeechCommandRecognitionRaspi` supporting function is the equivalent of the `HelperSpeechCommandRecognitionRaspi` function, except that the former returns the speech command index and auditory spectrogram while the latter sends the same parameters using UDP. The time taken by the UDP calls is less than 1 ms, which is relatively small compared to the overall execution time.

Create a PIL configuration object.

```
cfg = coder.config('lib','ecoder',true);
cfg.VerificationMode = 'PIL';
```

Set the ARM compute library and architecture.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
cfg.DeepLearningConfig = dlcfg ;
cfg.DeepLearningConfig.ArmArchitecture = 'armv7';
cfg.DeepLearningConfig.ArmComputeVersion = '19.05';
```

Set up the connection with your target hardware.

```
if (~exist('r','var'))
    r = raspi('raspiname','pi','password');
end
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Set the build directory and target language.

```
buildDir = '~/remoteBuildDir';
cfg.Hardware.BuildDir = buildDir;
cfg.TargetLang = 'C++';
```

Enable profiling and then generate the PIL code. A MEX file named `ProfileSpeechCommandRecognition_pil` is generated in your current folder.

```
cfg.CodeExecutionProfiling = true;
codegen -config cfg ProfileSpeechCommandRecognitionRaspi -args {rand(samplesPerCapture, 1, 'single')}

### Target device has no native communication support. Checking connectivity configuration registers...
Deploying code. This may take a few minutes.
### Target device has no native communication support. Checking connectivity configuration registers...
### Connectivity configuration for function 'ProfileSpeechCommandRecognitionRaspi': 'Raspberry Pi'
### Using toolchain: GNU GCC Raspberry Pi
### Creating 'C:\Users\sporwal\OneDrive - MathWorks\Documents\MATLAB\Examples\deeplearning_shared-ex00376b\ProfileSpeechCommandRecognitionRaspi_ca': make -f ProfileSpeechCommandRecognitionRaspi_ca.mk
### Using toolchain: GNU GCC Raspberry Pi
### Creating 'C:\Users\sporwal\OneDrive - MathWorks\Documents\MATLAB\Examples\deeplearning_shared-ex00376b\ProfileSpeechCommandRecognitionRaspi': make -f ProfileSpeechCommandRecognitionRaspi.mk
Location of the generated elf : /home/pi/remoteBuildDir/MATLAB_ws/R2020b/C/Users/sporwal/OneDrive - MathWorks\Documents\MATLAB\Examples\deeplearning_shared-ex00376b\ProfileSpeechCommandRecognitionRaspi.elf
### Using toolchain: GNU GCC Raspberry Pi
### 'C:\Users\sporwal\OneDrive - MathWorks\Documents\MATLAB\Examples\deeplearning_shared-ex00376b\ProfileSpeechCommandRecognitionRaspi': make -f ProfileSpeechCommandRecognitionRaspi.mk
```

Code generation successful: [View report](#)

Evaluate Raspberry Pi Execution Time

Call the generated PIL function multiple times to get the average execution time.

```
testDur = 50e-3;
numCalls = 100;

for k = 1:numCalls
    x = pinknoise(fs*testDur,'single');
    [speechCommandIdx, auditoryFeatures] = ProfileSpeechCommandRecognitionRaspi_pil(x);
end
```

```

### Starting application: 'codegen\lib\ProfileSpeechCommandRecognitionRaspi\pil\ProfileSpeechCom
To terminate execution: clear ProfileSpeechCommandRecognitionRaspi_pil
### Launching application ProfileSpeechCommandRecognitionRaspi.elf...
Execution profiling data is available for viewing. Open Simulation Data Inspector.
Execution profiling report available after termination.

```

Terminate the PIL execution.

```
clear ProfileSpeechCommandRecognitionRaspi_pil
```

```
### Host application produced the following standard output (stdout) and standard error (stderr)
```

```
Execution profiling report: report(getCoderExecutionProfile('ProfileSpeechCommandRecognitionRaspi'))
```

Generate an execution profile report to evaluate execution time.

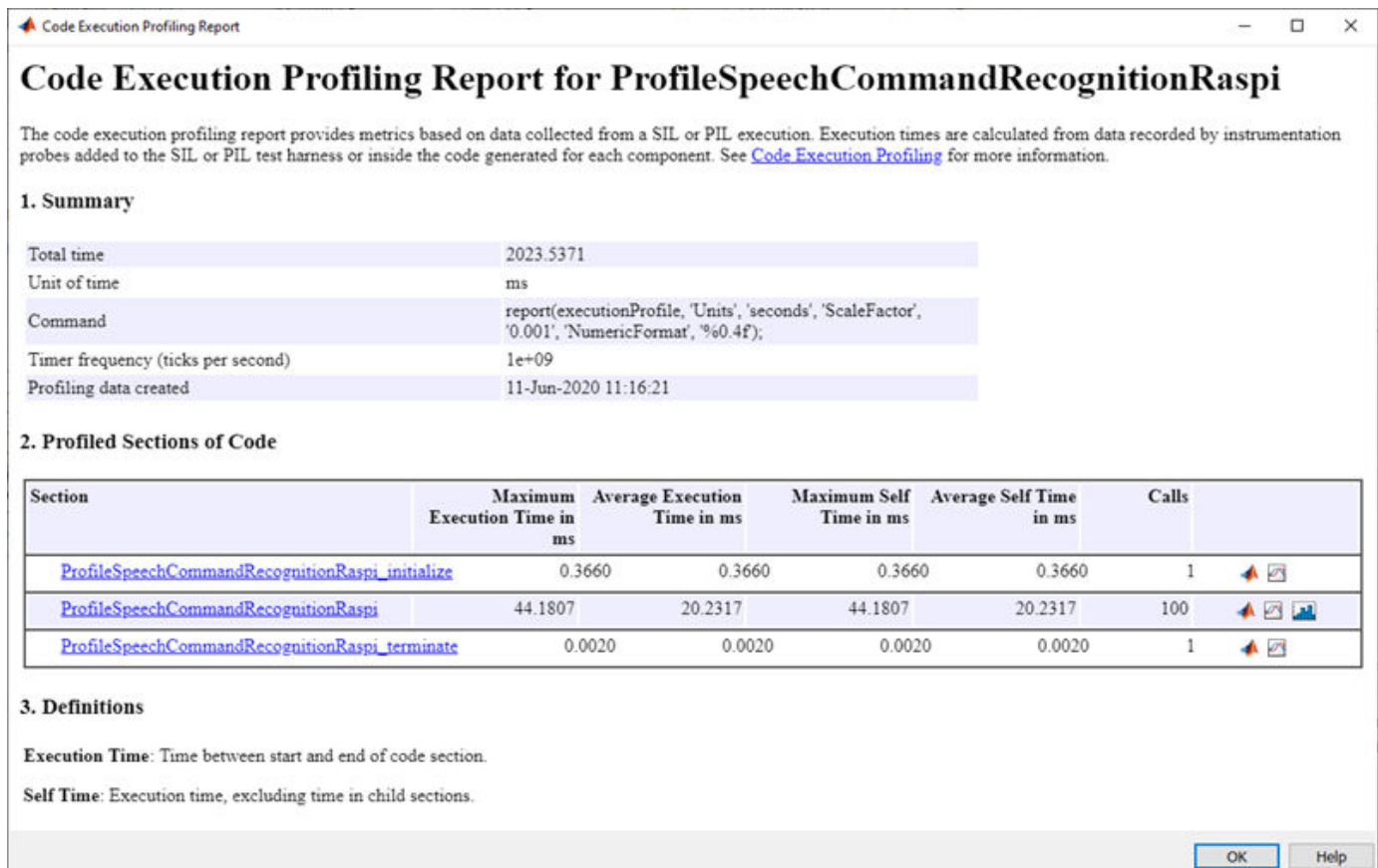
```

executionProfile = getCoderExecutionProfile('ProfileSpeechCommandRecognitionRaspi');
report(executionProfile, ...
    'Units','Seconds', ...
    'ScaleFactor','1e-03', ...
    'NumericFormat','%0.4f')

```

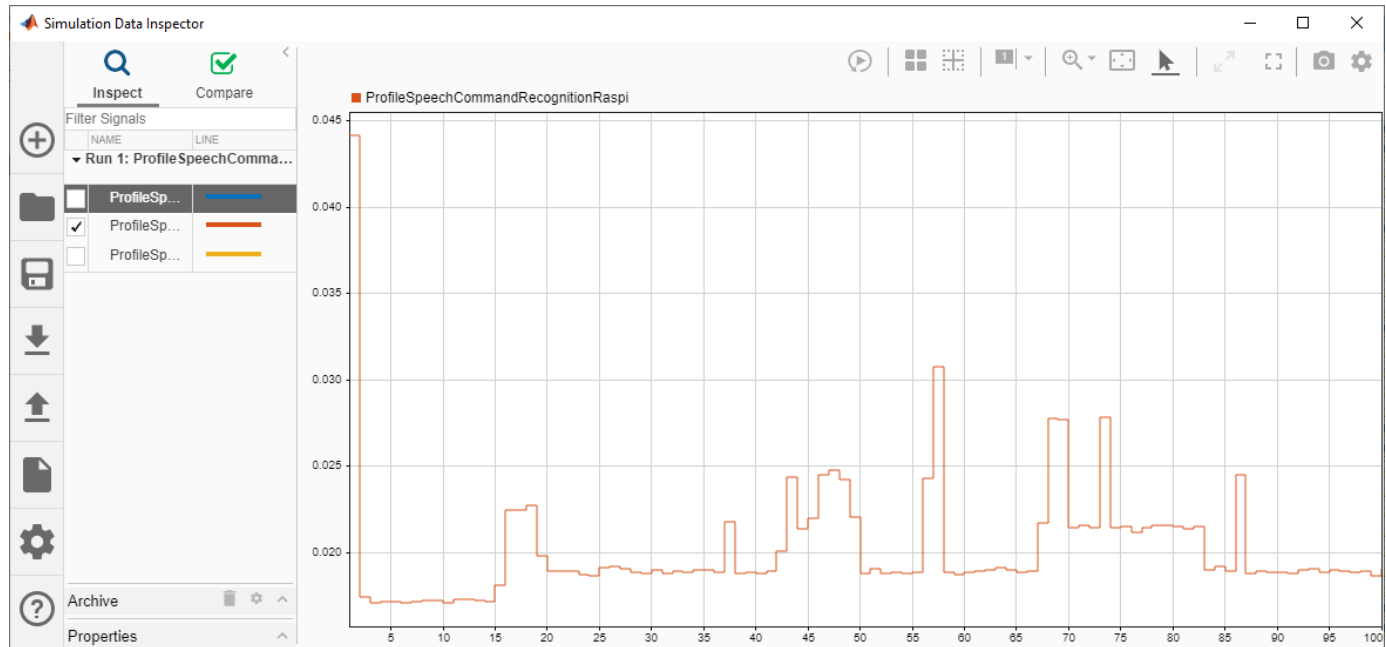
```
ans =
```

```
'C:\Users\sporwal\OneDrive - MathWorks\Documents\MATLAB\Examples\deeplearning_shared-ex00376115\
```



The maximum execution time taken by the ProfileSpeechCommandRecognitionRaspi function is nearly twice the average execution time. You can notice that the execution time is maximum for the

first call of the PIL function and it is due to the initialization happening in the first call. The average execution time is approximately 20 ms, which is below the 50 ms budget (audio capture time). The performance is measured on Raspberry Pi 4 Model B Rev 1.1.



Speech Command Recognition Code Generation with Intel MKL-DNN

This example shows how to deploy feature extraction and a convolutional neural network (CNN) for speech command recognition on Intel® processors. To generate the feature extraction and network code, you use MATLAB Coder and the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN). In this example, the generated code is a MATLAB executable (MEX) function, which is called by a MATLAB script that displays the predicted speech command along with the time domain signal and auditory spectrogram. For details about audio preprocessing and network training, see “Speech Command Recognition Using Deep Learning” on page 1-331.

Prerequisites

- The MATLAB Coder Interface for Deep Learning Support Package
- Xeon processor with support for Intel Advanced Vector Extensions 2 (Intel AVX2)
- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- Environment variables for Intel MKL-DNN

For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder).

Streaming Demonstration in MATLAB

Use the same parameters for the feature extraction pipeline and classification as developed in “Speech Command Recognition Using Deep Learning” on page 1-331.

Define the same sample rate the network was trained on (16 kHz). Define the classification rate and the number of audio samples input per frame. The feature input to the network is a Bark spectrogram that corresponds to 1 second of audio data. The Bark spectrogram is calculated for 25 ms windows with 10 ms hops.

```
fs = 16000;
classificationRate = 20;
samplesPerCapture = fs/classificationRate;

segmentDuration = 1;
segmentSamples = round(segmentDuration*fs);

frameDuration = 0.025;
frameSamples = round(frameDuration*fs);

hopDuration = 0.010;
hopSamples = round(hopDuration*fs);
```

Create an `audioFeatureExtractor` object to extract 50-band Bark spectrograms without window normalization.

```
afe = audioFeatureExtractor( ...
    'SampleRate',fs, ...
    'FFTLength',512, ...
    'Window',hann(frameSamples,'periodic'), ...
    'OverlapLength',frameSamples - hopSamples, ...
    'barkSpectrum',true);
```

```
numBands = 50;
setExtractorParams(afe, 'barkSpectrum', 'NumBands', numBands, 'WindowNormalization', false);
```

Load the pretrained convolutional neural network and labels.

```
load('commandNet.mat')
labels = trainedNet.Layers(end).Classes;
numLabels = numel(labels);
backgroundIdx = find(labels == 'background');
```

Define buffers and decision thresholds to post process network predictions.

```
probBuffer = single(zeros([numLabels, classificationRate/2]));
YBuffer = single(numLabels * ones(1, classificationRate/2));

countThreshold = ceil(classificationRate*0.2);
probThreshold = single(0.7);
```

Create an `audioDeviceReader` object to read audio from your device. Create a `dsp.AsyncBuffer` object to buffer the audio into chunks.

```
adr = audioDeviceReader('SampleRate', fs, 'SamplesPerFrame', samplesPerCapture, 'OutputDataType', 'single');
audioBuffer = dsp.AsyncBuffer(fs);
```

Create a `dsp.MatrixViewer` object and a `timescope` object to display the results.

```
matrixViewer = dsp.MatrixViewer("ColorBarLabel", "Power per band (dB/Band)", ...
    "XLabel", "Frames", ...
    "YLabel", "Bark Bands", ...
    "Position", [400 100 600 250], ...
    "ColorLimits", [-4 2.6445], ...
    "AxisOrigin", 'Lower left corner', ...
    "Name", "Speech Command Recognition Using Deep Learning");

timeScope = timescope('SampleRate', fs, ...
    'YLimits', [-1 1], 'Position', [400 380 600 250], ...
    'Name', 'Speech Command Recognition Using Deep Learning', ...
    'TimeSpanSource', 'Property', ...
    'TimeSpan', 1, ...
    'BufferLength', fs);
```

```
timeScope.YLabel = 'Amplitude';
timeScope.ShowGrid = true;
```

Show the time scope and matrix viewer. Detect commands as long as both the time scope and matrix viewer are open or until the time limit is reached. To stop the live detection before the time limit is reached, close the time scope window or matrix viewer window.

```
show(timeScope)
show(matrixViewer)
timeLimit = 10;

tic
while isVisible(timeScope) && isVisible(matrixViewer) && toc < timeLimit
    %% Capture Audio
    x = adr();
    write(audioBuffer, x);
    y = read(audioBuffer, fs, fs-samplesPerCapture);
```

```
% Compute auditory features
features = extract(afe,y);
auditory_features = log10(features + 1e-6);

% Transpose to get the auditory spectrum
auditorySpectrum = auditory_features';

% Perform prediction
probs = predict(trainedNet, auditory_features);
[~, YPredicted] = max(probs);

% Perform statistical post processing
YBuffer = [YBuffer(2:end),YPredicted];
probBuffer = [probBuffer(:,2:end),probs(:)];

[YMode_idx, count] = mode(YBuffer);
count = single(count);
maxProb = max(probBuffer(YMode_idx,:));

if (YMode_idx == single(backgroundIdx) || count < countThreshold || maxProb < probThreshold)
    speechCommandIdx = backgroundIdx;
else
    speechCommandIdx = YMode_idx;
end

% Update plots
matrixViewer(auditorySpectrum);
timeScope(x);

if (speechCommandIdx == backgroundIdx)
    timeScope.Title = ' ';
else
    timeScope.Title = char(labels(speechCommandIdx));
end
drawnow
end
```

Hide the scopes.

```
hide(matrixViewer)
hide(timeScope)
```

Prepare MATLAB Code for Deployment

To create a function to perform feature extraction compatible with code generation, call `generateMATLABFunction` on the `audioFeatureExtractor` object. The `generateMATLABFunction` object function creates a standalone function that performs equivalent feature extraction and is compatible with code generation.

```
generateMATLABFunction(afe,'extractSpeechFeatures')
```

The `HelperSpeechCommandRecognition` supporting function encapsulates the feature extraction and network prediction process demonstrated previously. So that the feature extraction is compatible with code generation, feature extraction is handled by the generated `extractSpeechFeatures` function. So that the network is compatible with code generation, the supporting function uses the `coder.loadDeepLearningNetwork` (MATLAB Coder) function to load the network.

Use the `HelperSpeechCommandRecognition` function to perform live detection of speech commands.

```
show(timeScope)
show(matrixViewer)
timeLimit = 10;

tic
while isVisible(timeScope) && isVisible(matrixViewer) && toc < timeLimit
    x = adr();

    [speechCommandIdx, auditorySpectrum] = HelperSpeechCommandRecognition(x);

    matrixViewer(auditorySpectrum);
    timeScope(x);

    if (speechCommandIdx == backgroundIdx)
        timeScope.Title = ' ';
    else
        timeScope.Title = char(labels(speechCommandIdx));
    end
    drawnow
end
```

Hide the scopes.

```
hide(timeScope)
hide(matrixViewer)
```

Generate MATLAB Executable

Create a code generation configuration object for generation of an executable program. Specify the target language as C++.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
```

Create a configuration object for deep learning code generation with the MKL-DNN library. Attach the configuration object to the code generation configuration object.

```
dlcfg = coder.DeepLearningConfig('mkl_dnn');
cfg.DeepLearningConfig = dlcfg;
```

Call `codegen` (MATLAB Coder) to generate C++ code for the `HelperSpeechCommandRecognition` function. Specify the configuration object and prototype arguments. A MEX file named `HelperSpeechCommandRecognition_mex` is generated to your current folder.

```
codegen HelperSpeechCommandRecognition -config cfg -args {rand(samplesPerCapture, 1, 'single')}
```

Code generation successful: [View report](#)

Perform Speech Command Recognition Using Deployed Code

Show the time scope and matrix viewer. Detect commands using the generated MEX for as long as both the time scope and matrix viewer are open or until the time limit is reached. To stop the live detection before the time limit is reached, close the time scope window or matrix viewer window.

```
show(timeScope)
show(matrixViewer)
```

```
timeLimit = 20;

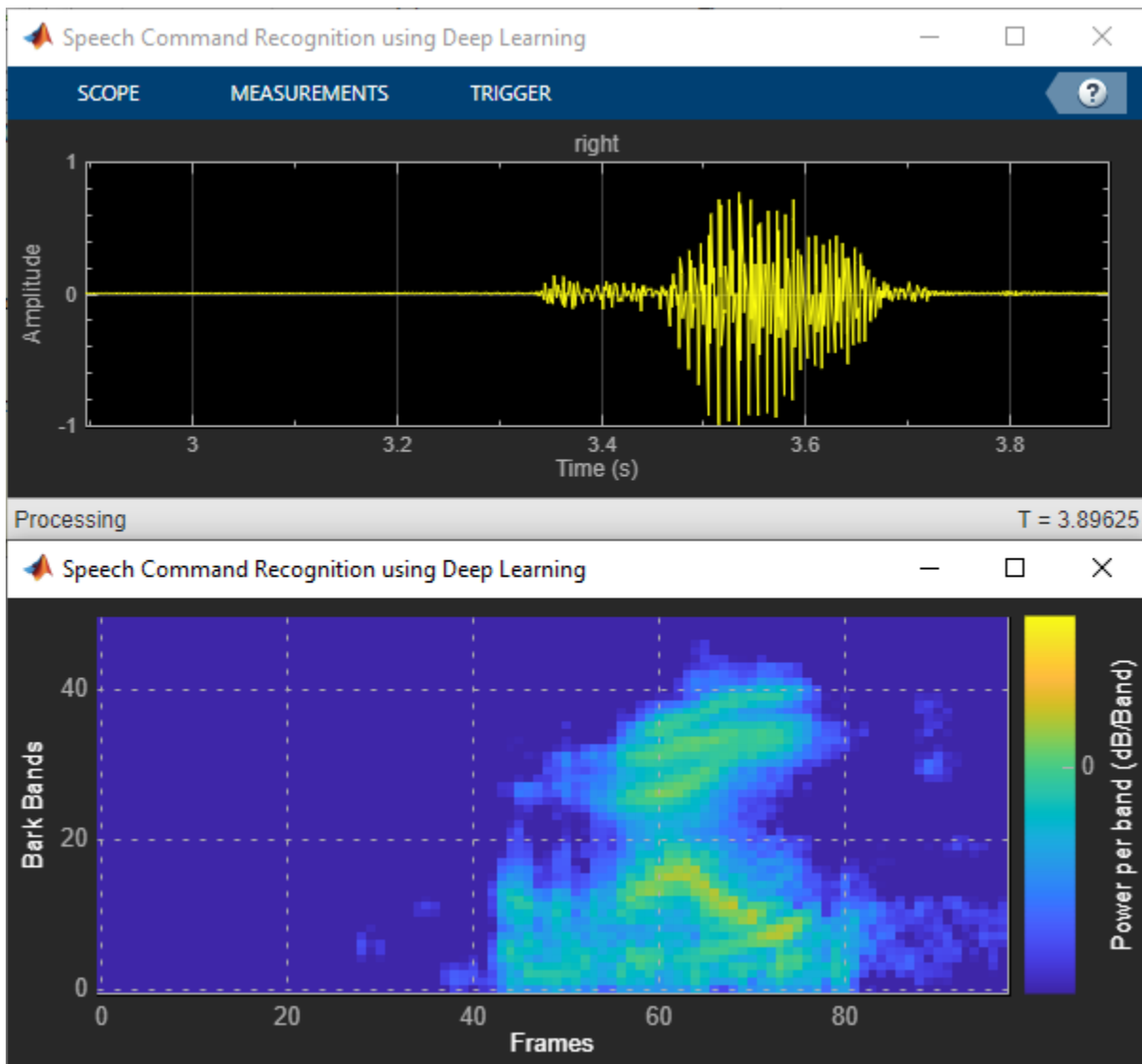
tic
while isVisible(timeScope) && isVisible(matrixViewer) && toc < timeLimit
    x = adr();

    [speechCommandIdx, auditorySpectrum] = HelperSpeechCommandRecognition_mex(x);

    matrixViewer(auditorySpectrum);
    timeScope(x);

    if (speechCommandIdx == backgroundIdx)
        timeScope.Title = ' ';
    else
        timeScope.Title = char(labels(speechCommandIdx));
    end
    drawnow
end

hide(matrixViewer)
hide(timeScope)
```



Evaluate MEX Execution Time

Use `tic` and `toc` to compare the execution time to run the simulation completely in MATLAB with the execution time of the MEX function.

Measure the performance of the simulation code.

```
testDur = 50e-3;
x = pinknoise(fs*testDur,'single');
numLoops = 100;
tic
for k = 1:numLoops
    [speechCommandIdx, auditory_features] = HelperSpeechCommandRecognition(x);
end
exeTime = toc;
fprintf('SIM execution time per 50 ms of audio = %0.4f ms\n',(exeTime/numLoops)*1000);
SIM execution time per 50 ms of audio = 6.6746 ms
```

Measure the performance of the MEX code.

```
tic
for k = 1:numLoops
    [speechCommandIdx, auditory_features] = HelperSpeechCommandRecognition_mex(x);
end
exeTimeMex = toc;
fprintf('MEX execution time per 50 ms of audio = %0.4f ms\n', (exeTimeMex/numLoops)*1000);

MEX execution time per 50 ms of audio = 1.5188 ms
```

Evaluate the performance gained from using the MEX function. This performance test is performed on a machine using NVIDIA Quadro P620 (Version 26) GPU and Intel(R) Xeon(R) W-2133 CPU running at 3.60 GHz.

```
PerformanceGain = exeTime/exeTimeMex
```

```
PerformanceGain = 4.3945
```

Time-Frequency Masking for Harmonic-Percussive Source Separation

Time-frequency masking is the process of applying weights to the bins of a time-frequency representation to enhance, diminish, or isolate portions of audio.

The goal of harmonic-percussive source separation (HPSS) is to decompose an audio signal into harmonic and percussive components. Applications of HPSS include audio remixing, improving the quality of chroma features, tempo estimation, and time-scale modification [1 on page 1-0]. Another use of HPSS is as a parallel representation when creating a late fusion deep learning system. Many of the top performing systems of the Detection and Classification of Acoustic Scenes and Events (DCASE) 2017 and 2018 challenges used HPSS for this reason.

This example walks through the algorithm described in [1 on page 1-0] to apply time-frequency masking to the task of harmonic-percussive source separation.

For an example of deriving time-frequency masks using deep learning, see “Cocktail Party Source Separation Using Deep Learning Networks” on page 1-384.

Create Harmonic-Percussive Mixture

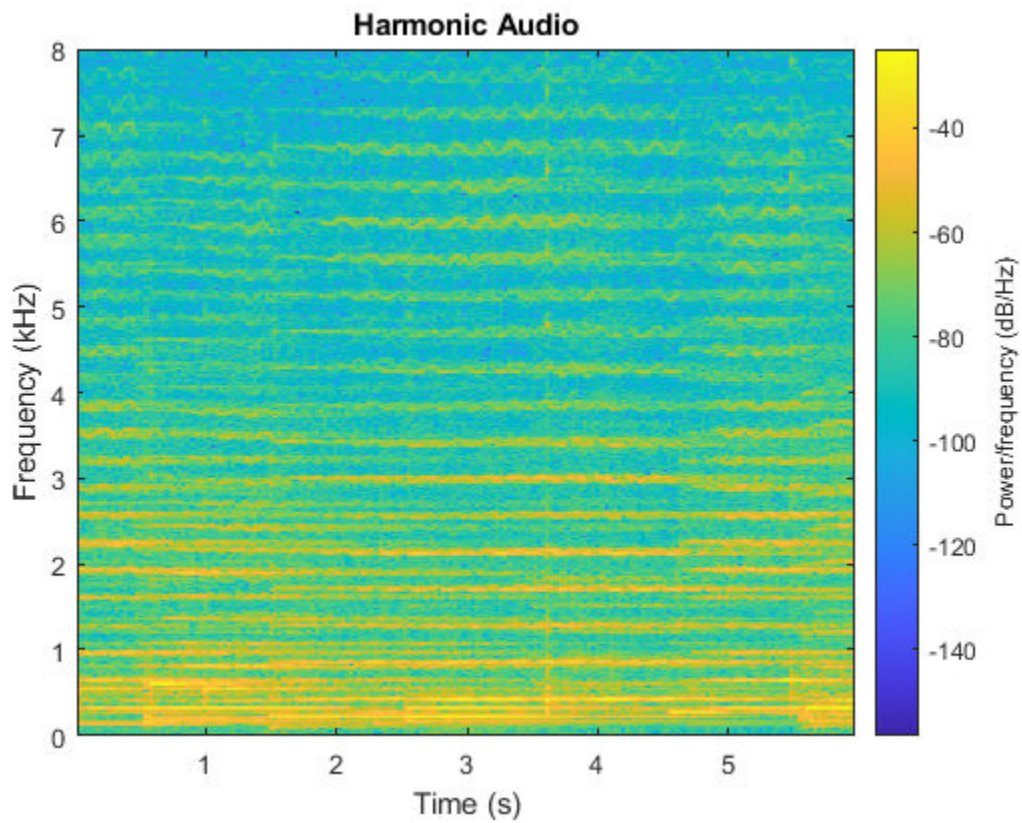
Read in harmonic and percussive audio files. Both have a sample rate of 16 kHz.

```
[harmonicAudio,fs] = audioread("violin.wav");
percussiveAudio = audioread("drums.wav");
```

Listen to the harmonic signal and plot the spectrogram. Note that there is continuity along the horizontal (time) axis.

```
sound(harmonicAudio,fs)
```

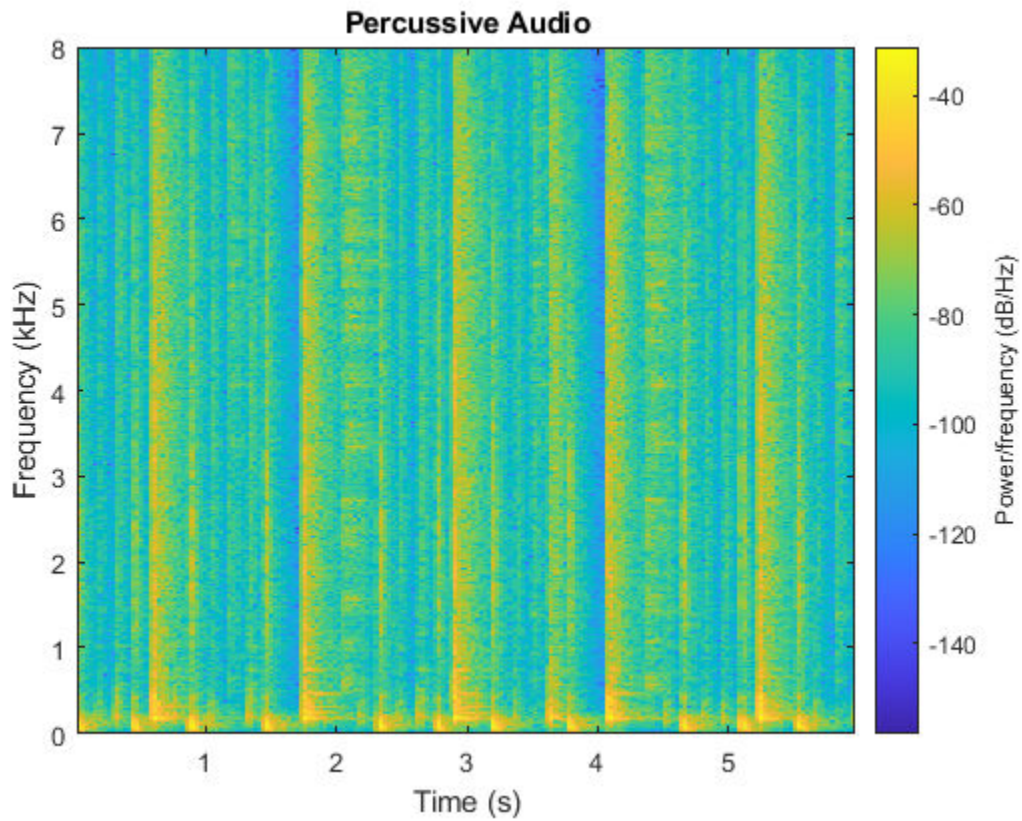
```
spectrogram(harmonicAudio,1024,512,1024,fs,"yaxis")
title("Harmonic Audio")
```



Listen to the percussive signal and plot the spectrogram. Note that there is continuity along the vertical (frequency) axis.

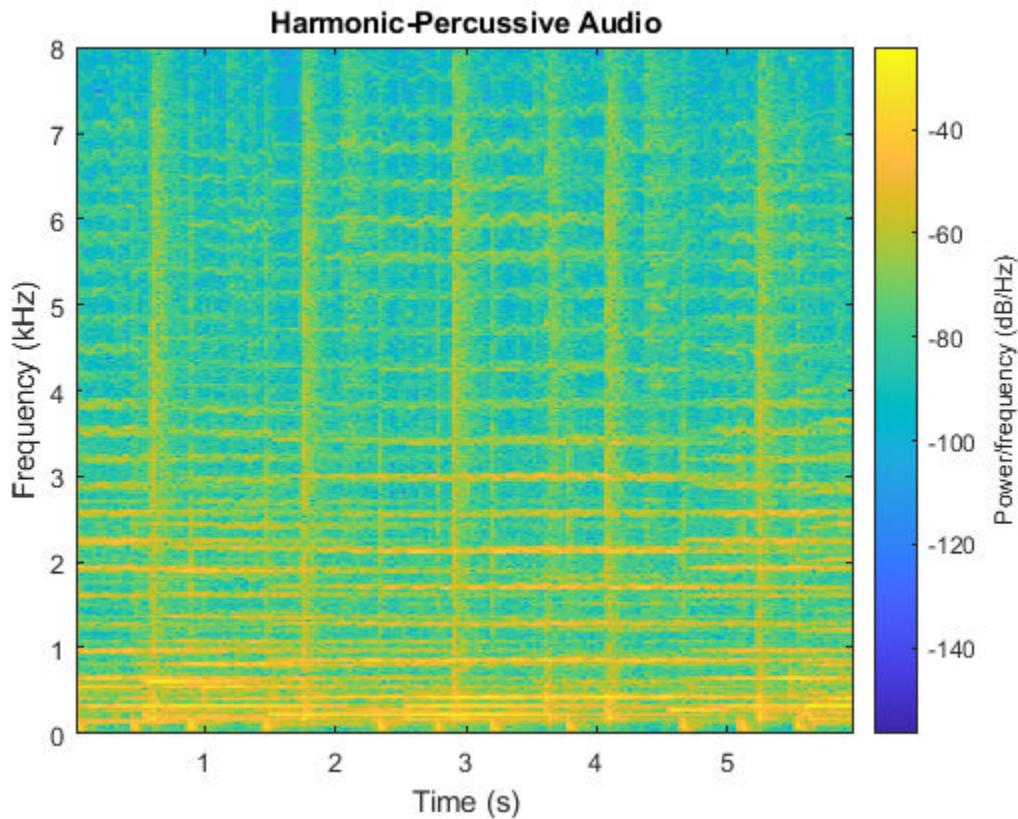
```
sound(percussiveAudio,fs)
```

```
spectrogram(percussiveAudio,1024,512,1024,fs,"yaxis")  
title("Percussive Audio")
```



Mix the harmonic and percussive signals. Listen to the harmonic-percussive audio and plot the spectrogram.

```
mix = harmonicAudio + percussiveAudio;  
  
sound(mix,fs)  
  
spectrogram(mix,1024,512,1024,fs,"yaxis")  
title("Harmonic-Percussive Audio")
```

The HPSS proposed by [1 on page 1-0] creates two enhanced spectrograms: a harmonic-enhanced spectrogram and a percussive-enhanced spectrogram. The harmonic-enhanced spectrogram is created by applying median filtering along the time axis. The percussive-enhanced spectrogram is created by applying median filtering along the frequency axis. The enhanced spectrograms are then compared to create harmonic and percussive time-frequency masks. In the simplest form, the masks are binary.

HPSS Using Binary Mask

Convert the mixed signal to a half-sided magnitude short-time Fourier transform (STFT).

```
win = sqrt(hann(1024,"periodic"));
overlapLength = floor(numel(win)/2);
fftLength = 2^nextpow2(numel(win) + 1);
y = stft(mix, ...
    "Window",win, ...
    "OverlapLength",overlapLength, ...
    "FFTLength",fftLength, ...
    "Centered",true);
halfIdx = 1:ceil(size(y,1)/2);
yhalf = y(halfIdx,:);
ymag = abs(yhalf);
```

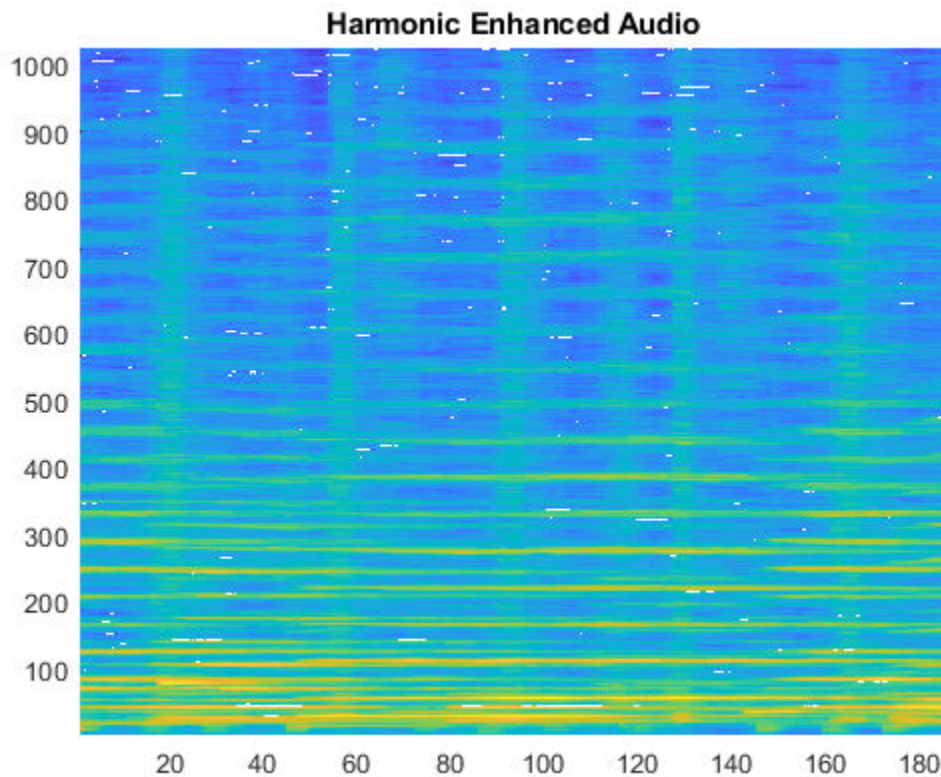
Apply median smoothing along the time axis to enhance the harmonic audio and diminish the percussive audio. Use a filter length of 200 ms, as suggested by [1 on page 1-0]. Plot the power spectrum of the harmonic-enhanced audio.


```

timeFilterLength = 0.2;
timeFilterLengthInSamples = timeFilterLength/((numel(win) - overlapLength)/fs);
ymagharm = movmedian(ymag,timeFilterLengthInSamples,2);

surf(flipud(log10(ymagharm.^2)), "EdgeColor", "none")
title("Harmonic Enhanced Audio")
view([0,90])
axis tight

```



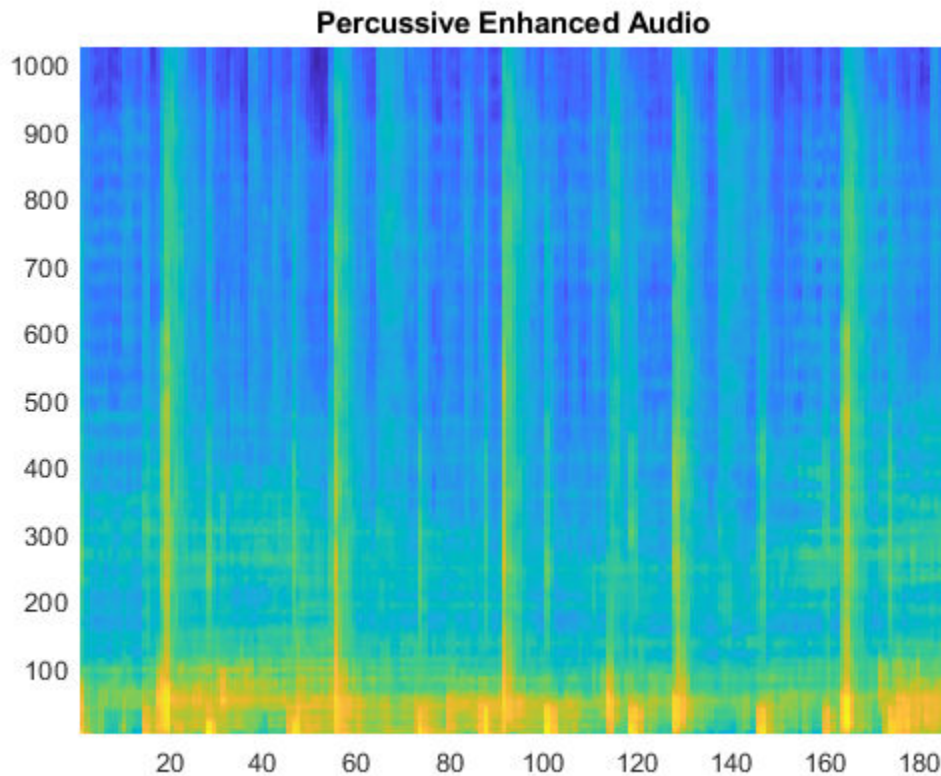
Apply median smoothing along the frequency axis to enhance the percussive audio and diminish the harmonic audio. Use a filter length of 500 Hz, as suggested by [1 on page 1-0]. Plot the power spectrum of the percussive-enhanced audio.

```

frequencyFilterLength = 500;
frequencyFilterLengthInSamples = frequencyFilterLength/(fs/fftLength);
ymagperc = movmedian(ymag,frequencyFilterLengthInSamples,1);

surf(flipud(log10(ymagperc.^2)), "EdgeColor", "none")
title("Percussive Enhanced Audio")
view([0,90])
axis tight

```



To create a binary mask, first sum the percussive- and harmonic-enhanced spectrums to determine the total magnitude per bin.

```
totalMagnitudePerBin = ymagharm + ymagperc;
```

If the magnitude in a given harmonic-enhanced or percussive-enhanced bin accounts for more than half of the total magnitude of that bin, then assign that bin to the corresponding mask.

```
harmonicMask = ymagharm > (totalMagnitudePerBin*0.5);
percussiveMask = ymagperc > (totalMagnitudePerBin*0.5);
```

Apply the harmonic and percussive masks and then return the masked audio to the time domain.

```
yharm = harmonicMask.*yhalf;
yperc = percussiveMask.*yhalf;
```

Mirror the half-sided spectrum to create a two-sided conjugate symmetric spectrum.

```
yharm = cat(1,yharm,flipud(conj(yharm)));
yperc = cat(1,yperc,flipud(conj(yperc)));
```

Perform the inverse short-time Fourier transform to return the signals to the time domain.

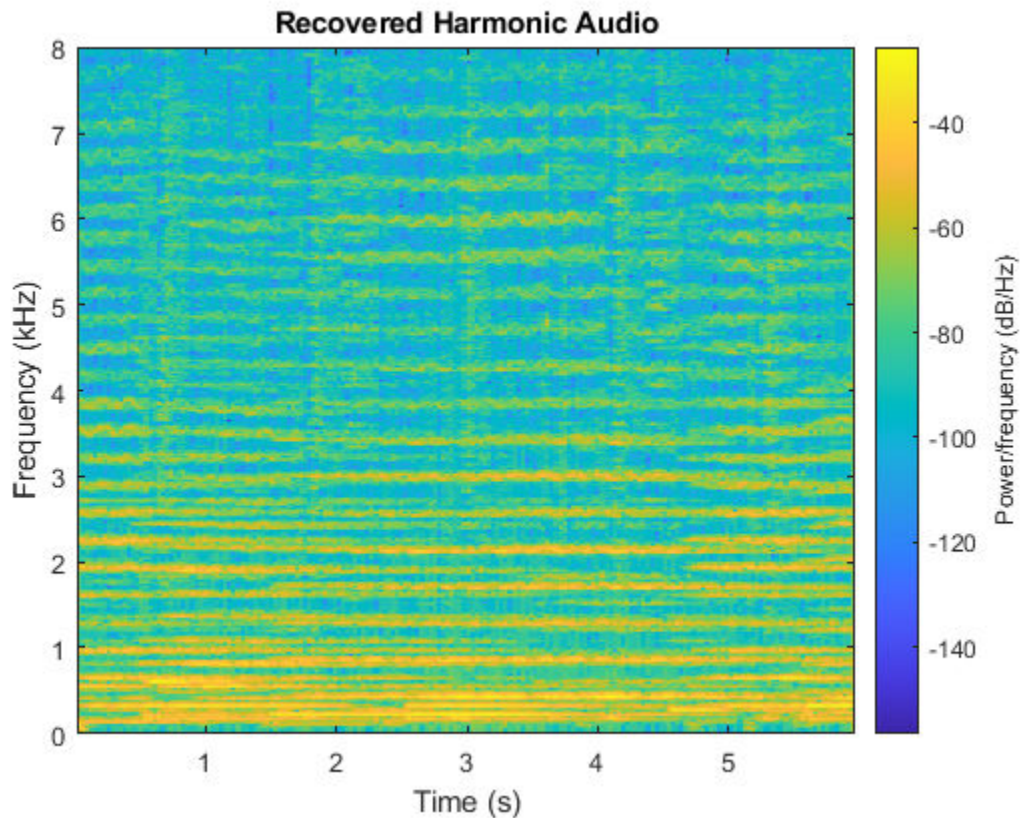
```
h = istfft(yharm, ...
    "Window",win, ...
    "OverlapLength",overlapLength, ...
    "FFTLength",fftLength, ...
    "ConjugateSymmetric",true);
```

```
p = istft(yperc, ...
    "Window",win, ...
    "OverlapLength",overlapLength, ...
    "FFTLength",fftLength, ...
    "ConjugateSymmetric",true);
```

Listen to the recovered harmonic audio and plot the spectrogram.

```
sound(h,fs)
```

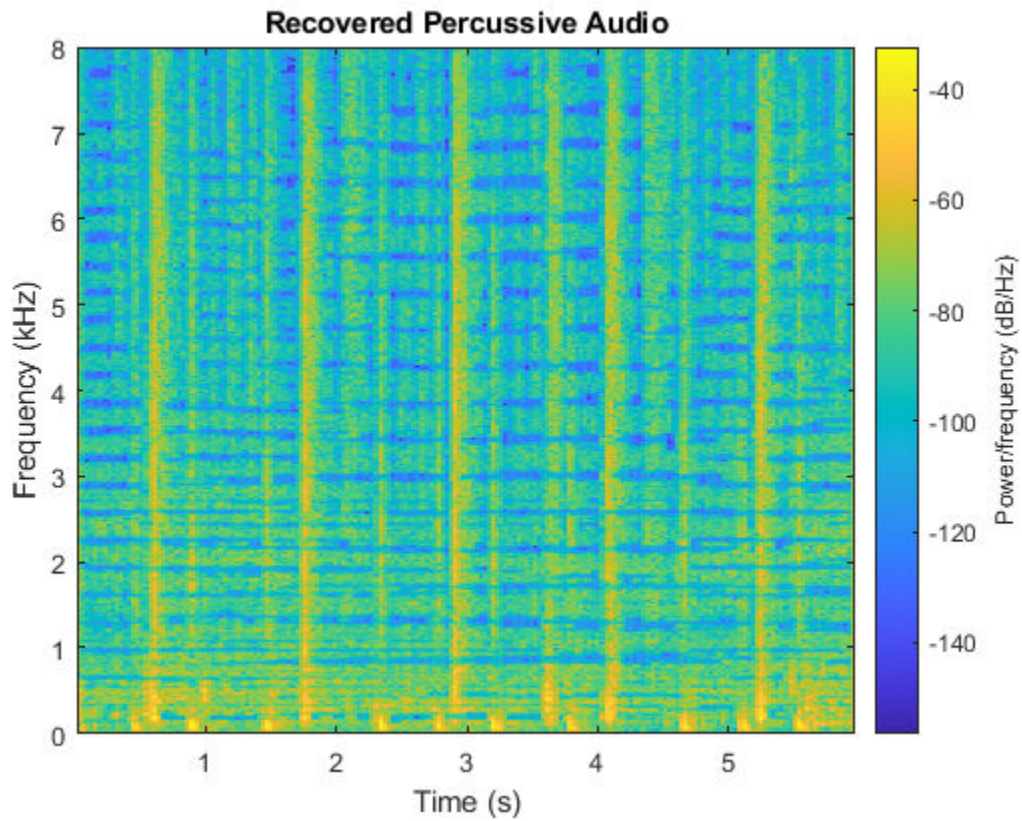
```
spectrogram(h,1024,512,1024,fs,"yaxis")
title("Recovered Harmonic Audio")
```



Listen to the recovered percussive audio and plot the spectrogram.

```
sound(p,fs)
```

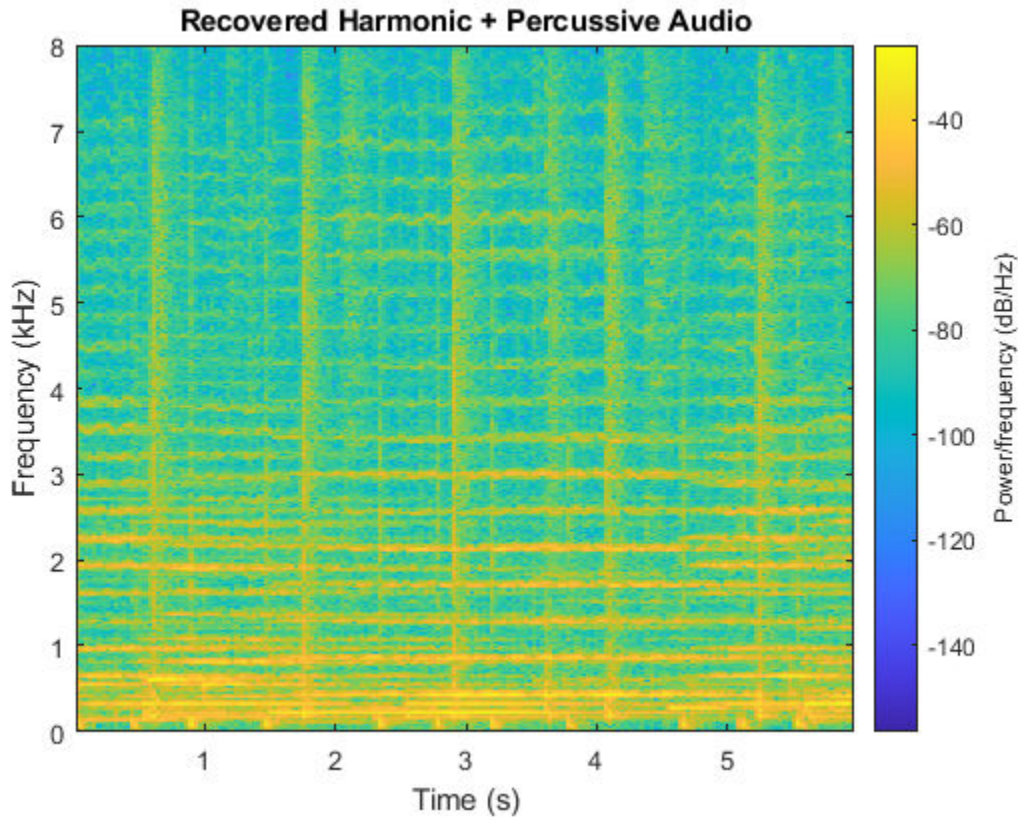
```
spectrogram(p,1024,512,1024,fs,"yaxis")
title("Recovered Percussive Audio")
```



Plot the combination of the recovered harmonic and percussive spectrograms.

```
sound(h + p,fs)

spectrogram(h + p,1024,512,1024,fs,"yaxis")
title("Recovered Harmonic + Percussive Audio")
```

HPSS Using Binary Mask and Residual

As suggested in [1 on page 1-0], decomposing a signal into harmonic and percussive sounds is often impossible. They propose adding a thresholding parameter: if the bin of the spectrogram is not clearly harmonic or percussive, categorize it as *residual*.

Perform the same steps described in HPSS Using Binary Mask on page 1-0 to create harmonic-enhanced and percussive-enhanced spectrograms.


```
win = sqrt(hann(1024,"periodic"));
overlapLength = floor(numel(win)/2);
fftLength = 2^nextpow2(numel(win) + 1);
y = stft(mix, ...
    "Window",win, ...
    "OverlapLength",overlapLength, ...
    "FFTLength",fftLength, ...
    "Centered",true);
halfIdx = 1:ceil(size(y,1)/2);
yhalf = y(halfIdx,:);
ymag = abs(yhalf);

timeFilterLength = 0.2;
timeFilterLengthInSamples = timeFilterLength/((numel(win) - overlapLength)/fs);
ymagharm = movmedian(ymag,timeFilterLengthInSamples,2);

frequencyFilterLength = 500;
frequencyFilterLengthInSamples = frequencyFilterLength/(fs/fftLength);
```

```
ymagperc = movmedian(ymag,frequencyFilterLengthInSamples,1);  
  
totalMagnitudePerBin = ymagharm + ymagperc;
```

Using a threshold, create three binary masks: harmonic, percussive, and residual. Set the threshold to 0.65. This means that if the magnitude of a bin of the harmonic-enhanced spectrogram is 65% of the total magnitude for that bin, you assign that bin to the harmonic portion. If the magnitude of a bin of the percussive-enhanced spectrogram is 65% of the total magnitude for that bin, you assign that bin to the percussive portion. Otherwise, the bin is assigned to the residual portion. The optimal thresholding parameter depends on the harmonic-percussive mix and your application.

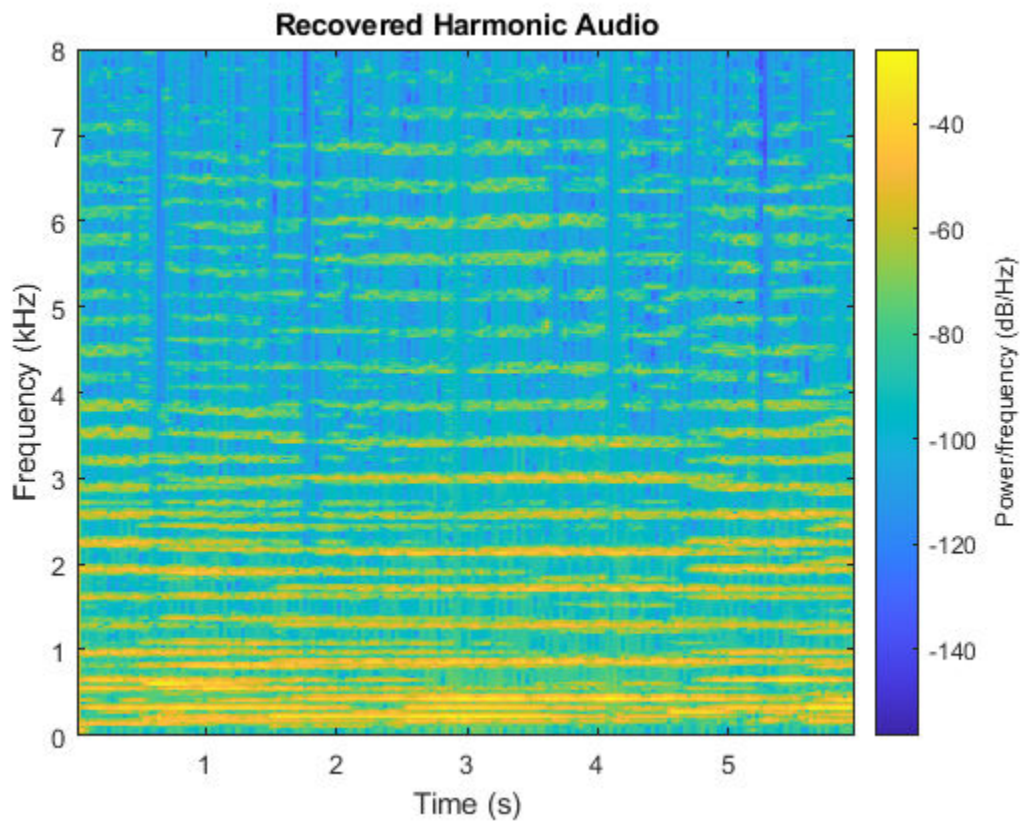
```
threshold = 0.65 ;  
harmonicMask = ymagharm > (totalMagnitudePerBin*threshold);  
percussiveMask = ymagperc > (totalMagnitudePerBin*threshold);  
residualMask = ~(harmonicMask+percussiveMask);
```

Perform the same steps described in HPSS Using Binary Mask on page 1-0 to return the masked signals to the time domain.

```
yharm = harmonicMask.*yhalf;  
yperc = percussiveMask.*yhalf;  
yresi = residualMask.*yhalf;  
  
yharm = cat(1,yharm,flipud(conj(yharm)));  
yperc = cat(1,yperc,flipud(conj(yperc)));  
yresi = cat(1,yresi,flipud(conj(yresi)));  
  
h = istft(yharm, ...  
    "Window",win, ...  
    "OverlapLength",overlapLength, ...  
    "FFTLength",fftLength, ...  
    "ConjugateSymmetric",true);  
p = istft(yperc, ...  
    "Window",win, ...  
    "OverlapLength",overlapLength, ...  
    "FFTLength",fftLength, ...  
    "ConjugateSymmetric",true);  
r = istft(yresi, ...  
    "Window",win, ...  
    "OverlapLength",overlapLength, ...  
    "FFTLength",fftLength, ...  
    "ConjugateSymmetric",true);
```

Listen to the recovered harmonic audio and plot the spectrogram.

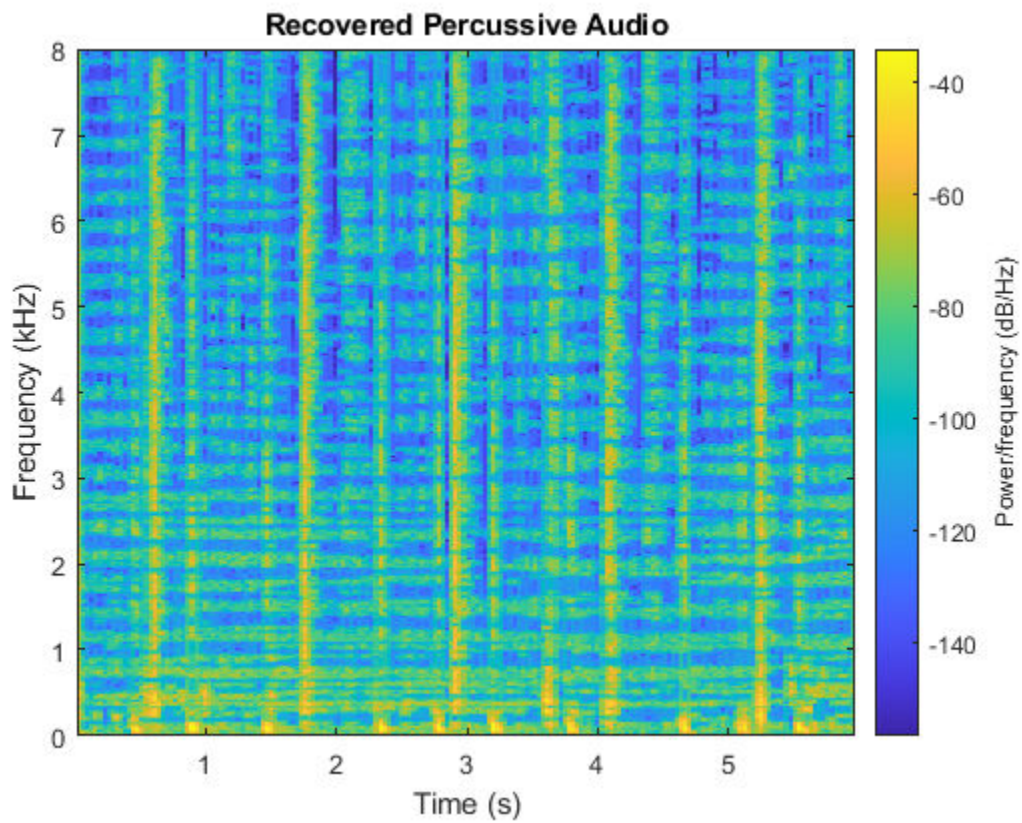
```
sound(h,fs)  
  
spectrogram(h,1024,512,1024,fs,"yaxis")  
title("Recovered Harmonic Audio")
```



Listen to the recovered percussive audio and plot the spectrogram.

```
sound(p, fs)
```

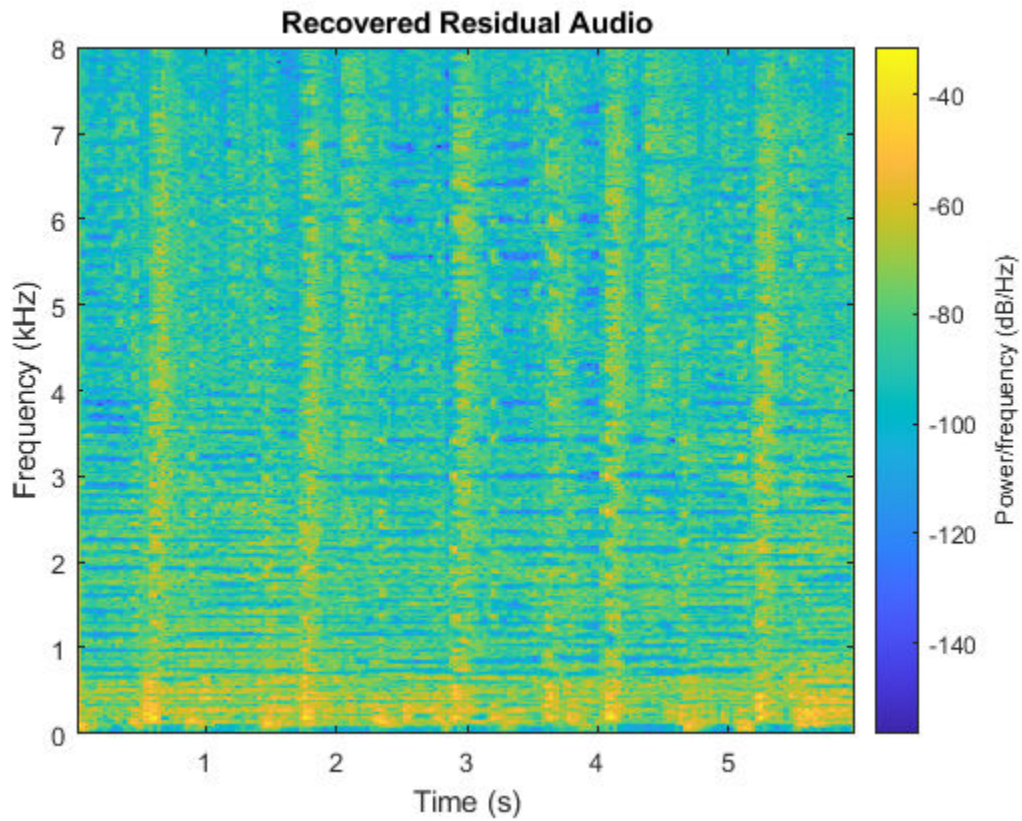
```
spectrogram(p, 1024, 512, 1024, fs, "yaxis")  
title("Recovered Percussive Audio")
```



Listen to the recovered residual audio and plot the spectrogram.

```
sound(r,fs)
```

```
spectrogram(r,1024,512,1024,fs,"yaxis")  
title("Recovered Residual Audio")
```

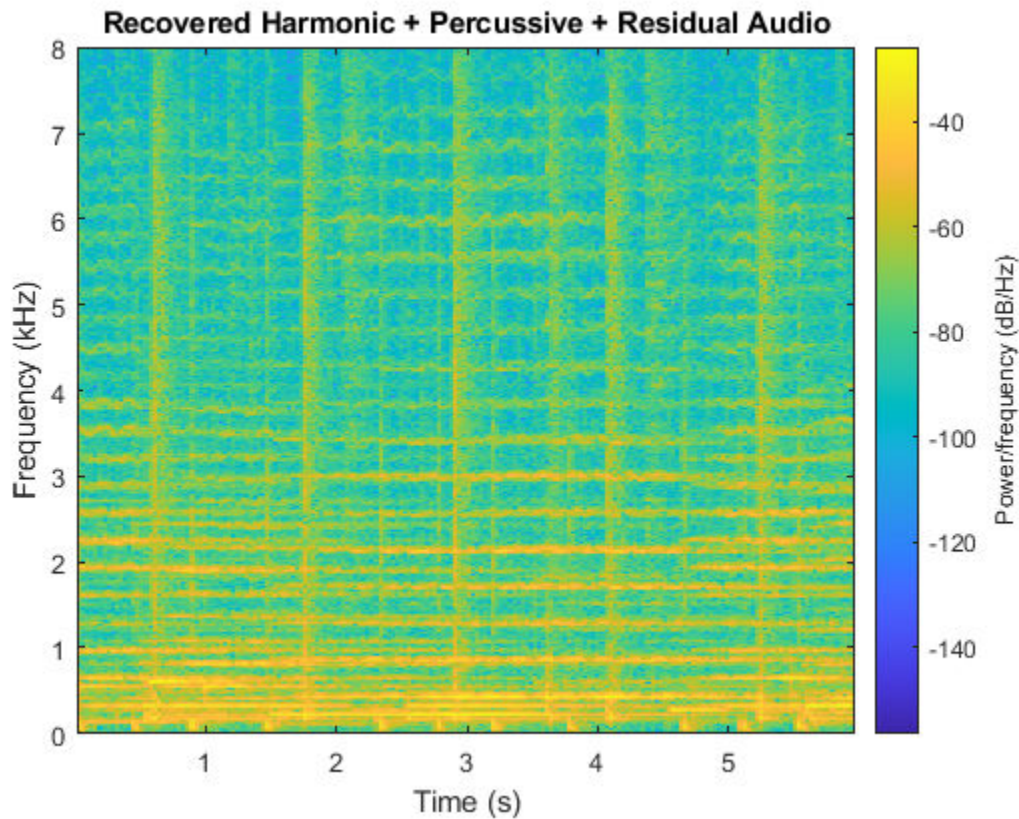



Listen to the combination of the harmonic, percussive, and residual signals and plot the spectrogram.

```
sound(h + p + r, fs)
```

```
spectrogram(h + p + r, 1024, 512, 1024, fs, "yaxis")
```

```
title("Recovered Harmonic + Percussive + Residual Audio")
```



HPSS Using Soft Mask

For time-frequency masking, masks are generally either binary or soft. Soft masking separates the energy of the mixed bins into harmonic and percussive portions depending on the relative weights of their enhanced spectrograms.

Perform the same steps described in HPSS Using Binary Mask on page 1-0 to create harmonic-enhanced and percussive-enhanced spectrograms.

```
win = sqrt(hann(1024,"periodic"));
overlapLength = floor(numel(win)/2);
fftLength = 2^nextpow2(numel(win) + 1);
y = stft(mix, ...
    "Window",win, ...
    "OverlapLength",overlapLength, ...
    "FFTLength",fftLength, ...
    "Centered",true);
halfIdx = 1:ceil(size(y,1)/2);
yhalf = y(halfIdx,:);
ymag = abs(yhalf);

timeFilterLength = 0.2;
timeFilterLengthInSamples = timeFilterLength/((numel(win)-overlapLength)/fs);
ymagharm = movmedian(ymag,timeFilterLengthInSamples,2);

frequencyFilterLength = 500;
frequencyFilterLengthInSamples = frequencyFilterLength/(fs/fftLength);
```

```
ymagperc = movmedian(ymag,frequencyFilterLengthInSamples,1);
```

```
totalMagnitudePerBin = ymagharm + ymagperc;
```

Create soft masks that separate the bin energy to the harmonic and percussive portions relative to the weights of their enhanced spectrograms.

```
harmonicMask = ymagharm ./ totalMagnitudePerBin;
percussiveMask = ymagperc ./ totalMagnitudePerBin;
```

Perform the same steps described in HPSS Using Binary Mask on page 1-0 to return the masked signals to the time domain.

```
yharm = harmonicMask.*yhalf;
yperc = percussiveMask.*yhalf;

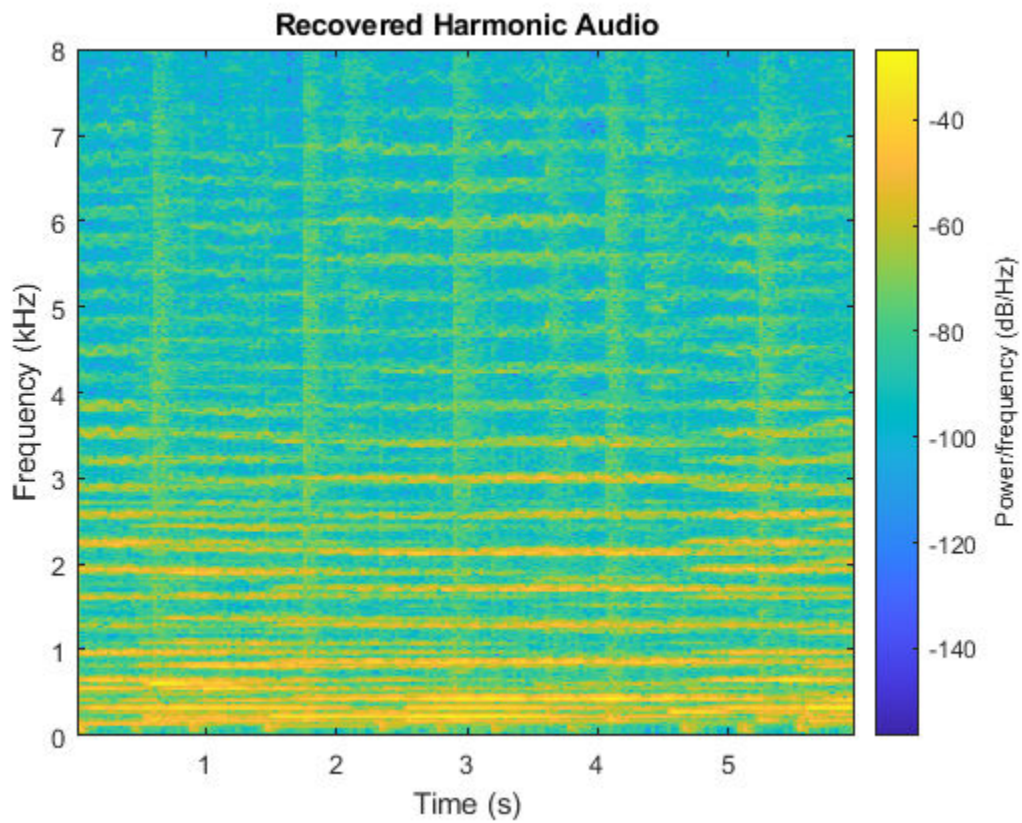
yharm = cat(1,yharm,flipud(conj(yharm)));
yperc = cat(1,yperc,flipud(conj(yperc)));
```

```
h = istft(yharm, ...
    "Window",win, ...
    "OverlapLength",overlapLength, ...
    "FFTLength",fftLength, ...
    "ConjugateSymmetric",true);
p = istft(yperc, ...
    "Window",win, ...
    "OverlapLength",overlapLength, ...
    "FFTLength",fftLength, ...
    "ConjugateSymmetric",true);
```

Listen to the recovered harmonic audio and plot the spectrogram.

```
sound(h,fs)

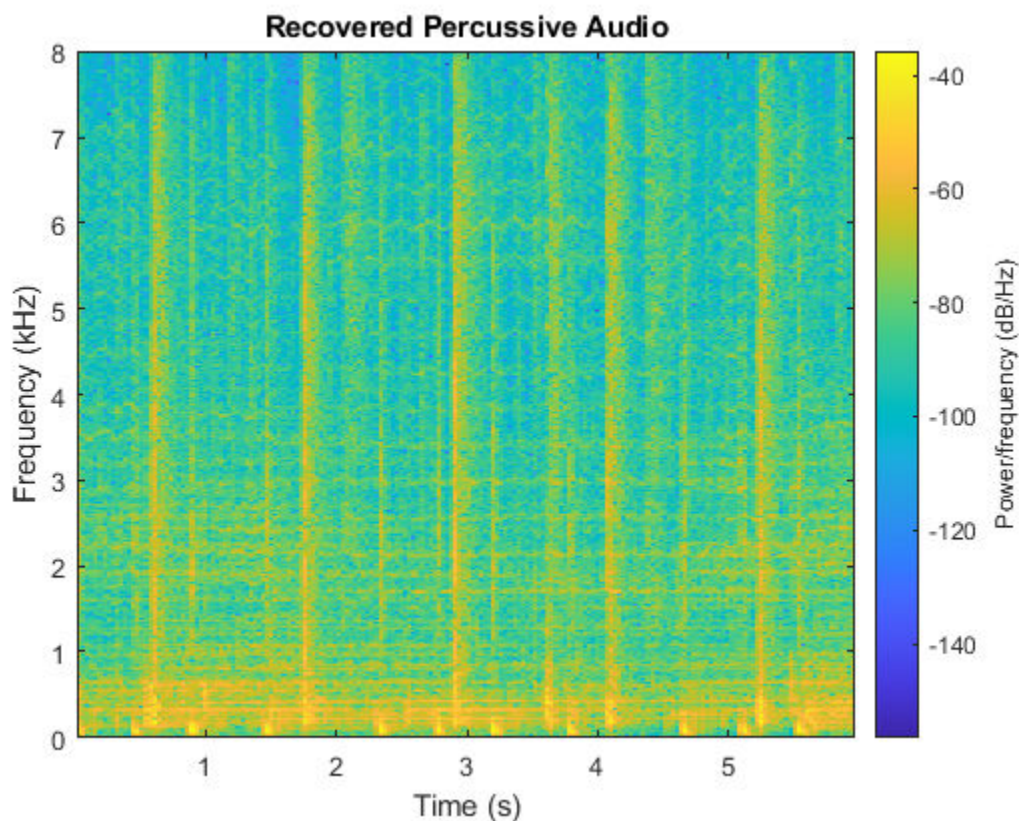
spectrogram(h,1024,512,1024,fs,"yaxis")
title("Recovered Harmonic Audio")
```



Listen to the recovered percussive audio and plot the spectrogram.

```
sound(p, fs)
```

```
spectrogram(p, 1024, 512, 1024, fs, "yaxis")  
title("Recovered Percussive Audio")
```

Example Function

The example function, `HelperHPSS`, provides the harmonic-percussive source separation capabilities described in this example. You can use it to quickly explore how parameters effect the algorithm performance.

help `HelperHPSS`

`[h,p] = HelperHPSS(x,fs)` separates the input signal, `x`, into harmonic (`h`) and percussive (`p`) portions. If `x` is input as a multichannel signal, it is converted to mono before processing.

`[h,p] = HelperHPSS(...,'TimeFilterLength',TIMEFILTERLENGTH)` specifies the median filter length along the time dimension of a spectrogram, in seconds. If unspecified, `TIMEFILTERLENGTH` defaults to 0.2 seconds.

`[h,p] = HelperHPSS(...,'FrequencyFilterLength',FREQUENCYFILTERLENGTH)` specifies the median filter length along the frequency dimension of a spectrogram, in Hz. If unspecified, `FREQUENCYFILTERLENGTH` defaults to 500 Hz.

`[h,p] = HelperHPSS(...,'MaskType',MASKTYPE)` specifies the mask type as 'binary' or 'soft'. If unspecified, `MASKTYPE` defaults to 'binary'.

`[h,p] = HelperHPSS(...,'Threshold',THRESHOLD)` specifies the threshold of the total energy for declaring an element as harmonic, percussive, or residual. Specify `THRESHOLD` as a scalar in the range [0 1]. This parameter is only valid if `MaskType` is set to 'binary'. If unspecified,

THRESHOLD defaults to 0.5.

`[h,p] = HelperHPSS(...,'Window',WINDOW)` specifies the analysis window used in the STFT. If unspecified, WINDOW defaults to `sqrt(hann(1024,'periodic'))`.

`[h,p] = HelperHPSS(...,'FFTLength',FFTLLENGTH)` specifies the number of points in the DFT for each analysis window. If unspecified, FFTLENGTH defaults to the number of elements in the WINDOW.

`[h,p] = HelperHPSS(...,'OverlapLength',OVERLAPLENGTH)` specifies the overlap length of the analysis windows. If unspecified, OVERLAPLENGTH defaults to 512.

`[h,p,r] = HelperHPSS(...)` returns the residual signal not classified as harmonic or percussive.

Example:



```
% Load a sound file and listen to it.
[audio,fs] = audioread('Laughter-16-8-mono-4secs.wav');
sound(audio,fs)



% Call HelperHPSS to separate the audio into harmonic and percussive
% portions. Listen to the portions separately.
[h,p] = HelperHPSS(audio,fs);
sound(h,fs)
sound(p,fs)
```

HPSS Using Iterative Masking

[1 on page 1-0] observed that a large frame size in the STFT calculation moves the energy towards the harmonic component, while a small frame size moves the energy towards the percussive component. [1 on page 1-0] proposed using an iterative procedure to take advantage of this insight. In the iterative procedure:

- 1 Perform HPSS using a large frame size to isolate the harmonic component.
- 2 Sum the residual and percussive portions.
- 3 Perform HPSS using a small frame size to isolate the percussive component.

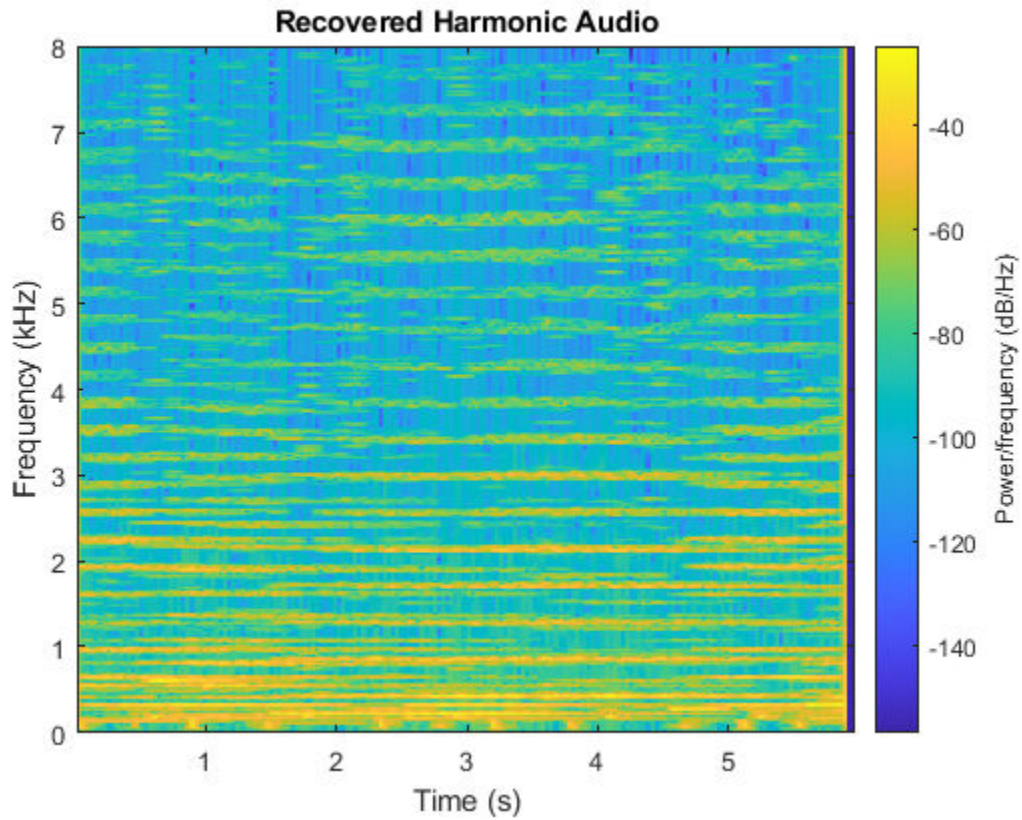
```
threshold1 = 0.7  ;
N1 = 4096  ;
[h1,p1,r1] = HelperHPSS(mix,fs,"Threshold",threshold1,"Window",sqrt(hann(N1,"periodic")),"OverlapLength",512);
mix1 = p1 + r1;

threshold2 = 0.6  ;
N2 = 256  ;
[h2,p2,r2] = HelperHPSS(mix1,fs,"Threshold",threshold2,"Window",sqrt(hann(N2,"periodic")),"OverlapLength",512);
h = h1;
p = p2;
r = h2 + r2;
```

Listen to the recovered percussive audio and plot the spectrogram.

```
sound(h, fs)
```

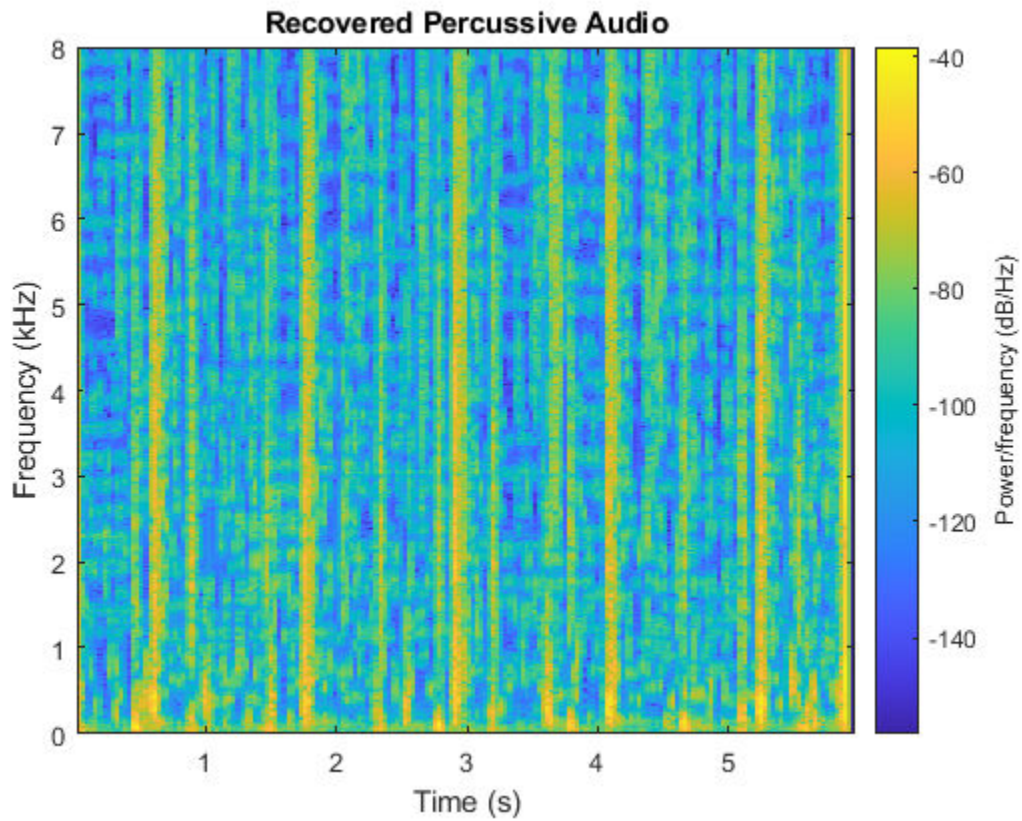
```
spectrogram(h, 1024, 512, 1024, fs, "yaxis")  
title("Recovered Harmonic Audio")
```



Listen to the recovered percussive audio and plot the spectrogram.

```
sound(p, fs)
```

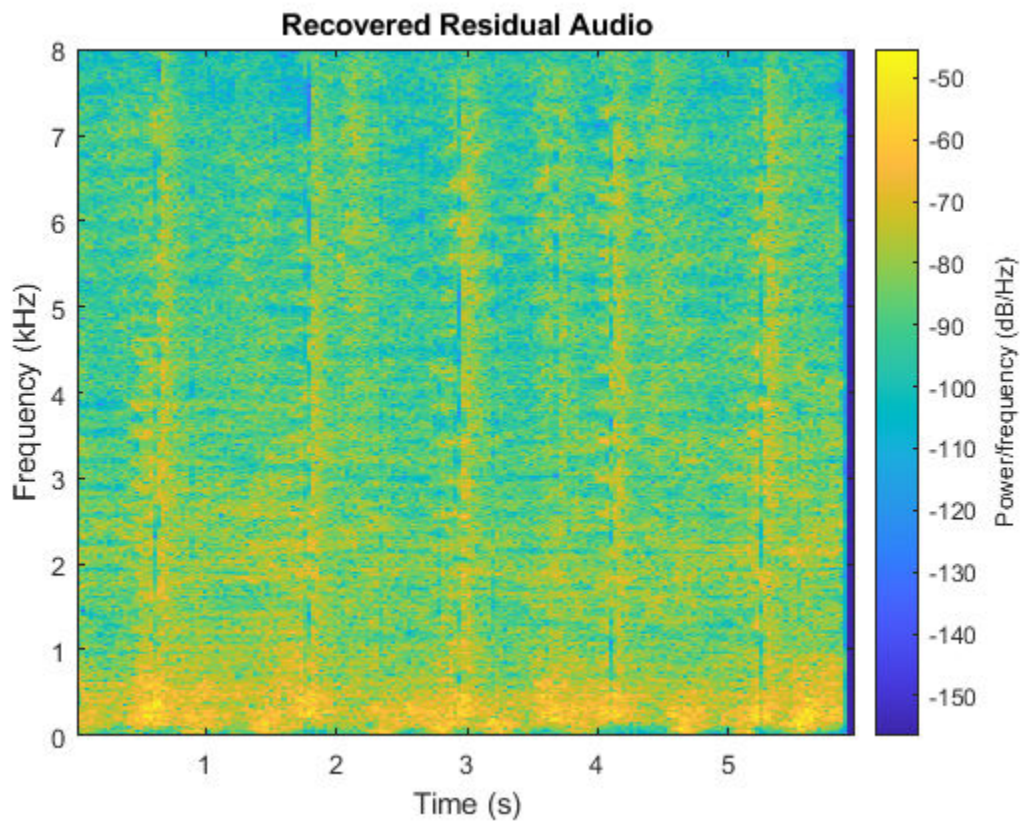
```
spectrogram(p, 1024, 512, 1024, fs, "yaxis")  
title("Recovered Percussive Audio")
```



Listen to the recovered residual audio and plot the spectrogram.

```
sound(r,fs)
```

```
spectrogram(r,1024,512,1024,fs,"yaxis")  
title("Recovered Residual Audio")
```

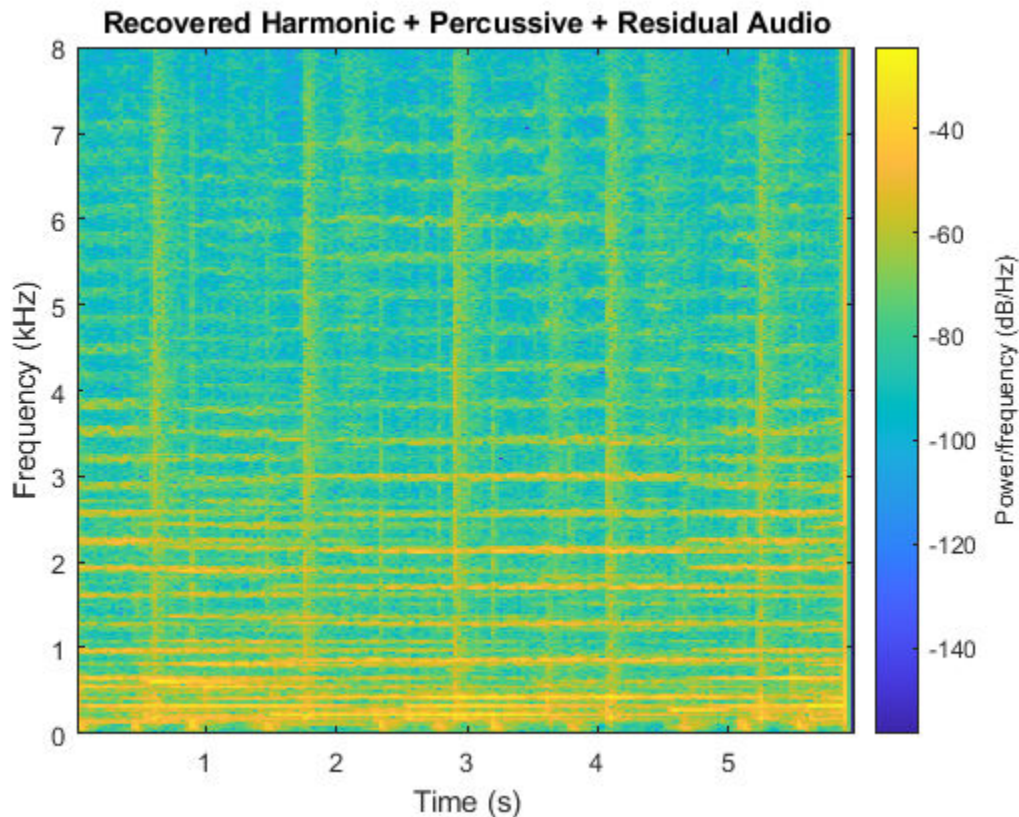



Listen to the combination of the harmonic, percussive, and residual signals and plot the spectrogram.

```
sound(h + p + r, fs)
```

```
spectrogram(h+p+r,1024,512,1024,fs,"yaxis")
```

```
title("Recovered Harmonic + Percussive + Residual Audio")
```



Enhanced Time Scale Modification Using HPSS

[2 on page 1-0] proposes that time scale modification (TSM) can be improved by first separating a signal into harmonic and percussive portions and then applying a TSM algorithm optimal for the portion. After TSM, the signal is reconstituted by summing the stretched audio.

To listen to a stretched audio without HPSS, apply time-scale modification using the default `stretchAudio` function. By default, `stretchAudio` uses the phase vocoder algorithm.

```
alpha = 1.5  ;
mixStretched = stretchAudio(mix,alpha);

sound(mixStretched,fs)
```

Separate the harmonic-percussive mix into harmonic and percussive portions using `HelperHPSS`. As proposed in [2 on page 1-0], use the default vocoder algorithm to stretch the harmonic portion and the WSOLA algorithm to stretch the percussive portion. Sum the stretched portions and listen to the results.

```
[h,p] = HelperHPSS(mix,fs);
hStretched = stretchAudio(h,alpha);
pStretched = stretchAudio(p,alpha,"Method","wsola");

mixStretched = hStretched + pStretched;
sound(mixStretched,fs);
```

References

- [1] Driedger, J., M. Muller, and S. Disch. "Extending harmonic-percussive separation of audio signals." *Proceedings of the International Society for Music Information Retrieval Conference*. Vol. 15, 2014.
- [2] Driedger, J., M. Muller, and S. Ewert. "Improving Time-Scale Modification of Music Signals Using Harmonic-Percussive Separation." *IEEE Signal Processing Letters*. Vol. 21. Issue 1. pp. 105-109, 2014.

Binaural Audio Rendering Using Head Tracking

Track head orientation by fusing data received from an IMU and then control the direction of arrival of a sound source by applying head-related transfer functions (HRTF).

In a typical virtual reality setup, the IMU sensor is attached to the user's headphones or VR headset so that the perceived position of a sound source is relative to a visual cue independent of head movements. For example, if the sound is perceived as coming from the monitor, it remains that way even if the user turns his head to the side.

Required Hardware

- Arduino Uno
- Invensense MPU-9250

Hardware Connection

First, connect the Invensense MPU-9250 to the Arduino board. For more details, see Estimating Orientation Using Inertial Sensor Fusion and MPU-9250.

Create Sensor Object and IMU Filter

Create an arduino object.

```
a = arduino;
```

Create the Invensense MPU-9250 sensor object.

```
imu = mpu9250(a);
```

Create and set the sample rate of the Kalman filter.

```
Fs = imu.SampleRate;  
imufilt = imufilter('SampleRate',Fs);
```

Load the ARI HRTF Dataset

When sound travels from a point in space to your ears, you can localize it based on interaural time and level differences (ITD and ILD). These frequency-dependent ITD and ILD's can be measured and represented as a pair of impulse responses for any given source elevation and azimuth. The ARI HRTF Dataset contains 1550 pairs of impulse responses which span azimuths over 360 degrees and elevations from -30 to 80 degrees. You use these impulse responses to filter a sound source so that it is perceived as coming from a position determined by the sensor's orientation. If the sensor is attached to a device on a user's head, the sound is perceived as coming from one fixed place despite head movements.

First, load the HRTF dataset.

```
ARIDataset = load('ReferenceHRTF.mat');
```

Then, get the relevant HRTF data from the dataset and put it in a useful format for our processing.

```
hrtfData = double(ARIDataset.hrtfData);  
hrtfData = permute(hrtfData,[2,3,1]);
```

Get the associated source positions. Angles should be in the same range as the sensor. Convert the azimuths from [0,360] to [-180,180].

```
sourcePosition = ARIDataset.sourcePosition(:,[1,2]);
sourcePosition(:,1) = sourcePosition(:,1) - 180;
```

Load Monaural Recording

Load an ambisonic recording of a helicopter. Keep only the first channel, which corresponds to an omnidirectional recording. Resample it to 48 kHz for compatibility with the HRTF data set.

```
[heli,originalSampleRate] = audioread('Heli_16ch_ACN_SN3D.wav');
heli = 12*heli(:,1); % keep only one channel
```

```
sampleRate = 48e3;
heli = resample(heli,sampleRate,originalSampleRate);
```

Load the audio data into a `SignalSource` object. Set the `SamplesPerFrame` to 0.1 seconds.

```
sigsrsc = dsp.SignalSource(heli, ...
    'SamplesPerFrame',sampleRate/10, ...
    'SignalEndAction','Cyclic repetition');
```

Set Up the Audio Device

Create an `audioDeviceWriter` with the same sample rate as the audio signal.

```
deviceWriter = audioDeviceWriter('SampleRate',sampleRate);
```

Create FIR Filters for the HRTF coefficients

Create a pair of FIR filters to perform binaural HRTF filtering.

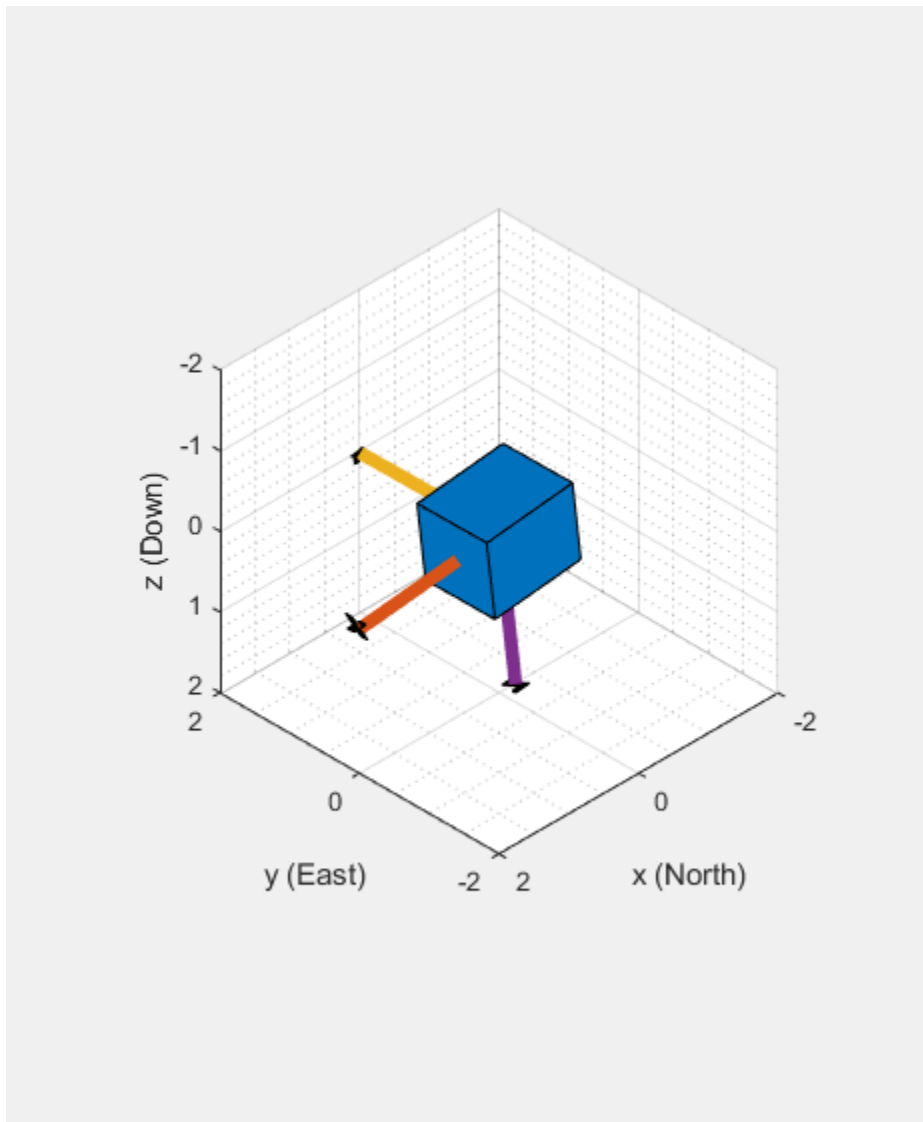
```
FIR = cell(1,2);
FIR{1} = dsp.FIRFilter('NumeratorSource','Input port');
FIR{2} = dsp.FIRFilter('NumeratorSource','Input port');
```

Initialize the Orientation Viewer

Create an object to perform real-time visualization for the orientation of the IMU sensor. Call the IMU filter once and display the initial orientation.

```
orientationScope = HelperOrientationViewer;
data = read(imu);

qimu = imufilt(data.Acceleration,data.AngularVelocity);
orientationScope(qimu);
```



Audio Processing Loop

Execute the processing loop for 30 seconds. This loop performs the following steps:

- 1 Read data from the IMU sensor.
- 2 Fuse IMU sensor data to estimate the orientation of the sensor. Visualize the current orientation.
- 3 Convert the orientation from a quaternion representation to pitch and yaw in Euler angles.
- 4 Use `interpolateHRTF` to obtain a pair of HRTFs at the desired position.
- 5 Read a frame of audio from the signal source.
- 6 Apply the HRTFs to the mono recording and play the stereo signal. This is best experienced using headphones.

```
imuOverruns = 0;  
audioUnderruns = 0;  
audioFiltered = zeros(sigsrc.SamplesPerFrame,2);
```

```

tic
while toc < 30

    % Read from the IMU sensor.
    [data,overrun] = read(imu);
    if overrun > 0
        imuOverruns = imuOverruns + overrun;
    end

    % Fuse IMU sensor data to estimate the orientation of the sensor.
    qimu = imufilt(data.Acceleration,data.AngularVelocity);
    orientationScope(qimu);

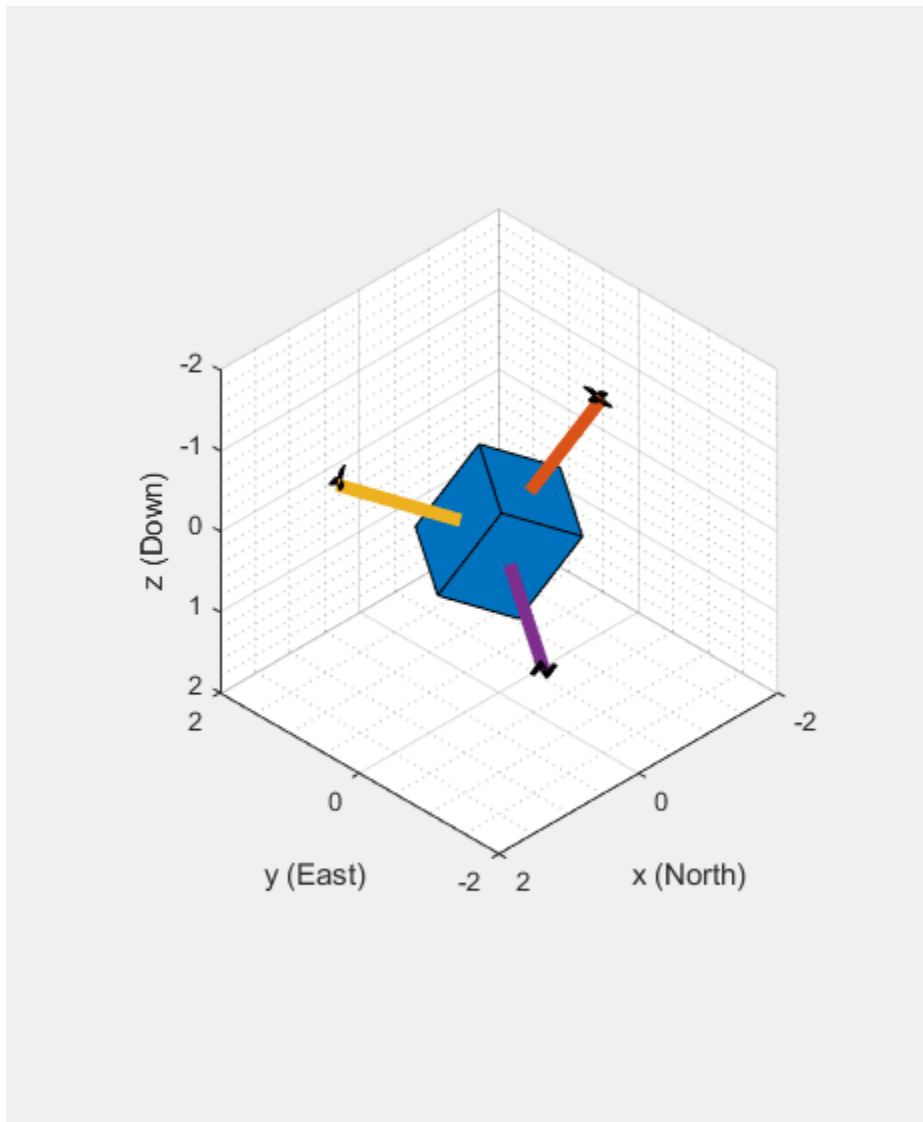
    % Convert the orientation from a quaternion representation to pitch and yaw in Euler angles.
    ypr = eulerd(qimu,'zyx','frame');
    yaw = ypr(end,1);
    pitch = ypr(end,2);
    desiredPosition = [yaw,pitch];

    % Obtain a pair of HRTFs at the desired position.
    interpolatedIR = squeeze(interpolateHRTF(hrtfData,sourcePosition,desiredPosition));

    % Read audio from file
    audioIn = sigsrc();

    % Apply HRTFs
    audioFiltered(:,1) = FIR{1}(audioIn, interpolatedIR(1,:)); % Left
    audioFiltered(:,2) = FIR{2}(audioIn, interpolatedIR(2,:)); % Right
    audioUnderruns = audioUnderruns + deviceWriter(squeeze(audioFiltered));
end

```

**Cleanup**

Release resources, including the sound device.

```
release(sigsrc)  
release(deviceWriter)  
clear imu a
```


Speech Emotion Recognition

This example illustrates a simple speech emotion recognition (SER) system using a BiLSTM network. You begin by downloading the data set and then testing the trained network on individual files. The network was trained on a small German-language database [1] on page 1-0 .

The example walks you through training the network, which includes downloading, augmenting, and training the dataset. Finally, you perform leave-one-speaker-out (LOSO) 10-fold cross validation to evaluate the network architecture.

The features used in this example were chosen using sequential feature selection, similar to the method described in “Sequential Feature Selection for Audio Features” on page 1-564.

Download Data Set

Download the Berlin Database of Emotional Speech [1] on page 1-0 . The database contains 535 utterances spoken by 10 actors intended to convey one of the following emotions: anger, boredom, disgust, anxiety/fear, happiness, sadness, or neutral. The emotions are text independent.

```
url = "http://emodb.bilderbar.info/download/download.zip";
downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder, "Emo-DB");

if ~exist(datasetFolder, 'dir')
    disp('Downloading Emo-DB (40.5 MB) ...')
    unzip(url, datasetFolder)
end
```

Create an `audioDatastore` that points to the audio files.

```
ads = audioDatastore(fullfile(datasetFolder, "wav"));
```

The file names are codes indicating the speaker ID, text spoken, emotion, and version. The website contains a key for interpreting the code and additional information about the speakers such as gender and age. Create a table with the variables `Speaker` and `Emotion`. Decode the file names into the table.

```
filepaths = ads.Files;
emotionCodes = cellfun(@(x)x(end-5), filepaths, 'UniformOutput', false);
emotions = replace(emotionCodes, {'W', 'L', 'E', 'A', 'F', 'T', 'N'}, ...
    {'Anger', 'Boredom', 'Disgust', 'Anxiety/Fear', 'Happiness', 'Sadness', 'Neutral'});

speakerCodes = cellfun(@(x)x(end-10:end-9), filepaths, 'UniformOutput', false);
labelTable = cell2table([speakerCodes, emotions], 'VariableNames', {'Speaker', 'Emotion'});
labelTable.Emotion = categorical(labelTable.Emotion);
labelTable.Speaker = categorical(labelTable.Speaker);
summary(labelTable)
```

Variables:

Speaker: 535×1 categorical

Values:

03	49
08	58
09	43

10	38
11	55
12	35
13	61
14	69
15	56
16	71

Emotion: 535×1 categorical

Values:

Anger	127
Anxiety/Fear	69
Boredom	81
Disgust	46
Happiness	71
Neutral	79
Sadness	62

`labelTable` is in the same order as the files in `audioDatastore`. Set the `Labels` property of the `audioDatastore` to the `labelTable`.

```
ads.Labels = labelTable;
```

Perform Speech Emotion Recognition

Load the pretrained network, the `audioFeatureExtractor` object used to train the network, and normalization factors for the features. This network was trained using all speakers in the data set except speaker 03.

```
load('network_Audio_SER.mat','net','afe','normalizers');
```

The sample rate set on the `audioFeatureExtractor` corresponds to the sample rate of the data set.

```
fs = afe.SampleRate;
```

Select a speaker and emotion, then subset the datastore to only include the chosen speaker and emotion. Read from the datastore and listen to the file.

```
speaker =  ;  
emotion =  ;
```


```
adsSubset = subset(ads,ads.Labels.Speaker==speaker & ads.Labels.Emotion == emotion);
```

```
audio = read(adsSubset);  
sound(audio,fs)
```

Use the `audioFeatureExtractor` object to extract the features and then transpose them so that time is along rows. Normalize the features and then convert them to 20-element sequences with 10-element overlap, which corresponds to approximately 600 ms windows with 300 ms overlap. Use the supporting function, `HelperFeatureVector2Sequence` on page 1-0 , to convert the array of feature vectors to sequences.

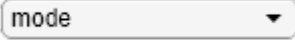
```
features = (extract(afe,audio))';
```

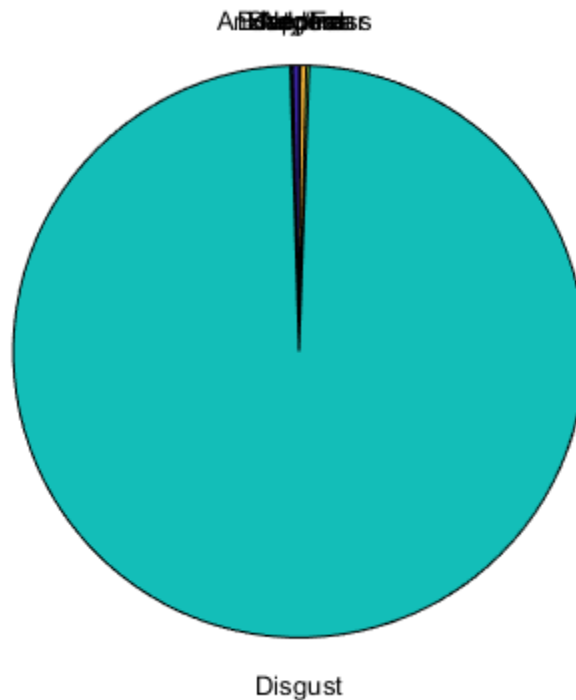
```
featuresNormalized = (features - normalizers.Mean)./normalizers.StandardDeviation;
```

```
numOverlap = 10  ;  
featureSequences = HelperFeatureVector2Sequence(featuresNormalized,20,numOverlap);
```

Feed the feature sequences into the network for prediction. Compute the mean prediction and plot the probability distribution of the chosen emotions as a pie chart. You can try different speakers, emotions, sequence overlap, and prediction average to test the network's performance. To get a realistic approximation of the network's performance, use speaker 03, which the network was not trained on.

```
YPred = double(predict(net,featureSequences));
```

```
average =  ;  
switch average  
    case 'mean'  
        probs = mean(YPred,1);  
    case 'median'  
        probs = median(YPred,1);  
    case 'mode'  
        probs = mode(YPred,1);  
end  
pie(probs./sum(probs),string(net.Layers(end).Classes))
```




The remainder of the example illustrates how the network was trained and validated.

Train Network

The 10-fold cross validation accuracy of a first attempt at training was about 60% because of insufficient training data. A model trained on the insufficient data overfits some folds and underfits others. To improve overall fit, increase the size of the dataset using `audioDataAugmenter`. 50 augmentations per file was chosen empirically as a good tradeoff between processing time and accuracy improvement. You can decrease the number of augmentations to speed up the example.

Create an `audioDataAugmenter` object. Set the probability of applying pitch shifting to 0.5 and use the default range. Set the probability of applying time shifting to 1 and use a range of $[-0.3, 0.3]$ seconds. Set the probability of adding noise to 1 and specify the SNR range as $[-20, 40]$ dB.

```
numAugmentations = 50  ;
augmenter = audioDataAugmenter('NumAugmentations',numAugmentations, ...
    'TimeStretchProbability',0, ...
    'VolumeControlProbability',0, ...
    ...
    'PitchShiftProbability',0.5, ...
    ...
    'TimeShiftProbability',1, ...
    'TimeShiftRange',[-0.3,0.3], ...
    ...
    'AddNoiseProbability',1, ...
    'SNRRange', [-20,40]);
```

Create a new folder in your current folder to hold the augmented data set.

```
currentDir = pwd;
writeDirectory = fullfile(currentDir,'augmentedData');
mkdir(writeDirectory)
```

For each file in the audio datastore:

- 1 Create 50 augmentations.
- 2 Normalize the audio to have a max absolute value of 1.
- 3 Write the augmented audio data as a WAV file. Append `_augK` to each of the file names, where *K* is the augmentation number. To speed up processing, use `parfor` and partition the datastore.

This method of augmenting the database is time consuming (approximately 1 hour) and space consuming (approximately 26 GB). However, when iterating on choosing a network architecture or feature extraction pipeline, this upfront cost is generally advantageous.

```
N = numel(ads.Files)*numAugmentations;
myWaitBar = HelperPoolWaitbar(N,"Augmenting Dataset...");

reset(ads)

numPartitions = 18;

tic
parfor ii = 1:numPartitions
    adsPart = partition(ads,numPartitions,ii);
    while hasdata(adsPart)
        [x,adsInfo] = read(adsPart);
        data = augment(augmenter,x,fs);
```

```

[~,fn] = fileparts(adsInfo.FileName);
for i = 1:size(data,1)
    augmentedAudio = data.Audio{i};
    augmentedAudio = augmentedAudio/max(abs(augmentedAudio),[],'all');
    augNum = num2str(i);
    if numel(augNum)==1
        iString = ['0',augNum];
    else
        iString = augNum;
    end
    audiowrite(fullfile(writeDirectory,sprintf('%s_aug%s.wav',fn,iString)),augmentedAudio);
    increment(myWaitBar)
end
end
end

```

Starting parallel pool (parpool) using the 'local' profile ...
 Connected to the parallel pool (number of workers: 6).

```

delete(myWaitBar)
fprintf('Augmentation complete (%0.2f minutes).\n',toc/60)

```

Augmentation complete (62.23 minutes).

Create an audio datastore that points to the augmented data set. Replicate the rows of the label table of the original datastore NumAugmentations times to determine the labels of the augmented datastore.

```

adsAug = audioDatastore(writeDirectory);
adsAug.Labels = repelem(ads.Labels,augmenter.NumAugmentations,1);

```

Create an audioFeatureExtractor object. Set Window to a periodic 30 ms Hamming window, OverlapLength to 0, and SampleRate to the sample rate of the database. Set gtcc, gtccDelta, mfccDelta, and spectralCrest to true to extract them. Set SpectralDescriptorInput to melSpectrum so that the spectralCrest is calculated for the mel spectrum.

```

win = hamming(round(0.03*fs),"periodic");
overlapLength = 0;

afe = audioFeatureExtractor( ...
    'Window',win, ...
    'OverlapLength',overlapLength, ...
    'SampleRate',fs, ...
    ...
    'gtcc',true, ...
    'gtccDelta',true, ...
    'mfccDelta',true, ...
    ...
    'SpectralDescriptorInput','melSpectrum', ...
    'spectralCrest',true);

```

Train for Deployment

When you train for deployment, use all available speakers in the data set. Set the training datastore to the augmented datastore.

```

adsTrain = adsAug;

```

Convert the training audio datastore to a tall array. If you have Parallel Computing Toolbox™, the extraction is automatically parallelized. If you do not have Parallel Computing Toolbox™, the code continues to run.

```
tallTrain = tall(adsTrain);
```

Extract the training features and reorient the features so that time is along rows to be compatible with `sequenceInputLayer` (Deep Learning Toolbox).

```
featuresTallTrain = cellfun(@(x)extract(afe,x),tallTrain,"UniformOutput",false);
featuresTallTrain = cellfun(@(x)x',featuresTallTrain,"UniformOutput",false);
featuresTrain = gather(featuresTallTrain);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 2 min 13 sec
Evaluation completed in 2 min 13 sec
```

Use the training set to determine the mean and standard deviation of each feature.

```
allFeatures = cat(2,featuresTrain{:});
M = mean(allFeatures,2,'omitnan');
S = std(allFeatures,0,2,'omitnan');
```

```
featuresTrain = cellfun(@(x)(x-M)./S,featuresTrain,'UniformOutput',false);
```

Buffer the feature vectors into sequences so that each sequence consists of 20 feature vectors with overlaps of 10 feature vectors.

```
featureVectorsPerSequence = 20;
featureVectorOverlap = 10;
[sequencesTrain,sequencePerFileTrain] = HelperFeatureVector2Sequence(featuresTrain,featureVectorsPerSequence,featureVectorOverlap);
```

Replicate the labels of the training and validation sets so that they are in one-to-one correspondence with the sequences. Not all speakers have utterances for all emotions. Create an empty categorical array that contains all the emotional categories and append it to the validation labels so that the categorical array contains all emotions.

```
labelsTrain = repelem(adsTrain.Labels.Emotion,[sequencePerFileTrain{:}]);

emptyEmotions = ads.Labels.Emotion;
emptyEmotions(:) = [];
```

Define a BiLSTM network using `bilstmLayer` (Deep Learning Toolbox). Place a `dropoutLayer` (Deep Learning Toolbox) before and after the `bilstmLayer` to help prevent overfitting.

```
dropoutProb1 = 0.3;
numUnits = 200;
dropoutProb2 = 0.6;
layers = [ ...
    sequenceInputLayer(size(sequencesTrain{1},1))
    dropoutLayer(dropoutProb1)
    bilstmLayer(numUnits,"OutputMode","last")
    dropoutLayer(dropoutProb2)
    fullyConnectedLayer(numel(categories(emptyEmotions)))
    softmaxLayer
    classificationLayer];
```

Define training options using `trainingOptions` (Deep Learning Toolbox).

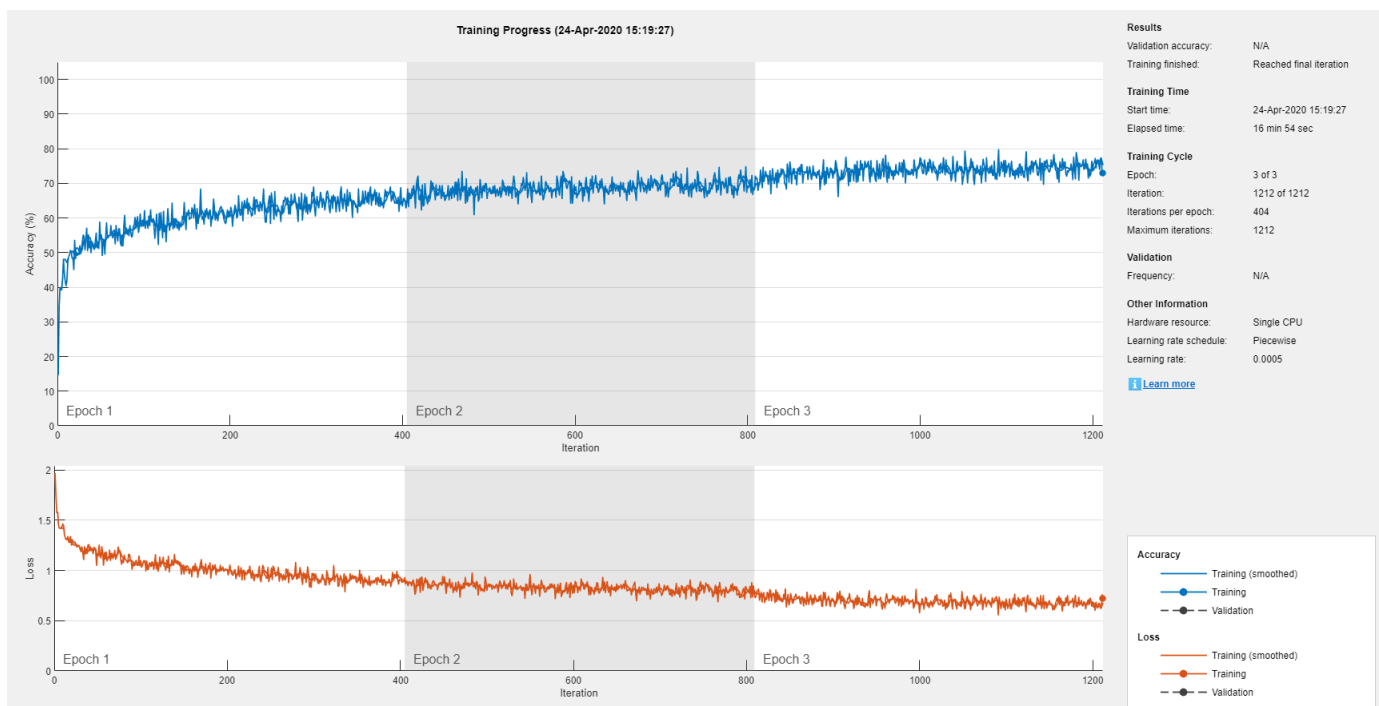
```

miniBatchSize = 512;
initialLearnRate = 0.005;
learnRateDropPeriod = 2;
maxEpochs = 3;
options = trainingOptions("adam", ...
    "MiniBatchSize",miniBatchSize, ...
    "InitialLearnRate",initialLearnRate, ...
    "LearnRateDropPeriod",learnRateDropPeriod, ...
    "LearnRateSchedule","piecewise", ...
    "MaxEpochs",maxEpochs, ...
    "Shuffle","every-epoch", ...
    "Verbose",false, ...
    "Plots","Training-Progress");

```

Train the network using `trainNetwork` (Deep Learning Toolbox).

```
net = trainNetwork(sequencesTrain,labelsTrain,layers,options);
```



To save the network, configured `audioFeatureExtractor`, and normalization factors, set `saveSERSystem` to true.

```

saveSERSystem = ☐ false ;
if saveSERSystem
    normalizers.Mean = M;
    normalizers.StandardDeviation = S;
    save('network_Audio_SER.mat','net','afe','normalizers')
end

```

Training for System Validation

To provide an accurate assessment of the model you created in this example, train and validate using leave-one-speaker-out (LOSO) k -fold cross validation. In this method, you train using $k - 1$ speakers

and then validate on the left-out speaker. You repeat this procedure k times. The final validation accuracy is the average of the k folds.

Create a variable that contains the speaker IDs. Determine the number of folds: 1 for each speaker. The database contains utterances from 10 unique speakers. Use `summary` to display the speaker IDs (left column) and the number of utterances they contribute to the database (right column).

```
speaker = ads.Labels.Speaker;
numFolds = numel(speaker);
summary(speaker)
```

```
03      49
08      58
09      43
10      38
11      55
12      35
13      61
14      69
15      56
16      71
```

The helper function `HelperTrainAndValidateNetwork` on page 1-0 performs the steps outlined above for all 10 folds and returns the true and predicted labels for each fold. Call `HelperTrainAndValidateNetwork` with the `audioDatastore`, the augmented `audioDatastore`, and the `audioFeatureExtractor`.

```
[labelsTrue,labelsPred] = HelperTrainAndValidateNetwork(ads,adsAug,afe);
```

Print the accuracy per fold and plot the 10-fold confusion chart.

```
for ii = 1:numel(labelsTrue)
    foldAcc = mean(labelsTrue{ii}==labelsPred{ii})*100;
    fprintf('Fold %1.0f, Accuracy = %0.1f\n',ii,foldAcc);
end
```

```
Fold 1, Accuracy = 83.7
Fold 2, Accuracy = 86.2
Fold 3, Accuracy = 83.7
Fold 4, Accuracy = 86.8
Fold 5, Accuracy = 74.5
Fold 6, Accuracy = 71.4
Fold 7, Accuracy = 60.7
Fold 8, Accuracy = 91.3
Fold 9, Accuracy = 78.6
Fold 10, Accuracy = 66.2
```

```
labelsTrueMat = cat(1,labelsTrue{:});
labelsPredMat = cat(1,labelsPred{:});
figure
cm = confusionchart(labelsTrueMat,labelsPredMat);
valAccuracy = mean(labelsTrueMat==labelsPredMat)*100;
cm.Title = sprintf('Confusion Matrix for 10-Fold Cross-Validation\nAverage Accuracy = %0.1f',valAccuracy);
sortClasses(cm,categories(emptyEmotions))
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
```


Confusion Matrix for 10-Fold Cross-Validation
Average Accuracy = 77.9

True Class	Anger	115			1	11			90.6%	9.4%
	Anxiety/Fear	5	45	1		8	9	1	65.2%	34.8%
	Boredom	2		68	1		2	8	84.0%	16.0%
	Disgust	1	2	4	34	2	2	1	73.9%	26.1%
	Happiness	18	6	1	1	45			63.4%	36.6%
	Neutral	3	2	9	3	1	57	4	72.2%	27.8%
	Sadness		2	6			1	53	85.5%	14.5%
		79.9%	78.9%	76.4%	85.0%	67.2%	80.3%	79.1%		
		20.1%	21.1%	23.6%	15.0%	32.8%	19.7%	20.9%		
		Anger	Anxiety/Fear	Boredom	Disgust	Happiness	Neutral	Sadness		
		Predicted Class								

Supporting Functions

Convert Array of Feature Vectors to Sequences

```
function [sequences,sequencePerFile] = HelperFeatureVector2Sequence(features,featureVectorsPerSequence)
% Copyright 2019 MathWorks, Inc.
if featureVectorsPerSequence <= featureVectorOverlap
    error('The number of overlapping feature vectors must be less than the number of feature vectors per sequence')
end

if ~iscell(features)
    features = {features};
end
hopLength = featureVectorsPerSequence - featureVectorOverlap;
idx1 = 1;
sequences = {};
sequencePerFile = cell(numel(features),1);
for ii = 1:numel(features)
    sequencePerFile{ii} = floor((size(features{ii},2) - featureVectorsPerSequence)/hopLength) + 1;
    idx2 = 1;
    for j = 1:sequencePerFile{ii}
        sequences{idx1,1} = features{ii}(:,idx2:idx2 + featureVectorsPerSequence - 1); %#ok<AS>
        idx1 = idx1 + 1;
        idx2 = idx2 + hopLength;
    end
end
end
```

Train and Validate Network

```
function [trueLabelsCrossFold,predictedLabelsCrossFold] = HelperTrainAndValidateNetwork(varargin)
% Copyright 2019 The MathWorks, Inc.
if nargin == 3
    ads = varargin{1};
    augads = varargin{2};
    extractor = varargin{3};
elseif nargin == 2
    ads = varargin{1};
    augads = varargin{1};
    extractor = varargin{2};
end
speaker = categories(ads.Labels.Speaker);
numFolds = numel(speaker);
emptyEmotions = (ads.Labels.Emotion);
emptyEmotions(:) = [];

% Loop over each fold.
trueLabelsCrossFold = {};
predictedLabelsCrossFold = {};

for i = 1:numFolds

    % 1. Divide the audio datastore into training and validation sets.
    % Convert the data to tall arrays.
    idxTrain = augads.Labels.Speaker~=speaker(i);
    augadsTrain = subset(augads,idxTrain);
    augadsTrain.Labels = augadsTrain.Labels.Emotion;
    tallTrain = tall(augadsTrain);
    idxValidation = ads.Labels.Speaker==speaker(i);
    adsValidation = subset(ads,idxValidation);
    adsValidation.Labels = adsValidation.Labels.Emotion;
    tallValidation = tall(adsValidation);

    % 2. Extract features from the training set. Reorient the features
    % so that time is along rows to be compatible with
    % sequenceInputLayer.
    tallTrain = cellfun(@(x)x/max(abs(x),[]),'all',tallTrain,"UniformOutput",false);
    tallFeaturesTrain = cellfun(@(x)extract(extractor,x),tallTrain,"UniformOutput",false);
    tallFeaturesTrain = cellfun(@(x)x',tallFeaturesTrain,"UniformOutput",false); %#ok<NASGU
    [~,featuresTrain] = evalc('gather(tallFeaturesTrain)'); % Use evalc to suppress command-
    tallValidation = cellfun(@(x)x/max(abs(x),[]),'all',tallValidation,"UniformOutput",false);
    tallFeaturesValidation = cellfun(@(x)extract(extractor,x),tallValidation,"UniformOutput",false);
    tallFeaturesValidation = cellfun(@(x)x',tallFeaturesValidation,"UniformOutput",false); %
    [~,featuresValidation] = evalc('gather(tallFeaturesValidation)'); % Use evalc to suppress

    % 3. Use the training set to determine the mean and standard
    % deviation of each feature. Normalize the training and validation
    % sets.
    allFeatures = cat(2,featuresTrain{:});
    M = mean(allFeatures,2,'omitnan');
    S = std(allFeatures,0,2,'omitnan');
    featuresTrain = cellfun(@(x)(x-M)./S,featuresTrain,"UniformOutput",false);
    for ii = 1:numel(featuresTrain)
        idx = find(isnan(featuresTrain{ii}));
        if ~isempty(idx)
            featuresTrain{ii}(idx) = 0;
        end
    end
end
```

```

    end
end
featuresValidation = cellfun(@(x)(x-M)./S,featuresValidation,'UniformOutput',false);
for ii = 1:numel(featuresValidation)
    idx = find(isnan(featuresValidation{ii}));
    if ~isempty(idx)
        featuresValidation{ii}(idx) = 0;
    end
end

% 4. Buffer the sequences so that each sequence consists of twenty
% feature vectors with overlaps of 10 feature vectors.
featureVectorsPerSequence = 20;
featureVectorOverlap = 10;
[sequencesTrain,sequencePerFileTrain] = HelperFeatureVector2Sequence(featuresTrain,featuresValidation);
[sequencesValidation,sequencePerFileValidation] = HelperFeatureVector2Sequence(featuresTrain,featuresValidation);

% 5. Replicate the labels of the train and validation sets so that
% they are in one-to-one correspondence with the sequences.
labelsTrain = [emptyEmotions;augadsTrain.Labels];
labelsTrain = labelsTrain(:);
labelsTrain = repelem(labelsTrain,[sequencePerFileTrain{:}]);

% 6. Define a BiLSTM network.
dropoutProb1 = 0.3;
numUnits      = 200;
dropoutProb2 = 0.6;
layers = [ ...
    sequenceInputLayer(size(sequencesTrain{1},1))
    dropoutLayer(dropoutProb1)
    bilstmLayer(numUnits,"OutputMode","last")
    dropoutLayer(dropoutProb2)
    fullyConnectedLayer(numel(categories(emptyEmotions)))
    softmaxLayer
    classificationLayer];

% 7. Define training options.
miniBatchSize      = 512;
initialLearnRate    = 0.005;
learnRateDropPeriod = 2;
maxEpochs          = 3;
options = trainingOptions("adam", ...
    "MiniBatchSize",miniBatchSize, ...
    "InitialLearnRate",initialLearnRate, ...
    "LearnRateDropPeriod",learnRateDropPeriod, ...
    "LearnRateSchedule","piecewise", ...
    "MaxEpochs",maxEpochs, ...
    "Shuffle","every-epoch", ...
    "Verbose",false);

% 8. Train the network.
net = trainNetwork(sequencesTrain,labelsTrain,layers,options);

% 9. Evaluate the network. Call classify to get the predicted labels
% for each sequence. Get the mode of the predicted labels of each
% sequence to get the predicted labels of each file.
predictedLabelsPerSequence = classify(net,sequencesValidation);
trueLabels = categorical(adsValidation.Labels);

```

```
    predictedLabels = trueLabels;
    idx1 = 1;
    for ii = 1:numel(trueLabels)
        predictedLabels(ii,:) = mode(predictedLabelsPerSequence(idx1:idx1 + sequencePerFileValidation{ii}));
        idx1 = idx1 + sequencePerFileValidation{ii};
    end
    trueLabelsCrossFold{i} = trueLabels; %#ok<AGROW>
    predictedLabelsCrossFold{i} = predictedLabels; %#ok<AGROW>
end
end
```

References

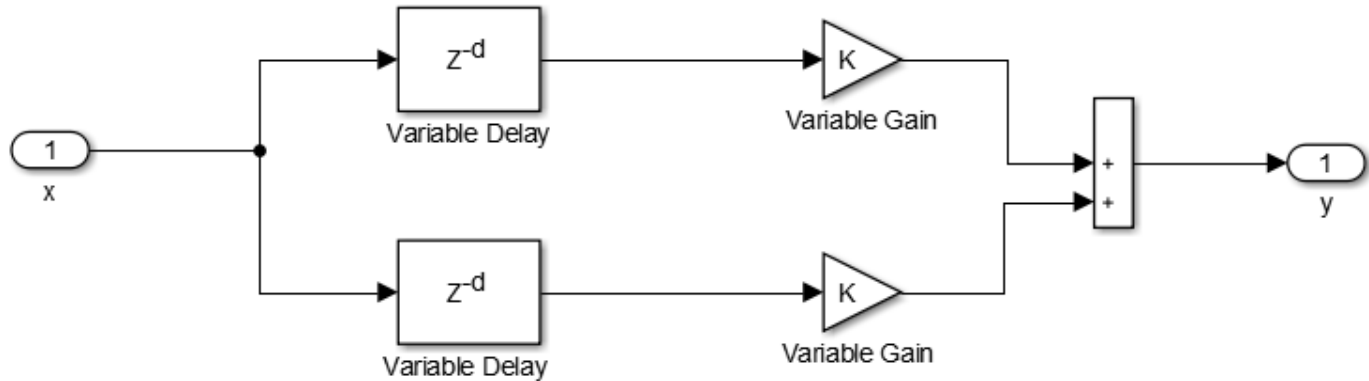
[1] Burkhardt, F., A. Paeschke, M. Rolfes, W.F. Sendlmeier, and B. Weiss, "A Database of German Emotional Speech." In *Proceedings Interspeech 2005*. Lisbon, Portugal: International Speech Communication Association, 2005.

Delay-Based Pitch Shifter

This example shows an audio plugin designed to shift the pitch of a sound in real time.

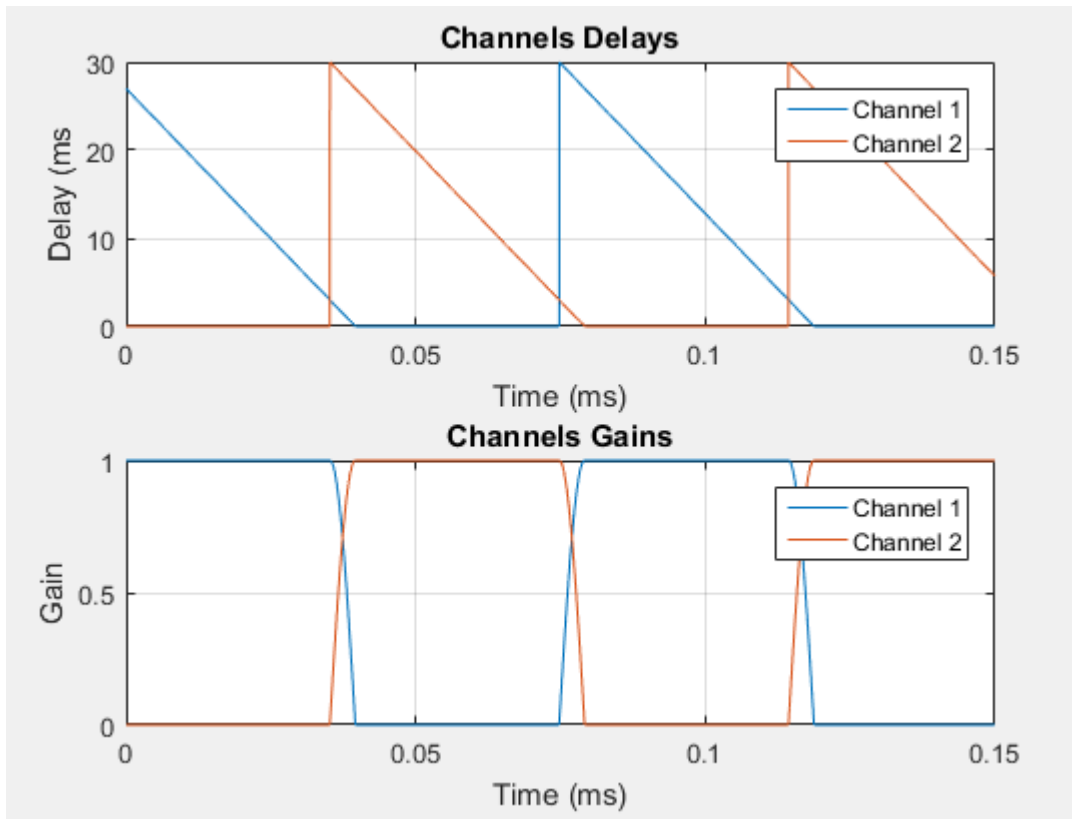
Algorithm

The figure below illustrates the pitch shifting algorithm.



The algorithm is based on cross-fading between two channels with time-varying delays and gains. This method takes advantage of the pitch-shift Doppler effect that occurs as a signal's delay is increased or decreased.

The figure below illustrates the variation of channel delays and gains for an upward pitch shift scenario: The delay of channel 1 decreases at a fixed rate from its maximum value (in this example, 30 ms). Since the gain of channel 2 is initially equal to zero, it does not contribute to the output. As the delay of channel 1 approaches zero, the delay of channel 2 starts decreasing down from 30 ms. In this cross-fading region, the gains of the two channels are adjusted to preserve the output power level. Channel 1 is completely faded out by the time its delay reaches zero. The process is then repeated, going back and forth between the two channels.



For a downward pitch effect, the delays are increased from zero to the maximum value.

The desired output pitch may be controlled by varying the rate of change of the channel delays. Cross-fading reduces the audible glitches that occur during the transition between channels. However, if cross-fading happens over too long a time, the repetitions present in the overlap area may create spurious modulation and comb-filtering effects.

Pitch Shifter Audio Plugin

`audiopluginexample.PitchShifter` is an audio plugin object that implements the delay-based pitch shifting algorithm. The plugin parameters are the pitch shift (in semi-tones), and the cross-fading factor (which controls the overlap between the two delay branches). You can incorporate the object into a MATLAB simulation, or use it to generate an audio plugin using `generateAudioPlugin`.

In addition to the output audio signal, the object returns two extra outputs, corresponding to the delays and gains of the two channels, respectively.

You can open a test bench for `audiopluginexample.PitchShifter` by using Audio Test Bench. The test bench provides a user interface (UI) to help you test your audio plugin in MATLAB. You can tune the plugin parameters as the test bench is executing. You can also open a `dsp.TimeScope` and a `dsp.SpectrumAnalyzer` to view and compare the input and output signals in the time and frequency domains, respectively.

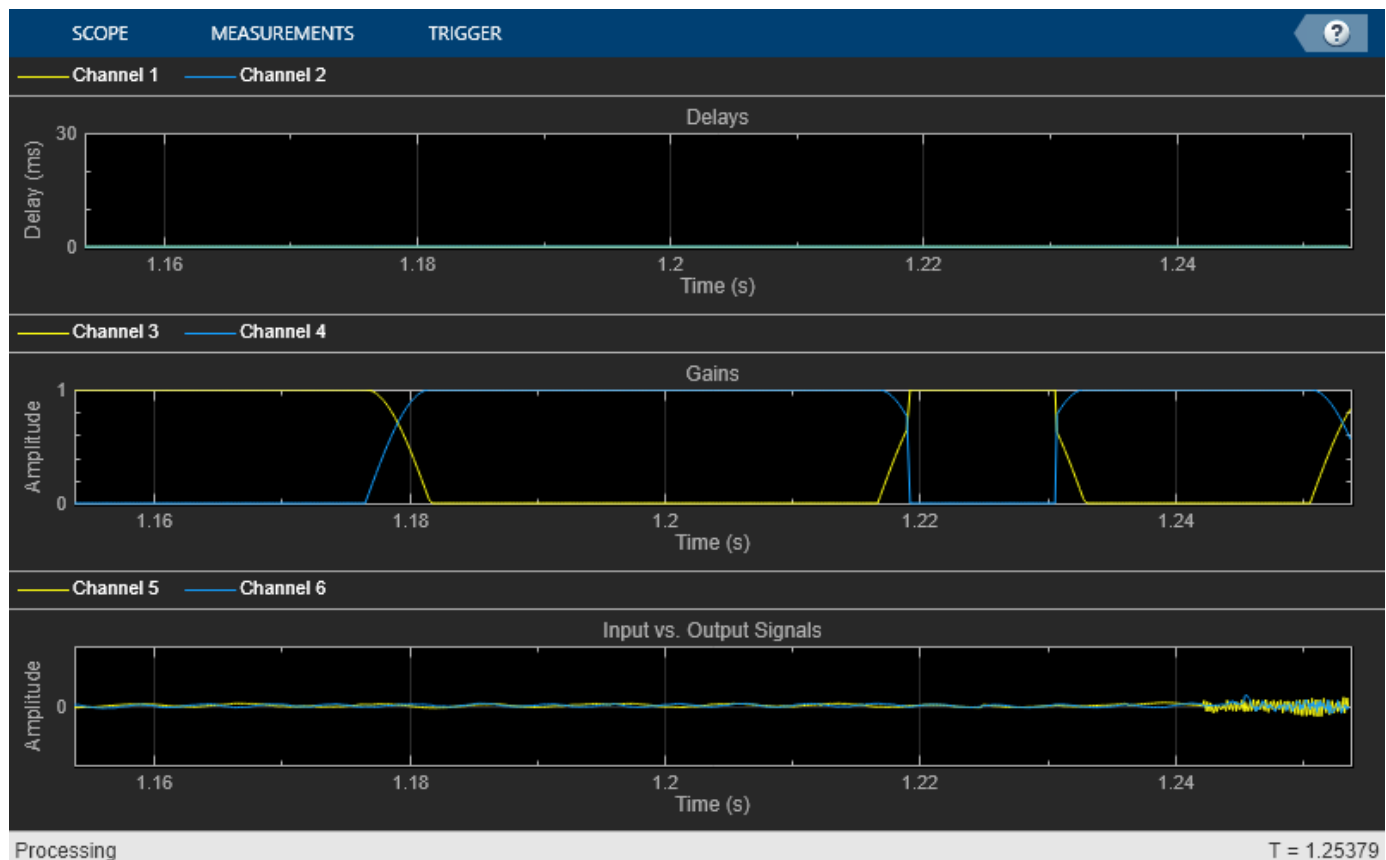
You can also use `audiopluginexample.PitchShifter` in MATLAB just as you would use any other MATLAB object. You can use the `configureMIDI` command to enable tuning the object via a MIDI device. This is particularly useful if the object is part of a streaming MATLAB simulation where the command window is not free.

`runPitchShift` is a simple function that may be used to perform pitch shifting as part of a larger MATLAB simulation. The function instantiates an `audiopluginexample.PitchShifter` plugin, and uses the `setSampleRate` method to set its sampling rate to the input argument `Fs`. The plugin's parameter's are tuned by setting their values to the input arguments `pitch` and `overlap`, respectively. Note that it is also possible to generate a MEX-file from this function using the `codegen` command. Performance is improved in this mode without compromising the ability to tune parameters.

MATLAB Simulation

`audioPitchShifterExampleApp` implements a real-time pitch shifting app.

Execute `audioPitchShifterExampleApp` to open the app. In addition to playing the pitch-shifted output audio, the app plots the time-varying channel delays and gains, as well as the input and output signals.



`audioPitchShifterExampleApp` opens a UI designed to interact with the simulation. The UI allows you to tune the parameters of the pitch shifting algorithm, and the results are reflected in the simulation instantly. The plots reflect your changes as you tune these parameters. For more information on the UI, call `help HelperCreateParamTuningUI`.

`audioPitchShifterExampleApp` wraps around `HelperPitchShifterSim` and iteratively calls it. `HelperPitchShifterSim` instantiates, initializes and steps through the objects forming the algorithm.

MATLAB Coder can be used to generate C code for `HelperPitchShifterSim`. In order to generate a MEX-file for your platform, execute `HelperPitchShifterCodeGeneration` from a folder with write permissions.

By calling `audioPitchShifterExampleApp` with `'true'` as an argument, the generated MEX-file `HelperPitchShifterSimMEX` can be used instead of `HelperPitchShifterSim` for the simulation. In this scenario, the UI is still running inside the MATLAB environment, but the main processing algorithm is being performed by a MEX-file. Performance is improved in this mode without compromising the ability to tune parameters.

Call `audioPitchShifterExampleApp` with `'true'` as argument to use the MEX-file for simulation. Again, the simulation runs till the user explicitly stops it from the UI.

References

- [1] 'Using Multiple Processors for Real-Time Audio Effects', Bogdanowicz, K. ; Belcher, R; AES - May 1989.
- [2] 'A Detailed Analysis of a Time-Domain Formant-Corrected Pitch-Shifting Algorithm', Bristow-Johnson, R. ; AES - October 1993.

Psychoacoustic Bass Enhancement for Band-Limited Signals

This example shows an audio plugin designed to enhance the perceived sound level in the lower part of the audible spectrum.

Introduction

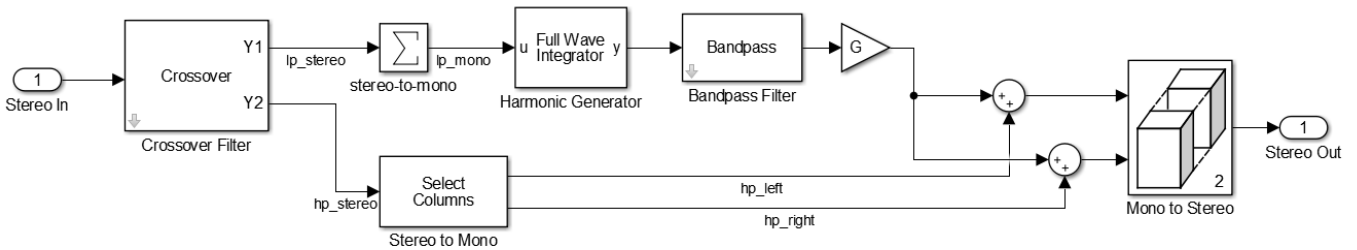
Small loudspeakers typically have a poor low frequency response, which can have a negative impact on overall sound quality. This example implements psychoacoustic bass enhancement to improve sound quality of audio played on small loudspeakers.

The example is based on the algorithm in [1 on page 1-0]. A non-linear device shifts the low-frequency range of the signal to a high-frequency range through the generation of harmonics. The pitch of the original signal is preserved due to the "virtual pitch" psychoacoustic phenomenon.

The algorithm is implemented using an audio plugin object.

Algorithm

The figure below illustrates the algorithm used in [1 on page 1-0].



1. The input stereo signal is split into lowpass and highpass components using a crossover filter. The filter's crossover frequency is equal to the speaker's cutoff frequency (set to 60 Hz in this example).

2. The highpass component, hp_{stereo} , is split into left and right channels: hp_{left} and hp_{right} , respectively.

3. The lowpass component, lp_{stereo} , is converted to mono, lp_{mono} , by adding the left and right channels element by element.

4. lp_{mono} is passed through a full wave integrator. The full wave integrator shifts lp_{mono} to higher harmonics.

$$y[n] = \begin{cases} 0 & \text{if } u[n] > 0 \text{ and } u[n-1] \leq 0 \\ y[n-1] + u[n-1] & \text{else} \end{cases}$$

- $u[n]$ is the input signal, lp_{mono}
- $y[n]$ is the output signal

- n is the time index

5. $y[n]$ is passed through a bandpass filter with lower cutoff frequency set to the speaker's cutoff frequency. The bandpass's upper cutoff frequency may be adjusted to fine-tune output sound quality.

6. $y_{BP}[n]$, the bandpass filtered signal, passes through tunable gain, G .

7. y_G is added to the left and right highpass channels.

8. The left and right channels are concatenated into a single matrix and output.

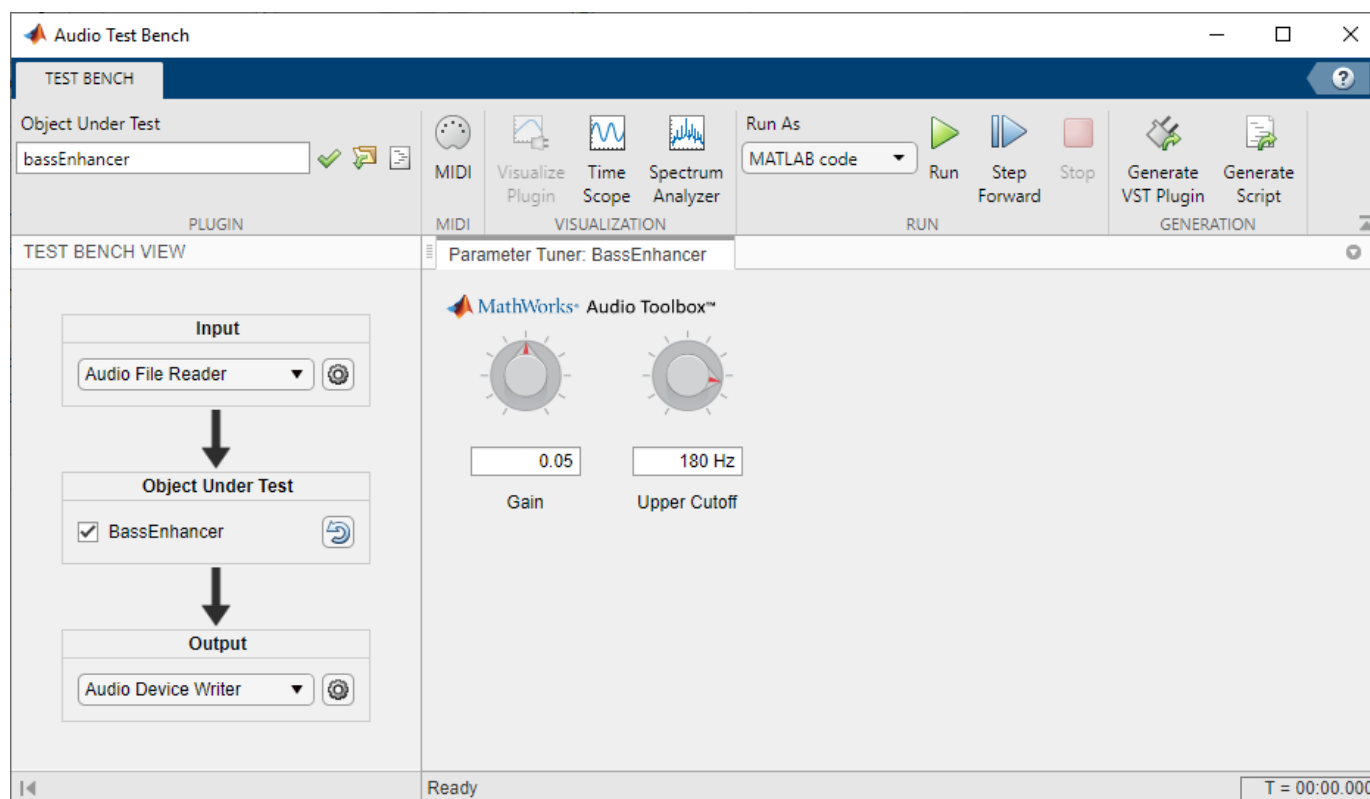
Although the resulting output stereo signal does not contain low-frequency elements, the input's bass pitch is preserved thanks to the generated harmonics.

Bass Enhancer Audio plugin

`audiopluginexample.BassEnhancer` is an audio plugin object that implements the psychoacoustic bass enhancement algorithm. The plugin parameters are the upper cutoff frequency of the bandpass filter, and the gain applied at the output of the bandpass filter (G in the diagram above). You can incorporate the object into a MATLAB simulation, or use it to generate an audio plugin using `generateAudioPlugin`.

You can open a test bench for `audiopluginexample.BassEnhancer` using Audio Test Bench. The test bench provides a graphical user interface to help you test your audio plugin in MATLAB. You can tune the plugin parameters as the test bench is executing. You can also open a `timescope` and a `dsp.SpectrumAnalyzer` to view and compare the input and output signals in the time and frequency domains, respectively.

```
bassEnhancer = audiopluginexample.BassEnhancer;  
audioTestBench(bassEnhancer)
```



You can also use `audiopluginexample.BassEnhancer` in MATLAB just as you would use any other MATLAB object. You can use `configureMIDI` to enable tuning the object using a MIDI device. This is particularly useful if the object is part of a streaming MATLAB simulation where the command window is not free.

`HelperBassEnhancerSim` is a simple function that may be used to perform bass enhancement as part of a larger MATLAB simulation. The function instantiates an `audiopluginexample.BassEnhancer` plugin, and uses the `setSampleRate` method to set its sampling rate to the input argument `Fs`. The plugin's parameters are tuned by setting their values to the input arguments `Fcutoff` and `G`, respectively. Note that it is also possible to generate a MEX-file from this function using the `codegen` command. Performance is improved in this mode without compromising the ability to tune parameters.

References

[1] Aarts, Ronald M, Erik Larsen, and Daniel Schobben. "Improving Perceived Bass and Reconstruction of High Frequencies for Band Limited Signals." *Proceedings 1st IEEE Benelux Workshop on Model Based Coding of Audio (MPCA-2002)*, November 15, 2002, 59-71.

Tunable Filtering and Visualization Using Audio Plugins

This example shows how to visualize the magnitude response of a tunable filter. The filters in this example are implemented as audio plugins. This example uses the `visualize` and `audioTestBench` functionality of the Audio Toolbox™.

Tunable Filter Examples

Audio Toolbox provides several examples of tunable filters that have been implemented as audio plugins:

```
audiopluginexample.BandpassIIRFilter  
audiopluginexample.HighpassIIRFilter  
audiopluginexample.LowpassIIRFilter  
audiopluginexample.ParametricEqualizerWithUDP  
audiopluginexample.ShelvingEqualizer  
audiopluginexample.VarSlopeBandpassFilter
```

visualize

All of these example audio plugins can be used with the `visualize` function in order to view the magnitude response of the filters as they are tuned in real time.

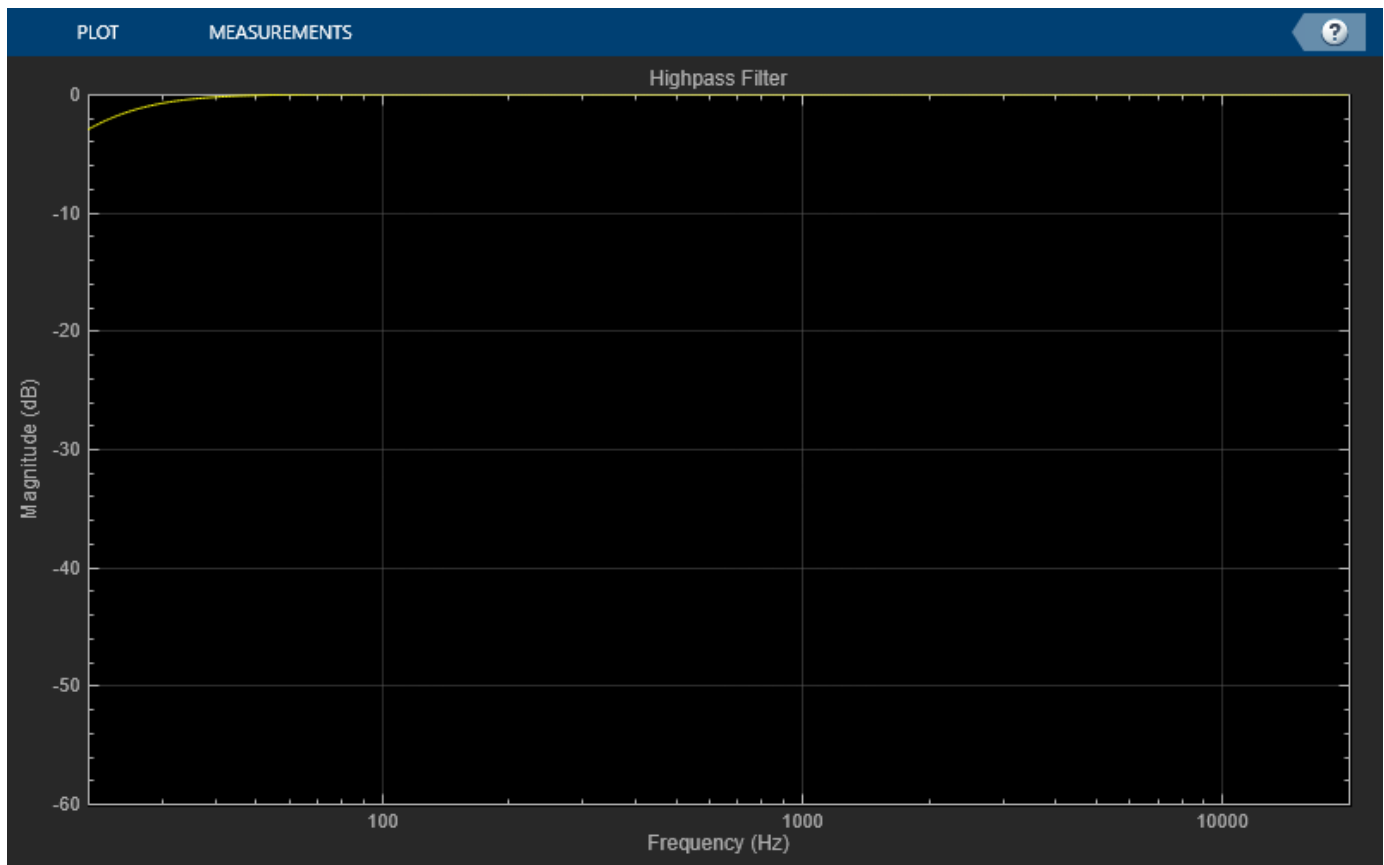
audioTestBench

Any audio plugin can be tuned in real time using `audioTestBench`. The tool allows you to test an audio plugin with audio signals from a file or device. The tool also enables you to view the power spectrum and the time-domain waveform for the input and output signals.

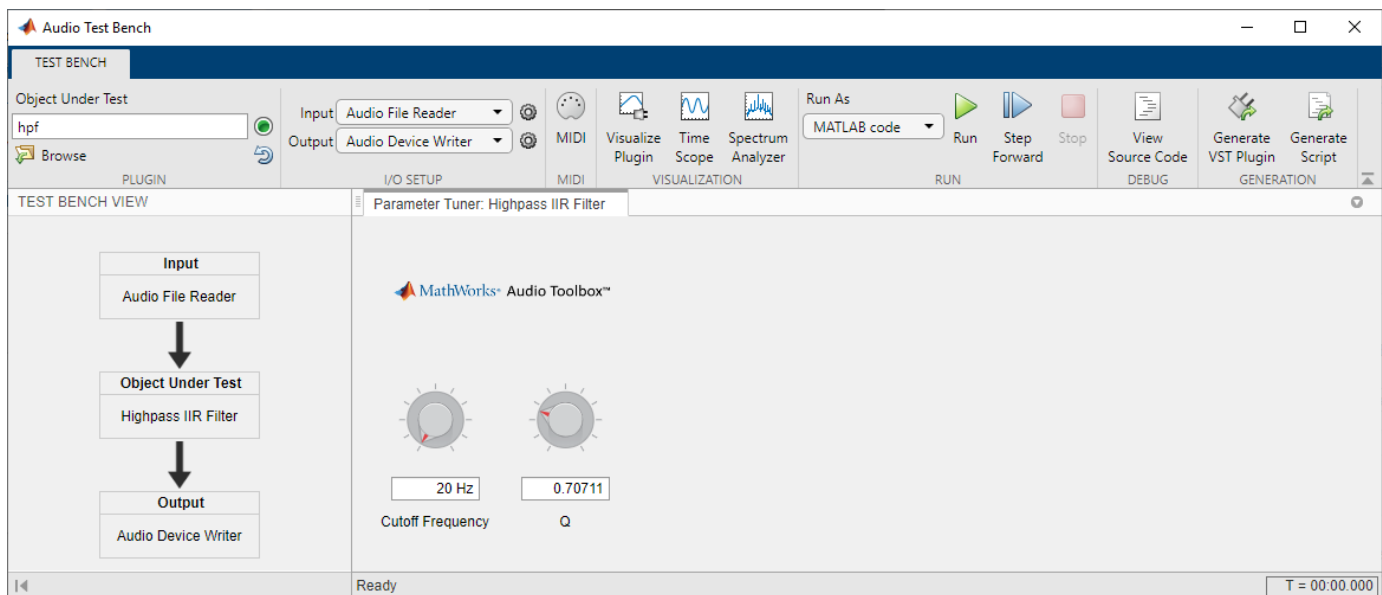
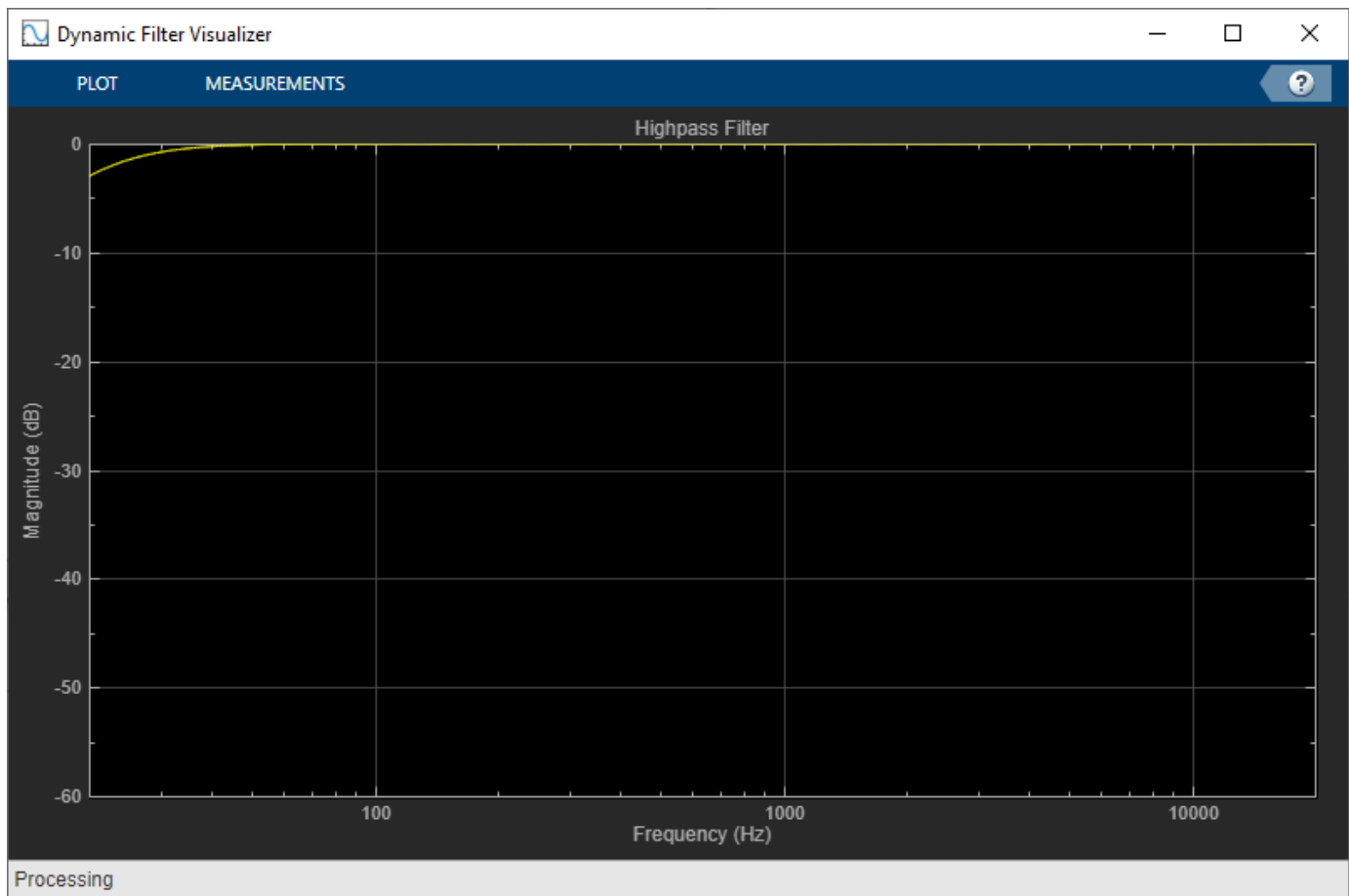
Update Visualization While Running Plugin

`audiopluginexample.BandpassIIRFilter`, `audiopluginexample.HighpassIIRFilter`, and `audiopluginexample.LowpassIIRFilter` are the simplest of the six examples because the code is written so that the visualization is updated only when data is processed by the filter. Create the audio plugin, then call `visualize` and `audioTestBench`

```
hpf = audiopluginexample.HighpassIIRFilter;  
visualize(hpf)
```



```
audioTestBench(hpf)
```

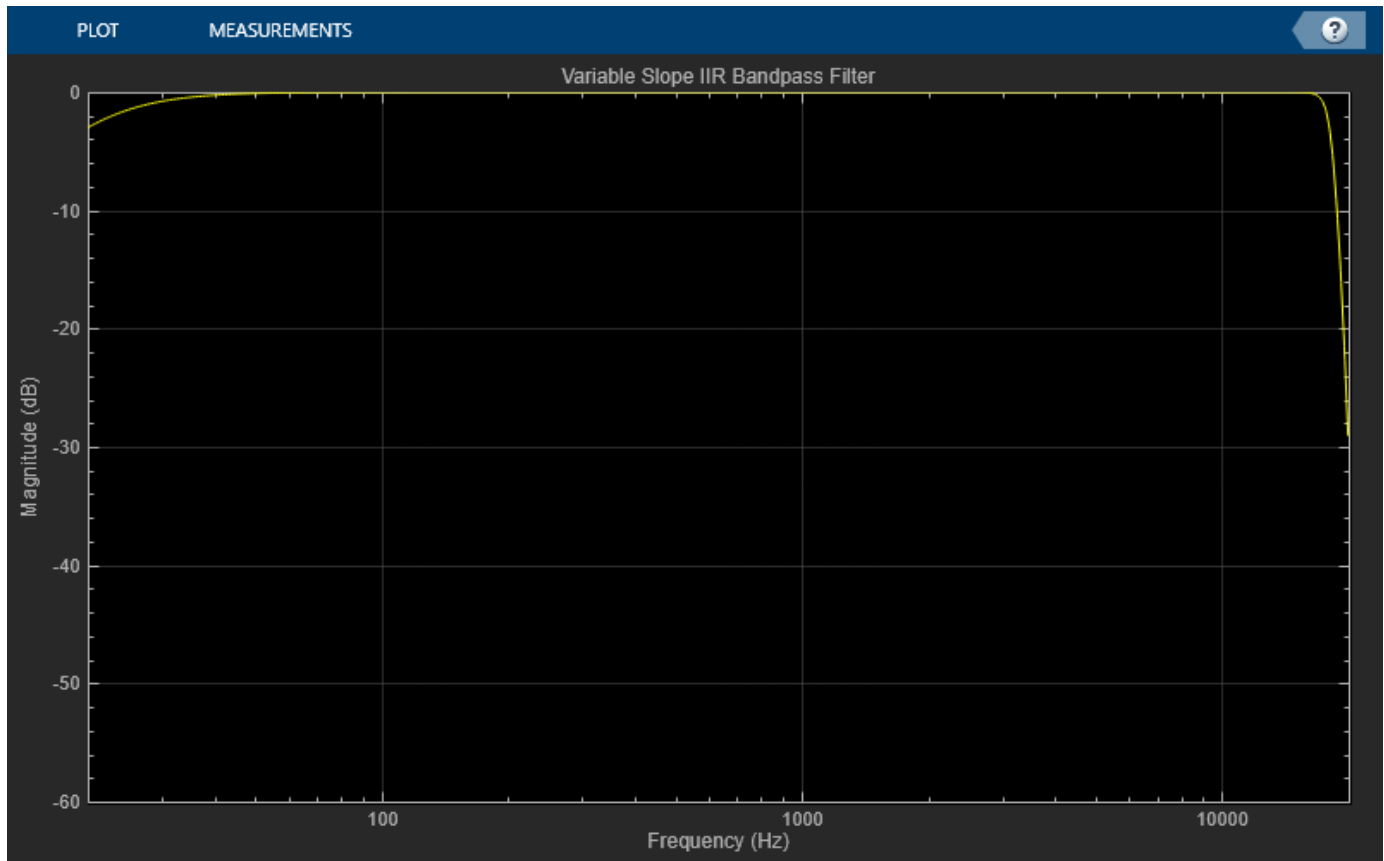


Note that moving the cutoff frequency in `audioTestBench` does not update the magnitude response plot. However, once the 'Run' (or play) button is pressed, you can see and hear the changing magnitude response of the filter as the cutoff frequency is tuned in real time.

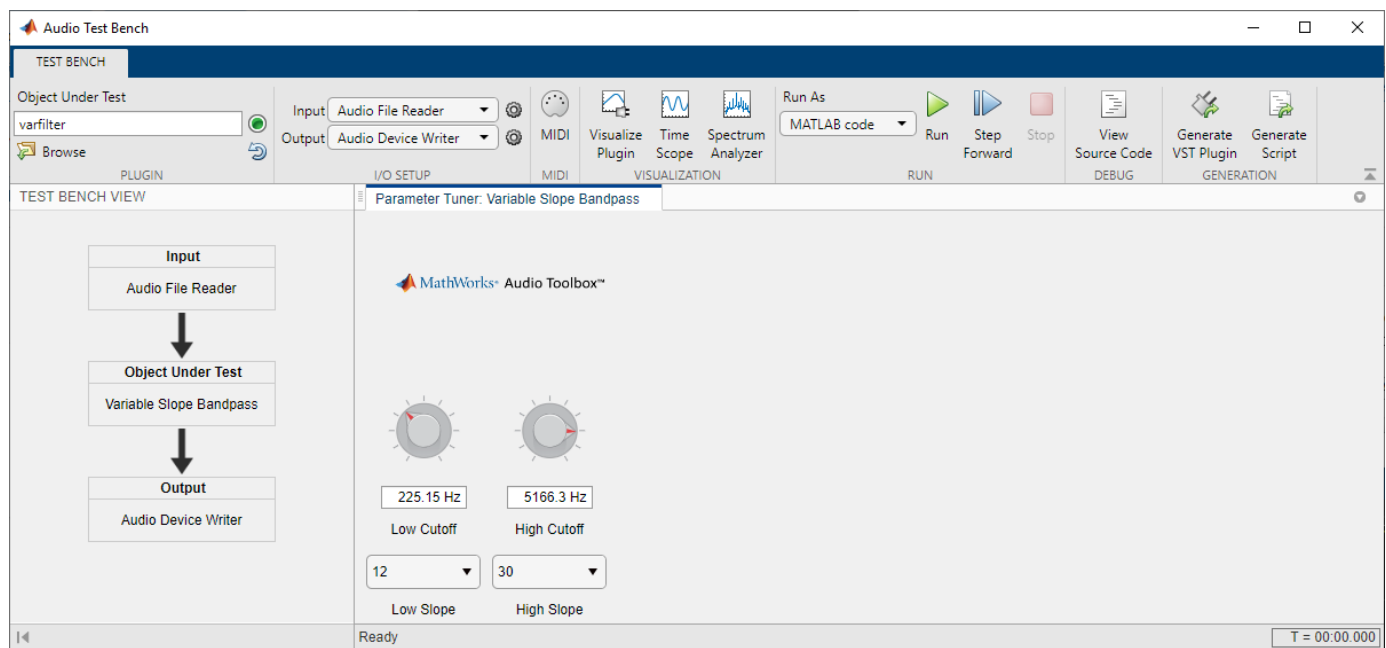
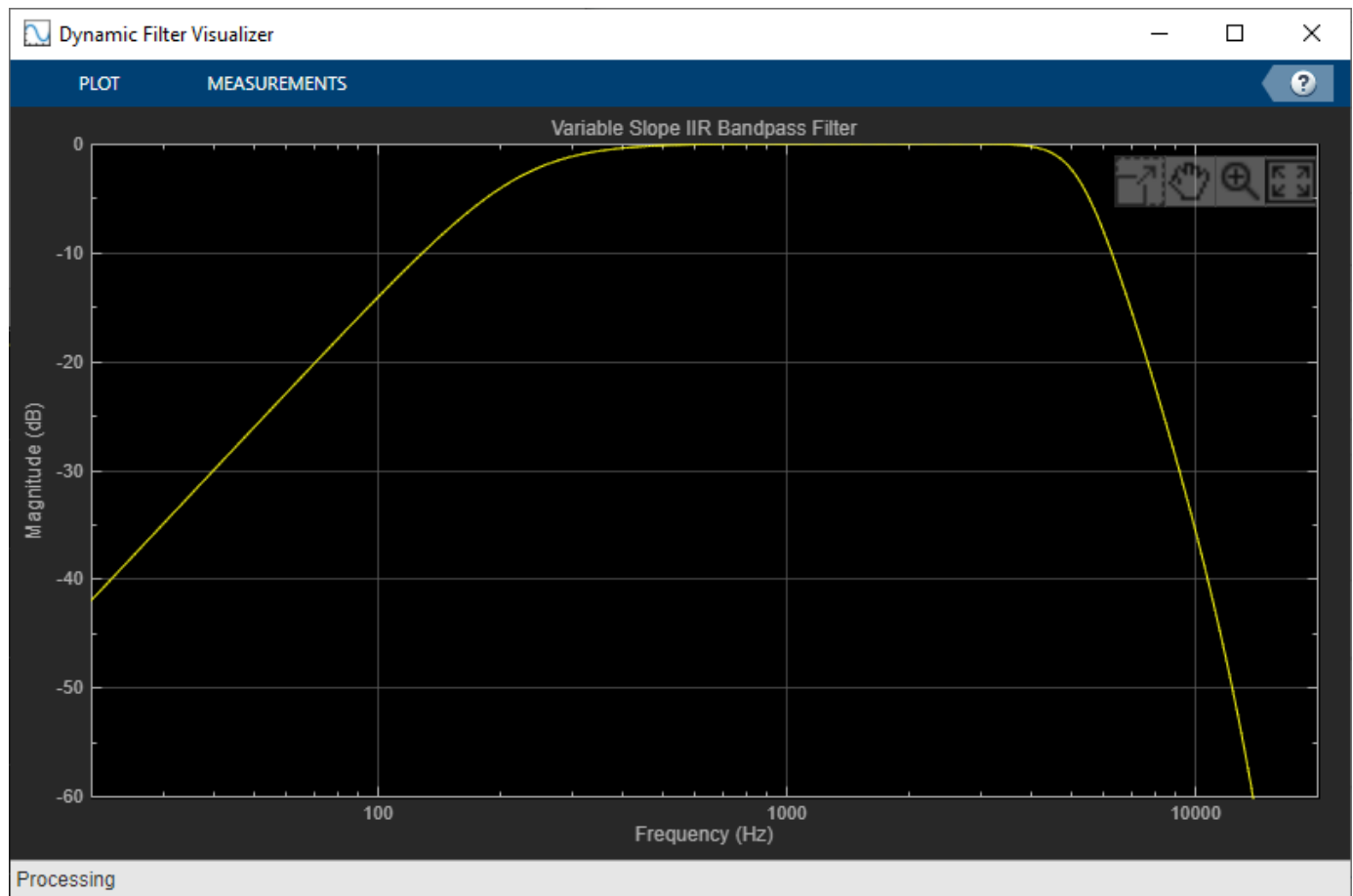
Update Visualization at Any Time

`audiopluginexample.ShelvingEqualizer` and `audiopluginexample.VarSlopeBandpassFilter` have `visualize` functions which update the magnitude response plot even when not processing data. The visualization is also updated in real time once audio is being processed.

```
audioTestBench('-close')  
varfilter = audiopluginexample.VarSlopeBandpassFilter;  
visualize(varfilter)
```



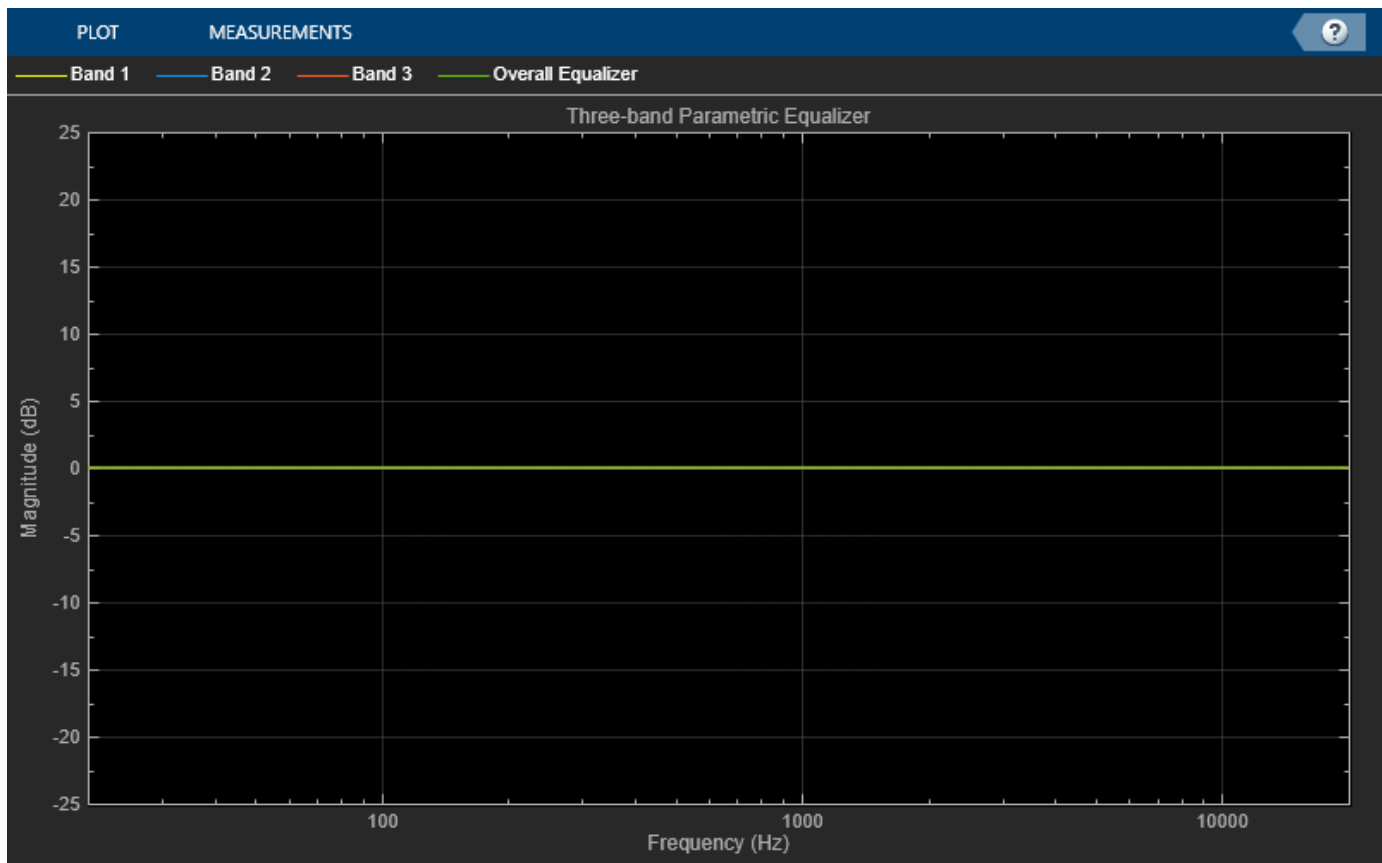
```
audioTestBench(varfilter)
```



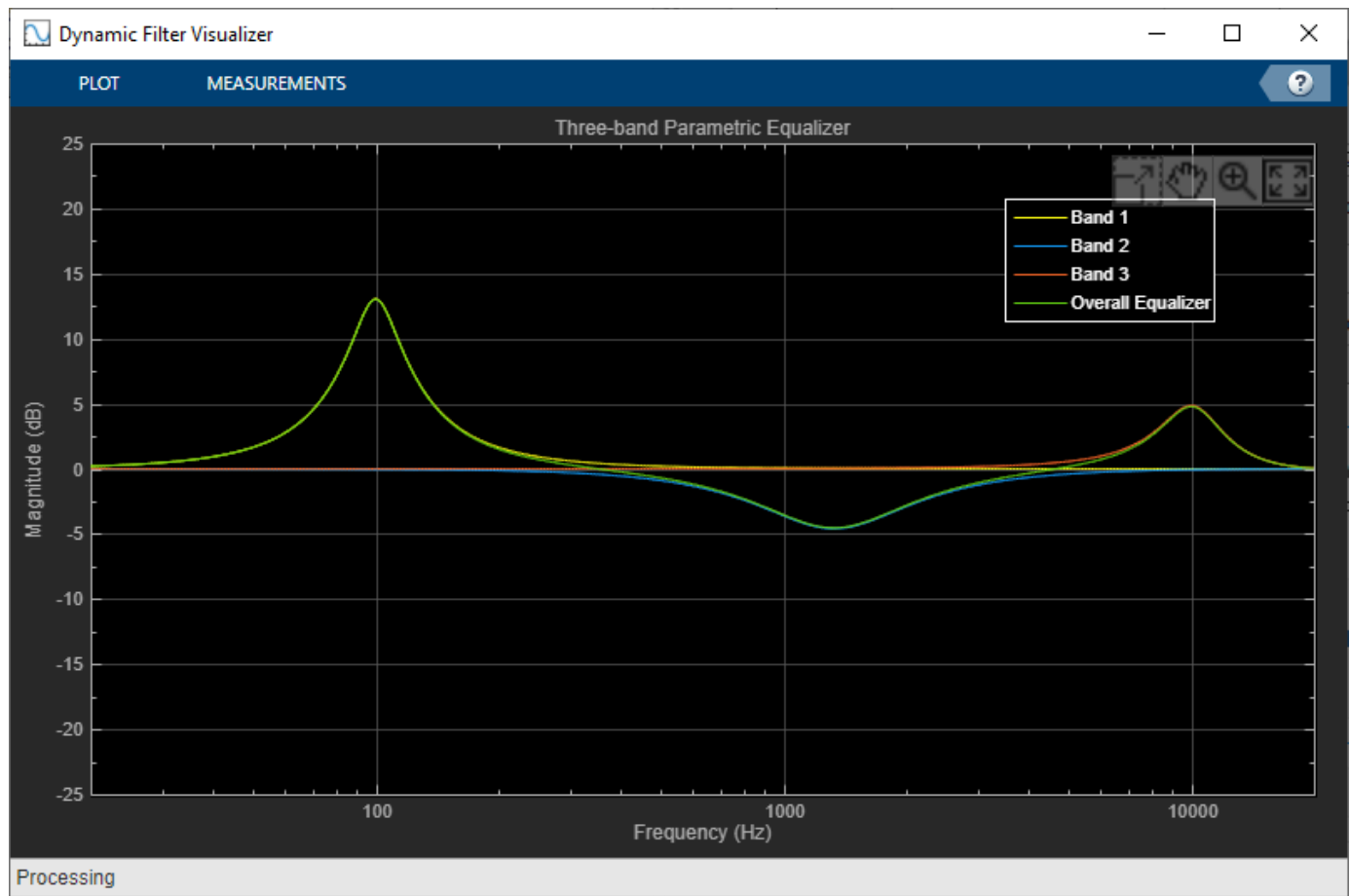
Visualize Individual and Combined Magnitude Response

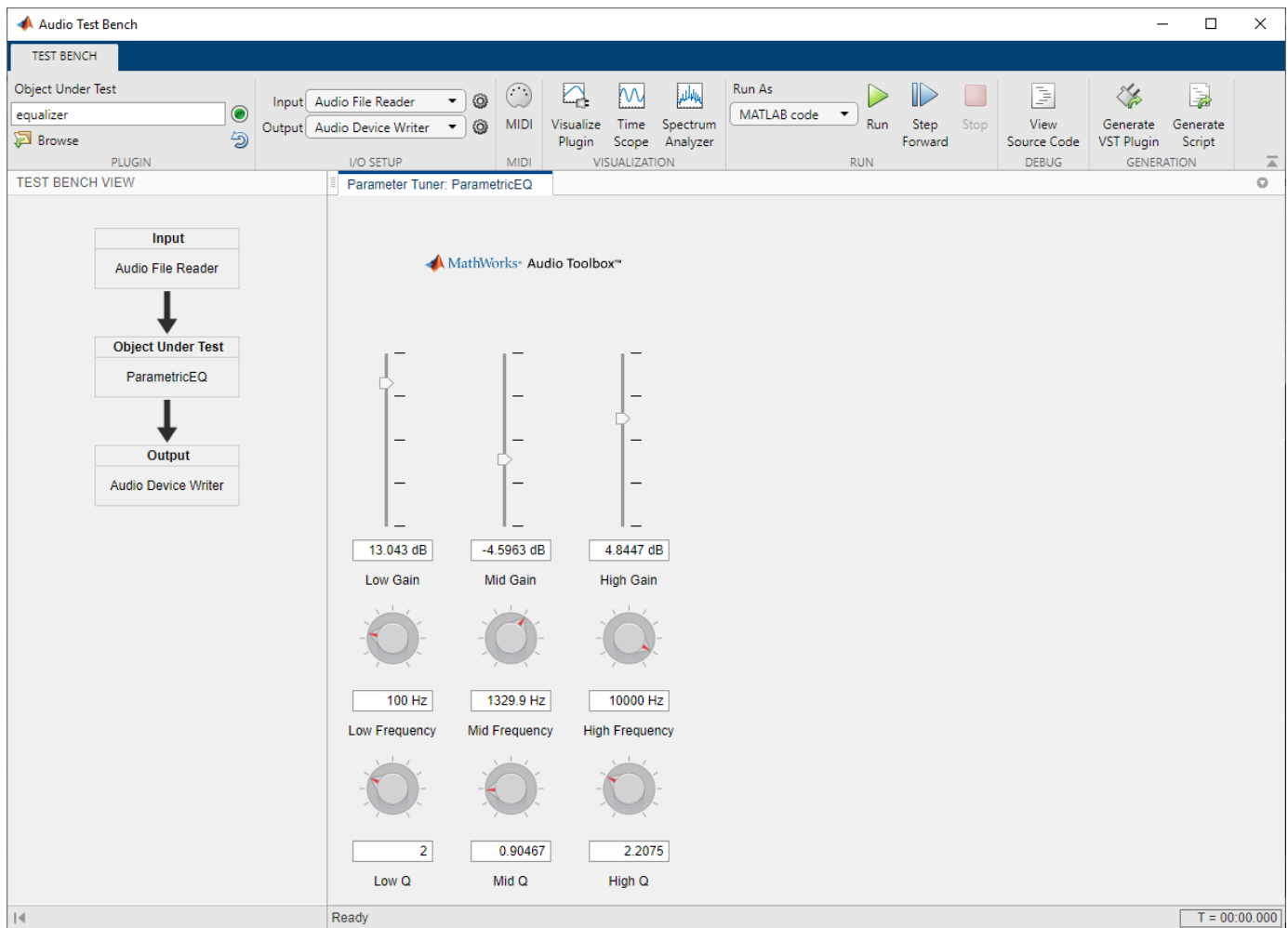
`audiopluginexample.ParametricEqualizerWithUDP` illustrates how to visualize individual sections in a 3-section biquad filter along with the overall response of the 3 sections combined.

```
audioTestBench('-close')  
equalizer = audiopluginexample.ParametricEqualizerWithUDP;  
visualize(equalizer)
```



```
audioTestBench(equalizer)
```





```
audioTestBench('-close')
```

Communicate Between a DAW and MATLAB Using UDP

This example shows how to communicate between a digital audio workstation (DAW) and MATLAB using the user datagram protocol (UDP). The information shared between the DAW and MATLAB can be used to perform visualization in real time in MATLAB on parameters that are being changed in the DAW.

User Datagram Protocol (UDP)

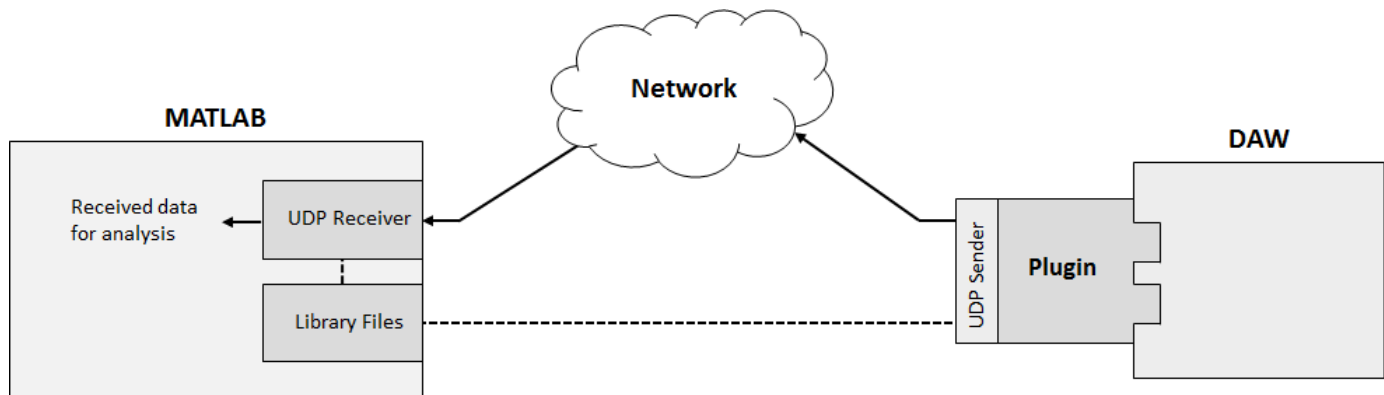
UDP is a core member of the Internet protocol suite. It is a simple connectionless transmission that does not employ any methods for error checking. Because it does not check for errors, UDP is a fast but unreliable alternative to the transmission control protocol (TCP) and stream control transmission protocol (SCTP). UDP is widely used in applications that are willing to trade fidelity for high-speed transmission, such as video conferencing and real-time computer games. If you use UDP for communication within a single machine, packets are less likely to drop. The tutorials outlined here work best when executed on a single machine.

UDP and MATLAB

These System objects enable you to use UDP with MATLAB:

- `dsp.UDPReceiver` - Receive UDP packets from network
- `dsp.UDPSender` - Send UDP packets to network

To communicate between a DAW and MATLAB using UDP, place a UDP sender in the plugin used in the DAW, and run a corresponding UDP receiver in MATLAB.



The `dsp.UDPSender` and `dsp.UDPReceiver` System objects use prebuilt library files that are included with MATLAB.

Example Plugins

These Audio Toolbox™ example plugins use UDP:

- `audiopluginexample.UDPSender` - Send an audio signal from a DAW to the network. If you generate this plugin and deploy it to a DAW, the plugin sends frames of a stereo signal to the network. The frame size is determined by the DAW. You can modify the example plugin to send any information you want to analyze in MATLAB.
- `audiopluginexample.ParametricEqualizerWithUDP` - Send a plugin's filter coefficients from a DAW to the network. If you generate this plugin and run it in a DAW, the plugin sends the

coefficients of the parametric equalizer you tune in the DAW to the network. The `HelperUDPPluginVisualizer` function contains a UDP receiver that receives the datagram, and uses it to plot the magnitude response of the filter you are tuning in a DAW.

Send Audio from DAW to MATLAB

Step 1: Generate a VST Plugin

To generate a VST plugin from `audiopluginexample.UDPSender`, use the `generateAudioPlugin` function. It is a best practice to move to a directory that can store the generated plugin before executing this command:

```
generateAudioPlugin audiopluginexample.UDPSender
```

.....

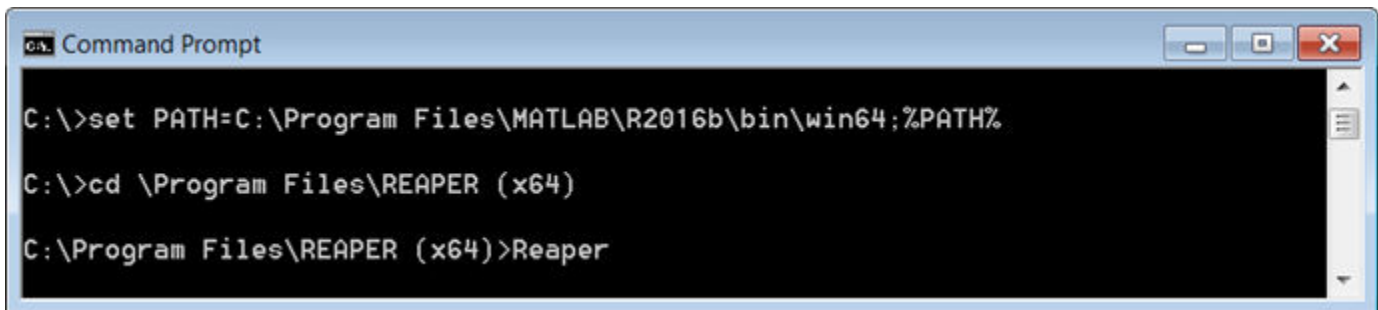
The generated plugin is saved to your current folder and named `UDPSender`.

Step 2: Open DAW with Appropriate Environment Variables Set

To run the UDP sender outside of MATLAB, you must open the DAW from a command terminal with the appropriate environment variables set. Setting environment variables enables the deployed UDP sender to use the necessary library files in MATLAB. To learn how to set the environment variables, see the tutorial specific to your system:

- “Set Run-Time Library Path on Windows Systems”
- “Set Run-Time Library Path on macOS Systems”

After you set the environment variables, open your DAW from the same command terminal, such as in this example terminal from a Windows system.

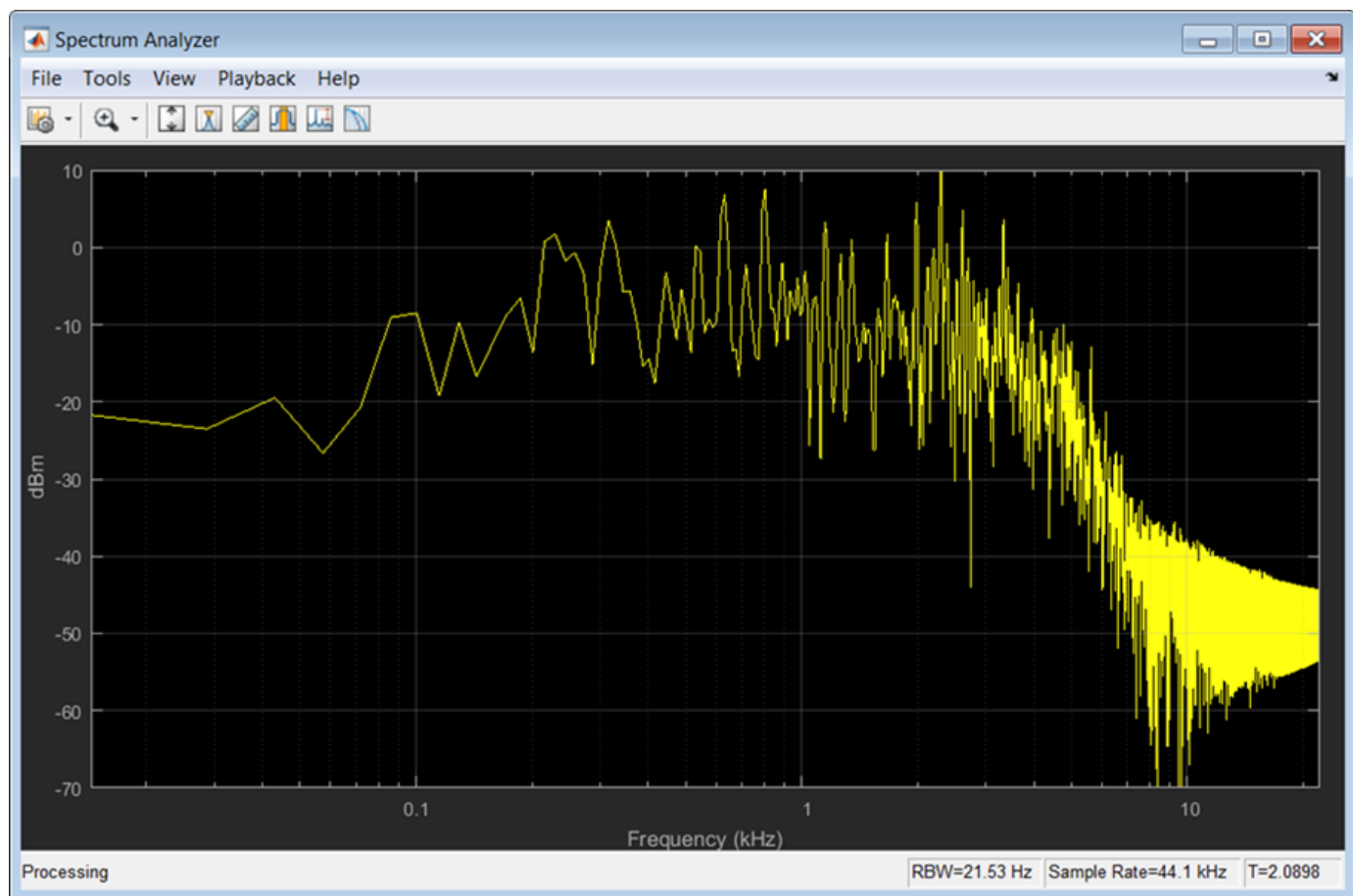


```
C:\>set PATH=C:\Program Files\MATLAB\R2016b\bin\win64;%PATH%
C:\>cd \Program Files\REAPER (x64)
C:\Program Files\REAPER (x64)>Reaper
```

Step 3: Receive and Process an Audio Signal

- In the DAW, open the generated `UDPSender` file.
- In MATLAB, run this function: `HelperUDPPluginReceiver`

The audio signal is displayed on the `dsp.SpectrumAnalyzer` for analysis.

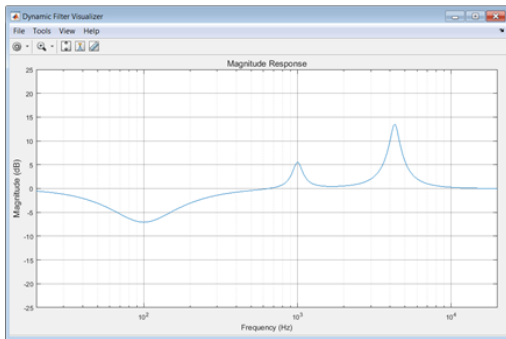


Send Coefficients from DAW to MATLAB

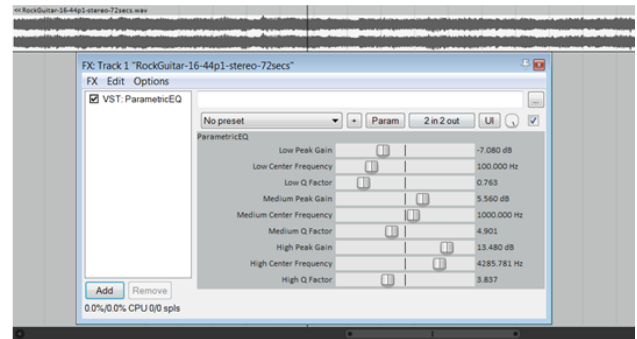
1. Follow steps 1-2 from **Send Audio from DAW to MATLAB**, replacing `audiopluginexample.UDPSender` with `audiopluginexample.ParametricEqualizerWithUDP`.
2. Receive and process filter coefficients
 - a. In the DAW, open the generated `ParameterEqualizerWithUDP` file. The plugin display name is `ParametricEQ`.
 - b. In MATLAB, run this command: `HelperUDPPluginVisualizer`

The `HelperUDPPluginVisualizer` function uses a `dsp.UDPReceiver` to receive the filter coefficients and then displays the magnitude response for 60 seconds. You can modify the code to extend or reduce the amount of time. The plotted magnitude response corresponds to the parametric equalizer plugin you tune in the DAW.

MATLAB



DAW



Acoustic Echo Cancellation (AEC)

This example shows how to apply adaptive filters to acoustic echo cancellation (AEC).

Author(s): Scott C. Douglas

Introduction

Acoustic echo cancellation is important for audio teleconferencing when simultaneous communication (or full-duplex transmission) of speech is necessary. In acoustic echo cancellation, a measured microphone signal $d(n)$ contains two signals:

- The near-end speech signal $v(n)$
- The far-end echoed speech signal $\hat{d}(n)$

The goal is to remove the far-end echoed speech signal from the microphone signal so that only the near-end speech signal is transmitted. This example has some sound clips, so you might want to adjust your computer's volume now.

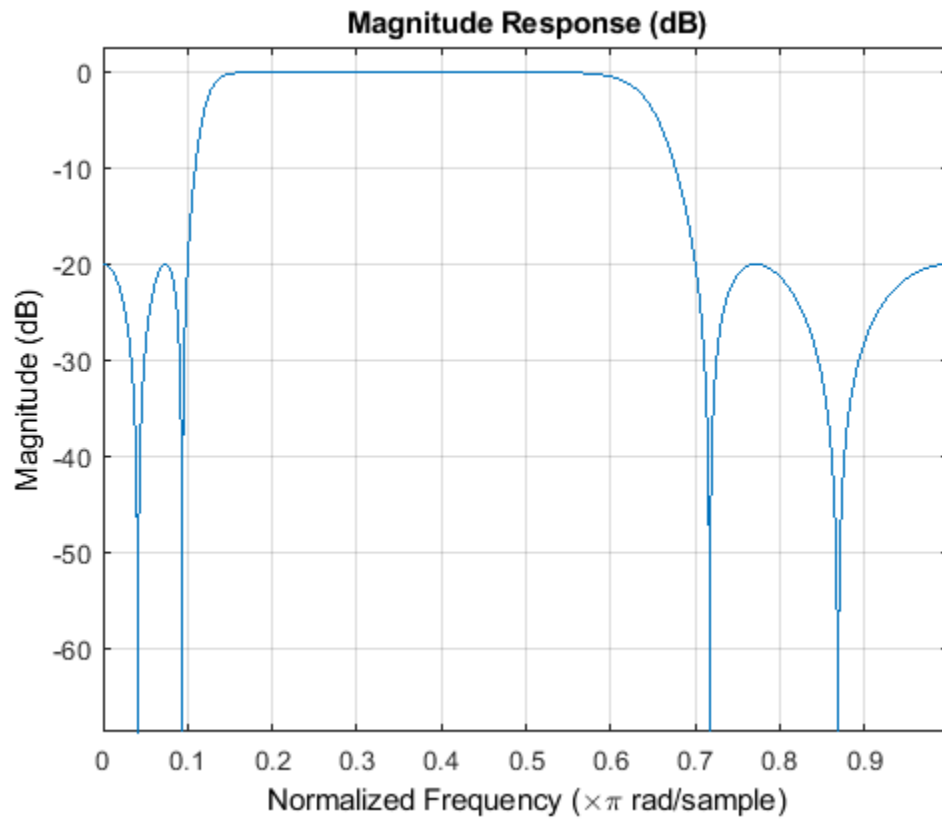
The Room Impulse Response

You first need to model the acoustics of the loudspeaker-to-microphone signal path where the speakerphone is located. Use a long finite impulse response filter to describe the characteristics of the room. The following code generates a random impulse response that is not unlike what a conference room would exhibit. Assume a system sample rate of 16000 Hz.

```
fs = 16000;
M = fs/2 + 1;
frameSize = 2048;

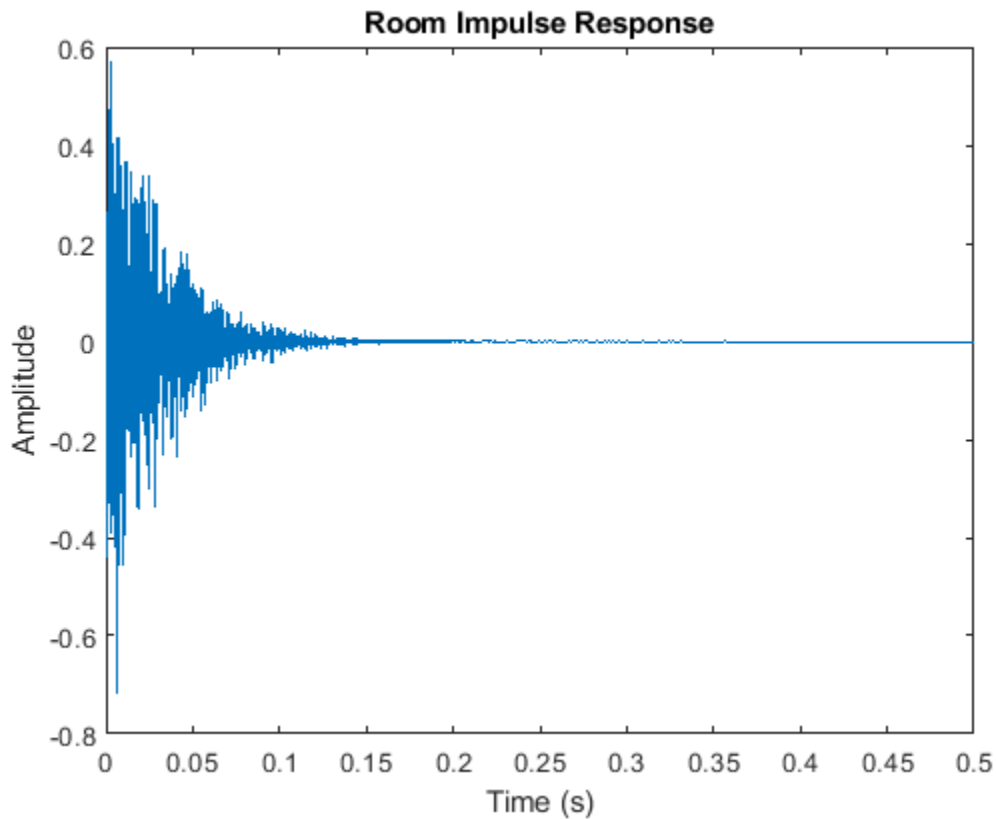
[B,A] = cheby2(4,20,[0.1 0.7]);
impulseResponseGenerator = dsp.IIRFilter('Numerator', [zeros(1,6) B], ...
    'Denominator', A);

FVT = fvtool(impulseResponseGenerator); % Analyze the filter
FVT.Color = [1 1 1];
```

```
roomImpulseResponse = impulseResponseGenerator( ...
    (log(0.99*rand(1,M)+0.01).*sign(randn(1,M)).*exp(-0.002*(1:M)))');
roomImpulseResponse = roomImpulseResponse/norm(roomImpulseResponse)*4;
room = dsp.FIRFilter('Numerator', roomImpulseResponse);

fig = figure;
plot(0:1/fs:0.5, roomImpulseResponse);
xlabel('Time (s)');
ylabel('Amplitude');
title('Room Impulse Response');
fig.Color = [1 1 1];
```



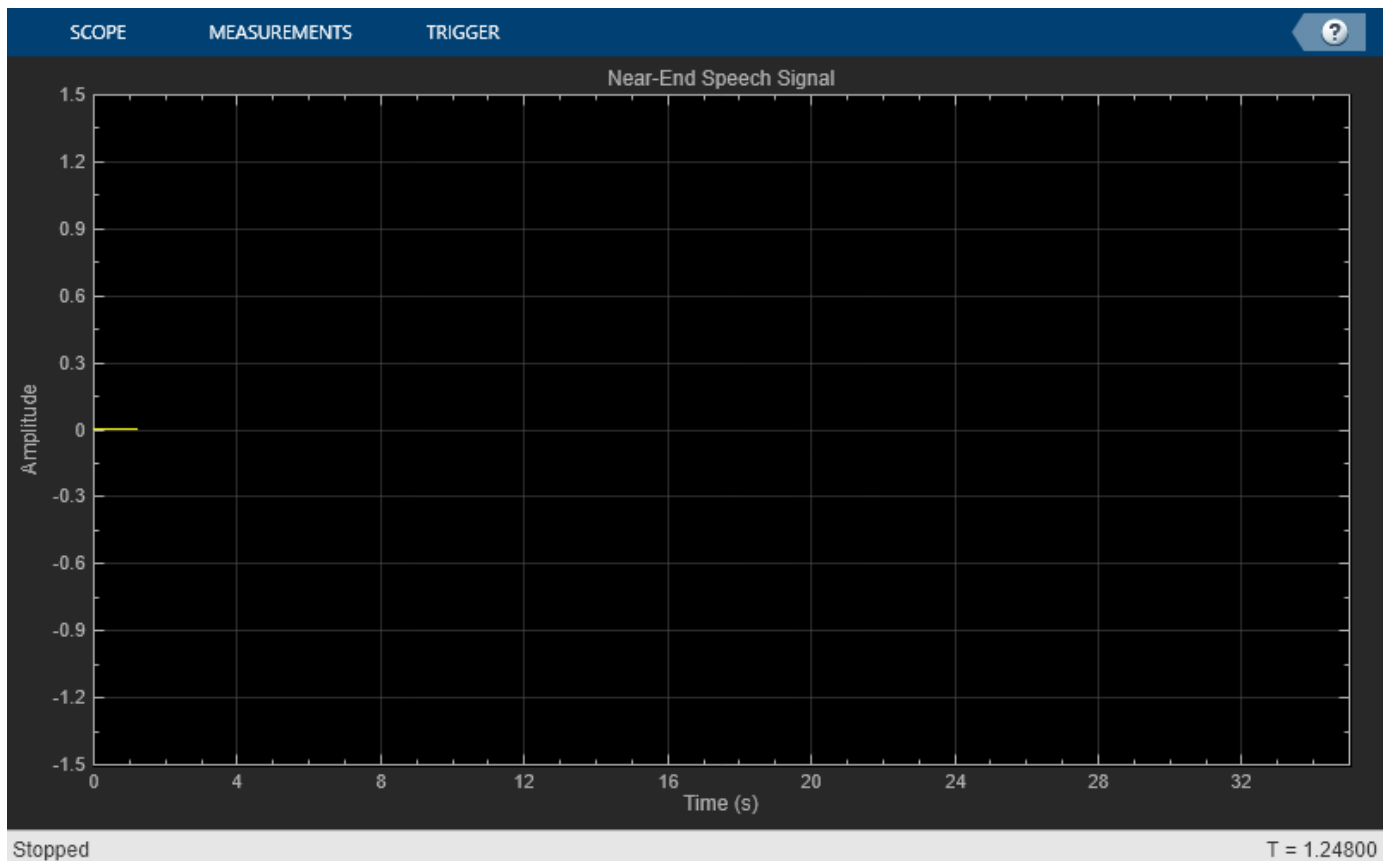
The Near-End Speech Signal

The teleconferencing system's user is typically located near the system's microphone. Here is what a male speech sounds like at the microphone.

```
load nearspeech

player      = audioDeviceWriter('SupportVariableSizeInput', true, ...
                                'BufferSize', 512, 'SampleRate', fs);
nearSpeechSrc = dsp.SignalSource('Signal', v, 'SamplesPerFrame', frameSize);
nearSpeechScope = timescope('SampleRate', fs, 'TimeSpanSource', 'Property', ...
                             'TimeSpan', 35, 'TimeSpanOvverrunAction', 'Scroll', ...
                             'YLimits', [-1.5 1.5], ...
                             'BufferLength', length(v), ...
                             'Title', 'Near-End Speech Signal', ...
                             'ShowGrid', true);

% Stream processing loop
while(~isDone(nearSpeechSrc))
    % Extract the speech samples from the input signal
    nearSpeech = nearSpeechSrc();
    % Send the speech samples to the output audio device
    player(nearSpeech);
    % Plot the signal
    nearSpeechScope(nearSpeech);
end
release(nearSpeechScope);
```



The Far-End Speech Signal

In a teleconferencing system, a voice travels out the loudspeaker, bounces around in the room, and then is picked up by the system's microphone. Listen to what the speech sounds like if it is picked up at the microphone without the near-end speech present.

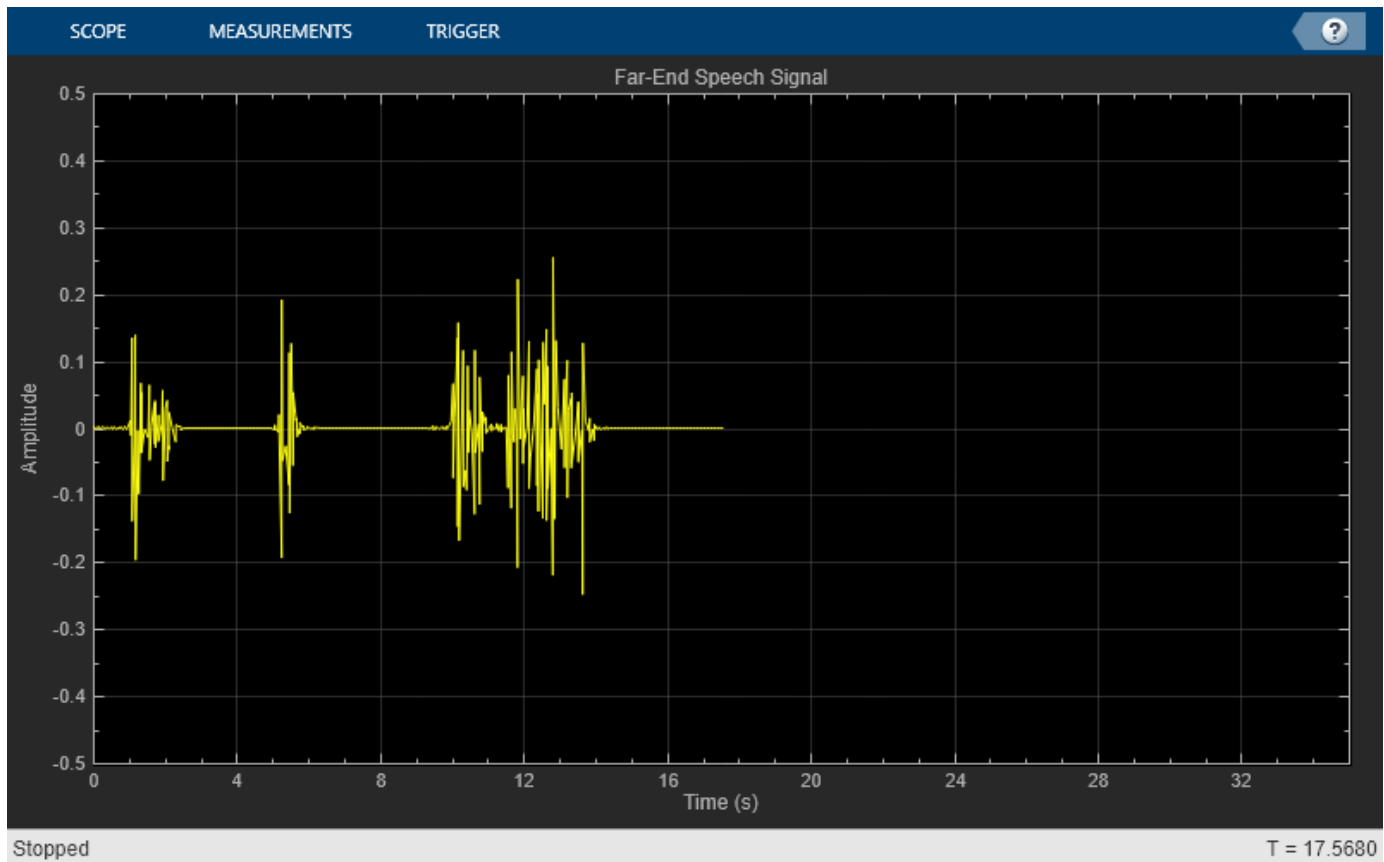
```
load farspeech
farSpeechSrc = dsp.SignalSource('Signal',x,'SamplesPerFrame',frameSize);
farSpeechSink = dsp.SignalSink;
farSpeechScope = timescope('SampleRate', fs, 'TimeSpanSource','Property',...
    'TimeSpan', 35, 'TimeSpanOverrunAction', 'Scroll', ...
    'YLimits', [-0.5 0.5], ...
    'BufferLength', length(x), ...
    'Title', 'Far-End Speech Signal', ...
    'ShowGrid', true);

% Stream processing loop
while(~isDone(farSpeechSrc))
    % Extract the speech samples from the input signal
    farSpeech = farSpeechSrc();
    % Add the room effect to the far-end speech signal
    farSpeechEcho = room(farSpeech);
    % Send the speech samples to the output audio device
    player(farSpeechEcho);
    % Plot the signal
    farSpeechScope(farSpeech);
    % Log the signal for further processing
```

```

        farSpeechSink(farSpeechEcho);
    end
    release(farSpeechScope);

```



The Microphone Signal

The signal at the microphone contains both the near-end speech and the far-end speech that has been echoed throughout the room. The goal of the acoustic echo canceler is to cancel out the far-end speech, such that only the near-end speech is transmitted back to the far-end listener.

```

reset(nearSpeechSrc);
farSpeechEchoSrc = dsp.SignalSource('Signal', farSpeechSink.Buffer, ...
    'SamplesPerFrame', frameSize);
micSink          = dsp.SignalSink;
micScope         = timescope('SampleRate', fs, 'TimeSpanSource', 'Property', ...
    'TimeSpan', 35, 'TimeSpanOverrunAction', 'Scroll', ...
    'YLimits', [-1 1], ...
    'BufferLength', length(x), ...
    'Title', 'Microphone Signal', ...
    'ShowGrid', true);

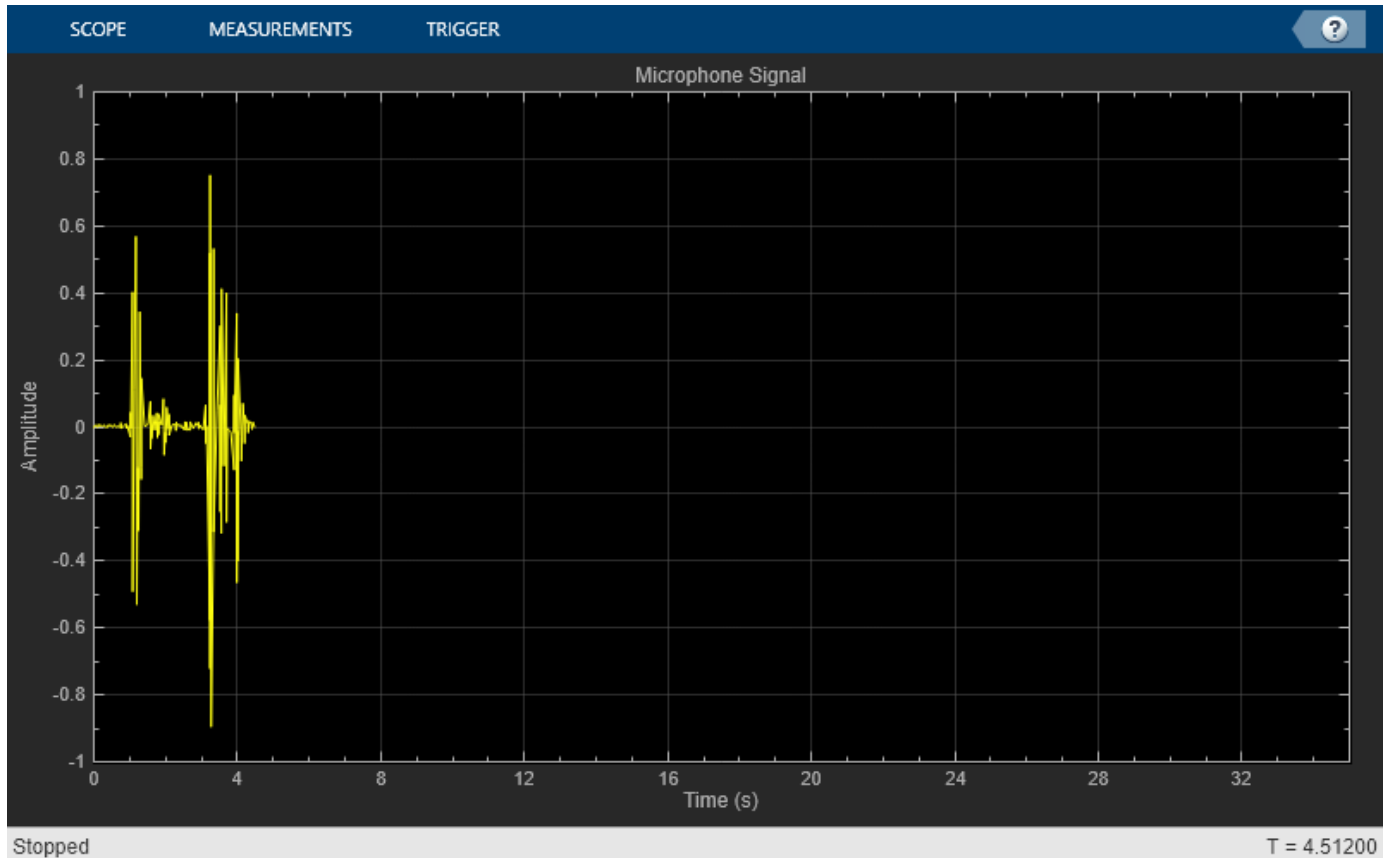
% Stream processing loop
while(~isDone(farSpeechEchoSrc))
    % Microphone signal = echoed far-end + near-end + noise
    micSignal = farSpeechEchoSrc() + nearSpeechSrc() + ...
        0.001*randn(frameSize,1);
    % Send the speech samples to the output audio device

```

```

    player(micSignal);
    % Plot the signal
    micScope(micSignal);
    % Log the signal
    micSink(micSignal);
end
release(micScope);

```



The Frequency-Domain Adaptive Filter (FDAF)

The algorithm in this example is the **Frequency-Domain Adaptive Filter (FDAF)**. This algorithm is very useful when the impulse response of the system to be identified is long. The FDAF uses a fast convolution technique to compute the output signal and filter updates. This computation executes quickly in MATLAB®. It also has fast convergence performance through frequency-bin step size normalization. Pick some initial parameters for the filter and see how well the far-end speech is cancelled in the error signal.

```

% Construct the Frequency-Domain Adaptive Filter
echoCanceller = dsp.FrequencyDomainAdaptiveFilter('Length', 2048, ...
    'StepSize', 0.025, ...
    'InitialPower', 0.01, ...
    'AveragingFactor', 0.98, ...
    'Method', 'Unconstrained FDAF');

AECScope1 = timescope(4, fs, ...
    'LayoutDimensions', [4,1], 'TimeSpanSource', 'Property', ...
    'TimeSpan', 35, 'TimeSpanOverrunAction', 'Scroll', ...

```

```
        'BufferLength', length(x));

AECScopel.ActiveDisplay = 1;
AECScopel.ShowGrid      = true;
AECScopel.YLimits       = [-1.5 1.5];
AECScopel.Title         = 'Near-End Speech Signal';

AECScopel.ActiveDisplay = 2;
AECScopel.ShowGrid      = true;
AECScopel.YLimits       = [-1.5 1.5];
AECScopel.Title         = 'Microphone Signal';

AECScopel.ActiveDisplay = 3;
AECScopel.ShowGrid      = true;
AECScopel.YLimits       = [-1.5 1.5];
AECScopel.Title         = 'Output of Acoustic Echo Celler mu=0.025';

AECScopel.ActiveDisplay = 4;
AECScopel.ShowGrid      = true;
AECScopel.YLimits       = [0 50];
AECScopel.YLabel        = 'ERLE (dB)';
AECScopel.Title         = 'Echo Return Loss Enhancement mu=0.025';

% Near-end speech signal
release(nearSpeechSrc);
nearSpeechSrc.SamplesPerFrame = frameSize;

% Far-end speech signal
release(farSpeechSrc);
farSpeechSrc.SamplesPerFrame = frameSize;

% Far-end speech signal echoed by the room
release(farSpeechEchoSrc);
farSpeechEchoSrc.SamplesPerFrame = frameSize;
```

Echo Return Loss Enhancement (ERLE)

Since you have access to both the near-end and far-end speech signals, you can compute the **echo return loss enhancement (ERLE)**, which is a smoothed measure of the amount (in dB) that the echo has been attenuated. From the plot, observe that you achieved about a 35 dB ERLE at the end of the convergence period.

```
diffAverager = dsp.FIRFilter('Numerator', ones(1,1024));
farEchoAverager = clone(diffAverager);
setfilter(FVT,diffAverager);

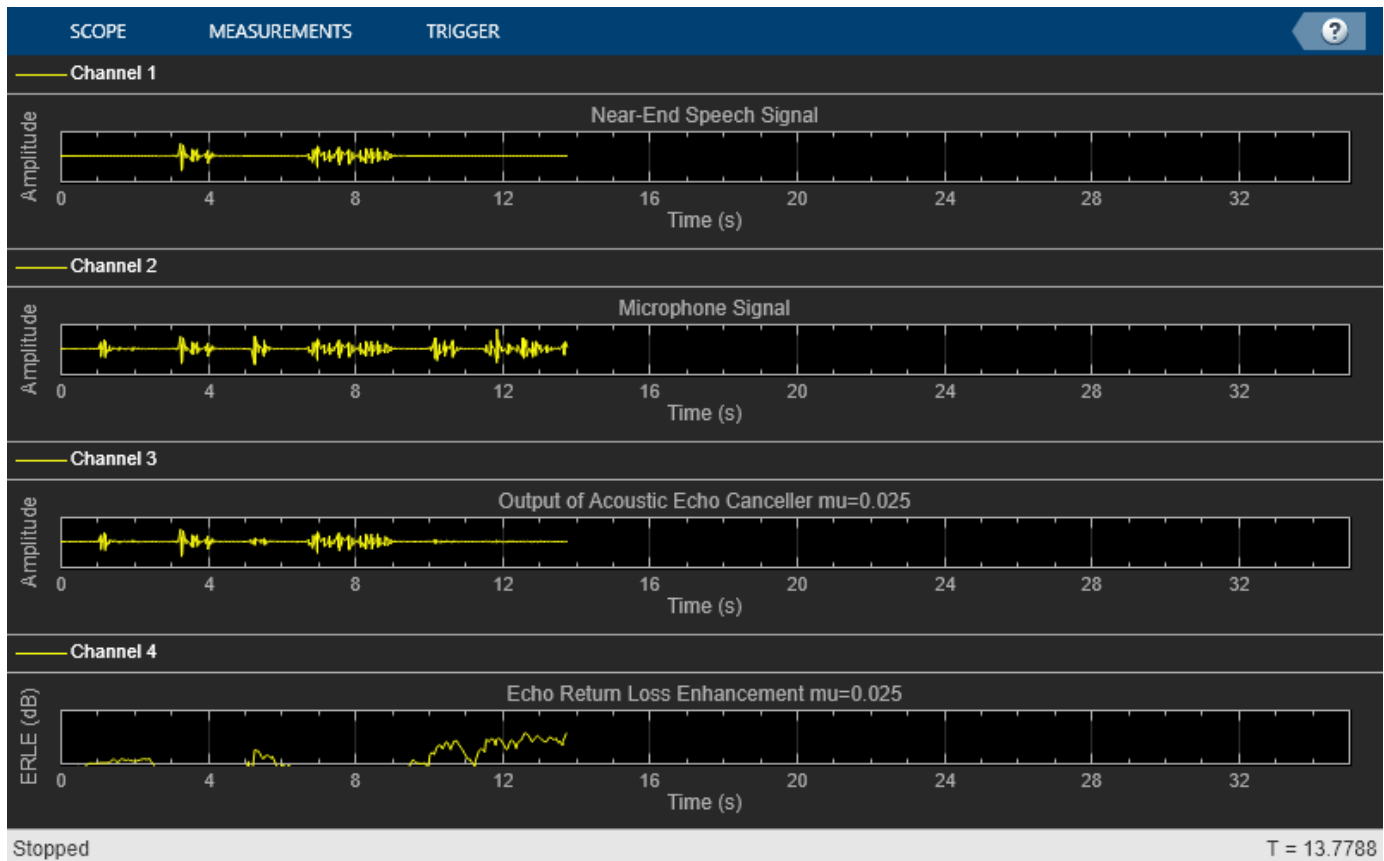
micSrc = dsp.SignalSource('Signal', micSink.Buffer, ...
    'SamplesPerFrame', frameSize);

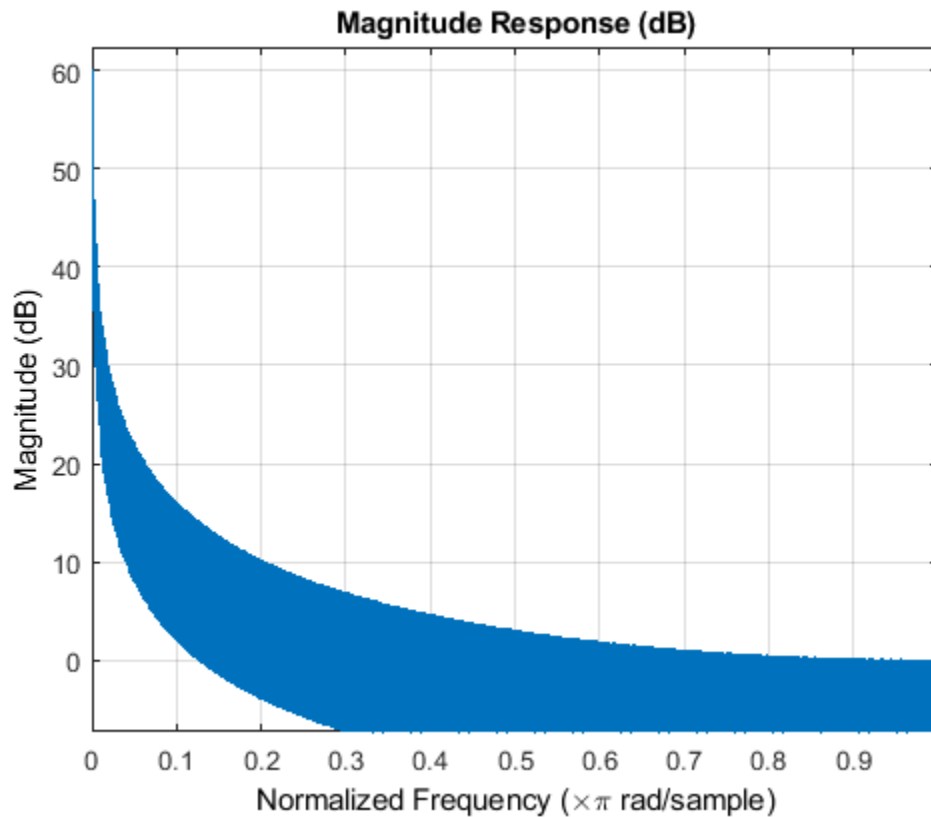
% Stream processing loop - adaptive filter step size = 0.025
while(~isDone(nearSpeechSrc))
    nearSpeech = nearSpeechSrc();
    farSpeech = farSpeechSrc();
    farSpeechEcho = farSpeechEchoSrc();
    micSignal = micSrc();
    % Apply FDAF
    [y,e] = echoCanceller(farSpeech, micSignal);
    % Send the speech samples to the output audio device
```

```

player(e);
% Compute ERLE
erle = diffAverager((e-nearSpeech).^2)./ farEchoAverager(farSpeechEcho.^2);
erledB = -10*log10(erle);
% Plot near-end, far-end, microphone, AEC output and ERLE
AECScope1(nearSpeech, micSignal, e, erledB);
end
release(AECScope1);

```





Effects of Different Step Size Values

To get faster convergence, you can try using a larger step size value. However, this increase causes another effect: the adaptive filter is "misadjusted" while the near-end speaker is talking. Listen to what happens when you choose a step size that is 60% larger than before.

```
% Change the step size value in FDAF
reset(echoCanceller);
echoCanceller.StepSize = 0.04;

AECScope2 = clone(AECScope1);
AECScope2.ActiveDisplay = 3;
AECScope2.Title = 'Output of Acoustic Echo Canceller mu=0.04';
AECScope2.ActiveDisplay = 4;
AECScope2.Title = 'Echo Return Loss Enhancement mu=0.04';

reset(nearSpeechSrc);
reset(farSpeechSrc);
reset(farSpeechEchoSrc);
reset(micSrc);
reset(diffAverager);
reset(farEchoAverager);

% Stream processing loop - adaptive filter step size = 0.04
while(~isDone(nearSpeechSrc))
    nearSpeech = nearSpeechSrc();
    farSpeech = farSpeechSrc();
```

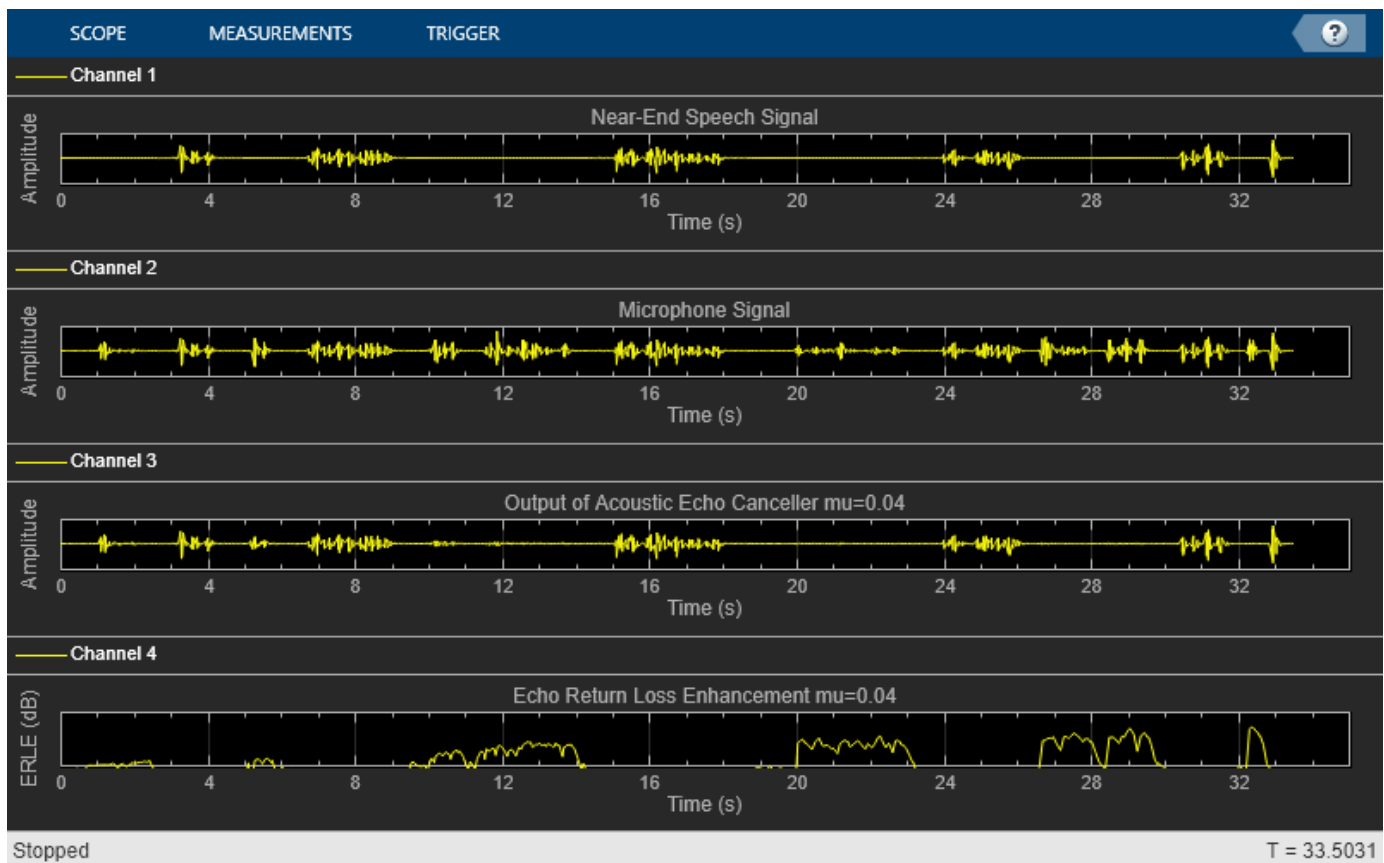


```

farSpeechEcho = farSpeechEchoSrc();
micSignal = micSrc();
% Apply FDAF
[y,e] = echoCanceller(farSpeech, micSignal);
% Send the speech samples to the output audio device
player(e);
% Compute ERLE
erle = diffAverager((e-nearSpeech).^2)./ farEchoAverager(farSpeechEcho.^2);
erleDB = -10*log10(erle);
% Plot near-end, far-end, microphone, AEC output and ERLE
AECScope2(nearSpeech, micSignal, e, erleDB);
end

release(nearSpeechSrc);
release(farSpeechSrc);
release(farSpeechEchoSrc);
release(micSrc);
release(diffAverager);
release(farEchoAverager);
release(echoCanceller);
release(AECScope2);

```



Echo Return Loss Enhancement Comparison

With a larger step size, the ERLE performance is not as good due to the misadjustment introduced by the near-end speech. To deal with this performance difficulty, acoustic echo cancelers include a detection scheme to tell when near-end speech is present and lower the step size value over these

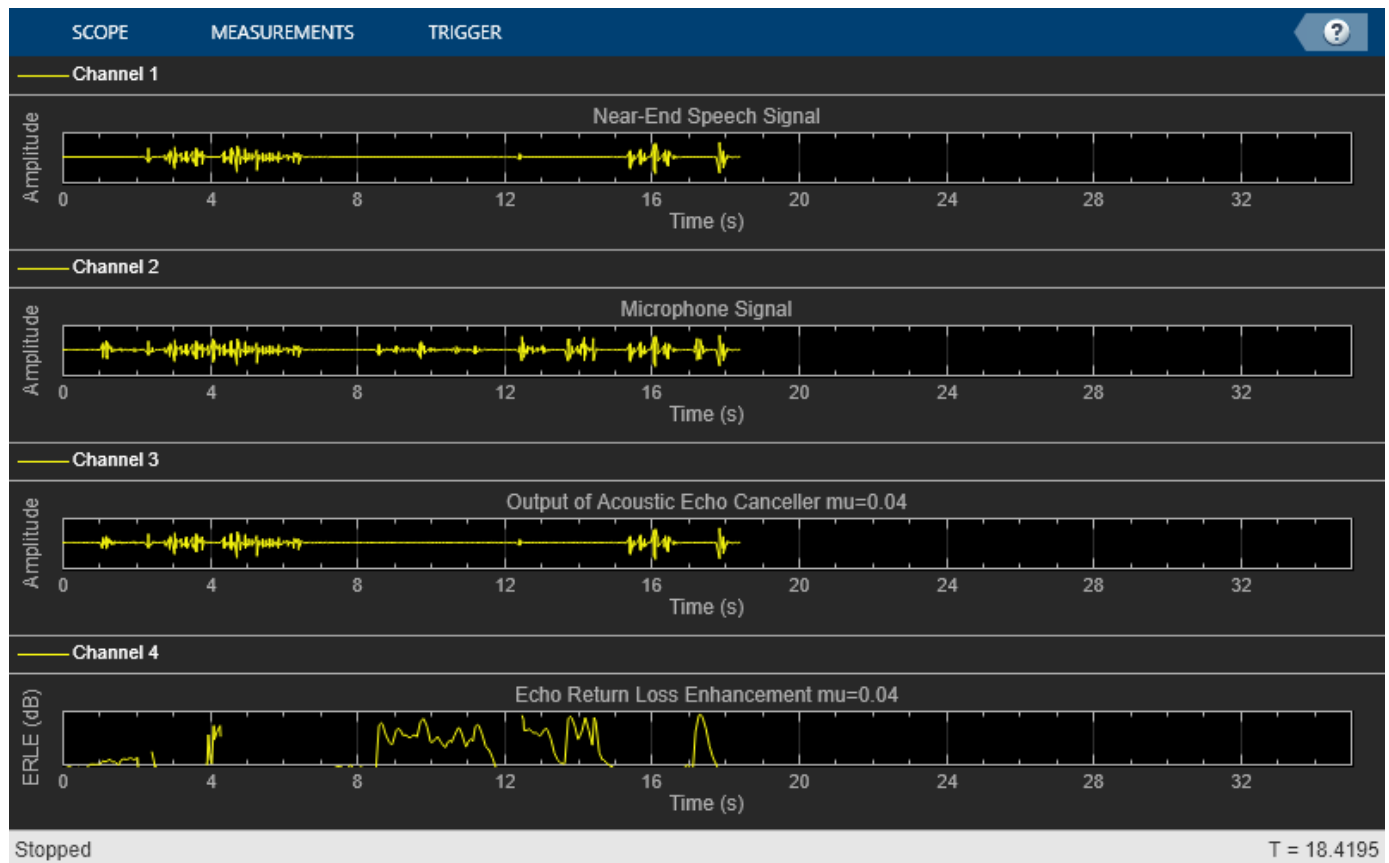
periods. Without such detection schemes, the performance of the system with the larger step size is not as good as the former, as can be seen from the ERLE plots.

Latency Reduction Using Partitioning

Traditional FDAF is numerically more efficient than time-domain adaptive filtering for long impulse responses, but it imposes high latency, because the input frame size must be a multiple of the specified filter length. This can be unacceptable for many real-world applications. Latency may be reduced by using partitioned FDAF, which partitions the filter impulse response into shorter segments, applies FDAF to each segment, and then combines the intermediate results. The frame size in that case must be a multiple of the partition (block) length, thereby greatly reducing the latency for long impulse responses.

```
% Reduce the frame size from 2048 to 256
frameSize = 256;
nearSpeechSrc.SamplesPerFrame    = frameSize;
farSpeechSrc.SamplesPerFrame     = frameSize;
farSpeechEchoSrc.SamplesPerFrame = frameSize;
micSrc.SamplesPerFrame          = frameSize;
% Switch the echo canceller to Partitioned constrained FDAF
echoCanceller.Method            = 'Partitioned constrained FDAF';
% Set the block length to frameSize
echoCanceller.BlockLength = frameSize;

% Stream processing loop
while(~isDone(nearSpeechSrc))
    nearSpeech = nearSpeechSrc();
    farSpeech = farSpeechSrc();
    farSpeechEcho = farSpeechEchoSrc();
    micSignal = micSrc();
    % Apply FDAF
    [y,e] = echoCanceller(farSpeech, micSignal);
    % Send the speech samples to the output audio device
    player(e);
    % Compute ERLE
    erle = diffAverager((e-nearSpeech).^2)./ farEchoAverager(farSpeechEcho.^2);
    erledB = -10*log10(erle);
    % Plot near-end, far-end, microphone, AEC output and ERLE
    AECScope2(nearSpeech, micSignal, e, erledB);
end
```



Active Noise Control Using a Filtered-X LMS FIR Adaptive Filter

This example shows how to apply adaptive filters to the attenuation of acoustic noise via active noise control.

Active Noise Control

In active noise control, one attempts to reduce the volume of an unwanted noise propagating through the air using an electro-acoustic system using measurement sensors such as microphones and output actuators such as loudspeakers. The noise signal usually comes from some device, such as a rotating machine, so that it is possible to measure the noise near its source. The goal of the active noise control system is to produce an "anti-noise" that attenuates the unwanted noise in a desired quiet region using an adaptive filter. This problem differs from traditional adaptive noise cancellation in that: - The desired response signal cannot be directly measured; only the attenuated signal is available. - The active noise control system must take into account the secondary loudspeaker-to-microphone error path in its adaptation.

For more implementation details on active noise control tasks, see S.M. Kuo and D.R. Morgan, "Active Noise Control Systems: Algorithms and DSP Implementations", Wiley-Interscience, New York, 1996.

The Secondary Propagation Path

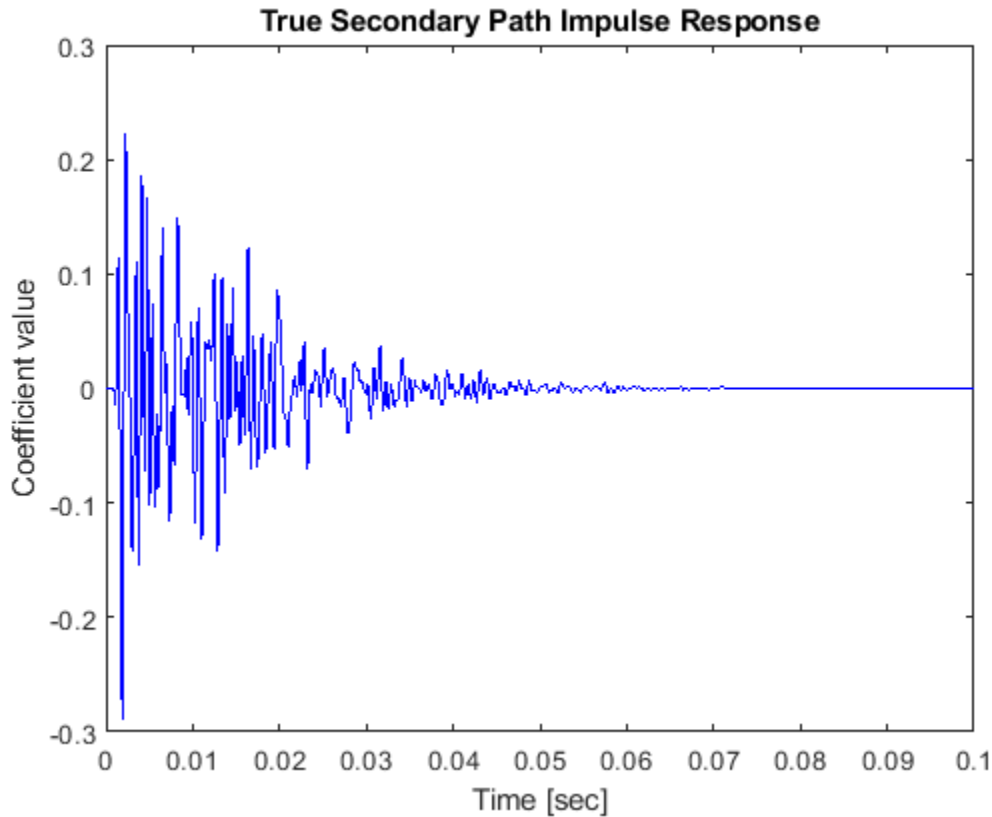
The secondary propagation path is the path the anti-noise takes from the output loudspeaker to the error microphone within the quiet zone. The following commands generate a loudspeaker-to-error microphone impulse response that is bandlimited to the range 160 - 2000 Hz and with a filter length of 0.1 seconds. For this active noise control task, we shall use a sampling frequency of 8000 Hz.

```
Fs      = 8e3; % 8 kHz
N       = 800; % 800 samples@8 kHz = 0.1 seconds
Flow    = 160; % Lower band-edge: 160 Hz
Fhigh   = 2000; % Upper band-edge: 2000 Hz
delayS  = 7;
Ast     = 20; % 20 dB stopband attenuation
Nfilt   = 8; % Filter order

% Design bandpass filter to generate bandlimited impulse response
filtSpecs = fdesign.bandpass('N,Fst1,Fst2,Ast',Nfilt,Flow,Fhigh,Ast,Fs);
bandpass = design(filtSpecs,'cheby2','FilterStructure','df2tsos', ...
    'SystemObject',true);

% Filter noise to generate impulse response
secondaryPathCoeffsActual = bandpass([zeros(delayS,1); ...
    log(0.99*rand(N-delayS,1)+0.01).* ...
    sign(randn(N-delayS,1)).*exp(-0.01*(1:N-delayS))]);
secondaryPathCoeffsActual = ...
    secondaryPathCoeffsActual/norm(secondaryPathCoeffsActual);

t = (1:N)/Fs;
plot(t,secondaryPathCoeffsActual,'b');
xlabel('Time [sec]');
ylabel('Coefficient value');
title('True Secondary Path Impulse Response');
```



Estimating the Secondary Propagation Path

The first task in active noise control is to estimate the impulse response of the secondary propagation path. This step is usually performed prior to noise control using a synthetic random signal played through the output loudspeaker while the unwanted noise is not present. The following commands generate 3.75 seconds of this random noise as well as the measured signal at the error microphone.

```
ntrS = 30000;
randomSignal = randn(ntrS,1); % Synthetic random signal to be played
secondaryPathGenerator = dsp.FIRFilter('Numerator',secondaryPathCoeffsActual. ');
secondaryPathMeasured = secondaryPathGenerator(randomSignal) + ... % random signal propagated th
    0.01*randn(ntrS,1); % measurement noise at the microphone
```

Designing the Secondary Propagation Path Estimate

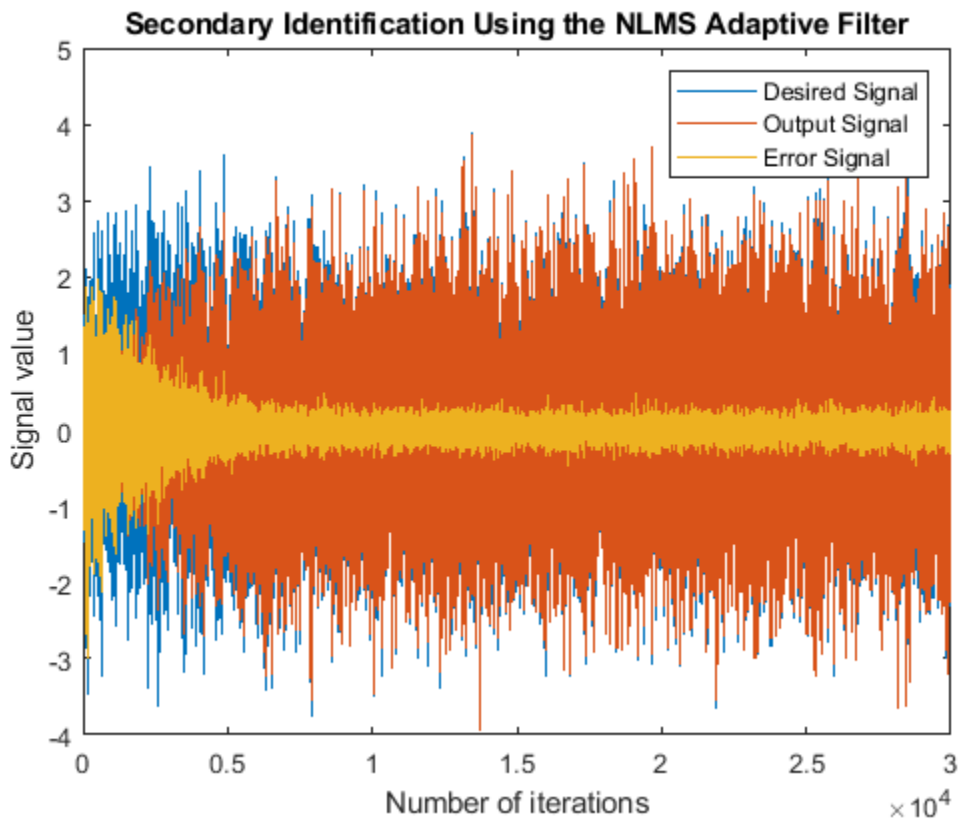
Typically, the length of the secondary path filter estimate is not as long as the actual secondary path and need not be for adequate control in most cases. We shall use a secondary path filter length of 250 taps, corresponding to an impulse response length of 31 ms. While any adaptive FIR filtering algorithm could be used for this purpose, the normalized LMS algorithm is often used due to its simplicity and robustness. Plots of the output and error signals show that the algorithm converges after about 10000 iterations.

```
M = 250;
muS = 0.1;
secondaryPathEstimator = dsp.LMSFilter('Method','Normalized LMS','StepSize', muS, ...
    'Length', M);
[yS,eS,SecondaryPathCoeffsEst] = secondaryPathEstimator(randomSignal,secondaryPathMeasured);
```

```

n = 1:ntrS;
figure, plot(n,secondaryPathMeasured,n,yS,n,eS);
xlabel('Number of iterations');
ylabel('Signal value');
title('Secondary Identification Using the NLMS Adaptive Filter');
legend('Desired Signal','Output Signal','Error Signal');

```



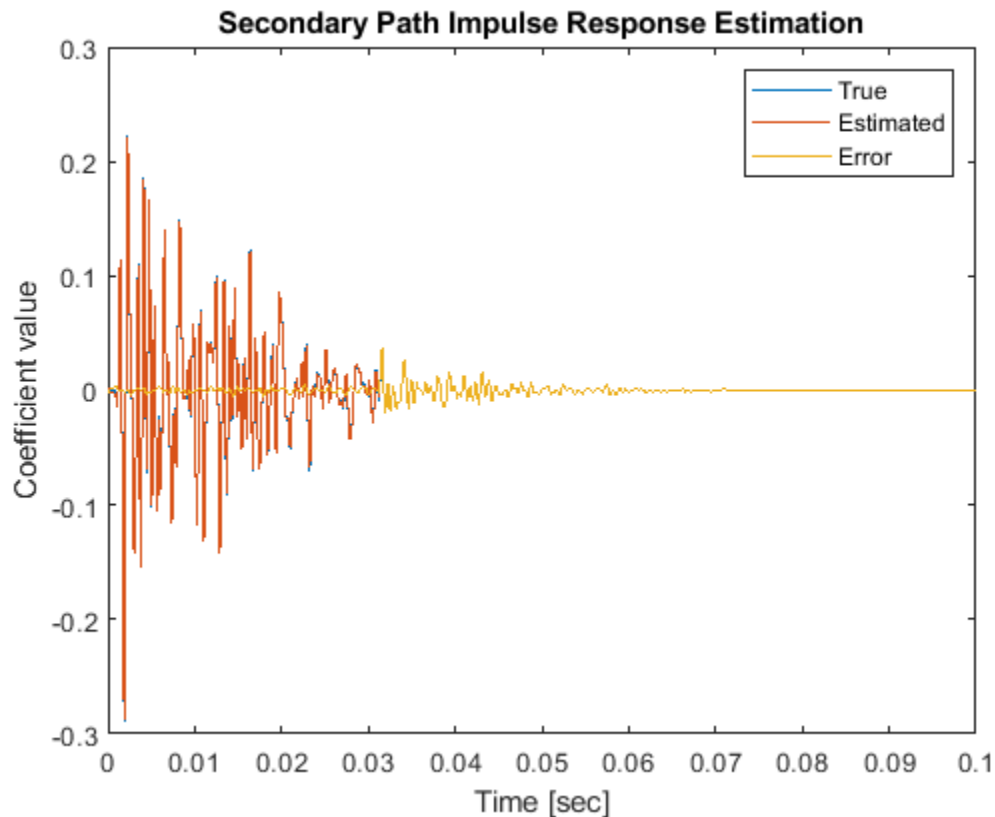
Accuracy of the Secondary Path Estimate

How accurate is the secondary path impulse response estimate? This plot shows the coefficients of both the true and estimated path. Only the tail of the true impulse response is not estimated accurately. This residual error does not significantly harm the performance of the active noise control system during its operation in the chosen task.

```

figure, plot(t,secondaryPathCoeffsActual, ...
    t(1:M),SecondaryPathCoeffsEst, ...
    t,[secondaryPathCoeffsActual(1:M)-SecondaryPathCoeffsEst(1:M); secondaryPathCoeffsActual(M+1:
    xlabel('Time [sec]');
    ylabel('Coefficient value');
    title('Secondary Path Impulse Response Estimation');
    legend('True','Estimated','Error');

```



The Primary Propagation Path

The propagation path of the noise to be cancelled can also be characterized by a linear filter. The following commands generate an input-to-error microphone impulse response that is bandlimited to the range 200 - 800 Hz and has a filter length of 0.1 seconds.

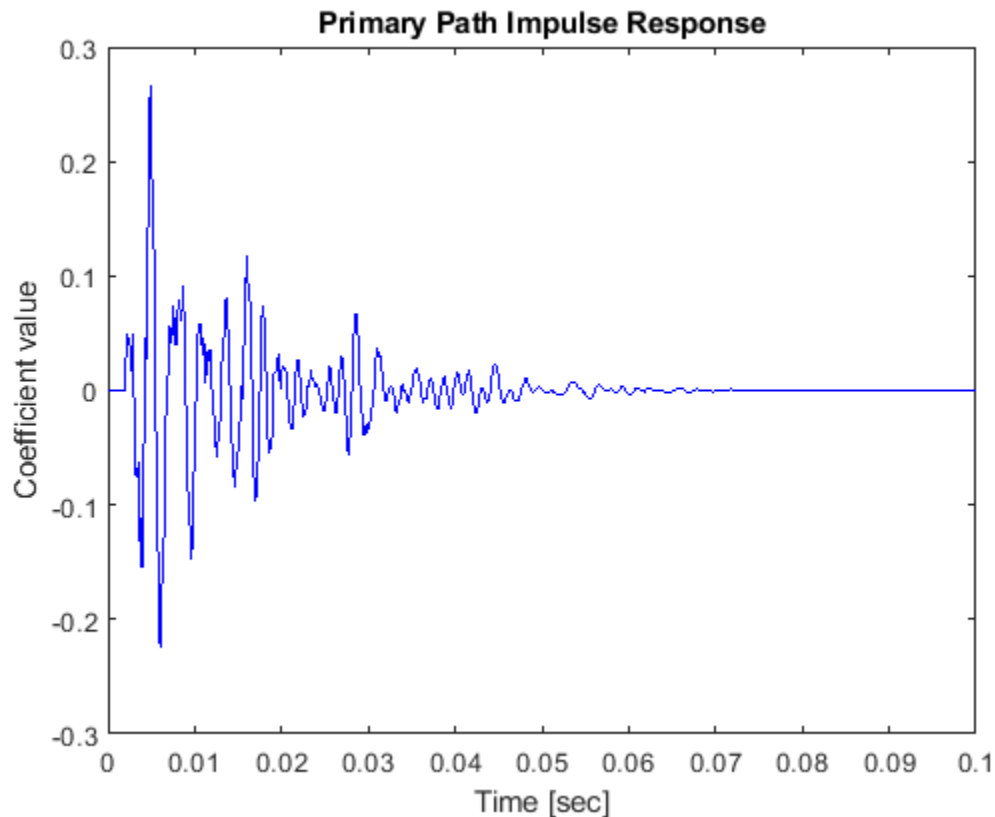
```

delayW = 15;
Flow   = 200; % Lower band-edge: 200 Hz
Fhigh  = 800; % Upper band-edge: 800 Hz
Ast    = 20;  % 20 dB stopband attenuation
Nfilt  = 10;  % Filter order

% Design bandpass filter to generate bandlimited impulse response
filtSpecs2 = fdesign.bandpass('N,Fst1,Fst2,Ast',Nfilt,Flow,Fhigh,Ast,Fs);
bandpass2 = design(filtSpecs2,'cheby2','FilterStructure','df2tsos', ...
    'SystemObject',true);

% Filter noise to generate impulse response
primaryPathCoeffs = bandpass2([zeros(delayW,1); log(0.99*rand(N-delayW,1)+0.01).* ...
    sign(randn(N-delayW,1)).*exp(-0.01*(1:N-delayW'))]);
primaryPathCoeffs = primaryPathCoeffs/norm(primaryPathCoeffs);

figure, plot(t,primaryPathCoeffs,'b');
xlabel('Time [sec]');
ylabel('Coefficient value');
title('Primary Path Impulse Response');
```



The Noise to Be Cancelled

Typical active noise control applications involve the sounds of rotating machinery due to their annoying characteristics. Here, we synthetically generate noise that might come from a typical electric motor.

Initialization of Active Noise Control

The most popular adaptive algorithm for active noise control is the filtered-X LMS algorithm. This algorithm uses the secondary path estimate to calculate an output signal whose contribution at the error sensor destructively interferes with the undesired noise. The reference signal is a noisy version of the undesired sound measured near its source. We shall use a controller filter length of about 44 ms and a step size of 0.0001 for these signal statistics.

```
% FIR Filter to be used to model primary propagation path
primaryPathGenerator = dsp.FIRFilter('Numerator',primaryPathCoeffs.');
```

```
% Filtered-X LMS adaptive filter to control the noise
L = 350;
muW = 0.0001;
noiseController = dsp.FilteredXLMSTFilter('Length',L,'StepSize',muW, ...
    'SecondaryPathCoefficients',SecondaryPathCoeffsEst);
```

```
% Sine wave generator to synthetically create the noise
A = [.01 .01 .02 .2 .3 .4 .3 .2 .1 .07 .02 .01];
La = length(A);
F0 = 60;
```



```

k = 1:La;
F = F0*k;
phase = rand(1,La); % Random initial phase
sine = audioOscillator('NumTones', La, 'Amplitude',A,'Frequency',F, ...
    'PhaseOffset',phase,'SamplesPerFrame',512,'SampleRate',Fs);

% Audio player to play noise before and after cancellation
player = audioDeviceWriter('SampleRate',Fs);

% Spectrum analyzer to show original and attenuated noise
scope = dsp.SpectrumAnalyzer('SampleRate',Fs,'OverlapPercent',80, ...
    'SpectralAverages',20,'PlotAsTwoSidedSpectrum',false, ...
    'ShowLegend',true, ...
    'ChannelNames', {'Original noisy signal', 'Attenuated noise'});

```

Simulation of Active Noise Control Using the Filtered-X LMS Algorithm

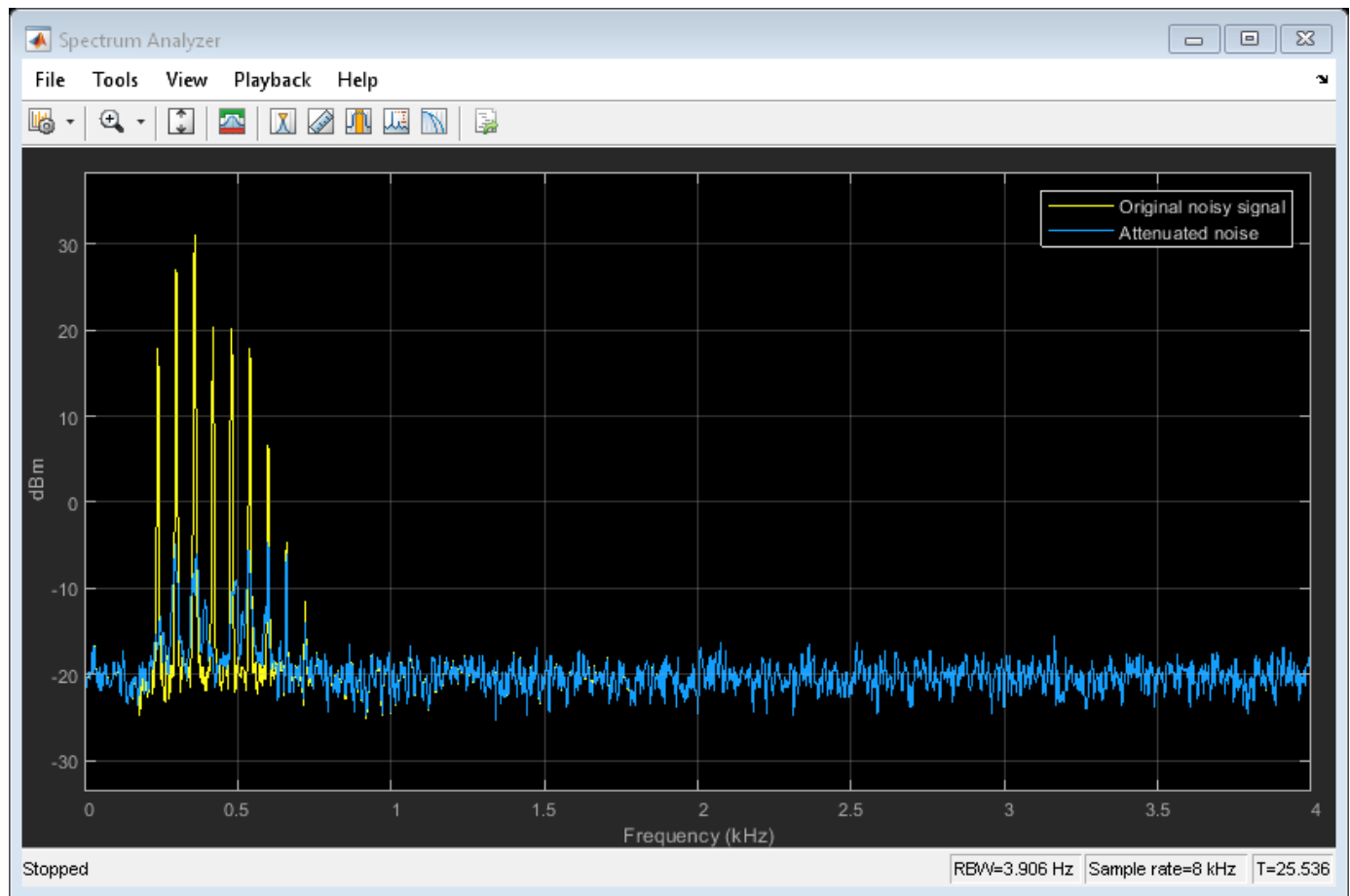
Here we simulate the active noise control system. To emphasize the difference we run the system with no active noise control for the first 200 iterations. Listening to its sound at the error microphone before cancellation, it has the characteristic industrial "whine" of such motors.

Once the adaptive filter is enabled, the resulting algorithm converges after about 5 (simulated) seconds of adaptation. Comparing the spectrum of the residual error signal with that of the original noise signal, we see that most of the periodic components have been attenuated considerably. The steady-state cancellation performance may not be uniform across all frequencies, however. Such is often the case for real-world systems applied to active noise control tasks. Listening to the error signal, the annoying "whine" is reduced considerably.

```

for m = 1:400
    % Generate synthetic noise by adding sine waves with random phase
    x = sine();
    d = primaryPathGenerator(x) + ... % Propagate noise through primary path
        0.1*randn(size(x)); % Add measurement noise
    if m <= 200
        % No noise control for first 200 iterations
        e = d;
    else
        % Enable active noise control after 200 iterations
        xhat = x + 0.1*randn(size(x));
        [y,e] = noiseController(xhat,d);
    end
    player(e); % Play noise signal
    scope([d,e]); % Show spectrum of original (Channel 1)
                % and attenuated noise (Channel 2)
end
release(player); % Release audio device
release(scope); % Release spectrum analyzer

```



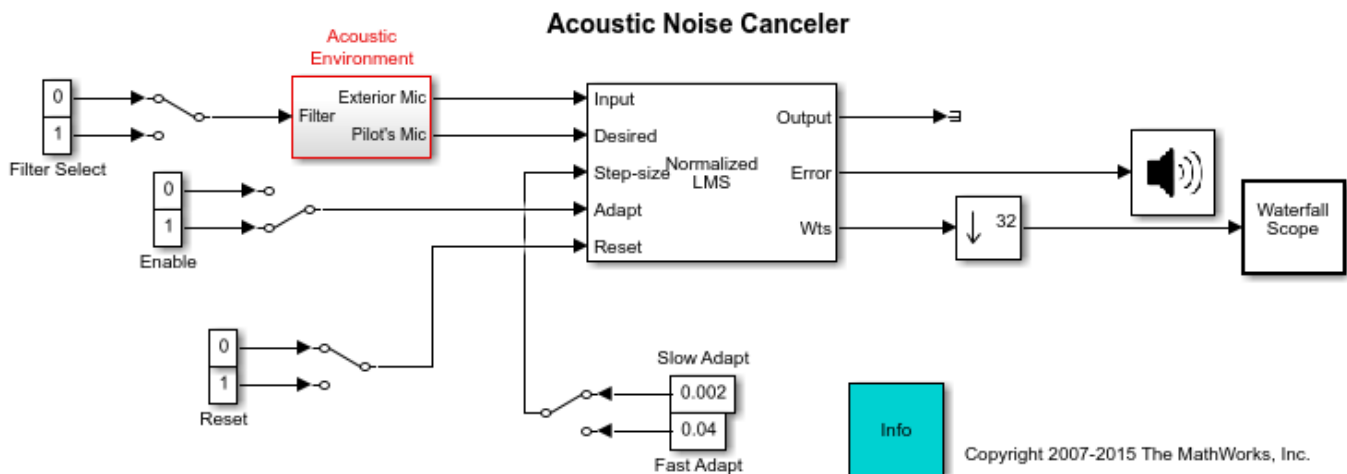
Acoustic Noise Cancellation Using LMS

This example shows how to use the Least Mean Square (LMS) algorithm to subtract noise from an input signal. The LMS adaptive filter uses the reference signal on the **Input** port and the desired signal on the **Desired** port to automatically match the filter response. As it converges to the correct filter model, the filtered noise is subtracted and the error signal should contain only the original signal.

Exploring the Example

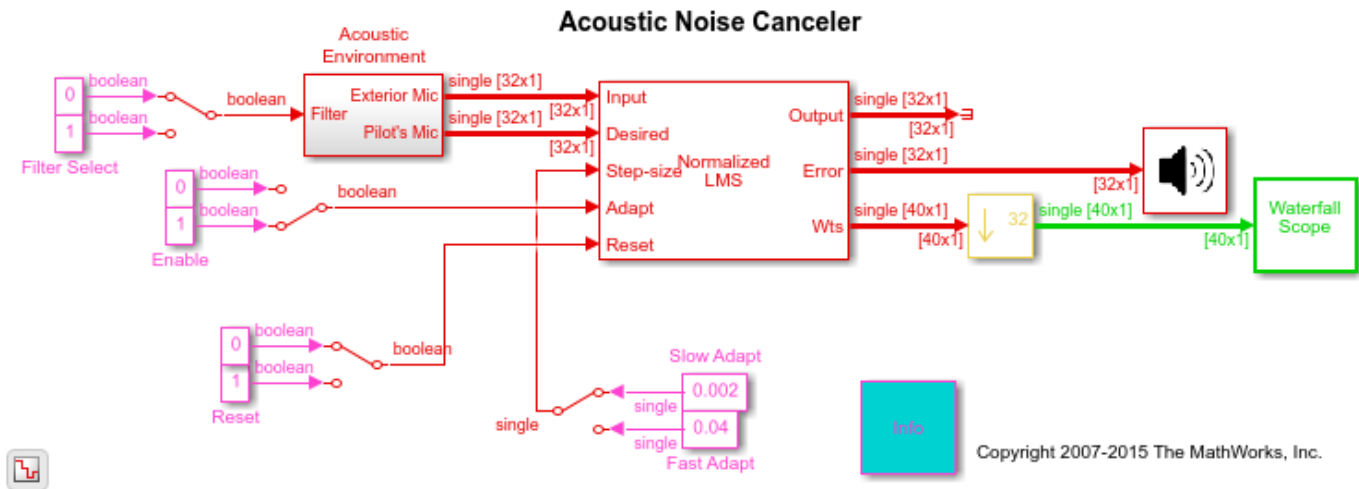
In the model, the signal output at the upper port of the Acoustic Environment subsystem is white noise. The signal output at the lower port is composed of colored noise and a signal from a WAV file. This example model uses an adaptive filter to remove the noise from the signal output at the lower port. When you run the simulation, you hear both noise and a person playing the drums. Over time, the adaptive filter in the model filters out the noise so you only hear the drums.

Acoustic Noise Cancellation Model



Utilizing Your Audio Device

Run the model to listen to the audio signal in real time. The stop time is set to infinity. This allows you to interact with the model while it is runs. For example, you can change the filter or alternate from slow adaptation to fast adaptation (and vice versa), and get a sense of the real-time audio processing behavior under these conditions.



Color Codes of the Blocks

Notice the colors of the blocks in the model. These are sample time colors that indicate how fast a block executes. Here, the fastest discrete sample time is red, and the second fastest discrete sample time is green. You can see that the color changes from red to green after down-sampling by 32 (in the Downsample block before the Waterfall Scope block). Further information on displaying sample time colors can be found in the Simulink® documentation.

Waterfall Scope

The Waterfall window displays the behavior of the adaptive filter's filter coefficients. It displays multiple vectors of data at one time. These vectors represent the values of the filter's coefficients of a normalized LMS adaptive filter, and are the input data at consecutive sample times. The data is displayed in a three-dimensional axis in the Waterfall window. By default, the x-axis represents amplitude, the y-axis represents samples, and the z-axis represents time. The Waterfall window has toolbar buttons that enable you to zoom in on displayed data, suspend data capture, freeze the scope's display, save the scope position, and export data to the workspace.

Acoustic Environment Subsystem

You can see the details of the Acoustic Environment subsystem by double clicking on that block. Gaussian noise is used to create the signal sent to the Exterior Mic output port. If the input to the Filter port changes from 0 to 1, the Digital Filter block changes from a lowpass filter to a bandpass filter. The filtered noise output from the Digital Filter block is added to the signal coming from a WAV-file to produce the signal sent to the Pilot's Mic output port.

References

[1] Haykin, Simon S. Adaptive Filter Theory. 3rd ed, Prentice Hall, 1996.

Delay-Based Audio Effects

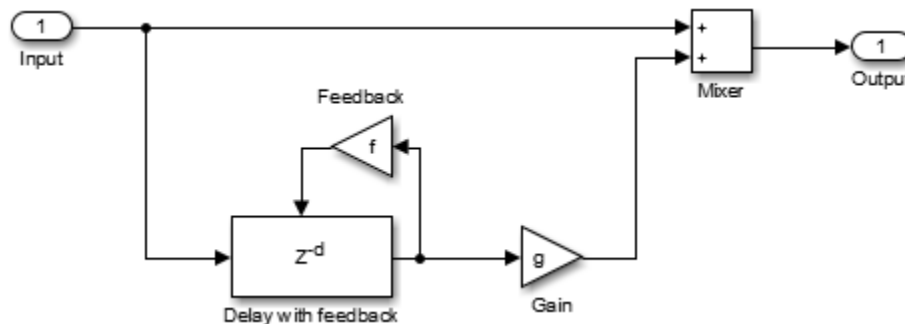
This example shows how to design and use three audio effects that are based on varying delay: echo, chorus and flanger. The example also shows how the algorithms, developed in MATLAB, can be easily ported to Simulink.

Introduction

Audio effects can be generated by adding a processed ('wet') signal to the original ('dry') audio signal. A simple effect, echo, adds a delayed version of the signal to the original. More complex effects, like chorus and flanger, modulate the delayed version of the signal.

Echo

You can model the echo effect by delaying the audio signal and adding it back. Feedback is often added to the delay line to give a fading effect. The echo effect is implemented in the `audioexample.Echo` class. The block diagram shows a high-level implementation of an echo effect.

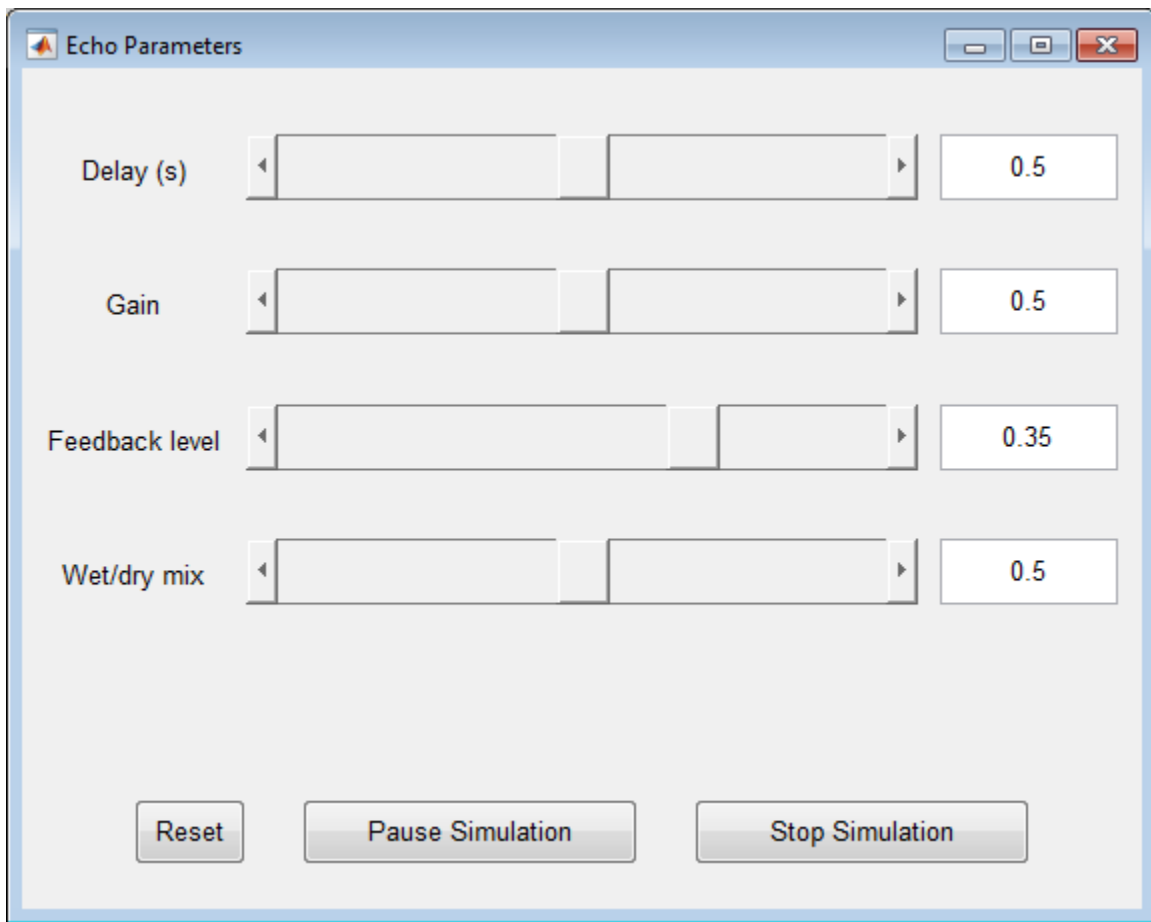


The echo effect example has four tunable parameters that can be modified while the simulation is running:

- Delay - Delay applied to audio signal, in seconds
- Gain - Linear gain of the delayed audio
- FeedbackLevel - Feedback gain applied to delay line
- WetDryMix - Ratio of wet signal added to dry signal

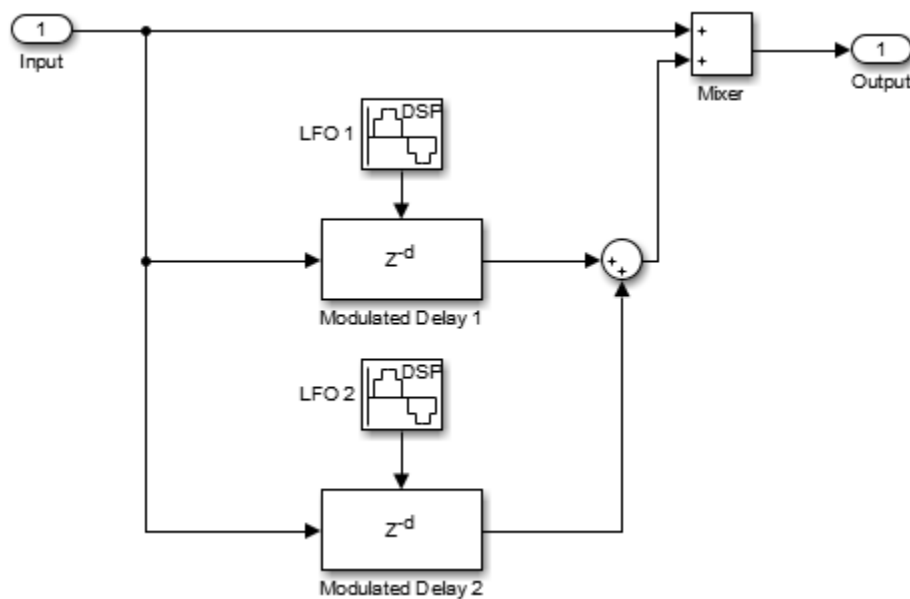
You can try out `audioexample.Echo` by running `audioDelayEffectsExampleApp` with 'echo' as input. The example reads an audio signal from a file, applies the echo effect, and then plays the processed signal through your audio output device. It also launches a UI that allows you to tune the parameters of the echo effect. You can pass an additional argument that determines duration to play the audio.

```
duration = 30; % in seconds
audioDelayEffectsExampleApp('echo',duration);
```



Chorus

The chorus effect usually has multiple independent delays, each modulated by a low-frequency oscillator. `audioexample.Chorus` implements this effect. The block diagram shows a high-level implementation of a chorus effect.

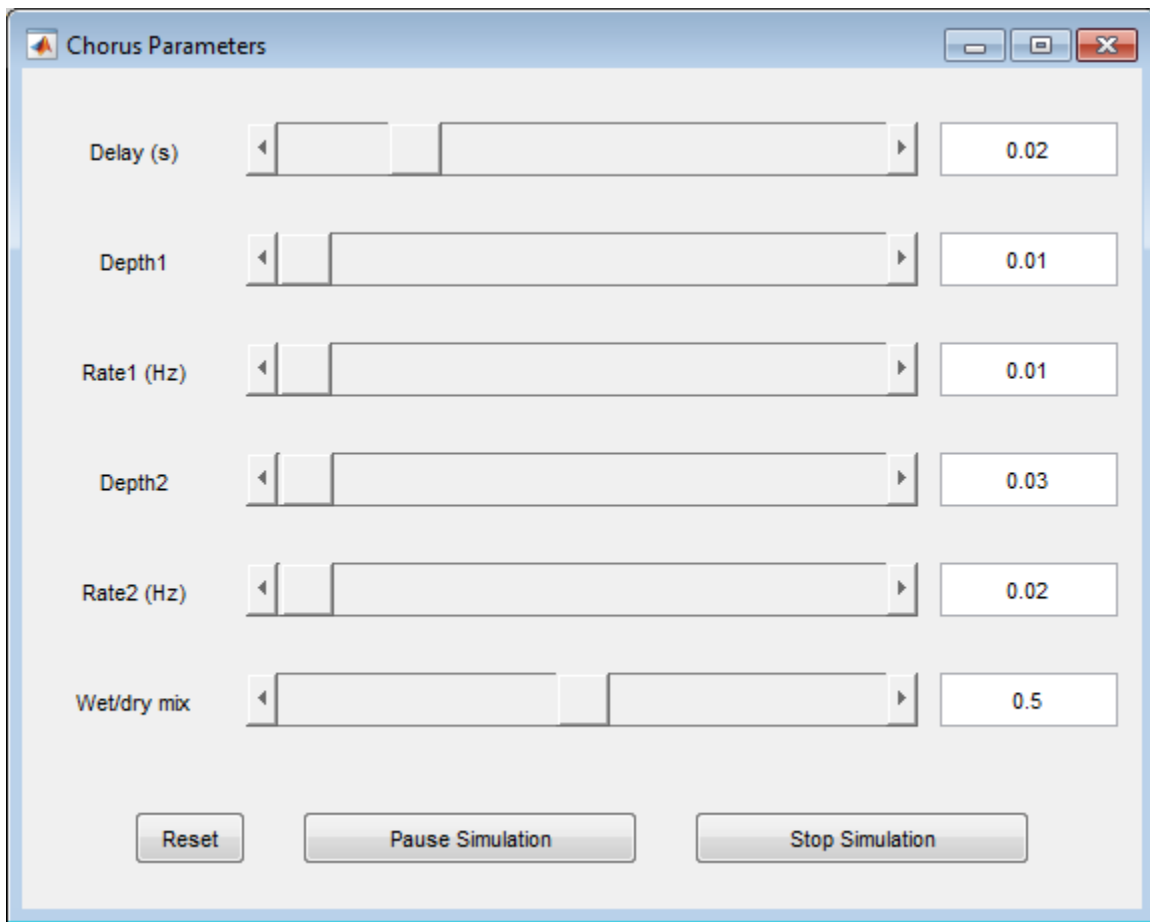


The chorus effect example has six tunable parameters that can be modified while the simulation is running:

- Delay - Base delay applied to audio signal, in seconds
- Depth 1 - Amplitude of modulator applied to first delay branch
- Rate 1 - Frequency of modulator applied to first delay branch, in Hz
- Depth 2 - Amplitude of modulator applied to second delay branch
- Rate 2 - Frequency of modulator applied to second delay branch, in Hz
- WetDryMix - Ratio of wet signal added to dry signal

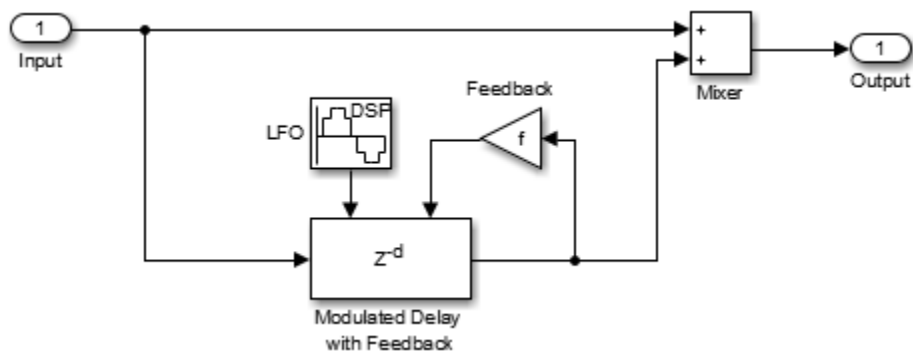
You can try out `audioexample.Chorus` by running `audioDelayEffectsExampleApp` with `'chorus'` as input. The example reads an audio signal from a file, applies the chorus effect, then plays the processed signal through your audio output device. It also launches a UI that allows you to tune the parameters of the chorus effect. You can pass an additional argument that determines duration to play the audio.

```
duration = 30; % in seconds
audioDelayEffectsExampleApp('chorus',duration);
```



Flanger

You can model the flanging effect by delaying the audio input by an amount that is modulated by a low-frequency oscillator (LFO). The delay line used in flanger can also have a feedback path. `audioexample.Flanger` implements this effect. The block diagram shows a high-level implementation of a flanger effect.

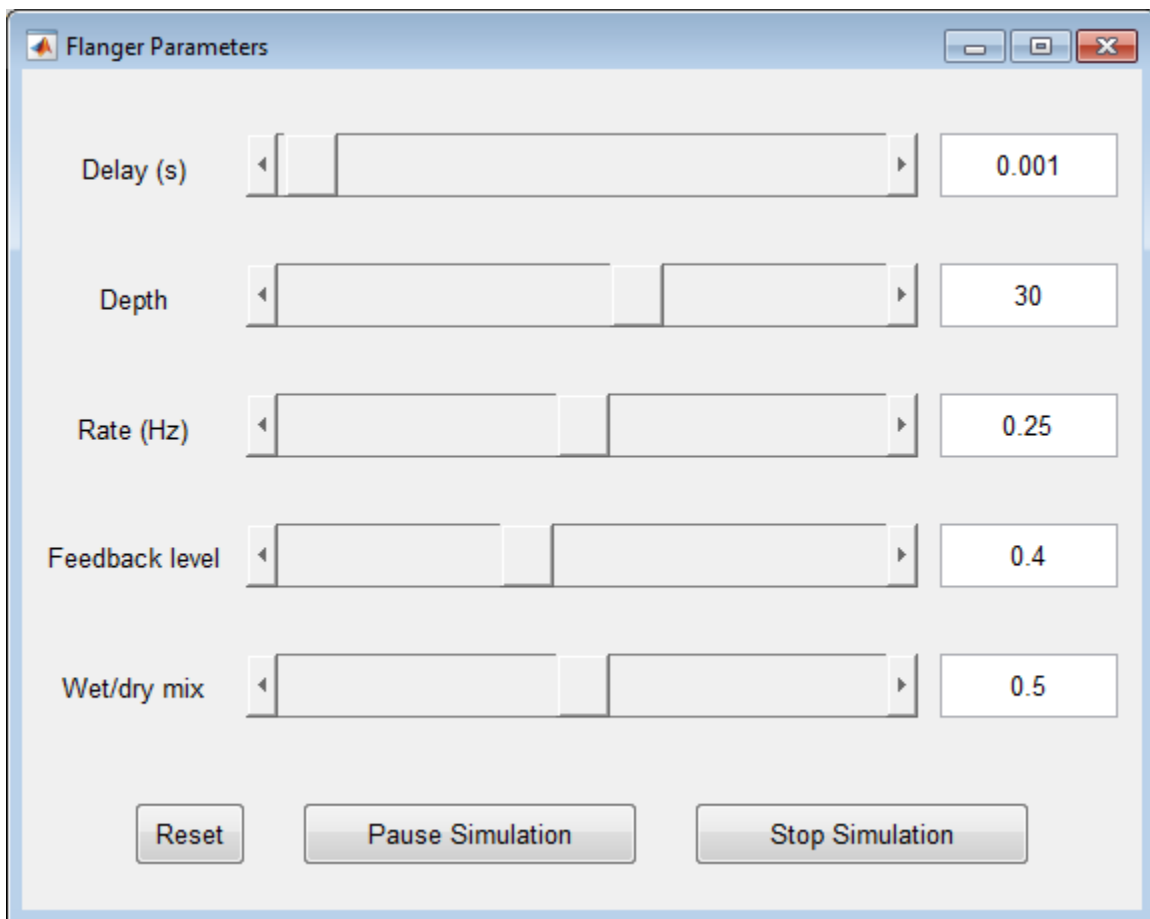


The flanger effect example has five tunable parameters that can be modified while the simulation is running:

- Delay - Base delay applied to audio signal, in seconds
- Depth - Amplitude of LFO
- Rate - Frequency of LFO, in Hz
- FeedbackLevel - Feedback gain applied to delay line
- WetDryMix - Ratio of wet signal added to dry signal

You can try out `audioexample.Flanger` by running `audioDelayEffectsExampleApp` with 'flanger' as input. The example reads an audio signal from a file, applies the flanger effect, then plays the processed signal through your audio output device. It also launches a UI that allows you to tune the parameters of the flanger effect. The second input to this function is optional, and decides how long the audio should be played. You can pass an additional argument that determines duration to play the audio.

```
duration = 30; % in seconds
audioDelayEffectsExampleApp('flanger',duration);
```

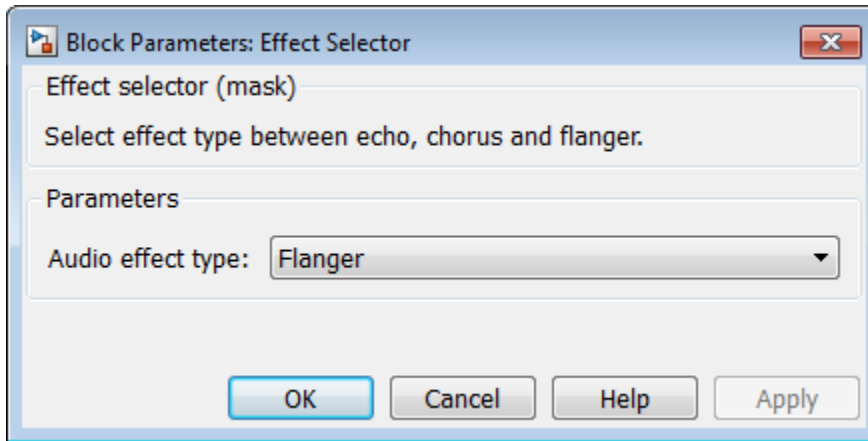


Audio Effects in Simulink

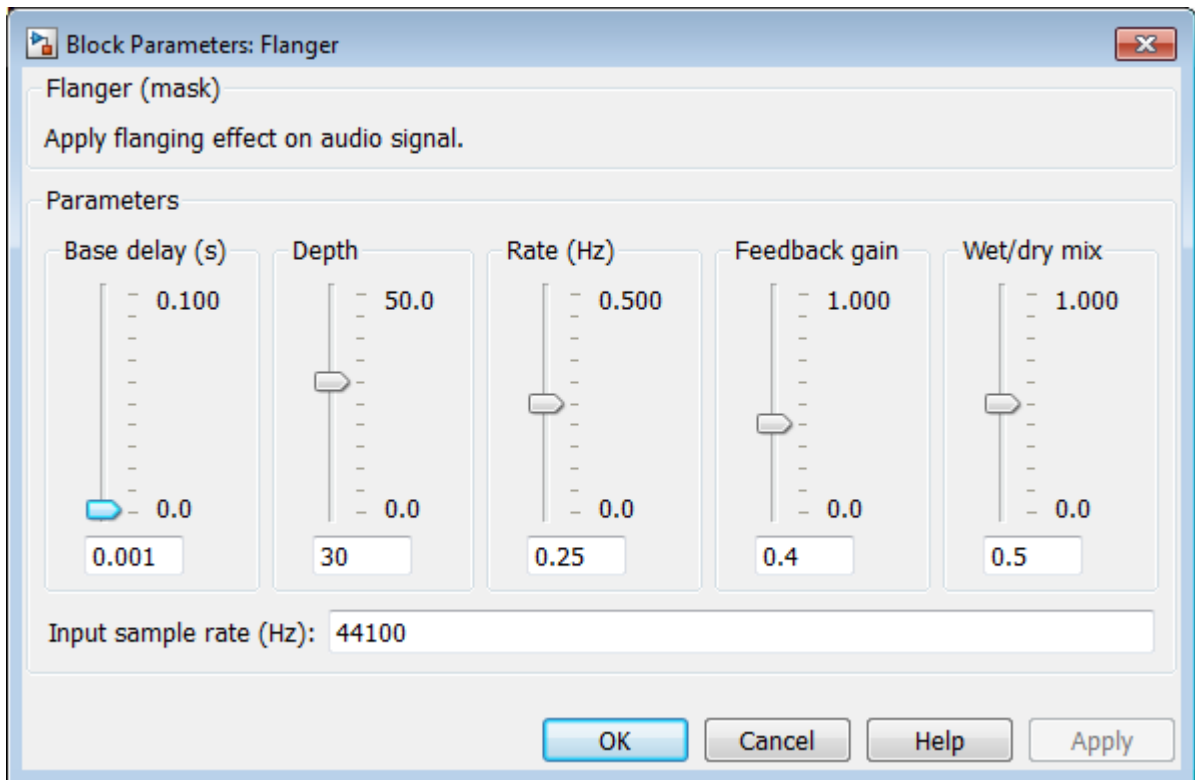
You can use the System objects `audioexample.Echo`, `audioexample.Chorus` and `audioexample.Flanger` in Simulink by using the MATLAB System (Simulink) block. The model `audiodelaybasedeffects` has these effects ready for simulation.

```
open_system('audiodelaybasedeffects')
```

You can select the effect to be applied by double-clicking on the **Effect Selector** block.



Once the effect has been selected, you can click on **Launch Parameter Tuning UI** button to bring up the dialog that has all tunable parameters of the effect.



This dialog will remain available even during simulation. You can run the model and tune properties of the effect to listen to how they affect the audio output.

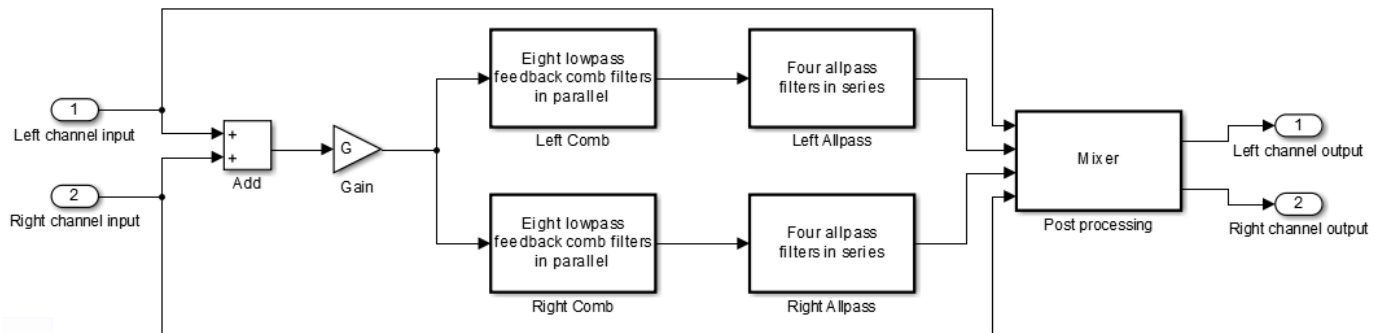
Add Reverberation Using Freeverb Algorithm

This example shows how to apply reverberation to audio by using the Freeverb reverberation algorithm. The reverberation can be tuned using a user interface (UI) in MATLAB or through a MIDI controller. This example illustrates MATLAB® and Simulink® implementations.

Introduction

Reverberators are used to add the effect of multiple decaying echoes, or reverbs, to audio signals. A common use of reverberation is to simulate music played in a closed room. Most digital audio workstations (DAWs) have options to add such effects to the sound track.

In this example, you add reverberation to audio through the Freeverb algorithm. Freeverb is a popular implementation of the Schroeder reverberator. A high-level model of the Freeverb algorithm is shown below:



Example Architecture

The reverberator is implemented in the System object `audioexample.FreeverbReverberator`. The object has five properties that can be tuned while the simulation is running: `RoomSize`, `StereoWidth`, `WetDryMix`, `Balance`, and `Volume`. `RoomSize` affects the feedback gain of the comb filters. `StereoWidth` and `WetDryMix` both take part in the mixing stage that happens after filtering is complete. The default values of the `StereoSpread`, `CombDelayLength`, and `AllpassDelayLength` properties are taken from the Freeverb specifications.

MATLAB Simulation

To use the reverberator on an audio signal, run `audioFreeverbReverberationExampleApp`.

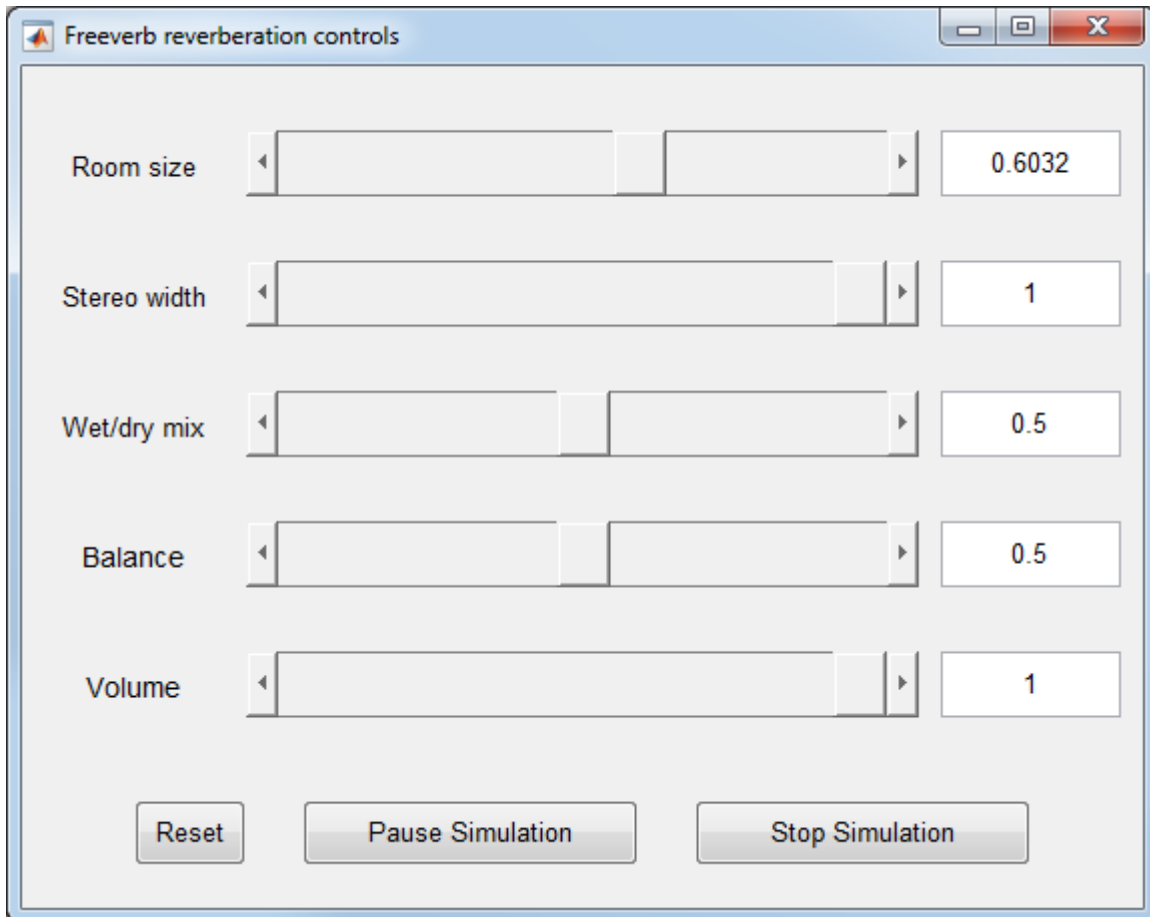
`audioFreeverbReverberationExampleApp`

The `audioFreeverbReverberationExampleApp` command first sets up the audio source and player. It then iteratively calls the `audioexample.FreeverbReverberator` System object with the audio input, providing addition of reverberation in a streaming fashion. The output of the object is played back so you can hear the effect added to the audio.

The simulation opens a UI to interact with `audioexample.FreeverbReverberator` while the simulation is running. The UI allows you to tune parameters and the results are reflected in the simulation instantly. For example, moving the slider **Room size** to the left while the simulation is running decreases the reflectivity of the walls of the room being simulated.

There are also three buttons on the UI - the **Reset** button will reset the states of the comb and allpass sections in reverberator to their initial values and the **Pause Simulation** button will hold the

simulation until you click on it again. The simulation may be terminated by either closing the UI or by clicking on the **Stop simulation** button. If you have a MIDI controller, it is possible to synchronize it with the UI. You can do this by choosing a MIDI control in the dialog that is opened when you right-click on the sliders or buttons and select "Synchronize" from the context menu. The chosen MIDI control then works in accordance with the slider or button so that operating one control is tracked by the other.



If you see a lot of queue underrun warnings, you will need to adjust the buffer and queue size of audio player used in `audioFreeverbReverberationExampleApp`. More information on this can be found at the documentation page for `audioDeviceWriter`. The audio source in this example is an audio file, but you can replace it with an audio input device (through `audioDeviceReader`) to add reverberation to live audio. For ways to reduce latency while not having any overruns/underruns, you can follow the example "Measure Audio Latency" on page 1-238.

Using a Generated MEX File

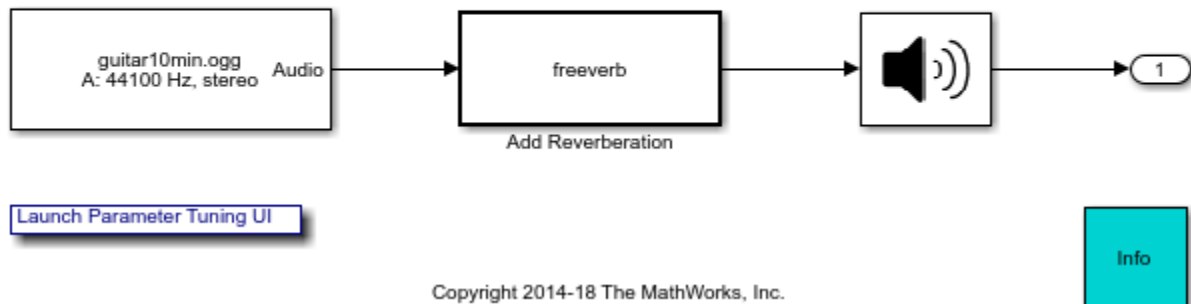
Using MATLAB Coder™, you can generate a MEX file for the main processing algorithm by executing the `HelperFreeverbCodeGeneration` command. You can use the generated MEX file by executing the `audioFreeverbReverberationExampleApp` command with `true` as an argument.

```
audioFreeverbReverberationExampleApp(true)
```

Simulink Version

`audiofreeverbreverberation` is a Simulink model that implements the same Freeverb reverberation example highlighted in the previous sections.

Adding Reverberation to Audio



In this model, the addition of reverberation is modeled using the `audioexample.FreeverbReverberator` System object used inside a MATLAB System block. Using the MATLAB System block saves you the effort of reimplementing a MATLAB algorithm in Simulink. You can open the UI to tune Freeverb parameters by clicking the 'Launch Parameter Tuning UI' link on the model.

The model generates code when it is simulated. Therefore, it must be executed from a folder with write permissions.

Acknowledgement

The algorithm in this example is based on the public domain 'Freeverb' model written by Jezar at Dreampoint (June 2000).

Reference

Smith, J.O. "Freeverb", in "Physical Audio Signal Processing", <https://ccrma.stanford.edu/~jos/pasp/Freeverb.html>, online book, 2010 edition, accessed April 24, 2014.

Multiband Dynamic Range Compression

This example shows how to simulate a digital audio multiband dynamic range compression system.

Introduction

Dynamic range compression reduces the dynamic range of a signal by attenuating the level of strong peaks, while leaving weaker peaks unchanged. Compression has applications in audio recording, mixing, and in broadcasting.

Multiband compression compresses different audio frequency bands separately, by first splitting the audio signal into multiple bands and then passing each band through its own independently adjustable compressor. Multiband compression is widely used in audio mastering and is often included in audio workstations.

The multiband compressor in this example first splits an audio signal into different bands using a multiband crossover filter. Linkwitz-Riley crossover filters are used to obtain an overall allpass frequency response. Each band is then compressed using a separate dynamic range compressor. Key compressor characteristics, such as the compression ratio, the attack and release time, the threshold and the knee width, are independently tunable for each band. The effect of compression on the dynamic range of the signal is showcased.

Linkwitz-Riley Crossover Filters

A Linkwitz-Riley crossover filter consists of a combination of a lowpass and highpass filter, each formed by cascading two lowpass or highpass Butterworth filters. Summing the response of the two filters yields a gain of 0 dB at the crossover frequency, so that the crossover acts like an allpass filter (and therefore introducing no distortion in the audio signal).

`crossoverFilter` may be used to implement a Linkwitz-Riley System object. Since a Linkwitz-Riley crossover filter is formed by cascading two Butterworth filters, its order is always even. A Butterworth filter's slope is equal to $6 \cdot N$ dB/octave, where N is the filter order. When the `CrossoverSlopes` property of `crossoverFilter` is divisible by 12 (i.e. the filter is even-ordered), the object implements a Linkwitz-Riley crossover. Otherwise, the object implements a Butterworth crossover, where the lowpass and highpass sections are each implemented using a single Butterworth filter of order `CrossoverSlopes/6`.

Here is an example where an fourth-order Linkwitz-Riley crossover is used to filter a signal. Notice that the lowpass and highpass sections each have a -6 dB gain at the crossover frequency. The sum of the lowpass and highpass sections is allpass.

```
Fs = 44100;

% Linkwitz-Riley filter
crossover = crossoverFilter(1,5000,4*6,Fs);

% Transfer function estimator
transferFuncEstimator = dsp.TransferFunctionEstimator( ...
    'FrequencyRange', 'onesided', 'SpectralAverages', 20);

frameLength = 1024;

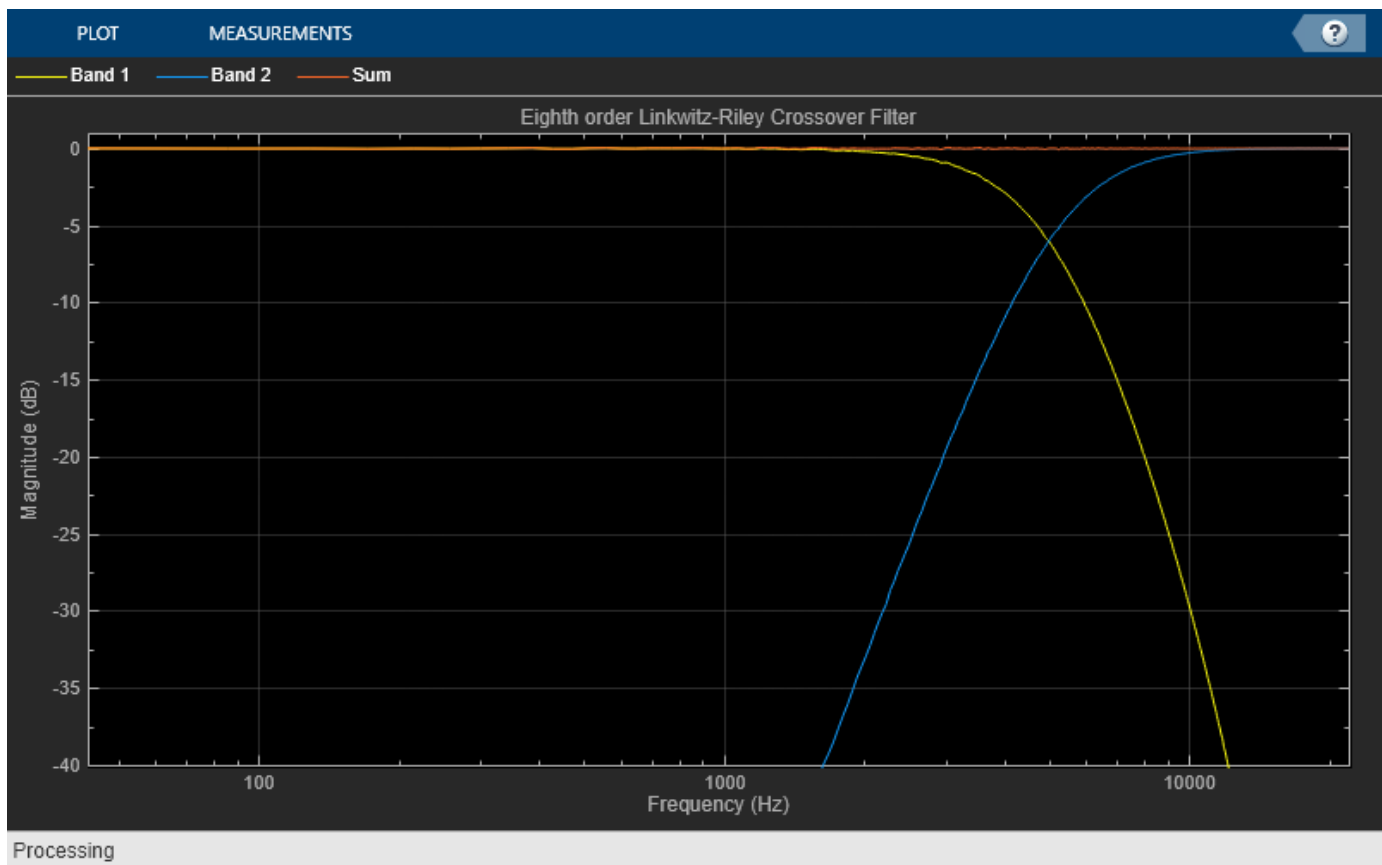
scope = dsp.ArrayPlot( ...
    'PlotType', 'Line', ...
    'YLimits', [-40 1], ...
```

```

'YLabel','Magnitude (dB)', ...
'XScale','log', ...
'SampleIncrement',(Fs/2)/(frameLength/2+1), ...
'XLabel','Frequency (Hz)', ...
'Title','Eighth order Linkwitz-Riley Crossover Filter', ...
'ShowLegend',true, ...
'ChannelNames',{'Band 1','Band 2','Sum'});

tic
while toc < 10
    in = randn(frameLength,1);
    % Return lowpass and highpass responses of the crossover filter
    [y1p,y1h] = crossover(in);
    % sum the responses
    y = y1p + y1h;
    v = transferFuncEstimator(repmat(in,1,3),[y1p y1h y]);
    scope(20*log10(abs(v)));
end

```



Multiband Crossover Filters

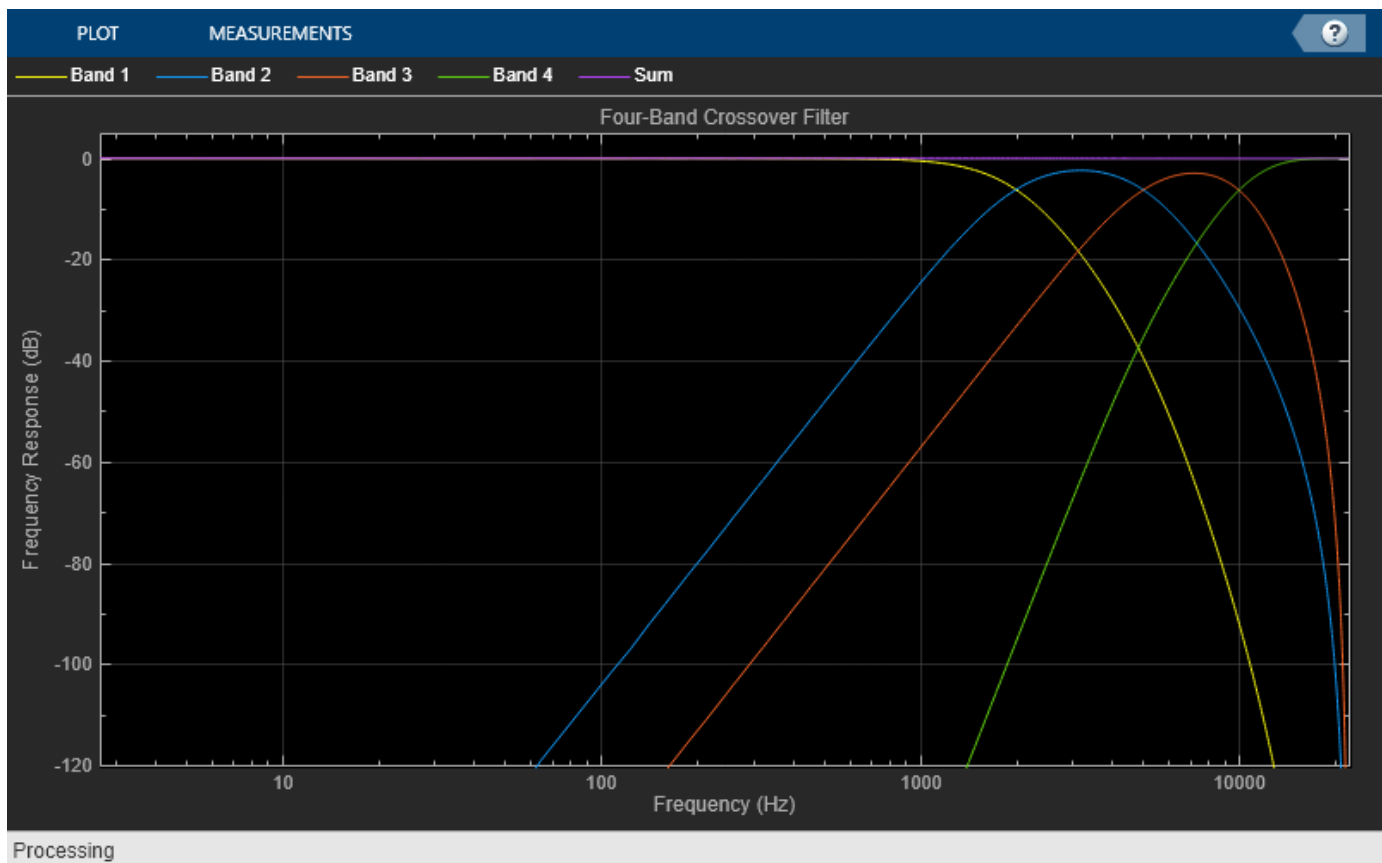
`crossoverFilter` may also be used to implement a multiband crossover filter by combining two-band crossover filters and allpass filters in a tree-like structure. The filter divides the spectrum into multiple bands such that their sum is a perfect allpass filter.

The example below shows a four-band crossover filter formed of fourth order Linkwitz-Riley crossover filters. Notice the allpass response of the sum of the four bands.

```

Fs = 44100;
crossover = crossoverFilter(3,[2e3 5e3 10e3],[24 24 24],44100);
transferFuncEstimator = dsp.TransferFunctionEstimator('FrequencyRange','onesided','SpectralAveraging','none');
L = 2^14;
scope = dsp.ArrayPlot( ...
    'PlotType','Line', ...
    'XOffset',0, ...
    'YLimits',[-120 5], ...
    'XScale','log', ...
    'SampleIncrement',.5 * Fs/(L/2 + 1), ...
    'YLabel','Frequency Response (dB)', ...
    'XLabel','Frequency (Hz)', ...
    'Title','Four-Band Crossover Filter', ...
    'ShowLegend',true, ...
    'ChannelNames',{'Band 1','Band 2','Band 3','Band 4','Sum'});
tic;
while toc < 10
    in = randn(L,1);
    % Split the signal into four bands
    [ylp,ybp1,ybp2,yhp] = crossover(in);
    y = ylp + ybp1 + ybp2 + yhp;
    z = transferFuncEstimator(repmat(in,1,5),[ylp,ybp1,ybp2,yhp,y]);
    scope(20*log10(abs(z)))
end

```

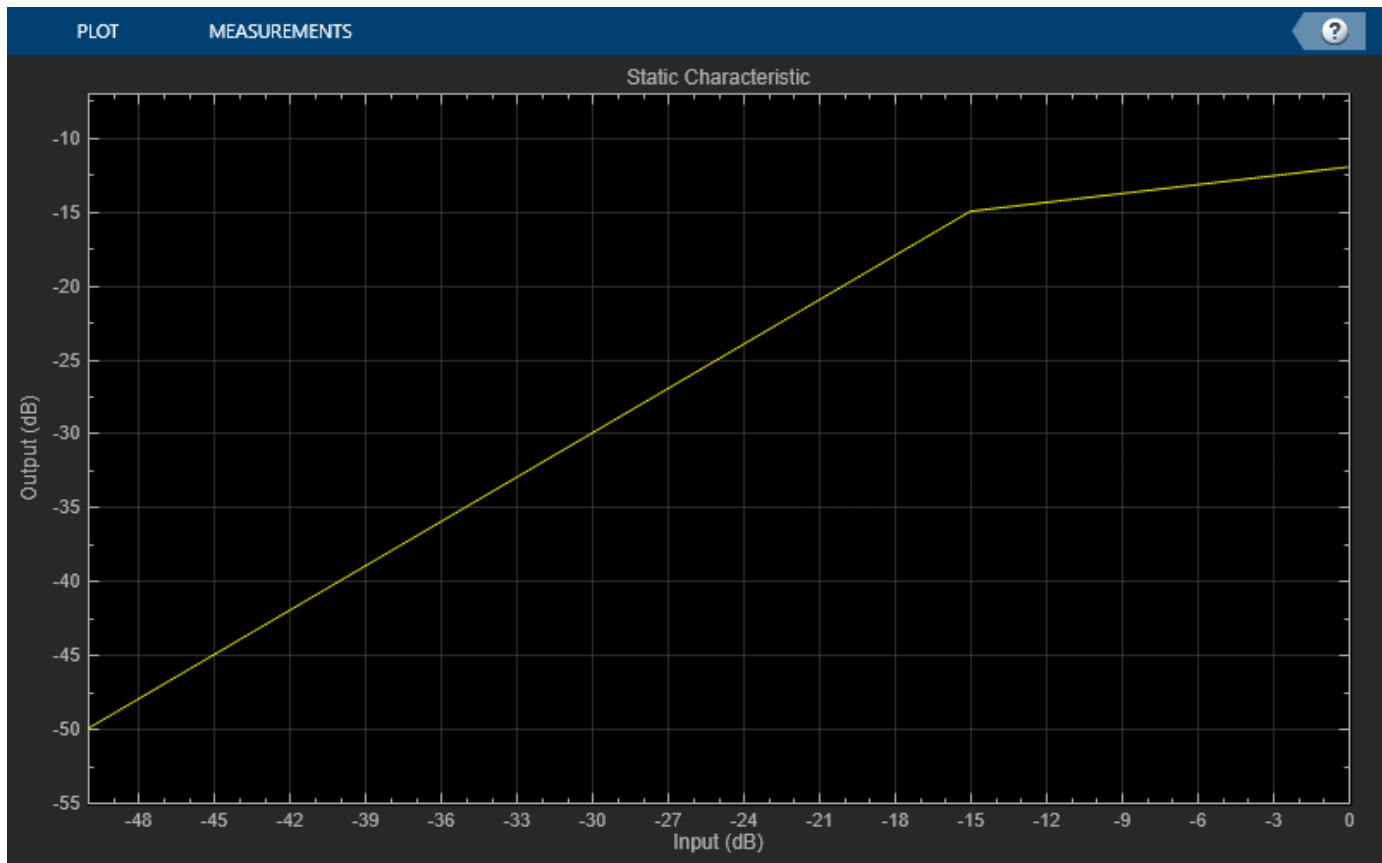


Dynamic Range Compression

`compressor` is a dynamic range compressor System object. The input signal is compressed when it exceeds the specified threshold. The amount of compression is controlled by the specified compression ratio. The attack and release times determine how quickly the compressor starts or stops compressing. The knee width provides a smooth transition for the compressor gain around the threshold. Finally, a make-up gain can be applied at the output of the compressor. This make-up gain amplifies both strong and weak peaks equally.

The static compression characteristic of the compressor depends on the compression ratio, the threshold and the knee width. The example below illustrates the static compression characteristic for a hard knee:

```
drc = compressor(-15,5);  
visualize(drc);
```



In order to view the effect of threshold, ratio and knee width on the compressor's static characteristic, change the values of the `Threshold`, `Ratio` and `KneeWidth` properties. The static characteristic plot should change accordingly.

The compressor's attack time is defined as the time (in msec) it takes for the compressor's gain to rise from 10% to 90% of its final value when the signal level exceeds the threshold. The compressor's release time is defined as the time (in seconds) it takes the compressor's gain to drop from 90% to 10% of its value when the signal level drops below the threshold. The example below illustrates the signal envelope for different release and attack times:

```
Fs = 44100;

drc = compressor(-10,5, ...
    'SampleRate',Fs, ...
    'AttackTime',0.050, ...
    'ReleaseTime',0.200, ...
    'MakeUpGainMode','Property');

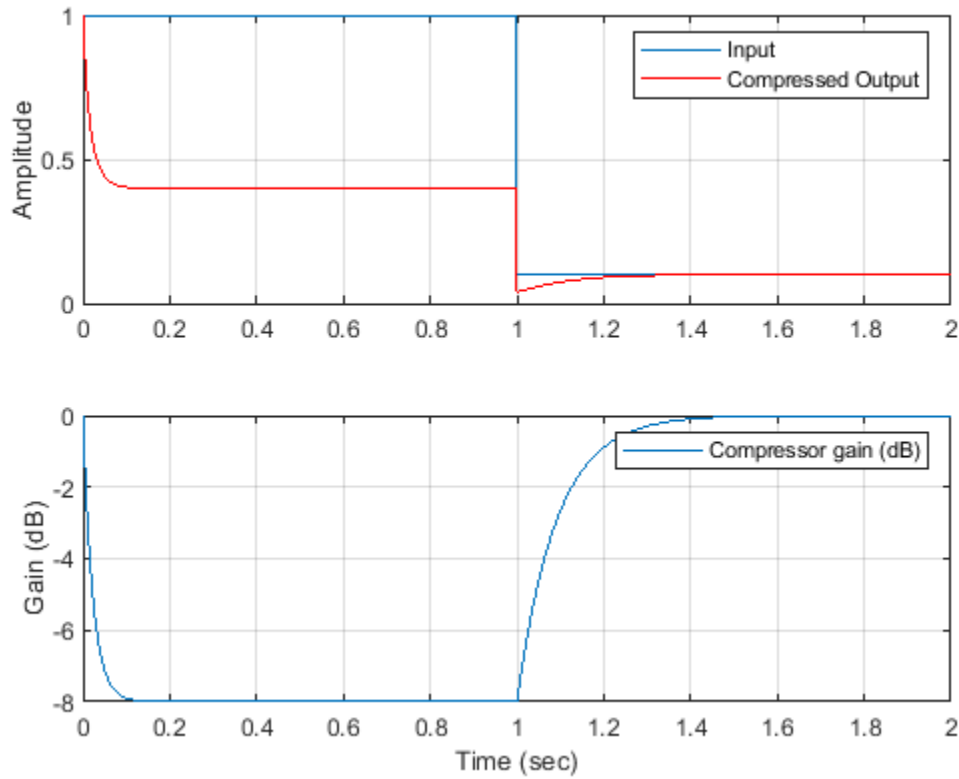
x = [ones(Fs,1);0.1*ones(Fs,1)];
[y,g] = drc(x);

t = (1/Fs) * (0: 2*Fs - 1);

figure

subplot(211)
plot(t,x);
hold on
grid on
plot(t,y,'r')
ylabel('Amplitude')
legend('Input','Compressed Output')

subplot(212)
plot(t,g)
grid on
legend('Compressor gain (dB)')
xlabel('Time (sec)')
ylabel('Gain (dB)')
```



The input maximum level is 0 dB, which is above the specified -10 dB threshold. The steady-state compressor output for a 0 dB input is $-10 + 10/5 = -8$ dB. The gain is therefore -8 dB. The attack time is defined as the time it takes the compressor gain to rise from 10% to 90% of its final value when the input level goes above the threshold, or in this case, from -0.8 dB to -7.2 dB. Let's find the times at which the gains in the compression stage are equal to -0.8 dB and -7.2 dB, respectively:

```
[~,t1] = min(abs(g(1:Fs) + 0.1 * 8));
[~,t2] = min(abs(g(1:Fs) + 0.9 * 8));
tAttack = (t2 - t1) / Fs;
fprintf('Attack time is %d s\n',tAttack)
```

```
Attack time is 5.000000e-02 s
```

The input signal then drops back down to 0, where there is no compression. The release time is defined as the time it takes the gain to drop from 90% to 10% of its absolute value when the input goes below the threshold, or in this case, -7.2 dB to -0.8 dB. Let's find the times at which the gains in the no-compression stage are equal to -7.2 dB and -0.8 dB, respectively:

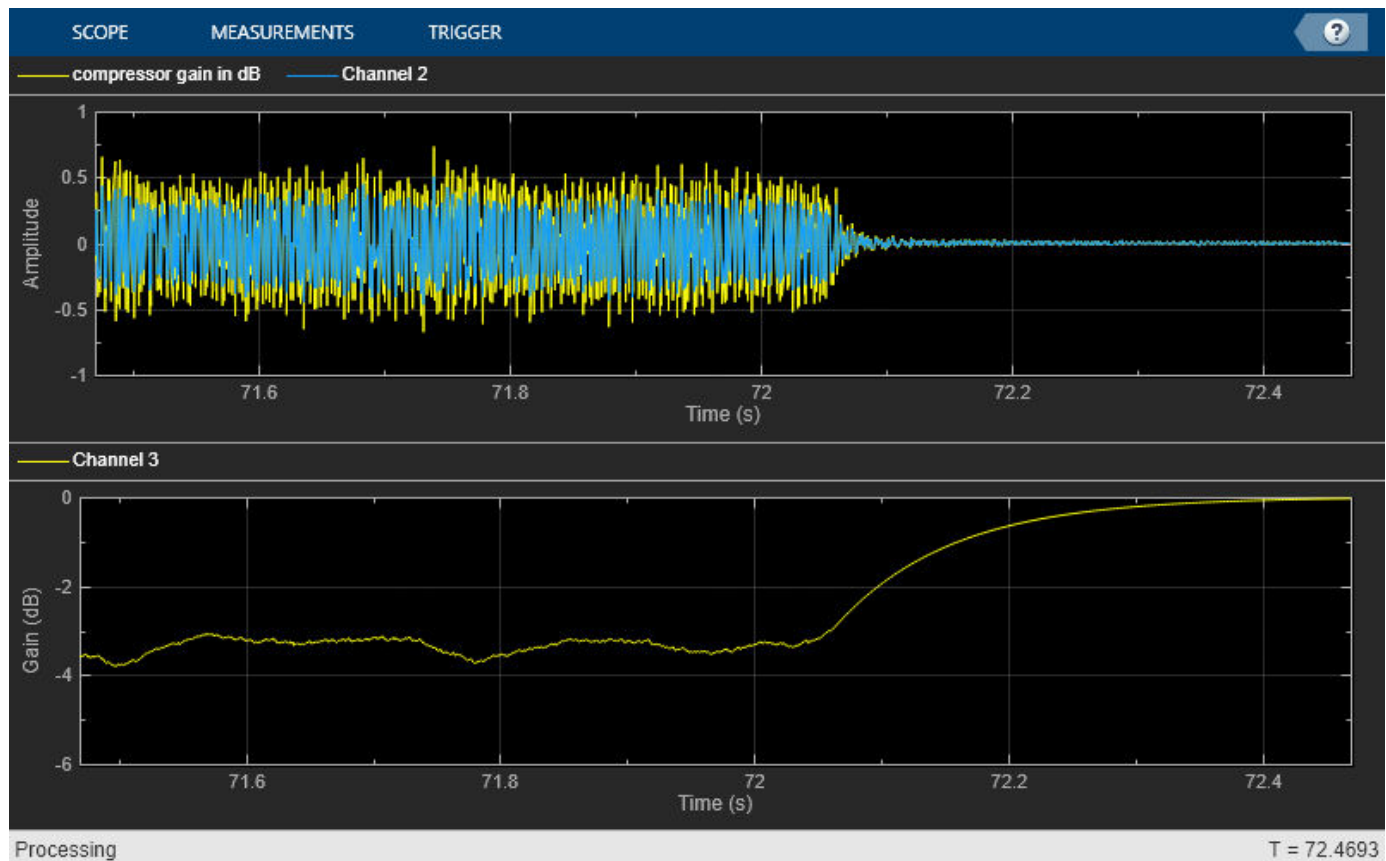
```
[~,t1] = min(abs(g(Fs:end) + 0.9 * 8));
[~,t2] = min(abs(g(Fs:end) + 0.1 * 8));
tRelease = (t2 - t1) / Fs;
fprintf('Release time is %d s\n',tRelease)
```

```
Release time is 2.000000e-01 s
```

The example below illustrates the effect of dynamic range compression on an audio signal. The compression threshold is set to -15 dB, and the compression ratio is 7.

```
frameLength = 1024;
reader = dsp.AudioFileReader('Filename', ...
    'RockGuitar-16-44p1-stereo-72secs.wav', ...
    'SamplesPerFrame',frameLength);
% Compressor. Threshold = -15 dB, ratio = 7
drc = compressor(-15,7, ...
    'SampleRate',reader.SampleRate, ...
    'MakeUpGainMode','Property', ...
    'KneeWidth',5);
scope = timescope('SampleRate',reader.SampleRate, ...
    'TimeSpanSource','property',...
    'TimeSpan',1,'BufferLength',Fs*4, ...
    'ShowGrid',true, ...
    'LayoutDimensions',[2 1], ...
    'NumInputPorts',2, ...
    'TimeSpanOvverrunAction','Scroll');
scope.ActiveDisplay = 1;
scope.YLimits = [-1 1];
scope.ShowLegend = true;
scope.ChannelNames = {'Original versus compressed audio'};
scope.ActiveDisplay = 2;
scope.YLimits = [-6 0];
scope.YLabel = 'Gain (dB)';
scope.ShowLegend = true;
scope.ChannelNames = {'compressor gain in dB'};

while ~isDone(reader)
    x = reader();
    [y,g] = drc(x);
    x1 = x(:,1);
    y1 = y(:,1);
    scope([x1,y1],g(:,1))
end
```



Simulink Version of the Multiband Dynamic Range Compression Example

The following model implements the multiband dynamic range compression example:

```
model = 'audiomultibanddynamiccompression';
open_system(model)
```

In this example, the audio signal is first divided into four bands using a multiband crossover filter. Each band is compressed using a separate compressor. The four bands are then recombined to form the audio output. The dynamic range of the uncompressed and compressed signals (defined as the ratio of the largest absolute value of the signal to the signal RMS) is computed. To hear the difference between the original and compressed audio signals, toggle the switch on the top level.

The model integrates a User Interface (UI) designed to interact with the simulation. The UI allows you to tune parameters and the results are reflected in the simulation instantly. To launch the UI that controls the simulation, click the 'Launch Parameter Tuning UI' link on the model.

```
set_param(model, 'StopTime', '(1/44100) * 8192 * 20');
sim(model);
```

Close the model:

```
bdclose(model)
```

MATLAB Version of the Multiband Dynamic Range Compression Example

`HelperMultibandCompressionSim` is the MATLAB function containing the multiband dynamic range compression example's implementation. It instantiates, initializes and steps through the objects forming the algorithm.

The function `multibandAudioCompressionExampleApp` wraps around `HelperMultibandCompressionSim` and iteratively calls it. It also plots the uncompressed versus compressed audio signals. Plotting occurs when the `plotResults` input to the function is 'true'.

Execute `multibandAudioCompressionExampleApp` to run the simulation and plot the results on scopes. Note that the simulation runs for as long as the user does not explicitly stop it.

`multibandAudioCompressionExampleApp` launches a UI designed to interact with the simulation. The UI allows you to tune parameters and the results are reflected in the simulation instantly. For more information on the UI, please refer to `HelperCreateParamTuningUI`.

MATLAB Coder can be used to generate C code for the function `HelperMultibandCompressionSim`. In order to generate a MEX-file for your platform, execute `HelperMultibandCompressionCodeGeneration`.

By calling the wrapper function `multibandAudioCompressionExampleApp` with 'true' as an argument, the generated MEX-file can be used instead of `HelperMultibandCompressionSim` for the simulation. In this scenario, the UI is still running inside the MATLAB environment, but the main processing algorithm is being performed by a MEX-file. Performance is improved in this mode without compromising the ability to tune parameters.

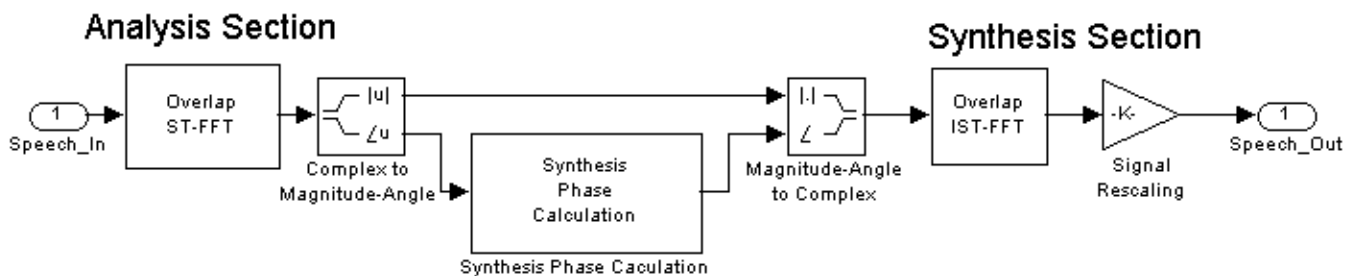
Call `multibandAudioCompressionExampleApp(true)` to use the MEX-file for simulation. Again, the simulation runs till the user explicitly stops it from the UI.

Pitch Shifting and Time Dilation Using a Phase Vocoder in MATLAB

This example shows how to implement a phase vocoder to time stretch and pitch scale an audio signal.

Introduction

The phase vocoder performs time stretching and pitch scaling by transforming the audio into frequency domain. The following block diagram shows the operations involved in the phase vocoder implementation.



The phase vocoder has an analysis section that performs an overlapped short-time FFT (ST-FFT) and a synthesis section that performs an overlapped inverse short-time FFT (IST-FFT). To time stretch a signal, the phase vocoder uses a larger hop size for the overlap-add operation in the synthesis section than the analysis section. Here, the hop size is the number of samples processed at one time. As a result, there are more samples at the output than at the input although the frequency content remains the same. Now, you can pitch scale this signal by playing it back at a higher sample rate, which produces a signal with the original duration but a higher pitch.

Initialization

To achieve optimal performance, you must create and initialize your System objects before using them in a processing loop. Use these next sections of code to initialize the required variables and load the input speech data. You set an analysis hop size of 64 and a synthesis hop size of 90 because you want to stretch the signal by a factor of 90/64.

Initialize some variables used in configuring the System objects you create below.

```
WindowLen = 256;
AnalysisLen = 64;
SynthesisLen = 90;
Hopratio = SynthesisLen/AnalysisLen;
```

Create a System object to read in the input speech signal from an audio file.

```
reader = dsp.AudioFileReader('SpeechDFT-16-8-mono-5secs.wav', ...
    'SamplesPerFrame',AnalysisLen, ...
    'OutputDataType','double');
```

Create STFT/ISTFT pair

```
win = sqrt(hanning(WindowLen,'periodic'));
stft = dsp.STFT(win, WindowLen - AnalysisLen, WindowLen);
istft = dsp.ISTFT(win, WindowLen - SynthesisLen );
```

Create a System object to play the original speech signal.

```
Fs = 8000;
player = audioDeviceWriter('SampleRate',Fs, ...
    'SupportVariableSizeInput',true, ...
    'BufferSize',512);
```

Create a System object to log your data.

```
logger = dsp.SignalSink;
```

Initialize the variables used in the processing loop.

```
unwrapdata = 2*pi*AnalysisLen*(0:WindowLen-1)'/WindowLen;
yangle = zeros(WindowLen,1);
firsttime = true;
```

Stream Processing Loop

Now that you have instantiated your System objects, you can create a processing loop that performs time stretching on the input signal. The loop is stopped when you reach the end of the input file, which is detected by the AudioFileReader System object.

```
while ~isDone(reader)
    y = reader();

    player(y); % Play back original audio

    % ST-FFT
    yfft = stft(y);

    % Convert complex FFT data to magnitude and phase.
    ymag      = abs(yfft);
    yprevangle = yangle;
    yangle     = angle(yfft);

    % Synthesis Phase Calculation
    % The synthesis phase is calculated by computing the phase increments
    % between successive frequency transforms, unwrapping them, and scaling
    % them by the ratio between the analysis and synthesis hop sizes.
    yunwrap = (yangle - yprevangle) - unwrapdata;
    yunwrap = yunwrap - round(yunwrap/(2*pi))*2*pi;
    yunwrap = (yunwrap + unwrapdata) * Hopratio;
    if firsttime
        ysangle = yangle;
        firsttime = false;
    else
        ysangle = ysangle + yunwrap;
    end

    % Convert magnitude and phase to complex numbers.
    ys = ymag .* complex(cos(ysangle), sin(ysangle));

    % IST-FFT
```



```

        yistfft = istfft(ys);

        logger(yistfft) % Log signal
    end

```

Release

Call release on the System objects to close any open files and devices.

```

release(reader)
release(player)

```

Play the Time-Stretched Signals

```

loggedSpeech = logger.Buffer(200:end)';
player = audioDeviceWriter('SampleRate',Fs, ...
    'SupportVariableSizeInput',true, ...
    'BufferSize',512);
player(loggedSpeech. ');

```

Play the Pitch-Scaled Signals

The pitch-scaled signal is the time-stretched signal played at a higher sampling rate which produces a signal with a higher pitch.

```

Fs_new = Fs*(SynthesisLen/AnalysisLen);
player = audioDeviceWriter('SampleRate',Fs_new, ...
    'SupportVariableSizeInput',true, ...
    'BufferSize',1024);
player(loggedSpeech. ');

```

Time Dilation with audioTimeScaler

You can easily apply time dilation with `audioTimeScaler`. `audioTimeScaler` implements an analysis-synthesis phase vocoder for time scaling.

Instantiate an `audioTimeScaler` with the desired speedup factor, window, and analysis hop length:

```

ats = audioTimeScaler(AnalysisLen/SynthesisLen,'Window',win,'OverlapLength',WindowLen-AnalysisLen);

```

Create a System object to play the time-stretched speech signal.

```

player = audioDeviceWriter('SampleRate',Fs, ...
    'SupportVariableSizeInput',true, ...
    'BufferSize',512);

```

Create a processing loop that performs time stretching on the input signal.

```

while ~isDone(reader)

    x = reader();

    % Time-scale the signal
    y = ats(x);

    % Play the time-scaled signal
    player(y);
end

```

```
release(reader)
release(player)
```

Summary

This example shows the implementation of a phase vocoder to perform time stretching and pitch scaling of a speech signal. You can hear these time-stretched and pitch-scaled signals when you run the example.

References

A. D. Gotzen, N. Bernardini and D. Arfib, "Traditional Implementations of a Phase-Vocoder: The Tricks of the Trade," Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00), Verona, Italy, December 7-9, 2000.

Pitch Shifting and Time Dilation Using a Phase Vocoder in Simulink

This example shows how to use a phase vocoder to implement time dilation and pitch shifting of an audio signal.

The Example Model

The phase vocoder in this example consists of an analysis section, a phase calculation section and a synthesis section. The analysis section consists of an overlapped, short-time windowed FFT. The start of each frame to be transformed is delayed from the previous frame by the amount specified in the **Analysis hop size** parameter. The synthesis section consists of a short-time windowed IFFT and an overlap add of the resulting frames. The overlap size during synthesis is specified by the **Synthesis hop size** parameter.

The vocoder output has a different sample rate than its input. The ratio of the output to input sample rates is the **Synthesis hop size** divided by the **Analysis hop size**. If the output is played at the input sample rate, it is time stretched or time reduced depending on that ratio. If the output is played at the output sample rate, the sound duration is identical to the input, but is pitch shifted either up or down.

To prevent distortion, the phase of the frequency domain signal is modified in the phase calculation section. In the frequency domain, the signal is split into its magnitude and phase components. For each bin, a phase difference between frames is calculated, then normalized by the nominal phase of the bin. Phase modification first requires that the normalized phase differences be unwrapped. The unwrapped differences are multiplied by the **Synthesis hop size** divided by the **Analysis hop size**. The differences are accumulated, frame by frame, to recover the phase components. Magnitude and phase components are then recombined.

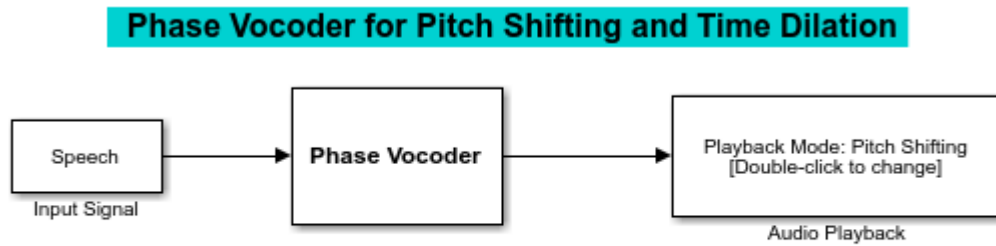
Exploring the Example

On running the model, the pitch-scaled signal is automatically played once the simulation has finished. The Audio Playback block allows you to choose between **Pitch Shifting** and **Time Dilation** modes.

Double-click the Phase Vocoder block. Change the **Synthesis hop-size** parameter to 64, the same value as the **Analysis hop-size** parameter. Run the simulation and listen to the three signals. The pitch-scaled signal has the same pitch as the original signal, and the time-stretched signal has the same speed as the original signal.

Next change the **Synthesis hop-size** parameter in the Phase Vocoder block to 48, which is less than the **Analysis hop-size** parameter. Run the simulation and listen to the three signals. The pitch-scaled signal has a lower pitch than the original signal. The time-stretched signal is faster than the original signal.

To see the implementation, right-click on the Phase Vocoder block and select Mask > Look Under Mask.



Copyright 2007-2015 The MathWorks, Inc.

References

A. D. Gotzen, N. Bernardini, and D. Arfib. "Traditional Implementations of a Phase-Vocoder: The Tricks of the Trade," *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00)*. Verona, Italy, December 7-9, 2000.

Remove Interfering Tone From Audio Stream

This example shows how to remove a 250 Hz interfering tone from a streaming audio signal using a notch filter.

Introduction

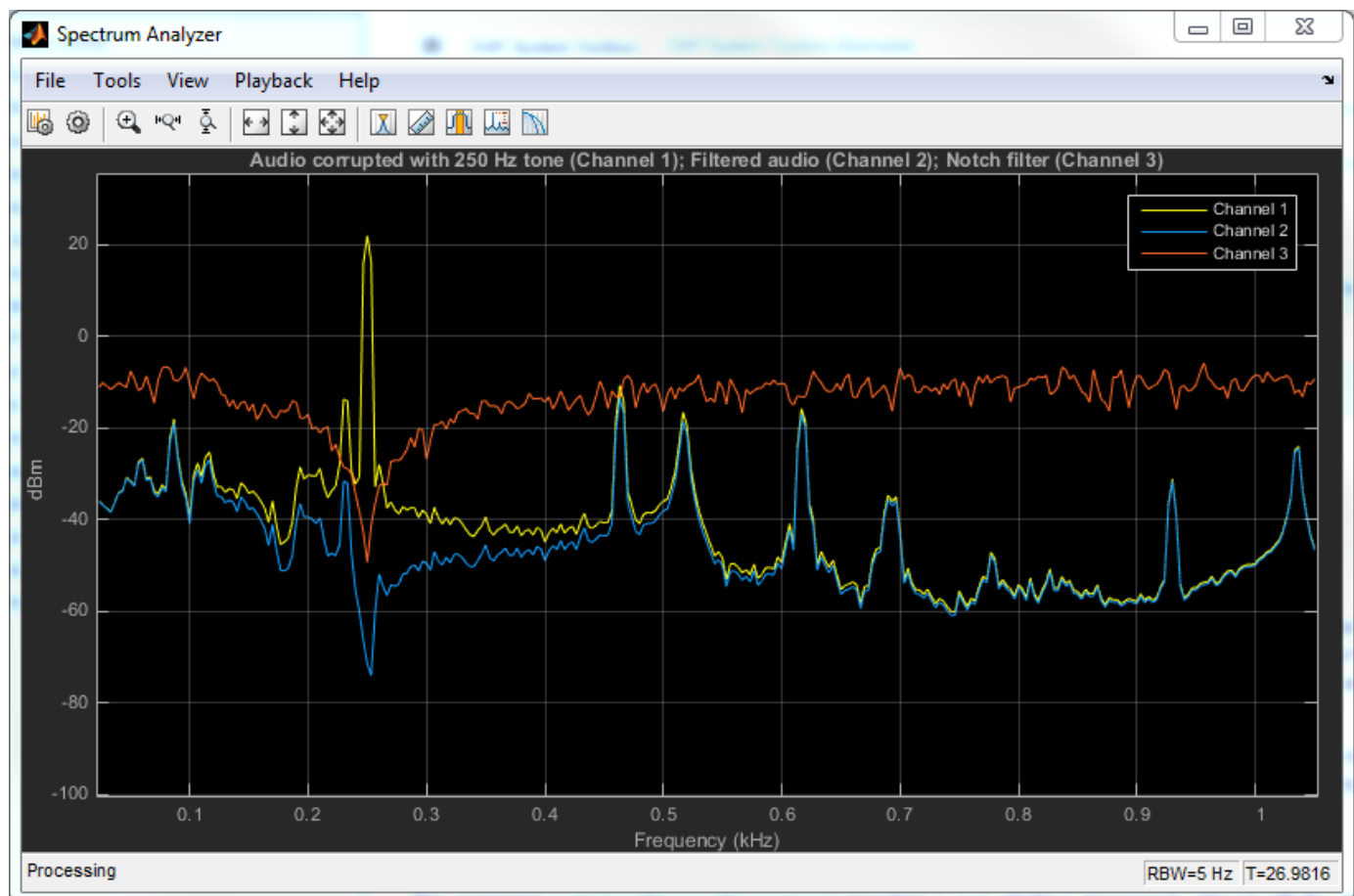
A notch filter is used to eliminate a specific frequency from a given signal. In their most common form, the filter design parameters for notch filters are center frequency for the notch and the 3 dB bandwidth. The center frequency is the frequency point at which the filter has a gain of zero. The 3 dB bandwidth measures the frequency width of the notch filter computed at the half-power, or 3 dB, attenuation point.

In this example, you tune a notch filter in order to eliminate a 250 Hz sinusoidal tone corrupting an audio signal. You can control both the center frequency and the bandwidth of the notch filter and listen to the filtered audio signal as you tune the design parameters.

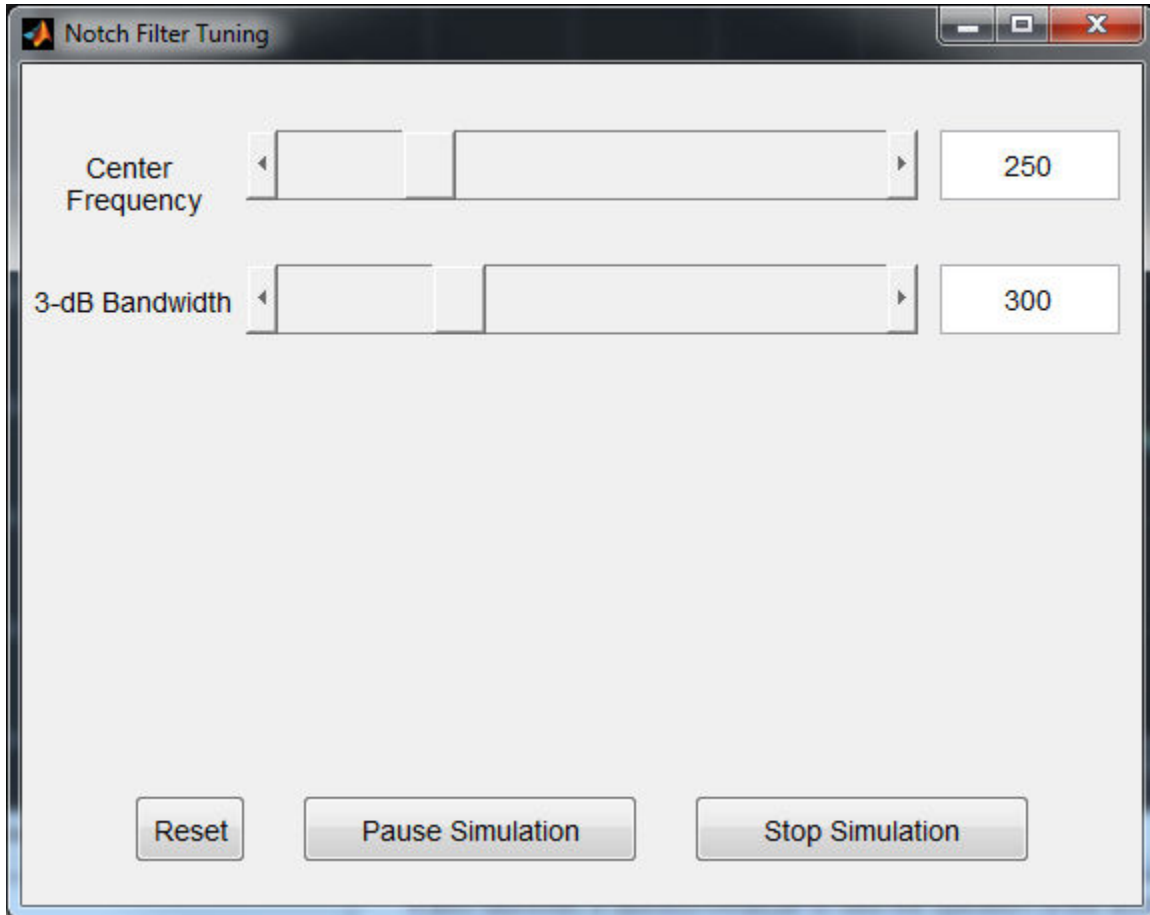
Example Architecture

The `audioToneRemovalExampleApp` command opens a user interface designed to interact with the simulation. It also opens a spectrum analyzer to view the spectrum of the audio with and without filtering and the magnitude response of the notch filter.

`audioToneRemovalExampleApp`



The notch filter is implemented using `dsp.NotchPeakFilter`. The filter has two specification modes: 'Design parameters' and 'Coefficients'. The 'Design parameters' mode allows you to specify the center frequency and bandwidth in Hz. This is the only mode used in this example. The 'Coefficients' mode allows you to specify the multipliers or coefficients in the filter directly. In the latter mode, each coefficient affects only one characteristic of the filter (either the center frequency or the 3 dB bandwidth). In other words, the effect of tuning the coefficients is completely decoupled.

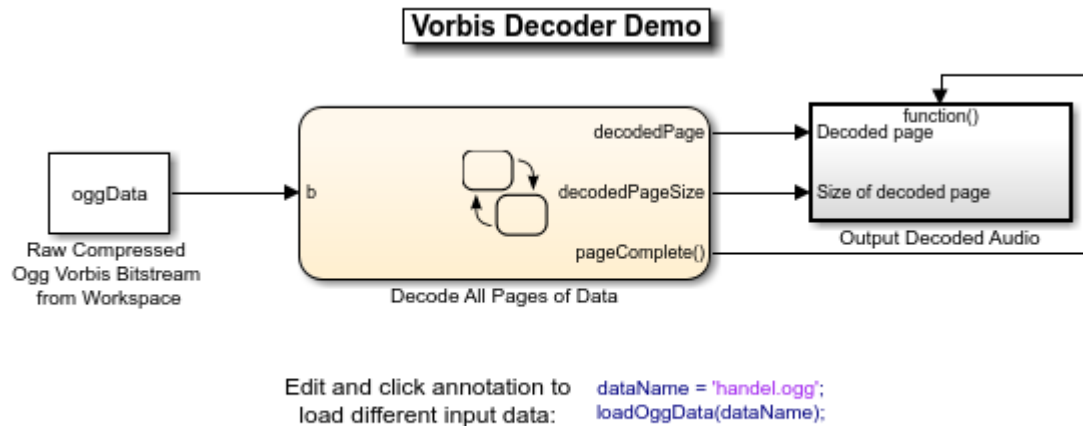


Using a Generated MEX File

Using MATLAB Coder, you can generate a MEX file for the main processing algorithm by executing the `HelperAudioToneRemovalCodeGeneration` command. You can use the generated MEX file by executing the `audioToneRemovalExampleApp(true)` command.

Vorbis Decoder

This example shows how to implement a Vorbis decoder, which is a freeware, open-source alternative to the MP3 standard. This audio decoding format supports the segmentation of encoded data into small packets for network transmission.



Copyright 2006-2020 The MathWorks, Inc.

Vorbis Basics

The Vorbis encoding format [1] is an open-source lossy audio compression algorithm similar to MPEG-1 Audio Layer 3, more commonly known as MP3. Vorbis has many of the same features as MP3, while adding flexibility and functionality. The Vorbis specification only defines the format of the bitstream and the decoding algorithm. This allows developers to improve the encoding algorithm over time and remain compatible with existing decoders.

Encoding starts by splitting the original signal into overlapping frames. Vorbis allows frames of different lengths so that it can efficiently handle stationary and transient signals. Each frame is multiplied by a window and transformed using the modified discrete cosine transform (mdct). The frames are then split into a rough approximation called the *floor*, and a remainder called the *residue*.

The flexibility of the Vorbis format is illustrated by its use of different methods to represent and encode the floor and residue portions of the signal. The algorithm introduces *modes* as a mechanism to specify these different methods and thereby code various frames differently.

Vorbis uses Huffman coding to compress the data contained in the floor and residue portions. Vorbis uses a dynamic probability model rather than the static probability model of MP3. Specifically, Vorbis builds custom codebooks for audio signals, which can differ for 'floor' and 'residue' and from frame to frame.

After Huffman encoding is complete, the frame data is bitpacked into a logical packet. In Vorbis, a series of such packets is always preceded by a header. The header contains all the information needed for correct decoding. This information includes a complete set of codebooks, descriptions of methods to represent the floor and residue, and the modes and mappings for multichannel support. The header can also include general information such as bit rates, sampling rate, song and artist names, etc.

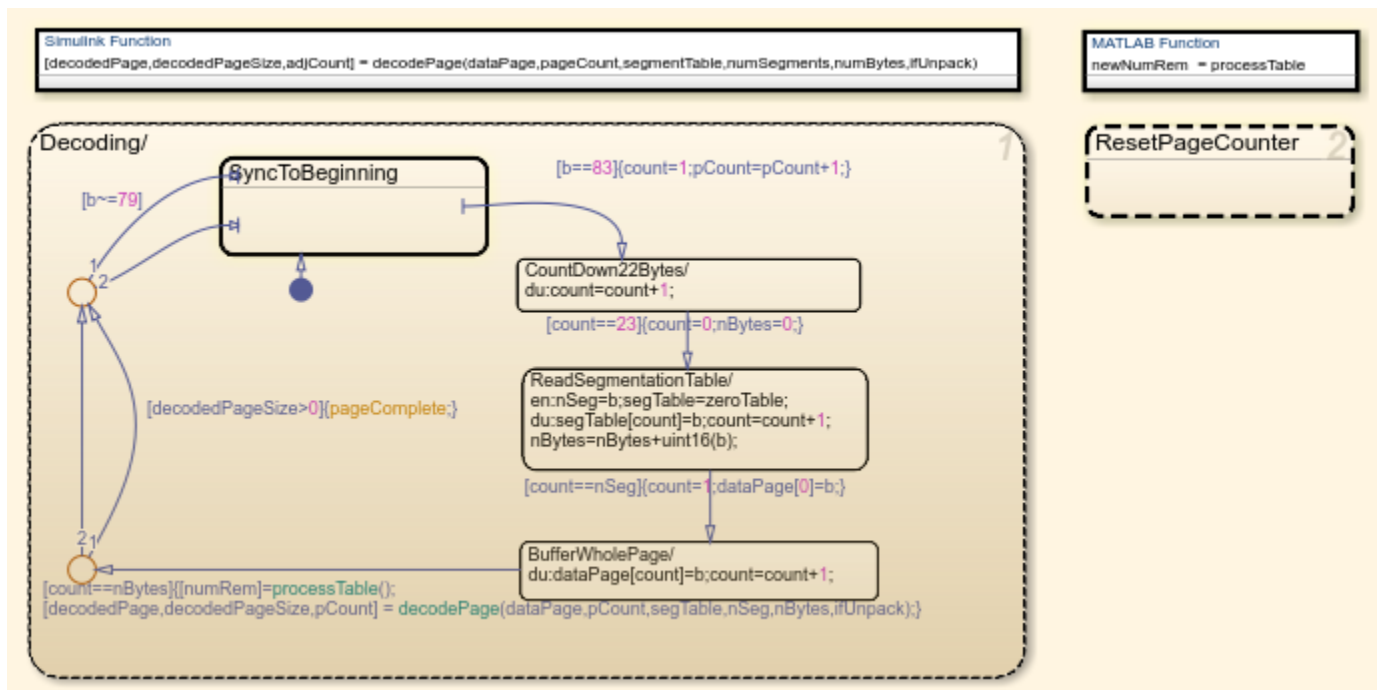
Vorbis provides its own format, known as 'Ogg', to encapsulate logical packets into transport streams. The Ogg format provides mechanisms such as framing, synchronization, positioning, and error correction, which are necessary for data transfer over networks.

Problem Overview and Design Details

The Vorbis decoder in this example implements the specifications of the Vorbis I format, which is a subset of Vorbis. The example model decodes any raw binary OGG file containing a mono or stereo audio signal. The example model has the capability to decode and play back a wide variety of Vorbis audio files in real time.

You can test this example with any Vorbis audio file, or with the included `handel` file. To load the file into the model, replace the file name in the annotated code at the top level of the model with the name of the file you want to test. When this step is complete, click the annotated code to load the new audio file. The model is configured to notify you if the output sampling rate has been changed due to a change in the input data. In this case, the simulation needs to be restarted with the new sample rate.

In order to implement a Vorbis decoder in Simulink®, you must address the variable-sized data packets. This example addresses the variable-sized packets by capturing a whole page of the Ogg bitstream using the 'OggS' synchronization pattern. For practical purposes, a page is assumed to be no larger than 5500 bytes. After obtaining a segmentation table at the beginning of the page, the model extracts logical packets from the remainder of the page. Asynchronous control over the decoding sequence is implemented using the Stateflow chart 'Decode All Pages of Data'.

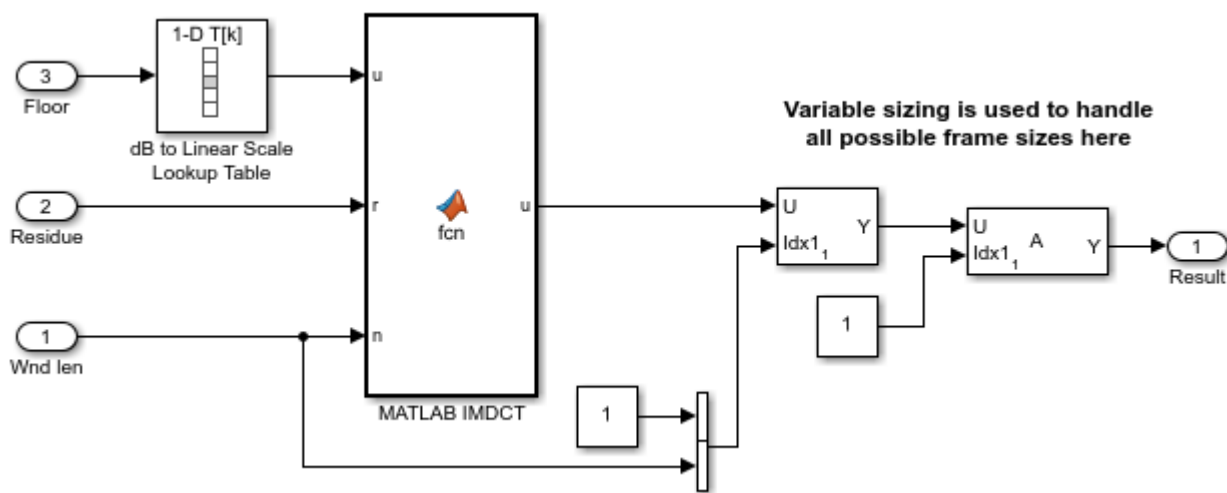


Initially, the chart tries to detect the 'OggS' synchronization pattern and then follow the decoding steps described above. Decoding the page is done with the Simulink function 'decodePage' and then the model immediately goes back to detecting the next 'OggS' sequence. The state 'ResetPageCounter' is added in parallel with the Stateflow algorithm described above to support the looping of the compressed input file for an unlimited number of iterations.

Data pages contain different types of information: header, codebooks, and audio signal data. The 'Read Setup Info', 'Read the Header', and 'Decode Audio' subsystems inside the 'decodePage' Simulink function are responsible for handling each of these different kinds of information.

The decoding process is implemented using MATLAB Function blocks. Most bit-unpacking routines in the example are implemented with MATLAB code.

The recombining of the *floor* and *residue* and the subsequent inverse MDCT (IMDCT) are also implemented with a MATLAB Function block that uses the fast `imdct` function of Audio Toolbox. The variable frame lengths are taken into account using a fixed-size maximum-length frame at the input and output of the Function block, and by using a window length parameter in both the Function block code and a Selector block immediately following the Function block.



The IMDCT transforms the frames back to the time domain, ready to be multiplied by the synthesis window and then combined with an overlap-add operation.

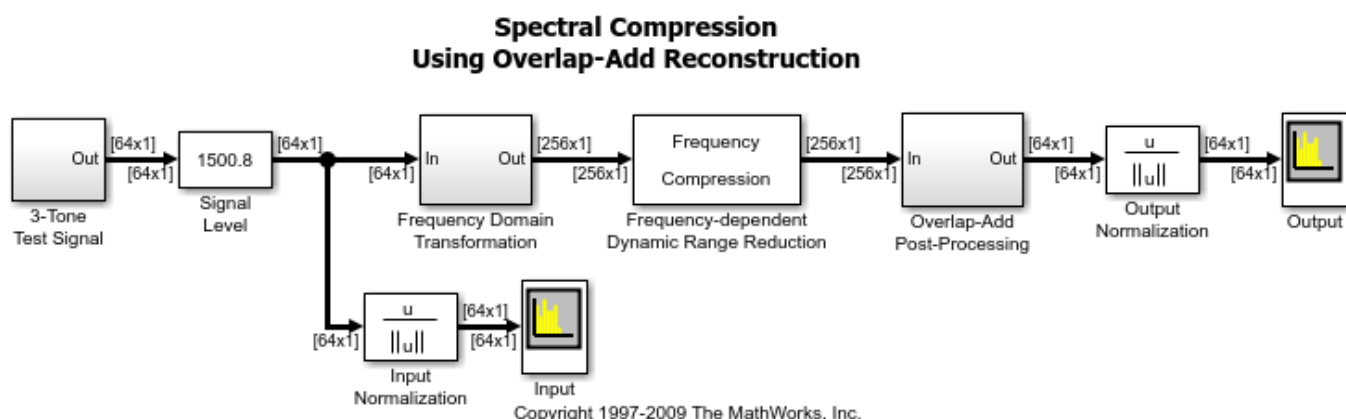
The output block in the top level of the model feeds the output of the decoding block to the audio playback device on your system. The valid portion of the decoded signal is input to the Audio Device Writer block.

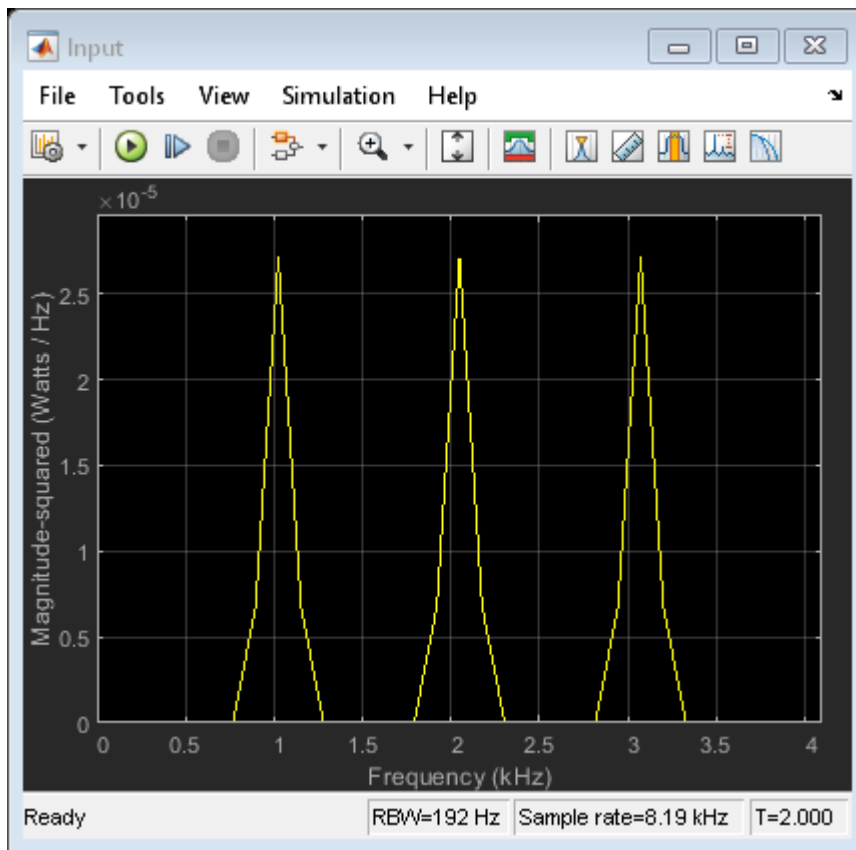
References

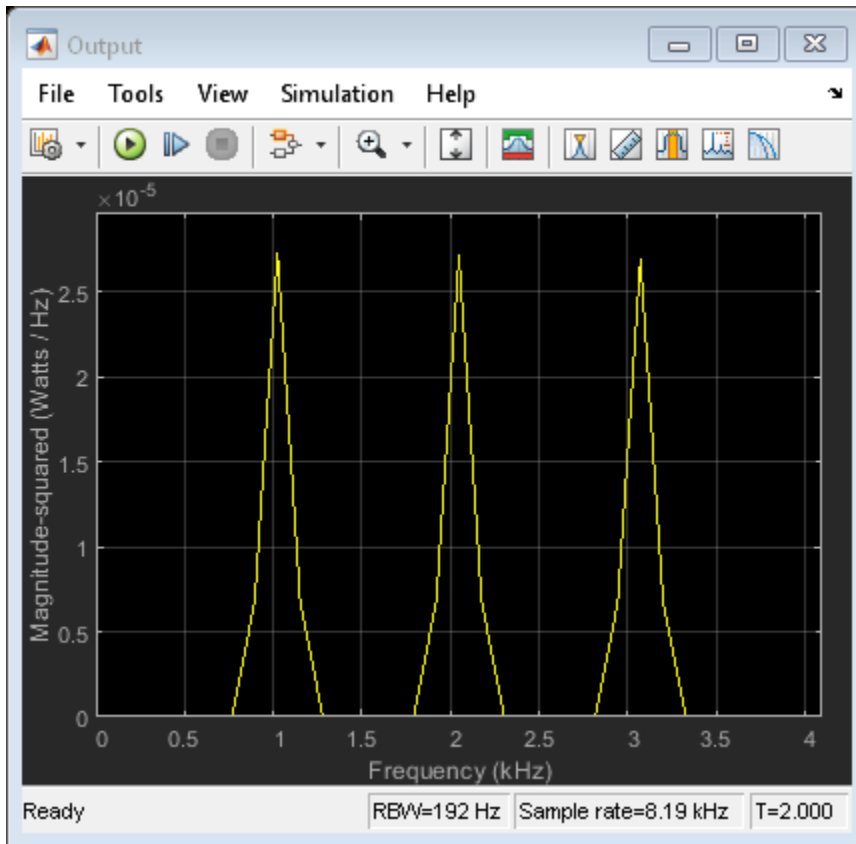
[1] Complete specification of the Vorbis decoder standard https://xiph.org/vorbis/doc/Vorbis_I_spec.html

Dynamic Range Compression Using Overlap-Add Reconstruction

This example shows how to compress the dynamic range of a signal by modifying the range of the magnitude at each frequency bin. This nonlinear spectral modification is followed by an overlap-add FFT algorithm for reconstruction. This system might be used as a speech enhancement system for the hearing impaired. The algorithm in this simulation is derived from a patented system for adaptive processing of telephone voice signals for the hearing impaired originally developed by Alvin M. Terry and Thomas P. Krauss at US West Advanced Technologies Inc., US patent number 5,388,185.







This system decomposes the input signal into overlapping sections of length 256. The overlap is 192 so that every 64 samples, a new section is defined and a new FFT is computed. After the spectrum is modified and the inverse FFT is computed, the overlapping parts of the sections are added together. If no spectral modification is performed, the output is a scaled replica of the input. A reference for the overlap-add method used for the audio signal reconstruction is Rabiner, L. R. and R. W. Schafer. **Digital Processing of Speech Signals**. Englewood Cliffs, NJ: Prentice Hall, 1978, pgs. 274-277.

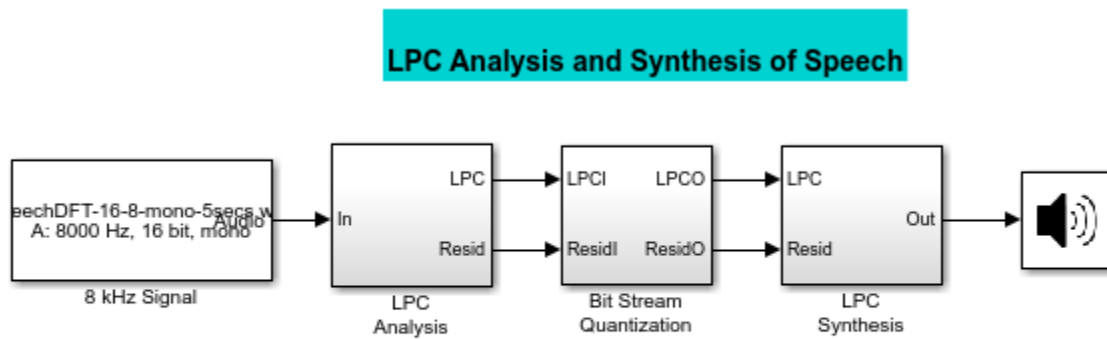
Compression maps the dynamic range of the magnitude at each frequency bin from the range 0 to 100 dB to the range y_{min} to y_{max} dB. y_{min} and y_{max} are vectors in the MATLAB® workspace with one element for each frequency bin; in this case 256. The phase is not altered. This is a non-linear spectral modification. By compressing the dynamic range at certain frequencies, the listener should be able to perceive quieter sounds without being blasted out when they get loud, as in linear equalization.

To use this system to demonstrate frequency-dependent dynamic range compression, start the simulation. After repositioning the input and output figures so you can see them at the same time, change the **Slider Gain** from 1 to 1000 to 10000. Notice the relative heights of the output peaks change as you increase the magnitude.

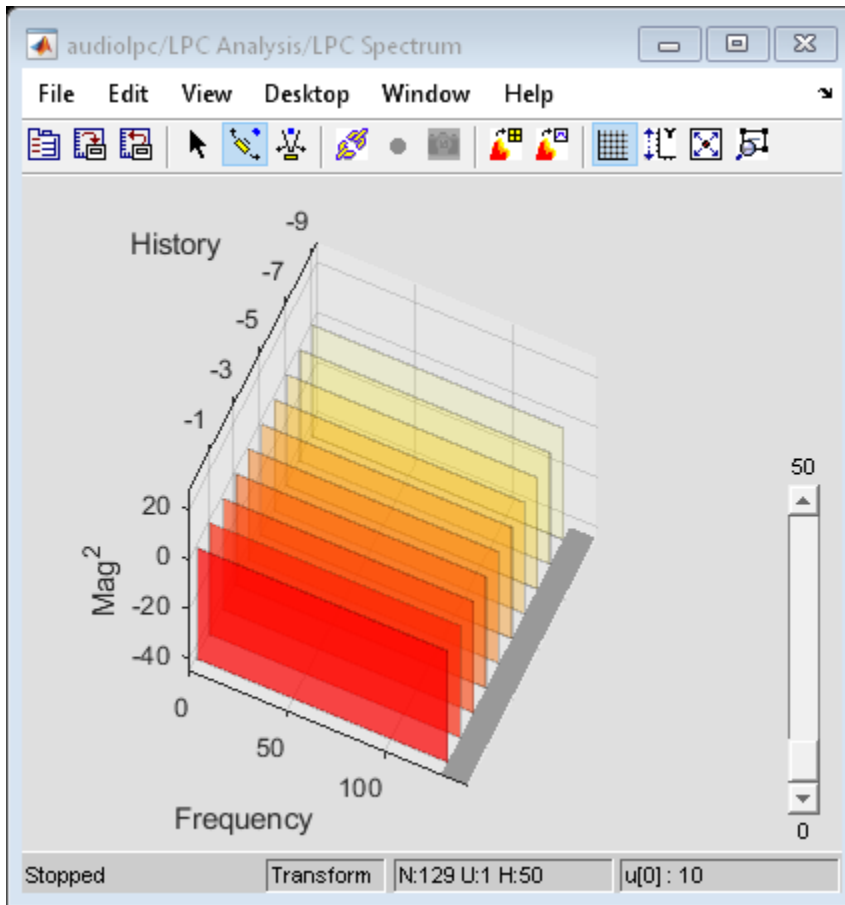
LPC Analysis and Synthesis of Speech

This example shows how to use the Levinson-Durbin and Time-Varying Lattice Filter blocks for low-bandwidth transmission of speech using linear predictive coding.

Example Model



Copyright 2007-2015 The MathWorks, Inc.



Example Description

The example consists of two parts: analysis and synthesis. The analysis portion 'LPC Analysis' is found in the transmitter section of the system. Reflection coefficients and the residual signal are extracted from the original speech signal and then transmitted over a channel. The synthesis portion 'LPC Synthesis', which is found in the receiver section of the system, reconstructs the original signal using the reflection coefficients and the residual signal.

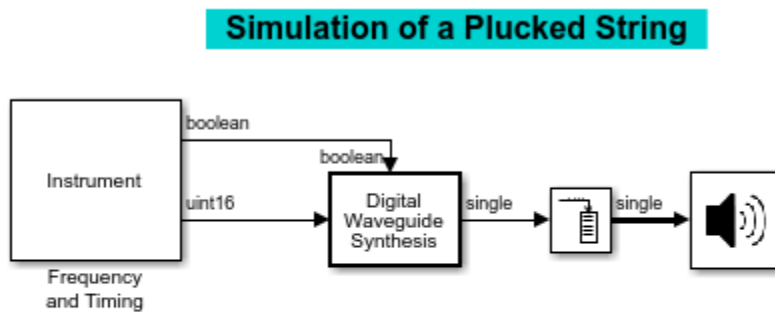
In this simulation, the speech signal is divided into 20 ms frames (160 samples), with an overlap of 10 ms (80 samples). Each frame is windowed using a Hamming window. Eleventh-order autocorrelation coefficients are found, and then the reflection coefficients are calculated from the autocorrelation coefficients using the Levinson-Durbin algorithm. The original speech signal is passed through an analysis filter, which is an all-zero filter with coefficients same as the reflection coefficients obtained above. The output of the filter is the residual signal. This residual signal is passed through a synthesis filter which is the inverse of the analysis filter. The output of the synthesis filter is the original signal. This is played through the 'Audio Device Writer' block.

Simulation of a Plucked String

This example shows how to simulate a plucked string using digital waveguide synthesis.

Introduction

A **digital waveguide** is a computational model for physical media through which sound propagates. They are essentially bidirectional delay lines with some wave impedance. Each delay line can be thought of as a sampled acoustic traveling wave. Using the digital waveguide, a linear one-dimensional acoustic system like the vibration of a guitar string can be modeled.



Copyright 2007-2015 The MathWorks, Inc.

Exploring the Example

The result of the simulation is automatically played back using the **Audio Device Writer** block. To see the implementation, look under the **Digital Waveguide Synthesis** block by right clicking on the block and selecting Mask > Look Under Mask.

Acknowledgements

This Simulink® implementation is based on a MATLAB® file implementation available from Daniel Ellis's home page at Columbia University.

References

The online textbook **Digital Waveguide Modeling of Musical Instruments** by Julius O. Smith III covers significant background related to digital waveguides.

The Harmony Central website also provides useful background information on a variety of related topics.

Audio Phaser Using Multiband Parametric Equalizer

This example shows how to implement a real-time audio "phaser" effect which can be tuned by a user interface (UI). It also shows how to generate a VST plugin for the phaser that you can import into a Digital Audio Workstation (DAW).

Introduction

The phaser is an audio effect produced when an audio signal is passed through one or more notch filters. The center frequencies of the notch filters are typically modulated at some consistent rate to produce a "swirling" effect on the audio. The modulation source is typically a low frequency oscillator such as a sine wave. Different waveform shapes create different phaser effects.

You can use any audio file with this example. However, the phasing effect is more audible with some audio files than with others. A file that is suggested for this example is `RockGuitar-16-44p1-stereo-72secs.wav`. Another option is to use a pink noise source instead of a file.

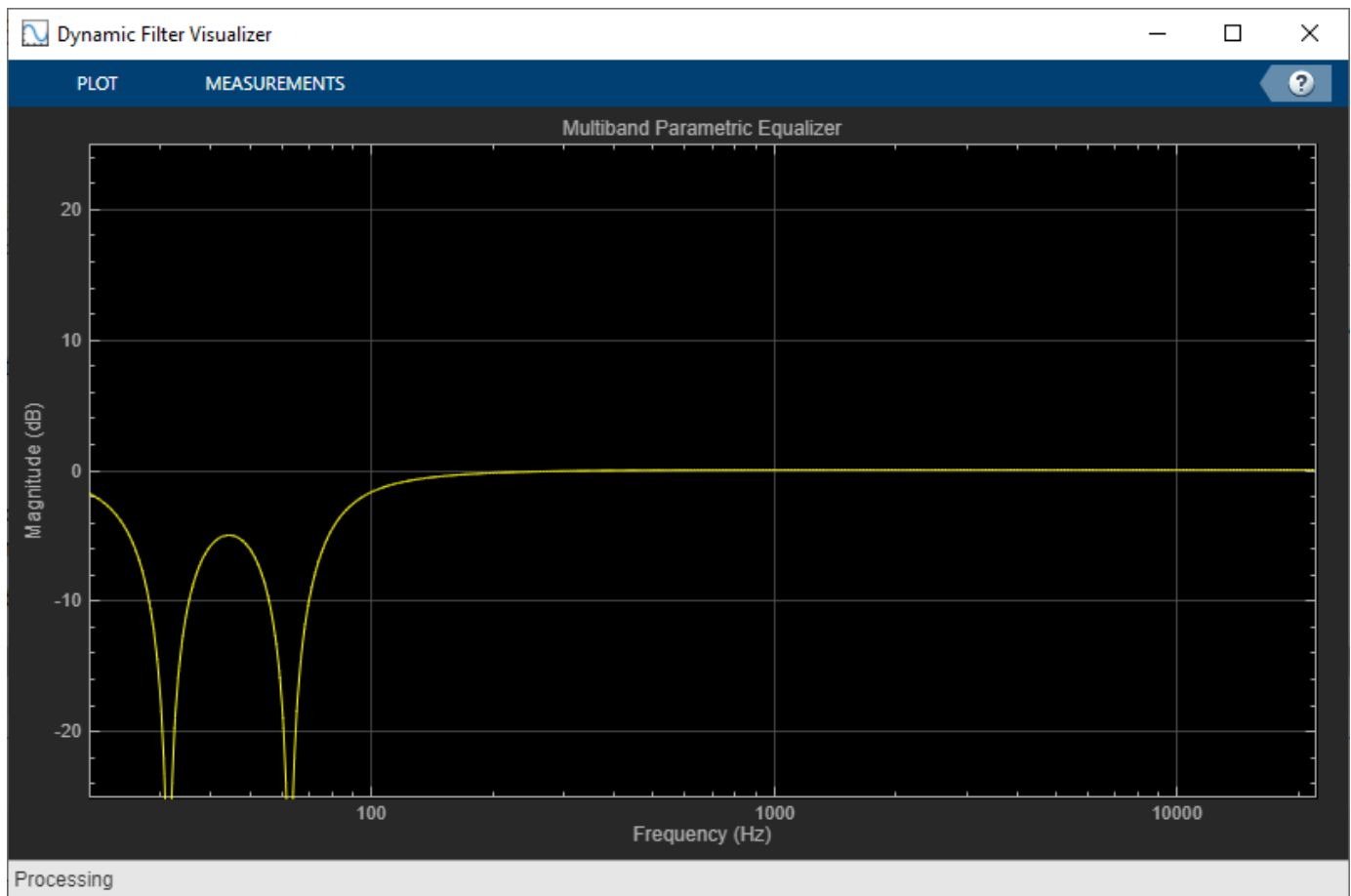
This example uses the `audiopluginexample.Phaser` audio plugin class. The plugin implements a multi-notch filter with notch frequencies modulated by an `audioOscillator`. The multi-notch filter is implemented through the `multibandParametricEQ` System object. The bands of the equalizer can be made to act as individual notch filters by setting their gain to `-inf`.

Test the Phaser

You can test the phaser implemented in `audiopluginexample.Phaser` using Audio Test Bench. The audio test bench sets up the audio file reader and audio device writer objects, and streams the audio through the phaser in a processing loop.

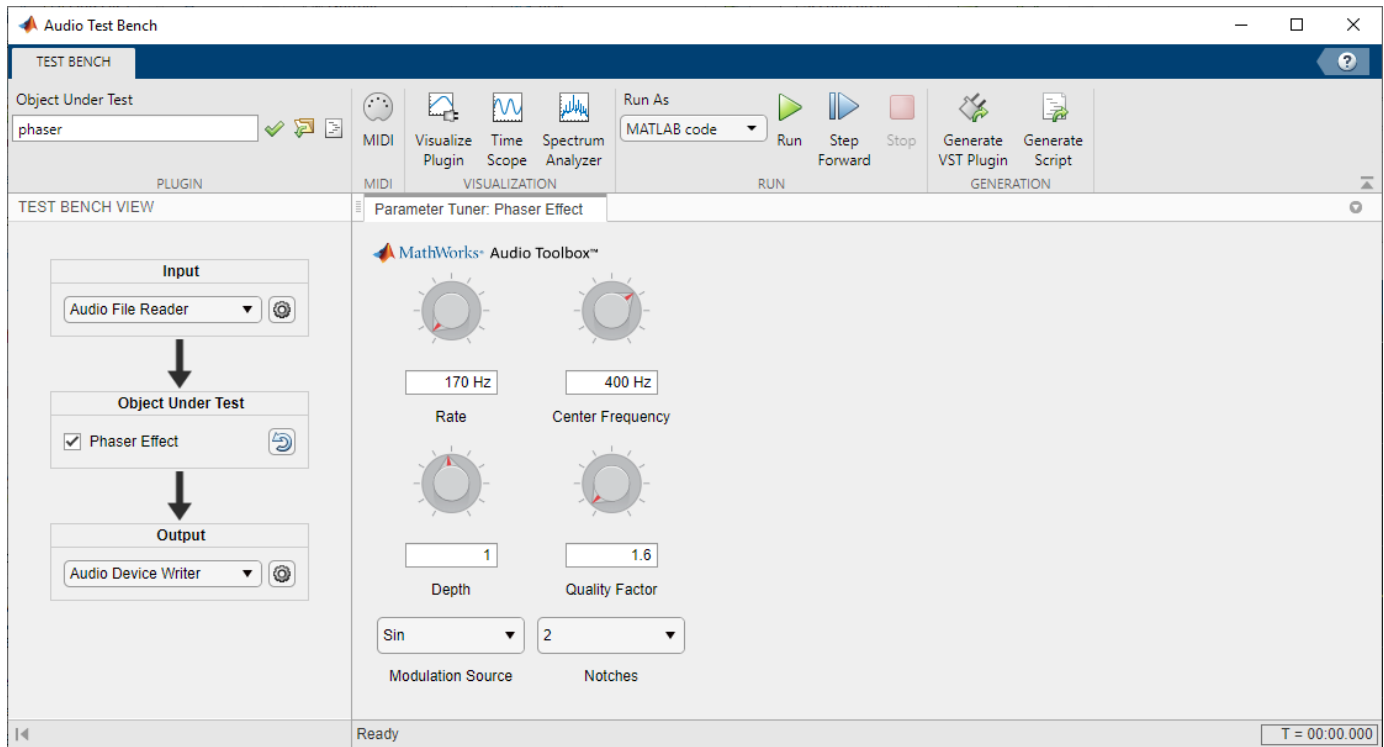
Initialize the phaser and visualize its magnitude response.

```
phaser = audiopluginexample.Phaser;  
visualize(phaser)
```

Launch the **Audio Test Bench**.

```
audioTestBench(phaser)
```



The **Audio Test Bench** enables you to tune the audio phaser using sliders and drop-down menus. Changing slider or drop-down values updates the magnitude response plot of the phaser in real time.

The four sliders are:

- *Rate* - Controls the rate at which the center frequency of the notch filters sweep up and down the audio spectrum.
- *Center Frequency* - Controls the center frequency of the lowest notch. The center frequency of other notches is calculated relative to this value and the modulation source.
- *Depth* - Controls how far the notch frequencies modulate around the center frequency.
- *Qualify Factor* - Sets the quality factor (or "Q") of each notch. A higher Q setting creates a narrower bandwidth notch.

There are also two drop-down menus:

- *Notches* - Sets the number of notch filters. More notches can be used to create a more dramatic effect.
- *Modulation Source* - The waveform that controls the center frequencies of the notch filters. Different waveforms create different sweep sounds.

The audio test bench by default streams audio from a file on disk. You can change it to a sound card microphone/line-in input, or pink noise (useful for testing).

Click the Run button on the UI to start streaming and hear the phaser effect.

Run as VST Plugin

You may find that audio dropouts occur when using higher numbers of notches or high Rate settings. One way to work around this is to generate a VST plugin to take the place of the portion of the code

that performs the actual audio processing. Switch the **Run As** dropdown to **VST Plugin**. On running the simulation now, a VST plugin will be generated and loaded back into MATLAB for use in the simulation.

Generate Audio Plugin

To generate and port a VST plugin to a Digital Audio Workstation, click on the **Generate VST 2 Audio Plugin** button on the toolbar of audio test bench, or run the `generateAudioPlugin` command.

```
generateAudioPlugin audiopluginexample.Phaser
```

Loudness Normalization in Accordance with EBU R 128 Standard

This example shows how to use tools from Audio Toolbox™ to measure loudness, loudness range, and true-peak value. It also shows how to normalize audio to meet the EBU R 128 standard compliance.

Introduction

Volume normalization was traditionally performed by looking at peak signal measurements. However, this method had the drawback that excessively compressed audio could pass a signal-level threshold but still be very loud to hear. The result was a **loudness war**, where recordings tended to be louder than before and inconsistent across genres.

The modern solution to the loudness war is to measure the **perceived loudness** in combination with a **true-peak** level measurement. International standards like ITU BS.1770-4, EBU R 128, and ATSC A/85 have been developed to standardize loudness measurements based on the power of the audio signal. Many countries have already passed legislations for compliance with broadcast standards on loudness levels.

In this example, you measure loudness and supplementary parameters for both offline (file-based) and live (streaming) audio signals. You also see ways to normalize audio to be compliant with target levels.

EBU R 128 Standard

Audio Toolbox enables you to measure loudness and associated parameters according to the EBU R 128 standard. This standard defines the following measures of loudness:

- **Momentary loudness:** Uses a sliding window of length 400 ms.
- **Short-term loudness:** Uses a sliding window of length 3 s.
- **Integrated loudness:** Aggregate loudness from start till end.
- **Loudness range:** Quantifies variation of loudness on a macroscopic timescale.
- **True-peak value:** Peak sample level of interpolated signal.

For a more detailed description of these parameters, refer to the documentation for EBU R 128 standard.

Offline Loudness Measurement and Normalization

For cases where you already have the recorded audio samples, you can use the `integratedLoudness` function to measure loudness. It returns the integrated loudness, in units of LUFS, and loudness range, in units of LU, of the complete audio file.

```
[x, fs] = audioread('RockGuitar-16-44p1-stereo-72secs.wav');  
[loudness, LRA] = integratedLoudness(x,fs);  
fprintf('Loudness before normalization: %.1f LUFS\n',loudness)
```

```
Loudness before normalization: -8.2 LUFS
```

EBU R 128 defines the target loudness level to be -23 LUFS. The loudness of the audio file is clearly above this level. A simple level reduction operation can be used to normalize the loudness.

```
target = -23;  
gaindB = target - loudness;
```

```
gain = 10^(gaindB/20);
xn = x.*gain;
audiowrite('RockGuitar_normalized.wav',xn,fs)
```

The loudness of the new audio file is at the target level.

```
[x, fs] = audioread('RockGuitar_normalized.wav');
loudness = integratedLoudness(x,fs);
fprintf('Loudness after normalization: %.1f LUFS\n',loudness)
```

```
Loudness after normalization: -23.0 LUFS
```

Live Loudness Measurement and Normalization

For streaming audio, EBU R 128 defines momentary and short-term loudness. You can use the `loudnessMeter` System object to measure momentary loudness, short-term loudness, integrated loudness, loudness range, and true-peak value of a live audio signal.

First, stream the audio signal to your sound card and measure its loudness using `loudnessMeter`. The **visualize** method of `loudnessMeter` opens a user interface (UI) that displays all the loudness-related measurements as the simulation progresses.

```
reader = dsp.AudioFileReader('RockGuitar-16-44p1-stereo-72secs.wav', ...
    'SamplesPerFrame',1024);
fs = reader.SampleRate;
inputLoudness = loudnessMeter('SampleRate',fs);
player = audioDeviceWriter('SampleRate',fs);
runningMax = dsp.MovingMaximum('SpecifyWindowLength',false);
visualize(inputLoudness)
while ~isDone(reader)
    audioIn = reader();
    [loudness,~,~,tp] = inputLoudness(audioIn);
    maxTP = runningMax(tp);
    player(audioIn);
end
fprintf('Max true-peak value before normalization: %.1f dBTP\n',maxTP(end))
```

```
Max true-peak value before normalization: -0.3 dBTP
```

```
release(reader)
release(player)
```

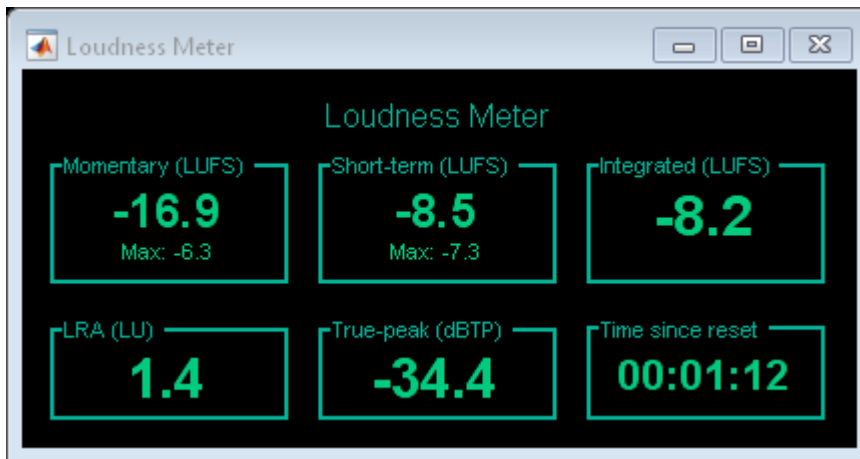
As you can see on the UI, the loudness of the audio stream is clearly above the -23 LUFS threshold. Its maximum true-peak level of -0.3 dBTP is also above the threshold of -1 dBTP specified by EBU R 128. Normalizing the loudness of a live audio stream is trickier than normalizing the loudness of a file. One way to help get the loudness value close to a target threshold is to use an Automatic Gain Controller (AGC). In the following code, you use the `audioexample.AGC` System object to normalize the power of an audio signal to -23 dB. The AGC estimates the audio signal's power by looking at the previous 400 ms, which is the window size used to calculate momentary loudness. There are two loudness meters used in this example - one for the input to AGC and one for the output from AGC. The UIs for the two loudness meters may launch at the same location on your screen, so you will have to move one to the side to compare the measured loudness before and after AGC.

```
outputLoudness = loudnessMeter('SampleRate',fs);
gainController = audioexample.AGC('DesiredOutputPower',-23, ...
    'AveragingLength',0.4*fs,'MaxPowerGain',20);
reset(inputLoudness) % Reuse the same loudness meter from before
```

```

reset(runningMax)
visualize(inputLoudness)
visualize(outputLoudness)
while ~isDone(reader)
    audioIn = reader();
    loudnessBeforeNorm = inputLoudness(audioIn);
    [audioOut, gain] = gainController(audioIn);
    [loudnessAfterNorm,~,~,tp] = outputLoudness(audioOut);
    maxTP = runningMax(tp);
    player(audioOut);
end

```



```
fprintf('Max true-peak value after normalization: %.1f dBTP\n',maxTP(end))
```

```
Max true-peak value after normalization: 8.3 dBTP
```

```
release(reader)
```

```
release(player)
```

Using AGC not only brought the loudness of the audio close to the target of -23 LUFS, but it also got the maximum true-peak value below the allowed -1 dBTP. In some cases, the maximum true-peak value remains above -1 dBTP although the loudness is at or below -23 LUFS. For such scenarios, you can pass the audio through a limiter.

Multistage Sample-Rate Conversion of Audio Signals

This example shows how to use a multistage/multirate approach to sample rate conversion between different audio sampling rates.

The example uses `dsp.SampleRateConverter`. This component automatically determines how many stages to use and designs the filter required for each stage in order to perform the sample rate conversion in a computationally efficient manner.

This example focuses on converting an audio signal sampled at 96 kHz (DVD quality) to an audio signal sampled at 44.1 kHz (CD quality). The comparison is done using data sampled at 96 kHz available online at <http://src.infinetwave.ca/>. In this example, the 96 kHz chirp signal is generated locally so that no download is needed.

Setup

Define some parameters to be used throughout the example.

```
frameSize = 64;
inFs      = 96e3;
```

Reading the 96 kHz File

The website above has 3 sets of files at different qualities in order to perform the comparison. In this example the focus will be on one of the files only: `Swept_int.wav`. This file contains a chirp sine wave sweeping from 0 Hz to 48 kHz over the course of 8 seconds. The format of the file is 32-bit integers, given it a very high dynamic range.

Here you create a `System` object to read from the audio file and determine the file's audio sampling rate. If you want to use the wav file instead of `dsp.Chirp`, uncomment the lines below and skip the call to `dsp.Chirp`.

```
source = dsp.AudioFileReader('Swept_int.wav', ... 'SamplesPerFrame',frameSize, ...
'OutputDataType','double');
```

Generating the 96 kHz Signal

In lieu of downloading the `Swept_int.wav` file, you can also generate the chirp signal using `dsp.Chirp` as follows:

```
source = dsp.Chirp('InitialFrequency',0,'TargetFrequency',48e3, ...
'SweepTime',8,'TargetTime',8,'SampleRate',inFs, ...
'SamplesPerFrame',frameSize,'Type','Quadratic');
```

Create Spectrum Analyzers

Create two spectrum analyzers. These will be used to visualize the frequency content of the original signal as well as that of the signals converted to 44.1 kHz.

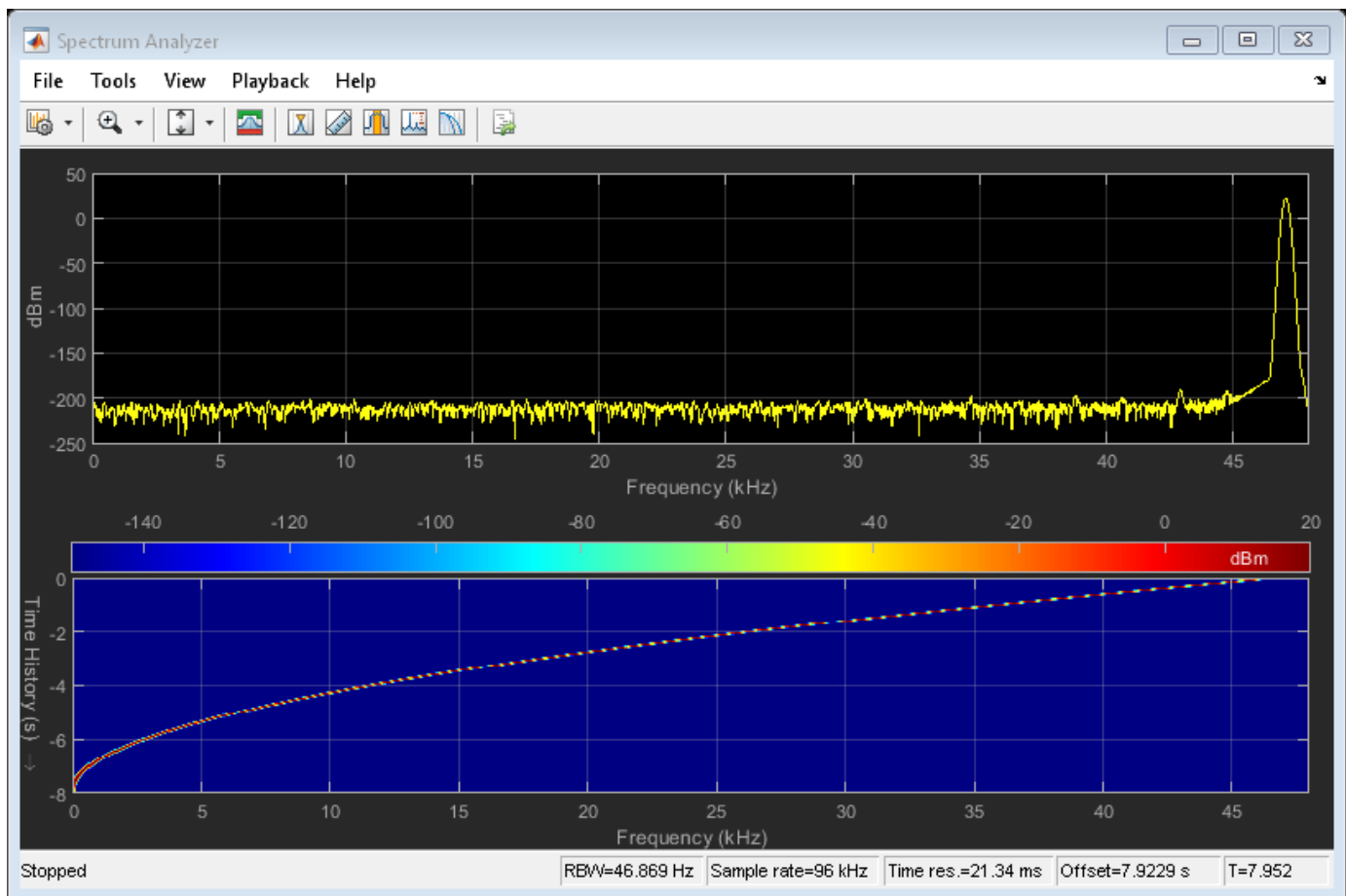
```
SpectrumAnalyzer44p1 = dsp.SpectrumAnalyzer( ...
'SampleRate',44100, ...
'ViewType','Spectrum and spectrogram', ...
'TimeSpanSource','Property','TimeSpan',8, ...
'Window','Kaiser','SidelobeAttenuation',220, ...
'YLimits',[-250, 50],'ColorLimits',[-150, 20], ...
'PlotAsTwoSidedSpectrum',false);
```

```
SpectrumAnalyzer96 = dsp.SpectrumAnalyzer( ...
    'SampleRate',96000, ...
    'ViewType','Spectrum and spectrogram', ...
    'TimeSpanSource','Property','TimeSpan',8, ...
    'Window','Kaiser','SidelobeAttenuation',220, ...
    'YLimits',[-250, 50],'ColorLimits',[-150, 20], ...
    'PlotAsTwoSidedSpectrum',false);
```

Spectrum of Original Signal Sampled at 96 kHz

The loop below plots the spectrogram and power spectrum of the original 96 kHz signal. The chirp signal starts at 0 and sweeps to 48 kHz over a simulated time of 8 seconds.

```
NFrames = 8*inFs/frameSize;
for k = 1:NFrames
    sig96 = source(); % Source
    SpectrumAnalyzer96(sig96); % Spectrogram
end
release(source)
release(SpectrumAnalyzer96)
```



Setting up the Sample Rate Converter

In order to convert the signal, `dsp.SampleRateConverter` is used. A first attempt sets the bandwidth of interest to 40 kHz, i.e. to cover the range [-20 kHz, 20 kHz]. This is the usually accepted range that is audible to humans. The stopband attenuation for the filters to be used to remove spectral replicas and aliased replicas is left at the default value of 80 dB.

```
BW40 = 40e3;
OutFs = 44.1e3;
SRC40kHz80dB = dsp.SampleRateConverter('Bandwidth',BW40, ...
    'InputSampleRate',inFs,'OutputSampleRate',OutFs);
```

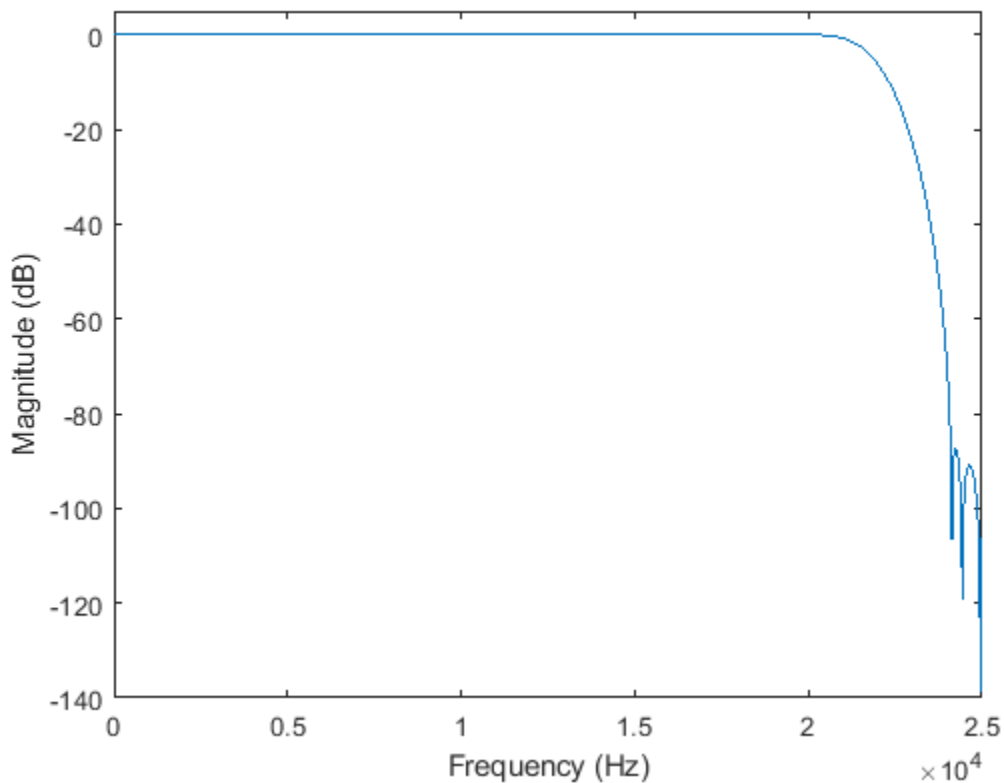
Analysis of the Filters Involved in the Conversion

Use `info` to get information on the filters that are designed to perform the conversion. This reveals that the conversion will be performed in two steps. The first step involves a decimation by two filter which converts the signal from 96 kHz to 48 kHz. The second step involves an FIR rate converter that interpolates by 147 and decimates by 160. This results in the 44.1 kHz required. The `freqz` command can be used to visualize the combined frequency response of the two stages involved. Zooming in reveals that the passband extends up to 20 kHz as specified and that the passband ripple is in the milli-dB range (less than 0.003 dB).

```
info(SRC40kHz80dB)
[H80dB,f] = freqz(SRC40kHz80dB,0:10:25e3);
plot(f,20*log10(abs(H80dB)/norm(H80dB,inf)))
xlabel('Frequency (Hz)')
ylabel('Magnitude (dB)')
axis([0 25e3 -140 5])
```

```
ans =
```

```
'Overall Interpolation Factor' : 147
'Overall Decimation Factor'   : 320
'Number of Filters'           : 2
'Multiplications per Input Sample': 42.334375
'Number of Coefficients'      : 8618
'Filters:'
    Filter 1:
        dsp.FIRDecimator      - Decimation Factor : 2
    Filter 2:
        dsp.FIRRateConverter  - Interpolation Factor: 147
                               - Decimation Factor  : 160
```



Asynchronous Buffer

The sample rate conversion from 96 kHz to 44.1 kHz produces 147 samples for every 320 input samples. Because the chirp signal is generated with frames of 64 samples, an asynchronous buffer is needed. The chirp signal is written 64 samples at a time, and whenever there are enough samples buffered, 320 of them are read and fed to the sample rate converter.

```
buff = dsp.AsyncBuffer;
```

Main Processing Loop

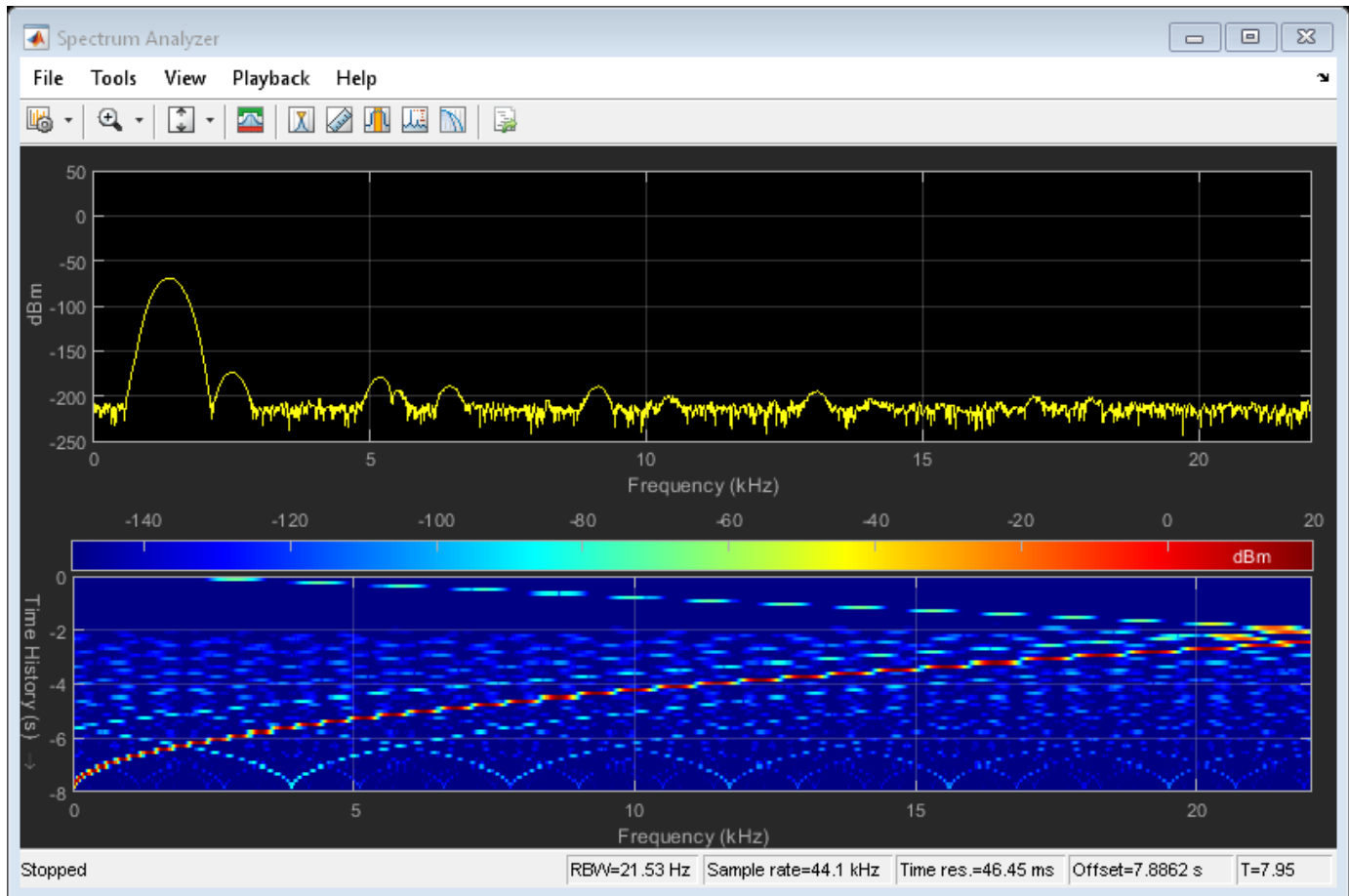
The loop below performs the sample rate conversion in streaming fashion. The computation is fast enough to operate in real time if need be.

The spectrogram and power spectrum of the converted signal are plotted. The extra lines in the spectrogram correspond to spectral aliases/images remaining after filtering. The replicas are attenuated by better than 80 dB as can be verified with the power spectrum plot.

```
srcFrameSize = 320;
for k = 1:Nframes
    sig96 = source();           % Generate chirp
    write(buff,sig96);          % Buffer data
    if buff.NumUnreadSamples >= srcFrameSize
        sig96buffered = read(buff,srcFrameSize);
        sig44p1 = SRC40kHz80dB(sig96buffered); % Convert sample-rate
        SpectrumAnalyzer44p1(sig44p1); % View spectrum of converted signal
    end
```

```
end
```

```
release(source)
release(SpectrumAnalyzer44p1)
release(buff)
```



A More Precise Sample Rate Converter

In order to improve the sample rate converter quality, two changes can be made. First, the bandwidth can be extended from 40 kHz to 43.5 kHz. This in turn requires filters with a sharper transition. Second, the stopband attenuation can be increased from 80 dB to 160 dB. Both these changes come at the expense of more filter coefficients over all as well as more multiplications per input sample.

```
BW43p5 = 43.5e3;
SRC43p5kHz160dB = dsp.SampleRateConverter('Bandwidth',BW43p5, ...
    'InputSampleRate',inFs,'OutputSampleRate',outFs, ...
    'StopbandAttenuation',160);
```

Analysis of the Filters Involved in the Conversion

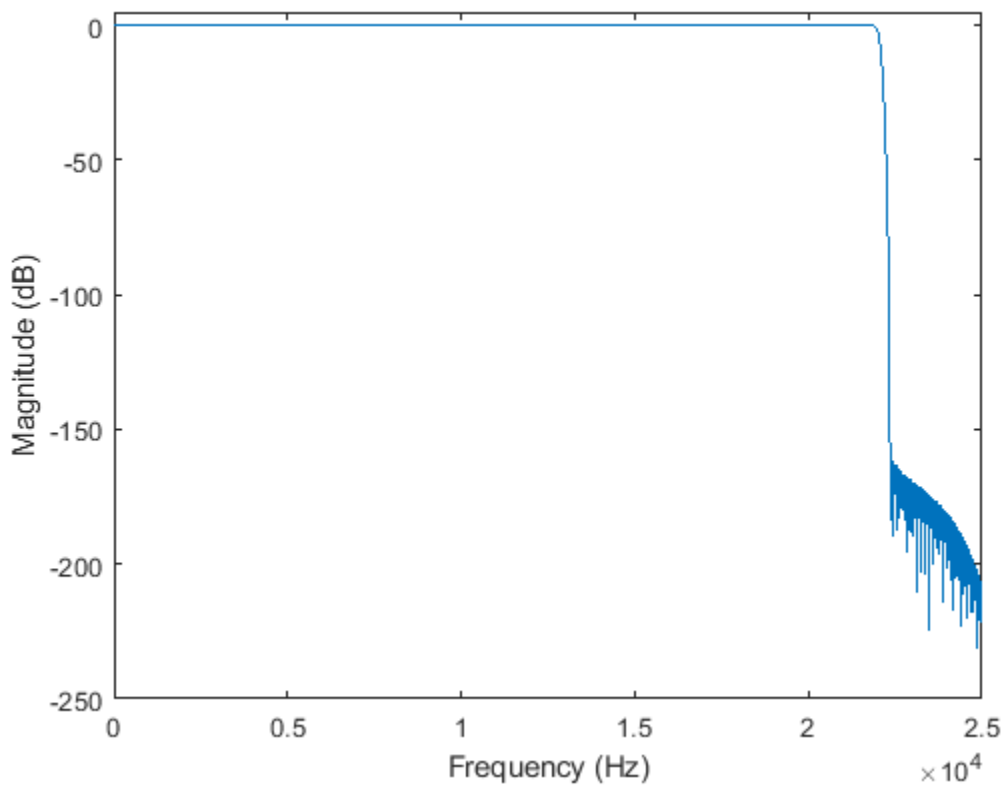
The previous sample rate converter involved 8618 filter coefficients and a computational cost of 42.3 multiplications per input sample. By increasing the bandwidth and stopband attenuation, the cost increases substantially to 123896 filter coefficients and 440.34 multiplications per input sample. The frequency response reveals a much sharper filter transition as well as larger stopband attenuation.

Moreover, the passband ripple is now in the micro-dB scale. NOTE: this implementation involves the design of very long filters which takes several minutes to complete. However, this is a one time cost which happens offline (before the actual sample rate conversion).

```
info(SRC43p5kHz160dB)
[H160dB,f] = freqz(SRC43p5kHz160dB,0:10:25e3);
plot(f,20*log10(abs(H160dB)/norm(H160dB,inf)));
xlabel('Frequency (Hz)')
ylabel('Magnitude (dB)')
axis([0 25e3 -250 5])
```

ans =

```
'Overall Interpolation Factor    : 147
Overall Decimation Factor      : 320
Number of Filters              : 2
Multiplications per Input Sample: 440.340625
Number of Coefficients         : 123896
Filters:
  Filter 1:
    dsp.FIRDecimator    - Decimation Factor : 2
  Filter 2:
    dsp.FIRRateConverter - Interpolation Factor: 147
                        - Decimation Factor : 160
```



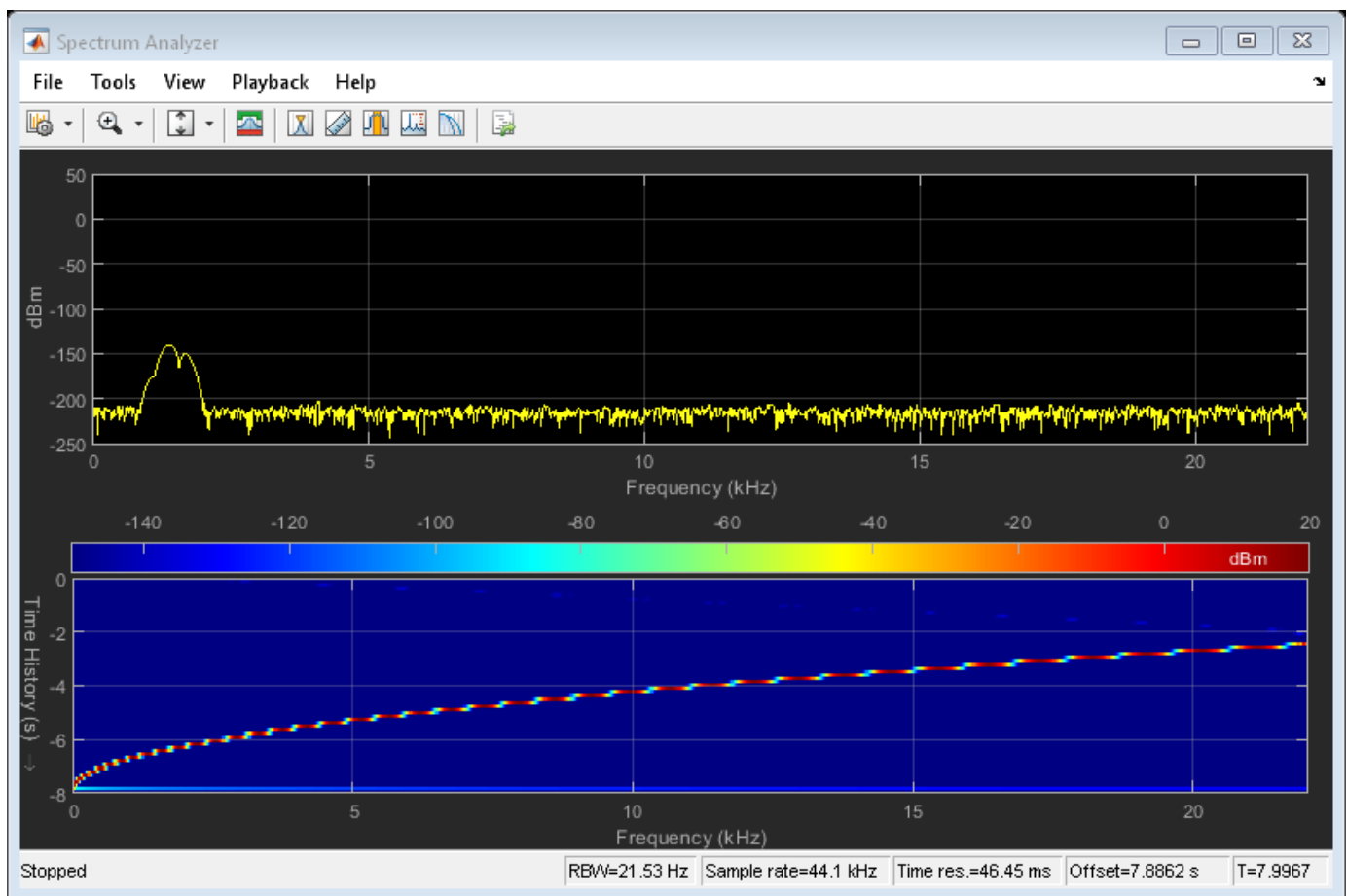
Main Processing Loop

The processing is repeated with the more precise sample rate converter.

Once again the spectrogram and power spectrum of the converted signal are plotted. Notice that the imaging/aliasing is attenuated enough that they are not visible in the spectrogram. The power spectrum shows spectral aliases attenuated by more than 160 dB (the peak is at about 20 dB).

```
for k = 1:NFrames
    sig96 = source();           % Generate chirp
    over = write(buff,sig96);   % Buffer data
    if buff.NumUnreadSamples >= srcFrameSize
        [sig96buffered,under] = read(buff,srcFrameSize);
        sig44p1 = SRC43p5kHz160dB(sig96buffered); % Convert sample-rate
        SpectrumAnalyzer44p1(sig44p1); % View spectrum of converted signal
    end
end

release(source)
release(SpectrumAnalyzer44p1)
release(buff)
```



Graphic Equalization

This example demonstrates two forms of graphic equalizers constructed using building blocks from Audio Toolbox™. It also shows how to export them as VST plugins to be used in a Digital Audio Workstation (DAW).

Graphic Equalizers

Equalizers are commonly used by audio engineers and consumers to adjust the frequency response of audio. For example, they can be used to compensate for bias introduced by speakers, or to add bass to a song. They are essentially a group of filters designed to provide a custom overall frequency response.

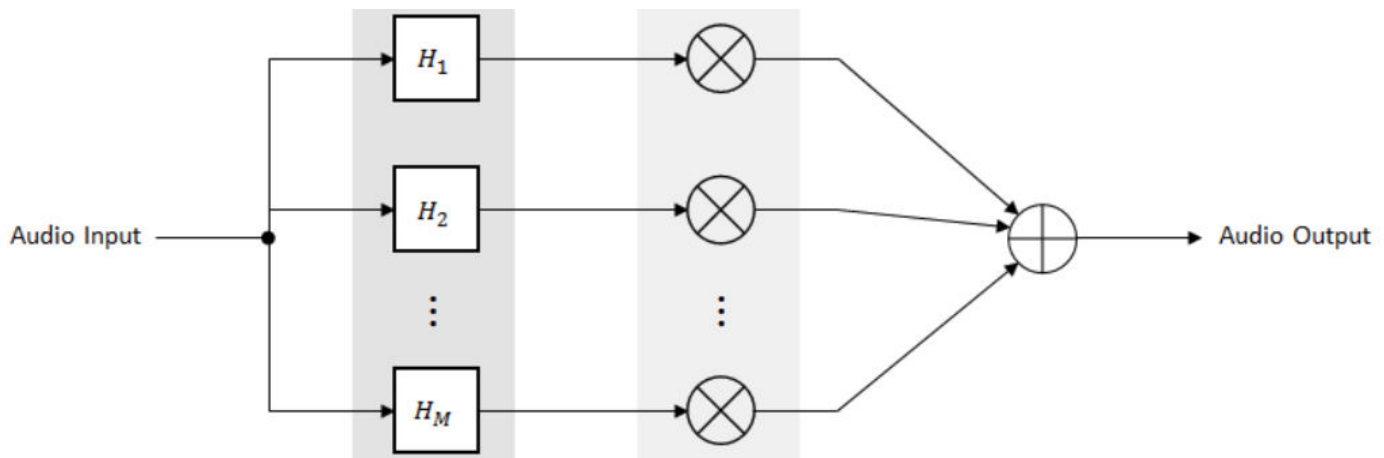
One of the more sophisticated equalization techniques is known as parametric equalization. Parametric equalizers provide control over three filter parameters: center frequency, bandwidth, and gain. Audio Toolbox™ provides the `multibandParametricEQ` System object and the Parametric EQ block for parametric equalization.

While parametric equalizers are useful when you want to fine-tune the frequency response, there are simpler equalizers for cases when you need fewer controls. Octave, two-third octave, and one-third octave have emerged as common bandwidths for equalizers based on the behavior of the human ear. Standards like ISO 266:1997(E), ANSI S1.11-2004, and IEC 61672-1:2013 define center frequencies for octave and fractional octave filters. This leaves only one parameter to tune: filter gain. *Graphic equalizers* provide control over the gain parameter while using standard center frequencies and common bandwidths.

In this example, you use two implementations of graphic equalizers. They differ in arrangement of constituent filters: One uses a bank of parallel octave- or fractional octave-band filters, and the other uses a cascade of biquad filters. The center frequencies in both implementations follow the ANSI S1.11-2004 standard.

Graphic Equalizers with Parallel Filters

One way to construct a graphic equalizer is to place a group of bandpass filters in parallel. The bandwidth of each filter is octave or fractional octave, and their center frequency is set so that together they cover the audio frequency range of [20, 20000] Hz.



The transfer function is a sum of transfer function of the branches.

$$H_{eq}(z) = \sum_{m=1}^M G_m H_m$$

You can tune the gains to boost or cut the corresponding frequency band while the simulation runs. Because the gains are independent of the filter design, tuning the gains does not have a significant computational cost. The parallel filter structure is well suited to parallel hardware implementation. The magnitude response of the bandpass filters should be close to zero at all other frequencies outside its bandwidth to avoid interaction between the filters. However, this is not practical, leading to inter-band interference.

You can use the `graphicEQ` System object to implement a graphic equalizer with a parallel structure.

```
eq = graphicEQ('Structure','Parallel')
```

```
eq =
  graphicEQ with properties:
    EQOrder: 2
    Bandwidth: '1 octave'
    Structure: 'Parallel'
    Gains: [0 0 0 0 0 0 0 0 0 0]
    SampleRate: 44100
```

This designs a parallel implementation of second order filters with 1-octave bandwidth. It takes ten octave filters to cover the range of audible frequencies. Each element of the `Gains` property controls the gain of one branch of the parallel configuration.

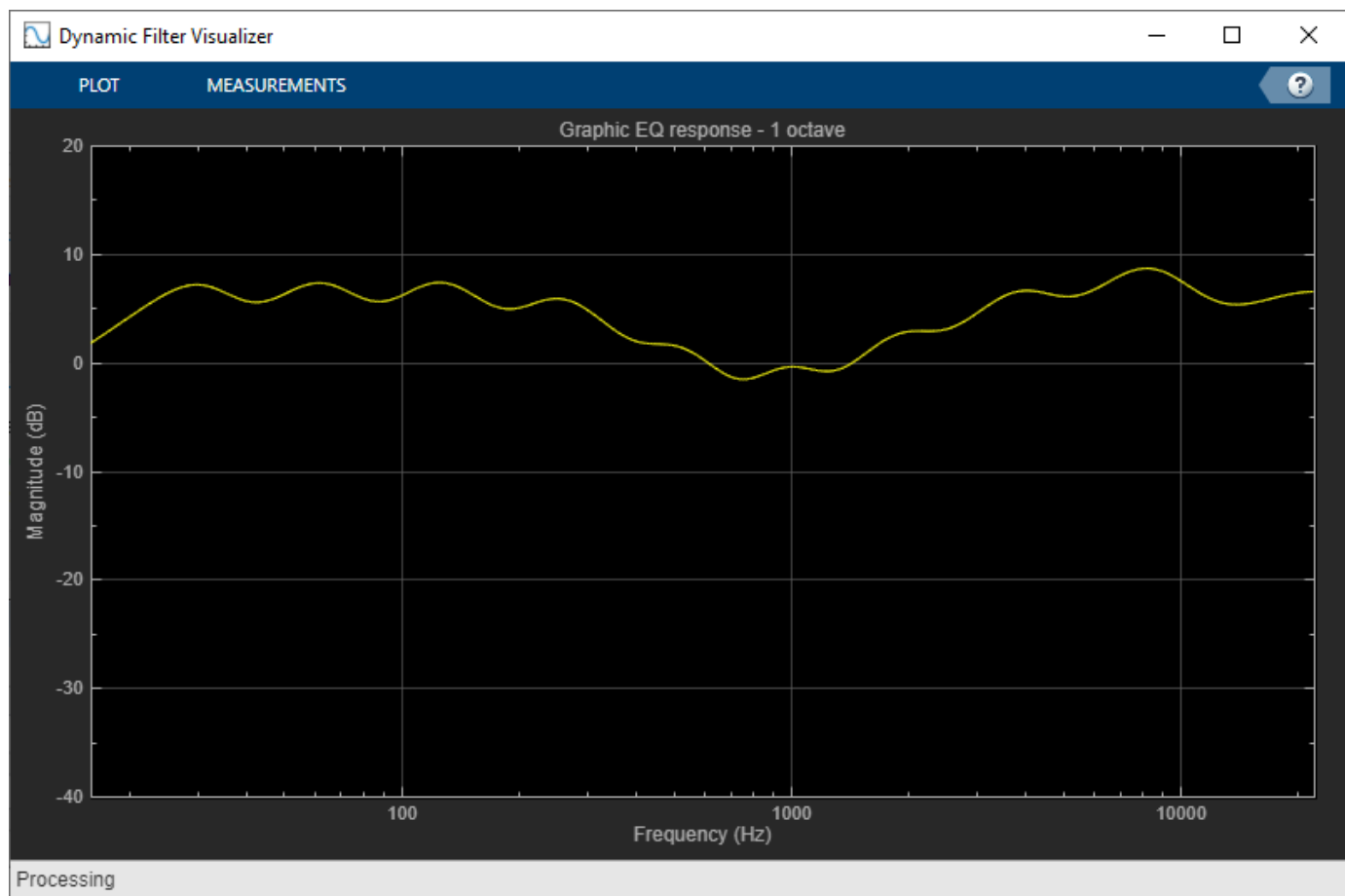
Configure the object you created to boost low and high frequencies, similar to a *rock* preset.

```
eq.Gains = [4, 4.2, 4.6, 2.7, -3.7, -5.2, -2.5, 2.3, 5.4, 6.5]
```

```
eq =
  graphicEQ with properties:
    EQOrder: 2
    Bandwidth: '1 octave'
    Structure: 'Parallel'
    Gains: [4 4.2000 4.6000 2.7000 -3.7000 -5.2000 -2.5000 2.3000 5.4000 6.5000]
    SampleRate: 44100
```

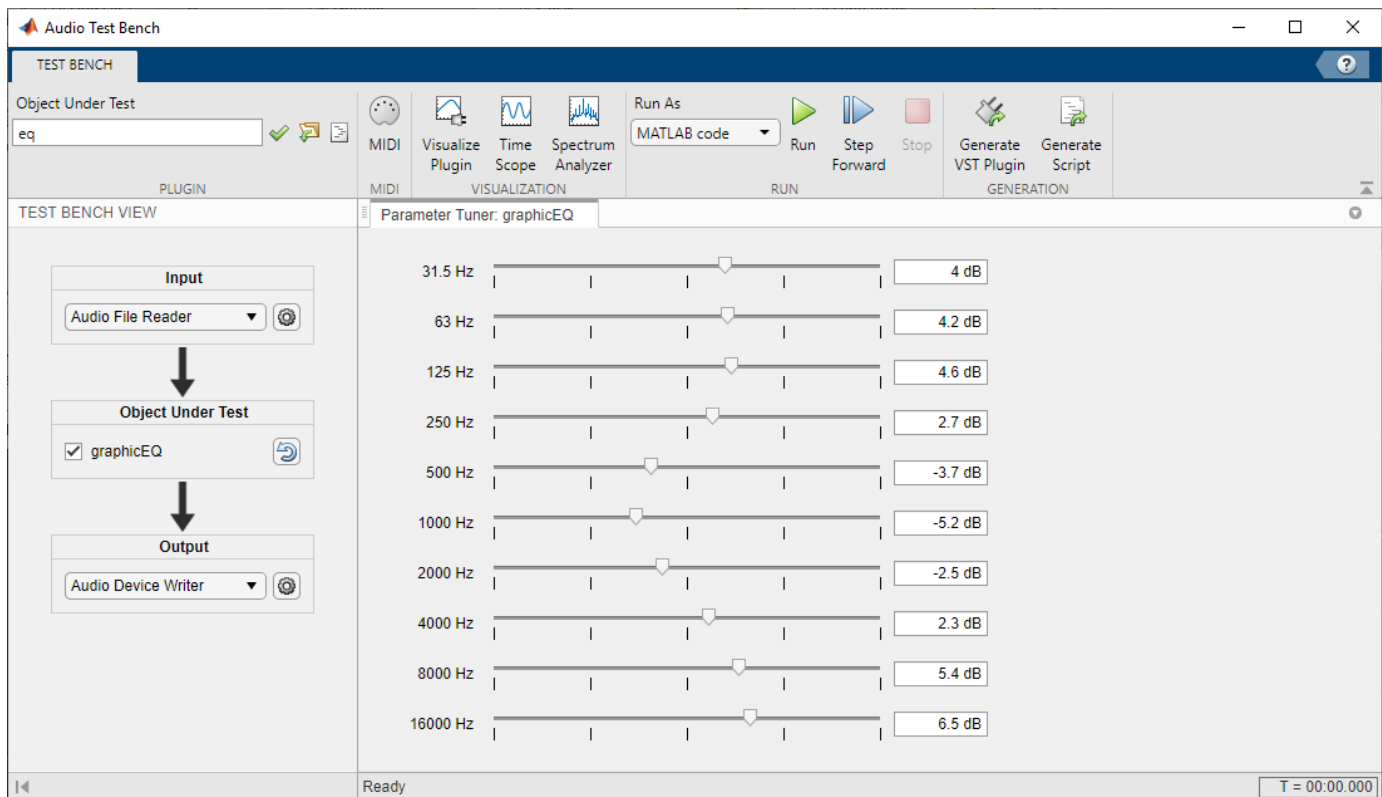
Call `visualize` to view the magnitude response of the equalizer design.

```
visualize(eq)
```



You can test the equalizer implemented in `graphicEQ` using Audio Test Bench. The audio test bench sets up the audio file reader and audio device writer objects, and streams the audio through the equalizer in a processing loop. It also assigns a slider to each gain value and labels the center frequency it corresponds to, so you can easily change the gain and hear its effect. Modifying the value of the slider simultaneously updates the magnitude response plot.

```
audioTestBench(eq)
```

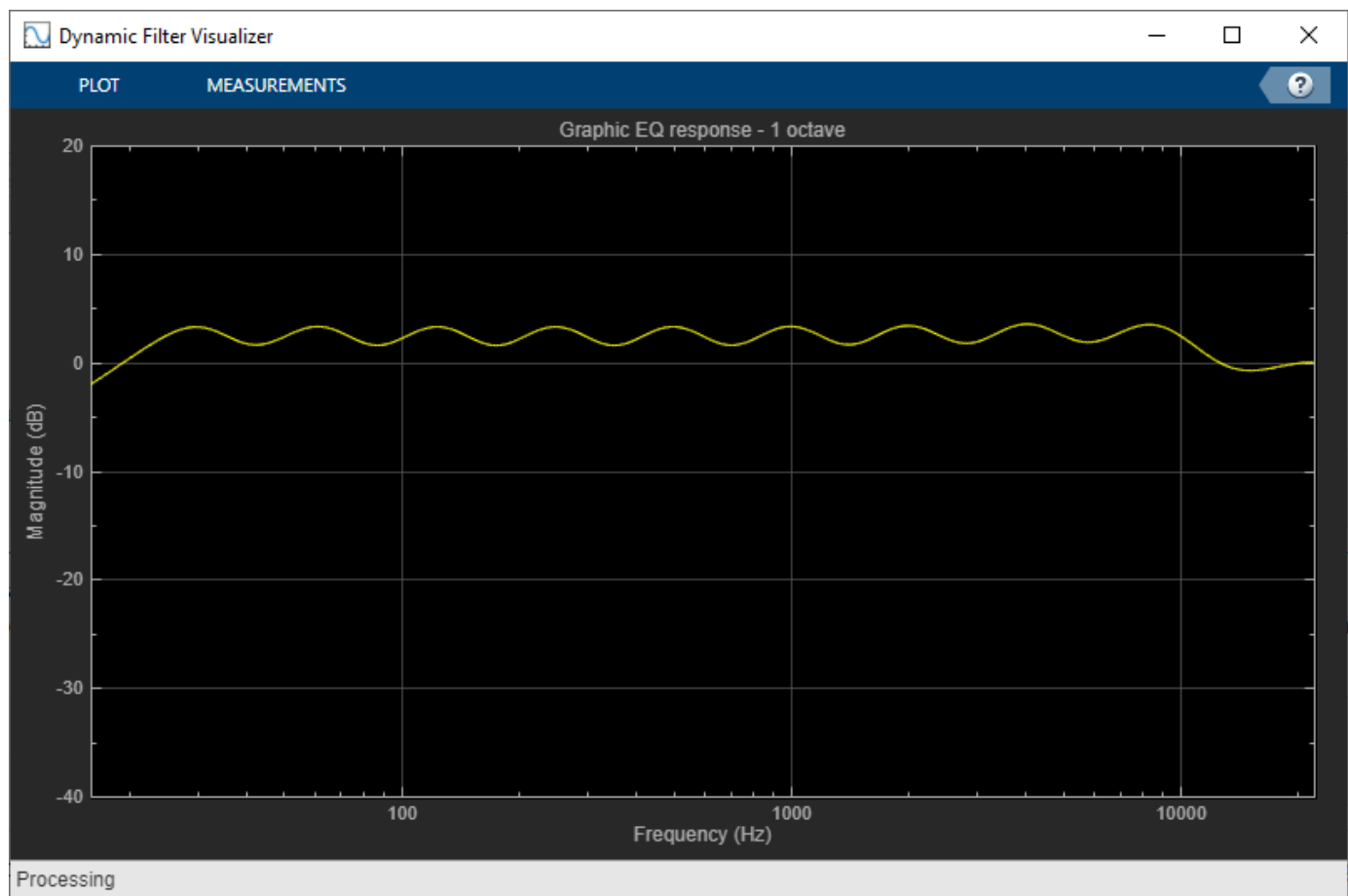
Graphic Equalizers with Cascade Filters

A different implementation of the graphic equalizer uses cascaded equalizing filters (peak or notch) implemented as biquad filters. The transfer function of the equalizer can be written as a product of the transfer function of individual biquads.

$$H_{eq}(z) = \prod_{m=1}^M H_m(z)$$

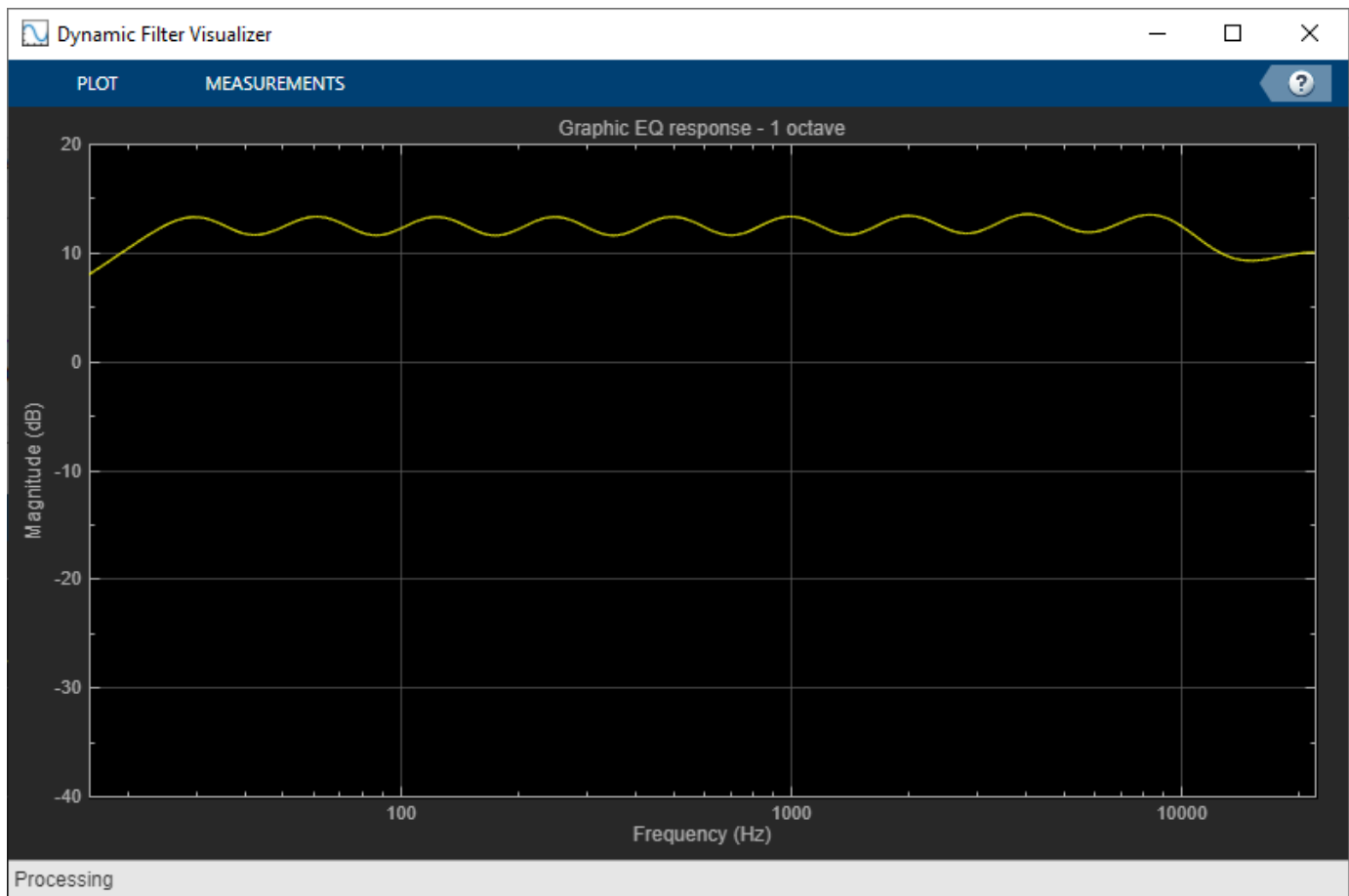
To motivate the usefulness of this implementation, first look at the magnitude response of the parallel structure when all gains are 0 dB.

```
parallelGraphicEQ = graphicEQ('Structure','Parallel');
visualize(parallelGraphicEQ)
```



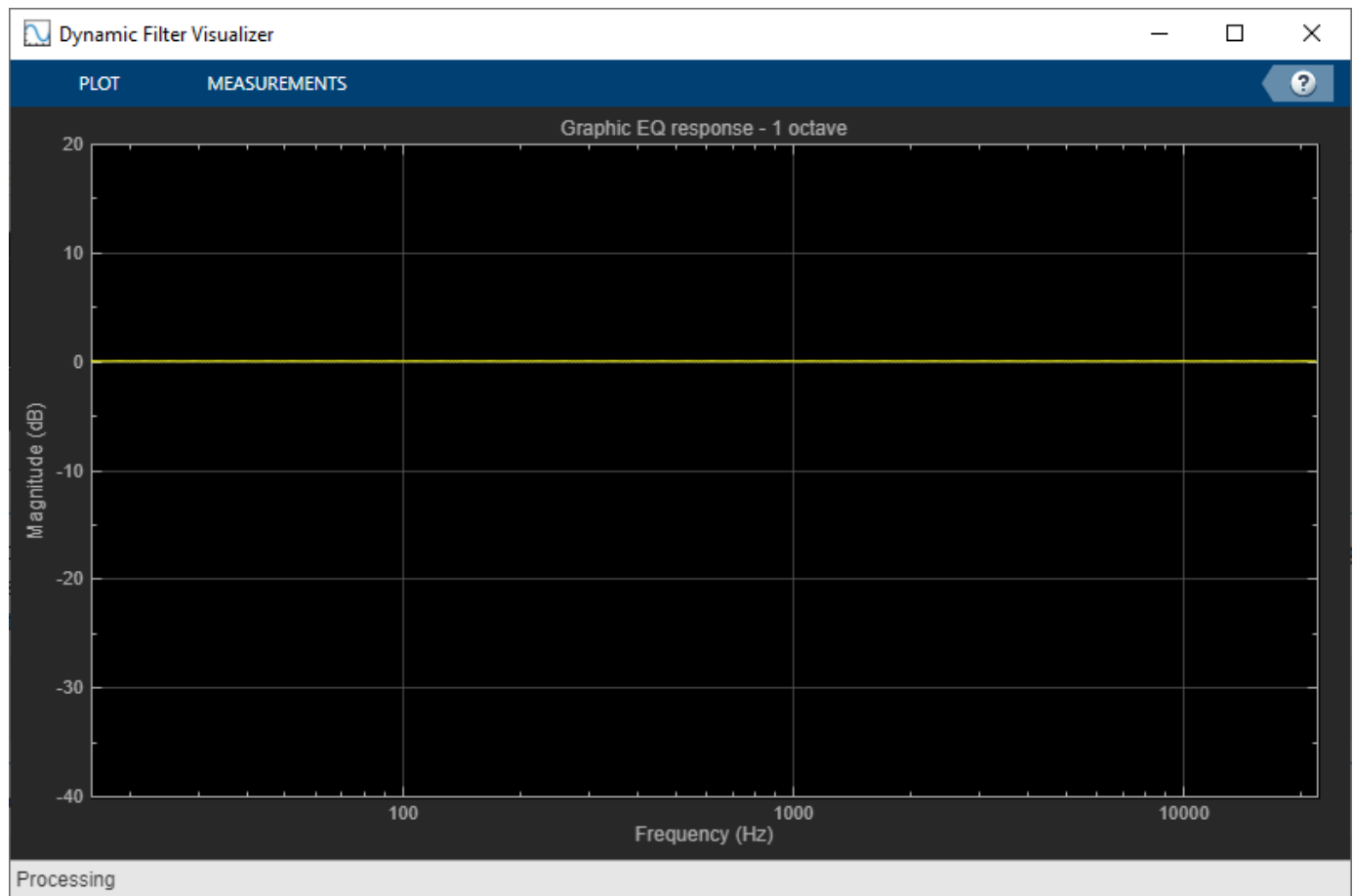
You will notice that the magnitude response is not flat. This is because the filters have been designed independently, and each has a transition width where the magnitude response droops. Moreover, because of non-ideal stopband, there is leakage from the stopband of one filter to the passband of its neighbor. The leakage can cause actual gains to differ from expected gains.

```
parallelGraphicEQ_10dB = graphicEQ('Structure','Parallel');  
parallelGraphicEQ_10dB.Gains = 10*ones(1,10);  
visualize(parallelGraphicEQ_10dB)
```



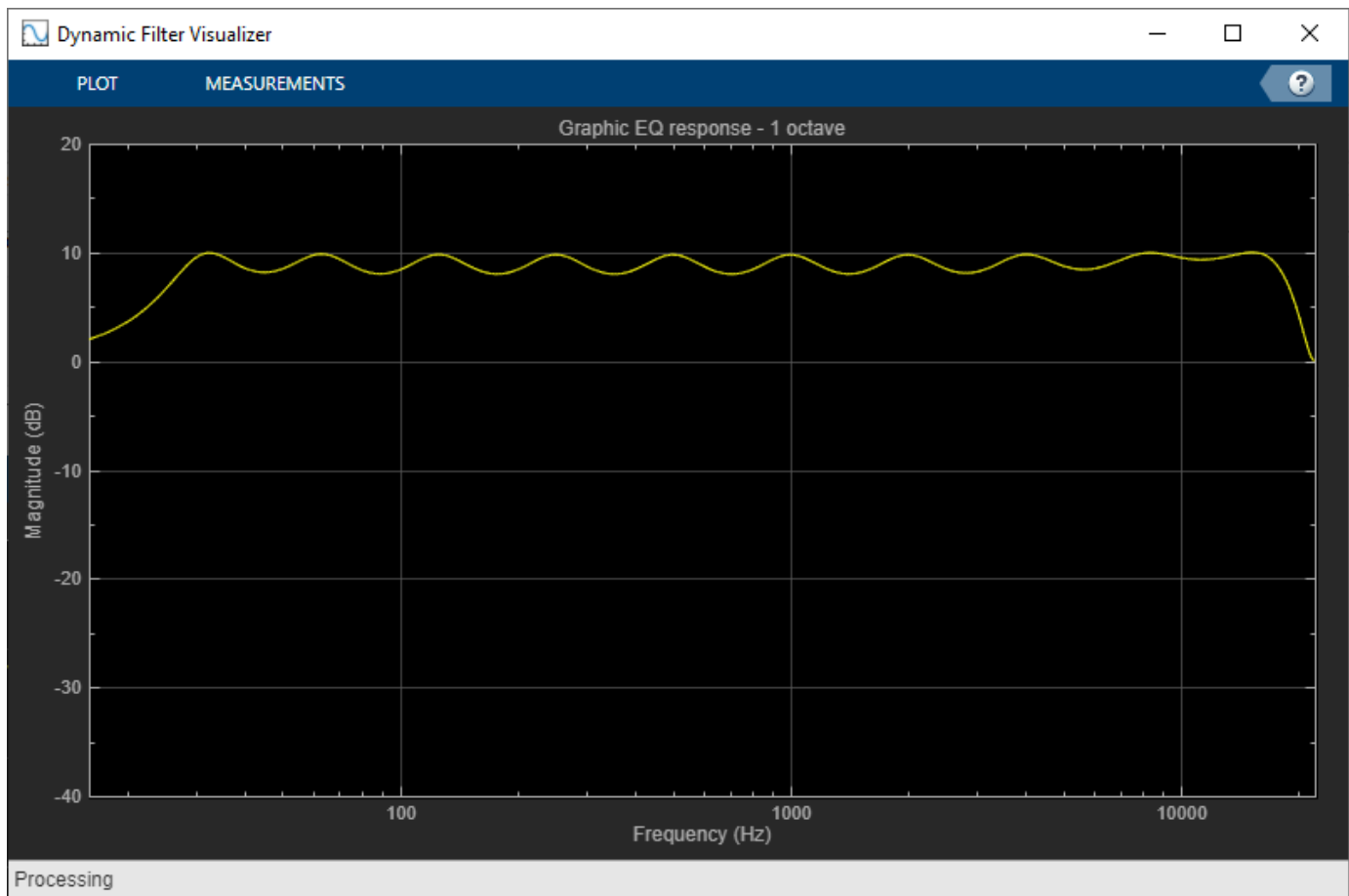
Note that the gains are never 10 dB in the frequency response. A cascaded structure can mitigate this to an extent because the gain is inherent in the design of the filter. Setting the gain of all cascaded biquads to 0 dB leads to them being bypassed. Since there are no branches in this type of structure, this means you have a no-gain path between the input and the output. `graphicEQ` implements the cascaded structure by default.

```
cascadeGraphicEQ = graphicEQ;
visualize(cascadeGraphicEQ)
```



Moreover, when you set the gains to 10 dB, notice that the resultant frequency response has close to 10 dB of gain at the center frequencies.

```
cascadeGraphicEQ_10dB = graphicEQ;  
cascadeGraphicEQ_10dB.Gains = 10*ones(1,10);  
visualize(cascadeGraphicEQ_10dB)
```

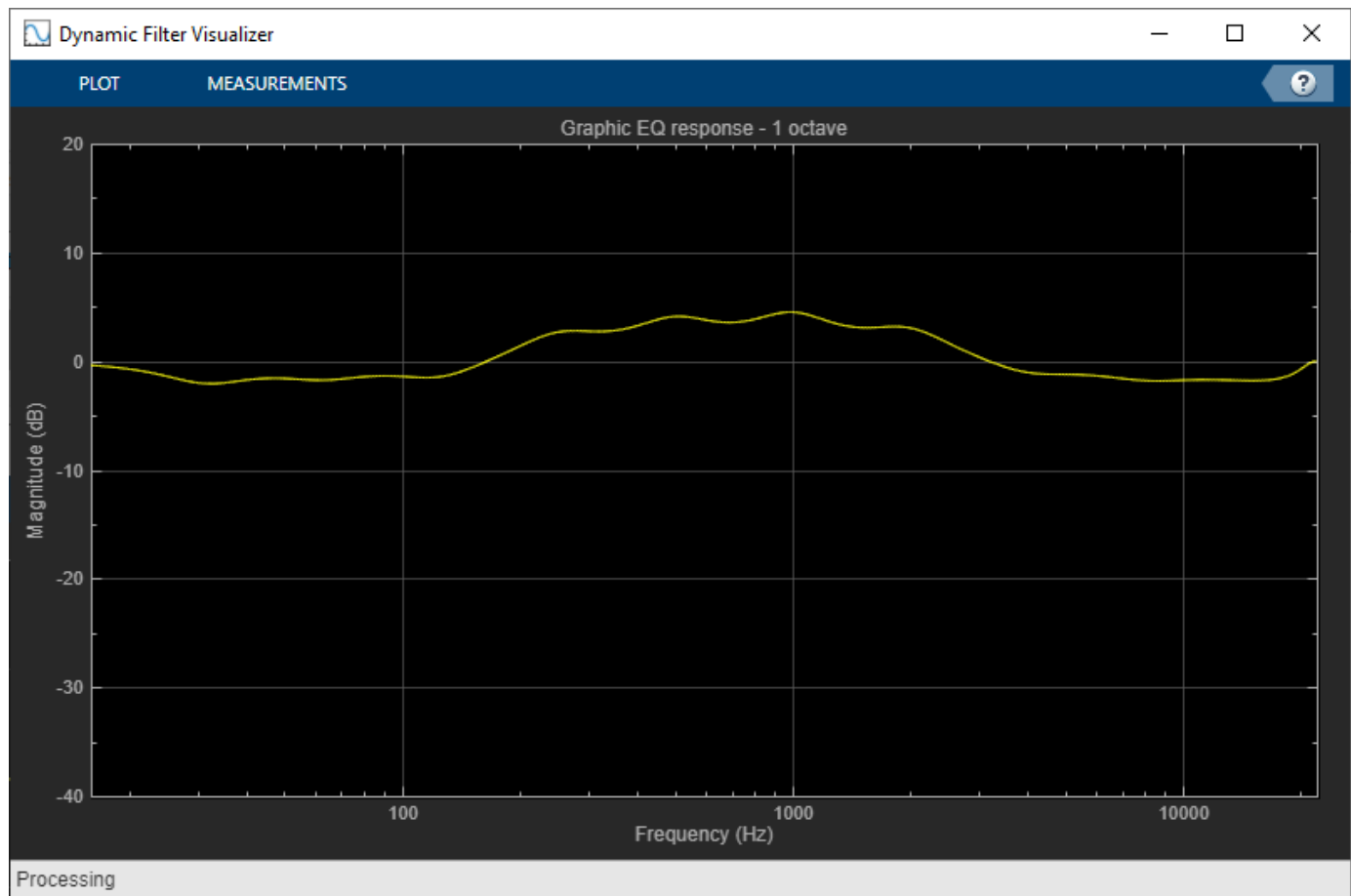


The drawback of cascade design is that the coefficients of a biquad stage need to be redesigned whenever the corresponding gain changes. This isn't needed for the parallel implementation because gain is just a multiplier to each parallel branch. A parallel connection of bandpass filters also avoids accumulating phase errors and quantization noise found in the cascade.

Fractional Octave Bandwidth

The `graphicEQ` object supports 1 octave, 2/3 octave, and 1/3 octave bandwidths. Reducing the bandwidth of individual filters allows you finer control over frequency response. To verify this, set the gains to boost mid frequencies, similar to a *pop* preset.

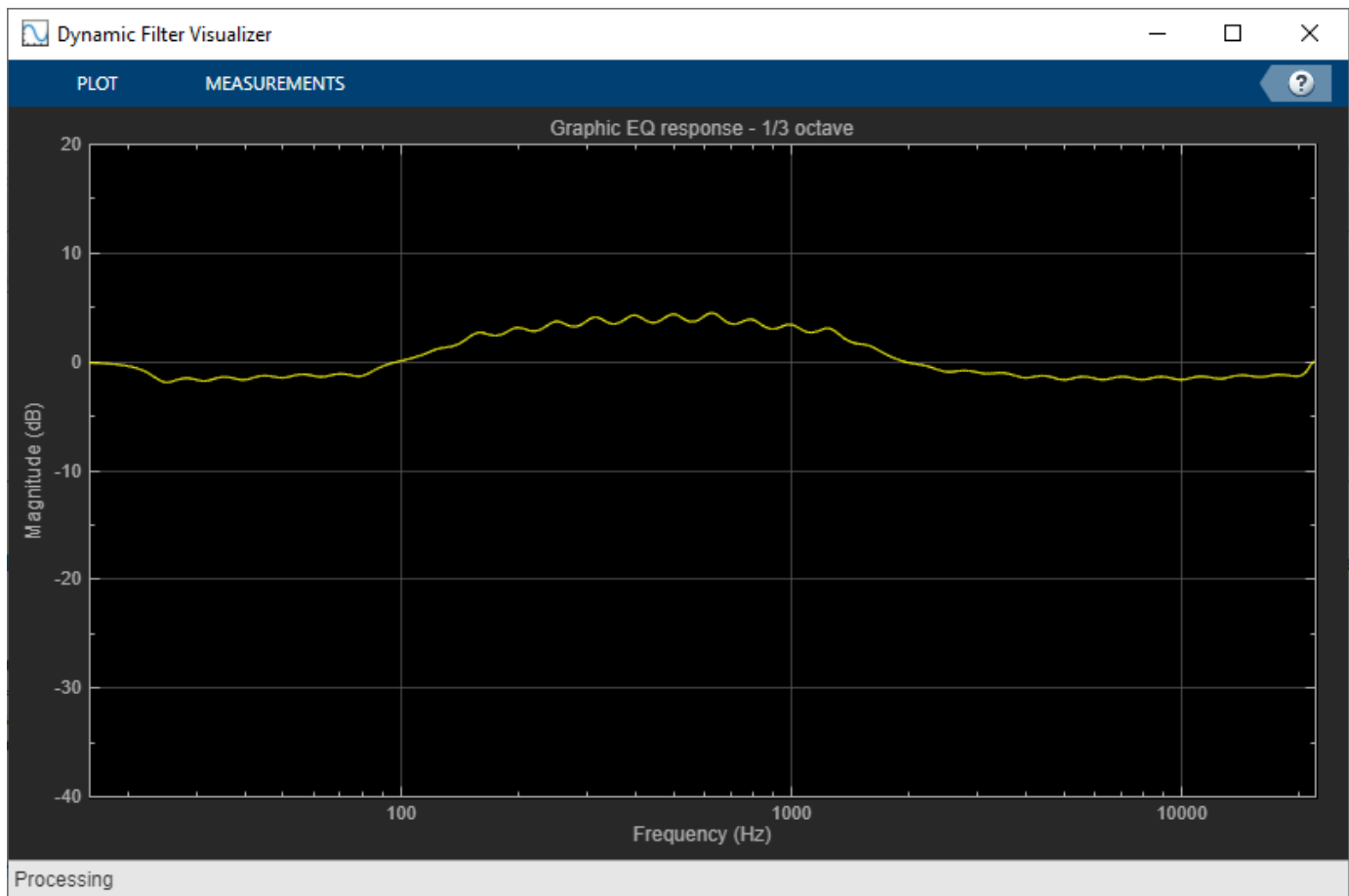
```
octaveGraphicEQ = graphicEQ;
octaveGraphicEQ.Gains = [-2.1,-1.8,-1.4,2.7,4.2,4.6,3.1,-1,-1.8,-1.8,-1.4];
visualize(octaveGraphicEQ)
```



```

oneThirdOctaveGraphicEQ = graphicEQ;
oneThirdOctaveGraphicEQ.Bandwidth = '1/3 octave';
oneThirdOctaveGraphicEQ.Gains = [-2,-1.9,-1.8,-1.6,-1.5,-1.4,0,1.2,2.7, ...
    3.2,3.8,4.2,4.4,4.5,4.6,4,3.5,3.1,1.5,-0.1,-1,-1.2,-1.6,-1.8,-1.8, ...
    -1.8,-1.8,-1.7,-1.5,-1.4,-1.3];
visualize(oneThirdOctaveGraphicEQ)

```



Generate Audio Plugin

To generate and port a VST plugin to a Digital Audio Workstation, run the `generateAudioPlugin` command. For example, you can generate a two-third octave graphic equalizer through the commands shown below. You will need to be in a directory with write permissions when you run these commands.

```
twoThirdOctaveGraphicEQ = graphicEQ;
twoThirdOctaveGraphicEQ.Bandwidth = '2/3 octave';
createAudioPluginClass(twoThirdOctaveGraphicEQ);
generateAudioPlugin twoThirdOctaveGraphicEQPlugin
```

Graphic Equalization in Simulink

You can use the same features described in this example in Simulink through the Graphic EQ block. It provides a slider for each gain value so you can easily boost or cut a frequency band while the simulation is running.

Audio Weighting Filters

This example shows how to obtain designs for the most common weighting filters - A-weighting, C-weighting, C-message, ITU-T 0.41, and ITU-R 468-4 - using the `weightingFilter` System object and audio weighting filter designer, `fdesign.audioweighting`, in the Audio Toolbox™.

In many applications involving acoustic measurements, the final sensor is the human ear. For this reason, acoustic measurements usually attempt to describe the subjective perception of a sound by this organ. Instrumentation devices are built to provide a linear response, but the ear is a nonlinear sensor. Special filters, known as weighting filters, are used to account for the nonlinearities.

A and C Weighting (ANSI® S1.42 Standard)

You can design A and C weighting filters that follow the ANSI S1.42 [1 on page 1-0] and IEC 61672-1 [2 on page 1-0] standards using `weightingFilter` System object. An A-weighting filter is a bandpass filter designed to simulate the perceived loudness of low-level tones. An A-weighting filter progressively de-emphasizes frequencies below 500 Hz. A C-weighting filter removes sounds outside the audio range of 20 Hz to 20 kHz and simulates the loudness perception of high-level tones. The following code designs an IIR filter for A-weighting with a sampling rate of 48 kHz.

```
AWeighting = weightingFilter('A-weighting',48000)
```

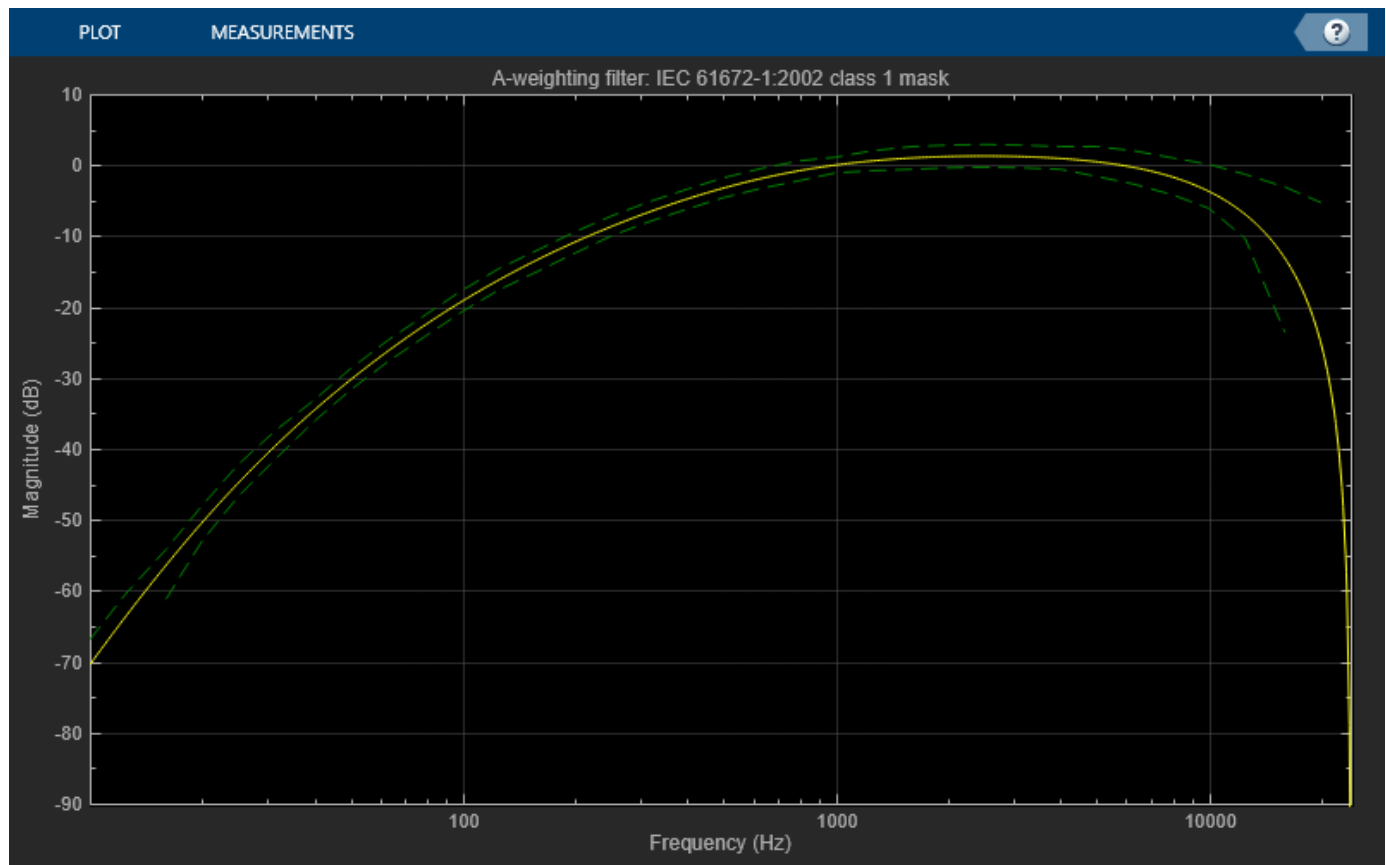
```
AWeighting =  
    weightingFilter with properties:
```

```
        Method: 'A-weighting'  
        SampleRate: 48000
```

A and C-weighting filter designs are based on direct implementation of the filter's transfer function based on poles and zeros specified in the ANSI S1.42 standard.

The IEC 61672-1 standard requires that the filter magnitudes fall within a specified tolerance mask. The standard defines two masks, one with stricter tolerance values than the other. A filter that meets the tolerance specifications of the stricter mask is referred to as a Class 1 filter. A filter that meets the specifications of the less strict mask is referred to as a Class 2 filter. You can view the magnitude response of the filter along with a mask corresponding to Class 1 or Class 2 specifications by calling the `visualize` method on the object. Note that the choice of the Class value will not affect the filter design itself but it will be used to render the correct tolerance mask in the visualization plot.

```
visualize(AWeighting,'class 1')
```

The A- and C-weighting standards specify tolerance magnitude values for up to 20 kHz. In the following example we use a sample rate of 28 kHz and design a C-weighting filter. Even though the Nyquist interval for this sample rate is below the maximum specified 20 kHz frequency, the design still meets the Class 2 tolerances as shown by the green mask around the magnitude response plot. The design, however, does not meet Class 1 tolerances due to the small sample rate value and you will see the mask around the magnitude response plot turn red.

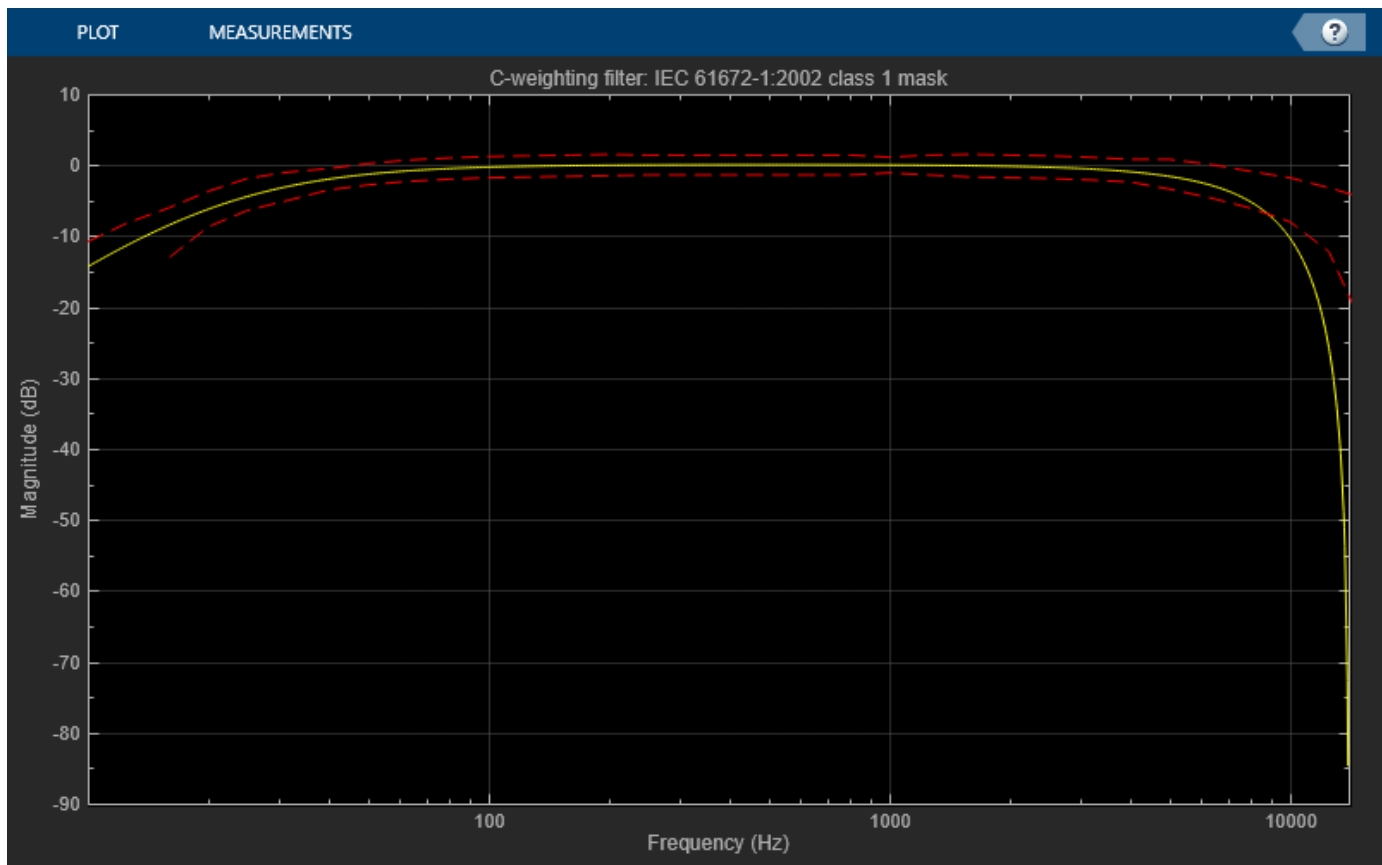
```
CWeighting = weightingFilter('C-weighting',28000)
```

```
CWeighting =  
  weightingFilter with properties:
```

```
    Method: 'C-weighting'  
    SampleRate: 28000
```

```
visualize(CWeighting,'class 2')
```

```
visualize(CWeighting,'class 1')
```



ITU-R 468-4 Weighting Filter

ITU-R 468-4 recommendation [3 on page 1-0] was developed to better reflect the subjective loudness of all types of noise, as opposed to tones. ITU-R 468-4 weighting was designed to maximize its response to the types of impulsive noise often coupled into audio cables as they pass through telephone switching facilities. ITU-R 468-4 weighting correlates well with noise perception, since perception studies have shown that frequencies between 1 kHz and 9 kHz are more "annoying" than indicated by A-weighting.

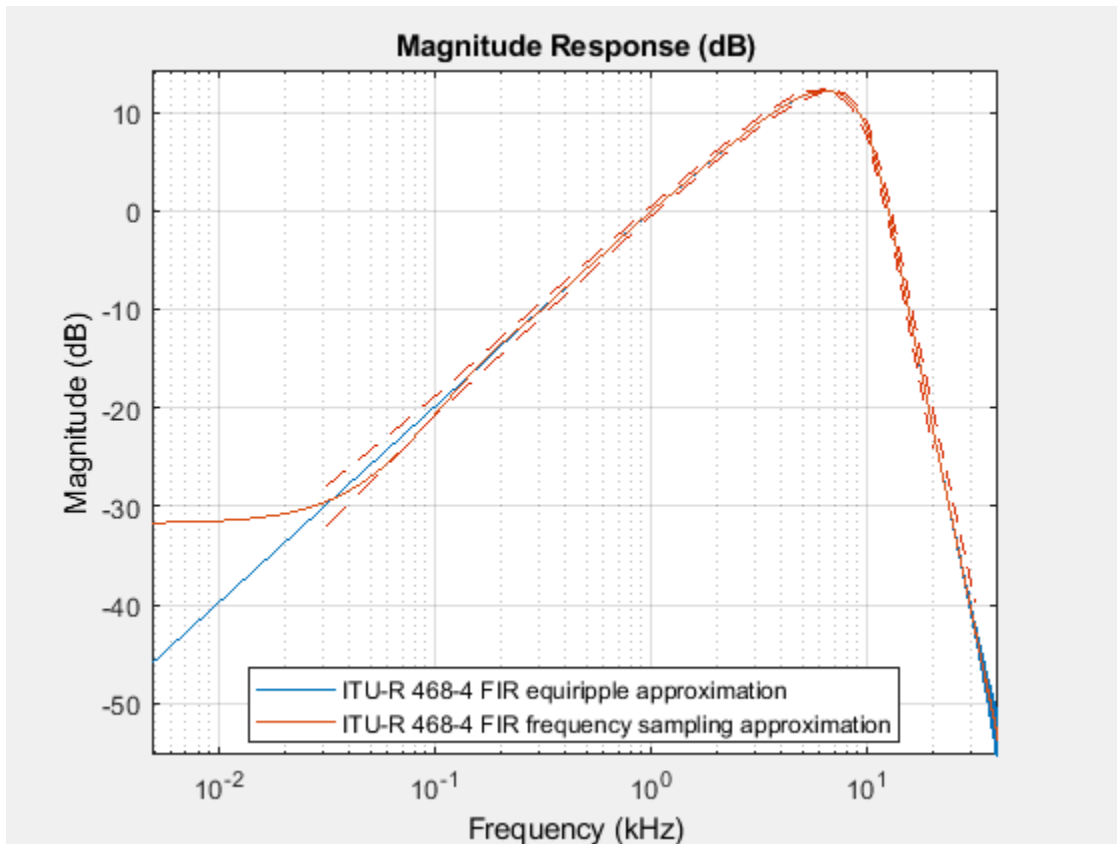
You design a weighting filter based on the ITU-R 468-4 standard using `fdesign.audioweighting` specification object. You can choose between frequency sampling or equiripple approximations for an FIR filter design, or use a least P-norm approximation for an IIR filter design. In all cases, the filters are designed with the minimum order that meets the standard specifications (mask) for the sample rate at hand.

```
ITUR4684Designer = fdesign.audioweighting('WT','ITUR4684',80e3)
```

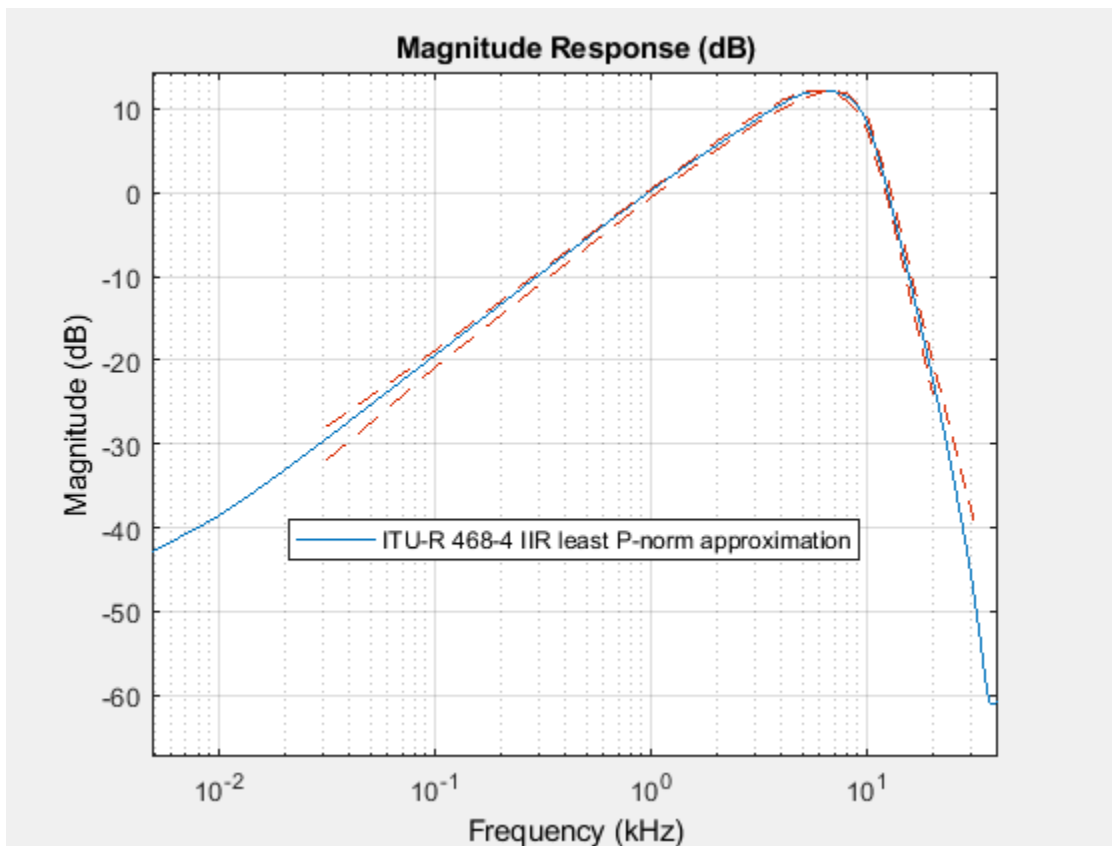
```
ITUR4684Designer =  
audioweighting with properties:  
    Response: 'Audio Weighting'  
    Specification: 'WT'  
    Description: {'Weighting type'}  
    NormalizedFrequency: 0  
    Fs: 80000
```

WeightingType: 'ITUR4684'

```
ITUR4684FIR = design(ITUR4684Designer,'allfir','SystemObject',true);
visualizer = fvtool(ITUR4684FIR{1});
addfilter(visualizer,ITUR4684FIR{2});
legend(visualizer,'ITU-R 468-4 FIR equiripple approximation', ...
    'ITU-R 468-4 FIR frequency sampling approximation')
```



```
ITUR4684IIR = design(ITUR4684Designer,'iirlpnorm','SystemObject',true);
setfilter(visualizer,ITUR4684IIR);
legend(visualizer,'ITU-R 468-4 IIR least P-norm approximation')
```



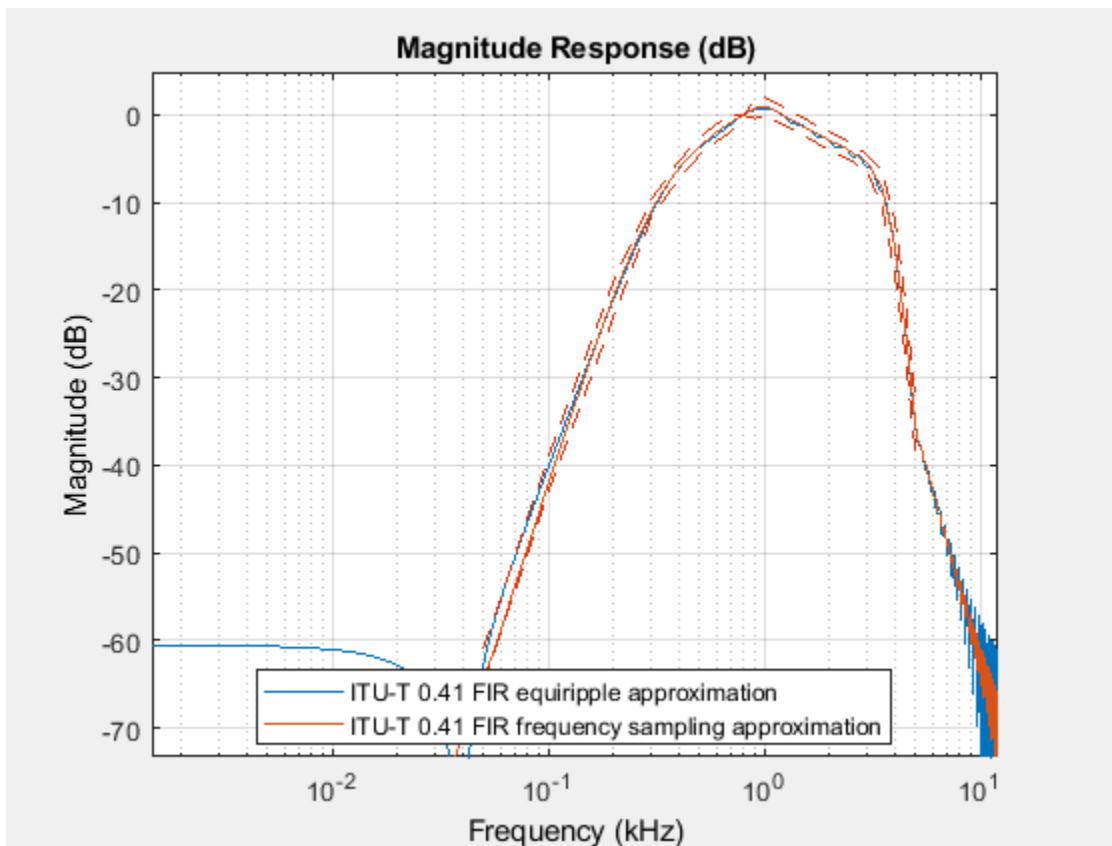
While IIR designs yield smaller filter orders, FIR designs have the advantage of a linear phase response. In the FIR designs, the equiripple design method will usually yield lower filter orders when compared to the frequency sampling method but might have some design-time convergence issues at large sample rates.

ITU-T 0.41 and C-message Weighting Filters

ITU-T 0.41 and C-message weighting filters are bandpass filters used to measure audio-frequency noise on telephone circuits. The ITU-T 0.41 filter is used for international telephone circuits. The C-message filter is typically used for North American telephone circuits. The frequency response of the ITU-T 0.41 and C-message weighting filters is specified in the ITU-T 0.41 standard [4 on page 1-0] and Bell System Technical Reference 41009 [5 on page 1-0], respectively.

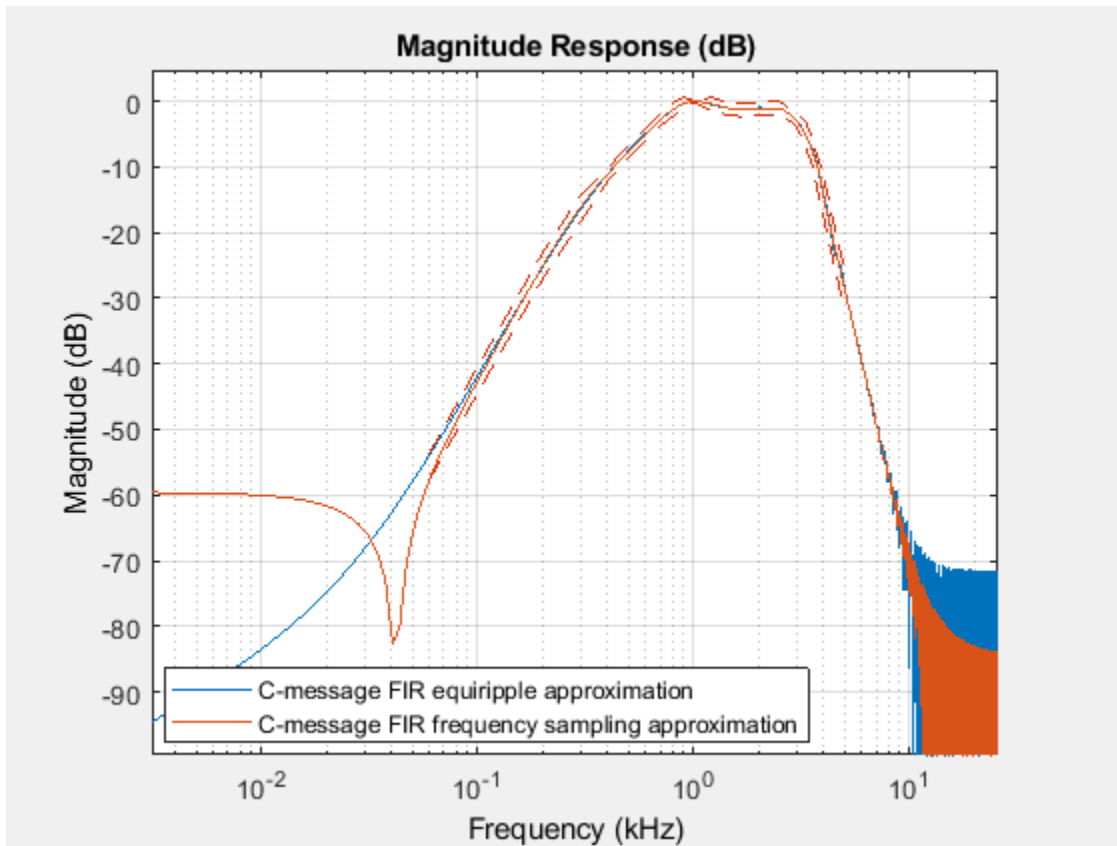
You design an ITU-T 0.41 weighting filter for a sample rate of 24 kHz using the following code. You can choose from FIR frequency sampling or equiripple approximations. The filters are designed with the minimum order that meets the standard specifications (mask) for the sampling frequency at hand.

```
ITUTDesigner = fdesign.audioweighting('WT','ITUT041',24e3);
ITUT = design(ITUTDesigner,'allfir','SystemObject',true);
setfilter(visualizer,ITUT{1});
addfilter(visualizer,ITUT{2});
legend(visualizer,'ITU-T 0.41 FIR equiripple approximation', ...
    'ITU-T 0.41 FIR frequency sampling approximation')
```

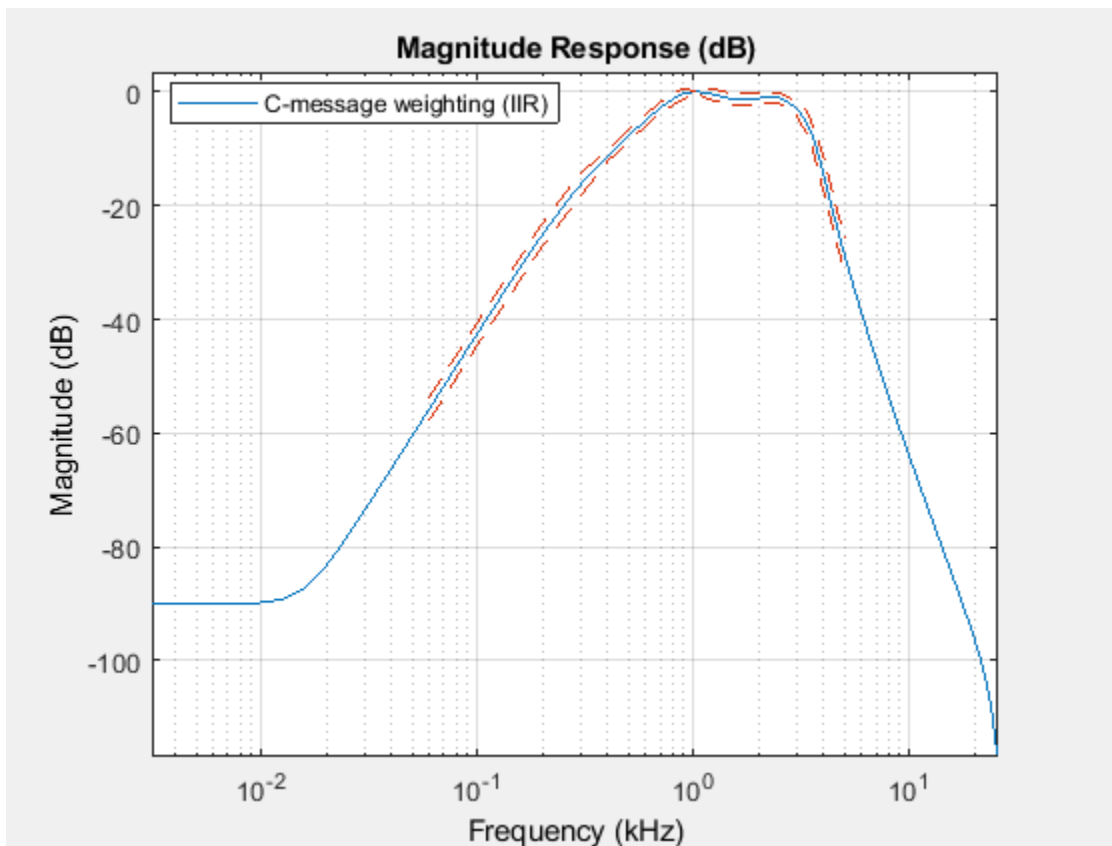


You design a C-message weighting filter for a sample rate of 51.2 kHz using the following code. You can choose from FIR frequency sampling or equiripple approximations or from an exact IIR implementation of poles and zeros based on the poles and zeros specified in [6 on page 1-0]. You obtain the IIR design by selecting the 'bell41009' design method. The FIR filter approximations are designed with the minimum order that meets the standard specifications (mask) for the sample rate at hand.

```
CMessageDesigner = fdesign.audioweighting('WT','Cmessage',51.2e3);
CMessageFIR = design(CMessageDesigner,'allfir','SystemObject',true);
setfilter(visualizer,CMessageFIR{1});
addfilter(visualizer,CMessageFIR{2});
legend(visualizer,'C-message FIR equiripple approximation', ...
    'C-message FIR frequency sampling approximation')
```



```
CMessageIIR = design(CMessageDesigner,'bell41009','SystemObject',true);
setfilter(visualizer,CMessageIIR);
legend(visualizer,'C-message weighting (IIR)')
```



Conclusions

Some audio weighting standards do not specify exact pole/zero values, instead, they specify a list of frequency values, magnitudes and tolerances. If the exact poles and zeros are not specified in the standard, filters are designed using frequency sampling, equiripple, or IIR least P-norm arbitrary magnitude approximations based on the aforementioned list of frequency values, attenuations, and tolerances. The filter order of the arbitrary magnitude designs is chosen as the minimum order for which the resulting filter response is within the tolerance mask limits. Designs target the specification mask tolerances only within the Nyquist interval. If $F_s/2$ is smaller than the largest mask frequency value specified by the standard, the design algorithm will try to meet the specifications up to $F_s/2$.

In the FIR designs, the equiripple design method usually yields lower filter orders when compared to the frequency sampling method but might have some convergence issues at large sample rates.

References

- [1] 'Design Response of Weighting Networks for Acoustical Measurements', American National Standard, ANSI S1.42-2001.
- [2] 'Electroacoustics Sound Level Meters Part 1: Specifications', IEC 61672-1, First Edition 2002-05.
- [3] 'Measurement of Audio-Frequency Noise Voltage Level in Sound Broadcasting', Recommendation ITU-R BS.468-4 (1970-1974-1978-1982-1986).
- [4] 'Specifications for Measuring Equipment for the Measurement of Analogue Parameters, Psophometer for Use on Telephone-Type Circuits', ITU-T Recommendation 0.41.

[5] 'Transmission Parameters Affecting Voiceband Data Transmission-Measuring Techniques', Bell System Technical Reference, PUB 41009, 1972.

[6] 'IEEE® Standard Equipment Requirements and Measurement Techniques for Analog Transmission Parameters for Telecommunications', IEEE Std 743-1995 Volume , Issue , 25, September 1996.

Sound Pressure Measurement of Octave Frequency Bands

This example demonstrates how to measure sound pressure levels of octave frequency bands. A user interface (UI) enables you to experiment with various parameters while the measurement is displayed.

Sound Pressure Measurement

Many applications involving acoustic measurements must take into account the non-linear characteristics of the human auditory system. For that reason, sound levels are generally reported in decibels (dB) and on a frequency scale that increases logarithmically. Frequency weighting adjusts levels to take into account the ear's frequency-dependent sensitivity. A-weighting is the most common, as it cuts low and high frequencies similarly to the auditory system for "normal" levels. C-weighting is an alternative for measuring very loud sounds, as it mimics the human ear's flatter response at level over 100 dB.

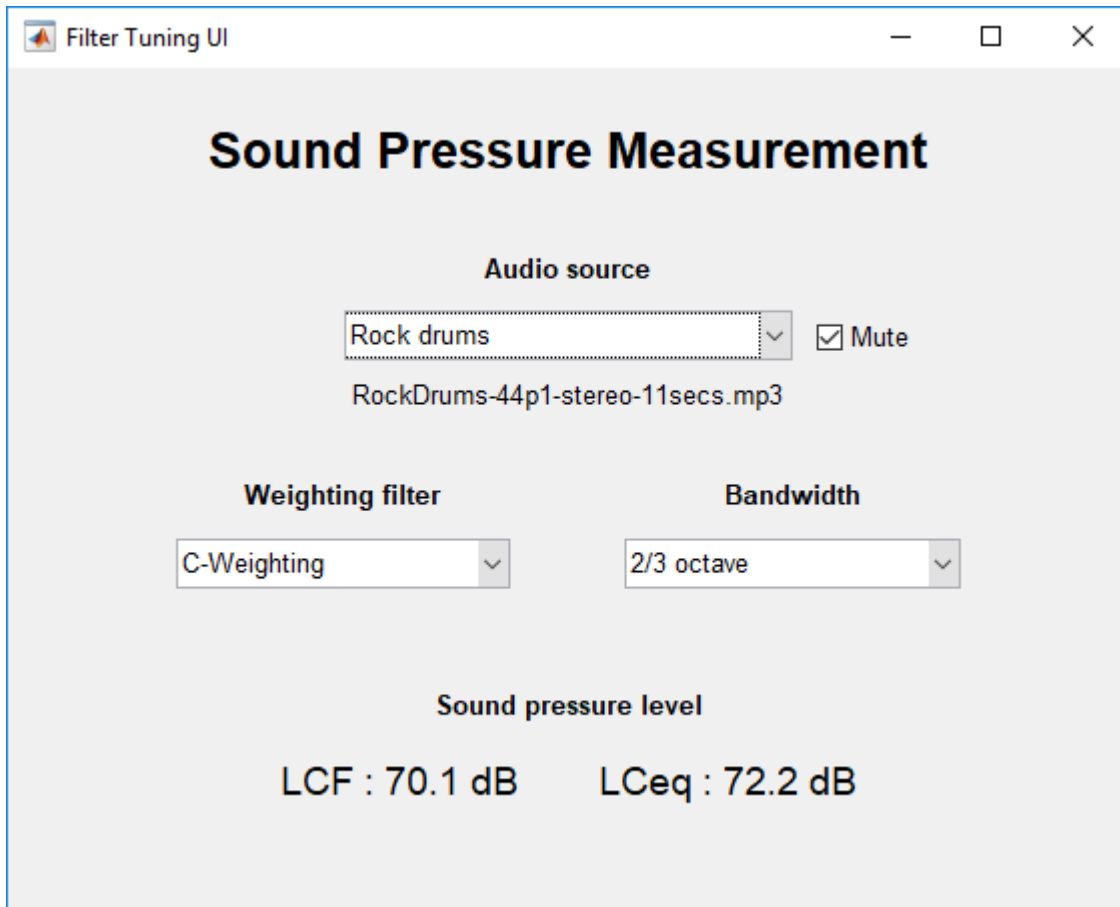
This example uses the `splMeter` System object to measure sound pressure levels (SPL). You can measure sound pressure levels of audio files or perform live SPL measurements with a microphone.

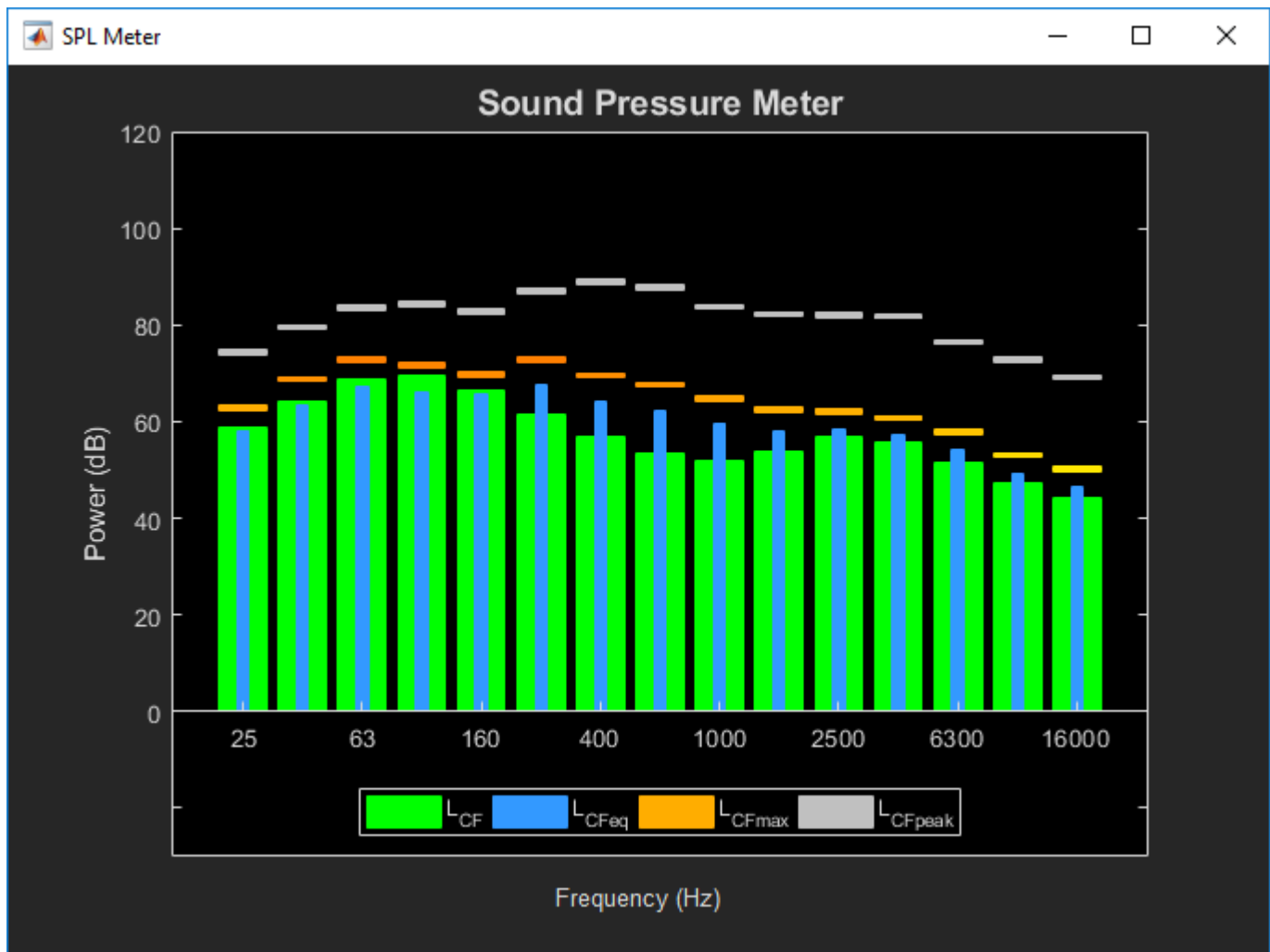
You can specify the weighting filter (Z/A/C) and frequency bandwidth used for the measurements. For more information on the weighting filters, see the "Audio Weighting Filters" on page 1-178 example.

MATLAB Simulation

`soundPressureMeasurementExampleApp` loads the SPL meter user interface (shown below). The demonstration begins with pink noise, which measures relatively flat on the octave frequency scale. You can experiment with different audio sources, frequency weightings, and bandwidths.

Execute `soundPressureMeasurementExampleApp` to run the demonstration and display the measurements.





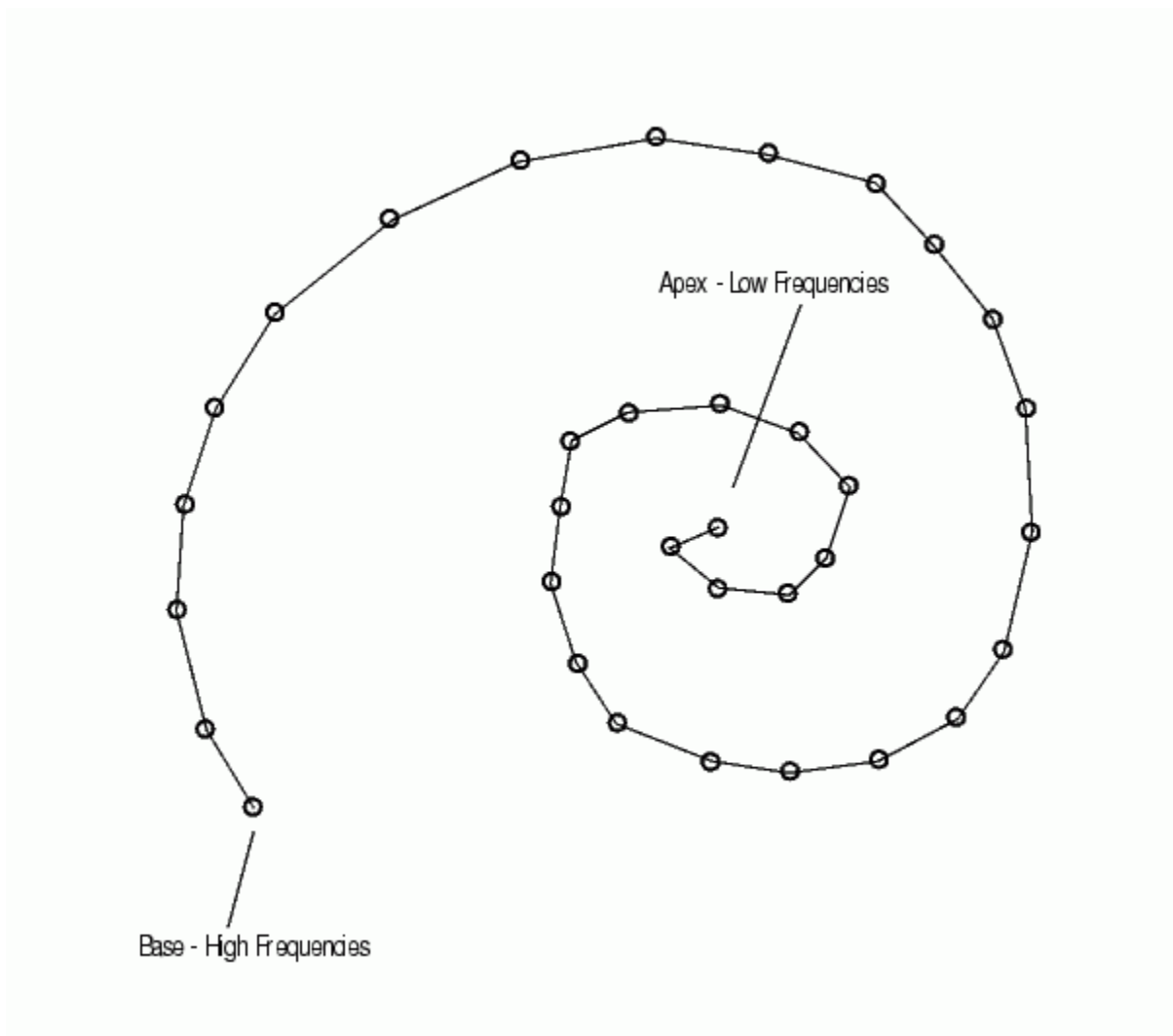
Cochlear Implant Speech Processor

This example shows how to simulate the design of a cochlear implant that can be placed in the inner ear of a profoundly deaf person to restore partial hearing. Signal processing is used in cochlear implants to convert sound to electrical pulses. The pulses can bypass the damaged parts of a deaf person's ear and be transmitted to the brain to provide partial hearing.

This example highlights some of the choices made when designing cochlear implant speech processors using Audio Toolbox™. In particular, the benefits of using a cascaded multirate, multistage FIR filter bank instead of a parallel, single-rate, second-order-section IIR filter bank are shown.

Human Hearing

Converting sound into something the human brain can understand involves the inner, middle, and outer ear, hair cells, neurons, and the central nervous system. When a sound is made, the outer ear picks up acoustic waves, which are converted into mechanical vibrations by tiny bones in the middle ear. The vibrations move to the inner ear, where they travel through fluid in a snail-shaped structure called the cochlea. The fluid displaces different points along the basilar membrane of the cochlea. Displacements along the basilar membrane contain the frequency information of the acoustic signal. A schematic of the membrane is shown here (not drawn to scale).



Frequency Sensitivity in the Cochlea

Different frequencies cause the membrane to displace maximally at different positions. Low frequencies cause the membrane to be displaced near its apex, while high frequencies stimulate the membrane at its base. The amplitude of the displacement of the membrane at a particular point is proportional to the amplitude of the frequency that has excited it. When a sound is composed of many frequencies, the basilar membrane is displaced at multiple points. In this way the cochlea separates complex sounds into frequency components.

Each region of the basilar membrane is attached to hair cells that bend proportionally to the displacement of the membrane. The bending causes an electrochemical reaction that stimulates neurons to communicate the sound information to the brain through the central nervous system.

Alleviating Deafness with Cochlear Implants

Deafness is most often caused by degeneration or loss of hair cells in the inner ear, rather than a problem with the associated neurons. This means that if the neurons can be stimulated by a means other than hair cells, some hearing can be restored. A cochlear implant does just that. The implant electrically stimulates neurons directly to provide information about sound to the brain.

The problem of how to convert acoustic waves to electrical impulses is one that Signal Processing helps to solve. Multichannel cochlear implants have the following components in common:

- A microphone to pick up sound
- A signal processor to convert acoustic waves to electrical signals
- A transmitter
- A bank of electrodes that receive the electrical signals from the transmitter, and then stimulate auditory nerves

Just as the basilar membrane of the cochlea resolves a wave into its component frequencies, so does the signal processor in a cochlear implant divide an acoustic signal into component frequencies, that are each then transmitted to an electrode. The electrodes are surgically implanted into the cochlea of the deaf person so that they each stimulate the appropriate regions in the cochlea for the frequency they are transmitting. Electrodes transmitting high-frequency (high-pitched) signals are placed near the base, while those transmitting low-frequency (low-pitched) signals are placed near the apex. Nerve fibers in the vicinity of the electrodes are stimulated and relay the information to the brain. Loud sounds produce high-amplitude electrical pulses that excite a greater number of nerve fibers, while quiet ones excite less. In this way, information about both the frequencies and amplitudes of the components making up a sound can be transmitted to the brain of a deaf person.

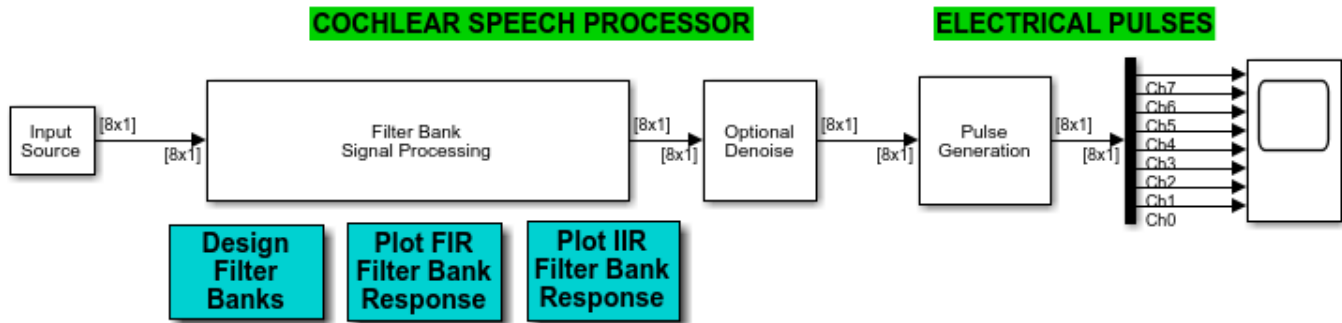
Exploring the Example

The block diagram at the top of the model represents a cochlear implant speech processor, from the microphone which picks up the sound (Input Source block) to the electrical pulses that are generated. The frequencies increase in pitch from Channel 0, which transmits the lowest frequency, to Channel 7, which transmits the highest.

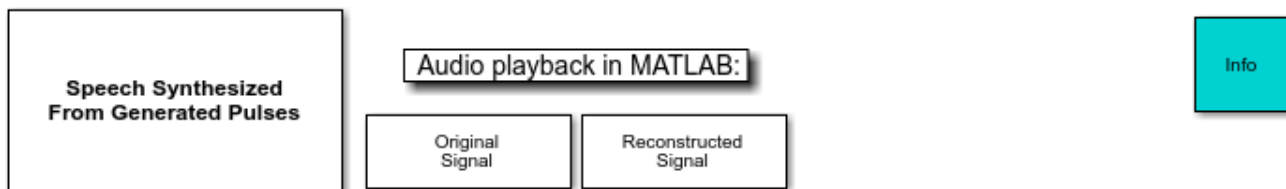
To hear the original input signal, double-click the Original Signal block at the bottom of the model. To hear the output signal of the simulated cochlear implant, double-click the Reconstructed Signal block.

There are a number of changes you can make to the model to see how different variables affect the output of the cochlear implant speech processor. Remember that after you make a change, you must rerun the model to implement the changes before you listen to the reconstructed signal again.

Cochlear Implant Speech Processor



SPEECH RECONSTRUCTION FOR NORMAL HEARING PEOPLE



Copyright 2005-2015 The MathWorks, Inc.

Simultaneous Versus Interleaved Playback

Research has shown that about eight frequency channels are necessary for an implant to provide good auditory understanding for a cochlear implant user. Above eight channels, the reconstructed signal usually does not improve sufficiently to justify the rising complexity. Therefore, this example resolves the input signal into eight component frequencies, or electrical pulses.

The Speech Synthesized from Generated Pulses block at the bottom left of the model allows you to either play each electrical channel simultaneously or sequentially. Oftentimes cochlear implant users experience inferior results with simultaneous frequencies, because the electrical pulses interact with each other and cause interference. Emitting the pulses in an interleaved manner mitigates this problem for many people. You can toggle the **Synthesis mode** of the Speech Synthesized From Generated Pulses block to hear the difference between these two modes. Zoom in on the Time Scope block to observe that the pulses are interleaved.

Adjusting for Noisy Environments

Noise presents a significant challenge to cochlear implant users. Select the **Add noise** parameter in the Input Source block to simulate the effects of a noisy environment on the reconstructed signal. Observe that the signal becomes difficult to hear. The Denoise block in the model uses a Soft Threshold block to attempt to remove noise from the signal. When the **Denoise** parameter in the Denoise block is selected, you can listen to the reconstructed signal and observe that not all the noise is removed. There is no perfect solution to the noise problem, and the results afforded by any denoising technology must be weighed against its cost.

Signal Processing Strategy

The purpose of the Filter Bank Signal Processing block is to decompose the input speech signal into eight overlapping subbands. More information is contained in the lower frequencies of speech signals than in the higher frequencies. To get as much resolution as possible where the most information is contained, the subbands are spaced such that the lower-frequency bands are more narrow than the higher-frequency bands. In this example, the four low-frequency bands are equally spaced, while each of the four remaining high-frequency bands is twice the bandwidth of its lower-frequency neighbor. To examine the frequency contents of the eight filter banks, run the model using the **Chirp Source type** in the Input Source block.

Two filter bank implementations are illustrated in this example: a parallel, single-rate, second-order-section IIR filter bank and a cascaded, multirate, multistage FIR filter bank. Double click on the **Design Filter Banks** button to examine their design and frequency specifications.

Parallel Single-Rate SOS IIR Filter Bank: In this bank, the sixth-order IIR filters are implemented as second-order-sections (SOS). The eight filters are running in parallel at the input signal rate. You can look at their frequency responses by double clicking the **Plot IIR Filter Bank Response** button.

Cascaded Multirate Multistage FIR Filter Bank: The design of this filter bank is based on the principles of an approach that combines downsampling and filtering at each filter stage. The overall filter response for each subband is obtained by cascading its components. Double click on the **Design Filter Banks** button to examine how design functions from the Audio Toolbox are used in constructing these filter banks.

Since downsampling is applied at each filter stage, the later stages are running at a fraction of the input signal rate. For example, the last filter stages are running at one-eighth of the input signal rate. Consequently, this design is very suitable for implementations on the low-power DSPs with limited processing cycles that are used in cochlear implant speech processors. You can look at the frequency responses for this filter bank by double clicking on the **Plot FIR Filter Bank Response** button. Notice that this design produces sharper and flatter subband definition compared to the parallel single-rate SOS IIR filter bank. This is another benefit of a multirate, multistage filter design approach. For a related example see "Multistage Design Of Decimators/Interpolators" in the DSP System Toolbox™ FIR Filter Design examples.

Acknowledgements and References

Thanks to Professor Philip Loizou for his help in creating this example.

More information on Professor Loizou's cochlear implant research is available at:

- Loizou, Philip C., "Mimicking the Human Ear," **IEEE® Signal Processing Magazine**, Vol. 15, No. 5, pp. 101-130, 1998.

Acoustic Beamforming Using a Microphone Array

This example illustrates microphone array beamforming to extract desired speech signals in an interference-dominant, noisy environment. Such operations are useful to enhance speech signal quality for perception or further processing. For example, the noisy environment can be a trading room, and the microphone array can be mounted on the monitor of a trading computer. If the trading computer must accept speech commands from a trader, the beamformer operation is crucial to enhance the received speech quality and achieve the designed speech recognition accuracy.

The example shows two types of time domain beamformers: the time delay beamformer and the Frost beamformer. It illustrates how you can use diagonal loading to improve the robustness of the Frost beamformer. You can listen to the speech signals at each processing step.

This example requires Phased Array System Toolbox.

Define a Uniform Linear Array

First, define a uniform linear array (ULA) to receive the signal. The array contains 10 omnidirectional elements (microphones) spaced 5 cm apart. Set the upper bound for frequency range of interest to 4 kHz because the signals used in this example are sampled at 8 kHz.

```
microphone = ...
    phased.OmnidirectionalMicrophoneElement('FrequencyRange',[20 4000]);

Nele = 10;
ula = phased.ULA(Nele,0.05,'Element',microphone);
c = 340; % speed of sound, in m/s
```

Simulate the Received Signals

Next, simulate the multichannel signal received by the microphone array. Two speech signals are used as audio of interest. A laughter audio segment is used as interference. The sampling frequency of the audio signals is 8 kHz.

Because audio signals are usually large, it is often not practical to read the entire signal into the memory. Therefore, in this example, you read and process the signal in a streaming fashion, i.e., break the signal into small blocks at the input, process each block, and then assemble them at the output.

The incident direction of the first speech signal is -30 degrees in azimuth and 0 degrees in elevation. The direction of the second speech signal is -10 degrees in azimuth and 10 degrees in elevation. The interference comes from 20 degrees in azimuth and 0 degrees in elevation.

```
ang_dft = [-30; 0];
ang_cleanspeech = [-10; 10];
ang_laughter = [20; 0];
```

Now you can use a wideband collector to simulate a 3-second signal received by the array. Notice that this approach assumes that each input single-channel signal is received at the origin of the array by a single microphone.

```
fs = 8000;
collector = phased.WidebandCollector('Sensor',ula,'PropagationSpeed',c, ...
    'SampleRate',fs,'NumSubbands',1000,'ModulatedInput',false);
```

```
t_duration = 3; % 3 seconds
t = 0:1/fs:t_duration-1/fs;
```

Generate a white noise signal with a power of $1e-4$ Watts to represent the thermal noise for each sensor. A local random number stream ensures reproducible results.

```
prevS = rng(2008);
noisePwr = 1e-4;
```

Run the simulation. At the output, the received signal is stored in a 10-column matrix. Each column of the matrix represents the signal collected by one microphone. Note that the audio is played back during the simulation.

```
% preallocate
NSampPerFrame = 1000;
NTSample = t_duration*fs;
sigArray = zeros(NTSample,Nele);
voice_dft = zeros(NTSample,1);
voice_cleanspeech = zeros(NTSample,1);
voice_laugh = zeros(NTSample,1);

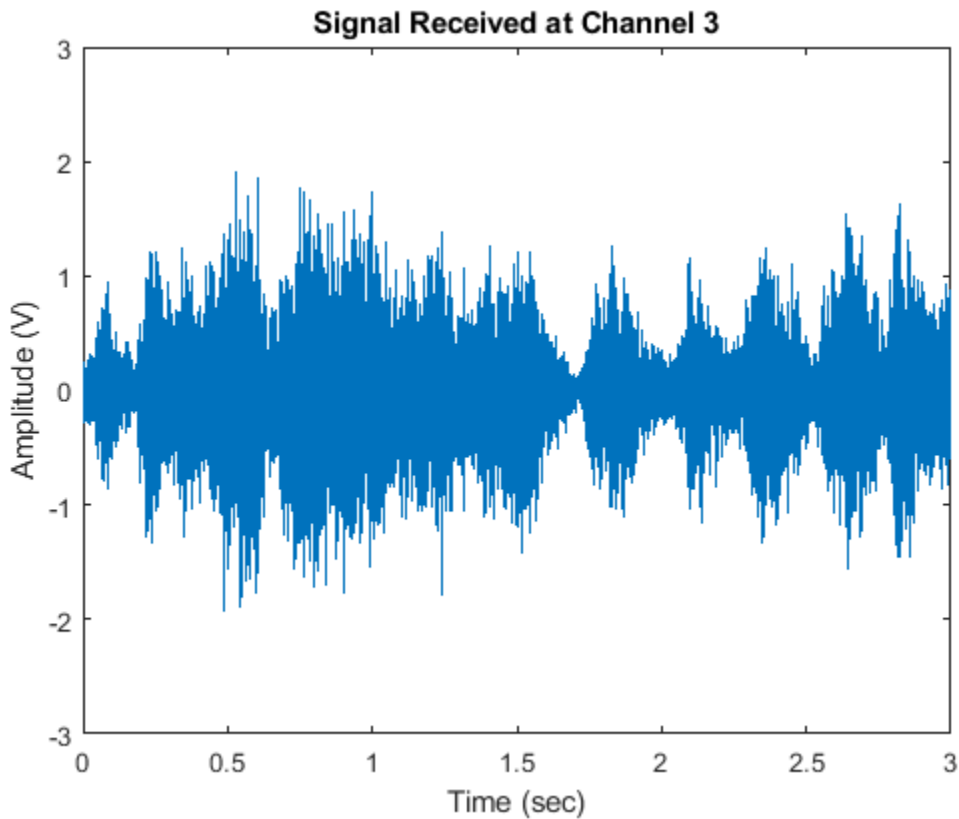
% set up audio device writer
player = audioDeviceWriter('SampleRate',fs);

dftFileReader = dsp.AudioFileReader('SpeechDFT-16-8-mono-5secs.wav', ...
    'SamplesPerFrame',NSampPerFrame);
speechFileReader = dsp.AudioFileReader('FemaleSpeech-16-8-mono-3secs.wav', ...
    'SamplesPerFrame',NSampPerFrame);
laughterFileReader = dsp.AudioFileReader('Laughter-16-8-mono-4secs.wav', ...
    'SamplesPerFrame',NSampPerFrame);

% simulate
for m = 1:NSampPerFrame:NTSample
    sig_idx = m:m+NSampPerFrame-1;
    x1 = dftFileReader();
    x2 = speechFileReader();
    x3 = 2*laughterFileReader();
    temp = collector([x1 x2 x3], ...
        [ang_dft ang_cleanspeech ang_laughter]) + ...
        sqrt(noisePwr)*randn(NSampPerFrame,Nele);
    player(0.5*temp(:,3));
    sigArray(sig_idx,:) = temp;
    voice_dft(sig_idx) = x1;
    voice_cleanspeech(sig_idx) = x2;
    voice_laugh(sig_idx) = x3;
end
```

Notice that the laughter masks the speech signals, rendering them unintelligible. Plot the signal in channel 3.

```
plot(t,sigArray(:,3));
xlabel('Time (sec)'); ylabel('Amplitude (V)');
title('Signal Received at Channel 3'); ylim([-3 3]);
```



Process with a Time Delay Beamformer

The time delay beamformer compensates for the arrival time differences across the array for a signal coming from a specific direction. The time aligned multichannel signals are coherently averaged to improve the signal-to-noise ratio (SNR). Define a steering angle corresponding to the incident direction of the first speech signal and construct a time delay beamformer.

```
angSteer = ang_dft;
beamformer = phased.TimeDelayBeamformer('SensorArray',ula, ...
    'SampleRate',fs,'Direction',angSteer,'PropagationSpeed',c)
```

```
beamformer =
```

```
phased.TimeDelayBeamformer with properties:
```

```
    SensorArray: [1x1 phased.ULA]
  PropagationSpeed: 340
        SampleRate: 8000
  DirectionSource: 'Property'
          Direction: [2x1 double]
WeightsOutputPort: false
```

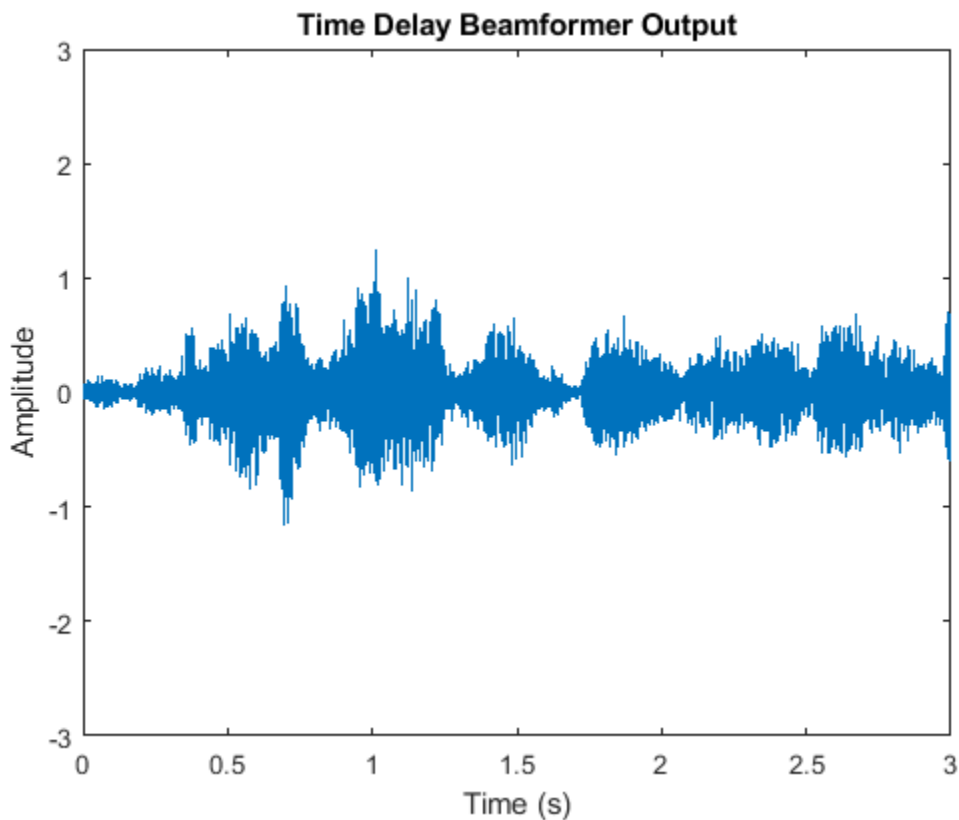
Process the synthesized signal, then plot and listen to the output of the conventional beamformer.

```
signalsource = dsp.SignalSource('Signal',sigArray, ...
    'SamplesPerFrame',NSampPerFrame);
```

```
cbf0ut = zeros(NTSample,1);

for m = 1:NSampPerFrame:NTSample
    temp = beamformer(signalsource());
    player(temp);
    cbf0ut(m:m+NSampPerFrame-1,:) = temp;
end

plot(t,cbf0ut);
xlabel('Time (s)'); ylabel ('Amplitude');
title('Time Delay Beamformer Output'); ylim([-3 3]);
```



You can measure the speech enhancement by the array gain, which is the ratio of the output signal-to-interference-plus-noise ratio (SINR) to the input SINR.

```
agCbf = pow2db(mean((voice_cleanspeech+voice_laugh).^2+noisePwr)/ ...
    mean((cbf0ut - voice_dft).^2))
```

```
agCbf =
```

```
9.5022
```

Notice that the first speech signal begins to emerge in the time delay beamformer output. You obtain an SINR improvement of 9.4 dB. However, the background laughter is still comparable to the speech. To obtain better beamformer performance, use a Frost beamformer.

Process with a Frost Beamformer

By attaching FIR filters to each sensor, the Frost beamformer has more beamforming weights to suppress the interference. It is an adaptive algorithm that places nulls at learned interference directions to better suppress the interference. In the steering direction, the Frost beamformer uses distortionless constraints to ensure desired signals are not suppressed. Create a Frost beamformer with a 20-tap FIR after each sensor.

```
frostbeamformer = ...
    phased.FrostBeamformer('SensorArray',ula,'SampleRate',fs, ...
        'PropagationSpeed',c,'FilterLength',20,'DirectionSource','Input port');
```

Process and play the synthesized signal using the Frost beamformer.

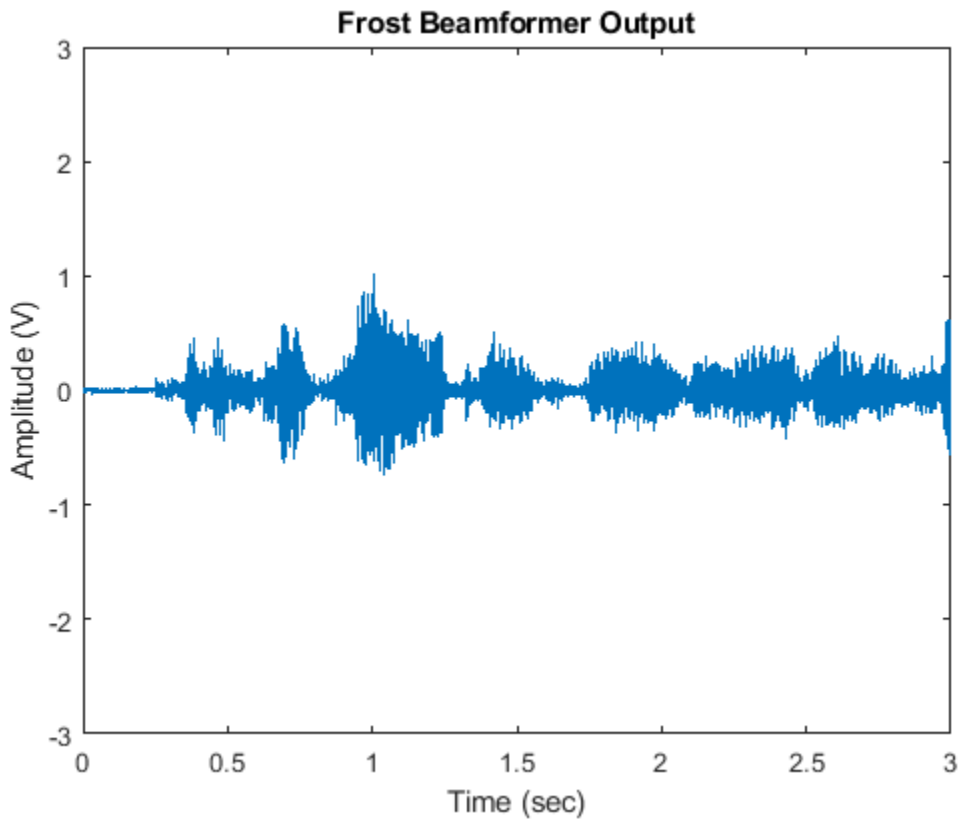
```
reset(signalsource);
FrostOut = zeros(NTSample,1);
for m = 1:NSampPerFrame:NTSample
    temp = frostbeamformer(signalsource(),ang_dft);
    player(temp);
    FrostOut(m:m+NSampPerFrame-1,:) = temp;
end

plot(t,FrostOut);
xlabel('Time (sec)'); ylabel ('Amplitude (V)');
title('Frost Beamformer Output'); ylim([-3 3]);

% Calculate the array gain
agFrost = pow2db(mean((voice_cleanspeech+voice_laugh).^2+noisePwr)/ ...
    mean((FrostOut - voice_dft).^2))

agFrost =

    14.4385
```



Notice that the interference is now canceled. The Frost beamformer has an array gain of 14.5 dB, which is about 5 dB higher than that of the time delay beamformer. The performance improvement is impressive, but has a high computational cost. In the preceding example, an FIR filter of order 20 is used for each microphone. With all 10 sensors, it needs to invert a 200-by-200 matrix, which may be expensive in real-time processing.

Use Diagonal Loading to Improve Robustness of the Frost Beamformer

Next, steer the array in the direction of the second speech signal. Suppose you only know a rough estimate of azimuth -5 degrees and elevation 5 degrees for the direction of the second speech signal.

```
release(frostbeamformer);
ang_cleanspeech_est = [-5; 5]; % Estimated steering direction

reset(signalsource);
FrostOut2 = zeros(NTSample,1);
for m = 1:NSampPerFrame:NTSample
    temp = frostbeamformer(signalsource(), ang_cleanspeech_est);
    player(temp);
    FrostOut2(m:m+NSampPerFrame-1,:) = temp;
end

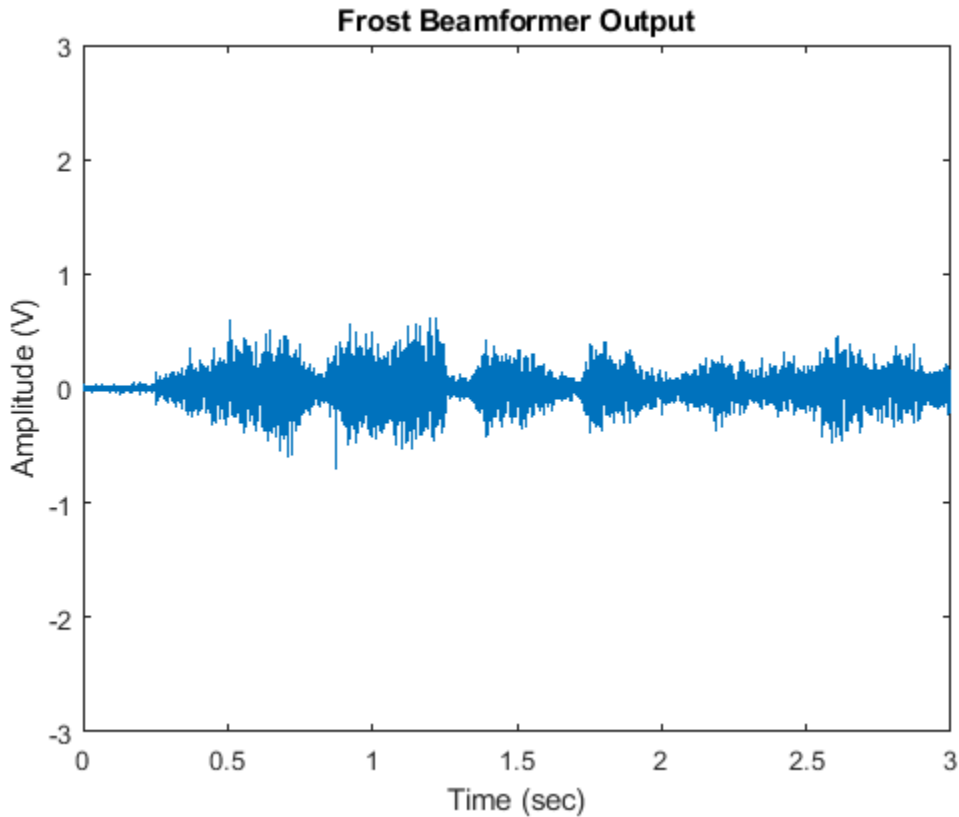
plot(t,FrostOut2);
xlabel('Time (sec)'); ylabel ('Amplitude (V)');
title('Frost Beamformer Output'); ylim([-3 3]);

% Calculate the array gain
```

```
agFrost2 = pow2db(mean((voice_dft+voice_laugh).^2+noisePwr)/ ...
    mean((FrostOut2 - voice_cleanspeech).^2))
```

```
agFrost2 =
```

```
6.1927
```



The speech is barely audible. Despite the 6.1 dB gain from the beamformer, performance suffers from the inaccurate steering direction. One way to improve the robustness of the Frost beamformer against direction of arrival mismatch is to use diagonal loading. This approach adds a small quantity to the diagonal elements of the estimated covariance matrix. The drawback of this method is that it is difficult to estimate the correct loading factor. Here you try diagonal loading with a value of $1e-3$.

```
% Specify diagonal loading value
release(frostbeamformer);
frostbeamformer.DiagonalLoadingFactor = 1e-3;

reset(signalsource);
FrostOut2_dl = zeros(NTSample,1);
for m = 1:NSampPerFrame:NTSample
    temp = frostbeamformer(signalsource(),ang_cleanspeech_est);
    player(temp);
    FrostOut2_dl(m:m+NSampPerFrame-1,:) = temp;
end
```

```

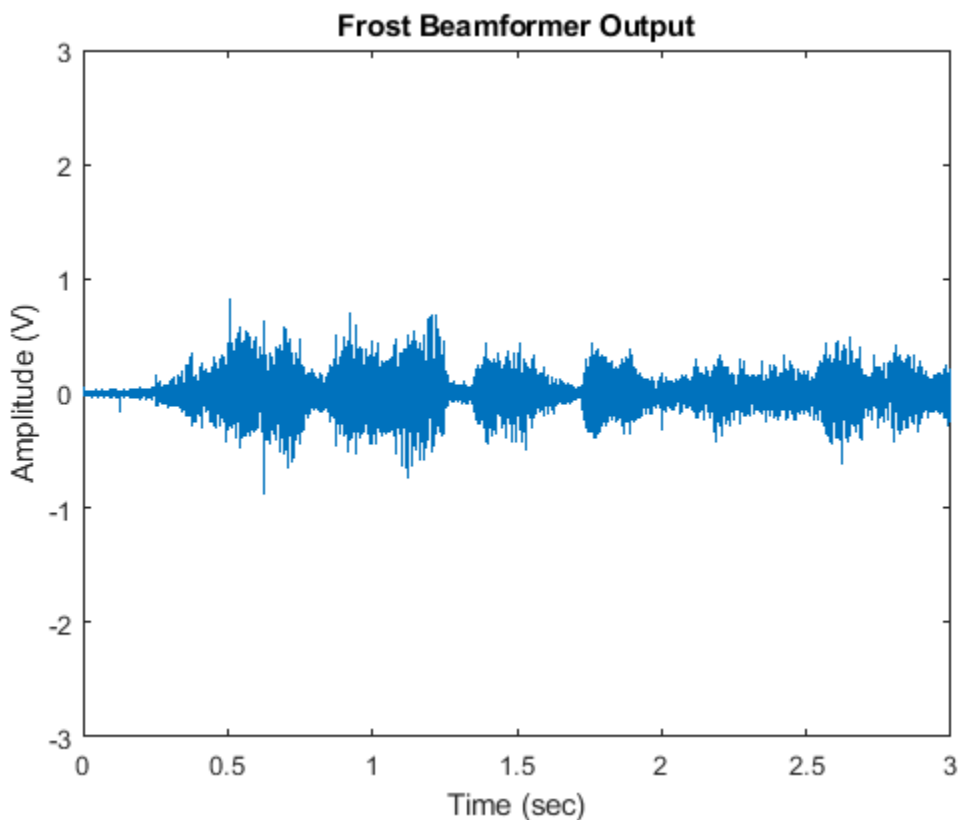
plot(t,FrostOut2_d1);
xlabel('Time (sec)'); ylabel ('Amplitude (V)');
title('Frost Beamformer Output'); ylim([-3 3]);

% Calculate the array gain
agFrost2_d1 = pow2db(mean((voice_dft+voice_laugh).^2+noisePwr)/ ...
    mean((FrostOut2_d1 - voice_cleanspeech).^2))

agFrost2_d1 =

    6.4788

```



The output speech signal is improved and you obtain a 0.3 dB gain improvement from the diagonal loading technique.

```

release(frostbeamformer);
release(signalsource);
release(player);

rng(prevS);

```

Summary

This example shows how to use time domain beamformers to retrieve speech signals from noisy microphone array measurements. The example also shows how to simulate an interference-dominant signal received by a microphone array. The example used both time delay and the Frost beamformers

and compared their performance. The Frost beamformer has a better interference suppression capability. The example also illustrates the use of diagonal loading to improve the robustness of the Frost beamformer.

Reference

[1] O. L. Frost III, An algorithm for linear constrained adaptive array processing, Proceedings of the IEEE, Vol. 60, Number 8, Aug. 1972, pp. 925-935.

Identification and Separation of Panned Audio Sources in a Stereo Mix

This example shows how to extract an audio source from a stereo mix based on its panning coefficient. This example illustrates MATLAB® and Simulink® implementations.

Introduction

Panning is a technique used to spread a mono or stereo sound signal into a new stereo or multi-channel sound signal. Panning can simulate the spatial perspective of the listener by varying the amplitude or power level of the original source across the new audio channels.

Panning is an essential component of sound engineering and stereo mixing. In studio stereo recordings, different sources or tracks (corresponding to different musical instruments, voices, and other sound sources) are often recorded separately and then mixed into a stereo signal. Panning is usually controlled by a physical or virtual control knob that may be placed anywhere from the "hard-left" position (usually referred to as 8 o'clock) to the hard-right position (4 o'clock). When a signal is panned to the 8 o'clock position, the sound only appears in the left channel (or speaker). Conversely, when a signal is panned to the 4 o'clock position, the sound only appears in the right speaker. At the 12 o'clock position, the sound is equally distributed across the two speakers. An artificial position or direction relative to the listener may be generated by varying the level of panning.

Source separation consists of the identification and extraction of individual audio sources from a stereo mix recording. Source separation has many applications, such as speech enhancement, sampling of musical sounds for electronic music composition, and real-time speech separation. It also plays a role in stereo-to-multichannel (e.g. 5.1 or 7.1) upmix, where the different extracted sources may be distributed across the channels of the new mix.

This example showcases a source separation algorithm applied to an audio stereo signal. The stereo signal is a mix of two independently panned audio sources: The first source is a man counting from one to ten, and the second source is a toy train whistle.

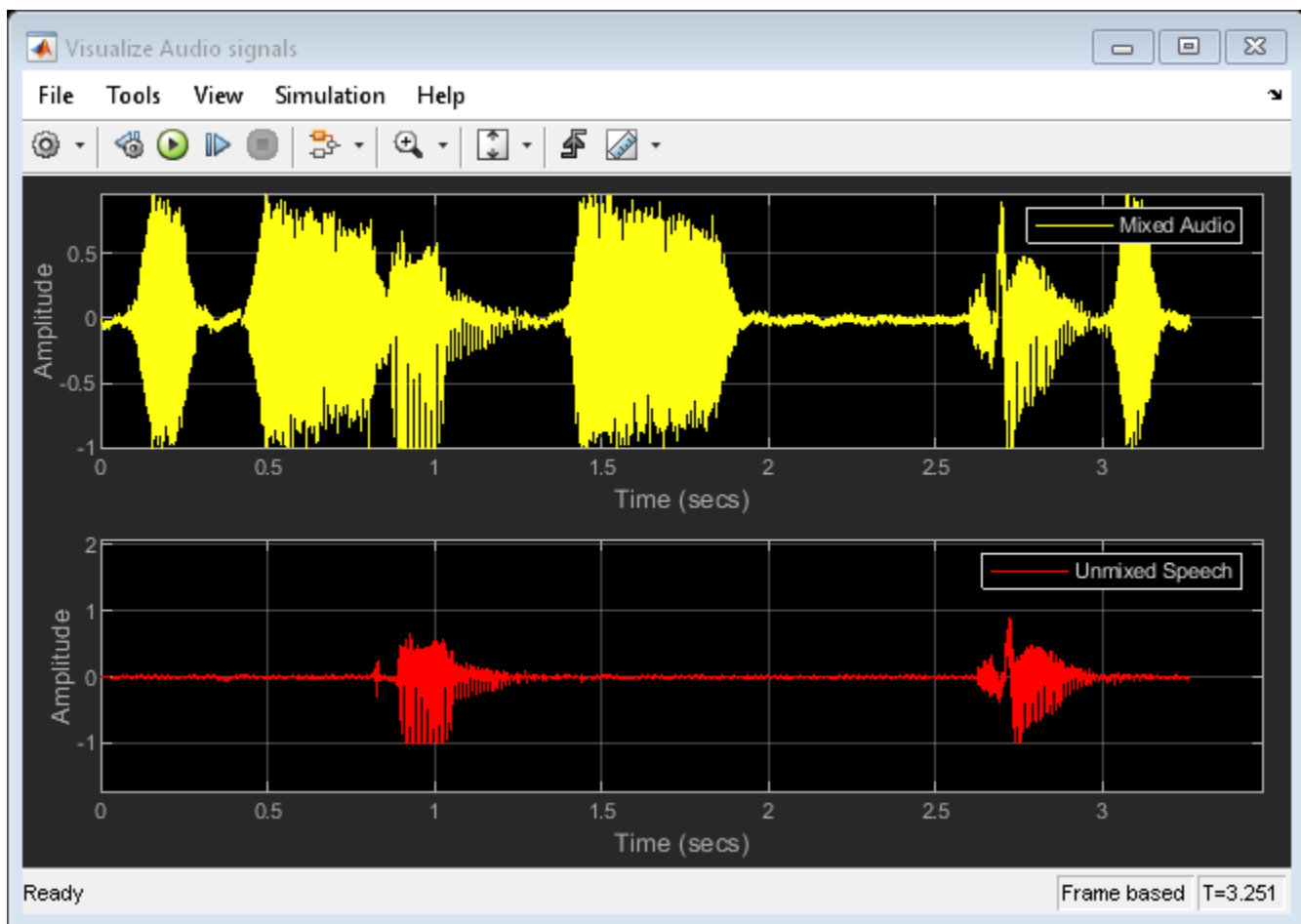
The example uses a frequency-domain technique based on short-time FFT analysis to identify and separate the sources based on their different panning coefficients.

Simulink Version

The model `audiosourceseparation` implements the panned audio source separation example.

channels of the stereo mix. A frequency-domain, time-varying panning index function [1] is computed based on the cross-correlations of the left and right short-time FFT pair. There is a one-to-one relationship between the panning coefficient of the sources and the derived panning index. A running-window histogram is implemented in the 'Panning Index Histogram' subsystem to identify the dominant panning indices in the mix. The desired source is then unmixed by applying a masking function modeled used a Gaussian window centered at the target panning index. Finally, the unmixed extracted source is obtained by applying a short-time IFFT.

The mixed signal and the extracted speech signal are visualized using a scope. The estimated panning coefficient is shown on a Display block. You can listen to either the mixed stereo or the unmixed speech source by flipping the manual switch at the input of the Audio Device Writer block. The streaming algorithm can adapt to a change in the value of the panning coefficient. For example, you can modify the panning coefficient from 0.4 to 0.6 and observe that the displayed panning coefficient value is updated with the correct value.



MATLAB Version

HelperAudioSourceSeparationSim is the MATLAB implementation of the panned source separation example. It instantiates, initializes and steps through the objects forming the algorithm.

The audioSourceSeparationApp function wraps around HelperAudioSourceSeparationSim and iteratively calls it. It plots the mixed audio and unmixed speech signals using a scope. It also

opens a UI designed to interact with the simulation. The UI allows you to tune the panning coefficient of the speech source. You can also toggle between listening to either the mixed signal (whistle + speech) or the unmixed speech signal by changing the value of the 'Audio Output' drop-down box in the UI. There are also three buttons on the UI - the 'Reset' button will reset the simulation internal state to its initial condition and the 'Pause Simulation' button will hold the simulation until you press on it again. The simulation may be terminated by either closing the UI or by clicking on the 'Stop simulation' button.

Execute `audioSourceSeparationApp` to run the simulation and plot the results. Note that the simulation runs until you explicitly stop it.

MATLAB Coder™ can be used to generate C code for the `HelperAudioSourceSeparationSim` function. In order to generate a MEX-file for your platform, execute the command `HelperSourceSeparationCodeGeneration` from a folder with write-permission.

By calling the wrapper function `audioSourceSeparationApp` with `'true'` as an argument, the generated MEX-file can be used instead of `HelperAudioSourceSeparationSim` for the simulation. In this scenario, the UI is still running inside the MATLAB environment, but the main processing algorithm is being performed by a MEX-file. Performance is improved in this mode without compromising the ability to tune parameters.

References

[1] 'A Frequency-Domain Approach to Multichannel Upmix', Advendano, Carlos; Jot, Jean-Marc, JAES Volume 52 Issue 7/8 pp. 740-749; July 2004

Live Direction Of Arrival Estimation with a Linear Microphone Array

This example shows how to acquire and process live multichannel audio. It also presents a simple algorithm for estimating the Direction Of Arrival (DOA) of a sound source using multiple microphone pairs within a linear array.

Select and Configure the Source of Audio Samples

If a multichannel input audio interface is available, then modify this script to set `sourceChoice` to `'live'`. In this mode the example uses live audio input signals. The example assumes all inputs (two or more) are driven by microphones arranged on a linear array. If no microphone array or multichannel audio card is available, then set `sourceChoice` to `'recorded'`. In this mode the example uses prerecorded audio samples acquired with a linear array. For `sourceChoice = 'live'`, the following code uses `audioDeviceReader` to acquire 4 live audio channels through a Microsoft Kinect™ for Windows®. To use another microphone array setup, ensure the installed audio device driver is one of the conventional types supported by MATLAB and set the `Device` property of `audioDeviceReader` accordingly. You can query valid `Device` assignments for your computer by calling the `getAudioDevices` object function of `audioDeviceReader`. Note that even when using Microsoft Kinect, the device name can vary across machines and may not match the one used in this example. Use tab completion to get the correct name on your machine.

```
sourceChoice =  ;
```

Set the duration of live processing. Set how many samples per channel to acquire and process each iteration.

```
endTime = 20;
audioFrameLength = 3200;
```

Create the source.

```
switch sourceChoice
    case 'live'
        fs = 16000;
        audioInput = audioDeviceReader( ...
            'Device','Microphone Array (Microsoft Kinect USB Audio)', ...
            'SampleRate',fs, ...
            'NumChannels',4, ...
            'OutputDataType','double', ...
            'SamplesPerFrame',audioFrameLength);
    case 'recorded'
        % This audio file holds a 20-second recording of 4 raw audio
        % channels acquired with a Microsoft Kinect(TM) for Windows(R) in
        % the presence of a noisy source moving in front of the array
        % roughly from -40 to about +40 degrees and then back to the
        % initial position.
        audioFileName = 'AudioArray-16-16-4channels-20secs.wav';
        audioInput = dsp.AudioFileReader( ...
            'OutputDataType','double', ...
            'Filename',audioFileName, ...
            'PlayCount',inf, ...
            'SamplesPerFrame',audioFrameLength);
        fs = audioInput.SampleRate;
end
```

Define Array Geometry

The following values identify the approximate linear coordinates of the 4 built-in microphones of the Microsoft Kinect™ relative to the position of the RGB camera (not used in this example). For 3D coordinates use `[[x1;y1;z1], [x2;y2;z2], ..., [xN;yN;zN]]`

```
micPositions = [-0.088, 0.042, 0.078, 0.11];
```

Form Microphone Pairs

The algorithm used in this example works with pairs of microphones independently. It then combines the individual DOA estimates to provide a single live DOA output. The more pairs available, the more robust (yet computationally expensive) DOA estimation. The maximum number of pairs available can be computed as `nchoosek(length(micPositions), 2)`. In this case, the 3 pairs with the largest inter-microphone distances are selected. The larger the inter-microphone distance the more sensitive the DOA estimate. Each column of the following matrix describes a choice of microphone pair within the array. All values must be integers between 1 and `length(micPositions)`.

```
micPairs = [1 4; 1 3; 1 2];
numPairs = size(micPairs, 1);
```

Initialize DOA Visualization

Create an instance of the helper plotting object `DOADisplay`. This displays the estimated DOA live with an arrow on a polar plot.

```
DOAPointer = DOADisplay();
```

Create and Configure the Algorithmic Building Blocks

Use a helper object to rearrange the input samples according to how the microphone pairs are selected.

```
bufferLength = 64;
preprocessor = PairArrayPreprocessor( ...
    'MicPositions', micPositions, ...
    'MicPairs', micPairs, ...
    'BufferLength', bufferLength);
micSeparations = getPairSeparations(preprocessor);
```

The main algorithmic building block of this example is a cross-correlator. That is used in conjunction with an interpolator to ensure a finer DOA resolution. In this simple case it is sufficient to use the same two objects across the different pairs available. In general, however, different channels may need to independently save their internal states and hence to be handled by separate objects.

```
XCorrelator = dsp.Crosscorrelator('Method', 'Frequency Domain');
interpFactor = 8;
b = interpFactor * fir1((2*interpFactor*8-1), 1/interpFactor);
groupDelay = median(grpdelay(b));
interpolator = dsp.FIRInterpolator('InterpolationFactor', interpFactor, 'Numerator', b);
```

Acquire and Process Signals in a Loop

For each iteration of the following while loop: read `audioFrameLength` samples for each audio channel, process the data to estimate a DOA value and display the result on a bespoke arrow-based polar visualization.

```
tic
for idx = 1:(endTime*fs/audioFrameLength)
    cycleStart = toc;
    % Read a multichannel frame from the audio source
    % The returned array is of size AudioFrameLength x size(micPositions,2)
    multichannelAudioFrame = audioInput();

    % Rearrange the acquired sample in 4-D array of size
    % bufferLength x numBuffers x 2 x numPairs where 2 is the number of
    % channels per microphone pair
    bufferedFrame = preprocessor(multichannelAudioFrame);

    % First, estimate the DOA for each pair, independently

    % Initialize arrays used across available pairs
    numBuffers = size(bufferedFrame, 2);
    delays = zeros(1,numPairs);
    anglesInRadians = zeros(1,numPairs);
    xcDense = zeros((2*bufferLength-1)*interpFactor, numPairs);

    % Loop through available pairs
    for kPair = 1:numPairs
        % Estimate inter-microphone delay for each 2-channel buffer
        delayVector = zeros(numBuffers, 1);
        for kBuffer = 1:numBuffers
            % Cross-correlate pair channels to get a coarse
            % crosscorrelation
            xcCoarse = XCorrelator( ...
                bufferedFrame(:,kBuffer,1,kPair), ...
                bufferedFrame(:,kBuffer,2,kPair));

            % Interpolate to increase spatial resolution
            xcDense = interpolator(flipud(xcCoarse));

            % Extract position of maximum, equal to delay in sample time
            % units, including the group delay of the interpolation filter
            [~,idxloc] = max(xcDense);
            delayVector(kBuffer) = ...
                (idxloc - groupDelay)/interpFactor - bufferLength;
        end

        % Combine DOA estimation across pairs by selecting the median value
        delays(kPair) = median(delayVector);

        % Convert delay into angle using the microsoft pair spatial
        % separations provided
        anglesInRadians(kPair) = HelperDelayToAngle(delays(kPair), fs, ...
            micSeparations(kPair));
    end

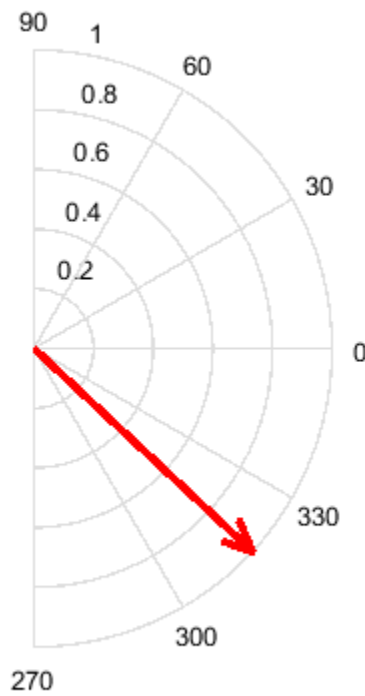
    % Combine DOA estimation across pairs by keeping only the median value
    DOAInRadians = median(anglesInRadians);

    % Arrow display
    DOAPointer(DOAInRadians)

    % Delay cycle execution artificially if using recorded data
    if(strcmp(sourceChoice,'recorded'))
```



```
        pause(audioFrameLength/fs - toc + cycleStart)
    end
end
```



```
release(audioInput)
```

Positional Audio

This example shows several basic aspects of audio signal positioning. The listener occupies a location in the center of a circle, and the position of the sound source is varied so that it remains within the circle. In this example, the sound source is a monaural recording of a helicopter. The sound field is represented by five discrete speaker locations on the circumference of the circle and a low-frequency output that is presumed to be in the center of the circle.

Example Prerequisites

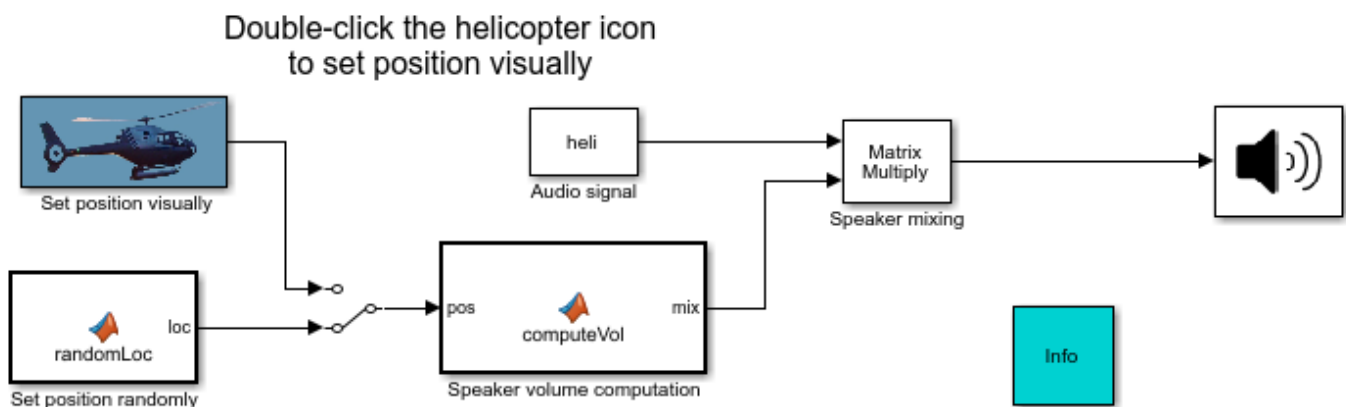
This example requires a 5.1-channel speaker configuration, and relies on the audio channels being mapped to physical locations as follows:

- 1 Front left
- 2 Front right
- 3 Front center
- 4 Low frequency
- 5 Rear left
- 6 Rear right

This is the default Windows® speaker configuration for 5.1 channels. Depending on the type of sound card used, this example may work reasonably well for other speaker configurations.

Example Basics

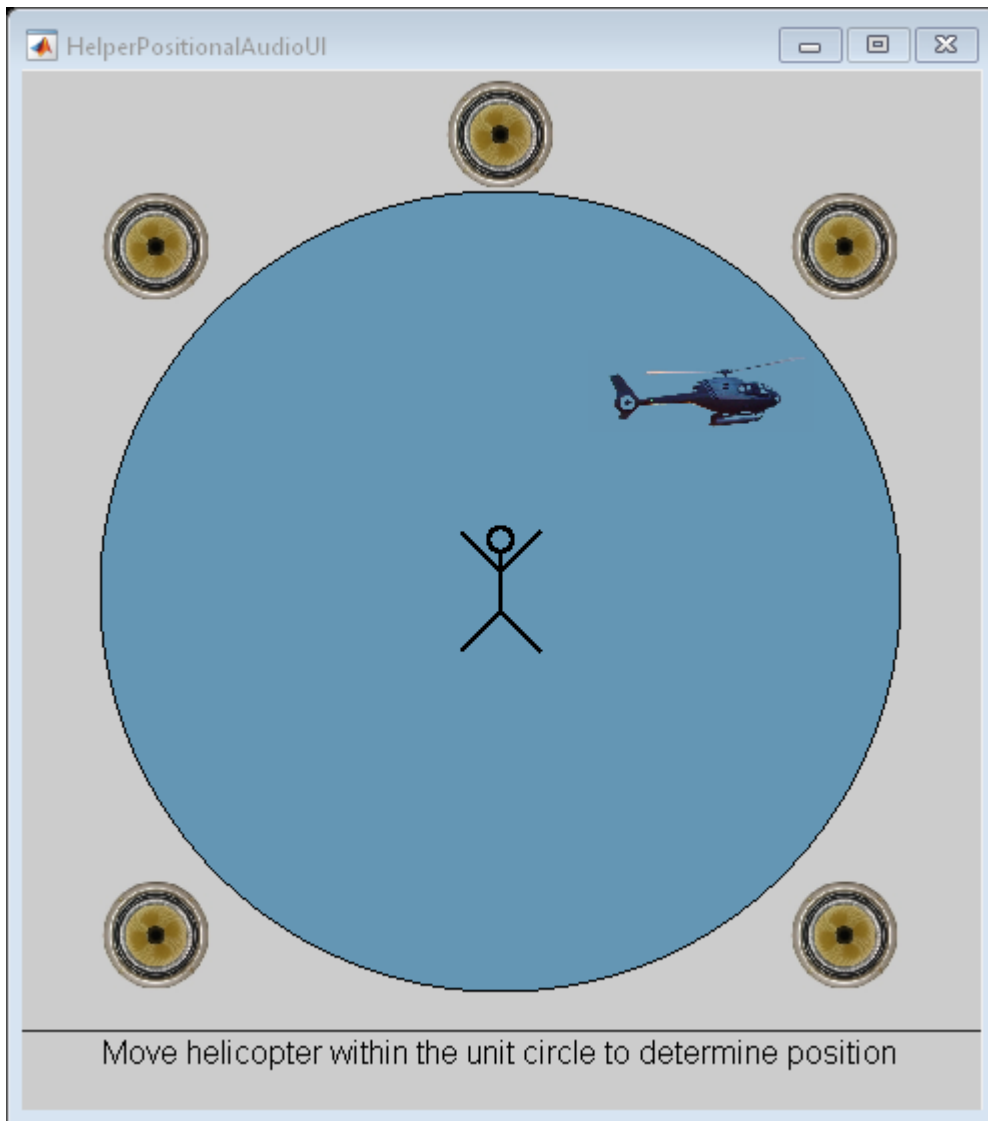
There are two source blocks of interest in the model. The first is the audio signal itself, and the second is the spatial location of the helicopter. The spatial location of the helicopter is represented by a pair of Cartesian coordinates that are constrained to lie within the unit circle. By default, this location is determined by the block labeled "Set position randomly." This block supplies the input for the MATLAB Function block labeled "Speaker volume computation," which determines a matrix of speaker volumes. The outer product of the sound source is then taken with the speaker position matrix, which is then supplied to the six speakers via the To Audio Device block.



Copyright 2007-2015 The MathWorks, Inc.

Manually Determining the Helicopter Position

You can also determine the helicopter position manually. To do this, select the switch in the model so that the signal being supplied to the computeVol block is coming from the block labeled "Set position visually." Then, double-click on the new source block. A GUI appears that enables you to move the helicopter to different locations within the circle using the mouse, thereby changing the speaker amplitudes.



Spatial Mixing Algorithm

The monaural audio source is mixed into six channels, each of which corresponds to a speaker. There is one low-frequency channel in the center of the circle and five speakers that lie on the circumference, as shown in the grey area of the GUI above. The listener is represented by a stick figure in the center of the circle.

The following algorithm is used to determine the speaker amplitudes:

1. At the center of the circle, all of the amplitudes are equal. The value for each speaker, including the low-frequency speaker, is set to $1/\sqrt{5}$.

2. On the perimeter of the circle, the amplitudes of the speakers are determined using Vector Base Amplitude Panning (VBAP). This algorithm operates as follows:

a) Determine the two speakers on either side of the source or, in the degenerate case, the single speaker.

b) Interpret the vectors determined by the speaker positions in (a) as basis vectors. Use these basis vectors to represent the normalized source position vector. The coefficients in this new basis represent the relative speaker amplitudes after normalization.

For this part of the algorithm, the amplitude of the low-frequency channel is set to zero.

3) As the source moves from the center to the periphery, there is a transition from algorithm (1) to algorithm (2). This transition decays as a cubic function of the radial distance. The amplitude vectors are normalized so the power is constant independent of source location.

4) Finally, the amplitudes decay as the distance from the center increases according to an inverse square law, such that the amplitude at the perimeter of the circle is one-quarter of the amplitude in the center.

For more details about Vector Base Amplitude Panning, please consult the references.

References

Pulki, Ville. "Virtual Sound Source Positioning Using Vector Base Amplitude Panning." *Journal Audio Engineering Society*. Vol 45, No 6. June 1997.

Surround Sound Matrix Encoding and Decoding

This example shows how to generate a stereo signal from a multichannel audio signal using matrix encoding, and how to recover the original channels from the stereo mix using matrix decoding. This example illustrates MATLAB® and Simulink® implementations. This example also shows how performance can be improved by using dataflow execution domain.

Introduction

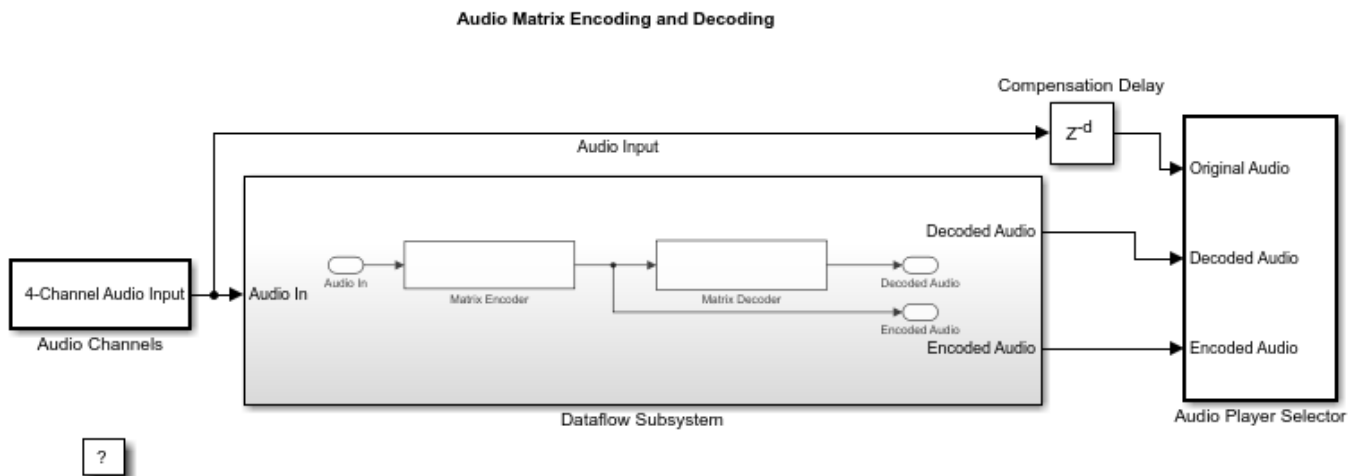
Matrix decoding is an audio technique that decodes an audio signal with M channels into an audio signal with N channels ($N > M$) for play back on a system with N speakers. The original audio signal is usually generated using a matrix encoder, which transforms N-channel signals to M-channel signals.

Matrix encoding and decoding enables the same audio content to be played on different systems. For example, a surround sound multichannel signal may be encoded into a stereo signal. The stereo signal may be played back on a stereo system to accommodate settings where a surround sound receiver does not exist, or it may be decoded and played as surround if surround equipment is present [1].

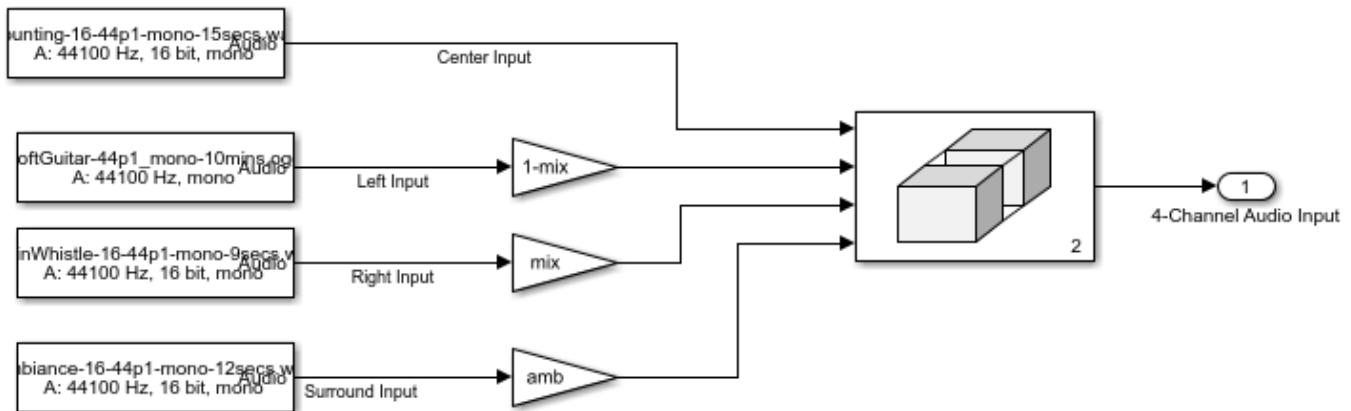
In this example, we showcase a matrix encoder used to encode a four-channel signal (left, right, center and surround) to a stereo signal. The four original signals are then regenerated using a matrix decoder. This example is a simplified version of the encoding and decoding scheme used in the Dolby Pro Logic system [2].

Simulink Version

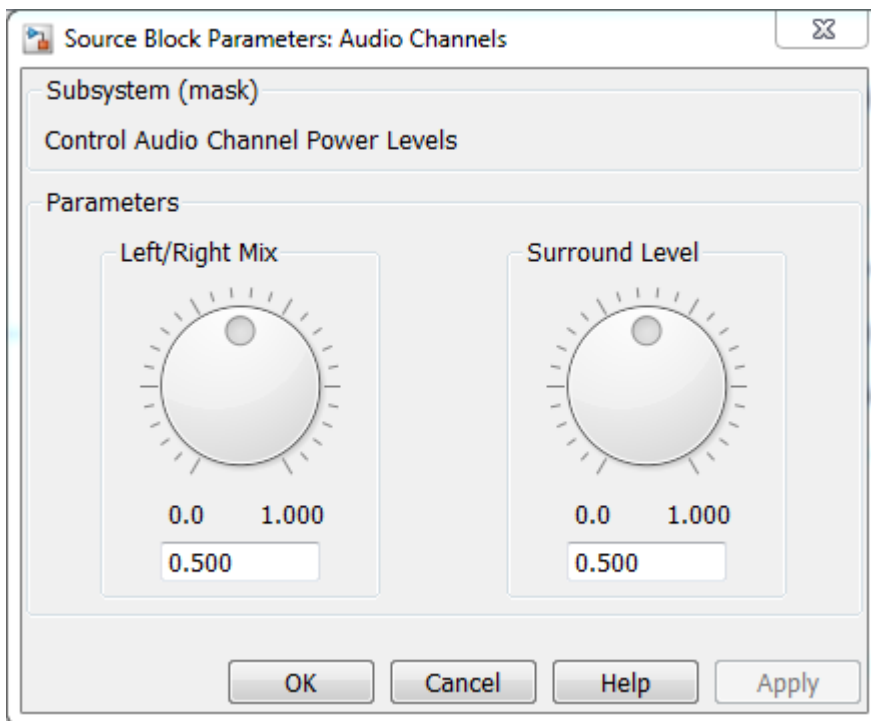
The **audiomatrixdecoding** model implements the audio matrix encoding/decoding example.



The input to the matrix encoder consists of four separate audio channels (center, left, right and surround).



Double-click the **Audio Channels** subsystem to launch a tuning dialog. The dialog enables you to control the relative power between the right channel and left channel inputs, as well as the power level of the surround channel.



You can also toggle between listening to any of the original, encoded or decoded audio channels by double-clicking the **Audio Player Selector** subsystem and selecting the channel of your choice from the dialog drop down menu.

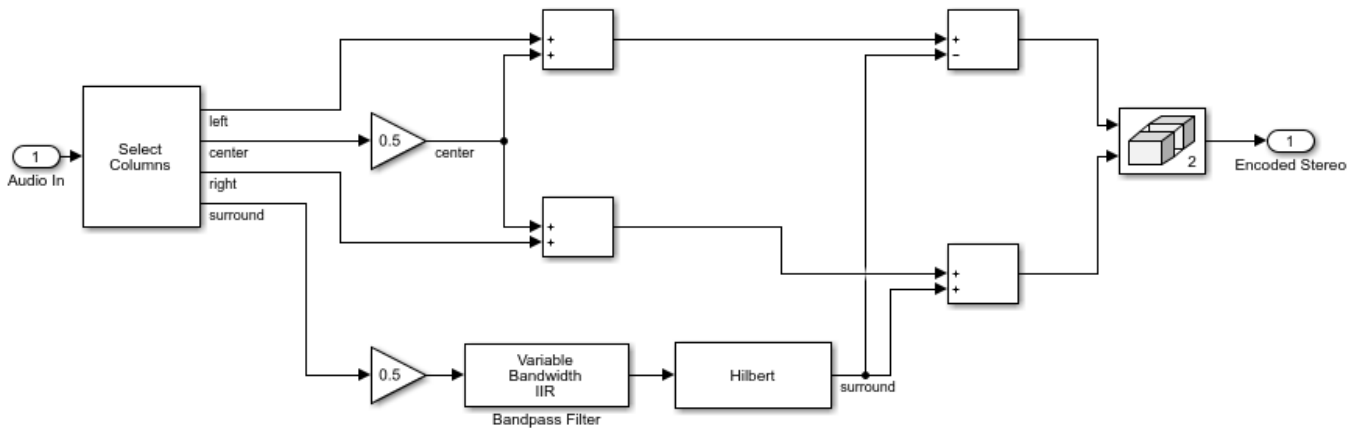
Matrix Encoder

The **Matrix Encoder** encodes the four input channels into a stereo signal.

Notice that since the input left and right channels only contribute to the output left and right channels, respectively, the output stereo signal conserves the balance between left and right channels.

The surround input channel is passed through a Hilbert transformer, thereby creating a 180 degree phase differential between the surround component feeding the left and right stereo outputs [2].

You may listen to the encoded left and right stereo signals by double-clicking the **Audio Player Selector** subsystem and selecting either the 'Encoded Total Left' or 'Encoded Total Right' channels.



Matrix Decoder

The **Matrix Decoder** extracts the four original channels from the encoded stereo signal.

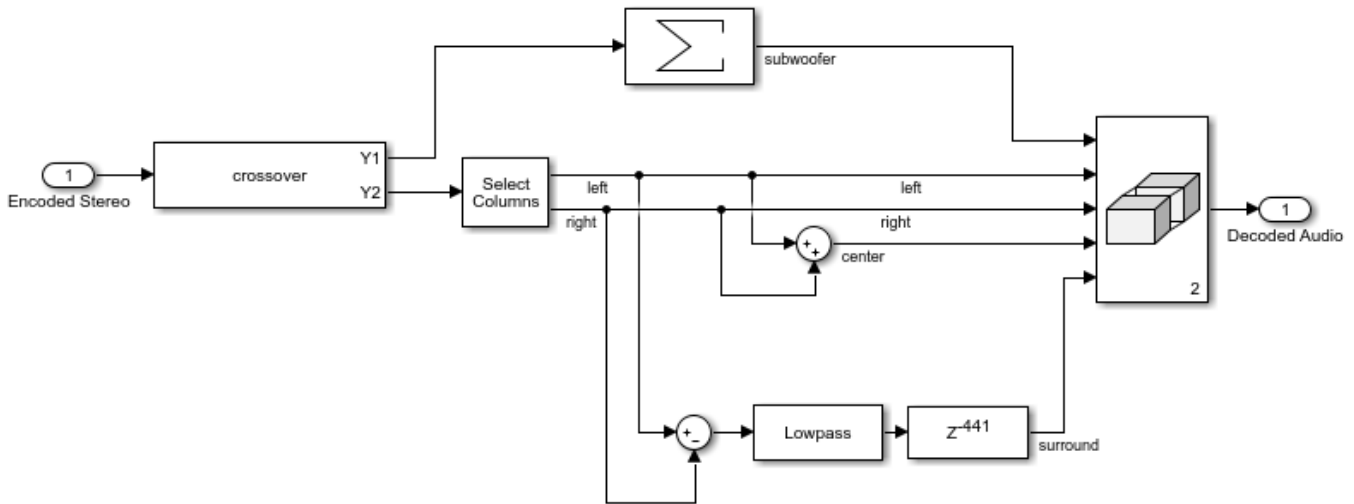
The lowpass frequencies are first separated using a Linkwitz-Riley cross-over filter. For more information about the implementation of the Linkwitz-Riley filter, refer to "Multiband Dynamic Range Compression" on page 1-128.

The left and right stereo channels are passed through to the left and right output channels, respectively. Therefore, there is no loss of separation between left and right channels in the output.

The center output channel is equal to the sum of the stereo input signals, thereby cancelling the phase-shifted surround left and right components.

The surround output channel is derived by first taking the difference of the stereo signals. Since the original input center signal contributes equally to both stereo channels, the center channel does not leak into the output surround signal. Moreover, note that the original left and right signals contribute to the output surround channel. The surround signal is delayed by 10 msec to achieve a precedence effect [3].

You may listen to the decoded surround signal by double-clicking the **Audio Player Selector** subsystem and selecting one of the decoded signals.

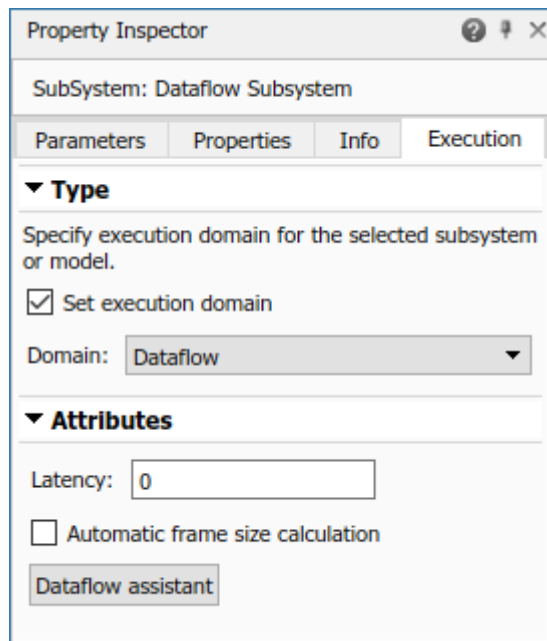


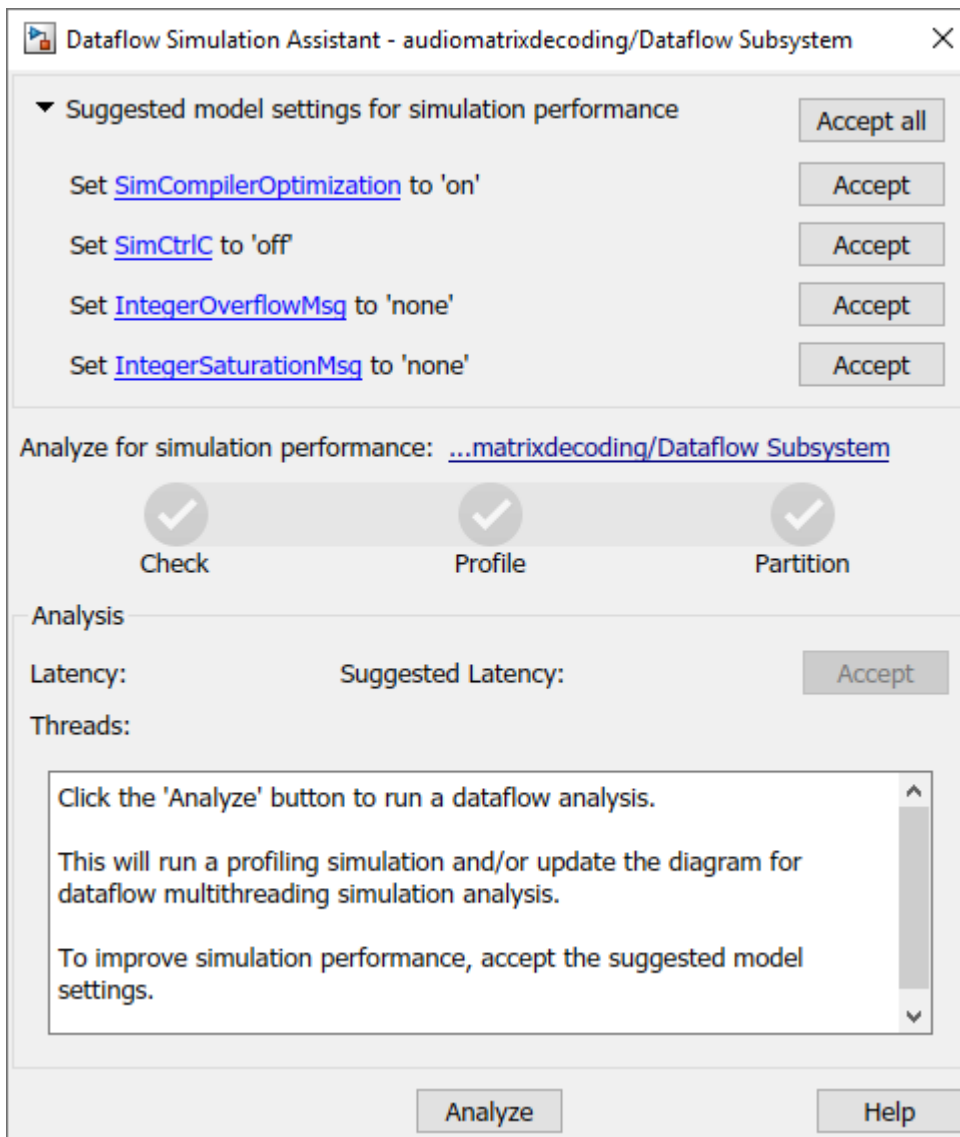
Improve Simulation Performance Using Dataflow Domain

This example can use dataflow execution domain in Simulink to make use of multiple cores on your desktop to improve simulation performance. To learn more about dataflow and how to run Simulink models using multiple threads, see “Multicore Execution using Dataflow Domain”.

Specify Dataflow Execution Domain

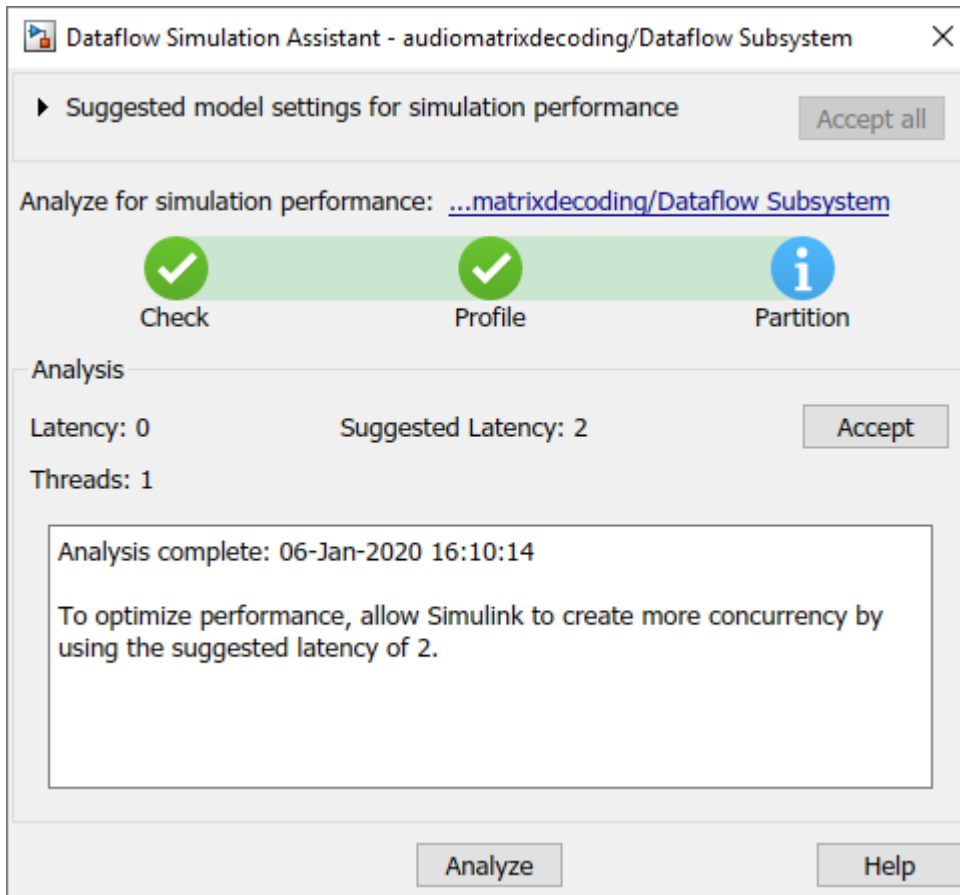
In Simulink, you specify dataflow as the execution domain for a subsystem by setting the Domain parameter to Dataflow using Property Inspector. Dataflow domains automatically partition your model and simulate the system using multiple threads for better simulation performance. Once you set the Domain parameter to Dataflow, you can use Dataflow Simulation Assistant to analyze your model to get better performance. You can open Dataflow Simulation Assistant by clicking on the **Dataflow assistant** button below the **Automatic frame size calculation** parameter in Property Inspector.



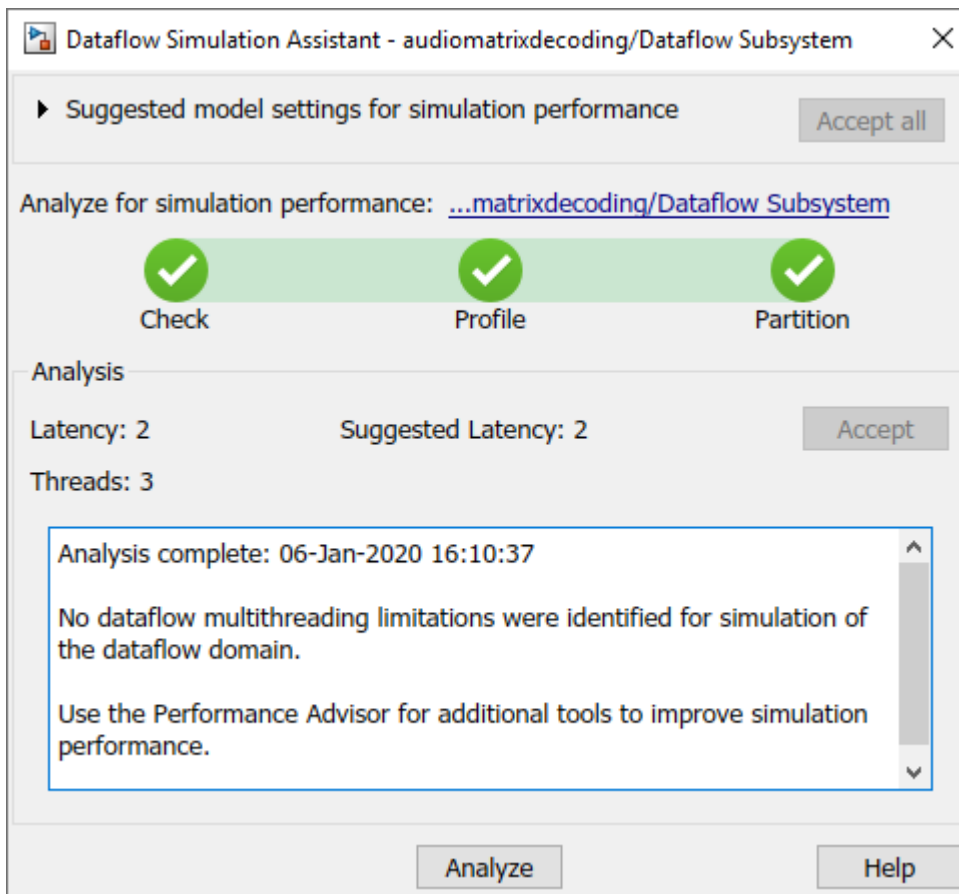


Multicore Simulation of Dataflow Domain

The Dataflow Simulation Assistant suggests changing model settings for optimal simulation performance. To accept the proposed model settings, next to **Suggested model settings for simulation performance**, click **Accept all**. Alternatively, you can expand the section to change the settings individually. In the Dataflow Simulation Assistant, click the **Analyze** button to start the analysis of the dataflow domain for simulation performance. Once the analysis is finished, the Dataflow Simulation Assistant shows how many threads the dataflow subsystem will use during simulation.

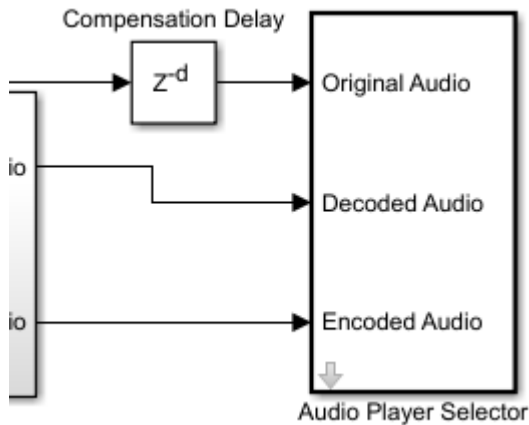


For this model, the assistant shows one thread because the data dependency between blocks prevents them from being executed concurrently. By pipelining the data dependent blocks, the Dataflow Subsystem can increase concurrency for higher data throughput. Dataflow Simulation Assistant shows the recommended number of pipeline delays as **Suggested Latency**. For this model, the suggested latency is two. Click the **Accept** button next to **Suggested Latency** in the Dataflow Simulation Assistant to use the recommended latency for Dataflow Subsystem. Dataflow Simulation Assistant now shows the number of threads as three, meaning that the blocks inside the dataflow subsystem simulate in parallel using three threads.



Compensate for Latency

When latency is increased in the dataflow execution domain to break data dependencies between blocks and create concurrency, that delay needs to be accounted for in other parts of the model. For example, signals that are compared or combined with the signals at the output ports of the Dataflow Subsystem must be delayed to align in time with the signals at the output ports of the Dataflow Subsystem. In this example, the audio signal from the Audio Channels block that goes to the Audio Player Selector must be delayed to align with other signals going into the Audio Player Selector block. To compensate for the latency specified on the dataflow subsystem, use a delay block to delay this signal by two frames. For this signal, the frame length is 1024. A delay value of 2048 is set in the delay block to align the signal from the Audio Channels block and the signal processed through Dataflow Subsystem.



Dataflow Simulation Performance

To measure performance improvement gained by using dataflow, compare execution time of the model with and without dataflow. The Audio Device Writer block runs in real time and limits the simulation speed of the model to real time. Comment out the Audio Device Writer block when measuring execution time. On a Windows desktop computer with Intel® Xeon® CPU W-2133 @ 3.6 GHz 6 Cores 12 Threads processor this model using dataflow domain executes 2.6x times faster compared to original model.

MATLAB Version

`HelperAudioMatrixDecoderSim` is the MATLAB function containing the audio matrix decoder example's implementation. It instantiates, initializes and steps through the objects forming the algorithm.

The function `audioMatrixDecoderApp` wraps around `HelperAudioMatrixDecoderSim` and iteratively calls it.

Execute `audioMatrixDecoderApp` to run the simulation. Note that the simulation runs until you explicitly stop it.

`audioMatrixDecoderApp` launches a UI designed to interact with the simulation. Similar to the Simulink version of the example, the UI allows you to tune the relative power between the right channel and left channel inputs, as well as the power level of the surround channel. You can also toggle between listening to any of the original, encoded or decoded audio channels by changing the value of the 'Audio Output' drop-down box in the UI.

There are also three buttons on the UI - the 'Reset' button will reset the simulation internal state to its initial condition and the 'Pause Simulation' button will hold the simulation until you press on it again. The simulation may be terminated by either closing the UI or by clicking on the 'Stop simulation' button.

MATLAB Coder can be used to generate C code for the function `HelperAudioMatrixDecoderSim`. In order to generate a MEX-file for your platform, execute the command `HelperMatrixDecodingCodeGeneration` from a folder with write permissions.

By calling the wrapper function `audioMatrixDecoderApp` with 'true' as an argument, the generated MEX-file can be used instead of `HelperAudioMatrixDecoderSim` for the simulation. In

this scenario, the UI is still running inside the MATLAB environment, but the main processing algorithm is being performed by a MEX-file. Performance is improved in this mode without compromising the ability to tune parameters.

References

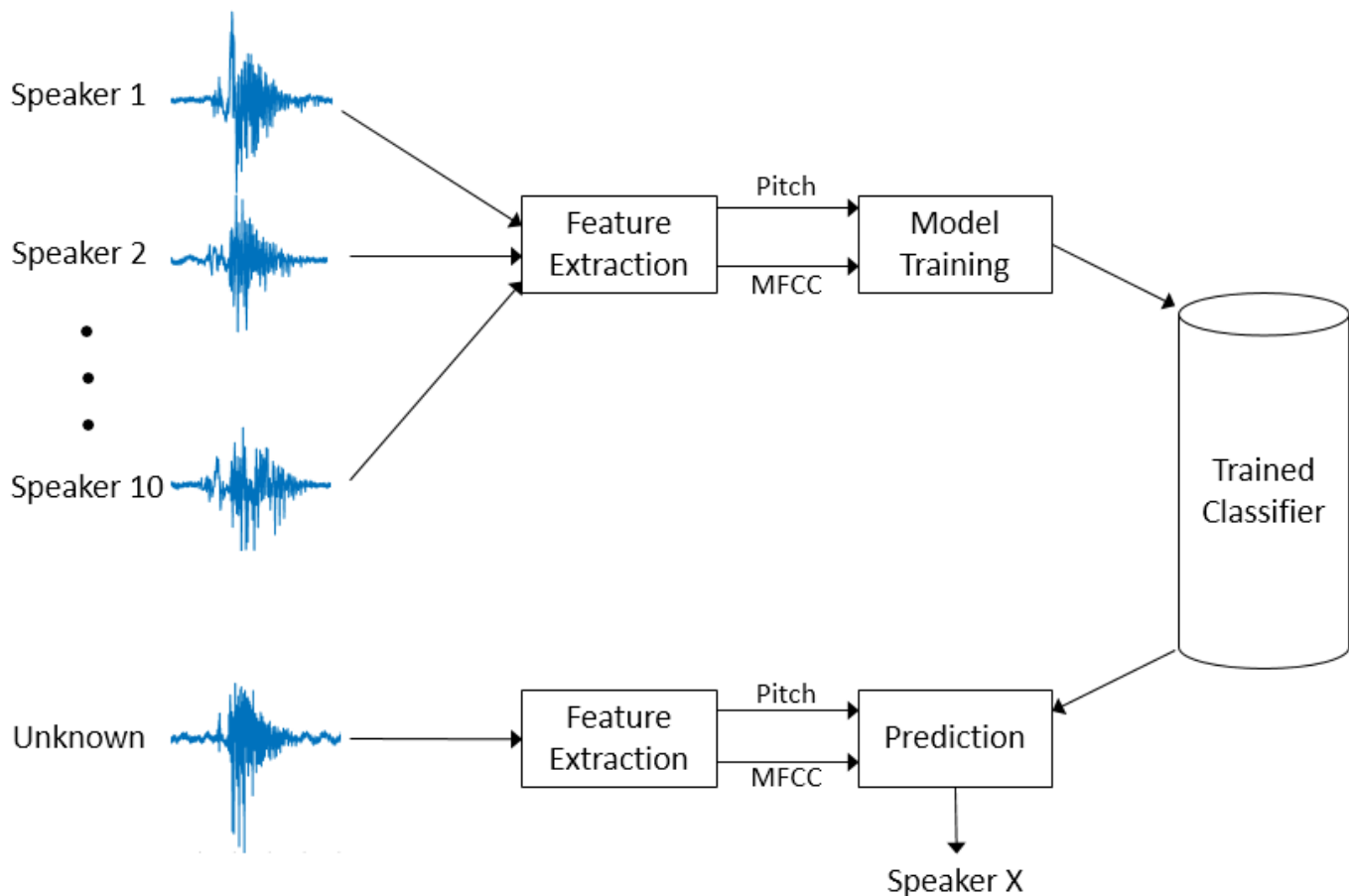
- [1] https://en.wikipedia.org/wiki/Matrix_decoder
- [2] Dolby Pro Logic Surround Decoder: Principles of Operation, Roger Dressler, Dolby Labs
- [3] https://en.wikipedia.org/wiki/Precedence_effect

Speaker Identification Using Pitch and MFCC

This example demonstrates a machine learning approach to identify people based on features extracted from recorded speech. The features used to train the classifier are the pitch of the voiced segments of the speech and the mel frequency cepstrum coefficients (MFCC). This is a closed-set speaker identification: the audio of the speaker under test is compared against all the available speaker models (a finite set) and the closest match is returned.

Introduction

The approach used in this example for speaker identification is shown in the diagram.



Pitch and MFCC are extracted from speech signals recorded for 10 speakers. These features are used to train a K-nearest neighbor (KNN) classifier. Then, new speech signals that need to be classified go through the same feature extraction. The trained KNN classifier predicts which one of the 10 speakers is the closest match.

Features Used for Classification

This section discusses pitch and MFCC, the two features that are used to classify speakers.

Pitch

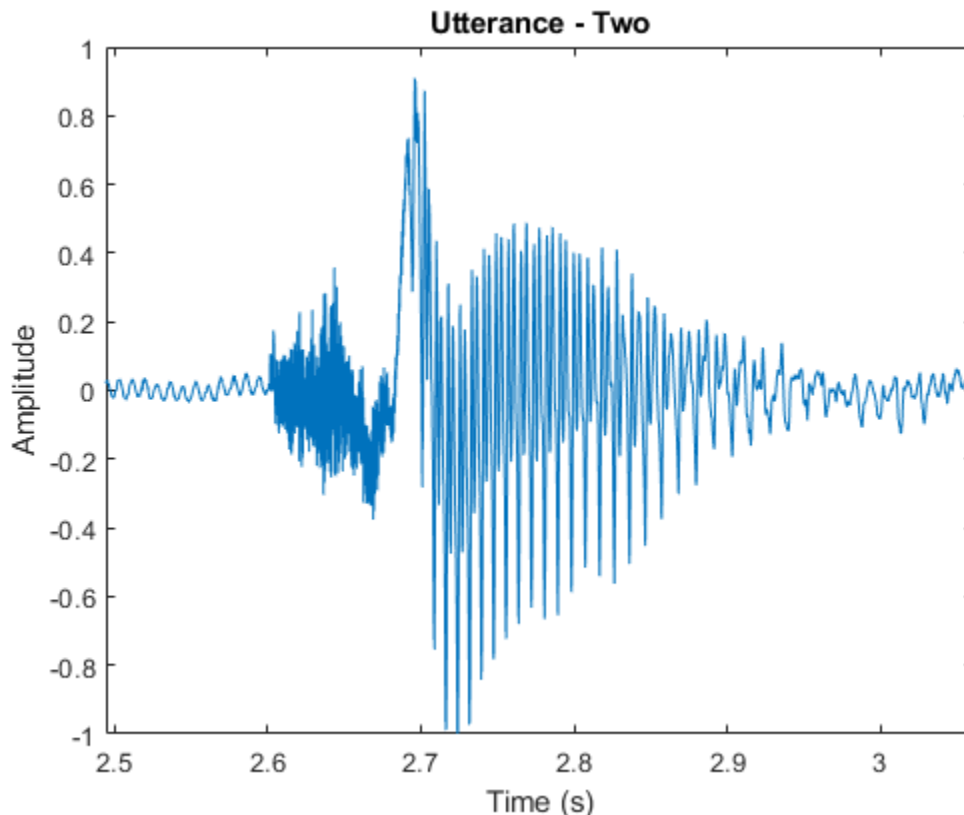
Speech can be broadly categorized as *voiced* and *unvoiced*. In the case of voiced speech, air from the lungs is modulated by vocal cords and results in a quasi-periodic excitation. The resulting sound is dominated by a relatively low-frequency oscillation, referred to as *pitch*. In the case of unvoiced speech, air from the lungs passes through a constriction in the vocal tract and becomes a turbulent, noise-like excitation. In the source-filter model of speech, the excitation is referred to as the source, and the vocal tract is referred to as the filter. Characterizing the source is an important part of characterizing the speech system.

As an example of voiced and unvoiced speech, consider a time-domain representation of the word "two" (/T UW/). The consonant /T/ (unvoiced speech) looks like noise, while the vowel /UW/ (voiced speech) is characterized by a strong fundamental frequency.

```
[audioIn, fs] = audioread('Counting-16-44p1-mono-15secs.wav');
twoStart = 110e3;
twoStop = 135e3;
audioIn = audioIn(twoStart:twoStop);
timeVector = linspace((twoStart/fs),(twoStop/fs),numel(audioIn));

sound(audioIn,fs)

figure
plot(timeVector,audioIn)
axis([(twoStart/fs) (twoStop/fs) -1 1])
ylabel('Amplitude')
xlabel('Time (s)')
title('Utterance - Two')
```



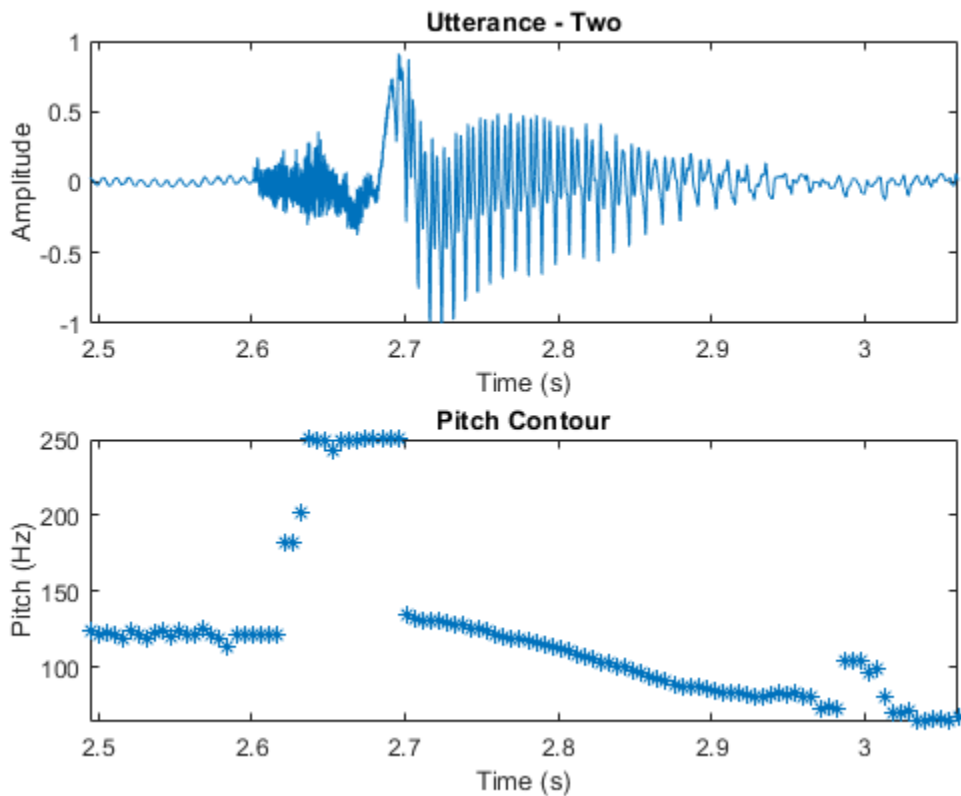
A speech signal is dynamic in nature and changes over time. It is assumed that speech signals are stationary on short time scales, and their processing is done in windows of 20-40 ms. This example uses a 30 ms window with a 25 ms overlap. Use the `pitch` function to see how pitch changes over time.

```
windowLength = round(0.03*fs);
overlapLength = round(0.025*fs);
```

```
f0 = pitch(audioIn,fs,'WindowLength',windowLength,'OverlapLength',overlapLength,'Range',[50,250])
```

```
figure
subplot(2,1,1)
plot(timeVector,audioIn)
axis([(110e3/fs) (135e3/fs) -1 1])
ylabel('Amplitude')
xlabel('Time (s)')
title('Utterance - Two')
```

```
subplot(2,1,2)
timeVectorPitch = linspace((twoStart/fs),(twoStop/fs),numel(f0));
plot(timeVectorPitch,f0,'*')
axis([(110e3/fs) (135e3/fs) min(f0) max(f0)])
ylabel('Pitch (Hz)')
xlabel('Time (s)')
title('Pitch Contour')
```



The `pitch` function estimates a pitch value for every frame. However, pitch is only characteristic of a source in regions of voiced speech. The simplest method to distinguish between silence and speech is to analyze the short term power. If the power in a frame is above a given threshold, you declare the frame as speech.

```
pwrThreshold = -20;
[segments,~] = buffer(audioIn>windowLength,overlapLength,'nodelay');
pwr = pow2db(var(segments));
isSpeech = (pwr > pwrThreshold);
```

The simplest method to distinguish between voiced and unvoiced speech is to analyze the zero crossing rate. A large number of zero crossings implies that there is no dominant low-frequency oscillation. If the zero crossing rate for a frame is below a given threshold, you declare it as voiced.

```
zcrThreshold = 300;
zeroLoc = (audioIn==0);
crossedZero = logical([0;diff(sign(audioIn))]);
crossedZero(zeroLoc) = false;
[crossedZeroBuffered,~] = buffer(crossedZero>windowLength,overlapLength,'nodelay');
zcr = (sum(crossedZeroBuffered,1)*fs)/(2*windowLength);
isVoiced = (zcr < zcrThreshold);
```

Combine `isSpeech` and `isVoiced` to determine whether a frame contains voiced speech.

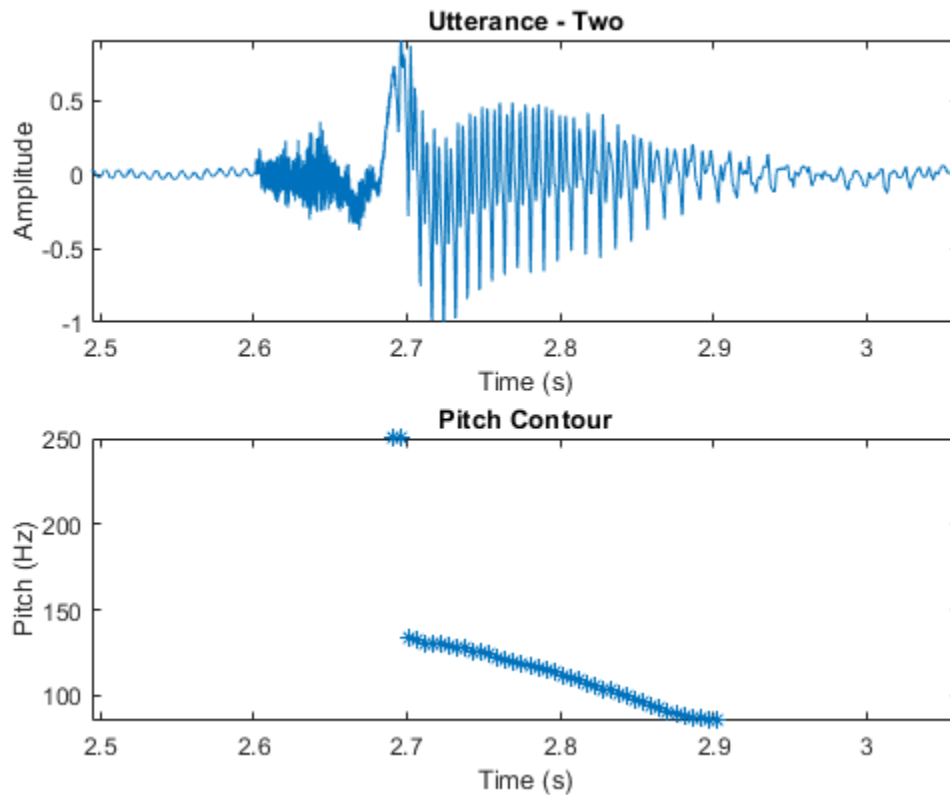
```
voicedSpeech = isSpeech & isVoiced;
```

Remove regions that do not correspond to voiced speech from the pitch estimate and plot.

```
f0(~voicedSpeech) = NaN;

figure
subplot(2,1,1)
plot(timeVector, audioIn)
axis([(110e3/fs) (135e3/fs) -1 1])
axis tight
ylabel('Amplitude')
xlabel('Time (s)')
title('Utterance - Two')

subplot(2,1,2)
plot(timeVectorPitch, f0, '*')
axis([(110e3/fs) (135e3/fs) min(f0) max(f0)])
ylabel('Pitch (Hz)')
xlabel('Time (s)')
title('Pitch Contour')
```

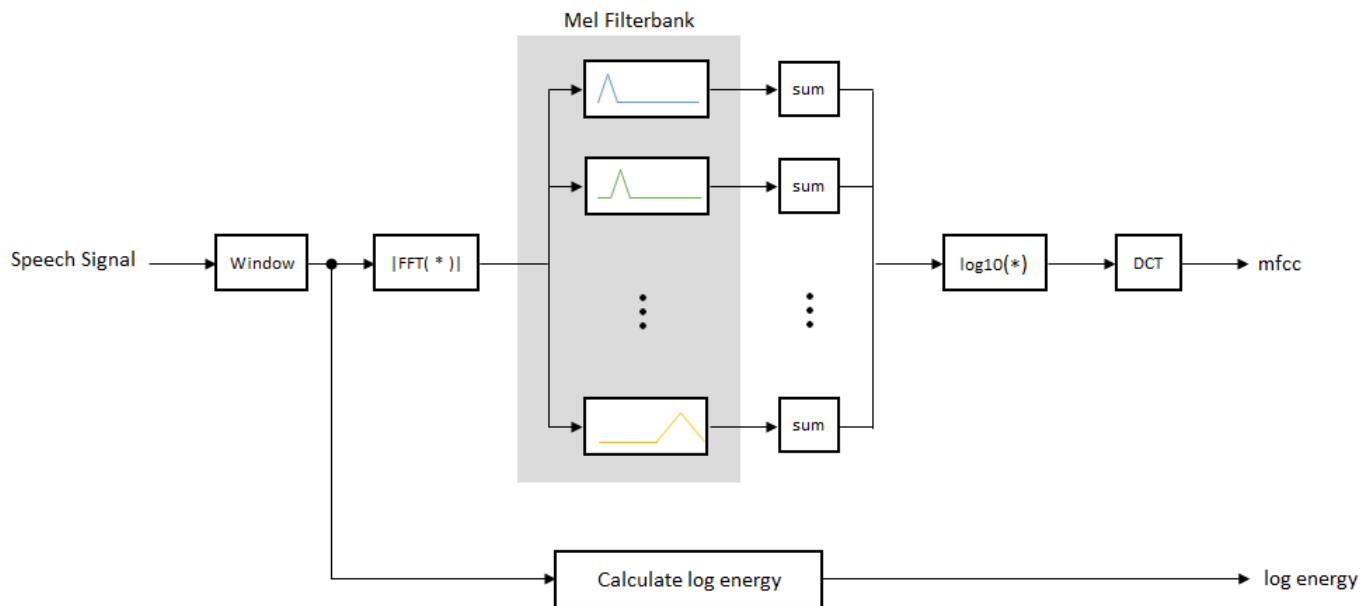


Mel-Frequency Cepstrum Coefficients (MFCC)

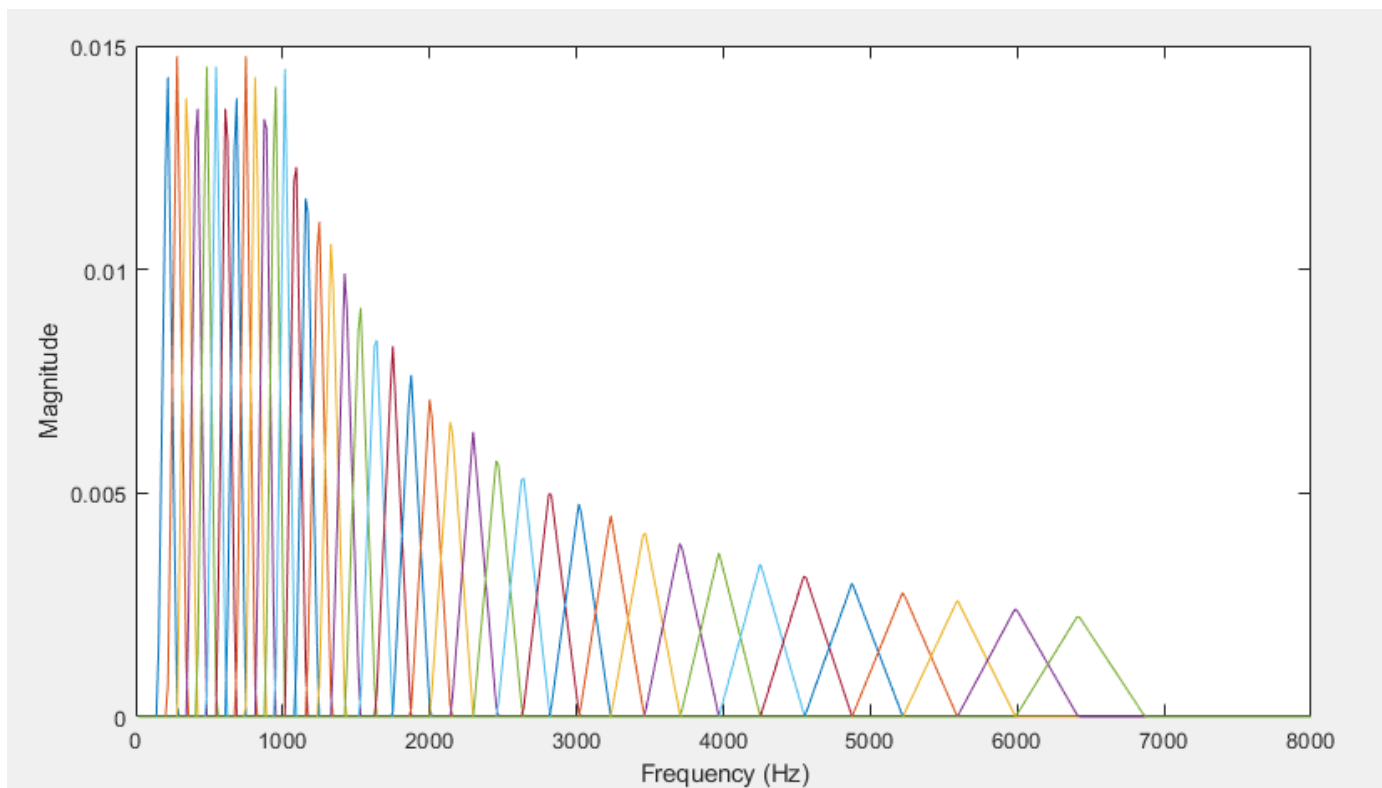
MFCC are popular features extracted from speech signals for use in recognition tasks. In the source-filter model of speech, MFCC are understood to represent the filter (vocal tract). The frequency response of the vocal tract is relatively smooth, whereas the source of voiced speech can be modeled as an impulse train. The result is that the vocal tract can be estimated by the spectral envelope of a speech segment.

The motivating idea of MFCC 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 MFCC, the basic steps are outlined by the diagram.



The mel filterbank linearly spaces the first 10 triangular filters and logarithmically spaces the remaining filters. The individual bands are weighted for even energy. The graph represents a typical mel filterbank.



This example uses `mfcc` to calculate the MFCC for every file.

Data Set

This example uses the Census Database (also known as AN4 Database) from the CMU Robust Speech Recognition Group [1 on page 1-0]. The data set contains recordings of male and female subjects speaking words and numbers. The helper function in this section downloads it for you and converts the raw files to FLAC. The speech files are partitioned into subdirectories based on the labels corresponding to the speakers. If you are unable to download it, you can load a table of features from `HelperAN4TrainingFeatures.mat` and proceed directly to the **Training a Classifier** section. The features have been extracted from the same data set.

Download and extract the speech files for 10 speakers (5 female and 5 male) into a temporary directory using the `HelperAN4Download` function.

```
dataDir = HelperAN4Download;
```

Create an `audioDatastore` object to manage this database for training. The datastore allows you to collect necessary files of a file format and read them.

```
ads = audioDatastore(dataDir,'IncludeSubfolders',true, ...  
    'FileExtensions','.flac', ...  
    'LabelSource','foldernames')
```

```
ads =  
    audioDatastore with properties:
```

```
        Files: {  
            '...\bhemmat\AppData\Local\Temp\an4\wav\flacData\fejs\an36-fejs-b  
            '...\bhemmat\AppData\Local\Temp\an4\wav\flacData\fejs\an37-fejs-b  
            '...\bhemmat\AppData\Local\Temp\an4\wav\flacData\fejs\an38-fejs-b  
            ... and 122 more  
        }  
    Folders: {  
        'C:\Users\bhemmat\AppData\Local\Temp\an4\wav\flacData'  
    }  
    Labels: [fejs; fejs; fejs ... and 122 more categorical]  
AlternateFileSystemRoots: {}  
    OutputDataType: 'double'  
SupportedOutputFormats: ["wav"      "flac"      "ogg"      "mp4"      "m4a"]  
    DefaultOutputFormat: "wav"
```

The `splitEachLabel` function of `audioDatastore` splits the datastore into two or more datastores. The resulting datastores have the specified proportion of the audio files from each label. In this example, the datastore is split into two parts. 80% of the data for each label is used for training, and the remaining 20% is used for testing. The `countEachLabel` method of `audioDatastore` is used to count the number of audio files per label. In this example, the label identifies the speaker.

```
[adsTrain, adsTest] = splitEachLabel(ads,0.8);
```

Display the datastore and the number of speakers in the train datastore.

```
adsTrain
```

```
adsTrain =  
    audioDatastore with properties:
```

```
        Files: {
```

```

        ' ...\bhemmat\AppData\Local\Temp\an4\wav\flacData\fejs\an36-fejs-b
        ' ...\bhemmat\AppData\Local\Temp\an4\wav\flacData\fejs\an37-fejs-b
        ' ...\bhemmat\AppData\Local\Temp\an4\wav\flacData\fejs\an38-fejs-b
        ... and 94 more
    }
    Folders: {
        'C:\Users\bhemmat\AppData\Local\Temp\an4\wav\flacData'
    }
    Labels: [fejs; fejs; fejs ... and 94 more categorical]
    AlternateFileSystemRoots: {}
    OutputDataType: 'double'
    SupportedOutputFormats: ["wav"      "flac"      "ogg"      "mp4"      "m4a"]
    DefaultOutputFormat: "wav"

```

```
trainDatastoreCount = countEachLabel(adsTrain)
```

```
trainDatastoreCount=10×2 table
```

Label	Count
fejs	10
fmjd	10
fsrb	10
ftmj	10
fwxs	10
mcen	10
mrcb	10
msjm	10
msjr	10
msmn	7

Display the datastore and the number of speakers in the test datastore.

```
adsTest
```

```
adsTest =
```

```
audioDatastore with properties:
```

```

        Files: {
        ' ...\bhemmat\AppData\Local\Temp\an4\wav\flacData\fejs\cen6-fejs-b
        ' ...\bhemmat\AppData\Local\Temp\an4\wav\flacData\fejs\cen7-fejs-b
        ' ...\bhemmat\AppData\Local\Temp\an4\wav\flacData\fejs\cen8-fejs-b
        ... and 25 more
        }
    Folders: {
        'C:\Users\bhemmat\AppData\Local\Temp\an4\wav\flacData'
    }
    Labels: [fejs; fejs; fejs ... and 25 more categorical]
    AlternateFileSystemRoots: {}
    OutputDataType: 'double'
    SupportedOutputFormats: ["wav"      "flac"      "ogg"      "mp4"      "m4a"]
    DefaultOutputFormat: "wav"

```

```
testDatastoreCount = countEachLabel(adsTest)
```

```
testDatastoreCount=10×2 table
```

Label	Count
-------	-------

fejs	3
fmjd	3
fsrb	3
ftmj	3
fwxs	2
mcen	3
mrcb	3
msjm	3
msjr	3
msmn	2

To preview the content of your datastore, read a sample file and play it using your default audio device.

```
[sampleTrain, dsInfo] = read(adsTrain);  
sound(sampleTrain,dsInfo.SampleRate)
```

Reading from the train datastore pushes the read pointer so that you can iterate through the database. Reset the train datastore to return the read pointer to the start for the following feature extraction.

```
reset(adsTrain)
```

Feature Extraction

Extract pitch and MFCC features from each frame that corresponds to voiced speech in the training datastore. The supporting function, `isVoicedSpeech` on page 1-0 , performs the voicing detection outlined in the description of pitch feature extraction on page 1-0 .

```
fs = dsInfo.SampleRate;  
windowLength = round(0.03*fs);  
overlapLength = round(0.025*fs);  
  
features = [];  
labels = [];  
while hasdata(adsTrain)  
    [audioIn,dsInfo] = read(adsTrain);  
  
    melC = mfcc(audioIn,fs,'Window',hamming(windowLength,'periodic'),'OverlapLength',overlapLength);  
    f0 = pitch(audioIn,fs,'WindowLength',windowLength,'OverlapLength',overlapLength);  
    feat = [melC,f0];  
  
    voicedSpeech = isVoicedSpeech(audioIn,fs>windowLength,overlapLength);  
  
    feat(~voicedSpeech,:) = [];  
    label = repelem(dsInfo.Label,size(feat,1));  
  
    features = [features;feat];  
    labels = [labels,label];  
end
```

Pitch and MFCC are not on the same scale. This will bias the classifier. Normalize the features by subtracting the mean and dividing the standard deviation.

```
M = mean(features,1);
S = std(features,[],1);
features = (features-M)./S;
```

Training a Classifier

Now that you have collected features for all 10 speakers, you can train a classifier based on them. In this example, you use a K-nearest neighbor (KNN) classifier. KNN is a classification technique naturally suited for multiclass classification. The hyperparameters for the nearest neighbor classifier include the number of nearest neighbors, the distance metric used to compute distance to the neighbors, and the weight of the distance metric. The hyperparameters are selected to optimize validation accuracy and performance on the test set. In this example, the number of neighbors is set to 5 and the metric for distance chosen is squared-inverse weighted Euclidean distance. For more information about the classifier, refer to `fitcknn` (Statistics and Machine Learning Toolbox).

Train the classifier and print the cross-validation accuracy. `crossval` (Statistics and Machine Learning Toolbox) and `kfoldLoss` (Statistics and Machine Learning Toolbox) are used to compute the cross-validation accuracy for the KNN classifier.

Specify all the classifier options and train the classifier.

```
trainedClassifier = fitcknn( ...
    features, ...
    labels, ...
    'Distance','euclidean', ...
    'NumNeighbors',5, ...
    'DistanceWeight','squaredinverse', ...
    'Standardize',false, ...
    'ClassNames',unique(labels));
```

Perform cross-validation.

```
k = 5;
group = labels;
c = cvpartition(group,'Kfold',k); % 5-fold stratified cross validation
partitionedModel = crossval(trainedClassifier,'CVPartition',c);
```

Compute the validation accuracy.

```
validationAccuracy = 1 - kfoldLoss(partitionedModel,'LossFun','ClassifError');
fprintf('\nValidation accuracy = %.2f%%\n', validationAccuracy*100);
```

Validation accuracy = 97.50%

Visualize the confusion chart.

```
validationPredictions = kfoldPredict(partitionedModel);
figure
cm = confusionchart(labels,validationPredictions,'title','Validation Accuracy');
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
```

Validation Accuracy													
True Class	fejs	1984	13	13	7	2		1		1	1	98.1%	1.9%
	fmjd	8	3033	16	30	15	1		1	1	1	97.6%	2.4%
	fsrb	13	17	2754	9	5	4		3			98.2%	1.8%
	ftmj	11	30	23	2610	9	2	3	3	2	6	96.7%	3.3%
	fwxs	13	35	8	12	2770			9	1	5	97.1%	2.9%
	mcen	2	3	1	1	1	1863	7	5	1	6	98.6%	1.4%
	mrcb	3	1	5	7	2	24	1861	2	8	4	97.1%	2.9%
	msjm	3	5	3	7	12	12	4	1893		10	97.1%	2.9%
	msjr	2		4	1		4	12	1	1047	2	97.6%	2.4%
	msmn	5	4	2	5	14	18	3	8	1	2028	97.1%	2.9%


```

numVectorsPerFile = [numVectorsPerFile,numVec];
features = [features;feat];
labels = [labels,label];
end
features = (features-M)./S;

```

Predict the label (speaker) for each frame by calling predict on trainedClassifier.

```

prediction = predict(trainedClassifier,features);
prediction = categorical(string(prediction));

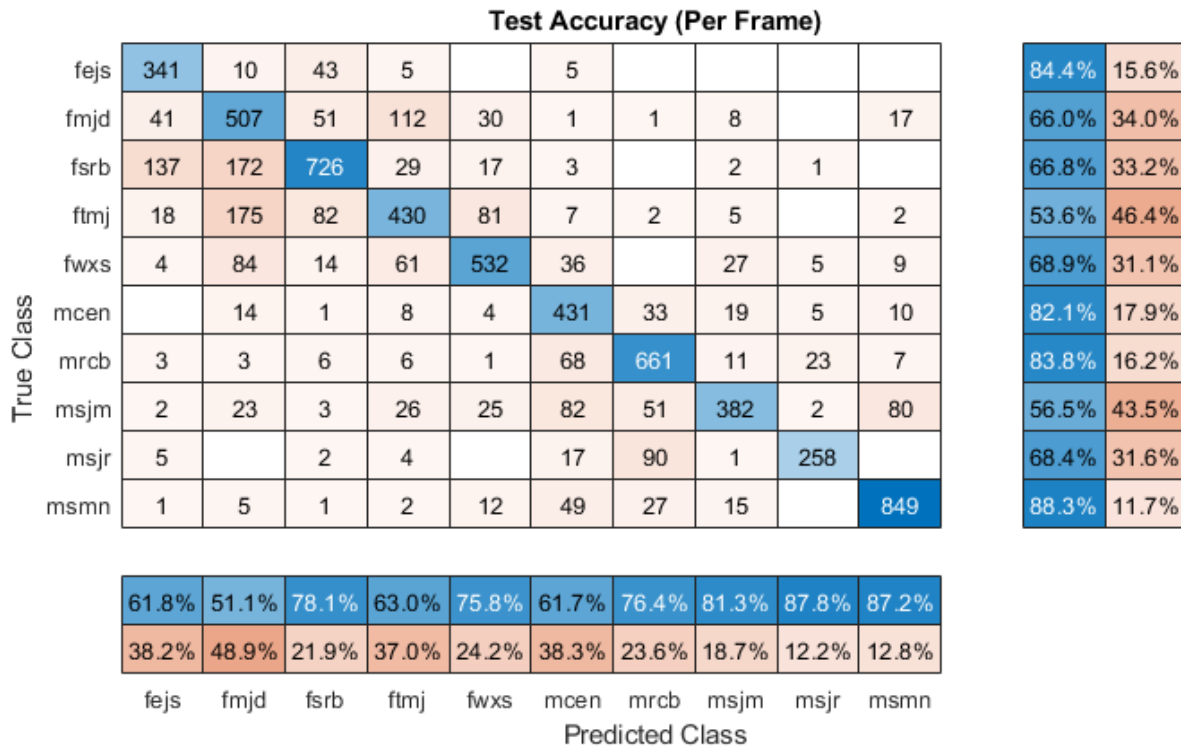
```

Visualize the confusion chart.

```

figure('Units','normalized','Position',[0.4 0.4 0.4 0.4])
cm = confusionchart(labels,prediction,'title','Test Accuracy (Per Frame)');
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';

```



For a given file, predictions are made for every frame. Determine the mode of predictions for each file and then plot the confusion chart.

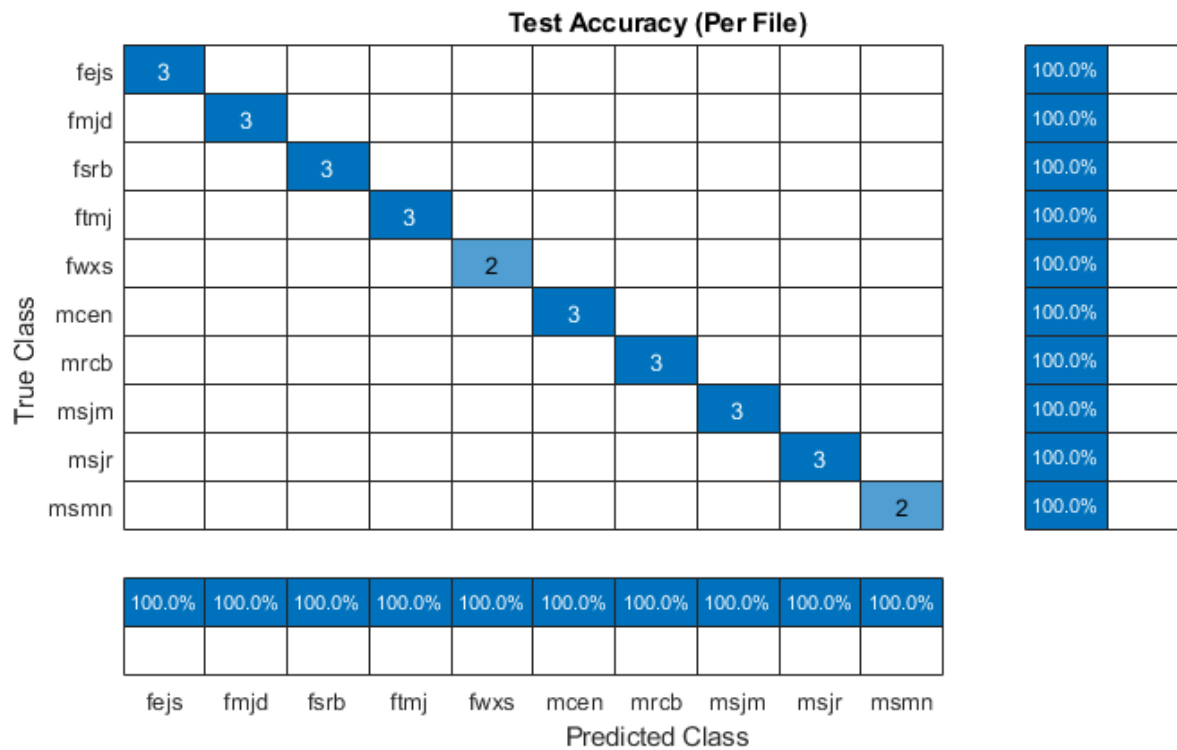
```

r2 = prediction(1:numel(adsTest.Files));
idx = 1;
for ii = 1:numel(adsTest.Files)
    r2(ii) = mode(prediction(idx:idx+numVectorsPerFile(ii)-1));
    idx = idx + numVectorsPerFile(ii);
end

figure('Units','normalized','Position',[0.4 0.4 0.4 0.4])
cm = confusionchart(adsTest.Labels,r2,'title','Test Accuracy (Per File)');

```

```
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
```



The predicted speakers match the expected speakers for all files under test.

The experiment was repeated using an internally developed dataset. The dataset consists of 20 speakers with each speaker speaking multiple sentences from the Harvard sentence list [2 on page 1-0]. For 20 speakers, the validation accuracy 89%.

Supporting Functions

```
function voicedSpeech = isVoicedSpeech(x,fs,windowLength,overlapLength)

pwrThreshold = -40;
[segments,~] = buffer(x,windowLength,overlapLength,'nodelay');
pwr = pow2db(var(segments));
isSpeech = (pwr > pwrThreshold);

zcrThreshold = 1000;
zeroLoc = (x==0);
crossedZero = logical([0;diff(sign(x))]);
crossedZero(zeroLoc) = false;
[crossedZeroBuffered,~] = buffer(crossedZero,windowLength,overlapLength,'nodelay');
zcr = (sum(crossedZeroBuffered,1)*fs)/(2*windowLength);
isVoiced = (zcr < zcrThreshold);

voicedSpeech = isSpeech & isVoiced;

end
```

References

[1] "CMU Sphinx Group - Audio Databases." Accessed December 19, 2019. <http://www.speech.cs.cmu.edu/databases/an4/>.

[2] "Harvard Sentences." *Wikipedia*, 27 Aug. 2019. *Wikipedia*, https://en.wikipedia.org/w/index.php?title=Harvard_sentences&oldid=912785385.

Measure Audio Latency

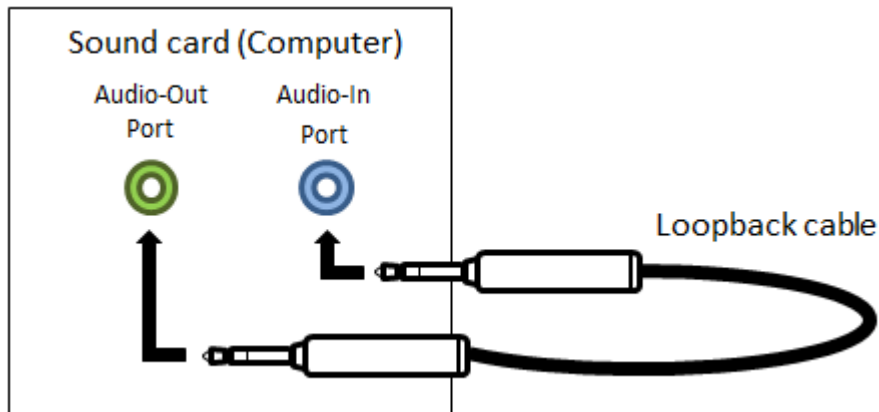
This example shows how to measure the latency of an audio device. The example uses `audioLatencyMeasurementExampleApp` which in turn uses `audioPlayerRecorder` along with a test signal and cross correlation to determine latency. To avoid disk access interference, the test signal is loaded into a `dsp.AsyncBuffer` object first, and frames are streamed from that object through the audio device.

Introduction

In general terms, **latency** is defined as the time from when the audio signal enters a system until it exits. In a digital audio processing chain, there are multiple parameters that cause latency:

- 1 Hardware (including A/D and D/A conversion)
- 2 Audio drivers that communicate with the system's sound card
- 3 Sampling rate
- 4 Samples per frame (buffer size)
- 5 Algorithmic latency (e.g. delay introduced by a filter or audio effect)

This example shows how to measure round trip latency. That is, the latency incurred when playing audio through a device, looping back the audio with a physical loopback cable, and recording the loopback audio with the same audio device. In order to compute latency for your own audio device, you need to connect the audio out and audio in ports using a loopback cable.



Roundtrip latency does not break down the measurement between output latency and input latency. It measures only the combined effect of the two. Also, most practical applications will not use a loopback setup. Typically the processing chain consists of recording audio, processing it, and playing the processed audio. However, the latency involved should be the same either way provided the other factors (frame size, sampling rate, algorithm latency) don't change.

Hardware Latency

Smaller frame sizes and higher sampling rates reduce the roundtrip latency. However, the tradeoff is a higher chance of dropouts occurring (overruns/underruns).

In addition to potentially increasing latency, the amount of processing involved in the audio algorithm can also cause dropouts.

Measuring Latency with audioLatencyMeasurementExampleApp.m

The function `audioLatencyMeasurementExampleApp` computes roundtrip latency in milliseconds for a given setup. Overruns and underruns are also presented. If the overruns/underruns are not zero, the results are likely invalid. For example:

```
audioLatencyMeasurementExampleApp('SamplesPerFrame',64,'SampleRate',48e3)

% The measurements in this example were done on macOS. For most
% measurements, a Steinberg UR22 external USB device was used. For the
% measurements with custom I/O channels, an RME Fireface UFX+ device was
% used. This RME device has lower latency than the Steinberg device for a
% given sample rate/frame size combination. Measurements on Windows using
% ASIO drivers should result in similar values.
```

Trial(s) done for frameSize 64.

```
ans =
    1x5 table
      SamplesPerFrame      SampleRate_kHz      Latency_ms      Overruns      Underruns
    _____      _____      _____      _____      _____
           64              48              8.3125              0              0
```

Some Tips When Measuring Latency

Real-time processing on a general purpose operating system is only possible if you minimize other tasks being performed by the computer. It is recommended to:

- 1 Close all other programs
- 2 Ensure no underruns/overruns occur
- 3 Use a large enough buffer size (`SamplesPerFrame`) to ensure consistent dropout-free behavior
- 4 Ensure your hardware settings (buffer size, sampling rate) match the inputs to `measureLatency`

On Windows, you can use the `asiosettings` function to launch the dialog to control the hardware settings. On macOS, you should launch the Audio MIDI Setup.

When using ASIO (or CoreAudio with Mac OS), the latency measurements are consistent as long as no dropouts occur. For small buffer sizes, it is possible to get a clean measurement in one instance and dropouts the next. The `Ntrials` option can be used to ensure consistent dropout behavior when measuring latency. For example, to perform 3 measurements, use:

```
audioLatencyMeasurementExampleApp('SamplesPerFrame',96,...
    'SampleRate',48e3,'Ntrials',3)

Trial(s) done for frameSize 96.
ans =
    3x5 table
      SamplesPerFrame      SampleRate_kHz      Latency_ms      Overruns      Underruns
    _____      _____      _____      _____      _____
           96              48              10.312              0              0
           96              48              10.312              0              0
           96              48              10.312              0              0
```

Measurements For Different Buffer Sizes

On macOS, it is also possible to try different frame sizes without changing the hardware settings. To make this convenient, you can specify a vector of `SamplesPerFrame`:

```

BufferSizes = [64;96;128];
t = audioLatencyMeasurementExampleApp('SamplesPerFrame',BufferSizes)

% Notice that for every sample increment in the buffer size, the additional
% latency is 3*SamplesPerFrameIncrement/SampleRate (macOS only).

Trial(s) done for frameSize 64.
Trial(s) done for frameSize 96.
Trial(s) done for frameSize 128.
t =
    3x5 table
      SamplesPerFrame   SampleRate_kHz   Latency_ms   Overruns   Underruns
    _____   _____   _____   _____   _____
         64             48         8.3125         0         0
         96             48        10.312         0         0
        128             48        12.312         0         0

```

Specifically, in the previous example, the increment is

```
3*[128-96, 96-64]/48e3
```

```

% In addition, notice that the actual buffering latency is also determined
% by 3*SamplesPerFrame/SampleRate. Subtracting this value from the measured
% latency gives a measure of the latency introduced by the device (combined
% effect of A/D conversion, D/A conversion, and drivers). The numbers above
% indicate about 4.3125 ms latency due to device-specific factors.

```

```
t.Latency_ms - 3*BufferSizes/48
```

```

ans =
    0.0020    0.0020
ans =
    4.3125
    4.3125
    4.3125

```

Specifying Custom Input/Output Channels

The measurements performed so far assume that channel #1 is used for both input and output. If your device has a loopback cable connected to other channels, you can specify them using the `IOChannels` option to `measureLatency`. This is specified as a 2-element vector, corresponding to the input and output channels to be used (the measurement is always on a mono signal). For example for an RME Fireface UFX+:

```

audioLatencyMeasurementExampleApp('SamplesPerFrame',[32 64 96],...
    'SampleRate',96e3,'Device','Fireface UFX+ (23767940)',...
    'IOChannels',[1 3])

```

```

Trial(s) done for frameSize 32.
Trial(s) done for frameSize 64.
Trial(s) done for frameSize 96.
ans =
    3x5 table
      SamplesPerFrame   SampleRate_kHz   Latency_ms   Overruns   Underruns
    _____   _____   _____   _____   _____
         32             96         2.6458         0         32
         64             96         3.6458         0         0
         96             96         4.6458         0         0

```

Algorithmic Latency

The measurements so far have not included algorithm latency. Therefore, they represent the minimal roundtrip latency that can be achieved for a given device, buffer size, and sampling rate. You can add a linear phase FIR filter the processing chain to verify that the latency measurements are as expected. Moreover, it provides a way of verifying robustness of the real-time audio processing under a given workload. For example,

```
L = 961;
Fs = 48e3;
audioLatencyMeasurementExampleApp('SamplesPerFrame',128,...
    'SampleRate',Fs,'FilterLength',L,'Ntrials',3)

% The latency introduced by the filter is given by the filter's
% group-delay.

GroupDelay = (L-1)/2/Fs

% The group delay accounts for the 10 ms of additional latency when using a
% 961-tap linear-phase FIR filter vs. the minimal achievable latency.

Trial(s) done for frameSize 128.
ans =
    3x6 table
      SamplesPerFrame   SampleRate_kHz   FilterLength   Latency_ms   Overruns   Underruns
    _____
      128              48              961          22.312           0           0
      128              48              961          22.312           0           0
      128              48              961          22.312           0           0
GroupDelay =
    0.0100
```

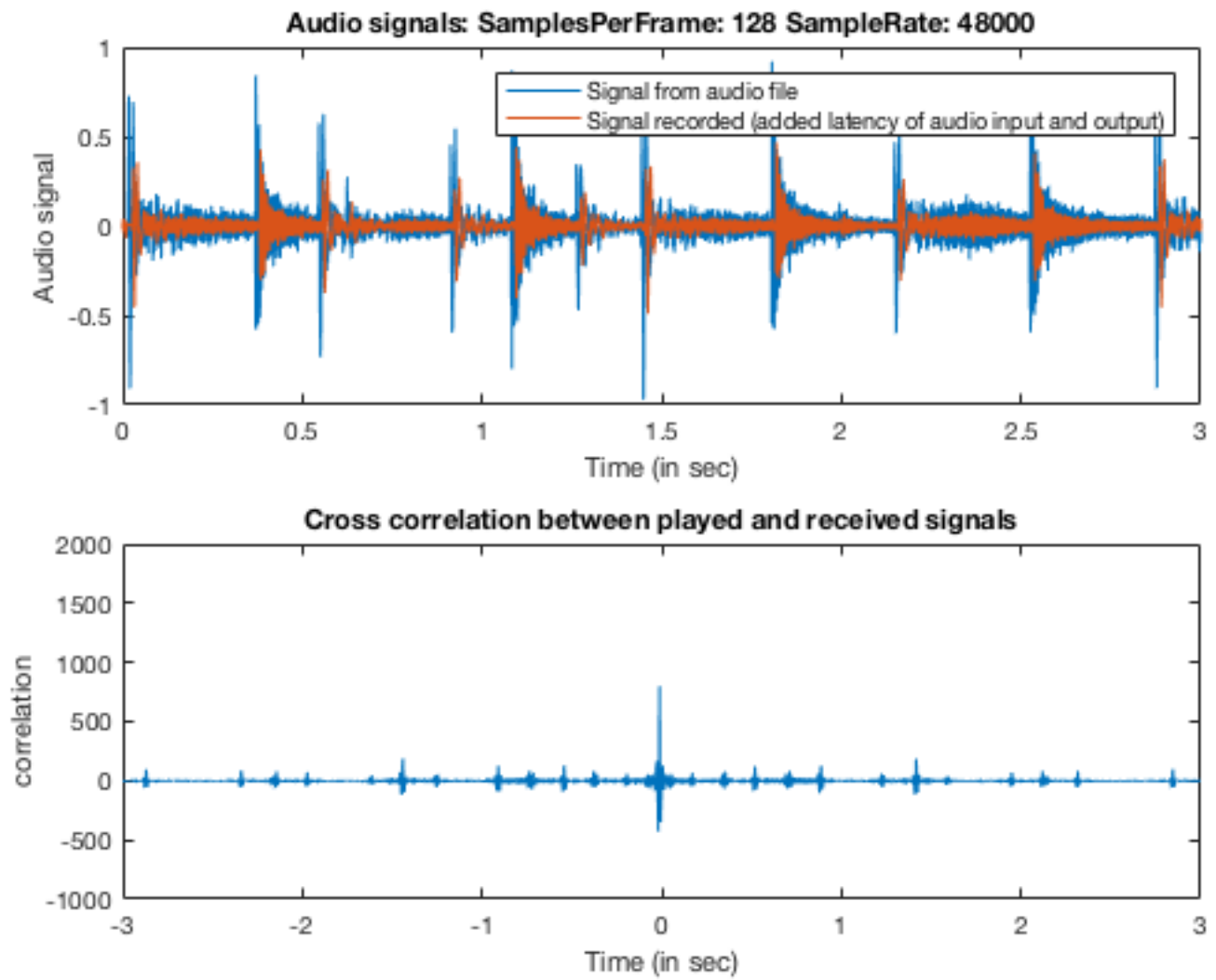
Plotting the Original and Recorded Signal

%The latency measurements are determined by cross-correlating a source
%audio signal with a delayed version of the signal that results after
%loopback through the audio device. You can use the Plot option in
%measureLatency to plot the original and delayed signal along with the
%cross correlation:

```
audioLatencyMeasurementExampleApp('SamplesPerFrame',128,'Plot',true)
```

% If the optional FIR filtering is used, the waveforms are not affected
% because the filter used has a broader bandwidth than the test audio
% signal.

```
Trial(s) done for frameSize 128.
Plotting...
ans =
    1x5 table
      SamplesPerFrame   SampleRate_kHz   Latency_ms   Overruns   Underruns
    _____
      128              48          12.312           0           0
```



Measure Performance of Streaming Real-Time Audio Algorithms

This example presents a utility that can be used to analyze the timing performance of signal processing algorithms designed for real-time streaming applications.

Introduction

The ability to prototype an audio signal processing algorithm in real time using MATLAB depends primarily on its execution performance. Performance is affected by a number of factors, such as the algorithm's complexity, the sampling frequency and the input frame size. Ultimately, the algorithm must be fast enough to ensure it can always execute within the available time budget and not drop any frames. Frames are dropped whenever the audio input queue is overrun with new samples (not read fast enough) or the audio output queue is underrun (not written fast enough). Dropped frames result in undesirable artifacts in the output audio signal.

This example presents a utility to profile the execution performance of an audio signal processing algorithm within MATLAB and compare it to the available time budget.

Results in this example were obtained on a machine running an Intel (R) Xeon (R) CPU with a clock speed of 3.50 GHz, and 64 GB of RAM. Results vary depending on system specifications.

Measure Performance of a Notch Filter Application

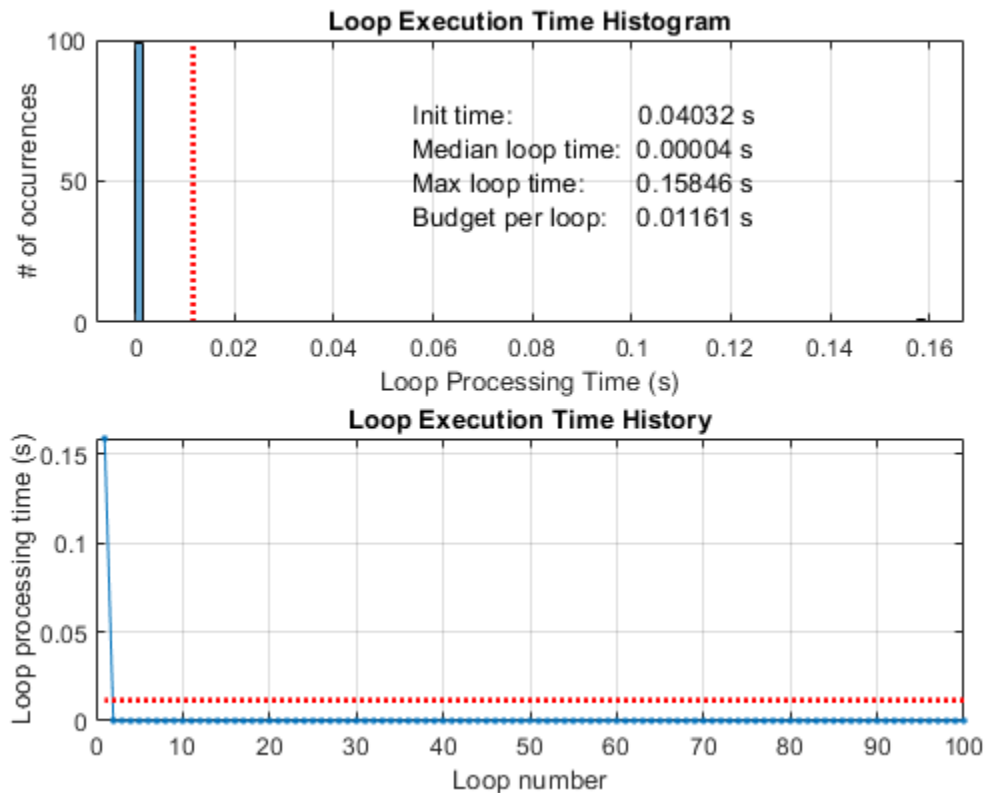
In this example, you measure performance of an eighth-order notch filter, implemented using `dsp.BiquadFilter`.

`helperAudioLoopTimerExample` defines and instantiates the variables used in the algorithm. The input is read from a file using a `dsp.AudioFileReader` object, and then streamed through the notch filter in a processing loop.

`audioexample.AudioLoopTimer` is the utility object used to profile execution performance and display a summary of the results. The utility uses simple `tic/toc` commands to log the timing of different stages of the simulation. The initialization time (which is the time it takes to instantiate and set up variables and objects before the simulation loop begins) is measured using the `ticInit` and `tocInit` methods. The individual simulation loop times are measured using the `ticLoop` and `tocLoop` methods. After the simulation loop is done, a performance report is generated using the object's `generateReport` method.

Execute `helperAudioLoopTimerExample` to run the simulation and view the performance report:

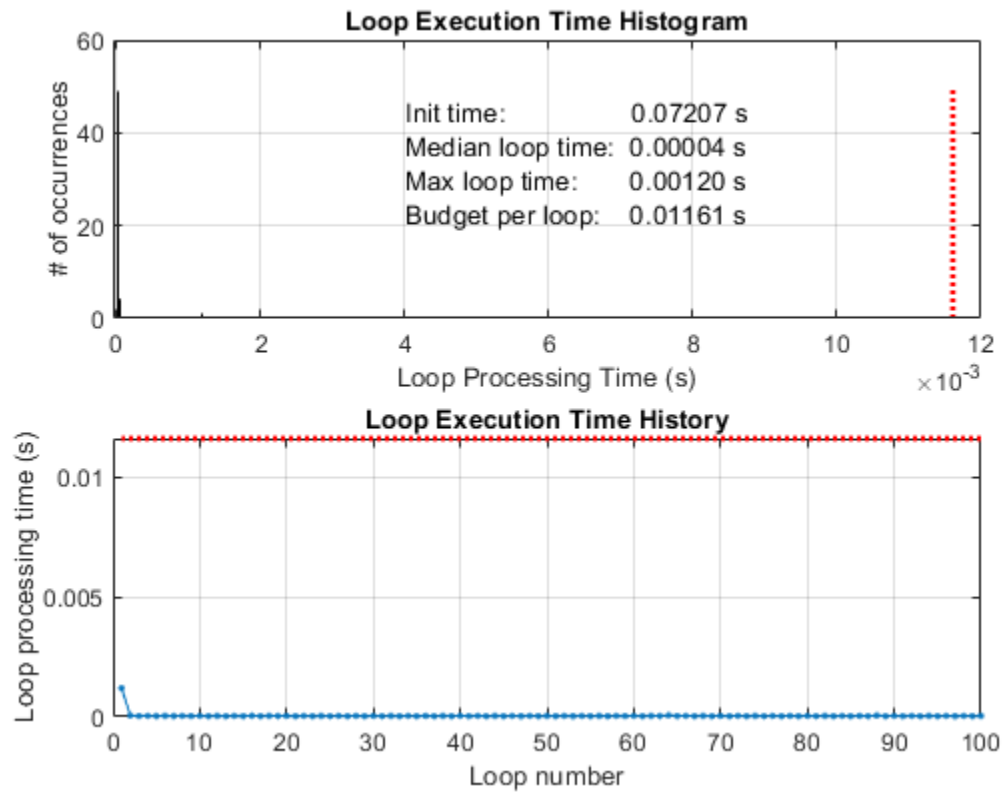
```
helperAudioLoopTimerExample;
```



The performance report figure displays a histogram of the loop execution times in the top plot. The red line represents the maximum allowed loop execution time, or budget, above which samples will be dropped. The budget per simulation loop is equal to L/F_s , where L is the input frame size, and F_s is the sampling rate. In this example, $L = 512$, $F_s = 44100$ Hz, and the budget per loop is around 11.6 milliseconds. The performance report also displays the runtime of the individual simulation loops in the bottom plot. Again, the red line represents the allowed budget per loop.

Notice that although the median loop time is well within the budget, the maximum loop time exceeds the budget. From the bottom plot, it is evident that the budget is exceeded on the very first loop pass, and that subsequent loop runs are within the budget. The relative slow performance of the first simulation loop is due to the penalty incurred the first time the `step` method is called on the `dsp.BiquadFilter` and `dsp.AudioFileReader` objects. The first call to `step` triggers the execution of one-time tasks that do not depend on the inputs to `step`, such as hardware resource allocation and state initialization. This problem can be alleviated by executing one-time tasks before the simulation loop. You can perform the one-time tasks by calling the `setup` method on the simulation objects in the initialization stage. Execute `helperAudioLoopTimerExample(true)` to re-run the simulation with pre-loop setup enabled.

```
helperAudioLoopTimerExample(true);
```



All loop runs are now within the budget. Notice that the maximum and total loop times have been drastically reduced compared to the first performance report, at the expense of a higher initialization time.

THD+N Measurement with Tone-Tracking

This example shows how to measure total harmonic distortion and noise level of audio input and output devices.

Introduction

Audio input and output devices are non-linear in nature. This causes harmonic distortion in the audio signal. Apart from the unwanted signals that may be harmonically related to the signal, these devices can also add uncorrelated noise to the audio signal.

Total Harmonic Distortion and Noise (THD+N) quantifies the sum of these two distortions. It is defined as the root mean square (RMS) level of all harmonics and noise components over a specified bandwidth. The signal level is also specified as a reference.

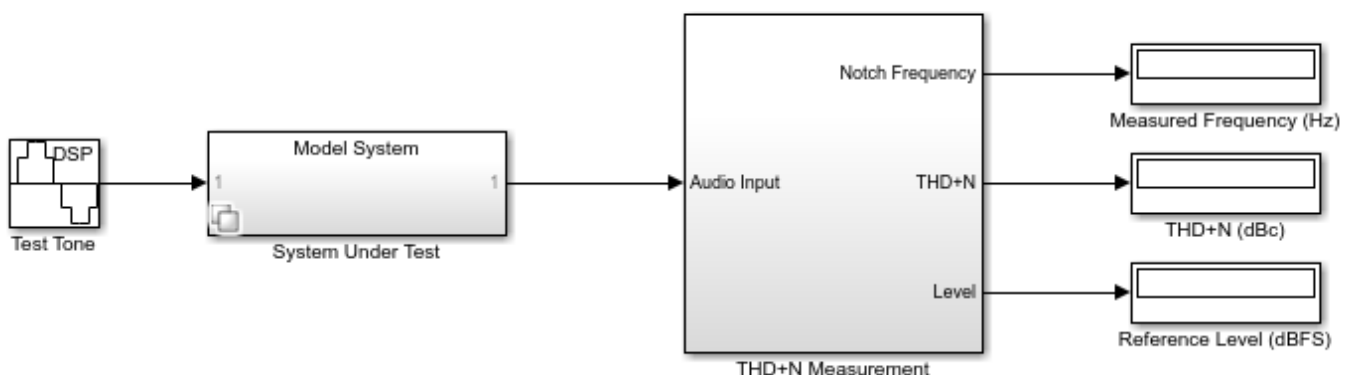
Measurement of THD+N

This example introduces a reference model that can be used for THD+N measurements of audio input and output devices. The steps involved in measurement are:

- 1 Generate a pure sine wave of a specific frequency.
- 2 Play the signal through an audio output device and record it through an audio input device.
- 3 From the recorded signal, identify the sine wave peak. This will give the reference signal RMS level.
- 4 Remove the identified sine wave from recorded signal. What remains is everything unwanted, and its RMS will give THD+N value.

This example follows the AES17-1998(r2004) [1] standard for THD+N measurement. The standard recommends a 997 Hz frequency sine wave. It also recommends a notch filter having Q between 1 and 5 for filtering out the sine wave from recorded signal. A Q value of 5 is used in this example.

Tone-Tracking THD+N Measurement System



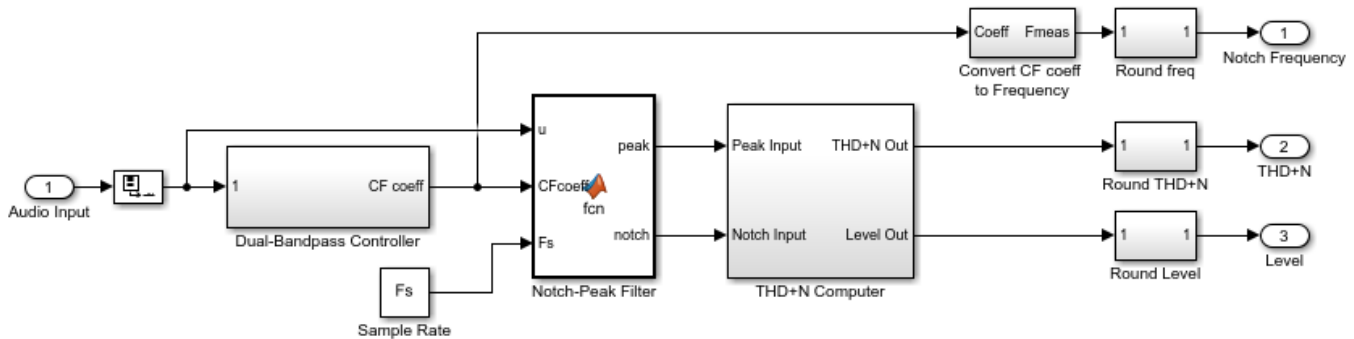
Copyright 2015-2017 The MathWorks, Inc.

Info

The `audioTHDNmeasurementexample` model implements a reference system for measuring THD+N. Following the AES17-1998(r2004) standard, the sine wave source `Test Tone` generates a frequency

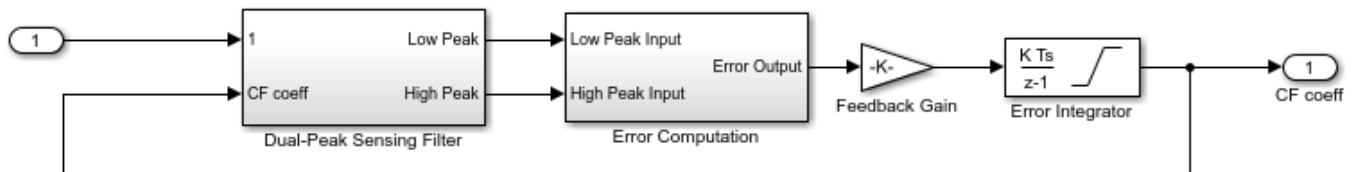
of 997 Hz. The subsystem **System Under Test** is a variant subsystem. By default, it selects a non-linear model implemented in Simulink for measuring the THD+N. To perform the measurement on your machine's audio input and output device, set the SUT variable in base workspace to THDNDemoSUT.AudioHardware.

The measurement is done by the THD+N Measurement subsystem.



Dual-Bandpass Controller

The measurement system in the model uses a dual-peak tracking filter to locate the notch at the test tone's fundamental. This accommodates signal generators that are not synchronized to the ADC clock. The output of this block is the center frequency coefficient of the notch filter that will be used to extract the test sine tone. The two peaking filters in the controller are implemented using `dsp.NotchPeakFilter` System objects. When the model is run, the feedback loop works to adjust the center frequencies of the two peaking filters in such a way that the output locks on to the peak tone of the input.

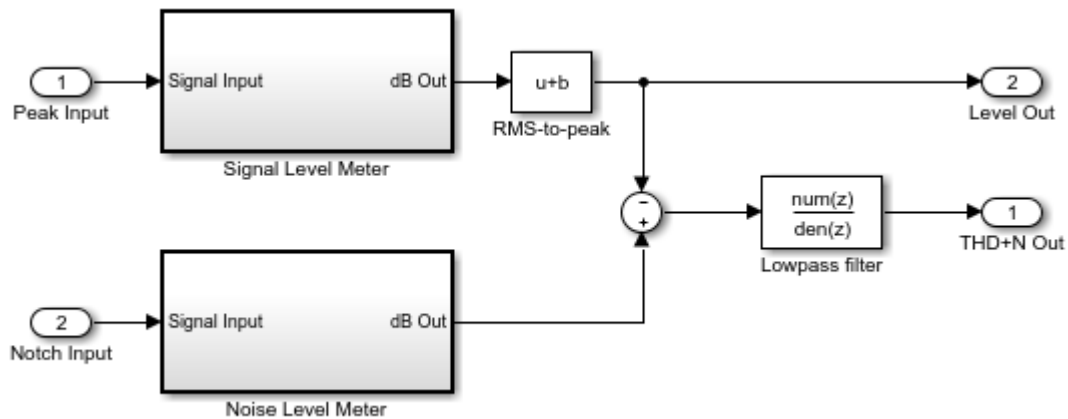


Notch-Peak Filter

Once the frequency of the sine wave has been identified, pass it to a peaking filter to extract the test tone signal. This will be used to determine the test signal's peak level. A notch filter will then use the same center frequency to remove the sine wave. The remaining signal is the sum of the total harmonic distortion and noise. Use a single `dsp.NotchPeakFilter` to get both - notch and peak outputs. The Q-factor of this filter is chosen as 5, conforming to AES17-1998 standard.

THD+N Computer

The THD+N Computer subsystem mimics a signal level meter. It takes the notch and peak outputs and smooths them using a lowpass filter. It then converts the level of the signals to dB.



You can run the model and see the displays update with measured sine wave frequency, THD+N level in dB, and reference signal level in dB.

References

[1] AES17-1998 "AES standard method for digital audio engineering - Measurement of digital audio equipment", Audio Engineering Society (1998), r2004.

Measure Impulse Response of an Audio System

The impulse response (IR) is an important tool for characterizing or representing a linear time-invariant (LTI) system. The Impulse Response Measurer enables you to measure and capture the impulse response of audio systems, including:

- Audio I/O hardware
- Rooms and halls
- Enclosed spaces like inside of a car or a recording studio

In this example, you use the Impulse Response Measurer to measure the impulse response of your room. You then use the acquired impulse response with `audiopluginexample.FastConvolver` to add reverberation to an audio signal.

This example requires that your machine has an audio device capable of full-duplex mode and an appropriate audio driver. To learn more about how the app records and plays audio data, see `audioPlayerRecorder`.

Description of IR Measurement Techniques

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].

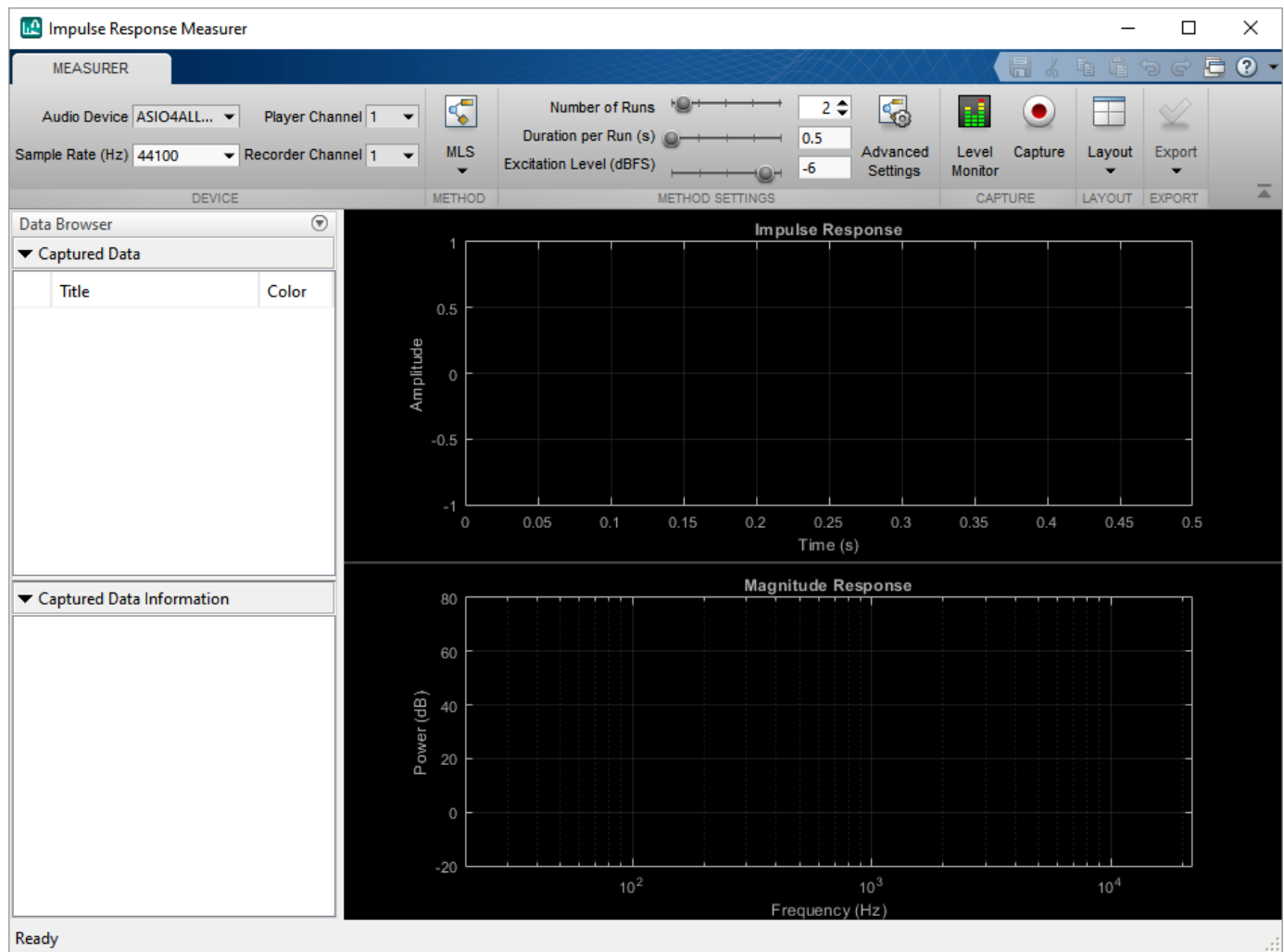
The Maximum-Length-Sequence (MLS) technique is based upon the excitation of the acoustical space by a periodic pseudo-random signal. The impulse response is obtained by circular cross-correlation between the measured output and the test tone (MLS sequence). For more details, see [2].

In this example, you use the MLS measurement technique.

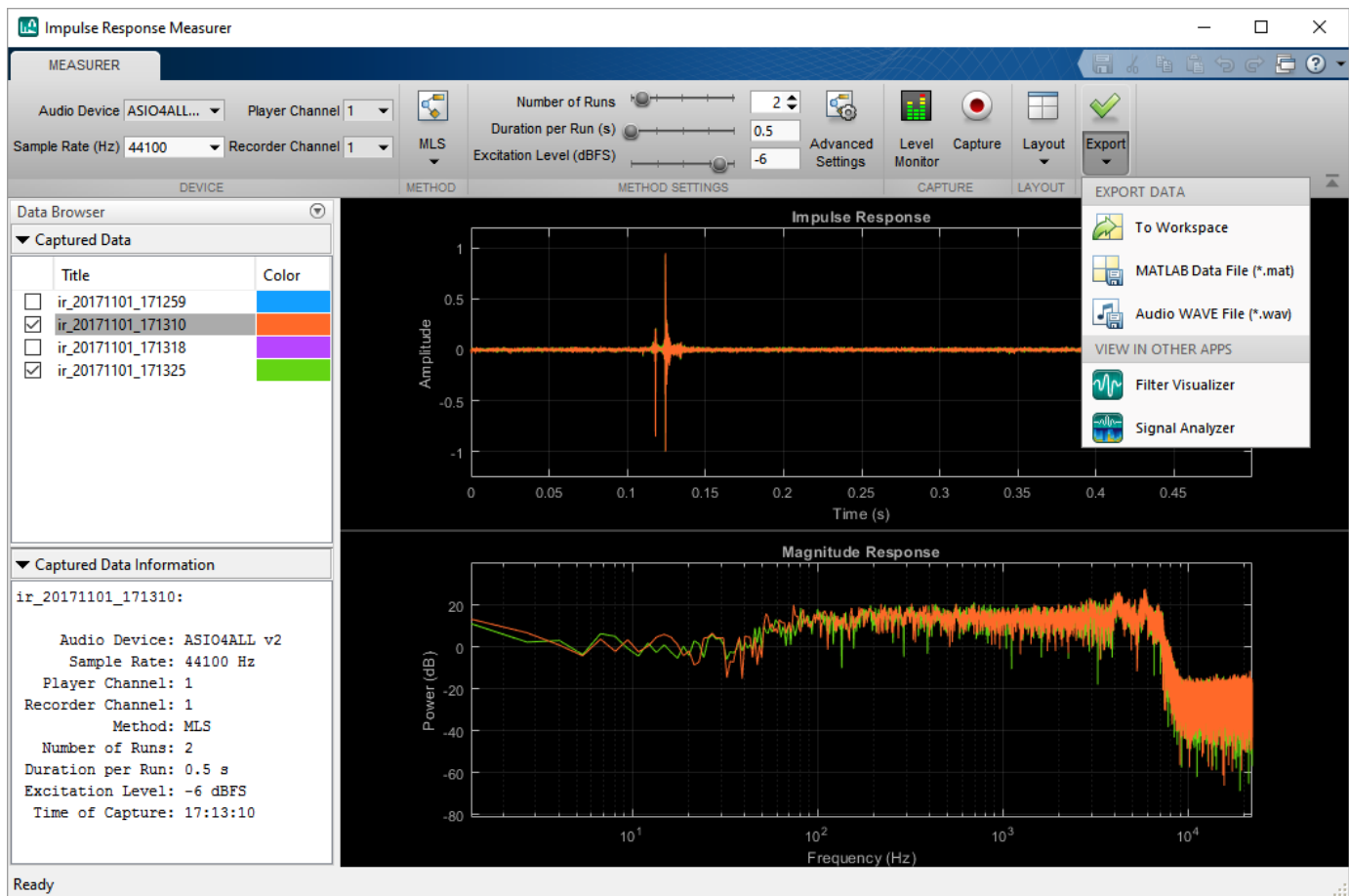
Acquire Impulse Response of Room

1. To open the app, at the MATLAB® command prompt, enter:

```
impulseResponseMeasurer
```



2. Use the default settings of the app and click **Capture**. Make sure the device name and the channel number match your system's configuration.
3. Once you capture the impulse response, click the **Export** button and select **To Workspace**.



Use Impulse Response to Add Reverb to an Audio Signal

Time-domain convolution of an input frame with a long impulse response adds latency equal to the length of the impulse response. The algorithm used by the `audiopluginexample.FastConvolver` plugin uses frequency-domain partitioned convolution to reduce the latency to twice the partition size [3]. `audiopluginexample.FastConvolver` is well-suited to impulse responses acquired using `impulseResponseMeasurer`.

1. To create an `audiopluginexample.FastConvolver` object, at the MATLAB® command prompt, enter:

```
fastConvolver = audiopluginexample.FastConvolver
```

```
fastConvolver =
```

```
audiopluginexample.FastConvolver with properties:
```

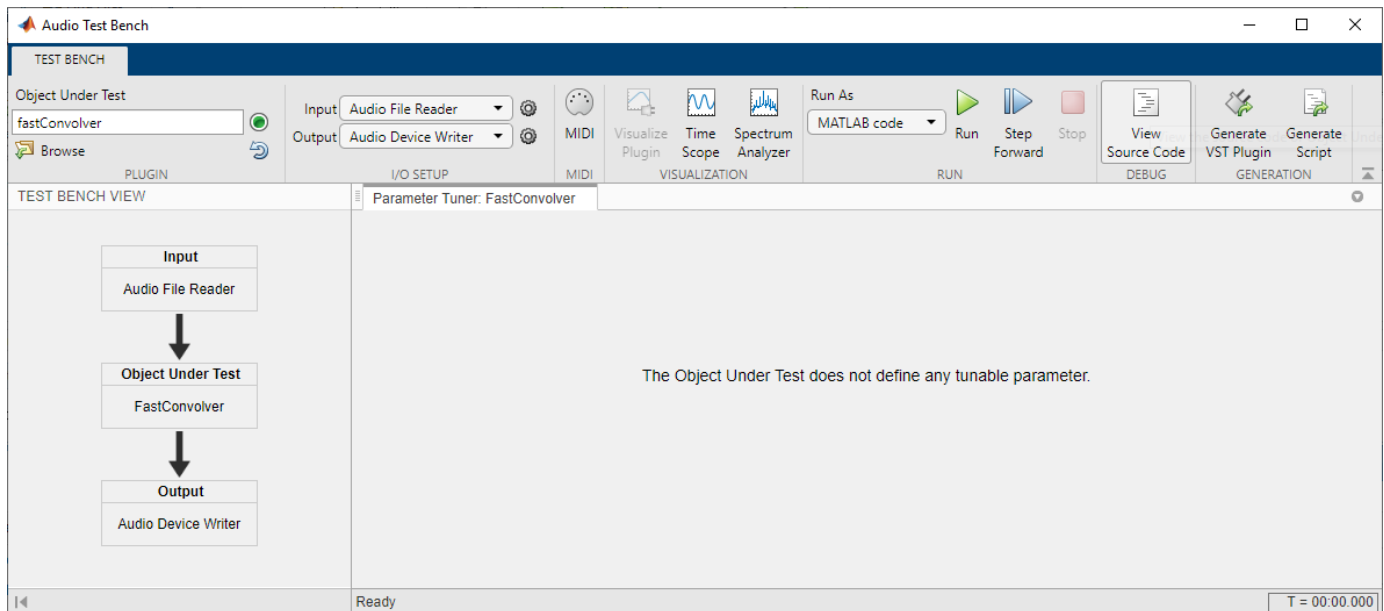
```
ImpulseResponse: [1x227497 double]
PartitionSize: 1024
```

2. Set the impulse response property to your acquired impulse response measurement. You can clear the impulse response for your workspace once it is saved to the fast convolver.

```
load measuredImpulseResponse
irEstimate = measuredImpulseResponse.ImpulseResponse.Amplitude(:,1);
fastConvolver.ImpulseResponse = irEstimate;
```

3. Open the audio test bench and specify your fast convolver object

```
audioTestBench(fastConvolver)
```



4. By default, the Audio Test Bench reads from an audio file and writes to your audio device. Click Run to listen to an audio file convolved with your acquired impulse response.

Tips and Tricks

The excitation level slider on the `impulseResponseMeasurer` applies gain to the output test tone. A higher output level is generally recommended to maximize signal-to-noise ratio (SNR). However, if the output level is too high, undesired distortion may occur.

Export to filter visualizer (FVTool) through the **Export** button to look at other useful plots like phase response, group delay, etc.

References

- [1] Farina, 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 Embrechts, and Dominique Archambeau. "Comparison of different impulse response measurement techniques." *Journal of Audio Engineering Society*. Vol. 50, Issue 4, pp. 249-262.
- [3] Armelloni, Enrico, Christian Giottoli, and Angelo Farina. "Implementation of real-time partitioned convolution on a DSP board." *Applications of Signal Processing to Audio and Acoustics, 2003 IEEE Workshop on.*, pp. 71-74. IEEE, 2003.

Measure Frequency Response of an Audio Device

The frequency response (FR) is an important tool for characterizing the fidelity of an audio device or component.

This example requires an audio device capable of recording and playing audio and an appropriate audio driver. To learn more about how the example records and plays audio data, see `audioDeviceReader` and `audioDeviceWriter`.

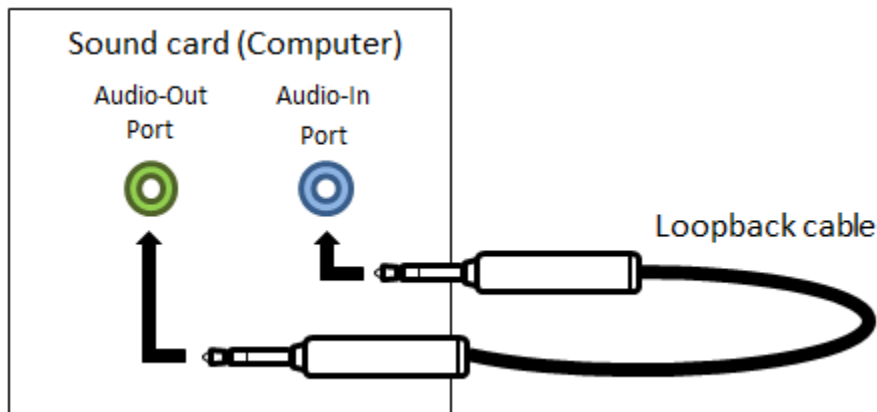
Description of FR Measurement Techniques

An FR measurement compares the output levels of an audio device to known input levels. A basic FR measurement consists of two or three test tones: mid, high, and low.

In this example you perform an audible range FR measurement by sweeping a sine wave from the lowest frequency in the range to the highest. A flat response indicates an audio device that responds equally to all frequencies.

Setup Experiment

In this example, you measure the FR by playing an audio signal through `audioDeviceWriter` and then recording the signal through `audioDeviceReader`. A loopback cable is used to physically connect the audio-out port of the sound card to its audio-in port.



Audio Device Reader and Writer

To start, use the `audioDeviceReader` System object™ and `audioDeviceWriter` System object to connect to the audio device. This example uses a Steinberg UR44 audio device with a 48 kHz sampling rate and a buffer size of 1024 samples.

```
sampleRate = 48e3;
device = 'Yamaha Steinberg USB ASIO';

aDR = audioDeviceReader( ...
    'SampleRate',sampleRate, ...
    'Device',device, ...
    'Driver','ASIO', ...
    'BitDepth','16-bit integer', ...
    'ChannelMappingSource','Property', ...
    'ChannelMapping',1);
```

```
aDW = audioDeviceWriter( ...  
    'SampleRate',sampleRate, ...  
    'Device',device, ...  
    'Driver','ASIO', ...  
    'BitDepth','16-bit integer', ...  
    'ChannelMappingSource','Property', ...  
    'ChannelMapping',1);
```

Test Signal

The test signal is a sine wave with 1024 samples per frame and an initial frequency of 0 Hz. The frequency is increased in 50 Hz increments to sweep the audible range.

```
samplesPerFrame = 1024;  
sineSource = audioOscillator( ...  
    'Frequency',0, ...  
    'SignalType','sine', ...  
    'SampleRate',sampleRate, ...  
    'SamplesPerFrame',samplesPerFrame);
```

Spectrum Analyzer

Use the `dsp.SpectrumAnalyzer` to visualize the FR of your audio I/O system. 20 averages of the spectrum estimate are used throughout the experiment and the resolution bandwidth is set to 50 Hz. The sampling frequency is set to 48 kHz.

```
RBW = 50;  
Navg = 20;  
  
scope = dsp.SpectrumAnalyzer( ...  
    'Method','Filter bank', ...  
    'SampleRate',sampleRate, ...  
    'RBWSource','Property','RBW',RBW, ...  
    'SpectralAverages',Navg, ...  
    'FrequencySpan','Start and stop frequencies',...  
    'StartFrequency',0, ...  
    'StopFrequency',sampleRate/2, ...  
    'ReducePlotRate',false, ...  
    'PlotAsTwoSidedSpectrum',false, ...  
    'FrequencyScale','Log', ...  
    'PlotMaxHoldTrace',true, ...  
    'ShowLegend',true, ...  
    'YLimits',[-110 20],...  
    'YLabel','Power', ...  
    'Title','Audio Device Frequency Response');
```

Frequency Response Measurement Loop

To avoid the impact of setup time on the FR measurement, prerun your audio loop for 5 seconds.

Once the actual FR measurement starts, sweep the test signal through the audible frequency range. Use the spectrum analyzer to visualize the FR.

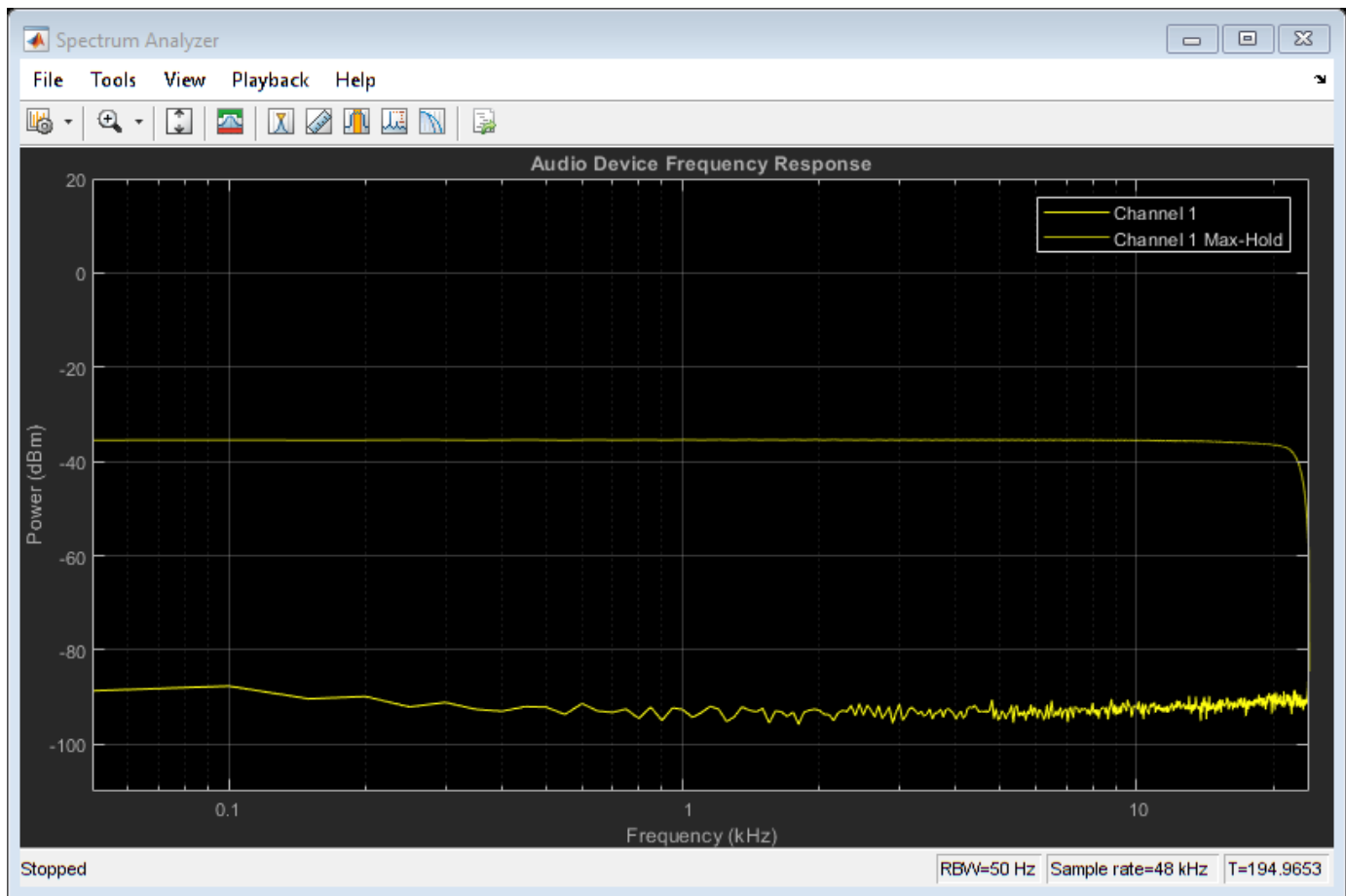
```
tic  
while toc < 5  
    x = sineSource();  
    aDW(x);
```

```
        y = aDR();
        scope(y);
    end

    count = 1;
    readerDrops = 0;
    writerDrops = 0;

    while true
        if count == Navg
            newFreq = sineSource.Frequency + RBW;
            if newFreq > sampleRate/2
                break
            end
            sineSource.Frequency = newFreq;
            count = 1;
        end
        x = sineSource();
        writerUnderruns = aDW(x);
        [y, readerOverruns] = aDR();
        readerDrops = readerDrops + readerOverruns;
        writerDrops = writerDrops + writerUnderruns;
        scope(y);
        count = count + 1;
    end

    release(aDR)
    release(aDW)
    release(scope)
```

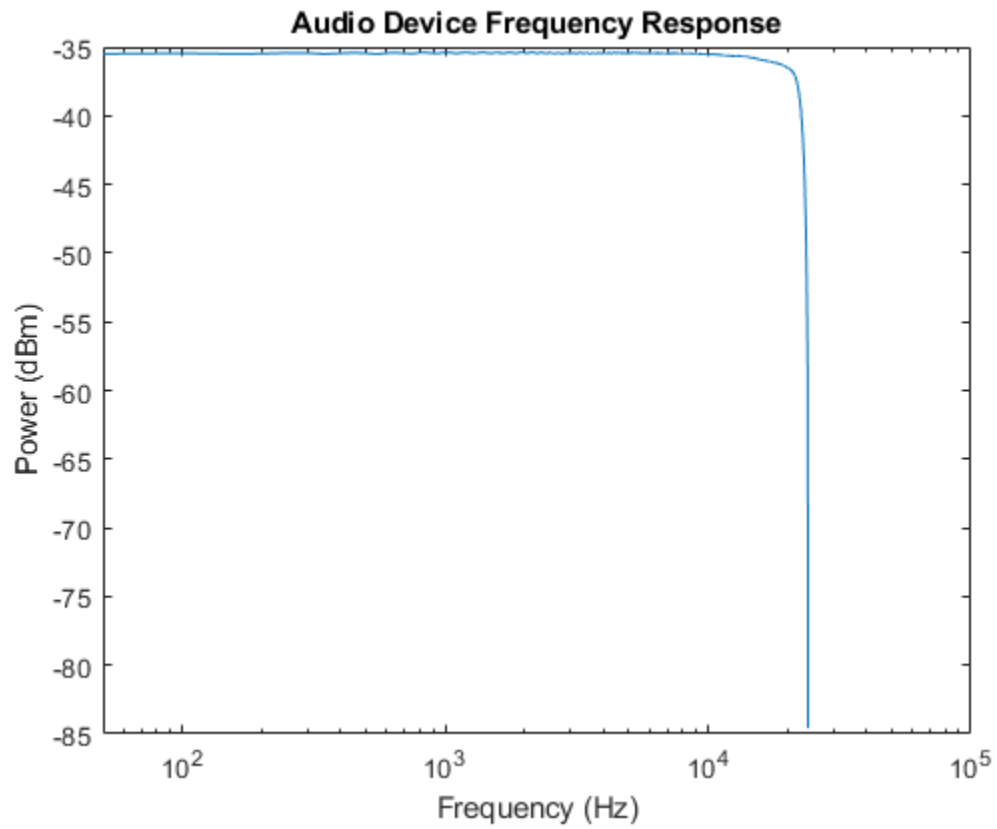


Frequency Response Measurement Results

The spectrum analyzer shows two plots. The first plot is the spectrum estimate of the last recorded data. The second plot is the maximum power the spectrum analyzer computed for each frequency bin, as the sine wave swept over the spectrum. To get the maximum hold plot data and the frequency vector, you can use the object function `getSpectrumData` and plot the maximum hold trace only.

```
data = getSpectrumData(scope);
freqVector = data.FrequencyVector{1};
freqResponse = data.MaxHoldTrace{1};

semilogx(freqVector, freqResponse);
xlabel('Frequency (Hz)');
ylabel('Power (dBm)');
title('Audio Device Frequency Response');
```



The frequency response plot indicates that the audio device tested in this example has a flat frequency response in the audible range.

Generate Standalone Executable for Parametric Audio Equalizer

This example shows how to generate a standalone executable for parametric equalization using MATLAB Coder™ and use it on an audio file. `multibandParametricEQ` is used for the equalization algorithm. The example allows you to dynamically adjust the coefficients of the filters using a user interface (UI) that is running in MATLAB.

Introduction

`multibandParametricEQ` allows up to ten equalizer bands in cascade. In this example, you create an equalizer with three bands. Each of the three biquad filters allows three parameters to be tuned: center frequency, Q factor, and the peak (or dip) gain.

`audioEqualizerEXEExampleApp` creates a UI to tune filter parameters and plot the magnitude response of the equalizer. `HelperEqualizerEXEProcessing` iteratively reads audio from a file, applies the 3-band parametric equalization algorithm on it, and plays the output of the equalization. Anytime during the simulation, it can also respond to the changes in the sliders of the MATLAB UI. This section goes into the standalone executable.

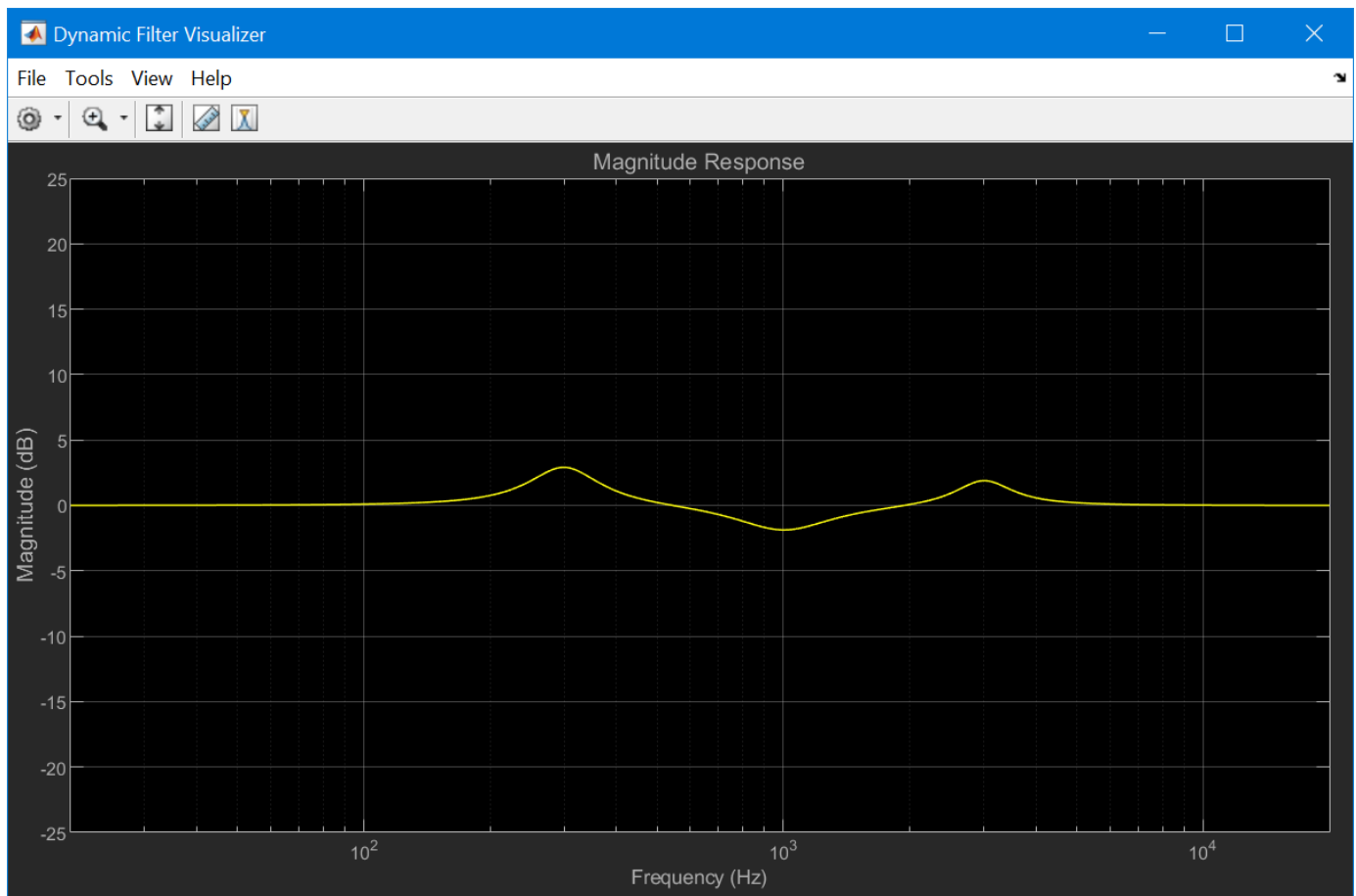
Generating Code and Building an Executable File

You can use MATLAB Coder to generate readable and standalone C-code from the parametric equalizer algorithm code. Because the algorithm code uses System objects for reading and playing audio files, there are additional dependencies for the generated code and executable file. These are available in the `/bin` directory of your MATLAB installation.

Run `HelperAudioEqualizerGenerateEXE` to invoke MATLAB Coder to automatically generate C-code and a standalone executable from the algorithm code present in `HelperEqualizerEXEProcessing`.

Running the example

Once you have generated the executable, run `audioEqualizerEXEExampleApp` to launch the executable and a user interface (UI) designed to interact with the simulation. The UI allows you to tune parameters and the results are reflected in the simulation instantly. For example, moving the slider for the 'Center Frequency1' to the right while the simulation is running increases the center frequency of the first parametric equalizer biquad filter. You can verify this by noticing the change immediately in the magnitude response plot.



 Parametric Equalizer Tuning

Center Frequency1

300

Quality Factor 1

2

Peak Gain1 (dB)

3

Center Frequency2

1000

Quality Factor 2

1.5

Peak Gain2 (dB)

-2

Center Frequency3

3000

Quality Factor 3

2.5

Peak Gain3 (dB)

2

Reset

Pause Simulation

Stop Simulation

Deploy Audio Applications with MATLAB Compiler

This example shows how to use MATLAB Compiler™ to create a standalone application from a MATLAB function. The function implements an audio processing algorithm and plays the result through your audio output device.

Introduction

In this example, you generate and run an executable application that applies artificial reverberation to an audio signal and plays it through your selected audio device. The benefit of such applications is that they can be run on a machine that need not have MATLAB installed. You would only need an installation of MATLAB Runtime to deploy the application created in this example.

Reverberation Algorithm

The reverberation algorithm is implemented using the System object `reverberator`. It allows you to add a reverberation effect to mono or stereo channel audio input. The object provides six properties that control the nature of reverberation. Each of them can be tuned while the simulation is running.

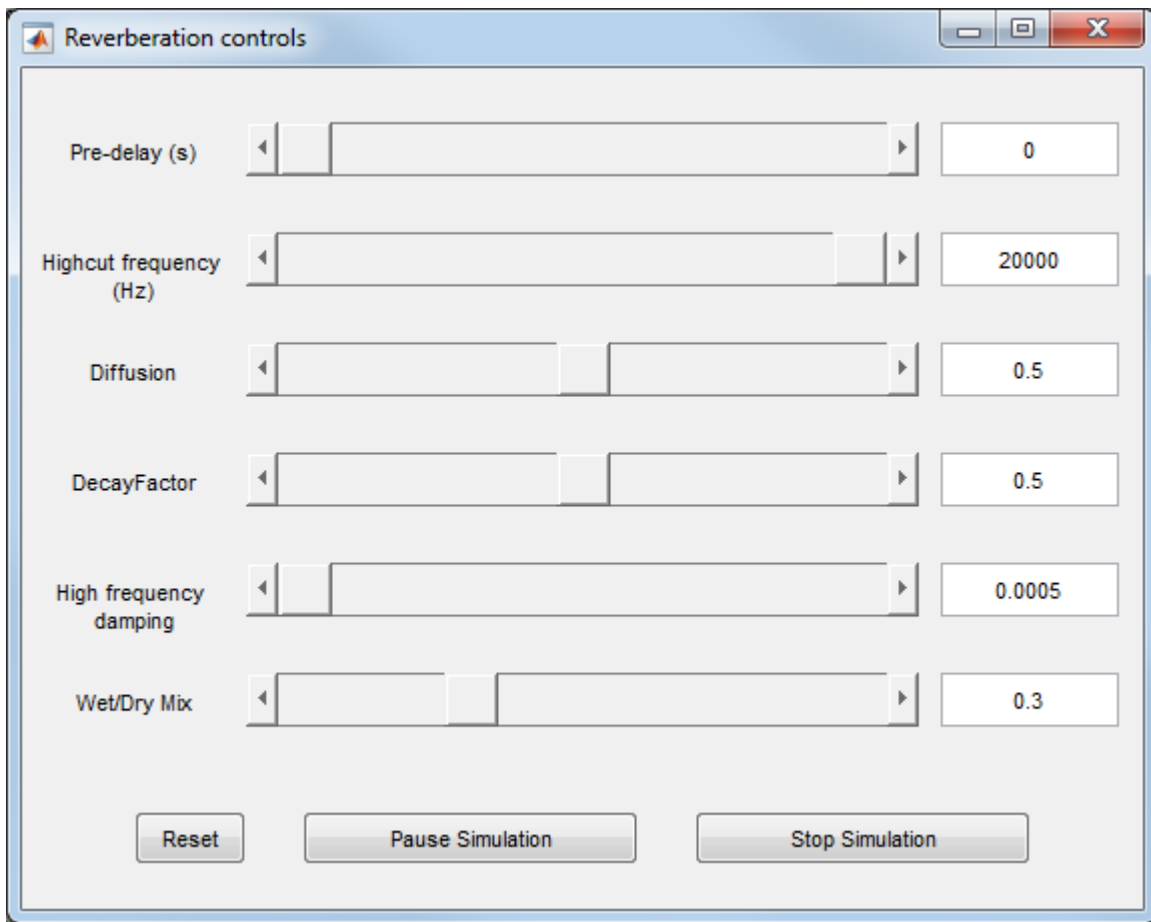
MATLAB Simulation

The function `audioReverberationCompilerExampleApp` is a wrapper around `reverberator`. To verify the behavior of `audioReverberationCompilerExampleApp`, run the function in MATLAB. It takes an optional input which is time, in seconds, for which you want to play the audio. The default value is 60.

`audioReverberationCompilerExampleApp`

The function `audioReverberationCompilerExampleApp` uses the `getAudioDevices` method of `audioDeviceWriter` to list the audio output devices available on the current machine so that you can play reverberated audio through the sound card of your choice. This is particularly helpful in deployed applications because function authors rarely know what device will be connected on the target machine.

`audioReverberationCompilerExampleApp` also maps the tunable properties of `reverberator` to a UI so that you can easily tune them while the simulation is running and observe its effect instantly. For example, move the slider 'Diffusion' to the right while the simulation is running. You will hear an effect of increase in the density of reflections. You can use the buttons on the UI to pause or stop the simulation.



Create a Temporary Directory for Compilation

Once you have verified the MATLAB simulation, you can compile the function. Before compiling, create a temporary directory in which you have write permissions. Copy the main MATLAB function and the associated helper files into this temporary directory.

```
compilerDir = fullfile(tempdir,'compilerDir'); %Name of temporary directory
if ~exist(compilerDir,'dir')
    mkdir(compilerDir); % Create temporary directory
end
curDir = cd(compilerDir);
copyfile(which('audioReverberationCompilerExampleApp'));
copyfile(which('HelperAudioReverberation'));
copyfile(which('FunkyDrums-44p1-stereo-25secs.mp3'));
copyfile(which('HelperCreateParamTuningUI'));
copyfile(which('HelperUnpackUIData'));
```

Compile the MATLAB Function into a Standalone Application

Use the `mcc` (MATLAB Compiler) function from MATLAB Compiler to compile `audioReverberationCompilerExampleApp` into a standalone application. This will be saved in the current directory. Specify the `-m` option to generate a standalone application, `-N` option to include only the directories in the path specified using the `-p` option.

```
mcc('-mN','audioReverberationCompilerExampleApp', ...
    '-p',fullfile(matlabroot,'toolbox','dsp'), ...
    '-p',fullfile(matlabroot,'toolbox','audio'));
```

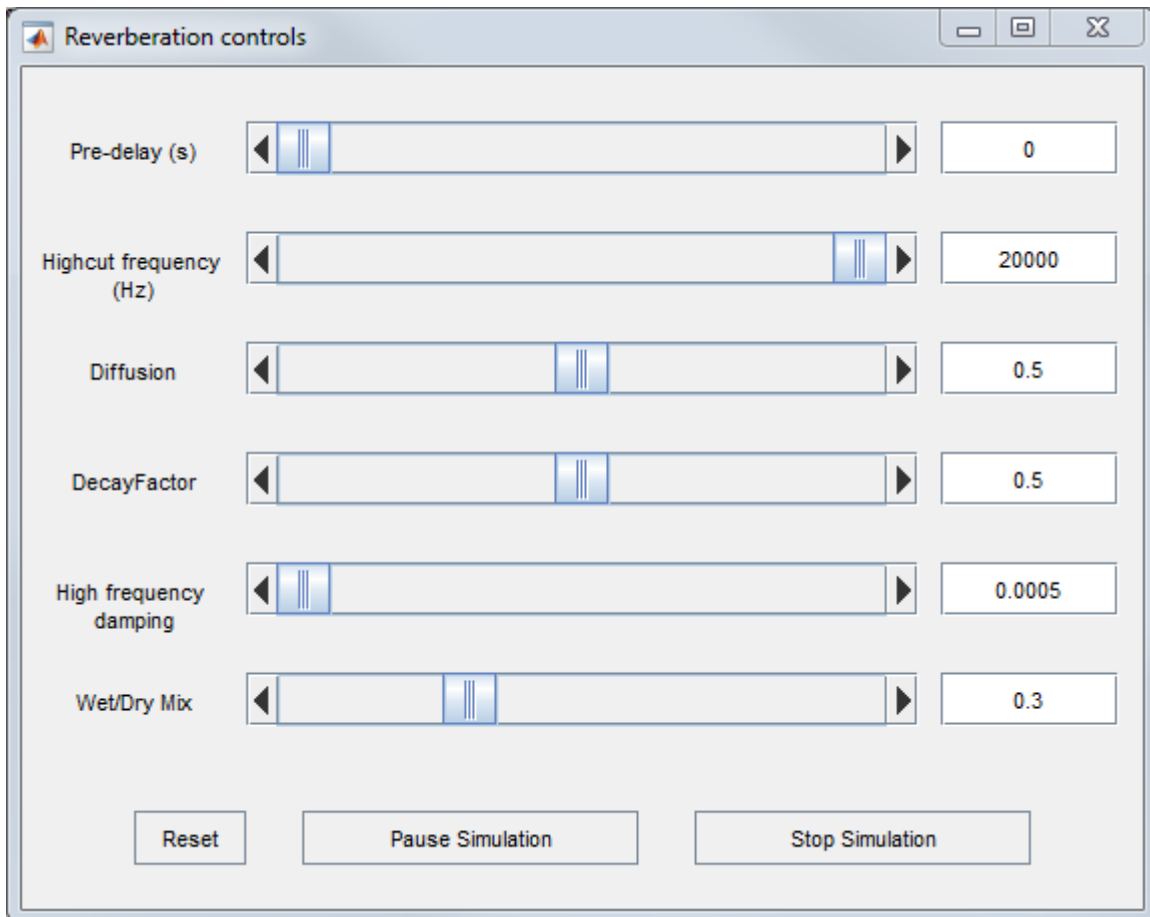
This step takes a few minutes to complete.

Run the Generated Application

Use the system command to run the generated standalone application. Note that running the standalone application using the system command uses the current MATLAB environment and any library files needed from this installation of MATLAB. To deploy this application on a machine which does not have MATLAB installed, refer to “About the MATLAB Runtime” (MATLAB Compiler).

```
if ismac
    status = system(fullfile('audioReverberationCompilerExampleApp', ...
        'Contents','MacOS','audioReverberationCompilerExampleApp'));
else
    status = system(fullfile(pwd,'audioReverberationCompilerExampleApp'));
end
```

Similar to the MATLAB simulation, running this deployed application will first ask you to choose the audio device that you want to use to play audio. Then, it launches the user interface (UI) to interact with the reverberation algorithm while the simulation is running.



Clean up Generated Files

After generating and deploying the executable, you can clean up the temporary directory by running the following in the MATLAB command prompt:

```
cd(curDir);  
rmdir(compilerDir, 's');
```

Parametric Audio Equalizer for Android Devices

This example shows how to use the Parametric EQ block and the `multibandParametricEQ System` object™ from the Audio Toolbox™ to implement a parametric audio equalizer model. You can run the model on your host computer or deploy it to an Android device.

Introduction

Parametric equalizers are used to adjust the frequency response of audio systems. For example, a parametric equalizer can compensate for biases introduced by specific speakers. Equalization is a primary tool in audio recording technologies.

In this example, you design a parametric audio equalizer in a Simulink® model. You can run your model on the host computer or an Android device. The equalization algorithm is a cascade of three filters with tunable center frequencies, bandwidths, and gains. You can visualize the frequency response in real time while adjusting the parameters.

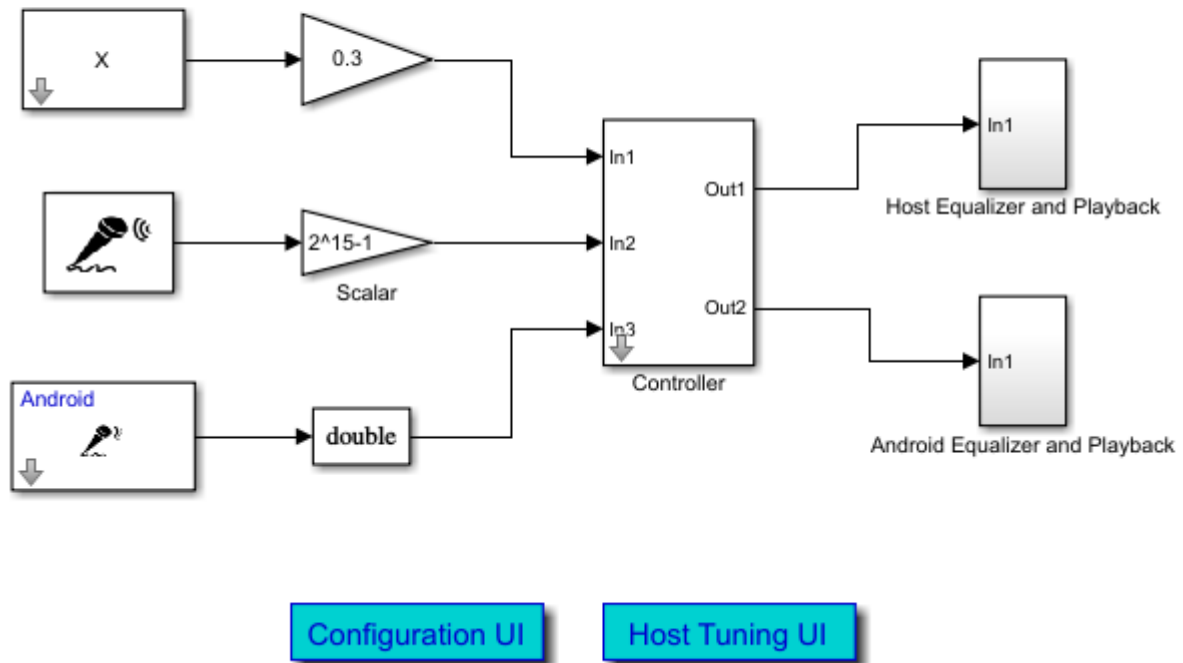
Required Hardware

To run this example on Android devices you need the following hardware:

- Android phone or tablet
- USB cable to connect the device to your development (host) computer

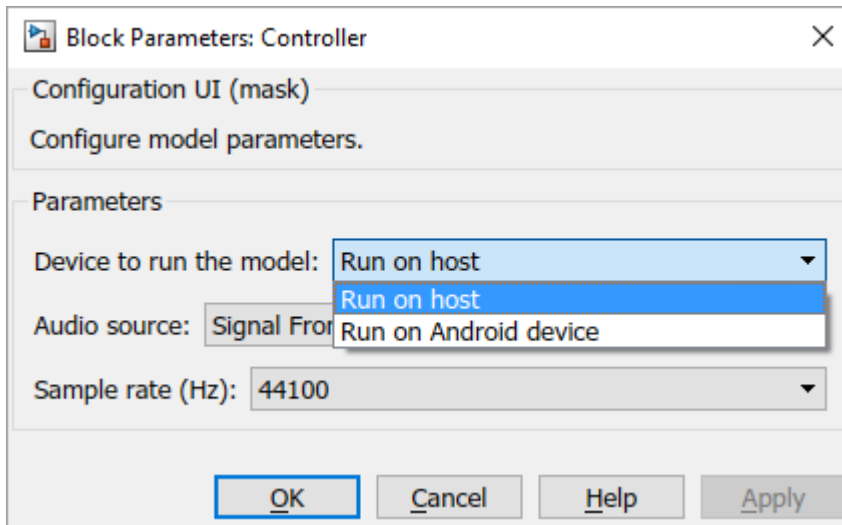
Model Setup

Parametric Audio Equalizer



The `audioEqualizerAndroid` model provides a choice of device (host computer or Android device), and audio source (MATLAB workspace or microphone). You can choose the configuration by clicking the Configuration UI button.

Configuration UI:



Run Model on the Host Computer

When you choose to run the model on the host computer, a UI designed to interact with the simulation is provided and can be opened by clicking **Host Tuning UI**.

Host Tuning UI:

Block Parameters: Equalizer [X]

Three-Band Equalizer (mask)

Three-band equalizer compatible with a mono or stereo input. Tune the sliders to adjust the Center Frequency, Bandwidth, and Gain of each filter. Click the "View Frequency Response" button to view the overall filter response.

Parameters

Filter 1

Center Frequency (Hz)	Bandwidth (Hz)	Gain (dB)
20.0 — 20000.0 1218.800	1.0 — 10000.0 300.970	-20.0 — 20.0 15.600

Filter 2

Center Frequency (Hz)	Bandwidth (Hz)	Gain (dB)
20.0 — 20000.0 4415.600	1.0 — 10000.0 1000.900	-20.0 — 20.0 12.800

Filter 3

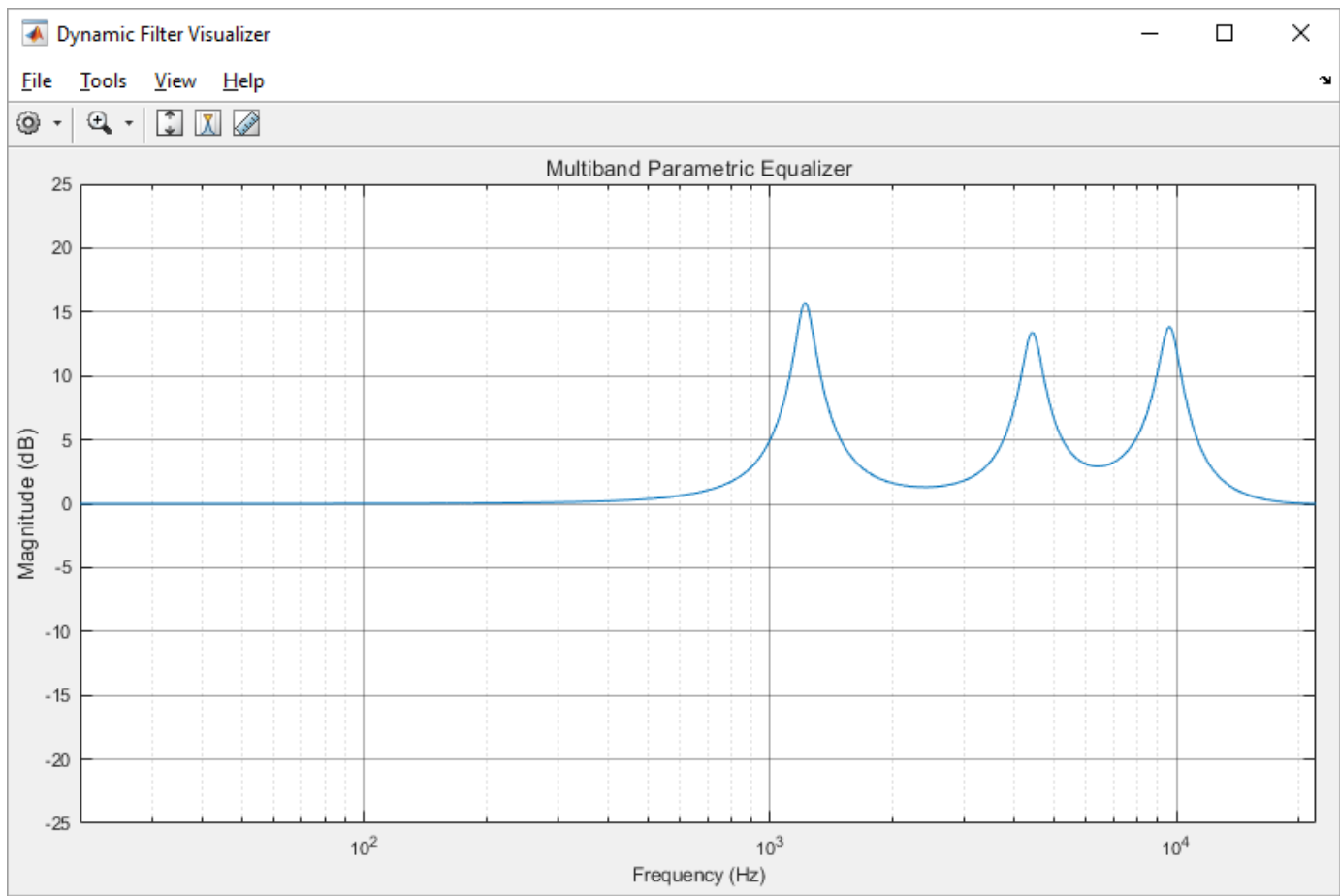
Center Frequency (Hz)	Bandwidth (Hz)	Gain (dB)
20.0 — 20000.0 9610.400	1.0 — 10000.0 2300.770	-20.0 — 20.0 13.600

☐ Bypass View Frequency Response

OK Cancel Help Apply

The UI allows you to tune the parameters of three filters individually, and view the frequency response in real time. You can also check the Bypass check box to compare the modified sound with the original sound.

Click the View Frequency Response button to visualize the frequency response of the filters.

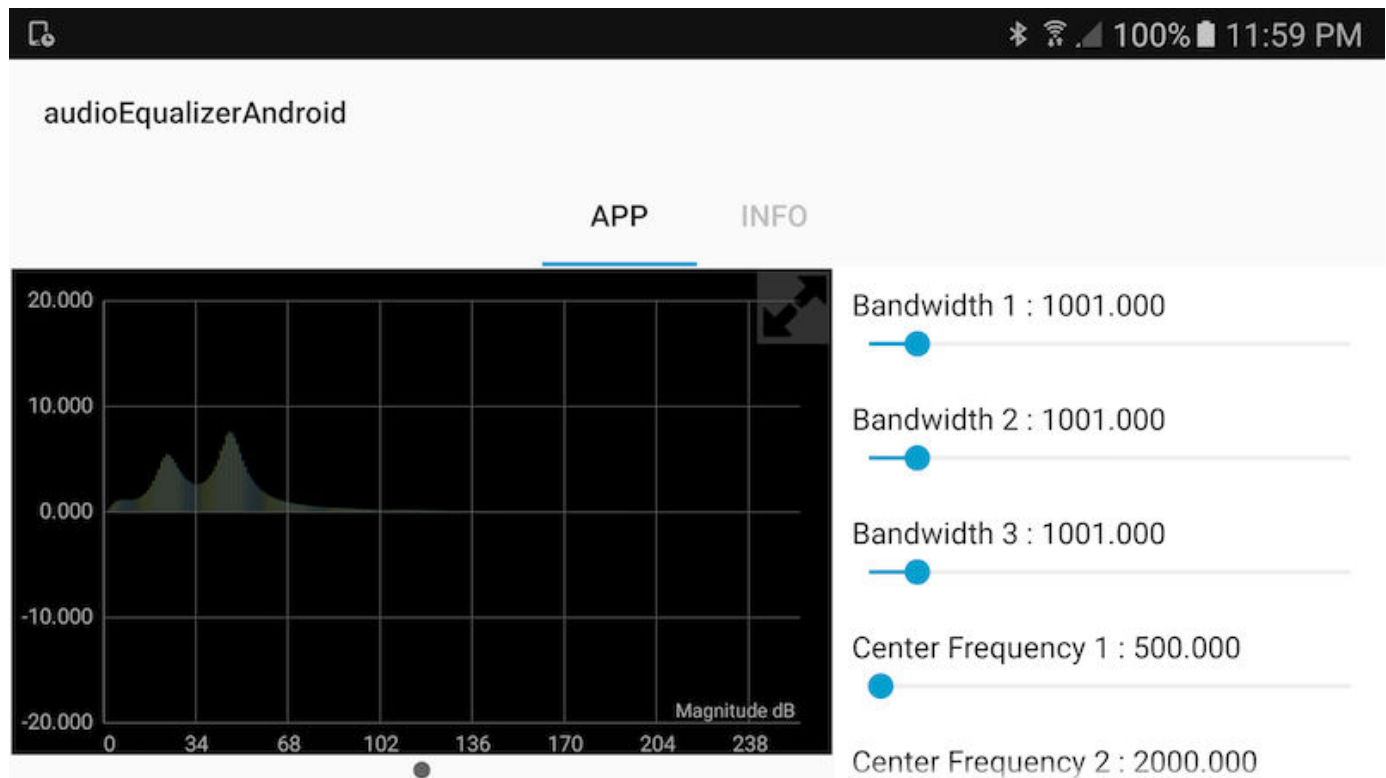


Run Model on an Android Device

To run the model on your Android device, you need to first ensure that you have installed **Simulink Support Package for Android Devices** and that your Android device is provisioned.

Once your Android device is properly configured, use a USB cable to connect the device to your host computer.

You can choose to make a standalone Android equalizer app by clicking the **Deploy to hardware** button on the Simulink Editor toolbar. After deployment, the app can run on your Android device even when it is disconnected from the host computer. The parameter tuning UI and the frequency response display on your Android device screen, as shown below:



Parametric Audio Equalizer for iOS Devices

This example shows how to use the Parametric EQ block and the `multibandParametricEQ` System object™ to implement a parametric audio equalizer model, that can run on your host computer or an Apple iOS device.

Introduction

Parametric equalizers are used to adjust the frequency response of audio systems. For example, a parametric equalizer can compensate for biases introduced by specific speakers. Equalization is a primary tool in audio recording technologies.

In this example, you design a parametric audio equalizer in a Simulink® model. You can run your model on the host computer or an iOS device. The equalization algorithm is a cascade of three filters with tunable center frequencies, bandwidths, and gains. You can visualize the frequency response in real time while adjusting the parameters.

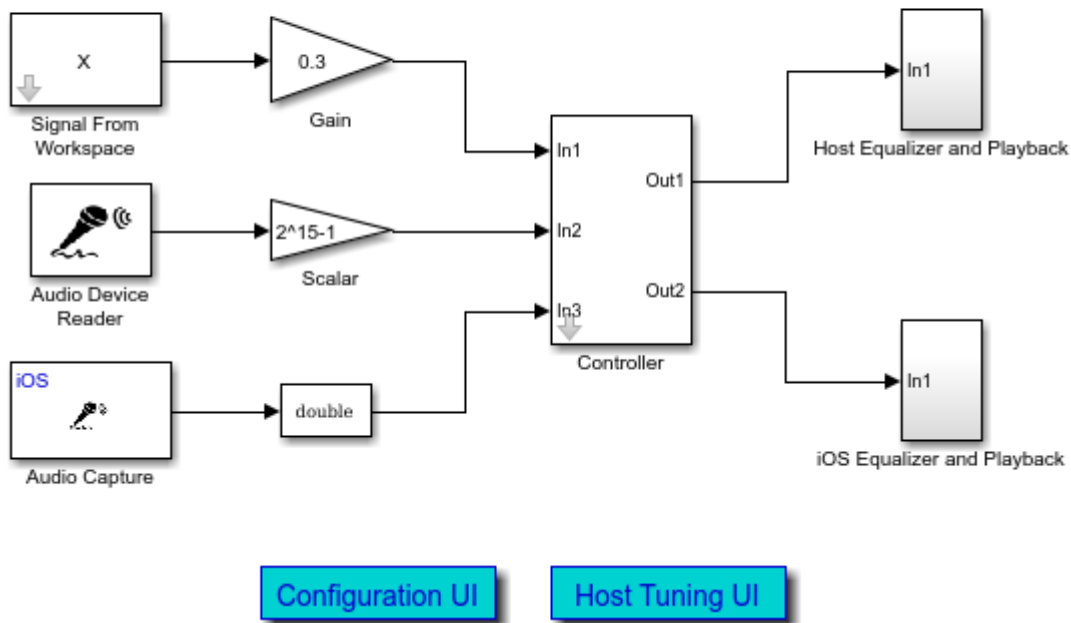
Required Hardware

To run this example on iOS devices you need the following hardware:

- iPhone, iPod or an iPad
- Host computer with Mac OS X system
- USB cable to connect the iOS device to host computer

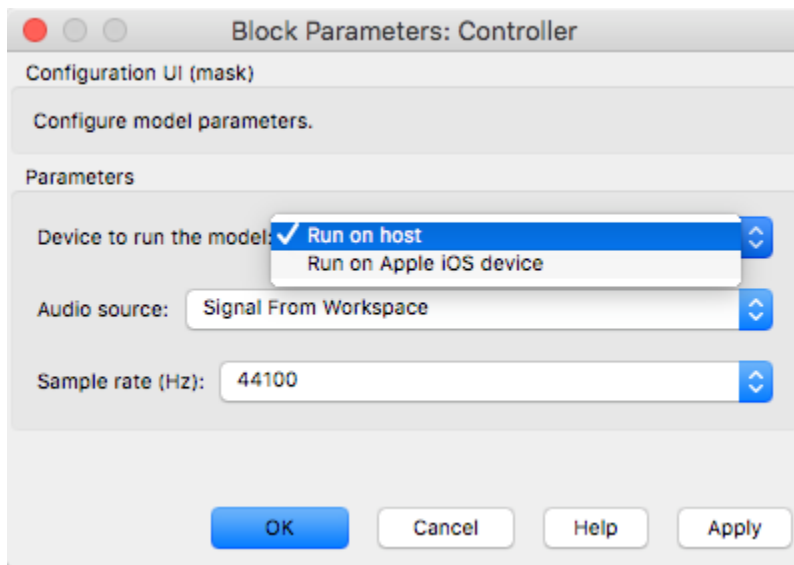
Model Setup

Parametric Audio Equalizer



The `audioequalizeriOS` model provides a choice of device (host computer or iOS device), and audio source (MATLAB workspace or microphone). You can choose the configuration by clicking the Configuration UI button.

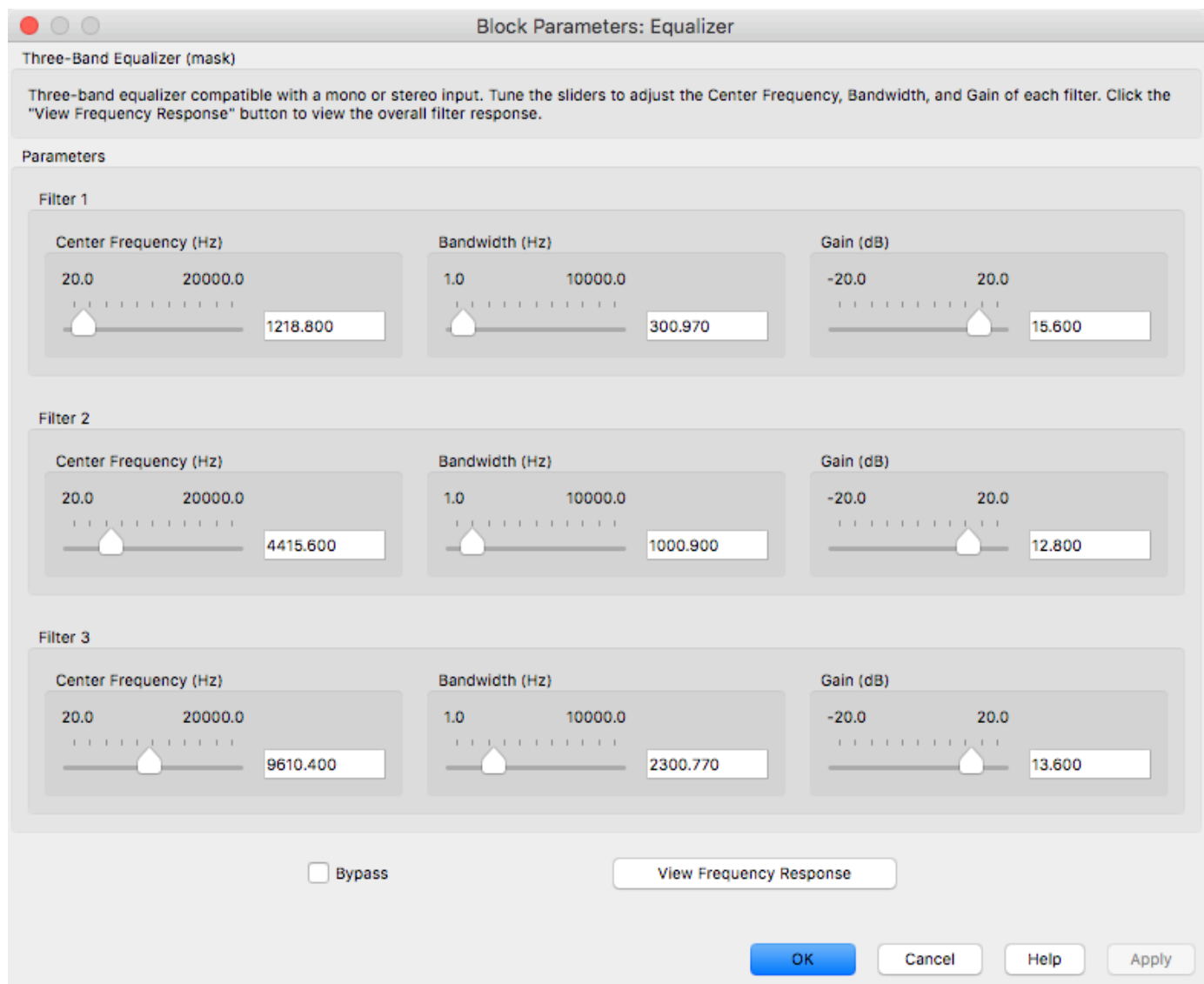
Configuration UI:



Run Model on the Host Computer

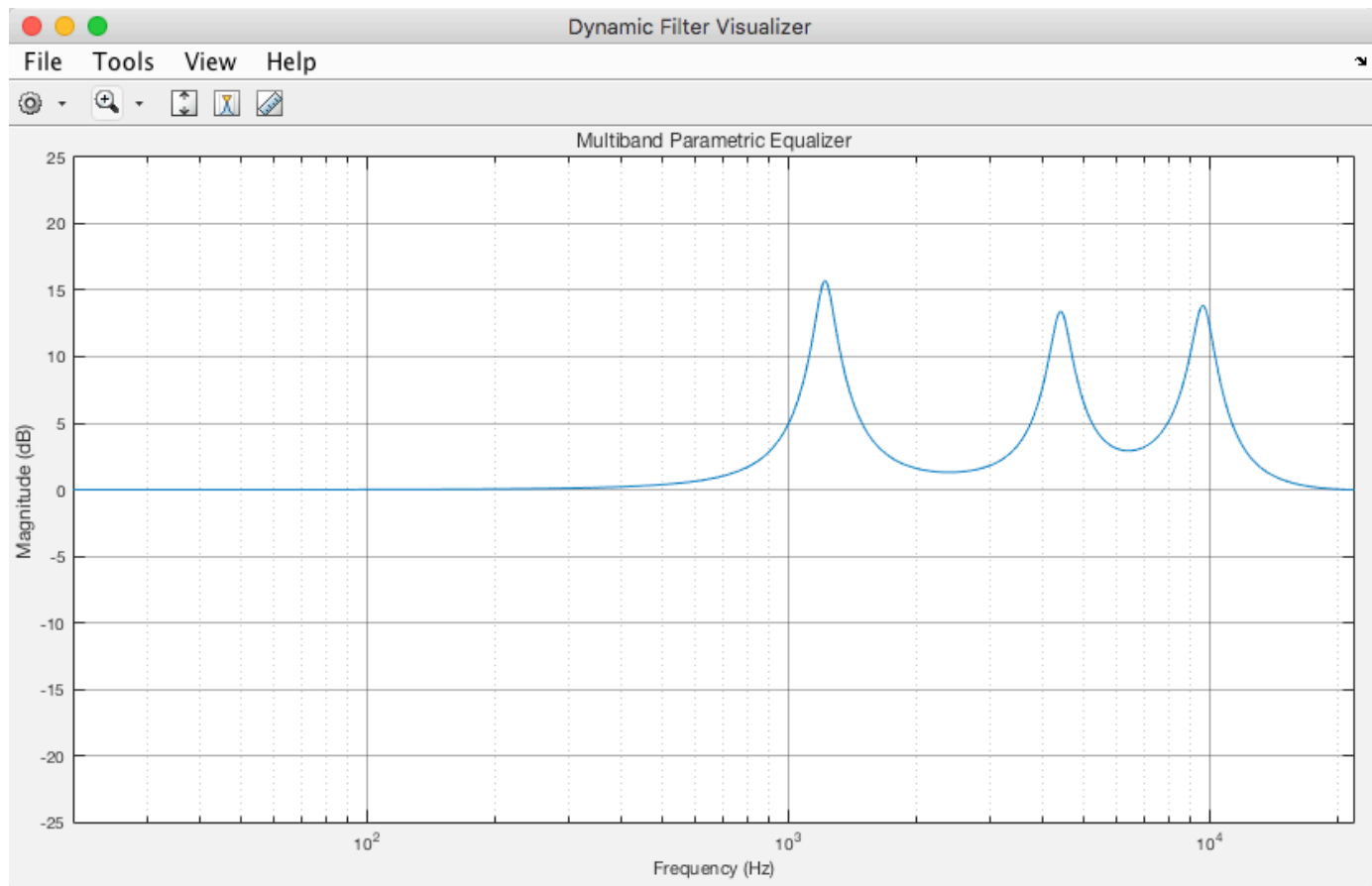
When you choose to run the model on the host computer, a UI designed to interact with the simulation is provided and can be opened by clicking **Host Tuning UI**.

Host Tuning UI:



The UI allows you to tune the parameters of three filters individually, and view the frequency response in real time. You can also check the Bypass check box to compare the modified sound with the original sound.

Click the View Frequency Response button to visualize the filters frequency response.

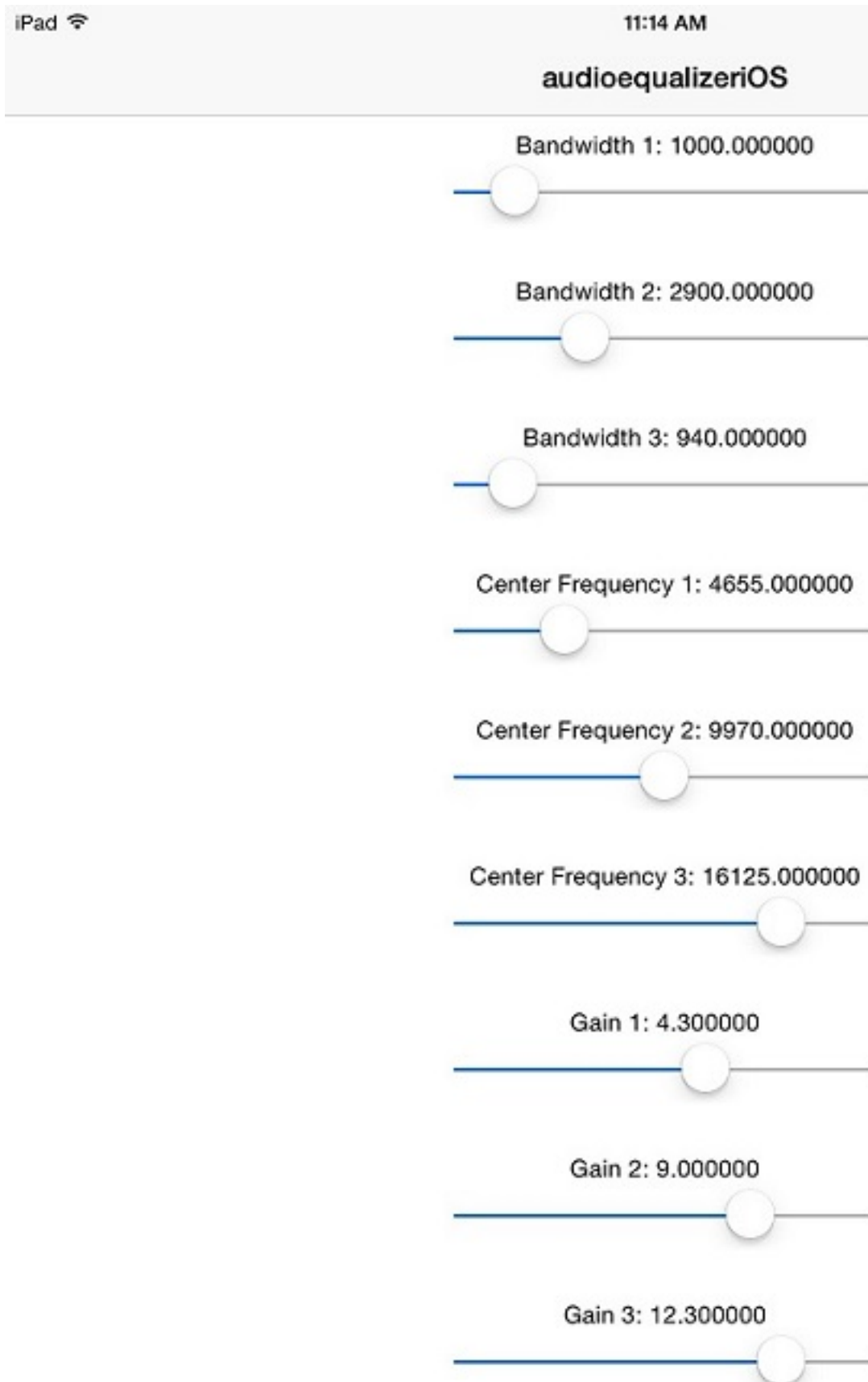


Run Model on an Apple iOS Device

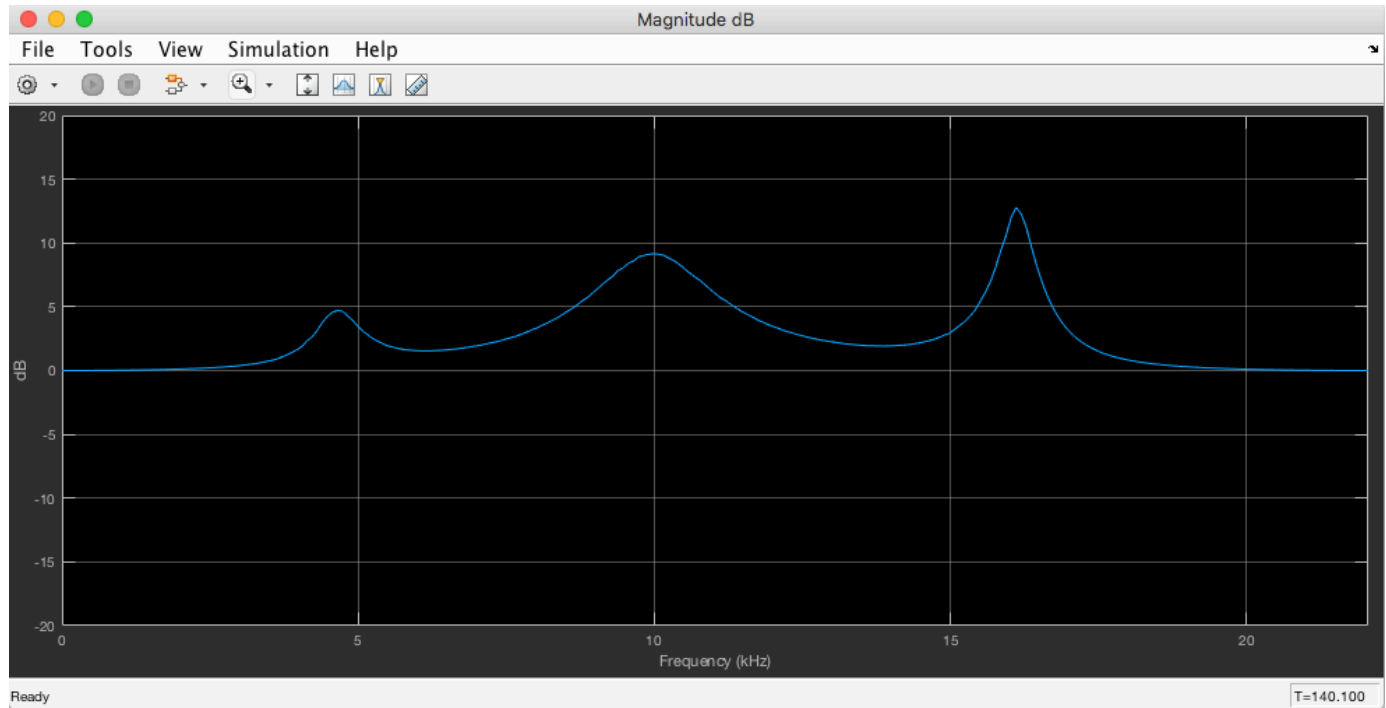
To run the model on your Apple iOS device, you need to first ensure that you have installed **Simulink Support Package for Apple iOS Devices** and that your iOS device is provisioned.

Once your iOS device is properly configured, use a USB cable to connect the device to your host computer.

You can choose to make an iOS standalone equalizer app by clicking the **Deploy to hardware** button on the Simulink Editor toolbar. After deployment, the app can run on your iOS device even when it is disconnected from the host computer. The parameter tuning UI displays on your iOS device screen, as shown below:



You can also run the model in **External** mode by clicking the Run button on the Simulink Editor toolbar. To run in **External** mode, the iOS device must stay connected to the host computer. This mode enables you to view the frequency response on the host computer while adjusting parameters on your iOS device. Frequency response will display on the host screen as follows:



Audio Effects for iOS Devices

This example shows how to use System objects™ from Audio Toolbox™ to implement echo and reverberation effects in a Simulink® model. You can run the model on your host computer or deploy it to an Apple iOS device.

Introduction

Echo and reverberation are two commonly-used audio effects in recording, movie making, and sound design. Echo is a reflection of sound that arrives at the listener with a delay after the direct sound. Echo can be produced by the bottom of a well or by the walls of a building. Reverberation is a large number of sound reflections building up and then decaying. A common use of reverberation is to simulate music played in a closed room. Most digital audio workstations (DAWs) have options to add echo and reverberation effects to sound tracks.

In this example, you design and implement echo and reverberation audio effects in a Simulink model. You can run your model on the host computer or an Apple iOS device.

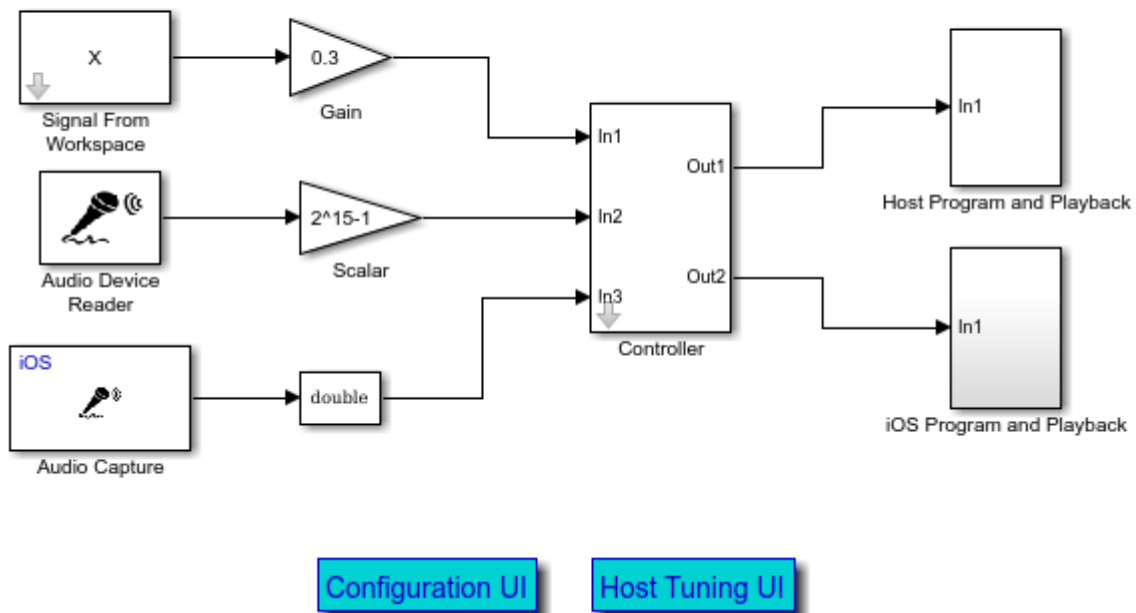
Required Hardware

To run this example on iOS devices you will need the following hardware:

- iPhone, iPod or an iPad
- Host computer with Mac OS X system
- USB cable to connect the device to host computer

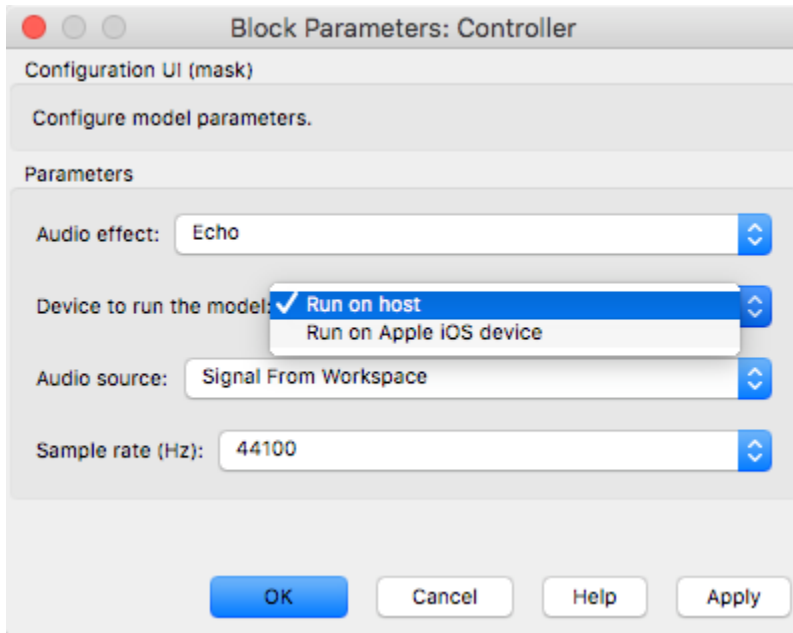
Model Setup

Audio Effects



The `audioeffectsiOS` model provides a choice of audio effect (echo or reverberation), device (host computer or iOS device), and audio source (MATLAB workspace or microphone). You can choose the configuration by clicking the Configuration UI button.

Configuration UI:



Audio Effect: Echo

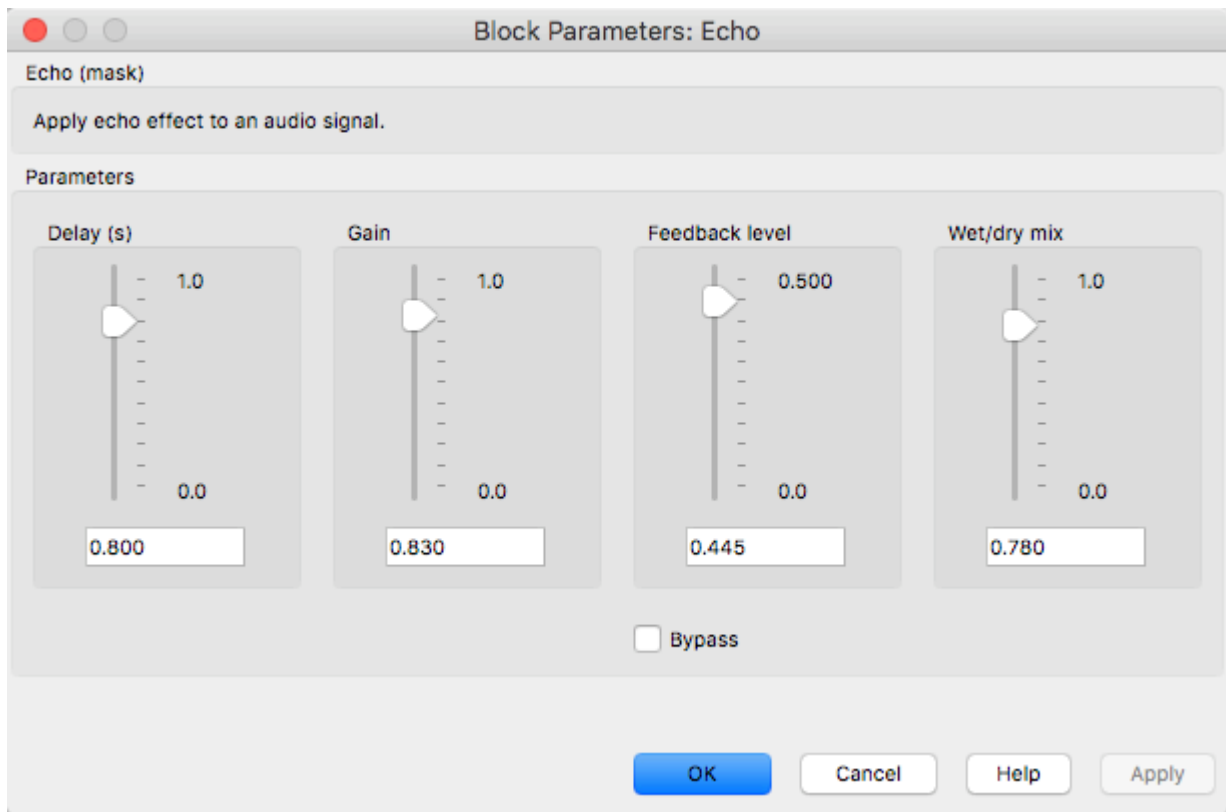
The echo effect has four tunable parameters that can be modified while the model is running:

- Delay - Delay applied to audio signal, in seconds
- Gain - Linear gain of the delayed audio
- FeedbackLevel - Feedback gain applied to delay line
- Wet/Dry Mix - Ratio of wet signal added to dry signal

Run Echo Effect on the Host Computer

If you choose to run the echo effect on your host computer, a UI designed to interact with the simulation is provided and can be opened by clicking **Host Tuning UI**. The UI allows you to tune echo parameters and hear the echo sound effect in real time.

Host tuning UI for echo effect:



Run Echo Effect on an Apple iOS Device

When you choose to run the echo effect on your Apple iOS device, you need to first ensure that you have installed Simulink Support Package for Apple iOS Devices and that your iOS device is provisioned.

Once your iOS device is properly configured, use a USB cable to connect the device to your host computer.

You can choose to make an iOS standalone echo effect app by clicking the **Deploy to hardware** button on the Simulink Editor toolbar. After deployment, the app can run on your iOS device even when it is disconnected from the host computer. You can also run the model in **External** mode by clicking the **Run** button on the Simulink Editor toolbar. To run in **External** mode, the iOS device must stay connected to the host computer.

The UI for the echo effect displays on your iOS device screen, as shown below:

iPad

3:29 PM

audioeffectsiOS

Delay/s: 0.500000



Feedback Level: 0.350000



Gain/dB: 0.000000



Wet/Dry Mix: 0.500000



Bypass

Mute

Audio Effect: Reverberation

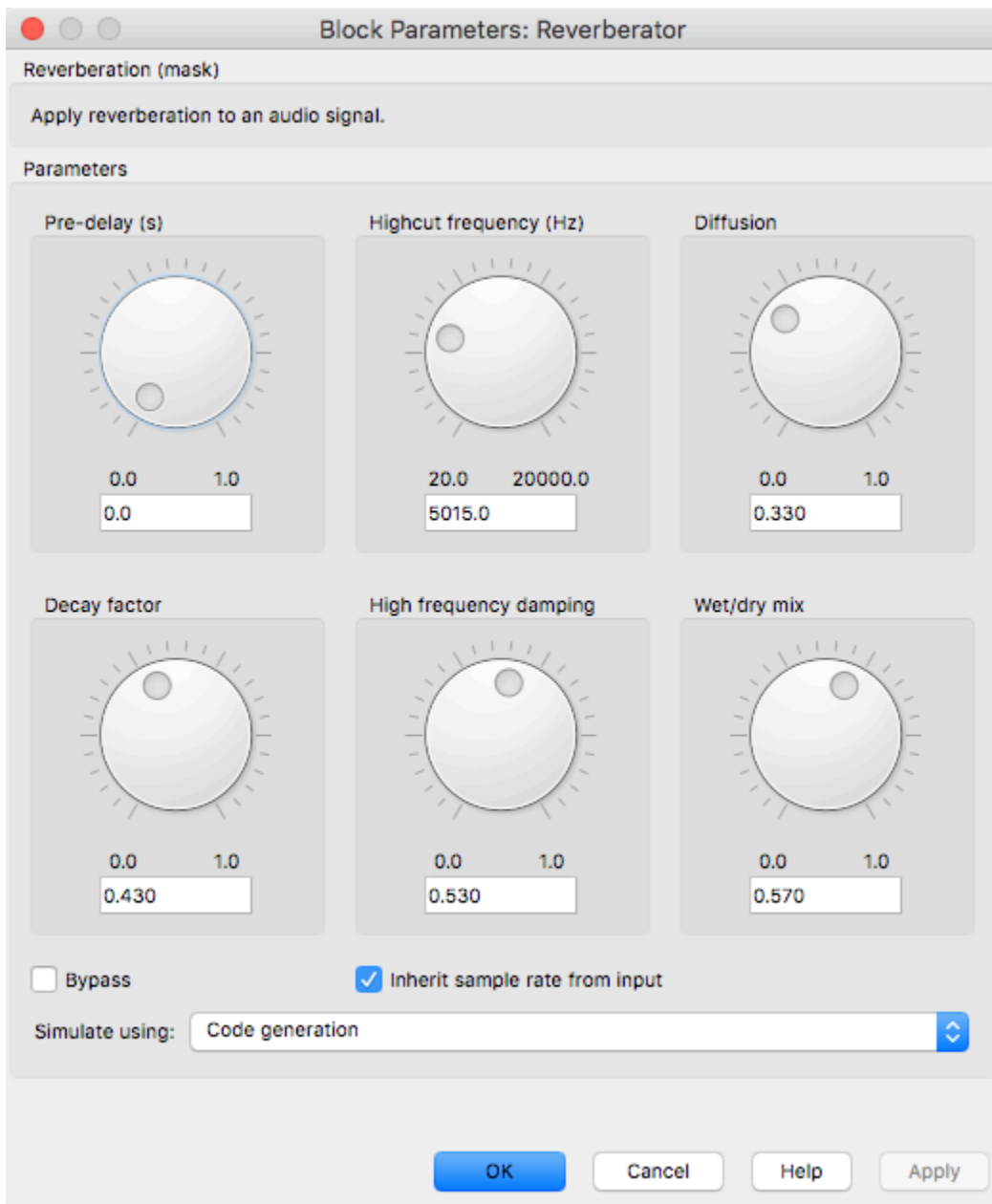
The reverberation effect has six tunable parameters that can be modified while the model is running:

- Pre-delay - Time between hearing direct sound and the first early reflection
- Highcut frequency - Cutoff frequency for the lowpass filter at the front of the reverberator structure
- Diffusion - Density of reverb tail
- Decay factor - Decay factor of reverb tail
- High Frequency Damping - Attenuation of high frequencies in the reverberation output
- Wet/Dry Mix - Ratio of wet signal added to dry signal

Run Reverberation Effect on the Host Computer

If you choose to run the reverberation effect on your host computer, a UI designed to interact with the simulation is provided and can be opened by clicking **Host Tuning UI**. The UI allows you to tune reverberation parameters and hear the reverberation sound effect in real time.

Host tuning UI for reverberation effect:



Run Reverberation Effect on an Apple iOS Device

When you choose to run the reverberation effect on your Apple iOS device, you need to first ensure that you have installed Simulink Support Package for Apple iOS Devices and that your iOS device is provisioned.

Once your iOS device is properly configured, use a USB cable to connect the device to your host computer.

You can choose to make an iOS standalone reverberation effect app by clicking the **Deploy to hardware** button on the Simulink Editor toolbar. After deployment, the app can run on your iOS device even when it is disconnected from the host computer. You can also run the model in External

mode by clicking the Run button on the Simulink Editor toolbar. To run in External mode, the iOS device must stay connected to the host computer.

The UI for the reverberation effect displays on your iOS device screen, as shown below:

iPad

3:25 PM

audioeffectsiOS

Decay Factor: 0.500000



Diffusion: 0.500000



Gain/dB: 0.000000



High Frequency Damping: 0.050000



High_cut Frequen...Hz: 18880.000000



Pre_delay/s: 0.110000



Wet/Dry Mix: 0.300000



Bypass

Multiband Dynamic Range Compression for iOS Devices

This example shows how to use the Crossover Filter block and compressor System object™ from the Audio Toolbox™ to implement a multiband dynamic range compressor model. You can run the model on your host computer or deploy it to an Apple iOS device.

Introduction

Dynamic range compression reduces the dynamic range of a signal by attenuating the level of strong peaks, while leaving weaker peaks unchanged. Compression has applications in audio recording, mixing, and broadcasting.

Multiband compression compresses different audio frequency bands separately, by first splitting the audio signal into multiple bands and then passing each band through its own independently adjustable compressor. Multiband compression is widely used in audio mastering and is often included in digital audio workstations.

The multiband compressor in this example first splits an audio signal into different bands using a multiband crossover filter. Linkwitz-Riley crossover filters are used to obtain an overall allpass frequency response. Each band is then compressed using a separate dynamic range compressor. Key compressor characteristics, such as the threshold, the compression ratio, the attack time and the release time are independently tunable for each band. You can run the model either on the host computer or an Apple iOS device.

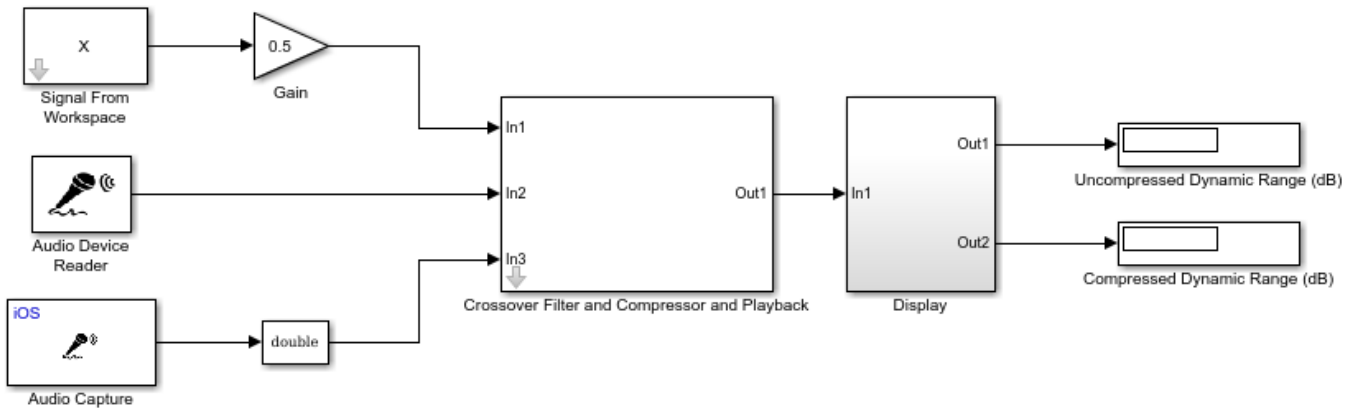
Required Hardware

To run this example on iOS devices you need the following hardware:

- iPhone, iPod or an iPad
- Host computer with Mac OS X system
- USB cable to connect the iOS device to host computer

Model Setup

Multiband Audio Dynamic Compression



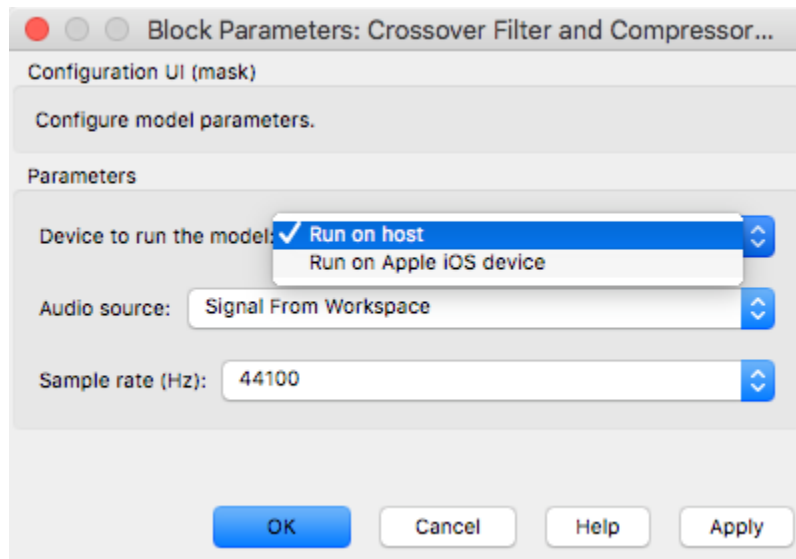
Configuration UI

Crossover Filter UI

Compressor Host Tuning UI

The `audiomultibandcompressoriOS` model is a cascade of audio sources, a multiband crossover filter, compressors, and a display subsystem. It provides a choice of model running device (host computer or iOS device) and audio source (MATLAB workspace or microphone). You can choose the configuration by clicking the Configuration UI button.

Configuration UI:



Crossover Filter

A crossover filter can split an audio signal into two or more frequency bands. Its overall magnitude frequency response is flat, which retains frequency domain properties of an input audio signal.

In this model, you use the Crossover Filter block from Audio Toolbox. You can open the block UI by clicking **Crossover Filter** UI and modify the cut-off frequencies.

Crossover Filter UI:

Block Parameters: Crossover Filter

Crossover Filter


Multiband audio crossover filter

[Source code](#)

Parameters

Number of crossovers

Crossover frequency (Hz)




20.0 20000.0

Crossover order

Slope: 12 dB/octave

Crossover frequency (Hz)




20.0 20000.0

Crossover order

Slope: 12 dB/octave

Crossover frequency (Hz)



20.0 20000.0

Crossover order

Slope: 12 dB/octave

☐ Inherit sample rate from input

Input sample rate (Hz):

Simulate using:

Note the `Number of crossovers` is set to 3 in this model to make a 4-band compressor. To make sure the model works properly, please keep `Number of crossovers` to be 3 and do not change it to other values.

Multiband Dynamic Range Compressor

In this example, the multiband dynamic range compressor is composed of four parallel single band compressors. Each single band compressor controls one frequency band, whose frequency range is set by the crossover filter.

There are four principal parameters for each single band compressor:

- `Threshold` - the level above which the input signal is compressed
- `Ratio` - the amount of compression
- `Attack time` - the time it takes the compressor gain to rise from 10% to 90% of its final value when the input goes above the threshold
- `Release time` - the time it takes the compressor gain to drop from 90% to 10% of its final value when the input goes below the threshold

In this example, you can modify the parameters for the four bands independently and view the static compression characteristic plots in real time.

Run Model on the Host Computer

When you choose to run the model on the host computer, you can tune the compressor parameters by clicking `Compressor Host Tuning UI`.

Compressor Host Tuning UI:

Block Parameters: Compressor Bank

Subsystem (mask)

Multiband Compressor

Parameters

☐ Bypass

Band 1

Threshold (dB):

-50.0 0.0
-5

Ratio:

1.0 50.0
5

Attack time (s):

0.0 4.0
0.005

Release time (s):

0.0 4.0
0.1

View static characteristic

Band 2

Threshold (dB):

-50.0 0.0
-10

Ratio:

1.0 50.0
5

Attack time (s):

0.0 4.0
0.005

Release time (s):

0.0 4.0
0.1

View static characteristic

Band 3

Threshold (dB):

-50.0 0.0
-20

Ratio:

1.0 50.0
5

Attack time (s):

0.0 4.0
0.002

Release time (s):

0.0 4.0
0.050

View static characteristic

Band 4

Threshold (dB):

-50.0 0.0
-30

Ratio:

1.0 50.0
5

Attack time (s):

0.0 4.0
0.002

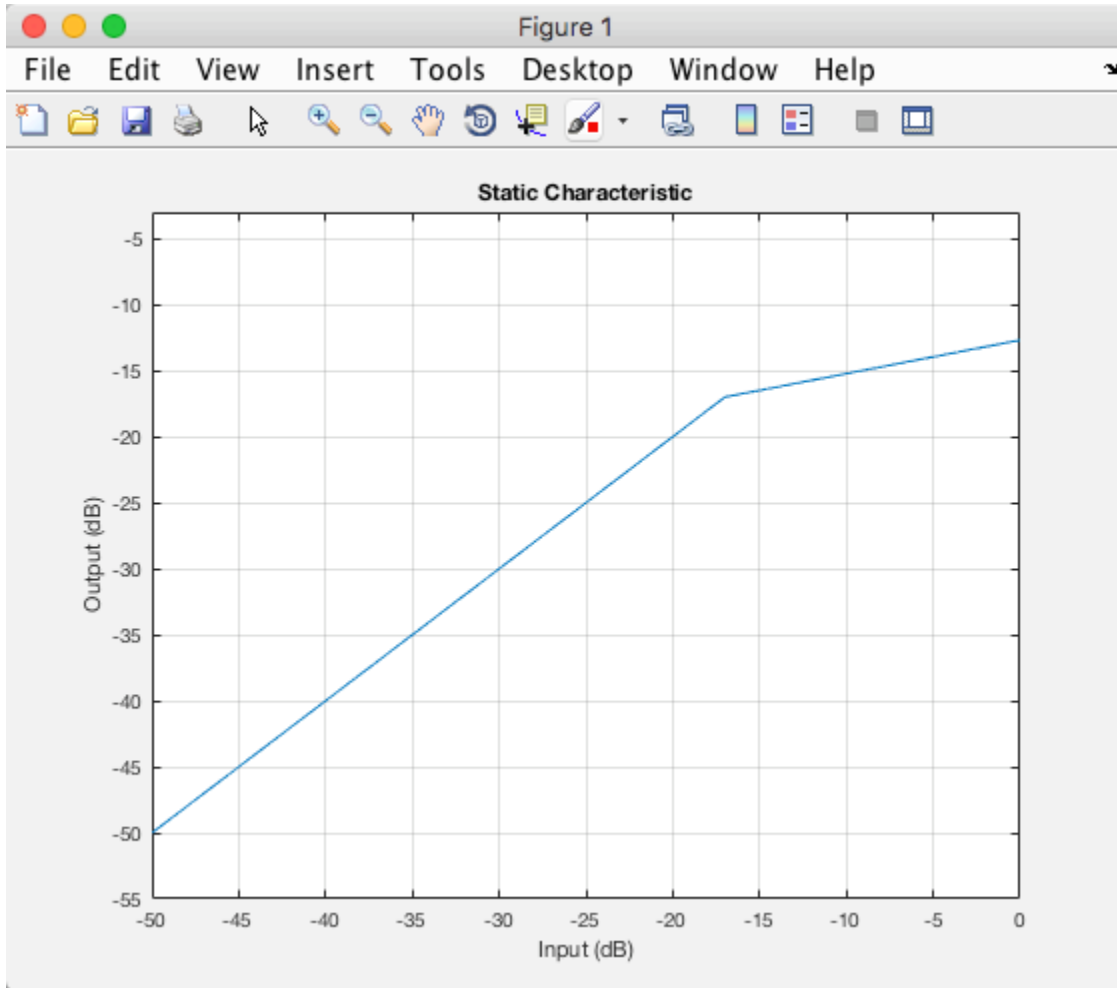
Release time (s):

0.0 4.0
0.050

View static characteristic

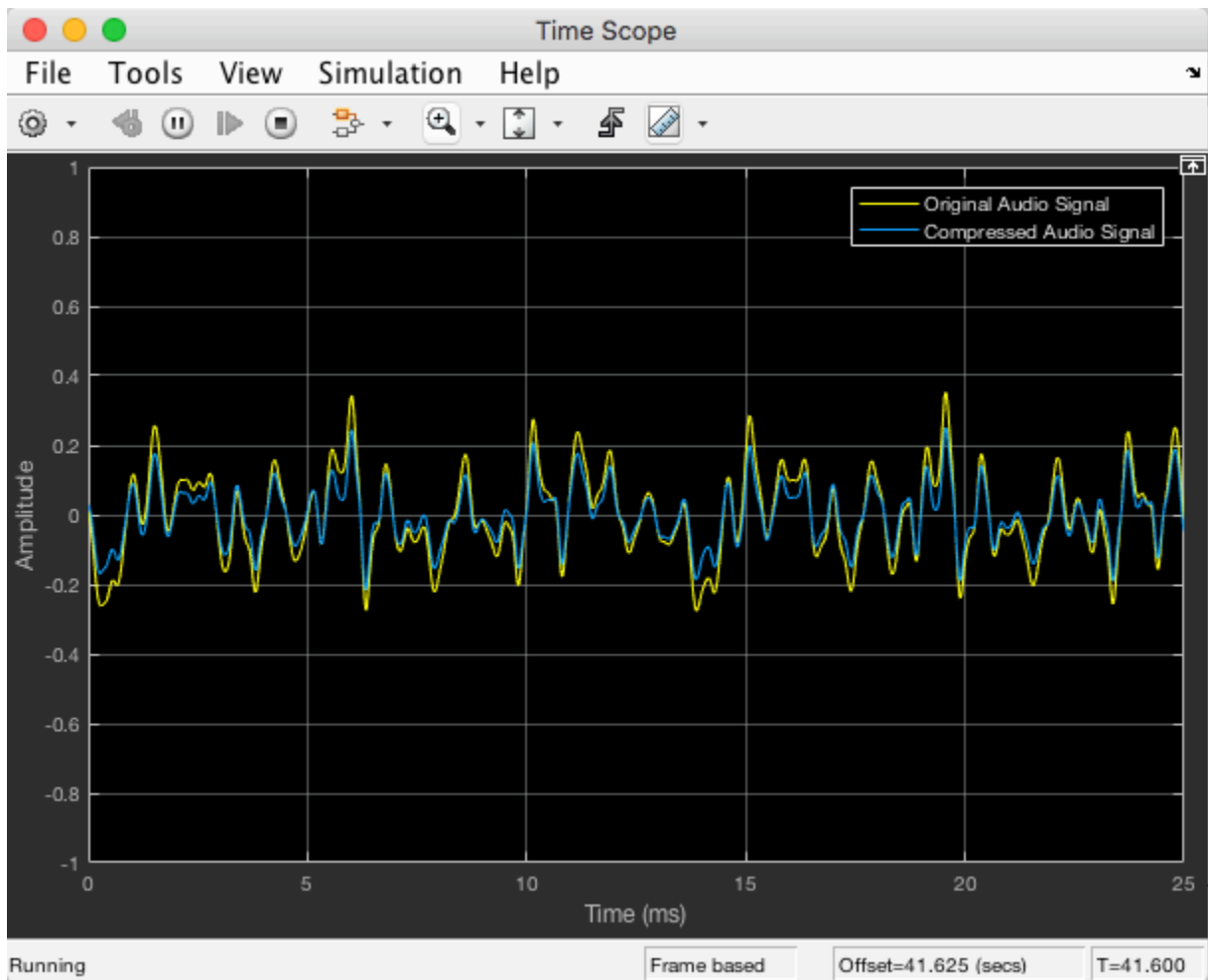
The UI enables you to tune the parameters of four single-band compressors individually, and view the static compression characteristics in real time. You can check the **Bypass** check box to compare the modified sound with the original sound.

Click the **View static characteristic** button to visualize the static compression characteristic plot.



To compare the dynamic range of the uncompressed and compressed signals, the dynamic range is computed and displayed on the Simulink Model Display bar. The waveform of the uncompressed and compressed signals is also plotted in real time.

Waveform of the uncompressed and compressed signals:



Run Model on an Apple iOS Device

To run the model on your Apple iOS device, you need to first ensure that you have installed Simulink Support Package for Apple iOS Devices and that your iOS device is provisioned.

Once your iOS device is properly configured, use a USB cable to connect the device to your host computer.

You can choose to make an iOS standalone app by clicking the **Deploy to hardware** button on the Simulink Editor toolbar. After deployment, the app can run on your iOS device even when it is disconnected from the host computer. The compressor parameter tuning UI and the dynamic range display are designed on your iOS device screen, as shown below:

iPad

3:11 PM

audiomultibandcompressoriOS

Attack time 1 (s): 1.000000



Ratio 1: 5.000000



Release time 1 (s): 0.100000



Threshold 1 (dB): -39.600002



Attack time 2 (s): 1.000000



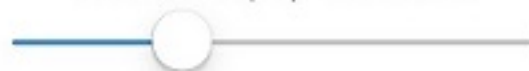
Ratio 2: 5.000000



Release time 2 (s): 0.100000



Threshold 2 (dB): -34.900002



Attack time 3 (s): 1.000000



iPad

3:11 PM

audiomultibandcompressor iOS

Release time 3 (s): 0.050000



Threshold 3 (dB): -20.000000



Attack time 4 (s): 1.000000



Ratio 4: 5.000000



Release time 4 (s): 0.050000



Threshold 4 (dB): -30.000000



Bypass

Mute

Compressed Dynamic Range (dB):

4.551470

You can also run the model in **External** mode by clicking the Run button on the Simulink Editor toolbar. To run in **External** mode, the iOS device must stay connected to the host computer. Besides tuning compressor parameters on the iOS device screen, in this mode, you can open the **Crossover Filter** UI on the host computer and modify the cut-off frequencies while the model is running. This mode also enables you to view the dynamic range of the uncompressed and compressed signals in real time on the host computer.

Denoise Speech Using Deep Learning Networks

This example shows how to denoise speech signals using deep learning networks. The example compares two types of networks applied to the same task: fully connected, and convolutional.

Introduction

The aim of speech denoising is to remove noise from speech signals while enhancing the quality and intelligibility of speech. This example showcases the removal of washing machine noise from speech signals using deep learning networks. The example compares two types of networks applied to the same task: fully connected, and convolutional.

Problem Summary

Consider the following speech signal sampled at 8 kHz.

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

Add washing machine noise to the speech signal. Set the noise power such that the signal-to-noise ratio (SNR) is zero dB.

```
noise = audioread("WashingMachine-16-8-mono-1000secs.mp3");

% Extract a noise segment from a random location in the noise file
ind = randi(numel(noise) - numel(cleanAudio) + 1, 1, 1);
noiseSegment = noise(ind:ind + numel(cleanAudio) - 1);

speechPower = sum(cleanAudio.^2);
noisePower = sum(noiseSegment.^2);
noisyAudio = cleanAudio + sqrt(speechPower/noisePower) * noiseSegment;
```

Listen to the noisy speech signal.

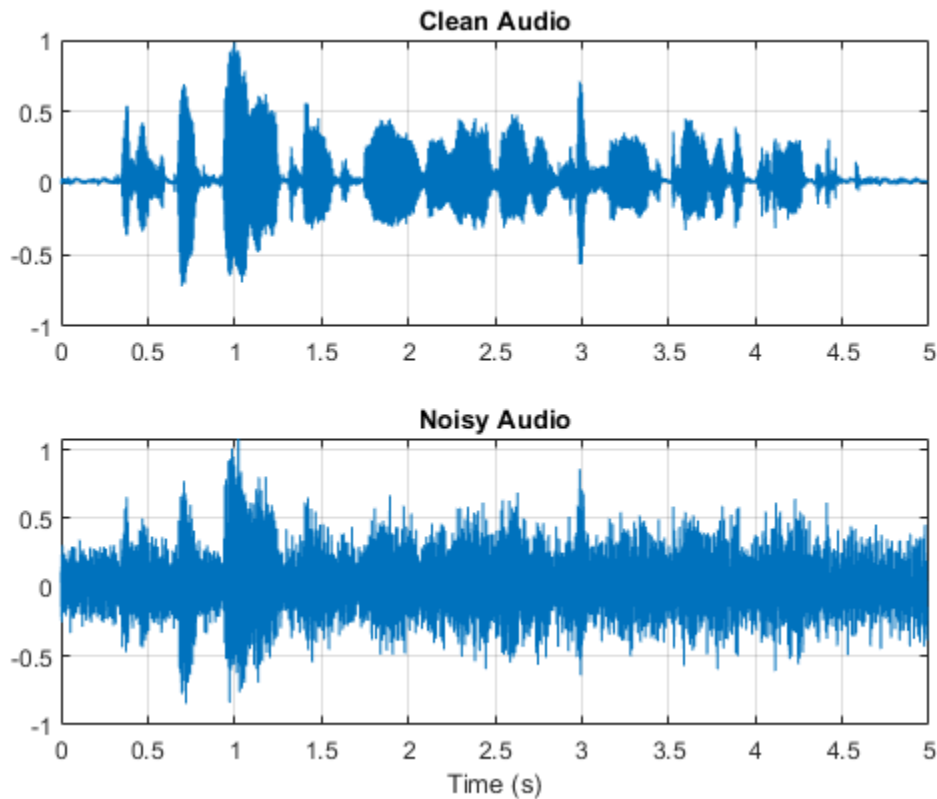
```
sound(noisyAudio,fs)
```

Visualize the original and noisy signals.

```
t = (1/fs) * (0:numel(cleanAudio)-1);

subplot(2,1,1)
plot(t,cleanAudio)
title("Clean Audio")
grid on

subplot(2,1,2)
plot(t,noisyAudio)
title("Noisy Audio")
xlabel("Time (s)")
grid on
```



The objective of speech denoising is to remove the washing machine noise from the speech signal while minimizing undesired artifacts in the output speech.

Examine the Dataset

This example uses a subset of the Mozilla Common Voice dataset [1 on page 1-0] to train and test the deep learning networks. The data set contains 48 kHz recordings of subjects speaking short sentences. Download the data set and unzip the downloaded file.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/commonvoice.zip';
downloadFolder = tempdir;
dataFolder = fullfile(downloadFolder,'commonvoice');

if ~exist(dataFolder,'dir')
    disp('Downloading data set (956 MB) ...')
    unzip(url,downloadFolder)
end
```

Downloading data set (956 MB) ...

Use `audioDatastore` to create a datastore for the training set. To speed up the runtime of the example at the cost of performance, set `reduceDataset` to `true`.

```
adsTrain = audioDatastore(fullfile(dataFolder,'train'),'IncludeSubfolders',true);
```

```
reduceDataset = ;
if reduceDataset
```



```

adsTrain = shuffle(adsTrain);
adsTrain = subset(adsTrain,1:1000);
end

Use read to get the contents of the first file in the datastore.

[audio,adsTrainInfo] = read(adsTrain);

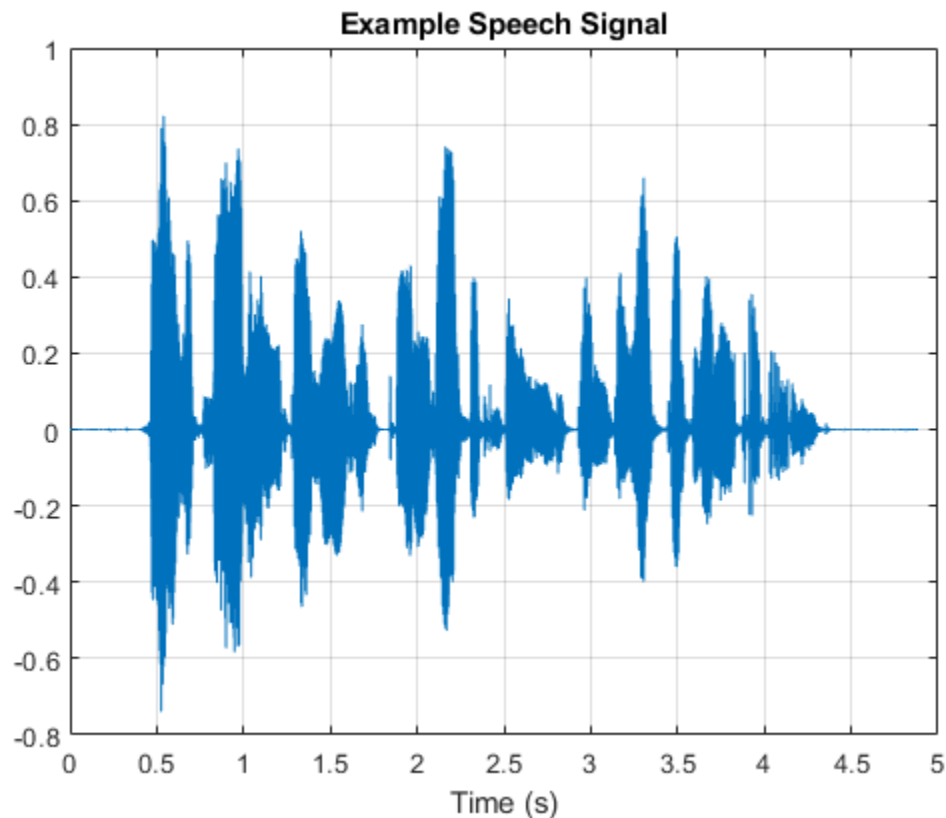
Listen to the speech signal.

sound(audio,adsTrainInfo.SampleRate)

Plot the speech signal.

figure
t = (1/adsTrainInfo.SampleRate) * (0:numel(audio)-1);
plot(t,audio)
title("Example Speech Signal")
xlabel("Time (s)")
grid on

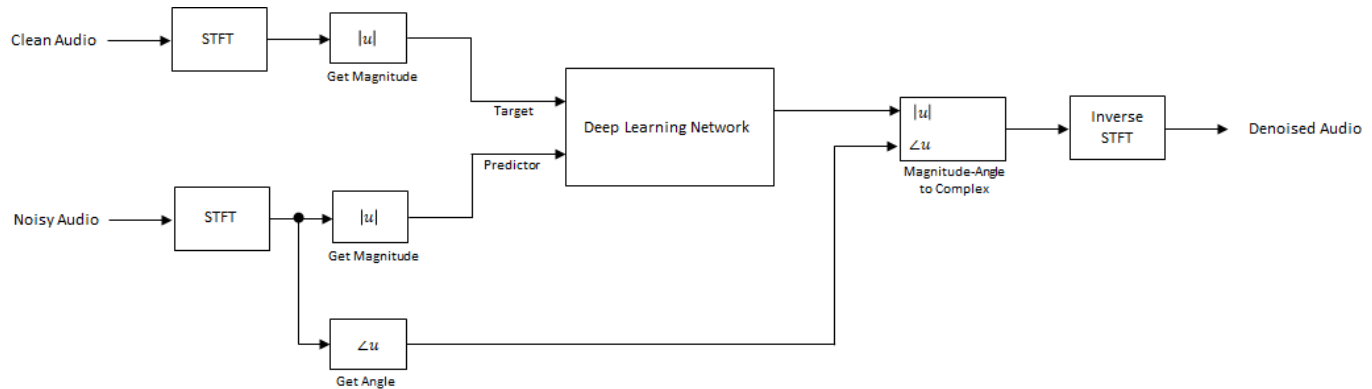
```



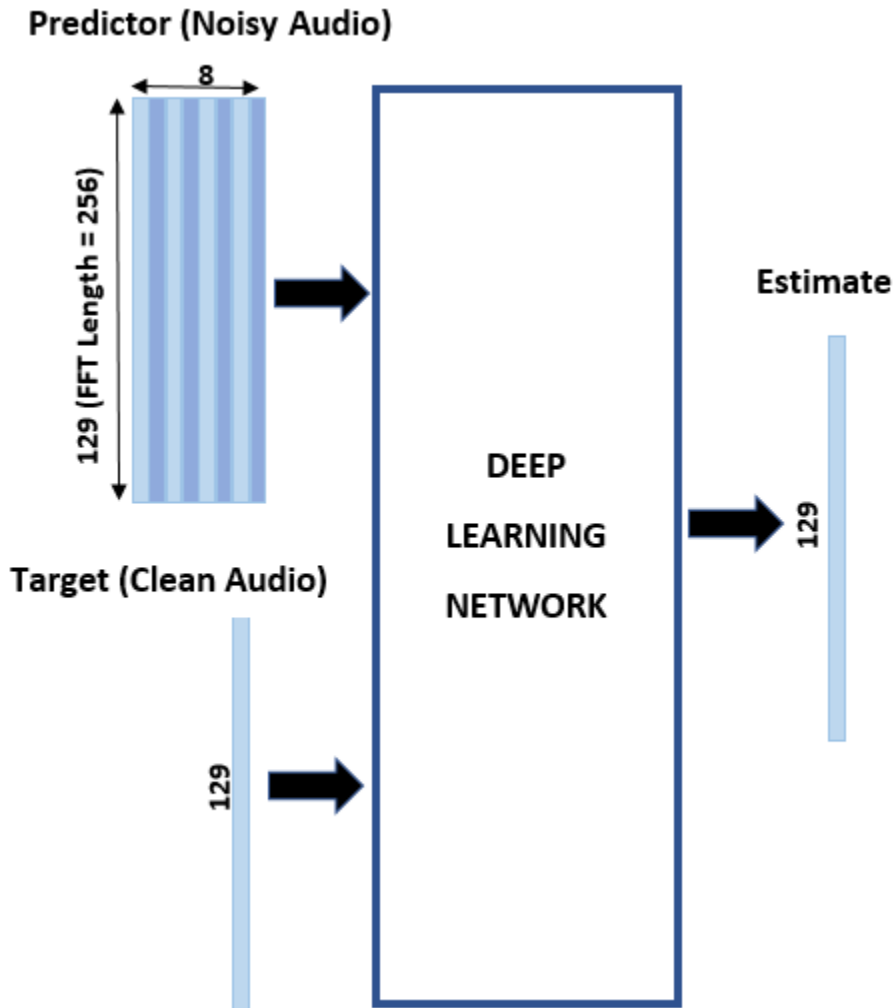
Deep Learning System Overview

The basic deep learning training scheme is shown below. Note that, since speech generally falls below 4 kHz, you first downsample the clean and noisy audio signals to 8 kHz to reduce the computational load of the network. The predictor and target network signals are the magnitude spectra of the noisy and clean audio signals, respectively. The network's output is the magnitude spectrum of the denoised signal. The regression network uses the predictor input to minimize the

mean square error between its output and the input target. The denoised audio is converted back to the time domain using the output magnitude spectrum and the phase of the noisy signal [2 on page 1-0].



You transform the audio to the frequency domain using the Short-Time Fourier transform (STFT), with a window length of 256 samples, an overlap of 75%, and a Hamming window. You reduce the size of the spectral vector to 129 by dropping the frequency samples corresponding to negative frequencies (because the time-domain speech signal is real, this does not lead to any information loss). The predictor input consists of 8 consecutive noisy STFT vectors, so that each STFT output estimate is computed based on the current noisy STFT and the 7 previous noisy STFT vectors.



STFT Targets and Predictors

This section illustrates how to generate the target and predictor signals from one training file.

First, define system parameters:

```
windowLength = 256;
win = hamming(windowLength,"periodic");
overlap = round(0.75 * windowLength);
fftLength = windowLength;
inputFs = 48e3;
fs = 8e3;
numFeatures = fftLength/2 + 1;
numSegments = 8;
```

Create a `dsp.SampleRateConverter` object to convert the 48 kHz audio to 8 kHz.

```
src = dsp.SampleRateConverter("InputSampleRate",inputFs, ...
                             "OutputSampleRate",fs, ...
                             "Bandwidth",7920);
```

Use `read` to get the contents of an audio file from the datastore.

```
audio = read(adsTrain);
```

Make sure the audio length is a multiple of the sample rate converter decimation factor.

```
decimationFactor = inputFs/fs;  
L = floor(numel(audio)/decimationFactor);  
audio = audio(1:decimationFactor*L);
```

Convert the audio signal to 8 kHz.

```
audio = src(audio);  
reset(src)
```

Create a random noise segment from the washing machine noise vector.

```
randind = randi(numel(noise) - numel(audio),[1 1]);  
noiseSegment = noise(randind : randind + numel(audio) - 1);
```

Add noise to the speech signal such that the SNR is 0 dB.

```
noisePower = sum(noiseSegment.^2);  
cleanPower = sum(audio.^2);  
noiseSegment = noiseSegment .* sqrt(cleanPower/noisePower);  
noisyAudio = audio + noiseSegment;
```

Use `stft` to generate magnitude STFT vectors from the original and noisy audio signals.

```
cleanSTFT = stft(audio,'Window',win,'OverlapLength',overlap,'FFTLength',fftLength);  
cleanSTFT = abs(cleanSTFT(numFeatures-1:end,:));  
noisySTFT = stft(noisyAudio,'Window',win,'OverlapLength',overlap,'FFTLength',fftLength);  
noisySTFT = abs(noisySTFT(numFeatures-1:end,:));
```

Generate the 8-segment training predictor signals from the noisy STFT. The overlap between consecutive predictors is 7 segments.

```
noisySTFT = [noisySTFT(:,1:numSegments - 1), noisySTFT];  
stftSegments = zeros(numFeatures, numSegments , size(noisySTFT,2) - numSegments + 1);  
for index = 1:size(noisySTFT,2) - numSegments + 1  
    stftSegments(:, :, index) = (noisySTFT(:, index:index + numSegments - 1));  
end
```

Set the targets and predictors. The last dimension of both variables corresponds to the number of distinct predictor/target pairs generated by the audio file. Each predictor is 129-by-8, and each target is 129-by-1.

```
targets = cleanSTFT;  
size(targets)
```

```
ans = 1×2  
  
129    544
```

```
predictors = stftSegments;  
size(predictors)
```

```
ans = 1×3
```

129 8 544

Extract Features Using Tall Arrays

To speed up processing, extract feature sequences from the speech segments of all audio files in the datastore using tall arrays. Unlike in-memory arrays, tall arrays typically remain unevaluated until you call the `gather` function. This deferred evaluation enables you to work quickly with large data sets. When you eventually request output using `gather`, MATLAB combines the queued calculations where possible and takes the minimum number of passes through the data. If you have Parallel Computing Toolbox™, you can use tall arrays in your local MATLAB session, or on a local parallel pool. You can also run tall array calculations on a cluster if you have MATLAB® Parallel Server™ installed.

First, convert the datastore to a tall array.

```
reset(adsTrain)
T = tall(adsTrain)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
T =
```

```
M×1 tall cell array
```

```
{234480×1 double}
{210288×1 double}
{282864×1 double}
{292080×1 double}
{410736×1 double}
{303600×1 double}
{326640×1 double}
{233328×1 double}
:
:
```

The display indicates that the number of rows (corresponding to the number of files in the datastore), *M*, is not yet known. *M* is a placeholder until the calculation completes.

Extract the target and predictor magnitude STFT from the tall table. This action creates new tall array variables to use in subsequent calculations. The function `HelperGenerateSpeechDenoisingFeatures` performs the steps already highlighted in the STFT Targets and Predictors on page 1-0 section. The `cellfun` command applies `HelperGenerateSpeechDenoisingFeatures` to the contents of each audio file in the datastore.

```
[targets,predictors] = cellfun(@(x)HelperGenerateSpeechDenoisingFeatures(x,noise,src),T,"Uniform",true);
```

Use `gather` to evaluate the targets and predictors.

```
[targets,predictors] = gather(targets,predictors);
```

```
Evaluating tall expression using the Parallel Pool 'local':
```

```
- Pass 1 of 1: Completed in 49 sec
Evaluation completed in 2 min 11 sec
```

It is good practice to normalize all features to zero mean and unity standard deviation.

Compute the mean and standard deviation of the predictors and targets, respectively, and use them to normalize the data.

```
predictors = cat(3,predictors{:});
noisyMean = mean(predictors(:));
noisyStd = std(predictors(:));
predictors(:) = (predictors(:) - noisyMean)/noisyStd;

targets = cat(2,targets{:});
cleanMean = mean(targets(:));
cleanStd = std(targets(:));
targets(:) = (targets(:) - cleanMean)/cleanStd;
```

Reshape predictors and targets to the dimensions expected by the deep learning networks.

```
predictors = reshape(predictors,size(predictors,1),size(predictors,2),1,size(predictors,3));
targets = reshape(targets,1,1,size(targets,1),size(targets,2));
```

You will use 1% of the data for validation during training. Validation is useful to detect scenarios where the network is overfitting the training data.

Randomly split the data into training and validation sets.

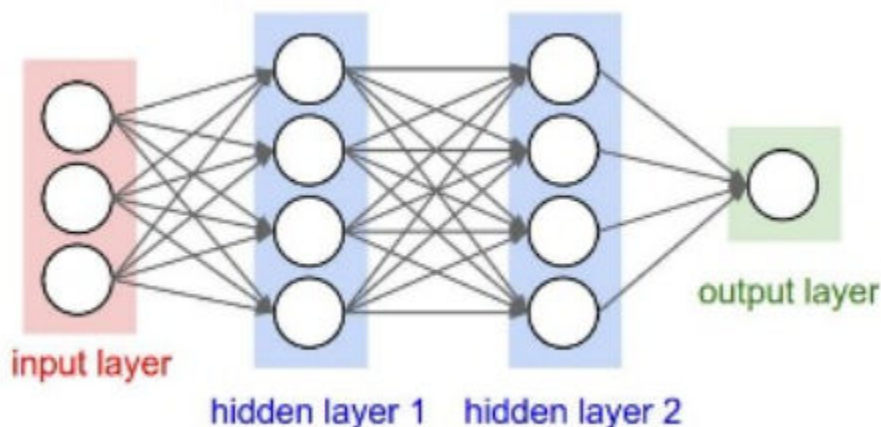
```
inds = randperm(size(predictors,4));
L = round(0.99 * size(predictors,4));

trainPredictors = predictors(:,:,:,inds(1:L));
trainTargets = targets(:,:,:,inds(1:L));

validatePredictors = predictors(:,:,:,inds(L+1:end));
validateTargets = targets(:,:,:,inds(L+1:end));
```

Speech Denoising with Fully Connected Layers

You first consider a denoising network comprised of fully connected layers. Each neuron in a fully connected layer is connected to all activations from the previous layer. A fully connected layer multiplies the input by a weight matrix and then adds a bias vector. The dimensions of the weight matrix and bias vector are determined by the number of neurons in the layer and the number of activations from the previous layer.



Define the layers of the network. Specify the input size to be images of size NumFeatures-by-NumSegments (129-by-8 in this example). Define two hidden fully connected layers, each with 1024 neurons. Since purely linear systems, follow each hidden fully connected layer with a Rectified Linear Unit (ReLU) layer. The batch normalization layers normalize the means and standard deviations of the outputs. Add a fully connected layer with 129 neurons, followed by a regression layer.

```
layers = [
    imageInputLayer([numFeatures,numSegments])
    fullyConnectedLayer(1024)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(1024)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(numFeatures)
    regressionLayer
];
```

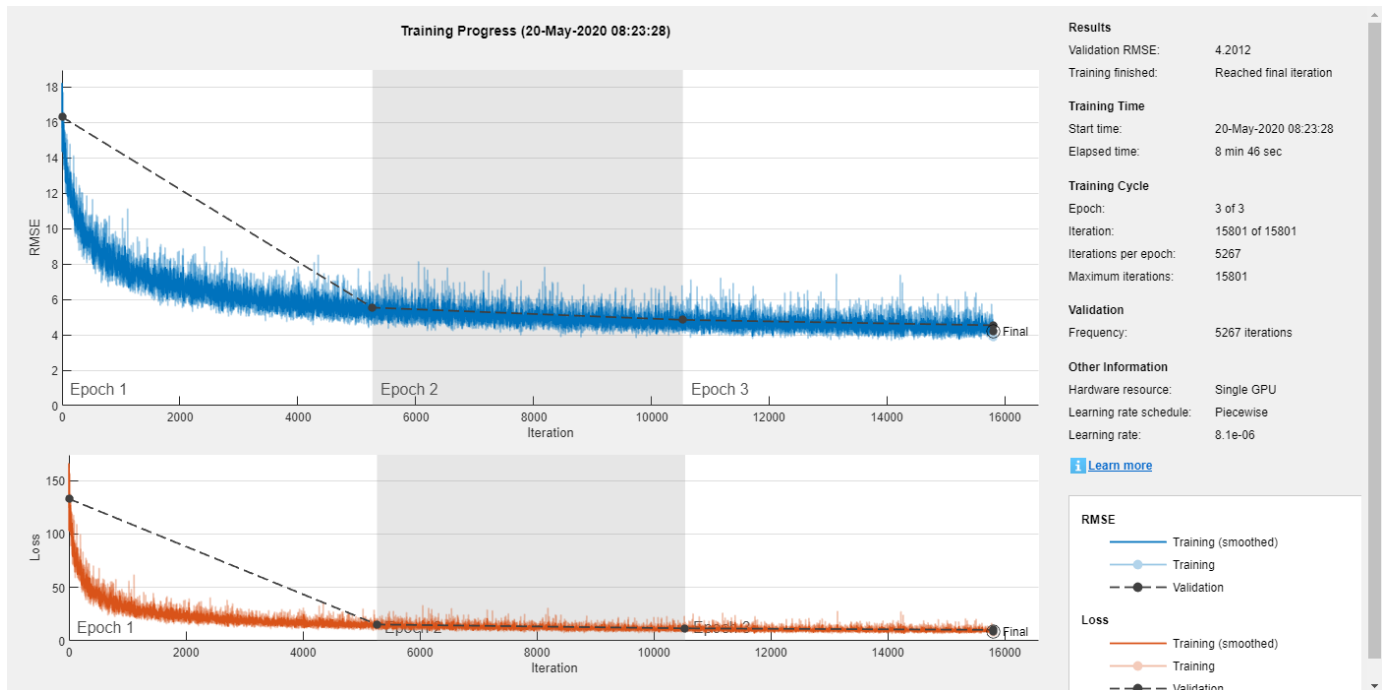
Next, specify the training options for the network. Set MaxEpochs to 3 so that the network makes 3 passes through the training data. Set MiniBatchSize of 128 so that the network looks at 128 training signals at a time. Specify Plots as "training-progress" to generate plots that show the training progress as the number of iterations increases. Set Verbose to false to disable printing the table output that corresponds to the data shown in the plot into the command line window. Specify Shuffle as "every-epoch" to shuffle the training sequences at the beginning of each epoch. Specify LearnRateSchedule to "piecewise" to decrease the learning rate by a specified factor (0.9) every time a certain number of epochs (1) has passed. Set ValidationData to the validation predictors and targets. Set ValidationFrequency such that the validation mean square error is computed once per epoch. This example uses the adaptive moment estimation (Adam) solver.

```
miniBatchSize = 128;
options = trainingOptions("adam", ...
    "MaxEpochs",3, ...
    "InitialLearnRate",1e-5,...
    "MiniBatchSize",miniBatchSize, ...
    "Shuffle","every-epoch", ...
    "Plots","training-progress", ...
    "Verbose",false, ...
    "ValidationFrequency",floor(size(trainPredictors,4)/miniBatchSize), ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.9, ...
    "LearnRateDropPeriod",1, ...
    "ValidationData",{validatePredictors,validateTargets});
```

Train the network with the specified training options and layer architecture using trainNetwork. Because the training set is large, the training process can take several minutes. To load a pre-trained network instead of training a network from scratch, set doTraining to false.

```
doTraining = ☐;
if doTraining
    denoiseNetFullyConnected = trainNetwork(trainPredictors,trainTargets,layers,options);
else
    s = load("denoisenet.mat");
    denoiseNetFullyConnected = s.denoiseNetFullyConnected;
    cleanMean = s.cleanMean;
    cleanStd = s.cleanStd;
    noisyMean = s.noisyMean;
```

```
noisyStd = s.noisyStd;
end
```



Count the number of weights in the fully connected layers of the network.

```
numWeights = 0;
for index = 1:numel(denoiseNetFullyConnected.Layers)
    if isa(denoiseNetFullyConnected.Layers(index),"nnet.cnn.layer.FullyConnectedLayer")
        numWeights = numWeights + numel(denoiseNetFullyConnected.Layers(index).Weights);
    end
end
fprintf("The number of weights is %d.\n",numWeights);
```

The number of weights is 2237440.

Speech Denoising with Convolutional Layers

Consider a network that uses convolutional layers instead of fully connected layers [3 on page 1-0]. A 2-D convolutional layer applies sliding filters to the input. The layer convolves the input by moving the filters along the input vertically and horizontally and computing the dot product of the weights and the input, and then adding a bias term. Convolutional layers typically consist of fewer parameters than fully connected layers.

Define the layers of the fully convolutional network described in [3 on page 1-0], comprising 16 convolutional layers. The first 15 convolutional layers are groups of 3 layers, repeated 5 times, with filter widths of 9, 5, and 9, and number of filters of 18, 30 and 8, respectively. The last convolutional layer has a filter width of 129 and 1 filter. In this network, convolutions are performed in only one direction (along the frequency dimension), and the filter width along the time dimension is set to 1 for all layers except the first one. Similar to the fully connected network, convolutional layers are followed by ReLu and batch normalization layers.

```
layers = [imageInputLayer([numFeatures,numSegments])
    convolution2dLayer([9 8],18,"Stride",[1 100],"Padding","same")
```



```

batchNormalizationLayer
reluLayer

repmat( ...
[convolution2dLayer([5 1],30,"Stride",[1 100],"Padding","same")
batchNormalizationLayer
reluLayer
convolution2dLayer([9 1],8,"Stride",[1 100],"Padding","same")
batchNormalizationLayer
reluLayer
convolution2dLayer([9 1],18,"Stride",[1 100],"Padding","same")
batchNormalizationLayer
reluLayer],4,1)

convolution2dLayer([5 1],30,"Stride",[1 100],"Padding","same")
batchNormalizationLayer
reluLayer
convolution2dLayer([9 1],8,"Stride",[1 100],"Padding","same")
batchNormalizationLayer
reluLayer

convolution2dLayer([129 1],1,"Stride",[1 100],"Padding","same")

regressionLayer
];

```

The training options are identical to the options for the fully connected network, except that the dimensions of the validation target signals are permuted to be consistent with the dimensions expected by the regression layer.

```

options = trainingOptions("adam", ...
    "MaxEpochs",3, ...
    "InitialLearnRate",1e-5, ...
    "MiniBatchSize",miniBatchSize, ...
    "Shuffle","every-epoch", ...
    "Plots","training-progress", ...
    "Verbose",false, ...
    "ValidationFrequency",floor(size(trainPredictors,4)/miniBatchSize), ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.9, ...
    "LearnRateDropPeriod",1, ...
    "ValidationData",{validatePredictors,permute(validateTargets,[3 1 2 4])});

```

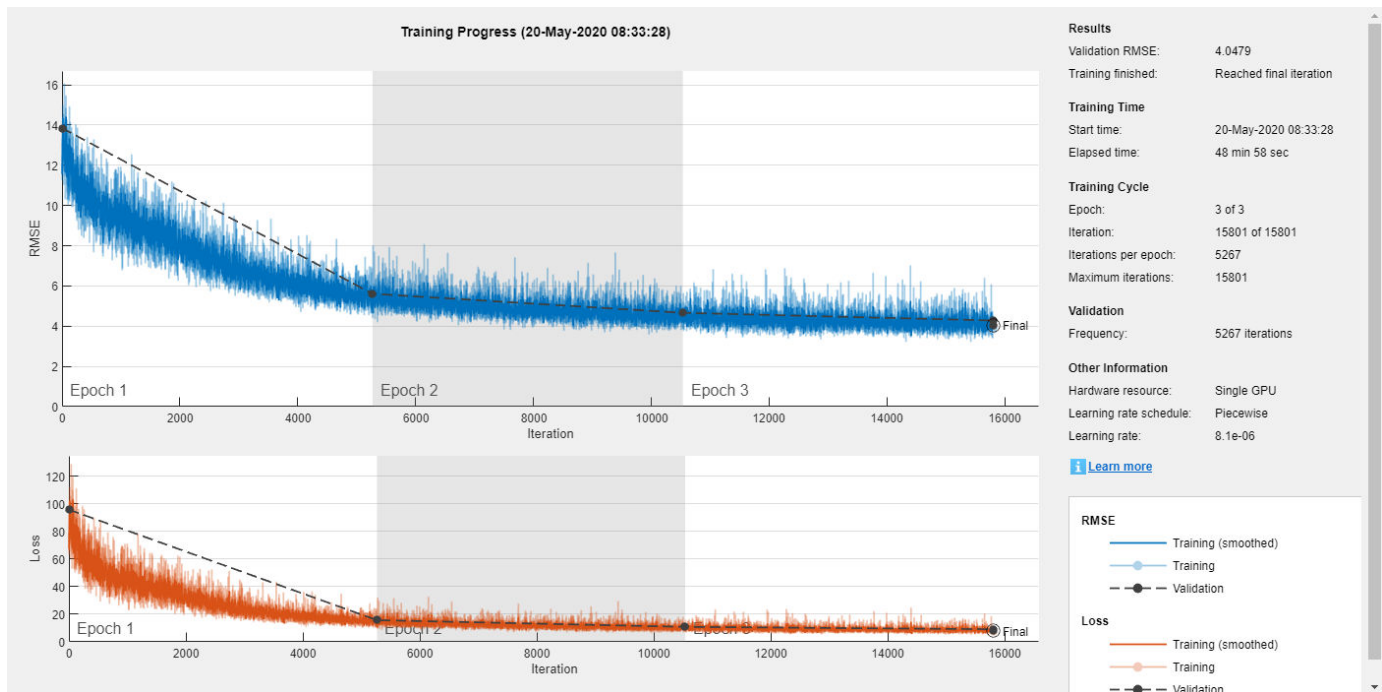
Train the network with the specified training options and layer architecture using `trainNetwork`. Because the training set is large, the training process can take several minutes. To load a pre-trained network instead of training a network from scratch, set `doTraining` to false.

```

doTraining = ☐ true ;
if doTraining
    denoiseNetFullyConvolutional = trainNetwork(trainPredictors,permute(trainTargets,[3 1 2 4]),options);
else
    s = load("denoisenet.mat");
    denoiseNetFullyConvolutional = s.denoiseNetFullyConvolutional;
    cleanMean = s.cleanMean;
    cleanStd = s.cleanStd;
    noisyMean = s.noisyMean;
end

```

```
noisyStd = s.noisyStd;
end
```



Count the number of weights in the fully connected layers of the network.

```
numWeights = 0;
for index = 1:numel(denoiseNetFullyConvolutional.Layers)
    if isa(denoiseNetFullyConvolutional.Layers(index), "nnet.cnn.layer.Convolution2DLayer")
        numWeights = numWeights + numel(denoiseNetFullyConvolutional.Layers(index).Weights);
    end
end
fprintf("The number of weights in convolutional layers is %d\n", numWeights);
```

The number of weights in convolutional layers is 31812

Test the Denoising Networks

Read in the test data set.

```
adsTest = audioDatastore(fullfile(dataFolder, 'test'), 'IncludeSubfolders', true);
```

Read the contents of a file from the datastore.

```
[cleanAudio, adsTestInfo] = read(adsTest);
```

Make sure the audio length is a multiple of the sample rate converter decimation factor.

```
L = floor(numel(cleanAudio)/decimationFactor);
cleanAudio = cleanAudio(1:decimationFactor*L);
```

Convert the audio signal to 8 kHz.

```
cleanAudio = src(cleanAudio);
reset(src)
```

In this testing stage, you corrupt speech with washing machine noise not used in the training stage.

```
noise = audioread("WashingMachine-16-8-mono-200secs.mp3");
```

Create a random noise segment from the washing machine noise vector.

```
randind = randi(numel(noise) - numel(cleanAudio), [1 1]);
noiseSegment = noise(randind : randind + numel(cleanAudio) - 1);
```

Add noise to the speech signal such that the SNR is 0 dB.

```
noisePower = sum(noiseSegment.^2);
cleanPower = sum(cleanAudio.^2);
noiseSegment = noiseSegment .* sqrt(cleanPower/noisePower);
noisyAudio = cleanAudio + noiseSegment;
```

Use `stft` to generate magnitude STFT vectors from the noisy audio signals.

```
noisySTFT = stft(noisyAudio, 'Window', win, 'OverlapLength', overlap, 'FFTLength', fftLength);
noisyPhase = angle(noisySTFT(numFeatures-1:end,:));
noisySTFT = abs(noisySTFT(numFeatures-1:end,:));
```

Generate the 8-segment training predictor signals from the noisy STFT. The overlap between consecutive predictors is 7 segments.

```
noisySTFT = [noisySTFT(:,1:numSegments-1) noisySTFT];
predictors = zeros( numFeatures, numSegments , size(noisySTFT,2) - numSegments + 1);
for index = 1:(size(noisySTFT,2) - numSegments + 1)
    predictors(:, :, index) = noisySTFT(:, index:index + numSegments - 1);
end
```

Normalize the predictors by the mean and standard deviation computed in the training stage.

```
predictors(:) = (predictors(:) - noisyMean) / noisyStd;
```

Compute the denoised magnitude STFT by using `predict` with the two trained networks.

```
predictors = reshape(predictors, [numFeatures, numSegments, 1, size(predictors, 3)]);
STFTFullyConnected = predict(denoiseNetFullyConnected, predictors);
STFTFullyConvolutional = predict(denoiseNetFullyConvolutional, predictors);
```

Scale the outputs by the mean and standard deviation used in the training stage.

```
STFTFullyConnected(:) = cleanStd * STFTFullyConnected(:) + cleanMean;
STFTFullyConvolutional(:) = cleanStd * STFTFullyConvolutional(:) + cleanMean;
```

Convert the one-sided STFT to a centered STFT.

```
STFTFullyConnected = STFTFullyConnected.' .* exp(1j*noisyPhase);
STFTFullyConnected = [conj(STFTFullyConnected(end-1:-1:2,:)); STFTFullyConnected];
STFTFullyConvolutional = squeeze(STFTFullyConvolutional) .* exp(1j*noisyPhase);
STFTFullyConvolutional = [conj(STFTFullyConvolutional(end-1:-1:2,:)) ; STFTFullyConvolutional];
```

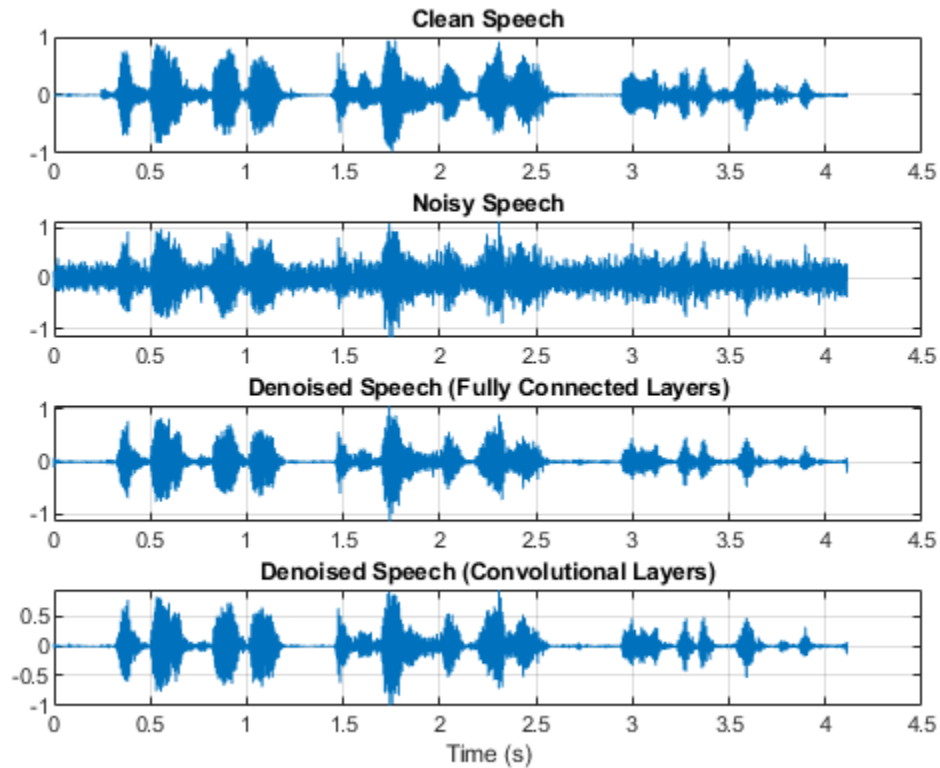
Compute the denoised speech signals. `istft` performs the inverse STFT. Use the phase of the noisy STFT vectors to reconstruct the time-domain signal.

```
denoisedAudioFullyConnected = istft(STFTFullyConnected, ...
    'Window', win, 'OverlapLength', overlap, ...
    'FFTLength', fftLength, 'ConjugateSymmetric', true);
```

```
denoisedAudioFullyConvolutional = istft(STFTFullyConvolutional, ...  
                                       'Window',win,'OverlapLength',overlap, ...  
                                       'FFTLength',fftLength,'ConjugateSymmetric',true);
```

Plot the clean, noisy and denoised audio signals.

```
t = (1/fs) * (0:numel(denoisedAudioFullyConnected)-1);  
  
figure  
  
subplot(4,1,1)  
plot(t, cleanAudio(1:numel(denoisedAudioFullyConnected)))  
title("Clean Speech")  
grid on  
  
subplot(4,1,2)  
plot(t, noisyAudio(1:numel(denoisedAudioFullyConnected)))  
title("Noisy Speech")  
grid on  
  
subplot(4,1,3)  
plot(t, denoisedAudioFullyConnected)  
title("Denoised Speech (Fully Connected Layers)")  
grid on  
  
subplot(4,1,4)  
plot(t, denoisedAudioFullyConvolutional)  
title("Denoised Speech (Convolutional Layers)")  
grid on  
xlabel("Time (s)")
```



Plot the clean, noisy, and denoised spectrograms.

```
h = figure;

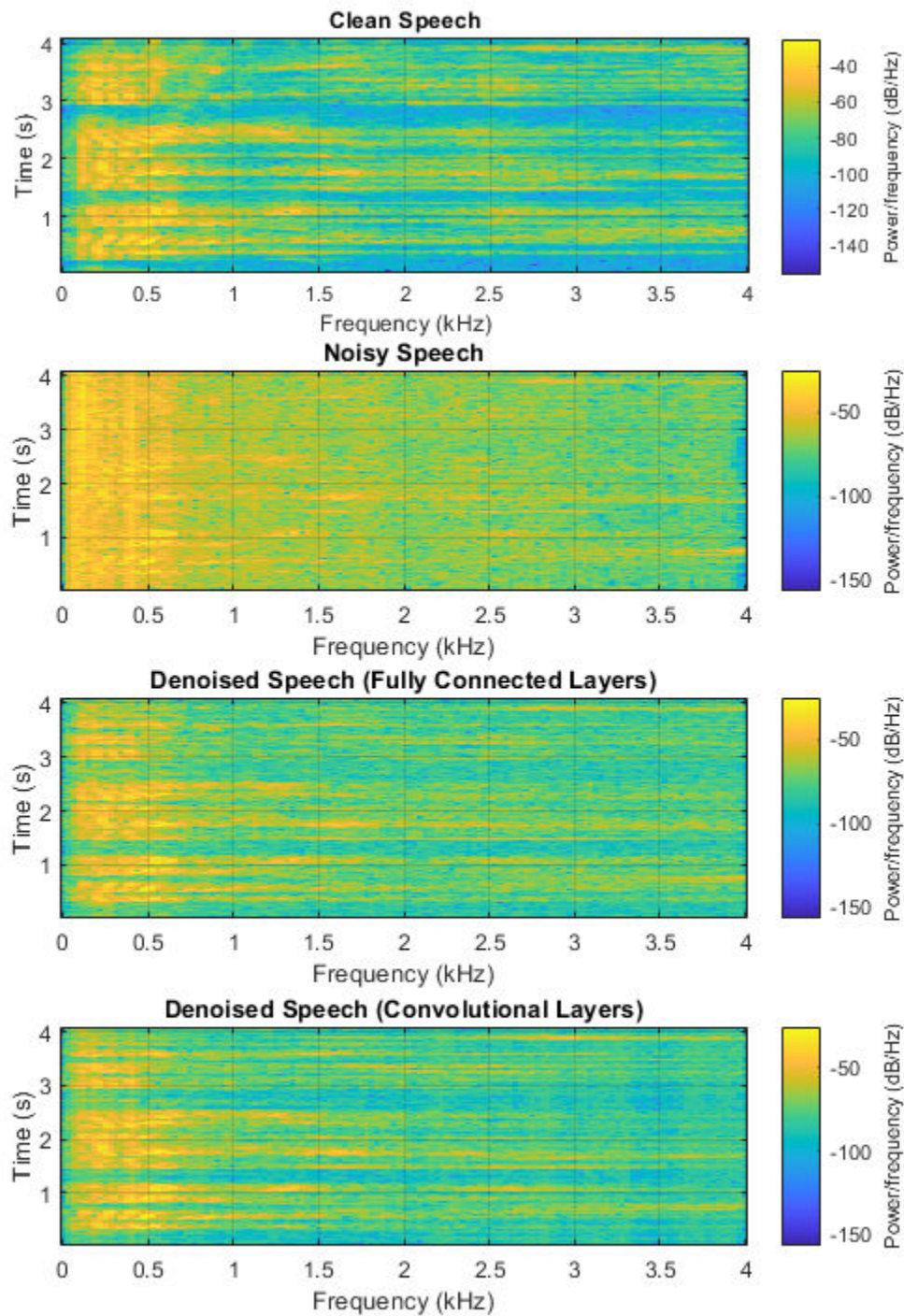
subplot(4,1,1)
spectrogram(cleanAudio,win,overlap,fftLength,fs);
title("Clean Speech")
grid on

subplot(4,1,2)
spectrogram(noisyAudio,win,overlap,fftLength,fs);
title("Noisy Speech")
grid on

subplot(4,1,3)
spectrogram(denoisedAudioFullyConnected,win,overlap,fftLength,fs);
title("Denoised Speech (Fully Connected Layers)")
grid on

subplot(4,1,4)
spectrogram(denoisedAudioFullyConvolutional,win,overlap,fftLength,fs);
title("Denoised Speech (Convolutional Layers)")
grid on

p = get(h,'Position');
set(h,'Position',[p(1) 65 p(3) 800]);
```



Listen to the noisy speech.

```
sound(noisyAudio,fs)
```

Listen to the denoised speech from the network with fully connected layers.

```
sound(denoisedAudioFullyConnected,fs)
```

Listen to the denoised speech from the network with convolutional layers.

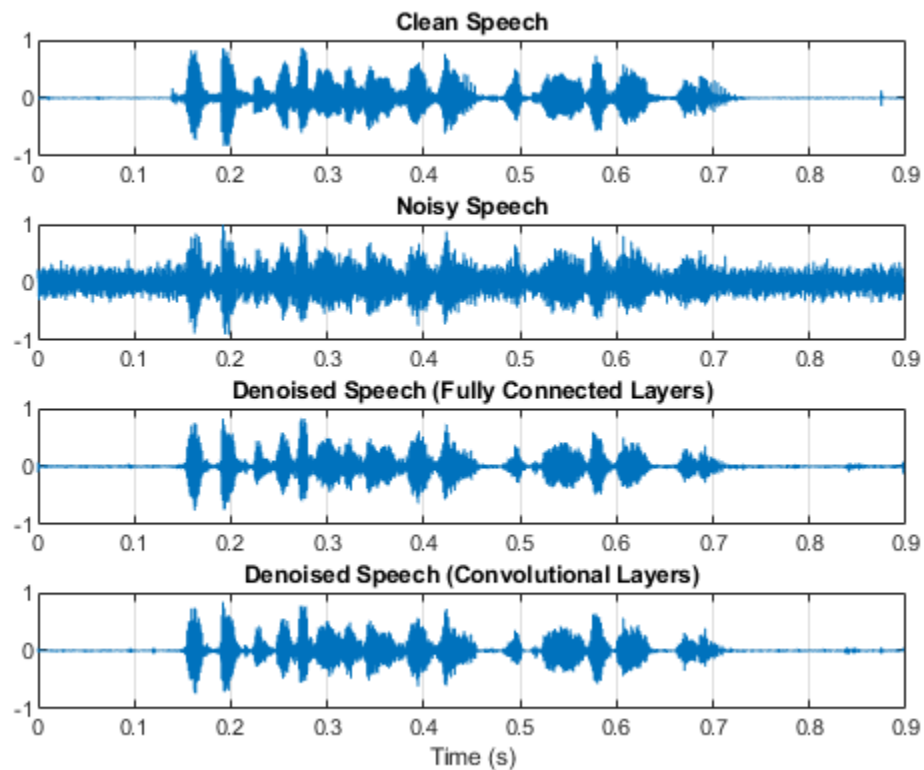
```
sound(denoisedAudioFullyConvolutional,fs)
```

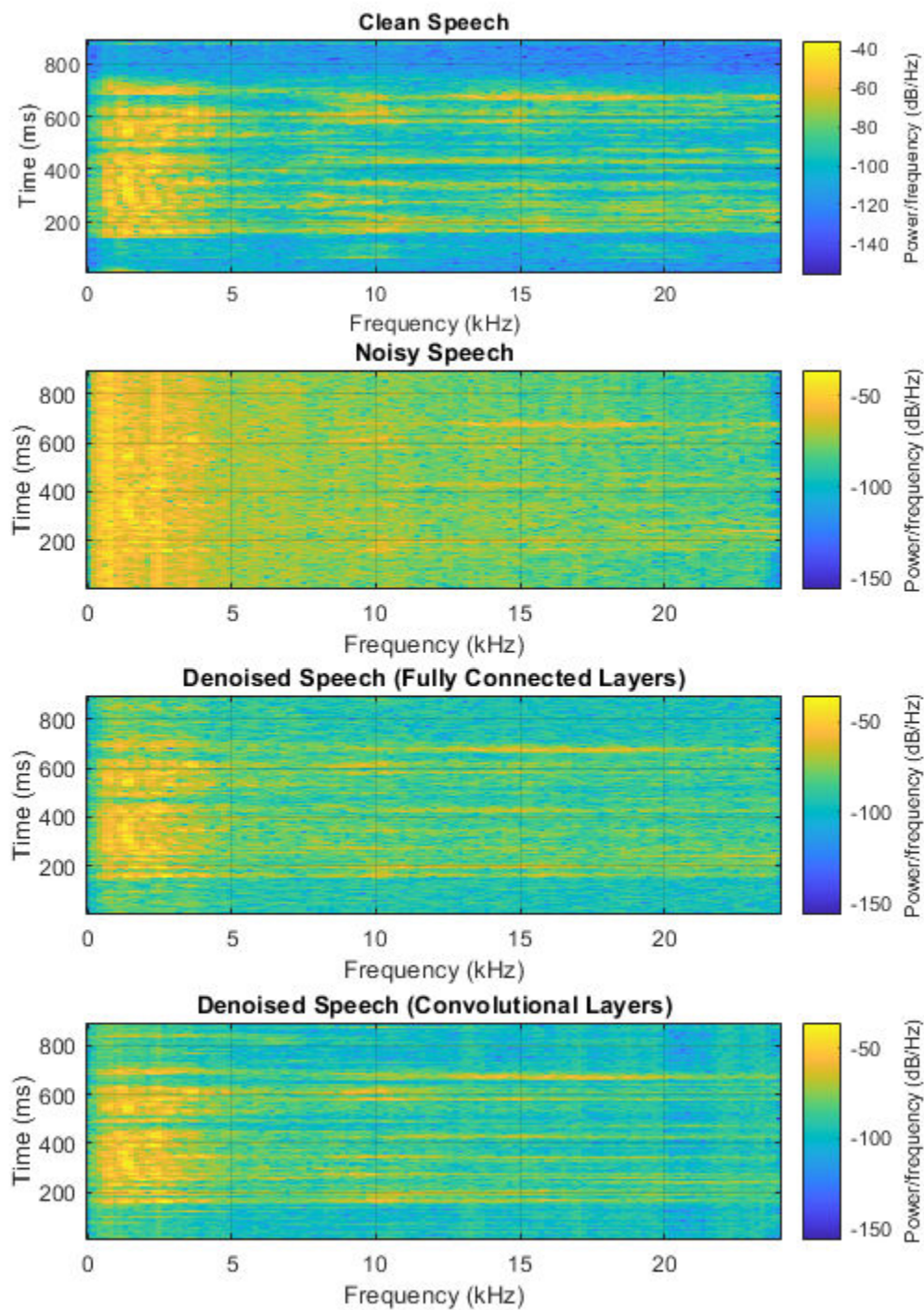
Listen to clean speech.

```
sound(cleanAudio,fs)
```

You can test more files from the datastore by calling `testDenoisingNets`. The function produces the time-domain and frequency-domain plots highlighted above, and also returns the clean, noisy, and denoised audio signals.

```
[cleanAudio,noisyAudio,denoisedAudioFullyConnected,denoisedAudioFullyConvolutional] = testDenoisingNets
```



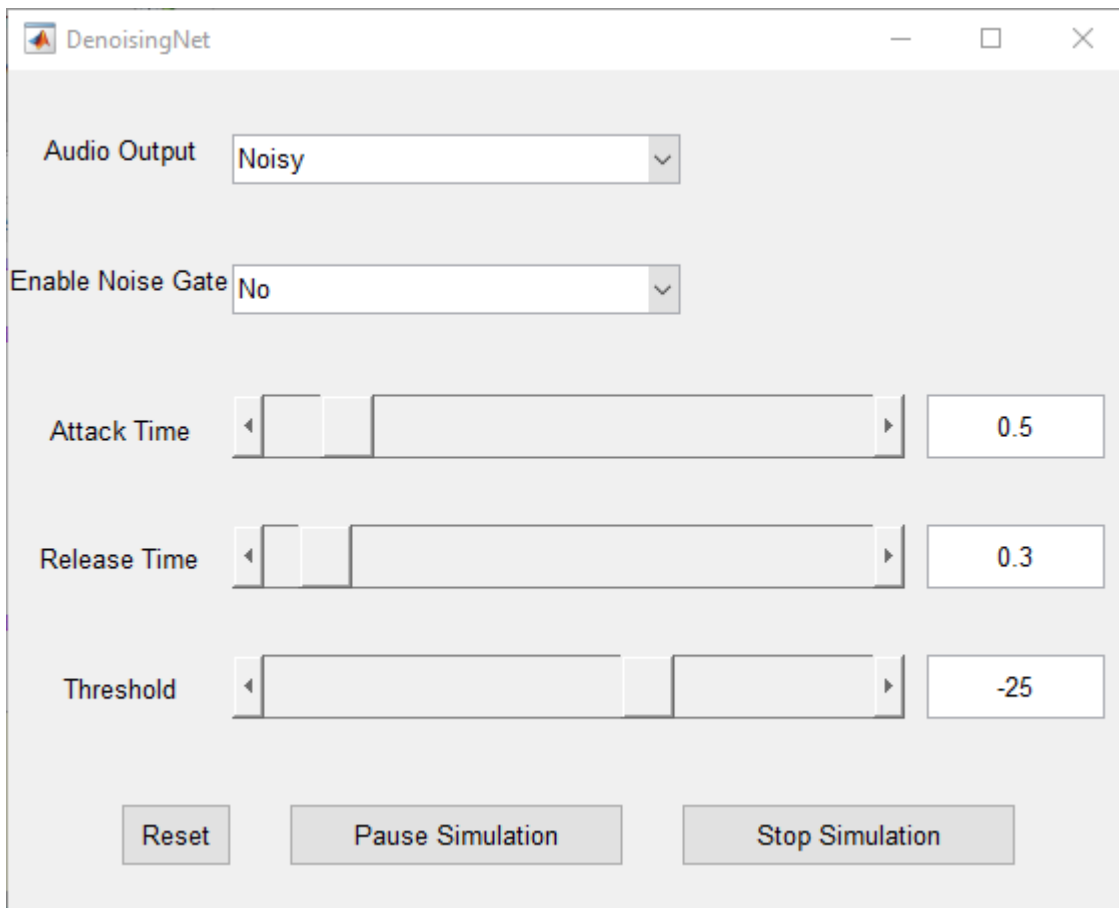


Real-Time Application

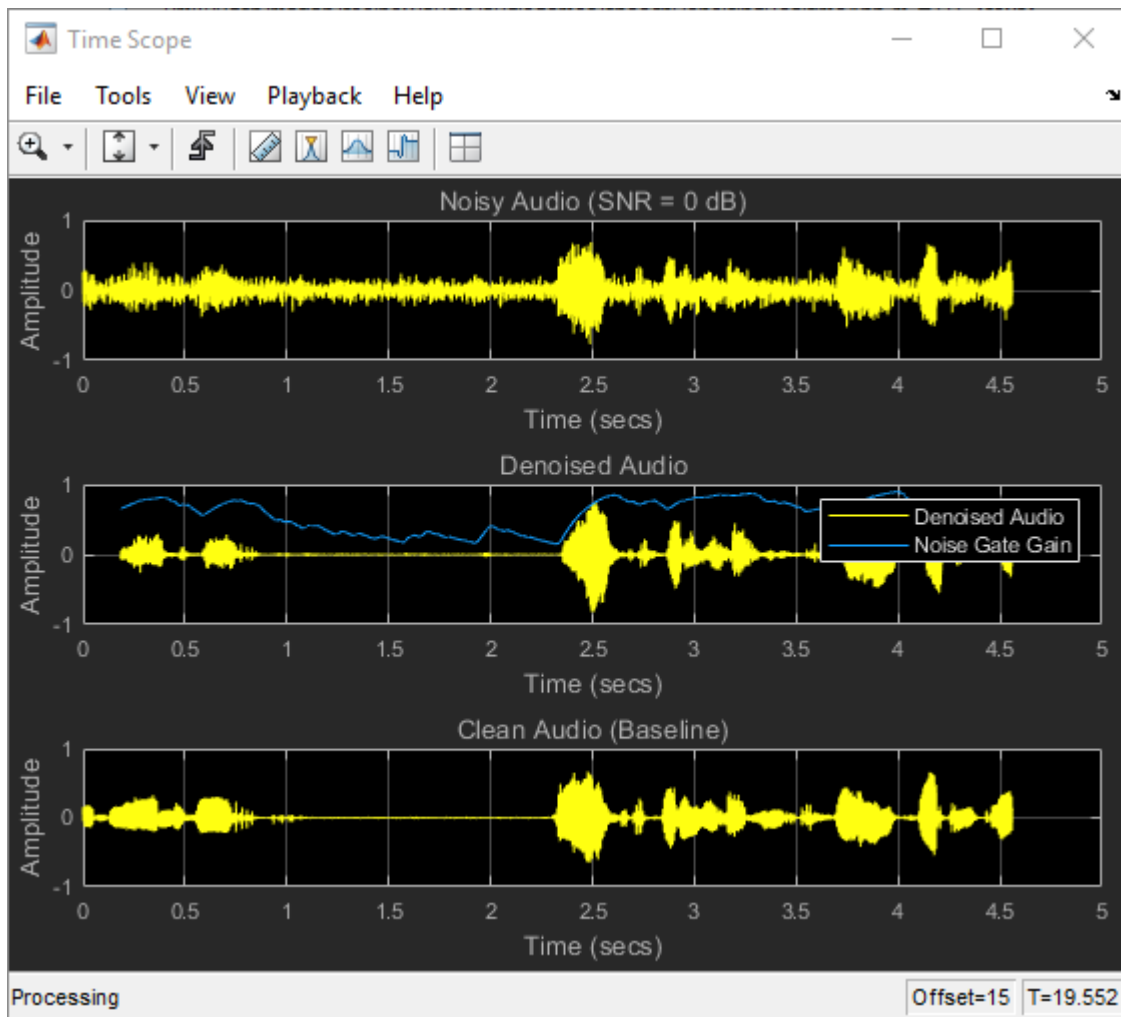
The procedure in the previous section passes the entire spectrum of the noisy signal to `predict`. This is not suitable for real-time applications where low latency is a requirement.

Run `speechDenoisingRealtimeApp` for an example of how to simulate a streaming, real-time version of the denoising network. The app uses the network with fully connected layers. The audio frame length is equal to the STFT hop size, which is $0.25 * 256 = 64$ samples.

`speechDenoisingRealtimeApp` launches a User Interface (UI) designed to interact with the simulation. The UI enables you to tune parameters and the results are reflected in the simulation instantly. You can also enable/disable a noise gate that operates on the denoised output to further reduce the noise, as well as tune the attack time, release time, and threshold of the noise gate. You can listen to the noisy, clean or denoised audio from the UI.



The scope plots the clean, noisy and denoised signals, as well as the gain of the noise gate.



References

- [1] <https://voice.mozilla.org/en>
- [2] "Experiments on Deep Learning for Speech Denoising", Ding Liu, Paris Smaragdis, Minje Kim, INTERSPEECH, 2014.
- [3] "A Fully Convolutional Neural Network for Speech Enhancement", Se Rim Park, Jin Won Lee, INTERSPEECH, 2017.

Classify Gender Using LSTM Networks

This example shows how to classify the gender of a speaker using deep learning. The example uses a Bidirectional Long Short-Term Memory (BiLSTM) network and Gammatone Cepstral Coefficients (gtcc), pitch, harmonic ratio, and several spectral shape descriptors.

Introduction

Gender classification based on speech signals is an essential component of many audio systems, such as automatic speech recognition, speaker recognition, and content-based multimedia indexing.

This example uses long short-term memory (LSTM) networks, a type of recurrent neural network (RNN) well-suited to study sequence and time-series data. An LSTM network can learn long-term dependencies between time steps of a sequence. An LSTM layer (`lstmLayer` (Deep Learning Toolbox)) can look at the time sequence in the forward direction, while a bidirectional LSTM layer (`biLstmLayer` (Deep Learning Toolbox)) can look at the time sequence in both forward and backward directions. This example uses bidirectional LSTM layers.

This example trains the LSTM network with sequences of gammatone cepstrum coefficients (gtcc), pitch estimates (pitch), harmonic ratio (harmonicRatio), and several spectral shape descriptors ("Spectral Descriptors" on page 20-2).

To accelerate the training process, run this example on a machine with a GPU. If your machine has a GPU and Parallel Computing Toolbox™, then MATLAB® automatically uses the GPU for training; otherwise, it uses the CPU.

Classify Gender with a Pre-Trained Network

Before going into the training process in detail, you will use a pre-trained network to classify the gender of the speaker in two test signals.

Load the pre-trained network along with pre-computed vectors used for feature normalization.

```
load('genderIDNet.mat', 'genderIDNet', 'M', 'S');
```

Load a test signal with a male speaker.

```
[audioIn, Fs] = audioread('maleSpeech.flac');
sound(audioIn, Fs)
```

Isolate the speech area in the signal.

```
boundaries = detectSpeech(audioIn, Fs);
audioIn = audioIn(boundaries(1):boundaries(2));
```

Create an `audioFeatureExtractor` to extract features from the audio data. You will use the same object to extract features for training.

```
extractor = audioFeatureExtractor( ...
    "SampleRate",Fs, ...
    "Window",hamming(round(0.03*Fs),"periodic"), ...
    "OverlapLength",round(0.02*Fs), ...
    ...
    "gtcc",true, ...
    "gtccDelta",true, ...
    "gtccDeltaDelta",true, ...
```

```
...
"SpectralDescriptorInput","melSpectrum", ...
"spectralCentroid",true, ...
"spectralEntropy",true, ...
"spectralFlux",true, ...
"spectralSlope",true, ...
...
"pitch",true, ...
"harmonicRatio",true);
```

Extract features from the signal and normalize them.

```
features = extract(extractor, audioIn);
features = (features.' - M)./S;
```

Classify the signal.

```
gender = classify(genderIDNet, features)

gender = categorical
      male
```

Classify another signal with a female speaker.

```
[audioIn, Fs] = audioread('femaleSpeech.flac');
sound(audioIn, Fs)

boundaries = detectSpeech(audioIn, Fs);
audioIn = audioIn(boundaries(1):boundaries(2));

features = extract(extractor, audioIn);
features = (features.' - M)./S;

classify(genderIDNet, features)

ans = categorical
      female
```

Preprocess Training Audio Data

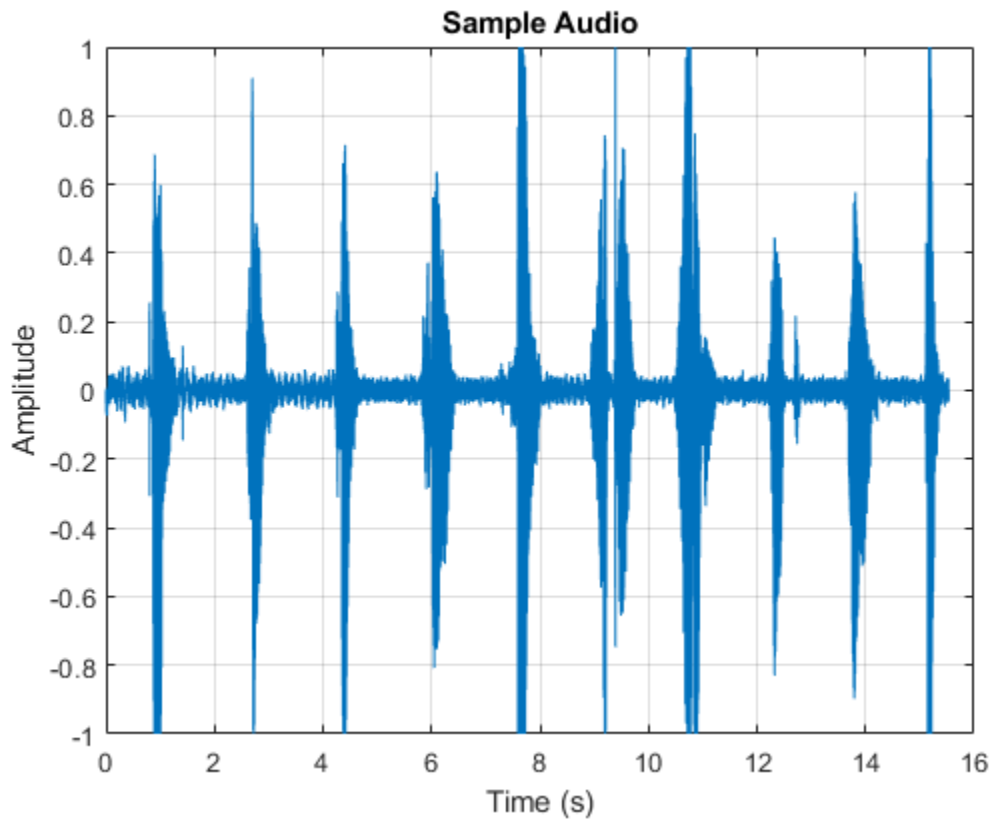
The BiLSTM network used in this example works best when using sequences of feature vectors. To illustrate the preprocessing pipeline, this example walks through the steps for a single audio file.

Read the contents of an audio file containing speech. The speaker gender is male.

```
[audioIn,Fs] = audioread('Counting-16-44p1-mono-15secs.wav');
labels = {'male'};
```

Plot the audio signal and then listen to it using the `sound` command.

```
timeVector = (1/Fs) * (0:size(audioIn,1)-1);
figure
plot(timeVector,audioIn)
ylabel("Amplitude")
xlabel("Time (s)")
title("Sample Audio")
grid on
```



```
sound(audioIn,Fs)
```

The speech signal has silence segments that do not contain useful information pertaining to the gender of the speaker. Use `detectSpeech` to locate segments of speech in the audio signal.

```
speechIndices = detectSpeech(audioIn,Fs);
```

Create an `audioFeatureExtractor` to extract features from the audio data. A speech signal is dynamic in nature and changes over time. It is assumed that speech signals are stationary on short time scales and their processing is often done in windows of 20-40 ms. Specify 30 ms windows with 20 ms overlap.

```
extractor = audioFeatureExtractor( ...
    "SampleRate",Fs, ...
    "Window",hamming(round(0.03*Fs),"periodic"), ...
    "OverlapLength",round(0.02*Fs), ...
    ...
    "gtcc",true, ...
    "gtccDelta",true, ...
    "gtccDeltaDelta",true, ...
    ...
    "SpectralDescriptorInput","melSpectrum", ...
    "spectralCentroid",true, ...
    "spectralEntropy",true, ...
    "spectralFlux",true, ...
    "spectralSlope",true, ...
    ...)
```

```
"pitch",true, ...  
"harmonicRatio",true);
```

Extract features from each audio segment. The output from `audioFeatureExtractor` is a `numFeatureVectors-by-numFeatures` array. The `sequenceInputLayer` used in this example requires time to be along the second dimension. Permute the output array so that time is along the second dimension.

```
featureVectorsSegment = {};  
for ii = 1:size(speechIndices,1)  
    featureVectorsSegment{end+1} = ( extract(extractor,audioIn(speechIndices(ii,1):speechIndices  
end  
numSegments = size(featureVectorsSegment)
```

```
numSegments = 1×2
```

```
1    11
```

```
[numFeatures,numFeatureVectorsSegment1] = size(featureVectorsSegment{1})
```

```
numFeatures = 45
```

```
numFeatureVectorsSegment1 = 124
```

Replicate the labels so that they are in one-to-one correspondence with segments.

```
labels = repelem(labels,size(speechIndices,1))
```

```
labels = 1×11 cell
```

```
    {'male'}    {'male'}    {'male'}    {'male'}    {'male'}    {'male'}    {'male'}    {'male'}
```

When using a `sequenceInputLayer`, it is often advantageous to use sequences of consistent length. Convert the arrays of feature vectors into sequences of feature vectors. Use 20 feature vectors per sequence with 5 feature vector overlap.

```
featureVectorsPerSequence = 20;  
featureVectorOverlap = 5;  
hopLength = featureVectorsPerSequence - featureVectorOverlap;
```

```
idx1 = 1;
```

```
featuresTrain = {};
```

```
sequencePerSegment = zeros(numel(featureVectorsSegment),1);
```

```
for ii = 1:numel(featureVectorsSegment)
```

```
    sequencePerSegment(ii) = max(floor((size(featureVectorsSegment{ii},2) - featureVectorsPerSequ
```

```
    idx2 = 1;
```

```
    for j = 1:sequencePerSegment(ii)
```

```
        featuresTrain{idx1,1} = featureVectorsSegment{ii}{:,idx2:idx2 + featureVectorsPerSequenc
```

```
        idx1 = idx1 + 1;
```

```
        idx2 = idx2 + hopLength;
```

```
    end
```

```
end
```

For conciseness, the helper function `HelperFeatureVector2Sequence` on page 1-0 encapsulates the above processing and is used throughout the rest of the example.

Replicate the labels so that they are in one-to-one correspondence with the training set.

```
labels = repelem(labels,sequencePerSegment);
```

The result of the preprocessing pipeline is a NumSequence-by-1 cell array of NumFeatures-by-FeatureVectorsPerSequence matrices. Labels is a NumSequence-by-1 array.

```
NumSequence = numel(featuresTrain)
```

```
NumSequence = 27
```

```
[NumFeatures,FeatureVectorsPerSequence] = size(featuresTrain{1})
```

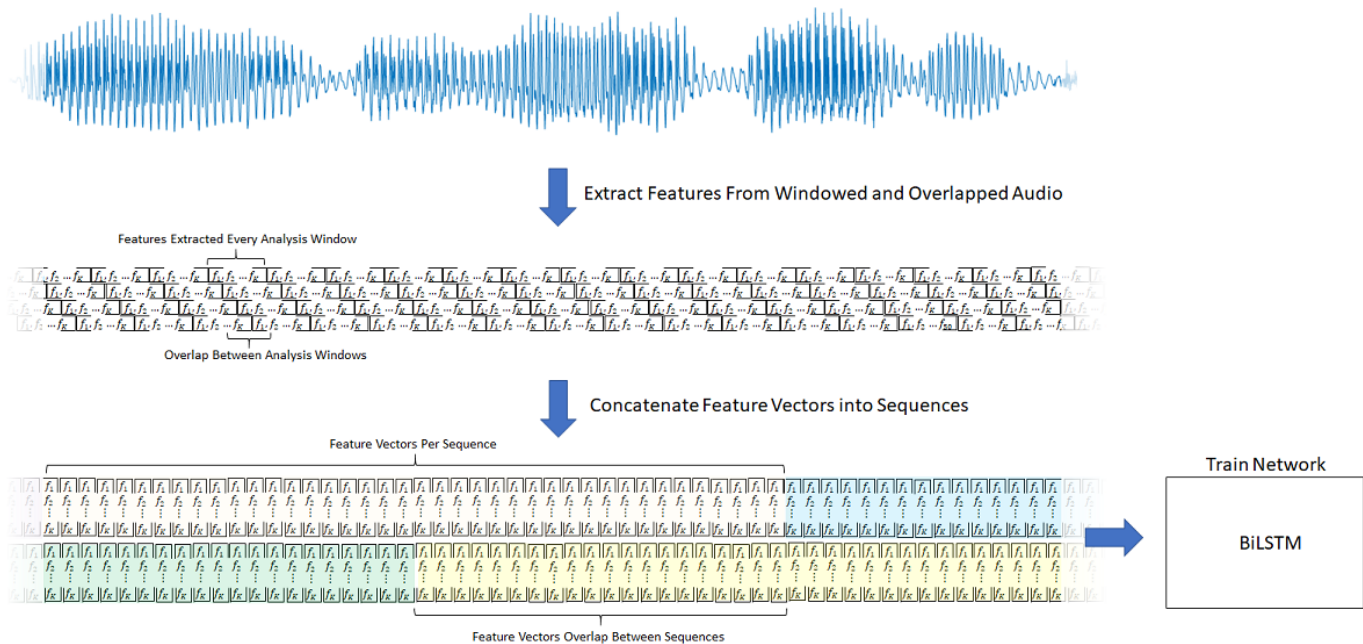
```
NumFeatures = 45
```

```
FeatureVectorsPerSequence = 20
```

```
NumSequence = numel(labels)
```

```
NumSequence = 27
```

The figure provides an overview of the feature extraction used per detected speech region.



Create Training and Test Datasets

This example uses a subset of the Mozilla Common Voice dataset [1] on page 1-0 . The dataset contains 48 kHz recordings of subjects speaking short sentences. Download the dataset and untar the downloaded file. Set PathToDatabase to the location of the data.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/commonvoice.zip';
downloadFolder = tempdir;
dataFolder = fullfile(downloadFolder,'commonvoice');

if ~exist(dataFolder,'dir')
    disp('Downloading data set (956 MB) ...')
    unzip(url,downloadFolder)
end
```

Use `audioDatastore` to create datastores for the training and validation sets. Use `readtable` to read the metadata associated with the audio files.

```
loc = fullfile(dataFolder);
adsTrain = audioDatastore(fullfile(loc,'train'),'IncludeSubfolders',true);
metadataTrain = readtable(fullfile(fullfile(loc,'train'),'train.tsv'),'FileType','text');
adsTrain.Labels = metadataTrain.gender;

adsValidation = audioDatastore(fullfile(loc,'validation'),'IncludeSubfolders',true);
metadataValidation = readtable(fullfile(fullfile(loc,'validation'),'validation.tsv'),'FileType','text');
adsValidation.Labels = metadataValidation.gender;
```

Use `countEachLabel` to inspect the gender breakdown of the training and validation sets.

```
countEachLabel(adsTrain)
```

```
ans=2x2 table
    Label    Count
    _____
    female   1000
    male     1000
```

```
countEachLabel(adsValidation)
```

```
ans=2x2 table
    Label    Count
    _____
    female    200
    male      200
```

To train the network with the entire dataset and achieve the highest possible accuracy, set `reduceDataset` to `false`. To run this example quickly, set `reduceDataset` to `true`.

```
reduceDataset = false;
if reduceDataset
    % Reduce the training dataset by a factor of 20
    adsTrain = splitEachLabel(adsTrain,round(numel(adsTrain.Files) / 2 / 20));
    adsValidation = splitEachLabel(adsValidation,20);
end
```

Create Training and Validation Sets

Determine the sample rate of audio files in the data set, and then update the sample rate, window, and overlap length of the audio feature extractor.

```
[~,adsInfo] = read(adsTrain);
Fs = adsInfo.SampleRate;
extractor.SampleRate = Fs;
extractor.Window = hamming(round(0.03*Fs),'periodic');
extractor.OverlapLength = round(0.02*Fs);
```

To speed up processing, distribute computations over multiple workers. If you have Parallel Computing Toolbox™, the example partitions the datastore so that the feature extraction occurs in parallel across available workers. Determine the optimal number of partitions for your system. If you do not have Parallel Computing Toolbox™, the example uses a single worker.


```

if ~isempty(ver('parallel')) && ~reduceDataset
    pool = gcp;
    numPar = numpartitions(adsTrain,pool);
else
    numPar = 1;
end

```

In a loop:

- 1 Read from the audio datastore.
- 2 Detect regions of speech.
- 3 Extract feature vectors from the regions of speech.

Replicate the labels so that they are in one-to-one correspondence with the feature vectors.

```

labelsTrain = [];
featureVectors = {};

```

```
% Loop over optimal number of partitions
```

```
parfor ii = 1:numPar
```

```
    % Partition datastore
```

```
    subds = partition(adsTrain,numPar,ii);
```

```
    % Preallocation
```

```
    featureVectorsInSubDS = {};
```

```
    segmentsPerFile = zeros(numel(subds.Files),1);
```

```
    % Loop over files in partitioned datastore
```

```
    for jj = 1:numel(subds.Files)
```

```
        % 1. Read in a single audio file
```

```
        audioIn = read(subds);
```

```
        % 2. Determine the regions of the audio that correspond to speech
```

```
        speechIndices = detectSpeech(audioIn,Fs);
```

```
        % 3. Extract features from each speech segment
```

```
        segmentsPerFile(jj) = size(speechIndices,1);
```

```
        features = cell(segmentsPerFile(jj),1);
```

```
        for kk = 1:size(speechIndices,1)
```

```
            features{kk} = ( extract(extractor,audioIn(speechIndices(kk,1):speechIndices(kk,2)))
```

```
        end
```

```
        featureVectorsInSubDS = [featureVectorsInSubDS;features(:)];
```

```
    end
```

```
    featureVectors = [featureVectors;featureVectorsInSubDS];
```

```
    % Replicate the labels so that they are in one-to-one correspondance
```

```
    % with the feature vectors.
```

```
    repedLabels = repelem(subds.Labels,segmentsPerFile);
```

```
    labelsTrain = [labelsTrain;repedLabels(:)];
```

```
end
```

In classification applications, it is good practice to normalize all features to have zero mean and unity standard deviation.

Compute the mean and standard deviation for each coefficient, and use them to normalize the data.

```
allFeatures = cat(2,featureVectors{:});
allFeatures(isinf(allFeatures)) = nan;
M = mean(allFeatures,2,'omitnan');
S = std(allFeatures,0,2,'omitnan');
featureVectors = cellfun(@(x)(x-M)./S,featureVectors,'UniformOutput',false);
for ii = 1:numel(featureVectors)
    idx = find(isnan(featureVectors{ii}));
    if ~isempty(idx)
        featureVectors{ii}(idx) = 0;
    end
end
```

Buffer the feature vectors into sequences of 20 feature vectors with 10 overlap. If a sequence has less than 20 feature vectors, drop it.

```
[featuresTrain,trainSequencePerSegment] = HelperFeatureVector2Sequence(featureVectors,featureVec
```

Replicate the labels so that they are in one-to-one correspondence with the sequences.

```
labelsTrain = repelem(labelsTrain,[trainSequencePerSegment{:}]);
labelsTrain = categorical(labelsTrain);
```

Create the validation set using the same steps used to create the training set.

```
labelsValidation = [];
featureVectors = {};
valSegmentsPerFile = [];
parfor ii = 1:numPar
    subds = partition(adsValidation,numPar,ii);
    featureVectorsInSubDS = {};
    valSegmentsPerFileInSubDS = zeros(numel(subds.Files),1);
    for jj = 1:numel(subds.Files)
        audioIn = read(subds);
        speechIndices = detectSpeech(audioIn,Fs);
        numSegments = size(speechIndices,1);
        features = cell(valSegmentsPerFileInSubDS(jj),1);
        for kk = 1:numSegments
            features{kk} = ( extract(extractor,audioIn(speechIndices(kk,1):speechIndices(kk,2)))
        end
        featureVectorsInSubDS = [featureVectorsInSubDS;features{:}];
        valSegmentsPerFileInSubDS(jj) = numSegments;
    end
    repedLabels = repelem(subds.Labels,valSegmentsPerFileInSubDS);
    labelsValidation = [labelsValidation;repedLabels{:}];
    featureVectors = [featureVectors;featureVectorsInSubDS];
    valSegmentsPerFile = [valSegmentsPerFile;valSegmentsPerFileInSubDS];
end

featureVectors = cellfun(@(x)(x-M)./S,featureVectors,'UniformOutput',false);
for ii = 1:numel(featureVectors)
    idx = find(isnan(featureVectors{ii}));
    if ~isempty(idx)
        featureVectors{ii}(idx) = 0;
    end
end
```

```
[featuresValidation,valSequencePerSegment] = HelperFeatureVector2Sequence(featureVectors,featureV
labelsValidation = repelem(labelsValidation,[valSequencePerSegment{:}]);
labelsValidation = categorical(labelsValidation);
```

Define the LSTM Network Architecture

LSTM networks can learn long-term dependencies between time steps of sequence data. This example uses the bidirectional LSTM layer `bilstmLayer` to look at the sequence in both forward and backward directions.

Specify the input size to be sequences of size `NumFeatures`. Specify a hidden bidirectional LSTM layer with an output size of 50 and output a sequence. Then, specify a bidirectional LSTM layer with an output size of 50 and output the last element of the sequence. This command instructs the bidirectional LSTM layer to map its input into 50 features and then prepares the output for the fully connected layer. Finally, specify two classes by including a fully connected layer of size 2, followed by a softmax layer and a classification layer.

```
layers = [ ...
    sequenceInputLayer(size(featuresTrain{1},1))
    bilstmLayer(50,"OutputMode","sequence")
    bilstmLayer(50,"OutputMode","last")
    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer];
```

Next, specify the training options for the classifier. Set `MaxEpochs` to 4 so that the network makes 4 passes through the training data. Set `MiniBatchSize` of 256 so that the network looks at 128 training signals at a time. Specify `Plots` as "training-progress" to generate plots that show the training progress as the number of iterations increases. Set `Verbose` to `false` to disable printing the table output that corresponds to the data shown in the plot. Specify `Shuffle` as "every-epoch" to shuffle the training sequence at the beginning of each epoch. Specify `LearnRateSchedule` to "piecewise" to decrease the learning rate by a specified factor (0.1) every time a certain number of epochs (1) has passed.

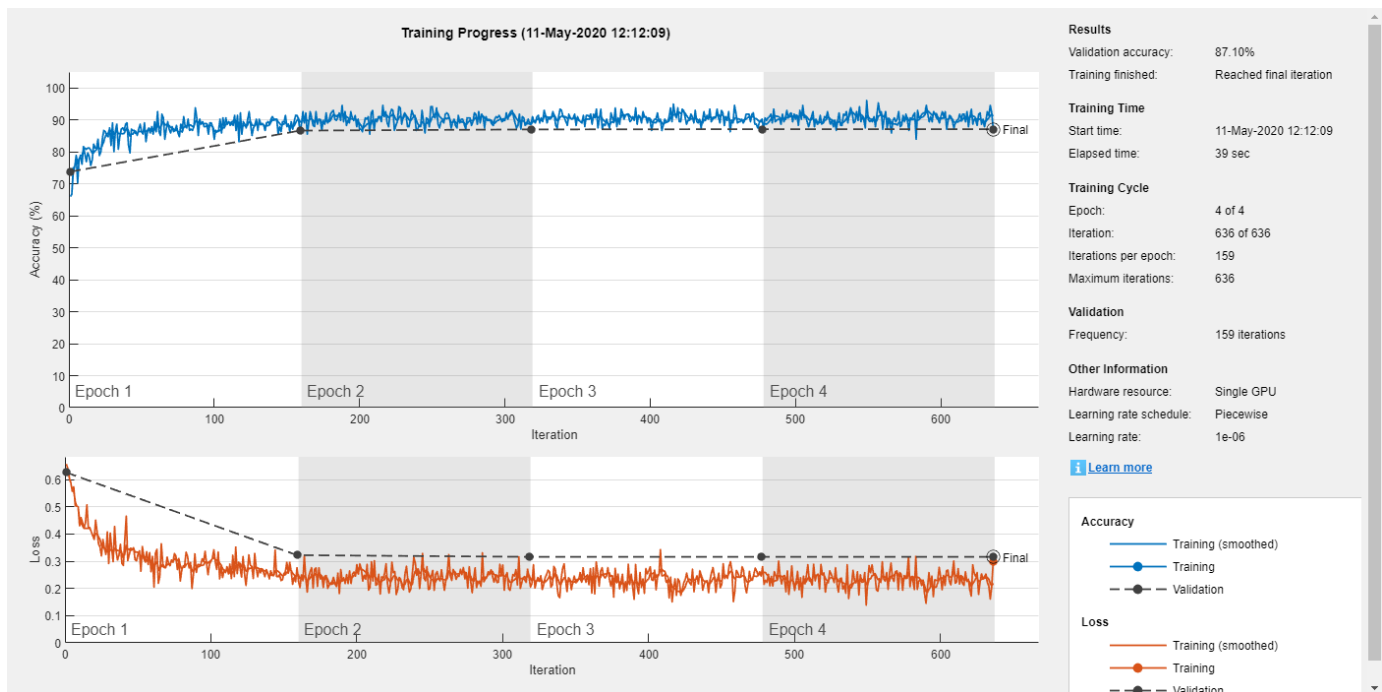
This example uses the adaptive moment estimation (ADAM) solver. ADAM performs better with recurrent neural networks (RNNs) like LSTMs than the default stochastic gradient descent with momentum (SGDM) solver.

```
miniBatchSize = 256;
validationFrequency = floor(numel(labelsTrain)/miniBatchSize);
options = trainingOptions("adam", ...
    "MaxEpochs",4, ...
    "MiniBatchSize",miniBatchSize, ...
    "Plots","training-progress", ...
    "Verbose",false, ...
    "Shuffle","every-epoch", ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.1, ...
    "LearnRateDropPeriod",1, ...
    'ValidationData',{featuresValidation,labelsValidation}, ...
    'ValidationFrequency',validationFrequency);
```

Train the LSTM Network

Train the LSTM network with the specified training options and layer architecture using `trainNetwork`. Because the training set is large, the training process can take several minutes.

```
net = trainNetwork(featuresTrain,labelsTrain,layers,options);
```



The top subplot of the training-progress plot represents the training accuracy, which is the classification accuracy on each mini-batch. When training progresses successfully, this value typically increases towards 100%. The bottom subplot displays the training loss, which is the cross-entropy loss on each mini-batch. When training progresses successfully, this value typically decreases towards zero.

If the training is not converging, the plots might oscillate between values without trending in a certain upward or downward direction. This oscillation means that the training accuracy is not improving and the training loss is not decreasing. This situation can occur at the start of training, or after some preliminary improvement in training accuracy. In many cases, changing the training options can help the network achieve convergence. Decreasing `MiniBatchSize` or decreasing `InitialLearnRate` might result in a longer training time, but it can help the network learn better.

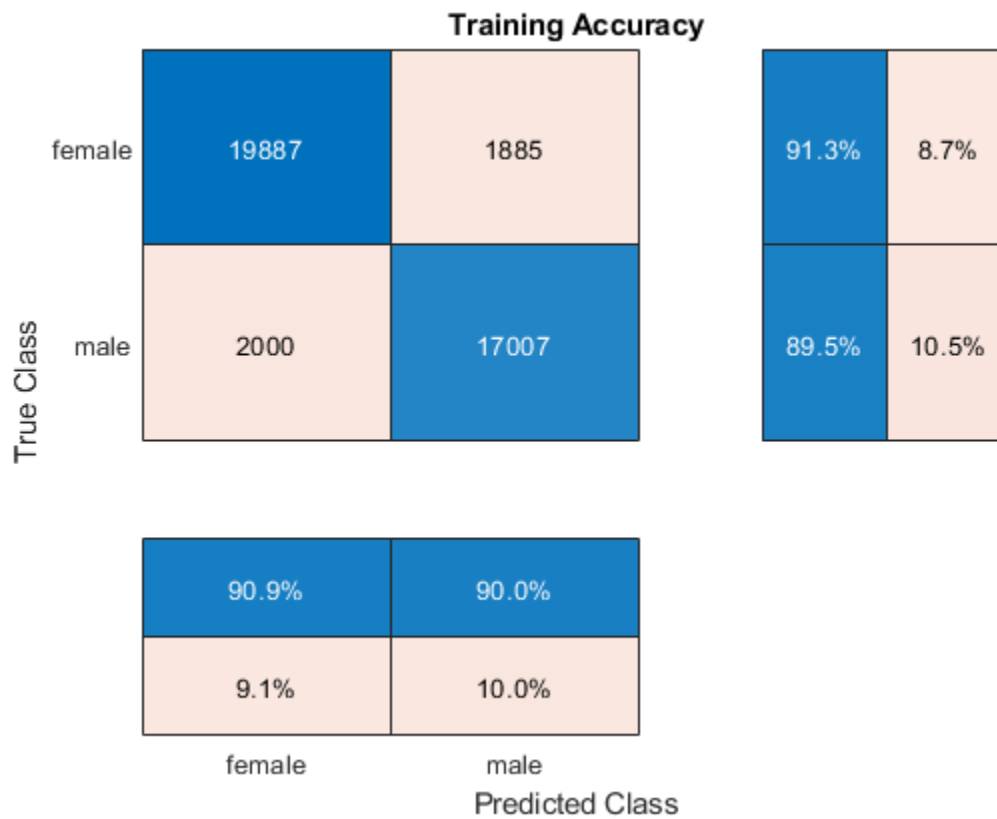
Visualize the Training Accuracy

Calculate the training accuracy, which represents the accuracy of the classifier on the signals on which it was trained. First, classify the training data.

```
prediction = classify(net,featuresTrain);
```

Plot the confusion matrix. Display the precision and recall for the two classes by using column and row summaries.

```
figure
cm = confusionchart(categorical(labelsTrain),prediction,'title','Training Accuracy');
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
```



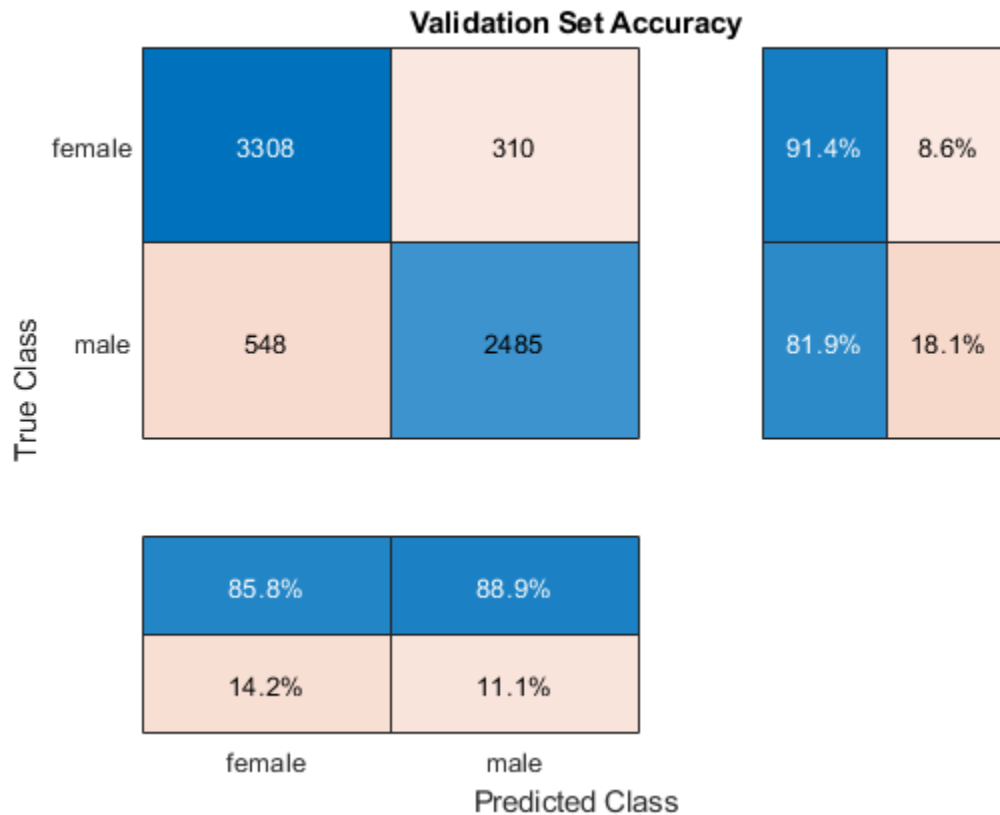
Visualize the Validation Accuracy

Calculate the validation accuracy. First, classify the training data.

```
[prediction,probabilities] = classify(net,featuresValidation);
```

Plot the confusion matrix. Display the precision and recall for the two classes by using column and row summaries.

```
figure
cm = confusionchart(categorical(labelsValidation),prediction,'title','Validation Set Accuracy');
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
```



The example generated multiple sequences from each training speech file. Higher accuracy can be achieved by considering the output class of all sequences corresponding to the same file, and applying a "max-rule" decision, where the class with the segment with the highest confidence score is selected.

Determine the number of sequences generated per file in the validation set.

```
sequencePerFile = zeros(size(valSegmentsPerFile));
valSequencePerSegmentMat = cell2mat(valSequencePerSegment);
idx = 1;
for ii = 1:numel(valSegmentsPerFile)
    sequencePerFile(ii) = sum(valSequencePerSegmentMat(idx:idx+valSegmentsPerFile(ii)-1));
    idx = idx + valSegmentsPerFile(ii);
end
```

Predict the gender from each training file by considering the output classes of all sequences generated from the same file.

```
numFiles = numel(adsValidation.Files);
actualGender = categorical(adsValidation.Labels);
predictedGender = actualGender;
scores = cell(1,numFiles);
counter = 1;
cats = unique(actualGender);
for index = 1:numFiles
    scores{index} = probabilities(counter: counter + sequencePerFile(index) - 1,:);
    m = max(mean(scores{index},1),[],1);
```

```

if m(1) >= m(2)
    predictedGender(index) = cats(1);
else
    predictedGender(index) = cats(2);
end
counter = counter + sequencePerFile(index);
end

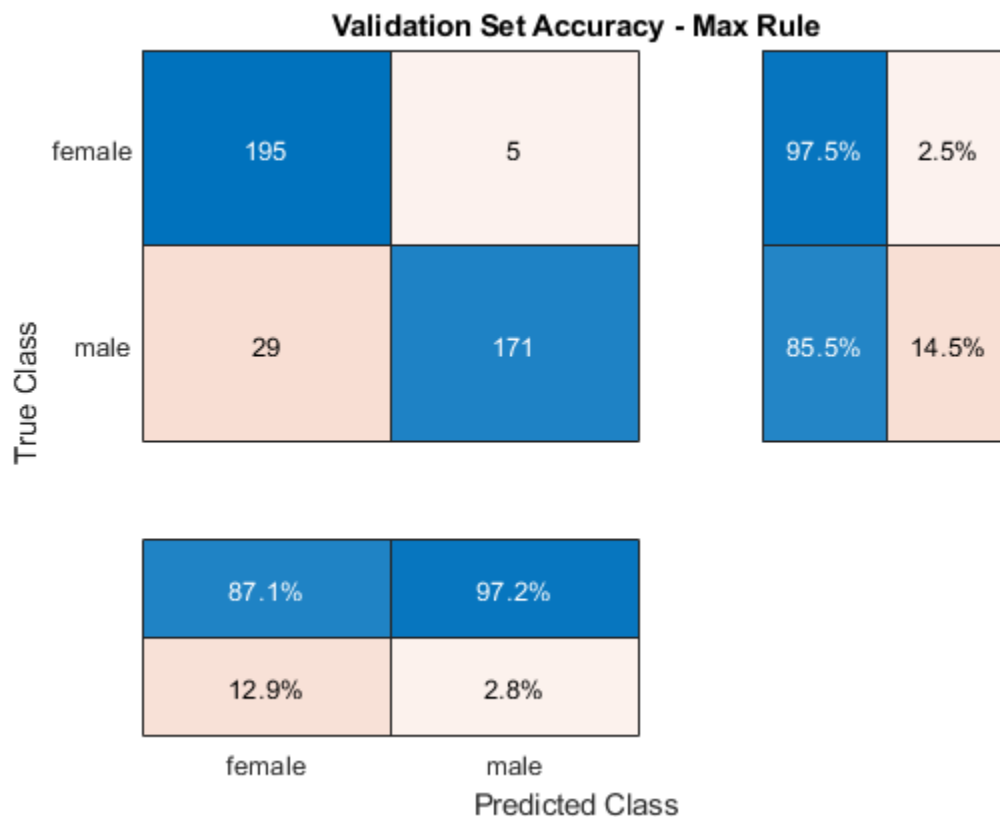
```

Visualize the confusion matrix on the majority-rule predictions.

```

figure
cm = confusionchart(actualGender,predictedGender,'title','Validation Set Accuracy - Max Rule');
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';

```



References

[1] Mozilla Common Voice Dataset

Supporting Functions

```

function [sequences,sequencePerSegment] = HelperFeatureVector2Sequence(features,featureVectorsPerSequence,featureVector0Overlap)
if featureVectorsPerSequence <= featureVector0Overlap
    error('The number of overlapping feature vectors must be less than the number of feature vectors per sequence');
end

hopLength = featureVectorsPerSequence - featureVector0Overlap;
idx1 = 1;

```

```
sequences = {};  
sequencePerSegment = cell(numel(features),1);  
for ii = 1:numel(features)  
    sequencePerSegment{ii} = max(floor((size(features{ii},2) - featureVectorsPerSequence)/hopLength),1);  
    idx2 = 1;  
    for j = 1:sequencePerSegment{ii}  
        sequences{idx1,1} = features{ii}(:,idx2:idx2 + featureVectorsPerSequence - 1);  
        idx1 = idx1 + 1;  
        idx2 = idx2 + hopLength;  
    end  
end  
end
```


Speech Command Recognition Using Deep Learning

This example shows how to train a deep learning model that detects the presence of speech commands in audio. The example uses the Speech Commands Dataset [1] to train a convolutional neural network to recognize a given set of commands.

To train a network from scratch, you must first download the data set. If you do not want to download the data set or train the network, then you can load a pretrained network provided with this example and execute the next two sections of the example: *Recognize Commands with a Pre-Trained Network* and *Detect Commands Using Streaming Audio from Microphone*.

Recognize Commands with a Pre-Trained Network

Before going into the training process in detail, you will use a pre-trained speech recognition network to identify speech commands.

Load the pre-trained network.

```
load('commandNet.mat')
```

The network is trained to recognize the following speech commands:

- "yes"
- "no"
- "up"
- "down"
- "left"
- "right"
- "on"
- "off"
- "stop"
- "go"

Load a short speech signal where a person says "stop".

```
[x,fs] = audioread('stop_command.flac');
```

Listen to the command.

```
sound(x,fs)
```

The pre-trained network takes auditory-based spectrograms as inputs. You will first convert the speech waveform to an auditory-based spectrogram.

Use the function `extractAuditoryFeature` to compute the auditory spectrogram. You will go through the details of feature extraction later in the example.

```
auditorySpect = helperExtractAuditoryFeatures(x,fs);
```

Classify the command based on its auditory spectrogram.

```
command = classify(trainedNet,auditorySpect)
```

```
command =  
    categorical  
        stop
```

The network is trained to classify words not belonging to this set as "unknown".

You will now classify a word ("play") that was not included in the list of command to identify.

Load the speech signal and listen to it.

```
x = audioread('play_command.flac');  
sound(x,fs)
```

Compute the auditory spectrogram.

```
auditorySpect = helperExtractAuditoryFeatures(x,fs);
```

Classify the signal.

```
command = classify(trainedNet,auditorySpect)
```

```
command =  
    categorical  
        unknown
```

The network is trained to classify background noise as "background".

Create a one-second signal consisting of random noise.

```
x = pinknoise(16e3);
```

Compute the auditory spectrogram.

```
auditorySpect = helperExtractAuditoryFeatures(x,fs);
```

Classify the background noise.

```
command = classify(trainedNet,auditorySpect)
```

```
command =  
    categorical  
        background
```

Detect Commands Using Streaming Audio from Microphone

Test your pre-trained command detection network on streaming audio from your microphone. Try saying one of the commands, for example, *yes*, *no*, or *stop*. Then, try saying one of the unknown words such as *Marvin*, *Sheila*, *bed*, *house*, *cat*, *bird*, or any number from zero to nine.

Specify the classification rate in Hz and create an audio device reader that can read audio from your microphone.

```
classificationRate = 20;
adr = audioDeviceReader('SampleRate',fs,'SamplesPerFrame',floor(fs/classificationRate));
```

Initialize a buffer for the audio. Extract the classification labels of the network. Initialize buffers of half a second for the labels and classification probabilities of the streaming audio. Use these buffers to compare the classification results over a longer period of time and by that build 'agreement' over when a command is detected. Specify thresholds for the decision logic.

```
audioBuffer = dsp.AsyncBuffer(fs);

labels = trainedNet.Layers(end).Classes;
YBuffer(1:classificationRate/2) = categorical("background");

probBuffer = zeros([numel(labels),classificationRate/2]);

countThreshold = ceil(classificationRate*0.2);
probThreshold = 0.7;
```

Create a figure and detect commands as long as the created figure exists. To run the loop indefinitely, set `timeLimit` to `Inf`. To stop the live detection, simply close the figure.

```
h = figure('Units','normalized','Position',[0.2 0.1 0.6 0.8]);

timeLimit = 20;

tic
while ishandle(h) && toc < timeLimit

    % Extract audio samples from the audio device and add the samples to
    % the buffer.
    x = adr();
    write(audioBuffer,x);
    y = read(audioBuffer,fs,fs-adr.SamplesPerFrame);

    spec = helperExtractAuditoryFeatures(y,fs);

    % Classify the current spectrogram, save the label to the label buffer,
    % and save the predicted probabilities to the probability buffer.
    [YPredicted,probs] = classify(trainedNet,spec,'ExecutionEnvironment','cpu');
    YBuffer = [YBuffer(2:end),YPredicted];
    probBuffer = [probBuffer(:,2:end),probs(:)];

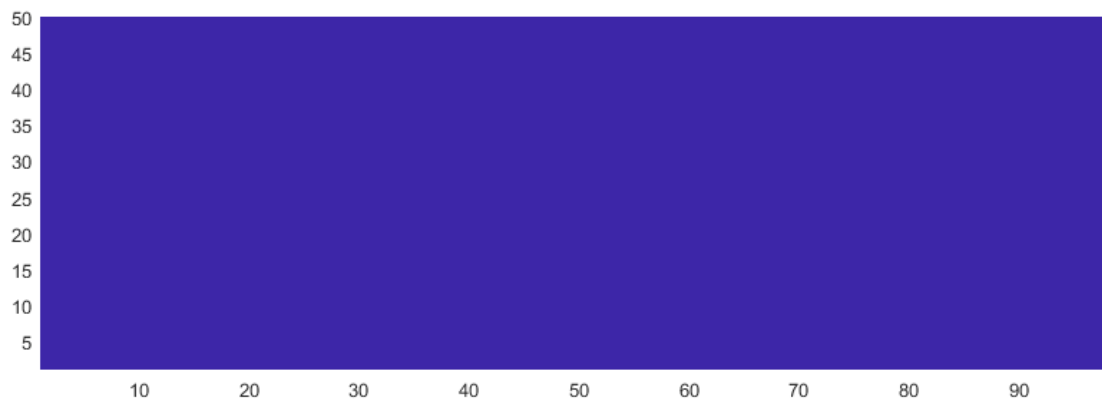
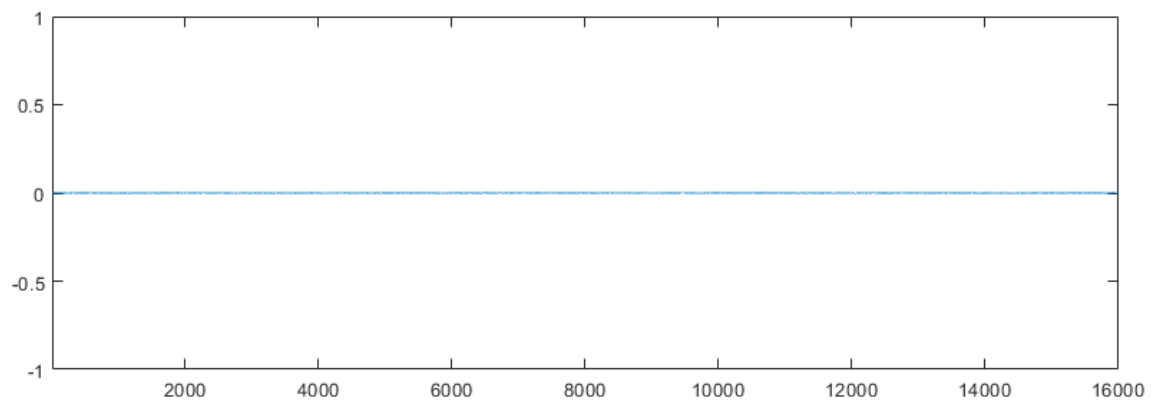
    % Plot the current waveform and spectrogram.
    subplot(2,1,1)
    plot(y)
    axis tight
    ylim([-1,1])
```

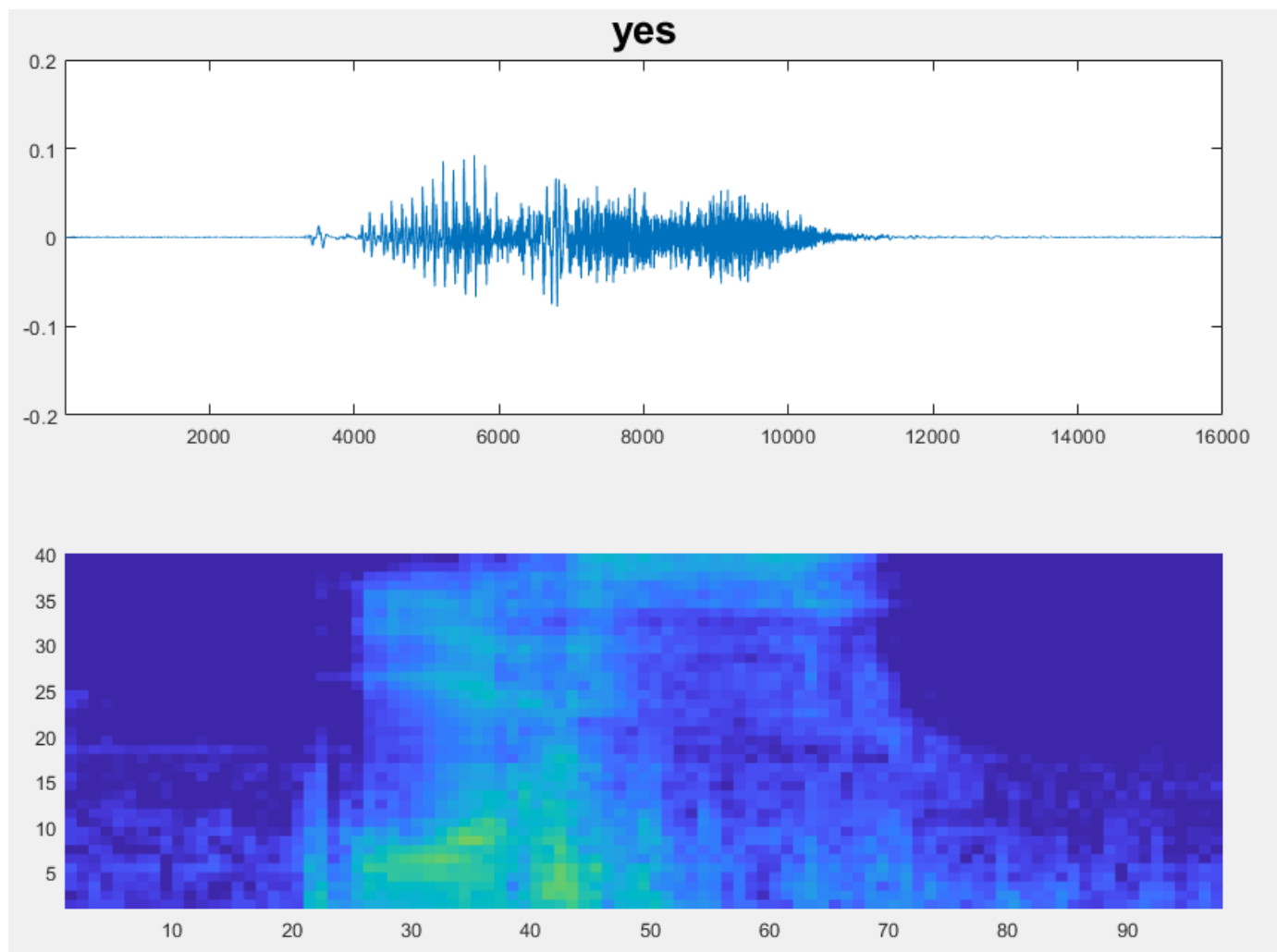
```
subplot(2,1,2)
pcolor(spec')
caxis([-4 2.6445])
shading flat

% Now do the actual command detection by performing a very simple
% thresholding operation. Declare a detection and display it in the
% figure title if all of the following hold: 1) The most common label
% is not background. 2) At least countThreshold of the latest frame
% labels agree. 3) The maximum probability of the predicted label is at
% least probThreshold. Otherwise, do not declare a detection.
[YMode,count] = mode(YBuffer);

maxProb = max(probBuffer(labels == YMode,:));
subplot(2,1,1)
if YMode == "background" || count < countThreshold || maxProb < probThreshold
    title(" ")
else
    title(string(YMode),'FontSize',20)
end

drawnow
end
```





Load Speech Commands Data Set

This example uses the Google Speech Commands Dataset [1]. Download the dataset and untar the downloaded file. Set PathToDatabase to the location of the data.

```
url = 'https://ssd.mathworks.com/supportfiles/audio/google_speech.zip';
downloadFolder = tempdir;
dataFolder = fullfile(downloadFolder,'google_speech');

if ~exist(dataFolder,'dir')
    disp('Downloading data set (1.4 GB) ...')
    unzip(url,downloadFolder)
end
```

Create Training Datastore

Create an audioDatastore that points to the training data set.

```
ads = audioDatastore(fullfile(dataFolder, 'train'), ...
    'IncludeSubfolders',true, ...
    'FileExtensions','.wav', ...
    'LabelSource','foldernames')
```

```
ads =
```

```
    audioDatastore with properties:
```

```
        Files: {
            ' ...\AppData\Local\Temp\google_speech\train\bed\00176480_nohash_0
            ' ...\AppData\Local\Temp\google_speech\train\bed\004ae714_nohash_0
            ' ...\AppData\Local\Temp\google_speech\train\bed\004ae714_nohash_1
            ... and 51085 more
        }
        Folders: {
            'C:\Users\jibrahim\AppData\Local\Temp\google_speech\train'
        }
        Labels: [bed; bed; bed ... and 51085 more categorical]
AlternateFileSystemRoots: {}
        OutputDataType: 'double'
SupportedOutputFormats: ["wav"      "flac"      "ogg"      "mp4"      "m4a"]
        DefaultOutputFormat: "wav"
```

Choose Words to Recognize

Specify the words that you want your model to recognize as commands. Label all words that are not commands as unknown. Labeling words that are not commands as unknown creates a group of words that approximates the distribution of all words other than the commands. The network uses this group to learn the difference between commands and all other words.

To reduce the class imbalance between the known and unknown words and speed up processing, only include a fraction of the unknown words in the training set.

Use `subset` to create a datastore that contains only the commands and the subset of unknown words. Count the number of examples belonging to each category.

```
commands = categorical(["yes","no","up","down","left","right","on","off","stop","go"]);

isCommand = ismember(ads.Labels,commands);
isUnknown = ~isCommand;

includeFraction = 0.2;
mask = rand(numel(ads.Labels),1) < includeFraction;
isUnknown = isUnknown & mask;
ads.Labels(isUnknown) = categorical("unknown");

adsTrain = subset(ads,isCommand|isUnknown);
countEachLabel(adsTrain)
```

```
ans =
```

```
    11x2 table
```

Label	Count
down	1842
go	1861
left	1839

no	1853
off	1839
on	1864
right	1852
stop	1885
unknown	6483
up	1843
yes	1860

Create Validation Datastore

Create an `audioDatastore` that points to the validation data set. Follow the same steps used to create the training datastore.

```
ads = audioDatastore(fullfile(dataFolder, 'validation'), ...  
    'IncludeSubfolders',true, ...  
    'FileExtensions','.wav', ...  
    'LabelSource','foldernames')
```

```
isCommand = ismember(ads.Labels,commands);  
isUnknown = ~isCommand;
```

```
includeFraction = 0.2;  
mask = rand(numel(ads.Labels),1) < includeFraction;  
isUnknown = isUnknown & mask;  
ads.Labels(isUnknown) = categorical("unknown");
```

```
adsValidation = subset(ads,isCommand|isUnknown);  
countEachLabel(adsValidation)
```

```
ads =
```

```
audioDatastore with properties:
```

```
Files: {  
    ' ...\AppData\Local\Temp\google_speech\validation\bed\026290a7_noha  
    ' ...\AppData\Local\Temp\google_speech\validation\bed\060cd039_noha  
    ' ...\AppData\Local\Temp\google_speech\validation\bed\060cd039_noha  
    ... and 6795 more  
}  
Folders: {  
    'C:\Users\jibrahim\AppData\Local\Temp\google_speech\validation'  
}  
Labels: [bed; bed; bed ... and 6795 more categorical]  
AlternateFileSystemRoots: {}  
OutputDataType: 'double'  
SupportedOutputFormats: ["wav"    "flac"    "ogg"    "mp4"    "m4a"]  
DefaultOutputFormat: "wav"
```

```
ans =
```

```
11×2 table
```

Label	Count
-------	-------

down	264
go	260
left	247
no	270
off	256
on	257
right	256
stop	246
unknown	850
up	260
yes	261

To train the network with the entire dataset and achieve the highest possible accuracy, set `reduceDataset` to `false`. To run this example quickly, set `reduceDataset` to `true`.

```
reduceDataset = false;
if reduceDataset
    numUniqueLabels = numel(unique(adsTrain.Labels));
    % Reduce the dataset by a factor of 20
    adsTrain = splitEachLabel(adsTrain,round(numel(adsTrain.Files) / numUniqueLabels / 20));
    adsValidation = splitEachLabel(adsValidation,round(numel(adsValidation.Files) / numUniqueLabels / 20));
end
```

Compute Auditory Spectrograms

To prepare the data for efficient training of a convolutional neural network, convert the speech waveforms to auditory-based spectrograms.

Define the parameters of the feature extraction. `segmentDuration` is the duration of each speech clip (in seconds). `frameDuration` is the duration of each frame for spectrum calculation. `hopDuration` is the time step between each spectrum. `numBands` is the number of filters in the auditory spectrogram.

Create an `audioFeatureExtractor` object to perform the feature extraction.

```
fs = 16e3; % Known sample rate of the data set.

segmentDuration = 1;
frameDuration = 0.025;
hopDuration = 0.010;

segmentSamples = round(segmentDuration*fs);
frameSamples = round(frameDuration*fs);
hopSamples = round(hopDuration*fs);
overlapSamples = frameSamples - hopSamples;

FFTLenght = 512;
numBands = 50;

afe = audioFeatureExtractor( ...
    'SampleRate',fs, ...
    'FFTLenght',FFTLenght, ...
    'Window',hann(frameSamples,'periodic'), ...
    'OverlapLength',overlapSamples, ...
    'barkSpectrum',true);
setExtractorParams(afe,'barkSpectrum','NumBands',numBands,'WindowNormalization',false);
```

Read a file from the dataset. Training a convolutional neural network requires input to be a consistent size. Some files in the data set are less than 1 second long. Apply zero-padding to the front and back of the audio signal so that it is of length `segmentSamples`.

```
x = read(adsTrain);

numSamples = size(x,1);

numToPadFront = floor( (segmentSamples - numSamples)/2 );
numToPadBack = ceil( (segmentSamples - numSamples)/2 );

xPadded = [zeros(numToPadFront,1,'like',x);x;zeros(numToPadBack,1,'like',x)];
```

To extract audio features, call `extract`. The output is a Bark spectrum with time across rows.

```
features = extract(afe,xPadded);
[numHops,numFeatures] = size(features)
```

```
numHops =
```

```
98
```

```
numFeatures =
```

```
50
```

In this example, you post-process the auditory spectrogram by applying a logarithm. Taking a log of small numbers can lead to roundoff error.

To speed up processing, you can distribute the feature extraction across multiple workers using `parfor`.

First, determine the number of partitions for the dataset. If you do not have Parallel Computing Toolbox™, use a single partition.

```
if ~isempty(ver('parallel')) && ~reduceDataset
    pool = gcp;
    numPar = numpartitions(adsTrain,pool);
else
    numPar = 1;
end
```

For each partition, read from the datastore, zero-pad the signal, and then extract the features.

```
parfor ii = 1:numPar
    subds = partition(adsTrain,numPar,ii);
    XTrain = zeros(numHops,numBands,1,numel(subds.Files));
    for idx = 1:numel(subds.Files)
        x = read(subds);
        xPadded = [zeros(floor((segmentSamples-size(x,1))/2),1);x;zeros(ceil((segmentSamples-size(x,1))/2),1)];
        XTrain(:,:,,idx) = extract(afe,xPadded);
    end
    XTrainC{ii} = XTrain;
end
```

Convert the output to a 4-dimensional array with auditory spectrograms along the fourth dimension.

```

XTrain = cat(4,XTrainC{:});

[numHops,numBands,numChannels,numSpec] = size(XTrain)

numHops =

    98

numBands =

    50

numChannels =

    1

numSpec =

    25021

```

Scale the features by the window power and then take the log. To obtain data with a smoother distribution, take the logarithm of the spectrograms using a small offset.

```

epsil = 1e-6;
XTrain = log10(XTrain + epsil);

```

Perform the feature extraction steps described above to the validation set.

```

if ~isempty(ver('parallel'))
    pool = gcp;
    numPar = numpartitions(adsValidation,pool);
else
    numPar = 1;
end
parfor ii = 1:numPar
    subds = partition(adsValidation,numPar,ii);
    XValidation = zeros(numHops,numBands,1,numel(subds.Files));
    for idx = 1:numel(subds.Files)
        x = read(subds);
        xPadded = [zeros(floor((segmentSamples-size(x,1))/2),1);x;zeros(ceil((segmentSamples-size(x,1))/2),1)];
        XValidation(:,:,,idx) = extract(afe,xPadded);
    end
    XValidationC{ii} = XValidation;
end
XValidation = cat(4,XValidationC{:});
XValidation = log10(XValidation + epsil);

```

Isolate the train and validation labels. Remove empty categories.

```

YTrain = removecats(adsTrain.Labels);
YValidation = removecats(adsValidation.Labels);

```

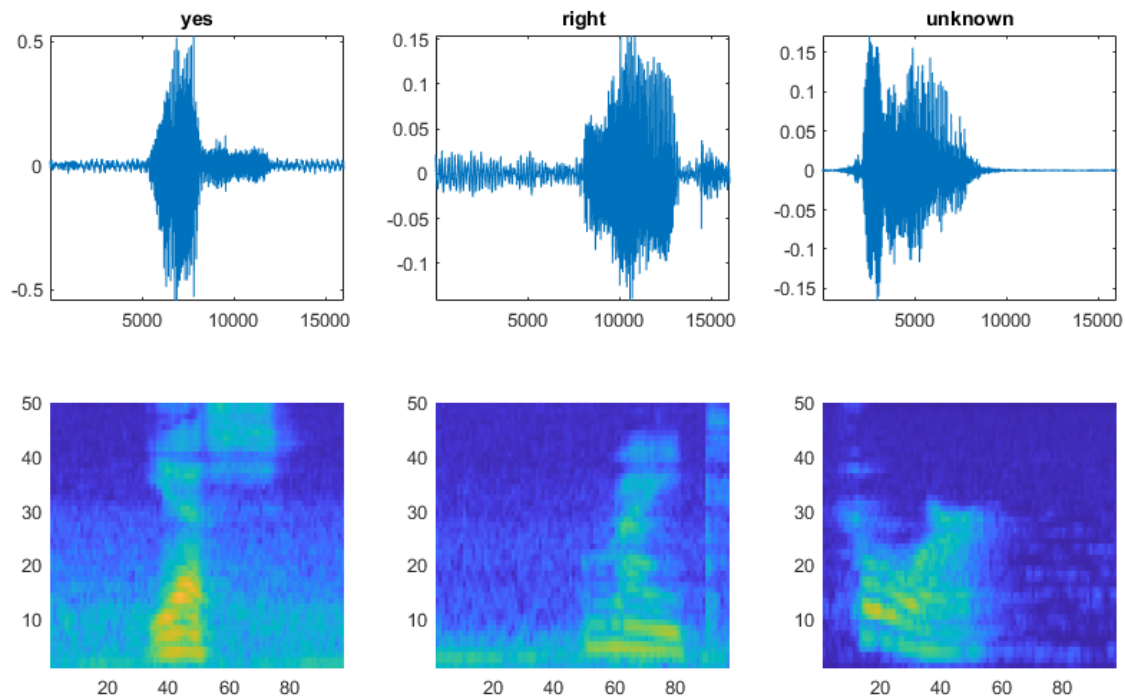
Visualize Data

Plot the waveforms and auditory spectrograms of a few training samples. Play the corresponding audio clips.

```
specMin = min(XTrain,[],'all');
specMax = max(XTrain,[],'all');
idx = randperm(numel(adsTrain.Files),3);
figure('Units','normalized','Position',[0.2 0.2 0.6 0.6]);
for i = 1:3
    [x,fs] = audioread(adsTrain.Files{idx(i)});
    subplot(2,3,i)
    plot(x)
    axis tight
    title(string(adsTrain.Labels(idx(i))))

    subplot(2,3,i+3)
    spect = (XTrain(:,:,1,idx(i)))';
    pcolor(spect)
    caxis([specMin specMax])
    shading flat

    sound(x,fs)
    pause(2)
end
```



Add Background Noise Data

The network must be able not only to recognize different spoken words but also to detect if the input contains silence or background noise.

Use the audio files in the `_background_` folder to create samples of one-second clips of background noise. Create an equal number of background clips from each background noise file. You can also create your own recordings of background noise and add them to the `_background_` folder. Before calculating the spectrograms, the function rescales each audio clip with a factor sampled from a log-uniform distribution in the range given by `volumeRange`.

```
adsBkg = audioDatastore(fullfile(dataFolder, 'background'))
numBkgClips = 4000;
if reduceDataset
    numBkgClips = numBkgClips/20;
end
volumeRange = log10([1e-4,1]);

numBkgFiles = numel(adsBkg.Files);
numClipsPerFile = histcounts(1:numBkgClips,linspace(1,numBkgClips,numBkgFiles+1));
Xbkg = zeros(size(XTrain,1),size(XTrain,2),1,numBkgClips,'single');
bkgAll = readall(adsBkg);
ind = 1;

for count = 1:numBkgFiles
    bkg = bkgAll{count};
    idxStart = randi(numel(bkg)-fs,numClipsPerFile(count),1);
    idxEnd = idxStart+fs-1;
    gain = 10.^((volumeRange(2)-volumeRange(1))*rand(numClipsPerFile(count),1) + volumeRange(1))
    for j = 1:numClipsPerFile(count)

        x = bkg(idxStart(j):idxEnd(j))*gain(j);

        x = max(min(x,1),-1);

        Xbkg(:,:,j,ind) = extract(afe,x);

        if mod(ind,1000)==0
            disp("Processed " + string(ind) + " background clips out of " + string(numBkgClips))
        end
        ind = ind + 1;
    end
end
Xbkg = log10(Xbkg + epsil);
```

adsBkg =

audioDatastore with properties:

```
Files: {
    ' ...\AppData\Local\Temp\google_speech\background\doing_the_dishes
    ' ...\AppData\Local\Temp\google_speech\background\dude_miaowing.wav
    ' ...\AppData\Local\Temp\google_speech\background\exercise_bike.wav
    ... and 3 more
}
Folders: {
    'C:\Users\jibrahim\AppData\Local\Temp\google_speech\background'
}
AlternateFileSystemRoots: {}
OutputDataType: 'double'
Labels: {}
SupportedOutputFormats: ["wav"    "flac"    "ogg"    "mp4"    "m4a"]
```

```
DefaultOutputFormat: "wav"
```

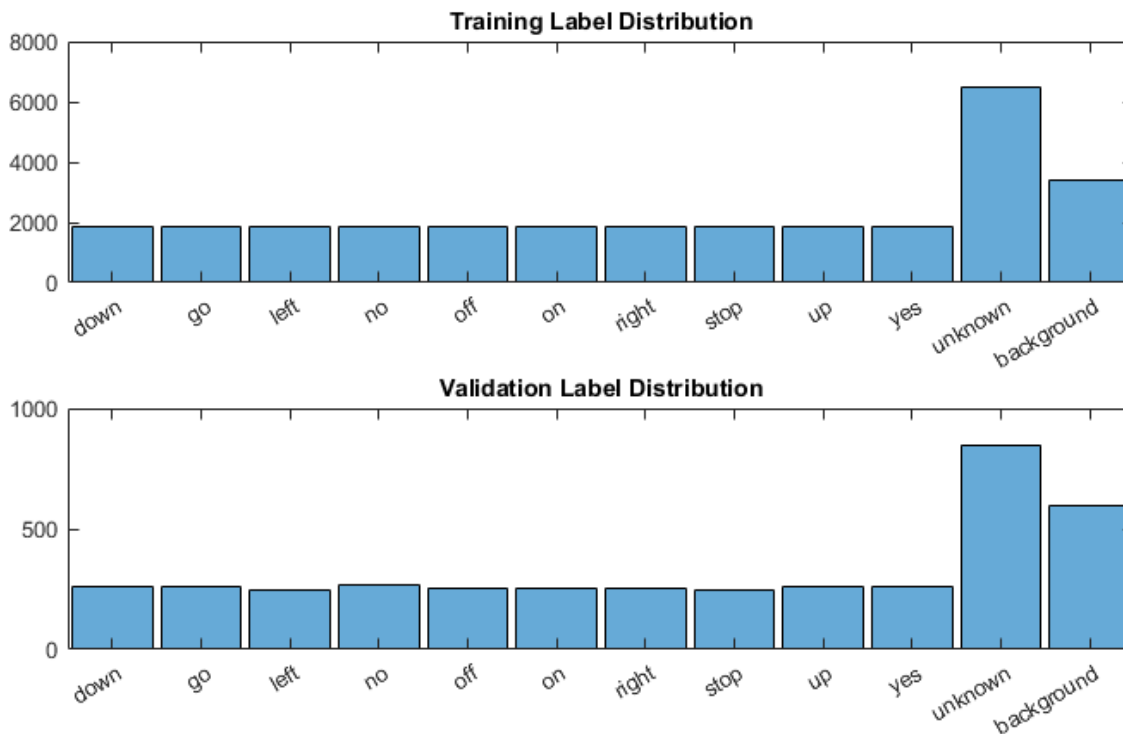
```
Processed 1000 background clips out of 4000  
Processed 2000 background clips out of 4000  
Processed 3000 background clips out of 4000  
Processed 4000 background clips out of 4000
```

Split the spectrograms of background noise between the training, validation, and test sets. Because the `_background_noise_` folder contains only about five and a half minutes of background noise, the background samples in the different data sets are highly correlated. To increase the variation in the background noise, you can create your own background files and add them to the folder. To increase the robustness of the network to noise, you can also try mixing background noise into the speech files.

```
numTrainBkg = floor(0.85*numBkgClips);  
numValidationBkg = floor(0.15*numBkgClips);  
  
XTrain(:,:,:,end+1:end+numTrainBkg) = Xbkg(:,:,:,1:numTrainBkg);  
YTrain(end+1:end+numTrainBkg) = "background";  
  
XValidation(:,:,:,end+1:end+numValidationBkg) = Xbkg(:,:,:,numTrainBkg+1:end);  
YValidation(end+1:end+numValidationBkg) = "background";
```

Plot the distribution of the different class labels in the training and validation sets.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5])  
  
subplot(2,1,1)  
histogram(YTrain)  
title("Training Label Distribution")  
  
subplot(2,1,2)  
histogram(YValidation)  
title("Validation Label Distribution")
```



Define Neural Network Architecture

Create a simple network architecture as an array of layers. Use convolutional and batch normalization layers, and downsample the feature maps "spatially" (that is, in time and frequency) using max pooling layers. Add a final max pooling layer that pools the input feature map globally over time. This enforces (approximate) time-translation invariance in the input spectrograms, allowing the network to perform the same classification independent of the exact position of the speech in time. Global pooling also significantly reduces the number of parameters in the final fully connected layer. To reduce the possibility of the network memorizing specific features of the training data, add a small amount of dropout to the input to the last fully connected layer.

The network is small, as it has only five convolutional layers with few filters. `numF` controls the number of filters in the convolutional layers. To increase the accuracy of the network, try increasing the network depth by adding identical blocks of convolutional, batch normalization, and ReLU layers. You can also try increasing the number of convolutional filters by increasing `numF`.

Use a weighted cross entropy classification loss.

`weightedClassificationLayer(classWeights)` creates a custom classification layer that calculates the cross entropy loss with observations weighted by `classWeights`. Specify the class weights in the same order as the classes appear in `categories(YTrain)`. To give each class equal total weight in the loss, use class weights that are inversely proportional to the number of training examples in each class. When using the Adam optimizer to train the network, the training algorithm is independent of the overall normalization of the class weights.

```
classWeights = 1./countcats(YTrain);
classWeights = classWeights'/mean(classWeights);
numClasses = numel(categories(YTrain));
```

```
timePoolSize = ceil(numHops/8);

dropoutProb = 0.2;
numF = 12;
layers = [
    imageInputLayer([numHops numBands])

    convolution2dLayer(3,numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,2*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer([timePoolSize,1])

    dropoutLayer(dropoutProb)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    weightedClassificationLayer(classWeights)];
```

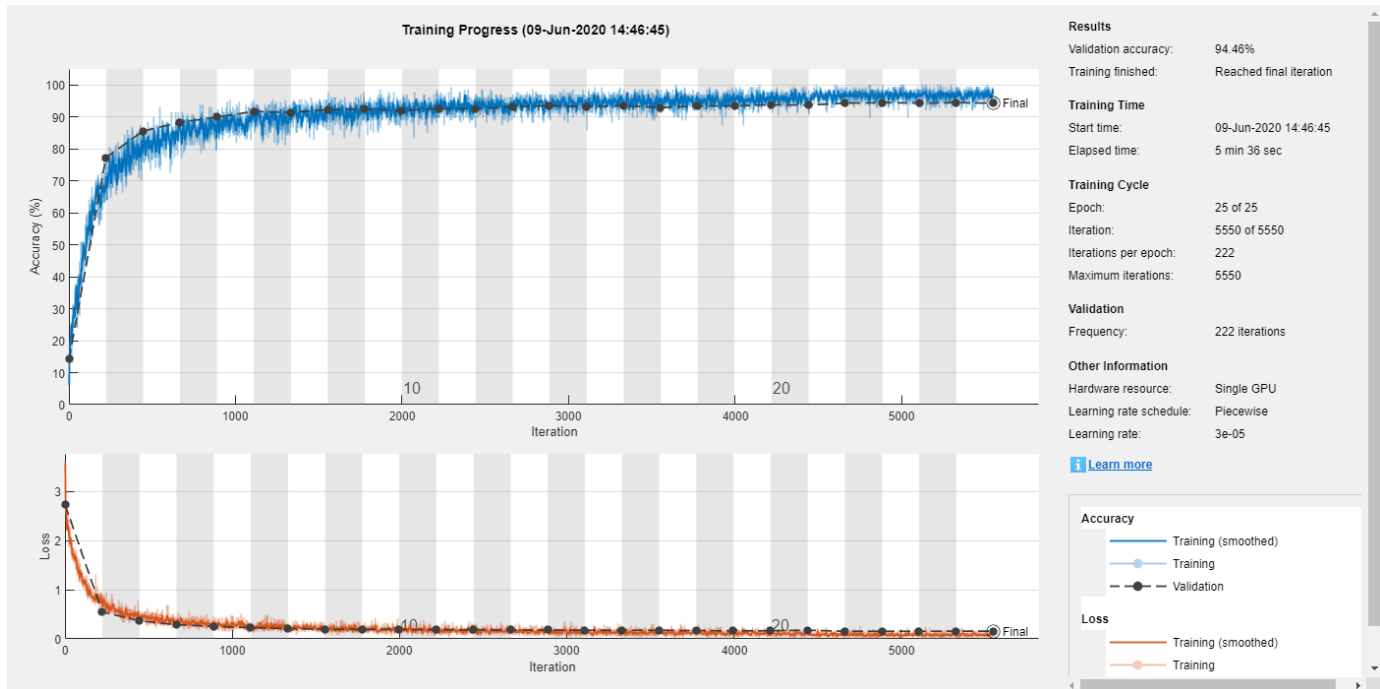
Train Network

Specify the training options. Use the Adam optimizer with a mini-batch size of 128. Train for 25 epochs and reduce the learning rate by a factor of 10 after 20 epochs.

```
miniBatchSize = 128;
validationFrequency = floor(numel(YTrain)/miniBatchSize);
options = trainingOptions('adam', ...
    'InitialLearnRate',3e-4, ...
    'MaxEpochs',25, ...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ValidationData',{XValidation,YValidation}, ...
    'ValidationFrequency',validationFrequency, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',0.1, ...
    'LearnRateDropPeriod',20);
```


Train the network. If you do not have a GPU, then training the network can take time.

```
trainedNet = trainNetwork(XTrain,YTrain,layers,options);
```



Evaluate Trained Network

Calculate the final accuracy of the network on the training set (without data augmentation) and validation set. The network is very accurate on this data set. However, the training, validation, and test data all have similar distributions that do not necessarily reflect real-world environments. This limitation particularly applies to the unknown category, which contains utterances of only a small number of words.

```
if reduceDataset
    load('commandNet.mat','trainedNet');
end
YValPred = classify(trainedNet,XValidation);
validationError = mean(YValPred ~= YValidation);
YTrainPred = classify(trainedNet,XTrain);
trainError = mean(YTrainPred ~= YTrain);
disp("Training error: " + trainError*100 + "%")
disp("Validation error: " + validationError*100 + "%")
```

```
Training error: 1.907%
Validation error: 5.5376%
```

Plot the confusion matrix. Display the precision and recall for each class by using column and row summaries. Sort the classes of the confusion matrix. The largest confusion is between unknown words and commands, *up* and *off*, *down* and *no*, and *go* and *no*.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5]);
cm = confusionchart(YValidation,YValPred);
cm.Title = 'Confusion Matrix for Validation Data';
cm.ColumnSummary = 'column-normalized';
```

```
cm.RowSummary = 'row-normalized';
sortClasses(cm, [commands,"unknown","background"])
```

Confusion Matrix for Validation Data

True Class	yes	252	2		1	1		1			1	1	2	96.6%	3.4%	
	no		256	1	2	2				1	3	5		94.8%	5.2%	
	up			245					7			5	3	94.2%	5.8%	
	down		13		238			1			8	4		90.2%	9.8%	
	left	2	1			242						1	1	98.0%	2.0%	
	right		1	1	1	2	249						2		97.3%	2.7%
	on			2				241	5			5	4	93.8%	6.2%	
	off			9		2		2	242		1			94.5%	5.5%	
	stop		1	3	1				2	236		1	2	95.9%	4.1%	
	go		11	4	1		2	1			232	5	4	89.2%	10.8%	
	unknown	2	10	8	7	3	8	9	6	4	15	771	7	90.7%	9.3%	
	background												600	100.0%		

98.4%	86.8%	89.7%	94.8%	96.0%	96.1%	94.5%	92.4%	97.9%	89.2%	96.4%	96.3%
1.6%	13.2%	10.3%	5.2%	4.0%	3.9%	5.5%	7.6%	2.1%	10.8%	3.6%	3.7%
yes	no	up	down	left	right	on	off	stop	go	unknown	background

Predicted Class

When working on applications with constrained hardware resources such as mobile applications, consider the limitations on available memory and computational resources. Compute the total size of the network in kilobytes and test its prediction speed when using a CPU. The prediction time is the time for classifying a single input image. If you input multiple images to the network, these can be classified simultaneously, leading to shorter prediction times per image. When classifying streaming audio, however, the single-image prediction time is the most relevant.

```
info = whos('trainedNet');
disp("Network size: " + info.bytes/1024 + " kB")

for i = 1:100
    x = randn([numHops,numBands]);
    tic
    [YPredicted,probs] = classify(trainedNet,x,"ExecutionEnvironment",'cpu');
    time(i) = toc;
end
disp("Single-image prediction time on CPU: " + mean(time(11:end))*1000 + " ms")

Network size: 286.7402 kB
Single-image prediction time on CPU: 2.5119 ms
```

References

[1] Warden P. "Speech Commands: A public dataset for single-word speech recognition", 2017. Available from https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz. Copyright Google 2017. The Speech Commands Dataset is licensed

under the Creative Commons Attribution 4.0 license, available here: <https://creativecommons.org/licenses/by/4.0/legalcode>.

Ambisonic Plugin Generation

This examples shows how to create ambisonic plugins using MATLAB® higher order ambisonic (HOA) demo functions. Ambisonics is a spatial audio technique which represents a three-dimensional sound field using spherical harmonics. This example contains an encoder plugin, a function to generate custom encoder plugins, a decoder plugin, and a function to generate custom decoder plugins. The customization of plugin generation enables a user to specify various ambisonic orders and various device lists for a given ambisonic configuration.

Background

Ambisonic encoding is the process of decomposing a sound field into spherical harmonics. The encoding matrix is the amount of spherical harmonics present at a specific device position. In mode-matching decoding, the decoding matrix is the pseudo-inverse of the encoding matrix. Ambisonic decoding is the process of reconstructing spherical harmonics into a sound field.

This example involves higher order ambisonics, which include traditional first-order ambisonics. In ambisonics, there is a relationship between the number of ambisonic channels and the ambisonic order:

$$\text{ambisonic_channels} = (\text{ambisonic_order} + 1)^2$$

For example: First-order ambisonics requires four audio channels while fourth-order ambisonics requires 25 audio channels.

The following conventions are supported

- ACN channel sequencing
- SN3D normalization
- azimuth from 0 to 360 degrees
- elevation from -90 to 90 degrees

The ambisonic design examples support up to seventh-order ambisonics with pseudo-inverse decoding.

Ambisonic Devices: Elements and Speakers

Ambisonic devices are divided into two groups: elements and speakers. Each device has an audio signal and metadata describing its position and operation. Elements correspond to multi-element microphone arrays, and speakers correspond to loudspeaker arrays for ambisonic playback.

The ambisonic encoder applies the ambisonic encoding matrix to raw audio from microphone elements. The position (azimuth, elevation) and deviceType of the microphone elements along with desired ambisonic order are needed to create the ambisonic encoding matrix.

The ambisonic decoder applies the ambisonic decoding matrix to ambisonic audio for playback on speakers. The position (azimuth, elevation) and deviceType of the speakers along with desired ambisonic order are needed to create the ambisonic decoding matrix.

Sound Field Representation

In order to capture, represent, or reproduce a sound field with ambisonics, the number of devices (elements or speakers) must be greater than or equal to the number of ambisonic channels.

For the encoding example, audio captured with a 32-channel spherical array microphone may be encoded up to fourth-order ambisonics (25 channels). For the decoding example, a loudspeaker array containing 64 speakers is configured for ambisonic playback up to seventh-order. If the playback content is fourth order ambisonics, then even though the array is set up for seventh-order, only fourth-order ambisonics is realized through the system.

```
number_devices >= number_ambisonic_channels
```

For an encoder, if the number of devices (elements) is less than the number of ambisonic channels, then audio from the device (elements) positions may be represented in ambisonics, but a sound field is not represented. One or more audio channels may be encoded into ambisonics in an effort to position sources in an ambisonic field. Each encoder represents the intensity of the sound field to be encoded at a specified device (element) location.

For a decoder, if the number of devices (speakers) is less than the number of ambisonic channels, the devices (speakers) do not fully reproduce a sound field at the specified ambisonic order. A sound field may be reproduced at a lower ambisonic order. For example, third-order ambisonics played on a speaker array with 10 speakers can be realized as a second-order (9 channel) system with an additional speaker for playback. Each decoder represents an intensity of the ambisonic field at the specified device (speaker) position.

pseudo-inverse decoding method

There are many decoding options, this example uses pseudo-inverse decoding, also known as mode matching. This decoding method favors regular-shaped device layouts. Other decoding methods may favor irregular-shaped device layouts.

DeviceType

The deviceType for encoders turns the device (element) encoding on or off for a particular element. The deviceType for decoders turns the device (speaker) decoding on or off for a particular speaker. If the deviceType vector is omitted, then the deviceTypes are set to 1 (on). The intention behind deviceType is to provide flexibility of padding encoder inputs or decoder outputs with off channels to fit an ambisonic encoder or decoder plugin into an environment with fixed channel counts such as an 8-, 16- or 32-channel audio bus.

For example: A second-order ambisonic encoder with 14 elements has 14 inputs and 9 outputs. If you add two more devices (elements) with deviceType 0 (off) to the encoder, then the encoder has 16 inputs and 9 outputs. A fourth-order ambisonic decoder with 29 devices (speakers) has 25 inputs and 29 outputs. If you add three more devices (speakers) with deviceType 0 (off) to the decoder, then the channel count becomes 25 inputs and 32 outputs.

When the deviceType is set to 0 (off), the azimuth and elevation for that channel are ignored; however, a value is still needed. It is recommended to set the azimuth and elevation to 0 degrees when the device types are set to 0 (off).

Ambisonic Encoder Plugin

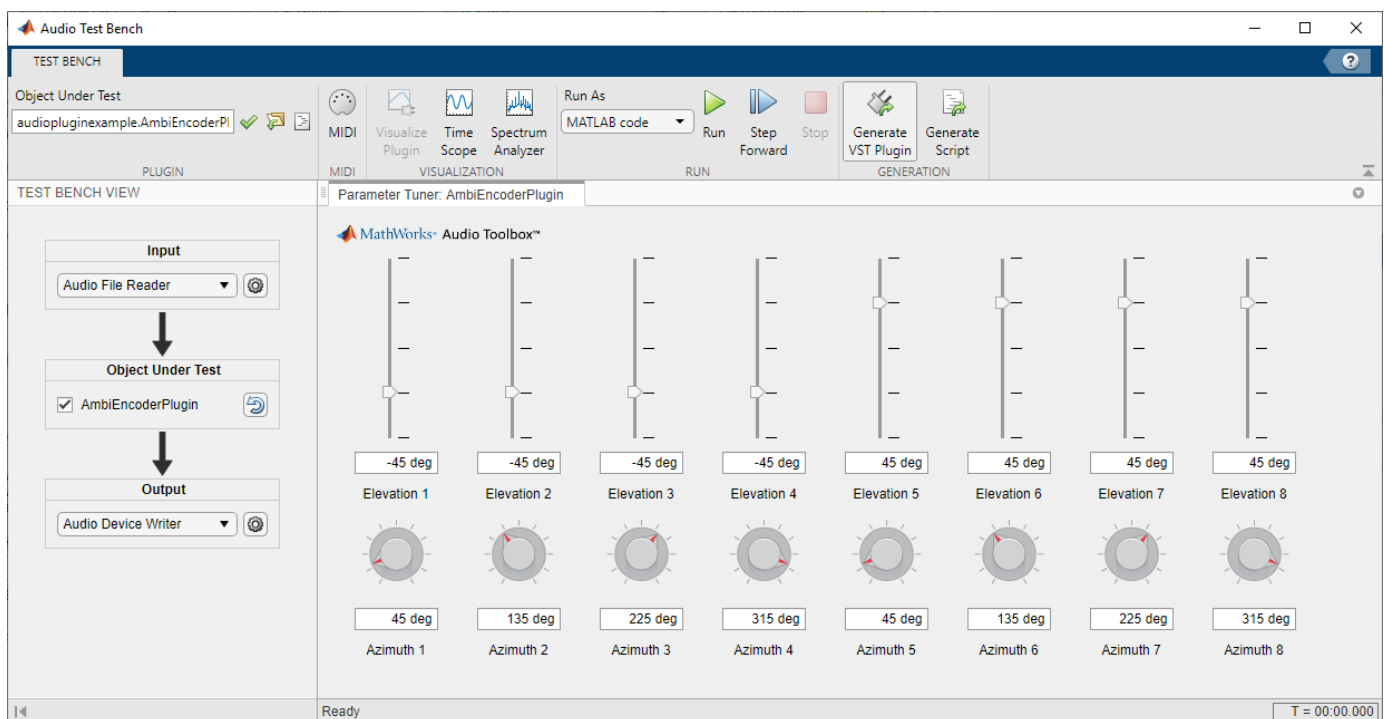
`audiopluginexample.AmbiEncoderPlugin` is built around the `audioexample.ambisonics.ambiencodemtrx` and `audioexample.ambisonics.ambiencode` functions. The number of devices (elements to be encoded) is the number of input channels of the encoder plugin. The ambisonic order determines the number of output channels of the encoder plugin.

`audioexample.ambisonics.ambiencodemtrx` generates the ambisonic encoder matrix from a given ambisonic order and a given device list. `audioexample.ambisonics.ambiencode` applies the ambisonic encoder matrix to raw audio resulting in ambisonic encoded audio. The formatting of the ambisonic audio may be specified with the `audioexample.ambisonics.ambiencode` function. The number of raw audio channels must equal the number of devices in the ambisonic encoder matrix.

The encoder plugin inherits directly from the `audioPlugin` base class. The plugin constructor calls `audioexample.ambisonics.ambiencodemtrx` to build the initial encoder matrix. The process function calls `audioexample.ambisonics.ambiencode` to apply the encoder matrix to the audio input. The output of the plugin is ambisonic encoded audio. The encoder matrix is recalculated only when a plugin property is modified which minimizes computations inside the process loop.

The plugin interface populates azimuth and elevation but not device type. The idea behind device type is to add off-channels to an encoder matrix to fit the matrix into a 8x-channel frame. For example: second-order has 9 channels, create a 16 channel encoder matrix, with the first 9 channels having device type of 1 (on) and the remaining 7 channels having device type of 0 (off).

```
audioTestBench(audiopluginexample.AmbiEncoderPlugin)
```



```
audioTestBench('-close')
```

[Inspect Code](#) | [Run Plugin](#) | [Generate Plugin](#)

Generate Custom Ambisonic Encoder Plugin

Generating ambisonic plugins can be an involved process. The `ambiGenerateEncoderPlugin` function streamlines the process of generating ambisonic encoder plugins. This function supports up to seventh-order ambisonics. Supported formats are 'acn-sn3d', 'acn-n3d', 'acn-fuma', 'acn-maxn', 'fuma-sn3d', 'fuma-n3d', 'fuma-fuma', 'fuma-maxn'. The function requires the following inputs:

- 1 name of the audioPlugin class
- 2 device list of encoder positions
- 3 ambisonic order
- 4 ambisonic format

```
% Provide a name for the audioPlugin class
name = 'myEncoderPlugin';

% Include a device list of element positions
device = [45 135 225 315 45 135 225 315; -45 -45 -45 -45 45 45 45 45];

% Specify the ambisonic order
order = 3;

% Specify the ambisonic format
format = 'acn-sn3d';
```

Run the function.

```
audioexample.ambisonics.ambiGenerateEncoderPlugin(name, device, order, format)
```

Once designed, the audio plugin can be validated, generated, and deployed to a third-party digital audio workstation (DAW).

Ambisonic Decoder Plugin

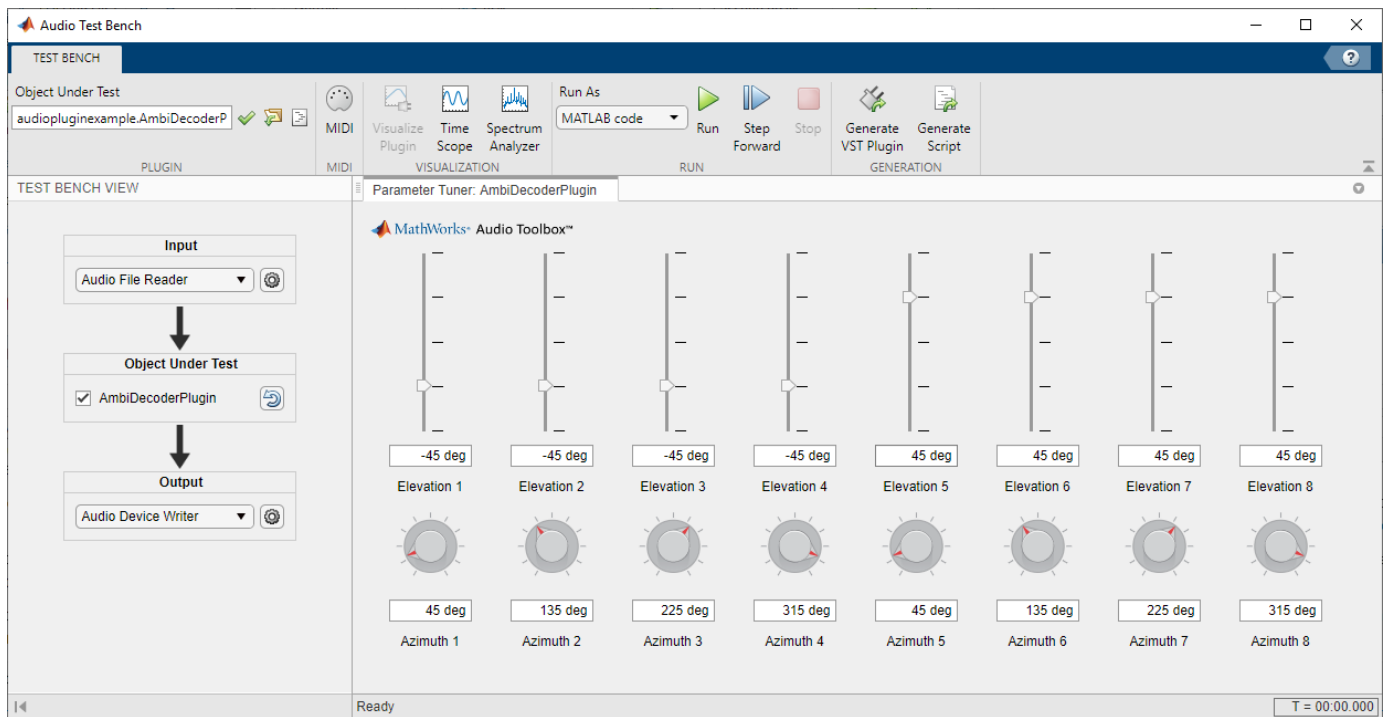
`audiopluginexample.AmbiDecoderPlugin` is built around the `audioexample.ambisonics.ambidecodemtx` and `audioexample.ambisonics.ambidecode` functions. The ambisonic order determines the number of input channels of the decoder plugin. The number of devices (speakers locations) is the number of output channels of the decoder plugin.

`audioexample.ambisonics.ambidecodemtx` generates the ambisonic decoder matrix from a given ambisonic order and a given device list. `audioexample.ambisonics.ambidecode` applies the ambisonic decoder matrix to ambisonic audio resulting in decoded audio. The formatting of the ambisonic audio may be specified with the `audioexample.ambisonics.ambidecode` function. `audioexample.ambisonics.ambidecode` determines the ambisonic order from the minimum of the ambisonic order of the input audio and the ambisonic order of the decoder matrix.

The decoder plugin inherits directly from the `audioPlugin` base class. The plugin constructor calls `audioexample.ambisonics.ambidecodemtx` to build the initial decoder matrix. The process function calls `audioexample.ambisonics.ambidecode` to apply the decoder matrix to the audio input. The output of the plugin is decoded audio. The decoder matrix is recalculated only when a plugin property is modified which minimizes computations inside the process loop.

The plugin interface populates azimuth and elevation but not device type. The idea behind device type is to add off-channels to an encoder matrix to fit the matrix into a 8x-channel frame. For example: second-order has 9 channels, create a 16 channel encoder matrix, with the first 9 channels having device type of 1 (on) and the remaining 7 channels having device type of 0 (off).

```
audioTestBench(audiopluginexample.AmbiDecoderPlugin)
```



```
audioTestBench(' -close')
```

[Inspect Code](#) | [Run Plugin](#) | [Generate Plugin](#)

Generate Custom Ambisonic Decoder Plugin

Generating ambisonic plugins can be an involved process. The `ambiGenerateDecoderPlugin` function streamlines the process of generating ambisonic decoder plugins. This function supports up to seventh-order ambisonics. Supported formats are 'acn-sn3d', 'acn-n3d', 'acn-fuma', 'acn-maxn', 'fuma-sn3d', 'fuma-n3d', 'fuma-fuma', 'fuma-maxn'. The function requires the following inputs:

- 1 name of the `audioPlugin` class
- 2 device list of decoder positions
- 3 ambisonic order
- 4 ambisonic format

```
% Provide a name for the audioPlugin class
```

```
name = 'myDecoderPlugin';
```

```
% Include a device list of speaker positions
```

```
device = [45 135 225 315 45 135 225 315; -45 -45 -45 -45 45 45 45 45];
```

```
% Specify the ambisonic order
```

```
order = 3;
```

```
% Specify the ambisonic format
```

```
format = 'acn-sn3d';
```

Run the function.

```
audioexample.ambisonics.ambiGenerateDecoderPlugin(name,device,order,format)
```


Once designed, the audio plugin can be validated, generated, and deployed to a third-party digital audio workstation (DAW).

See Also

“Ambisonic Binaural Decoding” on page 1-356

Related Topics

- “Audio Plugins in MATLAB”
- “Audio Plugin Example Gallery” on page 12-2
- “Audio Test Bench Walkthrough” on page 11-2

References

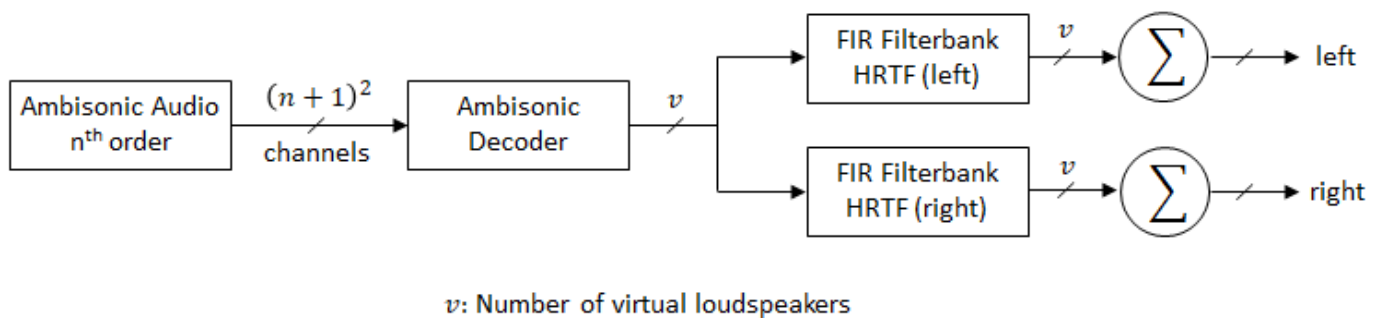
[1] Kronlachner, M. (2014). Spatial Transformations for the Alteration of Ambisonic Recordings (Master's thesis).

[2] <https://en.wikipedia.org/wiki/Ambisonics>

[3] https://en.wikipedia.org/wiki/Ambisonic_data_exchange_formats

Ambisonic Binaural Decoding

This example shows how to decode ambisonic audio into binaural audio using virtual loudspeakers. A virtual loudspeaker is a sound source positioned on the surface of a sphere, with the listener located at the center of the sphere. Each virtual loudspeaker has a pair of Head-Related Transfer Functions (HRTF) associated with it: one for the left ear and one for the right ear. The virtual loudspeaker locations along with the ambisonic order are used to calculate the ambisonic decoder matrix. The output of the decoder is filtered by the HRTFs corresponding to the virtual loudspeaker position. The signals from the left HRTFs are summed together and fed to the left ear and the signals from the right HRTFs are summed together and fed to the right ear. A block diagram of the audio signal flow is shown here.



Load the ARI HRTF Dataset

```
ARIDataset = load('ReferenceHRTF.mat');
```

Get the HRTF data in the required dimension of: [NumOfSourceMeasurements x 2 x LengthOfSamples]

```
hrtfData = ARIDataset.hrtfData;  
sourcePosition = ARIDataset.sourcePosition(:,[1,2]);
```

The ARI HRTF Databases used in this example is based on the work by Acoustics Research Institute <https://www.kfs.oew.ac.at/hrtf>. The HRTF data and source position in `ReferenceHRTF.mat` are from ARI NH2 subject.

The HRTF Databases by Acoustics Research Institute, Austrian Academy of Sciences are licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License: <https://creativecommons.org/licenses/by-sa/3.0/>.

Select Points from ARI HRTF Dataset

Now that the HRTF Dataset is loaded, determine which points to pick for virtual loudspeakers. This example picks random points distributed on the surface of a sphere and selects the points of the HRTF dataset closest to the picked points.

- 1 Pick random points from a spherical distribution
- 2 Compare sphere to points from the HRTF dataset
- 3 Pick the points with the shortest distance between them

```
% Create a sphere with a distribution of points  
nPoints = 24; % number of points to pick
```

```

rng(0); % seed random number generator
sphereAZ = 360*rand(1,nPoints);
sphereEL = rad2deg(acos(2*rand(1,nPoints)-1))-90;
pickedSphere = [sphereAZ' sphereEL'];

% Compare distributed points on the sphere to points from the HRTF dataset
pick = zeros(1, nPoints);
d = zeros(size(pickedSphere,1), size(sourcePosition,1));
for ii = 1:size(pickedSphere,1)
    for jj = 1:size(sourcePosition,1)
        % Calculate arc length
        d(ii,jj) = acos( ...
            sind(pickedSphere(ii,2))*sind(sourcePosition(jj,2)) + ...
            cosd(pickedSphere(ii,2))*cosd(sourcePosition(jj,2)) * ...
            cosd(pickedSphere(ii,1) - sourcePosition(jj,1)));
    end
    [~,Idx] = sort(d(ii,:)); % Sort points
    pick(ii) = Idx(1); % Pick the closest point
end

```

Create Ambisonic Decoder

Specify a desired ambisonic order and desired virtual loudspeaker source positions as inputs to the `audioexample.ambisonics.ambidecodemtrx` helper function. The function returns an ambisonics decoder matrix.

```

order = 7;
devices = sourcePosition(pick,:);
dmtrx = audioexample.ambisonics.ambidecodemtrx(order, devices);

```

Create HRTF Filters

Create an array of FIR filters to perform binaural HRTF filtering based on the position of the virtual loudspeakers.

```

FIR = cell(size(pickedSphere));
for ii = 1:length(pick)
    FIR{ii,1} = dsp.FrequencyDomainFIRFilter(hrtfData(:,pick(ii),1));
    FIR{ii,2} = dsp.FrequencyDomainFIRFilter(hrtfData(:,pick(ii),2));
end

```

Create Audio Input and Output Objects

Load the ambisonic audio file of helicopter sound and convert it to 48 kHz for compatibility with the HRTF dataset. Specify the ambisonic format of the audio file.

Create an audio file sampled at 48 kHz for compatibility with the HRTF dataset.

```

desiredFs = 48e3;
[audio,fs] = audioread('Heli_16ch_ACN_SN3D.wav');
audio = resample(audio,desiredFs,fs);
audiowrite('Heli_16ch_ACN_SN3D_48.wav',audio,desiredFs);

```

Specify the ambisonic format of the audio file. Set up the audio input and audio output objects.

```

format = 'acn-sn3d';
samplesPerFrame = 2048;
fileReader = dsp.AudioFileReader('Heli_16ch_ACN_SN3D_48.wav', ...

```

```
        'SamplesPerFrame',samplesPerFrame);  
deviceWriter = audioDeviceWriter('SampleRate',desiredFs);  
audioFiltered = zeros(samplesPerFrame,size(FIR,1),2);
```

Process Audio

```
while ~isDone(fileReader)  
    audioAmbi = fileReader();  
    audioDecoded = audioexample.ambisonics.ambidecode(audioAmbi, dmtrx, format);  
    for ii = 1:size(FIR,1)  
        audioFiltered(:,ii,1) = step(FIR{ii,1}, audioDecoded(:,ii)); % Left  
        audioFiltered(:,ii,2) = step(FIR{ii,2}, audioDecoded(:,ii)); % Right  
    end  
    audioOut = 10*squeeze(sum(audioFiltered,2)); % Sum at each ear  
    numUnderrun = deviceWriter(audioOut);  
end  
  
% Release resources  
release(fileReader)  
release(deviceWriter)
```

See Also

Ambisonic Plugin Generation Example

References

- [1] Kronlachner, M. (2014). Spatial Transformations for the Alteration of Ambisonic Recordings (Master's thesis).
- [2] Noisternig, Markus. et al. "A 3D Ambisonic Based Binaural Sound Reproduction System." Presented at 24th AES International Conference: Multichannel Audio, The New Reality, Alberta, June 2003.

Music Genre Classification Using Wavelet Time Scattering

This example shows how to classify the genre of a musical excerpt using wavelet time scattering and the audio datastore. In wavelet scattering, data is propagated through a series of wavelet transforms, nonlinearities, and averaging to produce low-variance representations of the data. These low-variance representations are then used as inputs to a classifier.

GTZAN Dataset

The data set used in this example is the GTZAN Genre Collection [7][8]. The data is provided as a zipped tar archive which is approximately 1.2 GB. The uncompressed data set requires about 3 GB of disk space. Extracting the compressed tar file from the link provided in the references creates a folder with ten subfolders. Each subfolder is named for the genre of music samples it contains. The genres are: blues, classical, country, disco, hiphop, jazz, metal, pop, reggae, and rock. There are 100 examples of each genre and each audio file consists of about 30 seconds of data sampled at 22050 Hz. In the original paper, the authors used a number of time-domain and frequency-domain features including mel-frequency cepstral (MFC) coefficients extracted from each music example and a Gaussian mixture model (GMM) classification to achieve an accuracy of 61 percent [7]. Subsequently, deep learning networks have been applied to this data. In most cases, these deep learning approaches consist of convolutional neural networks (CNN) with the MFC coefficients or spectrograms as the input to the deep CNN. These approaches have resulted in performance of around 84% [4]. An LSTM approach with spectrogram time slices resulted in 79% accuracy and time-domain and frequency-domain features coupled with an ensemble learning approach (AdaBoost) resulted in 82% accuracy on a test set [2][3]. Recently, a sparse representation machine learning approach achieved approximately 89% accuracy [6].

Wavelet Scattering Framework

The only parameters to specify in a wavelet time scattering framework are the duration of the time invariance, the number of wavelet filter banks, and the number of wavelets per octave. For most applications, cascading the data through two wavelet filter banks is sufficient. In this example, we use the default scattering framework which uses two wavelet filter banks. The first filter bank has 8 wavelets per octave and the second filter bank has 1 wavelet per octave. For this example, set the invariant scale to be 0.5 seconds, which corresponds to slightly more than 11,000 samples for the given sampling rate. Create the wavelet time scattering decomposition framework.

```
sf = waveletScattering('SignalLength',2^19,'SamplingFrequency',22050,...
    'InvarianceScale',0.5);
```

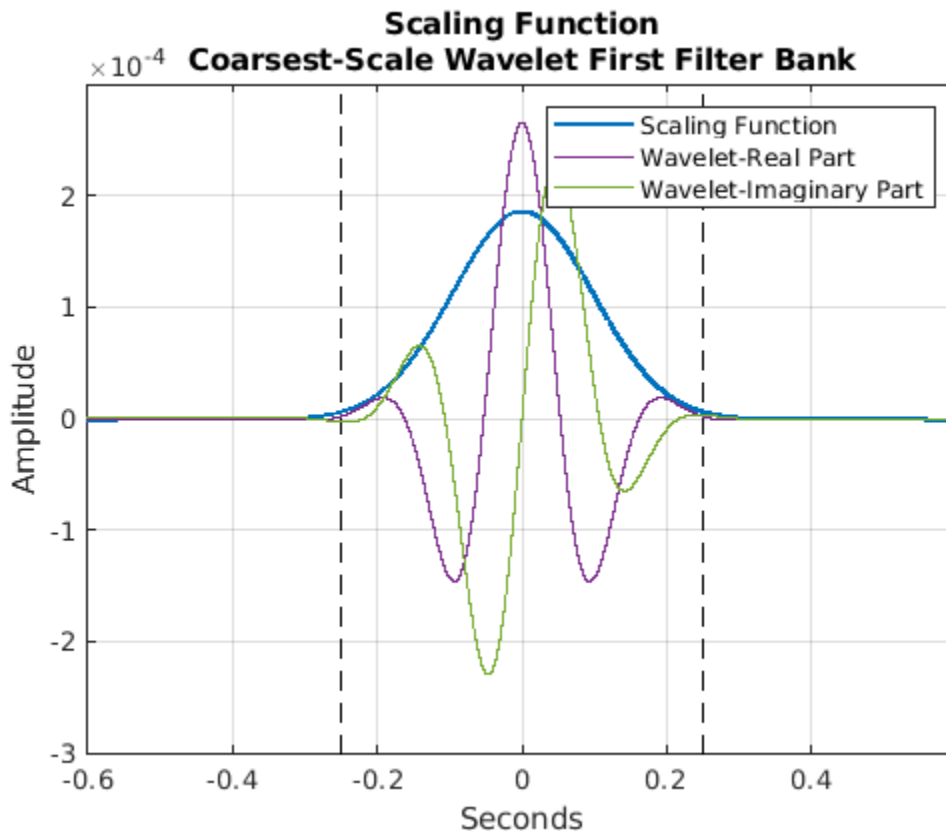
To understand the role of the invariance scale, obtain and plot the scaling filter in time along with the real and imaginary parts of the coarsest-scale wavelet from the first filter bank. Note that the time-support of the scaling filter is essentially 0.5 seconds as designed. Further, the time support of the coarsest-scale wavelet does not exceed the invariant scale of the wavelet scattering decomposition.

```
[fb,f,filterparams] = filterbank(sf);
phi = ifftshift(ifft(fb{1}.phift));
psi11 = ifftshift(ifft(fb{2}.psift(:,end)));
dt = 1/22050;
time = -2^18*dt:dt:2^18*dt-dt;
scalplt = plot(time,phi,'linewidth',1.5);
hold on
grid on
ylimits = [-3e-4 3e-4];
ylim(ylimits);
plot([-0.25 -0.25],ylimits,'k--');
```

```

plot([0.25 0.25],ylimits,'k--');
xlim([-0.6 0.6]);
xlabel('Seconds'); ylabel('Amplitude');
wavplt = plot(time,[real(psiL1) imag(psiL1)]);
legend([scalplt wavplt(1) wavplt(2)],{'Scaling Function','Wavelet-Real Part','Wavelet-Imaginary Part'});
title({'Scaling Function','Coarsest-Scale Wavelet First Filter Bank'})
hold off

```



Audio Datastore

The audio datastore enables you to manage collections of audio data files. For machine or deep learning, the audio datastore not only manages the flow of audio data from files and folders, the audio datastore also manages the association of labels with the data and provides the ability to randomly partition your data into different sets for training, validation, and testing. In this example, use the audio datastore to manage the GTZAN music genre collection. Recall each subfolder of the collection is named for the genre it represents. Set the 'IncludeSubFolders' property to true to instruct the audio datastore to use subfolders and set the 'LabelSource' property to 'foldernames' to create data labels based on the subfolder names. This example assumes the top-level directory is inside your MATLAB tempdir directory and is called 'genres'. Ensure that location is the correct path to the top-level data folder on your machine. The top-level data folder on your machine should contain ten subfolders each named for the ten genres and must only contain audio files corresponding to those genres.

```

location = fullfile(tempdir,'genres');
ads = audioDatastore(location,'IncludeSubFolders',true,...
    'LabelSource','foldernames');

```

Run the following to obtain a count of the musical genres in the data set.

```
countEachLabel(ads)
```

```
ans =
```

```
10×2 table
```

Label	Count
blues	100
classical	100
country	100
disco	100
hiphop	100
jazz	100
metal	100
pop	100
reggae	100
rock	100

As previously stated, there are 10 genres with 100 files each.

Training and Test Sets

Create training and test sets to develop and test our classifier. We use 80% of the data for training and hold out the remaining 20% for testing. The `shuffle` function of the audio datastore randomly shuffles the data. Do this prior to splitting the data by label to randomize the data. In this example, we set the random number generator seed for reproducibility. Use the audio datastore `splitEachLabel` function to perform the 80-20 split. `splitEachLabel` ensures that all classes are equally represented.

```
rng(100);
ads = shuffle(ads);
[adsTrain,adsTest] = splitEachLabel(ads,0.8);
countEachLabel(adsTrain)
countEachLabel(adsTest)
```

```
ans =
```

```
10×2 table
```

Label	Count
blues	80
classical	80
country	80
disco	80
hiphop	80
jazz	80
metal	80
pop	80
reggae	80

```
        rock            80

ans =

10x2 table

    Label    Count
    _____  _____
    blues      20
    classical  20
    country    20
    disco      20
    hiphop     20
    jazz       20
    metal      20
    pop        20
    reggae     20
    rock       20
```

You see that there are 800 records in the training data as expected and 200 records in the test data. Additionally, there are 80 examples of each genre in the training set and 20 examples of each genre in the test set.

`audioDatastore` works with MATLAB tall arrays. Create tall arrays for both the training and test sets. Depending on your system, the number of workers in the parallel pool MATLAB creates may be different.

```
Ttrain = tall(adsTrain);
Ttest = tall(adsTest);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

To obtain the scattering features, define a helper function, `helperscatfeatures`, that obtains the natural logarithm of the scattering features for 2^{19} samples of each audio file and subsamples the number of scattering windows by 8. The source code for `helperscatfeatures` is listed in the appendix. We will compute the wavelet scattering features for both the training and test data.

```
scatteringTrain = cellfun(@(x)helperscatfeatures(x,sf),Ttrain,'UniformOutput',false);
scatteringTest = cellfun(@(x)helperscatfeatures(x,sf),Ttest,'UniformOutput',false);
```

Compute the scattering features on the training data and bundle all the features together in a matrix. This process takes several minutes.

```
TrainFeatures = gather(scatteringTrain);
TrainFeatures = cell2mat(TrainFeatures);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 3 min 40 sec
Evaluation completed in 3 min 40 sec
```

Repeat this process for the test data.

```
TestFeatures = gather(scatteringTest);
TestFeatures = cell2mat(TestFeatures);
```



```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 56 sec
Evaluation completed in 56 sec
```

Each row of `TrainFeatures` and `TestFeatures` is one scattering time window across the 341 paths in the scattering transform of each audio signal. For each music sample, we have 32 such time windows. Accordingly, the feature matrix for the training data is 25600-by-341. The number of rows is equal to the number of training examples (800) multiplied by the number of scattering windows per example (32). Similarly, the scattering feature matrix for the test data is 6400-by-341. There are 200 test examples and 32 windows per example. Create a genre label for each of the 32 windows in the wavelet scattering feature matrix for the training data.

```
numTimeWindows = 32;
trainLabels = adsTrain.Labels;
numTrainSignals = numel(trainLabels);
trainLabels = repmat(trainLabels,1,numTimeWindows);
trainLabels = reshape(trainLabels',numTrainSignals*numTimeWindows,1);
```

Repeat the process for the test data.

```
testLabels = adsTest.Labels;
numTestSignals = numel(testLabels);
testLabels = repmat(testLabels,1,numTimeWindows);
testLabels = reshape(testLabels',numTestSignals*numTimeWindows,1);
```

In this example, use a multi-class support vector machine (SVM) classifier with a cubic polynomial kernel. Fit the SVM to the training data.

```
template = templateSVM(...
    'KernelFunction', 'polynomial', ...
    'PolynomialOrder', 3, ...
    'KernelScale', 'auto', ...
    'BoxConstraint', 1, ...
    'Standardize', true);
Classes = {'blues','classical','country','disco','hiphop','jazz',...
    'metal','pop','reggae','rock'};
classificationSVM = fitcecoc(...
    TrainFeatures, ...
    trainLabels, ...
    'Learners', template, ...
    'Coding', 'onevsone', 'ClassNames', categorical(Classes));
```

Test Set Prediction

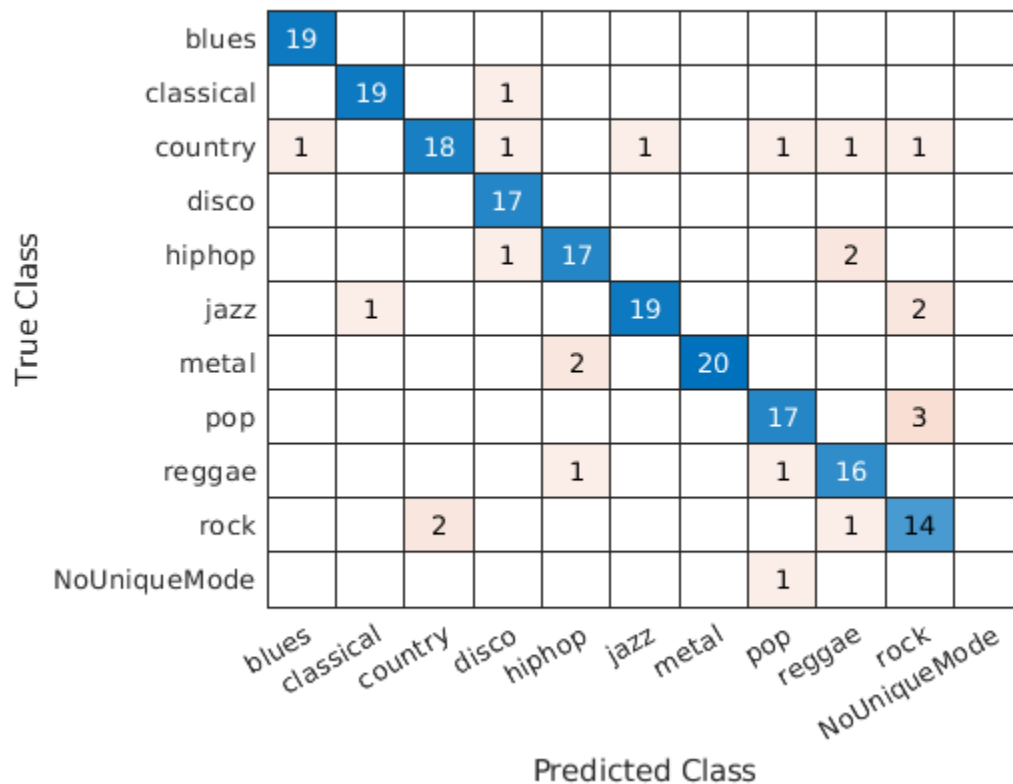
Use the SVM model fit to the scattering transforms of the training data to predict the music genres for the test data. Recall there are 32 time windows for each signal in the scattering transform. Use a simple majority vote to predict the genre. The helper function `helperMajorityVote` obtains the mode of the genre labels over all 32 scattering windows. If there is no unique mode, `helperMajorityVote` returns a classification error indicated by `'NoUniqueMode'`. This results in an extra column in the confusion matrix. The source code for `helperMajorityVote` is listed in the appendix.

```
predLabels = predict(classificationSVM,TestFeatures);
[TestVotes,TestCounts] = helperMajorityVote(predLabels,adsTest.Labels,categorical(Classes));
testAccuracy = sum(eq(TestVotes,adsTest.Labels))/numTestSignals*100;
```

The test accuracy, `testAccuracy`, is 88 percent. This accuracy is comparable with the state of the art of the GTZAN dataset.

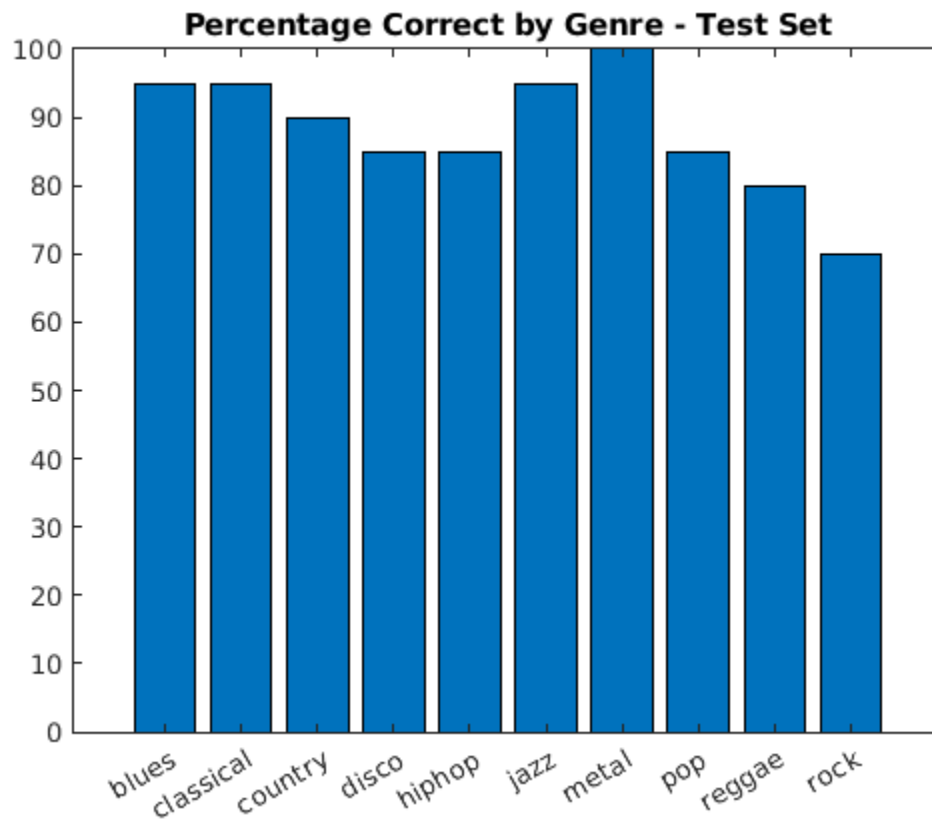
Display the confusion matrix to inspect the genre-by-genre accuracy rates. Recall there are 20 examples in each class.

```
confusionchart(TestVotes,adsTest.Labels)
```



The diagonal of the confusion matrix plot shows that the classification accuracies for the individual genres is quite good in general. Extract these genre accuracies and plot separately.

```
cm = confusionmat(TestVotes,adsTest.Labels);
cm(:,end) = [];
genreAccuracy = diag(cm)./20*100;
figure;
bar(genreAccuracy)
set(gca,'XTickLabels',Classes);
xtickangle(gca,30);
title('Percentage Correct by Genre - Test Set');
```



Summary

This example demonstrated the use of wavelet time scattering and the audio datastore in music genre classification. In this example, wavelet time scattering achieved a classification accuracy comparable to state of the art performance for the GTZAN dataset. As opposed to other approaches requiring the extraction of a number of time-domain and frequency-domain features, wavelet scattering only required the specification of a single parameter, the scale of the time invariant. The audio datastore enabled us to efficiently manage the transfer of a large dataset from disk into MATLAB and permitted us to randomize the data and accurately retain genre membership of the randomized data through the classification workflow.

References

- 1 Anden, J. and Mallat, S. 2014. Deep scattering spectrum. *IEEE Transactions on Signal Processing*, Vol. 62, 16, pp. 4114-4128.
- 2 Bergstra, J., Casagrande, N., Erhan, D., Eck, D., and Kegl, B. Aggregate features and AdaBoost for music classification. *Machine Learning*, Vol. 65, Issue 2-3, pp. 473-484.
- 3 Irvin, J., Chartock, E., and Hollander, N. 2016. Recurrent neural networks with attention for genre classification. <https://www.semanticscholar.org/paper/Recurrent-Neural-Networks-with-Attention-for-Genre-Irvin/6da301817851f19107447e4c72e682e3f183ae8a>
- 4 Li, T., Chan, A.B., and Chun, A. 2010. Automatic musical pattern feature extraction using convolutional neural network. *International Conference Data Mining and Applications*.
- 5 Mallat, S. 2012. Group invariant scattering. *Communications on Pure and Applied Mathematics*, Vol. 65, 10, pp. 1331-1398.

- 6 Panagakis, Y., Kotropoulos, C.L., and Arce, G.R. 2014. Music genre classification via joint sparse low-rank representation of audio features. *IEEE Transactions on Audio, Speech, and Language Processing*, 22, 12, pp. 1905-1917.
- 7 Tzanetakis, G. and Cook, P. 2002. Music genre classification of audio signals. *IEEE Transactions on Speech and Audio Processing*, Vol. 10, No. 5, pp. 293-302.
- 8 *GTZAN Genre Collection*. <http://marsyas.info/downloads/datasets.html>

Appendix -- Supporting Functions

helperMajorityVote -- This function returns the mode of the class labels predicted over a number of feature vectors. In wavelet time scattering, we obtain a class label for each time window. If no unique mode is found a label of 'NoUniqueMode' is returned to denote a classification error.

```
function [ClassVotes,ClassCounts] = helperMajorityVote(predLabels,origLabels,classes)
% This function is in support of wavelet scattering examples only. It may
% change or be removed in a future release.

% Make categorical arrays if the labels are not already categorical
predLabels = categorical(predLabels);
origLabels = categorical(origLabels);
% Expects both predLabels and origLabels to be categorical vectors
Npred = numel(predLabels);
Norig = numel(origLabels);
Nwin = Npred/Norig;
predLabels = reshape(predLabels,Nwin,Norig);
ClassCounts = countcats(predLabels);
[mxcount,idx] = max(ClassCounts);
ClassVotes = classes(idx);
% Check for any ties in the maximum values and ensure they are marked as
% error if the mode occurs more than once
modecnt = modecount(ClassCounts,mxcount);
ClassVotes(modecnt>1) = categorical({'NoUniqueMode'});
ClassVotes = ClassVotes(:);

%-----
function modecnt = modecount(ClassCounts,mxcount)
    modecnt = Inf(size(ClassCounts,2),1);
    for nc = 1:size(ClassCounts,2)
        modecnt(nc) = histc(ClassCounts(:,nc),mxcount(nc));
    end
end
end
```

helperscatfeatures - This function returns the wavelet time scattering feature matrix for a given input signal. In this case, we use the natural logarithm of the wavelet scattering coefficients. The scattering feature matrix is computed on 2^{19} samples of a signal. The scattering features are subsampled by a factor of 8.

```
function features = helperscatfeatures(x,sf)
% This function is in support of wavelet scattering examples only. It may
% change or be removed in a future release.

features = featureMatrix(sf,x(1:2^19),'Transform','log');
```

```
features = features(:,1:8:end)';  
end
```

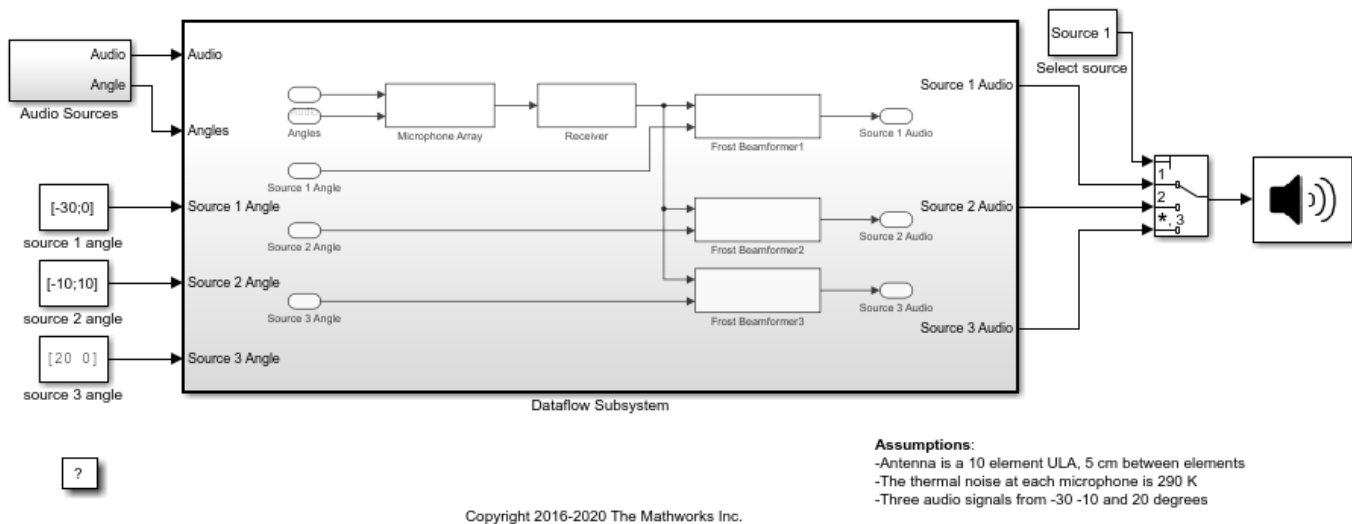
Multicore Simulation of Acoustic Beamforming Using a Microphone Array

This example shows how to beamform signals received by an array of microphones to extract a desired speech signal in a noisy environment. It uses the dataflow domain in Simulink® to partition the data-driven portions of the system into multiple threads and thereby improving the performance of the simulation by executing it on your desktop's multiple cores.

Introduction

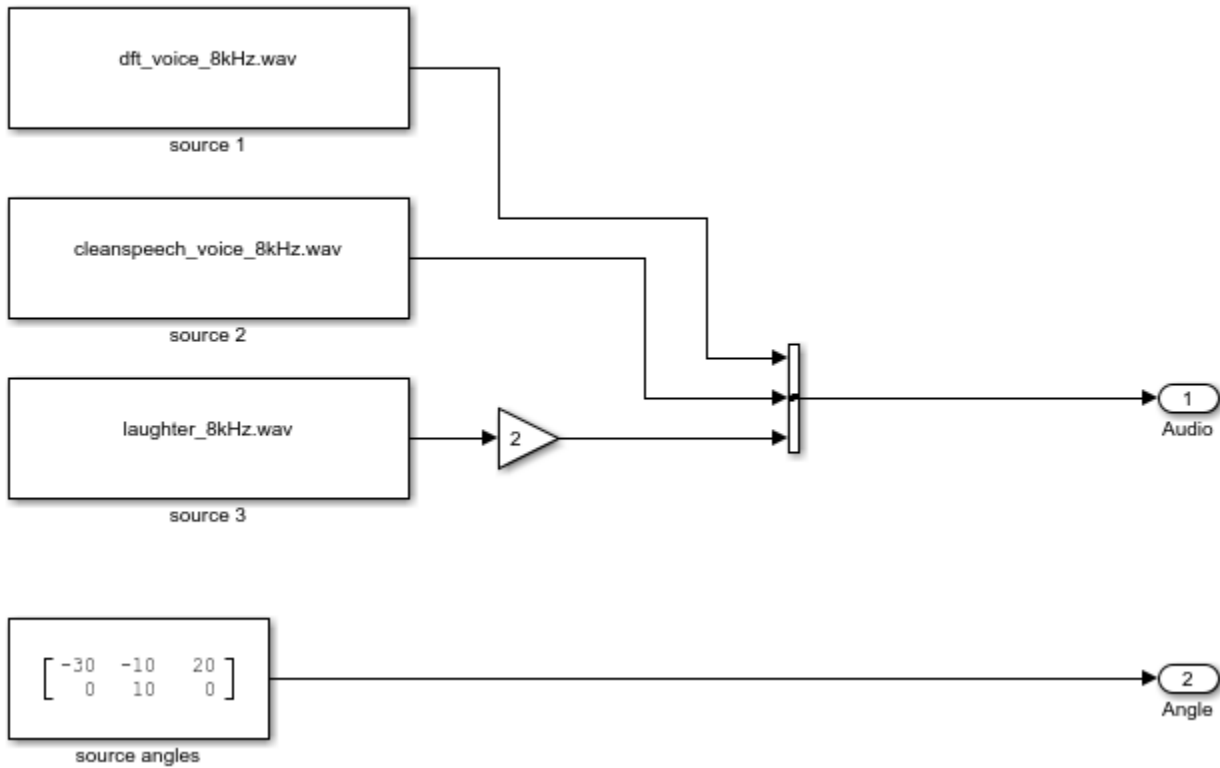
The model simulates receiving three audio signals from different directions on a 10-element uniformly linear microphone array (ULA). After the addition of thermal noise at the receiver, beamforming is applied and the result played on a sound device.

Acoustic Beamforming using Microphone Arrays



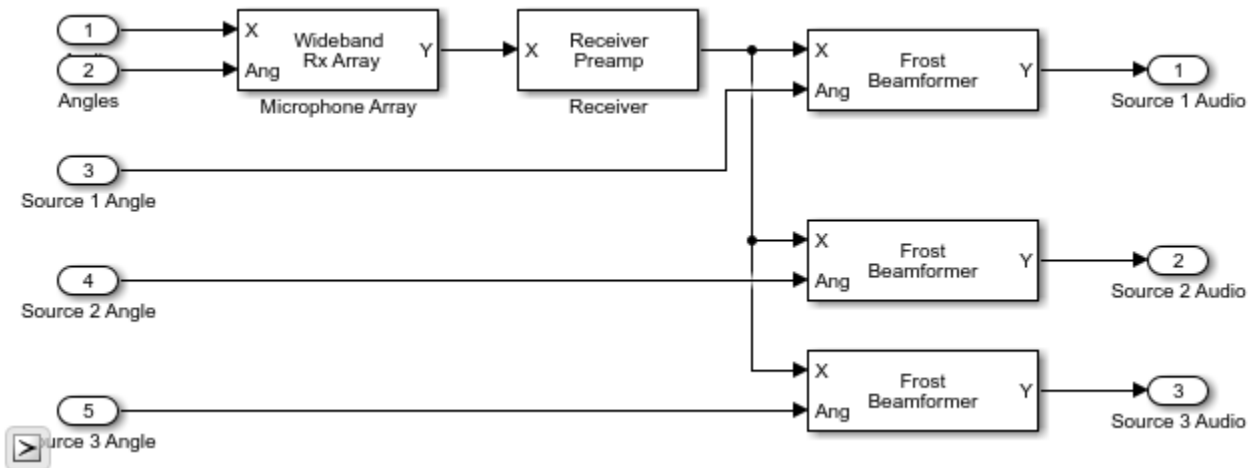
Received Audio Simulation

The Audio Sources subsystem reads from audio files and specifies the direction for each audio source. The Wideband Rx Array block simulates receiving audio signals at the ULA. The first input to the Wideband Rx Array block is a 1000x3 matrix, where the three columns of the input correspond to the three audio sources. The second input (Ang) specifies the incident direction of the signals. The first row of Ang specifies the azimuth angle in degrees for each signal and the second row specifies the elevation angle in degrees for each signal. The output of this block is a 1000x10 matrix. Each column of the output corresponds to the audio recorded at each element of the microphone array. The microphone array's configuration is specified in the Sensor Array tab of the block dialog panel. The Receiver Preamp block adds white noise to the received signals.



Beamforming

There are three Frost Beamformer blocks that perform beamforming on the matrix passed via the input port X along the direction specified by the input port Ang. Each of the three beamformers steers their beam towards one of the three sources. The output of the beamformer is played in the Audio Device Writer block. Different sources can be selected using the Select Source block.

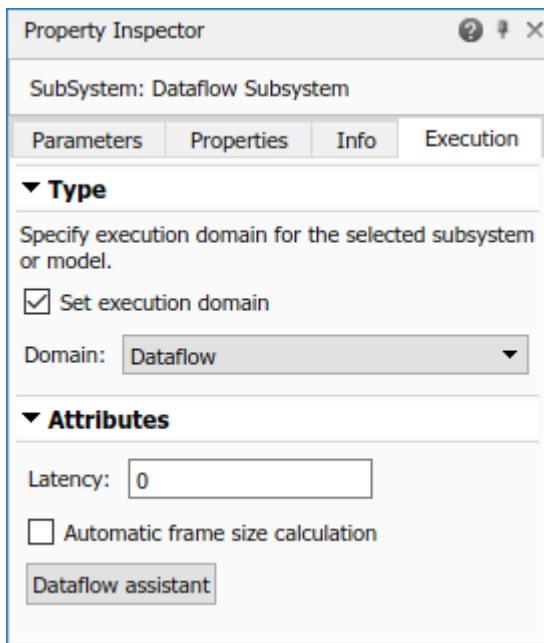


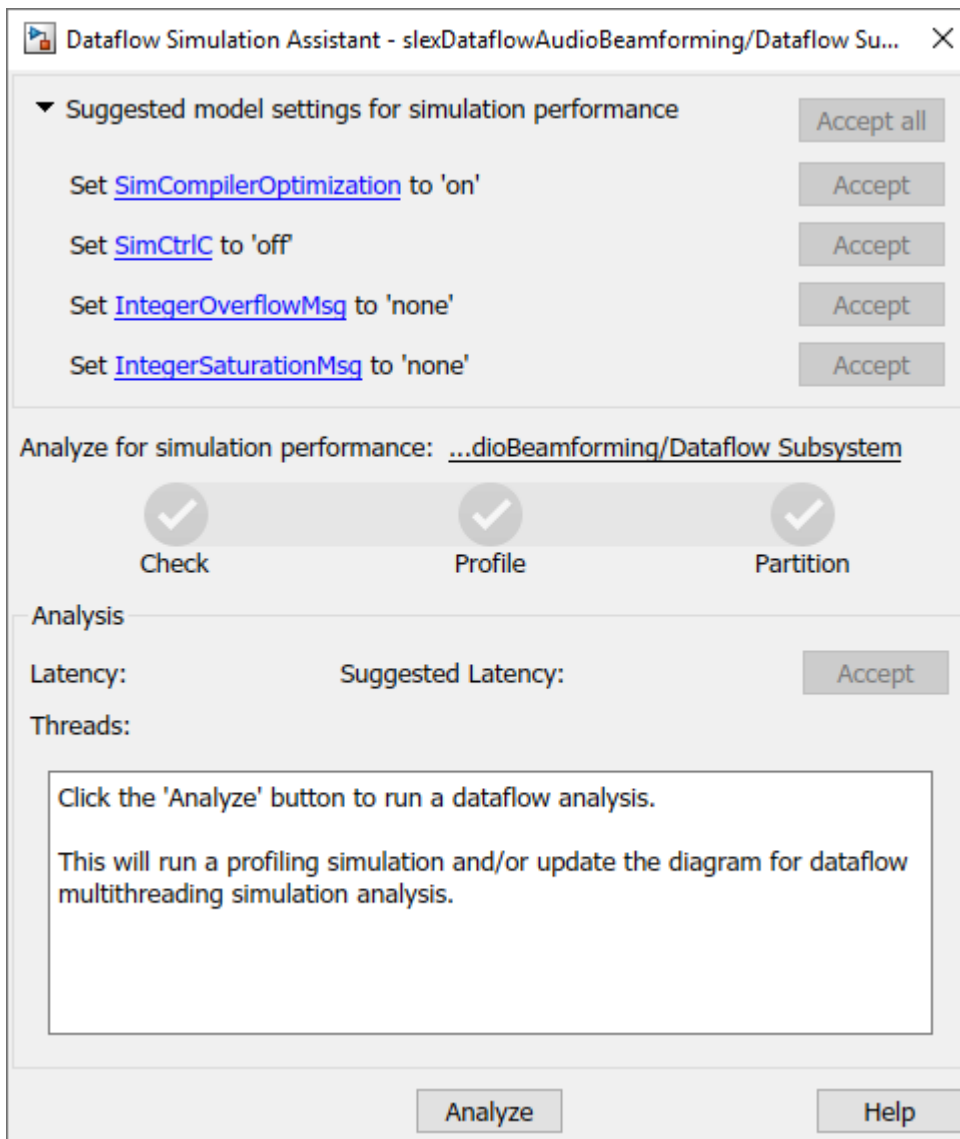
Improve Simulation Performance Using Multithreading

This example can use the dataflow domain in Simulink to automatically partition the data-driven portions of the system into multiple threads and thereby improving the performance of the simulation by executing it on your desktop's multiple cores. To learn more about dataflow and how to run Simulink models using multiple threads, see “Multicore Execution using Dataflow Domain”.

Setting up Dataflow Subsystem

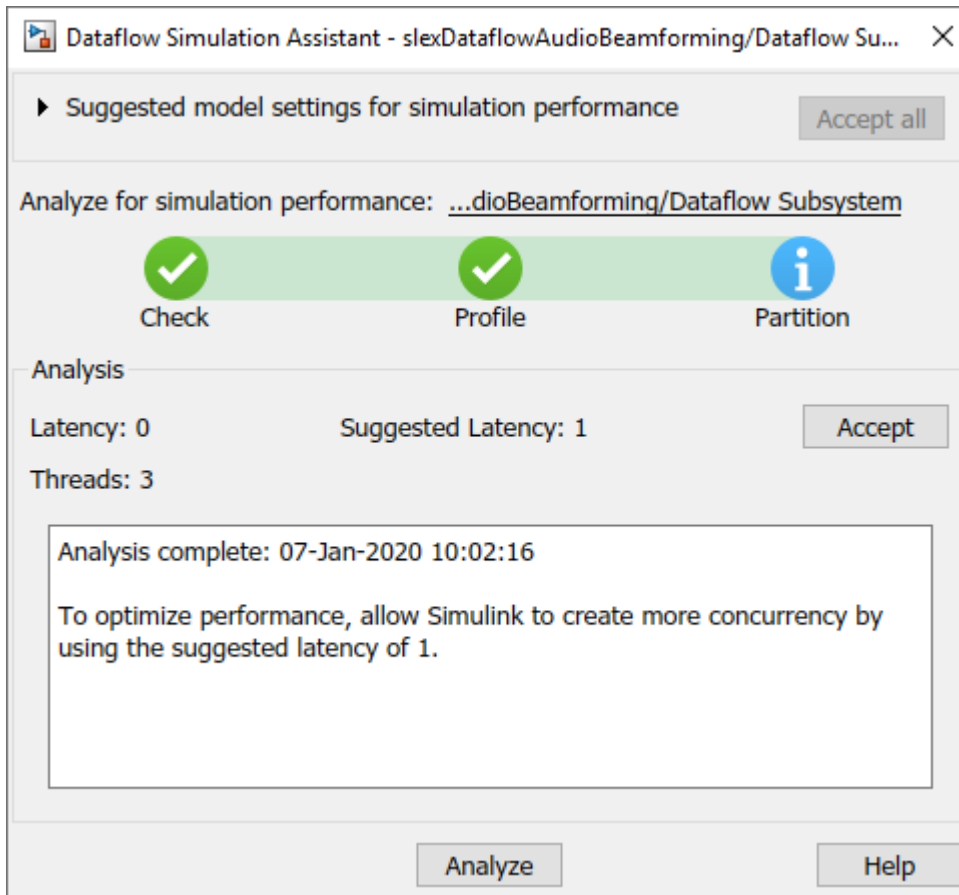
In Simulink, you specify dataflow as the execution domain for a subsystem by setting the Domain parameter to Dataflow using Property Inspector. Dataflow domains automatically partition your model and simulate the system using multiple threads for better simulation performance. Once you set the Domain parameter to Dataflow, you can use Dataflow Simulation Assistant to analyze your model to get better performance. You can open the Dataflow Simulation Assistant by clicking on the **Dataflow assistant** button below the **Automatic frame size calculation** parameter in Property Inspector.





Analyzing Concurrency in Dataflow Subsystem

The Dataflow Simulation Assistant suggests changing model settings for optimal simulation performance. To accept the proposed model settings, next to **Suggested model settings for simulation performance**, click **Accept all**. Alternatively, you can expand the section to change the settings individually. In this example the model settings are already optimal. In the Dataflow Simulation Assistant, click the **Analyze** button to start the analysis of the dataflow domain for simulation performance. Once the analysis is finished, the Dataflow Simulation Assistant shows how many threads the dataflow subsystem will use during simulation.



For this model the assistant shows three threads because the three Frost Beamformer blocks are computationally intensive and they can run in parallel. However, the three Frost Beamformer blocks depend on the Microphone Array and Receiver blocks to finish before they start execution. Concurrency can be increased for this model by using pipeline delays between the beamformer blocks and the source simulation blocks. Dataflow Simulation Assistant shows the recommended number of pipeline delays as **Suggested Latency**. For this model, the suggested latency is one. Click the **Accept** button next to **Suggested Latency** in the Dataflow Simulation Assistant to use the recommended latency for the Dataflow Subsystem.

Multicore Simulation Performance

To measure performance improvement gained by using dataflow, compare execution time of the model with and without dataflow. The Audio Device Writer runs in real time and limits the simulation speed of the model to real time. Comment out the Audio Device Writer block when measuring execution time. On a Windows desktop computer with Intel® Xeon® CPU W-2133 @ 3.6 GHz 6 Cores 12 Threads processor this model using dataflow domain executes 1.8x times faster compared to original model.

Summary

This example showed how to beamform signals received by an array of microphones to extract a desired speech signal in a noisy environment. It also shows how to use the dataflow domain to

automatically partition the data-driven part of the model into concurrent execution threads and run the model using multiple threads.

Convert MIDI Files into MIDI Messages

This example shows how to convert ordinary MIDI files into MIDI message representation using Audio Toolbox™. In this example, you:

- 1 Read a binary MIDI file into the MATLAB® workspace.
- 2 Convert the MIDI file data into `midimsg` objects.
- 3 Play the MIDI messages to your sound card using a simple synthesizer.

For more information about interacting with MIDI devices using MATLAB, see “MIDI Device Interface” on page 7-2. To learn more about MIDI in general, consult The MIDI Manufacturers Association.

Introduction

MIDI files contain MIDI messages, timing information, and metadata about the encoded music. This example shows how to extract MIDI messages and timing information. To simplify the code, this example ignores metadata. Because metadata includes information like time signature and tempo, this example assumes the MIDI file is in 4/4 time at 120 beats per minute (BPM).

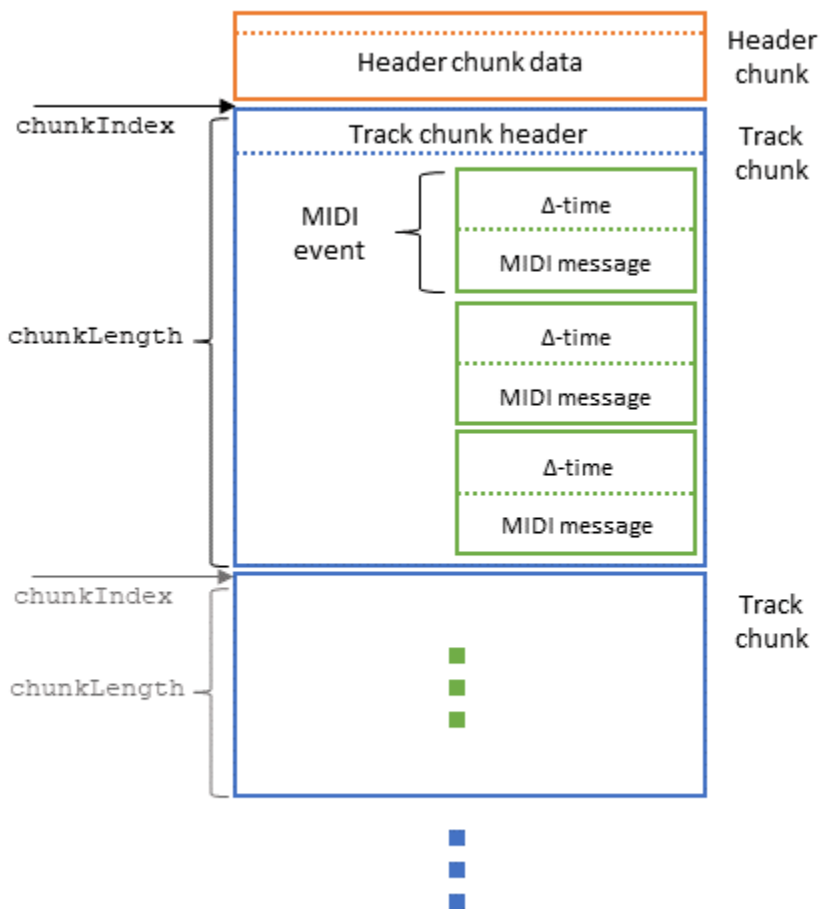
Read MIDI File

Read a MIDI file using the `fread` function. The `fread` function returns a vector of bytes, represented as integers.

```
readme = fopen('CmajorScale.mid');  
[readOut, byteCount] = fread(readme);  
fclose(readme);
```

Convert MIDI Data into `midimsg` Objects

MIDI files have *header chunks* and *track chunks*. Header chunks provide basic information required to interpret the rest of the file. MIDI files always start with a header chunk. Track chunks come after the header chunk. Track chunks provide the MIDI messages, timing information, and metadata of the file. Each track chunk has a track chunk header that includes the length of the track chunk. The track chunk contains MIDI events after the track chunk header. Every MIDI event has a delta-time and a MIDI message.



Parse MIDI Header Chunk

The MIDI header chunk includes the timing division of the file. The timing division determines how to interpret the resolution of ticks in the MIDI file. Ticks are the unit of time used to set timestamps for MIDI files. A MIDI file with more ticks per unit time has MIDI messages with more granular time stamps. *Timing division does not determine tempo*. MIDI files specify timing division either by ticks per quarter note or frames per second. This example assumes the MIDI timing division is in ticks per quarter note.

The `fread` function reads binary files byte-by-byte, but the timing division is stored as a 16-bit (2-byte) value. To evaluate multiple bytes as one value, use the `polyval` function. A vector of bytes can be evaluated as a polynomial where x is set at 256. For example, the vector of bytes `[1 2 3]` can be evaluated as:

$$1 \cdot 256^2 + 2 \cdot 256^1 + 3 \cdot 256^0$$

```
% Concatenate ticksPerQNote from 2 bytes
ticksPerQNote = polyval(readOut(13:14),256);
```

Parse MIDI Track Chunk

The MIDI track chunk contains a header and MIDI events. The track chunk header contains the length of the track chunk. The rest of the track chunk contains one or more MIDI events.

All MIDI events have two main components:

- A delta-time value—The time difference in ticks between the previous MIDI track event and the current one
- A MIDI message—The raw data of the MIDI track event

To parse MIDI track events sequentially, construct a loop within a loop. In the outer loop, parse track chunks, iterating by `chunkIndex`. In the inner loop, parse MIDI events, iterating by a pointer `ptr`.

To parse MIDI track events:

- Read the delta-time value at a pointer.
- Increment the pointer to the beginning of the MIDI message.
- Read the MIDI message and extract the relevant data.
- Add the MIDI message to a MIDI message array.

Display the MIDI message array when complete.

```
% Initialize values
chunkIndex = 14;      % Header chunk is always 14 bytes
ts = 0;               % Timestamp - Starts at zero
BPM = 120;
msgArray = [];

% Parse track chunks in outer loop
while chunkIndex < byteCount

    % Read header of track chunk, find chunk length
    % Add 8 to chunk length to account for track chunk header length
    chunkLength = polyval(readOut(chunkIndex+(5:8)),256)+8;

    ptr = 8+chunkIndex;      % Determine start for MIDI event parsing
    statusByte = -1;         % Initialize statusByte. Used for running status support

    % Parse MIDI track events in inner loop
    while ptr < chunkIndex+chunkLength
        % Read delta-time
        [deltaTime,deltaLen] = findVariableLength(ptr,readOut);
        % Push pointer to beginning of MIDI message
        ptr = ptr+deltaLen;

        % Read MIDI message
        [statusByte,messageLen,message] = interpretMessage(statusByte,ptr,readOut);
        % Extract relevant data - Create midimsg object
        [ts,msg] = createMessage(message,ts,deltaTime,ticksPerQNote,BPM);

        % Add midimsg to msgArray
        msgArray = [msgArray;msg];
        % Push pointer to next MIDI message
        ptr = ptr+messageLen;
    end
end
```

```

    % Push chunkIndex to next track chunk
    chunkIndex = chunkIndex+chunkLength;
end
disp(msgArray)

MIDI message:
NoteOn      Channel: 1  Note: 60  Velocity: 127  Timestamp: 0  [ 90 3C 7F ]
NoteOff     Channel: 1  Note: 60  Velocity: 0    Timestamp: 0.5 [ 80 3C 00 ]
NoteOn      Channel: 1  Note: 62  Velocity: 127  Timestamp: 0.5 [ 90 3E 7F ]
NoteOff     Channel: 1  Note: 62  Velocity: 0    Timestamp: 1  [ 80 3E 00 ]
NoteOn      Channel: 1  Note: 64  Velocity: 127  Timestamp: 1  [ 90 40 7F ]
NoteOff     Channel: 1  Note: 64  Velocity: 0    Timestamp: 1.5 [ 80 40 00 ]
NoteOn      Channel: 1  Note: 65  Velocity: 127  Timestamp: 1.5 [ 90 41 7F ]
NoteOff     Channel: 1  Note: 65  Velocity: 0    Timestamp: 1.75 [ 80 41 00 ]
NoteOn      Channel: 1  Note: 67  Velocity: 127  Timestamp: 2  [ 90 43 7F ]
NoteOff     Channel: 1  Note: 67  Velocity: 0    Timestamp: 2.5 [ 80 43 00 ]
NoteOn      Channel: 1  Note: 69  Velocity: 127  Timestamp: 2.5 [ 90 45 7F ]
NoteOff     Channel: 1  Note: 69  Velocity: 0    Timestamp: 3  [ 80 45 00 ]
NoteOn      Channel: 1  Note: 71  Velocity: 127  Timestamp: 3  [ 90 47 7F ]
NoteOff     Channel: 1  Note: 71  Velocity: 0    Timestamp: 3.5 [ 80 47 00 ]
NoteOn      Channel: 1  Note: 72  Velocity: 127  Timestamp: 3.5 [ 90 48 7F ]
NoteOff     Channel: 1  Note: 72  Velocity: 0    Timestamp: 3.75 [ 80 48 00 ]
NoteOn      Channel: 1  Note: 72  Velocity: 127  Timestamp: 4  [ 90 48 7F ]
NoteOff     Channel: 1  Note: 72  Velocity: 0    Timestamp: 4.5 [ 80 48 00 ]
NoteOn      Channel: 1  Note: 71  Velocity: 127  Timestamp: 4.5 [ 90 47 7F ]
NoteOff     Channel: 1  Note: 71  Velocity: 0    Timestamp: 5  [ 80 47 00 ]
NoteOn      Channel: 1  Note: 69  Velocity: 127  Timestamp: 5  [ 90 45 7F ]
NoteOff     Channel: 1  Note: 69  Velocity: 0    Timestamp: 5.5 [ 80 45 00 ]
NoteOn      Channel: 1  Note: 67  Velocity: 127  Timestamp: 5.5 [ 90 43 7F ]
NoteOff     Channel: 1  Note: 67  Velocity: 0    Timestamp: 5.75 [ 80 43 00 ]
NoteOn      Channel: 1  Note: 65  Velocity: 127  Timestamp: 6  [ 90 41 7F ]
NoteOff     Channel: 1  Note: 65  Velocity: 0    Timestamp: 6.5 [ 80 41 00 ]
NoteOn      Channel: 1  Note: 64  Velocity: 127  Timestamp: 6.5 [ 90 40 7F ]
NoteOff     Channel: 1  Note: 64  Velocity: 0    Timestamp: 7  [ 80 40 00 ]
NoteOn      Channel: 1  Note: 62  Velocity: 127  Timestamp: 7  [ 90 3E 7F ]
NoteOff     Channel: 1  Note: 62  Velocity: 0    Timestamp: 7.5 [ 80 3E 00 ]
NoteOn      Channel: 1  Note: 60  Velocity: 127  Timestamp: 7.5 [ 90 3C 7F ]
NoteOff     Channel: 1  Note: 60  Velocity: 0    Timestamp: 7.75 [ 80 3C 00 ]
AllNotesOff Channel: 1  Timestamp: 8  [ B0 7B 00 ]

```

Synthesize MIDI Messages

This example plays parsed MIDI messages using a simple monophonic synthesizer. To see a demonstration of this synthesizer, see “Design and Play a MIDI Synthesizer” on page 6-2.

```

% Initialize System objects for playing MIDI messages
osc = audioOscillator('square', 'Amplitude', 0, 'DutyCycle', 0.75);
deviceWriter = audioDeviceWriter;

simplesynth(msgArray, osc, deviceWriter);

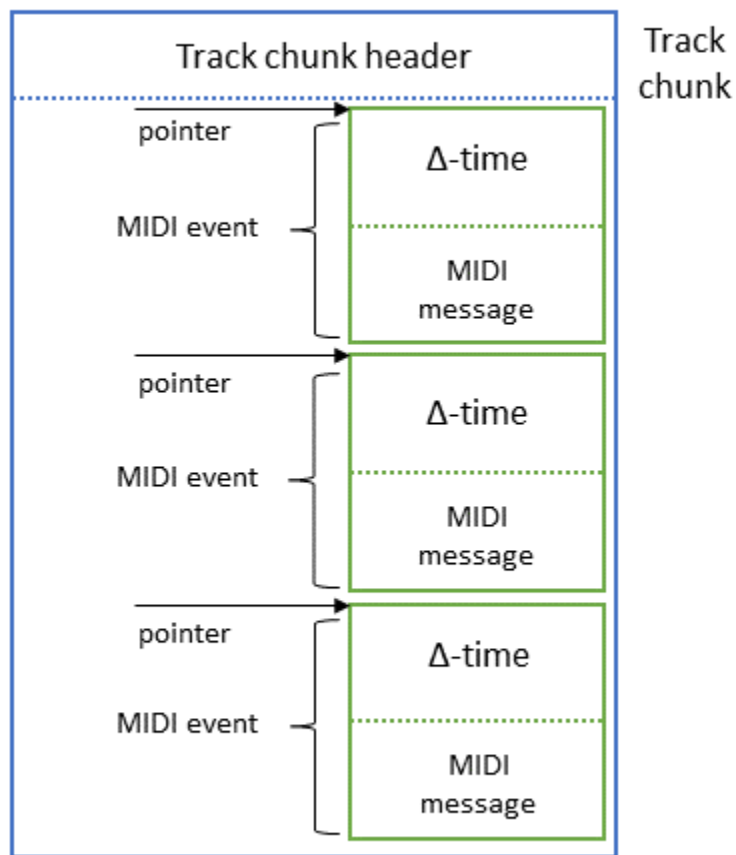
```

You can also send parsed MIDI messages to a MIDI device using `midisend`. For more information about interacting with MIDI devices using MATLAB, see “MIDI Device Interface” on page 7-2.

Helper Functions

Read Delta-Times

The delta-times of MIDI track events are stored as variable-length values. These values are 1 to 4 bytes long, with the most significant bit of each byte serving as a flag. The most significant bit of the final byte is set to 0, and the most significant bit of every other byte is set to 1.



In a MIDI track event, the delta-time is always placed before the MIDI message. There is no gap between a delta-time and the end of the previous MIDI event.

The `findVariableLength` function reads variable-length values like delta-times. It returns the length of the input value and the value itself. First, the function creates a 4-byte vector `byteStream`, which is set to all zeros. Then, it pushes a pointer to the beginning of the MIDI event. The function checks the four bytes after the pointer in a loop. For each byte, it checks the most significant bit (MSB). If the MSB is zero, `findVariableLength` adds the byte to `byteStream` and exits the loop. Otherwise, it adds the byte to `byteStream` and continues to the next byte.

Once the `findVariableLength` function reaches the final byte of the variable-length value, it evaluates the bytes collected in `byteStream` using the `polyval` function.

```
function [valueOut,byteLength] = findVariableLength(lengthIndex,readOut)

byteStream = zeros(4,1);

for i = 1:4
    valCheck = readOut(lengthIndex+i);
    byteStream(i) = bitand(valCheck,127);    % Mask MSB for value
    if ~bitand(valCheck,uint32(128))         % If MSB is 0, no need to append further
        break
    end
end

valueOut = polyval(byteStream(1:i),128);    % Base is 128 because 7 bits are used for value
byteLength = i;

end
```

Interpret MIDI Messages

There are three main types of messages in MIDI files:

- Sysex messages — System-exclusive messages ignored by this example.
- Meta-events — Can occur in place of MIDI messages to provide metadata for MIDI files, including song title and tempo. The `midimsg` object does not support meta-events. This example ignores meta-events.
- MIDI messages — Parsed by this example.

To interpret a MIDI message, read the status byte. The status byte is the first byte of a MIDI message.

Even though this example ignores Sysex messages and meta-events, it is important to identify these messages and determine their lengths. The lengths of Sysex messages and meta-events are key to determining where the next message starts. Sysex messages have 'F0' or 'F7' as the status byte, and meta-events have 'FF' as the status byte. Sysex messages and meta-events can be of varying lengths. After the status byte, Sysex messages and meta-events specify event lengths. The event length values are variable-length values like delta-time values. The length of the event can be determined using the `findVariableLength` function.

For MIDI messages, the message length can be determined by the value of the status byte. However, MIDI files support *running status*. If a MIDI message has the same status byte as the previous MIDI message, the status byte can be omitted. If the first byte of an incoming message is *not* a valid status byte, use the status byte of the previous MIDI message.

The `interpretMessage` function returns a status byte, a length, and a vector of bytes. The status byte is returned to the inner loop in case the next message is a running status message. The length is returned to the inner loop, where it specifies how far to push the inner loop pointer. Finally, the vector of bytes carries the raw binary data of a MIDI message. `interpretMessage` requires an output even if the function ignores a given message. For Sysex messages and meta-events, `interpretMessage` returns -1 instead of a vector of bytes.

```
function [statusOut,lenOut,message] = interpretMessage(statusIn,eventIn,readOut)

% Check if running status
```

```

introValue = readOut(eventIn+1);
if isStatusByte(introValue)
    statusOut = introValue;           % New status
    running = false;
else
    statusOut = statusIn;             % Running status—Keep old status
    running = true;
end

switch statusOut
case 255 % Meta-event (FF)—IGNORE
    [eventLength, lengthLen] = findVariableLength(eventIn+2, ...
        readOut); % Meta-events have an extra byte for type of meta-event
    lenOut = 2+lengthLen+eventLength;
    message = -1;
case 240 % Sysex message (F0)—IGNORE
    [eventLength, lengthLen] = findVariableLength(eventIn+1, ...
        readOut);
    lenOut = 1+lengthLen+eventLength;
    message = -1;

case 247 % Sysex message (F7)—IGNORE
    [eventLength, lengthLen] = findVariableLength(eventIn+1, ...
        readOut);
    lenOut = 1+lengthLen+eventLength;
    message = -1;
otherwise % MIDI message—READ
    eventLength = msgnbytes(statusOut);
    if running
        % Running msgs don't retransmit status—Drop a bit
        lenOut = eventLength-1;
        message = uint8([statusOut; readOut(eventIn+(1:lenOut))]);
    else
        lenOut = eventLength;
        message = uint8(readOut(eventIn+(1:lenOut)));
    end
end

end

% ----

function n = msgnbytes(statusByte)

if statusByte <= 191 % hex2dec('BF')
    n = 3;
elseif statusByte <= 223 % hex2dec('DF')
    n = 2;
elseif statusByte <= 239 % hex2dec('EF')
    n = 3;
elseif statusByte == 240 % hex2dec('F0')
    n = 1;
elseif statusByte == 241 % hex2dec('F1')
    n = 2;
elseif statusByte == 242 % hex2dec('F2')
    n = 3;
elseif statusByte <= 243 % hex2dec('F3')

```

```

        n = 2;
    else
        n = 1;
    end

end

% ----

function yes = isStatusByte(b)
yes = b > 127;
end

```

Create MIDI Messages

The `midimsg` object can generate a MIDI message from a struct using the format:

```

midistruct = struct('RawBytes', [144 65 127 0 0 0 0 0], 'Timestamp',1);
msg = midimsg.fromStruct(midistruct)

```

This returns:

```

msg =
    MIDI message:
        NoteOn          Channel: 1  Note: 65  Velocity: 127  Timestamp: 1  [ 90 41 7F ]

```

The `createMessage` function returns a `midimsg` object and a timestamp. The `midimsg` object requires its input struct to have two fields:

- **RawBytes**—A 1-by-8 vector of bytes
- **Timestamp**—A time in seconds

To set the **RawBytes** field, take the vector of bytes created by `interpretMessage` and append enough zeros to create a 1-by-8 vector of bytes.

To set the **Timestamp** field, create a timestamp variable `ts`. Set `ts` to 0 before parsing any track chunks. For every MIDI message sent, convert the delta-time value from ticks to seconds. Then, add that value to `ts`. To convert MIDI ticks to seconds, use:

$$\text{timeAdd} = \frac{\text{numTicks} \cdot \text{tempo}}{\text{ticksPerQuarterNote} \cdot 1\text{e}6}$$

Where `tempo` is in microseconds (μs) per quarter note. To convert beats per minute (BPM) to μs per quarter note, use:

$$\text{tempo} = \frac{6\text{e}7}{\text{BPM}}$$

Once you fill both fields of the struct, create a `midimsg` object. Return the `midimsg` object and the modified value of `ts`.

The `createMessage` function ignores Sysex messages and meta-events. When `interpretMessage` handles Sysex messages and meta-events, it returns -1 instead of a vector of bytes. The `createMessage` function then checks for that value. If `createMessage` identifies a Sysex message or meta-event, it returns the `ts` value it was given and an empty `midimsg` object.

```

function [tsOut,msgOut] = createMessage(messageIn,tsIn,deltaTimeIn,ticksPerQNoteIn,bpmIn)

```

```
if messageIn < 0      % Ignore Sysex message/meta-event data
    tsOut = tsIn;
    msgOut = midimsg(0);
    return
end

% Create RawBytes field
messageLength = length(messageIn);
zeroAppend = zeros(8-messageLength,1);
bytesIn = transpose([messageIn;zeroAppend]);

% deltaTimeIn and ticksPerQNoteIn are both uints
% Recast both values as doubles
d = double(deltaTimeIn);
t = double(ticksPerQNoteIn);

% Create Timestamp field and tsOut
msPerQNote = 6e7/bpmIn;
timeAdd = d*(msPerQNote/t)/1e6;
tsOut = tsIn+timeAdd;

% Create midimsg object
midiStruct = struct('RawBytes',bytesIn,'Timestamp',tsOut);
msgOut = midimsg.fromStruct(midiStruct);

end
```

Play MIDI Messages Using a Synthesizer

This example plays parsed MIDI messages using a simple monophonic synthesizer. To see a demonstration of this synthesizer, see “Design and Play a MIDI Synthesizer” on page 6-2.

You can also send parsed MIDI messages to a MIDI device using `midisend`. For more information about interacting with MIDI devices using MATLAB, see “MIDI Device Interface” on page 7-2.

```
function simplesynth(msgArray,osc,deviceWriter)

i = 1;
tic
endTime = msgArray(length(msgArray)).Timestamp;

while toc < endTime
    if toc >= msgArray(i).Timestamp      % At new note, update deviceWriter
        msg = msgArray(i);
        i = i+1;
        if isNoteOn(msg)
            osc.Frequency = note2freq(msg.Note);
            osc.Amplitude = msg.Velocity/127;
        elseif isNoteOff(msg)
            if msg.Note == msg.Note
                osc.Amplitude = 0;
            end
        end
        end
        deviceWriter(osc());      % Keep calling deviceWriter as it is updated
    end
end
```

```
% ----

function yes = isNoteOn(msg)
yes = strcmp(msg.Type, 'NoteOn') ...
    && msg.Velocity > 0;
end

% ----

function yes = isNoteOff(msg)
yes = strcmp(msg.Type, 'NoteOff') ...
    || (strcmp(msg.Type, 'NoteOn') && msg.Velocity == 0);
end

% ----

function freq = note2freq(note)
freqA = 440;
noteA = 69;
freq = freqA * 2.^((note-noteA)/12);
end
```

Cocktail Party Source Separation Using Deep Learning Networks

This example shows how to isolate a speech signal using a deep learning network.

Introduction

The cocktail party effect refers to the ability of the brain to focus on a single speaker while filtering out other voices and background noise. Humans perform very well at the cocktail party problem. This example shows how to use a deep learning network to separate individual speakers from a speech mix where one male and one female are speaking simultaneously.

Problem Summary

Load audio files containing male and female speech sampled at 4 kHz. Listen to the audio files individually for reference.

```
[mSpeech,Fs] = audioread("MaleSpeech-16-4-mono-20secs.wav");  
sound(mSpeech,Fs)
```

```
[fSpeech] = audioread("FemaleSpeech-16-4-mono-20secs.wav");  
sound(fSpeech,Fs)
```

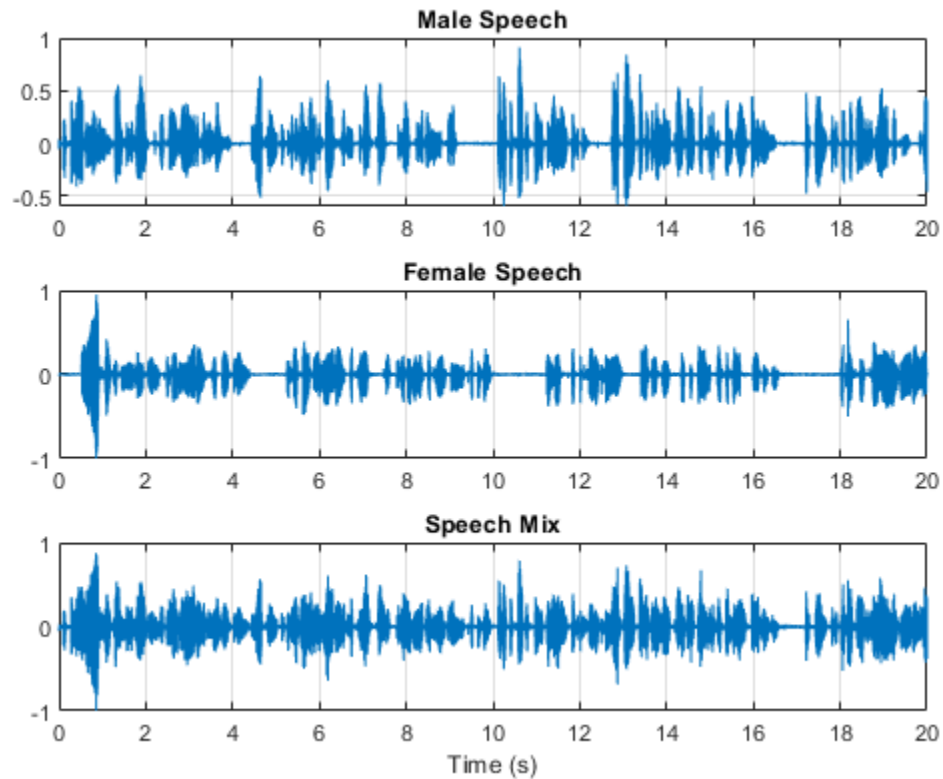
Combine the two speech sources. Ensure the sources have equal power in the mix. Normalize the mix so that its max amplitude is one.

```
mSpeech = mSpeech/norm(mSpeech);  
fSpeech = fSpeech/norm(fSpeech);  
ampAdj = max(abs([mSpeech;fSpeech]));  
mSpeech = mSpeech/ampAdj;  
fSpeech = fSpeech/ampAdj;  
mix = mSpeech + fSpeech;  
mix = mix ./ max(abs(mix));
```

Visualize the original and mix signals. Listen to the mixed speech signal. This example shows a source separation scheme that extracts the male and female sources from the speech mix.

```
t = (0: numel(mix)-1)*(1/Fs);
```

```
figure(1)  
subplot(3,1,1)  
plot(t,mSpeech)  
title("Male Speech")  
grid on  
subplot(3,1,2)  
plot(t,fSpeech)  
title("Female Speech")  
grid on  
subplot(3,1,3)  
plot(t,mix)  
title("Speech Mix")  
xlabel("Time (s)")  
grid on
```



Listen to the mix audio.

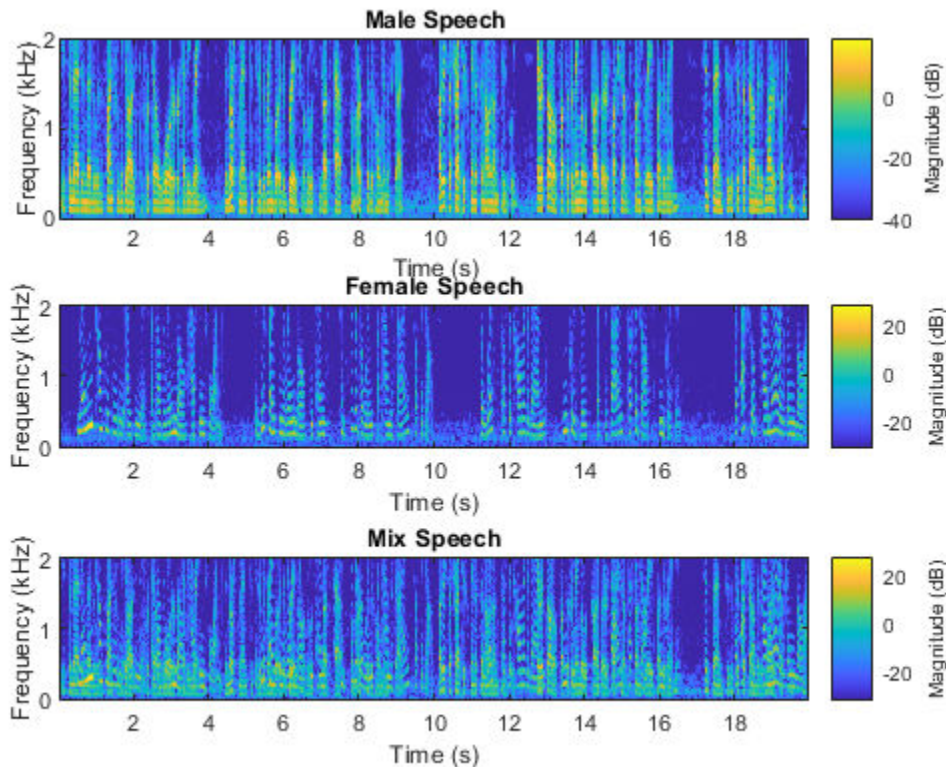
```
sound(mix,Fs)
```

Time-Frequency Representation

Use `stft` to visualize the time-frequency (TF) representation of the male, female, and mix speech signals. Use a Hann window of length 128, an FFT length of 128, and an overlap length of 96.

```
WindowLength = 128;
FFTLenght    = 128;
OverlapLength = 96;
win          = hann(WindowLength,"periodic");

figure(2)
subplot(3,1,1)
stft(mSpeech, Fs, 'Window', win, 'OverlapLength', OverlapLength, ...
    'FFTLenght', FFTLength, 'FrequencyRange', 'onesided');
title("Male Speech")
subplot(3,1,2)
stft(fSpeech, Fs, 'Window', win, 'OverlapLength', OverlapLength, ...
    'FFTLenght', FFTLength, 'FrequencyRange', 'onesided');
title("Female Speech")
subplot(3,1,3)
stft(mix, Fs, 'Window', win, 'OverlapLength', OverlapLength, ...
    'FFTLenght', FFTLength, 'FrequencyRange', 'onesided');
title("Mix Speech")
```



Source Separation Using Ideal Time-Frequency Masks

The application of a TF mask has been shown to be an effective method for separating desired audio signals from competing sounds. A TF mask is a matrix of the same size as the underlying STFT. The mask is multiplied element-by-element with the underlying STFT to isolate the desired source. The TF mask can be binary or soft.

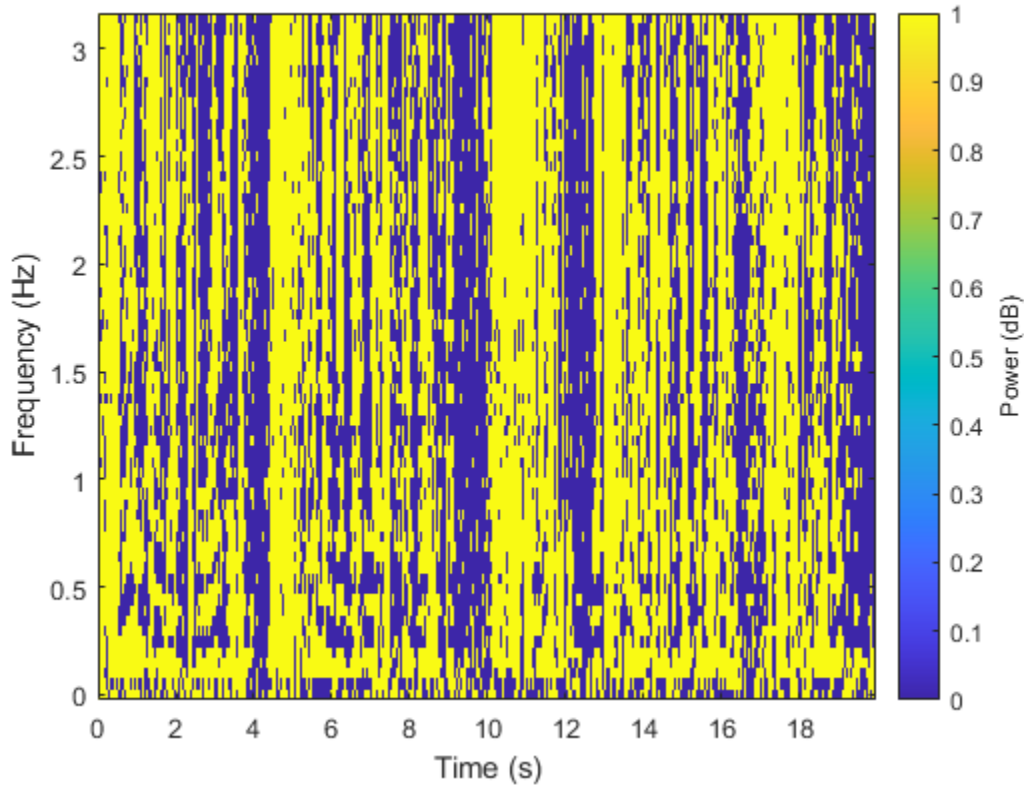
Source Separation Using Ideal Binary Masks

In an ideal binary mask, the mask cell values are either 0 or 1. If the power of the desired source is greater than the combined power of other sources at a particular TF cell, then that cell is set to 1. Otherwise, the cell is set to 0.

Compute the ideal binary mask for the male speaker and then visualize it.

```
P_M      = stft(mSpeech, 'Window', win, 'OverlapLength', OverlapLength,...
               'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
P_F      = stft(fSpeech, 'Window', win, 'OverlapLength', OverlapLength,...
               'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
[P_mix,F] = stft(mix, 'Window', win, 'OverlapLength', OverlapLength,...
               'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
binaryMask = abs(P_M) >= abs(P_F);

figure(3)
plotMask(binaryMask, WindowLength - OverlapLength, F, Fs)
```

Estimate the male speech STFT by multiplying the mix STFT by the male speaker's binary mask. Estimate the female speech STFT by multiplying the mix STFT by the inverse of the male speaker's binary mask.

```
P_M_Hard = P_mix .* binaryMask;
P_F_Hard = P_mix .* (1-binaryMask);
```

Estimate the male and female audio signals using the inverse short-time FFT (ISTFT). Visualize the estimated and original signals. Listen to the estimated male and female speech signals.

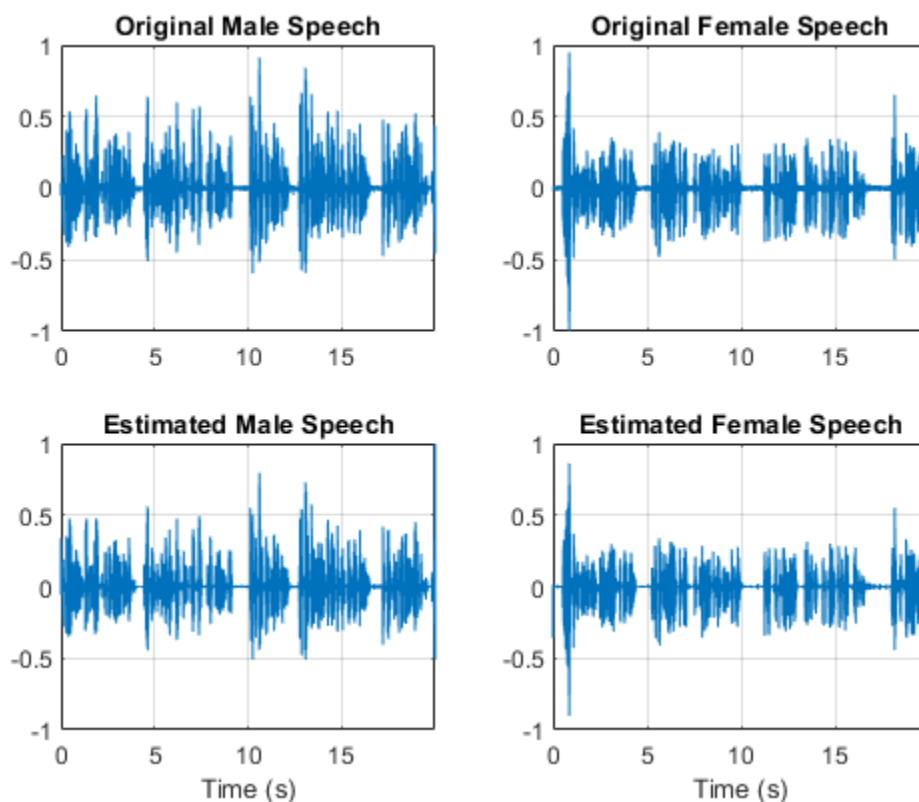
```
mSpeech_Hard = istft(P_M_Hard , 'Window', win, 'OverlapLength', OverlapLength,...
    'FFTLenght', FFTLength, 'FrequencyRange', 'onesided');
fSpeech_Hard = istft(P_F_Hard , 'Window', win, 'OverlapLength', OverlapLength,...
    'FFTLenght', FFTLength, 'FrequencyRange', 'onesided');

figure(4)
subplot(2,2,1)
plot(t,mSpeech)
axis([t(1) t(end) -1 1])
title("Original Male Speech")
grid on

subplot(2,2,3)
plot(t,mSpeech_Hard)
axis([t(1) t(end) -1 1])
xlabel("Time (s)")
title("Estimated Male Speech")
grid on
```

```
subplot(2,2,2)
plot(t,fSpeech)
axis([t(1) t(end) -1 1])
title("Original Female Speech")
grid on

subplot(2,2,4)
plot(t,fSpeech_Hard)
axis([t(1) t(end) -1 1])
title("Estimated Female Speech")
xlabel("Time (s)")
grid on
```



```
sound(mSpeech_Hard,Fs)
```

```
sound(fSpeech_Hard,Fs)
```

Source Separation Using Ideal Soft Masks

In a soft mask, the TF mask cell value is equal to the ratio of the desired source power to the total mix power. TF cells have values in the range [0,1].

Compute the soft mask for the male speaker. Estimate the STFT of the male speaker by multiplying the mix STFT by the male speaker's soft mask. Estimate the STFT of the female speaker by multiplying the mix STFT by the female speaker's soft mask.

Estimate the male and female audio signals using the ISTFT.

```

softMask = abs(P_M) ./ (abs(P_F) + abs(P_M) + eps);

P_M_Soft = P_mix .* softMask;
P_F_Soft = P_mix .* (1-softMask);

mSpeech_Soft = istft(P_M_Soft, 'Window', win, 'OverlapLength', OverlapLength, ...
                    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
fSpeech_Soft = istft(P_F_Soft, 'Window', win, 'OverlapLength', OverlapLength, ...
                    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');

```

Visualize the estimated and original signals. Listen to the estimated male and female speech signals. Note that the results are very good because the mask is created with full knowledge of the separated male and female signals.

```

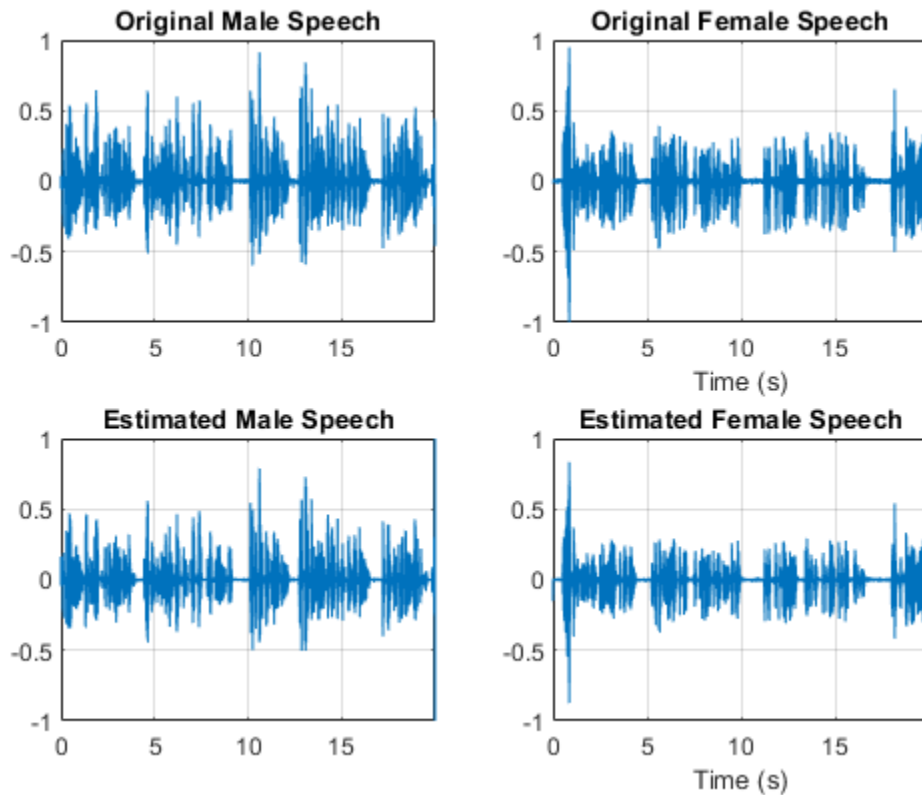
figure(5)
subplot(2,2,1)
plot(t,mSpeech)
axis([t(1) t(end) -1 1])
title("Original Male Speech")
grid on

subplot(2,2,3)
plot(t,mSpeech_Soft)
axis([t(1) t(end) -1 1])
title("Estimated Male Speech")
grid on

subplot(2,2,2)
plot(t,fSpeech)
axis([t(1) t(end) -1 1])
xlabel("Time (s)")
title("Original Female Speech")
grid on

subplot(2,2,4)
plot(t,fSpeech_Soft)
axis([t(1) t(end) -1 1])
xlabel("Time (s)")
title("Estimated Female Speech")
grid on

```



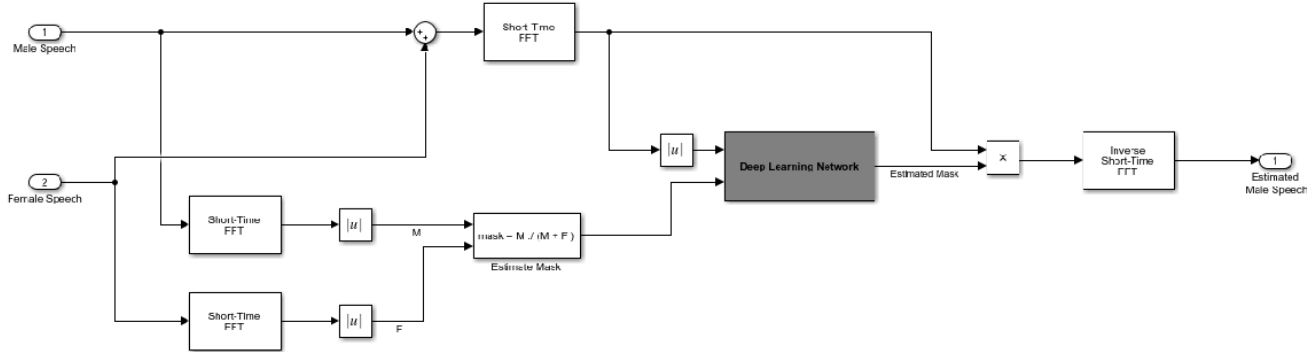
```
sound(mSpeech_Soft,Fs)
```

```
sound(fSpeech_Soft,Fs)
```

Mask Estimation Using Deep Learning

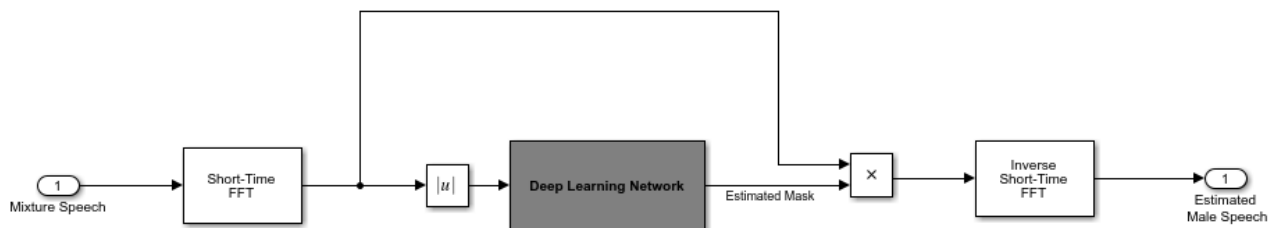
The goal of the deep learning network in this example is to estimate the ideal soft mask described above. The network estimates the mask corresponding to the male speaker. The female speaker mask is derived directly from the male mask.

The basic deep learning training scheme is shown below. The predictor is the magnitude spectra of the mixed (male + female) audio. The target is the ideal soft masks corresponding to the male speaker. The regression network uses the predictor input to minimize the mean square error between its output and the input target. At the output, the audio STFT is converted back to the time domain using the output magnitude spectrum and the phase of the mix signal.



You transform the audio to the frequency domain using the Short-Time Fourier transform (STFT), with a window length of 128 samples, an overlap of 127, and a Hann window. You reduce the size of the spectral vector to 65 by dropping the frequency samples corresponding to negative frequencies (because the time-domain speech signal is real, this does not lead to any information loss). The predictor input consists of 20 consecutive STFT vectors. The output is a 65-by-20 soft mask.

You use the trained network to estimate the male speech. The input to the trained network is the mixture (male + female) speech audio.



STFT Targets and Predictors

This section illustrates how to generate the target and predictor signals from the training dataset.

Read in training signals consisting of around 400 seconds of speech from male and female speakers, respectively, sampled at 4 kHz. The low sample rate is used to speed up training. Trim the training signals so that they are the same length.

```
maleTrainingAudioFile = "MaleSpeech-16-4-mono-405secs.wav";
femaleTrainingAudioFile = "FemaleSpeech-16-4-mono-405secs.wav";

maleSpeechTrain = audioread(maleTrainingAudioFile);
femaleSpeechTrain = audioread(femaleTrainingAudioFile);

L = min(length(maleSpeechTrain), length(femaleSpeechTrain));
maleSpeechTrain = maleSpeechTrain(1:L);
femaleSpeechTrain = femaleSpeechTrain(1:L);
```

Read in validation signals consisting of around 20 seconds of speech from male and female speakers, respectively, sampled at 4 kHz. Trim the validation signals so that they are the same length

```
maleValidationAudioFile = "MaleSpeech-16-4-mono-20secs.wav";  
femaleValidationAudioFile = "FemaleSpeech-16-4-mono-20secs.wav";
```

```
maleSpeechValidate = audioread(maleValidationAudioFile);  
femaleSpeechValidate = audioread(femaleValidationAudioFile);
```

```
L = min(length(maleSpeechValidate),length(femaleSpeechValidate));  
maleSpeechValidate = maleSpeechValidate(1:L);  
femaleSpeechValidate = femaleSpeechValidate(1:L);
```

Scale the training signals to the same power. Scale the validation signals to the same power.

```
maleSpeechTrain = maleSpeechTrain/norm(maleSpeechTrain);  
femaleSpeechTrain = femaleSpeechTrain/norm(femaleSpeechTrain);  
ampAdj = max(abs([maleSpeechTrain;femaleSpeechTrain]));  
maleSpeechTrain = maleSpeechTrain/ampAdj;  
femaleSpeechTrain = femaleSpeechTrain/ampAdj;
```

```
maleSpeechValidate = maleSpeechValidate/norm(maleSpeechValidate);  
femaleSpeechValidate = femaleSpeechValidate/norm(femaleSpeechValidate);  
ampAdj = max(abs([maleSpeechValidate;femaleSpeechValidate]));  
maleSpeechValidate = maleSpeechValidate/ampAdj;  
femaleSpeechValidate = femaleSpeechValidate/ampAdj;
```

Create the training and validation "cocktail party" mixes.

```
mixTrain = maleSpeechTrain + femaleSpeechTrain;  
mixTrain = mixTrain / max(mixTrain);
```

```
mixValidate = maleSpeechValidate + femaleSpeechValidate;  
mixValidate = mixValidate / max(mixValidate);
```

Generate training STFTs.

```
WindowLength = 128;  
FFTLenght = 128;  
OverlapLength = 128-1;  
Fs = 4000;  
win = hann(WindowLength,"periodic");  
  
P_mix0 = stft(mixTrain, 'Window', win, 'OverlapLength', OverlapLength,...  
             'FFTLenght', FFTLength, 'FrequencyRange', 'onesided');  
P_M = abs(stft(maleSpeechTrain, 'Window', win, 'OverlapLength', OverlapLength,...  
             'FFTLenght', FFTLength, 'FrequencyRange', 'onesided'));  
P_F = abs(stft(femaleSpeechTrain, 'Window', win, 'OverlapLength', OverlapLength,...  
             'FFTLenght', FFTLength, 'FrequencyRange', 'onesided'));
```

Take the log of the mix STFT. Normalize the values by their mean and standard deviation.

```
P_mix = log(abs(P_mix0) + eps);  
MP = mean(P_mix(:));  
SP = std(P_mix(:));  
P_mix = (P_mix - MP) / SP;
```

Generate validation STFTs. Take the log of the mix STFT. Normalize the values by their mean and standard deviation.

```
P_Val_mix0 = stft(mixValidate, 'Window', win, 'OverlapLength', OverlapLength,...  
                 'FFTLenght', FFTLength, 'FrequencyRange', 'onesided');
```

```

P_Val_M    = abs(stft(maleSpeechValidate, 'Window', win, 'OverlapLength', OverlapLength,...
                    'FFTLenght', FFTLength, 'FrequencyRange', 'onesided'));
P_Val_F    = abs(stft(femaleSpeechValidate, 'Window', win, 'OverlapLength', OverlapLength,...
                    'FFTLenght', FFTLength, 'FrequencyRange', 'onesided'));

P_Val_mix = log(abs(P_Val_mix0) + eps);
MP        = mean(P_Val_mix(:));
SP        = std(P_Val_mix(:));
P_Val_mix = (P_Val_mix - MP) / SP;

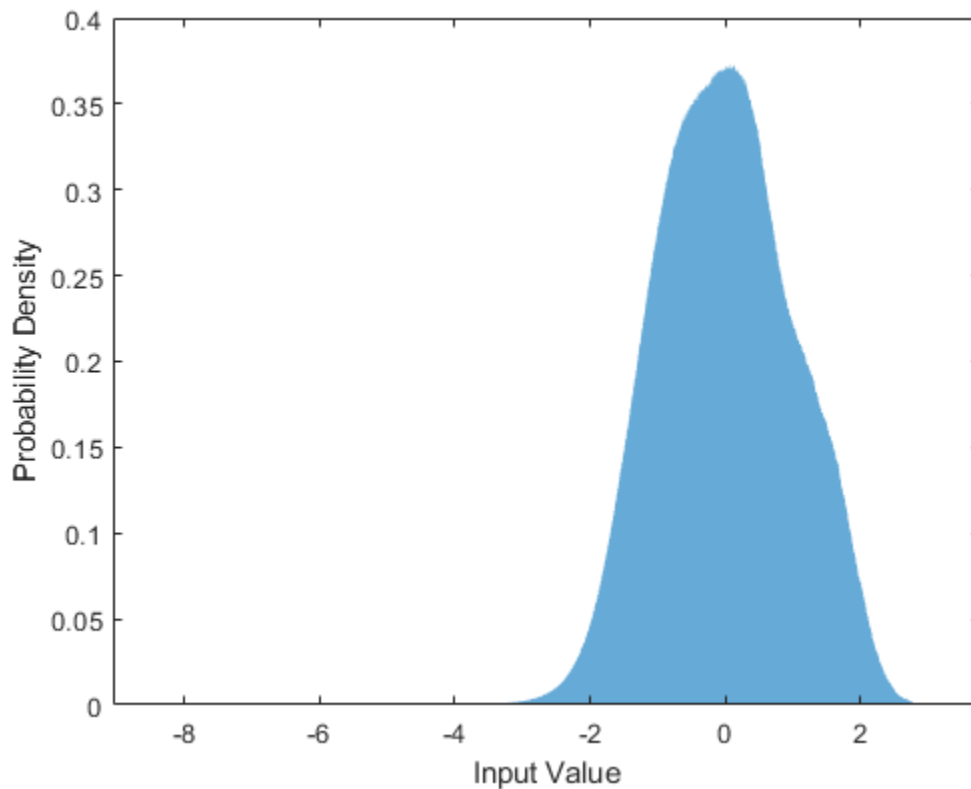
```

Training neural networks is easiest when the inputs to the network have a reasonably smooth distribution and are normalized. To check that the data distribution is smooth, plot a histogram of the STFT values of the training data.

```

figure(6)
histogram(P_mix,"EdgeColor","none","Normalization","pdf")
xlabel("Input Value")
ylabel("Probability Density")

```



Compute the training soft mask. Use this mask as the target signal while training the network.

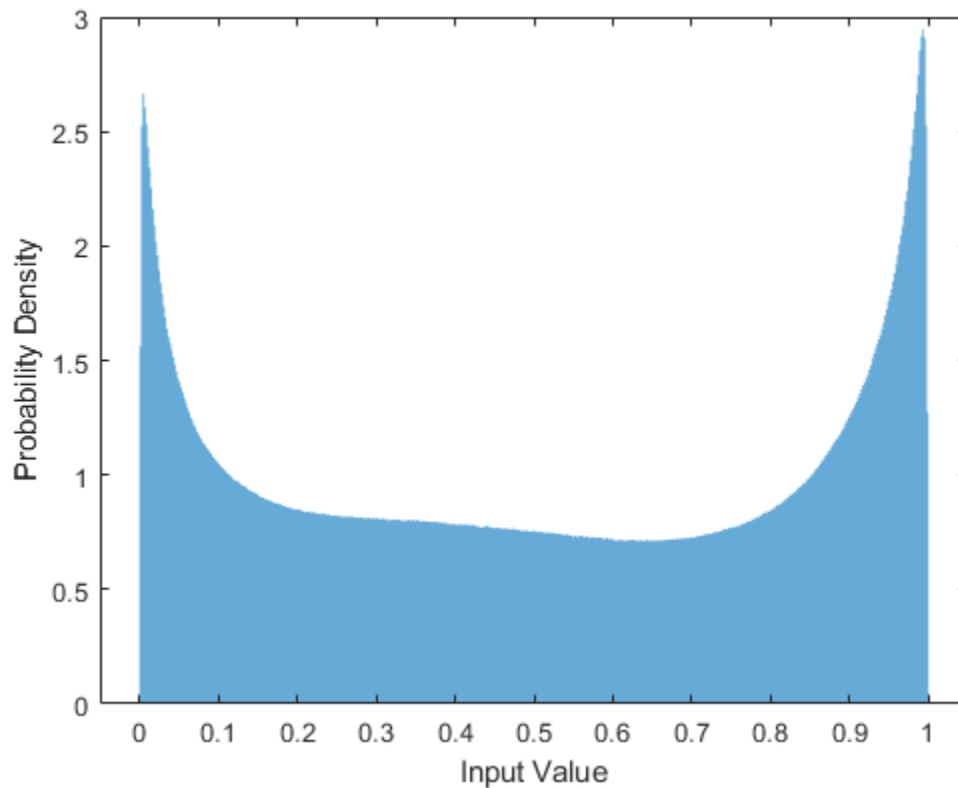
```
maskTrain = P_M ./ (P_M + P_F + eps);
```

Compute the validation soft mask. Use this mask to evaluate the mask emitted by the trained network.

```
maskValidate = P_Val_M ./ (P_Val_M + P_Val_F + eps);
```

To check that the target data distribution is smooth, plot a histogram of the mask values of the training data.

```
figure(7)
histogram(maskTrain,"EdgeColor","none","Normalization","pdf")
xlabel("Input Value")
ylabel("Probability Density")
```



Create chunks of size (65, 20) from the predictor and target signals. In order to get more training samples, use an overlap of 10 segments between consecutive chunks.

```
seqLen      = 20;
seqOverlap  = 10;
mixSequences = zeros(1 + FFTLength/2, seqLen, 1, 0);
maskSequences = zeros(1 + FFTLength/2, seqLen, 1, 0);

loc = 1;
while loc < size(P_mix, 2) - seqLen
    mixSequences(:, :, :, end+1) = P_mix(:, loc:loc+seqLen-1); %#ok
    maskSequences(:, :, :, end+1) = maskTrain(:, loc:loc+seqLen-1); %#ok
    loc = loc + seqOverlap;
end
```

Create chunks of size (65,20) from the validation predictor and target signals.

```
mixValSequences = zeros(1 + FFTLength/2, seqLen, 1, 0);
maskValSequences = zeros(1 + FFTLength/2, seqLen, 1, 0);
seqOverlap      = seqLen;
```



```

loc = 1;
while loc < size(P_Val_mix,2) - seqLen
    mixValSequences(:,loc:loc+seqLen-1) = P_Val_mix(:,loc:loc+seqLen-1); %#ok
    maskValSequences(:,loc:loc+seqLen-1) = maskValidate(:,loc:loc+seqLen-1); %#ok
    loc = loc + seqOverlap;
end

```

Reshape the training and validation signals.

```

mixSequencesT = reshape(mixSequences, [1 1 (1 + FFTLength/2) * seqLen size(mixSequences,4)]);
mixSequencesV = reshape(mixValSequences, [1 1 (1 + FFTLength/2) * seqLen size(mixValSequences,4)]);
maskSequencesT = reshape(maskSequences, [1 1 (1 + FFTLength/2) * seqLen size(maskSequences,4)]);
maskSequencesV = reshape(maskValSequences, [1 1 (1 + FFTLength/2) * seqLen size(maskValSequences,4)]);

```

Define Deep Learning Network

Define the layers of the network. Specify the input size to be images of size 1-by-1-by-1300. Define two hidden fully connected layers, each with 1300 neurons. Follow each hidden fully connected layer with a sigmoid layer. The batch normalization layers normalize the means and standard deviations of the outputs. Add a fully connected layer with 1300 neurons, followed by a regression layer.

```
numNodes = (1 + FFTLength/2) * seqLen;
```

```
layers = [ ...
```

```
    imageInputLayer([1 1 (1 + FFTLength/2)*seqLen], "Normalization", "None")
```

```
    fullyConnectedLayer(numNodes)
    BiasedSigmoidLayer(6)
    batchNormalizationLayer
    dropoutLayer(0.1)
```

```
    fullyConnectedLayer(numNodes)
    BiasedSigmoidLayer(6)
    batchNormalizationLayer
    dropoutLayer(0.1)
```

```
    fullyConnectedLayer(numNodes)
    BiasedSigmoidLayer(0)
```

```
    regressionLayer
```

```
];
```

Specify the training options for the network. Set `MaxEpochs` to 3 so that the network makes three passes through the training data. Set `MiniBatchSize` to 64 so that the network looks at 64 training signals at a time. Set `Plots` to `training-progress` to generate plots that show the training progress as the number of iterations increases. Set `Verbose` to `false` to disable printing the table output that corresponds to the data shown in the plot into the command line window. Set `Shuffle` to `every-epoch` to shuffle the training sequences at the beginning of each epoch. Set `LearnRateSchedule` to `piecewise` to decrease the learning rate by a specified factor (0.1) every time a certain number of epochs (1) has passed. Set `ValidationData` to the validation predictors and targets. Set `ValidationFrequency` such that the validation mean square error is computed once per epoch. This example uses the adaptive moment estimation (ADAM) solver.

```

maxEpochs      = 3;
miniBatchSize   = 64;

```

```

options = trainingOptions("adam", ...
    "MaxEpochs",maxEpochs, ...
    "MiniBatchSize",miniBatchSize, ...
    "SequenceLength","longest", ...
    "Shuffle","every-epoch",...
    "Verbose",0, ...
    "Plots","training-progress",...
    "ValidationFrequency",floor(size(mixSequencesT,4)/miniBatchSize),...
    "ValidationData",{mixSequencesV,maskSequencesV},...
    "LearnRateSchedule","piecewise",...
    "LearnRateDropFactor",0.9, ...
    "LearnRateDropPeriod",1);

```

Train Deep Learning Network

Train the network with the specified training options and layer architecture using `trainNetwork`. Because the training set is large, the training process can take several minutes. To load a pre-trained network, set `doTraining` to false.

```

doTraining = true;
if doTraining
    CocktailPartyNet = trainNetwork(mixSequencesT,maskSequencesT,layers,options);
else
    s = load("CocktailPartyNet.mat");
    CocktailPartyNet = s.CocktailPartyNet;
end

```



Pass the validation predictors to the network. The output is the estimated mask. Reshape the estimated mask.

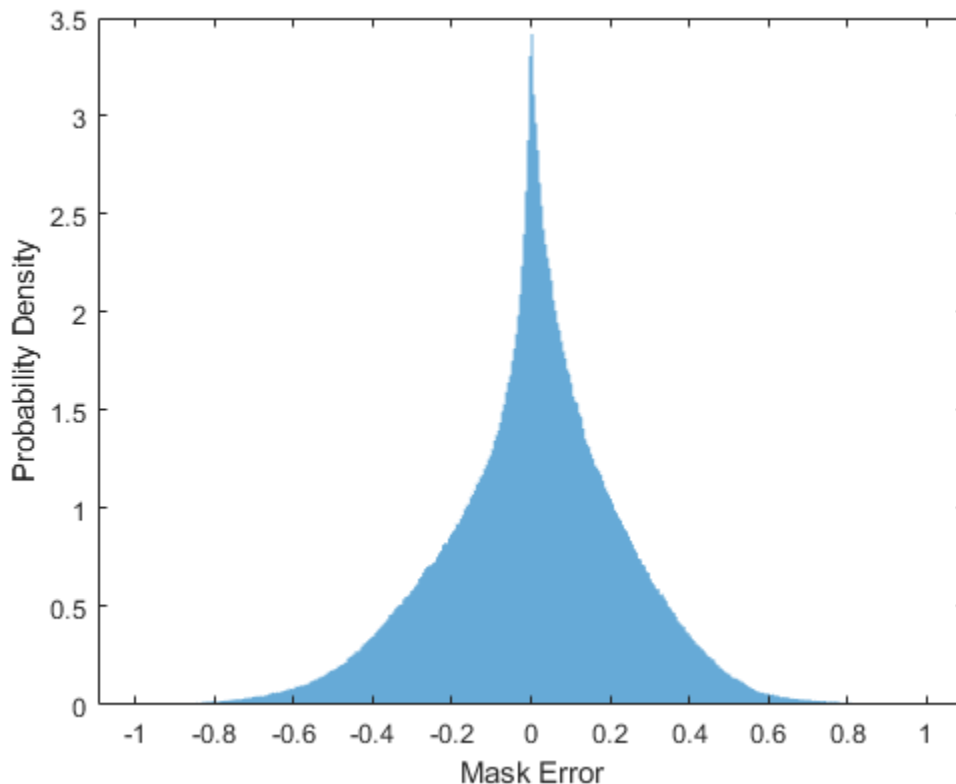
```
estimatedMasks0 = predict(CocktailPartyNet,mixSequencesV);
```

```
estimatedMasks0 = estimatedMasks0.';
estimatedMasks0 = reshape(estimatedMasks0,1 + FFTLength/2,numel(estimatedMasks0)/(1 + FFTLength/2))
```

Evaluate Deep Learning Network

Plot a histogram of the error between the actual and expected mask.

```
figure(8)
histogram(maskValSequences(:) - estimatedMasks0(:),"EdgeColor","none","Normalization","pdf")
xlabel("Mask Error")
ylabel("Probability Density")
```



Evaluate Soft Mask Estimation

Estimate male and female soft masks. Estimate male and female binary masks by thresholding the soft masks.

```
SoftMaleMask = estimatedMasks0;
SoftFemaleMask = 1 - SoftMaleMask;
```

Shorten the mix STFT to match the size of the estimated mask.

```
P_Val_mix0 = P_Val_mix0(:,1:size(SoftMaleMask,2));
```

Multiply the mix STFT by the male soft mask to get the estimated male speech STFT.

```
P_Male = P_Val_mix0 .* SoftMaleMask;
```

Use the ISTFT to get the estimated male audio signal. Scale the audio.

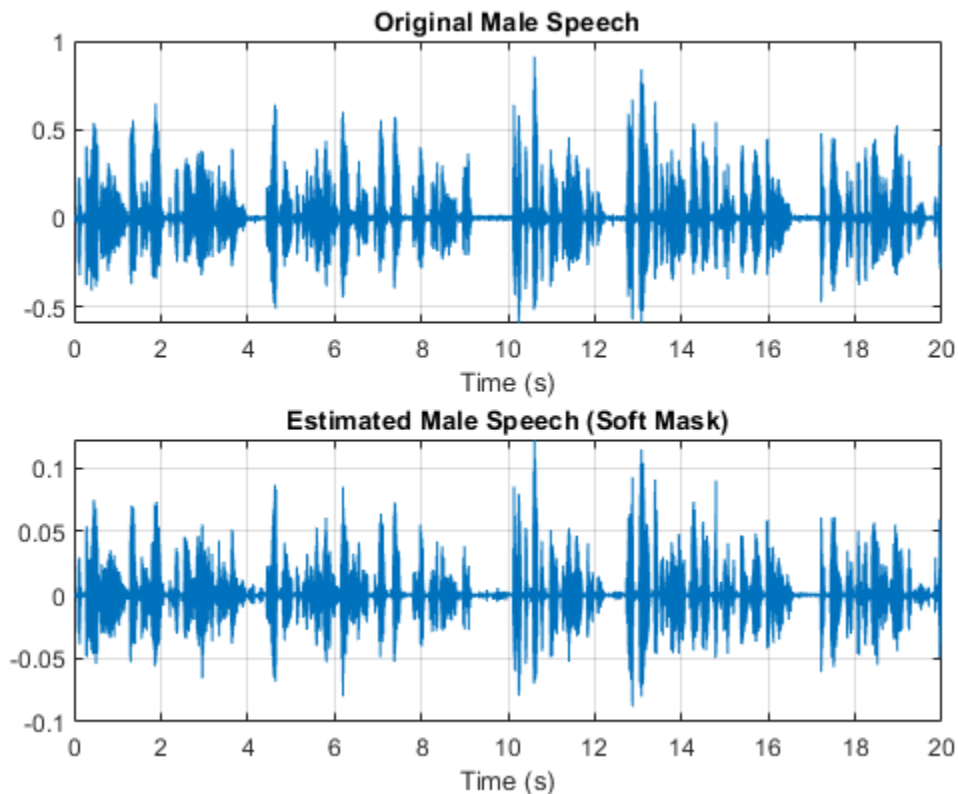
```
maleSpeech_est_soft = istft(P_Male, 'Window', win, 'OverlapLength', OverlapLength,...
    'FFTLength', FFTLength, 'ConjugateSymmetric', true,...
    'FrequencyRange', 'onesided');
maleSpeech_est_soft = maleSpeech_est_soft / max(abs(maleSpeech_est_soft));
```

Visualize the estimated and original male speech signals. Listen to the estimated soft mask male speech.

```
range = (numel(win):numel(maleSpeech_est_soft)-numel(win));
t      = range * (1/Fs);
```

```
figure(9)
subplot(2,1,1)
plot(t,maleSpeechValidate(range))
title("Original Male Speech")
xlabel("Time (s)")
grid on

subplot(2,1,2)
plot(t,maleSpeech_est_soft(range))
xlabel("Time (s)")
title("Estimated Male Speech (Soft Mask)")
grid on
```



```
sound(maleSpeech_est_soft(range),Fs)
```

Multiply the mix STFT by the female soft mask to get the estimated female speech STFT. Use the ISTFT to get the estimated male audio signal. Scale the audio.

```

P_Female = P_Val_mix0 .* SoftFemaleMask;

femaleSpeech_est_soft = istft(P_Female, 'Window', win, 'OverlapLength', OverlapLength,...
                              'FFTLength', FFTLength, 'ConjugateSymmetric', true,...
                              'FrequencyRange', 'onesided');
femaleSpeech_est_soft = femaleSpeech_est_soft / max(femaleSpeech_est_soft);

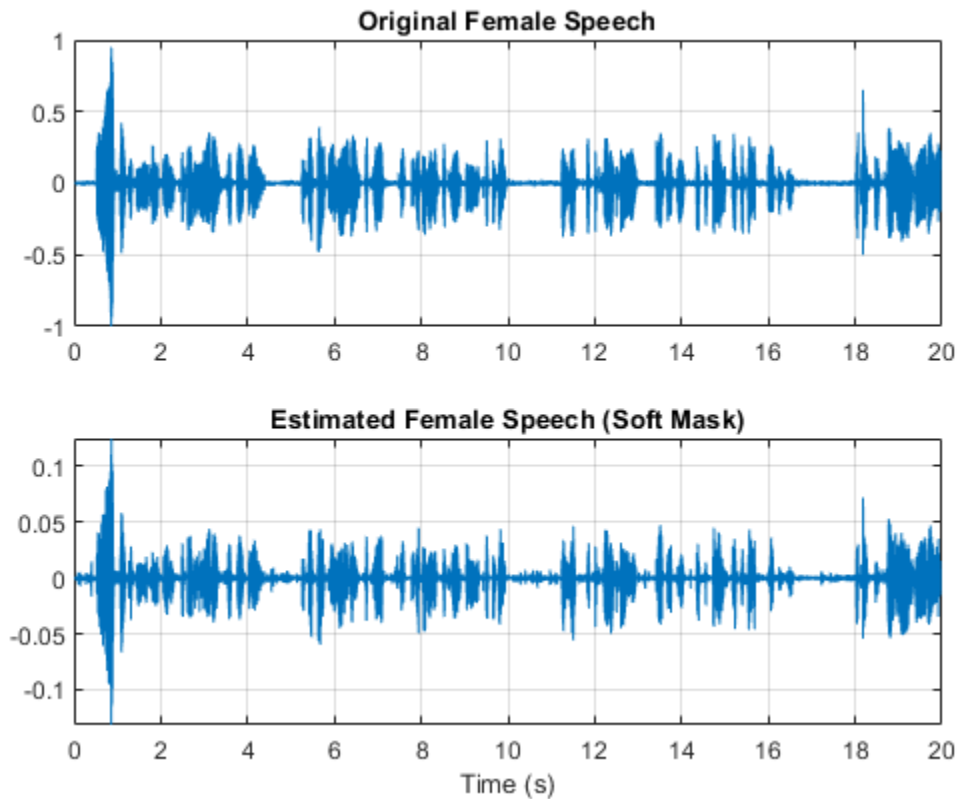
Visualize the estimated and original female signals. Listen to the estimated female speech.

range = (numel(win):numel(maleSpeech_est_soft) - numel(win));
t      = range * (1/Fs);

figure(10)
subplot(2,1,1)
plot(t, femaleSpeechValidate(range))
title("Original Female Speech")
grid on

subplot(2,1,2)
plot(t, femaleSpeech_est_soft(range))
xlabel("Time (s)")
title("Estimated Female Speech (Soft Mask)")
grid on

```



```

sound(femaleSpeech_est_soft(range), Fs)

```

Evaluate Binary Mask Estimation

Estimate male and female binary masks by thresholding the soft masks.

```
HardMaleMask = SoftMaleMask >= 0.5;  
HardFemaleMask = SoftMaleMask < 0.5;
```

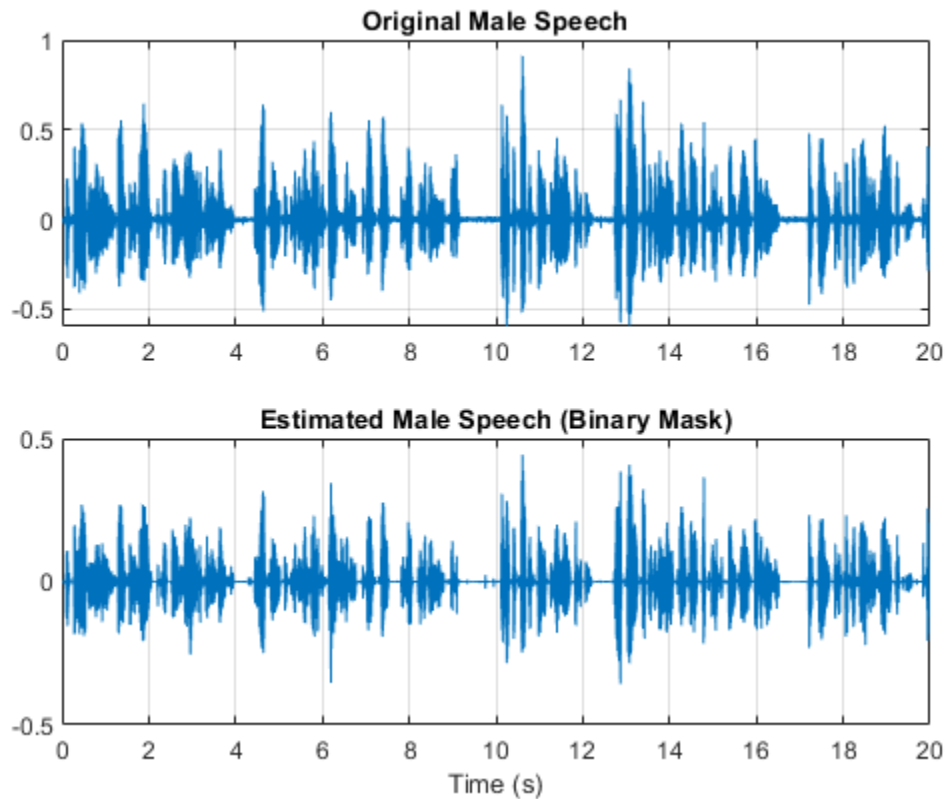
Multiply the mix STFT by the male binary mask to get the estimated male speech STFT. Use the ISTFT to get the estimated male audio signal. Scale the audio.

```
P_Male = P_Val_mix0 .* HardMaleMask;  
  
maleSpeech_est_hard = istft(P_Male, 'Window', win, 'OverlapLength', OverlapLength,...  
                             'FFTLength', FFTLength, 'ConjugateSymmetric', true,...  
                             'FrequencyRange', 'onesided');  
maleSpeech_est_hard = maleSpeech_est_hard / max(maleSpeech_est_hard);
```

Visualize the estimated and original male speech signals. Listen to the estimated binary mask male speech.

```
range = (numel(win):numel(maleSpeech_est_soft)-numel(win));  
t = range * (1/Fs);
```

```
figure(11)  
subplot(2,1,1)  
plot(t,maleSpeechValidate(range))  
title("Original Male Speech")  
grid on  
  
subplot(2,1,2)  
plot(t,maleSpeech_est_hard(range))  
xlabel("Time (s)")  
title("Estimated Male Speech (Binary Mask)")  
grid on
```



```
sound(maleSpeech_est_hard(range),Fs)
```

Multiply the mix STFT by the female binary mask to get the estimated male speech STFT. Use the ISTFT to get the estimated male audio signal. Scale the audio.

```
P_Female = P_Val_mix0 .* HardFemaleMask;

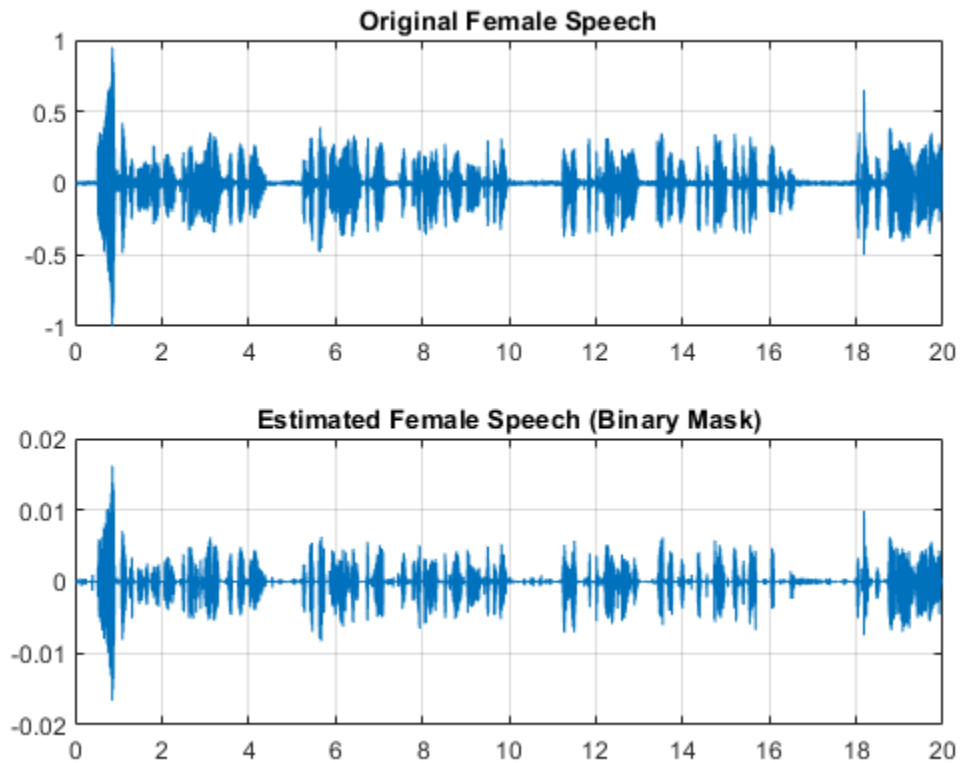
femaleSpeech_est_hard = istft(P_Female, 'Window', win, 'OverlapLength', OverlapLength,...
                              'FFTLength', FFTLength, 'ConjugateSymmetric', true,...
                              'FrequencyRange', 'onesided');
femaleSpeech_est_hard = femaleSpeech_est_hard / max(femaleSpeech_est_hard);
```

Visualize the estimated and original female speech signals. Listen to the estimated female speech.

```
range = (numel(win):numel(maleSpeech_est_soft)-numel(win));
t = range * (1/Fs);
```

```
figure(12)
subplot(2,1,1)
plot(t,femaleSpeechValidate(range))
title("Original Female Speech")
grid on

subplot(2,1,2)
plot(t,femaleSpeech_est_hard(range))
title("Estimated Female Speech (Binary Mask)")
grid on
```

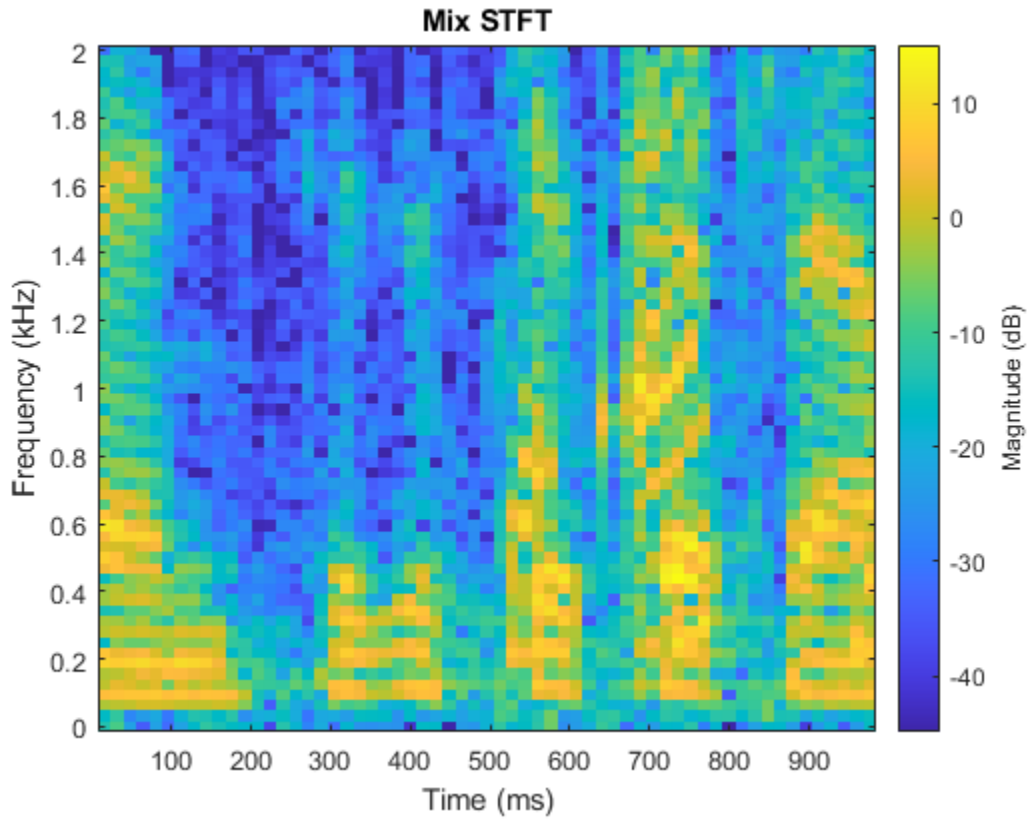


```
sound(femaleSpeech_est_hard(range),Fs)
```

Compare STFTs of a one-second segment for mix, original female and male, and estimated female and male, respectively.

```
range = 7e4:7.4e4;
```

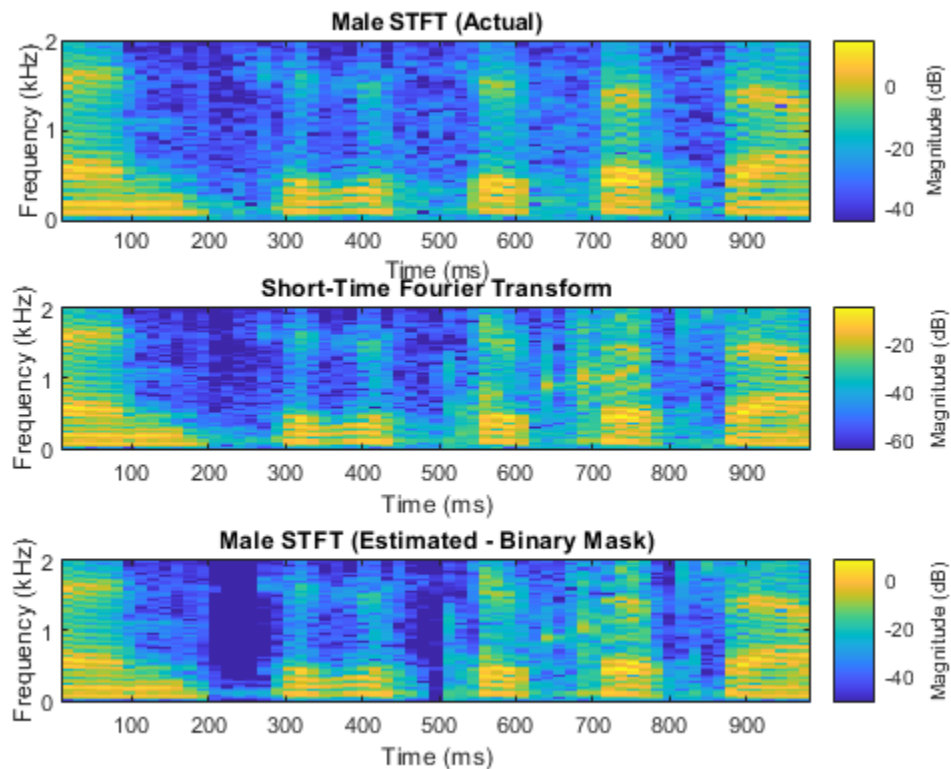
```
figure(13)
stft(mixValidate(range), Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
title("Mix STFT")
```

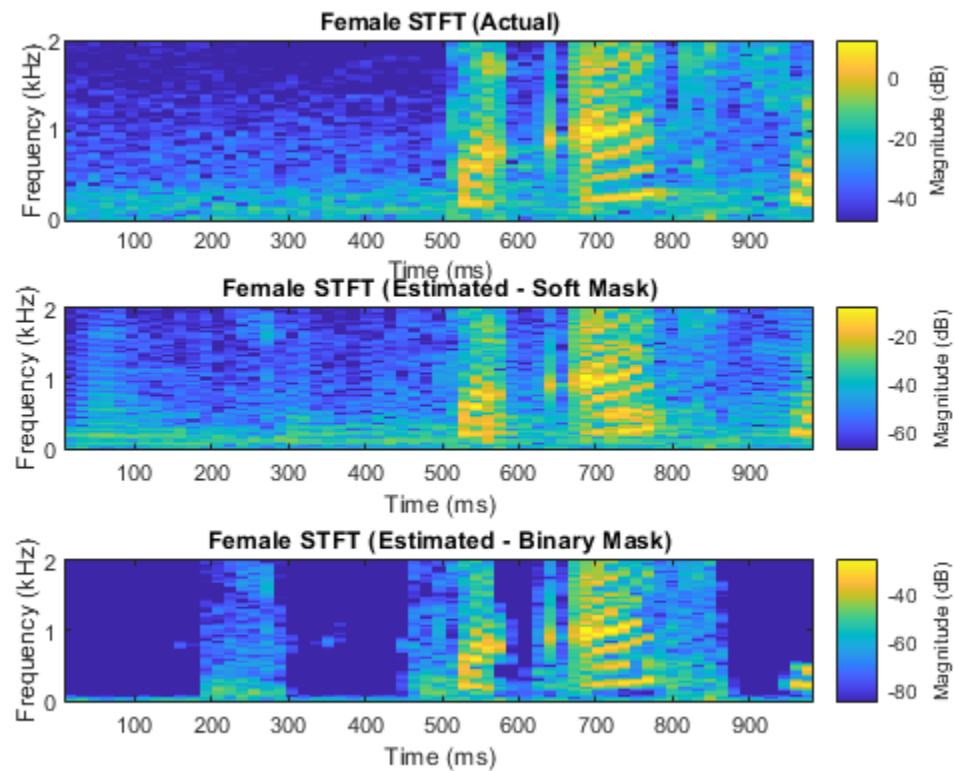
```

figure(14)
subplot(3,1,1)
stft(maleSpeechValidate(range),Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
title("Male STFT (Actual)")
subplot(3,1,2)
stft(maleSpeech_est_soft(range),Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
subplot(3,1,3)
stft(maleSpeech_est_hard(range),Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
title("Male STFT (Estimated - Binary Mask)");

```



```
figure(15)
subplot(3,1,1)
stft(femaleSpeechValidate(range),Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLenght', FFTLength, 'FrequencyRange', 'onesided');
title("Female STFT (Actual)")
subplot(3,1,2)
stft(femaleSpeech_est_soft(range),Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLenght', FFTLength, 'FrequencyRange', 'onesided');
title("Female STFT (Estimated - Soft Mask)")
subplot(3,1,3)
stft(femaleSpeech_est_hard(range),Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLenght', FFTLength, 'FrequencyRange', 'onesided');
title("Female STFT (Estimated - Binary Mask)")
```



References

- [1] "Probabilistic Binary-Mask Cocktail-Party Source Separation in a Convolutional Deep Neural Network", Andrew J.R. Simpson, 2015.

Parametric Equalizer Design

This example shows how to design parametric equalizer filters. Parametric equalizers are digital filters used in audio for adjusting the frequency content of a sound signal. Parametric equalizers provide capabilities beyond those of graphic equalizers by allowing the adjustment of gain, center frequency, and bandwidth of each filter. In contrast, graphic equalizers only allow for the adjustment of the gain of each filter.

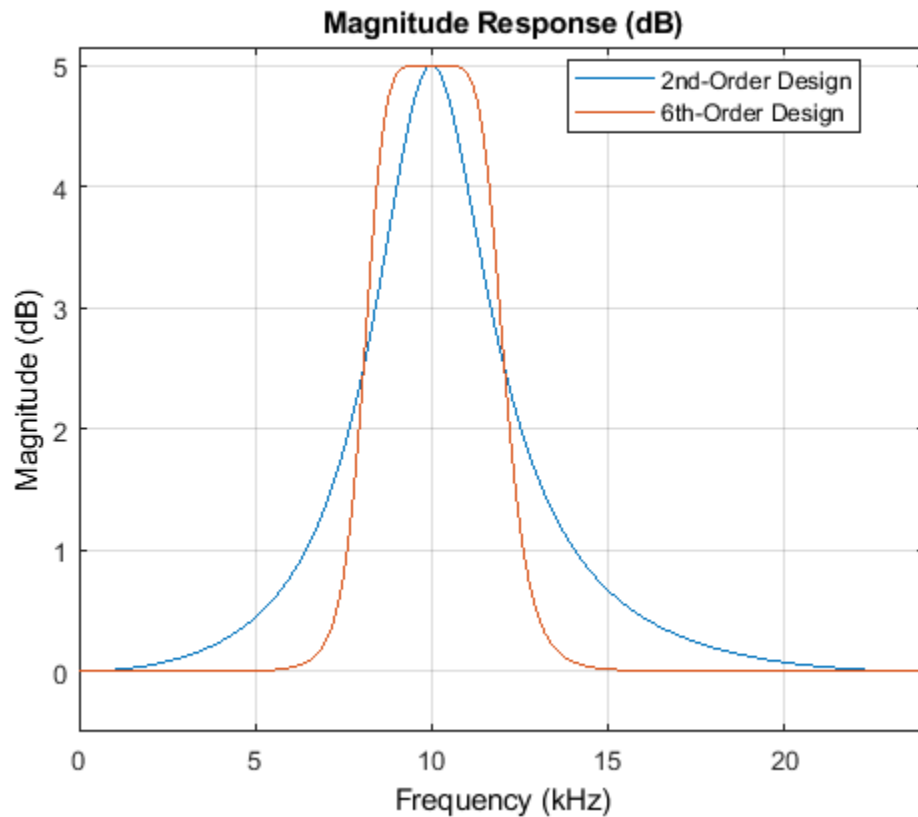
Typically, parametric equalizers are designed as second-order IIR filters. These filters have the drawback that because of their low order, they can present relatively large ripple or transition regions and may overlap with each other when several of them are connected in cascade. The DSP System Toolbox™ provides the capability to design high-order IIR parametric equalizers. Such high-order designs provide much more control over the shape of each filter. In addition, the designs special-case to traditional second-order parametric equalizers if the order of the filter is set to two.

This example discusses two separate approaches to parametric equalizer design. The first is using `designParamEQ` and the second is using `fdesign.parmeq`. `designParamEQ` should serve most needs. It is simpler and provides the ability for most common designs. It also supports C code generation which is needed if there is a desire to tune the filter at run-time with generated code. `fdesign.parmeq` provides many advanced design options for ultimate control of the resulting filter. Not all design options are explored in this example.

Some Basic Designs

Consider the following two designs of parametric equalizers. The design specifications are the same except for the filter order. The first design is a typical second-order parametric equalizer that boosts the signal around 10 kHz by 5 dB. The second design does the same with a sixth-order filter. Notice how the sixth-order filter is closer to an ideal brickwall filter when compared to the second-order design. Obviously the approximation can be improved by increasing the filter order even further. The price to pay for such improved approximation is increased implementation cost as more multipliers are required.

```
Fs = 48e3;  
N1 = 2;  
N2 = 6;  
G = 5; % 5 dB  
Wo = 10000/(Fs/2);  
BW = 4000/(Fs/2);  
[B1,A1] = designParamEQ(N1,G,Wo,BW);  
[B2,A2] = designParamEQ(N2,G,Wo,BW);  
BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[1,A1.']]');  
BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.',[ones(3,1),A2.']]');  
hfv1 = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white');  
legend(hfv1,'2nd-Order Design','6th-Order Design');
```

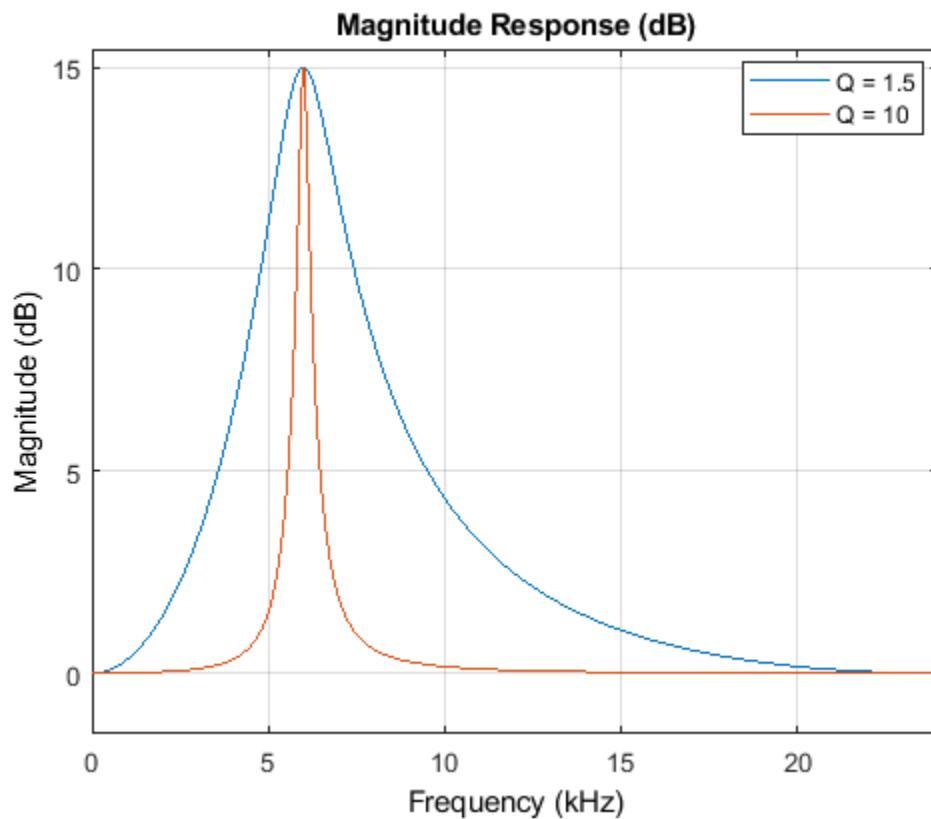


One of the design parameters is the filter bandwidth, BW. In the previous example, the bandwidth was specified as 4 kHz. The 4 kHz bandwidth occurs at half the gain (2.5 dB).

Designs Based on Quality Factor

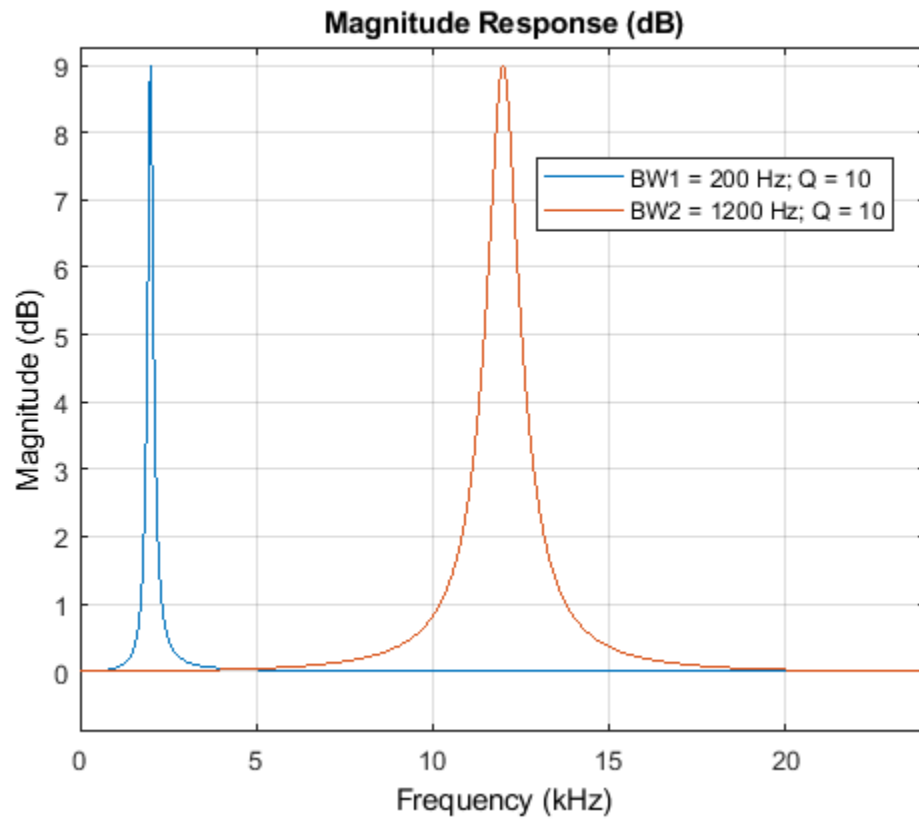
Another common design parameter is the quality factor, Q . The Q of the filter is defined as W_o/BW (center frequency/bandwidth). It provides a measure of the sharpness of the filter, i.e., how sharply the filter transitions between the reference value (0 dB) and the gain G . Consider two designs with same G and W_o , but different Q values.

```
Fs = 48e3;
N = 2;
Q1 = 1.5;
Q2 = 10;
G = 15; % 15 dB
Wo = 6000/(Fs/2);
BW1 = Wo/Q1;
BW2 = Wo/Q2;
[B1,A1] = designParamEQ(N,G,Wo,BW1);
[B2,A2] = designParamEQ(N,G,Wo,BW2);
BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[1,A1.]]);
BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.',[1,A2.]]);
hfv = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white');
legend(hfv,'Q = 1.5','Q = 10');
```



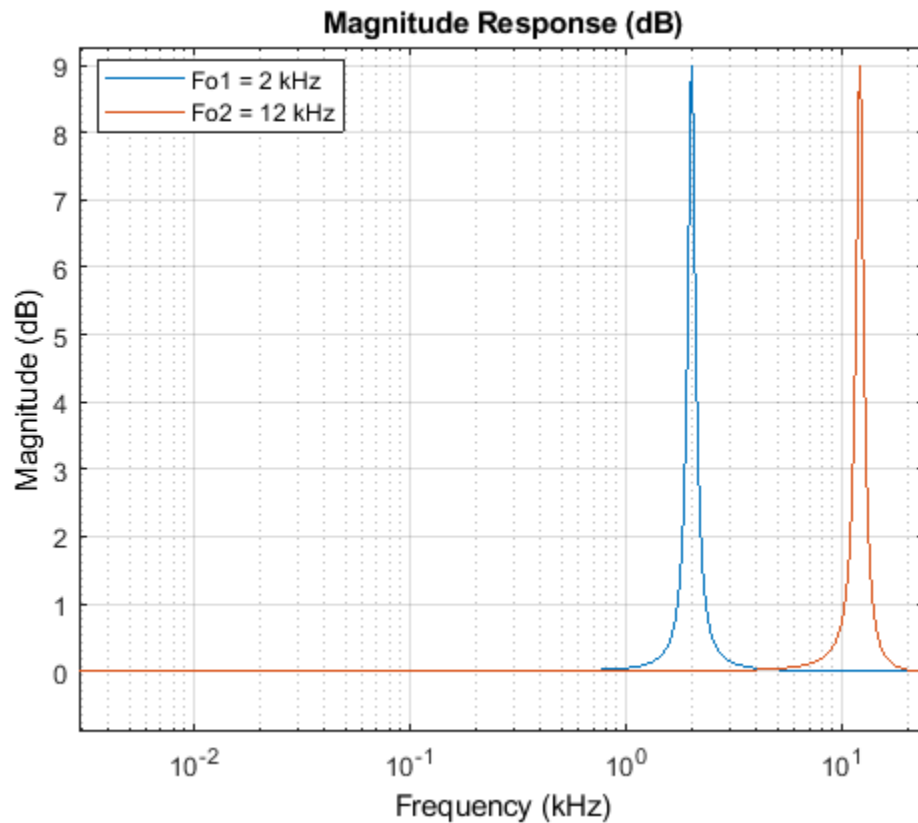
Although a higher Q factor corresponds to a sharper filter, it must also be noted that for a given bandwidth, the Q factor increases simply by increasing the center frequency. This might seem unintuitive. For example, the following two filters have the same Q factor, but one clearly occupies a larger bandwidth than the other.

```
Fs = 48e3;
N = 2;
Q = 10;
G = 9; % 9 dB
Wo1 = 2000/(Fs/2);
Wo2 = 12000/(Fs/2);
BW1 = Wo1/Q;
BW2 = Wo2/Q;
[B1,A1] = designParamEQ(N,G,Wo1,BW1);
[B2,A2] = designParamEQ(N,G,Wo2,BW2);
BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[1,A1.]]);
BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.',[1,A2.]]);
hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white');
legend(hfvt,'BW1 = 200 Hz; Q = 10','BW2 = 1200 Hz; Q = 10');
```



When viewed on a log-frequency scale though, the "octave bandwidth" of the two filters is the same.

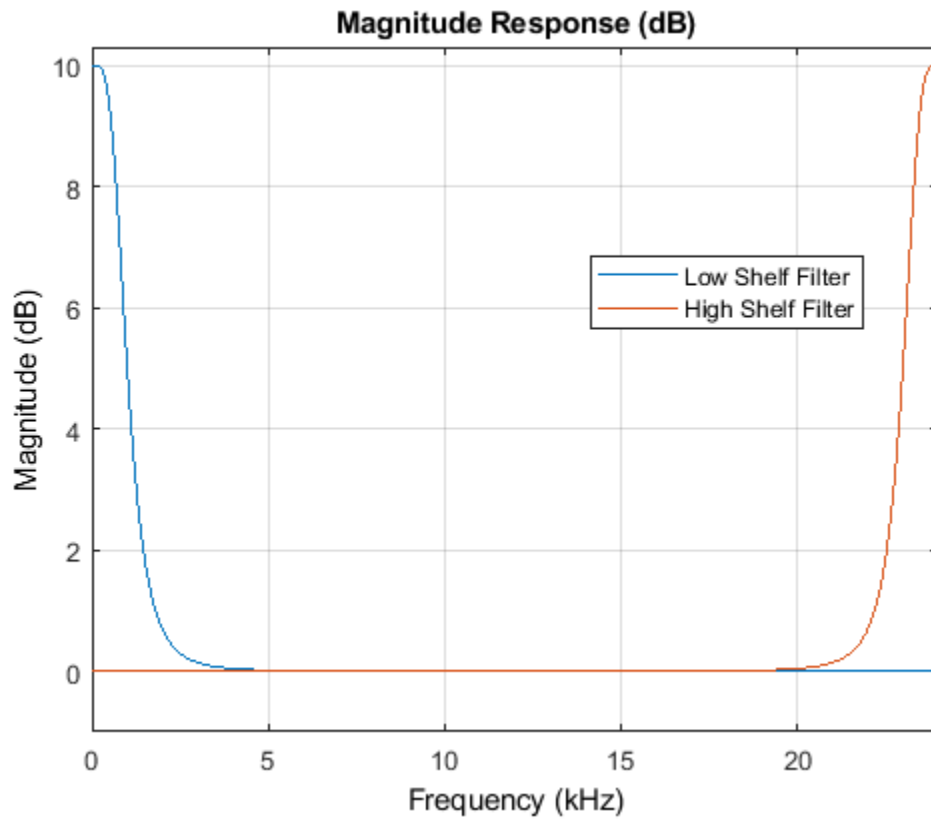
```
hfvt = fvtool(BQ1,BQ2,'FrequencyScale','log','Fs',Fs,'Color','white');  
legend(hfvt,'Fo1 = 2 kHz','Fo2 = 12 kHz');
```



Low Shelf and High Shelf Filters

The filter's bandwidth BW is only perfectly centered around the center frequency Wo when such frequency is set to 0.5π (half the Nyquist rate). When Wo is closer to 0 or to π , there is a warping effect that makes a larger portion of the bandwidth to occur at one side of the center frequency. In the edge cases, if the center frequency is set to 0 (π), the entire bandwidth of the filter occurs to the right (left) of the center frequency. The result is a so-called shelving low (high) filter.

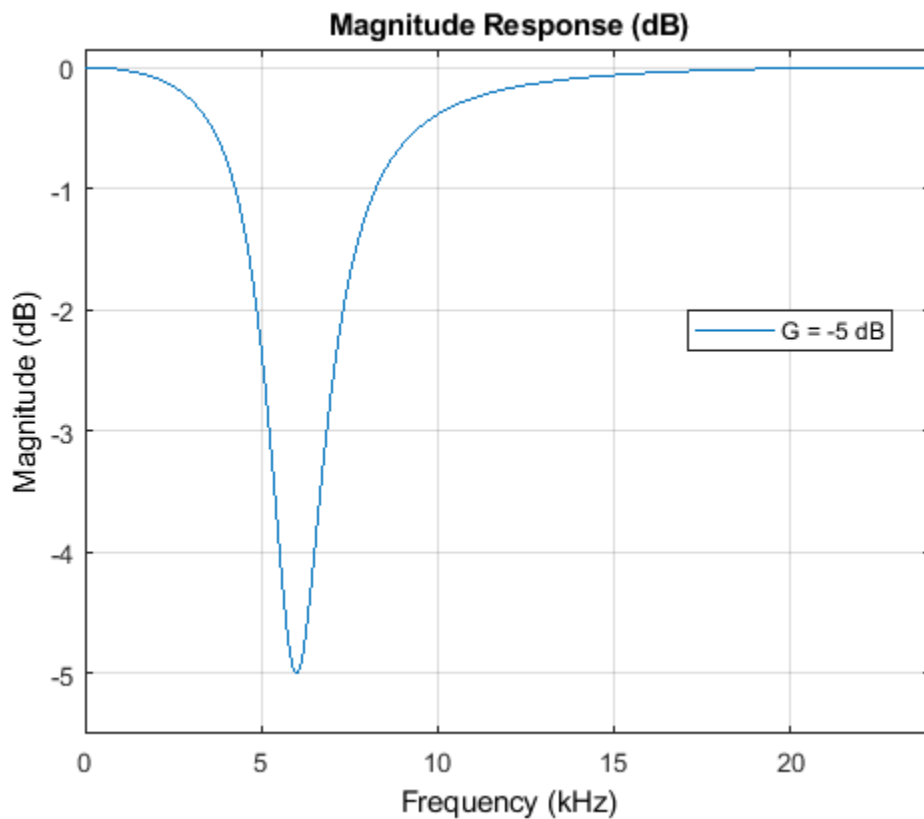
```
Fs = 48e3;
N = 4;
G = 10; % 10 dB
Wo1 = 0;
Wo2 = 1; % Corresponds to Fs/2 (Hz) or pi (rad/sample)
BW = 1000/(Fs/2); % Bandwidth occurs at 7.4 dB in this case
[B1,A1] = designParamEQ(N,G,Wo1,BW);
[B2,A2] = designParamEQ(N,G,Wo2,BW);
BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[ones(2,1),A1.']]');
BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.',[ones(2,1),A2.']]');
hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white');
legend(hfvt,'Low Shelf Filter','High Shelf Filter');
```

A Parametric Equalizer That Cuts

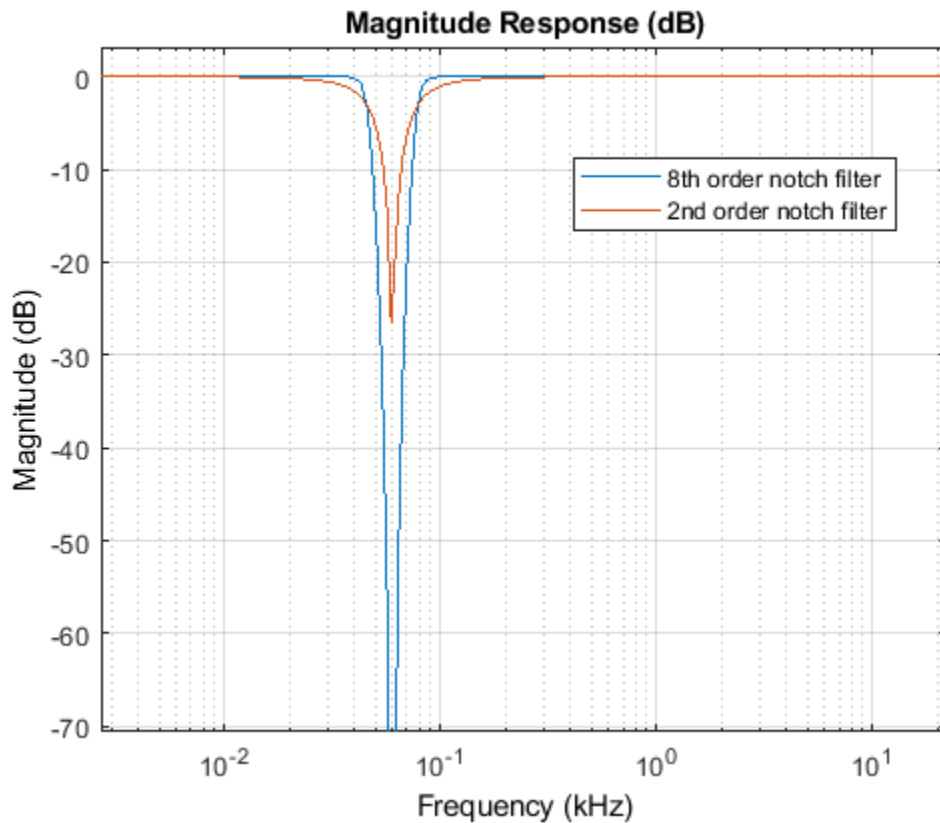
All previous designs are examples of a parametric equalizer that boosts the signal over a certain frequency band. You can also design equalizers that cut (attenuate) the signal in a given region.

```
Fs = 48e3;
N = 2;
G = -5; % -5 dB
Wo = 6000/(Fs/2);
BW = 2000/(Fs/2);
[B,A] = designParamEQ(N,G,Wo,BW);
BQ = dsp.BiquadFilter('SOSMatrix',[B.],[1,A.]);
hfvt = fvtool(BQ,'Fs',Fs,'Color','white');
legend(hfvt,'G = -5 dB');
```



At the limit, the filter can be designed to have a gain of zero ($-\infty$ dB) at the frequency specified. This allows to design 2nd order or higher order notch filters.

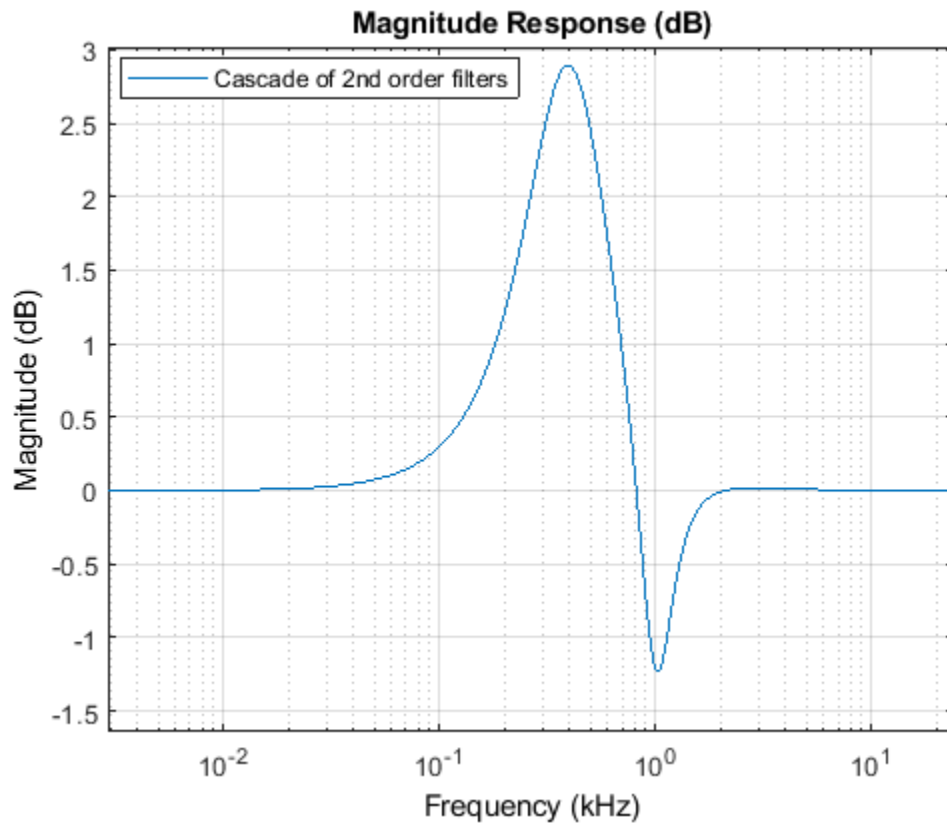
```
Fs = 44.1e3;
N = 8;
G = -inf;
Q = 1.8;
Wo = 60/(Fs/2); % Notch at 60 Hz
BW = Wo/Q; % Bandwidth will occur at -3 dB for this special case
[B1,A1] = designParamEQ(N,G,Wo,BW);
[NUM,DEN] = iirnotch(Wo,BW); % or [NUM,DEN] = designParamEQ(2,G,Wo,BW);
SOS2 = [NUM,DEN];
BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[ones(4,1),A1.']]');
BQ2 = dsp.BiquadFilter('SOSMatrix',SOS2);
hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'FrequencyScale','Log','Color','white');
legend(hfvt,'8th order notch filter','2nd order notch filter');
```



Cascading Parametric Equalizers

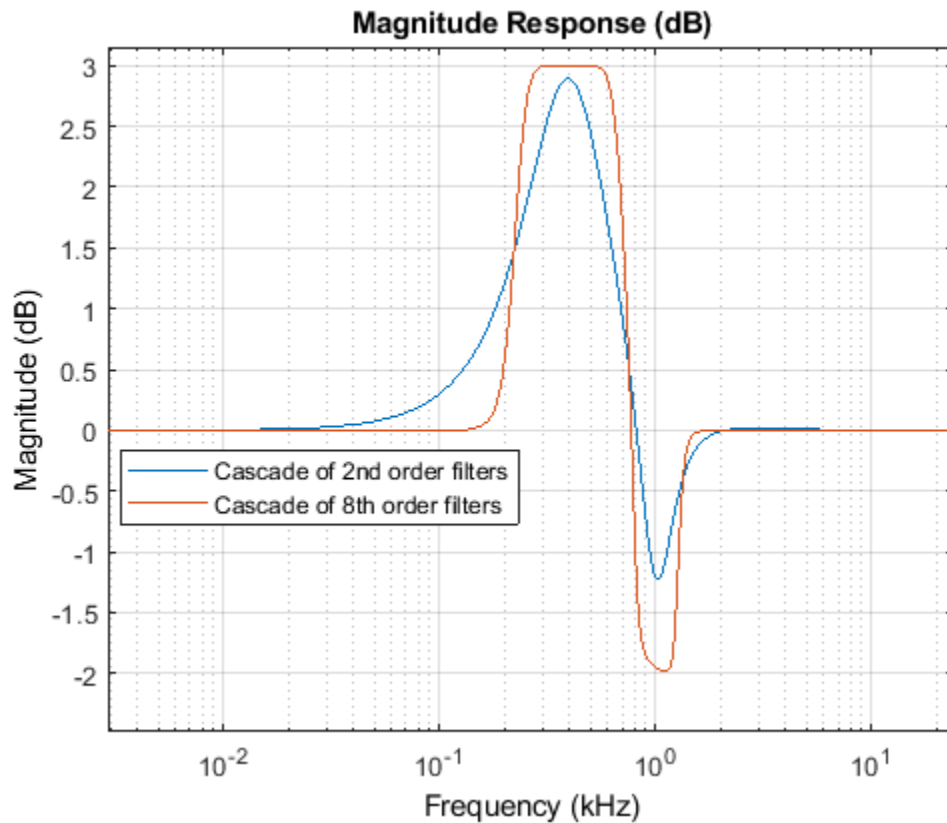
Parametric equalizers are usually connected in cascade (in series) so that several are used simultaneously to equalize an audio signal. To connect several equalizers in this way, use the `dsp.FilterCascade`.

```
Fs = 48e3;
N = 2;
G1 = 3; % 3 dB
G2 = -2; % -2 dB
Wo1 = 400/(Fs/2);
Wo2 = 1000/(Fs/2);
BW = 500/(Fs/2); % Bandwidth occurs at 7.4 dB in this case
[B1,A1] = designParamEQ(N,G1,Wo1,BW);
[B2,A2] = designParamEQ(N,G2,Wo2,BW);
BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[1,A1.]]);
BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.',[1,A2.]]);
FC = dsp.FilterCascade(BQ1,BQ2);
hfvt = fvtool(FC,'Fs',Fs,'Color','white','FrequencyScale','Log');
legend(hfvt,'Cascade of 2nd order filters');
```



Low-order designs such as the second-order filters above can interfere with each other if their center frequencies are closely spaced. In the example above, the filter centered at 1 kHz was supposed to have a gain of -2 dB. Due to the interference from the other filter, the actual gain is more like -1 dB. Higher-order designs are less prone to such interference.

```
Fs = 48e3;
N = 8;
G1 = 3; % 3 dB
G2 = -2; % -2 dB
Wo1 = 400/(Fs/2);
Wo2 = 1000/(Fs/2);
BW = 500/(Fs/2); % Bandwidth occurs at 7.4 dB in this case
[B1,A1] = designParamEQ(N,G1,Wo1,BW);
[B2,A2] = designParamEQ(N,G2,Wo2,BW);
BQ1a = dsp.BiquadFilter('SOSMatrix',[B1.',[ones(4,1),A1.]]);
BQ2a = dsp.BiquadFilter('SOSMatrix',[B2.',[ones(4,1),A2.]]);
FC2 = dsp.FilterCascade(BQ1a,BQ2a);
hfvt = fvtool(FC,FC2,'Fs',Fs,'Color','white','FrequencyScale','Log');
legend(hfvt,'Cascade of 2nd order filters','Cascade of 8th order filters');
```

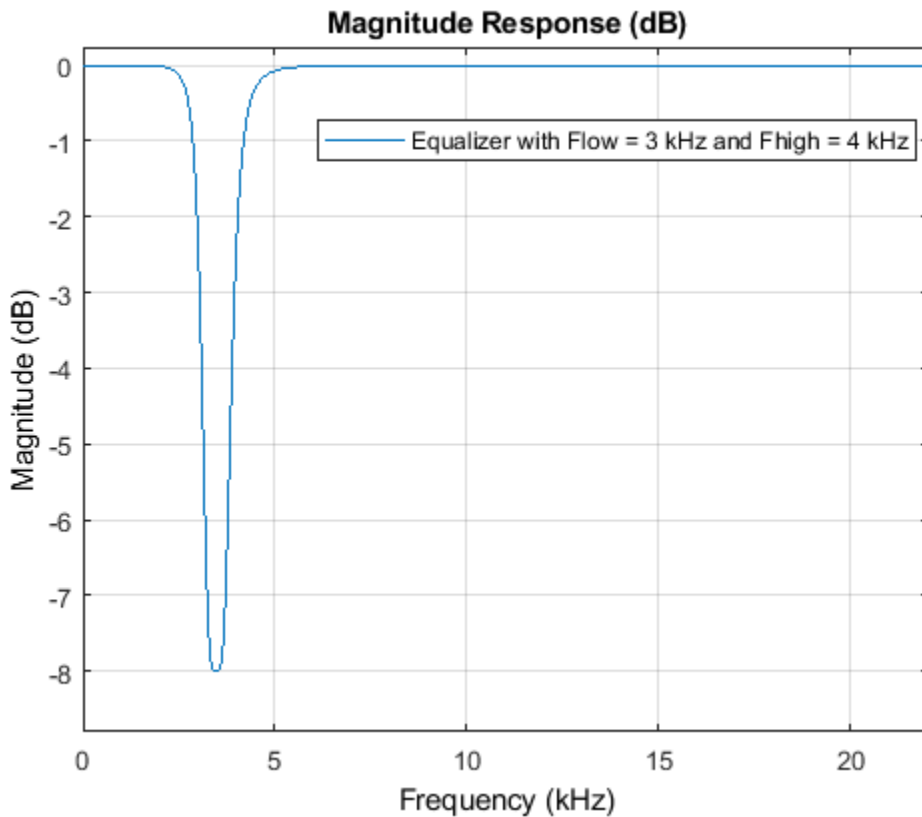


Advanced Design Options: Specifying Low and High Frequencies

For more advanced designs, `fdesign.parmeq` can be used. For example, because of frequency warping, in general it can be difficult to control the exact frequency edges at which the bandwidth occurs. To do so, the following can be used:

```
Fs    = 44.1e3;
N      = 4;
Flow   = 3000;
Fhigh  = 4000;
Grsq   = 1;
Gref   = 10*log10(Grsq);
G       = -8;
Gsq    = 10.^(G/10); % Magnitude squared of filter; G = 5 dB
GBW    = 10*log10((Gsq + Grsq)/2); % Flow and Fhigh occur at -2.37 dB

PEQ = fdesign.parmeq('N,Flow,Fhigh,Gref,G0,GBW',...
    N,Flow/(Fs/2),Fhigh/(Fs/2),Gref,G,GBW);
BQ = design(PEQ,'SystemObject',true);
hfvt = fvtool(BQ,'Fs',Fs,'Color','white');
legend(hfvt,'Equalizer with Flow = 3 kHz and Fhigh = 4 kHz');
```



Notice that the filter has a gain of -2.37 dB at 3 kHz and 4 kHz as specified.

Shelving Filters with a Variable Transition Bandwidth or Slope

One of the characteristics of a shelving filter is the transition bandwidth (sometimes also called transition slope) which may be specified by a shelf slope parameter S . The bandwidth reference gain GBW is always set to half the boost or cut gain of the shelving filter. All other parameters being constant, as S increases the transition bandwidth decreases, (and the slope of the response increases) creating a "slope rotation" around the GBW point as illustrated in the example below.

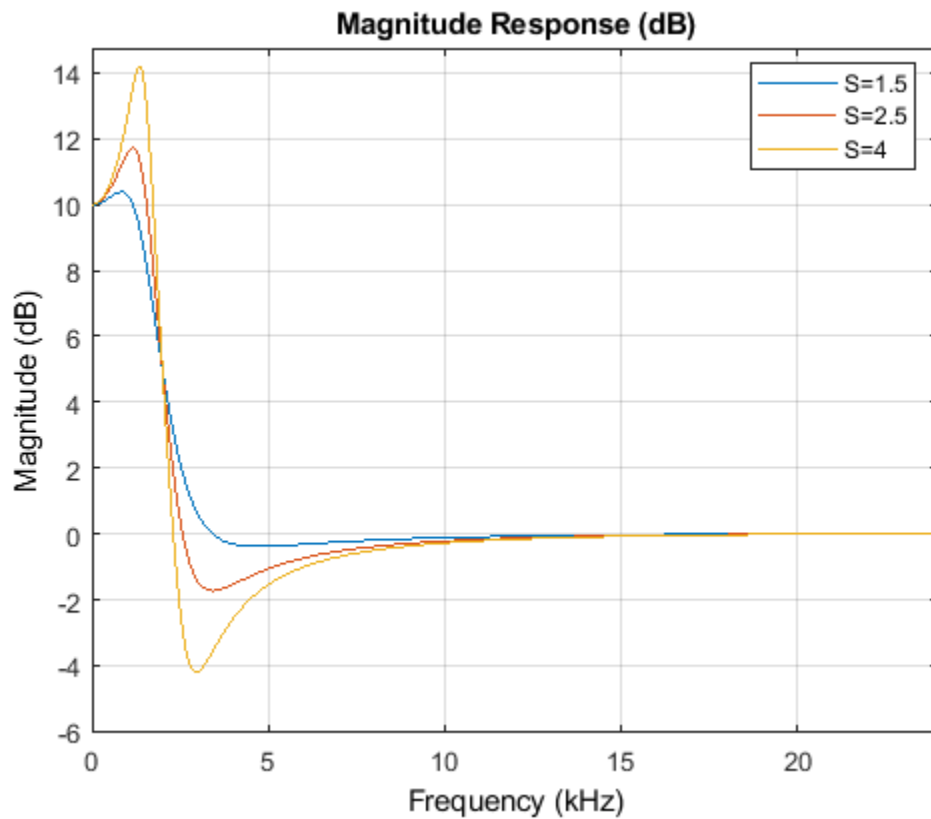
```
N = 2;
Fs = 48e3;
Fo = 0; % F0=0 designs a lowpass filter, F0=1 designs a highpass filter
Fc = 2e3/(Fs/2); % Cutoff Frequency
G = 10;

S = 1.5;
PEQ = fdesign.pareq('N,F0,Fc,S,G0',N,Fo,Fc,S,G);
BQ1 = design(PEQ,'SystemObject',true);

PEQ.S = 2.5;
BQ2 = design(PEQ,'SystemObject',true);

PEQ.S = 4;
BQ3 = design(PEQ,'SystemObject',true);
```

```
hfvt = fvtool(BQ1,BQ2,BQ3,'Fs',Fs,'Color','white');
legend(hfvt,'S=1.5','S=2.5','S=4');
```



The transition bandwidth and the bandwidth gain corresponding to each value of S can be obtained using the `measure` function. We verify that the bandwidth reference gain GBW is the same for the three designs and we quantify by how much the transition width decreases when S increases.

```
m1 = measure(BQ1);
get(m1,'GBW')

ans = 5

m2 = measure(BQ2);
get(m2,'GBW')

ans = 5

m3 = measure(BQ3);
get(m3,'GBW')

ans = 5

get(m1,'HighTransitionWidth')

ans = 0.0945

get(m2,'HighTransitionWidth')

ans = 0.0425
```

```
get(m3,'HighTransitionWidth')
```

```
ans = 0.0238
```

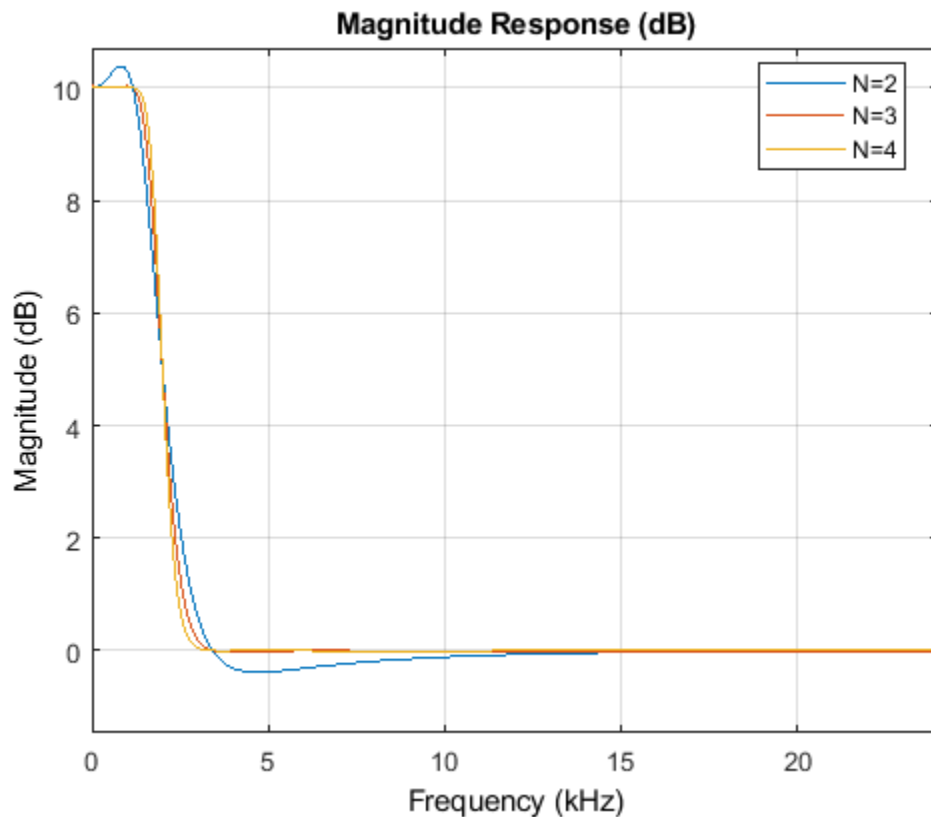
As the shelf slope parameter S increases, the ripple of the filters also increases. We can increase the filter order to reduce the ripple while maintaining the desired transition bandwidth.

```
N = 2;
PEQ = fdesign.parmeq('N,F0,Fc,S,G0',N,Fo,Fc,S,G);
BQ1 = design(PEQ,'SystemObject',true);
```

```
PEQ.FilterOrder = 3;
BQ2 = design(PEQ,'SystemObject',true);
```

```
PEQ.FilterOrder = 4;
BQ3 = design(PEQ,'SystemObject',true);
```

```
hfvt = fvtool(BQ1,BQ2,BQ3,'Fs',Fs,'Color','white');
legend(hfvt,'N=2','N=3','N=4');
```



Shelving Filters with a Prescribed Quality Factor

The quality factor Q_a may be used instead of the shelf slope parameter S to design shelving filters with variable transition bandwidths.

```
N = 2;
Fs = 48e3;
Fo = 1; % F0=0 designs a lowpass filter, F0=1 designs a highpass filter
```



```

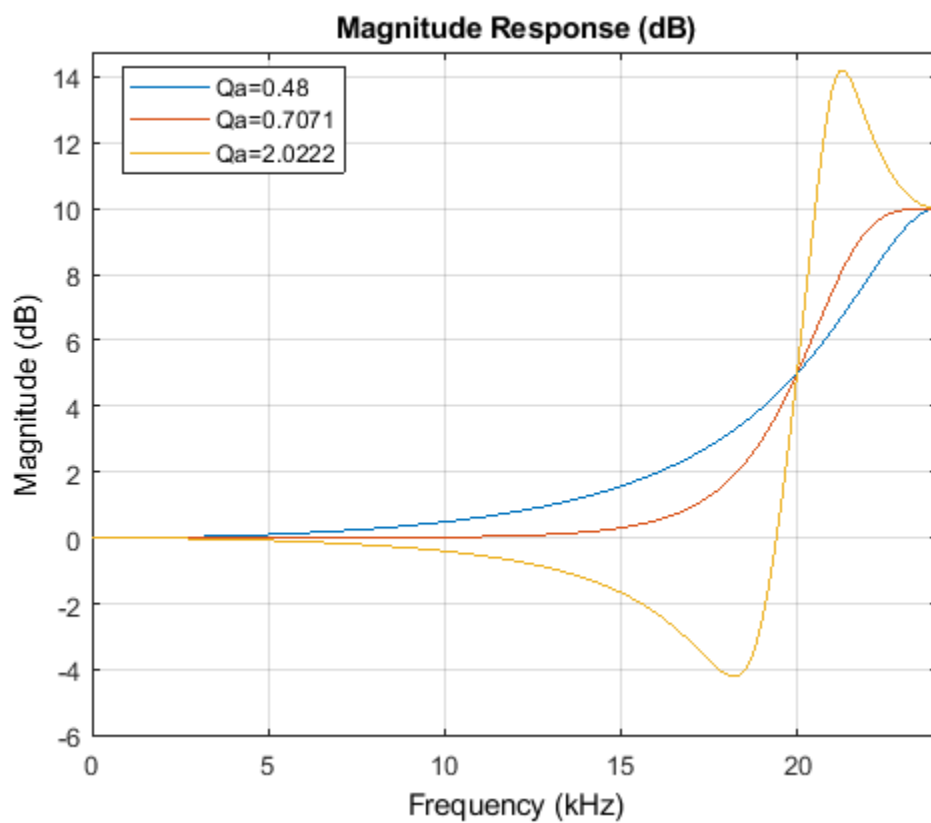
Fc = 20e3/(Fs/2); % Cutoff Frequency
G = 10;
Q = 0.48;
PEQ = fdesign.parmeq('N,F0,Fc,Qa,G0',N,F0,Fc,Q,G);
BQ1 = design(PEQ,'SystemObject',true);

PEQ.Qa = 1/sqrt(2);
BQ2 = design(PEQ,'SystemObject',true);

PEQ.Qa = 2.0222;
BQ3 = design(PEQ,'SystemObject',true);

hfvtool(BQ1,BQ2,BQ3,'Fs',Fs,'Color','white');
legend(hfvtool,'Qa=0.48','Qa=0.7071','Qa=2.0222');

```



Octave-Band and Fractional Octave-Band Filters

This example shows how to design octave-band and fractional octave-band filters, including filter banks and octave SPL meters. Octave-band and fractional-octave-band filters are commonly used in acoustics. For example, octave filters are used to perform spectral analysis for noise control. Acousticians work with octave or fractional (often 1/3) octave filter banks because it provides a meaningful measure of the noise power in different frequency bands.

Octave-Band Filter

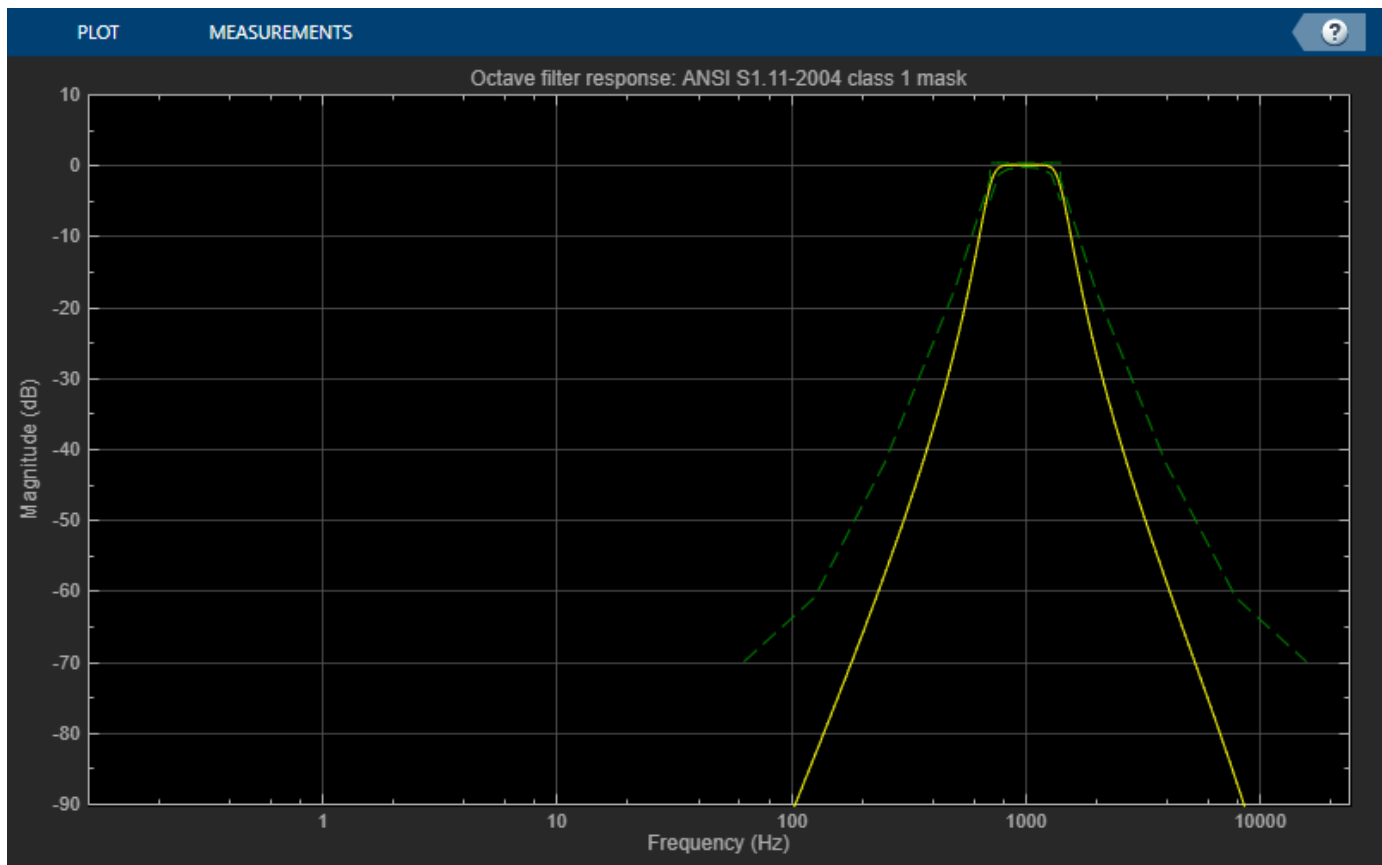
An octave is the interval between two frequencies having a ratio of 2:1 (or $10^{3/10} \approx 1.995$ for base-10 octave ratios). An octave-band or fractional-octave-band filter is a bandpass filter determined by its center frequency, order, and bandwidth. The magnitude attenuation limits are defined in the ANSI® S1.11-2004 standard for three classes of filters: class 0, class 1 and class 2. Class 0 allows only +/-0.15 dB of ripple in the passband, while class 1 allows +/-0.3 dB and class 2 allows +/-0.5 dB. Levels of stopband attenuation vary from 60 to 75 dB, depending on the class of the filter.

Design a full octave-band filter using `octaveFilter`.

```
BW = '1 octave'; % Bandwidth
N = 8;           % Filter order
F0 = 1000;       % Center frequency (Hz)
Fs = 48000;      % Sampling frequency (Hz)
of = octaveFilter('FilterOrder',N,'CenterFrequency',F0, ...
                 'Bandwidth',BW,'SampleRate',Fs);
```

Visualize the magnitude response of the filter.

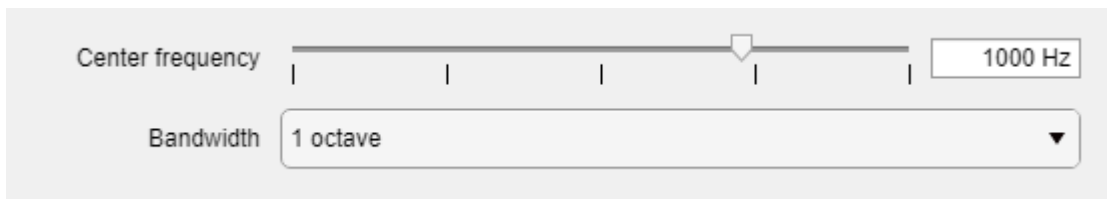
```
visualize(of,'class 1')
```



The visualizer plot is synchronized to the object, so you can see the magnitude response update as you change the filter parameters. The mask around the magnitude response is green if the filter complies with the ANSI S1.11-2004 standard (including being centered at a valid frequency), and red otherwise. To change the specifications of the filter with a graphical user interface, use `parameterTuner`. You can also use the Audio Test Bench app to quickly set up a test bench for the octave filter you designed. For example, run `audioTestBench(of)` to launch a test bench with octave filter.

Open a parameter tuner that enables you to modify the filter in real time.

`parameterTuner(of)`



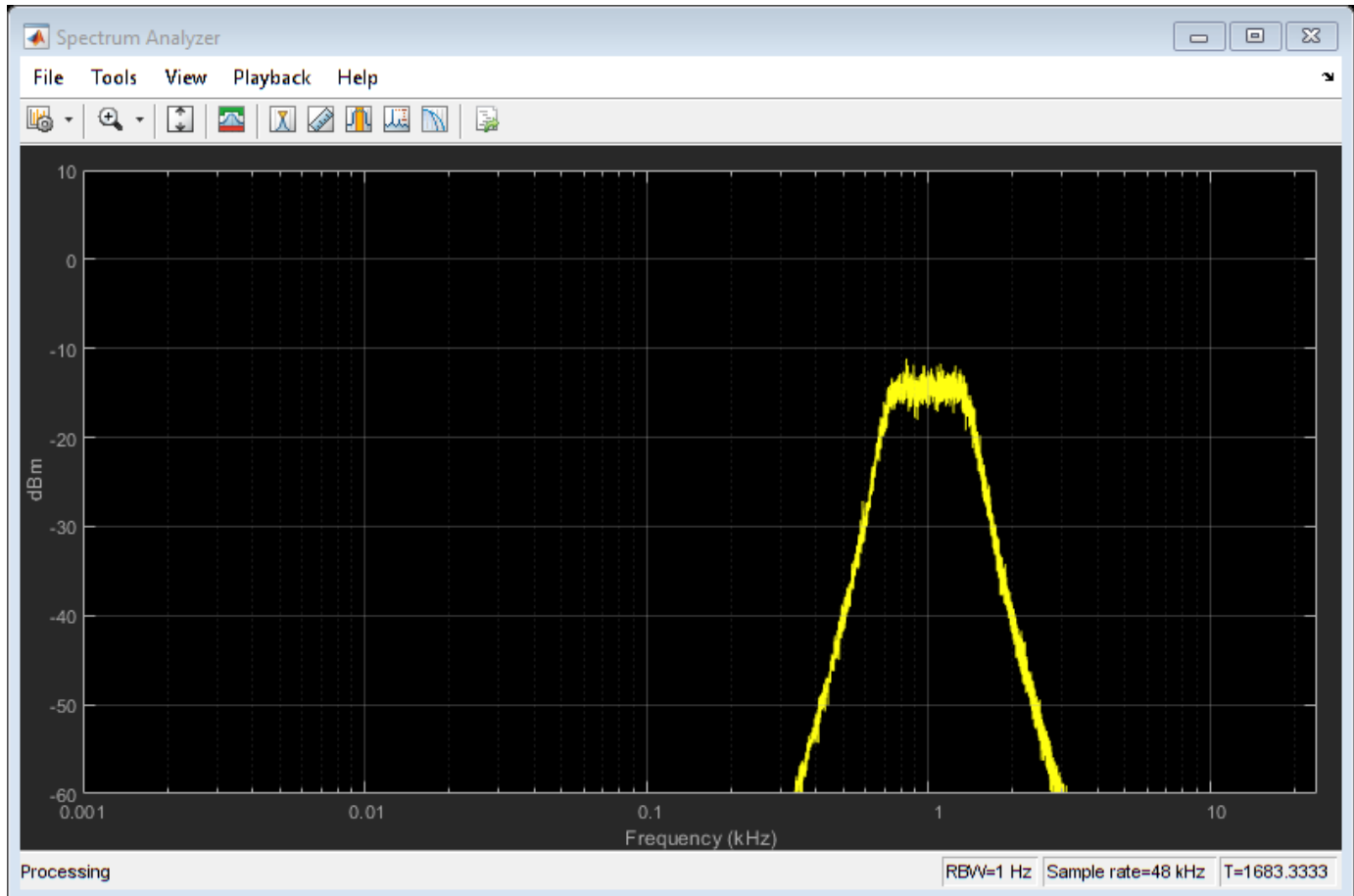
Open a spectrum analyzer to display white noise filtered by the octave filter. You can modify the filter settings with the parameter tuner while the loop runs.

```
Nx = 100000;
scope1 = dsp.SpectrumAnalyzer('SampleRate',Fs,'Method','Filter bank', ...
    'AveragingMethod','Exponential','PlotAsTwoSidedSpectrum',false, ...
```

```

'FrequencyScale','Log','FrequencySpan','Start and stop frequencies', ...
'StartFrequency',1,'StopFrequency',Fs/2,'YLimits',[-60 10], ...
'RBWSource','Property','RBW',1);
tic
while toc < 20
    % Run for 20 seconds
    x1 = randn(Nx,1);
    y1 = of(x1);
    scope1(y1)
end

```



Octave-Band Filter Bank

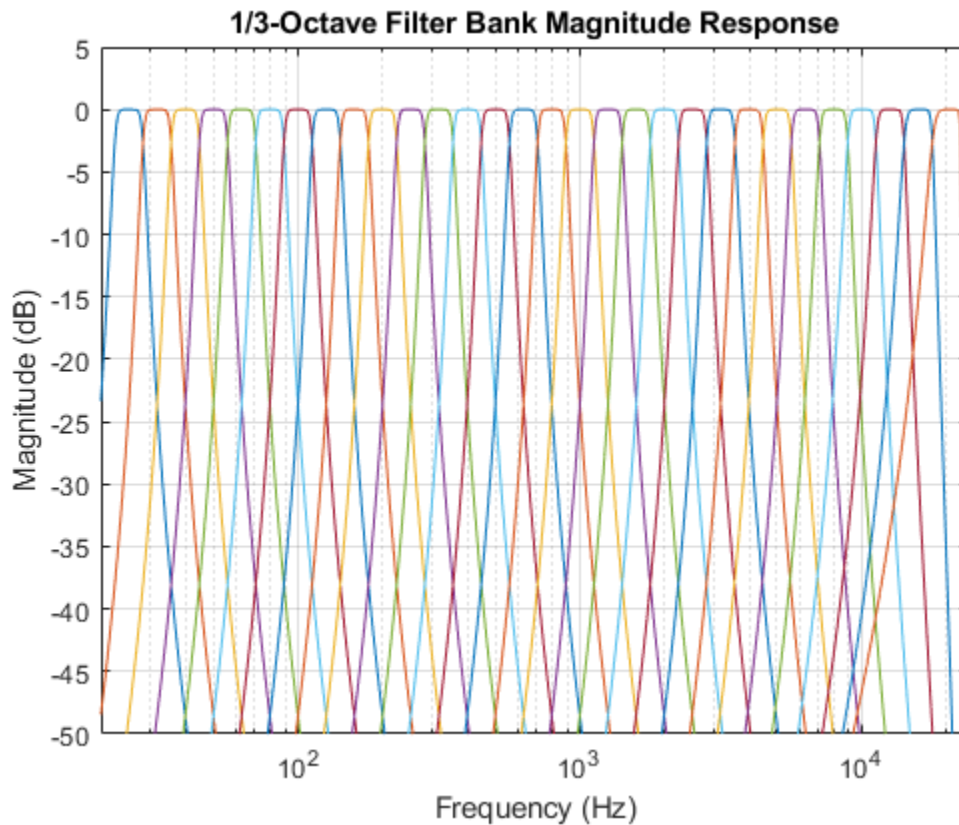
Many applications require a complete set of octave filters to form a filter bank. To design each filter manually, you would use `getANSICenterFrequencies(of)` to get a list of center frequencies for each individual filter. However, it is usually much simpler to use the `octaveFilterBank` object.

Create an `octaveFilterBank` object and plot its magnitude response.

```

ofb = octaveFilterBank('1/3 octave',Fs,'FilterOrder',N);
freqz(ofb,'NFFT',2^16) % Increase FFT length for better low-frequency resolution
set(gca,'XScale','log')
axis([20 Fs/2 -50 5])
title('1/3-Octave Filter Bank Magnitude Response')

```



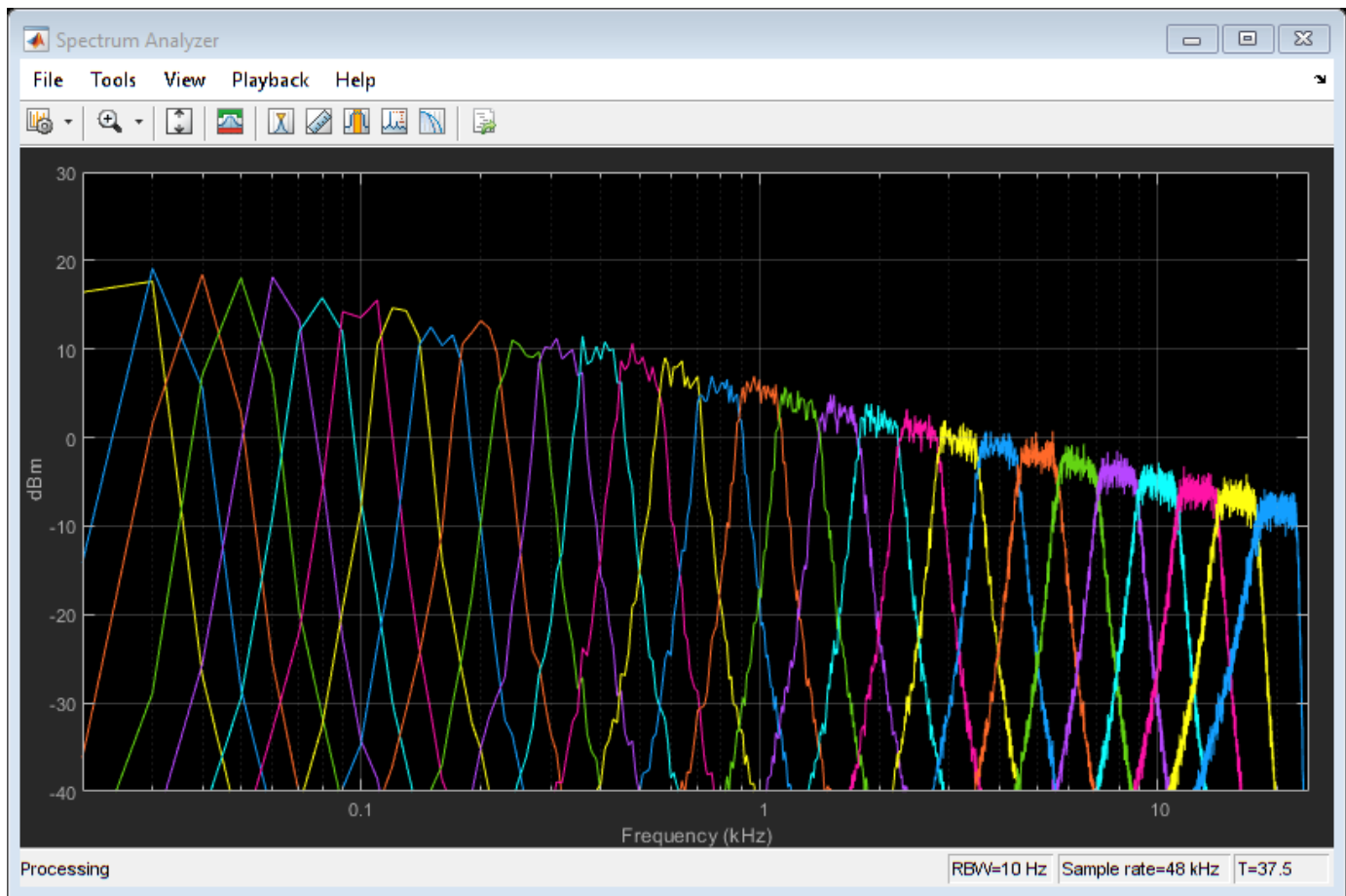
Filter the output of a pink noise generator with the 1/3-octave filter bank and compute the total power at the output of each filter.

```
pinkNoise = dsp.ColoredNoise('Color','pink', ...
                             'SamplesPerFrame',Nx, ...
                             'NumChannels',1);

scope2 = dsp.SpectrumAnalyzer('SampleRate',Fs,'Method','Filter bank', ...
                              'AveragingMethod','Exponential','PlotAsTwoSidedSpectrum',false, ...
                              'FrequencyScale','Log','FrequencySpan','Start and stop frequencies', ...
                              'StartFrequency',20,'StopFrequency',Fs/2,'YLimits',[-40 30], ...
                              'RBWSource','Property','RBW',10);

centerOct = getCenterFrequencies(ofb);
nbOct = numel(centerOct);
bandPower = zeros(1,nbOct);
nbSamples = 0;

tic
while toc < 10
    xp = pinkNoise();
    yp = ofb(xp);
    bandPower = bandPower + sum(yp.^2,1);
    nbSamples = nbSamples + Nx;
    scope2(yp)
end
```

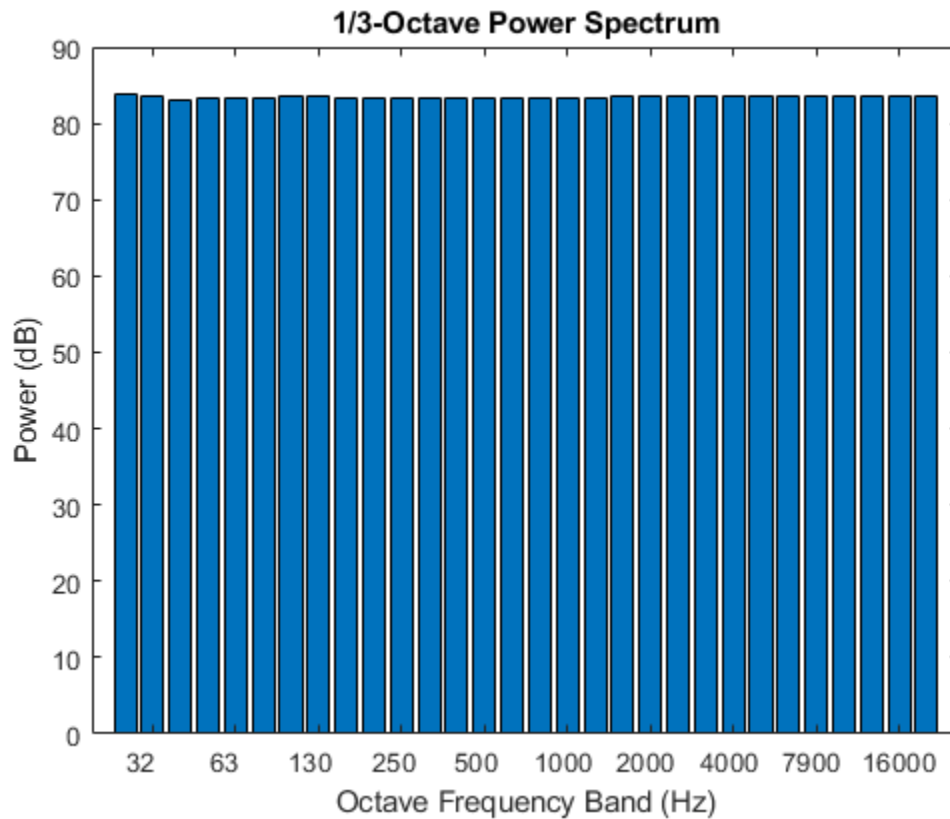


Pink noise has the same total power in each octave band, so the power between 5 Hz and 10 Hz is the same as between 5,000 Hz and 10,000 Hz. Consequently, in the spectrum analyzer, you can observe the 10 dB/decade fall-off that is characteristic of pink noise on a log-log scale, and how that signal is split into the 30 1/3-octave bands. The higher frequency bands have less power density, but the log scale means that they are also wider, so that their total power is constant.

Plot the power spectrum to show that pink noise has a flat octave spectrum.

```
b = 10^(3/10); % base-10 octave ratio
% Compute power (including pressure reference)
octPower = 10*log10(bandPower/nbSamples/4e-10);

bar(log(centerOct)/log(b),octPower);
set(gca,'Xticklabel',round(b.^get(gca,'Xtick'),2,'significant'));
title('1/3-Octave Power Spectrum')
xlabel('Octave Frequency Band (Hz)')
ylabel('Power (dB)')
```



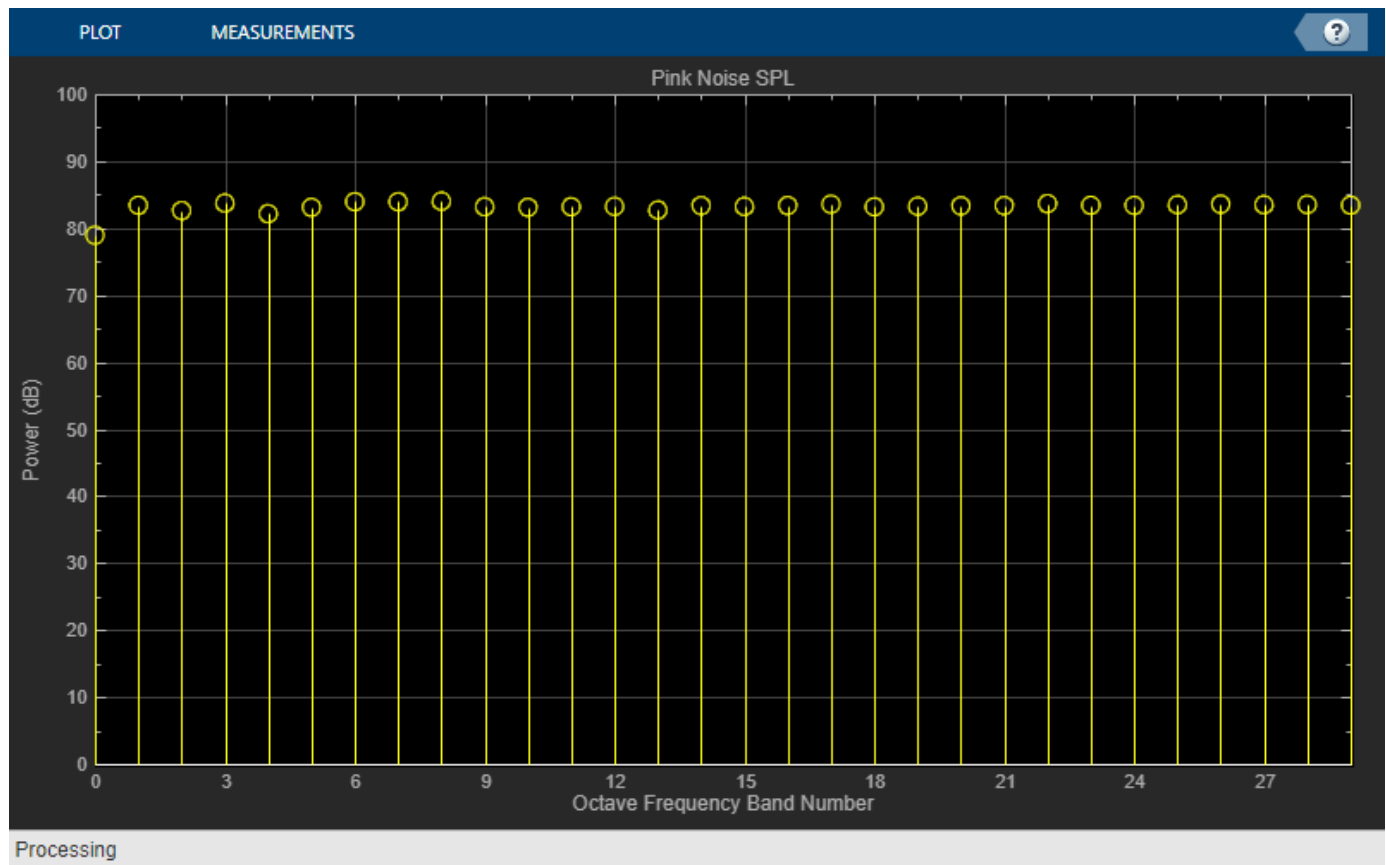
Octave SPL

The SPL Meter object (`splMeter`) also supports octave-band measurements. Reproduce the same power spectrum measurement in real time. Use a `dsp.ArrayPlot` object to visualize the power per band. Use the Z-weighting option to omit the frequency weighting filter.

```
spl = splMeter('Bandwidth','1/3 octave', ...
               'OctaveFilterOrder',N, ...
               'SampleRate',Fs, ...
               'FrequencyWeighting','Z-weighting');

scope3 = dsp.ArrayPlot('Title','Pink Noise SPL', ...
                       'XLabel','Octave Frequency Band Number', ...
                       'YLabel','Power (dB)','YLimits',[0 100]);

tic
while toc < 10
    xp = pinkNoise();
    yp = spl(xp);
    ypm = mean(yp,1).';
    scope3(ypm)
end
```



Pitch Tracking Using Multiple Pitch Estimations and HMM

This example shows how to perform pitch tracking using multiple pitch estimations, octave and median smoothing, and a hidden Markov model (HMM).

Introduction

Pitch detection is a fundamental building block in speech processing, speech coding, and music information retrieval (MIR). In speech and speaker recognition, pitch is used as a feature in a machine learning system. For call centers, pitch is used to indicate the emotional state and gender of customers. In speech therapy, pitch is used to indicate and analyze pathologies and diagnose physical defects. In MIR, pitch is used to categorize music, for query-by-humming systems, and as a primary feature in song identification systems.

Pitch detection for clean speech is mostly considered a solved problem. Pitch detection with noise and multi-pitch tracking remain difficult problems. There are many algorithms that have been extensively reported on in the literature with known trade-offs between computational cost and robustness to different types of noise.

Usually, a pitch detection algorithm (PDA) estimates the pitch for a given time instant. The pitch estimate is then validated or corrected within a pitch tracking system. Pitch tracking systems enforce continuity of pitch estimates over time.

This example provides an example function, `HelperPitchTracker`, which implements a pitch tracking system. The example walks through the algorithm implemented by the `HelperPitchTracker` function.

Problem Summary

Load an audio file and corresponding reference pitch for the audio file. The reference pitch is reported every 10 ms and was determined as an average of several third-party algorithms on the clean speech file. Regions without voiced speech are represented as `nan`.

```
[x,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
load TruePitch.mat truePitch
```

Use the `pitch` function to estimate the pitch of the audio over time.

```
[f0,locs] = pitch(x,fs);
```

Two metrics are commonly reported when defining pitch error: gross pitch error (GPE) and voicing decision error (VDE). Because the pitch algorithms in this example do not provide a voicing decision, only GPE is reported. In this example, GPE is calculated as the percent of pitch estimates outside $\pm 10\%$ of the reference pitch over the span of the voiced segments.

Calculate the GPE for regions of speech and plot the results. Listen to the clean audio signal.

```
isVoiced = ~isnan(truePitch);
f0(~isVoiced) = nan;

p = 0.1;
GPE = mean(abs(f0(isVoiced)-truePitch(isVoiced)) > truePitch(isVoiced).*p).*100;

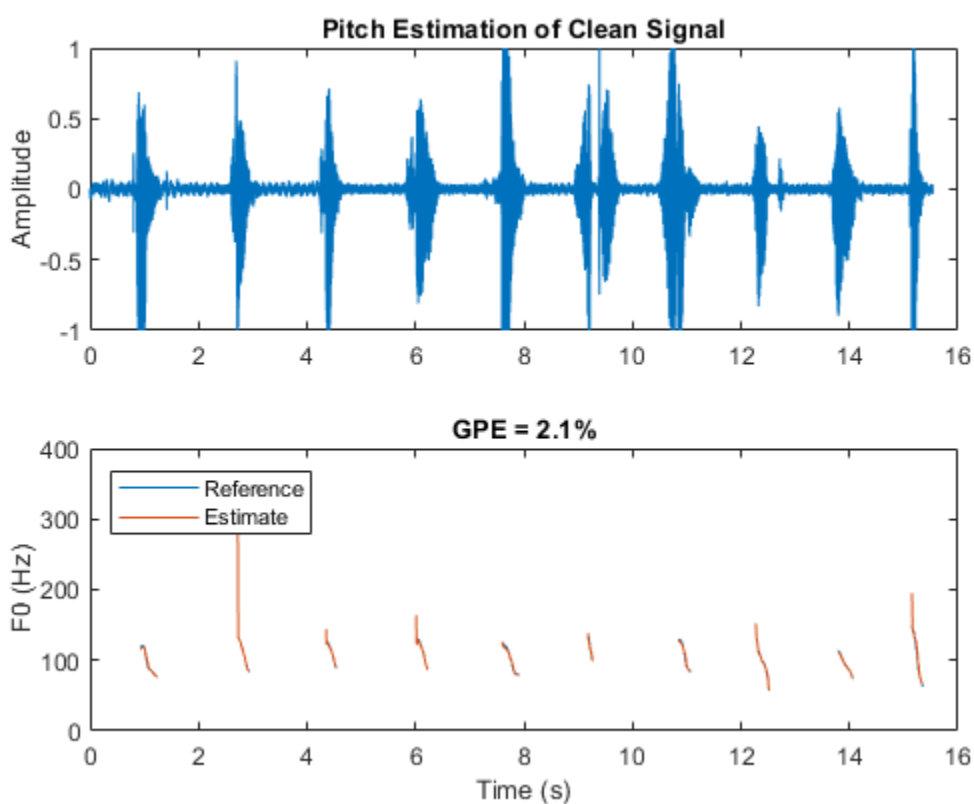
t = (0:length(x)-1)/fs;
t0 = (locs-1)/fs;
sound(x,fs)
```

```

figure(1)
tiledlayout(2,1)
nexttile
plot(t,x)
ylabel('Amplitude')
title('Pitch Estimation of Clean Signal')

nexttile
plot(t0,[truePitch,f0])
legend('Reference','Estimate','Location','NW')
ylabel('F0 (Hz)')
xlabel('Time (s)')
title(sprintf('GPE = %0.1f%%',GPE))

```



Mix the speech signal with noise at -5 dB SNR.

Use the `pitch` function on the noisy audio to estimate the pitch over time. Calculate the GPE for regions of voiced speech and plot the results. Listen to the noisy audio signal.

```

desiredSNR = -5;
x = mixSNR(x,rand(size(x)),desiredSNR);

[f0,locs] = pitch(x,fs);
f0(~isVoiced) = nan;
GPE = mean(abs(f0(isVoiced) - truePitch(isVoiced)) > truePitch(isVoiced).*p).*100;

sound(x,fs)

```

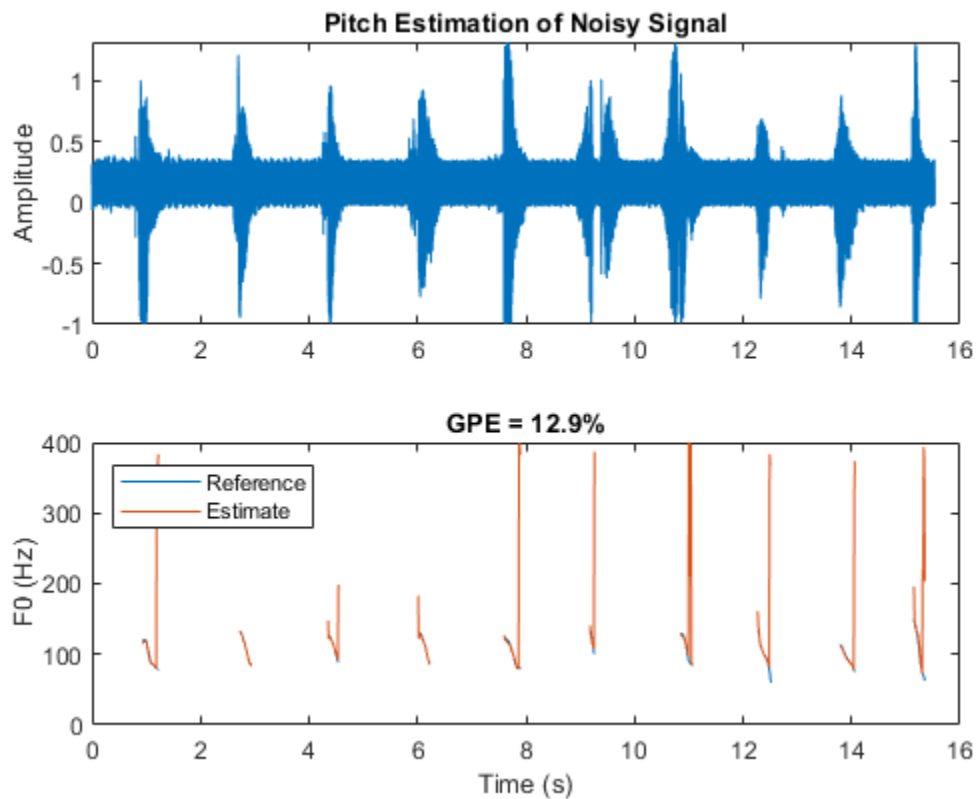
```

figure(2)
tiledlayout(2,1)

nexttile
plot(t,x)
ylabel('Amplitude')
title('Pitch Estimation of Noisy Signal')

nexttile
plot(t0,[truePitch,f0])
legend('Reference','Estimate','Location','NW')
ylabel('F0 (Hz)')
xlabel('Time (s)')
title(sprintf('GPE = %0.1f%%',GPE))

```



This example shows how to improve the pitch estimation of noisy speech signals using multiple pitch candidate generation, octave-smoothing, median-smoothing, and an HMM.

The algorithm described in this example is implemented in the example function `HelperPitchTracker`. To learn about the `HelperPitchTracker` function, enter `help HelperPitchTracker` at the command line.

`help HelperPitchTracker`

```

HelperPitchTracker Track the fundamental frequency of audio signal
f0 = HelperPitchTracker(audioIn,fs) returns an estimate of the
fundamental frequency contour for the audio input. Columns of the

```

input are treated as individual channels. The `HelperPitchTracker` function uses multiple pitch detection algorithms to generate pitch candidates, and uses octave smoothing and a Hidden Markov Model to return an estimate of the fundamental frequency.

`f0 = HelperPitchTracker(...,'HopLength',HOPLength)` specifies the number of samples in each hop. The pitch estimate is updated every hop. Specify `HOPLength` as a scalar integer. If unspecified, `HOPLength` defaults to `round(0.01*fs)`.

`f0 = HelperPitchTracker(...,'OctaveSmoothing',TF)` specifies whether or not to apply octave smoothing. Specify as true or false. If unspecified, `TF` defaults to true.

`f0 = HelperPitchTracker(...,'EmissionMatrix',EMISSIONMATRIX)` specifies the emission matrix used for the HMM during the forward pass. The default emission matrix was trained on the Pitch Tracking Database from Graz University of Technology. The database consists of 4720 speech segments with corresponding pitch trajectories derived from laryngograph signals. The emission matrix corresponds to the probability that a speaker leaves one pitch state to another, in the range [50, 400] Hz. Specify the emission matrix such that rows correspond to the current state, columns correspond to the possible future state, and the values of the matrix correspond to the probability of moving from the current state to the future state. If you specify your own emission matrix, specify its corresponding `EMISSIONMATRIXRANGE`. `EMISSIONMATRIX` must be a real N-by-N matrix of integers.

`f0 = HelperPitchTracker(...,'EmissionMatrixRange',EMISSIONMATRIXRANGE)` specifies how the `EMISSIONMATRIX` corresponds to Hz. If unspecified, `EMISSIONMATRIXRANGE` defaults to 50:400.

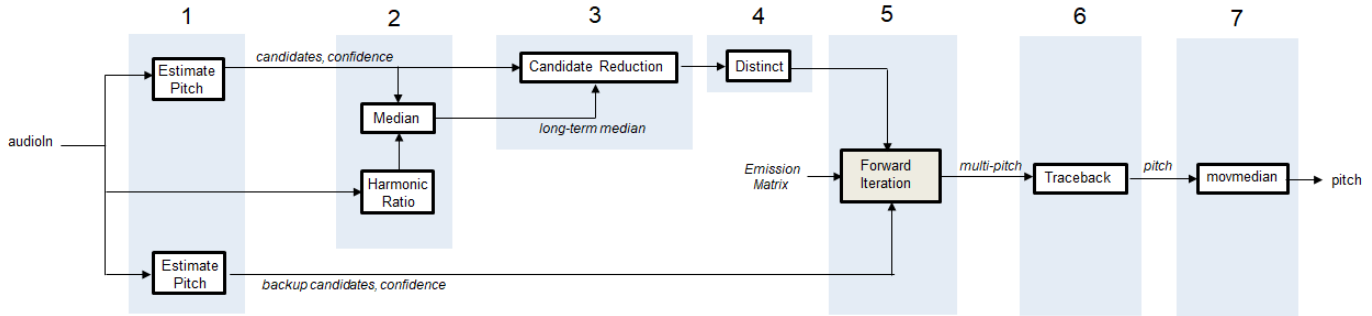
`[f0,loc] = HelperPitchTracker(...)` returns the locations associated with each pitch decision. The locations correspond to the ceiling of the center of the analysis frames.

`[f0,loc,hr] = HelperPitchTracker(...)` returns the harmonic ratio associated with each pitch decision.

See also `pitch`, `voiceActivityDetector`

Description of Pitch Tracking System

The graphic provides an overview of the pitch tracking system implemented in the example function. The following code walks through the internal workings of the `HelperPitchTracker` example function.



1. Generate Multiple Pitch Candidates

In the first stage of the pitch tracking system, you generate multiple pitch candidates using multiple pitch detection algorithms. The primary pitch candidates, which are generally more accurate, are generated using algorithms based on the Summation of Residual Harmonics (SRH) [2 on page 1-0] algorithm and the Pitch Estimation Filter with Amplitude Compression (PEFAC) [3 on page 1-0] algorithm.

Buffer the noisy input signal into overlapped frames, and then use `audio.internal.pitch.SRH` to generate 5 pitch candidates for each hop. Also return the relative confidence of each pitch candidate. Plot the results.

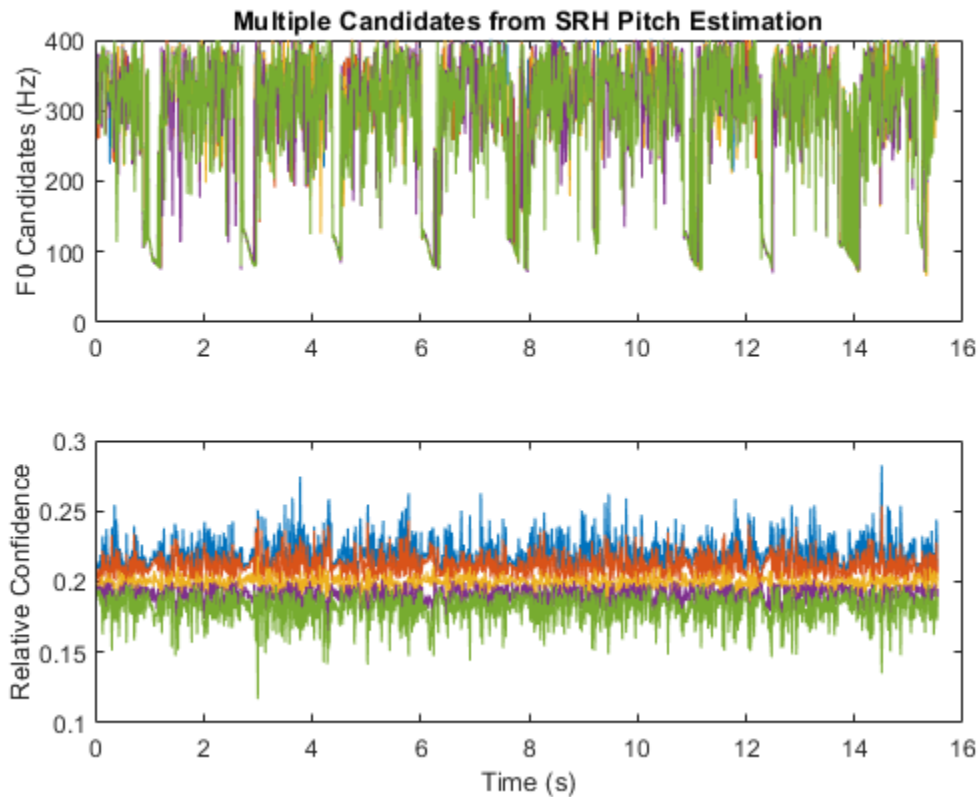
```
RANGE = [50,400];
HOPLength = round(fs.*0.01);

% Buffer into required sizes
xBuff_SRH = buffer(x,round(0.025*fs),round(0.02*fs),'nodelay');

% Define pitch parameters
params_SRH = struct('Method','SRH', ...
    'Range',RANGE, ...
    'WindowLength',round(fs*0.06), ...
    'OverlapLength',round(fs*0.06-HOPLength), ...
    'SampleRate',fs, ...
    'NumChannels',size(x,2), ...
    'SamplesPerChannel',size(x,1));
multiCandidate_params_SRH = struct('NumCandidates',5,'MinPeakDistance',1);

% Get pitch estimate and confidence
[f0_SRH,conf_SRH] = audio.internal.pitch.SRH(xBuff_SRH,x, ...
    params_SRH, ...
    multiCandidate_params_SRH);

figure(3)
tiledlayout(2,1)
nexttile
plot(t0,f0_SRH)
ylabel('F0 Candidates (Hz)')
title('Multiple Candidates from SRH Pitch Estimation')
nexttile
plot(t0,conf_SRH)
ylabel('Relative Confidence')
xlabel('Time (s)')
```



Generate an additional set of primary pitch candidates and associated confidence using the PEF algorithm. Generate backup candidates and associated confidences using the normalized correlation function (NCF) algorithm and cepstrum pitch determination (CEP) algorithm. Log only the most confident estimate from the backup candidates.

```
xBuff_PEF = buffer(x,round(0.06*fs),round(0.05*fs),'nodelay');
params_PEF = struct('Method','PEF', ...
    'Range',RANGE, ...
    'WindowLength',round(fs*0.06), ...
    'OverlapLength',round(fs*0.06-HOPLength), ...
    'SampleRate',fs, ...
    'NumChannels',size(x,2), ...
    'SamplesPerChannel',size(x,1));
multiCandidate_params_PEF = struct('NumCandidates',5,'MinPeakDistance',5);
[f0_PEF,conf_PEF] = audio.internal.pitch.PEF(xBuff_PEF, ...
    params_PEF, ...
    multiCandidate_params_PEF);

xBuff_NCF = buffer(x,round(0.04*fs),round(0.03*fs),'nodelay');
xBuff_NCF = xBuff_NCF(:,2:end-1);
params_NCF = struct('Method','NCF', ...
    'Range',RANGE, ...
    'WindowLength',round(fs*0.04), ...
    'OverlapLength',round(fs*0.04-HOPLength), ...
    'SampleRate',fs, ...
    'NumChannels',size(x,2), ...
    'SamplesPerChannel',size(x,1));
```

```

multiCandidate_params_NCF = struct('NumCandidates',5,'MinPeakDistance',1);
f0_NCF = audio.internal.pitch.NCF(xBuff_NCF, ...
                                params_NCF, ...
                                multiCandidate_params_NCF);

xBuff_CEP = buffer(x,round(0.04*fs),round(0.03*fs),'nodelay');
xBuff_CEP = xBuff_CEP(:,2:end-1);
params_CEP = struct('Method','CEP', ...
                    'Range',RANGE, ...
                    'WindowLength',round(fs*0.04), ...
                    'OverlapLength',round(fs*0.04-HOPLength), ...
                    'SampleRate',fs, ...
                    'NumChannels',size(x,2), ...
                    'SamplesPerChannel',size(x,1));
multiCandidate_params_CEP = struct('NumCandidates',5,'MinPeakDistance',1);
f0_CEP = audio.internal.pitch.CEP(xBuff_CEP, ...
                                params_CEP, ...
                                multiCandidate_params_CEP);

BackupCandidates = [f0_NCF(:,1),f0_CEP(:,1)];

```

2. Determine Long-Term Median

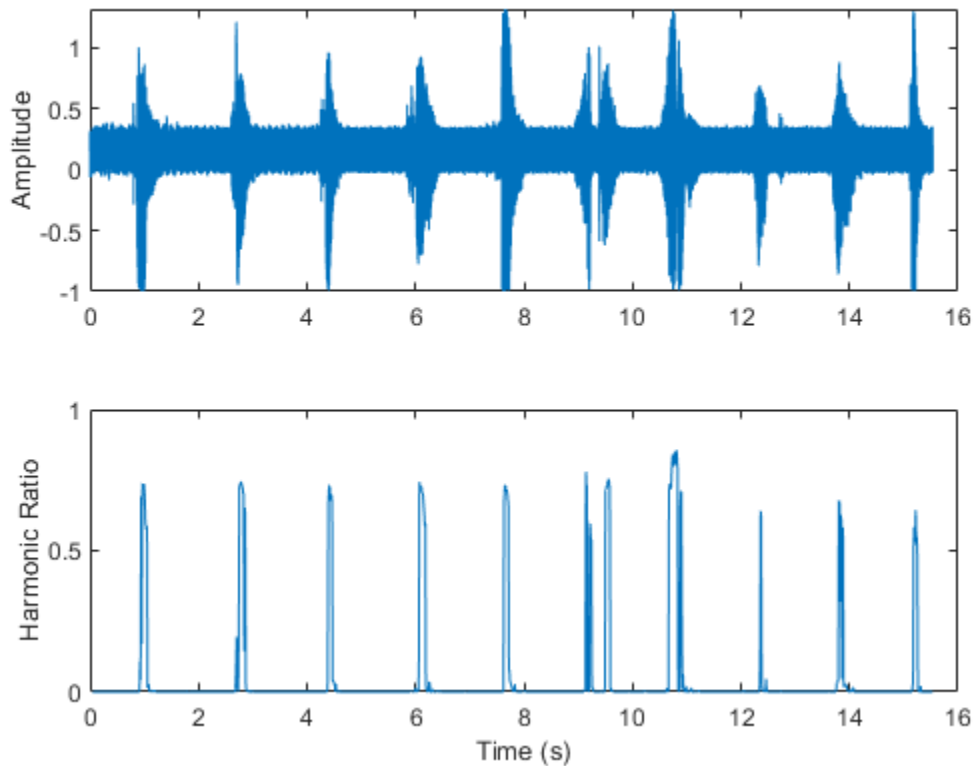
The long-term median of the pitch candidates is used to reduce the number of pitch candidates. To calculate the long-term median pitch, first calculate the harmonic ratio. Pitch estimates are only valid in regions of voiced speech, where the harmonic ratio is high.

```

hr = harmonicRatio(xBuff_PEF,fs, ...
                  'Window',hamming(size(xBuff_NCF,1),'periodic'), ...
                  'OverlapLength',0);

figure(4)
tiledlayout(2,1)
nexttile
plot(t,x)
ylabel('Amplitude')
nexttile
plot(t0,hr)
ylabel('Harmonic Ratio')
xlabel('Time (s)')

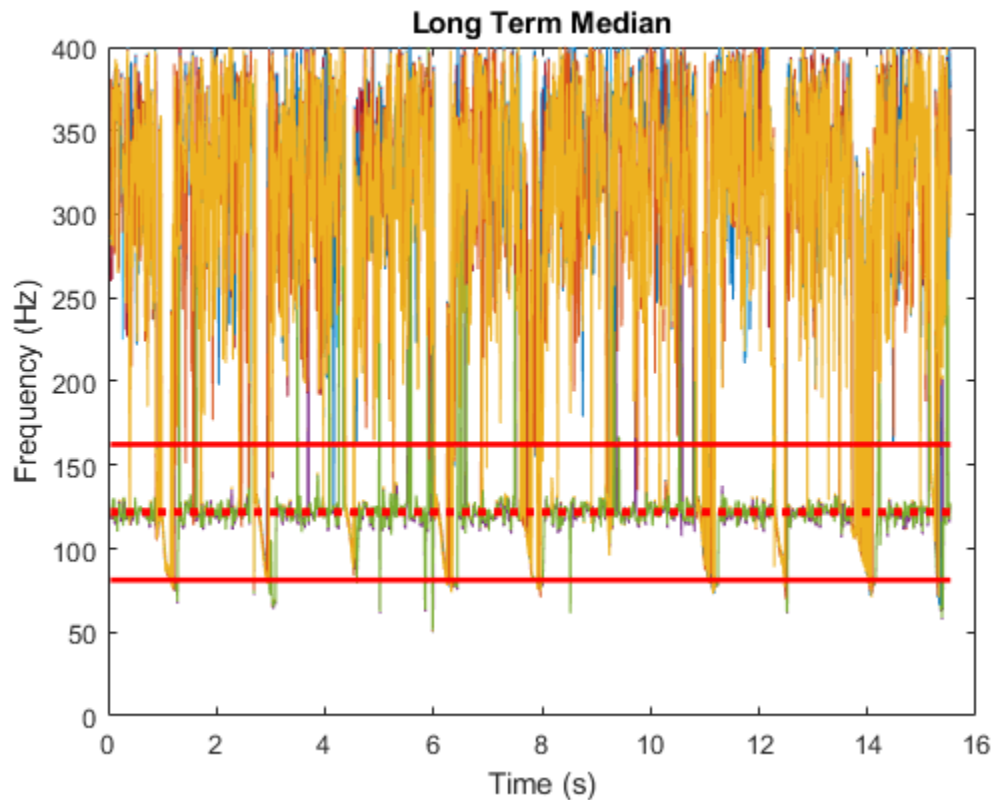
```



Use the harmonic ratio to threshold out regions that do not include voiced speech in the long-term median calculation. After determining the long-term median, calculate lower and upper bounds for pitch candidates. In this example, the lower and upper bounds were determined empirically as $2/3$ and $4/3$ the median pitch. Candidates outside of these bounds are penalized in the following stage.

```
idxToKeep = logical(movmedian(hr>((3/4)*max(hr)),3));
longTermMedian = median([f0_PEF(idxToKeep,1);f0_SRH(idxToKeep,1)]);
lower = max((2/3)*longTermMedian,RANGE(1));
upper = min((4/3)*longTermMedian,RANGE(2));
```

```
figure(5)
tiledlayout(1,1)
nexttile
plot(t0,[f0_PEF,f0_SRH])
hold on
plot(t0,longTermMedian.*ones(size(f0_PEF,1)),'r','LineWidth',3)
plot(t0,upper.*ones(size(f0_PEF,1)),'r','LineWidth',2)
plot(t0,lower.*ones(size(f0_PEF,1)),'r','LineWidth',2)
hold off
xlabel('Time (s)')
ylabel('Frequency (Hz)')
title('Long Term Median')
```

3. Candidate Reduction

By default, candidates returned by the pitch detection algorithm are sorted in descending order of confidence. Decrease the confidence of any primary candidate outside the lower and upper bounds. Decrease the confidence by a factor of 10. Re-sort the candidates for both the PEF and SRH algorithms so they are in descending order of confidence. Concatenate the candidates, keeping only the two most confident candidates from each algorithm.

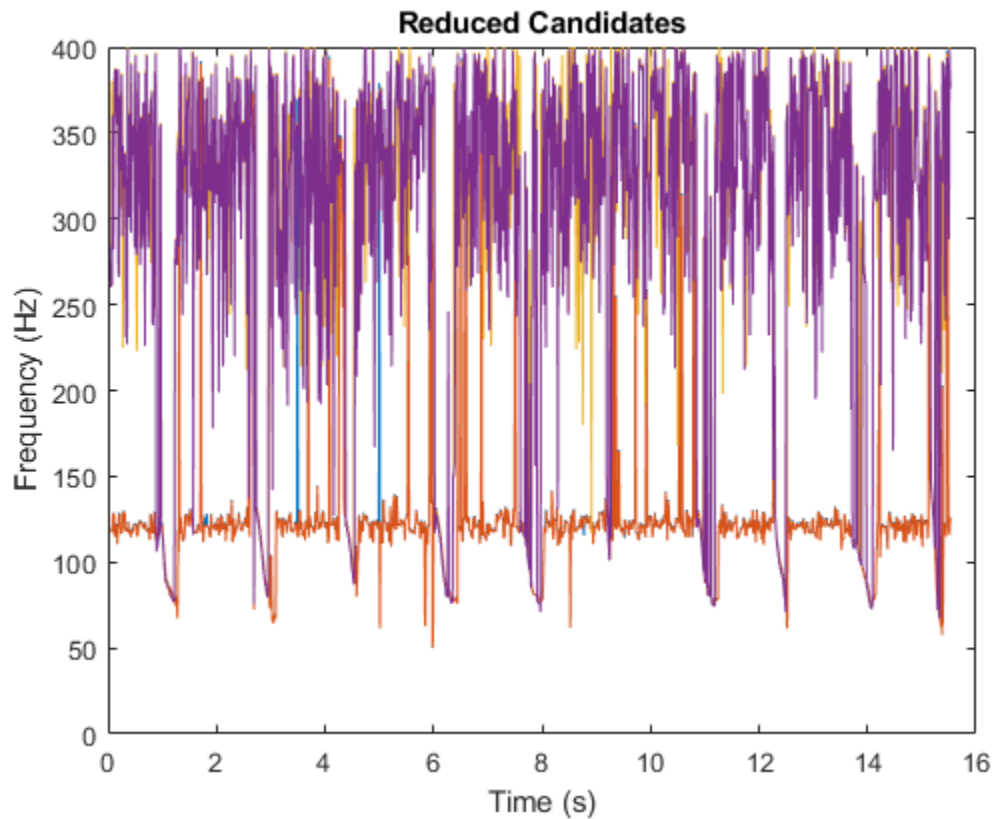
Plot the reduced candidates.

```
invalid = f0_PEF > lower | f0_PEF > upper;
conf_PEF(invalid) = conf_PEF(invalid)/10;
[conf_PEF, order] = sort(conf_PEF, 2, 'descend');
for i = 1:size(f0_PEF, 1)
    f0_PEF(i, :) = f0_PEF(i, order(i, :));
end

invalid = f0_SRH > lower | f0_SRH > upper;
conf_SRH(invalid) = conf_SRH(invalid)/10;
[conf_SRH, order] = sort(conf_SRH, 2, 'descend');
for i = 1:size(f0_SRH, 1)
    f0_SRH(i, :) = f0_SRH(i, order(i, :));
end

candidates = [f0_PEF(:, 1:2), f0_SRH(:, 1:2)];
confidence = [conf_PEF(:, 1:2), conf_SRH(:, 1:2)];
```

```
figure(6)
plot(t0,candidates)
xlabel('Time (s)')
ylabel('Frequency (Hz)')
title('Reduced Candidates')
```



4. Make Distinctive

If two or more candidates are within a given 5 Hz span, set the candidates to their mean and sum their confidence.

```
span = 5;
confidenceFactor = 1;
for r = 1:size(candidates,1)
    for c = 1:size(candidates,2)
        tf = abs(candidates(r,c)-candidates(r,:)) < span;
        candidates(r,c) = mean(candidates(r,tf));
        confidence(r,c) = sum(confidence(r,tf))*confidenceFactor;
    end
end
candidates = max(min(candidates,400),50);
```


5. Forward Iteration of HMM with Octave Smoothing

Now that the candidates have been reduced, you can feed them into an HMM to enforce continuity constraints. Pitch contours are generally continuous for speech signals when analyzed in 10 ms hops. The probability of a pitch moving from one state to another across time is referred to as the *emission*

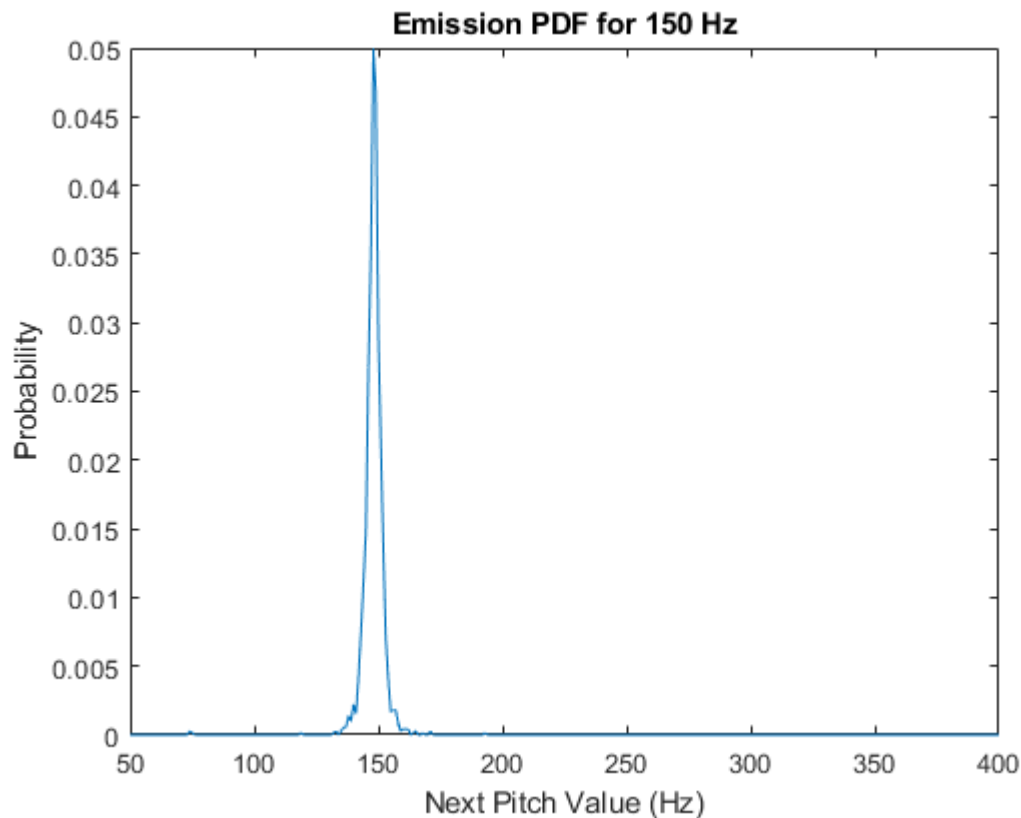
probability. Emission probabilities can be encapsulated into a matrix which describes the probability of going from any pitch value in a set range to any other in a set range. The emission matrix used in this example was created using the Graz database. [1 on page 1-0]

Load the emission matrix and associated range. Plot the probability density function (PDF) of a pitch in 150 Hz state.

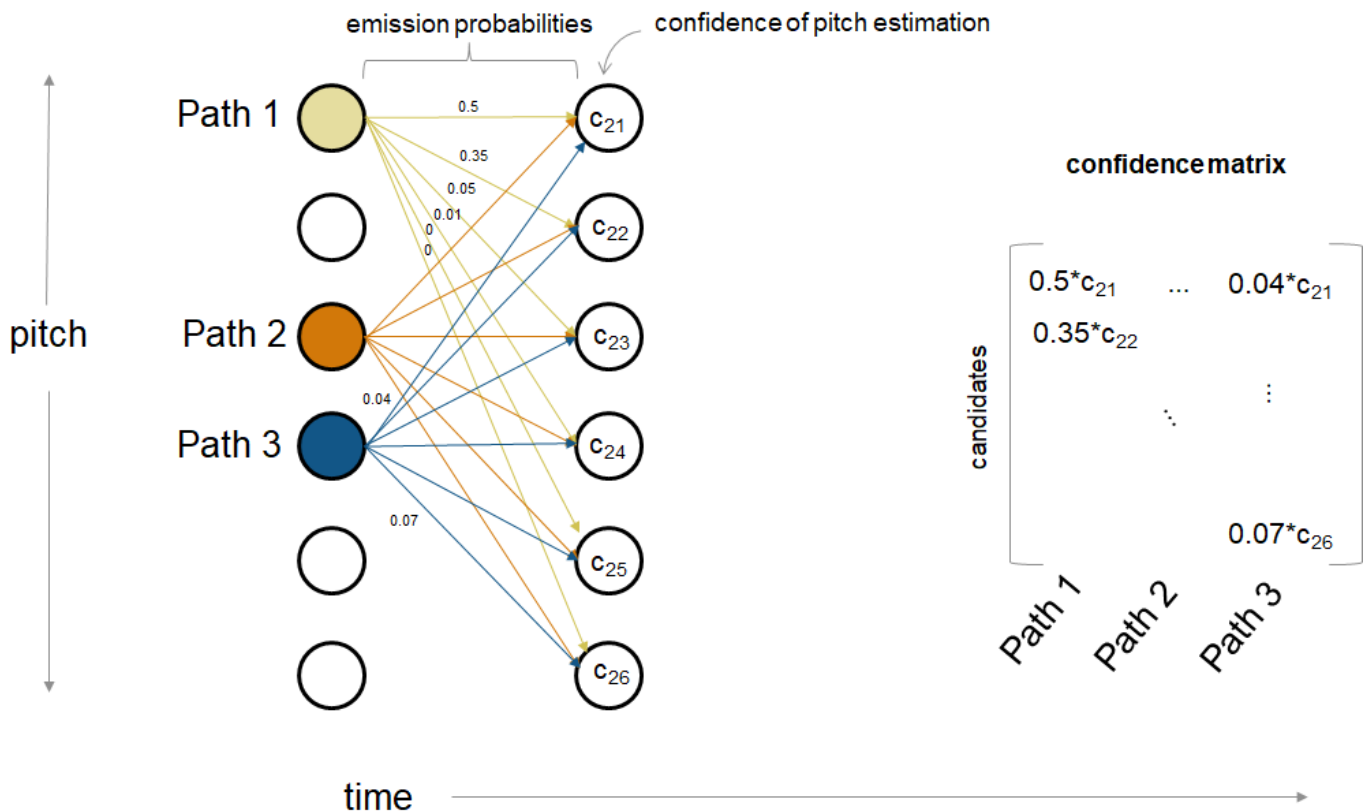
```
load EmissionMatrix.mat emissionMatrix emissionMatrixRange
```

```
currentState = 150  ;
```

```
figure(7)
plot(emissionMatrixRange(1):emissionMatrixRange(2),emissionMatrix(currentState - emissionMatrixRange(1):emissionMatrixRange(2)))
title(sprintf('Emission PDF for %d Hz',currentState))
xlabel('Next Pitch Value (Hz)')
ylabel('Probability')
```



The HMM used in this example combines the emission probabilities, which enforce continuity, and the relative confidence of the pitch. At each hop, the emission probabilities are combined with the relative confidence to create a confidence matrix. A best choice for each path is determined as the max of the confidence matrix. The HMM used in this example also assumes that only one path can be assigned to a given state (an assumption of the Viterbi algorithm).



In addition to the HMM, this example monitors for octave jumps relative to the short-term median of the pitch paths. If an octave jump is detected, then the backup pitch candidates are added as options for the HMM.

```
% Preallocation
numPaths = 4;
numHops = size(candidates,1);
logbook = zeros(numHops,3,numPaths);
suspectHops = zeros(numHops,1);

% Forward iteration with octave-smoothing
for hopNumber = 1:numHops
    nowCandidates = candidates(hopNumber,:);
    nowConfidence = confidence(hopNumber,:);

    % Remove octave jumps
    if hopNumber > 100
        numCandidates = numel(nowCandidates);

        % Weighted short term median
        medianWindowLength = 50;
        aTemp = logbook(max(hopNumber-min(hopNumber,medianWindowLength),1):hopNumber-1,1,:);
        shortTermMedian = median(aTemp(:));
        previousM = mean([longTermMedian,shortTermMedian]);
        lowerTight = max((4/3)*previousM,emissionMatrixRange(1));
        upperTight = min((2/3)*previousM,emissionMatrixRange(2));
        numCandidateOutside = sum([nowCandidates < lowerTight, nowCandidates > upperTight]);
```

```

% If at least 1 candidate is outside the octave centered on the
% short-term median, add the backup pitch candidates that were
% generated by the normalized correlation function and cepstrum
% pitch determination algorithms as potential candidates.
if numCandidateOutside > 0
    % Apply the backup pitch estimators
    nowCandidates = [nowCandidates, BackupCandidates(hopNumber, :)]; %#ok<AGROW>
    nowConfidence = [nowConfidence, repmat(mean(nowConfidence), 1, 2)]; %#ok<AGROW>

    % Make distinctive
    span = 10;
    confidenceFactor = 1.2;
    for r = 1:size(nowCandidates, 1)
        for c = 1:size(nowCandidates, 2)
            tf = abs(nowCandidates(r, c) - nowCandidates(r, :)) < span;
            nowCandidates(r, c) = mean(nowCandidates(r, tf));
            nowConfidence(r, c) = sum(nowConfidence(r, tf)) * confidenceFactor;
        end
    end
end

% Create confidence matrix
confidenceMatrix = zeros(numel(nowCandidates), size(logbook, 3));
for pageIndex = 1:size(logbook, 3)
    if hopNumber ~= 1
        pastPitch = floor(logbook(hopNumber-1, 1, pageIndex)) - emissionMatrixRange(1) + 1;
    else
        pastPitch = nan;
    end
    for candidateNumber = 1:numel(nowCandidates)
        if hopNumber ~= 1
            % Get the current pitch and convert to an index in the range
            currentPitch = floor(nowCandidates(candidateNumber)) - emissionMatrixRange(1) + 1;
            confidenceMatrix(candidateNumber, pageIndex) = ...
                emissionMatrix(currentPitch, pastPitch) * logbook(hopNumber-1, 2, pageIndex) * nowConfidence(candidateNumber);
        else
            confidenceMatrix(candidateNumber, pageIndex) = 1;
        end
    end
end

% Assign an estimate for each path
for pageIndex = 1:size(logbook, 3)
    % Determine most confident transition from past to current pitch
    [~, idx] = max(confidenceMatrix(:));

    % Convert confidence matrix index to pitch and logbook page
    [chosenPitch, pastPitchIdx] = ind2sub([numel(nowCandidates), size(logbook, 3)], idx);

    logbook(hopNumber, :, pageIndex) = ...
        [nowCandidates(chosenPitch), ...
        confidenceMatrix(chosenPitch, pastPitchIdx), ...
        pastPitchIdx];

    % Remove the chosen current pitch from the confidence matrix
    confidenceMatrix(chosenPitch, :) = NaN;
end

```

```
end
% Normalize confidence
logbook(hopNumber,2,:) = logbook(hopNumber,2,:)/sum(logbook(hopNumber,2,:));
end
```

6. Traceback of HMM

Once the forward iteration of the HMM is complete, the final pitch contour is chosen as the most confident path. Walk backward through the log book to determine the pitch contour output by the HMM. Calculate the GPE and plot the new pitch contour and the known contour.

```
numHops = size(logbook,1);
keepLooking = true;
index = numHops + 1;

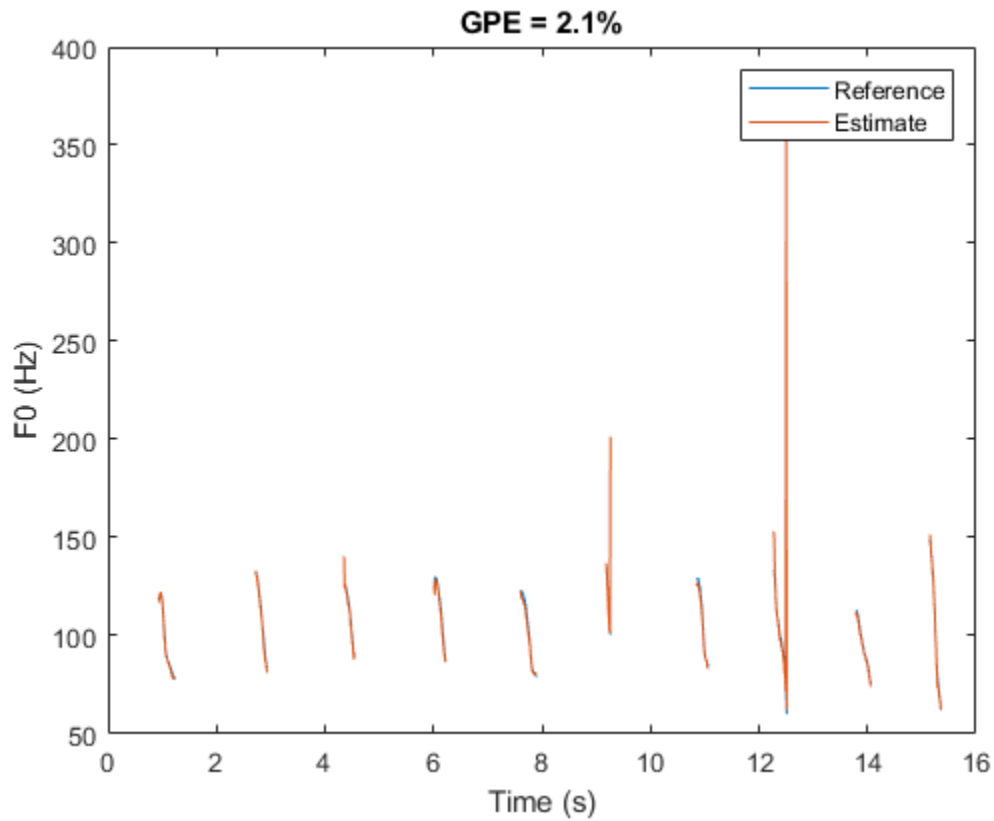
while keepLooking
    index = index - 1;
    if abs(max(logbook(index,2,:))-min(logbook(index,2,:)))~=0
        keepLooking = false;
    end
end

[~,bestPathIdx] = max(logbook(index,2,:));
bestIndices = zeros(numHops,1);
bestIndices(index) = bestPathIdx;

for ii = index:-1:2
    bestIndices(ii-1) = logbook(ii,3,bestIndices(ii));
end

bestIndices(bestIndices==0) = 1;
f0 = zeros(numHops,1);
for ii = (numHops):-1:2
    f0(ii) = logbook(ii,1,bestIndices(ii));
end

f0toPlot = f0;
f0toPlot(~isVoiced) = NaN;
GPE = mean( abs(f0toPlot(isVoiced) - truePitch(isVoiced)) > truePitch(isVoiced).*p).*100;
figure(8)
plot(t0,[truePitch,f0toPlot])
legend('Reference','Estimate')
ylabel('F0 (Hz)')
xlabel('Time (s)')
title(sprintf('GPE = %0.1f%%',GPE))
```

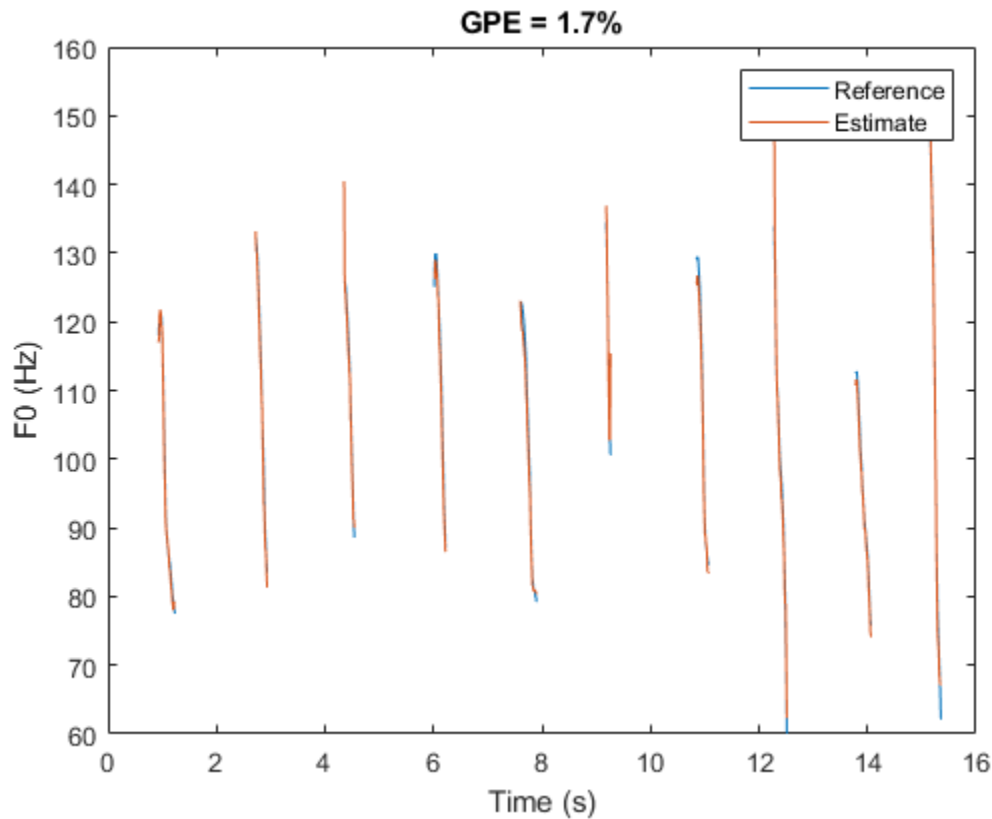


7. Moving Median Filter

As a final post-processing step, apply a moving median filter with a window length of three hops. Calculate the final GPE and plot the final pitch contour and the known contour.

```
f0 = movmedian(f0,3);
f0(~isVoiced) = NaN;

GPE = mean(abs(f0(isVoiced) - truePitch(isVoiced)) > truePitch(isVoiced).*p).*100;
figure(9)
plot(t0,[truePitch,f0])
legend('Reference','Estimate')
ylabel('F0 (Hz)')
xlabel('Time (s)')
title(sprintf('GPE = %0.1f%%',GPE))
```



Performance Evaluation

The `HelperPitchTracker` function uses an HMM to apply continuity constraints to pitch contours. The emission matrix of the HMM can be set directly. It is best to train the emission matrix on sound sources similar to the ones you want to track.

This example uses the Pitch Tracking Database from Graz University of Technology (PTDB-TUG) [4] on page 1-0 . The data set consists of 20 English native speakers reading 2342 phonetically rich sentences from the TIMIT corpus. Download and extract the data set. Depending on your system, downloading and extracting the data set can take approximately 1.5 hours.

```
url = 'https://www2.spsc.tugraz.at/databases/PTDB-TUG/SPEECH_DATA_ZIPPED.zip';
downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder,'PTDB-TUG');

if ~exist(datasetFolder,'dir')
    disp('Downloading PTDB-TUG (3.9 G) ...')
    unzip(url,datasetFolder)
end
```

Create an audio datastore that points to the microphone recordings in the database. Set the label associated with each file to the location of the associated known pitch file. The dataset contains recordings of 10 female and 10 male speakers. Use `subset` to isolate the 10th female and male speakers. Train an emission matrix based on the reference pitch contours for both male and female speakers 1 through 9.


```

ads = audioDatastore([fullfile(datasetFolder,"SPEECH DATA","FEMALE","MIC"),fullfile(datasetFolder,
    'IncludeSubfolders',true, ...
    'FileExtensions','.wav')];
wavFileNames = ads.Files;
ads.Labels = replace(wavFileNames,{ 'MIC','mic','wav'},{ 'REF','ref','f0'});

idxToRemove = contains(ads.Files,{ 'F10','M10'});
ads1 = subset(ads,idxToRemove);
ads9 = subset(ads,~idxToRemove);

```

Shuffle the audio datastores.

```

ads1 = shuffle(ads1);
ads9 = shuffle(ads9);

```

The emission matrix describes the probability of going from one pitch state to another. In the following step, you create an emission matrix based on speakers 1 through 9 for both male and female. The database stores reference pitch values, short-term energy, and additional information in the text files with files extension f0. The `getReferencePitch` function reads in the pitch values if the short-term energy is above a threshold. The threshold was determined empirically in listening tests. The `HelperUpdateEmissionMatrix` creates a 2-dimensional histogram based on the current pitch state and the next pitch state. After the histogram is created, it is normalized to create an emission matrix.

```

emissionMatrixRange = [50,400];
emissionMatrix = [];

for i = 1:numel(ads9.Files)
    x = getReferencePitch(ads9.Labels{i});
    emissionMatrix = HelperUpdateEmissionMatrix(x,emissionMatrixRange,emissionMatrix);
end
emissionMatrix = emissionMatrix + sqrt(eps);
emissionMatrix = emissionMatrix./norm(emissionMatrix);

```

Define different types of background noise: white, ambiance, engine, jet, and street. Resample them to 16 kHz to help speed up testing the database.

Define the SNR to test, in dB, as 10, 5, 0, -5, and -10.

```

noiseType = { 'white','ambiance','engine','jet','street'};
numNoiseToTest = numel(noiseType);

desiredFs = 16e3;

whiteNoiseMaker = dsp.ColoredNoise('Color','white','SamplesPerFrame',40000,'RandomStream','mt19937');
noise{1} = whiteNoiseMaker();
[ambiance,ambianceFs] = audioread('Ambiance-16-44p1-mono-12secs.wav');
noise{2} = resample(ambiance,desiredFs,ambianceFs);
[engine,engineFs] = audioread('Engine-16-44p1-stereo-20sec.wav');
noise{3} = resample(engine,desiredFs,engineFs);
[jet,jetFs] = audioread('JetAirplane-16-11p025-mono-16secs.wav');
noise{4} = resample(jet,desiredFs,jetFs);
[street,streetFs] = audioread('MainStreetOne-16-16-mono-12secs.wav');
noise{5} = resample(street,desiredFs,streetFs);

snrToTest = [10,5,0,-5,-10];
numSNRtoTest = numel(snrToTest);

```

Run the pitch detection algorithm for each SNR and noise type for each file. Calculate the average GPE across speech files. This example compares performance with the popular pitch tracking algorithm: Sawtooth Waveform Inspired Pitch Estimator (SWIPE). A MATLAB® implementation of the algorithm can be found at [5 on page 1-0]. To run this example without comparing to other algorithms, set `compare` to `false`. The following comparison takes around 15 minutes.

```
compare =  ;
numFilesToTest = 20;
p = 0.1;
GPE_pitchTracker = zeros(numSNRtoTest,numNoiseToTest,numFilesToTest);
if compare
    GPE_swipe = GPE_pitchTracker;
end
for i = 1:numFilesToTest
    [cleanSpeech,info] = read(ads1);
    cleanSpeech = resample(cleanSpeech,desiredFs,info.SampleRate);

    truePitch = getReferencePitch(info.Label{:});
    isVoiced = truePitch~=0;
    truePitchInVoicedRegions = truePitch(isVoiced);

    for j = 1:numSNRtoTest
        for k = 1:numNoiseToTest
            noisySpeech = mixSNR(cleanSpeech,noise{k},snrToTest(j));
            f0 = HelperPitchTracker(noisySpeech,desiredFs,'EmissionMatrix',emissionMatrix,'EmissionMatrix',emissionMatrix);
            f0 = [0;f0]; % manual alignment with database.
            GPE_pitchTracker(j,k,i) = mean(abs(f0(isVoiced) - truePitchInVoicedRegions) > truePitchInVoicedRegions);

            if compare
                f0 = swipep(noisySpeech,desiredFs,[50,400],0.01);
                f0 = f0(3:end); % manual alignment with database.
                GPE_swipe(j,k,i) = mean(abs(f0(isVoiced) - truePitchInVoicedRegions) > truePitchInVoicedRegions);
            end
        end
    end
end
GPE_pitchTracker = mean(GPE_pitchTracker,3);

if compare
    GPE_swipe = mean(GPE_swipe,3);
end
```

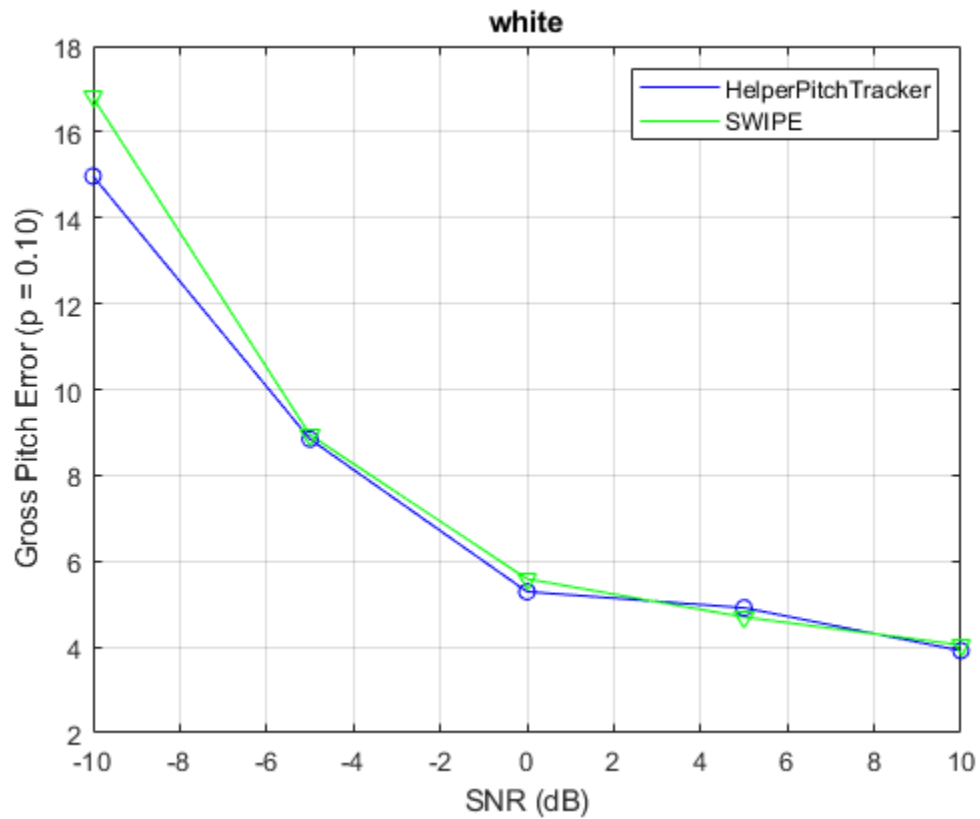
Plot the gross pitch error for each noise type.

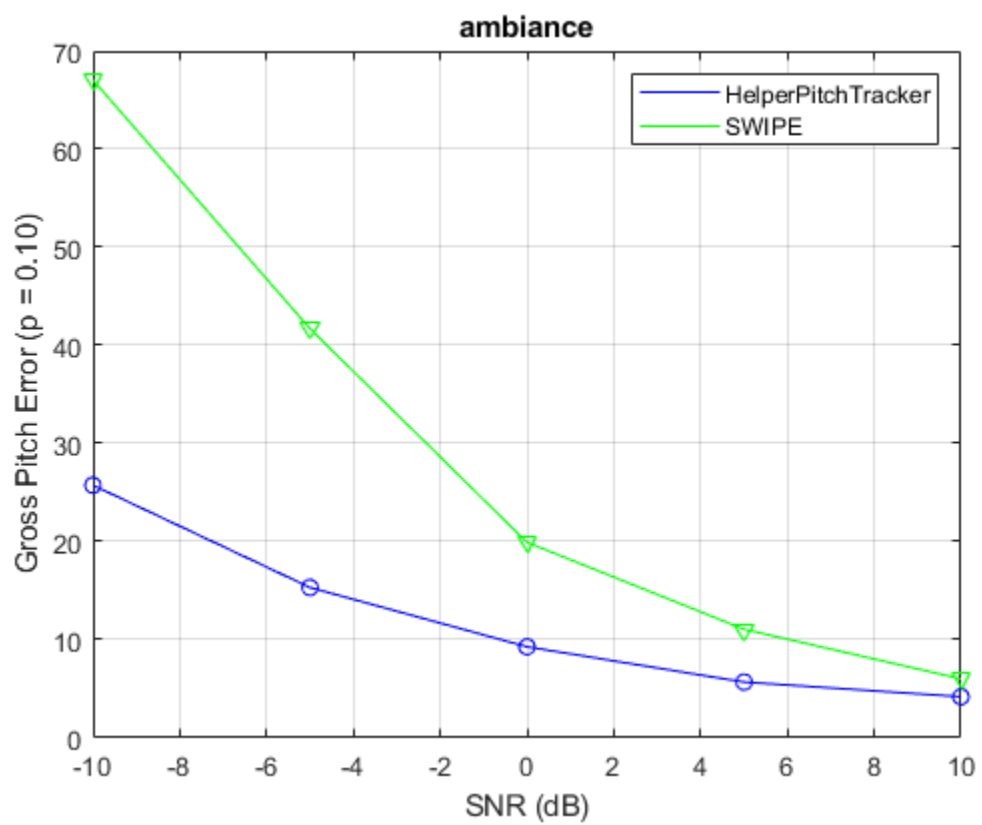
```
for ii = 1:numel(noise)
    figure(9+ii)
    plot(snrToTest,GPE_pitchTracker(:,ii),'b')
    hold on
    if compare
        plot(snrToTest,GPE_swipe(:,ii),'g')
    end
    plot(snrToTest,GPE_pitchTracker(:,ii),'bo')
    if compare
        plot(snrToTest,GPE_swipe(:,ii),'gv')
    end
    title(noiseType(ii))
    xlabel('SNR (dB)')
```

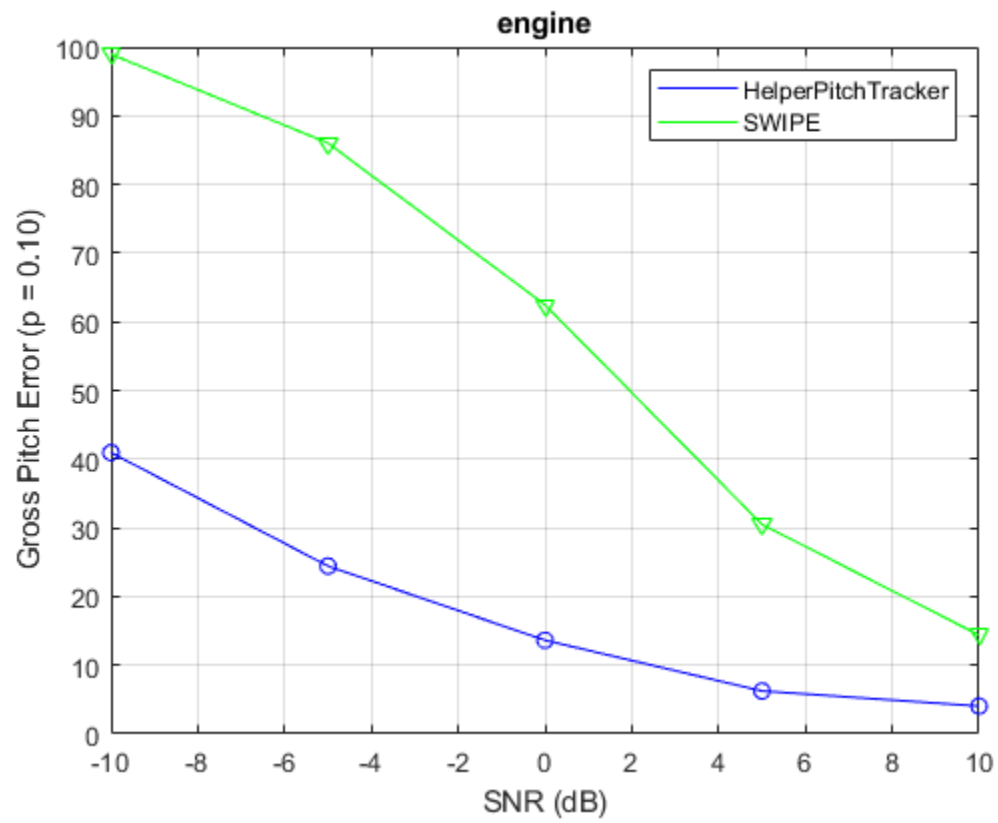
```

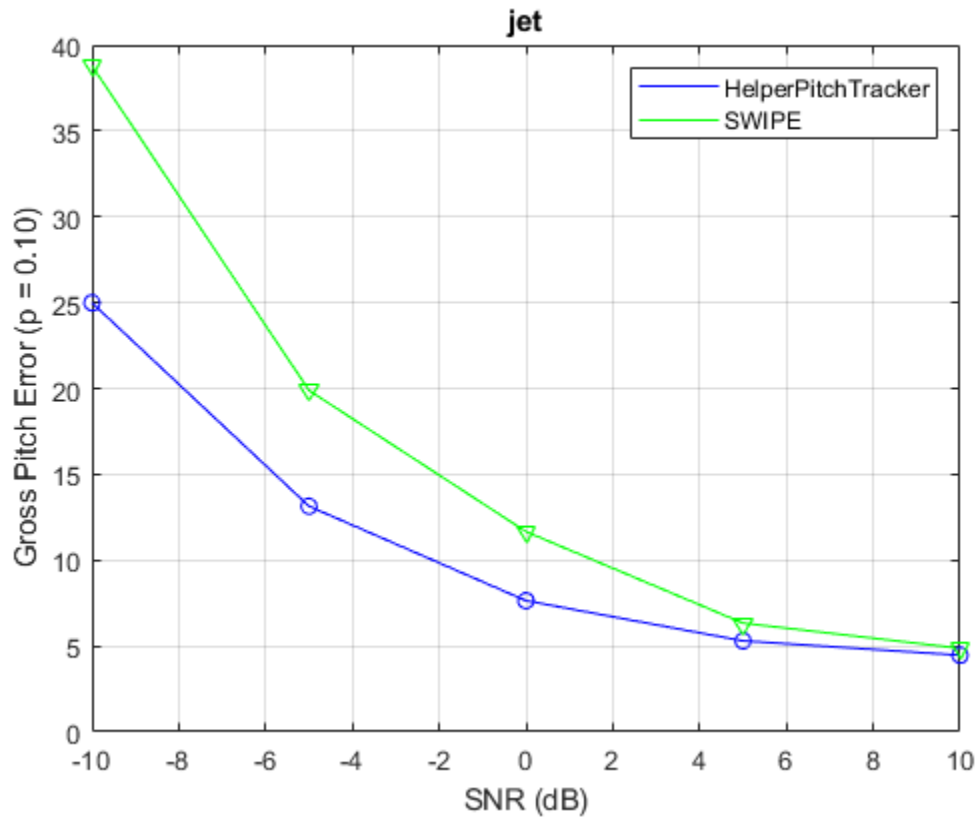
ylabel(sprintf('Gross Pitch Error (p = %0.2f)',p))
if compare
    legend('HelperPitchTracker','SWIPE')
else
    legend('HelperPitchTracker')
end
grid on
hold off
end

```









Conclusion

You can use `HelperPitchTracker` as a baseline for evaluating GPE performance of your pitch tracking system, or adapt this example to your application.

References

- [1] G. Pirker, M. Wohlmayr, S. Petrik, and F. Pernkopf, "A Pitch Tracking Corpus with Evaluation on Multipitch Tracking Scenario", *Interspeech*, pp. 1509-1512, 2011.
- [2] 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.
- [3] 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.
- [4] Signal Processing and Speech Communication Laboratory. Accessed September 26, 2018. <https://www.spsc.tugraz.at/databases-and-tools/ptdb-tug-pitch-tracking-database-from-graz-university-of-technology.html>.
- [5] "Arturo Camacho." Accessed September 26, 2018. <https://www.cise.ufl.edu/~acamacho/english/>.
- [6] "Fxpefac." Description of Fxpefac. Accessed September 26, 2018. <http://www.ee.ic.ac.uk/hp/staff/dmb/voicebox/voicebox.html>.

Voice Activity Detection in Noise Using Deep Learning

This example shows how to detect regions of speech in a low signal-to-noise environment using deep learning. The example uses the Speech Commands Dataset to train a Bidirectional Long Short-Term Memory (BiLSTM) network to detect voice activity.

Introduction

Voice activity detection is an essential component of many audio systems, such as automatic speech recognition and speaker recognition. Voice activity detection can be especially challenging in low signal-to-noise (SNR) situations, where speech is obstructed by noise.

This example uses long short-term memory (LSTM) networks, which are a type of recurrent neural network (RNN) well-suited to study sequence and time-series data. An LSTM network can learn long-term dependencies between time steps of a sequence. An LSTM layer (`lstmLayer` (Deep Learning Toolbox)) can look at the time sequence in the forward direction, while a bidirectional LSTM layer (`biLstmLayer` (Deep Learning Toolbox)) can look at the time sequence in both forward and backward directions. This example uses a bidirectional LSTM layer.

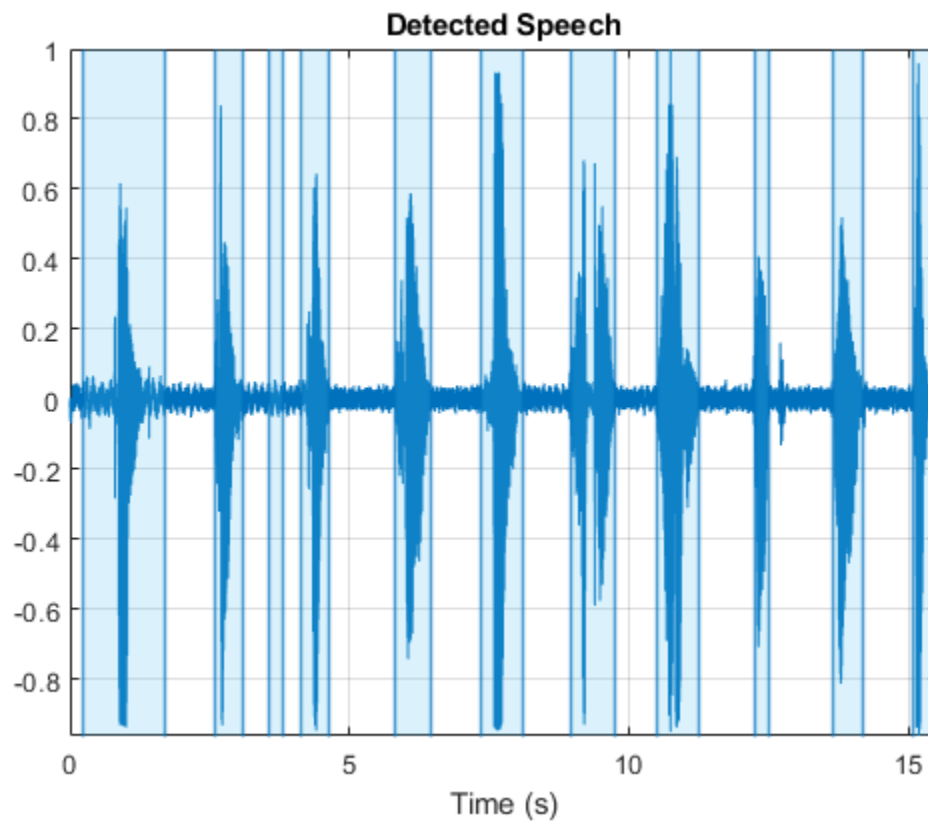
This example trains a voice activity detection bidirectional LSTM network with feature sequences of spectral characteristics and a harmonic ratio metric.

In high SNR scenarios, traditional speech detection algorithms perform adequately. Read in an audio file that consists of words spoken with pauses between. Resample the audio to 16 kHz. Listen to the audio.

```
fs = 16e3;
[speech,fileFs] = audioread('Counting-16-44p1-mono-15secs.wav');
speech = resample(speech,fs,fileFs);
speech = speech/max(abs(speech));
sound(speech,fs)
```

Use the `detectSpeech` function to locate regions of speech. The `detectSpeech` function correctly identifies all regions of speech.

```
win = hamming(50e-3 * fs,'periodic');
detectSpeech(speech,fs,'Window',win)
```



Corrupt the audio signal with washing machine noise at a -20 dB SNR. Listen to the corrupted audio.

```
[noise,fileFs] = audioread('WashingMachine-16-8-mono-200secs.mp3');  
noise = resample(noise,fs,fileFs);
```

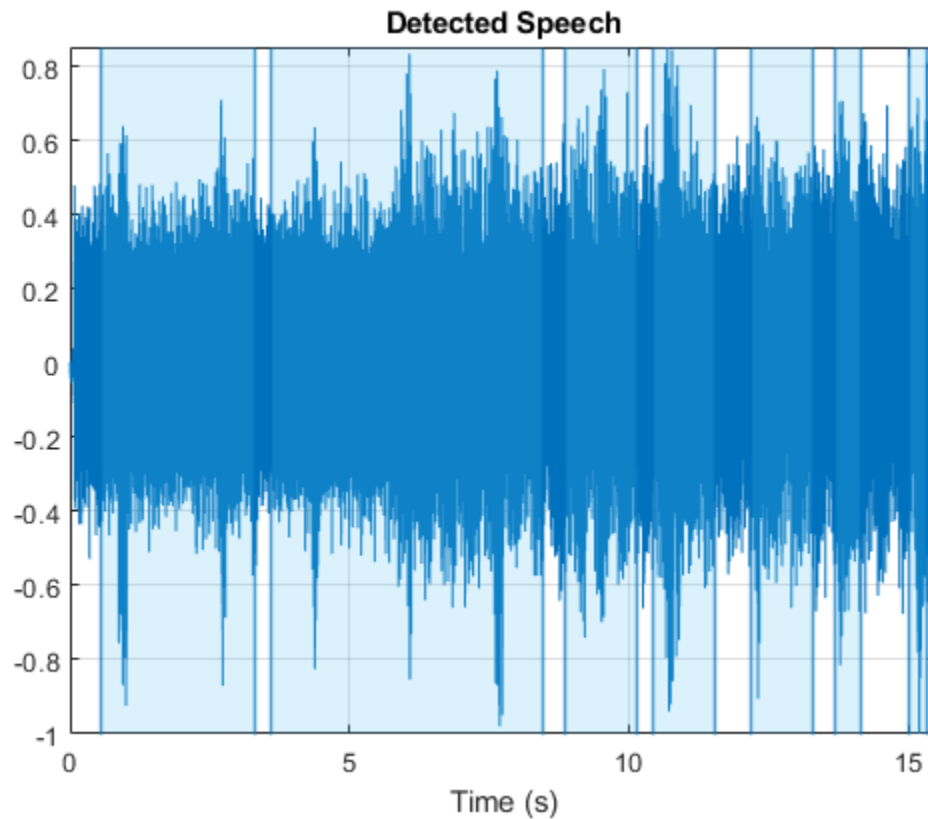
```
SNR = -20;  
noiseGain = 10^(-SNR/20) * norm(speech) / norm(noise);
```

```
noisySpeech = speech + noiseGain*noise(1:numel(speech));  
noisySpeech = noisySpeech./max(abs(noisySpeech));
```

```
sound(noisySpeech,fs)
```

Call `detectSpeech` on the noisy audio signal. The function fails to detect the speech regions given the very low SNR.

```
detectSpeech(noisySpeech,fs,'Window',win)
```

Load a pretrained network and a configured `audioFeatureExtractor` object. The network was trained to detect speech in a low SNR environments given features output from the `audioFeatureExtractor` object.

```
load('Audio_VoiceActivityDetectionExample.mat','speechDetectNet','afe')
```

```
speechDetectNet
```

```
speechDetectNet =  
  SeriesNetwork with properties:  
  
    Layers: [6×1 nnet.cnn.layer.Layer]  
  InputNames: {'sequenceinput'}  
 OutputNames: {'classoutput'}
```

```
afe
```

```
afe =  
  audioFeatureExtractor with properties:  
  
  Properties  
      Window: [256×1 double]  
  OverlapLength: 128  
    SampleRate: 16000  
      FFTLength: []  
SpectralDescriptorInput: 'linearSpectrum'
```

Enabled Features

```
spectralCentroid, spectralCrest, spectralEntropy, spectralFlux, spectralKurtosis, spectralR...  
spectralSkewness, spectralSlope, harmonicRatio
```

Disabled Features

```
linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta  
mfccDeltaDelta, gtcc, gtccDelta, gtccDeltaDelta, spectralDecrease, spectralFlatness  
spectralSpread, pitch
```

To extract a feature, set the corresponding property to true.

For example, `obj.mfcc = true`, adds mfcc to the list of enabled features.

Extract features from the speech data and then normalize them. Orient the features so that time is across columns.

```
features = extract(afe,noisySpeech);  
features = (features - mean(features,1)) ./ std(features,[],1);  
features = features';
```

Pass the features through the speech detection network to classify each feature vector as belonging to a frame of speech or not.

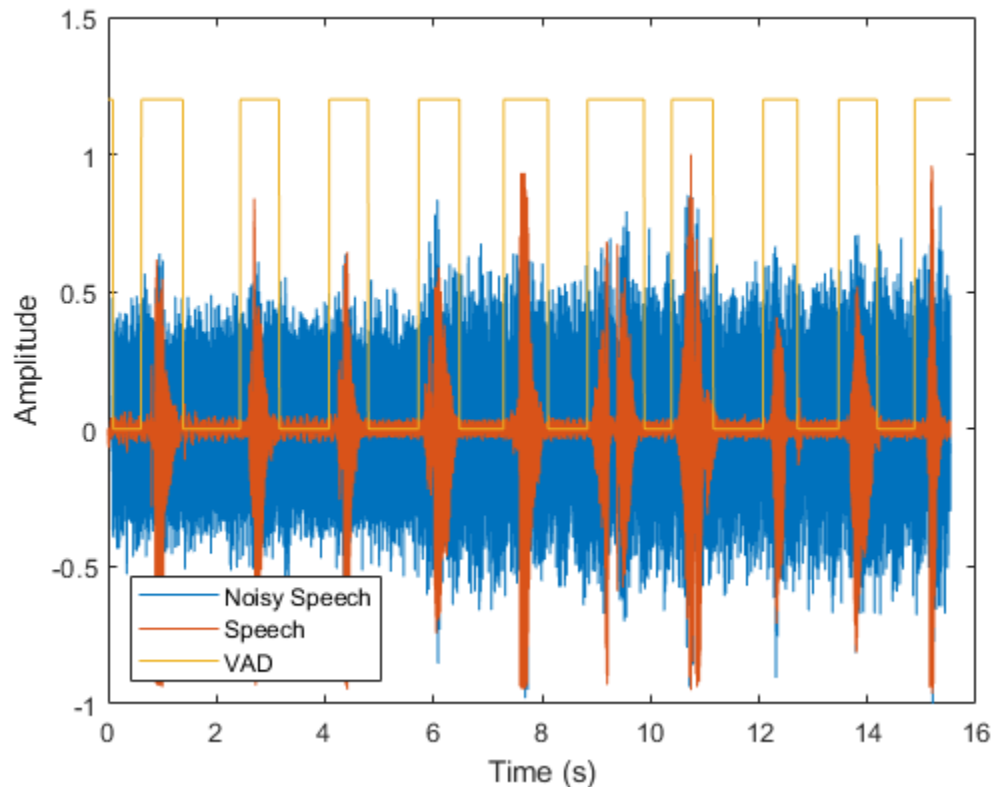
```
decisionsCategorical = classify(speechDetectNet,features);
```

Each decision corresponds to an analysis window analyzed by the `audioFeatureExtractor`.

Replicate the decisions so that they are in one-to-one correspondence with the audio samples. Plot the speech, the noisy speech, and the VAD decisions.

```
decisionsWindow = 1.2*(double(decisionsCategorical)-1);  
decisionsSample = [repelem(decisionsWindow(1),numel(afe.Window)), ...  
                   repelem(decisionsWindow(2:end),numel(afe.Window)-afe.OverlapLength)];
```

```
t = (0:numel(decisionsSample)-1)/afe.SampleRate;  
plot(t,noisySpeech(1:numel(t)), ...  
      t,speech(1:numel(t)), ...  
      t,decisionsSample);  
xlabel('Time (s)')  
ylabel('Amplitude')  
legend('Noisy Speech','Speech','VAD','Location','southwest')
```



You can also use the trained VAD network in a streaming context. To simulate a streaming environment, first save the speech and noise signals as WAV files. To simulate streaming input, you will read frames from the files and mix them at a desired SNR.

```
audiowrite('Speech.wav',speech,fs)
audiowrite('Noise.wav',noise,fs)
```

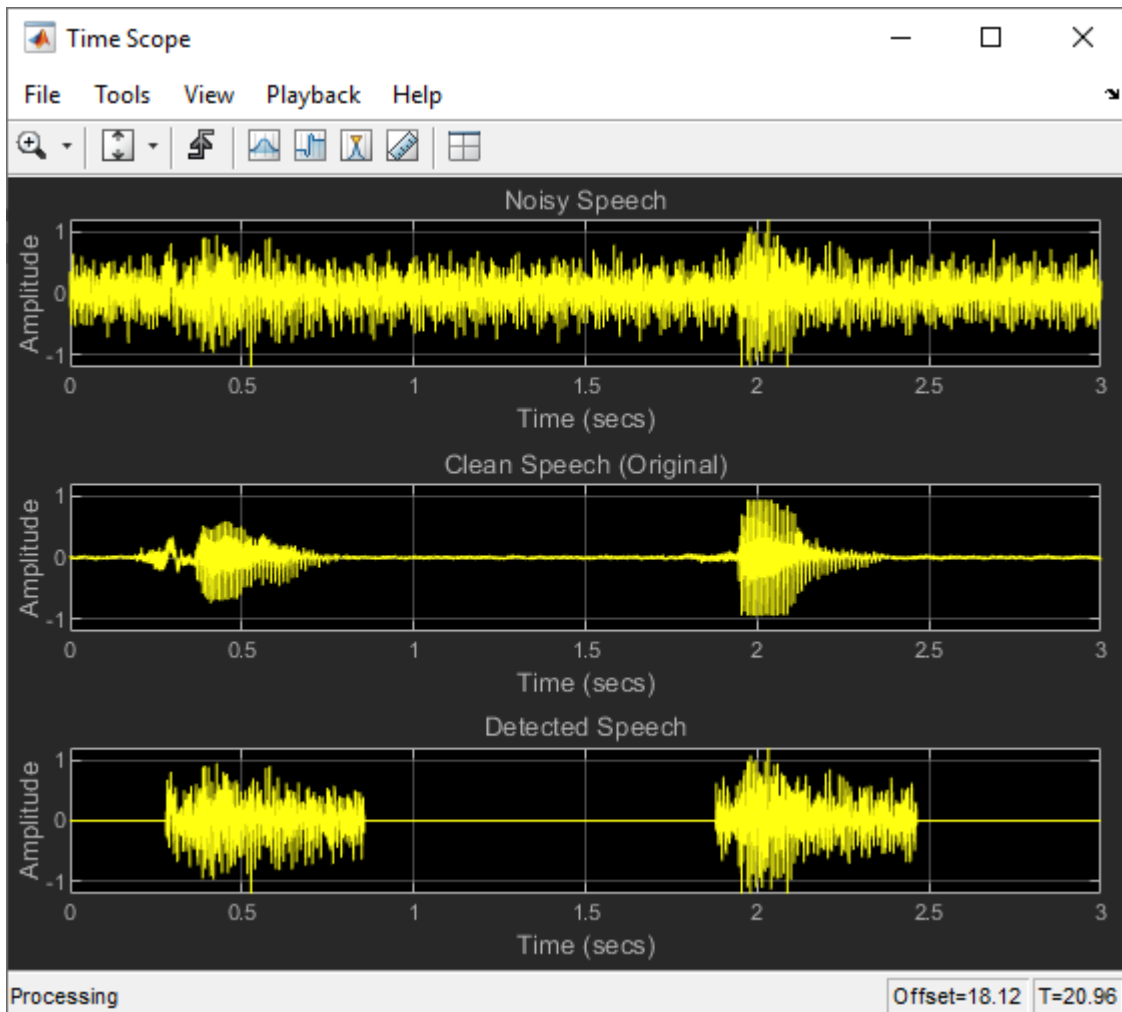
To apply the VAD network to streaming audio, you have to trade off between delay and accuracy. Define parameters for the streaming voice activity detection in noise demonstration. You can set the duration of the test, the sequence length fed into the network, the sequence hop length, and the SNR to test. Generally, increasing the sequence length increases the accuracy but also increases the lag. You can also choose the signal output to your device as the original signal or the noisy signal.

```
testDuration = 20 ;
sequenceLength = 400 ;
sequenceHop = 20 ;
SNR = -20 ;
noiseGain = 10^(-SNR/20) * norm(speech) / norm(noise);

signalToListenTo = "noisy" ;
```

Call the streaming demo helper function to observe the performance of the VAD network on streaming audio. The parameters you set using the live controls do not interrupt the streaming example. After the streaming demo is complete, you can modify parameters of the demonstration, then run the streaming demo again. You can find the code for the streaming demo in the Supporting Functions on page 1-0 .

```
helperStreamingDemo(speechDetectNet,afe, ...
    'Speech.wav','Noise.wav', ...
    testDuration,sequenceLength,sequenceHop,signalToListenTo,noiseGain);
```



The remainder of the example walks through training and evaluating the VAD network.

Train and Evaluate VAD Network

Training:

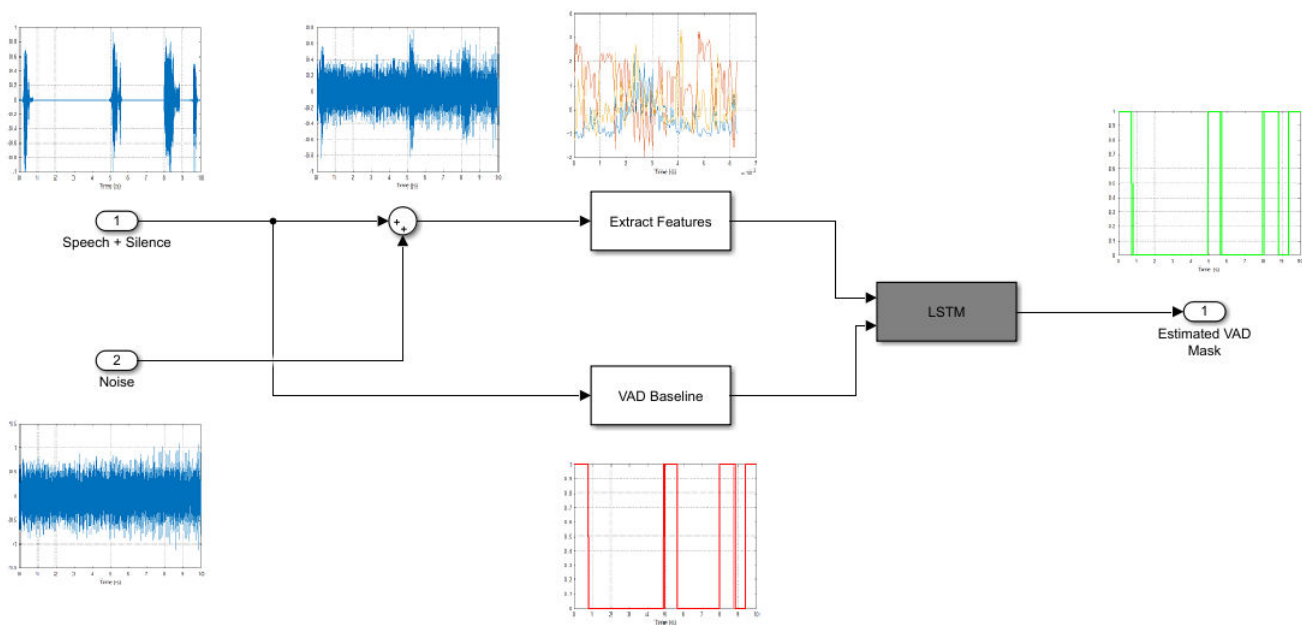
- 1 Create an `audioDatastore` that points to the audio speech files used to train the LSTM network.
- 2 Create a training signal consisting of speech segments separated by segments of silence of varying durations.

- 3 Corrupt the speech-plus-silence signal with washing machine noise (SNR = -10 dB).
- 4 Extract feature sequences consisting of spectral characteristics and harmonic ratio from the noisy signal.
- 5 Train the LSTM network using the feature sequences to identify regions of voice activity.

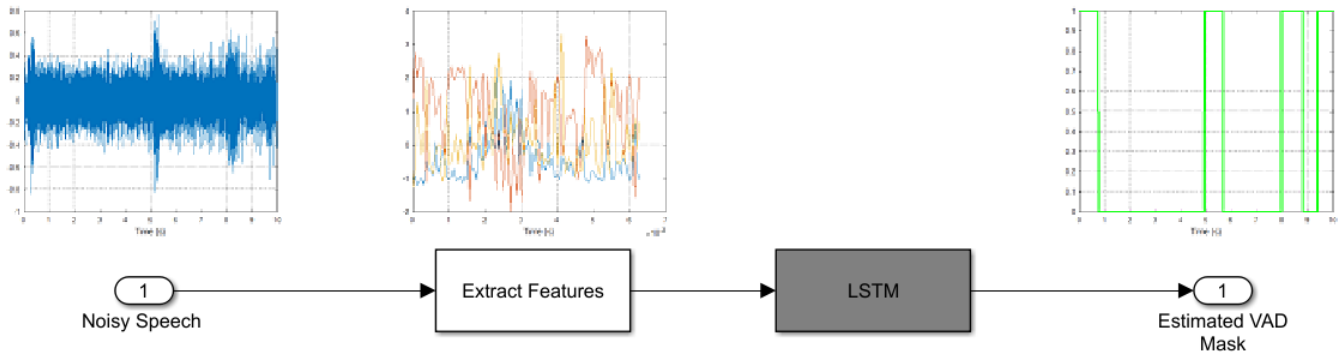
Prediction:

- 1 Create an `audioDatastore` of speech files used to test the trained network, and create a test signal consisting of speech separated by segments of silence.
- 2 Corrupt the test signal with washing machine noise (SNR = -10 dB).
- 3 Extract feature sequences from the noisy test signal.
- 4 Identify regions of voice activity by passing the test features through the trained network.
- 5 Compare the network's accuracy to the voice activity baseline from the signal-plus-silence test signal.

Here is a sketch of the training process.



Here is a sketch of the prediction process. You use the trained network to make predictions.



Load Speech Commands Data Set

Download and extract the Google Speech Commands Dataset [1] on page 1-0 .

```
url = 'https://ssd.mathworks.com/supportfiles/audio/google_speech.zip';
```

```
downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder, 'google_speech');
```

```
if ~exist(datasetFolder, 'dir')
    disp('Downloading Google speech commands data set (1.9 GB)...')
    unzip(url, downloadFolder)
end
```

```
Downloading Google speech commands data set (1.9 GB)...
```

Create an `audioDatastore` that points to the training data set.

```
adsTrain = audioDatastore(fullfile(datasetFolder, 'train'), "Includesubfolders", true);
```

Create an `audioDatastore` that points to the validation data set.

```
adsValidation = audioDatastore(fullfile(datasetFolder, 'validation'), "Includesubfolders", true);
```

Create Speech-Plus-Silence Training Signal

Read the contents of an audio file using `read`. Get the sample rate from the `adsInfo` struct.

```
[data, adsInfo] = read(adsTrain);
Fs = adsInfo.SampleRate;
```

Listen to the audio signal using the `sound` command.

```
sound(data, Fs)
```

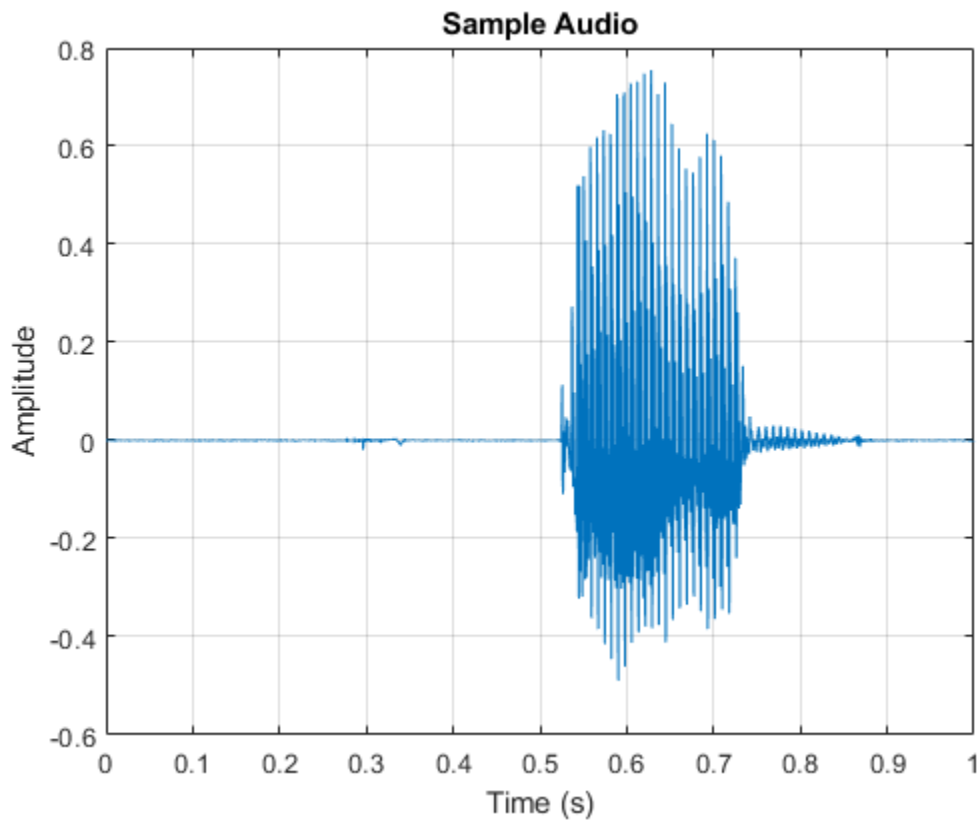
Plot the audio signal.

```
timeVector = (1/Fs) * (0:numel(data)-1);
plot(timeVector, data)
```

```

ylabel("Amplitude")
xlabel("Time (s)")
title("Sample Audio")
grid on

```



The signal has non-speech portions (silence, background noise, etc) that do not contain useful speech information. This example removes silence using the `detectSpeech` function.

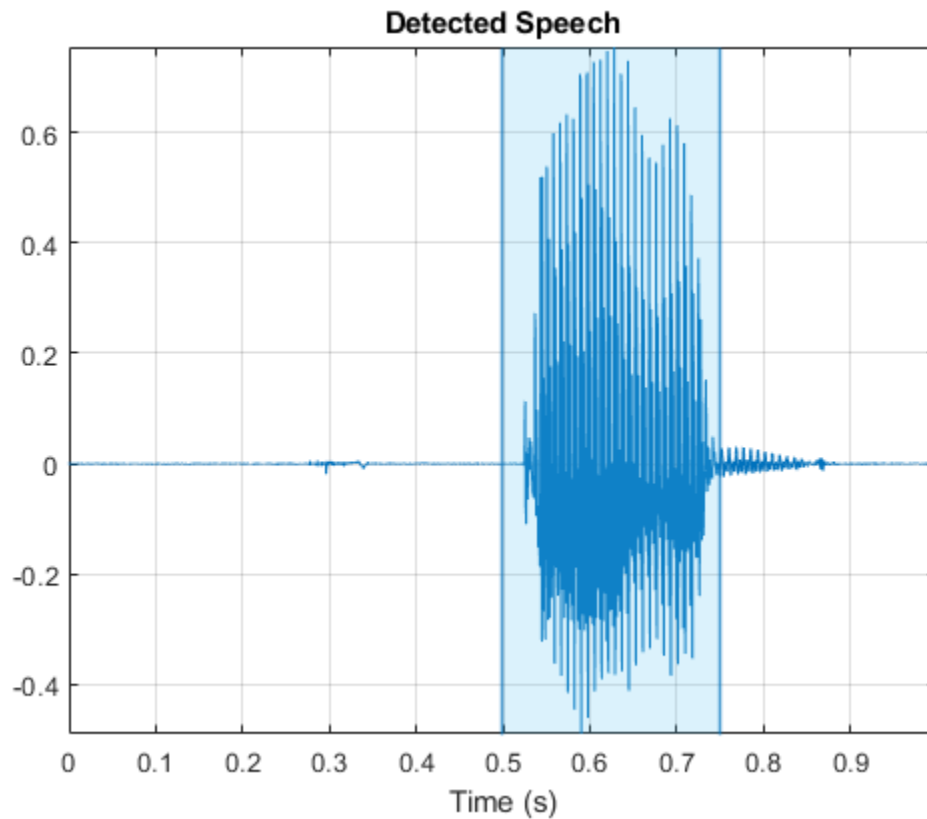
Extract the useful portion of data. Define a 50 ms periodic Hamming window for analysis. Call `detectSpeech` with no output arguments to plot the detected speech regions. Call `detectSpeech` again to return the indices of the detected speech. Isolate the detected speech regions and then use the `sound` command to listen to the audio.

```

win = hamming(50e-3 * Fs, 'periodic');

detectSpeech(data, Fs, 'Window', win);

```



```
speechIndices = detectSpeech(data,Fs,'Window',win);
sound(data(speechIndices(1,1):speechIndices(1,2)),Fs)
```

The `detectSpeech` function returns indices that tightly surround the detected speech region. It was determined empirically that, for this example, extending the indices of the detected speech by five frames on either side increased the final model's performance. Extend the speech indices by five frames and then listen to the speech.

```
speechIndices(1,1) = max(speechIndices(1,1) - 5*numel(win),1);
speechIndices(1,2) = min(speechIndices(1,2) + 5*numel(win),numel(data));

sound(data(speechIndices(1,1):speechIndices(1,2)),Fs)
```

Reset the training datastore and shuffle the order of files in the datastores.

```
reset(adsTrain)
adsTrain = shuffle(adsTrain);
adsValidation = shuffle(adsValidation);
```

The `detectSpeech` function calculates statistics-based thresholds to determine the speech regions. You can skip the threshold calculation and speed up the `detectSpeech` function by specifying the thresholds directly. To determine thresholds for a data set, call `detectSpeech` on a sampling of files and get the thresholds it calculates. Take the mean of the thresholds.

```
TM = [];
for index1 = 1:500
```



```

    data = read(adsTrain);
    [~,T] = detectSpeech(data,Fs,'Window',win);
    TM = [TM;T];
end

```

```
T = mean(TM);
```

```
reset(adsTrain)
```

Create a 1000-second training signal by combining multiple speech files from the training data set. Use `detectSpeech` to remove unwanted portions of each file. Insert a random period of silence between speech segments.

Preallocate the training signal.

```

duration = 2000*Fs;
audioTraining = zeros(duration,1);

```

Preallocate the voice activity training mask. Values of 1 in the mask correspond to samples located in areas with voice activity. Values of 0 correspond to areas with no voice activity.

```
maskTraining = zeros(duration,1);
```

Specify a maximum silence segment duration of 2 seconds.

```
maxSilenceSegment = 2;
```

Construct the training signal by calling `read` on the datastore in a loop.

```
numSamples = 1;
```

```

while numSamples < duration
    data = read(adsTrain);
    data = data ./ max(abs(data)); % Normalize amplitude

    % Determine regions of speech
    idx = detectSpeech(data,Fs,'Window',win,'Thresholds',T);

    % If a region of speech is detected
    if ~isempty(idx)

        % Extend the indices by five frames
        idx(1,1) = max(1,idx(1,1) - 5*numel(win));
        idx(1,2) = min(length(data),idx(1,2) + 5*numel(win));

        % Isolate the speech
        data = data(idx(1,1):idx(1,2));

        % Write speech segment to training signal
        audioTraining(numSamples:numSamples+numel(data)-1) = data;

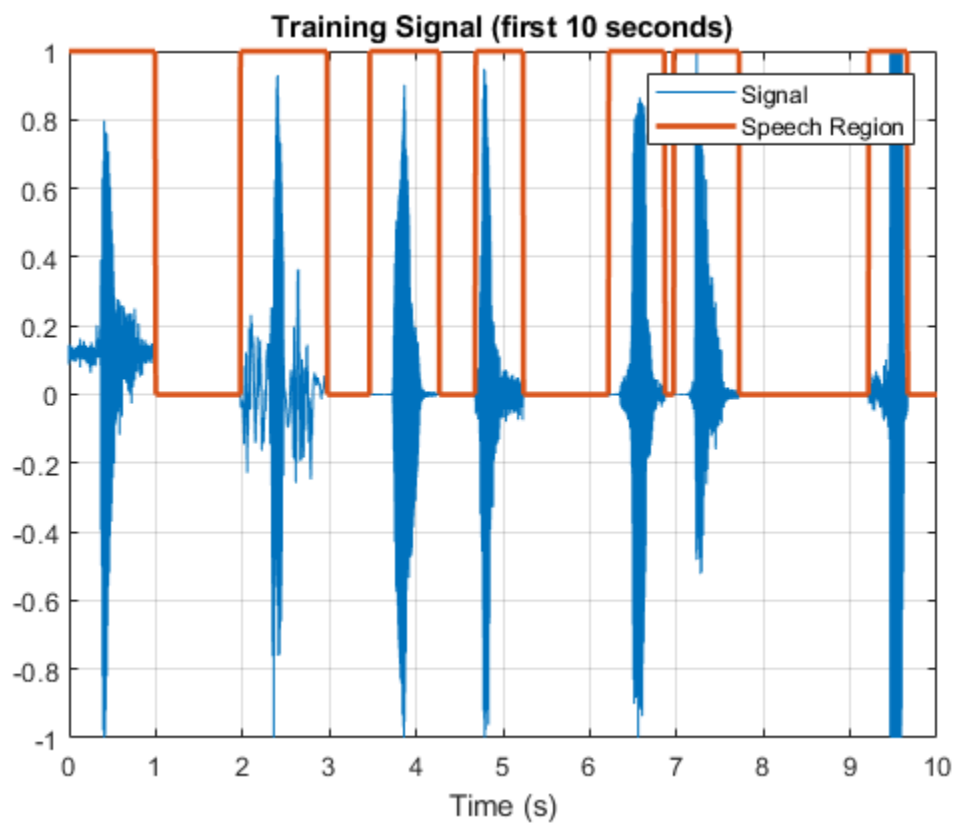
        % Set VAD baseline
        maskTraining(numSamples:numSamples+numel(data)-1) = true;

        % Random silence period
        numSilenceSamples = randi(maxSilenceSegment*Fs,1,1);
        numSamples = numSamples + numel(data) + numSilenceSamples;
    end
end

```

Visualize a 10-second portion of the training signal. Plot the baseline voice activity mask.

```
figure
range = 1:10*Fs;
plot((1/Fs)*(range-1),audioTraining(range));
hold on
plot((1/Fs)*(range-1),maskTraining(range));
grid on
lines = findall(gcf,"Type","Line");
lines(1).LineWidth = 2;
xlabel("Time (s)")
legend("Signal","Speech Region")
title("Training Signal (first 10 seconds)");
```



Listen to the first 10 seconds of the training signal.

```
sound(audioTraining(range),Fs);
```

Add Noise to the Training Signal

Corrupt the training signal with washing machine noise by adding washing machine noise to the speech signal such that the signal-to-noise ratio is -10 dB.

Read 8 kHz noise and convert it to 16 kHz.

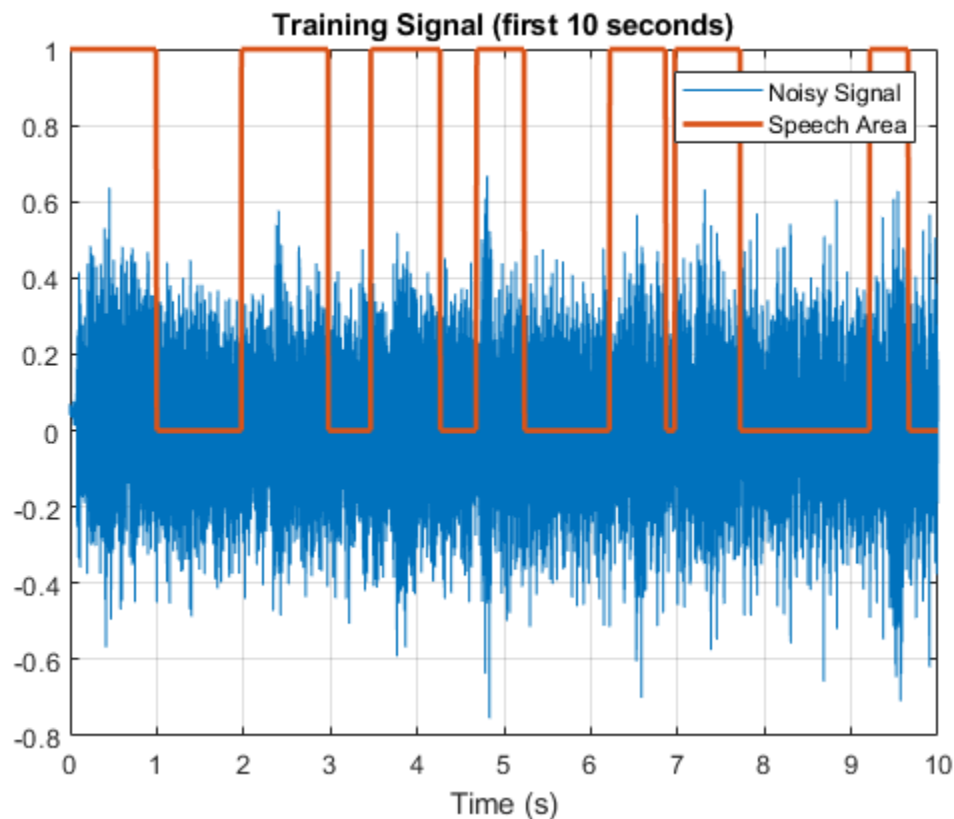
```
noise = audioread("WashingMachine-16-8-mono-1000secs.mp3");
noise = resample(noise,2,1);
```

Corrupt training signal with noise.

```
audioTraining = audioTraining(1:numel(noise));
SNR = -10;
noise = 10^(-SNR/20) * noise * norm(audioTraining) / norm(noise);
audioTrainingNoisy = audioTraining + noise;
audioTrainingNoisy = audioTrainingNoisy / max(abs(audioTrainingNoisy));
```

Visualize a 10-second portion of the noisy training signal. Plot the baseline voice activity mask.

```
figure
plot((1/Fs)*(range-1),audioTrainingNoisy(range));
hold on
plot((1/Fs)*(range-1),maskTraining(range));
grid on
lines = findall(gcf,"Type","Line");
lines(1).LineWidth = 2;
xlabel("Time (s)")
legend("Noisy Signal","Speech Area")
title("Training Signal (first 10 seconds)");
```



Listen to the first 10 seconds of the noisy training signal.

```
sound(audioTrainingNoisy(range),Fs)
```

Note that you obtained the baseline voice activity mask using the noiseless speech-plus-silence signal. Verify that using `detectSpeech` on the noise-corrupted signal does not yield good results.

```

speechIndices = detectSpeech(audioTrainingNoisy,Fs,'Window',win);

speechIndices(:,1) = max(1,speechIndices(:,1) - 5*numel(win));
speechIndices(:,2) = min(numel(audioTrainingNoisy),speechIndices(:,2) + 5*numel(win));

noisyMask = zeros(size(audioTrainingNoisy));
for ii = 1:size(speechIndices)
    noisyMask(speechIndices(ii,1):speechIndices(ii,2)) = 1;
end

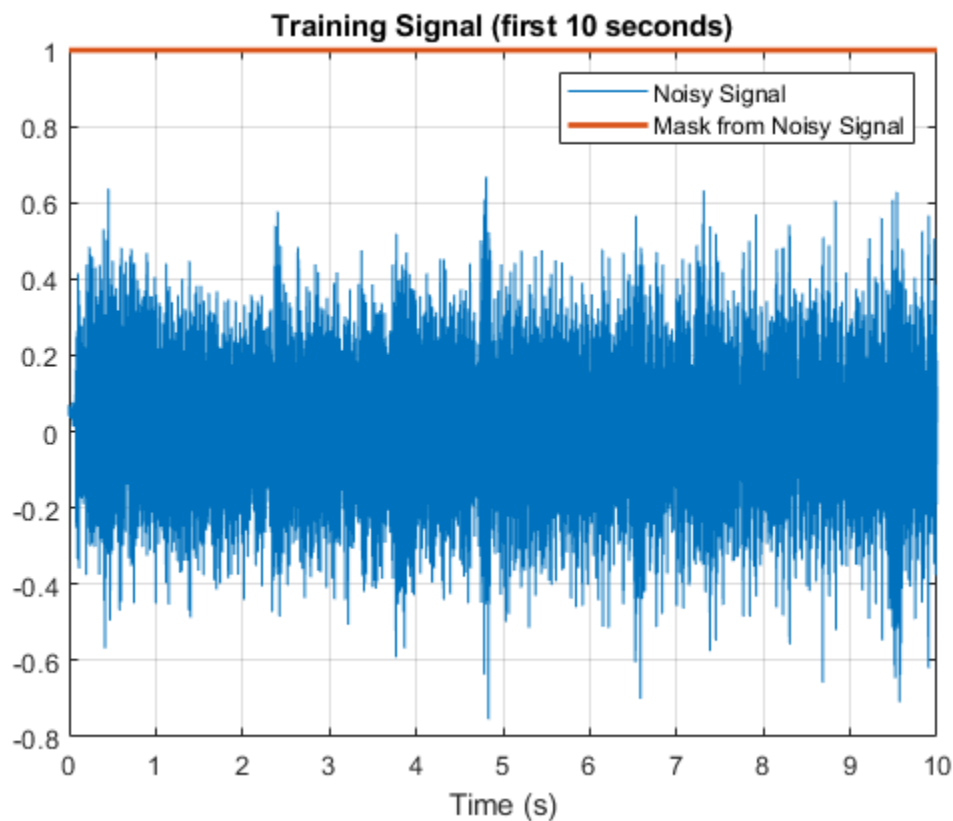
```

Visualize a 10-second portion of the noisy training signal. Plot the voice activity mask obtained by analyzing the noisy signal.

```

figure
plot((1/Fs)*(range-1),audioTrainingNoisy(range));
hold on
plot((1/Fs)*(range-1),noisyMask(range));
grid on
lines = findall(gcf,"Type","Line");
lines(1).LineWidth = 2;
xlabel("Time (s)")
legend("Noisy Signal","Mask from Noisy Signal")
title("Training Signal (first 10 seconds)");

```



Create Speech-Plus-Silence Validation Signal

Create a 200-second noisy speech signal to validate the trained network. Use the validation datastore. Note that the validation and training datastores have different speakers.

Preallocate the validation signal and the validation mask. You will use this mask to assess the accuracy of the trained network.

```
duration = 200*Fs;
audioValidation = zeros(duration,1);
maskValidation = zeros(duration,1);
```

Construct the validation signal by calling `read` on the datastore in a loop.

```
numSamples = 1;
while numSamples < duration
    data = read(adsValidation);
    data = data ./ max(abs(data)); % Normalize amplitude

    % Determine regions of speech
    idx = detectSpeech(data,Fs,'Window',win,'Thresholds',T);

    % If a region of speech is detected
    if ~isempty(idx)

        % Extend the indices by five frames
        idx(1,1) = max(1,idx(1,1) - 5*numel(win));
        idx(1,2) = min(length(data),idx(1,2) + 5*numel(win));

        % Isolate the speech
        data = data(idx(1,1):idx(1,2));

        % Write speech segment to training signal
        audioValidation(numSamples:numSamples+numel(data)-1) = data;

        % Set VAD Baseline
        maskValidation(numSamples:numSamples+numel(data)-1) = true;

        % Random silence period
        numSilenceSamples = randi(maxSilenceSegment*Fs,1,1);
        numSamples = numSamples + numel(data) + numSilenceSamples;
    end
end
```

Corrupt the validation signal with washing machine noise by adding washing machine noise to the speech signal such that the signal-to-noise ratio is -10 dB. Use a different noise file for the validation signal than you did for the training signal.

```
noise = audioread("WashingMachine-16-8-mono-200secs.mp3");
noise = resample(noise,2,1);
noise = noise(1:duration);
audioValidation = audioValidation(1:numel(noise));

noise = 10^(-SNR/20) * noise * norm(audioValidation) / norm(noise);
audioValidationNoisy = audioValidation + noise;
audioValidationNoisy = audioValidationNoisy / max(abs(audioValidationNoisy));
```

Extract Training Features

This example trains the LSTM network using the following features:

- 1 spectralCentroid

```
2 spectralCrest
3 spectralEntropy
4 spectralFlux
5 spectralKurtosis
6 spectralRolloffPoint
7 spectralSkewness
8 spectralSlope
9 harmonicRatio
```

This example uses `audioFeatureExtractor` to create an optimal feature extraction pipeline for the feature set. Create an `audioFeatureExtractor` object to extract the feature set. Use a 256-point Hann window with 50% overlap.

```
afe = audioFeatureExtractor('SampleRate',Fs, ...
    'Window',hann(256,"Periodic"), ...
    'OverlapLength',128, ...
    ...
    'spectralCentroid',true, ...
    'spectralCrest',true, ...
    'spectralEntropy',true, ...
    'spectralFlux',true, ...
    'spectralKurtosis',true, ...
    'spectralRolloffPoint',true, ...
    'spectralSkewness',true, ...
    'spectralSlope',true, ...
    'harmonicRatio',true);

featuresTraining = extract(afe,audioTrainingNoisy);
```

Display the dimensions of the features matrix. The first dimension corresponds to the number of windows the signal was broken into (it depends on the window length and the overlap length). The second dimension is the number of features used in this example.

```
[numWindows,numFeatures] = size(featuresTraining)
```

```
numWindows = 125009
```

```
numFeatures = 9
```

In classification applications, it is a good practice to normalize all features to have zero mean and unity standard deviation.

Compute the mean and standard deviation for each coefficient, and use them to normalize the data.

```
M = mean(featuresTraining,1);
S = std(featuresTraining,[],1);
featuresTraining = (featuresTraining - M) ./ S;
```

Extract the features from the validation signal using the same process.

```
featuresValidation = extract(afe,audioValidationNoisy);
featuresValidation = (featuresValidation - mean(featuresValidation,1)) ./ std(featuresValidation,1);
```

Each feature corresponds to 128 samples of data (the hop length). For each hop, set the expected voice/no voice value to the mode of the baseline mask values corresponding to those 128 samples. Convert the voice/no voice mask to categorical.

```
windowLength = numel(afe.Window);
hopLength = windowLength - afe.OverlapLength;
range = (hopLength) * (1:size(featuresTraining,1)) + hopLength;
maskMode = zeros(size(range));
for index = 1:numel(range)
    maskMode(index) = mode(maskTraining( (index-1)*hopLength+1:(index-1)*hopLength+windowLength ));
end
maskTraining = maskMode.';

maskTrainingCat = categorical(maskTraining);
```

Do the same for the validation mask.

```
range = (hopLength) * (1:size(featuresValidation,1)) + hopLength;
maskMode = zeros(size(range));
for index = 1:numel(range)
    maskMode(index) = mode(maskValidation( (index-1)*hopLength+1:(index-1)*hopLength+windowLength ));
end
maskValidation = maskMode.';

maskValidationCat = categorical(maskValidation);
```

Split the training features and the mask into sequences of length 800, with 75% overlap between consecutive sequences.

```
sequenceLength = 800;
sequenceOverlap = round(0.75*sequenceLength);

trainFeatureCell = helperFeatureVector2Sequence(featuresTraining',sequenceLength,sequenceOverlap);
trainLabelCell = helperFeatureVector2Sequence(maskTrainingCat',sequenceLength,sequenceOverlap);
```

Define the LSTM Network Architecture

LSTM networks can learn long-term dependencies between time steps of sequence data. This example uses the bidirectional LSTM layer `bilstmLayer` (Deep Learning Toolbox) to look at the sequence in both forward and backward directions.

Specify the input size to be sequences of length 9 (the number of features). Specify a hidden bidirectional LSTM layer with an output size of 200 and output a sequence. This command instructs the bidirectional LSTM layer to map the input time series into 200 features that are passed to the next layer. Then, specify a bidirectional LSTM layer with an output size of 200 and output the last element of the sequence. This command instructs the bidirectional LSTM layer to map its input into 200 features and then prepares the output for the fully connected layer. Finally, specify two classes by including a fully connected layer of size 2, followed by a softmax layer and a classification layer.

```
layers = [ ...
    sequenceInputLayer( size(featuresValidation,2) )
    bilstmLayer(200,"OutputMode","sequence")
    bilstmLayer(200,"OutputMode","sequence")
    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer
];
```

Next, specify the training options for the classifier. Set `MaxEpochs` to 20 so that the network makes 20 passes through the training data. Set `MiniBatchSize` to 64 so that the network looks at 64 training signals at a time. Set `Plots` to "training-progress" to generate plots that show the training progress as the number of iterations increases. Set `Verbose` to `false` to disable printing the table output that corresponds to the data shown in the plot. Set `Shuffle` to "every-epoch" to shuffle the training sequence at the beginning of each epoch. Set `LearnRateSchedule` to "piecewise" to decrease the learning rate by a specified factor (0.1) every time a certain number of epochs (10) has passed. Set `ValidationData` to the validation predictors and targets.

This example uses the adaptive moment estimation (ADAM) solver. ADAM performs better with recurrent neural networks (RNNs) like LSTMs than the default stochastic gradient descent with momentum (SGDM) solver.

```
maxEpochs = 20;
miniBatchSize = 64;
options = trainingOptions("adam", ...
    "MaxEpochs",maxEpochs, ...
    "MiniBatchSize",miniBatchSize, ...
    "Shuffle","every-epoch", ...
    "Verbose",0, ...
    "SequenceLength",sequenceLength, ...
    "ValidationFrequency",floor(numel(trainFeatureCell)/miniBatchSize), ...
    "ValidationData",{featuresValidation.',maskValidationCat.'}, ...
    "Plots","training-progress", ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.1, ...
    "LearnRateDropPeriod",5);
```

Train the LSTM Network

Train the LSTM network with the specified training options and layer architecture using `trainNetwork`. Because the training set is large, the training process can take several minutes.

```
doTraining = true;
if doTraining
    [speechDetectNet,netInfo] = trainNetwork(trainFeatureCell,trainLabelCell,layers,options);
    fprintf("Validation accuracy: %f percent.\n", netInfo.FinalValidationAccuracy);
else
    load speechDetectNet
end

Validation accuracy: 90.089844 percent.
```

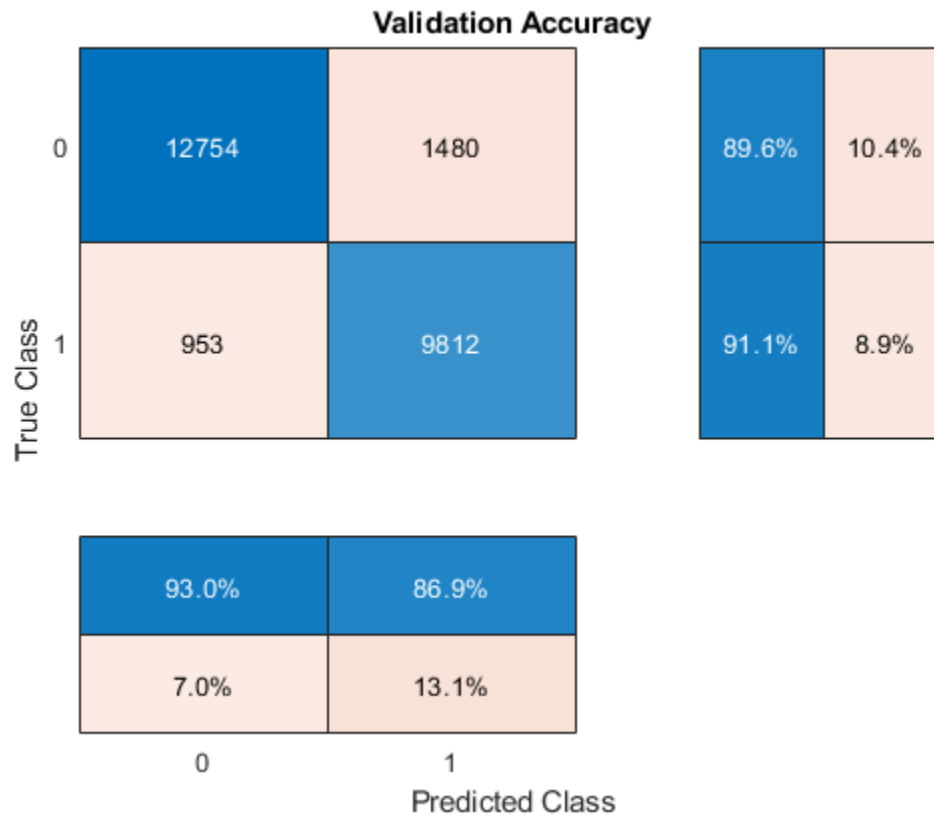
Use Trained Network to Detect Voice Activity

Estimate voice activity in the validation signal using the trained network. Convert the estimated VAD mask from categorical to double.

```
EstimatedVADMask = classify(speechDetectNet,featuresValidation. ');
EstimatedVADMask = double(EstimatedVADMask);
EstimatedVADMask = EstimatedVADMask.' - 1;
```

Calculate and plot the validation confusion matrix from the vectors of actual and estimated labels.

```
figure
cm = confusionchart(maskValidation,EstimatedVADMask,"title","Validation Accuracy");
cm.ColumnSummary = "column-normalized";
cm.RowSummary = "row-normalized";
```

If you changed parameters of your network or feature extraction pipeline, consider resaving the MAT file with the new network and audioFeatureExtractor object.

```
resaveNetwork = ;
if resaveNetwork
    save('Audio_VoiceActivityDetectionExample.mat','speechDetectNet','afe');
end
```

Supporting Functions

Convert Feature Vectors to Sequences

```
function [sequences,sequencePerFile] = helperFeatureVector2Sequence(features,featureVectorsPerSequence,featureVectorOverlap)
    if featureVectorsPerSequence <= featureVectorOverlap
        error('The number of overlapping feature vectors must be less than the number of feature vectors');
    end

    if ~iscell(features)
        features = {features};
    end
    hopLength = featureVectorsPerSequence - featureVectorOverlap;
    idx1 = 1;
    sequences = {};
    sequencePerFile = cell(numel(features),1);
    for ii = 1:numel(features)
        sequencePerFile{ii} = floor((size(features{ii},2) - featureVectorsPerSequence)/hopLength) + 1;
        idx2 = 1;
```

```
        for j = 1:sequencePerFile{ii}
            sequences{idx1,1} = features{ii}(:,idx2:idx2 + featureVectorsPerSequence - 1); %#ok<
            idx1 = idx1 + 1;
            idx2 = idx2 + hopLength;
        end
    end
end
```

Streaming Demo

`function` helperStreamingDemo(speechDetectNet,afe,cleanSpeech,noise,testDuration,sequenceLength,s

Create `dsp.AudioFileReader` objects to read from the speech and noise files frame by frame.

```
speechReader = dsp.AudioFileReader(cleanSpeech,'PlayCount',inf);
noiseReader = dsp.AudioFileReader(noise,'PlayCount',inf);
fs = speechReader.SampleRate;
```

Create a `dsp.MovingStandardDeviation` object and a `dsp.MovingAverage` object. You will use these to determine the standard deviation and mean of the audio features for normalization. The statistics should improve over time.

```
movSTD = dsp.MovingStandardDeviation('Method','Exponential weighting','ForgettingFactor',1);
movMean = dsp.MovingAverage('Method','Exponential weighting','ForgettingFactor',1);
```

Create three `dsp.AsyncBuffer` objects. One to buffer the input audio, one to buffer the extracted features, and one to buffer the output buffer. The output buffer is only necessary for visualizing the decisions in real time.

```
audioInBuffer = dsp.AsyncBuffer;
featureBuffer = dsp.AsyncBuffer;
audioOutBuffer = dsp.AsyncBuffer;
```

For the audio buffers, you will buffer both the original clean speech signal, and the noisy signal. You will play back only the specified `signalToListenTo`. Convert the `signalToListenTo` variable to the channel you want to listen to.

```
channelToListenTo = 1;
if strcmp(signalToListenTo,"clean")
    channelToListenTo = 2;
end
```

Create a `dsp.TimeScope` to visualize the original speech signal, the noisy signal that the network is applied to, and the decision output from the network.

```
scope = dsp.TimeScope('SampleRate',fs, ...
    'TimeSpan',3, ...
    'BufferLength',fs*3*3, ...
    'YLimits',[-1.2 1.2], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'NumInputPorts',3, ...
    'LayoutDimensions',[3,1], ...
    'Title','Noisy Speech');
scope.ActiveDisplay = 2;
scope.Title = 'Clean Speech (Original)';
scope.ActiveDisplay = 3;
scope.Title = 'Detected Speech';
```

Create an `audioDeviceWriter` object to play either the original or noisy audio from your speakers.

```
deviceWriter = audioDeviceWriter('SampleRate',fs);
```

Initialize variables used in the loop.

```
windowLength = numel(afe.Window);
hopLength = windowLength - afe.OverlapLength;
myMax = 0;
audioBufferInitialized = false;
featureBufferInitialized = false;
```

Run the streaming demonstration.

```
tic
while toc < testDuration

    % Read a frame of the speech signal and a frame of the noise signal
    speechIn = speechReader();
    noiseIn = noiseReader();

    % Mix the speech and noise at the specified SNR
    noisyAudio = speechIn + noiseGain*noiseIn;

    % Update a running max for normalization
    myMax = max(myMax,max(abs(noisyAudio)));

    % Write the noisy audio and speech to buffers
    write(audioInBuffer,[noisyAudio,speechIn]);

    % If enough samples are buffered,
    % mark the audio buffer as initialized and push the read pointer
    % for the audio buffer up a window length.
    if audioInBuffer.NumUnreadSamples >= windowLength && ~audioBufferInitialized
        audioBufferInitialized = true;
        read(audioInBuffer,windowLength);
    end

    % If enough samples are in the audio buffer to calculate a feature
    % vector, read the samples, normalize them, extract the feature vectors, and write
    % the latest feature vector to the features buffer.
    while (audioInBuffer.NumUnreadSamples >= hopLength) && audioBufferInitialized
        x = read(audioInBuffer,windowLength + hopLength,windowLength);
        write(audioOutBuffer,x(end-hopLength+1:end,:));
        noisyAudio = x(:,1);
        noisyAudio = noisyAudio/myMax;
        features = extract(afe,noisyAudio);
        write(featureBuffer,features(2,:));
    end

    % If enough feature vectors are buffered, mark the feature buffer
    % as initialized and push the read pointer for the feature buffer
    % and the audio output buffer (so that they are in sync).
    if featureBuffer.NumUnreadSamples >= (sequenceLength + sequenceHop) && ~featureBufferIni
        featureBufferInitialized = true;
        read(featureBuffer,sequenceLength - sequenceHop);
        read(audioOutBuffer,(sequenceLength - sequenceHop)*windowLength);
    end
end
```

```
while featureBuffer.NumUnreadSamples >= sequenceHop && featureBufferInitialized
    features = read(featureBuffer,sequenceLength,sequenceLength - sequenceHop);
    features(isnan(features)) = 0;

    % Use only the new features to update the
    % standard deviation and mean. Normalize the features.
    localSTD = movSTD(features(end-sequenceHop+1:end,:));
    localMean = movMean(features(end-sequenceHop+1:end,:));
    features = (features - localMean(end,:)) ./ localSTD(end,:);

    decision = classify(speechDetectNet,features');
    decision = decision(end-sequenceHop+1:end);
    decision = double(decision)' - 1;
    decision = repelem(decision,hopLength);

    audioHop = read(audioOutBuffer,sequenceHop*hopLength);

    % Listen to the speech or speech+noise
    deviceWriter(audioHop(:,channelToListenTo));

    % Visualize the speech+noise, the original speech, and the
    % voice activity detection.
    scope(audioHop(:,1),audioHop(:,2),audioHop(:,1).*decision)
end
end
release(deviceWriter)
release(audioInBuffer)
release(audioOutBuffer)
release(featureBuffer)
release(movSTD)
release(movMean)
release(scope)
end
```

References

[1] Warden P. "Speech Commands: A public dataset for single-word speech recognition", 2017. Available from https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz. Copyright Google 2017. The Speech Commands Dataset is licensed under the Creative Commons Attribution 4.0 license

Using a MIDI Control Surface to Interact with a Simulink Model

This example shows how to use a MIDI control surface as a physical user interface to a Simulink model, allowing you to use knobs, sliders and buttons to interact with that model. It can be used in Simulink as well as with generated code running on a workstation.

Introduction

Although MIDI is best known for its use in audio applications, this example illustrates that MIDI control surfaces have uses in many other applications besides audio. In this example, we use a MIDI controller to provide a user configurable value that can vary at runtime, we use it to control the amplitude of signals, and for several other illustrative purposes. This example is not comprehensive, but rather can provide inspiration for other creative uses of the control surface to interact with a model.

By "MIDI control surfaces", we mean a physical device that

- 1 has knobs, sliders and push buttons,
- 2 and uses the MIDI (Musical Instrument Digital Interface) protocol.

Many MIDI controllers plug into the USB port on a computer and make use of the MIDI support built into modern operating systems. Specific MIDI control surfaces that we have used include the Korg nanoKONTROL and the Behringer BCF2000. An advantage of the Korg device is its cost: it is readily available online at prices comparable to that of a good mouse. The Behringer device is more costly, but has the enhanced capability to both send and receive MIDI signals (the Korg can only send signals). This ability can be used to send data back from a model to keep a control surface in sync with changes to the model. We use this capability to bring a control surface in sync with the starting point of a model, so that initially changes to a specific control do not produce abrupt changes in the block output.

To use your own controller with this example, plug it into the USB port on the computer and run the model `audiomidi`. Be sure that the model is not running when you plug in the control device. The model is originally configured such that it responds to movement of any control on the default MIDI device. This construction is meant to make it easier and more likely that this example works out of the box for all users. In a real use case, you would probably want to tie individual controls to each sub-portion of the model. For that purpose, you can use the `midid` function to explicitly set the MIDI device parameter on the appropriate blocks in the model to recognize a specific control. For example, running `midid` with the Korg nanoKONTROL device produces the following information:

```
>> [ctl device]=midid
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done

ctl =

    1002

device =

nanoKONTROL
```

The actual value of `ctl` depends on which control you moved.

If you will be using a particular controller repeatedly, you may want to use the `setpref` command to set that controller as the default midi device:

```
>> setpref('midi','DefaultDevice','nanoKONTROL')
```

This capability is particularly helpful on Linux, where your control surface may not be immediately recognized as the default device.

After the controller is plugged in, hit the play button on audiomidi. Now move any knob or slider. You should see variations in the signals that are plotted in the various scopes in the model as you move any knob or slider. The model is initially configured to respond to any control.

Examples

Next, several example use cases are provided. Each example uses the basic **MIDI Controls** block to accomplish a different task. Look under the mask of the appropriate block in each example to see how that use case was accomplished. To reuse these in your own model, just drag a copy of the desired block into your model.

Example 1: MIDI Controls as a User Defined Source

In example 1 of the model, we see the simplest use of this control. It can act as a source that is under user control. The original block **MIDI Controls** (in the DSP sources block library), outputs a value between 0 and 1. We have also created a slightly modified block, by placing a mask on the original block to output a source with values that cover a user defined range.

Example 2: MIDI Controls to Adjust the Level of a Single Signal

In this example, a straightforward application of the **MIDI controls** block uses the 0 to 1 range as an amplitude control on a given signal.

Example 3: MIDI Controls to Split a Signal Into Two Streams With User Controlled Relative Amplitudes.

In this example, we see an example where a signal is split into two streams: αu and $(1 - \alpha) u$ where α can be interactively controlled by the user with the control surface.

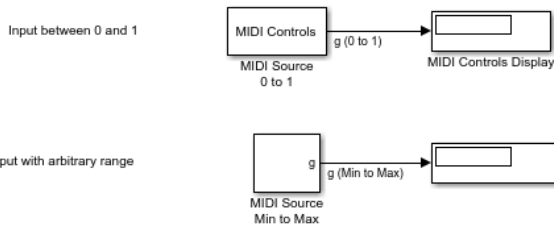
Example 4: MIDI Controls to Mix Two Signals Into One

In this example, we create an arbitrary linear combination of two inputs: $y = \alpha u_1 + (1 - \alpha) u_2$ with α being set interactively by the user with the control surface.

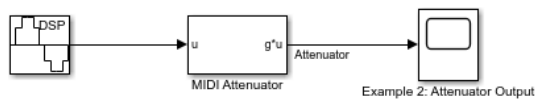
Example 5: MIDI Controls to Generate a Sinusoid with Arbitrary Phase

Lastly, example 5 allows the user input a desired phase with the control surface. A sinusoid with that phase is then generated. The phase can be interactively varied as the model runs.

Example 1: Use the MIDI control surface as a user-provided input to your model



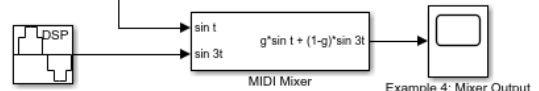
Example 2: Use the MIDI control surface to adjust the gain on a single signal



Example 3: Use the MIDI control surface to control the relative amplitudes as you split a single signal into two

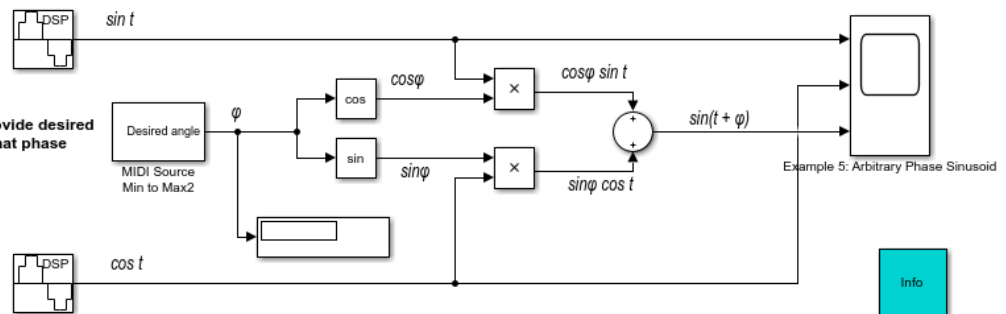


Example 4: Use the MIDI control surface to control the mix as you combine two signals into one



Example 5: Use the MIDI control surface to provide desired phase and then generate a sinusoid with that phase

$$\sin(t+\varphi) = \cos\varphi \sin t + \sin\varphi \cos t$$



Copyright 2011 The MathWorks Inc.

Conclusions

This model is provided to give inspiration for how the MIDI Controls block can be used to interact with a model. Other uses are possible and encouraged, including use with generated code.

Spoken Digit Recognition with Wavelet Scattering and Deep Learning

This example shows how to classify spoken digits using both machine and deep learning techniques. In the example, you perform classification using wavelet time scattering with a support vector machine (SVM) and with a long short-term memory (LSTM) network. You also apply Bayesian optimization to determine suitable hyperparameters to improve the accuracy of the LSTM network. In addition, the example illustrates an approach using a deep convolutional neural network (CNN) and mel-frequency spectrograms.

Data

Clone or download the Free Spoken Digit Dataset (FSDD), available at <https://github.com/Jakobovski/free-spoken-digit-dataset>. FSDD is an open data set, which means that it can grow over time. This example uses the version committed on January 29, 2019, which consists of 2000 recordings in English of the digits 0 through 9 obtained from four speakers. In this version, two of the speakers are native speakers of American English, one speaker is a nonnative speaker of English with a Belgian French accent, and one speaker is a nonnative speaker of English with a German accent. The data is sampled at 8000 Hz.

Use `audioDatastore` to manage data access and ensure the random division of the recordings into training and test sets. Set the `location` property to the location of the FSDD recordings folder on your computer, for example:

```
pathToRecordingsFolder = fullfile(tempdir, 'free-spoken-digit-dataset', 'recordings');  
location = pathToRecordingsFolder;
```

Point `audioDatastore` to that location.

```
ads = audioDatastore(location);
```

The helper function `helpergenLabels` creates a categorical array of labels from the FSDD files. The source code for `helpergenLabels` is listed in the appendix. List the classes and the number of examples in each class.

```
ads.Labels = helpergenLabels(ads);  
summary(ads.Labels)
```

```
0      200  
1      200  
2      200  
3      200  
4      200  
5      200  
6      200  
7      200  
8      200  
9      200
```

The FSDD data set consists of 10 balanced classes with 200 recordings each. The recordings in the FSDD are not of equal duration. The FSDD is not prohibitively large, so read through the FSDD files and construct a histogram of the signal lengths.

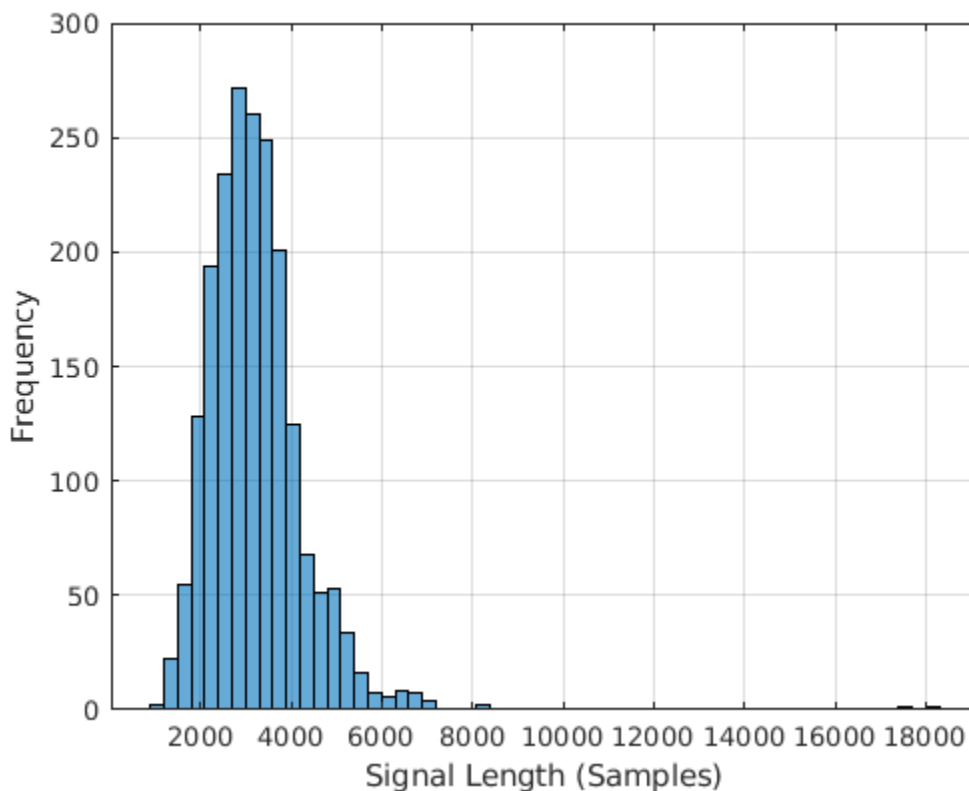
```
LenSig = zeros(numel(ads.Files),1);  
nr = 1;
```



```

while hasdata(ads)
    digit = read(ads);
    LenSig(nr) = numel(digit);
    nr = nr+1;
end
reset(ads)
histogram(LenSig)
grid on
xlabel('Signal Length (Samples)')
ylabel('Frequency')

```



The histogram shows that the distribution of recording lengths is positively skewed. For classification, this example uses a common signal length of 8192 samples, a conservative value that ensures that truncating longer recordings does not cut off speech content. If the signal is greater than 8192 samples (1.024 seconds) in length, the recording is truncated to 8192 samples. If the signal is less than 8192 samples in length, the signal is prepadded and postpadded symmetrically with zeros out to a length of 8192 samples.

Wavelet Time Scattering

Use `waveletScattering` (Wavelet Toolbox) to create a wavelet time scattering framework using an invariant scale of 0.22 seconds. In this example, you create feature vectors by averaging the scattering transform over all time samples. To have a sufficient number of scattering coefficients per time window to average, set `OversamplingFactor` to 2 to produce a four-fold increase in the number of scattering coefficients for each path with respect to the critically downsampled value.

```
sf = waveletScattering('SignalLength',8192,'InvarianceScale',0.22,...
    'SamplingFrequency',8000,'OversamplingFactor',2);
```

Split the FSDD into training and test sets. Allocate 80% of the data to the training set and retain 20% for the test set. The training data is for training the classifier based on the scattering transform. The test data is for validating the model.

```
rng default;
ads = shuffle(ads);
[adsTrain,adsTest] = splitEachLabel(ads,0.8);
countEachLabel(adsTrain)
```

```
ans=10x2 table
    Label    Count
    _____
         0      160
         1      160
         2      160
         3      160
         4      160
         5      160
         6      160
         7      160
         8      160
         9      160
```

```
countEachLabel(adsTest)
```

```
ans=10x2 table
    Label    Count
    _____
         0      40
         1      40
         2      40
         3      40
         4      40
         5      40
         6      40
         7      40
         8      40
         9      40
```

The helper function `helperReadSPData` truncates or pads the data to a length of 8192 and normalizes each recording by its maximum value. The source code for `helperReadSPData` is listed in the appendix. Create an 8192-by-1600 matrix where each column is a spoken-digit recording.

```
Xtrain = [];
scatds_Train = transform(adsTrain,@(x)helperReadSPData(x));
while hasdata(scatds_Train)
    smat = read(scatds_Train);
    Xtrain = cat(2,Xtrain,smat);
```

```
end
```

Repeat the process for the test set. The resulting matrix is 8192-by-400.

```

Xtest = [];
scatds_Test = transform(adsTest,@(x)helperReadSPData(x));
while hasdata(scatds_Test)
    smat = read(scatds_Test);
    Xtest = cat(2,Xtest,smat);
end

```

Apply the wavelet scattering transform to the training and test sets.

```

Strain = sf.featureMatrix(Xtrain);
Stest = sf.featureMatrix(Xtest);

```

Obtain the mean scattering features for the training and test sets. Exclude the zeroth-order scattering coefficients.

```

TrainFeatures = Strain(2:end,:,:);
TrainFeatures = squeeze(mean(TrainFeatures,2));
TestFeatures = Stest(2:end,:,:);
TestFeatures = squeeze(mean(TestFeatures,2));

```

SVM Classifier

Now that the data has been reduced to a feature vector for each recording, the next step is to use these features for classifying the recordings. Create an SVM learner template with a quadratic polynomial kernel. Fit the SVM to the training data.

```

template = templateSVM(...
    'KernelFunction', 'polynomial', ...
    'PolynomialOrder', 2, ...
    'KernelScale', 'auto', ...
    'BoxConstraint', 1, ...
    'Standardize', true);
classificationSVM = fitcecoc(...
    TrainFeatures, ...
    adsTrain.Labels, ...
    'Learners', template, ...
    'Coding', 'onevsone', ...
    'ClassNames', categorical({'0'; '1'; '2'; '3'; '4'; '5'; '6'; '7'; '8'; '9'}));

```

Use k-fold cross-validation to predict the generalization accuracy of the model based on the training data. Split the training set into five groups.

```

partitionedModel = crossval(classificationSVM, 'KFold', 5);
[validationPredictions, validationScores] = kfoldPredict(partitionedModel);
validationAccuracy = (1 - kfoldLoss(partitionedModel, 'LossFun', 'ClassificationError'))*100

validationAccuracy = 96.8125

```

The estimated generalization accuracy is approximately 97%. Use the trained SVM to predict the spoken-digit classes in the test set.

```

predLabels = predict(classificationSVM,TestFeatures);
testAccuracy = sum(predLabels==adsTest.Labels)/numel(predLabels)*100

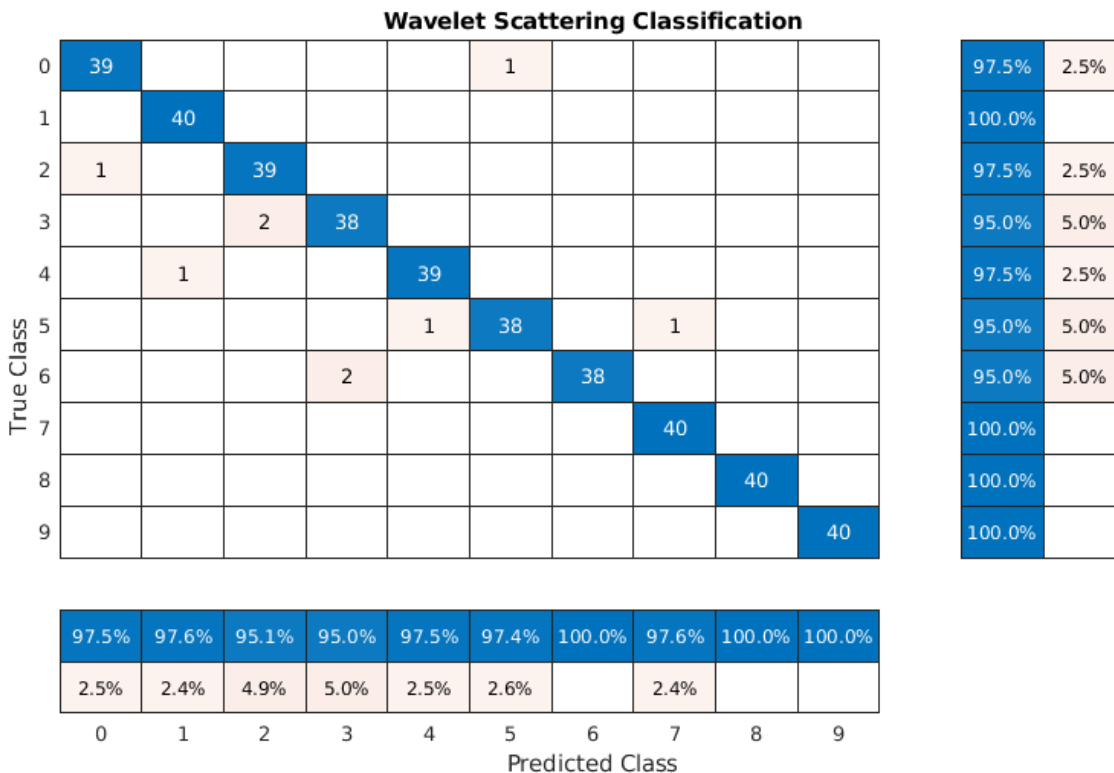
testAccuracy = 97.7500

```

Summarize the performance of the model on the test set with a confusion chart. Display the precision and recall for each class by using column and row summaries. The table at the bottom of the

confusion chart shows the precision values for each class. The table to the right of the confusion chart shows the recall values.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5]);
ccscat = confusionchart(adsTest.Labels,predLabels);
ccscat.Title = 'Wavelet Scattering Classification';
ccscat.ColumnSummary = 'column-normalized';
ccscat.RowSummary = 'row-normalized';
```



The scattering transform coupled with a SVM classifier classifies the spoken digits in the test set with an accuracy of 98% (or an error rate of 2%).

Long Short-Term Memory (LSTM) Networks

An LSTM network is a type of recurrent neural network (RNN). RNNs are neural networks that are specialized for working with sequential or temporal data such as speech data. Because the wavelet scattering coefficients are sequences, they can be used as inputs to an LSTM. By using scattering features as opposed to the raw data, you can reduce the variability that your network needs to learn.

Modify the training and testing scattering features to be used with the LSTM network. Exclude the zeroth-order scattering coefficients and convert the features to cell arrays.

```
TrainFeatures = Strain(2:end,:,:);
TrainFeatures = squeeze(num2cell(TrainFeatures,[1 2]));
TestFeatures = Stest(2:end,:,:);
TestFeatures = squeeze(num2cell(TestFeatures,[1 2]));
```

Construct a simple LSTM network with 512 hidden layers.

```
[inputSize, ~] = size(TrainFeatures{1});
YTrain = adsTrain.Labels;

numHiddenUnits = 512;
numClasses = numel(unique(YTrain));

layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits, 'OutputMode', 'last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

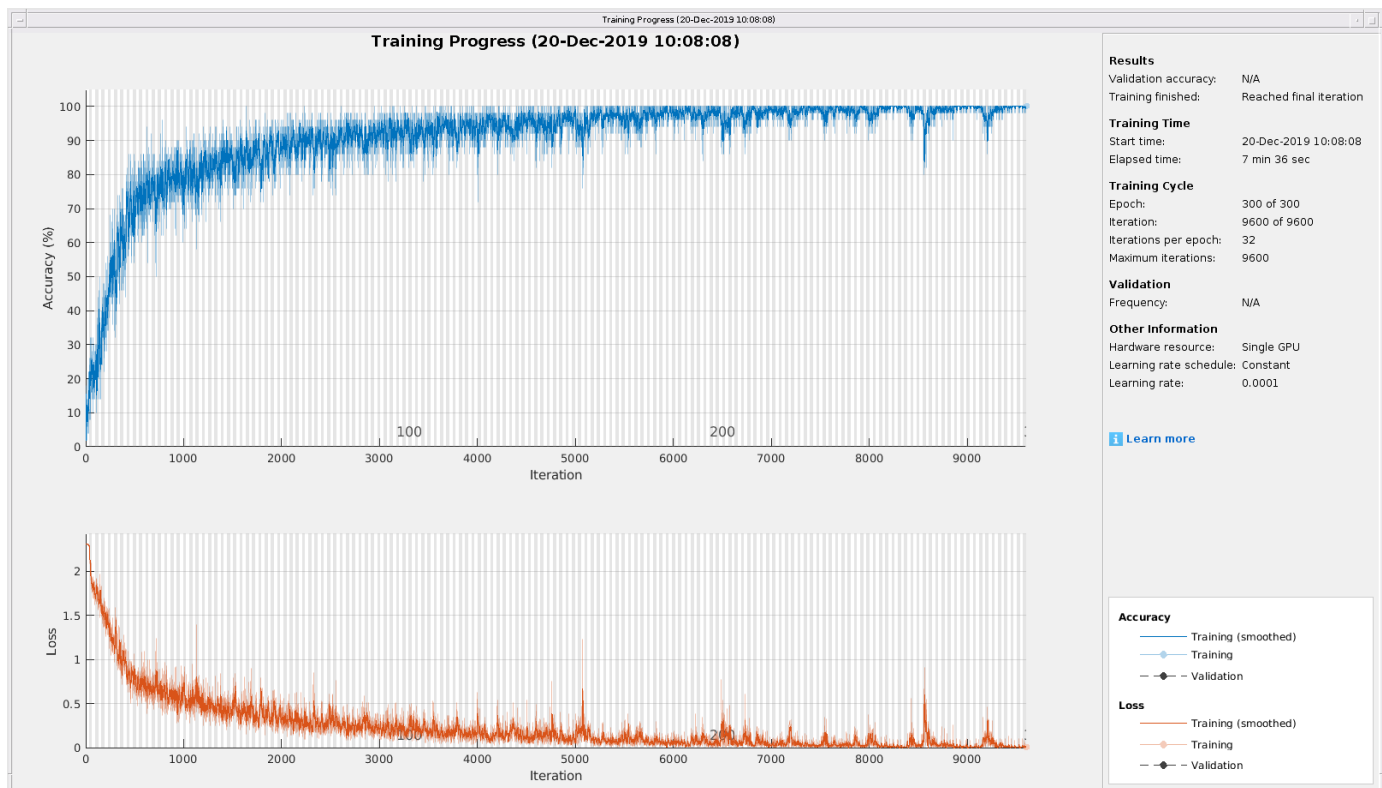
Set the hyperparameters. Use Adam optimization and a mini-batch size of 50. Set the maximum number of epochs to 300. Use a learning rate of $1e-4$. You can turn off the training progress plot if you do not want to track the progress using plots. The training uses a GPU by default if one is available. Otherwise, it uses a CPU. For more information, see `trainingOptions` (Deep Learning Toolbox).

```
maxEpochs = 300;
miniBatchSize = 50;

options = trainingOptions('adam', ...
    'InitialLearnRate', 0.0001, ...
    'MaxEpochs', maxEpochs, ...
    'MiniBatchSize', miniBatchSize, ...
    'SequenceLength', 'shortest', ...
    'Shuffle', 'every-epoch', ...
    'Verbose', false, ...
    'Plots', 'training-progress');
```

Train the network.

```
net = trainNetwork(TrainFeatures, YTrain, layers, options);
```



```
predLabels = classify(net,TestFeatures);
testAccuracy = sum(predLabels==adsTest.Labels)/numel(predLabels)*100

testAccuracy = 94
```

Bayesian Optimization

Determining suitable hyperparameter settings is often one of the most difficult parts of training a deep network. To mitigate this, you can use Bayesian optimization. In this example, you optimize the number of hidden layers and the initial learning rate by using Bayesian techniques. Create a new directory to store the MAT-files containing information about hyperparameter settings and the network along with the corresponding error rates.

```
YTrain = adsTrain.Labels;
YTest = adsTest.Labels;
```

```
if ~exist('results/','dir')
    mkdir results
end
```

Initialize the variables to be optimized and their value ranges. Because the number of hidden layers must be an integer, set 'type' to 'integer'.

```
optVars = [
    optimizableVariable('InitialLearnRate',[1e-5, 1e-1],'Transform','log')
    optimizableVariable('NumHiddenUnits',[10, 1000],'Type','integer')
];
```

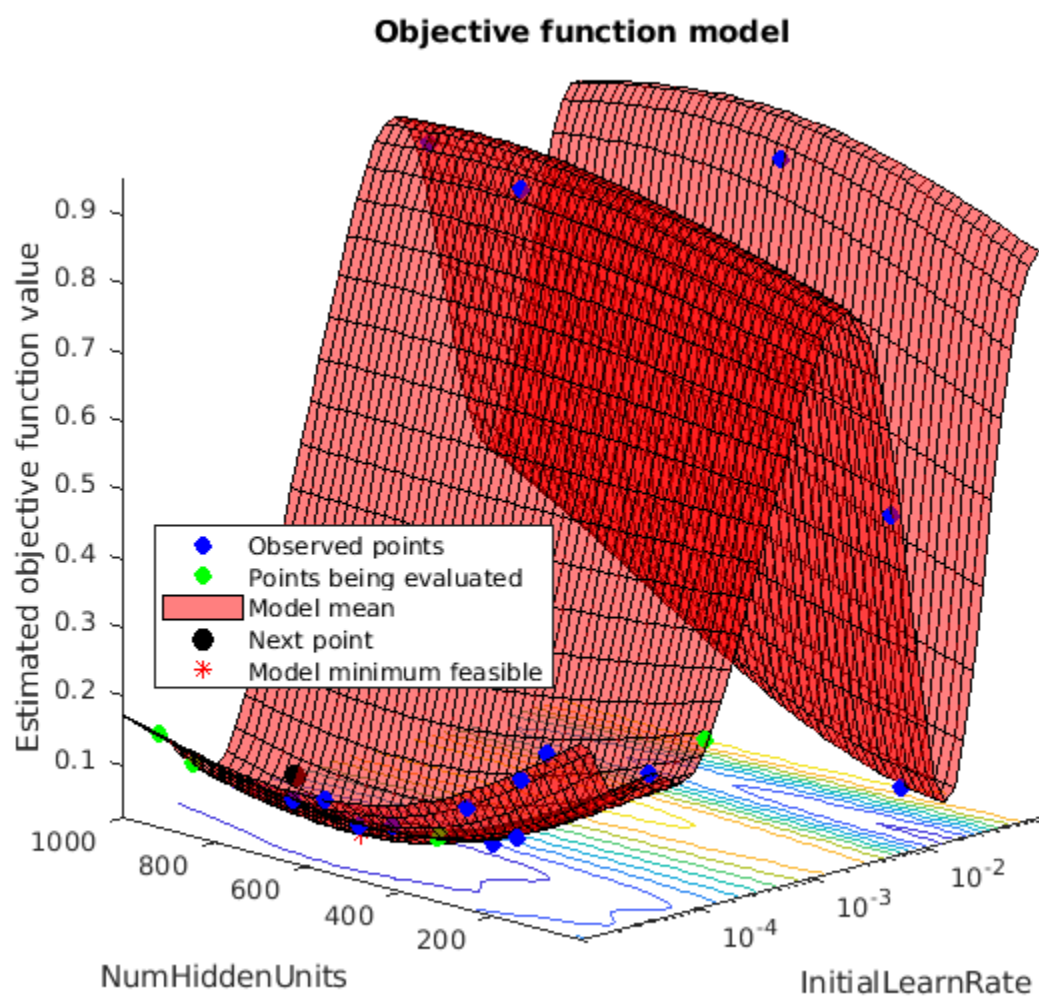
Bayesian optimization is computationally intensive and can take several hours to finish. For the purposes of this example, set `optimizeCondition` to `false` to load predetermined optimized

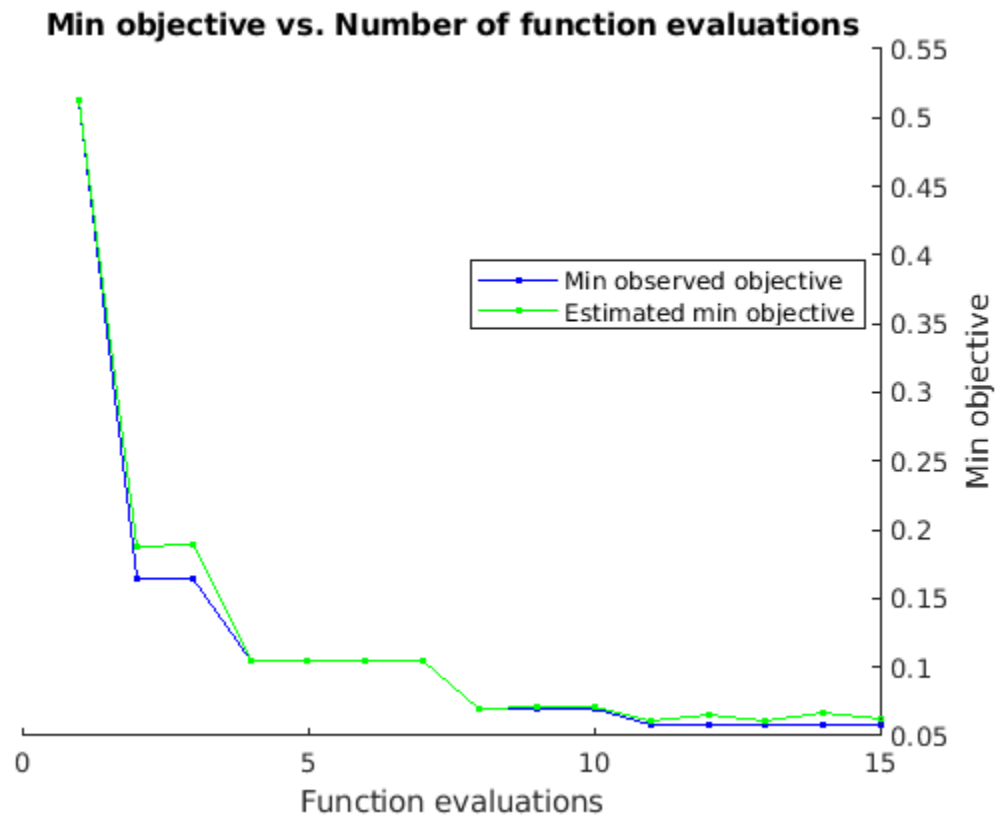
hyperparameter settings. If you set `optimizeCondition` to `true`, the objective function `helperBayesOptLSTM` is minimized using Bayesian optimization. The objective function, listed in the appendix, is the error rate of the network given specific hyperparameter settings. The loaded settings are for the objective function minimum of 0.02 (2% error rate).

```
ObjFcn = helperBayesOptLSTM(TrainFeatures, YTrain, TestFeatures, YTest);

optimizeCondition = false;
if optimizeCondition
    BayesObject = bayesopt(ObjFcn,optVars,...
        'MaxObjectiveEvaluations',15,...
        'IsObjectiveDeterministic',false,...
        'UseParallel',true);
else
    load 0.02.mat
end
```

If you perform Bayesian optimization, figures similar to the following are generated to track the objective function values with the corresponding hyperparameter values and the number of iterations. You can increase the number of Bayesian optimization iterations to ensure that the global minimum of the objective function is reached.





Use the optimized values for the number of hidden units and initial learning rate and retrain the network.

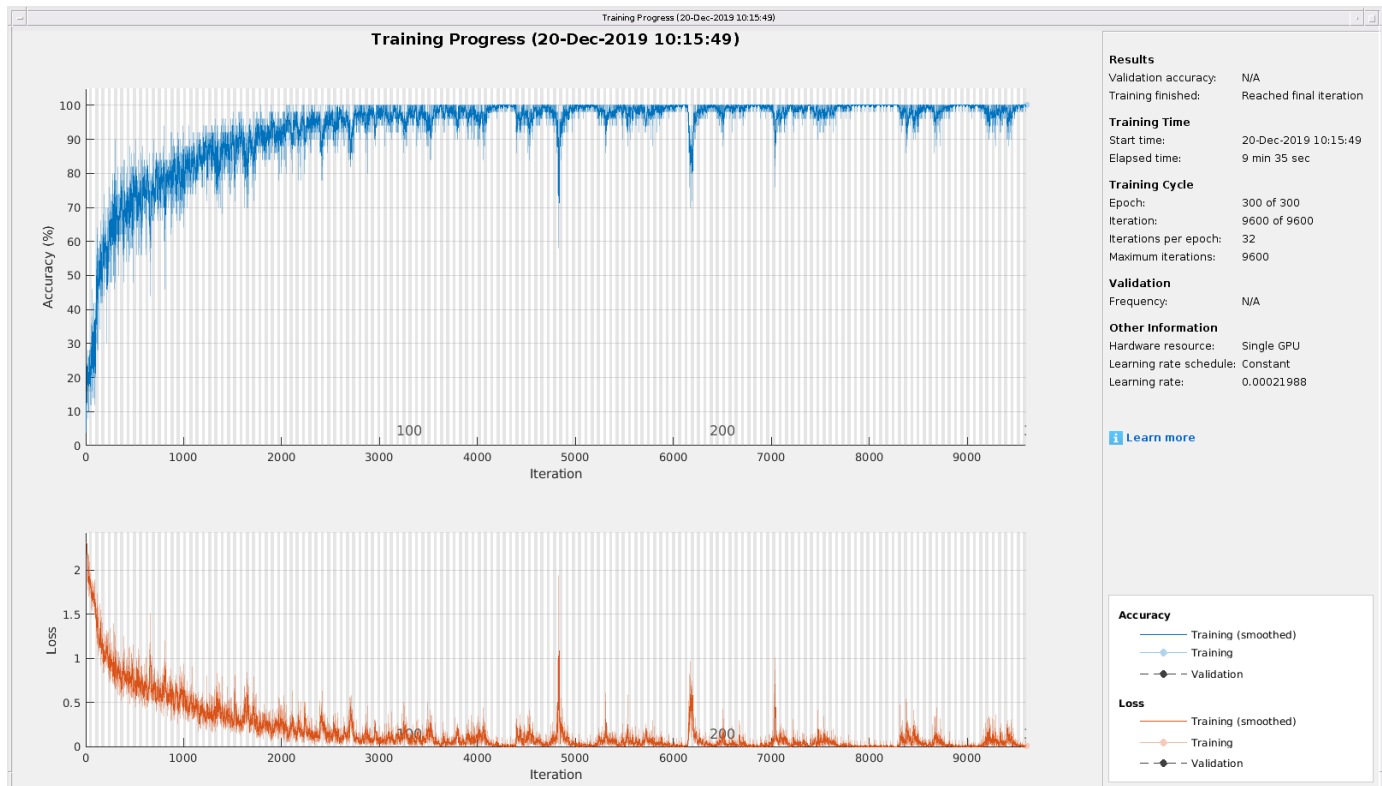
```
numHiddenUnits = 768;
numClasses = numel(unique(YTrain));

layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];

maxEpochs = 300;
miniBatchSize = 50;

options = trainingOptions('adam', ...
    'InitialLearnRate',2.198827960269379e-04,...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'SequenceLength','shortest', ...
    'Shuffle','every-epoch',...
    'Verbose', false, ...
    'Plots','training-progress');

net = trainNetwork(TrainFeatures,YTrain,layers,options);
```



```
predLabels = classify(net,TestFeatures);
testAccuracy = sum(predLabels==adsTest.Labels)/numel(predLabels)*100

testAccuracy = 97.7500
```

As the plot shows, using Bayesian optimization yields an LSTM with a higher accuracy.

Deep Convolutional Network Using Mel-Frequency Spectrograms

As another approach to the task of spoken digit recognition, use a deep convolutional neural network (DCNN) based on mel-frequency spectrograms to classify the FSDD data set. Use the same signal truncation/padding procedure as in the scattering transform. Similarly, normalize each recording by dividing each signal sample by the maximum absolute value. For consistency, use the same training and test sets as for the scattering transform.

Set the parameters for the mel-frequency spectrograms. Use the same window, or frame, duration as in the scattering transform, 0.22 seconds. Set the hop between windows to 10 ms. Use 40 frequency bands.

```
segmentDuration = 8192*(1/8000);
frameDuration = 0.22;
hopDuration = 0.01;
numBands = 40;
```

Reset the training and test datastores.

```
reset(adsTrain);
reset(adsTest);
```

The helper function `helperspeechSpectrograms`, defined at the end of this example, uses `melSpectrogram` to obtain the mel-frequency spectrogram after standardizing the recording length and normalizing the amplitude. Use the logarithm of the mel-frequency spectrograms as the inputs to the DCNN. To avoid taking the logarithm of zero, add a small epsilon to each element.

```

epsilon = 1e-6;
XTrain = helperspeechSpectrograms(adsTrain,segmentDuration,frameDuration,hopDuration,numBands);

Computing speech spectrograms...
Processed 500 files out of 1600
Processed 1000 files out of 1600
Processed 1500 files out of 1600
...done

XTrain = log10(XTrain + epsilon);

XTest = helperspeechSpectrograms(adsTest,segmentDuration,frameDuration,hopDuration,numBands);

Computing speech spectrograms...
...done

XTest = log10(XTest + epsilon);

YTrain = adsTrain.Labels;
YTest = adsTest.Labels;

```

Define DCNN Architecture

Construct a small DCNN as an array of layers. Use convolutional and batch normalization layers, and downsample the feature maps using max pooling layers. To reduce the possibility of the network memorizing specific features of the training data, add a small amount of dropout to the input to the last fully connected layer.

```

sz = size(XTrain);
specSize = sz(1:2);
imageSize = [specSize 1];

numClasses = numel(categories(YTrain));

dropoutProb = 0.2;
numF = 12;
layers = [
    imageInputLayer(imageSize)

    convolution2dLayer(5,numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,2*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer

```

```
reluLayer

maxPooling2dLayer(3,'Stride',2,'Padding','same')

convolution2dLayer(3,4*numF,'Padding','same')
batchNormalizationLayer
reluLayer
convolution2dLayer(3,4*numF,'Padding','same')
batchNormalizationLayer
reluLayer

maxPooling2dLayer(2)

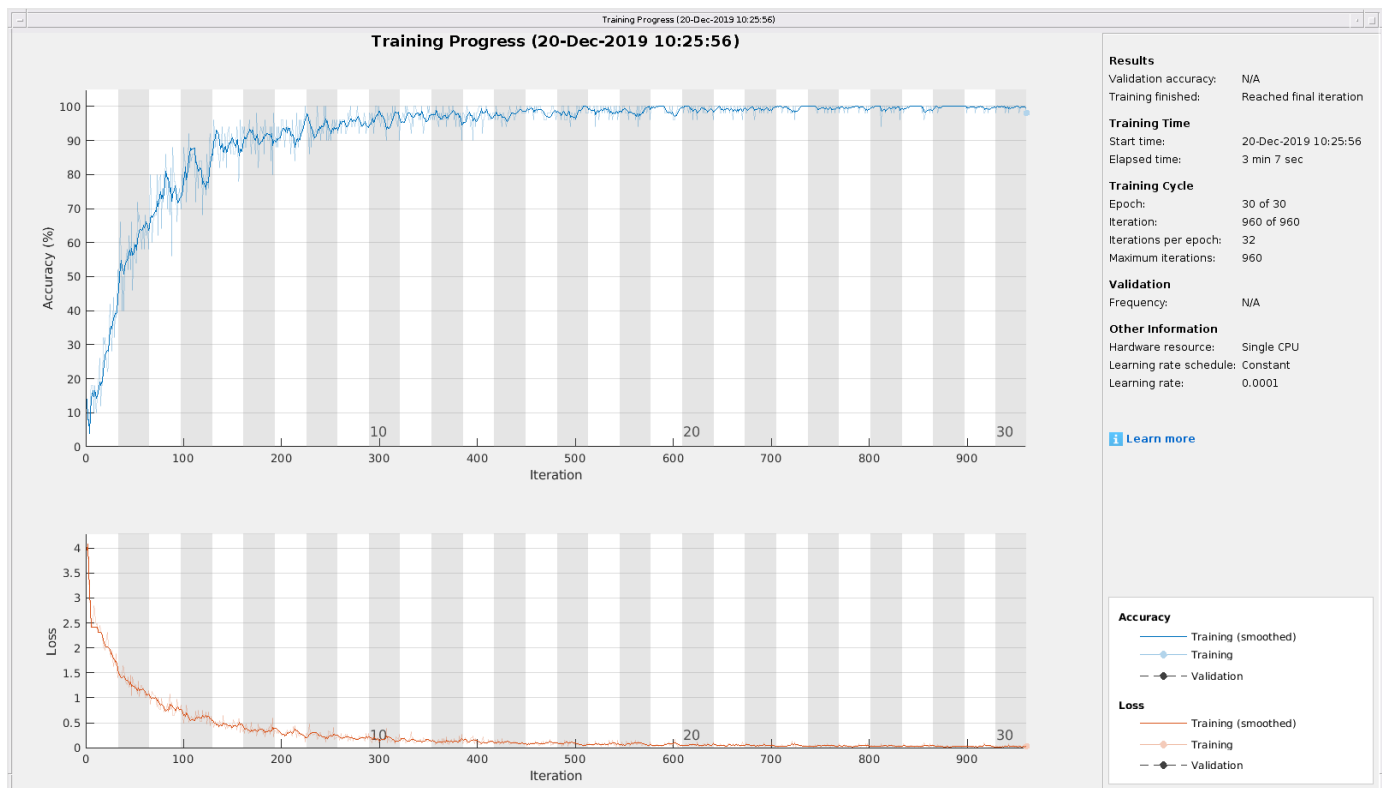
dropoutLayer(dropoutProb)
fullyConnectedLayer(numClasses)
softmaxLayer
classificationLayer('Classes',categories(YTrain));
];
```

Set the hyperparameters to use in training the network. Use a mini-batch size of 50 and a learning rate of 1e-4. Specify Adam optimization. Because the amount of data in this example is relatively small, set the execution environment to 'cpu' for reproducibility. You can also train the network on an available GPU by setting the execution environment to either 'gpu' or 'auto'. For more information, see `trainingOptions` (Deep Learning Toolbox).

```
miniBatchSize = 50;
options = trainingOptions('adam', ...
    'InitialLearnRate',1e-4, ...
    'MaxEpochs',30, ...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ExecutionEnvironment','cpu');
```

Train the network.

```
trainedNet = trainNetwork(XTrain,YTrain,layers,options);
```



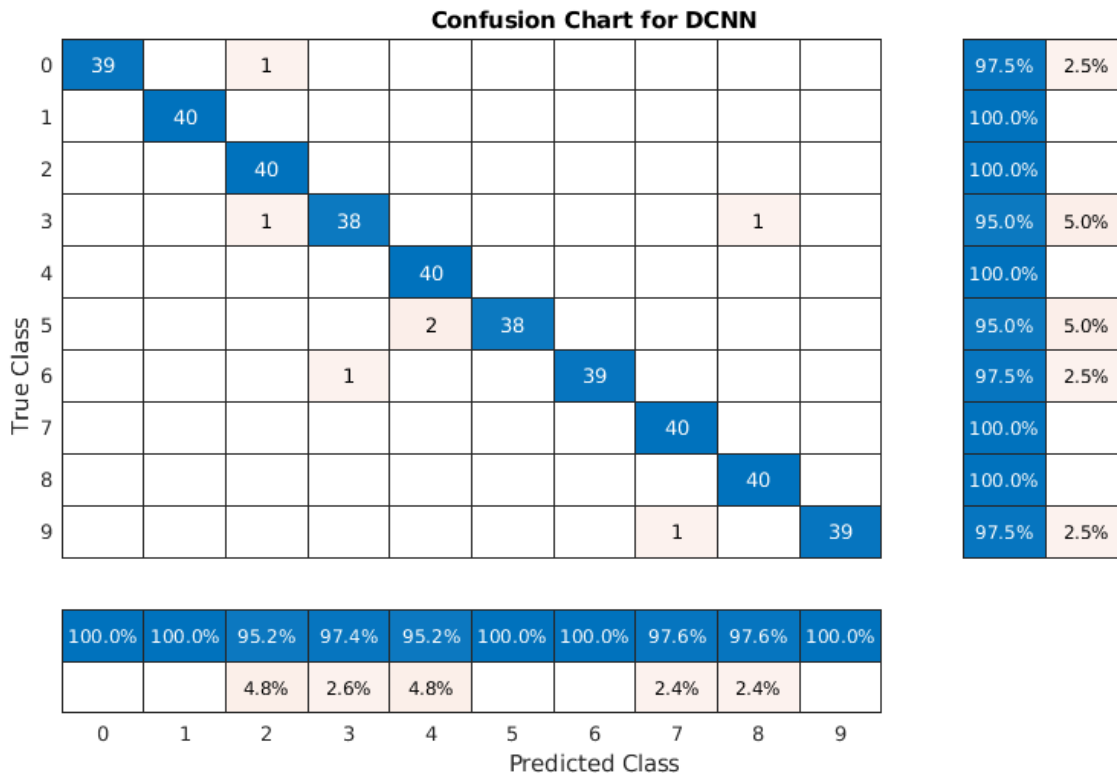
Use the trained network to predict the digit labels for the test set.

```
[Ypredicted,probs] = classify(trainedNet,XTest,'ExecutionEnvironment','CPU');
cnnAccuracy = sum(Ypredicted==YTest)/numel(YTest)*100

cnnAccuracy = 98.2500
```

Summarize the performance of the trained network on the test set with a confusion chart. Display the precision and recall for each class by using column and row summaries. The table at the bottom of the confusion chart shows the precision values. The table to the right of the confusion chart shows the recall values.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5]);
ccDCNN = confusionchart(YTest,Ypredicted);
ccDCNN.Title = 'Confusion Chart for DCNN';
ccDCNN.ColumnSummary = 'column-normalized';
ccDCNN.RowSummary = 'row-normalized';
```



The DCNN using mel-frequency spectrograms as inputs classifies the spoken digits in the test set with an accuracy rate of approximately 98% as well.

Summary

This example shows how to use different machine and deep learning approaches for classifying spoken digits in the FSDD. The example illustrated wavelet scattering paired with both an SVM and a LSTM. Bayesian techniques were used to optimize LSTM hyperparameters. Finally, the example shows how to use a CNN with mel-frequency spectrograms.

The goal of the example is to demonstrate how to use MathWorks® tools to approach the problem in fundamentally different but complementary ways. All workflows use `audioDatastore` to manage flow of data from disk and ensure proper randomization.

All approaches used in this example performed equally well on the test set. This example is not intended as a direct comparison between the various approaches. For example, you can also use Bayesian optimization for hyperparameter selection in the CNN. An additional strategy that is useful in deep learning with small training sets like this version of the FSDD is to use data augmentation. How manipulations affect class is not always known, so data augmentation is not always feasible. However, for speech, established data augmentation strategies are available through `audioDataAugmenter`.

In the case of wavelet time scattering, there are also a number of modifications you can try. For example, you can change the invariant scale of the transform, vary the number of wavelet filters per filter bank, and try different classifiers.

Appendix: Helper Functions

```

function Labels = helpergenLabels(ads)
% This function is only for use in Wavelet Toolbox examples. It may be
% changed or removed in a future release.
tmp = cell(numel(ads.Files),1);
expression = "[0-9]+_";
for nf = 1:numel(ads.Files)
    idx = regexp(ads.Files{nf},expression);
    tmp{nf} = ads.Files{nf}(idx);
end
Labels = categorical(tmp);
end

function x = helperReadSPData(x)
% This function is only for use Wavelet Toolbox examples. It may change or
% be removed in a future release.

N = numel(x);
if N > 8192
    x = x(1:8192);
elseif N < 8192
    pad = 8192-N;
    prepad = floor(pad/2);
    postpad = ceil(pad/2);
    x = [zeros(prepad,1) ; x ; zeros(postpad,1)];
end
x = x./max(abs(x));

end

function x = helperBayesOptLSTM(X_train, Y_train, X_val, Y_val)
% This function is only for use in the
% "Spoken Digit Recognition with Wavelet Scattering and Deep Learning"
% example. It may change or be removed in a future release.
x = @valErrorFun;

function [valError,cons, fileName] = valErrorFun(optVars)
%% LSTM Architecture
[inputSize,~] = size(X_train{1});
numClasses = numel(unique(Y_train));

layers = [ ...
    sequenceInputLayer(inputSize)
    bilstmLayer(optVars.NumHiddenUnits,'OutputMode','last') % Using number of hidden layers
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];

% Plots not displayed during training
options = trainingOptions('adam', ...
    'InitialLearnRate',optVars.InitialLearnRate, ... % Using initial learning rate value
    'MaxEpochs',300, ...
    'MiniBatchSize',30, ...
    'SequenceLength','shortest', ...
    'Shuffle','never', ...
    'Verbose', false);

```

```
%% Train the network
net = trainNetwork(X_train, Y_train, layers, options);
%% Training accuracy
X_val_P = net.classify(X_val);
accuracy_training = sum(X_val_P == Y_val)./numel(Y_val);
valError = 1 - accuracy_training;
%% save results of network and options in a MAT file in the results folder along with the
fileName = fullfile('results', num2str(valError) + ".mat");
save(fileName, 'net', 'valError', 'options')
cons = [];
end % end for inner function
end % end for outer function

function X = helperspeechSpectrograms(ads, segmentDuration, frameDuration, hopDuration, numBands)
% This function is only for use in the
% "Spoken Digit Recognition with Wavelet Scattering and Deep Learning"
% example. It may change or be removed in a future release.
%
% helperspeechSpectrograms(ads, segmentDuration, frameDuration, hopDuration, numBands)
% computes speech spectrograms for the files in the datastore ads.
% segmentDuration is the total duration of the speech clips (in seconds),
% frameDuration the duration of each spectrogram frame, hopDuration the
% time shift between each spectrogram frame, and numBands the number of
% frequency bands.
disp("Computing speech spectrograms...");

numHops = ceil((segmentDuration - frameDuration)/hopDuration);
numFiles = length(ads.Files);
X = zeros([numBands, numHops, 1, numFiles], 'single');

for i = 1:numFiles

    [x, info] = read(ads);
    x = normalizeAndResize(x);
    fs = info.SampleRate;
    frameLength = round(frameDuration*fs);
    hopLength = round(hopDuration*fs);

    spec = melSpectrogram(x, fs, ...
        'Window', hamming(frameLength, 'periodic'), ...
        'OverlapLength', frameLength - hopLength, ...
        'FFTLength', 2048, ...
        'NumBands', numBands, ...
        'FrequencyRange', [50, 4000]);

    % If the spectrogram is less wide than numHops, then put spectrogram in
    % the middle of X.
    w = size(spec, 2);
    left = floor((numHops-w)/2)+1;
    ind = left:left+w-1;
    X(:, ind, 1, i) = spec;

    if mod(i, 500) == 0
        disp("Processed " + i + " files out of " + numFiles)
    end

end

end
```



```
disp('...done');

end

%-----
function x = normalizeAndResize(x)
% This function is only for use in the
% "Spoken Digit Recognition with Wavelet Scattering and Deep Learning"
% example. It may change or be removed in a future release.

N = numel(x);
if N > 8192
    x = x(1:8192);
elseif N < 8192
    pad = 8192-N;
    prepad = floor(pad/2);
    postpad = ceil(pad/2);
    x = [zeros(prepad,1) ; x ; zeros(postpad,1)];
end
x = x./max(abs(x));
end
```

Copyright 2018, The MathWorks, Inc.

Active Noise Control with Simulink Real-Time

Design a real-time active noise control system using a Speedgoat Simulink Real-Time target.

Active Noise Control (ANC)

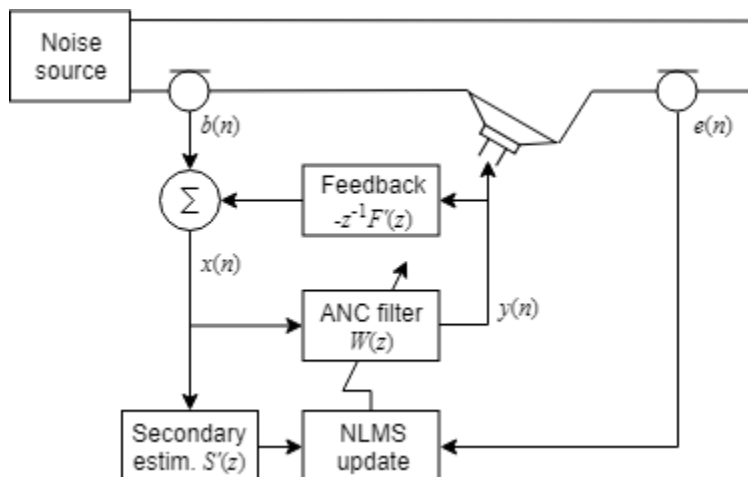
The goal of active noise control is to reduce unwanted sound by producing an “anti-noise” signal that cancels the undesired sound wave. This principle has been applied successfully to a wide variety of applications, such as noise-cancelling headphones, active sound design in car interiors, and noise reduction in ventilation conduits and ventilated enclosures.

In this example, we apply the principles of model-based design. First, we design the ANC without any hardware by using a simple acoustic model in our simulation. Then, we complete our prototype by replacing the simulated acoustic path by the “Speedgoat Target Computers and Speedgoat Support” (Simulink Real-Time) and its IO104 analog module. The Speedgoat is an external Real-Time target for Simulink, which allows us to execute our model in real time and observe any data of interest, such as the adaptive filter coefficients, in real time.

This example has a companion video: Active Noise Control – From Modeling to Real-Time Prototyping.

ANC Feedforward Model

The following figure illustrates a classic example of *feedforward* ANC. A noise source at the entrance of a duct, such as a fan, is “cancelled” by a loudspeaker. The noise source $b(n)$ is measured with a reference microphone, and the signal present at the output of the system is monitored with an error microphone, $e(n)$. Note that the smaller the distance between the reference microphone and the loudspeaker, the faster the ANC must be able to compute and play back the “anti-noise”.



The primary path is the transfer function between the two microphones, $W(z)$ is the adaptive filter computed from the last available error signal $e(n)$, and the secondary path $S(z)$ is the transfer function between the ANC output and the error microphone. The secondary path estimate $S'(z)$ is used to filter the input of the NLMS update function. Also, the acoustic feedback $F(z)$ from the ANC loudspeaker to the reference microphone can be estimated ($F'(z)$) and removed from the reference signal $b(n)$.

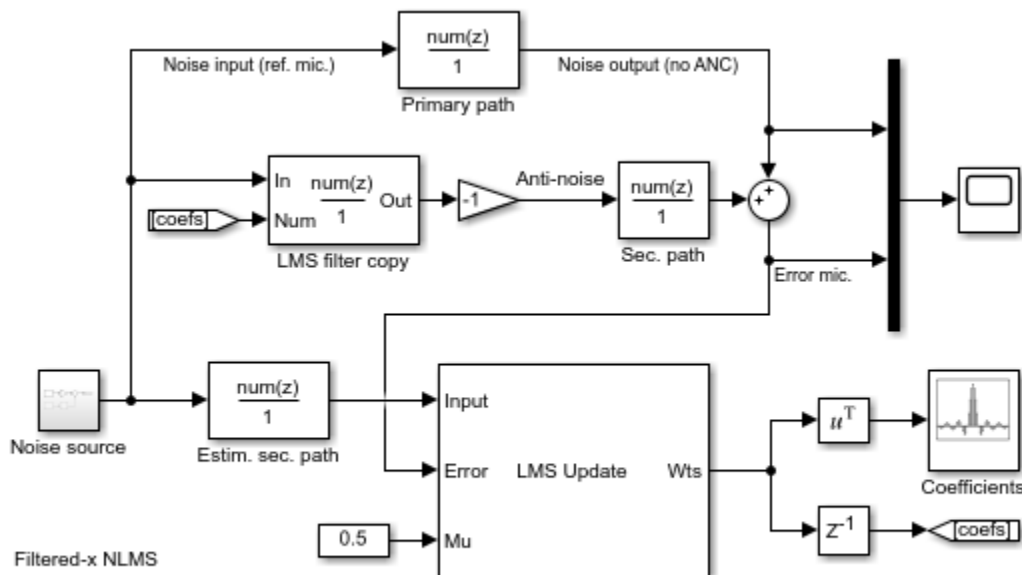
To implement a successful ANC system, we must estimate both the primary and the secondary paths. In this example, we estimate the secondary path and the acoustic feedback first and then keep it constant while the ANC system adapts the primary path.

Filtered-X ANC Model

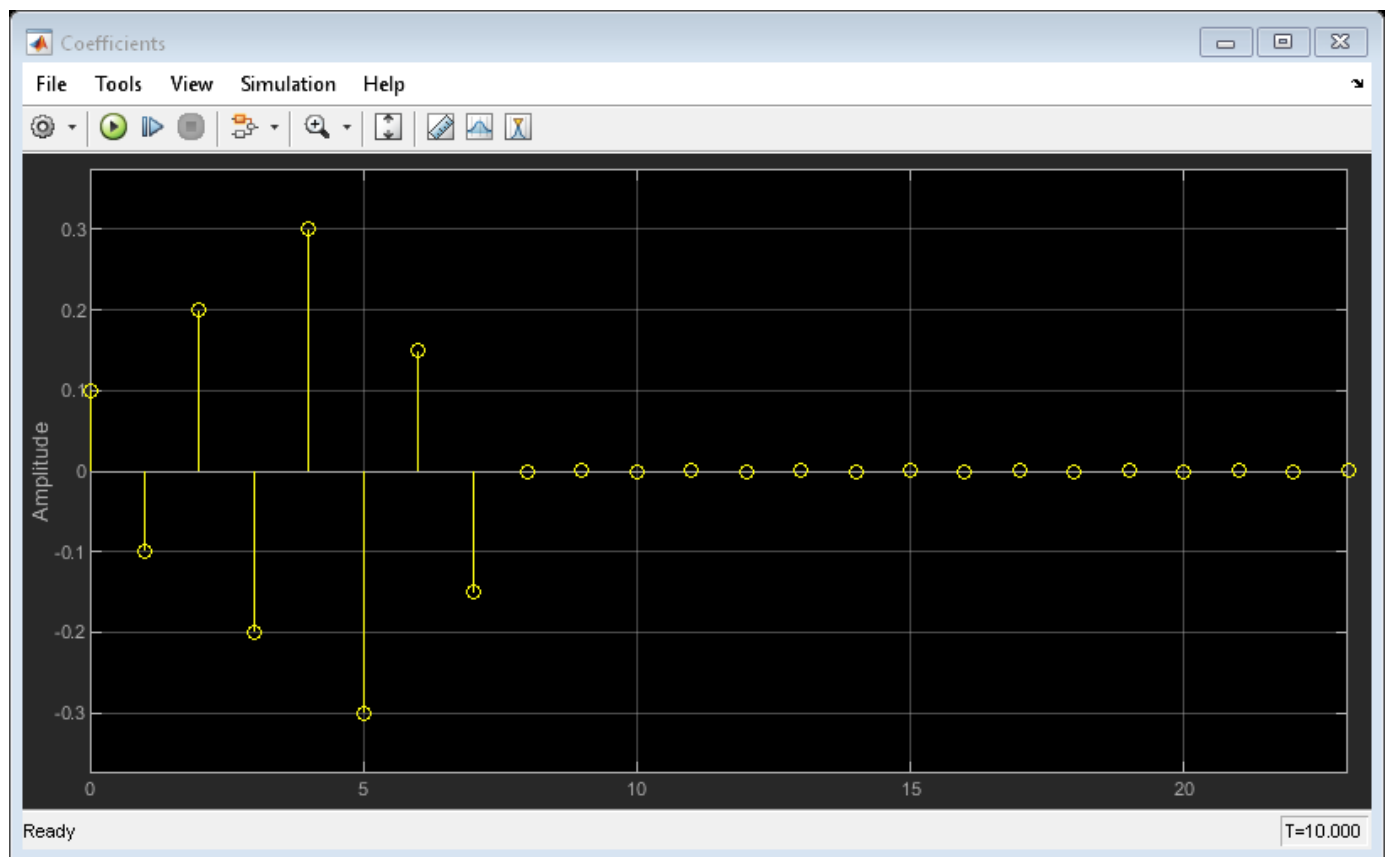
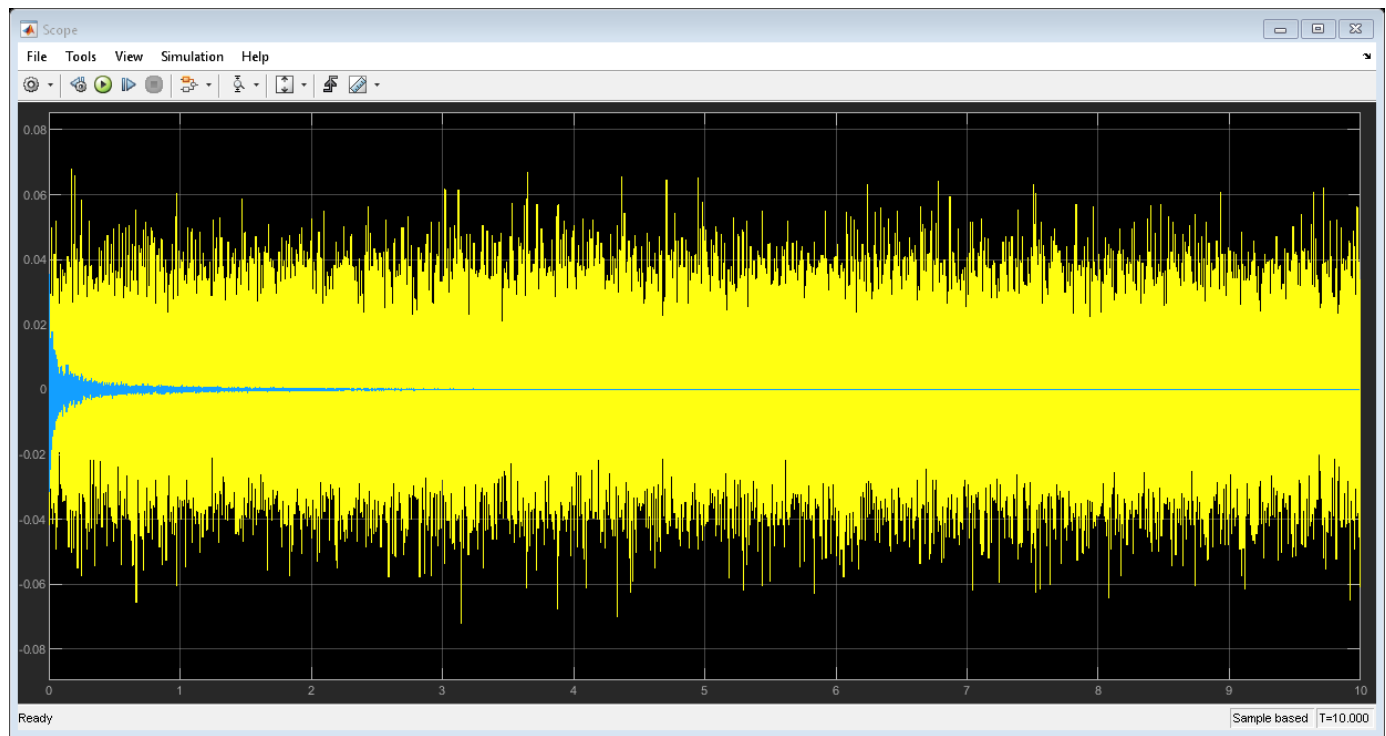
With Simulink and model-based design, you can start with a basic model of the desired system and a simulated environment. Then, you can improve the realism of that model or replace the simulated environment by the real one. You can also iterate by refining your simulated environment when you learn more about the challenges of the real-world system. For example, you could add acoustic feedback or measurement noise to the simulated environment if those are elements that limit the performance of the real-world system.

Start with a model of a Filtered-X NLMS ANC system, including both the ANC controller and the duct's acoustic environment. Assume that we already have an estimate of the secondary path, since we will design a system to measure that later. Simulate the signal at the error microphone as the sum of the noise source filtered by the primary acoustic path and the ANC output filtered by the secondary acoustic path. Use an "LMS Update" block in a configuration that minimizes the signal captured by the error microphone. In a Filtered-X system, the NLMS update's input is the noise source filtered by the estimate of the secondary path. To avoid an algebraic loop, there is a delay of one sample between the computation of the new filter coefficients and their use by the LMS filter.

Set the secondary path to $s(n) = [0.5 \ 0.5 \ -0.3 \ -0.3 \ -0.2 \ -0.2]$ and the primary path to $\text{conv}(s(n), f(n))$, where $f(n) = [0.1 \ -0.1 \ 0.2 \ -0.2 \ 0.3 \ -0.3 \ 0.15 \ -0.15]$. Verify that the adaptive filter properly converges to $f(n)$, in which case it matches the primary path in our model once convolved with the secondary path. Note that $s(n)$ and $f(n)$ were set arbitrarily, but we could try any FIR transfer functions, such as an actual impulse response measurement.

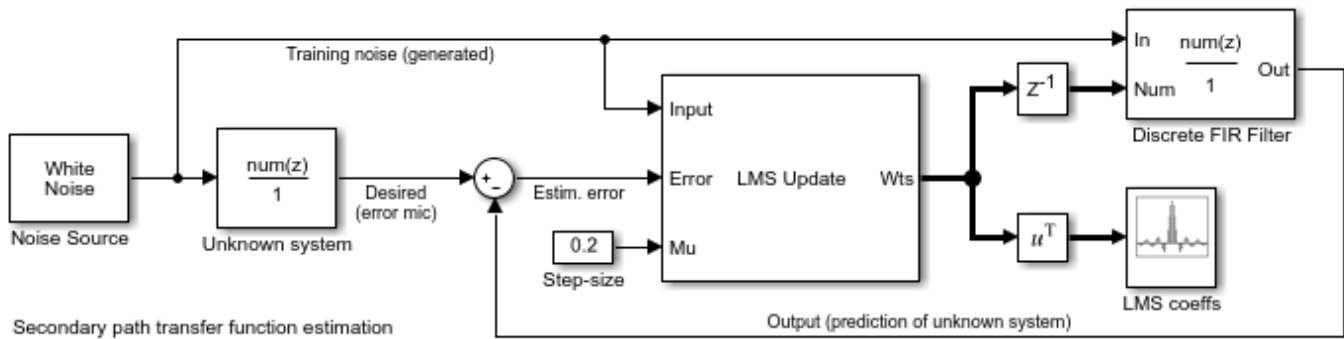


Copyright 2019 The MathWorks, Inc.

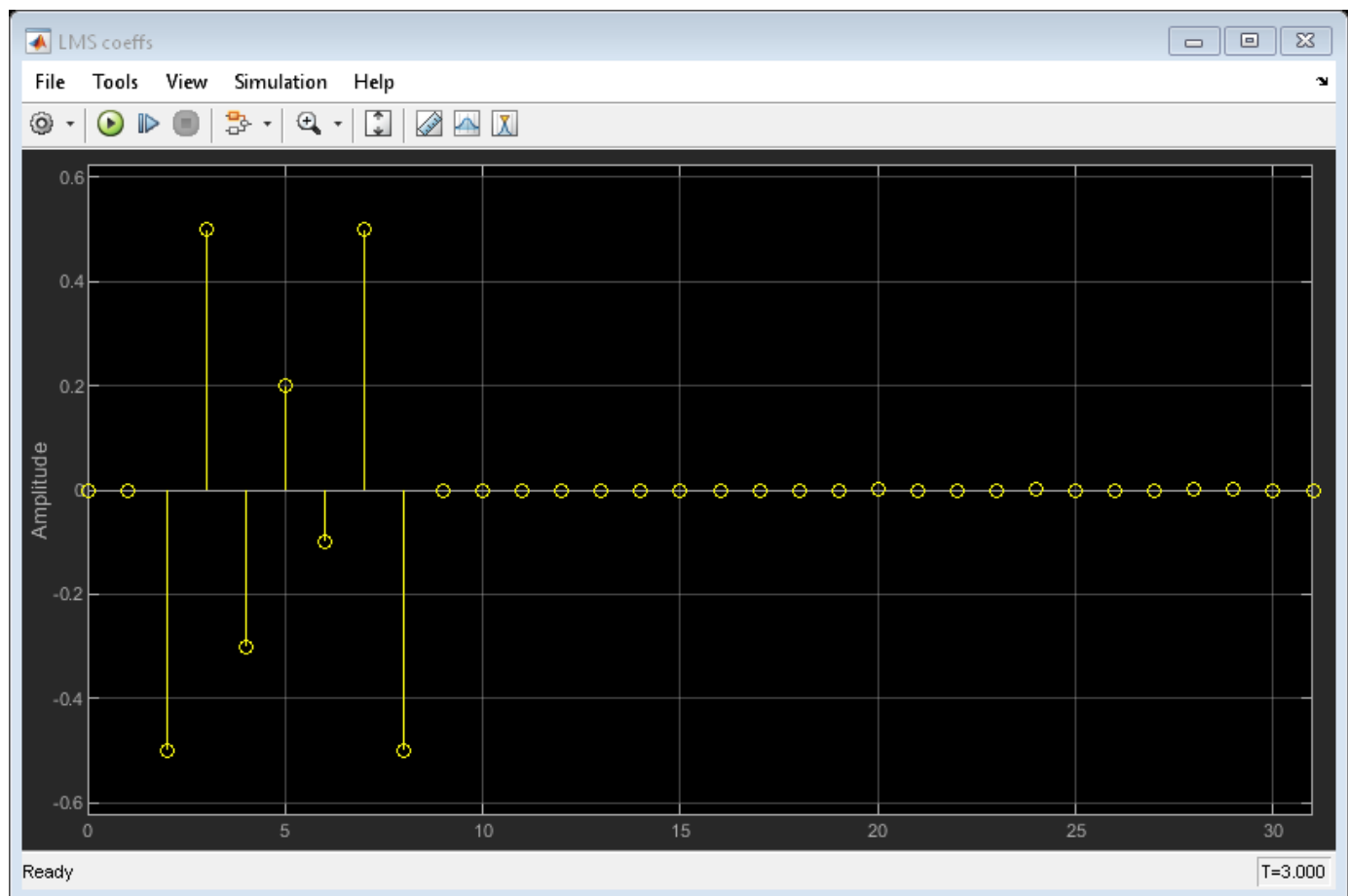


Secondary Path Estimation Model

Design a model to estimate the secondary path. Use an adaptive filter in a configuration appropriate for the identification of an unknown system. We can then verify that it converges to $f(n)$.

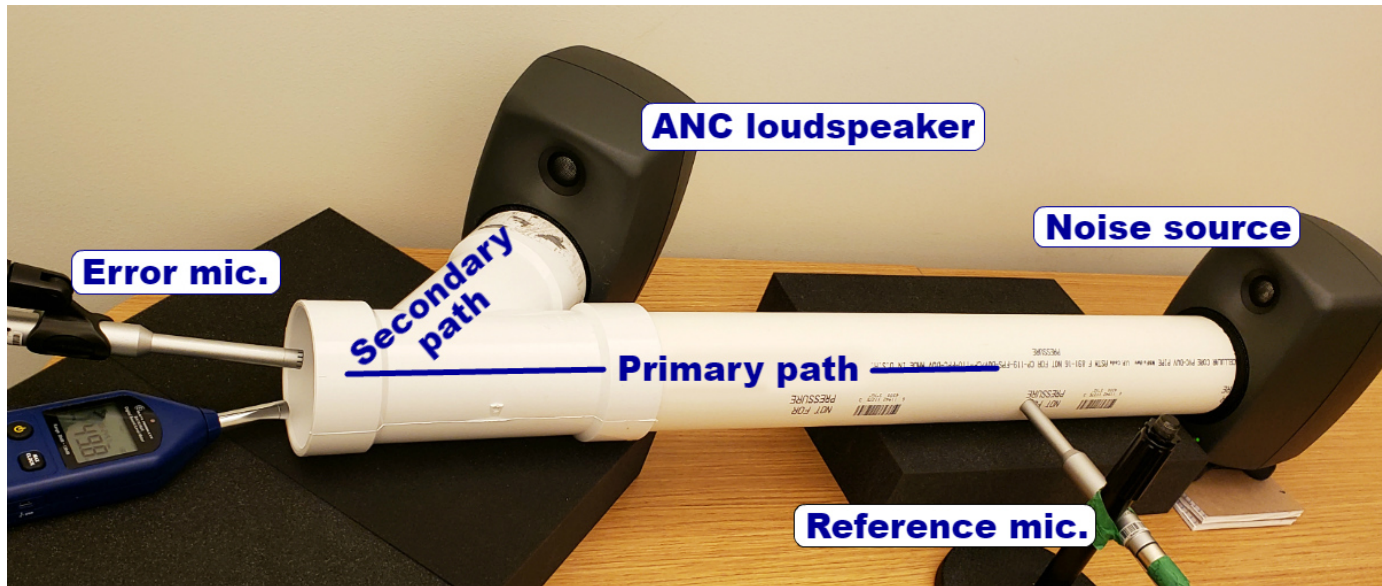


Copyright 2019 The MathWorks, Inc.



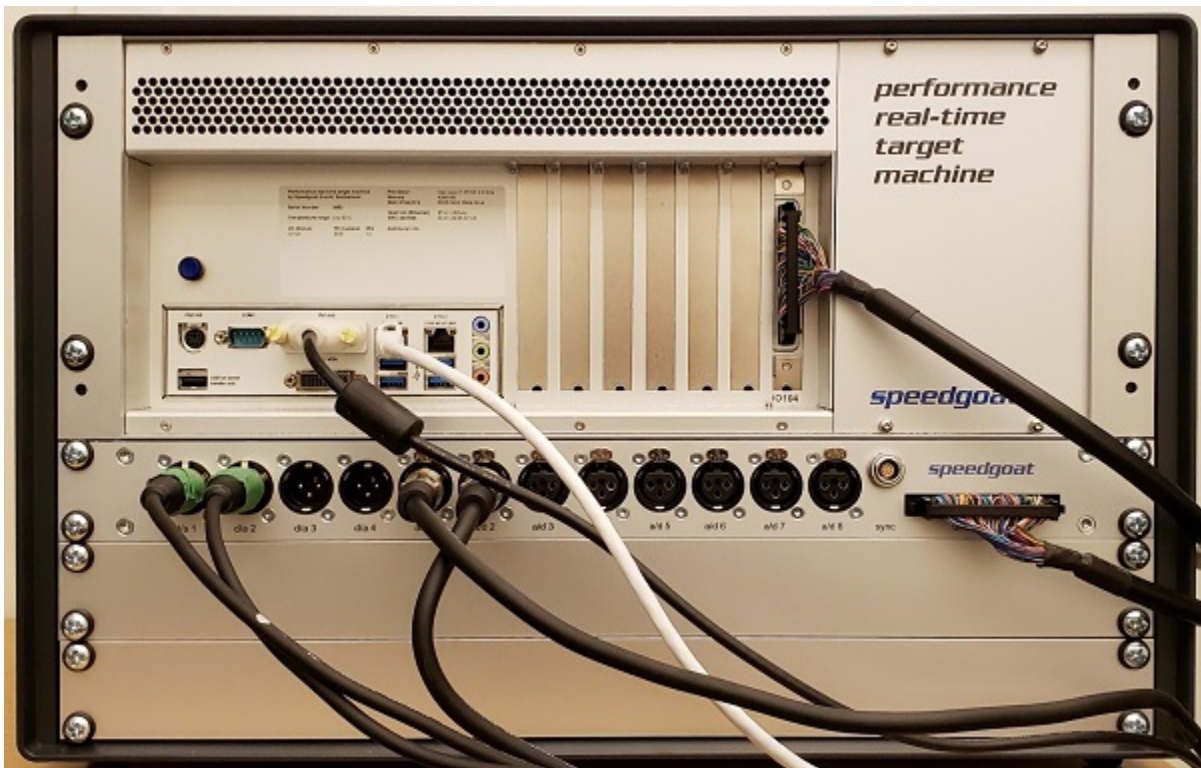
Real-Time Implementation with Speedgoat

To experiment with ANC in a real-time environment, we built the classic duct example. In the following image, from right to left, we have a loudspeaker playing the noise source, the reference microphone, the ANC loudspeaker, and the error microphone.

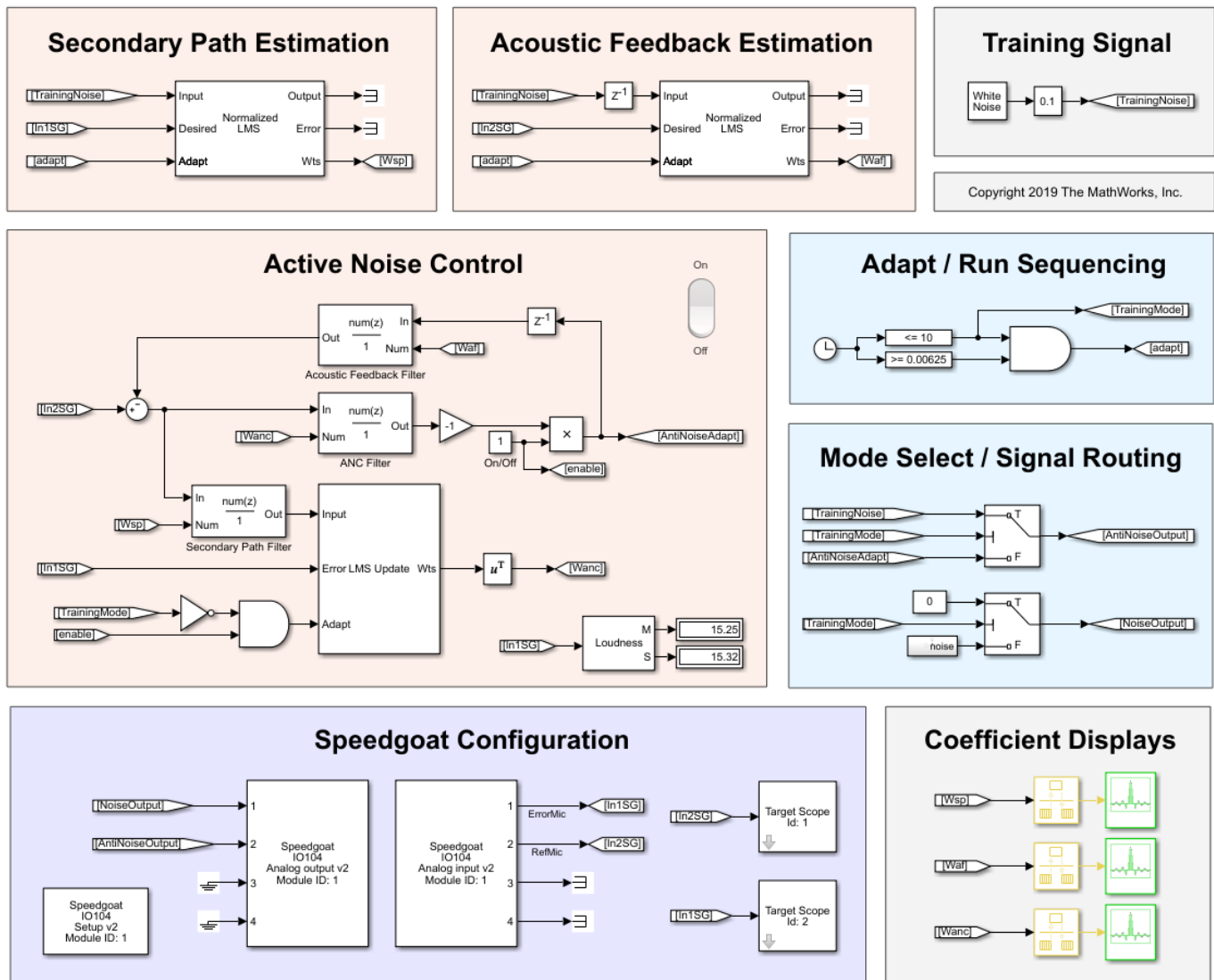


Latency is critical: the system must record the reference microphone, compute the response and play it back on the ANC loudspeaker in the time it takes for sound to travel between these points. In this example, the distance between the reference microphone and the beginning of the “Y” section is 34 cm. The speed of sound is 343 m/s, thus our maximum latency is 1 ms, or 8 samples at the 8 kHz sampling rate used in this example.

We will be using the Speedgoat real-time target in Simulink, with the IO104 analog I/O interface card. The Speedgoat allows us to achieve a latency as low as one or two samples.

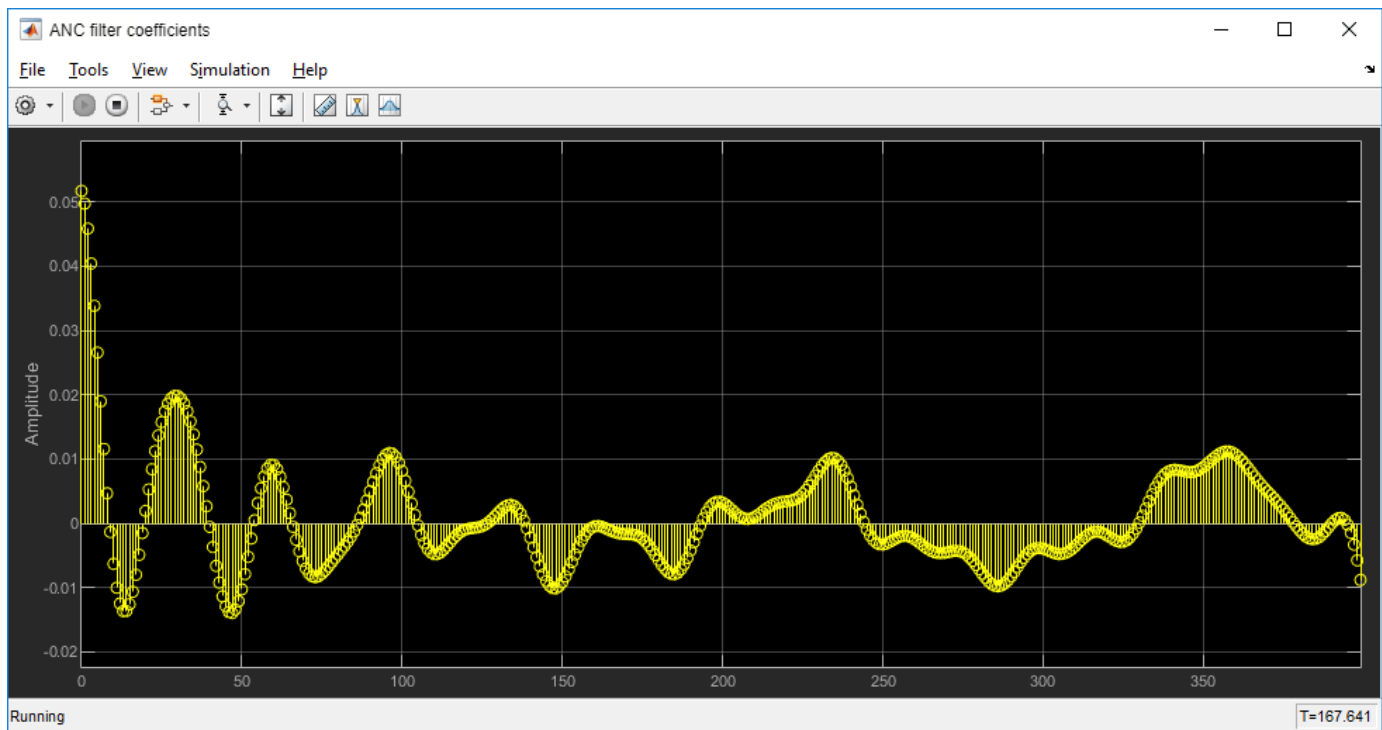


To realize our real-time model, we use the building blocks that we tested earlier, and simply replace the acoustic models by the Speedgoat I/O blocks. We also included the measurement of the acoustic feedback from the ANC loudspeaker to the reference microphone, and we added some logic to automatically measure the secondary path for 10 seconds before switching to the actual ANC mode. During the first 10 seconds, white noise is played back on the ANC loudspeaker and two NLMS filters are enabled, one per microphone. Then, a “noise source” is played back by the model for convenience, but the actual input of the ANC system is the reference microphone (this playback could be replaced by a real noise source, such as a fan at the right end of the duct). The system records the reference microphone, adapts the ANC NLMS filter and computes a signal for the ANC loudspeaker. We take care to set up our model properties so that the IO104 card is driving the cadence of the Simulink model (see IO104 in interrupt-driven mode). To access the model’s folder, open the example by clicking the “Open Script” button. The model’s file name is “Speedgoat_FXLMS_ANC_model.slx”.



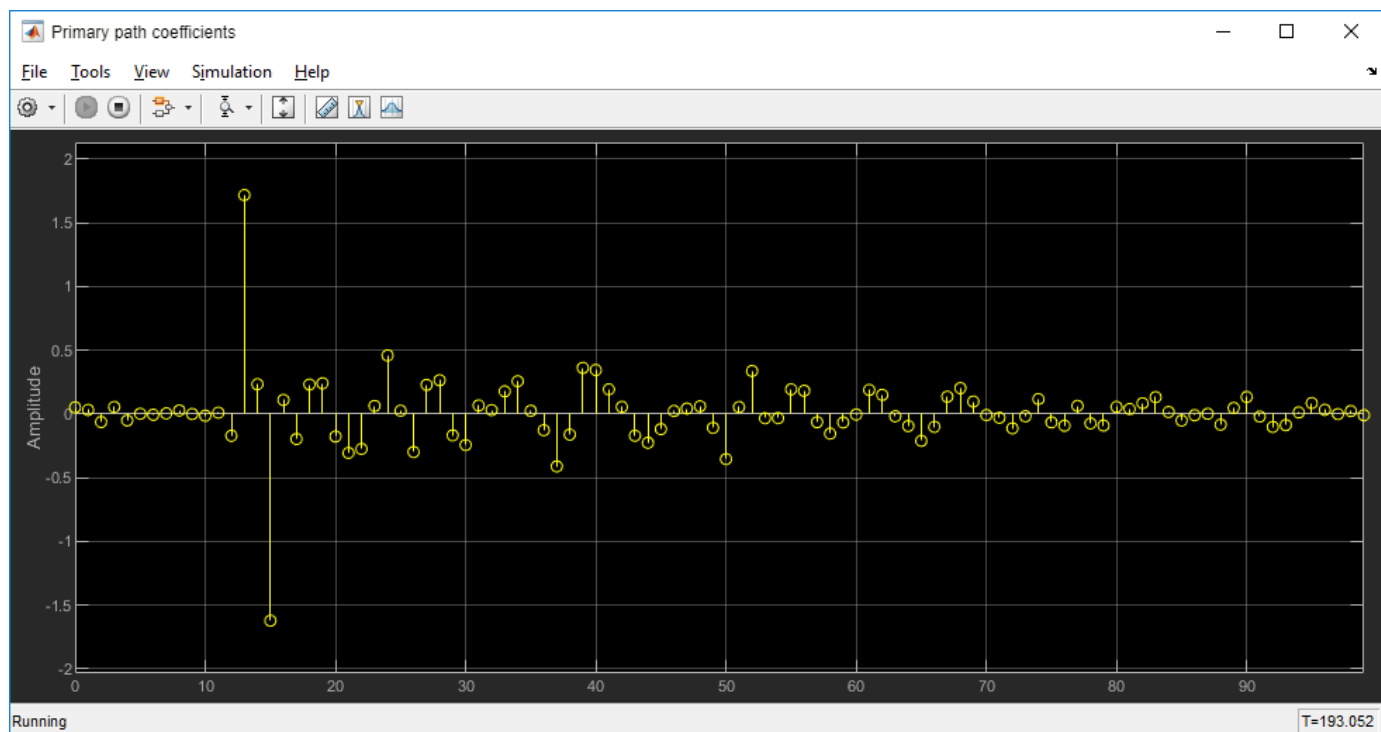
Noise Reduction Performance

We have measured the performance of this ANC prototype with both dual tones and the actual recording of a muffled washing machine. We obtained a noise reduction of 20-30 dB for the dual tones and 8-10 dB for the recording, which is a more realistic but also more difficult case. The convergence rate for the filter is less than a few seconds with tones, but requires much more time for the real case (one or two minutes).

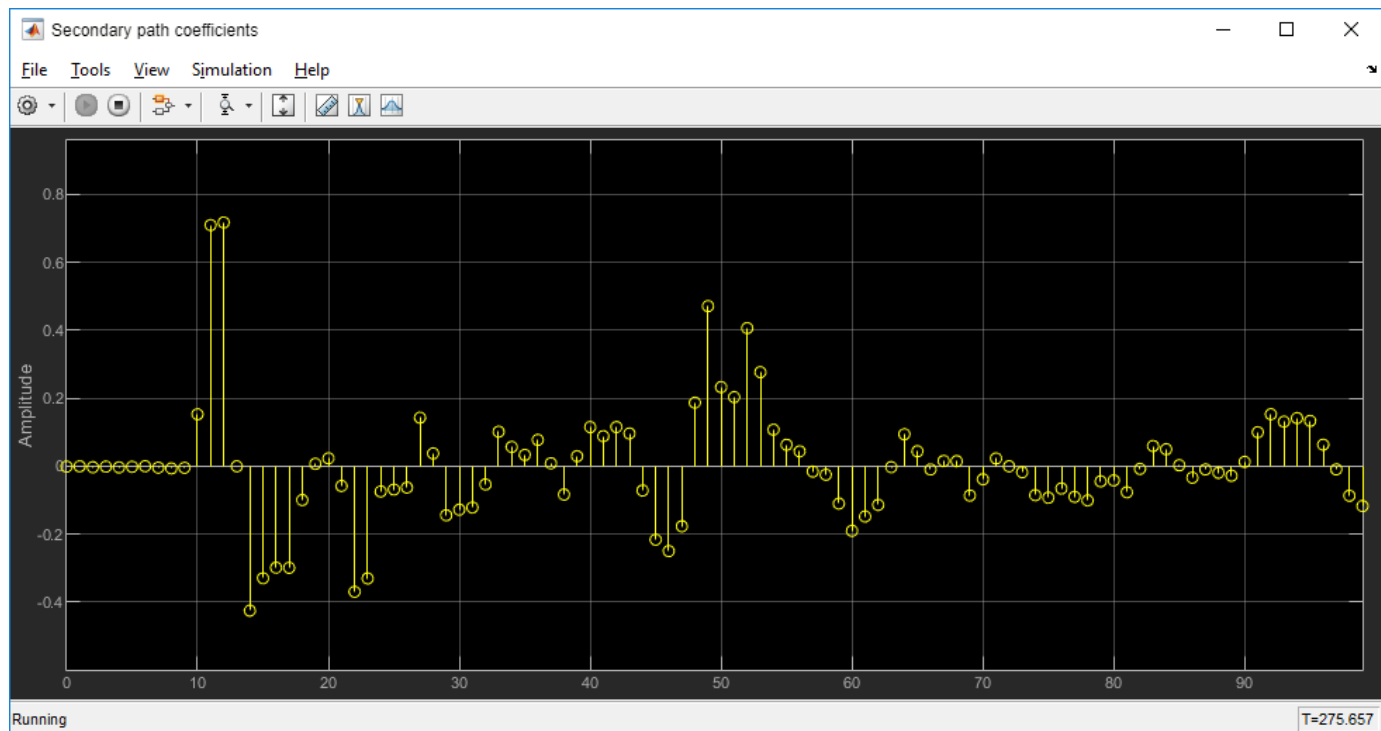


Latency Measurements

Another aspect of performance is the latency of the system, as this determines the minimum distance between the reference microphone and the ANC loudspeaker. In our prototype, the active ANC loudspeaker that we are using may introduce latency, so we can make sure that this is not an issue by comparing the response between the two microphones to the response between the ANC output signal and the error microphone. The difference between these two delays is the maximum time the system has available to compute the anti-noise signal from the reference microphone. Using the same NLMS identification technique, we obtain the following response from the reference microphone to the error microphone:



Then, we may compare that response to the secondary path estimation:



The difference is only 2 or 3 samples, so using our current active loudspeaker and the Speedgoat, we cannot significantly reduce the distance between the reference microphone and the ANC loudspeaker in our prototype. To reduce the distance, we would need a loudspeaker that does not introduce any

extra latency. We could also increase the sampling rate of the Simulink model (the Speedgoat latency is set to 1 or 2 samples, regardless of the sample rate).

References

S. M. Kuo and D. R. Morgan, "Active noise control: a tutorial review," in *Proceedings of the IEEE*, vol. 87, no. 6, pp. 943-973, June 1999.

K.-C. Chen, C.-Y. Chang, and S. M. Kuo, "Active noise control in a duct to cancel broadband noise," in *IOP Conference Series: Materials Science and Engineering*, vol. 237, no. 1, 2017.

"Speedgoat Target Computers and Speedgoat Support" (Simulink Real-Time)

Setting up the IO104 module in Simulink

Setting up the IO104 in interrupt-driven mode

See also: Active Noise Control Using a Filtered-X LMS FIR Adaptive Filter

Acoustic Scene Recognition Using Late Fusion

This example shows how to create a multi-model late fusion system for acoustic scene recognition. The example trains a convolutional neural network (CNN) using mel spectrograms and an ensemble classifier using wavelet scattering. The example uses the TUT dataset for training and evaluation [1].

Introduction

Acoustic scene classification (ASC) is the task of classifying environments from the sounds they produce. ASC is a generic classification problem that is foundational for context awareness in devices, robots, and many other applications [1]. Early attempts at ASC used mel-frequency cepstral coefficients (mfcc) and Gaussian mixture models (GMMs) to describe their statistical distribution. Other popular features used for ASC include zero crossing rate, spectral centroid (`spectralCentroid`), spectral rolloff (`spectralRolloffPoint`), spectral flux (`spectralFlux`), and linear prediction coefficients (`lpc`) [5]. Hidden Markov models (HMMs) were trained to describe the temporal evolution of the GMMs. More recently, the best performing systems have used deep learning, usually CNNs, and a fusion of multiple models. The most popular feature for top-ranked systems in the DCASE 2017 contest was the mel spectrogram (`melSpectrogram`). The top-ranked systems in the challenge used late fusion and data augmentation to help their systems generalize.

To illustrate a simple approach that produces reasonable results, this example trains a CNN using mel spectrograms and an ensemble classifier using wavelet scattering. The CNN and ensemble classifier produce roughly equivalent overall accuracy, but perform better at distinguishing different acoustic scenes. To increase overall accuracy, you merge the CNN and ensemble classifier results using late fusion.

Load Acoustic Scene Recognition Data Set

To run the example, you must first download the data set [1]. The full data set is approximately 15.5 GB. Depending on your machine and internet connection, downloading the data can take about 4 hours.

```
downloadFolder = tempdir;  
datasetFolder = fullfile(downloadFolder, 'TUT-acoustic-scenes-2017');
```

```
if ~exist(datasetFolder, 'dir')  
    disp('Downloading TUT-acoustic-scenes-2017 (15.5 GB)...')  
    HelperDownload_TUT_acoustic_scenes_2017(datasetFolder);  
end
```

Read in the development set metadata as a table. Name the table variables `FileName`, `AcousticScene`, and `SpecificLocation`.

```
metadata_train = readtable(fullfile(datasetFolder, 'TUT-acoustic-scenes-2017-development', 'meta.t'),  
    'Delimiter', {'\t'}, ...  
    'ReadVariableNames', false);  
metadata_train.Properties.VariableNames = {'FileName', 'AcousticScene', 'SpecificLocation'};  
head(metadata_train)
```

```
ans =
```

```
8×3 table
```

FileName	AcousticScene	SpecificLocation
----------	---------------	------------------

```
{'audio/b020_90_100.wav' } {'beach'} {'b020'}
{'audio/b020_110_120.wav'} {'beach'} {'b020'}
{'audio/b020_100_110.wav'} {'beach'} {'b020'}
{'audio/b020_40_50.wav' } {'beach'} {'b020'}
{'audio/b020_50_60.wav' } {'beach'} {'b020'}
{'audio/b020_30_40.wav' } {'beach'} {'b020'}
{'audio/b020_160_170.wav'} {'beach'} {'b020'}
{'audio/b020_170_180.wav'} {'beach'} {'b020'}
```

```
metadata_test = readtable(fullfile(datasetFolder,'TUT-acoustic-scenes-2017-evaluation','meta.txt'),
    'Delimiter',{'\t'}, ...
    'ReadVariableNames',false);
metadata_test.Properties.VariableNames = {'FileName','AcousticScene','SpecificLocation'};
head(metadata_test)
```

```
ans =
```

```
8×3 table
```

FileName	AcousticScene	SpecificLocation
{'audio/1245.wav'}	{'beach'}	{'b174'}
{'audio/1456.wav'}	{'beach'}	{'b174'}
{'audio/1318.wav'}	{'beach'}	{'b174'}
{'audio/967.wav' }	{'beach'}	{'b174'}
{'audio/203.wav' }	{'beach'}	{'b174'}
{'audio/777.wav' }	{'beach'}	{'b174'}
{'audio/231.wav' }	{'beach'}	{'b174'}
{'audio/768.wav' }	{'beach'}	{'b174'}

Note that the specific recording locations in the test set do not intersect with the specific recording locations in the development set. This makes it easier to validate that the trained models can generalize to real-world scenarios.

```
sharedRecordingLocations = intersect(metadata_test.SpecificLocation,metadata_train.SpecificLocation);
fprintf('Number of specific recording locations in both train and test sets = %d\n',numel(sharedRecordingLocations));
```

```
Number of specific recording locations in both train and test sets = 0
```

The first variable of the metadata tables contains the file names. Concatenate the file names with the file paths.

```
train_filePaths = fullfile(datasetFolder,'TUT-acoustic-scenes-2017-development',metadata_train.FileName);
```

```
test_filePaths = fullfile(datasetFolder,'TUT-acoustic-scenes-2017-evaluation',metadata_test.FileName);
```

Create audio datastores for the train and test sets. Set the `Labels` property of the `audioDatastore` to the acoustic scene. Call `countEachLabel` to verify an even distribution of labels in both the train and test sets.

```
adsTrain = audioDatastore(train_filePaths, ...
    'Labels',categorical(metadata_train.AcousticScene), ...
    'IncludeSubfolders',true);
```

```
display(countEachLabel(adsTrain))

adsTest = audioDatastore(test_filePaths, ...
    'Labels',categorical(metadata_test.AcousticScene), ...
    'IncludeSubfolders',true);
display(countEachLabel(adsTest))
```

15×2 table

Label	Count
beach	312
bus	312
cafe/restaurant	312
car	312
city_center	312
forest_path	312
grocery_store	312
home	312
library	312
metro_station	312
office	312
park	312
residential_area	312
train	312
tram	312

15×2 table

Label	Count
beach	108
bus	108
cafe/restaurant	108
car	108
city_center	108
forest_path	108
grocery_store	108
home	108
library	108
metro_station	108
office	108
park	108
residential_area	108
train	108
tram	108

You can reduce the data set used in this example to speed up the run time at the cost of performance. In general, reducing the data set is a good practice for development and debugging. Set `reduceDataset` to `true` to reduce the data set.

```
reduceDataset = false;
if reduceDataset
    adsTrain = splitEachLabel(adsTrain,20);
```

```

        adsTest = splitEachLabel(adsTest,10);
    end

```

Call `read` to get the data and sample rate of a file from the train set. Audio in the database has consistent sample rate and duration. Normalize the audio and listen to it. Display the corresponding label.

```

[data,adsInfo] = read(adsTrain);
data = data./max(data,[], 'all');

fs = adsInfo.SampleRate;
sound(data,fs)

fprintf('Acoustic scene = %s\n',adsTrain.Labels(1))
Acoustic scene = beach

```

Call `reset` to return the datastore to its initial condition.

```
reset(adsTrain)
```

Feature Extraction for CNN

Each audio clip in the dataset consists of 10 seconds of stereo (left-right) audio. The feature extraction pipeline and the CNN architecture in this example are based on [3]. Hyperparameters for the feature extraction, the CNN architecture, and the training options were modified from the original paper using a systematic hyperparameter optimization workflow.

First, convert the audio to mid-side encoding. [3] suggests that mid-side encoded data provides better spatial information that the CNN can use to identify moving sources (such as a train moving across an acoustic scene).

```
dataMidSide = [sum(data,2),data(:,1)-data(:,2)];
```

Divide the signal into one-second segments with overlap. The final system uses a probability-weighted average on the one-second segments to predict the scene for each 10-second audio clip in the test set. Dividing the audio clips into one-second segments makes the network easier to train and helps prevent overfitting to specific acoustic events in the training set. The overlap helps to ensure all combinations of features relative to one another are captured by the training data. It also provides the system with additional data that can be mixed uniquely during augmentation.

```
segmentLength = 1;
segmentOverlap = 0.5;
```

```

[dataBufferedMid,~] = buffer(dataMidSide(:,1),round(segmentLength*fs),round(segmentOverlap*fs), 'l');
[dataBufferedSide,~] = buffer(dataMidSide(:,2),round(segmentLength*fs),round(segmentOverlap*fs), 'l');
dataBuffered = zeros(size(dataBufferedMid,1),size(dataBufferedMid,2)+size(dataBufferedSide,2));
dataBuffered(:,1:2:end) = dataBufferedMid;
dataBuffered(:,2:2:end) = dataBufferedSide;

```

Use `melSpectrogram` to transform the data into a compact frequency-domain representation. Define parameters for the mel spectrogram as suggested by [3].

```

windowLength = 2048;
samplesPerHop = 1024;
samplesOverlap = windowLength - samplesPerHop;
fftLength = 2*windowLength;
numBands = 128;

```

`melSpectrogram` operates along channels independently. To optimize processing time, call `melSpectrogram` with the entire buffered signal.

```
spec = melSpectrogram(dataBuffered,fs, ...  
    'Window',hamming(windowLength,'periodic'), ...  
    'OverlapLength',samplesOverlap, ...  
    'FFTLength',fftLength, ...  
    'NumBands',numBands);
```

Convert the mel spectrogram into the logarithmic scale.

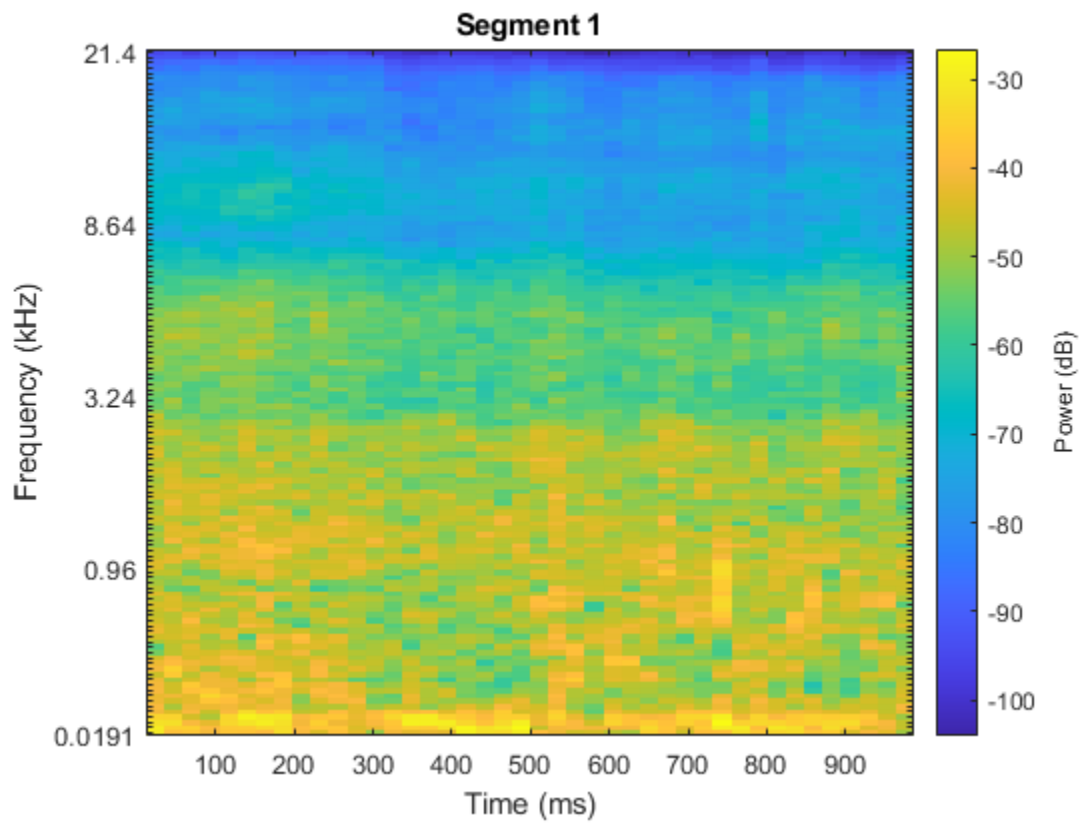
```
spec = log10(spec+eps);
```

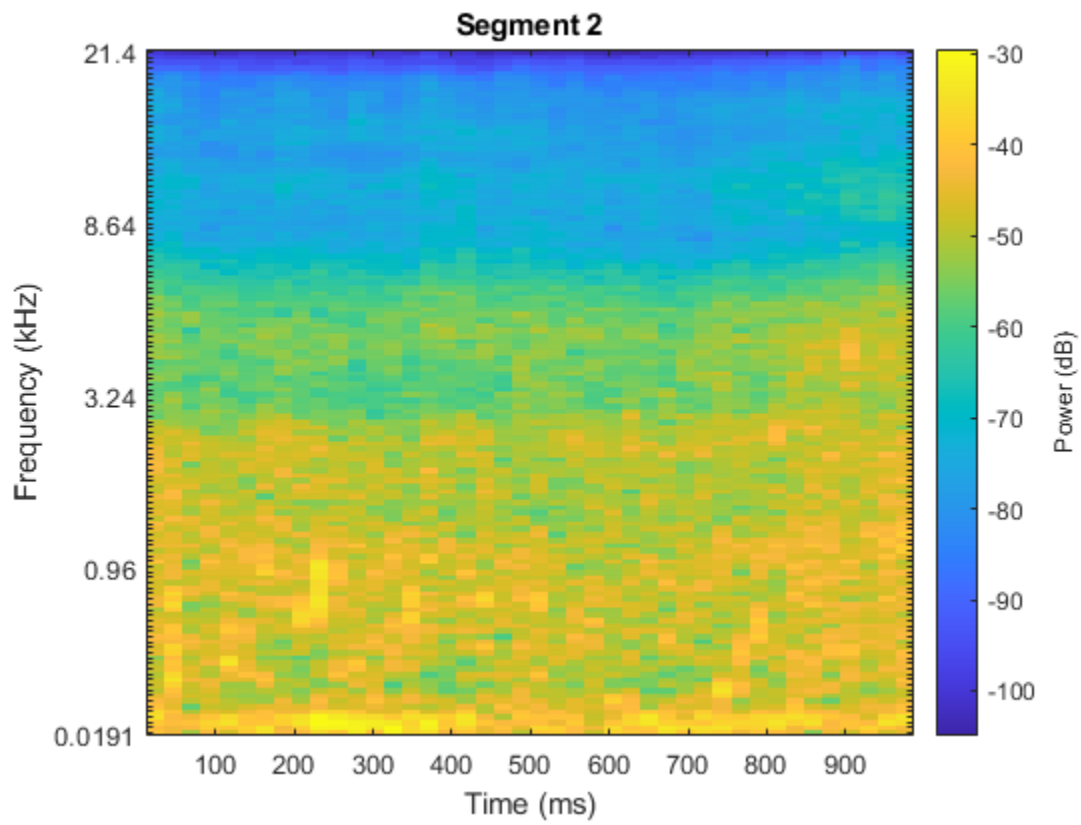
Reshape the array to dimensions (Number of bands)-by-(Number of hops)-by-(Number of channels)-by-(Number of segments). When you feed an image into a neural network, the first two dimensions are the height and width of the image, the third dimension is the channels, and the fourth dimension separates the individual images.

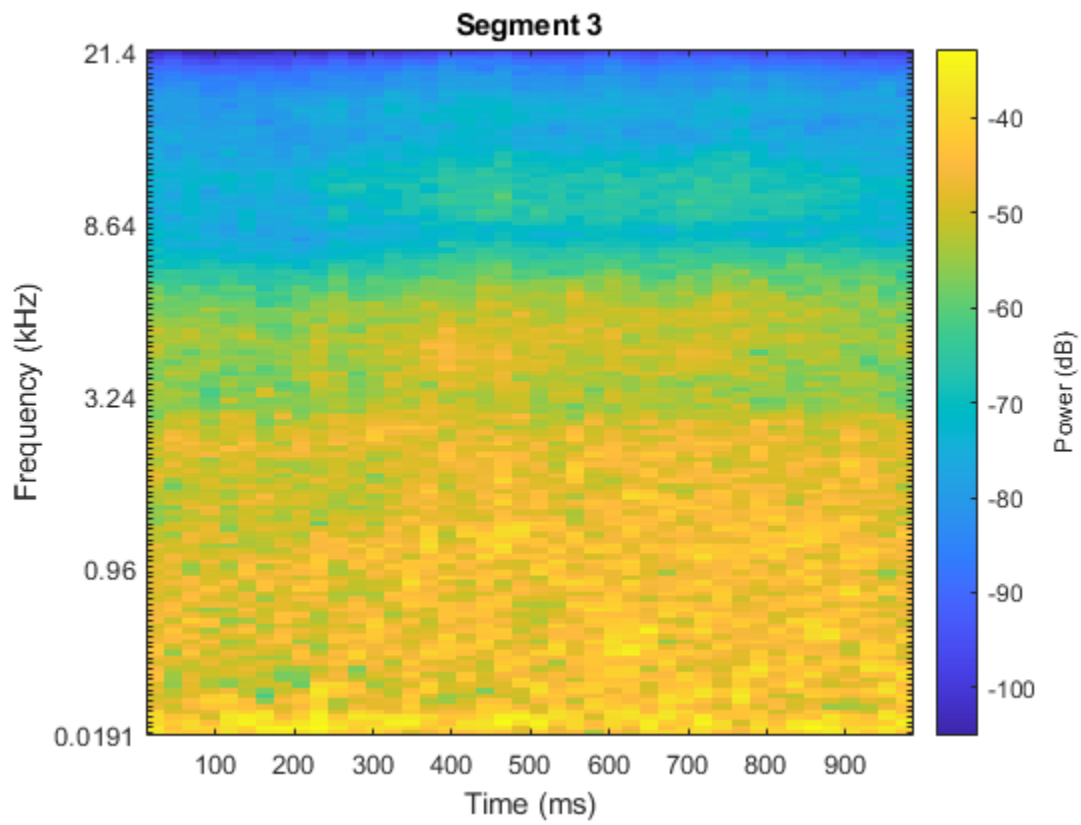
```
X = reshape(spec,size(spec,1),size(spec,2),size(data,2),[]);
```

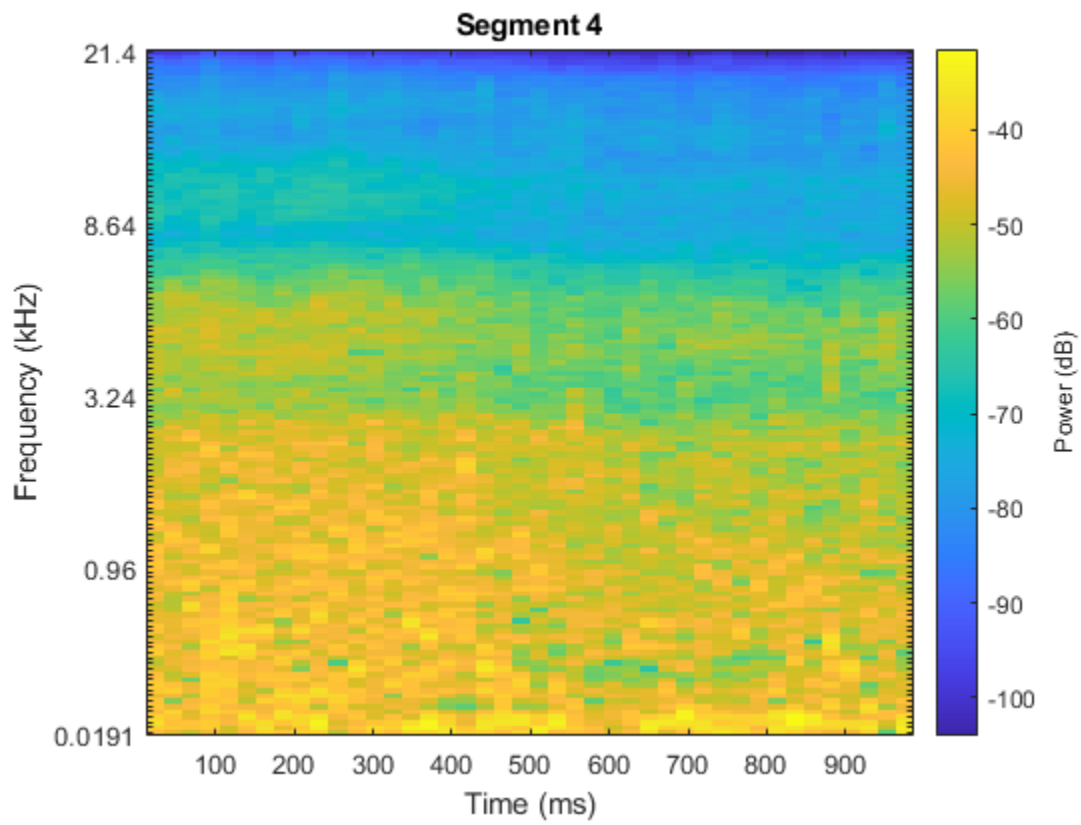
Call `melSpectrogram` without output arguments to plot the mel spectrogram of the mid channel for the first six of the one-second increments.

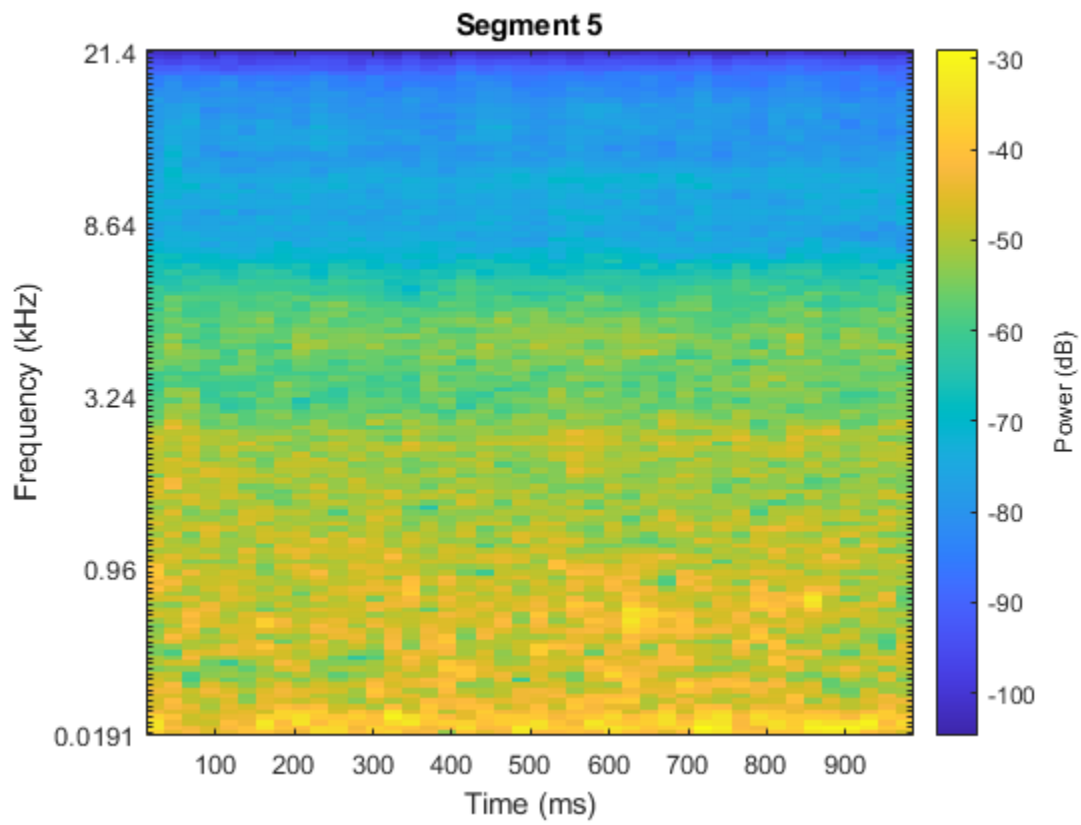
```
for channel = 1:2:11  
    figure  
    melSpectrogram(dataBuffered(:,channel),fs, ...  
        'Window',hamming(windowLength,'periodic'), ...  
        'OverlapLength',samplesOverlap, ...  
        'FFTLength',fftLength, ...  
        'NumBands',numBands);  
    title(sprintf('Segment %d',ceil(channel/2)))  
end
```

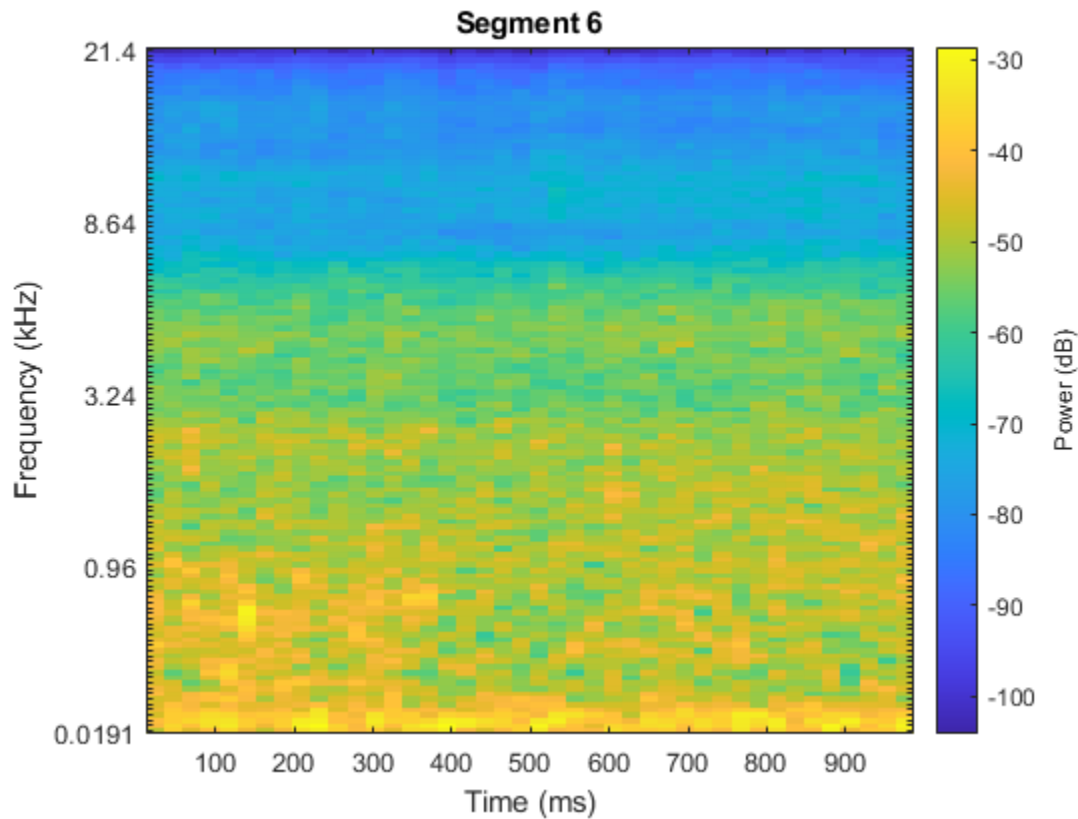













The helper function `HelperSegmentedMelSpectrograms` performs the feature extraction steps outlined above.

To speed up processing, extract mel spectrograms of all audio files in the datastore using `tall` arrays. Unlike in-memory arrays, tall arrays remain unevaluated until you request that the calculations be performed using the `gather` function. This deferred evaluation enables you to work quickly with large data sets. When you eventually request the output using `gather`, MATLAB combines the queued calculations where possible and takes the minimum number of passes through the data. If you have Parallel Computing Toolbox™, you can use tall arrays in your local MATLAB session, or on a local parallel pool. You can also run tall array calculations on a cluster if you have MATLAB® Parallel Server™ installed.

If you do not have Parallel Computing Toolbox™, the code in this example still runs.

```
pp = parpool('IdleTimeout',inf);

train_set_tall = tall(adsTrain);
xTrain = cellfun(@(x)HelperSegmentedMelSpectrograms(x,fs, ...
    'SegmentLength',segmentLength, ...
    'SegmentOverlap',segmentOverlap, ...
    'WindowLength',windowLength, ...
    'HopLength',samplesPerHop, ...
    'NumBands',numBands, ...
    'FFTLength',fftLength), ...
    train_set_tall, ...
    'UniformOutput',false);
xTrain = gather(xTrain);
```

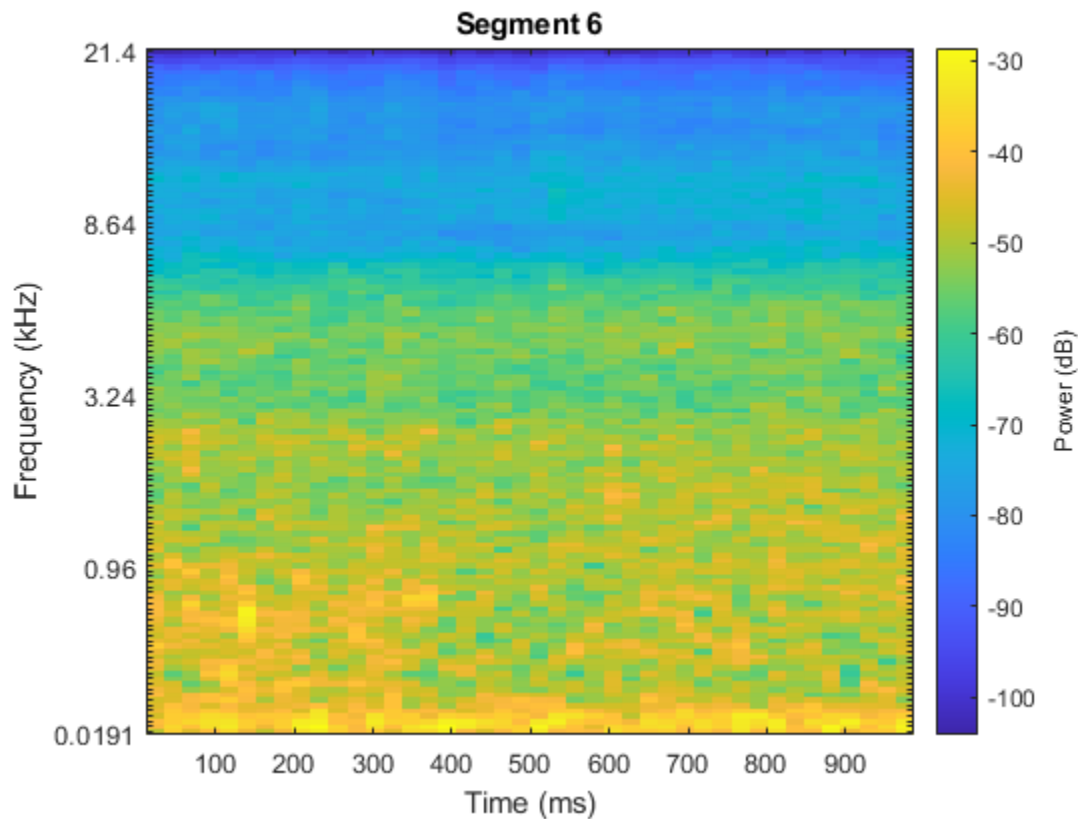
```

xTrain = cat(4,xTrain{:});

test_set_tall = tall(adsTest);
xTest = cellfun(@(x)HelperSegmentedMelSpectrograms(x,fs, ...
    'SegmentLength',segmentLength, ...
    'SegmentOverlap',segmentOverlap, ...
    'WindowLength',windowLength, ...
    'HopLength',samplesPerHop, ...
    'NumBands',numBands, ...
    'FFTLength',fftLength), ...
    test_set_tall, ...
    'UniformOutput',false);
xTest = gather(xTest);
xTest = cat(4,xTest{:});

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 4 min 48 sec
Evaluation completed in 4 min 48 sec
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1 min 40 sec
Evaluation completed in 1 min 40 sec

```



Replicate the labels of the training set so that they are in one-to-one correspondence with the segments.

```
numSegmentsPer10seconds = size(dataBuffered,2)/2;  
yTrain = repmat(adsTrain.Labels,1,numSegmentsPer10seconds)';  
yTrain = yTrain(:);
```

Data Augmentation for CNN

The DCASE 2017 dataset contains a relatively small number of acoustic recordings for the task, and the development set and evaluation set were recorded at different specific locations. As a result, it is easy to overfit to the data during training. One popular method to reduce overfitting is *mixup*. In mixup, you augment your dataset by mixing the features of two different classes. When you mix the features, you mix the labels in equal proportion. That is:

$$\tilde{x} = \lambda x_i + (1 - \lambda) x_j$$
$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j$$

Mixup was reformulated by [2] as labels drawn from a probability distribution instead of mixed labels. The implementation of mixup in this example is a simplified version of mixup: each spectrogram is mixed with a spectrogram of a different label with lambda set to 0.5. The original and mixed datasets are combined for training.

```
xTrainExtra = xTrain;  
yTrainExtra = yTrain;  
lambda = 0.5;  
for i = 1:size(xTrain,4)  
  
    % Find all available spectrograms with different labels.  
    availableSpectrograms = find(yTrain~=yTrain(i));  
  
    % Randomly choose one of the available spectrograms with a different label.  
    numAvailableSpectrograms = numel(availableSpectrograms);  
    idx = randi([1,numAvailableSpectrograms]);  
  
    % Mix.  
    xTrainExtra(:,:,i) = lambda*xTrain(:,:,i) + (1-lambda)*xTrain(:,:,availableSpectrograms);  
  
    % Specify the label as randomly set by lambda.  
    if rand > lambda  
        yTrainExtra(i) = yTrain(availableSpectrograms(idx));  
    end  
end  
xTrain = cat(4,xTrain,xTrainExtra);  
yTrain = [yTrain;yTrainExtra];
```

Call `summary` to display the distribution of labels for the augmented training set.

```
summary(yTrain)
```

beach	11769
bus	11904
cafe/restaurant	11873
car	11820
city_center	11886
forest_path	11936
grocery_store	11914
home	11923
library	11817
metro_station	11804

office	11922
park	11871
residential_area	11704
train	11773
tram	11924

Define and Train CNN

Define the CNN architecture. This architecture is based on [1] and modified through trial and error. See “List of Deep Learning Layers” (Deep Learning Toolbox) to learn more about deep learning layers available in MATLAB®.

```
imgSize = [size(xTrain,1),size(xTrain,2),size(xTrain,3)];
numF = 32;
layers = [ ...
    imageInputLayer(imgSize)

    batchNormalizationLayer

    convolution2dLayer(3,numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,2*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,2*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,8*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,8*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    averagePooling2dLayer(ceil(imgSize(1:2)/8))

    dropoutLayer(0.5)

    fullyConnectedLayer(15)
```

```
softmaxLayer  
classificationLayer];
```

Define `trainingOptions` (Deep Learning Toolbox) for the CNN. These options are based on [3] and modified through a systematic hyperparameter optimization workflow.

```
miniBatchSize = 128;  
tuneme = 128;  
lr = 0.05*miniBatchSize/tuneme;  
options = trainingOptions('sgdm', ...  
    'InitialLearnRate',lr, ...  
    'MiniBatchSize',miniBatchSize, ...  
    'Momentum',0.9, ...  
    'L2Regularization',0.005, ...  
    'MaxEpochs',8, ...  
    'Shuffle','every-epoch', ...  
    'Plots','training-progress', ...  
    'Verbose',false, ...  
    'LearnRateSchedule','piecewise', ...  
    'LearnRateDropPeriod',2, ...  
    'LearnRateDropFactor',0.2);
```

Call `trainNetwork` (Deep Learning Toolbox) to train the network.

```
trainedNet = trainNetwork(xTrain,yTrain,layers,options);
```

Evaluate CNN

Call `predict` (Deep Learning Toolbox) to predict responses from the trained network using the held-out test set.

```
cnnResponsesPerSegment = predict(trainedNet,xTest);
```

Average the responses over each 10-second audio clip.

```
classes = trainedNet.Layers(end).Classes;  
numFiles = numel(adsTest.Files);
```

```
counter = 1;  
cnnResponses = zeros(numFiles,numel(classes));  
for channel = 1:numFiles  
    cnnResponses(channel,:) = sum(cnnResponsesPerSegment(counter:counter+numSegmentsPer10seconds-1),2);  
    counter = counter + numSegmentsPer10seconds;  
end
```

For each 10-second audio clip, choose the maximum of the predictions, then map it to the corresponding predicted location.

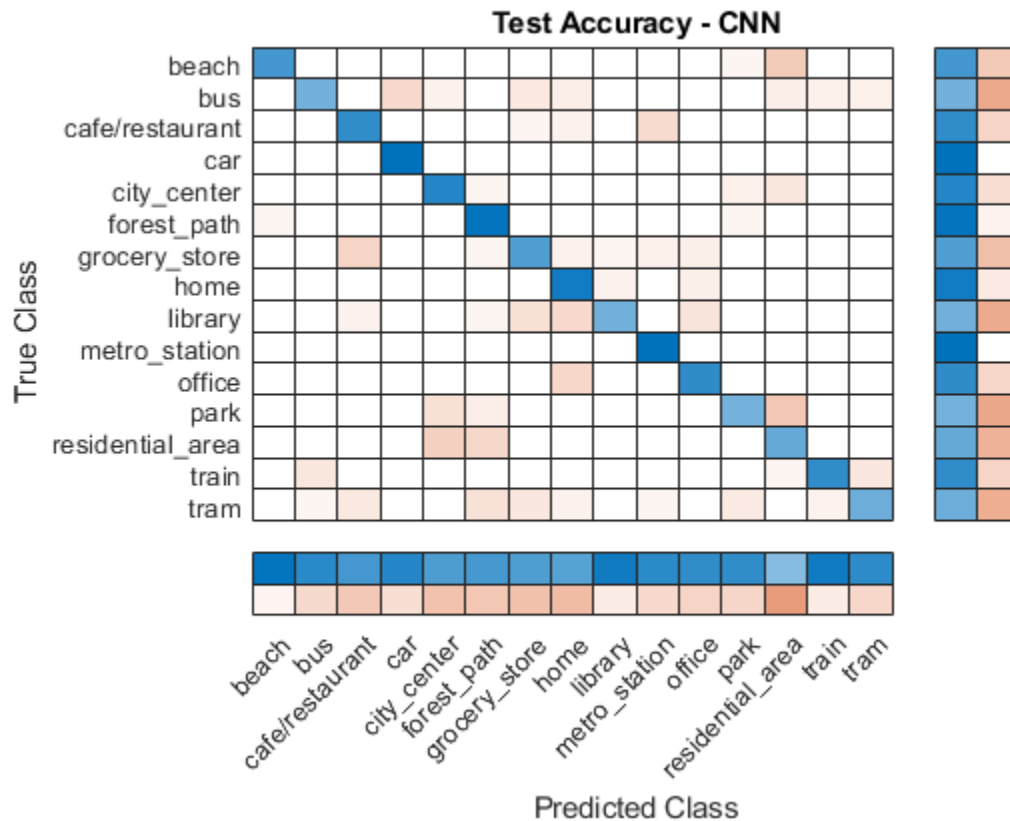
```
[~,classIdx] = max(cnnResponses,[],2);  
cnnPredictedLabels = classes(classIdx);
```

Call `confusionchart` (Deep Learning Toolbox) to visualize the accuracy on the test set. Return the average accuracy to the Command Window.

```
figure  
cm = confusionchart(adsTest.Labels,cnnPredictedLabels,'title','Test Accuracy - CNN');  
cm.ColumnSummary = 'column-normalized';  
cm.RowSummary = 'row-normalized';
```

```
fprintf('Average accuracy of CNN = %0.2f\n',mean(adsTest.Labels==cnnPredictedLabels)*100)
```

Average accuracy of CNN = 75.12



Feature Extraction for Ensemble Classifier

Wavelet scattering has been shown in [4] to provide a good representation of acoustic scenes. Define a `waveletScattering` (Wavelet Toolbox) object. The invariance scale and quality factors were determined through trial and error.

```
sf = waveletScattering('SignalLength',size(data,1), ...
    'SamplingFrequency',fs, ...
    'InvarianceScale',0.75, ...
    'QualityFactors',[4 1]);
```

Convert the audio signal to mono, and then call `featureMatrix` (Wavelet Toolbox) to return the scattering coefficients for the scattering decomposition framework, `sf`.

```
dataMono = mean(data,2);
scatteringCoefficients = featureMatrix(sf,dataMono,'Transform','log');
```

Average the scattering coefficients over the 10-second audio clip.

```
featureVector = mean(scatteringCoefficients,2);
fprintf('Number of wavelet features per 10-second clip = %d\n',numel(featureVector))
```

Number of wavelet features per 10-second clip = 290

The helper function `HelperWaveletFeatureVector` performs the above steps. Use a tall array with `cellfun` and `HelperWaveletFeatureVector` to parallelize the feature extraction. Extract wavelet feature vectors for the train and test sets.

```
scatteringTrain = cellfun(@(x)HelperWaveletFeatureVector(x,sf),train_set_tall,'UniformOutput',false);
xTrain = gather(scatteringTrain);
xTrain = cell2mat(xTrain)';

scatteringTest = cellfun(@(x)HelperWaveletFeatureVector(x,sf),test_set_tall,'UniformOutput',false);
xTest = gather(scatteringTest);
xTest = cell2mat(xTest)';
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 30 min 22 sec
Evaluation completed in 30 min 22 sec
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 10 min 30 sec
Evaluation completed in 10 min 30 sec
```

Define and Train Ensemble Classifier

Use `fitcensemble` to create a trained classification ensemble model (`ClassificationEnsemble`).

```
subspaceDimension = min(150,size(xTrain,2) - 1);
numLearningCycles = 30;
classificationEnsemble = fitcensemble(xTrain,adsTrain.Labels, ...
    'Method','Subspace', ...
    'NumLearningCycles',numLearningCycles, ...
    'Learners','discriminant', ...
    'NPredToSample',subspaceDimension, ...
    'ClassNames',removecats(unique(adsTrain.Labels))));
```

Evaluate Ensemble Classifier

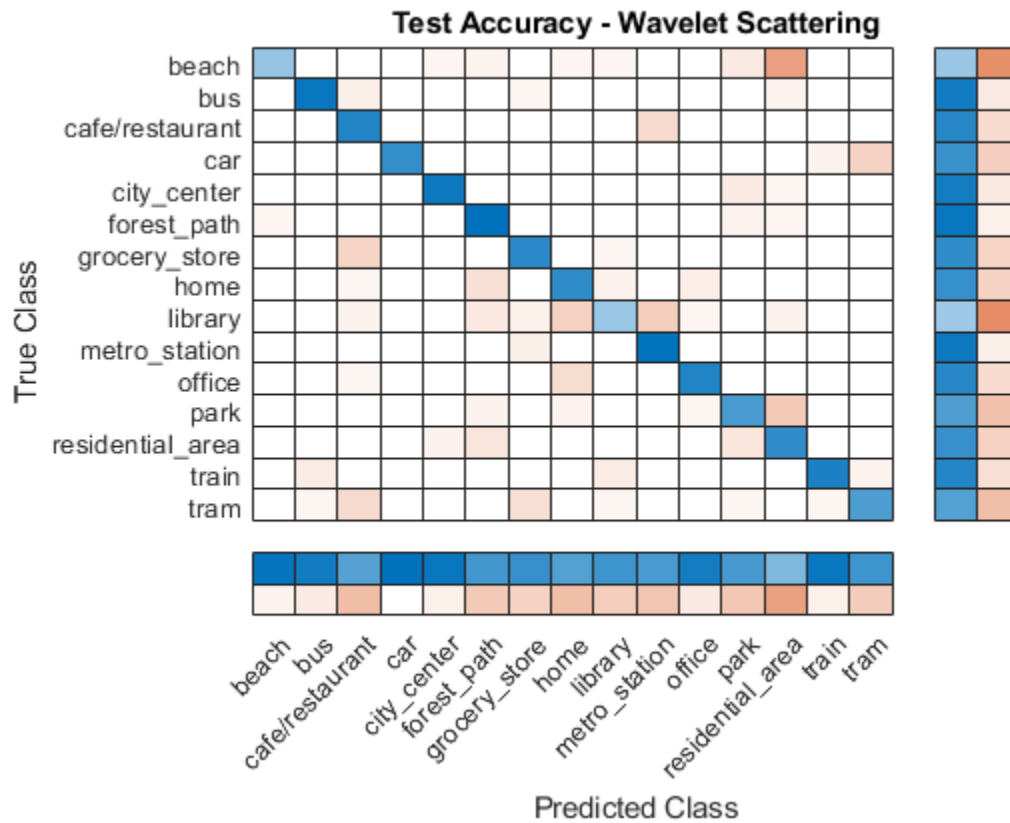
For each 10-second audio clip, call `predict` to return the labels and the weights, then map it to the corresponding predicted location. Call `confusionchart` (Deep Learning Toolbox) to visualize the accuracy on the test set. Print the average.

```
[waveletPredictedLabels,waveletResponses] = predict(classificationEnsemble,xTest);

figure
cm = confusionchart(adsTest.Labels,waveletPredictedLabels,'title','Test Accuracy - Wavelet Scattering');
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';

fprintf('Average accuracy of classifier = %0.2f\n',mean(adsTest.Labels==waveletPredictedLabels)*100);

Average accuracy of classifier = 76.23
```



Apply Late Fusion

For each 10-second clip, calling predict on the wavelet classifier and the CNN returns a vector indicating the relative confidence in their decision. Multiply the `waveletResponses` with the `cnnResponses` to create a late fusion system.

```
fused = waveletResponses .* cnnResponses;
[~,classIdx] = max(fused,[],2);
```

```
predictedLabels = classes(classIdx);
```

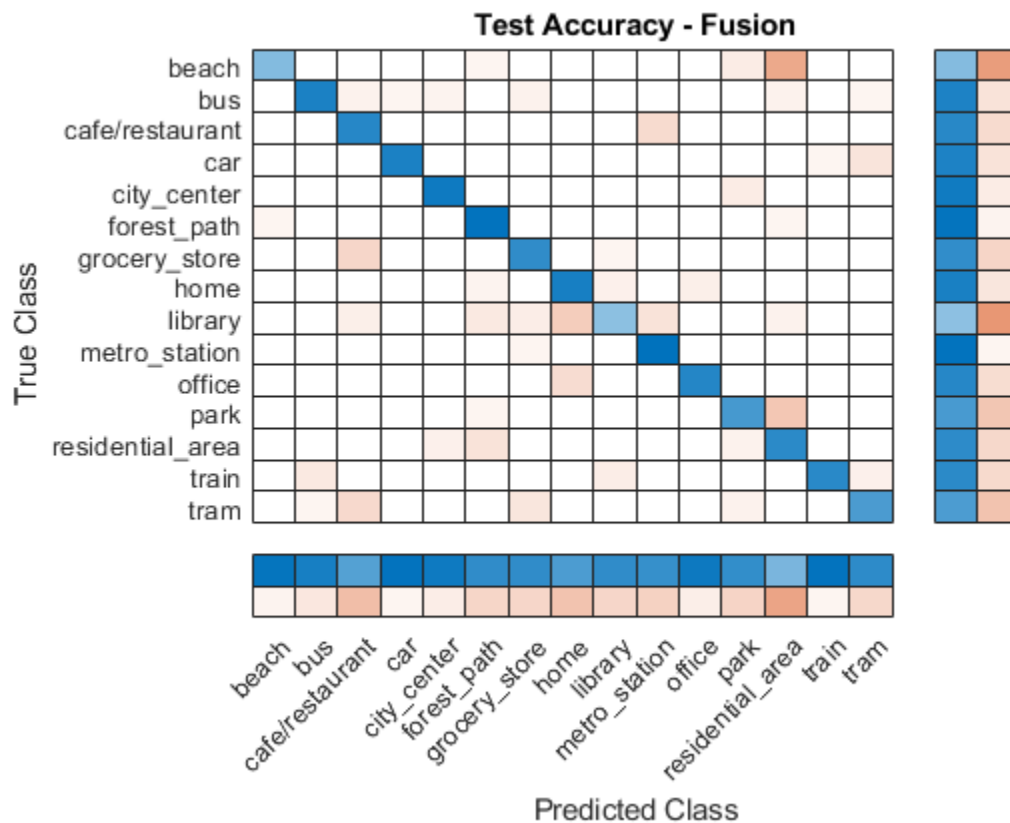
Evaluate Late Fusion

Call `confusionchart` to visualize the fused classification accuracy. Print the average accuracy to the Command Window.

```
figure
cm = confusionchart(adsTest.Labels,predictedLabels,'title','Test Accuracy - Fusion');
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';

fprintf('Average accuracy of fused models = %0.2f\n',mean(adsTest.Labels==predictedLabels)*100)

Average accuracy of fused models = 79.57
```



Close the parallel pool.

```
delete(pp)
```

Parallel pool using the 'local' profile is shutting down.

References

- [1] A. Mesaros, T. Heittola, and T. Virtanen. Acoustic Scene Classification: an Overview of DCASE 2017 Challenge Entries. In proc. International Workshop on Acoustic Signal Enhancement, 2018.
- [2] Huszar, Ferenc. "Mixup: Data-Dependent Data Augmentation." InFERENCe. November 03, 2017. Accessed January 15, 2019. <https://www.inference.vc/mixup-data-dependent-data-augmentation/>.
- [3] Han, Yoonchang, Jeongsoo Park, and Kyogu Lee. "Convolutional neural networks with binaural representations and background subtraction for acoustic scene classification." the Detection and Classification of Acoustic Scenes and Events (DCASE) (2017): 1-5.
- [4] Lostanlen, Vincent, and Joakim Anden. Binaural scene classification with wavelet scattering. Technical Report, DCASE2016 Challenge, 2016.
- [5] A. J. Eronen, V. T. Peltonen, J. T. Tuomi, A. P. Klapuri, S. Fagerlund, T. Sorsa, G. Lorho, and J. Huopaniemi, "Audio-based context recognition," IEEE Trans. on Audio, Speech, and Language Processing, vol 14, no. 1, pp. 321-329, Jan 2006.
- [6] TUT Acoustic scenes 2017, Development dataset

[7] TUT Acoustic scenes 2017, Evaluation dataset

Appendix -- Supporting Functions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%HelperSegmentedMelSpectrograms
```

```
function X = HelperSegmentedMelSpectrograms(x,fs,varargin)
```

```
p = inputParser;
addParameter(p,'WindowLength',1024);
addParameter(p,'HopLength',512);
addParameter(p,'NumBands',128);
addParameter(p,'SegmentLength',1);
addParameter(p,'SegmentOverlap',0);
addParameter(p,'FFTLenght',1024);
parse(p,varargin{:})
params = p.Results;
```

```
x = [sum(x,2),x(:,1)-x(:,2)];
x = x./max(max(x));
```

```
[xb_m,~] = buffer(x(:,1),round(params.SegmentLength*fs),round(params.SegmentOverlap*fs),'nodelay');
[xb_s,~] = buffer(x(:,2),round(params.SegmentLength*fs),round(params.SegmentOverlap*fs),'nodelay');
xb = zeros(size(xb_m,1),size(xb_m,2)+size(xb_s,2));
xb(:,1:2:end) = xb_m;
xb(:,2:2:end) = xb_s;
```

```
spec = melSpectrogram(xb,fs, ...
    'Window',hamming(params.WindowLength,'periodic'), ...
    'OverlapLength',params.WindowLength - params.HopLength, ...
    'FFTLenght',params.FFTLength, ...
    'NumBands',params.NumBands, ...
    'FrequencyRange',[0,floor(fs/2)]);
spec = log10(spec+eps);
```

```
X = reshape(spec,size(spec,1),size(spec,2),size(x,2),[]);
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%HelperWaveletFeatureVector
```

```
function features = HelperWaveletFeatureVector(x,sf)
```

```
x = mean(x,2);
features = featureMatrix(sf,x,'Transform','log');
features = mean(features,2);
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Keyword Spotting in Noise Using MFCC and LSTM Networks

This example shows how to identify a keyword in noisy speech using a deep learning network. In particular, the example uses a Bidirectional Long Short-Term Memory (BiLSTM) network and mel frequency cepstral coefficients (MFCC).

Introduction

Keyword spotting (KWS) is an essential component of voice-assist technologies, where the user speaks a predefined keyword to wake-up a system before speaking a complete command or query to the device.

This example trains a KWS deep network with feature sequences of mel-frequency cepstral coefficients (MFCC). The example also demonstrates how network accuracy in a noisy environment can be improved using data augmentation.

This example uses long short-term memory (LSTM) networks, which are a type of recurrent neural network (RNN) well-suited to study sequence and time-series data. An LSTM network can learn long-term dependencies between time steps of a sequence. An LSTM layer (`lstmLayer` (Deep Learning Toolbox)) can look at the time sequence in the forward direction, while a bidirectional LSTM layer (`biLstmLayer` (Deep Learning Toolbox)) can look at the time sequence in both forward and backward directions. This example uses a bidirectional LSTM layer.

The example uses the google Speech Commands Dataset to train the deep learning model. To run the example, you must first download the data set. If you do not want to download the data set or train the network, then you can load a pretrained network by opening this example in MATLAB® and typing `load("KWSNet.mat")` at the command line.

Spot Keyword with Pretrained Network

Before going into the training process in detail, you will use a pretrained keyword spotting network to identify a keyword.

In this example, the keyword to spot is **YES**.

Read a test signal where the keyword is uttered.

```
[audioIn, fs] = audioread('keywordTestSignal.wav');  
sound(audioIn,fs)
```

Load the pretrained network, as well as the mean and standard deviation vectors used for feature normalization.

```
load('keywordNetNoAugmentation.mat','keywordNetNoAugmentation','M','S');
```

Create an `audioFeatureExtractor` object to perform feature extraction.

```
WindowLength = 512;  
OverlapLength = 384;  
afe = audioFeatureExtractor('SampleRate',fs, ...  
    'Window',hann(WindowLength,'periodic'), ...  
    'OverlapLength',OverlapLength, ...  
    'mfcc',true, ...  
    'mfccDelta',true, ...  
    'mfccDeltaDelta',true);  
setExtractorParams(afe,'mfcc','NumCoeffs',13);
```


Extract features from the test signal and normalize them.

```
features = extract(afe, audioIn);
```

```
features = (features - M)./S;
```

Compute the keyword spotting binary mask. A mask value of one corresponds to a segment where the keyword was spotted.

```
mask = classify(keywordNetNoAugmentation, features.');
```

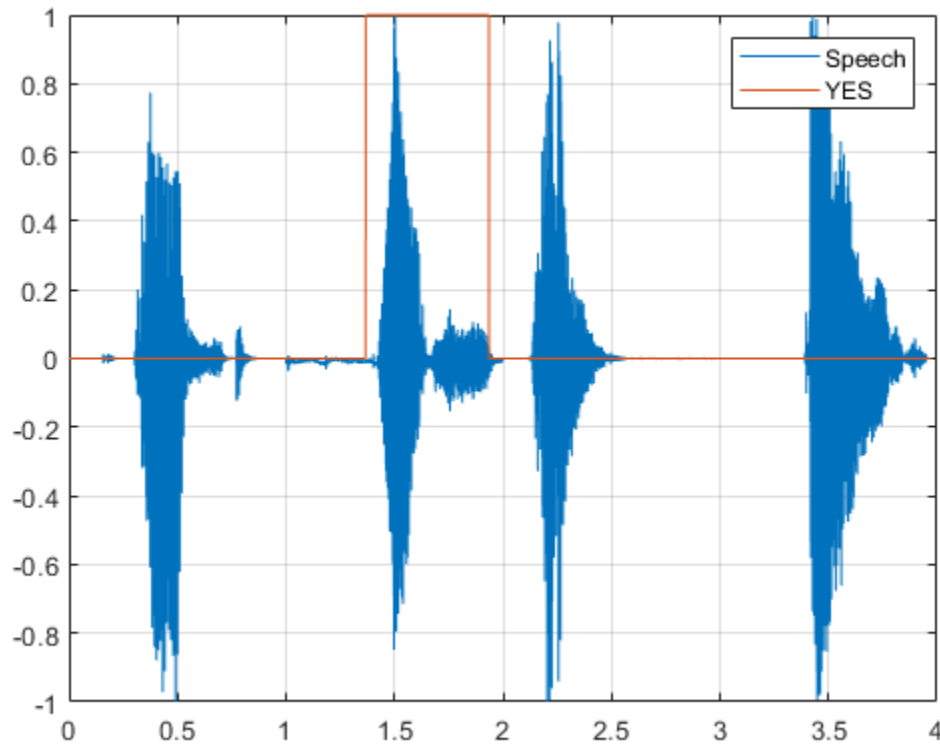
Each sample in the mask corresponds to 128 samples from the speech signal (WindowLength - OverlapLength).

Expand the mask to the length of the signal.

```
mask = repmat(mask, WindowLength-OverlapLength, 1);
mask = double(mask) - 1;
mask = mask(:);
```

Plot the test signal and the mask.

```
figure
audioIn = audioIn(1:length(mask));
t = (0:length(audioIn)-1)/fs;
plot(t, audioIn)
grid on
hold on
plot(t, mask)
legend('Speech', 'YES')
```



Listen to the spotted keyword.

```
sound(audioIn(mask==1),fs)
```

Detect Commands Using Streaming Audio from Microphone

Test your pre-trained command detection network on streaming audio from your microphone. Try saying random words, including the keyword (**YES**).

Call `generateMATLABFunction` on the `audioFeatureExtractor` object to create the feature extraction function. You will use this function in the processing loop.

```
generateMATLABFunction(afe,'generateKeywordFeatures','IsStreaming',true);
```

Define an audio device reader that can read audio from your microphone. Set the frame length to the hop length. This enables you to compute a new set of features for every new audio frame from the microphone.

```
HopLength = WindowLength - OverlapLength;
FrameLength = HopLength;
adr = audioDeviceReader('SampleRate',fs, ...
    'SamplesPerFrame',FrameLength);
```

Create a scope for visualizing the speech signal and the estimated mask.

```
scope = timescope('SampleRate',fs, ...
    'TimeSpanSource','property', ...
    'TimeSpan',5, ...
```

```

'TimeSpanOvverrunAction','Scroll', ...
'BufferLength',fs*5*2, ...
'ShowLegend',true, ...
'ChannelNames',{'Speech','Keyword Mask'}, ...
'YLimits',[-1.2 1.2], ...
'Title','Keyword Spotting');

```

Define the rate at which you estimate the mask. You will generate a mask once every NumHopsPerUpdate audio frames.

```
NumHopsPerUpdate = 16;
```

Initialize a buffer for the audio.

```
dataBuff = dsp.AsyncBuffer(WindowLength);
```

Initialize a buffer for the computed features.

```
featureBuff = dsp.AsyncBuffer(NumHopsPerUpdate);
```

Initialize a buffer to manage plotting the audio and the mask.

```
plotBuff = dsp.AsyncBuffer(NumHopsPerUpdate*WindowLength);
```

To run the loop indefinitely, set timeLimit to Inf. To stop the simulation, close the scope.

```
timeLimit = 20;
```

```
tic
```

```
while toc < timeLimit
```

```

    data = adr();
    write(dataBuff, data);
    write(plotBuff, data);

```

```

    frame = read(dataBuff,WindowLength,OverlapLength);
    features = generateKeywordFeatures(frame,fs);
    write(featureBuff,features. ');

```

```

    if featureBuff.NumUnreadSamples == NumHopsPerUpdate
        featureMatrix = read(featureBuff);
        featureMatrix(~isfinite(featureMatrix)) = 0;
        featureMatrix = (featureMatrix - M)./S;

```

```

        [keywordNet, v] = classifyAndUpdateState(keywordNetNoAugmentation,featureMatrix. ');
        v = double(v) - 1;
        v = repmat(v, HopLength, 1);
        v = v(:);
        v = mode(v);
        v = repmat(v, NumHopsPerUpdate * HopLength,1);

```

```

        data = read(plotBuff);
        scope([data, v]);

```

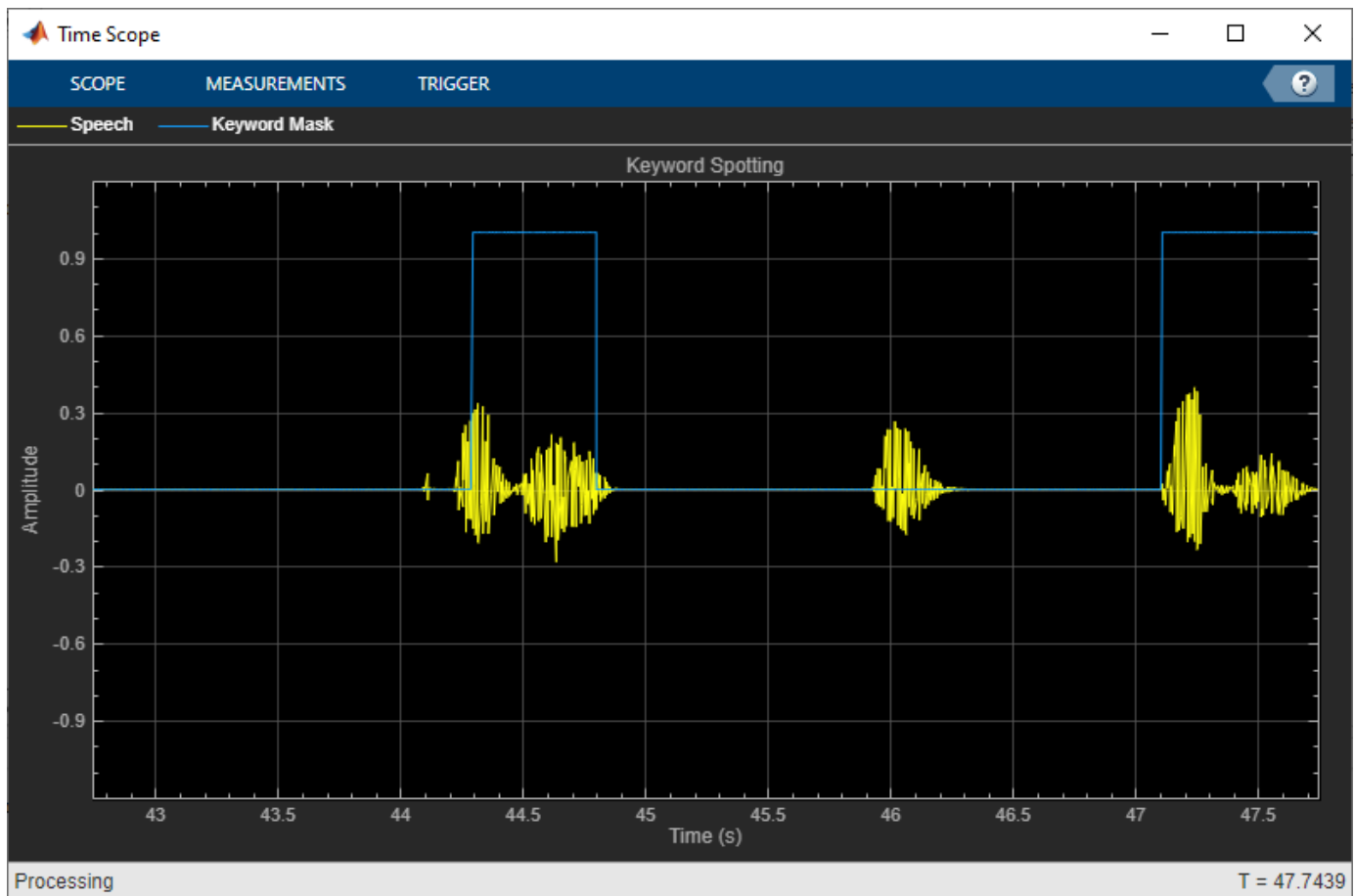
```

        if ~isVisible(scope)
            break;
        end

```

```
end
```

```
end
hide(scope)
```



In the rest of the example, you will learn how to train the keyword spotting network.

Training Process Summary

The training process goes through the following steps:

- 1 Inspect a "gold standard" keyword spotting baseline on a validation signal.
- 2 Create training utterances from a noise-free dataset.
- 3 Train a keyword spotting LSTM network using MFCC sequences extracted from those utterances.
- 4 Check the network accuracy by comparing the validation baseline to the output of the network when applied to the validation signal.
- 5 Check the network accuracy for a validation signal corrupted by noise.
- 6 Augment the training dataset by injecting noise to the speech data using `audioDataAugmenter`.
- 7 Retrain the network with the augmented dataset.
- 8 Verify that the retrained network now yields higher accuracy when applied to the noisy validation signal.

Inspect the Validation Signal

You use a sample speech signal to validate the KWS network. The validation signal consists 34 seconds of speech with the keyword **YES** appearing intermittently.

Load the validation signal.

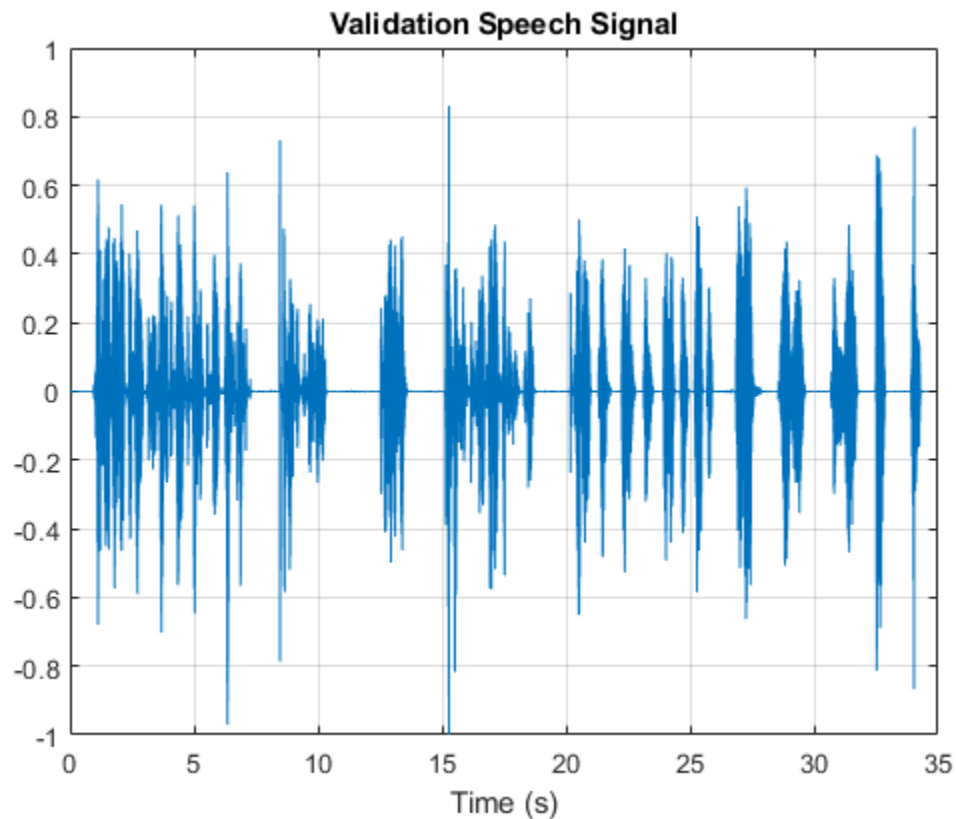
```
[audioIn,fs] = audioread('KeywordSpeech-16-16-mono-34secs.flac');
```

Listen to the signal.

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

Visualize the signal.

```
figure
t = (1/fs) * (0:length(audioIn)-1);
plot(t,audioIn);
grid on
xlabel('Time (s)')
title('Validation Speech Signal')
```



Inspect the KWS Baseline

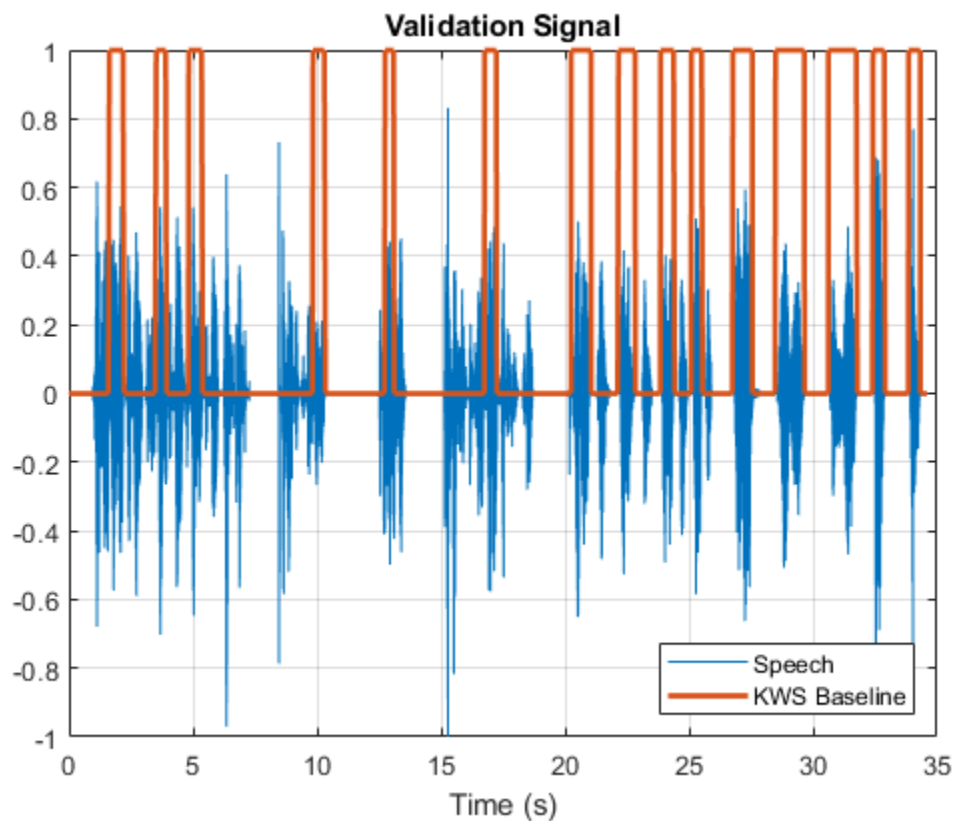
Load the KWS baseline. This baseline was obtained using speech2text: “Create Keyword Spotting Mask Using Audio Labeler”.

```
load('KWSBaseline.mat','KWSBaseline')
```

The baseline is a logical vector of the same length as the validation audio signal. Segments in `audioIn` where the keyword is uttered are set to one in `KWSBaseline`.

Visualize the speech signal along with the KWS baseline.

```
fig = figure;
plot(t,[audioIn,KWSBaseline'])
grid on
xlabel('Time (s)')
legend('Speech','KWS Baseline','Location','southeast')
l = findall(fig,'type','line');
l(1).LineWidth = 2;
title("Validation Signal")
```



Listen to the speech segments identified as keywords.

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

The objective of the network that you train is to output a KWS mask of zeros and ones like this baseline.

Load Speech Commands Data Set

Download and extract the Google Speech Commands Dataset [1] on page 1-0 .

```
url = 'https://ssd.mathworks.com/supportfiles/audio/google_speech.zip';
```

```
downloadFolder = tempdir;
```

```
datasetFolder = fullfile(downloadFolder, 'google_speech');

if ~exist(datasetFolder, 'dir')
    disp('Downloading Google speech commands data set (1.5 GB)...')
    unzip(url, datasetFolder)
end
```

Create an `audioDatastore` that points to the data set.

```
ads = audioDatastore(datasetFolder, 'LabelSource', 'foldername', 'Includesubfolders', true);
ads = shuffle(ads);
```

The dataset contains background noise files that are not used in this example. Use `subset` to create a new datastore that does not have the background noise files.

```
isBackNoise = ismember(ads.Labels, "background");
ads = subset(ads, ~isBackNoise);
```

The dataset has approximately 65,000 one-second long utterances of 30 short words (including the keyword YES). Get a breakdown of the word distribution in the datastore.

```
countEachLabel(ads)
```

```
ans=30x2 table
    Label    Count
    _____
    bed      1713
    bird     1731
    cat      1733
    dog      1746
    down     2359
    eight    2352
    five     2357
    four     2372
    go       2372
    happy    1742
    house    1750
    left     2353
    marvin   1746
    nine     2364
    no       2375
    off      2357
    :
```

Split `ads` into two datastores: The first datastore contains files corresponding to the keyword. The second datastore contains all the other words.

```
keyword = 'yes';
isKeyword = ismember(ads.Labels, keyword);
ads_keyword = subset(ads, isKeyword);
ads_other = subset(ads, ~isKeyword);
```

To train the network with the entire dataset and achieve the highest possible accuracy, set `reduceDataset` to `false`. To run this example quickly, set `reduceDataset` to `true`.

```
reduceDataset = ;
if reduceDataset
```

```
% Reduce the dataset by a factor of 20
ads_keyword = splitEachLabel(ads_keyword,round(numel(ads_keyword.Files) / 20));
numUniqueLabels = numel(unique(ads_other.Labels));
ads_other = splitEachLabel(ads_other,round(numel(ads_other.Files) / numUniqueLabels / 20));
end
```

Get a breakdown of the word distribution in each datastore. Shuffle the `ads_other` datastore so that consecutive reads return different words.

```
countEachLabel(ads_keyword)
```

```
ans=1x2 table
  Label    Count
  _____
  yes      2377
```

```
countEachLabel(ads_other)
```

```
ans=29x2 table
  Label    Count
  _____
  bed      1713
  bird     1731
  cat      1733
  dog      1746
  down     2359
  eight    2352
  five     2357
  four     2372
  go       2372
  happy    1742
  house    1750
  left     2353
  marvin   1746
  nine     2364
  no       2375
  off      2357
  :
```

```
ads_other = shuffle(ads_other);
```

Create Training Sentences and Labels

The training datastores contain one-second speech signals where one word is uttered. You will create more complex training speech utterances that contain a mixture of the keyword along with other words.

Here is an example of a constructed utterance. Read one keyword from the keyword datastore and normalize it to have a maximum value of one.

```
yes = read(ads_keyword);
yes = yes / max(abs(yes));
```

The signal has non-speech portions (silence, background noise, etc.) that do not contain useful speech information. This example removes silence using `detectSpeech`.

Get the start and end indices of the useful portion of the signal.

```
speechIndices = detectSpeech(yes,fs);
```

Randomly select the number of words to use in the synthesized training sentence. Use a maximum of 10 words.

```
numWords = randi([0 10]);
```

Randomly pick the location at which the keyword occurs.

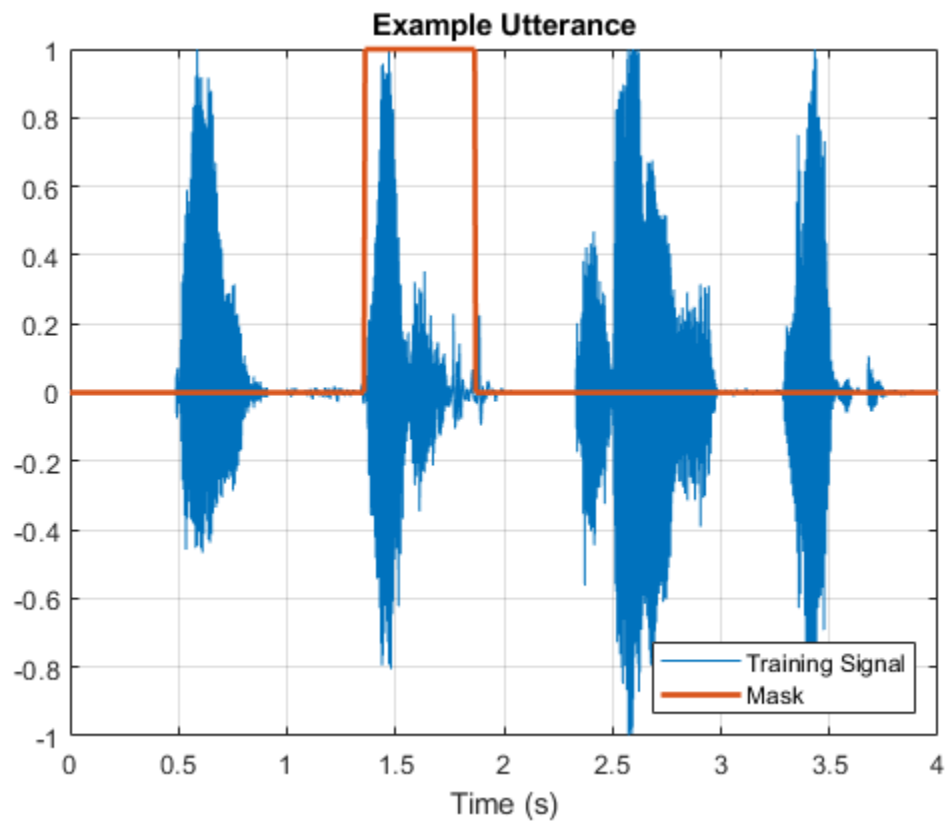
```
keywordLocation = randi([1 numWords+1]);
```

Read the desired number of non-keyword utterances, and construct the training sentence and mask.

```
sentence = [];
mask = [];
for index = 1:numWords+1
    if index == keywordLocation
        sentence = [sentence;yes]; %#ok
        newMask = zeros(size(yes));
        newMask(speechIndices(1,1):speechIndices(1,2)) = 1;
        mask = [mask;newMask]; %#ok
    else
        other = read(ads_other);
        other = other ./ max(abs(other));
        sentence = [sentence;other]; %#ok
        mask = [mask;zeros(size(other))]; %#ok
    end
end
```

Plot the training sentence along with the mask.

```
figure
t = (1/fs) * (0:length(sentence)-1);
fig = figure;
plot(t,[sentence,mask])
grid on
xlabel('Time (s)')
legend('Training Signal','Mask','Location','southeast')
l = findall(fig,'type','line');
l(1).LineWidth = 2;
title("Example Utterance")
```



Listen to the training sentence.

```
sound(sentence, fs)
```

Extract Features

This example trains a deep learning network using 39 MFCC coefficients (13 MFCC, 13 delta and 13 delta-delta coefficients).

Define parameters required for MFCC extraction.

```
WindowLength = 512;
OverlapLength = 384;
```

Create an `audioFeatureExtractor` object to perform the feature extraction.

```
afe = audioFeatureExtractor('SampleRate',fs, ...
    'Window',hann(WindowLength,'periodic'), ...
    'OverlapLength',OverlapLength, ...
    'mfcc',true, ...
    'mfccDelta',true, ...
    'mfccDeltaDelta',true);
```

Extract the features.

```
featureMatrix = extract(afe,sentence);
size(featureMatrix)
```

```
ans = 1×2

    497     39
```

Note that you compute MFCC by sliding a window through the input, so the feature matrix is shorter than the input speech signal. Each row in `featureMatrix` corresponds to 128 samples from the speech signal (`WindowLength - OverlapLength`).

Compute a mask of the same length as `featureMatrix`.

```
HopLength = WindowLength - OverlapLength;
range = HopLength * (1:size(featureMatrix,1)) + HopLength;
featureMask = zeros(size(range));
for index = 1:numel(range)
    featureMask(index) = mode(mask( (index-1)*HopLength+1:(index-1)*HopLength+WindowLength ));
end
```

Extract Features from Training Dataset

Sentence synthesis and feature extraction for the whole training dataset can be quite time-consuming. To speed up processing, if you have Parallel Computing Toolbox™, partition the training datastore, and process each partition on a separate worker.

Select a number of datastore partitions.

```
numPartitions = 6;
```

Initialize cell arrays for the feature matrices and masks.

```
TrainingFeatures = {};
TrainingMasks = {};
```

Perform sentence synthesis, feature extraction, and mask creation using `parfor`.

```
emptyCategories = categorical([1 0]);
emptyCategories(:) = [];

tic
parfor ii = 1:numPartitions

    subads_keyword = partition(ads_keyword,numPartitions,ii);
    subads_other = partition(ads_other,numPartitions,ii);

    count = 1;
    localFeatures = cell(length(subads_keyword.Files),1);
    localMasks = cell(length(subads_keyword.Files),1);

    while hasdata(subads_keyword)

        % Create a training sentence
        [sentence,mask] = HelperSynthesizeSentence(subads_keyword,subads_other,fs,WindowLength);

        % Compute mfcc features
        featureMatrix = extract(afe, sentence);
        featureMatrix(~isfinite(featureMatrix)) = 0;

        % Create mask
```

```
hopLength = WindowLength - OverlapLength;
range = (hopLength) * (1:size(featureMatrix,1)) + hopLength;
featureMask = zeros(size(range));
for index = 1:numel(range)
    featureMask(index) = mode(mask( (index-1)*hopLength+1:(index-1)*hopLength+WindowLength));
end

localFeatures{count} = featureMatrix;
localMasks{count} = [emptyCategories,categorical(featureMask)];

count = count + 1;
end

TrainingFeatures = [TrainingFeatures;localFeatures];
TrainingMasks = [TrainingMasks;localMasks];
end

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).

fprintf('Training feature extraction took %f seconds.\n',toc)

Training feature extraction took 83.790340 seconds.
```

It is good practice to normalize all features to have zero mean and unity standard deviation. Compute the mean and standard deviation for each coefficient and use them to normalize the data.

```
sampleFeature = TrainingFeatures{1};
numFeatures = size(sampleFeature,2);
featuresMatrix = cat(1,TrainingFeatures{:});
M = mean(featuresMatrix);
S = std(featuresMatrix);
for index = 1:length(TrainingFeatures)
    f = TrainingFeatures{index};
    f = (f - M) ./ S;
    TrainingFeatures{index} = f.'; %#ok
end
```

Extract Validation Features

Extract MFCC features from the validation signal.

```
featureMatrix = extract(afe, audioIn);
featureMatrix(~isfinite(featureMatrix)) = 0;
```

Normalize the validation features.

```
FeaturesValidationClean = (featureMatrix - M)./S;
range = HopLength * (1:size(FeaturesValidationClean,1)) + HopLength;
```

Construct the validation KWS mask.

```
featureMask = zeros(size(range));
for index = 1:numel(range)
    featureMask(index) = mode(KWSBaseline( (index-1)*HopLength+1:(index-1)*HopLength+WindowLength));
end
BaselineV = categorical(featureMask);
```

Define the LSTM Network Architecture

LSTM networks can learn long-term dependencies between time steps of sequence data. This example uses the bidirectional LSTM layer `bilstmLayer` (Deep Learning Toolbox) to look at the sequence in both forward and backward directions.

Specify the input size to be sequences of size `numFeatures`. Specify two hidden bidirectional LSTM layers with an output size of 150 and output a sequence. This command instructs the bidirectional LSTM layer to map the input time series into 150 features that are passed to the next layer. Specify two classes by including a fully connected layer of size 2, followed by a softmax layer and a classification layer.

```
layers = [ ...
    sequenceInputLayer(numFeatures)
    bilstmLayer(150,"OutputMode","sequence")
    bilstmLayer(150,"OutputMode","sequence")
    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer
];
```

Define Training Options

Specify the training options for the classifier. Set `MaxEpochs` to 10 so that the network makes 10 passes through the training data. Set `MiniBatchSize` to 64 so that the network looks at 64 training signals at a time. Set `Plots` to "training-progress" to generate plots that show the training progress as the number of iterations increases. Set `Verbose` to `false` to disable printing the table output that corresponds to the data shown in the plot. Set `Shuffle` to "every-epoch" to shuffle the training sequence at the beginning of each epoch. Set `LearnRateSchedule` to "piecewise" to decrease the learning rate by a specified factor (0.1) every time a certain number of epochs (5) has passed. Set `ValidationData` to the validation predictors and targets.

This example uses the adaptive moment estimation (ADAM) solver. ADAM performs better with recurrent neural networks (RNNs) like LSTMs than the default stochastic gradient descent with momentum (SGDM) solver.

```
maxEpochs = 10;
miniBatchSize = 64;
options = trainingOptions("adam", ...
    "InitialLearnRate",1e-4, ...
    "MaxEpochs",maxEpochs, ...
    "MiniBatchSize",miniBatchSize, ...
    "Shuffle","every-epoch", ...
    "Verbose",false, ...
    "ValidationFrequency",floor(numel(TrainingFeatures)/miniBatchSize), ...
    "ValidationData",{FeaturesValidationClean.',BaselineV}, ...
    "Plots","training-progress", ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.1, ...
    "LearnRateDropPeriod",5);
```

Train the LSTM Network

Train the LSTM network with the specified training options and layer architecture using `trainNetwork` (Deep Learning Toolbox). Because the training set is large, the training process can take several minutes.

```
[keywordNetNoAugmentation,netInfo] = trainNetwork(TrainingFeatures,TrainingMasks,layers,options)
if reduceDataset
    load('keywordNetNoAugmentation.mat','keywordNetNoAugmentation','M','S');
end
```

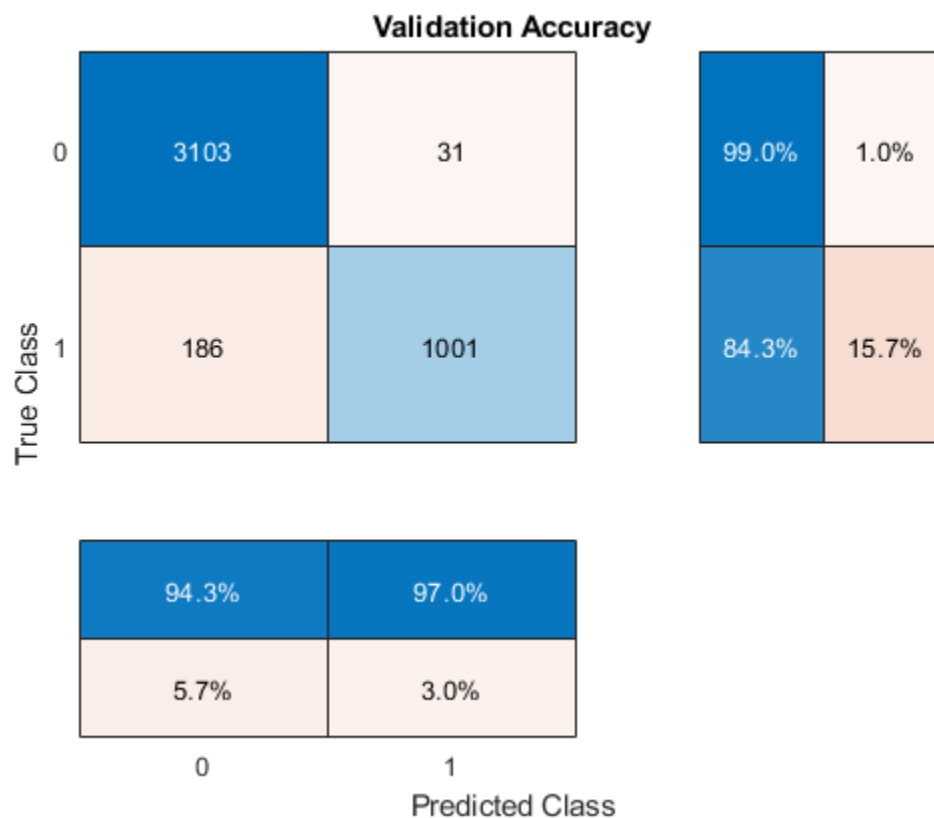
Check Network Accuracy for Noise-Free Validation Signal

Estimate the KWS mask for the validation signal using the trained network.

```
v = classify(keywordNetNoAugmentation,FeaturesValidationClean.');
```

Calculate and plot the validation confusion matrix from the vectors of actual and estimated labels.

```
figure
cm = confusionchart(BaselineV,v,"title","Validation Accuracy");
cm.ColumnSummary = "column-normalized";
cm.RowSummary = "row-normalized";
```



Convert the network output from categorical to double.

```
v = double(v) - 1;
v = repmat(v,HopLength,1);
v = v(:);
```

Listen to the keyword areas identified by the network.

```
sound(audioIn(logical(v)),fs)
```

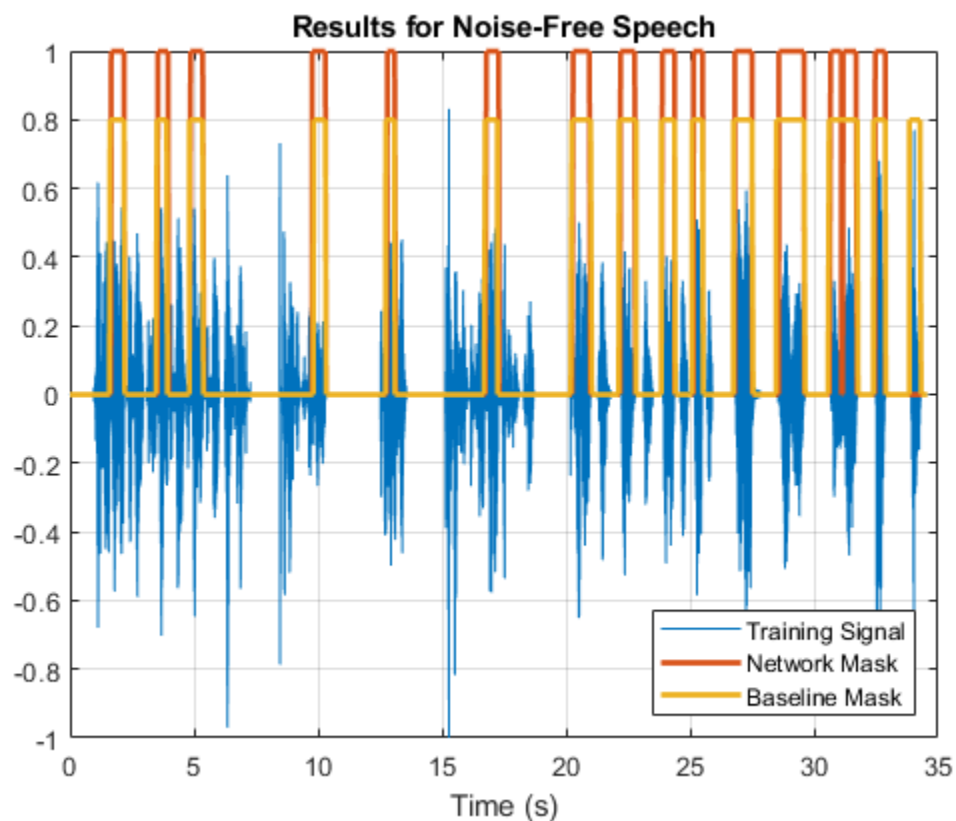
Visualize the estimated and expected KWS masks.

```

baseline = double(BaselineV) - 1;
baseline = repmat(baseline,HopLength,1);
baseline = baseline(:);

t = (1/fs) * (0:length(v)-1);
fig = figure;
plot(t,[audioIn(1:length(v)),v,0.8*baseline])
grid on
xlabel('Time (s)')
legend('Training Signal','Network Mask','Baseline Mask','Location','southeast')
l = findall(fig,'type','line');
l(1).LineWidth = 2;
l(2).LineWidth = 2;
title('Results for Noise-Free Speech')

```



Check Network Accuracy for a Noisy Validation Signal

You will now check the network accuracy for a noisy speech signal. The noisy signal was obtained by corrupting the clean validation signal by additive white Gaussian noise.

Load the noisy signal.

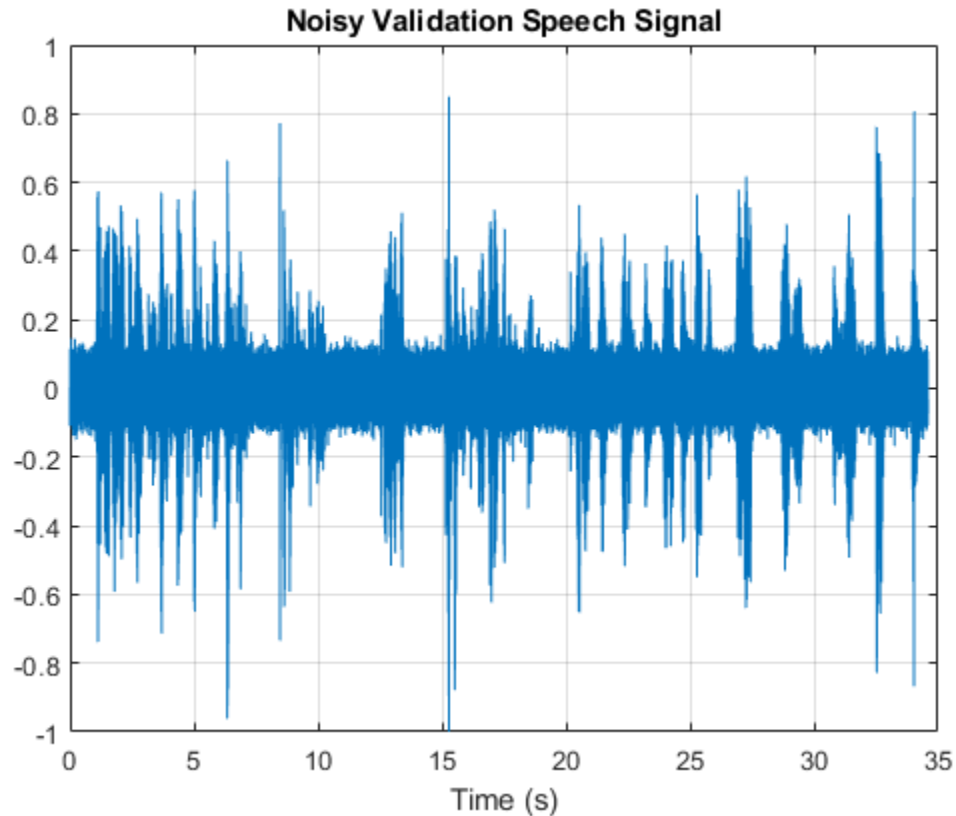
```

[audioInNoisy,fs] = audioread('NoisyKeywordSpeech-16-16-mono-34secs.flac');
sound(audioInNoisy,fs)

```

Visualize the signal.

```
figure
t = (1/fs) * (0:length(audioInNoisy)-1);
plot(t, audioInNoisy)
grid on
xlabel('Time (s)')
title('Noisy Validation Speech Signal')
```



Extract the feature matrix from the noisy signal.

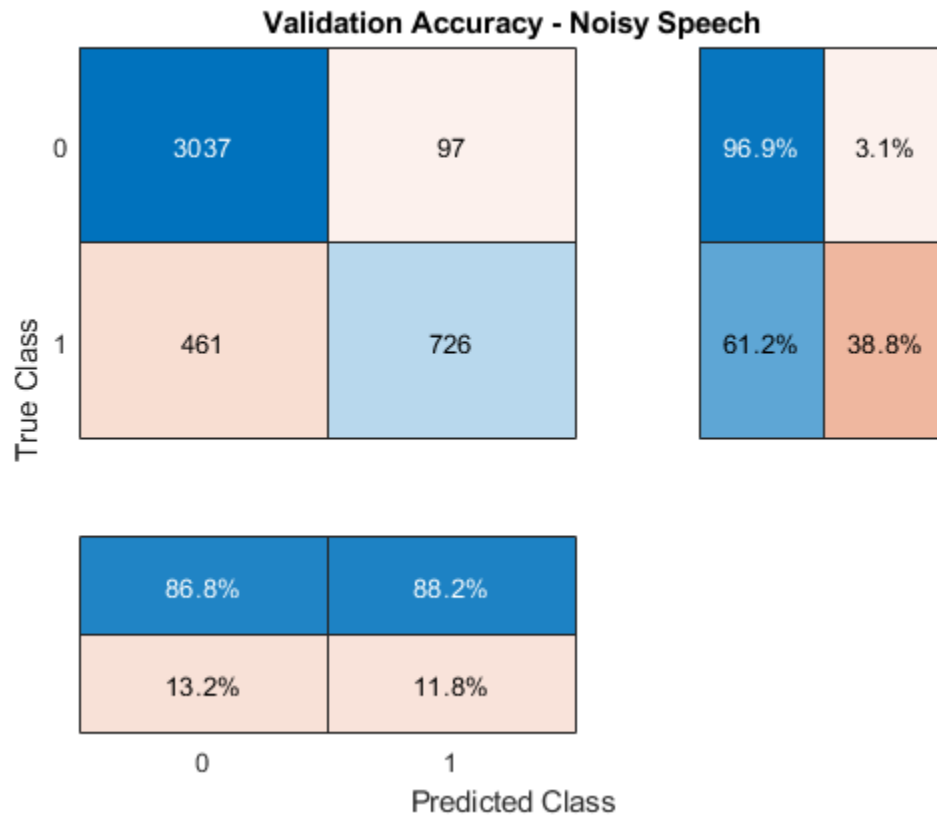
```
featureMatrixV = extract(afe, audioInNoisy);
featureMatrixV(~isfinite(featureMatrixV)) = 0;
FeaturesValidationNoisy = (featureMatrixV - M)./S;
```

Pass the feature matrix to the network.

```
v = classify(keywordNetNoAugmentation, FeaturesValidationNoisy.');
```

Compare the network output to the baseline. Note that the accuracy is lower than the one you got for a clean signal.

```
figure
cm = confusionchart(BaselineV, v, "title", "Validation Accuracy - Noisy Speech");
cm.ColumnSummary = "column-normalized";
cm.RowSummary = "row-normalized";
```

Convert the network output from categorical to double.

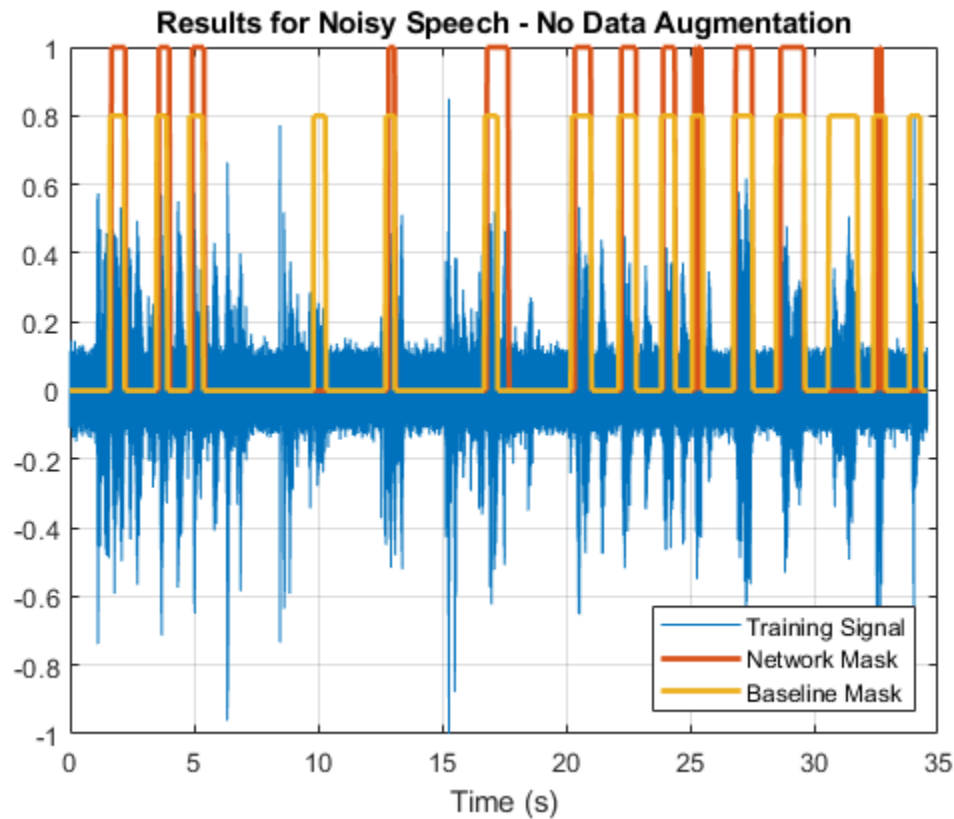
```
v = double(v) - 1;
v = repmat(v,HopLength,1);
v = v(:);
```

Listen to the keyword areas identified by the network.

```
sound(audioIn(logical(v)),fs)
```

Visualize the estimated and baseline masks.

```
t = (1/fs)*(0:length(v)-1);
fig = figure;
plot(t,[audioInNoisy(1:length(v)),v,0.8*baseline])
grid on
xlabel('Time (s)')
legend('Training Signal','Network Mask','Baseline Mask','Location','southeast')
l = findall(fig,'type','line');
l(1).LineWidth = 2;
l(2).LineWidth = 2;
title('Results for Noisy Speech - No Data Augmentation')
```



Perform Data Augmentation

The trained network did not perform well on a noisy signal because the trained dataset contained only noise-free sentences. You will rectify this by augmenting your dataset to include noisy sentences.

Use `audioDataAugmenter` to augment your dataset.

```
ada = audioDataAugmenter('TimeStretchProbability',0, ...
                        'PitchShiftProbability',0, ...
                        'VolumeControlProbability',0, ...
                        'TimeShiftProbability',0, ...
                        'SNRRange',[-1, 1], ...
                        'AddNoiseProbability',0.85);
```

With these settings, the `audioDataAugmenter` object corrupts an input audio signal with white Gaussian noise with a probability of 85%. The SNR is randomly selected from the range $[-1, 1]$ (in dB). There is a 15% probability that the augments does not modify your input signal.

As an example, pass an audio signal to the augments.

```
reset(ads_keyword)
x = read(ads_keyword);
data = augment(ada,x,fs)

data=1x2 table
      Audio      AugmentationInfo
```

```
{16000×1 double}      [1×1 struct]
```

Inspect the AugmentationInfo variable in data to verify how the signal was modified.

```
data.AugmentationInfo
```

```
ans = struct with fields:
    SNR: 0.1031
```

Reset the datastores.

```
reset(ads_keyword)
reset(ads_other)
```

Initialize the feature and mask cells.

```
TrainingFeatures = {};
TrainingMasks = {};
```

Perform feature extraction again. Each signal is corrupted by noise with a probability of 85%, so your augmented dataset has approximately 85% noisy data and 15% noise-free data.

```
tic
parfor ii = 1:numPartitions

    subads_keyword = partition(ads_keyword,numPartitions,ii);
    subads_other = partition(ads_other,numPartitions,ii);

    count = 1;
    localFeatures = cell(length(subads_keyword.Files),1);
    localMasks = cell(length(subads_keyword.Files),1);

    while hasdata(subads_keyword)

        [sentence,mask] = HelperSynthesizeSentence(subads_keyword,subads_other,fs,WindowLength);

        % Corrupt with noise
        augmentedData = augment(ada,sentence,fs);
        sentence = augmentedData.Audio{1};

        % Compute mfcc features
        featureMatrix = extract(afe, sentence);
        featureMatrix(~isfinite(featureMatrix)) = 0;

        hopLength = WindowLength - OverlapLength;
        range = hopLength * (1:size(featureMatrix,1)) + hopLength;
        featureMask = zeros(size(range));
        for index = 1:numel(range)
            featureMask(index) = mode(mask( (index-1)*hopLength+1:(index-1)*hopLength+WindowLength));
        end

        localFeatures{count} = featureMatrix;
        localMasks{count} = [emptyCategories,categorical(featureMask)];

        count = count + 1;
    end
end
```

```
    TrainingFeatures = [TrainingFeatures;localFeatures];
    TrainingMasks = [TrainingMasks;localMasks];
end
fprintf('Training feature extraction took %f seconds.\n',toc)
```

Training feature extraction took 36.809452 seconds.

Compute the mean and standard deviation for each coefficient; use them to normalize the data.

```
sampleFeature = TrainingFeatures{1};
numFeatures = size(sampleFeature,2);
featuresMatrix = cat(1,TrainingFeatures{:});
M = mean(featuresMatrix);
S = std(featuresMatrix);
for index = 1:length(TrainingFeatures)
    f = TrainingFeatures{index};
    f = (f - M) ./ S;
    TrainingFeatures{index} = f.'; %#ok
end
```

Normalize the validation features with the new mean and standard deviation values.

```
FeaturesValidationNoisy = (featureMatrixV - M)./S;
```

Retrain Network with Augmented Dataset

Recreate the training options. Use the noisy baseline features and mask for validation.

```
options = trainingOptions("adam", ...
    "InitialLearnRate",1e-4, ...
    "MaxEpochs",maxEpochs, ...
    "MiniBatchSize",miniBatchSize, ...
    "Shuffle","every-epoch", ...
    "Verbose",false, ...
    "ValidationFrequency",floor(numel(TrainingFeatures)/miniBatchSize), ...
    "ValidationData",{FeaturesValidationNoisy.',BaselineV}, ...
    "Plots","training-progress", ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.1, ...
    "LearnRateDropPeriod",5);
```

Train the network.

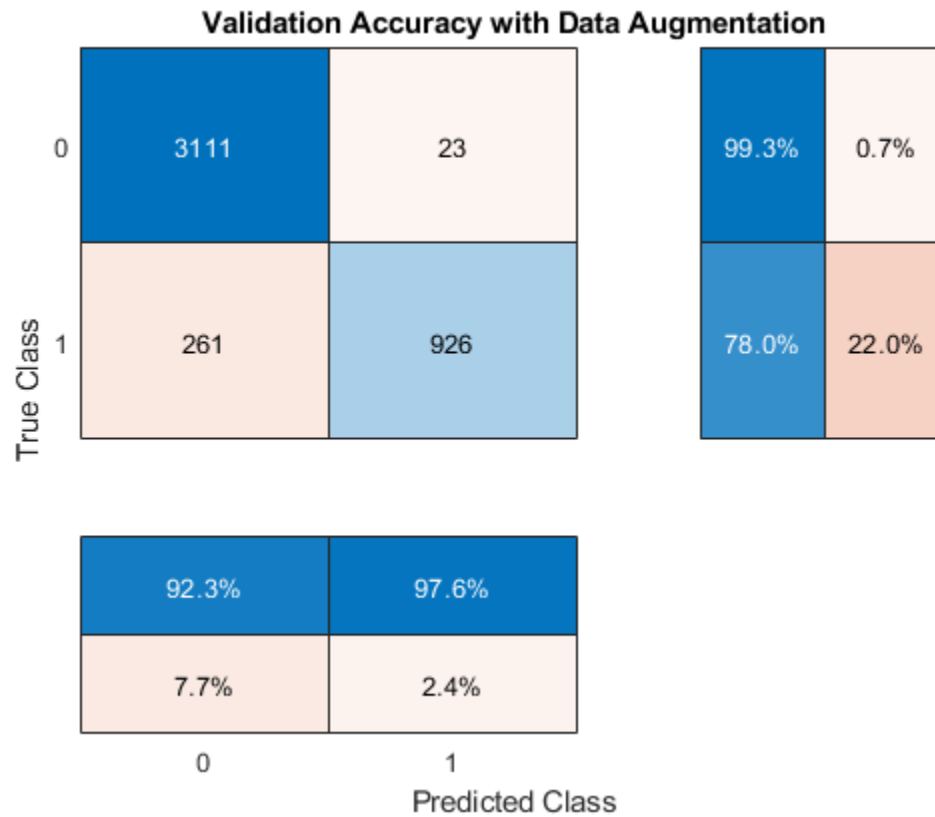
```
[KWSNet,netInfo] = trainNetwork(TrainingFeatures,TrainingMasks,layers,options);
if reduceDataset
    load('KWSNet.mat','KWSNet');
end
```

Verify the network accuracy on the validation signal.

```
v = classify(KWSNet,FeaturesValidationNoisy.');
```

Compare the estimated and expected KWS masks.

```
figure
cm = confusionchart(BaselineV,v,"title","Validation Accuracy with Data Augmentation");
cm.ColumnSummary = "column-normalized";
cm.RowSummary = "row-normalized";
```



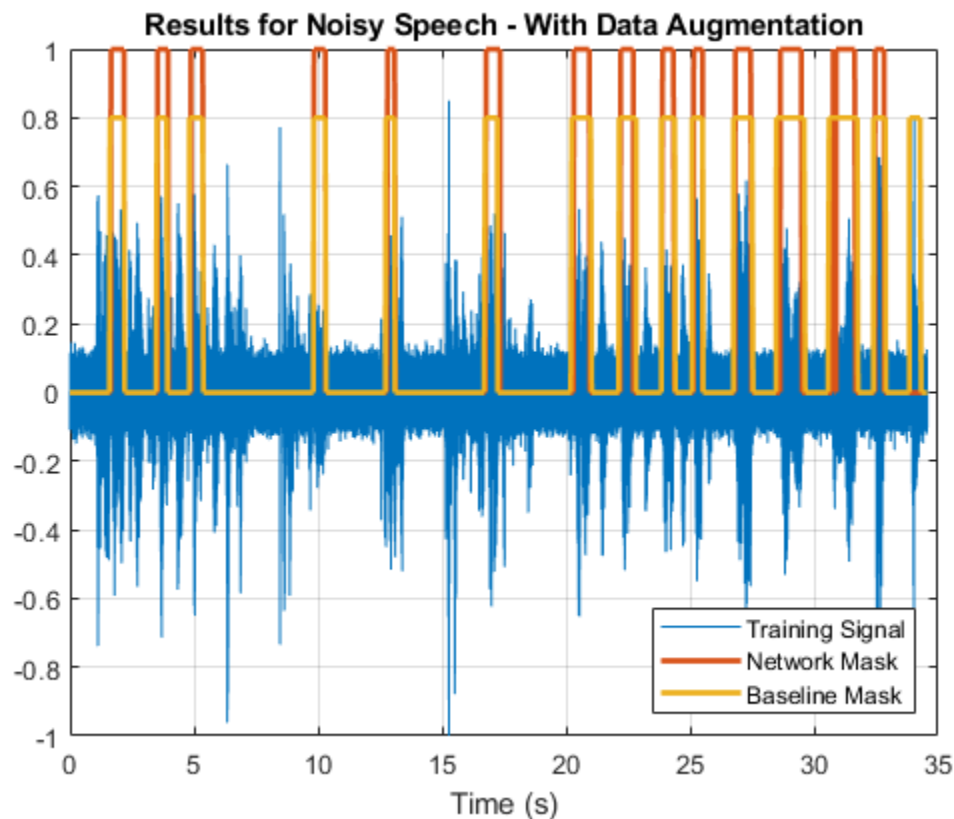
Listen to the identified keyword regions.

```
v = double(v) - 1;
v = repmat(v,HopLength,1);
v = v(:);

sound(audioIn(logical(v)),fs)
```

Visualize the estimated and expected masks.

```
fig = figure;
plot(t,[audioInNoisy(1:length(v)),v,0.8*baseline])
grid on
xlabel('Time (s)')
legend('Training Signal','Network Mask','Baseline Mask','Location','southeast')
l = findall(fig,'type','line');
l(1).LineWidth = 2;
l(2).LineWidth = 2;
title('Results for Noisy Speech - With Data Augmentation')
```



References

[1] Warden P. "Speech Commands: A public dataset for single-word speech recognition", 2017. Available from https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz. Copyright Google 2017. The Speech Commands Dataset is licensed under the Creative Commons Attribution 4.0 license.

Appendix - Helper Functions

```
function [sentence,mask] = HelperSynthesizeSentence(ads_keyword,ads_other,fs,minlength)

% Read one keyword
keyword = read(ads_keyword);
keyword = keyword ./ max(abs(keyword));

% Identify region of interest
speechIndices = detectSpeech(keyword,fs);
if isempty(speechIndices) || diff(speechIndices(1,:)) <= minlength
    speechIndices = [1,length(keyword)];
end
keyword = keyword(speechIndices(1,1):speechIndices(1,2));

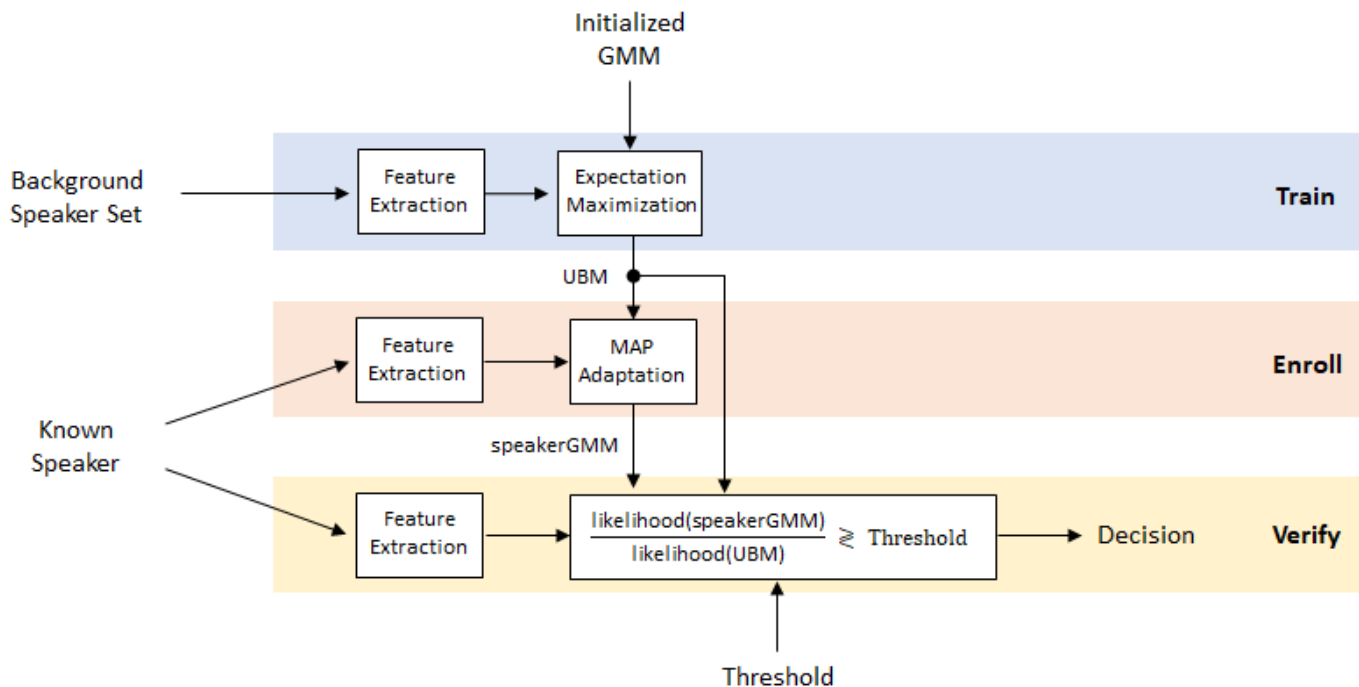
% Pick a random number of other words (between 0 and 10)
numWords = randi([0 10]);
% Pick where to insert keyword
loc = randi([1 numWords+1]);
sentence = [];
```

```
mask = [];  
for index = 1:numWords+1  
    if index==loc  
        sentence = [sentence;keyword];  
        newMask = ones(size(keyword));  
        mask = [mask ;newMask];  
    else  
        other = read(ads_other);  
        other = other ./ max(abs(other));  
        sentence = [sentence;other];  
        mask = [mask;zeros(size(other))];  
    end  
end  
end
```

Speaker Verification Using Gaussian Mixture Model

Speaker verification, or authentication, is the task of verifying that a given speech segment belongs to a given speaker. In speaker verification systems, there is an unknown set of all other speakers, so the likelihood that an utterance belongs to the verification target is compared to the likelihood that it does not. This contrasts with speaker identification tasks, where the likelihood of each speaker is calculated, and those likelihoods are compared. Both speaker verification and speaker identification can be text dependent or text independent. In this example, you create a text-dependent speaker verification system using a Gaussian mixture model/universal background model (GMM-UBM).

A sketch of the GMM-UBM system is shown:



Perform Speaker Verification

To motivate this example, you will first perform speaker verification using a pre-trained universal background model (UBM). The model was trained using the word "stop" from the Google Speech Commands data set [1] on page 1-0.

The MAT file, `speakerVerificationExampleData.mat`, includes the UBM, a configured `audioFeatureExtractor` object, and normalization factors used to normalize the features.

```
load speakerVerificationExampleData.mat ubm afe normFactors
```

Enroll

If you would like to test enrolling yourself, set `enrollYourself` to `true`. You will be prompted to record yourself saying "stop" several times. Say "stop" only once per prompt. Increasing the number of recordings should increase the verification accuracy.


```

enrollYourself =  ;
if enrollYourself
    numToRecord = 5  ;
    ID =  ;
    helperAddUser(afe.SampleRate,numToRecord,ID);
end

```

Create an `audioDatastore` object to point to the five audio files included with this example, and, if you enrolled yourself, the audio files you just recorded. The audio files included with this example are part of an internally created data set and were not used to train the UBM.

```
ads = audioDatastore(pwd);
```

The files included with this example consist of the word "stop" spoken five times by three different speakers: BFn (1), BHm (3), and RPalanim (1). The file names are in the format *SpeakerID_RecordingNumber*. Set the datastore labels to the corresponding speaker ID.

```

[~,fileName] = cellfun(@(x)fileparts(x),ads.Files,'UniformOutput',false);
fileName = split(fileName,'_');
speaker = strcat(fileName(:,1));
ads.Labels = categorical(speaker);

```

Use all but one file from the speaker you are enrolling for the enrollment process. The remaining files are used to test the system.

```

if enrollYourself
    enrollLabel = ID;
else
    enrollLabel = 'BHm';
end

forEnrollment = ads.Labels==enrollLabel;
forEnrollment(find(forEnrollment==1,1)) = false;
adsEnroll = subset(ads,forEnrollment);
adsTest = subset(ads,~forEnrollment);

```

Enroll the chosen speaker using maximum a posteriori (MAP) adaptation. You can find details of the enrollment algorithm later in the example on page 1-0 .

```
speakerGMM = helperEnroll(ubm,afe,normFactors,adsEnroll);
```

Verification

For each of the files in the test set, use the likelihood ratio test and a threshold to determine whether the speaker is the enrolled speaker or an imposter.

```

threshold = 0.7  ;
reset(adsTest)
while hasdata(adsTest)
    fprintf('Identity to confirm: %s\n',enrollLabel)
    [audioData,adsInfo] = read(adsTest);

    fprintf(' | Speaker identity: %s\n',string(adsInfo.Label))
end

```

```
verificationStatus = helperVerify(audioData,afe,normFactors,speakerGMM,ubm,threshold);  
  
if verificationStatus  
    fprintf(' | Confirmed.\n');  
else  
    fprintf(' | Imposter!\n');  
end  
end
```

Identity to confirm: BHm

| Speaker identity: BFn

| Imposter!

Identity to confirm: BHm

| Speaker identity: BHm

| Confirmed.

Identity to confirm: BHm

| Speaker identity: RPalanim

| Imposter!

The remainder of the example details the creation of the UBM and the enrollment algorithm, and then evaluates the system using commonly reported metrics.

Create Universal Background Model

The UBM used in this example is trained using [1] on page 1-0 . Download and extract the data set.

```
url = 'https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz';  
  
downloadFolder = tempdir;  
datasetFolder = fullfile(downloadFolder,'google_speech');  
  
if ~exist(datasetFolder,'dir')  
    disp('Downloading Google speech commands data set (1.9 GB)...')  
    untar(url,datasetFolder)  
end
```

Create an `audioDatastore` that points to the dataset. Use the folder names as the labels. The folder names indicate the words spoken in the dataset.

```
ads = audioDatastore(datasetFolder,"Includesubfolders",true,'LabelSource','folderNames');
```

subset the dataset to only include the word "stop".

```
ads = subset(ads,ads.Labels==categorical("stop"));
```

Set the labels to the unique speaker IDs encoded in the file names. The speaker IDs sometimes start with a number: add an 'a' to all the IDs to make the names more variable friendly.

```
[~,fileName] = cellfun(@(x)fileparts(x),ads.Files,'UniformOutput',false);  
fileName = split(fileName,'_');
```

```
speaker = strcat('a',fileName(:,1));
ads.Labels = categorical(speaker);
```

Create three datastores: one for enrollment, one for evaluating the verification system, and one for training the UBM. Enroll speakers who have at least three utterances. For each of the speakers, place two of the utterances in the enrollment set. The others will go in the test set. The test set consists of utterances from all speakers who have three or more utterances in the dataset. The UBM training set consists of the remaining utterances.

```
numSpeakersToEnroll = 10 ;
labelCount = countEachLabel(ads);
forEnrollAndTestSet = labelCount(:,1){labelCount(:,2)>=3};
forEnroll = forEnrollAndTestSet(randi([1,numel(forEnrollAndTestSet)],numSpeakersToEnroll,1));
tf = ismember(ads.Labels,forEnroll);
adsEnrollAndValidate = subset(ads,tf);
adsEnroll = splitEachLabel(adsEnrollAndValidate,2);

adsTest = subset(ads,ismember(ads.Labels,forEnrollAndTestSet));
adsTest = subset(adsTest,~ismember(adsTest.Files,adsEnroll.Files));

forUBMTraining = ~(ismember(ads.Files,adsTest.Files) | ismember(ads.Files,adsEnroll.Files));
adsTrainUBM = subset(ads,forUBMTraining);
```

Read from the training datastore and listen to a file. Reset the datastore.

```
[audioData,audioInfo] = read(adsTrainUBM);
fs = audioInfo.SampleRate;

sound(audioData,fs)

reset(adsTrainUBM)
```

Feature Extraction

In the feature extraction pipeline for this example, you:

- 1 Normalize the audio
- 2 Use `detectSpeech` to remove nonspeech regions from the audio
- 3 Extract features from the audio
- 4 Normalize the features
- 5 Apply cepstral mean normalization

First, create an `audioFeatureExtractor` object to extract the MFCC. Specify a 40 ms duration and 10 ms hop for the frames.

```
windowDuration = 0.04;
hopDuration = 0.01;
windowSamples = round(windowDuration*fs);
hopSamples = round(hopDuration*fs);
overlapSamples = windowSamples - hopSamples;

afe = audioFeatureExtractor( ...
    'SampleRate',fs, ...
    'Window',hann(windowSamples,'periodic'), ...
    'OverlapLength',overlapSamples, ...
```

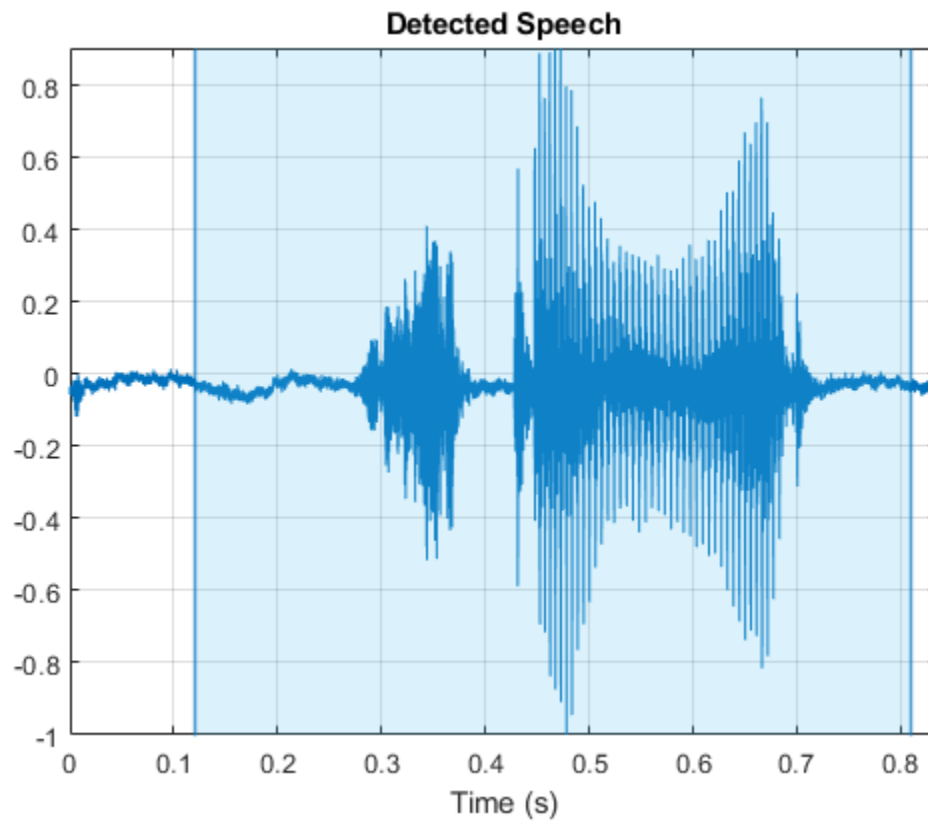
```
...  
'mfcc',true);
```

Normalize the audio.

```
audioData = audioData./max(abs(audioData));
```

Use the `detectSpeech` function to locate the region of speech in the audio clip. Call `detectSpeech` without any output arguments to visualize the detected region of speech.

```
detectSpeech(audioData,fs);
```



Call `detectSpeech` again. This time, return the indices of the speech region and use them to remove nonspeech regions from the audio clip.

```
idx = detectSpeech(audioData,fs);  
audioData = audioData(idx(1,1):idx(1,2));
```

Call `extract` on the `audioFeatureExtractor` object to extract features from audio data. The size output from `extract` is `numHops`-by-`numFeatures`.

```
features = extract(afe,audioData);  
[numHops,numFeatures] = size(features)
```

```
numHops = 66
```

```
numFeatures = 13
```

Normalize the features by their global mean and variance. The next section of the example walks through calculating the global mean and variance. For now, just use the precalculated mean and variance already loaded.

```
features = (features' - normFactors.Mean) ./ normFactors.Variance;
```

Apply a local cepstral mean normalization.

```
features = features - mean(features, 'all');
```

The feature extraction pipeline is encapsulated in the helper function, `helperFeatureExtraction` on page 1-0 .

Calculate Global Feature Normalization Factors

Extract all features from the data set. If you have the Parallel Computing Toolbox™, determine the optimal number of partitions for the dataset and spread the computation across available workers. If you do not have Parallel Computing Toolbox™, use a single partition.

```
featuresAll = {};
if ~isempty(ver('parallel'))
    numPar = 18;
else
    numPar = 1;
end
```

Use the helper function, `helperFeatureExtraction`, to extract all features from the dataset. Calling `helperFeatureExtraction` with an empty third argument performs the feature extraction steps described in Feature Extraction on page 1-0 except for the normalization by global mean and variance.

```
parfor ii = 1:numPar
    adsPart = partition(ads,numPar,ii);
    featuresPart = cell(0,numel(adsPart.Files));
    for iiii = 1:numel(adsPart.Files)
        audioData = read(adsPart);
        featuresPart{iiii} = helperFeatureExtraction(audioData,afe,[]);
    end
    featuresAll = [featuresAll,featuresPart];
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
allFeatures = cat(2,featuresAll{:});
```

Calculate the mean and variance of each feature.

```
normFactors.Mean = mean(allFeatures,2,'omitnan');
normFactors.STD = std(allFeatures,[],2,'omitnan');
```

Initialize GMM

The universal background model is a Gaussian mixture model. Define the number of components in the mixture. [2] on page 1-0 suggests more than 512 for text-independent systems. The component weights begin evenly distributed.

```
numComponents = 32 ;  
alpha = ones(1,numComponents)/numComponents;
```

Use random initialization for the `mu` and `sigma` of each GMM component. Create a structure to hold the necessary UBM information.

```
mu = randn(numFeatures,numComponents);  
sigma = rand(numFeatures,numComponents);  
ubm = struct('ComponentProportion',alpha,'mu',mu,'sigma',sigma);
```

Train UBM Using Expectation-Maximization

Fit the GMM to the training set to create the UBM. Use the expectation-maximization algorithm.

The expectation-maximization algorithm is recursive. First, define the stopping criteria.

```
maxIter = 20;  
targetLogLikelihood = 0;  
tol = 0.5;  
pastL = -inf; % initialization of previous log-likelihood
```

In a loop, train the UBM using the expectation-maximization algorithm.

```
tic  
for iter = 1:maxIter  
    % EXPECTATION  
    N = zeros(1,numComponents);  
    F = zeros(numFeatures,numComponents);  
    S = zeros(numFeatures,numComponents);  
    L = 0;  
    parfor ii = 1:numPar  
        adsPart = partition(adsTrainUBM,numPar,ii);  
        while hasdata(adsPart)  
            audioData = read(adsPart);  
  
            % Extract features  
            features = helperFeatureExtraction(audioData,afe,normFactors);  
  
            % Compute a posteriori log-likelihood  
            logLikelihood = helperGMMLogLikelihood(features,ubm);  
  
            % Compute a posteriori normalized probability  
            logLikelihoodSum = helperLogSumExp(logLikelihood);  
            gamma = exp(logLikelihood - logLikelihoodSum)';  
  
            % Compute Baum-Welch statistics  
            n = sum(gamma,1);  
            f = features * gamma;  
            s = (features.*features) * gamma;  
  
            % Update the sufficient statistics over utterances  
            N = N + n;  
            F = F + f;  
            S = S + s;  
  
            % Update the log-likelihood
```

```

        L = L + sum(logLikelihoodSum);
    end
end

% Print current log-likelihood and stop if it meets criteria.
L = L/numel(adsTrainUBM.Files);
fprintf('\tIteration %d, Log-likelihood = %0.3f\n',iter,L)
if L > targetLogLikelihood || abs(pastL - L) < tol
    break
else
    pastL = L;
end

% MAXIMIZATION
N = max(N,eps);
ubm.ComponentProportion = max(N/sum(N),eps);
ubm.ComponentProportion = ubm.ComponentProportion/sum(ubm.ComponentProportion);
ubm.mu = bsxfun(@rdivide,F,N);
ubm.sigma = max(bsxfun(@rdivide,S,N) - ubm.mu.^2,eps);
end

Iteration 1, Log-likelihood = -826.174
Iteration 2, Log-likelihood = -538.546
Iteration 3, Log-likelihood = -522.670
Iteration 4, Log-likelihood = -517.458
Iteration 5, Log-likelihood = -514.852
Iteration 6, Log-likelihood = -513.068
Iteration 7, Log-likelihood = -511.644
Iteration 8, Log-likelihood = -510.588
Iteration 9, Log-likelihood = -509.788
Iteration 10, Log-likelihood = -509.135
Iteration 11, Log-likelihood = -508.529
Iteration 12, Log-likelihood = -508.032

fprintf('UBM training completed in %0.2f seconds.\n',toc)

UBM training completed in 32.31 seconds.

```

Enrollment: Maximum a Posteriori (MAP) Estimation

Once you have a universal background model, you can enroll speakers and adapt the UBM to the speakers. [2] on page 1-0 suggests an adaptation relevance factor of 16. The relevance factor controls how much to move each component of the UBM to the speaker GMM.

```

relevanceFactor = 16;

speakers = unique(adsEnroll.Labels);
numSpeakers = numel(speakers);

gmmCellArray = cell(numSpeakers,1);
tic
parfor ii = 1:numSpeakers
    % Subset the datastore to the speaker you are adapting.
    adsTrainSubset = subset(adsEnroll,adsEnroll.Labels==speakers(ii));

    N = zeros(1,numComponents);
    F = zeros(numFeatures,numComponents);
    S = zeros(numFeatures,numComponents);

```

```
while hasdata(adsTrainSubset)
    audioData = read(adsTrainSubset);
    features = helperFeatureExtraction(audioData,afe,normFactors);
    [n,f,s,l] = helperExpectation(features,ubm);
    N = N + n;
    F = F + f;
    S = S + s;
end

% Determine the maximum likelihood
gmm = helperMaximization(N,F,S);

% Determine adaption coefficient
alpha = N ./ (N + relevanceFactor);

% Adapt the means
gmm.mu = alpha.*gmm.mu + (1-alpha).*ubm.mu;

% Adapt the variances
gmm.sigma = alpha.*(S./N) + (1-alpha).*(ubm.sigma + ubm.mu.^2) - gmm.mu.^2;
gmm.sigma = max(gmm.sigma,eps);

% Adapt the weights
gmm.ComponentProportion = alpha.*(N/sum(N)) + (1-alpha).*ubm.ComponentProportion;
gmm.ComponentProportion = gmm.ComponentProportion./sum(gmm.ComponentProportion);

gmmCellArray{ii} = gmm;
end
fprintf('Enrollment completed in %0.2f seconds.\n',toc)

Enrollment completed in 0.27 seconds.
```

For bookkeeping purposes, convert the cell array of GMMs to a struct, with the fields being the speaker IDs and the values being the GMM structs.

```
for i = 1:numel(gmmCellArray)
    enrolledGMMs.(string(speakers(i))) = gmmCellArray{i};
end
```

Evaluation

Speaker False Rejection Rate

The speaker false rejection rate (FRR) is the rate that a given speaker is incorrectly rejected. Use the known speaker set to determine the speaker false rejection rate for a set of thresholds.

```
speakers = unique(adsEnroll.Labels);
numSpeakers = numel(speakers);
llr = cell(numSpeakers,1);
tic
parfor speakerIdx = 1:numSpeakers
    localGMM = enrolledGMMs.(string(speakers(speakerIdx)));
    adsTestSubset = subset(adsTest,adsTest.Labels==speakers(speakerIdx));
    llrPerSpeaker = zeros(numel(adsTestSubset.Files),1);
    for fileIdx = 1:numel(adsTestSubset.Files)
        audioData = read(adsTestSubset);
        [x,numFrames] = helperFeatureExtraction(audioData,afe,normFactors);
```



```

logLikelihood = helperGMMLogLikelihood(x,localGMM);
Lspeaker = helperLogSumExp(logLikelihood);

logLikelihood = helperGMMLogLikelihood(x,ubm);
Lubm = helperLogSumExp(logLikelihood);

llrPerSpeaker(fileIdx) = mean(movmedian(Lspeaker - Lubm,3));
end
llr{speakerIdx} = llrPerSpeaker;
end
fprintf('False rejection rate computed in %0.2f seconds.\n',toc)

```

False rejection rate computed in 0.20 seconds.

Plot the false rejection rate as a function of the threshold.

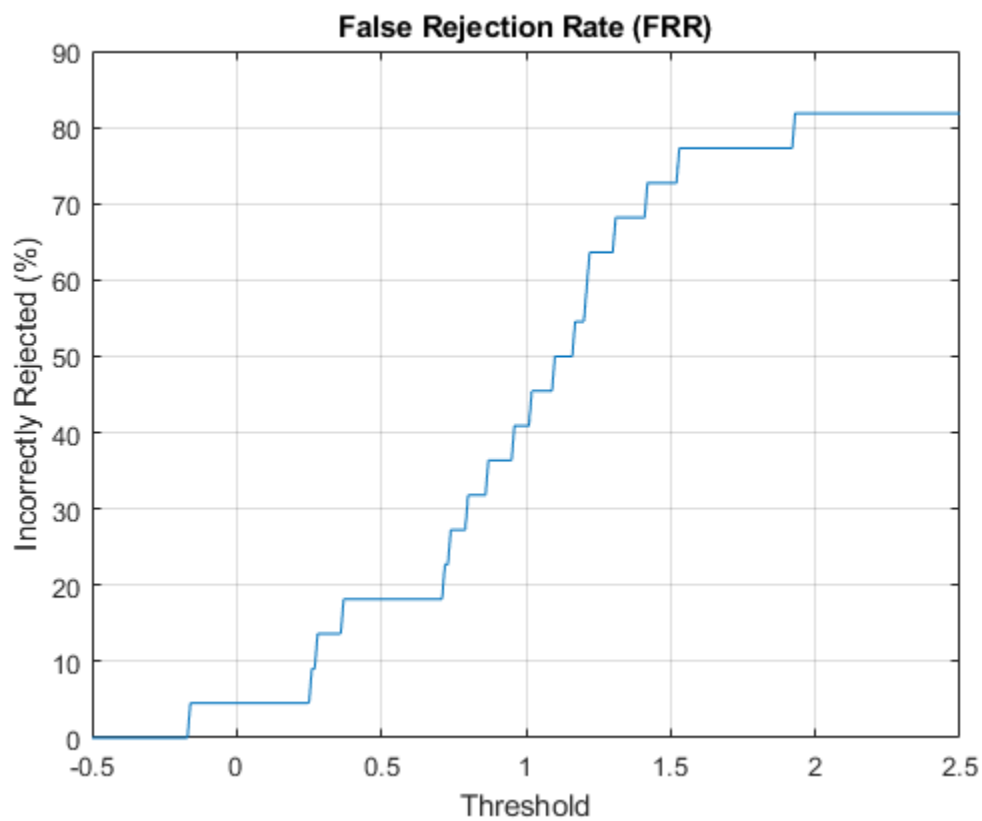
```

llr = cat(1,llr{:});

thresholds = -0.5:0.01:2.5;
FRR = mean(llr<thresholds);

plot(thresholds,FRR*100)
title('False Rejection Rate (FRR)')
xlabel('Threshold')
ylabel('Incorrectly Rejected (%)')
grid on

```



Speaker False Acceptance

The speaker false acceptance rate (FAR) is the rate that utterances not belonging to an enrolled speaker are incorrectly accepted as belonging to the enrolled speaker. Use the known speaker set to determine the speaker FAR for a set of thresholds. Use the same set of thresholds used to determine FRR.

```
speakersTest = unique(adsTest.Labels);
llr = cell(numSpeakers,1);
tic
parfor speakerIdx = 1:numel(speakers)
    localGMM = enrolledGMMs.(string(speakers(speakerIdx)));
    adsTestSubset = subset(adsTest,adsTest.Labels~=speakers(speakerIdx));
    llrPerSpeaker = zeros(numel(adsTestSubset.Files),1);
    for fileIdx = 1:numel(adsTestSubset.Files)
        audioData = read(adsTestSubset);
        [x,numFrames] = helperFeatureExtraction(audioData,afe,normFactors);

        logLikelihood = helperGMMLogLikelihood(x,localGMM);
        Lspeaker = helperLogSumExp(logLikelihood);

        logLikelihood = helperGMMLogLikelihood(x,ubm);
        Lubm = helperLogSumExp(logLikelihood);

        llrPerSpeaker(fileIdx) = mean(movmedian(Lspeaker - Lubm,3));
    end
    llr{speakerIdx} = llrPerSpeaker;
end
fprintf('FAR computed in %0.2f seconds.\n',toc)

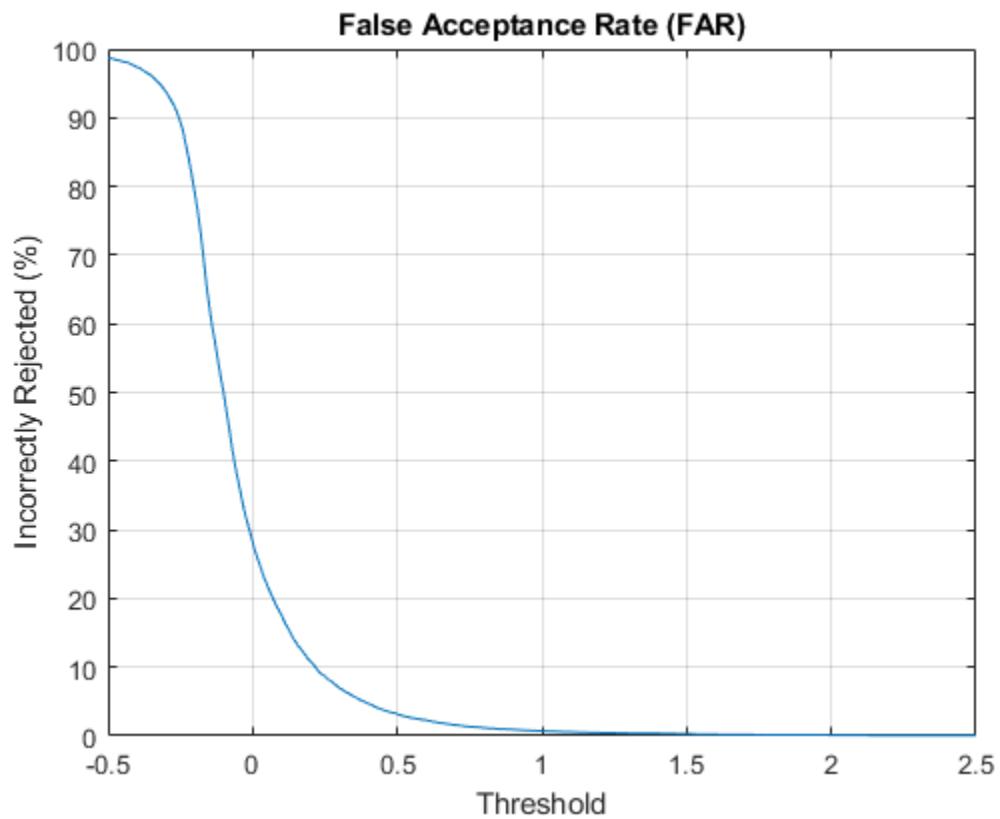
FAR computed in 22.64 seconds.
```

Plot the FAR as a function of the threshold.

```
llr = cat(1,llr{:});

FAR = mean(llr>thresholds);

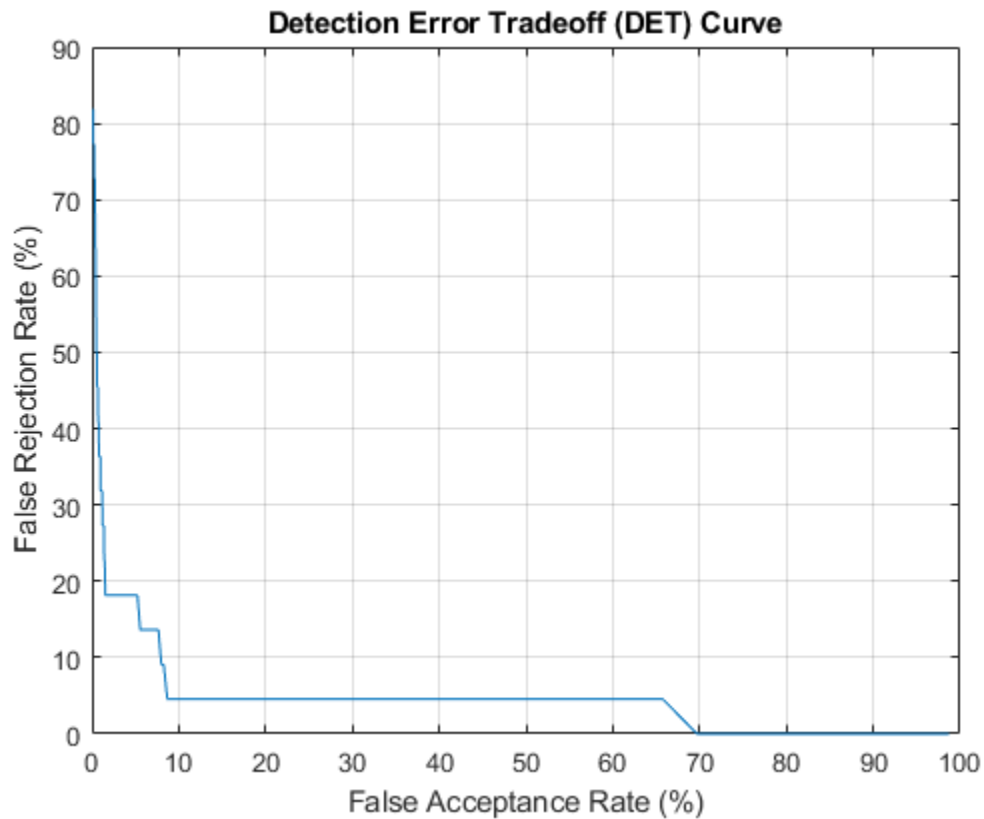
plot(thresholds,FAR*100)
title('False Acceptance Rate (FAR)')
xlabel('Threshold')
ylabel('Incorrectly Rejected (%)')
grid on
```



Detection Error Tradeoff (DET)

As you move the threshold in a speaker verification system, you trade off between FAR and FRR. This is referred to as the detection error tradeoff (DET) and is commonly reported for binary classification problems.

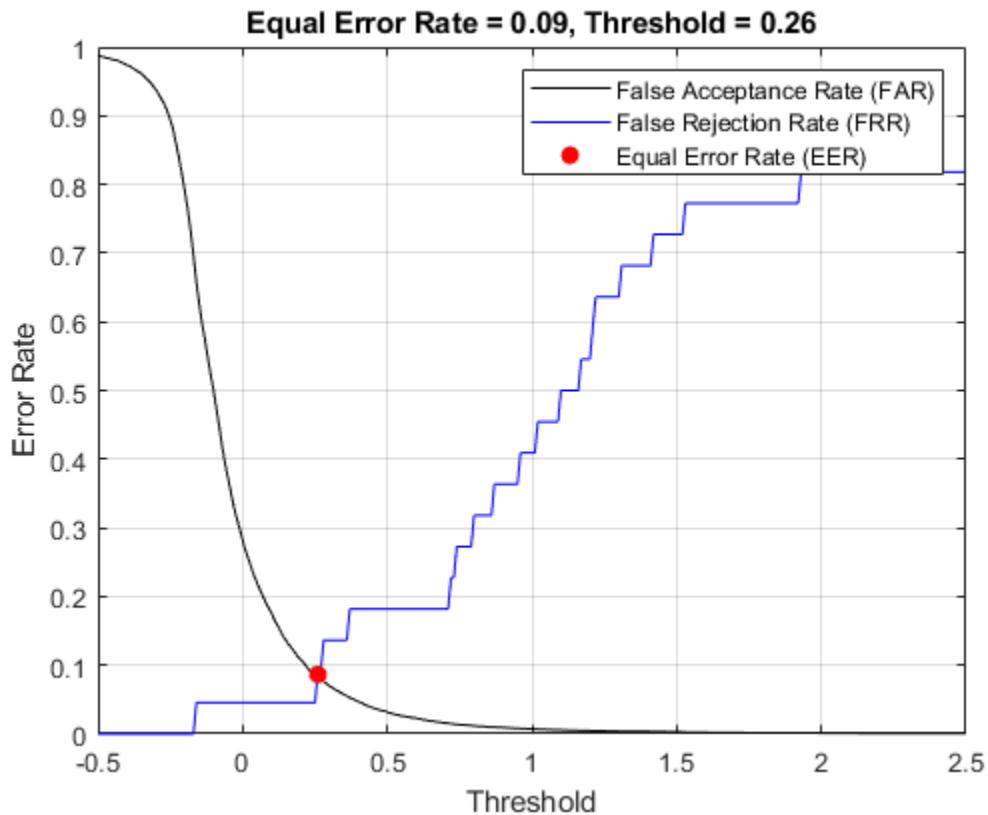
```
x1 = FAR*100;  
y1 = FRR*100;  
plot(x1,y1)  
grid on  
xlabel('False Acceptance Rate (%)')  
ylabel('False Rejection Rate (%)')  
title('Detection Error Tradeoff (DET) Curve')
```



Equal Error Rate (EER)

To compare multiple systems, you need a single metric that combines the FAR and FRR performances. For this, you determine the equal error rate (EER), which is the threshold where the FAR and FRR curves meet. In practice, the EER threshold may not be the best choice. For example, if speaker verification is used as part of a multi-authentication approach for wire transfers, FAR would most likely be weighed more heavily than FRR.

```
[~,EERThresholdIdx] = min(abs(FAR - FRR));
EERThreshold = thresholds(EERThresholdIdx);
EER = mean([FAR(EERThresholdIdx),FRR(EERThresholdIdx)]);
plot(thresholds,FAR,'k', ...
      thresholds,FRR,'b', ...
      EERThreshold,EER,'ro','MarkerFaceColor','r')
title(sprintf('Equal Error Rate = %0.2f, Threshold = %0.2f',EER,EERThreshold))
xlabel('Threshold')
ylabel('Error Rate')
legend('False Acceptance Rate (FAR)','False Rejection Rate (FRR)','Equal Error Rate (EER)')
grid on
```



If you changed parameters of the UBM training, consider resaving the MAT file with the new universal background model, `audioFeatureExtractor`, and norm factors.

```
resave =  ;
if resave
    save('speakerVerificationExampleData.mat','ubm','afe','normFactors')
end
```

Supporting Functions

Add User to Data Set

```
function helperAddUser(fs,numToRecord,ID)
% Create an audio device reader to read from your audio device
deviceReader = audioDeviceReader('SampleRate',fs);

% Initialize variables
numRecordings = 1;
audioIn = [];

% Record the requested number
while numRecordings <= numToRecord
    fprintf('Say "stop" once (recording %i of %i) ...',numRecordings,numToRecord)
    tic
    while toc<2
        audioIn = [audioIn;deviceReader()];
    end
end
```

```
fprintf('complete.\n')
idx = detectSpeech(audioIn,fs);
if isempty(idx)
    fprintf('Speech not detected. Try again.\n')
else
    audiowrite(sprintf('%s_%i.flac',ID,numRecordings),audioIn,fs)
    numRecordings = numRecordings+1;
end
pause(0.2)
audioIn = [];
end

% Release the device
release(deviceReader)
end
```

Enroll

```
function speakerGMM = helperEnroll(ubm,afe,normFactors,adsEnroll)
% Initialization
numComponents = numel(ubm.ComponentProportion);
numFeatures = size(ubm.mu,1);
N = zeros(1,numComponents);
F = zeros(numFeatures,numComponents);
S = zeros(numFeatures,numComponents);
NumFrames = 0;

while hasdata(adsEnroll)
    % Read from the enrollment datastore
    audioData = read(adsEnroll);

    % 1. Extract the features and apply feature normalization
    [features,numFrames] = helperFeatureExtraction(audioData,afe,normFactors);

    % 2. Calculate the a posteriori probability. Use it to determine the
    % sufficient statistics (the count, and the first and second moments)
    [n,f,s] = helperExpectation(features,ubm);

    % 3. Update the sufficient statistics
    N = N + n;
    F = F + f;
    S = S + s;
    NumFrames = NumFrames + numFrames;
end

% Create the Gaussian mixture model that maximizes the expectation
speakerGMM = helperMaximization(N,F,S);

% Adapt the UBM to create the speaker model. Use a relevance factor of 16,
% as proposed in [2]
relevanceFactor = 16;

% Determine adaption coefficient
alpha = N ./ (N + relevanceFactor);

% Adapt the means
speakerGMM.mu = alpha.*speakerGMM.mu + (1-alpha).*ubm.mu;

% Adapt the variances
```

```
speakerGMM.sigma = alpha.*(S./N) + (1-alpha).*(ubm.sigma + ubm.mu.^2) - speakerGMM.mu.^2;
speakerGMM.sigma = max(speakerGMM.sigma,eps);
```

```
% Adapt the weights
```

```
speakerGMM.ComponentProportion = alpha.*(N/sum(N)) + (1-alpha).*ubm.ComponentProportion;
speakerGMM.ComponentProportion = speakerGMM.ComponentProportion./sum(speakerGMM.ComponentProportion);
end
```

Verify

```
function verificationStatus = helperVerify(audioData,afe,normFactors,speakerGMM,ubm,threshold)
    % Extract features
    x = helperFeatureExtraction(audioData,afe,normFactors);

    % Determine the log-likelihood the audio came from the GMM adapted to
    % the speaker
    post = helperGMMLogLikelihood(x,speakerGMM);
    Lspeaker = helperLogSumExp(post);

    % Determine the log-likelihood the audio came from the GMM fit to all
    % speakers
    post = helperGMMLogLikelihood(x,ubm);
    Lubm = helperLogSumExp(post);

    % Calculate the ratio for all frames. Apply a moving median filter
    % to remove outliers, and then take the mean across the frames
    llr = mean(movmedian(Lspeaker - Lubm,3));

    if llr > threshold
        verificationStatus = true;
    else
        verificationStatus = false;
    end
end
```

Feature Extraction

```
function [features,numFrames] = helperFeatureExtraction(audioData,afe,normFactors)
    % Normalize
    audioData = audioData/max(abs(audioData(:)));

    % Protect against NaNs
    audioData(isnan(audioData)) = 0;

    % Isolate speech segment
    % The dataset used in this example has one word per audioData, if more
    % than one is speech section is detected, just use the longest
    % detected.
    idx = detectSpeech(audioData,afe.SampleRate);
    if size(idx,1)>1
        [~,seg] = max(idx(:,2) - idx(:,1));
    else
        seg = 1;
    end
    audioData = audioData(idx(seg,1):idx(seg,2));

    % Feature extraction
    features = extract(afe,audioData);
```

```
% Feature normalization
if ~isempty(normFactors)
    features = (features-normFactors.Mean')./normFactors.STD';
end
features = features';

% Cepstral mean subtraction (for channel noise)
if ~isempty(normFactors)
    features = features - mean(features,'all');
end

numFrames = size(features,2);
end
```

Log-sum-exponent

```
function y = helperLogSumExp(x)
% Calculate the log-sum-exponent while avoiding overflow
a = max(x,[],1);
y = a + sum(exp(bsxfun(@minus,x,a)),1);
end
```

Expectation

```
function [N,F,S,L] = helperExpectation(features,gmm)

post = helperGMMLogLikelihood(features,gmm);

% Sum the likelihood over the frames
L = helperLogSumExp(post);

% Compute the sufficient statistics
gamma = exp(post-L)';

N = sum(gamma,1);
F = features * gamma;
S = (features.*features) * gamma;
L = sum(L);
end
```

Maximization

```
function gmm = helperMaximization(N,F,S)
    N = max(N,eps);
    gmm.ComponentProportion = max(N/sum(N),eps);
    gmm.mu = bsxfun(@rdivide,F,N);
    gmm.sigma = max(bsxfun(@rdivide,S,N) - gmm.mu.^2,eps);
end
```

Gaussian Multi-Component Mixture Log-Likelihood

```
function L = helperGMMLogLikelihood(x,gmm)
    xMinusMu = repmat(x,1,1,numel(gmm.ComponentProportion)) - permute(gmm.mu,[1,3,2]);
    permuteSigma = permute(gmm.sigma,[1,3,2]);

    Lunweighted = -0.5*(sum(log(permuteSigma),1) + sum(bsxfun(@times,xMinusMu,(bsxfun(@rdivide,xMinusMu,permuteSigma)),1)));
    temp = squeeze(permute(Lunweighted,[1,3,2]));
```



```
if size(temp,1)==1
    % If there is only one frame, the trailing singleton dimension was
    % removed in the permute. This accounts for that edge case
    temp = temp';
end
L = bsxfun(@plus,temp,log(gmm.ComponentProportion)');
end
```

References

- [1] Warden P. "Speech Commands: A public dataset for single-word speech recognition", 2017. Available from https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz. Copyright Google 2017. The Speech Commands Dataset is licensed under the Creative Commons Attribution 4.0 license, available here: <https://creativecommons.org/licenses/by/4.0/legalcode>.
- [2] Reynolds, Douglas A., Thomas F. Quatieri, and Robert B. Dunn. "Speaker Verification Using Adapted Gaussian Mixture Models." *Digital Signal Processing* 10, no. 1-3 (2000): 19-41. <https://doi.org/10.1006/dspr.1999.0361>.

Sequential Feature Selection for Audio Features

This example shows a typical workflow for feature selection applied to the task of spoken digit recognition.

In sequential feature selection, you train a network on a given feature set and then incrementally add or remove features until the highest accuracy is reached [1] on page 1-0 . In this example, you apply sequential forward selection to the task of spoken digit recognition using the Free Spoken Digit Dataset [2] on page 1-0 .

Streaming Spoken Digit Recognition

To motivate the example, begin by loading a pretrained network, the `audioFeatureExtractor` object used to train the network, and normalization factors for the features.

```
load('network_Audio_SequentialFeatureSelection.mat','bestNet','afe','normalizers');
```

Create an `audioDeviceReader` to read audio from a microphone. Create three `dsp.AsyncBuffer` objects: one to buffer audio read from your microphone, one to buffer short-term energy of the input audio for speech detection, and one to buffer predictions.

```
fs = afe.SampleRate;

deviceReader = audioDeviceReader('SampleRate',fs,'SamplesPerFrame',256);

audioBuffer = dsp.AsyncBuffer(fs*3);
steBuffer = dsp.AsyncBuffer(1000);
predictionBuffer = dsp.AsyncBuffer(5);
```

Create a plot to display the streaming audio, the probability the network outputs during inference, and the prediction.

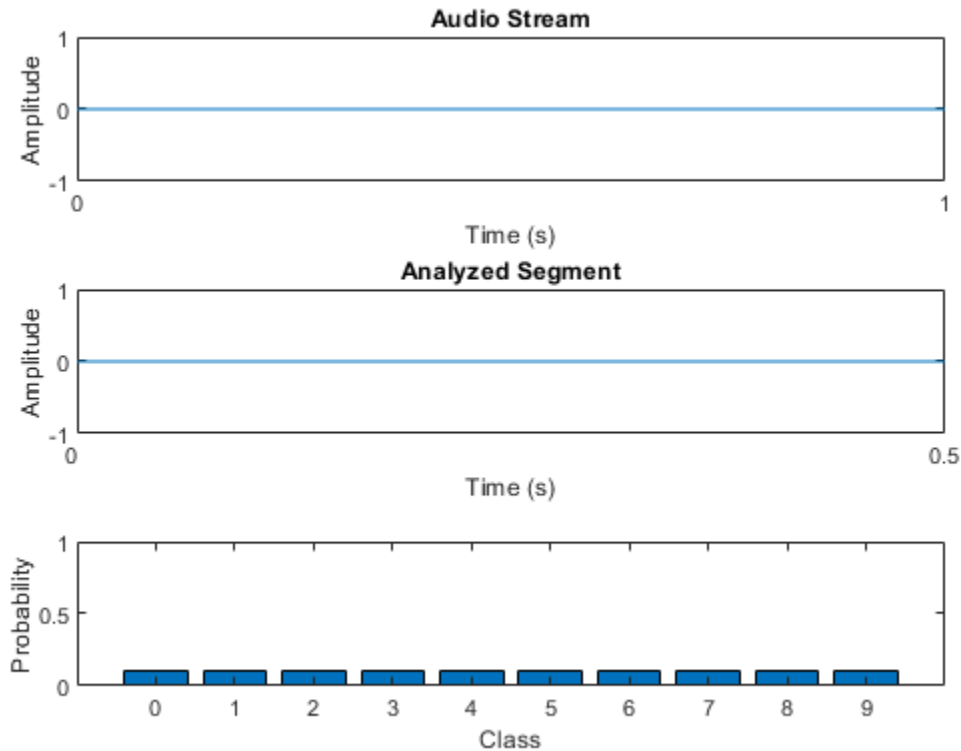
```
fig = figure;

streamAxes = subplot(3,1,1);
streamPlot = plot(zeros(fs,1));
ylabel('Amplitude')
xlabel('Time (s)')
title('Audio Stream')
streamAxes.XTick = [0,fs];
streamAxes.XTickLabel = [0,1];
streamAxes.YLim = [-1,1];

analyzedAxes = subplot(3,1,2);
analyzedPlot = plot(zeros(fs/2,1));
title('Analyzed Segment')
ylabel('Amplitude')
xlabel('Time (s)')
set(gca,'XTickLabel',[])
analyzedAxes.XTick = [0,fs/2];
analyzedAxes.XTickLabel = [0,0.5];
analyzedAxes.YLim = [-1,1];

probabilityAxes = subplot(3,1,3);
probabilityPlot = bar(0:9,0.1*ones(1,10));
axis([-1,10,0,1])
```

```
ylabel('Probability')
xlabel('Class')
```



Perform streaming digit recognition (digits 0 through 9) for 20 seconds. While the loop runs, speak one of the digits and test its accuracy.

First, define a short-term energy threshold under which to assume a signal contains no speech.

```
steThreshold = 0.015;
idxVec = 1:fs;
tic
while toc < 20

    % Read in a frame of audio from your device.
    audioIn = deviceReader();

    % Write the audio into a the buffer.
    write(audioBuffer,audioIn);

    % While 200 ms of data is unused, continue this loop.
    while audioBuffer.NumUnreadSamples > 0.2*fs

        % Read 1 second from the audio buffer. Of that 1 second, 800 ms
        % is rereading old data and 200 ms is new data.
        audioToAnalyze = read(audioBuffer,fs,0.8*fs);

        % Update the figure to plot the current audio data.
        streamPlot.YData = audioToAnalyze;
```

```
ste = mean(abs(audioToAnalyze));
write(steBuffer,ste);
if steBuffer.NumUnreadSamples > 5
    abc = sort(peek(steBuffer));
    steThreshold = abc(round(0.4*numel(abc)));
end
if ste > steThreshold

    % Use the detectSpeech function to determine if a region of speech
    % is present.
    idx = detectSpeech(audioToAnalyze,fs);

    % If a region of speech is present, perform the following.
    if ~isempty(idx)
        % Zero out all parts of the signal except the speech
        % region, and trim to 0.5 seconds.
        audioToAnalyze = HelperTrimOrPad(audioToAnalyze(idx(1,1):idx(1,2)),fs/2);

        % Normalize the audio.
        audioToAnalyze = audioToAnalyze/max(abs(audioToAnalyze));

        % Update the analyzed segment plot
        analyzedPlot.YData = audioToAnalyze;

        % Extract the features and transpose them so that time is
        % across columns.
        features = (extract(afe,audioToAnalyze))';

        % Normalize the features.
        features = (features - normalizers.Mean) ./ normalizers.StandardDeviation;

        % Call classify to determine the probabilities and the
        % winning label.
        features(isnan(features)) = 0;
        [label,probs] = classify(bestNet,features);

        % Update the plot with the probabilities and the winning
        % label.
        probabilityPlot.YData = probs;
        write(predictionBuffer,probs);

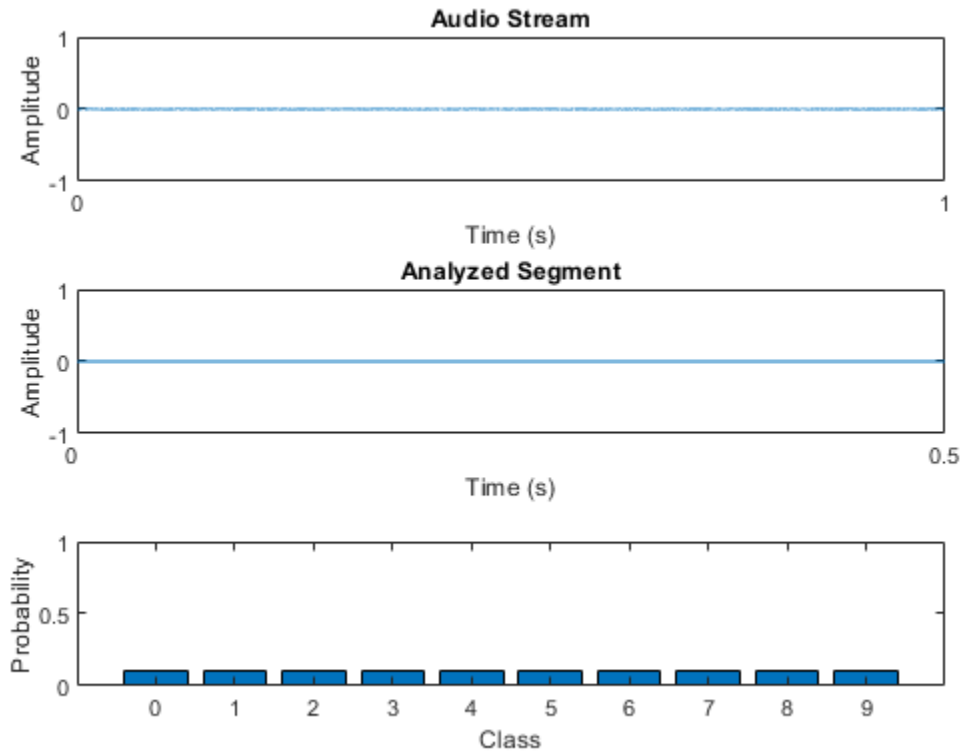
        if predictionBuffer.NumUnreadSamples == predictionBuffer.Capacity
            lastTen = peek(predictionBuffer);
            [~,decision] = max(mean(lastTen.*hann(size(lastTen,1)),1));
            probabilityAxes.Title.String = num2str(decision-1);
        end
    end
else
    % If the signal energy is below the threshold, assume no speech
    % detected.
    probabilityAxes.Title.String = '';
    probabilityPlot.YData = 0.1*ones(10,1);
    analyzedPlot.YData = zeros(fs/2,1);
    reset(predictionBuffer)
end

drawnow limitrate
```

```

end
end

```



The remainder of the example illustrates how the network used in the streaming detection was trained and how the features fed into the network were chosen.

Create Train and Validation Data Sets

Download the Free Spoken Digit Dataset (FSDD) [2] on page 1-0 . FSDD consists of short audio files with spoken digits (0-9).

```

url = "https://zenodo.org/record/1342401/files/Jakobovski/free-spoken-digit-dataset-v1.0.8.zip";
downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder, 'FSDD');

if ~exist(datasetFolder, 'dir')
    fprintf('Downloading Free Spoken Digit Dataset ...\n')
    unzip(url, datasetFolder)
end

```

Create an `audioDatastore` to point to the recordings. Get the sample rate of the data set.

```

ads = audioDatastore(datasetFolder, 'IncludeSubfolders', true);
[~, adsInfo] = read(ads);
fs = adsInfo.SampleRate;

```

The first element of the file names is the digit spoken in the file. Get the first element of the file names, convert them to categorical, and then set the `Labels` property of the `audioDatastore`.

```
[~,filenames] = cellfun(@(x)fileparts(x),ads.Files,'UniformOutput',false);
ads.Labels = categorical(string(cellfun(@(x)x(1),filenames)));
```

To split the datastore into a development set and a validation set, use `splitEachLabel`. Allocate 80% of the data for development and the remaining 20% for validation.

```
[adsTrain,adsValidation] = splitEachLabel(ads,0.8);
```

Set Up Audio Feature Extractor

Create an `audioFeatureExtractor` object to extract audio features over 30 ms windows with an update rate of 10 ms. Set all features you would like to test in this example to `true`.

```
win = hamming(round(0.03*fs),"periodic");
overlapLength = round(0.02*fs);
```

```
afe = audioFeatureExtractor( ...
    'Window', win, ...
    'OverlapLength',overlapLength, ...
    'SampleRate', fs, ...
    ...
    'linearSpectrum', false, ...
    'melSpectrum', false, ...
    'barkSpectrum', false, ...
    'erbSpectrum', false, ...
    ...
    'mfcc', true, ...
    'mfccDelta', true, ...
    'mfccDeltaDelta', true, ...
    'gtcc', true, ...
    'gtccDelta', true, ...
    'gtccDeltaDelta', true, ...
    ...
    'spectralCentroid', true, ...
    'spectralCrest', true, ...
    'spectralDecrease', true, ...
    'spectralEntropy', true, ...
    'spectralFlatness', true, ...
    'spectralFlux', true, ...
    'spectralKurtosis', true, ...
    'spectralRolloffPoint',true, ...
    'spectralSkewness', true, ...
    'spectralSlope', true, ...
    'spectralSpread', true, ...
    ...
    'pitch', false, ...
    'harmonicRatio', false);
```

Define Layers and Training Options

Define the “List of Deep Learning Layers” (Deep Learning Toolbox) and `trainingOptions` (Deep Learning Toolbox) used in this example. The first layer, `sequenceInputLayer` (Deep Learning Toolbox), is just a placeholder. Depending on which features you test during sequential feature selection, the first layer is replaced with a `sequenceInputLayer` of the appropriate size.

```
numUnits = 100  ;
layers = [ ...
```

```

sequenceInputLayer(1)
biLstmLayer(numUnits,"OutputMode","last")
fullyConnectedLayer(numel(categories(adsTrain.Labels)))
softmaxLayer
classificationLayer];

options = trainingOptions("adam", ...
    "LearnRateSchedule","piecewise", ...
    "Shuffle","every-epoch", ...
    "Verbose",false, ...
    "MaxEpochs",20);

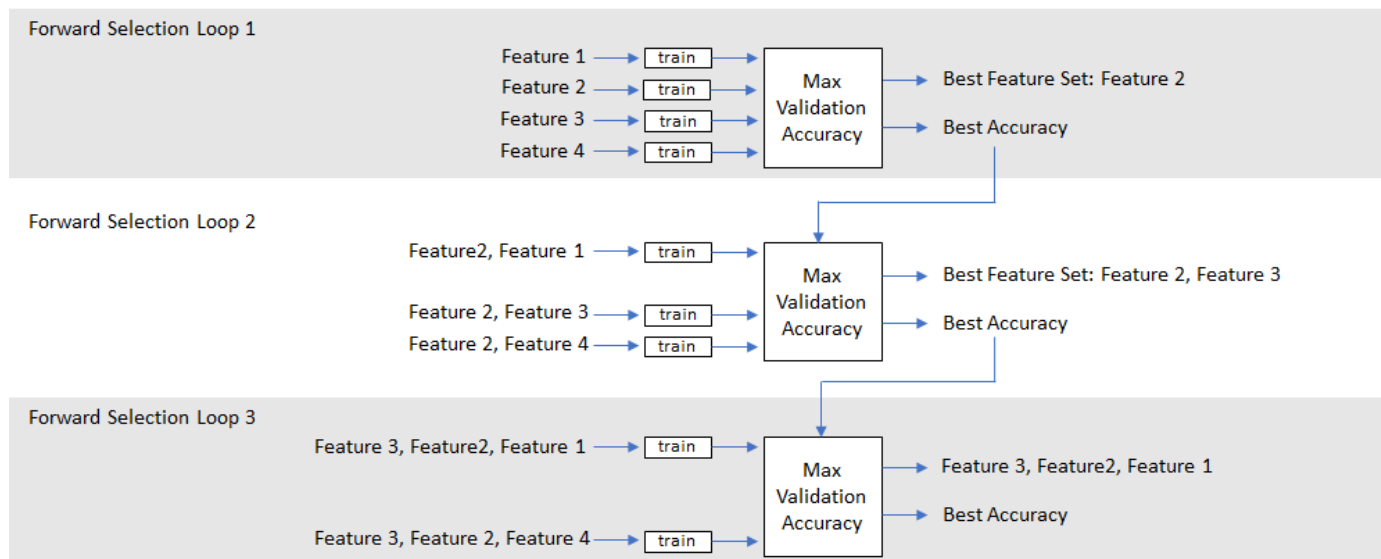
```

Sequential Feature Selection

In the basic form of sequential feature selection, you train a network on a given feature set and then incrementally add or remove features until the accuracy no longer improves [1] on page 1-0 .

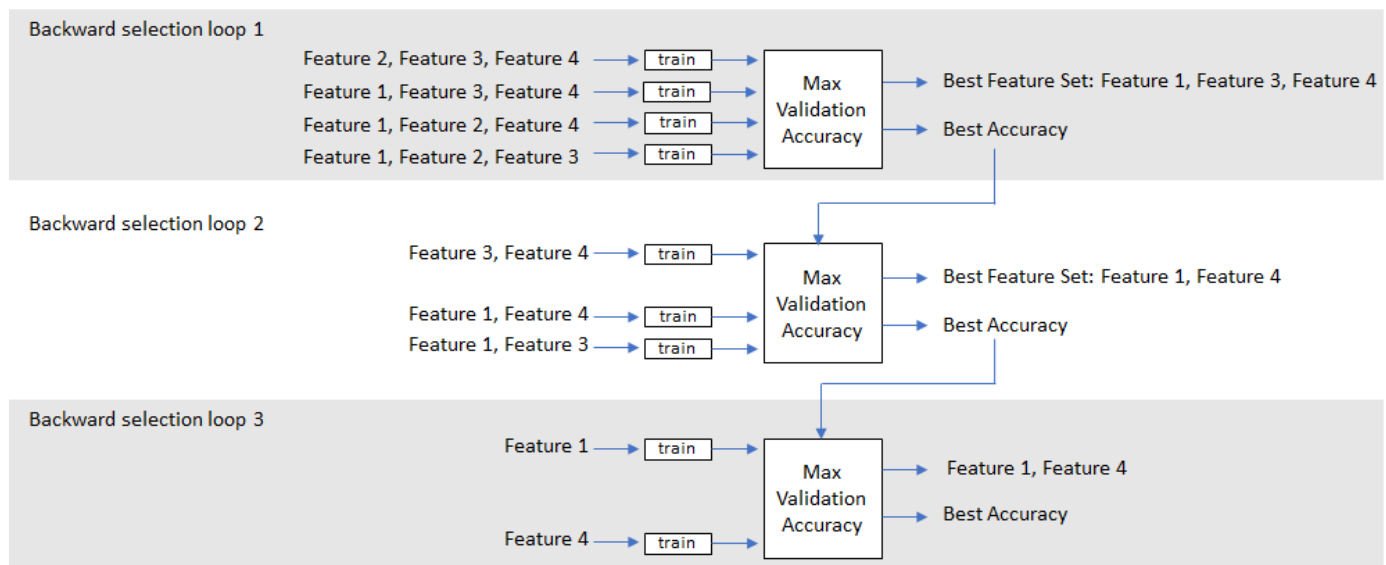
Forward Selection

Consider a simple case of forward selection on a set of four features. In the first forward selection loop, each of the four features are tested independently by training a network and comparing their validation accuracy. The feature that resulted in the highest validation accuracy is noted. In the second forward selection loop, the best feature from the first loop is combined with each of the remaining features. Now each pair of features is used for training. If the accuracy in the second loop did not improve over the accuracy in the first loop, the selection process ends. Otherwise, a new best feature set is selected. The forward selection loop continues until the accuracy no longer improves.



Backward Selection

In backward feature selection, you begin by training on a feature set that consists of all features and test whether or not accuracy improves as you remove features.



Run Sequential Feature Selection

The helper functions (HelperSFS on page 1-0 , HelperTrainAndValidateNetwork on page 1-0 , and HelperTrimOrPad on page 1-0) implement forward or backward sequential feature selection. Specify the training datastore, validation datastore, audio feature extractor, network layers, network options, and direction. As a general rule, choose forward if you anticipate a small feature set or backward if you anticipate a large feature set.

```
direction = forward;
[logbook,bestFeatures,bestNet,normalizers] = HelperSFS(adsTrain,adsValidation,afe,layers,options)
```

Starting parallel pool (parpool) using the 'local' profile ...
 Connected to the parallel pool (number of workers: 6).

The logbook output from HelperFeatureExtractor is a table containing all feature configurations tested and the corresponding validation accuracy.

logbook

```
logbook=48x2 table
```

Features	Accuracy
"mfcc, gtcc"	97.333
"mfcc, mfccDelta, gtcc"	97
"mfcc, gtcc, spectralEntropy"	97
"mfcc, gtcc, spectralFlatness"	97
"mfcc, gtcc, spectralFlux"	97
"mfcc, gtcc, spectralSpread"	97
"gtcc"	96.667
"gtcc, spectralCentroid"	96.667
"gtcc, spectralFlux"	96.667
"mfcc, gtcc, spectralRolloffPoint"	96.667
"mfcc, gtcc, spectralSkewness"	96.667
"gtcc, spectralEntropy"	96.333
"mfcc, gtcc, gtccDeltaDelta"	96.333


```

"mfcc, gtcc, spectralKurtosis"    96.333
"mfccDelta, gtcc"                96
"gtcc, gtccDelta"                96
⋮

```

The `bestFeatures` output from `HelperSFS` contains a struct with the optimal features set to `true`.

bestFeatures

```
bestFeatures = struct with fields:
```

```

    mfcc: 1
    mfccDelta: 0
    mfccDeltaDelta: 0
    gtcc: 1
    gtccDelta: 0
    gtccDeltaDelta: 0
    spectralCentroid: 0
    spectralCrest: 0
    spectralDecrease: 0
    spectralEntropy: 0
    spectralFlatness: 0
    spectralFlux: 0
    spectralKurtosis: 0
    spectralRolloffPoint: 0
    spectralSkewness: 0
    spectralSlope: 0
    spectralSpread: 0

```

You can set your `audioFeatureExtractor` using the struct.

```
set(afe,bestFeatures)
```

```
afe
```

```
afe =
```

```
audioFeatureExtractor with properties:
```

```
Properties
```

```

    Window: [240×1 double]
    OverlapLength: 160
    SampleRate: 8000
    FFTLength: []
    SpectralDescriptorInput: 'linearSpectrum'

```

```
Enabled Features
```

```
mfcc, gtcc
```

```
Disabled Features
```

```

linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfccDelta, mfccDeltaDelta
gtccDelta, gtccDeltaDelta, spectralCentroid, spectralCrest, spectralDecrease, spectralEntropy
spectralFlatness, spectralFlux, spectralKurtosis, spectralRolloffPoint, 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.

HelperSFS also outputs the best performing network and the normalization factors that correspond to the chosen features. To save the network, configured audioFeatureExtractor, and normalization factors, uncomment this line:

```
% save('network_Audio_SequentialFeatureSelection.mat','bestNet','afe','normalizers')
```

Conclusion

This example illustrates a workflow for sequential feature selection for a Recurrent Neural Network (LSTM or BiLSTM). It could easily be adapted for CNN and RNN-CNN workflows.

Supporting Functions

HelperTrainAndValidateNetwork

```
function [trueLabels,predictedLabels,net,normalizers] = HelperTrainAndValidateNetwork(adsTrain,a
% Train and validate a network.
%
% INPUTS:
% adsTrain      - audioDatastore object that points to training set
% adsValidation - audioDatastore object that points to validation set
% afe           - audioFeatureExtractor object.
% layers        - Layers of LSTM or BiLSTM network
% options       - trainingOptions object
%
% OUTPUTS:
% trueLabels    - true labels of validation set
% predictedLabels - predicted labels of validation set
% net           - trained network
% normalizers   - normalization factors for features under test

% Copyright 2019 The MathWorks, Inc.

% Convert the data to tall arrays.
tallTrain      = tall(adsTrain);
tallValidation = tall(adsValidation);

% Extract features from the training set. Reorient the features so that
% time is along rows to be compatible with sequenceInputLayer.
fs = afe.SampleRate;
tallTrain      = cellfun(@(x)HelperTrimOrPad(x,fs/2),tallTrain,"UniformOutput",false);
tallTrain      = cellfun(@(x)x/max(abs(x),[]),'all'),tallTrain,"UniformOutput",false);
tallFeaturesTrain = cellfun(@(x)extract(afe,x),tallTrain,"UniformOutput",false);
tallFeaturesTrain = cellfun(@(x)x',tallFeaturesTrain,"UniformOutput",false); %#ok<NASGU>
[~,featuresTrain] = evalc('gather(tallFeaturesTrain)'); % Use evalc to suppress command-line outp

tallValidation      = cellfun(@(x)HelperTrimOrPad(x,fs/2),tallValidation,"UniformOutput",false);
tallValidation      = cellfun(@(x)x/max(abs(x),[]),'all'),tallValidation,"UniformOutput",false);
tallFeaturesValidation = cellfun(@(x)extract(afe,x),tallValidation,"UniformOutput",false);
tallFeaturesValidation = cellfun(@(x)x',tallFeaturesValidation,"UniformOutput",false); %#ok<NASGU>
[~,featuresValidation] = evalc('gather(tallFeaturesValidation)'); % Use evalc to suppress comman

% Use the training set to determine the mean and standard deviation of each
% feature. Normalize the training and validation sets.
allFeatures = cat(2,featuresTrain{:});
M = mean(allFeatures,2,'omitnan');
S = std(allFeatures,0,2,'omitnan');
featuresTrain = cellfun(@(x)(x-M)./S,featuresTrain,'UniformOutput',false);
```

```

for ii = 1:numel(featuresTrain)
    idx = find(isnan(featuresTrain{ii}));
    if ~isempty(idx)
        featuresTrain{ii}(idx) = 0;
    end
end
featuresValidation = cellfun(@(x)(x-M)./S,featuresValidation,'UniformOutput',false);
for ii = 1:numel(featuresValidation)
    idx = find(isnan(featuresValidation{ii}));
    if ~isempty(idx)
        featuresValidation{ii}(idx) = 0;
    end
end

% Replicate the labels of the train and validation sets so that they are in
% one-to-one correspondence with the sequences.
labelsTrain = adsTrain.Labels;

% Update input layer for the number of features under test.
layers(1) = sequenceInputLayer(size(featuresTrain{1},1));

% Train the network.
net = trainNetwork(featuresTrain,labelsTrain,layers,options);

% Evaluate the network. Call classify to get the predicted labels for each
% sequence.
predictedLabels = classify(net,featuresValidation);
trueLabels = adsValidation.Labels;

% Save the normalization factors as a struct.
normalizers.Mean = M;
normalizers.StandardDeviation = S;
end

```

HelperSFS

```

function [logbook,bestFeatures,bestNet,bestNormalizers] = HelperSFS(adsTrain,adsValidate,afeThis
%
% INPUTS:
% adsTrain - audioDatastore object that points to training set
% adsValidate - audioDatastore object that points to validation set
% afe - audioFeatureExtractor object. Set all features to test to true
% layers - Layers of LSTM or BiLSTM network
% options - traingOptions object
% direction - SFS direction, specify as 'forward' or 'backward'
%
% OUTPUTS:
% logbook - table containing feature configurations tested and corresponding validation
% bestFeatures - struct containing best feature configuration
% bestNet - Trained network with highest validation accuracy
% bestNormalizers - Feature normalization factors for best features

% Copyright 2019 The MathWorks, Inc.

afe = copy(afeThis);
featuresToTest = fieldnames(info(afe));
N = numel(featuresToTest);
bestValidationAccuracy = 0;

```

```
% Set the initial feature configuration: all on for backward selection
% or all off for forward selection.
featureConfig = info(afe);
for i = 1:N
    if strcmpi(direction,"backward")
        featureConfig.(featuresToTest{i}) = true;
    else
        featureConfig.(featuresToTest{i}) = false;
    end
end

% Initialize logbook to track feature configuration and accuracy.
logbook = table(featureConfig,0,'VariableNames',["Feature Configuration","Accuracy"]);

% Perform sequential feature evaluation.
wrapperIdx = 1;
bestAccuracy = 0;
while wrapperIdx <= N
    % Create a cell array containing all feature configurations to test
    % in the current loop.
    featureConfigsToTest = cell(numel(featuresToTest),1);
    for ii = 1:numel(featuresToTest)
        if strcmpi(direction,"backward")
            featureConfig.(featuresToTest{ii}) = false;
        else
            featureConfig.(featuresToTest{ii}) = true;
        end
        featureConfigsToTest{ii} = featureConfig;
        if strcmpi(direction,"backward")
            featureConfig.(featuresToTest{ii}) = true;
        else
            featureConfig.(featuresToTest{ii}) = false;
        end
    end

    % Loop over every feature set.
    for ii = 1:numel(featureConfigsToTest)

        % Determine the current feature configuration to test. Update
        % the feature afe.
        currentConfig = featureConfigsToTest{ii};
        set(afe,currentConfig)

        % Train and get k-fold cross-validation accuracy for current
        % feature configuration.
        [trueLabels,predictedLabels,net,normalizers] = HelperTrainAndValidateNetwork(adsTrain,adsTest,normalizers);
        valAccuracy = mean(trueLabels==predictedLabels)*100;
        if valAccuracy > bestValidationAccuracy
            bestValidationAccuracy = valAccuracy;
            bestNet = net;
            bestNormalizers = normalizers;
        end

        % Update Logbook
        result = table(currentConfig,valAccuracy,'VariableNames',["Feature Configuration","Accuracy"]);
        logbook = [logbook;result]; %#ok<AGROW>
```

```

end

% Determine and print the setting with the best accuracy. If accuracy
% did not improve, end the run.
[a,b] = max(logbook{:,'Accuracy'});
if a <= bestAccuracy
    wrapperIdx = inf;
else
    wrapperIdx = wrapperIdx + 1;
end
bestAccuracy = a;

% Update the features-to-test based on the most recent winner.
winner = logbook{b,'Feature Configuration'};
fn = fieldnames(winner);
tf = structfun(@(x)(x),winner);
if strcmpi(direction,"backward")
    featuresToRemove = fn(~tf);
else
    featuresToRemove = fn(tf);
end
for ii = 1:numel(featuresToRemove)
    loc = strcmp(featuresToTest,featuresToRemove{ii});
    featuresToTest(loc) = [];
    if strcmpi(direction,"backward")
        featureConfig.(featuresToRemove{ii}) = false;
    else
        featureConfig.(featuresToRemove{ii}) = true;
    end
end
end

end

% Sort the logbook and make it more readable.
logbook(1,:) = []; % Delete placeholder first row.
logbook = sortrows(logbook',{'Accuracy'},{'descend'});
bestFeatures = logbook{1,'Feature Configuration'};
m = logbook{:,'Feature Configuration'};
fn = fieldnames(m);
myString = strings(numel(m),1);
for wrapperIdx = 1:numel(m)
    tf = structfun(@(x)(x),logbook{wrapperIdx,'Feature Configuration'});
    myString(wrapperIdx) = strjoin(fn(tf)," ");
end
logbook = table(myString,logbook{:,'Accuracy'},'VariableNames',['Features',"Accuracy"]);
end

```

HelperTrimOrPad

```

function y = HelperTrimOrPad(x,n)
% y = HelperTrimOrPad(x,n) trims or pads the input x to n samples. If x is
% trimmed, it is trimmed equally on the front and back. If x is padded, it is
% padded equally on the front and back with zeros. For odd-length trimming or
% padding, the extra sample is trimmed or padded from the back.

% Copyright 2019 The MathWorks, Inc.
a = size(x,1);
if a < n

```

```
    frontPad = floor((n-a)/2);
    backPad = n - a - frontPad;
    y = [zeros(frontPad,1);x;zeros(backPad,1)];
elseif a > n
    frontTrim = floor((a-n)/2)+1;
    backTrim = a - n - frontTrim;
    y = x(frontTrim:end-backTrim);
else
    y = x;
end
end
```

References

[1] Jain, A., and D. Zongker. "Feature Selection: Evaluation, Application, and Small Sample Performance." IEEE Transactions on Pattern Analysis and Machine Intelligence. Vol. 19, Issue 2, 1997, pp. 153-158.

[2] Jakobovski. "Jakobovski/Free-Spoken-Digit-Dataset." GitHub, May 30, 2019. <https://github.com/Jakobovski/free-spoken-digit-dataset>.

Train Generative Adversarial Network (GAN) for Sound Synthesis

This example shows how to train and use a generative adversarial network (GAN) to generate sounds.

Introduction

In generative adversarial networks, a generator and a discriminator compete against each other to improve the generation quality.

GANs have generated significant interest in the field of audio and speech processing. Applications include text-to-speech synthesis, voice conversion, and speech enhancement.

This example trains a GAN for unsupervised synthesis of audio waveforms. The GAN in this example generates drumbeat sounds. The same approach can be followed to generate other types of sound, including speech.

Synthesize Audio with Pre-Trained GAN

Before you train a GAN from scratch, you will use a pretrained GAN generator to synthesize drum beats.

Download the pretrained generator.

```
matFileName = 'drumGeneratorWeights.mat';
if ~exist(matFileName,'file')
    websave(matFileName,'https://www.mathworks.com/supportfiles/audio/GanAudioSynthesis/drumGene
end
```

The function `synthesizeDrumBeat` calls a pretrained network to synthesize a drumbeat sampled at 16 kHz. The `synthesizeDrumBeat` function is included at the end of this example.

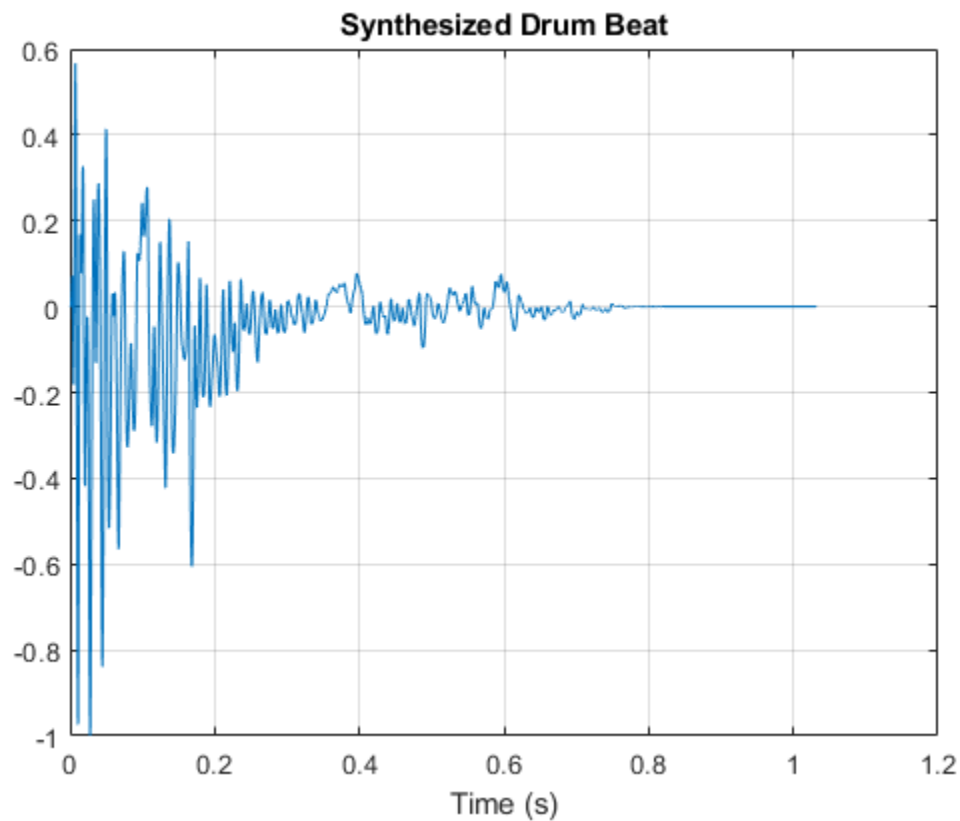
Synthesize a drumbeat and listen to it.

```
drum = synthesizeDrumBeat;
```

```
fs = 16e3;
sound(drum,fs)
```

Plot the synthesized drumbeat.

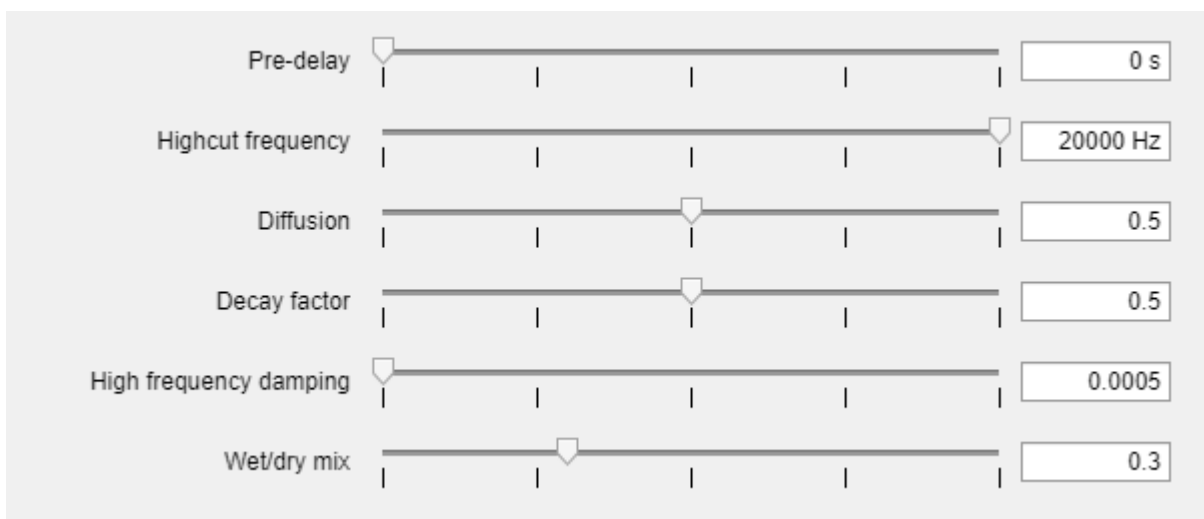
```
t = (0:length(drum)-1)/fs;
plot(t,drum)
grid on
xlabel('Time (s)')
title('Synthesized Drum Beat')
```



You can use the drumbeat synthesizer with other audio effects to create more complex applications. For example, you can apply reverberation to the synthesized drum beats.

Create a `reverberator` object and open its parameter tuner UI. This UI enables you to tune the reverberator parameters as the simulation runs.

```
reverb = reverberator('SampleRate',fs);  
parameterTuner(reverb);
```

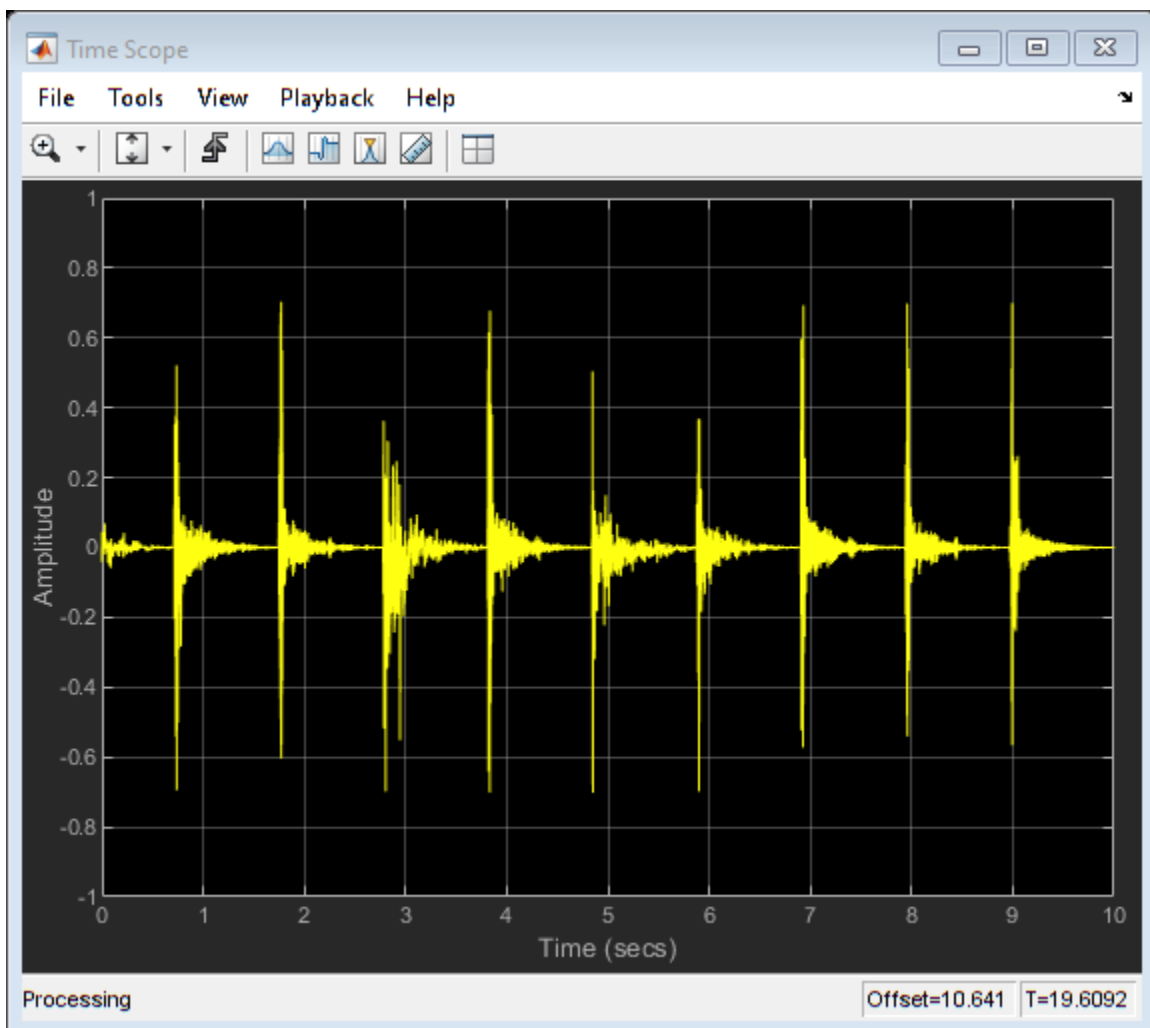


Create a `dsp.TimeScope` object to visualize the drum beats.

```
ts = dsp.TimeScope('SampleRate',fs, ...
    'TimeSpanOvverrunAction','Scroll', ...
    'TimeSpan',10, ...
    'BufferLength',10*256*64, ...
    'ShowGrid',true, ...
    'YLimits',[-1 1]);
```

In a loop, synthesize the drum beats and apply reverberation. Use the parameter tuner UI to tune reverberation. If you want to run the simulation for a longer time, increase the value of the `loopCount` parameter.

```
loopCount = 20;
for ii = 1:loopCount
    drum = synthesizeDrumBeat;
    drum = reverb(drum);
    ts(drum(:,1));
    soundsc(drum,fs)
    pause(0.5)
end
```



Train the GAN

Now that you have seen the pretrained drumbeat generator in action, you can investigate the training process in detail.

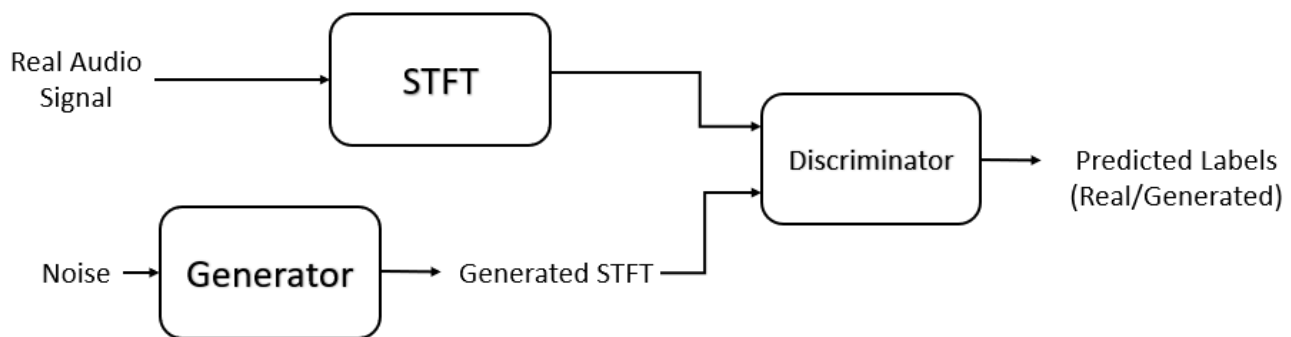
A GAN is a type of deep learning network that generates data with characteristics similar to the training data.

A GAN consists of two networks that train together, a *generator* and a *discriminator*:

- Generator - Given a vector or random values as input, this network generates data with the same structure as the training data. It is the generator's job to fool the discriminator.
- Discriminator - Given batches of data containing observations from both the training data and the generated data, this network attempts to classify the observations as real or generated.

To maximize the performance of the generator, maximize the loss of the discriminator when given generated data. That is, the objective of the generator is to generate data that the discriminator classifies as real. To maximize the performance of the discriminator, minimize the loss of the discriminator when given batches of both real and generated data. Ideally, these strategies result in a generator that generates convincingly realistic data and a discriminator that has learned strong feature representations that are characteristic of the training data.

In this example, you train the generator to create fake time-frequency short-time Fourier transform (STFT) representations of drum beats. You train the discriminator to identify real STFTs. You create the real STFTs by computing the STFT of short recordings of real drum beats.



Load Training Data

Train a GAN using the Drum Sound Effects dataset [1]. Download and extract the dataset.

```
url = 'http://deepyeti.ucsd.edu/cdonahue/wavegan/data/drums.tar.gz';  
downloadFolder = tempdir;  
filename = fullfile(downloadFolder, 'drums_dataset.tgz');  
  
drumsFolder = fullfile(downloadFolder, 'drums');  
if ~exist(drumsFolder, 'dir')  
    disp('Downloading Drum Sound Effects Dataset (218 MB)...')  
    websave(filename, url);  
    untar(filename, downloadFolder)  
end
```

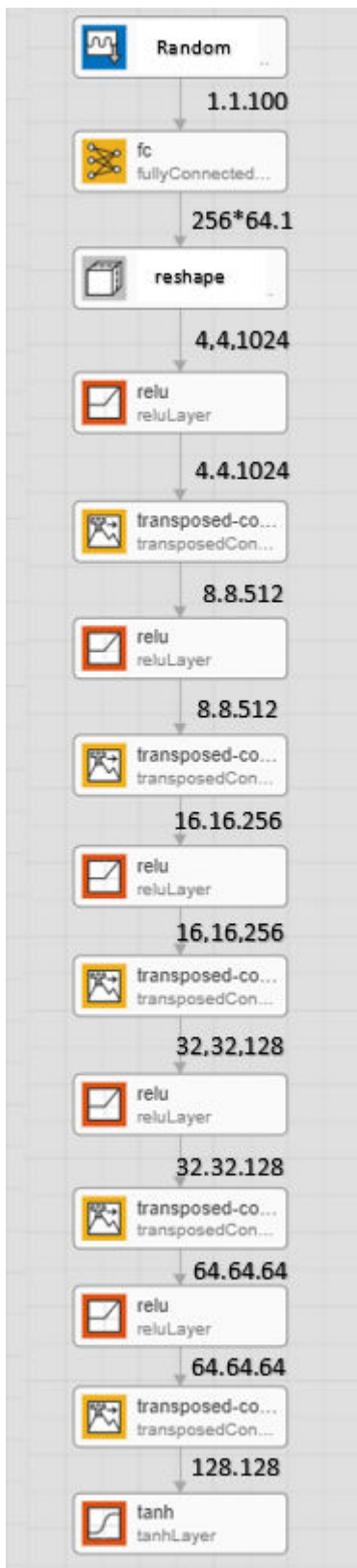
Create an `audioDatastore` object that points to the drums dataset.

```
ads = audioDatastore(drumsFolder, 'IncludeSubfolders', true);
```

Define Generator Network

Define a network that generates STFTs from 1-by-1-by-100 arrays of random values. Create a network that upscales 1-by-1-by-100 arrays to 128-by-128-by-1 arrays using a fully connected layer followed by a reshape layer and a series of transposed convolution layers with ReLU layers.

This figure shows the dimensions of the signal as it travels through the generator. The generator architecture is defined in Table 4 of [1].



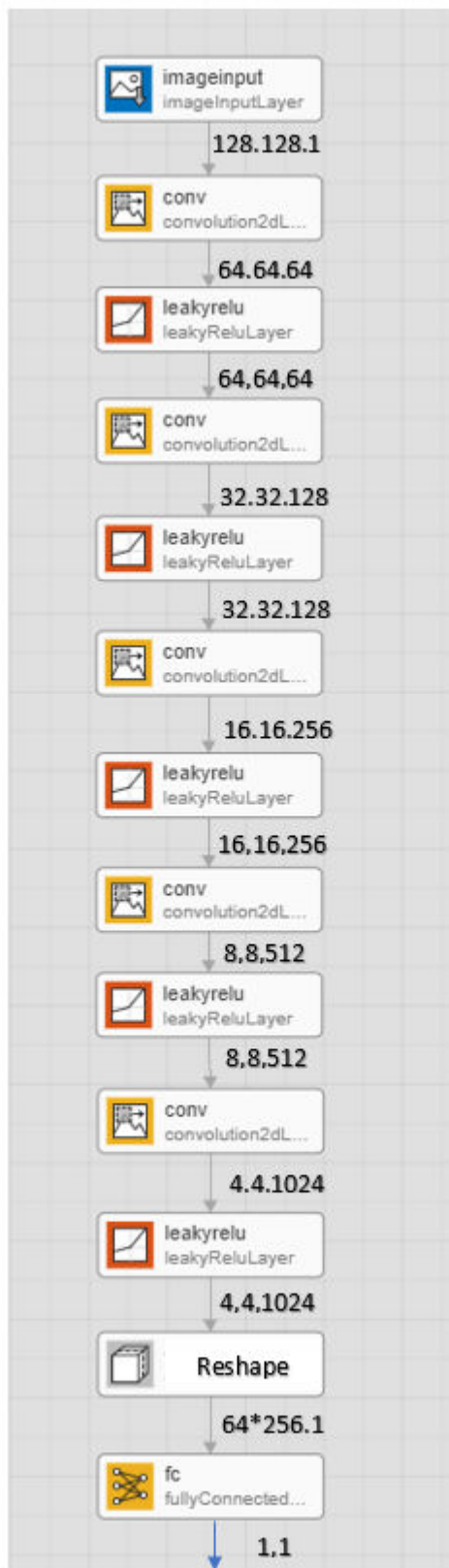
The generator network is defined in `modelGenerator`, which is included at the end of this example.

Define Discriminator Network

Define a network that classifies real and generated 128-by-128 STFTs.

Create a network that takes 128-by-128 images and outputs a scalar prediction score using a series of convolution layers with leaky ReLU layers followed by a fully connected layer.

This figure shows the dimensions of the signal as it travels through the discriminator. The discriminator architecture is defined in Table 5 of [1].



The discriminator network is defined in `modelDiscriminator`, which is included at the end of this example.

Generate Real Drumbeat Training Data

Generate STFT data from the drumbeat signals in the datastore.

Define the STFT parameters.

```
fftLength = 256;
win = hann(fftLength, 'periodic');
overlapLength = 128;
```

To speed up processing, distribute the feature extraction across multiple workers using `parfor`.

First, determine the number of partitions for the dataset. If you do not have Parallel Computing Toolbox™, use a single partition.

```
if ~isempty(ver('parallel'))
    pool = gcp;
    numPar = numpartitions(ads,pool);
else
    numPar = 1;
end
```

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).

For each partition, read from the datastore and compute the STFT.

```
parfor ii = 1:numPar

    subds = partition(ads,numPar,ii);
    STrain = zeros(fftLength/2+1,128,1,numel(subds.Files));

    for idx = 1:numel(subds.Files)

        x = read(subds);

        if length(x) > fftLength*64
            % Lengthen the signal if it is too short
            x = x(1:fftLength*64);
        end

        % Convert from double-precision to single-precision
        x = single(x);

        % Scale the signal
        x = x ./ max(abs(x));

        % Zero-pad to ensure stft returns 128 windows.
        x = [x ; zeros(overlapLength,1,'like',x)];

        S0 = stft(x,'Window',win,'OverlapLength',overlapLength,'Centered',false);

        % Convert from two-sided to one-sided.
        S = S0(1:129,:);
        S = abs(S);
```

```
        STrain(:,:,:,idx) = S;  
    end  
    STrainC{ii} = STrain;  
end
```

Convert the output to a four-dimensional array with STFTs along the fourth dimension.

```
STrain = cat(4,STrainC{:});
```

Convert the data to the log scale to better align with human perception.

```
STrain = log(STrain + 1e-6);
```

Normalize training data to have zero mean and unit standard deviation.

Compute the STFT mean and standard deviation of each frequency bin.

```
SMean = mean(STrain,[2 3 4]);  
SStd = std(STrain,1,[2 3 4]);
```

Normalize each frequency bin.

```
STrain = (STrain-SMean)./SStd;
```

The computed STFTs have unbounded values. Following the approach in [1], make the data bounded by clipping the spectra to 3 standard deviations and rescaling to [-1 1].

```
STrain = STrain/3;  
Y = reshape(STrain,numel(STrain),1);  
Y(Y<-1) = -1;  
Y(Y>1) = 1;  
STrain = reshape(Y,size(STrain));
```

Discard the last frequency bin to force the number of STFT bins to a power of two (which works well with convolutional layers).

```
STrain = STrain(1:end-1,:,:,:);
```

Permute the dimensions in preparation for feeding to the discriminator.

```
STrain = permute(STrain,[2 1 3 4]);
```

Specify Training Options

Train with a mini-batch size of 64 for 1000 epochs.

```
maxEpochs = 1000;  
miniBatchSize = 64;
```

Compute the number of iterations required to consume the data.

```
numIterationsPerEpoch = floor(size(STrain,4)/miniBatchSize);
```

Specify the options for Adam optimization. Set the learn rate of the generator and discriminator to 0.0002. For both networks, use a gradient decay factor of 0.5 and a squared gradient decay factor of 0.999.

```
learnRateGenerator = 0.0002;  
learnRateDiscriminator = 0.0002;
```



```
gradientDecayFactor = 0.5;
squaredGradientDecayFactor = 0.999;
```

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA-enabled NVIDIA GPU with compute capability of 3.0 or higher.

```
executionEnvironment = "auto";
```

Initialize the generator and discriminator weights. The `initializeGeneratorWeights` and `initializeDiscriminatorWeights` functions return random weights obtained using Glorot uniform initialization. The functions are included at the end of this example.

```
generatorParameters = initializeGeneratorWeights;
discriminatorParameters = initializeDiscriminatorWeights;
```

Train Model

Train the model using a custom training loop. Loop over the training data and update the network parameters at each iteration.

For each epoch, shuffle the training data and loop over mini-batches of data.

For each mini-batch:

- Generate a `dlarray` object containing an array of random values for the generator network.
- For GPU training, convert the data to a `gpuArray` (Parallel Computing Toolbox) object.
- Evaluate the model gradients using `dlfeval` (Deep Learning Toolbox) and the helper functions, `modelDiscriminatorGradients` and `modelGeneratorGradients`.
- Update the network parameters using the `adamupdate` (Deep Learning Toolbox) function.

Initialize the parameters for Adam.

```
trailingAvgGenerator = [];
trailingAvgSqGenerator = [];
trailingAvgDiscriminator = [];
trailingAvgSqDiscriminator = [];
```

You can set `saveCheckpoints` to `true` to save the updated weights and states to a MAT file every ten epochs. You can then use this MAT file to resume training if it is interrupted. For the purpose of this example, set `saveCheckpoints` to `false`.

```
saveCheckpoints = false;
```

Specify the length of the generator input.

```
numLatentInputs = 100;
```

Train the GAN. This can take multiple hours to run.

```
iteration = 0;
```

```
for epoch = 1:maxEpochs
```

```
    % Shuffle the data.
    idx = randperm(size(STrain,4));
    STrain = STrain(:, :, :, idx);
```

```
% Loop over mini-batches.
for index = 1:numIterationsPerEpoch

    iteration = iteration + 1;

    % Read mini-batch of data.
    dLX = STrain(:, :, :, (index-1)*miniBatchSize+1:index*miniBatchSize);
    dLX = dlarray(dLX, 'SSCB');

    % Generate latent inputs for the generator network.
    Z = 2 * ( rand(1,1,numLatentInputs,miniBatchSize,'single') - 0.5 ) ;
    dLZ = dlarray(Z);

    % If training on a GPU, then convert data to gpuArray.
    if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
        dLZ = gpuArray(dLZ);
        dLX = gpuArray(dLX);
    end

    % Evaluate the discriminator gradients using dlfeval and the
    % |modelDiscriminatorGradients| helper function.
    gradientsDiscriminator = ...
        dlfeval(@modelDiscriminatorGradients,discriminatorParameters,generatorParameters,dLX);

    % Update the discriminator network parameters.
    [discriminatorParameters,trailingAvgDiscriminator,trailingAvgSqDiscriminator] = ...
        adamupdate(discriminatorParameters,gradientsDiscriminator, ...
            trailingAvgDiscriminator,trailingAvgSqDiscriminator,iteration, ...
            learnRateDiscriminator,gradientDecayFactor,squaredGradientDecayFactor);

    % Generate latent inputs for the generator network.
    Z = 2 * ( rand(1,1,numLatentInputs,miniBatchSize,'single') - 0.5 ) ;
    dLZ = dlarray(Z);

    % If training on a GPU, then convert data to gpuArray.
    if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
        dLZ = gpuArray(dLZ);
    end

    % Evaluate the generator gradients using dlfeval and the
    % |modelGeneratorGradients| helper function.
    gradientsGenerator = ...
        dlfeval(@modelGeneratorGradients,discriminatorParameters,generatorParameters,dLZ);

    % Update the generator network parameters.
    [generatorParameters,trailingAvgGenerator,trailingAvgSqGenerator] = ...
        adamupdate(generatorParameters,gradientsGenerator, ...
            trailingAvgGenerator,trailingAvgSqGenerator,iteration, ...
            learnRateGenerator,gradientDecayFactor,squaredGradientDecayFactor);
end

% Every 10 iterations, save a training snapshot to a MAT file.
if saveCheckpoints && mod(epoch,10)==0
    fprintf('Epoch %d out of %d complete\n',epoch,maxEpochs);
    % Save checkpoint in case training is interrupted.
    save('audiogancheckpoint.mat',...
        'generatorParameters','discriminatorParameters',...
        'trailingAvgDiscriminator','trailingAvgSqDiscriminator',...

```

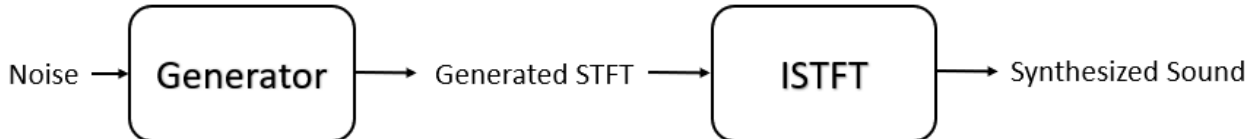
```

        'trailingAvgGenerator','trailingAvgSqGenerator','iteration');
    end
end

```

Synthesize Sounds

Now that you have trained the network, you can investigate the synthesis process in more detail.



The trained drumbeat generator synthesizes short-time Fourier transform (STFT) matrices from input arrays of random values. An inverse STFT (ISTFT) operation converts the time-frequency STFT to a synthesized time-domain audio signal.

Load the weights of a pretrained generator. These weights were obtained by running the training highlighted in the previous section for 1000 epochs.

```
load(matFileName, 'generatorParameters', 'SMean', 'SStd');
```

The generator takes 1-by-1-by-100 vectors of random values as an input. Generate a sample input vector.

```
numLatentInputs = 100;
dLZ = dldarray(2 * ( rand(1,1,numLatentInputs,1,'single') - 0.5 ));
```

Pass the random vector to the generator to create an STFT image. `generatorParameters` is a structure containing the weights of the pretrained generator.

```
dLXGenerated = modelGenerator(dLZ,generatorParameters);
```

Convert the STFT `dldarray` to a single-precision matrix.

```
S = dLXGenerated.extractdata;
```

Transpose the STFT to align its dimensions with the `istft` function.

```
S = S.';
```

The STFT is a 128-by-128 matrix, where the first dimension represents 128 frequency bins linearly spaced from 0 to 8 kHz. The generator was trained to generate a one-sided STFT from an FFT length of 256, with the last bin omitted. Reintroduce that bin by inserting a row of zeros into the STFT.

```
S = [S ; zeros(1,128)];
```

Revert the normalization and scaling steps used when you generated the STFTs for training.

```
S = S * 3;
S = (S.*SStd) + SMean;
```

Convert the STFT from the log domain to the linear domain.

```
S = exp(S);
```

Convert the STFT from one-sided to two-sided.

```
S = [S; S(end-1:-1:2,:)];
```

Pad with zeros to remove window edge-effects.

```
S = [zeros(256,100) S zeros(256,100)];
```

The STFT matrix does not contain any phase information. Use a fast version of the Griffin-Lim algorithm with 20 iterations to estimate the signal phase and produce audio samples.

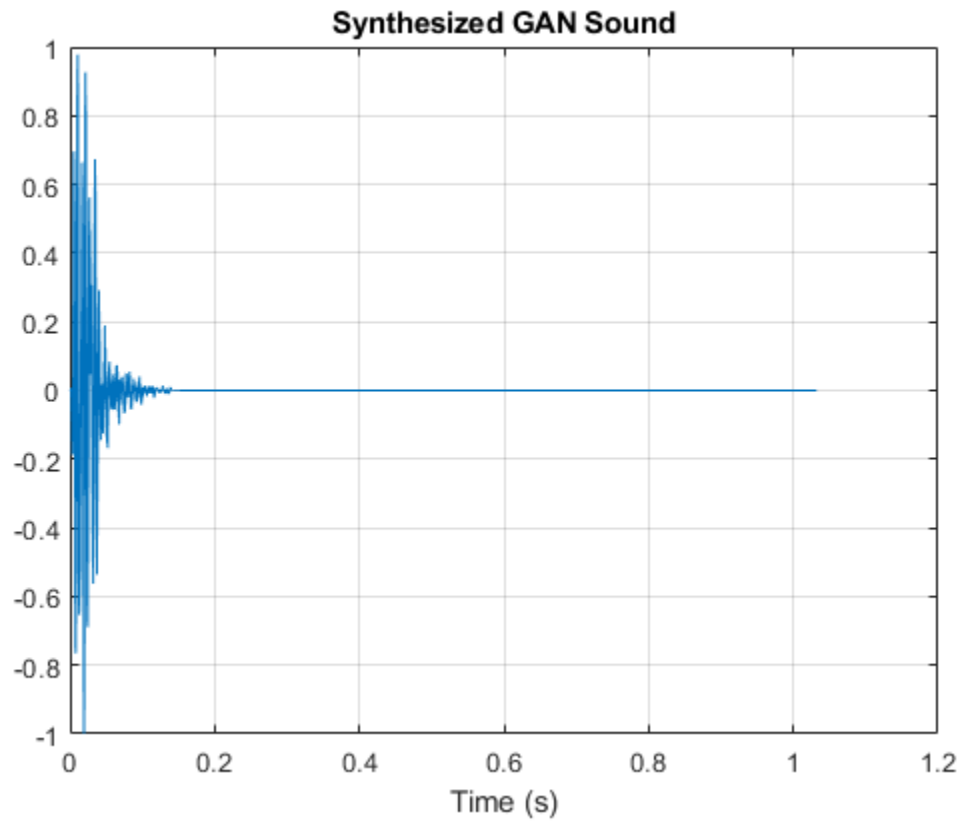
```
myAudio = stftmag2sig(S,256, ...  
    'FrequencyRange','twosided', ...  
    'Window',hann(256,'periodic'), ...  
    'OverlapLength',128, ...  
    'MaxIterations',20, ...  
    'Method','fgla');  
myAudio = myAudio./max(abs(myAudio),[],'all');  
myAudio = myAudio(128*100:end-128*100);
```

Listen to the synthesized drumbeat.

```
sound(myAudio,fs)
```

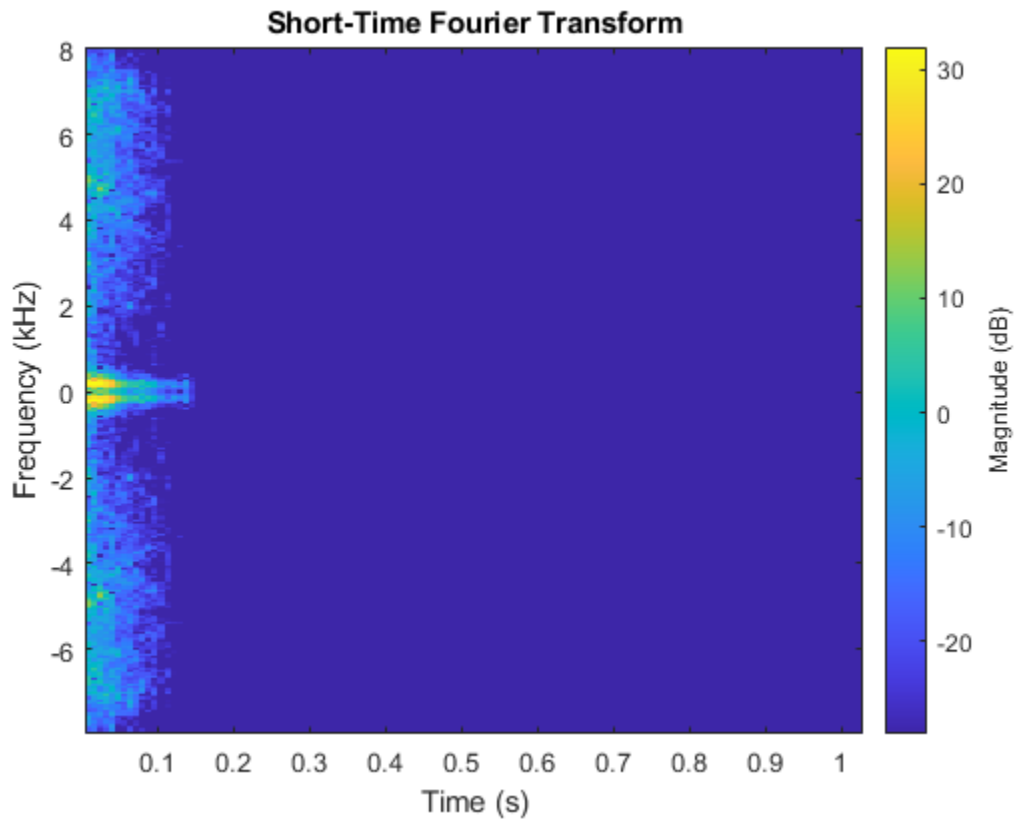
Plot the synthesized drumbeat.

```
t = (0:length(myAudio)-1)/fs;  
plot(t,myAudio)  
grid on  
xlabel('Time (s)')  
title('Synthesized GAN Sound')
```



Plot the STFT of the synthesized drumbeat.

```
figure  
stft(myAudio,fs,'Window',hann(256,'periodic'),'OverlapLength',128);
```



Model Generator Function

The `modelGenerator` function upscales 1-by-1-by-100 arrays (`dlX`) to 128-by-128-by-1 arrays (`dlY`). `parameters` is a structure holding the weights of the generator layers. The generator architecture is defined in Table 4 of [1].

```
function dlY = modelGenerator(dlX,parameters)

dlY = fullyconnect(dlX,parameters.FC.Weights,parameters.FC.Bias,'Dataformat','SSCB');

dlY = reshape(dlY,[1024 4 4 size(dlY,2)]);
dlY = permute(dlY,[3 2 1 4]);
dlY = relu(dlY);

dlY = dltranspconv(dlY,parameters.Conv1.Weights,parameters.Conv1.Bias,'Stride',2,'Cropping','same');
dlY = relu(dlY);

dlY = dltranspconv(dlY,parameters.Conv2.Weights,parameters.Conv2.Bias,'Stride',2,'Cropping','same');
dlY = relu(dlY);

dlY = dltranspconv(dlY,parameters.Conv3.Weights,parameters.Conv3.Bias,'Stride',2,'Cropping','same');
dlY = relu(dlY);

dlY = dltranspconv(dlY,parameters.Conv4.Weights,parameters.Conv4.Bias,'Stride',2,'Cropping','same');
dlY = relu(dlY);

dlY = dltranspconv(dlY,parameters.Conv5.Weights,parameters.Conv5.Bias,'Stride',2,'Cropping','same');
```

```
dLY = tanh(dLY);
end
```

Model Discriminator Function

The `modelDiscriminator` function takes 128-by-128 images and outputs a scalar prediction score. The discriminator architecture is defined in Table 5 of [1].

```
function dLY = modelDiscriminator(dLX,parameters)

dLY = dlconv(dLX,parameters.Conv1.Weights,parameters.Conv1.Bias,'Stride',2,'Padding','same');
dLY = leakyrelu(dLY,0.2);

dLY = dlconv(dLY,parameters.Conv2.Weights,parameters.Conv2.Bias,'Stride',2,'Padding','same');
dLY = leakyrelu(dLY,0.2);

dLY = dlconv(dLY,parameters.Conv3.Weights,parameters.Conv3.Bias,'Stride',2,'Padding','same');
dLY = leakyrelu(dLY,0.2);

dLY = dlconv(dLY,parameters.Conv4.Weights,parameters.Conv4.Bias,'Stride',2,'Padding','same');
dLY = leakyrelu(dLY,0.2);

dLY = dlconv(dLY,parameters.Conv5.Weights,parameters.Conv5.Bias,'Stride',2,'Padding','same');
dLY = leakyrelu(dLY,0.2);

dLY = stripdims(dLY);
dLY = permute(dLY,[3 2 1 4]);
dLY = reshape(dLY,4*4*64*16,numel(dLY)/(4*4*64*16));

weights = parameters.FC.Weights;
bias = parameters.FC.Bias;
dLY = fullyconnect(dLY,weights,bias,'Dataformat','CB');

end
```

Model Discriminator Gradients Function

The `modelDiscriminatorGradients` functions takes as input the generator and discriminator parameters `generatorParameters` and `discriminatorParameters`, a mini-batch of input data `dLX`, and an array of random values `dLZ`, and returns the gradients of the discriminator loss with respect to the learnable parameters in the networks.

```
function gradientsDiscriminator = modelDiscriminatorGradients(discriminatorParameters , generatorParameters, dLX, dLZ)

% Calculate the predictions for real data with the discriminator network.
dLYPred = modelDiscriminator(dLX,discriminatorParameters);

% Calculate the predictions for generated data with the discriminator network.
dLXGenerated = modelGenerator(dLZ,generatorParameters);
dLYPredGenerated = modelDiscriminator(dLarray(dLXGenerated,'SSCB'),discriminatorParameters);

% Calculate the GAN loss
lossDiscriminator = ganDiscriminatorLoss(dLYPred,dLYPredGenerated);

% For each network, calculate the gradients with respect to the loss.
gradientsDiscriminator = dlgradient(lossDiscriminator,discriminatorParameters);

end
```

Model Generator Gradients Function

The `modelGeneratorGradients` function takes as input the discriminator and generator learnable parameters and an array of random values `dLZ`, and returns the gradients of the generator loss with respect to the learnable parameters in the networks.

```
function gradientsGenerator = modelGeneratorGradients(discriminatorParameters, generatorParameters, dLZ)

% Calculate the predictions for generated data with the discriminator network.
dLXGenerated = modelGenerator(dLZ, generatorParameters);
dLYPredGenerated = modelDiscriminator(dLarray(dLXGenerated, 'SSCB'), discriminatorParameters);

% Calculate the GAN loss
lossGenerator = ganGeneratorLoss(dLYPredGenerated);

% For each network, calculate the gradients with respect to the loss.
gradientsGenerator = dlgradient(lossGenerator, generatorParameters);

end
```

Discriminator Loss Function

The objective of the discriminator is to not be fooled by the generator. To maximize the probability that the discriminator successfully discriminates between the real and generated images, minimize the discriminator loss function. The loss function for the generator follows the DCGAN approach highlighted in [1].

```
function lossDiscriminator = ganDiscriminatorLoss(dLYPred, dLYPredGenerated)

fake = dLarray(zeros(1, size(dLYPred, 2)));
real = dLarray(ones(1, size(dLYPred, 2)));

D_loss = mean(sigmoid_cross_entropy_with_logits(dLYPredGenerated, fake));
D_loss = D_loss + mean(sigmoid_cross_entropy_with_logits(dLYPred, real));
lossDiscriminator = D_loss / 2;

end
```

Generator Loss Function

The objective of the generator is to generate data that the discriminator classifies as "real". To maximize the probability that images from the generator are classified as real by the discriminator, minimize the generator loss function. The loss function for the generator follows the deep convolutional generative adversarial network (DCGAN) approach highlighted in [1].

```
function lossGenerator = ganGeneratorLoss(dLYPredGenerated)
real = dLarray(ones(1, size(dLYPredGenerated, 2)));
lossGenerator = mean(sigmoid_cross_entropy_with_logits(dLYPredGenerated, real));

end
```

Discriminator Weights Initializer

`initializeDiscriminatorWeights` initializes discriminator weights using the Glorot algorithm.

```
function discriminatorParameters = initializeDiscriminatorWeights

filterSize = [5 5];
dim = 64;
```



```

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 1 dim]);
bias = zeros(1,1,dim,'single');
discriminatorParameters.Conv1.Weights = darray(weights);
discriminatorParameters.Conv1.Bias = darray(bias);

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) dim 2*dim]);
bias = zeros(1,1,2*dim,'single');
discriminatorParameters.Conv2.Weights = darray(weights);
discriminatorParameters.Conv2.Bias = darray(bias);

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 2*dim 4*dim]);
bias = zeros(1,1,4*dim,'single');
discriminatorParameters.Conv3.Weights = darray(weights);
discriminatorParameters.Conv3.Bias = darray(bias);

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 4*dim 8*dim]);
bias = zeros(1,1,8*dim,'single');
discriminatorParameters.Conv4.Weights = darray(weights);
discriminatorParameters.Conv4.Bias = darray(bias);

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 8*dim 16*dim]);
bias = zeros(1,1,16*dim,'single');
discriminatorParameters.Conv5.Weights = darray(weights);
discriminatorParameters.Conv5.Bias = darray(bias);

% fully connected
weights = iGlorotInitialize([1,4 * 4 * dim * 16]);
bias = zeros(1,1,'single');
discriminatorParameters.FC.Weights = darray(weights);
discriminatorParameters.FC.Bias = darray(bias);
end

```

Generator Weights Initializer

`initializeGeneratorWeights` initializes generator weights using the Glorot algorithm.

```

function generatorParameters = initializeGeneratorWeights

dim = 64;

% Dense 1
weights = iGlorotInitialize([dim*256,100]);
bias = zeros(dim*256,1,'single');
generatorParameters.FC.Weights = darray(weights);
generatorParameters.FC.Bias = darray(bias);

filterSize = [5 5];

% Trans Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 8*dim 16*dim]);
bias = zeros(1,1,dim*8,'single');
generatorParameters.Conv1.Weights = darray(weights);
generatorParameters.Conv1.Bias = darray(bias);

```

```
% Trans Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 4*dim 8*dim]);
bias = zeros(1,1,dim*4,'single');
generatorParameters.Conv2.Weights = dlarray(weights);
generatorParameters.Conv2.Bias = dlarray(bias);

% Trans Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 2*dim 4*dim]);
bias = zeros(1,1,dim*2,'single');
generatorParameters.Conv3.Weights = dlarray(weights);
generatorParameters.Conv3.Bias = dlarray(bias);

% Trans Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) dim 2*dim]);
bias = zeros(1,1,dim,'single');
generatorParameters.Conv4.Weights = dlarray(weights);
generatorParameters.Conv4.Bias = dlarray(bias);

% Trans Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 1 dim]);
bias = zeros(1,1,1,'single');
generatorParameters.Conv5.Weights = dlarray(weights);
generatorParameters.Conv5.Bias = dlarray(bias);
end
```

Synthesize Drumbeat

synthesizeDrumBeat uses a pretrained network to synthesize drum beats.

```
function y = synthesizeDrumBeat

persistent pGeneratorParameters pMean pSTD
if isempty(pGeneratorParameters)
    % If the MAT file does not exist, download it
    filename = 'drumGeneratorWeights.mat';
    load(filename,'SMean','SStd','generatorParameters');
    pMean = SMean;
    pSTD = SStd;
    pGeneratorParameters = generatorParameters;
end

% Generate random vector
dlZ = dlarray(2 * ( rand(1,1,100,1,'single') - 0.5 ));

% Generate spectrograms
dlXGenerated = modelGenerator(dlZ,pGeneratorParameters);

% Convert from dlarray to single
S = dlXGenerated.extractdata;

S = S.';
% Zero-pad to remove edge effects
S = [S ; zeros(1,128)];

% Reverse steps from training
S = S * 3;
S = (S.*pSTD) + pMean;
```

```

S = exp(S);

% Make it two-sided
S = [S ; S(end-1:-1:2,:)];
% Pad with zeros at end and start
S = [zeros(256,100) S zeros(256,100)];

% Reconstruct the signal using a fast Griffin-Lim algorithm.
myAudio = stftmag2sig(gather(S),256, ...
    'FrequencyRange','twosided', ...
    'Window',hann(256,'periodic'), ...
    'OverlapLength',128, ...
    'MaxIterations',20, ...
    'Method','fgla');
myAudio = myAudio./max(abs(myAudio),[],'all');
y = myAudio(128*100:end-128*100);
end

```

Utility Functions

```

function out = sigmoid_cross_entropy_with_logits(x,z)
out = max(x, 0) - x .* z + log(1 + exp(-abs(x)));
end

function w = iGlorotInitialize(sz)
if numel(sz) == 2
    numInputs = sz(2);
    numOutputs = sz(1);
else
    numInputs = prod(sz(1:3));
    numOutputs = prod(sz([1 2 4]));
end
multiplier = sqrt(2 / (numInputs + numOutputs));
w = multiplier * sqrt(3) * (2 * rand(sz,'single') - 1);
end

```

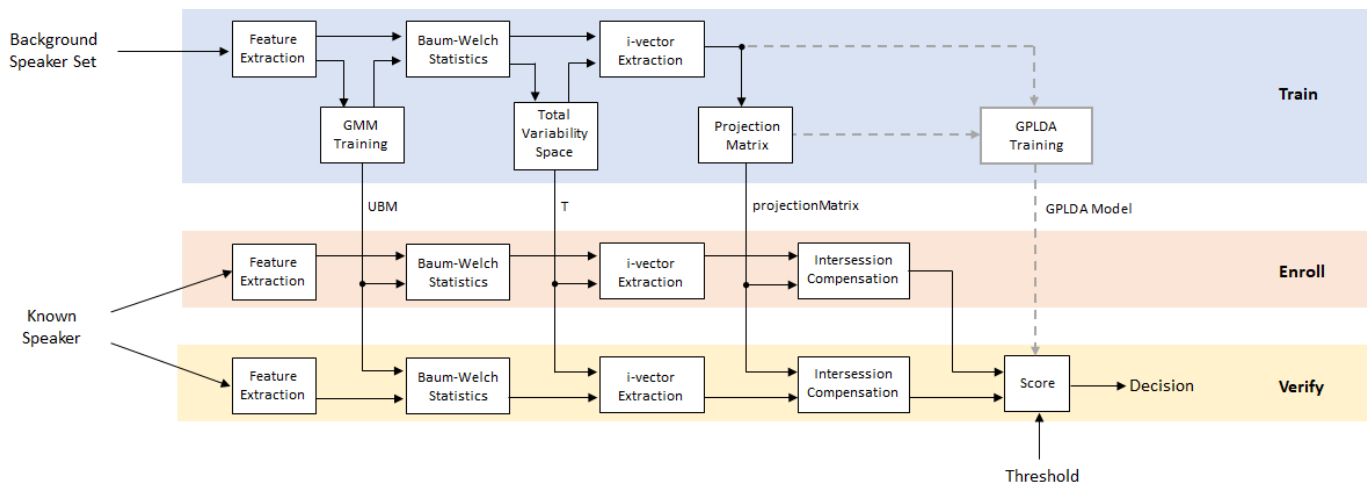
Reference

[1] Donahue, C., J. McAuley, and M. Puckette. 2019. "Adversarial Audio Synthesis." ICLR.

Speaker Verification Using i-Vectors

Speaker verification, or authentication, is the task of confirming that the identity of a speaker is who they purport to be. Speaker verification has been an active research area for many years. An early performance breakthrough was to use a Gaussian mixture model and universal background model (GMM-UBM) [1] on page 1-0 on acoustic features (usually mfcc). For an example, see “Speaker Verification Using Gaussian Mixture Model” on page 1-546. One of the main difficulties of GMM-UBM systems involves intersession variability. Joint factor analysis (JFA) was proposed to compensate for this variability by separately modeling inter-speaker variability and channel or session variability [2] on page 1-0 [3] on page 1-0. However, [4] on page 1-0 discovered that channel factors in the JFA also contained information about the speakers, and proposed combining the channel and speaker spaces into a *total variability* space. Intersession variability was then compensated for by using backend procedures, such as linear discriminant analysis (LDA) and within-class covariance normalization (WCCN), followed by a scoring, such as the cosine similarity score. [5] on page 1-0 proposed replacing the cosine similarity scoring with a probabilistic LDA (PLDA). [11] on page 1-0 and [12] on page 1-0 proposed a method to Gaussianize the i-vectors and therefore make Gaussian assumptions in the PLDA, referred to as G-PLDA or simplified PLDA. Further described the common While i-vectors were originally proposed for speaker verification, they have been applied to many problems, like language recognition, speaker diarization, emotion recognition, age estimation, and anti-spoofing [10] on page 1-0. Recently, deep learning techniques have been proposed to replace i-vectors with *d-vectors* or *x-vectors* [8] on page 1-0 [6] on page 1-0.

In this example, you develop a simple i-vector system for speaker verification that uses an LDA-WCCN backend with either cosine similarity scoring or a G-PLDA scoring.



Throughout the example, you will find live controls on tunable parameters. Changing the controls does not rerun the example. If you change a control, you must rerun the example.

Data Set Management

This example uses the Pitch Tracking Database from Graz University of Technology (PTDB-TUG) [7] on page 1-0. The data set consists of 20 English native speakers reading 2342 phonetically rich sentences from the TIMIT corpus. Download and extract the data set. Depending on your system, downloading and extracting the data set can take approximately 1.5 hours.

```
url = 'https://www2.spsec.tugraz.at/databases/PTDB-TUG/SPEECH_DATA_ZIPPED.zip';
downloadFolder = tempdir;
```

```
datasetFolder = fullfile(downloadFolder, 'PTDB-TUG');

if ~exist(datasetFolder, 'dir')
    disp('Downloading PTDB-TUG (3.9 G) ...')
    unzip(url, datasetFolder)
end
```

Create an `audioDatastore` object that points to the data set. The data set was originally intended for use in pitch-tracking training and evaluation, and includes laryngograph readings and baseline pitch decisions. Use only the original audio recordings.

```
ads = audioDatastore([fullfile(datasetFolder, "SPEECH DATA", "FEMALE", "MIC"), fullfile(datasetFolder, "SPEECH DATA", "MALE", "MIC")], ...
    'IncludeSubfolders', true, ...
    'FileExtensions', '.wav');
fileNames = ads.Files;
```

The file names contain the speaker IDs. Decode the file names to set the labels on the `audioDatastore` object.

```
speakerIDs = extractBetween(fileNames, 'mic_', '_');
ads.Labels = categorical(speakerIDs);
countEachLabel(ads)
```

```
ans=20x2 table
    Label    Count
    _____
    F01      236
    F02      236
    F03      236
    F04      236
    F05      236
    F06      236
    F07      236
    F08      234
    F09      236
    F10      236
    M01      236
    M02      236
    M03      236
    M04      236
    M05      236
    M06      236
    :
```

Separate the `audioDatastore` object in two: one for training, and one for enrollment and verification. The training set contains 16 speakers. The enrollment and verification set contains the other four speakers. You will split the enrollment and verification set later in the example.

```
speakersToTest = categorical(["M01", "M05", "F01", "F05"]);
adsTrain = subset(ads, ~ismember(ads.Labels, speakersToTest));
adsEnrollAndVerify = subset(ads, ismember(ads.Labels, speakersToTest));
adsTrain = shuffle(adsTrain);
adsEnrollAndVerify = shuffle(adsEnrollAndVerify);
```

Display the label distributions of the two `audioDatastore` objects.

```
countEachLabel(adsTrain)
```

```
ans=16×2 table
    Label    Count
    _____
    F02      236
    F03      236
    F04      236
    F06      236
    F07      236
    F08      234
    F09      236
    F10      236
    M02      236
    M03      236
    M04      236
    M06      236
    M07      236
    M08      236
    M09      236
    M10      236
```

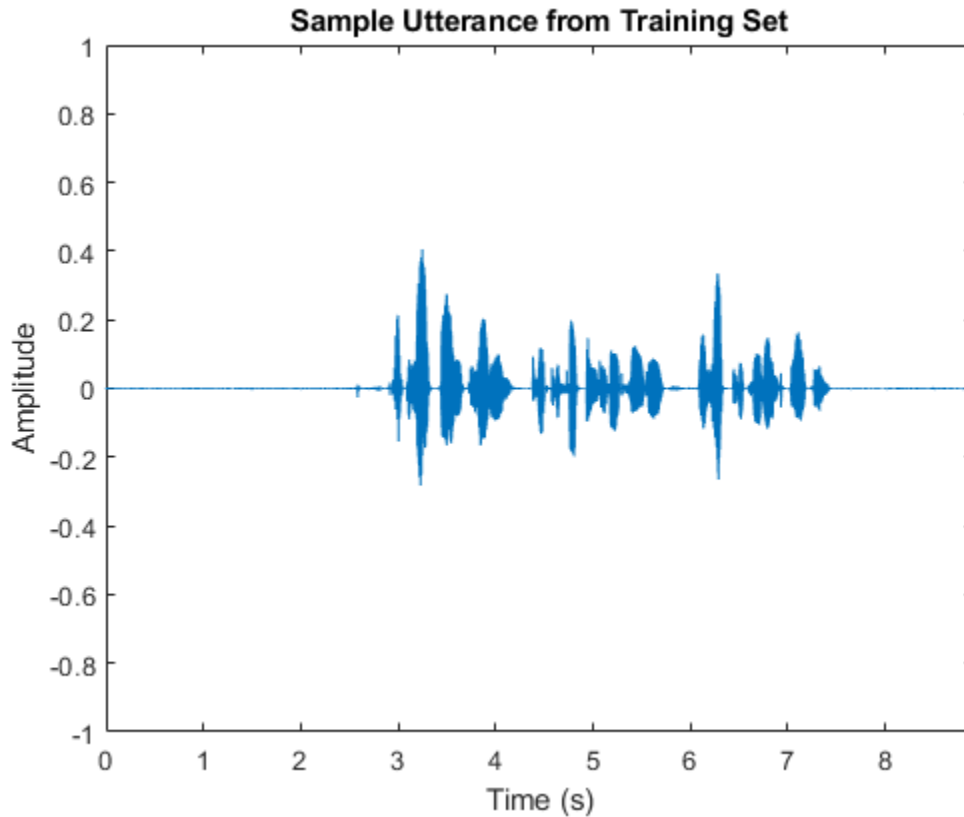
```
countEachLabel(adsEnrollAndVerify)
```

```
ans=4×2 table
    Label    Count
    _____
    F01      236
    F05      236
    M01      236
    M05      236
```

Read an audio file from the training data set, listen to it, and plot it. Reset the datastore.

```
[audio, audioInfo] = read(adsTrain);
fs = audioInfo.SampleRate;

t = (0:size(audio,1)-1)/fs;
sound(audio, fs)
plot(t, audio)
xlabel('Time (s)')
ylabel('Amplitude')
axis([0 t(end) -1 1])
title('Sample Utterance from Training Set')
```



```
reset(adsTrain)
```

You can reduce the data set and the number of parameters used in this example to speed up the runtime at the cost of performance. In general, reducing the data set is a good practice for development and debugging.

```
speedUpExample =  ;
if speedUpExample
    adsTrain = splitEachLabel(adsTrain,30);
    adsEnrollAndVerify = splitEachLabel(adsEnrollAndVerify,21);
end
```

Feature Extraction

Create an `audioFeatureExtractor` object to extract 20 MFCCs, 20 delta-MFCCs, and 20 delta-delta MFCCs. Use a delta window length of 9. Extract features from 25 ms Hann windows with a 10 ms hop.

```
numCoeffs = 20  ;
deltaWindowLength = 9  ;
windowDuration = 0.025  ;
hopDuration = 0.01  ;
```

```
windowSamples = round(windowDuration*fs);
hopSamples = round(hopDuration*fs);
overlapSamples = windowSamples - hopSamples;

afe = audioFeatureExtractor( ...
    'SampleRate',fs, ...
    'Window',hann(windowSamples,'periodic'), ...
    'OverlapLength',overlapSamples, ...
    ...
    'mfcc',true, ...
    'mfccDelta',true, ...
    'mfccDeltaDelta',true);
setExtractorParams(afe,'mfcc','DeltaWindowLength',deltaWindowLength,'NumCoeffs',numCoeffs)
```

Extract features from the audio read from the training datastore. Features are returned as a numHops-by-numFeatures matrix.

```
features = extract(afe,audio);
[numHops,numFeatures] = size(features)

numHops = 889
numFeatures = 60
```

Training

Training an i-vector system is computationally expensive and time-consuming. If you have Parallel Computing Toolbox™, you can spread the work across multiple cores to speed up the example. Determine the optimal number of partitions for your system. If you do not have Parallel Computing Toolbox™, use a single partition.

```
if ~isempty(ver('parallel')) && ~speedUpExample
    pool = gcp;
    numPar = numpartitions(adsTrain,pool);
else
    numPar = 1;
end
```

Feature Normalization Factors

Use the helper function, `helperFeatureExtraction`, to extract all features from the data set. The `helperFeatureExtraction` on page 1-0 function extracts MFCC from regions of speech in the audio. The speech detection is performed by the `detectSpeech` function.

```
featuresAll = {};
tic
parfor ii = 1:numPar
    adsPart = partition(adsTrain,numPar,ii);
    featuresPart = cell(0,numel(adsPart.Files));
    for iii = 1:numel(adsPart.Files)
        audioData = read(adsPart);
        featuresPart{iii} = helperFeatureExtraction(audioData,afe,[]);
    end
    featuresAll = [featuresAll,featuresPart];
end
allFeatures = cat(2,featuresAll{:});
fprintf('Feature extraction from training set complete (%0.0f seconds).',toc)
```


Feature extraction from training set complete (65 seconds).

Calculate the global mean and standard deviation of each feature. You will use these in future calls to the `helperFeatureExtraction` function to normalize the features.

```
normFactors.Mean = mean(allFeatures,2,'omitnan');
normFactors.STD = std(allFeatures,[],2,'omitnan');
```

Universal Background Model (UBM)

Initialize the Gaussian mixture model (GMM) that will be the universal background model (UBM) in the i-vector system. The component weights are initialized as evenly distributed. Systems trained on the TIMIT data set usually contain around 2048 components.

```
numComponents = 128 ;
if speedUpExample
    numComponents = 64;
end
alpha = ones(1,numComponents)/numComponents;
mu = randn(numFeatures,numComponents);
vari = rand(numFeatures,numComponents) + eps;
ubm = struct('ComponentProportion',alpha,'mu',mu,'sigma',vari);
```

Train the UBM using the expectation-maximization (EM) algorithm.

```
maxIter = 3 ;
if speedUpExample
    maxIter = 2;
end
tic
for iter = 1:maxIter
    tic
    % EXPECTATION
    N = zeros(1,numComponents);
    F = zeros(numFeatures,numComponents);
    S = zeros(numFeatures,numComponents);
    L = 0;
    parfor ii = 1:numPar
        adsPart = partition(adsTrain,numPar,ii);
        while hasdata(adsPart)
            audioData = read(adsPart);

            % Extract features
            Y = helperFeatureExtraction(audioData,afe,normFactors);

            % Compute a posteriori log-likelihood
            logLikelihood = helperGMMLogLikelihood(Y,ubm);

            % Compute a posteriori normalized probability
            amax = max(logLikelihood,[],1);
            logLikelihoodSum = amax + log(sum(exp(logLikelihood-amax),1));
            gamma = exp(logLikelihood - logLikelihoodSum)';

            % Compute Baum-Welch statistics
            n = sum(gamma,1);
            f = Y * gamma;
```

```

        s = (Y.*Y) * gamma;

        % Update the sufficient statistics over utterances
        N = N + n;
        F = F + f;
        S = S + s;

        % Update the log-likelihood
        L = L + sum(logLikelihoodSum);
    end
end

% Print current log-likelihood
fprintf('Training UBM: %d/%d complete (%0.0f seconds), Log-likelihood = %0.0f\n',iter,maxIter,logLikelihoodSum/L);

% MAXIMIZATION
N = max(N,eps);
ubm.ComponentProportion = max(N/sum(N),eps);
ubm.ComponentProportion = ubm.ComponentProportion/sum(ubm.ComponentProportion);
ubm.mu = F./N;
ubm.sigma = max(S./N - ubm.mu.^2,eps);
end

Training UBM: 1/3 complete (86 seconds), Log-likelihood = -149252841
Training UBM: 2/3 complete (85 seconds), Log-likelihood = -80011645
Training UBM: 3/3 complete (84 seconds), Log-likelihood = -76235688

```

Calculate Baum-Welch Statistics

The Baum-Welch statistics are the N (zeroth order) and F (first order) statistics used in the EM algorithm, calculated using the final UBM.

$$N_c(s) = \sum_t \gamma_t(c)$$

$$F_c(s) = \sum_t \gamma_t(c) Y_t$$

- Y_t is the feature vector at time t .
- $s \in \{s_1, s_2, \dots, s_N\}$, where N is the number of speakers. For the purposes of training the total variability space, each audio file is considered a separate speaker (whether or not it belongs to a physical single speaker).
- $\gamma_t(c)$ is the posterior probability that the UBM component c accounts for the feature vector Y_t .

Calculate the zeroth and first order Baum-Welch statistics over the training set.

```

numSpeakers = numel(adsTrain.Files);
Nc = {};
Fc = {};

tic
parfor ii = 1:numPar
    adsPart = partition(adsTrain,numPar,ii);
    numFiles = numel(adsPart.Files);

    Npart = cell(1,numFiles);

```

```

Fpart = cell(1,numFiles);
for jj = 1:numFiles
    audioData = read(adsPart);

    % Extract features
    Y = helperFeatureExtraction(audioData,afe,normFactors);

    % Compute a posteriori log-likelihood
    logLikelihood = helperGMMLogLikelihood(Y,ubm);

    % Compute a posteriori normalized probability
    amax = max(logLikelihood,[],1);
    logLikelihoodSum = amax + log(sum(exp(logLikelihood-amax),1));
    gamma = exp(logLikelihood - logLikelihoodSum)';

    % Compute Baum-Welch statistics
    n = sum(gamma,1);
    f = Y * gamma;

    Npart{jj} = reshape(n,1,1,numComponents);
    Fpart{jj} = reshape(f,numFeatures,1,numComponents);
end
Nc = [Nc,Npart];
Fc = [Fc,Fpart];
end
fprintf('Baum-Welch statistics completed (%0.0f seconds).\n',toc)
Baum-Welch statistics completed (82 seconds).

```

Expand the statistics into matrices and center $F(s)$, as described in [3] on page 1-0 , such that

- $N(s)$ is a $CF \times CF$ diagonal matrix whose blocks are $N_c(s)I$ ($c = 1, \dots, C$).
- $F(s)$ is a $CF \times 1$ supervector obtained by concatenating $F_c(s)$ ($c = 1, \dots, C$).
- C is the number of components in the UBM.
- F is the number of features in a feature vector.

```

N = Nc;
F = Fc;
muc = reshape(ubm.mu,numFeatures,1,[]);
for s = 1:numSpeakers
    N{s} = repelem(reshape(Nc{s},1,[]),numFeatures);
    F{s} = reshape(Fc{s} - Nc{s}.*muc,[],1);
end

```

Because this example assumes a diagonal covariance matrix for the UBM, N are also diagonal matrices, and are saved as vectors for efficient computation.

Total Variability Space

In the i-vector model, the ideal speaker supervector consists of a speaker-independent component and a speaker-dependent component. The speaker-dependent component consists of the total variability space model and the speaker's i-vector.

$$M = m + Tw$$

- M is the speaker utterance supervector

- m is the speaker- and channel-independent supervector, which can be taken to be the UBM supervector.
- T is a low-rank rectangular matrix and represents the total variability subspace.
- w is the i-vector for the speaker

The dimensionality of the i-vector, w , is typically much lower than the $C \times F$ -dimensional speaker utterance supervector, making the i-vector, or i-vectors, a much more compact and tractable representation.

To train the total variability space, T , first randomly initialize T , then perform these steps iteratively [3] on page 1-0 :

- 1 Calculate the posterior distribution of the hidden variable.

$$l_T(s) = I + T' \times \Sigma^{-1} \times N(s) \times T$$

2. Accumulate statistics across the speakers.

$$K = \sum_s F(s) \times (l_T^{-1}(s) \times T' \times \Sigma^{-1} \times F(s))'$$

$$A_c = \sum_s N_c(s) l_T^{-1}(s)$$

3. Update the total variability space.

$$T_c = A_c^{-1} \times K$$

$$T = \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_C \end{bmatrix}$$

[3] on page 1-0 proposes initializing Σ by the UBM variance, and then updating Σ according to the equation:


$$\Sigma = \left(\sum_s N(s) \right)^{-1} \left(\left(\sum_s S(s) \right) - \text{diag}(K \times T') \right)$$

where $S(s)$ is the centered second-order Baum-Welch statistic. However, updating Σ is often dropped in practice as it has little effect. This example does not update Σ .

Create the sigma variable.

```
Sigma = ubm.sigma(:);
```

Specify the dimension of the total variability space. A typical value used for the TIMIT data set is 1000.

```
numTdim = 128  ;
if speedUpExample
    numTdim = 64;
end
```

Initialize T and the identity matrix, and preallocate cell arrays.

```
T = randn(numel(ubm.sigma), numTdim);
T = T/norm(T);
```

```
I = eye(numTdim);
```

```
Ey = cell(numSpeakers,1);
Eyy = cell(numSpeakers,1);
Linv = cell(numSpeakers,1);
```

Set the number of iterations for training. A typical value reported is 20.

```
numIterations = 3  ;
```

Run the training loop.

```
for iterIdx = 1:numIterations
    tic

    % 1. Calculate the posterior distribution of the hidden variable
    TtimesInverseSSdiag = (T./Sigma)';
    parfor s = 1:numSpeakers
        L = (I + TtimesInverseSSdiag.*N{s}*T);
        Linv{s} = pinv(L);
        Ey{s} = Linv{s}*TtimesInverseSSdiag*F{s};
        Eyy{s} = Linv{s} + Ey{s}*Ey{s}';
    end

    % 2. Accumlate statistics across the speakers
    Eymat = cat(2,Ey{:});
    FFmat = cat(2,F{:});
    Kt = FFmat*Eymat';
    K = mat2cell(Kt', numTdim, repelem(numFeatures, numComponents));

    newT = cell(numComponents,1);
    for c = 1:numComponents
        AcLocal = zeros(numTdim);
        for s = 1:numSpeakers
            AcLocal = AcLocal + Nc{s}(:, :, c)*Eyy{s};
        end

        % 3. Update the Total Variability Space
        newT{c} = (pinv(AcLocal)*K{c})';
    end
    T = cat(1,newT{:});

    fprintf('Training Total Variability Space: %d/%d complete (%0.0f seconds).\n', iterIdx, numIterations, tic-toc);
end

Training Total Variability Space: 1/3 complete (27 seconds).
Training Total Variability Space: 2/3 complete (28 seconds).
Training Total Variability Space: 3/3 complete (26 seconds).
```

i-Vector Extraction

Once the total variability space is calculated, you can calculate the i-vectors as [4] on page 1-0 :

$$w = (I + T' \Sigma^{-1} N T)' T' \Sigma^{-1} F$$

At this point, you are still considering each training file as a separate speaker. However, in the next step, when you train a projection matrix to reduce dimensionality and increase inter-speaker differences, the i-vectors must be labeled with the appropriate, distinct speaker IDs.

Create a cell array where each element of the cell array contains a matrix of i-vectors across files for a particular speaker.

```
speakers = unique(adsTrain.Labels);
numSpeakers = numel(speakers);
ivectorPerSpeaker = cell(numSpeakers,1);
TS = T./Sigma;
TSi = TS';
ubmMu = ubm.mu;
tic
parfor speakerIdx = 1:numSpeakers

    % Subset the datastore to the speaker you are adapting.
    adsPart = subset(adsTrain,adsTrain.Labels==speakers(speakerIdx));
    numFiles = numel(adsPart.Files);

    ivectorPerFile = zeros(numTdim,numFiles);
    for fileIdx = 1:numFiles
        audioData = read(adsPart);

        % Extract features
        Y = helperFeatureExtraction(audioData,afe,normFactors);

        % Compute a posteriori log-likelihood
        logLikelihood = helperGMMLogLikelihood(Y,ubm);

        % Compute a posteriori normalized probability
        amax = max(logLikelihood,[],1);
        logLikelihoodSum = amax + log(sum(exp(logLikelihood-amax),1));
        gamma = exp(logLikelihood - logLikelihoodSum)';

        % Compute Baum-Welch statistics
        n = sum(gamma,1);
        f = Y * gamma - n.*(ubmMu);

        ivectorPerFile(:,fileIdx) = pinv(I + (TS.*repelem(n(:),numFeatures))' * T) * TSi * f(:);
    end
    ivectorPerSpeaker{speakerIdx} = ivectorPerFile;
end
fprintf('I-vectors extracted from training set (%0.0f seconds).\n',toc)

I-vectors extracted from training set (110 seconds).
```

Projection Matrix

Many different backends have been proposed for i-vectors. The most straightforward and still well-performing one is the combination of linear discriminant analysis (LDA) and within-class covariance normalization (WCCN).

Create a matrix of the training vectors and a map indicating which i-vector corresponds to which speaker. Initialize the projection matrix as an identity matrix.

```
w = ivectorPerSpeaker;
utterancePerSpeaker = cellfun(@(x)size(x,2),w);

ivectorsTrain = cat(2,w{:});
projectionMatrix = eye(size(w{1},1));
```

LDA attempts to minimize the intra-class variance and maximize the variance between speakers. It can be calculated as outlined in [4] on page 1-0 :

Given:

$$S_b = \sum_{s=1}^S (\bar{w}_s - \bar{w})(\bar{w}_s - \bar{w})'$$

$$S_w = \sum_{s=1}^S \frac{1}{n_s} \sum_{i=1}^{n_s} (w_i^s - \bar{w}_s)(w_i^s - \bar{w}_s)'$$

where

- $\bar{w}_s = \left(\frac{1}{n_s}\right) \sum_{i=1}^{n_s} w_i^s$ is the mean of i-vectors for each speaker.
- $\bar{w} = \frac{1}{N} \sum_{s=1}^S \sum_{i=1}^{n_s} w_i^s$ is the mean i-vector across all speakers.
- n_s is the number of utterances for each speaker.

Solve the eigenvalue equation for the best eigenvectors:

$$S_b v = \lambda S_w v$$

The best eigenvectors are those with the highest eigenvalues.

```
performLDA =  ;
if performLDA
    tic

    numEigenvectors =  ;

    Sw = zeros(size(projectionMatrix,1));
    Sb = zeros(size(projectionMatrix,1));
    wbar = mean(cat(2,w{:}),2);
    for ii = 1:numel(w)
        ws = w{ii};
        wsbar = mean(ws,2);
        Sb = Sb + (wsbar - wbar)*(wsbar - wbar)';
        Sw = Sw + cov(ws',1);
    end

    [A,~] = eigs(Sb,Sw,numEigenvectors);
    A = (A./vecnorm(A))';

    ivectorsTrain = A * ivectorsTrain;

    w = mat2cell(ivectorsTrain,size(ivectorsTrain,1),utterancePerSpeaker);
```

```

projectionMatrix = A * projectionMatrix;

fprintf('LDA projection matrix calculated (%0.2f seconds).',toc)
end

LDA projection matrix calculated (0.06 seconds).

```

WCCN attempts to scale the i-vector space inversely to the in-class covariance, so that directions of high intra-speaker variability are de-emphasized in i-vector comparisons [9] on page 1-0 .

Given the within-class covariance matrix:

$$W = \frac{1}{S} \sum_{s=1}^S \frac{1}{n_s} \sum_{i=1}^{n_s} (w_i^s - \overline{w_s})(w_i^s - \overline{w_s})'$$


where

- $\overline{w_s} = \left(\frac{1}{n_s}\right) \sum_{i=1}^{n_s} w_i^s$ is the mean of i-vectors for each speaker.
- n_s is the number of utterances for each speaker.

Solve for B using Cholesky decomposition:

```

W-1 = BB'

performWCCN = ☐ true ;
if performWCCN
    tic
    alpha = 0.9  ;

    W = zeros(size(projectionMatrix,1));
    for ii = 1:numel(w)
        W = W + cov(w{ii}',1);
    end
    W = W/numel(w);

    W = (1 - alpha)*W + alpha*eye(size(W,1));

    B = chol(pinv(W), 'lower');

    projectionMatrix = B * projectionMatrix;

    fprintf('WCCN projection matrix calculated (%0.4f seconds).',toc)
end

WCCN projection matrix calculated (0.0033 seconds).

```

The training stage is now complete. You can now use the universal background model (UBM), total variability space (T), and projection matrix to enroll and verify speakers.

Train G-PLDA Model

Apply the projection matrix to the train set.

```
ivectors = cellfun(@(x)projectionMatrix*x,ivectorPerSpeaker,'UniformOutput',false);
```


This algorithm implemented in this example is a Gaussian PLDA as outlined in [13] on page 1-0 . In the Gaussian PLDA, the i-vector is represented with the following equation:

$$\phi_{ij} = \mu + Vy_i + \varepsilon_{ij}$$

$$y_i \sim N(0, I)$$

$$\varepsilon_{ij} \sim N(0, \Lambda^{-1})$$

where μ is a global mean of the i-vectors, Λ is a full precision matrix of the noise term ε_{ij} , and V is the factor loading matrix, also known as the eigenvoices.

Specify the number of eigenvoices to use. Typically numbers are between 10 and 400.

numEigenVoices = 16 ;

Determine the number of disjoint persons, the number of dimensions in the feature vectors, and the number of utterances per speaker.

```
K = numel(ivectors);
D = size(ivectors{1},1);
utterancePerSpeaker = cellfun(@(x)size(x,2),ivectors);
```

Find the total number of samples and center the i-vectors.

$$N = \sum_{i=1}^K n_i$$

$$\mu = \frac{1}{N} \sum_{i,j} \phi_{i,j}$$

$$\varphi_{ij} = \phi_{ij} - \mu$$

```
ivectorsMatrix = cat(2,ivectors{:});
N = size(ivectorsMatrix,2);
mu = mean(ivectorsMatrix,2);
```

```
ivectorsMatrix = ivectorsMatrix - mu;
```

Determine a whitening matrix from the training i-vectors and then whiten the i-vectors. Specify either ZCA whitening, PCA whitening, or no whitening.

whiteningType = ;

```
if strcmpi(whiteningType,'ZCA')
    S = cov(ivectorsMatrix');
    [~,sD,sV] = svd(S);
    W = diag(1./(sqrt(diag(sD)) + eps))*sV';
    ivectorsMatrix = W * ivectorsMatrix;
elseif strcmpi(whiteningType,'PCA')
    S = cov(ivectorsMatrix');
    [sV,sD] = eig(S);
    W = diag(1./(sqrt(diag(sD)) + eps))*sV';
```

```

        ivectorsMatrix = W * ivectorsMatrix;
    else
        W = eye(size(ivectorsMatrix,1));
    end

```

Apply length normalization and then convert the training i-vector matrix back to a cell array.

```
ivectorsMatrix = ivectorsMatrix./vecnorm(ivectorsMatrix);
```

Compute the global second-order moment as

$$S = \sum_{ij} \phi_{ij} \phi_{ij}^T$$

```
S = ivectorsMatrix*ivectorsMatrix';
```

Convert the training i-vector matrix back to a cell array.

```
ivectors = mat2cell(ivectorsMatrix,D,utterancePerSpeaker);
```

Sort persons according to the number of samples and then group the i-vectors by number of utterances per speaker. Precalculate the first-order moment of the i -th person as

$$f_i = \sum_{j=1}^{n_i} \phi_{ij}$$

```
uniqueLengths = unique(utterancePerSpeaker);
numUniqueLengths = numel(uniqueLengths);
```

```

speakerIdx = 1;
f = zeros(D,K);
for uniqueLengthIdx = 1:numUniqueLengths
    idx = find(utterancePerSpeaker==uniqueLengths(uniqueLengthIdx));
    temp = {};
    for speakerIdxWithinUniqueLength = 1:numel(idx)
        rho = ivectors(idx(speakerIdxWithinUniqueLength));
        temp = [temp;rho]; %#ok<AGROW>

        f(:,speakerIdx) = sum(rho{:,2});
        speakerIdx = speakerIdx+1;
    end
    ivectorsSorted{uniqueLengthIdx} = temp; %#ok<SAGROW>
end

```

Initialize the eigenvoices matrix, V , and the inverse noise variance term, Λ .



```

V = randn(D,numEigenVoices);
Lambda = pinv(S/N);

```

Specify the number of iterations for the EM algorithm and whether or not to apply the minimum divergence.

```

numIter = 5  _____ ;
minimumDivergence =  true ;

```

Train the G-PLDA model using the EM algorithm described in [13] on page 1-0 .

```

for iter = 1:numIter
    % EXPECTATION
    gamma = zeros(numEigenVoices,numEigenVoices);
    EyTotal = zeros(numEigenVoices,K);
    R = zeros(numEigenVoices,numEigenVoices);

    idx = 1;
    for lengthIndex = 1:numUniqueLengths
        ivectorLength = uniqueLengths(lengthIndex);

        % Isolate i-vectors of the same given length
        iv = ivectorsSorted{lengthIndex};

        % Calculate M
        M = pinv(ivectorLength*(V'*(Lambda*V)) + eye(numEigenVoices)); % Equation (A.7) in [13]

        % Loop over each speaker for the current i-vector length
        for speakerIndex = 1:numel(iv)

            % First moment of latent variable for V
            Ey = M*V'*Lambda*f(:,idx); % Equation (A.8) in [13]

            % Calculate second moment.
            Eyy = Ey * Ey';

            % Update Ryy
            R = R + ivectorLength*(M + Eyy); % Equation (A.13) in [13]

            % Append EyTotal
            EyTotal(:,idx) = Ey;
            idx = idx + 1;

            % If using minimum divergence, update gamma.
            if minimumDivergence
                gamma = gamma + (M + Eyy); % Equation (A.18) in [13]
            end
        end
    end

    % Calculate T
    TT = EyTotal*f'; % Equation (A.12) in [13]

    % MAXIMIZATION
    V = TT'*pinv(R); % Equation (A.16) in [13]
    Lambda = pinv((S - V*TT)/N); % Equation (A.17) in [13]

    % MINIMUM DIVERGENCE
    if minimumDivergence
        gamma = gamma/K; % Equation (A.18) in [13]
        V = V*chol(gamma,'lower'); % Equation (A.22) in [13]
    end
end
end



```



Once you've trained the G-PLDA model, you can use it to calculate a score based on the log-likelihood ratio as described in [14] on page 1-0 . Given two i-vectors that have been centered, whitened, and length-normalized, the score is calculated as:

$$\text{score}(w_1, w_t) = \begin{bmatrix} w_1^T & w_t^T \end{bmatrix} \begin{bmatrix} \Sigma + VV^T & VV^T \\ VV^T & \Sigma + VV^T \end{bmatrix} \begin{bmatrix} w_1 \\ w_t \end{bmatrix} - w_1^T [\Sigma + VV^T]^{-1} w_1 - w_t^T [\Sigma + VV^T]^{-1} w_t + C$$

where w_1 and w_t are the enrollment and test i-vectors, Σ is the variance matrix of the noise term, V is the eigenvoice matrix. The C term are factored-out constants and can be dropped in practice.

```

speakerIdx = 2  ;
utteranceIdx = 1  ;
w1 = ivectors{speakerIdx}(:,utteranceIdx);

speakerIdx = 1  ;
utteranceIdx = 10  ;
wt = ivectors{speakerIdx}(:,utteranceIdx);

VVt = V*V';
SigmaPlusVVt = pinv(Lambda) + VVt;

term1 = pinv([SigmaPlusVVt VVt; VVt SigmaPlusVVt]);
term2 = pinv(SigmaPlusVVt);

wlwt = [w1;wt];
score = wlwt'*term1*wlwt - w1'*term2*w1 - wt'*term2*wt

score = 59.0358

```

In practice, the test i-vectors, and depending on your system, the enrollment ivectors, are not used in the training of the G-PLDA model. In the following evaluation section, you use previously unseen data for enrollment and verification. The supporting function, `gpldaScore` on page 1-0 encapsulates the scoring steps above, and additionally performs centering, whitening, and normalization. Save the trained G-PLDA model as a struct for use with the supporting function `gpldaScore`.

```


gpldaModel = struct('mu',mu, ...
    'WhiteningMatrix',W, ...
    'EigenVoices',V, ...
    'Sigma',pinv(Lambda));

```

Enroll

Enroll new speakers that were not in the training data set. First, split the `adsEnrollAndVerify` audio datastore object into enroll and verify. Increasing the number of utterances per speaker for enrollment should increase the performance of the system.

```

numFilesPerSpeakerForEnrollment = 3  ;
[adsEnroll,adsVerify] = splitEachLabel(adsEnrollAndVerify,numFilesPerSpeakerForEnrollment);
adsVerify = shuffle(adsVerify);
adsEnroll = shuffle(adsEnroll);

countEachLabel(adsEnroll)

ans=4x2 table
    Label    Count
    _____

```

```

F01      3
F05      3
M01      3
M05      3

```

```
countEachLabel(adsVerify)
```

```
ans=4x2 table
Label      Count
-----
F01        233
F05        233
M01        233
M05        233
```

Create i-vectors for each file for each speaker in the enroll set using the this sequence of steps:

- 1 Feature Extraction
- 2 Baum-Welch Statistics: Determine the zeroth and first order statistics
- 3 i-vector Extraction
- 4 Intersession compensation

Then average the i-vectors across files to create an i-vector model for the speaker. Repeat the for each speaker.

```

speakers = unique(adsEnroll.Labels);
numSpeakers = numel(speakers);
enrolledSpeakersByIdx = cell(numSpeakers,1);
tic
parfor speakerIdx = 1:numSpeakers
    % Subset the datastore to the speaker you are adapting.
    adsPart = subset(adsEnroll,adsEnroll.Labels==speakers(speakerIdx));
    numFiles = numel(adsPart.Files);

    ivectorMat = zeros(size(projectionMatrix,1),numFiles);
    for fileIdx = 1:numFiles
        audioData = read(adsPart);

        % Extract features
        Y = helperFeatureExtraction(audioData,afe,normFactors);

        % Compute a posteriori log-likelihood
        logLikelihood = helperGMMLogLikelihood(Y,ubm);

        % Compute a posteriori normalized probability
        amax = max(logLikelihood,[],1);
        logLikelihoodSum = amax + log(sum(exp(logLikelihood-amax),1));
        gamma = exp(logLikelihood - logLikelihoodSum)';

        % Compute Baum-Welch statistics
        n = sum(gamma,1);
        f = Y * gamma - n.*(ubmMu);
    end
end

```

```
%i-vector Extraction
ivector = pinv(I + (TS.*repelem(n(:),numFeatures))' * T) * TSi * f(:);

% Inter-session Compensation
ivector = projectionMatrix*ivector;

ivectorMat(:,fileIdx) = ivector;
end
% i-vector model
enrolledSpeakersByIdx{speakerIdx} = mean(ivectorMat,2);
end
fprintf('Speakers enrolled (%0.0f seconds).\n',toc)

Speakers enrolled (1 seconds).
```

For bookkeeping purposes, convert the cell array of i-vectors to a structure, with the speaker IDs as fields and the i-vectors as values

```
enrolledSpeakers = struct;
for s = 1:numSpeakers
    enrolledSpeakers.(string(speakers(s))) = enrolledSpeakersByIdx{s};
end
```

Verification

Specify either the CSS or G-PLDA scoring method.

```
scoringMethod = 
```

False Rejection Rate (FRR)

The speaker false rejection rate (FRR) is the rate that a given speaker is incorrectly rejected. Create an array of scores for enrolled speaker i-vectors and i-vectors of the same speaker.

```
speakersToTest = unique(adsVerify.Labels);
numSpeakers = numel(speakersToTest);
scoreFRR = cell(numSpeakers,1);
tic
parfor speakerIdx = 1:numSpeakers
    adsPart = subset(adsVerify,adsVerify.Labels==speakersToTest(speakerIdx));
    numFiles = numel(adsPart.Files);

    ivectorToTest = enrolledSpeakers.(string(speakersToTest(speakerIdx))); %#ok<PFBNS>

    score = zeros(numFiles,1);
    for fileIdx = 1:numFiles
        audioData = read(adsPart);

        % Extract features
        Y = helperFeatureExtraction(audioData,afe,normFactors);

        % Compute a posteriori log-likelihood
        logLikelihood = helperGMMLogLikelihood(Y,ubm);

        % Compute a posteriori normalized probability
        amax = max(logLikelihood,[],1);
        logLikelihoodSum = amax + log(sum(exp(logLikelihood-amax),1));
        gamma = exp(logLikelihood - logLikelihoodSum)';
```

```

% Compute Baum-Welch statistics
n = sum(gamma,1);
f = Y * gamma - n.*(ubmMu);

% Extract i-vector
ivector = pinv(I + (TS.*repelem(n(:),numFeatures))' * T) * TSi * f(:);

% Intersession Compensation
ivector = projectionMatrix*ivector;

% Score
if strcmpi(scoringMethod,'CSS')
    score(fileIdx) = dot(ivectorToTest,ivector)/(norm(ivector)*norm(ivectorToTest));
else
    score(fileIdx) = gpldaScore(gpldaModel,ivector,ivectorToTest);
end
end
scoreFRR{speakerIdx} = score;
end
fprintf('FRR calculated (%0.0f seconds).\n',toc)

FRR calculated (35 seconds).

```

False Acceptance Rate (FAR)

The speaker false acceptance rate (FAR) is the rate that utterances not belonging to an enrolled speaker are incorrectly accepted as belonging to the enrolled speaker. Create an array of scores for enrolled speakers and i-vectors of different speakers.

```

speakersToTest = unique(adsVerify.Labels);
numSpeakers = numel(speakersToTest);
scoreFAR = cell(numSpeakers,1);
tic
parfor speakerIdx = 1:numSpeakers
    adsPart = subset(adsVerify,adsVerify.Labels~=speakersToTest(speakerIdx));
    adsPart = splitEachLabel(adsPart,0.2) % Reduce the number of test files to speed up the exampl
    numFiles = numel(adsPart.Files);

    ivectorToTest = enrolledSpeakers.(string(speakersToTest(speakerIdx))); %#ok<PFBNS>
    score = zeros(numFiles,1);
    for fileIdx = 1:numFiles
        audioData = read(adsPart);

        % Extract features
        Y = helperFeatureExtraction(audioData,afe,normFactors);

        % Compute a posteriori log-likelihood
        logLikelihood = helperGMMLogLikelihood(Y,ubm);

        % Compute a posteriori normalized probability
        amax = max(logLikelihood,[],1);
        logLikelihoodSum = amax + log(sum(exp(logLikelihood-amax),1));
        gamma = exp(logLikelihood - logLikelihoodSum)';

        % Compute Baum-Welch statistics
        n = sum(gamma,1);
        f = Y * gamma - n.*(ubmMu);
    end
end
scoreFAR{speakerIdx} = score;
end

```

```
% Extract i-vector
ivector = pinv(I + (TS.*repelem(n(:),numFeatures))' * T) * TSi * f(:);

% Inter-session compensation
ivector = projectionMatrix * ivector;

% Score
if strcmpi(scoringMethod,'CSS')
    score(fileIdx) = dot(ivectorToTest,ivector)/(norm(ivector)*norm(ivectorToTest));
else
    score(fileIdx) = gpldaScore(gpldaModel,ivector,ivectorToTest);
end
end
scoreFAR{speakerIdx} = score;
end
```

Equal Error Rate (EER)

To compare multiple systems, you need a single metric that combines the FAR and FRR performance. For this, you determine the equal error rate (EER), which is the threshold where the FAR and FRR curves meet. In practice, the EER threshold might not be the best choice. For example, if speaker verification is used as part of a multi-authentication approach for wire transfers, FAR would most likely be more heavily weighted than FRR.

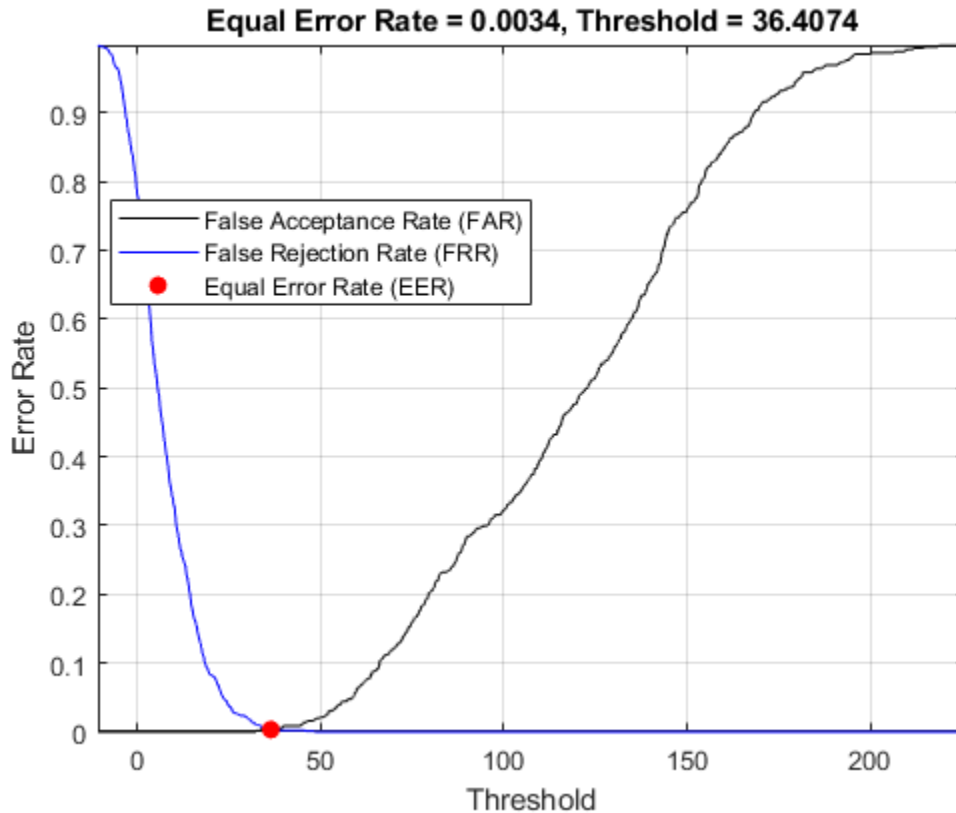
```
amin = min(cat(1,scoreFRR{:},scoreFAR{:}));
amax = max(cat(1,scoreFRR{:},scoreFAR{:}));

thresholdsToTest = linspace(amin,amax,1000);

% Compute the FRR and FAR for each of the thresholds.
if strcmpi(scoringMethod,'CSS')
    % In CSS, a larger score indicates the enroll and test ivectors are
    % similar.
    FRR = mean(cat(1,scoreFRR{:})<thresholdsToTest);
    FAR = mean(cat(1,scoreFAR{:})>thresholdsToTest);
else
    % In G-PLDA, a smaller score indicates the enroll and test ivectors are
    % similar.
    FRR = mean(cat(1,scoreFRR{:})>thresholdsToTest);
    FAR = mean(cat(1,scoreFAR{:})<thresholdsToTest);
end

[~,EERThresholdIdx] = min(abs(FAR - FRR));
EERThreshold = thresholdsToTest(EERThresholdIdx);
EER = mean([FAR(EERThresholdIdx),FRR(EERThresholdIdx)]);

figure
plot(thresholdsToTest,FAR,'k', ...
      thresholdsToTest,FRR,'b', ...
      EERThreshold,EER,'ro','MarkerFaceColor','r')
title(sprintf('Equal Error Rate = %0.4f, Threshold = %0.4f',EER,EERThreshold))
xlabel('Threshold')
ylabel('Error Rate')
legend('False Acceptance Rate (FAR)','False Rejection Rate (FRR)','Equal Error Rate (EER)','Location')
grid on
axis tight
```

Supporting Functions

Feature Extraction and Normalization

```
function [features,numFrames] = helperFeatureExtraction(audioData,afe,normFactors)
% Input:
% audioData - column vector of audio data
% afe       - audioFeatureExtractor object
% normFactors - mean and standard deviation of the features used for normalization.
%             If normFactors is empty, no normalization is applied.
%
% Output
% features   - matrix of features extracted
% numFrames  - number of frames (feature vectors) returned

% Normalize
audioData = audioData/max(abs(audioData(:)));

% Protect against NaNs
audioData(isnan(audioData)) = 0;

% Isolate speech segment
idx = detectSpeech(audioData,afe.SampleRate);
features = [];
for ii = 1:size(idx,1)
    f = extract(afe,audioData(idx(ii,1):idx(ii,2)));
    features = [features;f]; %#ok<AGROW>
```

```
end

% Feature normalization
if ~isempty(normFactors)
    features = (features-normFactors.Mean')./normFactors.STD';
end
features = features';

% Cepstral mean subtraction (for channel noise)
if ~isempty(normFactors)
    features = features - mean(features,'all');
end

numFrames = size(features,2);
end
```

Gaussian Multi-Component Mixture Log-Likelihood

```
function L = helperGMMLogLikelihood(x,gmm)
    xMinusMu = repmat(x,1,1,numel(gmm.ComponentProportion)) - permute(gmm.mu,[1,3,2]);
    permuteSigma = permute(gmm.sigma,[1,3,2]);

    Lunweighted = -0.5*(sum(log(permuteSigma),1) + sum(xMinusMu.*(xMinusMu./permuteSigma),1) + sum(xMinusMu.^2./permuteSigma,1));

    temp = squeeze(permute(Lunweighted,[1,3,2]));
    if size(temp,1)==1
        % If there is only one frame, the trailing singleton dimension was
        % removed in the permute. This accounts for that edge case.
        temp = temp';
    end

    L = temp + log(gmm.ComponentProportion)';
end
```

G-PLDA Score

```
function score = gpldaScore(gpldaModel,w1,wt)
% Center the data
w1 = w1 - gpldaModel.mu;
wt = wt - gpldaModel.mu;

% Whiten the data
w1 = gpldaModel.WhiteningMatrix*w1;
wt = gpldaModel.WhiteningMatrix*wt;

% Length-normalize the data
w1 = w1./vecnorm(w1);
wt = wt./vecnorm(wt);

% Score the similarity of the i-vectors based on the log-likelihood.
VVt = gpldaModel.EigenVoices * gpldaModel.EigenVoices';
SVVt = gpldaModel.Sigma + VVt;

term1 = pinv([SVVt VVt;VVt SVVt]);
term2 = pinv(SVVt);

wlwt = [w1;wt];
```

```
score = w1wt'*term1*w1wt - w1'*term2*w1 - wt'*term2*wt;
end
```

References

- [1] Reynolds, Douglas A., et al. "Speaker Verification Using Adapted Gaussian Mixture Models." *Digital Signal Processing*, vol. 10, no. 1-3, Jan. 2000, pp. 19-41. *DOI.org (Crossref)*, doi:10.1006/dspr.1999.0361.
- [2] Kenny, Patrick, et al. "Joint Factor Analysis Versus Eigenchannels in Speaker Recognition." *IEEE Transactions on Audio, Speech and Language Processing*, vol. 15, no. 4, May 2007, pp. 1435-47. *DOI.org (Crossref)*, doi:10.1109/TASL.2006.881693.
- [3] Kenny, P., et al. "A Study of Interspeaker Variability in Speaker Verification." *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 16, no. 5, July 2008, pp. 980-88. *DOI.org (Crossref)*, doi:10.1109/TASL.2008.925147.
- [4] Dehak, Najim, et al. "Front-End Factor Analysis for Speaker Verification." *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 19, no. 4, May 2011, pp. 788-98. *DOI.org (Crossref)*, doi:10.1109/TASL.2010.2064307.
- [5] Matejka, Pavel, Ondrej Glembek, Fabio Castaldo, M.j. Alam, Oldrich Plchot, Patrick Kenny, Lukas Burget, and Jan Cernocky. "Full-Covariance UBM and Heavy-Tailed PLDA in i-Vector Speaker Verification." *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2011. <https://doi.org/10.1109/icassp.2011.5947436>.
- [6] Snyder, David, et al. "X-Vectors: Robust DNN Embeddings for Speaker Recognition." *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2018, pp. 5329-33. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2018.8461375.
- [7] Signal Processing and Speech Communication Laboratory. Accessed December 12, 2019. <https://www.spsc.tugraz.at/databases-and-tools/ptdb-tug-pitch-tracking-database-from-graz-university-of-technology.html>.
- [8] Variani, Ehsan, et al. "Deep Neural Networks for Small Footprint Text-Dependent Speaker Verification." *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2014, pp. 4052-56. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2014.6854363.
- [9] Dehak, Najim, Réda Dehak, James R. Glass, Douglas A. Reynolds and Patrick Kenny. "Cosine Similarity Scoring without Score Normalization Techniques." *Odyssey* (2010).
- [10] Verma, Pulkit, and Pradip K. Das. "I-Vectors in Speech Processing Applications: A Survey." *International Journal of Speech Technology*, vol. 18, no. 4, Dec. 2015, pp. 529-46. *DOI.org (Crossref)*, doi:10.1007/s10772-015-9295-3.
- [11] D. Garcia-Romero and C. Espy-Wilson, "Analysis of I-vector Length Normalization in Speaker Recognition Systems." *Interspeech*, 2011, pp. 249-252.
- [12] Kenny, Patrick. "Bayesian Speaker Verification with Heavy-Tailed Priors". *Odyssey 2010 - The Speaker and Language Recognition Workshop*, Brno, Czech Republic, 2010.
- [13] Sizov, Aleksandr, Kong Aik Lee, and Tomi Kinnunen. "Unifying Probabilistic Linear Discriminant Analysis Variants in Biometric Authentication." *Lecture Notes in Computer Science Structural, Syntactic, and Statistical Pattern Recognition*, 2014, 464-75. https://doi.org/10.1007/978-3-662-44415-3_47.

[14] Rajan, Padmanabhan, Anton Afanasyev, Ville Hautamäki, and Tomi Kinnunen. 2014. "From Single to Multiple Enrollment I-Vectors: Practical PLDA Scoring Variants for Speaker Verification." *Digital Signal Processing* 31 (August): 93-101. <https://doi.org/10.1016/j.dsp.2014.05.001>.

Plugin GUI Design

Design User Interface for Audio Plugin

Audio plugins enable you to tune parameters of a processing algorithm while streaming audio in real time. To enhance usability, you can define a custom user interface (UI) that maps parameters to intuitively designed and positioned controls. You can use `audioPluginInterface`, `audioPluginParameter`, and `audioPluginGridLayout` to define the custom UI. You can interact with the custom UI in MATLAB® using `parameterTuner`, or deploy the plugin with a custom UI to a digital audio workstation (DAW). This tutorial walks through key design capabilities of audio plugins by sequentially enhancing a basic audio plugin UI.

To learn more about audio plugins in general, see “Audio Plugins in MATLAB”.

Default User Interface

The `equalizerV1` audio plugin enables you to tune the gains and center frequencies of a three-band equalizer, tune the overall volume, and toggle between enabled and disabled states.

```
classdef equalizerV1 < audioPlugin
    properties
        GainLow = 0
        FreqLow = sqrt(20*500)
        GainMid = 0
        FreqMid = sqrt(500*3e3)
        GainHigh = 0
        FreqHigh = sqrt(3e3*20e3)
        Volume = 1
        Enable = true
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('GainLow', ...
                'Label','dB', ...
                'Mapping',{'lin',-20,20}), ...
            audioPluginParameter('FreqLow', ...
                'Label','Hz', ...
                'Mapping',{'log',20,500}), ...
            audioPluginParameter('GainMid', ...
                'Label','dB', ...
                'Mapping',{'lin',-20,20}), ...
            audioPluginParameter('FreqMid', ...
                'Label','Hz', ...
                'Mapping',{'log',500,3e3}), ...
            audioPluginParameter('GainHigh', ...
                'Label','dB', ...
                'Mapping',{'lin',-20,20}), ...
            audioPluginParameter('FreqHigh', ...
                'Label','Hz', ...
                'Mapping',{'log',3e3,20e3}), ...
            audioPluginParameter('Volume', ...
                'Mapping',{'lin',0,2}), ...
            audioPluginParameter('Enable'))
    end
    properties (Access = private)
        mPEQ
    end
    methods
        function obj = equalizerV1
```

```

        obj.mPEQ = multibandParametricEQ('HasHighpassFilter',false, ...
            'HasLowShelfFilter',false,'HasHighShelfFilter',false, ...
            'HasLowpassFilter',false,'Oversample',false,'NumEQBands',3, ...
            'EQOrder',2);
    end
    function y = process(obj, x)
        if obj.Enable
            y = step(obj.mPEQ,x);
            y = y*obj.Volume;
        else
            y = x;
        end
    end
    function reset(obj)
        obj.mPEQ.SampleRate = getSampleRate(obj);
        reset(obj.mPEQ);
    end
    function set.FreqLow(obj,val)
        obj.FreqLow = val;
        obj.mPEQ.Frequencies(1) = val; %#ok<*>MCSUP>
    end
    function set.GainLow(obj,val)
        obj.GainLow = val;
        obj.mPEQ.PeakGains(1) = val;
    end
    function set.FreqMid(obj,val)
        obj.FreqMid = val;
        obj.mPEQ.Frequencies(2) = val;
    end
    function set.GainMid(obj,val)
        obj.GainMid = val;
        obj.mPEQ.PeakGains(2) = val;
    end
    function set.FreqHigh(obj,val)
        obj.FreqHigh = val;
        obj.mPEQ.Frequencies(3) = val;
    end
    function set.GainHigh(obj,val)
        obj.GainHigh = val;
        obj.mPEQ.PeakGains(3) = val;
    end
end
end

```

Call parameterTuner to visualize the default UI of the audio plugin.

```
parameterTuner(equalizerV1)
```

GainLow 0 dB

FreqLow 100 Hz

GainMid 0 dB

FreqMid 1224.7 Hz

GainHigh 0 dB

FreqHigh 7746 Hz

Volume 1

Enable ☒

Audio Parameter Tuner: equalizerV1

GainLow 0 dB

FreqLow 100 Hz

GainMid 0 dB

FreqMid 1224.7 Hz

GainHigh 0 dB

FreqHigh 7746 Hz

Volume 1

Enable ☒

Control Style and Layout

To define the UI grid, add `audioPluginGridLayout` to the `audioPluginInterface`. You can specify the number, size, spacing, and border of cells in the UI grid. In this example, specify `"RowHeight"` as `[20,20,160,20,100]` and `"ColumnWidth"` as `[100,100,100,50,150]`. This creates the following UI grid:



To define the UI control style, update the `audioPluginParameter` definition of each parameter to include the `"Style"` and `"Layout"` name-value pairs. `Style` defines the type of control (rotary knob, slider, or switch, for example). `Layout` defines which cells the controls occupy on the UI grid. You can specify `Layout` as the `[row, column]` of the grid to occupy, or as the `[upper, left; lower, right]` of the group of cells to occupy. By default, control display names are also displayed and occupy their own cells on the UI grid. The cells they occupy depend on the `"DisplayNameLocation"` name-value pair.

The commented arrows indicate the difference between `equalizerV1` and `equalzierV2`.

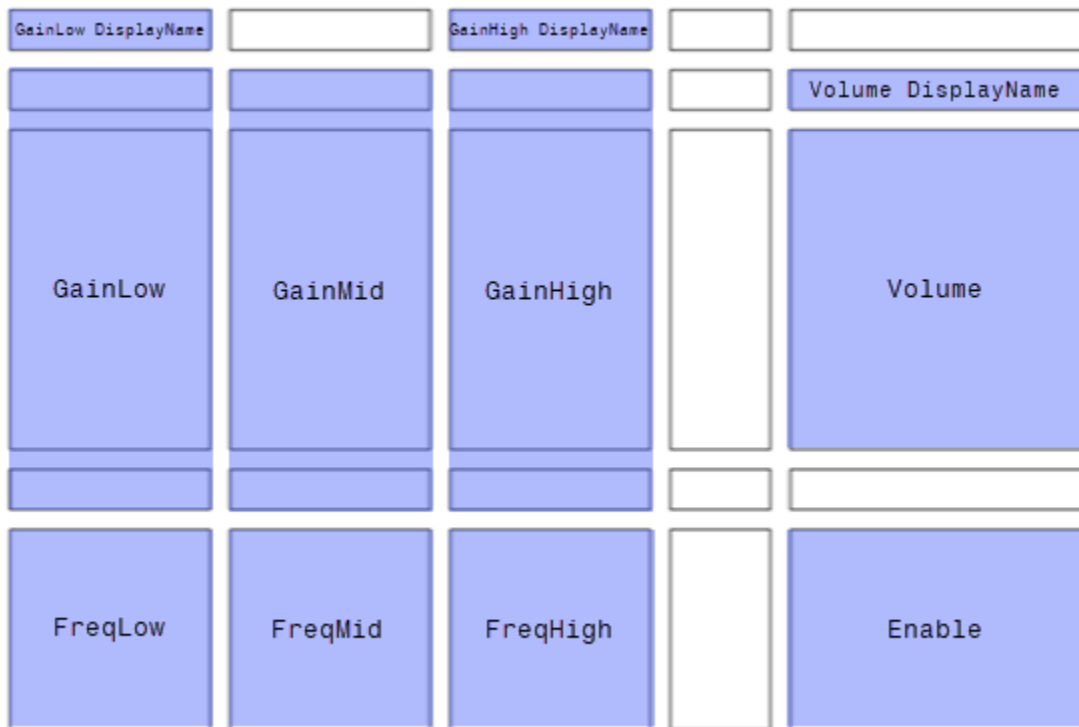
```
classdef equalizerV2 < audioPlugin
    ... % omitted for example purposes
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('GainLow', ...
                'Label','dB', ...
                'Mapping',{'lin',-20,20}, ...
                'Style','vslider', ...           %<--
                'Layout',[2,1;4,1], ...           %<--
                'DisplayName','Low','DisplayNameLocation','Above'), ... %<--
            audioPluginParameter('FreqLow', ...
                'Label','Hz', ...
                'Mapping',{'log',20,500}, ...
```

```

        'Style','rotaryknob', ... %<- -
        'Layout',[5,1], ... %<- -
        'DisplayNameLocation','None'), ... %<- -
    audioPluginParameter('GainMid', ...
        'Label','dB', ...
        'Mapping',{'lin',-20,20}, ...
        'Style','vslider', ... %<- -
        'Layout',[2,2;4,2], ... %<- -
        'DisplayNameLocation','None'), ... %<- -
    audioPluginParameter('FreqMid', ...
        'Label','Hz', ...
        'Mapping',{'log',500,3e3}, ...
        'Style','rotaryknob', ... %<- -
        'Layout',[5,2], ... %<- -
        'DisplayNameLocation','None'), ... %<- -
    audioPluginParameter('GainHigh', ...
        'Label','dB', ...
        'Mapping',{'lin',-20,20}, ...
        'Style','vslider', ... %<- -
        'Layout',[2,3;4,3], ... %<- -
        'DisplayName','High','DisplayNameLocation','Above'), ... %<- -
    audioPluginParameter('FreqHigh', ...
        'Label','Hz', ...
        'Mapping',{'log',3e3,20e3}, ...
        'Style','rotaryknob', ... %<- -
        'Layout',[5,3], ... %<- -
        'DisplayNameLocation','None'), ... %<- -
    audioPluginParameter('Volume', ...
        'Mapping',{'lin',0,2}, ...
        'Style','rotaryknob', ... %<- -
        'Layout',[3,5], ... %<- -
        'DisplayNameLocation','Above'), ... %<- -
    audioPluginParameter('Enable', ...
        'Style','vtoggle', ... %<- -
        'Layout',[5,5], ... %<- -
        'DisplayNameLocation','None'), ... %<- -
    ...
    audioPluginGridLayout( ... %<- -
        'RowHeight',[20,20,160,20,100], ... %<- -
        'ColumnWidth',[100,100,100,50,150]) %<- -
end
... % omitted for example purposes
end

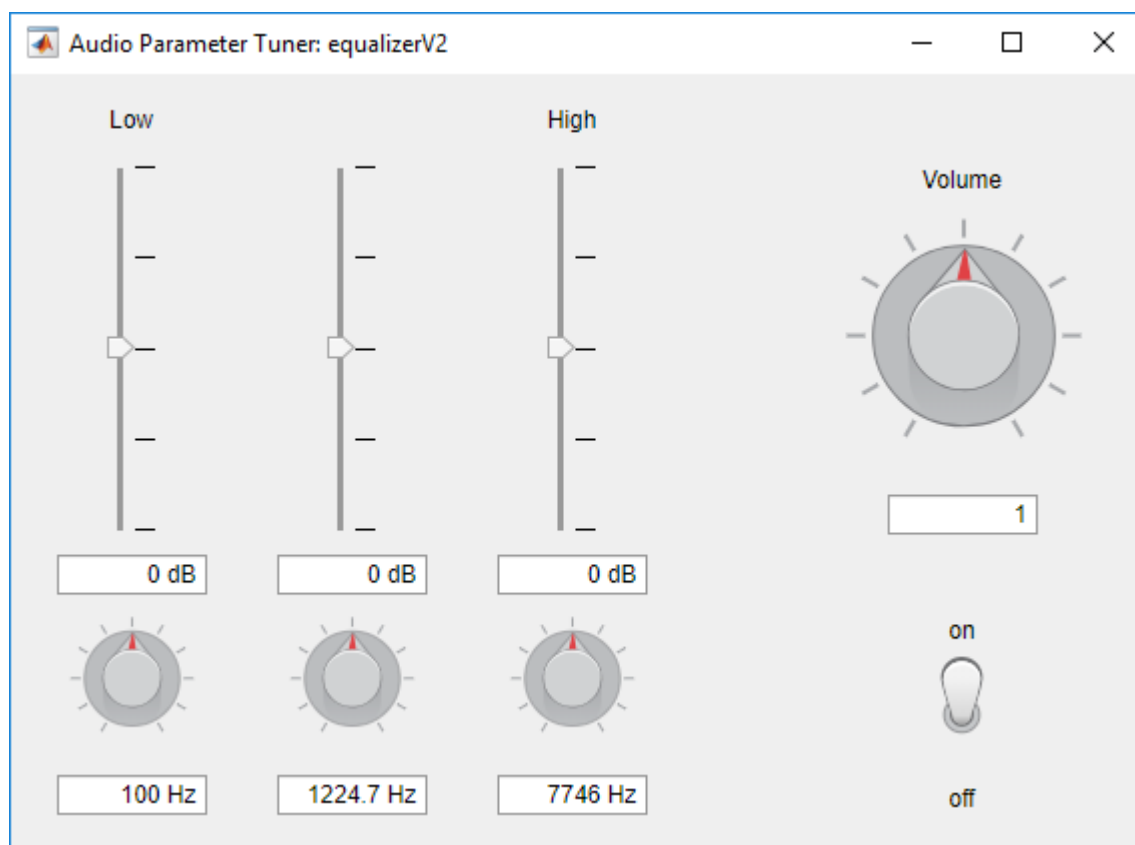
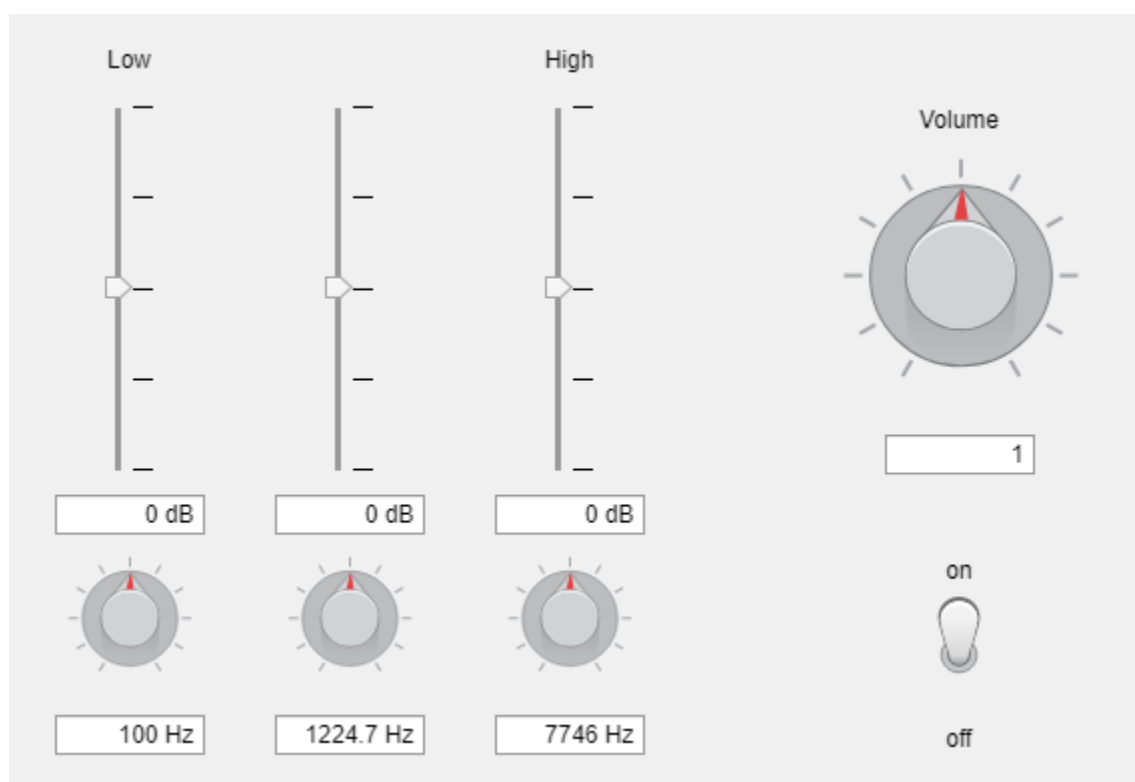
```

The Layout and DisplayNameLocation defined in the audioPluginParameters maps the respective parameters to the control grid as follows:



Call `parameterTuner` to visualize the UI of `equalizerV2`.

```
parameterTuner(equalizerV2)
```



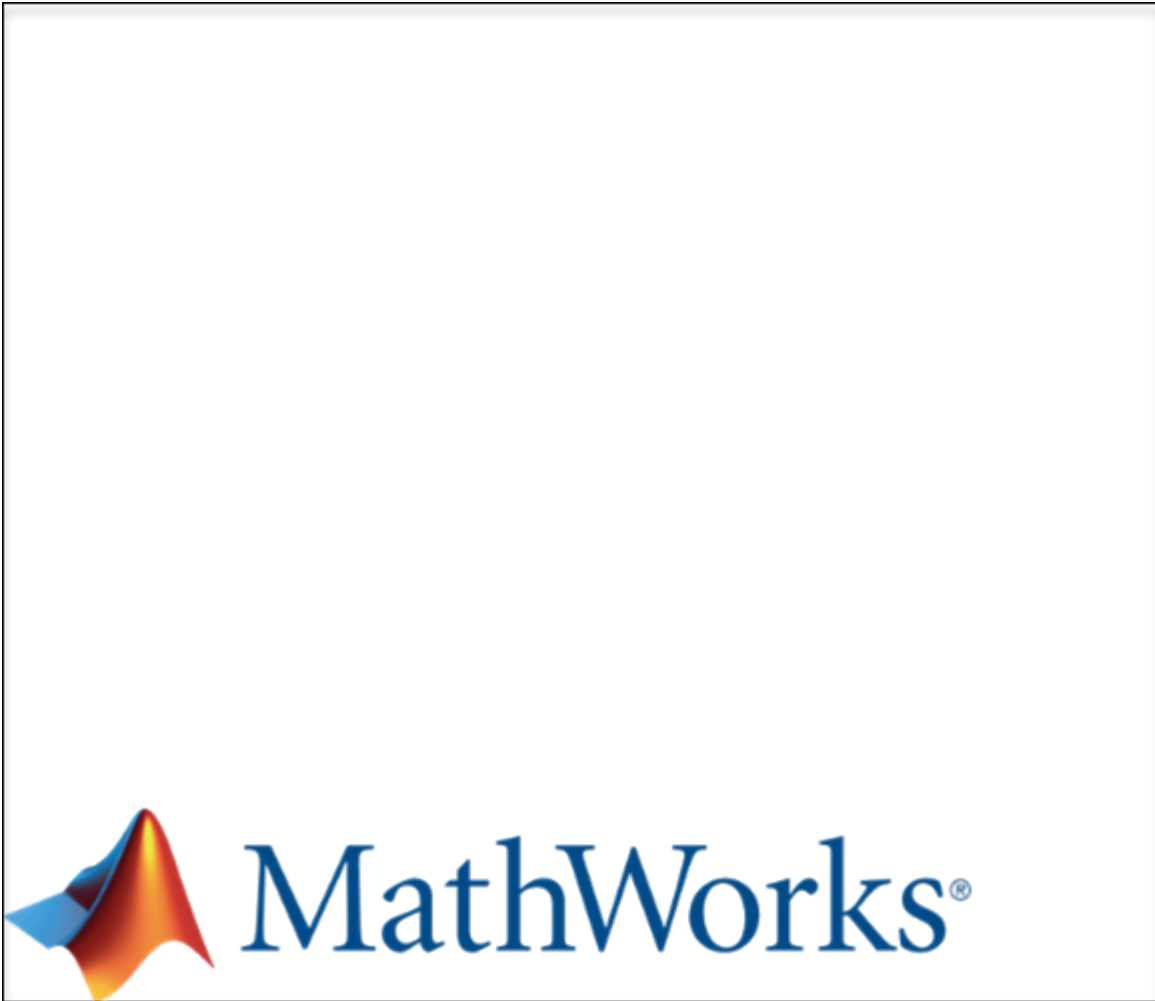
Background Image and Color

To customize the background of your UI, specify `"BackgroundImage"` and `"BackgroundColor"` in `audioPluginInterface`.

The `BackgroundColor` can be specified as a short or long color name string or as an RGB triplet. See `ColorSpec` ([Color Specification](#)) for details. When you specify `BackgroundColor`, the color is applied to all space on the UI except space occupied by controls or a `BackgroundImage`. If the control or background image includes a transparency, then the background color shows through the transparency.

The `BackgroundImage` can be specified as a PNG, GIF, or JPG file. The image is applied to the UI grid by aligning the top left corners of the UI grid and image. If the image is larger than the UI grid size defined in `audioPluginGridLayout`, then the image is clipped to the UI grid size. The background image is not resized. If the image is smaller than the UI grid, then unoccupied regions of the UI grid are treated as transparent.

In this example, you increase the padding around the perimeter of the grid to create space for the MathWorks® logo. You can calculate the total width of the UI grid as the sum of all column widths plus the left and right padding plus the column spacing (the default column spacing of 10 pixels is used in this example): $(100 + 100 + 100 + 50 + 150) + (20 + 20) + (4 \times 10) = 580$. The total height of the UI grid is the sum of all row heights plus the top and bottom padding plus the row spacing (the default row spacing of 10 pixels is used in this example): $(20 + 20 + 160 + 20 + 100) + (20 + 120) + (4 \times 10) = 500$. To locate the logo at the bottom of the UI grid, use a 580-by-500 image:



```

classdef equalizerV3 < audioPlugin
    ... % omitted for example purposes
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('GainLow', ...
                'Label','dB', ...
                'Mapping',{'lin',-20,20}, ...
                'Style','vslider', ...
                'Layout',[2,1;4,1], ...
                'DisplayName','Low','DisplayNameLocation','Above'), ...
            audioPluginParameter('FreqLow', ...
                'Label','Hz', ...
                'Mapping',{'log',20,500}, ...
                'Style','rotaryknob', ...
                'Layout',[5,1], ...
                'DisplayNameLocation','None'), ...
            audioPluginParameter('GainMid', ...
                'Label','dB', ...
                'Mapping',{'lin',-20,20}, ...
                'Style','vslider', ...
                'Layout',[2,2;4,2], ...
                'DisplayNameLocation','None'), ...

```

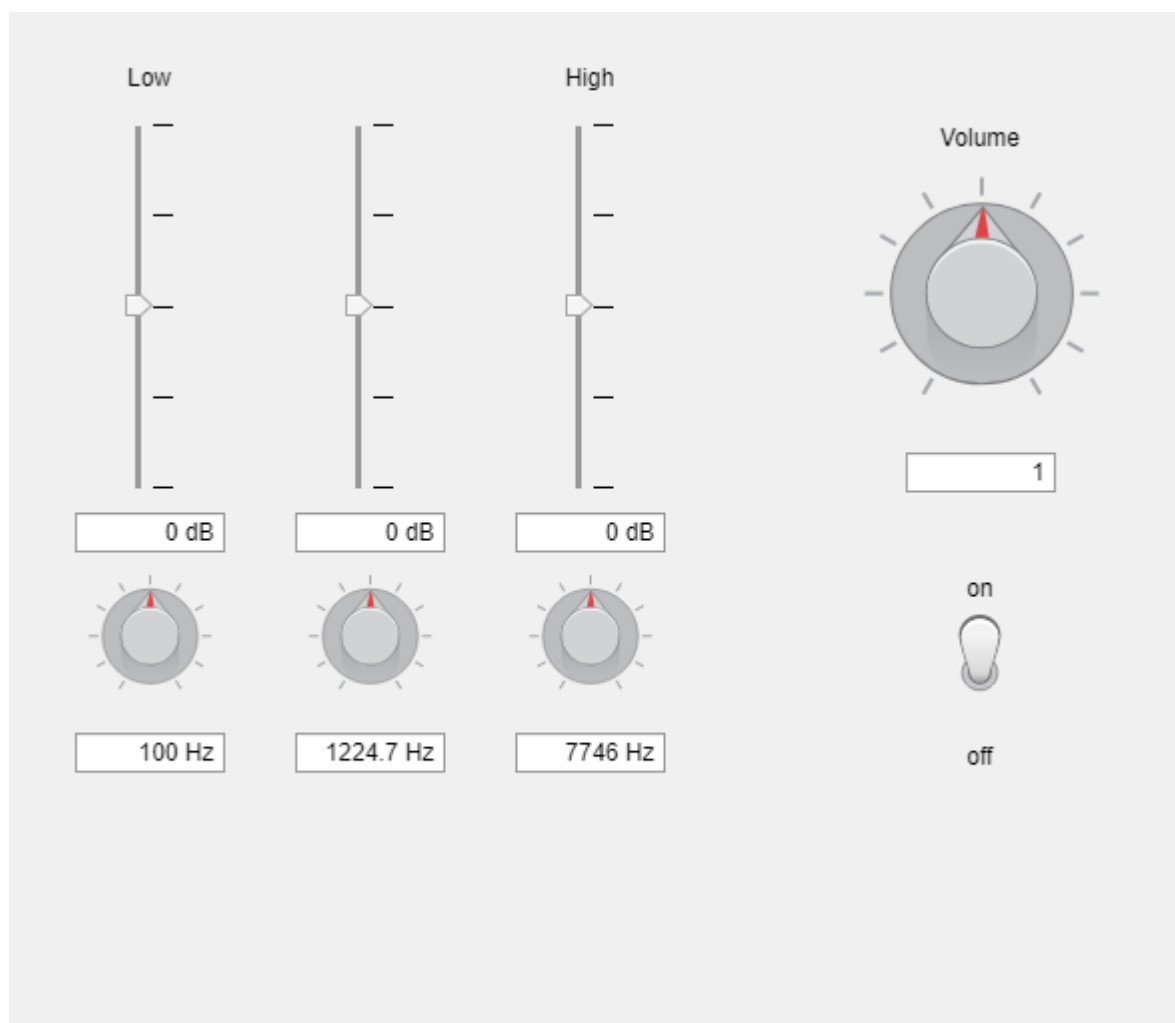
```

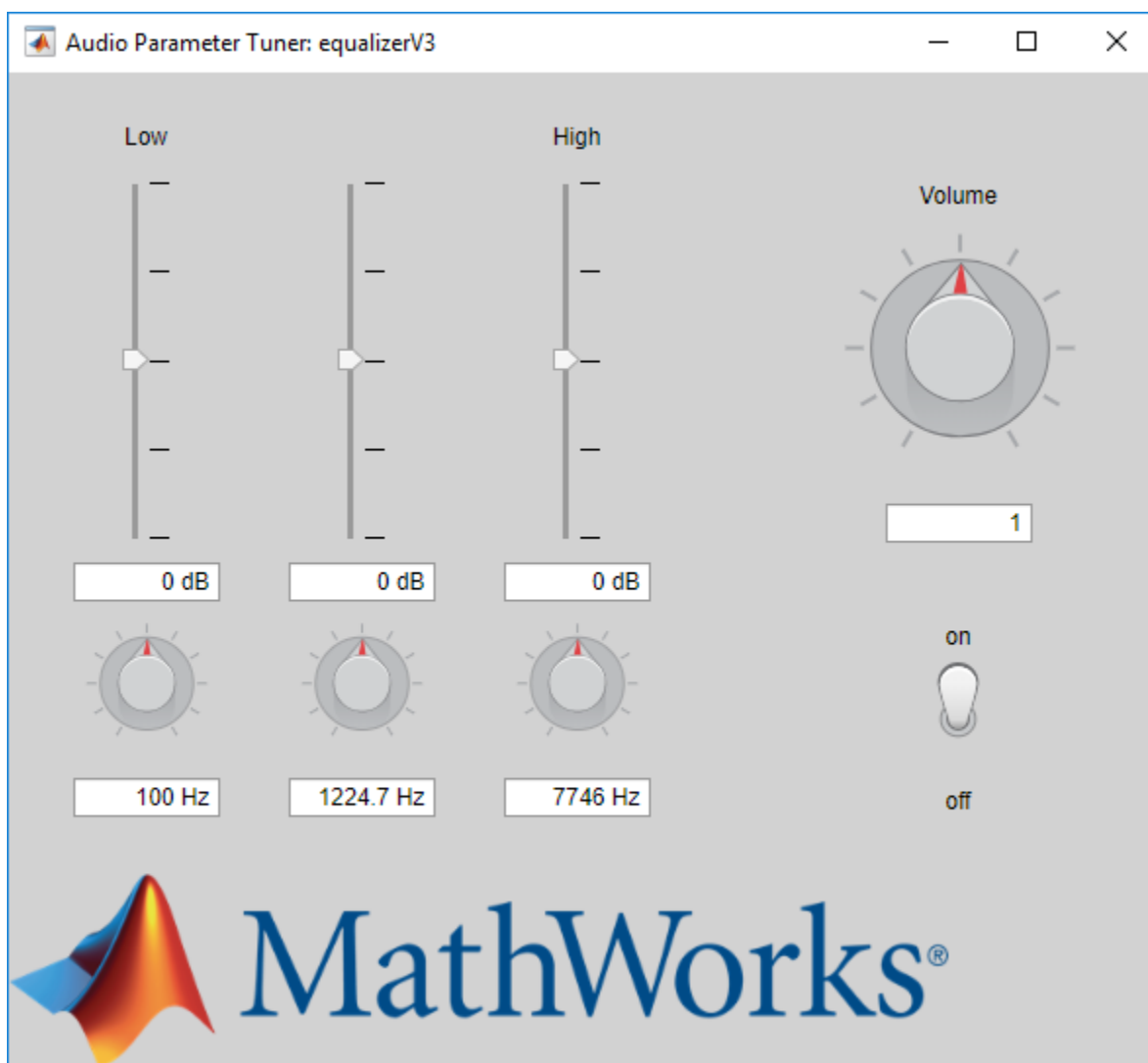
audioPluginParameter('FreqMid', ...
    'Label','Hz', ...
    'Mapping',{ 'log',500,3e3}, ...
    'Style','rotaryknob', ...
    'Layout',[5,2], ...
    'DisplayNameLocation','None'), ...
audioPluginParameter('GainHigh', ...
    'Label','dB', ...
    'Mapping',{ 'lin',-20,20}, ...
    'Style','vslider', ...
    'Layout',[2,3;4,3], ...
    'DisplayName','High','DisplayNameLocation','Above'), ...
audioPluginParameter('FreqHigh', ...
    'Label','Hz', ...
    'Mapping',{ 'log',3e3,20e3}, ...
    'Style','rotaryknob', ...
    'Layout',[5,3], ...
    'DisplayNameLocation','None'), ...
audioPluginParameter('Volume', ...
    'DisplayName','Volume', ...
    'Mapping',{ 'lin',0,2}, ...
    'Style','rotaryknob', ...
    'Layout',[3,5], ...
    'DisplayNameLocation','Above'), ...
audioPluginParameter('Enable', ...
    'Style','vtoggle', ...
    'Layout',[5,5], ...
    'DisplayNameLocation','None'), ...
...
audioPluginGridLayout( ...
    'RowHeight',[20,20,160,20,100], ...
    'ColumnWidth',[100,100,100,50,150], ...
    'Padding',[20,120,20,20]), ...
...
    'BackgroundImage','background.png', ...
    'BackgroundColor',[210/255,210/255,210/255])
end
... % omitted for example purposes
end

```

Call parameterTuner to visualize the UI of equalizerV3.

```
parameterTuner(equalizerV3)
```





Custom Control Filmstrips

To use custom filmstrips, specify the `"Filmstrip"` and `"FilmstripFrameSize"` name-value pairs in `audioPluginParameter`. The filmstrip can be a PNG, GIF, or JPG file, and should consist of frames placed end-to-end either vertically or horizontally. The filmstrip is mapped to the control's range so that the corresponding filmstrip frame is displayed on the plugin UI as you tune parameters. In this example, specify a two-frame filmstrip for the `Enable` parameter. As a best practice, the size of each frame of the film strip should equal the size of the region occupied by the parameter. The `Enable` parameter occupies one cell that is 150-by-100 pixels. To create a vertical filmstrip where each frame is 150-by-100, make the total filmstrip size 150-by-200 and set `FilmstripFrameSize` to `[150, 100]`. The filmstrip used in this example contains the frame corresponding to the off position first, then the on position:



```

classdef equalizerV4 < audioPlugin
    ... % omitted for example purposes
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('GainLow', ...
                'Label','dB', ...
                'Mapping',{'lin',-20,20}, ...
                'Style','vslider', ...
                'Layout',[2,1;4,1], ...
                'DisplayName','Low','DisplayNameLocation','Above'), ...
            audioPluginParameter('FreqLow', ...
                'Label','Hz', ...
                'Mapping',{'log',20,500}, ...
                'Style','rotaryknob', ...
                'Layout',[5,1], ...
                'DisplayNameLocation','None'), ...
            audioPluginParameter('GainMid', ...
                'Label','dB', ...
                'Mapping',{'lin',-20,20}, ...
                'Style','vslider', ...
                'Layout',[2,2;4,2], ...
                'DisplayNameLocation','None'), ...
            audioPluginParameter('FreqMid', ...
                'Label','Hz', ...
                'Mapping',{'log',500,3e3}, ...
                'Style','rotaryknob', ...
                'Layout',[5,2], ...
                'DisplayNameLocation','None'), ...
            audioPluginParameter('GainHigh', ...
                'Label','dB', ...
                'Mapping',{'lin',-20,20}, ...
                'Style','vslider', ...
                'Layout',[2,3;4,3], ...
                'DisplayName','High','DisplayNameLocation','Above'), ...
            audioPluginParameter('FreqHigh', ...
                'Label','Hz', ...
                'Mapping',{'log',3e3,20e3}, ...
                'Style','rotaryknob', ...
                'Layout',[5,3], ...
                'DisplayNameLocation','None'), ...
            audioPluginParameter('Volume', ...
                'Mapping',{'lin',0,2}, ...

```

```

        'Style','rotaryknob', ...
        'Layout',[3,5], ...
        'DisplayNameLocation','Above'), ...
audioPluginParameter('Enable', ...
    'Style','vtoggle', ...
    'Layout',[5,5], ...
    'DisplayNameLocation','None', ...
    'Filmstrip','vtoggle.png', ... %< - -
    'FilmstripFrameSize',[150,100]), ... %< - -
    ...
audioPluginGridLayout( ...
    'RowHeight',[20,20,160,20,100], ...
    'ColumnWidth',[100,100,100,50,150], ...
    'Padding',[20,120,20,20]), ...
    ...
    'BackgroundImage','background.png', ...
    'BackgroundColor',[210/255,210/255,210/255])
end
... % omitted for example purposes
end

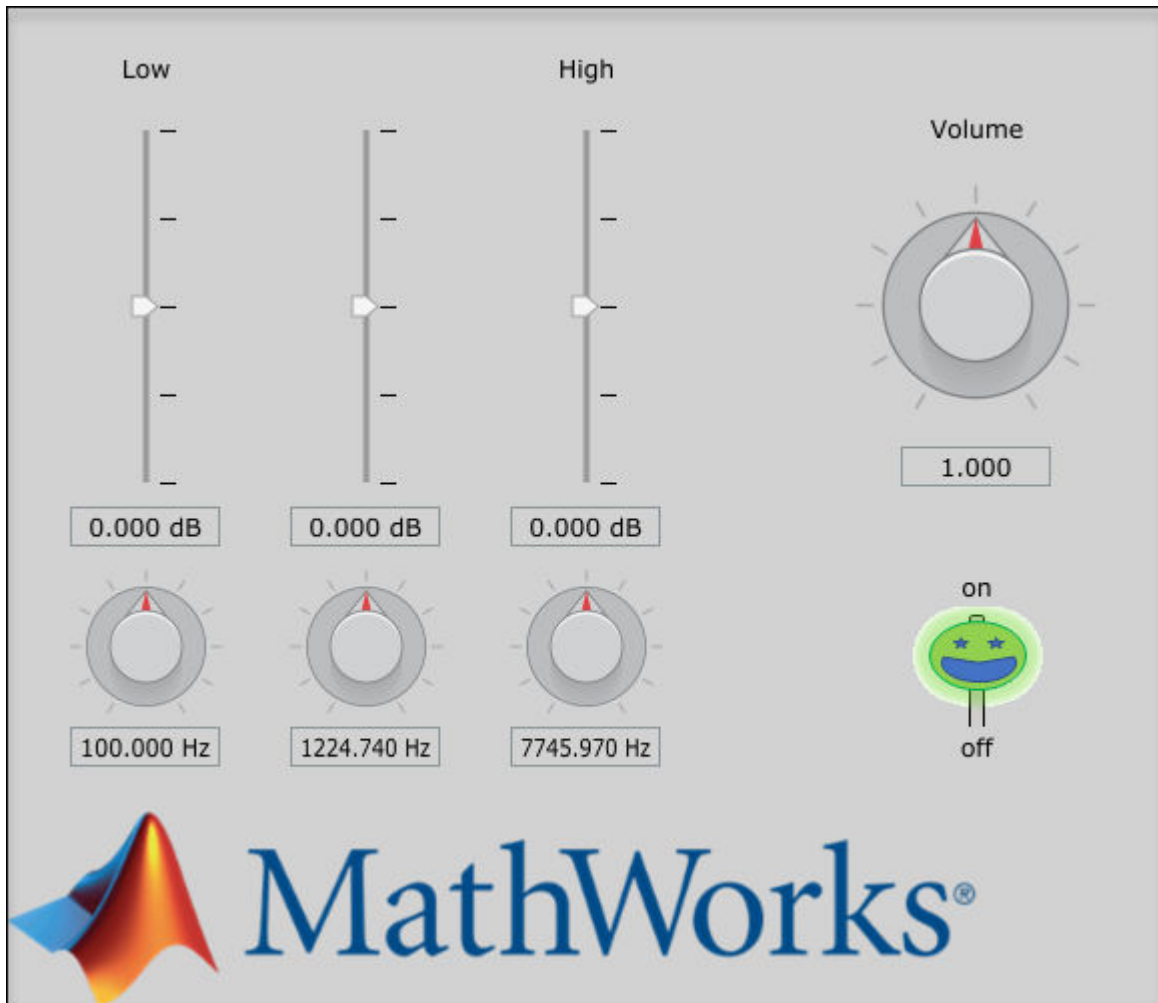
```

Filmstrips are not supported by `parameterTuner`. To see the custom plugin UI, you must deploy the plugin to a DAW. Use `generateAudioPlugin` to create a VST plugin.

```
generateAudioPlugin equalizerV4
```

```
.....
```

In this example, the plugin was opened in REAPER. A screenshot of the UI in REAPER is displayed below.



See Also

More About

- “Audio Plugins in MATLAB”
- “Export a MATLAB Plugin to a DAW”

See Also

[audioPlugin](#) | [audioPluginGridLayout](#) | [audioPluginInterface](#) | [audioPluginParameter](#) | [generateAudioPlugin](#) | [parameterTuner](#)

Use the Audio Labeler

Label Audio Using Audio Labeler

The **Audio Labeler** app enables you to interactively define and visualize ground-truth labels for audio data sets. This example shows how you can create label definitions and then interactively label a set of audio files. The example also shows how to export the labeled ground-truth data, which you can then use with `audioDatastore` to train a machine learning system.

Load Unlabeled Data

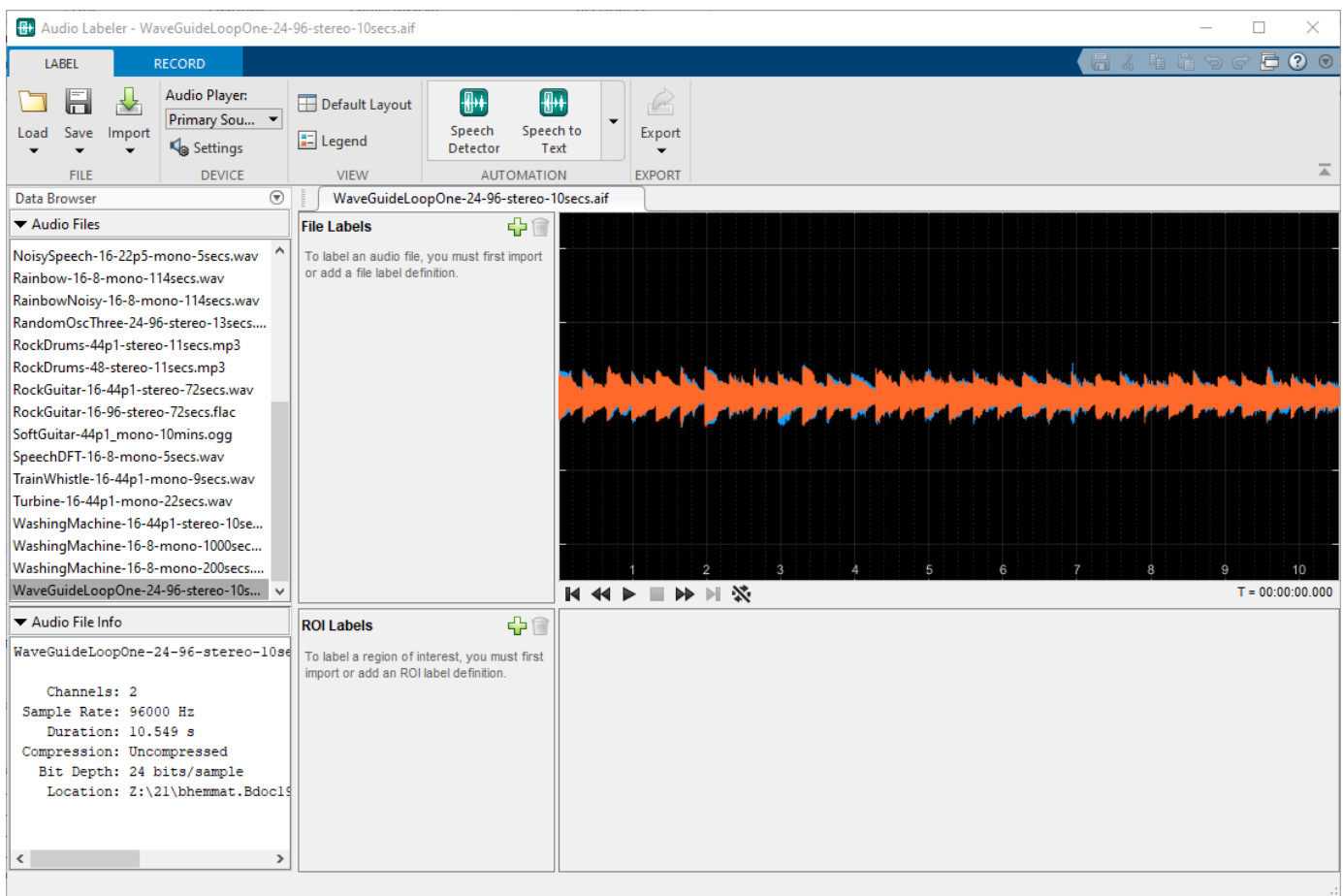
- 1 To open the **Audio Labeler**, at the MATLAB® command prompt, enter:

```
audioLabeler
```

- 2 This example uses the audio files included with Audio Toolbox. To locate the file path on your system, at the MATLAB command prompt, enter:

```
fullfile(matlabroot, 'toolbox', 'audio', 'samples')
```

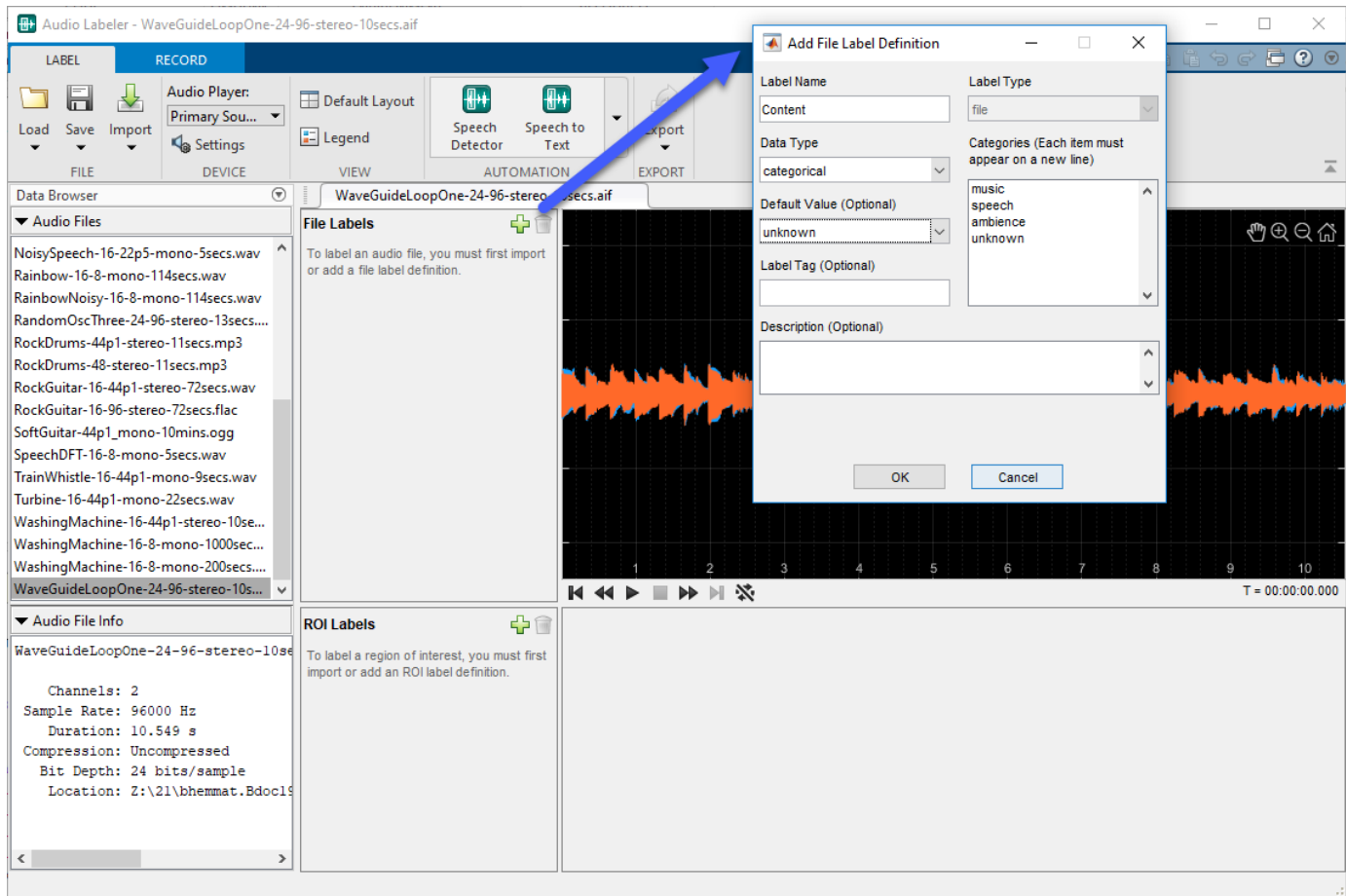
To load audio from a file, click **Load > Audio Folders** and select the folder containing audio files you want to label.



Define and Assign Labels

File-Level Labels

The audio samples include music, speech, and ambience. To create a file-level label that defines the contents of the audio file as music, speech, ambience, or unknown, click **+**. Specify the **Label Name** as Content, the **Data Type** as categorical, and the **Categories** as music, speech, ambience, or unknown. Set the **Default Value** of the label definition to unknown.



All audio files in the **Data Browser** are now associated with the Content label name. To listen to the audio file selected in the **Data Browser** and confirm that it is a music file, click **▶**. To set the value of the Contents label, click unknown in the **File Labels** panel and select music from the drop-down menu.

The selected audio file now has the label name Content with value music assigned to it. You can continue setting the Content value for each file by selecting a file in the **Data Browser** and then selecting a value from the **File Labels** panel.

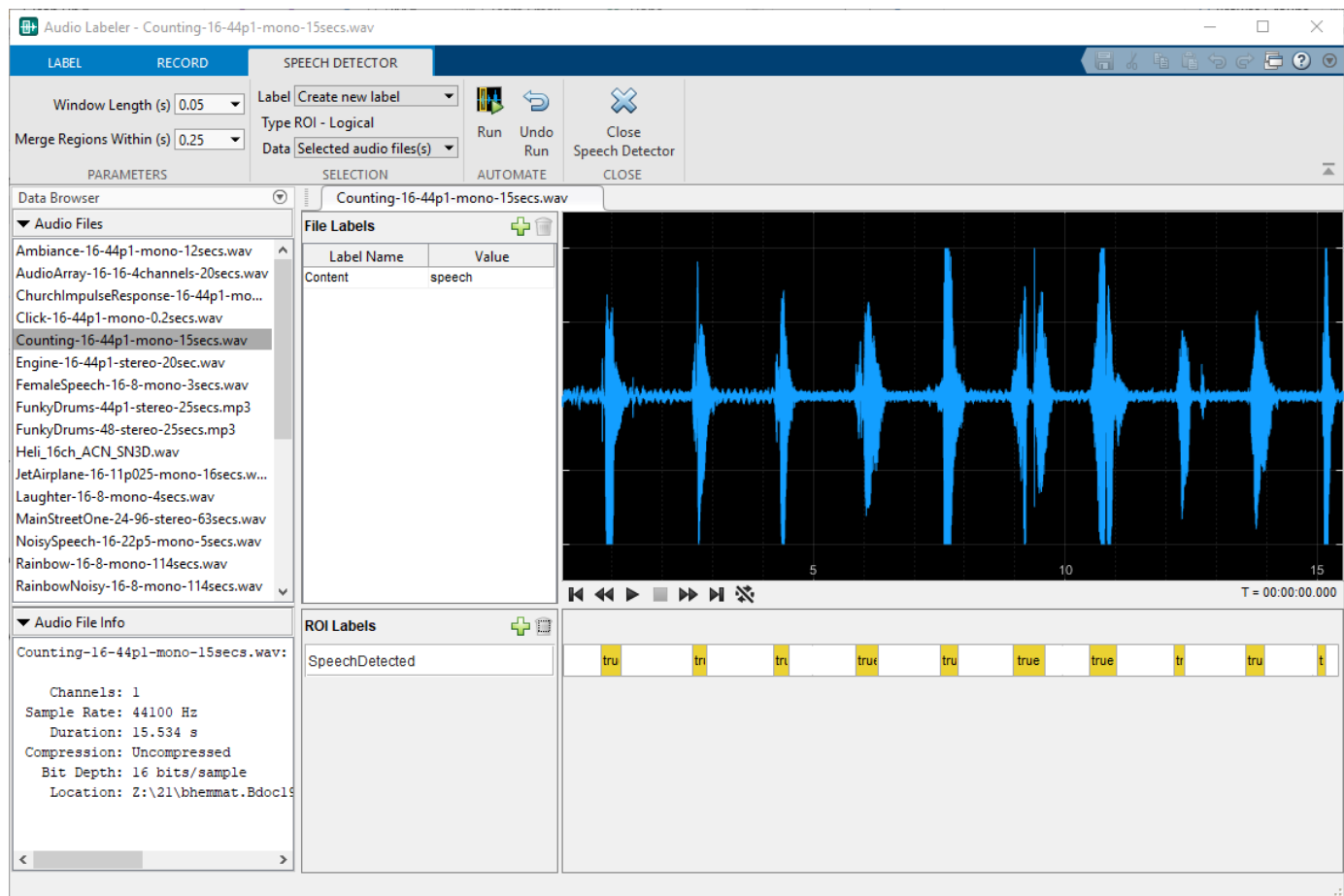
Region-Level Labels


You can define region-level labels manually or by using the provided automated algorithms. Audio Toolbox includes automatic labeling algorithms for speech detection and speech-to-text transcription.

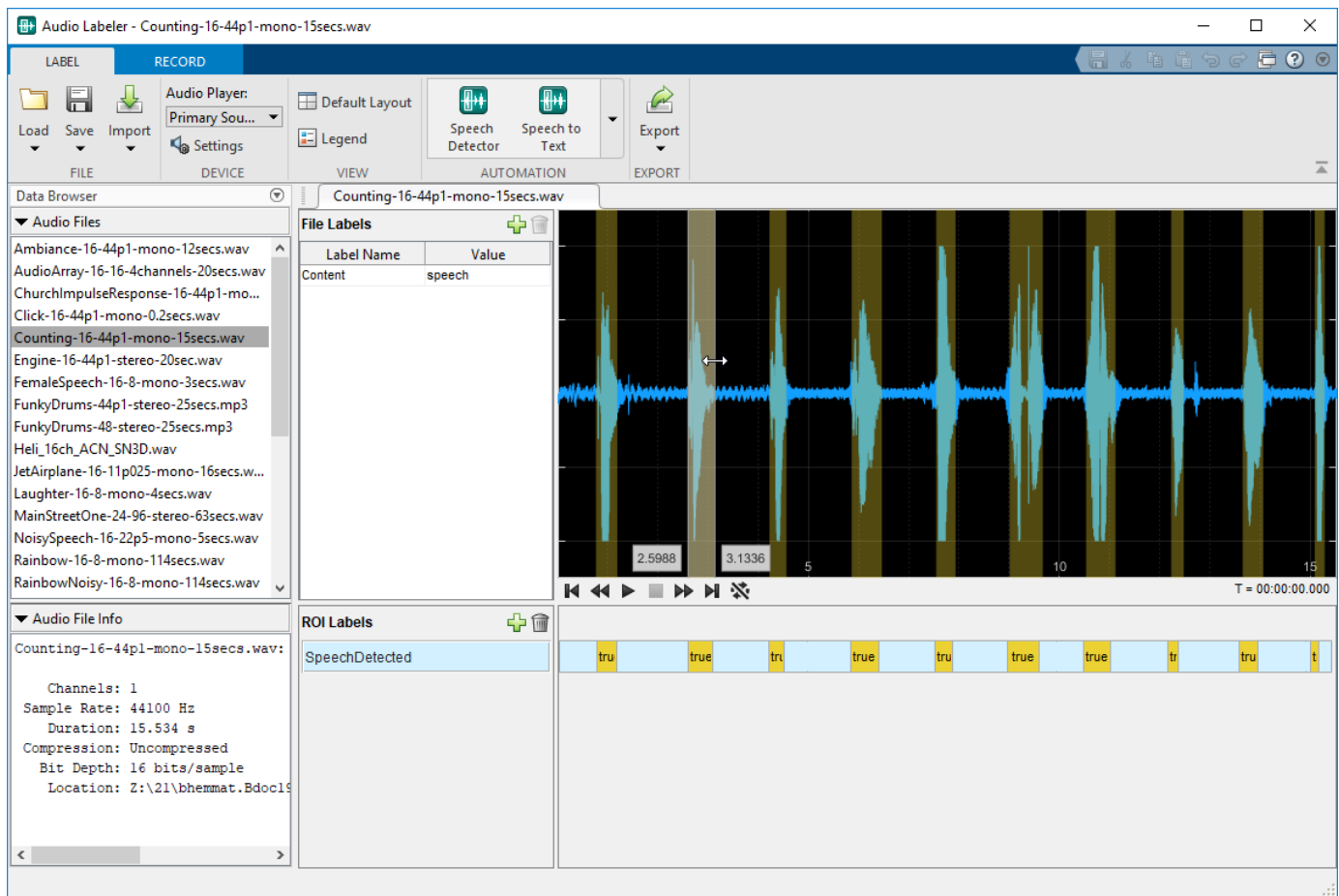
Note To enable automatic speech-to-text transcription, you must download and set up the “Speech-to-Text Transcription” on page 4-2 functionality. Once you download and set up the speech-to-text transcription functionality, the **Speech to Text** automation algorithm appears as an option on the toolstrip.

Select `Counting-16-44p1-mono-15secs.wav` from the **Data Browser**.

To create a region-level label that indicates if speech is detected, first select **Speech Detector** from the **AUTOMATION** section. You can control the speech detection algorithm using the **Window Length (s)** and **Merge Regions Within (s)** parameters. Use the default parameters for the speech detection algorithm. To create an ROI label and to label regions of the selected audio file, select **Run**.

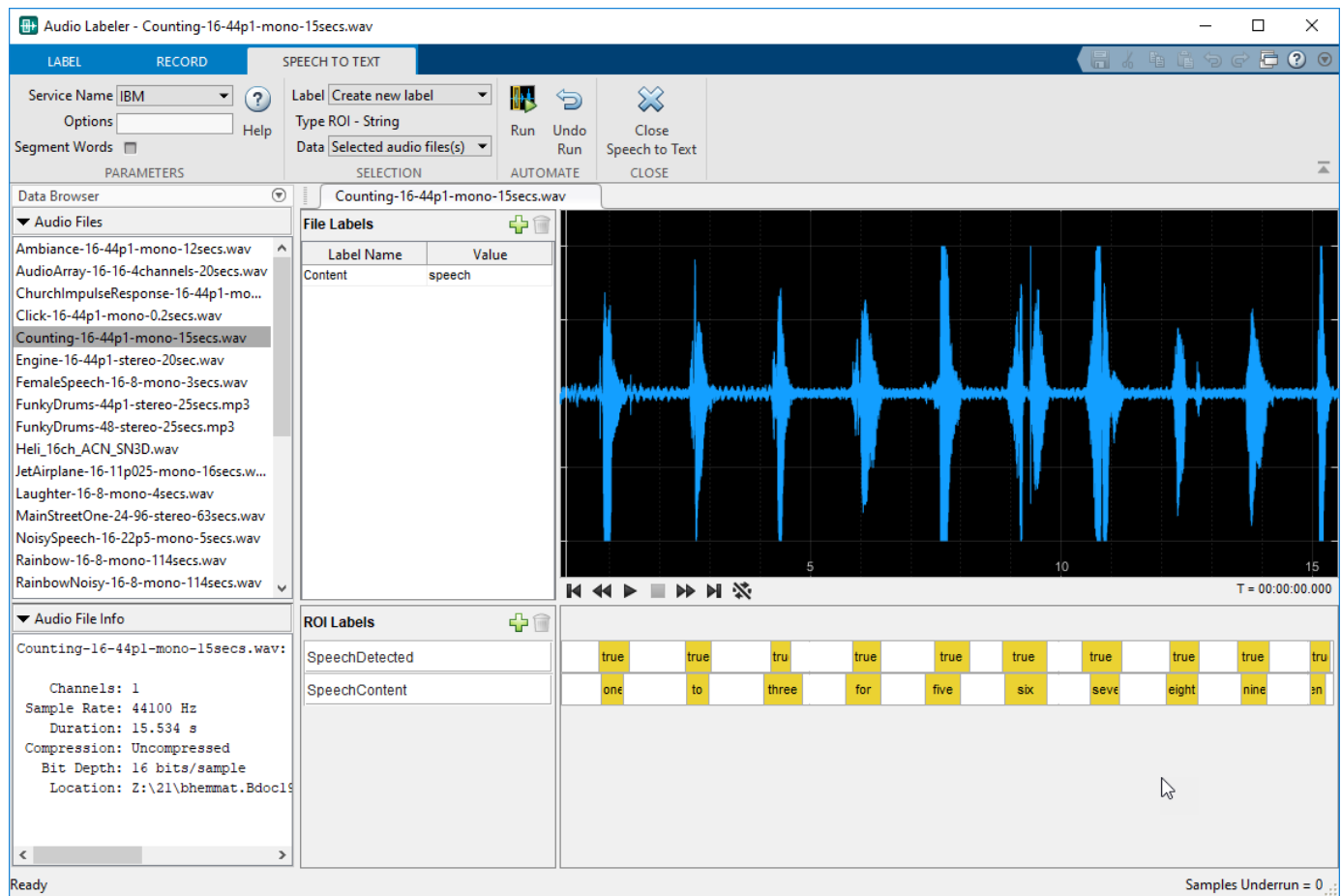


Close the **Speech Detector** tab. You can correct or fine-tune the automatically generated **SpeechDetected** regions by selecting the ROI from the ROI bar, and then dragging the edges of the region. The ROI bar is directly to the right of the ROI label. When a region is selected, clicking  plays only the selected region, enabling you to verify whether the selected region captures all relevant auditory information.




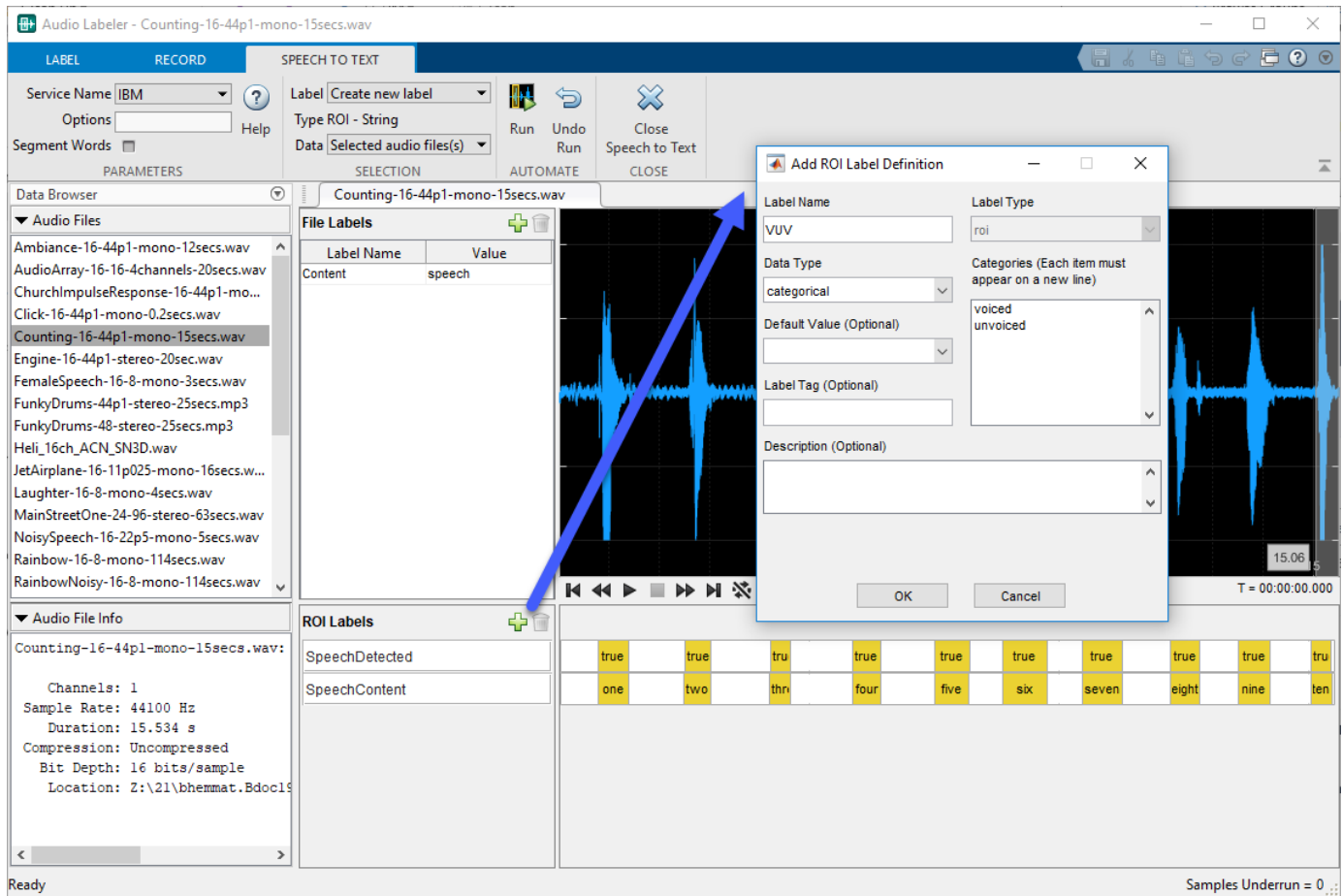
If you have set up a speech-to-text transcription service, select **Speech to Text** from the **Automation** section. You can control the speech-to-text transcription using name-value pair options specific to your selected service. This example uses the IBM® service and specifies no additional options.

3 Use the Audio Labeler



The ROI labels returned from the transcription service are strings with beginning and end points. The beginning and end points do not exactly correspond to the beginning and end points of the manually corrected speech detection regions. You can correct the endpoints of the **SpeechContent** ROI label by selecting the region and then dragging the edges of the region. The transcription service misclassified the words "two" as "to," "four" as "for," and "ten" as "then." You can correct the string by selecting the region and then entering a new string.

Create another region-level label by clicking  in the **ROI Labels** panel. Set **Label Name** to VUV, set **Data Type** to categorical, and **Categories** to voiced and unvoiced.



By default, the waveform viewer shows the entire file. To display tools for zooming and panning, hover over the top right corner of the plot. Zoom in on the first five seconds of the audio file.

When you select a region in the plot and then hover over any of the two ROI bars, the shadow of the region appears. To assign the selected region to the category **voiced**, click **one** on the **SpeechContent** label bar. Hover over the **VUV** label bar and then click the shadow and choose **voiced**.

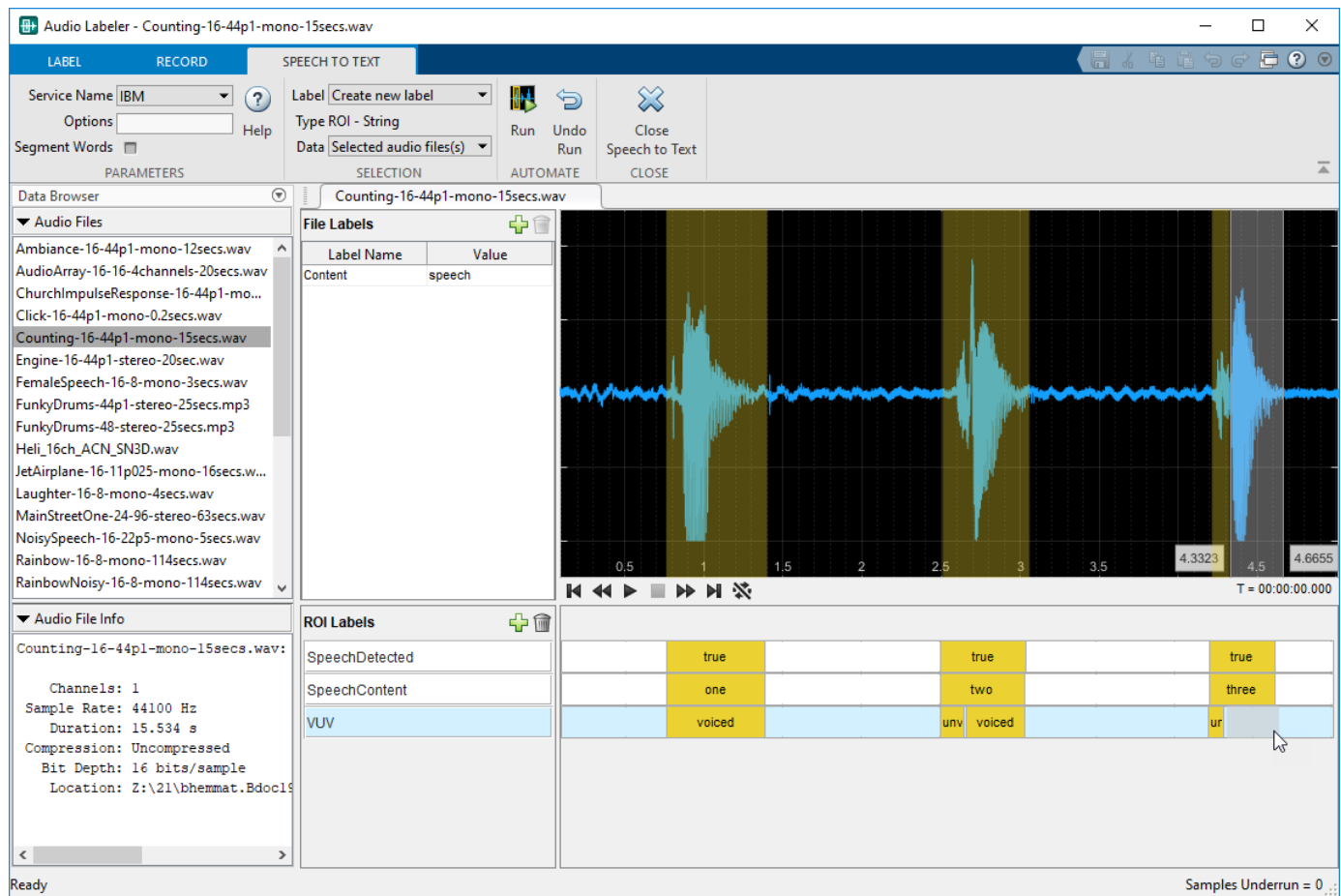
3 Use the Audio Labeler

The screenshot shows the Audio Labeler software interface for the file 'Counting-16-44p1-mono-15secs.wav'. The 'SPEECH TO TEXT' tab is active. The 'Data Browser' on the left lists various audio files, with 'Counting-16-44p1-mono-15secs.wav' selected. The 'Audio File Info' section shows details for the selected file: Channels: 1, Sample Rate: 44100 Hz, Duration: 15.534 s, Compression: Uncompressed, Bit Depth: 16 bits/sample, and Location: Z:\21\bhemmat.Bdoc18. The waveform plot on the right shows the audio signal with time markers at 0.76075, 1.3992, 2, 2.5, 3, 3.5, 4, and 4.5. The 'File Labels' section shows a table with 'Label Name' and 'Value' columns, with 'Content' labeled as 'speech'. The 'ROI Labels' section shows a table with 'SpeechDetected' (true), 'SpeechContent' (one, two, three), and 'VUV' (voiced, unvoiced) labels. The 'VUV' label bar is currently set to 'voiced'.

Label Name	Value
Content	speech

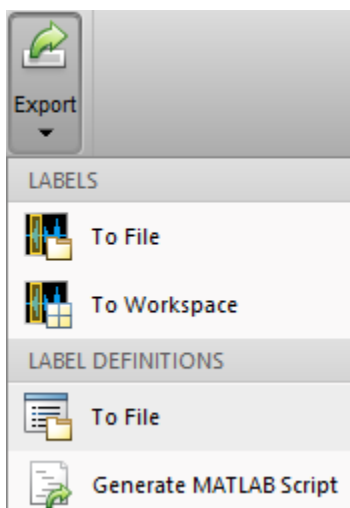
SpeechDetected	SpeechContent	VUV
true	one	voiced
true	two	voiced
true	three	voiced

The next two words, "two" and "three," contain both voiced and unvoiced speech. Select each region of speech on the plot, hover over the VUV label bar, and select the correct category for that region.



Export Label Definitions

You can export label definitions as a MAT file or as a MATLAB script. Maintaining label definitions enables consistent labeling between users and sessions. Select **Export > Label Definitions > To File**.



The labels are saved as an array of `signalLabelDefinition` objects. In your next session, you can import the label definitions by selecting **Import > Label Definitions > From File**.

Export Labeled Audio Data

You can export the labeled signal set to a file or to your workspace. Select **Export > Labels > To Workspace**.

The **Audio Labeler** creates a `labeledSignalSet` object named `labeledSet_HHMMSS`, where *HHMMSS* is the time the object is created in hours, minutes, and seconds.

```
labeledSet_104620
labeledSet_104620 =
```

```
labeledSignalSet with properties:
```

```
    Source: {29x1 cell}
  NumMembers: 29
TimeInformation: "inherent"
    Labels: [29x4 table]
  Description: ""
```

```
Use labelDefinitionsHierarchy to see a list of labels and sublabels.
Use setLabelValue to add data to the set.
```

The labels you created are saved as a table to the `Labels` property.

```
labeledSet_142356.Labels
```

```
ans =
```

```
29x4 table
```

```
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\AudioArray-16-16-4channels-20secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Click-16-44p1-mono-0.2secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Counting-16-44p1-mono-15secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Engine-16-44p1-stereo-20sec.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\FemaleSpeech-16-8-mono-3secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\FunkyDrums-44p1-stereo-25secs.mp3
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\FunkyDrums-48-stereo-25secs.mp3
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Heli_16ch_ACN_SN3D.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\JetAirplane-16-11p025-mono-16secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Laughter-16-8-mono-4secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples>MainStreetOne-24-96-stereo-63secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\NoisySpeech-16-22p5-mono-5secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Rainbow-16-8-mono-114secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\RainbowNoisy-16-8-mono-114secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\RandomOscThree-24-96-stereo-13secs.aif
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\RockDrums-44p1-stereo-11secs.mp3
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\RockDrums-48-stereo-11secs.mp3
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\RockGuitar-16-44p1-stereo-72secs.wav
```

```
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\RockGuitar-16-96-stereo-72secs.flac
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\SoftGuitar-44p1-mono-10mins.ogg
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\SpeechDFT-16-8-mono-5secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\TrainWhistle-16-44p1-mono-9secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Turbine-16-44p1-mono-22secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\WashingMachine-16-44p1-stereo-10secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\WashingMachine-16-8-mono-1000secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\WashingMachine-16-8-mono-200secs.wav
C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\WaveGuideLoopOne-24-96-stereo-10secs.ai
```

The file names associated with the labels are saved as a cell array to the Source property.

```
labeledSet_104620.Source
```

```
ans =
```

```
29x1 cell array
```

```
{'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\AudioArray-16-16-4channels-20secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Click-16-44p1-mono-0.2secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Counting-16-44p1-mono-15secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Engine-16-44p1-stereo-20sec.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\FemaleSpeech-16-8-mono-3secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\FunkyDrums-44p1-stereo-25secs.mp3'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\FunkyDrums-48-stereo-25secs.mp3'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Heli_16ch_ACN_SN3D.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\JetAirplane-16-11p025-mono-16secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Laughter-16-8-mono-4secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples>MainStreetOne-24-96-stereo-63secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\NoisySpeech-16-22p5-mono-5secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Rainbow-16-8-mono-114secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\RainbowNoisy-16-8-mono-114secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\RandomOscThree-24-96-stereo-13secs.ai'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\RockDrums-44p1-stereo-11secs.mp3'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\RockDrums-48-stereo-11secs.mp3'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\RockGuitar-16-44p1-stereo-72secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\RockGuitar-16-96-stereo-72secs.flac'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\SoftGuitar-44p1-mono-10mins.ogg'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\SpeechDFT-16-8-mono-5secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\TrainWhistle-16-44p1-mono-9secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\Turbine-16-44p1-mono-22secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\WashingMachine-16-44p1-stereo-10secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\WashingMachine-16-8-mono-1000secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\WashingMachine-16-8-mono-200secs.wav'
'C:\Program Files\MATLAB\R2019b\toolbox\audio\samples\WaveGuideLoopOne-24-96-stereo-10secs.ai'}
```

Prepare Audio Datastore for Deep Learning Workflow

To continue on to a deep learning or machine learning workflow, use `audioDatastore`. Using an audio datastore enables you to apply capabilities that are common to machine learning applications, such as `splitEachLabel`. `splitEachLabel` enables you split your data into train and test sets.

Create an audio datastore for your labeled signal set. Specify the location of the audio files as the first argument of `audioDatastore` and set the `Labels` property of `audioDatastore` to the `Labels` property of the labeled signal set.

```
ADS = audioDatastore(labeledSet_104620.Source, 'Labels', labeledSet_104620.Labels)
```

```
ADS =
```

```
audioDatastore with properties:
```

```
Files: {  
    '...\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav';  
    '...\toolbox\audio\samples\AudioArray-16-16-4channels-20secs.wav';  
    '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5secs.wav';  
    ... and 26 more  
}  
Labels: 29-by-4 table  
AlternateFileSystemRoots: {}  
OutputDataType: 'double'
```

Call `countEachLabel` and specify the `Content` table variable to count the number of files that are labeled as `ambience`, `music`, `speech`, or `unknown`.

```
countEachLabel(ADS, 'TableVariable', 'Content')
```

```
ans =
```

```
4×2 table
```

Content	Count
ambience	13
music	9
speech	6
unknown	1

For examples of using labeled audio data in a machine learning or deep learning workflow, see:

- “Speech Command Recognition Using Deep Learning” on page 1-331
- “Speaker Identification Using Pitch and MFCC” on page 1-224
- “Denoise Speech Using Deep Learning Networks” on page 1-297
- “Classify Gender Using LSTM Networks” on page 1-317
- “Music Genre Classification Using Wavelet Time Scattering” on page 1-359

See Also

`audioDatastore` | `audioDeviceReader` | `audioDeviceWriter` | `labeledSignalSet` | `signalLabelDefinition`

Speech2Text and Text2Speech Chapter

- “Speech-to-Text Transcription” on page 4-2
- “Text-to-Speech Conversion” on page 4-3

Speech-to-Text Transcription

Audio Toolbox enables you to interface with third-party speech-to-text APIs from MATLAB.

```
ans =  
  
"hello world"  
  
>>
```



To interface with third-party speech-to-text APIs, you must have the following:

- Audio Toolbox release R2017a or above
- Audio Toolbox extended functionality available from File Exchange
- One of the following APIs:
 - Google® Speech API
 - IBM Watson Speech API
 - Microsoft® Azure Speech API

The third-party APIs require you to generate keys for identification purposes. To begin, download the extended Audio Toolbox functionality from File Exchange. The File Exchange submission includes a tutorial to get you started. Once you have installed the speech-to-text functionality and set up your API keys, you can perform speech-to-text transcription programmatically or using the **Audio Labeler** app.

See Also

Audio Labeler

Related Examples

- “Text-to-Speech Conversion” on page 4-3

External Websites

- [speech2text](#) on File Exchange
- [text2speech](#) on File Exchange

Text-to-Speech Conversion

Audio Toolbox enables you to interface with third-party text-to-speech (TTS) APIs from MATLAB.



To interface with third-party text-to-speech APIs and synthesize speech, you must have the following:

- Audio Toolbox release R2019a or above
- Audio Toolbox extended functionality available from File Exchange
- One of the following APIs:
 - Google Speech API
 - IBM Watson Speech API
 - Microsoft Azure Speech API

The third-party APIs require you to generate keys for identification purposes. To begin, download the extended Audio Toolbox functionality from File Exchange. The File Exchange submission includes a tutorial to get you started. Once you have installed the text-to-speech functionality and set up your API keys, you can perform text-to-speech conversion programmatically.

See Also

Audio Labeler

Related Examples

- “Speech-to-Text Transcription” on page 4-2

External Websites

- [speech2text](#) on File Exchange
- [text2speech](#) on File Exchange

Measure Impulse Response of an Audio System

Impulse Response Measurer Walkthrough

In this tutorial, explore key functionality of the **Impulse Response Measurer**. The **Impulse Response Measurer** app enables you to

- Configure your audio I/O system.
- Acquire impulse response (IR) measurements using either the Exponential Swept Sine or Maximum Length Sequences methods.
- View and manage captured IR data.
- Export the data to a file, workspace, or other app for further study.

To begin, open the **Impulse Response Measurer** app by selecting the  icon from the app gallery.

Configure Audio I/O System

The **Impulse Response Measurer** app enables you to specify an audio device, sample rate, player channel, and recorder channel. The audio device must be a real or virtual device enabled for simultaneous playback and recording (full-duplex mode) and must use a supported driver. Supported drivers are platform-specific:

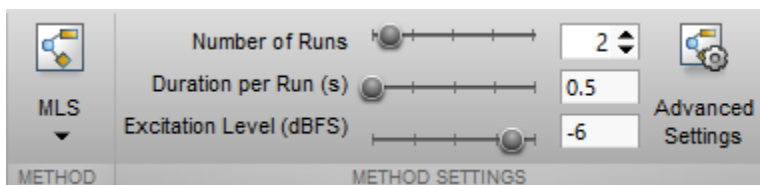
- Windows® -- ASIO™
- Mac -- CoreAudio
- Linux® -- ALSA

Valid sample rates depend on your specified audio device.

You can use the level monitor to verify the configuration of your audio I/O system.

Configure IR Acquisition Method

To configure your IR acquisition method, use the **Method** and **Method Settings** sections of the toolstrip.



You can select the method to acquire IR measurements as either:

- Maximum Length Sequences (**MLS**)
- Exponential Swept Sine (**Exponential Swept Sine**)

Both methods for IR acquisition have the same basic settings, including:

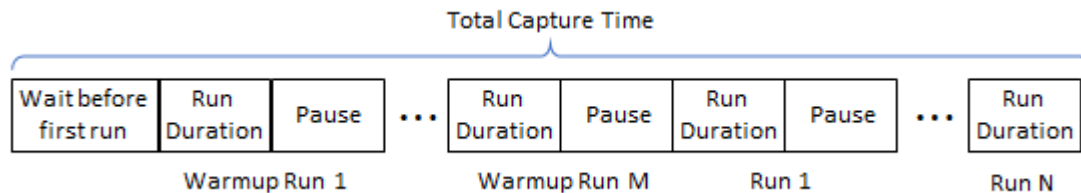
- **Number of Runs** -- Number of times the excitation signal is sent within a single capture. Multiple runs are used to average individual impulse response captures to reduce measurement noise.

- **Duration per Run (s)** -- Total time of each run in seconds.
- **Excitation Level (dBFS)** -- The level of the excitation signal in dBFS.

Both methods for IR acquisition also have the same advanced run settings, including:

- **Wait before first run** -- Delay before starting first run. The delay allows time for any last-minute tasks, such as exiting a room before testing its acoustics.
- **Pause between runs** -- Duration of pause between runs. During a pause, the excitation signal is not sent, and audio is not recorded. When using the **Exponential Swept Sine** method, include a pause between runs to avoid buildup of reverberations. Pause between runs is always zero for the **MLS** method.
- **Number of warmup runs** -- Number of times to output the excitation signal before acquisition. The **MLS** method assumes the signal it acquires is a combination of the excitation signal and its impulse response. Use warmup runs to remove transients.

The total capture time is a sum of run durations, pauses, and the initial wait:



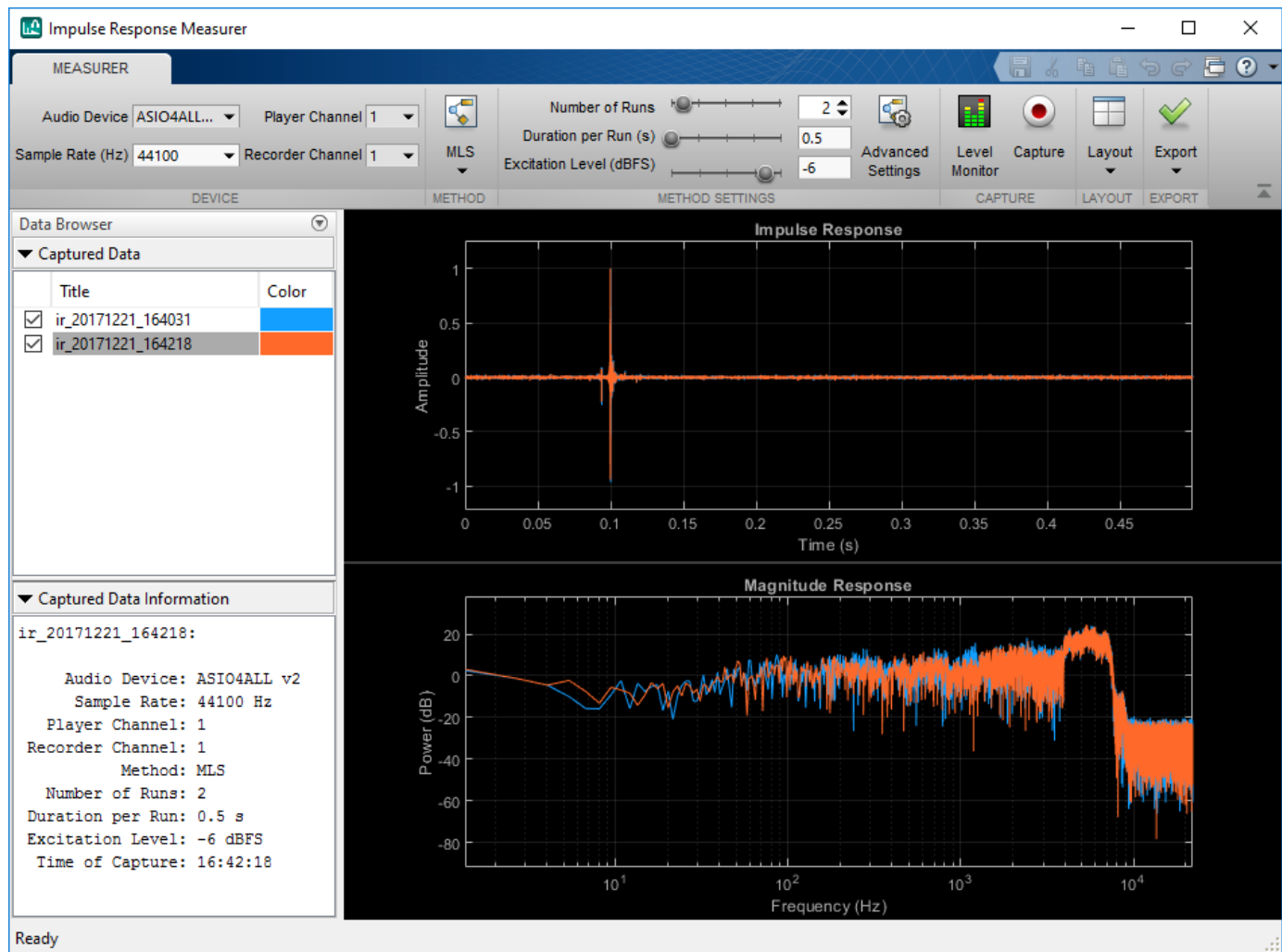
The **Exponential Swept Sine** method has additional **Advanced Settings** to control the excitation signal, including:

- **Sweep start frequency**
- **Sweep stop frequency**
- **Sweep duration**
- **End silence duration**

When using the **Exponential Swept Sine** method, the **Run Duration** is divided into **Sweep duration** and **End silence duration**. During the end silence, the app continues to record audio, enabling acquisition of the response over the entire range of the frequency sweep.

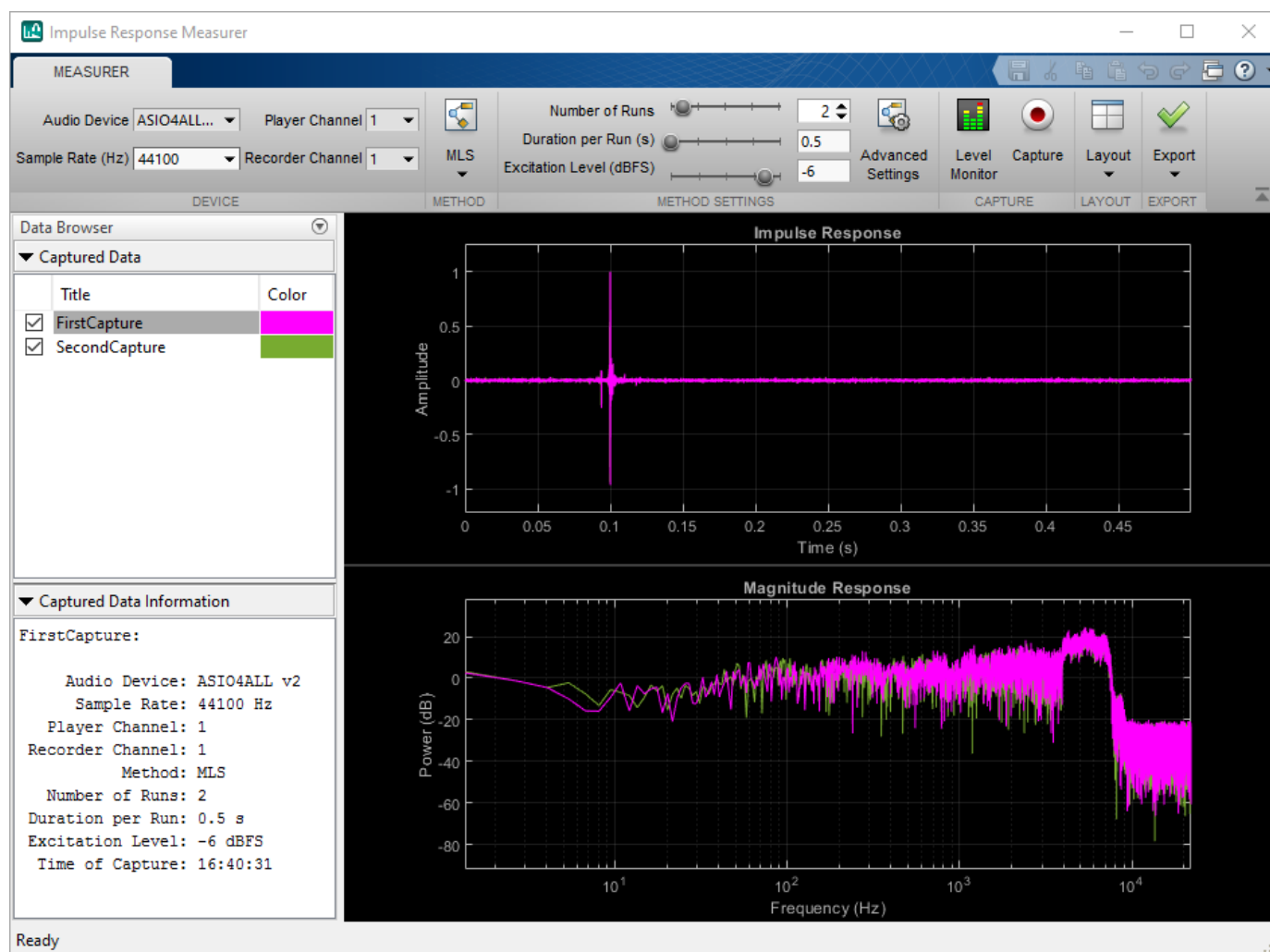
Acquire IR Measurements

For this example, use the **MLS** method with default settings. Once you have your audio device set up, click **Capture**. A dialog box opens that displays the progress of your capture. IR Measurements are captured twice.

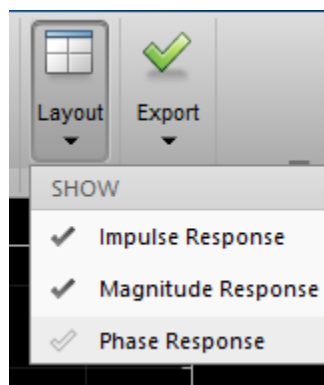


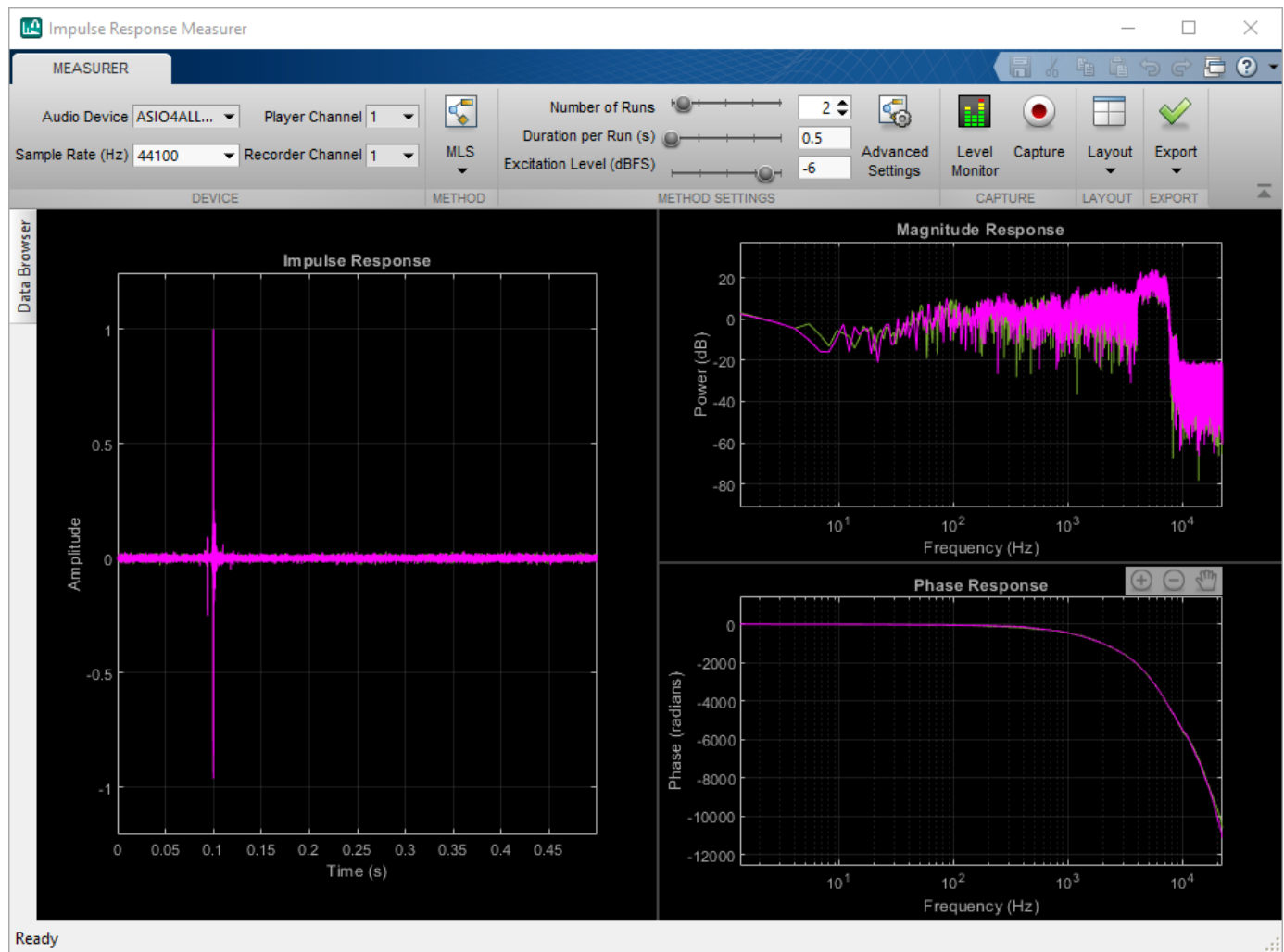
Analyze and Manage IR Measurements

After the capture, the **Impulse Response Measurer** app stores the captured data locally. The **Data Browser** displays the title of the captured data, the colors used for plotting, and information about the settings used to acquire the data. You can double-click a color in the **Data Browser** to choose which color you want associated with each impulse response. You can also double-click the title to rename your captured data. Rename your captures as **FirstCapture** and **SecondCapture**, and change the colors to pink and green. To make one impulse response plot appear on top of the other, select the title in the **Data Browser**. Select the capture you relabeled **FirstCapture**.

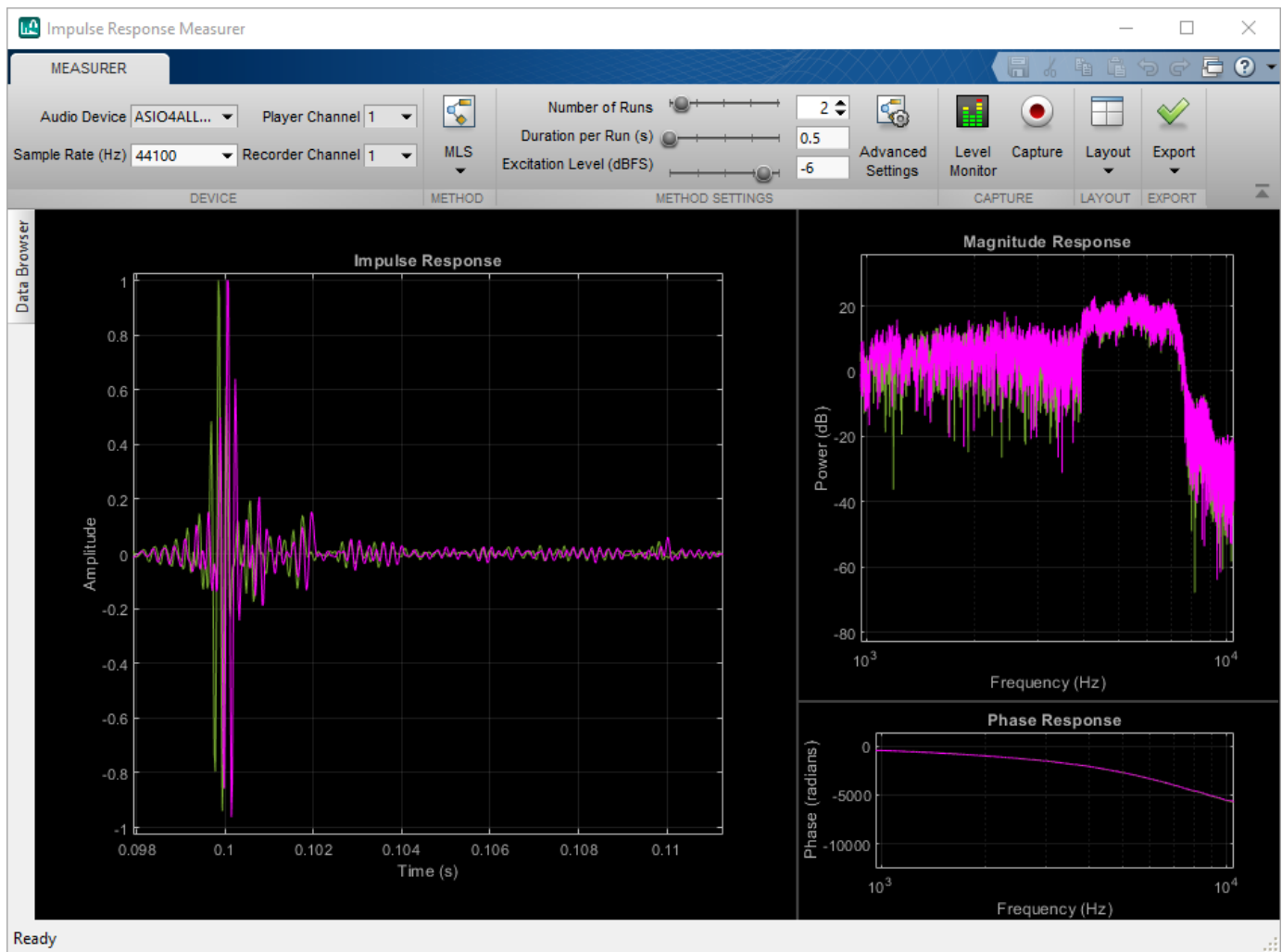


By default, the impulse response and magnitude response are plotted. You can view any combination of the impulse response, magnitude, and phase response using the **Layout** button. Minimize the **Data Browser**, then select the phase response plot for inclusion.



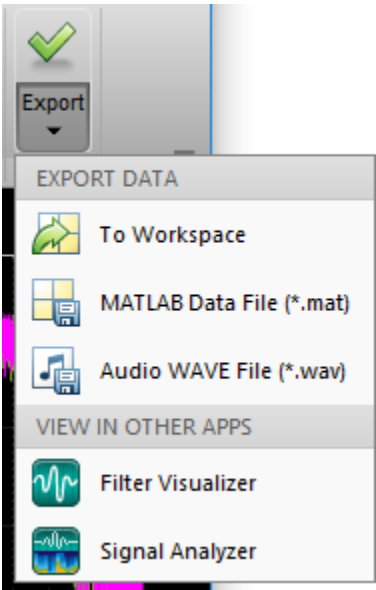


You can toggle the relative size of the plot by moving the dividers. You can zoom in and out by selecting the plus and minus icons on the UI. The icons appear when your pointer is over the plot. Zooming in and out of either the magnitude response or the phase response updates the other. Zoom in on the impulse response plot and in the range 100–1000 Hz of your frequency response plots.



Export IR Measurements

To view export options for further analysis or use, select the **Export** button.



Export the data to your workspace. The data is saved as a table. To inspect how the data is saved, display the table you exported.

```
irdata_172519
```

```
irdata_172519 =
```

```
2x14 table
```

	TimeOfCapture	ImpulseResponse	MagnitudeResponse	PhaseResponse	Device
FirstCapture	21-Dec-2017 16:40:31 -0500	[1x1 struct]	[1x1 struct]	[1x1 struct]	"ASI04ALL v2"
SecondCapture	21-Dec-2017 16:42:18 -0500	[1x1 struct]	[1x1 struct]	[1x1 struct]	"ASI04ALL v2"

When you export the data as a MAT-file, the same table is saved as when you export to the workspace. When you select to export the data as a WAV file, each impulse response is saved as a separate WAV file. The title of the capture as the name of the WAV file. In this example, selecting to export data to audio WAV file places two WAV files in the specified folder, `FirstCapture.wav` and `SecondCapture.wav`.

To analyze your captured data further, view the data in the **Filter Visualization Tool** or **Signal Analyzer** app.

See Also

Impulse Response Measurer | `audioPlayerRecorder` | `reverberator` | `splMeter`

Related Examples

- “Measure Impulse Response of an Audio System” on page 1-249
- “Measure Frequency Response of an Audio Device” on page 1-253

Design and Play a MIDI Synthesizer

Design and Play a MIDI Synthesizer

The MIDI protocol enables you to send and receive information describing sound. A MIDI synthesizer is a device or software that synthesizes sound in response to incoming MIDI data. In its simplest form, a MIDI synthesizer converts MIDI note messages to an audio signal. More complicated synthesizers provide fine-tune control of the resulting sound, enabling you to mimic instruments. In this tutorial, you create a monophonic synthesizer that converts a stream of MIDI note messages to an audio signal in real time.

To learn about interfacing with MIDI devices in general, see “MIDI Device Interface” on page 7-2.

Convert MIDI Note Messages to Sound Waves

MIDI note information is packaged as a `NoteOn` or `NoteOff` `midimsg` object in Audio Toolbox. Both `NoteOn` and `NoteOff` `midimsg` objects have `Note` and `Velocity` properties:

- **Velocity** indicates how hard a note is played. By convention, Note On messages with velocity set to zero represent note off messages. Representing note off messages with note on messages is more efficient when using Running Status.
- **Note** indicates the frequency of the audio signal. The `Note` property takes a value between zero and 127, inclusive. The MIDI protocol specifies that 60 is Middle C, with all other notes relative to that note. Create a MIDI note on message that indicates to play Middle C.

```
channel = 1;
note = 60;
velocity = 64;
msg = midimsg('NoteOn',channel,note,velocity)

msg =

    MIDI message:
      NoteOn      Channel: 1  Note: 60  Velocity: 64  Timestamp: 0  [ 90 3C 40 ]
```

To interpret the note property as frequency, use the equal tempered scale and the A440 convention:

```
frequency = 440 * 2^((msg.Note-69)/12)

frequency =

    261.6256
```

Some MIDI synthesizers use an Attack Decay Sustain Release (ADSR) envelope to control the volume, or amplitude, of a note over time. For simplicity, use the note velocity to determine the amplitude. Conceptually, if a key is hit harder, the resulting sound is louder. The `Velocity` property takes a value between zero and 127, inclusive. Normalize the velocity and interpret as the note amplitude.

```
amplitude = msg(1).Velocity/127

amplitude =

    0.5039
```

To synthesize a sine wave, create an `audioOscillator` System object™. To play the sound to your computer's default audio output device, create an `audioDeviceWriter` System object. Step the objects for two seconds and listen to the note.

```
osc = audioOscillator('Frequency',frequency,'Amplitude',amplitude);
deviceWriter = audioDeviceWriter('SampleRate',osc.SampleRate);
```

```
tic
while toc < 2
    synthesizedAudio = osc();
    deviceWriter(synthesizedAudio);
end
```

Synthesize MIDI Messages

To play an array of midimsg objects with appropriate timing, create a loop.

First, create an array of midimsg objects and cache the note on and note off times to the variable, `eventTimes`.

```
msgs = [midimsg('Note',channel,60,64,0.5,0), ...
        midimsg('Note',channel,62,64,0.5,.75), ...
        midimsg('Note',channel,57,40,0.5,1.5), ...
        midimsg('Note',channel,60,50,1,3)];
eventTimes = [msgs.Timestamp];
```

To mimic receiving notes in real time, create a for-loop that uses the `eventTimes` variable and `tic` and `toc` to play notes according to the MIDI message timestamps. Release your audio device after the loop is complete.

```
i = 1;
tic
while toc < max(eventTimes)
    if toc > eventTimes(i)
        msg = msgs(i);
        i = i+1;

        if msg.Velocity~= 0
            osc.Frequency = 440 * 2^((msg.Note-69)/12);
            osc.Amplitude = msg.Velocity/127;
        else
            osc.Amplitude = 0;
        end
    end
    deviceWriter(osc());
end
release(deviceWriter)
```

Synthesize Real-Time Note Messages from MIDI Device

To receive and synthesize note messages in real time, create an interface to a MIDI device. The `simplesynth` example function:

- receives MIDI note messages from a specified MIDI device
- synthesizes an audio signal
- plays them to your audio output device in real time

Save the `simplesynth` function to your current folder.

simplesynth

```
function simplesynth(midiDeviceName)
```

```

midiInput = mididevice(midiDeviceName);
osc = audioOscillator('square', 'Amplitude', 0);
deviceWriter = audioDeviceWriter;
deviceWriter.SupportVariableSizeInput = true;
deviceWriter.BufferSize = 64; % small buffer keeps MIDI latency low

while true
    msgs = midireceive(midiInput);
    for i = 1:numel(msgs)
        msg = msgs(i);
        if isNoteOn(msg)
            osc.Frequency = note2freq(msg.Note);
            osc.Amplitude = msg.Velocity/127;
        elseif isNoteOff(msg)
            if msg.Note == msg.Note
                osc.Amplitude = 0;
            end
        end
    end
    deviceWriter(osc());
end

function yes = isNoteOn(msg)
    yes = msg.Type == midimsgtype.NoteOn ...
        && msg.Velocity > 0;
end

function yes = isNoteOff(msg)
    yes = msg.Type == midimsgtype.NoteOff ...
        || (msg.Type == midimsgtype.NoteOn && msg.Velocity == 0);
end

function freq = note2freq(note)
    freqA = 440;
    noteA = 69;
    freq = freqA * 2.^((note-noteA)/12);
end

```

To query your system for your device name, use `mididevinfo`. To listen to your chosen device, call the `simplesynth` function with the device name. This example uses an M-Audio KeyRig 25 device, which registers with device name `USB 02` on the machine used in this example.

`mididevinfo`

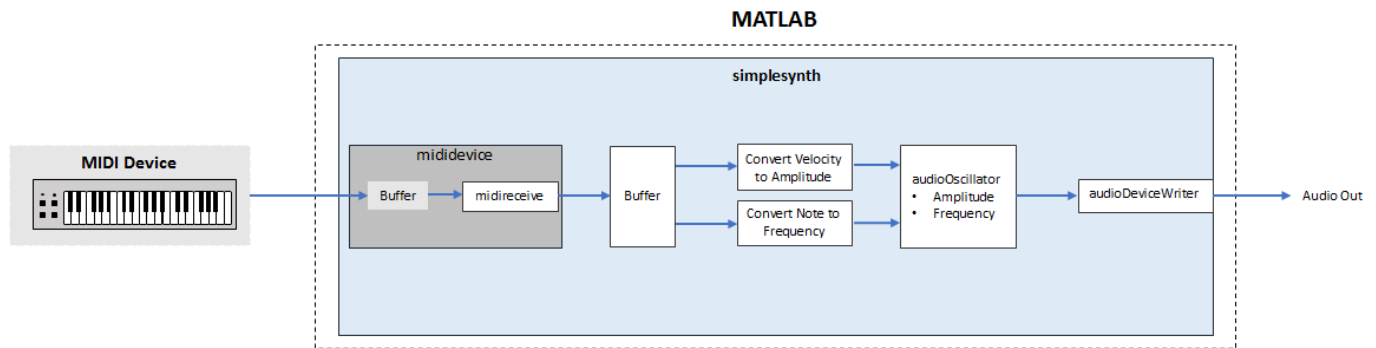
```

MIDI devices available:
ID Direction Interface Name
0 output MMSystem 'Microsoft MIDI Mapper'
1 input MMSystem 'USB MIDI Interface '
2 input MMSystem 'USB 02'
3 output MMSystem 'Microsoft GS Wavetable Synth'
4 output MMSystem 'USB MIDI Interface '
5 output MMSystem 'USB 02'

```

Call the `simplesynth` function with your device name. The `simplesynth` function listens for note messages and plays them to your default audio output device. Play notes on your MIDI device and listen to the synthesized audio.


```
simplesynth('USB 02')
```



Use Ctrl-C to end the connection.

See Also

Classes

mididevice | midimsg

Functions

mididevinfo | midireceive | midisend

External Websites

- <https://www.midi.org>

MIDI Device Interface

MIDI Device Interface

MIDI

This tutorial introduces the Musical Instrument Digital Interface (MIDI) protocol and how you can use Audio Toolbox to interact with MIDI devices. The tools described here enable you to send and receive all MIDI messages as described by the MIDI protocol. If you are interested only in sending and receiving Control Change messages with a MIDI control surface, see “MIDI Control Surface Interface” on page 10-2. If you are interested in using MIDI to control your audio plugins, see “MIDI Control for Audio Plugins” on page 9-2. To learn more about MIDI in general, consult The MIDI Manufacturers Association.

MIDI is a technical standard for communication between electronic instruments, computers, and related devices. MIDI carries event messages specific to audio signals, such as pitch and velocity, as well as control signals for parameters and clock signals to synchronize tempo.

MIDI Devices

A MIDI device is any device capable of sending or receiving MIDI messages. MIDI devices have input ports, output ports, or both. The MIDI protocol defines messages as unidirectional. A MIDI device can be real-world or virtual.

Audio Toolbox enables you to create an interface to a MIDI device using `mididevice`. To create a MIDI interface to a specific device, use `mididevinfo` to query your system for available devices. Then create a `mididevice` object by specifying a MIDI device by name or ID.

```
mididevinfo
```

```
MIDI devices available:
```

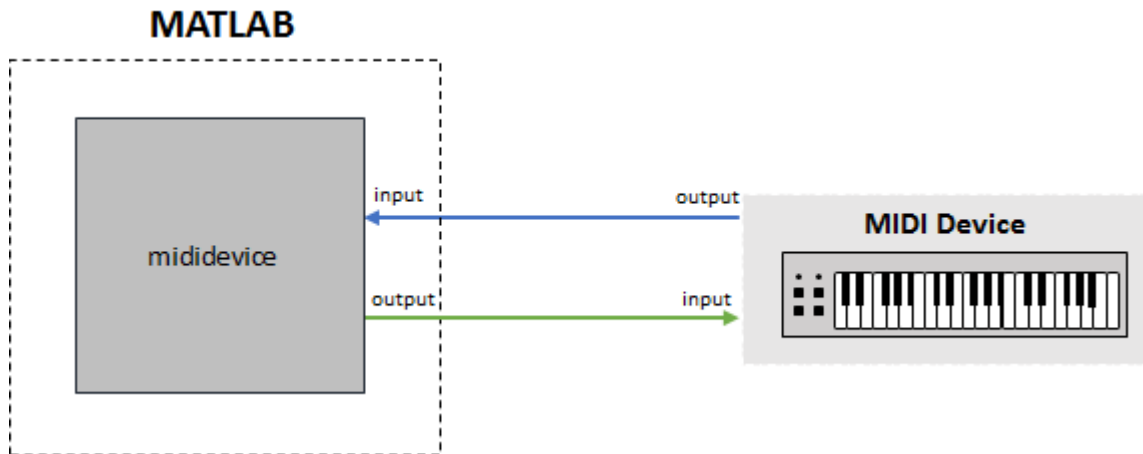
ID	Direction	Interface	Name
0	output	MMSystem	'Microsoft MIDI Mapper'
1	input	MMSystem	'USB MIDI Interface '
2	output	MMSystem	'Microsoft GS Wavetable Synth'
3	output	MMSystem	'USB MIDI Interface '

```
device = mididevice('USB MIDI Interface ')
```

```
device =
```

```
mididevice connected to  
  Input: 'USB MIDI Interface ' (1)  
  Output: 'USB MIDI Interface ' (3)
```

You can specify a `mididevice` object to listen for input messages, send output messages, or both. In this example, the `mididevice` object receives MIDI messages at the input port named 'USB MIDI Interface ', and sends MIDI messages from the output port named 'USB MIDI Interface '.



MIDI Messages

A MIDI message contains information that describes an audio-related action. For example, when you press a key on a keyboard, the corresponding MIDI message contains 3 bytes:

- 1 The first byte describes the kind of action and the channel. The first byte is referred to as the Status Byte.
- 2 The second byte describes which key is pressed. The second byte is referred to as a Data Byte.
- 3 The third byte describes how hard the key is played. The third byte is also a Data Byte.

This message is a Note On message. Note On is referred to as the message name, command, or type.

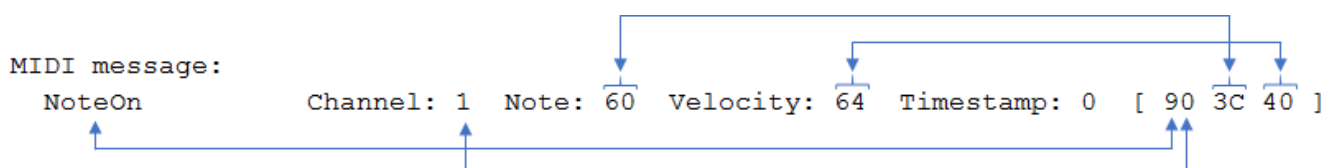
In MATLAB, a MIDI message is packaged as a `midimsg` object and can be manipulated as scalars or arrays. To create a MIDI message, call `midimsg` with a message type and then specify the required parameters for the specific message type. For example, to create a note on message, specify the `midimsg` Type as `'NoteOn'` and then specify the required inputs: channel, note, and velocity.

```
channel = 1;
note = 60;
velocity = 64;
msg = midimsg('NoteOn',channel,note,velocity)
```

```
msg =
```

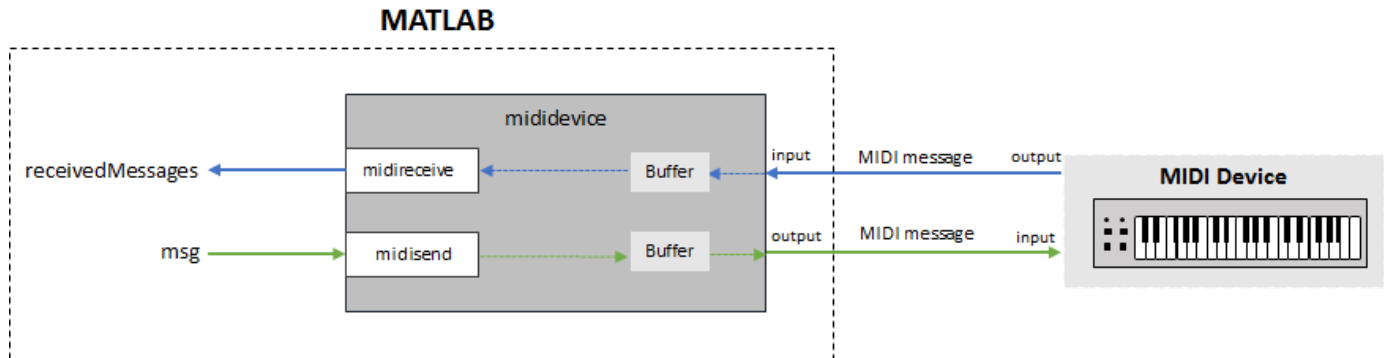
```
MIDI message:
NoteOn      Channel: 1 Note: 60 Velocity: 64 Timestamp: 0 [ 90 3C 40 ]
```

For convenience, `midimsg` displays the message type, channel, additional parameters, timestamp, and the constructed message in hexadecimal form. Hexadecimal is the preferred form because it has a straightforward interpretation:



Sending and Receiving MIDI Messages

To send and receive MIDI messages, use the `mididevice` object functions `midisend` and `midireceive`. When you create a `mididevice` object, it begins receiving data at its input and placing it in a buffer.



To retrieve MIDI messages from the buffer, call `midireceive`.

```
receivedMessages = midireceive(device)
```

```
receivedMessages =
```

```
MIDI message:
NoteOn      Channel: 1  Note: 36  Velocity: 64  Timestamp: 15861.9  [ 90 24 40 ]
NoteOn      Channel: 1  Note: 36  Velocity: 0   Timestamp: 15862.1  [ 90 24 00 ]
```

The MIDI messages are returned as an array of `midimsg` objects. In this example, a MIDI keyboard key is pressed.

To send MIDI messages to a MIDI device, call `midisend`.

```
midisend(device,msg)
```

MIDI Message Types

The type of MIDI message you create is defined as a character vector or string. To create a MIDI message, specify it by its type and the required property values. For example, create a Channel Pressure MIDI message by entering the following at the command prompt:

```
channelPressureMessage = midimsg('ChannelPressure',1,20)
```

```
channelPressureMessage =
```

```
MIDI message:
ChannelPressure Channel: 1  ChannelPressure: 20  Timestamp: 0  [ D0 14 ]
```

After you create a MIDI message, you can modify the properties, but you cannot modify the type.

```
channelPressureMessage.ChannelPressure = 37
```

```
channelPressureMessage =
```

```
MIDI message:
ChannelPressure Channel: 1  ChannelPressure: 37  Timestamp: 0  [ D0 25 ]
```

The table summarizes valid MIDI message types.

MIDI Spec Description	midimsg(Type, PropertyValue1,...,PropertyValueN)			
	Type	properties		
Channel Voice Messages				
Note Off event	NoteOff	Channel	Note	Velocity
Note On event	NoteOn	Channel	Note	Velocity
Polyphonic Key Pressure (Aftertouch)	PolyKeyPressure	Channel	Note	KeyPressure
Control Change	ControlChange	Channel	CCNumber	CCValue
Program Change	ProgramChange	Channel	Program	
Channel Pressure (Aftertouch)	ChannelPressure	Channel	ChannelPressure	
Pitch Bend Change	PitchBend	Channel	PitchChange	
Channel Mode Messages				
All Sound Off	AllSoundOff	Channel		
Reset All Controllers	ResetAllControllers	Channel		
Local Control	LocalControl	Channel	LocalControl	
All Notes Off	AllNotesOff	Channel		
Omni Mode Off	OmniOff	Channel		
Omni Mode On	OmniOn	Channel		
Mono Mode On (Poly Off)	MonoOn	Channel	MonoChannels	
Poly Mode On (Mono Off)	PolyOn	Channel		
System Common Messages				
System Exclusive	SystemExclusive			
MIDI Time Code Quarter Frame	MIDITimeCodeQuarterFrame	TimeCodeSequence	TimeCodeValue	
Song Position Pointer	Song	SongPosition		
Song Select	SongSelect	Song		
Tune Request	TuneRequest			
End of Exclusive	EOX			
System Real-Time Messages				
Timing Clock	TimingClock			
Start	Start			
Continue	Continue			
Stop	Stop			
Active Sensing	ActiveSensing			
Reset	SystemReset			

The Audio Toolbox provides convenience syntaxes to create multiple MIDI messages used in sequence and to create arrays of MIDI messages. See `midimsg` for a complete list of syntaxes.

MIDI Message Timing

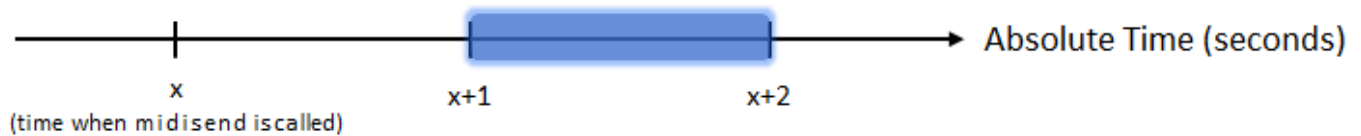
The MIDI protocol does not define message timing and assumes that messages are acted on immediately. Many applications require timing information for queuing and batch processing. For convenience, the Audio Toolbox packages timing information with MIDI messages into a single `midimsg` object. All `midimsg` objects have a `Timestamp` property, which is set during creation as an optional last argument or after creation. The default `Timestamp` is zero.

The interpretation of the `Timestamp` property depends on how a MIDI message is created and used:

- When receiving MIDI messages using `midireceive`, the underlying infrastructure assigns a timestamp when receiving MIDI messages. Conceptually, the timing clock starts when a `mididevice` object is created and attached as a listener to a given MIDI input port. If another `mididevice` is attached to the same input port, it receives timestamps from the same timing clock as the first object.
- When sending MIDI messages using `midisend`, timestamps are interpreted as when to send the message.

If there have been no recent calls to `midisend`, then `midisend` interprets timestamps as relative to the current real-world time. A message with a timestamp of zero is sent immediately. If there has been a recent call to `midisend`, then `midisend` interprets timestamps as relative to the largest timestamp of the last call to `midisend`. The timestamp clock for `midisend` is specific to the MIDI output port that `mididevice` is connected to.

Consider a pair of MIDI messages that turn a note on and off. The messages specify that the note starts after one second and is sustained for one second.

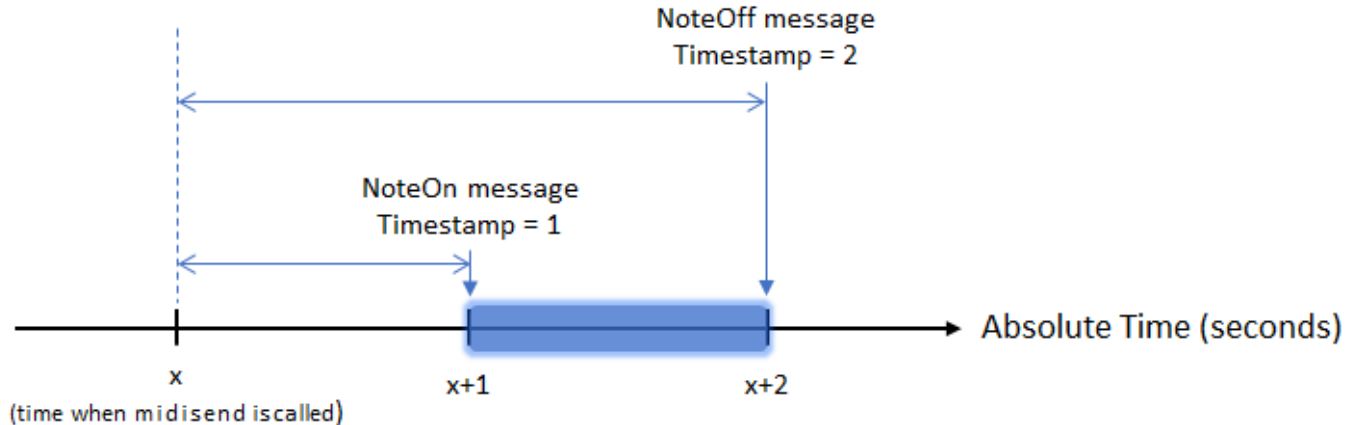


Create Note On and Note Off messages.

```
OnMsg = midimsg('NoteOn',1,59,64);
OffMsg = midimsg('NoteOff',1,59,0);
```

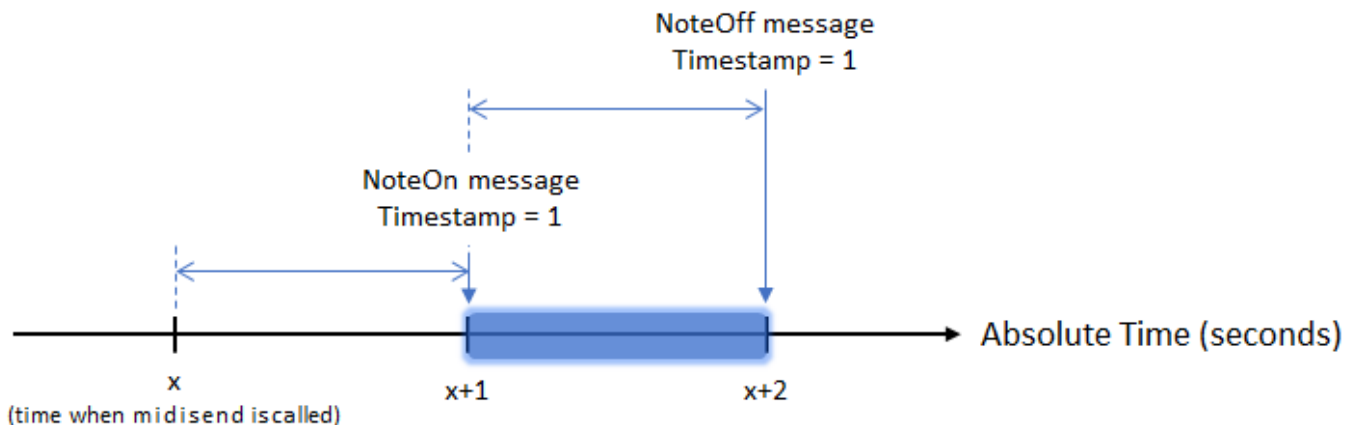
To send on and off messages using a single call to `midisend`, specify the timestamps of the messages relative to the same start time.

```
OnMsg.Timestamp = 1;
OffMsg.Timestamp = 2;
midisend(device,[OnMsg;OffMsg])
```

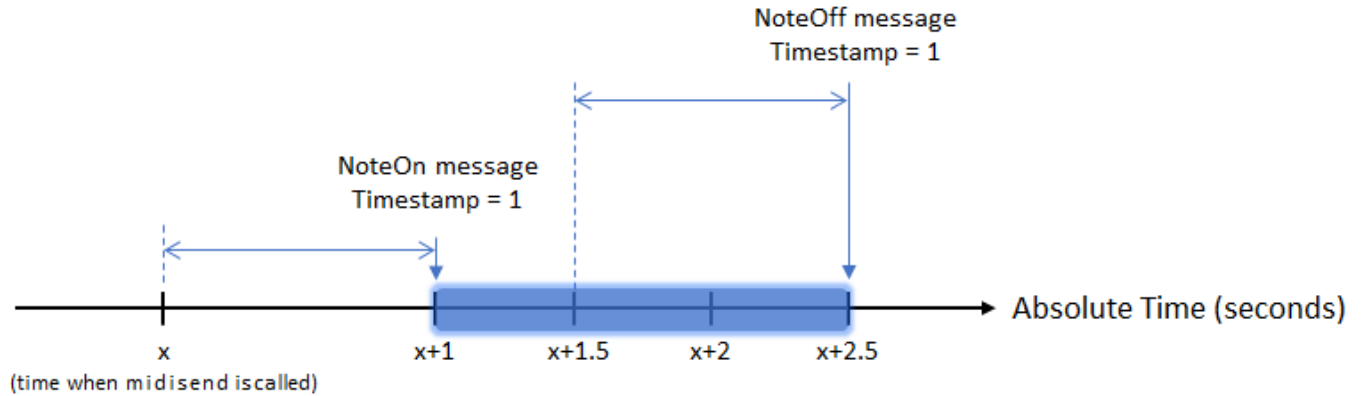
To send the Note Off message separately, specify the timestamp of the Note Off message relative to the largest timestamp of the previous call to `midisend`.

```
OnMsg.Timestamp = 1;
OffMsg.Timestamp = 1;
midisend(device, OnMsg)
midisend(device, OffMsg)
```



The "start" time, or reference time, for `midisend` is the max between the absolute time and the largest timestamp in the last call to `midisend`. For example, consider that x , the arbitrary start time, is equal to the current absolute time. If there is a 1.5-second pause between sending the note on and note off messages, the resulting note duration is 1.5 seconds.

```
OnMsg.Timestamp = 1;
OffMsg.Timestamp = 1;
midisend(device, OnMsg)
pause(1.5)
midisend(device, OffMsg)
```



Usually, MIDI messages are sent faster than or at real-time speeds so there is no need to track the absolute time.

For live performances or to enable interrupts in a MIDI stream, you can set timestamps to zero and then call `midisend` at appropriate real-world time intervals. Depending on your use case, you can divide your MIDI stream into small repeatable time chunks.

See Also

Classes

`mididevice` | `midimsg`

Functions

`mididevinfo` | `midireceive` | `midisend`

Related Examples

- “Design and Play a MIDI Synthesizer” on page 6-2

External Websites

- MIDI Manufacturers Association
- Summary of MIDI Messages

Dynamic Range Control

Dynamic Range Control

Dynamic range control is the adaptive adjustment of the dynamic range of a signal. The dynamic range of a signal is the logarithmic ratio of maximum to minimum signal amplitude specified in dB.

You can use dynamic range control to:

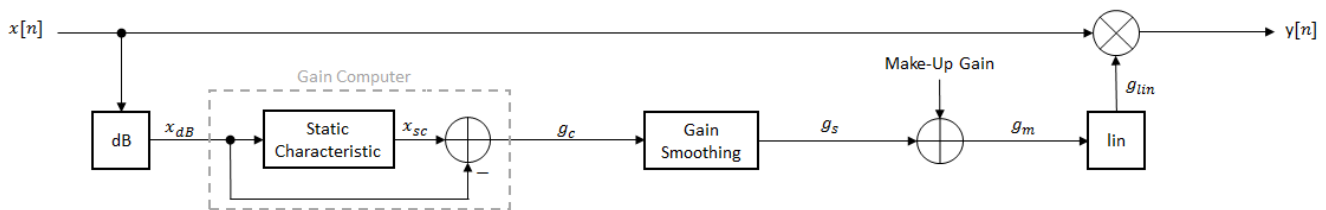
- Match an audio signal level to its environment
- Protect AD converters from overload
- Optimize information
- Suppress low-level noise

Types of dynamic range control include:

- Dynamic range compressor -- Attenuates the volume of loud sounds that cross a given threshold. They are often used in recording systems to protect hardware and to increase overall loudness.
- Dynamic range limiter -- A type of compressor that brickwalls sound above a given threshold.
- Dynamic range expander -- Attenuates the volume of quiet sounds below a given threshold. They are often used to make quiet sounds even quieter.
- Noise gate -- A type of expander that brickwalls sound below a given threshold.

This tutorial shows how to implement dynamic range control systems using the `compressor`, `expander`, `limiter`, and `noiseGate` System objects from Audio Toolbox. The tutorial also provides an illustrated example of dynamic range limiting at various stages of a dynamic range limiting system.

The diagram depicts a general dynamic range control system.



In a dynamic range control system, a gain signal is calculated in a sidechain and then applied to the input audio signal. The sidechain consists of:

- Linear to dB conversion: $x \rightarrow x_{dB}$
- Gain computation, by passing the dB signal through a static characteristic equation, and then taking the difference: $g_c = x_{sc} - x_{dB}$
- Gain smoothing over time: $g_c \rightarrow g_s$
- Addition of make-up gain (for compressors and limiters only): $g_s \rightarrow g_m$
- dB to linear conversion: $g_m \rightarrow g_{lin}$
- Application of the calculated gain signal to the original audio signal: $y = g_{lin} \times x$

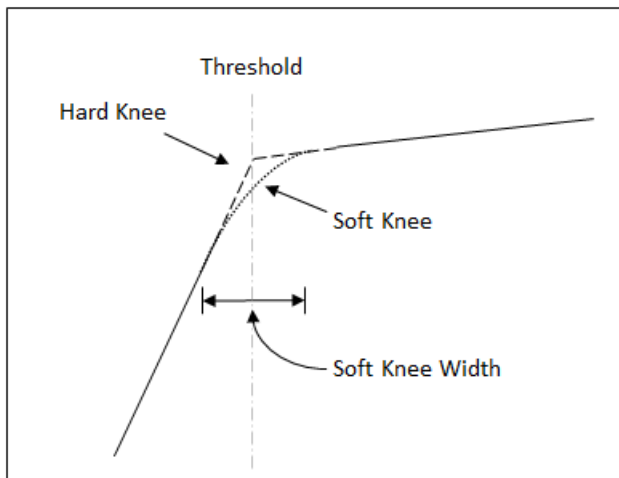
Linear to dB Conversion

The gain signal used in dynamic range control is processed on a dB scale for all dynamic range controllers. There is no reference for the dB output; it is a straight conversion: $x_{dB} = 20\log_{10}(x)$. You might need to adjust the output of a dynamic range control system to the range of your system.

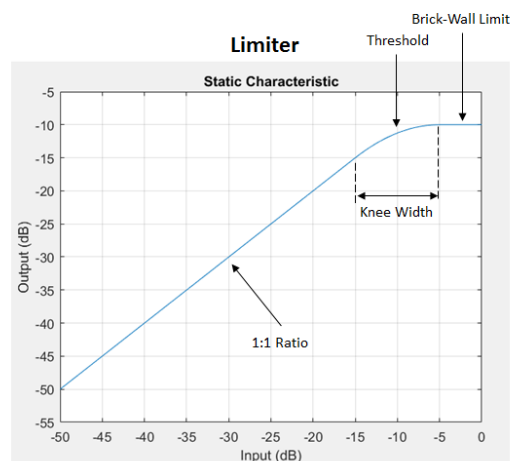
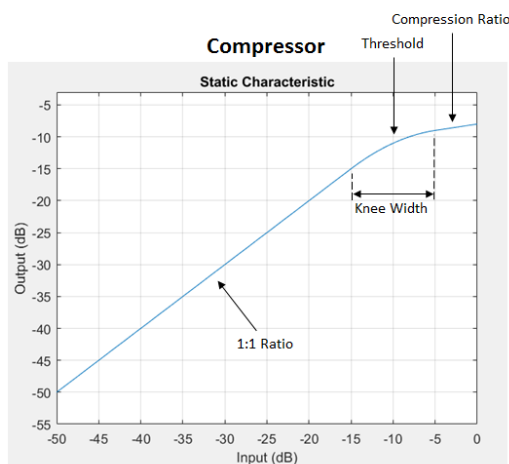
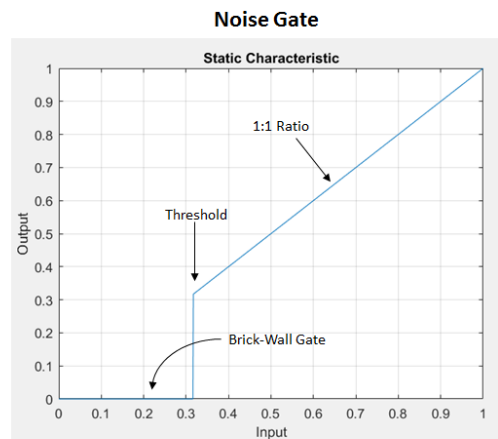
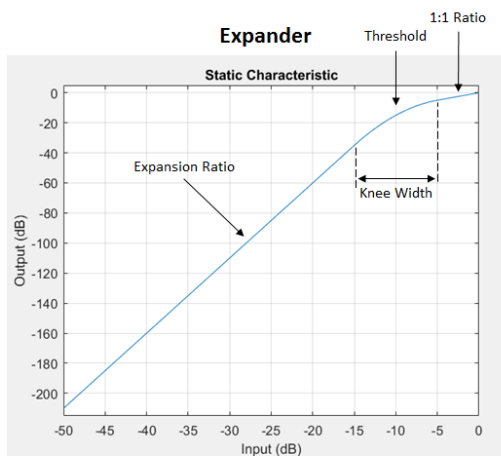
Gain Computer

The gain computer provides the first rough estimate of a gain signal for dynamic range control. The principal component of the gain computer is the static characteristic. Each type of dynamic range control has a different static characteristic with different tunable properties:

- **Threshold** -- All static characteristics have a threshold. On one side of the threshold, the input is given to the output with no modification. On the other side of the threshold, compression, expansion, brickwall limiting, or brickwall gating is applied.
- **Ratio** -- Expanders and compressors enable you to adjust the input-to-output ratio of the static characteristic above or below a given threshold.
- **KneeWidth** -- Expanders, compressors, and limiters enable you to adjust the knee width of the static characteristic. The knee of a static characteristic is centered at the threshold. An increase in knee width creates a smoother transition around the threshold. A knee width of zero provides no smoothing and is known as a hard knee. A knee width greater than zero is known as a soft knee.



In these static characteristic plots, the expander, limiter, and compressor each have a 10 dB knee width.



Gain Smoothing

All dynamic range controllers provide gain smoothing over time. Gain smoothing diminishes sharp jumps in the applied gain, which can result in artifacts and an unnatural sound. You can conceptualize gain smoothing as the addition of impedance to your gain signal.

The `expander` and `noiseGate` objects have the same smoothing equation, because a noise gate is a type of expander. The `limiter` and `compressor` objects have the same smoothing equation, because a limiter is a type of compressor.

The type of gain smoothing is specified by a combination of attack time, release time, and hold time coefficients. Attack time and release time correspond to the time it takes the gain signal to go from 10% to 90% of its final value. Hold time is a delay period before gain is applied. See the algorithms of individual dynamic range controller pages for more detailed explanations.

Smoothing Equations

expander and noiseGate

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) g_c[n] & \text{if } (C_A > k) \ \& \ (g_c[n] \leq g_s[n-1]) \\ g_s[n-1] & \text{if } C_A \leq k \\ \alpha_R g_s[n-1] + (1 - \alpha_R) g_c[n] & \text{if } (C_R > k) \ \& \ (g_c[n] > g_s[n-1]) \\ g_s[n-1] & \text{if } C_R \leq k \end{cases}$$

- α_A and α_R are determined by the sample rate and specified attack and release time:

$$\alpha_A = \exp\left(\frac{-\log(9)}{F_S \times T_A}\right), \quad \alpha_R = \exp\left(\frac{-\log(9)}{F_S \times T_R}\right)$$

- k is the specified hold time in samples.
- C_A and C_R are hold counters for attack and release, respectively.

compressor and limiter

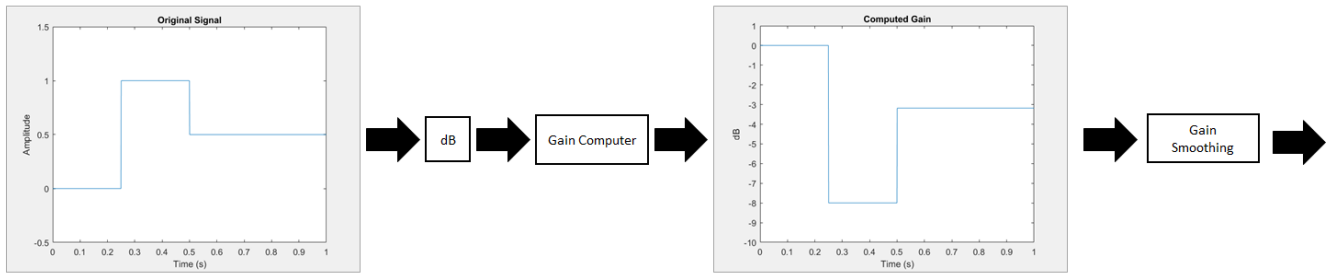
$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) g_c[n] & \text{if } g_c[n] \leq g_s[n-1] \\ \alpha_R g_s[n-1] + (1 - \alpha_R) g_c[n] & \text{if } g_c[n] > g_s[n-1] \end{cases}$$

- α_A and α_R are determined by the sample rate and specified attack and release time:

$$\alpha_A = \exp\left(\frac{-\log(9)}{F_S \times T_A}\right), \quad \alpha_R = \exp\left(\frac{-\log(9)}{F_S \times T_R}\right)$$

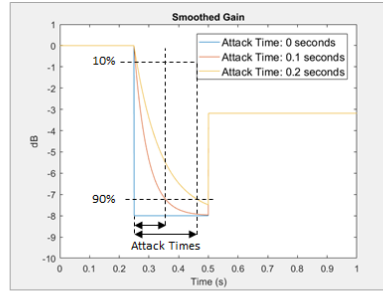
Gain Smoothing Example

Examine a trivial case of dynamic range compression for a two-step input signal. In this example, the compressor has a threshold of -10 dB, a compression ratio of 5, and a hard knee.

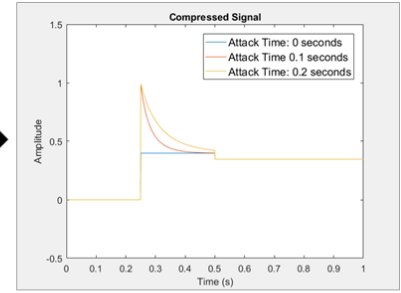


Several variations of gain smoothing are shown. On the top, a smoothed gain curve is shown for different attack time values, with release time set to zero seconds. In the middle, release time is varied and attack time is held constant at zero seconds. On the bottom, both attack and release time are specified by nonzero values.

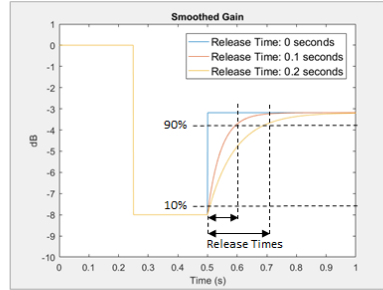
Vary Attack Time
(Release Time = 0 s)



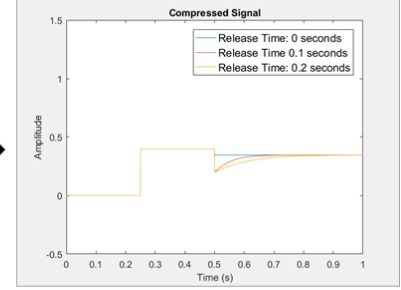
...



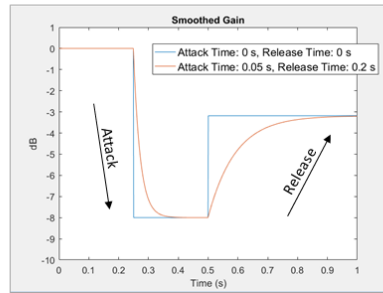
Vary Release Time
(Attack Time = 0 s)



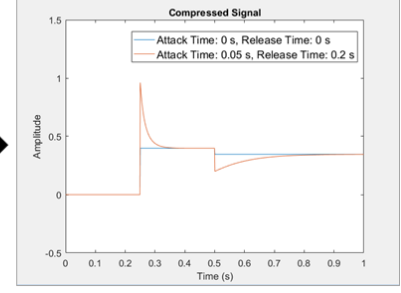
...



Vary Release and Attack Time



...



Make-Up Gain

Make-up gain applies for compressors and limiters, where higher dB portions of a signal are attenuated or brickwalled. The dB reduction can significantly reduce total signal power. In these cases, make-up gain is applied after gain smoothing to increase the signal power. In Audio Toolbox, you can specify a set amount of make-up gain or specify the make-up gain mode as 'auto'.

The 'auto' make-up gain ensures that a 0 dB input results in a 0 dB output. For example, assume a static characteristic of a compressor with a soft knee:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{\left(\frac{1}{R} - 1\right)\left(x_{dB} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \leq x_{dB} \leq \left(T + \frac{W}{2}\right) \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases}$$

T is the threshold, W is the knee width, and R is the compression ratio. The calculated auto make-up gain is the negative of the static characteristic equation evaluated at 0 dB:

$$\text{MAKE-UP GAIN} = -x_{sc}(0) = \begin{cases} 0 & \frac{W}{2} < T \\ -\frac{\left(\frac{1}{R} - 1\right)\left(T - \frac{W}{2}\right)^2}{2W} & -\frac{W}{2} \leq T \leq \frac{W}{2} \\ -T + \frac{T}{R} & -\frac{W}{2} > T \end{cases}$$

dB to Linear Conversion

Once the gain signal is determined in dB, it is transferred to the linear domain: $g_{lin} = 10^{g_m/20}$.

Apply Calculated Gain

The final step in a dynamic control system is to apply the calculated gain by multiplication in the linear domain.

Example: Dynamic Range Limiter

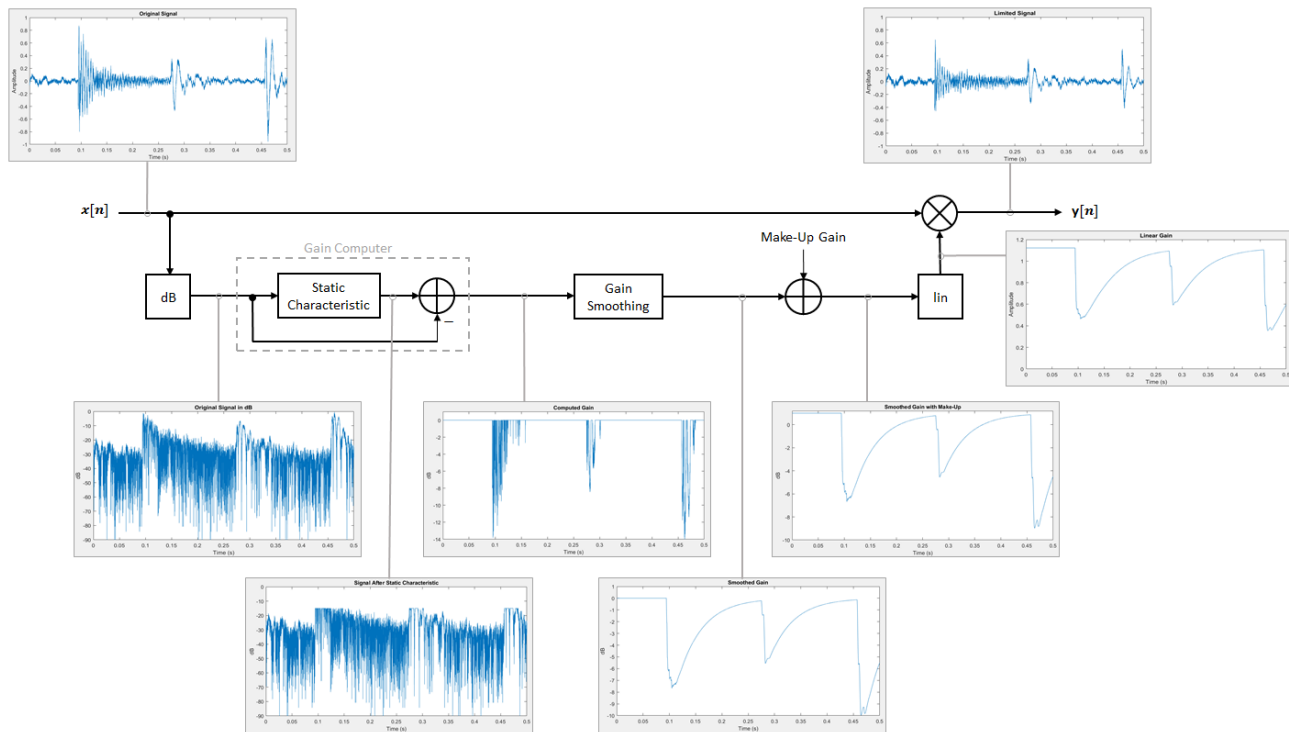
The audio signal described in this example is a 0.5 second interval of a drum track. The limiter properties are:

- Threshold = -15 dB
- Knee width = 0 (hard knee)
- Attack time = 0.004 seconds
- Release time = 0.1 seconds
- Make-up gain = 1 dB

To create a `limiter` System object with these properties, at the MATLAB command prompt, enter:

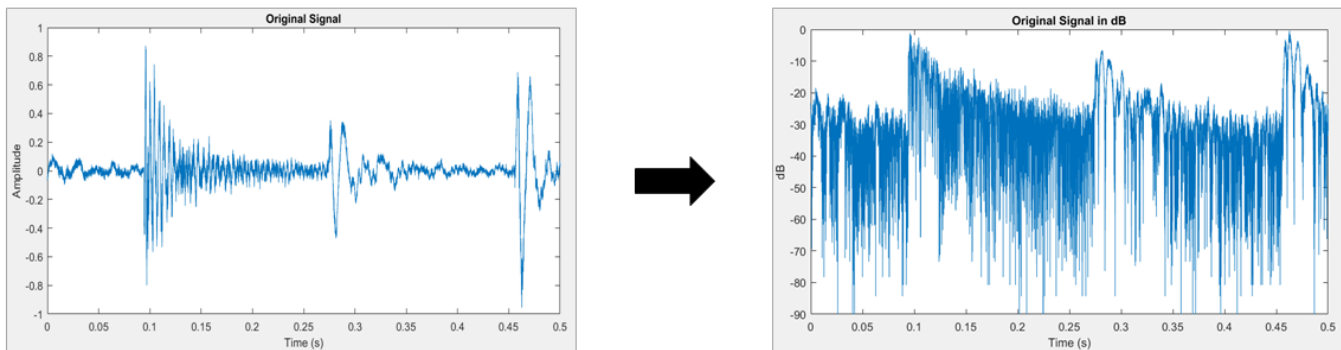
```
dRL = limiter('Threshold',-15,...
              'KneeWidth',0,...
              'AttackTime',0.004,...
              'ReleaseTime',0.1,...
              'MakeUpGainMode','property',...
              'MakeUpGain',1);
```

This example provides a visual walkthrough of the various stages of the dynamic range limiter system.



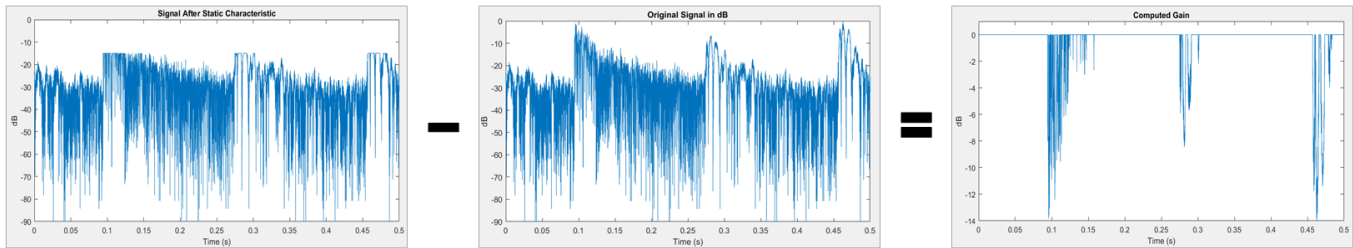
Linear to dB Conversion

The input signal is converted to a dB scale element by element.



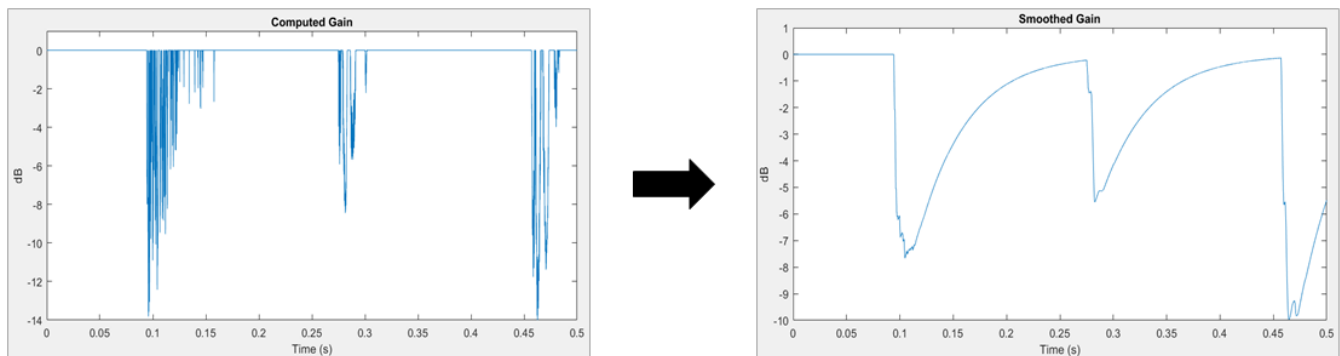
Gain Computer

The static characteristic brickwall limits the dB signal at -15 dB. To determine the dB gain that results in this limiting, the gain computer subtracts the original dB signal from the dB signal processed by the static characteristic.



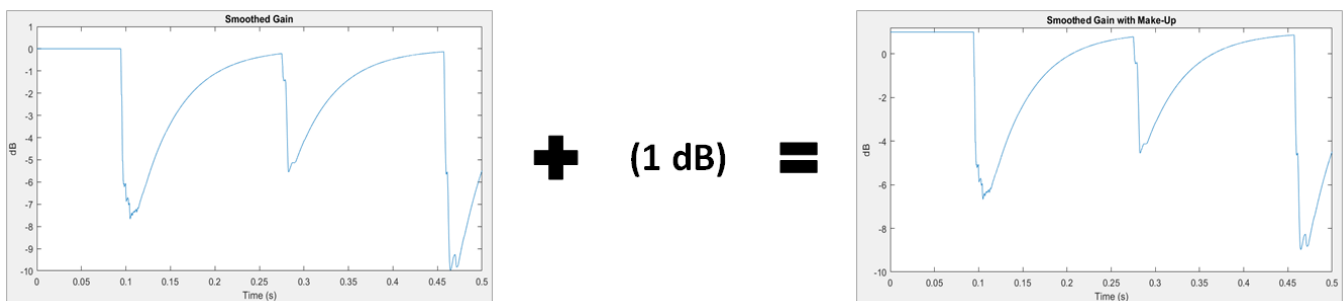
Gain Smoothing

The relatively short attack time specification results in a steep curve when the applied gain is suddenly increased. The relatively long release time results in a gradual diminishing of the applied gain.



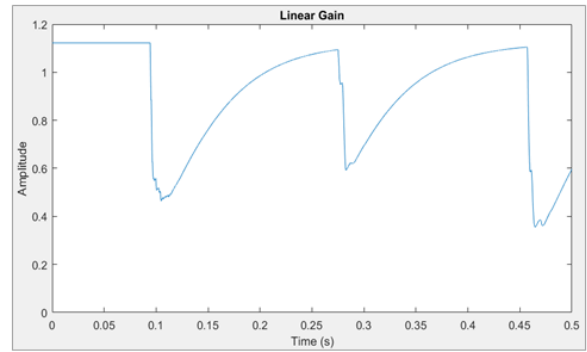
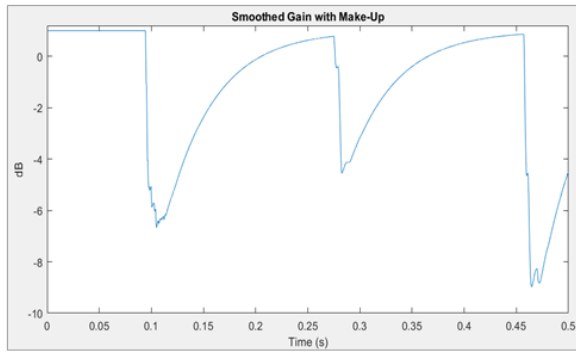
Make-Up Gain

Assume a limiter with a 1 dB make-up gain value. The make-up gain is added to the smoothed gain signal.



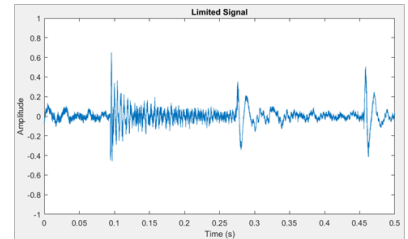
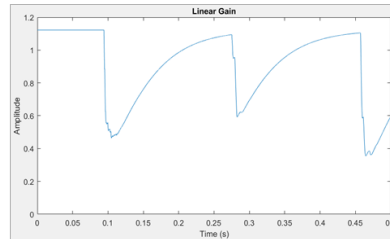
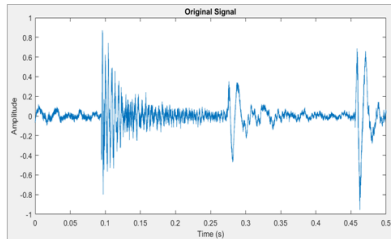
dB to Linear Conversion

The gain in dB is converted to a linear scale element by element.



Apply Calculated Gain

The original signal is multiplied by the linear gain.



References

- [1] Zolzer, Udo. "Dynamic Range Control." *Digital Audio Signal Processing*. 2nd ed. Chichester, UK: Wiley, 2008.
- [2] 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.

See Also

Compressor | Expander | Limiter | Noise Gate | compressor | expander | limiter | noiseGate

More About

- "Dynamic Range Compression Using Overlap-Add Reconstruction" on page 1-148

MIDI Control for Audio Plugins

MIDI Control for Audio Plugins

MIDI and Plugins

MIDI control surfaces are commonly used in conjunction with audio plugins in digital audio workstation (DAW) environments. Synchronizing MIDI controls with plugin parameters provides a tangible interface for audio processing and is an efficient approach to parameter tuning.

In the MATLAB environment, audio plugins are defined as any valid class that derives from the `audioPlugin` base class or the `audioPluginSource` base class. For more information about how audio plugins are defined in the MATLAB environment, see “Audio Plugins in MATLAB”.

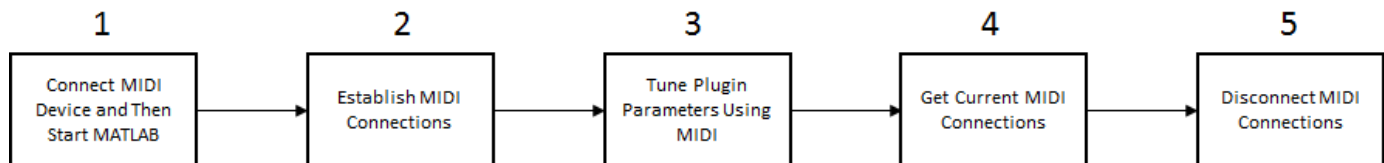
Use MIDI with MATLAB Plugins

The Audio Toolbox product provides three functions for enabling the interface between MIDI control surfaces and audio plugins:

- `configureMIDI` -- Configure MIDI connections between audio plugin and MIDI controller.
- `getMIDIConnections` -- Get MIDI connections of audio plugin.
- `disconnectMIDI` -- Disconnect MIDI controls from audio plugin.

These functions combine the abilities of general MIDI functions into a streamlined and user-friendly interface suited to audio plugins in MATLAB. For a tutorial on the general functions and the MIDI protocol, see “MIDI Control Surface Interface” on page 10-2.

This tutorial walks you through the MIDI functions for audio plugins in MATLAB.



1. Connect MIDI Device and Then Start MATLAB

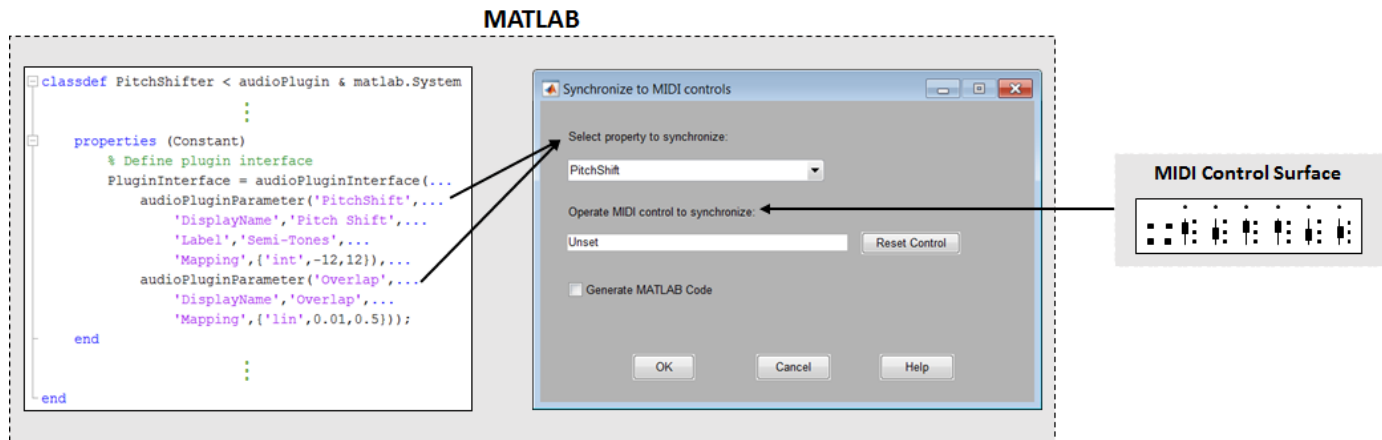
Before starting MATLAB, connect your MIDI control surface to your computer and turn it on. For connection instructions, see the instructions for your MIDI device. If you start MATLAB before connecting your device, MATLAB might not recognize your device when you connect it. To correct the problem, restart MATLAB with the device already connected.

2. Establish MIDI Connections

Use `configureMIDI` to establish MIDI connections between your default MIDI device and an audio plugin. You can use `configureMIDI` programmatically, or you can open a user interface (UI) to guide you through the process. The `configureMIDI` UI reads from your audio plugin and populates a drop-down list of tunable plugin properties. You are then prompted to move individual controls on your MIDI control surface to associate the position of each control with the normalized value of each property you select. For example, create an object of `audiopluginexample.PitchShifter` and then call `configureMIDI` with the object as the argument:

```
ctrlPitch = audiopluginexample.PitchShifter;
configureMIDI(ctrlPitch)
```

The Synchronize to MIDI controls dialog box opens with the tunable properties of your plugin automatically populated. When you operate a MIDI control, its identification is entered into the **Operate MIDI control to synchronize** box. After you synchronize tunable properties with MIDI controls, click **OK** to complete the configuration. If your MIDI control surface is bidirectional, it automatically shifts the position of the synchronized controls to the initial property values specified by your plugin.



To open a MATLAB function with the programmatic equivalent of your actions in the UI, select the **Generate MATLAB Code** check box. Saving this function enables you to reuse your settings and quickly establish the configuration in future sessions.

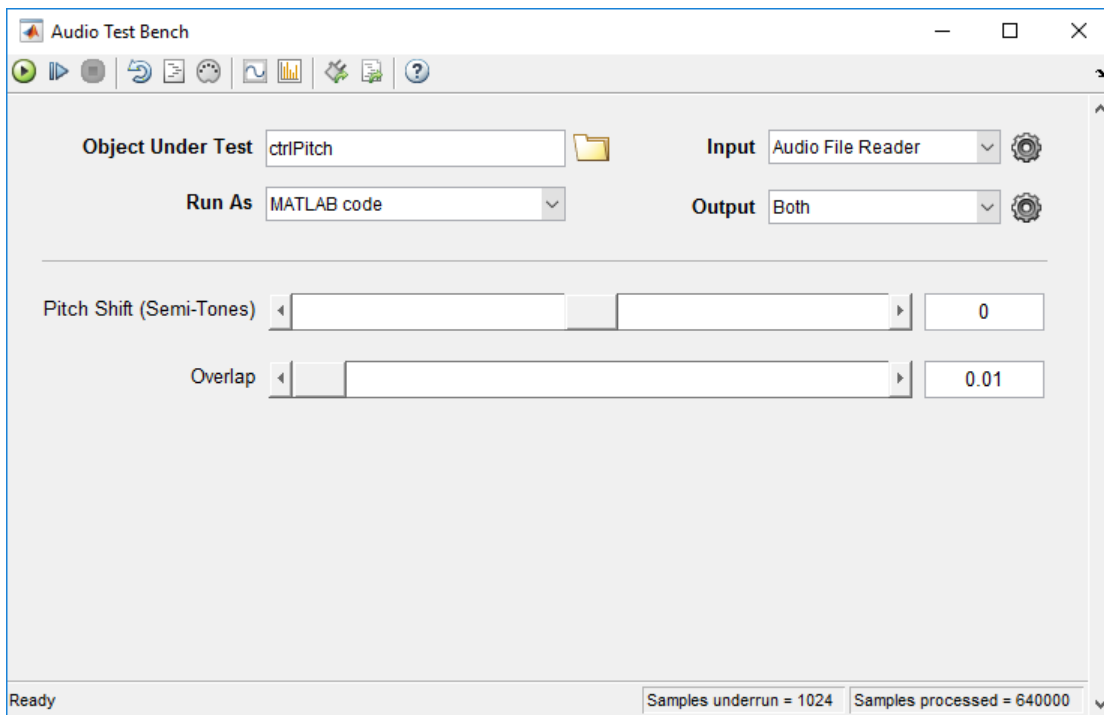
3. Tune Plugin Parameters Using MIDI


After you establish connections between plugin properties and MIDI controls, you can tune the properties in real time using your MIDI control surface.

Audio Toolbox provides an all-in-one app for running and testing your audio plugin. The test bench mimics how a DAW interacts with plugins.

Open the **Audio Test Bench** for your ctrlPitch object.

```
audioTestBench(ctrlPitch)
```



When you adjust the controls on your MIDI surface, the corresponding plugin parameter sliders move. Click  to run the plugin. Move the controls on your MIDI surface to hear the effect of tuning the plugin parameters.

To establish MIDI connections and modify existing ones, click the Synchronize to MIDI Controls  button to open a configureMIDI UI.

Alternatively, you can use the MIDI connections you established in a script or function. For example, run the following code and move your synchronized MIDI controls to hear the pitch-shifting effect:

```
fileReader = dsp.AudioFileReader(...
    'Filename','Counting-16-44p1-mono-15secs.wav');
deviceWriter = audioDeviceWriter;

% Audio stream loop
while ~isDone(fileReader)
    input = fileReader();
    output = ctrlPitch(input);
    deviceWriter(output);
    drawnow limitrate; % Process callback immediately
end

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

4. Get Current MIDI Connections

To query the MIDI connections established with your audio plugin, use the `getMIDIConnections` function. `getMIDIConnections` returns a structure with fields corresponding to the tunable

properties of your plugin. The corresponding values are nested structures containing information about the mapping between your plugin property and the specified MIDI control.

```
connectionInfo = getMIDIConnections(ctrlPitch)
```

```
connectionInfo =
```

```
    struct with fields:
```

```
        PitchShift: [1×1 struct]
        Overlap: [1×1 struct]
```

```
connectionInfo.PitchShift
```

```
ans =
```

```
    struct with fields:
```

```
        Law: 'int'
        Min: -12
        Max: 12
        MIDIControl: 'control 1081 on 'BCF2000''
```

5. Disconnect MIDI Surface

As a best practice, release external devices such as MIDI control surfaces when you are done.

```
disconnectMIDI(ctrlPitch)
```

See Also

Apps

Audio Test Bench

Classes

audioPlugin | audioPluginSource

Functions

configureMIDI | disconnectMIDI | getMIDIConnections

More About

- “What Are DAWs, Audio Plugins, and MIDI Controllers?”
- “MIDI Control Surface Interface” on page 10-2
- “Audio Plugins in MATLAB”
- “Host External Audio Plugins”

External Websites

- <https://www.midi.org>

MIDI Control Surface Interface

MIDI Control Surface Interface

In this section...

“About MIDI” on page 10-2

“MIDI Control Surfaces” on page 10-2

“Use MIDI Control Surfaces with MATLAB and Simulink” on page 10-3

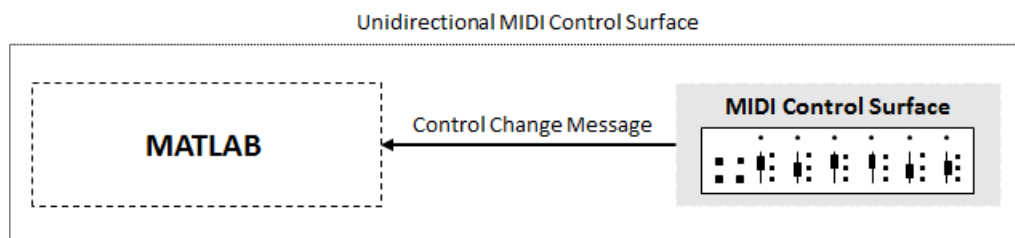
About MIDI

Musical Instrument Digital Interface (MIDI) was originally developed to interconnect electronic musical instruments. This interface is flexible and has uses in applications far beyond musical instruments. Its simple unidirectional messaging protocol supports many different kinds of messaging. One kind of MIDI message is the Control Change message, which is used to communicate changes in controls, such as knobs, sliders, and buttons.

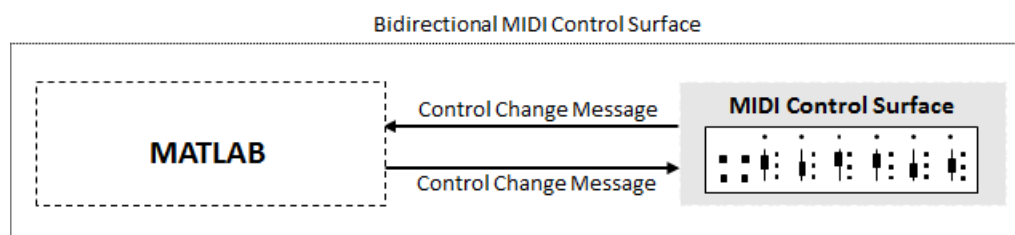
MIDI Control Surfaces

A MIDI control surface is a device with controls that sends MIDI Control Change messages when you turn a knob, move a slider, or push a button on its surface. Each Control Change message indicates which control changed and what its new position is.

Because the MIDI messaging protocol is unidirectional, determining a particular controller position requires that the receiver listen for Control Change messages that the controller sends. The protocol does not support querying the MIDI controller for its position.



The simplest MIDI control surfaces are unidirectional: They send MIDI Control Change messages but do not receive them. More sophisticated control surfaces are bidirectional: They can both send and receive Control Change messages. These control surfaces have knobs or sliders that can operate automatically. For example, a control surface can have motorized sliders or knobs. When it receives a Control Change message, the appropriate control moves to the position in the message.

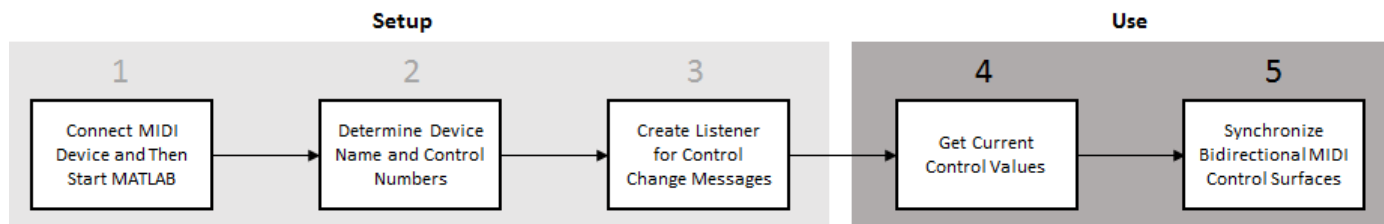


Use MIDI Control Surfaces with MATLAB and Simulink

Audio Toolbox enables you to use MIDI control surfaces to control MATLAB programs and Simulink® models by providing the capability to listen to Control Change messages. The toolbox also provides a limited capability to send Control Change messages to support synchronizing MIDI controls. This tutorial covers general MIDI functions. For functions specific to audio plugins in MATLAB, see “MIDI Control for Audio Plugins” on page 9-2. The Audio Toolbox general interface to MIDI control surfaces includes five functions and one block:

- `midid` -- Interactively identify MIDI control.
- `midicontrols` -- Open group of MIDI controls for reading.
- `midiread` -- Return most recent value of MIDI controls.
- `midisync` -- Send values to MIDI controls for synchronization.
- `midicallback` -- Call function handle when MIDI controls change value.
- MIDI Controls (block) -- Output values from controls on MIDI control surface. The MIDI Controls block combines functionality of the general MIDI functions into one block for the Simulink environment.

This diagram shows a typical workflow involving general MIDI functions in MATLAB. For the Simulink environment, follow steps 1 and 2, and then use the MIDI Controls block for a user-interface guided workflow.



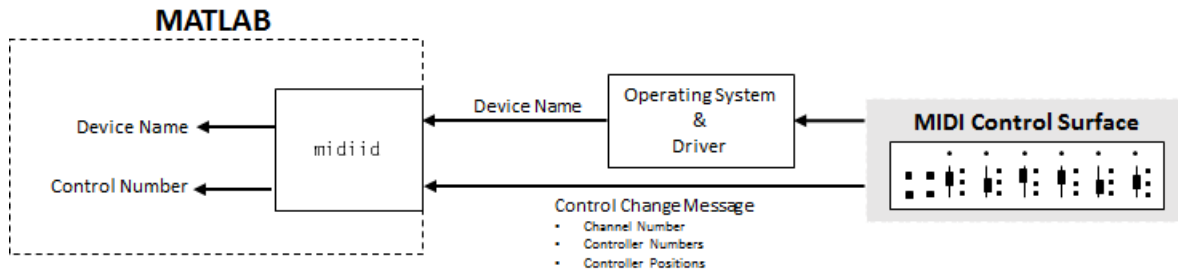
1. Connect MIDI Device and Then Start MATLAB

Before starting MATLAB, connect your MIDI control surface to your computer and turn it on. 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.

2. Determine Device Name and Control Numbers

Use the `midid` function to determine the device name and control numbers of your MIDI control surface. After you call `midid`, it continues to listen until it receives a Control Change message. When it receives a Control Change message, it returns the control number associated with the MIDI controller number that you manipulated, and optionally returns the device name of your MIDI control surface. The manufacturer and host operating system determine the device name. See “Control Numbers” on page 10-7 for an explanation of how MATLAB calculates the control number.

To set a default device name, see “Set Default MIDI Device” on page 10-7.



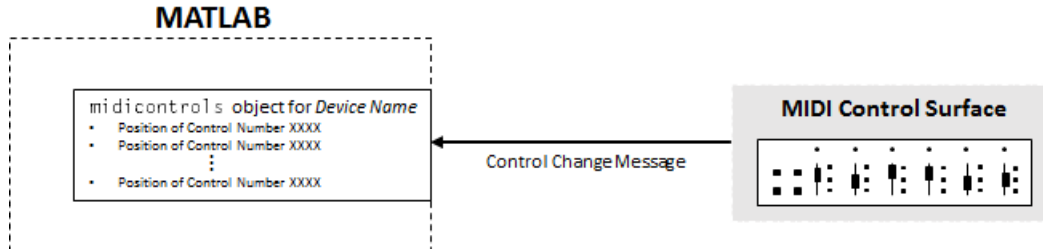
View Example

Call `midid` with two outputs and then move a controller on your MIDI device. `midid` returns the control number specific to the controller you moved and the device name of the MIDI control surface.

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

3. Create Listener for Control Change Messages

Use the `midicontrols` function to create an object that listens for Control Change messages and caches the most recent values corresponding to specified controllers. When you create a `midicontrols` object, you specify a MIDI control surface by its device name and specific controllers on the surface by their associated control numbers. Because the `midicontrols` object cannot query the MIDI control surface for initial values, consider setting initial values when creating the object.



View Example

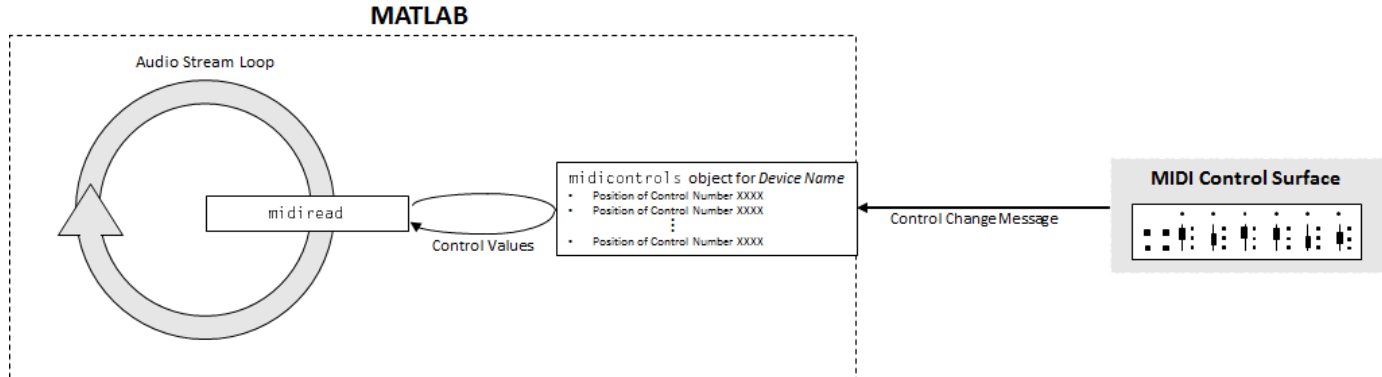
Identify two control numbers on your MIDI control surface. Choose initial control values for the controls you identified. Create a `midicontrols` object that listens to Control Change messages that arrive from the controllers you identified on the device you identified. When you create your `midicontrols` object, also specify initial control values. These initial control values work as default values until a Control Change message is received.

```
controlNum1 = midid;
[controlNum2,deviceName] = midid;
initialControlValues = [0.1,0.9];

midicontrolsObject = midicontrols([controlNum1,controlNum2], ...
    initialControlValues, ...
    'MIDI Device',deviceName);
```

4. Get Current Control Values

Use the `midiread` function to query your `midicontrols` object for current control values. `midiread` returns a matrix with values corresponding to all controllers the `midicontrols` object is listening to. Generally, you want to place `midiread` in an audio stream loop for continuous updating.



View Example

Place `midiread` in an audio stream loop to return the current control value of a specified controller. Use the control value to apply gain to an audio signal.

```
[controlNumber, deviceName] = midiid;
initialControlValue = 1;
midicontrolsObject = midicontrols(controlNumber, initialControlValue, 'MIDI Device', deviceName);

% 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);
```

5. Synchronize Bidirectional MIDI Control Surfaces

You can use `midisync` to send Control Change messages to your MIDI control surface. If the MIDI control surface is bidirectional, it adjusts the specified controllers. One important use of `midisync` is to set the controller positions on your MIDI control surface to initial values.

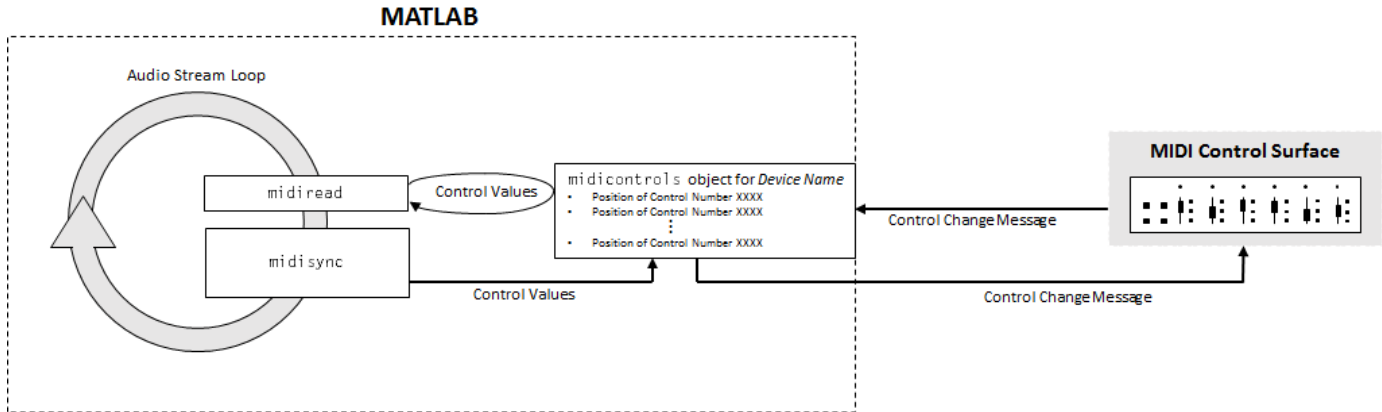
View Example

In this example, you initialize a controller on your MIDI control surface to a specified position. Calling `midisync(midicontrolsObject)` sends a Control Change message to your MIDI control surface, using the initial control values specified when you created the `midicontrols` object.

```
[controlNumber, deviceName] = midiid;
initialControlValue = 0.5;
midicontrolsObject = midicontrols(controlNumber, initialControlValue, 'MIDI Device', deviceName);

midisync(midicontrolsObject);
```

Another important use of `midisync` is to update your MIDI control surface if control values are adjusted in an audio stream loop. In this case, you call `midisync` with both your `midicontrols` object and the updated control values.



View Example

In this example, you check the normalized output volume in an audio stream loop. If the volume is above a given threshold, `midisync` is called and the MIDI controller that controls the applied gain is reduced.

```
[controlNumber, deviceName] = midiid;
initialControlValue = 0.5;
midicontrolsObject = midicontrols(controlNumber,initialControlValue);
fileReader = dsp.AudioFileReader('Ambiance-16-44p1-mono-12secs.wav');
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);

% Synchronize specified initial value with the MIDI control surface.
midisync(midicontrolsObject);

while ~isDone(fileReader)
    audioData = step(fileReader);
    controlValue = midiread(midicontrolsObject);
    gain = controlValue*2;
    audioDataWithGain = audioData*gain;

    % Check if max output is above a given threshold.
    if max(audioDataWithGain) > 0.7

        % Force new control value to be nonnegative.
        newControlValue = max(0,controlValue-0.5);

        % Send a Control Change message to the MIDI control surface.
        midisync(midicontrolsObject,newControlValue)
    end

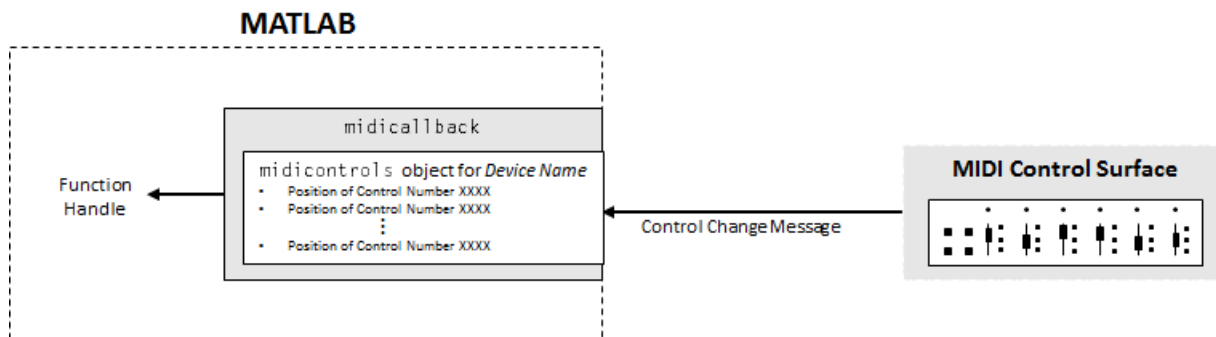
    play(deviceWriter,audioDataWithGain);
end

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


`midisync` is also a powerful tool in systems that also involve user interfaces (UIs), so that when one control is changed, the other control tracks it. Typically, you implement such tracking by setting callback functions on both the `midicontrols` object (using `midicallback`) and the UI control. The callback for the `midicontrols` object sends new values to the UI control. The UI uses `midisync` to send new values to the `midicontrols` object and MIDI control surface. See `midisync` for examples.

Alternative to Stream Processing

You can use `midicallback` as an alternative to placing `midiread` in an audio stream loop. If a `midicontrols` object receives a Control Change message, `midicallback` automatically calls a specified function handle. The callback function typically calls `midiread` to determine the new value of the MIDI controls. You can use this callback when you want a MIDI controller to trigger an action, such as updating a UI. Using this approach prevents having a MATLAB program continuously running in the command window.



Set Default MIDI Device

You can set the default MIDI device in the MATLAB environment by using the `setpref` function. Use `midiid` to determine the name of the device, and then use `setpref` to set the preference.

```
[~,deviceName] = midiid
```

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

```
deviceName =
```

```
BCF2000
```

```
setpref('midi','DefaultDevice',deviceName)
```

This preference persists across MATLAB sessions, so you only have to set it once, unless you want to change devices.

If you do not set this preference, MATLAB and the host operating system choose a device for you. However, such autoselection can cause unpredictable results because many computers have "virtual" (software) MIDI devices installed that you might not be aware of. For predictable behavior, set the preference.

Control Numbers

MATLAB defines 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

Blocks

MIDI Controls

Functions

`midicallback` | `midicontrols` | `midiid` | `midiread` | `midisync`

More About

- “What Are DAWs, Audio Plugins, and MIDI Controllers?”
- “Real-Time Audio in MATLAB”
- “MIDI Device Interface” on page 7-2
- “MIDI Control for Audio Plugins” on page 9-2

External Websites

- <https://www.midi.org>

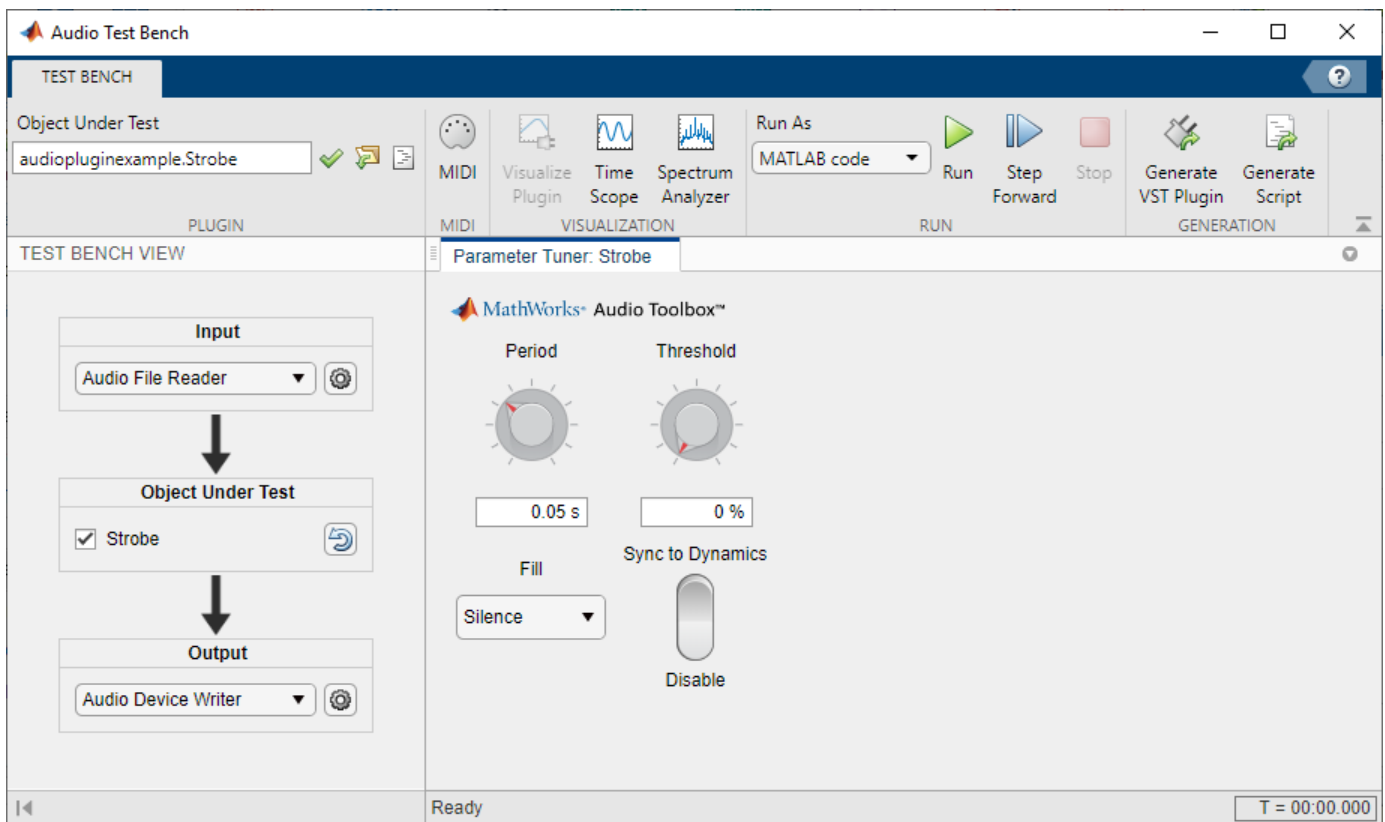
Use the Audio Test Bench

Audio Test Bench Walkthrough

In this tutorial, explore key functionality of the **Audio Test Bench**. The **Audio Test Bench** app enables you to debug, visualize, and configure audio plugins.

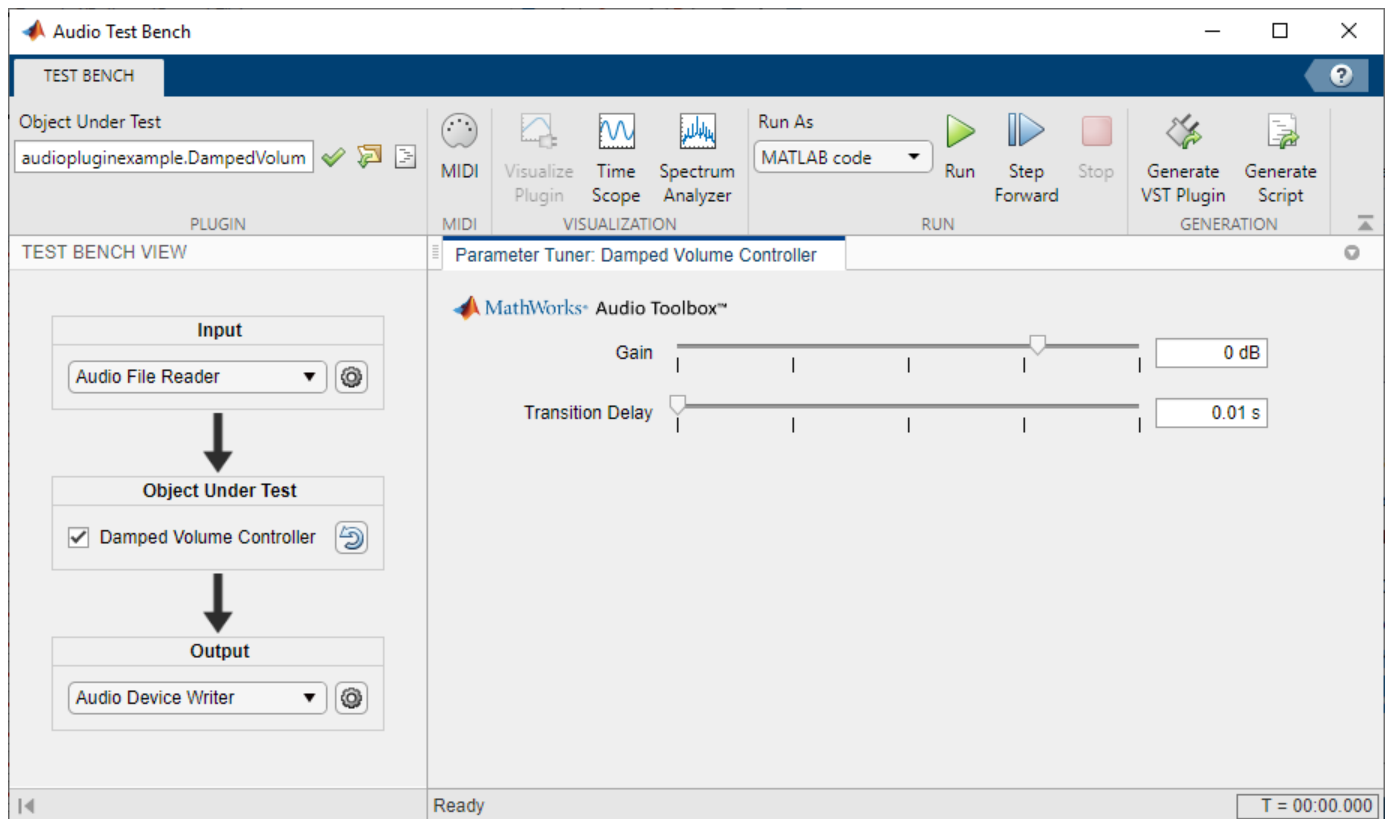
Choose Object Under Test

- 1 To open the **Audio Test Bench**, at the MATLAB command prompt, enter:
`audioTestBench`
- 2 In the **Object Under Test** box, enter `audiopluginexample.Strobe` and press **Enter**. The **Audio Test Bench** automatically displays the tunable parameters of the `audiopluginexample.Strobe` audio plugin.



The mapping between the tunable parameters of your object and the UI widgets on the **Audio Test Bench** is determined by `audioPluginInterface` and `audioPluginParameter` in the class definition of your object.

- 3 In the **Object Under Test** box, enter `audiopluginexample.DampedVolumeController` and press **Enter**. The **Audio Test Bench** automatically displays the tunable parameters of the `audiopluginexample.DampedVolumeController` audio plugin.



Run Audio Test Bench

To run the **Audio Test Bench** for your plugin with default settings, click . Move the sliders to modify the **Gain** and **Transition Delay** parameters while streaming.

To stop the audio stream loop, click . The MATLAB command line and objects used by the test bench are now released.

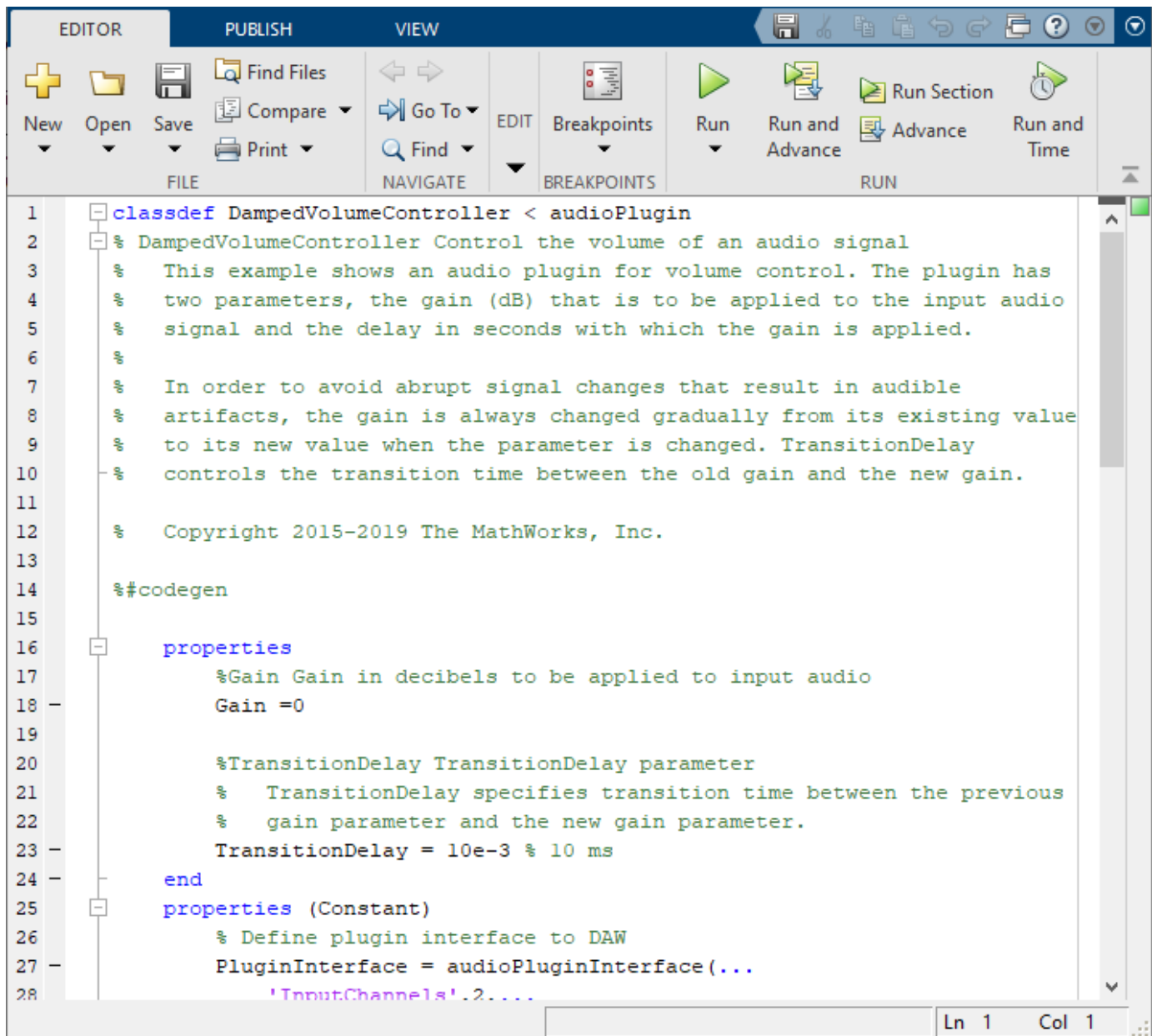
To reset internal states of your audio plugin and return the sliders to their initial positions, click .


Click to run the **Audio Test Bench** again.

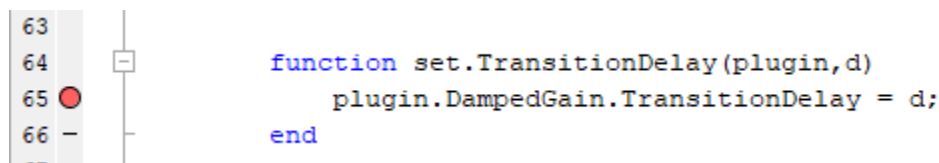
Debug Source Code of Audio Plugin

To pause the **Audio Test Bench**, click .

To open the source file of your audio plugin, click .





You can inspect the source code of your audio plugin, set breakpoints on it, and modify the code. Set a breakpoint at line 65 and then click  on the **Audio Test Bench**.

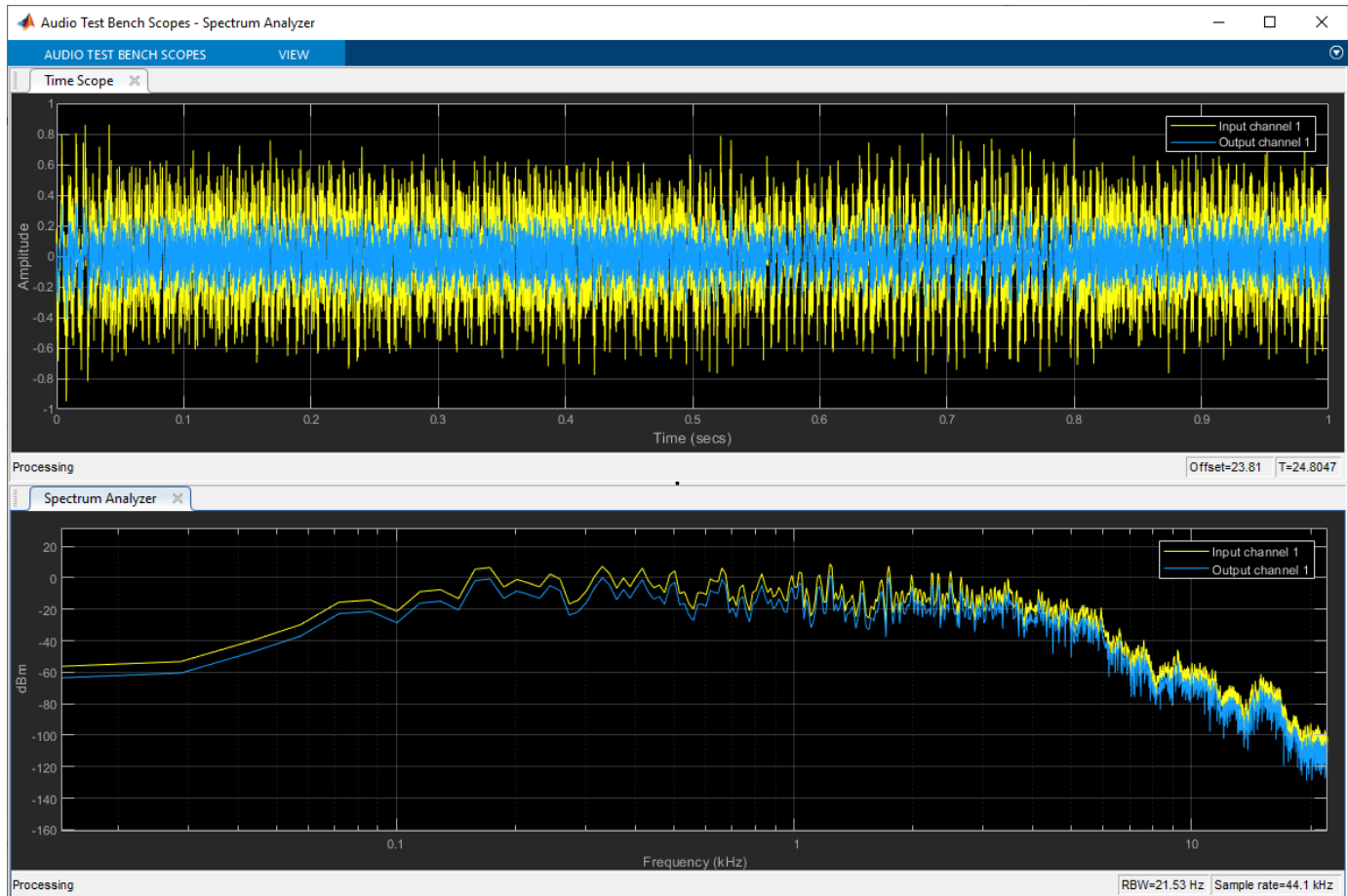



The **Audio Test Bench** runs your plugin until it reaches the breakpoint. To reach the breakpoint, move the **Transition Delay** slider. To quit debugging, remove the breakpoint. In the MATLAB editor, click **Quit Debugging**.

Open Scopes

To open a time scope to visualize the time-domain input and output for your audio plugin, click .

To open a spectrum analyzer to visualize the frequency-domain input and output, click .




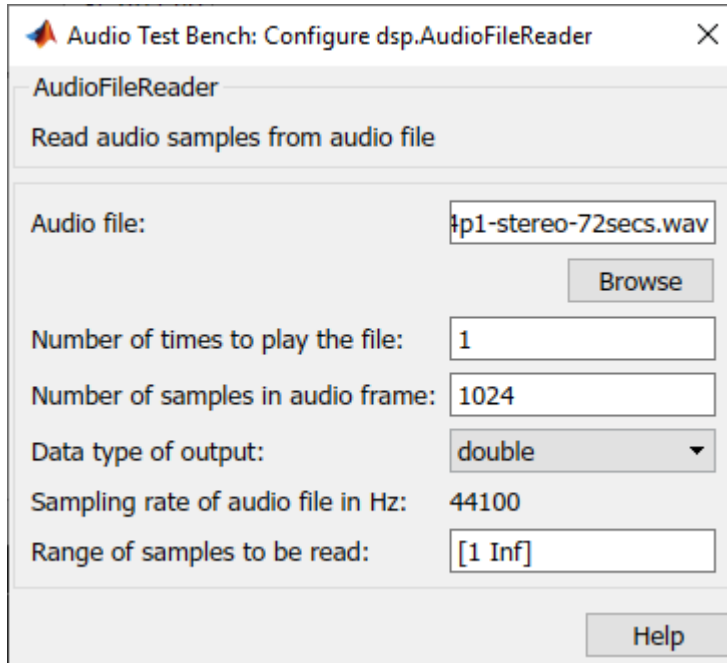
To release objects and stop the audio stream loop, click .

Configure Input to Audio Test Bench

The **Input** list contains these options:


- Audio File Reader -- `dsp.AudioFileReader`
- Audio Device Reader -- `audioDeviceReader`
- Audio Oscillator -- `audioOscillator`
- Wavetable Synthesizer -- `wavetableSynthesizer`
- Chirp Signal -- `dsp.Chirp`
- Colored Noise -- `dsp.ColoredNoise`

- 1 Select Audio File Reader.
- 2 Click  to open a dialog for Audio File Reader configuration.



You can enter any file name included on the MATLAB path. To specify a file that is not on the MATLAB path, specify the full file path.

- 3 In the **Audio file** box, enter: RockDrums-44p1-stereo-11secs.mp3


Press **Enter**, and then exit the Audio File Reader configuration dialog. To run the audio test bench with your new input, click .

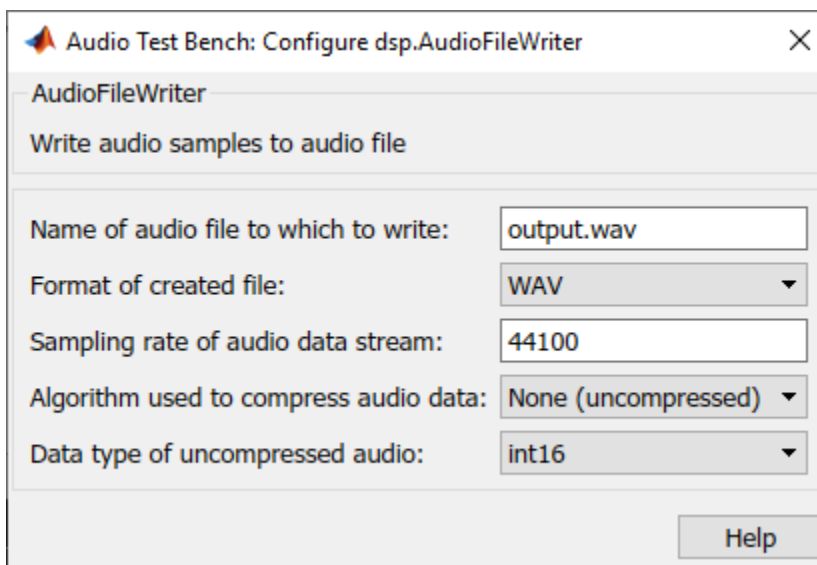
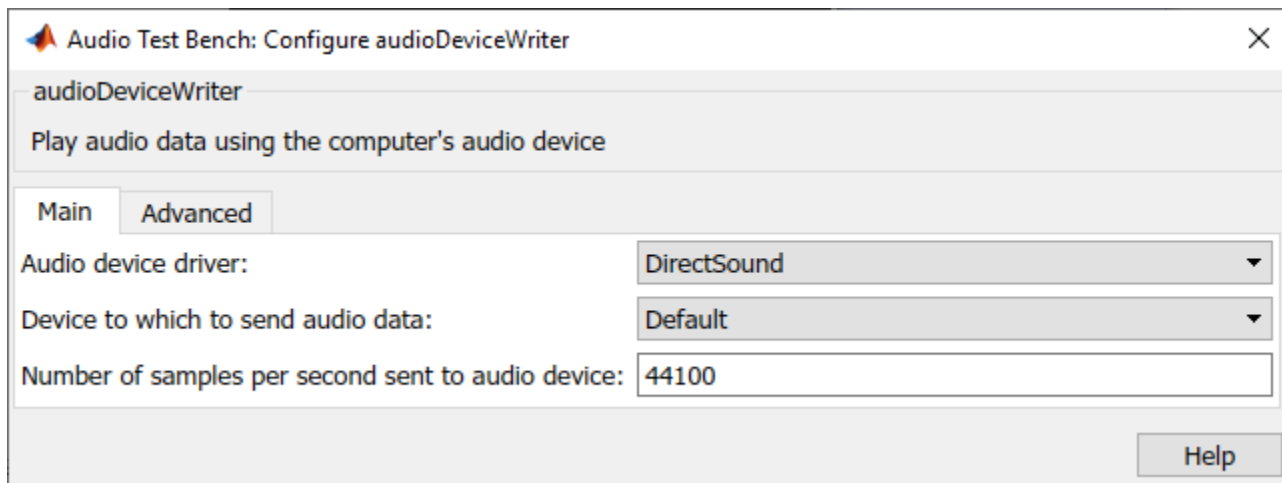
To release your output object and stop the audio stream loop, click .

Configure Output from Audio Test Bench


The **Output** list contains these options:


- Audio Device Writer -- audioDeviceWriter
- Audio File Writer -- dsp.AudioFileWriter
- Both -- audioDeviceWriter and dsp.AudioFileWriter
- None -- The audio signal is not routed to a file or device. Use this option if you are only interested in using the visualization and tuning capabilities of the test bench.

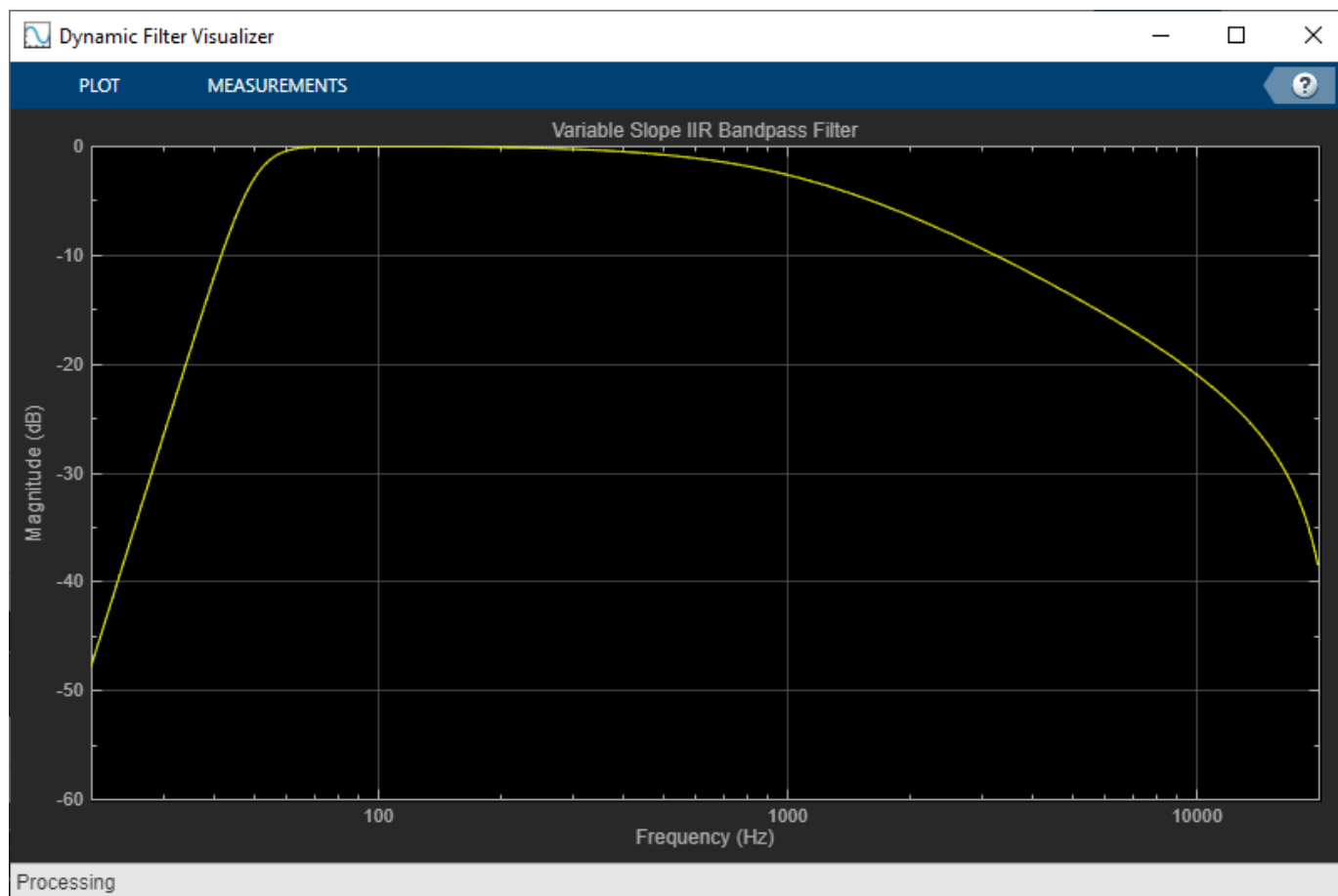
- 1 Choose to output to device and file by selecting Both from the **Output** menu.
- 2 To open a dialog for Audio Device Writer and Audio File Writer configuration, click .



Call Custom Visualization of Audio Plugin


If your audio plugin has a custom visualization method, the  button appears on the **Audio Test Bench**. In the **Object Under Test** box, enter `audiopluginexample.VarSlopeBandpassFilter` and press **Enter**. To open the custom visualization of

`audiopluginexample.VarSlopeBandpassFilter`, click . The custom visualization plots the frequency response of the filter. Tune the plugin parameters and observe the plot update in real time.




Custom visualizations are MATLAB-only features. Custom visualizations are not available for generated plugins.

Synchronize Plugin Property with MIDI Control

If you have a MIDI device connected to your computer, you can synchronize plugin properties with MIDI controls. To open a MIDI configuration UI, click . Synchronize the LowCutoff and HighCutoff properties with MIDI controls you choose. Click **OK**.

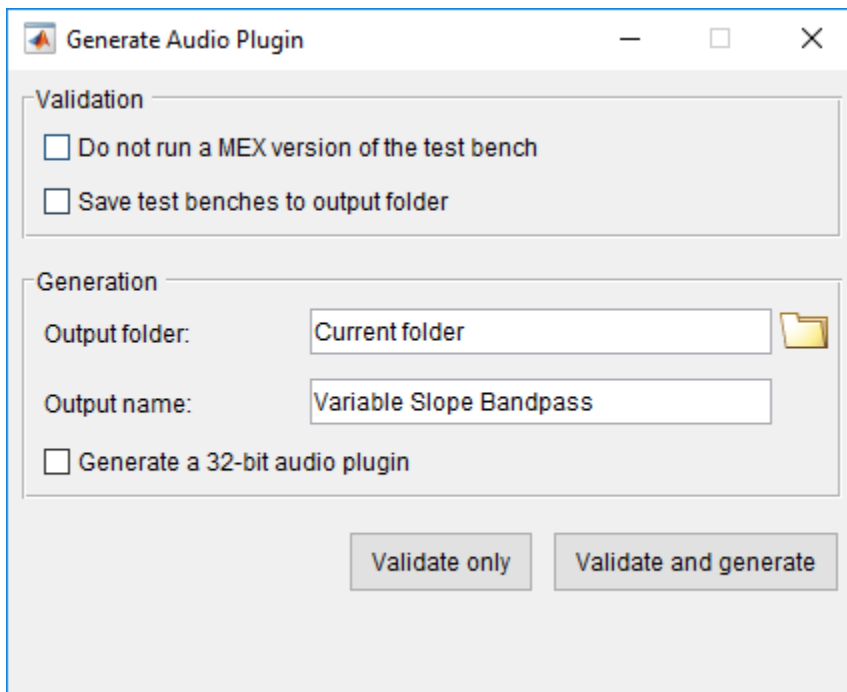
See `configureMIDI` for more information.

Play the Audio and Save the Output File

To run your audio plugin, click . Adjust your plugin properties in real time using your synchronized MIDI controls and sliders. Your processed audio file is saved to the current folder.


Validate and Generate Audio Plugin

To open the validation and generation dialog box, click .



You can validate only, or validate and generate your MATLAB audio plugin code in VST 2 plugin format. The **Generate a 32-bit audio plugin** check box is available only on win64 machines. See `validateAudioPlugin` and `generateAudioPlugin` for more information.

Generate MATLAB Script

To generate a MATLAB script that implements a test bench for your audio plugin, click .

```

1 % Test bench script for 'audiopluginexample.VarSlopeBandpassFilter'.
2 % Generated by Audio Test Bench on 31-Oct-2019 11:16:58 -0400.
3
4 % Create test bench input and output(s)
5 fileReader = dsp.AudioFileReader('Filename','C:\Program Files\MATLAB\R2020a
6 deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
7 fileWriter = dsp.AudioFileWriter;
8
9 % Create scopes
10 timeScope = dsp.TimeScope('SampleRate',44100, ...
11     'TimeSpan',1, ...
12     'TimeSpanOverrunAction','Scroll', ...
13     'AxesScaling','Manual', ...
14     'BufferLength',176400, ...
15     'ShowLegend',true, ...
16     'ChannelNames',{'Input channel 1','Output channel 1'}, ...
17     'ShowGrid',true, ...
18     'YLimits',[-1 1]);
19 specScope = dsp.SpectrumAnalyzer('SampleRate',44100, ...
20     'PlotAsTwoSidedSpectrum',false, ...
21     'FrequencyScale','Log', ...
22     'ShowLegend',true, ...
23     'ChannelNames',{'Input channel 1','Output channel 1'}, ...
24     'YLimits',[-147.68803417236001 6.3222170053428961]);
25
26 % Set up the system under test
27 sut = audiopluginexample.VarSlopeBandpassFilter;
28 setSampleRate(sut,fileReader.SampleRate);
29 sut.LowCutoff = 50.1189;
30 sut.HighCutoff = 1091.33;
31 sut.LowSlope = '36';
32 sut.HighSlope = '6';
33
34 % Open visualizer
35 visualize(sut);
36
37 % Open parameterTuner for interactive tuning during simulation
38 tuner = parameterTuner(sut);
39
40 % Stream processing loop
41 nUnderruns = 0;
42 while ~isDone(fileReader)
43     % Read from input, process, and write to outputs
44     in = fileReader();
45     out = sut(in);
46     nUnderruns = nUnderruns + deviceWriter(out);
47     fileWriter(out);
48
49     % Visualize input and output data in scopes
50     timeScope([in(:,1),out(:,1)]);
51     specScope([in(:,1),out(:,1)]);
52
53     % Process parameterTuner callbacks
54     drawnow limitrate
55 end

```

You can modify the code for complete control over the test bench environment, including the ability to create processing chains by placing plugins in cascade.

See Also

Audio Test Bench | `audioPlugin` | `generateAudioPlugin` | `validateAudioPlugin`

More About

- “Audio Plugins in MATLAB”
- “Audio Plugin Example Gallery” on page 12-2
- “Export a MATLAB Plugin to a DAW”

Audio Plugin Example Gallery

Audio Plugin Example Gallery

Use these Audio Toolbox plugin examples as building blocks in larger systems, as models for design patterns, or as benchmarks for comparison. Search the plugin descriptions to find an example that meets your needs.

Audio Effects

Filters

Gain Control

Spatial Audio

Communicate Between MATLAB and DAW

Music Information Retrieval

Speech Processing

Audio Plugin Examples

For a list of available audio plugins, see the online documentation.
--

See Also

Audio Test Bench | `audioPlugin` | `audioPluginInterface` | `audioPluginParameter` | `audioPluginSource`

More About

- “Audio Test Bench Walkthrough” on page 11-2
- “Audio Plugins in MATLAB”

Equalization

Equalization

Equalization (EQ) is the process of weighting the frequency spectrum of an audio signal.

You can use equalization to:

- Enhance audio recordings
- Analyze spectral content

Types of equalization include:

- Lowpass and highpass filters -- Attenuate high frequency and low frequency content, respectively.
- Low-shelf and high-shelf equalizers -- Boost or cut frequencies equally above or below a desired cutoff point.
- Parametric equalizers -- Selectively boost or cut frequency bands. Also known as peaking filters.
- Graphic equalizers -- Selectively boost or cut octave or fractional octave frequency bands. The bands have standards-based center frequencies. Graphic equalizers are a special case of parametric equalizers.

This tutorial describes how Audio Toolbox implements the design functions: `designParamEQ`, `designShelvingEQ`, and `designVarSlopeFilter`. The `multibandParametricEQ` System object combines the filter design functions into a multiband parametric equalizer. The `graphicEQ` System object combines the filter design functions and the `octaveFilter` System object for standards-based graphic equalization. For a tutorial focused on using the design functions in MATLAB, see “Parametric Equalizer Design” on page 1-406.

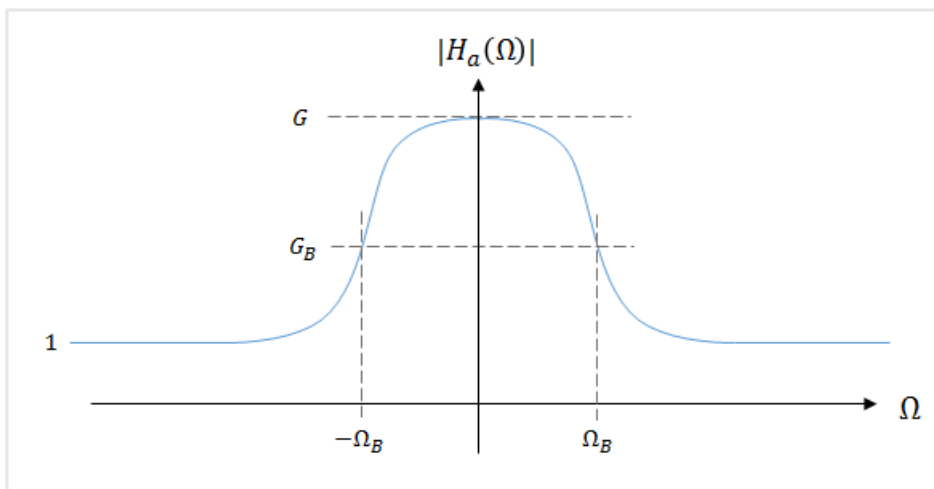
Equalization Design Using Audio Toolbox

EQ Filter Design

Audio Toolbox design functions use the bilinear transform method of digital filter design to determine your equalizer coefficients. In the bilinear transform method, you:

- 1 Choose an analog prototype.
- 2 Specify filter design parameters.
- 3 Perform the bilinear transformation.

Analog Low-Shelf Prototype



Audio Toolbox uses the high-order parametric equalizer design presented in [1]. In this design method, the analog prototype is taken to be a low-shelf Butterworth filter:

$$H_a(s) = \left[\frac{g\beta + s}{\beta + s} \right]^r \prod_{i=1}^L \left[\frac{g^2\beta^2 + 2gs_i\beta s + s^2}{\beta^2 + 2s_i\beta s + s^2} \right]$$

- L = Number of analog SOS sections
- N = Analog filter order
- $r = \begin{cases} 0 & N \text{ even} \\ 1 & N \text{ odd} \end{cases}$
- $g = G^{1/N}$
- $\beta = \Omega_B \times \left(\sqrt{\frac{G^2 - G_B^2}{G_B^2 - 1}} \right)^{-1/N} = \tan\left(\pi \frac{\Delta\omega}{2}\right) \times \left(\sqrt{\frac{G^2 - G_B^2}{G_B^2 - 1}} \right)^{-1/N}$, where $\Delta\omega$ is the desired digital bandwidth
- $s_i = \sin\left(\frac{(2i-1)\pi}{2N}\right)$, $i = 1, 2, \dots, L$

For parametric equalizers, the analog prototype is reduced by setting the bandwidth gain to the square root of the peak gain ($G_B = \text{sqrt}(G)$).

After the design parameters are specified, the analog prototype is transformed directly to the desired digital equalizer by a bandpass bilinear transformation:

$$s = \frac{1 - 2\cos(\omega_0)z^{-1} + z^{-2}}{1 - z^{-2}}$$

ω_0 is the desired digital center frequency.

This transformation doubles the filter order. Every first-order analog section becomes a second-order digital section. Every second-order analog section becomes a fourth-order digital section. Audio Toolbox always calculates fourth-order digital sections, which means that returning second-order sections requires the computation of roots, and is less efficient.

Digital Coefficients

The digital transfer function is implemented as a cascade of second-order and fourth-order sections.

$$H(z) = \left[\frac{b_{00} + b_{01}z^{-1} + b_{02}z^{-2}}{1 + a_{01}z^{-1} + a_{02}z^{-2}} \right]^r \prod_{i=1}^L \left[\frac{b_{i0} + b_{i1}z^{-1} + b_{i2}z^{-2} + b_{i3}z^{-3} + b_{i4}z^{-4}}{1 + a_{i1}z^{-1} + a_{i2}z^{-2} + a_{i3}z^{-3} + a_{i4}z^{-4}} \right]$$

The coefficients are given by performing the bandpass bilinear transformation on the analog prototype design.

Second-Order Section Coefficients	Fourth-Order Section Coefficients
$D_0 = \beta + 1$ $b_{00} = (1 + g\beta)/D_0$ $b_{01} = -2\cos(\omega_0)/D_0$ $b_{02} = (1 - g\beta)/D_0$ $a_{01} = -2\cos(\omega_0)/D_0$ $a_{02} = (1 - \beta)/D_0$	$D_i = \beta^2 + 2s_i\beta + 1$ $b_{i0} = (g^2\beta^2 + 2gs_i\beta + 1)/D_i$ $b_{i1} = -4c_0(1 + gs_i\beta)/D_i$ $b_{i2} = 2(1 + 2\cos^2(\omega_0) - g^2\beta^2)/D_i$ $b_{i3} = -4c_0(1 - gs_i\beta)/D_i$ $b_{i4} = (g^2\beta^2 - 2gs_i\beta + 1)/D_i$ $a_{i1} = -4c_0(1 + s_i\beta)/D_i$ $a_{i2} = 2(1 + 2\cos^2(\omega_0) - \beta^2)/D_i$ $a_{i3} = -4\cos(\omega_0)(1 - s_i\beta)/D_i$ $a_{i4} = (\beta^2 - 2s_i\beta + 1)/D_i$

Biquadratic Case

In the biquadratic case, when $N = 1$, the coefficients reduce to:

$$D_0 = \frac{\Omega_B}{\sqrt{G}} + 1$$

$$b_{00} = (1 + \Omega_B\sqrt{G})/D_0, \quad b_{01} = -2\cos(\omega_0)/D_0, \quad b_{02} = (1 - \Omega_B\sqrt{G})/D_0$$

$$a_{01} = -2\cos(\omega_0)/D_0, \quad a_{02} = \left(1 - \frac{\Omega_B}{\sqrt{G}}\right)/D_0$$

Denormalizing the a_{00} coefficient, and making substitutions of $A = \sqrt{G}$, $\Omega_B \cong \alpha$ yields the familiar peaking EQ coefficients described in [2].

Orfanidis notes the approximate equivalence of Ω_B and α in [1].

By using trigonometric identities,

$$\Omega_B = \tan\left(\frac{\Delta\omega}{2}\right) = \sin(\omega_0)\sinh\left(\frac{\ln 2}{2}B\right),$$

where B plays the role of an equivalent octave bandwidth.

Bristow-Johnson obtained an approximate solution for B in [4]:

$$B = \frac{\omega_0}{\sin \omega_0} \times BW$$

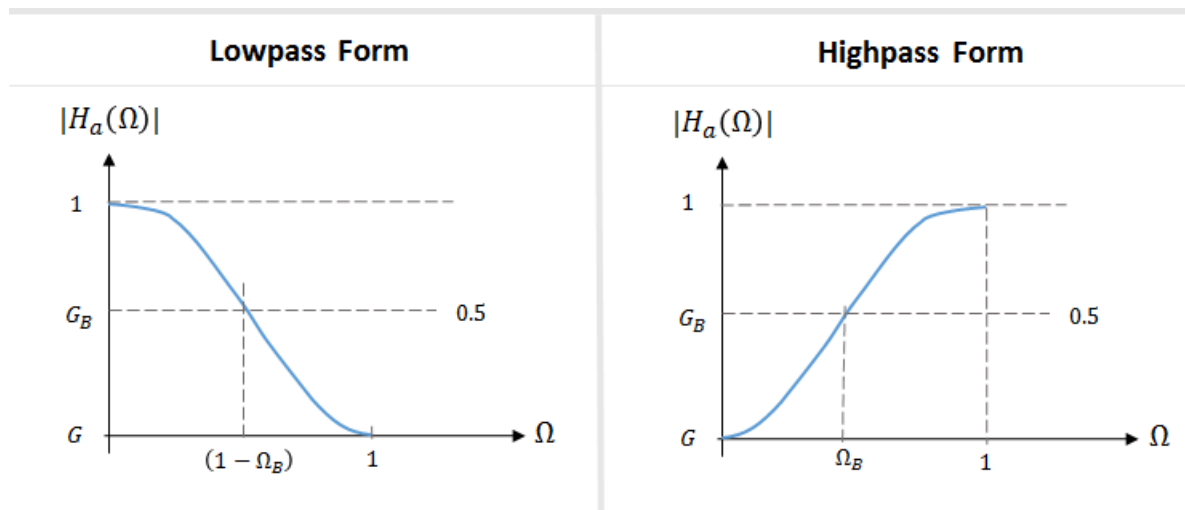
Substituting the approximation for B into the Ω_B equation yields the definition of α in [2]:

$$\alpha = \sin(\omega_0) \sinh\left(\frac{\ln 2}{2} \times \frac{\omega_0}{\sin \omega_0} \times BW\right)$$

Lowpass and Highpass Filter Design

Analog Low-Shelf Prototype

To design lowpass and highpass filters, Audio Toolbox uses a special case of the filter design for parametric equalizers. In this design, the peak gain, G , is set to 0, and G_B^2 is set to 0.5 (-3 dB cutoff). The cutoff frequency of the lowpass filter corresponds to $1 - \Omega_B$. The cutoff frequency of the highpass filter corresponds to Ω_B .



Digital Coefficients

The table summarizes the results of the bandpass bilinear transformation. The digital center frequency, ω_0 , is set to π for lowpass filters and 0 for highpass filters.

Second Order Section Coefficients	Fourth Order Section Coefficients
$D_0 = 1 + \tan\left(\pi\frac{\Delta\omega}{2}\right)$ $b_{00} = 1/D_0$ $b_{01} = -2\cos(\omega_0)/D_0$ $b_{02} = 1/D_0$ $a_{01} = -2\cos(\omega_0)/D_0$ $a_{02} = \left(1 - \tan\left(\pi\frac{\Delta\omega}{2}\right)\right)/D_0$	$D_i = \tan^2\left(\pi\frac{\Delta\omega}{2}\right) + 2s_i\tan\left(\pi\frac{\Delta\omega}{2}\right) + 1$ $b_{i0} = 1/D_i$ $b_{i1} = -4\cos(\omega_0)/D_i$ $b_{i2} = 2(1 + 2\cos^2(\omega_0))/D_i$ $b_{i3} = -4\cos(\omega_0)/D_i$ $b_{i4} = 1/D_i$ $a_{i1} = -4\cos(\omega_0)\left(1 + s_i\tan\left(\pi\frac{\Delta\omega}{2}\right)\right)/D_i$ $a_{i2} = 2\left(1 + 2\cos^2(\omega_0) - \tan^2\left(\pi\frac{\Delta\omega}{2}\right)\right)/D_i$ $a_{i3} = -4\cos(\omega_0)\left(1 - s_i\tan\left(\pi\frac{\Delta\omega}{2}\right)\right)/D_i$ $a_{i4} = \left(\tan^2\left(\pi\frac{\Delta\omega}{2}\right) - 2s_i\tan\left(\pi\frac{\Delta\omega}{2}\right) + 1\right)/D_i$

Shelving Filter Design

Analog Prototype

Audio Toolbox implements the shelving filter design presented in [2]. In this design, the high-shelf and low-shelf analog prototypes are presented separately:

$$H_L(s) = A \left(\frac{As^2 + (\sqrt{A}/Q)s + 1}{s^2 + (\sqrt{A}/Q)s + A} \right) \quad H_H(s) = A \left(\frac{s^2 + (\sqrt{A}/Q)s + A}{As^2 + (\sqrt{A}/Q)s + 1} \right)$$

For compactness, the analog filters are presented with variables A and Q . You can convert A and Q to available Audio Toolbox design parameters:

$$A = 10^{G/40}$$

$$\frac{1}{Q} = \sqrt{(A + 1/A)(1/slope - 1) + 2}$$

After you specify the design parameters, the analog prototype is transformed to the desired digital shelving filter by a bilinear transformation with prewarping:

$$s = \left(\frac{z-1}{z+1} \right) \times \left(\frac{1}{\tan\left(\frac{\omega_0}{2}\right)} \right)$$

Digital Coefficients

The table summarizes the results of the bilinear transformation with prewarping.

Low-Shelf	$b_0 = A((A + 1) - (A - 1)\cos(\omega_0) + 2\alpha\sqrt{A})$
	$b_1 = 2A((A - 1) - (A + 1)\cos(\omega_0))$
	$b_2 = A((A + 1) - (A - 1)\cos(\omega_0) - 2\alpha\sqrt{A})$
	$a_0 = (A + 1) + (A - 1)\cos(\omega_0) + 2\alpha\sqrt{A}$
	$a_1 = -2((A - 1) + (A + 1)\cos(\omega_0))$
	$a_2 = (A + 1) + (A - 1)\cos(\omega_0) - 2\alpha\sqrt{A}$
High-Shelf	$b_0 = A((A + 1) + (A - 1)\cos(\omega_0) + 2\alpha\sqrt{A})$
	$b_1 = -2A((A - 1) + (A + 1)\cos(\omega_0))$
	$b_2 = A((A + 1) + (A - 1)\cos(\omega_0) - 2\alpha\sqrt{A})$
	$a_0 = (A + 1) - (A - 1)\cos(\omega_0) + 2\alpha\sqrt{A}$
	$a_1 = 2((A - 1) + (A + 1)\cos(\omega_0))$
	$a_2 = (A + 1) - (A - 1)\cos(\omega_0) - 2\alpha\sqrt{A}$
Intermediate Variables	$\alpha = \frac{\sin(\omega_0)}{2} \sqrt{\left(A + \frac{1}{A}\right) \left(\frac{1}{slope} - 1\right)} + 2A$
	$\omega_0 = 2\pi \frac{Cutoff\ Frequency}{F_s}$

References

- [1] Orfanidis, Sophocles J. "High-Order Digital Parametric Equalizer Design." *Journal of the Audio Engineering Society*. Vol. 53, November 2005, pp. 1026-1046.
- [2] Bristow-Johnson, Robert. "Cookbook Formulae for Audio EQ Biquad Filter Coefficients." Accessed March 02, 2016. <http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt>.
- [3] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 2010.
- [4] Bristow-Johnson, Robert. "The Equivalence of Various Methods of Computing Biquad Coefficients for Audio Parametric Equalizers." Presented at the 97th Convention of the AES, San Francisco, November 1994, AES Preprint 3906.

See Also

designParamEQ | designShelvingEQ | designVarSlopeFilter | graphicEQ | multibandParametricEQ

More About

- "Parametric Equalizer Design" on page 1-406
- "Graphic Equalization" on page 1-168
- "Octave-Band and Fractional Octave-Band Filters" on page 1-420
- "Audio Weighting Filters" on page 1-178

Deployment

Desktop Real-Time Audio Acceleration with MATLAB Coder

This example shows how to accelerate a real-time audio application using C code generation with MATLAB® Coder™. You must have the MATLAB Coder™ software installed to run this example.

Introduction

Replacing parts of your MATLAB code with an automatically generated MATLAB executable (MEX-function) can speed up simulation. Using MATLAB Coder, you can generate readable and portable C code and compile it into a MEX-function that replaces the equivalent section of your MATLAB algorithm.

This example showcases code generation using an audio notch filtering application.

Notch Filtering

A notch filter is used to eliminate a specific frequency from a signal. Typical filter design parameters for notch filters are the notch center frequency and the 3 dB bandwidth. The center frequency is the frequency at which the filter has a linear gain of zero. The 3 dB bandwidth measures the frequency width of the notch of the filter computed at the half-power or 3 dB attenuation point.

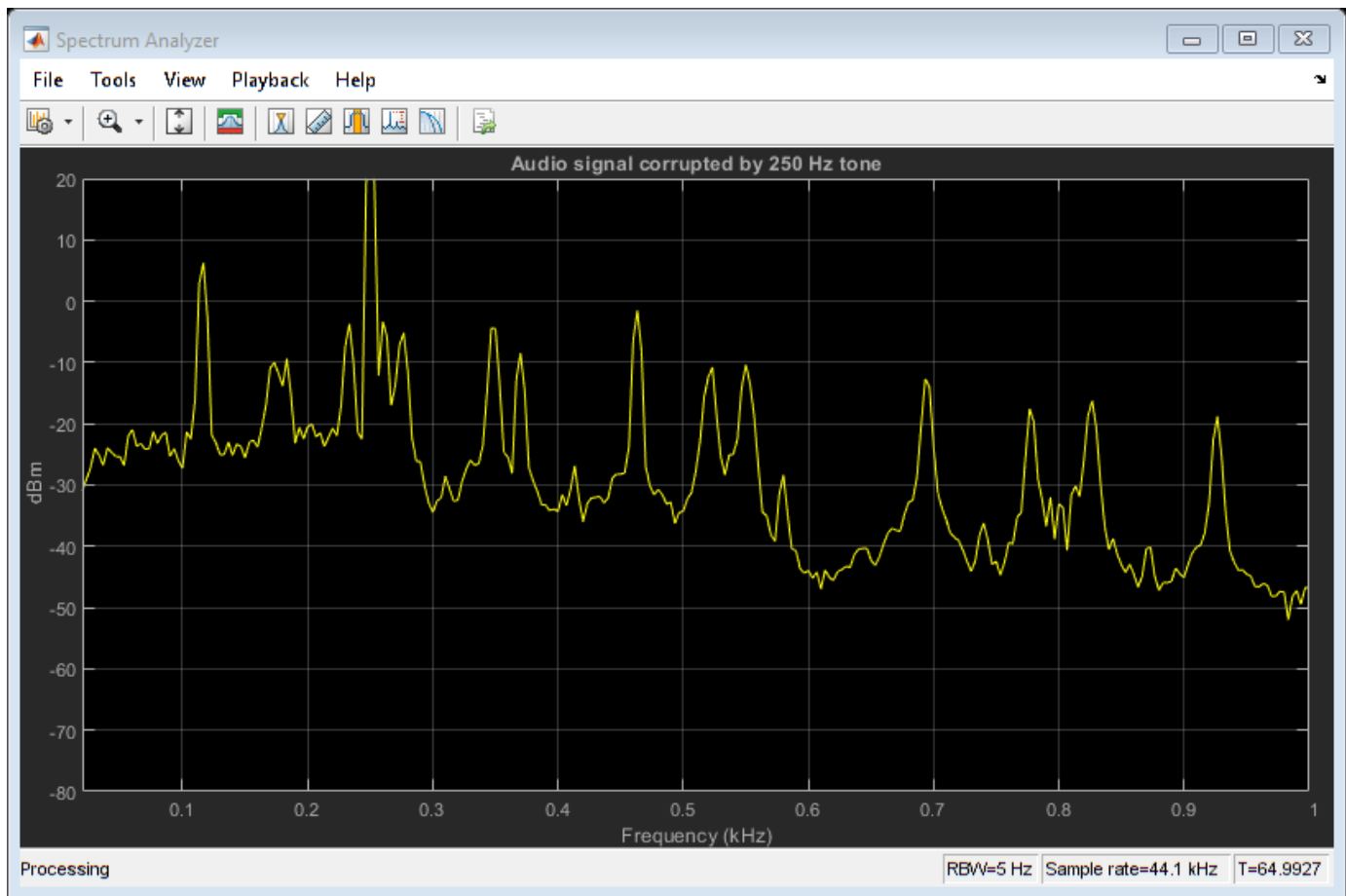
The helper function used in this example is `helperAudioToneRemoval`. The function reads an audio signal corrupted by a 250 Hz sinusoidal tone from a file. `helperAudioToneRemoval` uses a notch filter to remove the interfering tone and writes the filtered signal to a file.

You can visualize the corrupted audio signal using a spectrum analyzer.

```
reader = dsp.AudioFileReader('guitar_plus_tone.ogg');

scope = dsp.SpectrumAnalyzer('SampleRate',reader.SampleRate, ...
    'RBWSource','Property','RBW',5, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'SpectralAverages',10, ...
    'FrequencySpan','Start and stop frequencies', ...
    'StartFrequency',20, ...
    'StopFrequency',1000, ...
    'Title','Audio signal corrupted by 250 Hz tone');

while ~isDone(reader)
    audio = reader();
    scope(audio(:,1));
end
```



C Code Generation Speedup

Measure the time it takes to read the audio file, filter out the interfering tone, and write the filtered output using MATLAB code.

```
tic
helperAudioToneRemoval
t1 = toc;

fprintf('MATLAB Simulation Time: %d\n',t1)
MATLAB Simulation Time: 3.701829e+00
```

Next, generate a MEX-function from `helperAudioToneRemoval` using the MATLAB Coder function, `codegen` (MATLAB Coder).

```
codegen helperAudioToneRemoval
```

Measure the time it takes to execute the MEX-function and calculate the speedup gain with a compiled function.

```
tic
helperAudioToneRemoval_mex
t2 = toc;

fprintf('Code Generation Simulation Time: %d\n',t2)
```

```
Code Generation Simulation Time: 2.167587e+00
```

```
fprintf('Speedup factor: %6.2f\n',t1/t2)
```

```
Speedup factor: 1.71
```

See Also

Related Examples

- “Generate Standalone Executable for Parametric Audio Equalizer” on page 1-258
- “Deploy Audio Applications with MATLAB Compiler” on page 1-261

Audio I/O User Guide

Run Audio I/O Features Outside MATLAB and Simulink

You can deploy these audio input and output features outside the MATLAB and Simulink environments:

System Objects

- `audioPlayerRecorder`
- `audioDeviceReader`
- `audioDeviceWriter`
- `dsp.AudioFileReader`
- `dsp.AudioFileWriter`

Blocks

- Audio Device Reader
- Audio Device Writer
- From Multimedia File
- To Multimedia File

The generated code for the audio I/O features relies on prebuilt dynamic library files included with MATLAB. You must account for these extra files when you run audio I/O features outside the MATLAB and Simulink environments. To run a standalone executable generated from a model or code containing the audio I/O features, set your system environment using commands specific to your platform.

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

The path in these commands is valid only on systems that have MATLAB installed. If you run the standalone app on a machine with only MCR, and no MATLAB installed, replace `$MATLABROOT/bin/...` with the path to the MCR.

To run the code generated from the above System objects and blocks on a machine does not have MCR or MATLAB installed, use the `packNGo` function. The `packNGo` function packages all relevant

files in a compressed zip file so that you can relocate, unpack, and rebuild your project in another development environment with no MATLAB installed.

You can use the `packNGo` function at the command line or the **Package** option in the MATLAB Coder app. The files are packaged in a compressed file that you can relocate and unpack using a standard zip utility. For more details on how to pack the code generated from MATLAB code, see “Package Code for Other Development Environments” (MATLAB Coder). For more details on how to pack the code generated from Simulink blocks, see the `packNGo` function.

See Also

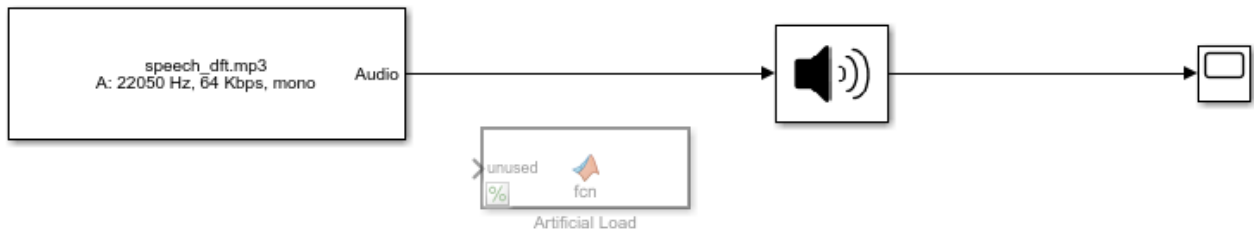
More About

- “MATLAB Programming for Code Generation” (MATLAB Coder)

Block Example Repository

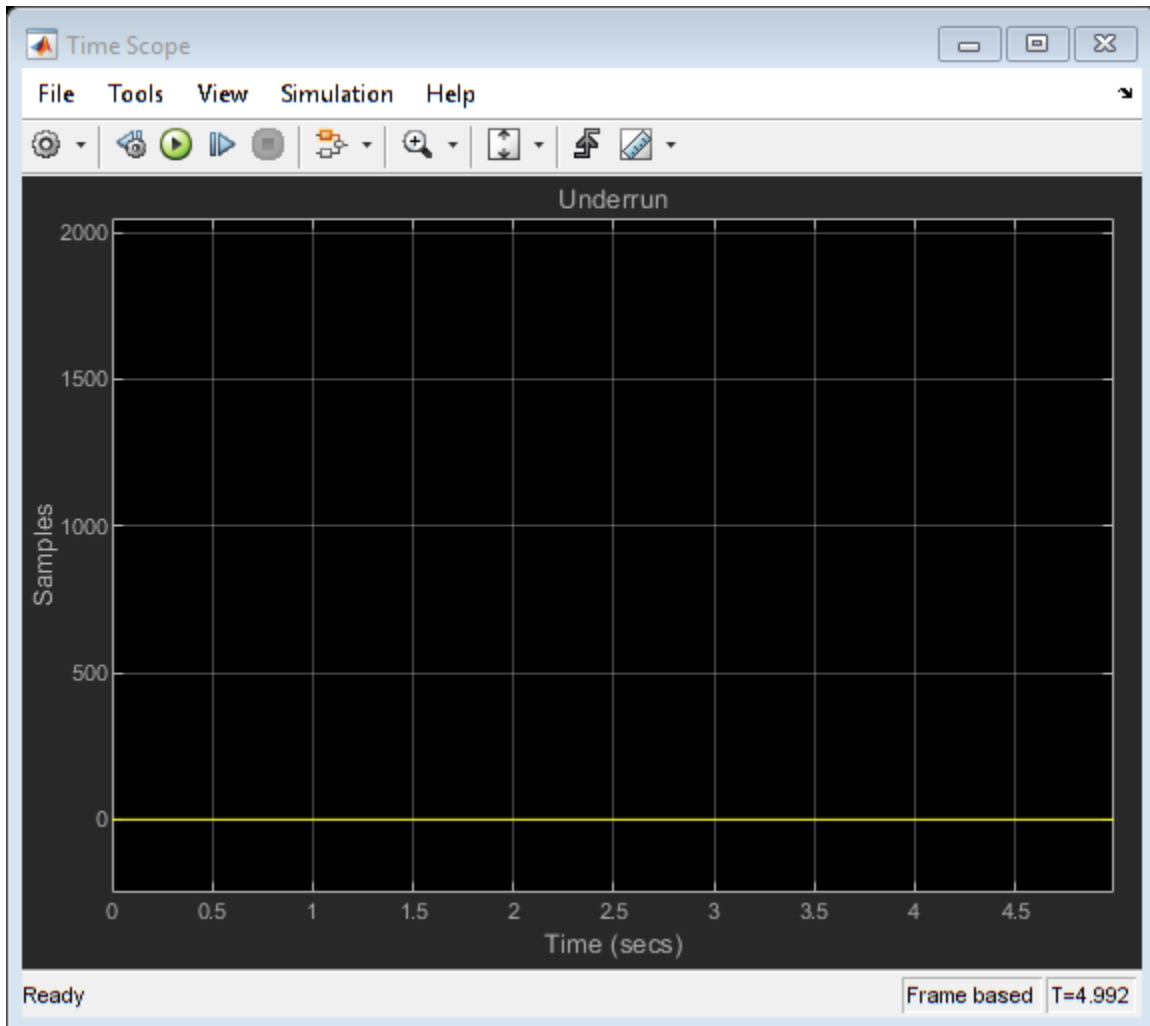
Decrease Underrun

Examine the Audio Device Writer block in a Simulink® model, determine underrun, and decrease underrun.



Copyright 2016 The MathWorks, Inc.

1. Run the model. The Audio Device Writer sends an audio stream to your computer's default audio output device. The Audio Device Writer block sends the number of samples underrun to your Time Scope.



2. Uncomment the Artificial Load block. This block performs computations that slow the simulation.
3. Run the model. If your device writer is dropping samples:
 - a. Stop the simulation.
 - b. Open the From Multimedia File block.
 - c. Set the **Samples per frame** parameter to 1024.
 - d. Close the block and run the simulation.

If your model continues to drop samples, increase the frame size again. The increased frame size increases the buffer size used by the sound card. A larger buffer size decreases the possibility of underruns at the cost of higher audio latency.

See Also

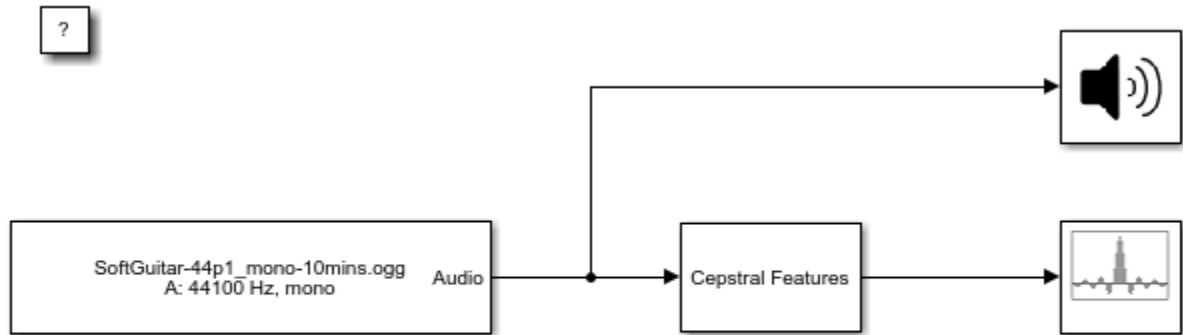
From Multimedia File | Time Scope

Block Example Repository

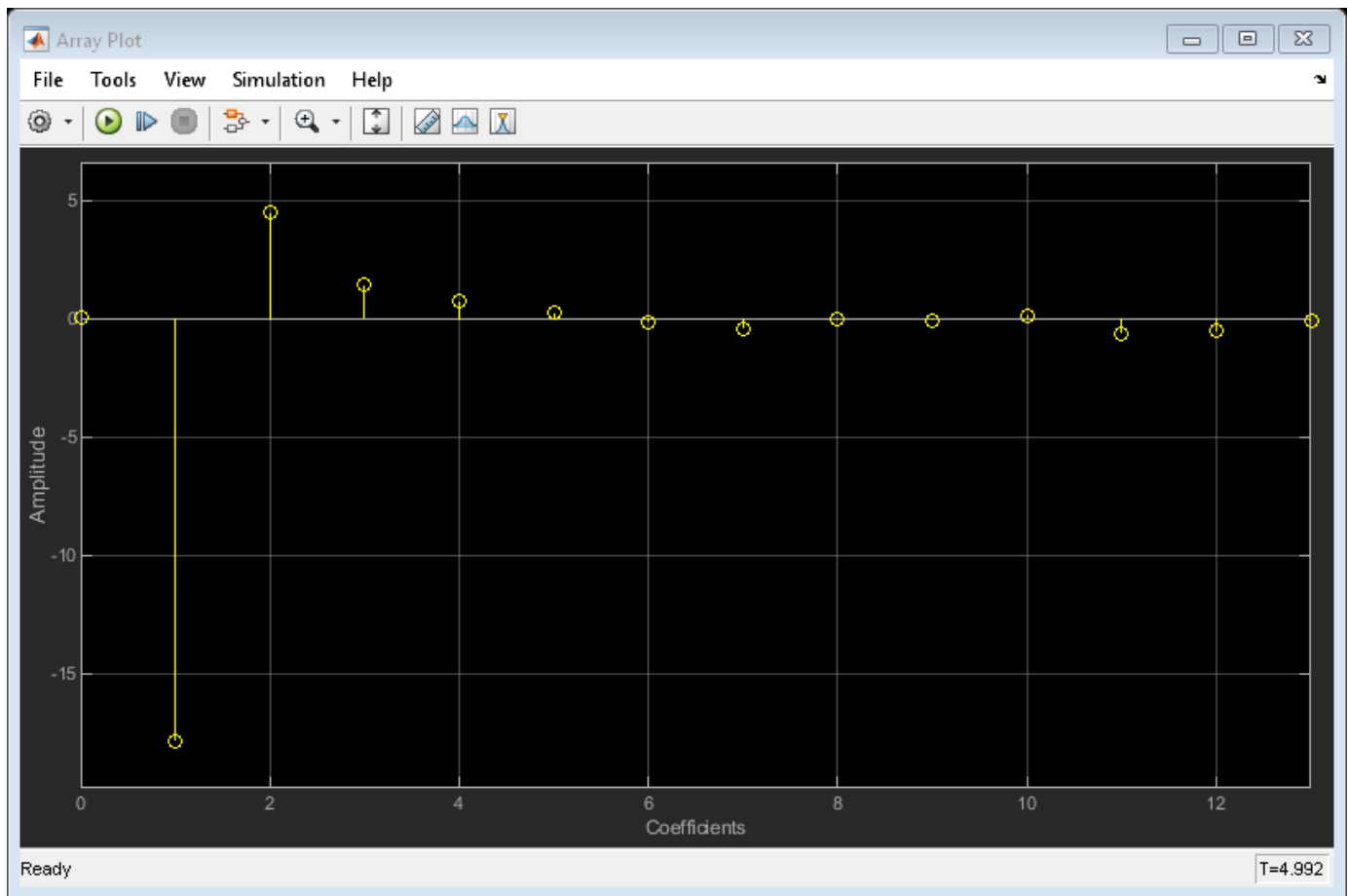
- “Extract Cepstral Coefficients” on page 17-2
- “Tune Center Frequency Using Input Port” on page 17-4
- “Gate Audio Signal Using VAD” on page 17-6
- “Frequency-Domain Voice Activity Detection” on page 17-8
- “Visualize Noise Power” on page 17-9
- “Detect Presence of Speech” on page 17-12
- “Perform Graphic Equalization” on page 17-14
- “Split-Band De-Essing” on page 17-16
- “Diminish Plosives from Speech” on page 17-17
- “Suppress Loud Sounds” on page 17-18
- “Attenuate Low-Level Noise” on page 17-20
- “Suppress Volume of Loud Sounds” on page 17-22
- “Gate Background Noise” on page 17-24
- “Output Values from MIDI Control Surface” on page 17-26
- “Apply Frequency Weighting” on page 17-28
- “Compare Loudness Before and After Audio Processing” on page 17-30
- “Two-Band Crossover Filtering for a Stereo Speaker System” on page 17-32
- “Mimic Acoustic Environments” on page 17-34
- “Perform Parametric Equalization” on page 17-35
- “Perform Octave Filtering” on page 17-37
- “Read from Microphone and Write to Speaker” on page 17-39
- “Channel Mapping” on page 17-41
- “Trigger Gain Control Based on Loudness Measurement” on page 17-42
- “Generate Variable-Frequency Tones in Simulink” on page 17-44
- “Trigger Reverberation Parameters” on page 17-47
- “Model Engine Noise” on page 17-48

Extract Cepstral Coefficients

Use the Cepstral Feature Extractor block to extract and visualize cepstral coefficients from an audio file.



Copyright 2018 The MathWorks, Inc.

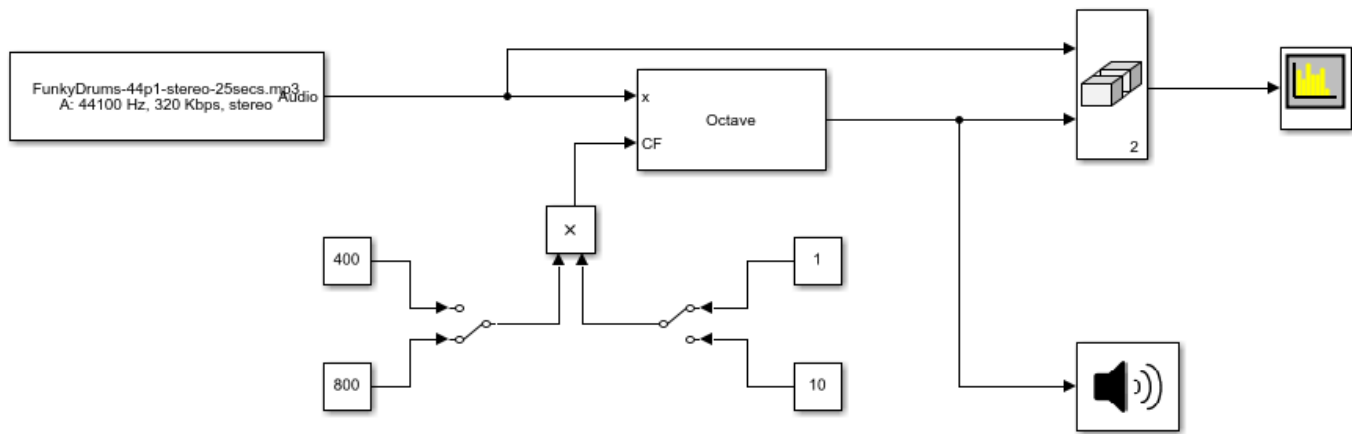


See Also

Array Plot | Audio Device Writer | Cepstral Feature Extractor | From Multimedia File | cepstralFeatureExtractor | gtcc | mfcc

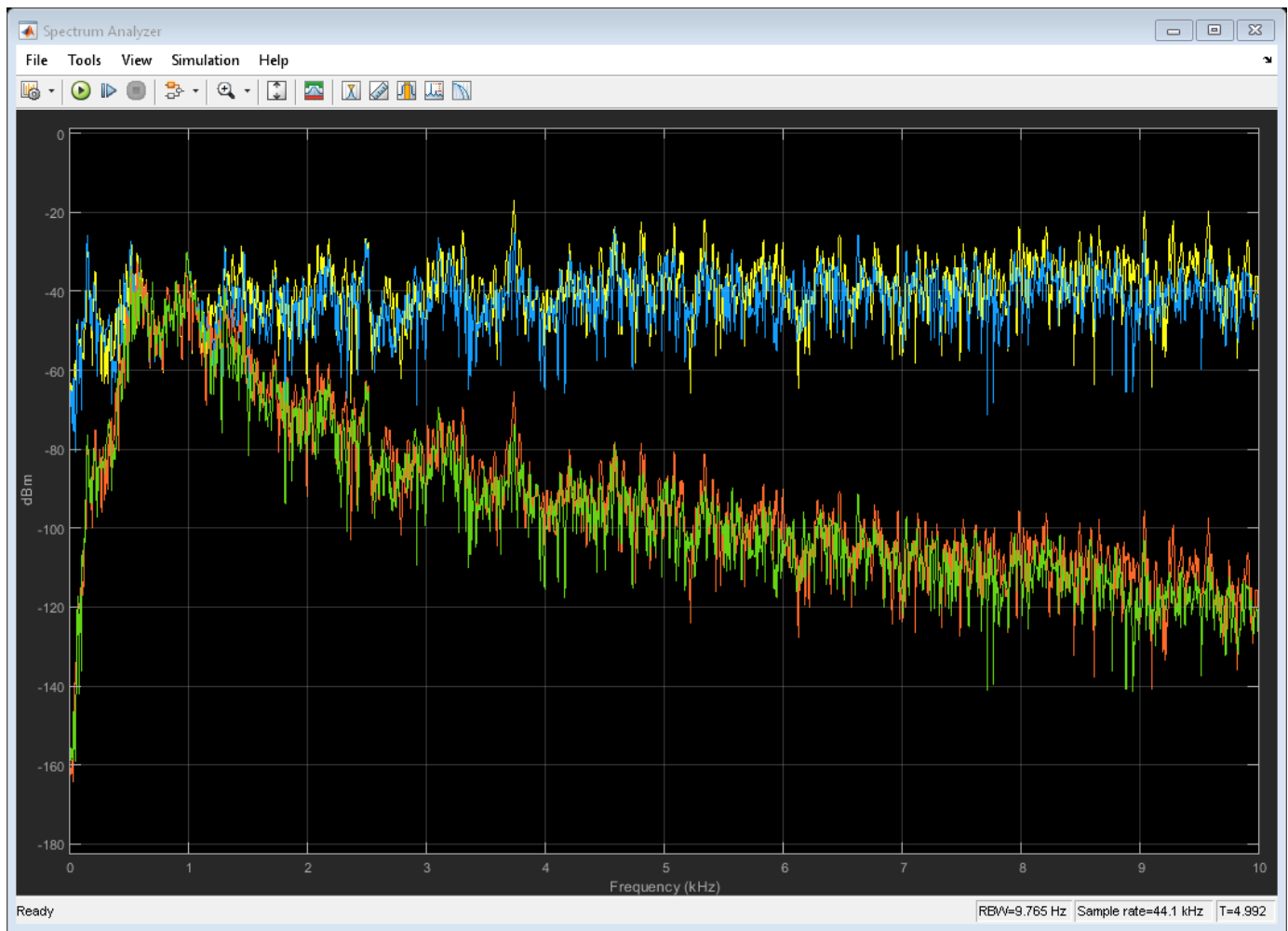
Tune Center Frequency Using Input Port

Tune the center frequency of an Octave Filter block in Simulink® using the optional input port.



Copyright 2018 The MathWorks Inc.

1. Run the simulation. The From Multimedia File block sends a stereo audio stream to the Octave Filter block. The center frequency of the Octave Filter block can be tuned using the manual switches routed into the optional input port. The filtered audio is sent to your computer's default audio device. The filtered audio and unfiltered audio are sent to a Spectrum Analyzer block for visualization.
2. Tune the center frequency by toggling manual switches routing constant values. The constant value routed from the left is multiplied with the constant value routed from the right. The center frequency of the Octave Filter block can be set at 400, 800, 4000, and 8000 Hz.
3. Observe the Spectrum Analyzer as you tune the center frequency. Note how the center frequency changes as you toggle the manual switches.



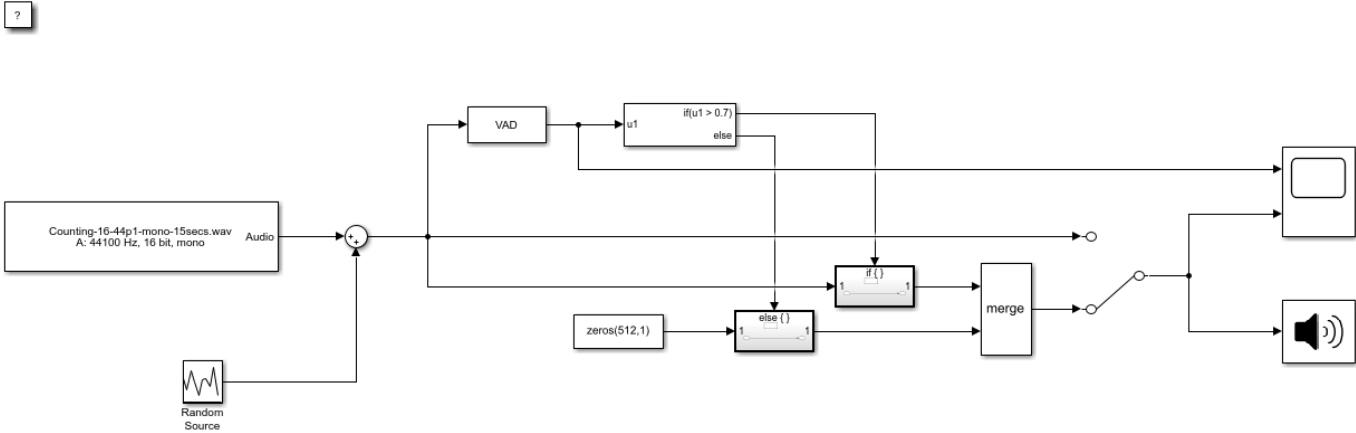
See Also

[Audio Device Writer](#) | [From Multimedia File](#) | [Manual Switch](#) | [Octave Filter](#) | [Time Scope](#)

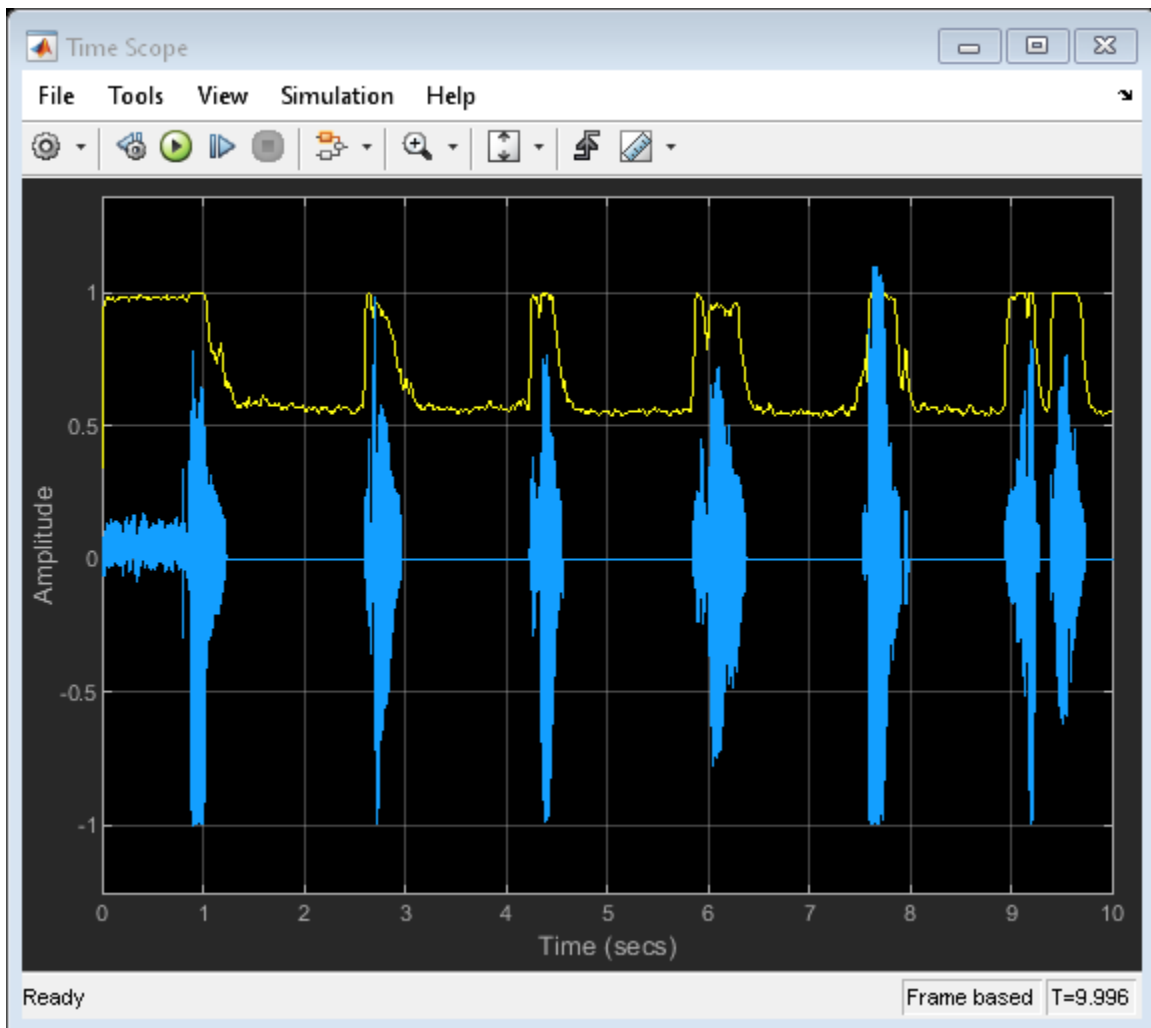
Gate Audio Signal Using VAD

This model uses if-else block signal routing to replace regions of no speech with zeros.

To explore this model, tune the **Probability of transition from a silence frame to a speech frame** and **Probability of transition from a speech frame to a silence frame** parameters of the Voice Activity Detector (VAD) and observe the effect on the speech presence probability. Toggle between the gated and original audio signal to assess the quality of your system.



Copyright 2017 The MathWorks, Inc.



See Also

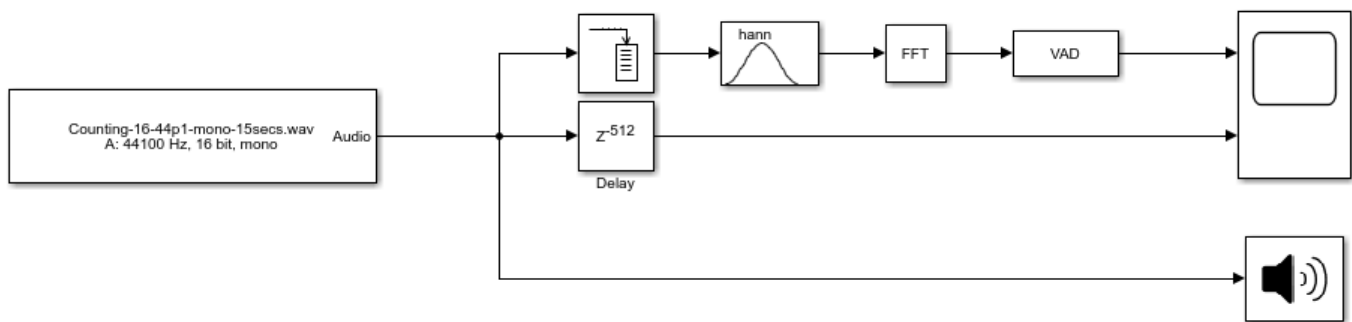
Audio Device Writer | From Multimedia File | If | If Action Subsystem | Manual Switch | Random Source | Time Scope | Voice Activity Detector

Frequency-Domain Voice Activity Detection

This model detects voice activity using a frequency-domain audio signal.

Voice Activity Detection is often used as an indication whether further processing or analysis of a signal is required. Many processing and analysis techniques require a frequency-domain representation of the signal. For example, the voice activity detection algorithm operates in the frequency domain. To save computation, you can convert the audio signal to the frequency domain once, and then feed the frequency-domain signal to downstream analysis and processing.

This model additionally buffers the signal so that the VAD operates on half-overlapped frames. Overlapping the input frames to the VAD increases the accuracy and resolution in time of the probability of speech.



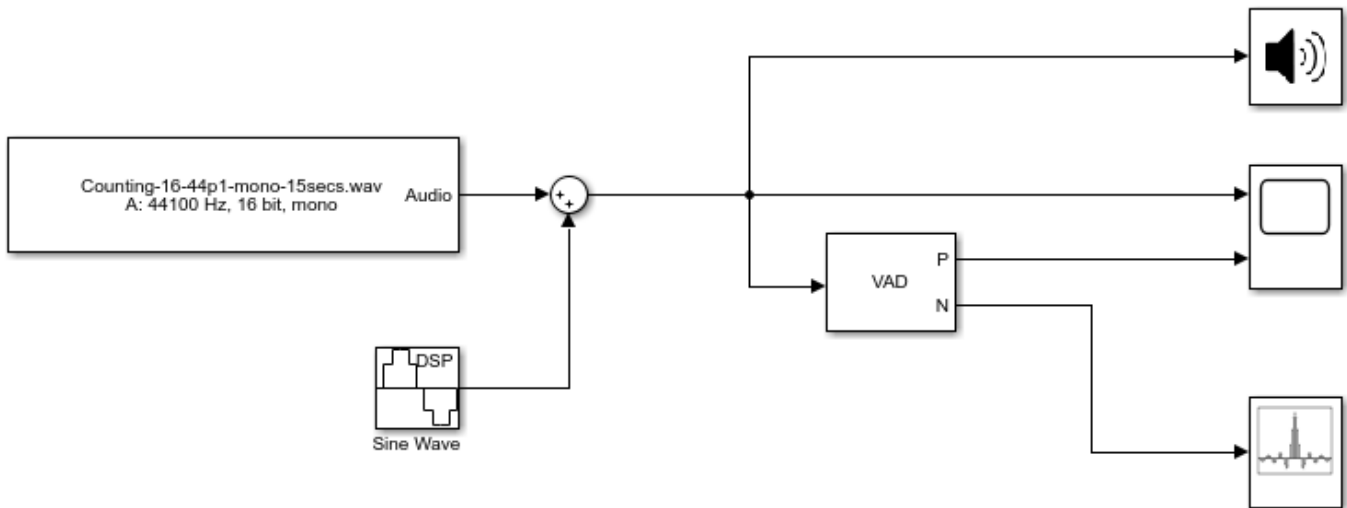
Copyright 2017 The MathWorks, Inc.

See Also

Buffer | Audio Device Writer | Delay | FFT | From Multimedia File | Time Scope | Voice Activity Detector | Window Function

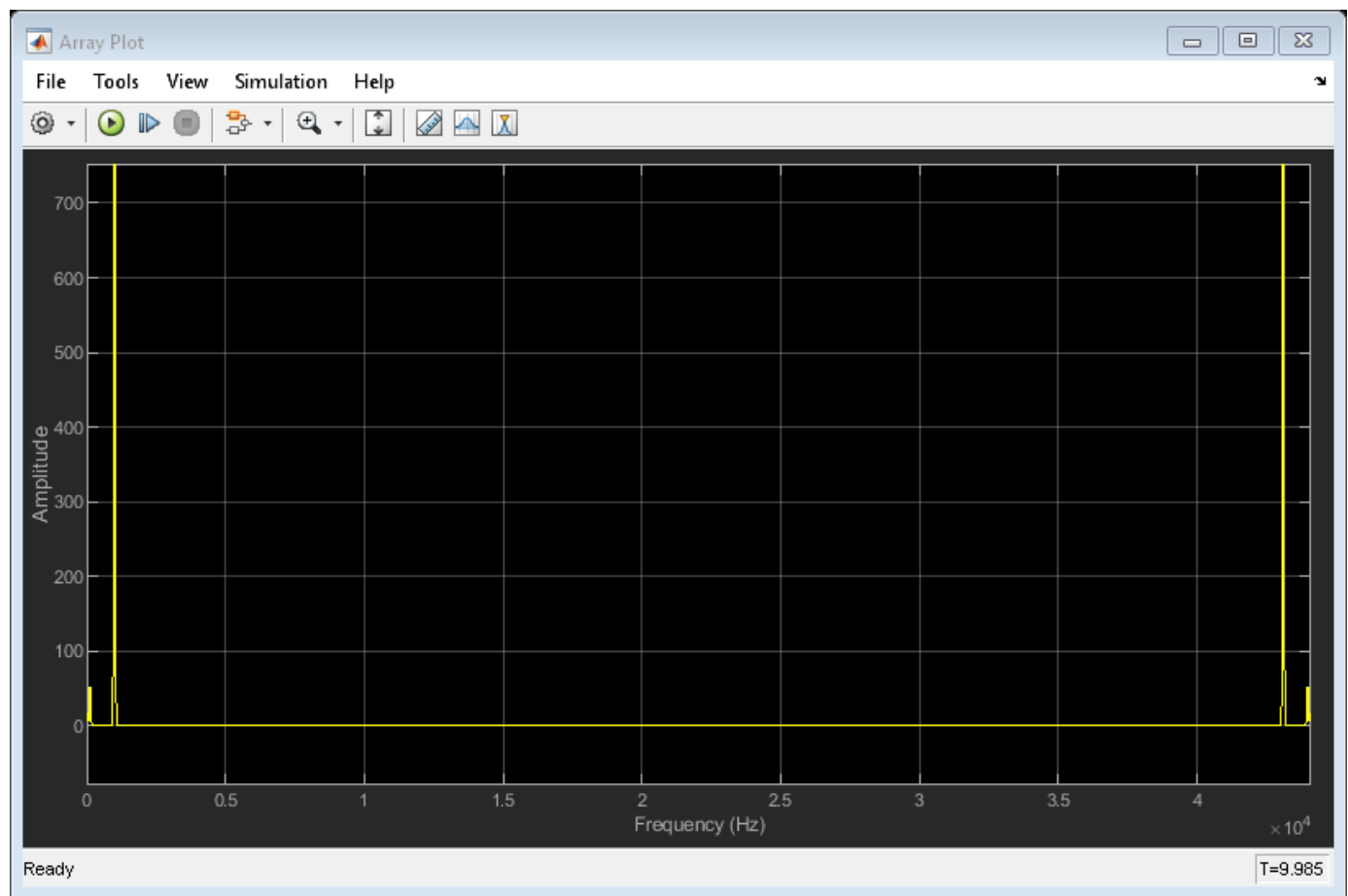
Visualize Noise Power

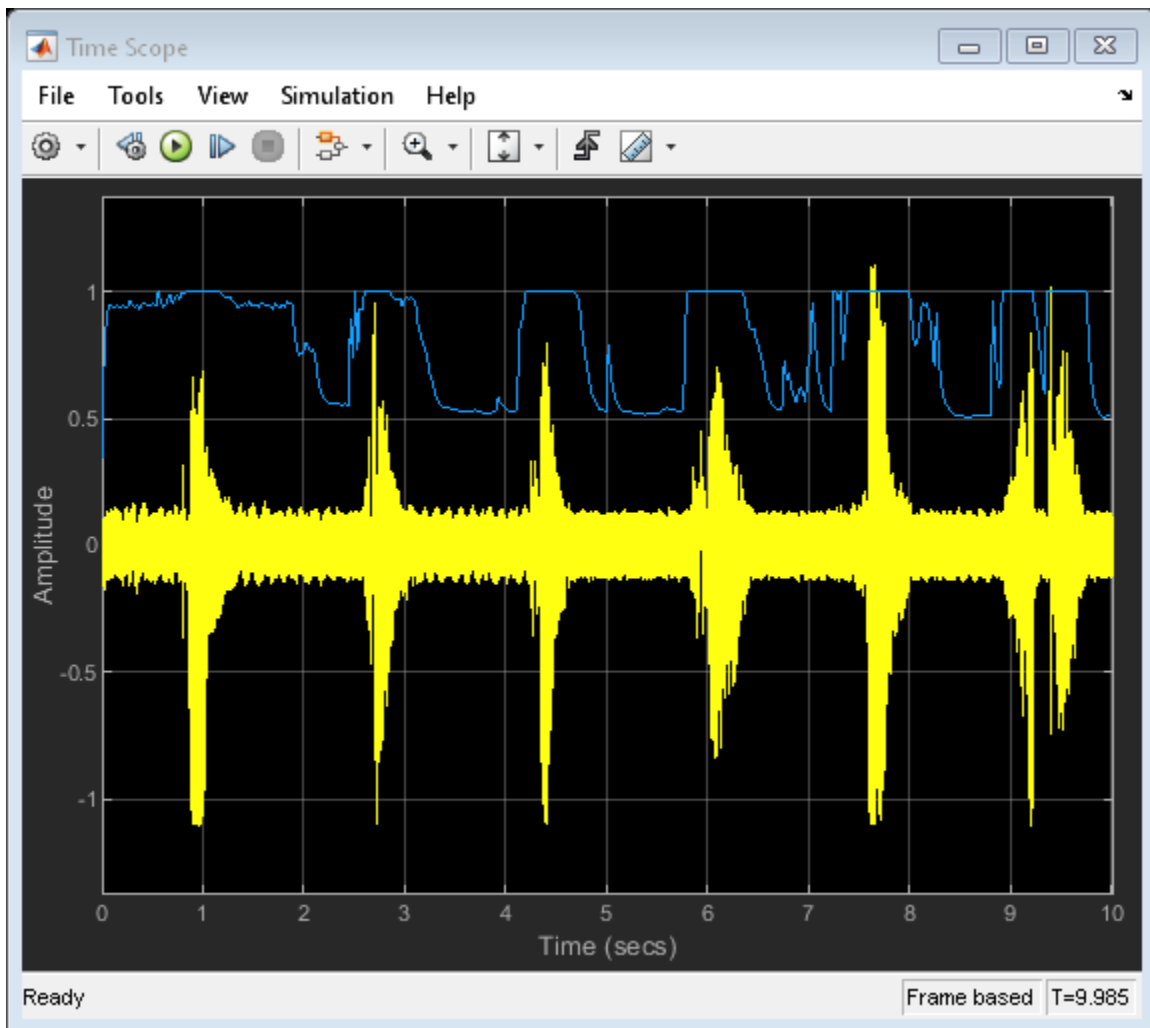
This model plots the noise power estimated by the Voice Activity Detector.



Copyright 2017 The MathWorks, Inc.

To explore this model, tune the **Frequency (Hz)** parameter of the Sine Wave block and observe the noise power estimate updated on the Array Plot block.





Zoom in on the Array Plot to verify that the Voice Activity Detector outputs a good estimate of the noise tone.

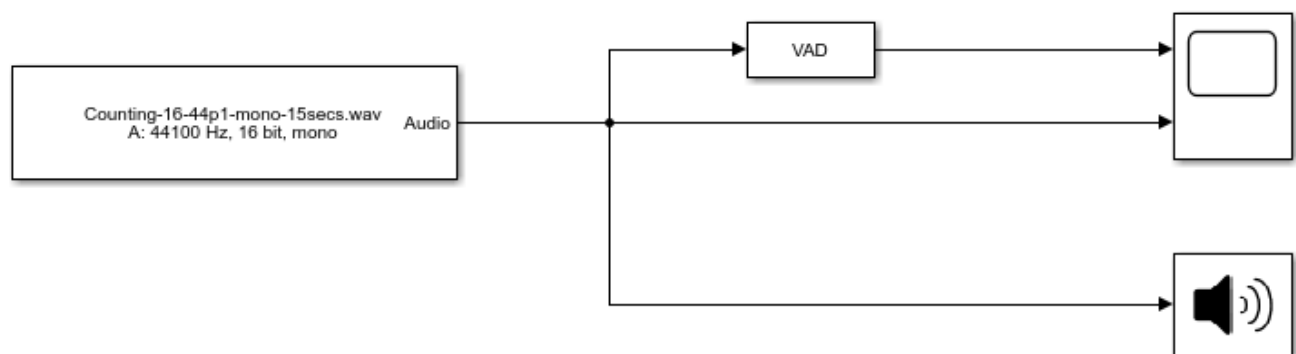
See Also

Array Plot | Audio Device Writer | From Multimedia File | Sine Wave | Time Scope | Voice Activity Detector

Detect Presence of Speech

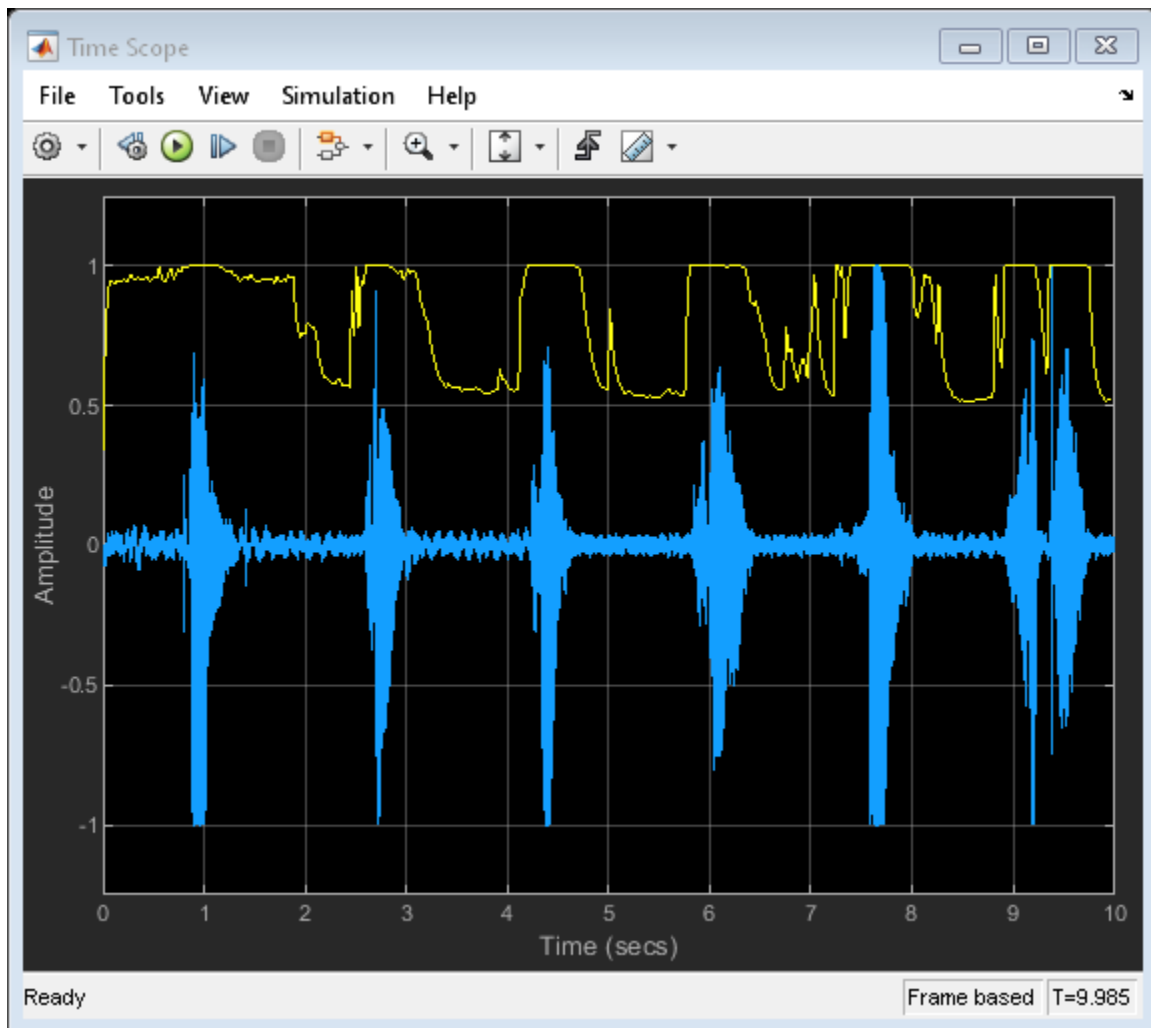
This model uses the Voice Activity Detector block to visualize the probability of speech presence in an audio signal.

To explore this model, tune the **Probability of transition from a silence frame to a speech frame** and **Probability of transition from a speech frame to a silence frame** parameters of the Voice Activity Detector (VAD) and observe the effect on the speech presence probability.



Copyright 2017 The MathWorks, Inc.

The Time Scope blocks plots the audio signal and associated voice activity probability.

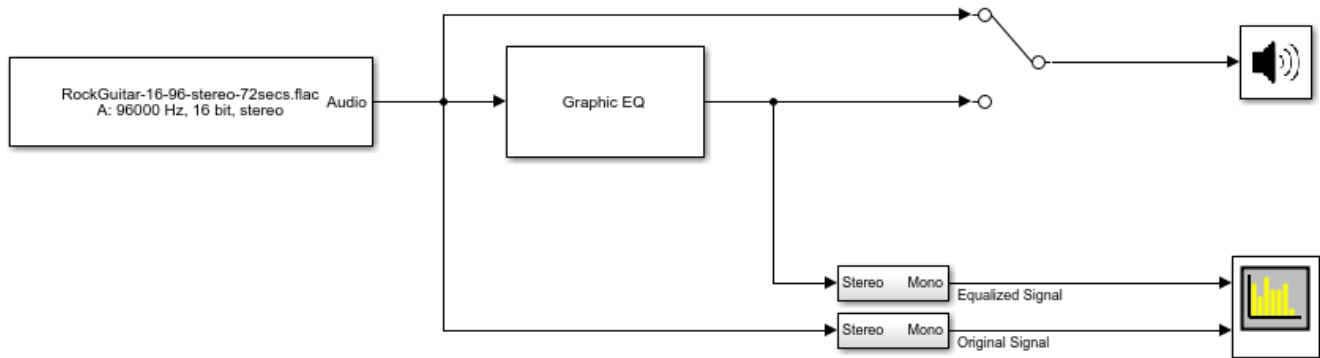


See Also

Audio Device Writer | From Multimedia File | Time Scope | Voice Activity Detector

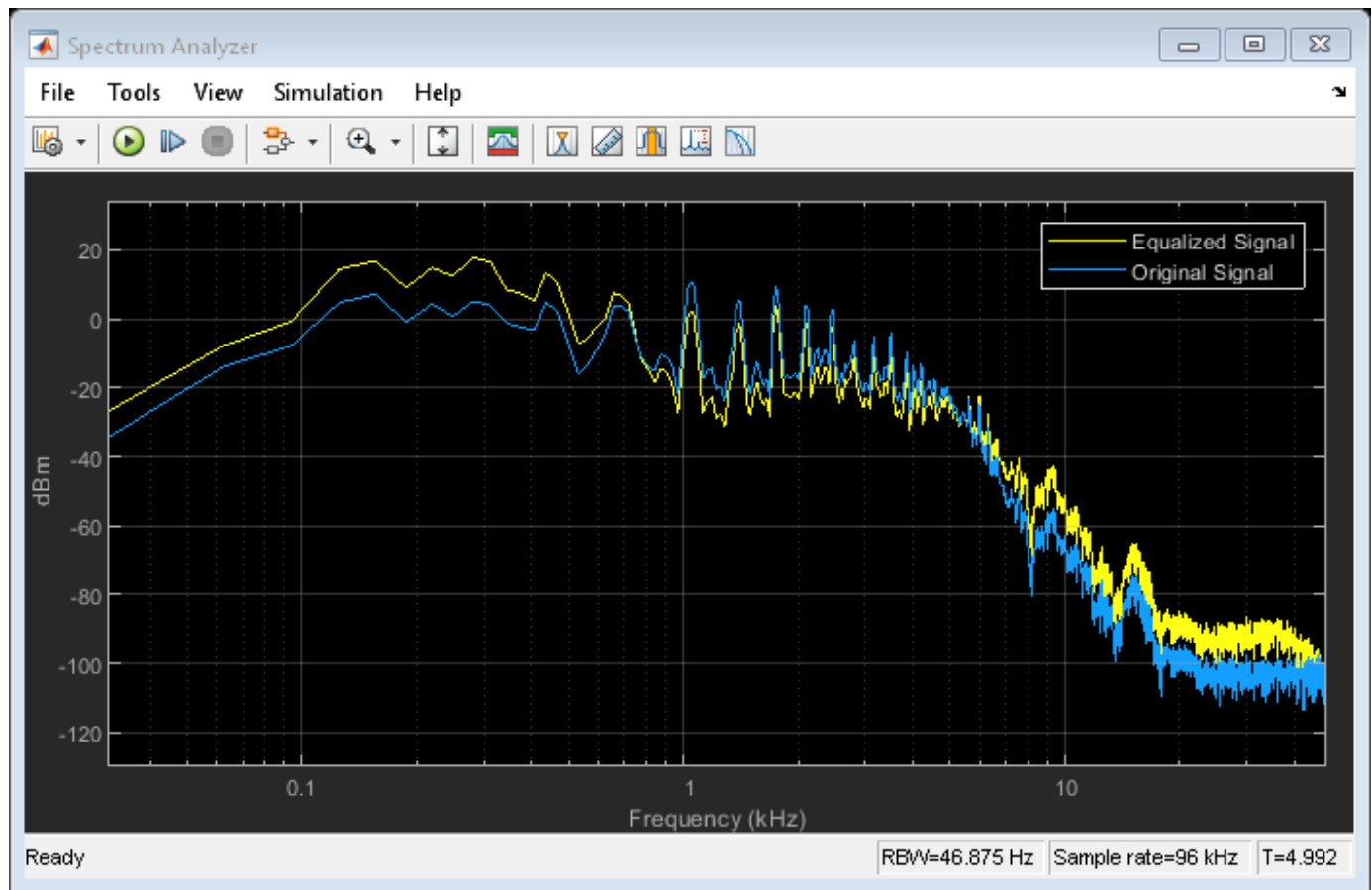
Perform Graphic Equalization

Examine the Graphic EQ block in a Simulink® model and tune parameters.



Copyright 2017 The MathWorks, Inc.

1. Open the Spectrum Analyzer and Graphic EQ blocks.
2. In the Graphic EQ block, click **Visualize equalizer response**. Modify gains of the graphic equalizer and see the magnitude response plot update automatically.
3. Run the model. Tune gains on the Graphic EQ to listen to the effect on your audio device and see the effect on the Spectrum Analyzer display. Double-click the Manual Switch (Simulink) block to toggle between the original and equalized signal as output.



See Also

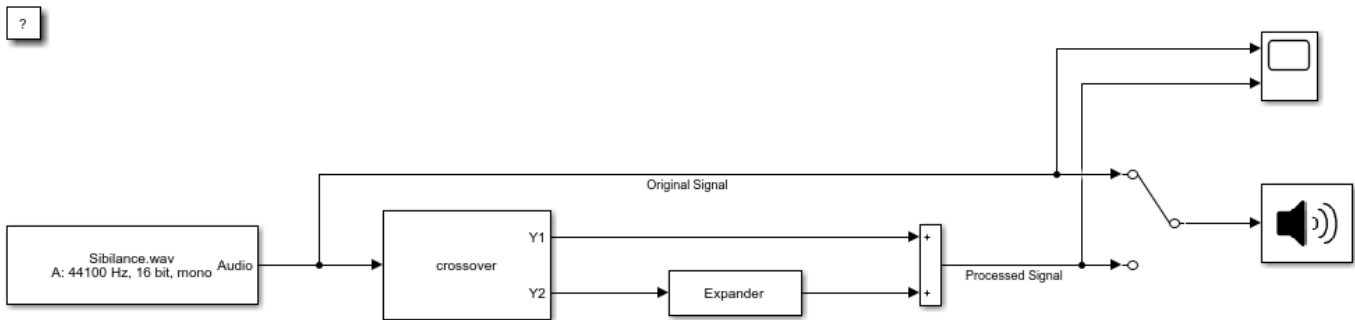
Audio Device Writer | From Multimedia File | Graphic EQ | Spectrum Analyzer

Split-Band De-Essing

This model implements split-band de-essing by separating a speech signal into high and low frequencies, applying dynamic range expansion to diminish the sibilant frequencies, and then remixing the channels.

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.

To explore the model, tune the parameters of the Expander and Crossover Filter blocks. To switch between listening to the processed and unprocessed speech signal, double-click the Manual Switch block. To view the effect of the processing, double-click the Time Scope block.



Copyright 2017 The MathWorks, Inc.

See Also

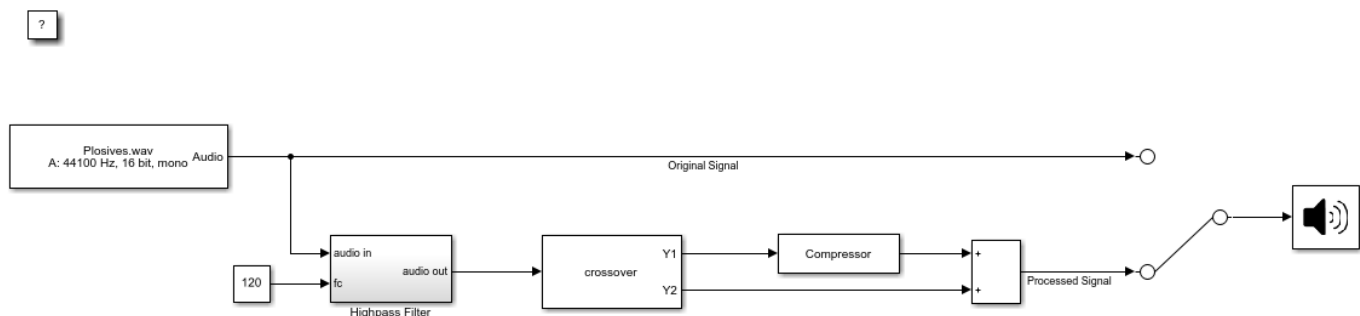
Audio Device Writer | Crossover Filter | Expander | From Multimedia File | Time Scope

Diminish Plosives from Speech

This model minimizes the plosives of a speech signal by applying highpass filtering and low-band compression.

Plosives are consonant sounds resulting from a sudden release of airflow. They are most pronounced in *p*, *d*, and *g* words. Plosives can be emphasized by the recording process and are often displeasurable to hear.

To explore this model, tune the highpass filter cutoff and the parameters on the Compressor and Crossover Filter blocks. Switch between listening to the original and processed signals by double-clicking the Manual Switch block.



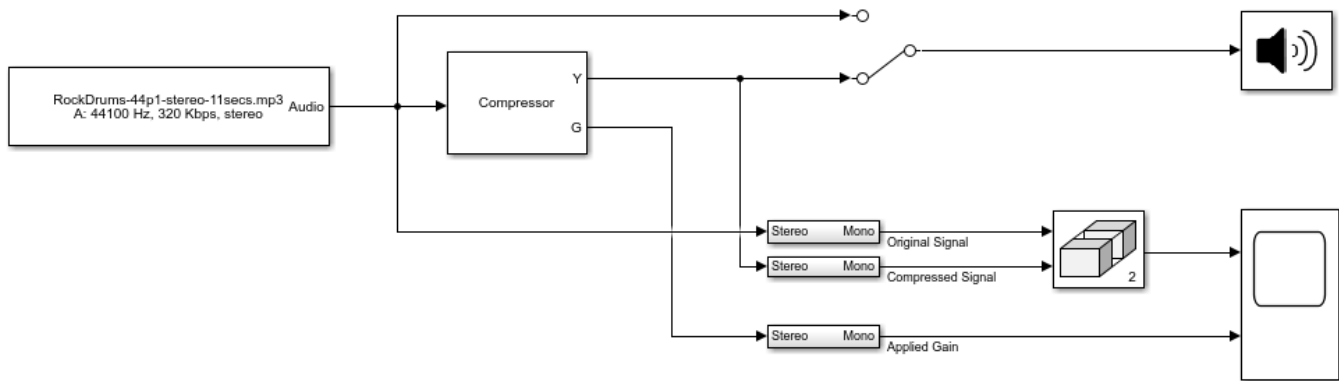
Copyright 2017 The MathWorks, Inc.

See Also

Audio Device Writer | Compressor | Crossover Filter | From Multimedia File

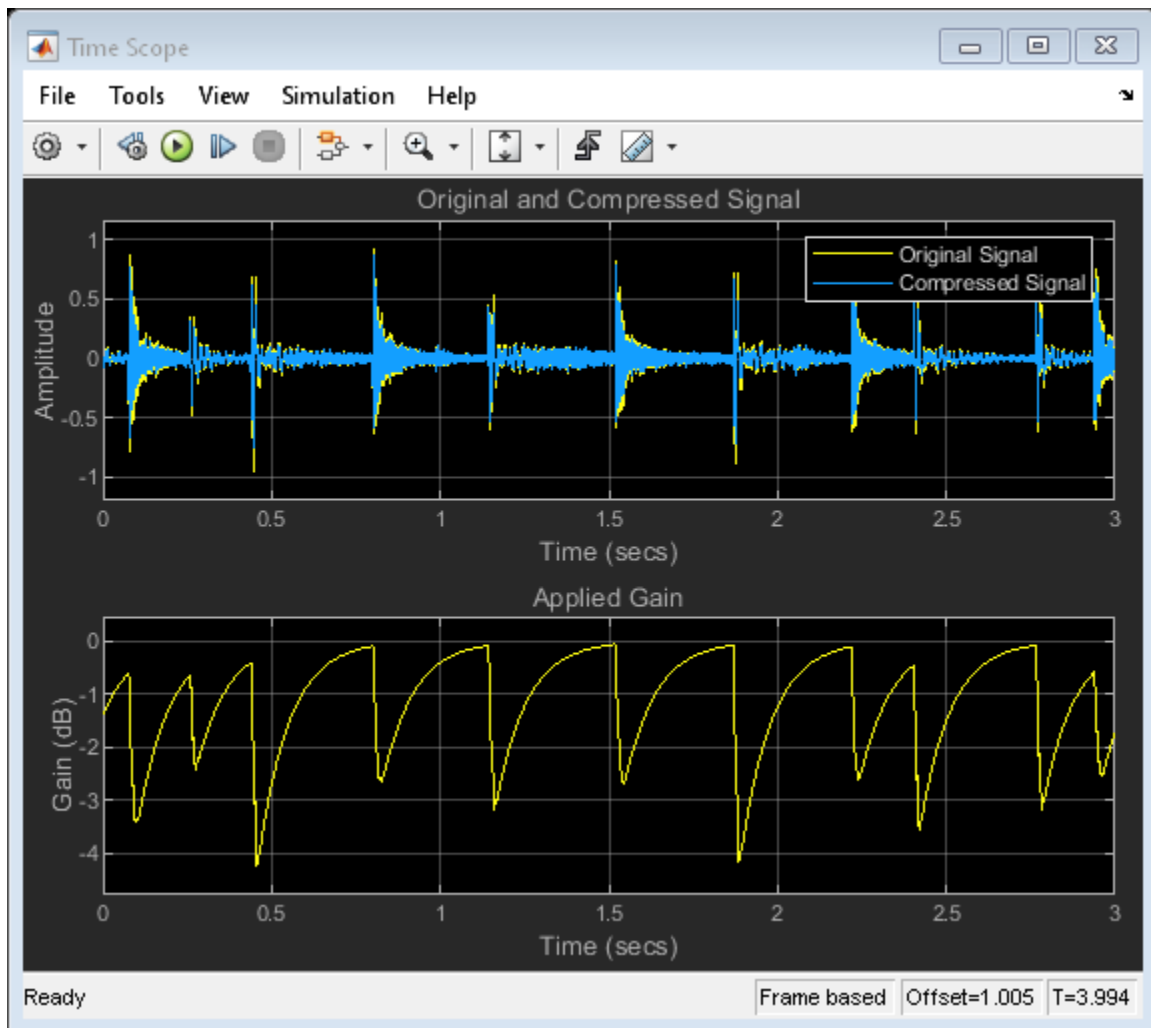
Suppress Loud Sounds

Use the Compressor block to suppress loud sounds and visualize the applied compression gain.



Copyright 2016-2017 The MathWorks, Inc.

1. Open the Time Scope and Compressor blocks.
2. Run the model. To switch between listening to the compressed signal and the original signal, double-click the Manual Switch (Simulink) block.
3. Observe how the applied gain depends on compression parameters and input signal dynamics by tuning the Compressor block parameters and viewing the results on the Time Scope.



See Also

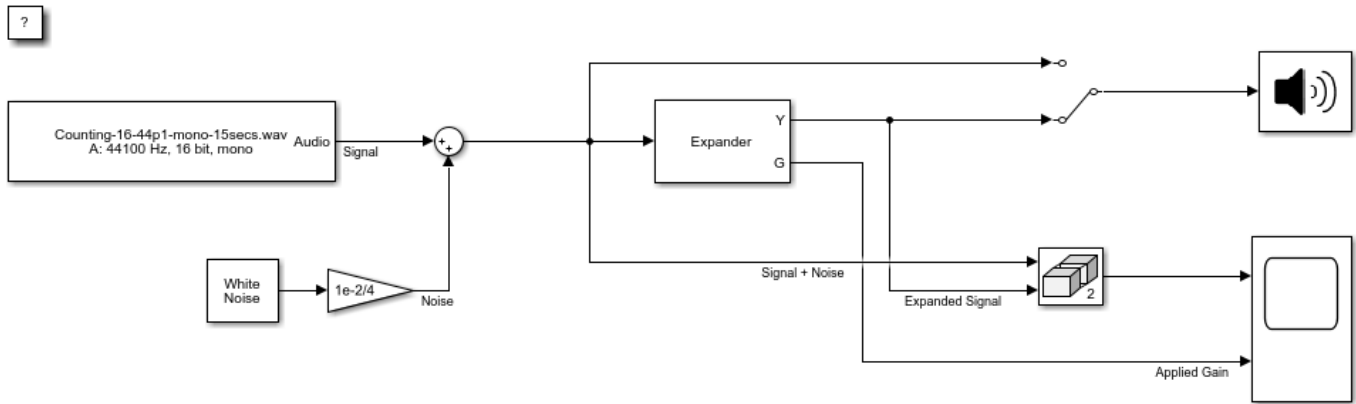
Audio Device Writer | Compressor | From Multimedia File | Time Scope | Vector Concatenate, Matrix Concatenate

More About

- "Dynamic Range Control" on page 8-2

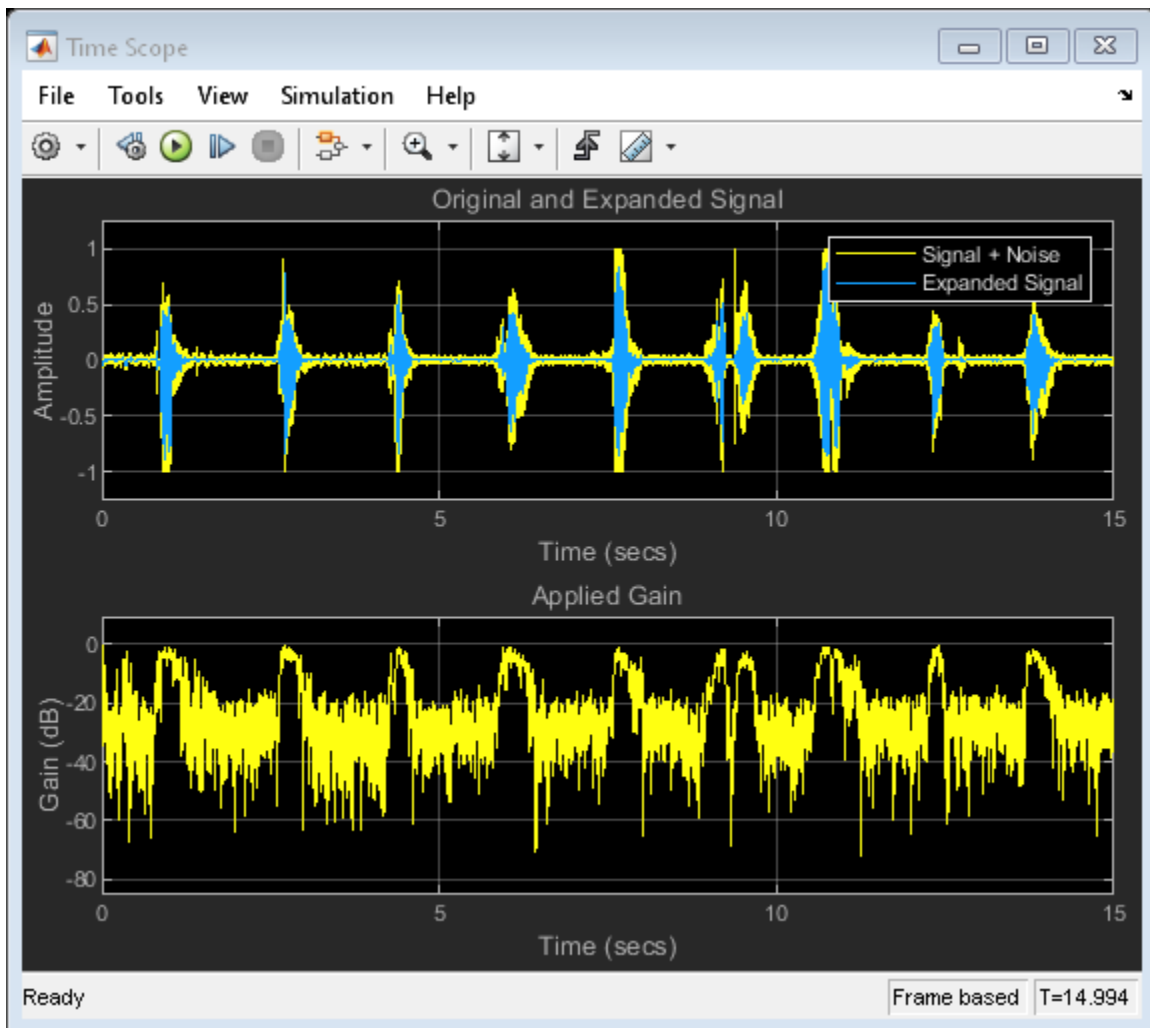
Attenuate Low-Level Noise

Use the Expander block to attenuate low-level noise and visualize the applied dynamic range control gain.



Copyright 2016 The MathWorks, Inc.

1. Open the Time Scope and Expander blocks.
2. Run the model. To switch between listening to the expanded signal and the original signal, double-click the Manual Switch (Simulink) block.
3. Observe how the applied gain depends on expansion parameters and input signal dynamics by tuning the Expander block parameters and viewing the results on the Time Scope.



See Also

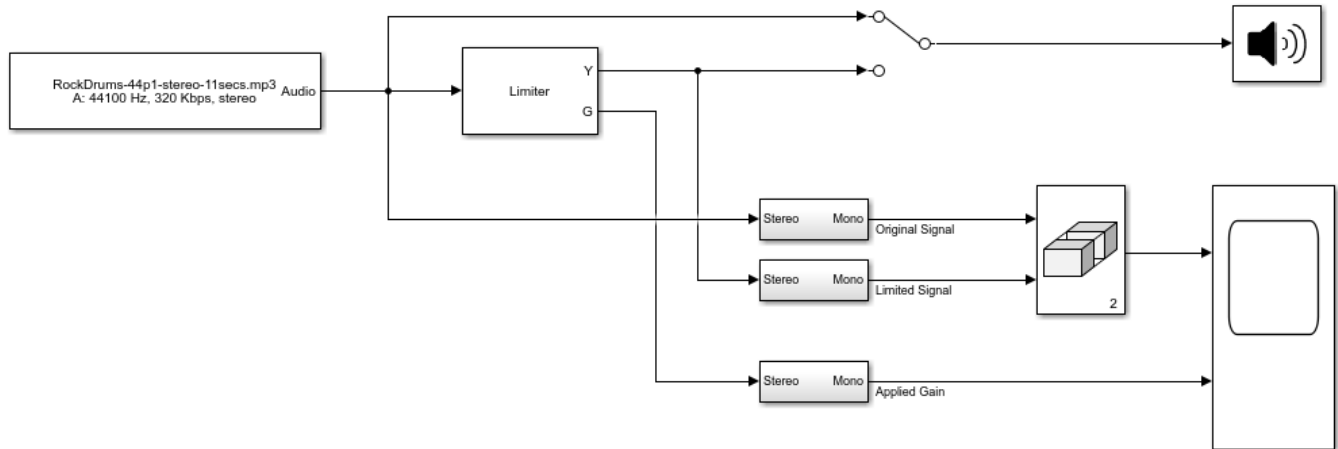
Audio Device Writer | Colored Noise | Expander | From Multimedia File | Time Scope | Vector Concatenate, Matrix Concatenate

More About

- "Dynamic Range Control" on page 8-2

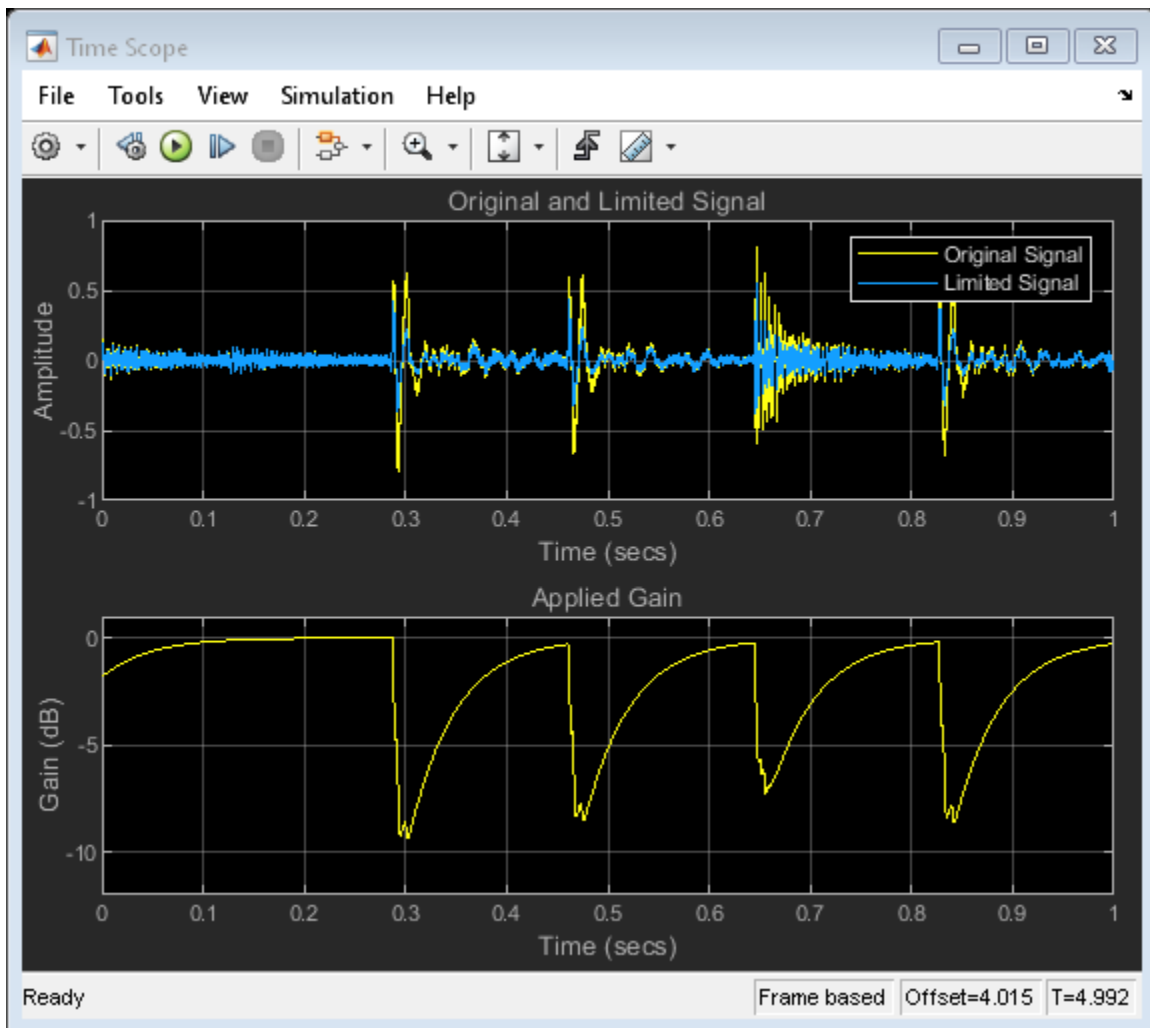
Suppress Volume of Loud Sounds

Suppress the volume of loud sounds and visualize the applied dynamic range control gain.



Copyright 2016 The Mathworks, Inc.

1. Open the Time Scope and Limiter blocks.
2. Run the model. To switch between listening to the gated signal and the original signal, double-click the Manual Switch (Simulink) block.
3. Observe how the applied gain depends on dynamic range limiting parameters and input signal dynamics by tuning Limiter block parameters and viewing the results on the Time Scope.



See Also

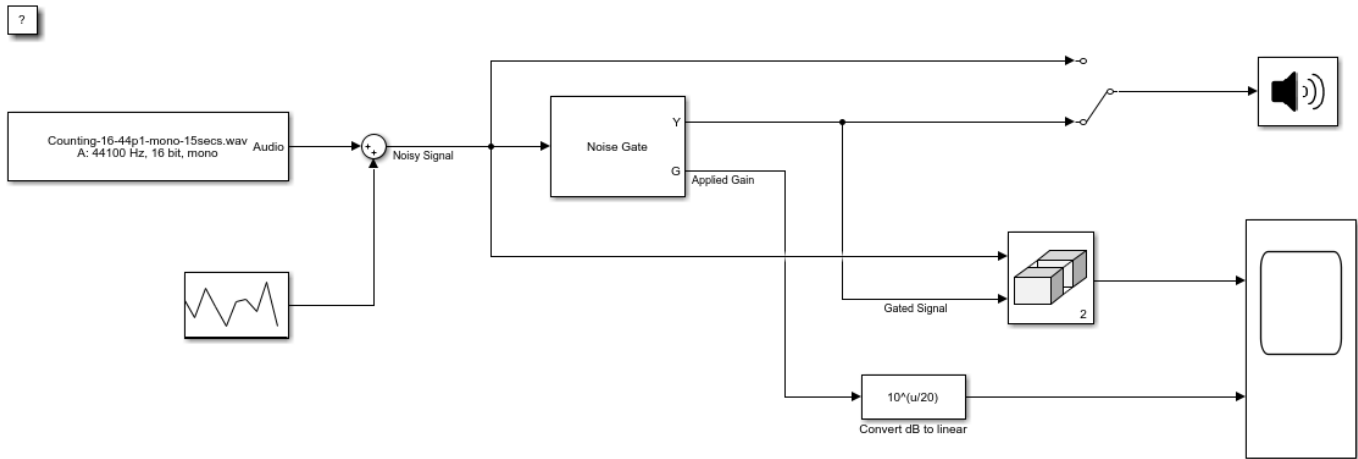
Audio Device Writer | From Multimedia File | Limiter | Time Scope | Vector Concatenate, Matrix Concatenate

More About

- "Dynamic Range Control" on page 8-2

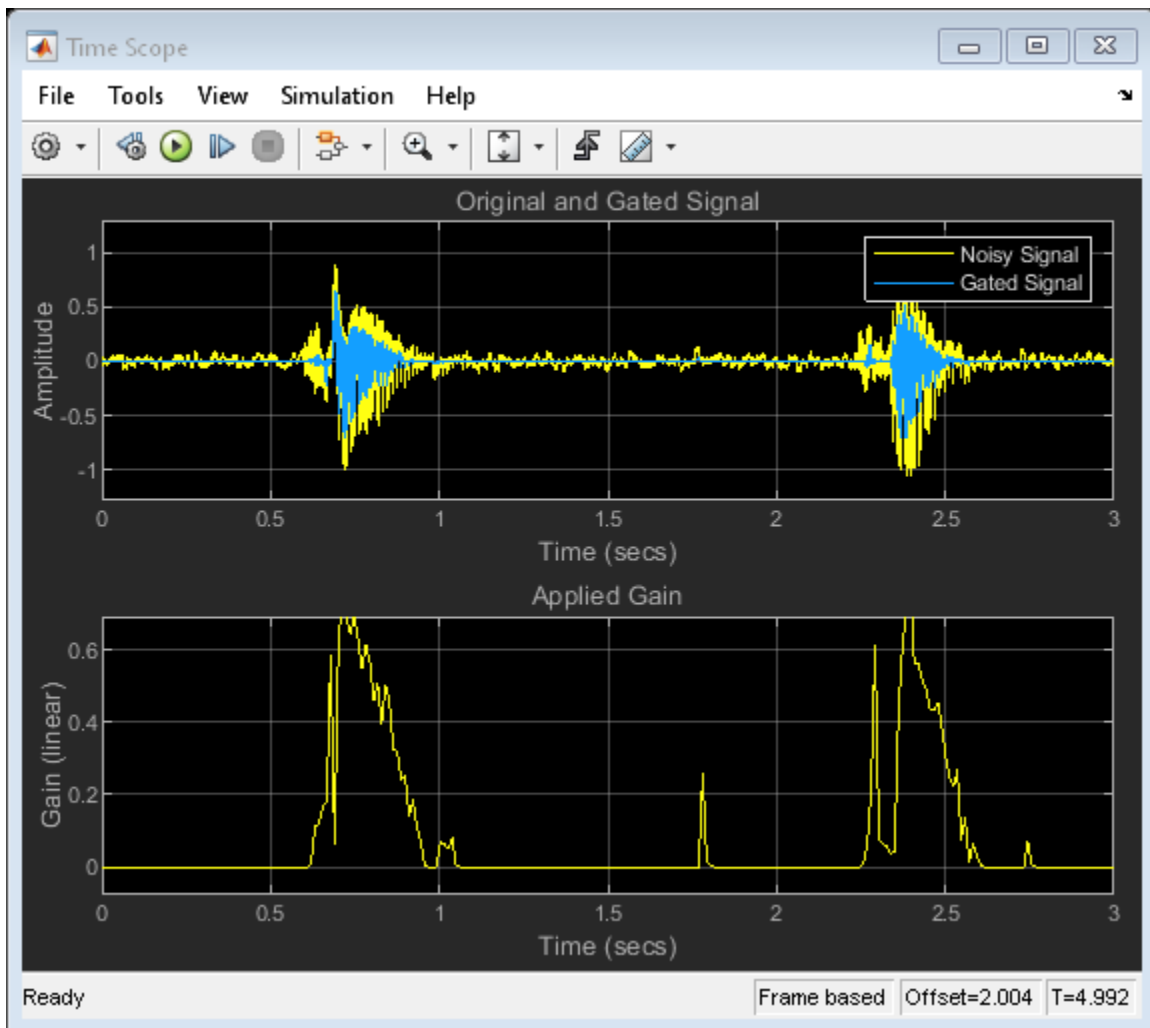
Gate Background Noise

Apply dynamic range gating to remove low-level noise from an audio file.



Copyright 2016 The MathWorks, Inc.

1. Open the Time Scope and Noise Gate blocks.
2. Run the model. To switch between listening to the gated signal and the original signal, double-click the Manual Switch (Simulink) block.
3. Observe how the applied gain depends on noise gate parameters and input signal dynamics by tuning Noise Gate block parameters and viewing the results on the Time Scope.



See Also

Audio Device Writer | From Multimedia File | Noise Gate | Random Source | Time Scope | Vector Concatenate, Matrix Concatenate

More About

- "Dynamic Range Control" on page 8-2

Output Values from MIDI Control Surface

The example shows how to set the MIDI Controls block parameters to output control values from your MIDI device.

1. Connect a MIDI device to your computer and then open the model.



Copyright 2016 The MathWorks, Inc.

2. Run the model with default settings. Move any controller on your default MIDI device to update the Display block.
3. Stop the simulation.
4. At the MATLAB™ command line, use `midiid` to determine the name of your MIDI device and two control numbers associated with your device.
5. In the MIDI Control block dialog box, set **MIDI device** to `Specify other` and enter the name of your MIDI device.
6. Set **MIDI controls** to `Respond to specified controls` and enter the control numbers determined using `midiid`.
7. Specify initial values as a vector the same size as **MIDI control numbers**. The initial values you specify are quantized according to the MIDI protocol and your particular MIDI surface.

The dialog box shows sample values for a 'BCF2000' MIDI device with control numbers 1081 and 1083.

Parameters

MIDI device: Specify other

MIDI device name: 'BCF2000'

MIDI controls: Respond to specified controls

MIDI control numbers: [1081,1083]

Initial values: [0.1,0.9]

☐ Send initial values to device at start

Output mode: Normalized (0 - 1)

8. Click **OK**, and then run the model. Verify that the Display block shows initial values and updates when you move the specified controls.

See Also

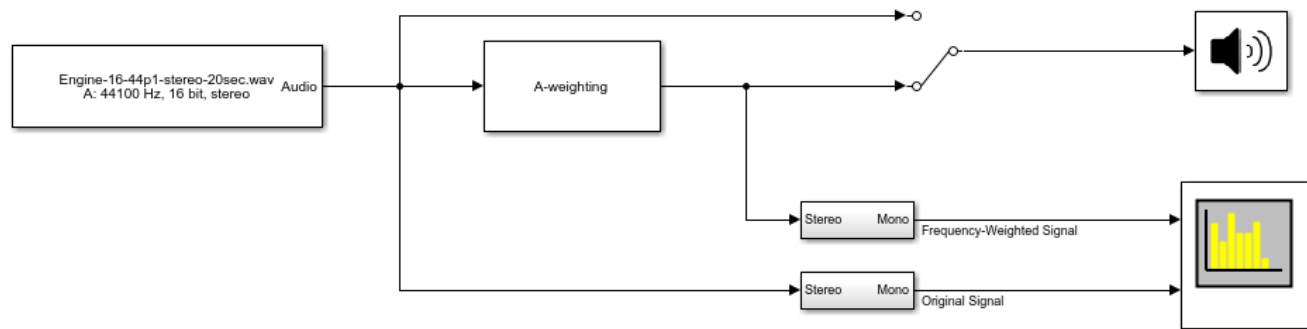
Audio Device Writer | From Multimedia File | MIDI Controls | Time Scope | Vector Concatenate, Matrix Concatenate

More About

- “MIDI Control Surface Interface” on page 10-2

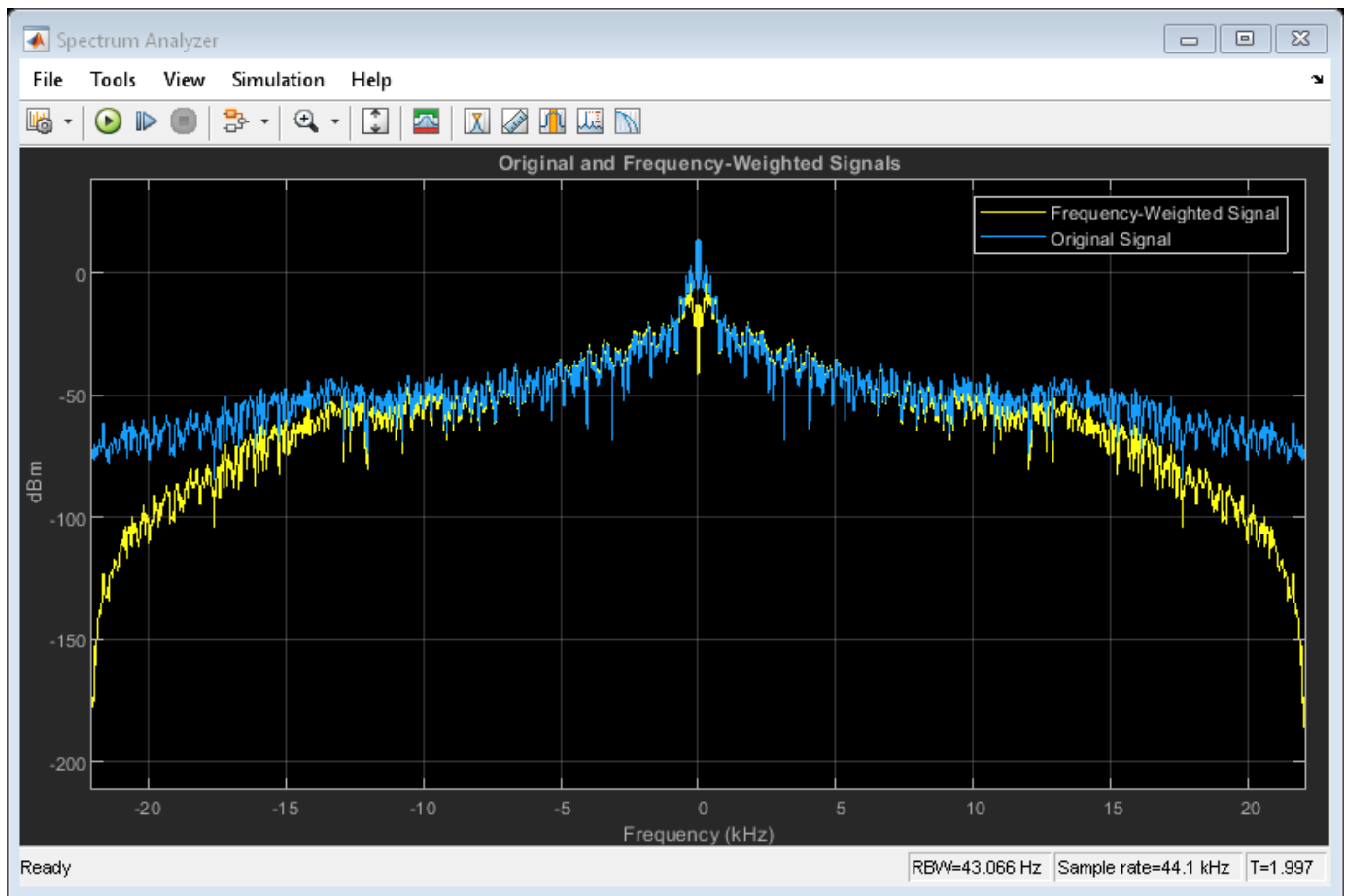
Apply Frequency Weighting

Examine the Weighting Filter block in a Simulink® model and tune parameters.



Copyright 2016 The MathWorks, Inc.

1. Open the Spectrum Analyzer block.
2. Run the model. Switch between listening to the frequency-weighted signal and the original signal by double-clicking the Manual Switch (Simulink) block.
3. Stop the model. Open the Weighting Filter block and choose a different weighting method. Observe the difference in simulation.

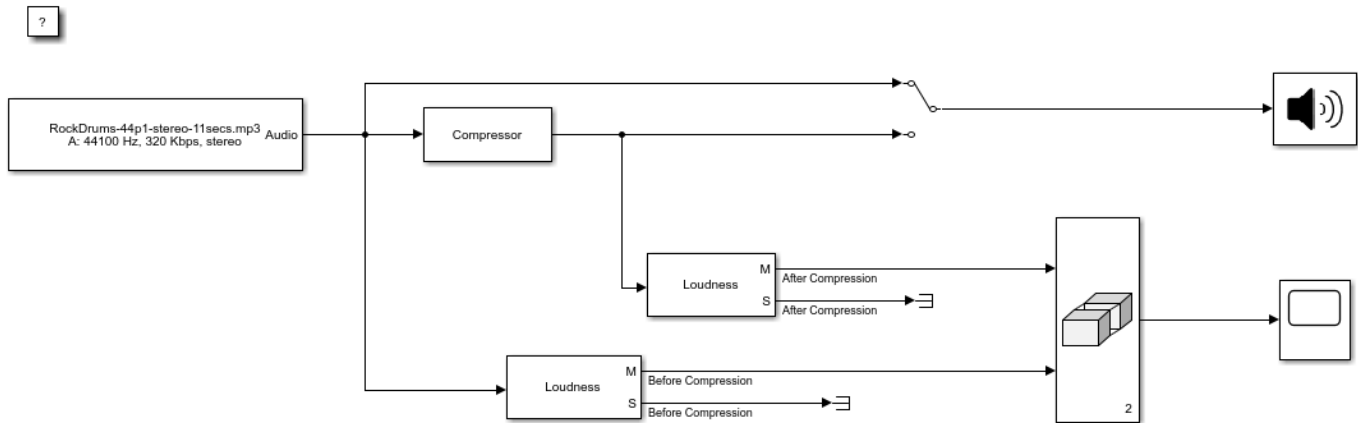


See Also

Audio Device Writer | From Multimedia File | Spectrum Analyzer | Weighting Filter

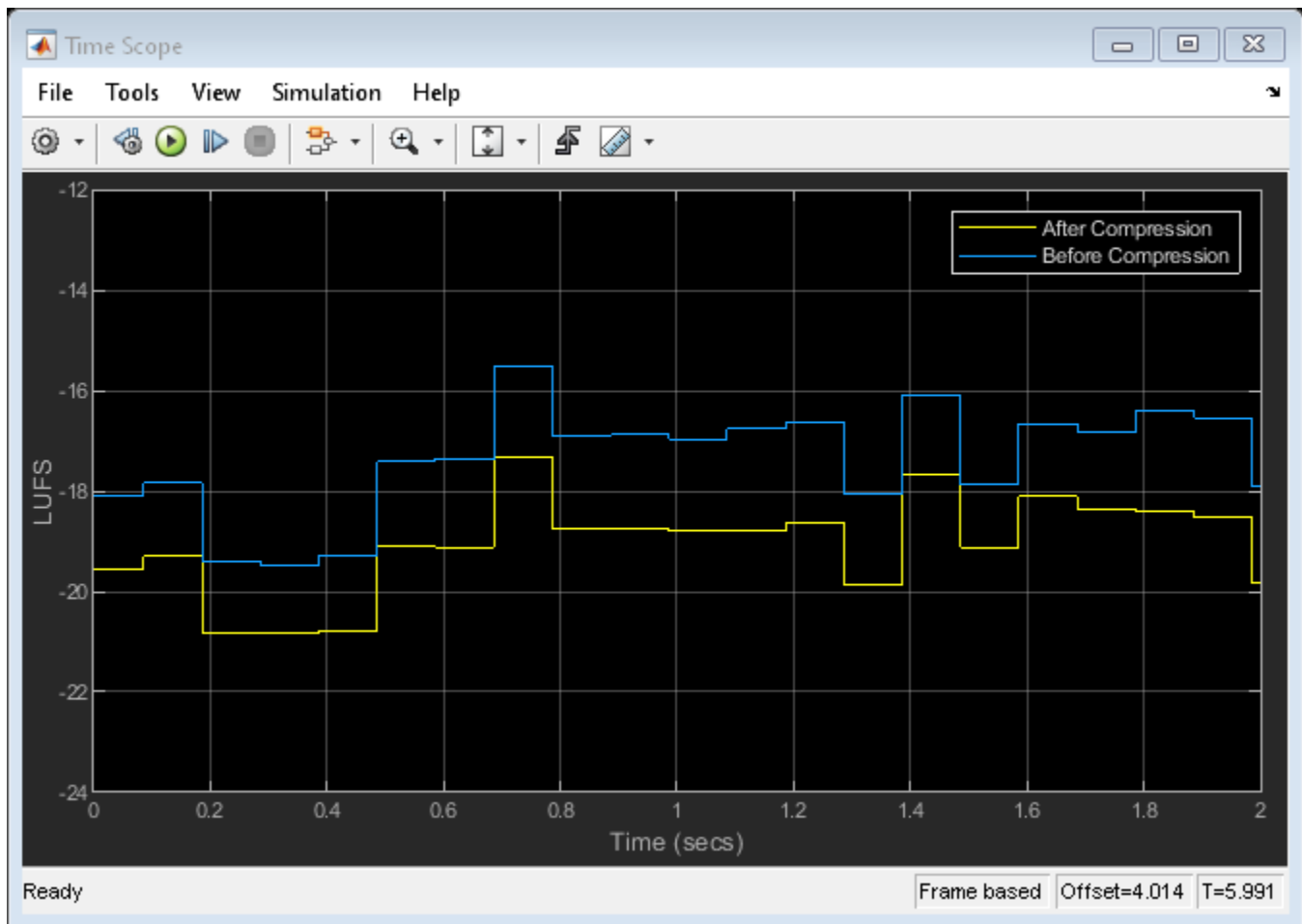
Compare Loudness Before and After Audio Processing

Measure momentary and short-term loudness before and after compression of a streaming audio signal in Simulink®.



Copyright 2016 The MathWorks, Inc.

1. Open the Time Scope and Compressor blocks.
2. Run the model. To switch between listening to the compressed signal and the original signal, double-click the switch.
3. Observe the effect of compression on loudness by tuning the Compressor block parameters and viewing the momentary loudness on the Time Scope block.



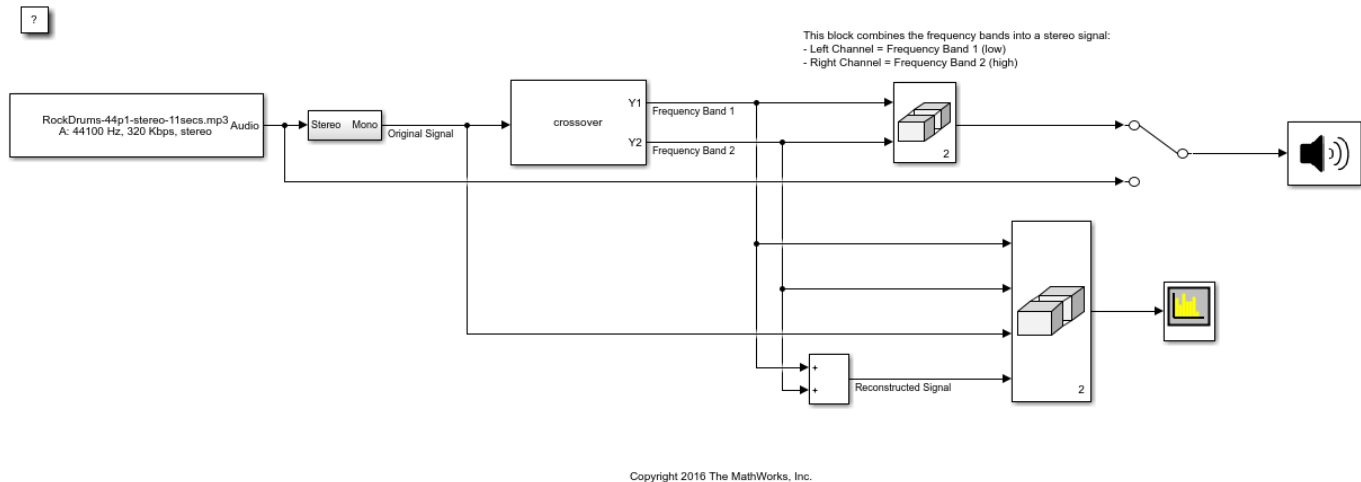
4. Stop the model. For both Loudness blocks, replace momentary loudness with short-term loudness as input to the Matrix Concatenate block. Run the model again and observe the effect of compression on short-term loudness.

See Also

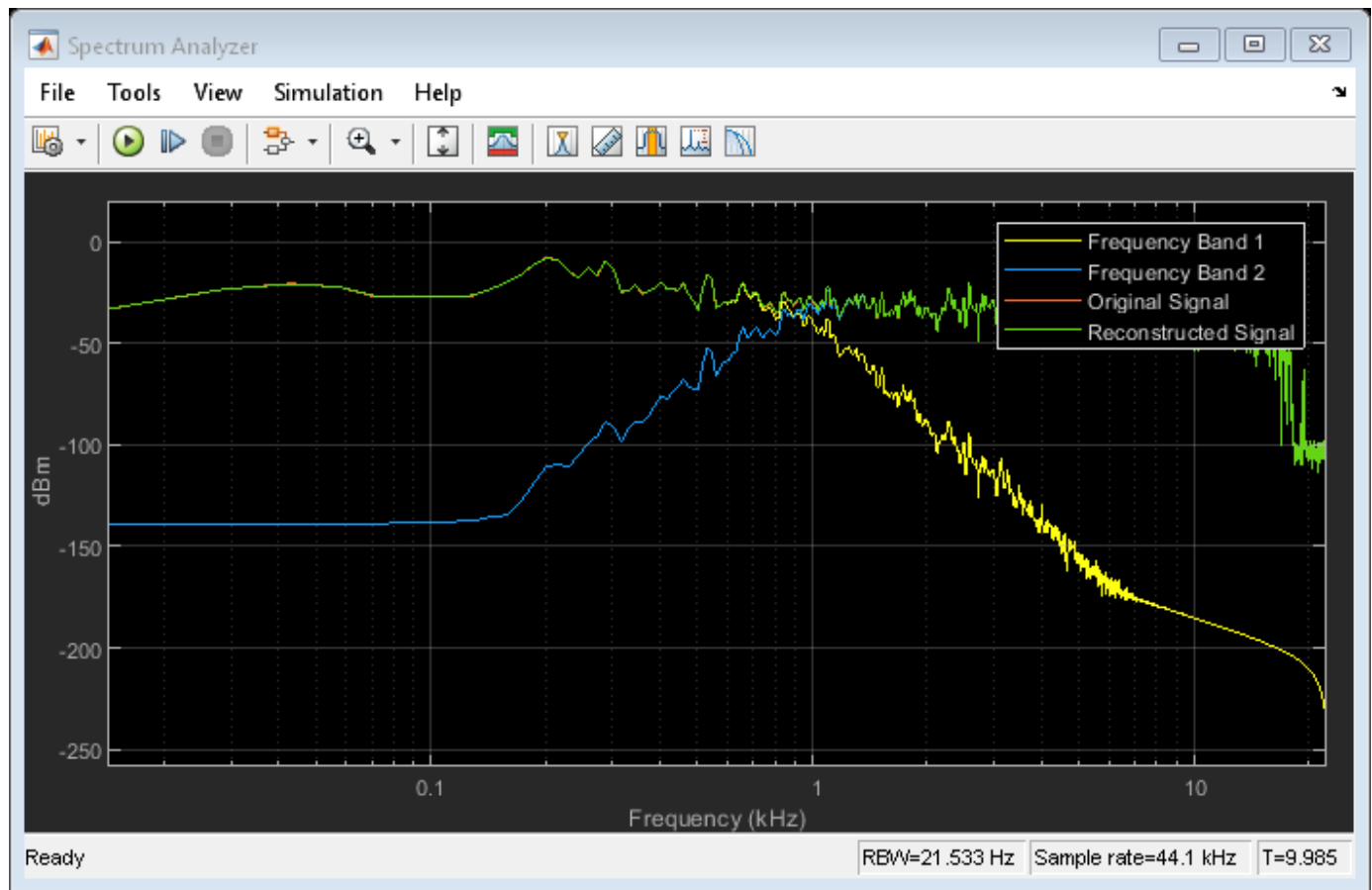
Audio Device Writer | Compressor | From Multimedia File | Loudness Meter | Time Scope | Vector Concatenate, Matrix Concatenate

Two-Band Crossover Filtering for a Stereo Speaker System

Divide a mono signal into a stereo signal with distinct frequency bands. To hear the full effect of this simulation, use a stereo speaker system, such as headphones.



1. Open the Spectrum Analyzer and Crossover Filter blocks.
2. Run the model. To switch between listening to the filtered and original signal, double-click the Manual Switch (Simulink) block.
3. Tune the crossover frequency on the Crossover Filter block to listen to the effect on your speakers and view the effect on the Spectrum Analyzer block.

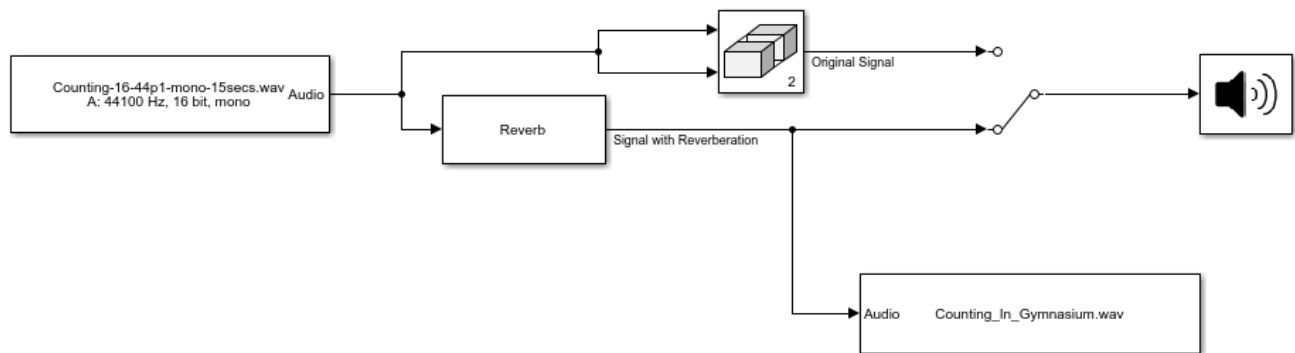


See Also

Audio Device Writer | Crossover Filter | From Multimedia File | Spectrum Analyzer | Vector Concatenate, Matrix Concatenate

Mimic Acoustic Environments

Examine the Reverberator block in a Simulink® model and tune parameters. The reverberation parameters in this model mimic a large room with hard walls, such as a gymnasium.



Copyright 2016 The Mathworks, Inc.

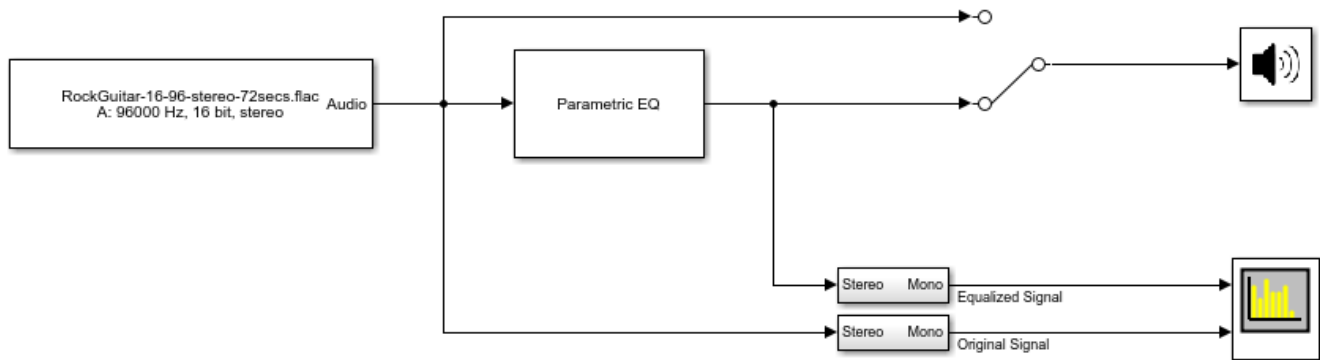
1. Run the simulation. Listen to the audio signal with and without reverberation by double-clicking the Manual Switch block.
2. Stop the simulation.
3. Disconnect the From Multimedia File block so that you can run the model without recording.
4. Open the Reverberator block.
5. Run the simulation and tune the parameters of the Reverberator block.
6. After you are satisfied with the reverberation environment, stop the simulation.
7. Reconnect the To Multimedia File block. Rename the output file with a description to match your reverberation environment, and rerun the model.

See Also

Audio Device Writer | From Multimedia File | Reverberator | To Multimedia File | Vector Concatenate, Matrix Concatenate

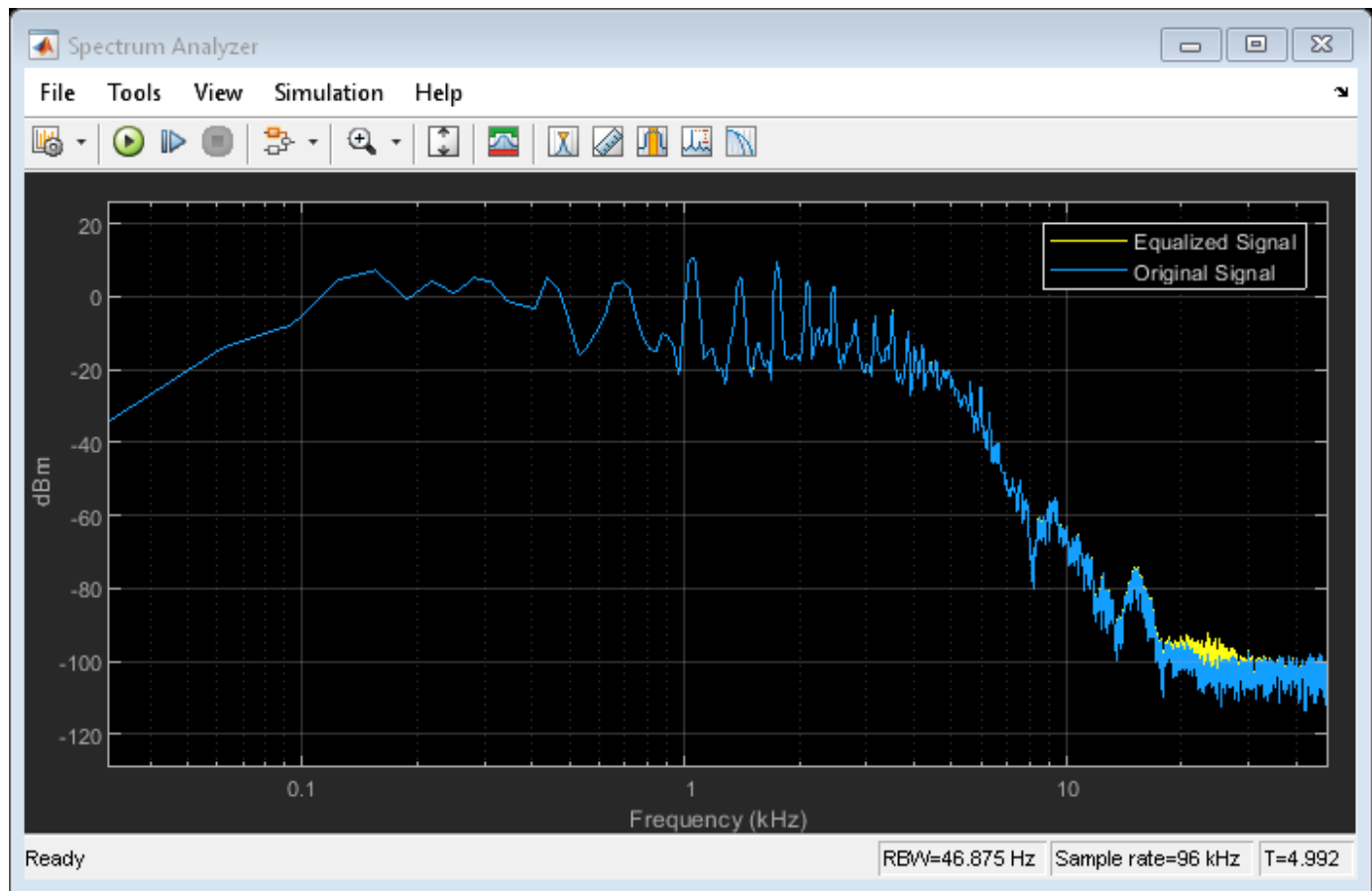
Perform Parametric Equalization

Examine the Parametric EQ block in a Simulink® model and tune parameters.



Copyright 2016-2019 The MathWorks, Inc.

1. Open the Spectrum Analyzer and Parametric EQ blocks.
2. In the Parametric EQ block, click **View Filter Response**. Modify parameters of the parametric equalizer and see the magnitude response plot update automatically.
3. Run the model. Tune parameters on the Parametric EQ to listen to the effect on your audio device and see the effect on the Spectrum Analyzer display. Double-click the Manual Switch (Simulink) block to toggle between the original and equalized signal as output.

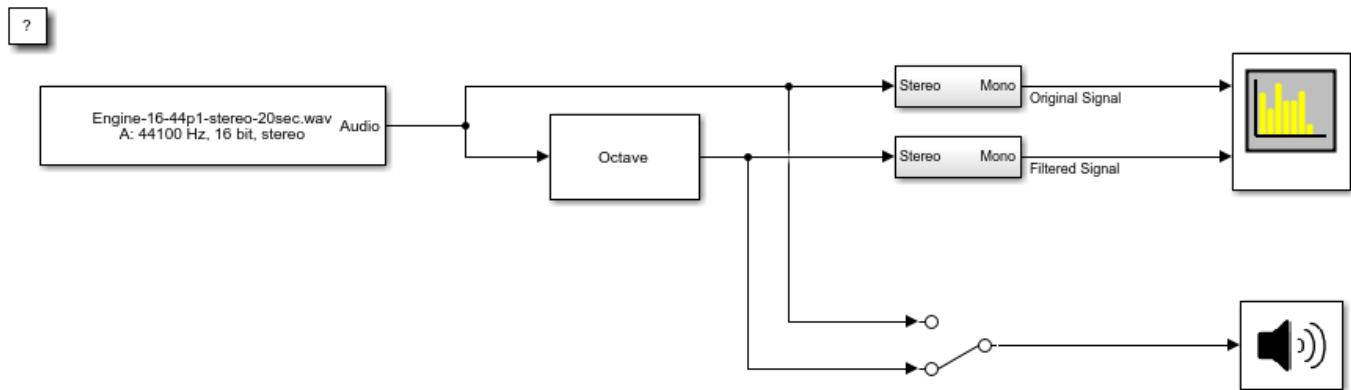


See Also

Audio Device Writer | From Multimedia File | Parametric EQ | Spectrum Analyzer | Vector Concatenate, Matrix Concatenate

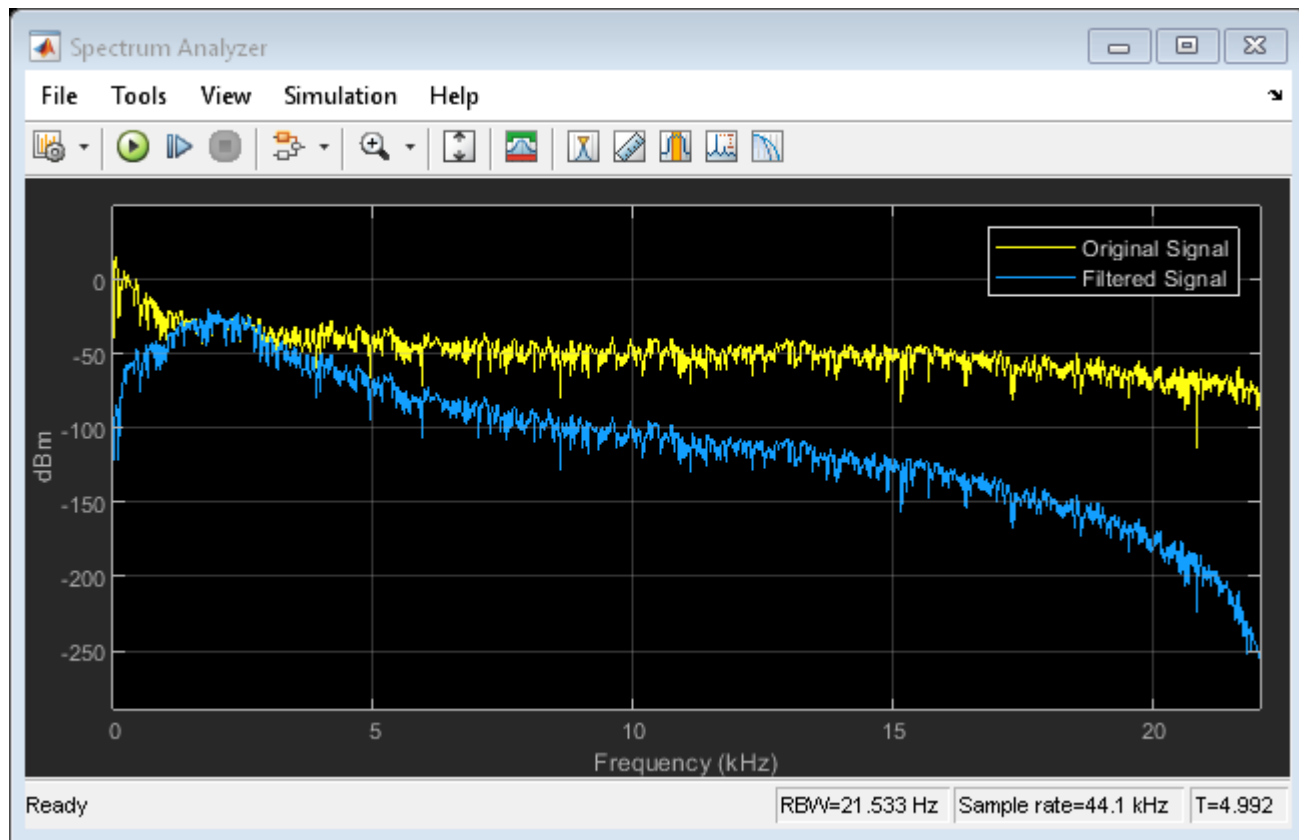
Perform Octave Filtering

Examine the Octave Filter block in a Simulink® model and tune parameters.



Copyright 2016 The MathWorks, Inc.

1. Open the Octave Filter block and click **Visualize filter response**. Tune parameters on the Octave Filter dialog. The filter response visualization updates automatically. If you break compliance with the ANSI S1.11-2004 standard, the filter mask is drawn in red.
2. Run the model. Open the Spectrum Analyzer block. Tune parameters on the Octave Filter block to listen to the effect on your audio device and see the effect on the Spectrum Analyzer display. Switch between listening to the filtered and unfiltered audio by double-clicking the Manual Switch (Simulink) block.

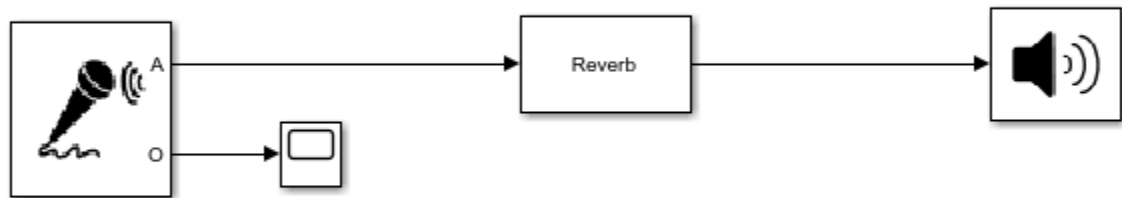


See Also

Audio Device Writer | From Multimedia File | Octave Filter | Spectrum Analyzer

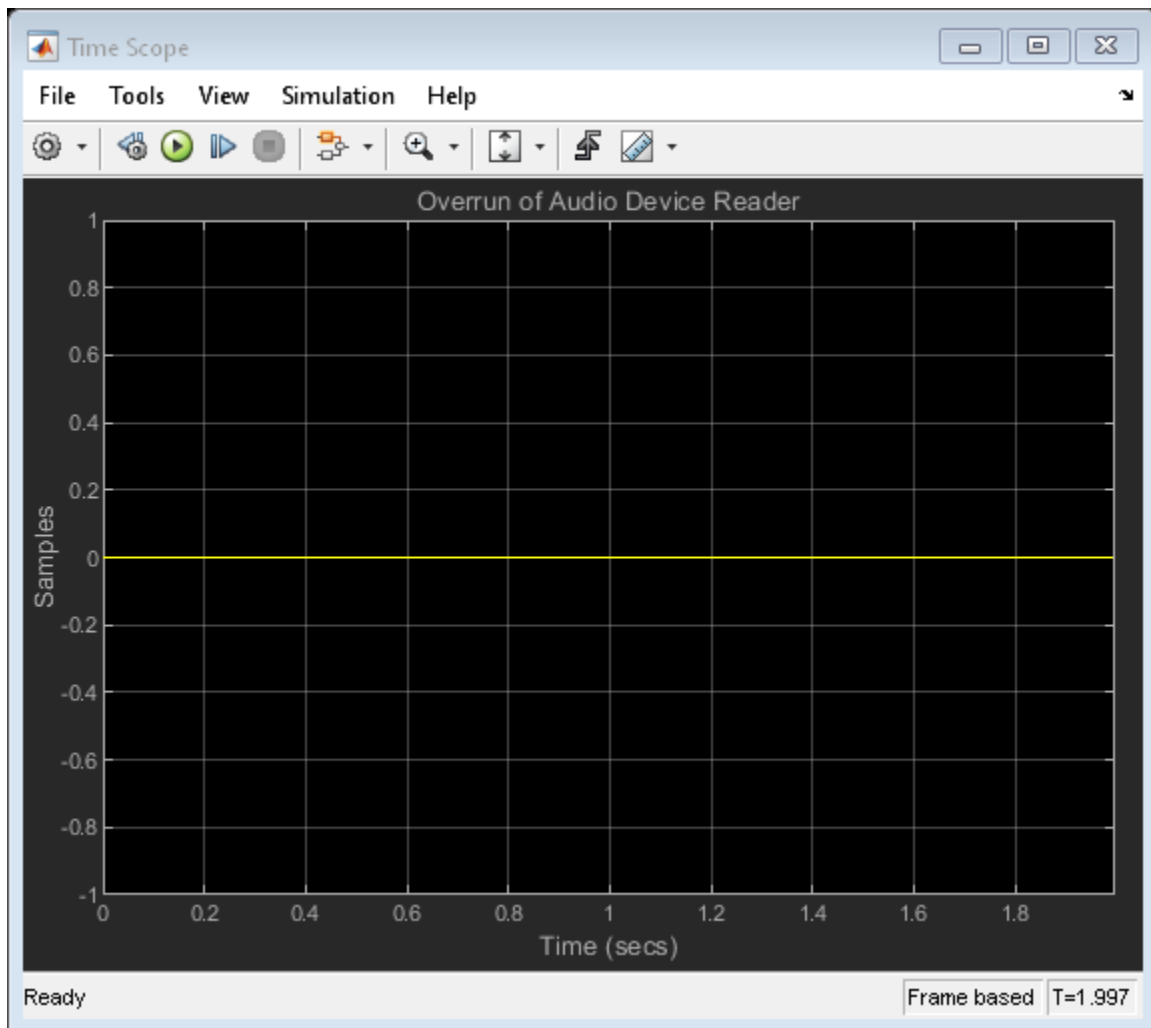
Read from Microphone and Write to Speaker

Examine the Audio Device Reader block in a Simulink® model, modify parameters, and explore overrun.



Copyright 2016 The MathWorks, Inc.

1. Run the model. The Audio Device Reader records an audio stream from your computer's default audio input device. The Reverberator block processes your input audio. The Audio Device Writer block sends the processed audio to your default audio output device.



2. Stop the model. Open the Audio Device Reader block and lower the **Samples per frame** parameter. Open the Time Scope block to view overrun.

3. Run the model again. Lowering the **Samples per frame** decreases the buffer size of your Audio Device Reader block. A smaller buffer size decreases audio latency while increasing the likelihood of overruns.

See Also

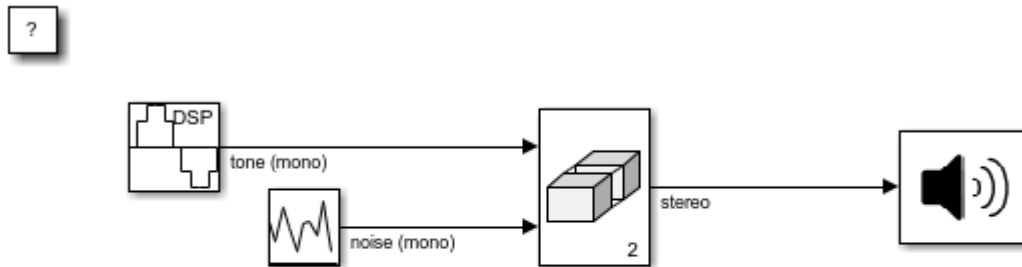
Audio Device Reader | Audio Device Writer | Reverberator | Time Scope

More About

- "Audio I/O: Buffering, Latency, and Throughput"

Channel Mapping

Examine the Audio Device Writer block in a Simulink® model and specify a nondefault channel mapping.



Copyright 2016 The MathWorks, Inc.

1. Run the simulation. The Audio Device Writer sends a stereo audio stream to your computer's default audio output device. If you are using a stereo audio output device, such as headphones, you can hear a tone from one speaker and noise from the other speaker.
2. Specify a nondefault channel mapping:
 - a. Stop the simulation.
 - b. Open the Audio Device Writer block to modify parameters.
 - c. On the **Advanced** tab, clear the **Use default channel mapping** parameter.
 - d. Specify the **Device output channels** in reverse order: `[2, 1]`. If you are using a stereo output device, such as headphones, you hear that the noise and tone have switched speakers.

See Also

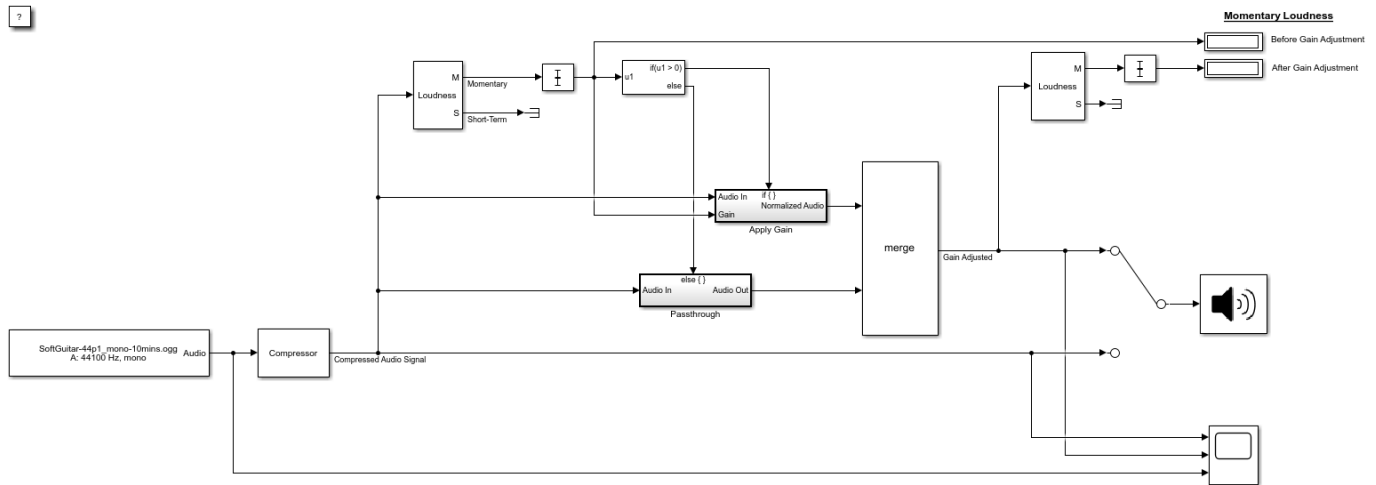
Audio Device Writer | Random Source | Sine Wave | Vector Concatenate, Matrix Concatenate

More About

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

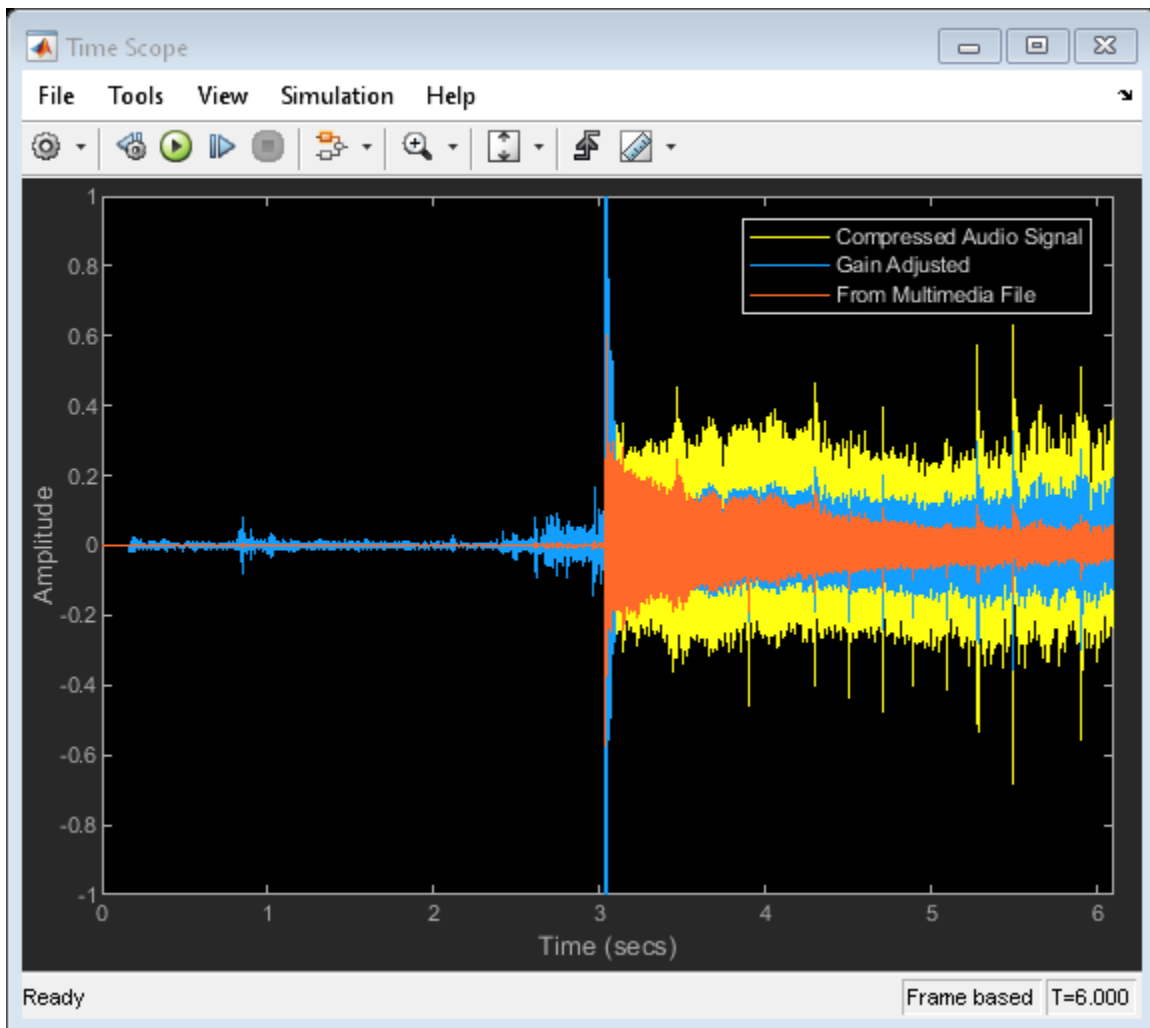
Trigger Gain Control Based on Loudness Measurement

This model enables you to apply dynamic range compression to an audio signal while staying inside a preset loudness range. In this model, a Compressor block increases the loudness and decreases the dynamic range of an audio signal. A Loudness Meter block calculates the momentary loudness of the compressed audio signal. If momentary loudness crosses a -23 LUFS threshold, an enabled subsystem applies gain to lower the corresponding level of the audio signal.



Copyright 2016 The MathWorks, Inc.

1. Open the Time Scope and Compressor blocks.
2. Run the model. To switch between listening to the compressed signal with and without gain adjustment, double-click the switch.
3. To observe the effect of compression on loudness, tune the Compressor block parameters and view the compressed audio signal on the Time Scope block.



See Also

Blocks

Audio Device Writer | Compressor | From Multimedia File | Loudness Meter | Time Scope

Objects

loudnessMeter

Functions

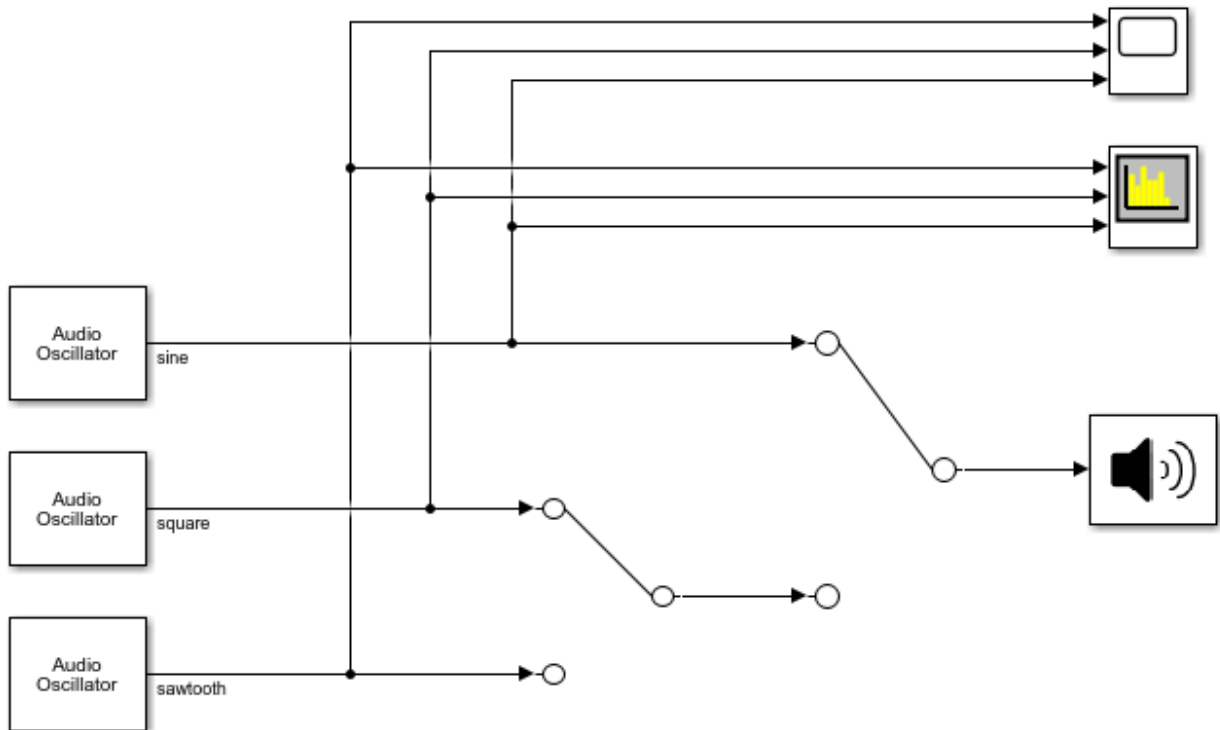
integratedLoudness

More About

- “Loudness Normalization in Accordance with EBU R 128 Standard” on page 1-158

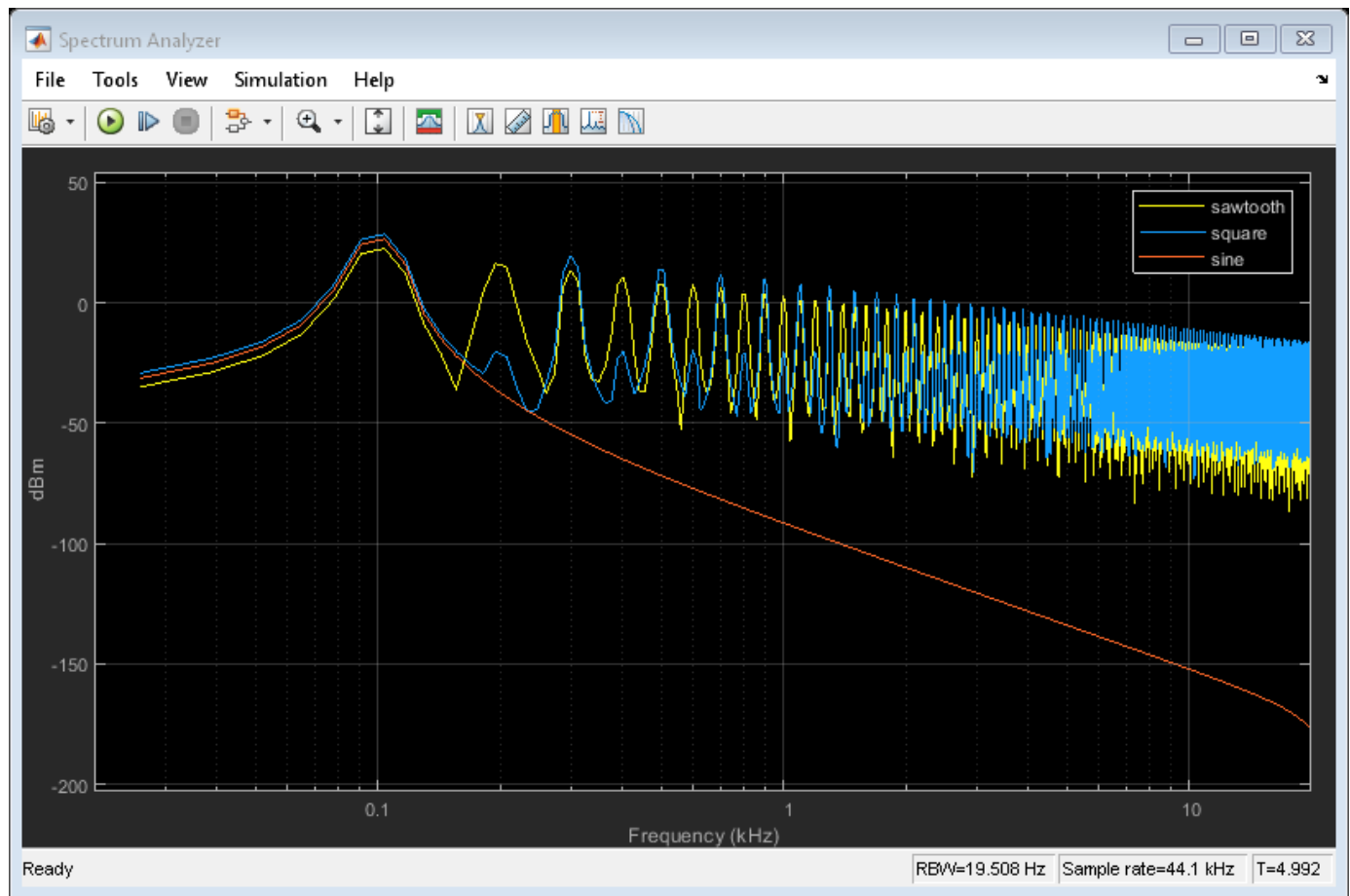
Generate Variable-Frequency Tones in Simulink

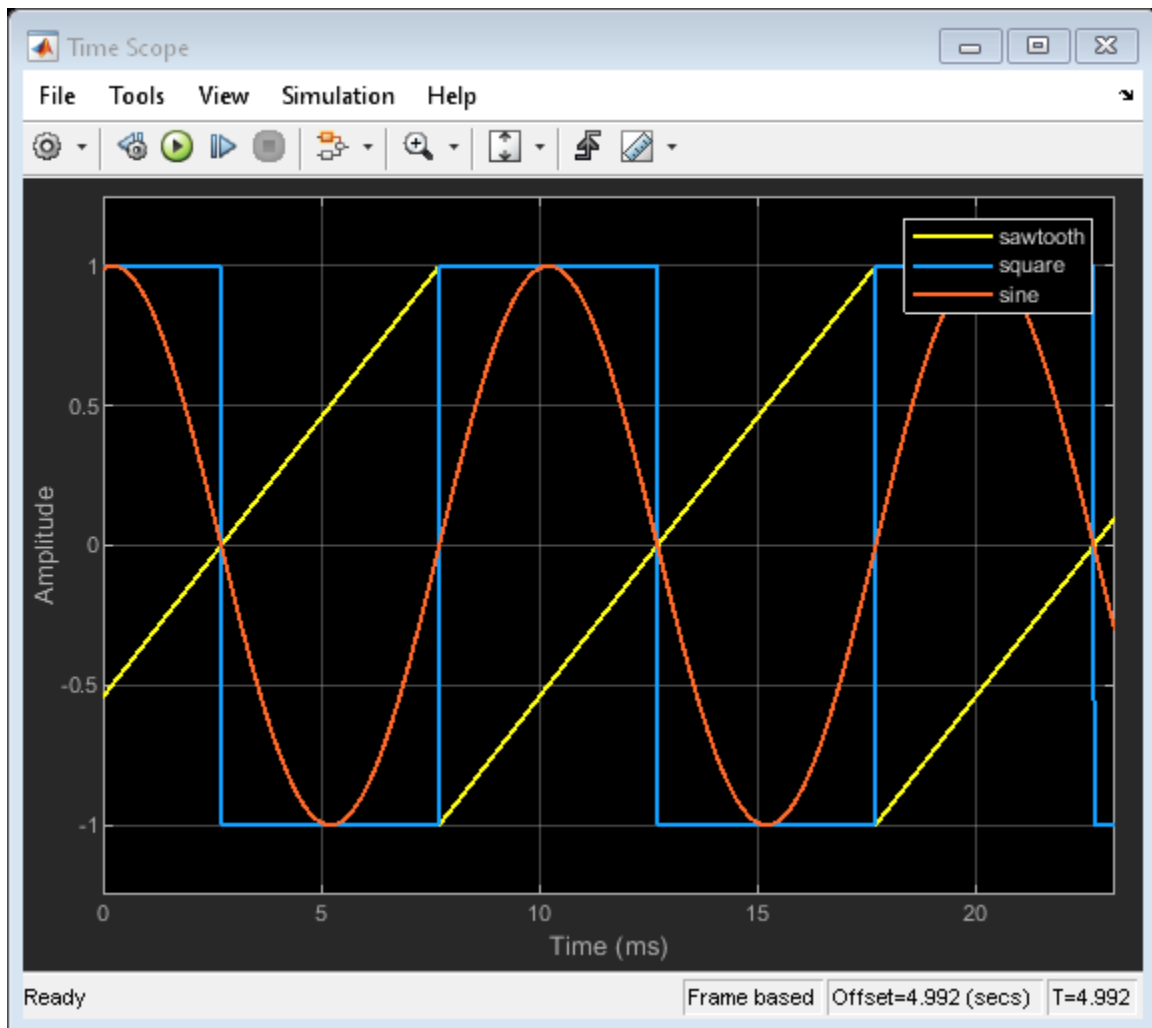
Examine the Audio Oscillator block in a Simulink® model and tune the parameters.



Copyright 2019 The MathWorks, Inc.

1. Run the simulation. Listen to the tone from the Audio Oscillator block generating a sine wave. Visualize the spectrums of all three waveforms on the Spectrum Analyzer. Visualize the waveforms on the Time Scope.
2. Toggle the manual switches to listen to the square and sawtooth waves.
3. Open any of the Audio Oscillator blocks and modify the Frequency (Hz) or Amplitude parameters to hear the effect and visualize the effect on the Spectrum Analyzer and Time Scope.

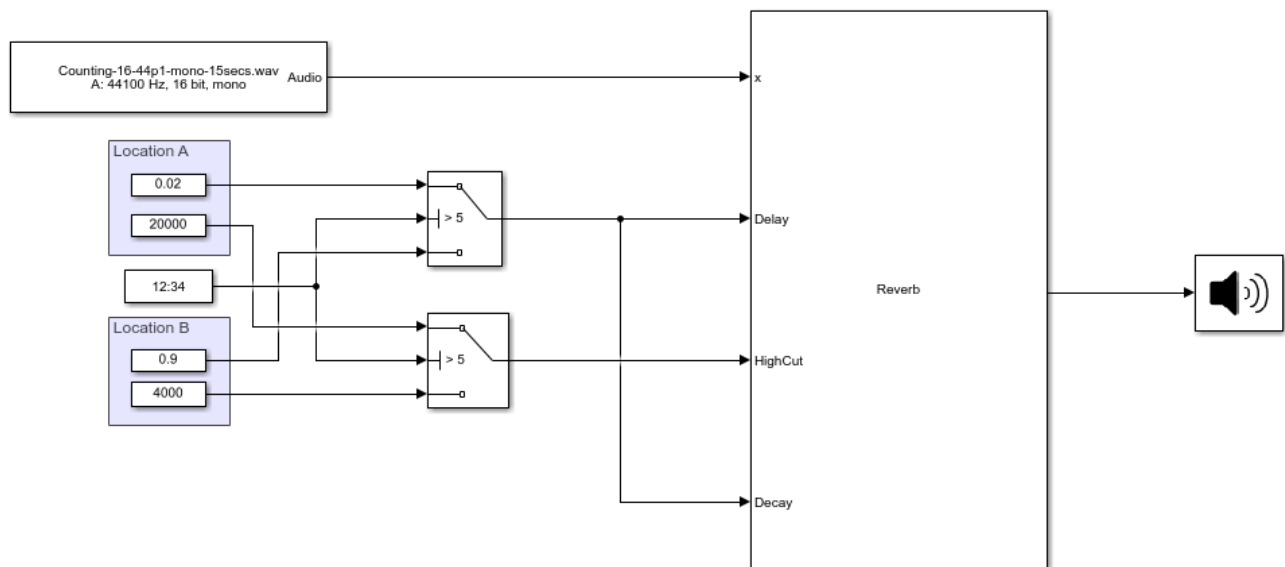




Trigger Reverberation Parameters

Examine the Reverberator block in a Simulink® model where the reverberation parameters are triggered by time.

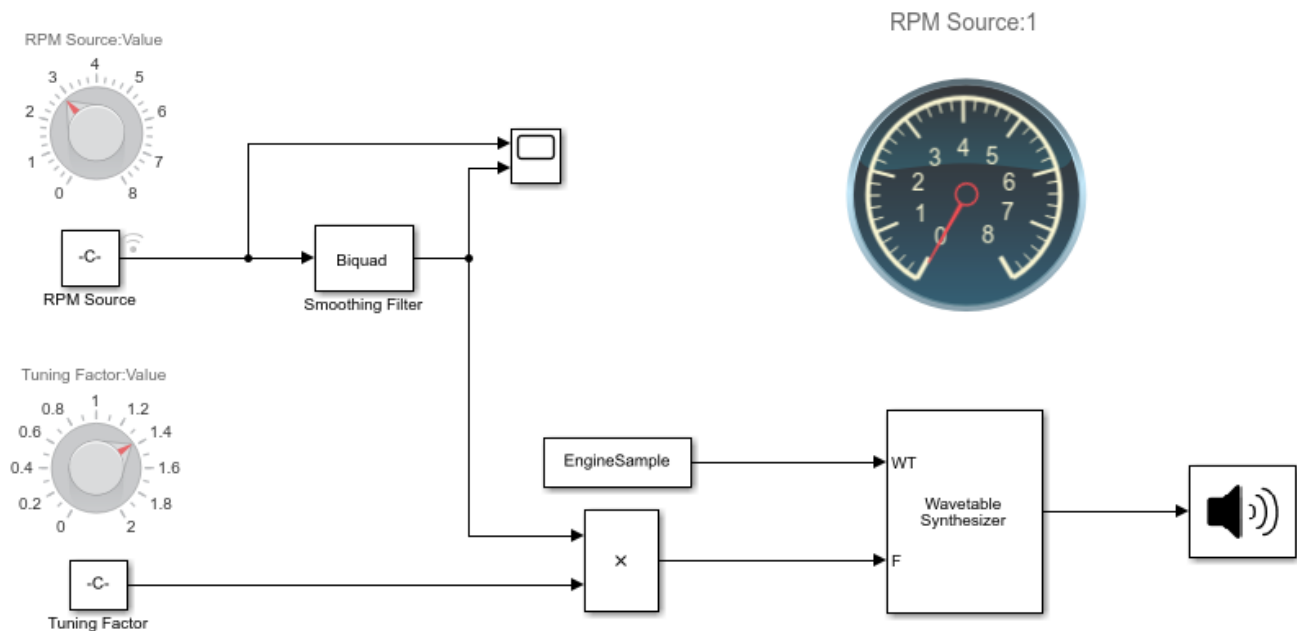
Run the simulation. Listen to the audio signal with the reverberation parameters set to Location A. After 5 seconds, the switches change to the reverberation parameters of Location B.



Copyright 2019 The MathWorks, Inc.

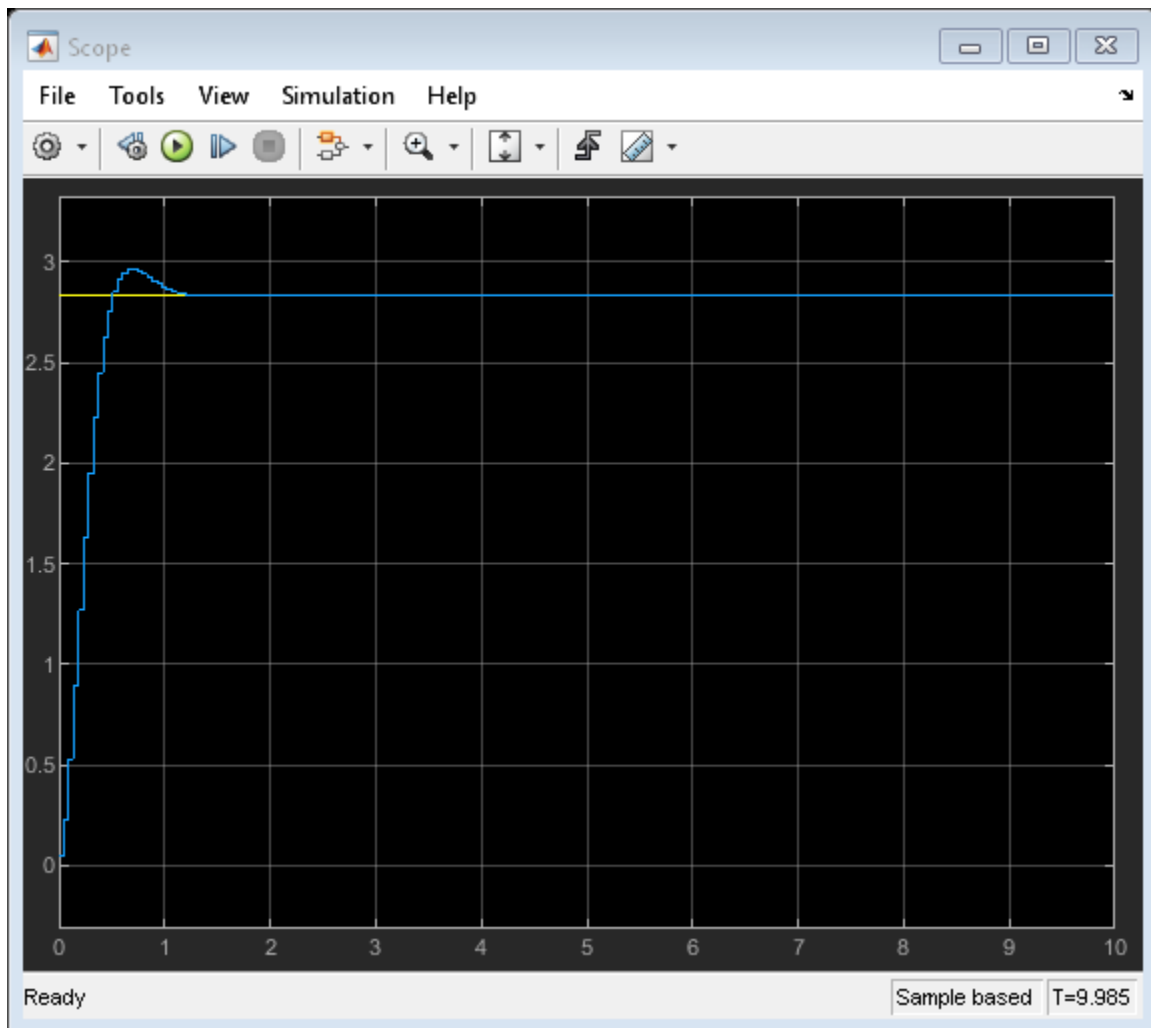
Model Engine Noise

In this model, the Wavetable Synthesizer block is used to synthesize realistic engine noise. Such a system may be found in a vehicle where artificial engine noise enhancement is desired. The wavetable sample is a real-world engine recorded at an unspecified RPM.



Copyright 2019 The MathWorks, Inc.

1. Run the simulation. Listen to the engine sound output from the Wavetable Synthesizer.
2. Tune the RPM source to adjust the perceived RPM of the generated engine sound. The RPM source is lowpass smoothed using a Biquad filter, so that the engine sound ramps in a realistic fashion. Visualize the RPM source before and after smoothing on a Scope.



3. The tuning factor can be used to increase or decrease the overall range of output frequencies. This is used because the wavetable sample RPM is unknown and the sound range might require calibration.

Real-Time Parameter Tuning

Real-Time Parameter Tuning

Parameter tuning is the ability to modify parameters of your audio system in real time while streaming an audio signal. In algorithm development, tunable parameters enable you to quickly prototype and test various parameter configurations. In deployed applications, tunable parameters enable users to fine-tune general algorithms for specific purposes, and to react to changing dynamics.

Audio Toolbox is optimized for parameter tuning in a real-time audio stream. The System objects, blocks, and audio plugins provide various tunable parameters, including sample rate and frame size, making them robust tools when used in an audio stream loop.

To optimize your use of Audio Toolbox, package your audio processing algorithm as an audio plugin. Packaging your audio algorithm as an audio plugin enables you to graphically tune your algorithm using `parameterTuner` or **Audio Test Bench**:

- **Audio Test Bench** -- Creates a user interface (UI) for tunable parameters, enables you to specify input and output from your audio stream loop, and provides access to analysis tools such as the time scope and spectrum analyzer. Packaging your code as an audio plugin also enables you to quickly synchronize your parameters with MIDI controls.
- `parameterTuner` -- Creates a UI for tunable parameters that can be used from any MATLAB programmatic environment. You can customize your parameter controls to render as knobs, sliders, rocker switches, toggle switches, check boxes, or drop-downs. You can also define a custom background color, background image, or both. You can then place your audio plugin in an audio processing loop in a programmatic environment such as a script, and then tune parameters while the loop executes.

For more information, see “Audio Plugins in MATLAB”.

Other methods to create UIs in MATLAB include:

- App Designer -- Development environment for a large set of interactive controls with support for 2-D plots. See “Create and Run a Simple App Using App Designer” for more information.
- Programmatic workflow -- Use MATLAB functions to define your app element-by-element. This tutorial uses a programmatic approach.

See “Ways to Build Apps” for a more detailed list of the costs and benefits of the different approaches to parameter tuning.

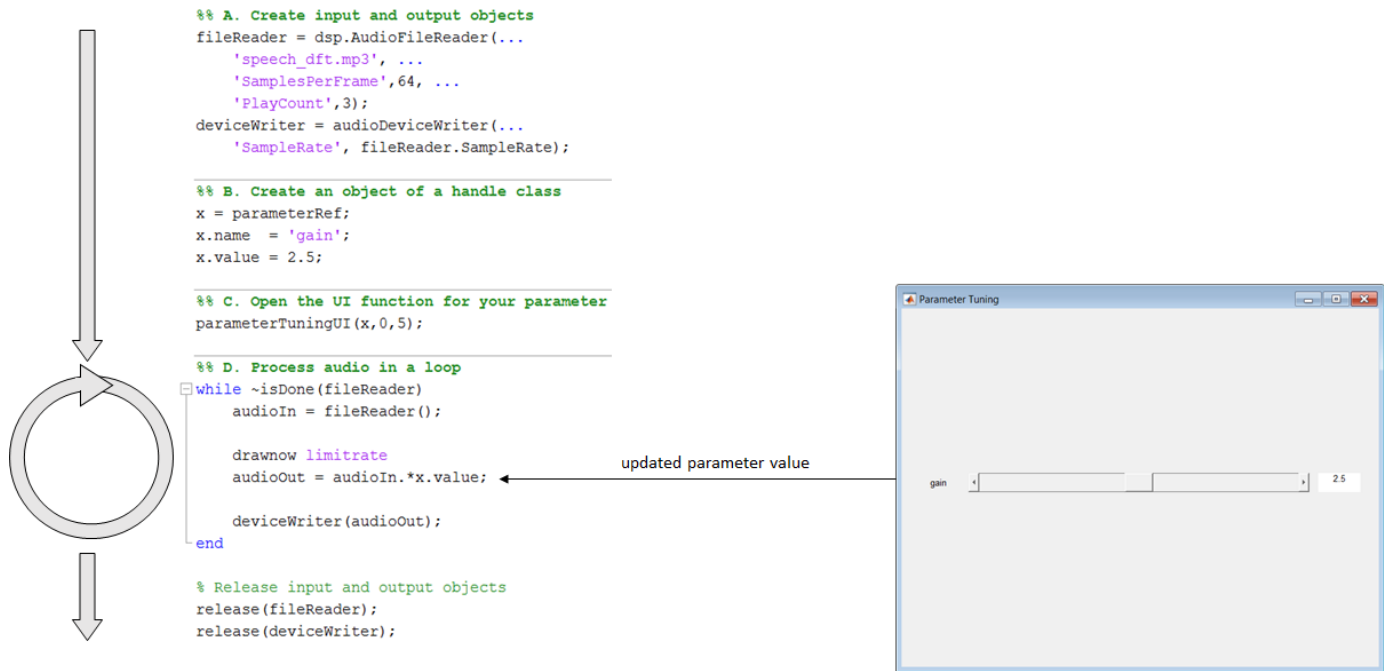
Programmatic Parameter Tuning

If you can not package your algorithm as an audio plugin, you can create a tuning UI using basic MATLAB techniques.

This tutorial contains three files:

- 1 `parameterRef` -- Class definition that contains tunable parameters
- 2 `parameterTuningUI` -- Function that creates a UI for parameter tuning
- 3 `AudioProcessingScript` -- Script for audio processing

Inspect the diagram for an overview of how real-time parameter tuning is implemented. To implement real-time parameter tuning, walk through the example for explanations and step-by-step instructions.



1. Create Class with Tunable Parameters

To tune a parameter in an audio stream loop using a UI, you need to associate the parameter with the position of a UI widget. To associate a parameter with a UI widget, make the parameter an object of a handle class. Objects of handle classes are passed by reference, meaning that you can modify the value of the object in one place and use the updated value in another. For example, you can modify the value of the object using a slider on a figure and use the updated value in an audio processing loop.

Save the `parameterRef` class definition file to your current folder.

```

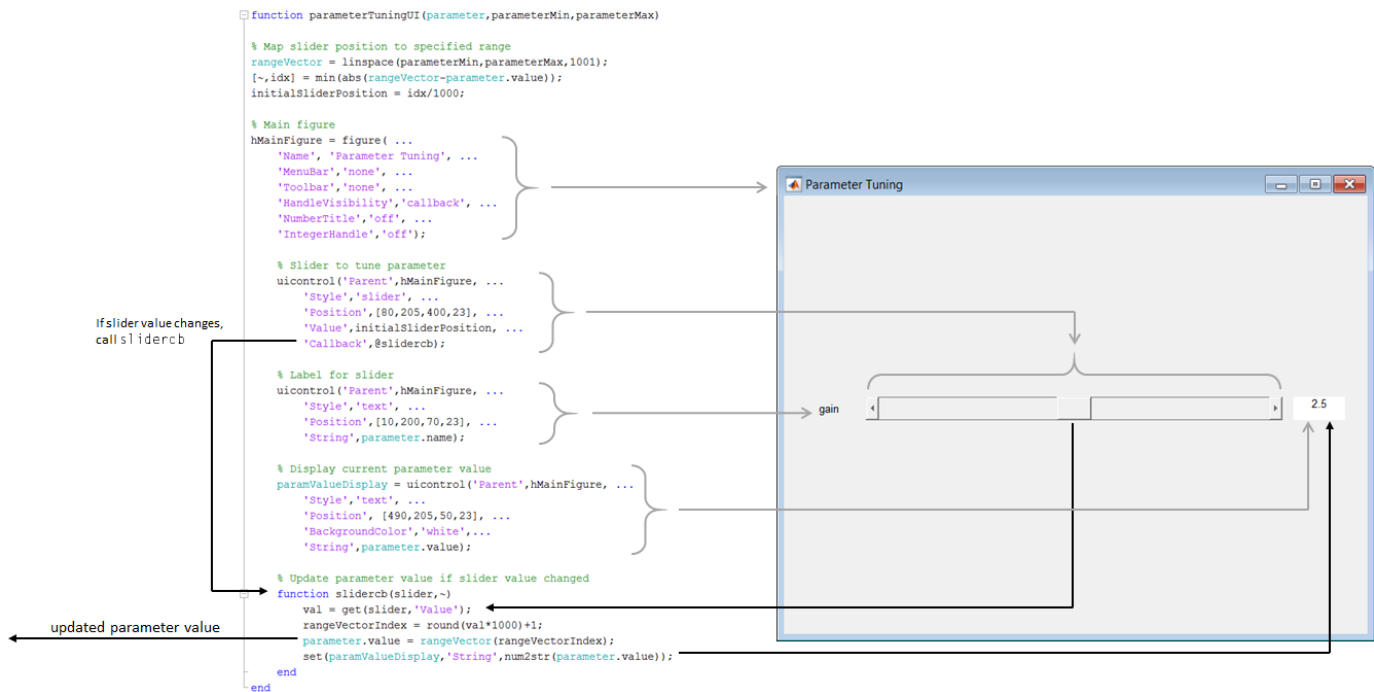
classdef parameterRef < handle
    properties
        name
        value
    end
end

```

Objects of the `parameterRef` class have a name and value. The name is for display purposes on the UI. You use the value for tuning.

2. Create Function to Generate a UI

The `parameterTuningUI` function accepts your parameter, specified as an object handle, and the desired range. The function creates a figure with a slider associated with your parameter. The nested function, `slidercb`, is called whenever the slider position changes. The slider callback function maps the position of the slider to the parameter range, updates the value of the parameter, and updates the text on the UI. You can easily modify this function to tune multiple parameters in the same UI.



Save parameterTuningUI to Current Folder

Open parameterTuningUI and save it to your current folder.

```
function parameterTuningUI(parameter,parameterMin,parameterMax)
```

```
% Map slider position to specified range
rangeVector = linspace(parameterMin,parameterMax,1001);
[~,idx] = min(abs(rangeVector-parameter.value));
initialSliderPosition = idx/1000;
```

```
% Main figure
hMainFigure = figure( ...
    'Name','Parameter Tuning', ...
    'MenuBar','none', ...
    'ToolBar','none', ...
    'HandleVisibility','callback', ...
    'NumberTitle','off', ...
    'IntegerHandle','off');
```

```
% Slider to tune parameter
uicontrol('Parent',hMainFigure, ...
    'Style','slider', ...
    'Position',[80,205,400,23], ...
    'Value',initialSliderPosition, ...
    'Callback',@slidercb);
```

```
% Label for slider
uicontrol('Parent',hMainFigure, ...
    'Style','text', ...
    'Position',[10,200,70,23], ...
    'String',parameter.name);
```

```

% Display current parameter value
paramValueDisplay = uicontrol('Parent',hMainFigure, ...
    'Style','text', ...
    'Position', [490,205,50,23], ...
    'BackgroundColor','white', ...
    'String',parameter.value);

% Update parameter value if slider value changed
function slidercb(slider,~)
    val = get(slider,'Value');
    rangeVectorIndex = round(val*1000)+1;
    parameter.value = rangeVector(rangeVectorIndex);
    set(paramValueDisplay,'String',num2str(parameter.value));
end
end

```

3. Create Script for Audio Processing

The audio processing script:

- A** Creates input and output objects for an audio stream loop.
- B** Creates an object of the handle class, `parameterRef`, that stores your parameter name and value.
- C** Calls the tuning UI function, `parameterTuningUI`, with your parameter and the parameter range.
- D** Processes the audio in a loop. You can tune your parameter, `x`, in the audio stream loop.

Run AudioProcessingScript

Open `AudioProcessingScript`, save it to your current folder, and then run the file.

```

%% A. Create input and output objects
fileReader = dsp.AudioFileReader( ...
    'speech_dft.mp3', ...
    'SamplesPerFrame',64, ...
    'PlayCount',3);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);

%% B. Create an object of a handle class
x = parameterRef;
x.name = 'gain';
x.value = 2.5;

%% C. Open the UI function for your parameter
parameterTuningUI(x,0,5);

%% D. Process audio in a loop
while ~isDone(fileReader)
    audioIn = fileReader();

    drawnow limitrate
    audioOut = audioIn.*x.value;

    deviceWriter(audioOut);
end

```

```
% Release input and output objects  
release(fileReader)  
release(deviceWriter)
```

While the script runs, move the position of the slider to update your parameter value and hear the result.

See Also

Audio Test Bench | parameterTuner

More About

- “Real-Time Audio in MATLAB”
- “Audio Plugins in MATLAB”
- “Audio Test Bench Walkthrough” on page 11-2
- “Create and Run a Simple App Using App Designer”
- “Ways to Build Apps”

Tips and Tricks for Plugin Authoring

Tips and Tricks for Plugin Authoring

To author your algorithm as an audio plugin, you must conform to the audio plugin API. When authoring audio plugins in the MATLAB environment, keep these common pitfalls and best practices in mind.

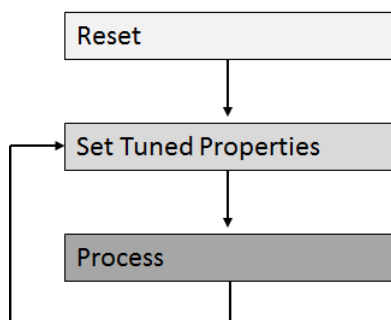
To learn more about audio plugins in general, see “Audio Plugins in MATLAB”.

Avoid Disrupting the Event Queue in MATLAB

When the **Audio Test Bench** runs an audio plugin, it sequentially:

- 1 Calls the reset method
- 2 Sets tunable properties associated with parameters
- 3 Calls the process method

While running, the **Audio Test Bench** calls in a loop the process method and then the `set` methods for tuned properties. The plugin API does not specify the order that the tuned properties are set.



It is possible to disrupt the normal methods timing by interrupting the event queue. Common ways to accidentally interrupt the event queue include using a `plot` or `drawnow` function.

Note `plot` and `drawnow` are only available in the MATLAB environment. `plot` and `drawnow` cannot be included in generated plugins. See “Separate Code for Features Not Supported for Plugin Generation” on page 19-4 for more information.

In the following code snippet, the gain applied to the left and right channels is not the same if the associated `Gain` parameter is tuned during the call to `process`:

```
...  
L = plugin.Gain*in(:,1);  
drawnow  
R = plugin.Gain*in(:,2);  
out = [L,R];  
...
```

See Full Code

```
classdef badPlugin < audioPlugin  
    properties  
        Gain = 0.5;  
    end
```

```

properties (Constant)
    PluginInterface = audioPluginInterface(audioPluginParameter('Gain'));
end
methods
    function out = process(plugin,in)

        L = plugin.Gain*in(:,1);

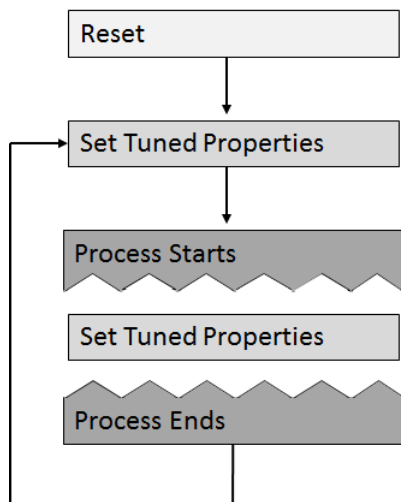
        drawnow

        R = plugin.Gain*in(:,2);

        out = [L,R];
    end
    function set.Gain(plugin,val)
        plugin.Gain = val;
    end
end
end

```

The author interrupts the event queue in the code snippet, causing the `set` methods of properties associated with parameters to be called while the `process` method is in the middle of execution.



Depending on your processing algorithm, interrupting the event queue can lead to inconsistent and buggy behavior. Also, the `set` method might not be explicit, which can make the issue difficult to track down. Possible fixes for the problem of event queue disruption include saving properties to local variables, and moving the queue disruption to the beginning or end of the process method.

Save Properties to Local Variables

You can save tunable property values to local variables at the start of your processing. This technique ensures that the values used during the process method are not updated within a single call to `process`. Because accessing the value of a local variable is cheaper than accessing the value of a property, saving properties to local variables that are accessed multiple times is a best practice.

```

...
gain = plugin.Gain;
L = gain*in(:,1);
drawnow
R = gain*in(:,2);
out = [L,R];
...

```

See Full Code

```
classdef goodPlugin < audioPlugin
    properties
        Gain = 0.5;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(audioPluginParameter('Gain'));
    end
    methods
        function out = process(plugin,in)
            gain = plugin.Gain;

            L = gain*in(:,1);

            drawnow

            R = gain*in(:,2);

            out = [L,R];
        end
        function set.Gain(plugin,val)
            plugin.Gain = val;
        end
    end
end
```

Move Queue Disruption to Bottom or Top of Process Method

You can move the disruption to the event queue to the bottom or top of the process method. This technique ensures that property values are not updated in the middle of the call.

```
...
L = plugin.Gain*in(:,1);
R = plugin.Gain*in(:,2);
out = [L,R];
drawnow
...
```

See Full Code

```
classdef goodPlugin < audioPlugin
    properties
        Gain = 0.5;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(audioPluginParameter('Gain'));
    end
    methods
        function out = process(plugin,in)

            L = plugin.Gain*in(:,1);

            R = plugin.Gain*in(:,2);

            out = [L,R];

            drawnow
        end
        function set.Gain(plugin,val)
            plugin.Gain = val;
        end
    end
end
```

Separate Code for Features Not Supported for Plugin Generation

The MATLAB environment offers functionality not supported for plugin generation. You can mark code to ignore during plugin generation by placing it inside a conditional statement by using `coder.target`.


```

...
    if coder.target('MATLAB')
    ...
    end
...

```

If you generate the plugin using `generateAudioPlugin`, code inside the statement `if coder.target('MATLAB')` is ignored.

For example, `timescope` is not enabled for code generation. If you run the following plugin in MATLAB, you can use the `visualize` function to open a time scope that plots the input and output power per frame.

See Full Example Code

```

classdef pluginWithMATLABOnlyFeatures < audioPlugin
    properties
        Threshold = -10;
    end
    properties (Access = private)
        aCompressor
        aScope
        SamplesPerFrame = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('Threshold','Mapping',{'lin',-60,20}));
    end
    methods
        function plugin = pluginWithMATLABOnlyFeatures
            plugin.aCompressor = compressor;
            setup(plugin.aCompressor,[0,0])
        end
        function out = process(plugin,in)
            out = plugin.aCompressor(in);

            % The contents of this if-statement are ignored during plugin
            % generation.
            if coder.target('MATLAB')
                if ~isempty(plugin.aScope) && isvalid(plugin.aScope)
                    numSamples = size(in,1);

                    % The time scope object is not enabled for
                    % variable-size signals. Call release if the samples
                    % per frame is changed.
                    % Because this code is intended for use in MATLAB only,
                    % it is okay to call release on the time scope object.
                    % Do not call release on a System object in generated
                    % code.
                    if plugin.SamplesPerFrame(1) ~= numSamples
                        release(plugin.aScope)
                        plugin.SamplesPerFrame = numSamples;
                    end

                    power = 20*log10(mean(var(in)))*ones(numSamples,1);
                    adjustedPower = 20*log10(mean(var(out)))*ones(numSamples,1);
                    plugin.aScope([power,adjustedPower]);
                end
            end
        end
        function reset(plugin)
            fs = getSampleRate(plugin);
            plugin.aCompressor.SampleRate = fs;
            reset(plugin.aCompressor)

            % The contents of this if-statement are ignored during plugin
            % generation.
            if coder.target('MATLAB')
                if ~isempty(plugin.aScope)
                    % Because this code is intended for use in MATLAB only,
                    % it is okay to call release on the time scope object.
                    % Do not call release on a System object in generated
                    % code.
                    release(plugin.aScope)
                    plugin.aScope.SampleRate = fs;
                    plugin.aScope.BufferLength = 2*fs;
                end
            end
        end
    end
end

```

```

        end
    end
end
function visualize(plugin)
    % Visualization function. This function is public in the MATLAB
    % environment. Because the plugin does not call this function
    % directly, the function is not part of the code generated by
    % generateAudioPlugin.

    % Create a time scope object for visualization in the MATLAB
    % environment.
    plugin.aScope = timescope( ...
        'SampleRate',getSampleRate(plugin), ...
        'TimeSpan',1, ...
        'YLimits',[-40,0], ...
        'BufferLength',2*getSampleRate(plugin), ...
        'TimeSpanOverrunAction','Scroll', ...
        'YLabel','Power (dB)');
    show(plugin.aScope)
end
function set.Threshold(plugin,val)
    plugin.Threshold = val;
    plugin.aCompressor.Threshold = val;
end
end
end
end

```

Implement Reset Correctly

A common error in audio plugin authoring is misusing the reset method. Valid uses of the reset method include:

- Clearing state
- Passing down calls to reset to component objects
- Updating properties which depend on sample rate

Invalid use of the reset method includes setting the value of any properties associated with parameters. Do not use your reset method to set properties associated with parameters to their initial conditions. Directly setting a property associated with a parameter causes the property to be out of sync with the parameter. For example, the following plugin is an example of incorrect use of the reset method.

```

classdef badReset < audioPlugin
    properties
        Gain = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(audioPluginParameter('Gain'));
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
        function reset(plugin) % <-- Incorrect use of reset method.
            plugin.Gain = 1; % <-- Never set values of a property that is
                             % associated with a plugin parameter.
        end
    end
end
end

```

Implement Plugin Composition Correctly

If your plugin is composed of other plugins, then you must pass down the sample rate and calls to reset to the component plugins. Call `setSampleRate` in the reset method to pass down the sample rate to the component plugins. To tune parameters of the component plugins, create an audio plugin interface in the composite plugin for tunable parameters of the component plugins. Then pass down the values in the `set` methods for the associated properties. The following is an example of plugin composition that was constructed using best practices.

Plugin Composition Using Basic Plugins

```
classdef compositePlugin < audioPlugin
    properties
        PhaserQ = 1.6;
        EchoGain = 0.5;
    end
    properties (Access = private)
        aEcho
        aPhaser
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('PhaserQ', ...
                'DisplayName','Phaser Q', ...
                'Mapping',{'lin',0.5, 25}), ...
            audioPluginParameter('EchoGain', ...
                'DisplayName','Gain'));
    end
    methods
        function plugin = compositePlugin
            % Construct your component plugins in the composite plugin's
            % constructor.
            plugin.aPhaser = audiopluginexample.Phaser;
            plugin.aEcho = audiopluginexample.Echo;
        end
        function out = process(plugin,in)
            % Call the process method of your component plugins inside the
            % call to the process method of your composite plugin.
            x = process(plugin.aPhaser,in);
            y = process(plugin.aEcho,x);
            out = y;
        end
        function reset(plugin)
            % Use the setSampleRate method to set the sample rate of
            % component plugins and pass the call to reset down.
            fs = getSampleRate(plugin);

            setSampleRate(plugin.aPhaser, fs)
            setSampleRate(plugin.aEcho, fs)

            reset(plugin.aPhaser)
            reset(plugin.aEcho);
        end
        % Use the set method of your properties to pass down property
        % values to your component plugins.
        function set.PhaserQ(plugin,val)
            plugin.PhaserQ = val;
            plugin.aPhaser.QualityFactor = val;
        end
        function set.EchoGain(plugin,val)
            plugin.EchoGain = val;
            plugin.aEcho.Gain = val;
        end
    end
end
end
```

Plugin composition using System objects has these key differences from plugin composition using basic plugins.

- Immediately call **setup** on your component System object after it is constructed. Construction and setup of the component object occurs inside the constructor of the composite plugin.
- If your component System object requires sample rate information, then it has a sample rate property. Set the sample rate property in the reset method.

Plugin Composition Using System Objects

```
classdef compositePluginWithSystemObjects < audioPlugin
    properties
        CrossoverFrequency = 100;
        CompressorThreshold = -40;
    end
    properties (Access = private)
        aCrossoverFilter
        aCompressor
    end
end
```

```

properties (Constant)
    PluginInterface = audioPluginInterface( ...
        audioPluginParameter('CrossoverFrequency', ...
            'DisplayName','Crossover Frequency', ...
            'Mapping',{'lin',50, 200}), ...
        audioPluginParameter('CompressorThreshold', ...
            'DisplayName','Compressor Threshold', ...
            'Mapping',{'lin',-100,0}));
end
methods
    function plugin = compositePluginWithSystemObjects
        % Construct your component System objects within the composite
        % plugin's constructor. Call setup immediately after
        % construction.
        %
        % The audio plugin API requires plugins to declare the number
        % of input and output channels in the plugin interface. This
        % plugin uses the default 2-in 2-out configuration. Call setup
        % with a sample input that has the same number of channels as
        % defined in the plugin interface.
        %
        sampleInput = zeros(1,2);

        plugin.aCrossoverFilter = crossoverFilter;
        setup(plugin.aCrossoverFilter,sampleInput)

        plugin.aCompressor = compressor;
        setup(plugin.aCompressor,sampleInput)
    end
    function out = process(plugin,in)
        % Call your component System objects inside the call to
        % process of your composite plugin.
        [band1,band2] = plugin.aCrossoverFilter(in);
        band1Compressed = plugin.aCompressor(band1);
        out = band1Compressed + band2;
    end
    function reset(plugin)
        % Set the sample rate properties of your component System
        % objects.
        fs = getSampleRate(plugin);

        plugin.aCrossoverFilter.SampleRate = fs;
        plugin.aCompressor.SampleRate = fs;

        reset(plugin.aCrossoverFilter)
        reset(plugin.aCompressor);
    end
    % Use the set method of your properties to pass down property
    % values to your component System objects.
    function set.CrossoverFrequency(plugin,val)
        plugin.CrossoverFrequency = val;
        plugin.aCrossoverFilter.CrossoverFrequencies = val;
    end
    function set.CompressorThreshold(plugin,val)
        plugin.CompressorThreshold = val;
        plugin.aCompressor.Threshold = val;
    end
end
end
end

```

Address "A set method for a non-Dependent property should not access another property" Warning in Plugin

It is recommended that you suppress the warning when authoring audio plugins.

The following code snippet follows the plugin authoring best practice for processing changes in parameter property Cutoff.

```

classdef highpassFilter < audioPlugin
    ...
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('Cutoff', ...
                'Label','Hz',...
                'Mapping',{'log',20,2000}));
    end
end

```

```

methods
    function y = process(plugin,x)
        [y,plugin.State] = filter(plugin.B,plugin.A,x,plugin.State);
    end

    function set.Cutoff(plugin,val)
        plugin.Cutoff = val;
        [plugin.B,plugin.A] = highpassCoeffs(plugin,val,getSampleRate(plugin)); % <<<< warning occurs here
    end
end
...
end

```

See Full Code Example

```

classdef highpassFilter < audioPlugin
    %-----
    % Public Properties - End user interacts with these
    %-----
    properties
        Cutoff = 20;
    end

    %-----
    % Private Properties - Used for internal storage
    %-----
    properties (Access = private)
        State = zeros(2);
        B      = zeros(1,3);
        A      = zeros(1,3);
    end

    %-----
    % Constant Properties - Used to define plugin interface
    %-----
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('Cutoff', ...
                'Label','Hz', ...
                'Mapping',{'log',20,2000}));
    end

    methods
        %-----
        % Main processing function
        %-----
        function y = process(plugin,x)
            [y,plugin.State] = filter(plugin.B,plugin.A,x,plugin.State);
        end

        %-----
        % Set Method
        %-----
        function set.Cutoff(plugin,val)
            plugin.Cutoff = val;
            [plugin.B,plugin.A] = highpassCoeffs(plugin,val,getSampleRate(plugin)); % <<<< warning occurs here
        end

        %-----
        % Reset Method
        %-----
        function reset(plugin)
            plugin.State = zeros(2);
            [plugin.B,plugin.A] = highpassCoeffs(plugin,plugin.Cutoff,getSampleRate(plugin));
        end
    end

    methods (Access = private)
        %-----
        % Calculate Filter Coefficients
        %-----
        function [B,A] = highpassCoeffs(~,fc,fs)
            w0 = 2*pi*fc/fs;
            alpha = sin(w0)/sqrt(2);
            cosw0 = cos(w0);
            norm = 1/(1+alpha);
            B = (1 + cosw0)*norm * [.5 -1 .5];
            A = [1 -2*cosw0*norm (1 - alpha)*norm];
        end
    end
end

```

The `highpassCoeffs` function might be expensive, and should be called only when necessary. You do not want to call `highpassCoeffs` in the `process` method, which runs in the real-time audio processing loop. The logical place to call `highpassCoeffs` is in `set.Cutoff`. However, `mlint` shows a warning for this practice. The warning is intended to help you avoid initialization order issues when saving and loading classes. See “Avoid Property Initialization Order Dependency” for more details. The solution recommended by the warning is to create a dependent property with a `get` method and compute the value there. However, following the recommendation complicates the design and pushes the computation back into the real-time processing method, which you are trying to avoid.

You might also incur the warning when correctly implementing plugin composition. For an example of a correct implementation of composition, see “Implement Plugin Composition Correctly” on page 19-6.

Use System Object That Does Not Support Variable-Size Signals

The audio plugin API requires audio plugins to support variable-size inputs and outputs. For a partial list of System objects that support variable-size signals, see “Variable-Size Signal Support DSP System Objects”. You might encounter issues if you attempt to use objects that do not support variable-size signals in your plugin.

For example, `dsp.AnalyticSignal` does not support variable-size signals. The `BrokenAnalyticSignalTransformer` plugin uses a `dsp.AnalyticSignal` object incorrectly and fails the `validateAudioPlugin` test bench:

```
validateAudioPlugin BrokenAnalyticSignalTransformer

Checking plug-in class 'BrokenAnalyticSignalTransformer'... passed.
Generating testbench file 'testbench_BrokenAnalyticSignalTransformer.m'... done.
Running testbench...
Error using dsp.AnalyticSignal/parenReference
Changing the size on input 1 is not allowed without first calling the release() method.

Error in BrokenAnalyticSignalTransformer/process (line 13)
    analyticSignal = plugin.Transformer(in);

Error in testbench_BrokenAnalyticSignalTransformer (line 61)
    o1 = process(plugin, in(:,1));

Error in validateAudioPlugin
```

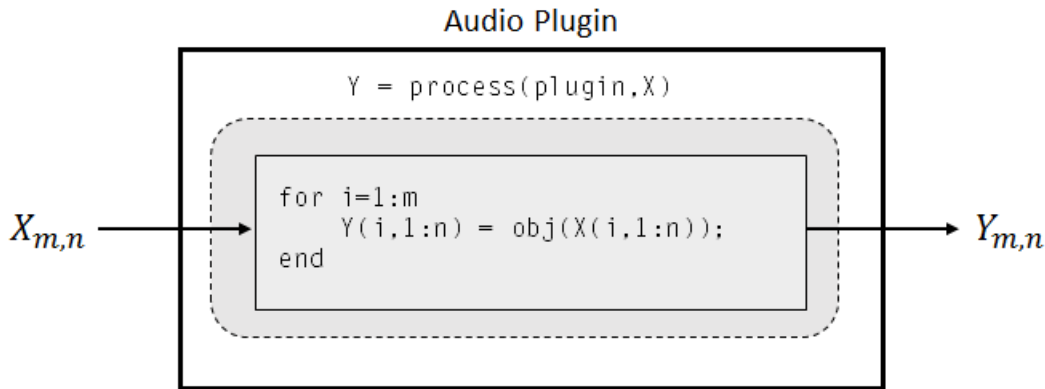
See BrokenAnalyticSignalTransformer Code

```
classdef BrokenAnalyticSignalTransformer < audioPlugin
    properties (Access = private)
        Transformer
    end
    properties (Constant)
        PluginInterface = audioPluginInterface('InputChannels',1,'OutputChannels',2);
    end
    methods
        function plugin = BrokenAnalyticSignalTransformer
            plugin.Transformer = dsp.AnalyticSignal;
        end
        function out = process(plugin,in)
            analyticSignal = plugin.Transformer(in);
            realPart = real(analyticSignal);
            imaginaryPart = imag(analyticSignal);
            out = [realPart,imaginaryPart];
        end
    end
end
```

If you want to use the functionality of a System object that does not support variable-size signals, you can buffer the input and output of the System object, or always call the object with one sample.

Always Call the Object with One Sample

You can create a loop around your call to an object. The loop iterates for the number of samples in your variable frame size. The call to the object inside the loop is always a single sample.



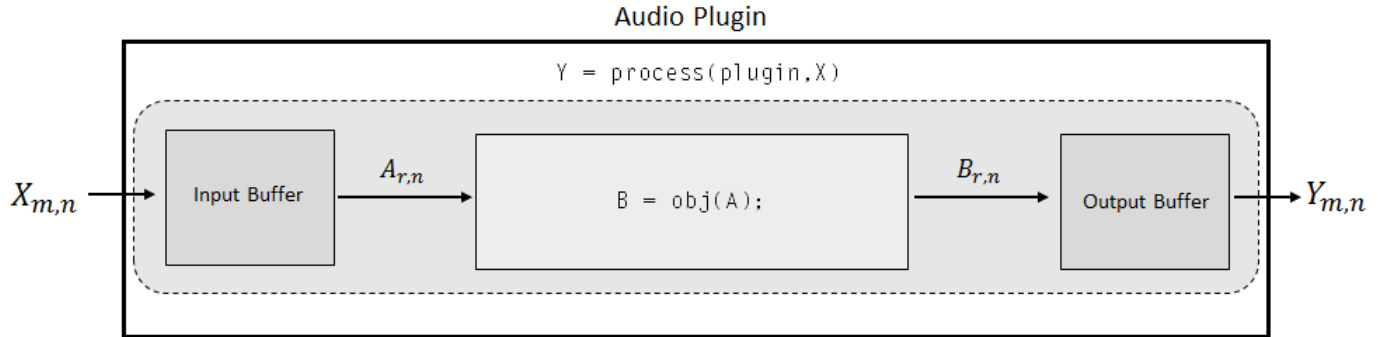
See Full Code Example

```
classdef ExpensiveAnalyticSignalTransformer < audioPlugin
    properties (Access = private)
        Transformer
    end
    properties (Constant)
        PluginInterface = audioPluginInterface('InputChannels',1,'OutputChannels',2);
    end
    methods
        function plugin = ExpensiveAnalyticSignalTransformer
            plugin.Transformer = dsp.AnalyticSignal;
        end
        function out = process(plugin,in)
            analyticSignal = complex(zeros(size(in,1),1),0);
            for i = 1:size(in,1)
                analyticSignal(i,:) = plugin.Transformer(in(i,1));
            end
            out = [real(analyticSignal),imag(analyticSignal)];
        end
    end
end
```

Note Depending on your implementation and the particular object, calling an object sample by sample in a loop might result in significant computational cost.

Buffer Input and Output of Object

You can buffer the input to your object to a consistent frame size, and then buffer the output of your object back to the original frame size. The `dsp.AsyncBuffer` System object is well-suited for this task.



See Full Code Example

```

classdef DelayedAnalyticSignalTransformer < audioPlugin
    properties (Access = private)
        Transformer
        InputBuffer
        OutputBuffer
    end
    properties (Constant)
        PluginInterface = audioPluginInterface('InputChannels',1,'OutputChannels',2);
        MinSampleDelay = 256;
    end
    methods
        function plugin = DelayedAnalyticSignalTransformer
            plugin.Transformer = dsp.AnalyticSignal;
            setup(plugin.Transformer,ones(plugin.MinSampleDelay,1));

            plugin.InputBuffer = dsp.AsyncBuffer;
            setup(plugin.InputBuffer,1);

            plugin.OutputBuffer = dsp.AsyncBuffer;
            setup(plugin.OutputBuffer,[1,1]);
        end
        function out = process(plugin,in)
            write(plugin.InputBuffer,in);

            while plugin.InputBuffer.NumUnreadSamples >= plugin.MinSampleDelay
                x = read(plugin.InputBuffer,plugin.MinSampleDelay);
                analyticSignal = plugin.Transformer(x(1:plugin.MinSampleDelay,:));
                write(plugin.OutputBuffer,[real(analyticSignal),imag(analyticSignal)]);
            end

            if plugin.OutputBuffer.NumUnreadSamples >= size(in,1)
                out = read(plugin.OutputBuffer,size(in,1));
            else
                out = zeros(size(in,1),2);
            end
        end
        function reset(plugin)
            reset(plugin.Transformer)
            reset(plugin.InputBuffer)
            reset(plugin.OutputBuffer)
        end
    end
end
end
  
```

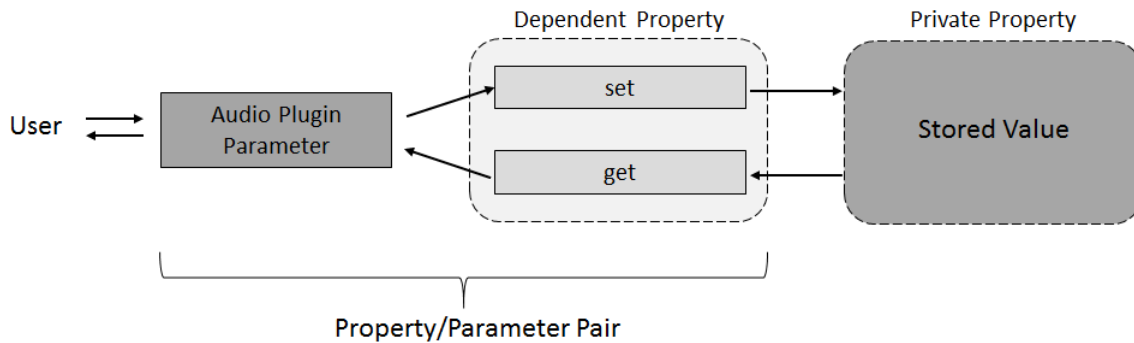
Note Use of the asynchronous buffering object forces a minimum latency of your specified frame size.

Using Enumeration Parameter Mapping

It is often useful to associate a property with a set of strings or character vectors. However, restrictions on plugin generation require cached values, such as property values, to have a static size.

To work around this issue, you can use a separate enumeration class that maps the strings to the enumerations, as described in the `audioPluginParameter` documentation.

Alternatively, if you want to avoid writing an enumeration class and keep all your code in one file, you can use a dependent property to map your parameter names to a set of values. In this scenario, you map your enumeration value to a value that you can cache.



See Full Code Example

```

classdef pluginWithEnumMapping < audioPlugin
    properties (Dependent)
        Mode = '+6 dB';
    end
    properties (Access = private)
        pMode = 1; % '+6 dB'
    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.pMode)
                case 1
                    out = in * 2;
                case 2
                    out = in / 2;
                case 3
                    out = zeros(size(in));
                otherwise % case 4
                    out = rand(size(in)) - 0.5;
            end
        end
        function set.Mode(plugin,val)
            validatestring(val',{'+6 dB','-6 dB','silence','white noise'},'set.Mode','Mode');
            switch val
                case '+6 dB'
                    plugin.pMode = 1;
                case '-6 dB'
                    plugin.pMode = 2;
                case 'silence'
                    plugin.pMode = 3;
                otherwise % 'white noise'
                    plugin.pMode = 4;
            end
        end
        function out = get.Mode(plugin)
            switch plugin.pMode
                case 1
                    out = '+6 dB';
                case 2
                    out = '-6 dB';
                case 3
                    out = 'silence';
                otherwise % case 4
                    out = 'white noise';
            end
        end
    end
end

```

```
end  
end  
end
```

See Also

More About

- “Audio Plugins in MATLAB”
- “Audio Plugin Example Gallery” on page 12-2
- “Export a MATLAB Plugin to a DAW”

Spectral Descriptors Chapter

Spectral Descriptors

Audio Toolbox™ provides a suite of functions that describe the shape, sometimes referred to as *timbre*, of audio. This example defines the equations used to determine the spectral features, cites common uses of each feature, and provides examples so that you can gain intuition about what the spectral descriptors are describing.

Spectral descriptors are widely used in machine and deep learning applications, and perceptual analysis. Spectral descriptors have been applied to a range of applications, including:

- Speaker identification and recognition [21 on page 20-0]
- Acoustic scene recognition [11 on page 20-0] [17 on page 20-0]
- Instrument recognition [22 on page 20-0]
- Music genre classification [16 on page 20-0] [18 on page 20-0]
- Mood recognition [19 on page 20-0] [20 on page 20-0]
- Voice activity detection [5 on page 20-0] [7 on page 20-0] [8 on page 20-0] [10 on page 20-0] [12 on page 20-0] [13 on page 20-0]

Spectral Centroid

The spectral centroid (`spectralCentroid`) is the frequency-weighted sum normalized by the unweighted sum [1 on page 20-0]:

$$\mu_1 = \frac{\sum_{k=b_1}^{b_2} f_k s_k}{\sum_{k=b_1}^{b_2} s_k}$$

where

- f_k is the frequency in Hz corresponding to bin k .
- s_k is the spectral value at bin k . The magnitude spectrum and power spectrum are both commonly used.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral centroid.

The spectral centroid represents the "center of gravity" of the spectrum. It is used as an indication of *brightness* [2 on page 20-0] and is commonly used in music analysis and genre classification. For example, observe the jumps in the centroid corresponding to high hat hits in the audio file.

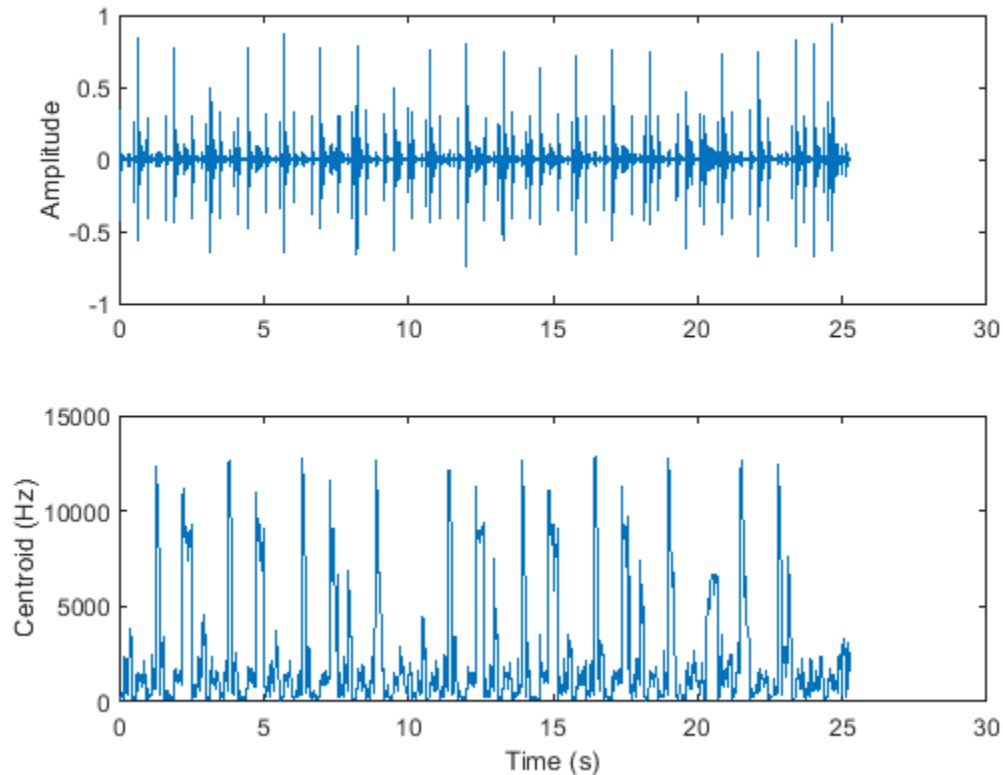
```
[audio,fs] = audioread('FunkyDrums-44p1-stereo-25secs.mp3');
audio = sum(audio,2)/2;
```

```
centroid = spectralCentroid(audio,fs);
```

```
subplot(2,1,1)
t = linspace(0,size(audio,1)/fs,size(audio,1));
plot(t,audio)
ylabel('Amplitude')
```

```
subplot(2,1,2)
t = linspace(0,size(audio,1)/fs,size(audio,1));
plot(t,centroid)
```

```
xlabel('Time (s)')
ylabel('Centroid (Hz)')
```

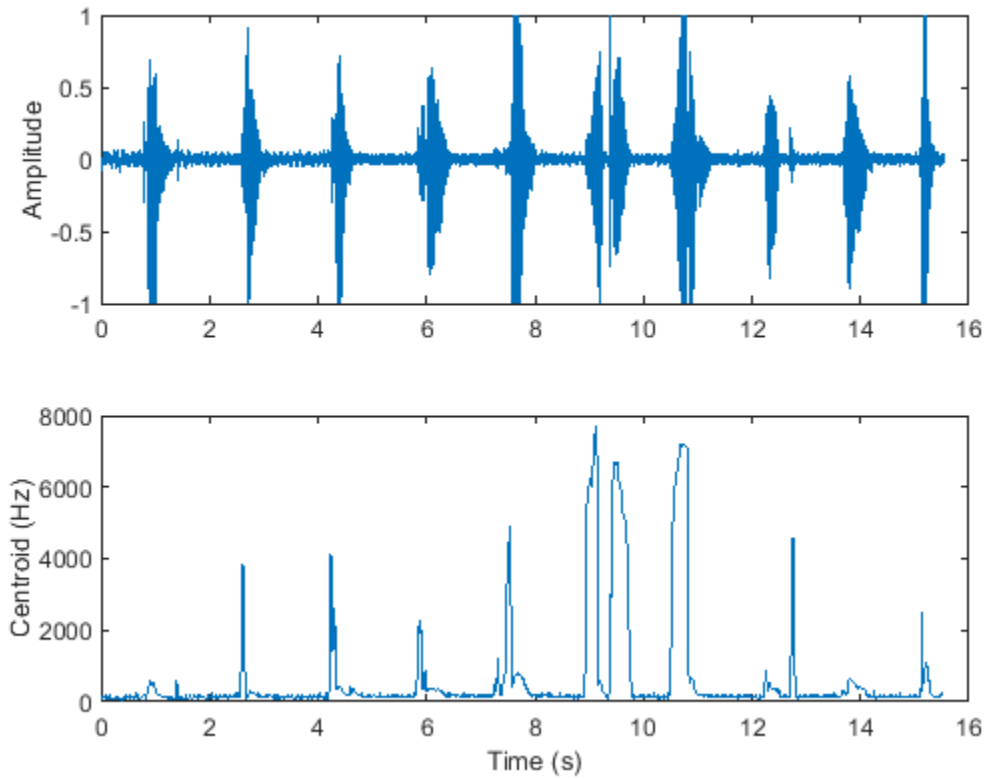


The spectral centroid is also commonly used to classify speech as voiced or unvoiced [3 on page 20-0]. For example, the centroid jumps in regions of unvoiced speech.

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

subplot(2,1,1)
t = linspace(0,size(audio,1)/fs,size(audio,1));
plot(t,audio)
ylabel('Amplitude')

subplot(2,1,2)
t = linspace(0,size(audio,1)/fs,size(centroid,1));
plot(t,centroid)
xlabel('Time (s)')
ylabel('Centroid (Hz)')
```



Spectral Spread

Spectral spread (`spectralSpread`) is the standard deviation around the spectral centroid [1 on page 20-0]:

$$\mu_2 = \sqrt{\frac{\sum_{k=b_1}^{b_2} (f_k - \mu_1)^2 s_k}{\sum_{k=b_1}^{b_2} s_k}}$$

where

- f_k is the frequency in Hz corresponding to bin k .
- s_k is the spectral value at bin k . The magnitude spectrum and power spectrum are both commonly used.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral spread.
- μ_1 is the spectral centroid.

The spectral spread represents the "instantaneous bandwidth" of the spectrum. It is used as an indication of the dominance of a tone. For example, the spread increases as the tones diverge and decreases as the tones converge.

```
fs = 16e3;
tone = audioOscillator('SampleRate',fs,'NumTones',2,'SamplesPerFrame',512,'Frequency',[2000,100]
duration = 5;
```

```

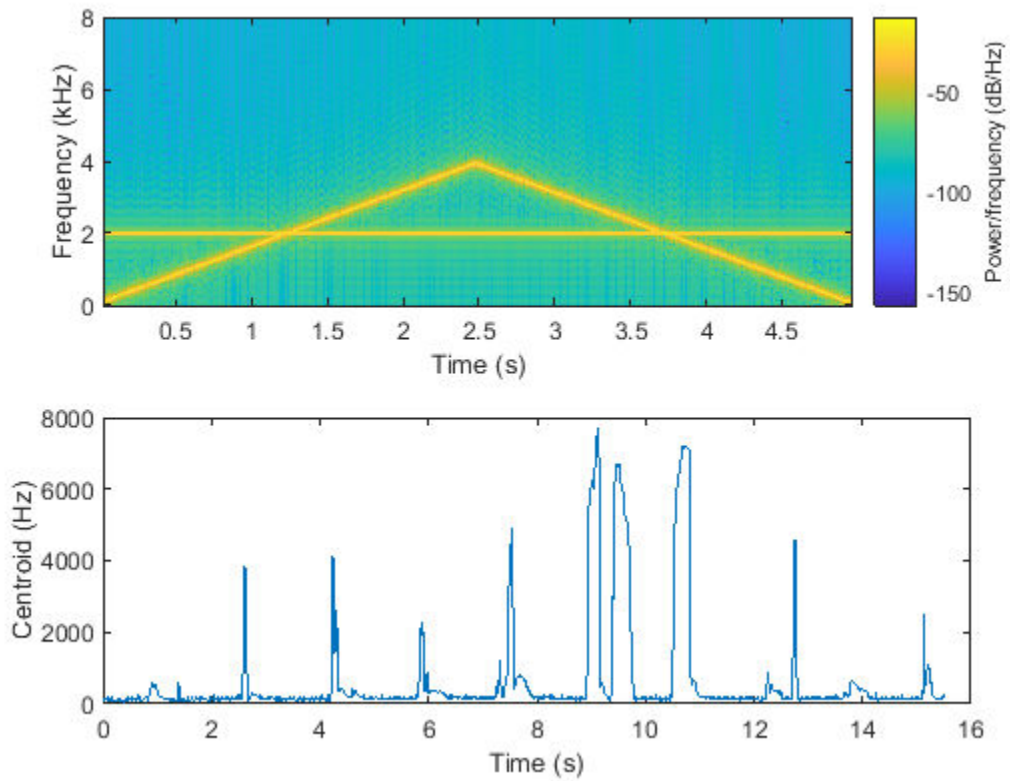
numLoops = floor(duration*fs/tone.SamplesPerFrame);
signal = [];
for i = 1:numLoops
    signal = [signal;tone()];
    if i<numLoops/2
        tone.Frequency = tone.Frequency + [0,50];
    else
        tone.Frequency = tone.Frequency - [0,50];
    end
end

spread = spectralSpread(signal,fs);

subplot(2,1,1)
spectrogram(signal,round(fs*0.05),round(fs*0.04),2048,fs,'yaxis')

subplot(2,1,2)

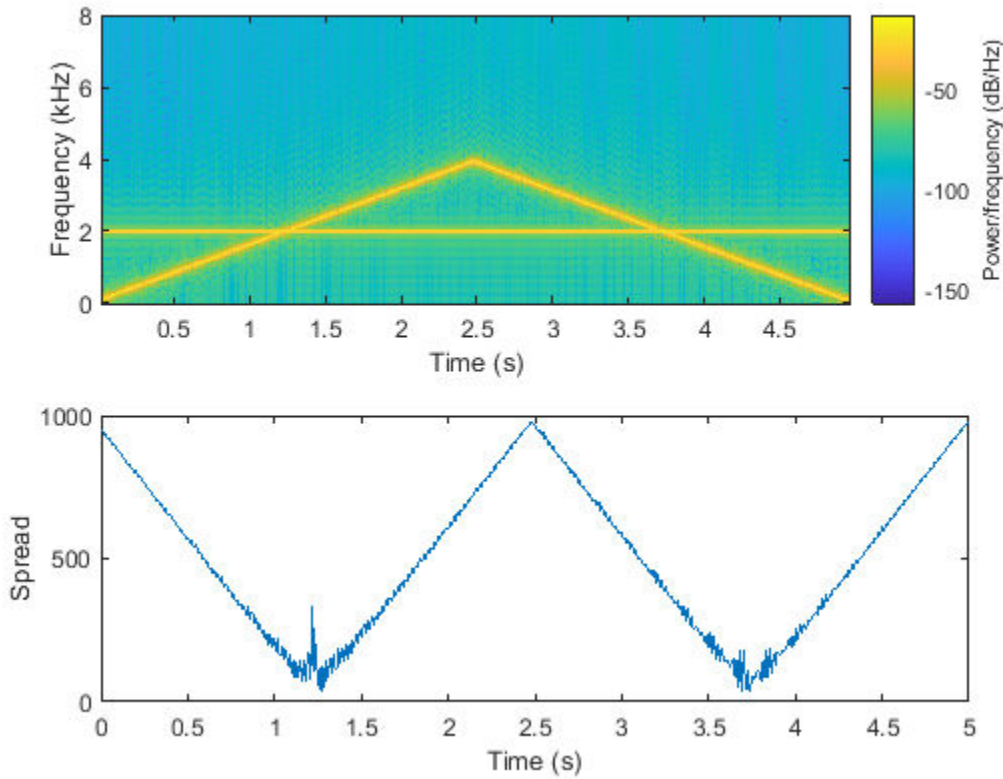
```



```

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

```



Spectral Skewness

Spectral skewness (`spectralSkewness`) is computed from the third order moment [1 on page 20-0]:

$$\mu_3 = \frac{\sum_{k=b_1}^{b_2} (f_k - \mu_1)^3 s_k}{(\mu_2)^3 \sum_{k=b_1}^{b_2} s_k}$$

where

- f_k is the frequency in Hz corresponding to bin k .
- s_k is the spectral value at bin k . The magnitude spectrum and power spectrum are both commonly used.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral skewness.
- μ_1 is the spectral centroid.
- μ_2 is the spectral spread.

The spectral skewness measures symmetry around the centroid. In phonetics, spectral skewness is often referred to as *spectral tilt* and is used with other spectral moments to distinguish the place of articulation [4 on page 20-0]. For harmonic signals, it indicates the relative strength of higher and lower harmonics. For example, in the four-tone signal, there is a positive skew when the lower tone is dominant and a negative skew when the upper tone is dominant.


```

fs = 16e3;
duration = 99;
tone = audioOscillator('SampleRate',fs,'NumTones',4,'SamplesPerFrame',fs,'Frequency',[500,2000,2000,500]);

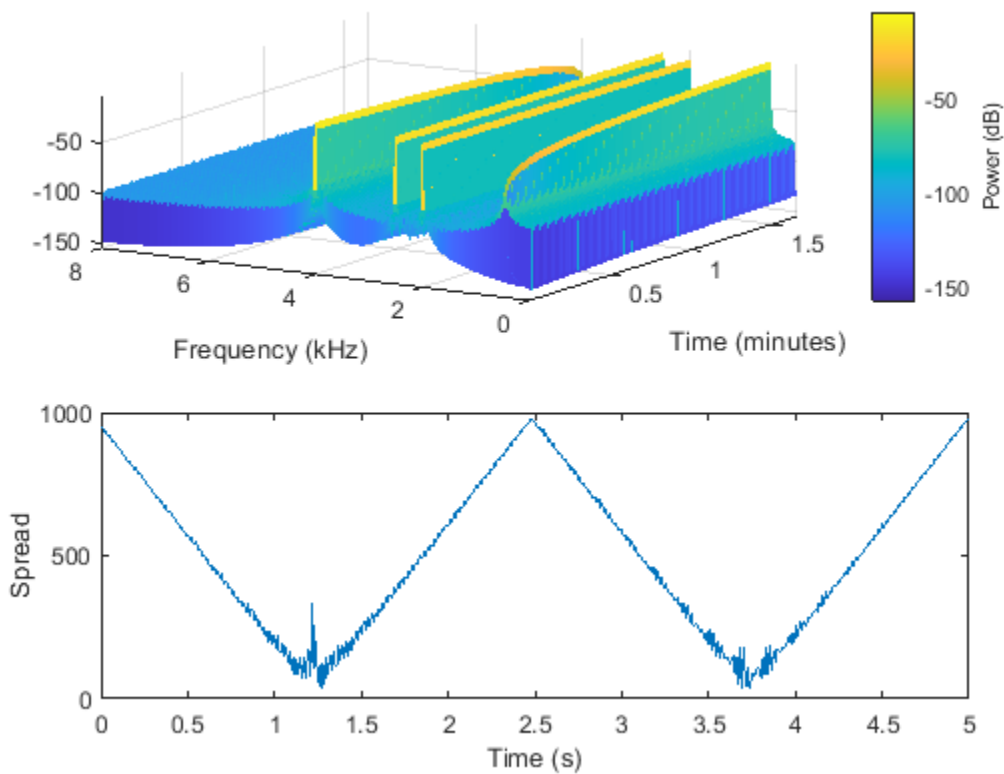
signal = [];
for i = 1:duration
    signal = [signal;tone()];
    tone.Amplitude = tone.Amplitude + [0.01,0,0,-0.01];
end

skewness = spectralSkewness(signal,fs);
t = linspace(0,size(signal,1)/fs,size(skewness,1))/60;

subplot(2,1,1)
spectrogram(signal,round(fs*0.05),round(fs*0.04),round(fs*0.05),fs,'yaxis','power')
view([-58 33])

subplot(2,1,2)

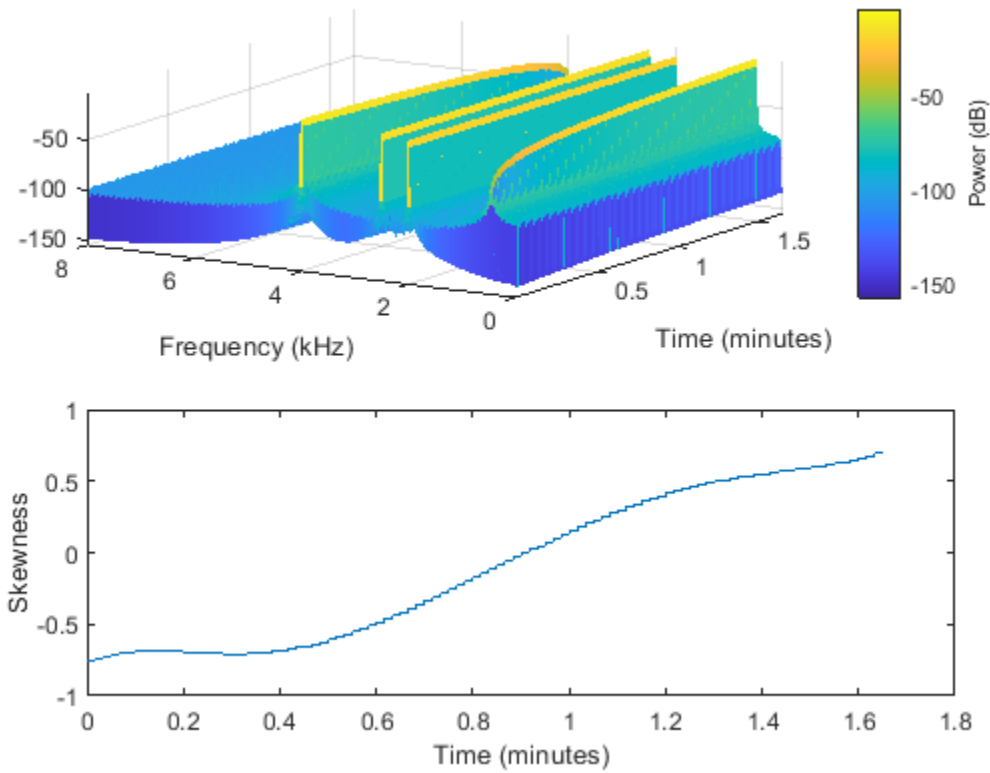
```



```

plot(t,skewness)
xlabel('Time (minutes)')
ylabel('Skewness')

```



Spectral Kurtosis

Spectral kurtosis (`spectralKurtosis`) is computed from the fourth order moment [1 on page 20-0]:

$$\mu_4 = \frac{\sum_{k=b_1}^{b_2} (f_k - \mu_1)^4 s_k}{(\mu_2)^4 \sum_{k=b_1}^{b_2} s_k}$$

where

- f_k is the frequency in Hz corresponding to bin k .
- s_k is the spectral value at bin k . The magnitude spectrum and power spectrum are both commonly used.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral kurtosis.
- μ_1 is the spectral centroid.
- μ_2 is the spectral spread.

The spectral kurtosis measures the flatness, or non-Gaussianity, of the spectrum around its centroid. Conversely, it is used to indicate the peakiness of a spectrum. For example, as the white noise is increased on the speech signal, the kurtosis decreases, indicating a less peaky spectrum.

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

```

noiseGenerator = dsp.ColoredNoise('Color','white','SamplesPerFrame',size(audioIn,1));

noise = noiseGenerator();
noise = noise/max(abs(noise));
ramp = linspace(0,.25,numel(noise))';
noise = noise.*ramp;

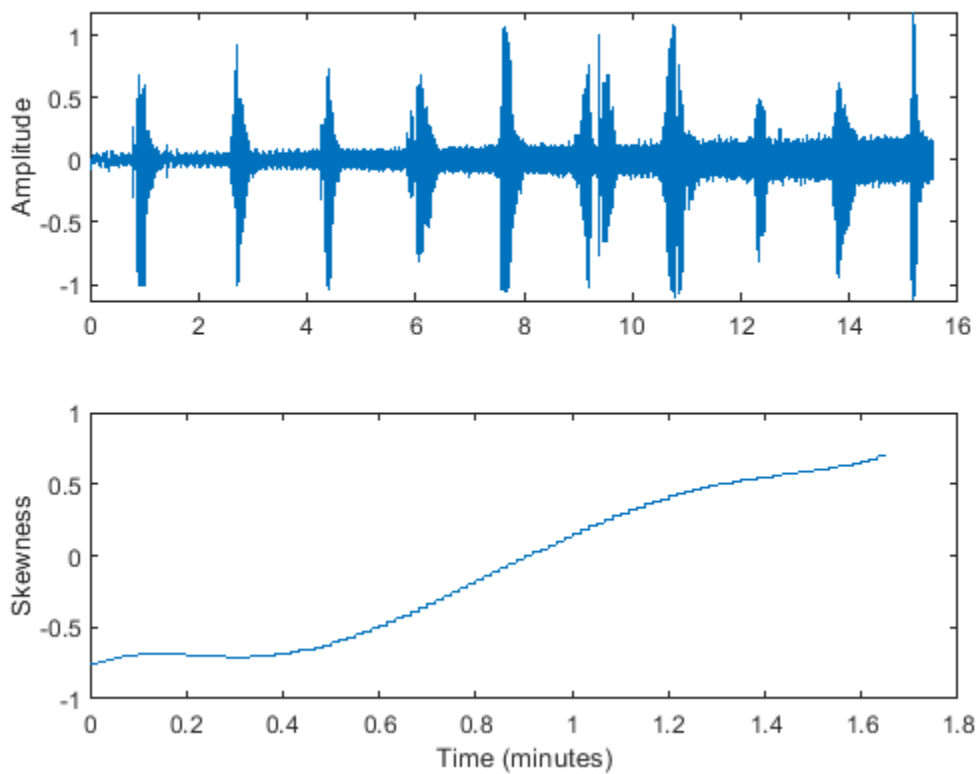
audioIn = audioIn + noise;

kurtosis = spectralKurtosis(audioIn,fs);

t = linspace(0,size(audioIn,1)/fs,size(audioIn,1));
subplot(2,1,1)
plot(t,audioIn)
ylabel('Amplitude')

t = linspace(0,size(audioIn,1)/fs,size(kurtosis,1));
subplot(2,1,2)

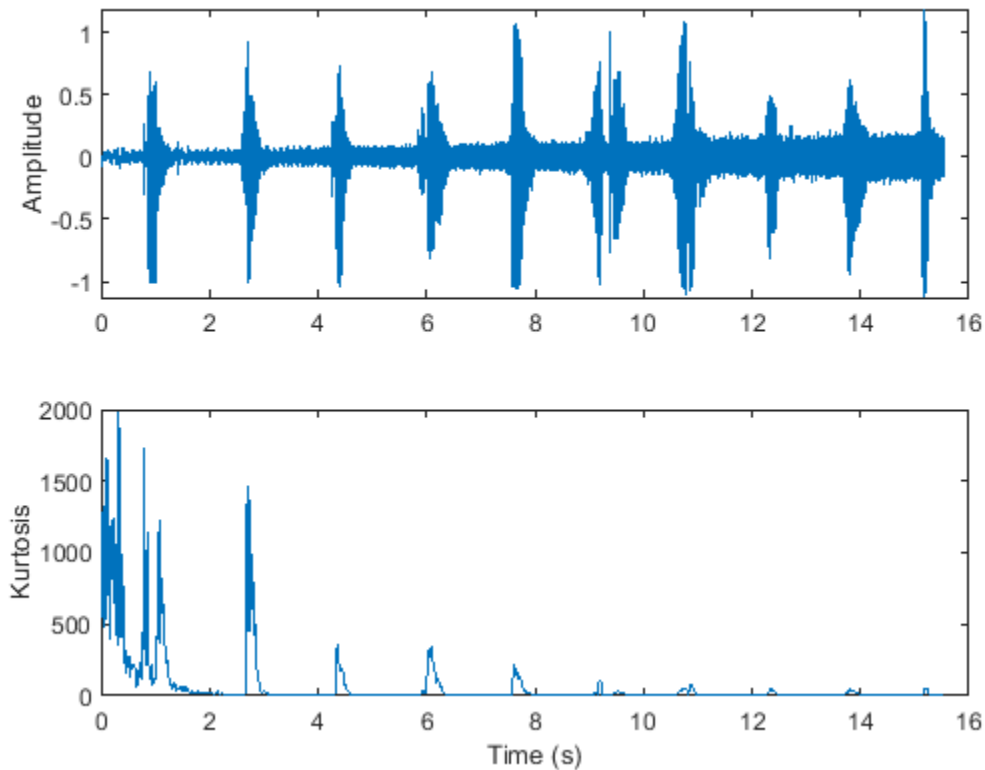
```



```

plot(t,kurtosis)
xlabel('Time (s)')
ylabel('Kurtosis')

```



Spectral Entropy

Spectral entropy (`spectralEntropy`) measures the peakiness of the spectrum [6 on page 20-0]:

$$\text{entropy} = \frac{-\sum_{k=b_1}^{b_2} s_k \log(s_k)}{\log(b_2 - b_1)}$$

where

- f_k is the frequency in Hz corresponding to bin k .
- s_k is the spectral value at bin k . The magnitude spectrum and power spectrum are both commonly used.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral entropy.

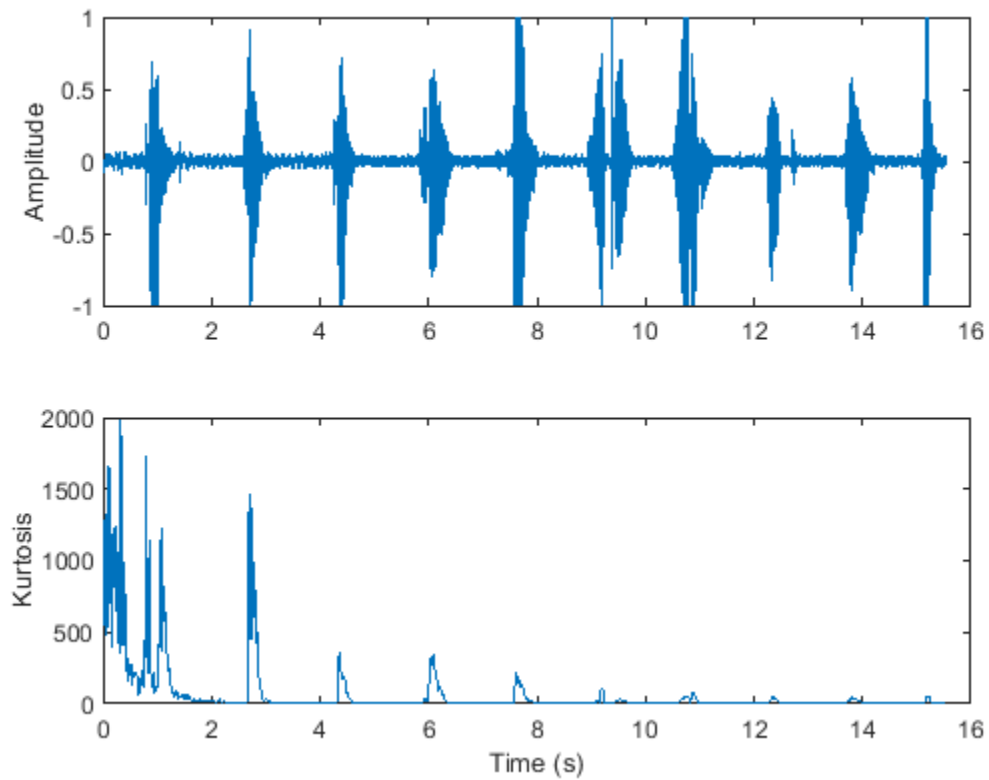
Spectral entropy has been used successfully in voiced/unvoiced decisions for automatic speech recognition [6 on page 20-0]. Because entropy is a measure of disorder, regions of voiced speech have lower entropy compared to regions of unvoiced speech.

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

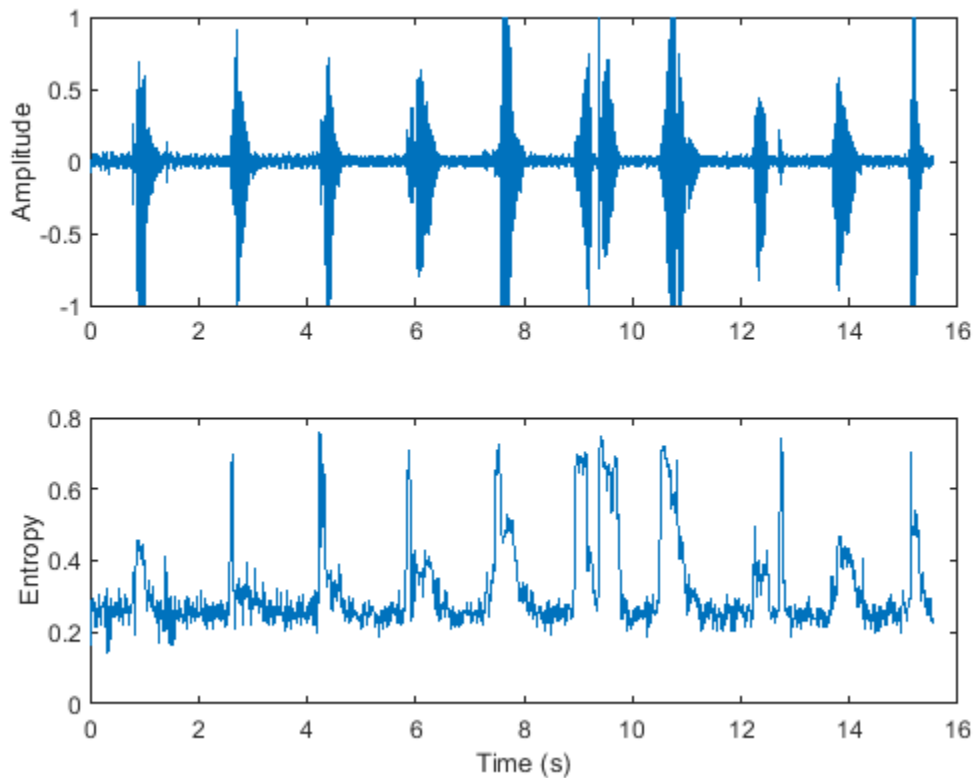
t = linspace(0,size(audioIn,1)/fs,size(audioIn,1));
subplot(2,1,1)
plot(t,audioIn)
```

```
ylabel('Amplitude')
```

```
t = linspace(0,size(audioIn,1)/fs,size(entropy,1));  
subplot(2,1,2)
```



```
plot(t,entropy)  
xlabel('Time (s)')  
ylabel('Entropy')
```



Spectral entropy has also been used to discriminate between speech and music [7 on page 20-0] [8 on page 20-0]. For example, compare histograms of entropy for speech, music, and background audio files.

```
fs = 8000;
[speech,speechFs] = audioread('Rainbow-16-8-mono-114secs.wav');
speech = resample(speech,fs,speechFs);
speech = speech./max(speech);

[music,musicFs] = audioread('RockGuitar-16-96-stereo-72secs.flac');
music = sum(music,2)/2;
music = resample(music,fs,musicFs);
music = music./max(music);

[background,backgroundFs] = audioread('Ambiance-16-44p1-mono-12secs.wav');
background = resample(background,fs,backgroundFs);
background = background./max(background);

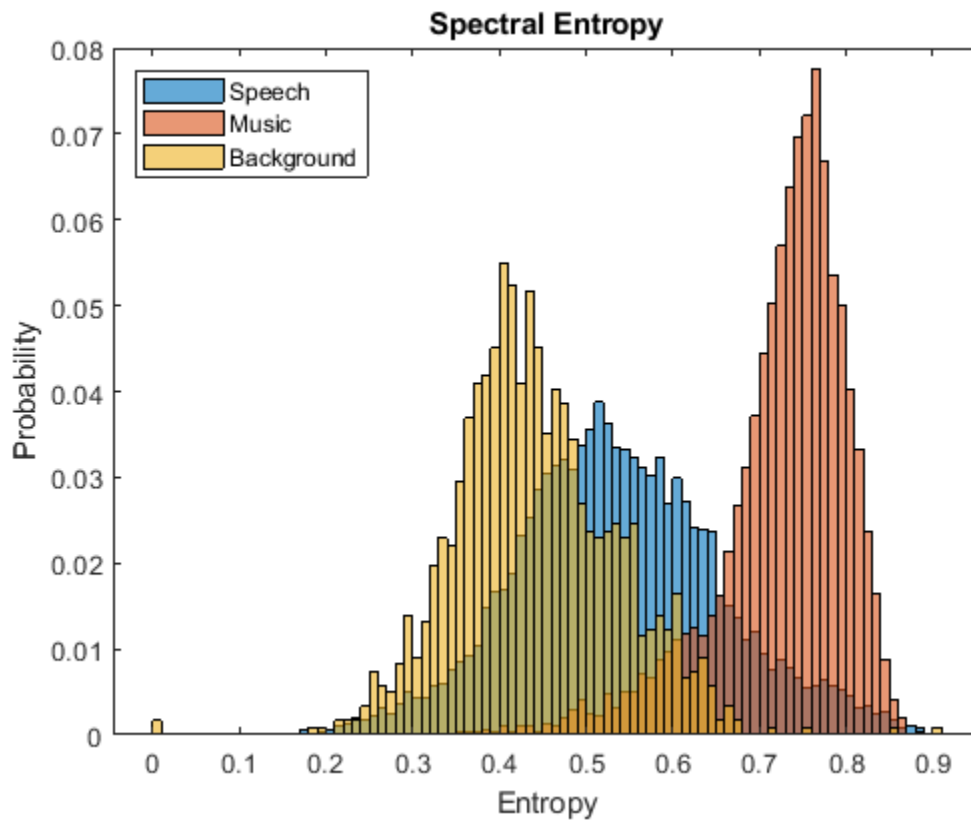
speechEntropy = spectralEntropy(speech,fs);
musicEntropy = spectralEntropy(music,fs);
backgroundEntropy = spectralEntropy(background,fs);

figure
h1 = histogram(speechEntropy);
hold on
h2 = histogram(musicEntropy);
h3 = histogram(backgroundEntropy);
```

```

h1.Normalization = 'probability';
h2.Normalization = 'probability';
h3.Normalization = 'probability';
h1.BinWidth = 0.01;
h2.BinWidth = 0.01;
h3.BinWidth = 0.01;
title('Spectral Entropy')
legend('Speech', 'Music', 'Background', 'Location', "northwest")
xlabel('Entropy')
ylabel('Probability')
hold off

```



Spectral Flatness

Spectral flatness (`spectralFlatness`) measures the ratio of the geometric mean of the spectrum to the arithmetic mean of the spectrum [9 on page 20-0]:

$$\text{flatness} = \frac{\left(\prod_{k=b_1}^{b_2} s_k \right)^{\frac{1}{b_2 - b_1}}}{\frac{1}{b_2 - b_1} \sum_{k=b_1}^{b_2} s_k}$$

where

- s_k is the spectral value at bin k . The magnitude spectrum and power spectrum are both commonly used.

- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral flatness.

Spectral flatness is an indication of the peakiness of the spectrum. A higher spectral flatness indicates noise, while a lower spectral flatness indicates tonality.

```
[audio,fs] = audioread('WaveGuideLoop0ne-24-96-stereo-10secs.aif');
audio = sum(audio,2)/2;

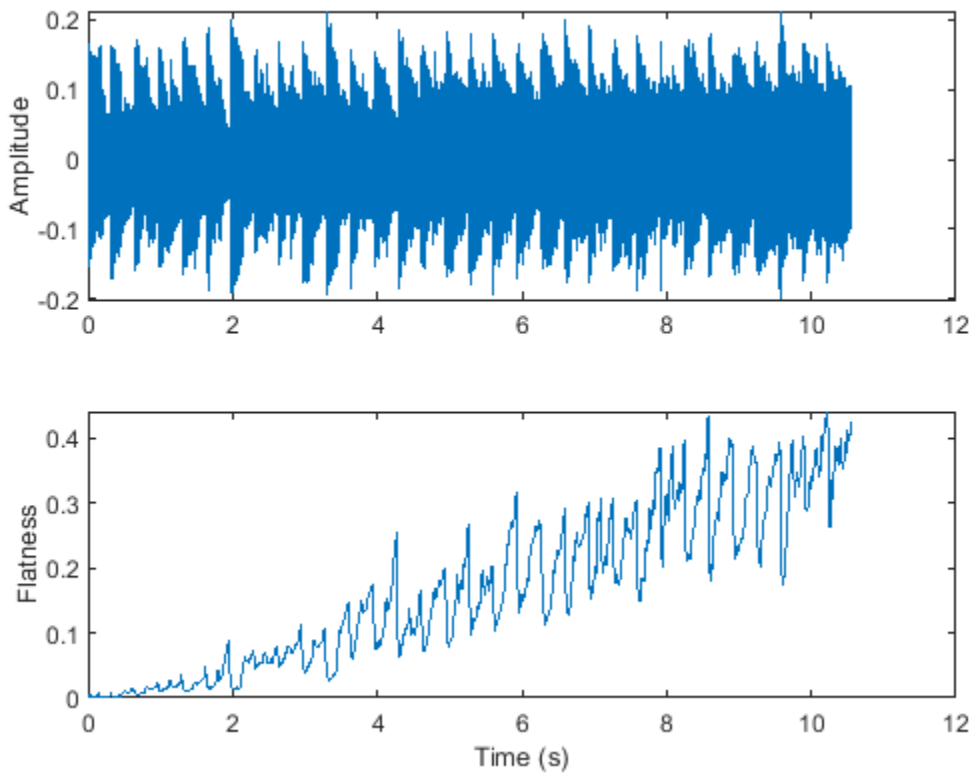
noise = (2*rand(numel(audio),1)-1).*linspace(0,0.05,numel(audio))';

audio = audio + noise;

flatness = spectralFlatness(audio,fs);

subplot(2,1,1)
t = linspace(0,size(audio,1)/fs,size(audio,1));
plot(t,audio)
ylabel('Amplitude')

subplot(2,1,2)
t = linspace(0,size(audio,1)/fs,size(flatness,1));
plot(t,flatness)
ylabel('Flatness')
xlabel('Time (s)')
```



Spectral flatness has also been applied successfully to singing voice detection [10 on page 20-0] and to audio scene recognition [11 on page 20-0].

Spectral Crest

Spectral crest (`spectralCrest`) measures the ratio of the maximum of the spectrum to the arithmetic mean of the spectrum [1 on page 20-0]:

$$\text{crest} = \frac{\max(s_{k \in [b_1, b_2]})}{\frac{1}{b_2 - b_1} \sum_{k=b_1}^{b_2} s_k}$$

where

- s_k is the spectral value at bin k . The magnitude spectrum and power spectrum are both commonly used.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral crest.

Spectral crest is an indication of the peakiness of the spectrum. A higher spectral crest indicates more tonality, while a lower spectral crest indicates more noise.

```
[audio,fs] = audioread('WaveGuideLoopOne-24-96-stereo-10secs.aif');
audio = sum(audio,2)/2;

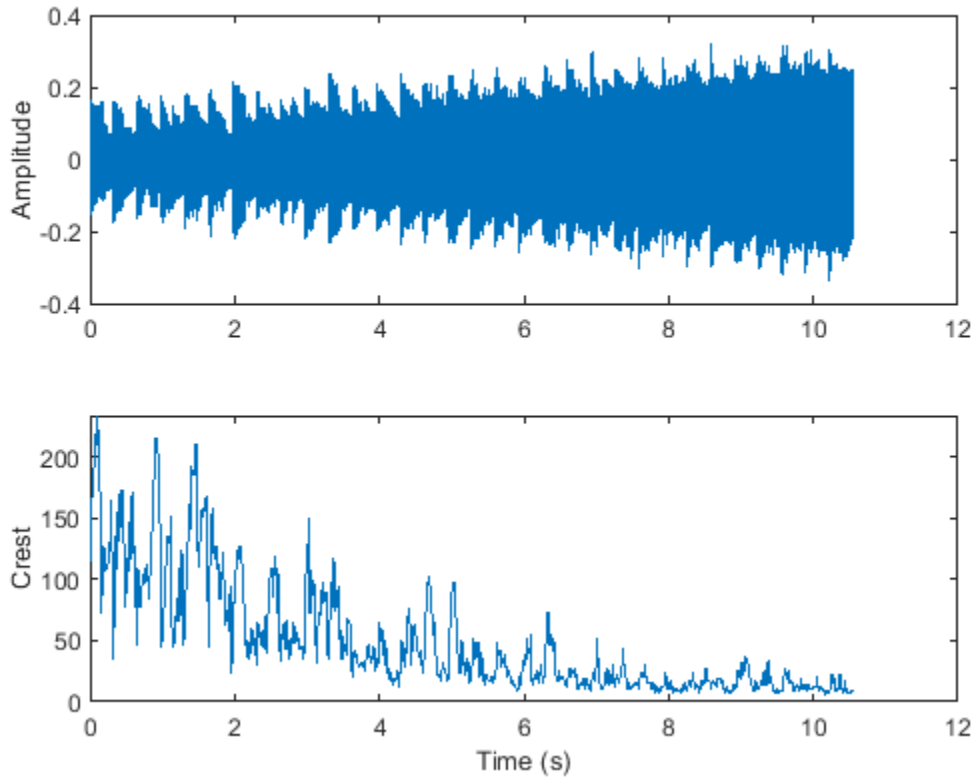
noise = (2*rand(numel(audio),1)-1).*linspace(0,0.2,numel(audio))';

audio = audio + noise;

crest = spectralCrest(audio,fs);

subplot(2,1,1)
t = linspace(0,size(audio,1)/fs,size(audio,1));
plot(t,audio)
ylabel('Amplitude')

subplot(2,1,2)
t = linspace(0,size(audio,1)/fs,size(crest,1));
plot(t,crest)
ylabel('Crest')
xlabel('Time (s)')
```



Spectral Flux

Spectral flux (`spectralFlux`) is a measure of the variability of the spectrum over time [12 on page 20-0]:

$$\text{flux}(t) = \left(\sum_{k=b_1}^{b_2} |s_k(t) - s_k(t-1)|^p \right)^{\frac{1}{p}}$$

where

- s_k is the spectral value at bin k . The magnitude spectrum and power spectrum are both commonly used.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral flux.
- p is the norm type.

Spectral flux is popularly used in onset detection [13 on page 20-0] and audio segmentation [14 on page 20-0]. For example, the beats in the drum track correspond to high spectral flux.

```
[audio,fs] = audioread('FunkyDrums-48-stereo-25secs.mp3');
audio = sum(audio,2)/2;

flux = spectralFlux(audio,fs);

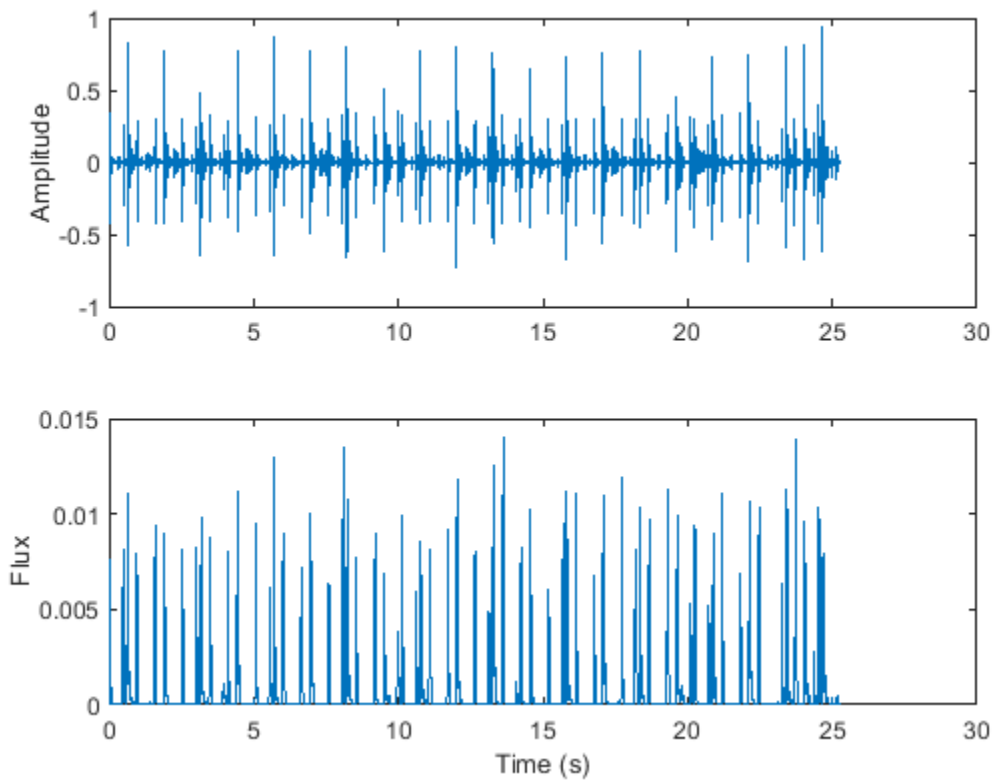
subplot(2,1,1)
```

```

t = linspace(0,size(audio,1)/fs,size(audio,1));
plot(t,audio)
ylabel('Amplitude')

subplot(2,1,2)
t = linspace(0,size(audio,1)/fs,size(flux,1));
plot(t,flux)
ylabel('Flux')
xlabel('Time (s)')

```



Spectral Slope

Spectral slope (`spectralSlope`) measures the amount of decrease of the spectrum [15 on page 20-0]:

$$\text{slope} = \frac{\sum_{k=b_1}^{b_2} (f_k - \mu_f)(s_k - \mu_s)}{\sum_{k=b_1}^{b_2} (f_k - \mu_f)^2}$$

where

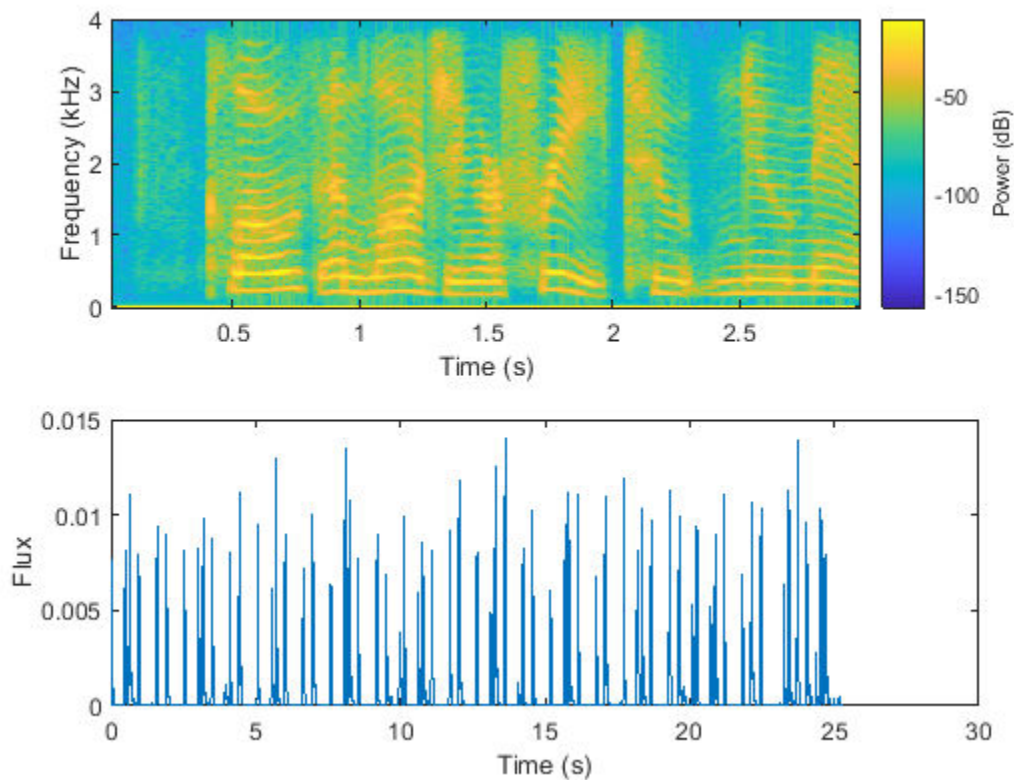
- f_k is the frequency in Hz corresponding to bin k .
- μ_f is the mean frequency.
- s_k is the spectral value at bin k . The magnitude spectrum is commonly used.

- μ_s is the mean spectral value.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral slope.

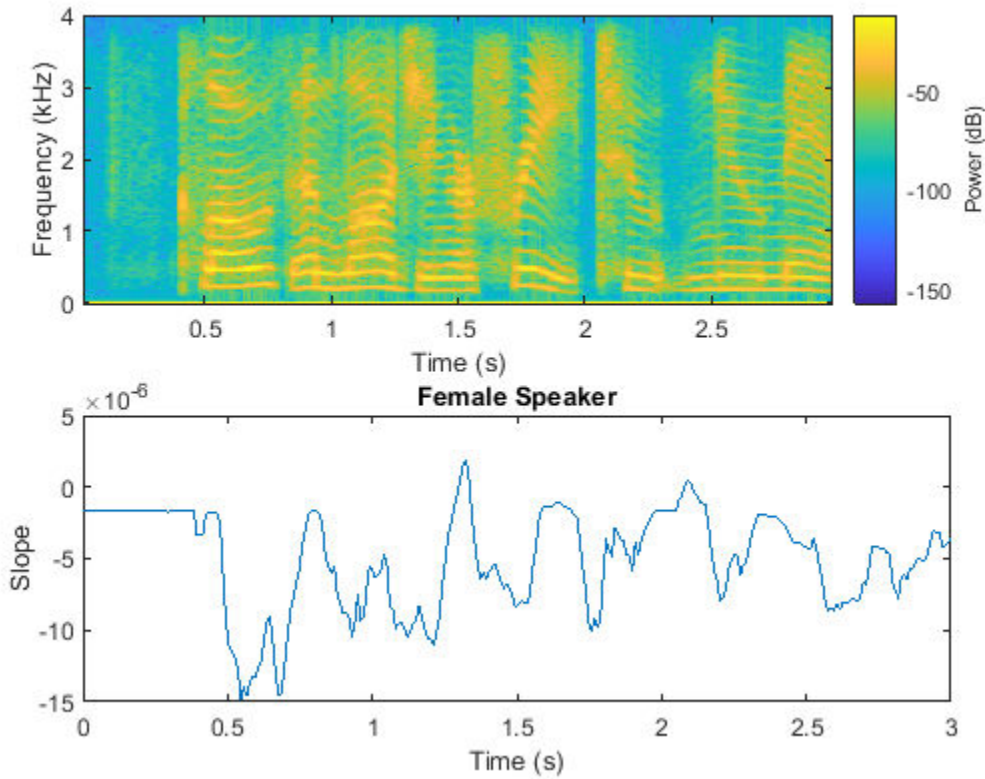
Spectral slope has been used extensively in speech analysis, particularly in modeling speaker stress [19 on page 20-0]. The slope is directly related to the resonant characteristics of the vocal folds and has also been applied to speaker identification [21 on page 20-0]. Spectral slope is a socially important aspect of timbre. Spectral slope discrimination has been shown to occur in early childhood development [20 on page 20-0]. Spectral slope is most pronounced when the energy in the lower formants is much greater than the energy in the higher formants.

```
[female,femaleFs] = audioread('FemaleSpeech-16-8-mono-3secs.wav');
female = female./max(female);
```

```
femaleSlope = spectralSlope(female,femaleFs);
t = linspace(0,size(female,1)/femaleFs,size(femaleSlope,1));
subplot(2,1,1)
spectrogram(female,round(femaleFs*0.05),round(femaleFs*0.04),round(femaleFs*0.05),femaleFs,'yaxis');
subplot(2,1,2)
```



```
plot(t,femaleSlope)
title('Female Speaker')
ylabel('Slope')
xlabel('Time (s)')
```



Spectral Decrease

Spectral decrease (`spectralDecrease`) represents the amount of decrease of the spectrum, while emphasizing the slopes of the lower frequencies [1 on page 20-0]:

$$\text{decrease} = \frac{\sum_{k=b_1+1}^{b_2} \frac{s_k - s_{b_1}}{k-1}}{\sum_{k=b_1+1}^{b_2} s_k}$$

where

- s_k is the spectral value at bin k . The magnitude spectrum is commonly used.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral decrease.

Spectral decrease is used less frequently than spectral slope in the speech literature, but it is commonly used, along with slope, in the analysis of music. In particular, spectral decrease has been shown to perform well as a feature in instrument recognition [22 on page 20-0].

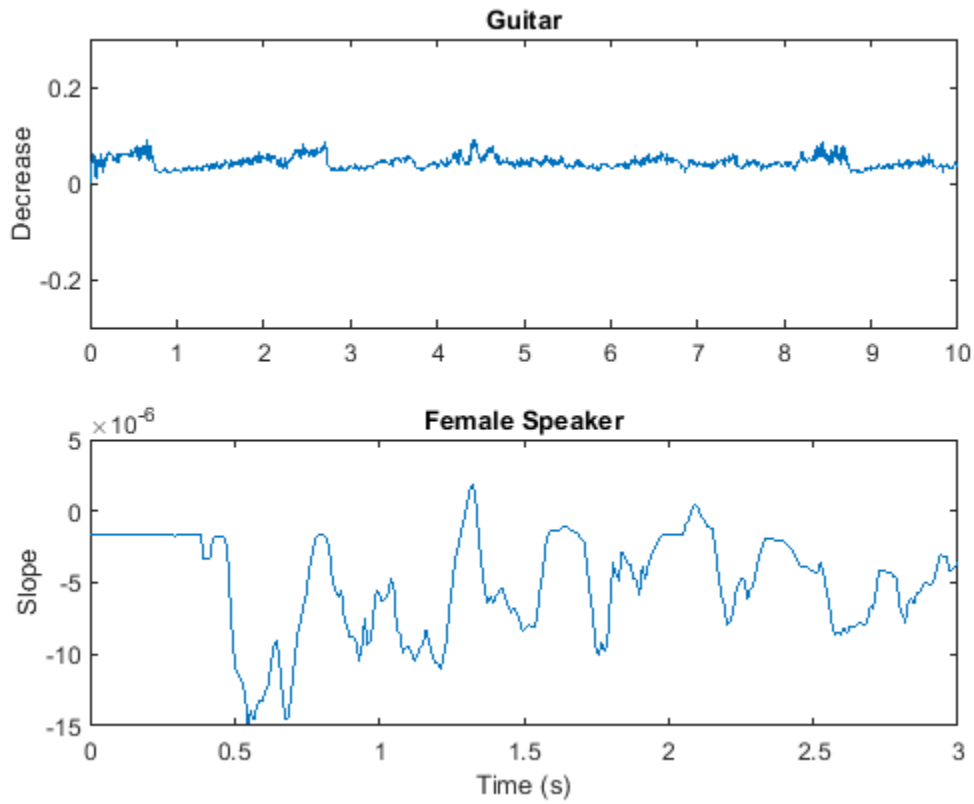
```
[guitar,guitarFs] = audioread('RockGuitar-16-44p1-stereo-72secs.wav');
guitar = mean(guitar,2);
[drums,drumsFs] = audioread('RockDrums-44p1-stereo-11secs.mp3');
drums = mean(drums,2);
```

```
guitarDecrease = spectralDecrease(guitar,guitarFs);
drumsDecrease = spectralDecrease(drums,drumsFs);
```

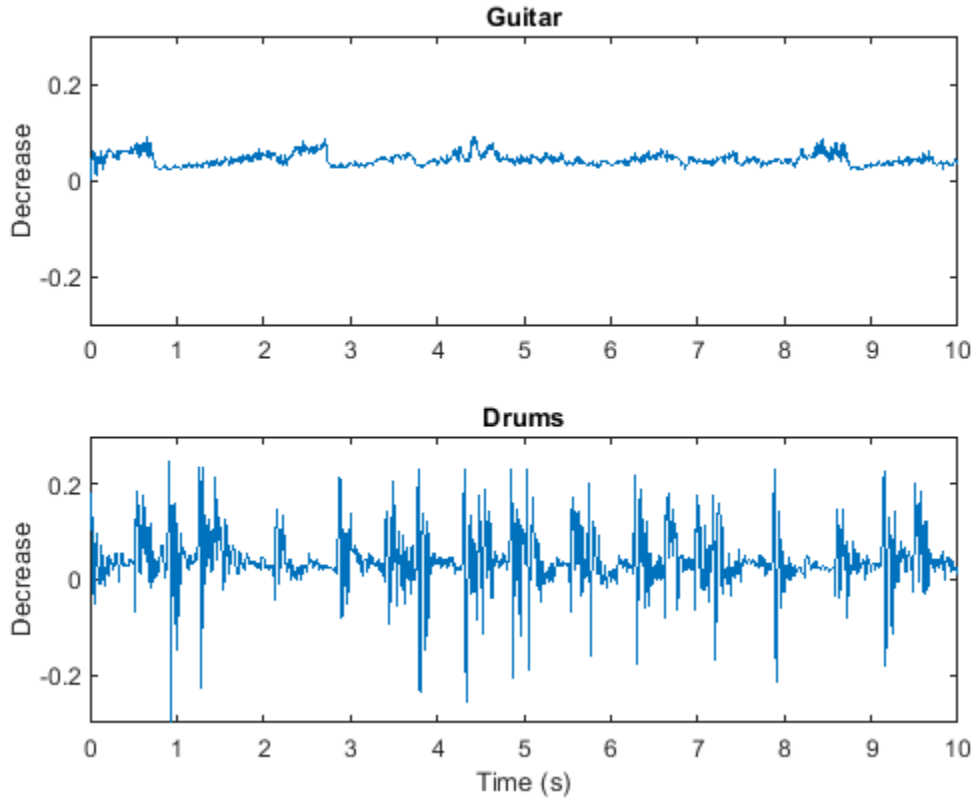
```
t1 = linspace(0,size(guitar,1)/guitarFs,size(guitarDecrease,1));
t2 = linspace(0,size(drums,1)/drumsFs,size(drumsDecrease,1));
```

```
subplot(2,1,1)
plot(t1,guitarDecrease)
title('Guitar')
ylabel('Decrease')
axis([0 10 -0.3 0.3])
```

```
subplot(2,1,2)
```



```
plot(t2,drumsDecrease)
title('Drums')
ylabel('Decrease')
xlabel('Time (s)')
axis([0 10 -0.3 0.3])
```



Spectral Rolloff Point

The spectral rolloff point (`spectralRolloffPoint`) measures the bandwidth of the audio signal by determining the frequency bin under which a given percentage of the total energy exists [12 on page 20-0]:

$$\text{Rolloff Point} = i \text{ such that } \sum_{k=b_1}^i |s_k| = \kappa \sum_{k=b_1}^{b_2} s_k$$

where

- s_k is the spectral value at bin k . The magnitude spectrum and power spectrum are both commonly used.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral rolloff point.
- κ is the specified energy threshold, usually 95% or 85%.

i is converted to Hz before it is returned by `spectralRolloffPoint`.

The spectral rolloff point has been used to distinguish between voiced and unvoiced speech, speech/music discrimination [12 on page 20-0], music genre classification [16 on page 20-0], acoustic scene recognition [17 on page 20-0], and music mood classification [18 on page 20-0]. For example, observe the different mean and variance of the rolloff point for speech, rock guitar, acoustic guitar, and an acoustic scene.

```
dur = 5; % Clip out 5 seconds from each file.

[speech,fs1] = audioread('SpeechDFT-16-8-mono-5secs.wav');
speech = speech(1:min(end,fs1*dur));

[electricGuitar,fs2] = audioread('RockGuitar-16-44p1-stereo-72secs.wav');
electricGuitar = mean(electricGuitar,2); % Convert to mono for comparison.
electricGuitar = electricGuitar(1:fs2*dur);

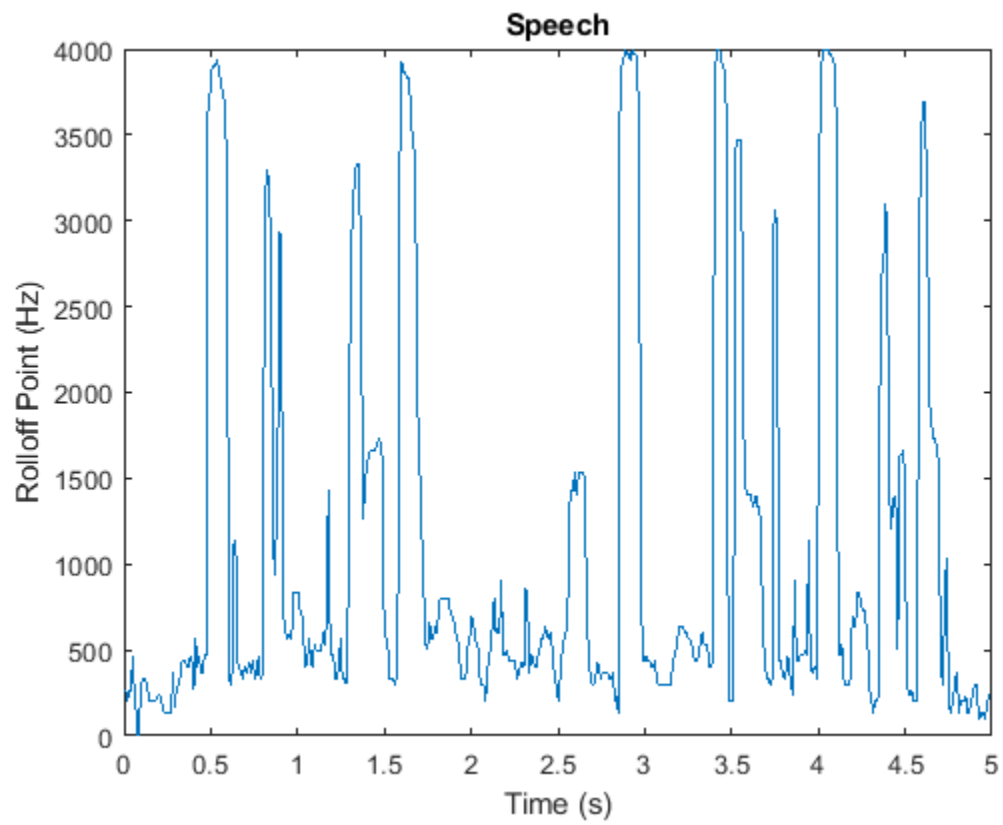
[acousticGuitar,fs3] = audioread('SoftGuitar-44p1_mono-10mins.ogg');
acousticGuitar = acousticGuitar(1:fs3*dur);

[acousticScene,fs4] = audioread('MainStreetOne-16-16-mono-12secs.wav');
acousticScene = acousticScene(1:fs4*dur);

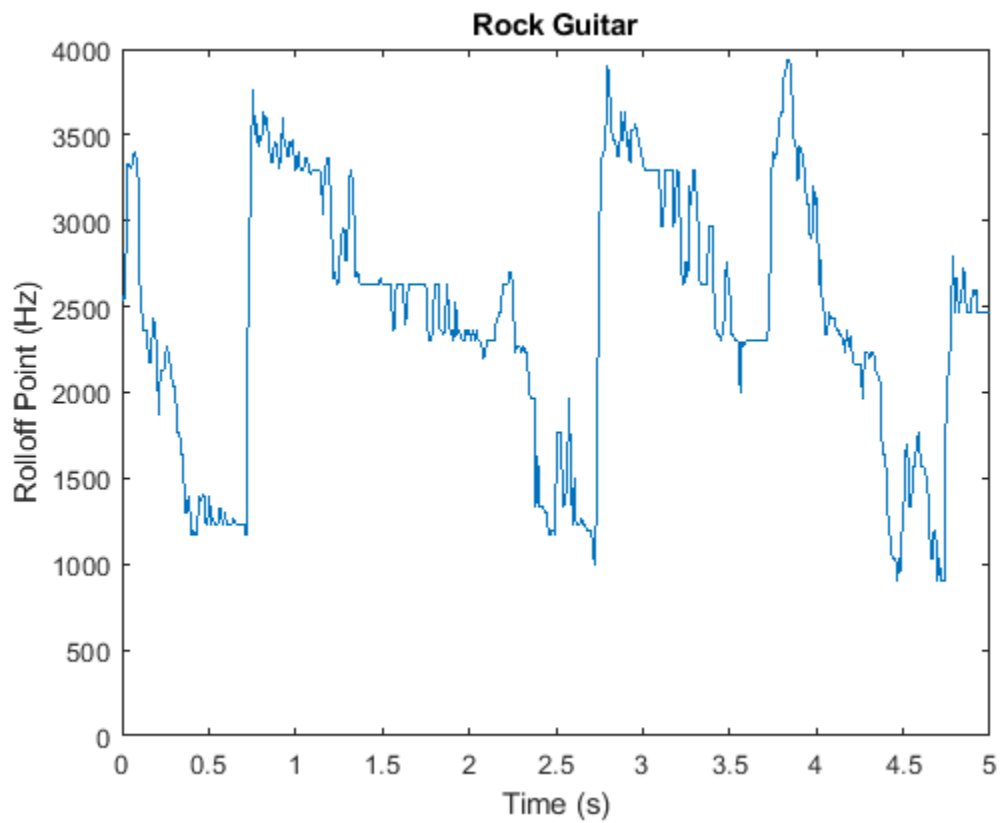
r1 = spectralRolloffPoint(speech,fs1);
r2 = spectralRolloffPoint(electricGuitar,fs2);
r3 = spectralRolloffPoint(acousticGuitar,fs3);
r4 = spectralRolloffPoint(acousticScene,fs4);

t1 = linspace(0,size(speech,1)/fs1,size(r1,1));
t2 = linspace(0,size(electricGuitar,1)/fs2,size(r2,1));
t3 = linspace(0,size(acousticGuitar,1)/fs3,size(r3,1));
t4 = linspace(0,size(acousticScene,1)/fs4,size(r4,1));

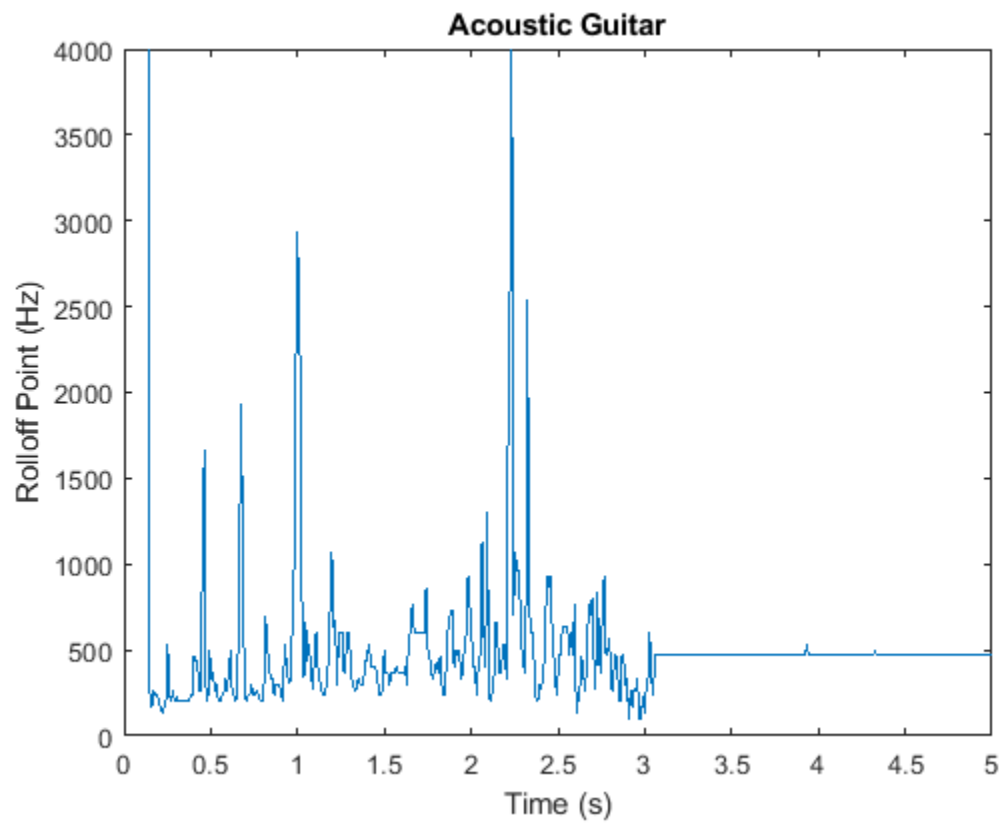
figure
plot(t1,r1)
title('Speech')
ylabel('Rolloff Point (Hz)')
xlabel('Time (s)')
axis([0 5 0 4000])
```

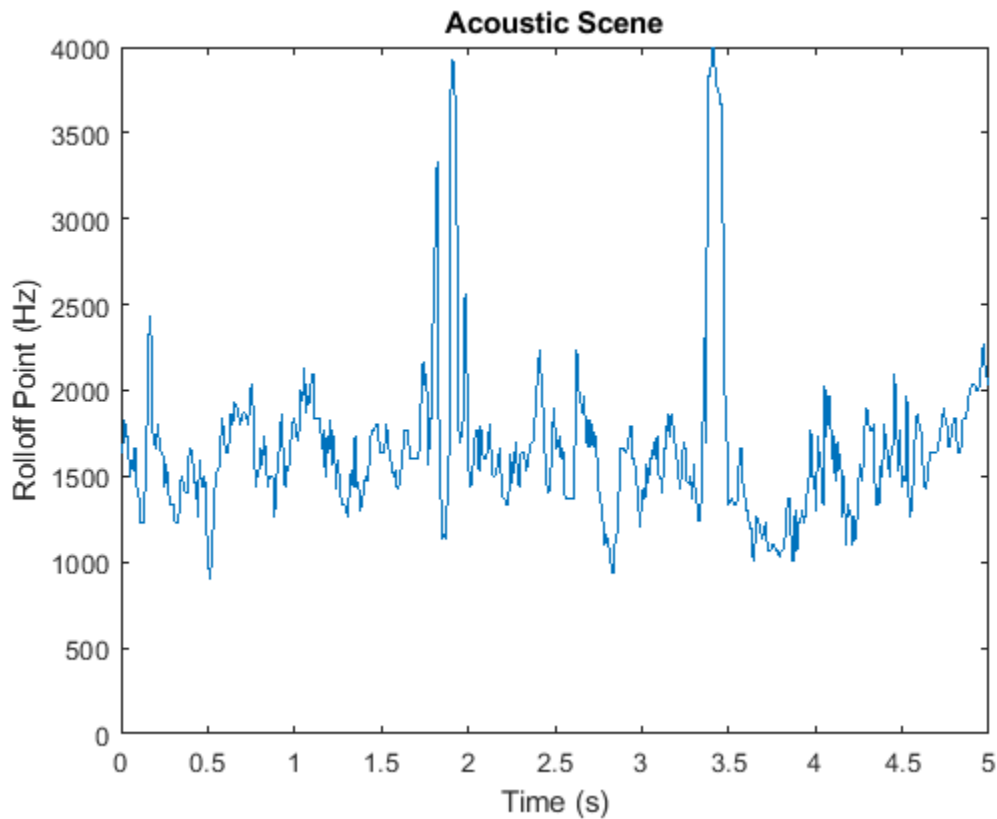
```
figure
plot(t2,r2)
title('Rock Guitar')
ylabel('Rolloff Point (Hz)')
xlabel('Time (s)')
axis([0 5 0 4000])
```



```
figure
plot(t3,r3)
title('Acoustic Guitar')
ylabel('Rolloff Point (Hz)')
xlabel('Time (s)')
axis([0 5 0 4000])
```



```
figure
plot(t4,r4)
title('Acoustic Scene')
ylabel('Rolloff Point (Hz)')
xlabel('Time (s)')
axis([0 5 0 4000])
```



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.
- [2] Grey, John M., and John W. Gordon. "Perceptual Effects of Spectral Modifications on Musical Timbres." *The Journal of the Acoustical Society of America*. Vol. 63, Issue 5, 1978, pp. 1493-1500.
- [3] Raimy, Eric, and Charles E. Cairns. *The Segment in Phonetics and Phonology*. Hoboken, NJ: John Wiley & Sons Inc., 2015.
- [4] Jongman, Allard, et al. "Acoustic Characteristics of English Fricatives." *The Journal of the Acoustical Society of America*. Vol. 108, Issue 3, 2000, pp. 1252-1263.
- [5] S. Zhang, Y. Guo, and Q. Zhang, "Robust Voice Activity Detection Feature Design Based on Spectral Kurtosis." *First International Workshop on Education Technology and Computer Science*, 2009, pp. 269-272.
- [6] Misra, H., S. Ikbali, H. Boulard, and H. Hermansky. "Spectral Entropy Based Feature for Robust ASR." *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*.
- [7] A. Pikrakis, T. Giannakopoulos, and S. Theodoridis. "A Computationally Efficient Speech/Music Discriminator for Radio Recordings." *International Conference on Music Information Retrieval and Related Activities*, 2006.

- [8] Pikrakis, A., et al. "A Speech/Music Discriminator of Radio Recordings Based on Dynamic Programming and Bayesian Networks." *IEEE Transactions on Multimedia*. Vol. 10, Issue 5, 2008, pp. 846-857.
- [9] Johnston, J.d. "Transform Coding of Audio Signals Using Perceptual Noise Criteria." *IEEE Journal on Selected Areas in Communications*. Vol. 6, Issue 2, 1988, pp. 314-323.
- [10] Lehner, Bernhard, et al. "On the Reduction of False Positives in Singing Voice Detection." *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014.
- [11] Y. Petetin, C. Laroche and A. Mayoue, "Deep Neural Networks for Audio Scene Recognition," *2015 23rd European Signal Processing Conference (EUSIPCO)*, 2015.
- [12] Scheirer, E., and M. Slaney. "Construction and Evaluation of a Robust Multifeature Speech/Music Discriminator." *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1997.
- [13] S. Dixon, "Onset Detection Revisited." *International Conference on Digital Audio Effects*. Vol. 120, 2006, pp. 133-137.
- [14] Tzanetakis, G., and P. Cook. "Multifeature Audio Segmentation for Browsing and Annotation." *Proceedings of the 1999 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, 1999.
- [15] Lerch, Alexander. *An Introduction to Audio Content Analysis Applications in Signal Processing and Music Informatics*. Piscataway, NJ: IEEE Press, 2012.
- [16] Li, Tao, and M. Ogihara. "Music Genre Classification with Taxonomy." *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2005.
- [17] Eronen, A.j., Vt. Peltonen, J.t. Tuomi, A.p. Klapuri, S. Fagerlund, T. Sorsa, G. Lorho, and J. Huopaniemi. "Audio-Based Context Recognition." *IEEE Transactions on Audio, Speech and Language Processing*. Vol. 14, Issue 1, 2006, pp. 321-329.
- [18] Ren, Jia-Min, Ming-Ju Wu, and Jyh-Shing Roger Jang. "Automatic Music Mood Classification Based on Timbre and Modulation Features." *IEEE Transactions on Affective Computing*. Vol. 6, Issue 3, 2015, pp. 236-246.
- [19] Hansen, John H. L., and Sanjay Patil. "Speech Under Stress: Analysis, Modeling and Recognition." *Lecture Notes in Computer Science*. Vol. 4343, 2007, pp. 108-137.
- [20] Tsang, Christine D., and Laurel J. Trainor. "Spectral Slope Discrimination in Infancy: Sensitivity to Socially Important Timbres." *Infant Behavior and Development*. Vol. 25, Issue 2, 2002, pp. 183-194.
- [21] Murthy, H.a., F. Beaufays, L.p. Heck, and M. Weintraub. "Robust Text-Independent Speaker Identification over Telephone Channels." *IEEE Transactions on Speech and Audio Processing*. Vol. 7, Issue 5, 1999, pp. 554-568.
- [22] Essid, S., G. Richard, and B. David. "Instrument Recognition in Polyphonic Music Based on Automatic Taxonomies." *IEEE Transactions on Audio, Speech and Language Processing*. Vol 14, Issue 1, 2006, pp. 68-80.

