

Audio Toolbox™

User's Guide



MATLAB® & SIMULINK®

R2023a



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 2016–2023 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)
March 2021	Online only	Revised for Version 3.0 (Release 2021a)
September 2021	Online only	Revised for Version 3.1 (Release 2021b)
March 2022	Online only	Revised for Version 3.2 (Release 2022a)
September 2022	Online only	Revised for Version 3.3 (Release 2022b)
March 2023	Online only	Revised for Version 3.4 (Release 2023a)

1

Audio Toolbox Examples

Transfer Learning with Pretrained Audio Networks	1-2
Effect of Soundproofing on Perceived Noise Levels	1-5
Speech Command Recognition Code Generation on Raspberry Pi	1-22
Speech Command Recognition Code Generation with Intel MKL-DNN .	1-32
Speech Command Recognition in Simulink	1-40
Time-Frequency Masking for Harmonic-Percussive Source Separation	1-43
Binaural Audio Rendering Using Head Tracking	1-65
Speech Emotion Recognition	1-70
End-to-End Deep Speech Separation	1-83
Delay-Based Pitch Shifter	1-99
Psychoacoustic Bass Enhancement for Band-Limited Signals	1-103
Tunable Filtering and Visualization Using Audio Plugins	1-106
Communicate Between a DAW and MATLAB Using UDP	1-114
Acoustic Echo Cancellation (AEC)	1-118
Active Noise Control Using a Filtered-X LMS FIR Adaptive Filter	1-130
Acoustic Noise Cancellation Using LMS	1-137
Delay-Based Audio Effects	1-139
Add Reverberation Using Freeverb Algorithm	1-145
Multiband Dynamic Range Compression	1-148
Pitch Shifting and Time Dilation Using a Phase Vocoder in MATLAB .	1-157
Pitch Shifting and Time Dilation Using a Phase Vocoder in Simulink .	1-161

Remove Interfering Tone From Audio Stream	1-163
Vorbis Decoder	1-166
Dynamic Range Compression Using Overlap-Add Reconstruction	1-169
LPC Analysis and Synthesis of Speech	1-171
Simulation of a Plucked String	1-173
Audio Phaser Using Multiband Parametric Equalizer	1-174
Loudness Normalization in Accordance with EBU R 128 Standard ...	1-178
Multistage Sample-Rate Conversion of Audio Signals	1-182
Graphic Equalization	1-189
Audio Weighting Filters	1-199
Sound Pressure Measurement of Octave Frequency Bands	1-202
Cochlear Implant Speech Processor	1-205
Acoustic Beamforming Using a Microphone Array	1-210
Identification and Separation of Panned Audio Sources in a Stereo Mix	1-219
Live Direction of Arrival Estimation with a Linear Microphone Array .	1-223
Positional Audio	1-227
Surround Sound Matrix Encoding and Decoding	1-230
Speaker Identification Using Pitch and MFCC	1-237
Measure Audio Latency	1-251
Measure Performance of Streaming Real-Time Audio Algorithms	1-256
THD+N Measurement with Tone-Tracking	1-259
Measure Impulse Response of an Audio System	1-262
Measure Frequency Response of an Audio Device	1-267
Generate Standalone Executable for Parametric Audio Equalizer	1-272
Deploy Audio Applications with MATLAB Compiler	1-275
Parametric Audio Equalizer for Android Devices	1-279
Parametric Audio Equalizer for iOS Devices	1-285

Audio Effects for iOS Devices	1-292
Multiband Dynamic Range Compression for iOS Devices	1-301
Denoise Speech Using Deep Learning Networks	1-312
Train Speech Command Recognition Model Using Deep Learning	1-332
Ambisonic Plugin Generation	1-344
Ambisonic Binaural Decoding	1-350
Multicore Simulation of Acoustic Beamforming Using a Microphone Array	1-353
Convert MIDI Files into MIDI Messages	1-358
Cocktail Party Source Separation Using Deep Learning Networks	1-368
Parametric Equalizer Design	1-391
Octave-Band and Fractional Octave-Band Filters	1-399
Pitch Tracking Using Multiple Pitch Estimations and HMM	1-406
Train Voice Activity Detection in Noise Model Using Deep Learning ..	1-430
Voice Activity Detection in Noise Using Deep Learning	1-449
Using a MIDI Control Surface to Interact with a Simulink Model	1-458
Spoken Digit Recognition with Wavelet Scattering and Deep Learning	1-461
Active Noise Control with Simulink Real-Time	1-477
Acoustic Scene Recognition Using Late Fusion	1-486
Keyword Spotting in Noise Using MFCC and LSTM Networks	1-501
Speaker Verification Using Gaussian Mixture Model	1-527
Sequential Feature Selection for Audio Features	1-545
Train Generative Adversarial Network (GAN) for Sound Synthesis	1-558
Speaker Verification Using i-Vectors	1-582
i-vector Score Normalization	1-607
i-vector Score Calibration	1-626
Speaker Recognition Using x-vectors	1-642

Speaker Diarization Using x-vectors	1-657
Train Spoken Digit Recognition Network Using Out-of-Memory Features	1-671
Train Spoken Digit Recognition Network Using Out-of-Memory Audio Data	1-678
Keyword Spotting in Noise Code Generation with Intel MKL-DNN . . .	1-685
Keyword Spotting in Noise Code Generation on Raspberry Pi	1-691
Dereverberate Speech Using Deep Learning Networks	1-699
Speaker Identification Using Custom SincNet Layer and Deep Learning	1-723
Acoustics-Based Machine Fault Recognition	1-737
Acoustics-Based Machine Fault Recognition Code Generation with Intel MKL-DNN	1-757
Acoustics-Based Machine Fault Recognition Code Generation on Raspberry Pi	1-764
audioDatastore Object Pointing to Audio Files	1-775
Accelerate Audio Deep Learning Using GPU-Based Feature Extraction	1-777
Train 3-D Sound Event Localization and Detection (SELD) Using Deep Learning	1-788
3-D Sound Event Localization and Detection Using Trained Recurrent Convolutional Neural Network	1-815
Import Audacity Labels to Signal Labeler	1-829
Room Impulse Response Simulation with the Image-Source Method and HRTF Interpolation	1-833
Room Impulse Response Simulation with Stochastic Ray Tracing	1-848
Feature Selection for Audio Classification	1-860
Transfer Learning with Pretrained Audio Networks in Deep Network Designer	1-872
Speech Command Recognition Code Generation with Intel MKL-DNN Using Simulink	1-881
Loudspeaker Modeling with Simscape™	1-891

Investigate Audio Classifications Using Deep Learning Interpretability Techniques	1-908
Speech Command Recognition on Raspberry Pi Using Simulink	1-922
Speech Command Recognition Using Deep Learning	1-929
Keyword Spotting in Simulink	1-940
Audio Transfer Learning Using Experiment Manager	1-942
Model Smart Speaker in Simulink	1-948
Train 3-D Speech Enhancement Network Using Deep Learning	1-951
3-D Speech Enhancement Using Trained Filter and Sum Network	1-973
Automated Design of Audio Filters for Room Equalization	1-983
Audio-Based Anomaly Detection for Machine Health Monitoring	1-997
Use Datastores to Manage Audio Data Sets	1-1010
Read, Analyze and Process SOFA Files	1-1014
Impulse Response Measurement Using a NI USB-4431 Device	1-1034
Plot Large Audio Files	1-1040
Audio Event Classification Using TensorFlow Lite on Raspberry Pi ..	1-1045
Deploy Smart Speaker System on Raspberry Pi Using Simulink	1-1053

Plugin GUI Design

2

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

Use the Audio Labeler

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

Choose an App to Label Ground Truth Data	3-13
---	-------------

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

Measure and Manage Impulse Responses	5-2
Configure Audio I/O System	5-2
Configure IR Acquisition Method	5-3
Acquire IR Measurements	5-4
Analyze and Manage IR Measurements	5-5
Export IR Measurements	5-7
Generate MATLAB Code	5-8

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

8

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

9

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

MIDI Control Surface Interface

10

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

11

Develop, Analyze, and Debug Plugins In Audio Test Bench	11-2
Choose Objects Under Test	11-2
Run Audio Test Bench	11-4
Debug Source Code of Audio Plugin	11-5
Open Scopes	11-5
Configure Input to Audio Test Bench	11-6
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	11-9
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
Deep Learning		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
What is C Code Generation from MATLAB?		14-5
Using MATLAB Coder		14-5
C/C++ Compiler Setup		14-5
Functions and System Objects That Support Code Generation		14-6

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	17-3
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
Use Octave Filter Bank to Create Flanging Chorus Effect for Guitar Layers (Overdubs)	17-50
Decompose Signal using Gammatone Filter Bank Block	17-52
Visualize Filter Response of Multiband Parametric Equalizer Block ..	17-54
Detect Music in Simulink Using YAMNet	17-57
Compare Sound Classifier block with Equivalent YAMNet blocks	17-60
Detect Air Compressor Sounds in Simulink Using YAMNet	17-62
Design Auditory Filter Bank	17-66
Design Mel Filter Bank	17-67
Extract Auditory Spectrogram	17-68
Extract Mel Spectrogram	17-69
Filter Audio Using Shelving Filter Block	17-71
Compare VGGish Embeddings Block with Equivalent VGGish Blocks .	17-72
Extract GTCC from Audio in Simulink	17-74
Include an Audio Plugin in Simulink	17-75
Generate Block from Plugin	17-75
Use Generated Audio Plugin Block in Model	17-76
Use VGGish Embeddings for Deep Learning in Simulink	17-78

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

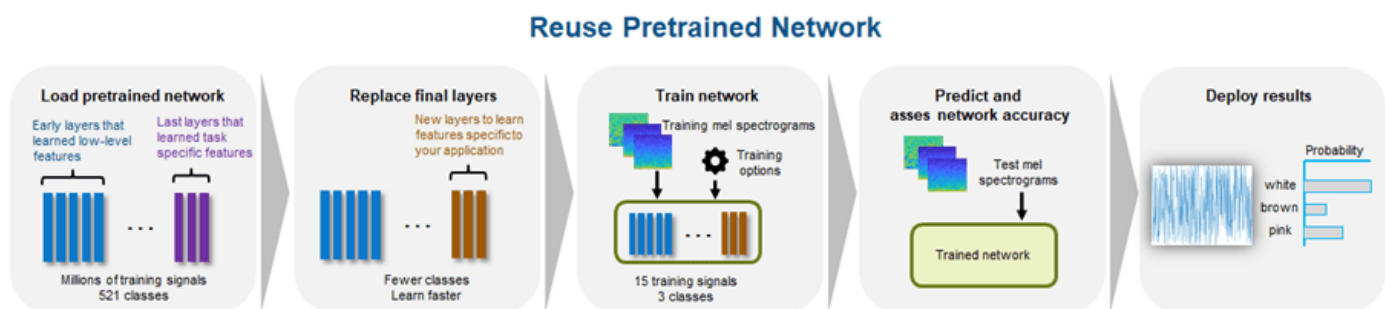
Spectral Descriptors	20-2
-----------------------------------	-------------

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 `vggishEmbeddings` 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 `yamnetPreprocess` 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.

```

trainFeatures = yamnetPreprocess(trainAudio,fs);
validationFeatures = yamnetPreprocess(validationAudio,fs);

```

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	Validat Loss
1	1	00:00:01	20.00%	88.42%	1.1922	0.0
30	30	00:00:14	100.00%	100.00%	5.0068e-06	0.0

Training finished: Max epochs completed.

Effect of Soundproofing on Perceived Noise Levels

In this example, you measure engine noise and use psychoacoustic metrics to model its perceived loudness, sharpness, fluctuation strength, roughness, and overall annoyance level. You then simulate the addition of soundproofing material and recompute the overall annoyance level. Finally, you compare annoyance levels and show the perceptual improvements gained from applying soundproofing.

Recording Level Calibration

Psychoacoustic measurements produce the most accurate results with a calibrated microphone input level. To use `calibrateMicrophone` to match your recording level to the reading of an SPL meter, you can use a 1 kHz tone source (such as an online tone generator or cell phone app) and a calibrated SPL meter. The SPL of the 1 kHz calibration tone should be loud enough to dominate any background noise. For a calibration example using MATLAB as the 1 kHz tone source, see `calibrateMicrophone`.

Simulate the tone recording and include some background noise. Assume an SPL meter reading of 83.1 dB (C-weighted).

```
FS = 48e3;
t = (1:2*FS)/FS;
s = rng('default');
testTone = 0.46*sin(2*pi*t*1000).' + .1*pinknoise(2*FS);
rng(s)

splMeterReading = 83.1;
```

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. To compensate for possible background noise and produce a precise calibration level, match the frequency weighting setting of the SPL meter.

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

Sound Pressure Levels (SPL)

Once you have a calibration factor for your recording chain, you can make acoustic measurements. When using a physical meter, you are limited to the settings selected during measurement time. With the `splMeter` object, you can change your settings after the recording has been made. This makes it easy to experiment with different time and frequency weighting options.

Load an engine recording and create the corresponding time vector.

```
[x,FS] = audioread('Engine-16-44p1-stereo-20sec.wav');
x = x(1:8*FS,1); % use only channel 1 and keep only 8 seconds.
t = (1:size(x,1))/FS;
```

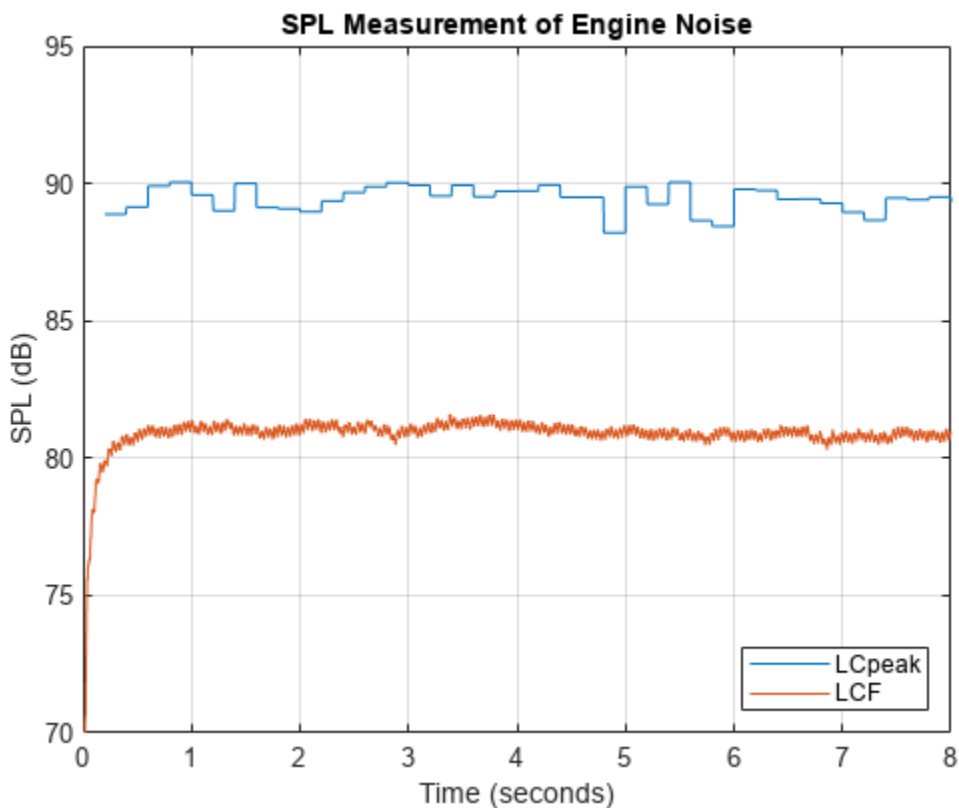
Create an `splMeter` object and select C-weighting, fast time weighting, and a 0.2 second interval for peak SPL measurement.

```
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 Engine Noise')
xlabel('Time (seconds)')
ylabel('SPL (dB)')
ylim([70 95])
grid on
```



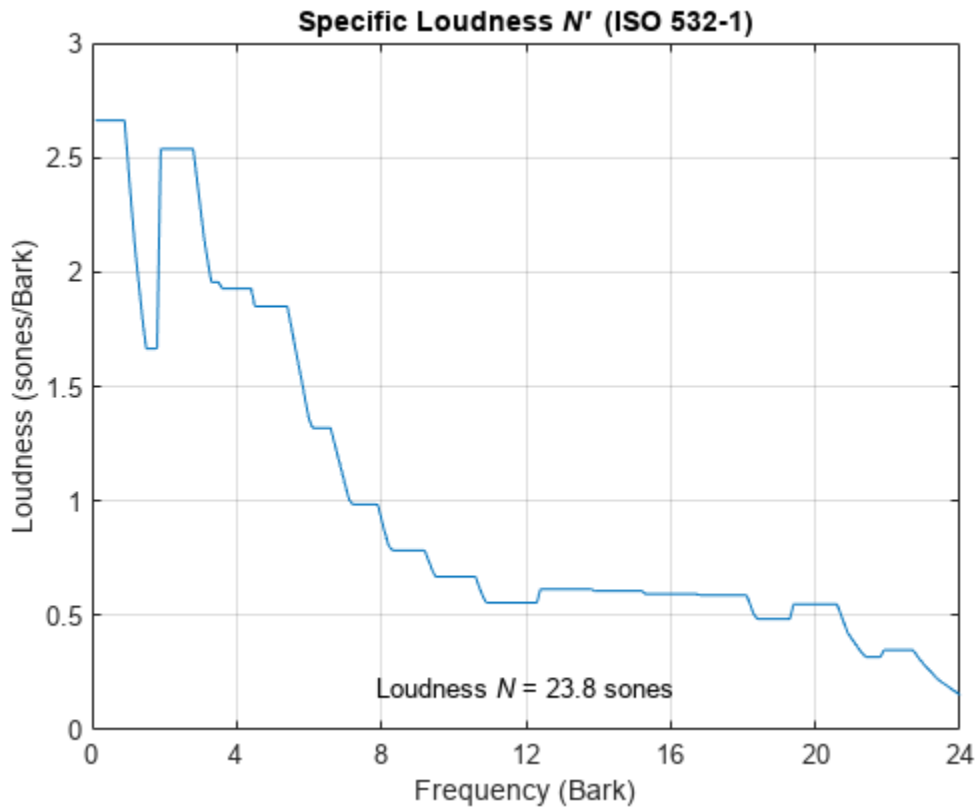
Psychoacoustic Metrics

Loudness level

Monitoring SPL is important for occupational safety compliance. However, SPL measurements do not reflect loudness as perceived by an actual listener. `acousticLoudness` measures loudness levels as perceived by a human listener with normal hearing (no hearing impairments). The `acousticLoudness` function also shows which frequency bands contribute the most to the perceptual 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 23.8 sones, and much of the noise is below 3.3 (Bark scale). Convert 3.3 Bark to Hz using `bark2hz`

```
fprintf("Loudness frequency of 3.3 Bark: %d Hz\n", round(bark2hz(3.3), -1));
```

```
Loudness frequency of 3.3 Bark: 330 Hz
```

The `acousticLoudness` function returns perceived loudness in sones. To understand the sone measurement, compare it to an SPL (dB) reading. A signal with a loudness of 60 phons is perceived to be as loud as a 1 kHz tone measured at 60 dB SPL. Converting 23.8 sones to phons using `sone2phon` demonstrates the loudness perception of the engine noise is as loud as a 1 kHz tone measured at 86 dB SPL.

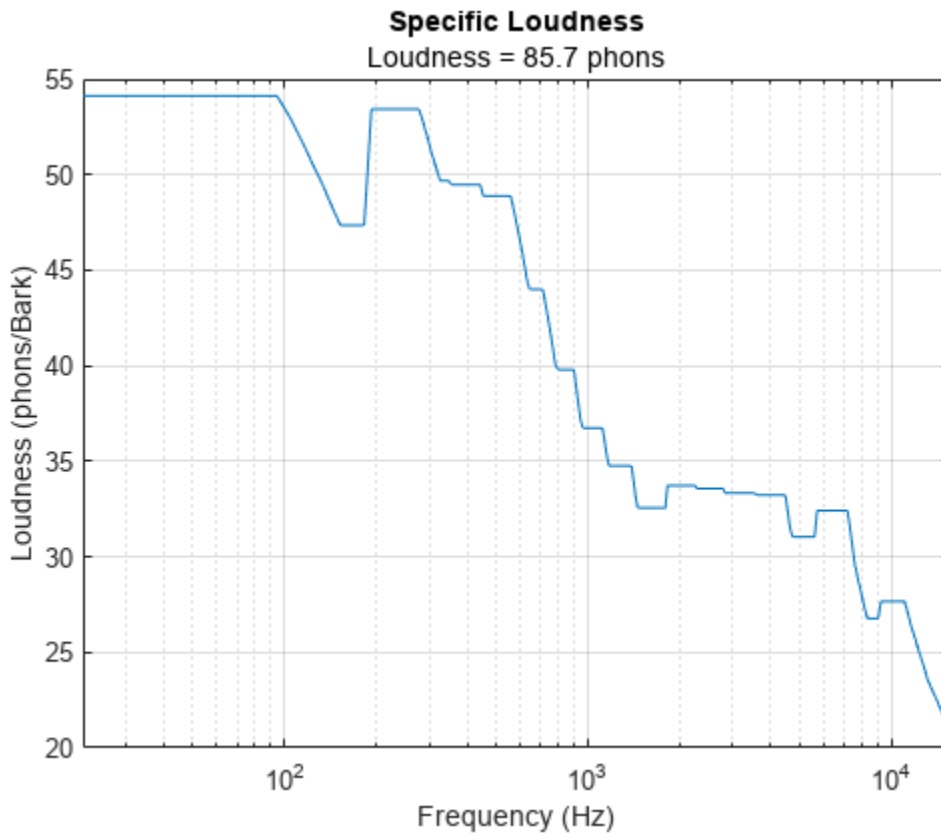
```
fprintf("Equivalent 1 kHz SPL: %d phons\n", round(sone2phon(23.8)));
```

```
Equivalent 1 kHz SPL: 86 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)')
```

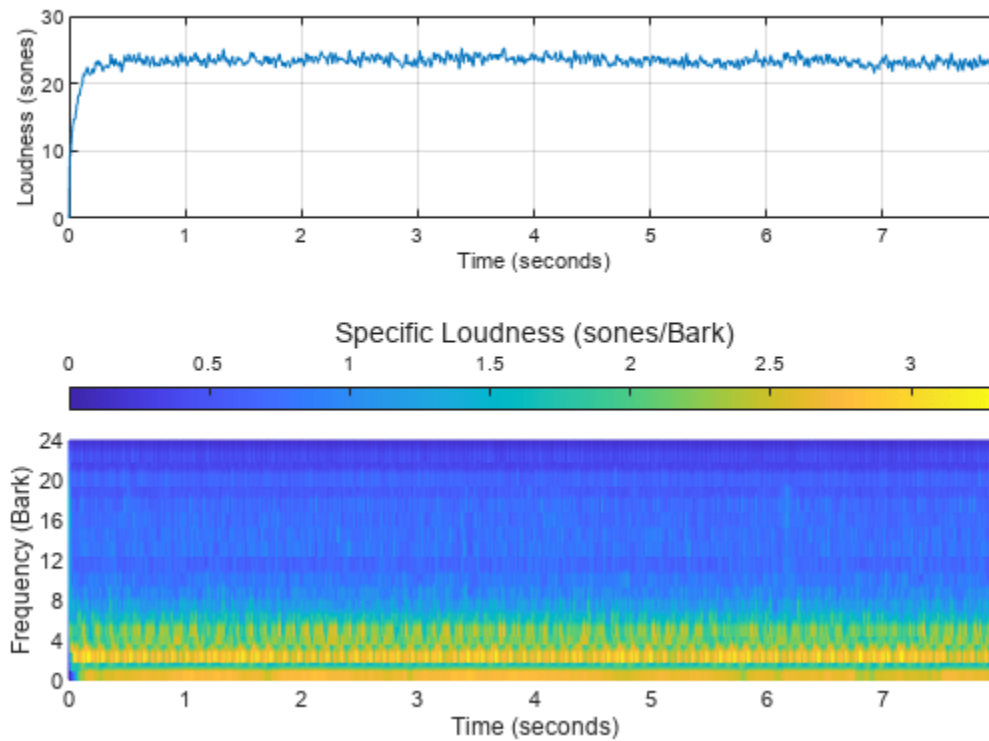
```
xlim(hz([1,end]))  
grid on
```



You can also plot time-varying loudness and specific loudness to analyze the sound of the engine if it changes with time. This can be displayed with other relevant time-varying data, such as engine revolutions per minute (RPMs). In this case, the noise is stationary, but you can observe the impulsive nature of the noise.

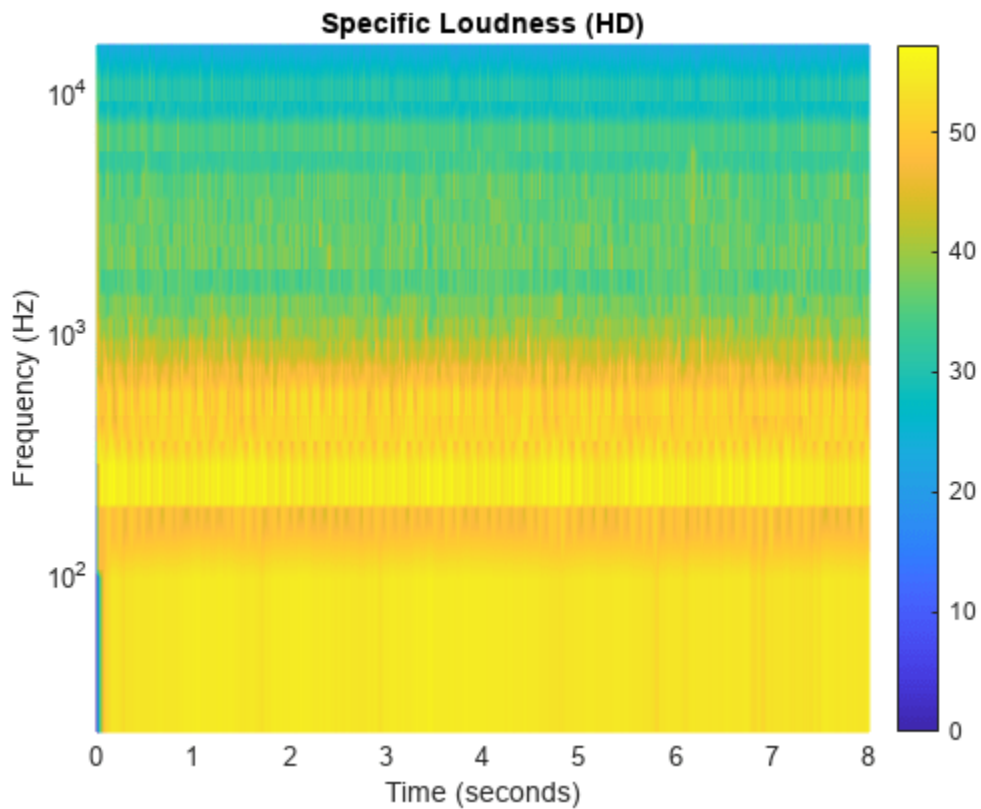
```
acousticLoudness(x,FS,calib,'TimeVarying',true,'TimeResolution','high')
```

Time-Varying Loudness (ISO 532-1)



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

```
[tvsoneHD,tvspecHD,perc] = acousticLoudness(x,FS,calib,'TimeVarying',true,'TimeResolution','high');
tvspec = tvspecHD(1:4:end,:); % for standard resolution measurements
spectimeHD = 0:5e-4:5e-4*(size(tvspecHD,1)-1); % time axis for loudness output
clf; % do not reuse the previous subplot
surf(spectimeHD,hz,sone2phon(tvspecHD).','EdgeColor','interp');
set(gca,'View',[0 90],'YScale','log','YLim',hz([1,end]));
title('Specific Loudness (HD)')
xlabel('Specific Loudness (phons/Bark)')
ylabel('Frequency (Hz)')
xlabel('Time (seconds)')
colorbar
```



Sharpness Level

The perceived sharpness of a sound can significantly contribute to its overall annoyance level. Estimate the perceived sharpness level of an acoustic signal using the `acousticSharpness` function.

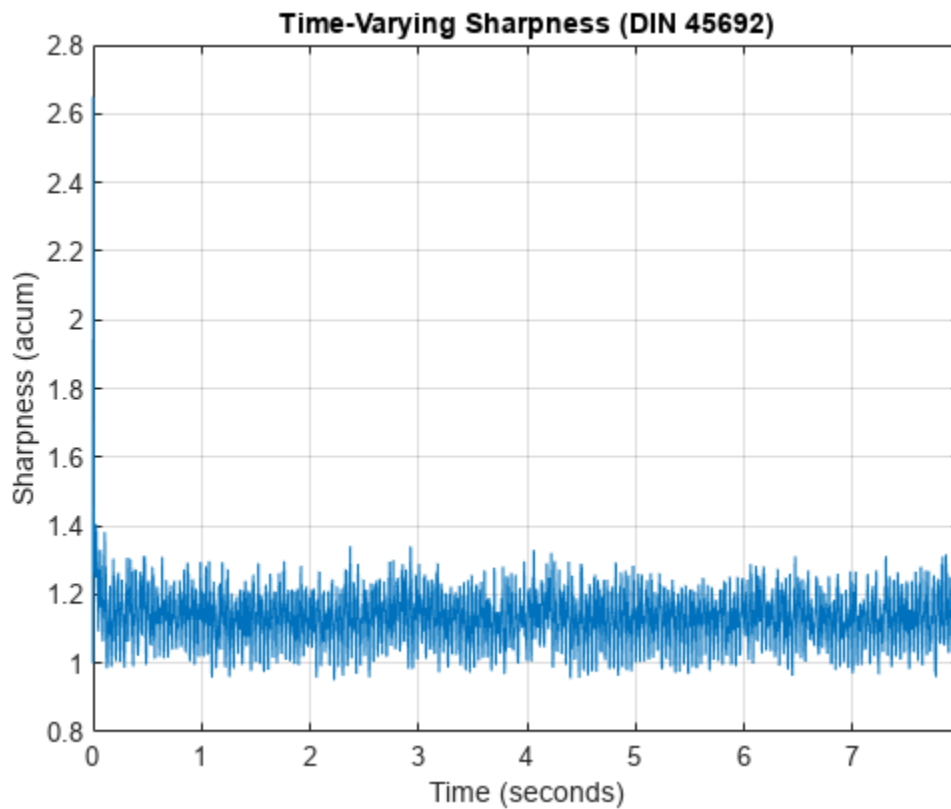
```
sharp = acousticSharpness(spec)
```

```
sharp = 1.1512
```

Pink noise has a sharpness of 2 acums. This means the engine noise is biased towards low frequencies.

Plot time-varying sharpness.

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

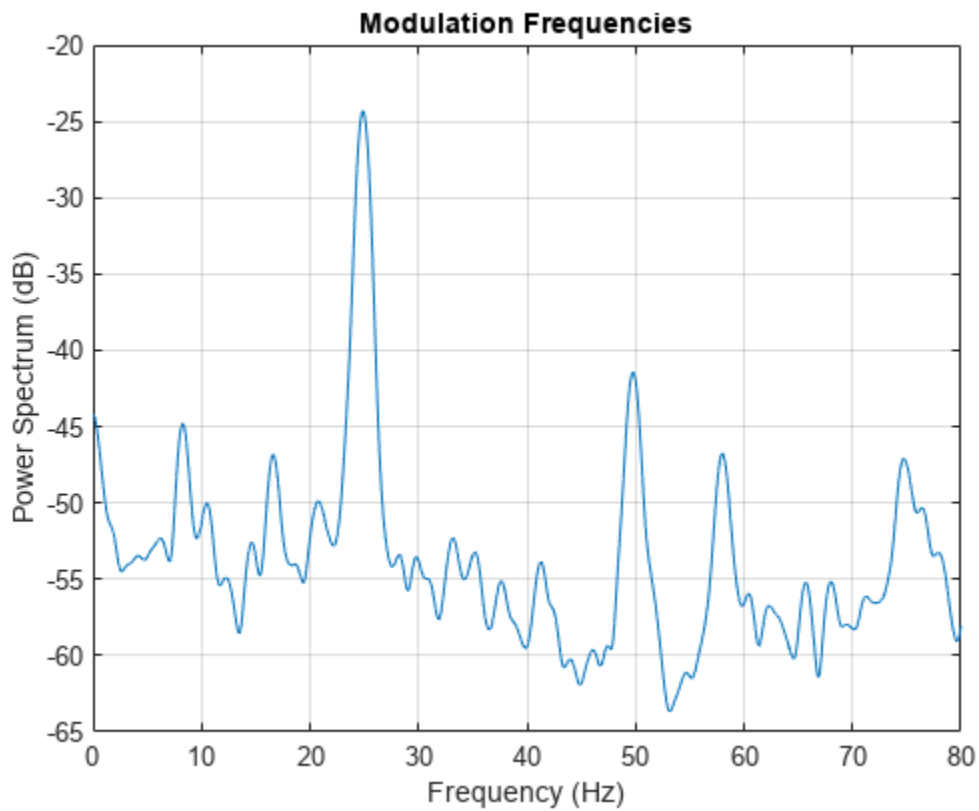


Fluctuation Strength

In the case of engine noise, low-frequency modulations contribute to the perceived annoyance level.

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

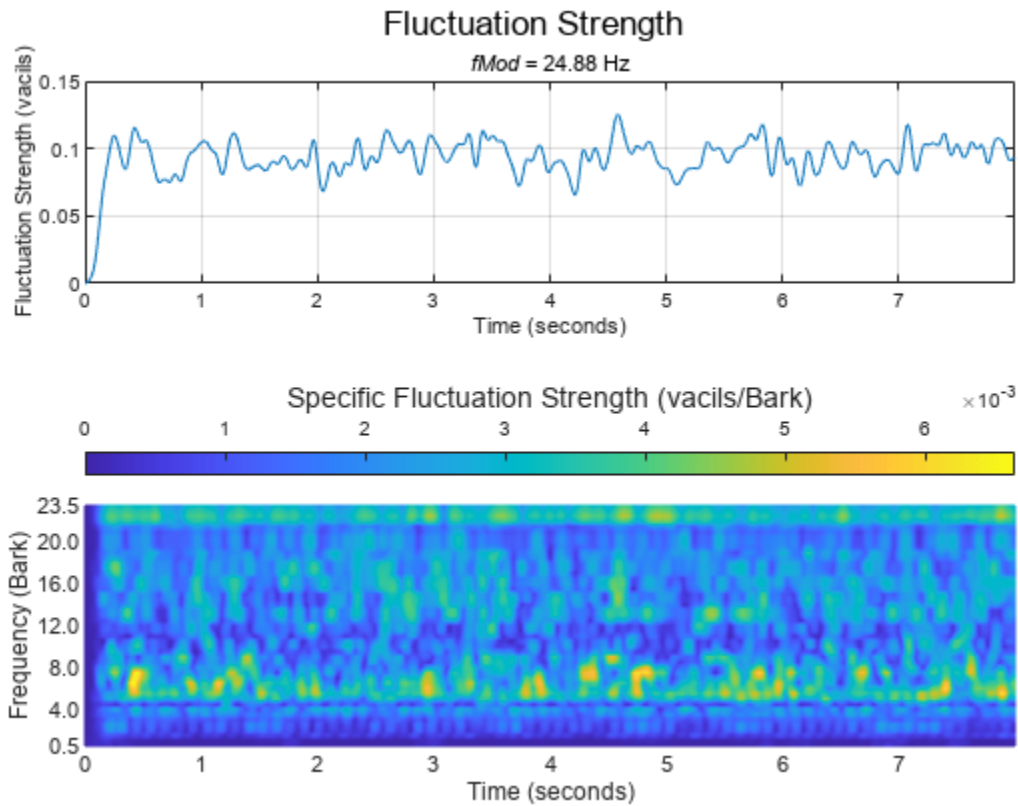
```
N = 2^nextpow2(size(x,1));  
xa = abs(x); % Use the rectified signal  
pspectrum(xa-mean(xa),FS,'FrequencyLimits',[0 80],'FrequencyResolution',1)  
title('Modulation Frequencies')
```



The modulation frequency peaks at 24.9 Hz. Below 30 Hz, modulation is perceived dominantly as fluctuation. There is a second peak at 49.7 Hz, which is in the range of roughness.

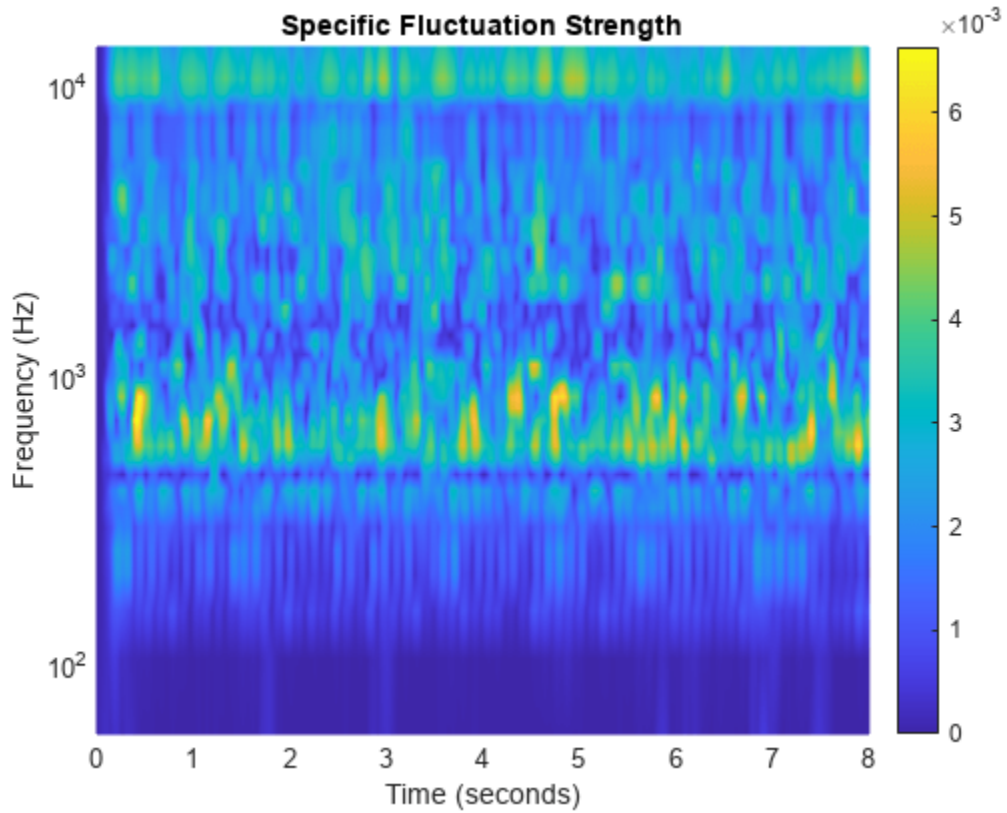
Use `acousticFluctuation` to compute the perceived fluctuation strength. The engine noise is relatively constant in this recording, so we have the algorithm automatically detect the most audible fluctuation frequency (`fMod`).

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



Interpret the results in Hertz instead of Bark. To reduce computations, reuse the previously computed time-varying specific loudness. Alternatively, you can also specify the modulation frequency that you are interested in.

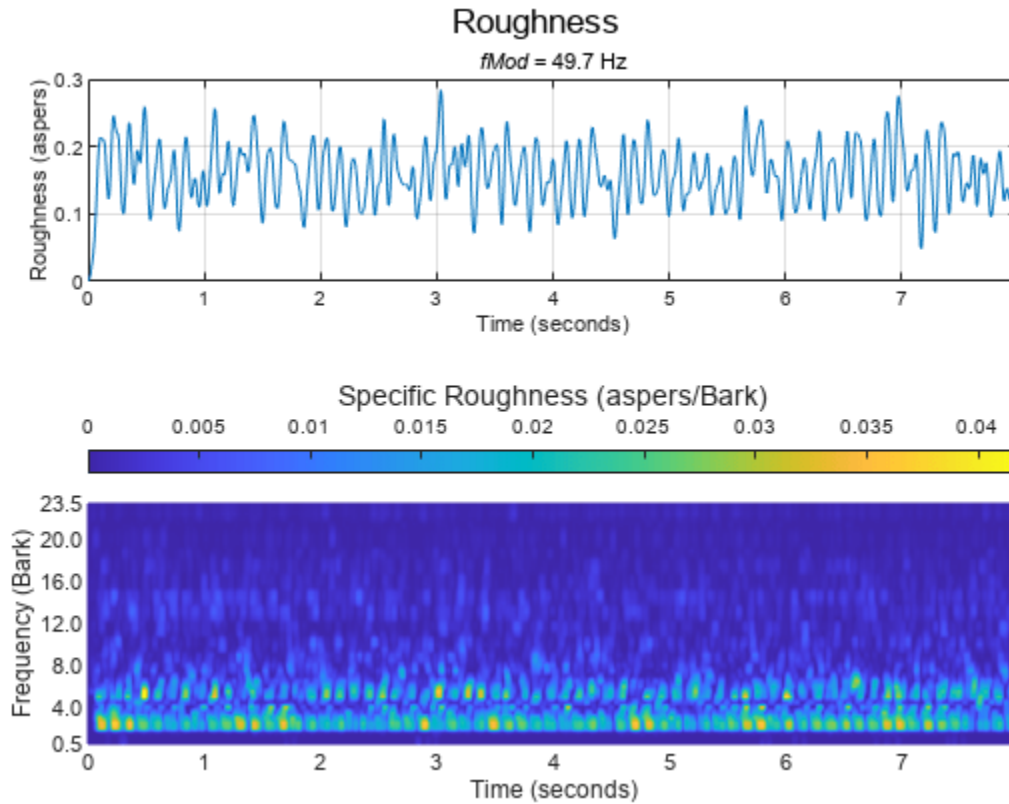
```
[vacil,spec,fMod] = acousticFluctuation(tvspec,'ModulationFrequency',24.9);
clf; % do not reuse previous subplot
flucHz = bark2hz(0.5:0.5:23.5);
spectime = 0:2e-3:2e-3*(size(spec,1)-1);
surf(spectime,flucHz,spec.','EdgeColor','interp');
set(gca,'View',[0 90],'YScale','log','YLim',flucHz([1,end]));
title('Specific Fluctuation Strength')
zlabel('Specific Fluctuation Strength (vacils/Bark)')
ylabel('Frequency (Hz)')
xlabel('Time (seconds)')
colorbar
```



Roughness

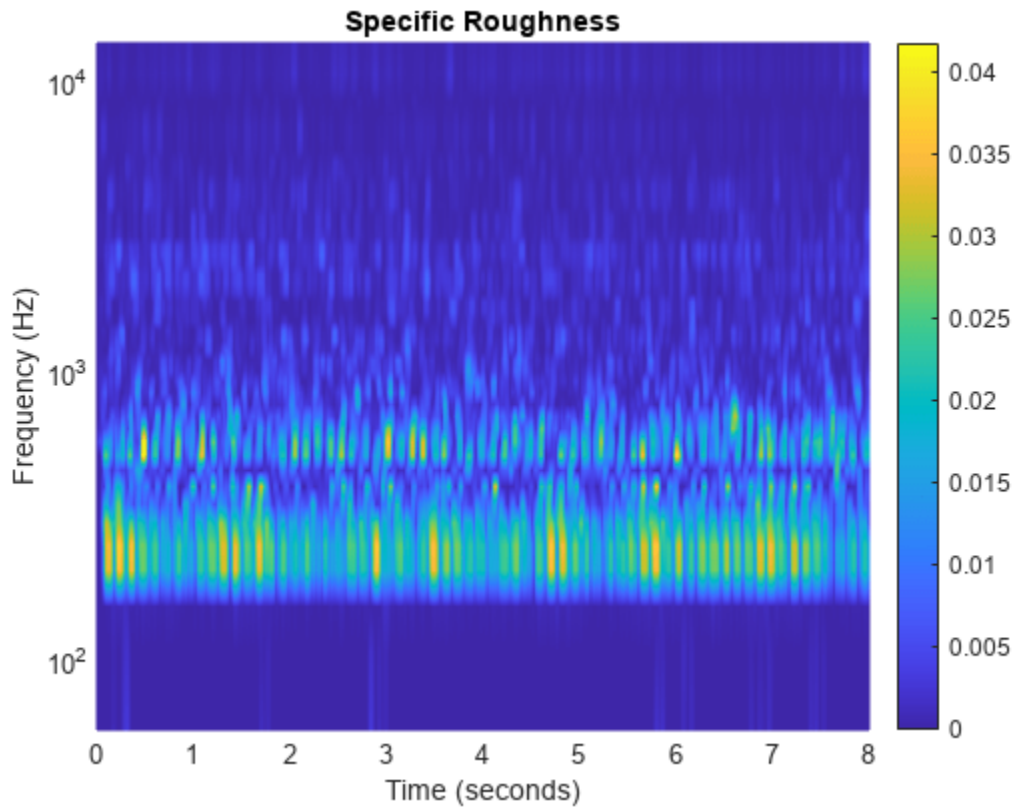
Use the `acousticRoughness` function to compute the perceived roughness of the signal. Let the algorithm automatically detect the most audible modulation frequency (`fMod`).

```
acousticRoughness(x,FS,calib)
```

Interpret the results in Hertz instead of Bark. To reduce computations, reuse the previously computed time-varying specific loudness. Specify the modulation frequency.

```
[asper,specR,fModR] = acousticRoughness(tvspechD,'ModulationFrequency',49.7);
clf; % do not reuse previous subplot
rougHz = bark2hz(0.5:0.5:23.5);
surf(spectimeHD,rougHz,specR.','EdgeColor','interp');
set(gca,'View',[0 90],'YScale','log','YLim',rougHz([1,end]));
title('Specific Roughness')
zlabel('Specific Roughness (aspers/Bark)')
ylabel('Frequency (Hz)')
xlabel('Time (seconds)')
colorbar
```



Sound Quality

For overall sound quality evaluation, combine the previous metrics to produce the psychoacoustic annoyance metric (defined by 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:

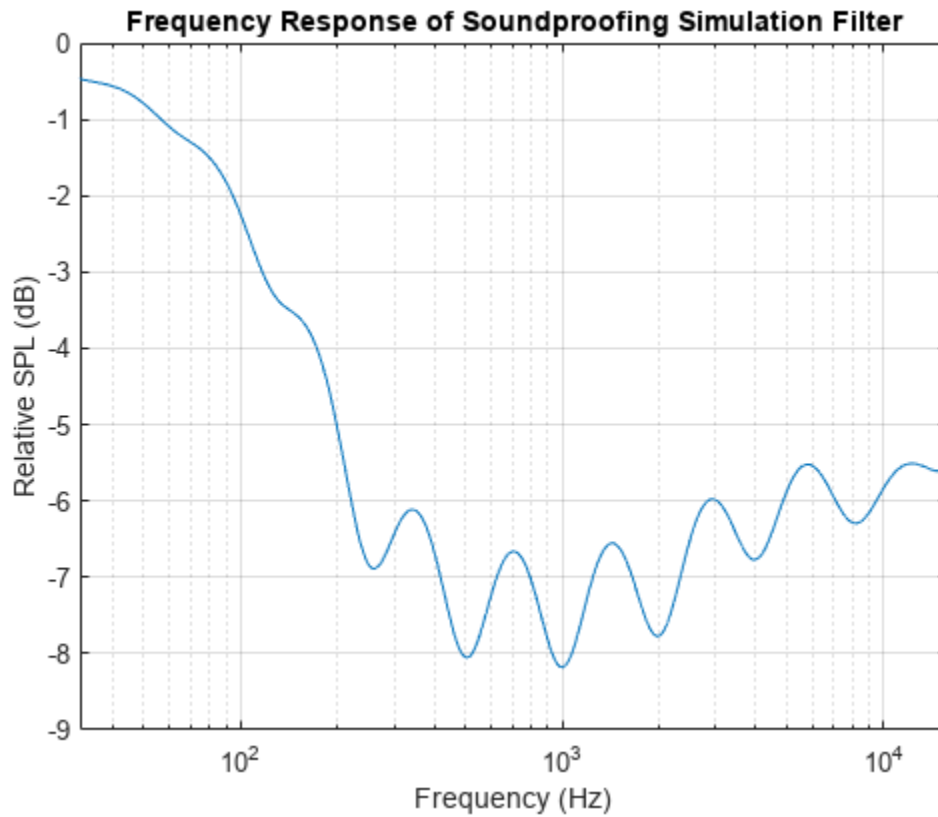
$$PA = N_5 \left(1 + \sqrt{w_S^2 + w_{FR}^2} \right)$$

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. Therefore, you can set 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`.

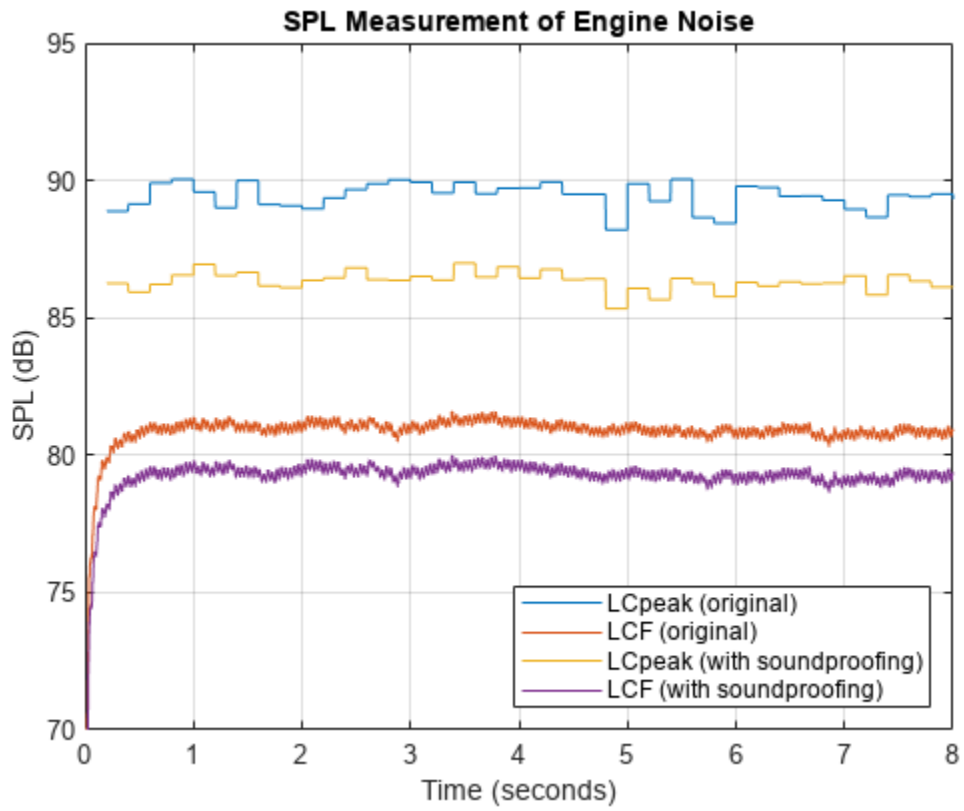


Filter the engine recording using the graphic EQ to simulate the soundproofing.

```
x2 = geq(x);
```

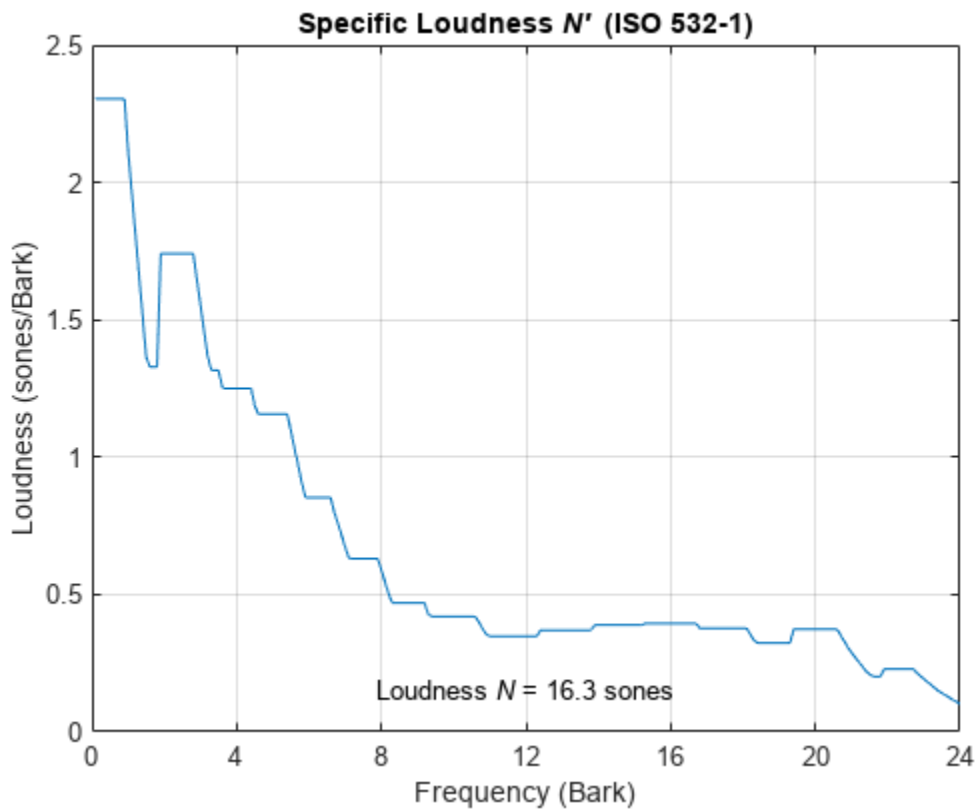
Compare the SPL with and without soundproofing. Reuse the same SPL meter settings, but use the filtered recording.

```
reset(spl)
[LCFnew,~,LCpeaknew] = spl(x2);
plot(t,LCpeak,t,LCF,t,LCpeaknew,t,LCFnew)
legend('LCpeak (original)', 'LCF (original)', ...
       'LCpeak (with soundproofing)', ...
       'LCF (with soundproofing)', ...
       'Location','southeast')
title('SPL Measurement of Engine Noise')
xlabel('Time (seconds)')
ylabel('SPL (dB)')
ylim([70 95])
grid on
```



Compare the perceived loudness measurements before and after soundproofing.

`acousticLoudness(x2,FS,calib)`



Loudness decreased from 23.8 to 16.3 sones. However, it might be easier to interpret loudness in phons. Convert the sone units to phons using `sone2phon`.

```
fprintf("Loudness without soundproofing:  \t%.1f phons\n",sone2phon(23.8));
```

```
Loudness without soundproofing:      85.7 phons
```

```
fprintf("Loudness with added soundproofing:\t%.1f phons\n",sone2phon(16.3));
```

```
Loudness with added soundproofing:    80.3 phons
```

```
fprintf("Perceived noise reduction:\t\t%.1f phons (dB SPL at 1 kHz)\n",sone2phon(23.8)-sone2phon(16.3));
```

```
Perceived noise reduction:           5.5 phons (dB SPL at 1 kHz)
```

After soundproofing, `acousticLoudness` shows the perception of the engine noise is approximately 5.5 dB quieter. Human perception of sound is limited at very low frequencies, where most of the engine noise is. The soundproofing is more effective at higher frequencies.

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

```
[~,spec2hd,perc2] = acousticLoudness(x2,FS,calib,"TimeVarying",true,"TimeResolution","high");
spec2 = spec2hd(1:4:end,:,:)
shp = acousticSharpness(spec2,'TimeVarying',true);
new_sharp = mean(shp(501:end))
```

```
new_sharp = 1.0796
```

Sharpness has decreased because the soundproofing is more effective at high frequency attenuation. It is below the threshold of 1.75, so it is ignored for the annoyance factor.

Now, compute the mean of fluctuation strength and roughness.

```
vacil2 = acousticFluctuation(spec2);  
f2 = mean(vacil2(501:end,1));  
asper2 = acousticRoughness(spec2hd);  
r2 = mean(asper2(2001:end,1));
```

Compute the new psychoacoustic annoyance factor. It has decreased, from 26.3 to 18.1.

```
N5hp = perc2(2); % N5 with soundproofing  
pahp = N5hp * (1 + abs(2.18/(N5hp^.4)*(.4*f2+.6*r2)))  
  
pahp = 18.0626
```

References

[1] Zwicker, Eberhard, and Hugo Fastl. *Psychoacoustics: Facts and Models*. Vol. 22. Springer Science & Business Media, 2013.

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 “Train Speech Command Recognition Model Using Deep Learning” on page 1-332.

Prerequisites

- MATLAB Coder Interface for Deep Learning Libraries
- ARM processor that supports the NEON extension
- ARM Compute Library version 20.02.1 (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 “Train Speech Command Recognition Model Using Deep Learning” on page 1-332.

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;
setExtractorParameters(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 = '20.02.1';
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.
### Compiling function(s) HelperSpeechCommandRecognitionRasPi ...
-----
Location of the generated elf : /home/pi/remoteBuildDir/MATLAB_ws/R2022a/W/Ex/ExampleManager/spo
### Using toolchain: GNU GCC Embedded Linux
### 'W:\Ex\ExampleManager\sporwal.Bdoc22a.j1844576\deeplearning_shared-ex00376115\codegen\exe\He
### Building 'HelperSpeechCommandRecognitionRasPi': make -j$(($(nproc)+1)) -Otarget -f HelperSp
-----
### Generating compilation report ...
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 = '172.18.231.92';
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);

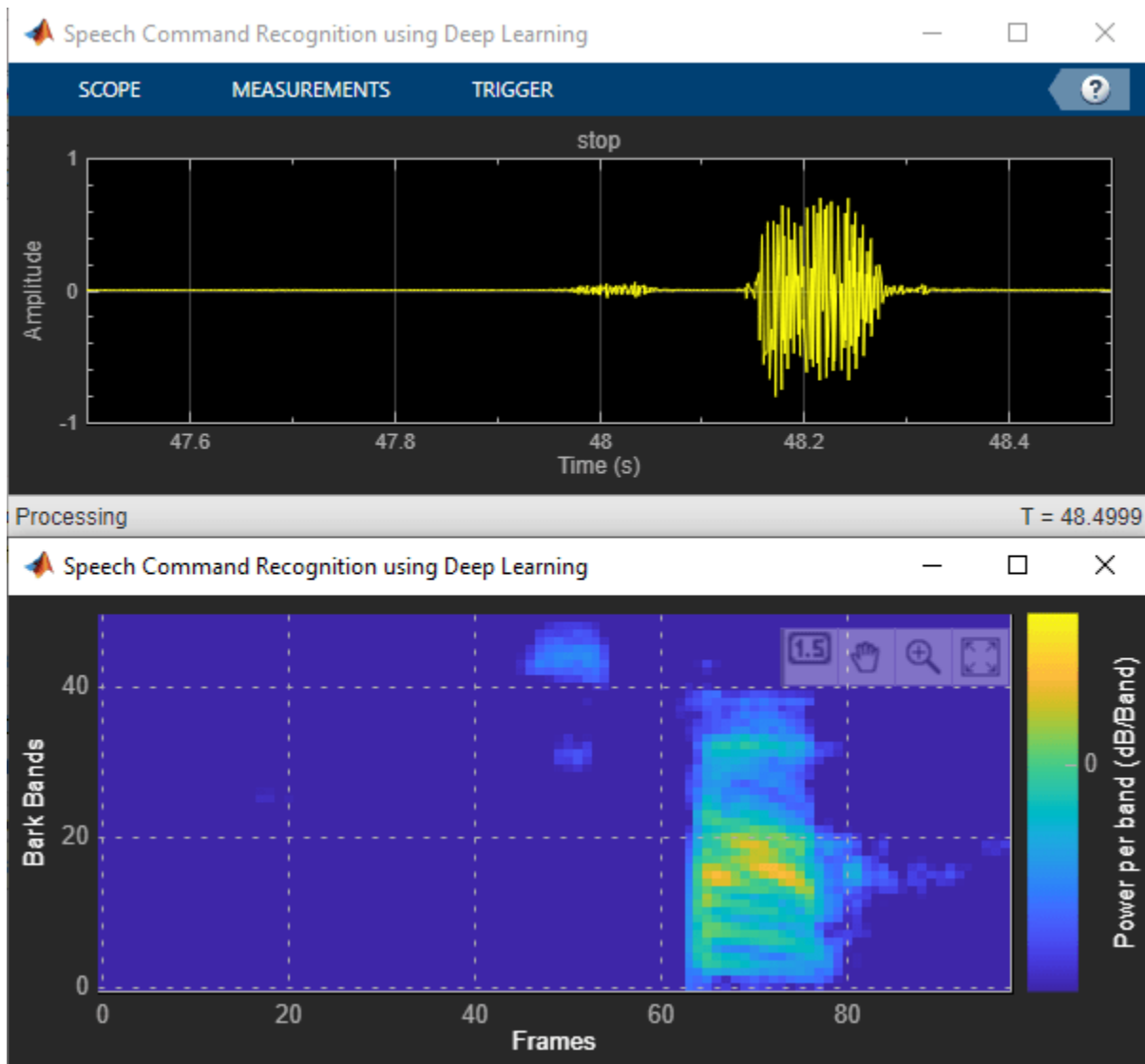
        % Extract auditory spectrogram
        spec = reshape(udpRec(1:numElementsPerSpectrogram), [numBands, numSpectrumPerSpectrogram]);

        % 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 processor-in-the-loop (PIL) workflow of Embedded Coder®. 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')}
```

```
Deploying code. This may take a few minutes.
### Compiling function(s) ProfileSpeechCommandRecognitionRaspi ...
### Connectivity configuration for function 'ProfileSpeechCommandRecognitionRaspi': 'Raspberry Pi'
### Using toolchain: GNU GCC Embedded Linux
### Creating 'W:\Ex\ExampleManager\sporwal.Bdoc22a.j1844576\deeplearning_shared-ex00376115\codegen\lib\ProfileSpeechCommandRecognitionRaspi_ca': make -j$((($nproc)+1)) -Otarget -f ProfileSpeechCommandRecognitionRaspi_ca
### Using toolchain: GNU GCC Embedded Linux
### Creating 'W:\Ex\ExampleManager\sporwal.Bdoc22a.j1844576\deeplearning_shared-ex00376115\codegen\lib\ProfileSpeechCommandRecognitionRaspi': make -j$((($nproc)+1)) -Otarget -f ProfileSpeechCommandRecognitionRaspi
Location of the generated elf : /home/pi/remoteBuildDir/MATLAB_ws/R2022a/W/Ex/ExampleManager/sporwal.Bdoc22a.j1844576\deeplearning_shared-ex00376115\codegen\lib\ProfileSpeechCommandRecognitionRaspi
-----
### Using toolchain: GNU GCC Embedded Linux
### 'W:\Ex\ExampleManager\sporwal.Bdoc22a.j1844576\deeplearning_shared-ex00376115\codegen\lib\ProfileSpeechCommandRecognitionRaspi': make -j$((($nproc)+1)) -Otarget -f ProfileSpeechCommandRecognitionRaspi
-----
### Generating compilation report ...
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 =
'W:\Ex\ExampleManager\sporwal.Bdoc22a.j1844576\deeplearning_shared-ex00376115\codegen\lib\ProfileSpeechCommandRecognitionRaspi\pil\ProfileSpeechCommandRecognitionRaspi.elf'
```

Code Execution Profiling Report for ProfileSpeechCommandRecognitionRaspi

The code execution profiling report provides metrics based on data collected from a SIL or PIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. See [Code Execution Profiling](#) for more information.

1. Summary

Total time	2023.5371
Unit of time	ms
Command	report(executionProfile, 'Units', 'seconds', 'ScaleFactor', '0.001', 'NumericFormat', '%0.4f');
Timer frequency (ticks per second)	1e+09
Profiling data created	11-Jun-2020 11:16:21

2. Profiled Sections of Code

Section	Maximum Execution Time in ms	Average Execution Time in ms	Maximum Self Time in ms	Average Self Time in ms	Calls
ProfileSpeechCommandRecognitionRaspi_initialize	0.3660	0.3660	0.3660	0.3660	1
ProfileSpeechCommandRecognitionRaspi	44.1807	20.2317	44.1807	20.2317	100
ProfileSpeechCommandRecognitionRaspi_terminate	0.0020	0.0020	0.0020	0.0020	1

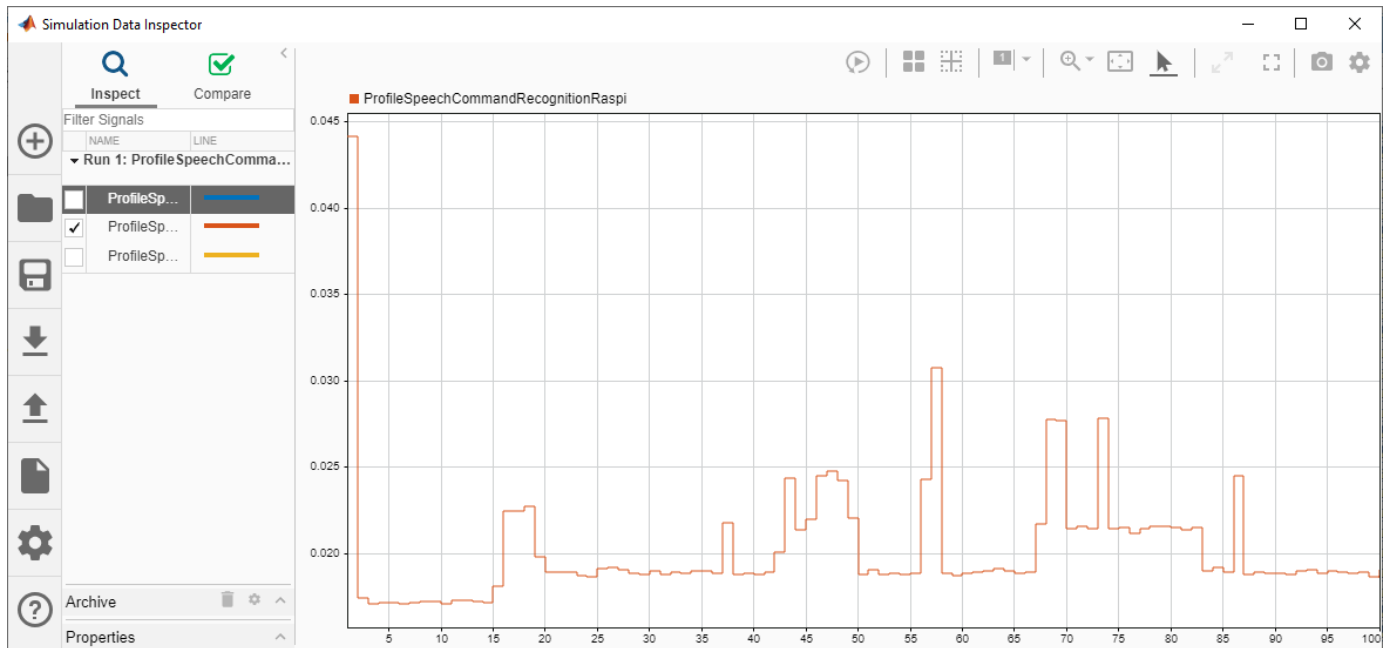
3. Definitions

Execution Time: Time between start and end of code section.

Self Time: Execution time, excluding time in child sections.

OK Help

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 “Train Speech Command Recognition Model Using Deep Learning” on page 1-332.

Prerequisites

- The MATLAB Coder Interface for Deep Learning Libraries 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 “Train Speech Command Recognition Model Using Deep Learning” on page 1-332.

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;
setExtractorParameters(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('mklDnn');
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')}
### Compiling function(s) HelperSpeechCommandRecognition ...
-----
[1/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MWTensorBase.cpp
[2/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MWElementwiseAffineLayer.cpp
```

```
[3/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwMaxPoolingLayer.cpp
[4/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwInputLayerImpl.cpp
[5/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwInputLayer.cpp
[6/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwOutputLayer.cpp
[7/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwFCLayer.cpp
[8/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwCNNLayer.cpp
[9/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwOutputLayerImpl.cpp
[10/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwFusedConvReLULayer.cpp
[11/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwMaxPoolingLayerImpl.cpp
[12/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
HelperSpeechCommandRecognition_data.cpp
[13/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
HelperSpeechCommandRecognition_terminate.cpp
[14/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
colon.cpp
[15/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
HelperSpeechCommandRecognition_initialize.cpp
[16/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwElementwiseAffineLayerImpl.cpp
[17/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
rt_nonfinite.cpp
[18/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwFCLayerImpl.cpp
[19/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwFusedConvReLULayerImpl.cpp
[20/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
eml_int_forloop_overflow_check.cpp
[21/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwSoftmaxLayerImpl.cpp
[22/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
stft.cpp
[23/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
sort.cpp
[24/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwSoftmaxLayer.cpp
[25/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
extractSpeechFeatures.cpp
[26/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
HelperSpeechCommandRecognition.cpp
[27/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
DeepLearningNetwork.cpp
[28/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
sortIdx.cpp
[29/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
_coder_HelperSpeechCommandRecognition_api.cpp
[30/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MwCNNLayerImpl.cpp
[31/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
permute.cpp
```

```

[32/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
predict.cpp
[33/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
_coder_HelperSpeechCommandRecognition_info.cpp
[34/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
HelperSpeechCommandRecognition_mexutil.cpp
[35/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MWTTargetNetworkImpl.cpp
[36/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
_coder_HelperSpeechCommandRecognition_mex.cpp
[37/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
indexShapeCheck.cpp
[38/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
cpp_mexapi_version.cpp
[39/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MWCcustomLayerForMKLDNN.cpp
[40/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
MWMkldnnUtils.cpp
[41/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
mtimes.cpp
[42/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
computeDFT.cpp
[43/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
formatSTFTOutput.cpp
[44/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED
AsyncBuffer.cpp
[45/45] link build\win64\MWCNNLayer.obj build\win64\MWElementwiseAffineLayer.obj build\win64\MWF
Creating library HelperSpeechCommandRecognition_mex.lib and object HelperSpeechCommandRecogni

```

```

-----
### Generating compilation report ...
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

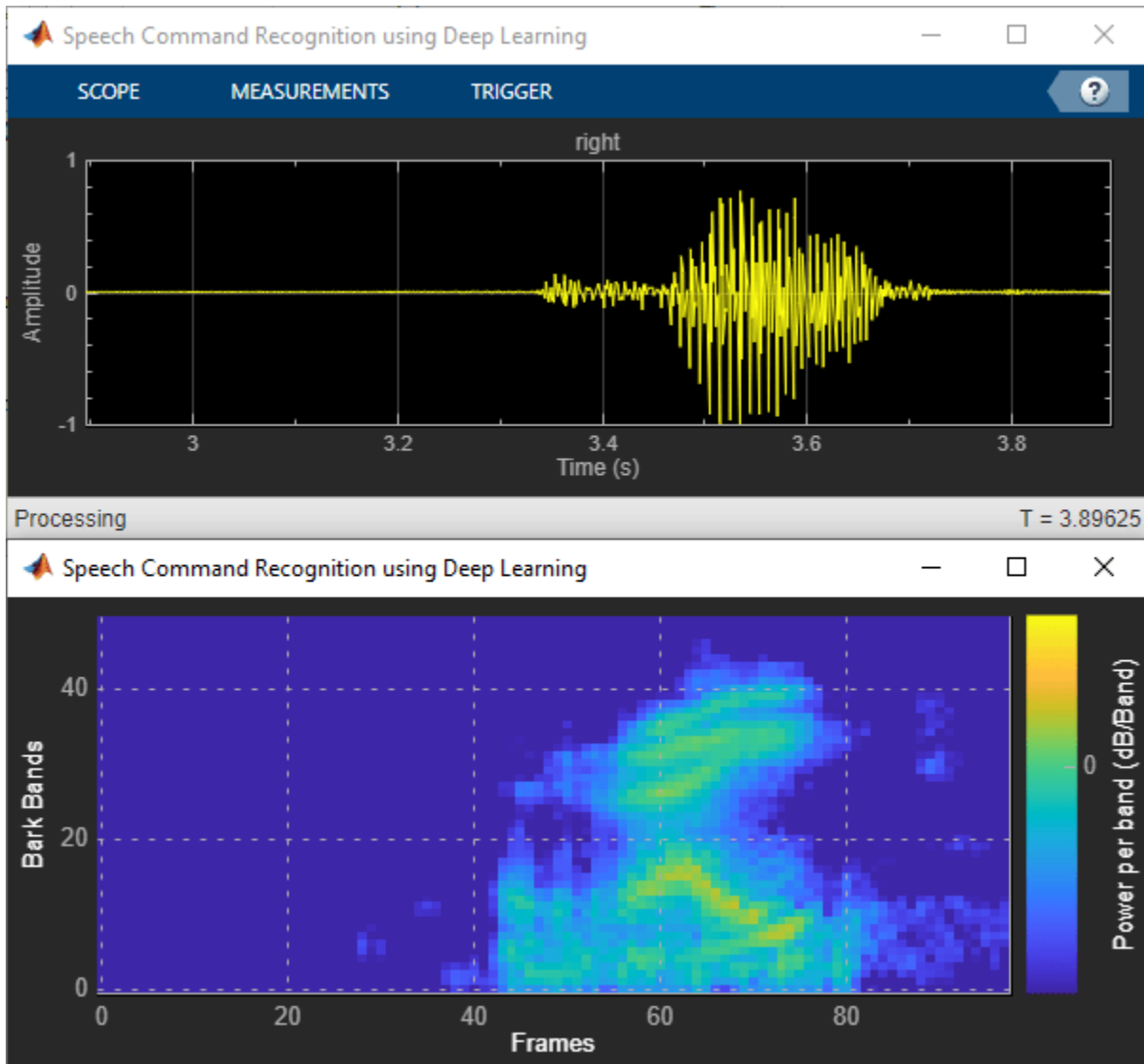
```

```

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 = 6.8212 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 = 1.3347 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 = 5.1107

Speech Command Recognition in Simulink

This example shows a Simulink® model that detects the presence of speech commands in audio. The model uses a pretrained convolutional neural network to recognize a given set of commands.

Speech Command Recognition Model

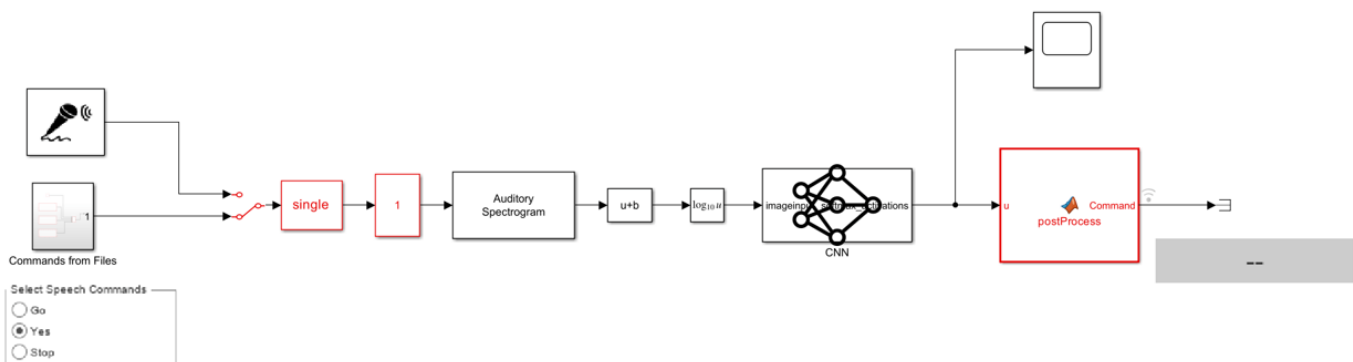
The model recognizes these speech commands:

- "yes"
- "no"
- "up"
- "down"
- "left"
- "right"
- "on"
- "off"
- "stop"
- "go"

The model uses a pretrained convolutional deep learning network. Refer to the example “Train Speech Command Recognition Model Using Deep Learning” on page 1-332 for details on the architecture of this network and how you train it.

Open the model.

```
model = "speechCommandRecognition";
open_system(model)
```



Copyright 2020-2021, The MathWorks, Inc.

The model breaks the audio stream into one-second overlapping segments. A bark spectrogram is computed from each segment. The spectrograms are fed to the pretrained network.

Use the manual switch to select either a live stream from your microphone or read commands stored in audio files. For commands on file, use the rotary switch to select one of three commands (Go, Yes, or Stop).

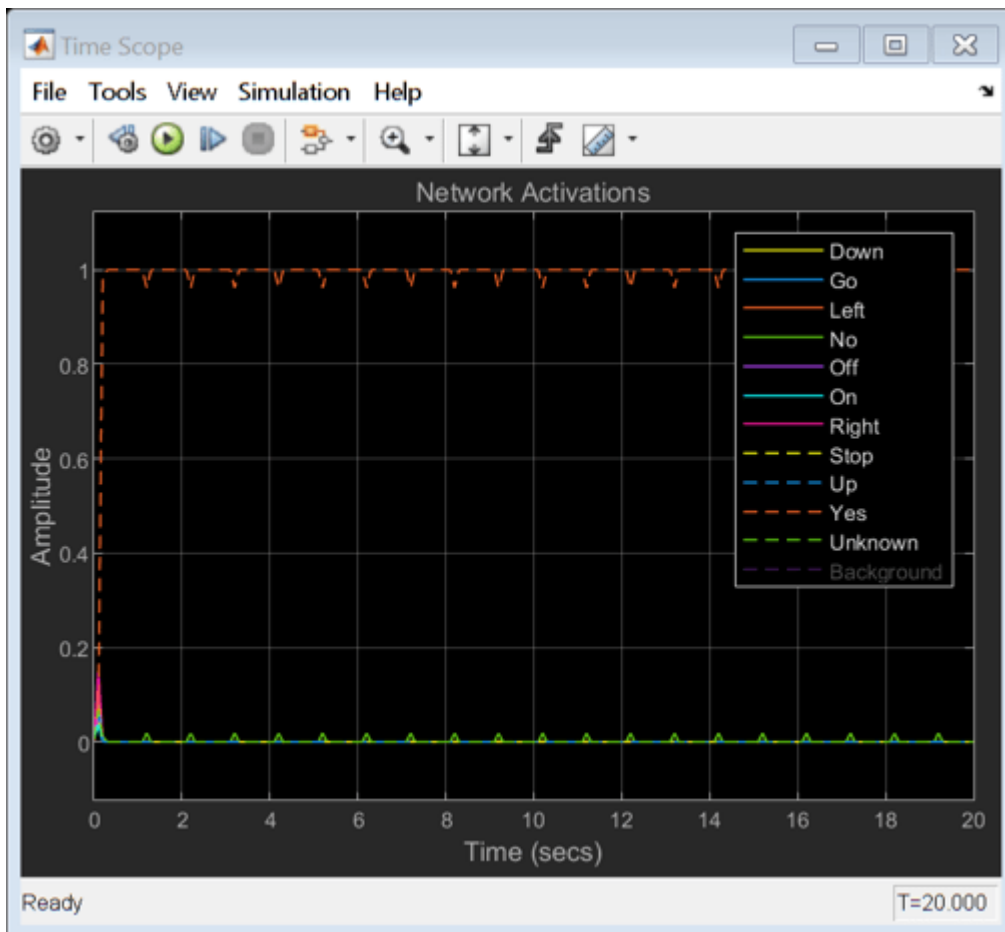
Auditory Spectrogram Extraction

The deep learning network was trained on auditory spectrograms computed using an `audioFeatureExtractor`. The Auditory Spectrogram block in the model has been configured to extract the same features as the network was trained on.

Run the model

Simulate the model for 20 seconds.

```
set_param(model, StopTime="20");
open_system(model + "/Time Scope")
sim(model);
```



The recognized command is printed in the display block. The network activations, which give a level of confidence in the different supported commands, are displayed in a time scope.

Close the model.

```
close_system(model,0)
```

See Also

Auditory Spectrogram

Related Examples

- “Train Speech Command Recognition Model Using Deep Learning” on page 1-332
- “Speech Command Recognition Using Deep Learning” on page 1-929
- “Speech Command Recognition Code Generation with Intel MKL-DNN Using Simulink” on page 1-881

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 1-64]. 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 1-64] 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 1-368.

Create Harmonic-Percussive Mixture

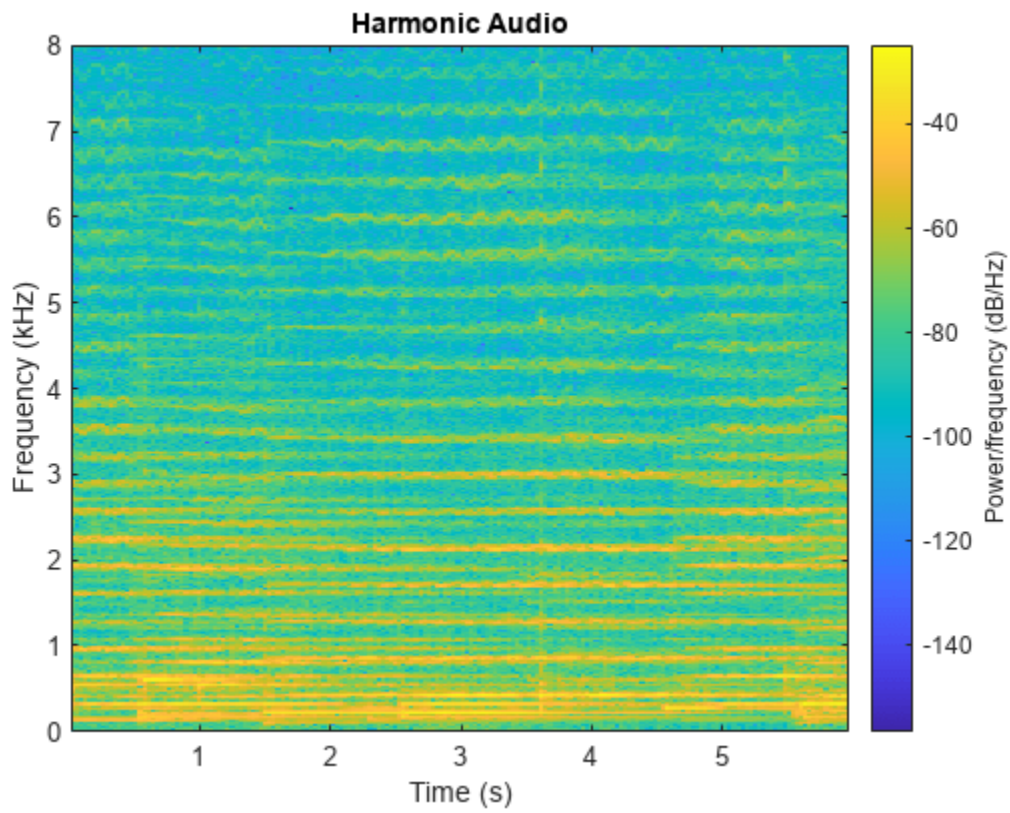
Read in harmonic and percussive audio files. Both have a sample rate of 16 kHz.

```
[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)
```

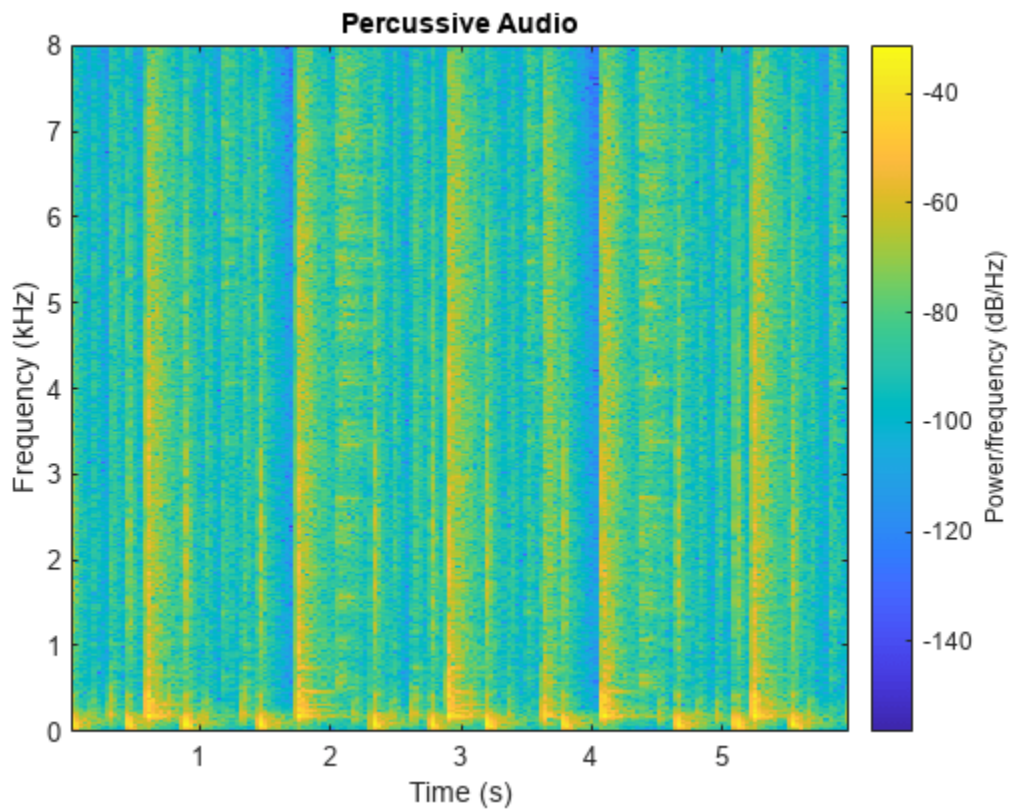
```
spectrogram(harmonicAudio,1024,512,1024,fs,"yaxis")  
title("Harmonic Audio")
```



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

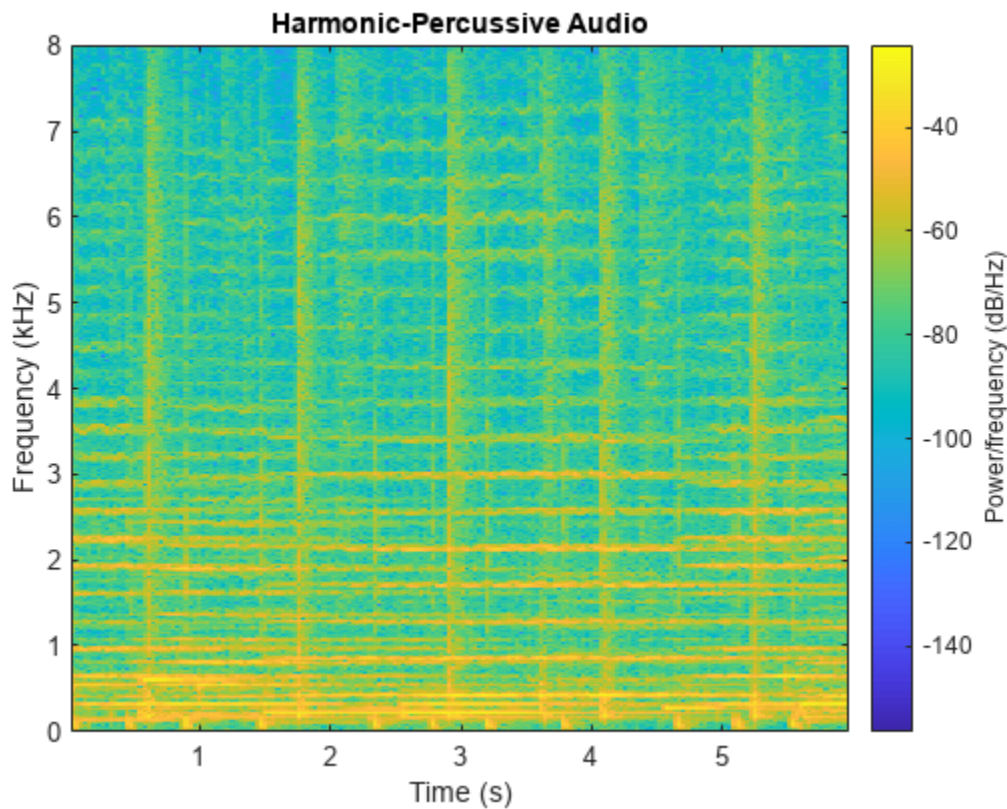
```
sound(percussiveAudio, fs)
```

```
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 1-64] 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, FrequencyRange="onesided");
ymag = abs(y);
```

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 1-64]. Plot the power spectrum of the harmonic-enhanced audio.

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

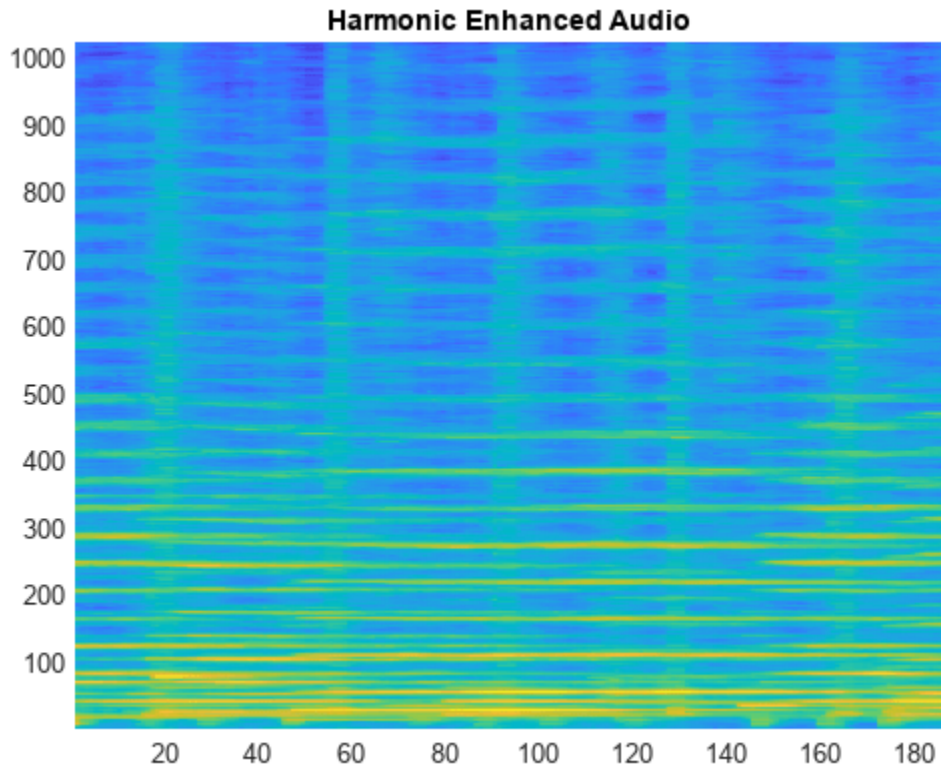
```
surf(log10(ymagharm.^2), EdgeColor="none")
```



```

title("Harmonic Enhanced Audio")
view([0,90])
axis tight

```



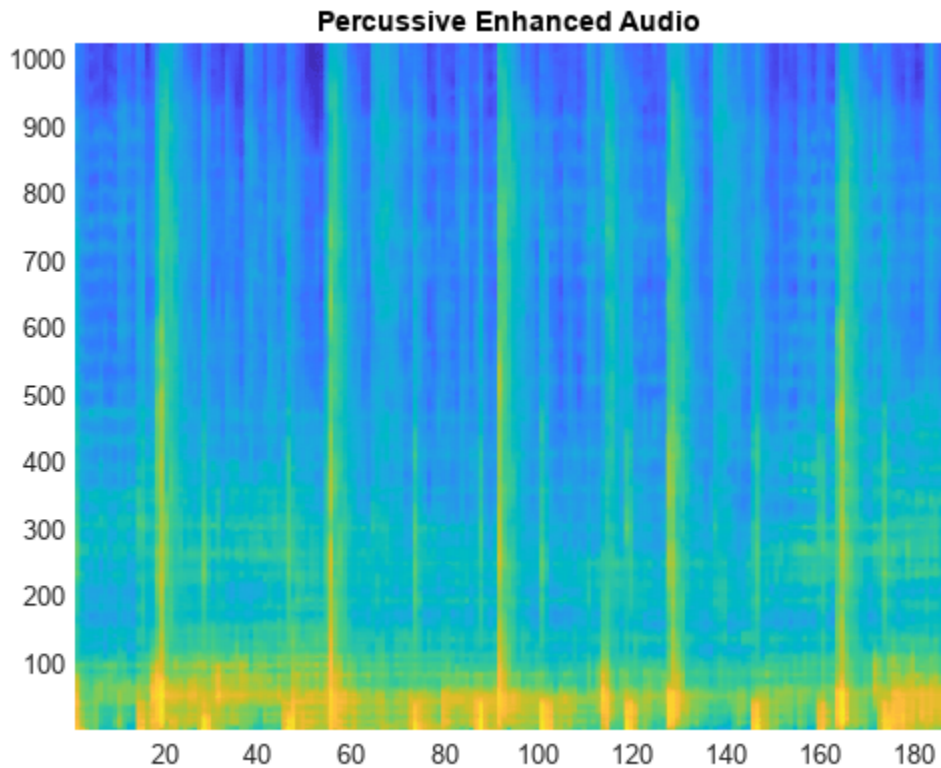
Apply median smoothing along the frequency axis to enhance the percussive audio and diminish the harmonic audio. Use a filter length of 500 Hz, as suggested by [1 on page 1-64]. Plot the power spectrum of the percussive-enhanced audio.

```

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

surf(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.*y;
yperc = percussiveMask.*y;
```

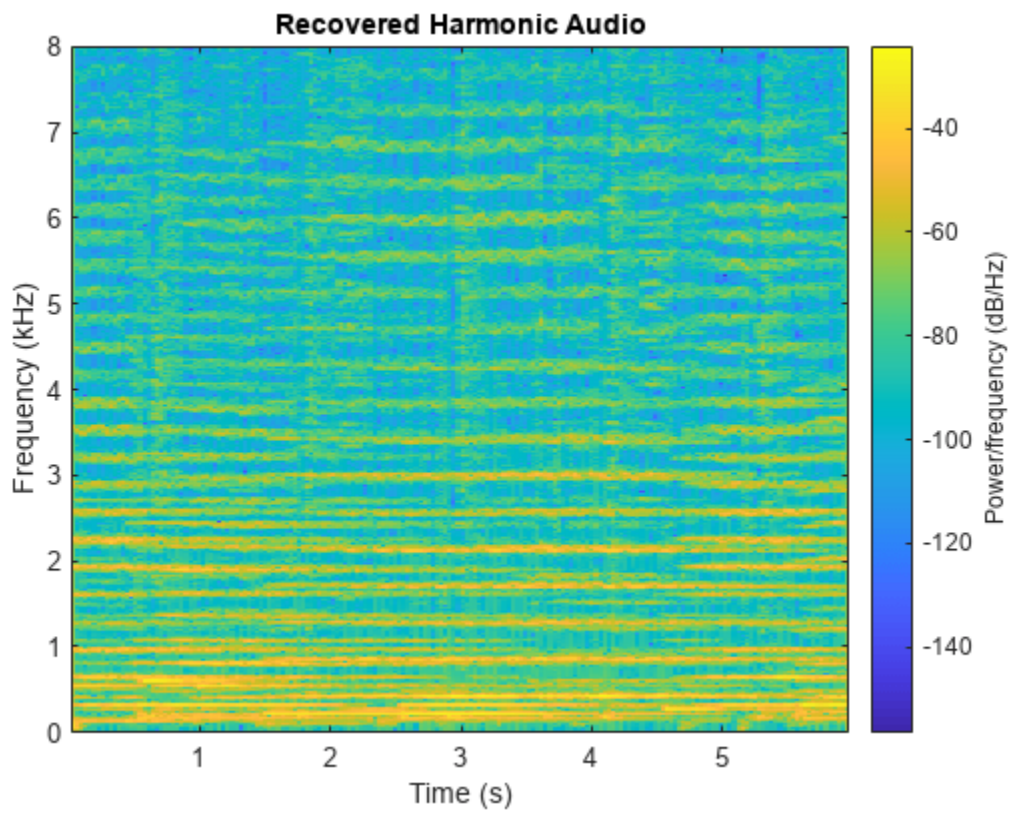
Perform the inverse short-time Fourier transform to return the signals to the time domain.

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

Listen to the recovered harmonic audio and plot the spectrogram.

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

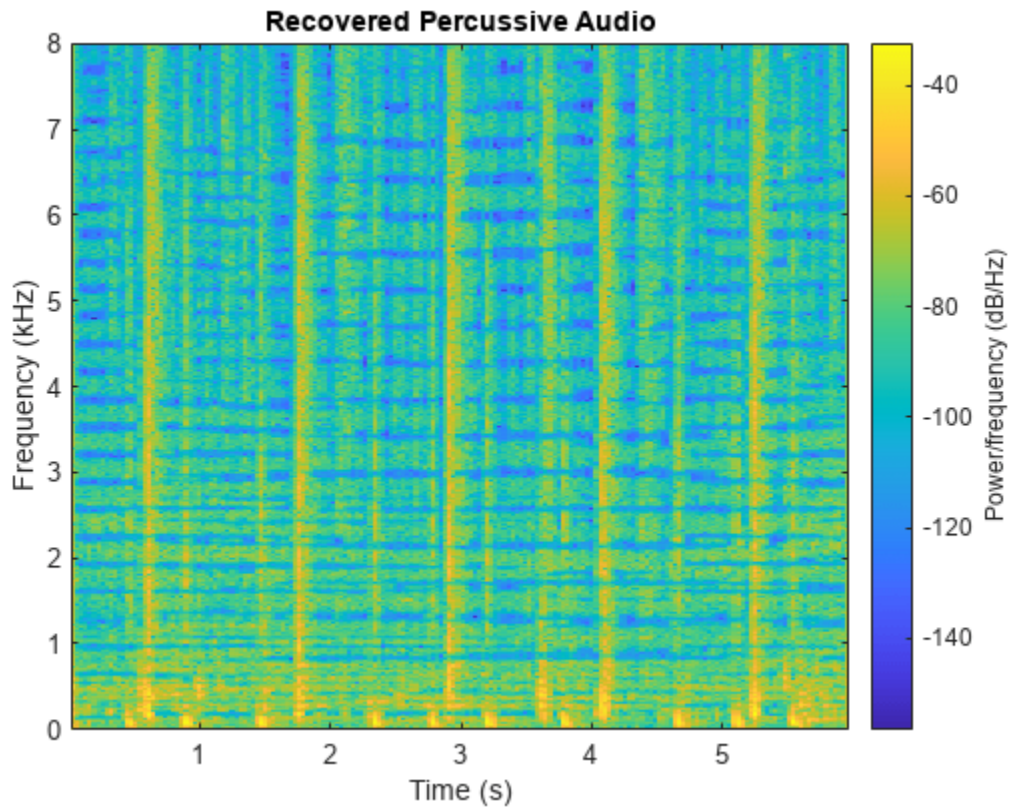
```
spectrogram(h,1024,512,1024,fs,"yaxis")  
title("Recovered Harmonic Audio")
```



Listen to the recovered percussive audio and plot the spectrogram.

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

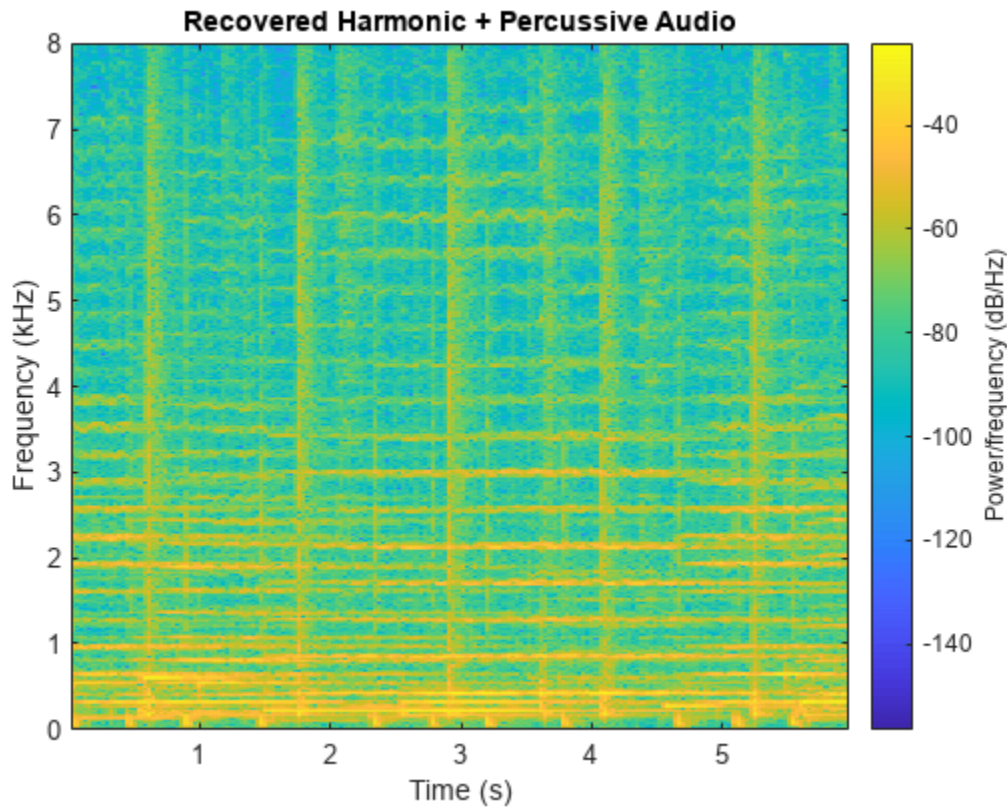
```
spectrogram(p,1024,512,1024,fs,"yaxis")  
title("Recovered Percussive Audio")
```



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 1-64], 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 1-46 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, FrequencyRange="onesided");
ymag = abs(y);

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.

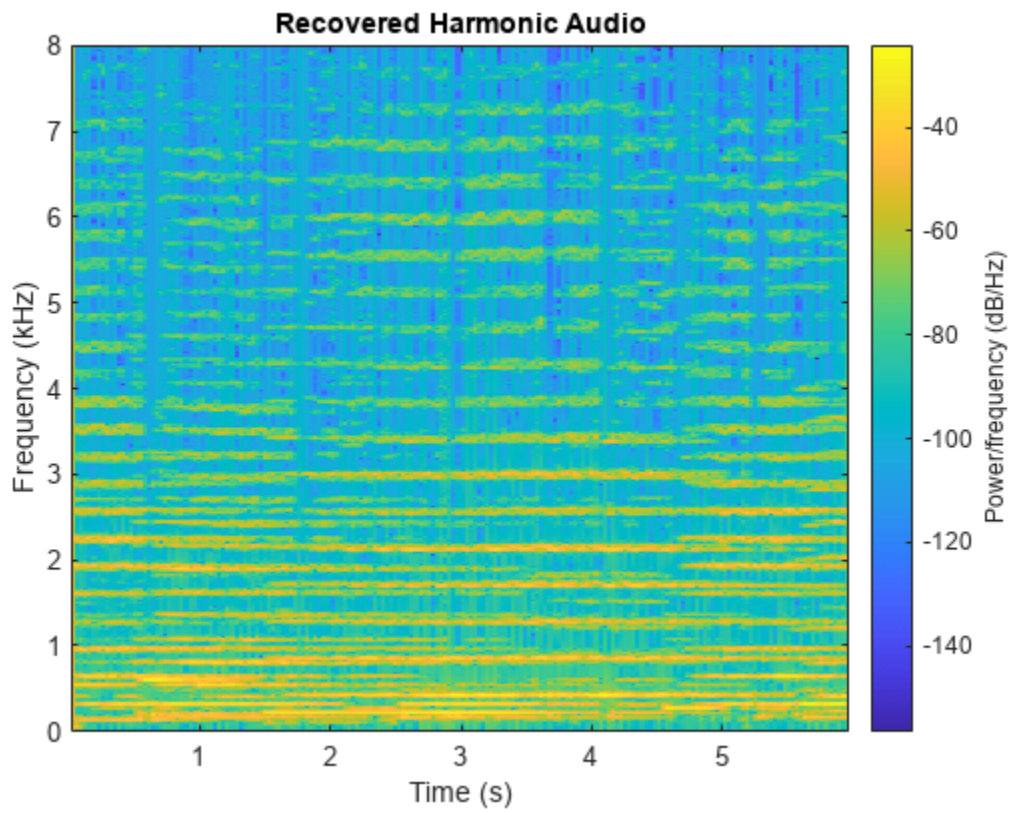
```
threshold = 0.65  ;  
harmonicMask = ymagharm > (totalMagnitudePerBin*threshold);  
percussiveMask = ymagperc > (totalMagnitudePerBin*threshold);  
residualMask = ~(harmonicMask+percussiveMask);
```

Perform the same steps described in HPSS Using Binary Mask on page 1-46 to return the masked signals to the time domain.

```
yharm = harmonicMask.*y;  
yperc = percussiveMask.*y;  
yresi = residualMask.*y;  
  
h = istft(yharm,Window=win,OverlapLength=overlapLength, ...  
    FFTLength=fftLength,ConjugateSymmetric=true,FrequencyRange="onesided");  
p = istft(yperc,Window=win,OverlapLength=overlapLength, ...  
    FFTLength=fftLength,ConjugateSymmetric=true,FrequencyRange="onesided");  
r = istft(yresi,Window=win,OverlapLength=overlapLength, ...  
    FFTLength=fftLength,ConjugateSymmetric=true,FrequencyRange="onesided");
```

Listen to the recovered harmonic audio and plot the spectrogram.

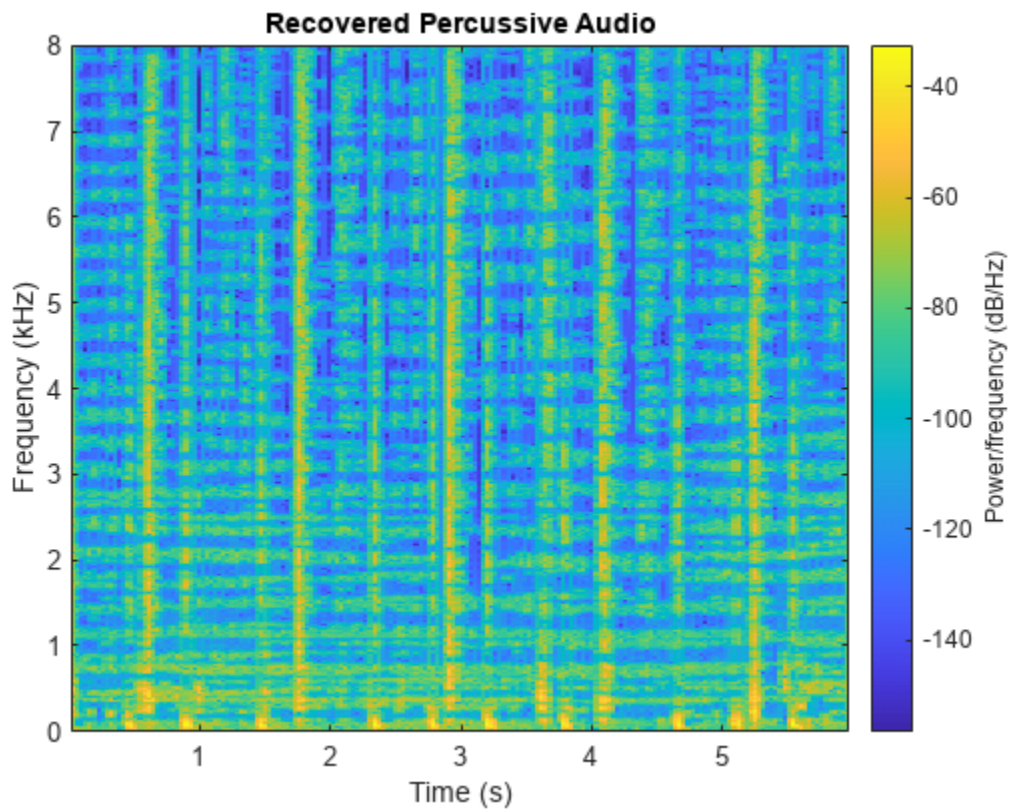
```
sound(h,fs)  
  
spectrogram(h,1024,512,1024,fs,"yaxis")  
title("Recovered Harmonic Audio")
```



Listen to the recovered percussive audio and plot the spectrogram.

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

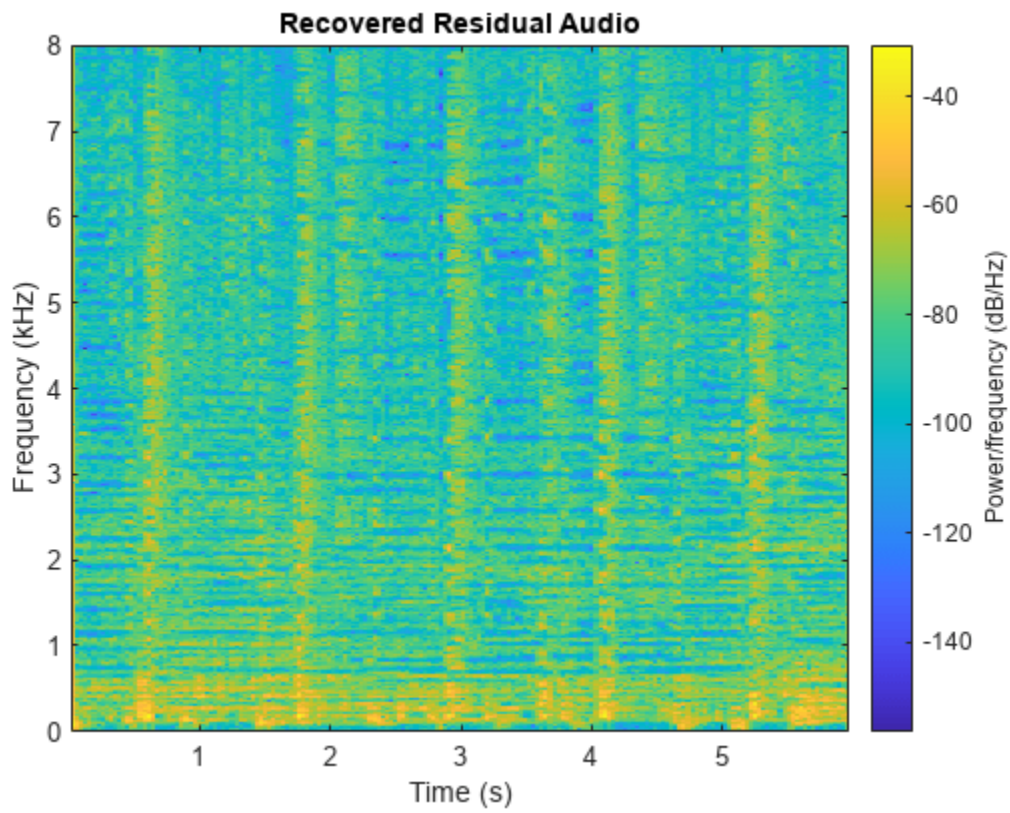
```
spectrogram(p, 1024, 512, 1024, fs, "yaxis")  
title("Recovered Percussive Audio")
```



Listen to the recovered residual audio and plot the spectrogram.

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

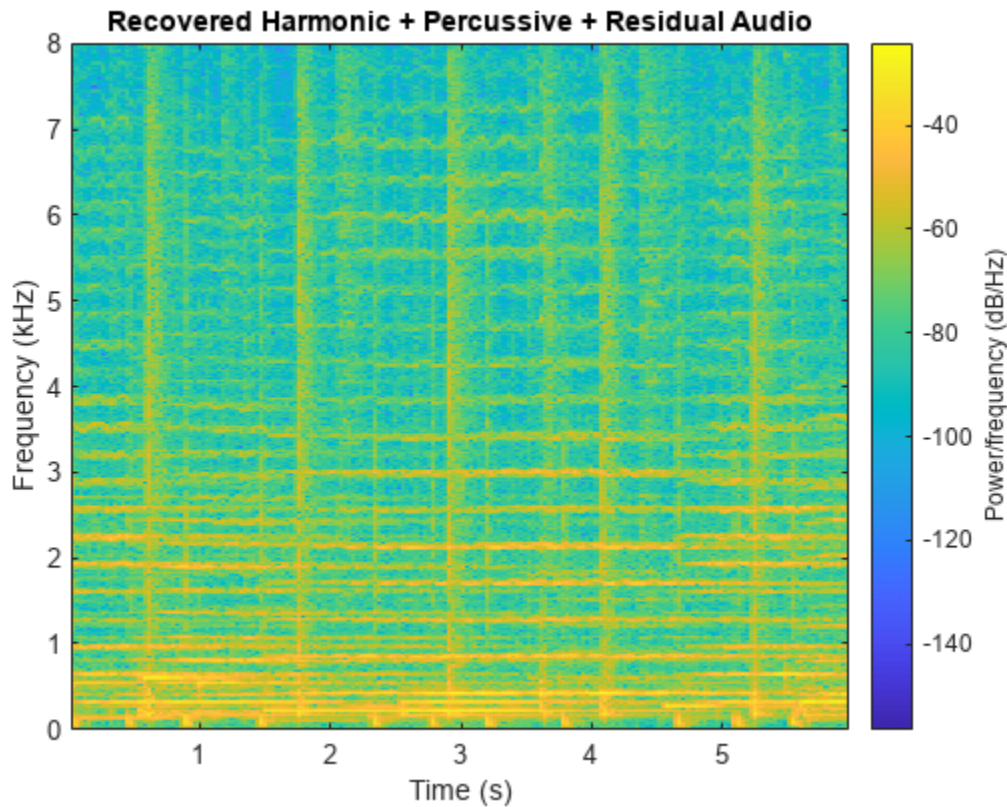
```
spectrogram(r, 1024, 512, 1024, fs, "yaxis")  
title("Recovered Residual Audio")
```

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 1-46 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, FrequencyRange="onesided");
ymag = abs(y);

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 1-46 to return the masked signals to the time domain.

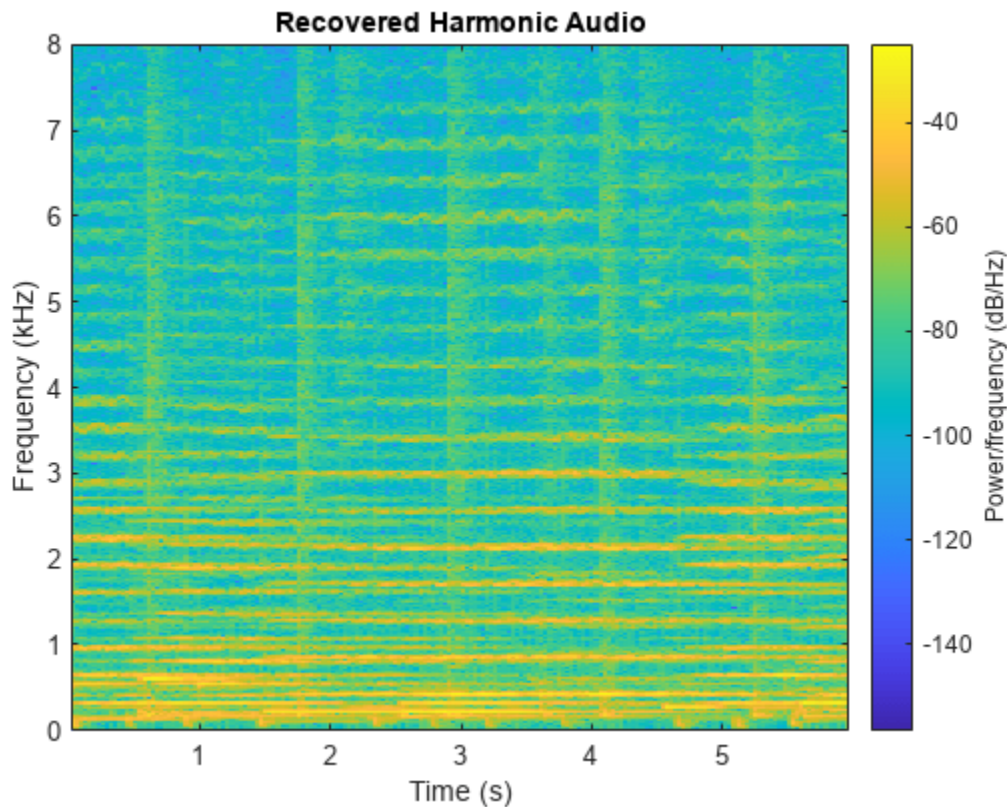
```
yharm = harmonicMask.*y;
yperc = percussiveMask.*y;
```

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

Listen to the recovered harmonic audio and plot the spectrogram.

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

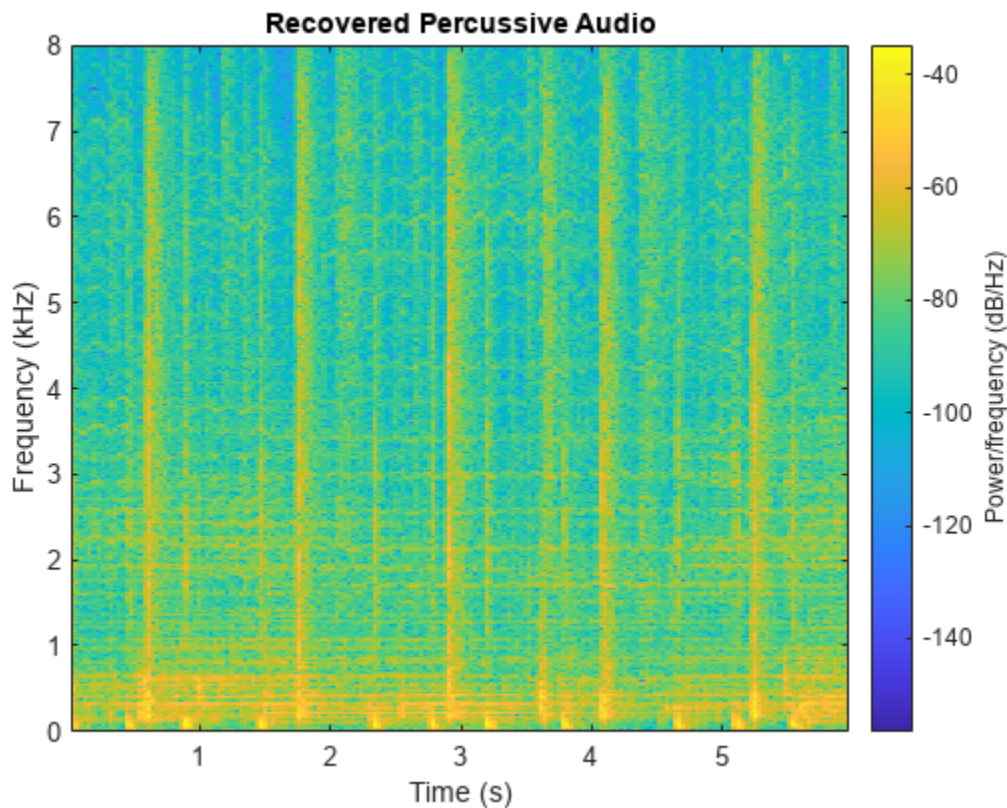
```
spectrogram(h,1024,512,1024,fs,"yaxis")
title("Recovered Harmonic Audio")
```



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',FFTLENGTH)` 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 1-64] 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 1-64] 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;

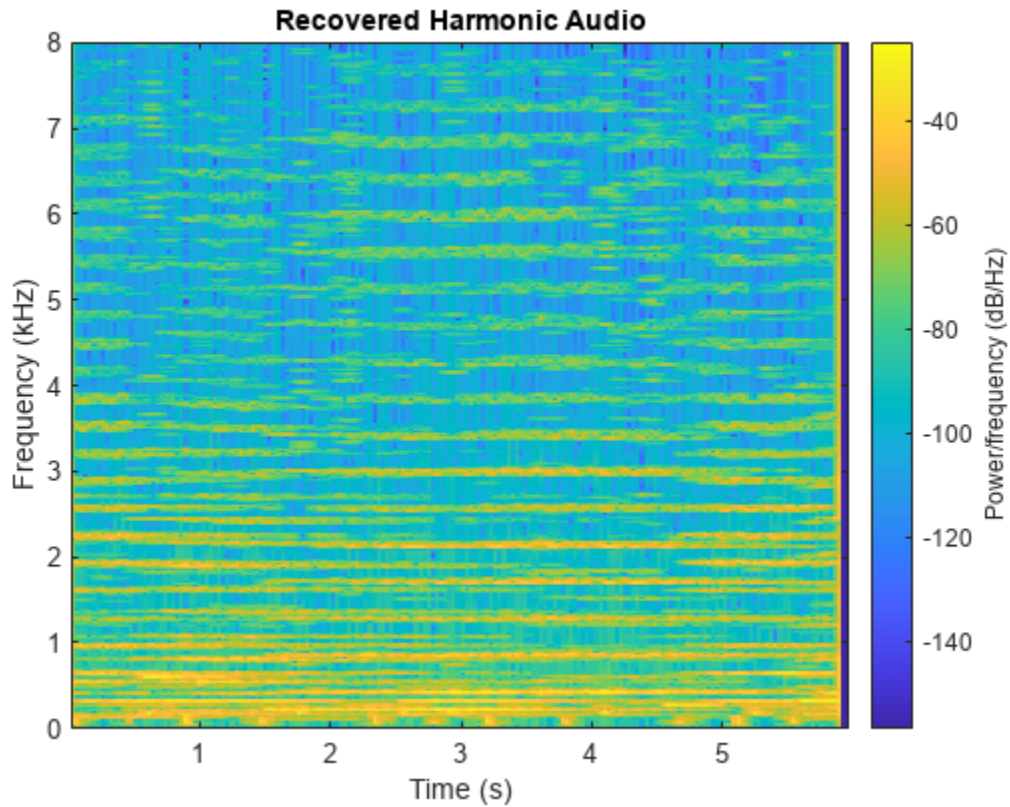
threshold2 = 0.6  ;
N2 = 256  ;
[h2,p2,r2] = HelperHPSS(mix1, fs, Threshold=threshold2, Window=sqrt(hann(N2, "periodic")), OverlapLength=512);
h = h1;
```

```
p = p2;  
r = h2 + r2;
```

Listen to the recovered percussive audio and plot the spectrogram.

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

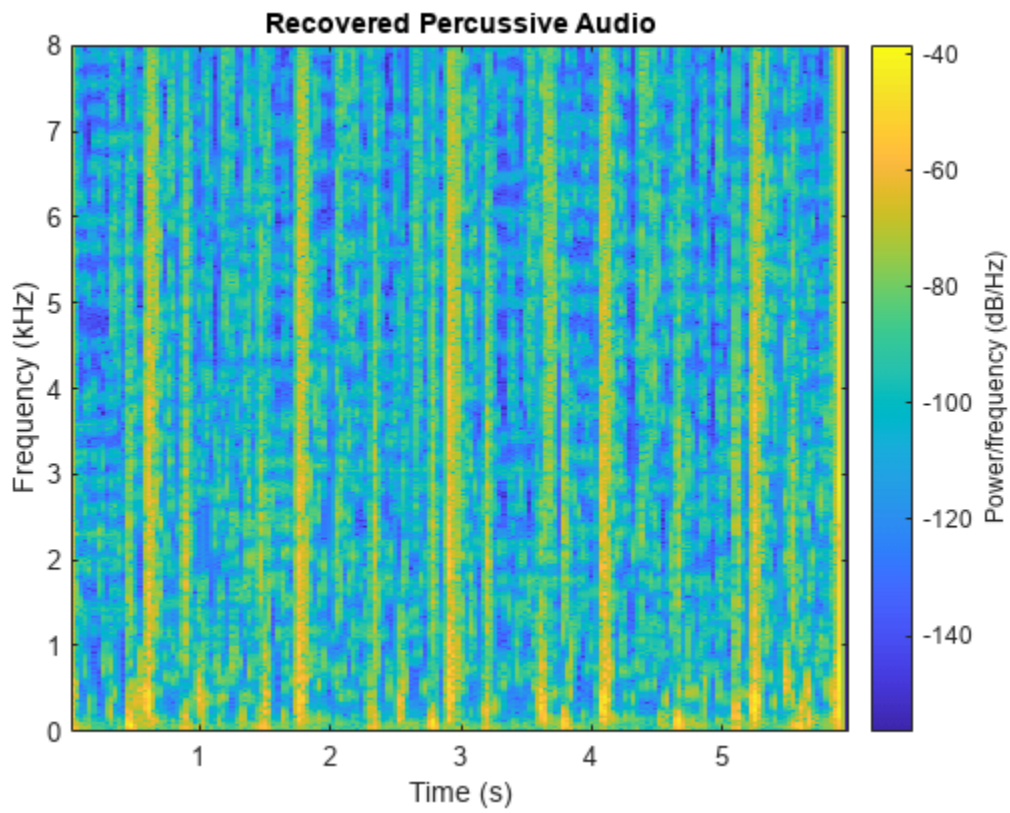
```
spectrogram(h, 1024, 512, 1024, fs, "yaxis")  
title("Recovered Harmonic Audio")
```



Listen to the recovered percussive audio and plot the spectrogram.

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

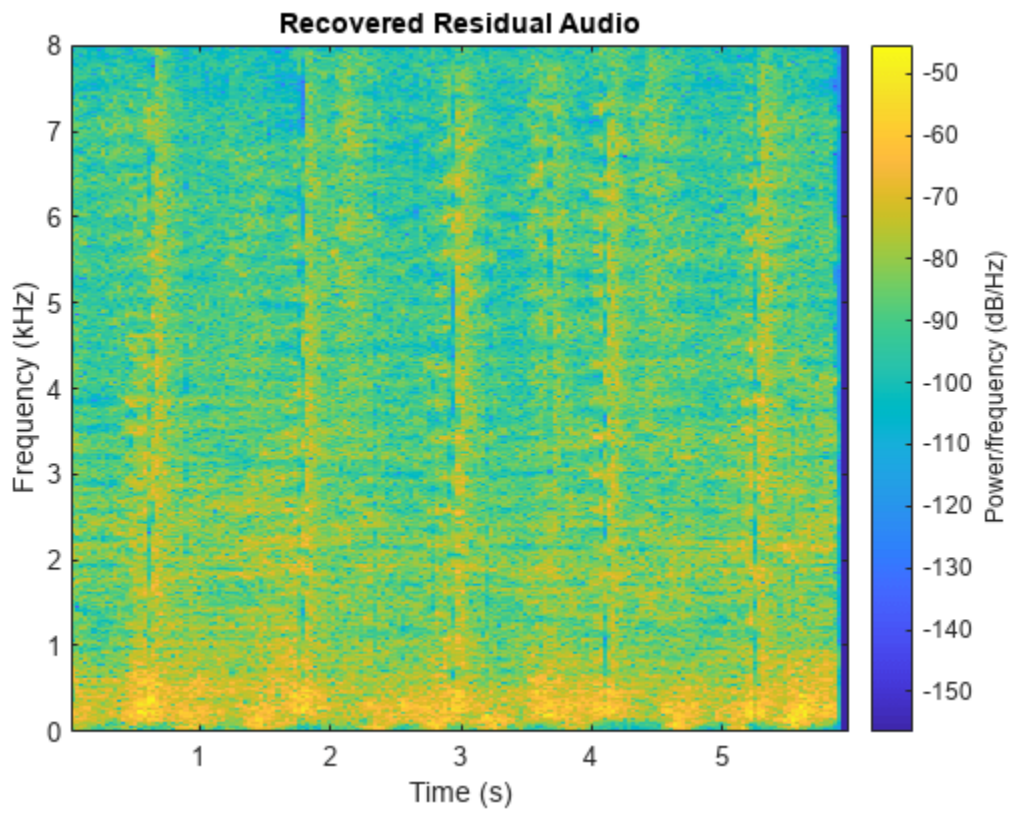
```
spectrogram(p, 1024, 512, 1024, fs, "yaxis")  
title("Recovered Percussive Audio")
```



Listen to the recovered residual audio and plot the spectrogram.

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

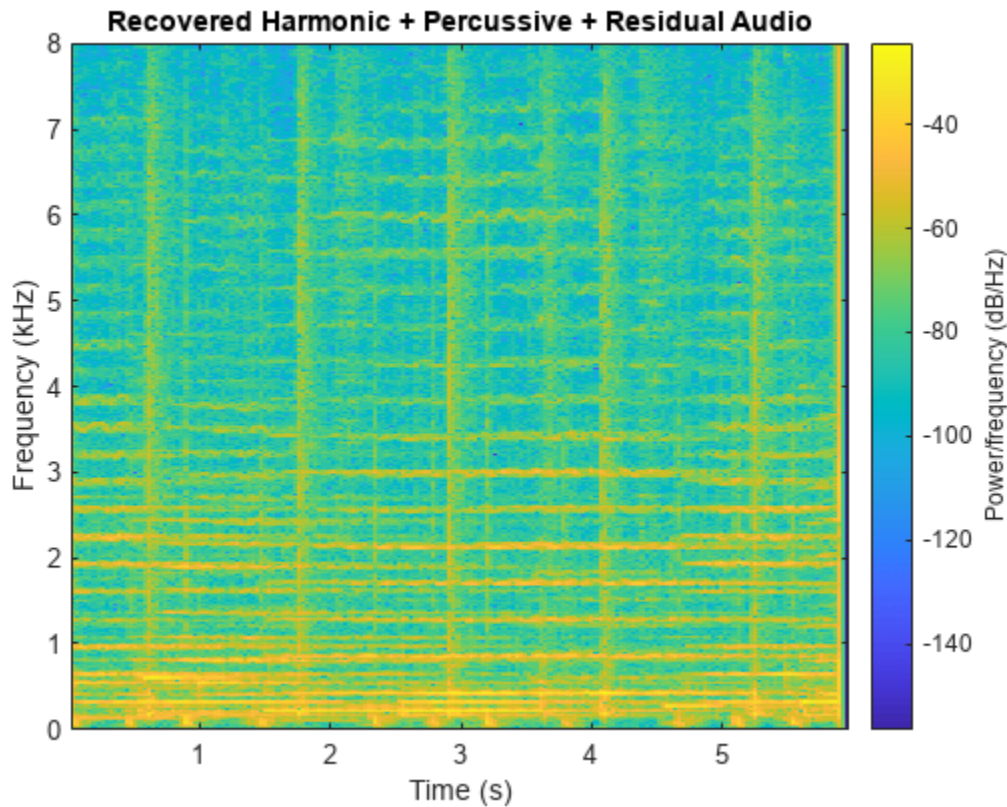
```
spectrogram(r, 1024, 512, 1024, fs, "yaxis")  
title("Recovered Residual Audio")
```



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 1-64] 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 1-64], 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.

See Also

Related Examples

- "Cocktail Party Source Separation Using Deep Learning Networks" on page 1-368

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” (Sensor Fusion and Tracking Toolbox).

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.

```
sigsrc = 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

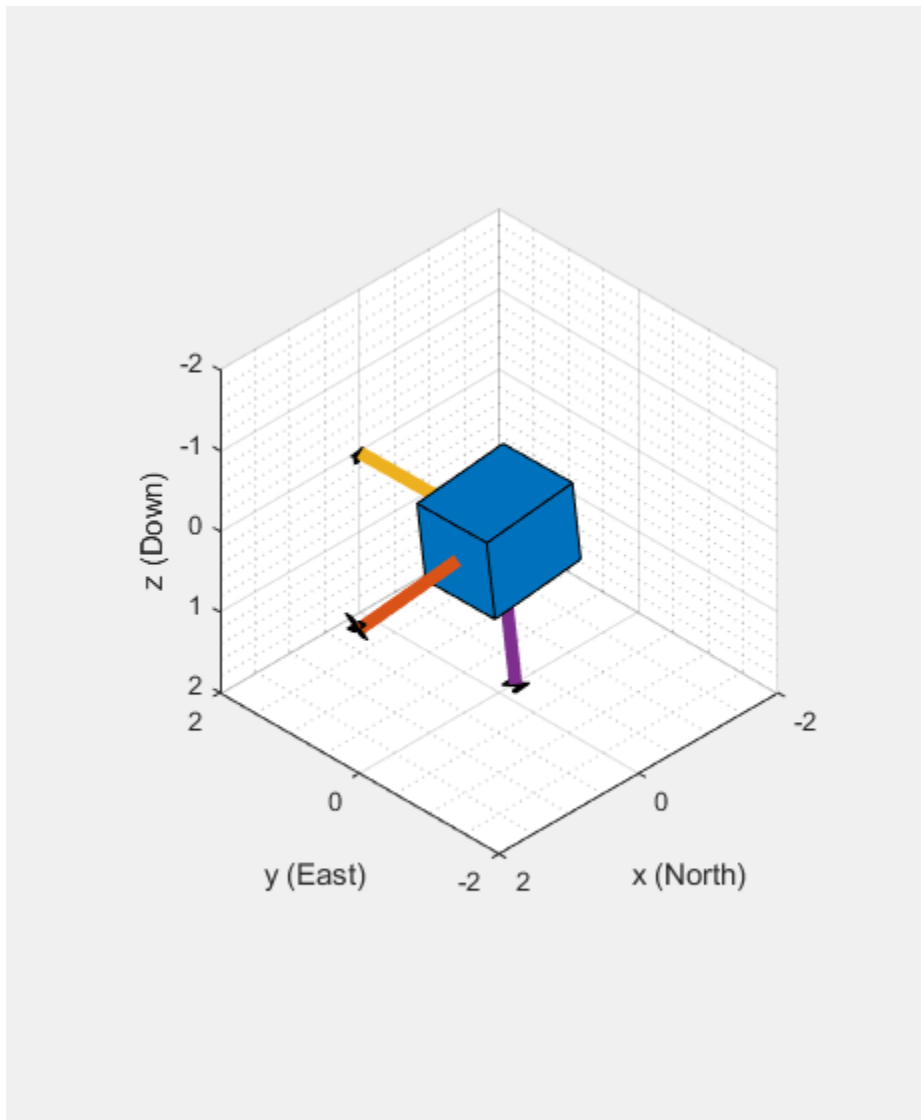
Create a pair of FIR filters to perform binaural HRTF filtering.

```
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.

```
imu0verruns = 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

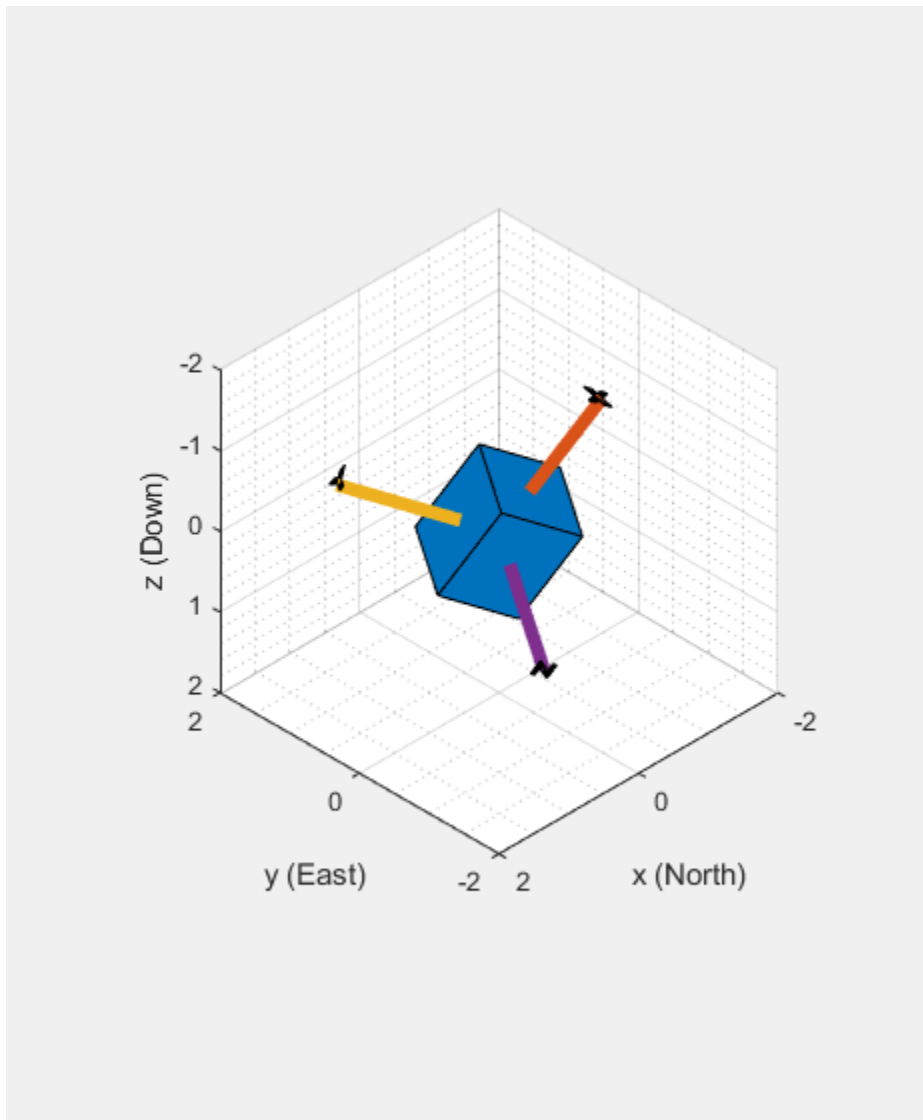
    % Fuse IMU sensor data to estimate the orientation of the sensor.
    qimu = imufilt(data.Acceleration,data.AngularVelocity);
    orientationScope(qimu);

    % 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 1-82.

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 1-545.

Download Data Set

Download the Berlin Database of Emotional Speech [1] on page 1-82. 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.

```
dataFolder = tempdir;
dataset = fullfile(dataFolder,"Emo-DB");
if ~datasetExists(dataset)
    url = "http://emodb.bilderbar.info/download/download.zip";
    disp("Downloading Emo-DB (40.5 MB) ...")
    unzip(url,dataset)
end
```

```
Downloading Emo-DB (40.5 MB) ...
```

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

```
ads = audioDatastore(fullfile(dataset,"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: 535x1 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

Download and 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.

```

downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "SpeechEmotionRecognition...");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
netFolder = fullfile(dataFolder, "SpeechEmotionRecognition");
load(fullfile(netFolder, "network_Audio_SER.mat"));

```

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)

```

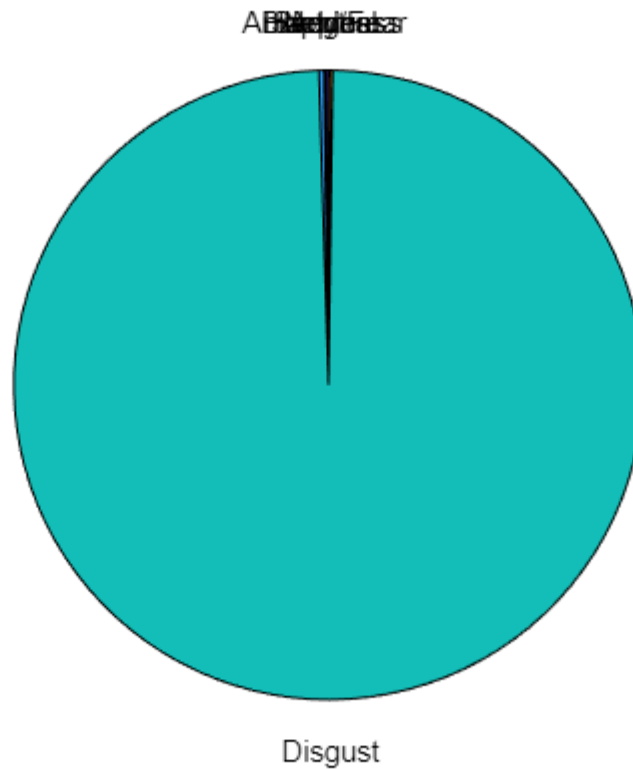
Use the audioFeatureExtractor object to extract the features and then transpose them so that time is along rows. Normalize the features and then convert them to 20-element sequences with 10-

element overlap, which corresponds to approximately 600 ms windows with 300 ms overlap. Use the supporting function, `HelperFeatureVector2Sequence` on page 1-79, 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 and space consuming. However, when iterating on choosing a network architecture or feature extraction pipeline, this upfront cost is generally advantageous.

```
N = numel(ads.Files)*numAugmentations;
```

```
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)
        end
    end
end
disp("Augmentation complete in " + round(toc/60,2) + " minutes.")
```

```
Augmentation complete in 3.84 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: 0% complete
Evaluation 0% complete
```

```
- Pass 1 of 1: Completed in 1 min 7 sec
Evaluation completed in 1 min 7 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,featureVectorsTrain);
```

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.

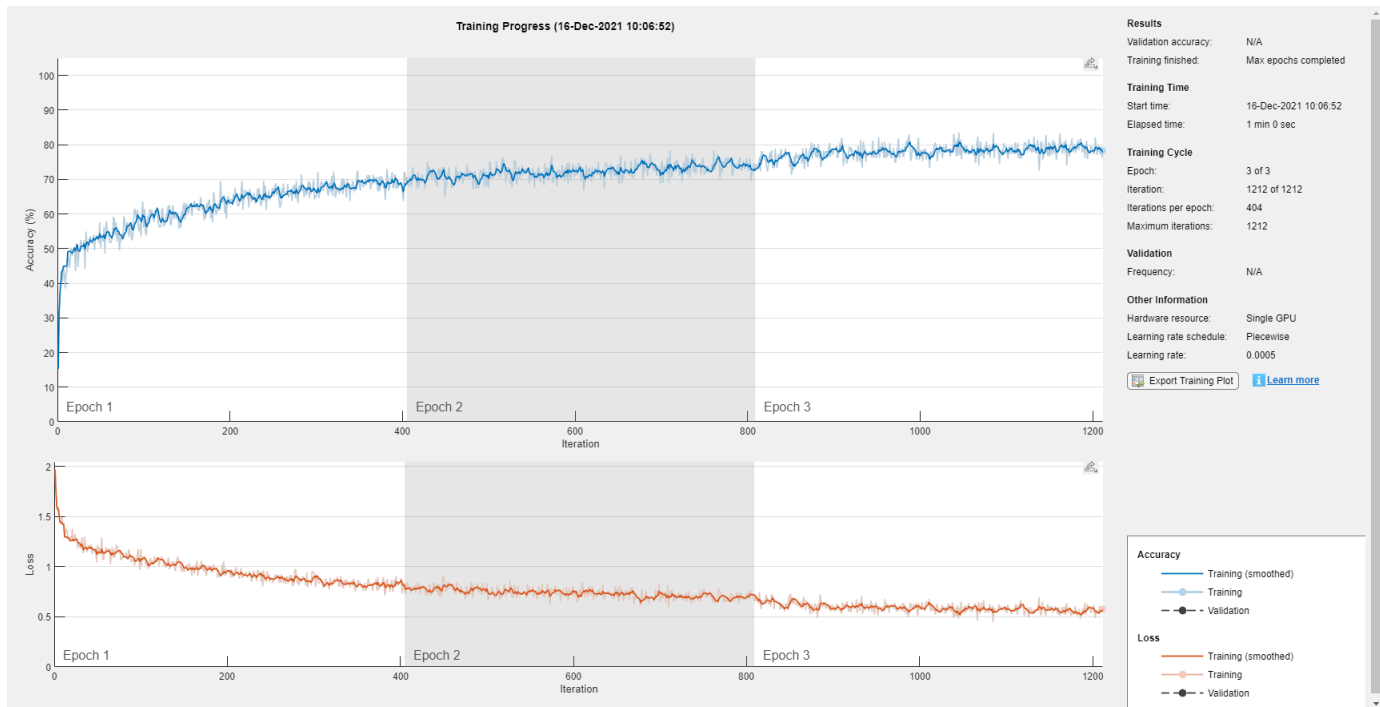
```
dropoutProb1 = 0.3;  
numUnits = 200;  
dropoutProb2 = 0.6;  
layers = [ ...  
    sequenceInputLayer(afe.FeatureVectorLength)  
    dropoutLayer(dropoutProb1)  
    bilstmLayer(numUnits,OutputMode="last")  
    dropoutLayer(dropoutProb2)  
    fullyConnectedLayer(numel(categories(emptyEmotions)))  
    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 =  ;
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 1-80 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;
    disp("Fold " + ii + ", Accuracy = " + round(foldAcc,2))
end
```

```
Fold 1, Accuracy = 65.31
Fold 2, Accuracy = 68.97
Fold 3, Accuracy = 79.07
Fold 4, Accuracy = 71.05
Fold 5, Accuracy = 72.73
Fold 6, Accuracy = 74.29
Fold 7, Accuracy = 67.21
Fold 8, Accuracy = 85.51
Fold 9, Accuracy = 71.43
Fold 10, Accuracy = 67.61
```

```
labelsTrueMat = cat(1,labelsTrue{:});
labelsPredMat = cat(1,labelsPred{:});
```

```
figure
```

```
cm = confusionchart(labelsTrueMat,labelsPredMat, ...
    Title=["Confusion Matrix for 10-Fold Cross-Validation", "Average Accuracy = " + round(mean(labelsTrue{1:10}),2)*100, ...
    ColumnSummary="column-normalized",RowSummary="row-normalized");
sortClasses(cm, categories(emptyEmotions))
```


Confusion Matrix for 10-Fold Cross-Validation

Average Accuracy =

72.3

True Class	Anger	105	1		2	19			82.7%	17.3%
	Anxiety/Fear	8	36	2	3	7	10	3	52.2%	47.8%
	Boredom	1	1	61	1		9	8	75.3%	24.7%
	Disgust	1	1	2	36	3	2	1	78.3%	21.7%
	Happiness	15	7	2	4	42	1		59.2%	40.8%
	Neutral	1	2	18			54	4	68.4%	31.6%
	Sadness		1	7			1	53	85.5%	14.5%
		80.2%	73.5%	66.3%	78.3%	59.2%	70.1%	76.8%		
		19.8%	26.5%	33.7%	21.7%	40.8%	29.9%	23.2%		
		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")
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
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=
    tallFeaturesValidation = cellfun(@(x)extract(extractor,x),tallValidation,"UniformOutput"
    tallFeaturesValidation = cellfun(@(x)x',tallFeaturesValidation,UniformOutput=false); %#ok
    [~,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
    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
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, featureVectorsPerSequence, featureVectorOverlap);
[sequencesValidation, sequencePerFileValidation] = HelperFeatureVector2Sequence(featuresValidation, featureVectorsPerSequence, featureVectorOverlap);

% 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),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.

End-to-End Deep Speech Separation

This example showcases an end-to-end deep learning network for speaker-independent speech separation.

Introduction

Speech separation is a challenging and critical speech processing task. A number of speech separation methods based on deep learning have been proposed recently, most of which rely on time-frequency transformations of the time-domain audio mixture (See “Cocktail Party Source Separation Using Deep Learning Networks” on page 1-368 for an implementation of such a deep learning system).

Solutions based on time-frequency methods suffer from two main drawbacks:

- The conversion of the time-frequency representations back to the time domain requires phase estimation, which introduces errors and leads to imperfect reconstruction.
- Relatively long windows are required to yield high resolution frequency representations, which leads to high computational complexity and unacceptable latency for real-time scenarios.

In this example, you explore a deep learning speech separation network (based on [1]) which acts directly on the audio signal and bypasses the issues arising from time-frequency transformations.

Separate Speech using the Pretrained Network

Download the Pretrained Network

Before training the deep learning network from scratch, you will use a pretrained version of the network to separate two speakers from an example mixture signal.

First, download the pretrained network and example audio files.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "speechSeparation.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
netFolder = fullfile(dataFolder, "speechSeparation");
```

Prepare Test Signal

Load two audio signals corresponding to two different speakers. Both signals are sampled at 8 kHz.

```
Fs = 8000;
s1 = audioread(fullfile(netFolder, "speaker1.wav"));
s2 = audioread(fullfile(netFolder, "speaker2.wav"));
```

Normalize the signals.

```
s1 = s1/max(abs(s1));
s2 = s2/max(abs(s2));
```

Listen to a few seconds of each signal.

```
T = 5;
sound(s1(1:T*Fs))
pause(T)
```

```
sound(s2(1:T*Fs))  
pause(T)
```

Combine the two signals into a mixture signal.

```
mix = s1+s2;  
mix = mix/max(abs(mix));
```

Listen to the first few seconds of the mixture signal.

```
sound(mix(1:T*Fs))  
pause(T)
```

Separate Speakers

Load the parameters of the pretrained speech separation network.

```
load(fullfile(netFolder, "paramsBest.mat"), "learnables", "states")
```

Separate the two speakers in the mixture signals by calling the `separateSpeakers` function.

```
[z1,z2] = separateSpeakers(mix, learnables, states, false);
```

Listen to the first few seconds of the first estimated speech signal.

```
sound(z1(1:T*Fs))  
pause(T)
```

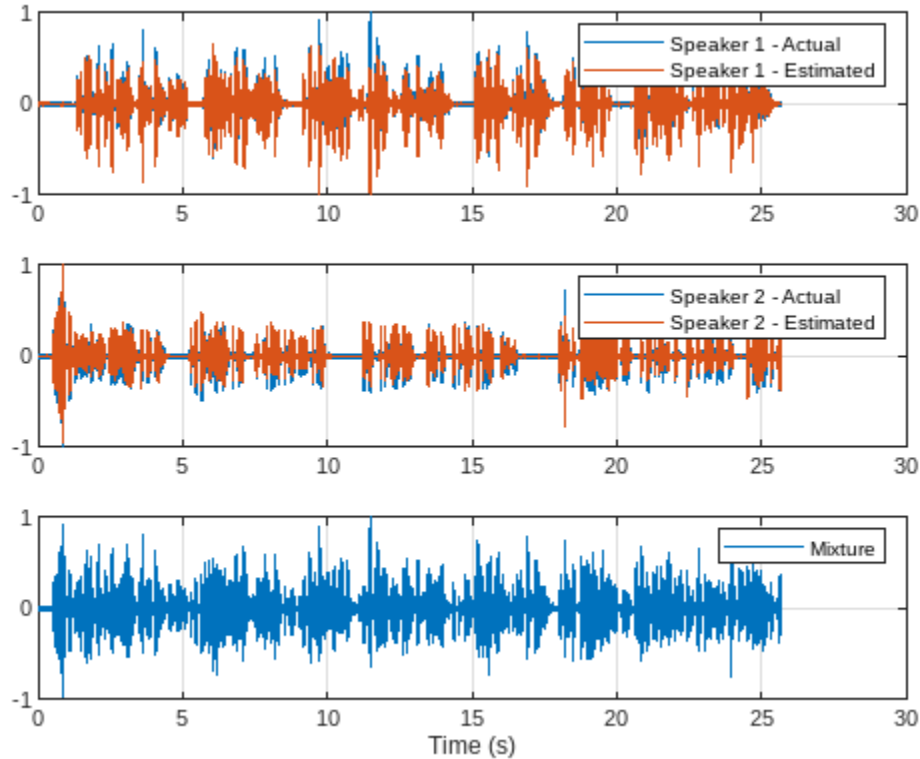
Listen to the second estimated signal.

```
sound(z2(1:T*Fs))  
pause(T)
```

To illustrate the effect of speech separation, plot the estimated and original separated signals along with the mixture signal.

```
s1 = s1(1:length(z1));  
s2 = s2(1:length(z2));  
mix = mix(1:length(s1));  
  
t = (0:length(s1)-1)/Fs;  
  
figure;  
subplot(311)  
plot(t,s1)  
hold on  
plot(t,z1)  
grid on  
legend("Speaker 1 - Actual", "Speaker 1 - Estimated")  
subplot(312)  
plot(t,s2)  
hold on  
plot(t,z2)  
grid on  
legend("Speaker 2 - Actual", "Speaker 2 - Estimated")  
subplot(313)  
plot(t,mix)  
grid on
```

```
legend("Mixture")
xlabel("Time (s)")
```



Compare to a Time-Frequency Transformation Deep Learning Network

Next, you compare the performance of the network to the network developed in the “Cocktail Party Source Separation Using Deep Learning Networks” on page 1-368 example. This speech separation network is based on traditional time-frequency representations of the audio mixture (using the short-time Fourier transform, STFT, and the inverse short-time Fourier transform, ISTFT).

Download the pretrained network.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "CocktailPartySourceSeparation");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
cocktailNetFolder = fullfile(dataFolder, "CocktailPartySourceSeparation");
```

The function `separateSpeakersTimeFrequency` encapsulates the steps required to separate speech using this network. The function performs the following steps:

- Compute the magnitude STFT of the input time-domain mixture.
- Compute a soft time-frequency mask by passing the STFT to the network.
- Compute the STFT of the separated signals by multiplying the mixture STFT by the mask.
- Reconstruct the time-domain separated signals using ISTFT. The phase of the mixture STFT is used.

Refer to the “Cocktail Party Source Separation Using Deep Learning Networks” on page 1-368 example for more details about this network.

Separate the two speakers.

```
[y1,y2] = separateSpeakersTimeFrequency(mix,cocktailNetFolder);
```

Listen to the first separated signal.

```
sound(y1(1:Fs*T))  
pause(T)
```

Listen to the second separated signal.

```
sound(y2(1:Fs*T))  
pause(T)
```

Evaluate Network Performance using SI-SNR

You will compare the two networks using the scale-invariant source-to-noise ratio (SI-SNR) objective measure [1].

Compute the SISNR for the first speaker with the end-to-end network.

First, normalize the actual and estimated signals.

```
s10 = s1 - mean(s1);  
z10 = z1 - mean(z1);
```

Compute the "signal" component of the SNR.

```
t = sum(s10.*z10) .* z10 ./ (sum(z10.^2)+eps);
```

Compute the "noise" component of the SNR.

```
n = s1 - t;
```

Now compute the SI-SNR (in dB).

```
v1 = 20*log((sqrt(sum(t.^2))+eps)./sqrt((sum(n.^2))+eps))/log(10);  
fprintf("End-to-end network - Speaker 1 SISNR: %f dB\n",v1)
```

```
End-to-end network - Speaker 1 SISNR: 14.316869 dB
```

The SI-SNR computation steps are encapsulated in the function `SISNR`. Use the function to compute the SI-SNR of the second speaker with the end-to-end network.

```
v2 = SISNR(z2,s2);  
fprintf("End-to-end network - Speaker 2 SISNR: %f dB\n",v2)
```

```
End-to-end network - Speaker 2 SISNR: 13.706419 dB
```

Next, compute the SI-SNR for each speaker for the STFT-based network.

```
w1 = SISNR(y1,s1(1:length(y1)));  
w2 = SISNR(y2,s2(1:length(y2)));  
fprintf("STFT network - Speaker 1 SISNR: %f dB\n",w1)
```

```
STFT network - Speaker 1 SISNR: 7.003789 dB
```

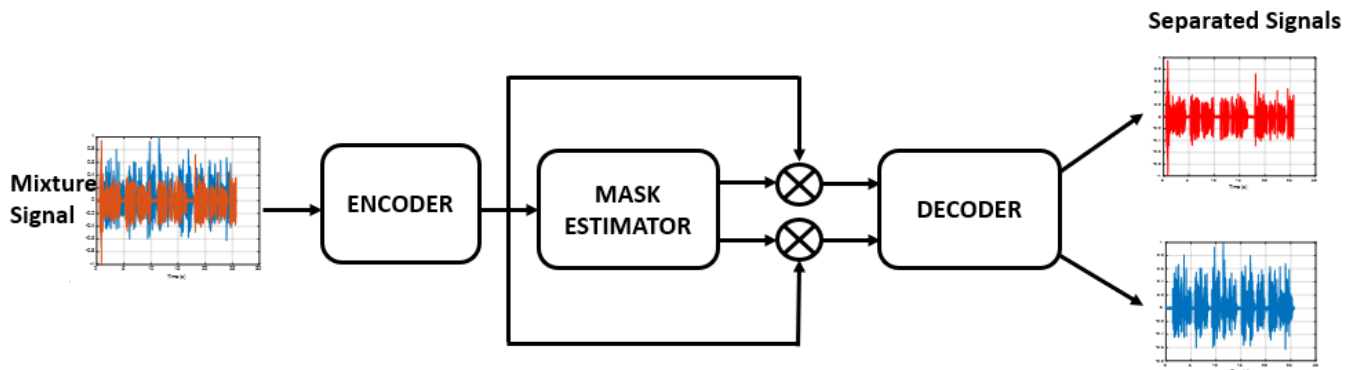


```
fprintf("STFT network - Speaker 2 SISNR: %f dB\n",w2)
```

```
STFT network - Speaker 2 SISNR: 7.382209 dB
```

Training the Speech Separation Network

Examine the Network Architecture



The network is based on [1] and consists of three stages: Encoding, mask estimation or separation, and decoding.

- The encoder transforms the time-domain input mixture signals into an intermediate representation using convolutional layers.
- The mask estimator computes one mask per speaker. The intermediate representation of each speaker is obtained by multiplying the encoder's output by its respective mask. The mask estimator is comprised of 32 blocks of convolutional and normalization layers with skip connections between blocks.
- The decoder transforms the intermediate representations to time-domain separated speech signals using transposed convolutional layers.

The operation of the network is encapsulated in `separateSpeakers`.

Optionally Reduce the Dataset Size

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`. This will run the rest of the example on only a handful of files.

```
reduceDataset = true;
```

Download the Training Dataset

You use a subset of the LibriSpeech Dataset [2] to train the network. The LibriSpeech Dataset is a large corpus of read English speech sampled at 16 kHz. The data is derived from audiobooks read from the LibriVox project.

Download the LibriSpeech dataset. If `reduceDataset` is `true`, this step is skipped.

```
downloadDatasetFolder = tempdir;  
datasetFolder = fullfile(downloadDatasetFolder, "LibriSpeech", "train-clean-360");
```

```
if ~reduceDataset
    filename = "train-clean-360.tar.gz";
    url = "http://www.openslr.org/resources/12/" + filename;
    if ~datasetExists(datasetFolder)
        gunzip(url,downloadDatasetFolder);
        unzippedFile = fullfile(downloadDatasetFolder,filename);
        untar(unzippedFile{1}(1:end-3),downloadDatasetFolder);
    end
end
```

Preprocess the Dataset

The LibriSpeech dataset is comprised of a large number of audio files with a single speaker. It does not contain mixture signals where 2 or more persons are speaking simultaneously.

You will process the original dataset to create a new dataset that is suitable for training the speech separation network.

The steps for creating the training dataset are encapsulated in `createTrainingDataset`. The function creates mixture signals comprised of utterances of two random speakers. The function returns three audio datastores:

- `mixDatastore` points to mixture files (where two speakers are talking simultaneously).
- `speaker1Datastore` points to files containing the isolated speech of the first speaker in the mixture.
- `speaker2Datastore` points to files containing the isolated speech of the second speaker in the mixture.

Define the mini-batch size and the maximum training signal length (in number of samples).

```
miniBatchSize = 2;
duration = 2*8000;
```

Create the training dataset.

```
[mixDatastore,speaker1Datastore,speaker2Datastore] = createTrainingDataset(netFolder,datasetFolder)
```

Combine the datastores. This ensures that the files stay in the correct order when you shuffle them at the start of each new epoch in the training loop.

```
ds = combine(mixDatastore,speaker1Datastore,speaker2Datastore);
```

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™.

```
executionEnvironment = "auto"; % Change to "cpu" to train on a CPU.
```

Create a minibatch queue from the datastore.

```
mqueue = minibatchqueue(ds,MiniBatchSize=miniBatchSize, OutputEnvironment=executionEnvironment,Options)
```

Specify Training Options

Define training parameters.

Train for 10 epochs.

```

if reduceDataset
    numEpochs = 1;
else
    numEpochs = 10; %#ok
end

```

Specify the options for Adam optimization. Set the initial learning rate to 1e-3. Use a gradient decay factor of 0.9 and a squared gradient decay factor of 0.999.

```

learnRate = 1e-3;
averageGrad = [];
averageSqGrad = [];

gradDecay = 0.9;
sqGradDecay = 0.999;

```

Set Up Validation Data

You will use the test signal you previously employed to test the pretrained network to compute a validation SI-SNR periodically during training.

If a GPU is available, move the validation signal to the GPU.

```

mix = dldarray(mix, 'SCB');
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    mix = gpuArray(mix);
end

```

Define the number of iterations between validation SI-SNR computations.

```
numIterPerValidation = 50;
```

Define a vector to hold the validation SI-SNR from each iteration.

```
valSNR = [];
```

Define a variable to hold the best validation SI-SNR.

```
bestSNR = -Inf;
```

Define a variable to hold the epoch in which the best validation score occurred.

```
bestEpoch = 1;
```

Initialize Network

Initialize the network parameters. `learnables` is a structure containing the learnable parameters from the network layers. `states` is a structure containing the states from the normalization layers.

```
[learnables,states] = initializeNetworkParams;
```

Train the Network

Execute the training loop. This can take many hours to run.

Note that there is no a priori way to associate the estimated output speaker signals with the expected speaker signals. This is resolved by using Utterance-level permutation invariant training (uPIT) [1]. The loss is based on computing the SI-SNR. uPIT minimizes the loss over all permutations between outputs and targets. It is defined in the function `uPIT`.

The validation SI-SNR is computed periodically. If the SI-SNR is the best value to-date, the network parameters are saved to `params.mat`.

```
iteration = 0;

% Loop over epochs.
for jj =1:numEpochs

    % Shuffle the data
    shuffle(mqueue);

    while hasdata(mqueue)

        % Compute validation loss/SNR periodically
        if mod(iteration,numIterPerValidation)==0

            [z1,z2] = separateSpeakers(mix, learnables,states,false);

            l = uPIT(z1,s1,z2,s2);
            valSNR(end+1) = l; %#ok

            if l > bestSNR
                bestSNR = l;
                bestEpoch = jj;
                filename = "params.mat";
                save(filename,"learnables","states");
            end
        end

        iteration = iteration + 1;

        % Get a new batch of training data
        [mixBatch,x1Batch,x2Batch] = next(mqueue);

        % Evaluate the model gradients and states using dlfeval and the modelLoss function.
        [~,gradients,states] = dlfeval(@modelLoss,mixBatch,x1Batch,x2Batch,learnables,states,minibatchSize);

        % Update the network parameters using the ADAM optimizer.
        [learnables,averageGrad,averageSqGrad] = adamupdate(learnables,gradients,averageGrad,averageSqGrad,learnables);

    end

    % Reduce the learning rate if the validation accuracy did not improve
    % during the epoch
    if bestEpoch ~= jj
        learnRate = learnRate/2;
    end
end

Plot the validation SNR values.

if ~reduceDataset
    valIterNum = 0:length(valSNR)-1;
    figure
    semilogx(numIterPerValidation*(valIterNum-1),valSNR,"b*-")
    grid on
    xlabel("Iteration #")
    ylabel("Validation SINR (dB)")
end
```

```
    valFig.Visible = 'on';
end
```

References

[1] Yi Luo, Nima Mesgarani, "Conv-tasnet: Surpassing ideal time-frequency magnitude masking for speech separation," 2019 IEEE/ACM transactions on audio, speech, and language processing, vol. 29, issue 8, pp. 1256-1266.

[2] V. Panayotov, G. Chen, D. Povey and S. Khudanpur, "Librispeech: An ASR corpus based on public domain audio books," 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Brisbane, QLD, 2015, pp. 5206-5210, doi: 10.1109/ICASSP.2015.7178964

Supporting Functions

```
function [mixDatastore, speaker1Datastore, speaker2Datastore] = createTrainingDataset(netFolder, datasetFolder)
% createTrainingDataset Create training dataset

newDatasetPath = fullfile(downloadDatasetFolder, "speech-sep-dataset");

% Create the new dataset folders.
if isfolder(newDatasetPath)
    rmdir(newDatasetPath, "s")
end
mkdir(newDatasetPath);
mkdir(fullfile(newDatasetPath, "sp1"));
mkdir(fullfile(newDatasetPath, "sp2"));
mkdir(fullfile(newDatasetPath, "mix"));

%Create an audioDatastore that points to the LibriSpeech dataset.
if reduceDataset
    netFolder = char(netFolder);
    ads = audioDatastore([repmat(fullfile(netFolder, "speaker1.wav"), 1, 4), ...
        repmat(fullfile(netFolder, "speaker2.wav"), 1, 4)]);
else
    ads = audioDatastore(datasetFolder, IncludeSubfolders=true);
end

% The LibriSpeech dataset is comprised of signals from different speakers.
% The unique speaker ID is encoded in the audio file names.

% Extract the speaker IDs from the file names.
if reduceDataset
    ads.Labels = categorical([repmat({'1'}, 1, 4), repmat({'2'}, 1, 4)]);
else
    ads.Labels = categorical(extractBetween(ads.Files, fullfile(datasetFolder, filesep), filesep));
end

% You will create mixture signals comprised of utterances of two random speakers.
% Randomize the IDs of all the speakers.
names = unique(ads.Labels);
names = names(randperm(length(names)));

% In this example, you create training signals based on 400 speakers. You
% generate mixture signals based on combining utterances from 200 pairs of
% speakers.

% Define the two groups of speakers.
```

```

numPairs = min(200,floor(numel(names)/2));
n1 = names(1:numPairs);
n2 = names(numPairs+1:2*numPairs);

% Create the new dataset. For each pair of speakers:
% * Use subset to create two audio datastores, each containing files
%   corresponding to their respective speaker.
% * Adjust the datastores so that they have the same number of files.
% * Combine the two datastores using combine.
% * Use writeall to preprocess the files of the combined datastore and write
%   the new resulting signals to disk.

% The preprocessing steps performed to create the signals before writing
% them to disk are encapsulated in the function createTrainingFiles. For
% each pair of signals:
% * You downsample the signals from 16 kHz to 8 kHz.
% * You randomly select 4 seconds from each downsampled signal.
% * You create the mixture by adding the 2 signal chunks.
% * You adjust the signal power to achieve a randomly selected
%   signal-to-noise value in the range [-5,5] dB.
% * You write the 3 signals (corresponding to the first speaker, the second
%   speaker, and the mixture, respectively) to disk.
parfor index=1:length(n1)
    spkInd1 = n1(index);
    spkInd2 = n2(index);
    spk1ds = subset(ads,ads.Labels==spkInd1);
    spk2ds = subset(ads,ads.Labels==spkInd2);
    L = min(length(spk1ds.Files),length(spk2ds.Files));
    L = floor(L/miniBatchSize) * miniBatchSize;
    spk1ds = subset(spk1ds,1:L);
    spk2ds = subset(spk2ds,1:L);
    pairs = combine(spk1ds,spk2ds);
    writeall(pairs,newDatasetPath,FolderLayout="flatten",WriteFcn=@(data,writeInfo,outputFmt)cro
end

% Create audio datastores pointing to the files corresponding to the individual speakers and the
mixDatastore = audioDatastore(fullfile(newDatasetPath,"mix"));
speaker1Datastore = audioDatastore(fullfile(newDatasetPath,"sp1"));
speaker2Datastore = audioDatastore(fullfile(newDatasetPath,"sp2"));
end

function mix = createTrainingFiles(data,writeInfo,~,varargin)
% createTrainingFiles - Preprocess the training signals and write them to disk

reduceDataset = varargin{1};
duration = varargin{2};

x1 = data{1};
x2 = data{2};

% Resample from 16 kHz to 8 kHz
if ~reduceDataset
    x1 = resample(x1,1,2);
    x2 = resample(x2,1,2);
end

% Read a chunk from the first speaker signal
x1 = readSpeakerSignalChunk(duration,x1);

```

```

% Read a chunk from the second speaker signal
x2 = readSpeakerSignalChunk(duration,x2);

% SNR [-5 5] dB
s = snr(x1,x2);
targetSNR = 10 * (rand - 0.5);
x1b = 10^((targetSNR-s)/20) * x1;
mix = x1b + x2;
mix = mix./max(abs(mix));

if reduceDataset
    [~,n] = fileparts(tempname);
    name = sprintf("%s.wav",n);
else
    [~,s1] = fileparts(writeInfo.ReadInfo{1}.FileName);
    [~,s2] = fileparts(writeInfo.ReadInfo{2}.FileName);
    name = sprintf("%s-%s.wav",s1,s2);
end

audiowrite(sprintf("%s",fullfile(writeInfo.Location,"sp1",name)),x1,8000);
audiowrite(sprintf("%s",fullfile(writeInfo.Location,"sp2",name)),x2,8000);
audiowrite(sprintf("%s",fullfile(writeInfo.Location,"mix",name)),mix,8000);

end

function sequence = readSpeakerSignalChunk(duration,sequence)
% readSpeakerSignalChunk - Read a chunk from the speaker signal
if length(sequence)<=duration
    sequence = [sequence;zeros(duration-length(sequence),1)];
else
    startInd = randi([1 length(sequence)-duration],1);
    endInd = startInd + duration - 1;
    sequence = sequence(startInd:endInd);
end
sequence = sequence./max(abs(sequence));
end

function [loss,gradients,states] = modelLoss(mix,x1,x2,learnables,states,miniBatchSize)
% modelLoss Compute the model loss, gradients, and states

[y1,y2,states] = separateSpeakers(mix,learnables,states,true);

m = uPIT(x1,y1,x2,y2);
l = sum(m);
loss = -l./miniBatchSize;

gradients = dlgradient(loss,learnables);

end

function m = uPIT(x1,y1,x2,y2)
% uPIT - Compute utterance-level permutation invariant training
v1 = SISNR(y1,x1);
v2 = SISNR(y2,x2);
m1 = mean([v1;v2]);

v1 = SISNR(y2,x1);

```

```
v2 = SISNR(y1,x2);
m2 = mean([v1;v2]);

m = max(m1,m2);
end

function z = SISNR(x,y)
% SISNR - Compute SI-SNR
x = x - mean(x);
y = y - mean(y);

t = sum(x.*y) .* y ./ (sum(y.^2)+eps);

z = 20*log((sqrt(sum(t.^2))+eps)./sqrt((sum((x-t).^2)+eps))/log(10));

end

function [learnables,states] = initializeNetworkParams
% initializeNetworkParams - Initialize the learnables and states of the
% network
learnables.Conv1W = initializeGlorot(20,1,256);
learnables.Conv1B = dlarray(zeros(256,1,"single"));

learnables.ln_weight = dlarray(ones(1,256,"single"));
learnables.ln_bias = dlarray(zeros(1,256,"single"));

learnables.Conv2W = initializeGlorot(1,256,256);
learnables.Conv2B = dlarray(zeros(256,1,"single"));

blk.Conv1B = dlarray(zeros(512,1,"single"));
blk.Prelu1 = dlarray(single(0.25));
blk.BN1offset = dlarray(zeros(512,1,"single"));
blk.BN1Scale = dlarray(ones(512,1,"single"));
blk.Conv2B = dlarray(zeros(512,1,"single"));
blk.Prelu2 = dlarray(single(0.25));
blk.BN2offset= dlarray(zeros(512,1,"single"));
blk.BN2Scale= dlarray(ones(512,1,"single"));
blk.Conv3B = dlarray(ones(256,1,"single"));

s.BN1Mean= dlarray(zeros(512,1,"single"));
s.BN1Var= dlarray(ones(512,1,"single"));
s.BN2Mean = dlarray(zeros(512,1,"single"));
s.BN2Var = dlarray(ones(512,1,"single"));

for index=1:32
    blk.Conv1W = initializeGlorot(1,256,512);
    blk.Conv2W = initializeGlorot(3,1,512);
    blk.Conv2W = reshape(blk.Conv2W,[3 1 1 512]);
    blk.Conv3W = initializeGlorot(1,512,256);
    learnables.Blocks(index) = blk;
    states(index) = s; %#ok
end

learnables.Conv3W = initializeGlorot(1,256,512);
learnables.Conv3B = dlarray(zeros(512,1,"single"));

learnables.TransConv1W = initializeGlorot(20,1,256);
learnables.TransConv1B = dlarray(zeros(1,1, "single"));
```



```

end

function weights = initializeGlorot(filterSize,numChannels,numFilters)
% initializeGlorot - Perform Glorot initialization
sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = numOut;

Z = 2*rand(sz,"single") - 1;
bound = sqrt(6 / (numIn + numOut));

weights = dlarray(bound * Z);
end

function [output1, output2, states] = separateSpeakers(input, learnables, states, training)
% separateSpeakers - Separate two speaker signals from a mixture input
if ~isdllarray(input)
    input = dlarray(input,"SCB");
end

x = dlconv(input, learnables.Conv1W,learnables.Conv1B, Stride= 10);

x = relu(x);
x0 = x;

x = x-mean(x, 2);
x = x./sqrt(mean(x.^2, 2) + 1e-5);
x = x.*learnables.ln_weight + learnables.ln_bias;

encoderOut = dlconv(x, learnables.Conv2W, learnables.Conv2B);

for index = 1:32
    [encoderOut,s] = convBlock(encoderOut, index-1,learnables.Blocks(index),states(index),training);
    states(index) = s;
end

masks = dlconv(encoderOut, learnables.Conv3W, learnables.Conv3B);
masks = relu(masks);

mask1 = masks(:,1:256,:);
mask2 = masks(:,257:512,:);

out1 = x0 .* mask1;
out2 = x0 .* mask2;

weights = learnables.TransConv1W;
bias = learnables.TransConv1B;
output2 = dltranspconv(out1, weights, bias, Stride=10);
output1 = dltranspconv(out2, weights, bias, Stride=10);

if ~training
    output1 = gather(extractdata(output1));
    output2 = gather(extractdata(output2));

    output1 = output1./max(abs(output1));
    output2 = output2./max(abs(output2));
end

```

```
end

function [output,state] = convBlock(input, count,learnables,state,training)

% Conv:
conv1Out = dlconv(input, learnables.Conv1W, learnables.Conv1B);

% PRelu:
conv1Out = relu(conv1Out) - learnables.Prelu1.*relu(-conv1Out);

% BatchNormalization:
offset = learnables.BN1Offset;
scale = learnables.BN1Scale;
datasetMean = state.BN1Mean;
datasetVariance = state.BN1Var;
if training
    [batchOut, dsmean, dsvar] = batchnorm(conv1Out, offset, scale, datasetMean, datasetVariance);
    state.BN1Mean = dsmean;
    state.BN1Var = dsvar;
else
    batchOut = batchnorm(conv1Out, offset, scale, datasetMean, datasetVariance);
end

% Conv:
padding = [1 1] * 2^(mod(count,8));
dilationFactor = 2^(mod(count,8));
conv0Out = dlconv(batchOut, learnables.Conv2W, learnables.Conv2B,DilationFactor=dilationFactor, Pa

% PRelu:
conv0Out = relu(conv0Out) - learnables.Prelu2.*relu(-conv0Out);

% BatchNormalization:
if training
    [batchOut, dsmean, dsvar] = batchnorm(conv0Out, learnables.BN2Offset, learnables.BN2Scale, sta
    state.BN2Mean = dsmean;
    state.BN2Var = dsvar;
else
    batchOut = batchnorm(conv0Out, learnables.BN2Offset, learnables.BN2Scale, state.BN2Mean, sta
end

% Conv:
output = dlconv(batchOut, learnables.Conv3W, learnables.Conv3B);

% Skip connection
output = output + input;

end

function [speaker1,speaker2] = separateSpeakersTimeFrequency(mix,pathToNet)
% separateSpeakersTimeFrequency - STFT-based speaker separation function
WindowLength = 128;
FFTLenght = 128;
OverlapLength = 128-1;
win = hann(WindowLength,"periodic");

% Downsample to 4 kHz
mix = resample(mix,1,2);
```

```

P0 = stft(mix, Window=win, OverlapLength=OverlapLength,...
    FFTLength=FFTLength, FrequencyRange="onesided");
P = log(abs(P0) + eps);
MP = mean(P(:));
SP = std(P(:));
P = (P-MP)/SP;

seqLen = 20;
PSeq = zeros(1 + FFTLength/2, seqLen, 1, 0);
seqOverlap = seqLen;

loc = 1;
while loc < size(P,2)-seqLen
    PSeq(:,:,,end+1) = P(:,loc:loc+seqLen-1); %#ok
    loc = loc + seqOverlap;
end

PSeq = reshape(PSeq, [1 1 (1 + FFTLength/2) * seqLen size(PSeq,4)]);

s = load(fullfile(pathToNet, "CocktailPartyNet.mat"));
CocktailPartyNet = s.CocktailPartyNet;
estimatedMasks = predict(CocktailPartyNet, PSeq);

estimatedMasks = estimatedMasks.';
estimatedMasks = reshape(estimatedMasks, 1 + FFTLength/2, numel(estimatedMasks)/(1 + FFTLength/2))

mask1 = estimatedMasks;
mask2 = 1 - mask1;

P0 = P0(:, 1:size(mask1,2));

P_speaker1 = P0 .* mask1;

speaker1 = istft(P_speaker1, Window=win, OverlapLength=OverlapLength,...
    FFTLength=FFTLength, ConjugateSymmetric=true,...
    FrequencyRange="onesided");
speaker1 = speaker1 / max(abs(speaker1));

P_speaker2 = P0 .* mask2;

speaker2 = istft(P_speaker2, Window=win, OverlapLength=OverlapLength,...
    FFTLength=FFTLength, ConjugateSymmetric=true,...
    FrequencyRange="onesided");
speaker2 = speaker2 / max(speaker2);

speaker1 = resample(double(speaker1), 2, 1);
speaker2 = resample(double(speaker2), 2, 1);
end

function [x1Batch, x2Batch, mixBatch] = preprocessMiniBatch(x1Batch, x2Batch, mixBatch)
% preprocessMiniBatch - Preprocess mini-batch
x1Batch = cat(3, x1Batch{:});
x2Batch = cat(3, x2Batch{:});

```

```
mixBatch = cat(3,mixBatch{:});  
end
```

See Also

Related Examples

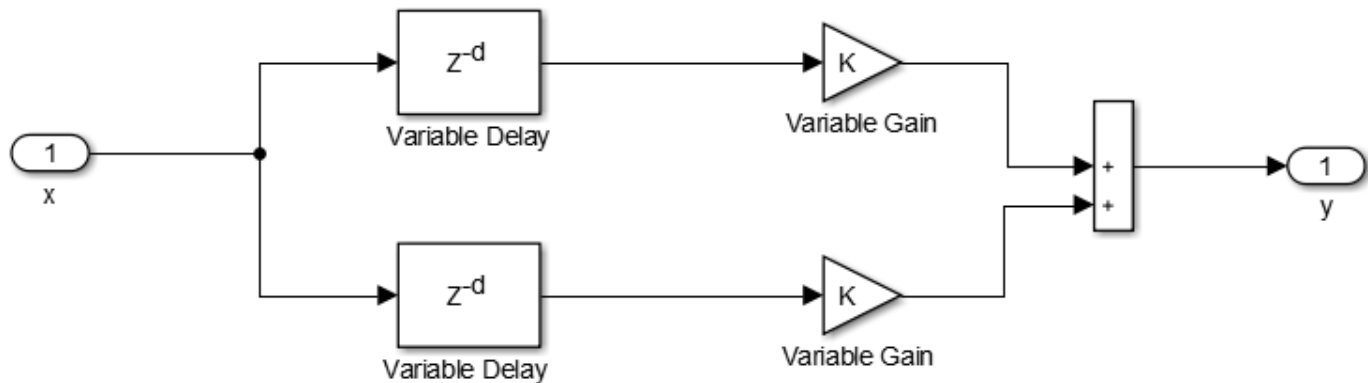
- “Cocktail Party Source Separation Using Deep Learning Networks” on page 1-368

Delay-Based Pitch Shifter

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

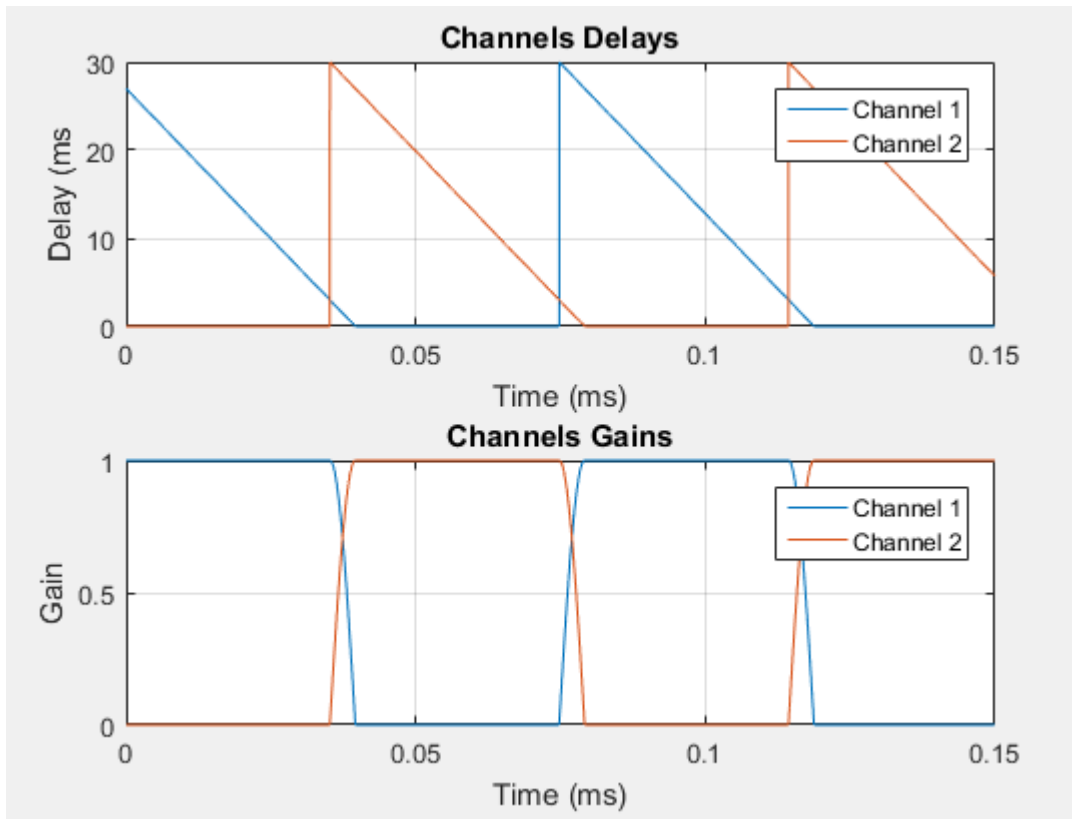
Algorithm

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 `spectrumAnalyzer` to view and compare the input and output signals in the time and frequency domains, respectively.

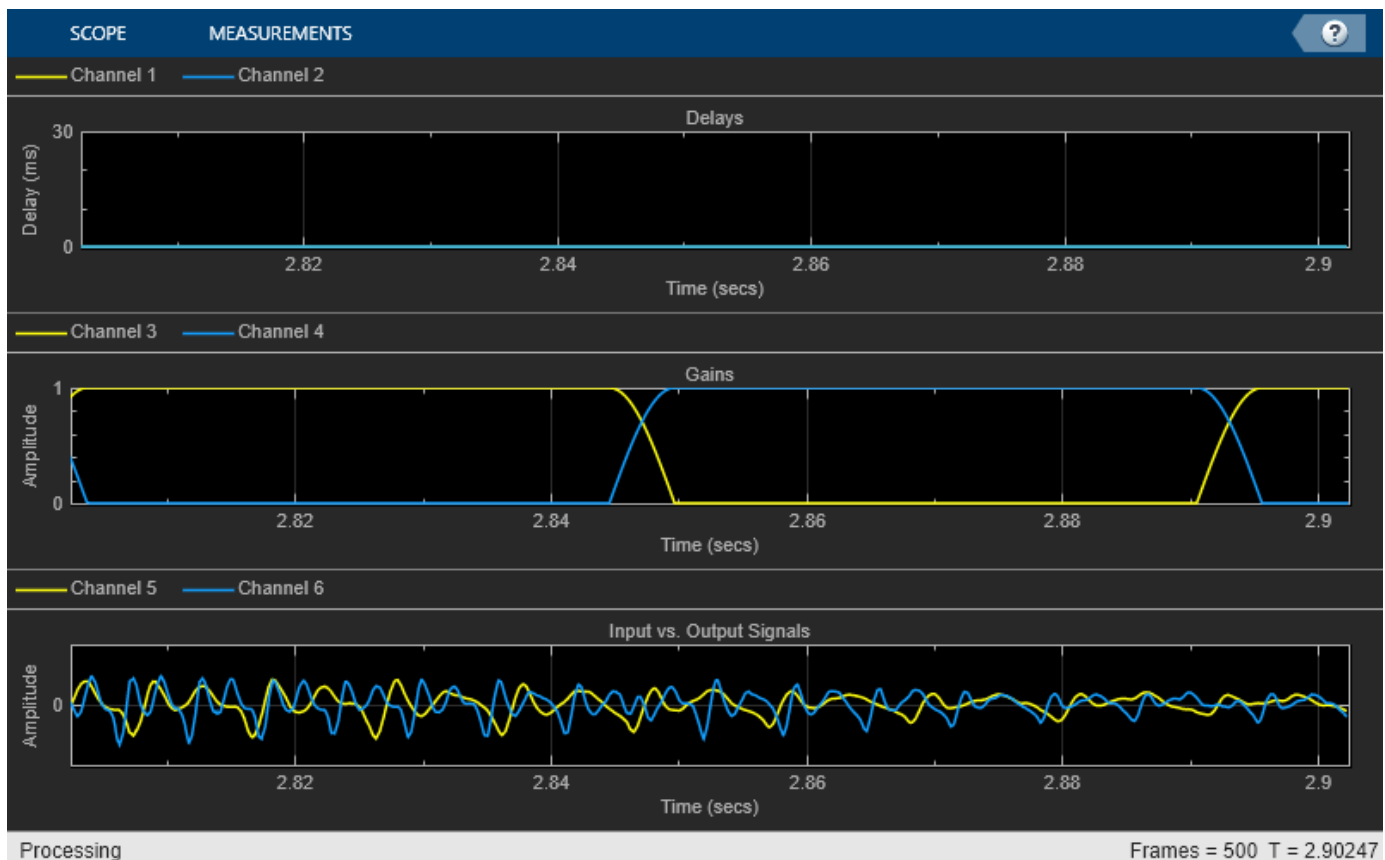
You can also use `audiopluginexample.PitchShifter` in MATLAB just as you would use any other MATLAB object. You can use the `configureMIDI` command to enable tuning the object via a MIDI device. This is particularly useful if the object is part of a streaming MATLAB simulation where the command window 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

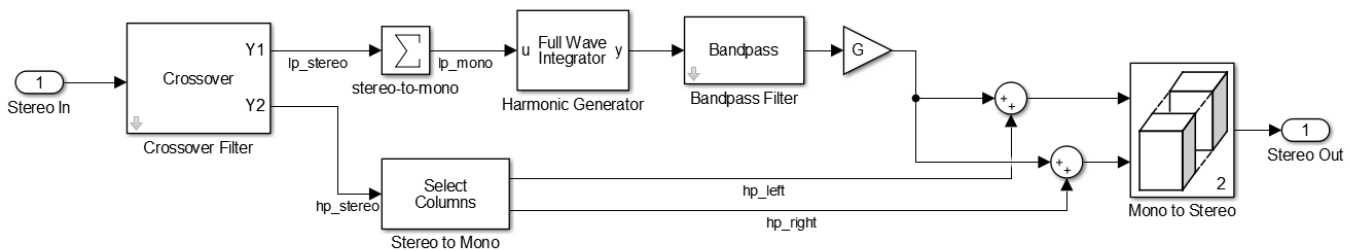
Small loudspeakers typically have a poor low frequency response, which can have a negative impact on overall sound quality. This example implements psychoacoustic bass enhancement to improve sound quality of audio played on small loudspeakers.

The example is based on the algorithm in [1 on page 1-105]. 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 1-105].



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.

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

- $u[n]$ 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 filter'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.

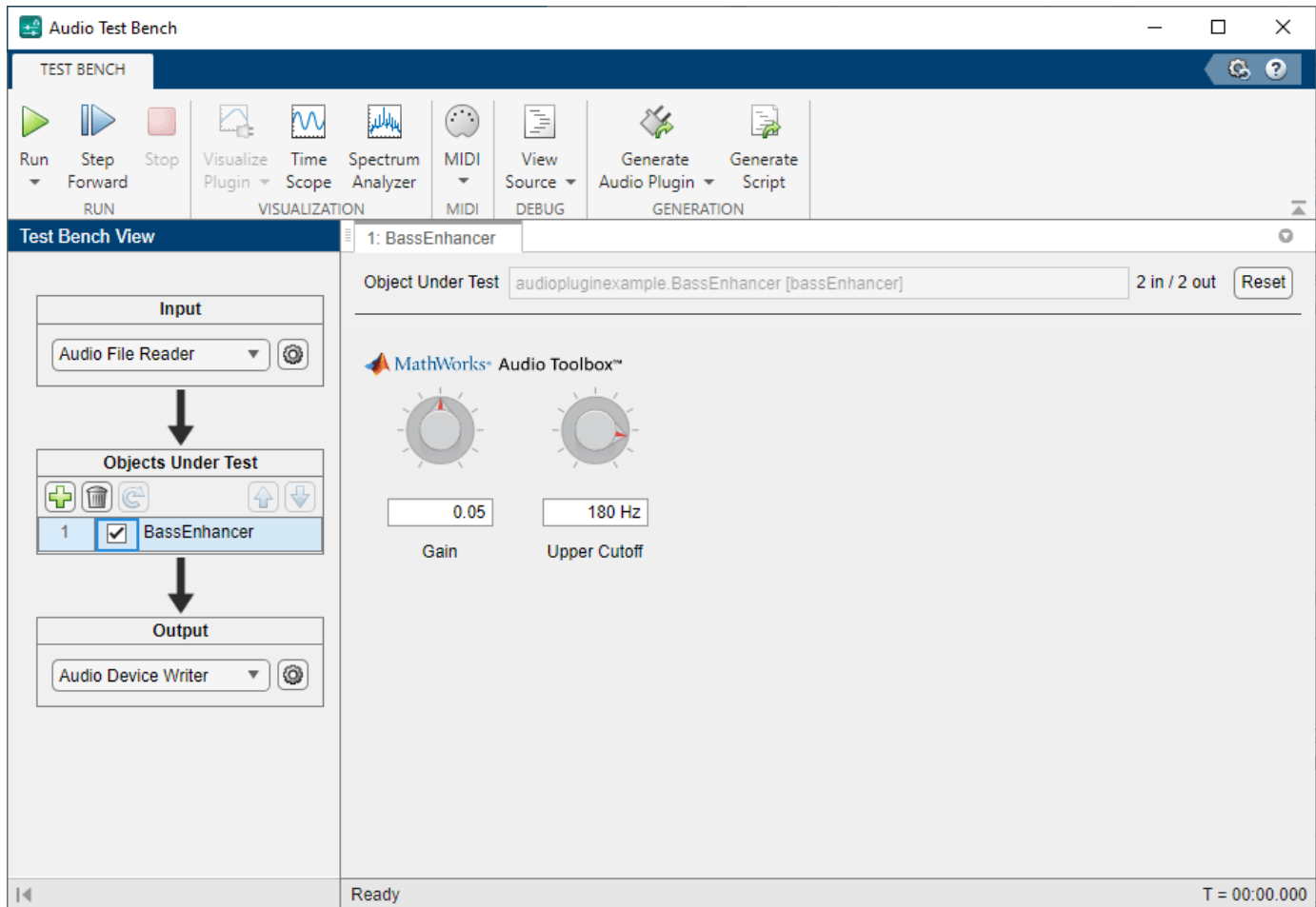
Although the resulting output stereo signal does not contain low-frequency elements, the input's bass pitch is preserved thanks to the generated harmonics.

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 `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.

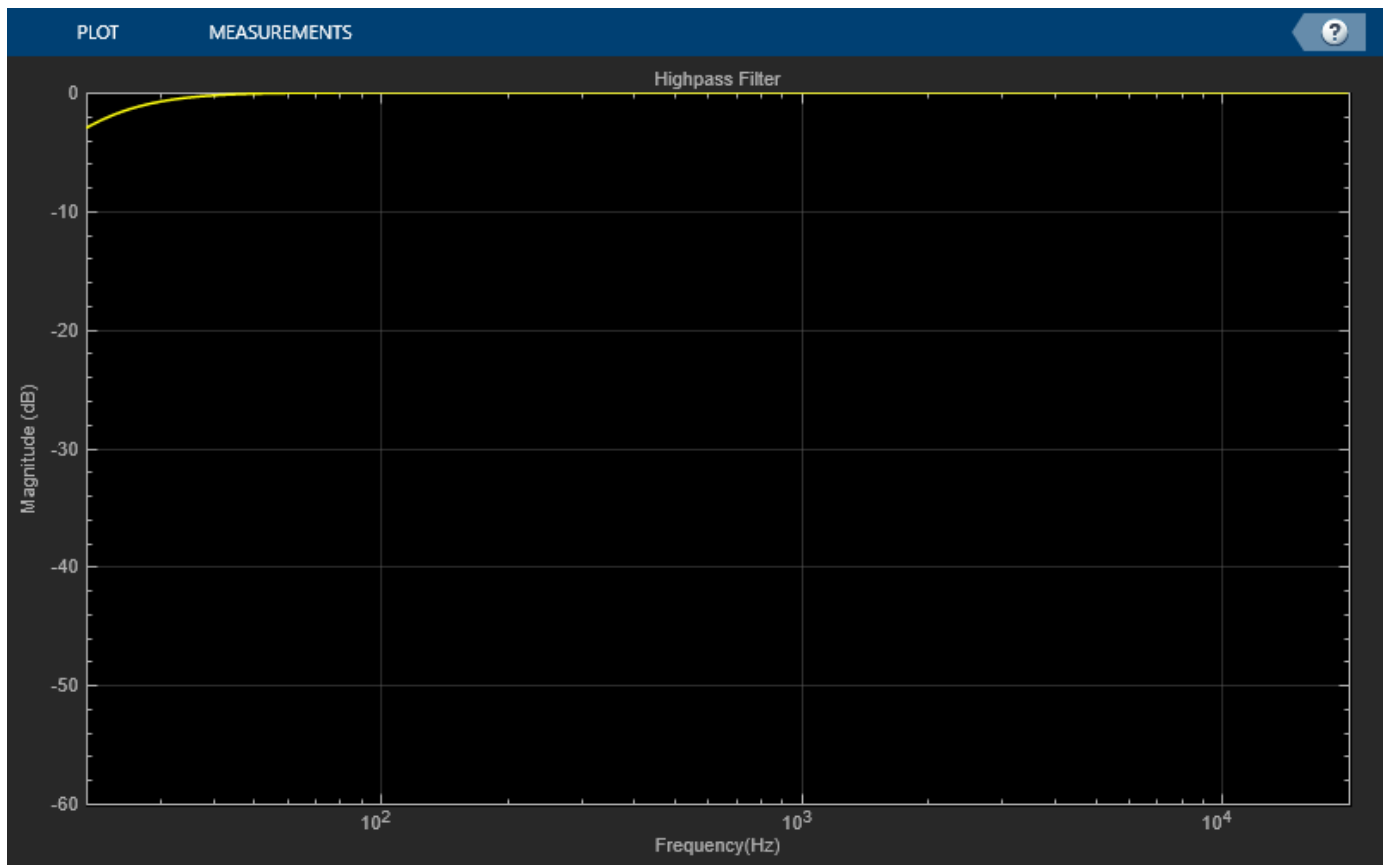
audioTestBench

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.

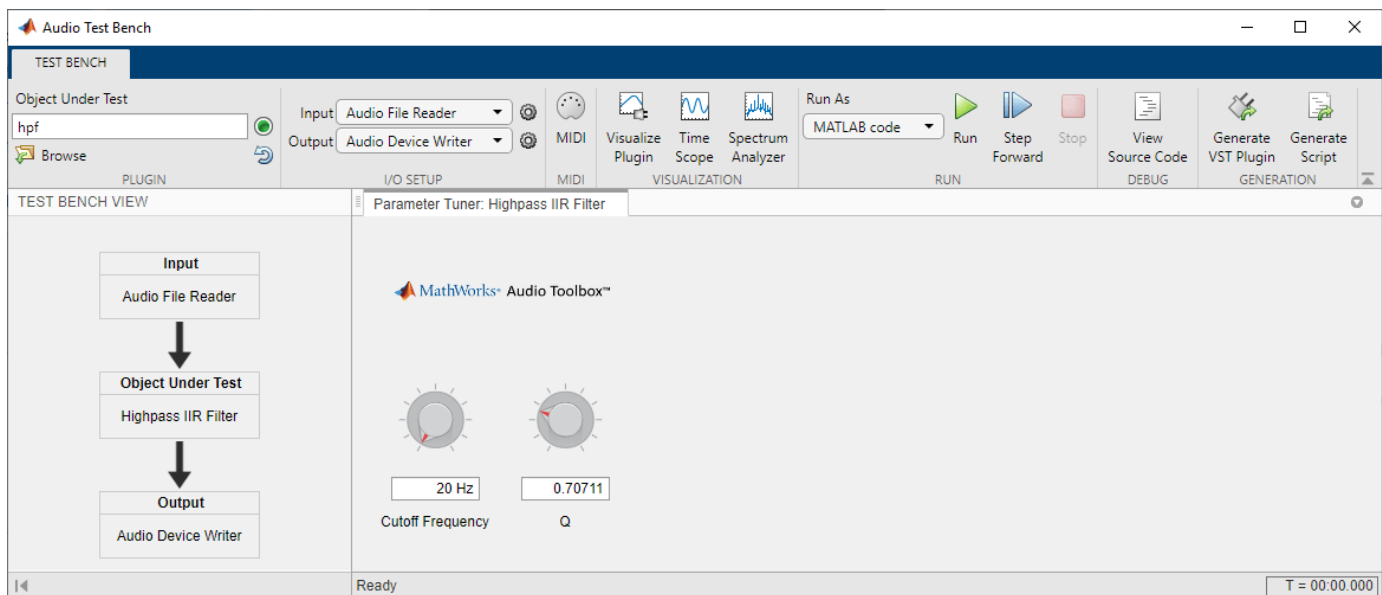
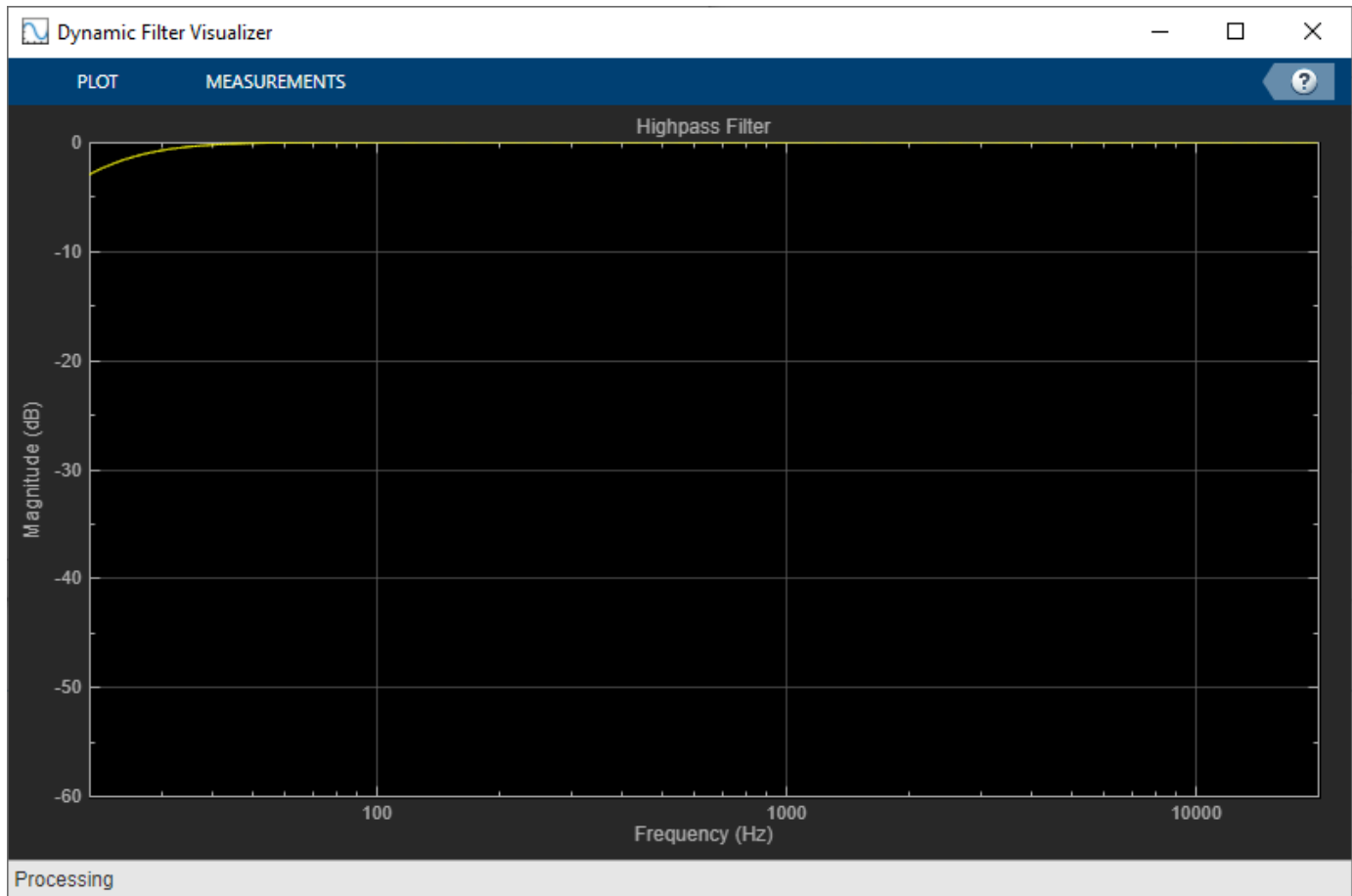
Update Visualization While Running Plugin

`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)

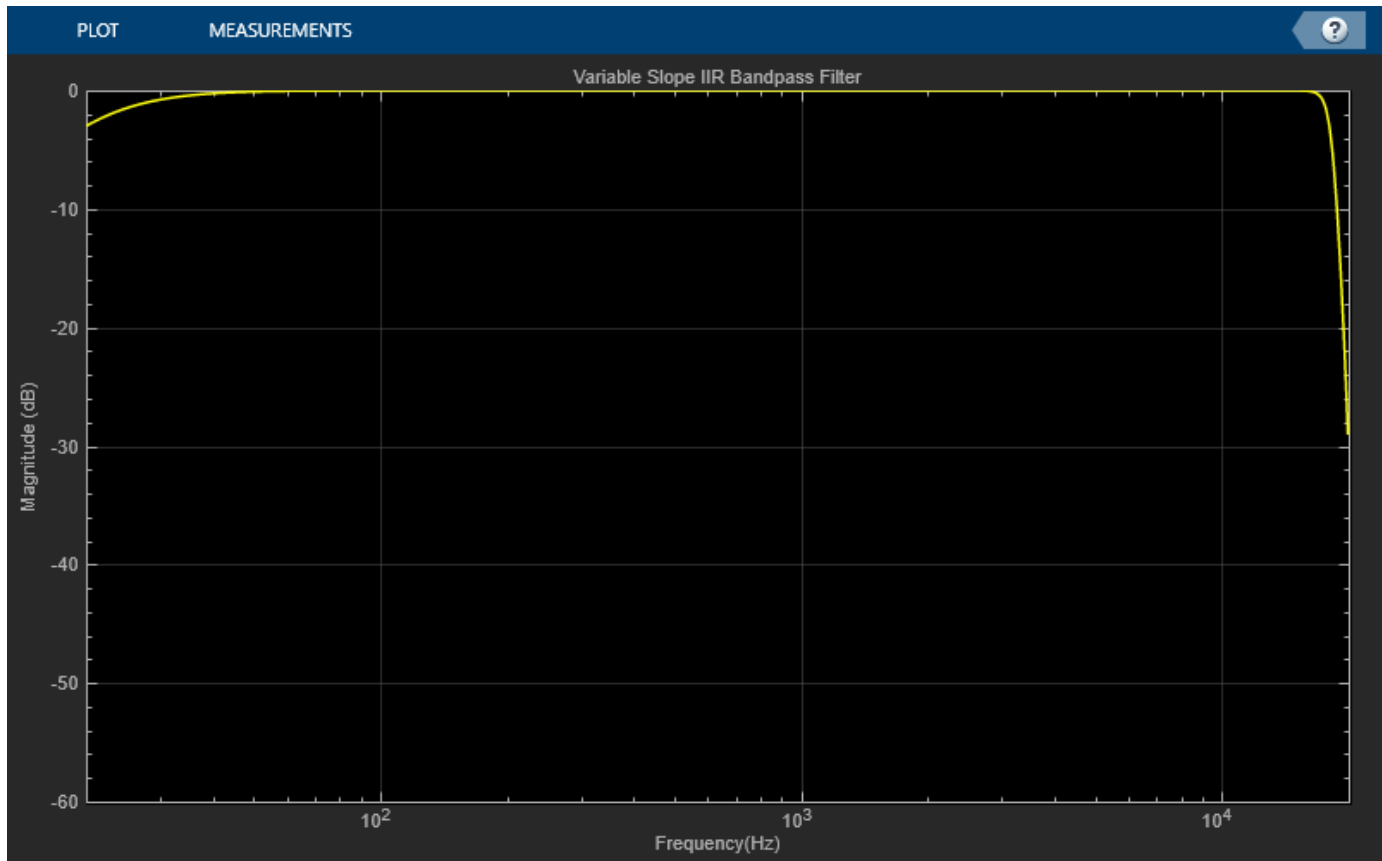


Note that moving the cutoff frequency in `audioTestBench` does not update the magnitude response plot. However, once the 'Run' (or play) button is pressed, you can see and hear the changing magnitude response of the filter as the cutoff frequency is tuned in real time.

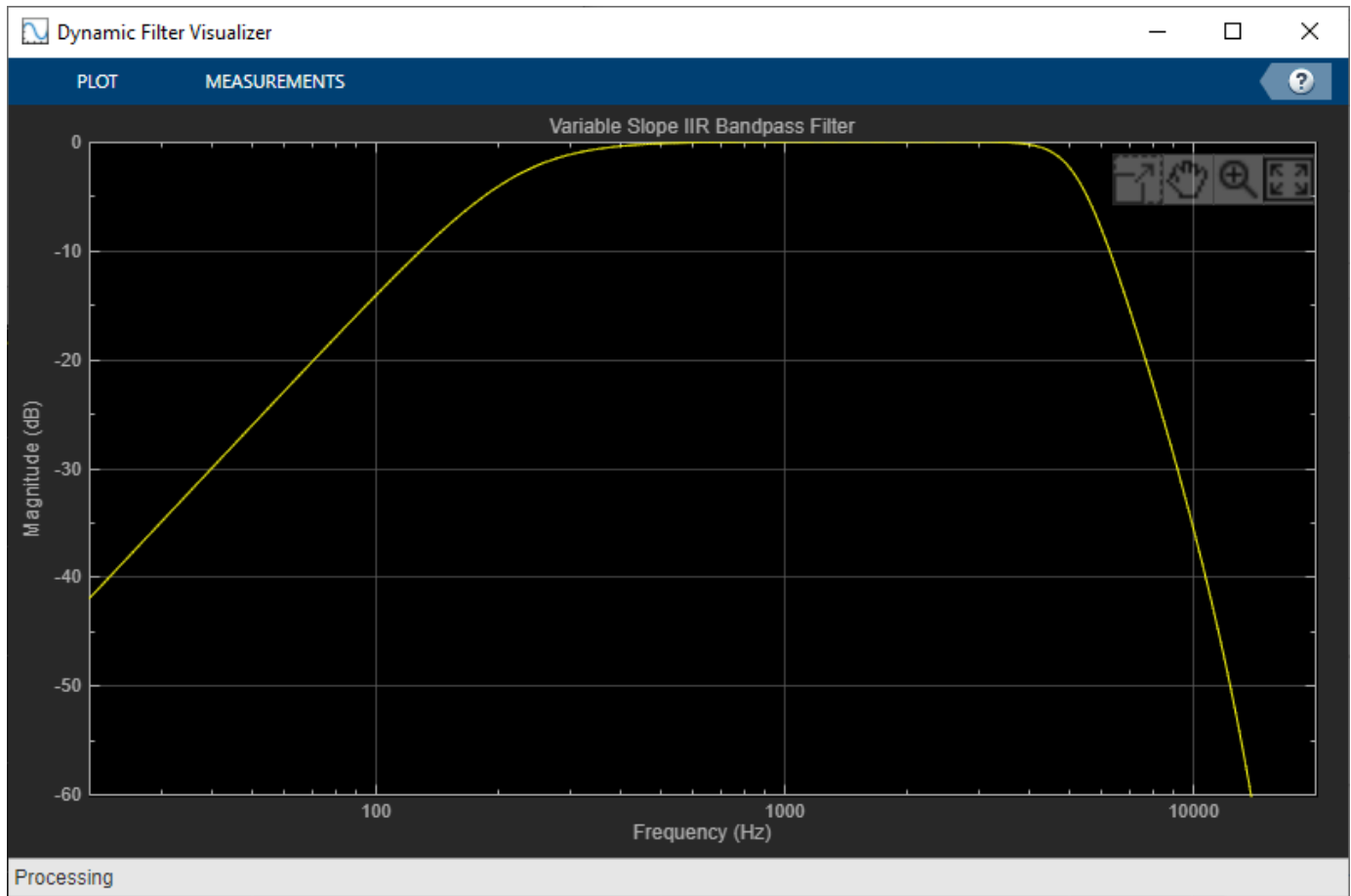
Update Visualization at Any Time

`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)
```

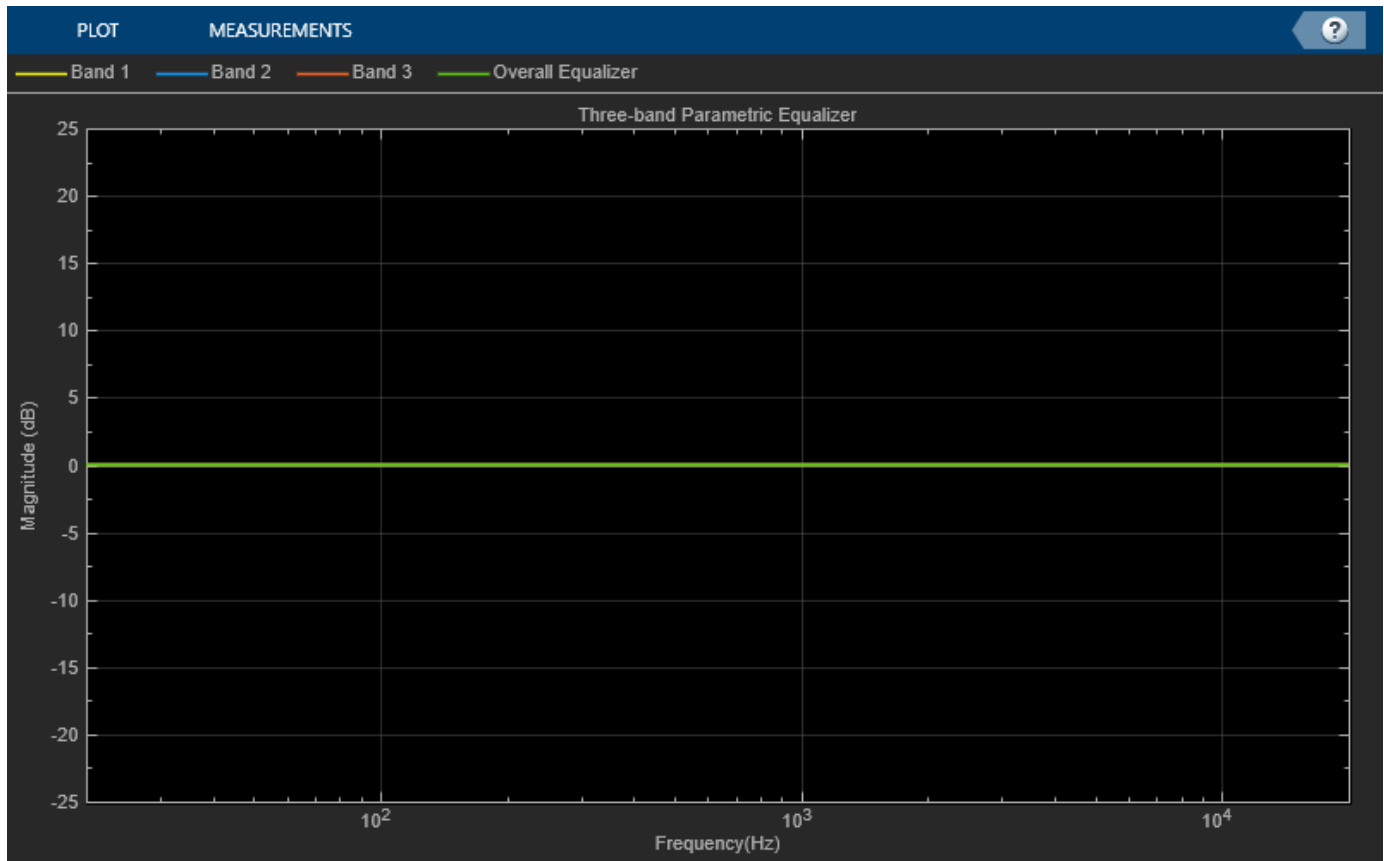


The figure shows a window titled "Audio Test Bench" with a "TEST BENCH" tab. The "Object Under Test" is set to "varfilter". The "Input" is "Audio File Reader" and the "Output" is "Audio Device Writer". The "Run As" dropdown is set to "MATLAB code". The "TEST BENCH VIEW" shows a block diagram with three stages: "Input" (Audio File Reader), "Object Under Test" (Variable Slope Bandpass), and "Output" (Audio Device Writer). The "Parameter Tuner: Variable Slope Bandpass" window is open, showing two frequency sliders for "Low Cutoff" (225.15 Hz) and "High Cutoff" (5166.3 Hz), and two slope dropdowns for "Low Slope" (12) and "High Slope" (30). The status bar at the bottom indicates "Ready" and "T = 00:00.000".

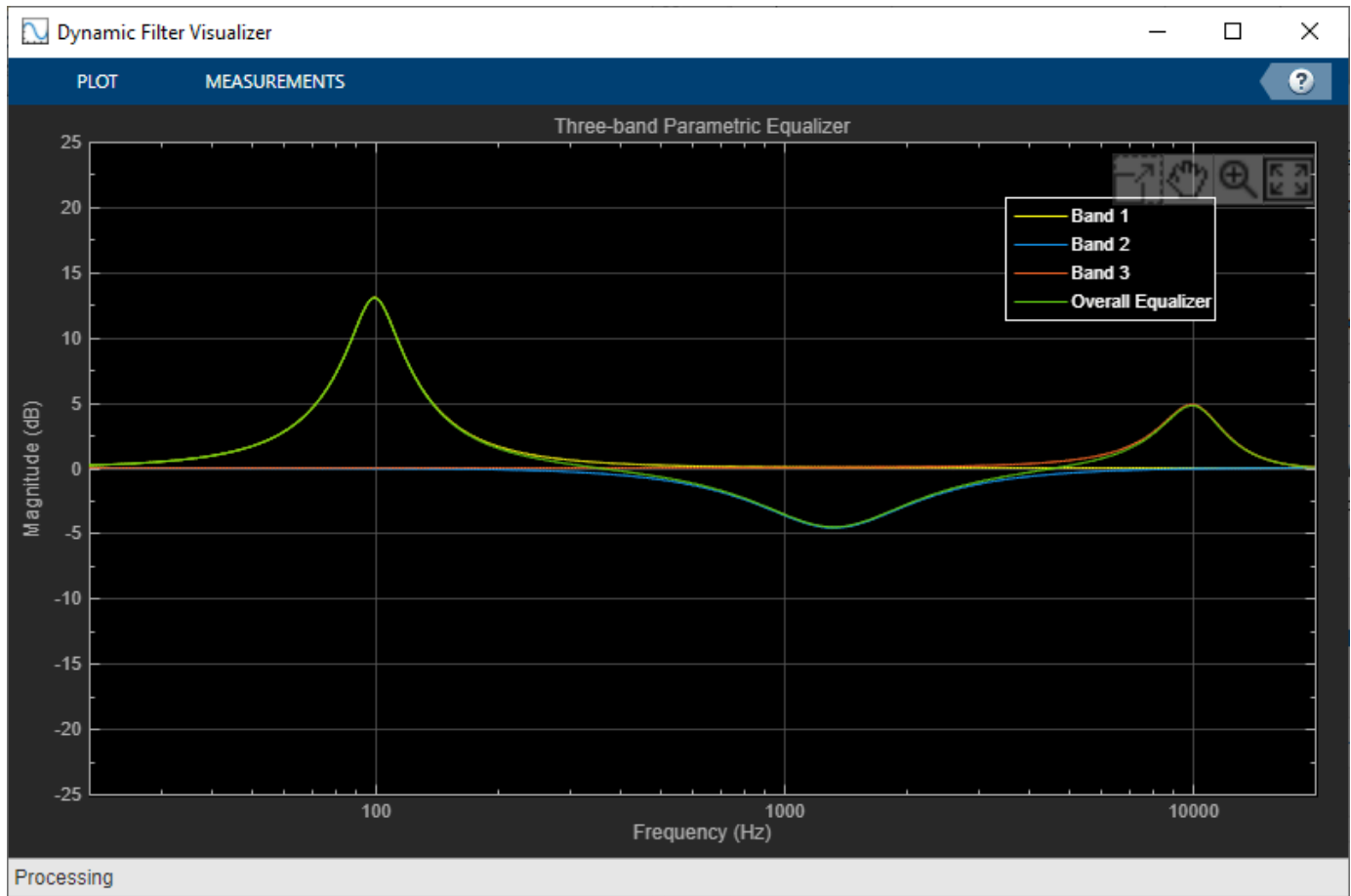
Visualize Individual and Combined Magnitude Response

`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)
```



The screenshot displays the Audio Test Bench software interface. The window title is "Audio Test Bench". The main area is divided into two panes. The left pane, titled "TEST BENCH VIEW", shows a block diagram of the test bench. It consists of an "Input" block containing "Audio File Reader", followed by an "Object Under Test" block containing "ParametricEQ", and finally an "Output" block containing "Audio Device Writer". The right pane, titled "Parameter Tuner: ParametricEQ", shows the configuration for the ParametricEQ plugin. It features three vertical sliders for gain, three rotary knobs for frequency, and three rotary knobs for Q factor. The current values are:

Parameter	Value
Low Gain	13.043 dB
Mid Gain	-4.5963 dB
High Gain	4.8447 dB
Low Frequency	100 Hz
Mid Frequency	1329.9 Hz
High Frequency	10000 Hz
Low Q	2
Mid Q	0.90467
High Q	2.2075

The status bar at the bottom indicates "Ready" and a timer showing "T = 00:00.000".

```
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)

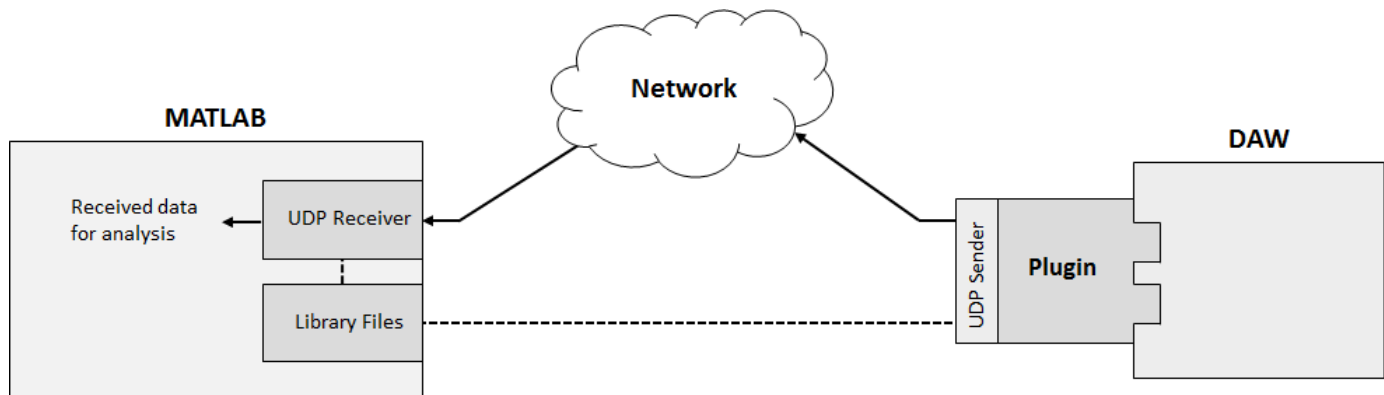
UDP is a core member of the Internet protocol suite. It is a simple connectionless transmission that does not employ any methods for error checking. Because it does not check for errors, UDP is a fast but unreliable alternative to the transmission control protocol (TCP) and stream control transmission 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
```

.....

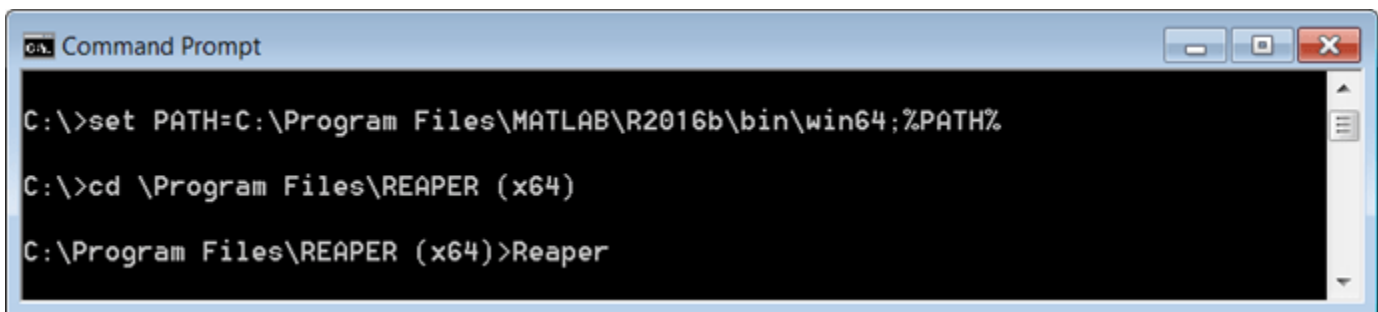
The generated plugin is saved to your current folder and named `UDPSender`.

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.



```

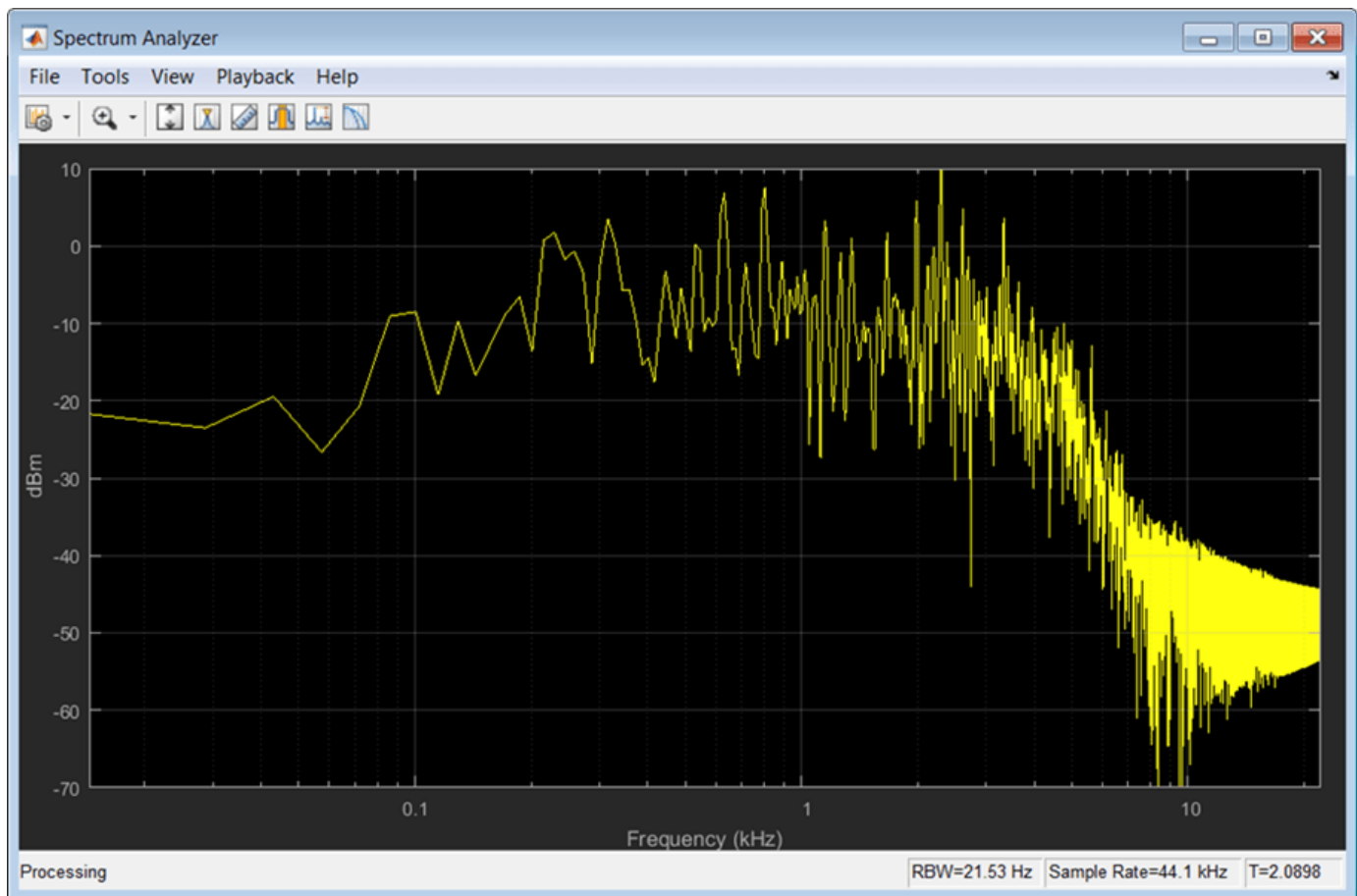
C:\>set PATH=C:\Program Files\MATLAB\R2016b\bin\win64;%PATH%
C:\>cd \Program Files\REAPER (x64)
C:\Program Files\REAPER (x64)>Reaper

```

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 spectrum analyzer for analysis.

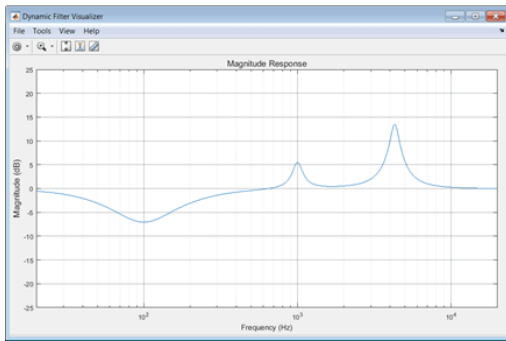


Send Coefficients from DAW to MATLAB

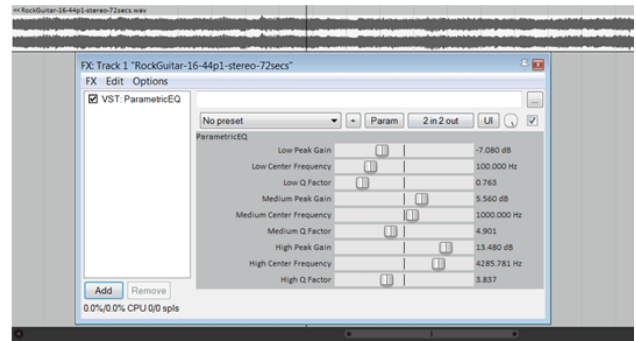
1. Follow steps 1-2 from **Send Audio from DAW to MATLAB**, replacing `audiopluginexample.UDPsender` with `audiopluginexample.ParametricEqualizerWithUDP`.
2. Receive and process filter coefficients
 - a. In the DAW, open the generated `ParameterEqualizerWithUDP` file. The plugin display name is `ParametricEQ`.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

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:

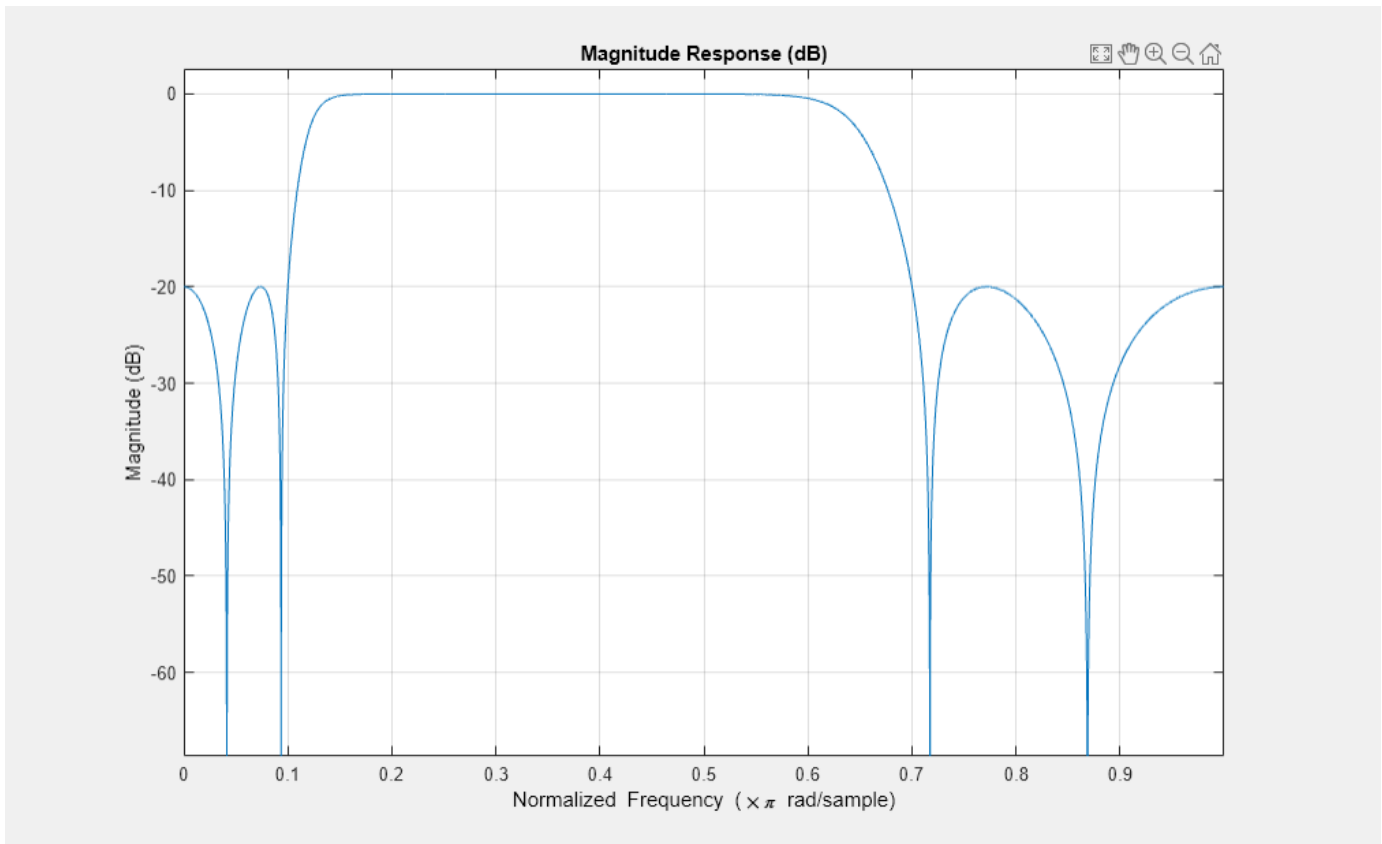
- The near-end speech signal $v(n)$
- The far-end echoed speech signal $\hat{d}(n)$

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
```

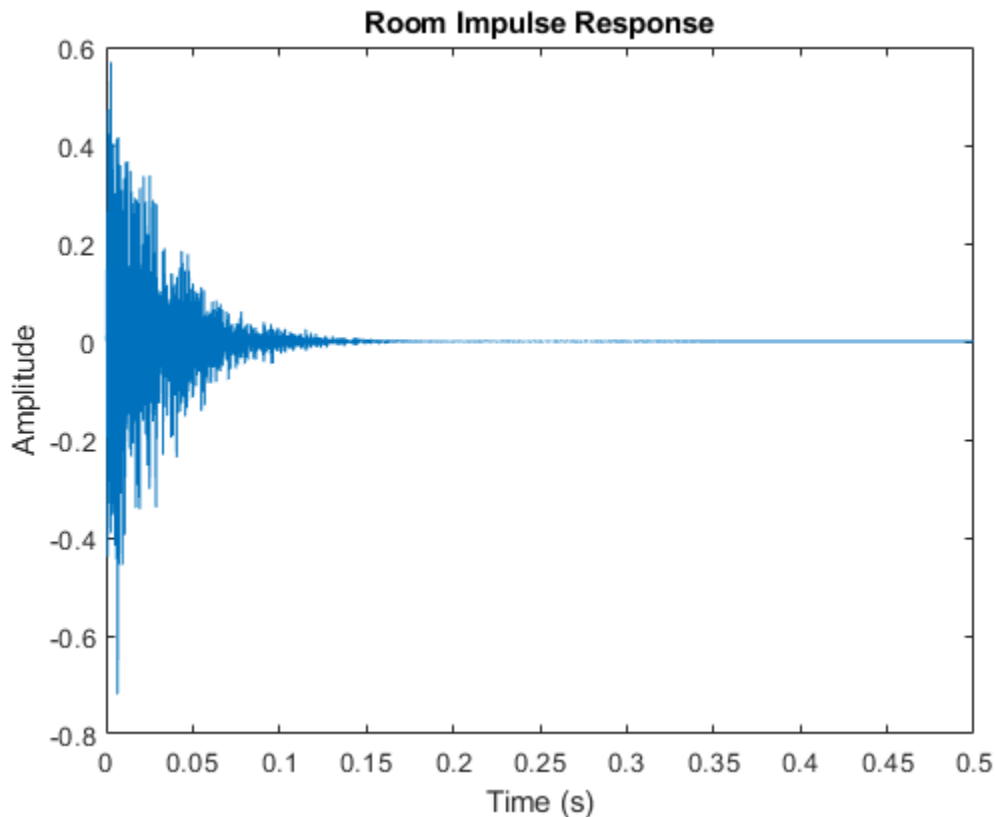



```

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];

```



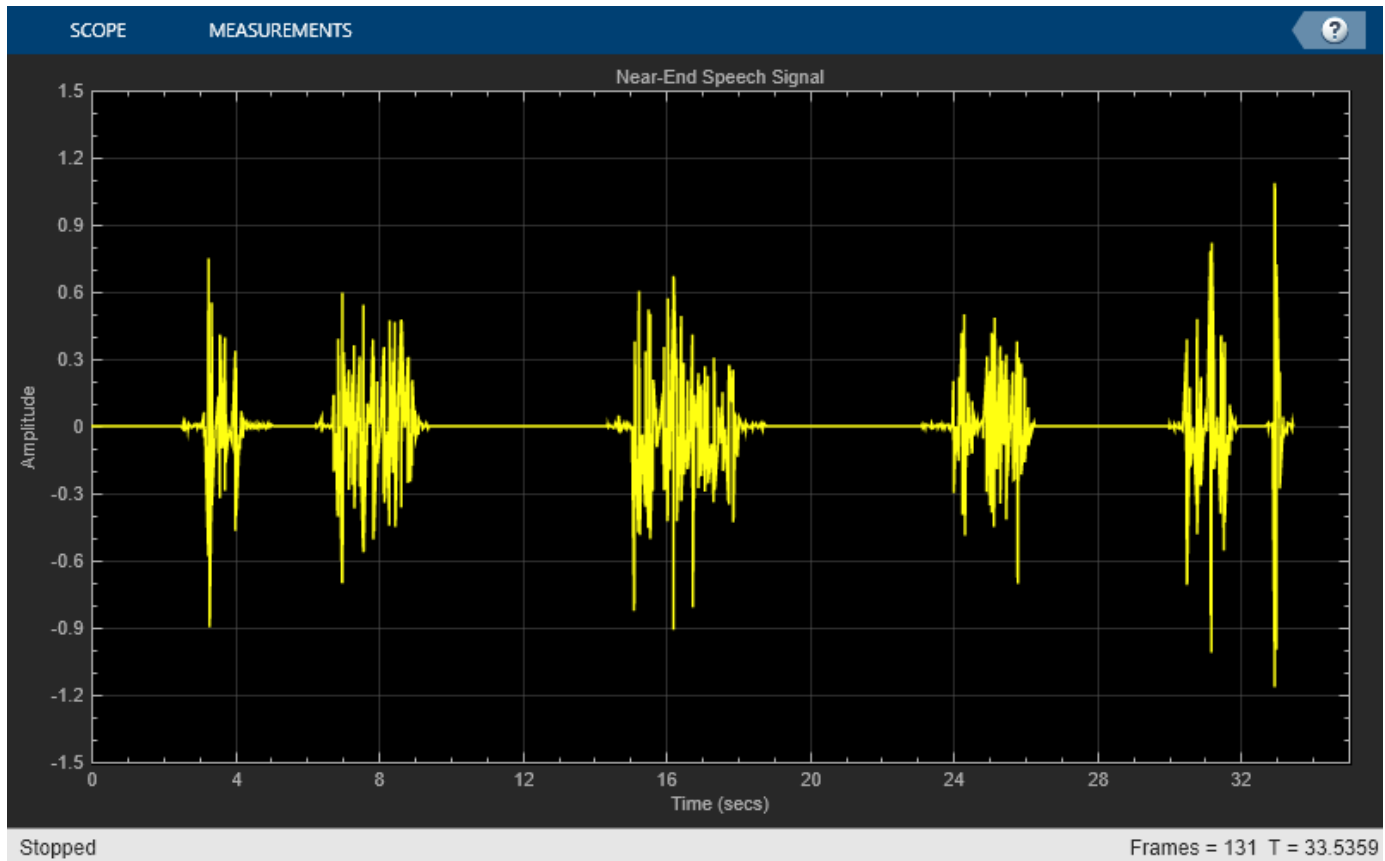
The Near-End Speech Signal

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.

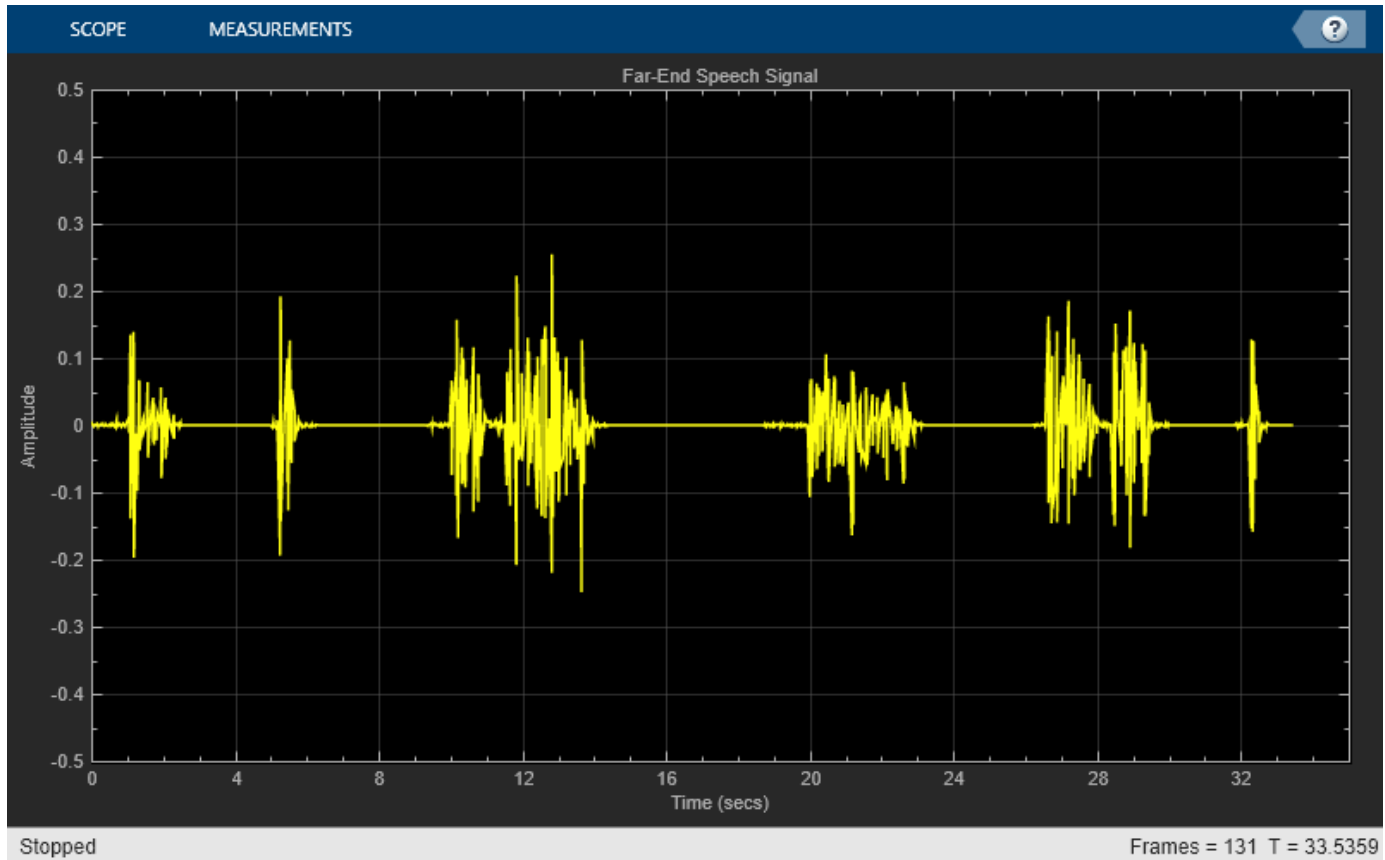
```
load farspeech
farSpeechSrc = dsp.SignalSource('Signal',x,'SamplesPerFrame',frameSize);
farSpeechSink = dsp.SignalSink;
farSpeechScope = timescope('SampleRate', fs, 'TimeSpanSource','Property',...
    'TimeSpan', 35, 'TimeSpanOverrunAction', 'Scroll', ...
    'YLimits', [-0.5 0.5], ...
    'BufferLength', 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, 'TimeSpanOvverrunAction', 'Scroll', ...
    'YLimits', [-1 1], ...
    'BufferLength', length(x), ...
    'Title', 'Microphone Signal', ...
    'ShowGrid', true);

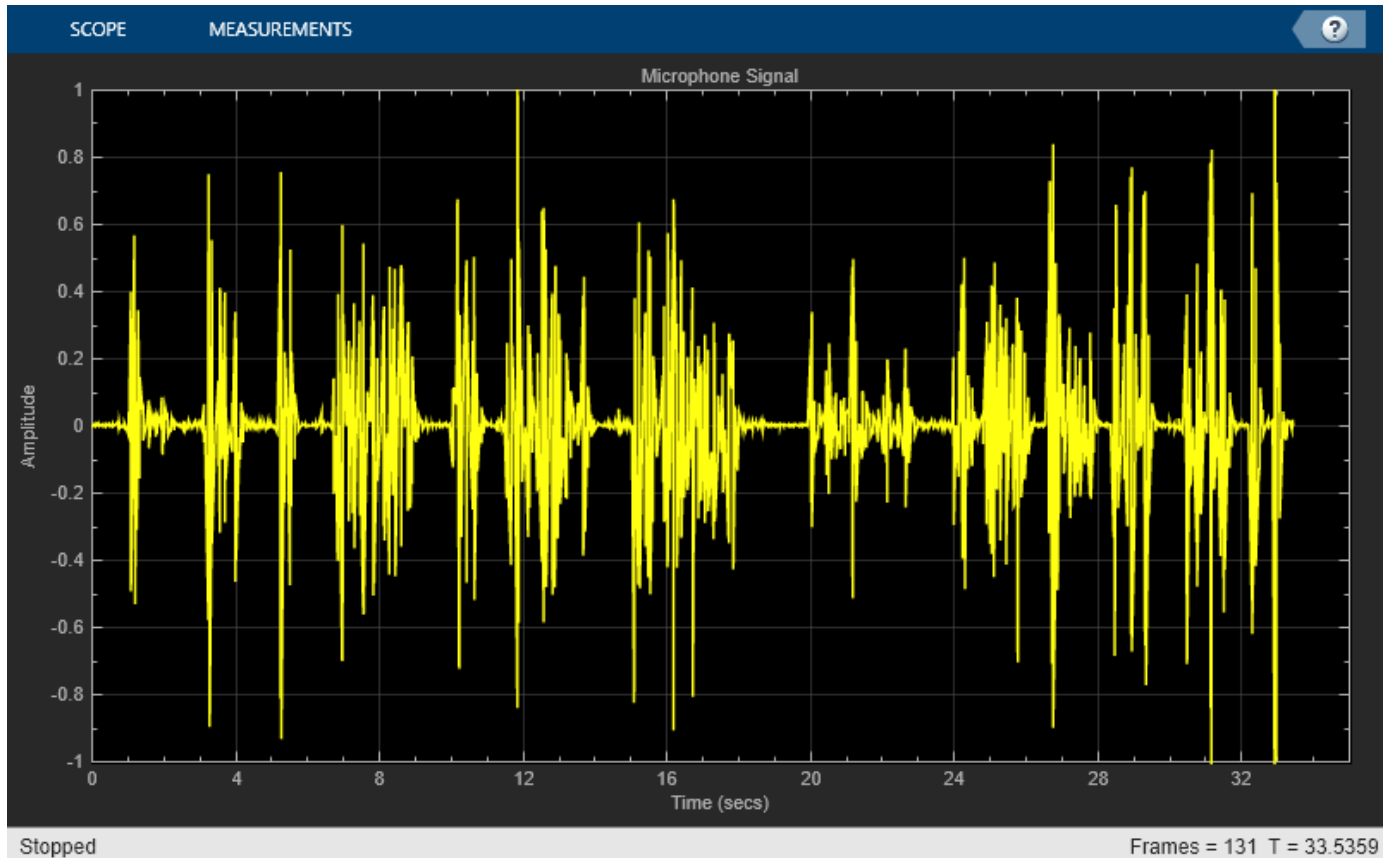
% Stream processing loop
while(~isDone(farSpeechEchoSrc))
    % Microphone signal = echoed far-end + near-end + noise
    micSignal = farSpeechEchoSrc() + nearSpeechSrc() + ...
        0.001*randn(frameSize,1);
    % Send the speech samples to the output audio device

```

```

    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 Canceller 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);

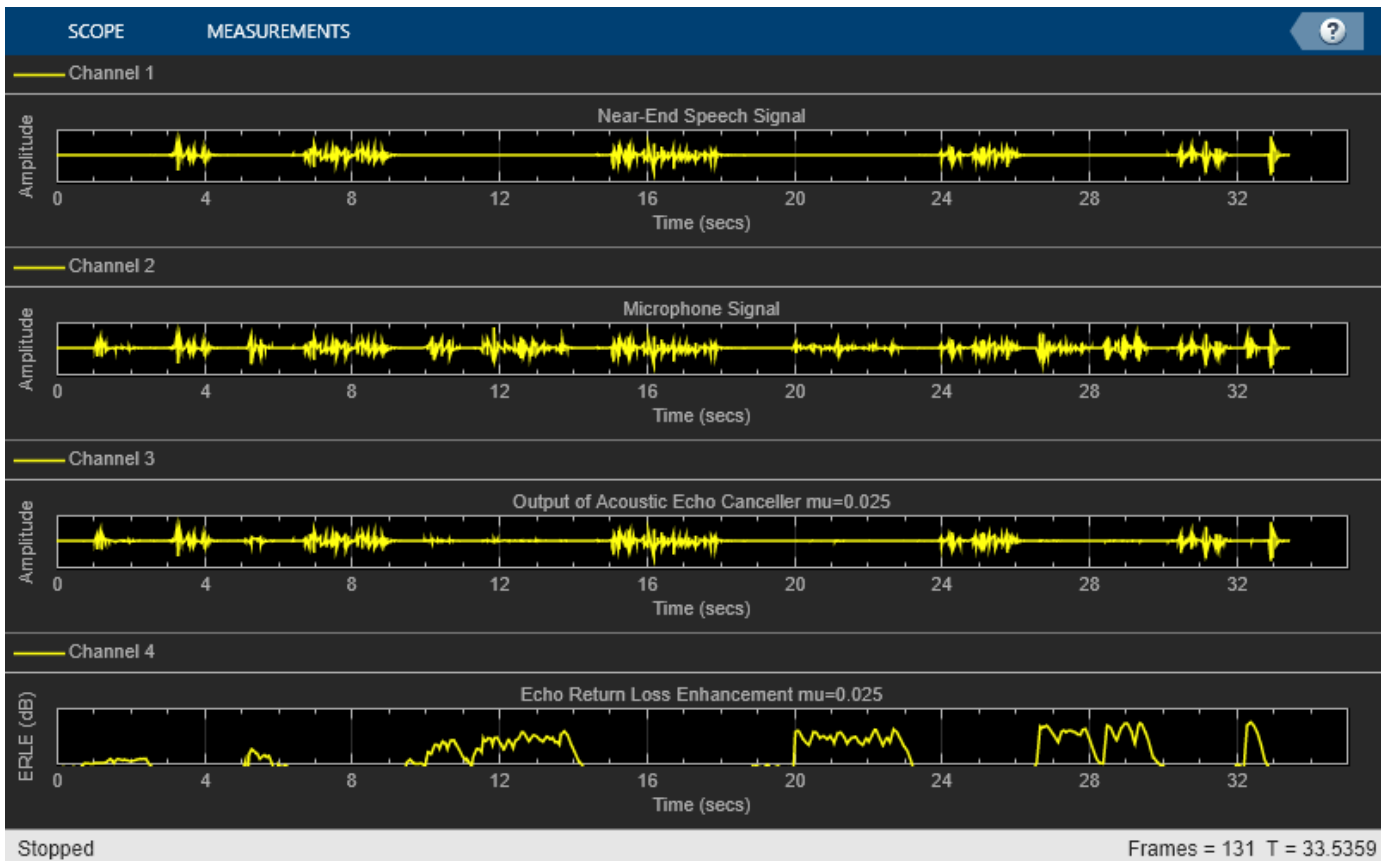
% Stream processing loop - adaptive filter step size = 0.025
while(~isDone(nearSpeechSrc))
    nearSpeech = nearSpeechSrc();
    farSpeech = farSpeechSrc();
    farSpeechEcho = farSpeechEchoSrc();
    micSignal = micSrc();
    % Apply FDAF
    [y,e] = echoCanceller(farSpeech, micSignal);
    % Send the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

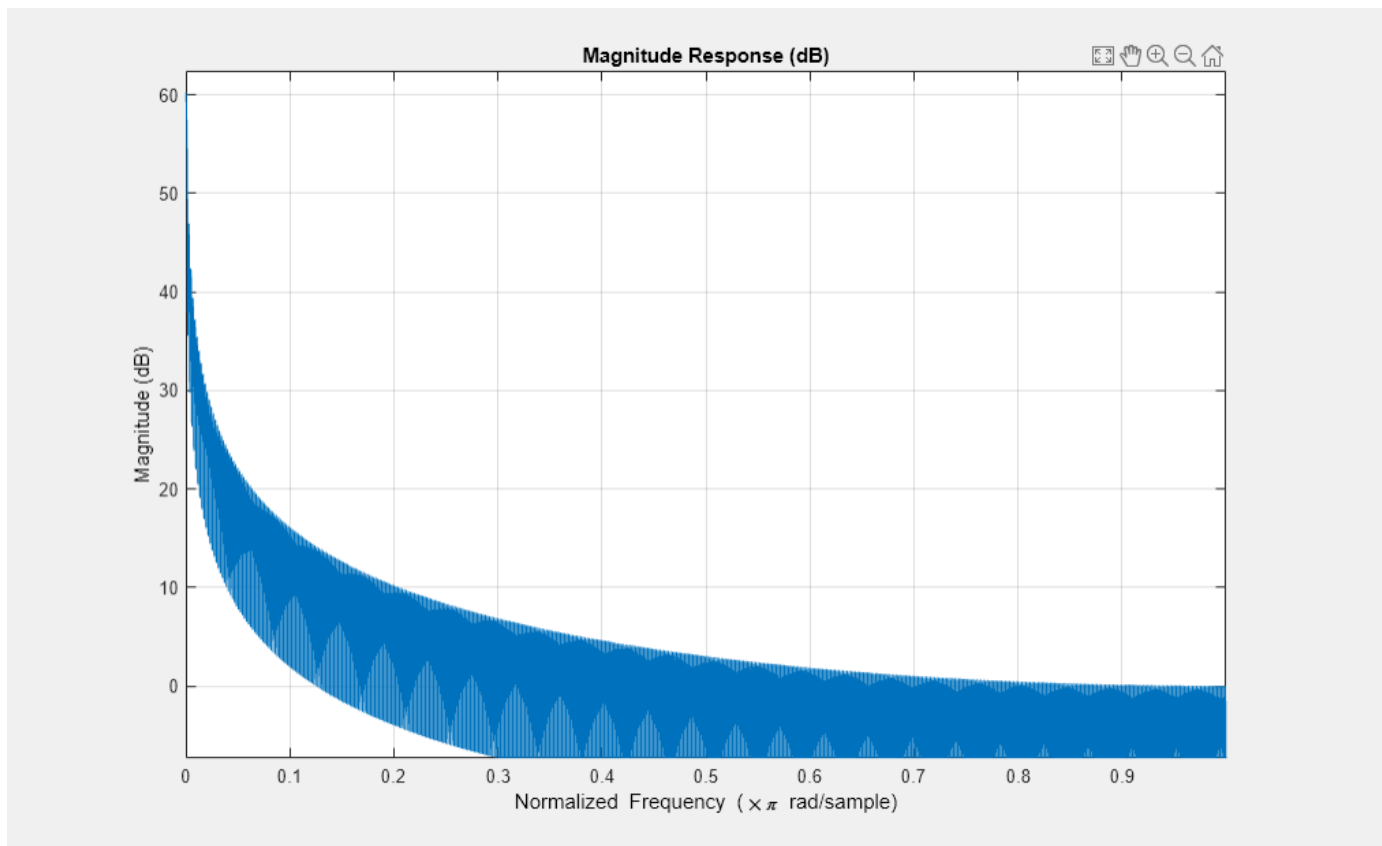
```

```

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 Celler 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();
```

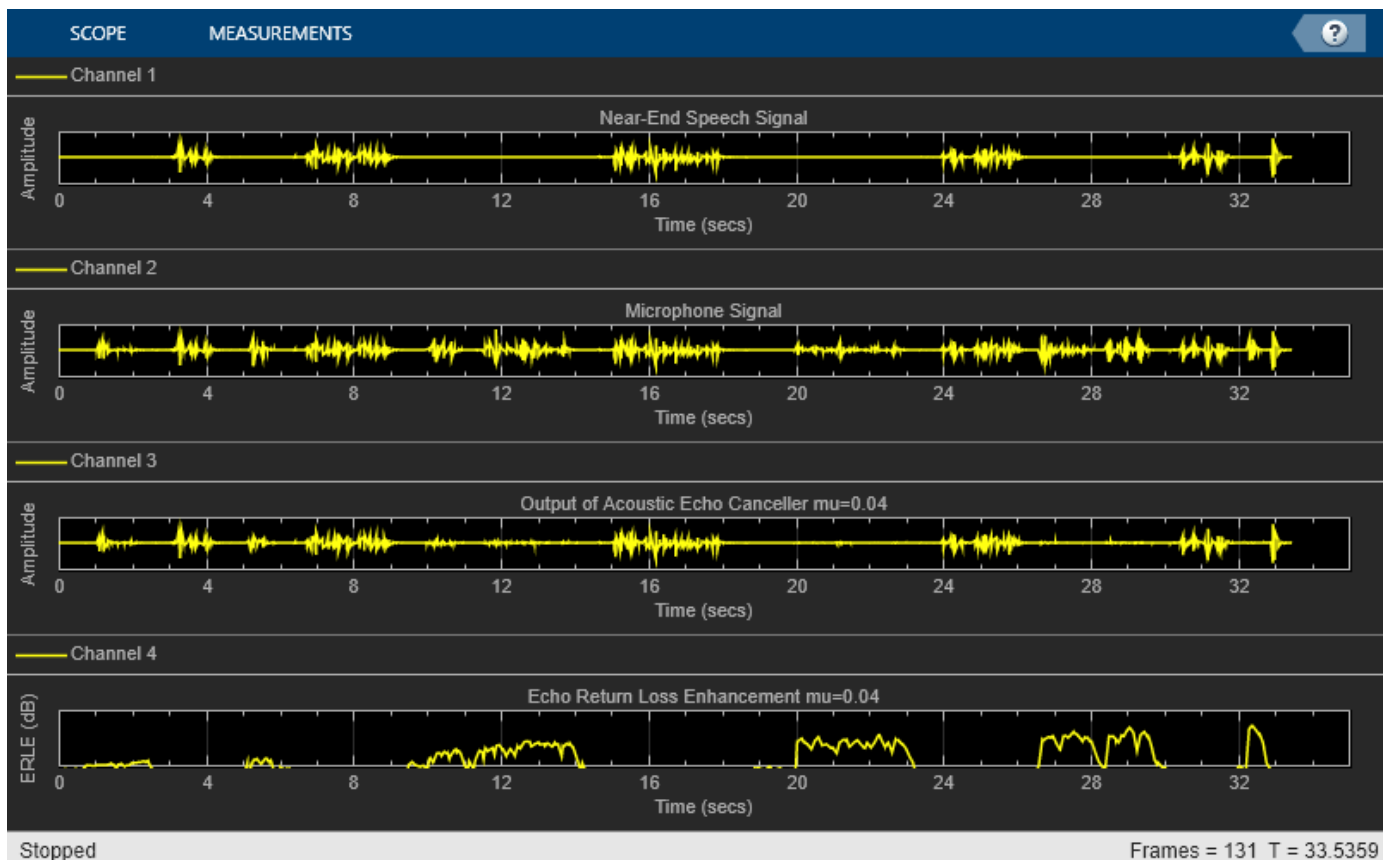


```

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

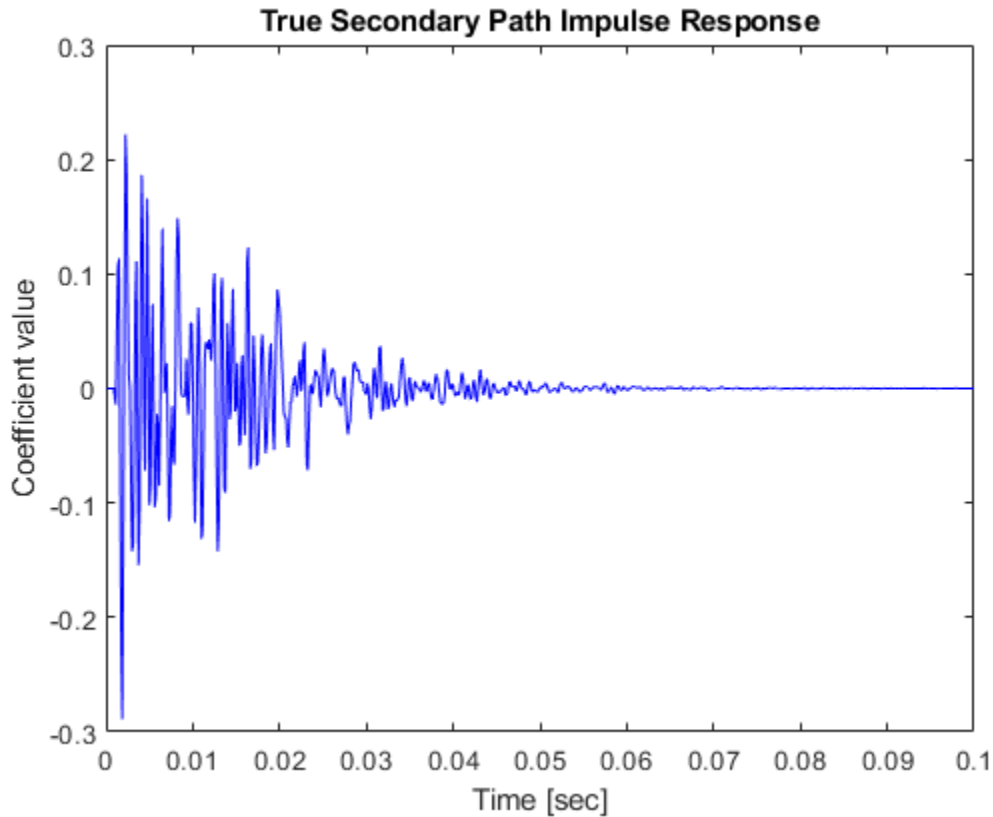
The secondary propagation path is the path the anti-noise takes from the output loudspeaker to the error microphone within the quiet zone. The following commands generate a loudspeaker-to-error microphone impulse response that is bandlimited to the range 160 - 2000 Hz and with a filter length of 0.1 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 through
    0.01*randn(ntrS,1); % measurement noise at the microphone
```

Designing the Secondary Propagation Path Estimate

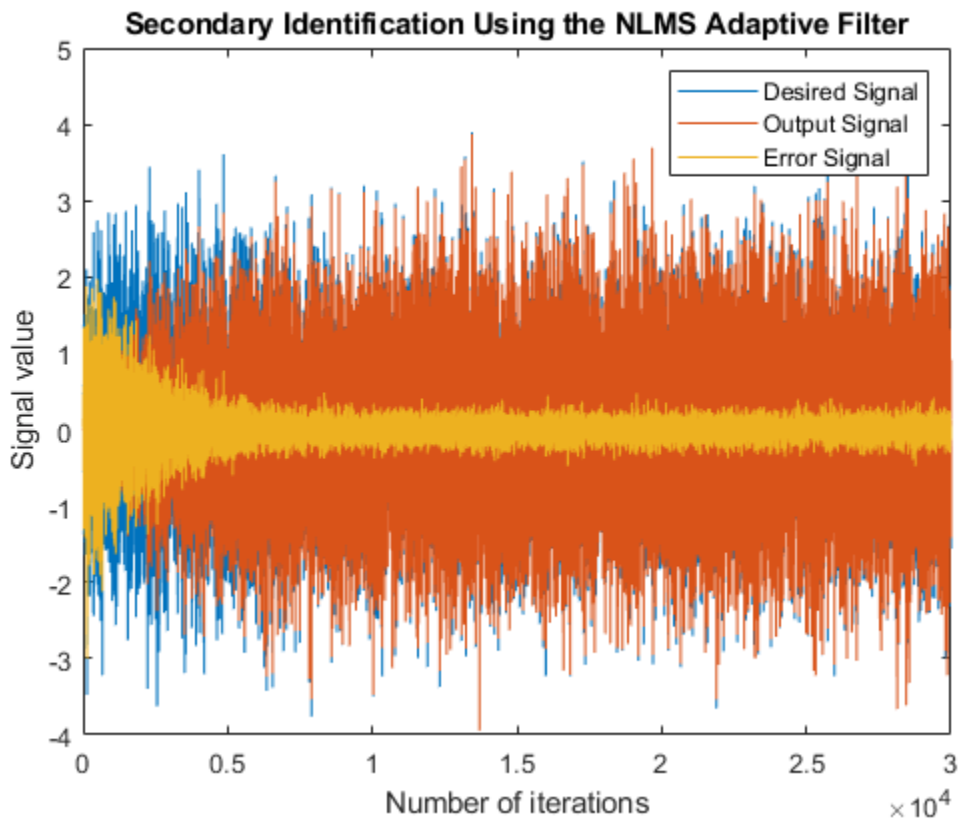
Typically, the length of the secondary path filter estimate is not as long as the actual secondary path and need not be for adequate control in most cases. We shall use a secondary path filter length of 250 taps, corresponding to an impulse response length of 31 ms. While any adaptive FIR filtering algorithm 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');

```



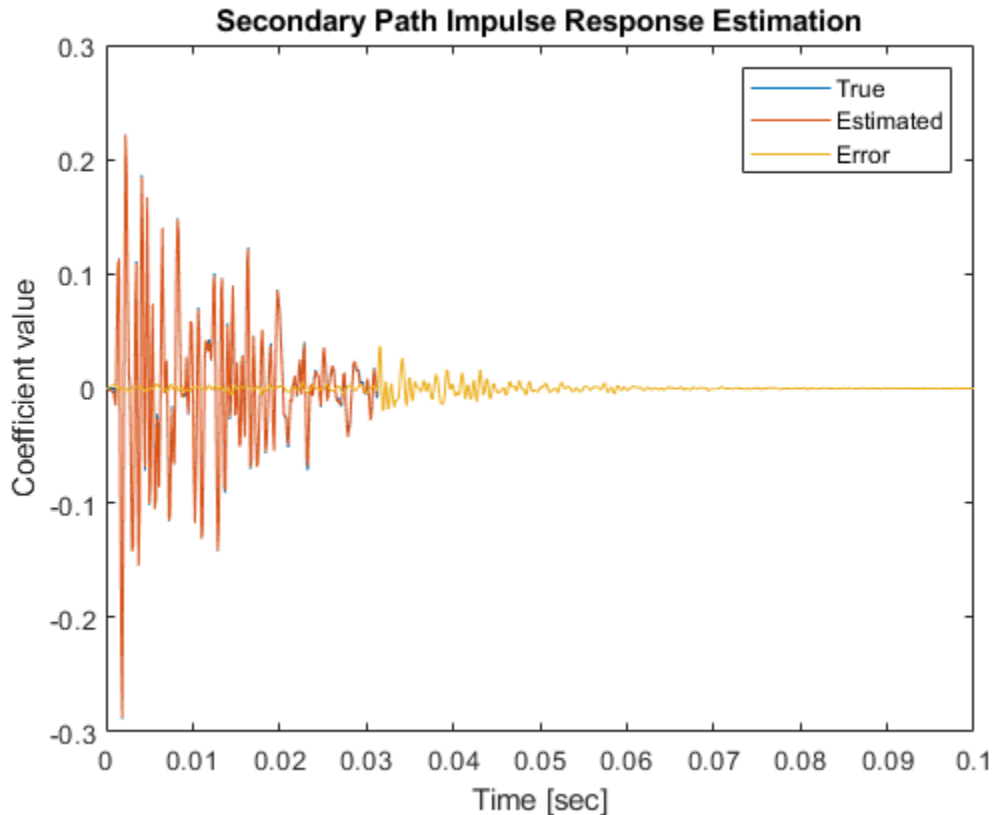
Accuracy of the Secondary Path Estimate

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.

```

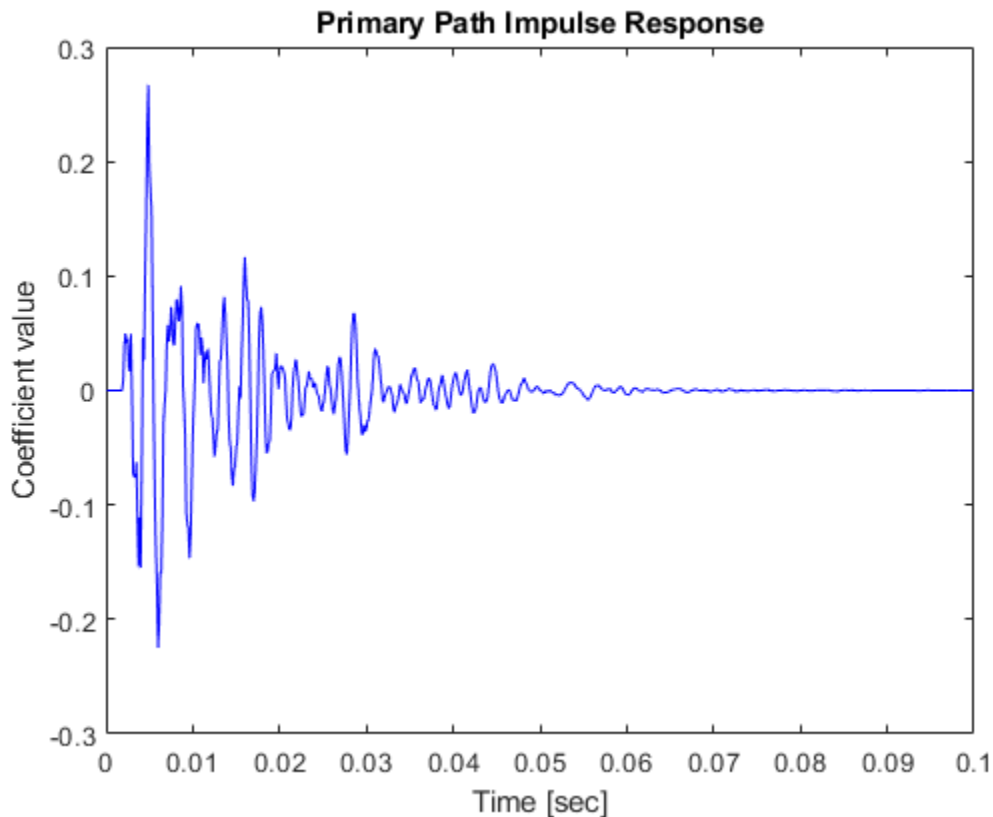
delayW = 15;
Flow   = 200; % Lower band-edge: 200 Hz
Fhigh  = 800; % Upper band-edge: 800 Hz
Ast    = 20;  % 20 dB stopband attenuation
Nfilt  = 10;  % Filter order

% 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.FilteredXLMSFilter('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 = spectrumAnalyzer('SampleRate',Fs,'OverlapPercent',80, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'ShowLegend',true, ...
    'ChannelNames', {'Original noisy signal', 'Attenuated noise'});

```

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

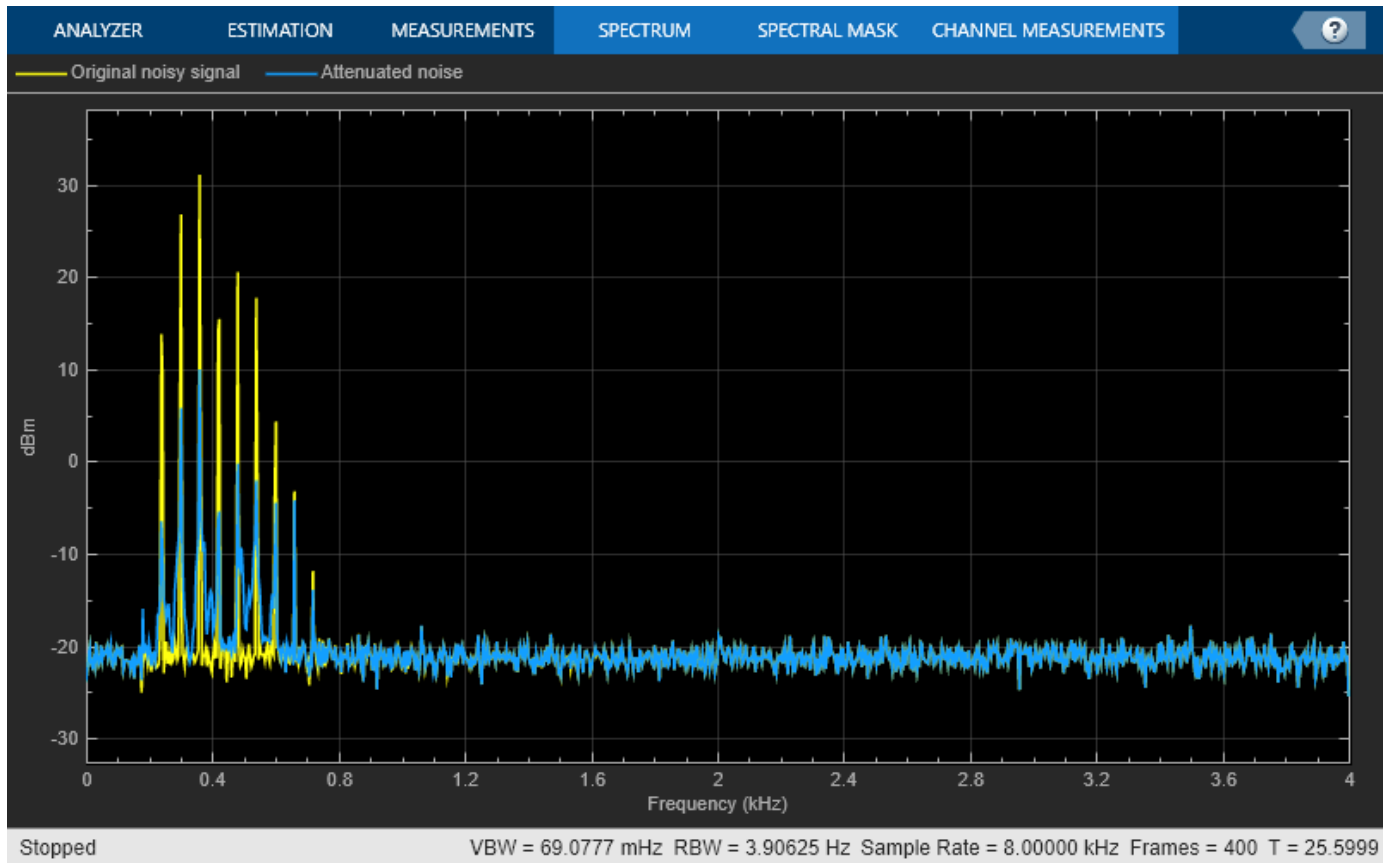
Here we simulate the active noise control system. To emphasize the difference we run the system with no active noise control for the first 200 iterations. Listening to its sound at the error microphone before cancellation, it has the characteristic industrial "whine" of such motors.

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

```



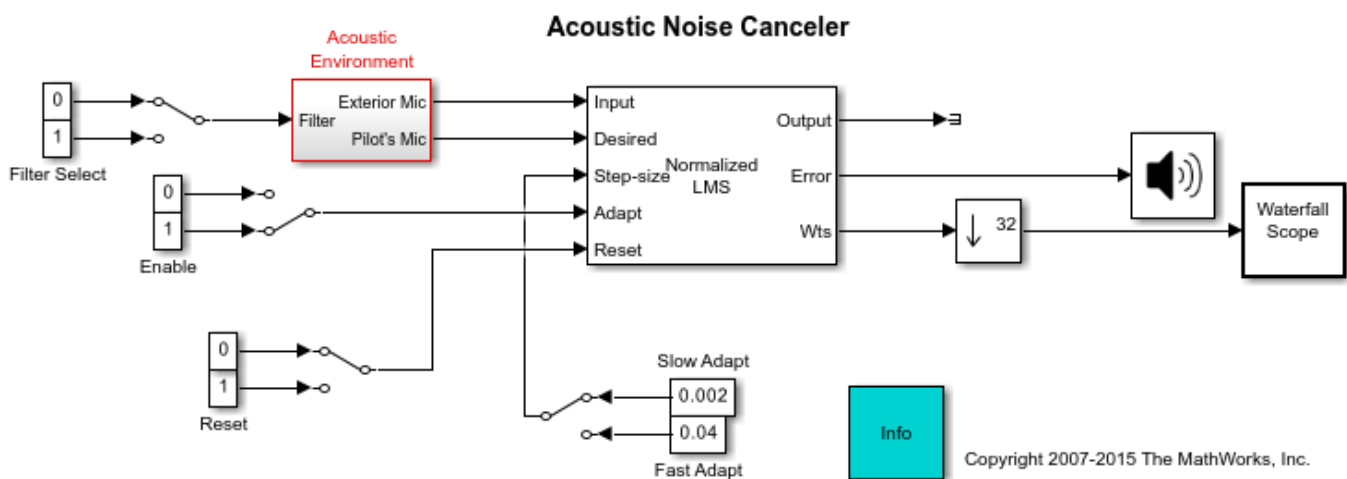
Acoustic Noise Cancellation Using LMS

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

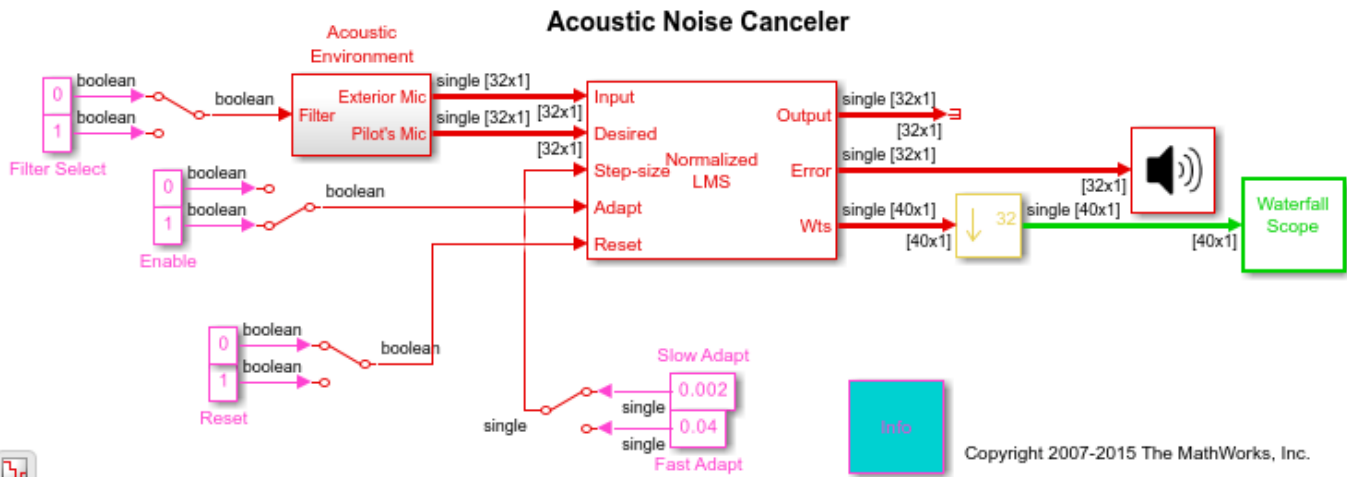
In the model, the signal output at the upper port of the Acoustic Environment subsystem is white noise. The signal output at the lower port is composed of colored noise and a signal from a WAV file. This example model uses an adaptive filter to remove the noise from the signal output at the lower port. 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

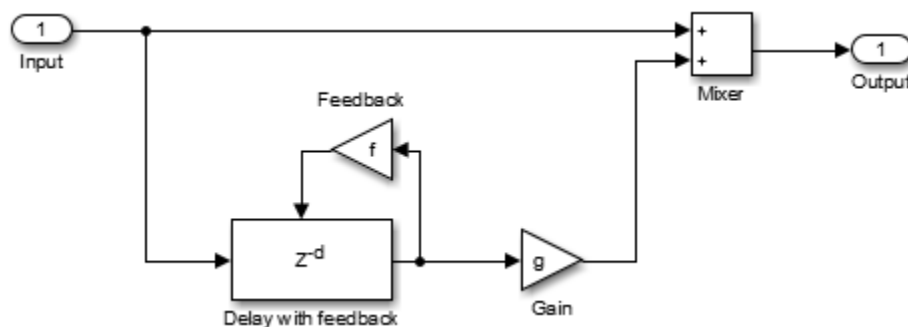
This example shows how to design and use three audio effects that are based on varying delay: echo, chorus and flanger. The example also shows how the algorithms, developed in MATLAB, can be easily ported to Simulink.

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.

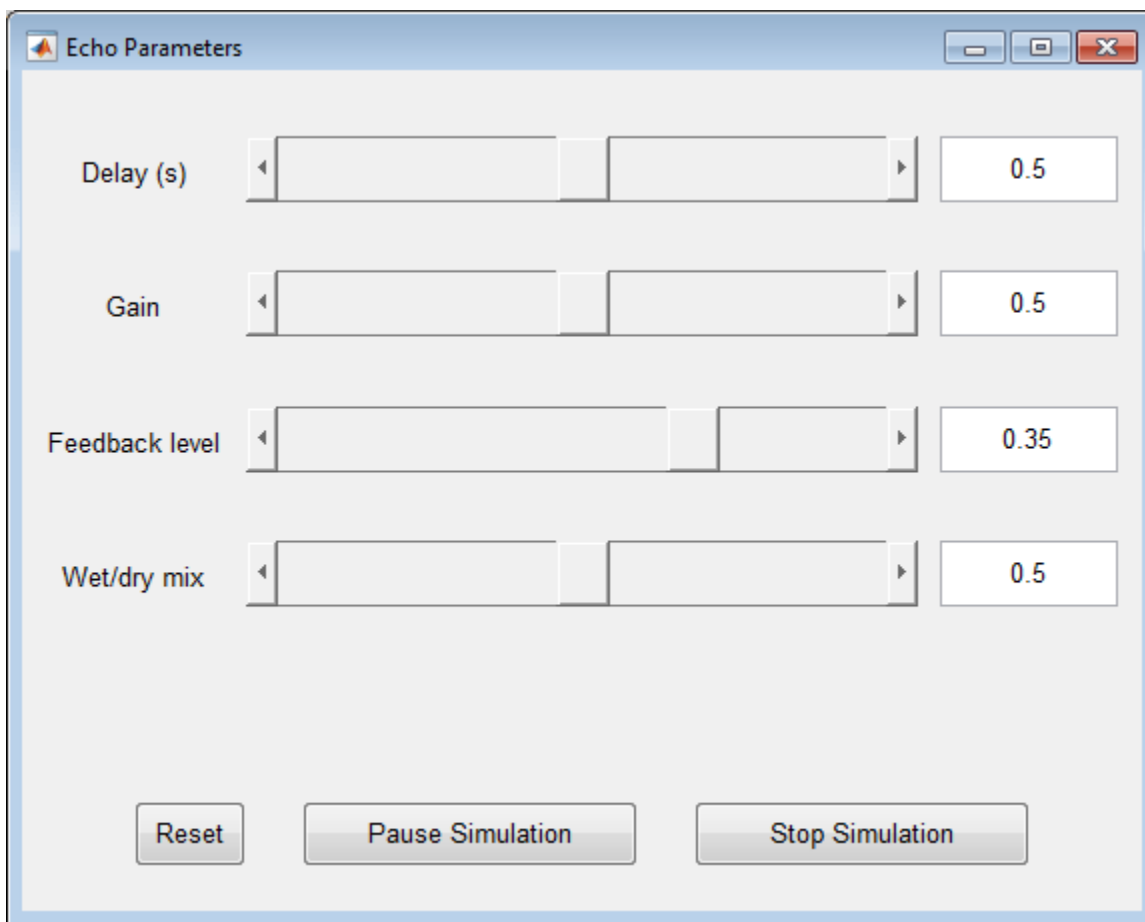


The echo effect example has four tunable parameters that can be modified while the simulation is running:

- 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

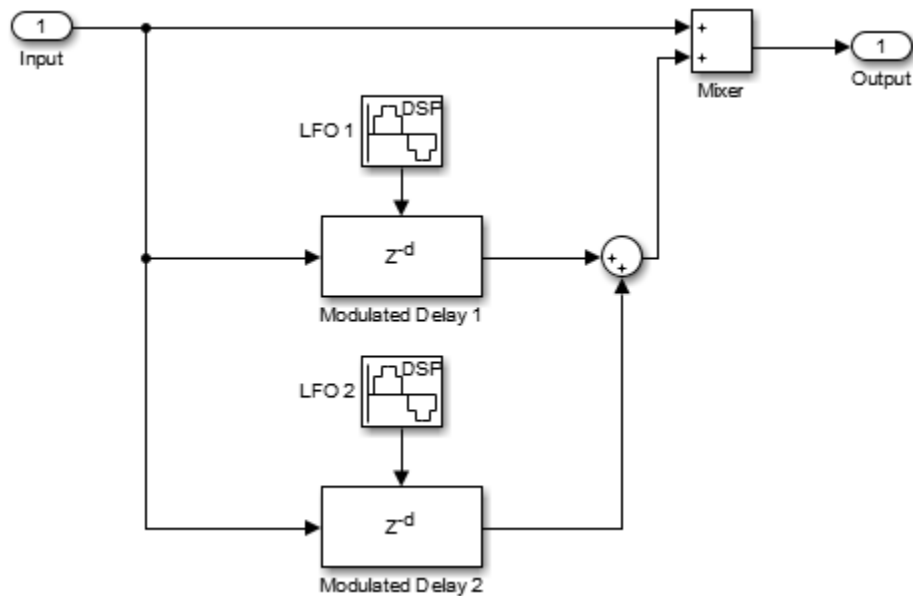
You can try out `audioexample.Echo` by running `audioDelayEffectsExampleApp` with 'echo' as input. The example reads an audio signal from a file, applies the echo effect, and then plays the processed signal through your audio output device. It also launches a UI that allows you to tune the parameters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

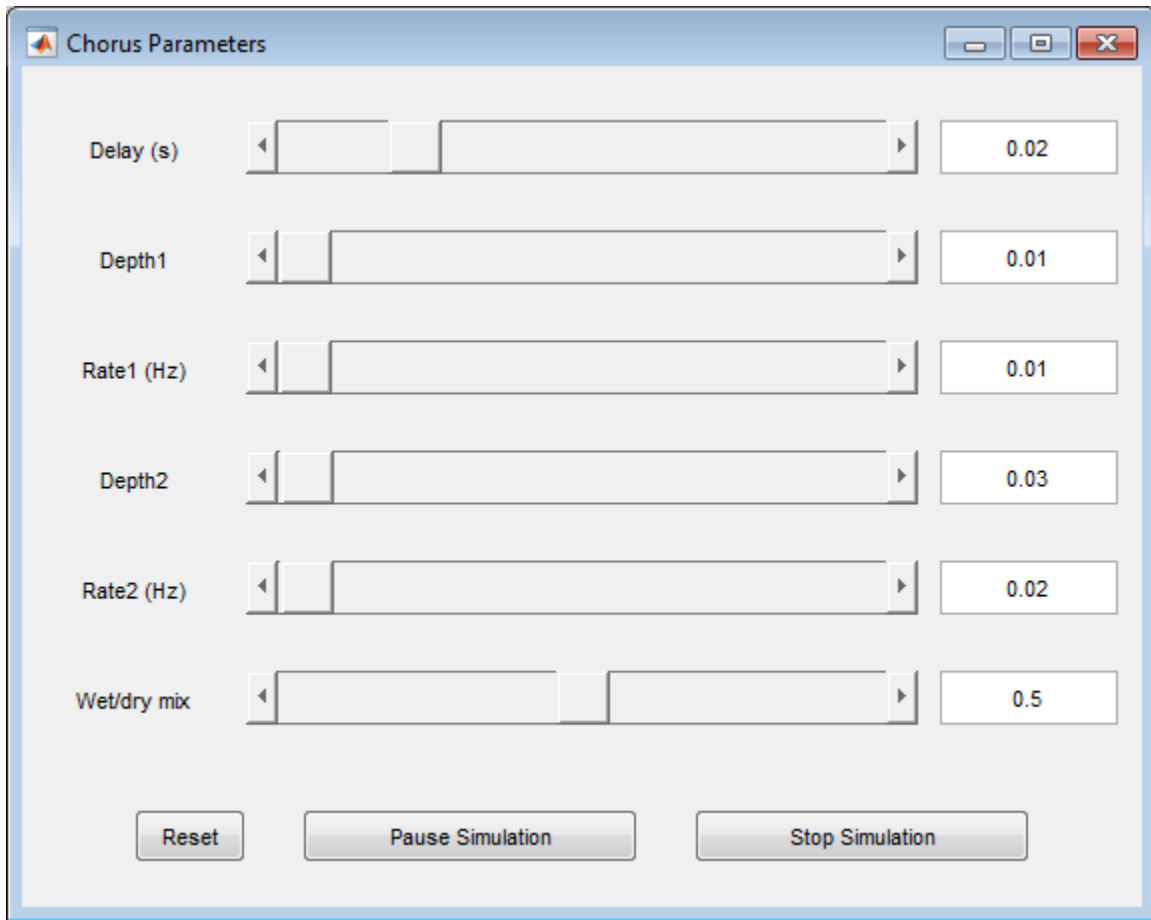


The chorus effect example has six tunable parameters that can be modified while the simulation is running:

- 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

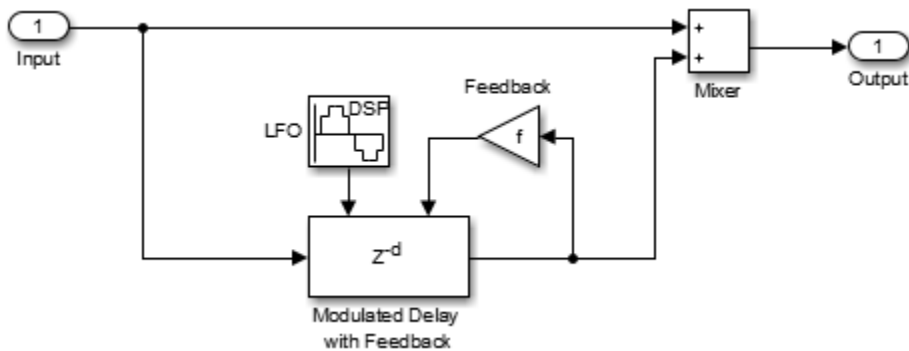
You can try out `audioexample.Chorus` by running `audioDelayEffectsExampleApp` with `'chorus'` as input. The example reads an audio signal from a file, applies the chorus effect, then plays the processed signal through your audio output device. It also launches a UI that allows you to tune the parameters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

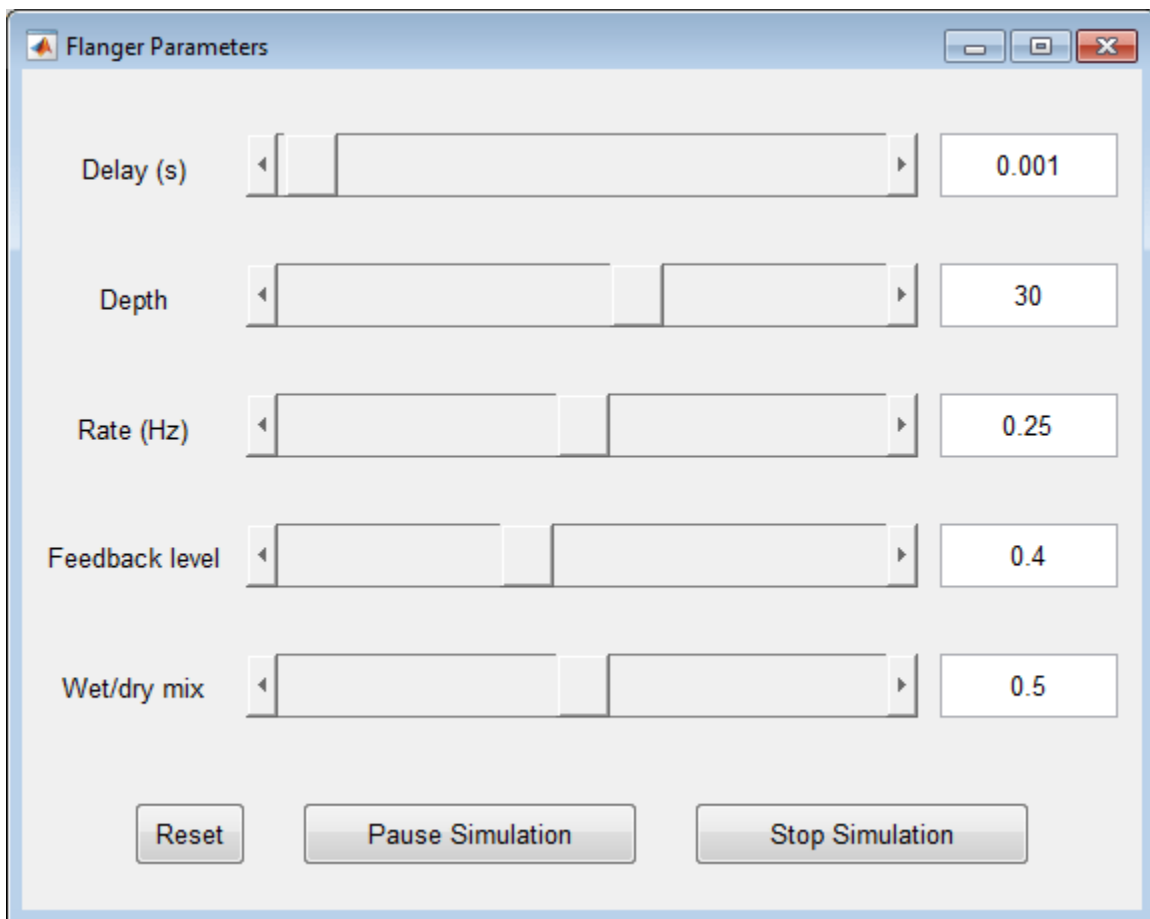


The flanger effect example has five tunable parameters that can be modified while the simulation is running:

- 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);
```

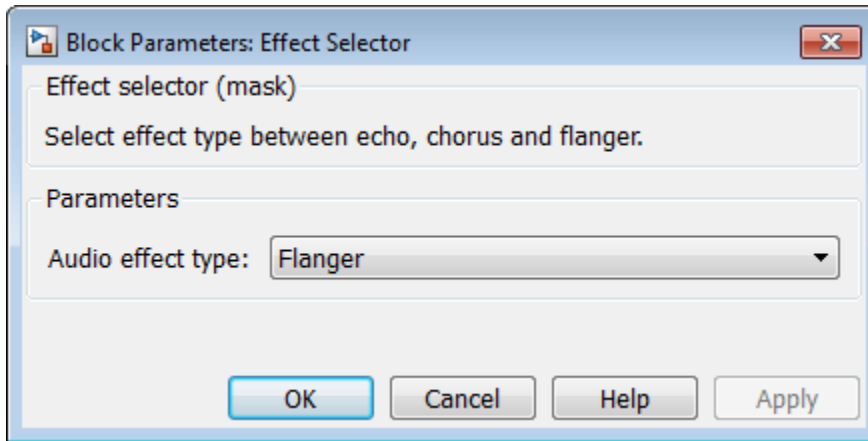


Audio Effects in Simulink

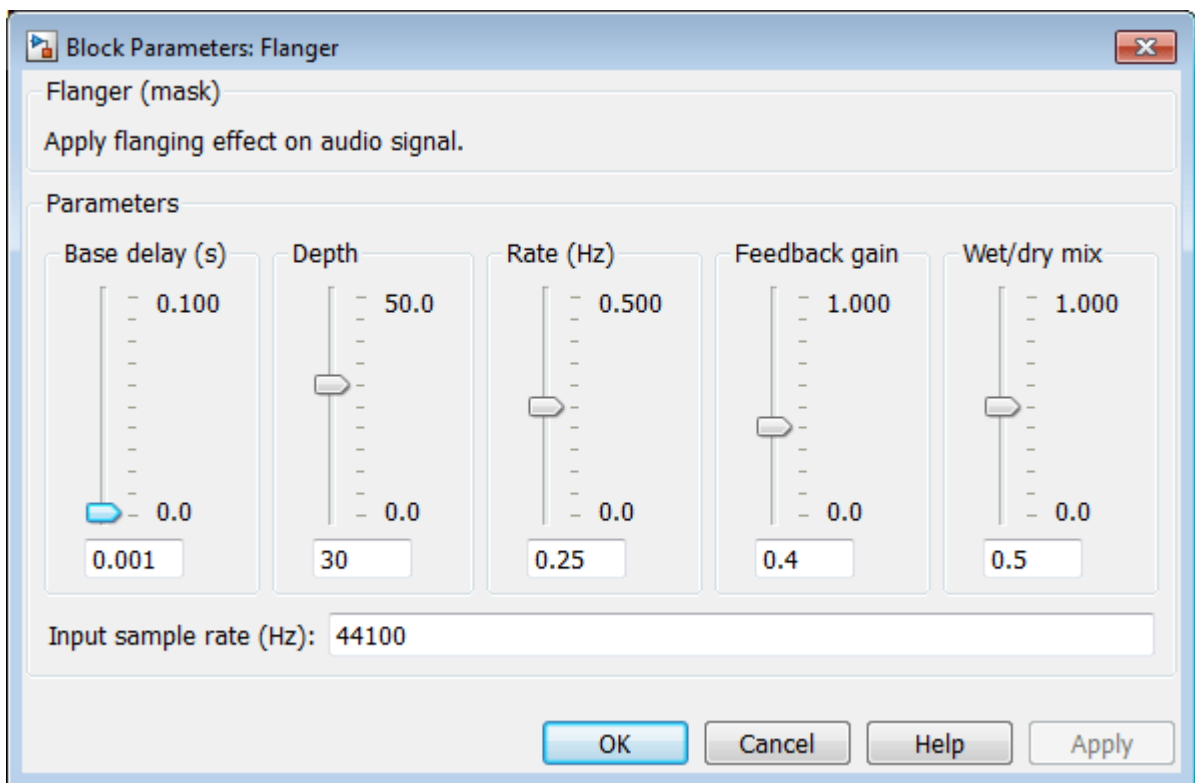
You can use the System objects `audioexample.Echo`, `audioexample.Chorus` and `audioexample.Flanger` in Simulink by using the MATLAB System (Simulink) block. The model `audiodelaybasedeffects` has these effects ready for simulation.

```
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.

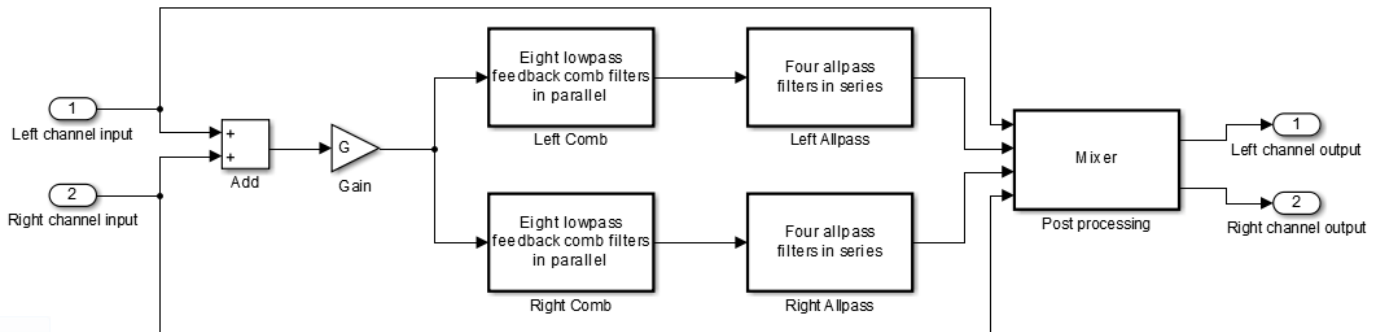
Add Reverberation Using Freeverb Algorithm

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

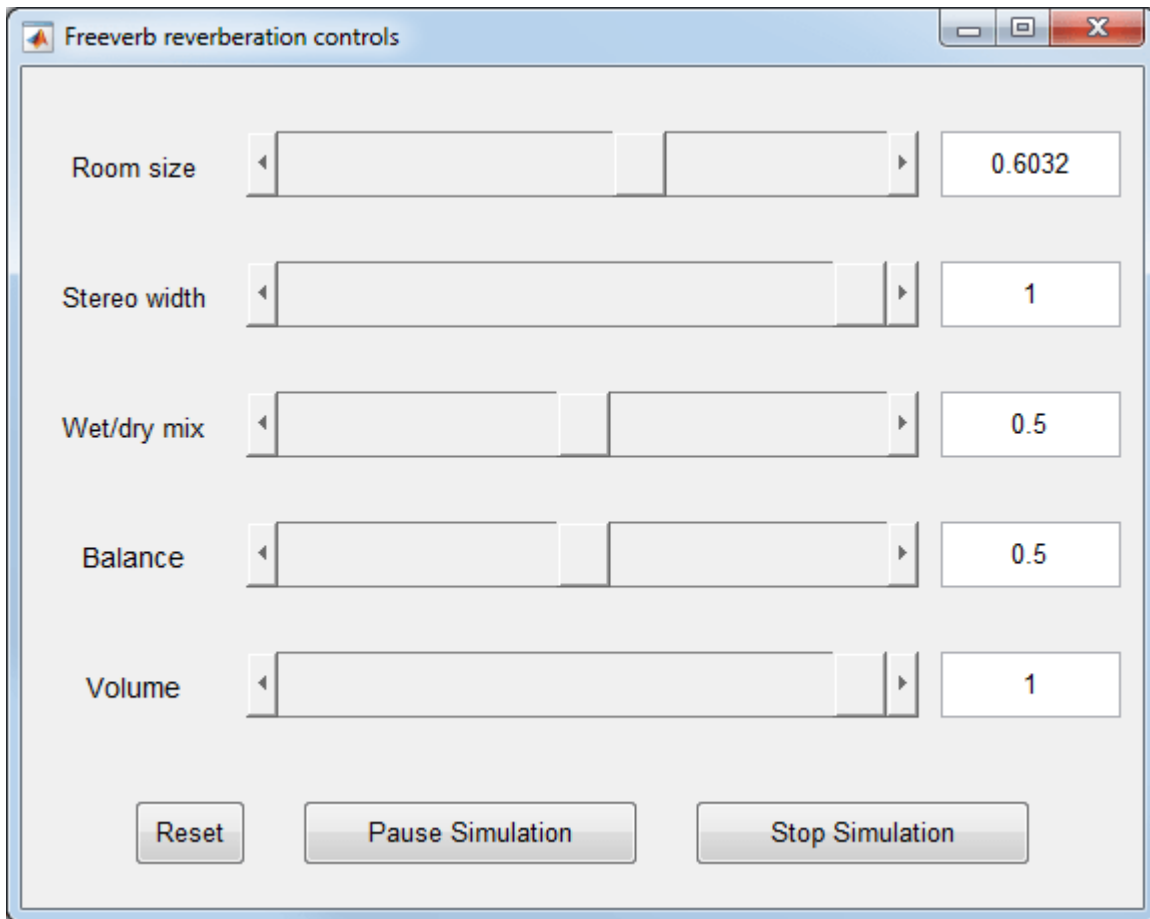
To use the reverberator on an audio signal, run `audioFreeverbReverberationExampleApp`.

```
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 1-251.

Using a Generated MEX File

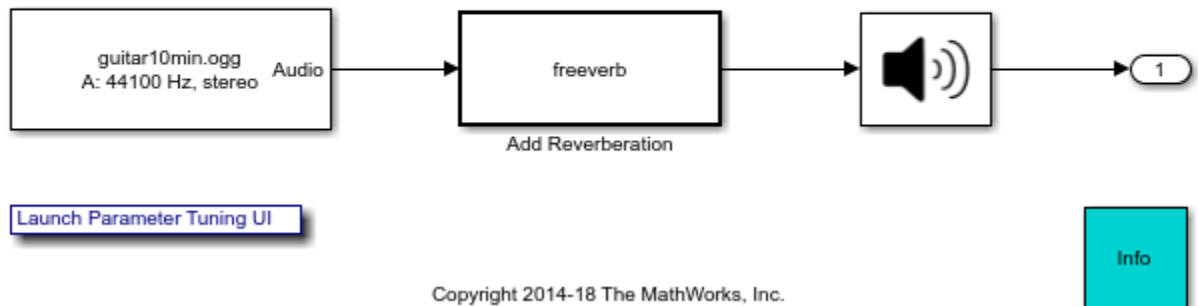
Using MATLAB Coder™, you can generate a MEX file for the main processing algorithm by executing the `HelperFreeverbCodeGeneration` command. You can use the generated MEX file by executing the `audioFreeverbReverberationExampleApp` command with `true` as an argument.

```
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 production 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 $6*N$ 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 a 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;

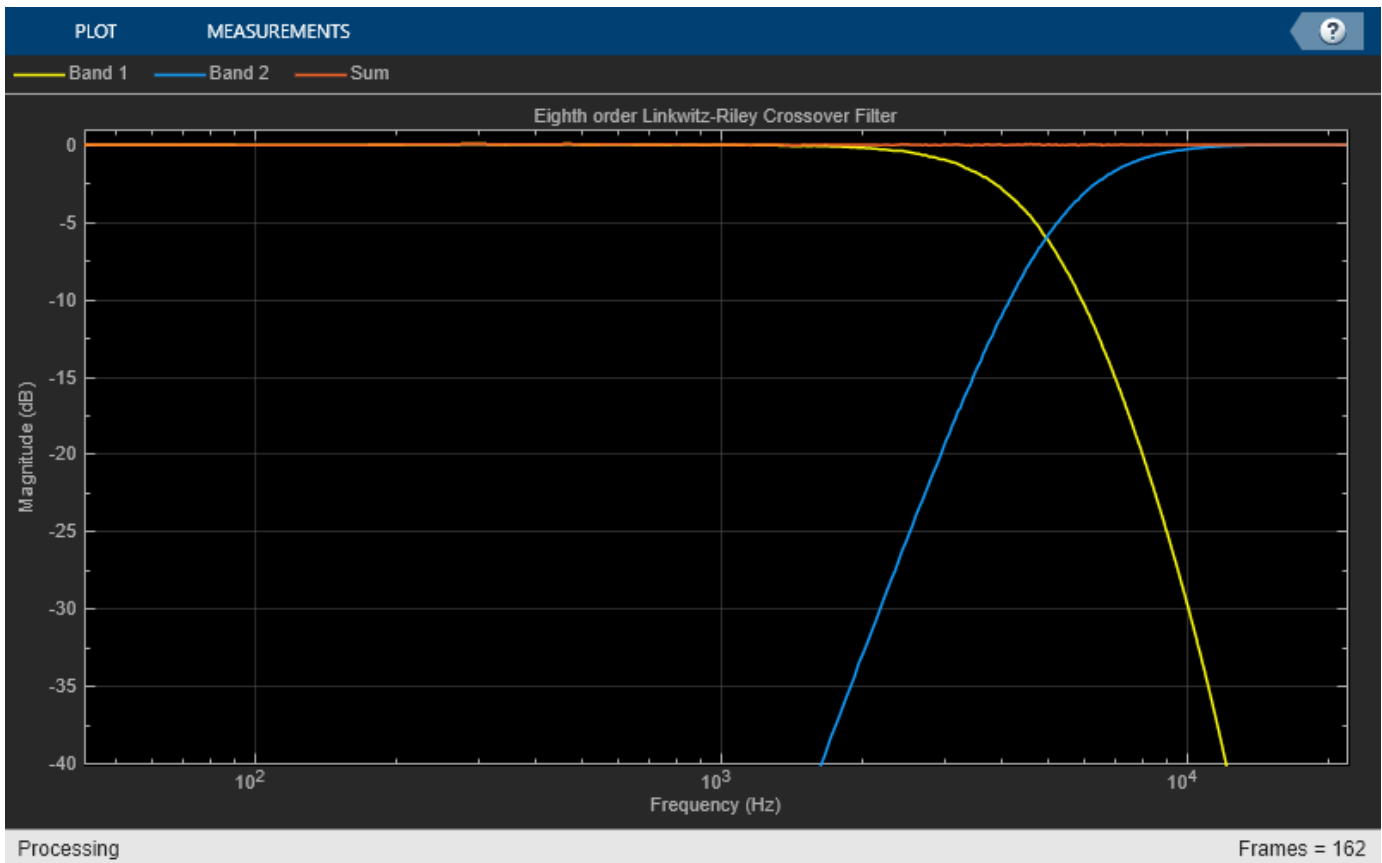
scope = dsp.ArrayPlot( ...
    'PlotType', 'Line', ...
```

```

'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

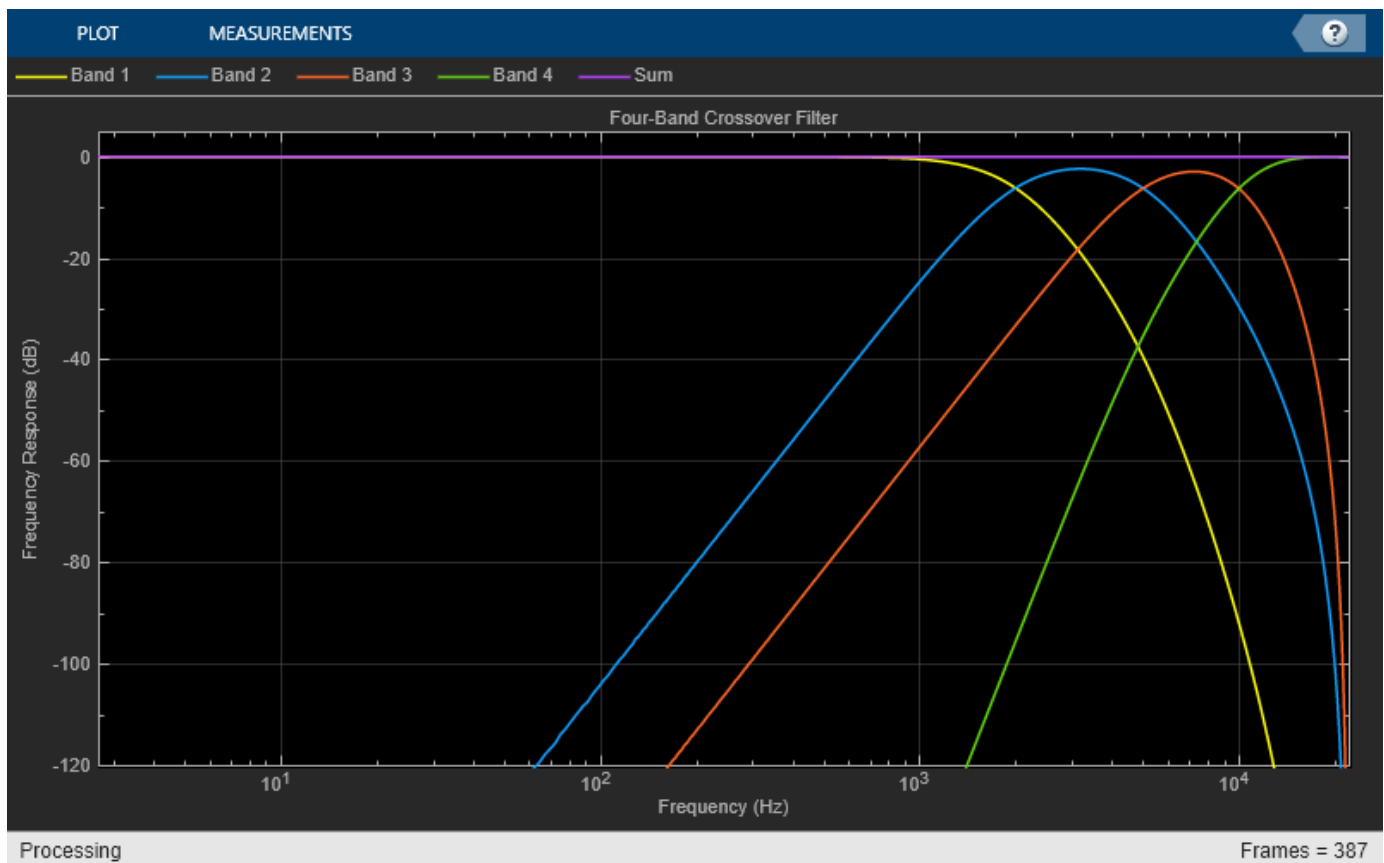
`crossoverFilter` may also be used to implement a multiband crossover filter by combining two-band crossover filters and allpass filters in a tree-like structure. The filter divides the spectrum into multiple bands such that their sum is a perfect allpass filter.

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
    [y1p,yb1,yb2,yhp] = crossover(in);
    y = y1p + yb1 + yb2 + yhp;
    z = transferFuncEstimator( repmat(in,1,5), [y1p,yb1,yb2,yhp,y] );
    scope(20*log10(abs(z)))
end

```

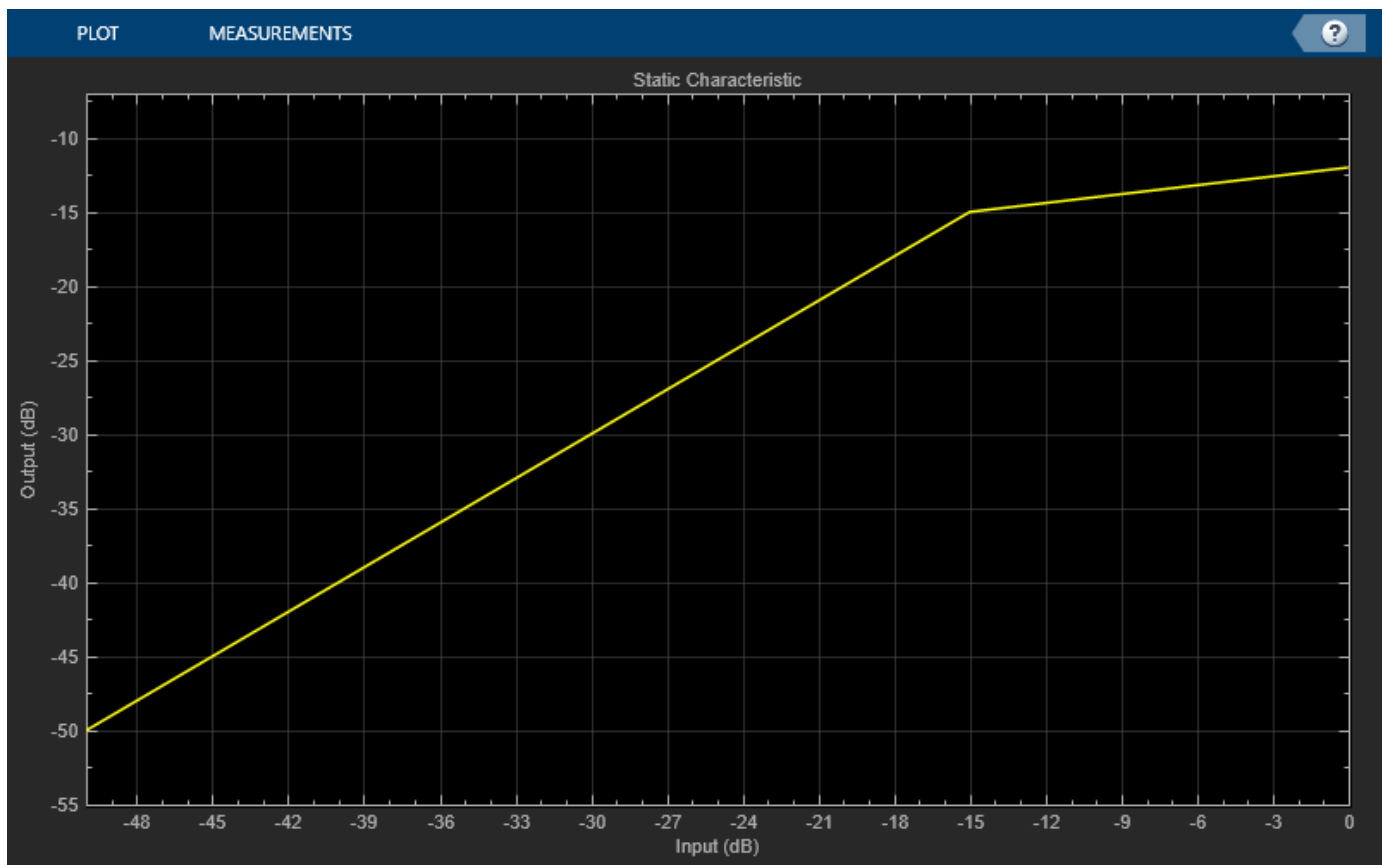


Dynamic Range Compression

`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');

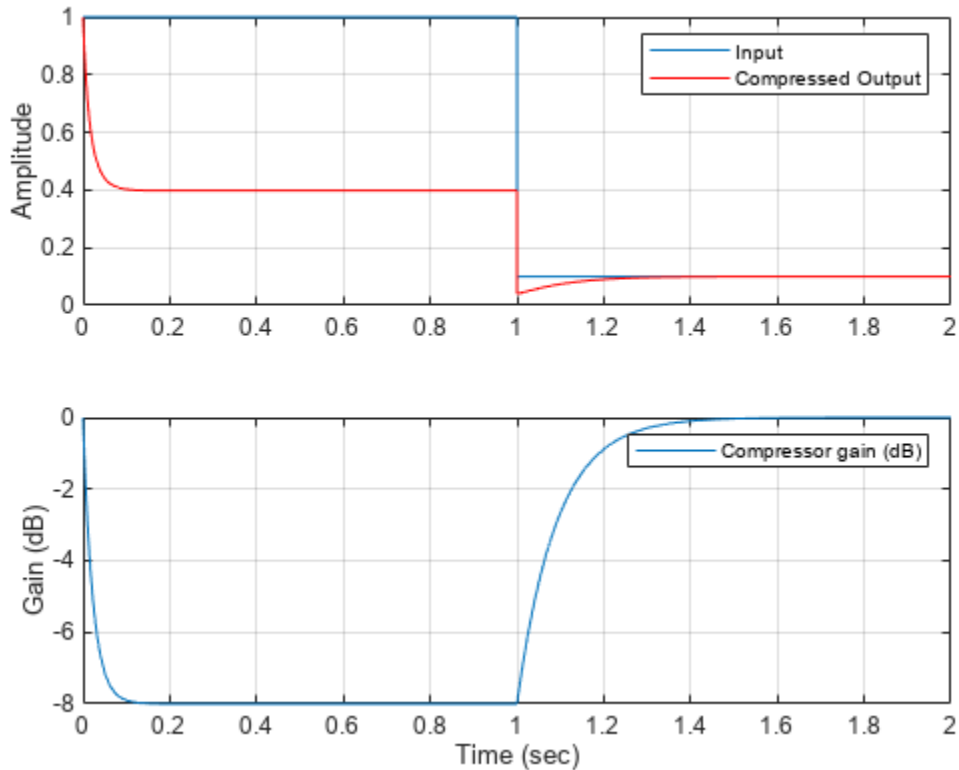
x = [ones(Fs,1);0.1*ones(Fs,1)];
[y,g] = drc(x);

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:

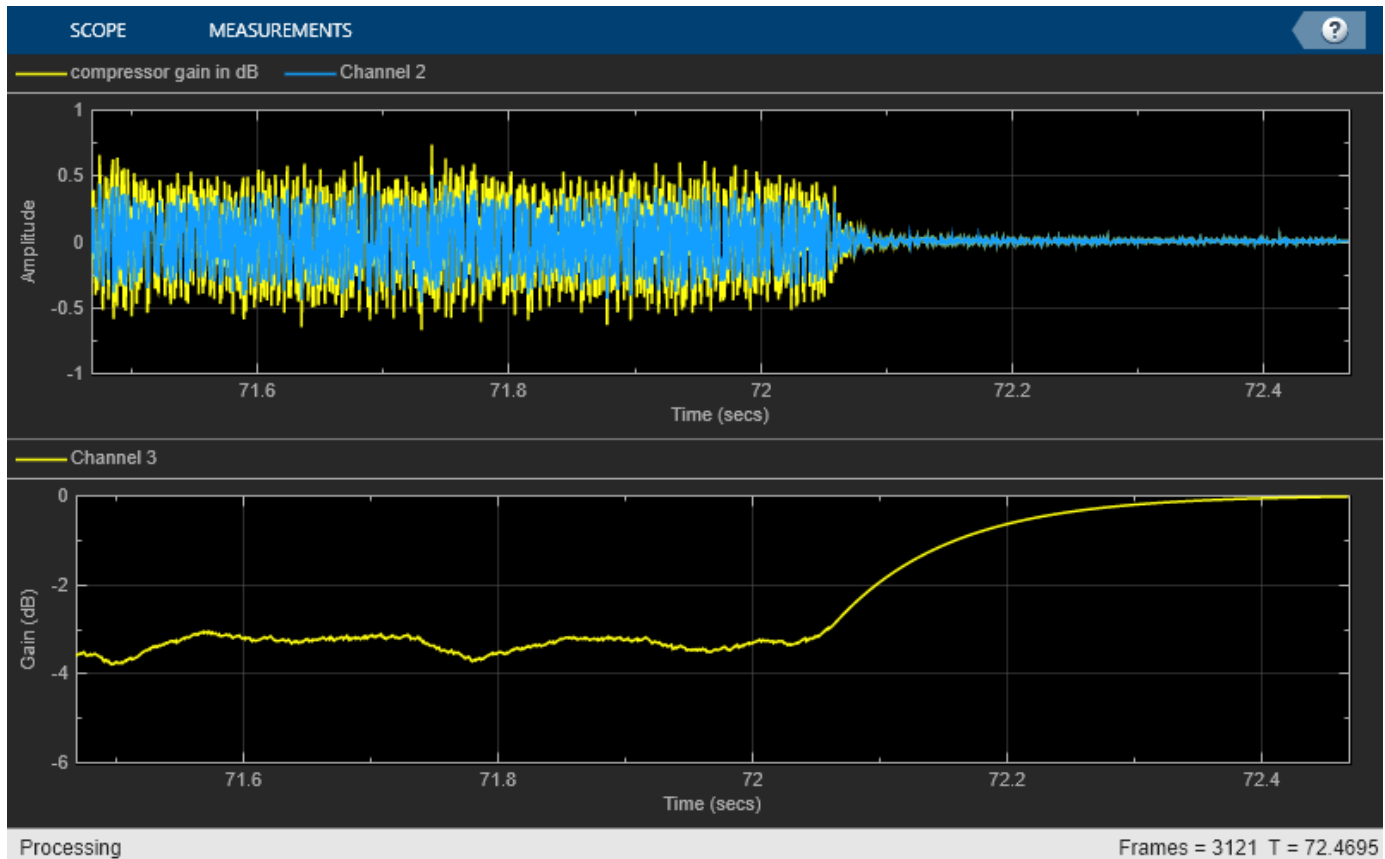
```
[~,t1] = min(abs(g(Fs:end) + 0.9 * 8));
[~,t2] = min(abs(g(Fs:end) + 0.1 * 8));
tRelease = (t2 - t1) / Fs;
fprintf('Release time is %d s\n',tRelease)
```

```
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.

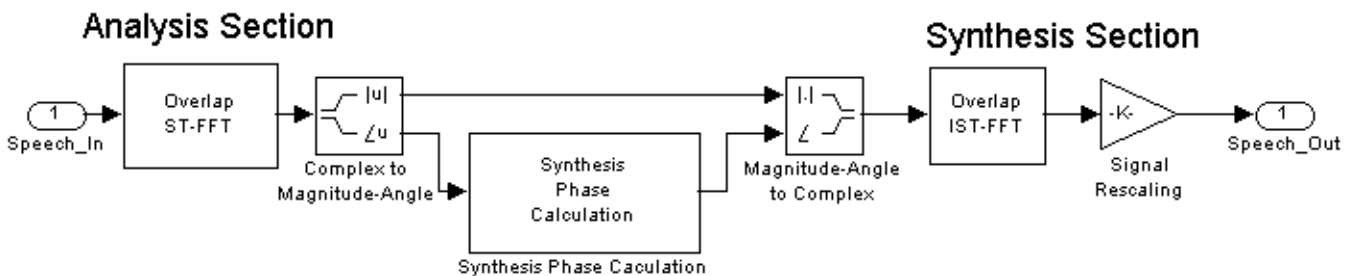
Call `multibandAudioCompressionExampleApp(true)` to use the MEX-file for simulation. Again, the simulation runs till the user explicitly stops it from the UI.

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

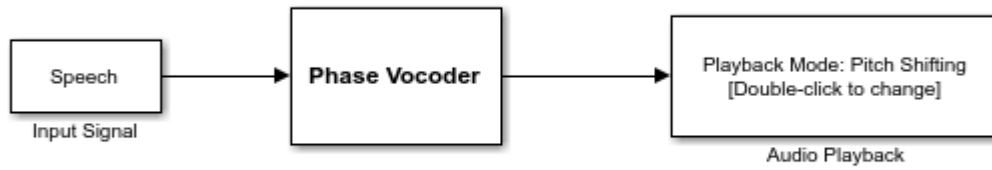
On running the model, the pitch-scaled signal is automatically played once the simulation has finished. The Audio Playback block allows you to choose between **Pitch Shifting** and **Time Dilation** modes.

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.

Phase Vocoder for Pitch Shifting and Time Dilation



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

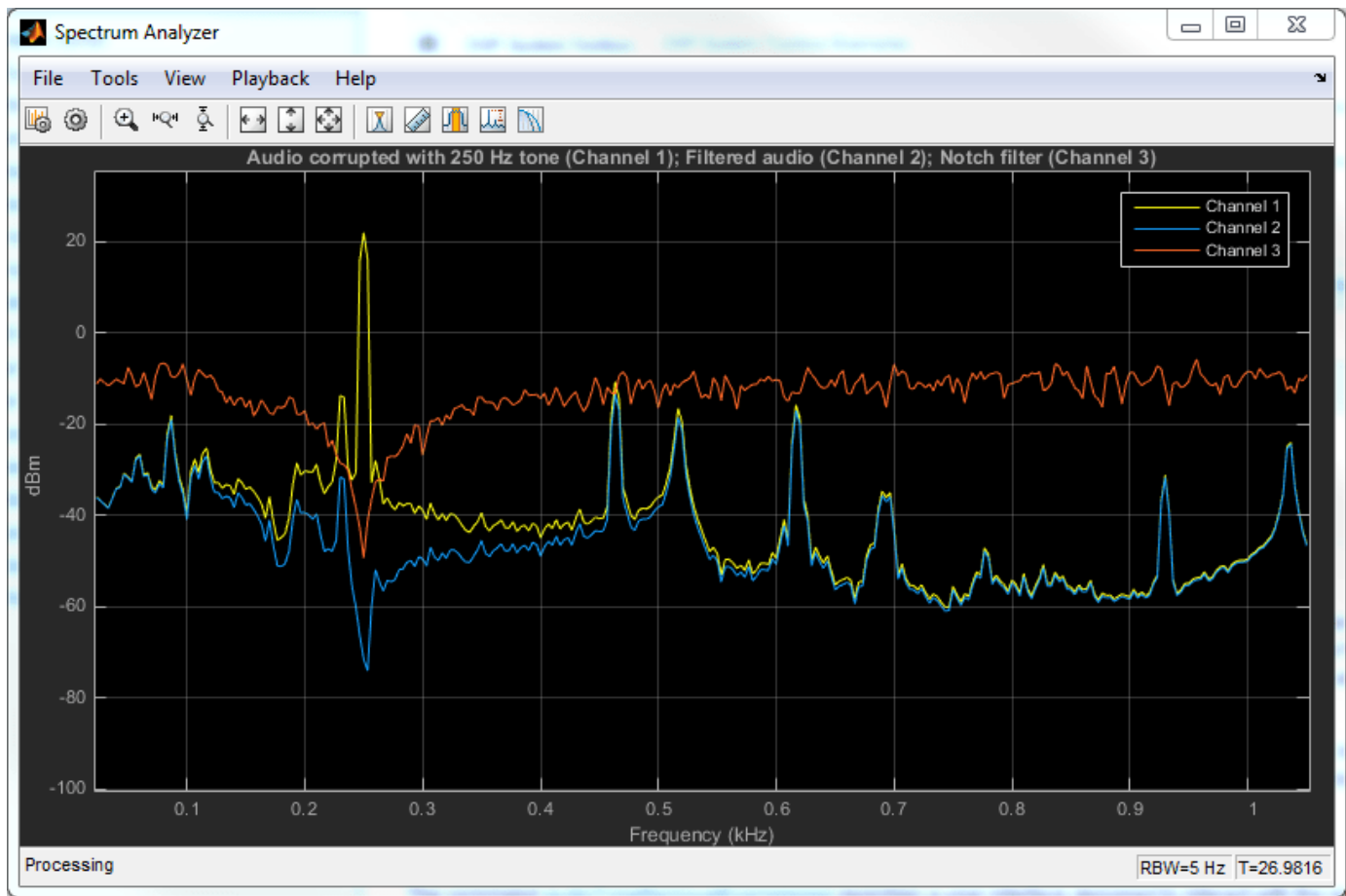
A notch filter is used to eliminate a specific frequency from a given signal. In their most common form, the filter design parameters for notch filters are center frequency for the notch and the 3 dB bandwidth. The center frequency is the frequency point at which the filter has a gain of zero. The 3 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.

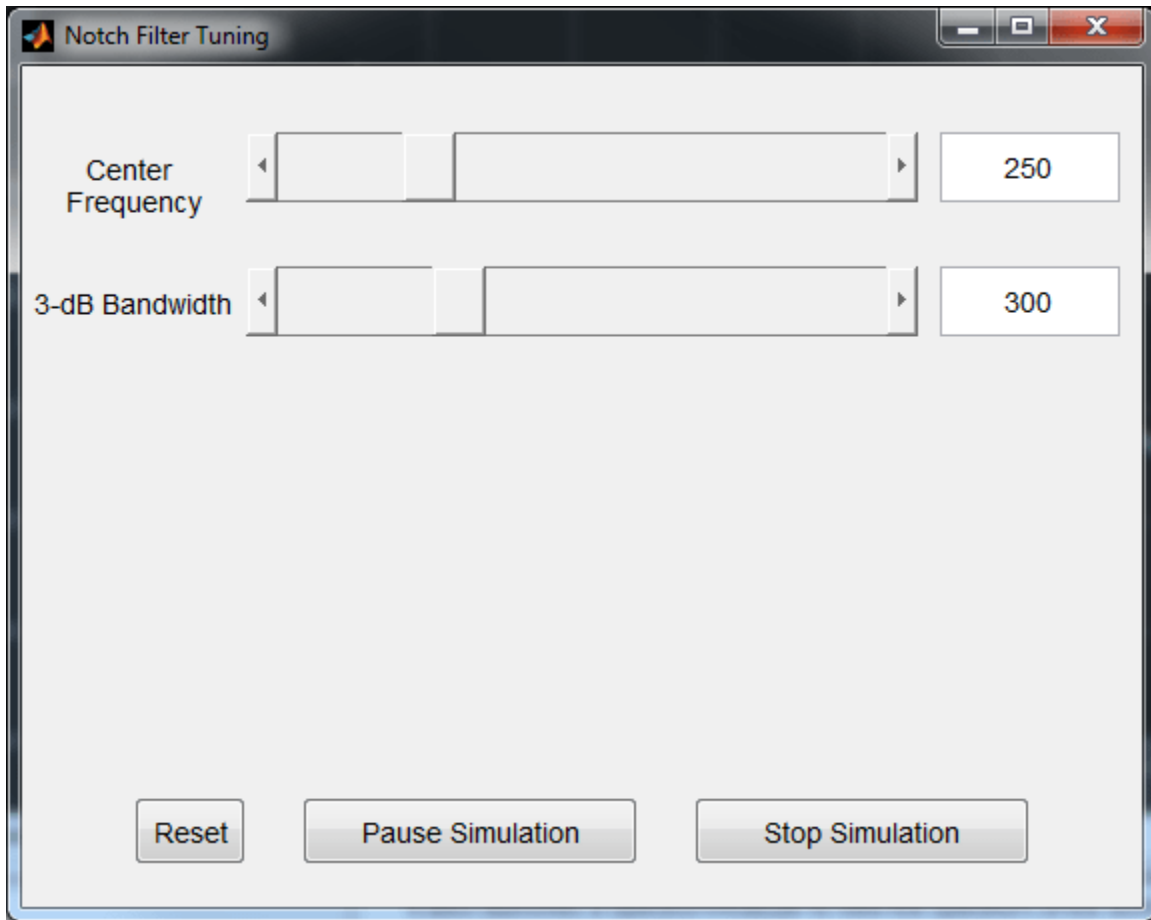
Example Architecture

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.

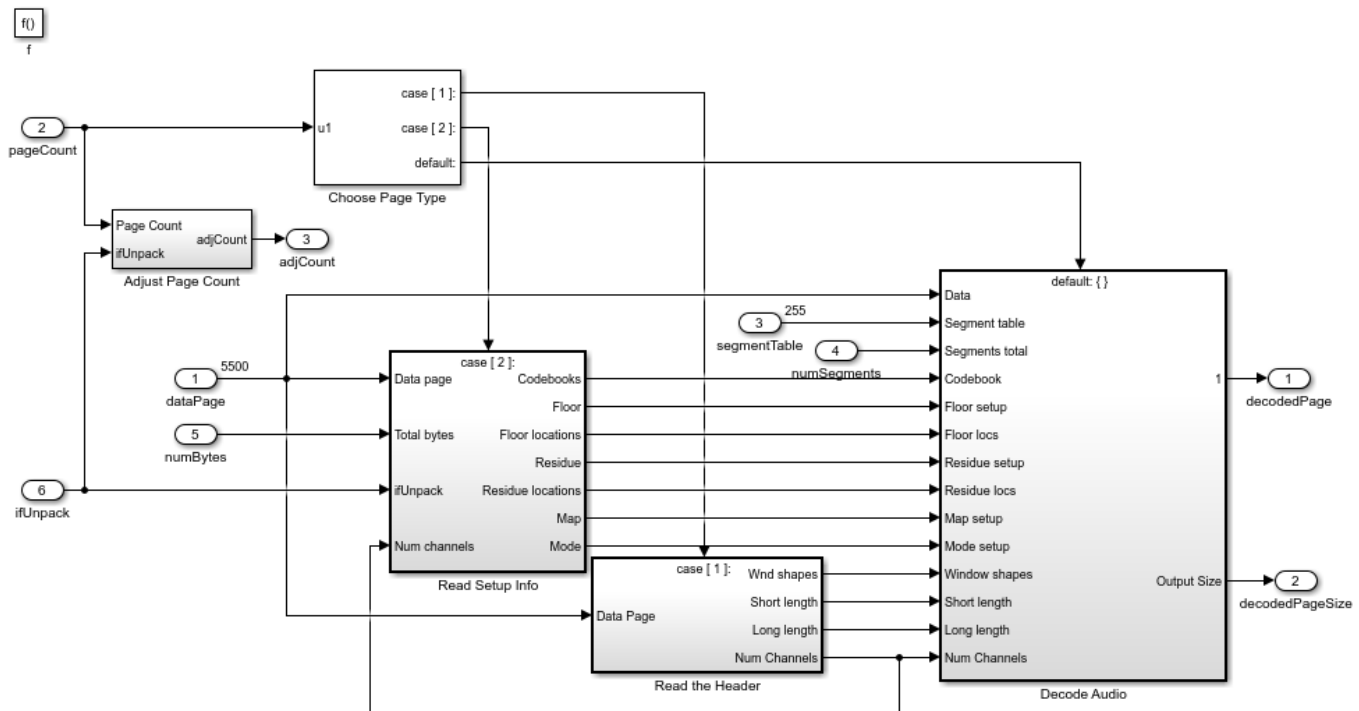


Using a Generated MEX File

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.



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.

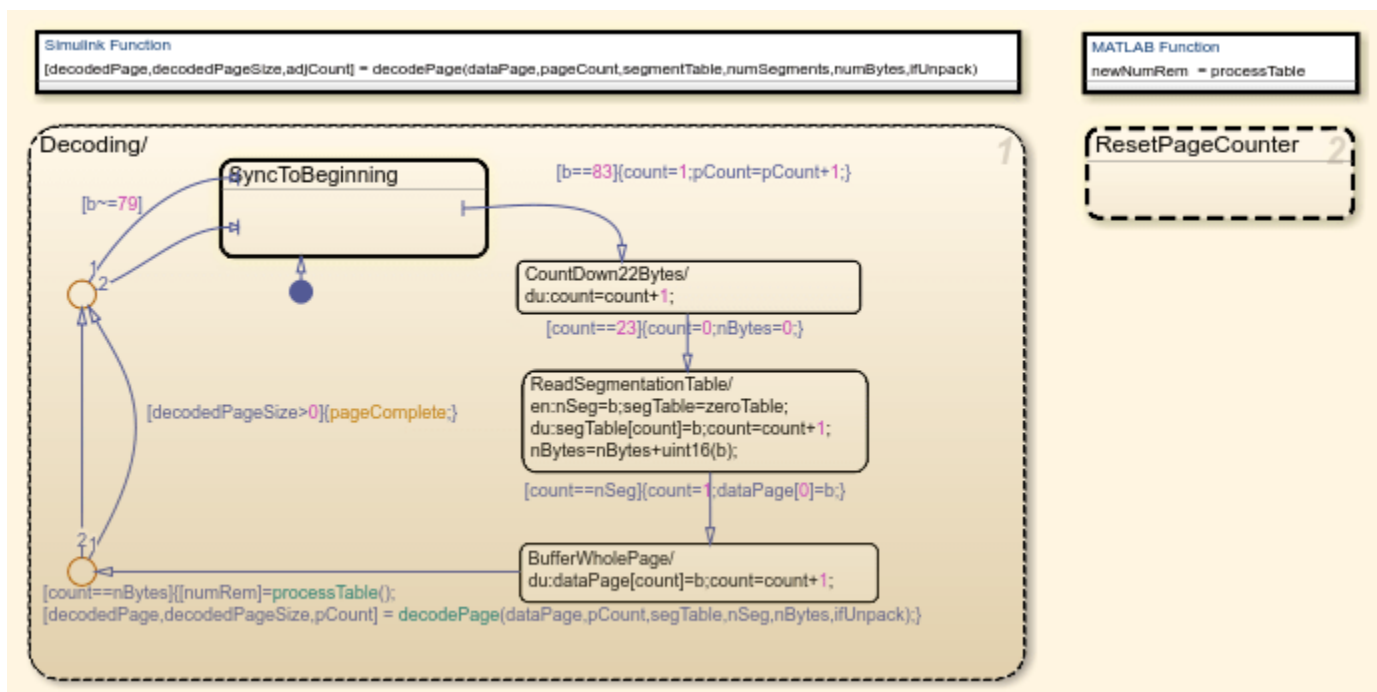
Vorbis provides its own format, known as 'Ogg', to encapsulate logical packets into transport streams. The Ogg format provides mechanisms such as framing, synchronization, positioning, and error correction, which are necessary for data transfer over networks.

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'.

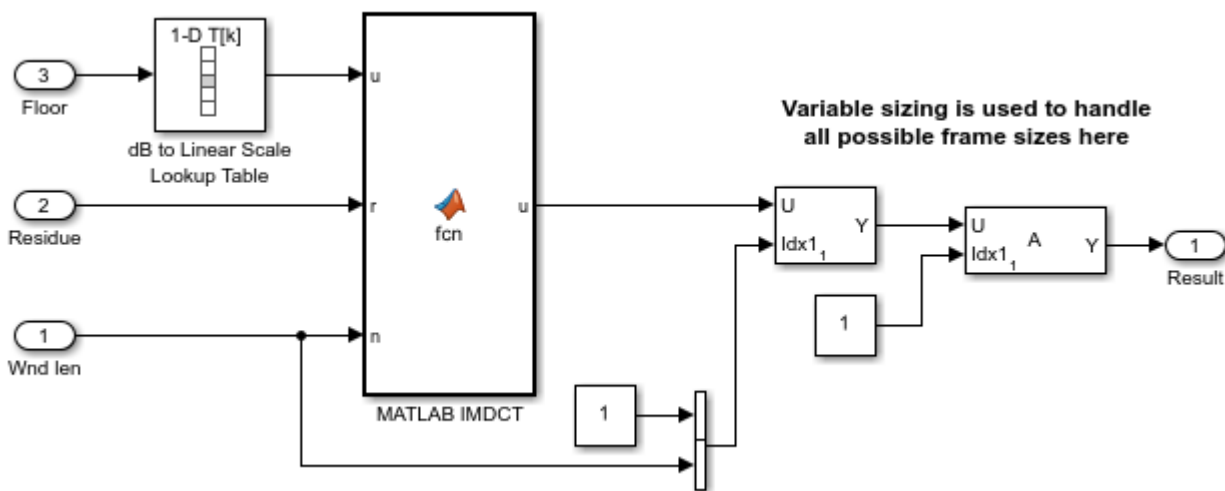


Initially, the chart tries to detect the 'OggS' synchronization pattern and then follow the decoding steps described above. Decoding the page is done with the Simulink function 'decodePage' and then the model immediately goes back to detecting the next 'OggS' sequence. The state 'ResetPageCounter' is 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.

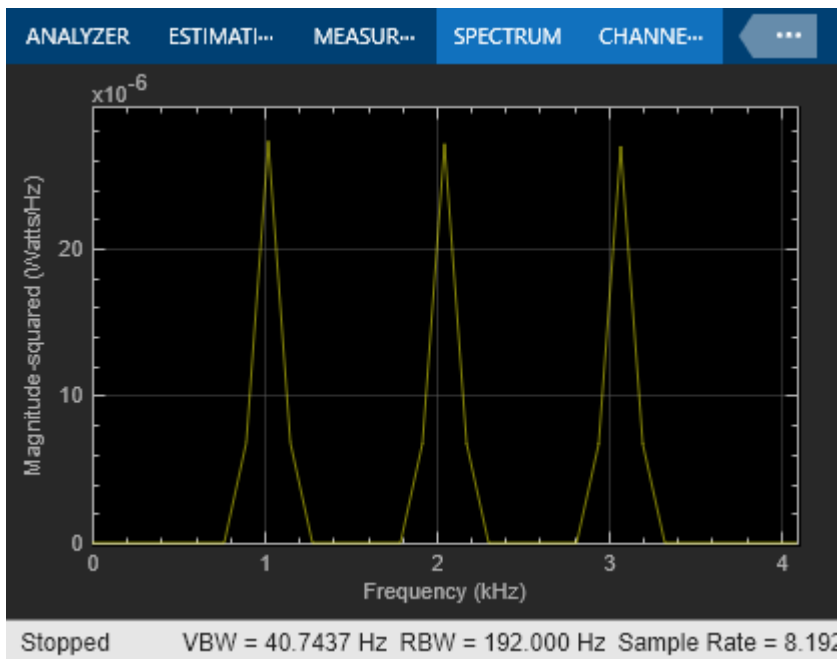
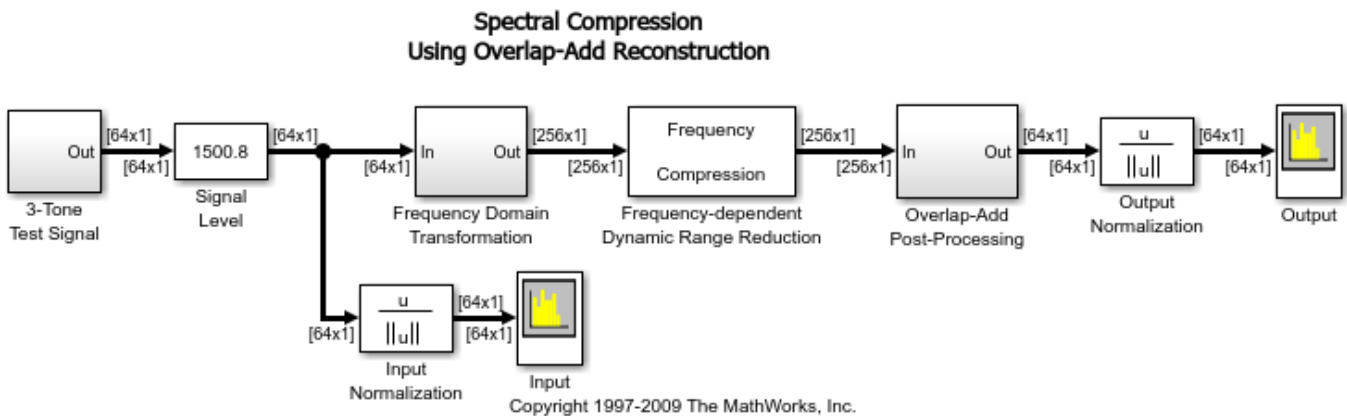
The output block in the top level of the model feeds the output of the decoding block to the audio playback device on your system. The valid portion of the decoded signal is input to the Audio Device Writer block.

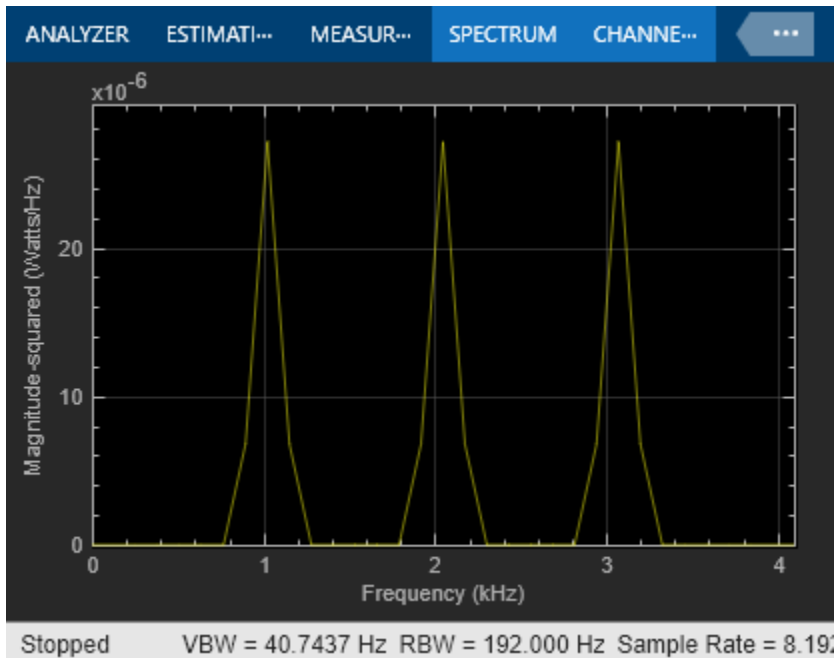
References

[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.

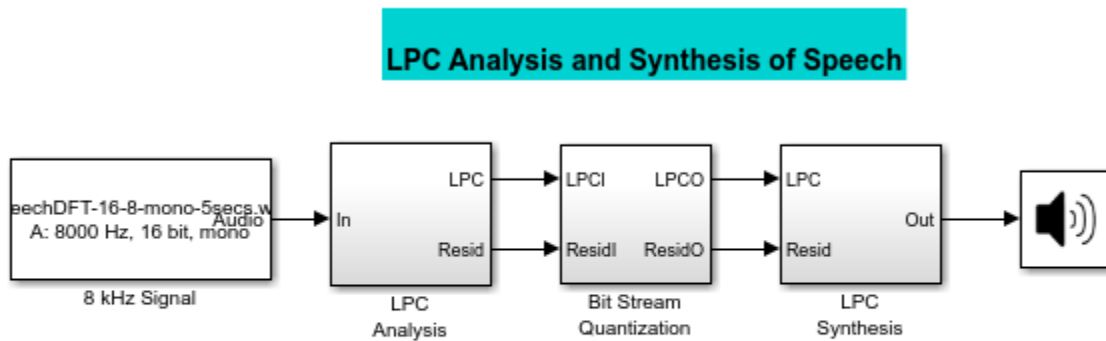
Compression maps the dynamic range of the magnitude at each frequency bin from the range 0 to 100 dB to the range y_{min} to y_{max} dB. y_{min} and y_{max} are vectors in the MATLAB® workspace with one element for each frequency bin; in this case 256. The phase is not altered. This is a non-linear spectral 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.

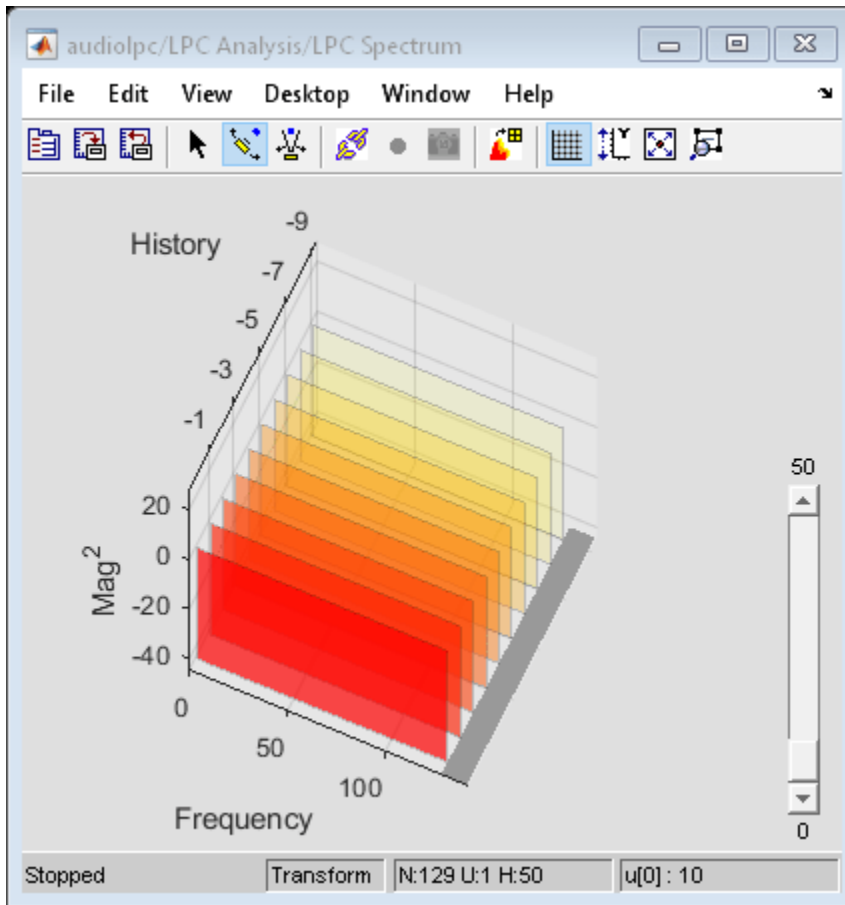
LPC Analysis and Synthesis of Speech

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.

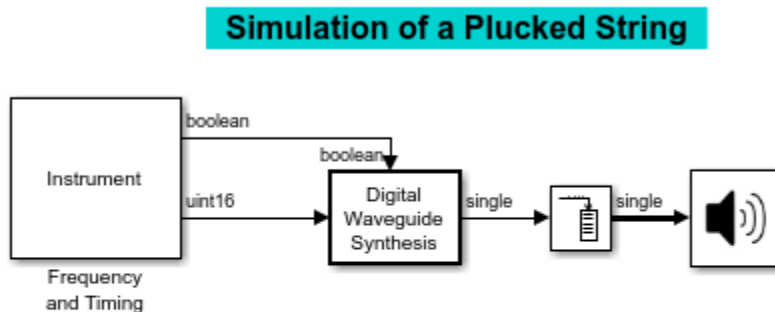
In this simulation, the speech signal is divided into 20 ms frames (160 samples), with an overlap of 10 ms (80 samples). Each frame is windowed using a Hamming window. Eleventh-order autocorrelation coefficients are found, and then the reflection coefficients are calculated from the autocorrelation coefficients 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.

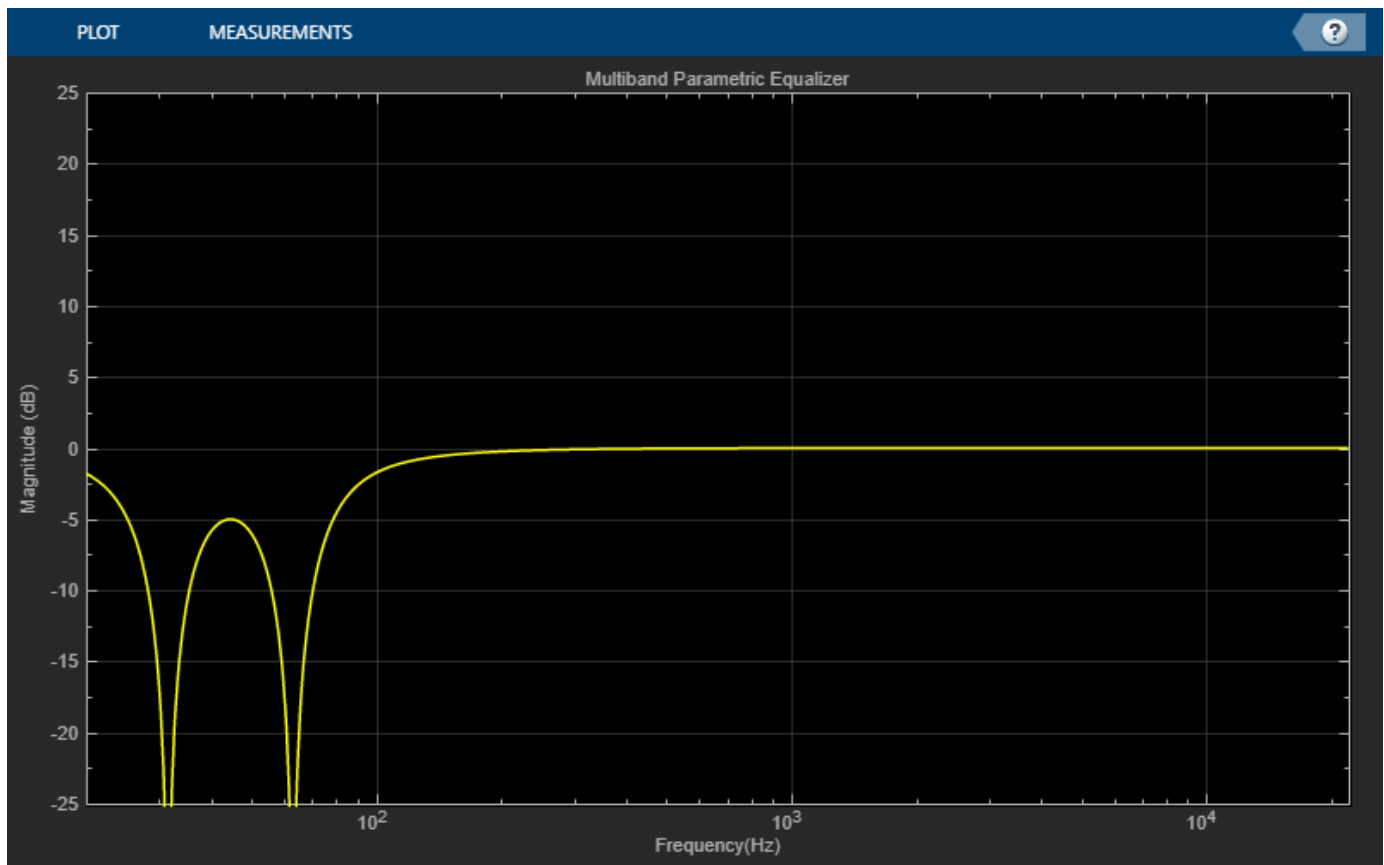
This example uses the `audiopluginexample.Phaser` audio plugin class. The plugin implements a multi-notch filter with notch frequencies modulated by an `audioOscillator`. The multi-notch filter is implemented through the `multibandParametricEQ` System object. The bands of the equalizer can be made to 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.

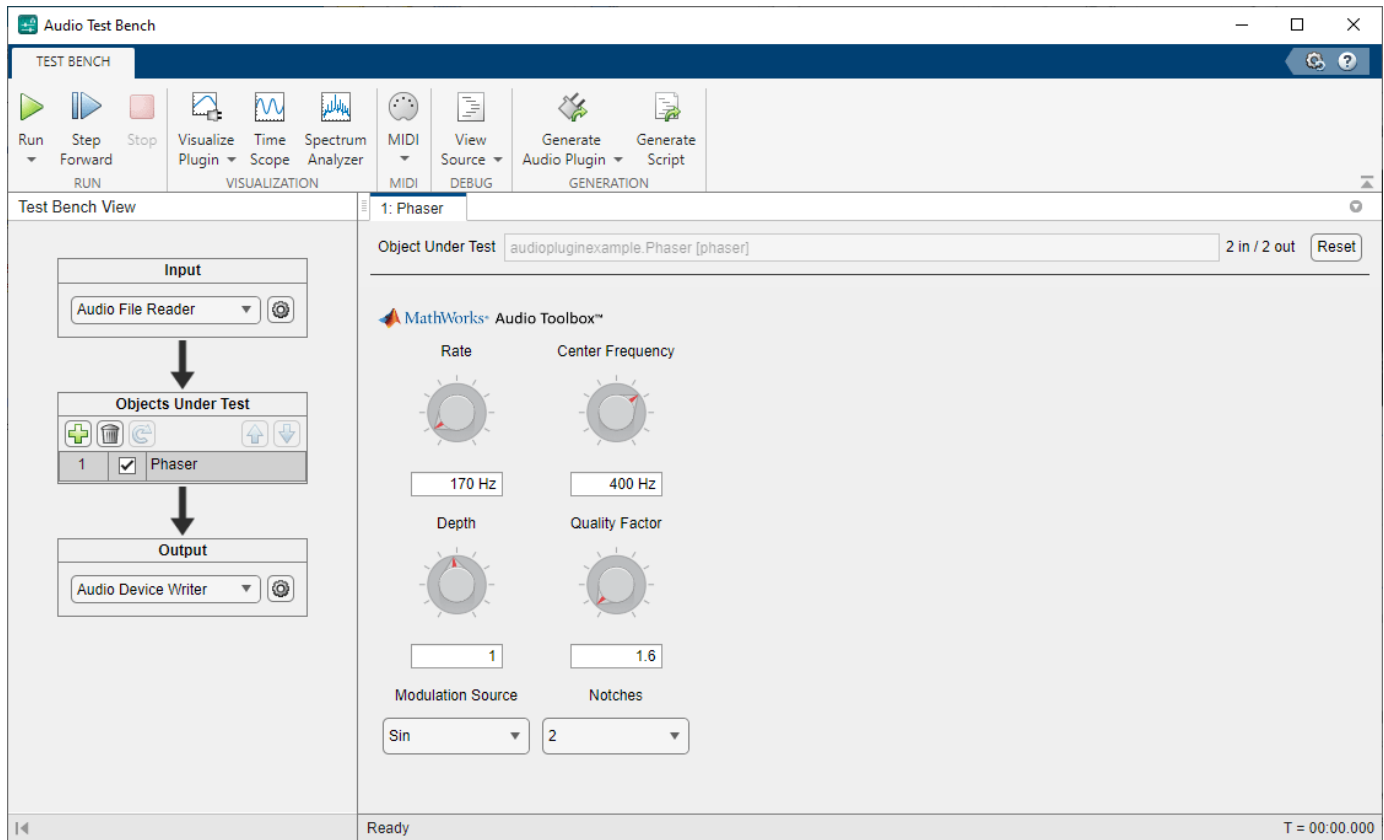
Initialize the phaser and visualize its magnitude response.

```
phaser = audiopluginexample.Phaser;  
visualize(phaser)
```

Launch the **Audio Test Bench**.

```
audioTestBench(phaser)
```



The **Audio Test Bench** enables you to tune the audio phaser using dials and drop-down menus. Changing dial or drop-down values updates the magnitude response plot of the phaser in real time.

The four dials 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.
- *Quality 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** drop-down 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 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)
```

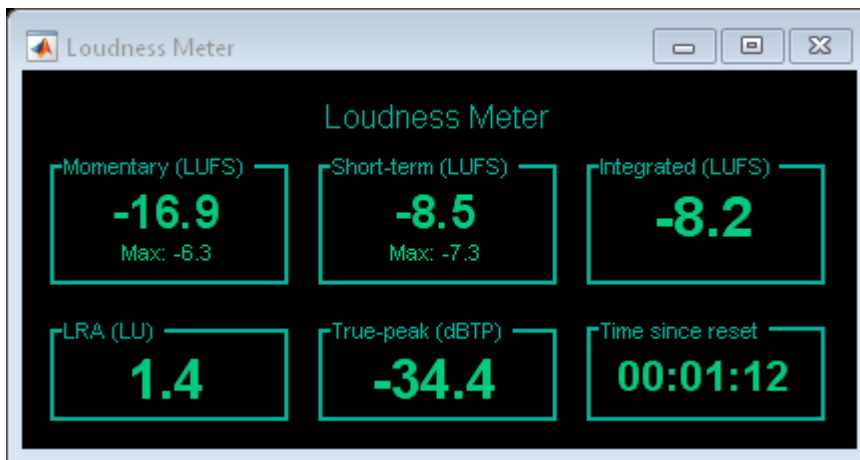
```
Loudness after normalization: -23.0 LUFS
```

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
```



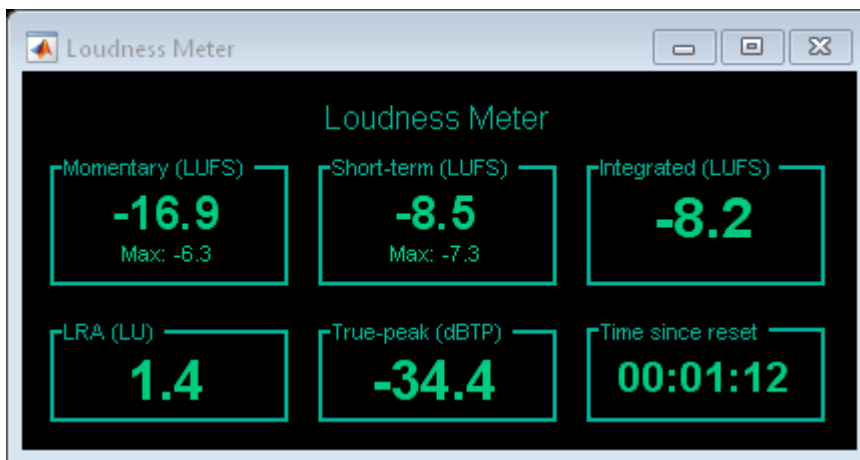
```
fprintf('Max true-peak value before normalization: %.1f dBTP\n',maxTP(end))
```

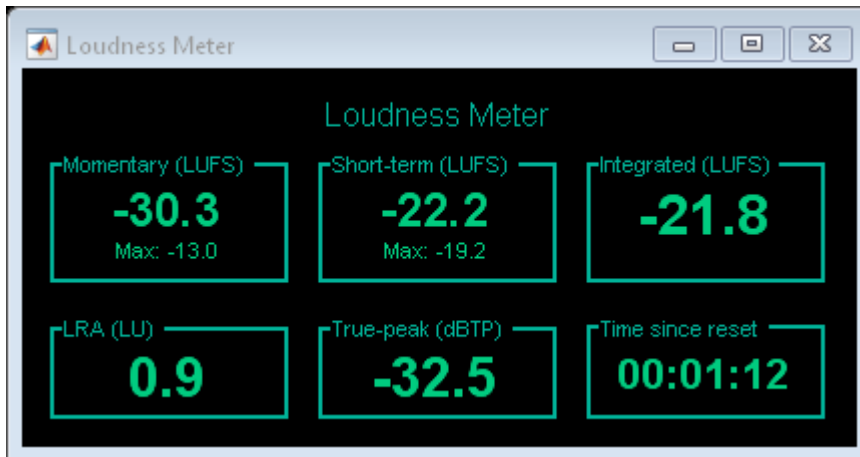
```
Max true-peak value before normalization: -0.3 dBTP
```

```
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).

Setup

Define some parameters to be used throughout the example.

```
frameSize = 64;  
inFs      = 96e3;
```

Generating the 96 kHz Signal

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.

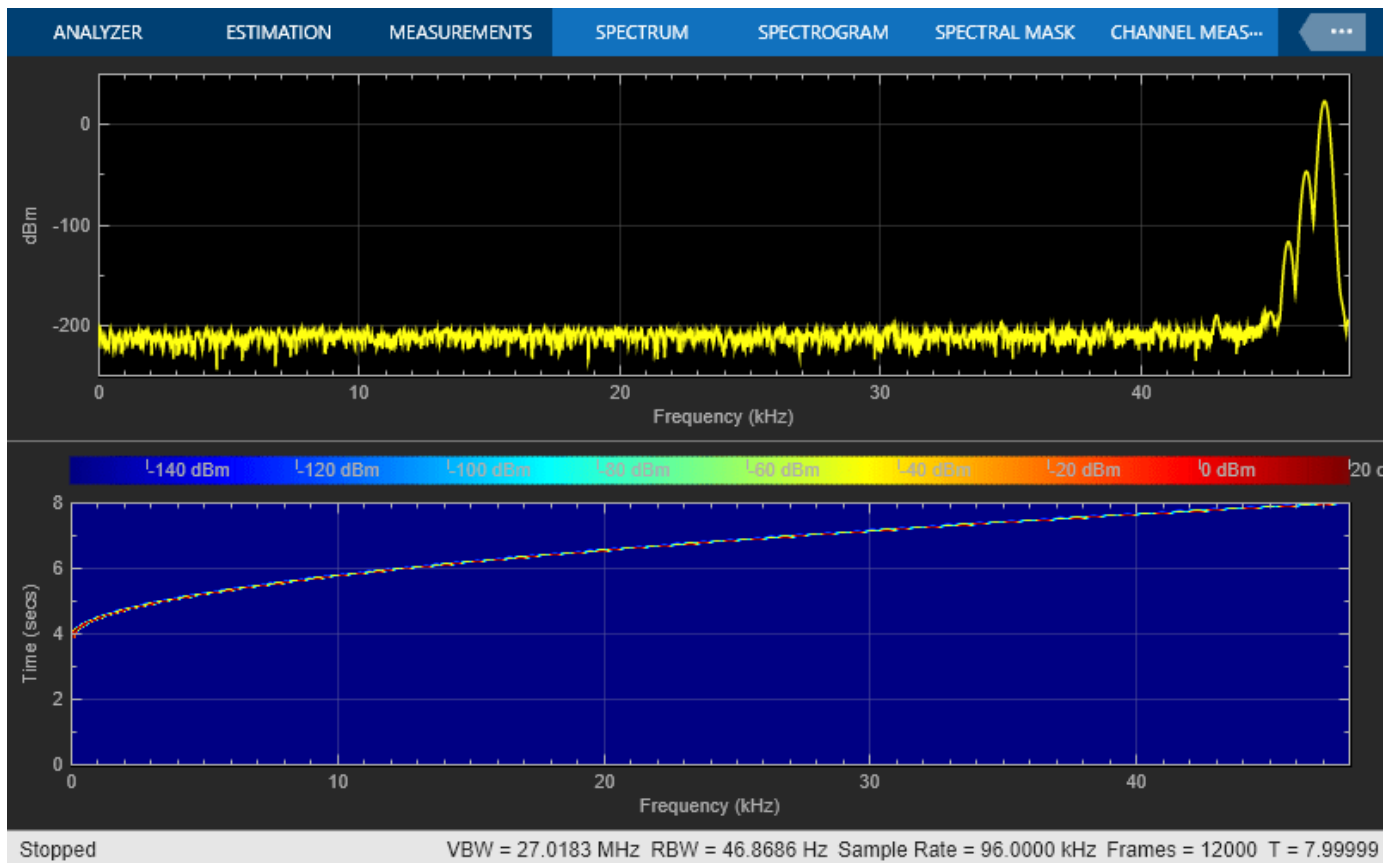
```
SpectrumAnalyzer44p1 = spectrumAnalyzer( ...  
    SampleRate=44100, ...  
    Method="welch", ...  
    AveragingMethod="exponential", ...  
    ForgettingFactor=1e-7, ...  
    ViewType="spectrum-and-spectrogram", ...  
    TimeSpanSource="property",TimeSpan=8, ...  
    Window="kaiser",SidelobeAttenuation=220, ...  
    YLimits=[-250, 50],ColorLimits=[-150, 20], ...  
    PlotAsTwoSidedSpectrum=false);
```

```
SpectrumAnalyzer96 = spectrumAnalyzer( ...  
    SampleRate=96000, ...  
    Method="welch", ...  
    AveragingMethod="exponential", ...  
    ForgettingFactor=1e-7, ...  
    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 $[-20 \text{ kHz}, 20 \text{ kHz}]$. 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);
```

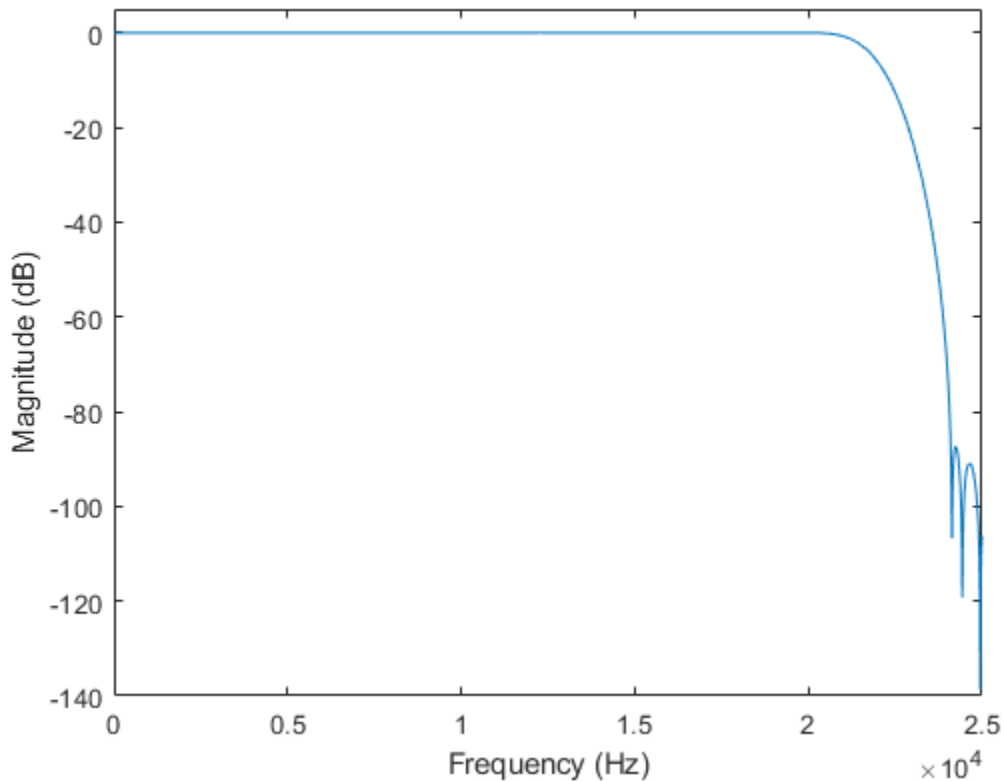
Analysis of the Filters Involved in the Conversion

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("Magnitute (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

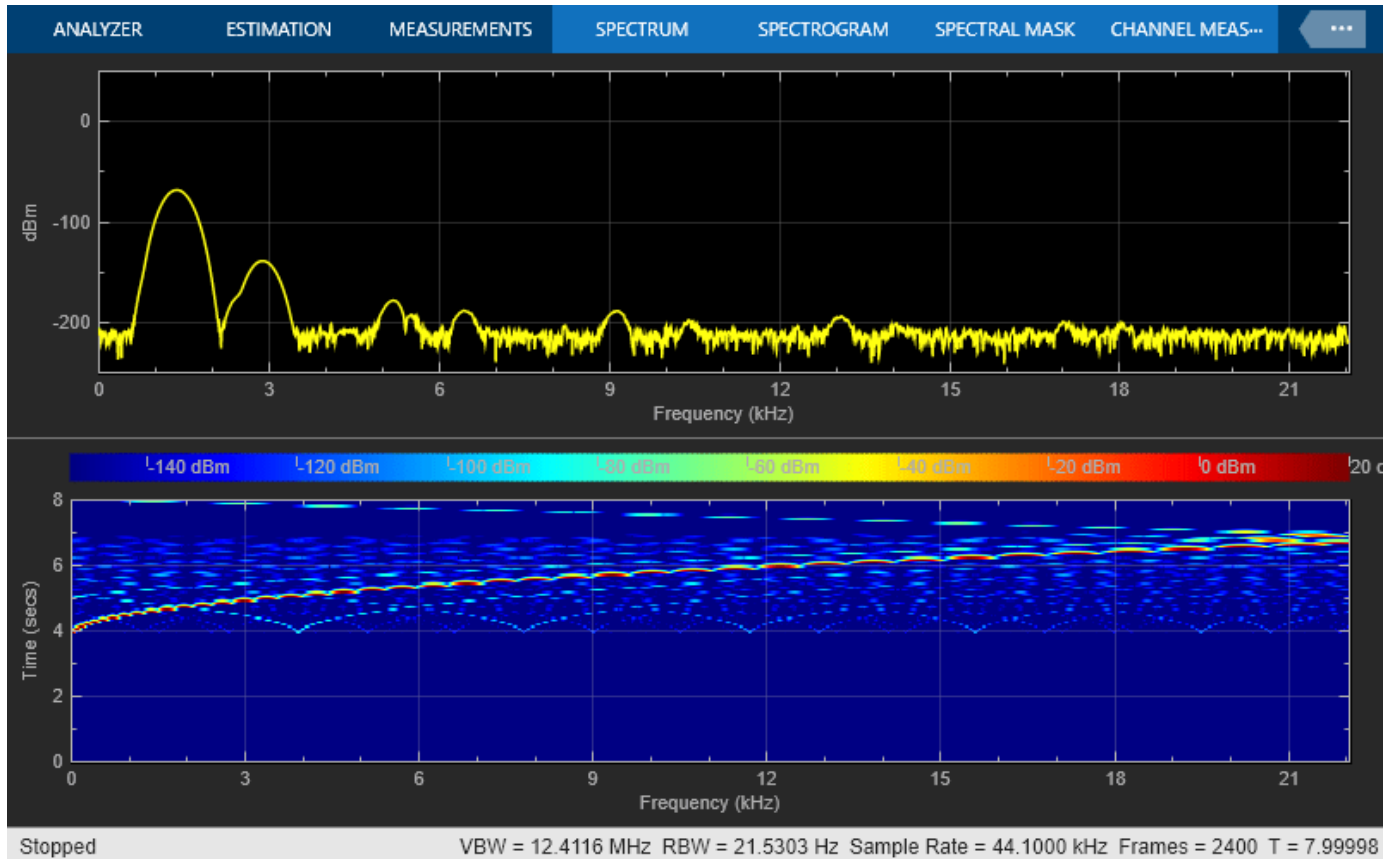
The loop below performs the sample rate conversion in streaming fashion. The computation is fast enough to operate in real time if need be.

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
```

```
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

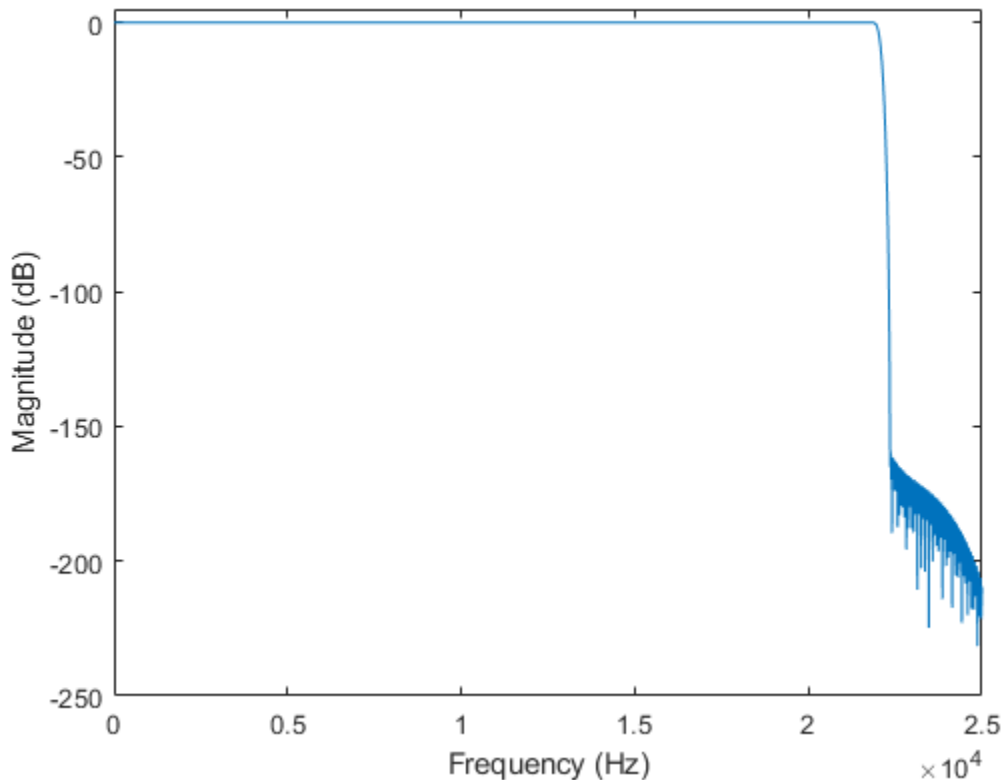
The previous sample rate converter involved 8618 filter coefficients and a computational cost of 42.3 multiplications per input sample. By increasing the bandwidth and stopband attenuation, the cost increases substantially to 123896 filter coefficients and 440.34 multiplications per input sample. The 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("Magnititude (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
```



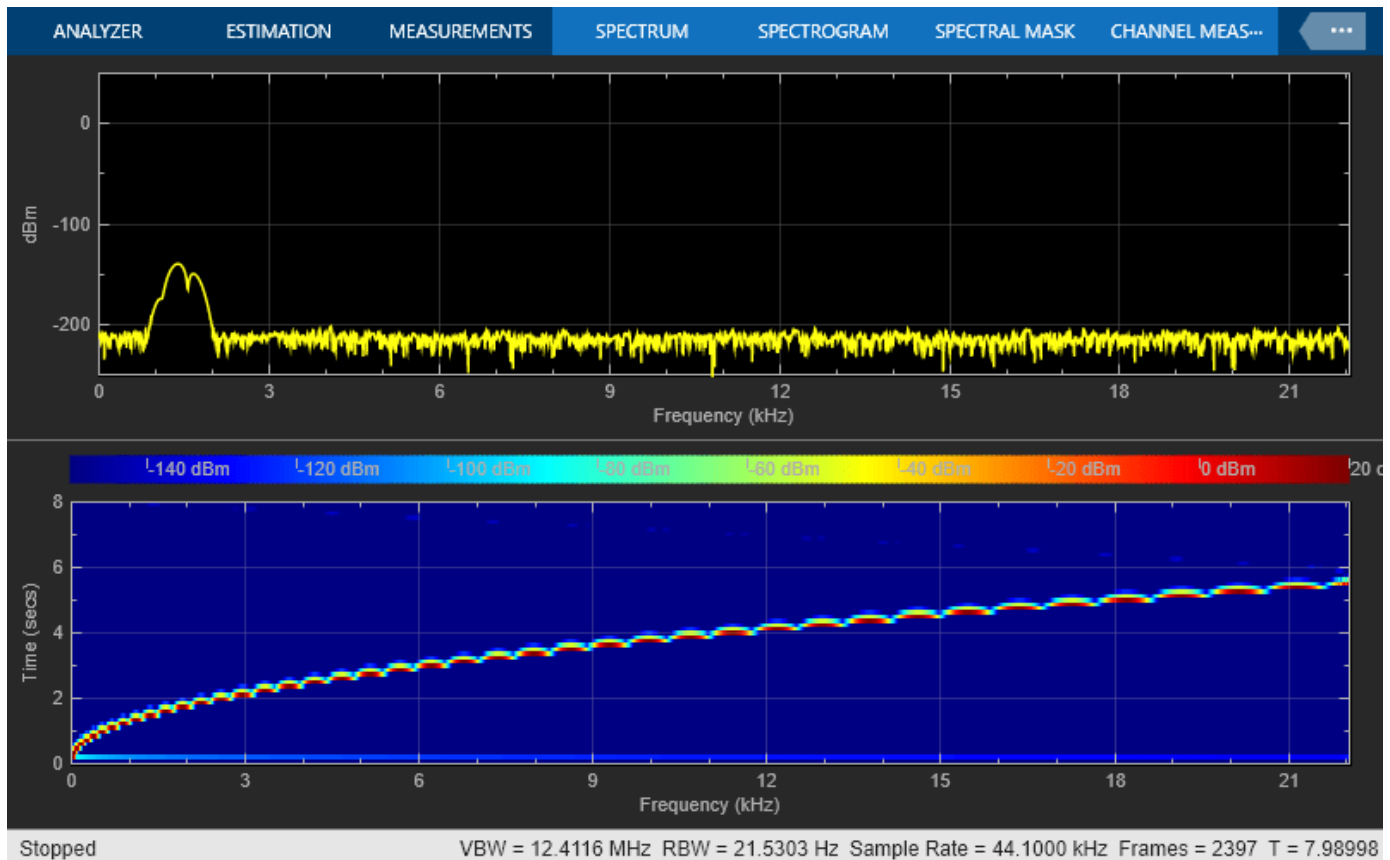
Main Processing Loop

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.

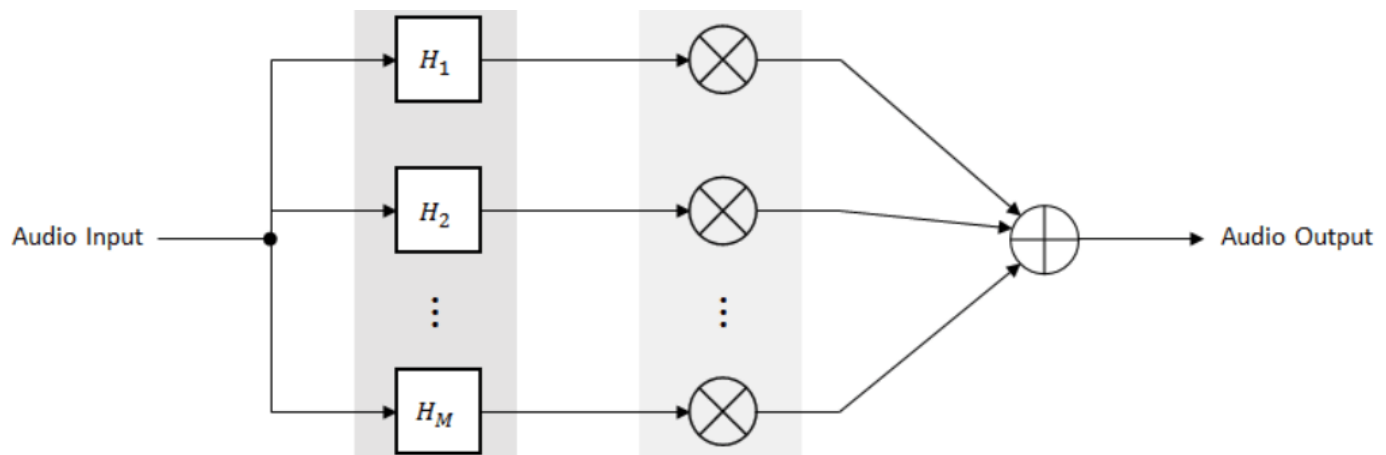
One of the more sophisticated equalization techniques is known as parametric equalization. Parametric equalizers provide control over three filter parameters: center frequency, bandwidth, and gain. Audio Toolbox™ provides the `multibandParametricEQ` System object and the Single-Band Parametric EQ block 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.

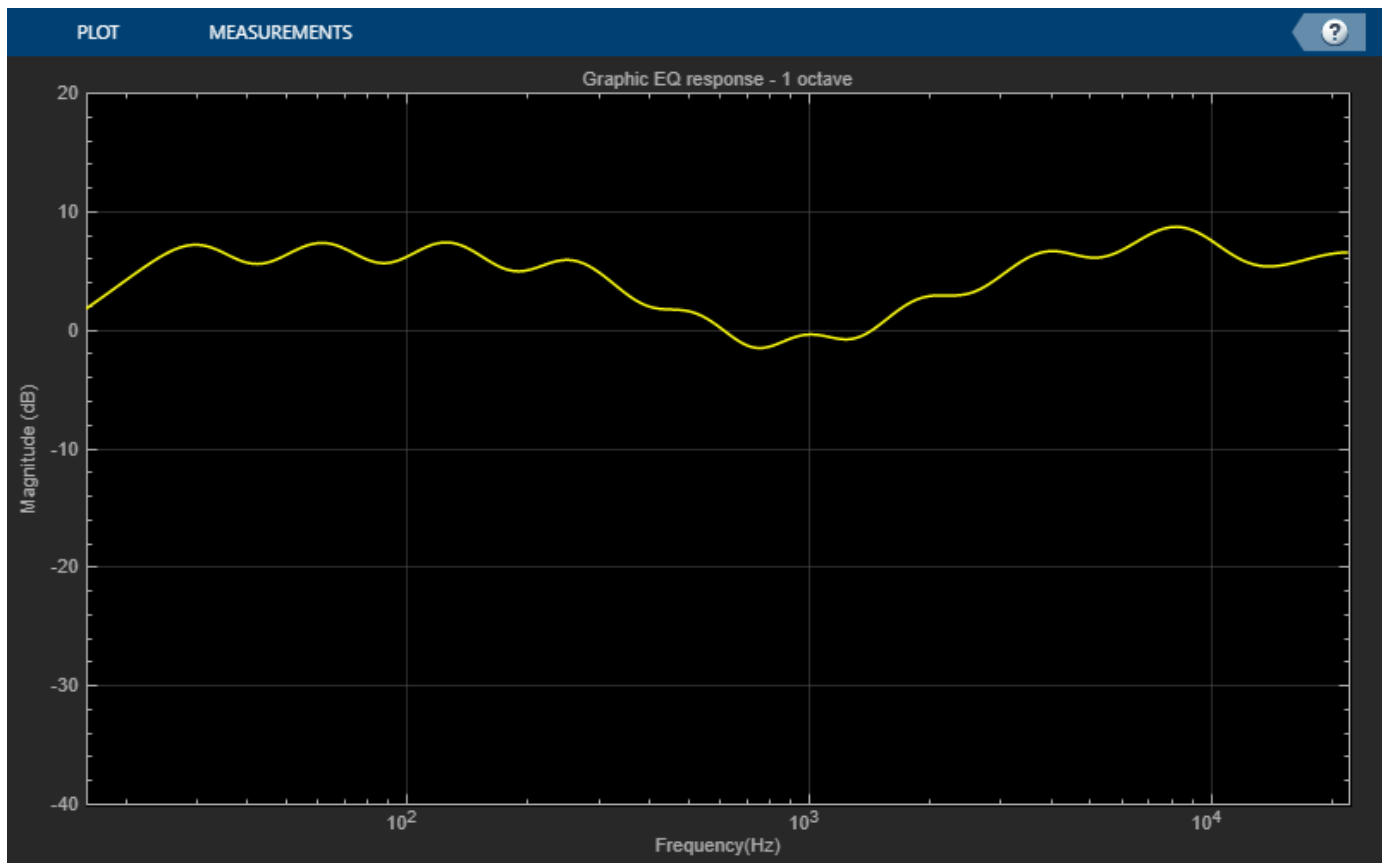
Configure the object you created to boost low and high frequencies, similar to a *rock* preset.

```
eq.Gains = [4, 4.2, 4.6, 2.7, -3.7, -5.2, -2.5, 2.3, 5.4, 6.5, 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
```

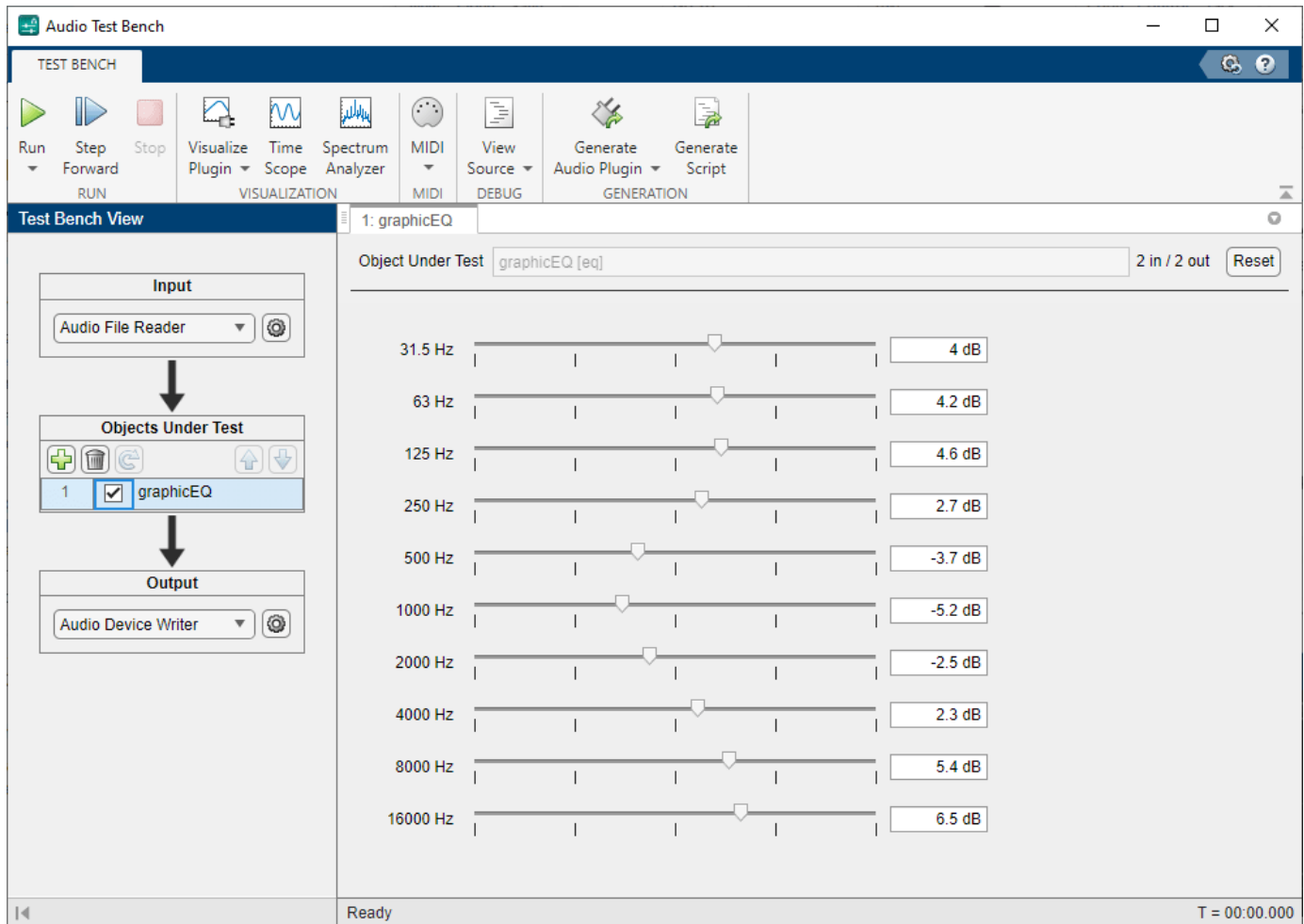
Call `visualize` to view the magnitude response of the equalizer design.

```
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)
```



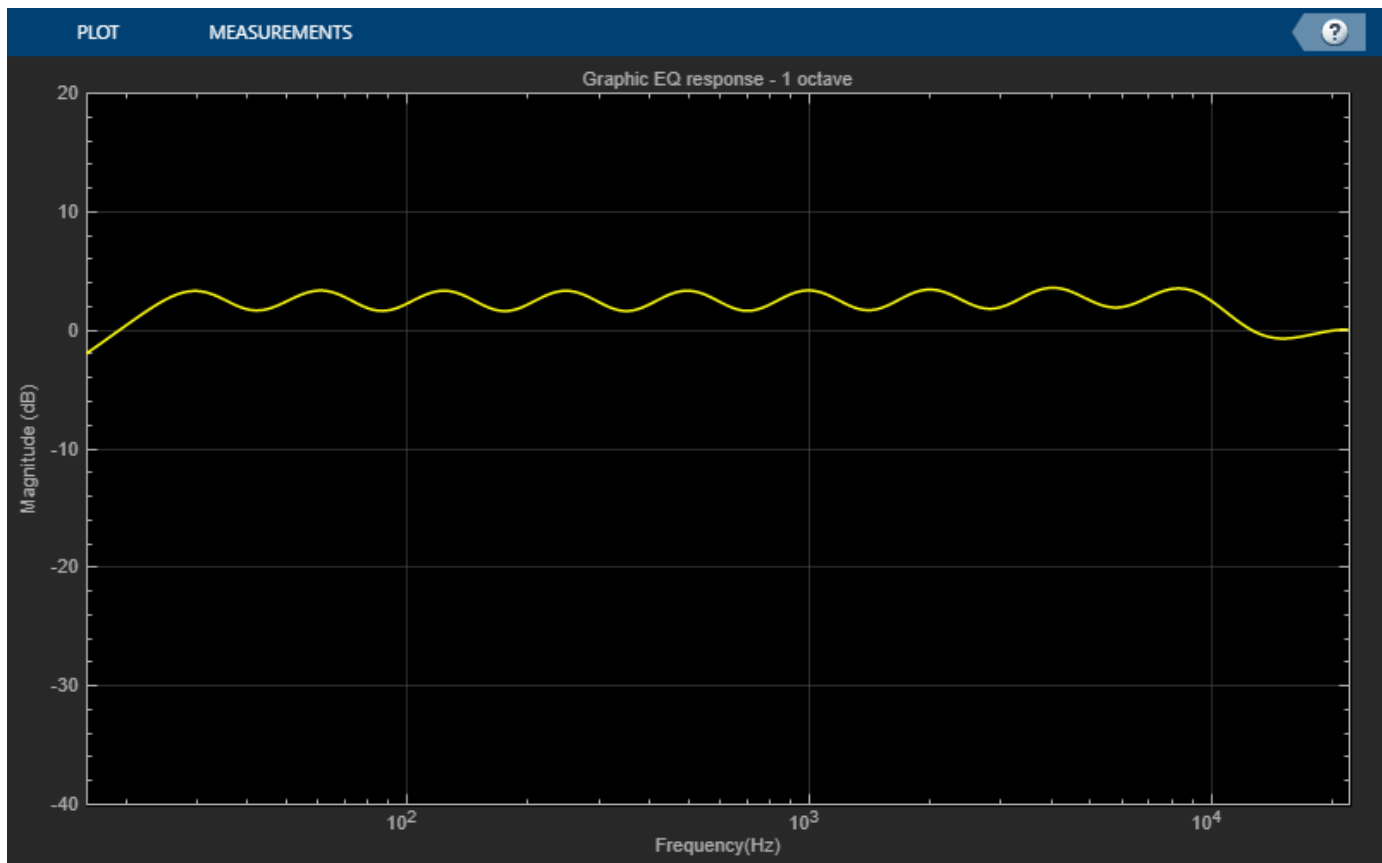
Graphic Equalizers with Cascade Filters

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.

$$H_{eq}(z) = \prod_{m=1}^M H_m(z)$$

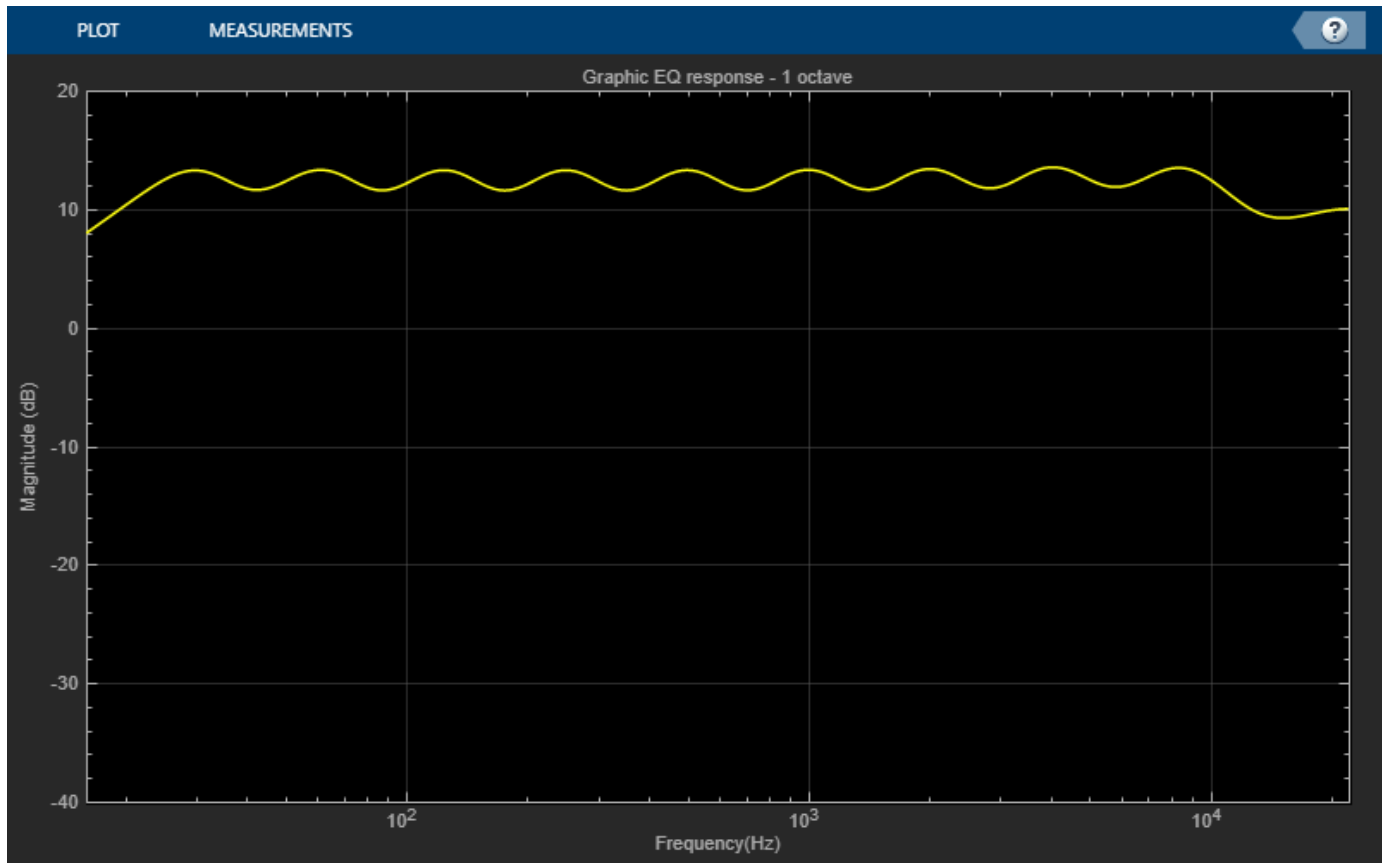
To motivate the usefulness of this implementation, first look at the magnitude response of the parallel structure when all gains are 0 dB.

```
parallelGraphicEQ = graphicEQ('Structure','Parallel');
visualize(parallelGraphicEQ)
```



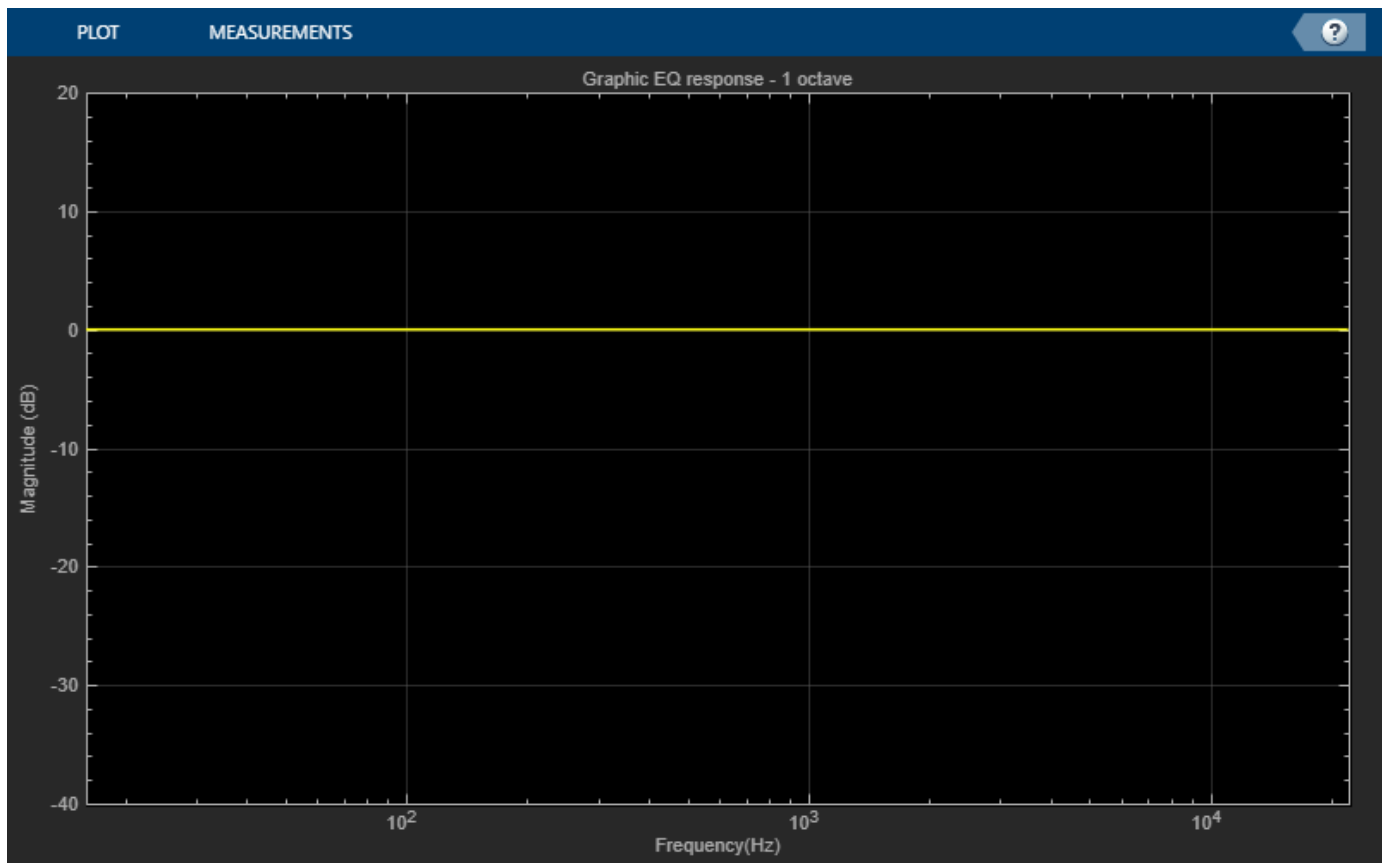
You will notice that the magnitude response is not flat. This is because the filters have been designed independently, and each has a transition width where the magnitude response droops. Moreover, because of non-ideal stopband, there is leakage from the stopband of one filter to the passband of its 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)
```



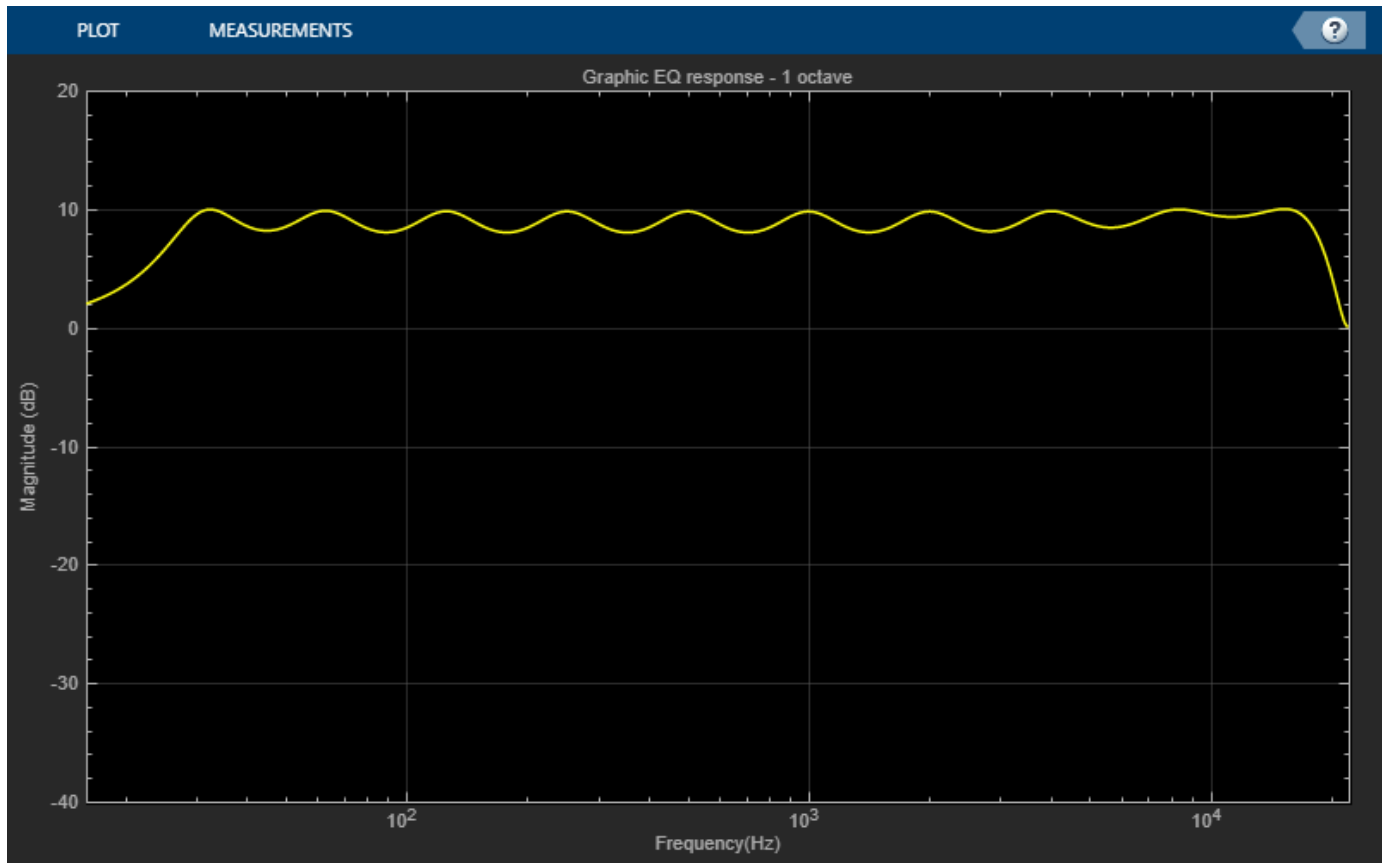
Note that the gains are never 10 dB in the frequency response. A cascaded structure can mitigate this to an extent because the gain is inherent in the design of the filter. Setting the gain of all cascaded biquads to 0 dB leads to them being bypassed. Since there are no branches in this type of structure, 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)
```

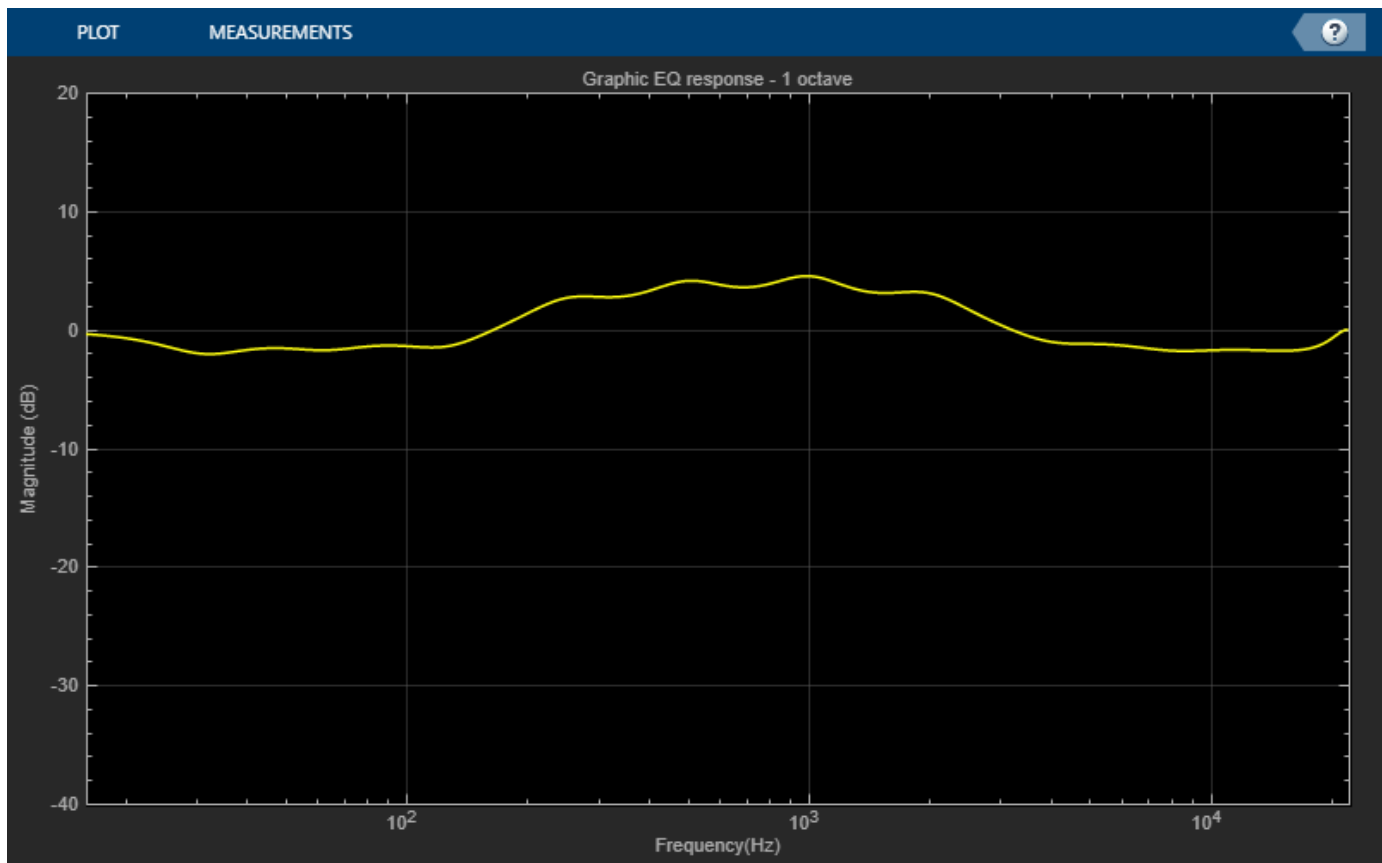


The drawback of cascade design is that the coefficients of a biquad stage need to be redesigned whenever the corresponding gain changes. This isn't needed for the parallel implementation because gain is just a multiplier to each parallel branch. A parallel connection of bandpass filters also avoids accumulating phase errors and quantization noise found in the cascade.

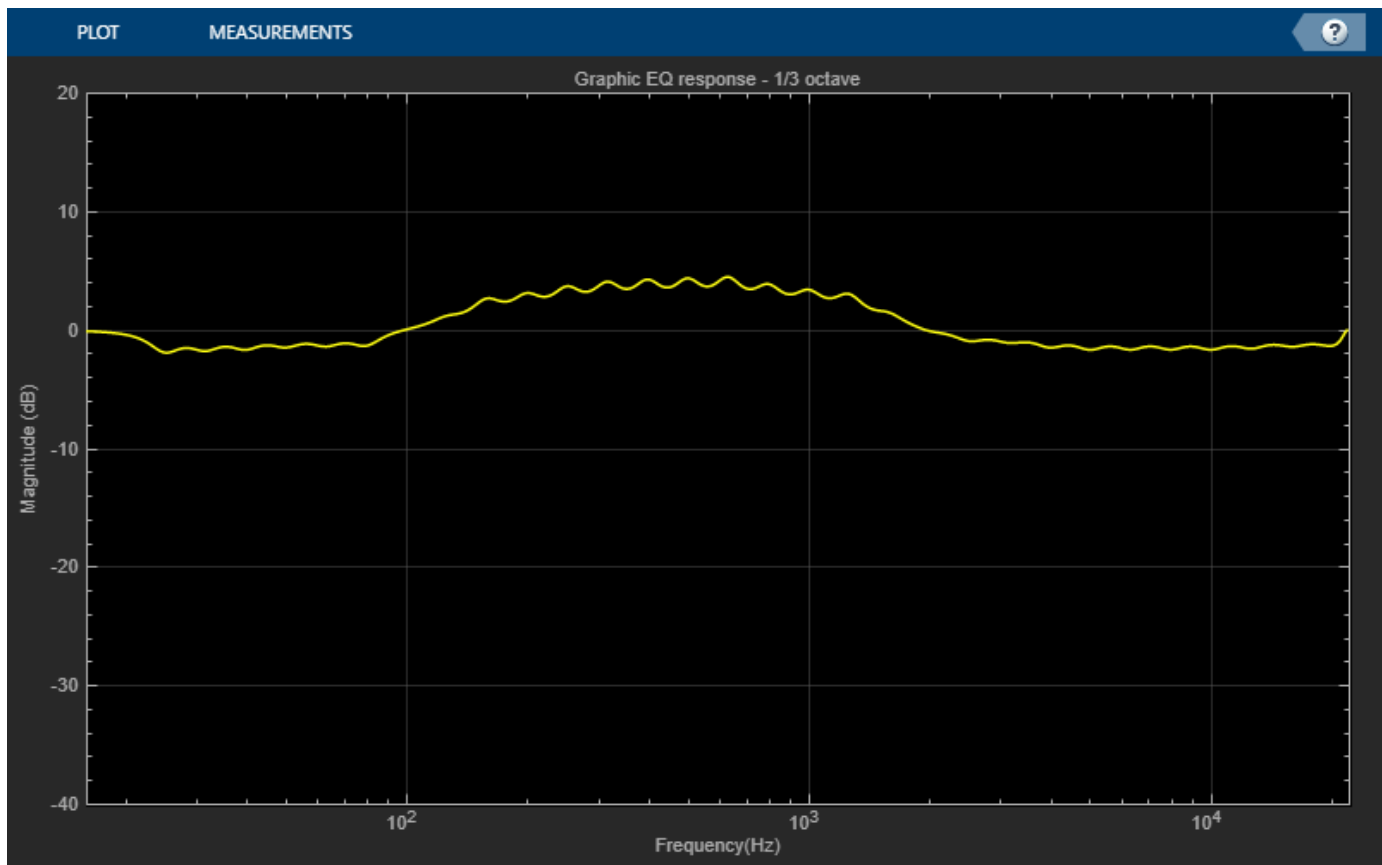
Fractional Octave Bandwidth

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 A-weighting and C-weighting filters the `weightingFilter` System object 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 1-201] and IEC 61672-1 [2 on page 1-201] 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)
```

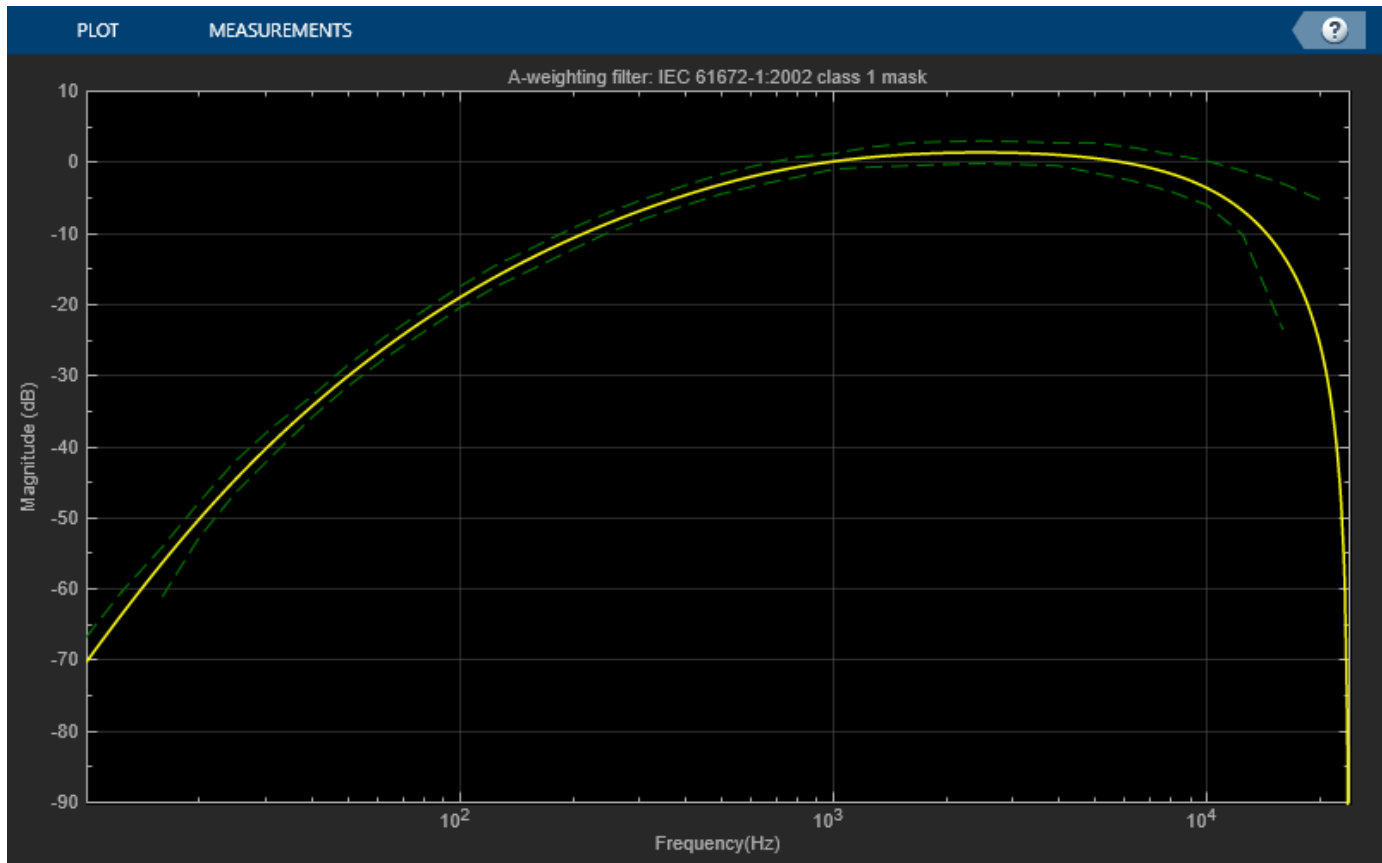
```
AWeighting =  
  weightingFilter with properties:
```

```
    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.

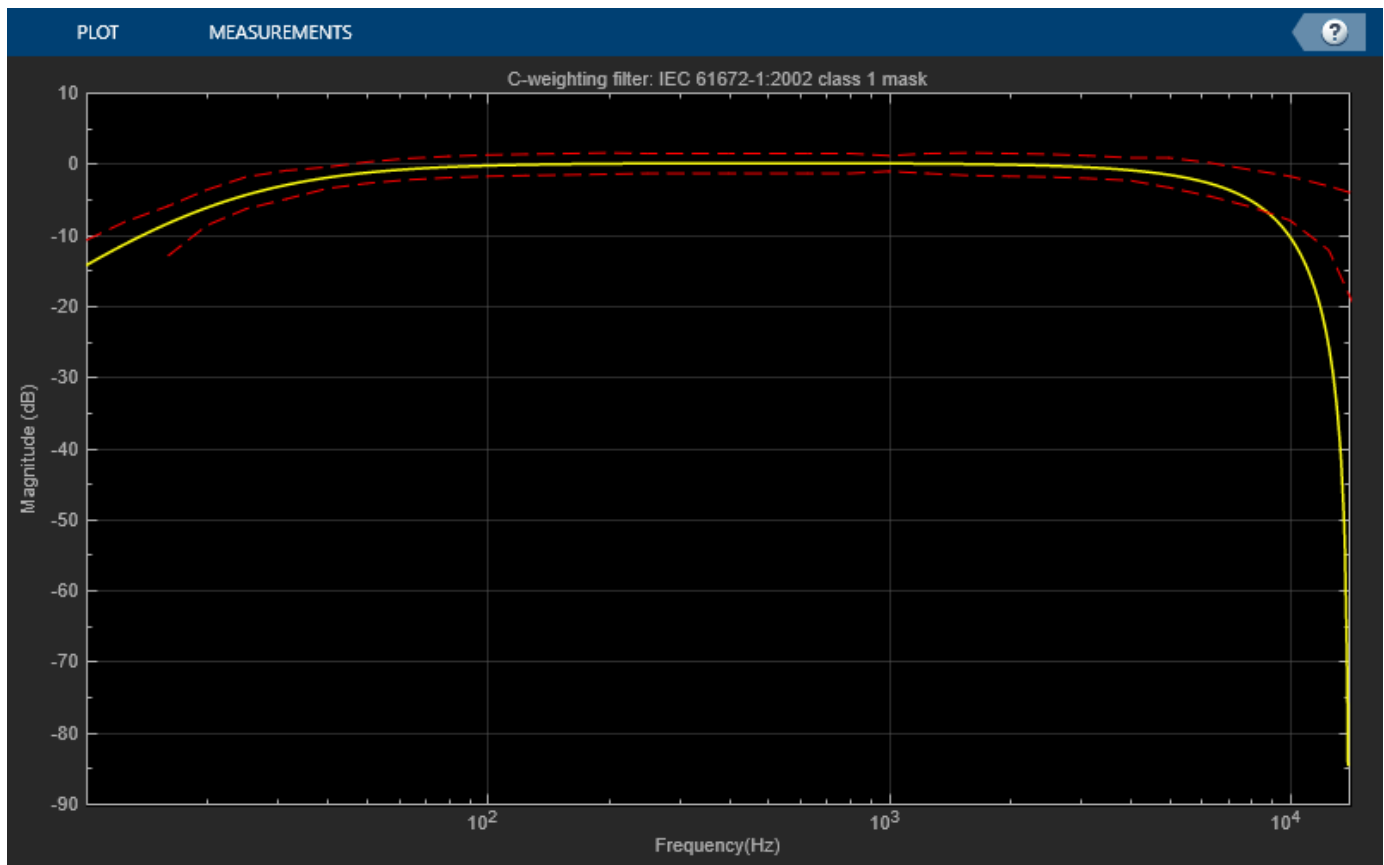
```
CWeighting = weightingFilter('C-weighting',28000)
```

```
CWeighting =  
  weightingFilter with properties:
```

```
    Method: 'C-weighting'  
    SampleRate: 28000
```

```
visualize(CWeighting,'class 2')
```

```
visualize(CWeighting,'class 1')
```



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.

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.

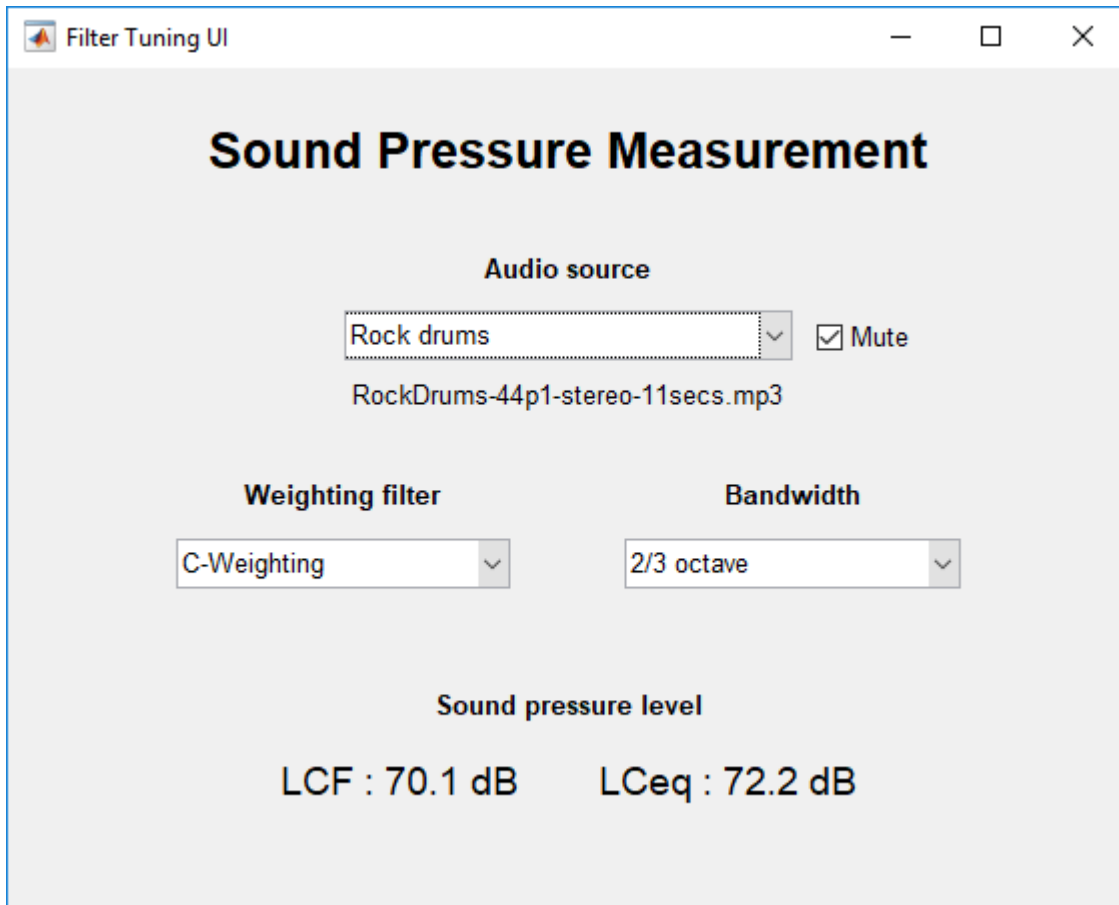
This example uses the `splMeter` System object to measure sound pressure levels (SPL). You can measure sound pressure levels of audio files or perform live SPL measurements with a microphone.

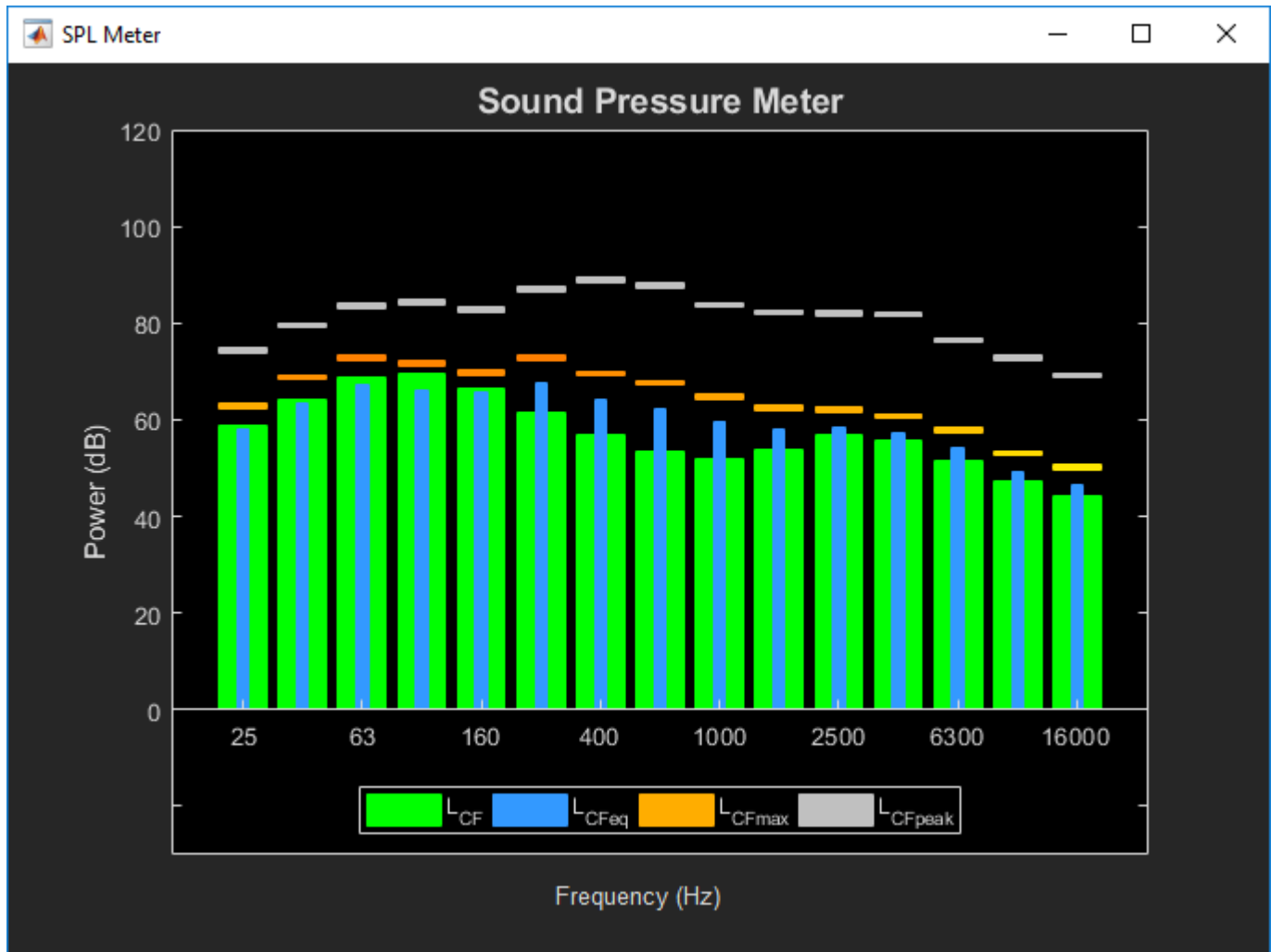
You can specify the weighting filter (Z/A/C) and frequency bandwidth used for the measurements. For more information on the weighting filters, see the "Audio Weighting Filters" on page 1-199 example.

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.





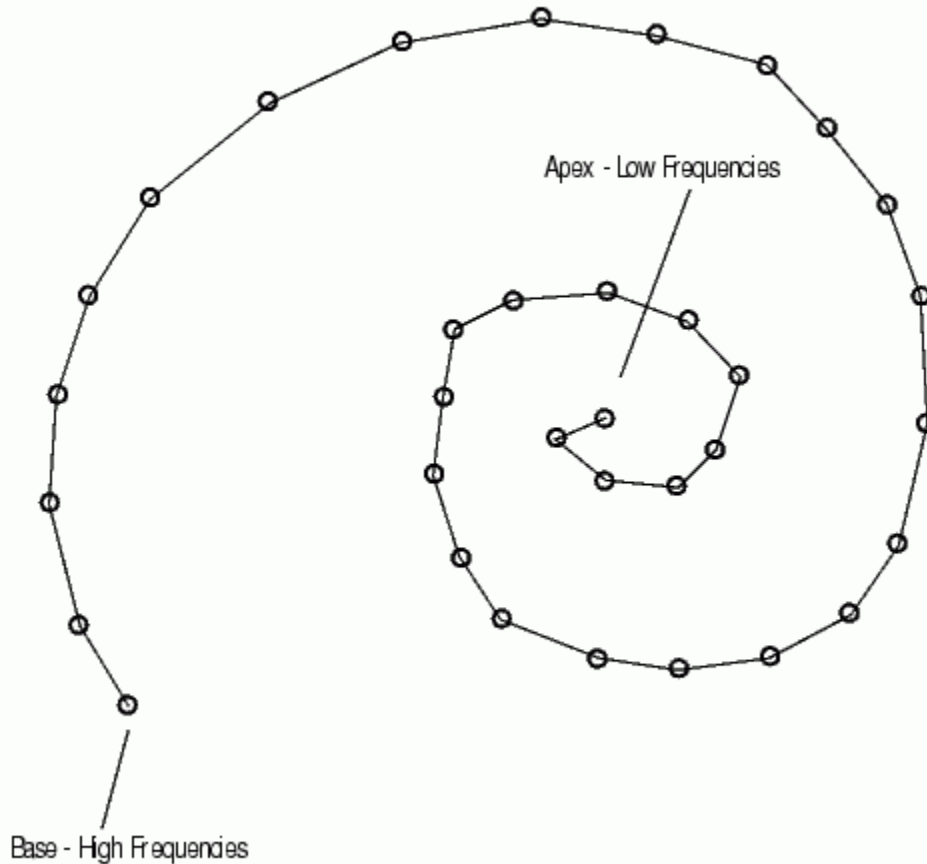
Cochlear Implant Speech Processor

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.

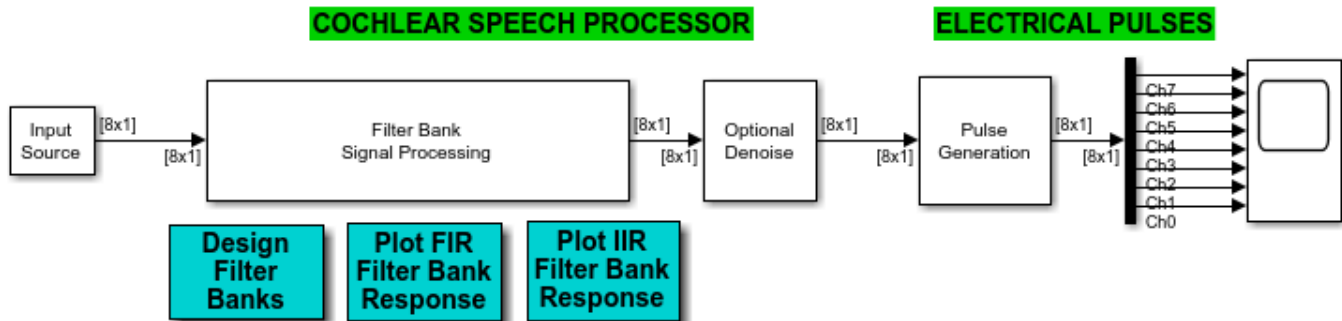
Exploring the Example

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.

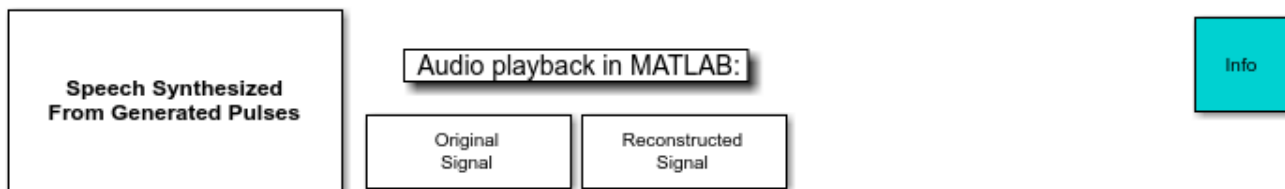
To hear the original input signal, double-click the Original Signal block at the bottom of the model. To hear the output signal of the simulated cochlear implant, double-click the Reconstructed Signal block.

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 Rate Conversion" 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 also 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);

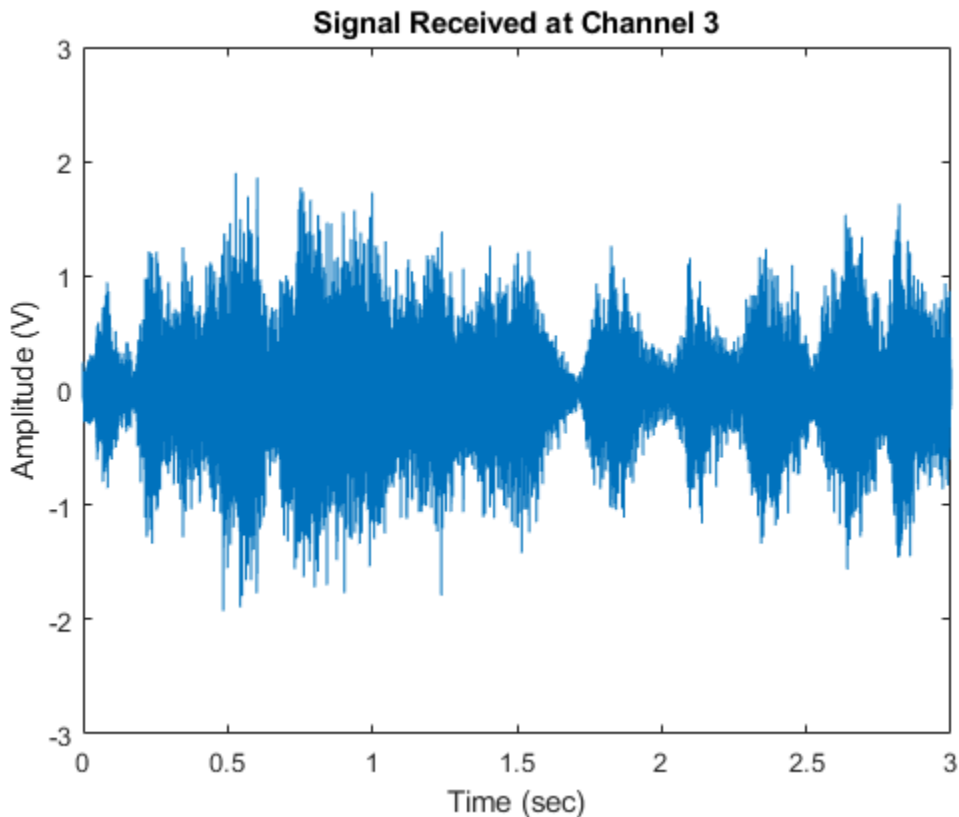
% set up audio device writer
player = audioDeviceWriter('SampleRate',fs);

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);
```

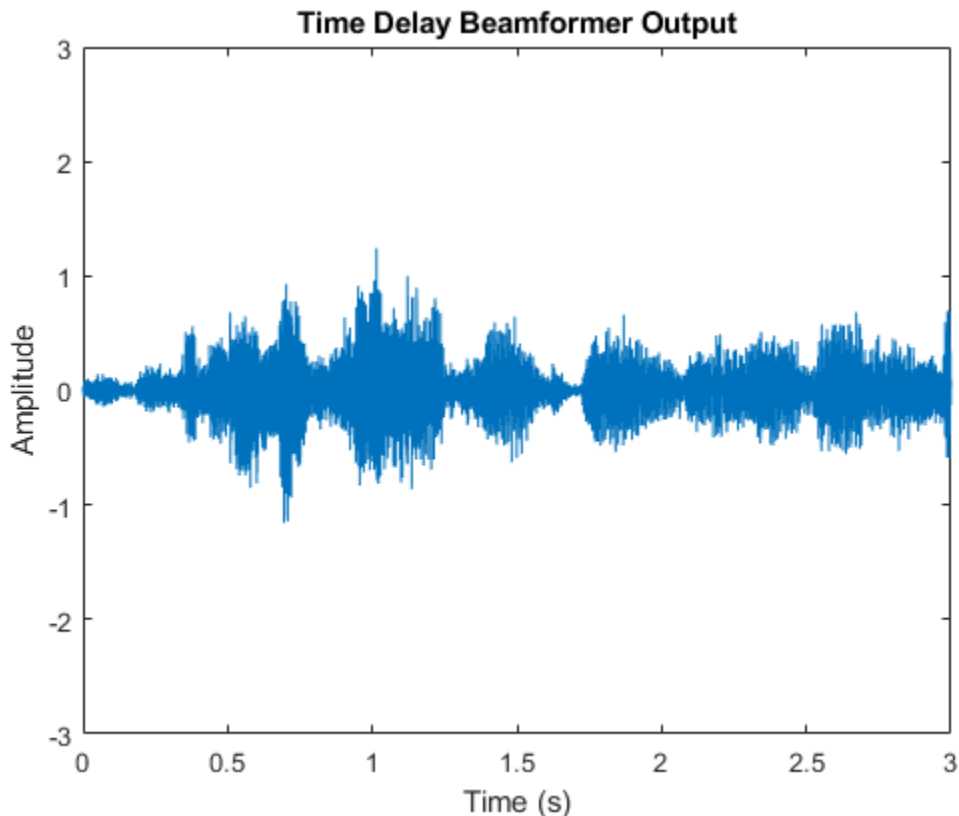
```

cbfOut = zeros(NTSample,1);

for m = 1:NSampPerFrame:NTSample
    temp = beamformer(signalSource());
    player(temp);
    cbfOut(m:m+NSampPerFrame-1,:) = temp;
end

plot(t,cbfOut);
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((cbfOut - 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.

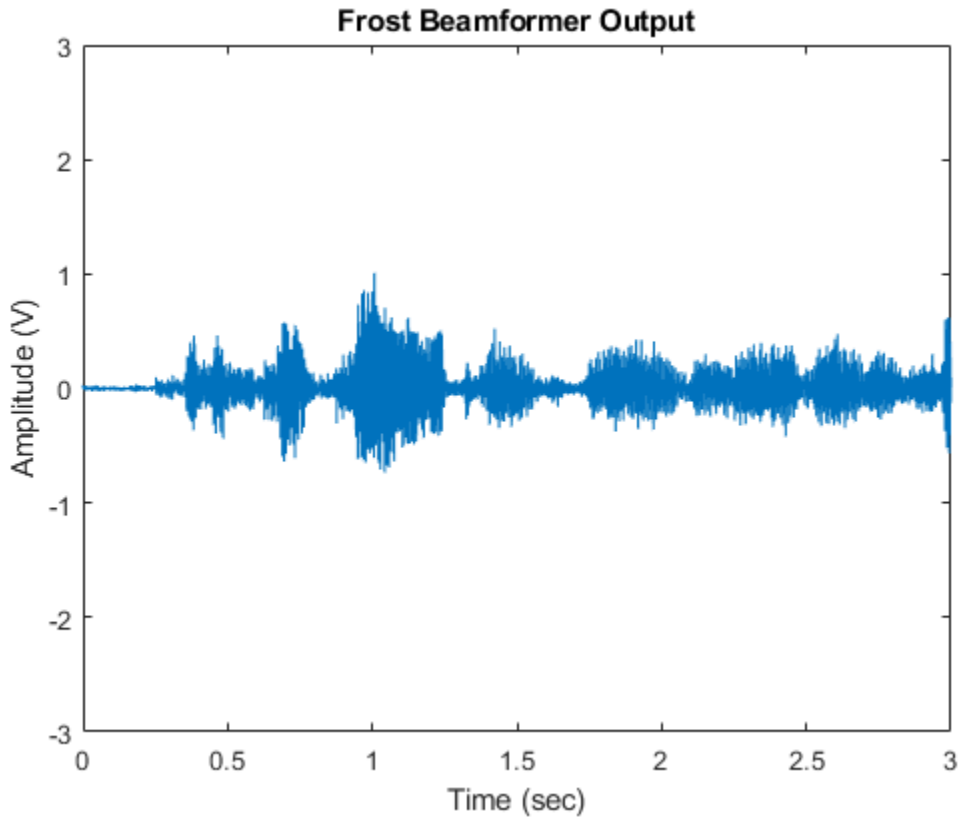
Process with a Frost Beamformer

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

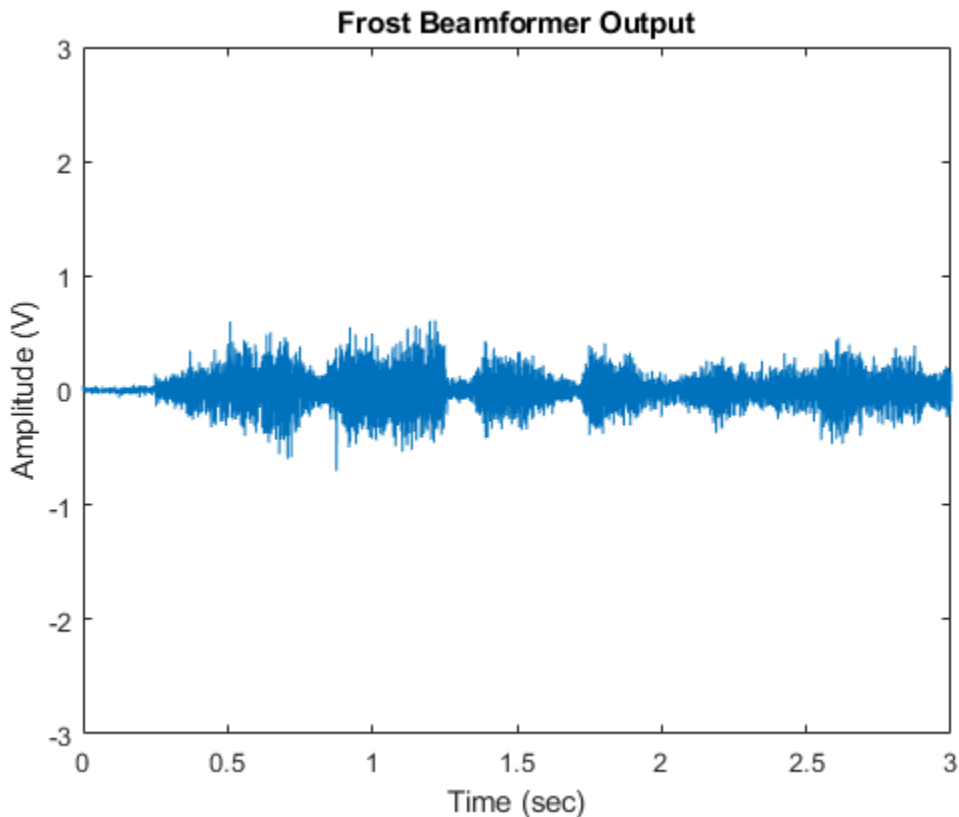
reset(signalsource);
FrostOut2 = zeros(NTSample,1);
for m = 1:NSampPerFrame:NTSample
    temp = frostbeamformer(signalsource(), ang_cleanspeech_est);
    player(temp);
    FrostOut2(m:m+NSampPerFrame-1,:) = temp;
end

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
```

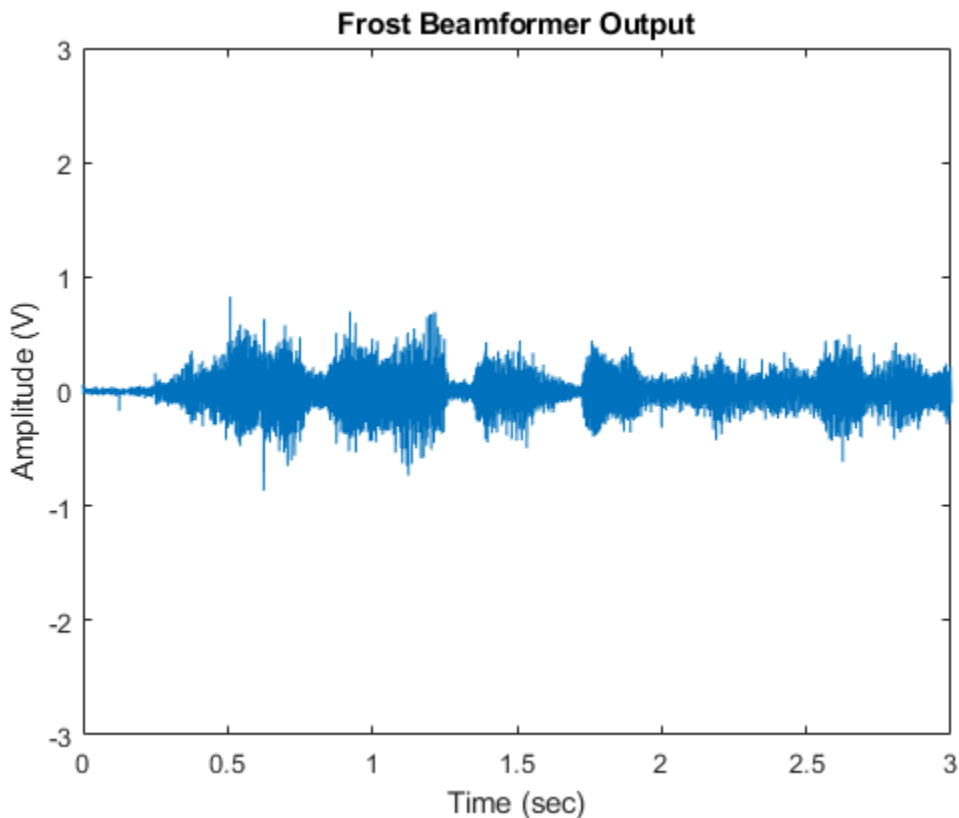
```

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.

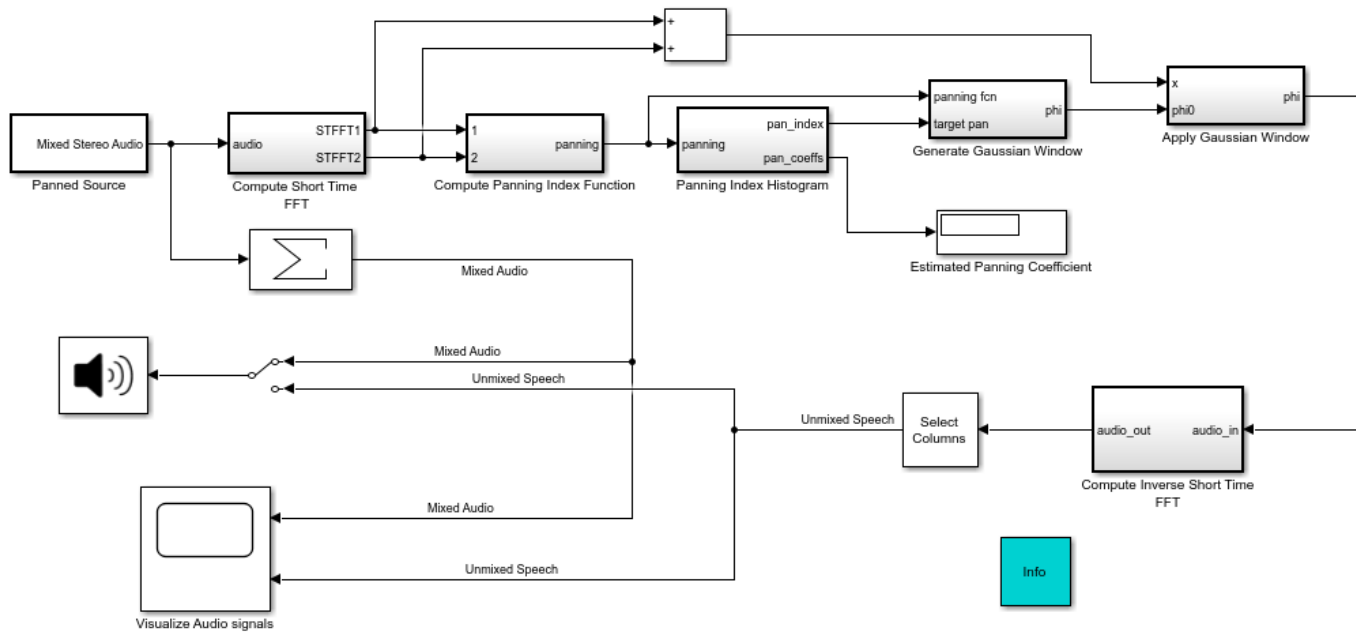
This example showcases a source separation algorithm applied to an audio stereo signal. The stereo signal is a mix of two independently panned audio sources: The first source is a man counting from one to ten, and the second source is a toy train whistle.

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

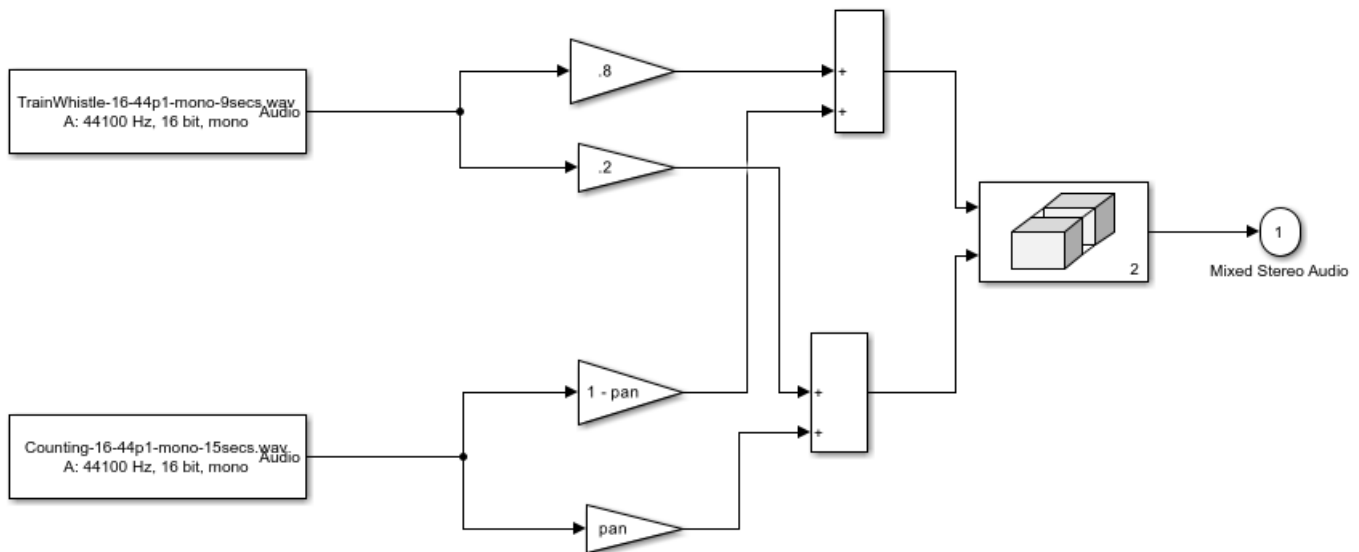
The model `audiosourceseparation` implements the panned audio source separation example.

Panned Audio Source Separation Example



Copyright 2014-2017 The MathWorks, Inc.

The stereo signal is mixed in the Panned Source subsystem. The stereo signal is formed of two panned signals as shown below.

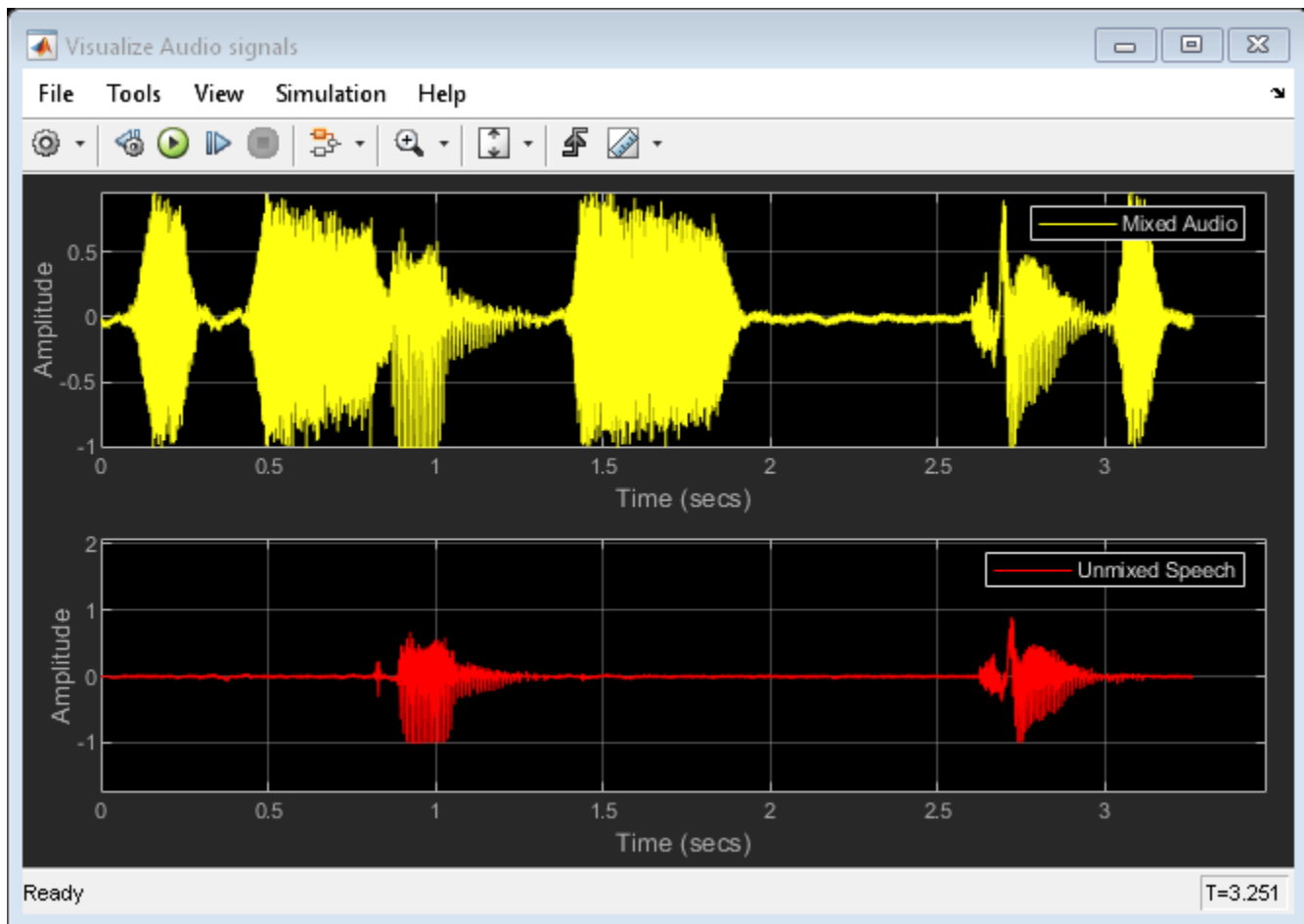


The train whistle source is panned with a constant panning coefficient of 0.2. You may vary the panning coefficient of the speech source by double-clicking the Panned Source subsystem and modifying the position of the 'Panning Index' knob.

The source separation algorithm is implemented in the 'Compute Panning Index Function' subsystem. The algorithm is based on the comparison of the short-time Fourier Transforms of the right and left

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', Avendano, 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.

```
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 = xcorr( ...
                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

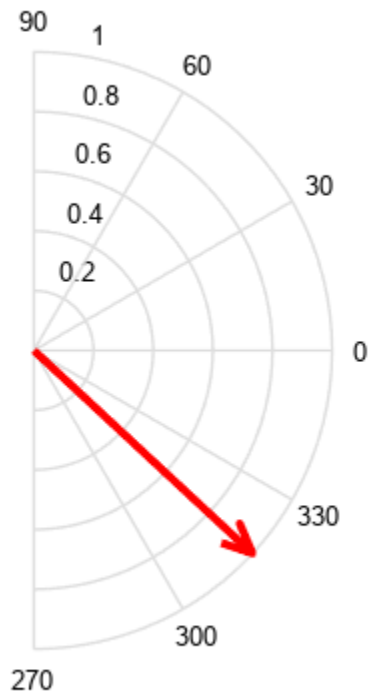
    % Combine DOA estimation across pairs by keeping only the median value
    DOAInRadians = median(anglesInRadians);

    % 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

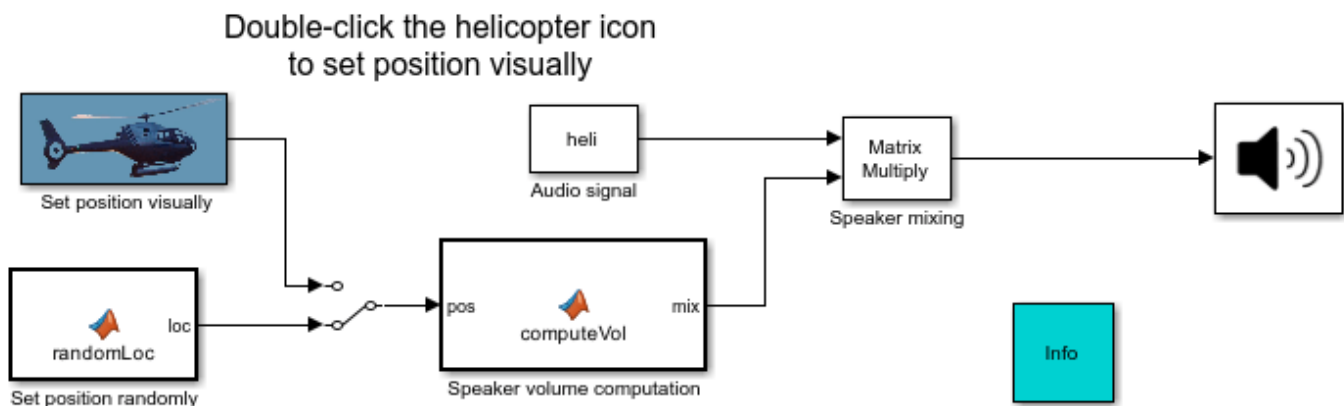
This example requires a 5.1-channel speaker configuration, and relies on the audio channels being mapped to physical locations as follows:

- 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

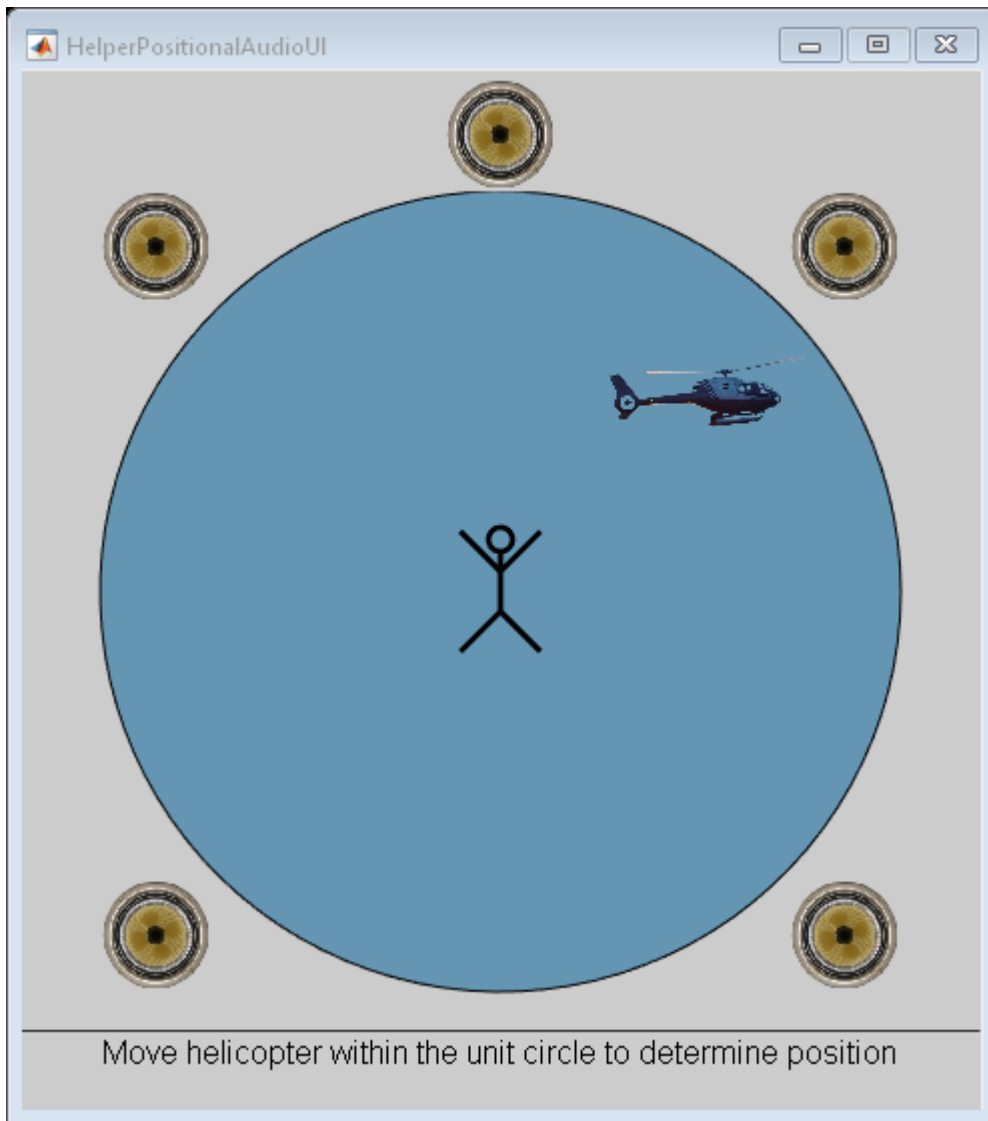
There are two source blocks of interest in the model. The first is the audio signal itself, and the second is the spatial location of the helicopter. The spatial location of the helicopter is represented by a pair of Cartesian coordinates that are constrained to lie within the unit circle. By default, 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

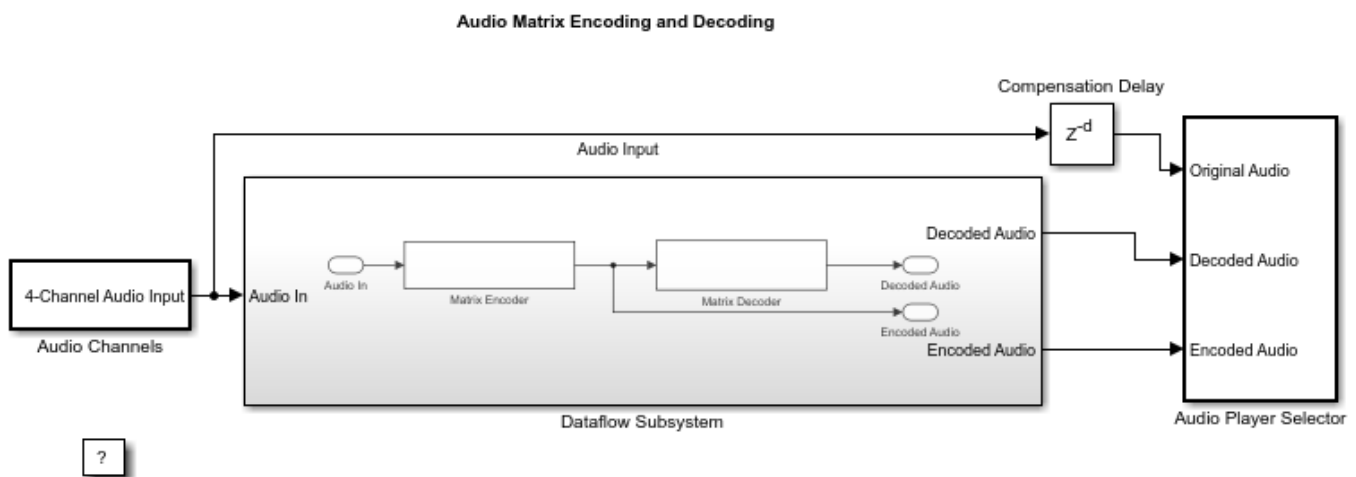
Matrix decoding is an audio technique that decodes an audio signal with M channels into an audio signal with N channels ($N > M$) for play back on a system with N speakers. The original audio signal is usually generated using a matrix encoder, which transforms N -channel signals to M -channel signals.

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].

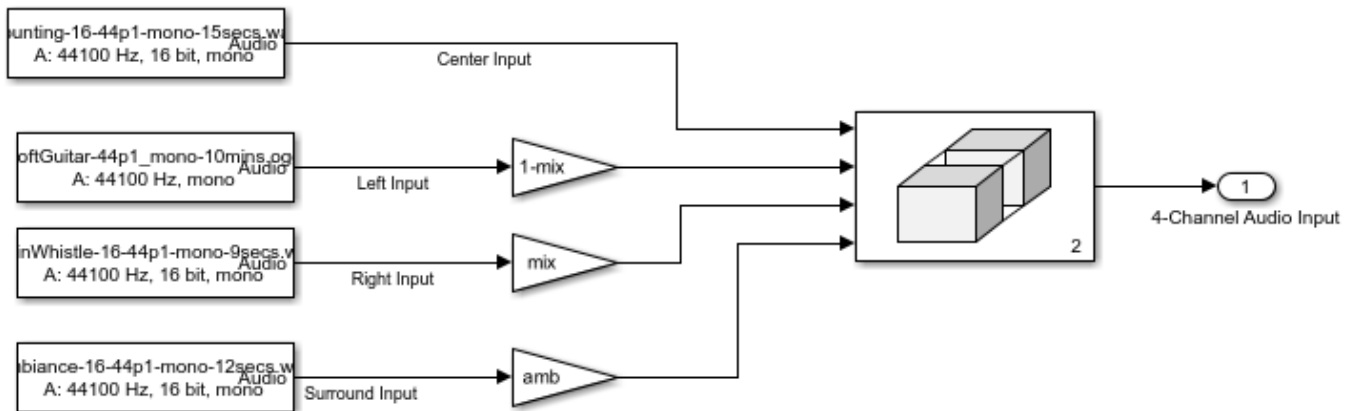
In this example, we showcase a matrix encoder used to encode a four-channel signal (left, right, center and surround) to a stereo signal. The four original signals are then regenerated using a matrix decoder. This example is a simplified version of the encoding and decoding scheme used in the Dolby Pro Logic system [2].

Simulink Version

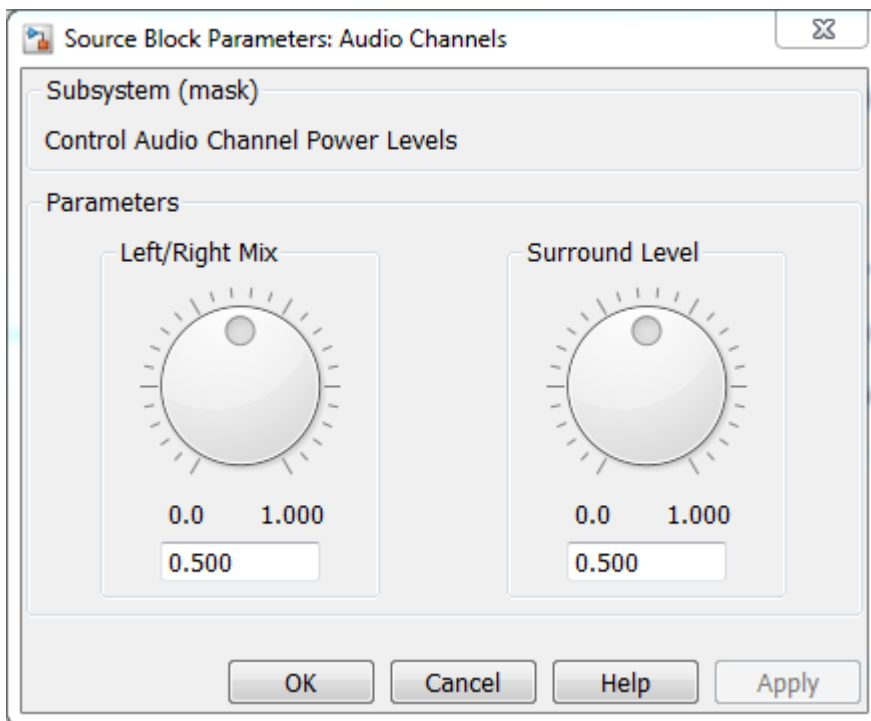
The `audiomatrixdecoding` model implements the audio matrix encoding/decoding example.



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.

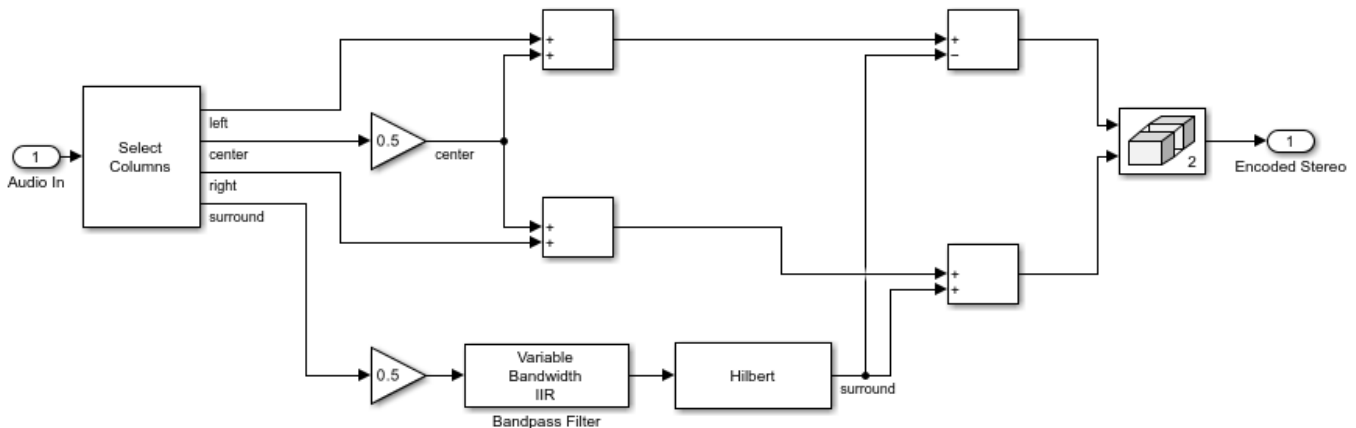
Matrix Encoder

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.

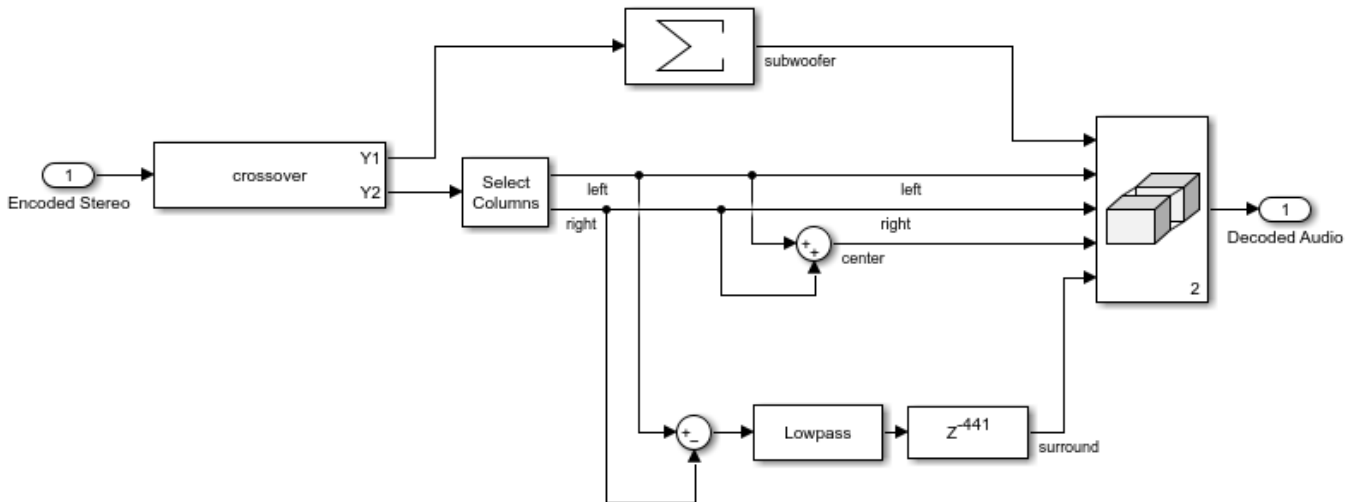
The lowpass frequencies are first separated using a Linkwitz-Riley cross-over filter. For more information about the implementation of the Linkwitz-Riley filter, refer to “Multiband Dynamic Range Compression” on page 1-148.

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.

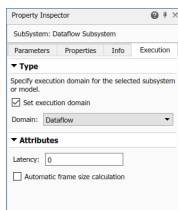


Improve Simulation Performance Using Dataflow Domain

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. To access Property Inspector, in the Simulink Toolstrip, on the Modeling tab, in the Design gallery select Property Inspector or on the Simulation tab, Prepare gallery, select Property Inspector.



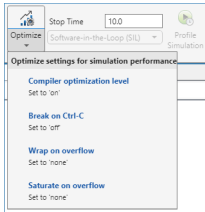
Dataflow domains automatically partition your model into multiple threads for better performance. Once you set the **Domain** parameter to **Dataflow**, you can use the **Multicore** tab analysis to analyze your model to get better performance. The **Multicore** tab is available in the toolstrip when there is a dataflow domain in the model. To learn more about the **Multicore** tab, see “Perform Multicore Analysis for Dataflow”.



Analyzing Concurrency in Dataflow Subsystem

For this example the **Multicore** tab mode is set to Simulation Profiling for simulation performance analysis.

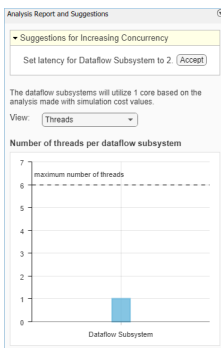
It is recommended to optimize model settings for optimal simulation performance. To accept the proposed model settings, on the **Multicore** tab, click **Optimize**. Alternatively, you can use the drop menu below the **Optimize** button to change the settings individually.



On the **Multicore** tab, click the **Run Analysis** button to start the analysis of the dataflow domain for simulation performance. Once the analysis is finished, the Analysis Report and Suggestions window shows how many threads the dataflow subsystem uses during simulation.

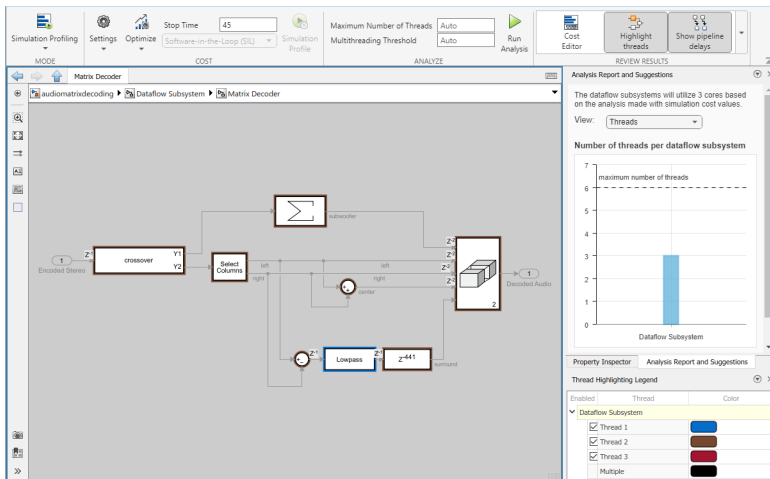
After analyzing the model, the Analysis Report and Suggestions window shows one thread because the data dependency between the blocks in the model prevents blocks from being executed concurrently. By pipelining the data dependent blocks, the dataflow subsystem can increase concurrency for higher data throughput. The Analysis Report and Suggestions window shows the recommended number of pipeline delays as Suggested for Increasing Concurrency. The suggested latency value is computed to give the best performance.

The following diagram shows the Analysis Report and Suggestions window where the suggested latency is 2 for the dataflow subsystem.



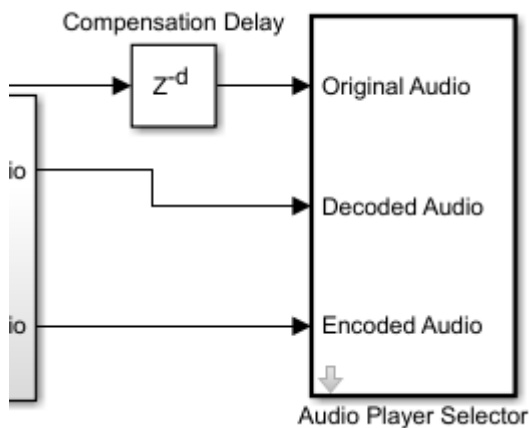
Click the **Accept** button to use the recommended latency for the dataflow subsystem. This value can also be entered directly in the Property Inspector for **Latency** parameter. Simulink shows the latency parameter value using Z^{-n} tags at the output ports of the dataflow subsystem.

The Analysis Report and Suggestions window now shows the number of threads as 2 meaning that the blocks inside the dataflow subsystem simulate in parallel using 2 threads. **Highlight threads** highlights the blocks with colors based on their thread allocation as shown in the **Thread Highlighting Legend**. **Show pipeline delays** shows where pipelining delays were inserted within the dataflow subsystem using Z^{-n} tags.



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 the 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 2.3x 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

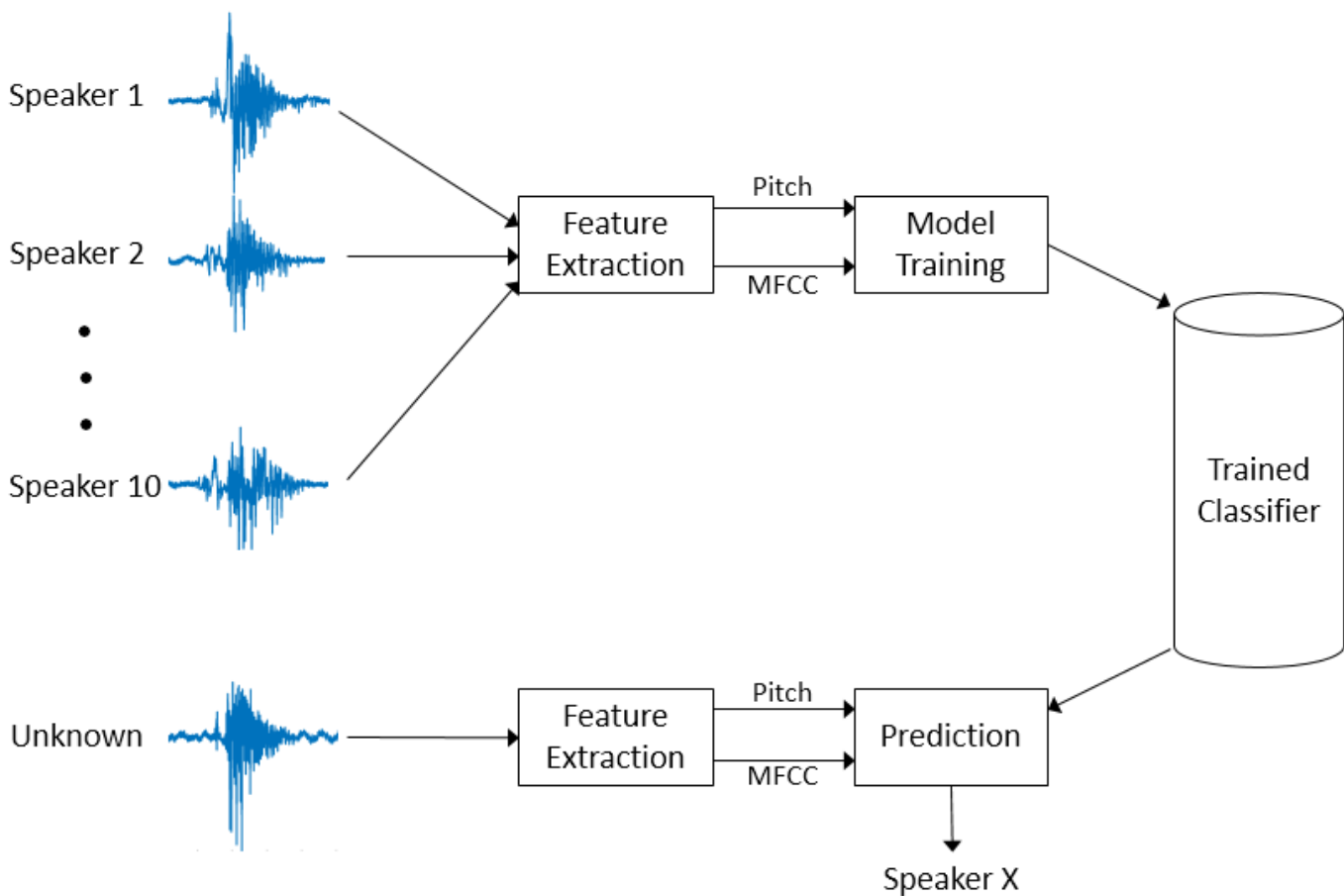
[3] https://en.wikipedia.org/wiki/Precedence_effect

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, zero-crossing rate, short-time energy, and MFCC. Pitch and MFCC are the two features that are used to classify speakers. Zero-crossing rate and short-time energy are used to determine when the pitch feature is used.

Pitch

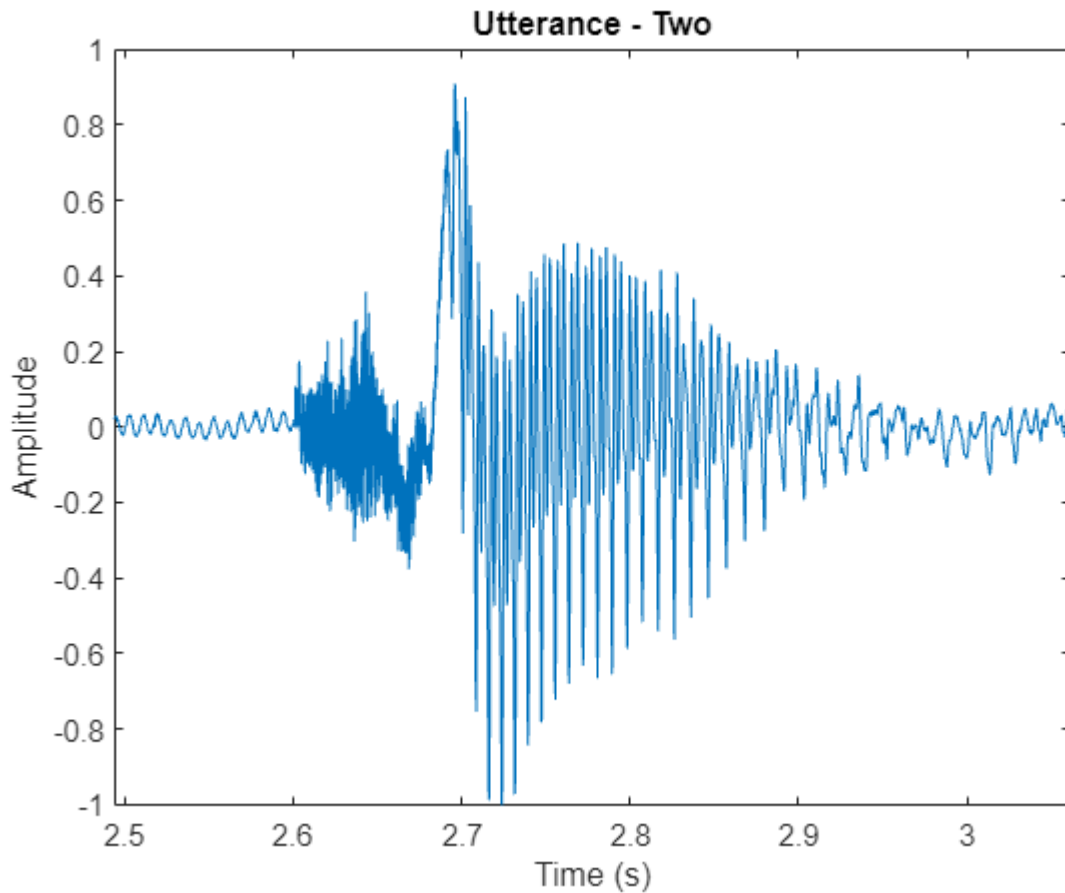
Speech can be broadly categorized as *voiced* and *unvoiced*. In the case of voiced speech, air from the lungs is modulated by vocal cords and results in a quasi-periodic excitation. The resulting sound is dominated by a relatively low-frequency oscillation, referred to as *pitch*. In the case of unvoiced 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 the pitch changes over time.

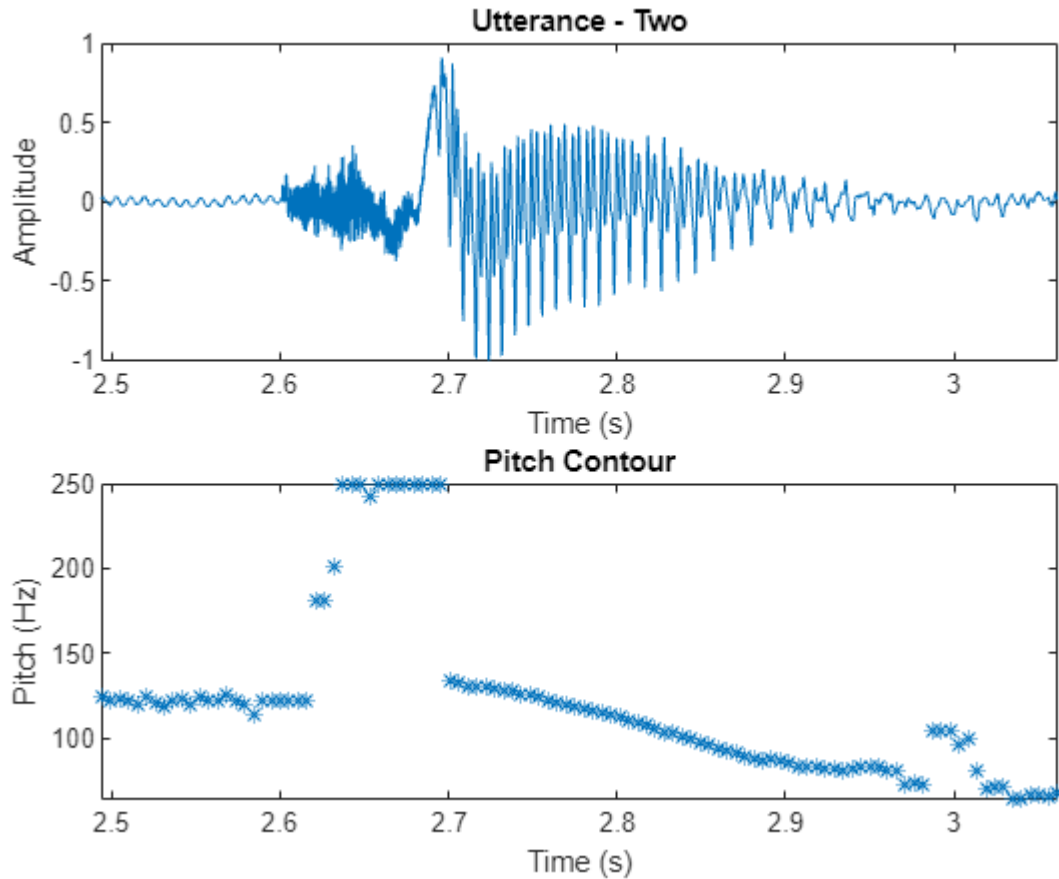
```
windowLength = round(0.03*fs);
overlapLength = round(0.025*fs);
```

```
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 time energy. If the energy in a frame is above a given threshold, you declare the frame as speech.

```
energyThreshold = 20;
[segments,~] = buffer(audioIn,windowLength,overlapLength,"nodelay");
ste = sum((segments.*hamming(windowLength,"periodic")).^2,1);
isSpeech = ste(:) > energyThreshold;
```

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 = 0.02;
zcr = zerocrossrate(audioIn,WindowLength=windowLength,OverlapLength=overlapLength);
isVoiced = zcr < zcrThreshold;
```

Combine `isSpeech` and `isVoiced` to determine whether a frame contains voiced speech.

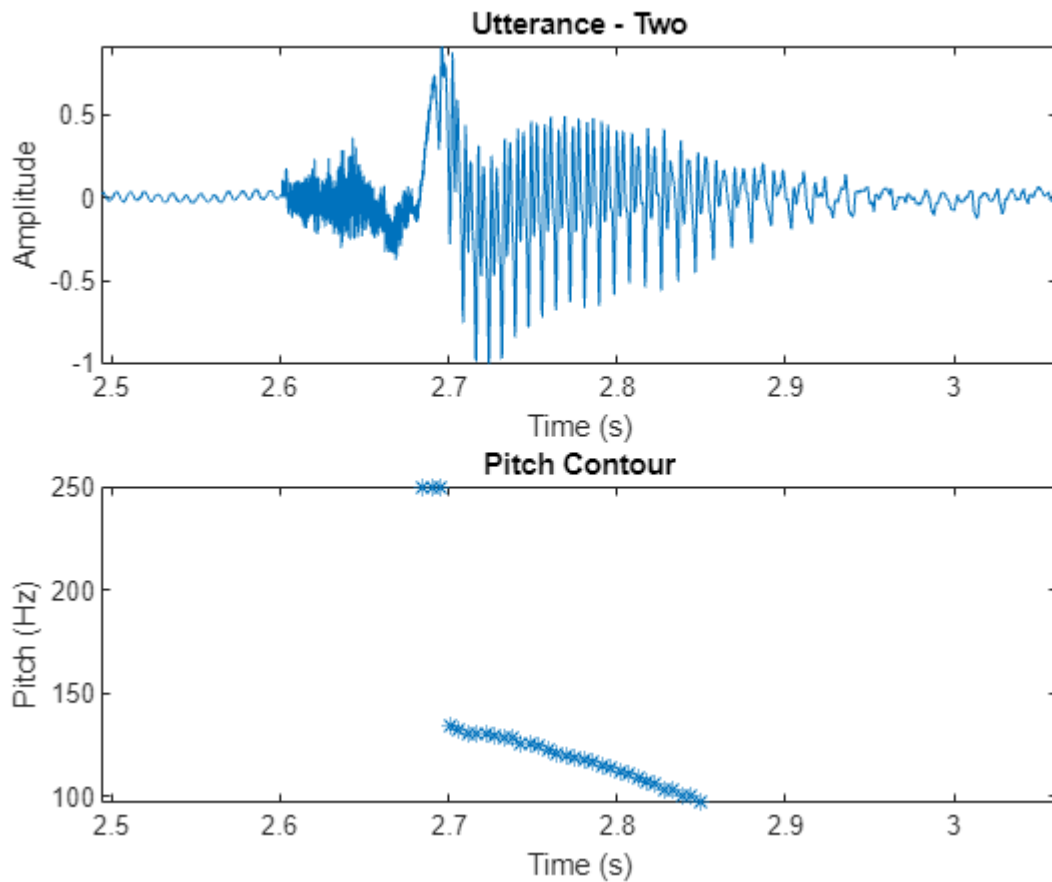
```
voicedSpeech = isSpeech & isVoiced;
```

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")
```

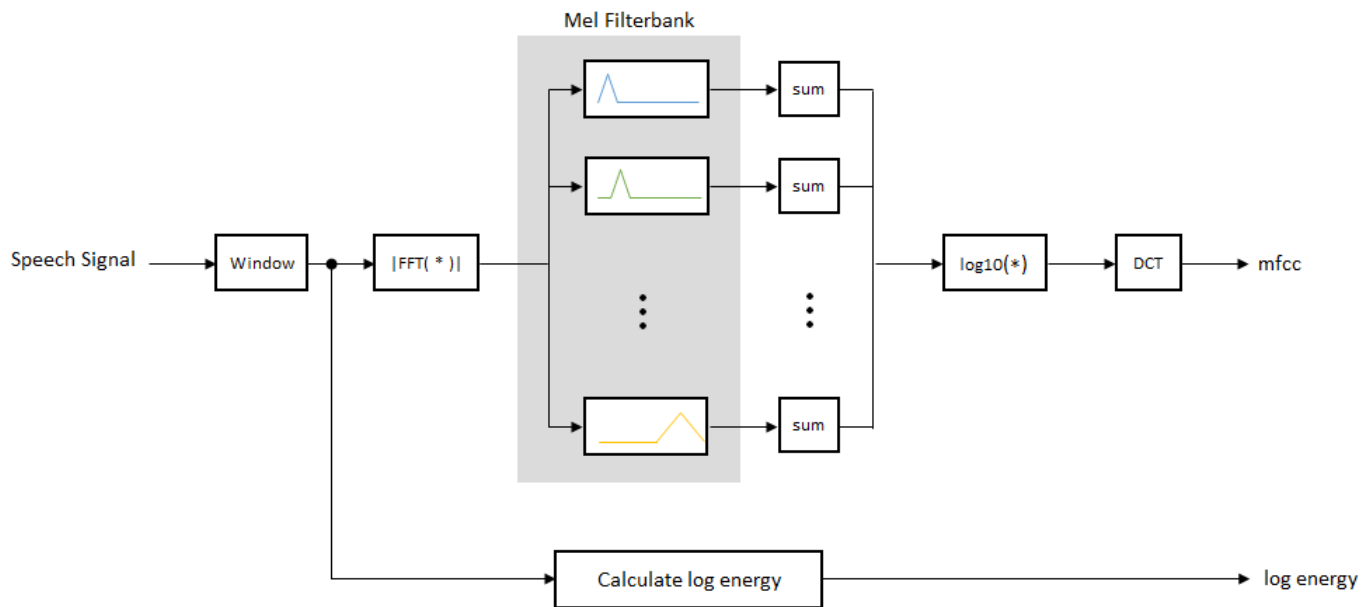


Mel-Frequency Cepstrum Coefficients (MFCC)

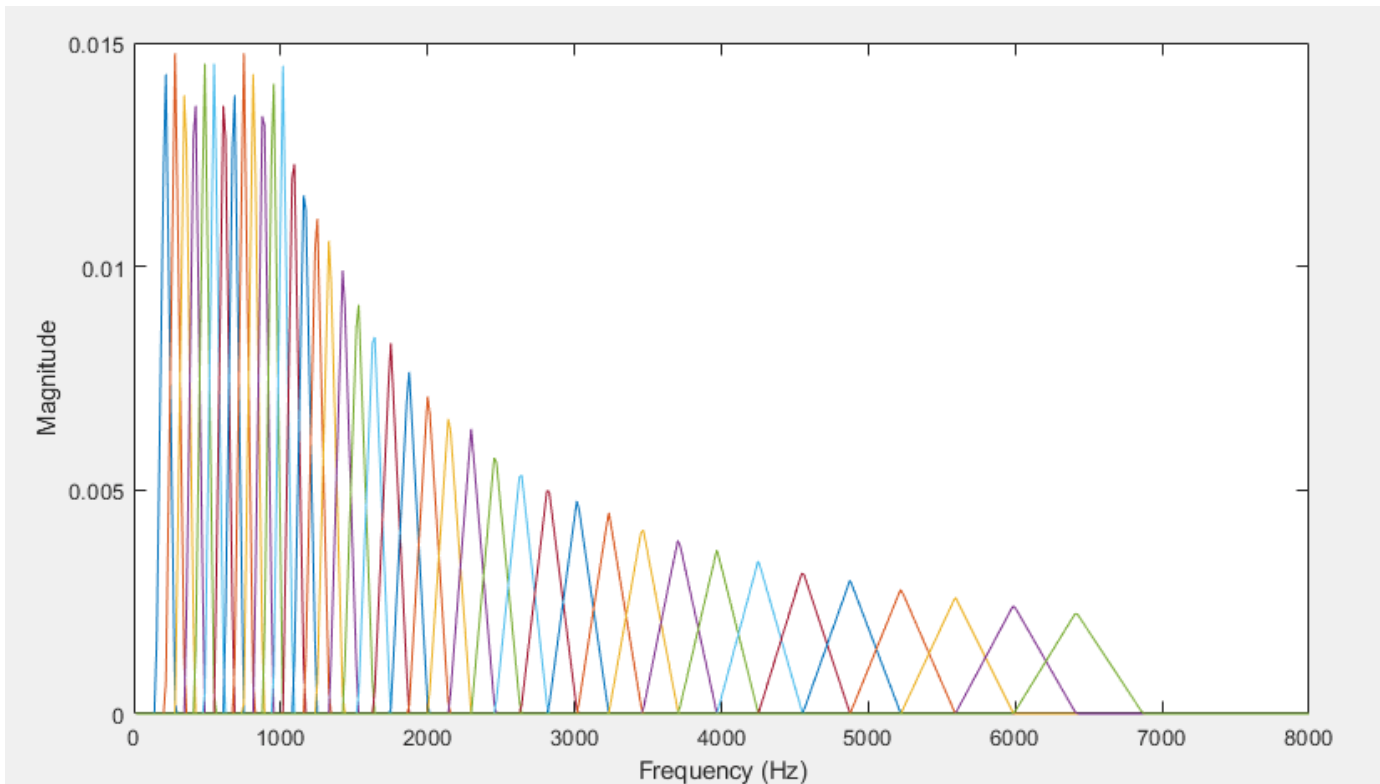
MFCC are popular features extracted from speech signals for use in recognition tasks. In the source-filter model of speech, MFCC are understood to represent the filter (vocal tract). The frequency response of the vocal tract is relatively smooth, whereas the source of voiced speech can be modeled as 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 a subset of the Common Voice dataset from Mozilla [1] on page 1-250. The dataset contains 48 kHz recordings of subjects speaking short sentences. The helper function in this section organizes the downloaded data and returns an `audioDatastore` object. The dataset uses 1.36 GB of memory.

Download the dataset if it doesn't already exist and unzip it into `tempdir`.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "commonvoice.zip");
dataFolder = tempdir;
if ~datasetExists(string(dataFolder) + "commonvoice")
    unzip(downloadFolder, dataFolder);
end
```

Extract the speech files for 10 speakers (5 female and 5 male) and place them into an `audioDatastore` using the `commonVoiceHelper` function. The datastore enables you to collect necessary files of a file format and read them. The function is placed in your current folder when you open this example.

```
ads = commonVoiceHelper
```

```
ads =
    audioDatastore with properties:
```

```
Files: {
    '...\AppData\Local\Temp\commonvoice\train\clips\common_voice_en_1'
```

```

        ' ...\AppData\Local\Temp\commonvoice\train\clips\common_voice_en_1
        ' ...\AppData\Local\Temp\commonvoice\train\clips\common_voice_en_1
        ... and 172 more
    }
    Folders: {
        'C:\Users\jblock\AppData\Local\Temp\commonvoice\train\clips'
    }
    Labels: [3; 3; 3 ... and 172 more categorical]
AlternateFileSystemRoots: {}
    OutputDataType: 'double'
    SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "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: {
        ' ...\AppData\Local\Temp\commonvoice\train\clips\common_voice_en_1
        ' ...\AppData\Local\Temp\commonvoice\train\clips\common_voice_en_1
        ' ...\AppData\Local\Temp\commonvoice\train\clips\common_voice_en_1
        ... and 136 more
    }
    Folders: {
        'C:\Users\jblock\AppData\Local\Temp\commonvoice\train\clips'
    }
    Labels: [3; 3; 3 ... and 136 more categorical]
AlternateFileSystemRoots: {}
    OutputDataType: 'double'
    SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
    DefaultOutputFormat: "wav"

```

```
trainDatastoreCount = countEachLabel(adsTrain)
```

```
trainDatastoreCount=10x2 table
```

Label	Count
1	14
10	12
2	12
3	18
4	14
5	16
6	17

```

7      11
8      11
9      14

```

Display the datastore and the number of speakers in the test datastore.

```
adsTest
```

```
adsTest =
```

```
  audioDatastore with properties:
```

```

      Files: {
        ' ... \AppData\Local\Temp\commonvoice\train\clips\common_voice_en_1
        ' ... \AppData\Local\Temp\commonvoice\train\clips\common_voice_en_1
        ' ... \AppData\Local\Temp\commonvoice\train\clips\common_voice_en_1
        ... and 33 more
      }
      Folders: {
        'C:\Users\jblock\AppData\Local\Temp\commonvoice\train\clips'
      }
      Labels: [3; 3; 3 ... and 33 more categorical]
  AlternateFileSystemRoots: {}
      OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
      DefaultOutputFormat: "wav"

```

```
testDatastoreCount = countEachLabel(adsTest)
```

```
testDatastoreCount=10x2 table
```

Label	Count
1	4
10	3
2	3
3	4
4	4
5	4
6	4
7	3
8	3
9	4

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. Audio Toolbox™ provides `audioFeatureExtractor` so that you can quickly and efficiently extract multiple features. Configure an `audioFeatureExtractor` to extract pitch, short-time energy, zcr, and MFCC.

```
fs = dsInfo.SampleRate;
windowLength = round(0.03*fs);
overlapLength = round(0.025*fs);
afe = audioFeatureExtractor(SampleRate=fs, ...
    Window=hamming(windowLength,"periodic"),OverlapLength=overlapLength, ...
    zerocrossrate=true,shortTimeEnergy=true,pitch=true,mfcc=true);
```

When you call the `extract` function of `audioFeatureExtractor`, all features are concatenated and returned in a matrix. You can use the `info` function to determine which columns of the matrix correspond to which features.

```
featureMap = info(afe)

featureMap = struct with fields:
    mfcc: [1 2 3 4 5 6 7 8 9 10 11 12 13]
    pitch: 14
    zerocrossrate: 15
    shortTimeEnergy: 16
```

Extract features from the data set.

```
features = [];
labels = [];
energyThreshold = 0.005;
zcrThreshold = 0.2;

keepLen = round(length(sampleTrain)/3);

while hasdata(adsTrain)
    [audioIn,dsInfo] = read(adsTrain);

    % Take the first portion of each recording to speed up code
    audioIn = audioIn(1:keepLen);

    feat = extract(afe,audioIn);
    isSpeech = feat(:,featureMap.shortTimeEnergy) > energyThreshold;
    isVoiced = feat(:,featureMap.zerocrossrate) < zcrThreshold;

    voicedSpeech = isSpeech & isVoiced;

    feat(~voicedSpeech,:) = [];
    feat(:,[featureMap.zerocrossrate,featureMap.shortTimeEnergy]) = [];
    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 = 97.69%

Visualize the confusion chart.

```
validationPredictions = kfoldPredict(partitionedModel);
figure(Units="normalized",Position=[0.4 0.4 0.4 0.4])
confusionchart(labels,validationPredictions,title="Validation Accuracy", ...
    ColumnSummary="column-normalized",RowSummary="row-normalized");
```

Validation Accuracy

True Class	1	3079		2	4	2	6			1		99.5%	0.5%
	2		1384	26		9		1		3	1	97.2%	2.8%
	3	1	17	4283	3	22	1			4	8	98.7%	1.3%
	4	2			2046	15	59	3		1	2	96.1%	3.9%
	5	4	2	27	7	2600	3	12		5	1	97.7%	2.3%
	6	25		4	103	28	2735	5		2	2	94.2%	5.8%
	7				1			2283		4	1	99.7%	0.3%
	8	1		3		5	2	3	930	13	11	96.1%	3.9%
	9		3	3	2	8	2	7	4	1817	11	97.8%	2.2%
	10		1	6		2	2	1	1	13	1430	98.2%	1.8%

98.9%	98.4%	98.4%	94.5%	96.6%	97.3%	98.6%	99.5%	97.5%	97.5%
1.1%	1.6%	1.6%	5.5%	3.4%	2.7%	1.4%	0.5%	2.5%	2.5%
1	2	3	4	5	6	7	8	9	10

Predicted Class

You can also use the Classification Learner (Statistics and Machine Learning Toolbox) app to try out and compare various classifiers with your table of features.

Testing the Classifier

In this section, you test the trained KNN classifier with speech signals from each of the 10 speakers to see how well it behaves with signals that were not used to train it.

Read files, extract features from the test set, and normalize them.

```

features = [];
labels = [];
numVectorsPerFile = [];
while hasdata(adsTest)
    [audioIn,dsInfo] = read(adsTest);

    % Take the same first portion of each recording to speed up code
    audioIn = audioIn(1:keepLen);

    feat = extract(afe,audioIn);

    isSpeech = feat(:,featureMap.shortTimeEnergy) > energyThreshold;
    isVoiced = feat(:,featureMap.zerocrossrate) < zcrThreshold;

    voicedSpeech = isSpeech & isVoiced;

    feat(~voicedSpeech,:) = [];
    
```

```

numVec = size(feats,1);
feats(:,[featureMap.zerocrossrate,featureMap.shortTimeEnergy]) = [];

label = repelem(dsInfo.Label,numVec);

numVectorsPerFile = [numVectorsPerFile,numVec];
features = [features;feats];
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));

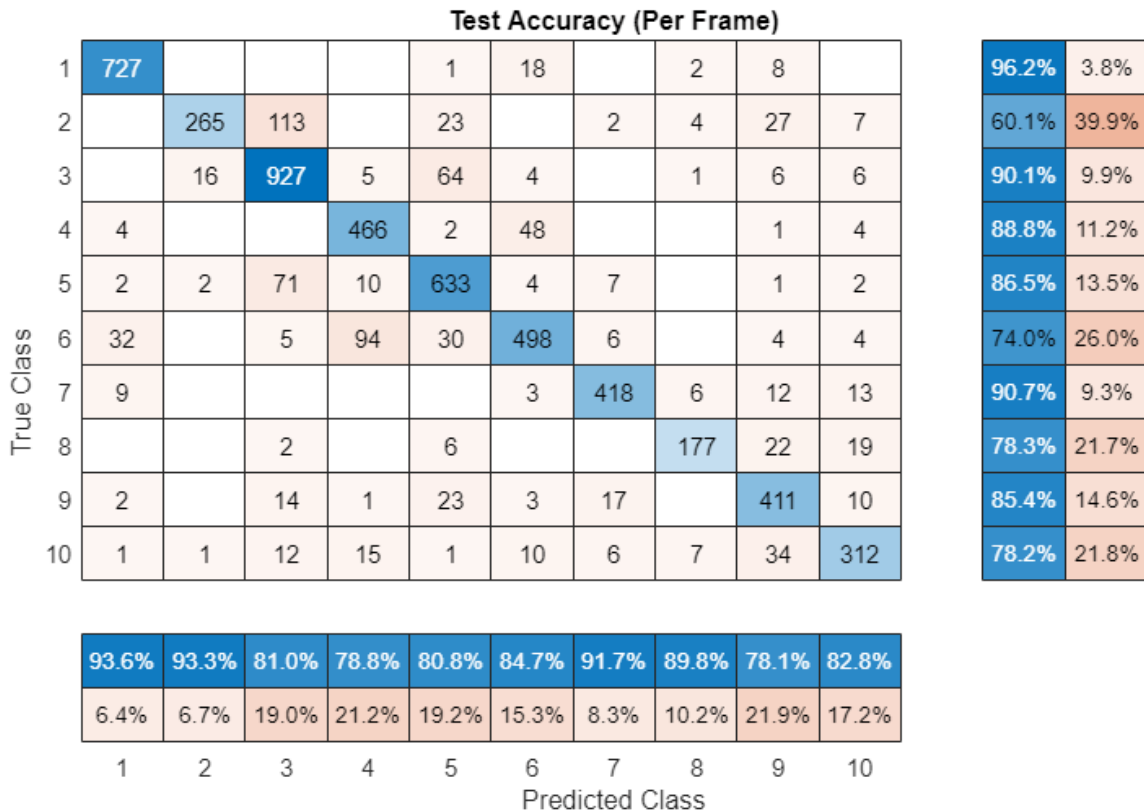
```

Visualize the confusion chart.

```

figure(Units="normalized",Position=[0.4 0.4 0.4 0.4])
confusionchart(labels(:),prediction,title="Test Accuracy (Per Frame)", ...
    ColumnSummary="column-normalized",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;

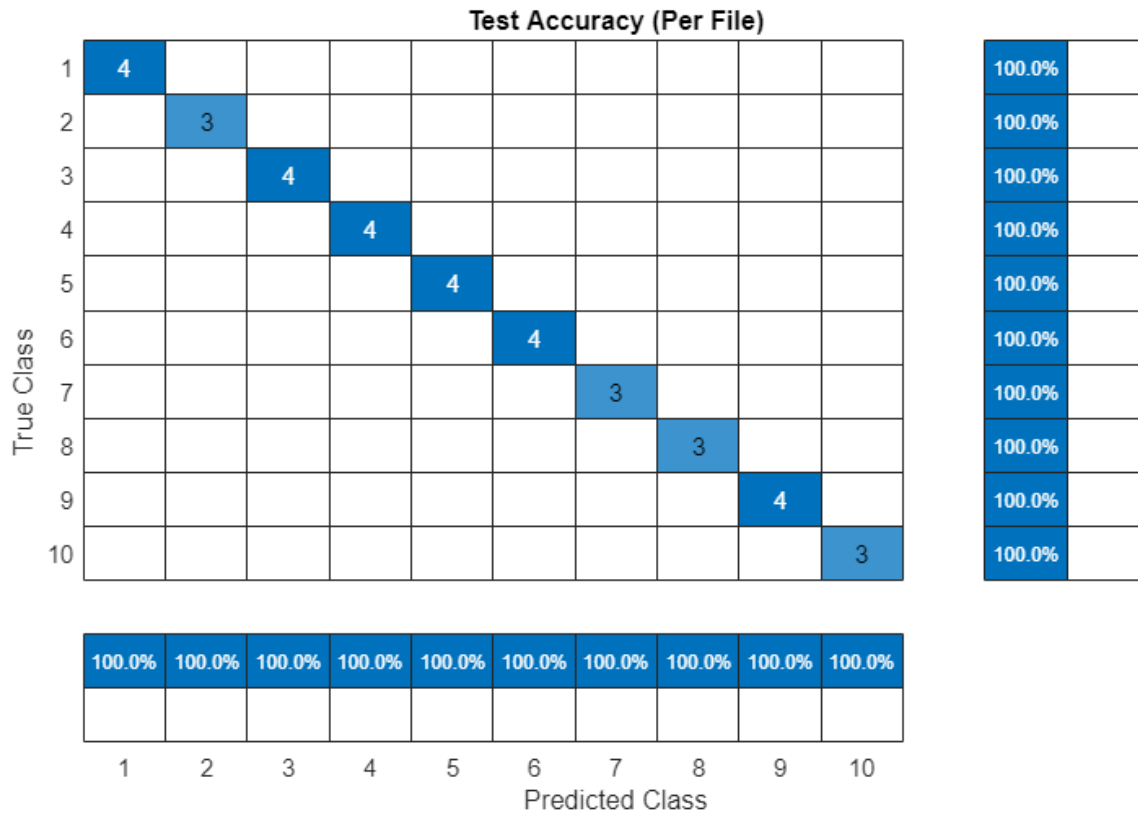
```

```

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])
confusionchart(adsTest.Labels,r2,title="Test Accuracy (Per File)", ...
    ColumnSummary="column-normalized",RowSummary="row-normalized");

```



The predicted speakers match the expected speakers for all files under test.

References

[1] Mozilla Common Voice Dataset

See Also

`pitch` | `mfcc`

Related Examples

- “Speaker Identification Using Custom SincNet Layer and Deep Learning” on page 1-723

Measure Audio Latency

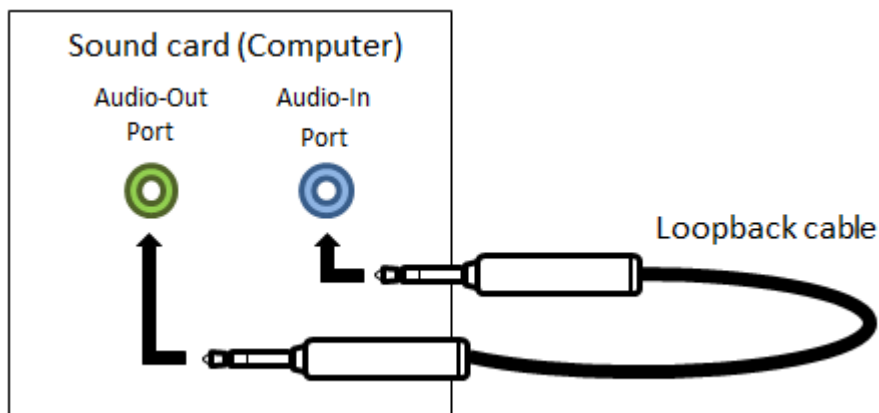
This example shows how to measure the latency of an audio device. The example uses `audioLatencyMeasurementExampleApp` which in turn uses `audioPlayerRecorder` along with a test signal and cross correlation to determine latency. To avoid disk access interference, the test signal is loaded into a `dsp.AsyncBuffer` 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 =
  1×5 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 =
  3×5 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
```

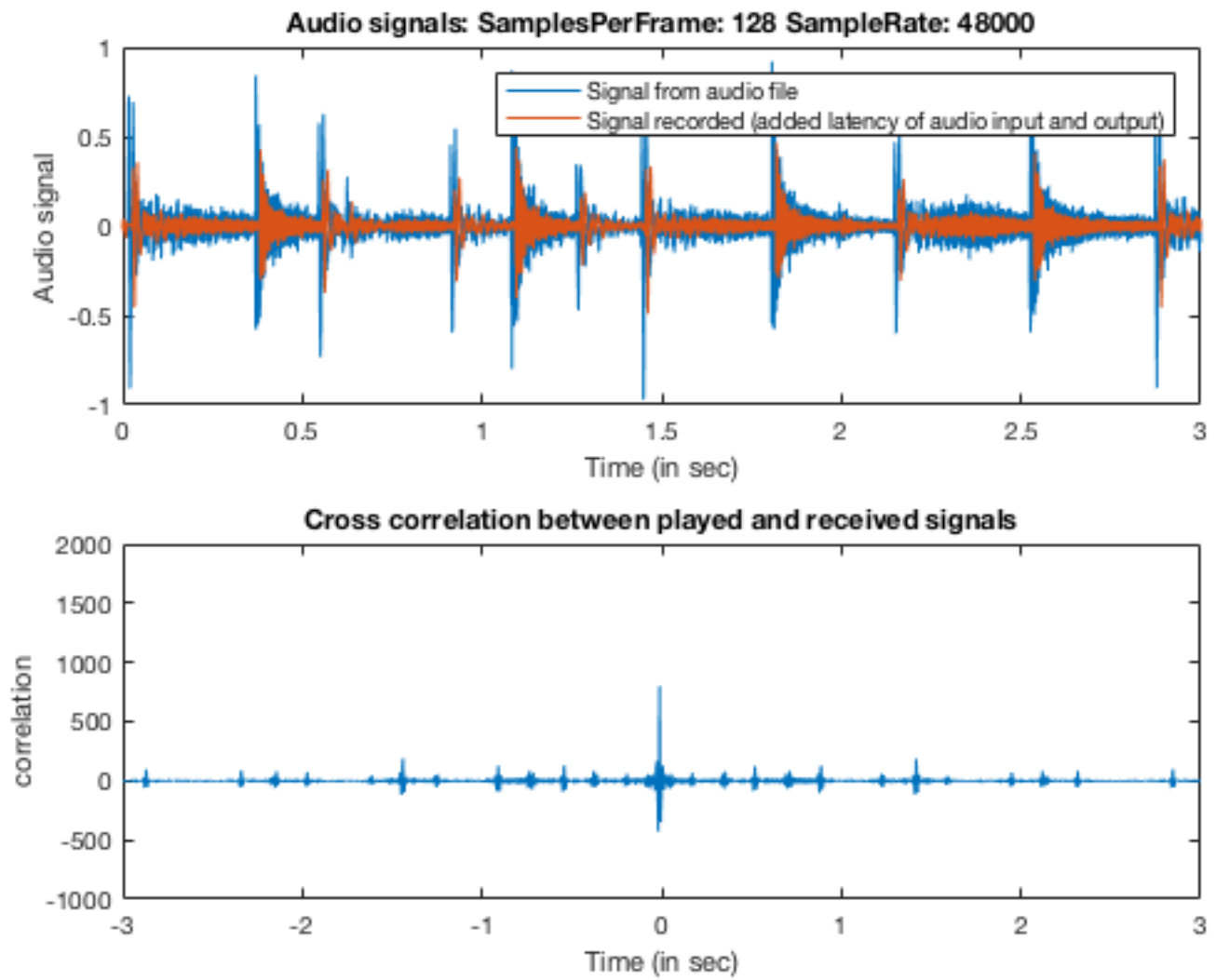
Plotting the Original and Recorded Signal

%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

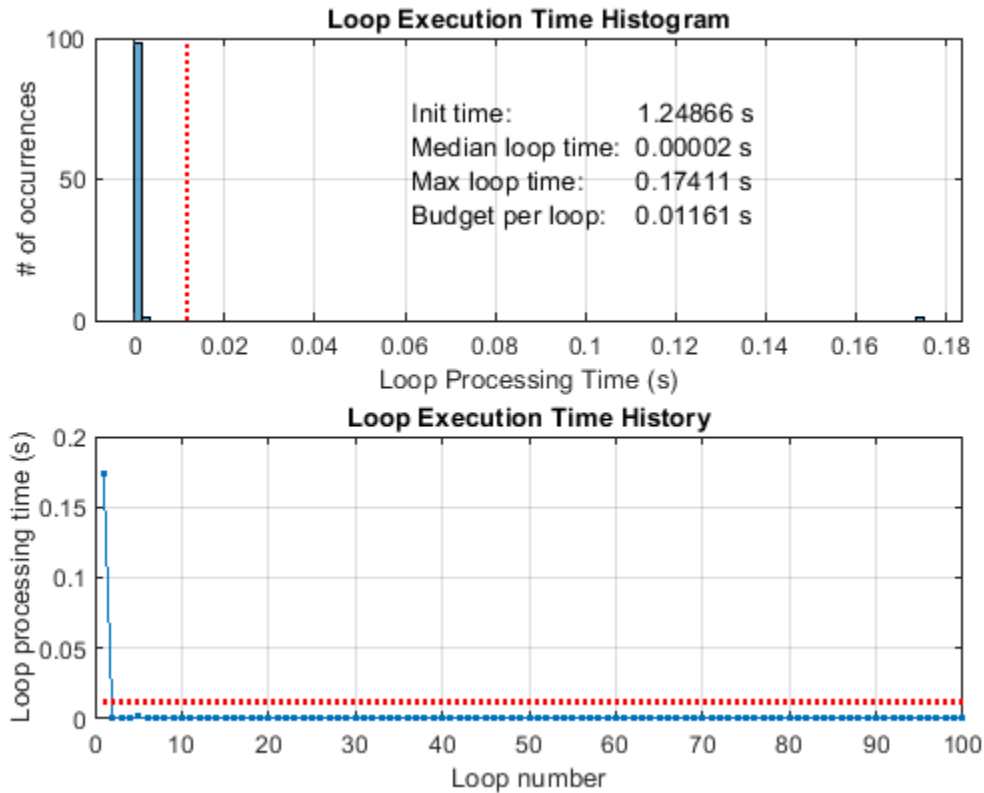
In this example, you measure performance of an eighth-order notch filter, implemented using `dsp.BiquadFilter`.

`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:

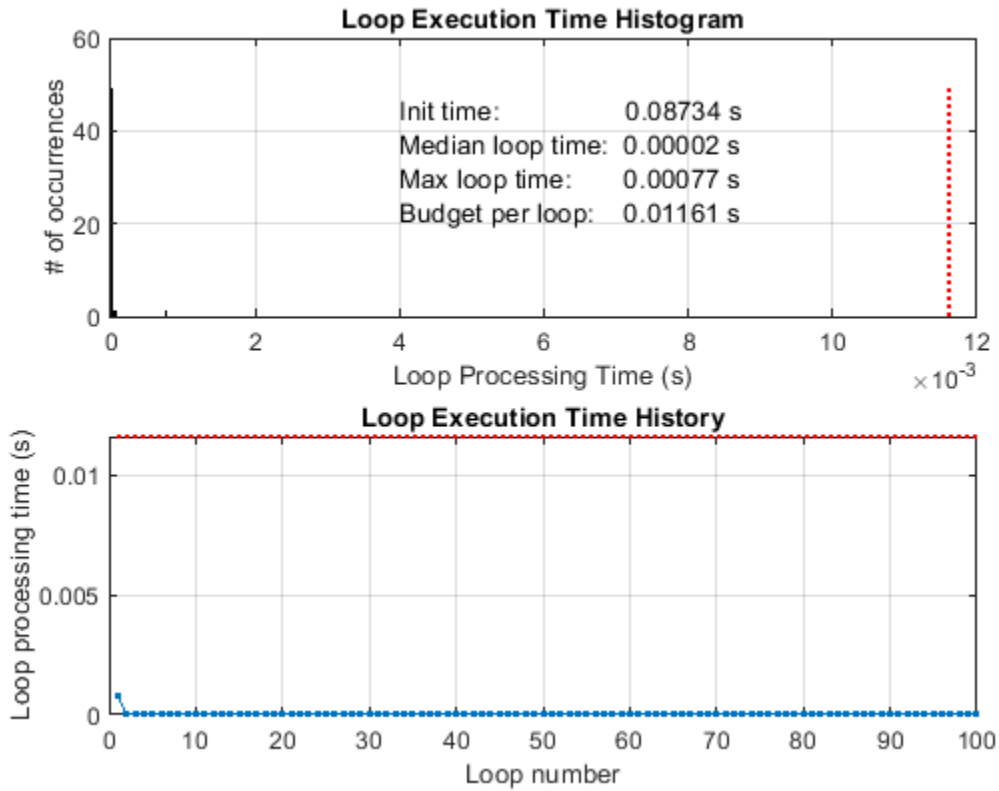
```
helperAudioLoopTimerExample;
```



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 you call the `dsp.BiquadFilter` and `dsp.AudioFileReader` objects. The first call to the object triggers the execution of one-time tasks that do not depend on the inputs, 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 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.

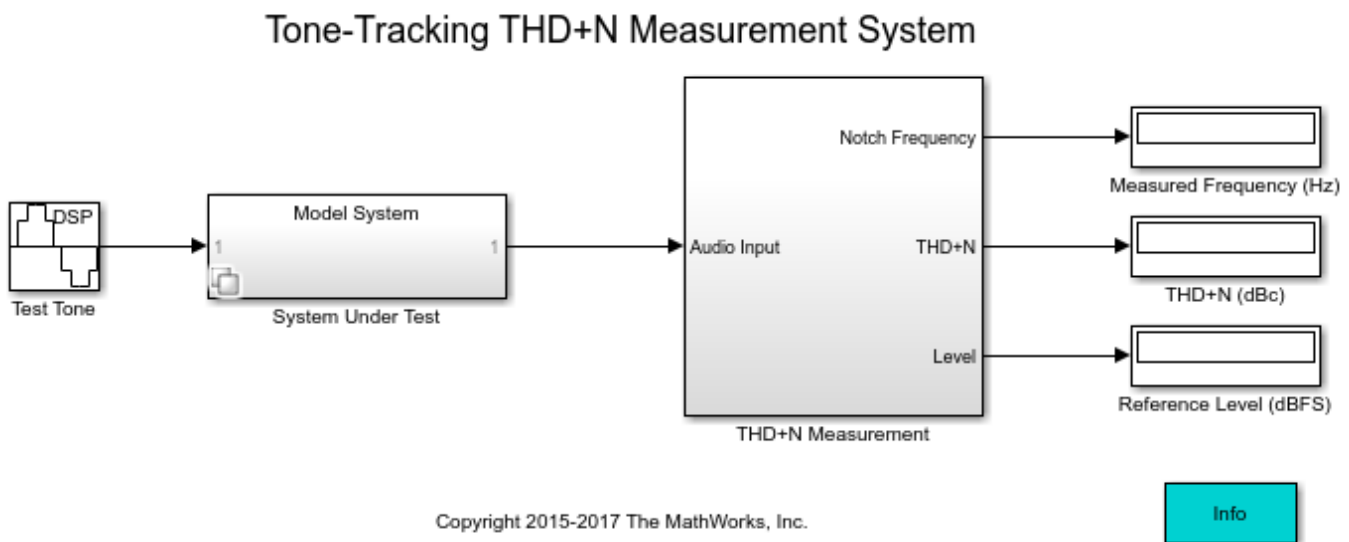
Total Harmonic Distortion and Noise (THD+N) quantifies the sum of these two distortions. It is defined as the root mean square (RMS) level of all harmonics and noise components over a specified bandwidth. The signal level is also specified as a reference.

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.

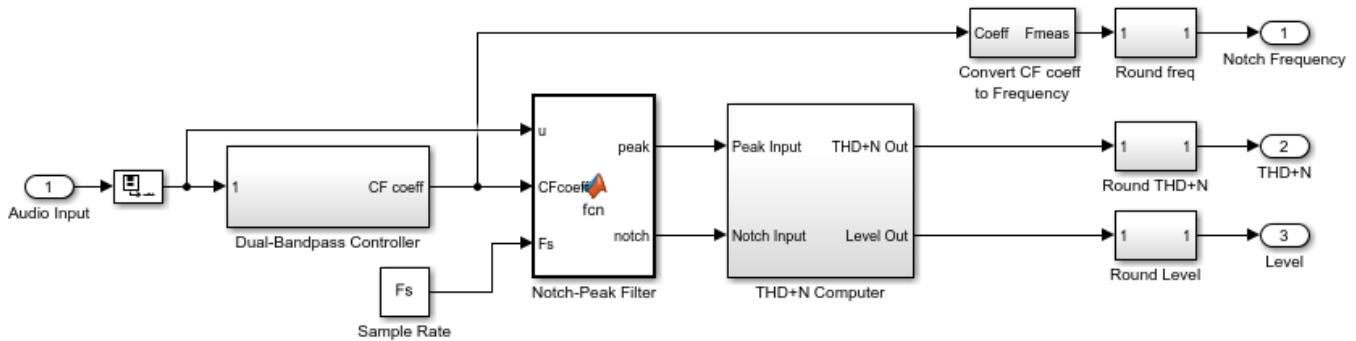
This example follows the AES17-1998(r2004) [1] standard for THD+N measurement. The standard recommends a 997 Hz frequency sine wave. It also recommends a notch filter having Q between 1 and 5 for filtering out the sine wave from recorded signal. A Q value of 5 is used in this example.



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

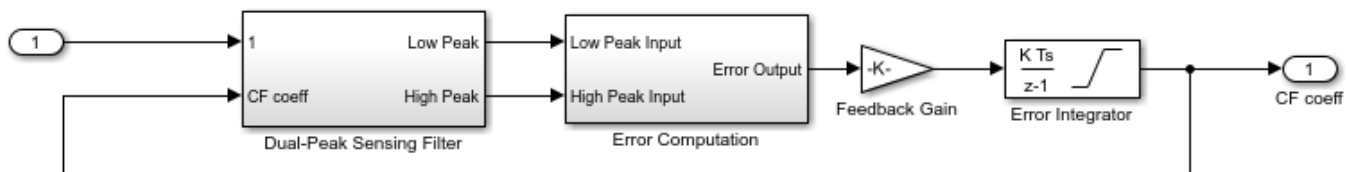
of 997 Hz. The subsystem **System Under Test** is a variant subsystem. By default, it selects a non-linear model implemented in Simulink for measuring the THD+N. To perform the measurement on your machine's audio input and output device, set the SUT variable in base workspace to THDNDemoSUT.AudioHardware.

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.

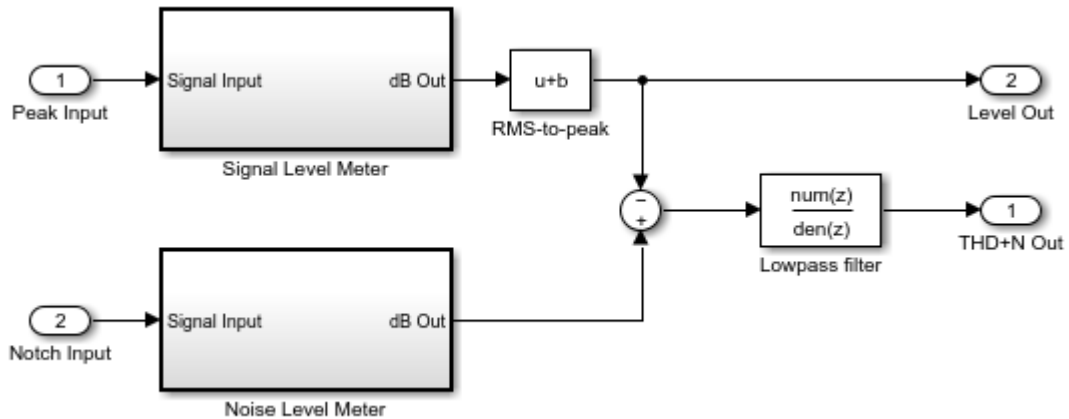


Notch-Peak Filter

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].

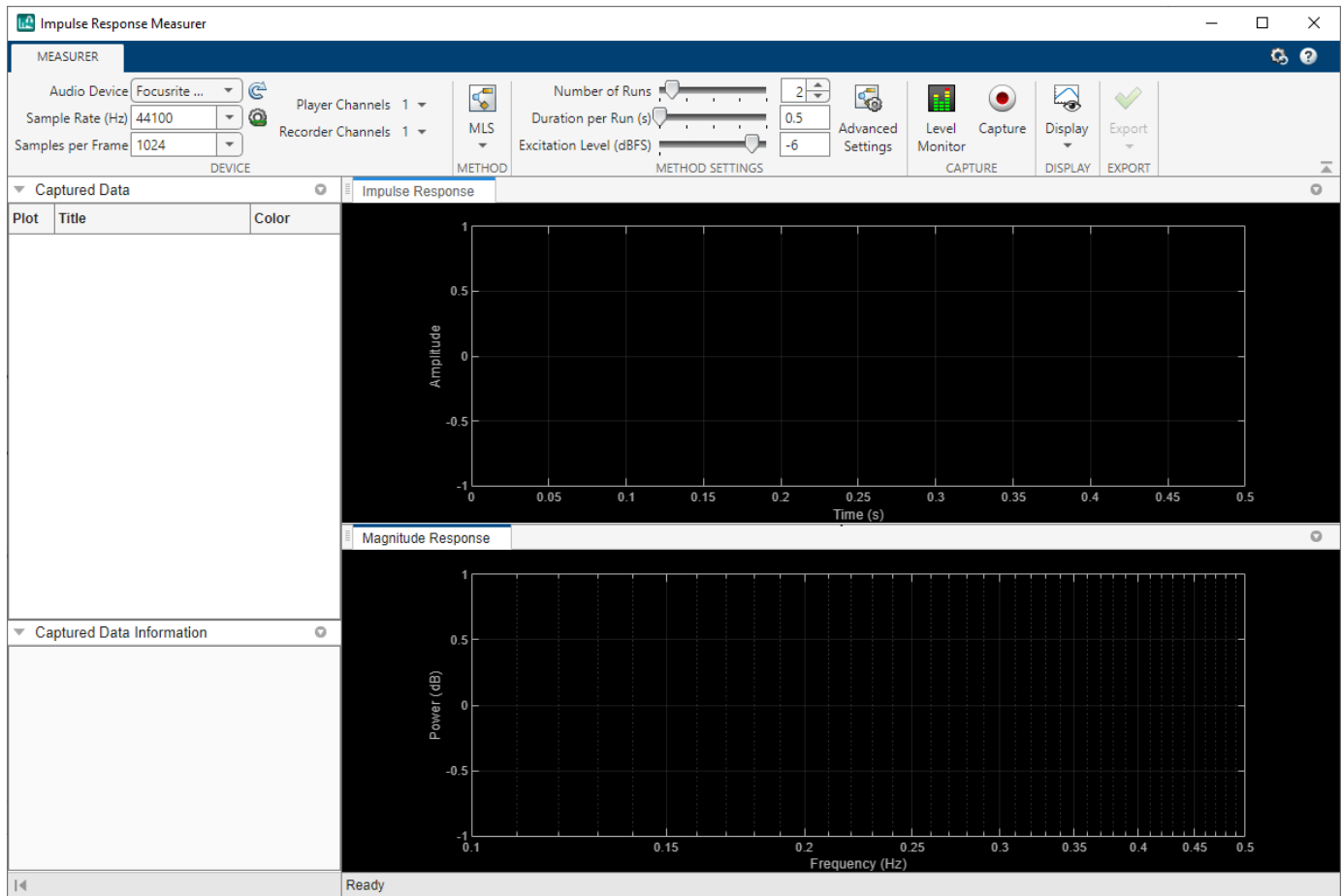
The Maximum-Length-Sequence (MLS) technique is based upon the excitation of the acoustical space by a periodic pseudo-random signal. The impulse response is obtained by circular cross-correlation between the measured output and the test tone (MLS sequence). For more details, see [2].

In this example, you use the MLS measurement technique.

Acquire Impulse Response of Room

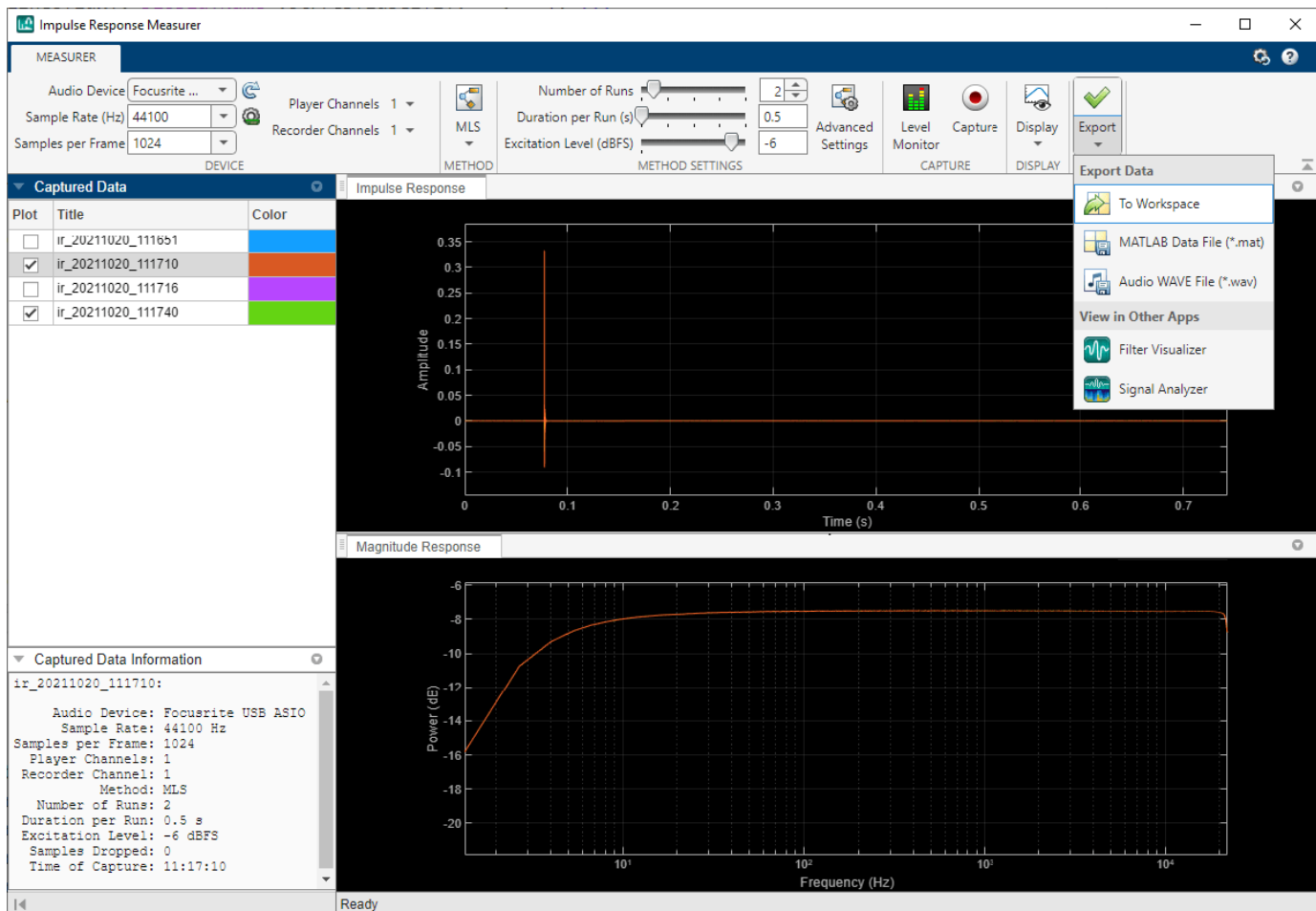
1. To open the app, at the MATLAB® command prompt, enter:

```
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:
```

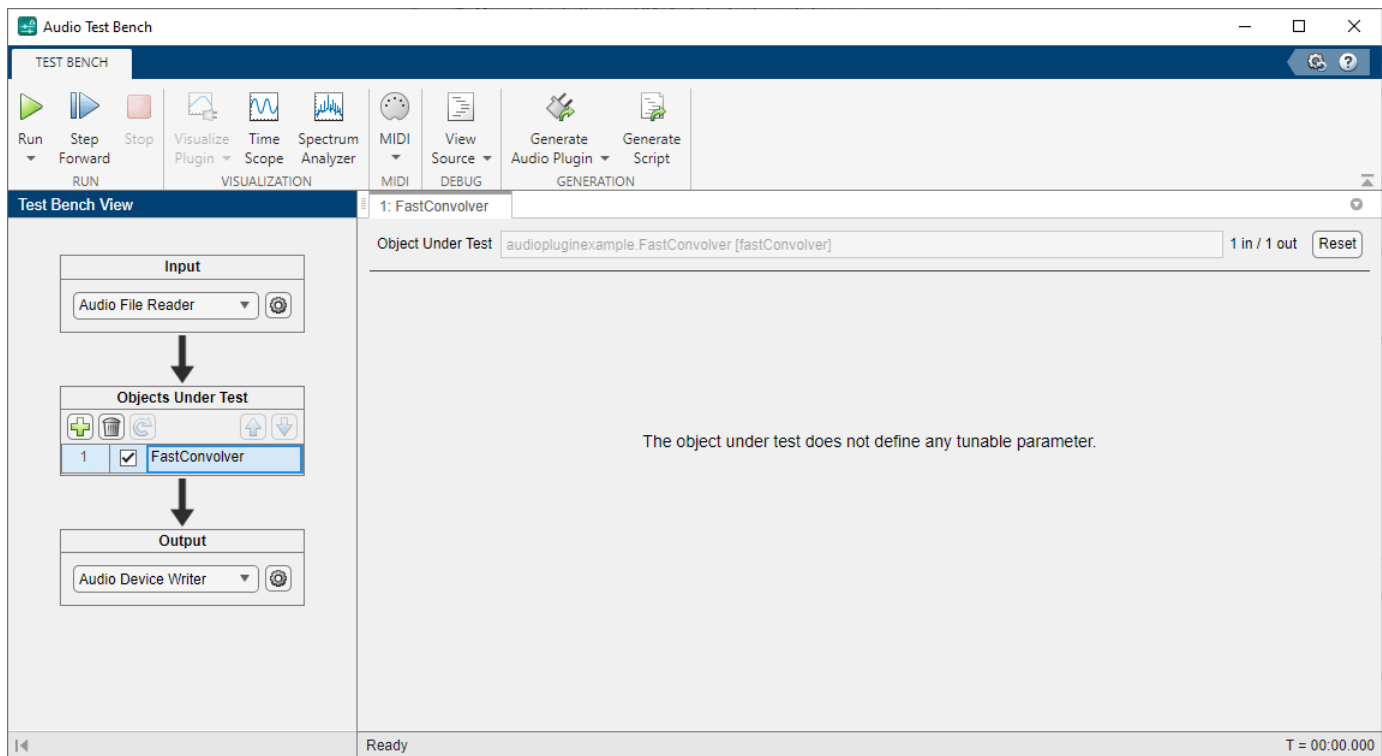
```
ImpulseResponse: [0 0 -3.0518e-05 3.0518e-05 0 0 0 3.0518e-05 0 0 0 3.0518e-05 0 0 0 0 0 0 0  
PartitionSize: 1024
```

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`.

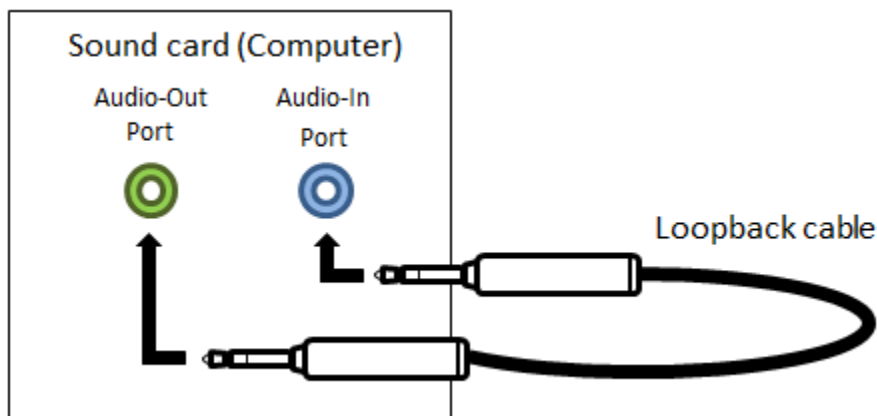
Description of FR Measurement Techniques

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 Focusrite Scarlett 2i2 audio device with a 48 kHz sampling rate.

```
sampleRate = 48e3;
device = "Focusrite 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 `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 = spectrumAnalyzer( ...  
    SampleRate=sampleRate, ...  
    RBWSource="property",RBW=RBW, ...  
    AveragingMethod="exponential", ...  
    ForgettingFactor=0, ...  
    FrequencySpan="start-and-stop-frequencies",...  
    StartFrequency=0, ...  
    StopFrequency=sampleRate/2, ...  
    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)
```

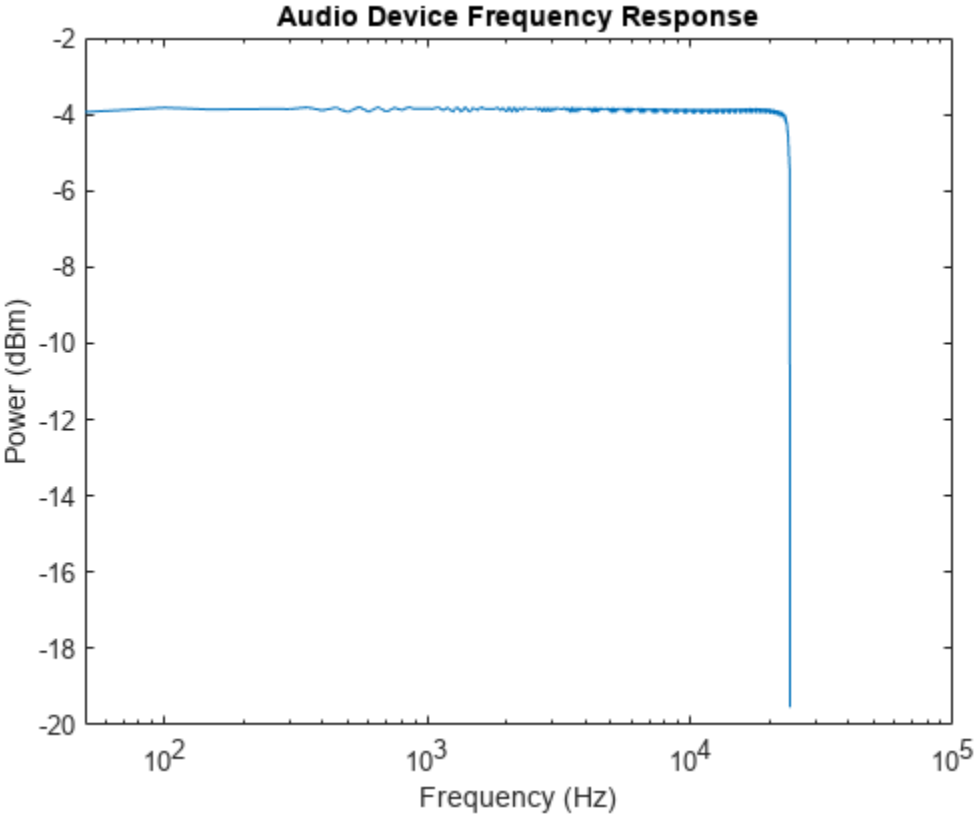


Frequency Response Measurement Results

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.

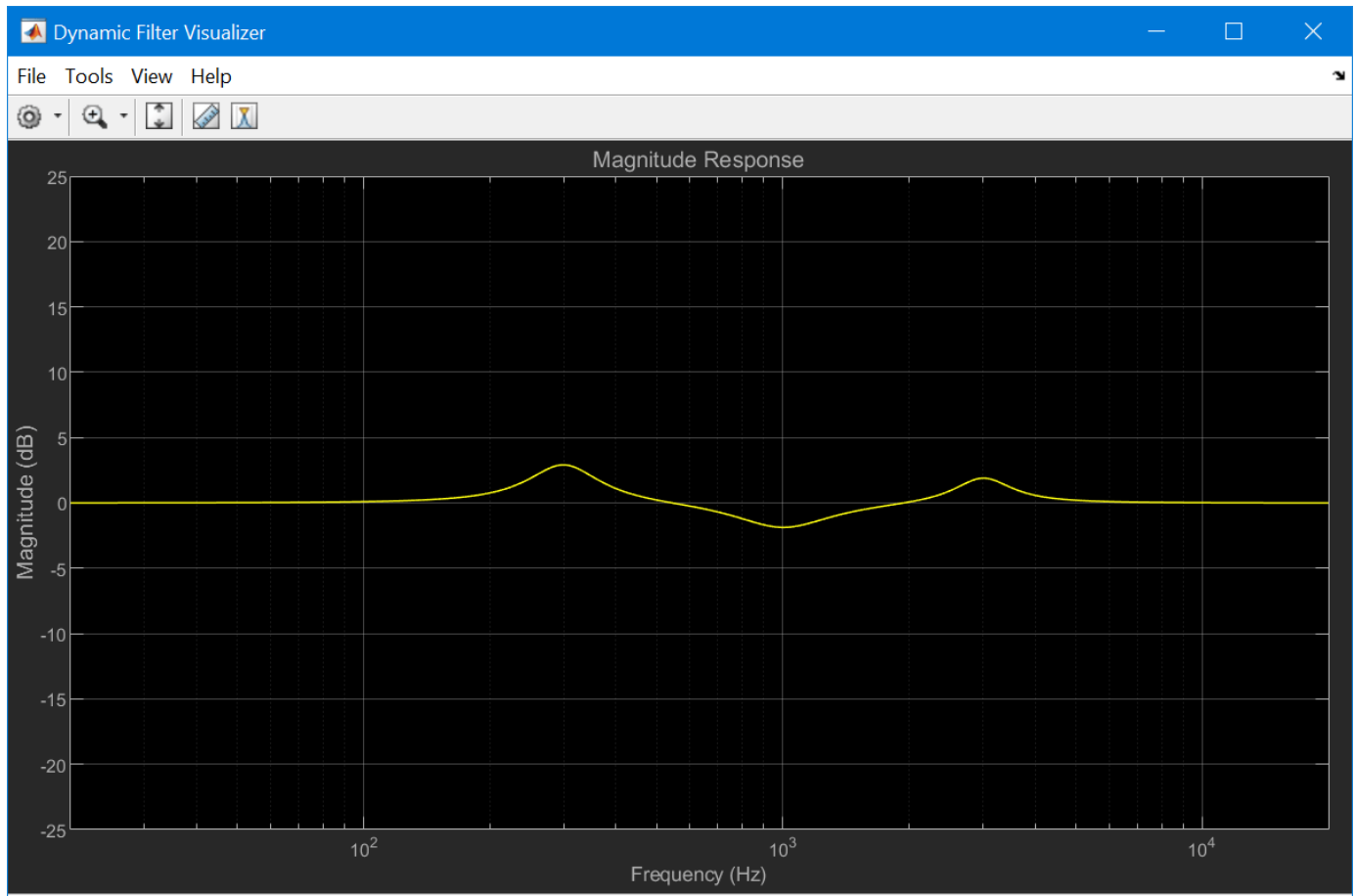
Generating Code and Building an Executable File

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.



The screenshot shows a software window titled "Parametric Equalizer Tuning". The interface is organized into three rows, each representing a different filter band. Each row contains three parameters: Center Frequency, Quality Factor, and Peak Gain (dB). Each parameter is controlled by a horizontal slider and a corresponding numerical input field. At the bottom of the window, there are three buttons: "Reset", "Pause Simulation", and "Stop Simulation".

Parameter	Value
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

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.

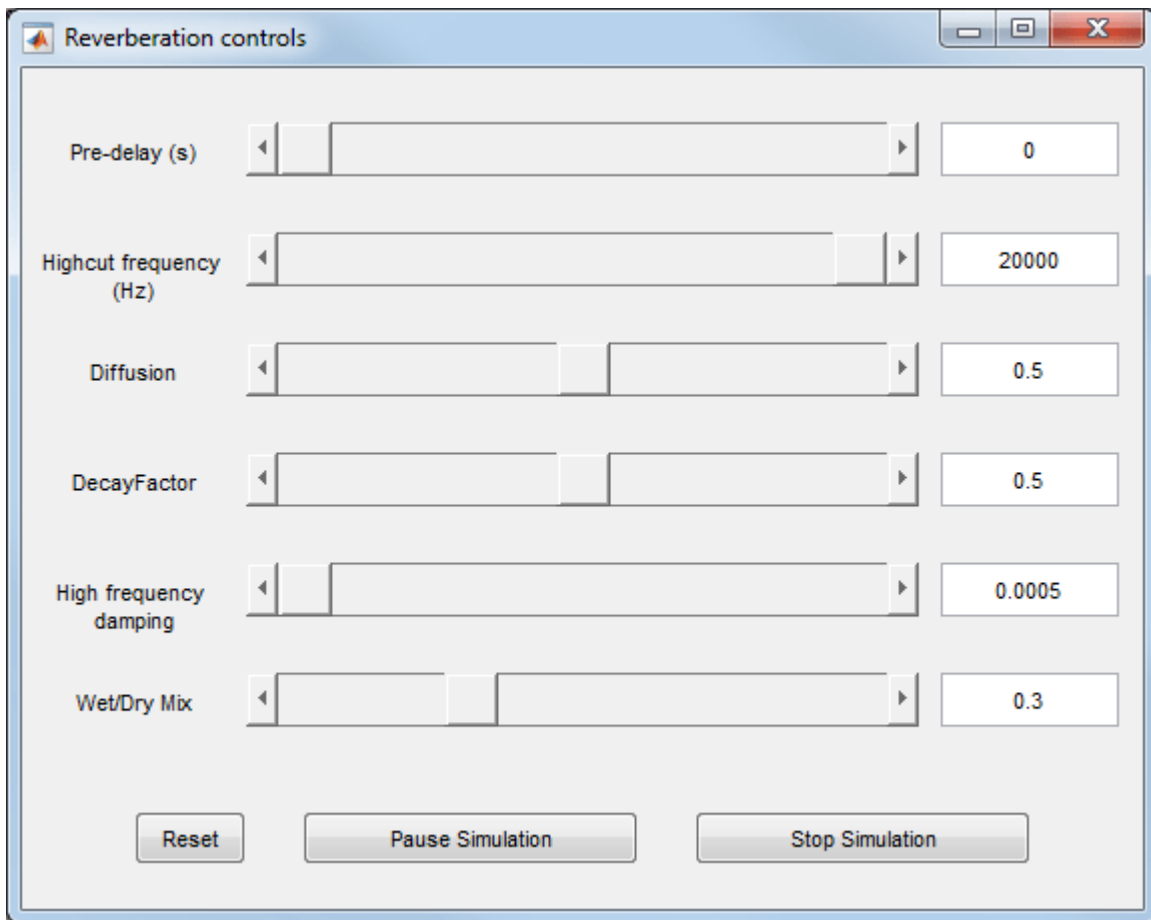
MATLAB Simulation

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

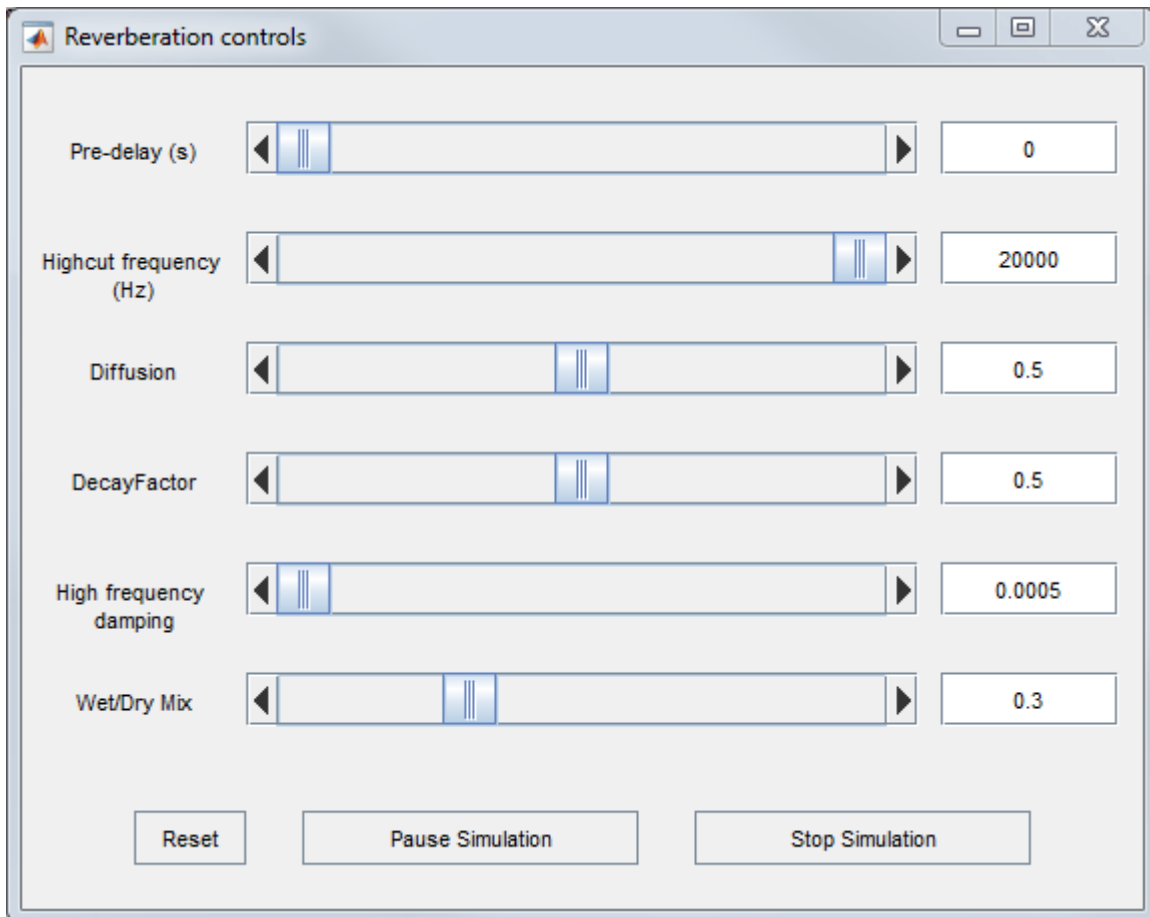
Use the system command to run the generated standalone application. Note that running the standalone application using the system command uses the current MATLAB environment and any library files needed from this installation of MATLAB. To deploy this application on a machine which does not have MATLAB installed, refer to “About the MATLAB Runtime” (MATLAB Compiler).

```

if ismac
    status = system(fullfile('audioReverberationCompilerExampleApp.app', ...
        '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 Single-Band 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.

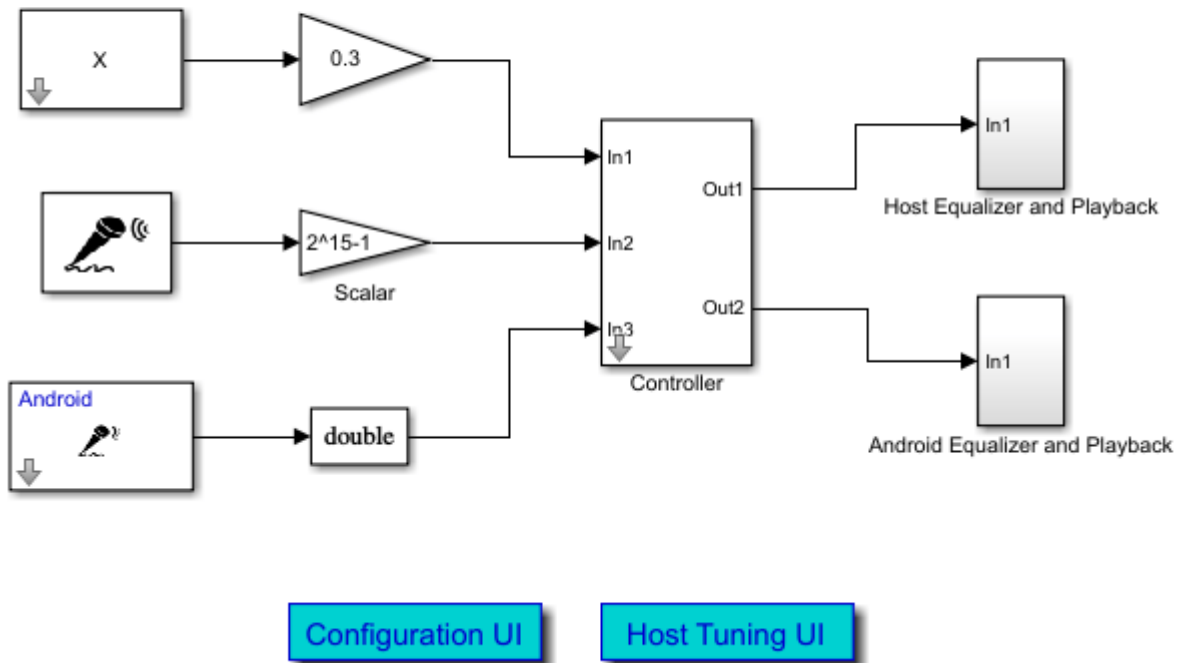
Required Hardware

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

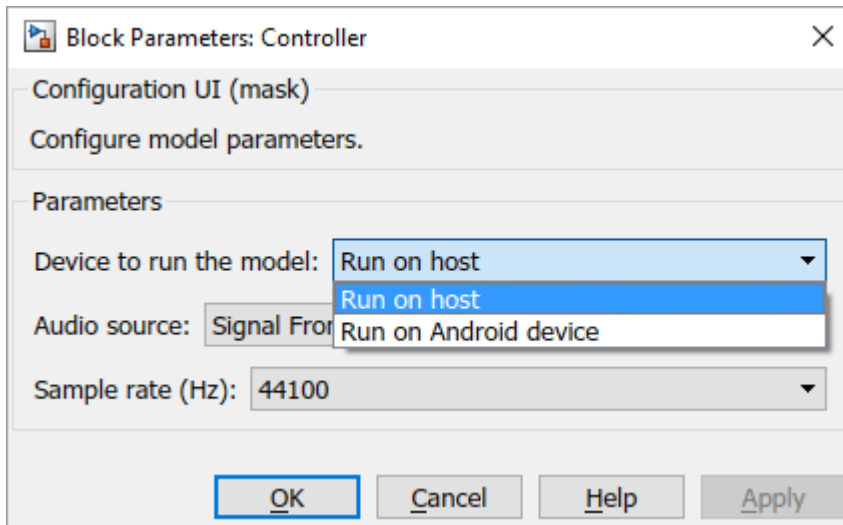
Model Setup

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.

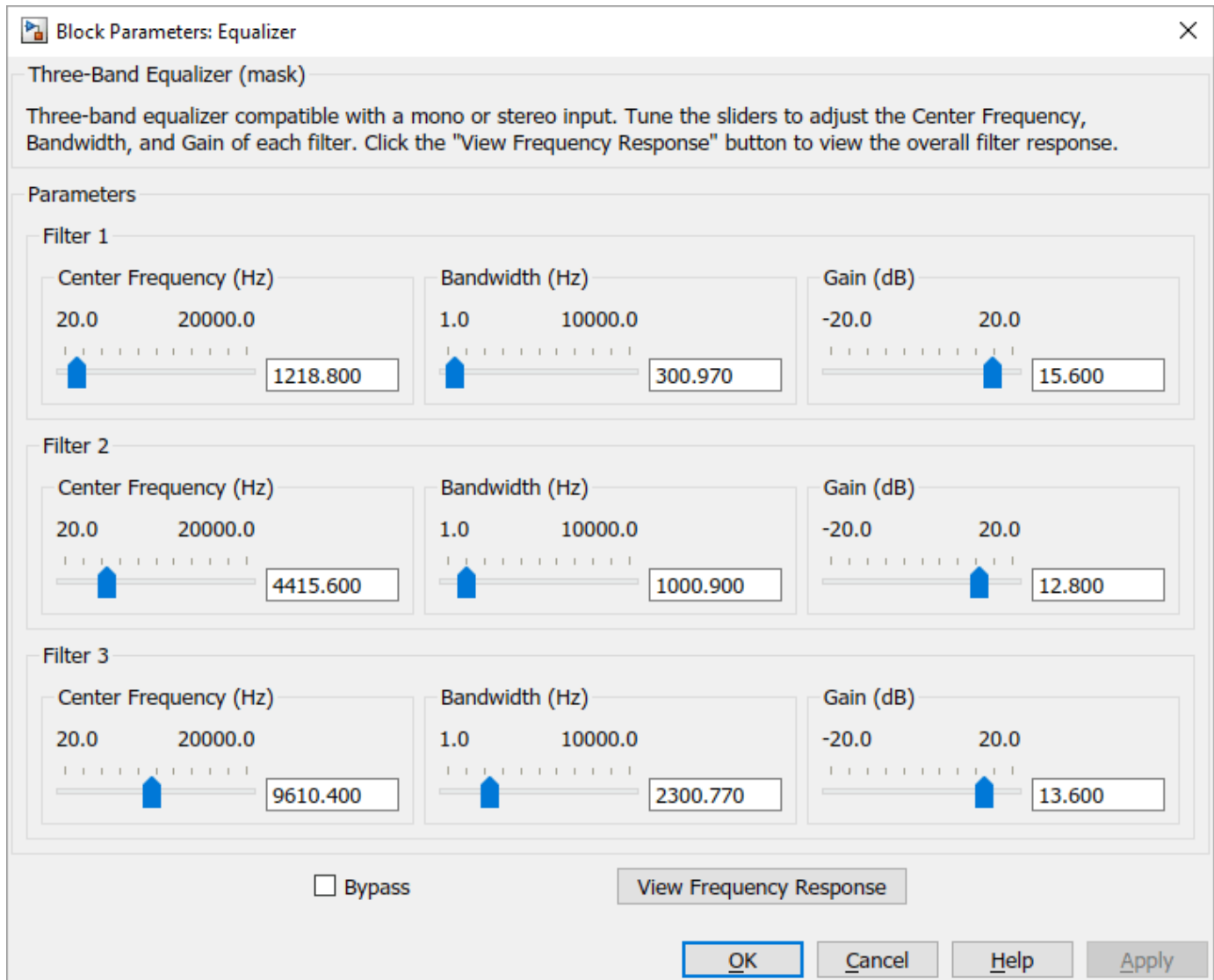
Configuration UI:



Run Model on the Host Computer

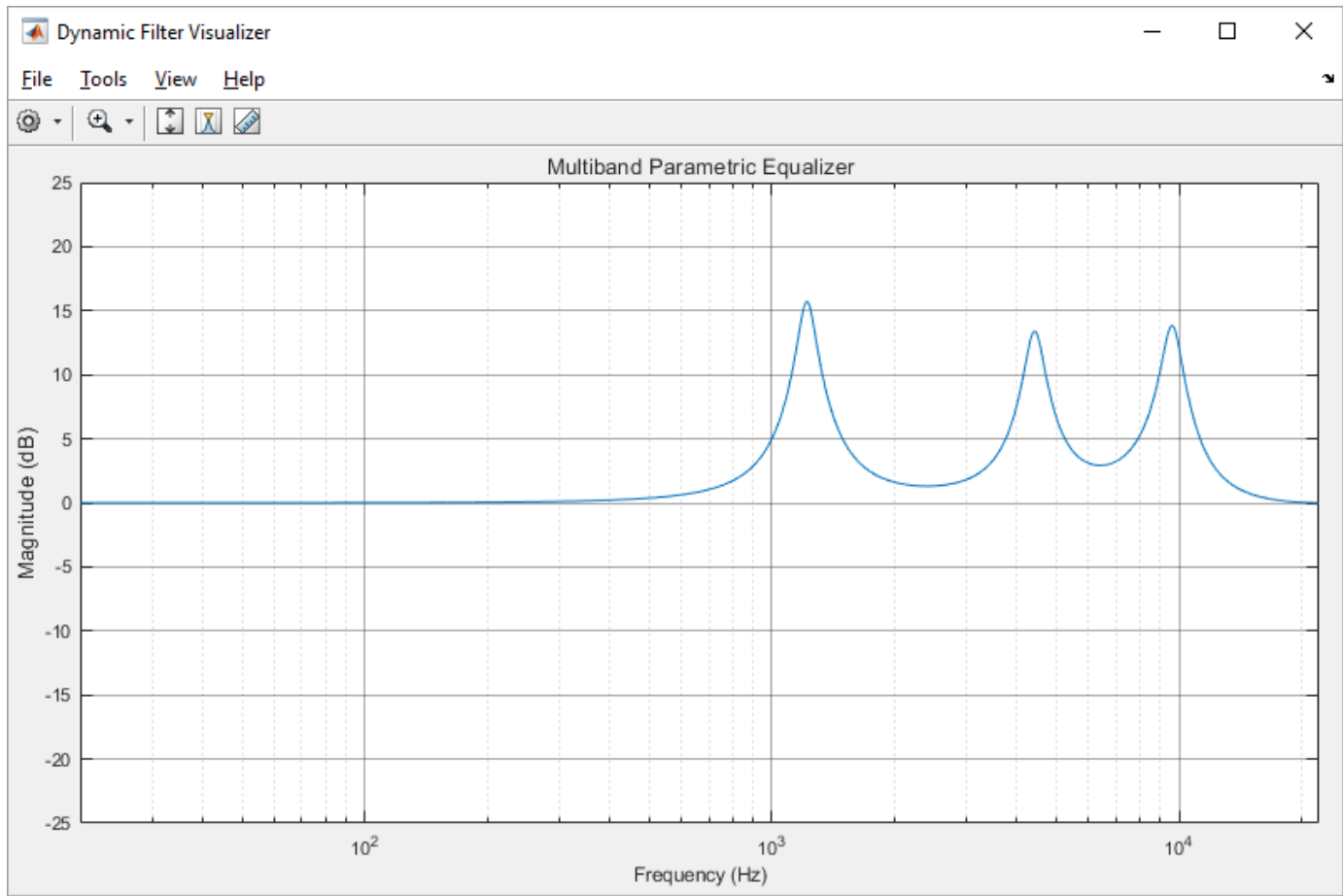
When you choose to run the model on the host computer, a UI designed to interact with the simulation is provided and can be opened by clicking Host Tuning UI.

Host Tuning UI:



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.

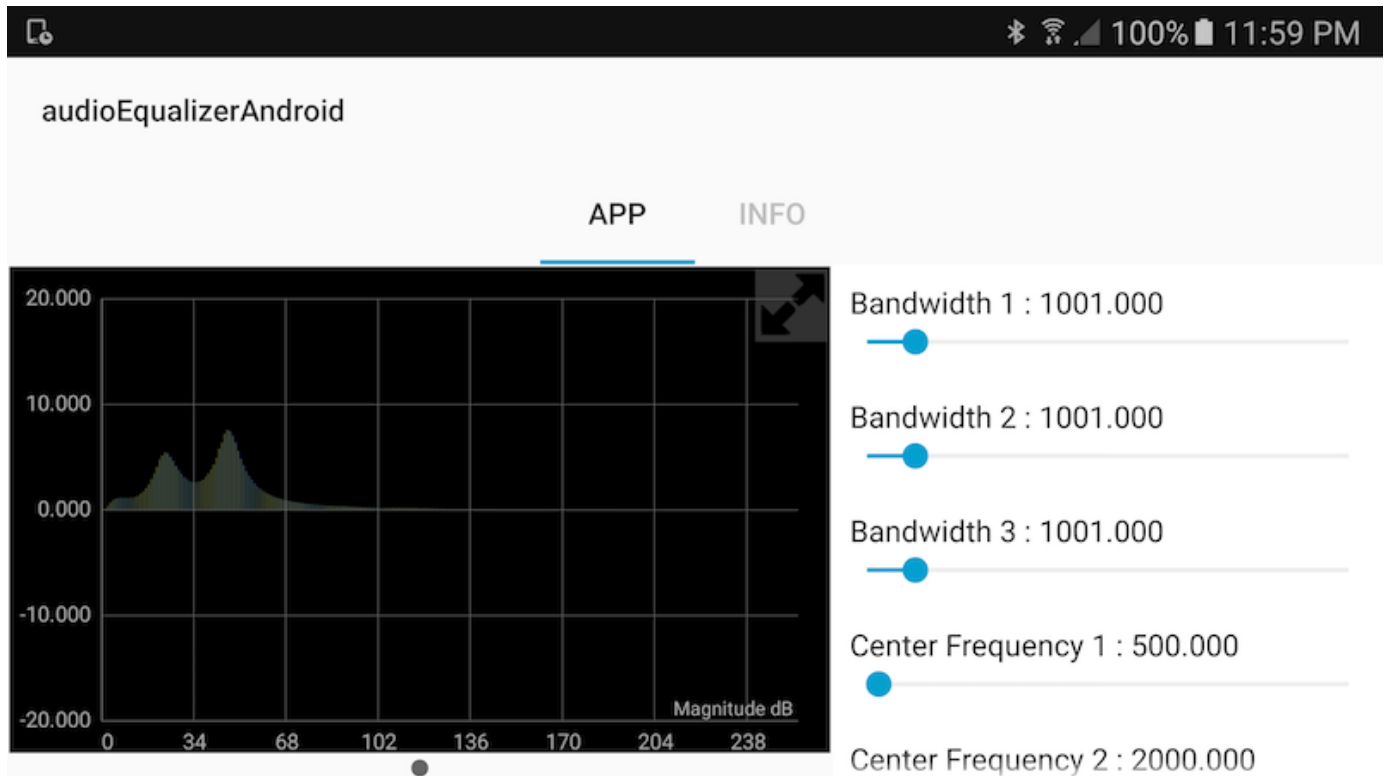


Run Model on an Android Device

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 Single-Band 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.

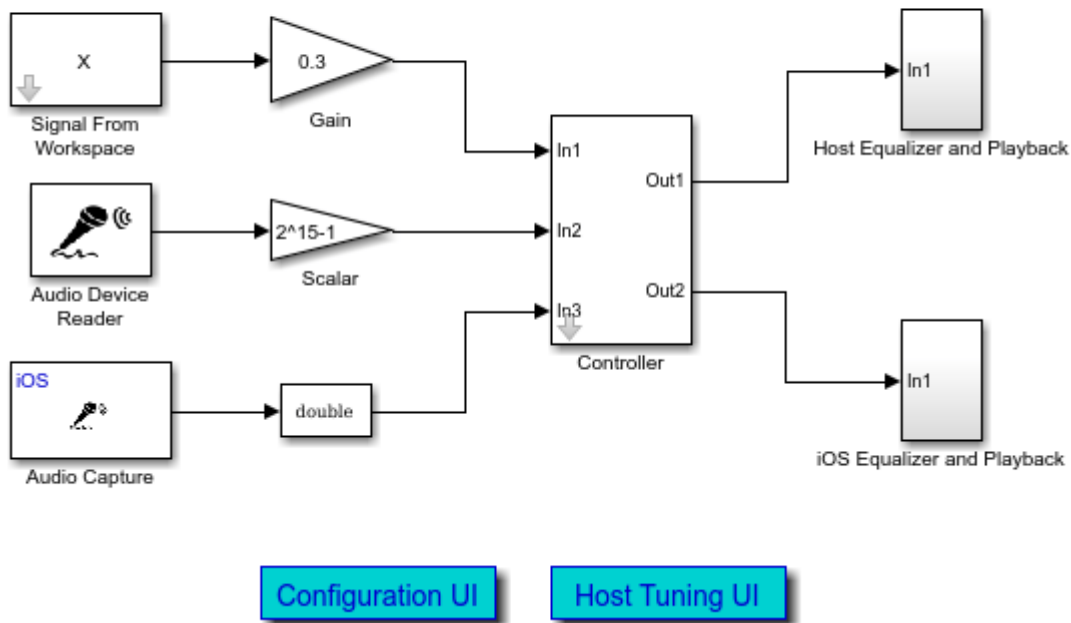
Required Hardware

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

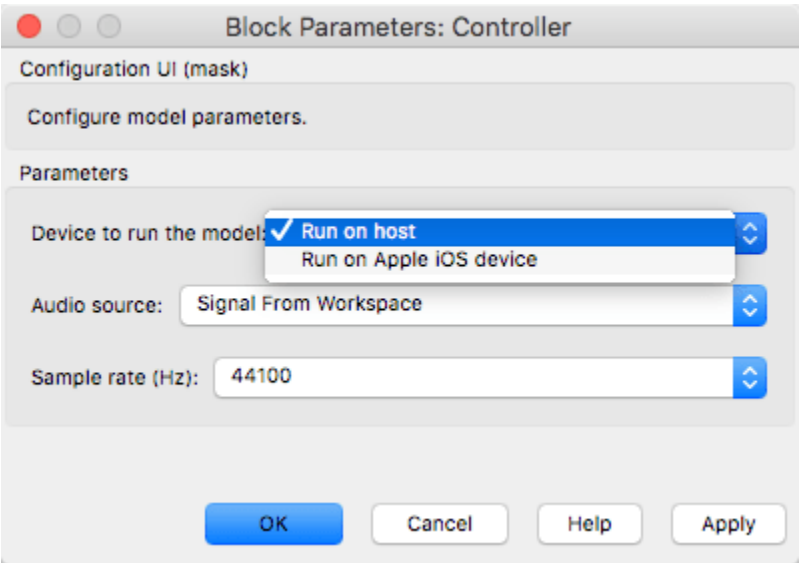
Model Setup

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.

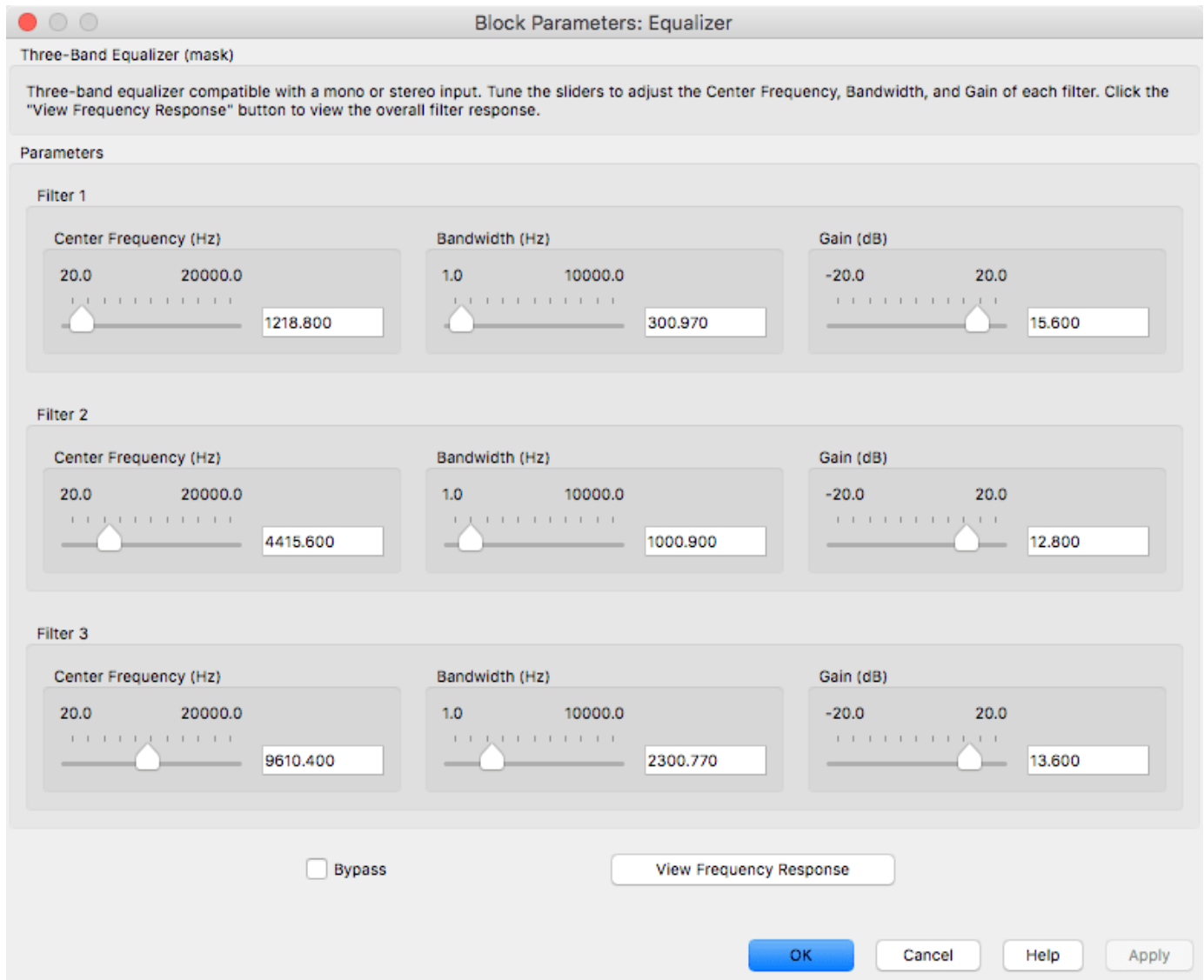
Configuration UI:



Run Model on the Host Computer

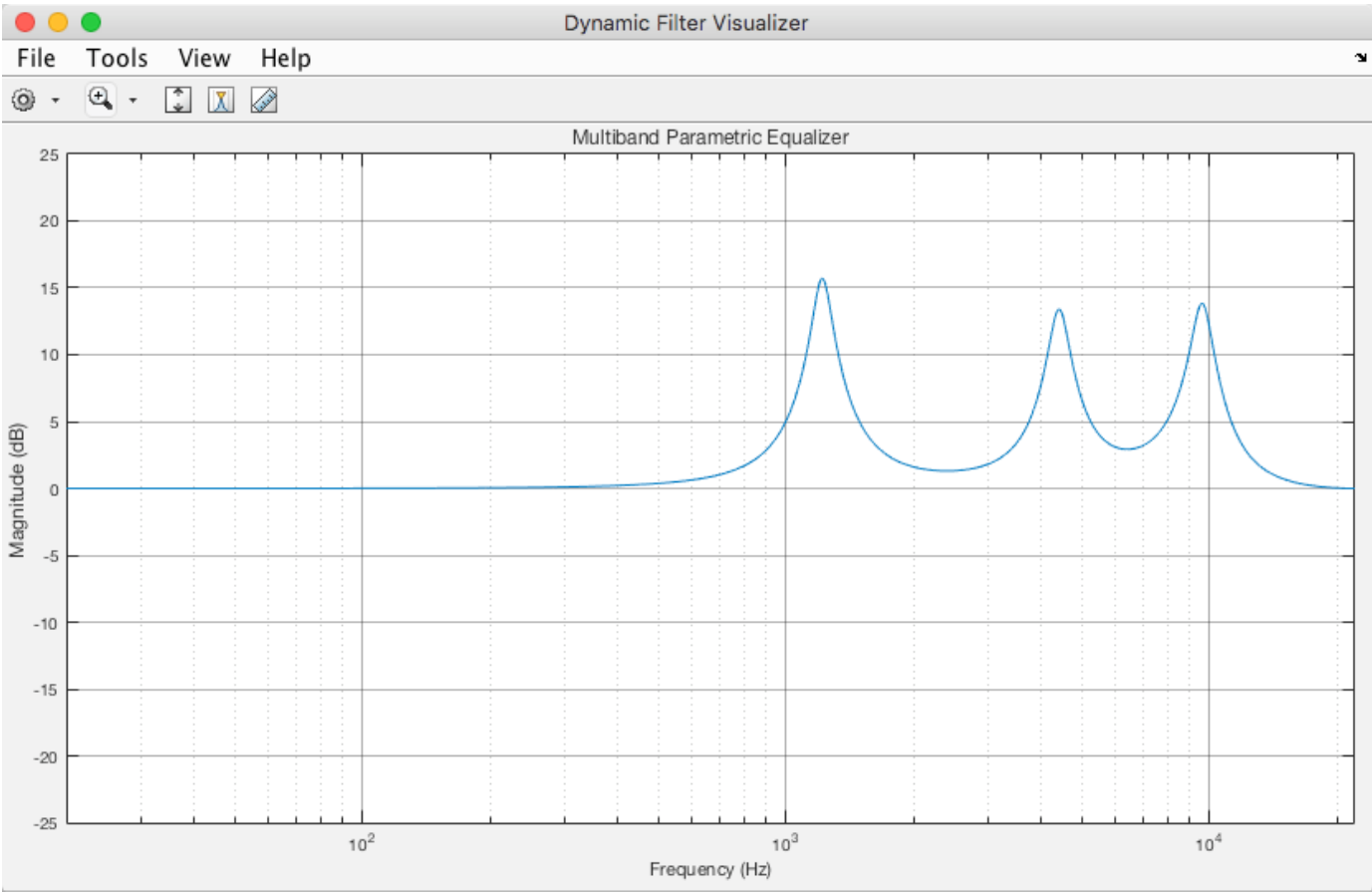
When you choose to run the model on the host computer, a UI designed to interact with the simulation is provided and can be opened by clicking Host Tuning UI.

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:

iPad 11:14 AM

audioequalizeriOS

Bandwidth 1: 1000.000000

Bandwidth 2: 2900.000000

Bandwidth 3: 940.000000

Center Frequency 1: 4655.000000

Center Frequency 2: 9970.000000

Center Frequency 3: 16125.000000

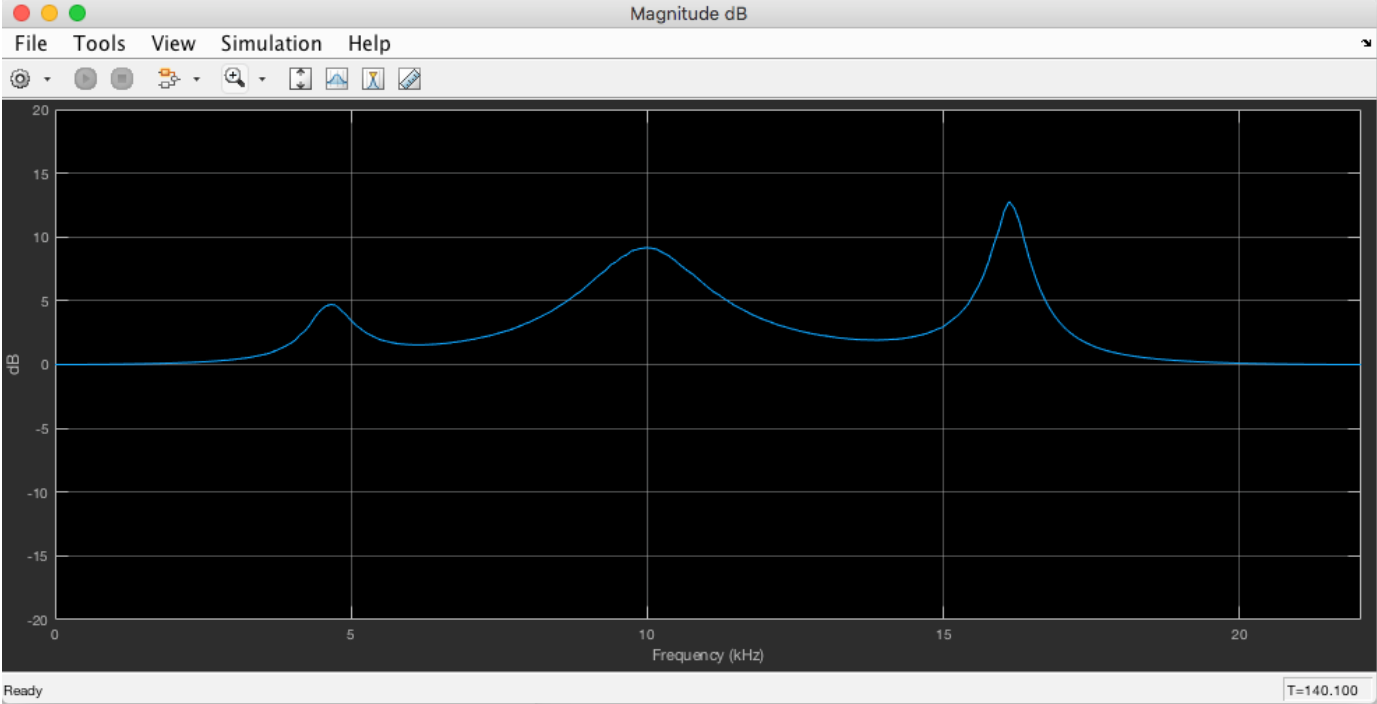
Gain 1: 4.300000

Gain 2: 9.000000

Gain 3: 12.300000

The screenshot displays the 'audioequalizeriOS' application interface. At the top, the status bar shows 'iPad' with a Wi-Fi signal icon and the time '11:14 AM'. The app title 'audioequalizeriOS' is centered below the status bar. The interface features three distinct frequency bands, each with three adjustable sliders. The sliders are represented by white circular knobs on a horizontal blue track. The values for each slider are displayed in text above the track. The first band has a Bandwidth of 1000.000000, a Center Frequency of 4655.000000, and a Gain of 4.300000. The second band has a Bandwidth of 2900.000000, a Center Frequency of 9970.000000, and a Gain of 9.000000. The third band has a Bandwidth of 940.000000, a Center Frequency of 16125.000000, and a Gain of 12.300000. At the bottom of the screen, a 'Reset' button is partially visible.

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.

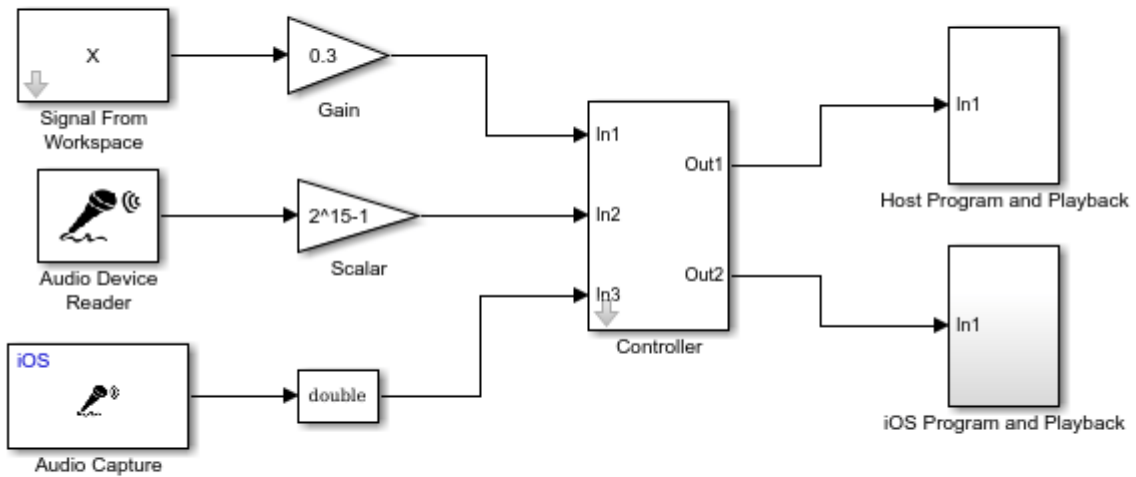
Required Hardware

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

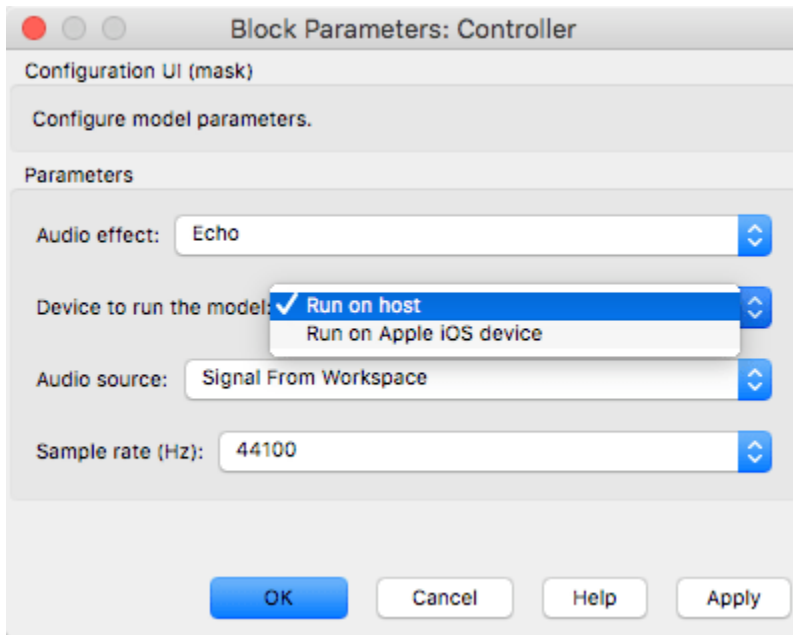


Configuration UI

Host Tuning UI

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

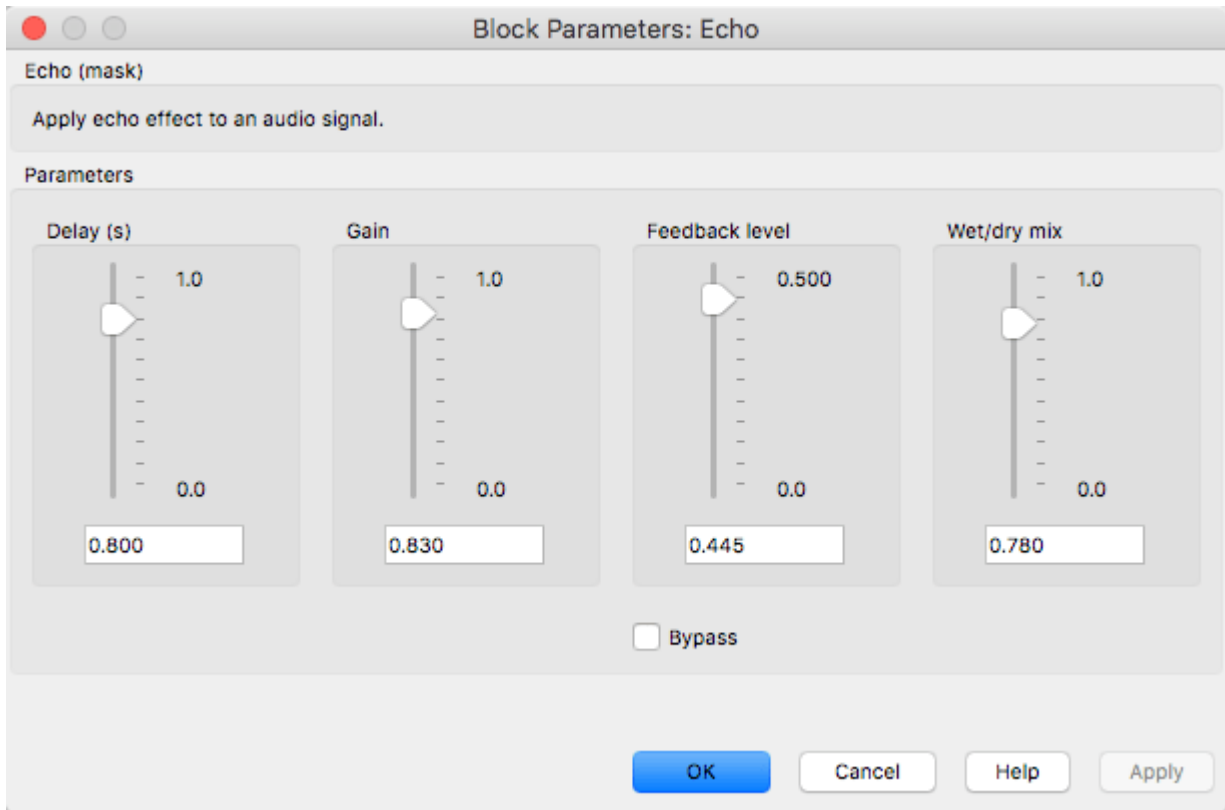
The echo effect has four tunable parameters that can be modified while the model is running:

- 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:

Delay/s: 0.500000

A horizontal slider control with a white circular knob positioned at the center of a blue track.

Feedback Level: 0.350000

A horizontal slider control with a white circular knob positioned approximately 35% of the way from the left end of a blue track.

Gain/dB: 0.000000

A horizontal slider control with a white circular knob positioned at the center of a blue track.

Wet/Dry Mix: 0.500000

A horizontal slider control with a white circular knob positioned at the center of a blue track.

Bypass

Mute

Audio Effect: Reverberation

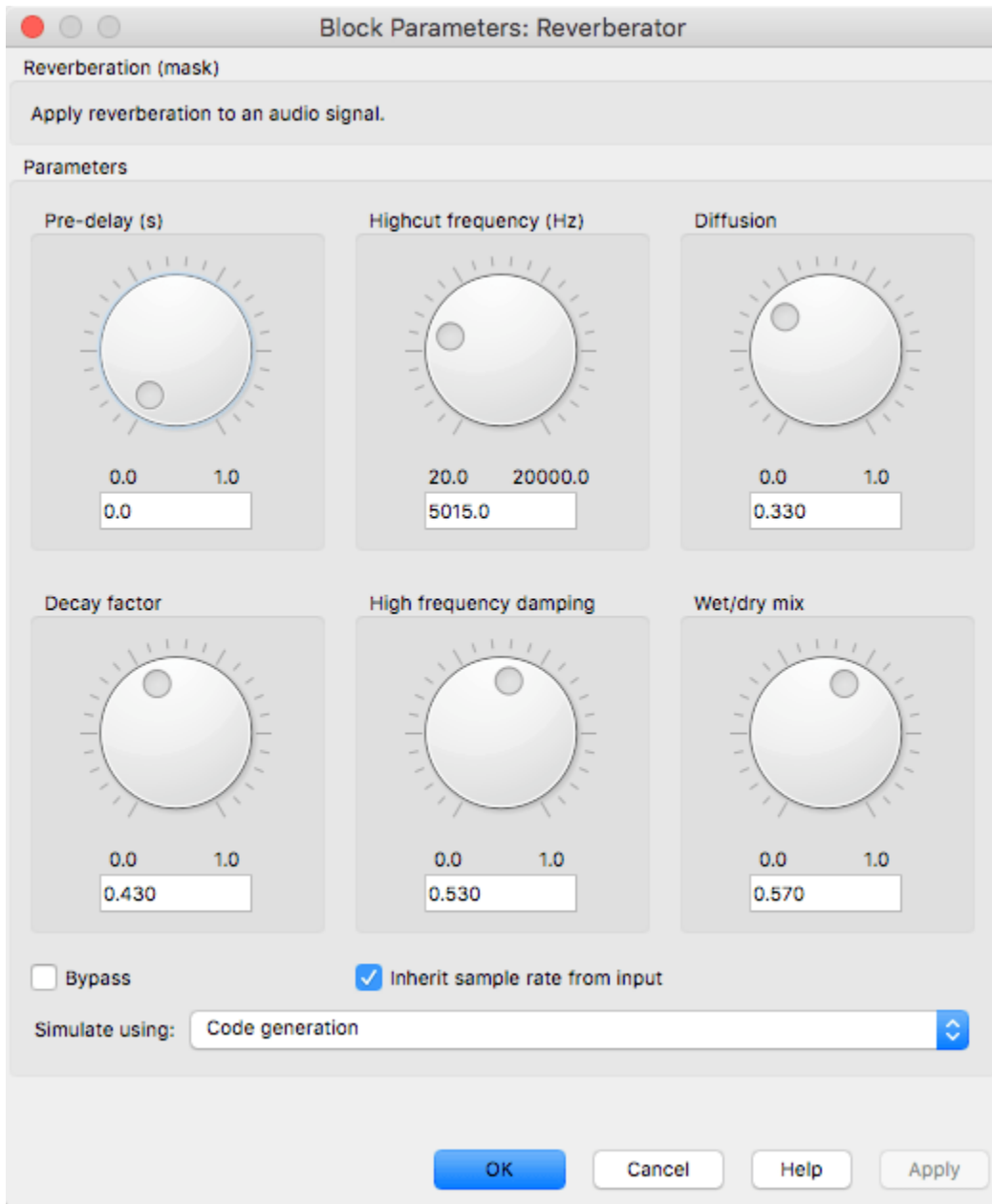
The reverberation effect has six tunable parameters that can be modified while the model is running:

- 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:

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



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 production 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.

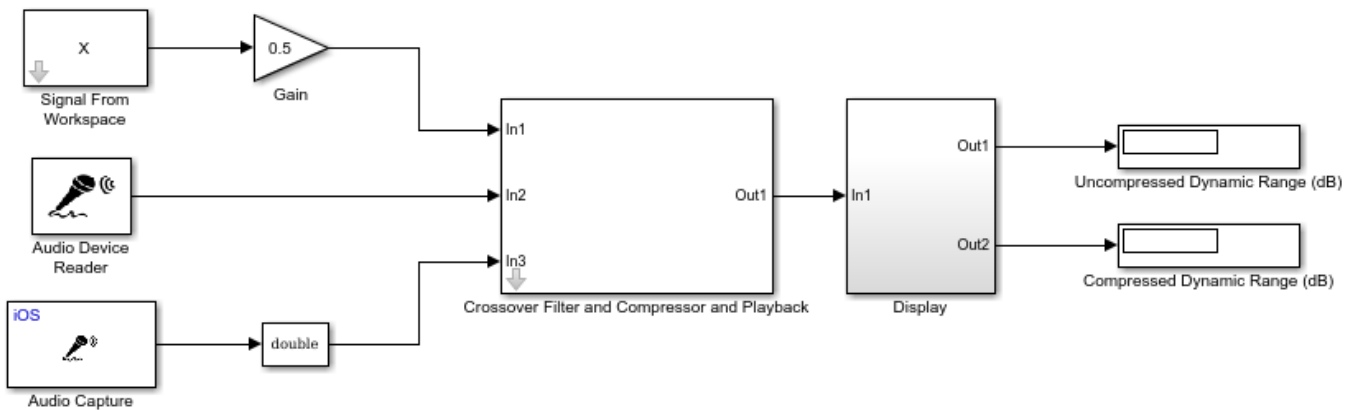
Required Hardware

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

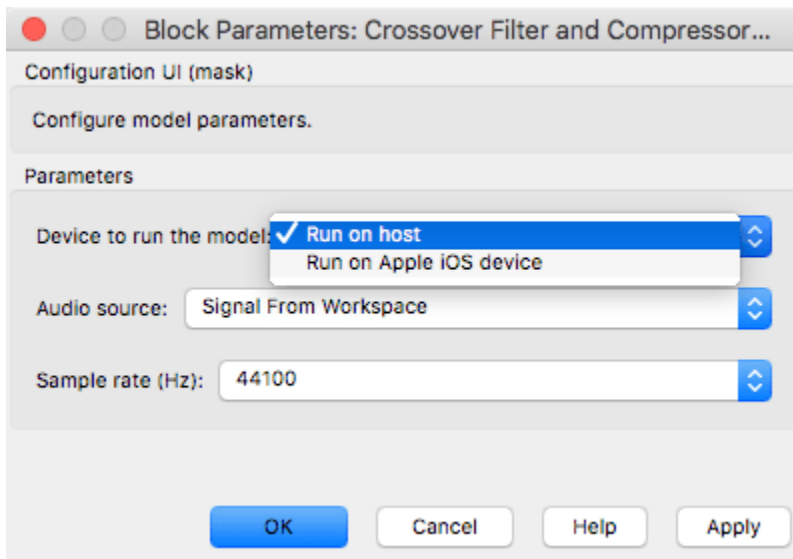
Model Setup

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:

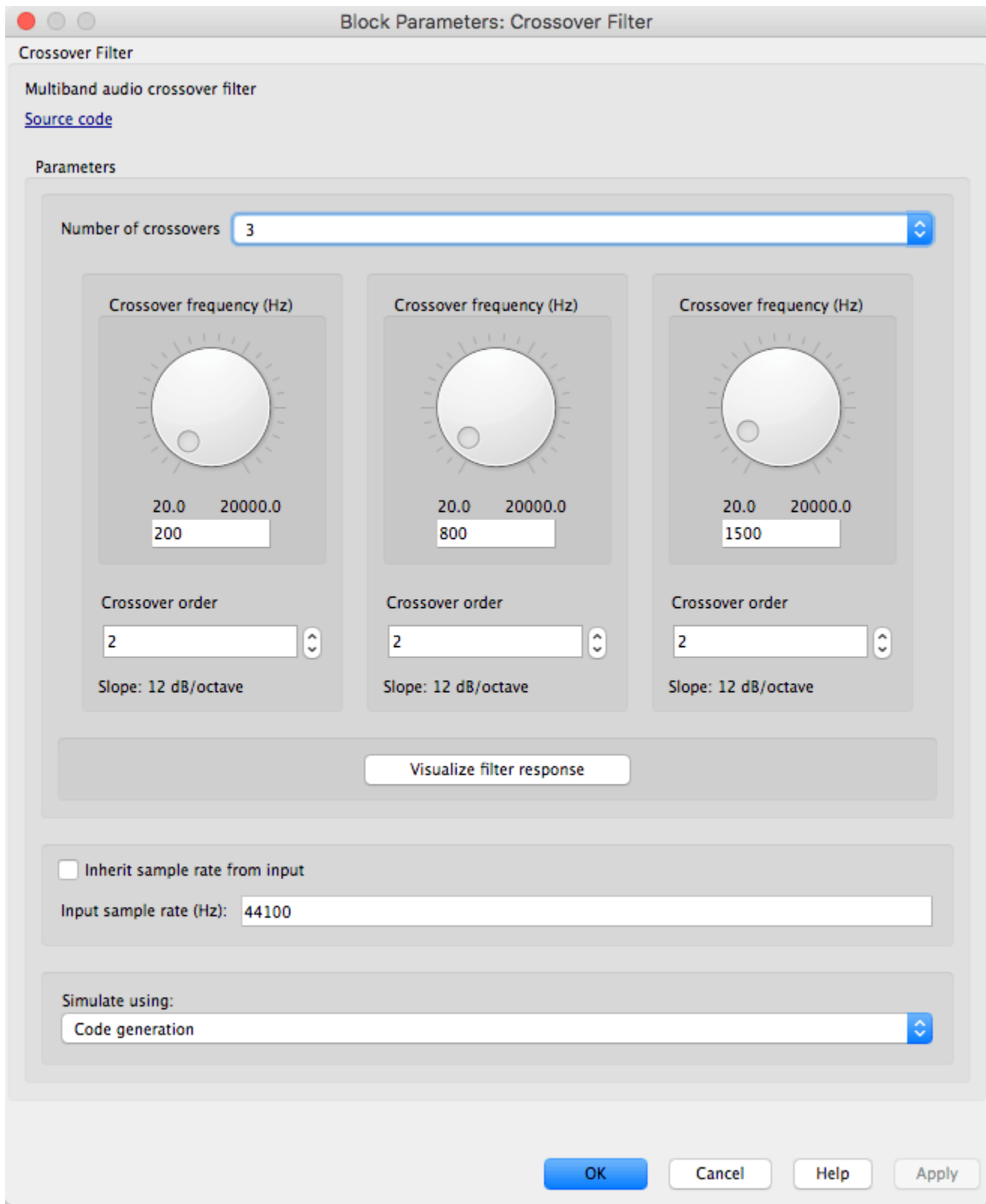


Crossover Filter

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:



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

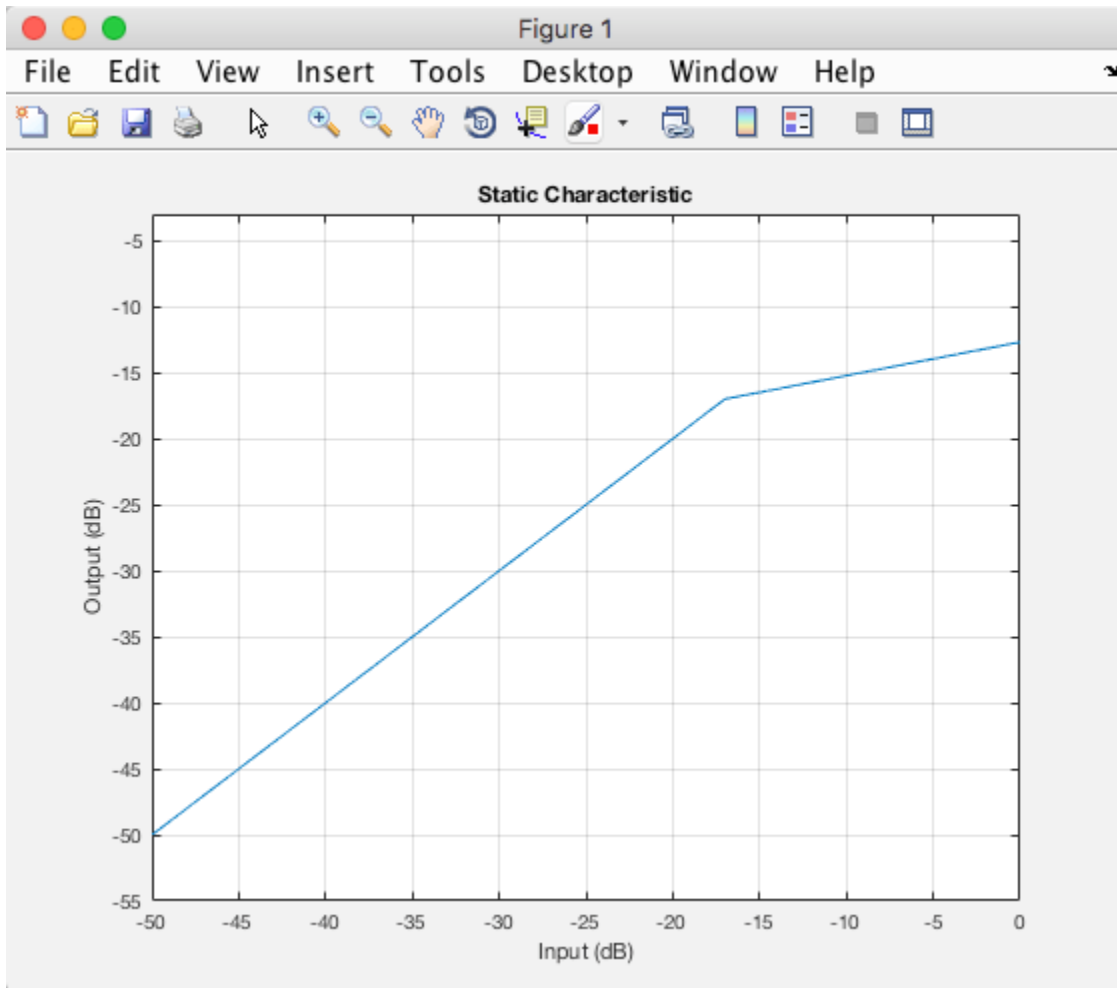
Parameters

Bypass

Band 1	Band 2	Band 3	Band 4
<p>Threshold (dB):</p> <p>-50.0 0.0 -5</p>	<p>Threshold (dB):</p> <p>-50.0 0.0 -10</p>	<p>Threshold (dB):</p> <p>-50.0 0.0 -20</p>	<p>Threshold (dB):</p> <p>-50.0 0.0 -30</p>
<p>Ratio:</p> <p>1.0 50.0 5</p>	<p>Ratio:</p> <p>1.0 50.0 5</p>	<p>Ratio:</p> <p>1.0 50.0 5</p>	<p>Ratio:</p> <p>1.0 50.0 5</p>
<p>Attack time (s):</p> <p>0.0 4.0 0.005</p>	<p>Attack time (s):</p> <p>0.0 4.0 0.005</p>	<p>Attack time (s):</p> <p>0.0 4.0 0.002</p>	<p>Attack time (s):</p> <p>0.0 4.0 0.002</p>
<p>Release time (s):</p> <p>0.0 4.0 0.1</p>	<p>Release time (s):</p> <p>0.0 4.0 0.1</p>	<p>Release time (s):</p> <p>0.0 4.0 0.050</p>	<p>Release time (s):</p> <p>0.0 4.0 0.050</p>
<p>View static characteristic</p>	<p>View static characteristic</p>	<p>View static characteristic</p>	<p>View static characteristic</p>

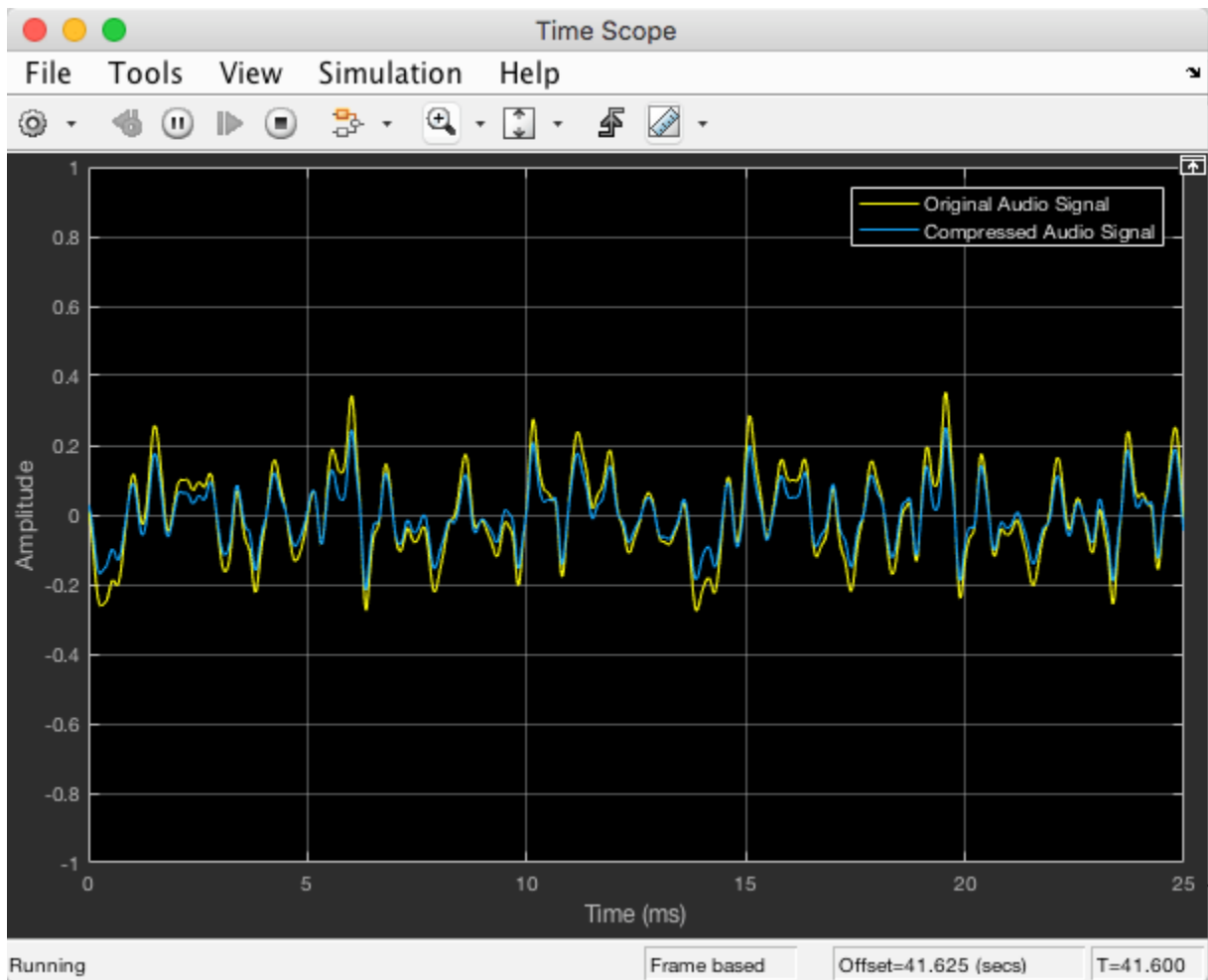
The UI enables you to tune the parameters of four single-band compressors individually, and view the static compression characteristics in real time. You can check the `Bypass` check box to compare the modified sound with the original sound.

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

audiomultibandcompressoriOS

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.

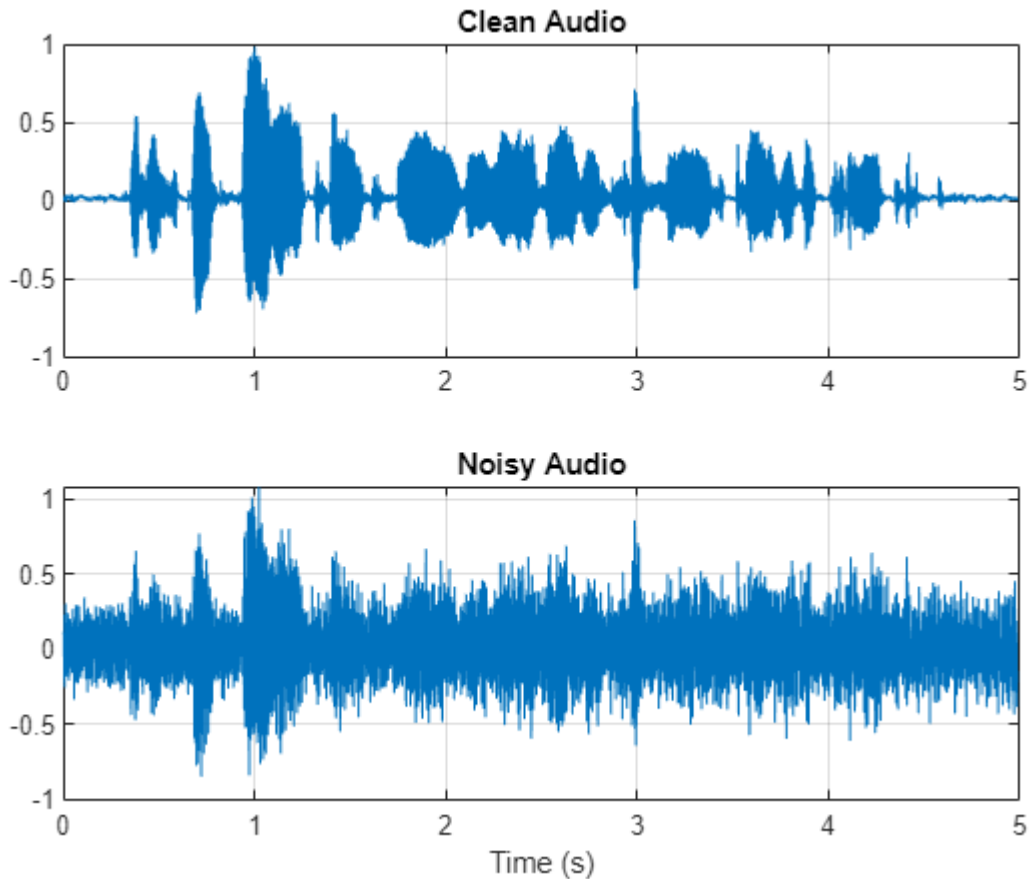
```
noise = audioread("WashingMachine-16-8-mono-1000secs.mp3");  
  
% Extract a noise segment from a random location in the noise file  
ind = randi(numel(noise) - numel(cleanAudio) + 1,1,1);  
noiseSegment = noise(ind:ind + numel(cleanAudio) - 1);  
  
speechPower = sum(cleanAudio.^2);  
noisePower = 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);  
  
figure(1)  
tiledlayout(2,1)  
  
nexttile  
plot(t,cleanAudio)  
title("Clean Audio")  
grid on  
  
nexttile  
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 1-331] 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.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","commonvoice.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
dataset = fullfile(dataFolder,"commonvoice");
```

Use `audioDatastore` to create a datastore for the training set. To speed up the runtime of the example at the cost of performance, set `speedupExample` to `true`.

```
adsTrain = audioDatastore(fullfile(dataset,"train"),IncludeSubfolders=true);
```

```
speedupExample =  ;
if speedupExample
    adsTrain = shuffle(adsTrain);
    adsTrain = subset(adsTrain,1:1000);
end
```

Use `read` to get the contents of the first file in the datastore.

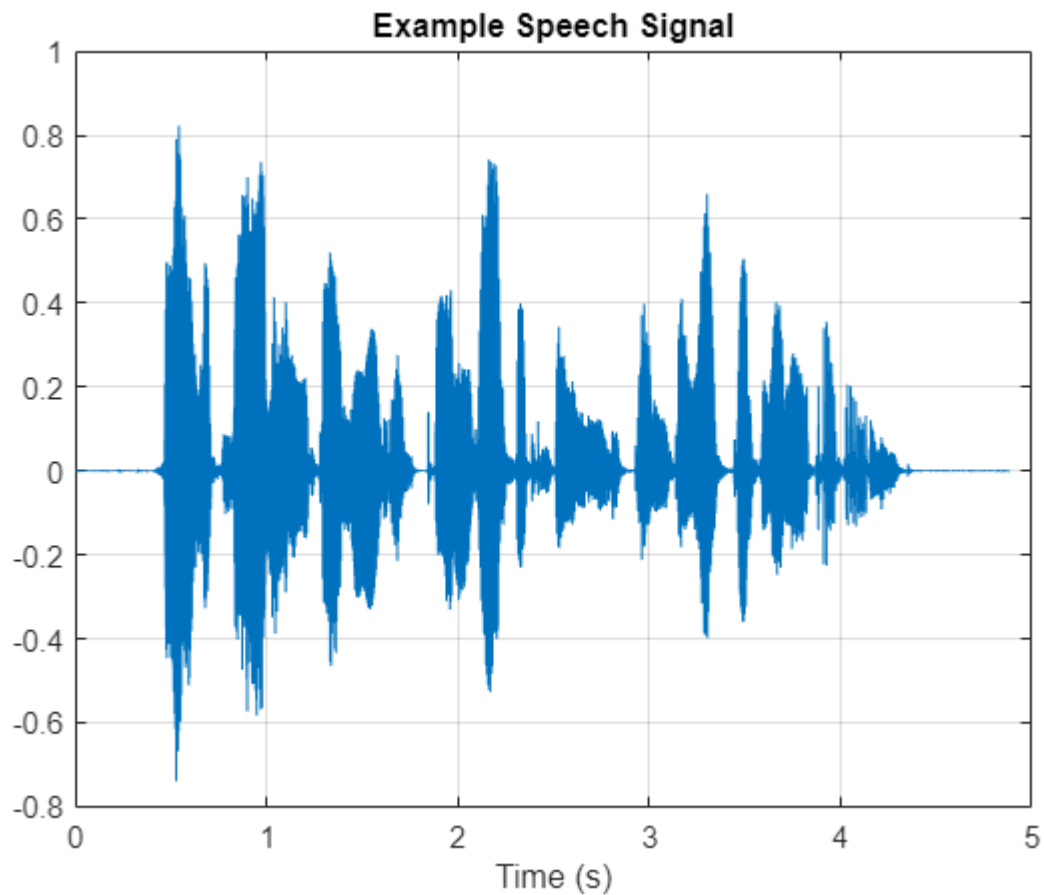
```
[audio,adsTrainInfo] = read(adsTrain);
```

Listen to the speech signal.

```
sound(audio,adsTrainInfo.SampleRate)
```

Plot the speech signal.

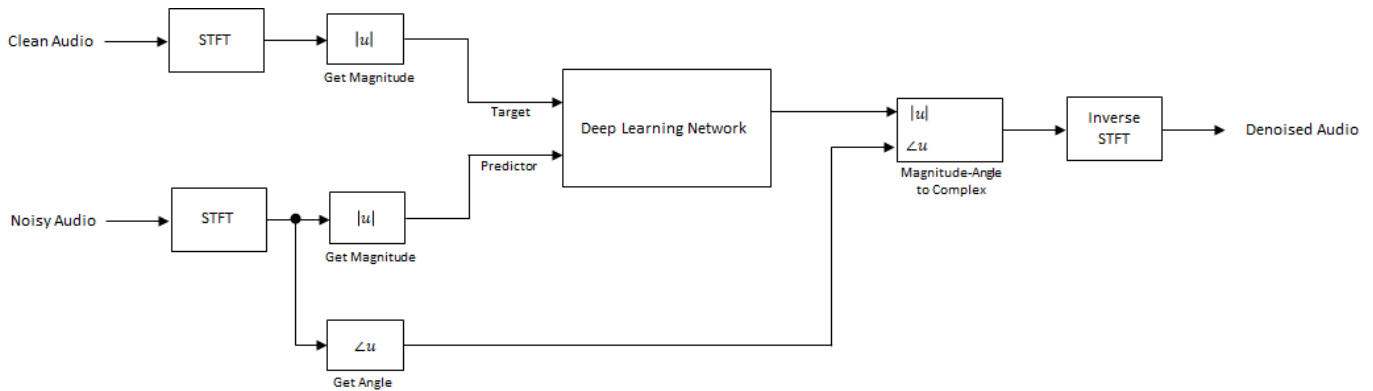
```
figure(2)
t = (1/adsTrainInfo.SampleRate) * (0:numel(audio)-1);
plot(t,audio)
title("Example Speech Signal")
xlabel("Time (s)")
grid on
```



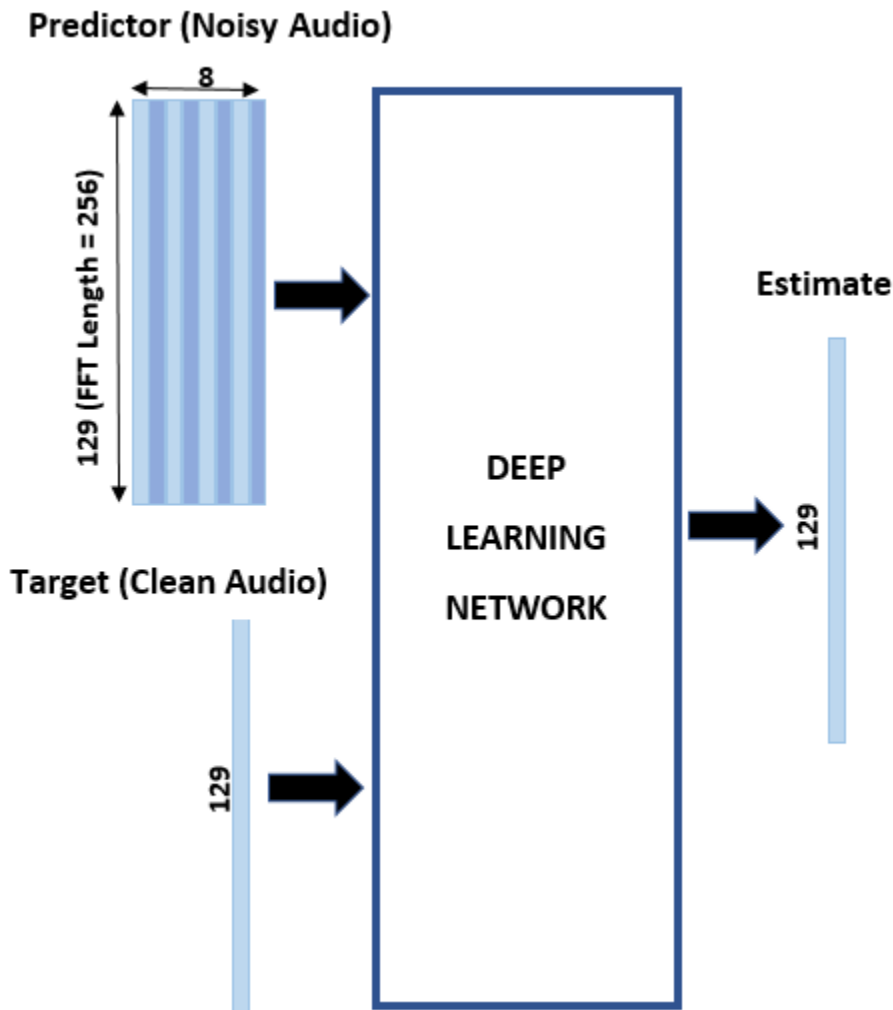
Deep Learning System Overview

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 1-331].



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 1-316 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,UniformOutput, false);
```

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 52 sec
Evaluation completed in 1 min 53 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));

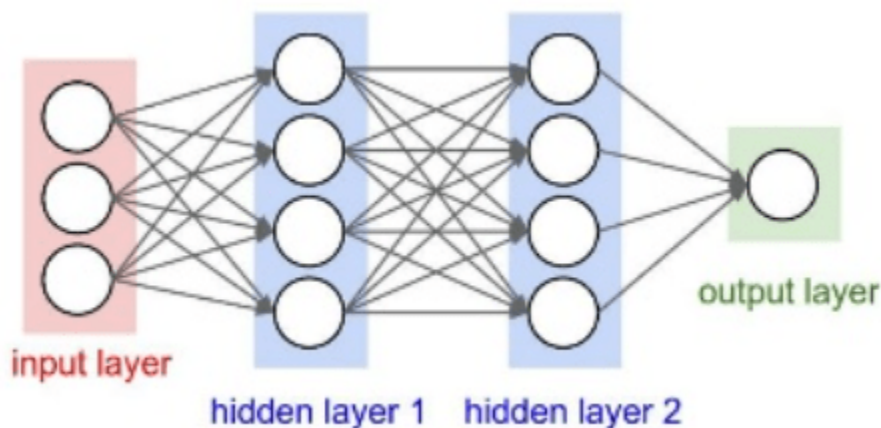
trainPredictors = predictors(:,:,,inds(1:L));
trainTargets = targets(:,:,,inds(1:L));

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 download and load a pre-trained network instead of training a network from scratch, set `downloadPretrainedSystem` to `true`.

```
downloadPretrainedSystem = ;
if downloadPretrainedSystem
    downloadFolder = matlab.internal.examples.downloadSupportFile("audio","SpeechDenoising.zip");
    dataFolder = tempdir;
    unzip(downloadFolder,dataFolder)
    netFolder = fullfile(dataFolder,"SpeechDenoising");

    s = load(fullfile(netFolder,"denoisenet.mat"));

    denoiseNetFullyConnected = s.denoiseNetFullyConnected;
    cleanMean = s.cleanMean;
```

```

        cleanStd = s.cleanStd;
        noisyMean = s.noisyMean;
        noisyStd = s.noisyStd;
else
    denoiseNetFullyConnected = trainNetwork(trainPredictors,trainTargets,layers,options);
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
disp("Number of weights = " + numWeights);

```

```
Number of weights = 2237440
```

Speech Denoising with Convolutional Layers

Consider a network that uses convolutional layers instead of fully connected layers [3 on page 1-331]. 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 1-331], 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 download and load a pre-trained network instead of training a network from scratch, set `downloadPretrainedSystem` to `true`.

```
downloadPretrainedSystem = ;
if downloadPretrainedSystem
    downloadFolder = matlab.internal.examples.downloadSupportFile("audio","SpeechDenoising.zip");
    dataFolder = tempdir;
    unzip(downloadFolder,dataFolder)
    netFolder = fullfile(dataFolder,"SpeechDenoising");

    s = load(fullfile(netFolder,"denoisenet.mat"));

    denoiseNetFullyConvolutional = s.denoiseNetFullyConvolutional;
    cleanMean = s.cleanMean;
    cleanStd = s.cleanStd;
    noisyMean = s.noisyMean;
    noisyStd = s.noisyStd;
else
    denoiseNetFullyConvolutional = trainNetwork(trainPredictors,permute(trainTargets,[3 1 2 4]),
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),"net.cnn.layer.Convolution2DLayer")
        numWeights = numWeights + numel(denoiseNetFullyConvolutional.Layers(index).Weights);
    end
end
disp("Number of weights in convolutional layers = " + numWeights);

Number of weights in convolutional layers = 31812
```

Test the Denoising Networks

Read in the test data set.

```
adsTest = audioDatastore(fullfile(dataset, "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);  
denoisedAudioFullyConvolutional = istft(STFTFullyConvolutional,Window=win,OverlapLength=overlap,fftLength=fftLength);
```

Plot the clean, noisy and denoised audio signals.

```
t = (1/fs)*(0:numel(denoisedAudioFullyConnected)-1);
```

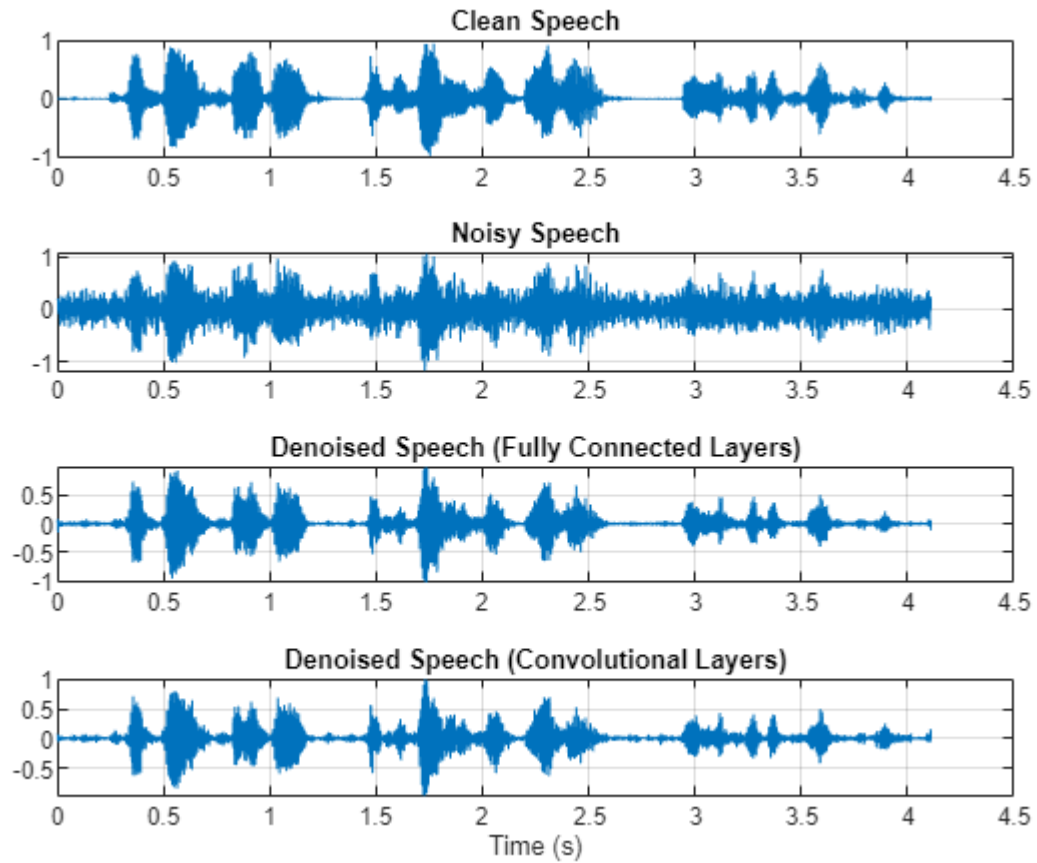
```
figure(3)  
tiledlayout(4,1)
```

```
nexttile  
plot(t,cleanAudio(1:numel(denoisedAudioFullyConnected)))  
title("Clean Speech")  
grid on
```

```
nexttile  
plot(t,noisyAudio(1:numel(denoisedAudioFullyConnected)))  
title("Noisy Speech")  
grid on
```

```
nexttile  
plot(t,denoisedAudioFullyConnected)  
title("Denoised Speech (Fully Connected Layers)")  
grid on
```

```
nexttile  
plot(t,denoisedAudioFullyConvolutional)  
title("Denoised Speech (Convolutional Layers)")  
grid on  
xlabel("Time (s)")
```



Plot the clean, noisy, and denoised spectrograms.

```
h = figure(4);
tiledlayout(4,1)

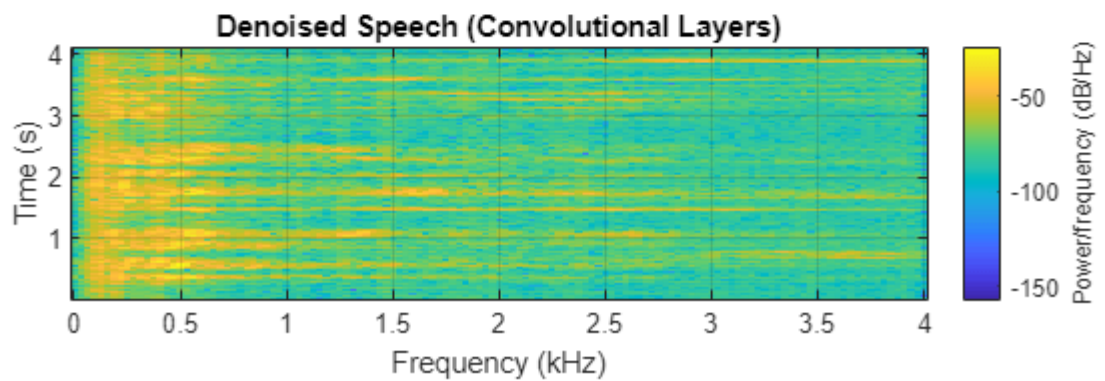
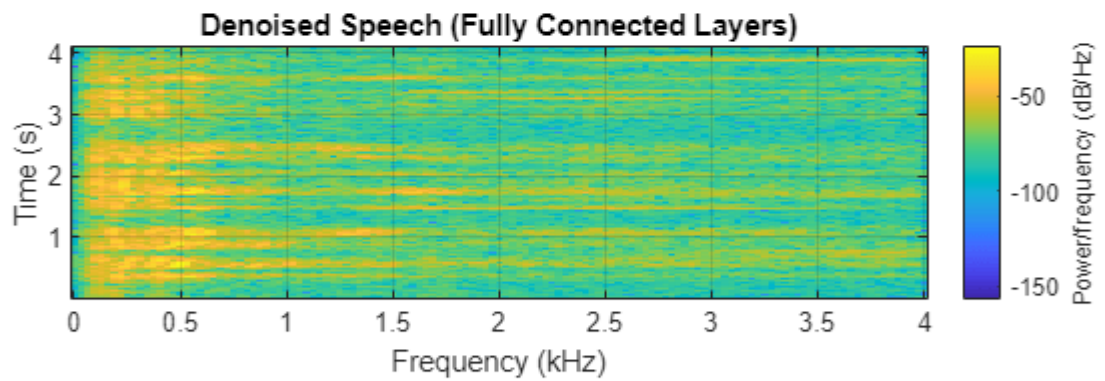
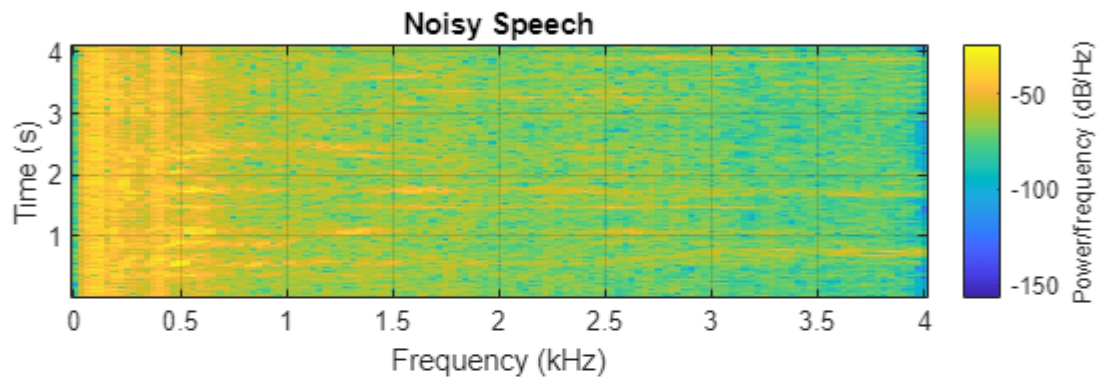
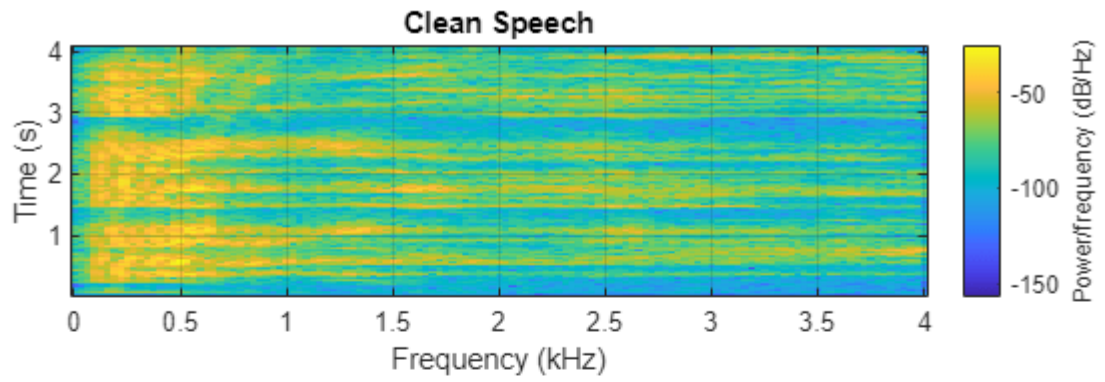
nexttile
spectrogram(cleanAudio,win,overlap,fftLength,fs);
title("Clean Speech")
grid on

nexttile
spectrogram(noisyAudio,win,overlap,fftLength,fs);
title("Noisy Speech")
grid on

nexttile
spectrogram(denoisedAudioFullyConnected,win,overlap,fftLength,fs);
title("Denoised Speech (Fully Connected Layers)")
grid on

nexttile
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.

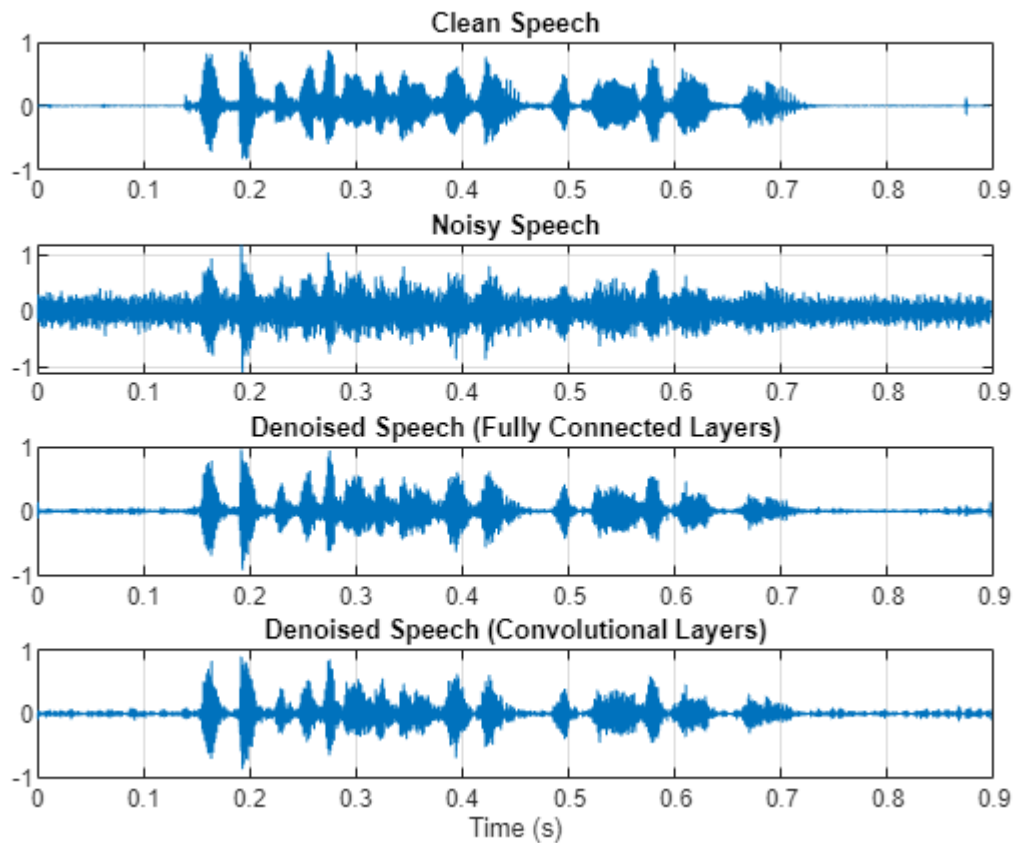
```
sound(denoisedAudioFullyConvolutional, fs)
```

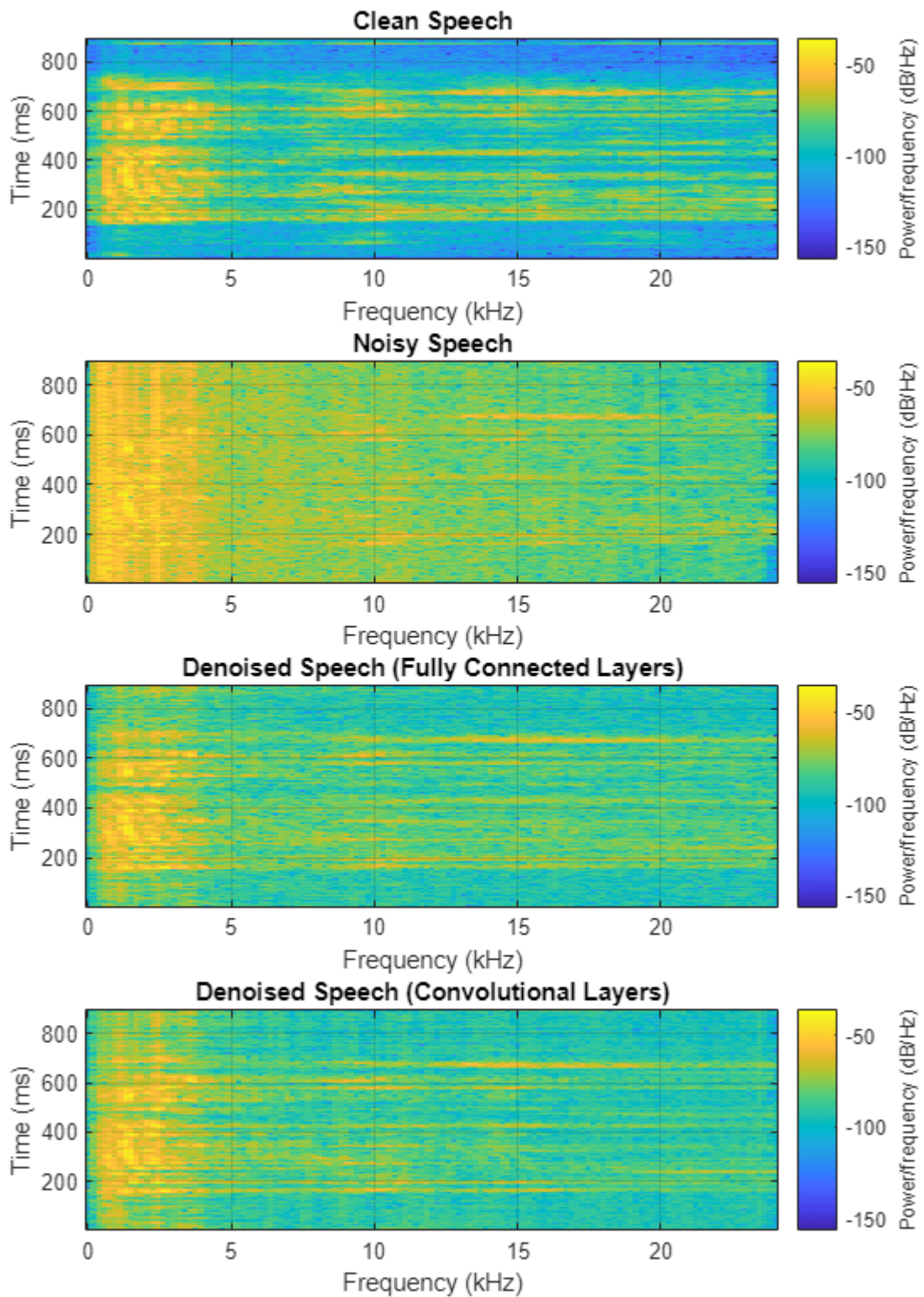
Listen to clean speech.

```
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
```



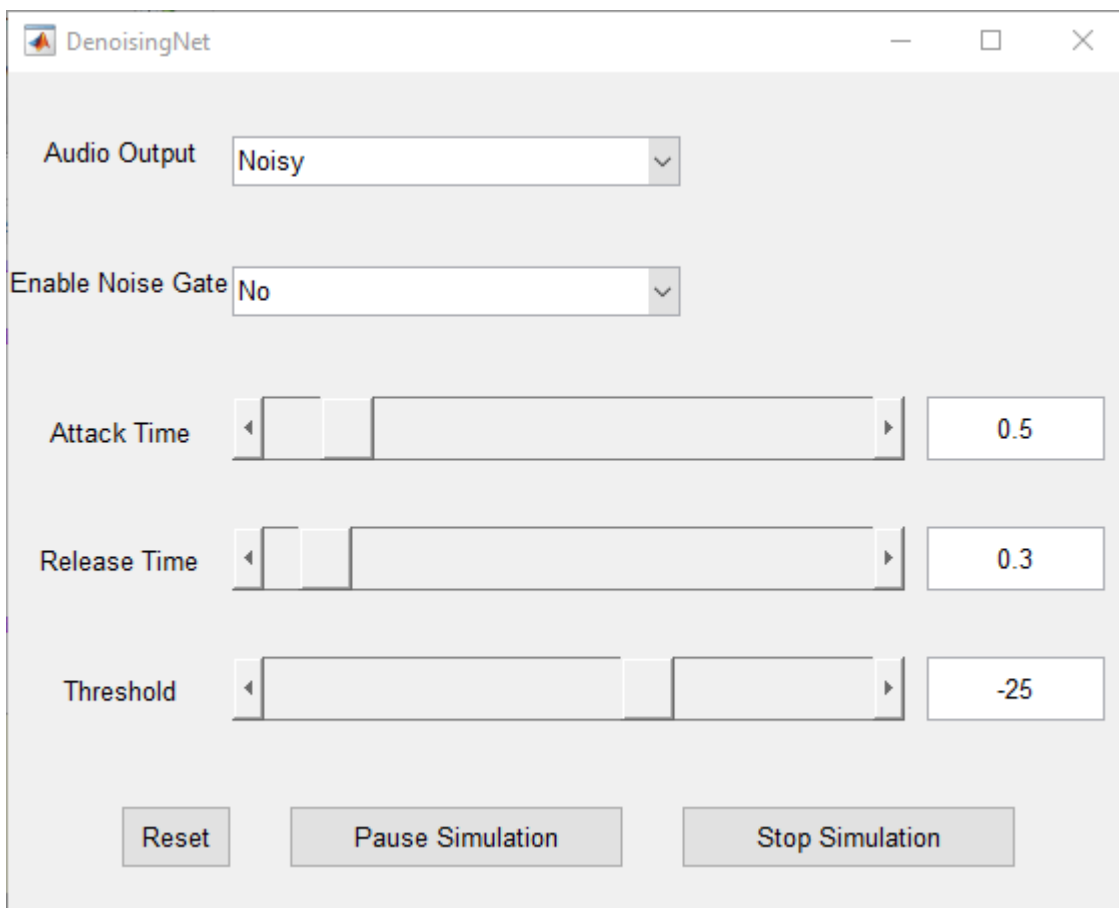


Real-Time Application

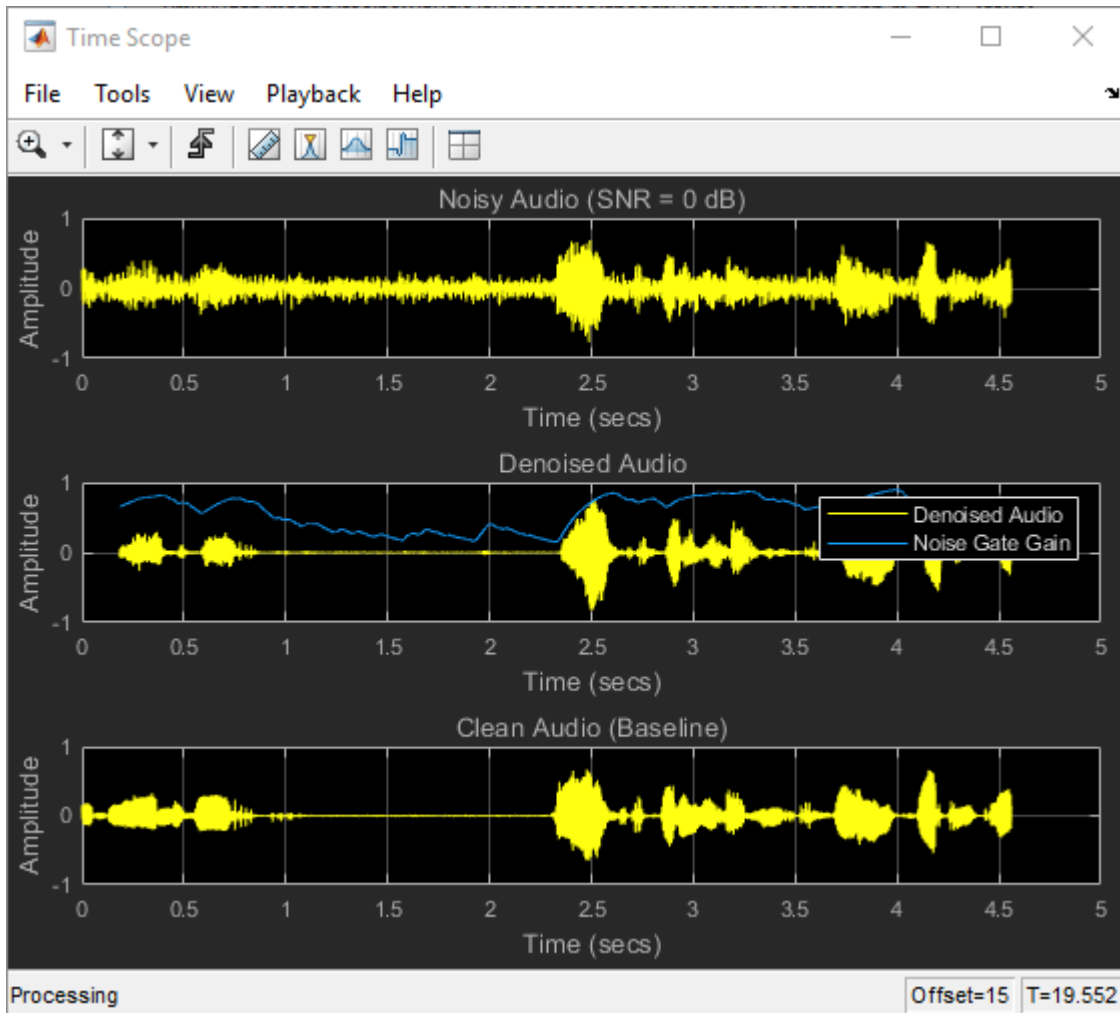
The procedure in the previous section passes the entire spectrum of the noisy signal to `predict`. This is not suitable for real-time applications where low latency is a requirement.

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.

See Also

Related Examples

- "3-D Speech Enhancement Using Trained Filter and Sum Network" on page 1-973

Train Speech Command Recognition Model 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] on page 1-343 to train a convolutional neural network to recognize a set of commands.

To use a pretrained speech command recognition system, see “Speech Command Recognition Using Deep Learning” on page 1-929.



To run the example quickly, set `speedupExample` to `true`. To run the full example as published, set `speedupExample` to `false`.

```
speedupExample =  ;
```

Set the random seed for reproducibility.

```
rng default
```

Load Data

This example uses the Google Speech Commands Dataset [1] on page 1-343. Download and unzip the data set.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "google_speech.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
dataset = fullfile(dataFolder, "google_speech");
```

Augment Data

The network should be able to not only recognize different spoken words but also to detect if the audio input is silence or background noise.

The supporting function, `augmentDataset` on page 1-342, uses the long audio files in the background folder of the Google Speech Commands Dataset to create one-second segments of background noise. The function creates an equal number of background segments from each background noise file and then splits the segments between the train and validation folders.

```
augmentDataset(dataset)
```

```

Progress = 17 (%)
Progress = 33 (%)
Progress = 50 (%)
Progress = 67 (%)
Progress = 83 (%)
Progress = 100 (%)

```

Create Training Datastore

Create an `audioDatastore` that points to the training data set.

```

ads = audioDatastore(fullfile(dataset,"train"), ...
    IncludeSubfolders=true, ...
    FileExtensions=".wav", ...
    LabelSource="foldernames");

```

Specify the words that you want your model to recognize as commands. Label all files that are not commands or background noise 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, the background noise, 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"]);
background = categorical("background");

isCommand = ismember(ads.Labels,commands);
isBackground = ismember(ads.Labels,background);
isUnknown = ~(isCommand|isBackground);

includeFraction = 0.2; % Fraction of unknowns to include.
idx = find(isUnknown);
idx = idx(randperm(numel(idx),round((1-includeFraction)*sum(isUnknown))));
isUnknown(idx) = false;

ads.Labels(isUnknown) = categorical("unknown");

adsTrain = subset(ads,isCommand|isUnknown|isBackground);
adsTrain.Labels = removecats(adsTrain.Labels);

```

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(dataset,"validation"), ...
    IncludeSubfolders=true, ...
    FileExtensions=".wav", ...
    LabelSource="foldernames");

isCommand = ismember(ads.Labels,commands);
isBackground = ismember(ads.Labels,background);
isUnknown = ~(isCommand|isBackground);

```



```

includeFraction = 0.2; % Fraction of unknowns to include.
idx = find(isUnknown);
idx = idx(randperm(numel(idx), round((1-includeFraction)*sum(isUnknown))));
isUnknown(idx) = false;

ads.Labels(isUnknown) = categorical("unknown");

adsValidation = subset(ads, isCommand|isUnknown|isBackground);
adsValidation.Labels = removecats(adsValidation.Labels);

```

Visualize the training and validation label distributions.

```

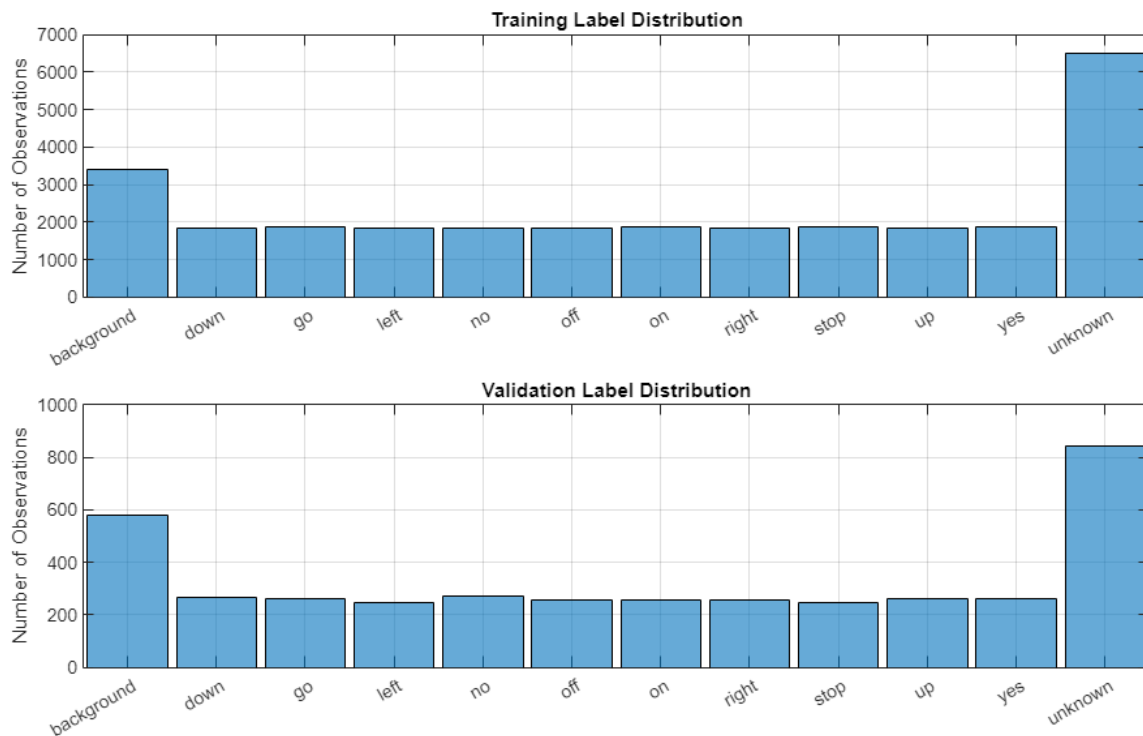
figure(Units="normalized", Position=[0.2,0.2,0.5,0.5])

tiledlayout(2,1)

nexttile
histogram(adsTrain.Labels)
title("Training Label Distribution")
ylabel("Number of Observations")
grid on

nexttile
histogram(adsValidation.Labels)
title("Validation Label Distribution")
ylabel("Number of Observations")
grid on

```



Speed up the example by reducing the data set, if requested.


```

if speedupExample
    numUniqueLabels = numel(unique(adsTrain.Labels)); %#ok<UNRCH>
    % 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

```

Prepare Data for Training

To prepare the data for efficient training of a convolutional neural network, convert the speech waveforms to auditory-based spectrograms.

To speed up processing, you can distribute the feature extraction across multiple workers. Start a parallel pool if you have access to Parallel Computing Toolbox™.

```

if canUseParallelPool && ~speedupExample
    useParallel = true;
    gcp;
else
    useParallel = false;
end

```

```

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

```

Extract Features

Define the parameters to extract auditory spectrograms from the audio input. `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.

```
fs = 16e3; % Known sample rate of the data set.
```

```
segmentDuration = 1;
frameDuration = 0.025;
hopDuration = 0.010;
```

```
FFTLength = 512;
numBands = 50;
```

```
segmentSamples = round(segmentDuration*fs);
frameSamples = round(frameDuration*fs);
hopSamples = round(hopDuration*fs);
overlapSamples = frameSamples - hopSamples;
```

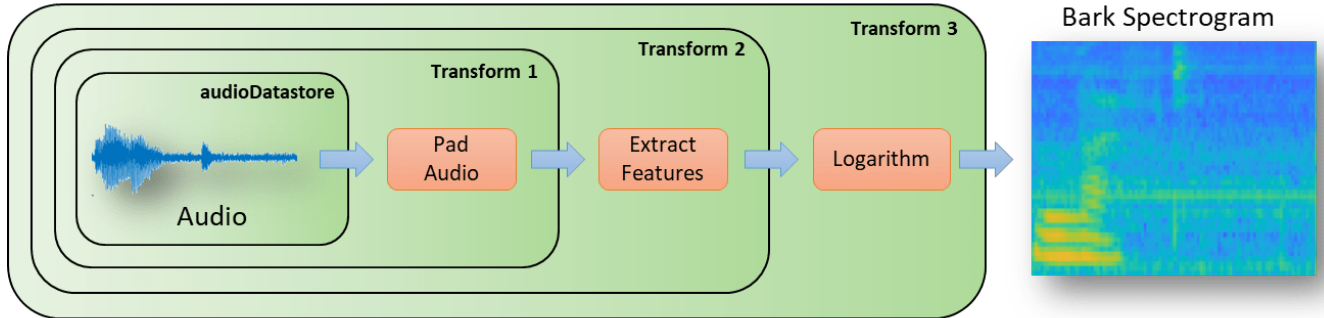
Create an `audioFeatureExtractor` object to perform the feature extraction.

```

afe = audioFeatureExtractor( ...
    SampleRate=fs, ...
    FFTLength=FFTLength, ...
    Window=hann(frameSamples,"periodic"), ...
    OverlapLength=overlapSamples, ...
    barkSpectrum=true);
setExtractorParameters(afe,"barkSpectrum",NumBands=numBands,WindowNormalization=false);

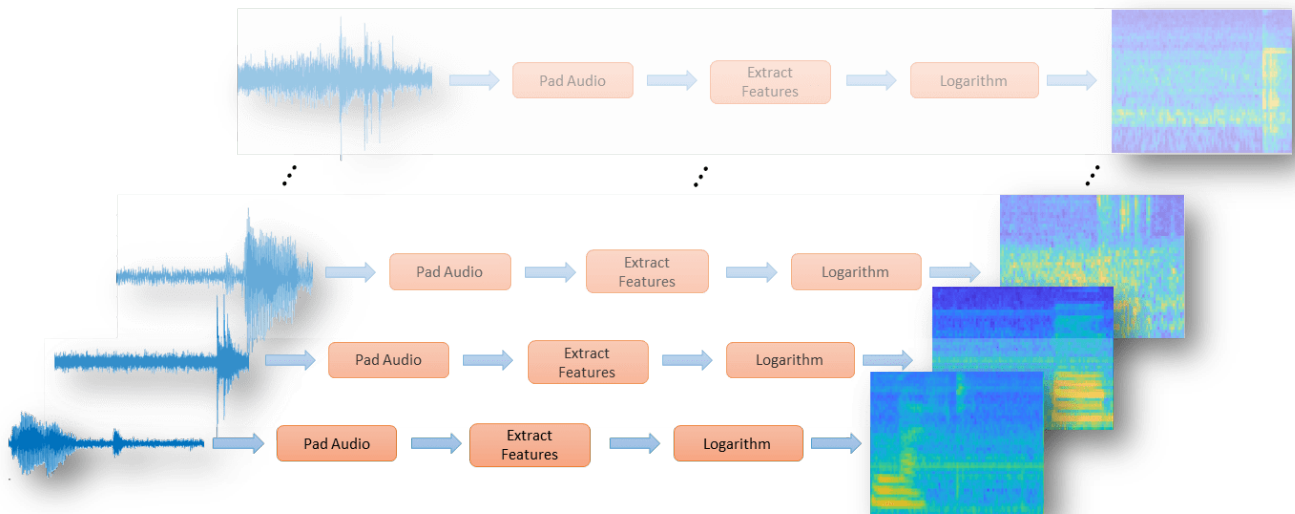
```

Define a series of transform on the `audioDatastore` to pad the audio to a consistent length, extract the features, and then apply a logarithm.



```
transform1 = transform(adsTrain,@(x)[zeros(floor((segmentSamples-size(x,1))/2),1);x;zeros(ceil((segmentSamples-size(x,1))/2),1)]);
transform2 = transform(transform1,@(x)extract(afe,x));
transform3 = transform(transform2,@(x){log10(x+1e-6)});
```

Use the `readall` function to read all data from the datastore. As each file is read, it is passed through the transforms before the data is returned.



```
XTrain = readall(transform3,UseParallel=useParallel);
```

The output is a `numFiles`-by-1 cell array. Each element of the cell array corresponds to the auditory spectrogram extracted from a file.

```
numFiles = numel(XTrain)
```

```
numFiles = 28463
```

```
[numHops,numBands,numChannels] = size(XTrain{1})
```

```
numHops = 98
```

```
numBands = 50
```

```
numChannels = 1
```

Convert the cell array to a 4-dimensional array with auditory spectrograms along the fourth dimension.

```
XTrain = cat(4,XTrain{:});

[numHops,numBands,numChannels,numFiles] = size(XTrain)

numHops = 98

numBands = 50

numChannels = 1

numFiles = 28463
```

Perform the feature extraction steps described above on the validation set.

```
transform1 = transform(adsValidation,@(x)[zeros(floor((segmentSamples-size(x,1))/2),1);x;zeros(c
transform2 = transform(transform1,@(x)extract(afe,x));
transform3 = transform(transform2,@(x){log10(x+1e-6)});
XValidation = readall(transform3,UseParallel=useParallel);
XValidation = cat(4,XValidation{:});
```

For convenience, isolate the train and validation target labels.

```
TTrain = adsTrain.Labels;
TValidation = adsValidation.Labels;
```

Visualize Data

Plot the waveforms and auditory spectrograms of a few training samples. Play the corresponding audio clips.

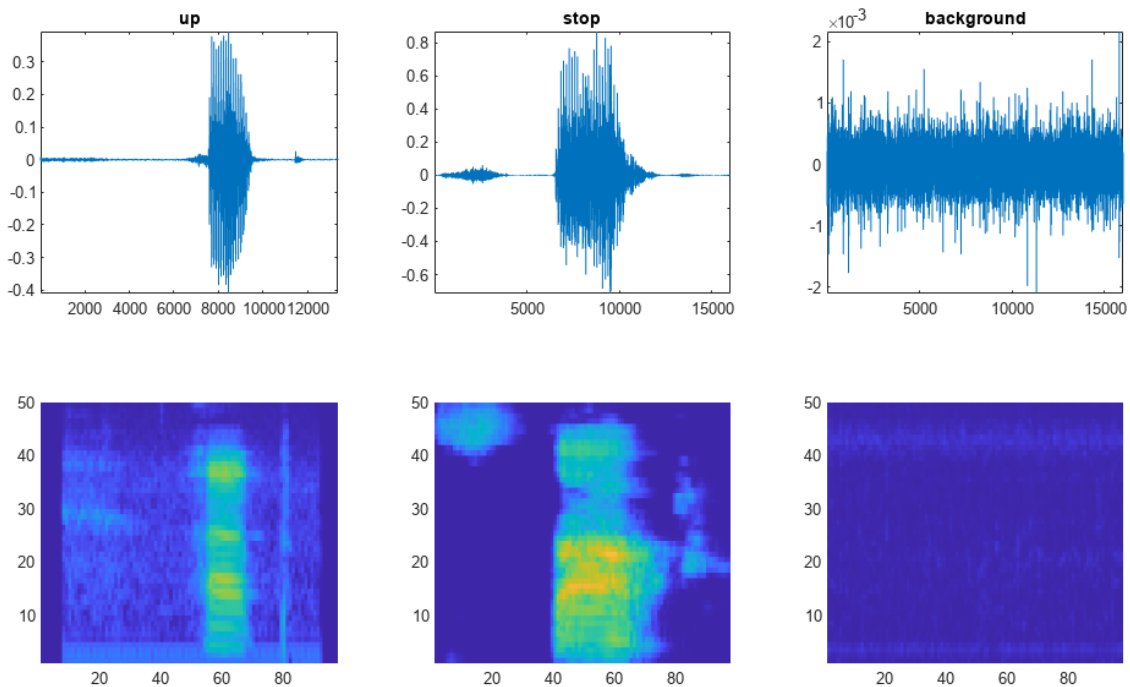
```
specMin = min(XTrain,[],"all");
specMax = max(XTrain,[],"all");
idx = randperm(numel(adsTrain.Files),3);
figure(Units="normalized",Position=[0.2,0.2,0.6,0.6]);

tiledlayout(2,3)
for ii = 1:3
    [x,fs] = audioread(adsTrain.Files{idx(ii)});

    nexttile(ii)
    plot(x)
    axis tight
    title(string(adsTrain.Labels(idx(ii))))

    nexttile(ii+3)
    spect = XTrain(:,:,1,idx(ii))';
    pcolor(spect)
    clim([specMin specMax])
    shading flat

    sound(x,fs)
    pause(2)
end
```

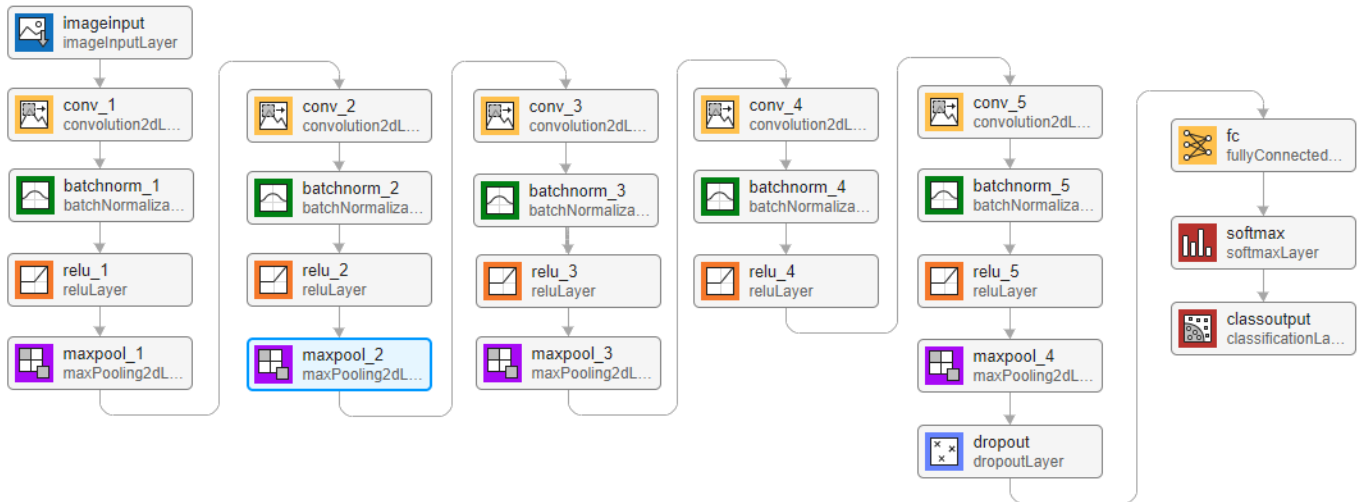


Define Network Architecture

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`.

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.



```

classes = categories(TTrain);
classWeights = 1./countcats(TTrain);
classWeights = classWeights'/mean(classWeights);
numClasses = numel(classes);

timePoolSize = ceil(numHops/8);

dropoutProb = 0.2;
numF = 12;
layers = [
    imageInputLayer([numHops,afe.FeatureVectorLength])

    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
classificationLayer(Classes=classes,ClassWeights=classWeights)];
```

Specify Training Options

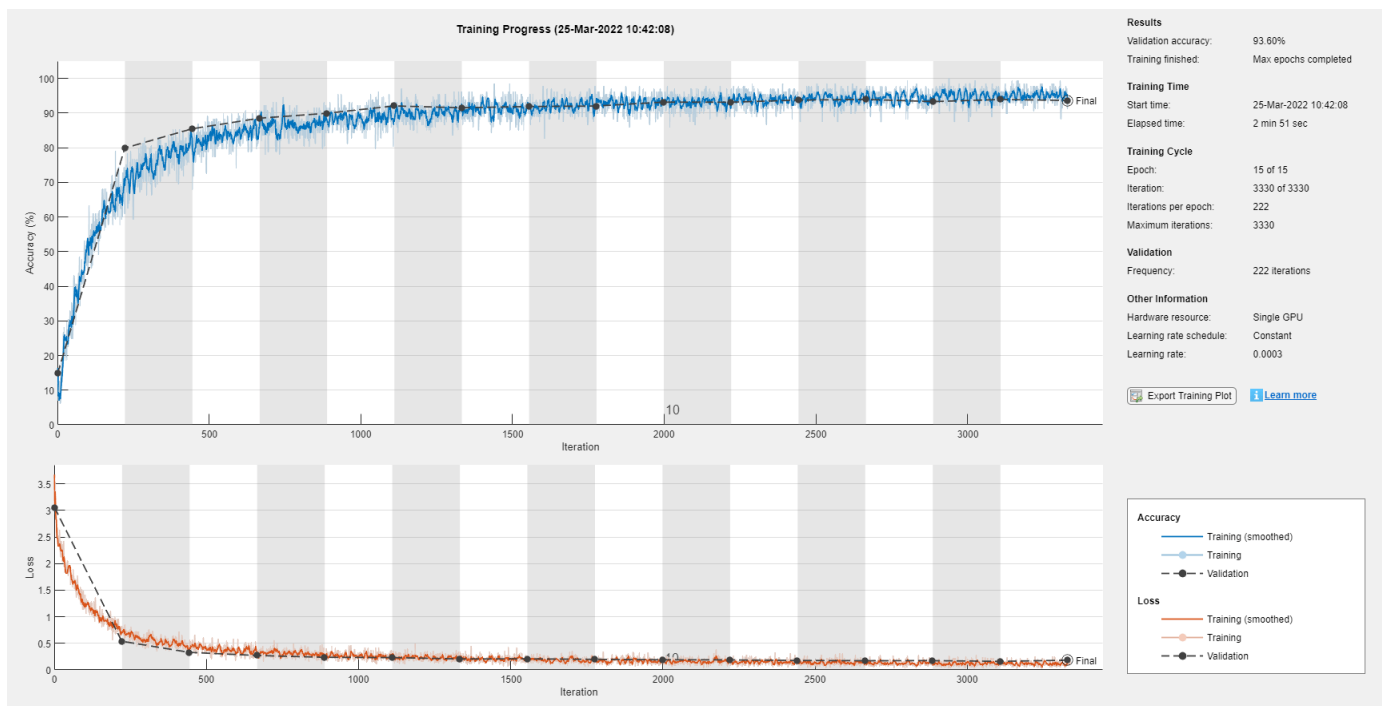
To define parameters for training, use `trainingOptions` (Deep Learning Toolbox). Use the Adam optimizer with a mini-batch size of 128.

```
miniBatchSize = 128;
validationFrequency = floor(numel(TTrain)/miniBatchSize);
options = trainingOptions("adam", ...
    InitialLearnRate=3e-4, ...
    MaxEpochs=15, ...
    MiniBatchSize=miniBatchSize, ...
    Shuffle="every-epoch", ...
    Plots="training-progress", ...
    Verbose=false, ...
    ValidationData={XValidation,TValidation}, ...
    ValidationFrequency=validationFrequency);
```

Train Network

To train the network, use `trainNetwork` (Deep Learning Toolbox). If you do not have a GPU, then training the network can take time.

```
trainedNet = trainNetwork(XTrain,TTrain,layers,options);
```



Evaluate Trained Network

To calculate the final accuracy of the network on the training and validation sets, use `classify` (Deep Learning Toolbox). 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.

```

YValidation = classify(trainedNet,XValidation);
validationError = mean(YValidation ~= TValidation);
YTrain = classify(trainedNet,XTrain);
trainError = mean(YTrain ~= TTrain);

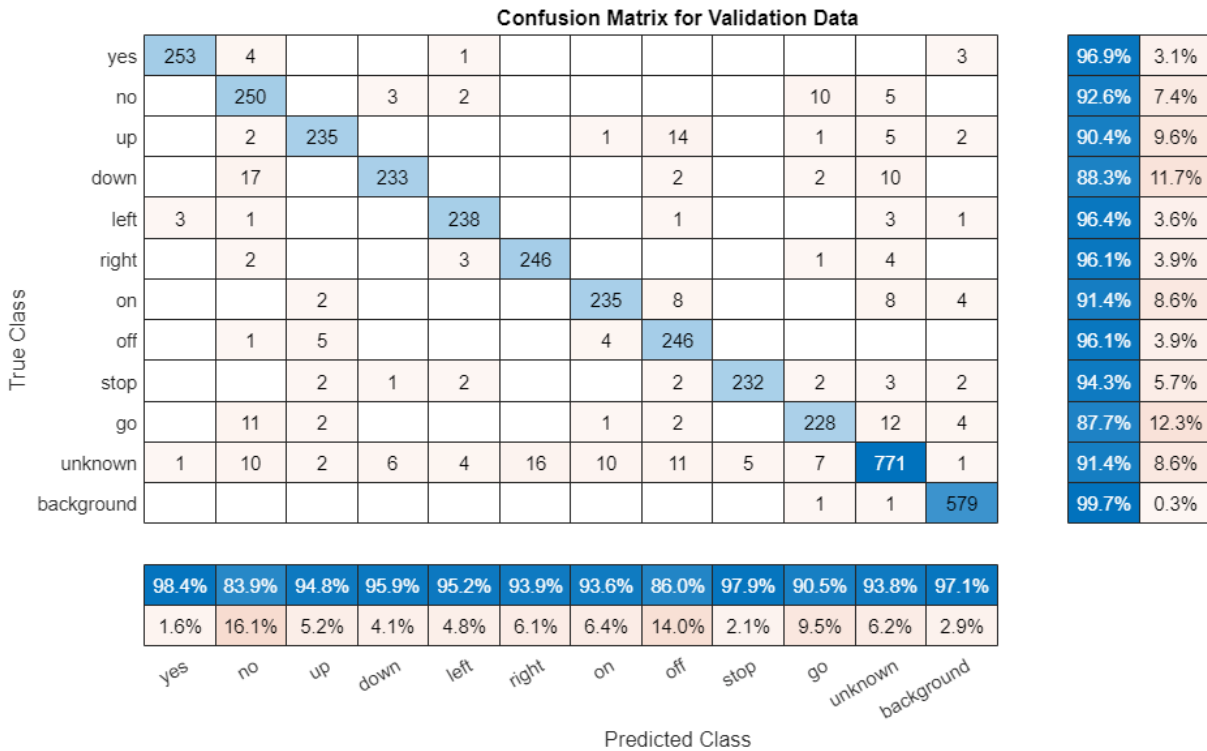
disp(["Training error: " + trainError*100 + "%";"Validation error: " + validationError*100 + "%"]);

    "Training error: 2.7263%"
    "Validation error: 6.3968%"
    
```

To plot the confusion matrix for the validation set, use confusionchart (Deep Learning Toolbox). Display the precision and recall for each class by using column and row summaries.

```

figure(Units="normalized",Position=[0.2,0.2,0.5,0.5]);
cm = confusionchart(TValidation,YValidation, ...
    Title="Confusion Matrix for Validation Data", ...
    ColumnSummary="column-normalized",RowSummary="row-normalized");
sortClasses(cm,[commands,"unknown","background"])
    
```



When working on applications with constrained hardware resources, such as mobile applications, it is important to 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.

```
for ii = 1:100
    x = randn([numHops,numBands]);
    predictionTimer = tic;
    [y,probs] = classify(trainedNet,x,ExecutionEnvironment="cpu");
    time(ii) = toc(predictionTimer);
end

disp(["Network size: " + whos("trainedNet").bytes/1024 + " kB"; ...
     "Single-image prediction time on CPU: " + mean(time(11:end))*1000 + " ms"])

     "Network size: 292.2842 kB"
     "Single-image prediction time on CPU: 3.7237 ms"
```

Supporting Functions

Augment Dataset With Background Noise

```
function augmentDataset(datasetloc)
adsBkg = audioDatastore(fullfile(datasetloc,"background"));
fs = 16e3; % Known sample rate of the data set
segmentDuration = 1;
segmentSamples = round(segmentDuration*fs);

volumeRange = log10([1e-4,1]);

numBkgSegments = 4000;
numBkgFiles = numel(adsBkg.Files);
numSegmentsPerFile = floor(numBkgSegments/numBkgFiles);

fpTrain = fullfile(datasetloc,"train","background");
fpValidation = fullfile(datasetloc,"validation","background");

if ~datasetExists(fpTrain)

    % Create directories
    mkdir(fpTrain)
    mkdir(fpValidation)

    for backgroundFileIndex = 1:numel(adsBkg.Files)
        [bkgFile,fileInfo] = read(adsBkg);
        [~,fn] = fileparts(fileInfo.FileName);

        % Determine starting index of each segment
        segmentStart = randi(size(bkgFile,1)-segmentSamples,numSegmentsPerFile,1);

        % Determine gain of each clip
        gain = 10.^((volumeRange(2)-volumeRange(1))*rand(numSegmentsPerFile,1) + volumeRange(1))

        for segmentIdx = 1:numSegmentsPerFile

            % Isolate the randomly chosen segment of data.
            bkgSegment = bkgFile(segmentStart(segmentIdx):segmentStart(segmentIdx)+segmentSamples);

            % Scale the segment by the specified gain.
            bkgSegment = bkgSegment*gain(segmentIdx);

            % Clip the audio between -1 and 1.
```



```
bkgSegment = max(min(bkgSegment,1),-1);

% Create a file name.
afn = fn + "_segment" + segmentIdx + ".wav";

% Randomly assign background segment to either the train or
% validation set.
if rand > 0.85 % Assign 15% to validation
    dirToWriteTo = fpValidation;
else % Assign 85% to train set.
    dirToWriteTo = fpTrain;
end

% Write the audio to the file location.
ffn = fullfile(dirToWriteTo,afn);
audiowrite(ffn,bkgSegment,fs)

end

% Print progress
fprintf('Progress = %d (%%)\n',round(100*progress(adsBkg)))

end
end
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>.

See Also

Related Examples

- "Speech Command Recognition Using Deep Learning" on page 1-929

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.

Supported Conventions

- 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.

Pseudoinverse Decoding Method

There are many decoding options. This example uses pseudoinverse decoding, also known as mode matching. This decoding method favors regular-shaped device layouts. Other decoding methods may favor irregular-shaped device layouts.

Device Type

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

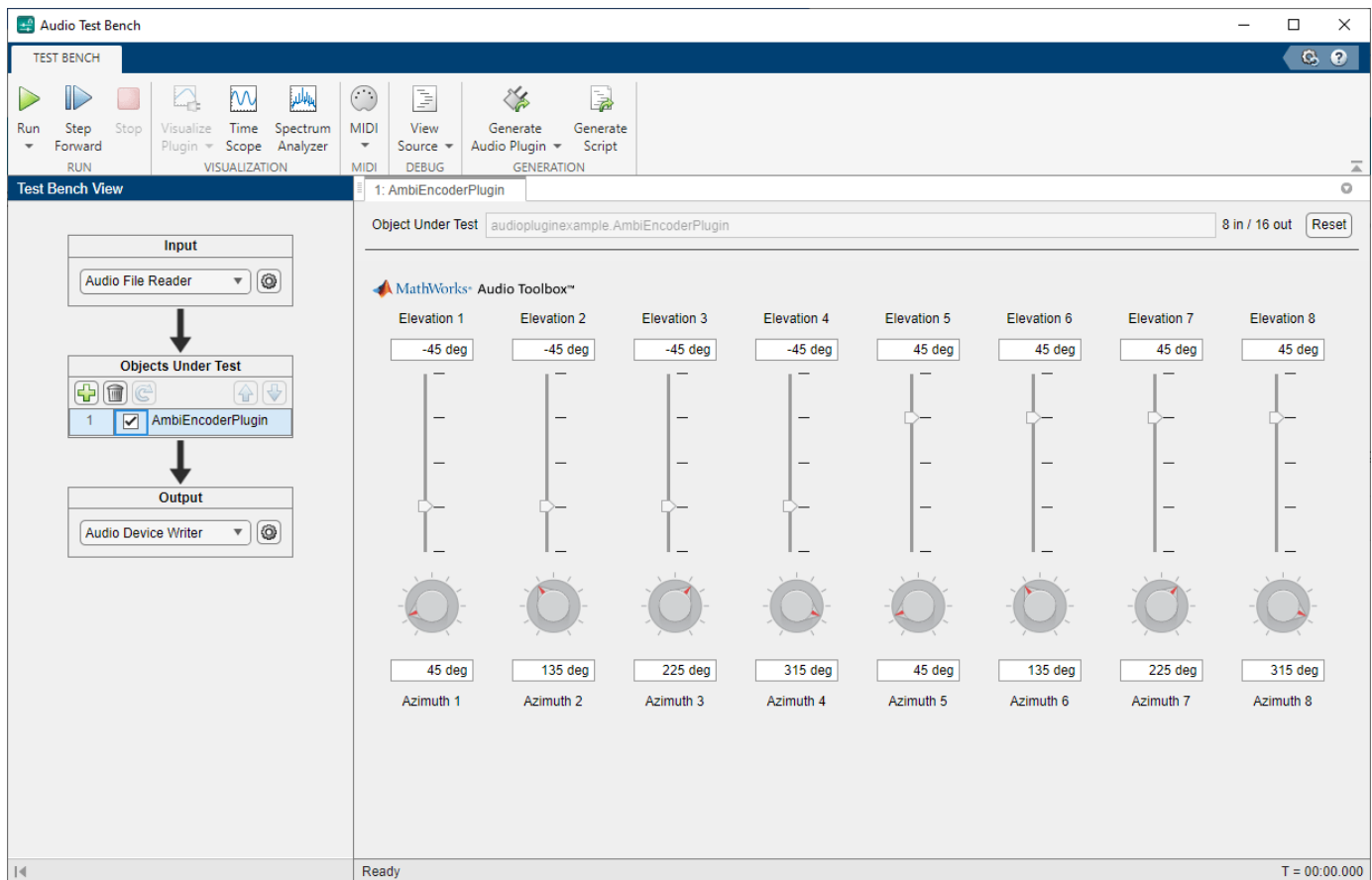
`audiopluginexample.AmbiEncoderPlugin` is built around the `audioexample.ambisonics.ambiencodemtrx` and `audioexample.ambisonics.ambiencode` functions. The number of devices (elements to be encoded) is the number of input channels of the encoder plugin. The ambisonic order determines the number of output 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

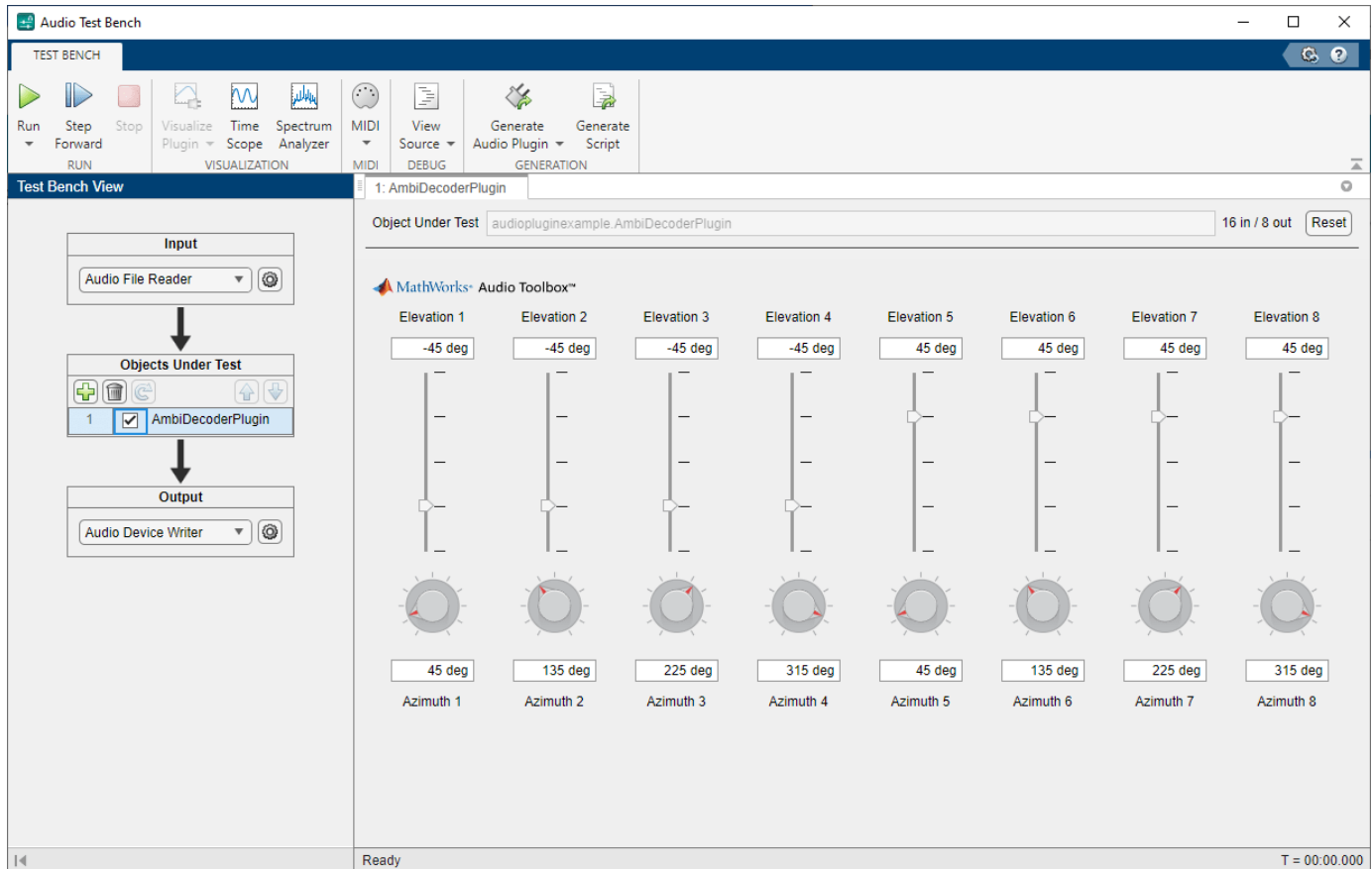
`audiopluginexample.AmbiDecoderPlugin` is built around the `audioexample.ambisonics.ambidecodemtx` and `audioexample.ambisonics.ambidecode` functions. The ambisonic order determines the number of input channels of the decoder plugin. The number of devices (speakers locations) is the number of output 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 1-350

Related Topics

- “Audio Plugins in MATLAB”
- “Audio Plugin Example Gallery” on page 12-2
- “Develop, Analyze, and Debug Plugins In Audio Test Bench” on page 11-2

References

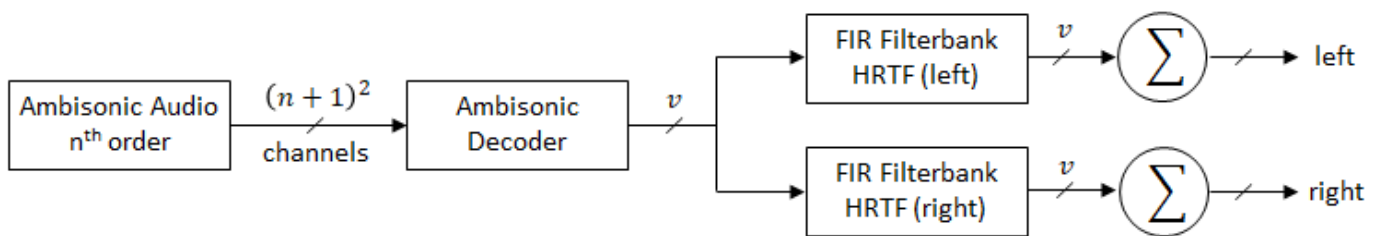
[1] Kronlachner, M. (2014). Spatial Transformations for the Alteration of Ambisonic Recordings (Master's thesis).

[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. 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.



v : Number of virtual loudspeakers

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. 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.ambidecodemtx` helper function. The function returns an ambisonics decoder matrix.

```

order = 7;
devices = sourcePosition(pick,:);
dmtrx = audioexample.ambisonics.ambidecodemtx(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)
```

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.

See Also

"Ambisonic Plugin Generation" on page 1-344

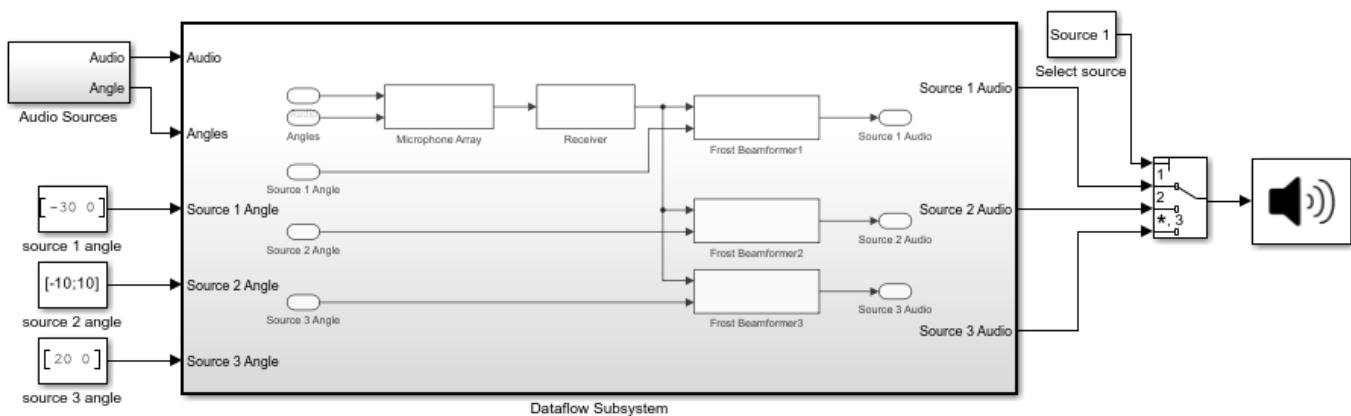
Multicore Simulation of Acoustic Beamforming Using a Microphone Array

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



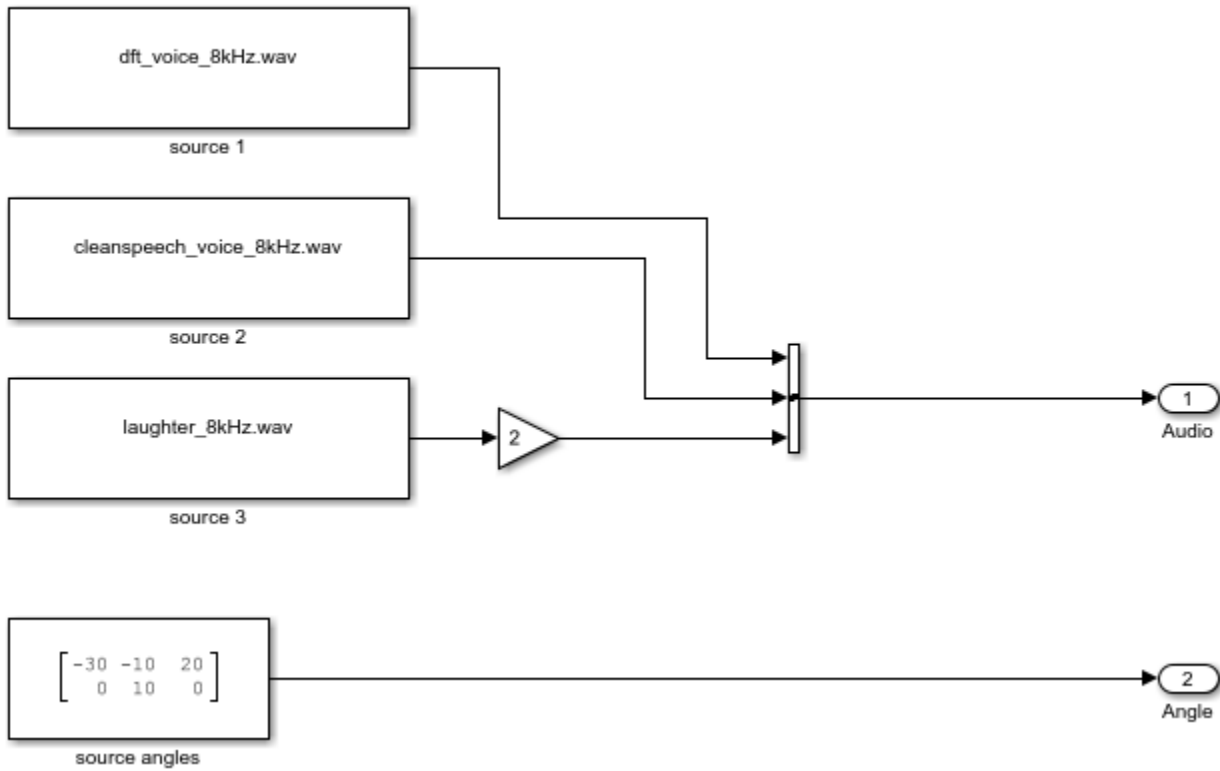
Assumptions:

- Antenna is a 10 element ULA, 5 cm between elements
- The thermal noise at each microphone is 290 K
- Three audio signals from -30 -10 and 20 degrees

Copyright 2016-2022 The Mathworks Inc.

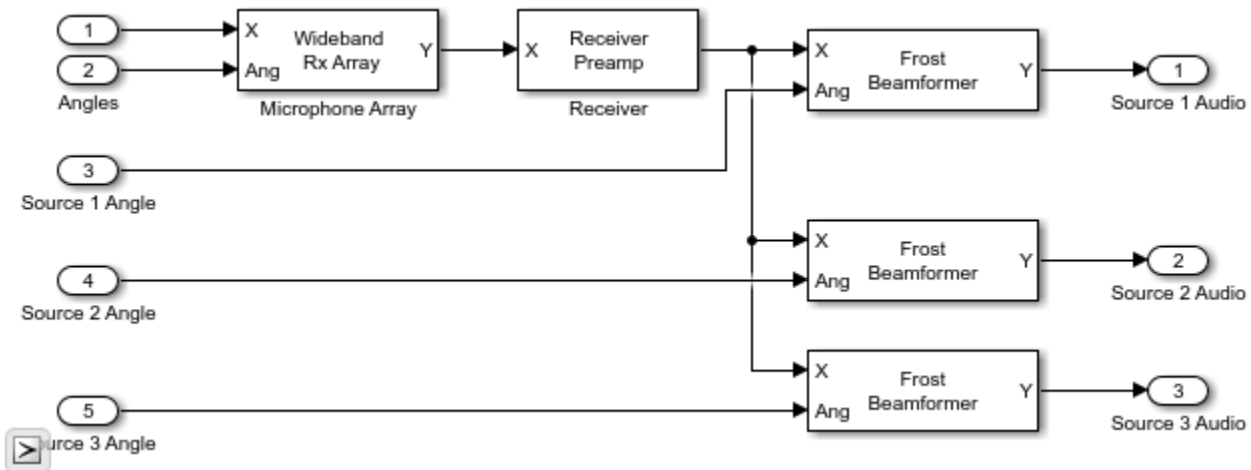
Received Audio Simulation

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.

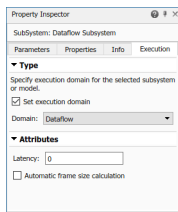


Improve Simulation Performance Using Multithreading

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

This example uses dataflow domain in Simulink to make use of multiple cores on your desktop to improve simulation performance. The **Domain** parameter of the dataflow subsystem in this model is set as **Dataflow**. You can view this by selecting the subsystem and then accessing Property Inspector. To access Property Inspector, in the Simulink Toolstrip, on the Modeling tab, in the Design gallery select Property Inspector or on the Simulation tab, Prepare gallery, select Property Inspector.



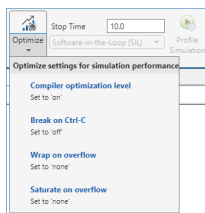
Dataflow domains automatically partition your model into multiple threads for better performance. Once you set the **Domain** parameter to **Dataflow**, you can use the **Multicore** tab analysis to analyze your model to get better performance. The **Multicore** tab is available in the toolstrip when there is a dataflow domain in the model. To learn more about the **Multicore** tab, see “Perform Multicore Analysis for Dataflow”.



Analyzing Concurrency in Dataflow Subsystem

For this example the **Multicore** tab mode is set to **Simulation Profiling** for simulation performance analysis.

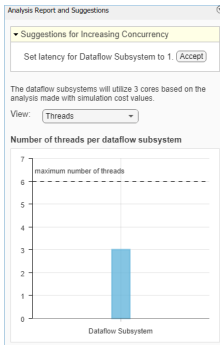
It is recommended to optimize model settings for optimal simulation performance. To accept the proposed model settings, on the **Multicore** tab, click **Optimize**. Alternatively, you can use the drop menu below the **Optimize** button to change the settings individually. In this example the model settings are already optimal.



On the **Multicore** tab, click the **Run Analysis** button to start the analysis of the dataflow domain for simulation performance. Once the analysis is finished, the Analysis Report and Suggestions window shows how many threads the dataflow subsystem uses during simulation.

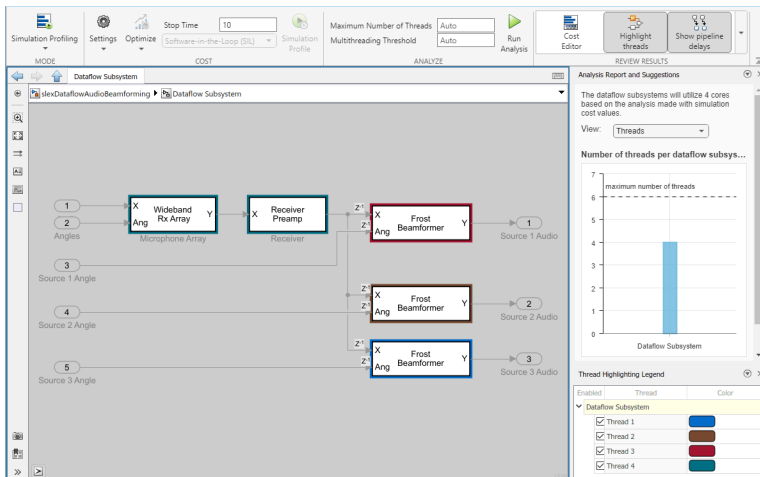
After analyzing the model, the Analysis Report and Suggestions window shows 3 threads. This is because the three Frost beamformer blocks are computationally intensive and can run in parallel. The three Frost beamformer blocks however, depend on the Microphone Array and the Receiver blocks. Pipeline delays can be used to break this dependency and increase concurrency. The Analysis Report and Suggestions window shows the recommended number of pipeline delays as Suggested for Increasing Concurrency. The suggested latency value is computed to give the best performance.

The following diagram shows the Analysis Report and Suggestions window where the suggested latency is 1 for the dataflow subsystem.



Click the **Accept** button to use the recommended latency for the dataflow subsystem. This value can also be entered directly in the Property Inspector for **Latency** parameter. Simulink shows the latency parameter value using Z^{-n} tags at the output ports of the dataflow subsystem.

The Analysis Report and Suggestions window now shows the number of threads as 4 meaning that the blocks inside the dataflow subsystem simulate in parallel using 4 threads. **Highlight threads** highlights the blocks with colors based on their thread allocation as shown in the **Thread Highlighting Legend**. **Show pipeline delays** shows where pipelining delays were inserted within the dataflow subsystem using Z^{-n} tags.



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).

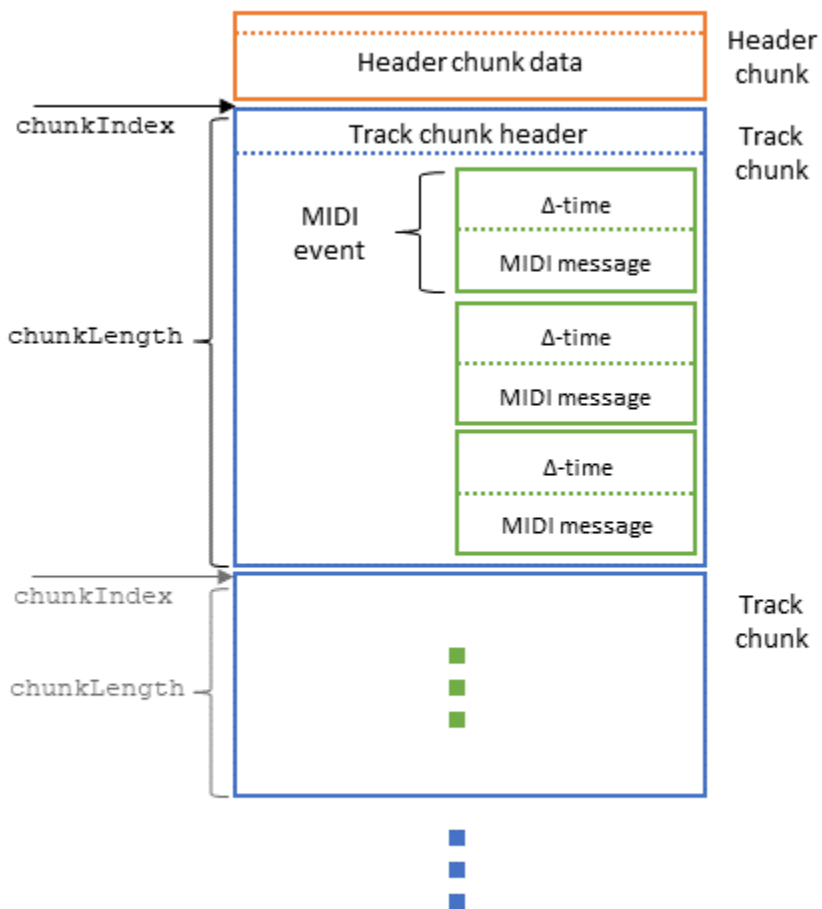
Read MIDI File

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);

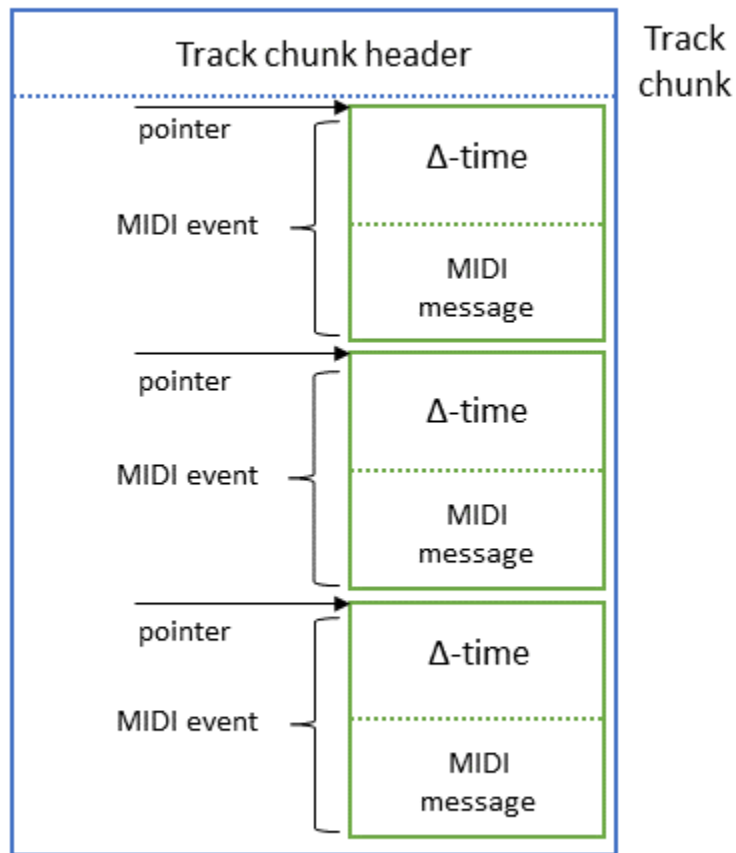
```

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:

$$\text{timeAdd} = \frac{\text{numTicks} \cdot \text{tempo}}{\text{ticksPerQuarterNote} \cdot 1e6}$$

Where `tempo` is in microseconds (μs) per quarter note. To convert beats per minute (BPM) to μs per quarter note, use:

$$\text{tempo} = \frac{6e7}{\text{BPM}}$$

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.

Download Required Files

Before going into the example in detail, you will download a pre-trained network and 4 audio files.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "CocktailPartySourceSeparation");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
dataset = fullfile(dataFolder, "CocktailPartySourceSeparation");
```

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(fullfile(dataset, "MaleSpeech-16-4-mono-20secs.wav"));
sound(mSpeech, Fs)

[fSpeech] = audioread(fullfile(dataset, "FemaleSpeech-16-4-mono-20secs.wav"));
sound(fSpeech, Fs)
```

Combine the two speech sources. Ensure the sources have equal power in the mix. Scale 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);

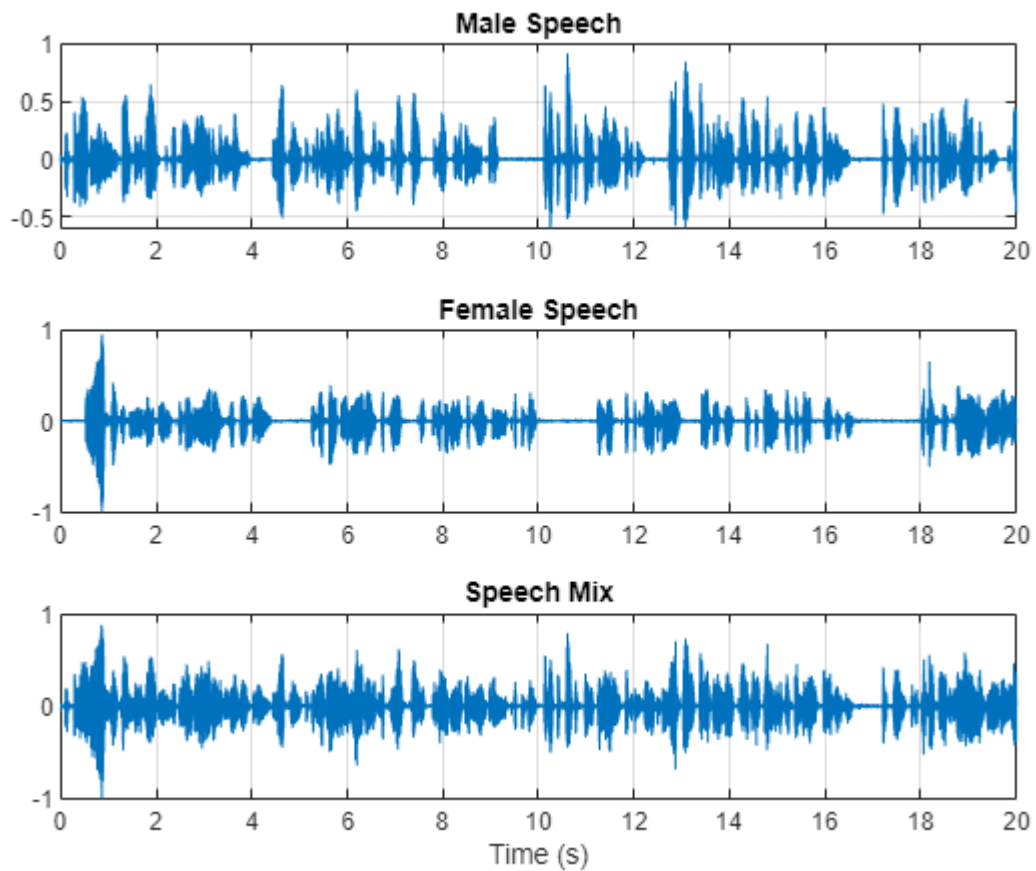
figure(1)
tiledlayout(3,1)

nexttile
plot(t, mSpeech)
```

```
title("Male Speech")
grid on

nexttile
plot(t,fSpeech)
title("Female Speech")
grid on

nexttile
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;
fftLength = 128;
```

```

overlapLength = 96;
win = hann(windowLength, "periodic");

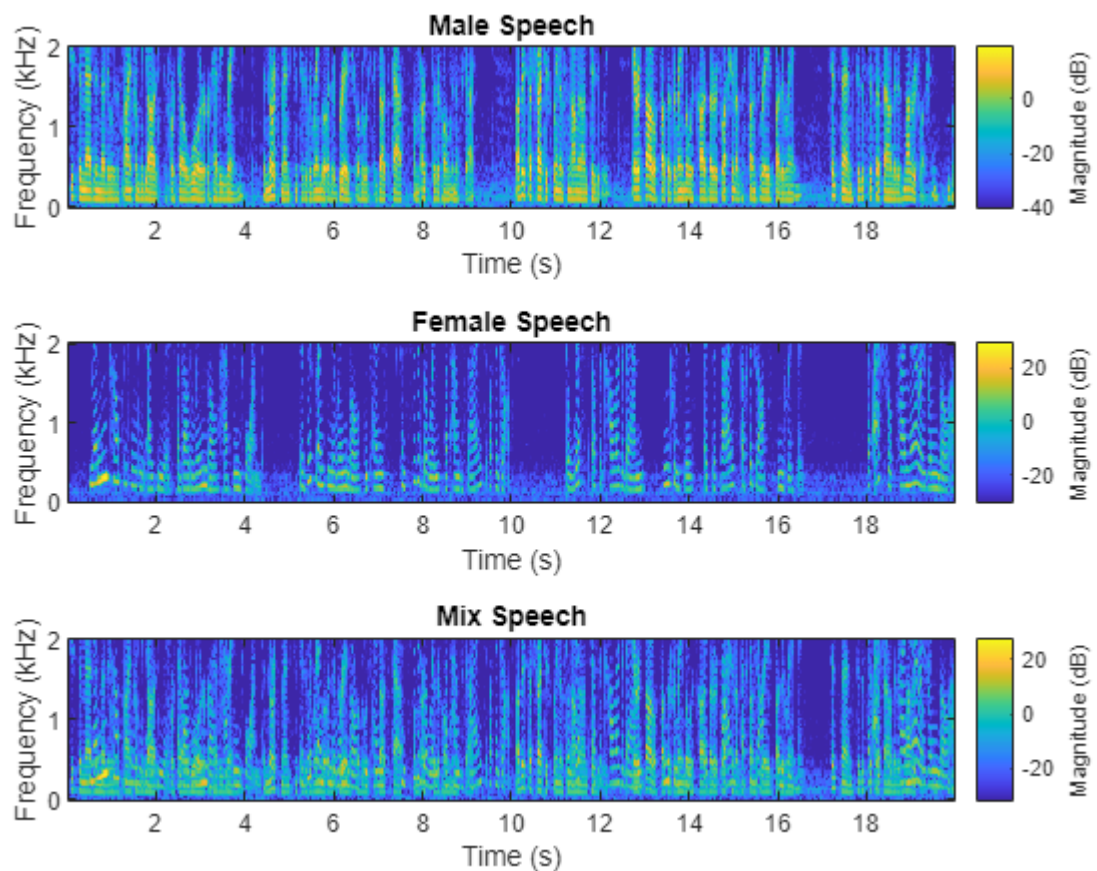
figure(2)
tiledlayout(3,1)

nexttile
stft(mSpeech, Fs, Window=win, OverlapLength=overlapLength, FFTLength=fftLength, FrequencyRange="onesided")
title("Male Speech")

nexttile
stft(fSpeech, Fs, Window=win, OverlapLength=overlapLength, FFTLength=fftLength, FrequencyRange="onesided")
title("Female Speech")

nexttile
stft(mix, Fs, Window=win, OverlapLength=overlapLength, FFTLength=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

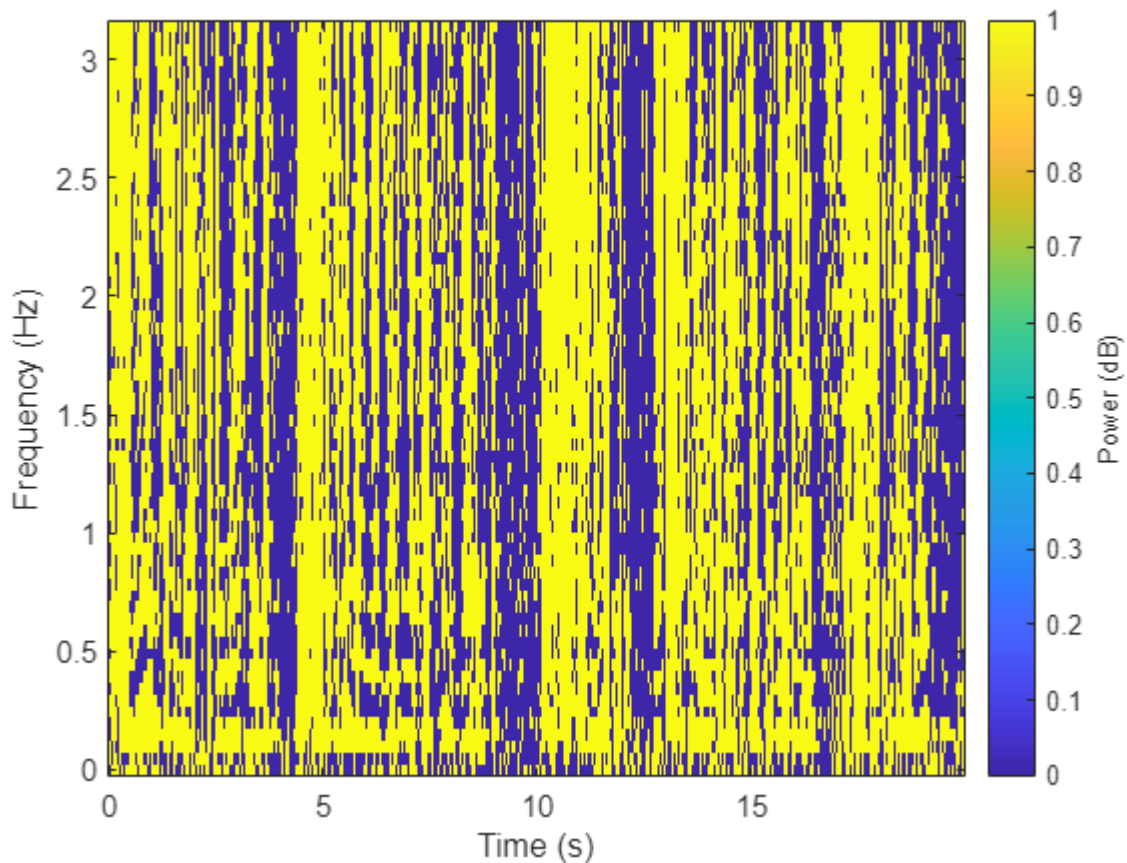
In an ideal binary mask, the mask cell values are either 0 or 1. If the power of the desired source is greater than the combined power of other sources at a particular TF cell, then that cell is set to 1. Otherwise, the cell is set to 0.

Compute the ideal binary mask for the male speaker and then visualize it.

```
P_M = stft(mSpeech,Window=win,OverlapLength=overlapLength,FFTLength=fftLength,FrequencyRange="one");
P_F = stft(fSpeech,Window=win,OverlapLength=overlapLength,FFTLength=fftLength,FrequencyRange="one");
[P_mix,F] = stft(mix,Window=win,OverlapLength=overlapLength,FFTLength=fftLength,FrequencyRange="one");

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,FFTLength=fftLength,Frequency=fftLength)
fSpeech_Hard = istft(P_F_Hard,Window=win,OverlapLength=overlapLength,FFTLength=fftLength,Frequency=fftLength)
```

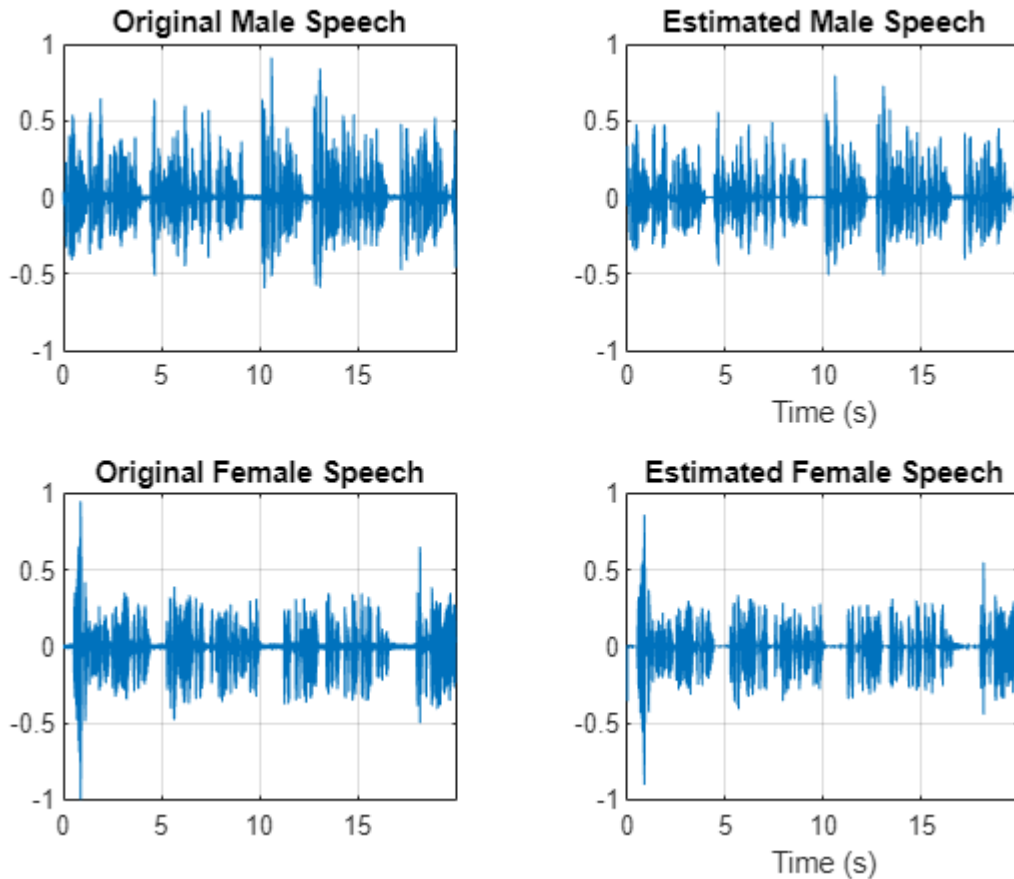
```
figure(4)
tiledlayout(2,2)
```

```
nexttile
plot(t,mSpeech)
axis([t(1) t(end) -1 1])
title("Original Male Speech")
grid on
```

```
nexttile
plot(t,mSpeech_Hard)
axis([t(1) t(end) -1 1])
xlabel("Time (s)")
title("Estimated Male Speech")
grid on
```

```
nexttile
plot(t,fSpeech)
axis([t(1) t(end) -1 1])
title("Original Female Speech")
grid on
```

```
nexttile
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,Frequency=fftLength);
```

```
fSpeech_Soft = istft(P_F_Soft,Window=win,OverlapLength=overlapLength,FFTLength=fftLength,Frequency=fftLength);
```

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.

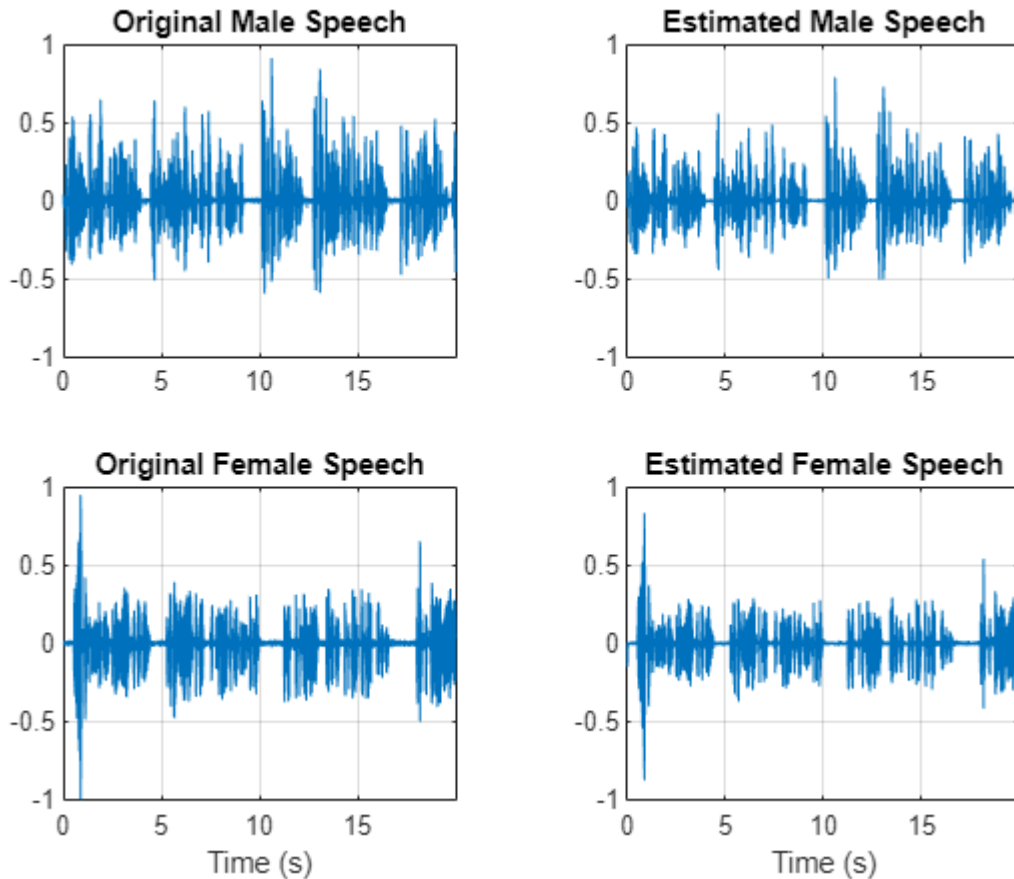
```
figure(5)
tiledlayout(2,2)

nexttile
plot(t,mSpeech)
axis([t(1) t(end) -1 1])
title("Original Male Speech")
grid on

nexttile
plot(t,mSpeech_Soft)
axis([t(1) t(end) -1 1])
title("Estimated Male Speech")
grid on

nexttile
plot(t,fSpeech)
axis([t(1) t(end) -1 1])
xlabel("Time (s)")
title("Original Female Speech")
grid on

nexttile
plot(t,fSpeech_Soft)
axis([t(1) t(end) -1 1])
xlabel("Time (s)")
title("Estimated Female Speech")
grid on
```

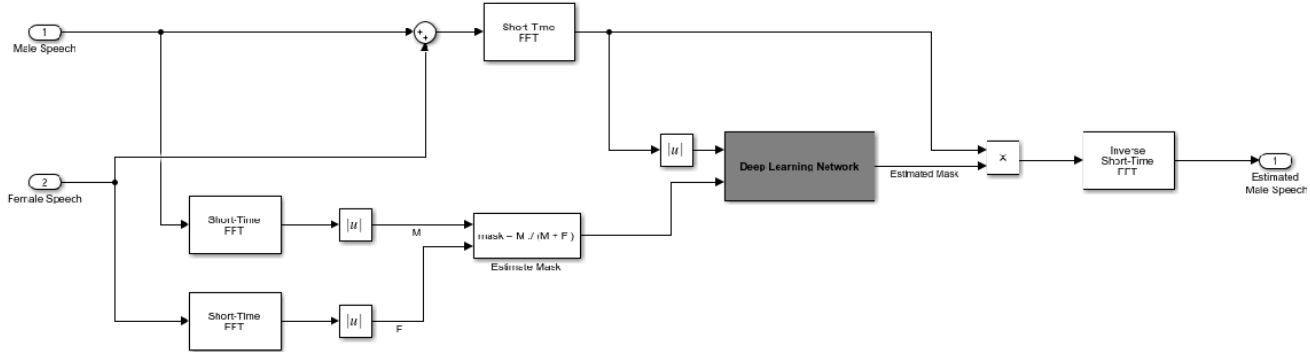
```
sound(mSpeech_Soft, Fs)
```

```
sound(fSpeech_Soft, Fs)
```

Mask Estimation Using Deep Learning

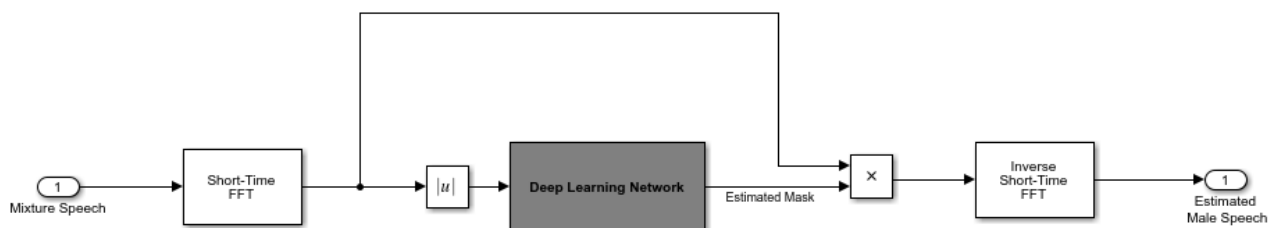
The goal of the deep learning network in this example is to estimate the ideal soft mask described above. The network estimates the mask corresponding to the male speaker. The female speaker mask is derived directly from the male mask.

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.

```
mSpeechTrain = audioread(fullfile(dataset, "MaleSpeech-16-4-mono-405secs.wav"));
fSpeechTrain = audioread(fullfile(dataset, "FemaleSpeech-16-4-mono-405secs.wav"));
```

```
L = min(length(mSpeechTrain), length(fSpeechTrain));
mSpeechTrain = mSpeechTrain(1:L);
fSpeechTrain = fSpeechTrain(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.

```
mSpeechValidate = audioread(fullfile(dataset, "MaleSpeech-16-4-mono-20secs.wav"));
fSpeechValidate = audioread(fullfile(dataset, "FemaleSpeech-16-4-mono-20secs.wav"));
```

```
L = min(length(mSpeechValidate), length(fSpeechValidate));
mSpeechValidate = mSpeechValidate(1:L);
fSpeechValidate = fSpeechValidate(1:L);
```

Scale the training signals to the same power. Scale the validation signals to the same power.

```
mSpeechTrain = mSpeechTrain/norm(mSpeechTrain);
fSpeechTrain = fSpeechTrain/norm(fSpeechTrain);
ampAdj = max(abs([mSpeechTrain; fSpeechTrain]));
```

```
mSpeechTrain = mSpeechTrain/ampAdj;
fSpeechTrain = fSpeechTrain/ampAdj;
```

```
mSpeechValidate = mSpeechValidate/norm(mSpeechValidate);
fSpeechValidate = fSpeechValidate/norm(fSpeechValidate);
ampAdj = max(abs([mSpeechValidate; fSpeechValidate]));
```

```
mSpeechValidate = mSpeechValidate/ampAdj;
fSpeechValidate = fSpeechValidate/ampAdj;
```

Create the training and validation "cocktail party" mixes.

```
mixTrain = mSpeechTrain + fSpeechTrain;
mixTrain = mixTrain/max(mixTrain);
```

```
mixValidate = mSpeechValidate + fSpeechValidate;
mixValidate = mixValidate/max(mixValidate);
```

Generate training STFTs.

```
windowLength = 128;
fftLength = 128;
overlapLength = 128-1;
Fs = 4000;
win = hann(windowLength, "periodic");
```

```
P_mix0 = abs(stft(mixTrain, Window=win, OverlapLength=overlapLength, FFTLength=fftLength, FrequencyRange=[0 Fs/2]));
P_M = abs(stft(mSpeechTrain, Window=win, OverlapLength=overlapLength, FFTLength=fftLength, FrequencyRange=[0 Fs/2]));
P_F = abs(stft(fSpeechTrain, Window=win, OverlapLength=overlapLength, FFTLength=fftLength, FrequencyRange=[0 Fs/2]));
```

Take the log of the mix STFT. Normalize the values by their mean and standard deviation.

```
P_mix = log(P_mix0 + eps);
MP = mean(P_mix(:));
SP = std(P_mix(:));
P_mix = (P_mix - MP)/SP;
```

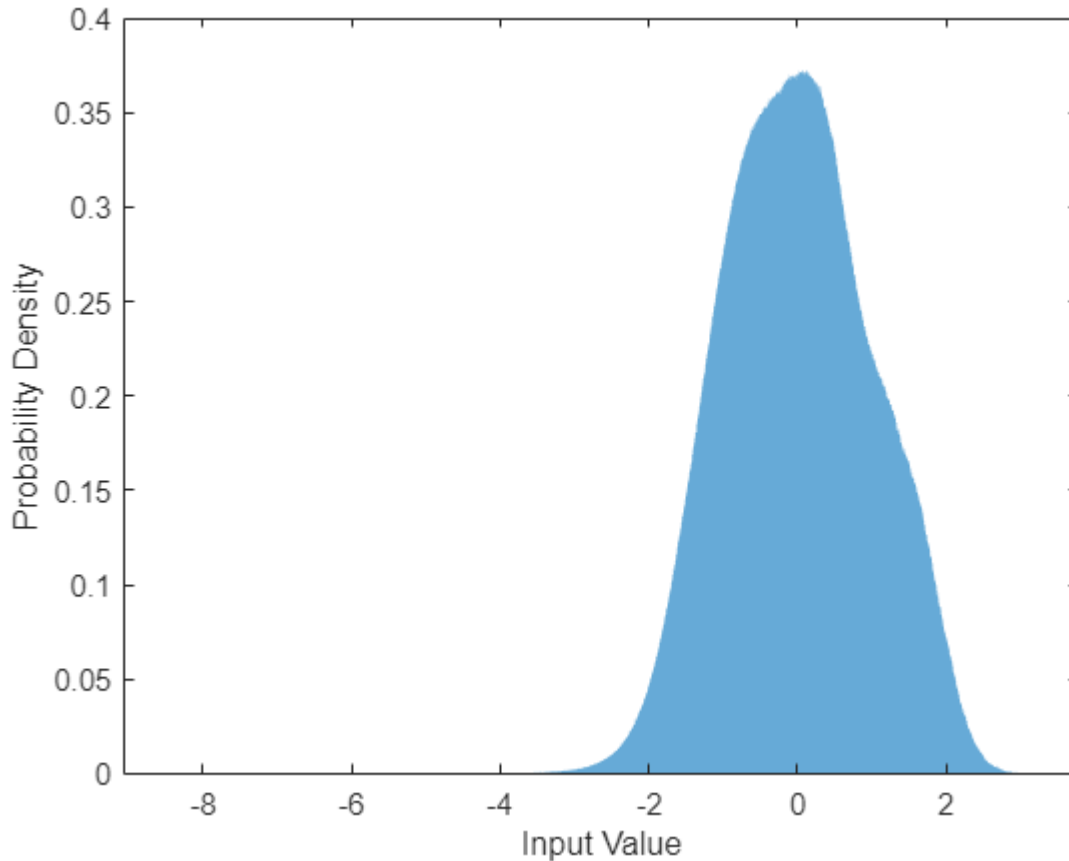
Generate validation STFTs. Take the log of the mix STFT. Normalize the values by their mean and standard deviation.

```
P_Val_mix0 = stft(mixValidate, Window=win, OverlapLength=overlapLength, FFTLength=fftLength, FrequencyRange=[0 Fs/2]);
P_Val_M = abs(stft(mSpeechValidate, Window=win, OverlapLength=overlapLength, FFTLength=fftLength, FrequencyRange=[0 Fs/2]));
P_Val_F = abs(stft(fSpeechValidate, Window=win, OverlapLength=overlapLength, FFTLength=fftLength, FrequencyRange=[0 Fs/2]));
```

```
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);
```

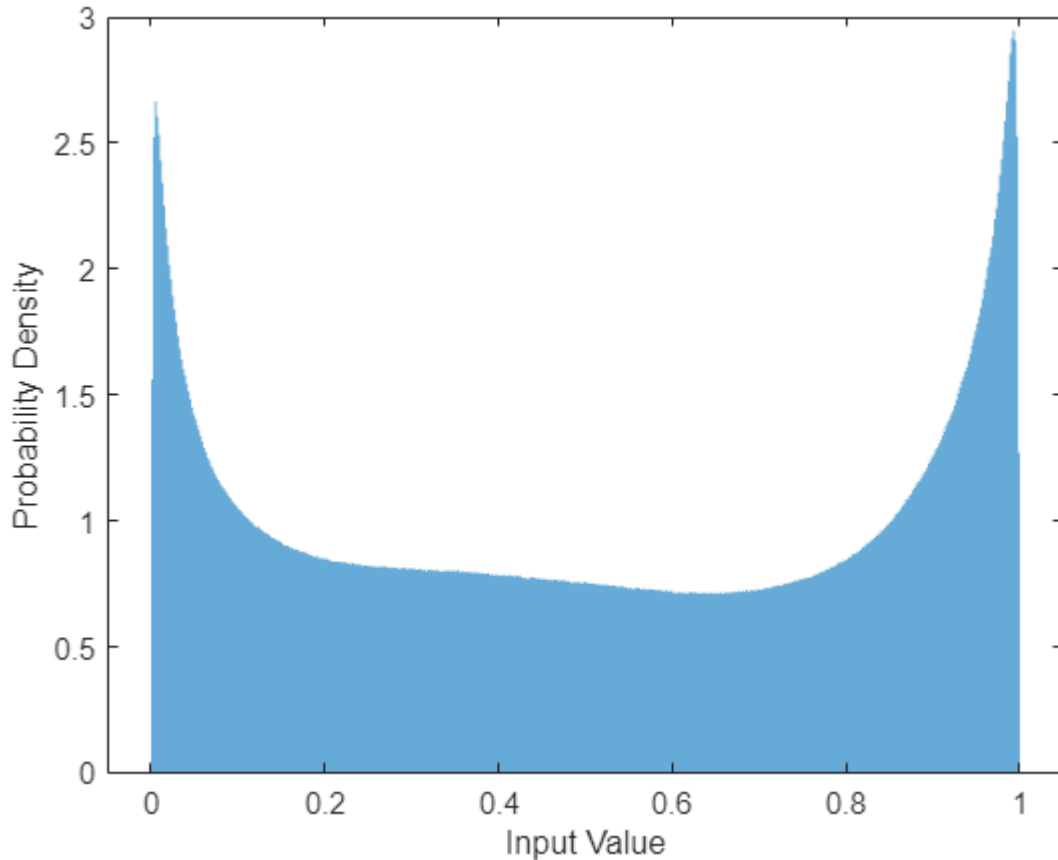
Compute the validation soft mask. Use this mask to evaluate the mask emitted by the trained network.

```
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);
    maskSequences(:, :, :, end+1) = maskTrain(:, loc:loc+seqLen-1);
    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(:,:,end+1) = P_Val_mix(:,loc:loc+seqLen-1);
    maskValSequences(:,:,end+1) = maskValidate(:,loc:loc+seqLen-1);
    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 `speedupExample` to true.

```

speedupExample = ;
if speedupExample
    CocktailPartyNet = trainNetwork(mixSequencesT,maskSequencesT,layers,options);
else
    s = load(fullfile(dataset,"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));

```

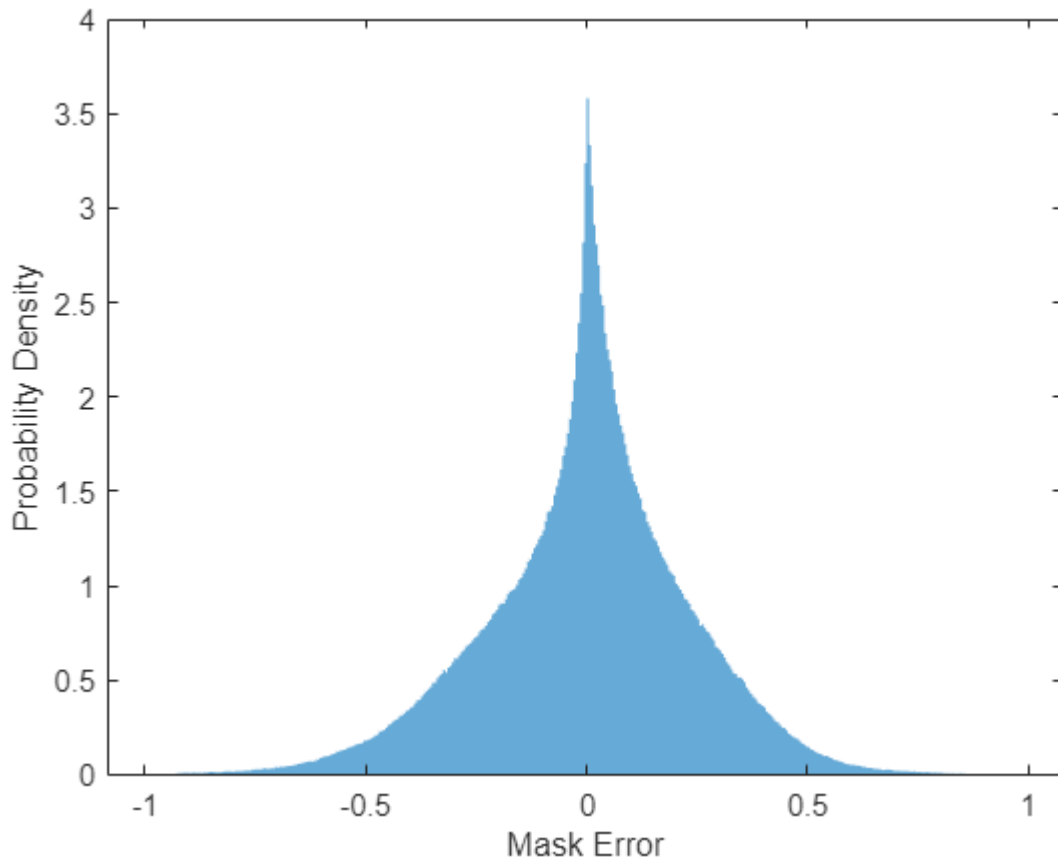
Evaluate Deep Learning Network

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,FFTLengh=fftLength,Frq...
maleSpeech_est_soft = maleSpeech_est_soft/max(abs(maleSpeech_est_soft));
```

Determine a range to analyze and the associated time vector.

```
range = windowLength:numel(maleSpeech_est_soft)-windowLength;
t = range*(1/Fs);
```


Visualize the estimated and original male speech signals. Listen to the estimated soft mask male speech.

```

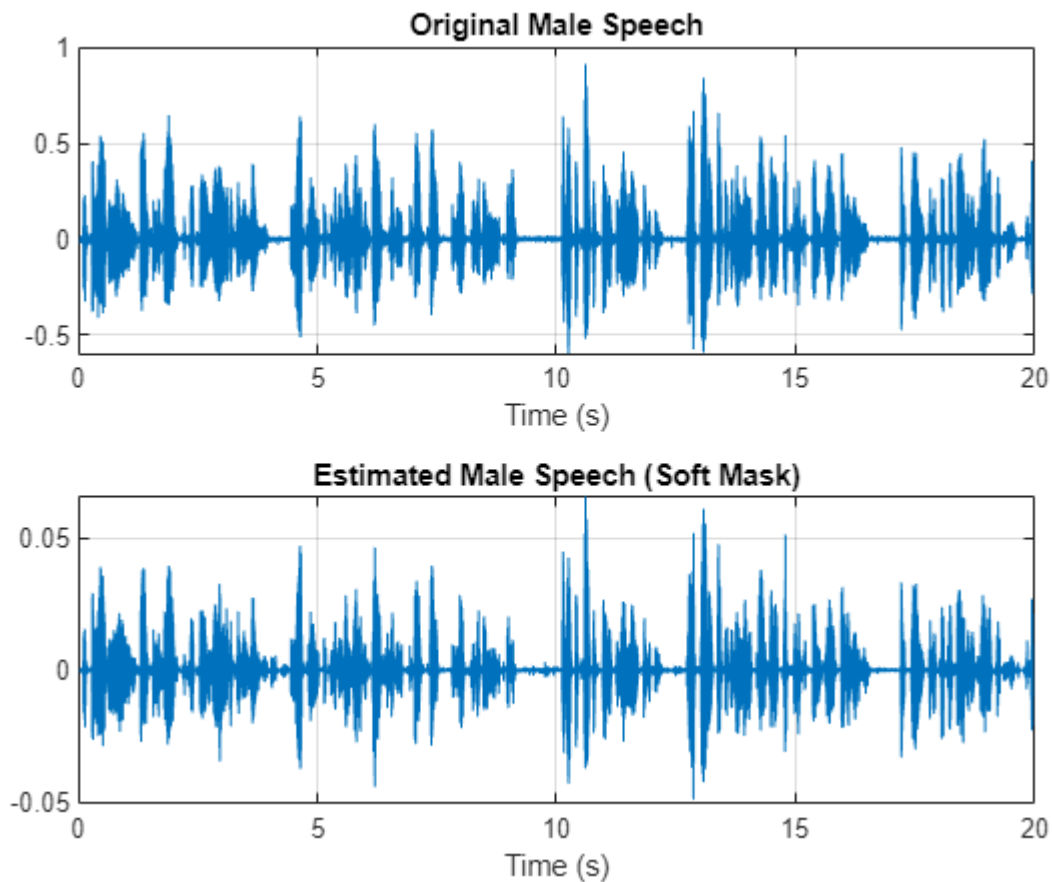
sound(maleSpeech_est_soft(range),Fs)

figure(9)
tiledlayout(2,1)

nexttile
plot(t,mSpeechValidate(range))
title("Original Male Speech")
xlabel("Time (s)")
grid on

nexttile
plot(t,maleSpeech_est_soft(range))
xlabel("Time (s)")
title("Estimated Male Speech (Soft Mask)")
grid on

```



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);
femaleSpeech_est_soft = femaleSpeech_est_soft/max(femaleSpeech_est_soft);
```

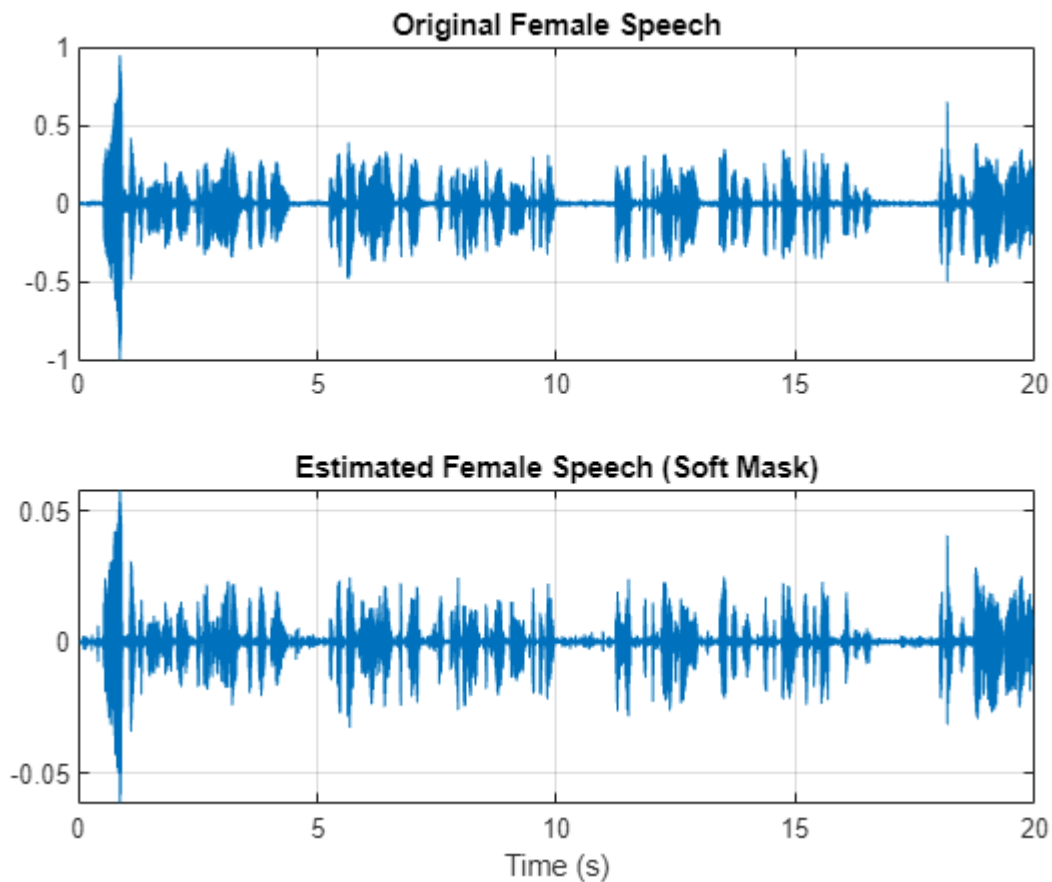
Visualize the estimated and original female signals. Listen to the estimated female speech.

```
sound(femaleSpeech_est_soft(range),Fs)
```

```
figure(10)
tiledlayout(2,1)
```

```
nexttile
plot(t,fSpeechValidate(range))
title("Original Female Speech")
grid on
```

```
nexttile
plot(t,femaleSpeech_est_soft(range))
xlabel("Time (s)")
title("Estimated Female Speech (Soft Mask)")
grid on
```



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,FFTLengh=fftLength,Fr
maleSpeech_est_hard = maleSpeech_est_hard/max(maleSpeech_est_hard);
```

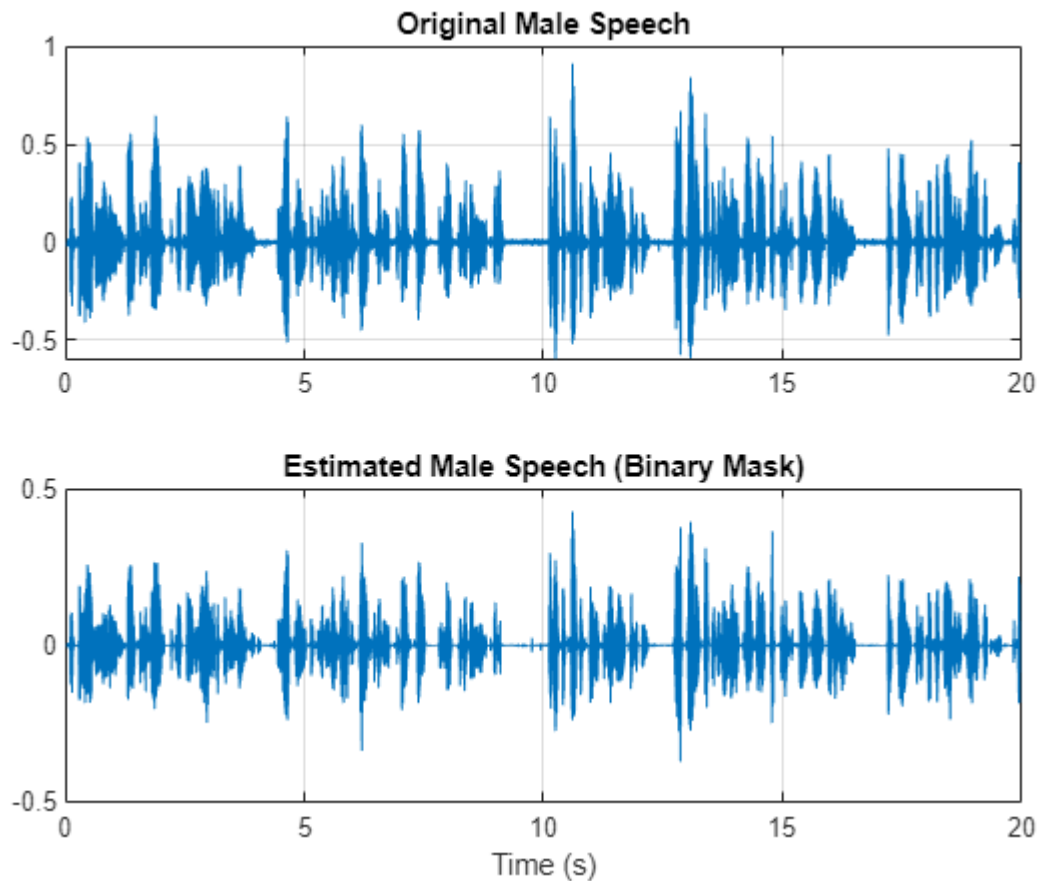
Visualize the estimated and original male speech signals. Listen to the estimated binary mask male speech.

```
sound(maleSpeech_est_hard(range),Fs)
```

```
figure(11)
tiledlayout(2,1)
```

```
nexttile
plot(t,mSpeechValidate(range))
title("Original Male Speech")
grid on
```

```
nexttile
plot(t,maleSpeech_est_hard(range))
xlabel("Time (s)")
title("Estimated Male Speech (Binary Mask)")
grid on
```



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,FFTLenght=fftLength);
femaleSpeech_est_hard = femaleSpeech_est_hard/max(femaleSpeech_est_hard);
```

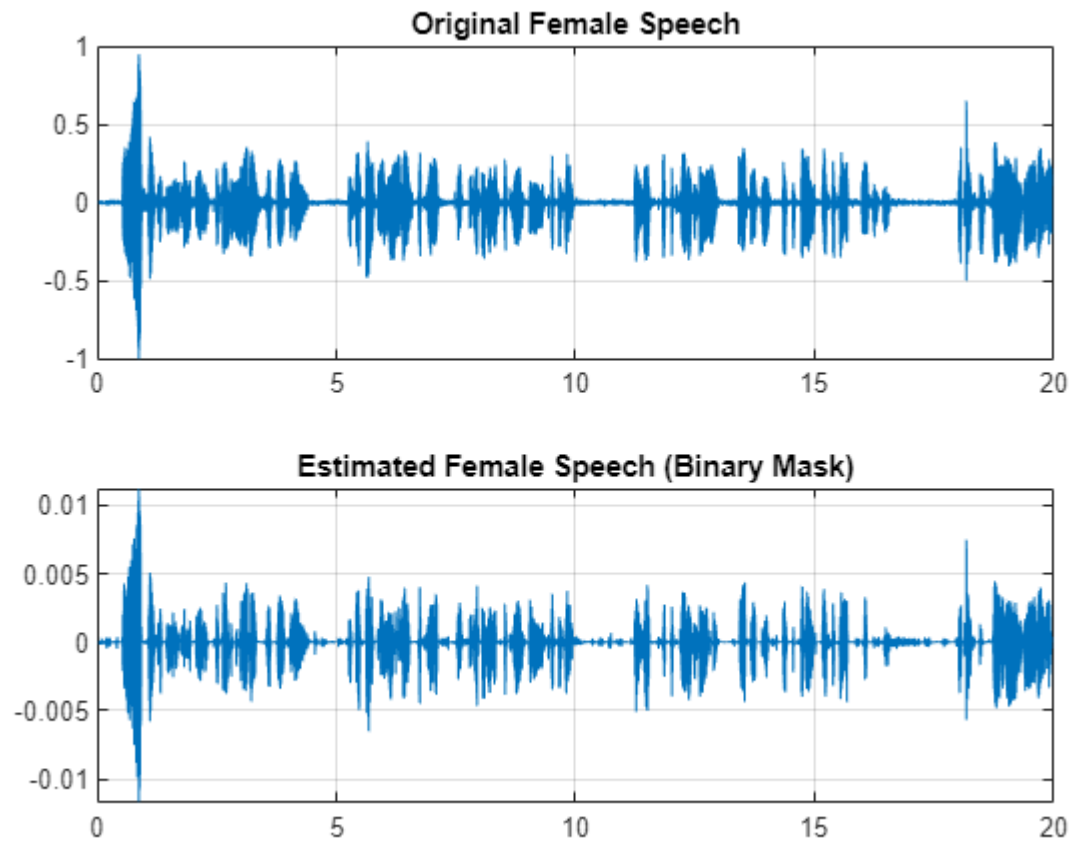
Visualize the estimated and original female speech signals. Listen to the estimated female speech.

```
sound(femaleSpeech_est_hard(range),Fs)
```

```
figure(12)
tiledlayout(2,1)
```

```
nexttile
plot(t,fSpeechValidate(range))
title("Original Female Speech")
grid on
```

```
nexttile
plot(t,femaleSpeech_est_hard(range))
title("Estimated Female Speech (Binary Mask)")
grid on
```

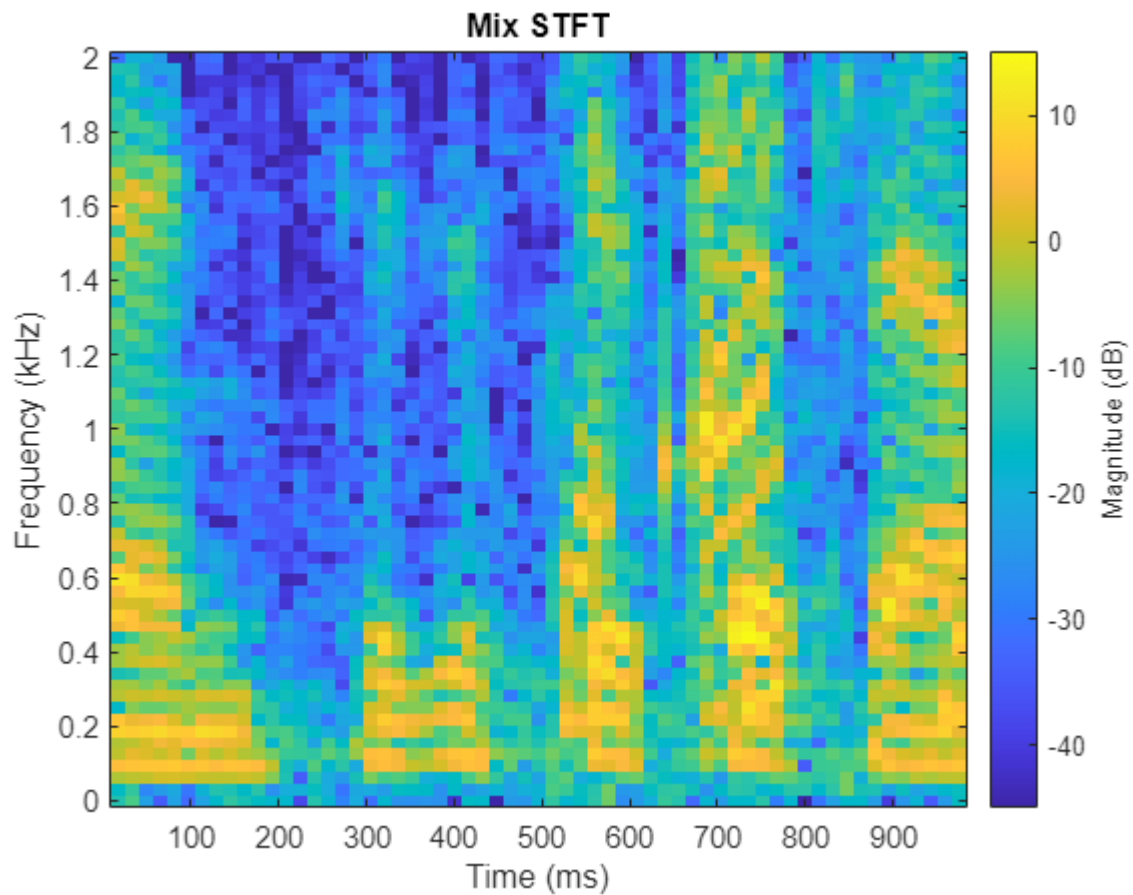


Compare STFTs of a one-second segment for mix, original female and male, and estimated female and male, respectively.

```
range = 7e4:7.4e4;
```

```
figure(13)
```

```
stft(mixValidate(range),Fs,Window=win,OverlapLength=64,FFTLength=fftLength,FrequencyRange="onesi  
title("Mix STFT")
```

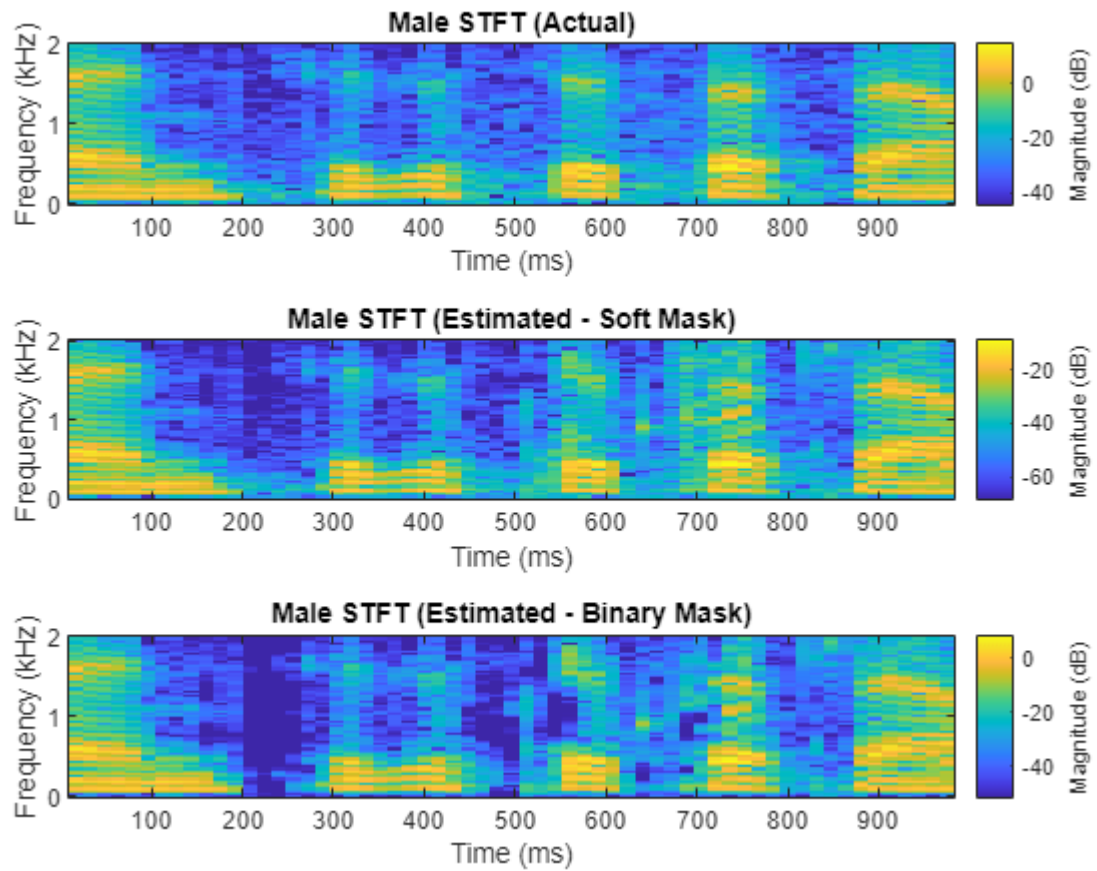


```
figure(14)
tiledlayout(3,1)
```

```
nexttile
stft(mSpeechValidate(range),Fs,Window=win,OverlapLength=64,FFTLength=fftLength,FrequencyRange="original")
title("Male STFT (Actual)")
```

```
nexttile
stft(maleSpeech_est_soft(range),Fs,Window=win,OverlapLength=64,FFTLength=fftLength,FrequencyRange="original")
title("Male STFT (Estimated - Soft Mask)")
```

```
nexttile
stft(maleSpeech_est_hard(range),Fs,Window=win,OverlapLength=64,FFTLength=fftLength,FrequencyRange="original")
title("Male STFT (Estimated - Binary Mask)");
```



```
figure(15)
tiledlayout(3,1)
```

```
nexttile
```

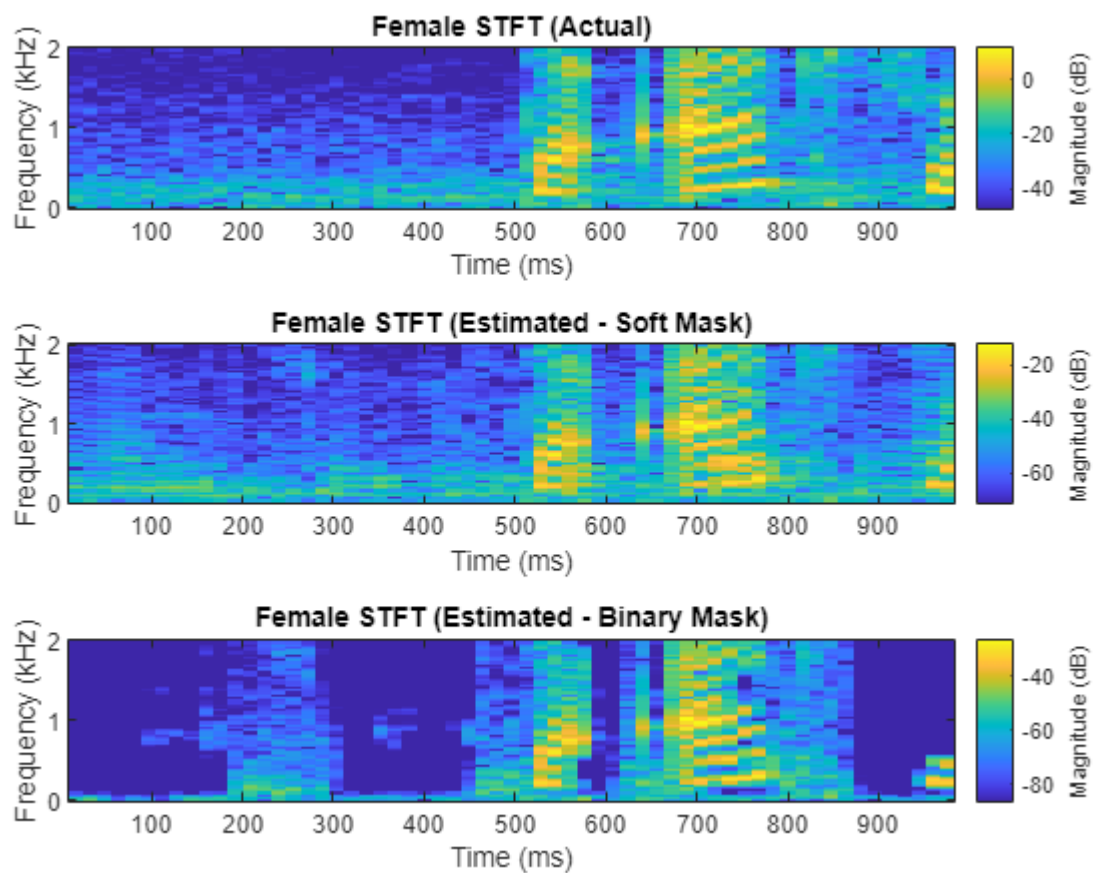
```
stft(fSpeechValidate(range),Fs,Window=win,OverlapLength=64,FFTLength=fftLength,FrequencyRange="original",...
title("Female STFT (Actual)")
```

```
nexttile
```

```
stft(femaleSpeech_est_soft(range),Fs,Window=win,OverlapLength=64,FFTLength=fftLength,FrequencyRange="original",...
title("Female STFT (Estimated - Soft Mask)")
```

```
nexttile
```

```
stft(femaleSpeech_est_hard(range),Fs,Window=win,OverlapLength=64,FFTLength=fftLength,FrequencyRange="original",...
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.

See Also

Related Examples

- "End-to-End Deep Speech Separation" on page 1-83

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.

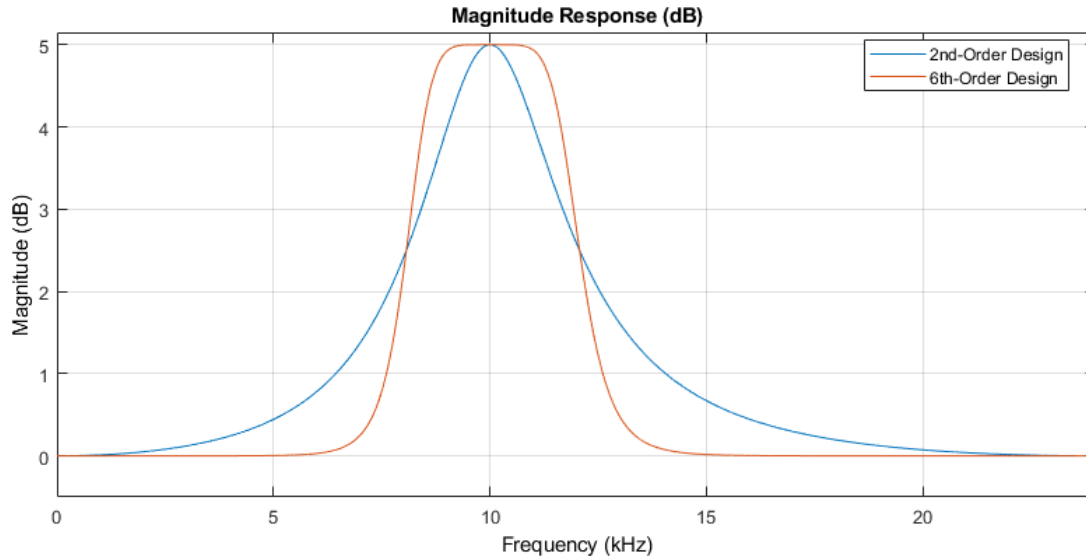
Typically, parametric equalizers are designed as second-order IIR filters. These filters have the drawback that because of their low order, they can present relatively large ripple or transition regions and may overlap with each other when several of them are connected in cascade. Audio Toolbox™ provides 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 uses `designParamEQ`. It is a simple function that provides support for the 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.

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, 'Orientation', 'row');
[B2,A2] = designParamEQ(N2,G,Wo,BW, 'Orientation', 'row');
BQ1 = dsp.SOSFilter('Numerator',B1,'Denominator',A1);
BQ2 = dsp.SOSFilter('Numerator',B2,'Denominator',A2);
hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white');
legend(hfvt,'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).

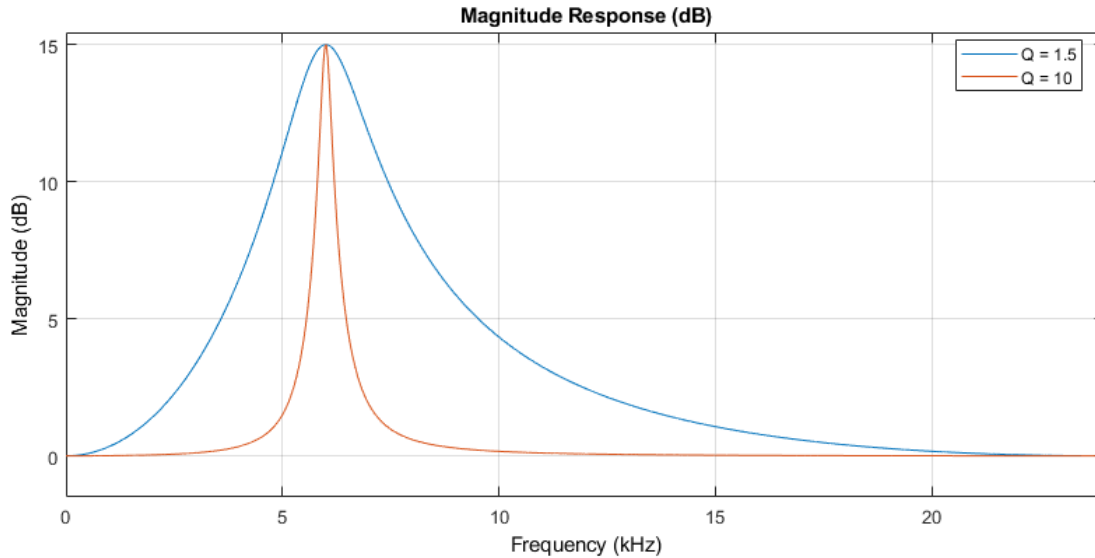
Designs Based on Quality Factor

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,'Orientation','row');
[B2,A2] = designParamEQ(N,G,Wo,BW2,'Orientation','row');
BQ1 = dsp.SOSFilter('Numerator',B1,'Denominator',A1);
BQ2 = dsp.SOSFilter('Numerator',B2,'Denominator',A2);
hfv = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white');
legend(hfv,'Q = 1.5','Q = 10');

```

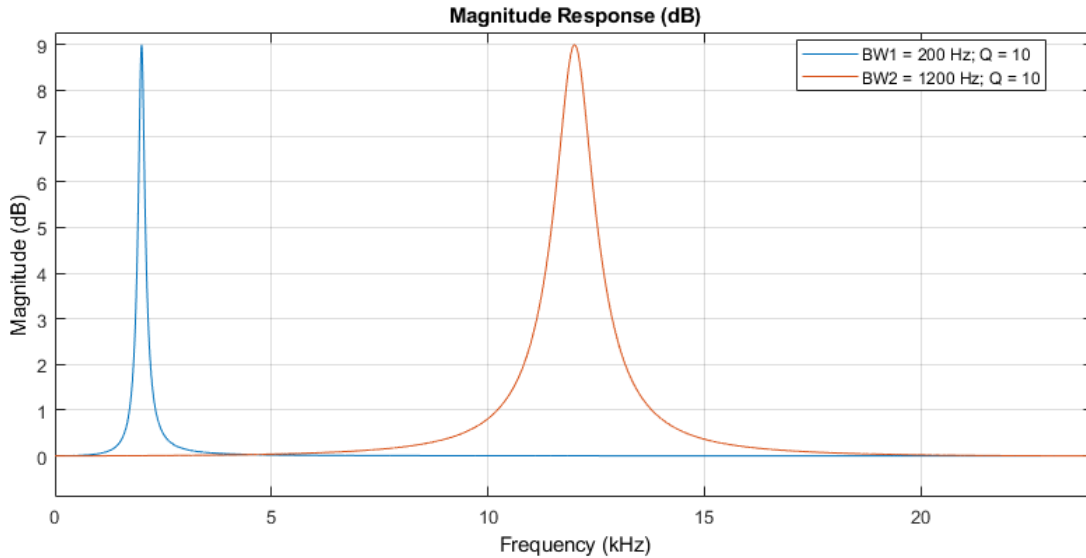


Although a higher Q factor corresponds to a sharper filter, it must also be noted that for a given bandwidth, the Q factor increases simply by increasing the center frequency. This might seem unintuitive. For example, the following two filters have the same Q factor, but one clearly occupies a larger bandwidth than the other.

```

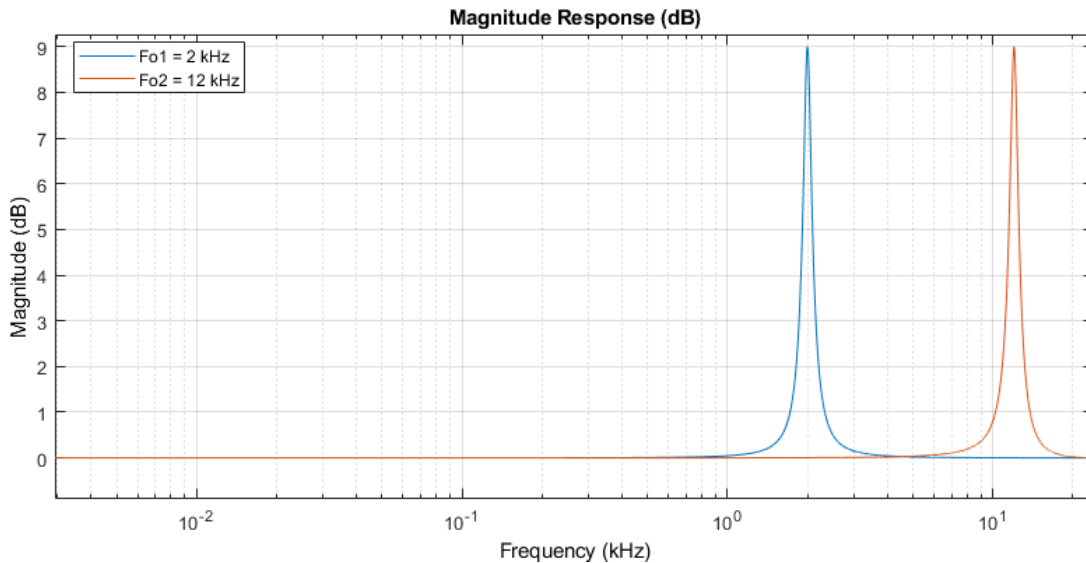
Fs = 48e3;
N = 2;
Q = 10;
G = 9; % 9 dB
Wo1 = 2000/(Fs/2);
Wo2 = 12000/(Fs/2);
BW1 = Wo1/Q;
BW2 = Wo2/Q;
[B1,A1] = designParamEQ(N,G,Wo1,BW1,'Orientation','row');
[B2,A2] = designParamEQ(N,G,Wo2,BW2,'Orientation','row');
BQ1 = dsp.SOSFilter('Numerator',B1,'Denominator',A1);
BQ2 = dsp.SOSFilter('Numerator',B2,'Denominator',A2);
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

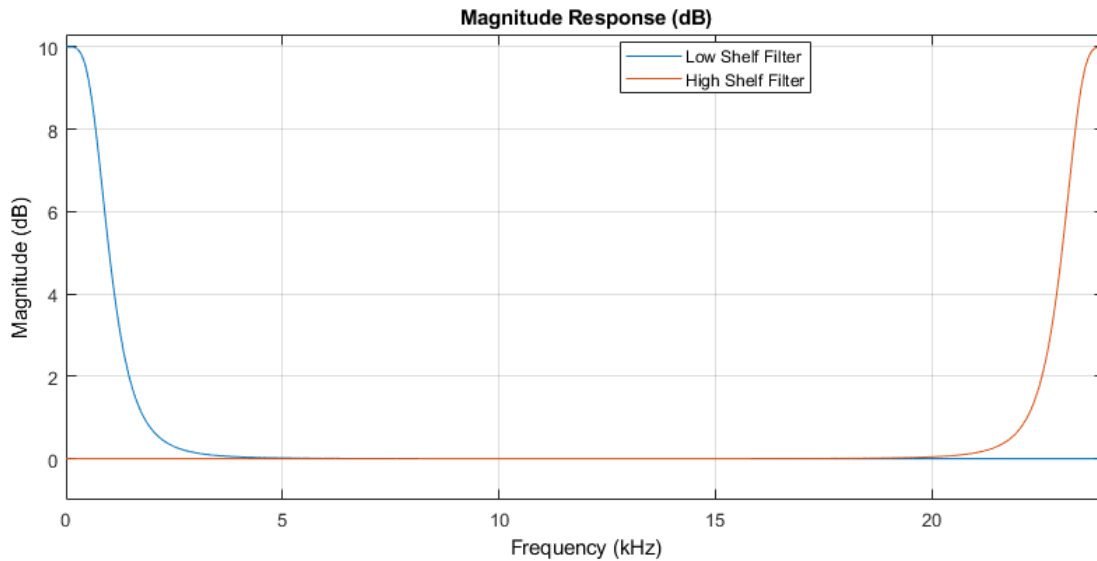
The filter's bandwidth BW is only perfectly centered around the center frequency W_o when such frequency is set to $0.5 \cdot \pi$ (half the Nyquist rate). When W_o is closer to 0 or to π , there is a warping effect that makes a larger portion of the bandwidth to occur at one side of the center frequency. 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,'Orientation','row');
[B2,A2] = designParamEQ(N,G,Wo2,BW,'Orientation','row');
BQ1 = dsp.SOSFilter('Numerator',B1,'Denominator',A1);
BQ2 = dsp.SOSFilter('Numerator',B2,'Denominator',A2);
hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white');
legend(hfvt,'Low Shelf Filter','High Shelf Filter');

```



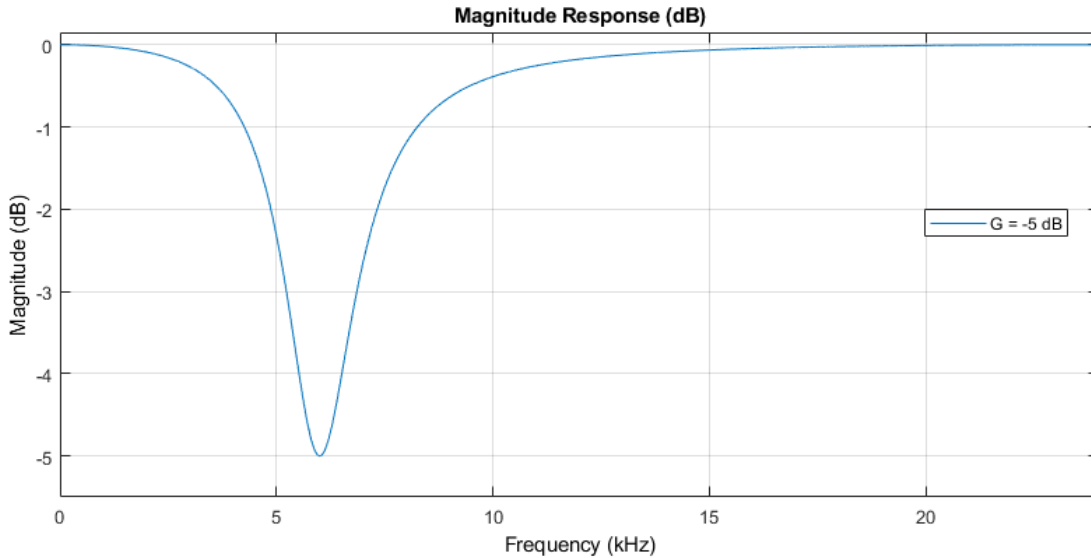
A Parametric Equalizer That Cuts

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,'Orientation','row');
BQ = dsp.SOSFilter('Numerator',B,'Denominator',A);
hfvt = fvtool(BQ,'Fs',Fs,'Color','white');
legend(hfvt,'G = -5 dB');

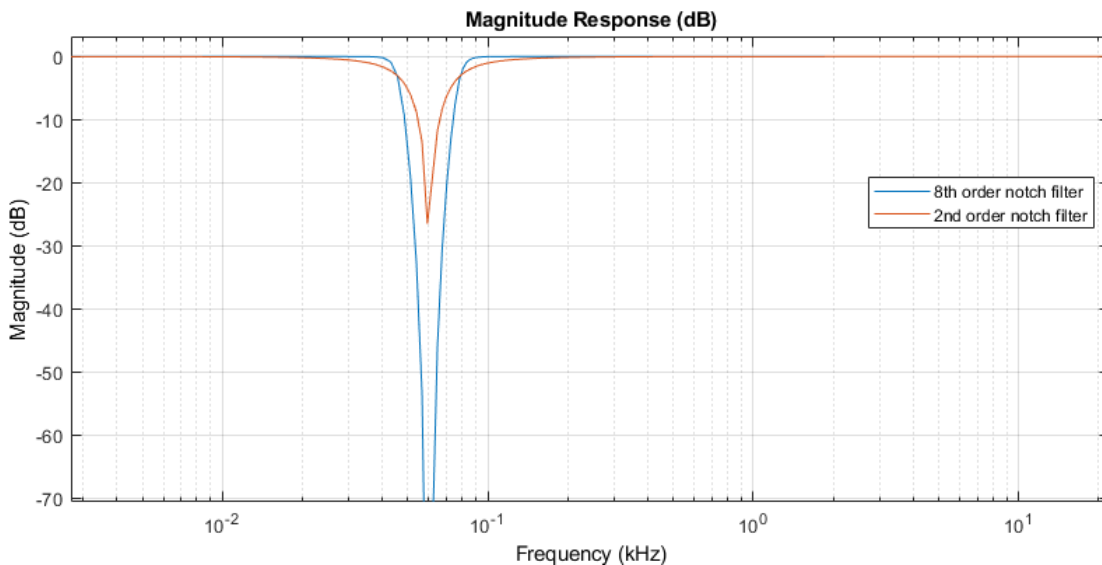
```



At the limit, the filter can be designed to have a gain of zero (-Inf dB) at the frequency specified. This allows to design 2nd order or higher order notch filters.

```

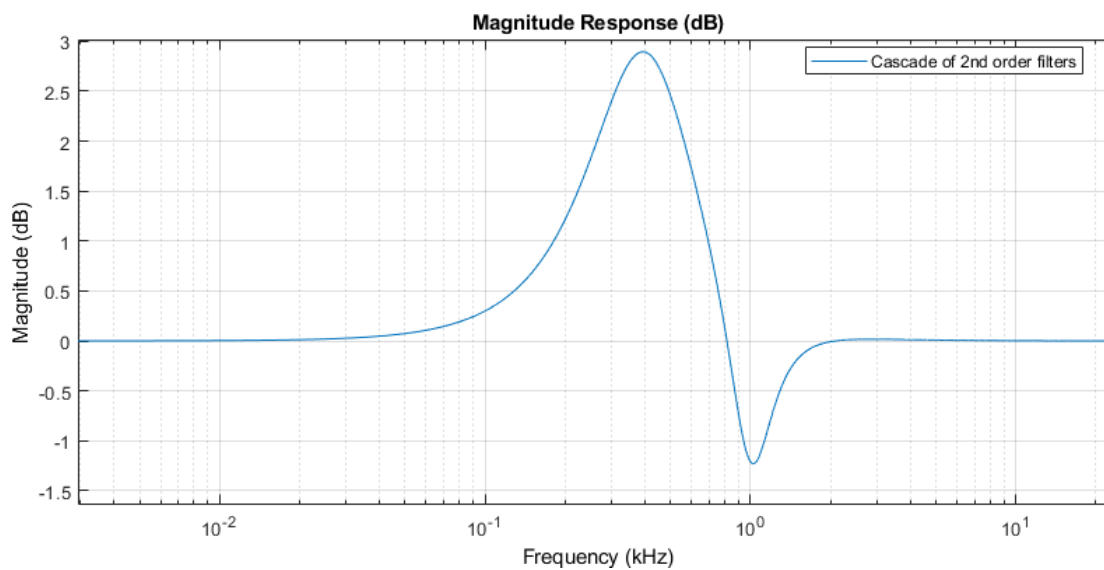
Fs = 44.1e3;
N = 8;
G = -inf;
Q = 1.8;
Wo = 60/(Fs/2); % Notch at 60 Hz
BW = Wo/Q; % Bandwidth will occur at -3 dB for this special case
[B1,A1] = designParamEQ(N,G,Wo,BW,'Orientation','row');
[NUM,DEN] = iirnotch(Wo,BW); % or [NUM,DEN] = designParamEQ(2,G,Wo,BW);
BQ1 = dsp.SOSFilter('Numerator',B1,'Denominator',A1);
BQ2 = dsp.SOSFilter('Numerator',NUM,'Denominator',DEN);
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`.

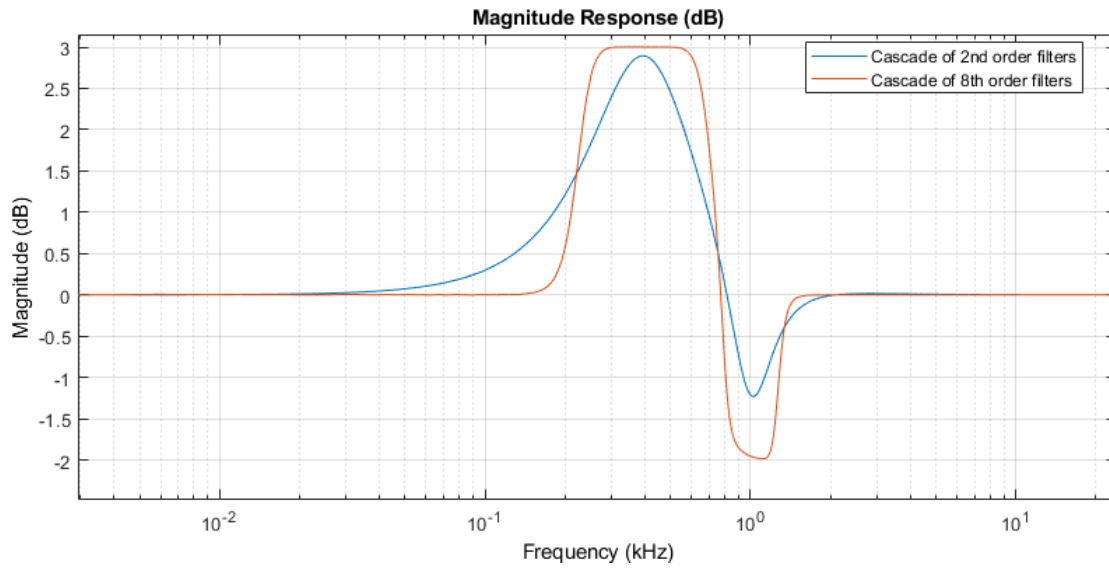
```
Fs = 48e3;
N = 2;
G1 = 3; % 3 dB
G2 = -2; % -2 dB
Wo1 = 400/(Fs/2);
Wo2 = 1000/(Fs/2);
BW = 500/(Fs/2); % Bandwidth occurs at 7.4 dB in this case
[B1,A1] = designParamEQ(N,G1,Wo1,BW,'Orientation','row');
[B2,A2] = designParamEQ(N,G2,Wo2,BW,'Orientation','row');
BQ1 = dsp.SOSFilter('Numerator',B1,'Denominator',A1);
BQ2 = dsp.SOSFilter('Numerator',B2,'Denominator',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,'Orientation','row');
[B2,A2] = designParamEQ(N,G2,Wo2,BW,'Orientation','row');
BQ1a = dsp.SOSFilter('Numerator',B1,'Denominator',A1);
BQ2a = dsp.SOSFilter('Numerator',B2,'Denominator',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');
```



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

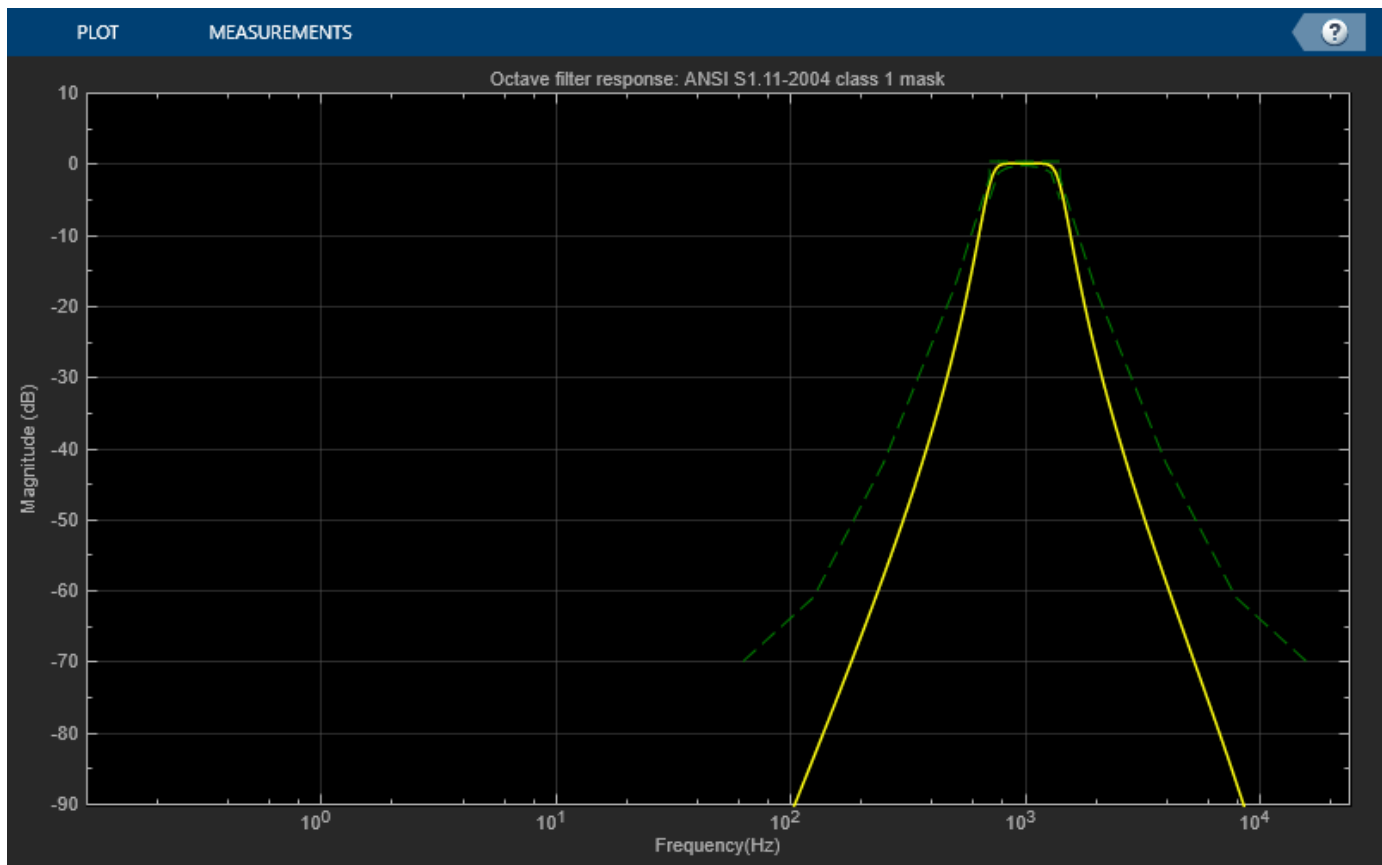
An octave is the interval between two frequencies having a ratio of 2:1 (or $10^{3/10} \approx 1.995$ for base-10 octave ratios). An octave-band or fractional-octave-band filter is a bandpass filter determined by its center frequency, order, and bandwidth. The magnitude attenuation limits are defined 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.

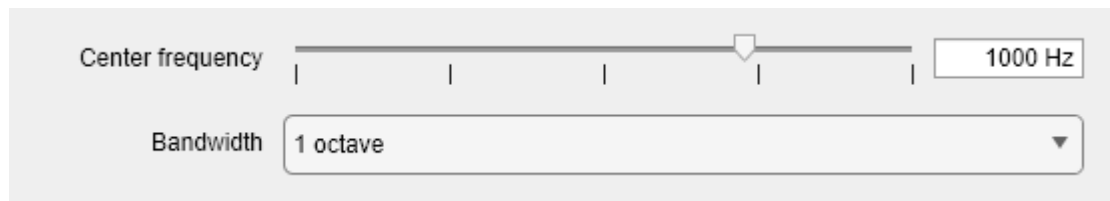
```
visualize(of,"class 1")
```



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)
```



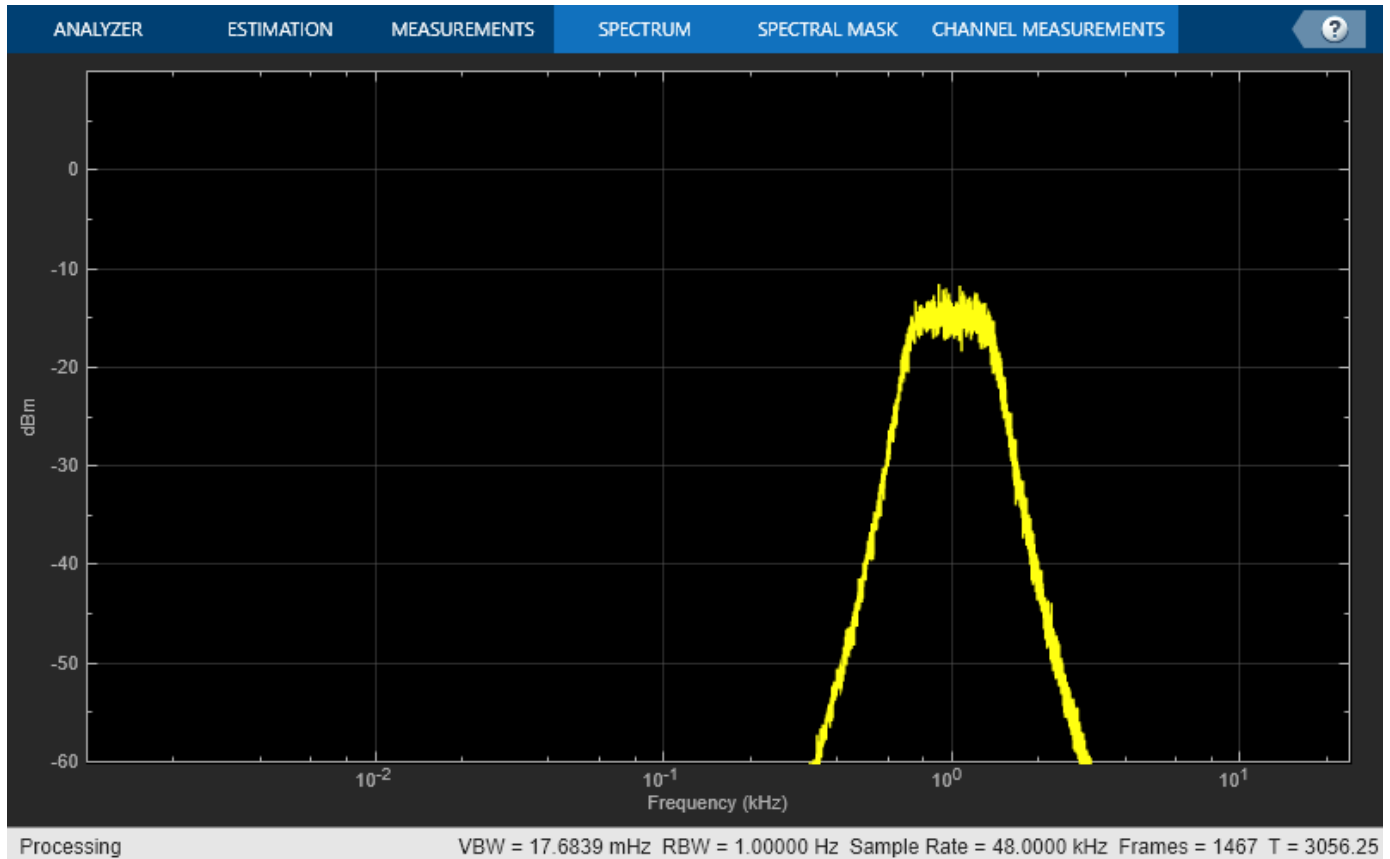
Open a spectrum analyzer to display white noise filtered by the octave filter. You can modify the filter settings with the parameter tuner while the loop runs.

```
Nx = 100000;
scope1 = 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

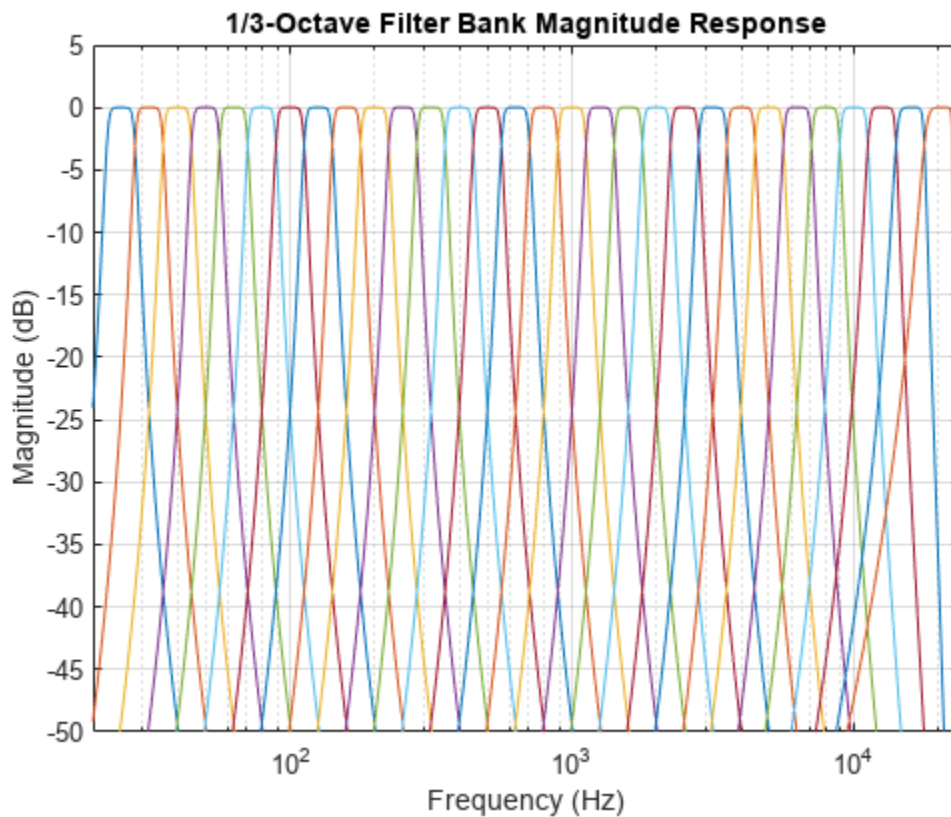
Many applications require a complete set of octave filters to form a filter bank. To design each filter manually, you would use `getANSICenterFrequencies(of)` to get a list of center frequencies for each individual filter. However, it is usually much simpler to use the `octaveFilterBank` object.

Create an `octaveFilterBank` object and plot its magnitude response.

```

ofb = octaveFilterBank("1/3 octave",Fs,FilterOrder=N);
freqz(ofb,N=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")

```



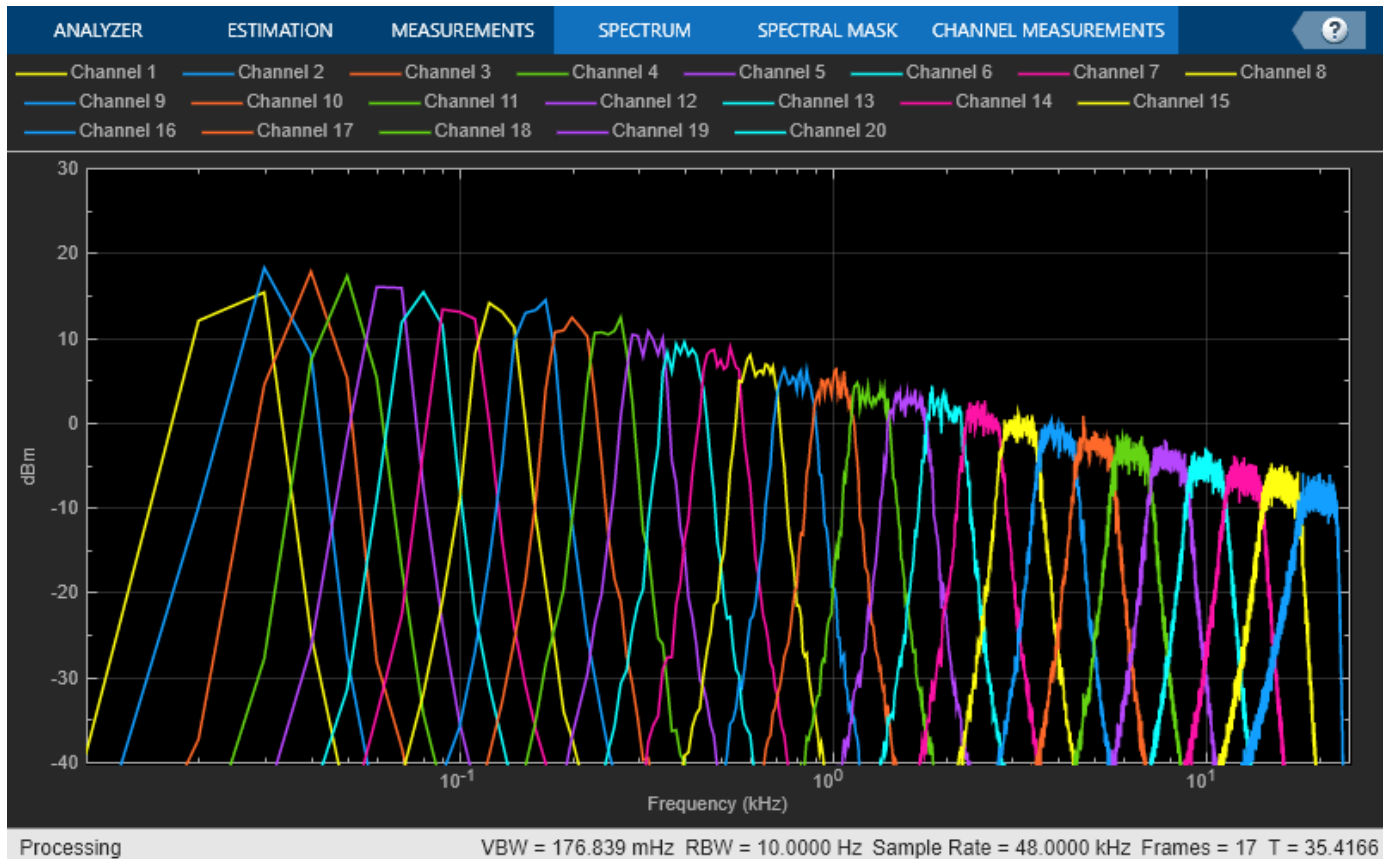
Filter the output of a pink noise generator with the 1/3-octave filter bank and compute the total power at the output of each filter.

```
pinkNoise = dsp.ColoredNoise(Color="pink", ...
                             SamplesPerFrame=Nx, ...
                             NumChannels=1);

scope2 = 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
```

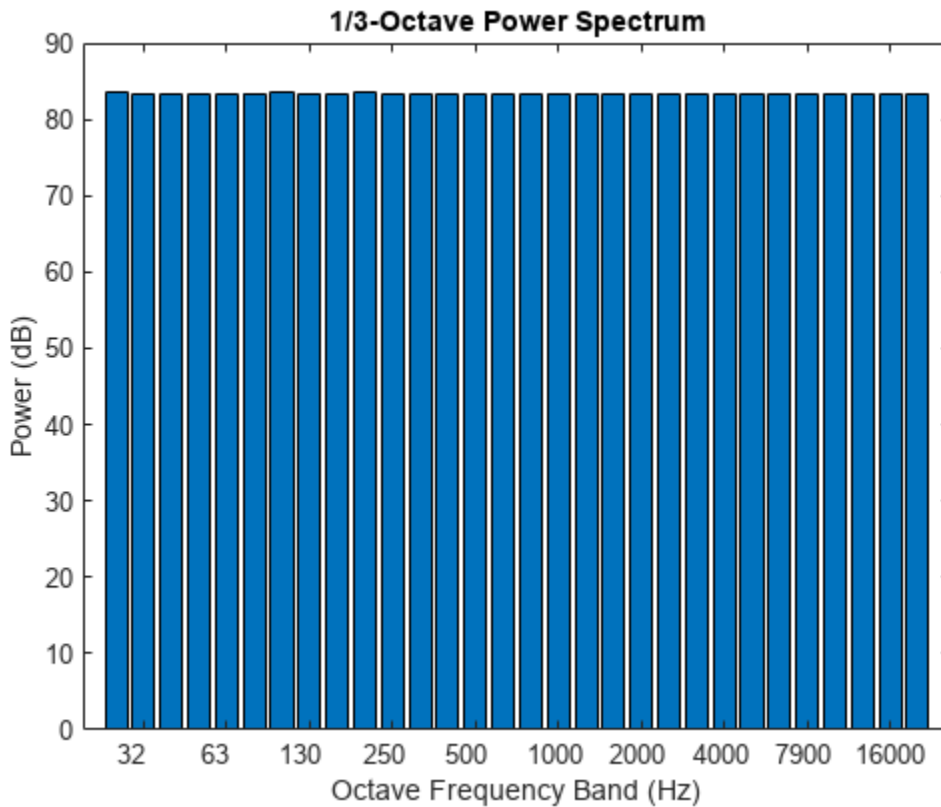


Pink noise has the same total power in each octave band, so the power between 5 Hz and 10 Hz is the same as between 5,000 Hz and 10,000 Hz. Consequently, in the spectrum analyzer, you can observe the 10 dB/decade fall-off that is characteristic of pink noise on a log-log scale, and how that signal is 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)")
```



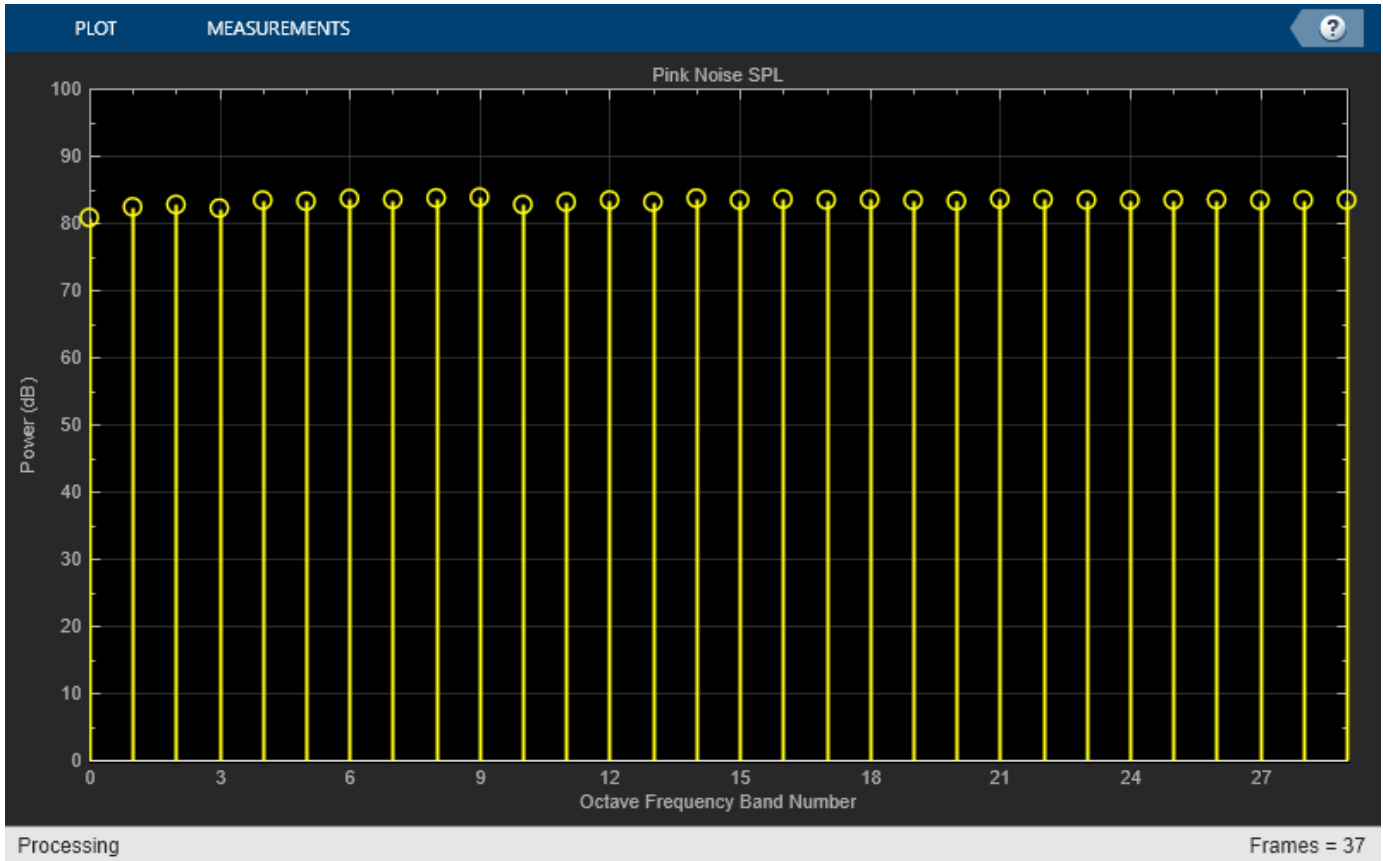
Octave SPL

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;
```



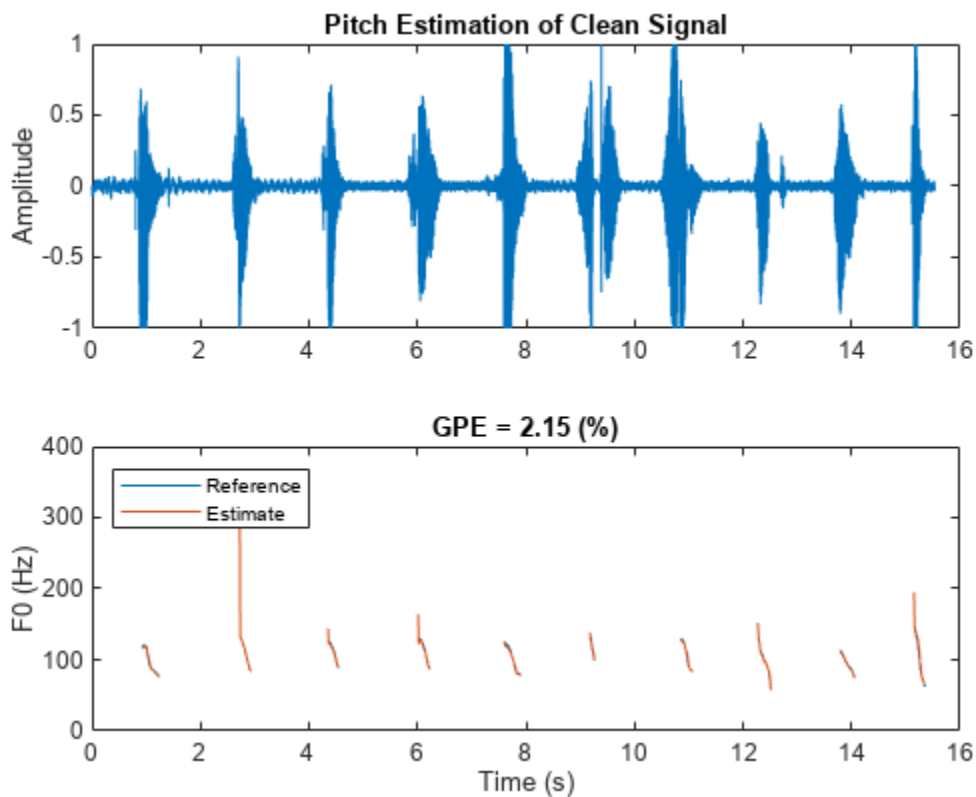
```

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="northwest")
ylabel("F0 (Hz)")
xlabel("Time (s)")
title("GPE = " + round(GPE,2) + " (%)")

```



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;

```

```

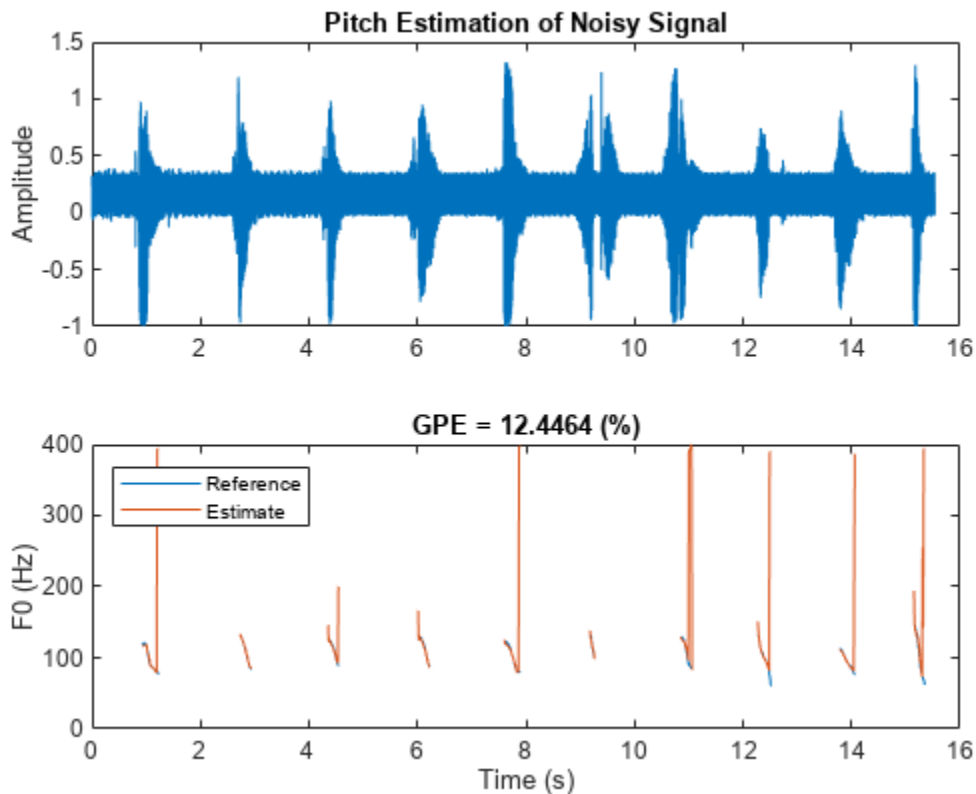
sound(x, fs)

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="northwest")
ylabel("F0 (Hz)")
xlabel("Time (s)")
title("GPE = " + 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.

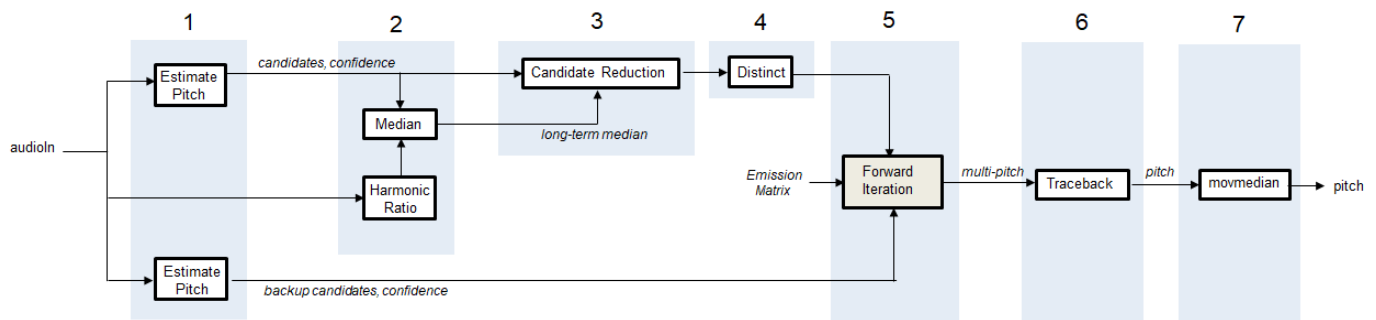
`[f0,loc] = HelperPitchTracker(...)` returns the locations associated with each pitch decision. The locations correspond to the ceiling of the center of the analysis frames.

`[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 1-428] algorithm and the Pitch Estimation Filter with Amplitude Compression (PEFAC) [3 on page 1-428] 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.

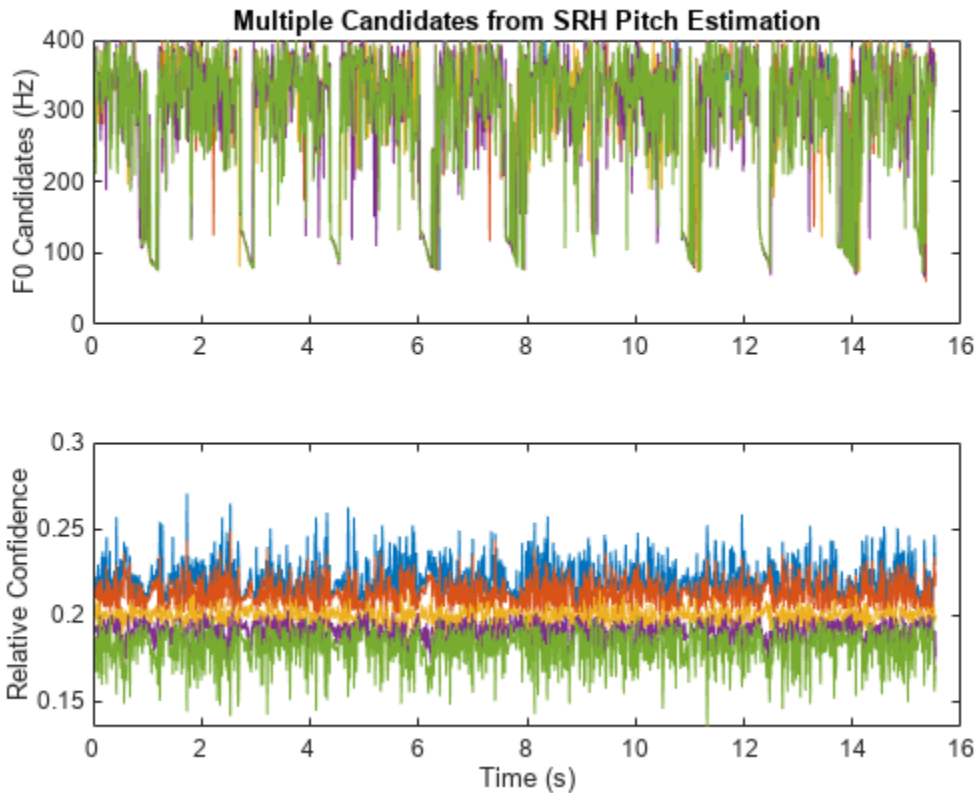
```
RANGE = [50,400];
HOPLength = round(fs.*0.01);

% 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);

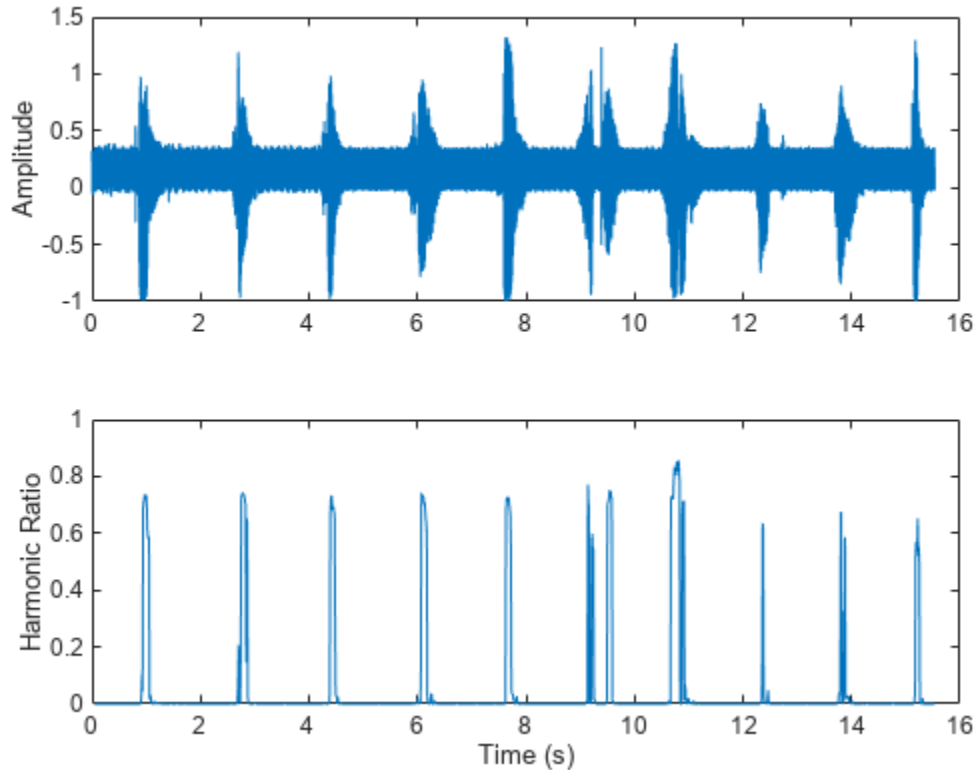
BackupCandidates = [f0_NCF(:,1),f0_CEP(:,1)];
```

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);

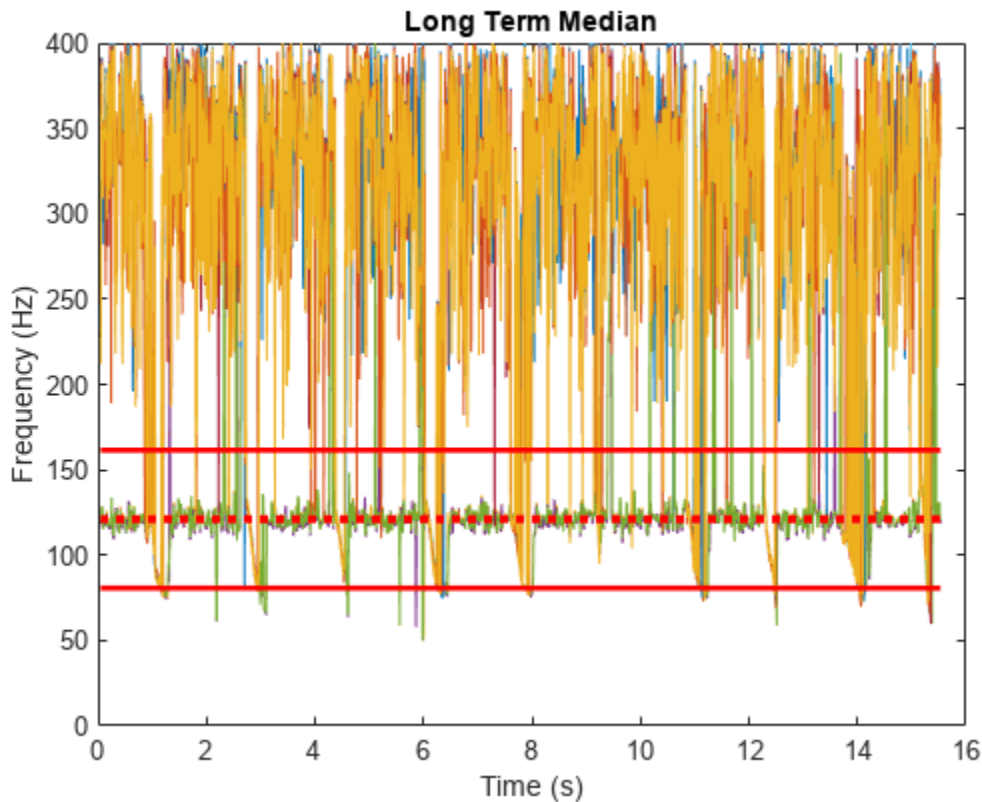
figure(4)
tiledlayout(2,1)
nexttile
plot(t,x)
ylabel("Amplitude")
nexttile
plot(t0,hr)
ylabel("Harmonic Ratio")
xlabel("Time (s)")
```



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

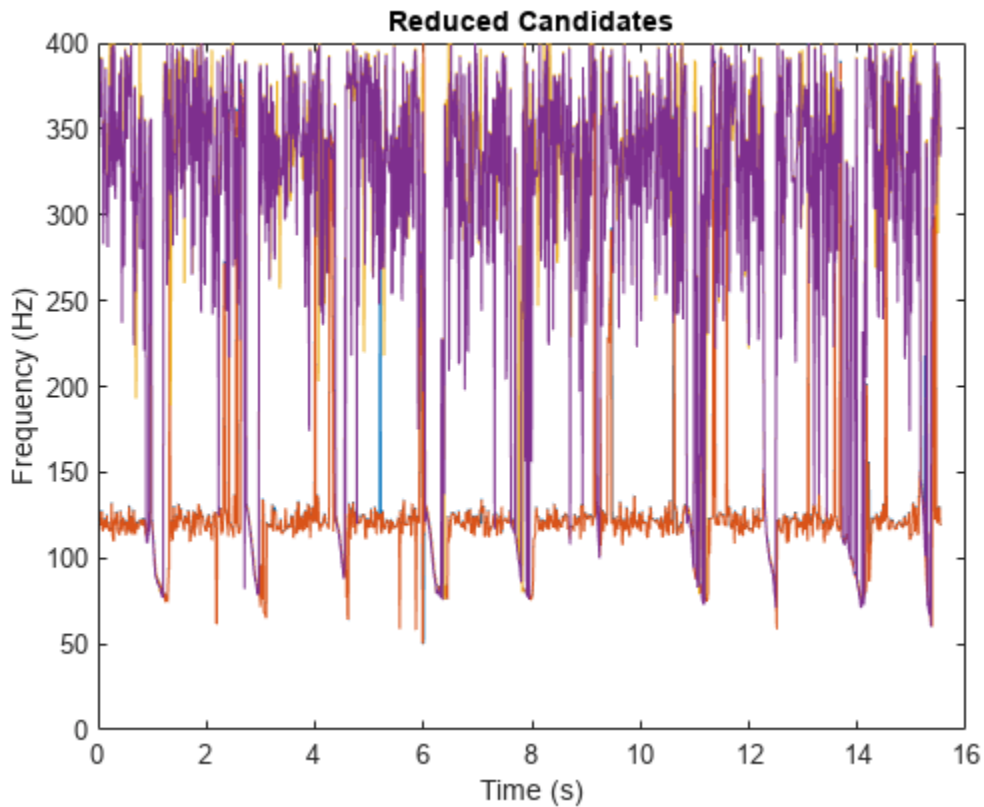
candidates = [f0_PEF(:,1:2),f0_SRH(:,1:2)];
confidence = [conf_PEF(:,1:2),conf_SRH(:,1:2)];
```



```

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*

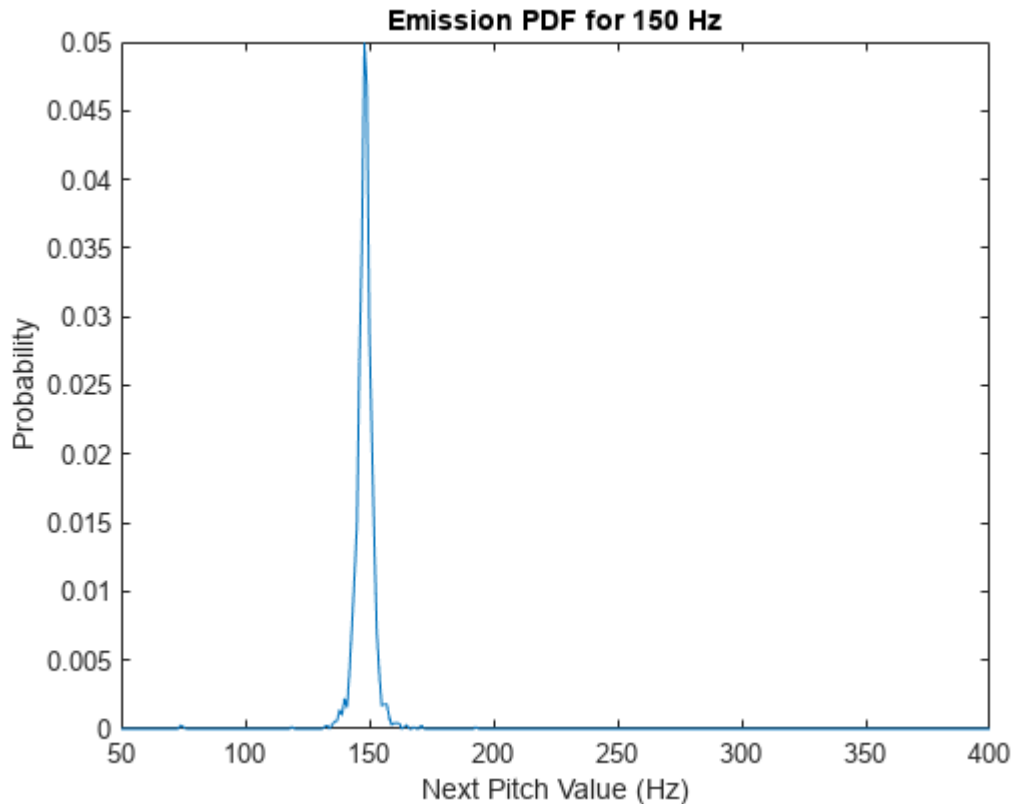
probability. Emission probabilities can be encapsulated into a matrix which describes the probability of going from any pitch value in a set range to any other in a set range. The emission matrix used in this example was created using the Graz database. [1 on page 1-428]

Load the emission matrix and associated range. Plot the probability density function (PDF) of a pitch in 150 Hz state.

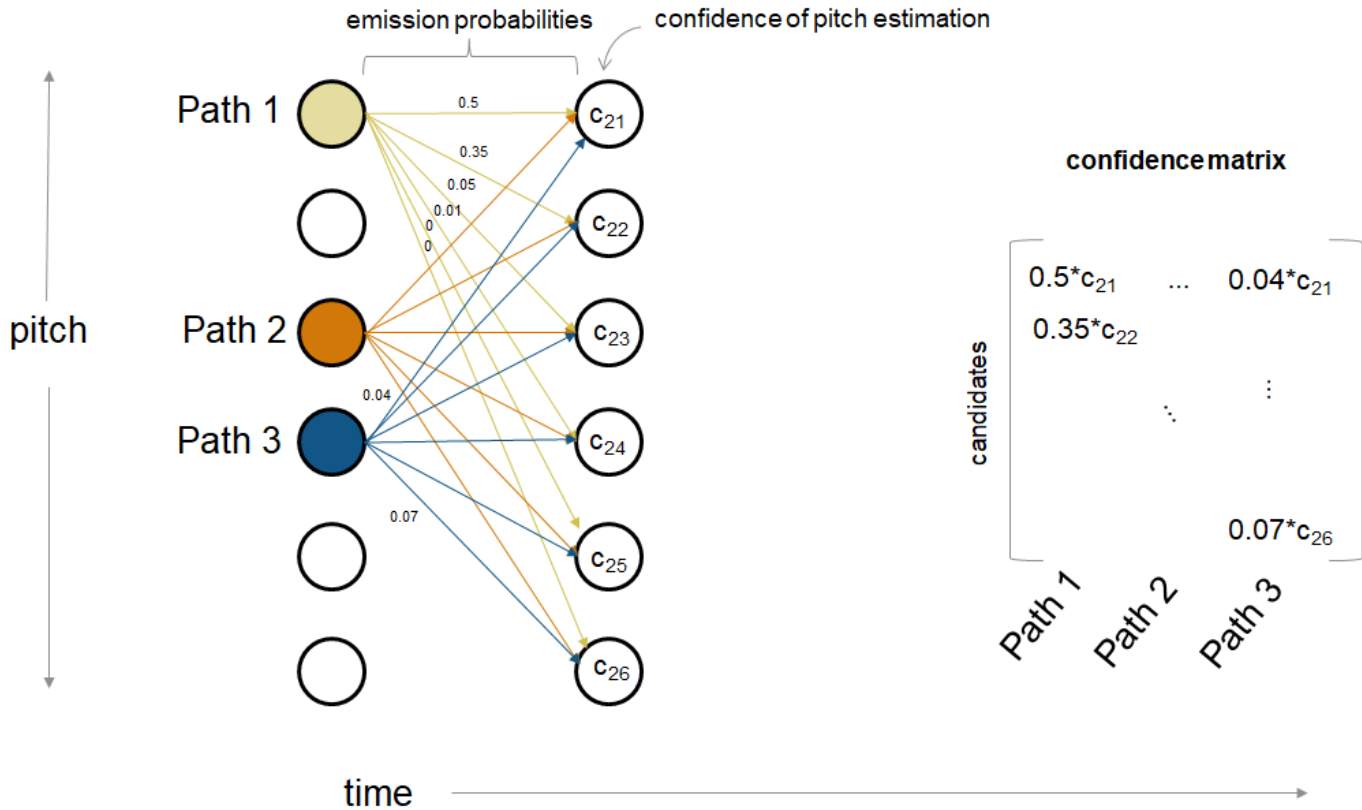
```
load EmissionMatrix.mat emissionMatrix emissionMatrixRange
```

```
currentState = 150  ;
```

```
figure(7)
plot(emissionMatrixRange(1):emissionMatrixRange(2),emissionMatrix(currentState - emissionMatrixR
title("Emission PDF for " + currentState + " Hz")
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

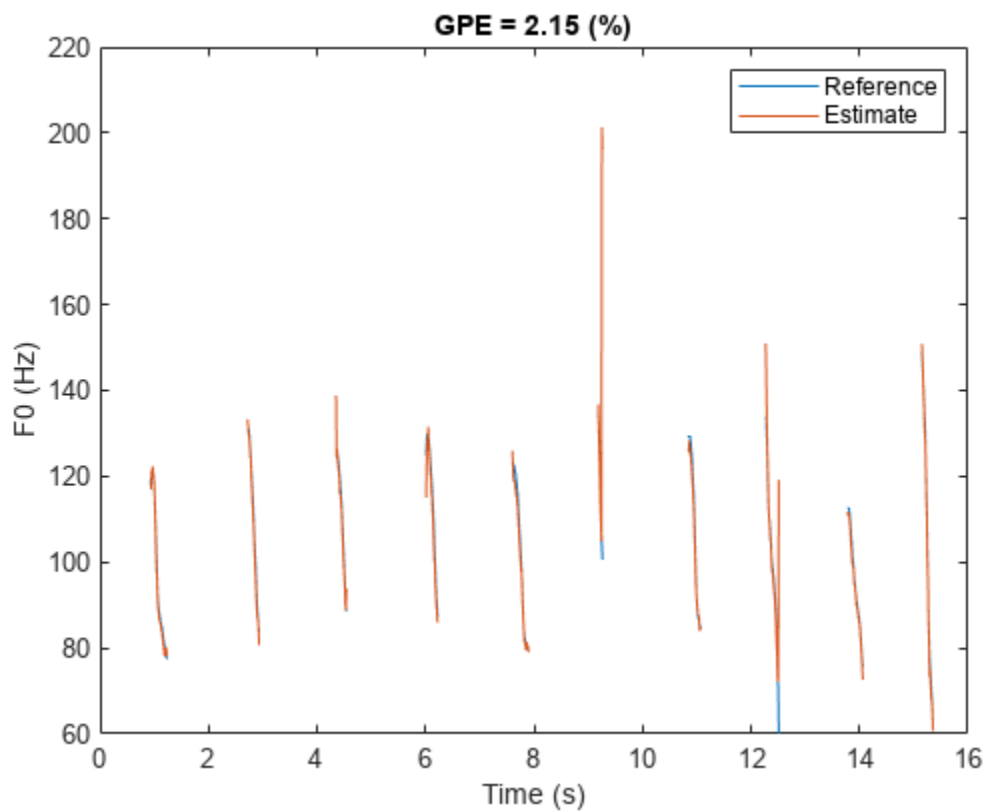
[~,bestPathIdx] = max(logbook(index,2,:));
bestIndices = zeros(numHops,1);
bestIndices(index) = bestPathIdx;

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("GPE = " + round(GPE,2) + " (%)")

```

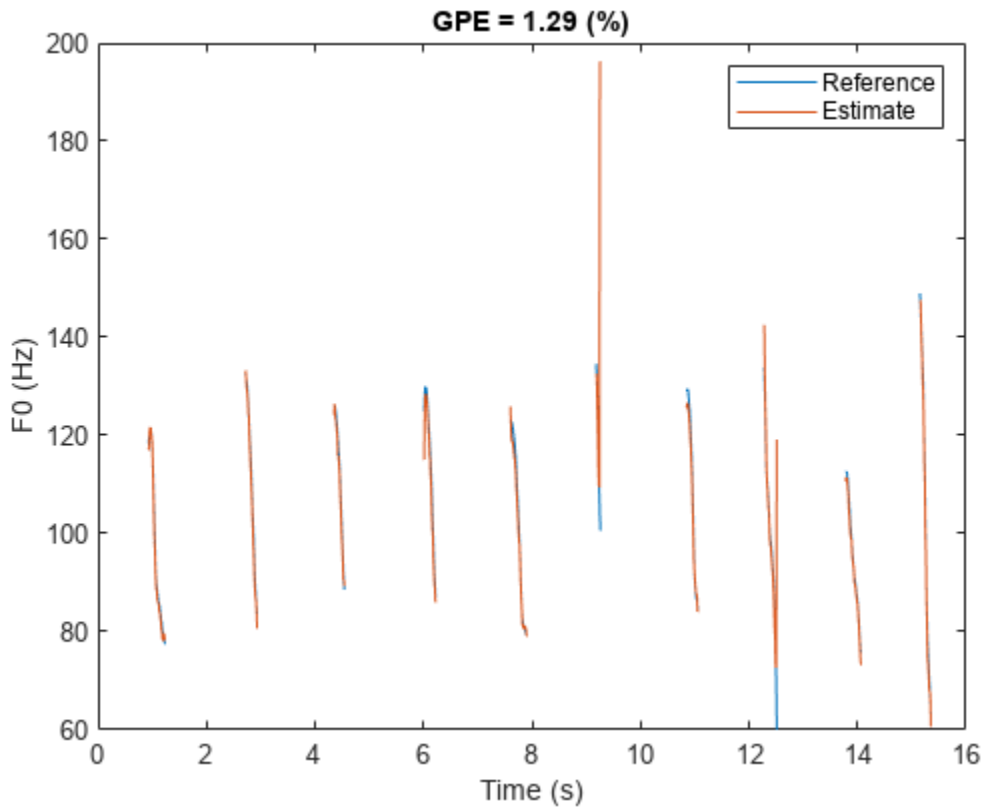


7. Moving Median Filter

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("GPE = " + round(GPE,2) + " (%)")
```



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 1-428. The data set consists of 20 English native speakers reading 2342 phonetically rich sentences from the TIMIT corpus. Download and extract the data set.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "ptdb-tug.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
dataset = fullfile(dataFolder, "ptdb-tug");
```

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(dataset, "SPEECH DATA", "FEMALE", "MIC"), fullfile(dataset, "SPEECH DATA", "MALE", "MIC")], ...
    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="mt19937ar w
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 1-428]. 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,EmissionPrior=emissionPrior);
            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

```

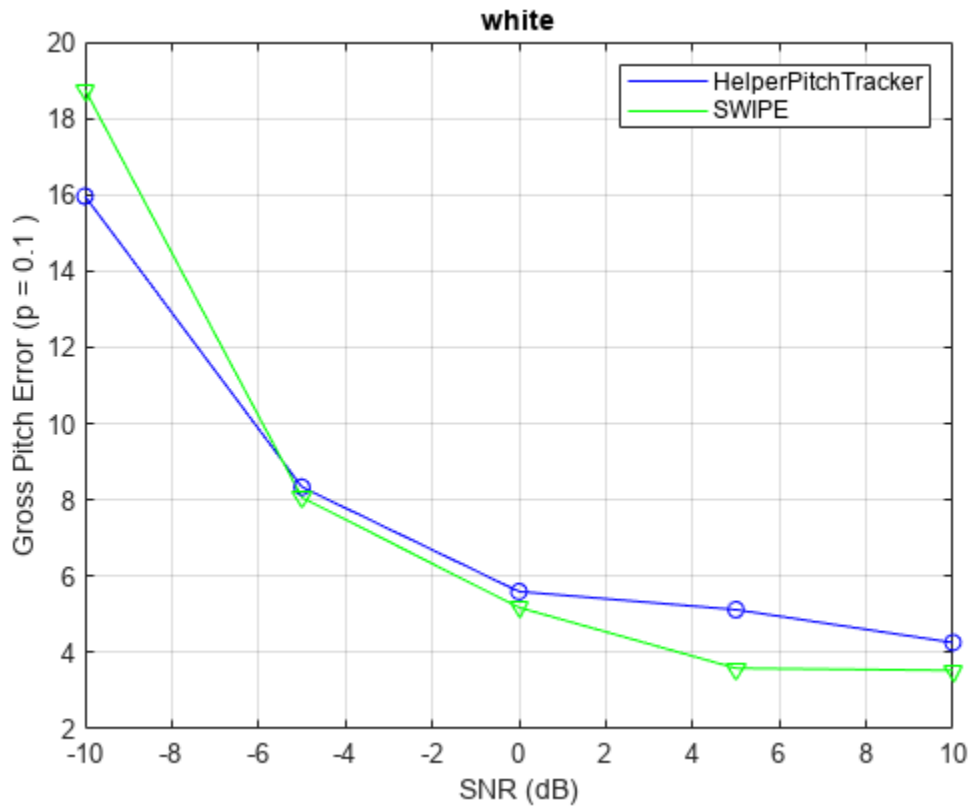
Plot the gross pitch error for each noise type.

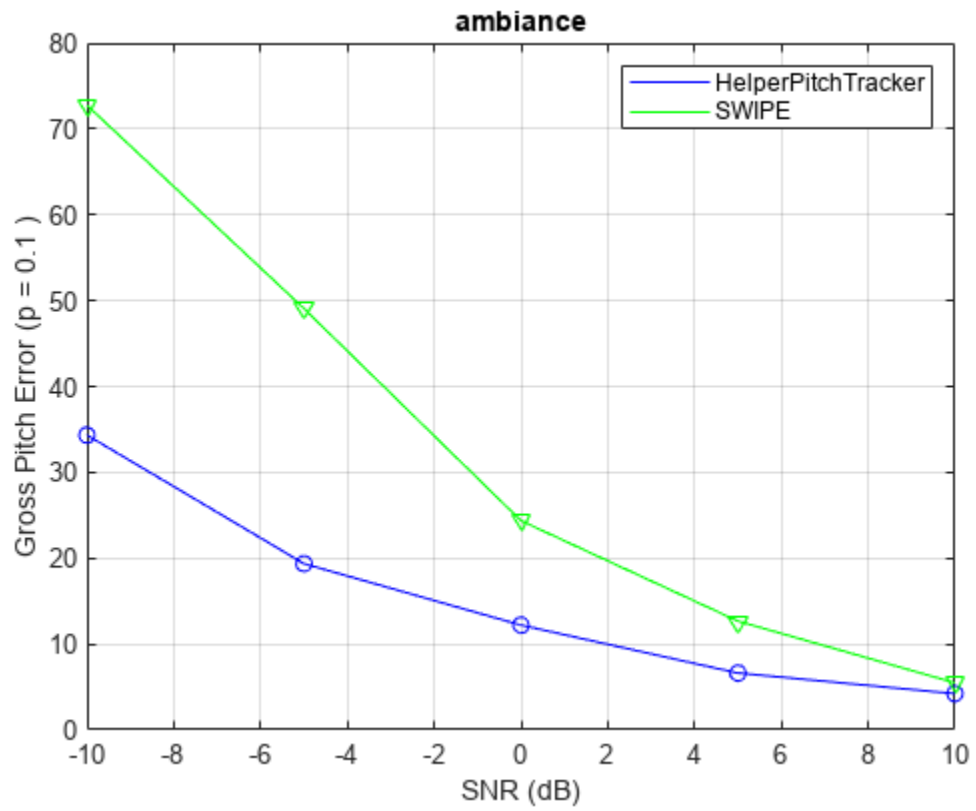
```

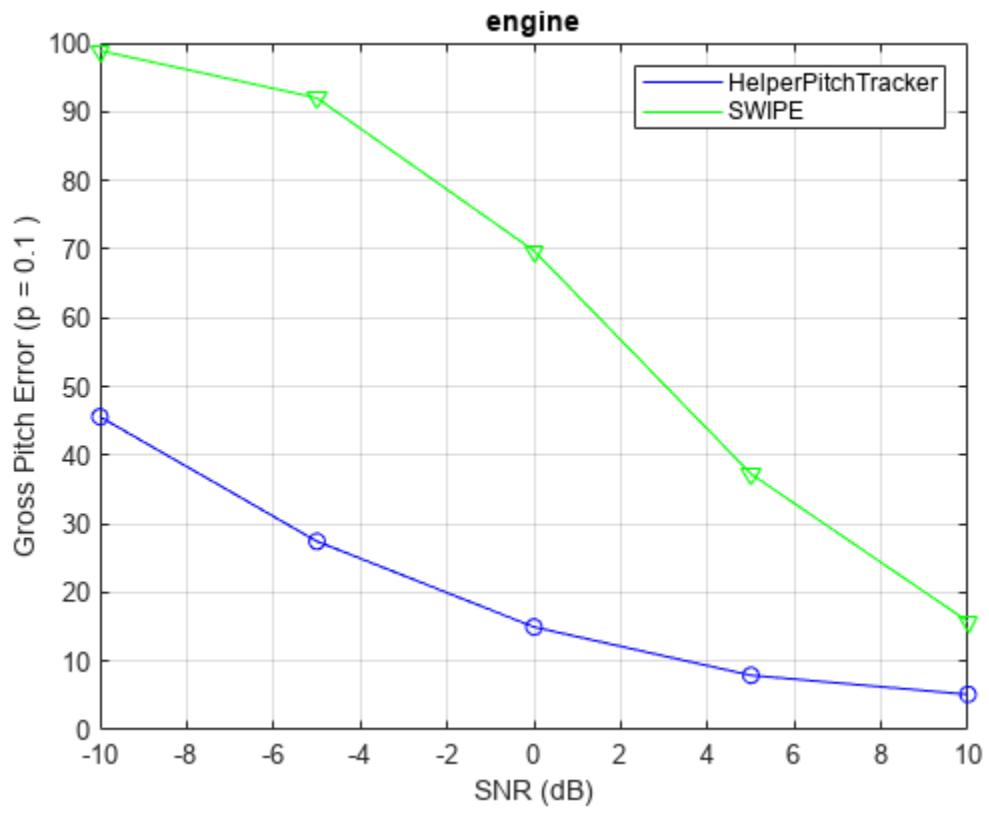
for ii = 1:numel(noise)
    figure(9+ii)
    plot(snrToTest,GPE_pitchTracker(:,ii),"b")
    hold on
    if compare
        plot(snrToTest,GPE_swipe(:,ii),"g")
    end
    plot(snrToTest,GPE_pitchTracker(:,ii),"bo")
    if compare
        plot(snrToTest,GPE_swipe(:,ii),"gv")
    end
    title(noiseType(ii))
    xlabel("SNR (dB)")
    ylabel("Gross Pitch Error (p = " + round(p,2) + " )")
    if compare
        legend("HelperPitchTracker","SWIPE")
    else
        legend("HelperPitchTracker")
    end
end

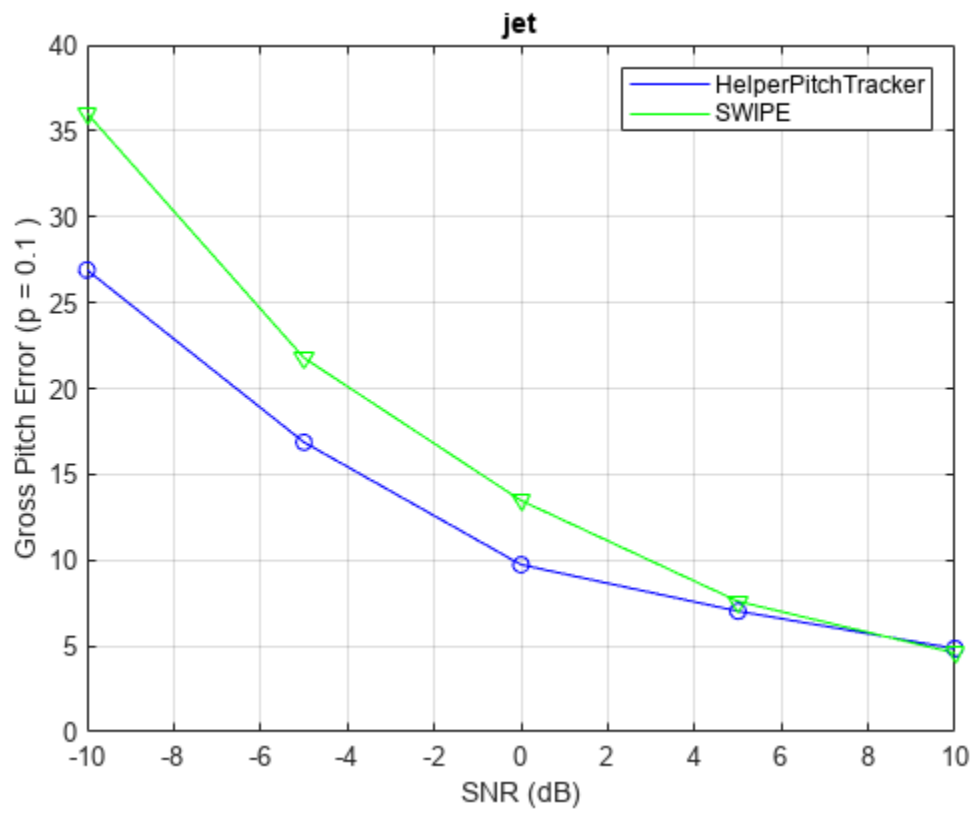
```

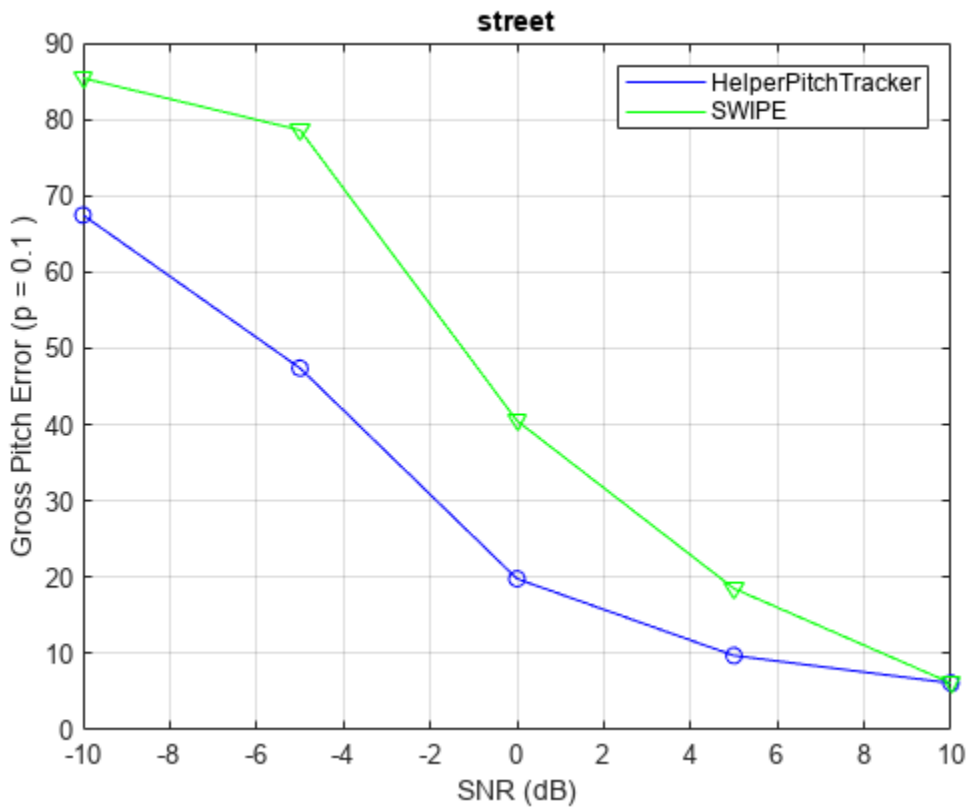
```
grid on  
hold off  
end
```











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>.

Train Voice Activity Detection in Noise Model Using Deep Learning

This example shows how to detect regions of speech in a low signal-to-noise environment using deep learning. You train a bidirectional long short-term memory (BiLSTM) network from scratch to perform voice activity detection (VAD) and compare that network to a pretrained deep learning-based VAD. To explore the model trained from scratch in this example, see “Voice Activity Detection in Noise Using Deep Learning” on page 1-449. To use an off-the-shelf deep learning-based VAD, see `detectspeechnn`.

Introduction

Voice activity detection is an essential component of many audio systems, such as automatic speech recognition, speaker recognition, and audio conferencing. Voice activity detection can be especially challenging in low signal-to-noise (SNR) situations, where speech is obstructed by noise.

For reproducibility, set the random seed to default.

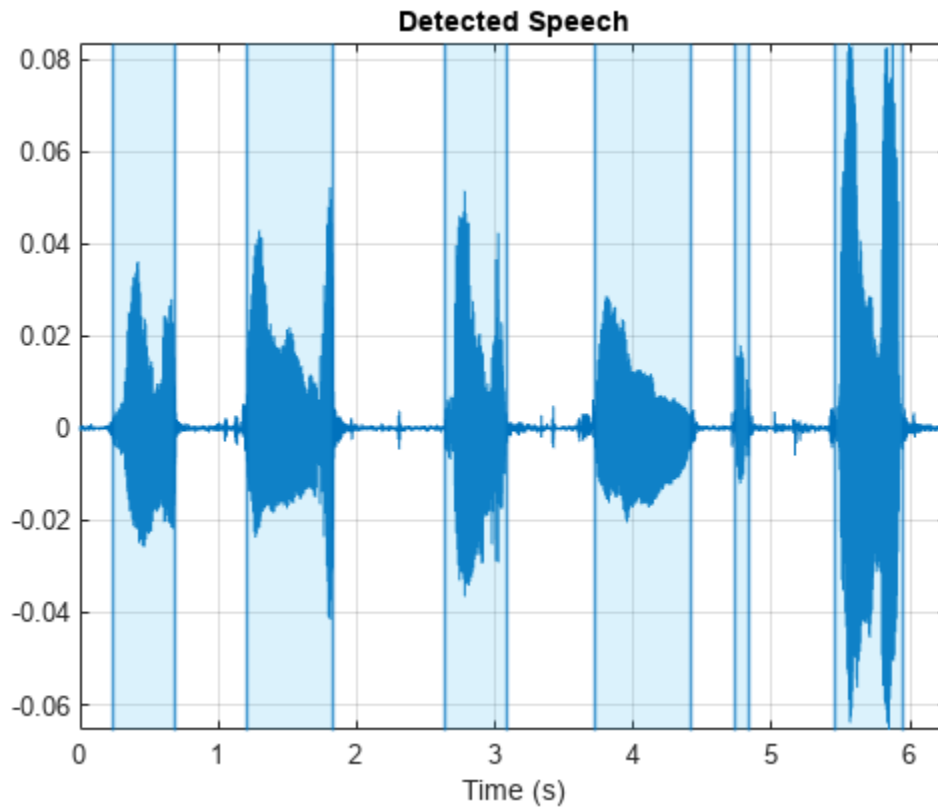
```
rng default
```

In high SNR scenarios, traditional speech detection algorithms perform adequately. Read in an audio file that consists of words spoken with pauses between and listen to it.

```
fs = 16e3;  
[speech,fileFs] = audioread("MaleVolumeUp-16-mono-6secs.ogg");  
sound(speech,fs)
```

Use the `detectSpeech` function to locate regions of speech. The `detectSpeech` function correctly identifies all regions of speech.

```
detectSpeech(speech,fs)
```

Load two noise signals and resample to the audio sample rate.

```
[noise200,fileFs200] = audioread("WashingMachine-16-8-mono-200secs.mp3");
[noise1000,fileFs1000] = audioread("WashingMachine-16-8-mono-1000secs.mp3");
noise200 = resample(noise200,fs,fileFs200);
noise1000 = resample(noise1000,fs,fileFs1000);
```

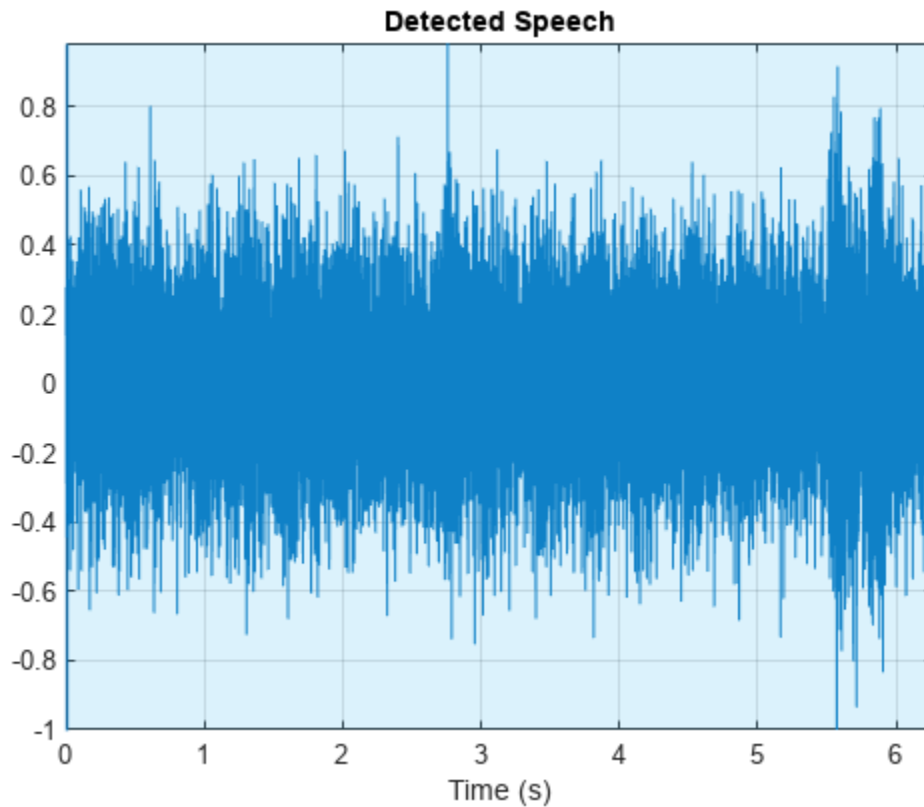
Use the supporting function `mixSNR` on page 1-447 to corrupt the clean speech signal with washing machine noise at a desired SNR level in dB. Listen to the corrupted audio.

```
SNR = -10  ;
noisySpeech = mixSNR(speech,noise200,SNR);

sound(noisySpeech,fs)
```

Call `detectSpeech` on the noisy speech signal. The function fails to detect the speech regions given the very low SNR. The remainder of the example walks through training and evaluating deep learning-based VAD networks that can perform well under low SNR.

```
detectSpeech(noisySpeech,fs)
```



Download and Prepare Data

Download and extract the Google Speech Commands Dataset [1] on page 1-448.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "google_speech.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
dataset = fullfile(dataFolder, "google_speech");
```

Create `audioDatastore` objects to point to the training and validation data sets.

```
adsTrain = audioDatastore(fullfile(dataset, "train"), IncludeSubfolders=true);
adsValidation = audioDatastore(fullfile(dataset, "validation"), IncludeSubfolders=true);
```

Construct Train and Validation Signals

The Google dataset consists of isolated words. Use the supporting function, `constructSignal` on page 1-447, to construct train and validation signals that consist of isolated words and regions of silence. The `constructSignal` function also returns ground truth binary masks indicating the regions of speech in the train and validation signals.

```
[audioTrain, TTrainPerSample] = constructSignal(adsTrain, fs, 1000);
[audioValidation, TValidationPerSample] = constructSignal(adsValidation, fs, 200);
```

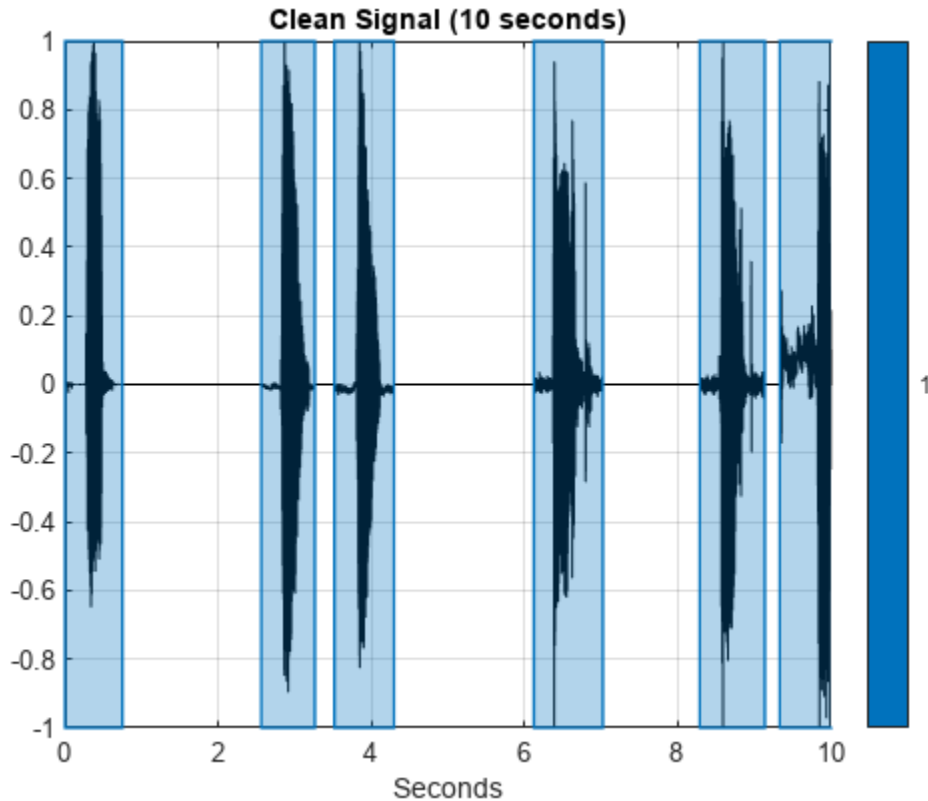
Listen to the first 10 seconds of the constructed signal. Use `signalMask` and `plotsigroi` to visualize the signal and ground truth binary mask.

```

duration = 10 ;
sound(audioTrain(1:duration*fs), fs)

mask = signalMask(TTrainPerSample, SampleRate=fs);
plotsigroi(mask, audioTrain, true)
xlim([0, duration])
title("Clean Signal (" + duration + " seconds)")

```



Add Noise to Train and Validation Signals

Use the supporting function `mixSNR` on page 1-447 to corrupt the train and validation signals with noise.

```

audioTrain = mixSNR(audioTrain, noise1000, SNR);
audioValidation = mixSNR(audioValidation, noise200, SNR);

```

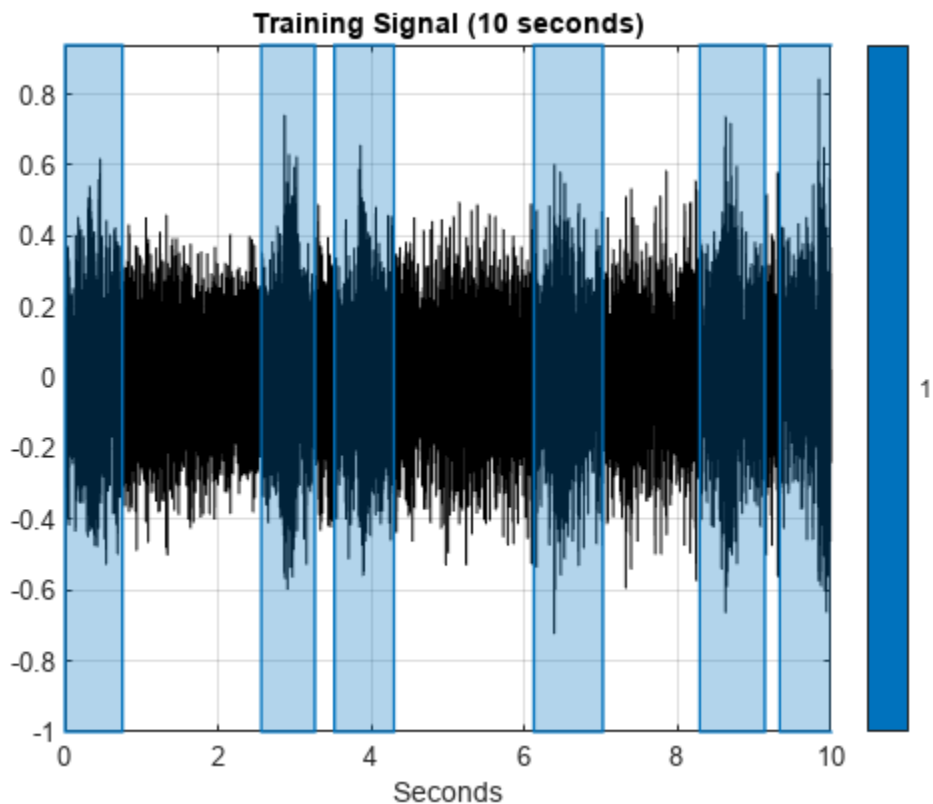
Listen to the first 10 seconds of the train signal and visualize the signal and mask.

```

sound(audioTrain(1:duration*fs), fs)

plotsigroi(mask, audioTrain, true)
xlim([0, duration])
title("Training Signal (" + duration + " seconds)")

```



Input Pipeline

Define an `audioFeatureExtractor` to extract the following spectral features: `spectralCentroid`, `spectralCrest`, `spectralEntropy`, `spectralFlux`, `spectralKurtosis`, `spectralRolloffPoint`, `spectralSkewness`, `spectralSlope`, and the periodicity feature `harmonicRatio`. Extract features using 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);
```

```
featuresTrain = extract(afe,audioTrain);
```

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 signal length, window length, and overlap length). The second dimension is the number of features used in this example.

```
[numWindows,numFeatures] = size(featuresTrain)
```

```
numWindows = 124999
```

```
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(featuresTrain,1);
S = std(featuresTrain,[],1);
featuresTrain = (featuresTrain - M) ./ S;
```

Extract features from the validation signal using the same process.

```
XValidation = extract(afe, audioValidation);
XValidation = (XValidation - mean(XValidation,1)) ./ std(XValidation,[],1);
```

Each feature corresponds to 256 samples of data (the window length), sampled every 128 samples (the hop length). For each window, set the expected voice/no voice value to the mode of the baseline mask values corresponding to those 256 samples. Convert the voice/no voice mask to categorical.

```
windowLength = numel(afe.Window);
overlapLength = afe.OverlapLength;
```

```
TTrain = mode(buffer(TTrainPerSample,windowLength,overlapLength,"nodelay"),1);
```


```
TTrain = categorical(TTrain);
```

Do the same for the validation mask.

```
TValidation = mode(buffer(TValidationPerSample,windowLength,overlapLength,"nodelay"),1);
```

```
TValidation = categorical(TValidation);
```

Use the supporting function `featureBuffer` on page 1-446 to split the training features and the mask into sequences with a duration approximately 8 seconds and a 75% overlap between consecutive sequences.

```
sequenceDuration = 8  ;
analysisHopLength = numel(afe.Window) - afe.OverlapLength;
sequenceLength = round(sequenceDuration*fs/analysisHopLength);
```

```
overlapPercent = 0.75  ;
```

```
XTrain = featureBuffer(featuresTrain',sequenceLength,overlapPercent);
TTrain = featureBuffer(TTrain,sequenceLength,overlapPercent);
```

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.

```
layers = [ ...
    sequenceInputLayer(afe.FeatureVectorLength)
```

```
bilstmLayer(200,OutputMode="sequence")
bilstmLayer(200,OutputMode="sequence")
fullyConnectedLayer(2)
softmaxLayer
classificationLayer
];
```

Training Options

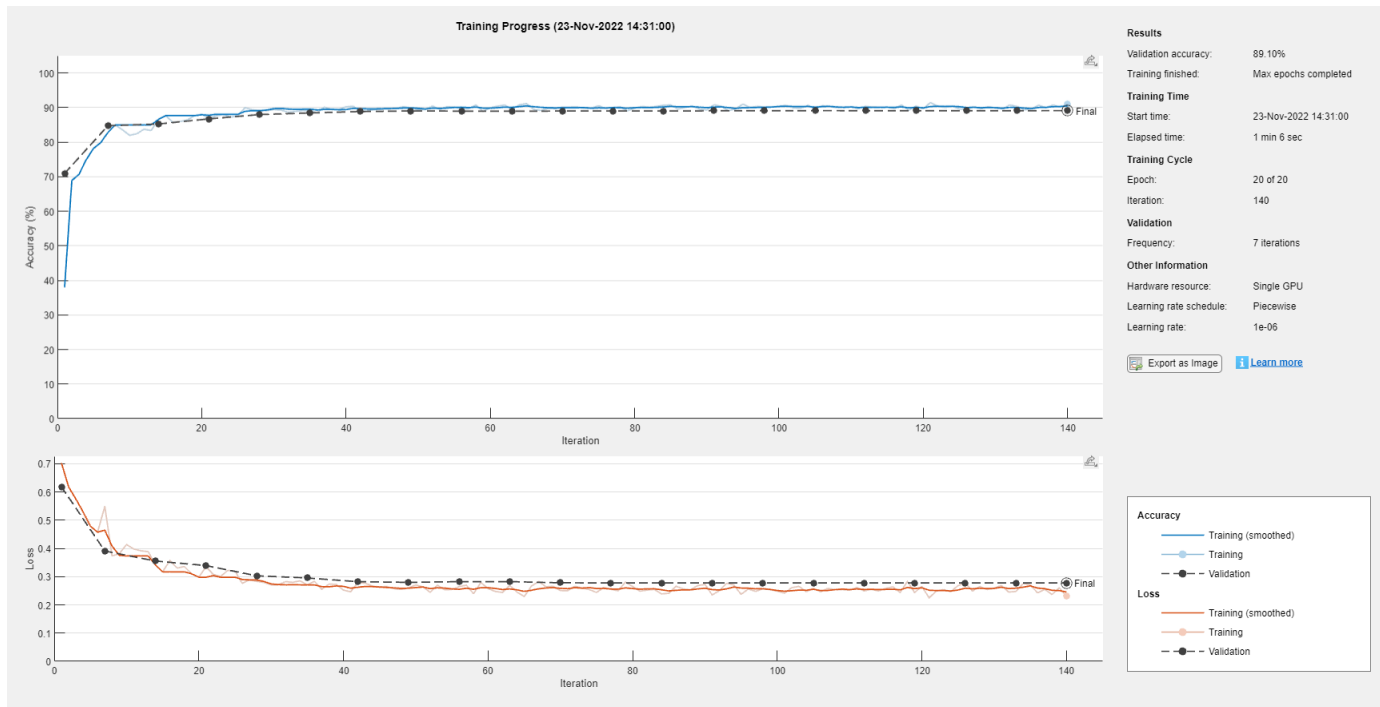
To define parameters for training, use `trainingOptions` (Deep Learning Toolbox). Use the Adam optimizer with a mini-batch size of 64 and a piecewise learn rate schedule.

```
maxEpochs = 20  ;
miniBatchSize = 64  ;
options = trainingOptions("adam", ...
    MaxEpochs=maxEpochs, ...
    MiniBatchSize=miniBatchSize, ...
    Shuffle="every-epoch", ...
    Verbose=false, ...
    SequenceLength=sequenceLength, ...
    ValidationFrequency=floor(numel(XTrain)/miniBatchSize), ...
    ValidationData={XValidation.',TValidation}, ...
    Plots="training-progress", ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropFactor=0.1, ...
    LearnRateDropPeriod=5, ...
    OutputNetwork="best-validation-loss");
```

Train Network

To train the network, use `trainNetwork` (Deep Learning Toolbox).

```
speechDetectNet = trainNetwork(XTrain,TTrain,layers,options);
```



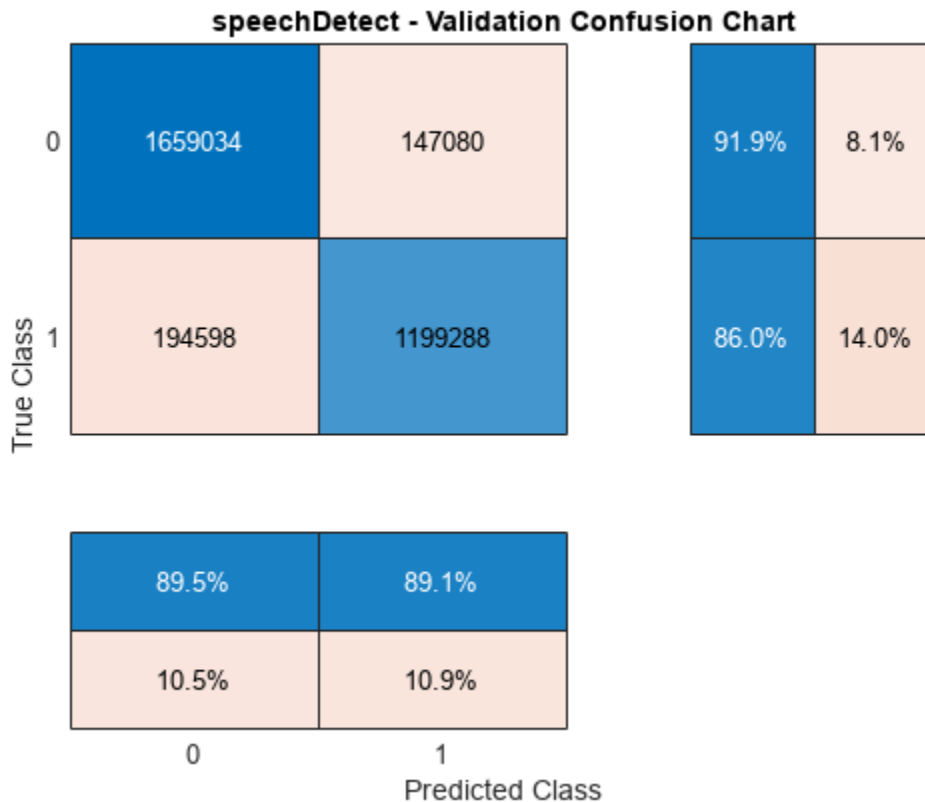
Evaluate Trained Network

Estimate voice activity in the validation signal using the trained network. Convert the estimated VAD mask from categorical to double, then replicate the window-based decisions to sample-based decisions.

```
YValidation = classify(speechDetectNet,XValidation. ');
YValidation = double(YValidation).' - 1;
wL = numel(afe.Window);
hL = wL - afe.OverlapLength;
YValidationPerSample = [repelem(YValidation(1),floor(wL/2 + hL/2),1);
    repelem(YValidation(2:end-1),hL,1);
    repelem(YValidation(end),ceil(wL/2 + hL/2),1)];
```

Calculate and plot the validation confusion matrix from the vectors of actual and estimated labels. Save the results for later analysis.

```
cc = confusionchart(TValidationPerSample,YValidationPerSample, ...
    title="speechDetect - Validation Confusion Chart", ...
    ColumnSummary="column-normalized",RowSummary="row-normalized");
```



```
speechDetectResults = cc.NormalizedValues;
```

Evaluate Pretrained VAD Network

The `vadnet` network is a pretrained network for voice activity detection. You can use it with the `vadnetPreprocess` and `vadnetPostprocess` functions for applications such as transfer learning, or you can use `detectspeechnn`, which encapsulates `vadnetPreprocess`, `vadnet`, and `vadnetPostprocess` for inference-only applications. The `vadnet` network performs well under every-day adverse conditions, however it fails in the cases of extreme SNR, such as the -10 dB SNR used in this example. Also, `vadnet` was trained to detect regions of continuous speech (meaning several words in a row), not isolated words. In short, the pretrained `vadnet` fails for the validation signal in this example.

Load in the pretrained `vadnet` model.

```
net = vadnet();
```

Extract features from the validation signal using the same input pipeline used to train the network.

```
XValidation = vadnetPreprocess(audioValidation, fs);
```

Predict the VAD mask.

```
y = predict(net, XValidation);
```

`vadnet` is a regression network and requires additional post-processing to determine decision boundaries. Use `vadnetPostprocess` to determine the boundaries of voice activity regions.

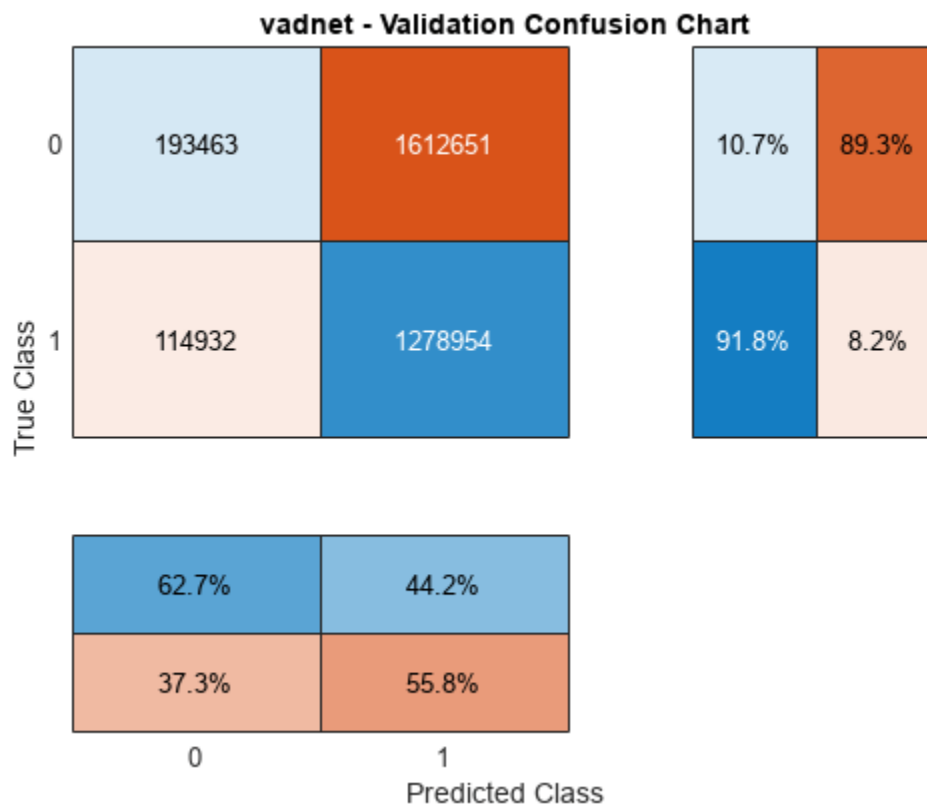

```
boundaries = vadnetPostprocess(audioValidation,16e3,y);
```

The `vadnetPostprocess` function returns the decisions as time boundaries. To convert the boundaries to a binary mask that corresponds to the original signal samples, use `sigroi2binmask`.

```
YValidationPerSample = double(sigroi2binmask(boundaries,size(audioValidation,1)));
```

To create a confusion chart to analyze the error, use `confusionchart` (Deep Learning Toolbox).

```
confusionchart(TValidationPerSample,YValidationPerSample, ...
    title="vadnet - Validation Confusion Chart", ...
    ColumnSummary="column-normalized",RowSummary="row-normalized");
```



Transfer Learning

Apply transfer learning to the pretrained `vadnet` to make use of both the pretrained weights and the network architecture.

To extract the layer graph from the pretrained network, use `layerGraph` (Deep Learning Toolbox).

```
layers = layerGraph(net);
```

Extract features from the audio.

```
featuresTrain = vadnetPreprocess(audioTrain,fs);
```

Buffer the ground truth mask so that decisions correspond to the analysis windows used in `vadnetPreprocess`.

```

windowLength = 400;
overlapLength = 240;
TTrainPerSamplePadded = [zeros(floor(windowLength/2),1);TTrainPerSample;zeros(ceil(windowLength/2)-TTrainPerSample,1)];
TTrain = mode(buffer(TTrainPerSamplePadded,windowLength,overlapLength,"nodelay"),1);

```

Buffer the validation mask.


```

TValidationPerSamplePadded = [zeros(floor(windowLength/2),1);TValidationPerSample;zeros(ceil(windowLength/2)-TValidationPerSample,1)];
TValidation = mode(buffer(TValidationPerSamplePadded,windowLength,overlapLength,"nodelay"),1);

```


Split the long training signal into overlapped sequences for training. Do the same for the ground-truth mask.

```

sequenceDuration = 8  ;
analysisHopLength = windowLength - overlapLength;
sequenceLength = round(sequenceDuration*fs/analysisHopLength);

```

```

overlapPercent = 0.75  ;

```



```

XTrain = featureBuffer(featuresTrain,sequenceLength,overlapPercent);
TTrain = featureBuffer(TTrain,sequenceLength,overlapPercent);

```

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

```

miniBatchSize = 12  ;
maxEpochs = 9  ;
options = trainingOptions("adam", ...
    InitialLearnRate=0.01, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropPeriod=3, ...
    MiniBatchSize=miniBatchSize, ...
    Shuffle="every-epoch", ...
    ValidationFrequency=floor(numel(XTrain)/miniBatchSize), ...
    ValidationData={XValidation,TValidation}, ...
    Verbose=false, ...
    Plots="training-progress", ...
    MaxEpochs=maxEpochs, ...
    OutputNetwork="best-validation-loss" ...
);

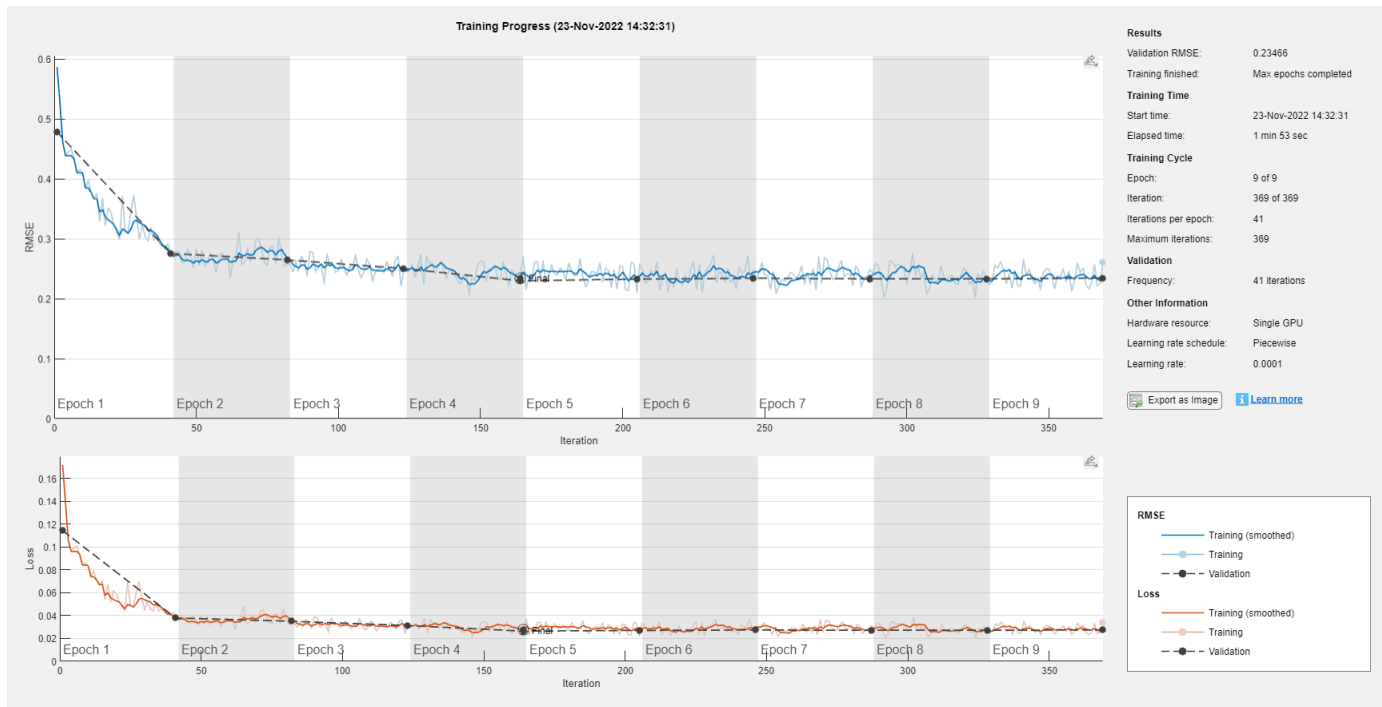
```

To train the network, use `trainNetwork` (Deep Learning Toolbox).

```

noisyvadnet = trainNetwork(XTrain,TTrain,layers,options);

```

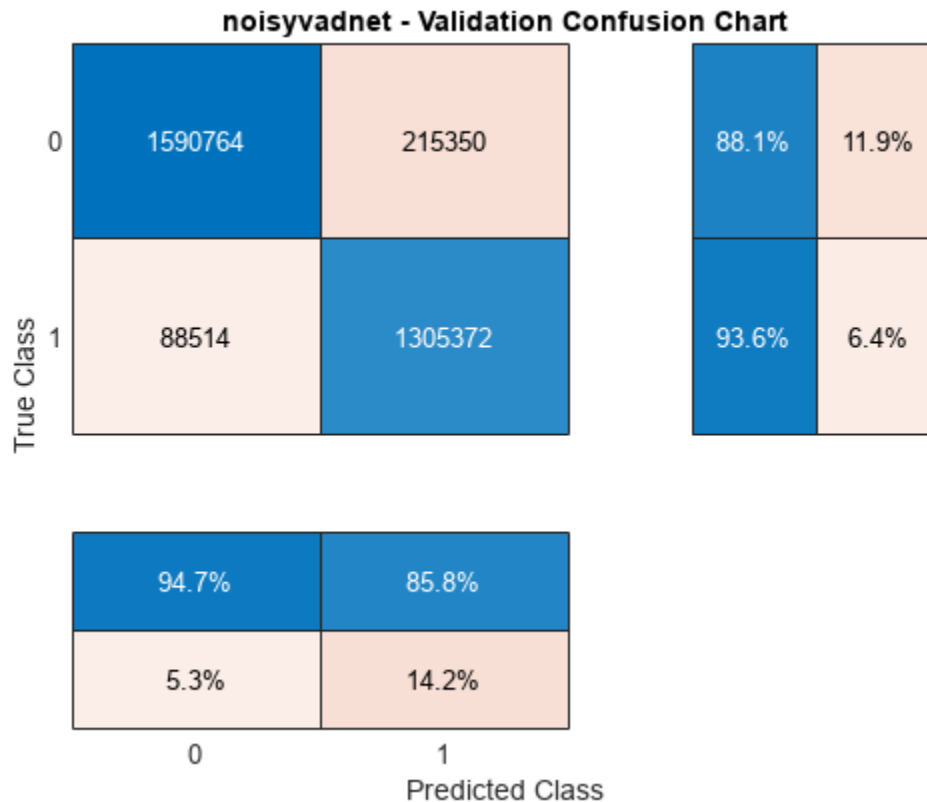


Estimate voice activity in the validation signal using the trained network. Postprocess the predictions using `vadnetPostprocess`, then convert the boundaries in time to a sample-based mask.

```
y = predict(noisyvadnet,XValidation);
boundaries = vadnetPostprocess(audioValidation,fs,y);
YValidationPerSample = double(sigroi2binmask(boundaries,size(audioValidation,1)));
```

Calculate and plot the validation confusion matrix from the vectors of actual and estimated labels. Save the results for later analysis.

```
cc = confusionchart(TValidationPerSample,YValidationPerSample, ...
    title="noisyvadnet - Validation Confusion Chart", ...
    ColumnSummary="column-normalized",RowSummary="row-normalized");
```



```
noisyvadnetResults = cc.NormalizedValues;
```

Compare Networks

There are several considerations when choosing a network, such as size, inference speed, error, and streaming capabilities.

Streaming

The `speechDetectNet` trained from scratch in this example is well-suited for streaming inference because its BiLSTM layers retain state between calls. See “Voice Activity Detection in Noise Using Deep Learning” on page 1-449 for an example of using `speechDetect` for streaming voice activity detection.

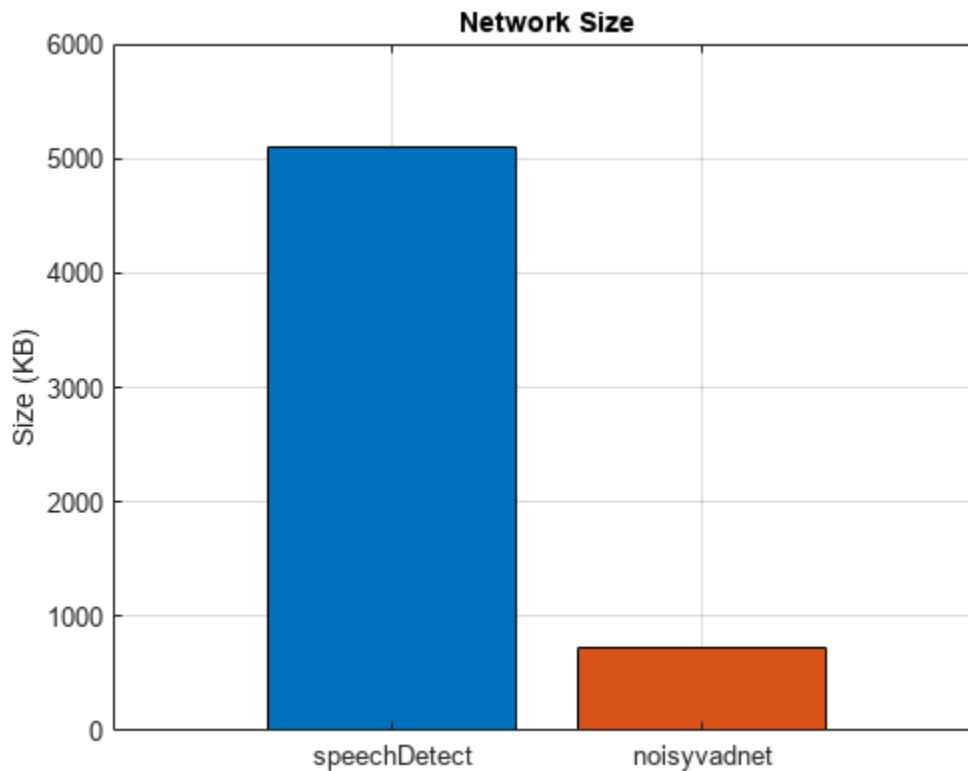
The `vadnet` architecture consists of convolutional, recurrent, and fully-connected layers, and is not well-suited for low-latency streaming. See the `vadnet` documentation for an example of streaming VAD detection using `vadnet`.

Network Size

Compare the network sizes.

```
networks = ["speechDetect", "noisyvadnet"];
b = bar(reordercats(categorical(networks), networks), [whos("speechDetectNet").bytes/1024, whos("noisyvadnet").bytes/1024]);
title("Network Size")
ylabel("Size (KB)")
grid on
```

```
b.FaceColor = "flat";
b.CData(2,:) = [0.8500 0.3250 0.0980];
```



Network Inference Speed

Compare the network inference speeds. The simple `speechDetect` architecture has faster inference speed on both the CPU and the GPU for short durations (approximately 8 second chunks or less). For longer durations, `speechDetect` is faster than `noisyvadnet` on the GPU and slower on the CPU.

```
durationsToTest = [1,5,10,20,40];
environment = ["CPU","GPU"];
```

```
speechDetectSpeed = zeros(numel(durationsToTest),numel(environment));
noisyvadnetSpeed = zeros(numel(durationsToTest),numel(environment));
```

```
for jj = 1:numel(environment)
    for ii = 1:numel(durationsToTest)
        idx = 1:durationsToTest(ii)*fs;
        speechDetectFeatures = extract(afe, audioValidation(idx))';
        vadnetFeatures = vadnetPreprocess(audioValidation(idx), fs);

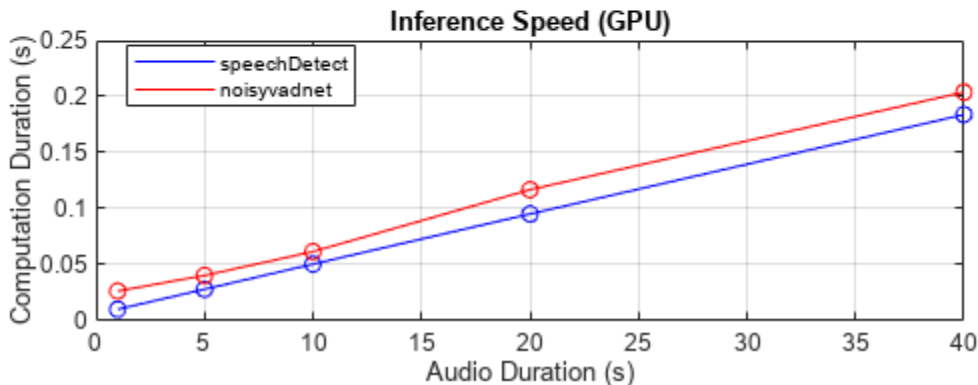
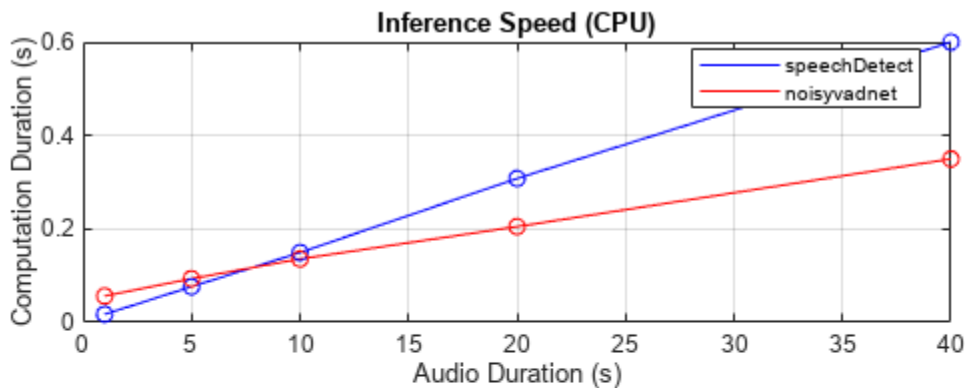
        switch environment(jj)
            case "CPU"
                speechDetectSpeed(ii,1) = timeit(@()classify(speechDetectNet,speechDetectFeatures));
                noisyvadnetSpeed(ii,1) = timeit(@()predict(noisyvadnet,vadnetFeatures,ExecutionEnvironment));
            case "GPU"
                speechDetectSpeed(ii,2) = gputimeit(@()classify(speechDetectNet,speechDetectFeatures));
                noisyvadnetSpeed(ii,2) = gputimeit(@()predict(noisyvadnet,vadnetFeatures,ExecutionEnvironment));
        end
    end
end
```

```

end
end

tiledlayout(2,1)
for ii = 1:numel(environment)
    nexttile
    plot(durationsToTest,speechDetectSpeed(:,ii),"b-", ...
        durationsToTest,noisyvadnetSpeed(:,ii),"r-", ...
        durationsToTest,speechDetectSpeed(:,ii),"bo", ...
        durationsToTest,noisyvadnetSpeed(:,ii),"ro")
    legend(["speechDetect","noisyvadnet"],Location="best")
    grid on
    xlabel("Audio Duration (s)")
    ylabel("Computation Duration (s)")
    title("Inference Speed (" + environment(ii) + ")")
end

```



Network Error

Use the previously calculated confusion charts to display common statistics for error analysis. Accuracy, recall, precision, and f1 score are all derived from the confusion matrices previously plotted.

Accuracy is defined as the ratio of correctly predicted observations to the total observations. It is the most intuitive metric but can be misleading for imbalanced data sets. For example, if speech is only present in 5% of the audio, then classifying all audio as non-speech would result in 95 % accuracy.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

Recall, also called sensitivity, is the ratio of correctly predicted positive observations to all observations that belong to the positive class. Recall answers the question: *Of all speech regions, how many were correctly classified?* A low recall indicates that regions of speech were misclassified as regions of nonspeech.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. Precision answers the question: *Of all the observations the network classified as speech, how many were actually speech?* A low precision indicates that regions of nonspeech were misclassified as regions of speech.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

F1 score is the harmonic mean of the precision and recall: it accounts for both false positives and false negatives.

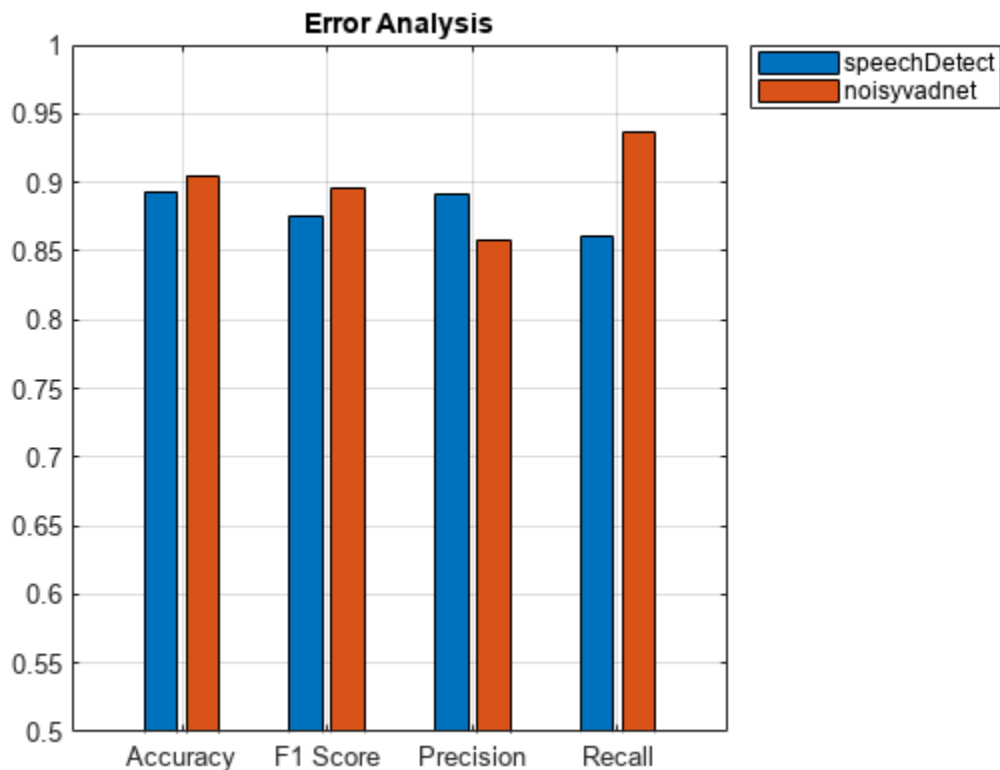
$$\text{F1 Score} = 2 \left(\frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \right)$$

The true measure of a network depends on your application. In real-world situations, a cost function is usually optimized which weights the costs of false positives and false negatives.

```
TP = speechDetectResults(2,2);
TN = speechDetectResults(1,1);
FP = speechDetectResults(1,2);
FN = speechDetectResults(2,1);
speechDetectAccuracy = (TP+TN)/(TP+TN+FP+FN);
speechDetectRecall = TP/(TP+FN);
speechDetectPrecision = TP/(TP+FP);
speechDetectF1Score = 2*(speechDetectRecall*speechDetectPrecision)/(speechDetectRecall+speechDetectPrecision);
```

```
TP = noisyvadnetResults(2,2);
TN = noisyvadnetResults(1,1);
FP = noisyvadnetResults(1,2);
FN = noisyvadnetResults(2,1);
noisyvadnetAccuracy = (TP+TN)/(TP+TN+FP+FN);
noisyvadnetRecall = TP/(TP+FN);
noisyvadnetPrecision = TP/(TP+FP);
noisyvadnetF1Score = 2*(noisyvadnetRecall*noisyvadnetPrecision)/(noisyvadnetRecall+noisyvadnetPrecision);
```

```
figure
bar(categorical(["Accuracy","Recall","Precision","F1 Score"]), ...
    [speechDetectAccuracy,noisyvadnetAccuracy; ...
    speechDetectRecall,noisyvadnetRecall; ...
    speechDetectPrecision,noisyvadnetPrecision; ...
    speechDetectF1Score,noisyvadnetF1Score]);
title("Error Analysis")
legend("speechDetect","noisyvadnet",Location="bestoutside")
ylim([0.5,1])
grid on
```



Supporting Functions

Convert Feature Vectors to Sequences

```
function sequences = featureBuffer(features,featureVectorsPerSequence,overlapPercent)
% y = featureBuffer(x,sequenceLength,overlapPercent) buffers a sequence of
% feature vectors, x, into sequences of length sequenceLength overlapped by
% overlapPercent. The sequences output are returns in a cell array for
% consumption by trainNetwork.
```

```
featureVectorOverlap = round(overlapPercent*featureVectorsPerSequence);
hopLength = featureVectorsPerSequence - featureVectorOverlap;
```

```
N = floor((size(features,2) - featureVectorsPerSequence)/hopLength) + 1;
sequences = cell(N,1);
```

```
idx = 1;
for jj = 1:N
    sequences{jj} = features(:,idx:idx + featureVectorsPerSequence - 1);
    idx = idx + hopLength;
```

```
end
```

```
end
```


Mix SNR

```

function [noisySignal,requestedNoise] = mixSNR(signal,noise,ratio)
% [noisySignal,requestedNoise] = mixSNR(signal,noise,ratio) returns a noisy
% version of the signal, noisySignal. The noisy signal has been mixed with
% noise at the specified ratio in dB.

numSamples = size(signal,1);

% Convert noise to mono
noise = mean(noise,2);

% Trim or expand noise to match signal size
if size(noise,1)>=numSamples
    % Choose a random starting index such that you still have numSamples
    % after indexing the noise.
    start = randi(size(noise,1) - numSamples + 1);
    noise = noise(start:start+numSamples-1);
else
    numReps = ceil(numSamples/size(noise,1));
    temp = repmat(noise,numReps,1);
    start = randi(size(temp,1) - numSamples - 1);
    noise = temp(start:start+numSamples-1);
end

signalNorm = norm(signal);
noiseNorm = norm(noise);

goalNoiseNorm = signalNorm/(10^(ratio/20));
factor = goalNoiseNorm/noiseNorm;

requestedNoise = noise.*factor;
noisySignal = signal + requestedNoise;

noisySignal = noisySignal./max(abs(noisySignal));
end

```

Construct Signal

```

function [audio,mask] = constructSignal(ds,fs,duration)
% [audio,mask] = constructSignal(ds,fs,duration) constructs an audio signal
% of the specified duration by concatenating samples from the
% audioDatastore ds with random duration of silence between.

win = hamming(50e-3*fs,"periodic");

% 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.
N = duration*fs;
audio = zeros(N,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.
mask = zeros(N,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 < N
    data = read(ds);
    data = data ./ max(abs(data)); % Scale amplitude

    % Determine regions of speech
    idx = detectSpeech(data,fs,Window=win);

    % 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
        audio(numSamples:numSamples+numel(data)-1) = data;

        % Set VAD baseline
        mask(numSamples:numSamples+numel(data)-1) = true;

        % Random silence period
        numSilenceSamples = randi(maxSilenceSegment*fs,1,1);
        numSamples = numSamples + numel(data) + numSilenceSamples;
    end
end
audio = audio(1:N);
mask = mask(1:N);
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

Voice Activity Detection in Noise Using Deep Learning

In this example, you perform batch and streaming voice activity detection (VAD) in a low SNR environment using a pretrained deep learning model. For details about the model and how it was trained, see “Train Voice Activity Detection in Noise Model Using Deep Learning” on page 1-430.

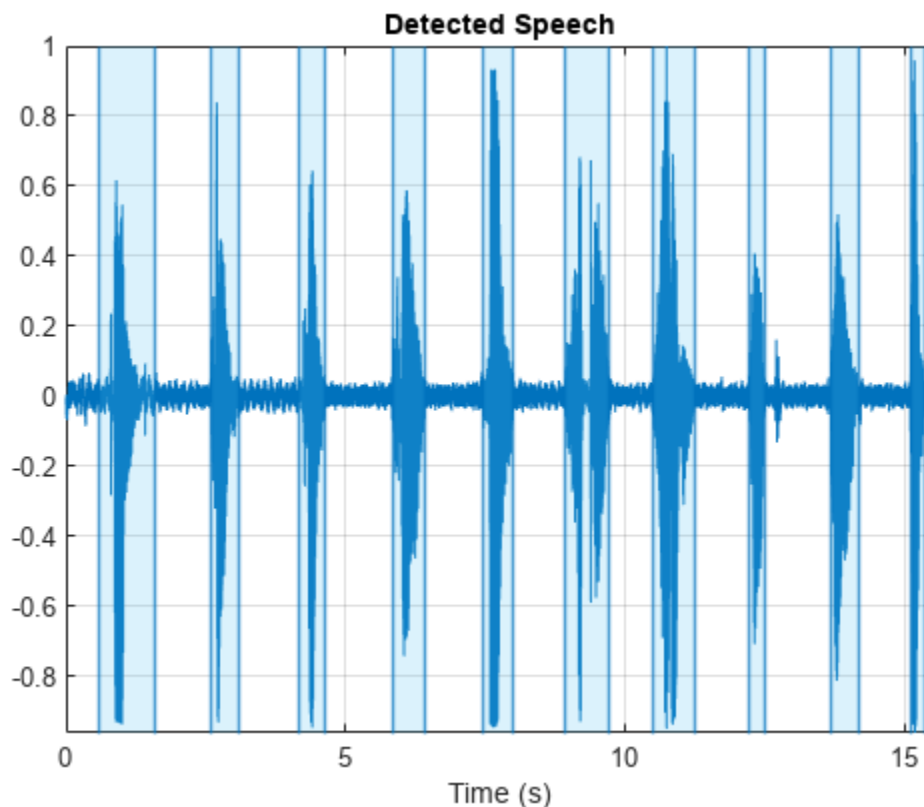
Load and Inspect Data

Read in an audio file that consists of words spoken with pauses between and listen to it. Use `resample` to resample the signal to the sample rate to 16 kHz. Use `detectSpeech` on the clean signal to determine the ground-truth speech regions.

```
fs = 16e3;
[speech,fileFs] = audioread("Counting-16-44p1-mono-15secs.wav");
speech = resample(speech,fs,fileFs);
speech = speech./max(abs(speech));

sound(speech,fs)

detectSpeech(speech,fs,Window=hamming(0.04*fs,"periodic"),MergeDistance=round(0.5*fs))
```



Load a noise signal and resample to the audio sample rate.

```
[noise,fileFs] = audioread("WashingMachine-16-8-mono-200secs.mp3");
noise = resample(noise,fs,fileFs);
```

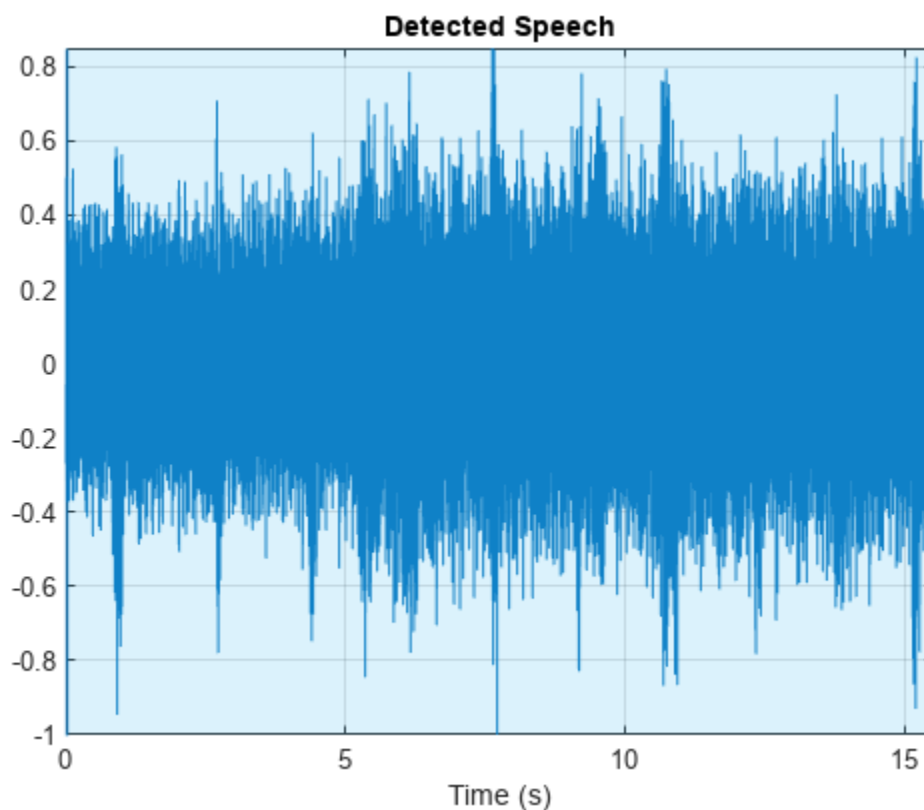
Use the supporting function `mixSNR` on page 1-457 to corrupt the clean speech signal with washing machine noise at a desired SNR level in dB. Listen to the corrupted audio. The network was trained under -10 dB SNR conditions.

```
SNR = -10  ;
noisySpeech = mixSNR(speech,noise,SNR);

sound(noisySpeech,fs)
```

The algorithm-based VAD, `detectSpeech`, fails under these noisy conditions.

```
detectSpeech(noisySpeech,fs,Window=hamming(0.04*fs,"periodic"),MergeDistance=round(0.5*fs))
```



Download Pretrained Network

Download and load a pretrained network and a configured `audioFeatureExtractor` object. The network was trained to detect speech in low SNR environments given features output from the `audioFeatureExtractor` object.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","VoiceActivityDetection.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
netFolder = fullfile(dataFolder,"VoiceActivityDetection");
pretrainedNetwork = load(fullfile(netFolder,"voiceActivityDetectionExample.mat"));

afe = pretrainedNetwork.afe;
net = pretrainedNetwork.speechDetectNet;
```

The `audioFeatureExtractor` object is configured to extract features from 256-sample windows with 128 samples overlap between windows. At a 16 kHz sample rate, features are extracted from 16 ms windows with 8 ms overlap. From each window, the `audioFeatureExtractor` object extracts nine features: spectral centroid, spectral crest, spectral entropy, spectral flux, spectral kurtosis, spectral rolloff point, spectral skewness, spectral slope, and harmonic ratio.

```
afe
```

```
afe =
```

```
audioFeatureExtractor with properties:
```

```
Properties
```

```

    Window: [256x1 double]
    OverlapLength: 128
    SampleRate: 16000
    FFTLength: []
    SpectralDescriptorInput: 'linearSpectrum'
    FeatureVectorLength: 9

```

```
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, zerocrossrate, shortTimeEnergy
```

To extract a feature, set the corresponding property to true.

For example, `obj.mfcc = true`, adds `mfcc` to the list of enabled features.

The network consists of two bidirectional LSTM layers, each with 200 hidden units, and a classification output that returns either class 0 corresponding to no voice activity detected or class 1 corresponding to voice activity detected.

```
net.Layers
```

```
ans =
```

```
6x1 Layer array with layers:
```

1	'sequenceinput'	Sequence Input	Sequence input with 9 dimensions
2	'biLSTM_1'	BiLSTM	BiLSTM with 200 hidden units
3	'biLSTM_2'	BiLSTM	BiLSTM with 200 hidden units
4	'fc'	Fully Connected	2 fully connected layer
5	'softmax'	Softmax	softmax
6	'classoutput'	Classification Output	crossentropyex with classes '0' and '1'

Perform Voice Activity Detection

Extract features from the speech data and then standardize 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(net, features);
```

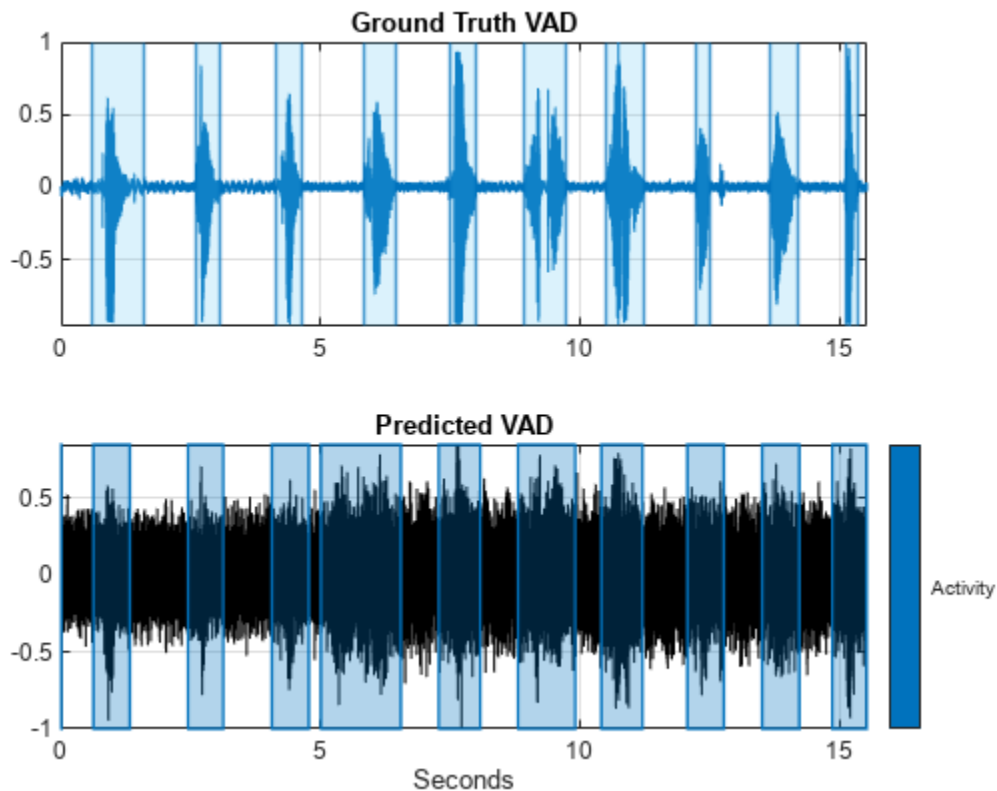
Each decision corresponds to an analysis window analyzed by the `audioFeatureExtractor`. Replicate the decisions so that they are in one-to-one correspondence with the audio samples. Use the `detectSpeech` convenience plot to plot the ground truth. Use `signalMask` and `plotsigroi` to plot the predicted VAD.

```
decisions = (double(decisionsCategorical) - 1)';
decisionsPerSample = [decisions(1:round(numel(afe.Window)/2))]; repelem(decisions, numel(afe.Window))

tiledlayout(2,1)

nexttile
detectSpeech(speech, fs, Window=hamming(0.04*fs, "periodic"), MergeDistance=round(0.5*fs))
title("Ground Truth VAD")
xlabel("")

nexttile
mask = signalMask(decisionsPerSample, SampleRate=fs, Categories="Activity");
plotsigroi(mask, noisySpeech, true)
title("Predicted VAD")
```



Perform Streaming Voice Activity Detection

The `audioFeatureExtractor` object is intended for batch processing and does not retain state between calls. Use `generateMATLABFunction` to create a streaming-friendly feature extractor. You can use the trained VAD network in a streaming context using `classifyAndUpdateState` (Deep Learning Toolbox).

```
generateMATLABFunction(afe, "featureExtractor", IsStreaming=true)
```

To simulate a streaming environment, save the speech and noise signals as WAV files. To simulate streaming input, you will use `dsp.AudioFileReader` to read frames from the files and mix them at a desired SNR. You can also use `audioDeviceReader` so that your microphone is the speech source.

```
audiowrite("Speech.wav", speech, fs)
audiowrite("Noise.wav", noise, fs)
```

Define parameters for the streaming voice activity detection in noise demonstration:

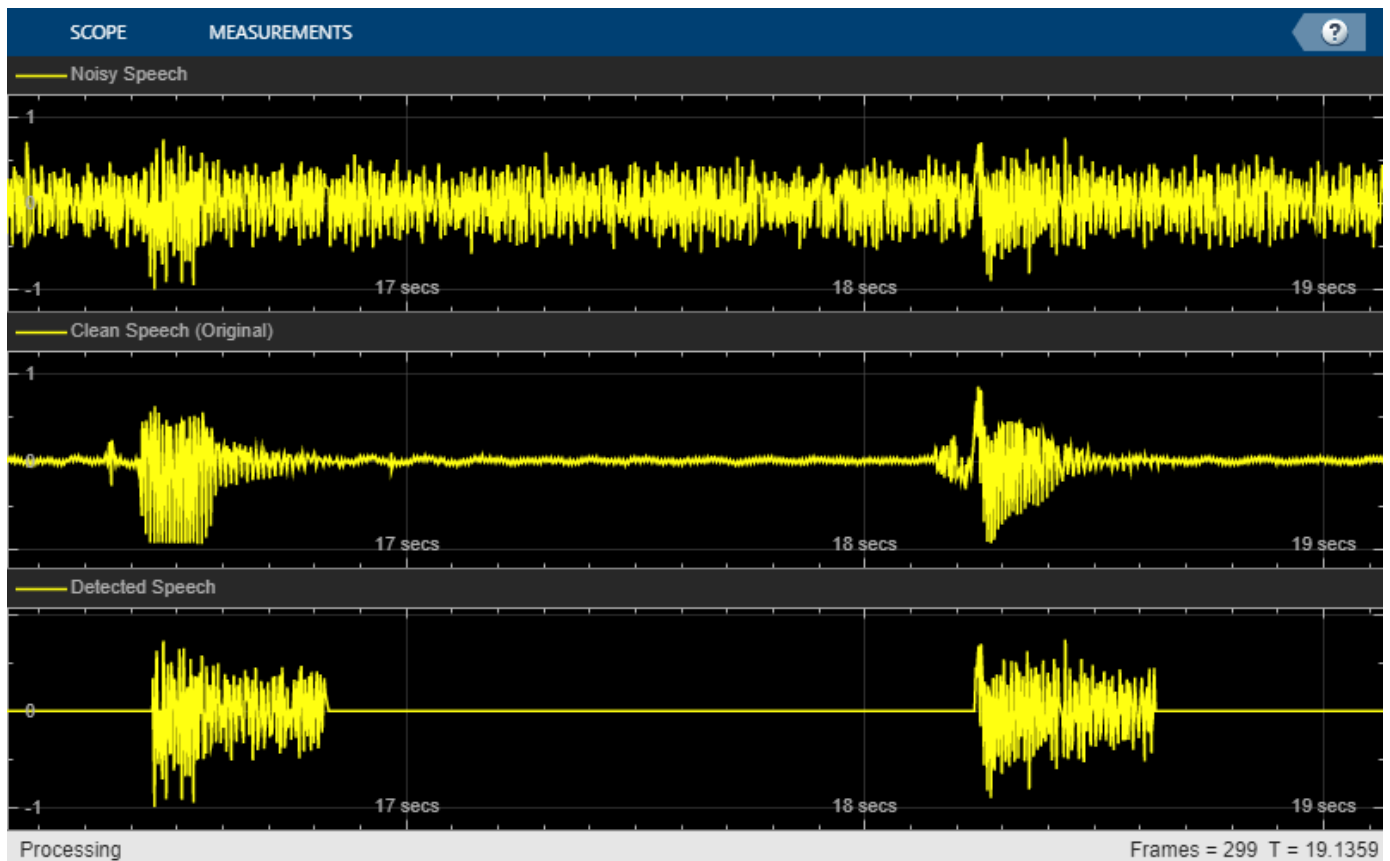
- `signal` - Signal source, specified as either the speech file previously recorded, or your microphone.
- `noise` - Noise source, specified as a noise sound file to mix with the signal.
- `SNR` - Signal-to-noise ratio to mix the signal and noise, specified in dB.
- `testDuration` - Test duration, specified in seconds.
- `playbackSource` - Playback source, specified as either the original clean signal, the noisy signal, or the detected speech. An `audioDeviceWriter` object is used to play the audio to your speakers.

```
signal =  ;
noise = "Noise.wav";
SNR = -10  ; % dB

testDuration = 20  ; % seconds
playbackSource =  ;
```

Call the supporting function `streamingDemo` on page 1-454 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.

```
streamingDemo(net, afe, ...
    signal, noise, SNR, ...
    testDuration, playbackSource);
```



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

Supporting Functions

Streaming Demo

```
function streamingDemo(net,afe,signal,noise,SNR,testDuration,playbackSource)
% streamingDemo(net,afe,signal,noise,SNR,testDuration,playbackSource) runs
% a real-time VAD demo.

% Create dsp.AudioFileReader objects to read speech and noise files frame
% by frame. If the speech signal is specified as Microphone, use an
% audioDeviceReader as the source.
if strcmpi(signal,"Microphone")
    speechReader = audioDeviceReader(afe.SampleRate);
else
    speechReader = dsp.AudioFileReader(signal,PlayCount=inf);
end
noiseReader = dsp.AudioFileReader(noise,PlayCount=inf,SamplesPerFrame=speechReader.SamplesPerFrame);
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 standardization. The statistics should improve
% over time.
movSTD = dsp.MovingStandardDeviation(Method="Exponential weighting",ForgettingFactor=1);
movMean = dsp.MovingAverage(Method="Exponential weighting",ForgettingFactor=1);

% Create a dsp.MovingMaximum object. You will use it to standardize the
% audio.
movMax = dsp.MovingMaximum(SpecifyWindowLength=false);

% Create a dsp.MovingRMS object. You will use this to determine the signal
% and noise mix at the desired SNR. This object is only useful for example
% purposes where you are artificially adding noise.
movRMS = dsp.MovingRMS(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 audio so
% that VAD decisions correspond to the audio signal. The output buffer is
% only necessary for visualizing the decisions in real time.
audioInBuffer = dsp.AsyncBuffer(2*speechReader.SamplesPerFrame);
featureBuffer = dsp.AsyncBuffer(ceil(2*speechReader.SamplesPerFrame/(numel(afe.Window)-afe.OverlapLength)));
audioOutBuffer = dsp.AsyncBuffer(2*speechReader.SamplesPerFrame);

% Create a time scope to visualize the original speech signal, the noisy
% signal that the network is applied to, and the decision output from the
% network.
scope = timescope(SampleRate=fs, ...
    TimeSpanSource="property", ...
    TimeSpan=3, ...
    BufferLength=fs*3*3, ...
    TimeSpanOvverrunAction="Scroll", ...
    AxesScaling="updates", ...
    MaximizeAxes="on", ...
    AxesScalingNumUpdates=20, ...
    NumInputPorts=3, ...
    LayoutDimensions=[3,1], ...
    ChannelNames=["Noisy Speech","Clean Speech (Original)","Detected Speech"], ...
    ...
    ActiveDisplay = 1, ...
    ShowGrid=true, ...
    ...
    ActiveDisplay = 2, ...
    ShowGrid=true, ...
    ...
    ActiveDisplay=3, ...
    ShowGrid=true); %#ok<DUPNAMEARG>
setup(scope,{1,1,1})

% 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;

```

```
% Run the streaming demonstration.
loopTimer = tic;
while toc(loopTimer) < 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
    energy = movRMS([speechIn,noiseIn]);
    noiseGain = 10^(-SNR/20) * energy(end,1) / energy(end,2);
    noisyAudio = speechIn + noiseGain*noiseIn;

    % Update a running max to scale the audio
    myMax = movMax(abs(noisyAudio));
    noisyAudio = noisyAudio/myMax(end);

    % Write the noisy audio and speech to buffers
    write(audioInBuffer,[noisyAudio,speechIn]);

    % 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)
        x = read(audioInBuffer,numel(afe.Window),afe.OverlapLength);
        write(audioOutBuffer,x(end-hopLength+1:end,:));
        noisyAudio = x(:,1);
        features = featureExtractor(noisyAudio);
        write(featureBuffer,features');
    end

    if featureBuffer.NumUnreadSamples >= 1
        % Read the audio data corresponding to the number of unread
        % feature vectors.
        audioHop = read(audioOutBuffer,featureBuffer.NumUnreadSamples*hopLength);

        % Read all unread feature vectors.
        features = read(featureBuffer);

        % Use only the new features to update the standard deviation and
        % mean. Normalize the features.
        rmean = movMean(features);
        rstdev = movSTD(features);
        features = (features - rmean(end,:)) ./ rstdev(end,:);

        % Network inference
        [net,decision] = classifyAndUpdateState(net,features');

        % Convert the decisions per feature vector to decisions per sample
        decision = repelem(decision,hopLength,1);

        % Apply a mask to the noisy speech for visualization
        vadResult = audioHop(:,1);
        vadResult(decision==categorical(0)) = 0;

        % Listen to the speech or speech+noise
        switch playbackSource
            case "clean"
```

```

        deviceWriter(audioHop(:,2));
    case "noisy"
        deviceWriter(audioHop(:,1));
    case "detectedSpeech"
        deviceWriter(vadResult);
    end

    % Visualize the speech+noise, the original speech, and the voice
    % activity detection.
    scope(audioHop(:,1),audioHop(:,2),vadResult)

    end
end
end

```

Mix SNR

```

function [noisySignal,requestedNoise] = mixSNR(signal,noise,ratio)
% [noisySignal,requestedNoise] = mixSNR(signal,noise,ratio) returns a noisy
% version of the signal, noisySignal. The noisy signal has been mixed with
% noise at the specified ratio in dB.

numSamples = size(signal,1);

% Convert noise to mono
noise = mean(noise,2);

% Trim or expand noise to match signal size
if size(noise,1)>=numSamples
    % Choose a random starting index such that you still have numSamples
    % after indexing the noise.
    start = randi(size(noise,1) - numSamples + 1);
    noise = noise(start:start+numSamples-1);
else
    numReps = ceil(numSamples/size(noise,1));
    temp = repmat(noise,numReps,1);
    start = randi(size(temp,1) - numSamples - 1);
    noise = temp(start:start+numSamples-1);
end

signalNorm = norm(signal);
noiseNorm = norm(noise);

goalNoiseNorm = signalNorm/(10^(ratio/20));
factor = goalNoiseNorm/noiseNorm;

requestedNoise = noise.*factor;
noisySignal = signal + requestedNoise;

noisySignal = noisySignal./max(abs(noisySignal));
end

```

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 `midiid` function to explicitly set the MIDI device parameter on the appropriate blocks in the model to recognize a specific control. For example, running `midiid` with the Korg nanoKONTROL device produces the following information:

```
>> [ctl device]=midiid
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.

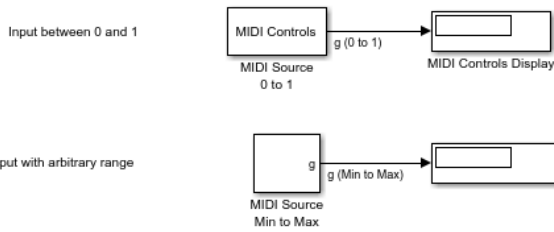
Example 4: MIDI Controls to Mix Two Signals Into One

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.

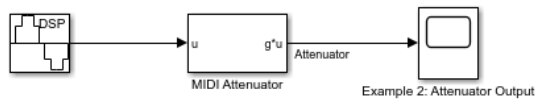
Example 5: MIDI Controls to Generate a Sinusoid with Arbitrary Phase

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.

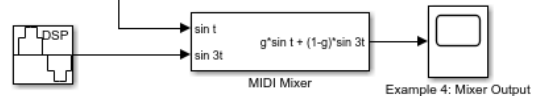
Example 1: Use the MIDI control surface as a user-provided input to your model



Example 2: Use the MIDI control surface to adjust the gain on a single signal



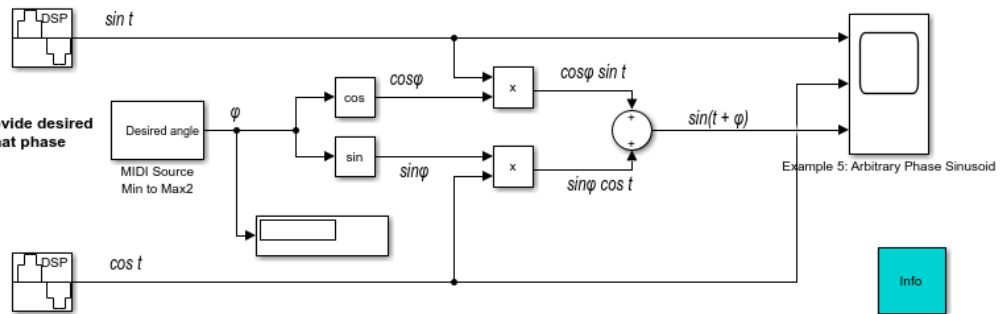
Example 3: Use the MIDI control surface to control the relative amplitudes as you split a single signal into two



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. In this example, the data is stored in a folder under `tempdir`.

```
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)
```

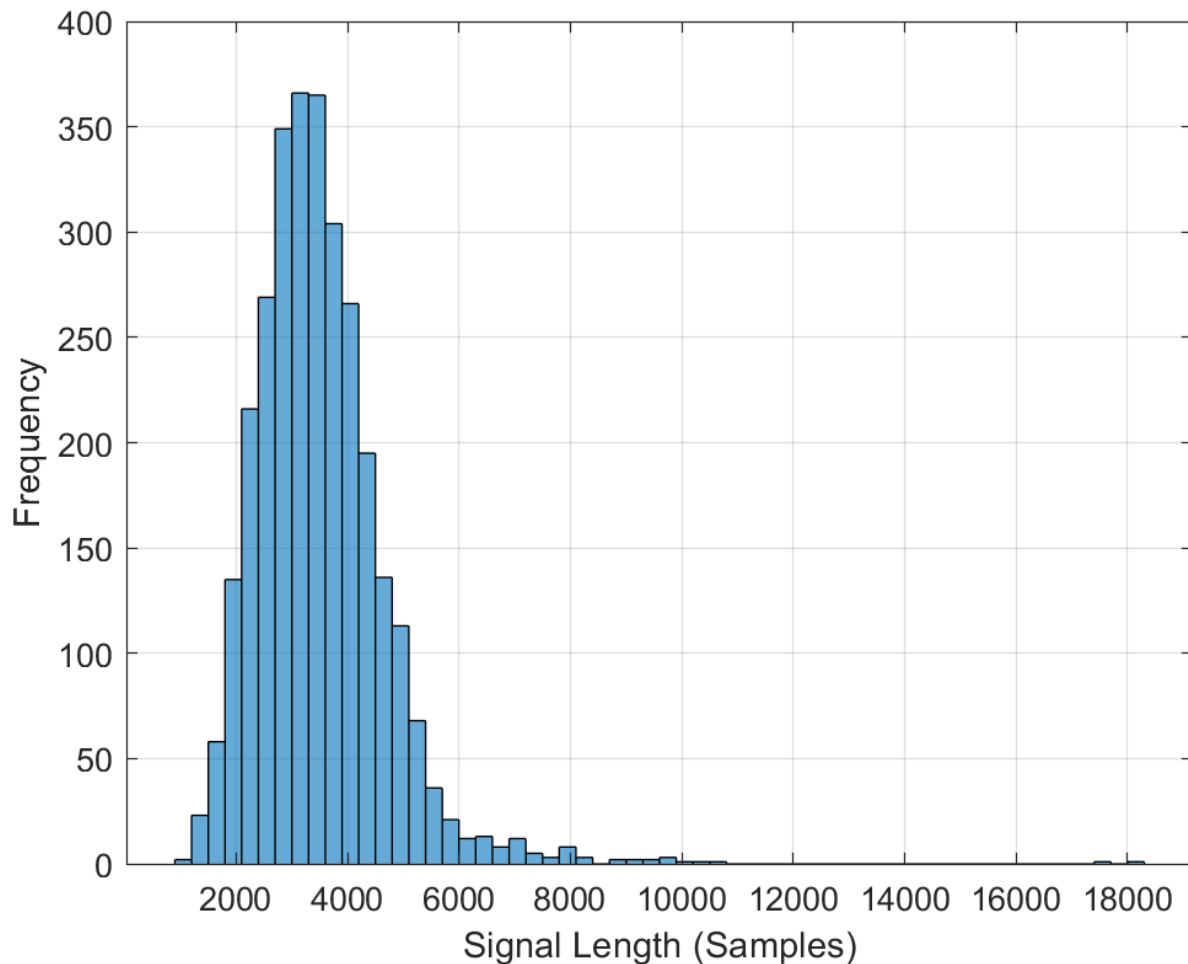
```

0      300
1      300
2      300
3      300
4      300
5      300
6      300
7      300
8      300
9      300
```

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     240
         1     240
         2     240
         3     240
         4     240
         5     240
         6     240
         7     240
         8     240
         9     240
```

```
countEachLabel(adsTest)
```

```
ans=10x2 table
    Label    Count
    -----
         0     60
         1     60
         2     60
         3     60
         4     60
         5     60
         6     60
         7     60
         8     60
         9     60
```

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', 'ClassifError'))*100  
  
validationAccuracy = 97.4167
```

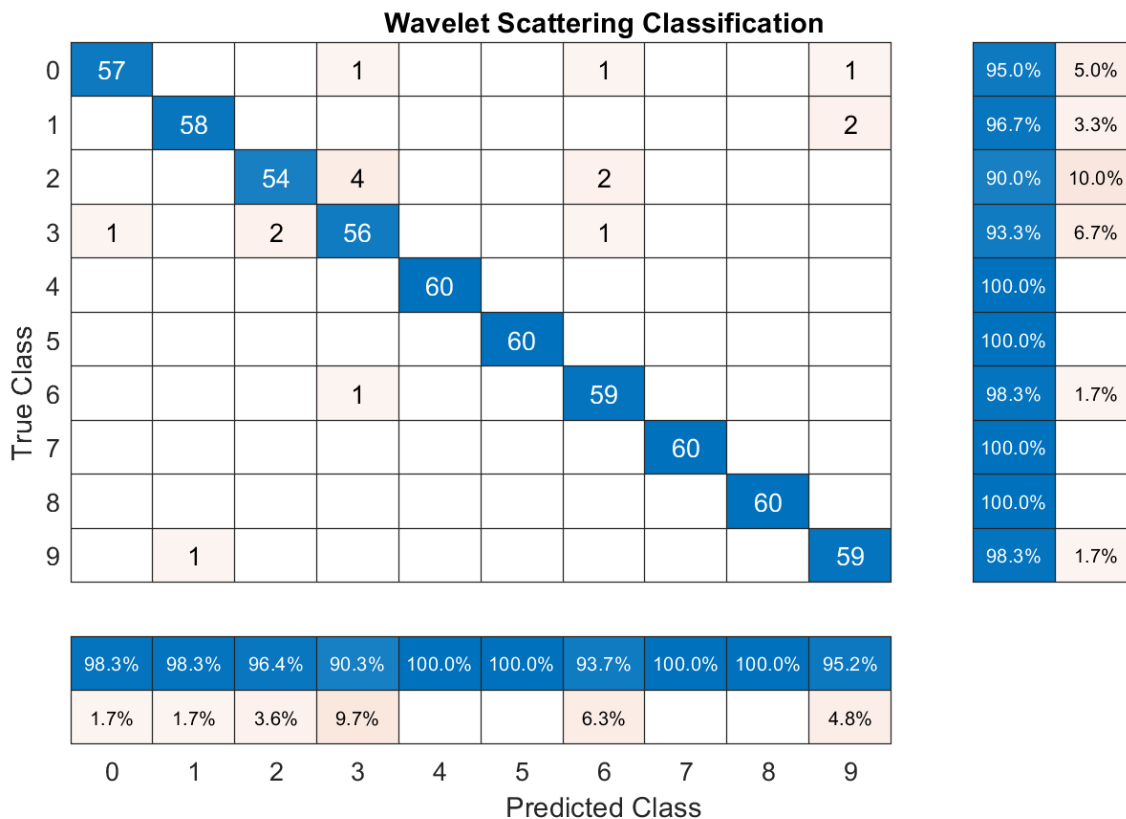
The estimated generalization accuracy is approximately 97%. Use the trained SVM to predict the spoken-digit classes in the test set.

```
predLabels = predict(classificationSVM,TestFeatures);
testAccuracy = sum(predLabels==adsTest.Labels)/numel(predLabels)*100
```

```
testAccuracy = 97.1667
```

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 = 96.3333
```

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')
];
```

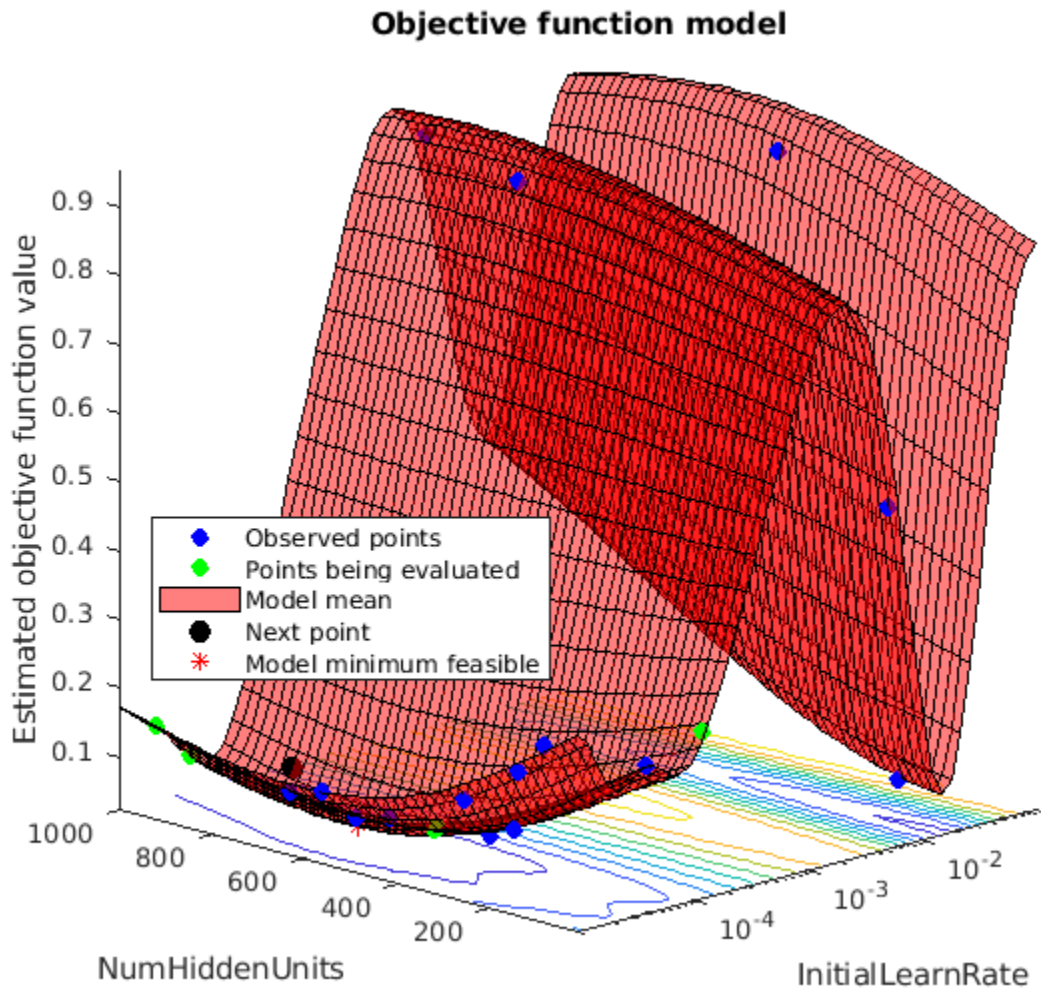
Bayesian optimization is computationally intensive and can take several hours to finish. For the purposes of this example, set `optimizeCondition` to `false` to download and use predetermined optimized hyperparameter settings. If you set `optimizeCondition` to `true`, the objective function `helperBayesOptLSTM` 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).

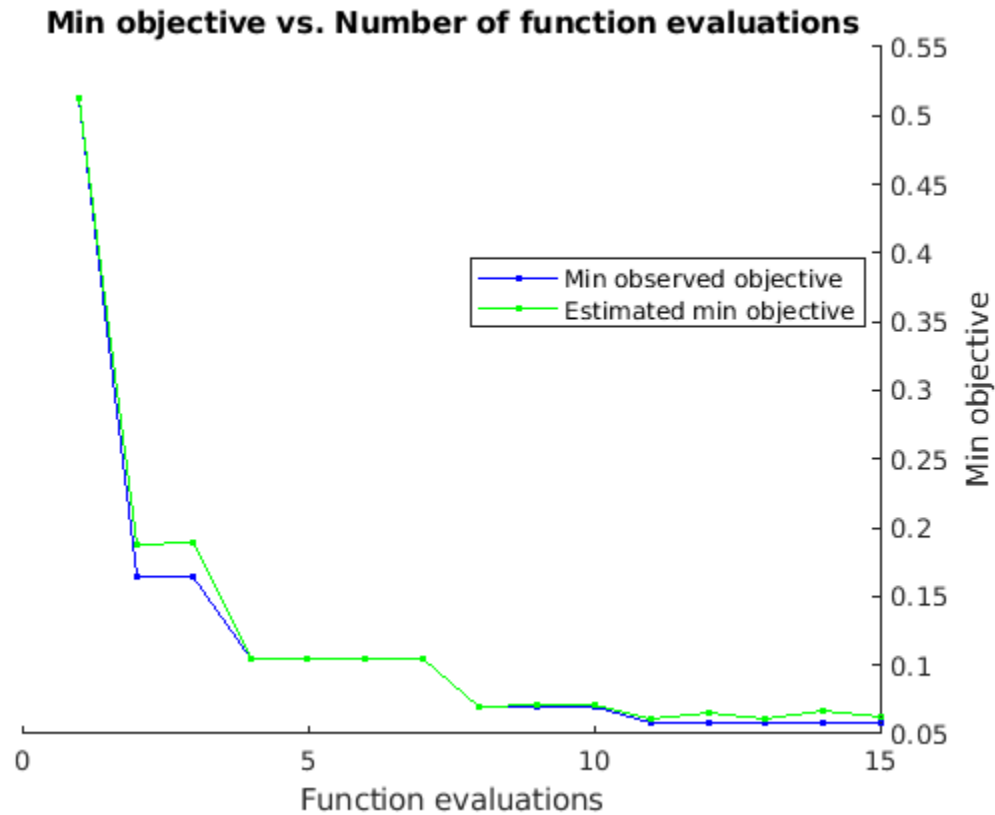
```
ObjFcn = helperBayesOptLSTM(TrainFeatures,YTrain,TestFeatures,YTest);

optimizeCondition = false;
if optimizeCondition
    BayesObject = bayesopt(ObjFcn,optVars,...
        'MaxObjectiveEvaluations',15,...
        'IsObjectiveDeterministic',false,...
        'UseParallel',true);
else
    url = 'http://ssd.mathworks.com/supportfiles/audio/SpokenDigitRecognition.zip';
    downloadNetFolder = tempdir;
    netFolder = fullfile(downloadNetFolder,'SpokenDigitRecognition');
    if ~exist(netFolder,'dir')
        disp('Downloading pretrained network (1 file - 12 MB) ...')
        unzip(url,downloadNetFolder)
    end
    load(fullfile(netFolder,'0.02.mat'));
end
```

```
Downloading pretrained network (1 file - 12 MB) ...
```

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 = 97.5000
```

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.

```
epsil = 1e-6;
XTrain = helperspeechSpectrograms(adsTrain,segmentDuration,frameDuration,hopDuration,numBands);
```

```
Computing speech spectrograms...
Processed 500 files out of 2400
Processed 1000 files out of 2400
Processed 1500 files out of 2400
Processed 2000 files out of 2400
...done
```

```
XTrain = log10(XTrain + epsil);
```

```
XTest = helperspeechSpectrograms(adsTest,segmentDuration,frameDuration,hopDuration,numBands);
```

```
Computing speech spectrograms...
Processed 500 files out of 600
...done
```

```
XTest = log10(XTest + epsil);
```

```
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);
```

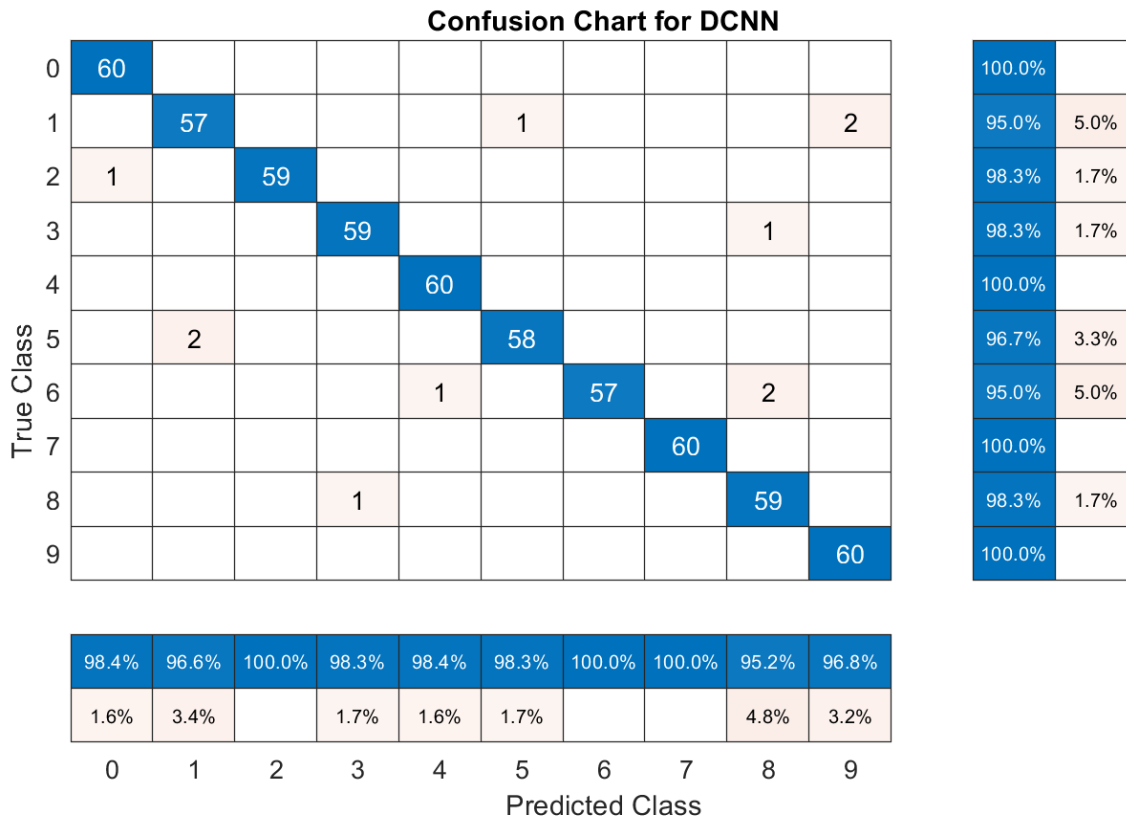
Use the trained network to predict the digit labels for the test set.

```
[Ypredicted,probs] = classify(trainedNet,XTest,'ExecutionEnvironment','CPU');
cnnAccuracy = sum(Ypredicted==YTest)/numel(YTest)*100
```

```
cnnAccuracy = 98.1667
```

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)

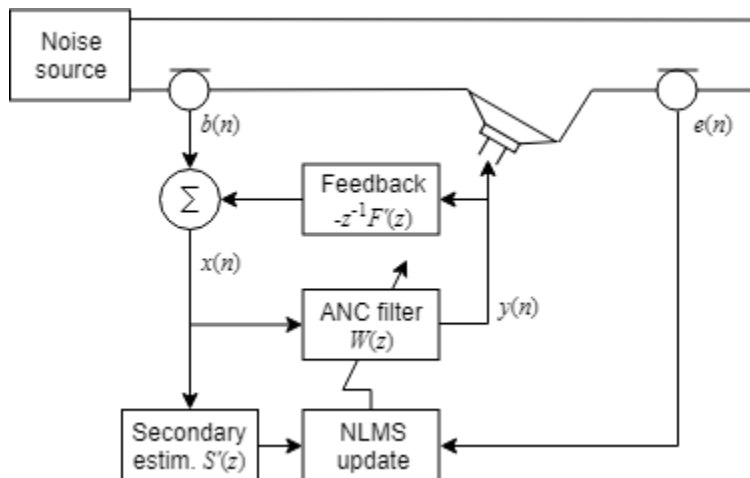
The goal of active noise control is to reduce unwanted sound by producing an “anti-noise” signal that cancels the undesired sound wave. This principle has been applied successfully to a wide variety of applications, such as noise-cancelling headphones, active sound design in car interiors, and noise 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 $b(n)$.

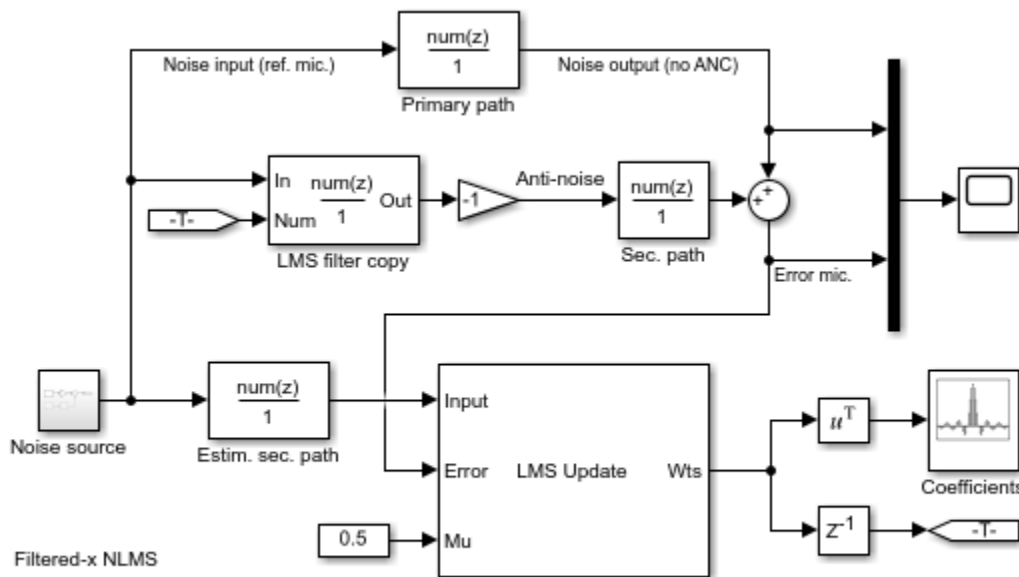
To implement a successful ANC system, we must estimate both the primary and the secondary paths. In this example, we estimate the secondary path and the acoustic feedback first and then keep it constant while the ANC system adapts the primary path.

Filtered-X ANC Model

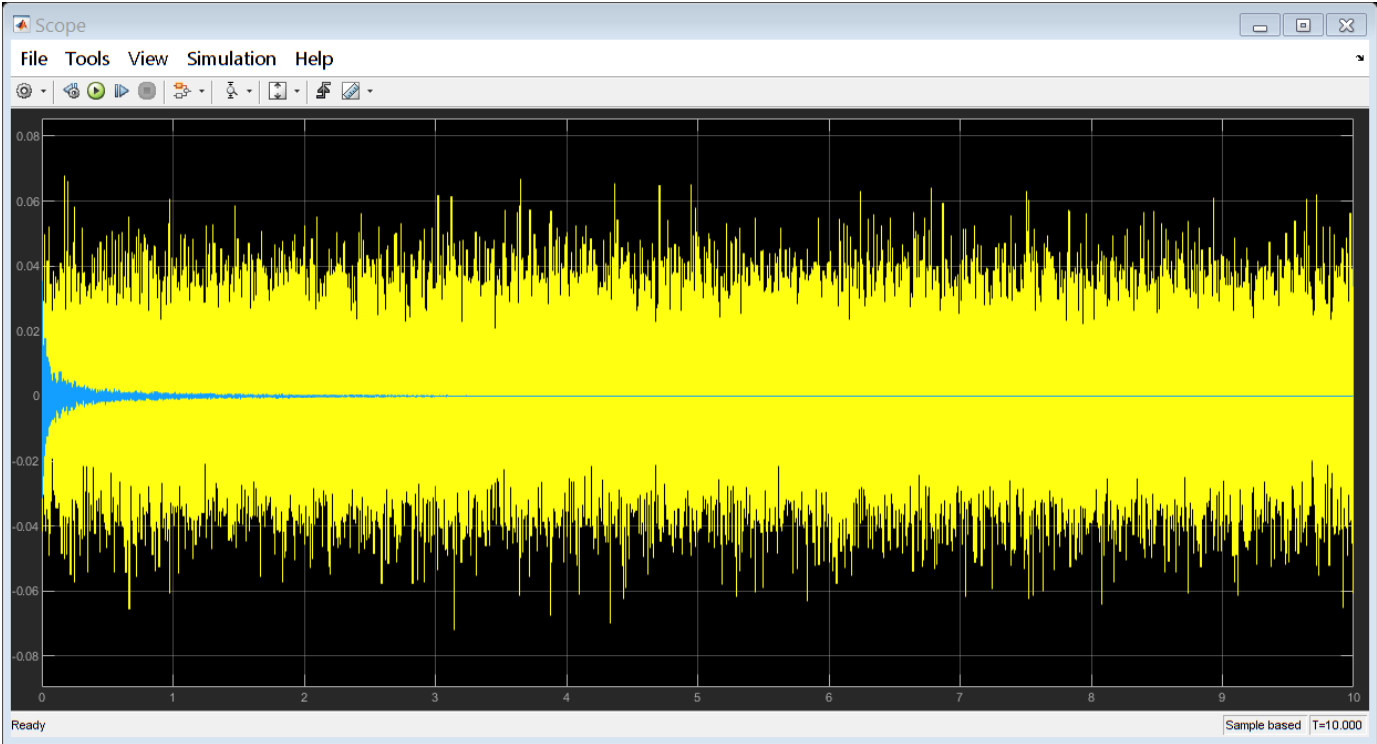
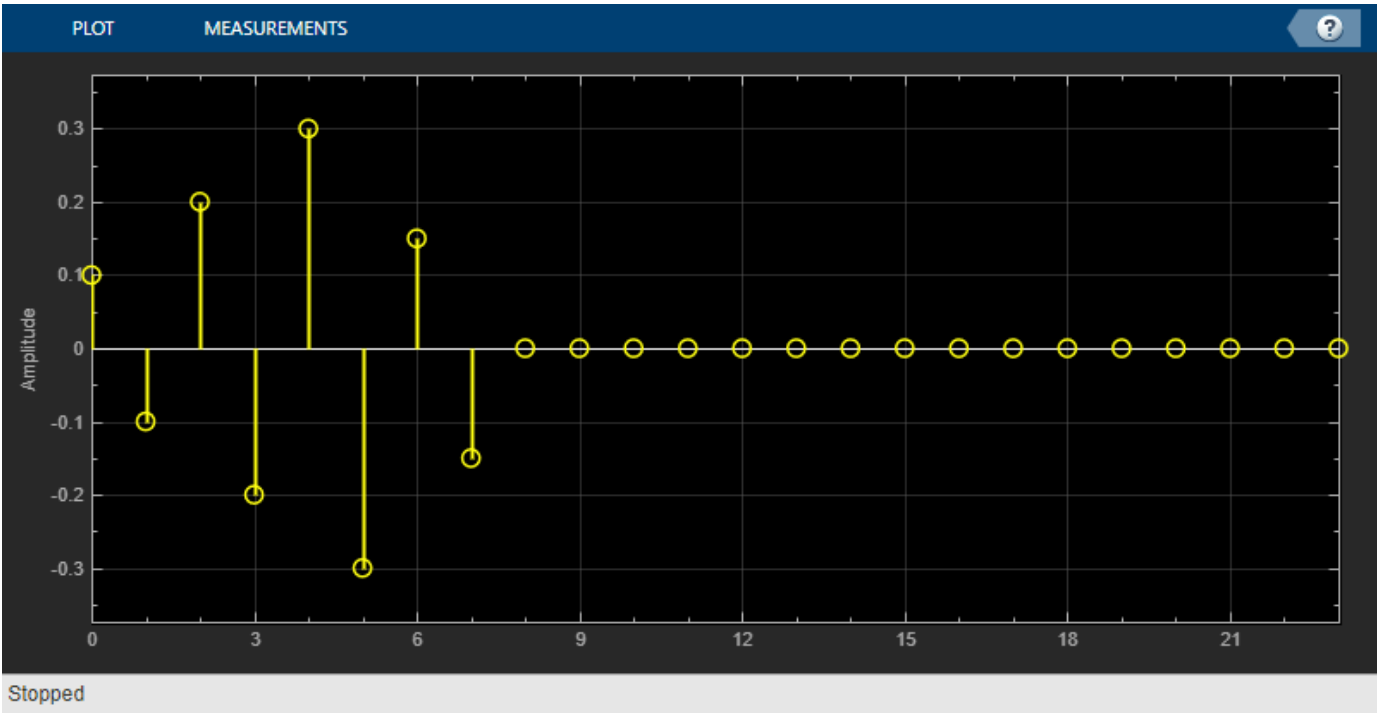
With Simulink and model-based design, you can start with a basic model of the desired system and a simulated environment. Then, you can improve the realism of that model or replace the simulated environment by the real one. You can also iterate by refining your simulated environment when you learn more 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 $conv(s(n), f(n))$, 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.

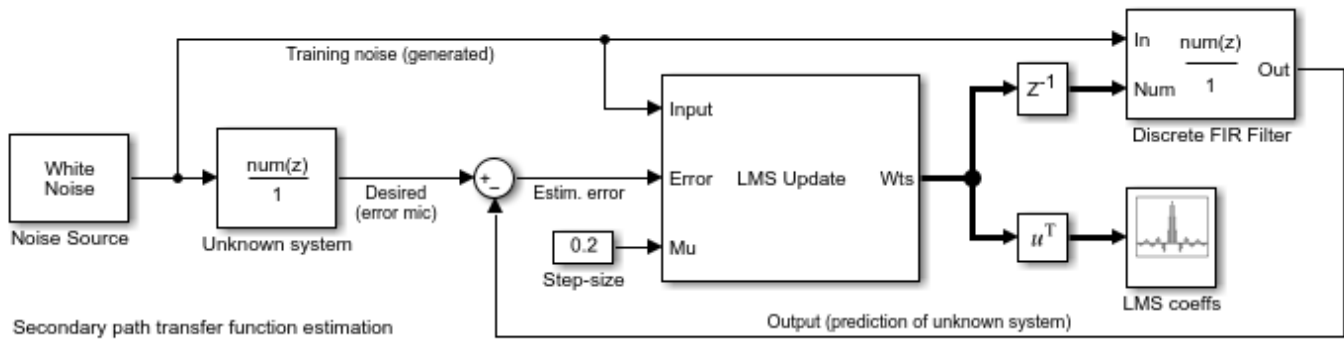


Copyright 2019 The MathWorks, Inc.

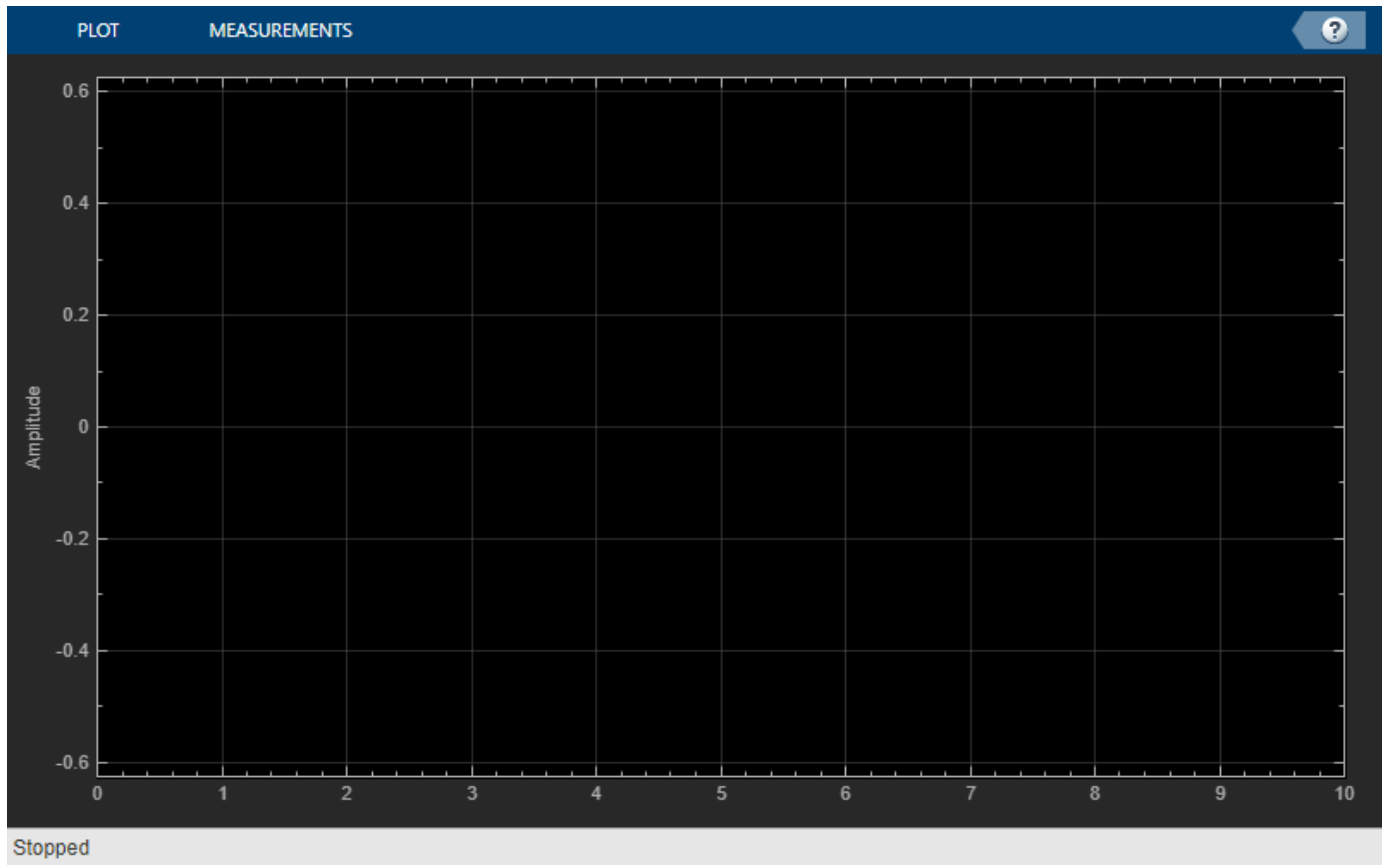


Secondary Path Estimation Model

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 $f(n)$.

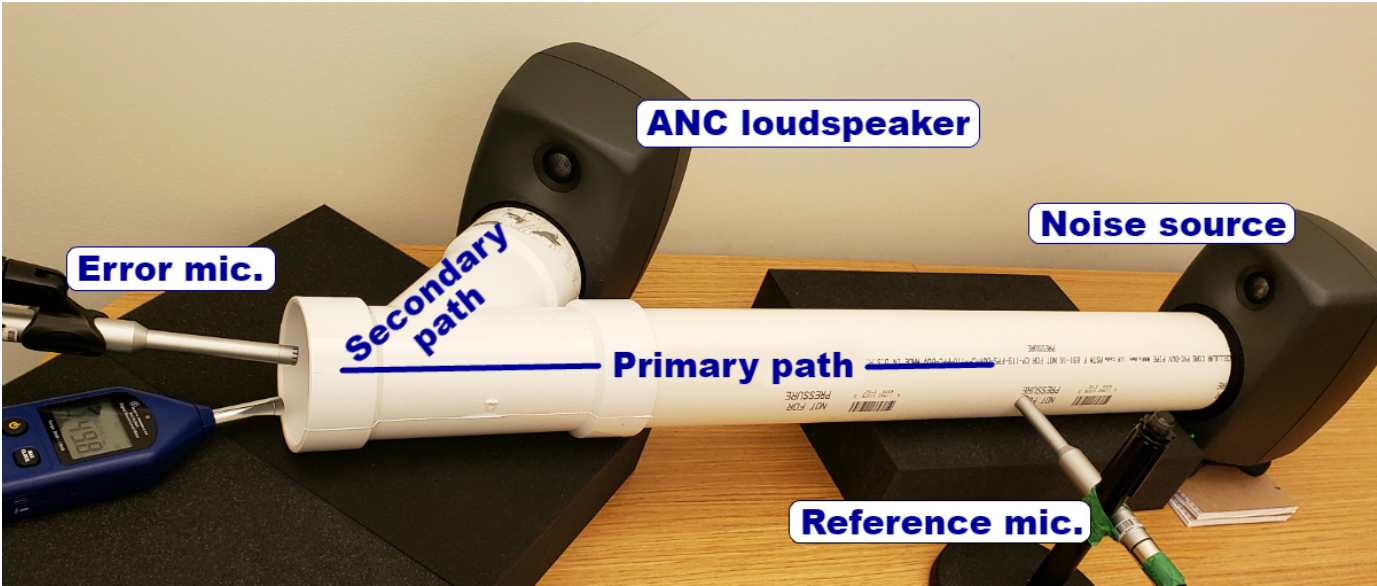


Copyright 2019 The MathWorks, Inc.



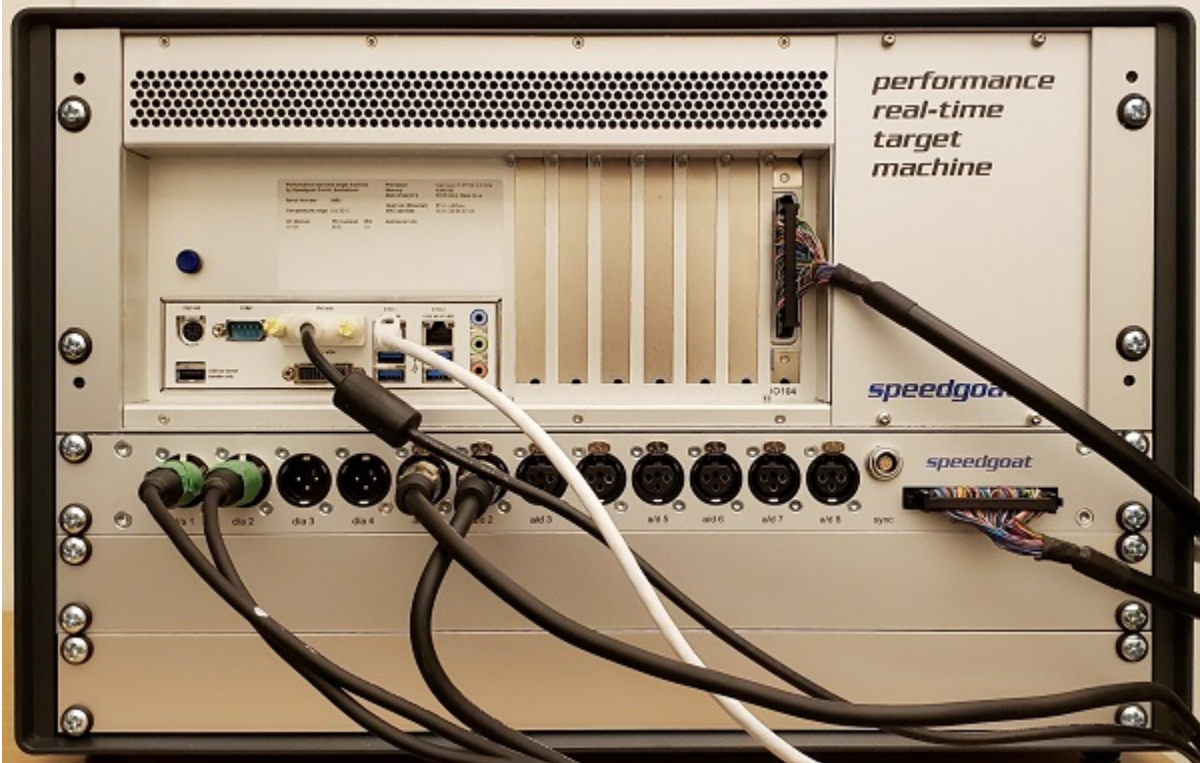
Real-Time Implementation with Speedgoat

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.

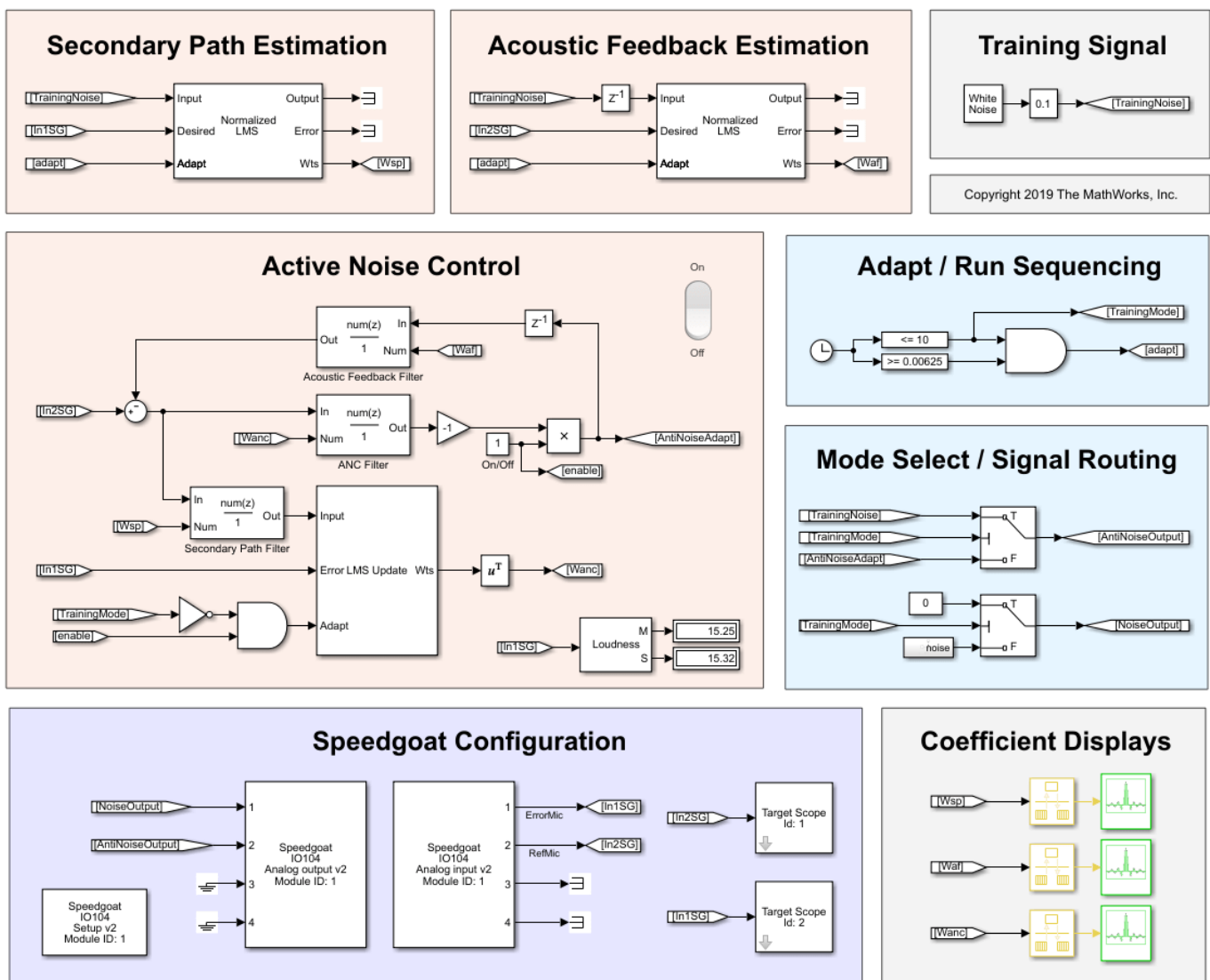


Latency is critical: the system must record the reference microphone, compute the response and play it back on the ANC loudspeaker in the time it takes for sound to travel between these points. In this example, the distance between the reference microphone and the beginning of the “Y” section is 34 cm. 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.

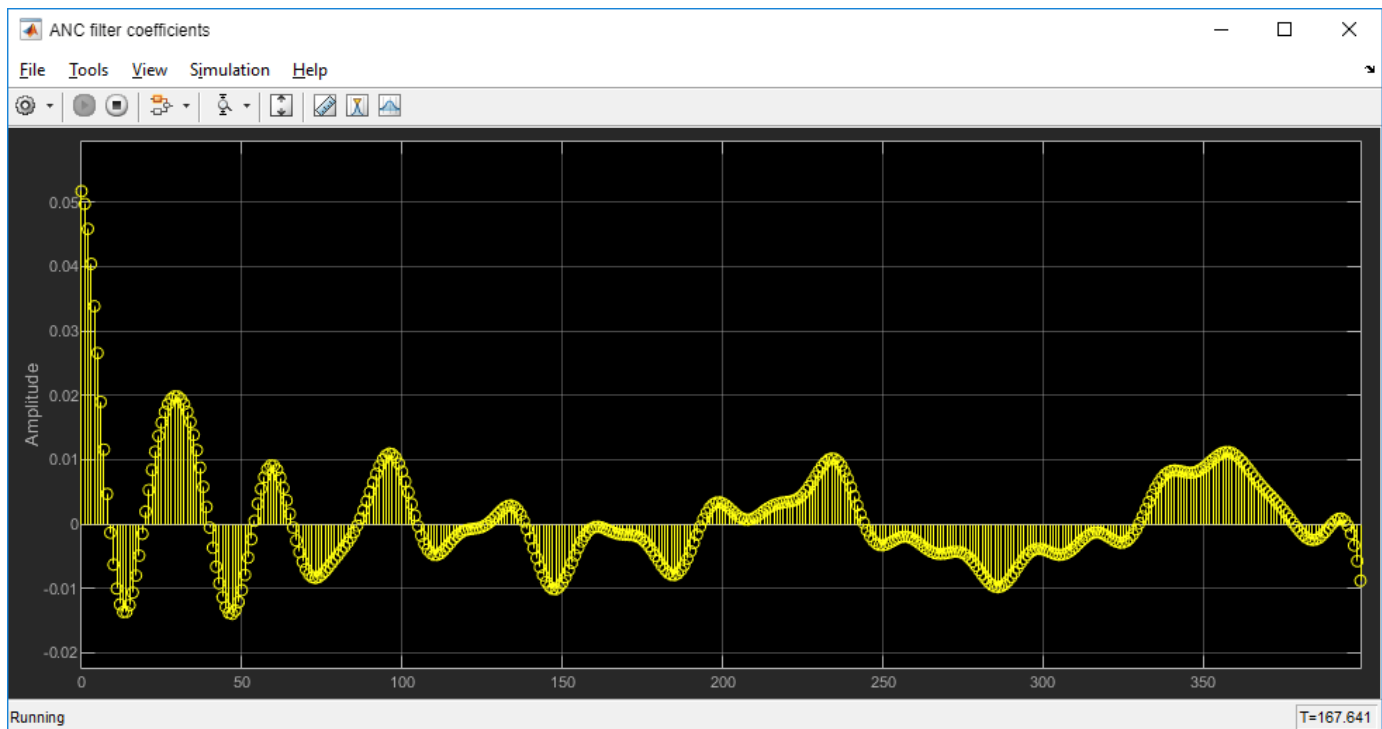


To realize our real-time model, we use the building blocks that we tested earlier, and simply replace the acoustic models by the Speedgoat I/O blocks. We also included the measurement of the acoustic feedback from the ANC loudspeaker to the reference microphone, and we added some logic to automatically 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”.



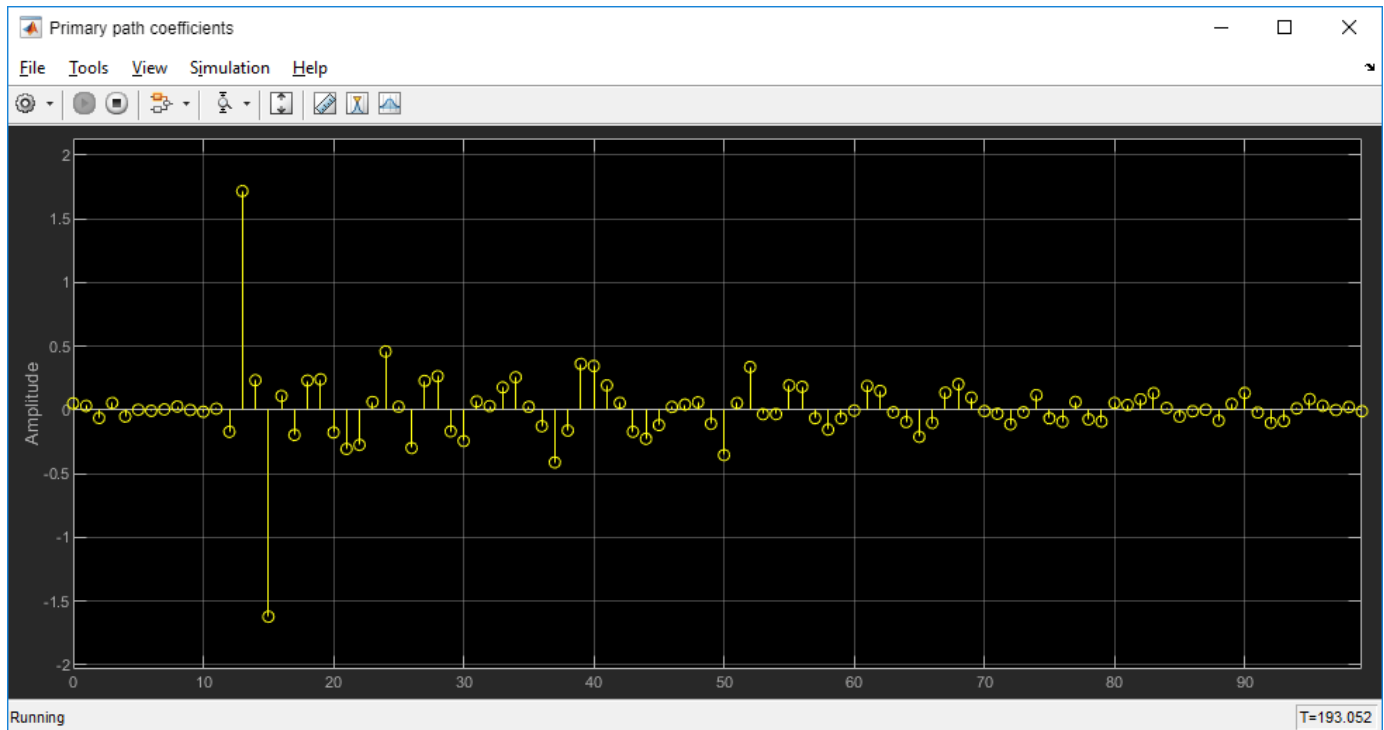
Noise Reduction Performance

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).

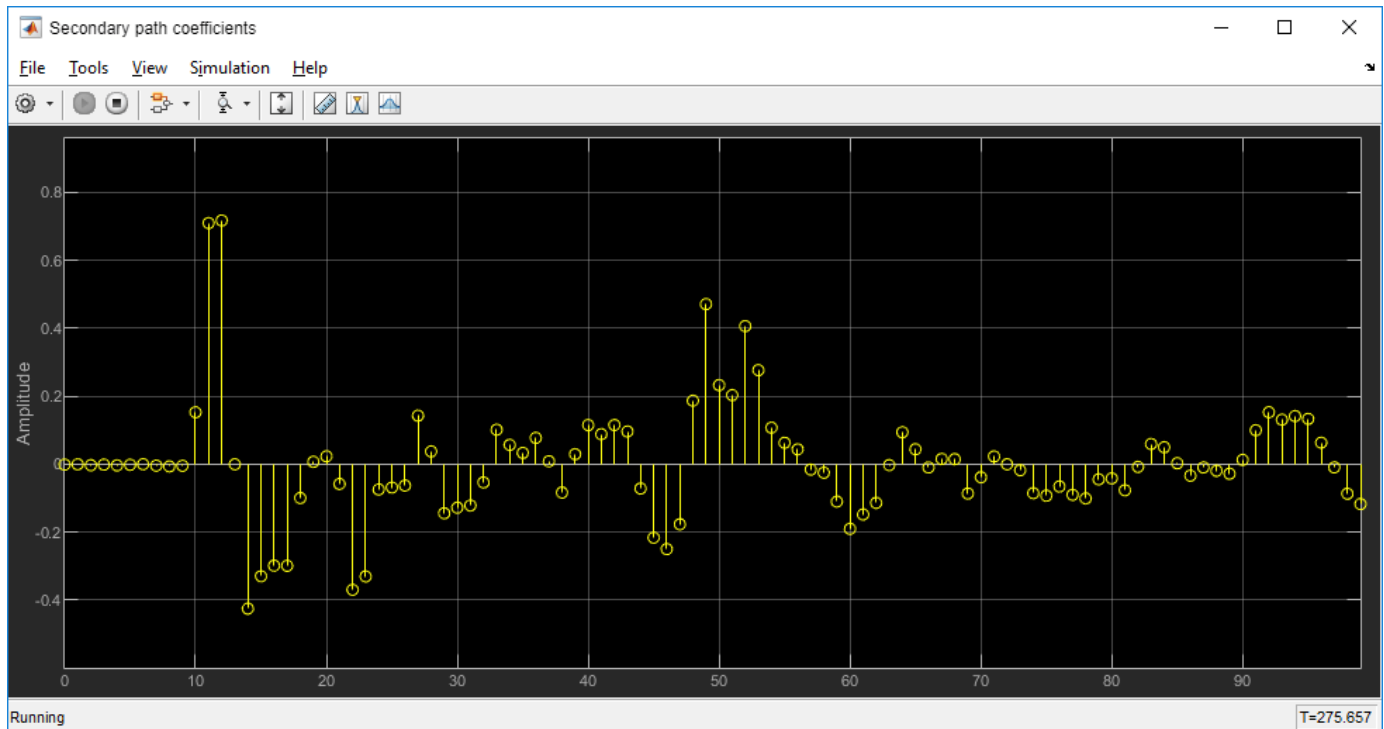


Latency Measurements

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 two or three 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 one or two 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. <https://iopscience.iop.org/article/10.1088/1757-899X/237/1/012015>.

"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" on page 1-130

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] on page 1-500.

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] on page 1-500. 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] on page 1-500. 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] on page 1-500. 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;
dataset = fullfile(downloadFolder,"TUT-acoustic-scenes-2017");

if ~datasetExists(dataset)
    disp("Downloading TUT-acoustic-scenes-2017 (15.5 GB) ...")
    HelperDownload_TUT_acoustic_scenes_2017(dataset);
end
```

Read in the development set metadata as a table. Name the table variables `FileName`, `AcousticScene`, and `SpecificLocation`.

```
trainMetaData = readtable(fullfile(dataset,"TUT-acoustic-scenes-2017-development","meta"), ...
    Delimiter={'\t'}, ...
    ReadVariableNames=false);
trainMetaData.Properties.VariableNames = ["FileName","AcousticScene","SpecificLocation"];
head(trainMetaData)
```

```
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'}

```

```

testMetaData = readtable(fullfile(dataset,"TUT-acoustic-scenes-2017-evaluation","meta"), ...
    Delimiter={'\t'}, ...
    ReadVariableNames=false);
testMetaData.Properties.VariableNames = ["FileName","AcousticScene","SpecificLocation"];
head(testMetaData)

```

```

ans=8x3 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(testMetaData.SpecificLocation,trainMetaData.SpecificLocation);
disp("Number of specific recording locations in both train and test sets = " + 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.

```

trainFilePaths = fullfile(dataset,"TUT-acoustic-scenes-2017-development",trainMetaData.FileName);
testFilePaths = fullfile(dataset,"TUT-acoustic-scenes-2017-evaluation",testMetaData.FileName);

```

There may be files listed in the metadata that are not present in the data set. Remove the filepaths and acoustic scene labels that correspond to the missing files.

```

ads = audioDatastore(dataset,IncludeSubfolders=true);
allFiles = ads.Files;

trainIdxToRemove = ~ismember(trainFilePaths,allFiles);
trainFilePaths(trainIdxToRemove) = [];
trainLabels = categorical(trainMetaData.AcousticScene);
trainLabels(trainIdxToRemove) = [];

```

```
testIdxToRemove = ~ismember(testFilePaths,allFiles);  
testFilePaths(testIdxToRemove) = [];  
testLabels = categorical(testMetaData.AcousticScene);  
testLabels(testIdxToRemove) = [];
```

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(trainFilePaths, ...  
    Labels=trainLabels, ...  
    IncludeSubfolders=true);  
display(countEachLabel(adsTrain))
```

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

```
adsTest = audioDatastore(testFilePaths, ...  
    Labels=categorical(testMetaData.AcousticScene), ...  
    IncludeSubfolders=true);  
display(countEachLabel(adsTest))
```

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 `speedupExample` to `true` to reduce the data set.

```
speedupExample = ;
if speedupExample
    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)

disp("Acoustic scene = " + string(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] on page 1-500. 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] on page 1-500 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),"n");
[dataBufferedSide,~] = buffer(dataMidSide(:,2),round(segmentLength*fs),round(segmentOverlap*fs),"n");
```

```
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] on page 1-500.

```
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.

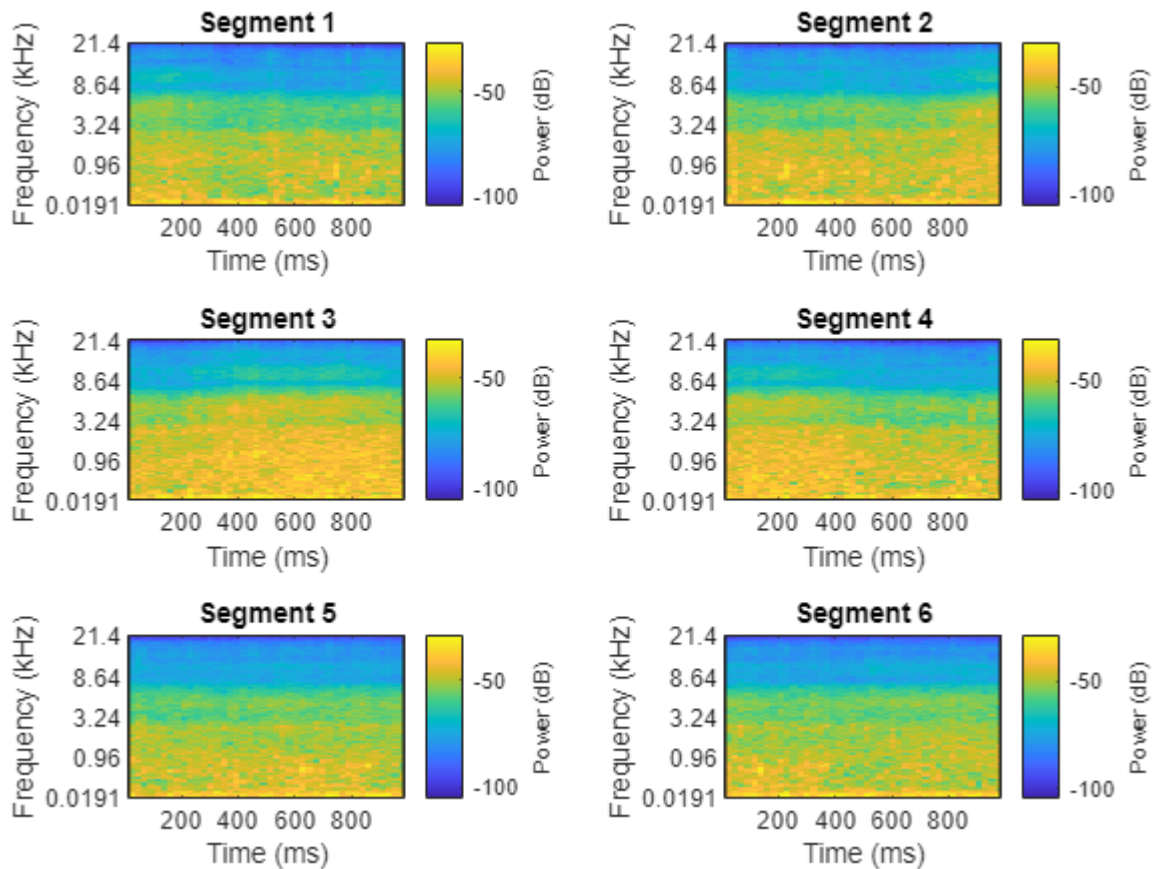
```
spec = log10(spec+eps);
```

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.

```
 tiledlayout(3,2)
 for channel = 1:2:11
     nexttile
     melSpectrogram(dataBuffered(:,channel),fs, ...
         Window=hamming(windowLength,"periodic"), ...
         OverlapLength=samplesOverlap, ...
         FFTLength=fftLength, ...
         NumBands=numBands);
     title("Segment " + ceil(channel/2))
 end
```



The helper function `HelperSegmentedMelSpectrograms` on page 1-499 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.

```
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);

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: 0% complete
Evaluation 0% complete

- Pass 1 of 1: Completed in 3 min 56 sec
Evaluation completed in 3 min 56 sec

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);

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1 min 26 sec
Evaluation completed in 1 min 26 sec

xTest = cat(4,xTest{:});
```

Replicate the labels of the training and test sets 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(:);
yTest = repmat(adsTest.Labels,1,numSegmentsPer10seconds)';
yTest = yTest(:);
```

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:

$$\begin{aligned}\tilde{x} &= \lambda x_i + (1 - \lambda)x_j \\ \tilde{y} &= \lambda y_i + (1 - \lambda)y_j\end{aligned}$$

Mixup was reformulated by [2] on page 1-500 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 ii = 1:size(xTrain,4)

    % Find all available spectrograms with different labels.
    availableSpectrograms = find(yTrain~=yTrain(ii));

    % Randomly choose one of the available spectrograms with a different label.
    numAvailableSpectrograms = numel(availableSpectrograms);
    idx = randi([1,numAvailableSpectrograms]);

    % Mix.
    xTrainExtra(:,:,,ii) = lambda*xTrain(:,:,,ii) + (1-lambda)*xTrain(:,:,,availableSpectrograms(idx));

    % Specify the label as randomly set by lambda.
    if rand > lambda
        yTrainExtra(ii) = 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] on page 1-500 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

globalAveragePooling2dLayer

dropoutLayer(0.5)

fullyConnectedLayer(15)
softmaxLayer
classificationLayer];
```

Define `trainingOptions` (Deep Learning Toolbox) for the CNN. These options are based on [3] on page 1-500 and modified through a systematic hyperparameter optimization workflow.

```
miniBatchSize = 128;
tuneme = 128;
lr = 0.05*miniBatchSize/tuneme;
options = trainingOptions( ...
    "sgdm", ...
    Momentum=0.9, ...
    L2Regularization=0.005, ...
    ...
    MiniBatchSize=miniBatchSize, ...
    MaxEpochs=8, ...
    Shuffle="every-epoch", ...
    ...
    Plots="training-progress", ...
    Verbose=false, ...
    ...
    InitialLearnRate=lr, ...
    LearnRateSchedule="piecewise", ...
```



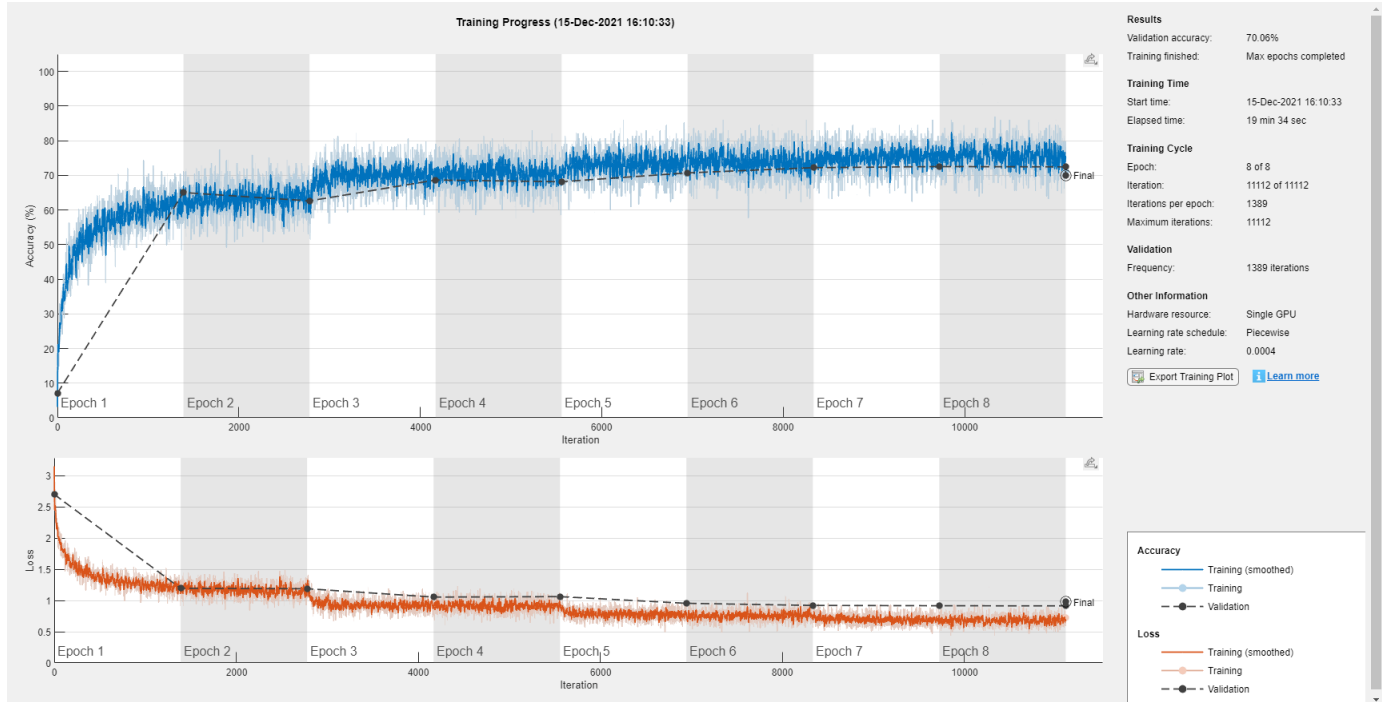
```

LearnRateDropPeriod=2, ...
LearnRateDropFactor=0.2, ...
...
ValidationData={xTest,yTest}, ...
ValidationFrequency=floor(size(xTrain,4)/miniBatchSize));

```

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),2);
    counter = counter + numSegmentsPer10seconds;
end

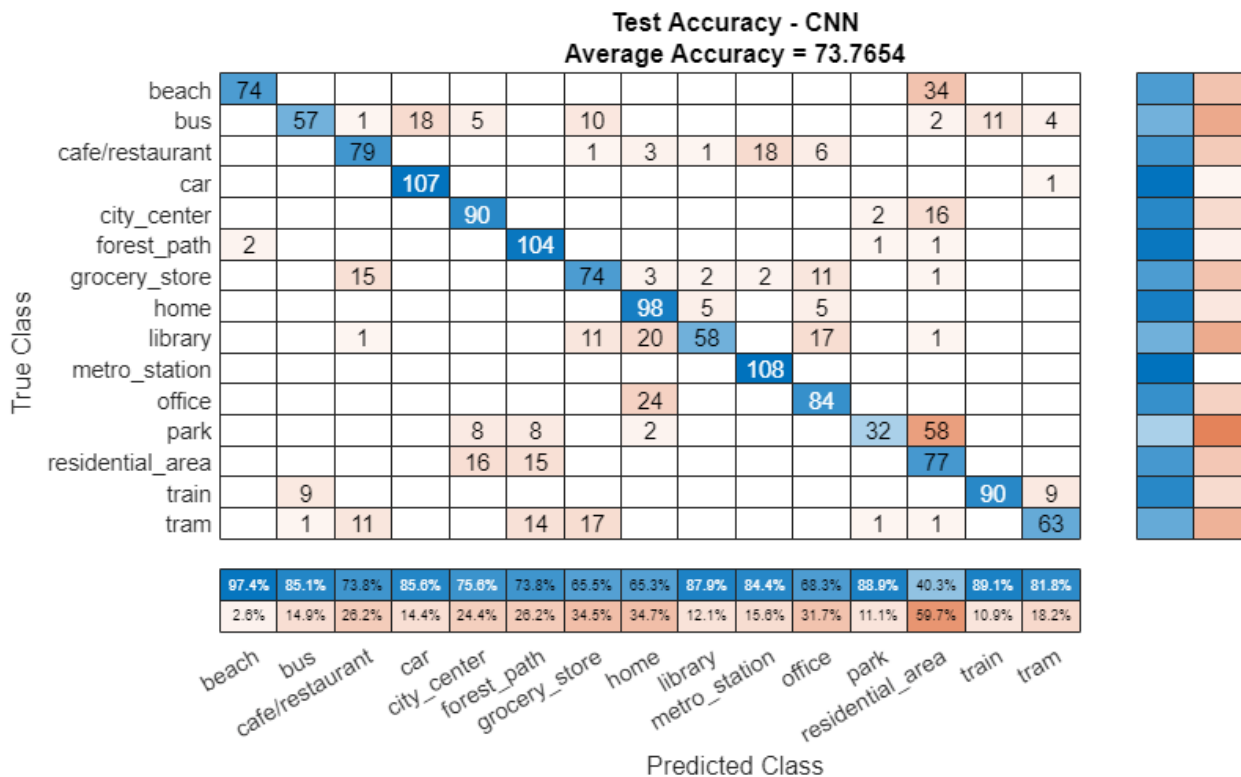
```

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.

```
figure(Units="normalized",Position=[0.2 0.2 0.5 0.5])
confusionchart(adsTest.Labels,cnnPredictedLabels, ...
    title=["Test Accuracy - CNN","Average Accuracy = " + mean(adsTest.Labels==cnnPredictedLabels)
    ColumnSummary="column-normalized",RowSummary="row-normalized");
```



Feature Extraction for Ensemble Classifier

Wavelet scattering has been shown in [4] on page 1-500 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);
disp("Number of wavelet features per 10-second clip = " + numel(featureVector));
```

Number of wavelet features per 10-second clip = 286

The helper function `HelperWaveletFeatureVector` on page 1-500 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)';
```

Define and Train Ensemble Classifier

Use `fitcensemble` to create a trained classification ensemble model (`ClassificationEnsemble`).

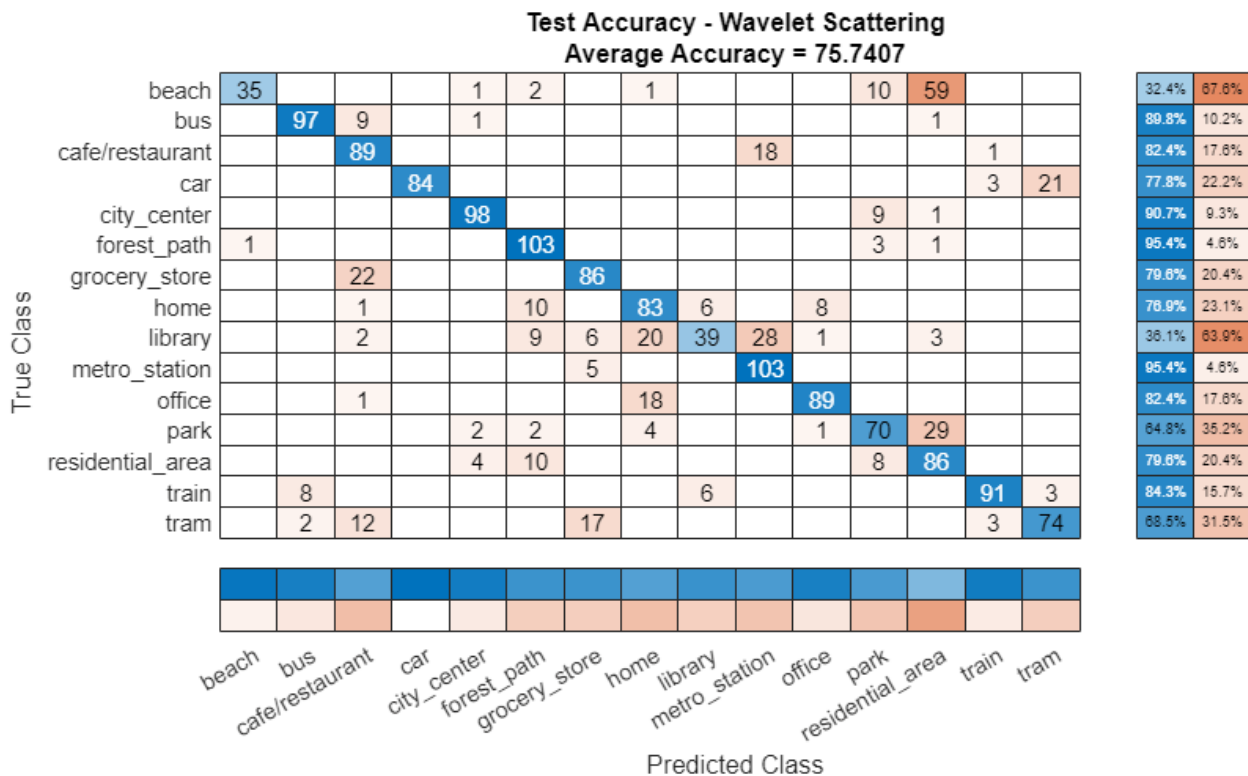
```
subspaceDimension = min(150,size(xTrain,2) - 1);
numLearningCycles = 30;
classificationEnsemble = fitcensemble(xTrain,adsTrain.Labels, ...
    Method="Subspace", ...
    NumLearningCycles=numLearningCycles, ...
    Learners="discriminant", ...
    NPredToSample=subspaceDimension, ...
    ClassNames=removecats(unique(adsTrain.Labels)));
```

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.

```
[waveletPredictedLabels,waveletResponses] = predict(classificationEnsemble,xTest);

figure(Units="normalized",Position=[0.2 0.2 0.5 0.5])
confusionchart(adsTest.Labels,waveletPredictedLabels, ...
    title=["Test Accuracy - Wavelet Scattering","Average Accuracy = " + mean(adsTest.Labels==waveletPredictedLabels)], ...
    ColumnSummary="column-normalized",RowSummary="row-normalized");
```



```
fprintf('Average accuracy of classifier = %0.2f\n',mean(adsTest.Labels==waveletPredictedLabels)*100);
```

Average accuracy of classifier = 75.74

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.

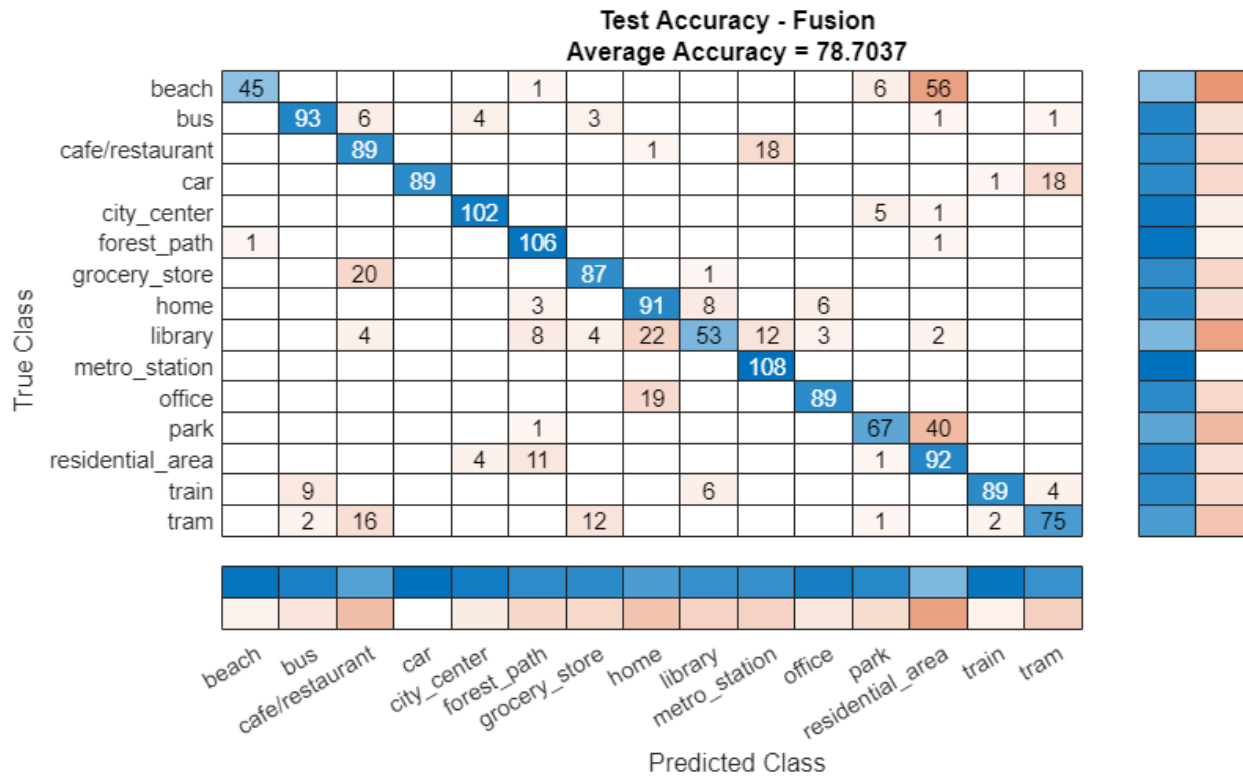
```
fused = waveletResponses.*cnnResponses;
[~,classIdx] = max(fused,[],2);
```

```
predictedLabels = classes(classIdx);
```

Evaluate Late Fusion

Call `confusionchart` to visualize the fused classification accuracy.

```
figure(Units="normalized",Position=[0.2 0.2 0.5 0.5])
confusionchart(adsTest.Labels,predictedLabels, ...
    Title=["Test Accuracy - Fusion","Average Accuracy = " + mean(adsTest.Labels==predictedLabels)*100],
    ColumnSummary="column-normalized",RowSummary="row-normalized");
```



Supporting Functions

HelperSegmentedMelSpectrograms

```
function X = HelperSegmentedMelSpectrograms(x,fs,varargin)
```

```
% Copyright 2019-2021 The MathWorks, Inc.
```

```
p = inputParser;
```

```
addParameter(p,WindowLength=1024);
```

```
addParameter(p,HopLength=512);
```

```
addParameter(p,NumBands=128);
```

```
addParameter(p,SegmentLength=1);
```

```
addParameter(p,SegmentOverlap=0);
```

```
addParameter(p,FFTLength=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, ...
```

```
        FFTLength=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
```

HelperWaveletFeatureExtractor

```
function features = HelperWaveletFeatureVector(x,sf)
% Copyright 2019-2021 The MathWorks, Inc.
x = mean(x,2);
features = featureMatrix(sf,x,Transform="log");
features = mean(features,2);
end
```

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

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 download and use a pretrained network by opening this example in MATLAB® and running the **Spot Keyword with Pretrained Network** section.

Spot Keyword with Pretrained Network

Before going into the training process in detail, you will download and 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)
```

Download and load the pretrained network, the mean (M) and standard deviation (S) vectors used for feature normalization, as well as 2 audio files used for validating the network later in the example.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","KeywordSpotting.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
netFolder = fullfile(dataFolder,"KeywordSpotting");
load(fullfile(netFolder,"KWSNet.mat"));
```

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);
```

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(KWSNet, features.');
```

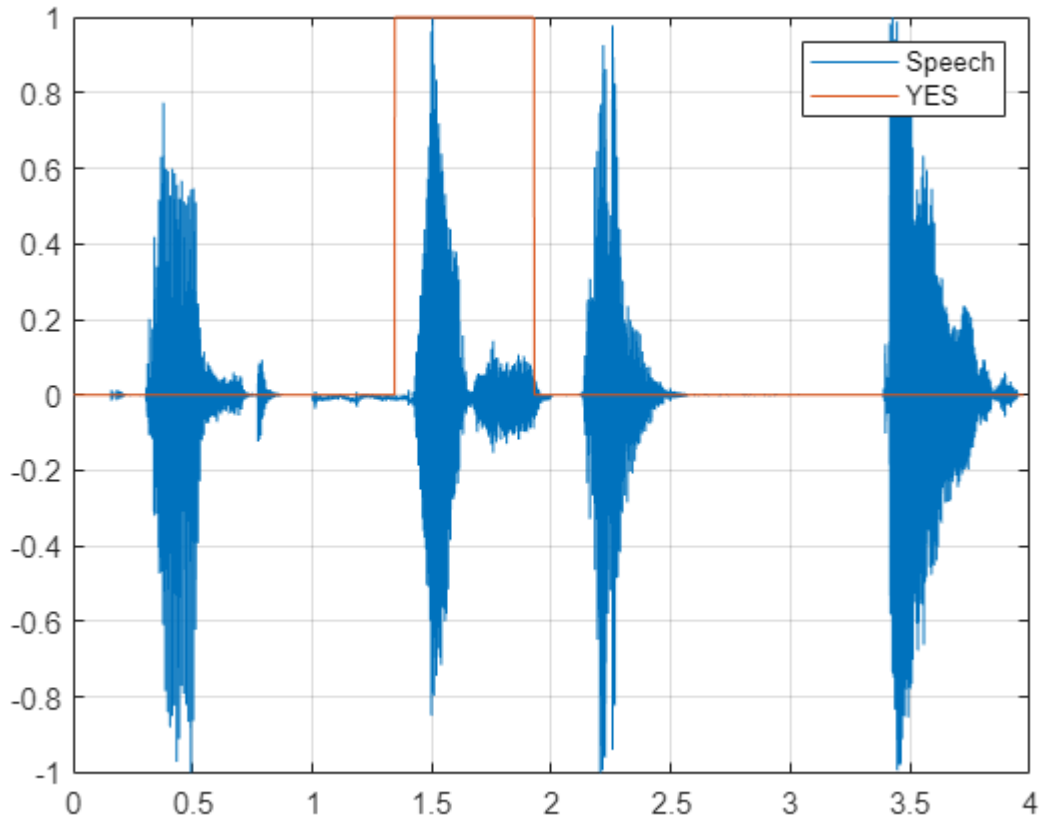
Each sample in the mask corresponds to 128 samples from the speech signal (`windowLength - overlapLength`).

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, ...  
TimeSpanOverrunAction="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(KWSNet,featureMatrix.');
```

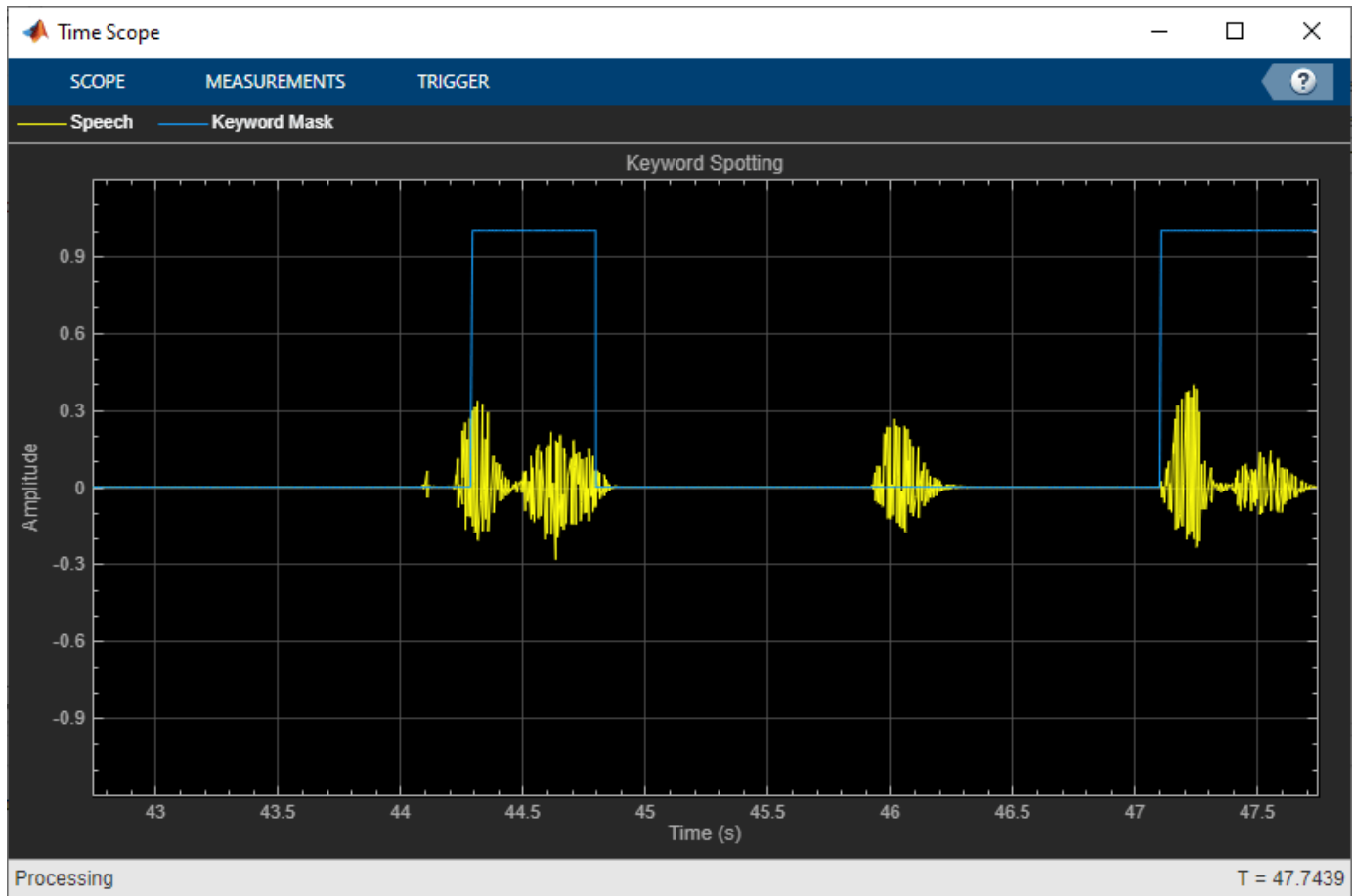
```
        v = double(v) - 1;  
        v = repmat(v,hopLength,1);  
        v = v(:);  
        v = mode(v);  
        v = repmat(v,numHopsPerUpdate*hopLength,1);
```

```
        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.

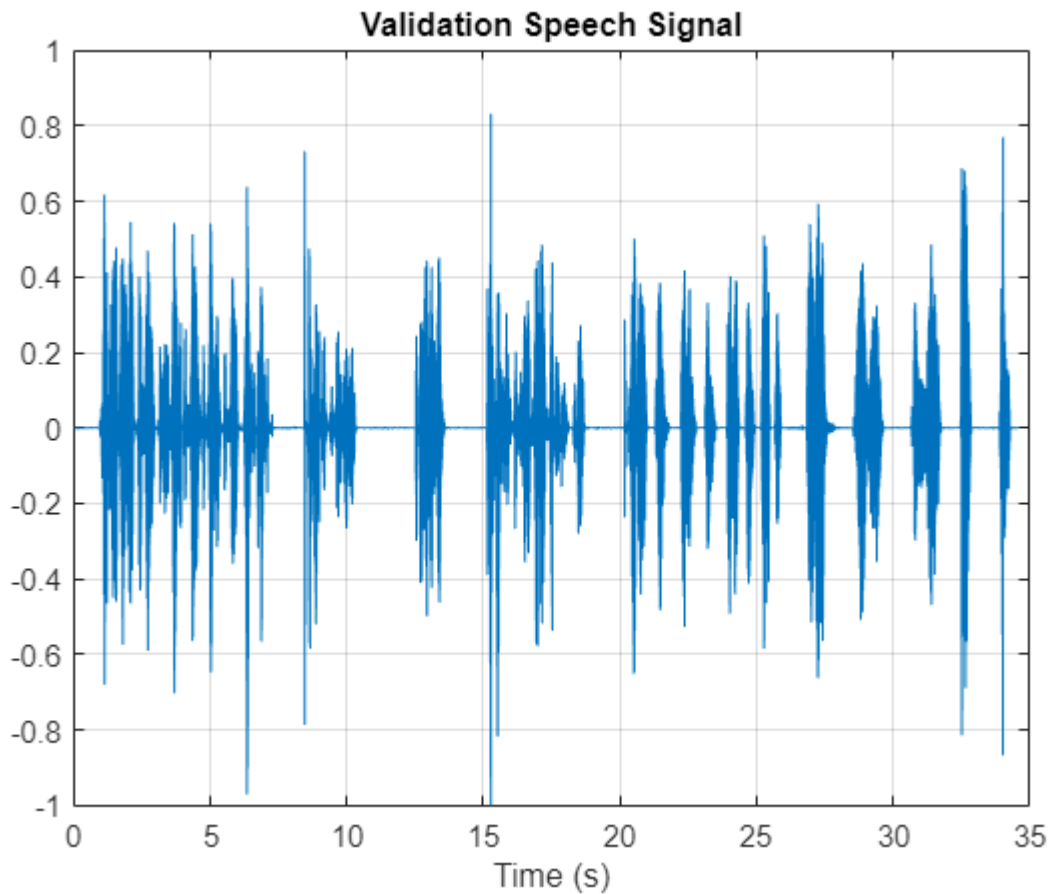
```
[audioIn,fs] = audioread(fullfile(netFolder,"KeywordSpeech-16-16-mono-34secs.flac"));
```

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

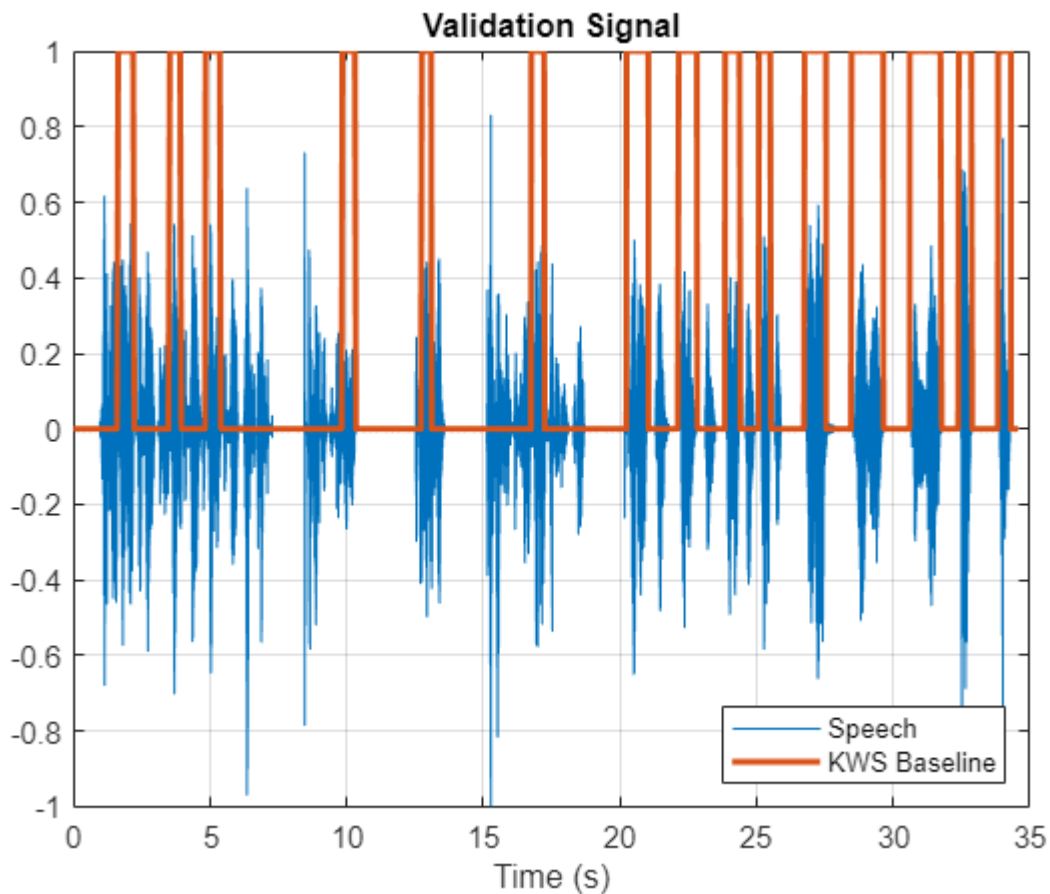
Load the KWS baseline. This baseline was obtained using `speech2text` and Signal Labeler. For a related example, see “Label Spoken Words in Audio Signals”.

```
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 1-526.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "google_speech.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
dataset = fullfile(dataFolder, "google_speech");
```

Create an `audioDatastore` that points to the data set.

```
ads = audioDatastore(dataset, 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);
adsKeyword = subset(ads, isKeyword);
adsOther = subset(ads, ~isKeyword);
```

To train the network with the entire dataset and achieve the highest possible accuracy, set `speedupExample` to `false`. To run this example quickly, set `speedupExample` to `true`.

```
speedupExample =  ;
if speedupExample
    % Reduce the dataset by a factor of 20
    adsKeyword = splitEachLabel(adsKeyword, round(numel(adsKeyword.Files)/20));
    numUniqueLabels = numel(unique(adsOther.Labels));
    adsOther = splitEachLabel(adsOther, round(numel(adsOther.Files)/numUniqueLabels/20));
end
```

Get a breakdown of the word distribution in each datastore. Shuffle the `adsOther` datastore so that consecutive reads return different words.

```
countEachLabel(adsKeyword)
```

```
ans=1x2 table
  Label    Count
  _____  _____
    yes     2377
```

```
countEachLabel(adsOther)
```

```
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
    :
```

```
adsOther = shuffle(adsOther);
```

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(adsKeyword);  
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.

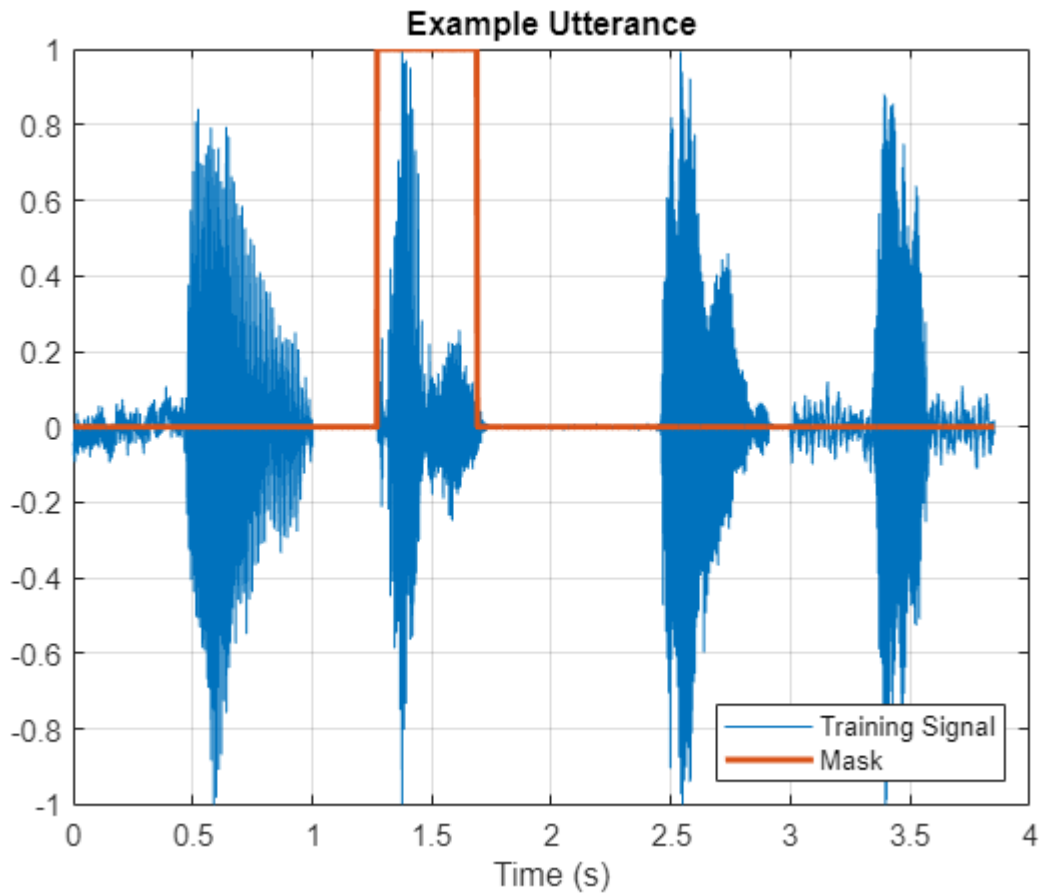
```
keywordLocation = randi([1,numWords+1]);
```

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(adsOther);  
        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
    478    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

    subadsKeyword = partition(adsKeyword,numPartitions,ii);
    subadsOther = partition(adsOther,numPartitions,ii);

    count = 1;
    localFeatures = cell(length(subadsKeyword.Files),1);
    localMasks = cell(length(subadsKeyword.Files),1);

    while hasdata(subadsKeyword)

        % Create a training sentence
        [sentence,mask] = synthesiseSentence(subadsKeyword,subadsOther,fs>windowLength);

        % Compute mfcc features
        featureMatrix = extract(afe, sentence);
        featureMatrix(~isfinite(featureMatrix)) = 0;

        % Create mask
```

```

        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

Analyzing and transferring files to the workers ...done.

disp("Training feature extraction took " + toc + " seconds.")

Training feature extraction took 41.0509 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{:});
if speedupExample
    load(fullfile(netFolder,"keywordNetNoAugmentation.mat"),"keywordNetNoAugmentation","M","S");
else
    M = mean(featuresMatrix);
    S = std(featuresMatrix);
end
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.

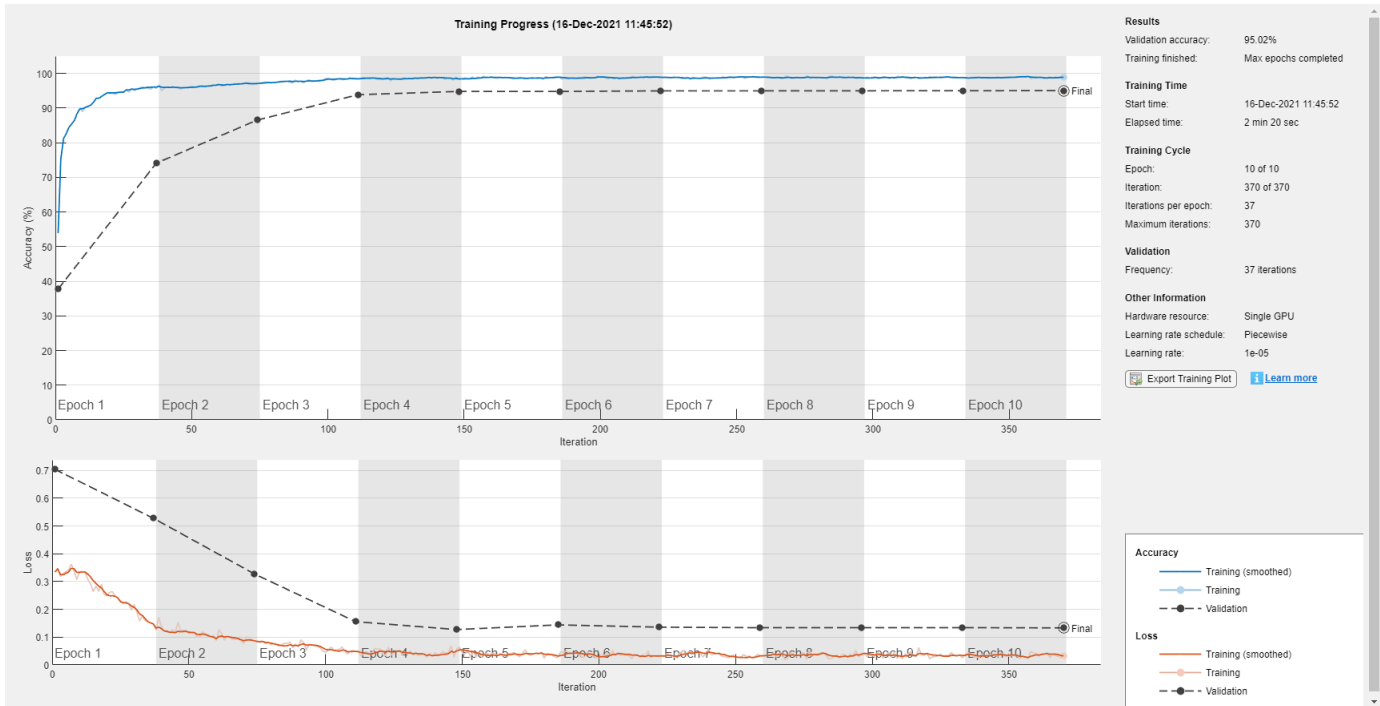
This example uses the adaptive moment estimation (ADAM) solver. ADAM performs better with recurrent neural networks (RNNs) like LSTMs than the default stochastic gradient descent with momentum (SGDM) solver.

```
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 speedupExample
    load(fullfile(netFolder,"keywordNetNoAugmentation.mat"),"keywordNetNoAugmentation","M","S");
end
```

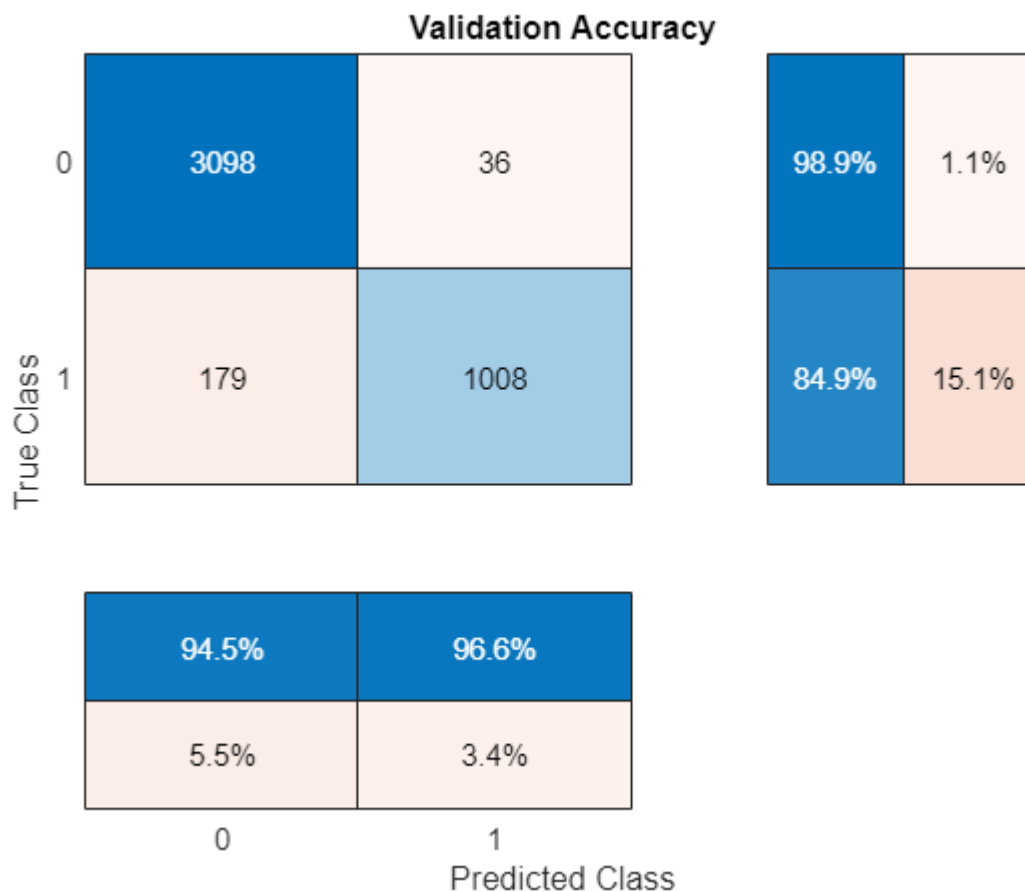
Check Network Accuracy for Noise-Free Validation Signal

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
confusionchart(BaselineV,v, ...
    Title="Validation Accuracy", ...
    ColumnSummary="column-normalized",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.

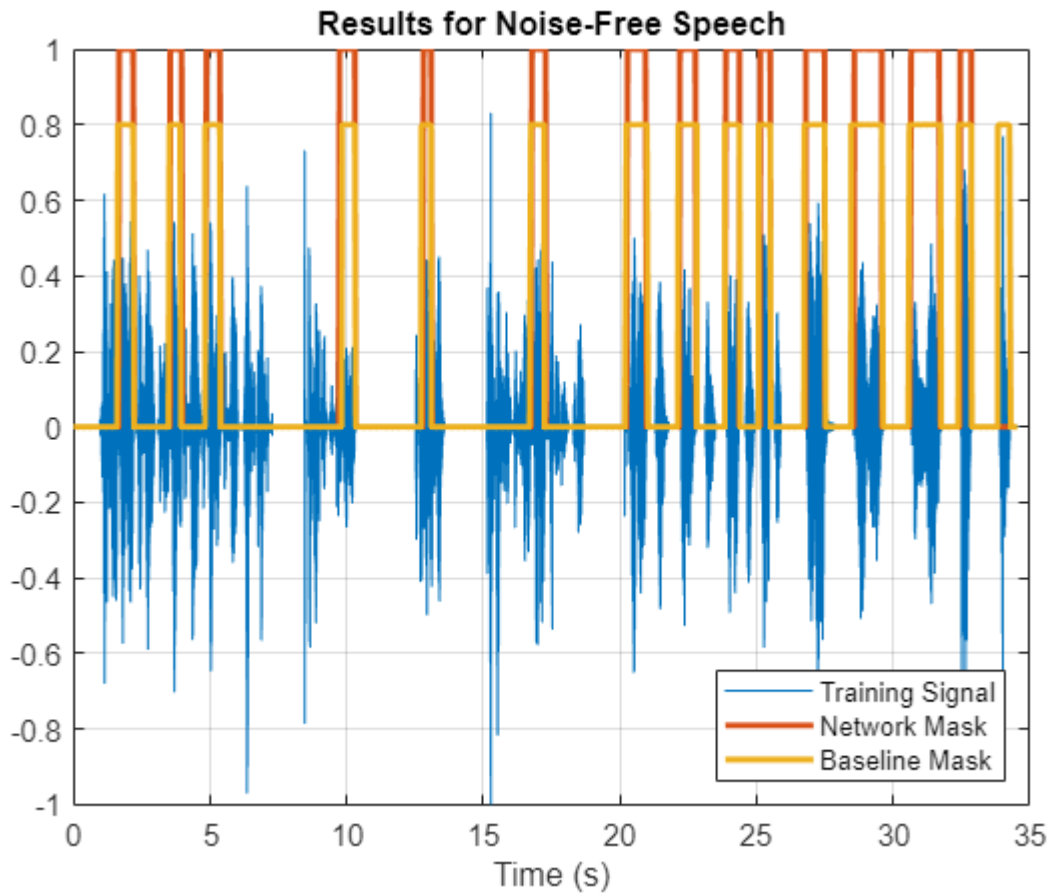
```
sound(audioIn(logical(v)),fs)
```

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

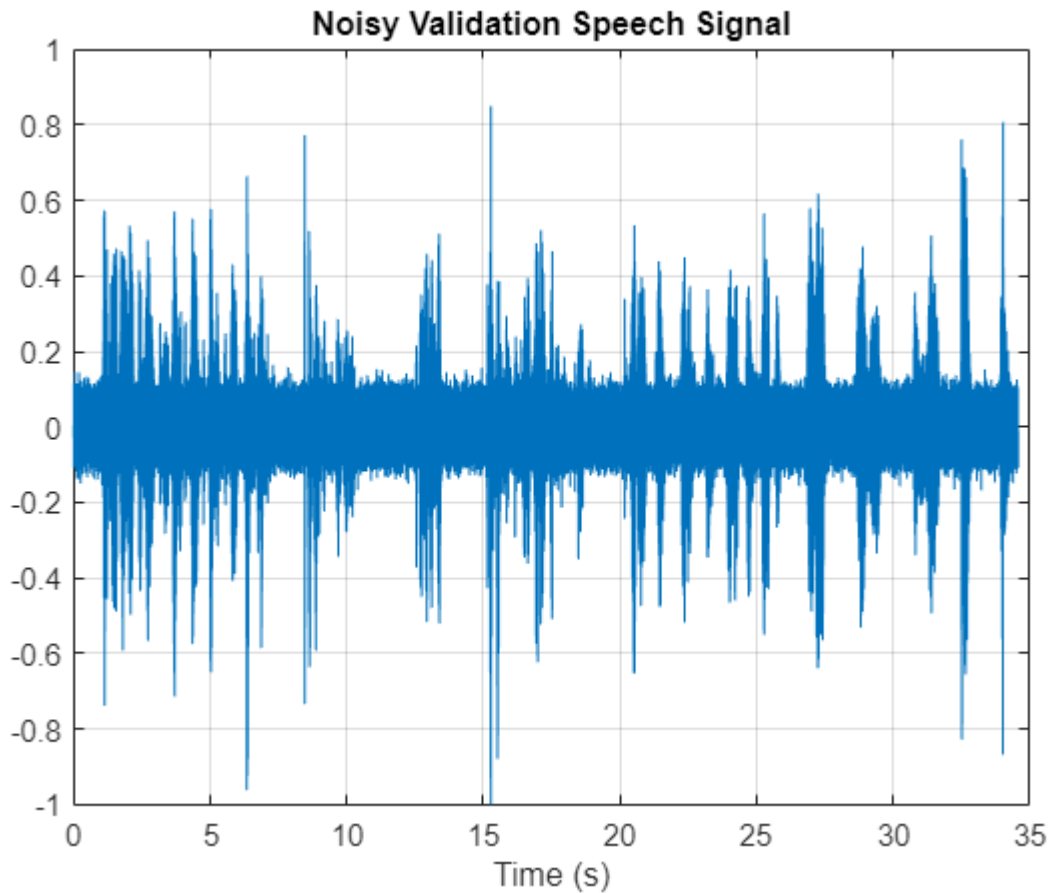
You will now check the network accuracy for a noisy speech signal. The noisy signal was obtained by corrupting the clean validation signal by additive white Gaussian noise.

Load the noisy signal.

```
[audioInNoisy,fs] = audioread(fullfile(netFolder,"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.

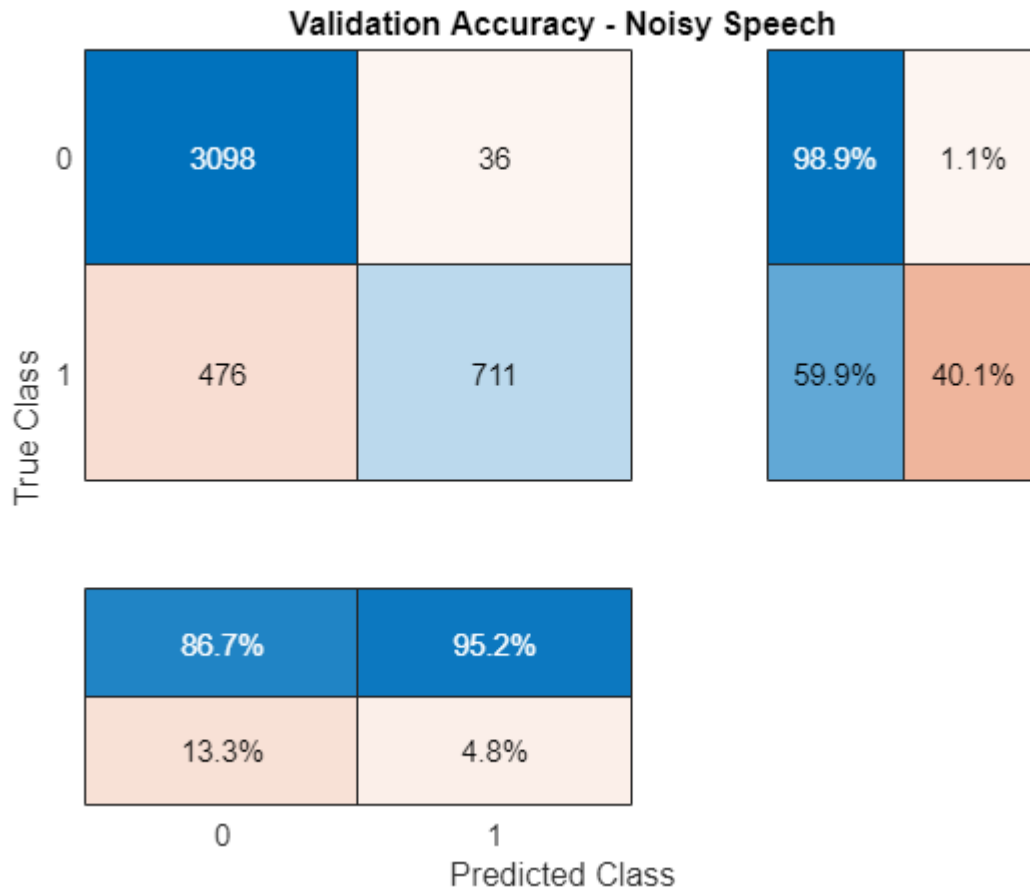
```
featureMatrixV = extract(afe, audioInNoisy);
featureMatrixV(~isfinite(featureMatrixV)) = 0;
FeaturesValidationNoisy = (featureMatrixV - M)./S;
```

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
confusionchart(BaselineV,v, ...
    Title="Validation Accuracy - Noisy Speech", ...
    ColumnSummary="column-normalized",RowSummary="row-normalized");
```

Convert the network output from categorical to double.

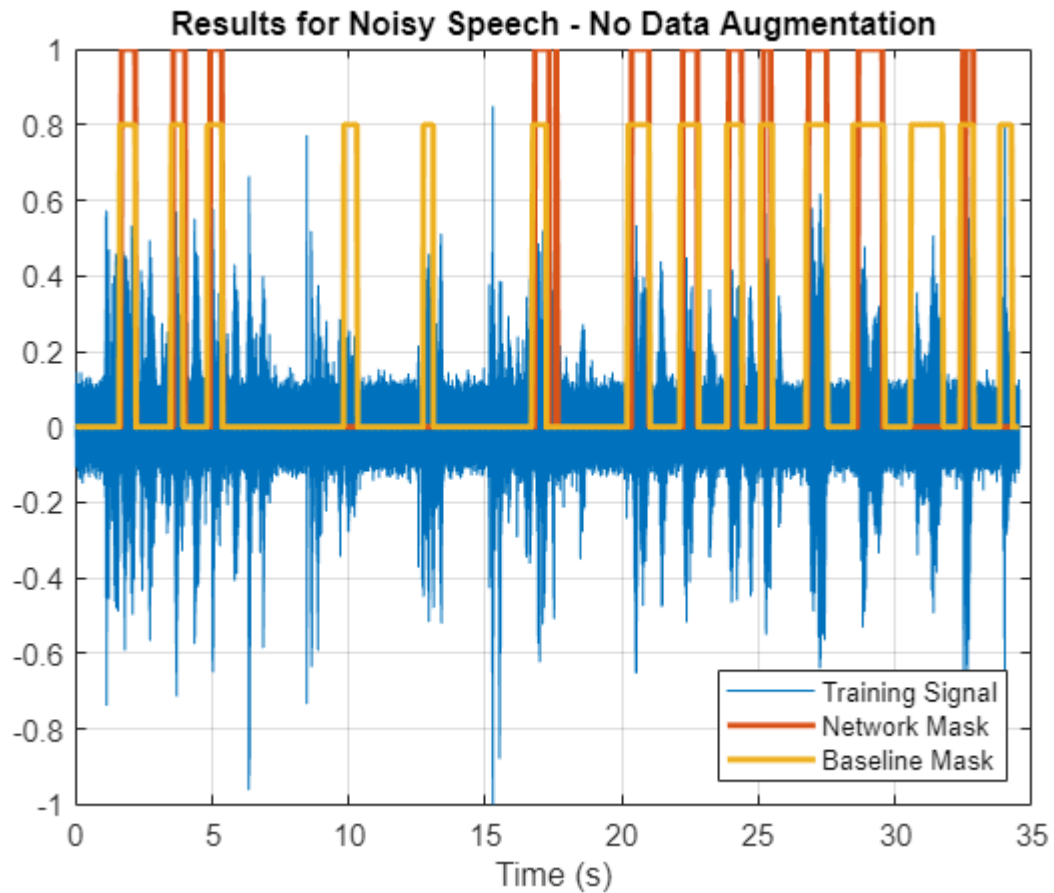
```
v = double(v) - 1;
v = repmat(v,hopLength,1);
v = v(:);
```

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 augmenter does not modify your input signal.

As an example, pass an audio signal to the augmenter.

```
reset(adsKeyword)
x = read(adsKeyword);
data = augment(ada,x,fs)

data=1x2 table           AugmentationInfo
      Audio
```

```
{16000x1 double}      1x1 struct
```

Inspect the `AugmentationInfo` variable in `data` to verify how the signal was modified.

```
data.AugmentationInfo
```

```
ans = struct with fields:
    SNR: 0.3410
```

Reset the datastores.

```
reset(adsKeyword)
reset(adsOther)
```

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

    subadsKeyword = partition(adsKeyword,numPartitions,ii);
    subadsOther = partition(adsOther,numPartitions,ii);

    count = 1;
    localFeatures = cell(length(subadsKeyword.Files),1);
    localMasks = cell(length(subadsKeyword.Files),1);

    while hasdata(subadsKeyword)

        [sentence,mask] = synthesizeSentence(subadsKeyword,subadsOther,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;

        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  
disp("Training feature extraction took " + toc + " seconds.")
```

Training feature extraction took 35.6612 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{:});  
if speedupExample  
    load(fullfile(netFolder,"KWSNet.mat"),"KWSNet","M","S");  
else  
    M = mean(featuresMatrix);  
    S = std(featuresMatrix);  
end  
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;
```

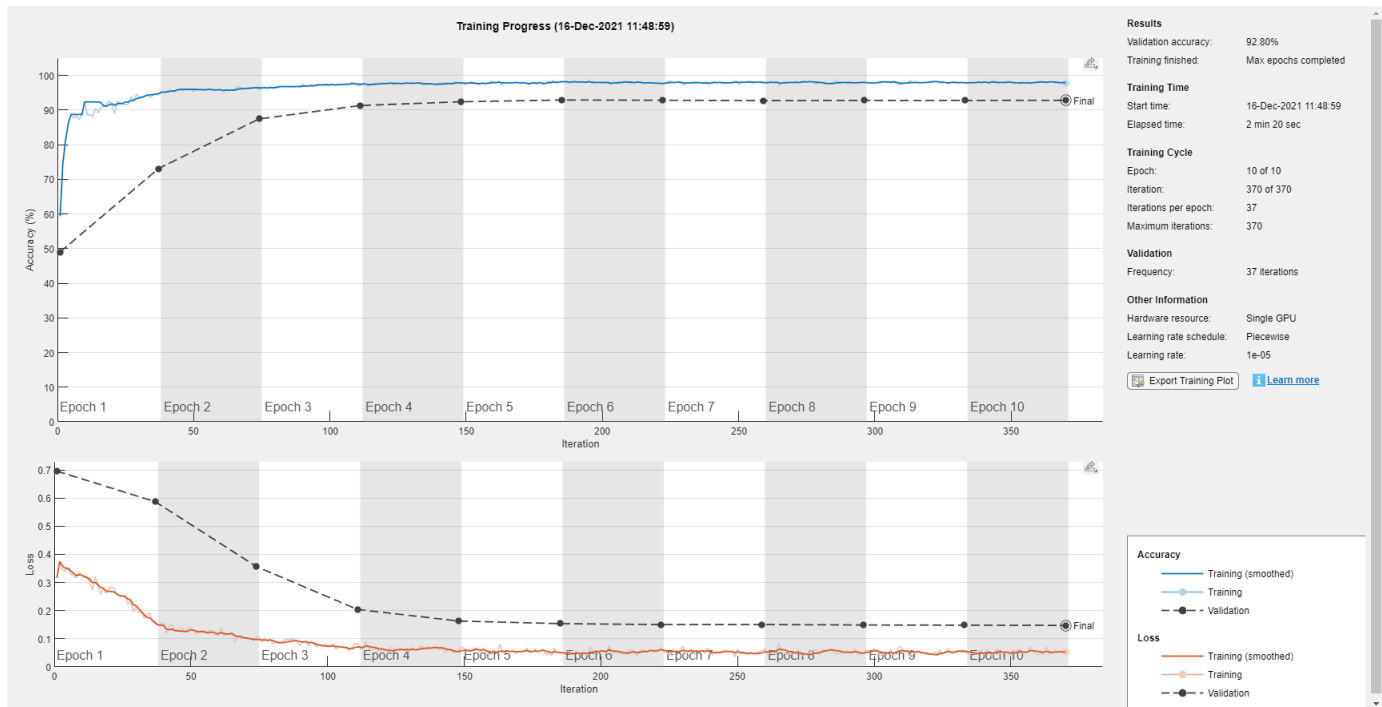
Retrain Network with Augmented Dataset

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);
```



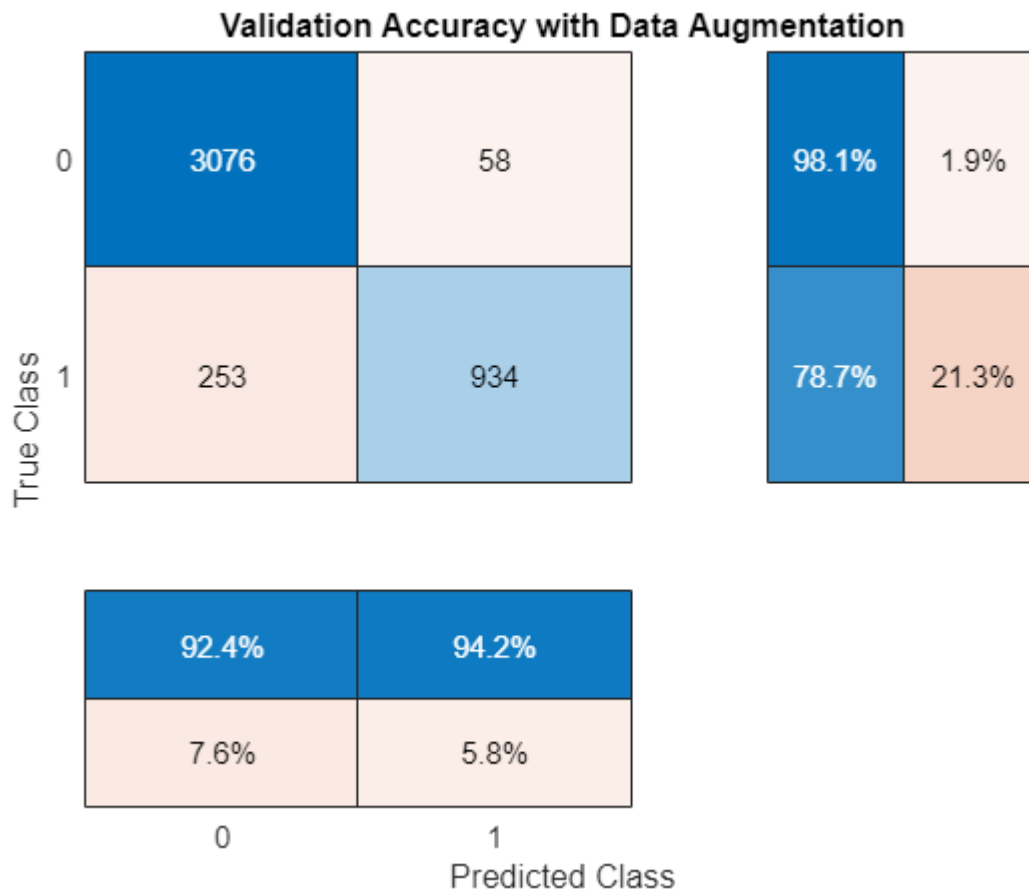
```
if speedupExample
    load(fullfile(netFolder, "KWSNet.mat"));
end
```

Verify the network accuracy on the validation signal.

```
v = classify(KWSNet, FeaturesValidationNoisy.');
```

Compare the estimated and expected KWS masks.

```
figure
confusionchart(BaselineV, v, ...
    Title="Validation Accuracy with Data Augmentation", ...
    ColumnSummary="column-normalized", RowSummary="row-normalized");
```



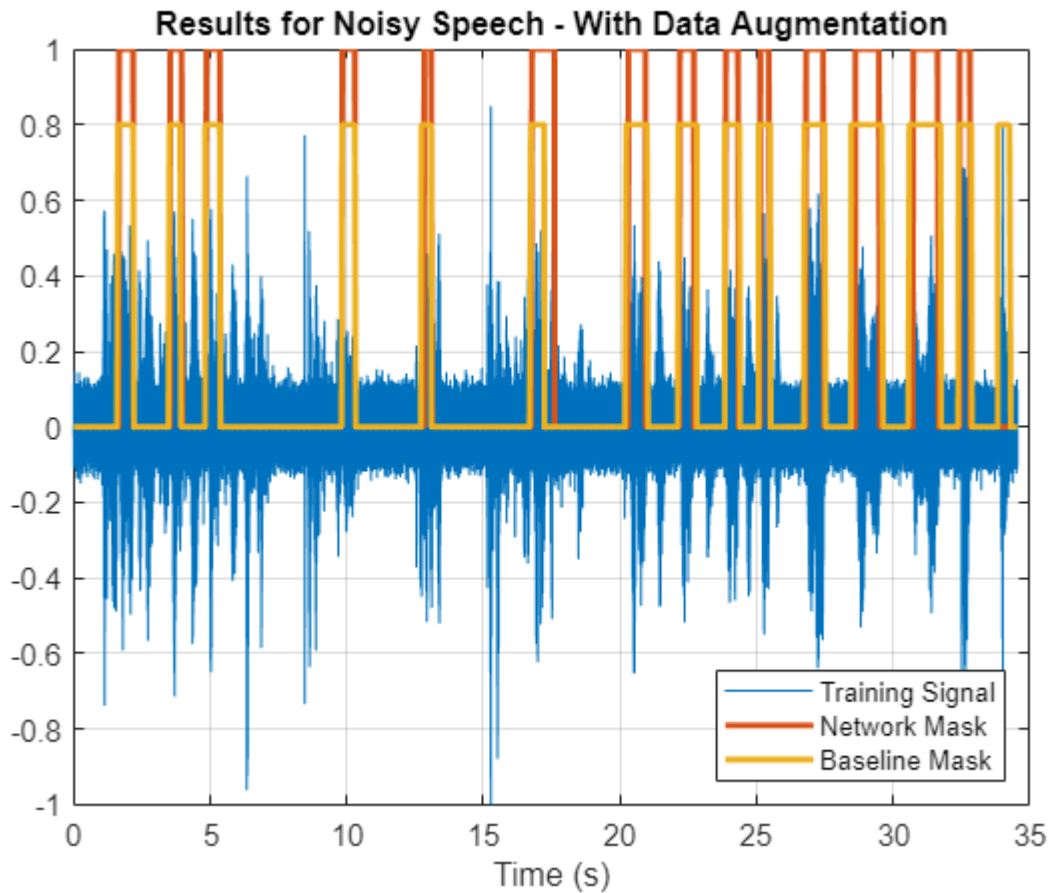
Listen to the identified keyword regions.

```
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")
```



Supporting Functions

Synthesize Sentence

```
function [sentence,mask] = synthesizeSentence(adsKeyword,adsOther,fs,minlength)
```

```
% Read one keyword
```

```
keyword = read(adsKeyword);  
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(adsOther);
        other = other./max(abs(other));
        sentence = [sentence;other];
        mask = [mask;zeros(size(other))];
    end
end
end
```

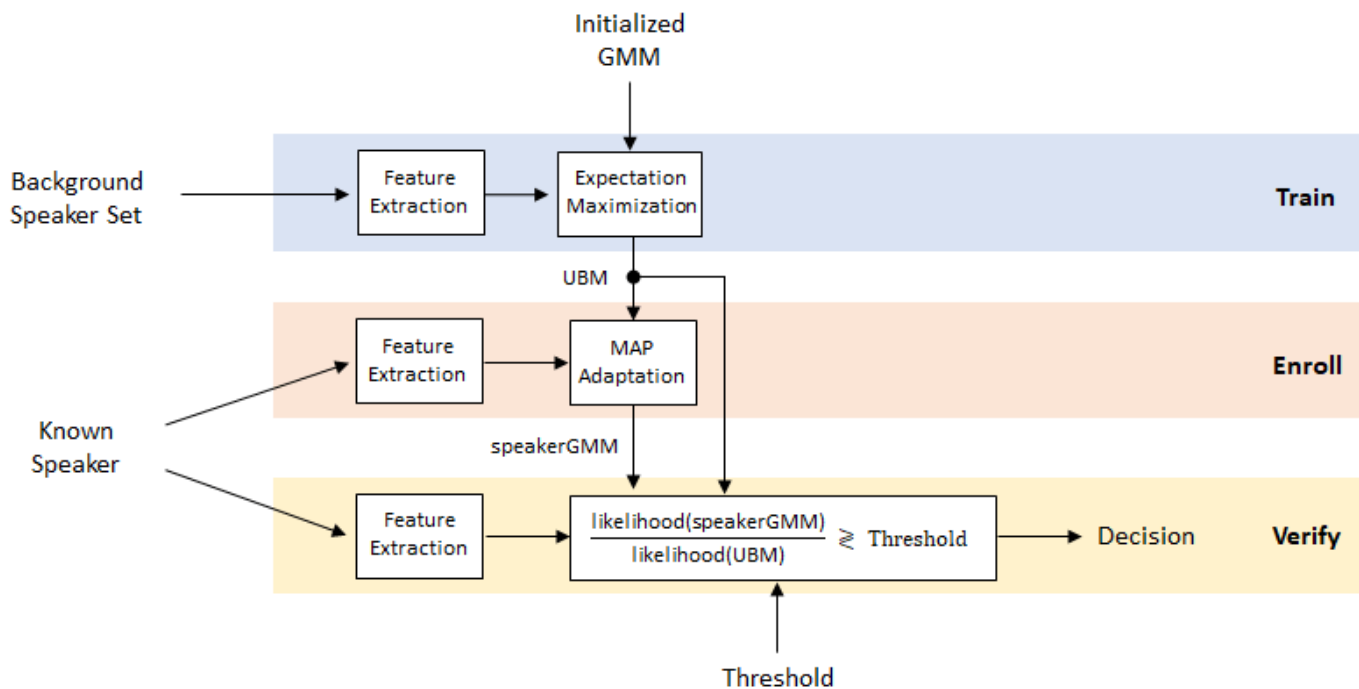
References

[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.

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 1-544.

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 1-534.

```
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 = 1  ;
reset(adsTest)
while hasdata(adsTest)
    disp("Identity to confirm: " + enrollLabel)
    [audioData,adsInfo] = read(adsTest);

    disp(" | Speaker identity: " + string(adsInfo.Label))
end

```

```

verificationStatus = helperVerify(audioData,afe,normFactors,speakerGMM,ubm,threshold);
if verificationStatus
    disp(" | Confirmed.");
else
    disp(" | Imposter!");
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 1-544. Download and extract the data set.

```

downloadFolder = matlab.internal.examples.downloadSupportFile("audio","google_speech.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)

```

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(dataFolder,IncludesSubfolders=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;
```

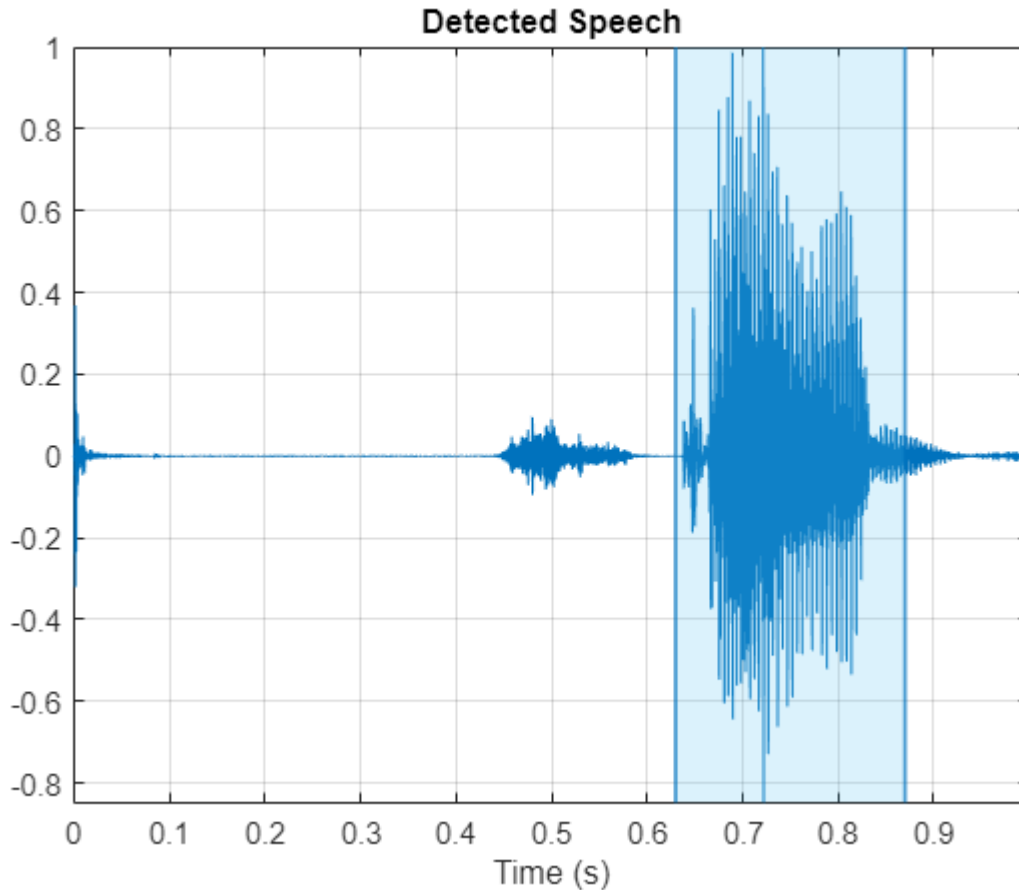
```
afe = audioFeatureExtractor( ...
    SampleRate=fs, ...
    Window=hann(windowSamples,"periodic"), ...
    OverlapLength=overlapSamples, ...
    ...
    mfcc=true);
```

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 = 21
```

```
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 1-542.

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 1-530 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 iii = 1:numel(adsPart.Files)  
        audioData = read(adsPart);  
        featuresPart{iii} = helperFeatureExtraction(audioData,afe,[]);  
    end  
    featuresAll = [featuresAll,featuresPart];  
end
```

Analyzing and transferring files to the workers ...done.


```
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 1-544 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 μ and σ 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);
    disp("Iteration " + iter + ", Log-likelihood = " + round(L,3))
```

```

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 = -538.56
Iteration 3, Log-likelihood = -522.685
Iteration 4, Log-likelihood = -517.473
Iteration 5, Log-likelihood = -514.866
Iteration 6, Log-likelihood = -513.083
Iteration 7, Log-likelihood = -511.658
Iteration 8, Log-likelihood = -510.602
Iteration 9, Log-likelihood = -509.802
Iteration 10, Log-likelihood = -509.148
Iteration 11, Log-likelihood = -508.543
Iteration 12, Log-likelihood = -508.046

disp("UBM training completed in " + round(toc,2) + " seconds.")

UBM training completed in 30.66 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 1-544 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
end

```



```

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
disp("Enrollment completed in " + round(toc,2) + " seconds.")

```

Enrollment completed in 0.31 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
end

```

```

end
    llr{speakerIdx} = llrPerSpeaker;
end
disp("False rejection rate computed in " + round(toc,2) + " seconds.")

```

False rejection rate computed in 0.23 seconds.

Plot the false rejection rate as a function of the threshold.

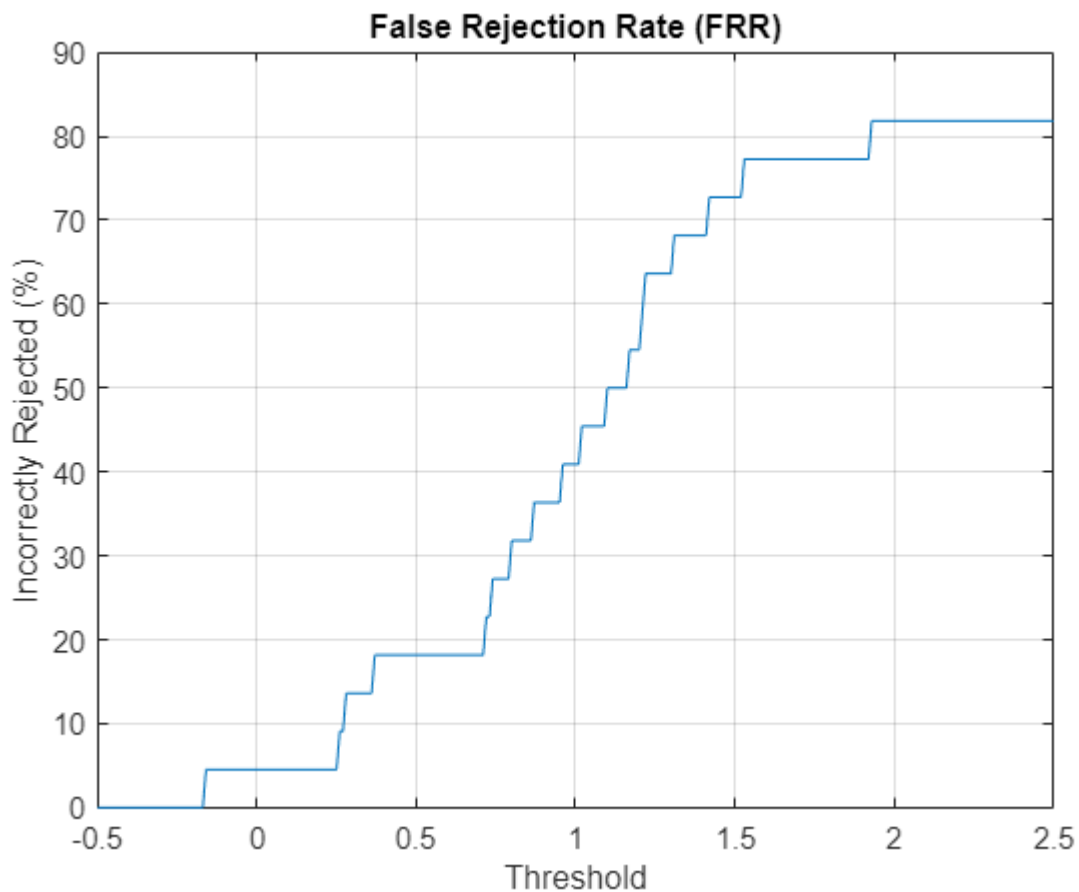
```

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);

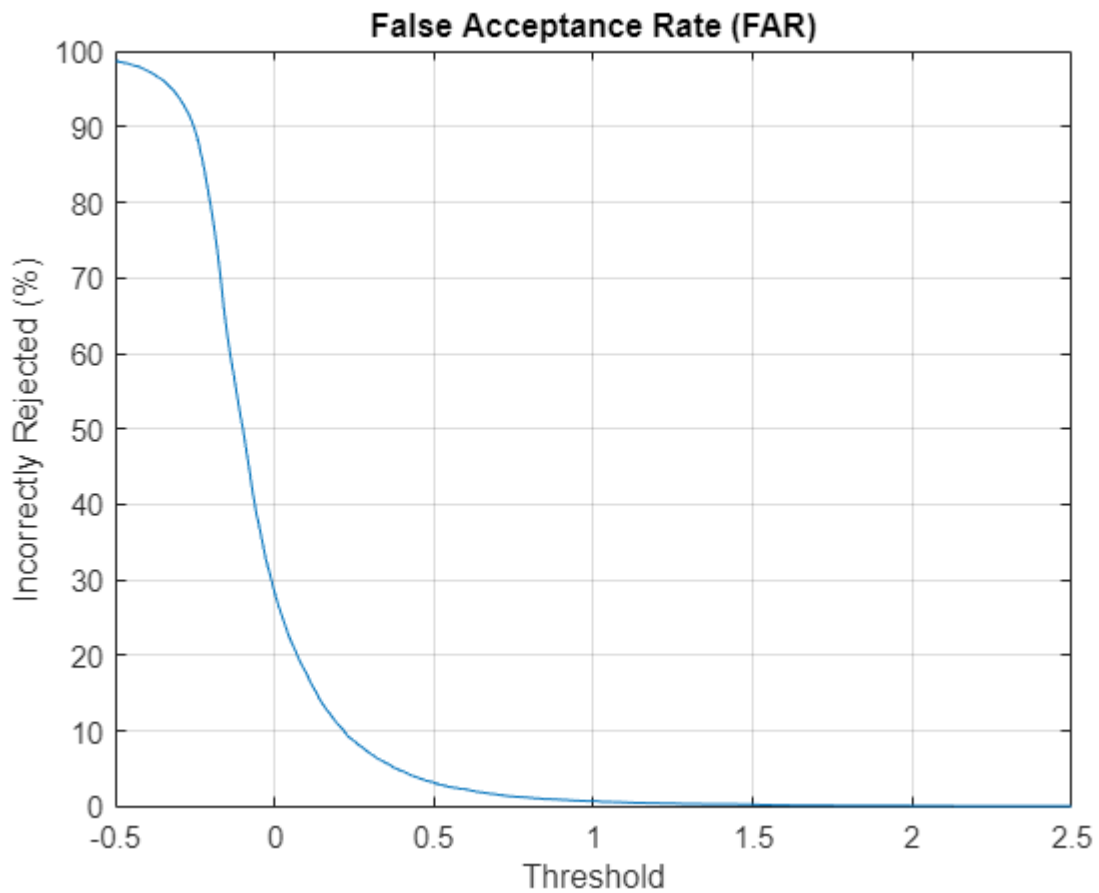
        logLikelihood = helperGMMLogLikelihood(x,ubm);
        Lubm = helperLogSumExp(logLikelihood);

        llrPerSpeaker(fileIdx) = mean(movmedian(Lspeaker - Lubm,3));
    end
    llr{speakerIdx} = llrPerSpeaker;
end
disp("FAR computed in " + round(toc,2) + " seconds.")
```

FAR computed in 22.52 seconds.

Plot the FAR as a function of the threshold.

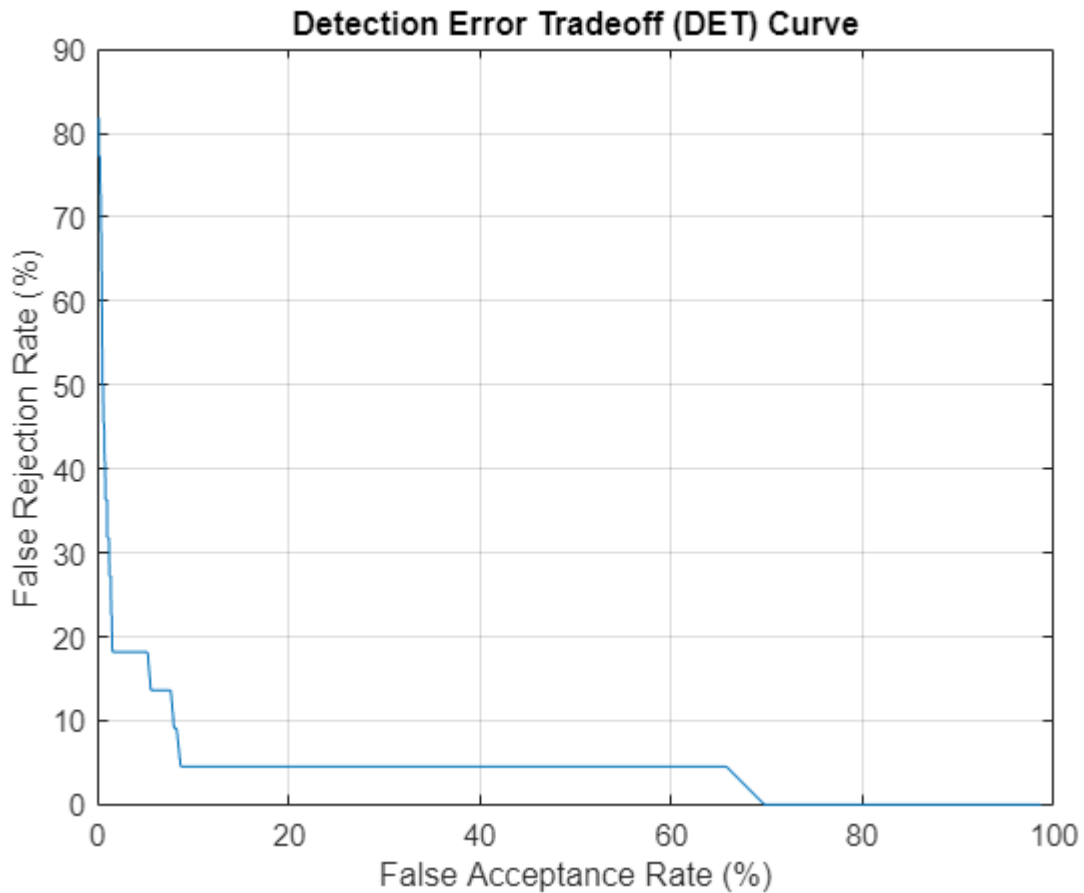
```
llr = cat(1,llr{:});
FAR = mean(llr>thresholds);
plot(thresholds,FAR*100)
title("False Acceptance Rate (FAR)")
xlabel("Threshold")
ylabel("Incorrectly Rejected (%)")
grid on
```



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.

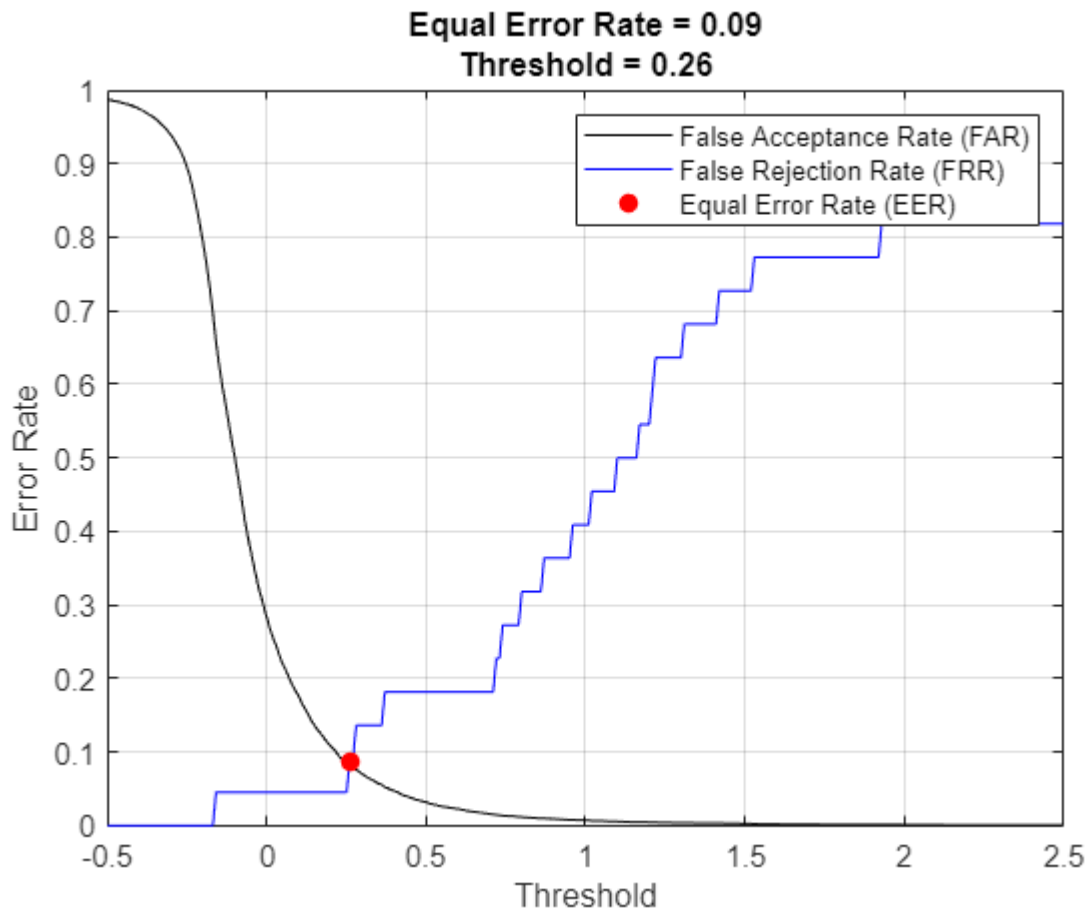
```
x1 = FAR*100;  
y1 = FRR*100;  
plot(x1,y1)  
grid on  
xlabel("False Acceptance Rate (%)")  
ylabel("False Rejection Rate (%)")  
title("Detection Error Tradeoff (DET) Curve")
```



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(["Equal Error Rate = " + round(EER,2), "Threshold = " + round(EERThreshold,2)])
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
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,x  
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)');
```

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 1-557. In this example, you apply sequential forward selection to the task of spoken digit recognition using the Free Spoken Digit Dataset [2] on page 1-557.

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 = trimOrPad(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.
```

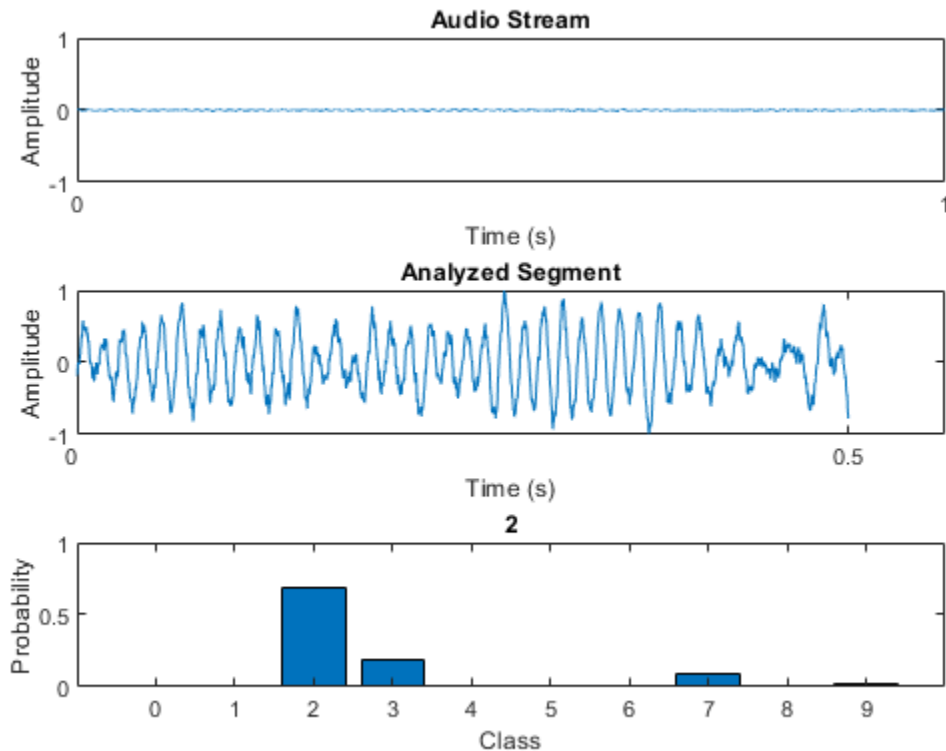
```
features = (features - normalizers.Mean) ./ normalizers.StandardDeviation;

% 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 1-557. FSDD consists of short audio files with spoken digits (0-9).

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "FSDD.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
dataset = fullfile(dataFolder, "FSDD");
```

Create an `audioDatastore` to point to the recordings. Get the sample rate of the data set.

```
ads = audioDatastore(dataset, 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, ...
    zerocrossrate=false, ...
    shortTimeEnergy=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);

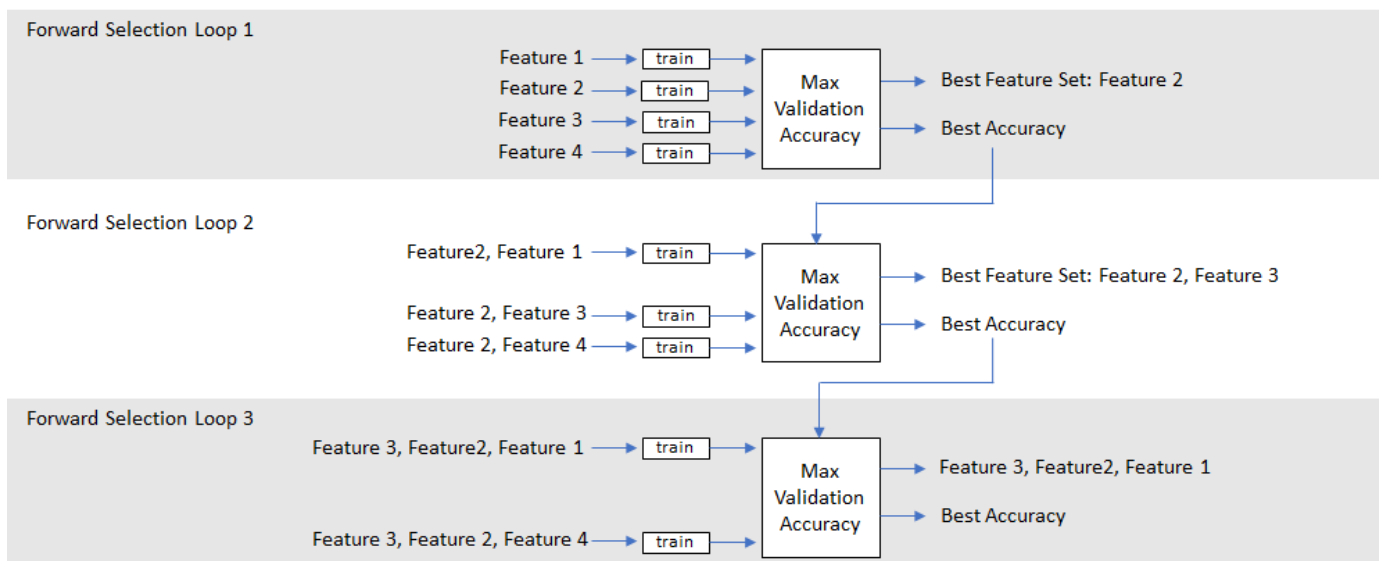
```

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 1-557.

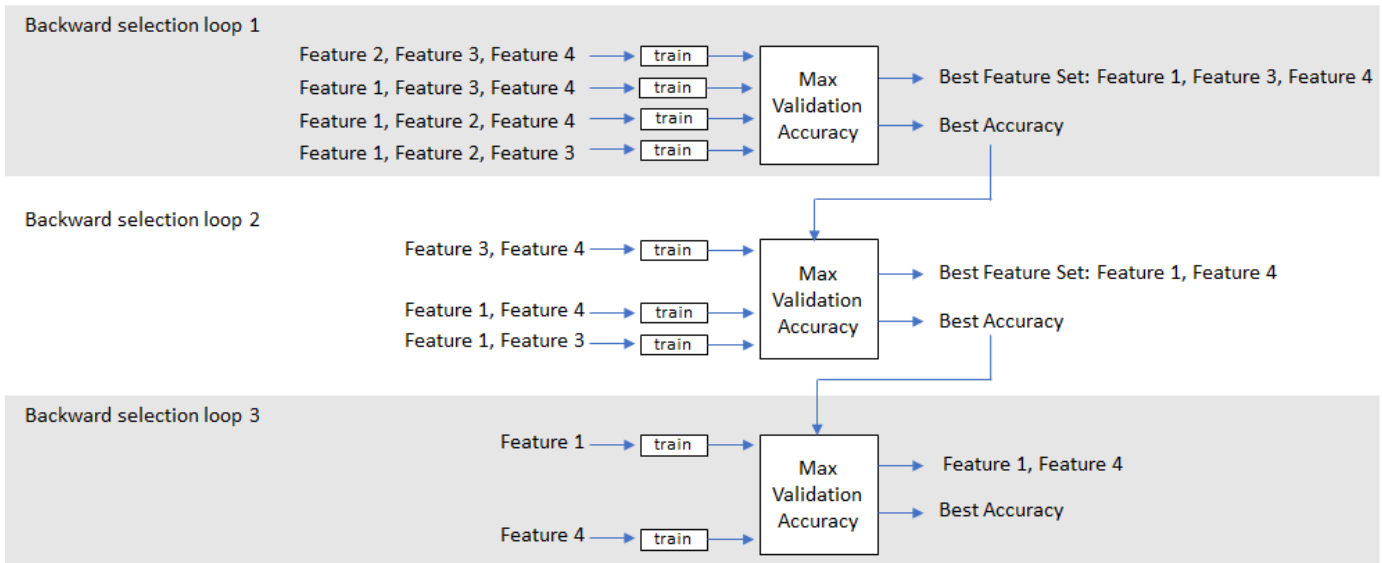
Forward Selection

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 (sequentialFeatureSelection on page 1-554, trainAndValidateNetwork on page 1-553, and trimOrPad on page 1-556) 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 =  ;
[logbook,bestFeatures,bestNet,normalizers] = sequentialFeatureSelection(adsTrain,adsValidation,a
```

The logbook output from HelperFeatureExtractor is a table containing all feature configurations tested and the corresponding validation accuracy.

logbook

logbook=62x2 table

Features	Accuracy
"mfccDelta, spectralKurtosis, spectralRolloffPoint"	98.25
"mfccDelta, spectralRolloffPoint"	97.75
"mfccDelta, spectralEntropy, spectralRolloffPoint"	97.75
"mfccDelta, spectralDecrease, spectralKurtosis, spectralRolloffPoint"	97.25
"mfccDelta, mfccDeltaDelta"	97
"mfccDelta, gtccDeltaDelta, spectralRolloffPoint"	97
"mfcc, mfccDelta, spectralKurtosis, spectralRolloffPoint"	97
"mfcc, mfccDelta"	96.75
"mfccDelta, gtccDeltaDelta, spectralKurtosis, spectralRolloffPoint"	96.75
"mfccDelta, spectralRolloffPoint, spectralSlope"	96.5
"mfccDelta"	96.25
"mfccDelta, spectralKurtosis"	96.25
"mfccDelta, spectralSpread"	96.25
"mfccDelta, spectralDecrease, spectralRolloffPoint"	96.25
"mfccDelta, spectralFlatness, spectralKurtosis, spectralRolloffPoint"	96.25
"mfccDelta, gtccDeltaDelta"	96

:

The `bestFeatures` output from `sequentialFeatureSelection` contains a struct with the optimal features set to `true`.

`bestFeatures`

```
bestFeatures = struct with fields:
    mfcc: 0
    mfccDelta: 1
    mfccDeltaDelta: 0
    gtcc: 0
    gtccDelta: 0
    gtccDeltaDelta: 0
    spectralCentroid: 0
    spectralCrest: 0
    spectralDecrease: 0
    spectralEntropy: 0
    spectralFlatness: 0
    spectralFlux: 0
    spectralKurtosis: 1
    spectralRolloffPoint: 1
    spectralSkewness: 0
    spectralSlope: 0
    spectralSpread: 0
```

You can set your `audioFeatureExtractor` using the struct.

```
set(afe,bestFeatures)
afe
```

```
afe =
```

```
audioFeatureExtractor with properties:
```

```
Properties
```

```
    Window: [240x1 double]
    OverlapLength: 160
    SampleRate: 8000
    FFTLength: []
    SpectralDescriptorInput: 'linearSpectrum'
    FeatureVectorLength: 15
```

```
Enabled Features
```

```
mfccDelta, spectralKurtosis, spectralRolloffPoint
```

```
Disabled Features
```

```
linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDeltaDelta
gtcc, gtccDelta, gtccDeltaDelta, spectralCentroid, spectralCrest, spectralDecrease
spectralEntropy, spectralFlatness, spectralFlux, spectralSkewness, spectralSlope, spectralS
pitch, harmonicRatio, zerocrossrate, shortTimeEnergy
```

To extract a feature, set the corresponding property to `true`.
For example, `obj.mfcc = true`, adds `mfcc` to the list of enabled features.

sequentialFeatureSelection 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

Train and Validate Network

```
function [trueLabels,predictedLabels,net,normalizers] = trainAndValidateNetwork(adsTrain,adsValida
% 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)trimOrPad(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)trimOrPad(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

```

Sequential Feature Selection

```

function [logbook,bestFeatures,bestNet,bestNormalizers] = sequentialFeatureSelection(adsTrain,adsValidate,afe,layers,options,direction)
%
% 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 - trainingOptions object
% direction - SFS direction, specify as 'forward' or 'backward'
%
% OUTPUTS:
% logbook - table containing feature configurations tested and corresponding validation accuracy
% 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
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] = trainAndValidateNetwork(adsTrain,adsValida
    valAccuracy = mean(trueLabels==predictedLabels)*100;
    if valAccuracy > bestValidationAccuracy
        bestValidationAccuracy = valAccuracy;
        bestNet = net;
        bestNormalizers = normalizers;
    end
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

% 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

```

Trim or Pad

```

function y = trimOrPad(x,n)
% y = trimOrPad(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 percussive 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, use a pretrained GAN generator to synthesize percussive sounds.

Download the pretrained generator.

```
matFileName = "drumGeneratorWeights.mat";  
loc = matlab.internal.examples.downloadSupportFile("audio","GanAudioSynthesis/" + matFileName);  
copyfile(loc,pwd)
```

The function `synthesizePercussiveSound` on page 1-578 calls a pretrained network to synthesize a percussive sound sampled at 16 kHz. The `synthesizePercussiveSound` function is included at the end of this example.

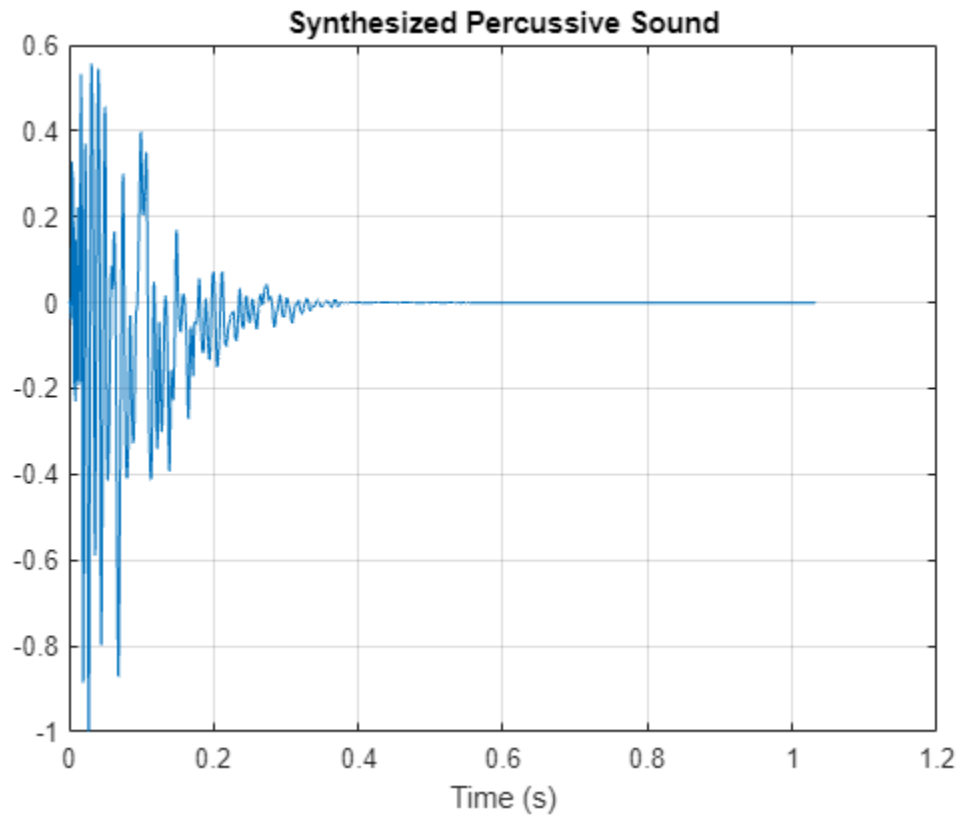
Synthesize a percussive sound and listen to it.

```
synthsound = synthesizePercussiveSound;
```

```
fs = 16e3;  
sound(synthsound, fs)
```

Plot the synthesized percussive sound.

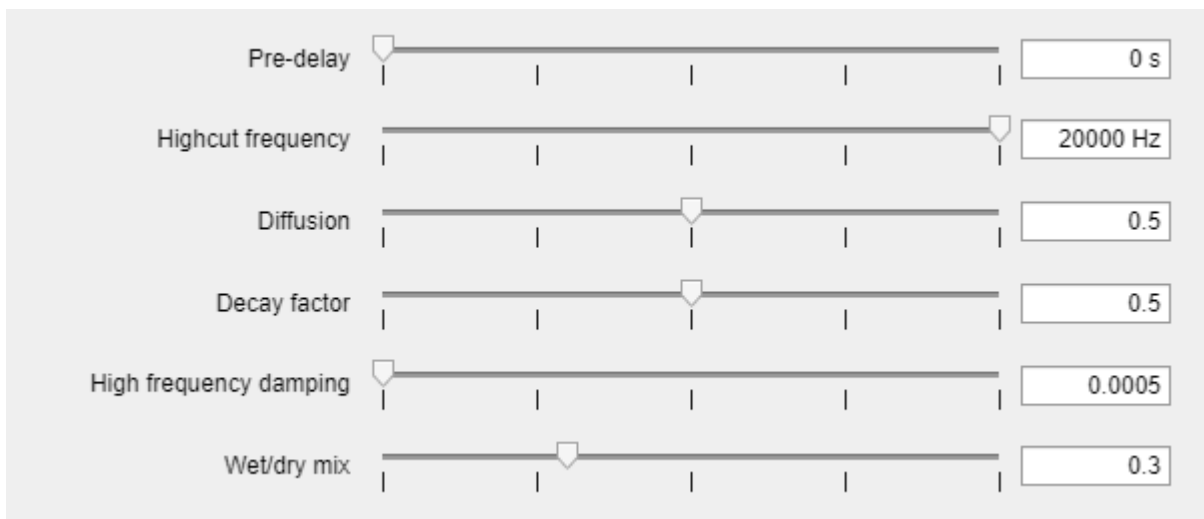
```
t = (0:length(synthsound)-1)/fs;  
plot(t,synthsound)  
grid on  
xlabel("Time (s)")  
title("Synthesized Percussive Sound")
```

You can use the percussive sounds synthesizer with other audio effects to create more complex applications. For example, you can apply reverberation to the synthesized percussive sounds.

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);
```

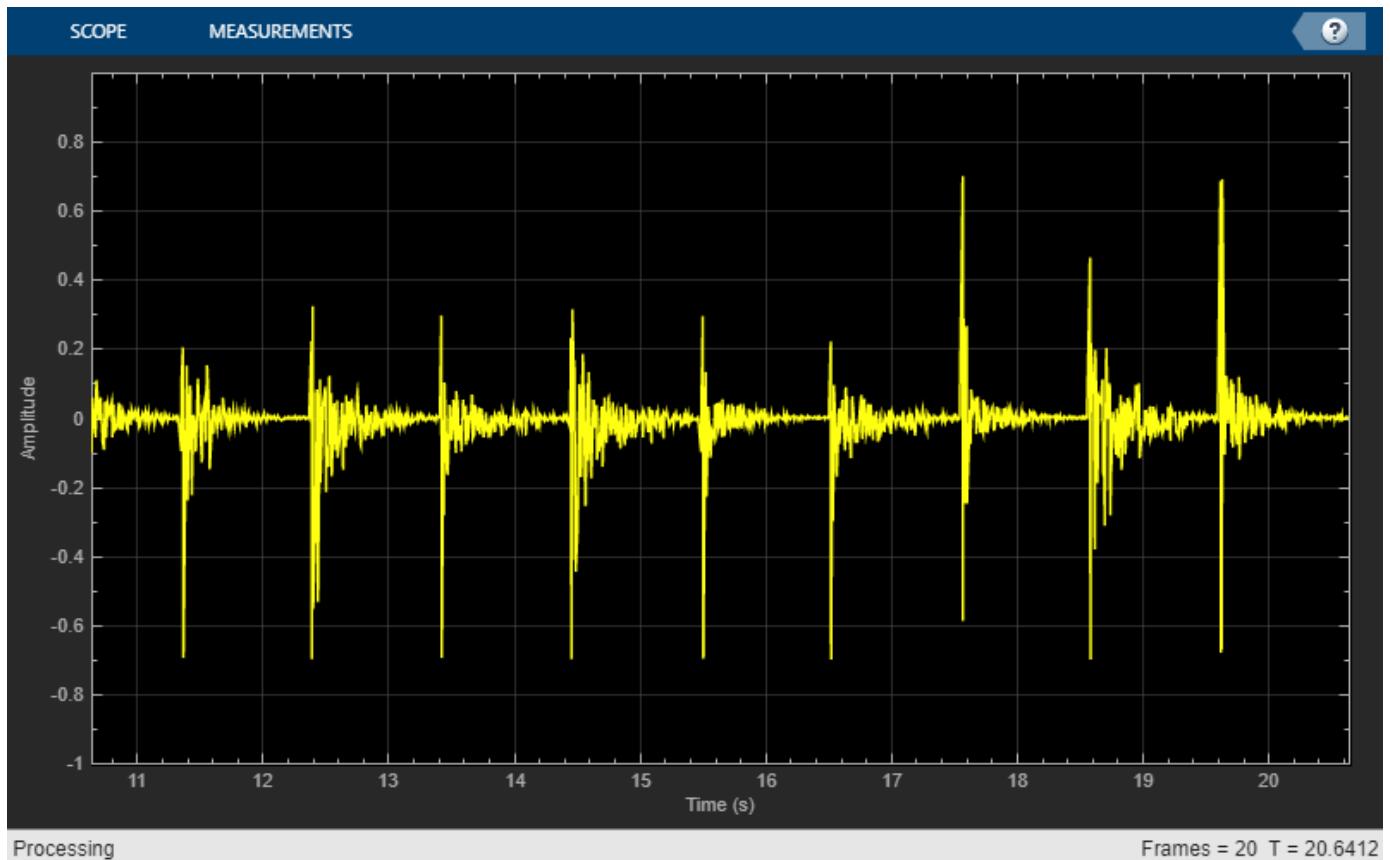


Create a `timescope` object to visualize the percussive sounds.

```
ts = timescope(SampleRate=fs, ...
    TimeSpanSource="Property", ...
    TimeSpanOverrunAction="Scroll", ...
    TimeSpan=10, ...
    BufferLength=10*256*64, ...
    ShowGrid=true, ...
    YLimits=[-1 1]);
```

In a loop, synthesize the percussive sounds 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
    synthsound = synthesizePercussiveSound;
    synthsound = reverb(synthsound);
    ts(synthsound(:,1));
    soundsc(synthsound, fs)
    pause(0.5)
end
```



Train the GAN

Now that you have seen the pretrained percussive sounds generator in action, you can investigate the training process in detail.

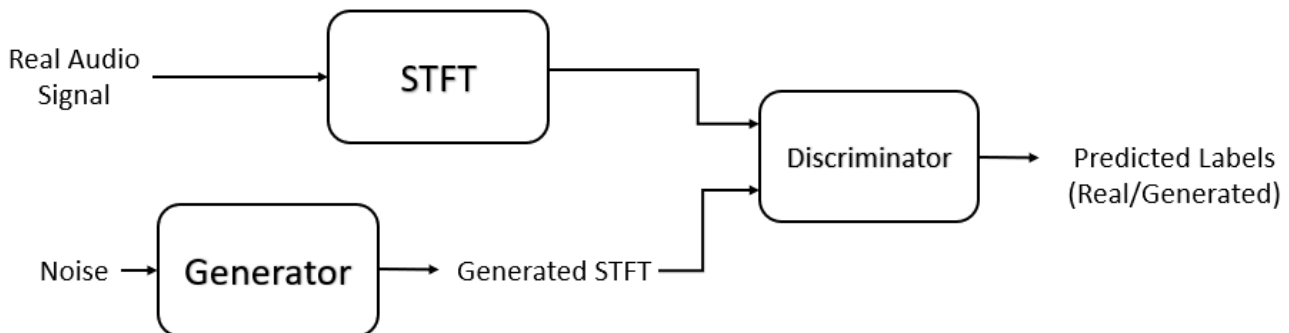
A GAN is a type of deep learning network that generates data with characteristics similar to the training data.

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 percussive sounds. You train the discriminator to identify whether an STFT was synthesized by the generator or computed from a real audio signal. You create the real STFTs by computing the STFT of short recordings of real percussive sounds.



Load Training Data

Train a GAN using the Freesound One-Shot Percussive Sounds dataset [2] on page 1-580. Download and extract the dataset. Remove any files with licenses that prohibit commercial use.

```

url1 = "https://zenodo.org/record/4687854/files/one_shot_percussive_sounds.zip";
url2 = "https://zenodo.org/record/4687854/files/licenses.txt";
downloadFolder = tempdir;

percussivesoundsFolder = fullfile(downloadFolder,"one_shot_percussive_sounds");
licensefilename = fullfile(percussivesoundsFolder,"licenses.txt");
if ~datasetExists(percussivesoundsFolder)
    disp("Downloading Freesound One-Shot Percussive Sounds Dataset (112.6 MB) ...")
    unzip(url1,downloadFolder)
    websave(licensefilename,url2);
    removeRestrictiveLicence(percussivesoundsFolder,licensefilename)
end
  
```

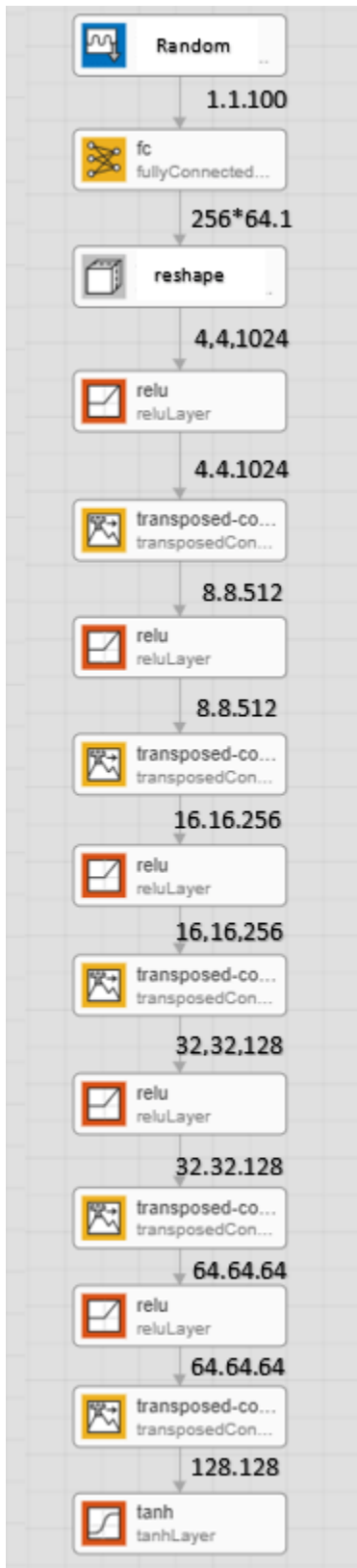
Create an `audioDatastore` object that points to the dataset.

```
ads = audioDatastore(percussivesoundsFolder,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] on page 1-580.



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] on page 1-580.



The discriminator network is defined in `modelDiscriminator` on page 1-575, which is included at the end of this example.

Generate Real Percussive Sounds Training Data

Generate STFT data from the percussive sound 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 canUseParallelPool
    pool = gcp;
    numPar = numpartitions(ads, pool);
else
    numPar = 1;
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to parallel pool with 6 workers.
```

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)

        % Read audio
        [x, xinfo] = read(subds);

        % Preprocess
        x = preprocessAudio(single(x), xinfo.SampleRate);

        % STFT
        S0 = stft(x, Window=win, OverlapLength=overlapLength, FrequencyRange="onesided");

        % Magnitude
        S = abs(S0);

        STrain(:,:, :, idx) = S;
    end
    STrainC{ii} = STrain;
end
```

```
Analyzing and transferring files to the workers ...done.
```

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] on page 1-580, 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™.

```
executionEnvironment = ;
```

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 = [];
```

Depending on your machine, training this network can take hours. To skip training, set `doTraining` to `false`.

```
doTraining = ;
```

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.

```
saveCheckpoints = ;
```

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 epochs, save a training snapshot to a MAT file.
if mod(epoch,10)==0
    disp("Epoch " + epoch + " out of " + maxEpochs + " complete.");
    if saveCheckpoints
        % Save checkpoint in case training is interrupted.
        save("audiogancheckpoint.mat", ...
            "generatorParameters","discriminatorParameters", ...
            "trailingAvgDiscriminator","trailingAvgSqDiscriminator", ...
            "trailingAvgGenerator","trailingAvgSqGenerator","iteration");
    end
end

```

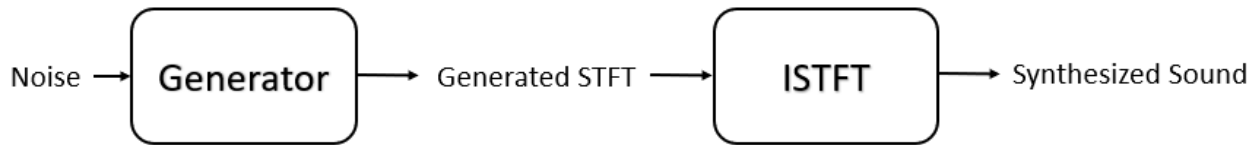
```
        end
    end
end
```

```
Epoch 10 out of 1000 complete.
Epoch 20 out of 1000 complete.
Epoch 30 out of 1000 complete.
Epoch 40 out of 1000 complete.
Epoch 50 out of 1000 complete.
Epoch 60 out of 1000 complete.
Epoch 70 out of 1000 complete.
Epoch 80 out of 1000 complete.
Epoch 90 out of 1000 complete.
Epoch 100 out of 1000 complete.
Epoch 110 out of 1000 complete.
Epoch 120 out of 1000 complete.
Epoch 130 out of 1000 complete.
Epoch 140 out of 1000 complete.
Epoch 150 out of 1000 complete.
Epoch 160 out of 1000 complete.
Epoch 170 out of 1000 complete.
Epoch 180 out of 1000 complete.
Epoch 190 out of 1000 complete.
Epoch 200 out of 1000 complete.
Epoch 210 out of 1000 complete.
Epoch 220 out of 1000 complete.
Epoch 230 out of 1000 complete.
Epoch 240 out of 1000 complete.
Epoch 250 out of 1000 complete.
Epoch 260 out of 1000 complete.
Epoch 270 out of 1000 complete.
Epoch 280 out of 1000 complete.
Epoch 290 out of 1000 complete.
Epoch 300 out of 1000 complete.
Epoch 310 out of 1000 complete.
Epoch 320 out of 1000 complete.
Epoch 330 out of 1000 complete.
Epoch 340 out of 1000 complete.
Epoch 350 out of 1000 complete.
Epoch 360 out of 1000 complete.
Epoch 370 out of 1000 complete.
Epoch 380 out of 1000 complete.
Epoch 390 out of 1000 complete.
Epoch 400 out of 1000 complete.
Epoch 410 out of 1000 complete.
Epoch 420 out of 1000 complete.
Epoch 430 out of 1000 complete.
Epoch 440 out of 1000 complete.
Epoch 450 out of 1000 complete.
Epoch 460 out of 1000 complete.
Epoch 470 out of 1000 complete.
Epoch 480 out of 1000 complete.
Epoch 490 out of 1000 complete.
Epoch 500 out of 1000 complete.
Epoch 510 out of 1000 complete.
Epoch 520 out of 1000 complete.
Epoch 530 out of 1000 complete.
Epoch 540 out of 1000 complete.
```

Epoch 550 out of 1000 complete.
Epoch 560 out of 1000 complete.
Epoch 570 out of 1000 complete.
Epoch 580 out of 1000 complete.
Epoch 590 out of 1000 complete.
Epoch 600 out of 1000 complete.
Epoch 610 out of 1000 complete.
Epoch 620 out of 1000 complete.
Epoch 630 out of 1000 complete.
Epoch 640 out of 1000 complete.
Epoch 650 out of 1000 complete.
Epoch 660 out of 1000 complete.
Epoch 670 out of 1000 complete.
Epoch 680 out of 1000 complete.
Epoch 690 out of 1000 complete.
Epoch 700 out of 1000 complete.
Epoch 710 out of 1000 complete.
Epoch 720 out of 1000 complete.
Epoch 730 out of 1000 complete.
Epoch 740 out of 1000 complete.
Epoch 750 out of 1000 complete.
Epoch 760 out of 1000 complete.
Epoch 770 out of 1000 complete.
Epoch 780 out of 1000 complete.
Epoch 790 out of 1000 complete.
Epoch 800 out of 1000 complete.
Epoch 810 out of 1000 complete.
Epoch 820 out of 1000 complete.
Epoch 830 out of 1000 complete.
Epoch 840 out of 1000 complete.
Epoch 850 out of 1000 complete.
Epoch 860 out of 1000 complete.
Epoch 870 out of 1000 complete.
Epoch 880 out of 1000 complete.
Epoch 890 out of 1000 complete.
Epoch 900 out of 1000 complete.
Epoch 910 out of 1000 complete.
Epoch 920 out of 1000 complete.
Epoch 930 out of 1000 complete.
Epoch 940 out of 1000 complete.
Epoch 950 out of 1000 complete.
Epoch 960 out of 1000 complete.
Epoch 970 out of 1000 complete.
Epoch 980 out of 1000 complete.
Epoch 990 out of 1000 complete.
Epoch 1000 out of 1000 complete.

Synthesize Sounds

Now that you have trained the network, you can investigate the synthesis process in more detail.



The trained percussive sound 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.

If you skipped training, load the weights of a pretrained generator.

```

if ~doTraining
    load(matFileName, "generatorParameters", "SMean", "SStd");
end
  
```

The generator takes 1-by-1-by-100 vectors of random values as an input. Generate a sample input vector.

```
dLZ = dLarray(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 `dLarray` 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.

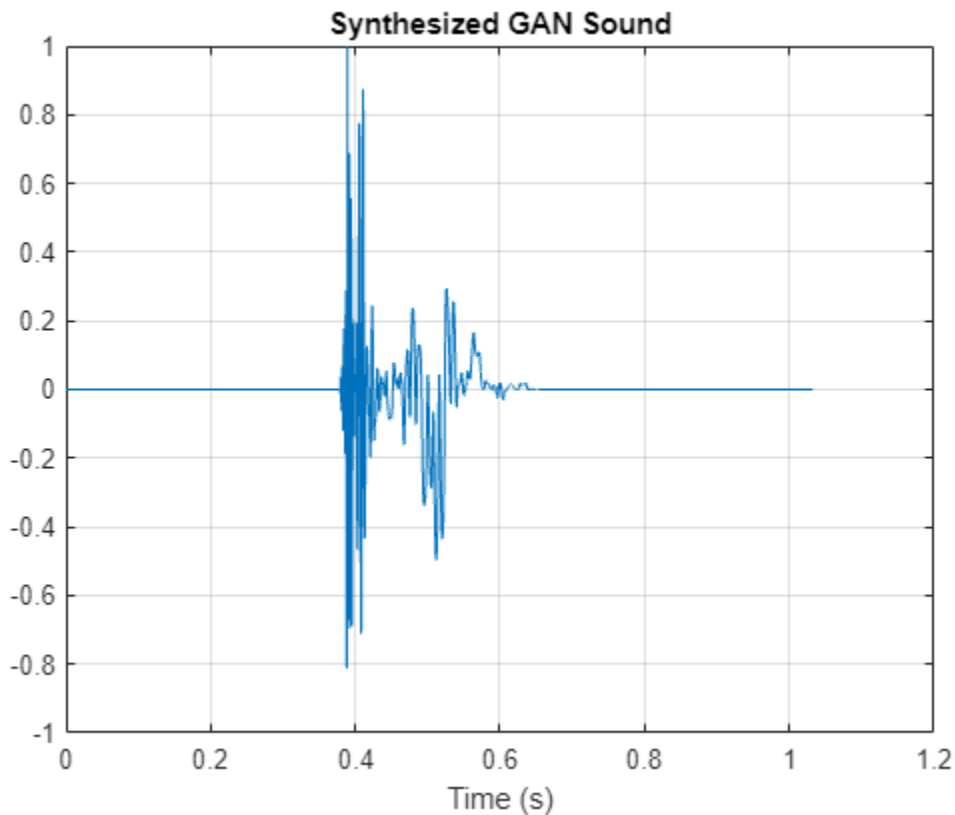
```
myAudio = stftmag2sig(S,256, ...
    FrequencyRange="twosided", ...
    Window=hann(256,"periodic"), ...
    OverlapLength=128, ...
    MaxIterations=20, ...
    Method="fgla");
myAudio = myAudio./max(abs(myAudio),[],"all");
myAudio = myAudio(128*100:end-128*100);
```

Listen to the synthesized percussive sound.

```
sound(gather(myAudio), fs)
```

Plot the synthesized percussive sound.

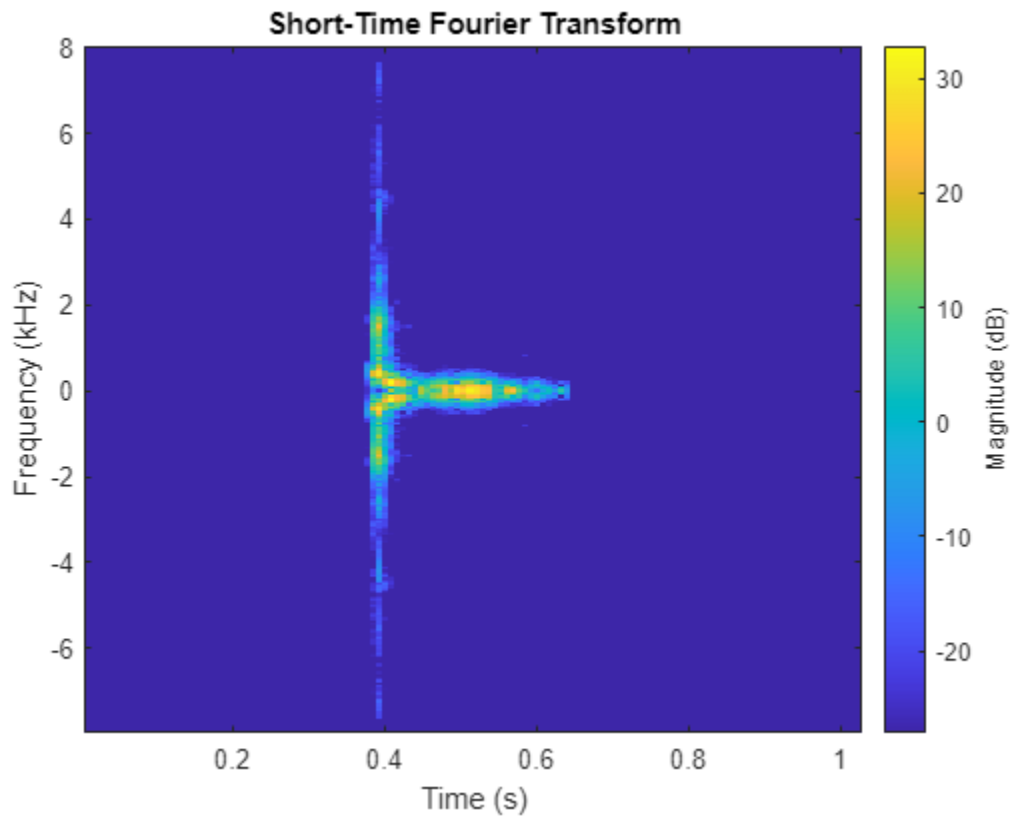
```
t = (0:length(myAudio)-1)/fs;
plot(t,myAudio)
grid on
xlabel("Time (s)")
title("Synthesized GAN Sound")
```



Plot the STFT of the synthesized percussive sound.

figure

```
stft(myAudio,fs,Window=hann(256,"periodic"),OverlapLength=128);
```



Model Generator Function

The `modelGenerator` function upscales 1-by-1-by-100 arrays (`d1X`) to 128-by-128-by-1 arrays (`d1Y`). `parameters` is a structure holding the weights of the generator layers. The generator architecture is defined in Table 4 of [1] on page 1-580.

```
function d1Y = modelGenerator(d1X,parameters)
```

```
d1Y = fullyconnect(d1X,parameters.FC.Weights,parameters.FC.Bias,Dataformat="SSCB");
```

```
d1Y = reshape(d1Y,[1024 4 4 size(d1Y,2)]);
```

```
d1Y = permute(d1Y,[3 2 1 4]);
```

```
d1Y = relu(d1Y);
```

```
d1Y = dltranspconv(d1Y,parameters.Conv1.Weights,parameters.Conv1.Bias,Stride=2,Cropping="same",Dataformat="SSCB");
```

```
d1Y = dltranspconv(d1Y,parameters.Conv2.Weights,parameters.Conv2.Bias,Stride=2,Cropping="same",Dataformat="SSCB");
```

```
d1Y = dltranspconv(d1Y,parameters.Conv3.Weights,parameters.Conv3.Bias,Stride=2,Cropping="same",Dataformat="SSCB");
```

```
d1Y = dltranspconv(d1Y,parameters.Conv4.Weights,parameters.Conv4.Bias,Stride=2,Cropping="same",Dataformat="SSCB");
```



```
dLY = dltranspconv(dLY,parameters.Conv5.Weights,parameters.Conv5.Bias,Stride=2,Cropping="same",D
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 `X`, and an array of random values `Z`, and returns the gradients of the discriminator loss with respect to the learnable parameters in the networks.

```
function gradientsDiscriminator = modelDiscriminatorGradients(discriminatorParameters,generatorPa

% Calculate the predictions for real data with the discriminator network.
Y = modelDiscriminator(X,discriminatorParameters);

% Calculate the predictions for generated data with the discriminator network.
Xgen = modelGenerator(Z,generatorParameters);
Ygen = modelDiscriminator(dlarray(Xgen,"SSCB"),discriminatorParameters);

% Calculate the GAN loss.
lossDiscriminator = ganDiscriminatorLoss(Y,Ygen);

% 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 `Z`, and returns the gradients of the generator loss with respect to the learnable parameters in the networks.

```
function gradientsGenerator = modelGeneratorGradients(discriminatorParameters,generatorParameters)  
  
% Calculate the predictions for generated data with the discriminator network.  
Xgen = modelGenerator(Z,generatorParameters);  
Ygen = modelDiscriminator(dlarray(Xgen,"SSCB"),discriminatorParameters);  
  
% Calculate the GAN loss  
lossGenerator = ganGeneratorLoss(Ygen);  
  
% 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] on page 1-580.

```
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] on page 1-580.

```
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 = dlarray(weights);
discriminatorParameters.Conv1.Bias = dlarray(bias);

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) dim 2*dim]);
bias = zeros(1,1,2*dim,"single");
discriminatorParameters.Conv2.Weights = dlarray(weights);
discriminatorParameters.Conv2.Bias = dlarray(bias);

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 2*dim 4*dim]);
bias = zeros(1,1,4*dim,"single");
discriminatorParameters.Conv3.Weights = dlarray(weights);
discriminatorParameters.Conv3.Bias = dlarray(bias);

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 4*dim 8*dim]);
bias = zeros(1,1,8*dim,"single");
discriminatorParameters.Conv4.Weights = dlarray(weights);
discriminatorParameters.Conv4.Bias = dlarray(bias);

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 8*dim 16*dim]);
bias = zeros(1,1,16*dim,"single");
discriminatorParameters.Conv5.Weights = dlarray(weights);
discriminatorParameters.Conv5.Bias = dlarray(bias);

% fully connected
weights = iGlorotInitialize([1,4 * 4 * dim * 16]);
bias = zeros(1,1,"single");
discriminatorParameters.FC.Weights = dlarray(weights);
discriminatorParameters.FC.Bias = dlarray(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 = dlarray(weights);
generatorParameters.FC.Bias = dlarray(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 = dlarray(weights);
generatorParameters.Conv1.Bias = dlarray(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 Percussive Sound

`synthesizePercussiveSound` uses a pretrained network to synthesize percussive sounds.

```

function y = synthesizePercussiveSound

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(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

```

```

function out = preprocessAudio(in,fs)
% Ensure mono
in = mean(in,2);

```

```

% Resample to 16kHz
x = resample(in,16e3,fs);

```

```

% Cut or pad to have approximately 1-second length plus padding to ensure
% 128 analysis windows for an STFT with 256-point window and 128-point
% overlap.
y = trimOrPad(x,16513);

```

```

% Normalize
out = y./max(abs(y));

```

```

end

```

```

function y = trimOrPad(x,n)

```

```

%trimOrPad Trim or pad audio
%
% y = trimOrPad(x,n) trims or pads the input x to n samples along the first
% dimension. 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.

a = size(x,1);
if a < n
    frontPad = floor((n-a)/2);
    backPad = n - a - frontPad;
    y = [zeros(frontPad,size(x,2),like=x);x;zeros(backPad,size(x,2),like=x)];
elseif a > n
    frontTrim = floor((a-n)/2) + 1;
    backTrim = a - n - frontTrim;
    y = x(frontTrim:end-backTrim,:);
else
    y = x;
end

end

function removeRestrictiveLicence(percussivesoundsFolder,licensefilename)
%removeRestrictiveLicense Remove restrictive license

% Parse the licenses file that maps ids to license. Create a table to hold the info.
f = fileread(licensefilename);
K = jsondecode(f);
fns = fields(K);
T = table(Size=[numel(fns),4], ...
    VariableTypes=["string","string","string","string"], ...
    VariableNames=["ID","FileName","UserName","License"]);
for ii = 1:numel(fns)
    fn = string(K.(fns{ii}).name);
    li = string(K.(fns{ii}).license);
    id = extractAfter(string(fns{ii}),"x");
    un = string(K.(fns{ii}).username);
    T(ii,:) = {id,fn,un,li};
end

% Remove any files that prohibit commercial use. Find the file inside the
% appropriate folder, and then delete it.
unsupportedLicense = "http://creativecommons.org/licenses/by-nc/3.0/";
fileToRemove = T.ID(strcmp(T.License,unsupportedLicense));
for ii = 1:numel(fileToRemove)
    fileInfo = dir(fullfile(percussivesoundsFolder,"**",fileToRemove(ii)+".wav"));
    delete(fullfile(fileInfo.folder,fileInfo.name))
end

end

```

Reference

[1] Donahue, C., J. McAuley, and M. Puckette. 2019. "Adversarial Audio Synthesis." ICLR.

[2] Ramires, Antonio, Pritish Chandna, Xavier Favory, Emilia Gomez, and Xavier Serra. "Neural Percussive Synthesis Parameterised by High-Level Timbral Features." *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020. <https://doi.org/10.1109/icassp40776.2020.9053128>.

See Also

text2speech

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 1-605 on acoustic features (usually mfcc). For an example, see “Speaker Verification Using Gaussian Mixture Model” on page 1-527. 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 1-605 [3] on page 1-605. However, [4] on page 1-605 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 1-605 proposed replacing the cosine similarity scoring with a probabilistic LDA (PLDA) model. [11] on page 1-606 and [12] on page 1-606 proposed a method to Gaussianize the i-vectors and therefore make Gaussian assumptions in the PLDA, referred to as G-PLDA or simplified PLDA. 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 1-606. Recently, deep learning techniques have been proposed to replace i-vectors with *d-vectors* or *x-vectors* [8] on page 1-605 [6] on page 1-605.

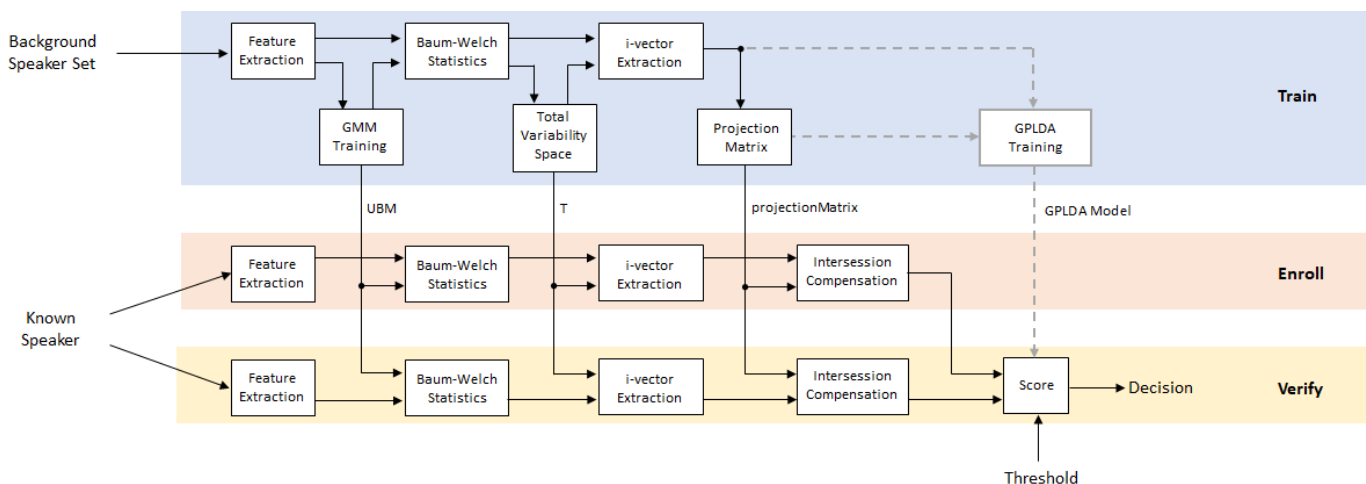
Use an i-Vector System

Audio Toolbox provides `ivectorSystem` which encapsulates the ability to train an i-vector system, enroll speakers or other audio labels, evaluate the system for a decision threshold, and identify or verify speakers or other audio labels. See `ivectorSystem` for examples of using this feature and applying it to several applications.

To learn more about how an i-vector system works, continue with the example.

Develop an i-Vector System

In this example, you develop a standard 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 1-605. The data set consists of 20 English native speakers reading 2342 phonetically rich sentences from the TIMIT corpus. Download and extract the data set.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "ptdb-tug.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
dataset = fullfile(dataFolder, "ptdb-tug");
```

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(dataset, "SPEECH DATA", "FEMALE", "MIC"), fullfile(dataset, "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 into training, evaluation, and test sets. The training set contains 16 speakers. The evaluation set contains four speakers and is further divided into an enrollment set and a set to evaluate the detection error tradeoff of the trained i-vector system, and a test set.

```
developmentLabels = categorical(["M01","M02","M03","M04","M06","M07","M08","M09","F01","F02","F03","F04","F06","F07","F08","F09"]);
evaluationLabels = categorical(["M05","M10","F05","F10"]);

adsTrain = subset(ads,ismember(ads.Labels,developmentLabels));

adsEvaluate = subset(ads,ismember(ads.Labels,evaluationLabels));

numFilesPerSpeakerForEnrollment = 3  ;
[adsEnroll,adsTest,adsDET] = splitEachLabel(adsEvaluate,numFilesPerSpeakerForEnrollment,2);
```

Display the label distributions of the resulting audioDatastore objects.

```
countEachLabel(adsTrain)
```

```
ans=16x2 table
  Label    Count
  -----  -
  F01      236
  F02      236
  F03      236
  F04      236
  F06      236
  F07      236
  F08      234
  F09      236
  M01      236
  M02      236
  M03      236
  M04      236
  M06      236
  M07      236
  M08      236
  M09      236
```

```
countEachLabel(adsEnroll)
```

```
ans=4x2 table
  Label    Count
  -----  -
  F05      3
  F10      3
  M05      3
  M10      3
```

```
countEachLabel(adsDET)
```

```
ans=4x2 table
  Label    Count
  -----  -
  F05      231
  F10      231
  M05      231
  M10      231
```

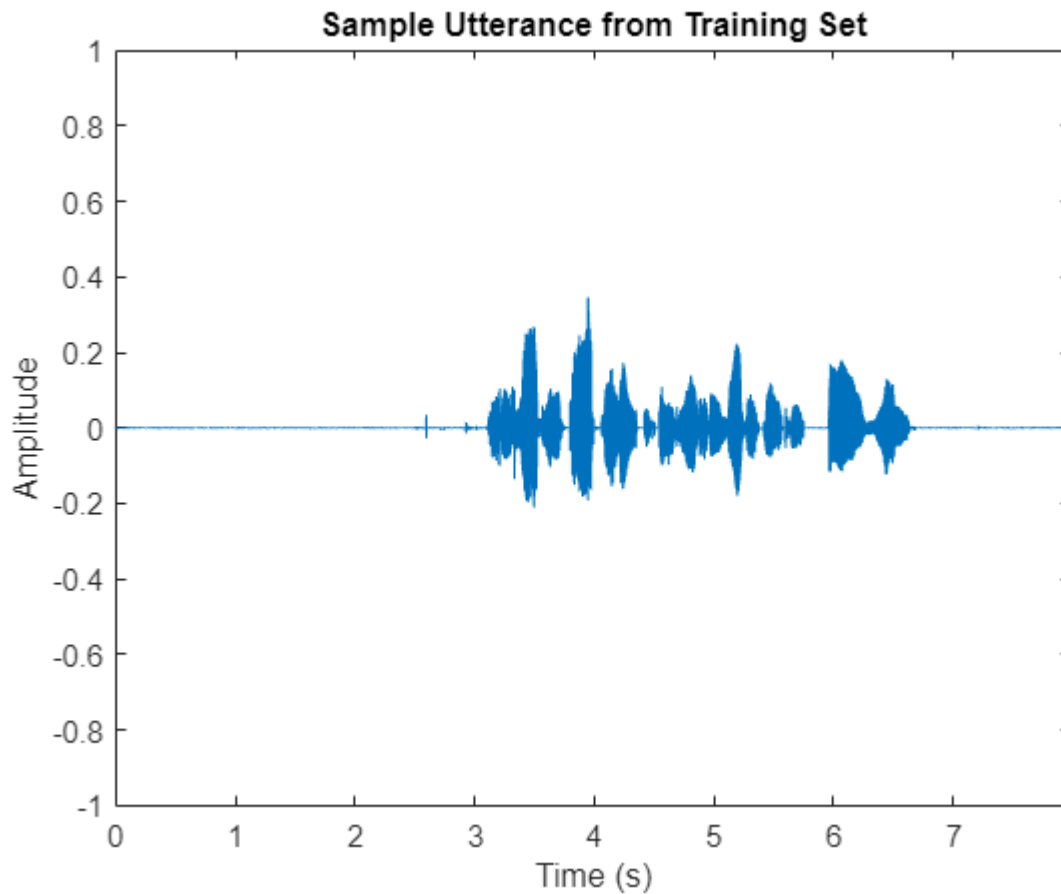
```
countEachLabel(adsTest)
```

```
ans=4x2 table
```

Label	Count
F05	2
F10	2
M05	2
M10	2

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 =  false ;
if speedUpExample
    adsTrain = splitEachLabel(adsTrain,30);
    adsDET = splitEachLabel(adsDET,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);
setExtractorParameters(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 = 797
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

```

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

Feature Normalization Factors

Use the helper function, `helperFeatureExtraction`, to extract all features from the data set. The `helperFeatureExtraction` on page 1-603 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{:});
disp("Feature extraction from training set complete (" + toc + " seconds).")

```

Feature extraction from training set complete (64.0731 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 = 64  ;
if speedUpExample
    numComponents = 32;
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 = 10  ;
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
end

% Print current log-likelihood
disp("Training UBM: " + iter + "/" + maxIter + " complete (" + round(toc) + " seconds), Log-likelihood = " + logLikelihoodSum);

% 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: 1/10 complete (57 seconds), Log-likelihood = -75180473
Training UBM: 2/10 complete (57 seconds), Log-likelihood = -75115244
Training UBM: 3/10 complete (57 seconds), Log-likelihood = -75064164
Training UBM: 4/10 complete (57 seconds), Log-likelihood = -75024270
Training UBM: 5/10 complete (57 seconds), Log-likelihood = -74994504
Training UBM: 6/10 complete (57 seconds), Log-likelihood = -74970605

```

Training UBM: 7/10 complete (55 seconds), Log-likelihood = -74950526
 Training UBM: 8/10 complete (58 seconds), Log-likelihood = -74933181
 Training UBM: 9/10 complete (58 seconds), Log-likelihood = -74917145
 Training UBM: 10/10 complete (55 seconds), Log-likelihood = -74901292

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
end
```

```

        Nc = [Nc,Npart];
        Fc = [Fc,Fpart];
    end
    disp("Baum-Welch statistics completed (" + toc + " seconds).")
    Baum-Welch statistics completed (57.5179 seconds).

```

Expand the statistics into matrices and center $F(s)$, as described in [3] on page 1-605, 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 CF -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 1-605:

- 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 1-605 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 = 32  _____ ;
if speedUpExample
    numTdim = 16;
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 = 5  _____ ;
```

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{:});

disp("Training Total Variability Space: " + iterIdx + "/" + numIterations + " complete (" +
end

Training Total Variability Space: 1/5 complete (1.97 seconds).
Training Total Variability Space: 2/5 complete (1.69 seconds).
Training Total Variability Space: 3/5 complete (1.79 seconds).
Training Total Variability Space: 4/5 complete (1.56 seconds).
Training Total Variability Space: 5/5 complete (1.74 seconds).

```

i-Vector Extraction

Once the total variability space is calculated, you can calculate the i-vectors as [4] on page 1-605:

$$w = (I + T\Sigma^{-1}NT)'T\Sigma^{-1}F$$

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) * TS * f(:);
end
ivectorPerSpeaker{speakerIdx} = ivectorPerFile;
end
disp("I-vectors extracted from training set (" + toc + " seconds).")

```

I-vectors extracted from training set (65.8347 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 1-605:

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 = true;
if performLDA
    tic

    numEigenvectors = 16;

    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;

    disp("LDA projection matrix calculated (" + round(toc,2) + " seconds).")
end
```

LDA projection matrix calculated (0.22 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 1-606.

Given the within-class covariance matrix:

$$W = \frac{1}{S} \sum_{s=1}^S \frac{1}{n_s} \sum_{i=1}^{n_s} (w_i^s - \bar{w}_s)(w_i^s - \bar{w}_s)'$$

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.

- 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;
    disp("WCCN projection matrix calculated (" + round(toc,4) + " seconds).")
end
```

WCCN projection matrix calculated (0.0096 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 1-606. 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

$$S = \sum_{ij} \varphi_{ij} \varphi_{ij}^T$$

```
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

$$f_i = \sum_{j=1}^{n_i} \varphi_{ij}$$

```
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 =  ;
```

Train the G-PLDA model using the EM algorithm described in [13] on page 1-606.

```
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 1-606. 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 = 56.2336

```

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 1-604 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.

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);

%i-vector Extraction
w = pinv(I + (TS.*repelem(n(:),numFeatures))' * T) * TSi * f(:);

% Intersession Compensation
w = projectionMatrix*w;

ivectorMat(:,fileIdx) = w;
end
% i-vector model
enrolledSpeakersByIdx{speakerIdx} = mean(ivectorMat,2);
end
disp("Speakers enrolled (" + round(toc,2) + " seconds).")
Speakers enrolled (0.44 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(adsDET.Labels);
numSpeakers = numel(speakersToTest);
scoreFRR = cell(numSpeakers,1);
tic
parfor speakerIdx = 1:numSpeakers
    adsPart = subset(adsDET,adsDET.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);
    end
end

```

```

% 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
w = pinv(I + (TS.*repelem(n(:),numFeatures))' * T) * TSi * f(:);

% Intersession Compensation
w = projectionMatrix*w;

% Score
if strcmpi(scoringMethod,"CSS")
    score(fileIdx) = dot(ivectorToTest,w)/(norm(w)*norm(ivectorToTest));
else
    score(fileIdx) = gpldaScore(gpldaModel,w,ivectorToTest);
end
end
scoreFRR{speakerIdx} = score;
end
disp("FRR calculated (" + round(toc,2) + " seconds).")
FRR calculated (20.77 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(adsDET.Labels);
numSpeakers = numel(speakersToTest);
scoreFAR = cell(numSpeakers,1);
tic
parfor speakerIdx = 1:numSpeakers
    adsPart = subset(adsDET,adsDET.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
w = pinv(I + (TS.*repelem(n(:),numFeatures))' * T) * TSi * f(:);

% Intersession compensation
w = projectionMatrix * w;

% Score
if strcmpi(scoringMethod,"CSS")
    score(fileIdx) = dot(ivectorToTest,w)/(norm(w)*norm(ivectorToTest));
else
    score(fileIdx) = gpldaScore(gpldaModel,w,ivectorToTest);
end
end
scoreFAR{speakerIdx} = score;
end
disp("FAR calculated (" + round(toc,2) + " seconds).")
FAR calculated (58.14 seconds).

```

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 vectors 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 vectors 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)]);

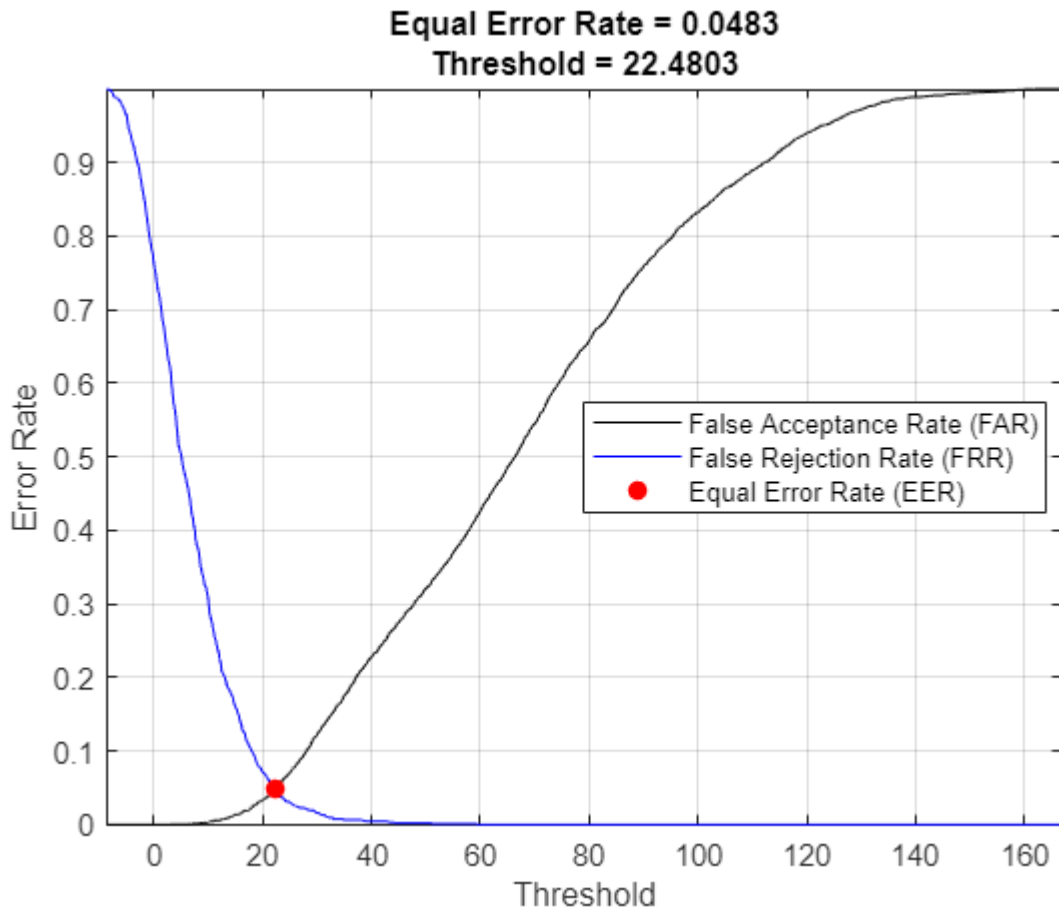
figure

```

```

plot(thresholdsToTest,FAR,"k", ...
     thresholdsToTest,FRR,"b", ...
     EERThreshold,EER,"ro",MarkerFaceColor="r")
title(["Equal Error Rate = " + round(EER,4),"Threshold = " + round(EERThreshold,4)])
xlabel("Threshold")
ylabel("Error Rate")
legend("False Acceptance Rate (FAR)","False Rejection Rate (FRR)","Equal Error Rate (EER)",Locat:
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) + s

    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 = wlwt'*term1*wlwt - 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>.

i-vector Score Normalization

An i-vector system outputs a raw score specific to the data and parameters used to develop the system. This makes interpreting the score and finding a consistent decision threshold for verification tasks difficult.

To address these difficulties, researchers developed *score normalization* and *score calibration* techniques.

- In *score normalization*, raw scores are normalized in relation to an 'imposter cohort'. Score normalization occurs before evaluating the detection error tradeoff and can improve the accuracy of a system and its ability to adapt to new data.
- In *score calibration*, raw scores are mapped to probabilities, which are used to better understand the system's confidence in decisions.

In this example, you apply score normalization to an i-vector system. To learn about score calibration, see “i-vector Score Calibration” on page 1-626.

For example purposes, you use cosine similarity scoring (CSS) throughout this example. Probabilistic linear discriminant analysis (PLDA) scoring is also improved by normalization, although less dramatically.

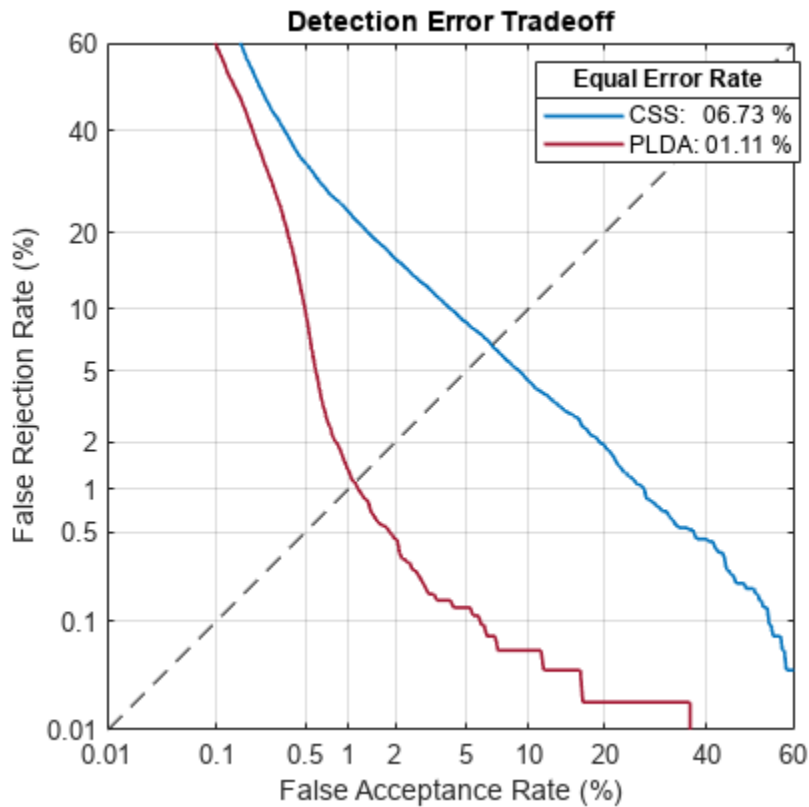
Download i-vector System and Data Set

To download a pretrained i-vector system suitable for speaker recognition, call `speakerRecognition`. The `ivectorSystem` returned was trained on the LibriSpeech data set, which consists of English-language 16 kHz recordings.

```
ivs = speakerRecognition();
```

The pretrained i-vector system achieves an equal error rate (EER) around 6.73% using CSS on the LibriSpeech test set. The EER achieved using PLDA is considerably better. However, because CSS is simpler, for the purposes of this example you investigate CSS only.

```
detectionErrorTradeoff(ivs)
```



The error rate on the LibriSpeech test set, and the accompanying default decision threshold for speaker verification, do not extend well to unseen data. To confirm this, download the PTDB-TUG [3] on page 1-624 data set. The supporting function, `loadDataset` on page 1-618, downloads the data set and then resamples it from 48 kHz to 16 kHz, which is the sample rate that the i-vector system was trained on. The `loadDataset` function returns four `audioDatastore` objects:

- `adsEnroll` - Contains files to enroll speakers into i-vector system.
- `adsTest` - Contains files to spot-check performance of the i-vector system.
- `adsDET` - Contains a large set of files to analyze the detection error tradeoff of the i-vector system.
- `adsImposter` - Contains a set of speakers not included in the other datastores. This set is used for score normalization.

```
targetSampleRate = ivs.SampleRate;
[adsEnroll,adsTest,adsDET,adsImposter] = loadDataset(targetSampleRate);
```

Enroll speakers from the enrollment dataset. When you `enroll` speakers, an i-vector template is created for each unique speaker label.

```
enroll(ivs,adsEnroll,adsEnroll.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .....done.
Enrollment complete.
```

```
enrolledLabels = categorical(ivs.EnrolledLabels.Properties.RowNames);
```

Spot-check the false rejection rate (FRR) and the false acceptance rate (FAR) using the `verify` object function. The `verify` object function scores the i-vector derived from the audio input against the i-vector template corresponding to the specified label. The function then compares the score to a decision threshold and either accepts or rejects the proposition that the audio input belongs to the specified speaker label. The default decision threshold corresponds to the equal error rate (EER) determined the last time the detection error tradeoff was evaluated.

```
FA = 0;
FR = 0;
reset(adsTest)
numToSpotCheck = 50;
for ii = 1:numToSpotCheck
    [audioIn,fileInfo] = read(adsTest);
    targetLabel = fileInfo.Label;

    FR = FR + ~verify(ivs,audioIn,targetLabel,"css");

    nontargetIdx = find(~ismember(enrolledLabels,targetLabel));
    nontargetLabel = enrolledLabels(nontargetIdx(randperm(numel(nontargetIdx),1))));

    FA = FA + verify(ivs,audioIn,nontargetLabel,"css");
end
FRR = FR./numToSpotCheck;
FAR = FA./numToSpotCheck;
disp(["False Rejection Rate = " + round(100*FRR,2) + " (%); ...
     "False Acceptance Rate = " + round(100*FAR,2) + " (%)"])

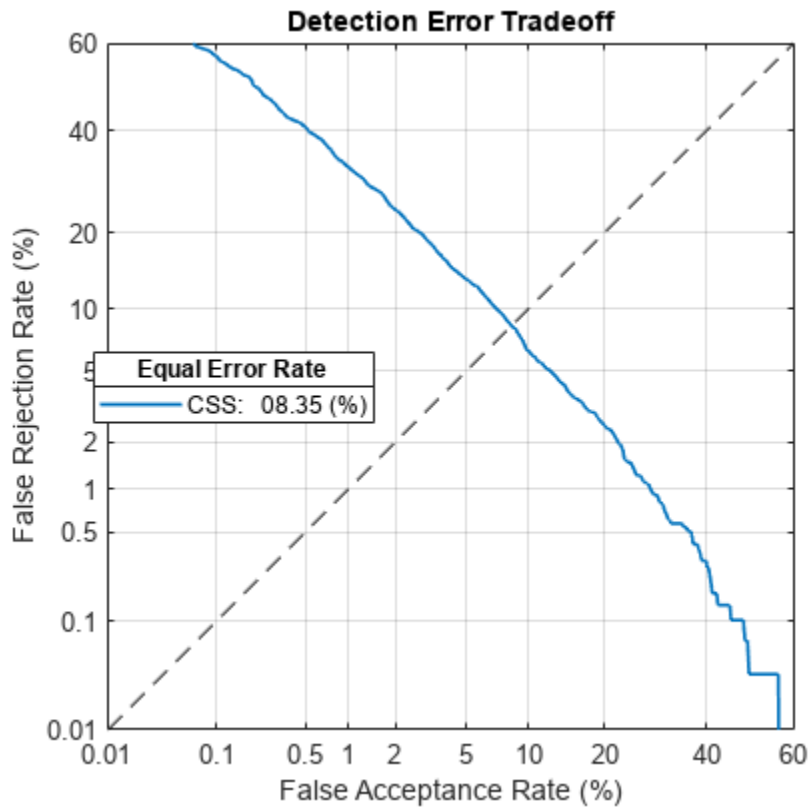
"False Rejection Rate = 0 (%)"
"False Acceptance Rate = 38 (%)"
```

The performance on this new dataset does not match performance reported when training and evaluating the i-vector system on the LibriSpeech data set. Also, the default decision threshold on the LibriSpeech data set does not correspond to the equal error rate on the PTDB-TUG data set.

To better evaluate the system's performance, and to choose a new decision threshold, call `detectionErrorTradeoff` again. This time call `detectionErrorTradeoff` with new evaluation data that is more suited to the target application. The evaluation data should be as close as possible to the data that your deployed system encounters in terms of vocabulary, prosody, signal duration, noise level, noise type, accents, channel characteristics, and so on.

```
detectionErrorTradeoff(ivs,adsDET,adsDET.Labels,Scorer="css")
```

```
Extracting i-vectors ...done.
Scoring i-vector pairs ...done.
Detection error tradeoff evaluation complete.
```



Spot-check the FAR and FRR of the updated system. The FAR and FRR are now reasonably close to the EER reported in the detection error tradeoff analysis. Note that calling `detectionErrorTradeoff` does not modify the i-vector extraction or scoring, only the default decision threshold for speaker verification. In the following sections, you enhance an i-vector system to perform score normalization. Score normalization helps an i-vector system extend to new datasets without the need to reevaluate the detection error tradeoff. Score normalization also helps bridge the performance gap between training a system and deploying it.

```

FA = 0;
FR = 0;
reset(adsTest)
for ii = 1:numToSpotCheck
    [audioIn,fileInfo] = read(adsTest);
    trueLabel = fileInfo.Label;

    FR = FR + ~verify(ivs,audioIn,trueLabel,"css");

    imposterIdx = find(~ismember(enrolledLabels,trueLabel));
    imposter = enrolledLabels(imposterIdx(randperm(numel(imposterIdx),1)));

    FA = FA + verify(ivs,audioIn,imposter,"css");
end
FRR = FR./numToSpotCheck;
FAR = FA./numToSpotCheck;
disp(["False Rejection Rate = " + round(100*FRR,2) + " (%)"; ...
     "False Acceptance Rate = " + round(100*FAR,2) + " (%)"])

```

```
"False Rejection Rate = 4 (%)"
"False Acceptance Rate = 8 (%)"
```

Score Normalization

Score normalization is a common approach to make target and non-target score distributions across speakers more similar. This enables a system to set a decision threshold that is closer to optimal for more speakers. In this example, you explore adaptive symmetric normalization variant 1 (S-norm1) [1] on page 1-624.

To motivate score normalization, first inspect the target and non-target score distributions for two enrolled labels against the same test cohort.

Isolate two template i-vectors corresponding to two speakers.

```
enrolledIvecs = cat(2,ivs.EnrolledLabels.ivector{1},ivs.EnrolledLabels.ivector{9});
label_e = categorical([ivs.EnrolledLabels.Properties.RowNames(1),ivs.EnrolledLabels.Properties.R
```

Extract i-vectors from the test set. The test set labels overlap with the enrolled labels.

```
testIvecs = ivector(ivs,adsTest);
label_t = adsTest.Labels;
```

Create indexing vectors to keep track of which test i-vectors correspond to which enrolled label. In the `targets` matrix, the columns correspond to the enrolled speakers, and the rows correspond to the test files. If the test label corresponds to the enrolled label, the value in the matrix is `true`, otherwise, the value is `false`.

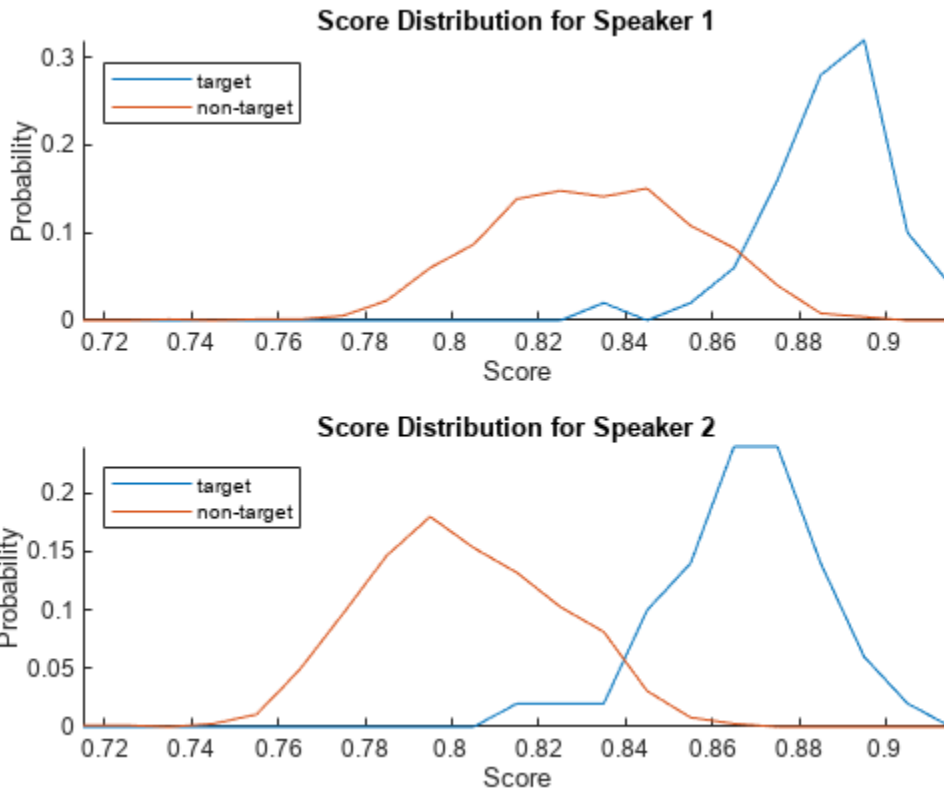
```
targets = [ismember(label_t,label_e(1)),ismember(label_t,label_e(2))];
```

Score the template i-vectors against the target and non-target i-vectors. The supporting function, `scoreTargets` on page 1-620, scores all combinations of the enrolled i-vectors against the test i-vectors and returns the scores separated into target scores (when the test and enroll labels are the same) and non-target scores (when the test and enroll labels are different).

```
[targetScores,nontargetScores] = scoreTargets(enrolledIvecs,testIvecs,targets);
```

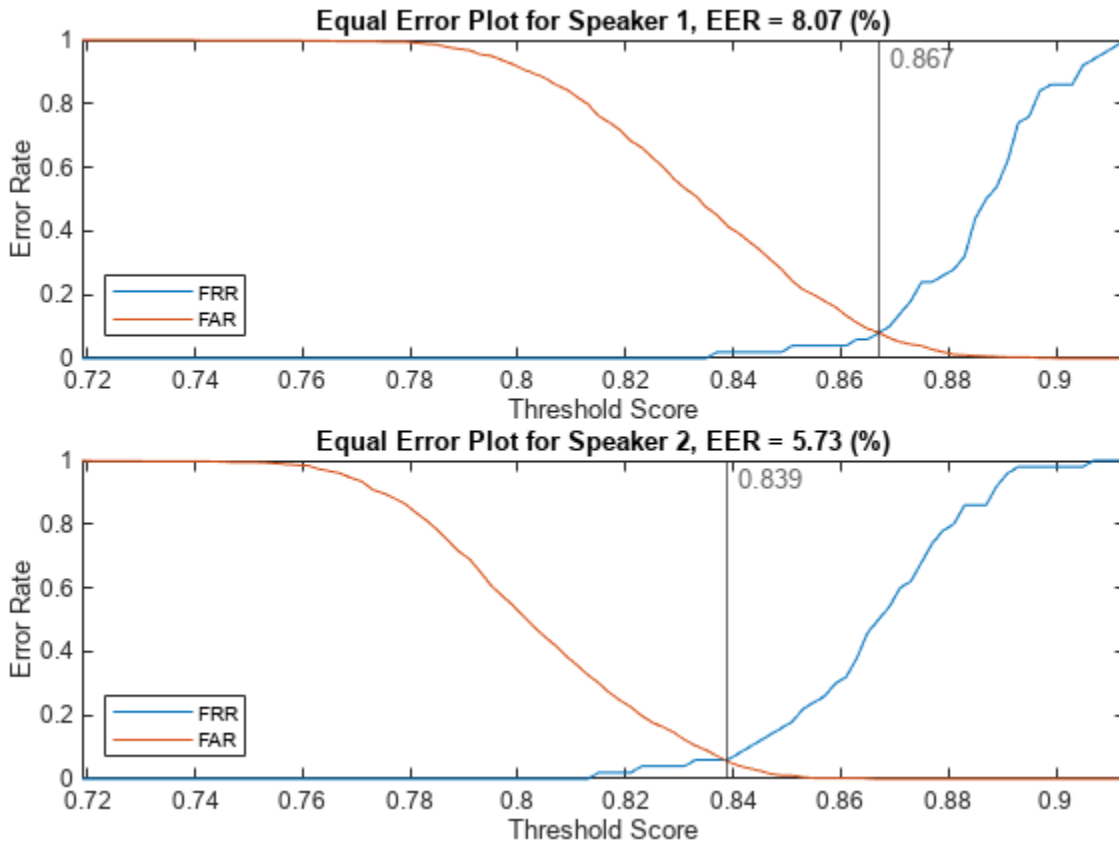
Use the supporting function, `plotScoreDistributions` on page 1-621, to display the target and non-target scores for each of the enrolled speakers. Note that the equal error rate (where the target and non-target distributions cross) is different for the two speakers. That is, assuming the equal error rate is the goal of the system, a single decision threshold cannot capture the equal error rate for both speakers.

```
plotScoreDistributions(targetScores,nontargetScores,Analyze="label")
```



Analyze the score distributions using an EER plot. EER plots reveal the relationship between a decision threshold and the probability of a false alarm or false rejection and are often analyzed to determine a decision threshold.

```
plotEER(targetScores, nontargetScores, Analyze="label")
```



Analyze Score Normalization on Two Speakers

In this example, you use adaptive symmetric normalization variant 1 (S-norm1) [1] on page 1-624. S-norm1 computes an average of normalized scores from Z-norm (zero score normalization) and T-norm (test score normalization).

$$s(e, t)_{s-norm1} = \frac{1}{2} \cdot \left(\frac{s(e, t) - \mu(S_e(\xi_e^{\text{top}}))}{\sigma(S_e(\xi_e^{\text{top}}))} + \frac{s(e, t) - \mu(S_t(\xi_t^{\text{top}}))}{\sigma(S_t(\xi_t^{\text{top}}))} \right)$$

where

- $s(e, t)$ is the raw score based on the enrollment e and test t i-vectors.
- ξ_e^{top} is the set of the top-scoring imposter cohort with enrolled i-vector, e .
- ξ_t^{top} is the set of the top-scoring imposter cohort with test i-vector, t .
- $S_e = \{s(e, \varepsilon_i)\}_{i=1}^N$, is the set of cohort scores formed by scoring enrollment utterance e with the top files from imposter cohort ξ .
- $S_e(\xi_e^{\text{top}}) = \{s(e, \varepsilon)\}_{\forall \varepsilon \in \xi_e^{\text{top}}}$, is the set of the top scores between an enrolled i-vector and imposter i-vectors.
- $S_t(\xi_t^{\text{top}}) = \{s(e, \varepsilon)\}_{\forall \varepsilon \in \xi_t^{\text{top}}}$, is the set of the top scores between a test i-vector and imposter i-vectors.

- $\mu(S)$ and $\sigma(S)$ are the mean and standard deviation of S .

To begin, extract i-vectors from the imposter cohort (ξ).

```
imposterIvecs = ivector(ivs,adsImposter);
```

Score the enrolled i-vectors against the imposter cohort ($S_e(\xi_e)$) and then isolate only the K best scores ($S_e(\xi_e^{\text{top}})$). [1] on page 1-624 suggests using the top 200-500 scoring files to create a speaker-dependent cohort. Finally, calculate the mean ($\mu(S_e(\xi_e^{\text{top}}))$) and standard deviation ($\sigma(S_e(\xi_e^{\text{top}}))$).

```
topK = 400  ;
```

```
imposterScores = sort(cosineSimilarityScore(enrolledIvecs,imposterIvecs),"descend");
imposterScores = imposterScores(1:min(topK,size(imposterScores,1)),:);
mu_e = mean(imposterScores,1);
std_e = std(imposterScores,[],1);
```

Calculate $\mu(S_t(\xi_t^{\text{top}}))$ and $\sigma(S_t(\xi_t^{\text{top}}))$ as above.

```
imposterScores = sort(cosineSimilarityScore(testIvecs,imposterIvecs),"descend");
imposterScores = imposterScores(1:min(topK,size(imposterScores,1)),:);
mu_t = mean(imposterScores,1);
std_t = std(imposterScores,[],1);
```

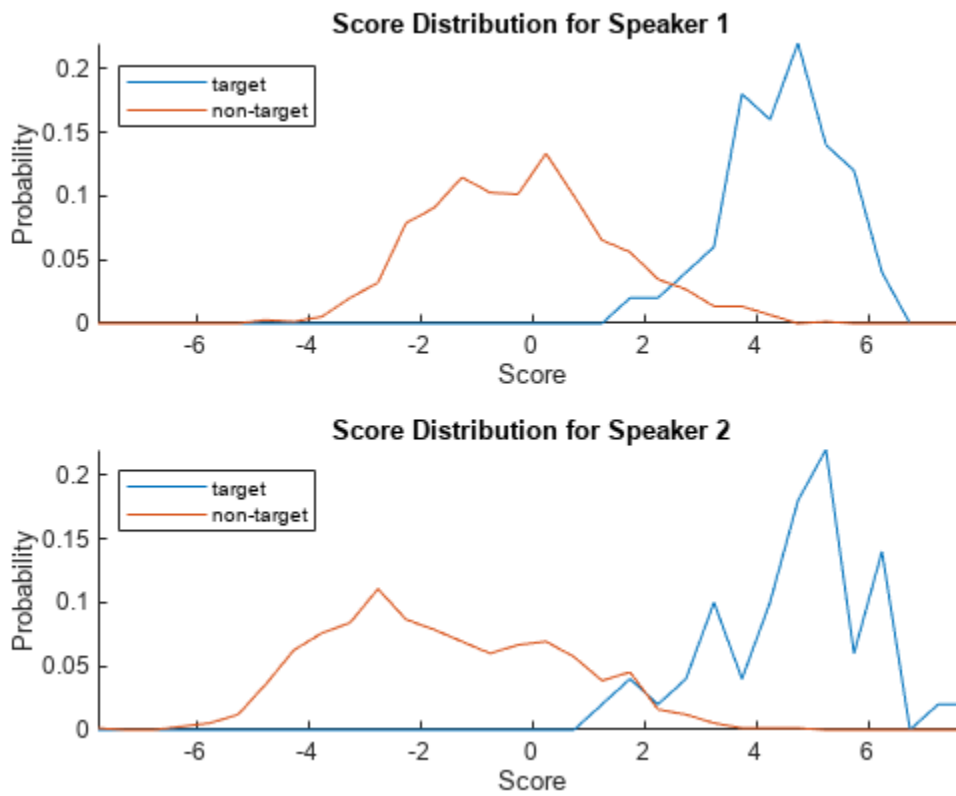
Score the test and enrollment i-vectors again, this time specifying the required normalization factors to perform adaptive s-norm1. The supporting function, `scoreTargets` on page 1-620, applies the normalization on the raw scores.

```
normFactorsSe = struct(mu=mu_e,std=std_e);
normFactorsSt = struct(mu=mu_t,std=std_t);

[targetScores,nontargetScores] = scoreTargets(enrolledIvecs,testIvecs,targets, ...
    NormFactorsSe=normFactorsSe,NormFactorsSt=normFactorsSt);
```

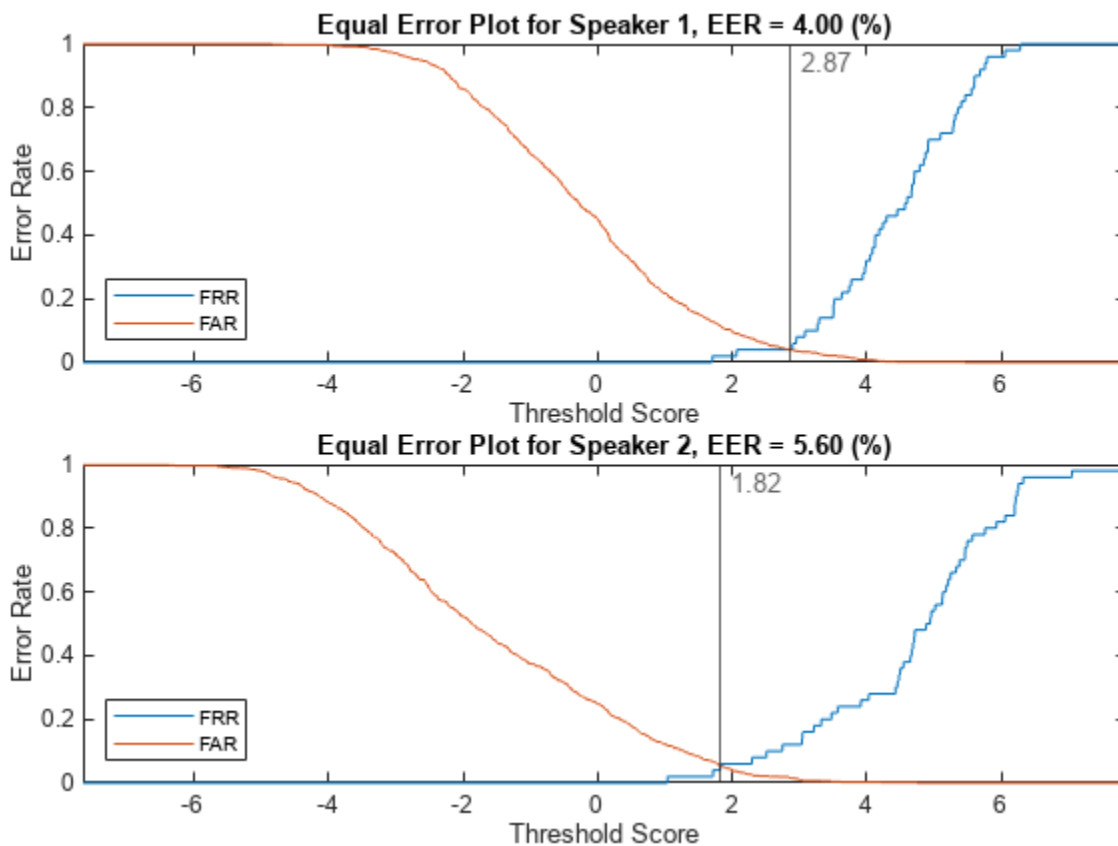
Plot the score distributions of the scores after applying adaptive s-norm1.

```
plotScoreDistributions(targetScores,nontargetScores,Analyze="label")
```

Analyze the equal error rate plots after applying adaptive s-norm1. The thresholds corresponding to the equal error rates for the two speakers are now closer together.

```
plotEER(targetScores, nontargetScores, Analyze="label")
```



Analyze Score Normalization on Detection Error Tradeoff Set

Extract i-vectors from the DET set.

```
testIvecs = ivector(ivs,adsDET);
```

Place all of the enrolled i-vectors into a matrix.

```
enrolledIvecs = cat(2,ivs.EnrolledLabels.ivector{:});
```

Calculate the normalization statistics for each enrolled and test i-vector. The supporting function, `getNormFactors` on page 1-624, performs the same operations as in `Analyze Score Normalization on Two Speakers` on page 1-613.

```
topK = 100  ;
```

```
normFactorsSe = getNormFactors(enrolledIvecs,imposterIvecs,TopK=topK);
normFactorsSt = getNormFactors(testIvecs,imposterIvecs,TopK=topK);
```

Create a targets matrix indicating which i-vector pairs have corresponding labels.

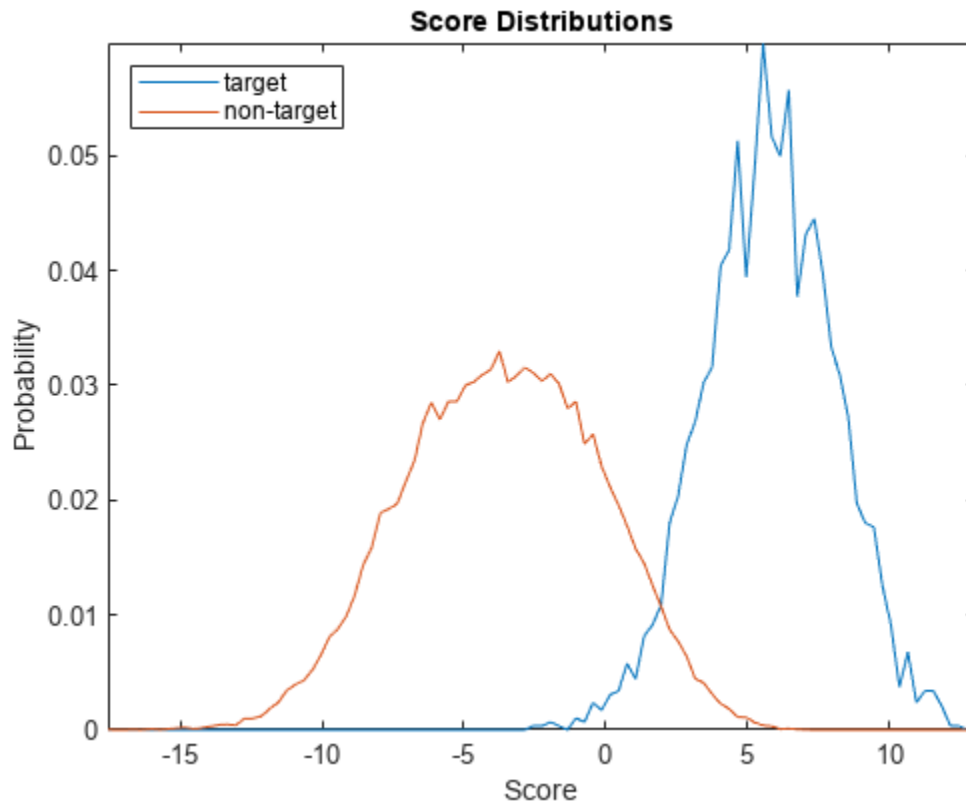
```
targets = true(numel(adsDET.Labels),height(ivs.EnrolledLabels));
for ii = 1:height(ivs.EnrolledLabels)
    targets(:,ii) = ismember(adsDET.Labels,ivs.EnrolledLabels.Properties.RowNames(ii));
end
```

Score each enrollment i-vector against each test i-vector.

```
[targetScores,nontargetScores] = scoreTargets(enrolledIvecs,testIvecs,targets, ...  
NormFactorsSe=normFactorsSe, NormFactorsSt=normFactorsSt);
```

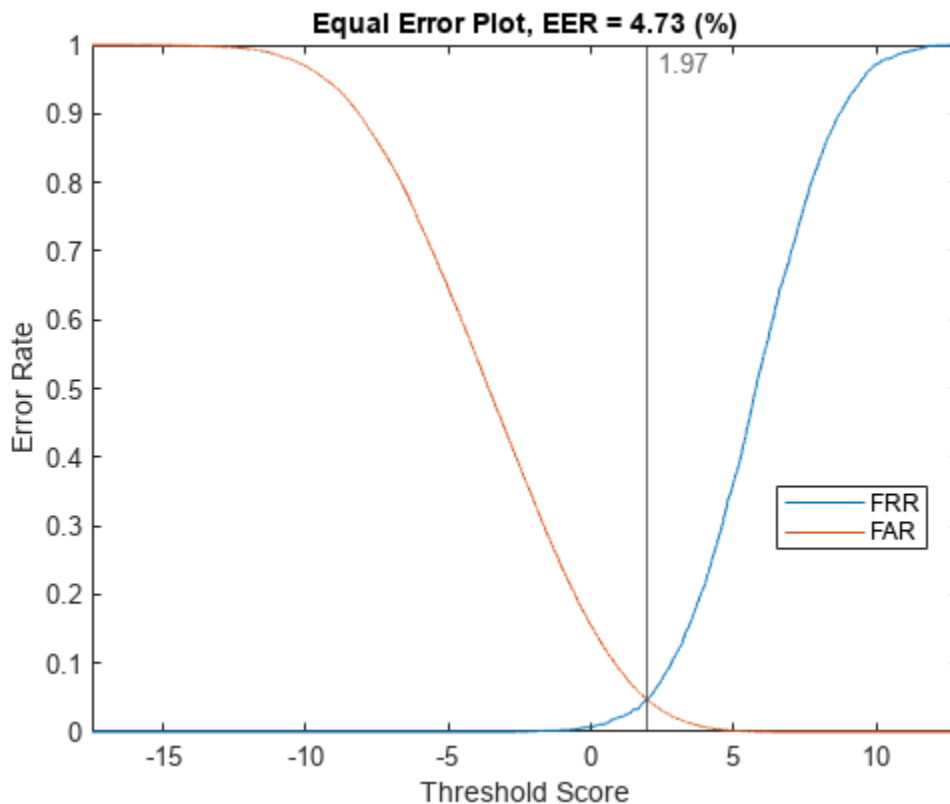
Plot the target and non-target score distributions for the group.

```
plotScoreDistributions(targetScores,nontargetScores,Analyze="group")
```



Plot the equal error rate of this new system. The equal error rate after applying adaptive s-norm1 is approximately 5 %. The equal error rate prior to adaptive s-norm1 is approximately 8 %.

```
plotEER(targetScores,nontargetScores,Analyze="group")
```



Supporting Functions

Load Dataset

```
function [adsEnroll,adsTest,adsDET,adsImposter] = loadDataset(targetSampleRate)
%LOADDATASET Load PTDB-TUG data set
% [adsEnroll,adsTest,adsDET,adsImposter] = loadDataset(targetSampleRate)
% downloads the PTDB-TUG data set, resamples it to the specified target
% sample rate and save the results in your current folder. The function
% then creates and returns four audioDatastore objects. The enrollment set
% includes two utterances per speaker. The imposter set does not overlap
% with the other data sets.

% Copyright 2021-2022 The MathWorks, Inc.

arguments
    targetSampleRate (1,1) {mustBeNumeric,mustBePositive}
end

downloadFolder = matlab.internal.examples.downloadSupportFile("audio","ptdb-tug.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
dataset = fullfile(dataFolder,"ptdb-tug");

% Resample the dataset and save to current folder if it doesn't already
% exist.
```

```

if ~isfolder(fullfile(pwd,"MIC"))
    ads = audioDatastore(fullfile(dataset,"SPEECH DATA","FEMALE","MIC"),fullfile(dataset,"SPEECH DATA","FEMALE","MIC"),
        IncludeSubfolders=true, ...
        FileExtensions=".wav", ...
        LabelSource="foldernames");
    reduceDataset = false;
    if reduceDataset
        ads = splitEachLabel(ads,55);
    end
    adsTransform = transform(ads,@(x,y)fileResampler(x,y,targetSampleRate),IncludeInfo=true);

    writeall(adsTransform,pwd,OutputFormat="flac",UseParallel=~isempty(ver("parallel")))

end

% Create a datastore that points to the resampled dataset. Use the folder
% names as the labels.
ads = audioDatastore(fullfile(pwd,"MIC"),IncludeSubfolders=true,LabelSource="foldernames");

% Split the data set into enrollment, test, DET, and imposter sets.
imposterLabels = categorical(["M05","M10","F05","F10"]);
adsImposter = subset(ads,ismember(ads.Labels,imposterLabels));

adsDev = subset(ads,~ismember(ads.Labels,imposterLabels));
rng default
numToEnroll = 2;
[adsEnroll,adsDev] = splitEachLabel(adsDev,numToEnroll);

numToTest = 50;
[adsTest,adsDET] = splitEachLabel(adsDev,numToTest);
end

```

File Resampler

```

function [audioOut,adsInfo] = fileResampler(audioIn,adsInfo,targetSampleRate)
%FILERESAMPLER Resample audio files
% [audioOut,adsInfo] = fileResampler(audioIn,adsInfo,targetSampleRate)
% resamples the input audio to the target sample rate and updates the info
% passed through the datastore.

% Copyright 2021 The MathWorks, Inc.

arguments
    audioIn (:,1) {mustBeA(audioIn,["single","double"])}
    adsInfo (1,1) {mustBeA(adsInfo,"struct")}
    targetSampleRate (1,1) {mustBeNumeric,mustBePositive}
end

% Isolate the original sample rate
originalSampleRate = adsInfo.SampleRate;

% Resample if necessary
if originalSampleRate ~= targetSampleRate
    audioOut = resample(audioIn,targetSampleRate,originalSampleRate);
    amax = max(abs(audioOut));
    if max(amax>1)
        audioOut = audioOut./amax;
    end
end

```

```
end
```

```
% Update the info passed through the datastore  
adsInfo.SampleRate = targetSampleRate;
```

```
end
```

Score Targets and Non-Targets

```
function [targetScores,nontargetScores] = scoreTargets(e,t,targetMap,nvars)  
%SCORETARGETS Score i-vector pairs  
% [targetScores,nontargetScores] = scoreTargets(e,t,targetMap) exhaustively  
% scores i-vectors in e against i-vectors in t. Specify e as an M-by-N  
% matrix, where M corresponds to the i-vector dimension, and N corresponds  
% to the number of i-vectors in e. Specify t as an M-by-P matrix, where P  
% corresponds to the number of i-vectors in t. Specify targetMap as a  
% P-by-N logical matrix that maps which i-vectors in e and t are target  
% pairs (derived from the same speaker) and which i-vectors in e and t  
% are non-target pairs (derived from different speakers). The  
% outputs, targetScores and nontargetScores, are N-element cell arrays.  
% Each cell contains a vector of scores between the i-vector in e and  
% either all the targets or nontargets in t.  
%  
% [targetScores,nontargetScores] =  
% scoreTargets(e,t,targetMap,NormFactorsSe=NFSe,NormFactorsSt=NFSt)  
% normalizes the scores by the specified normalization statistics contained  
% in structs NFSe and NFSt. If unspecified, no normalization is applied.  
  
% Copyright 2021 The MathWorks, Inc.
```

arguments

```
e (:,:) {mustBeA(e,["single","double"])}  
t (:,:) {mustBeA(t,["single","double"])}  
targetMap (:,:) {mustBeA(targetMap,"logical")}  
nvars.NormFactorsSe = [];  
nvars.NormFactorsSt = [];
```

```
end
```

```
% Score the i-vector pairs  
scores = cosineSimilarityScore(e,t);
```

```
% Apply as-norm1 if normalization factors supplied
```

```
if ~isempty(nvars.NormFactorsSe) && ~isempty(nvars.NormFactorsSt)  
    scores = 0.5*( (scores - nvars.NormFactorsSe.mu)./nvars.NormFactorsSe.std + (scores - nvars.NormFactorsSt.mu)./nvars.NormFactorsSt.std);  
end
```

```
% Separate the scores into targets and non-targets
```

```
targetScores = cell(size(targetMap,2),1);  
nontargetScores = cell(size(targetMap,2),1);  
for ii = 1:size(targetMap,2)  
    targetScores{ii} = scores(targetMap(:,ii),ii);  
    nontargetScores{ii} = scores(~targetMap(:,ii),ii);  
end
```

```
end  
end
```

Cosine Similarity Score (CSS)

```
function scores = cosineSimilarityScore(a,b)
%COSINESIMILARITYSCORE Cosine similarity score
% scores = cosineSimilarityScore(a,b) scores matrix of i-vectors, a,
% against matrix of i-vectors b. Specify a as an M-by-N matrix of
% i-vectors. Specify b as an M-by-P matrix of i-vectors. scores is returned
% as a P-by-N matrix, where columns corresponds the i-vectors in a
% and rows corresponds to the i-vectors in b and the elements of the array
% are the cosine similarity scores between them.

% Copyright 2021 The MathWorks, Inc.

arguments
    a (:,:) {mustBeA(a,["single","double"])}
    b (:,:) {mustBeA(b,["single","double"])}
end

scores = squeeze(sum(a.*reshape(b,size(b,1),1,[]),1)./(vecnorm(a).*reshape(vecnorm(b),1,1,[])));
scores = scores';
end
```

Plot Score Distributions

```
function plotScoreDistributions(targetScores,nontargetScores,nvars)
%PLOTScoreDISTRIBUTIONS Plot target and non-target score distributions
% plotScoreDistribution(targetScores,nontargetScores) plots empirical
% estimations of the distribution for target scores and nontarget scores.
% Specify targetScores and nontargetScores as cell arrays where each
% element contains a vector of speaker-specific scores.
%
% plotScoreDistributions(targetScores,nontargetScores,Analyze=ANALYZE)
% specifies the scope for analysis as either 'label' or 'group'. If ANALYZE
% is set to 'label', then a score distribution plot is created for each
% label. If ANALYZE is set to 'group', then a score distribution plot is
% created for the entire group by combining scores across speakers. If
% unspecified, ANALYZE defaults to 'group'.

% Copyright 2021 The MathWorks, Inc.

arguments
    targetScores (1,:) cell
    nontargetScores (1,:) cell
    nvars.Analyze (1,:) char {mustBeMember(nvars.Analyze,{'label','group'})} = 'group'
end

% Combine all scores to determine good bins for analyzing both the target
% and non-target scores together.
allScores = cat(1,targetScores{:},nontargetScores{:});
[~,edges] = histcounts(allScores);

% Determine the center of each bin for plotting purposes.
centers = movmedian(edges(:),2,Endpoints="discard");

if strcmpi(nvars.Analyze,"group")
    % Plot the score distributions for the group.
```

```
targetScoresBinCounts = histcounts(cat(1,targetScores{:}),edges);
targetScoresBinProb = targetScoresBinCounts(:)./sum(targetScoresBinCounts);
nontargetScoresBinCounts = histcounts(cat(1,nontargetScores{:}),edges);
nontargetScoresBinProb = nontargetScoresBinCounts(:)./sum(nontargetScoresBinCounts);

figure
plot(centers,[targetScoresBinProb,nontargetScoresBinProb])
title("Score Distributions")
xlabel("Score")
ylabel("Probability")
legend(["target","non-target"],Location="northwest")
axis tight
```

else

```
% Create a tiled layout and plot the score distributions for each speaker.
N = numel(targetScores);
tiledlayout(N,1)
for ii = 1:N
    targetScoresBinCounts = histcounts(targetScores{ii},edges);
    targetScoresBinProb = targetScoresBinCounts(:)./sum(targetScoresBinCounts);
    nontargetScoresBinCounts = histcounts(nontargetScores{ii},edges);
    nontargetScoresBinProb = nontargetScoresBinCounts(:)./sum(nontargetScoresBinCounts);

    nexttile
    hold on
    plot(centers,[targetScoresBinProb,nontargetScoresBinProb])
    title("Score Distribution for Speaker " + string(ii))
    xlabel("Score")
    ylabel("Probability")
    legend(["target","non-target"],Location="northwest")
    axis tight
end
end
end
```

Plot Equal Error Rate (EER)

```
function plotEER(targetScores,nontargetScores,nvars)
%PLOTEER Plot equal error rate (EER)
% plotEER(targetScores,nontargetScores) creates an equal error rate plot
% using the target scores and the non-target scores. Specify targetScores
% and nontargetScores as cell arrays where each element contains a vector
% of speaker-specific scores.
%
% plotEER(targetScores,nontargetScores,Analyze=ANALYZE) specifies the
% scope for analysis as either 'label' or 'group'. If ANALYZE is set to
% 'label', then an equal error rate plot is created for each label. If
% ANALYZE is set to 'group', then an equal error rate plot is created for
% the entire group by combining scores across speakers. If unspecified,
% ANALYZE defaults to 'group'.

% Copyright 2021 The MathWorks, Inc.

arguments
    targetScores (1,:) cell
    nontargetScores (1,:) cell
```



```

    nvars.Analyze (1,:) char {mustBeMember(nvars.Analyze,{'label','group'})} = 'group'
end

% Combine all scores to determine good bins for analyzing both the target
% and non-target scores together.
allScores = cat(1,targetScores{:},nontargetScores{:});
[~,edges] = histcounts(allScores,BinWidth=0.002);

% Determine the center of each bin for plotting purposes.
centers = movmedian(edges(:),2,Endpoints="discard");

if strcmpi(nvars.Analyze,"group")
    % Plot the equal error rate for the group.

    targetScoresBinCounts = histcounts(cat(1,targetScores{:}),edges);
    targetScoresBinProb = targetScoresBinCounts(:)./sum(targetScoresBinCounts);
    nontargetScoresBinCounts = histcounts(cat(1,nontargetScores{:}),edges);
    nontargetScoresBinProb = nontargetScoresBinCounts(:)./sum(nontargetScoresBinCounts);

    targetScoresCDF = cumsum(targetScoresBinProb);
    nontargetScoresCDF = cumsum(nontargetScoresBinProb,"reverse");
    [~,idx] = min(abs(targetScoresCDF(:) - nontargetScoresCDF));

    figure
    plot(centers,[targetScoresCDF,nontargetScoresCDF])
    xline(centers(idx),"-",num2str(centers(idx),3),LabelOrientation="horizontal")
    legend(["FRR","FAR"],Location="best")
    xlabel("Threshold Score")
    ylabel("Error Rate")
    title(sprintf("Equal Error Plot, EER = %0.2f (%)",100*mean([targetScoresCDF(idx);nontargetScoresCDF(idx)])))
    axis tight
else
    % Create a tiled layout and plot the equal error rate for each speaker.
    N = numel(targetScores);
    f = figure;
    tiledlayout(f,N,1,Padding="tight",TileSpacing="tight")
    for ii = 1:N
        targetScoresBinCounts = histcounts(targetScores{ii},edges);
        targetScoresBinProb = targetScoresBinCounts(:)./sum(targetScoresBinCounts);
        nontargetScoresBinCounts = histcounts(nontargetScores{ii},edges);
        nontargetScoresBinProb = nontargetScoresBinCounts(:)./sum(nontargetScoresBinCounts);

        targetScoresCDF = cumsum(targetScoresBinProb);
        nontargetScoresCDF = cumsum(nontargetScoresBinProb,"reverse");
        [~,idx] = min(abs(targetScoresCDF(:) - nontargetScoresCDF));

        nexttile
        plot(centers,[targetScoresCDF,nontargetScoresCDF])
        xline(centers(idx),"-",num2str(centers(idx),3),LabelOrientation="horizontal")
        legend(["FRR","FAR"],Location="southwest")
        xlabel("Threshold Score")
        ylabel("Error Rate")
        title(sprintf("Equal Error Plot for Speaker " + string(ii) + ", EER = %0.2f (%)", ...
            100*mean([targetScoresCDF(idx);nontargetScoresCDF(idx)])))
        axis tight
    end
end

```

```
end  
end
```

Get Norm Factors

```
function normFactors = getNormFactors(w,imposterCohort,nvars)  
%GETNORMFACTORS Get norm factors  
% normFactors = getNormFactors(w,imposterCohort) returns the mean and  
% standard deviation of the scores between the i-vectors in w and the i-vectors  
% in the imposter cohort. Specify w as a matrix of i-vectors. Specify  
% imposterCohort as a matrix of i-vectors. Each column corresponds to an  
% i-vector the same length as w.  
%  
% normFactors = getNormFactors(w,imposterCohort,TopK=TOPK) calculates the  
% normalization statistics using only the top K highest scores. If  
% unspecified, all scores are used.  
  
% Copyright 2021 The MathWorks, Inc.  
  
arguments  
    w (:,:) {mustBeA(w,["single","double"])}  
    imposterCohort (:,:) {mustBeA(imposterCohort,["single","double"])}  
    nvars.TopK (1,1) {mustBePositive} = inf  
end  
topK = min(ceil(nvars.TopK),size(imposterCohort,2));  
  
% Score the template i-vector against the imposter cohort.  
imposterScores = cosineSimilarityScore(w,imposterCohort);  
  
% Isolate the top K scores.  
imposterScores = sort(imposterScores,"descend");  
imposterScores = imposterScores(1:topK,:);  
  
% Calculate the score normalization statistics  
MU = mean(imposterScores,1);  
STD = std(imposterScores,[],1);  
  
% Return normalization statistics as a struct  
normFactors = struct(mu=MU,std=STD);  
end
```

References

- [1] Matejka, Pavel, Ondrej Novotny, Oldrich Plchot, Lukas Burget, Mireia Diez Sanchez, and Jan Cernocky. "Analysis of Score Normalization in Multilingual Speaker Recognition." *Interspeech 2017*, 2017. <https://doi.org/10.21437/interspeech.2017-803>.
- [2] van Leeuwen, David A., and Niko Brummer. "An Introduction to Application-Independent Evaluation of Speaker Recognition Systems." *Lecture Notes in Computer Science*, 2007, 330-53. https://doi.org/10.1007/978-3-540-74200-5_19.

[3] G. Pirker, M. Wohlmayr, S. Petrik, and F. Pernkopf, "A Pitch Tracking Corpus with Evaluation on Multipitch Tracking Scenario", Interspeech, pp. 1509-1512, 2011.

i-vector Score Calibration

An i-vector system outputs a raw score specific to the data and parameters used to develop the system. This makes interpreting the score and finding a consistent decision threshold for verification tasks difficult.

To address these difficulties, researchers developed *score normalization* and *score calibration* techniques.

- In *score normalization*, raw scores are normalized in relation to an 'imposter cohort'. Score normalization occurs before evaluating the detection error tradeoff and can improve the accuracy of a system and its ability to adapt to new data.
- In *score calibration*, raw scores are mapped to probabilities, which in turn are used to better understand the system's confidence in decisions.

In this example, you apply score calibration to an i-vector system. To learn about score normalization, see "i-vector Score Normalization" on page 1-607.

For example purposes, you use cosine similarity scoring (CSS) throughout this example. The interpretability of probabilistic linear discriminant analysis (PLDA) scoring is also improved by calibration.

Starting in R2022a, you can use the `calibrate` method of `ivectorSystem` to calibrate both CSS and PLDA scoring.

Download i-vector System and Data Set

To download a pretrained i-vector system suitable for speaker recognition, call `speakerRecognition`. The `ivectorSystem` returned was trained on the LibriSpeech data set, which consists of English-language 16 kHz recordings.

```
ivs = speakerRecognition;
```

Download the PTDB-TUG data set [1] on page 1-641. The supporting function, `loadDataset` on page 1-633, downloads the data set and then resamples it from 48 kHz to 16 kHz, which is the sample rate that the i-vector system was trained on. The `loadDataset` function returns these `audioDatastore` objects:

- `adsEnroll` - Contains files to enroll speakers into the i-vector system.
- `adsDev` - Contains a large set of files to analyze the detection error tradeoff of the i-vector system, and to spot-check performance.
- `adsCalibrate` - Contains a set of speakers used to calibrate the i-vector system. The calibration set does not overlap with the enroll and dev sets.

```
targetSampleRate = ivs.SampleRate;  
[adsEnroll,adsDev,adsCalibrate] = loadDataset(targetSampleRate);
```

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

Score Calibration

In score calibration, you apply a warping function to scores so that they are more easily and consistently interpretable as measures of confidence. Generally, score calibration has no effect on the

performance of a verification system because the mapping is an affine transformation. The two most popular approaches to calibration are Platt scaling and isotonic regression. Isotonic regression usually results in better performance, but is more prone to overfitting if the calibration data is too small [2] on page 1-641.

In this example, you perform calibration using both Platt scaling and isotonic regression, and then compare the calibrations using reliability diagrams.

Extract i-vectors

To properly calibrate a system, you must use data that does not overlap with the evaluation data. Extract i-vectors from the calibration set. You will use these i-vectors to create a calibration warping function.

```
calibrationIvecs = ivector(ivs,adsCalibrate);
```

Score i-vector Pairs

You will score each i-vector against each other i-vector to create a matrix of scores, some of which correspond to target scores where both i-vectors belong to the same speaker, and some of which correspond to non-target scores where the i-vectors belong to two different speakers. First, create a `targets` matrix to keep track of which scores are target and which are non-target.

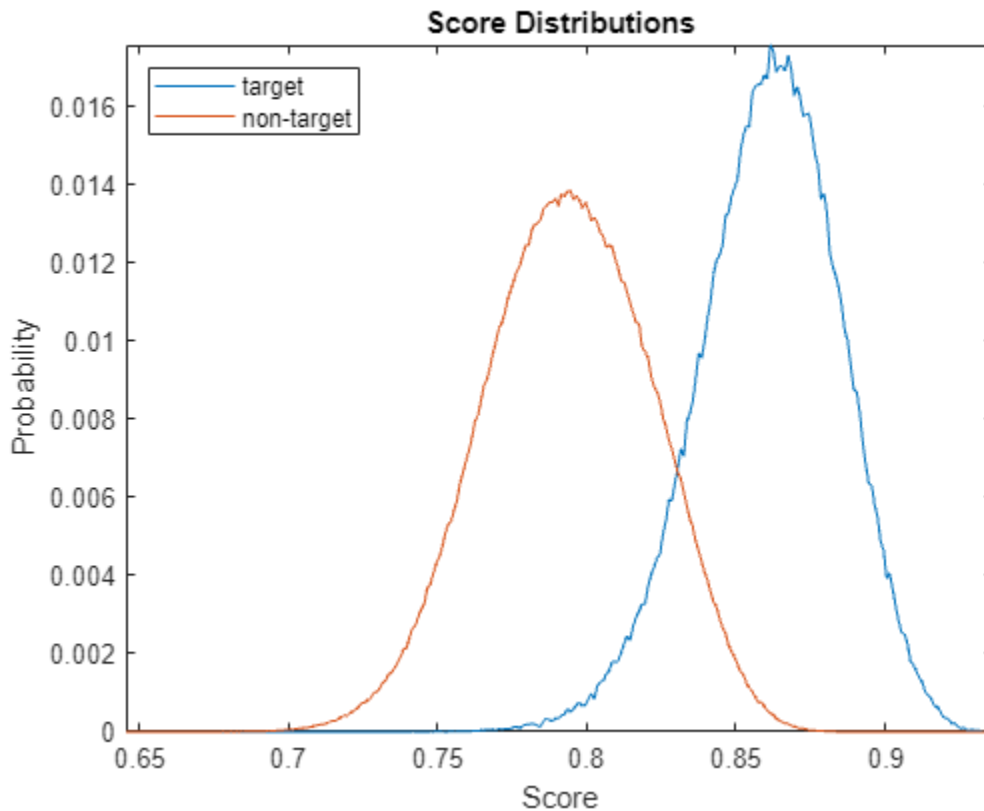
```
targets = true(size(calibrationIvecs,2),size(calibrationIvecs,2));
calibrationLabels = adsCalibrate.Labels;
for ii = 1:size(calibrationIvecs,2)
    targets(:,ii) = ismember(calibrationLabels,calibrationLabels(ii));
end
```

Discard the target scores that corresponds to the i-vector scored with itself by setting the corresponding value in the target matrix to `NaN`. The supporting function, `scoreTargets` on page 1-635, scores each valid i-vector pair and returns the results in cell arrays of target and non-target scores.

```
targets = targets + diag(diag(targets)*nan);
[targetScores,nontargetScores] = scoreTargets(calibrationIvecs,calibrationIvecs,targets);
```

Use the supporting function, `plotScoreDistributions` on page 1-636, to plot the target and non-target score distributions for the group. The scores range from around 0.64 to 1. In a properly calibrated system, scores should range from 0 to 1. The job of calibrating a binary classification system is to map the raw score to a score between 0 and 1. The calibrated score should be interpretable as the probability that the score corresponds to a target pair.

```
plotScoreDistributions(targetScores,nontargetScores,Analyze="group")
```



Platt Scaling

Platt scaling (also referred to as Platt calibration or logistic regression) works by fitting a logistic regression model to a classifier's scores.

The supporting function `logistic` on page 1-639 implements a general logistic function defined as

$$p(x) = \frac{1}{1 + e^{(B + Ax)}}$$

where A and B are the scalar learned parameters.

The supporting function `logRegCost` on page 1-639 defines the cost function for logistic regression as defined in [3] on page 1-641:

$$\operatorname{argmin}_{A, B} \left\{ -\sum_i y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \right\}$$

As described in [3] on page 1-641, the target values are modified from 0 and 1 to avoid overfitting:

$$y_+ = \frac{N_+ + 1}{N_+ + 2}; y_- = \frac{1}{N_- + 2}$$

where y_+ is the positive sample value and N_+ is the number of positive samples, and y_- is the negative sample value and N_- is the number of negative samples.

Create a vector of the raw target and non-target scores.

```
tS = cat(1,targetScores{:});
ntS = cat(1,nontargetScores{:});
x = [tS;ntS];
```

Create a vector of ideal target probabilities.

```
yplus = (numel(tS) + 1)./(numel(tS) + 2);
yminus = 1./(numel(ntS) + 2);
y = [yplus*ones(numel(tS),1);yminus*ones(numel(ntS),1)];
```

Use `fminsearch` to find the values of A and B that minimize the cost function.

```
init = [1,1];
AB = fminsearch(@(AB)logRegCost(y,x,AB),init);
```

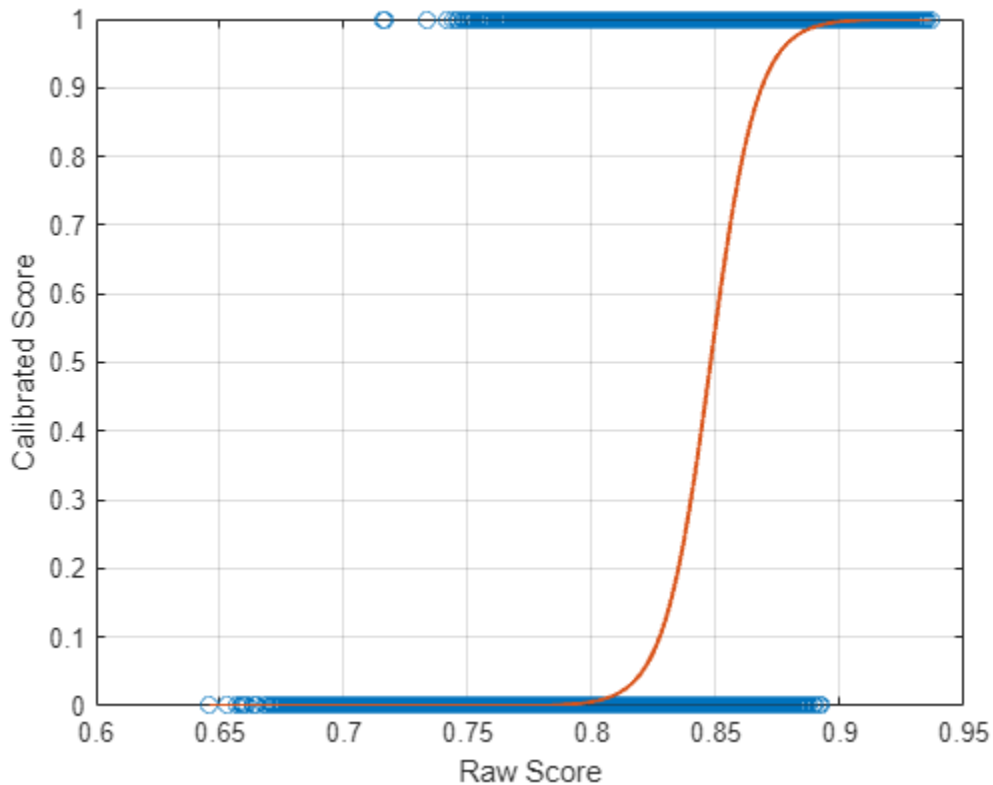
Sort the scores in ascending order for visualization purposes.

```
[x,idx] = sort(x,"ascend");
trueLabel = [ones(numel(tS),1);zeros(numel(ntS),1)];
trueLabel = trueLabel(idx);
```

Use the supporting function `calibrateScores` on page 1-637 to calibrate the raw scores. Plot the warping function that maps the raw scores to the calibrated scores. Also plot the target scores you are modeling.

```
calibratedScores = calibrateScores(x,AB);

plot(x,trueLabel,"o")
hold on
plot(x,calibratedScores,LineWidth=1.5)
grid on
xlabel("Raw Score")
ylabel("Calibrated Score")
hold off
```



Isotonic Regression

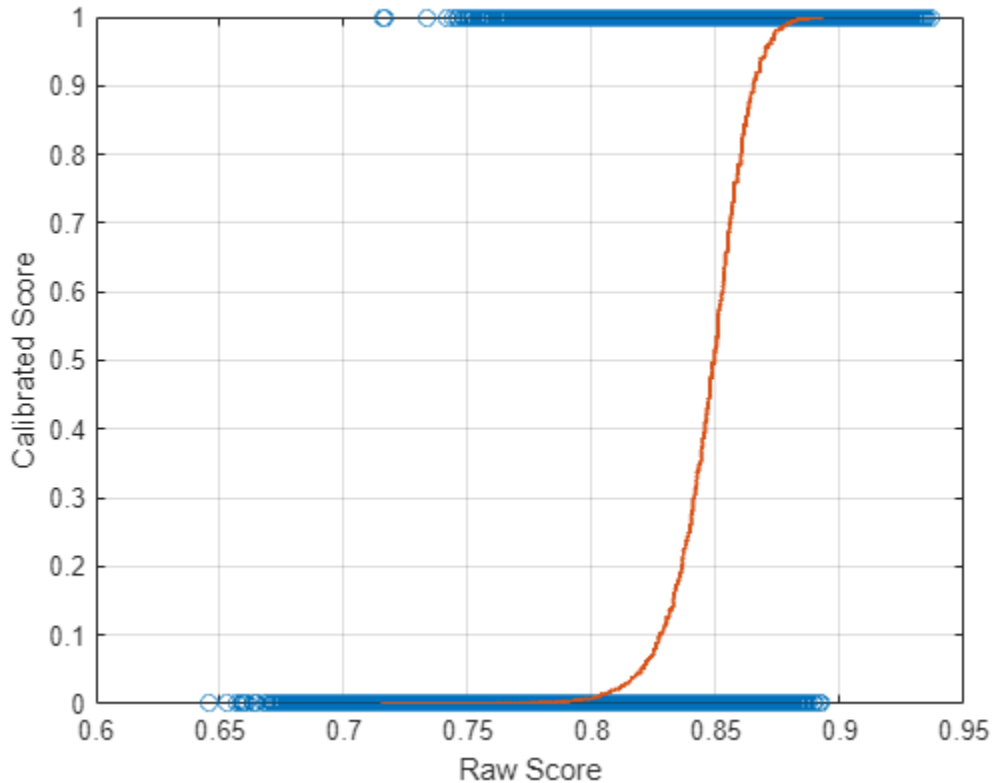
Isotonic regression fits a free-form line to observations with the only condition being that it is non-decreasing (or non-increasing). The supporting function `isotonicRegression` on page 1-640 uses the pool adjacent violators (PAV) algorithm [3] on page 1-641 for isotonic regression.

Call `isotonicRegression` with the raw score and true labels. The function outputs a struct containing a map between raw scores and calibrated scores.

```
scoringMap = isotonicRegression(x,trueLabel);
```

Plot the raw score against the calibrated score. The line is the learned isotonic fit. The circles are the data you are fitting.

```
plot(x,trueLabel,"o")
hold on
plot(scoringMap.Raw,scoringMap.Calibrated,LineWidth=1.5)
grid on
xlabel("Raw Score")
ylabel("Calibrated Score")
hold off
```

Reliability Diagram

Reliability diagrams reveal reliability by plotting the mean of the predicted value against the known fraction of positives. A system is reliable if the mean of the predicted value is equal to the fraction of positives [4] on page 1-641.

Reliability must be assessed using a different data set than the one used to calibrate the system. Extract i-vectors from the development data set, `adsDev`. The development data set has no speaker overlap with the calibration data set.

```
devIvecs = ivector(ivs,adsDev);
```

Create a `targets` map and score all i-vector pairs.

```
devLabels = adsDev.Labels;
targets = true(size(devIvecs,2),size(devIvecs,2));
for ii = 1:size(devIvecs,2)
    targets(:,ii) = ismember(devLabels,devLabels(ii));
end
targets = targets + diag(diag(targets)*nan);
```

```
[targetScores,nontargetScores] = scoreTargets(devIvecs,devIvecs,targets);
```

Combine all the scores and labels for faster processing.

```
ts = cat(1,targetScores{:});
nts = cat(1,nontargetScores{:});
```

```
scores = [ts;nts];
trueLabels = [true(numel(ts),1);false(numel(nts),1)];
```

Calibrate the scores using Platt scaling.

```
calibratedScoresPlattScaling = calibrateScores(scores,AB);
```

Calibrate the scores using isotonic regression.

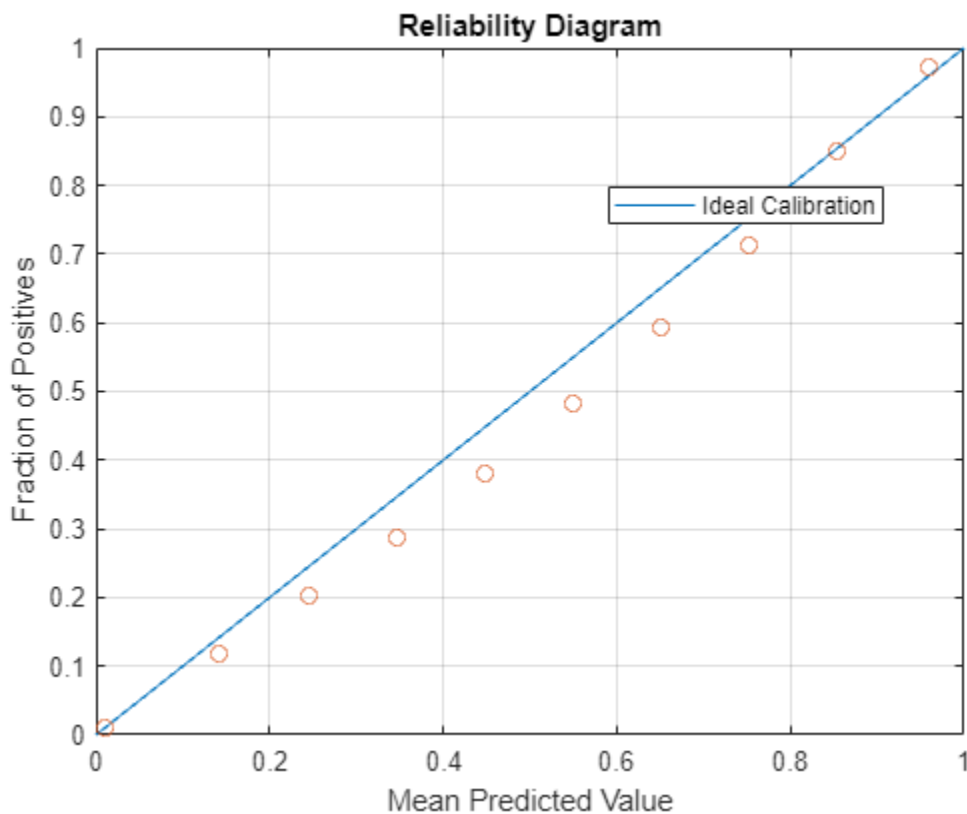
```
calibratedScoresIsotonicRegression = calibrateScores(scores,scoringMap);
```

When interpreting the reliability diagram, values below the diagonal indicate that the system is giving higher probability scores than it should be, and values above the diagonal indicate the system is giving lower probability scores than it should. In both cases, increasing the amount of calibration data, and using calibration data like the target application, should improve performance.

```
numBins = 10  ;
```

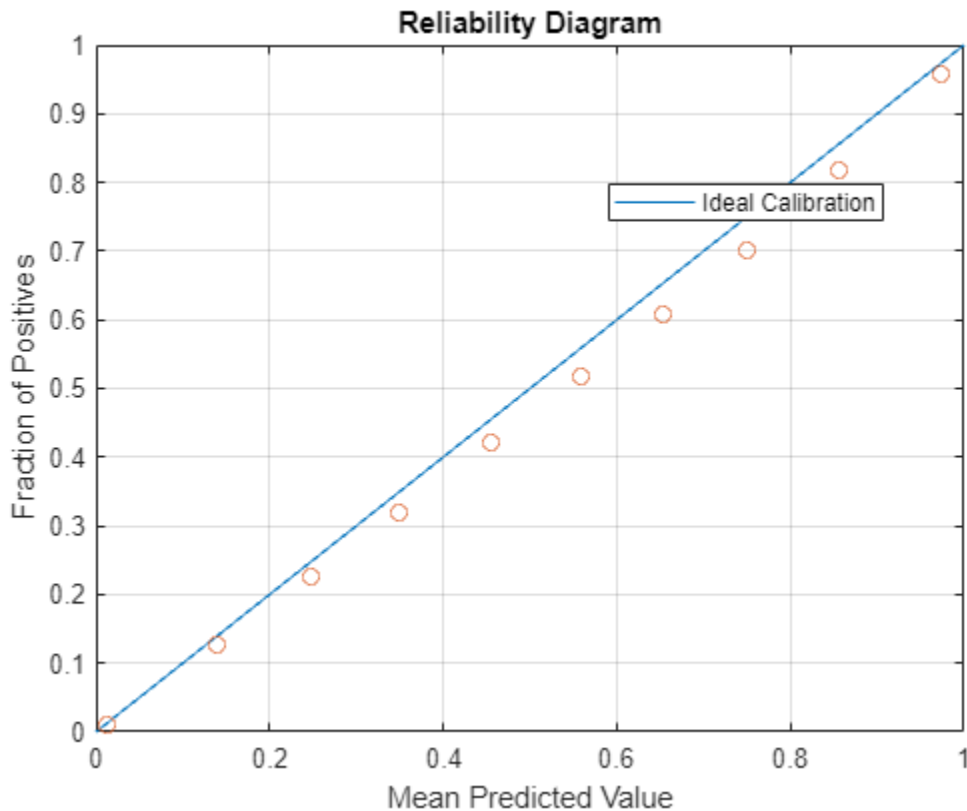
Plot the reliability diagram for the i-vector system calibrated using Platt scaling.

```
reliabilityDiagram(trueLabels,calibratedScoresPlattScaling,numBins)
```



Plot the reliability diagram for the i-vector system calibrated using isotonic regression.

```
reliabilityDiagram(trueLabels,calibratedScoresIsotonicRegression,numBins)
```



Supporting Functions

Load Dataset

```
function [adsEnroll,adsDev,adsCalibrate] = loadDataset(targetSampleRate)
%LOADDATASET Load PTDB-TUG data set
% [adsEnroll,adsDev,adsCalibrate] = loadDataset(targetSampleRate)
% downloads the PTDB-TUG data set, resamples it to the specified target
% sample rate and save the results in your current folder. The function
% then creates and returns three audioDatastore objects. The enrollment set
% includes two utterances per speaker. The calibrate set does not overlap
% with the other data sets.

% Copyright 2021-2022 The MathWorks, Inc.

rng(0)
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","ptdb-tug.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
dataset = fullfile(dataFolder,"ptdb-tug");

% Resample the dataset and save to current folder if it doesn't already
% exist.
if ~isfolder(fullfile(pwd,"MIC"))
    ads = audioDatastore([fullfile(dataset,"SPEECH DATA","FEMALE","MIC"),fullfile(dataset,"SPEECH
        IncludeSubfolders=true, ...
```

```

        FileExtensions=".wav", ...
        LabelSource="foldernames");
    reduceDataset = false;
    if reduceDataset
        ads = splitEachLabel(ads,10);
    end
    adsTransform = transform(ads,@(x,y)fileResampler(x,y,targetSampleRate),IncludeInfo=true);
    writeall(adsTransform,pwd,OutputFormat="flac",UseParallel=canUseParallelPool)
end

% Create a datastore that points to the resampled dataset. Use the folder
% names as the labels.
ads = audioDatastore(fullfile(pwd,"MIC"),IncludeSubfolders=true,LabelSource="foldernames");

% Split the data set into enrollment, development, and calibration sets.
calibrationLabels = categorical(["M01","M03","M05","M7","M9","F01","F03","F05","F07","F09"]);

adsCalibrate = subset(ads,ismember(ads.Labels,calibrationLabels));

adsDev = subset(ads,~ismember(ads.Labels,calibrationLabels));

numToEnroll = 2;
[adsEnroll,adsDev] = splitEachLabel(adsDev,numToEnroll);

end

```

File Resampler

```

function [audioOut,adsInfo] = fileResampler(audioIn,adsInfo,targetSampleRate)
%FILERESAMPLER Resample audio files
% [audioOut,adsInfo] = fileResampler(audioIn,adsInfo,targetSampleRate)
% resamples the input audio to the target sample rate and updates the info
% passed through the datastore.

% Copyright 2021 The MathWorks, Inc.

arguments
    audioIn (:,1) {mustBeA(audioIn,["single","double"])}
    adsInfo (1,1) {mustBeA(adsInfo,"struct")}
    targetSampleRate (1,1) {mustBeNumeric,mustBePositive}
end

% Isolate the original sample rate
originalSampleRate = adsInfo.SampleRate;

% Resample if necessary
if originalSampleRate ~= targetSampleRate
    audioOut = resample(audioIn,targetSampleRate,originalSampleRate);
    amax = max(abs(audioOut));
    if max(amax>1)
        audioOut = audioOut./amax;
    end
end

% Update the info passed through the datastore
adsInfo.SampleRate = targetSampleRate;

end

```

Score Targets and Non-Targets

```

function [targetScores,nontargetScores] = scoreTargets(e,t,targetMap,nvars)
%SCORETARGETS Score i-vector pairs
% [targetScores,nontargetScores] = scoreTargets(e,t,targetMap) exhaustively
% scores i-vectors in e against i-vectors in t. Specify e as an M-by-N
% matrix, where M corresponds to the i-vector dimension, and N corresponds
% to the number of i-vectors in e. Specify t as an M-by-P matrix, where P
% corresponds to the number of i-vectors in t. Specify targetMap as a
% P-by-N numeric matrix that maps which i-vectors in e and t are target
% pairs (derived from the same speaker) and which i-vectors in e and t are
% non-target pairs (derived from different speakers). Values in targetMap
% set to NaN are discarded. The outputs, targetScores and nontargetScores,
% are N-element cell arrays. Each cell contains a vector of scores between
% the i-vector in e and either all the targets or nontargets in t.
%
% [targetScores,nontargetScores] =
% scoreTargets(e,t,targetMap,NormFactorsSe=NFSe,NormFactorsSt=NFSt)
% normalizes the scores by the specified normalization statistics contained
% in structs NFSe and NFSt. If unspecified, no normalization is applied.

% Copyright 2021 The MathWorks, Inc.

arguments
    e (:,:) {mustBeA(e,["single","double"])}
    t (:,:) {mustBeA(t,["single","double"])}
    targetMap (:,:)
    nvars.NormFactorsSe = [];
    nvars.NormFactorsSt = [];
end

% Score the i-vector pairs
scores = cosineSimilarityScore(e,t);

% Apply as-norm1 if normalization factors supplied.
if ~isempty(nvars.NormFactorsSe) && ~isempty(nvars.NormFactorsSt)
    scores = 0.5*( (scores - nvars.NormFactorsSe.mu)./nvars.NormFactorsSe.std + (scores - nvars.NormFactorsSt.mu)./nvars.NormFactorsSt.std );
end

% Separate the scores into targets and non-targets
targetScores = cell(size(targetMap,2),1);
nontargetScores = cell(size(targetMap,2),1);
removeIndex = isnan(targetMap);
for ii = 1:size(targetMap,2)
    localScores = scores(:,ii);
    localMap = targetMap(:,ii);
    localScores(removeIndex(:,ii)) = [];
    localMap(removeIndex(:,ii)) = [];

    targetScores{ii} = localScores(logical(localMap));
    nontargetScores{ii} = localScores(~logical(localMap));
end
end

```

Cosine Similarity Score (CSS)

```
function scores = cosineSimilarityScore(a,b)
%COSINESIMILARITYSCORE Cosine similarity score
% scores = cosineSimilarityScore(a,b) scores matrix of i-vectors, a,
% against matrix of i-vectors b. Specify a as an M-by-N matrix of
% i-vectors. Specify b as an M-by-P matrix of i-vectors. scores is returned
% as a P-by-N matrix, where columns corresponds the i-vectors in a
% and rows corresponds to the i-vectors in b and the elements of the array
% are the cosine similarity scores between them.

% Copyright 2021 The MathWorks, Inc.

arguments
    a (:,:) {mustBeA(a,["single","double"])}
    b (:,:) {mustBeA(b,["single","double"])}
end

scores = squeeze(sum(a.*reshape(b,size(b,1),1,[]),1)./(vecnorm(a).*reshape(vecnorm(b),1,1,[])));
scores = scores';
end
```

Plot Score Distributions

```
function plotScoreDistributions(targetScores,nontargetScores,nvars)
%PLOTScoreDISTRIBUTIONS Plot target and non-target score distributions
% plotScoreDistribution(targetScores,nontargetScores) plots empirical
% estimations of the distribution for target scores and nontarget scores.
% Specify targetScores and nontargetScores as cell arrays where each
% element contains a vector of speaker-specific scores.
%
% plotScoreDistributions(targetScores,nontargetScores,Analyze=ANALYZE)
% specifies the scope for analysis as either "label" or "group". If ANALYZE
% is set to "label", then a score distribution plot is created for each
% label. If ANALYZE is set to "group", then a score distribution plot is
% created for the entire group by combining scores across speakers. If
% unspecified, ANALYZE defaults to "group".

% Copyright 2021 The MathWorks, Inc.

arguments
    targetScores (1,:) cell
    nontargetScores (1,:) cell
    nvars.Analyze (1,:) char {mustBeMember(nvars.Analyze,["label","group"])} = "group"
end

% Combine all scores to determine good bins for analyzing both the target
% and non-target scores together.
allScores = cat(1,targetScores{:},nontargetScores{:});
[~,edges] = histcounts(allScores);

% Determine the center of each bin for plotting purposes.
centers = movmedian(edges(:),2,Endpoints="discard");

if strcmpi(nvars.Analyze,"group")
    % Plot the score distributions for the group.
```

```

targetScoresBinCounts = histcounts(cat(1,targetScores{:}),edges);
targetScoresBinProb = targetScoresBinCounts(:)./sum(targetScoresBinCounts);
nontargetScoresBinCounts = histcounts(cat(1,nontargetScores{:}),edges);
nontargetScoresBinProb = nontargetScoresBinCounts(:)./sum(nontargetScoresBinCounts);

figure
plot(centers,[targetScoresBinProb,nontargetScoresBinProb])
title("Score Distributions")
xlabel("Score")
ylabel("Probability")
legend(["target","non-target"],Location="northwest")
axis tight

else

% Create a tiled layout and plot the score distributions for each speaker.

N = numel(targetScores);
tiledlayout(N,1)
for ii = 1:N
    targetScoresBinCounts = histcounts(targetScores{ii},edges);
    targetScoresBinProb = targetScoresBinCounts(:)./sum(targetScoresBinCounts);
    nontargetScoresBinCounts = histcounts(nontargetScores{ii},edges);
    nontargetScoresBinProb = nontargetScoresBinCounts(:)./sum(nontargetScoresBinCounts);
    nexttile
    hold on
    plot(centers,[targetScoresBinProb,nontargetScoresBinProb])
    title("Score Distribution for Speaker " + string(ii))
    xlabel("Score")
    ylabel("Probability")
    legend(["target","non-target"],Location="northwest")
    axis tight
end
end
end

```

Calibrate Scores

```

function y = calibrateScores(score,scoreMapping)
%CALIBRATESCORES Calibrate scores
% y = calibrateScores(score,scoreMapping) maps the raw scores to calibrated
% scores, y, using the score mapping information in scoreMapping.
% Specify score as a vector or matrix of raw scores. Specify score mapping
% as either struct or a two-element vector. If scoreMapping is specified as
% a struct, then it should have two fields: Raw and Calibrated, that
% together form a score mapping. If scoreMapping is specified as a vector,
% then the elements are used as the coefficients in the logistic function.
% y is returned as vector or matrix the same size as the raw scores.

% Copyright 2021 The MathWorks, Inc.

arguments
    score (:,:) {mustBeA(score,["single","double"])}
    scoreMapping
end

```

```
if isstruct(scoreMapping)
    % Calibration using isotonic regression

    rawScore = scoreMapping.Raw;
    interpretedScore = scoreMapping.Calibrated;

    n = numel(score);

    % Find the index of the raw score in the mapping closest to the score provided.
    idx = zeros(n,1);
    for ii = 1:n
        [~,idx(ii)] = min(abs(score(ii)-rawScore));
    end

    % Get the calibrated score.
    y = interpretedScore(idx);

else

    % Calibration using logistic regression
    y = logistic(score,scoreMapping);

end
end
```

Reliability Diagram

```
function reliabilityDiagram(targets,predictions,numBins)
%RELIABILITYDIAGRAM Plot reliability diagram
% reliabilityDiagram(targets,predictions) plots a reliability diagram for
% targets and predictions. Specify targets an M-by-1 logical vector.
% Specify predictions as an M-by-1 numeric vector.
%
% reliabilityDiagram(targets,predictions,numBins) specifies the number of
% bins for the reliability diagram. If unspecified, numBins defaults to 10.

% Copyright 2021 The MathWorks, Inc.

arguments
    targets (:,1) {mustBeA(targets,"logical")}
    predictions (:,1) {mustBeA(predictions,["single","double"])}
    numBins (1,1) {mustBePositive,mustBeInteger} = 10;
end

% Bin the predictions into the requested number of bins. Count the number of
% predictions per bin.
[predictionsPerBin,~,predictionsInBin] = histcounts(predictions,numBins);

% Determine the mean of the predictions in the bin.
meanPredictions = accumarray(predictionsInBin,predictions)./predictionsPerBin(:);

% Determine the mean of the targets per bin. This is the fraction of
% positives--the number of targets in the bin over the total number of
% predictions in the bin.
meanTargets = accumarray(predictionsInBin,targets)./predictionsPerBin(:);

plot([0,1],[0,1])
```



```

hold on
plot(meanPredictions,meanTargets,"o")
legend("Ideal Calibration",Location="best")
xlabel("Mean Predicted Value")
ylabel("Fraction of Positives")
title("Reliability Diagram")
grid on
hold off

end

```

Logistic Regression Cost Function

```

function cost = logRegCost(y,f,iparams)
%LOGREGCOST Logistic regression cost
% cost = logRegCost(y,f,iparams) calculates the cost of the logistic
% function given truth y, prediction f, and logistic params iparams.
% Specify y and f as column vectors. Specify iparams as a two-element row
% vector in the form [A,B], where A and B are the model parameters:
%
%
%           1
% p(x) =  -----
%          1 + e^(-A*f - B)
%
% Copyright 2021 The MathWorks, Inc.

arguments
    y (:,1) {mustBeA(y,["single","double"])}
    f (:,1) {mustBeA(f,["single","double"])}
    iparams (1,2) {mustBeA(iparams,["single","double"])}
end
p = logistic(f,iparams);
cost = -sum(y.*log(p) + (1-y).*log(1-p));
end

```

Logistic Function

```

function p = logistic(f,iparams)
%LOGISTIC Logistic function
% p = logistic(f,iparams) applies the general logistic function to input f
% with parameters iparams. Specify f as a numeric array. Specify iparams as
% a two-element vector. p is returned as the same size as f.

% Copyright 2021 The MathWorks, Inc.

arguments
    f
    iparams = [1 0];
end
p = 1./(1+exp(-iparams(1).*f - iparams(2)));
end

```

Isotonic Regression

```
function scoreMapping = isotonicRegression(x,y)
%ISOTONICREGRESSION Isotonic regression
% scoreMapping = isotonicRegression(x,y) fits a line yhat to data y under
% the monotonicity constraint that x(i)>x(j) -> yhat(i)>=yhat(j). That is,
% the values in yhat are monotonically non-decreasing with respect to x.
% The output, scoreMapping, is a struct containing the changepoints of yhat
% and the corresponding raw score in x.

% Copyright 2021, The MathWorks, Inc.

N = numel(x);

% Sort points in ascending order of x.
[x,idx] = sort(x(:),"ascend");
y = y(idx);

% Initialize fitted values to the given values.
m = y;

% Initialize blocks, one per point. These will merge and the number of
% blocks will reduce as the algorithm proceeds.
blockMap = 1:N;
w = ones(size(m));

while true

    diffs = diff(m);

    if all(diffs >= 0)

        % If all blocks are monotonic, end the loop.
        break;

    else

        % Find all positive changepoints. These are the beginnings of each
        % block.
        blockStartIndex = diffs>0;

        % Create group indices for each unique block.
        blockIndices = cumsum([1;blockStartIndex]);

        % Calculate the mean of each block and update the weights for the
        % blocks. We're merging all the points in the blocks here.
        m = accumarray(blockIndices,w.*m);
        w = accumarray(blockIndices,w);
        m = m ./ w;

        % Map which block corresponds to which index.
        blockMap = blockIndices(blockMap);

    end
end
```

```
% Broadcast merged blocks out to original points.
m = m(blockMap);

% Find the changepoints
changepoints = find(diff(m)>0);
changepoints = [changepoints;changepoints+1];
changepoints = sort(changepoints);

% Remove all points that aren't changepoints.
a = m(changepoints);
b = x(changepoints);

scoreMapping = struct(Raw=b,Calibrated=a);
end
```

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] van Leeuwen, David A., and Niko Brummer. "An Introduction to Application-Independent Evaluation of Speaker Recognition Systems." *Lecture Notes in Computer Science*, 2007, 330-53.

[3] Niculescu-Mizil, A., & Caruana, R. (2005). Predicting good probabilities with supervised learning. *Proceedings of the 22nd International Conference on Machine Learning - ICML '05*. doi:10.1145/1102351.1102430

[4] Brocker, Jochen, and Leonard A. Smith. "Increasing the Reliability of Reliability Diagrams." *Weather and Forecasting* 22, no. 3 (2007): 651-61. <https://doi.org/10.1175/waf993.1>.

Speaker Recognition Using x-vectors

Speaker recognition answers the question "Who is speaking?". Speaker recognition is usually divided into two tasks: *speaker identification* and *speaker verification*. In speaker identification, a speaker is recognized by comparing their speech to a closed set of templates. In speaker verification, a speaker is recognized by comparing the likelihood that the speech belongs to a particular speaker against a predetermined threshold. Traditional machine learning methods perform well at these tasks in ideal conditions. For examples of speaker identification using traditional machine learning methods, see "Speaker Identification Using Pitch and MFCC" on page 1-237 and "Speaker Verification Using i-Vectors" on page 1-582. Audio Toolbox™ provides `ivectorSystem` which encapsulates the ability to train an i-vector system, enroll speakers or other audio labels, evaluate the system for a decision threshold, and identify or verify speakers or other audio labels.

In adverse conditions, the deep learning approach of x-vectors has been shown to achieve state-of-the-art results for many scenarios and applications [1] on page 1-653. The x-vector system is an evolution of i-vectors originally developed for the task of speaker verification.

In this example, you develop an x-vector system. First, you train a time-delay neural network (TDNN) to perform speaker identification. Then you train the traditional backends for an x-vector-based speaker verification system: an LDA projection matrix and a PLDA model. You then perform speaker verification using the TDNN and the backend dimensionality reduction and scoring. The x-vector system backend, or classifier, is the same as developed for i-vector systems. For details on the backend, see "Speaker Verification Using i-Vectors" on page 1-582 and `ivectorSystem`.

In "Speaker Diarization Using x-vectors" on page 1-657, you use the x-vector system trained in this example to perform speaker diarization. Speaker diarization answers the question, "Who spoke when?".

Throughout this 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 a subset of the LibriSpeech Dataset [2] on page 1-653. The LibriSpeech Dataset is a large corpus of read English speech sampled at 16 kHz. The data is derived from audiobooks read from the LibriVox project. Download the 100-hour subset of the LibriSpeech training data, the clean development set, and the clean test set.

```
dataFolder = tempdir;

datasetTrain = fullfile(dataFolder, "LibriSpeech", "train-clean-100");
if ~datasetExists(datasetTrain)
    filename = "train-clean-100.tar.gz";
    url = "http://www.openslr.org/resources/12/" + filename;
    gunzip(url, dataFolder);
    unzippedFile = fullfile(dataFolder, filename);
    untar(unzippedFile{1}(1:end-3), dataFolder);
end

datasetDev = fullfile(dataFolder, "LibriSpeech", "dev-clean");
if ~datasetExists(datasetDev)
    filename = "dev-clean.tar.gz";
    url = "http://www.openslr.org/resources/12/" + filename;
    gunzip(url, dataFolder);
end
```

```

        unzippedFile = fullfile(dataFolder,filename);
        untar(unzippedFile{1}(1:end-3),dataFolder);
    end

    datasetTest = fullfile(dataFolder,"LibriSpeech","test-clean");
    if ~datasetExists(datasetTest)
        filename = "test-clean.tar.gz";
        url = "http://www.openslr.org/resources/12/" + filename;
        gunzip(url,dataFolder);
        unzippedFile = fullfile(dataFolder,filename);
        untar(unzippedFile{1}(1:end-3),dataFolder);
    end
end

```

Create `audioDatastore` objects that point to the data. The speaker labels are encoded in the file names. Split the datastore into train, validation, and test sets. You will use these sets to train, validate, and test a TDNN.

```

adsTrain = audioDatastore(datasetTrain,IncludeSubfolders=true);
adsTrain.Labels = categorical(extractBetween(adsTrain.Files,fullfile(datasetTrain,filesep),filesep));

adsDev = audioDatastore(datasetDev,IncludeSubfolders=true);
adsDev.Labels = categorical(extractBetween(adsDev.Files,fullfile(datasetDev,filesep),filesep));

adsEvaluate = audioDatastore(datasetTest,IncludeSubfolders=true);
adsEvaluate.Labels = categorical(extractBetween(adsEvaluate.Files,fullfile(datasetTest,filesep),filesep));

```

Separate the `audioDatastore` objects into five sets:

- `adsTrain` - Contains training set for the TDNN and backend classifier.
- `adsValidation` - Contains validation set to evaluate TDNN training progress.
- `adsTest` - Contains test set to evaluate the TDNN performance for speaker identification.
- `adsEnroll` - Contains enrollment set to evaluate the detection error tradeoff of the x-vector system for speaker verification.
- `adsDET` - Contains evaluation set used to determine the detection error tradeoff of the x-vector system for speaker verification.

```

[adsTrain,adsValidation,adsTest] = splitEachLabel(adsTrain,0.8,0.1,0.1,"randomized");

[adsEnroll,adsLeftover] = splitEachLabel(adsEvaluate,3,"randomized");

adsDET = audioDatastore([adsLeftover.Files;adsDev.Files]);
adsDET.Labels = [adsLeftover.Labels;adsDev.Labels];

```

You can reduce the training and detection error trade-off datasets 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,5);
    adsValidation = splitEachLabel(adsValidation,2);
    adsDET = splitEachLabel(adsDET,5);
end

```

Feature Extraction

Create an `audioFeatureExtractor` object to extract 30 MFCCs from 30 ms Hann windows with a 10 ms hop. The sample rate of the data set is 16 kHz.

```
fs = 16e3;

windowDuration = 0.03 ;
hopDuration = 0.01 ;
windowSamples = round(windowDuration*fs);
hopSamples = round(hopDuration*fs);
overlapSamples = windowSamples - hopSamples;

numCoeffs = 30 ;
afe = audioFeatureExtractor( ...
    SampleRate=fs, ...
    Window=hann(windowSamples,"periodic"), ...
    OverlapLength=overlapSamples, ...
    mfcc=true);
setExtractorParameters(afe,"mfcc",NumCoeffs=numCoeffs)
```

Create a transform datastore that applies preprocessing to the audio and outputs features. The supporting function, `xVectorPreprocess` on page 1-653, performs speech detection, extract features from regions of speech. When the parameter `Segment` is set to `false`, the detect regions of speech are concatenated together.

```
adsTrainTransform = transform(adsTrain,@(x)xVectorPreprocess(x,afe,Segment=false,MinimumDuration:
features = preview(adsTrainTransform)

features = 1x1 cell array
    {30x363 single}
```

In a loop, extract all features from the training set. If you have Parallel Computing Toolbox™, then the computations are spread across multiple workers.

```
numPar = numpartitions(adsTrain);
features = cell(1,numPar);
parfor ii = 1:numPar
    adsPart = partition(adsTrainTransform,numPar,ii);
    N = numel(adsPart.UnderlyingDatastores{1}.Files);
    f = cell(1,N);
    for jj = 1:N
        f{jj} = read(adsPart);
    end
    features{ii} = cat(2,f{:});
end
```

Concatenate the features and then save the global mean and standard deviation in a struct. You will use these factors to normalize features.

```
features = cat(2,features{:});
features = cat(2,features{:});
factors = struct("Mean",mean(features,2),"STD",std(features,0,2));
clear features f
```

Create a new transform datastore for the training set, this time specifying the normalization factors and `Segment` as `true`. Now, features are normalized by the global mean and standard deviation, and

then the file-level mean. The individual speech regions detected are not concatenated. The output is a table with the first variable containing feature matrices and the second variable containing the label.

```
adsTrainTransform = transform(adsTrain,@(x,myInfo)xVectorPreprocess(x,afe,myInfo, ...
    Segment=true,Factors=factors,MinimumDuration=0.5), ...
    IncludeInfo=true);
featuresTable = preview(adsTrainTransform)
```

```
featuresTable=3x2 table
      features      labels
-----
{30x142 single}    1034
{30x64  single}    1034
{30x157 single}    1034
```

Apply the same transformation to the validation, test, enrollment, and DET sets.

```
adsValidationTransform = transform(adsValidation,@(x,myInfo)xVectorPreprocess(x,afe,myInfo, ...
    Segment=true,Factors=factors,MinimumDuration=0.5), ...
    IncludeInfo=true);
adsTestTransform = transform(adsTest,@(x,myInfo)xVectorPreprocess(x,afe,myInfo, ...
    Segment=true,Factors=factors,MinimumDuration=0.5), ...
    IncludeInfo=true);
adsEnrollTransform = transform(adsEnroll,@(x,myInfo)xVectorPreprocess(x,afe,myInfo, ...
    Segment=true,Factors=factors,MinimumDuration=0.5), ...
    IncludeInfo=true);
adsDETRTransform = transform(adsDET,@(x,myInfo)xVectorPreprocess(x,afe,myInfo, ...
    Segment=true,Factors=factors,MinimumDuration=0.5), ...
    IncludeInfo=true);
```

x-vector Feature Extraction Model

In this example, you define the x-vector feature extractor model [1] on page 1-653 as a layer graph and train it using a custom training loop. This paradigm enables you to preprocess the mini-batches and trim the sequences to a consistent length.

The table summarizes the architecture of the network described in [1] on page 1-653 and implemented in this example. T is the total number of frames (feature vectors over time) in an audio signal. N is the number of classes (speakers) in the training set.

Layer	Description	Layer Context	Total Context	Input-by-Output
1	1-d convolutional batch normalization ReLU activation	$[t - 2, t + 2]$	5	$(5 \times numFeatures) - by - numFilters$
2	1-d convolutional batch normalization ReLU activation	$\{t - 2, t, t + 2\}$	9	$(3 \times numFilters) - by - numFilters$
3	1-d convolutional batch normalization ReLU activation	$\{t - 3, t, t + 3\}$	15	$(3 \times numFilters) - by - numFilters$
4	1-d convolutional batch normalization ReLU activation	$\{t\}$	15	$numFilters - by - numFilters$
5	1-d convolutional batch normalization ReLU activation	$\{t\}$	15	$numFilters - by - 1500$
6	statistics pooling	$[0, T)$	T	$(1500 \times T) - by - 3000$
7	fully-connected batch normalization ReLU activation	$\{0\}$	T	$3000 - by - numFilters$
8	fully-connected batch normalization ReLU activation	$\{0\}$	T	$numFilters - by - numFilters$
9	fully-connected softmax	$\{0\}$	T	$numFilters - by - N$

Define the network. You can change the model size by increasing or decreasing the `numFilters` parameter.

```

numFilters = 512;
dropProb = 0.2;
numClasses = numel(unique(adTrain.Labels));
layers = [
    sequenceInputLayer(afe.FeatureVectorLength,MinLength=15,Name="input")

    convolution1dLayer(5,numFilters,DilationFactor=1,Name="conv_1")
    batchNormalizationLayer(Name="BN_1")
    dropoutLayer(dropProb,Name="drop_1")
    reluLayer(Name="act_1")

    convolution1dLayer(3,numFilters,DilationFactor=2,Name="conv_2")
    batchNormalizationLayer(Name="BN_2")
    dropoutLayer(dropProb,Name="drop_2")
    reluLayer(Name="act_2")

    convolution1dLayer(3,numFilters,DilationFactor=3,Name="conv_3")
    batchNormalizationLayer(Name="BN_3")
    dropoutLayer(dropProb,Name="drop_3")
    reluLayer(Name="act_3")

    convolution1dLayer(1,numFilters,DilationFactor=1,Name="conv_4")
    batchNormalizationLayer(Name="BN_4")

```



```

dropoutLayer(dropProb,Name="drop_4")
reluLayer(Name="act_4")

convolution1dLayer(1,1500,DilationFactor=1,Name="conv_5")
batchNormalizationLayer(Name="BN_5")
dropoutLayer(dropProb,Name="drop_5")
reluLayer(Name="act_5")

statisticsPooling1dLayer(Name="statistics_pooling")

fullyConnectedLayer(numFilters,Name="fc_1")
batchNormalizationLayer(Name="BN_6")
dropoutLayer(dropProb,Name="drop_6")
reluLayer(Name="act_6")

fullyConnectedLayer(numFilters,Name="fc_2")
batchNormalizationLayer(Name="BN_7")
dropoutLayer(dropProb,Name="drop_7")
reluLayer(Name="act_7")

fullyConnectedLayer(numClasses,Name="fc_3")
softmaxLayer(Name="softmax")
];

dlnet = dlnetwork(layerGraph(layers));

```

The model requires statistical pooling which is implemented as a custom layer and placed in your current folder when you open this example. Display the contents of the custom layer.

```

type("statisticsPooling1dLayer.m")

classdef statisticsPooling1dLayer < nnet.layer.Layer & nnet.layer.Formattable
    % This class is only for use in this example. It may be changed or
    % removed in a future release.

    methods
        function this = statisticsPooling1dLayer(options)
            arguments
                options.Name = ''
            end
            this.Name = options.Name;
        end


        function X = predict(~, X)
            X = dlarray(stripdims([mean(X,3);std(X,0,3)]),"CB");
        end
        function X = forward(~, X)
            X = X + 0.0001*rand(size(X),"single");
            X = dlarray(stripdims([mean(X,3);std(X,0,3)]),"CB");
        end
    end
end
end

```

Train Model

Use `minibatchqueue` (Deep Learning Toolbox) to create a mini-batch queue for the training data. Set the mini-batch size as appropriate for your hardware.


```


miniBatchSize = 256 ;
mbq = minibatchqueue(adsTrainTransform, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFormat=["CTB", ""], ...
    MiniBatchFcn=@preprocessMiniBatch, ...
    OutputEnvironment="auto");



```

Set the number of training epochs, the initial learn rate, the learn rate drop period, the learn rate drop factor, and the validations per epoch.

```

numEpochs = 5 ;

learnRate = 0.001 ;
gradDecay = 0.5;
sqGradDecay = 0.999;
trailingAvg = [];
trailingAvgSq = [];

LearnRateDropPeriod = 2 ;
LearnRateDropFactor = 0.1 ;

```

To display training progress, initialize the supporting object `progressPlotter`. The supporting object, `progressPlotter`, is placed in your current folder when you open this example.

Run the training loop.

```

classes = unique(adsTrain.Labels);
pp = progressPlotter(string(classes));

iteration = 0;
for epoch = 1:numEpochs

    % Shuffle mini-batch queue
    shuffle(mbq)

    while hasdata(mbq)

        % Update iteration counter
        iteration = iteration + 1;

        % Get mini-batch from mini-batch queue
        [dlX,Y] = next(mbq);

        % Evaluate the model gradients, state, and loss using dlfeval and the modelGradients function
        [gradients,dlnet.State,loss,predictions] = dlfeval(@modelGradients,dlnet,dlX,Y);

        % Update the network parameters using the Adam optimizer
        [dlnet,trailingAvg,trailingAvgSq] = adamupdate(dlnet,gradients, ...
            trailingAvg,trailingAvgSq,iteration,learnRate,gradDecay,sqGradDecay,eps("single"));

        % Update the training progress plot
        updateTrainingProgress(pp,Epoch=epoch,Iteration=iteration,LearnRate=learnRate,Predictions=predictions);

    end
end

```

```

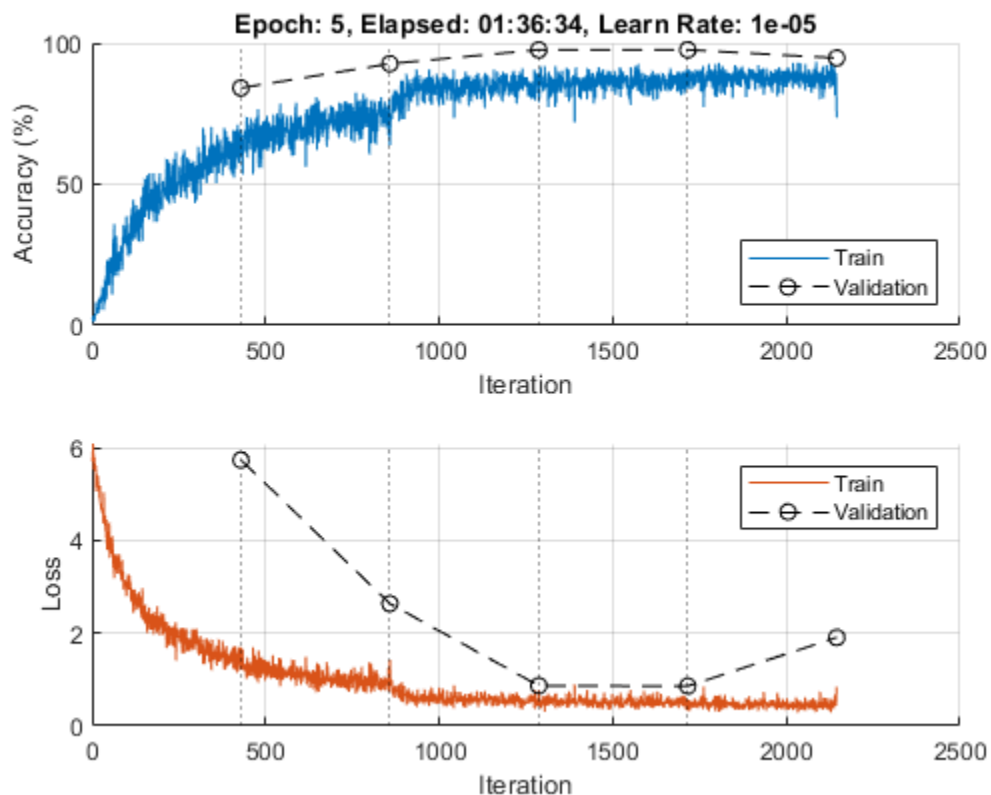
% Pass validation data through model
[predictionValidation,labelsValidation] = predictBatch(dlNet,adsValidationTransform);
predictionValidation = onehotdecode(predictionValidation,string(classes),1);

% Update the training progress plot with validation results
updateValidation(pp,Iteration=iteration,Predictions=predictionValidation,Targets=labelsValida

% Update learn rate
if rem(epoch,LearnRateDropPeriod)==0
    learnRate = learnRate*LearnRateDropFactor;
end

end

```



Evaluate the TDNN speaker recognition accuracy using the held-out test set. The supporting function, `predictBatch`, parallelizes the prediction computation if you have Parallel Computing Toolbox™. Decode the predictions and then compute the prediction accuracy.

```

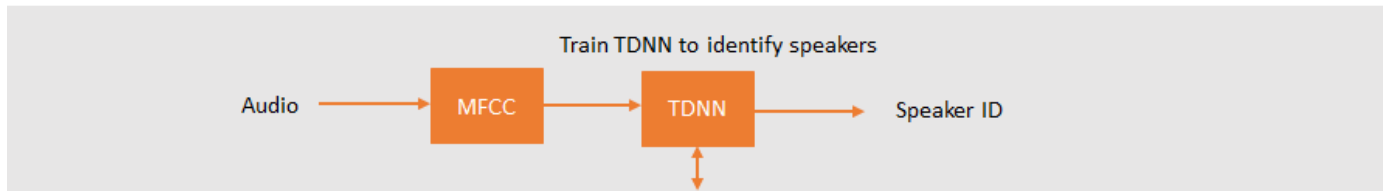
[predictionTest,targetTest] = predictBatch(dlNet,adsTestTransform);
predictionTest = onehotdecode(predictionTest,string(classes),1);
accuracy = mean(targetTest(:)==predictionTest(:))
accuracy = 0.9460

```

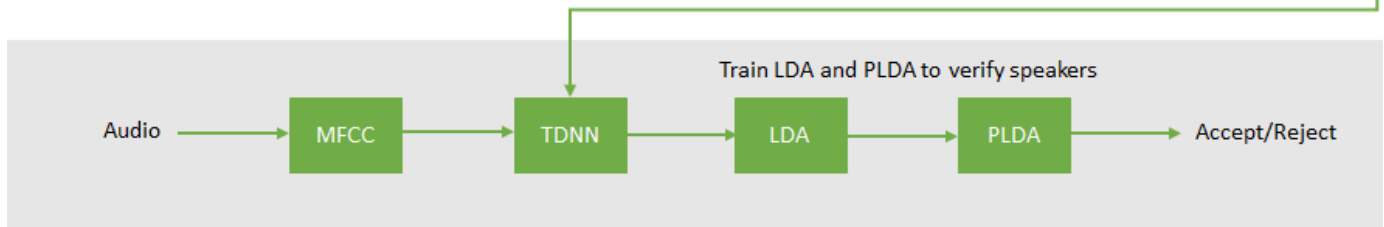
Train x-vector System Backend

In the x-vector system for speaker verification, the TDNN you just trained is used to output an embedding layer. The output from the embedding layer ("fc_1" in this example) are the "x-vectors" in an x-vector system.

Speaker Identification



Layer	Description	Layer Context	Total Context	Input-by-Output
1	1-d convolutional batch normalization ReLU activation	$[t - 2, t + 2]$	5	$(5 \times numFeatures) - by - numFilters$
2	1-d convolutional batch normalization ReLU activation	$\{t - 2, t, t + 2\}$	9	$(3 \times numFilters) - by - numFilters$
3	1-d convolutional batch normalization ReLU activation	$\{t - 3, t, t + 3\}$	15	$(3 \times numFilters) - by - numFilters$
4	1-d convolutional batch normalization ReLU activation	$\{t\}$	15	$numFilters - by - numFilters$
5	1-d convolutional batch normalization ReLU activation	$\{t\}$	15	$numFilters - by - 1500$
6	statistics pooling	$[0, T)$	T	$(1500 \times T) - by - 3000$
7	fully-connected batch normalization ReLU activation	$\{0\}$	T	$3000 - by - numFilters$
8	fully-connected batch normalization ReLU activation	$\{0\}$	T	$numFilters - by - numFilters$
9	fully-connected softmax	$\{0\}$	T	$numFilters - by - N$



Speaker Verification

The backend (or classifier) of an x-vector system is the same as the backend of an i-vector system. For details on the algorithms, see `ivectorSystem` and "Speaker Verification Using i-Vectors" on page 1-582.

Extract x-vectors from the train set.

```
[xvecsTrain, labelsTrain] = predictBatch(dlNet, adsTrainTransform, Outputs="fc_1");
```

Create a linear discriminant analysis (LDA) projection matrix to reduce the dimensionality of the x-vectors. LDA attempts to minimize the intra-class variance and maximize the variance between speakers.

```
numEigenvectors = 150 _____ ▾ ;
```

```
projMat = helperTrainProjectionMatrix(xvecsTrain, labelsTrain, numEigenvectors);
```

Apply the LDA projection matrix to the x-vectors.

```
xvecsTrainP = projMat*xvecsTrain;
```

Train a G-PLDA model to perform scoring.

```
numIterations = 3 _____ ▾ ;
```

```
numDimensions = 150 _____ ▾ ;
```

```
plda = helperTrainPLDA(xvecsTrainP, labelsTrain, numIterations, numDimensions);
```

Evaluate x-vector System

Speaker verification systems verify that a speaker is who they purport to be. Before a speaker can be verified, they must be enrolled in the system. Enrollment in the system means that the system has a template x-vector representation of the speaker.

Enroll Speakers

Extract x-vectors from the held-out data set, `adsEnroll`.

```
[xvecsEnroll, labelsEnroll] = predictBatch(dlNet, adsEnrollTransform, Outputs="fc_1");
```

Apply the LDA projection matrix to the x-vectors.

```
xvecsEnrollP = projMat*xvecsEnroll;
```

Create template x-vectors for each speaker by averaging the x-vectors of individual speakers across enrollment files.

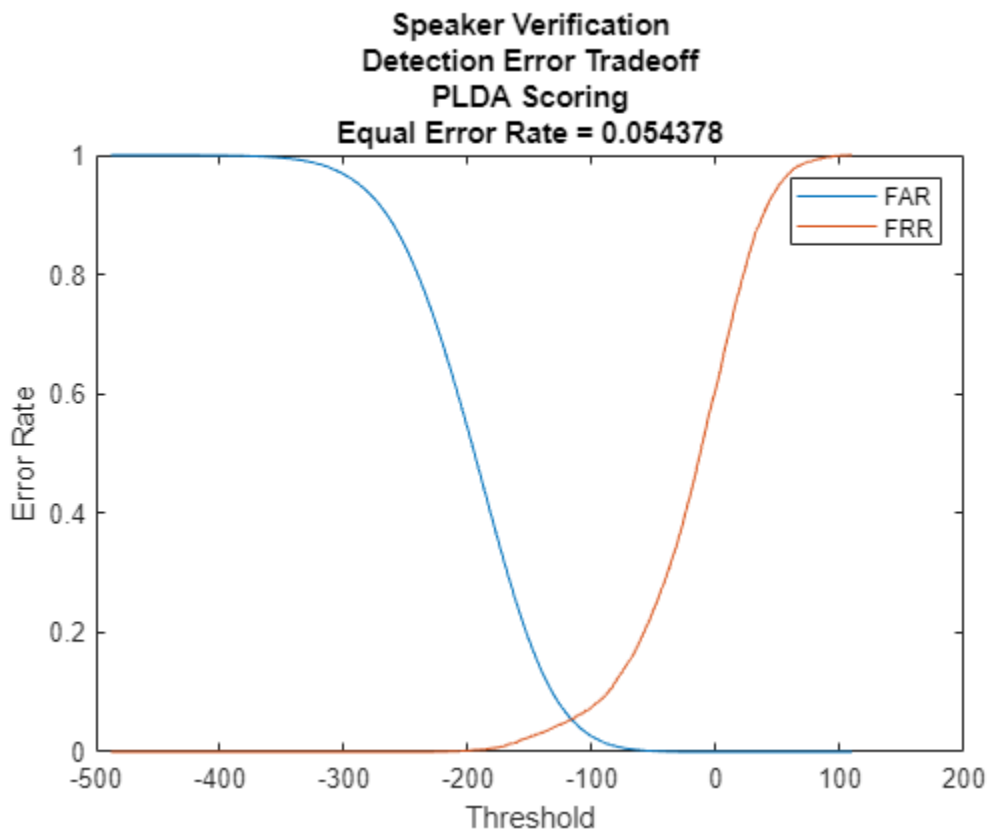
```
uniqueLabels = unique(labelsEnroll);
enrollmentTable = cell2table(cell(0,2), VariableNames=["xvector", "NumSamples"]);
for ii = 1:numel(uniqueLabels)
    idx = uniqueLabels(ii)==labelsEnroll;
    wLocalMean = mean(xvecsEnrollP(:,idx),2);
    localTable = table({wLocalMean},(sum(idx)), ...
        VariableNames=["xvector", "NumSamples"], ...
        RowNames=string(uniqueLabels(ii)));
    enrollmentTable = [enrollmentTable; localTable]; %#ok<AGROW>
end
```

Speaker verification systems require you to set a threshold that balances the probability of a false acceptance (FA) and the probability of a false rejection (FR), according to the requirements of your application. To determine the threshold that meets your FA/FR requirements, evaluate the detection error tradeoff of the system.

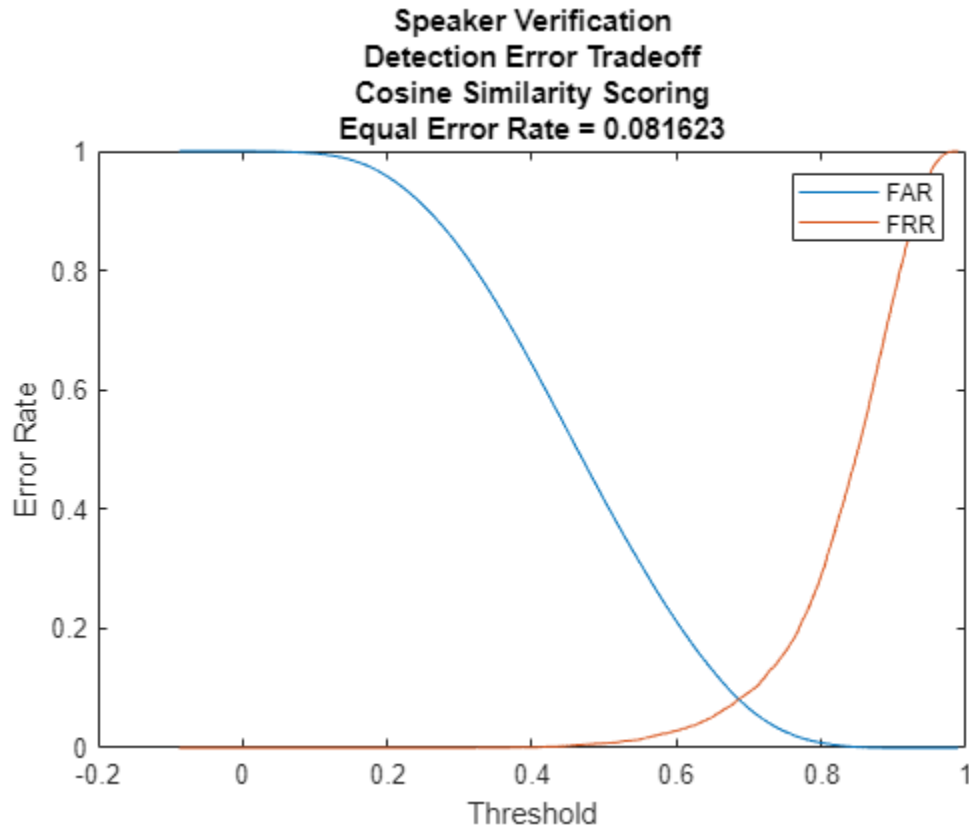
```
[xvecsDET,labelsDET] = predictBatch(dlnet,adsDETTransform,Outputs="fc_1");
xvecsDETP = projMat*xvecsDET;
detTable = helperDetectionErrorTradeoff(xvecsDETP,labelsDET,enrollmentTable,plda);
```

Plot the results of the detection error tradeoff evaluation for both PLDA scoring and cosine similarity scoring (CSS).

```
figure
plot(detTable.PLDA.Threshold,detTable.PLDA.FAR, ...
     detTable.PLDA.Threshold,detTable.PLDA.FRR)
eer = helperEqualErrorRate(detTable.PLDA);
title(["Speaker Verification","Detection Error Tradeoff","PLDA Scoring","Equal Error Rate = " +
xlablel("Threshold")
ylabel("Error Rate")
legend(["FAR","FRR"])
```



```
figure
plot(detTable.CSS.Threshold,detTable.CSS.FAR, ...
     detTable.CSS.Threshold,detTable.CSS.FRR)
eer = helperEqualErrorRate(detTable.CSS);
title(["Speaker Verification","Detection Error Tradeoff","Cosine Similarity Scoring","Equal Error
xlablel("Threshold")
ylabel("Error Rate")
legend(["FAR","FRR"])
```



References

[1] 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.

[2] V. Panayotov, G. Chen, D. Povey and S. Khudanpur, "Librispeech: An ASR corpus based on public domain audio books," *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Brisbane, QLD, 2015, pp. 5206-5210, doi: 10.1109/ICASSP.2015.7178964

Supporting Functions

Feature Extraction and Normalization

```
function [output,myInfo] = xVectorPreprocess(audioData,afe,myInfo,nvars)
% This function is only for use in this example. It may be changed or
% removed in a future release.
arguments
    audioData
    afe
    myInfo = []
    nvars.Factors = []
```

```
        nvars.Segment = true;
        nvars.MinimumDuration = 1;
        nvars.UseGPU = false;
    end

    % Place on GPU if requested
    if nvars.UseGPU
        audioData = gpuArray(audioData);
    end

    % Scale
    audioData = audioData/max(abs(audioData(:)));

    % Protect against NaNs
    audioData(isnan(audioData)) = 0;

    % Determine regions of speech
    mergeDur = 0.2; % seconds
    idx = detectSpeech(audioData,afe.SampleRate,MergeDistance=afe.SampleRate*mergeDur);

    % If a region is less than MinimumDuration seconds, drop it.
    if nvars.Segment
        idxToRemove = (idx(:,2)-idx(:,1))<afe.SampleRate*nvars.MinimumDuration;
        idx(idxToRemove,:) = [];
    end

    % Extract features
    numSegments = size(idx,1);
    features = cell(numSegments,1);
    for ii = 1:numSegments
        temp = (single(extract(afe,audioData(idx(ii,1):idx(ii,2)))));
        if isempty(temp)
            temp = zeros(30,15,"single");
        end
        features{ii} = temp;
    end

    % Standardize features
    if ~isempty(nvars.Factors)
        features = cellfun(@(x)(x-nvars.Factors.Mean)./nvars.Factors.STD,features,UniformOutput=false);
    end

    % Cepstral mean subtraction (for channel noise)
    if ~isempty(nvars.Factors)
        fileMean = mean(cat(2,features{:}),"all");
        features = cellfun(@(x)x - fileMean,features,UniformOutput=false);
    end

    if ~nvars.Segment
        features = {cat(2,features{:})};
    end
    if isempty(myInfo)
        output = features;
    else
        labels = repelem(myInfo.Label,numel(features),1);
        output = table(features,labels);
    end
```



```
end
end
```

Calculate Model Gradients and Updated State

```
function [gradients,state,loss,YPred] = modelGradients(dlnet,X,Y)
% This function is only for use in this example. It may be changed or
% removed in a future release.
```

```
[YPred,state] = forward(dlnet,X);

loss = crossentropy(YPred,Y);
gradients = dlgradient(loss,dlnet.Learnables);

loss = double(gather(extractdata(loss)));
```

```
end
```

Preprocess Mini-Batch

```
function [sequences,labels] = preprocessMiniBatch(sequences,labels)
% This function is only for use in this example. It may be changed or
% removed in a future release.
```

```
trimDimension = 2;
lengths = cellfun(@(x)size(x,trimDimension),sequences);
minLength = min(lengths);
sequences = cellfun(@(x)randomTruncate(x,trimDimension,minLength),sequences,UniformOutput=false);
sequences = cat(3,sequences{:});
```

```
labels = cat(2,labels{:});
labels = onehotencode(labels,1);
labels(isnan(labels)) = 0;
```

```
end
```

Randomly Truncate Audio Signals to Specified Length

```
function y = randomTruncate(x,dim,minLength)
% This function is only for use in this example. It may be changed or
% removed in a future release.
```

```
N = size(x,dim);
if N > minLength
    start = randperm(N-minLength,1);
    if dim==1
        y = x(start:start+minLength-1,:);
    elseif dim ==2
        y = x(:,start:start+minLength-1);
    end
end
```

```
else
    y = x;
end
```

```
end
end
```

Predict Batch

```
function [xvecs,labels] = predictBatch(dlnet,ds,nvars)
```

```
arguments
    dlnet
    ds
```

```
        nvars.Outputs = [];  
    end  
    if ~isempty(ver("parallel"))  
        pool = gcp;  
        numPartition = numpartitions(ds,pool);  
    else  
        numPartition = 1;  
    end  
    xvecs = [];  
    labels = [];  
    outputs = nvars.Outputs;  
    parfor partitionIndex = 1:numPartition  
        dsPart = partition(ds,numPartition,partitionIndex);  
        partitionFeatures = [];  
        partitionLabels = [];  
        while hasdata(dsPart)  
            atable = read(dsPart);  
            F = atable.features;  
            L = atable.labels;  
            for kk = 1:numel(L)  
                if isempty(outputs)  
                    f = gather(extractdata(predict(dlnet,(dlarray(F{kk},"CTB")))));  
                else  
                    f = gather(extractdata(predict(dlnet,(dlarray(F{kk},"CTB")),Outputs=outputs)));  
                end  
                l = L(kk);  
                partitionFeatures = [partitionFeatures,f];  
                partitionLabels = [partitionLabels,l];  
            end  
        end  
        xvecs = [xvecs,partitionFeatures];  
        labels = [labels,partitionLabels];  
    end  
end
```

Speaker Diarization Using x-vectors

Speaker diarization is the process of partitioning an audio signal into segments according to speaker identity. It answers the question "who spoke when" without prior knowledge of the speakers and, depending on the application, without prior knowledge of the number of speakers.

Speaker diarization has many applications, including: enhancing speech transcription by structuring text according to active speaker, video captioning, content retrieval (*what did Jane say?*) and speaker counting (*how many speakers were present in the meeting?*).

In this example, you perform speaker diarization using a pretrained x-vector system [1] on page 1-670 to characterize regions of audio and agglomerative hierarchical clustering (AHC) to group similar regions of audio [2] on page 1-670. To see how the x-vector system was defined and trained, see "Speaker Recognition Using x-vectors" on page 1-642.

Download Pretrained Speaker Diarization System

Download the pretrained speaker diarization system and supporting files. The total size is approximately 22 MB.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "SpeakerDiarization.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
netFolder = fullfile(dataFolder, "SpeakerDiarization");
```

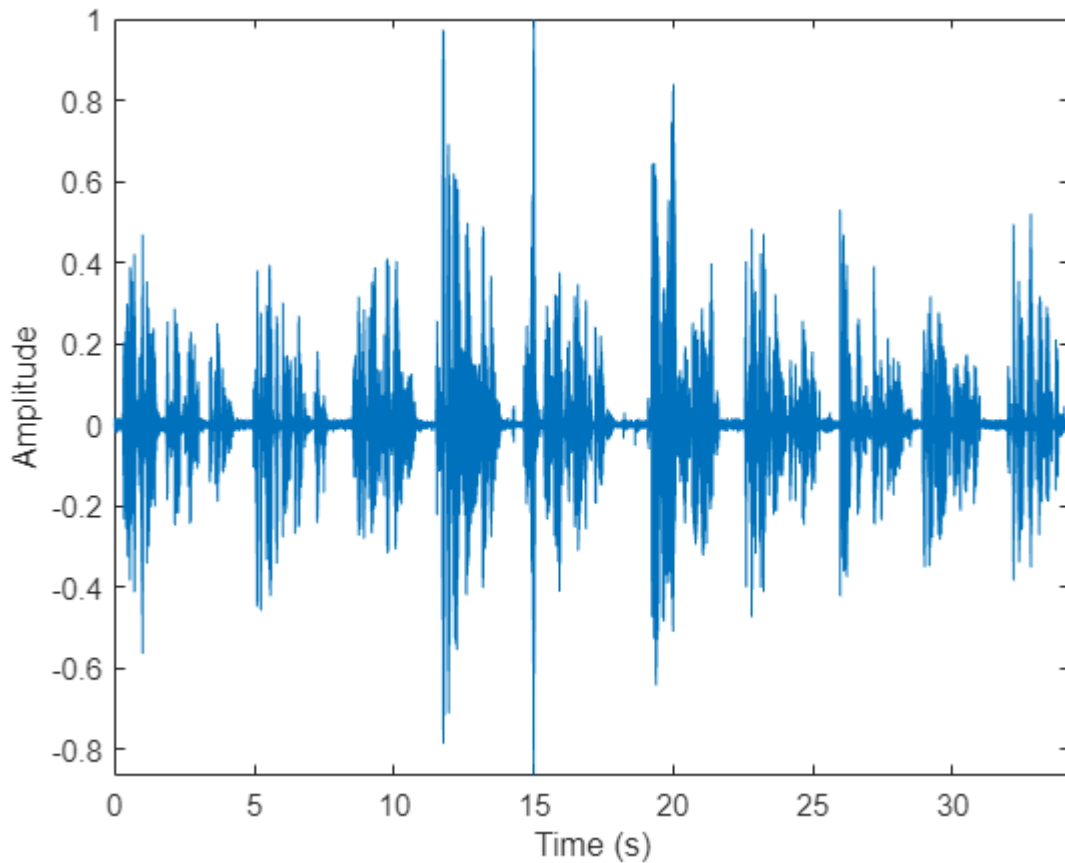
```
addpath(netFolder)
```

Load an audio signal and a table containing ground truth annotations. The signal consists of five speakers. Listen to the audio signal and plot its time-domain waveform.

```
[audioIn, fs] = audioread("exampleconversation.flac");
load("exampleconversationlabels.mat")
audioIn = audioIn./max(abs(audioIn));
sound(audioIn, fs)
```

```
t = (0:size(audioIn,1)-1)/fs;
```

```
figure(1)
plot(t, audioIn)
xlabel("Time (s)")
ylabel("Amplitude")
axis tight
```



Extract x-vectors

In this example, you used a pretrained x-vector system based on [1] on page 1-670. To see how the x-vector system was defined and trained, see “Speaker Recognition Using x-vectors” on page 1-642.

Load Pretrained x-Vector System

Load the lightweight pretrained x-vector system. The x-vector system consists of:

- `afe` - an `audioFeatureExtractor` object to extract mel frequency cepstral coefficients (MFCCs).
- `factors` - a struct containing the mean and standard deviation of MFCCs determined from a representative data set. These factors are used to standardize the MFCCs.
- `dlnet` - a trained `dlnetwork`. The network is used to extract x-vectors from the MFCCs.
- `projMat` - a trained projection matrix to reduce the dimensionality of x-vectors.
- `plda` - a trained PLDA model for scoring x-vectors.

```
xvecs = load("xvectorSystem.mat");
```

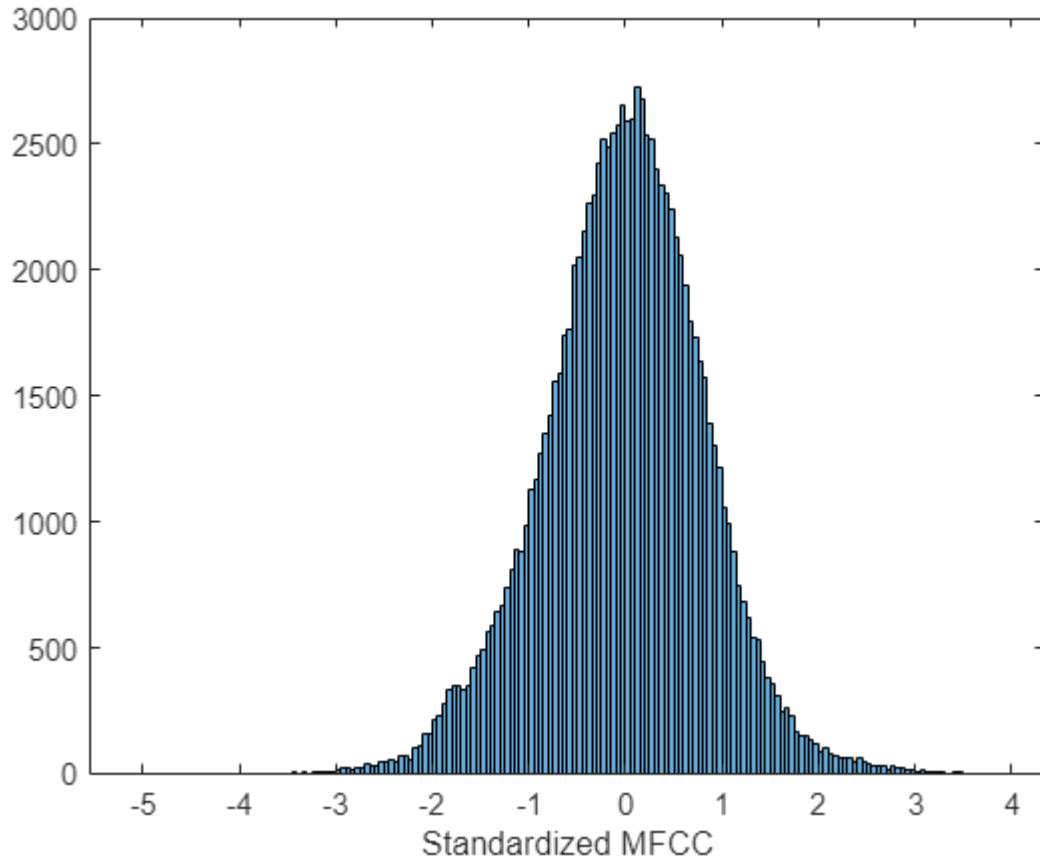
Extract Standardized Acoustic Features

Extract standardized MFCC features from the audio data. View the feature distributions to confirm that the standardization factors learned from a separate data set approximately standardize the

features derived in this example. A standard distribution has a mean of zero and a standard deviation of 1.

```
features = single((extract(xvecsyst.afe, audioIn) - xvecsyst.factors.Mean') ./ xvecsyst.factors.STD');
```

```
figure(2)
histogram(features)
xlabel("Standardized MFCC")
```



Extract x-Vectors

Each acoustic feature vector represents approximately 0.01 seconds of audio data. Group the features into approximately 2 second segments with 0.1 second hops between segments.

```
featureVectorHopDur = (numel(xvecsyst.afe.Window) - xvecsyst.afe.OverlapLength) / xvecsyst.afe.SampleRate;
```

```
segmentDur = 2 ;
```

```
segmentHopDur = 0.1 ;
```

```
segmentLength = round(segmentDur / featureVectorHopDur);
```

```
segmentHop = round(segmentHopDur / featureVectorHopDur);
```

```
idx = 1:segmentLength;
```

```

featuresSegmented = [];
while idx(end) < size(features,1)
    featuresSegmented = cat(3,featuresSegmented,features(idx,:));
    idx = idx + segmentHop;
end

```

Extract x-vectors from each segment. x-vectors correspond to the output from the first fully-connected layer in the x-vector model trained in “Speaker Recognition Using x-vectors” on page 1-642. The first fully-connected layer is the first segment-level layer after statistics are calculated for the time-dilated frame-level layers. Visualize the x-vectors over time.

```

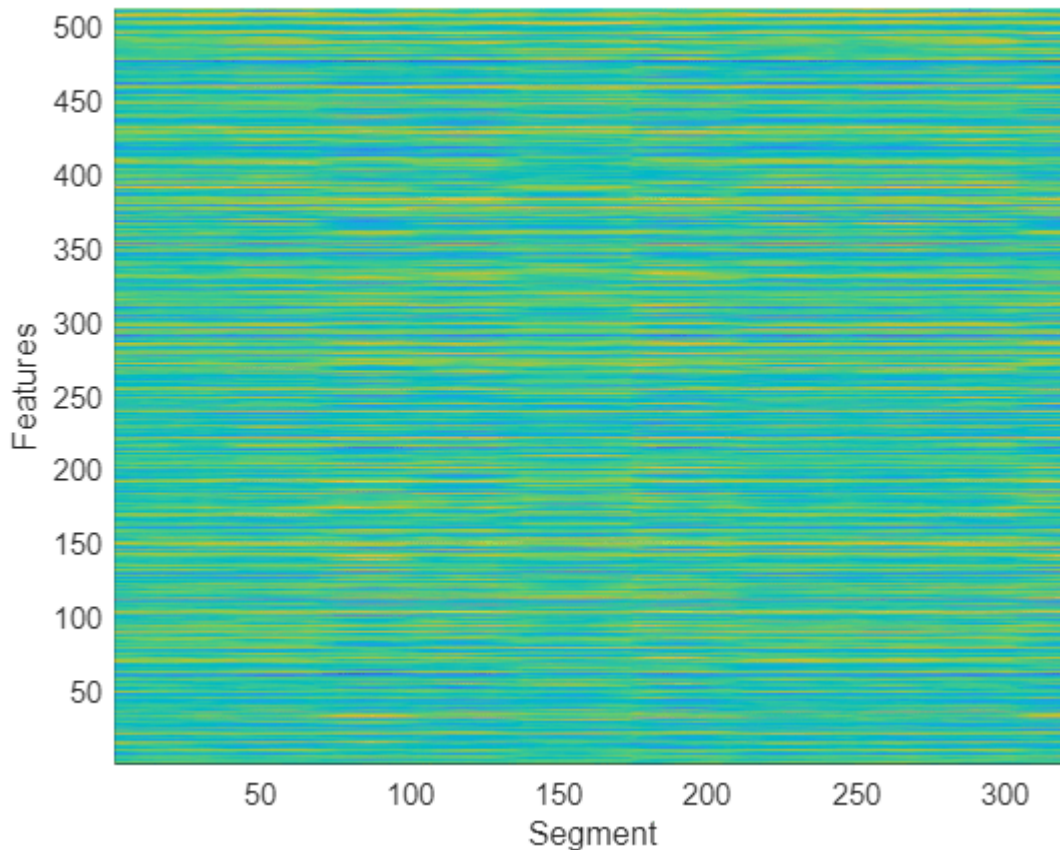
xvecs = zeros(512,size(featuresSegmented,3));
for sample = 1:size(featuresSegmented,3)
    dlX = dlarray(featuresSegmented(:,:,sample),"TCB");
    xvecs(:,sample) = predict(xvecsys.dlnet,dlX,Outputs="fc_1");
end

```

```

figure(3)
surf(xvecs',EdgeColor="none")
view([90,-90])
axis([1 size(xvecs,1) 1 size(xvecs,2)])
xlabel("Features")
ylabel("Segment")

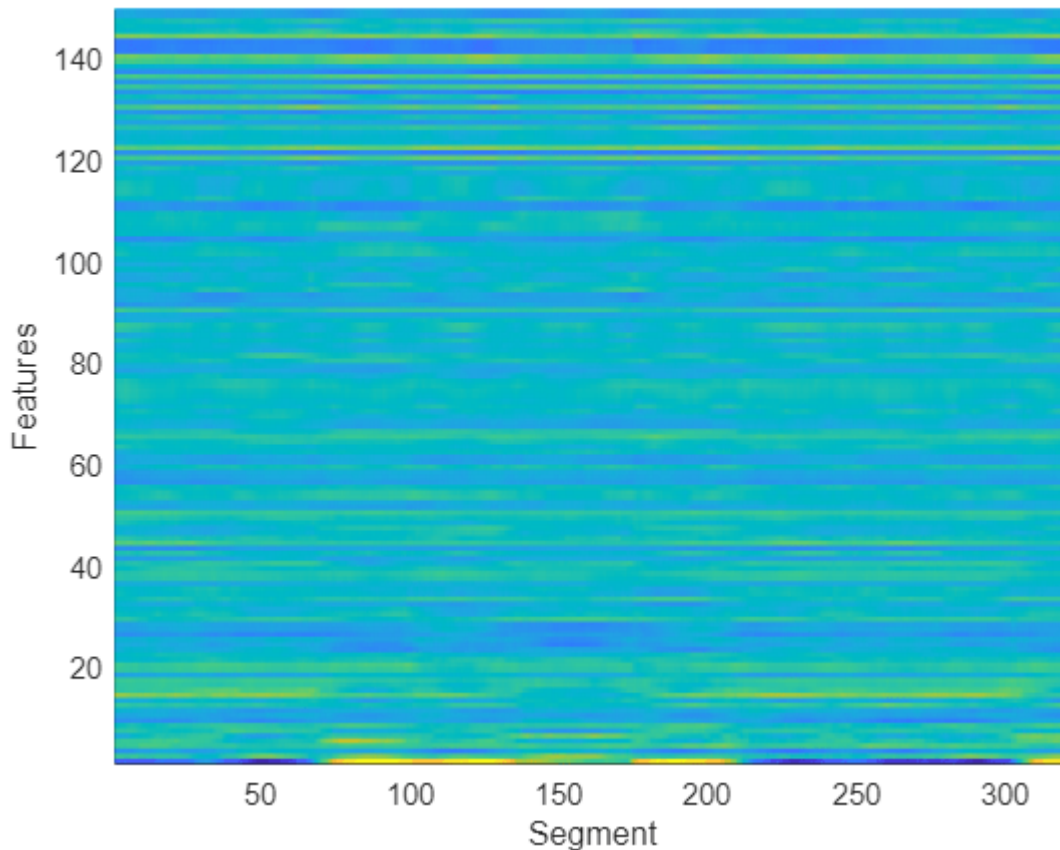
```



Apply the pretrained linear discriminant analysis (LDA) projection matrix to reduce the dimensionality of the x-vectors and then visualize the x-vectors over time.

```
x = xvecsyst.projMat*xvecs;

figure(4)
surf(x',EdgeColor="none")
view([90,-90])
axis([1 size(x,1) 1 size(x,2)])
xlabel("Features")
ylabel("Segment")
```



Cluster x-vectors

An x-vector system learns to extract compact representations (x-vectors) of speakers. Cluster the x-vectors to group similar regions of audio using either agglomerative hierarchical clustering (`clusterdata` (Statistics and Machine Learning Toolbox)) or k-means clustering (`kmeans` (Statistics and Machine Learning Toolbox)). [2] on page 1-670 suggests using agglomerative hierarchical clustering with PLDA scoring as the distance measurement. K-means clustering using a cosine similarity score is also commonly used. Assume prior knowledge of the number of speakers in the audio. Set the maximum clusters to the number of known speakers + 1 so that the background is clustered independently.

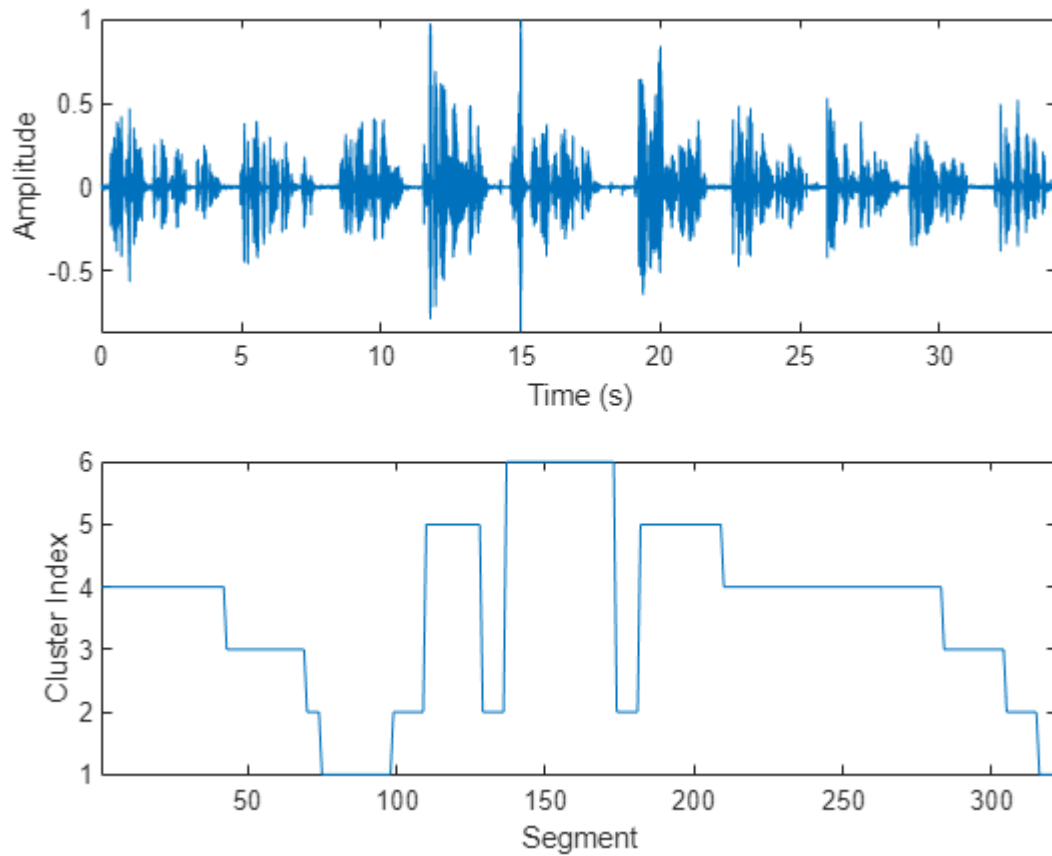
```
knownNumberOfSpeakers = numel(unique(groundTruth.Label));
maxclusters = knownNumberOfSpeakers + 1;
clusterMethod = 'agglomerative - PL...';
switch clusterMethod
    case "agglomerative - PLDA scoring"
        T = clusterdata(x',Criterion="distance",distance=@(a,b)helperPLDAScorer(a,b,xvecsys.plda
    case "agglomerative - CSS scoring"
        T = clusterdata(x',Criterion="distance",distance="cosine",linkage="average",maxclust=max
    case "kmeans - CSS scoring"
        T = kmeans(x',maxclusters,Distance="cosine");
end
```

Plot the cluster decisions over time.

```
figure(5)
tiledlayout(2,1)

nexttile
plot(t, audioIn)
axis tight
ylabel("Amplitude")
xlabel("Time (s)")

nexttile
plot(T)
axis tight
ylabel("Cluster Index")
xlabel("Segment")
```

To isolate segments of speech corresponding to clusters, map the segments back to audio samples. Plot the results.

```

mask = zeros(size(audioIn,1),1);
start = round((segmentDur/2)*fs);

segmentHopSamples = round(segmentHopDur*fs);

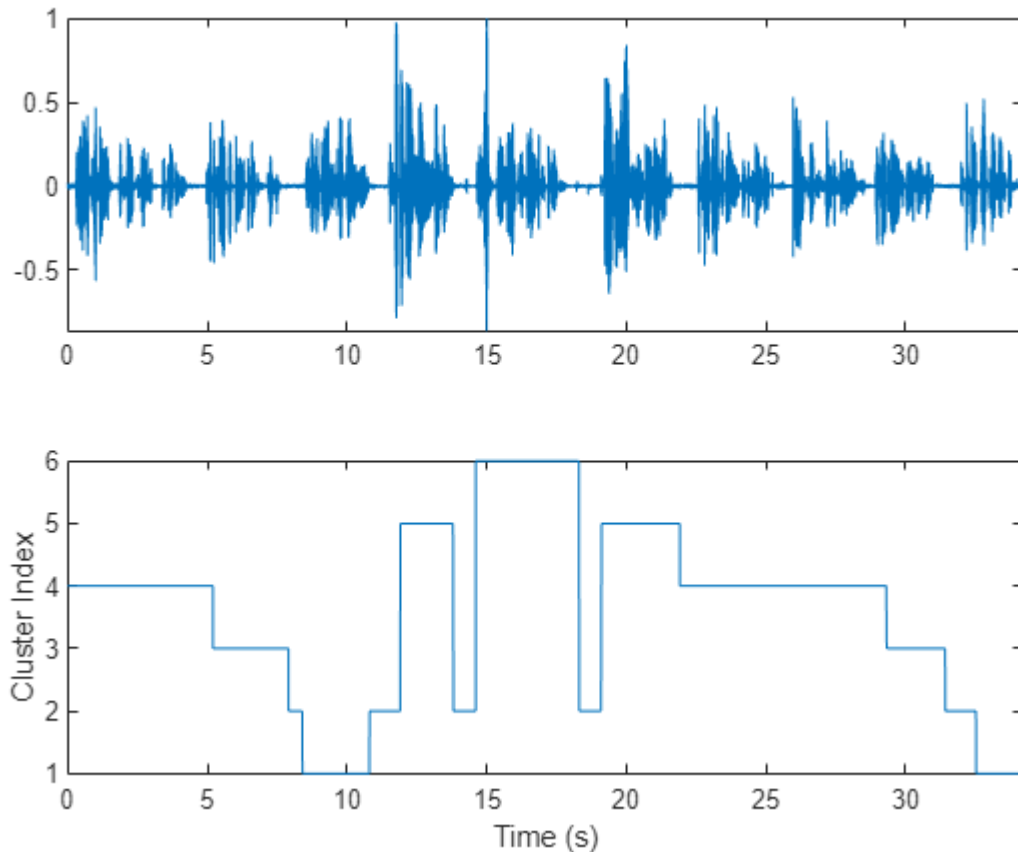
mask(1:start) = T(1);
start = start + 1;
for ii = 1:numel(T)
    finish = start + segmentHopSamples;
    mask(start:start + segmentHopSamples) = T(ii);
    start = finish + 1;
end
mask(finish:end) = T(end);

figure(6)
tiledlayout(2,1)


nexttile
plot(t,audioIn)
axis tight

```

```
nexttile
plot(t,mask)
ylabel("Cluster Index")
axis tight
xlabel("Time (s)")
```

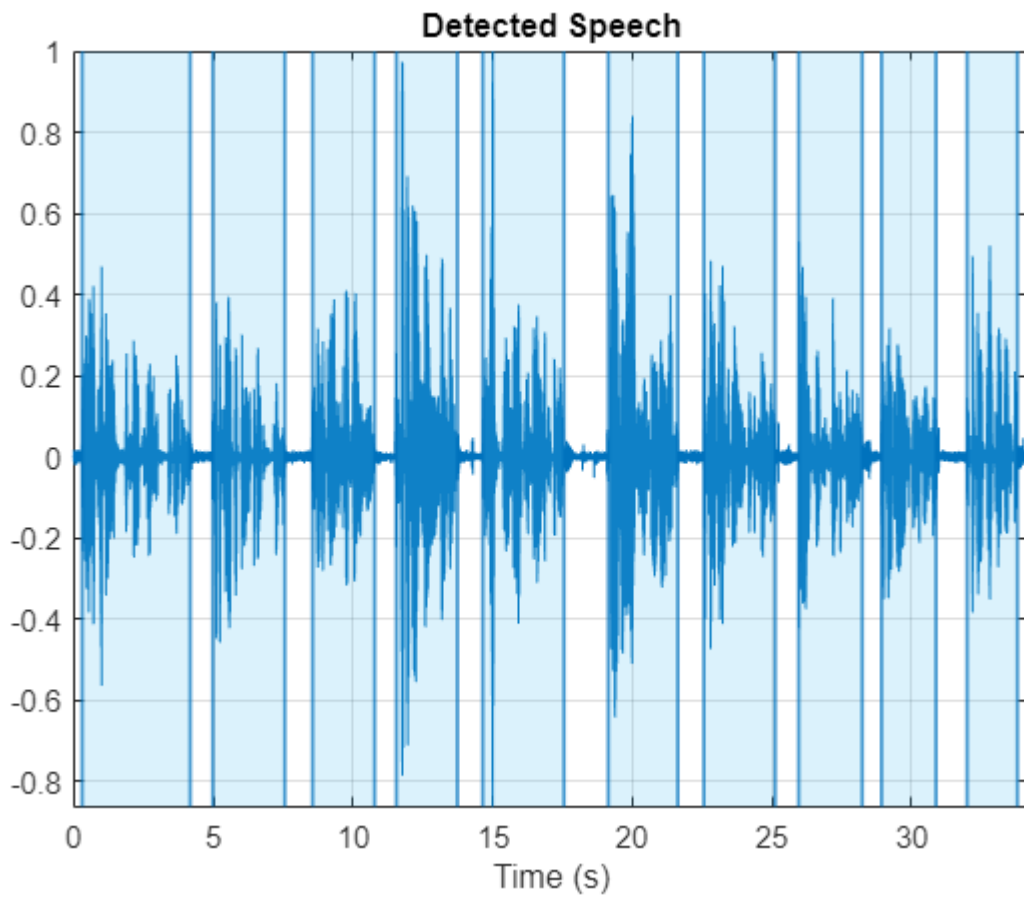


Use `detectSpeech` to determine speech regions. Use `sigroi2binmask` to convert speech regions to a binary voice activity detection (VAD) mask. Call `detectSpeech` a second time without any arguments to plot the detected speech regions.

```
mergeDuration = 0.5  ;
VADidx = detectSpeech(audioIn,fs,MergeDistance=fs*mergeDuration);

VADmask = sigroi2binmask(VADidx,numel(audioIn));

figure(7)
detectSpeech(audioIn,fs,MergeDistance=fs*mergeDuration)
```



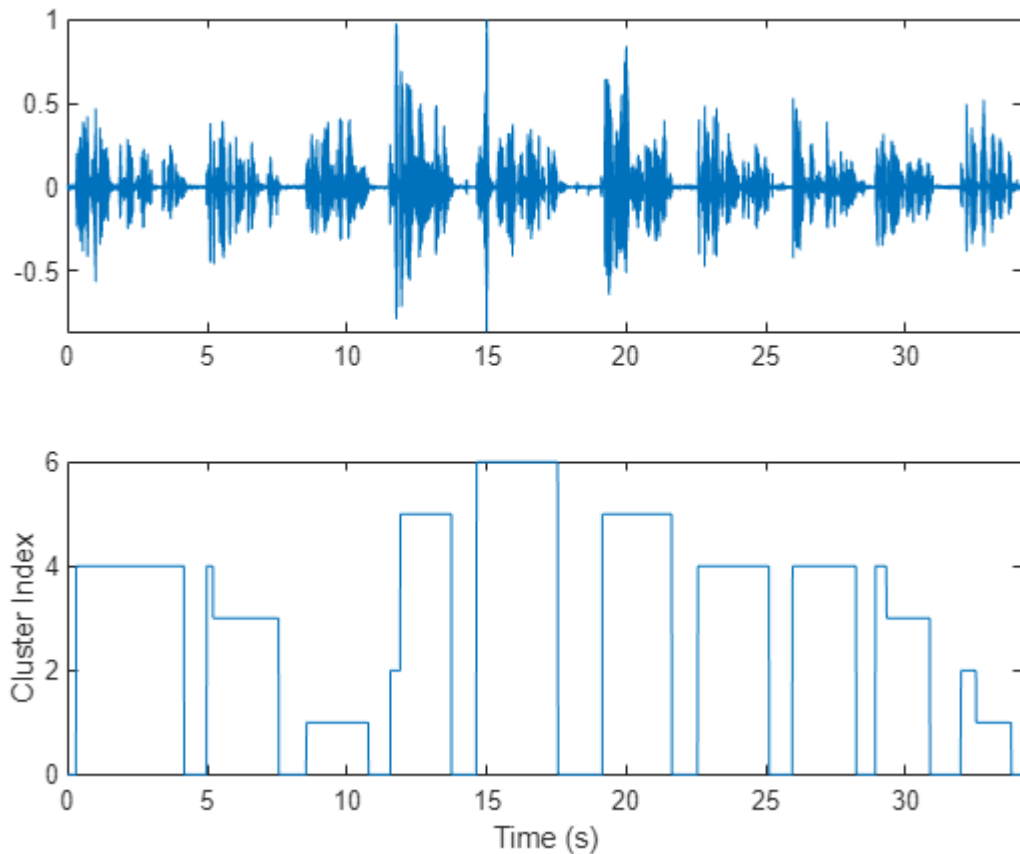
Apply the VAD mask to the speaker mask and plot the results. A cluster index of 0 indicates a region of no speech.

```
mask = mask.*VADmask;
```

```
figure(8)
tiledlayout(2,1)
```

```
nexttile
plot(t, audioIn)
axis tight
```

```
nexttile
plot(t, mask)
ylabel("Cluster Index")
axis tight
xlabel("Time (s)")
```



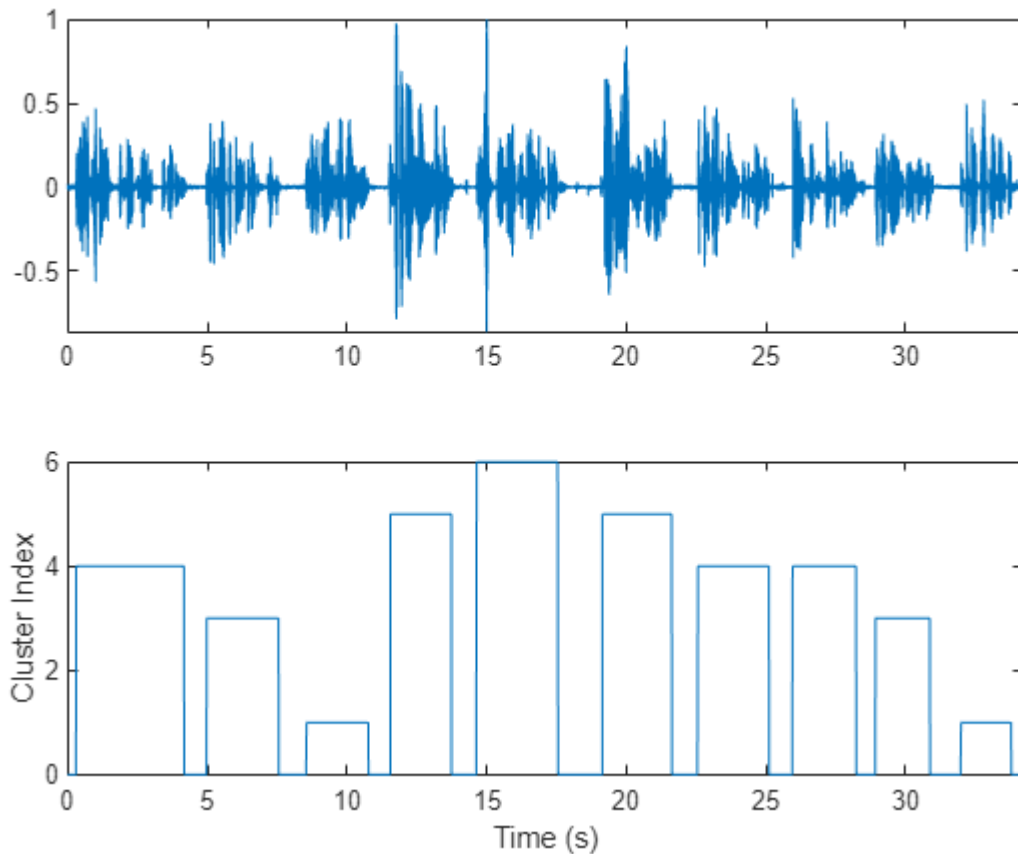
In this example, you assume each detected speech region belongs to a single speaker. If more than two labels are present in a speech region, merge them to the most frequently occurring label.

```
maskLabels = zeros(size(VADidx,1),1);
for ii = 1:size(VADidx,1)
    maskLabels(ii) = mode(mask(VADidx(ii,1):VADidx(ii,2)),"all");
    mask(VADidx(ii,1):VADidx(ii,2)) = maskLabels(ii);
end
```

```
figure(9)
tiledlayout(2,1)
```

```
nexttile
plot(t, audioIn)
axis tight
```

```
nexttile
plot(t, mask)
ylabel("Cluster Index")
axis tight
xlabel("Time (s)")
```



Count the number of remaining speaker clusters.

```
uniqueSpeakerClusters = unique(maskLabels);
numSpeakers = numel(uniqueSpeakerClusters)

numSpeakers = 5
```

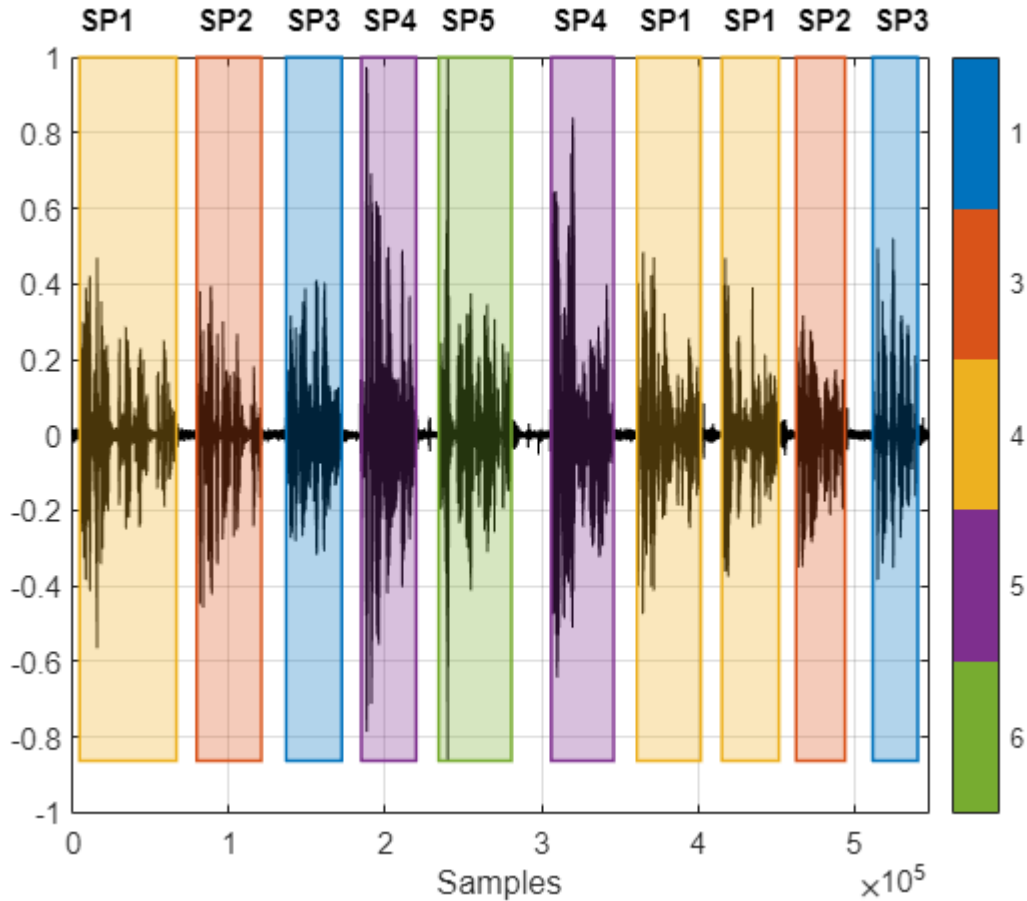
Visualize Diarization Results

Create a `signalMask` object and then plot the speaker clusters. Label the plot with the ground truth labels. The cluster labels are color coded with a key on the right of the plot. The true labels are printed above the plot.

```
msk = signalMask(table(VADidx,categorical(maskLabels)));

figure(10)
plotsigroi(msk, audioIn, true)
axis([0 numel(audioIn) -1 1])

trueLabel = groundTruth.Label;
for ii = 1:numel(trueLabel)
    text(VADidx(ii,1), 1.1, trueLabel(ii), FontWeight="bold")
end
```



Choose a cluster to inspect and then use `binmask` to isolate the speaker. Plot the isolated speech signal and listen to the speaker cluster.

```

speakerToInspect = 2;
cutOutSilenceFromAudio = true;

bmsk = binmask(msk,numel(audioIn));

audioToPlay = audioIn;
if cutOutSilenceFromAudio
    audioToPlay(~bmsk(:,speakerToInspect)) = [];
end
sound(audioToPlay, fs)

figure(11)
tiledlayout(2,1)

nexttile
plot(t,audioIn)
axis tight
ylabel("Amplitude")

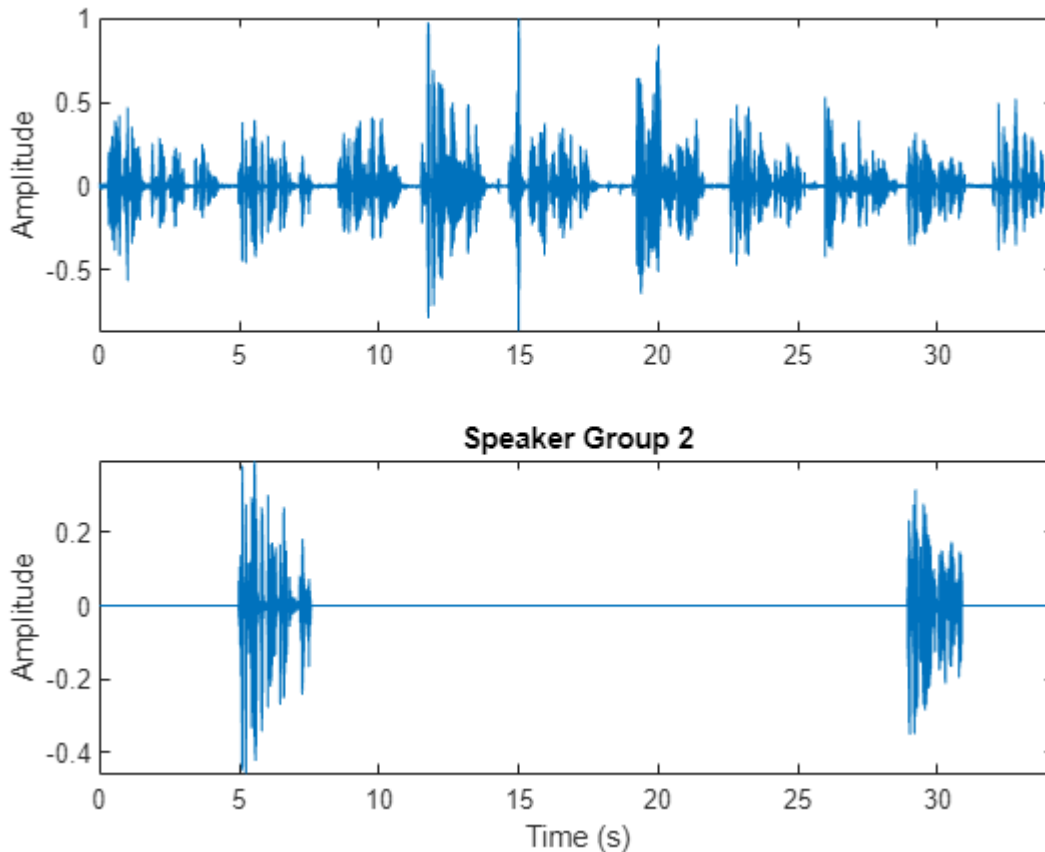
nexttile

```

```

plot(t, audioIn.*bmsk(:, speakerToInspect))
axis tight
xlabel("Time (s)")
ylabel("Amplitude")
title("Speaker Group "+speakerToInspect)

```



Diarization System Evaluation

The common metric for speaker diarization systems is the diarization error rate (DER). The DER is the sum of the miss rate (classifying speech as non-speech), the false alarm rate (classifying non-speech as speech) and the speaker error rate (confusing one speaker's speech for another).

In this simple example, the miss rate and false alarm rate are trivial problems. You evaluate the speaker error rate only.

Map each true speaker to the corresponding best-fitting speaker cluster. To determine the speaker error rate, count the number of mismatches between the true speakers and the best-fitting speaker clusters, and then divide by the number of true speaker regions.

```

uniqueLabels = unique(trueLabel);
guessLabels = maskLabels;
uniqueGuessLabels = unique(guessLabels);

totalNumErrors = 0;

```

```
for ii = 1:numel(uniqueLabels)
    isSpeaker = uniqueLabels(ii)==trueLabel;
    minNumErrors = inf;

    for jj = 1:numel(uniqueGuessLabels)
        groupCandidate = uniqueGuessLabels(jj) == guessLabels;
        numErrors = nnz(isSpeaker - groupCandidate);
        if numErrors < minNumErrors
            minNumErrors = numErrors;
            bestCandidate = jj;
        end
        minNumErrors = min(minNumErrors,numErrors);
    end
    uniqueGuessLabels(bestCandidate) = [];
    totalNumErrors = totalNumErrors + minNumErrors;
    if isempty(uniqueGuessLabels)
        break
    end
end
SpeakerErrorRate = totalNumErrors/numel(trueLabel)

SpeakerErrorRate = 0
```

References

- [1] 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.
- [2] Sell, G., Snyder, D., McCree, A., Garcia-Romero, D., Villalba, J., Maciejewski, M., Manohar, V., Dehak, N., Povey, D., Watanabe, S., Khudanpur, S. (2018) Diarization is Hard: Some Experiences and Lessons Learned for the JHU Team in the Inaugural DIHARD Challenge. Proc. Interspeech 2018, 2808-2812, DOI: 10.21437/Interspeech.2018-1893.

Train Spoken Digit Recognition Network Using Out-of-Memory Features

This example trains a spoken digit recognition network on out-of-memory auditory spectrograms using a transformed datastore. In this example, you extract auditory spectrograms from audio using `audioDatastore` and `audioFeatureExtractor`, and you write them to disk. You then use a `signalDatastore` to access the features during training. The workflow is useful when the training features do not fit in memory. In this workflow, you only extract features once, which speeds up your workflow if you are iterating on the deep learning model design.

Data

Download the Free Spoken Digit Data Set (FSDD). FSDD consists of 2000 recordings of four speakers saying the numbers 0 through 9 in English.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "FSDD.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
dataset = fullfile(dataFolder, "FSDD");
```

Create an `audioDatastore` that points to the dataset.

```
ads = audioDatastore(dataset, IncludeSubfolders=true);
```

Display the classes and the number of examples in each class.

```
[~, filenames] = fileparts(ads.Files);
ads.Labels = categorical(extractBefore(filenames, '_'));
summary(ads.Labels)
```

```

0      200
1      200
2      200
3      200
4      200
5      200
6      200
7      200
8      200
9      200
```

Split the FSDD into training and test sets. Allocate 80% of the data to the training set and retain 20% for the test set. You use the training set to train the model and the test set to validate the trained 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
```

Reduce Training Dataset

To train the network with the entire dataset and achieve the highest possible accuracy, set `speedupExample` to `false`. To run this example quickly, set `speedupExample` to `true`.

```
speedupExample =  ;
if speedupExample
    adsTrain = splitEachLabel(adsTrain,2);
    adsTest = splitEachLabel(adsTest,2);
end
```

Set up Auditory Spectrogram Extraction

The CNN accepts mel-frequency spectrograms.

Define parameters used to extract mel-frequency spectrograms. Use 220 ms windows with 10 ms hops between windows. Use a 2048-point DFT and 40 frequency bands.

```
fs = 8000;

frameDuration = 0.22;
frameLength = round(frameDuration*fs);

hopDuration = 0.01;
hopLength = round(hopDuration*fs);

segmentLength = 8192;
```

```
numBands = 40;
fftLength = 2048;
```

Create an `audioFeatureExtractor` object to compute mel-frequency spectrograms from input audio signals.

```
afe = audioFeatureExtractor(melSpectrum=true,SampleRate=fs, ...
    Window=hamming(frameLength,"periodic"),OverlapLength=frameLength - hopLength, ...
    FFTLength=fftLength);
```

Set the parameters for the mel-frequency spectrogram.

```
setExtractorParameters(afe,"melSpectrum",NumBands=numBands,FrequencyRange=[50 fs/2],WindowNormalL
```

Create a transformed datastore that computes mel-frequency spectrograms from audio data. The supporting function, `getSpeechSpectrogram` on page 1-676, standardizes the recording length and normalizes the amplitude of the audio input. `getSpeechSpectrogram` uses the `audioFeatureExtractor` object `afe` to obtain the log-based mel-frequency spectrograms.

```
adsSpecTrain = transform(adsTrain,@(x)getSpeechSpectrogram(x,afe,segmentLength));
```

Write Auditory Spectrograms to Disk

Use `writeall` to write auditory spectrograms to disk. Set `UseParallel` to true to perform writing in parallel.

```
outputLocation = fullfile(tempdir,"FSDD_Features");
writeall(adsSpecTrain,outputLocation,WriteFcn=@myCustomWriter,UseParallel=true);
```

Set up Training Signal Datastore

Create a `signalDatastore` that points to the out-of-memory features. The read function returns a spectrogram/label pair.

```
sds = signalDatastore(outputLocation,IncludeSubfolders=true, ...
    SignalVariableNames=["spec","label"],ReadOutputOrientation="row");
```

Validation Data

The validation dataset fits into memory. Precompute validation features.

```
adsTestT = transform(adsTest,@(x){getSpeechSpectrogram(x,afe,segmentLength)});
XTest = readall(adsTestT);
XTest = cat(4,XTest{:});
```

Get the validation labels.

```
YTest = adsTest.Labels;
```

Define CNN Architecture

Construct a small CNN 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(XTest);
specSize = sz(1:2);
```

```
imageSize = [specSize 1];
numClasses = numel(categories(YTest));
dropoutProb = 0.2;
numF = 12;
layers = [
    imageInputLayer(imageSize,Normalization="none")

    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(YTest));
];
```

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. To use the parallel pool to read the transformed datastore, set `DispatchInBackground` to `true`. For more information, see `trainingOptions` (Deep Learning Toolbox).

```
miniBatchSize = 50;
options = trainingOptions("adam", ...
    InitialLearnRate=1e-4, ...
    MaxEpochs=30, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropFactor=0.1, ...
    LearnRateDropPeriod=15, ...
    MiniBatchSize=miniBatchSize, ...
    Shuffle="every-epoch", ...
    Plots="training-progress", ...
    Verbose=false, ...
```

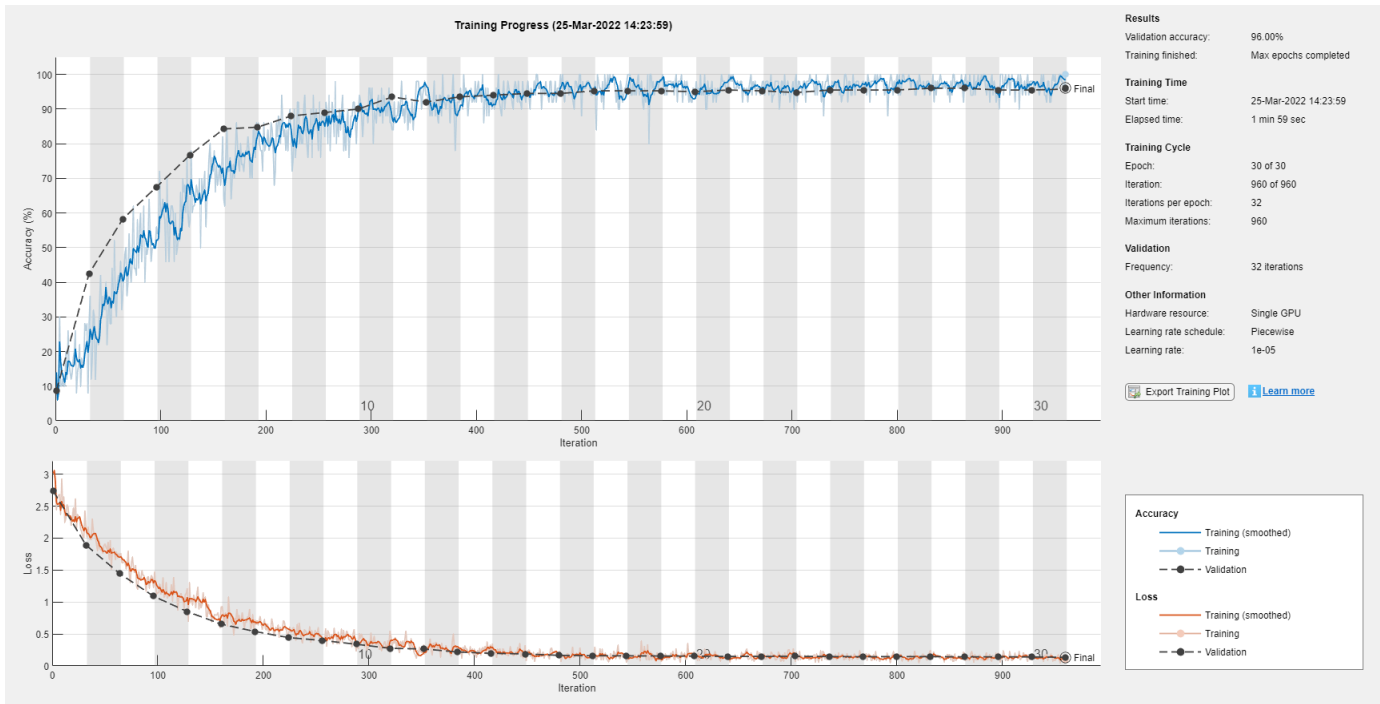
```

ValidationData={XTest,YTest}, ...
ValidationFrequency=ceil(numel(adsTrain.Files)/miniBatchSize), ...
ExecutionEnvironment="auto", ...
DispatchInBackground=true);

```

Train the network by passing the training datastore to `trainNetwork`.

```
trainedNet = trainNetwork(sds, layers, options);
```



Use the trained network to predict the digit labels for the test set.

```

[Ypredicted,probs] = classify(trainedNet,XTest);
cnnAccuracy = sum(Ypredicted==YTest)/numel(YTest)*100

```

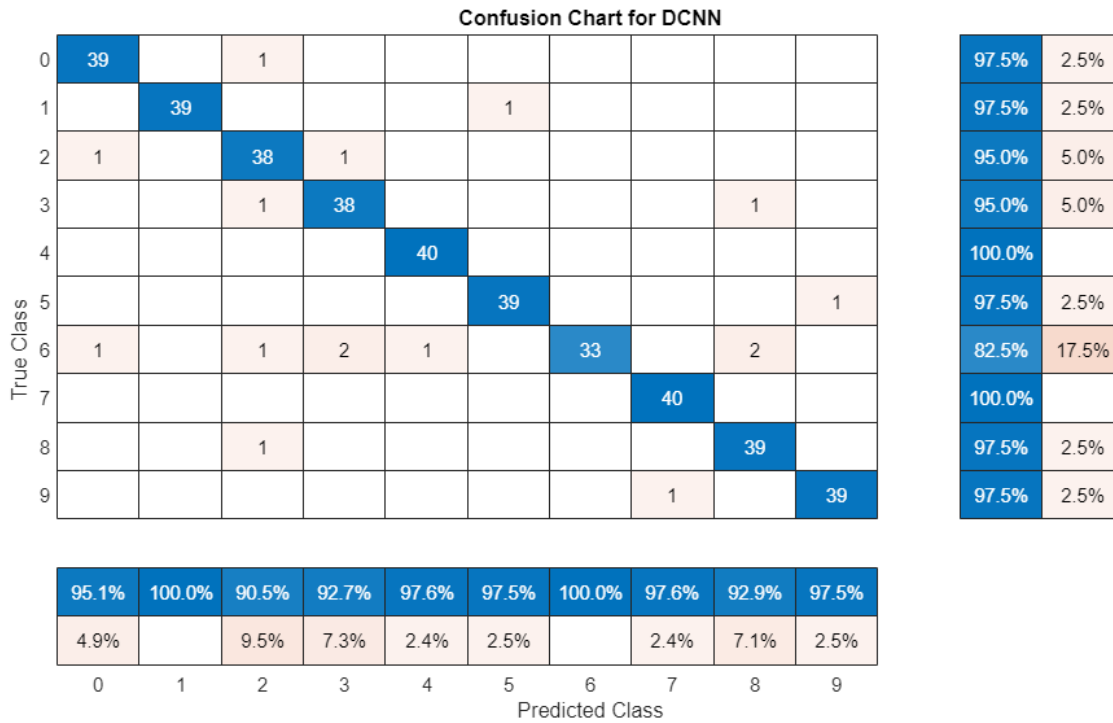
```
cnnAccuracy = 96
```

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 1.5 1.5]);
confusionchart(YTest,Ypredicted, ...
    Title="Confusion Chart for DCNN", ...
    ColumnSummary="column-normalized",RowSummary="row-normalized");

```



Supporting Functions

Get Speech Spectrograms

```
function X = getSpeechSpectrogram(x,afe,segmentLength)
% getSpeechSpectrogram(x,afe,params) computes a speech spectrogram for the
% signal x using the audioFeatureExtractor afe.
```

```
x = scaleAndResize(single(x),segmentLength);
```

```
spec = extract(afe,x).';
```

```
X = log10(spec + 1e-6);
```

```
end
```

Scale and Resize

```
function x = scaleAndResize(x,segmentLength)
% scaleAndResize(x,segmentLength) scales x by its max absolute value and forces
% its length to be segmentLength by trimming or zero-padding.
```

```
L = segmentLength;
```

```
N = size(x,1);
```

```
if N > L
```

```
    x = x(1:L,:);
```

```
elseif N < L
```

```
    pad = L - N;
```

```
    prepad = floor(pad/2);  
    postpad = ceil(pad/2);  
    x = [zeros(prepad,1);x;zeros(postpad,1)];  
end  
x = x./max(abs(x));
```

```
end
```

Custom Write Function

```
function myCustomWriter(spec,writeInfo,~)  
% myCustomWriter(spec,writeInfo,~) writes an auditory spectrogram/label  
% pair to MAT files.  
  
filename = strrep(writeInfo.SuggestedOutputName, ".wav", ".mat");  
label = writeInfo.ReadInfo.Label;  
save(filename, "label", "spec");  
  
end
```

See Also

[audioFeatureExtractor](#) | [audioDatastore](#) | [signalDatastore](#)

Related Examples

- “Train Spoken Digit Recognition Network Using Out-of-Memory Audio Data” on page 1-678

Train Spoken Digit Recognition Network Using Out-of-Memory Audio Data

This example trains a spoken digit recognition network on out-of-memory audio data using a transformed datastore. In this example, you apply a random pitch shift to audio data used to train a convolutional neural network (CNN). For each training iteration, the audio data is augmented using the `audioDataAugmenter` object and then features are extracted using the `audioFeatureExtractor` object. The workflow in this example applies to any random data augmentation used in a training loop. The workflow also applies when the underlying audio data set or training features do not fit in memory.

Data

Download the Free Spoken Digit Data Set (FSDD). FSDD consists of 2000 recordings of four speakers saying the numbers 0 through 9 in English.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "FSDD.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
dataset = fullfile(dataFolder, "FSDD");
```

Create an `audioDatastore` that points to the dataset.

```
ads = audioDatastore(dataset, IncludeSubfolders=true);
```

Decode the file names to set the labels on the datastore. Display the classes and the number of examples in each class.

```
[~, filenames] = fileparts(ads.Files);
ads.Labels = categorical(extractBefore(filenames, '_'));
summary(ads.Labels)
```

```

0      200
1      200
2      200
3      200
4      200
5      200
6      200
7      200
8      200
9      200
```

Split the FSDD into training and test sets. Allocate 80% of the data to the training set and retain 20% for the test set. You use the training set to train the model and the test set to validate the trained 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=10×2 table
```

Label	Count
0	40
1	40
2	40
3	40
4	40
5	40
6	40
7	40
8	40
9	40

Reduce Training Dataset

To train the network with the entire dataset and achieve the highest possible accuracy, set `speedupExample` to `false`. To run this example quickly, set `speedupExample` to `true`.

```

speedupExample =  ;
if speedupExample
    adsTrain = splitEachLabel(adsTrain,2);
    adsTest = splitEachLabel(adsTest,2);
end

```

Transformed Training Datastore

Data Augmentation

Augment the training data by applying pitch shifting with an `audioDataAugmenter` object.

Create an `audioDataAugmenter`. The augmenter applies pitch shifting on an input audio signal with a 0.5 probability. The augmenter selects a random pitch shifting value in the range [-12 12] semitones.

```

augmenter = audioDataAugmenter( ...
    PitchShiftProbability=0.5, ...
    SemitoneShiftRange=[-12 12], ...
    TimeShiftProbability=0, ...
    VolumeControlProbability=0, ...
    AddNoiseProbability=0);

```

Set custom pitch-shifting parameters. Use identity phase locking and preserve formants using spectral envelope estimation with 30th order cepstral analysis.

```
setAugmenterParams(augmenter, "shiftPitch", LockPhase=true, PreserveFormants=true, CepstralOrder=30)
```

Create a transformed datastore that applies data augmentation to the training data.

```
fs = 8000;  
adsAugTrain = transform(adsTrain, @(y) deal(augment(augmenter, y, fs).Audio{1}));
```

Mel Spectrogram Feature Extraction

The CNN accepts mel-frequency spectrograms.

Define parameters used to extract mel-frequency spectrograms. Use 220 ms windows with 10 ms hops between windows. Use a 2048-point DFT and 40 frequency bands.

```
frameDuration = 0.22;  
frameLength = round(frameDuration*fs);
```

```
hopDuration = 0.01;  
hopLength = round(hopDuration*fs);
```

```
segmentLength = 8192;
```

```
numBands = 40;  
fftLength = 2048;
```

Create an `audioFeatureExtractor` object to compute mel-frequency spectrograms from input audio signals.

```
afe = audioFeatureExtractor(melSpectrum=true, SampleRate=fs, ...  
    Window=hamming(frameLength, "periodic"), OverlapLength=frameLength - hopLength, ...  
    FFTLength=fftLength);
```

Set the parameters for the mel-frequency spectrogram.

```
setExtractorParameters(afe, "melSpectrum", NumBands=numBands, FrequencyRange=[50 fs/2], WindowNormalL...
```

Create a transformed datastore that computes mel-frequency spectrograms from pitch-shifted audio data. The supporting function, `getSpeechSpectrogram` on page 1-683, standardizes the recording length and normalizes the amplitude of the audio input. `getSpeechSpectrogram` uses the `audioFeatureExtractor` object (`afe`) to obtain the log-based mel-frequency spectrograms.

```
adsSpecTrain = transform(adsAugTrain, @(x) getSpeechSpectrogram(x, afe, segmentLength));
```

Training Labels

Use an `arrayDatastore` to hold the training labels.

```
labelsTrain = arrayDatastore(adsTrain.Labels);
```

Combined Training Datastore

Create a combined datastore that points to the mel-frequency spectrogram data and the corresponding labels.

```
tdsTrain = combine(adsSpecTrain, labelsTrain);
```

Validation Data

The validation dataset fits into memory. Precompute validation features.

```
adsTestT = transform(adsTest,@(x){getSpeechSpectrogram(x,afe,segmentLength)});
XTest = readall(adsTestT);
XTest = cat(4,XTest{:});
```

Get the validation labels.

```
YTest = adsTest.Labels;
```

Define CNN Architecture

Construct a small CNN 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(XTest);
specSize = sz(1:2);
imageSize = [specSize 1];

numClasses = numel(categories(YTest));

dropoutProb = 0.2;
numF = 12;
layers = [
    imageInputLayer(imageSize,Normalization="none")

    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(YTest));
];

```

Set the hyperparameters to use in training the network. Use a mini-batch size of 128 and a learning rate of $1e-4$. Specify 'adam' optimization. To use the parallel pool to read the transformed datastore, set `DispatchInBackground` to `true`. For more information, see `trainingOptions` (Deep Learning Toolbox).

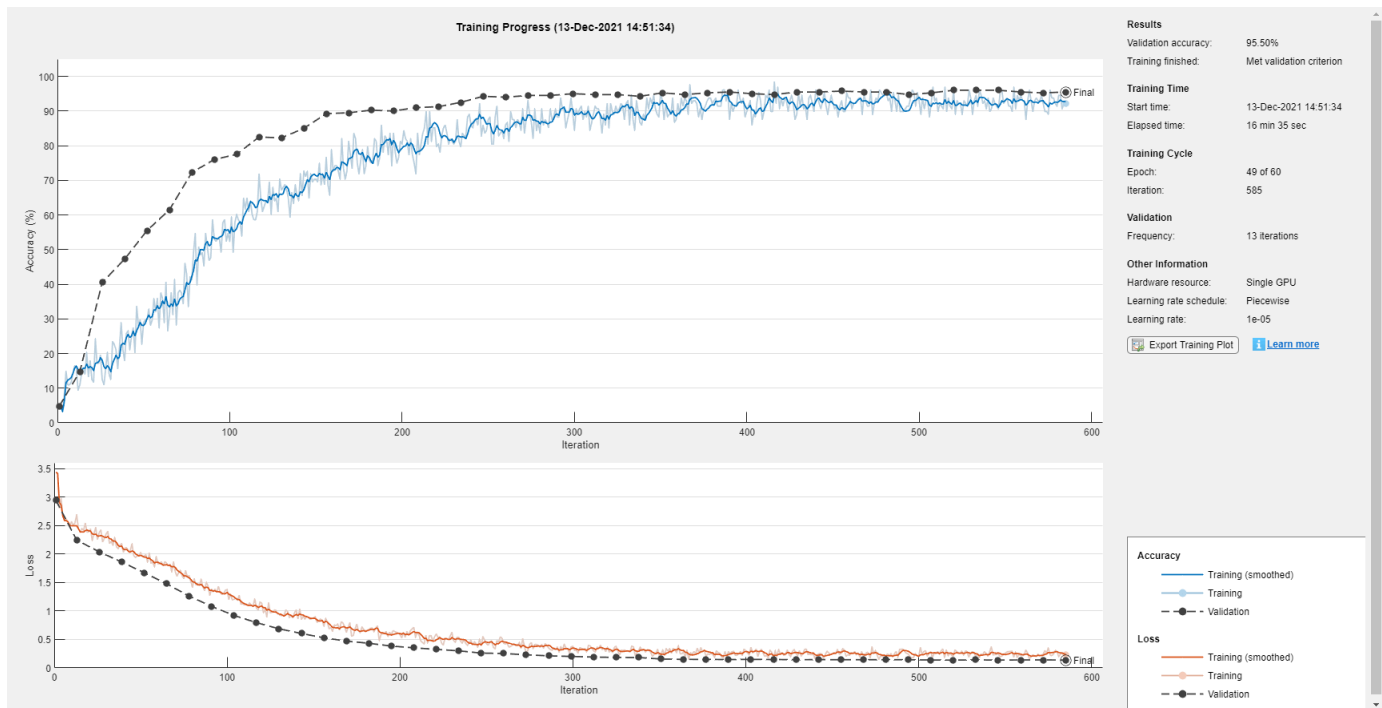
```

miniBatchSize = 128;
options = trainingOptions("adam", ...
    InitialLearnRate=1e-4, ...
    MaxEpochs=60, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropFactor=0.1, ...
    LearnRateDropPeriod=30, ...
    MiniBatchSize=miniBatchSize, ...
    Shuffle="every-epoch", ...
    Plots="training-progress", ...
    Verbose=false, ...
    ValidationData={XTest,YTest}, ...
    ValidationFrequency=ceil(numel(adsTrain.Files)/miniBatchSize), ...
    ValidationPatience=5, ...
    ExecutionEnvironment="auto", ...
    DispatchInBackground=true);

```

Train the network by passing the transformed training datastore to `trainNetwork`.

```
trainedNet = trainNetwork(tdsTrain, layers, options);
```



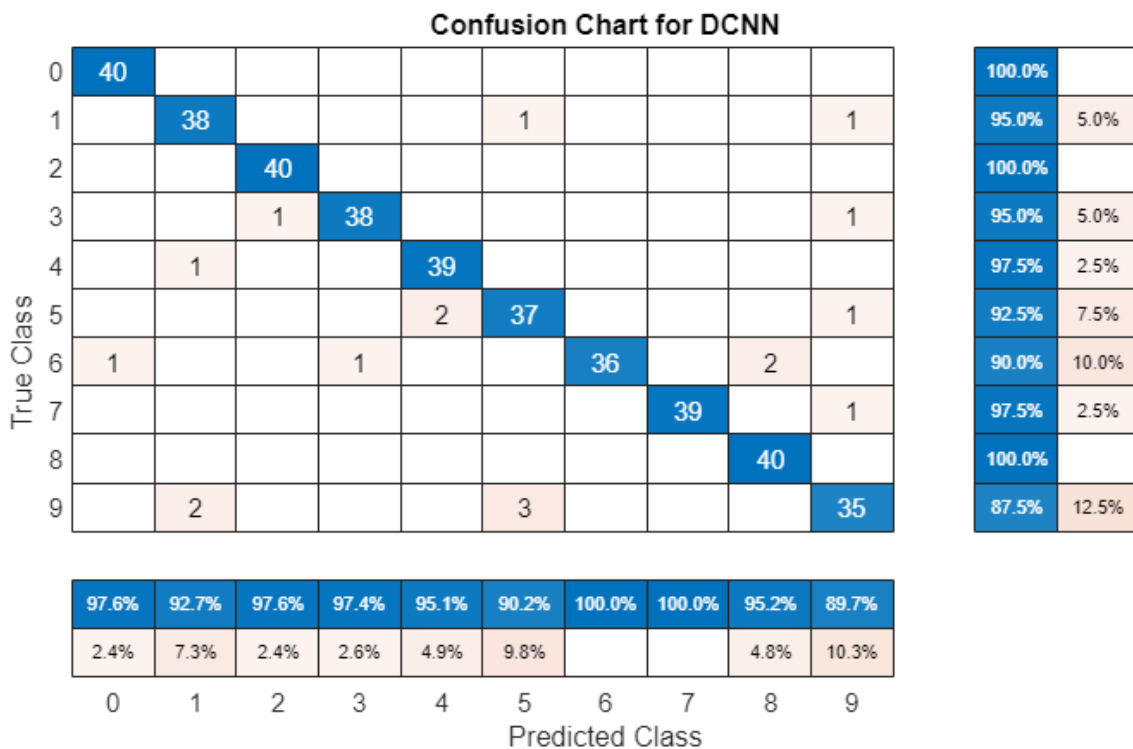
Use the trained network to predict the digit labels for the test set.

```
[Ypredicted,probs] = classify(trainedNet,XTest);
cnnAccuracy = sum(Ypredicted==YTest)/numel(YTest)*100
```

```
cnnAccuracy = 95.5000
```

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]);
confusionchart(YTest,Ypredicted, ...
    Title="Confusion Chart for DCNN", ...
    ColumnSummary="column-normalized",RowSummary="row-normalized");
```



Supporting Functions

Get Speech Spectrograms

```
function X = getSpeechSpectrogram(x,afe,segmentLength)
% getSpeechSpectrogram(x,afe,params) computes a speech spectrogram for the
% signal x using the audioFeatureExtractor afe.
```

```
x = scaleAndResize(single(x),segmentLength);
```

```
spec = extract(afe,x).';
```

```
X = log10(spec + 1e-6);
```

end

Normalize and Resize

```
function x = scaleAndResize(x,segmentLength)
% scaleAndResize(x,segmentLength) scales x by its max absolute value and forces
% its length to be segmentLength by trimming or zero-padding.
```

```
L = segmentLength;
N = size(x,1);
if N > L
    x = x(1:L,:);
elseif N < L
    pad = L - N;
    prepad = floor(pad/2);
    postpad = ceil(pad/2);
    x = [zeros(prepad,1);x;zeros(postpad,1)];
end
x = x./max(abs(x));
```

end

Keyword Spotting in Noise Code Generation with Intel MKL-DNN

This example demonstrates code generation for keyword spotting using a Bidirectional Long Short-Term Memory (BiLSTM) network and mel frequency cepstral coefficient (MFCC) feature extraction. MATLAB® Coder™ with Deep Learning Support enables the generation of a standalone executable (.exe) file. Communication between the MATLAB® (.mlx) file and the generated executable file occurs over asynchronous User Datagram Protocol (UDP). The incoming speech signal is displayed using a timescope. A mask is shown as a blue rectangle surrounding spotted instances of the keyword, YES. For more details on MFCC feature extraction and deep learning network training, visit “Keyword Spotting in Noise Using MFCC and LSTM Networks” on page 1-501.

Example Requirements

- MATLAB® Coder Interface for Deep Learning Support Package
- Intel® 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).

Pretrained Network Keyword Spotting Using MATLAB and Streaming Audio from Microphone

The sample rate of the pretrained network is 16 kHz. Set the window length to 512 samples, with an overlap length of 384 samples, and a hop length defined as the difference between the window and overlap lengths. Define the rate at which the mask is estimated. A mask is generated once for every numHopsPerUpdate audio frames.

```
fs = 16e3;
windowLength = 512;
overlapLength = 384;
hopLength = windowLength - overlapLength;
numHopsPerUpdate = 16;
maskLength = hopLength*numHopsPerUpdate;
```

Create an audioFeatureExtractor object to perform MFCC feature extraction.

```
afe = audioFeatureExtractor('SampleRate',fs, ...
    'Window',hann(windowLength,'periodic'), ...
    'OverlapLength',overlapLength, ...
    'mfcc',true, ...
    'mfccDelta',true, ...
    'mfccDeltaDelta',true);
```

Download and load the pretrained network, as well as the mean (M) and the standard deviation (S) vectors used for Feature Standardization.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/KeywordSpotting.zip';
downloadNetFolder = './';
netFolder = fullfile(downloadNetFolder,'KeywordSpotting');
if ~exist(netFolder,'dir')
```

```
        disp('Downloading pretrained network and audio files (4 files - 7 MB) ...')
        unzip(url,downloadNetFolder)
    end
    load(fullfile(netFolder,'KWSNet.mat'),'KWSNet','M','S');
```

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 equal to the hop length. This enables you to compute a new set of features for every new audio frame from the microphone.

```
frameLength = hopLength;
adr = audioDeviceReader('SampleRate',fs, ...
    'SamplesPerFrame',frameLength);
```

Create a Time Scope to visualize the speech signals and estimated mask.

```
scope = timescope('SampleRate',fs, ...
    'TimeSpanSource','property', ...
    'TimeSpan',5, ...
    'TimeSpanOverrunAction','Scroll', ...
    'BufferLength',fs*5*2, ...
    'ShowLegend',true, ...
    'ChannelNames',{'Speech','Keyword Mask'}, ...
    'YLimits',[-1.2 1.2], ...
    'Title','Keyword Spotting');
```

Initialize a buffer for the audio data, a buffer for the computed features, and a buffer to plot the input audio and the output speech mask.

```
dataBuff = dsp.AsyncBuffer(windowLength);
featureBuff = dsp.AsyncBuffer(numHopsPerUpdate);
plotBuff = dsp.AsyncBuffer(numHopsPerUpdate*windowLength);
```

Perform keyword spotting on speech received from your microphone. To run the loop indefinitely, set `timeLimit` to `Inf`. To stop the simulation, close the scope.

```
timeLimit = 20;
show(scope);
tic
while toc < timeLimit && isVisible(scope)

    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(KWSNet, featureMatrix. ');

    v = double(v) - 1;
    v = repmat(v, hopLength, 1);
    v = v(:);
    v = mode(v);
    predictedMask = repmat(v, numHopsPerUpdate*hopLength, 1);

    data = read(plotBuff);
    scope([data, predictedMask]);

    drawnow limitrate;
end
end

release(adr)
hide(scope)

```

The `helperKeywordSpotting` supporting function encapsulates capturing the audio, feature extraction and network prediction process demonstrated previously. To make feature extraction compatible with code generation, feature extraction is handled by the generated `generateKeywordFeatures` function. To make the network compatible with code generation, the supporting function uses the `coder.loadDeepLearningNetwork` (MATLAB Coder) (MATLAB Coder) function to load the network.

The supporting function uses a `dsp.UDPSender` System object to send the input data along with the output mask predicted by the network to MATLAB. The MATLAB script uses the `dsp.UDPReceiver` System object to receive the input data along with the output mask predicted by the network running in the supporting function.

Generate Executable on Desktop

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

```

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

```

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

```

dlcfg = coder.DeepLearningConfig('mklDnn');
cfg.DeepLearningConfig = dlcfg;

```

Generate the C++ main file required to produce the standalone executable.

```

cfg.GenerateExampleMain = 'GenerateCodeAndCompile';

```

Generate `helperKeywordSpotting`, a supporting function that encapsulates the audio capture, feature extraction, and network prediction processes. You get a warning in the code generation logs that you can disregard because `helperKeywordSpotting` has an infinite loop that continuously looks for an audio frame from MATLAB.

```

codegen helperKeywordSpotting -config cfg -report

```

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

```
Warning in ==> helperKeywordSpotting Line: 73 Column: 1  
Code generation successful (with warnings): View report
```

Prepare Dependencies and Run the Generated Executable

In this section, you generate all the required dependency files and put them into a single folder. During the build process, MATLAB Coder generates `buildInfo.mat`, a file that contains the compilation and run-time dependency information for the standalone executable.

Set the project name to `helperKeywordSpotting`.

```
projName = 'helperKeywordSpotting';  
packageName = [projName, 'Package'];  
if ispc  
    exeName = [projName, '.exe'];  
else  
    exeName = projName;  
end
```

Load `buildinfo.mat` and use `packNGo` (MATLAB Coder) to produce a `.zip` package.

```
load(['codegen',filesep,'exe',filesep,projName,filesep,'buildInfo.mat']);  
packNGo(buildInfo,'fileName',[packageName, '.zip'],'minimalHeaders',false);
```

Unzip the package and place the executable file in the unzipped directory.

```
unzip([packageName, '.zip'],packageName);  
copyfile(exeName, packageName, 'f');
```

To invoke a standalone executable that depends on the MKL-DNN Dynamic Link Library, append the path to the MKL-DNN library location to the environment variable `PATH`.

```
setenv('PATH',[getenv('INTEL_MKLDNN'),filesep,'lib',pathsep,getenv('PATH')]);
```

Run the generated executable.

```
if ispc  
    system(['start cmd /k "title ',packageName,' && cd ',packageName,' && ',exeName]);  
else  
    cd(packageName);  
    system(['./',exeName,' &']);  
    cd ..;  
end
```

Perform Keyword Spotting Using Deployed Code

Create a `dsp.UDPReceiver` System object to receive speech data and the predicted speech mask from the standalone executable. Each UDP packet received from the executable consists of `maskLength` mask samples and speech samples. 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;  
speechDataLength = maskLength;  
numElementsPerUDPPacket = maskLength + speechDataLength;  
  
maxUDPMessageLength = floor(65507/sizeofFloatInBytes);  
samplesPerPacket = 1 + numElementsPerUDPPacket;
```

```

numPackets = floor(maxUDPMessageLength/samplesPerPacket);
bufferSize = numPackets*samplesPerPacket*sizeofFloatInBytes;

UDPReceive = dsp.UDPReceiver('LocalIPPort',20000, ...
    'MessageDataType','single', ...
    'MaximumMessageLength',samplesPerPacket, ...
    'ReceiveBufferSize',bufferSize);

```

To run the keyword spotting indefinitely, set `timelimit` to `Inf`. To stop the simulation, close the scope.

```

tic;
timelimit = 20;
show(scope);

while toc < timelimit && isVisible(scope)
    data = UDPReceive();
    if ~isempty(data)
        plotMask = data(1:maskLength);
        plotAudio = data(maskLength+1 : maskLength+speechDataLength);
        scope([plotAudio,plotMask]);
    end
    drawnow limitrate;
end

hide(scope);

```

Release the system objects and terminate the standalone executable.

```

release(UDPReceive);
release(scope);
if ispc
    system(['taskkill /F /FI "WindowTitle eq ',projName,'* " /T']);
else
    system(['killall ',exeName]);
end

```

```

SUCCESS: The process with PID 4644 (child process of PID 21188) has been terminated.
SUCCESS: The process with PID 20052 (child process of PID 21188) has been terminated.
SUCCESS: The process with PID 21188 (child process of PID 22940) has been terminated.

```

Evaluate Execution Time Using Alternative MEX Function Workflow

A similar workflow involves using a MEX file instead of the standalone executable. Perform MEX profiling to measure the computation time for the workflow.

Create a code generation configuration object to generate the MEX function. 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 deep learning configuration object to the code generation configuration object.

```

dlcfg = coder.DeepLearningConfig('mklDnn');
cfg.DeepLearningConfig = dlcfg;

```

Call `codegen` to generate the MEX function for `profileKeywordSpotting`.

```
inputAudioFrame = ones(hopLength,1,'single');
codegen profileKeywordSpotting -config cfg -args {inputAudioFrame} -report
```

Code generation successful: [View report](#)

Measure the execution time of the MATLAB code.

```
x = pinknoise(hopLength,1,'single');
numPredictCalls = 100;
totalNumCalls = numPredictCalls*numHopsPerUpdate;
exeTimeStart = tic;
for call = 1:totalNumCalls
    [outputMask,inputData,plotFlag] = profileKeywordSpotting(x);
end
exeTime = toc(exeTimeStart);
fprintf('MATLAB execution time per %d ms of audio = %0.4f ms\n',int32(1000*numHopsPerUpdate*hopLength),exeTime)
```

MATLAB execution time per 128 ms of audio = 24.9238 ms

Measure the execution time of the MEX function.

```
exeTimeMexStart = tic;
for call = 1:totalNumCalls
    [outputMask,inputData,plotFlag] = profileKeywordSpotting_mex(x);
end
exeTimeMex = toc(exeTimeMexStart);
fprintf('MEX execution time per %d ms of audio = %0.4f ms\n',int32(1000*numHopsPerUpdate*hopLength),exeTimeMex)
```

MEX execution time per 128 ms of audio = 5.2710 ms

Compare total execution time of the standalone executable approach with the MEX function approach. This performance test is done on a machine using an NVIDIA Quadro® P620 (Version 26) GPU and an Intel Xeon W-2133 CPU running at 3.60 GHz.

```
PerformanceGain = exeTime/exeTimeMex
```

PerformanceGain = 4.7285

Keyword Spotting in Noise Code Generation on Raspberry Pi

This example demonstrates code generation for keyword spotting using a Bidirectional Long Short-Term Memory (BiLSTM) network and mel frequency cepstral coefficient (MFCC) feature extraction on Raspberry Pi™. MATLAB® Coder™ with Deep Learning Support enables the generation of a standalone executable (.elf) file on Raspberry Pi. Communication between MATLAB® (.mlx) file and the generated executable file occurs over asynchronous User Datagram Protocol (UDP). The incoming speech signal is displayed using a `timescope`. A mask is shown as a blue rectangle surrounding spotted instances of the keyword, YES. For more details on MFCC feature extraction and deep learning network training, visit “Keyword Spotting in Noise Using MFCC and LSTM Networks” on page 1-501.

Example Requirements

- MATLAB® Coder Interface for Deep Learning Support Package
- ARM processor that supports the NEON extension
- ARM Compute Library version 20.02.1 (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).

Pretrained Network Keyword Spotting Using MATLAB® and Streaming Audio from Microphone

The sample rate of the pretrained network is 16 kHz. Set the window length to 512 samples, with an overlap length of 384 samples, and a hop length defined as the difference between the window and overlap lengths. Define the rate at which the mask is estimated. A mask is generated once for every `numHopsPerUpdate` audio frames.

```
fs = 16e3;
windowLength = 512;
overlapLength = 384;
hopLength = windowLength - overlapLength;
```

```
numHopsPerUpdate = 16;
maskLength = hopLength * numHopsPerUpdate;
```

Create an `audioFeatureExtractor` object to perform MFCC feature extraction.

```
afe = audioFeatureExtractor('SampleRate',fs, ...
    'Window',hann(windowLength,'periodic'), ...
    'OverlapLength',overlapLength, ...
    'mfcc',true, ...
    'mfccDelta',true, ...
    'mfccDeltaDelta',true);
```

Download and load the pretrained network, as well as the mean (M) and the standard deviation (S) vectors used for feature standardization.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/KeywordSpotting.zip';
downloadNetFolder = './';
netFolder = fullfile(downloadNetFolder,'KeywordSpotting');
if ~exist(netFolder,'dir')
```

```
        disp('Downloading pretrained network and audio files (4 files - 7 MB) ...')
        unzip(url,downloadNetFolder)
    end
    load(fullfile(netFolder,'KWSNet.mat'),'KWSNet','M','S');
```

Call `generateMATLABFunction` on the `audioFeatureExtractor` object to create the feature extraction function.

```
generateMATLABFunction(afe,'generateKeywordFeatures','IsStreaming',true);
```

Define an Audio Device Reader System object™ to read audio from your microphone. Set the frame length equal to the hop length. This enables the computation of a new set of features for every new audio frame received from the microphone.

```
frameLength = hopLength;
adr = audioDeviceReader('SampleRate',fs, ...
    'SamplesPerFrame',frameLength,'OutputDataType','single');
```

Create a Time Scope to visualize the speech signals and estimated mask.

```
scope = timescope('SampleRate',fs, ...
    'TimeSpanSource','property', ...
    'TimeSpan',5, ...
    'TimeSpanOverrunAction','Scroll', ...
    'BufferLength',fs*5*2, ...
    'ShowLegend',true, ...
    'ChannelNames',{'Speech','Keyword Mask'}, ...
    'YLimits',[-1.2 1.2], ...
    'Title','Keyword Spotting');
```

Initialize a buffer for the audio data, a buffer for the computed features, and a buffer to plot the input audio and the output speech mask.

```
dataBuff = dsp.AsyncBuffer(windowLength);
featureBuff = dsp.AsyncBuffer(numHopsPerUpdate);
plotBuff = dsp.AsyncBuffer(numHopsPerUpdate*windowLength);
```

Perform keyword spotting on speech received from your microphone. To run the loop indefinitely, set `timeLimit` to `Inf`. To stop the simulation, close the scope.

```
show(scope);
timeLimit = 20;
tic
while toc < timeLimit && isVisible(scope)

    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(KWSNet,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]);

    drawnow limitrate;
end
end
hide(scope)

```

The helper `KeywordSpottingRaspi` supporting function encapsulates the feature extraction and network prediction process demonstrated previously. To make feature extraction compatible with code generation, feature extraction is handled by the generated `generateKeywordFeatures` function. To make the network 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 receive the captured audio from MATLAB® and uses a `dsp.UDPSender` System object to send the input speech signal along with the estimated mask predicted by the network to MATLAB®. Similarly, the MATLAB® live script uses the `dsp.UDPSender` System object to send the captured speech signal to the executable running on Raspberry Pi and the `dsp.UDPReceiver` System object to receive the speech signal and estimated mask from Raspberry Pi.

Generate Executable on Raspberry Pi

Replace the `hostIPAddress` with your machine's address. Your Raspberry Pi sends the input speech signal and estimated mask to the specified 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 = '20.02.1';
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;
```

Generate the C++ main file required to produce the standalone executable.

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

Generate C++ code for `helperKeywordSpottingRaspi` on your Raspberry Pi.

```
codegen -config cfg helperKeywordSpottingRaspi -args {hostIPAddress} -report
```

```
    Deploying code. This may take a few minutes.
```

```
Warning: Function 'helperKeywordSpottingRaspi' does not terminate due to an infinite loop.
```

```
Warning in ==> helperKeywordSpottingRaspi Line: 78 Column: 1
```

```
Code generation successful (with warnings): View report
```

Perform Keyword Spotting Using Deployed Code

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

```
applicationName = 'helperKeywordSpottingRaspi';
```

```
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.UDPSender` 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 = '172.18.231.92';  
UDPSend = dsp.UDPSender('RemoteIPPort',26000,'RemoteIPAddress',targetIPAddress);
```

Create a `dsp.UDPReceiver` System object to receive speech data and the predicted speech mask from your Raspberry Pi. Each UDP packet received from the Raspberry Pi consists of `maskLength` mask and speech samples. 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;  
speechDataLength = maskLength;
```



```

numElementsPerUDPPacket = maskLength + speechDataLength;
maxUDPMessageLength = floor(65507/sizeofFloatInBytes);
numPackets = floor(maxUDPMessageLength/numElementsPerUDPPacket);
bufferSize = numPackets*numElementsPerUDPPacket*sizeofFloatInBytes;

UDPReceive = dsp.UDPReceiver("LocalIPPort",21000, ...
    "MessageDataType","single", ...
    "MaximumMessageLength",1+numElementsPerUDPPacket, ...
    "ReceiveBufferSize",bufferSize);

```

Spot the keyword as long as time scope is open or until the time limit is reached. To stop the live detection before the time limit is reached, close the time scope.

```

tic;
show(scope);
timelimit = 20;
while toc < timelimit && isVisible(scope)
    x = adr();
    UDPSend(x);
    data = UDPReceive();
    if ~isempty(data)
        mask = data(1:maskLength);
        dataForPlot = data(maskLength + 1 : numElementsPerUDPPacket);
        scope([dataForPlot,mask]);
    end
    drawnow limitrate;
end

```

Release the system objects and terminate the standalone executable.

```

hide(scope)
release(UDPSend)
release(UDPReceive)
release(scope)
release(adr)
stopExecutable(codertarget.raspi.raspberrypi,exeName)

```

Evaluate Execution Time Using Alternative PIL Function Workflow

To evaluate execution time taken by standalone executable on Raspberry Pi, use a PIL (processor-in-loop) workflow. To perform PIL profiling, generate a PIL function for the supporting function `profileKeywordSpotting`. The `profileKeywordSpotting` is equivalent to `helperKeywordSpottingRaspi`, except that the former returns the speech and predicted speech mask 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 code generation configuration object to generate the PIL function.

```

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 = '20.02.1';

```

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 generate the PIL code. A MEX file named `profileKeywordSpotting_pil` is generated in your current folder.

```
cfg.CodeExecutionProfiling = true;
codegen -config cfg profileKeywordSpotting -args {pinknoise(hopLength,1,'single')} -report

    Deploying code. This may take a few minutes.
### Connectivity configuration for function 'profileKeywordSpotting': 'Raspberry Pi'
Location of the generated elf : /home/pi/remoteBuildDir/MATLAB_ws/R2022a/W/Ex/ExampleManager/spo
Code generation successful: View report
```

Evaluate Raspberry Pi Execution Time

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

```
numPredictCalls = 10;
totalCalls = numHopsPerUpdate * numPredictCalls;

x = pinknoise(hopLength,1,'single');
for k = 1:totalCalls
    [maskReceived,inputSignal,plotFlag] = profileKeywordSpotting_pil(x);
end

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

Terminate the PIL execution.

```
clear profileKeywordSpotting_pil

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

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

Generate an execution profile report to evaluate execution time.

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

```
ans =
```

```
'W:\Ex\ExampleManager\sporwal.Bdoc22a.j1844576\deeplearning_shared-ex18742368\codegen\lib\profil
```

Code Execution Profiling Report

Code Execution Profiling Report for profileKeywordSpotting

The code execution profiling report provides metrics based on data collected from a SIL or PIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. See [Code Execution Profiling](#) for more information.

1. Summary

Total time	278.7215
Unit of time	ms
Command	report(executionProfile, 'Units', 'seconds', 'ScaleFactor', '0.001', 'NumericFormat', '%0.4f');
Timer frequency (ticks per second)	1e+09
Profiling data created	28-Oct-2020 18:52:30

2. Profiled Sections of Code

Section	Maximum Execution Time in ms	Average Execution Time in ms	Maximum Self Time in ms	Average Self Time in ms	Calls	
profileKeywordSpotting_initialize	0.0868	0.0868	0.0868	0.0868	1	
profileKeywordSpotting	26.9524	1.7414	26.9524	1.7414	160	
profileKeywordSpotting_terminate	0.0029	0.0029	0.0029	0.0029	1	

3. Definitions

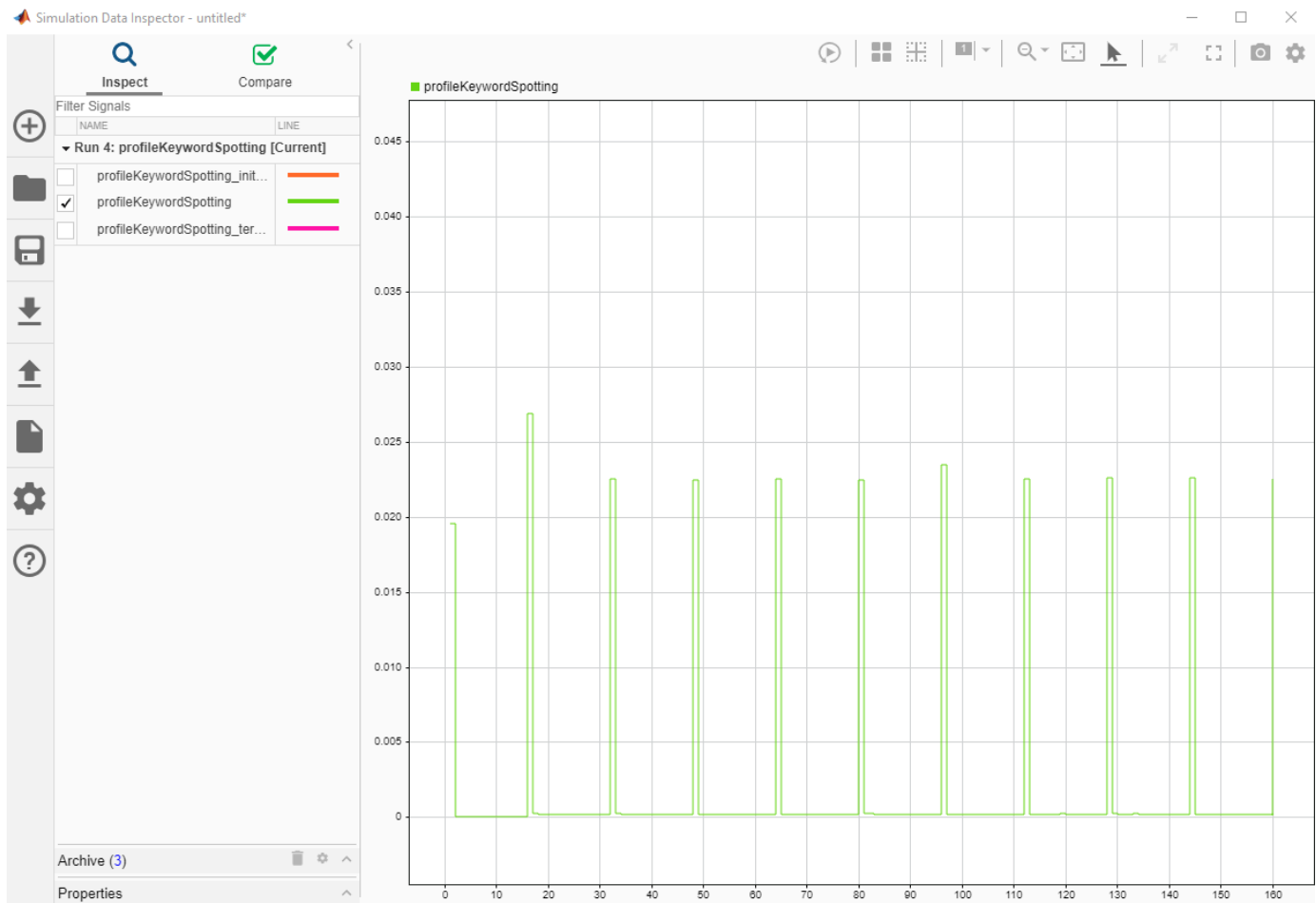
Execution Time: Time between start and end of code section.

Self Time: Execution time, excluding time in child sections.

OK

Help

Plot the Execution Time of each frame from the generated report.



Processing of the first frame took ~20 ms due to initialization overhead costs. The spikes in the time graph at every 16th frame (`numHopsPerUpdate`) correspond to the computationally intensive predict function called every 16th frame. The maximum execution time is ~30 ms, which is below the 128 ms budget for real-time streaming. The performance is measured on Raspberry Pi 4 Model B Rev 1.1.

Dereverberate Speech Using Deep Learning Networks

This example shows how to train a U-Net fully convolutional network (FCN) [1] on page 1-722 to dereverberate a speech signals.

Introduction

Reverberation occurs when a speech signal is reflected off objects in space, causing multiple reflections to build up and eventually leads to degradation of speech quality. Dereverberation is the process of reducing the reverberation effects in a speech signal.

Dereverberate Speech Signal Using Pretrained Network

Before going into the training process in detail, use a pretrained network to dereverberate a speech signal.

Download the pretrained network. This network was trained on 56-speaker versions of the training datasets. The example walks through training on the 28-speaker version.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","dereverbnet.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
netFolder = fullfile(dataFolder,"derevernet");
load(fullfile(netFolder,"dereverbNet.mat"));
```

Listen to a clean speech signal sampled at 16 kHz.

```
[cleanAudio,fs] = audioread("clean_signal.wav");

sound(cleanAudio,fs)
```

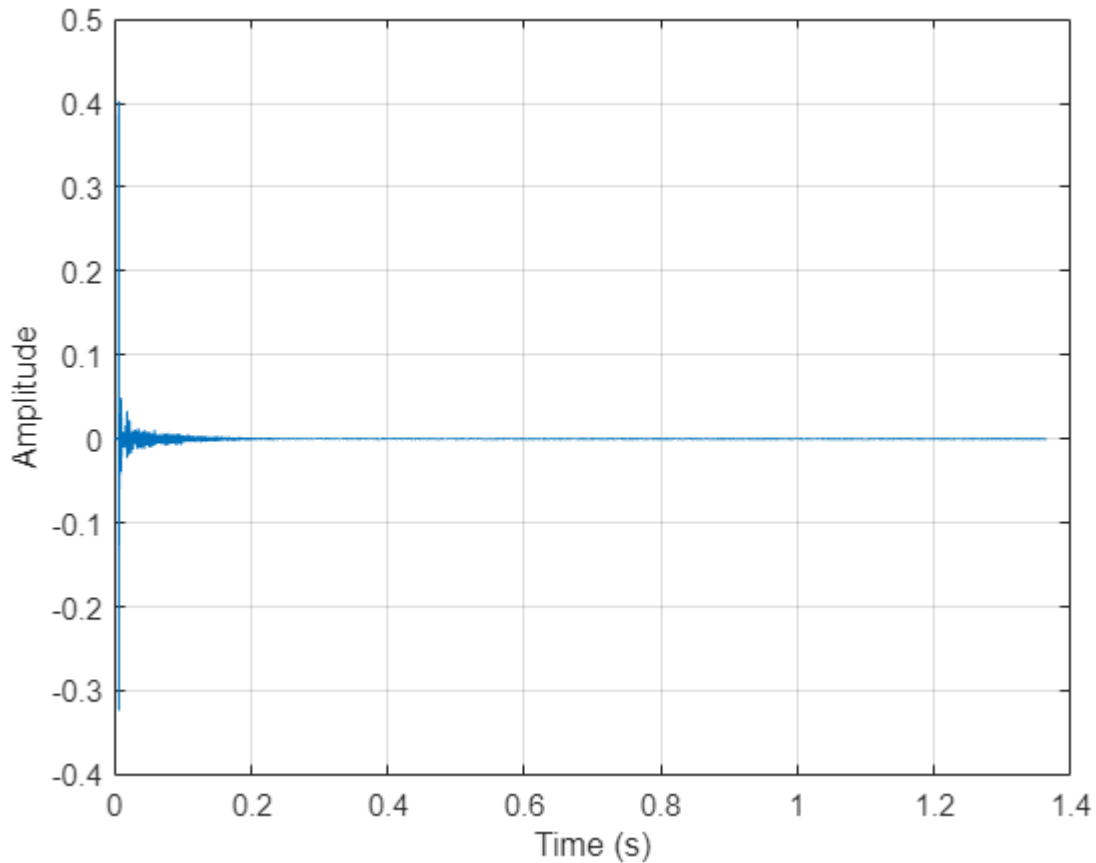
An acoustic path can be modelled using a room impulse response. You can model reverberation by convolving an anechoic signal with a room impulse response.

Load and plot a room impulse response.

```
[rirAudio,fsR] = audioread("room_impulse_response.wav");

tAxis = (1/fsR)*(0:numel(rirAudio)-1);

figure
plot(tAxis,rirAudio)
xlabel("Time (s)")
ylabel("Amplitude")
grid on
```



Convolve the clean speech with the room impulse response to obtain reverberated speech. Align the lengths and amplitudes of the reverberated and clean speech signals.

```
revAudio = conv(cleanAudio,rirAudio);

revAudio = revAudio(1:numel(cleanAudio));
revAudio = revAudio.*(max(abs(cleanAudio))/max(abs(revAudio))));
```

Listen to the reverberated speech signal.

```
sound(revAudio,fs)
```

The input to the pretrained network is the log-magnitude short-time Fourier transform (STFT) of the reverberant audio. The network predicts the log-magnitude STFT of the dereverberated input. To estimate the original time-domain audio signal, you perform an inverse STFT and assume the phase of the reverberant audio.

Use the following parameters to compute the STFT.

```
params.WindowLength = 512;
params.Window = hamming(params.WindowLength,"periodic");
params.OverlapLength = round(0.75*params.WindowLength);
params.FFTLength = params.WindowLength;
```

Use `stft` to compute the one-sided log-magnitude STFT. Use single precision when computing features to better utilize memory usage and to speed up the training. Even though the one-sided STFT yields 257 frequency bins, consider only 256 bins and ignore the highest frequency bin.

```
revAudio = single(revAudio);
audioSTFT = stft(revAudio,Window=params.Window,OverlapLength=params.OverlapLength, ...
    FFTLength=params.FFTLength,FrequencyRange="onesided");
Eps = realmin("single");
reverbFeats = log(abs(audioSTFT(1:end-1,:)) + Eps);
```

Extract the phase of the STFT.

```
phaseOriginal = angle(audioSTFT(1:end-1,:));
```

Each input will have dimensions 256-by-256 (frequency bins by time steps). Split the log-magnitude STFT into segments of 256 time-steps.

```
params.NumSegments = 256;
params.NumFeatures = 256;
totalFrames = size(reverbFeats,2);
chunks = ceil(totalFrames/params.NumSegments);
reverbSTFTSegments = mat2cell(reverbFeats,params.NumFeatures, ...
    [params.NumSegments*ones(1,chunks - 1),(totalFrames - (chunks-1)*params.NumSegments)]);
reverbSTFTSegments{chunks} = reverbFeats(:,end-params.NumSegments + 1:end);
```

Scale the segmented features to the range [-1,1]. Retain the minimum and maximum values used to scale for reconstructing the dereverberated signal.

```
minVals = num2cell(cellfun(@(x)min(x,[],"all"),reverbSTFTSegments));
maxVals = num2cell(cellfun(@(x)max(x,[],"all"),reverbSTFTSegments));

featNorm = cellfun(@(feat,minFeat,maxFeat)2.*(feat - minFeat)./(maxFeat - minFeat) - 1, ...
    reverbSTFTSegments,minVals,maxVals,UniformOutput=false);
```

Reshape the features so that chunks are along the fourth dimension.

```
featNorm = reshape(cell2mat(featNorm),params.NumFeatures,params.NumSegments,1,chunks);
```

Predict the log-magnitude spectra of the reverberated speech signal using the pretrained network.

```
predictedSTFT4D = predict(dereverbNet,featNorm);
```

Reshape to 3-dimensions and scale the predicted STFTs to the original range using the saved minimum-maximum pairs.

```
predictedSTFT = squeeze(mat2cell(predictedSTFT4D,params.NumFeatures,params.NumSegments,1,ones(1, ...
    featDeNorm = cellfun(@(feat,minFeat,maxFeat) (feat + 1).*(maxFeat-minFeat)./2 + minFeat, ...
    predictedSTFT,minVals,maxVals,UniformOutput=false);
```

Reverse the log-scaling.

```
predictedSTFT = cellfun(@exp,featDeNorm,UniformOutput=false);
```

Concatenate the predicted 256-by-256 magnitude STFT segments to obtain the magnitude spectrogram of original length.

```
predictedSTFTAll = predictedSTFT(1:chunks - 1);
predictedSTFTAll = cat(2,predictedSTFTAll{:});
predictedSTFTAll(:,totalFrames - params.NumSegments + 1:totalFrames) = predictedSTFT{chunks};
```

Before taking the inverse STFT, append zeros to the predicted log-magnitude spectrum and the phase in lieu of the highest frequency bin which was excluded when preparing input features.

```
nCount = size(predictedSTFTAll,3);
predictedSTFTAll = cat(1,predictedSTFTAll,zeros(1,totalFrames,nCount));
phase = cat(1,phaseOriginal,zeros(1,totalFrames,nCount));
```

Use the inverse STFT function to reconstruct the dereverberated time-domain speech signal using the predicted log-magnitude STFT and the phase of reverberant speech signal.

```
oneSidedSTFT = predictedSTFTAll.*exp(1j*phase);
dereverbedAudio = istft(oneSidedSTFT, ...
    Window=params.Window,OverlapLength=params.OverlapLength, ...
    FFTLength=params.FFTLength,ConjugateSymmetric=true, ...
    FrequencyRange="onesided");

dereverbedAudio = dereverbedAudio./max(abs([dereverbedAudio;revAudio]));
dereverbedAudio = [dereverbedAudio;zeros(length(revAudio) - numel(dereverbedAudio), 1)];
```

Listen to the dereverberated audio signal.

```
sound(dereverbedAudio,fs)
```

Plot the clean, reverberant, and dereverberated speech signals.

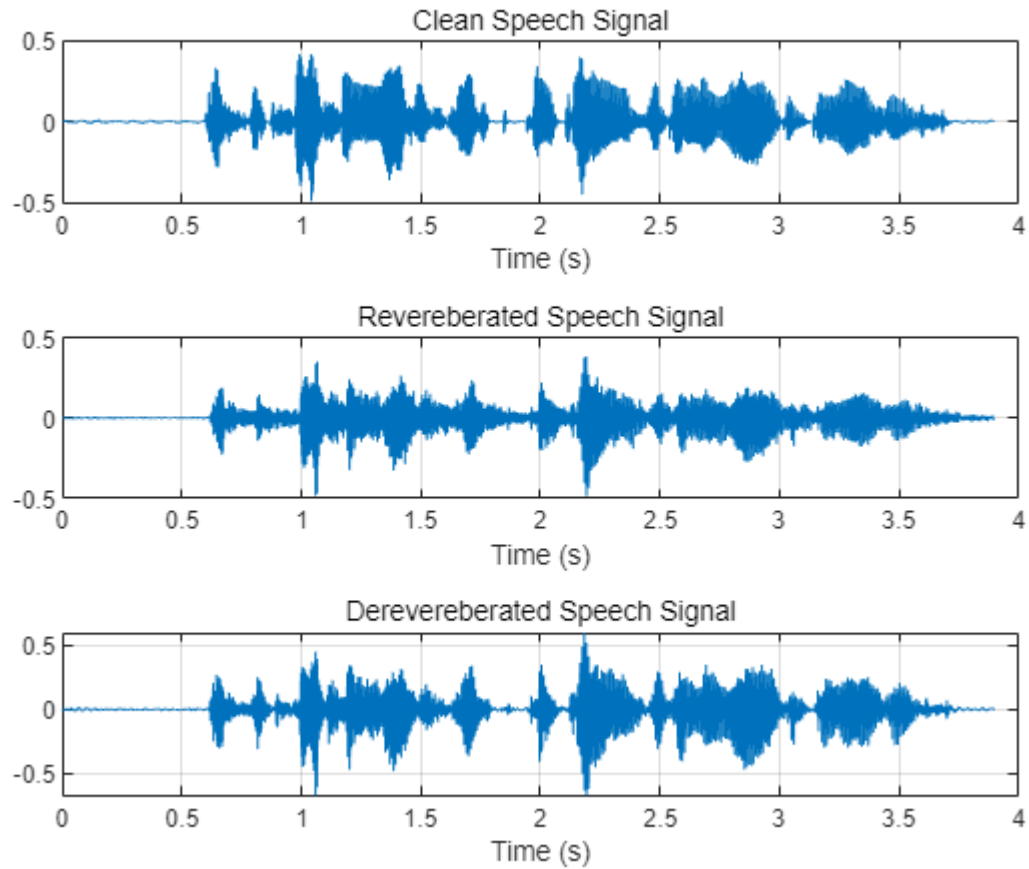
```
t = (1/fs)*(0:numel(cleanAudio)-1);

figure
tiledlayout(3,1)

nexttile
plot(t,cleanAudio)
xlabel("Time (s)")
grid on
subtitle("Clean Speech Signal")

nexttile
plot(t,revAudio)
xlabel("Time (s)")
grid on
subtitle("Reverberated Speech Signal")

nexttile
plot(t,dereverbedAudio)
xlabel("Time (s)")
grid on
subtitle("Dereverberated Speech Signal")
```

Visualize the spectrograms of the clean, reverberant, and dereverberated speech signals.

```
figure(Position=[100,100,800,800])
```

```
tilayout(3,1)
```

```
nexttile
```

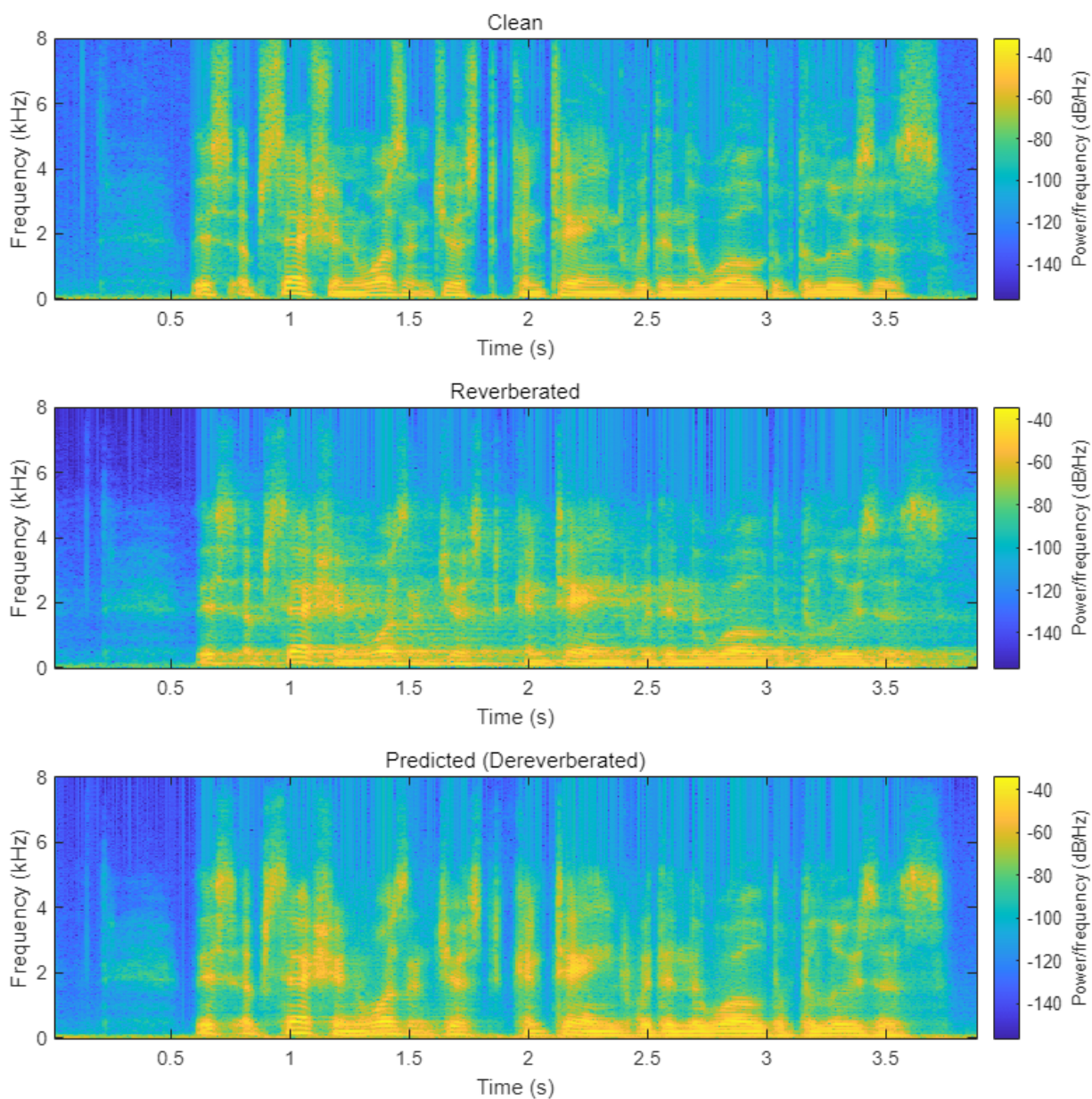
```
spectrogram(cleanAudio,params.Window,params.OverlapLength,params.FFTLength,fs,"yaxis");  
subtitle("Clean")
```

```
nexttile
```

```
spectrogram(revAudio,params.Window,params.OverlapLength,params.FFTLength,fs,"yaxis");  
subtitle("Reverberated")
```

```
nexttile
```

```
spectrogram(dereverbedAudio,params.Window,params.OverlapLength,params.FFTLength,fs,"yaxis");  
subtitle("Predicted (Dereverberated)")
```



Download the Dataset

This example uses the Reverberant Speech Database [2] on page 1-722 and the corresponding Clean Speech Database [3] on page 1-722 to train the network.

Download the clean speech data set.

```
url1 = "https://datashare.is.ed.ac.uk/bitstream/handle/10283/2791/clean_trainset_28spk_wav.zip";  
url2 = "https://datashare.is.ed.ac.uk/bitstream/handle/10283/2791/clean_testset_wav.zip";
```

```
downloadFolder = tempdir;
cleanDataFolder = fullfile(downloadFolder,"DS_10283_2791");

if ~datasetExists(cleanDataFolder)
    disp("Downloading data set (6 GB) ...")
    unzip(url1,cleanDataFolder)
    unzip(url2,cleanDataFolder)
end
```

Downloading data set (6 GB) ...

Download the reverberated speech dataset.

```
url3 = "https://datashare.is.ed.ac.uk/bitstream/handle/10283/2031/reverb_trainset_28spk_wav.zip";
url4 = "https://datashare.is.ed.ac.uk/bitstream/handle/10283/2031/reverb_testset_wav.zip";
downloadFolder = tempdir;
reverbDataFolder = fullfile(downloadFolder,"DS_10283_2031");

if ~datasetExists(reverbDataFolder)
    disp("Downloading data set (6 GB) ...")
    unzip(url3,reverbDataFolder)
    unzip(url4,reverbDataFolder)
end
```

Downloading data set (6 GB) ...

Data Preprocessing and Feature Extraction

Once the data is downloaded, preprocess the downloaded data and extract features before training the DNN model:

- 1 Synthetically generate reverberant data using the `reverberator` object
- 2 Split each speech signal into small segments of 2.072s duration
- 3 Discard segments which contain significant silent regions
- 4 Extract log-magnitude STFTs as predictor and target features
- 5 Scale and reshape features

First, create two `audioDatastore` objects that point to the clean and reverberant speech datasets.

```
adsCleanTrain = audioDatastore(fullfile(cleanDataFolder,"clean_trainset_28spk_wav"),IncludeSubfolders);
adsReverbTrain = audioDatastore(fullfile(reverbDataFolder,"reverb_trainset_28spk_wav"),IncludeSubfolders);
```

Synthetic Reverberant Speech Data Generation

The amount of reverberation in the original data is relatively small. You will augment the reverberant speech data with significant reverberation effects using the `reverberator` object.

Create an `audioDatastore` that points to the clean speech dataset allocated for synthetic reverberant data generation.

```
adsSyntheticCleanTrain = subset(adsCleanTrain,10e3+1:length(adsCleanTrain.Files));
adsCleanTrain = subset(adsCleanTrain,1:10e3);
adsReverbTrain = subset(adsReverbTrain,1:10e3);
```

Resample from 48 kHz to 16 kHz.

```
adsSyntheticCleanTrain = transform(adsSyntheticCleanTrain,@(x)resample(x,16e3,48e3));
adsCleanTrain = transform(adsCleanTrain,@(x)resample(x,16e3,48e3));
adsReverbTrain = transform(adsReverbTrain,@(x)resample(x,16e3,48e3));
```

Combine the two audio datastores, maintaining the correspondence between the clean and reverberant speech samples.

```
adsCombinedTrain = combine(adsCleanTrain,adsReverbTrain);
```

The `applyReverb` on page 1-717 function creates a `reverberator` object, updates the pre delay, decay factor, and wet-dry mix parameters as specified, and then applies reverberation. Use `audioDataAugmenter` to create synthetically generated reverberant data.

```
augmenter = audioDataAugmenter(AugmentationMode="independent",NumAugmentations=1,ApplyAddNoise=0
    ApplyTimeStretch=0,ApplyPitchShift=0,ApplyVolumeControl=0,ApplyTimeShift=0);
algorithmHandle = @(y,preDelay,decayFactor,wetDryMix,samplingRate) ...
    applyReverb(y,preDelay,decayFactor,wetDryMix,samplingRate);

addAugmentationMethod(augmenter,"Reverb",algorithmHandle, ...
    AugmentationParameter={'PreDelay','DecayFactor','WetDryMix','SamplingRate'}, ...
    ParameterRange={[0.15,0.25],[0.2,0.5],[0.3,0.45],[16000,16000]})

augmenter.ReverbProbability = 1;
disp(augmenter)
```

`audioDataAugmenter` with properties:

```
AugmentationMode: "independent"
AugmentationParameterSource: 'random'
NumAugmentations: 1
ApplyTimeStretch: 0
ApplyPitchShift: 0
ApplyVolumeControl: 0
ApplyAddNoise: 0
ApplyTimeShift: 0
ApplyReverb: 1
PreDelayRange: [0.1500 0.2500]
DecayFactorRange: [0.2000 0.5000]
WetDryMixRange: [0.3000 0.4500]
SamplingRateRange: [16000 16000]
```

Create a new `audioDatastore` corresponding to synthetically generated reverberant data by calling `transform` to apply data augmentation.

```
adsSyntheticReverbTrain = transform(adsSyntheticCleanTrain,@(y)deal(augment(augmenter,y,16e3)).Au
```

Combine the two audio datastores.

```
adsSyntheticCombinedTrain = combine(adsSyntheticCleanTrain,adsSyntheticReverbTrain);
```

Next, based on the dimensions of the input features to the network, segment the audio into chunks of 2.072 s duration with an overlap of 50%.

Having too many silent segments can adversely affect the DNN model training. Remove the segments which are mostly silent (more than 50% of the duration) and exclude those from the model training. Do not completely remove silence because the model will not be robust to silent regions and slight reverberation effects could be identified as silence. `detectSpeech` can identify the start and end

points of silent regions. After these two steps, the feature extraction process can be carried out as explained in the first section. `helperFeatureExtract` on page 1-718 implements these steps.

Define the feature extraction parameters. By setting `speedupExample` to `true`, you choose a small subset of the datasets to perform the subsequent steps.

```
speedupExample = ;
params.fs = 16000;
params.WindowdowLength = 512;
params.Window = hamming(params.WindowdowLength,"periodic");
params.OverlapLength = round(0.75*params.WindowdowLength);
params.FFTLength = params.WindowdowLength;
samplesPerMs = params.fs/1000;
params.samplesPerImage = (24+256*8)*samplesPerMs;
params.shiftImage = params.samplesPerImage/2;
params.NumSegments = 256;
params.NumFeatures = 256

params = struct with fields:
  WindowdowLength: 512
  Window: [512x1 double]
  OverlapLength: 384
  FFTLength: 512
  NumSegments: 256
  NumFeatures: 256
  fs: 16000
  samplesPerImage: 33152
  shiftImage: 16576
```

To speed up processing, distribute the preprocessing and feature extraction task across multiple workers using `parfor`.

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(adsCombinedTrain,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, preprocess the audio signal, and then extract the features.

```
if speedupExample
    adsCombinedTrain = shuffle(adsCombinedTrain); %#ok
    adsCombinedTrain = subset(adsCombinedTrain,1:200);

    adsSyntheticCombinedTrain = shuffle(adsSyntheticCombinedTrain);
    adsSyntheticCombinedTrain = subset(adsSyntheticCombinedTrain,1:200);
end
```

```
allCleanFeatures = cell(1,numPar);
allReverbFeatures = cell(1,numPar);

parfor iPartition = 1:numPar
    combinedPartition = partition(adsCombinedTrain,numPar,iPartition);
    combinedSyntheticPartition = partition(adsSyntheticCombinedTrain,numPar,iPartition);

    cPartitionSize = numel(combinedPartition.UnderlyingDatastores{1}.UnderlyingDatastores{1}.Files);
    cSyntheticPartitionSize = numel(combinedSyntheticPartition.UnderlyingDatastores{1}.UnderlyingDatastores{1}.Files);
    partitionSize = cPartitionSize + cSyntheticPartitionSize;

    cleanFeaturesPartition = cell(1,partitionSize);
    reverbFeaturesPartition = cell(1,partitionSize);

    for idx = 1:partitionSize
        if idx <= cPartitionSize
            audios = read(combinedPartition);
        else
            audios = read(combinedSyntheticPartition);
        end
        cleanAudio = single(audios(:,1));
        reverbAudio = single(audios(:,2));
        [featuresClean,featuresReverb] = helperFeatureExtract(cleanAudio,reverbAudio,false,parameters);
        cleanFeaturesPartition{idx} = featuresClean;
        reverbFeaturesPartition{idx} = featuresReverb;
    end
    allCleanFeatures{iPartition} = cat(2,cleanFeaturesPartition{:});
    allReverbFeatures{iPartition} = cat(2,reverbFeaturesPartition{:});
end
```

Analyzing and transferring files to the workers ...done.

```
allCleanFeatures = cat(2,allCleanFeatures{:});
allReverbFeatures = cat(2,allReverbFeatures{:});
```

Normalize the extracted features to the range [-1,1] and then reshape as explained in the first section, using the `featureNormalizeAndReshape` on page 1-719 function.

```
trainClean = featureNormalizeAndReshape(allCleanFeatures);
trainReverb = featureNormalizeAndReshape(allReverbFeatures);
```

Now that you have extracted the log-magnitude STFT features from the training datasets, follow the same procedure to extract features from the validation datasets. For reconstruction purposes, retain the phase of the reverberant speech samples of the validation dataset. In addition, retain the audio data for both the clean and reverberant speech samples in the validation set to use in the evaluation process (next section).

```
adsCleanVal = audioDatastore(fullfile(cleanDataFolder,"clean_testset_wav"),IncludeSubfolders=true);
adsReverbVal = audioDatastore(fullfile(reverbDataFolder,"reverb_testset_wav"),IncludeSubfolders=true);
```

Resample from 48 kHz to 16 kHz.

```
adsCleanVal = transform(adsCleanVal,@(x)resample(x,16e3,48e3));
adsReverbVal = transform(adsReverbVal,@(x)resample(x,16e3,48e3));
```

```
adsCombinedVal = combine(adsCleanVal,adsReverbVal);
```



```

if speedupExample
    adsCombinedVal = shuffle(adsCombinedVal);%#ok
    adsCombinedVal = subset(adsCombinedVal,1:50);
end

allValCleanFeatures = cell(1,numPar);
allValReverbFeatures = cell(1,numPar);
allValReverbPhase = cell(1,numPar);
allValCleanAudios = cell(1,numPar);
allValReverbAudios = cell(1,numPar);

parfor iPartition = 1:numPar
    combinedPartition = partition(adsCombinedVal,numPar,iPartition);

    partitionSize = numel(combinedPartition.UnderlyingDatastores{1}.UnderlyingDatastores{1}.Files);

    cleanFeaturesPartition = cell(1,partitionSize);
    reverbFeaturesPartition = cell(1,partitionSize);
    reverbPhasePartition = cell(1,partitionSize);
    cleanAudiosPartition = cell(1,partitionSize);
    reverbAudiosPartition = cell(1,partitionSize);

    for idx = 1:partitionSize
        audios = read(combinedPartition);

        cleanAudio = single(audios(:,1));
        reverbAudio = single(audios(:,2));

        [a,b,c,d,e] = helperFeatureExtract(cleanAudio,reverbAudio,true,params);

        cleanFeaturesPartition{idx} = a;
        reverbFeaturesPartition{idx} = b;
        reverbPhasePartition{idx} = c;
        cleanAudiosPartition{idx} = d;
        reverbAudiosPartition{idx} = e;
    end
    allValCleanFeatures{iPartition} = cat(2,cleanFeaturesPartition{:});
    allValReverbFeatures{iPartition} = cat(2,reverbFeaturesPartition{:});
    allValReverbPhase{iPartition} = cat(2,reverbPhasePartition{:});
    allValCleanAudios{iPartition} = cat(2,cleanAudiosPartition{:});
    allValReverbAudios{iPartition} = cat(2,reverbAudiosPartition{:});
end

allValCleanFeatures = cat(2,allValCleanFeatures{:});
allValReverbFeatures = cat(2,allValReverbFeatures{:});
allValReverbPhase = cat(2,allValReverbPhase{:});
allValCleanAudios = cat(2,allValCleanAudios{:});
allValReverbAudios = cat(2,allValReverbAudios{:});

```

```
valClean = featureNormalizeAndReshape(allValCleanFeatures);
```

Retain the minimum and maximum values of each feature of the reverberant validation set. You will use these values in the reconstruction process.

```
[valReverb,valMinMaxPairs] = featureNormalizeAndReshape(allValReverbFeatures);
```

Define Neural Network Architecture

A fully convolutional network architecture named **U-Net** was adapted for this speech dereverberation task as proposed in [1] on page 1-722. "U-Net" is an encoder-decoder network with skip connections. In the U-Net model, each layer downsamples its input (stride of 2) until a bottleneck layer is reached (encoding path). In subsequent layers, the input is upsampled by each layer until the output is returned to the original shape (decoding path). To minimize the loss of low-level information during the downsampling process, connections are made between the mirrored layers by directly concatenating outputs of corresponding layers (*skip connections*).

Define the network architecture and return the layer graph with connections.

```

params.WindowdowLength = 512;
params.FFTLength = params.WindowdowLength;
params.NumFeatures = params.FFTLength/2;
params.NumSegments = 256;

filterH = 6;
filterW = 6;
numChannels = 1;
nFilters = [64,128,256,512,512,512,512];

inputLayer = imageInputLayer([params.NumFeatures,params.NumSegments,numChannels], ...
    Normalization="none",Name="input");
layers = inputLayer;

% U-Net squeezing path
layers = [layers;
    convolution2dLayer([filterH,filterW],nFilters(1),Stride=2,Padding="same",Name="conv"+string(1)),
    leakyReluLayer(0.2,Name="leaky-relu"+string(1))];

for ii = 2:8
    layers = [layers;
        convolution2dLayer([filterH,filterW],nFilters(ii),Stride=2,Padding="same",Name="conv"+string(ii)),
        batchNormalizationLayer(Name="batchnorm"+string(ii))];%#ok
    if ii ~= 8
        layers = [layers;leakyReluLayer(0.2,Name="leaky-relu"+string(ii))];%#ok
    else
        layers = [layers;reluLayer(Name="relu"+string(ii))];%#ok
    end
end

% U-Net expanding path
for ii = 7:-1:0
    nChannels = numChannels;
    if ii > 0
        nChannels = nFilters(ii);
    end
    layers = [layers;
        transposedConv2dLayer([filterH,filterW],nChannels,Stride=2,Cropping="same",Name="deconv"+string(ii)),
        batchNormalizationLayer(Name="de-batchnorm"+string(ii))];%#ok
    end
    if ii > 4
        layers = [layers;dropoutLayer(0.5,Name="de-dropout"+string(ii))];%#ok
    end
    if ii > 0

```



```

        layers = [layers;
            reluLayer(Name="de-relu"+string(ii));
            concatenationLayer(3,2,Name="concat"+string(ii))];%#ok
    else
        layers = [layers;tanhLayer(Name="de-tanh"+string(ii))];%#ok
    end
end

layers = [layers;regressionLayer(Name="output")];

unetLayerGraph = layerGraph(layers);

% Define skip-connections
for ii = 1:7
    unetLayerGraph = connectLayers(unetLayerGraph,"leaky-relu"+string(ii),"concat"+string(ii)+"/");
end

```

Use `analyzeNetwork` to view the model architecture. This is a good way to visualize the connections between layers.

```
analyzeNetwork(unetLayerGraph);
```

Train the Network

You will use the mean squared error (MSE) between the log-magnitude spectra of the dereverberated speech sample (output of the model) and the corresponding clean speech sample (target) as the loss function. Use the `adam` optimizer and a mini-batch size of 128 for the training. Allow the model to train for a maximum of 50 epochs. If the validation loss doesn't improve for 5 consecutive epochs, terminate the training process. Reduce the learning rate by a factor of 10 every 15 epochs.

Define the training options as below. Change the execution environment and whether to perform background dispatching depending on your hardware availability and whether you have access to Parallel Computing Toolbox™.

```

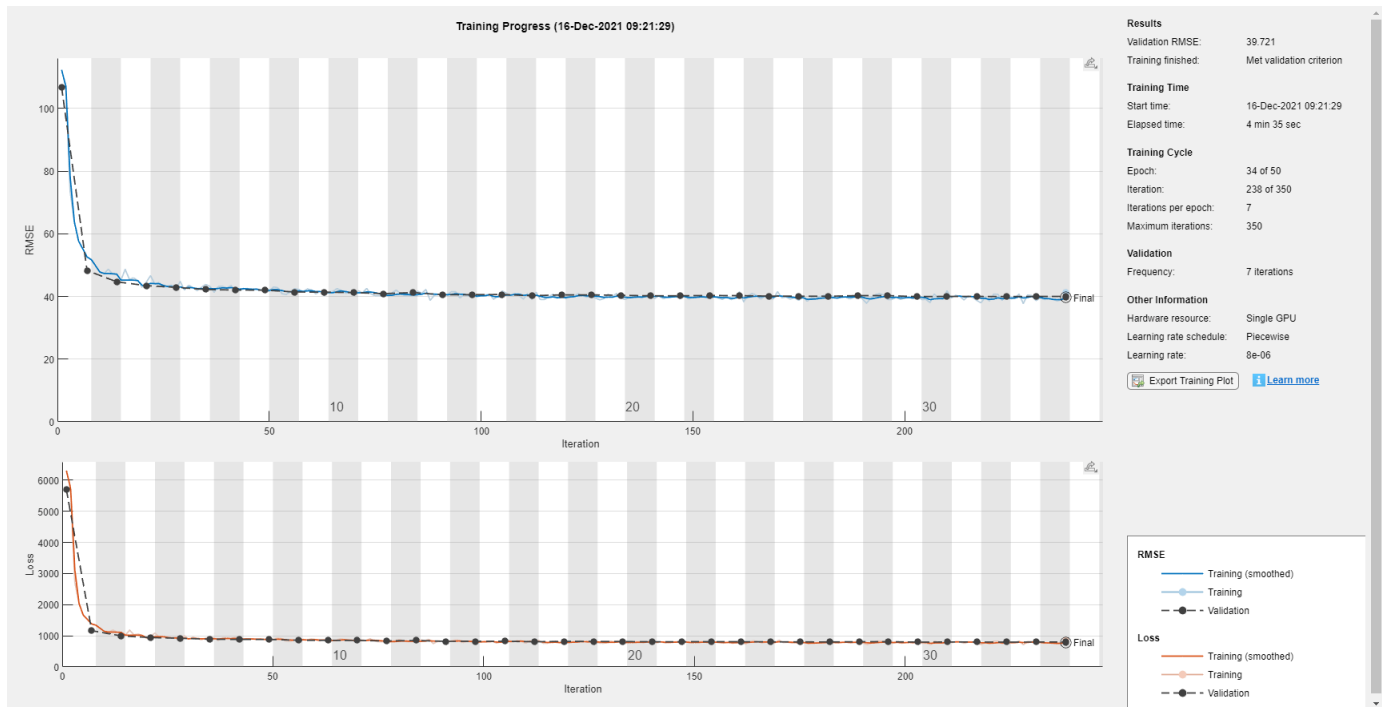
initialLearnRate = 8e-4;
miniBatchSize = 64;

options = trainingOptions("adam", ...
    MaxEpochs=50, ...
    InitialLearnRate=initialLearnRate, ...
    MiniBatchSize=miniBatchSize, ...
    Shuffle="every-epoch", ...
    Plots="training-progress", ...
    Verbose=false, ...
    ValidationFrequency=max(1,floor(size(trainReverb,4)/miniBatchSize)), ...
    ValidationPatience=5, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropFactor=0.1, ...
    LearnRateDropPeriod=15, ...
    ExecutionEnvironment="gpu", ...
    DispatchInBackground=true, ...
    ValidationData={valReverb,valClean});

```

Train the network.

```
dereverbNet = trainNetwork(trainReverb,trainClean,unetLayerGraph,options);
```



Evaluate Network Performance

Prediction and Reconstruction

Predict the log-magnitude spectra of the validation set.

```
predictedSTFT4D = predict(dereverbNet, valReverb);
```

Use the helperReconstructPredictedAudios on page 1-719 function to reconstruct the predicted speech. This function performs actions outlined in the first section.

```
params.WindowLength = 512;
params.Window = hamming(params.WindowLength, "periodic");
params.OverlapLength = round(0.75*params.WindowLength);
params.FFTLength = params.WindowLength;
params.fs = 16000;
```

```
dereverbedAudioAll = helperReconstructPredictedAudios(predictedSTFT4D, valMinMaxPairs, allValReverb);
```

Visualize the log-magnitude STFTs of the clean, reverberant, and corresponding dereverberated speech signals.

```
figure(Position=[100,100,1024,1200])
```

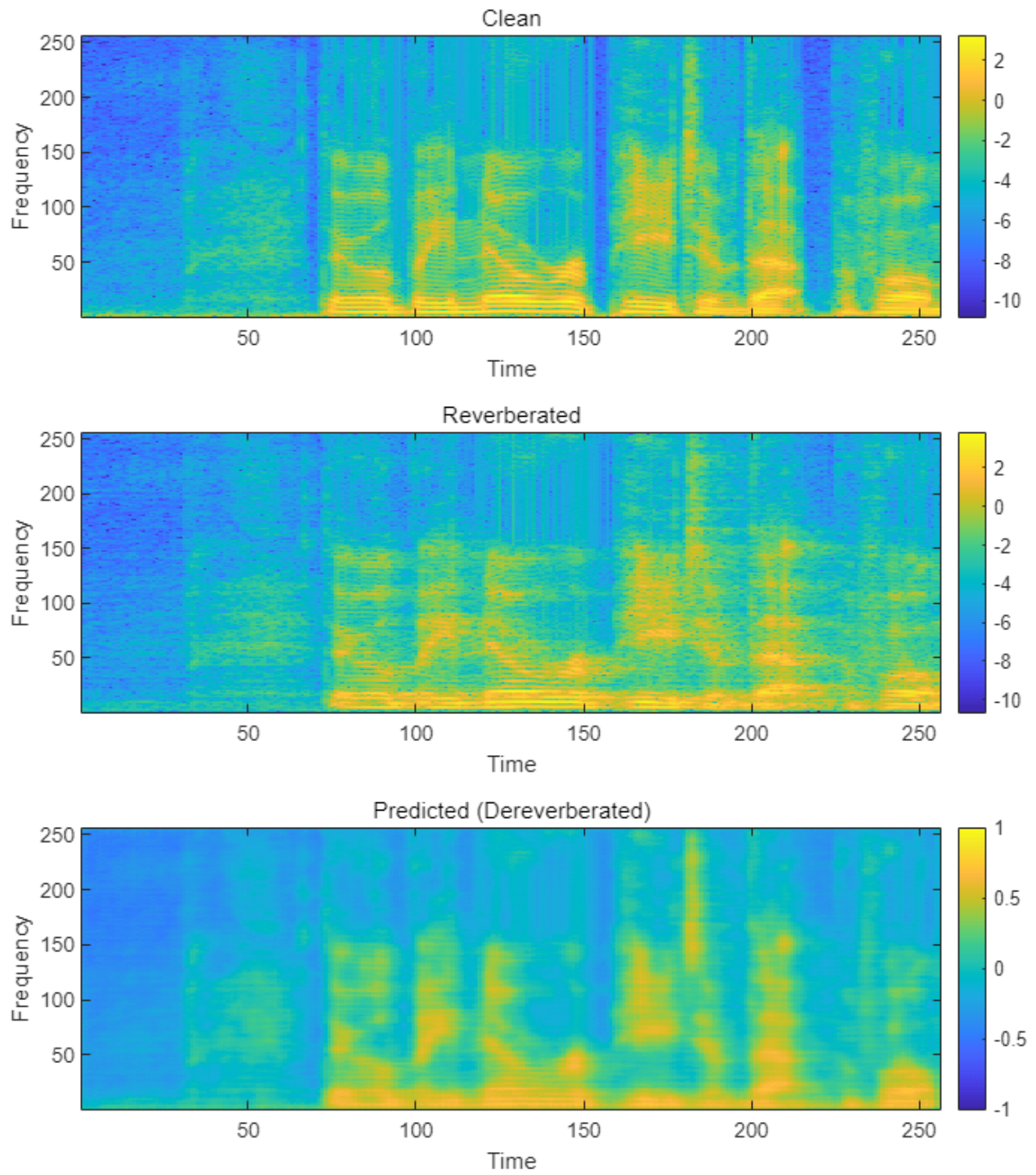
```
 tiledlayout(3,1)
```

```
 nexttile
 imagesc(squeeze(allValCleanFeatures{1}))
 set(gca, Ydir="normal")
 subtitle("Clean")
 xlabel("Time")
 ylabel("Frequency")
```

```
colorbar

nexttile
imagesc(squeeze(allValReverbFeatures{1}))
set(gca,Ydir="normal")
subtitle("Reverberated")
xlabel("Time")
ylabel("Frequency")
colorbar

nexttile
imagesc(squeeze(predictedSTFT4D(:,:,,1)))
set(gca,Ydir="normal")
subtitle("Predicted (Dereverberated)")
xlabel("Time")
ylabel("Frequency")
caxis([-1,1])
colorbar
```



Evaluation Metrics

You will use a subset of objective measures used in [1] on page 1-722 to evaluate the performance of the network. These metrics are computed on the time-domain signals.

- Cepstrum distance (CD) - Provides an estimate of the log spectral distance between two spectra (predicted and clean). Smaller values indicate better quality.
- Log likelihood ratio (LLR) - Linear predictive coding (LPC) based objective measurement. Smaller values indicate better quality.

Compute these measurements for the reverberant speech and the dereverberated speech signals.

```
[summaryMeasuresReconstructed,allMeasuresReconstructed] = calculateObjectiveMeasures(dereverbedA
[summaryMeasuresReverb,allMeasuresReverb] = calculateObjectiveMeasures(allValReverbAudios,allVal
disp(summaryMeasuresReconstructed)
```

```
    avgCdMean: 3.8310
    avgCdMedian: 3.3536
    avgLlrMean: 0.9103
    avgLlrMedian: 0.8007
```

```
disp(summaryMeasuresReverb)
```

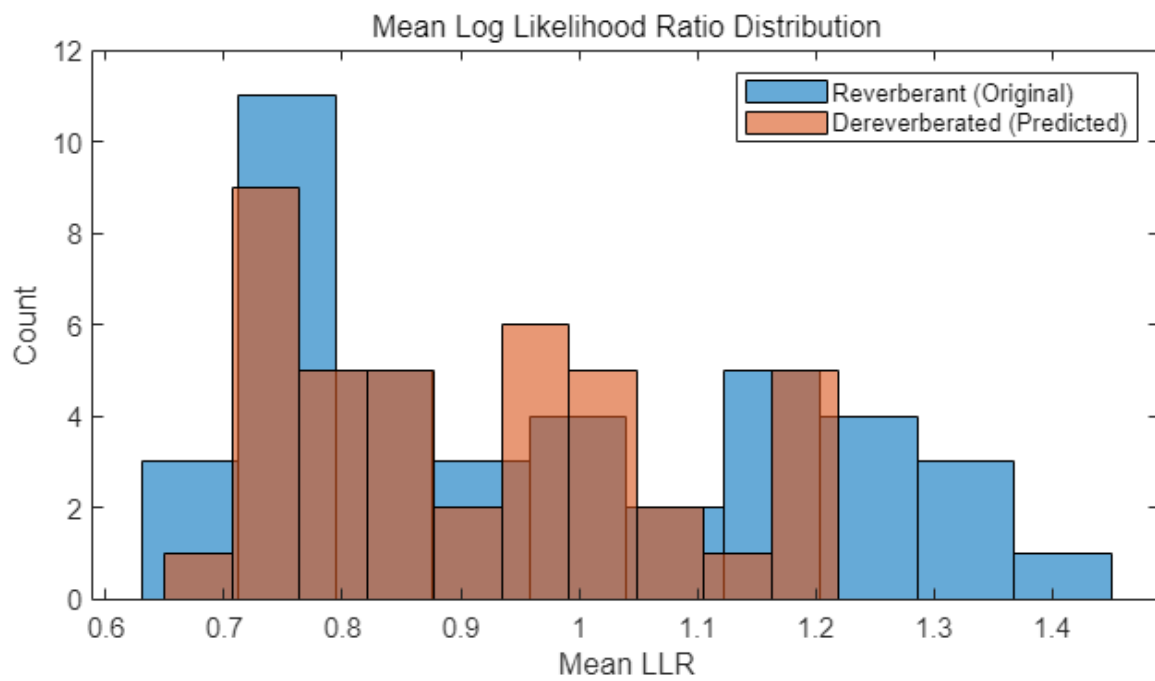
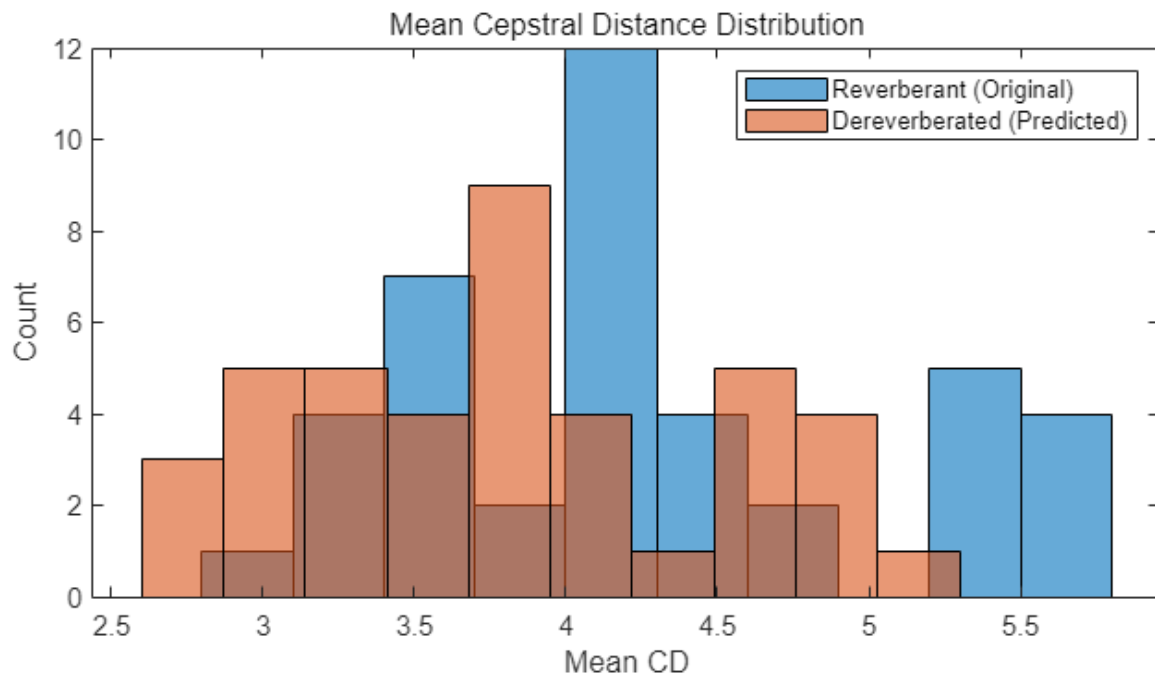
```
    avgCdMean: 4.2591
    avgCdMedian: 3.6336
    avgLlrMean: 0.9726
    avgLlrMedian: 0.8714
```

The histograms illustrate the distribution of mean CD, mean SRMR and mean LLR of the reverberant and dereverberated data.

```
figure(Position=[50,50,1100,1300])
tiledlayout(2,1)

nexttile
histogram(allMeasuresReverb.cdMean,10)
hold on
histogram(allMeasuresReconstructed.cdMean,10)
subtitle("Mean Cepstral Distance Distribution")
ylabel("Count")
xlabel("Mean CD")
legend("Reverberant (Original)","Dereverberated (Predicted)")

nexttile
histogram(allMeasuresReverb.llrMean,10)
hold on
histogram(allMeasuresReconstructed.llrMean,10)
subtitle("Mean Log Likelihood Ratio Distribution")
ylabel("Count")
xlabel("Mean LLR")
legend("Reverberant (Original)","Dereverberated (Predicted)")
```



Supporting Functions

Apply Reverberation

```
function yOut = applyReverb(y,preDelay,decayFactor,wetDryMix,fs)
% This function generates reverberant speech data using the reverberator
% object
%
% inputs:
% y - clean speech sample
% preDelay, decayFactor, wetDryMix - reverberation parameters
% fs - sampling rate of y
%
% outputs:
% yOut - corresponding reverberated speech sample

revObj = reverberator(SampleRate=fs, ...
    DecayFactor=decayFactor, ...
    WetDryMix=wetDryMix, ...
    PreDelay=preDelay);
yOut = revObj(y);
yOut = yOut(1:length(y),1);
end
```

Extract Features Batch

```
function [featuresClean,featuresReverb,phaseReverb,cleanAudios,verrbAudios] ...
    = helperFeatureExtract(cleanAudio,verrbAudio,isVal,params)
% This function performs the preprocessing and features extraction task on
% the audio files used for dereverberation model training and testing.
%
% inputs:
% cleanAudio - the clean audio file (reference)
% verrbAudio - corresponding reverberant speech file
% isVal - Boolean flag indicating if it is the validation set
% params - a structure containing feature extraction parameters
%
% outputs:
% featuresClean - log-magnitude STFT features of clean audio
% featuresReverb - log-magnitude STFT features of reverberant audio
% phaseReverb - phase of STFT of reverberant audio
% cleanAudios - 2.072s-segments of clean audio file used for feature extraction
% verrbAudios - 2.072s-segments of corresponding reverberant audio

assert(length(cleanAudio) == length(verrbAudio));
nSegments = floor((length(verrbAudio) - (params.samplesPerImage - params.shiftImage))/params.sh

featuresClean = {};
featuresReverb = {};
phaseReverb = {};
cleanAudios = {};
verrbAudios = {};
nGood = 0;
nonSilentRegions = detectSpeech(verrbAudio, params.fs);
nonSilentRegionIdx = 1;
totalRegions = size(nonSilentRegions, 1);
```

```
for cid = 1:nSegments
    start = (cid - 1)*params.shiftImage + 1;
    en = start + params.samplesPerImage - 1;

    nonSilentSamples = 0;
    while nonSilentRegionIdx < totalRegions && nonSilentRegions(nonSilentRegionIdx, 2) < start
        nonSilentRegionIdx = nonSilentRegionIdx + 1;
    end

    nonSilentStart = nonSilentRegionIdx;
    while nonSilentStart <= totalRegions && nonSilentRegions(nonSilentStart, 1) <= en
        nonSilentDuration = min(en, nonSilentRegions(nonSilentStart,2)) - max(start,nonSilentReg
        nonSilentSamples = nonSilentSamples + nonSilentDuration;
        nonSilentStart = nonSilentStart + 1;
    end

    nonSilentPerc = nonSilentSamples * 100 / (en - start + 1);
    silent = nonSilentPerc < 50;

    reverbAudioSegment = reverbAudio(start:en);
    if ~silent
        nGood = nGood + 1;
        cleanAudioSegment = cleanAudio(start:en);
        assert(length(cleanAudioSegment)==length(reverbAudioSegment),"Lengths do not match after

        % Clean Audio
        [featsUnit, ~] = featureExtract(cleanAudioSegment, params);
        featuresClean{nGood} = featsUnit; %#ok

        % Reverb Audio
        [featsUnit, phaseUnit] = featureExtract(reverbAudioSegment, params);
        featuresReverb{nGood} = featsUnit; %#ok
        if isVal
            phaseReverb{nGood} = phaseUnit; %#ok
            reverbAudios{nGood} = reverbAudioSegment;%#ok
            cleanAudios{nGood} = cleanAudioSegment;%#ok
        end
    end
end
end
```

Extract Features

```
function [features, phase, lastFBin] = featureExtract(audio, params)
% Function to extract features for a speech file
audio = single(audio);

audioSTFT = stft(audio,Window=params.Window,OverlapLength=params.OverlapLength, ...
    FFTLength=params.FFTLength,FrequencyRange="onesided");

phase = single(angle(audioSTFT(1:end-1,:)));
features = single(log(abs(audioSTFT(1:end-1,:)) + 10e-30));
lastFBin = audioSTFT(end,:);

end
```


Normalize and Reshape Features

```
function [featNorm,minMaxPairs] = featureNormalizeAndReshape(feats)
% function to normalize features - range [-1, 1] and reshape to 4
% dimensions
%
% inputs:
% feats - 3-dimensional array of extracted features
%
% outputs:
% featNorm - normalized and reshaped features
% minMaxPairs - array of original min and max pairs used for normalization

nSamples = length(feats);
minMaxPairs = zeros(nSamples,2,"single");
featNorm = zeros([size(feats{1}),nSamples],"single");
parfor i = 1:nSamples
    feat = feats{i};
    maxFeat = max(feat,[],"all");
    minFeat = min(feat,[],"all");
    featNorm(:,:,i) = 2.*(feat - minFeat)./(maxFeat - minFeat) - 1;
    minMaxPairs(i,:) = [minFeat,maxFeat];
end
featNorm = reshape(featNorm,size(featNorm,1),size(featNorm,2),1,size(featNorm,3));
end
```

Reconstruct Predicted Audio

```
function dereverbedAudioAll = helperReconstructPredictedAudios(predictedSTFT4D,minMaxPairs,reverse)
% This function will reconstruct the 2.072s long audios predicted by the
% model using the predicted log-magnitude spectrogram and the phase of the
% reverberant audio file
%
% inputs:
% predictedSTFT4D - Predicted 4-dimensional STFT log-magnitude features
% minMaxPairs     - Original minimum/maximum value pairs used in normalization
% reverbPhase     - Array of phases of STFT of reverberant audio files
% reverbAudios    - 2.072s-segments of corresponding reverberant audios
% params          - Structure containing feature extraction parameters

predictedSTFT = squeeze(predictedSTFT4D);
denormalizedFeatures = zeros(size(predictedSTFT),"single");
for ii = 1:size(predictedSTFT,3)
    feat = predictedSTFT(:,:,ii);
    maxFeat = minMaxPairs(ii,2);
    minFeat = minMaxPairs(ii,1);
    denormalizedFeatures(:,:,ii) = (feat + 1).*(maxFeat-minFeat)./2 + minFeat;
end

predictedSTFT = exp(denormalizedFeatures);

nCount = size(predictedSTFT,3);
dereverbedAudioAll = cell(1,nCount);
```

```

nSeg = params.NumSegments;
win = params.Window;
ovrlp = params.OverlapLength;
FFTLength = params.FFTLength;
parfor ii = 1:nCount
    % Append zeros to the highest frequency bin
    stftUnit = predictedSTFT(:, :, ii);
    stftUnit = cat(1, stftUnit, zeros(1, nSeg));
    phase = reverbPhase{ii};
    phase = cat(1, phase, zeros(1, nSeg));

    oneSidedSTFT = stftUnit.*exp(1j*phase);
    dereverbedAudio = istft(oneSidedSTFT, ...
        Window=win, OverlapLength=ovrlp, ...
        FFTLength=FFTLength, ConjugateSymmetric=true, ...
        FrequencyRange="onesided");

    dereverbedAudioAll{ii} = dereverbedAudio./max(max(abs(dereverbedAudio)), max(abs(reverbAudios)));
end
end

```

Calculate Objective Measures

```

function [summaryMeasures, allMeasures] = calculateObjectiveMeasures(reconstructedAudios, cleanAudios, fs)
% This function computes the objective measures on time-domain signals.
%
% inputs:
% reconstructedAudios - An array of audio files to evaluate.
% cleanAudios - An array of reference audio files
% fs - Sampling rate of audio files
%
% outputs:
% summaryMeasures - Global means of CD, LLR individual mean and median values
% allMeasures - Individual mean and median values

nAudios = length(reconstructedAudios);
cdMean = zeros(nAudios, 1);
cdMedian = zeros(nAudios, 1);
llrMean = zeros(nAudios, 1);
llrMedian = zeros(nAudios, 1);

parfor k = 1 : nAudios
    y = reconstructedAudios{k};
    x = cleanAudios{k};

    y = y./max(abs(y));
    x = x./max(abs(x));

    [cdMean(k), cdMedian(k)] = cepstralDistance(x, y, fs);
    [llrMean(k), llrMedian(k)] = lpcLogLikelihoodRatio(y, x, fs);
end

summaryMeasures.avgCdMean = mean(cdMean);
summaryMeasures.avgCdMedian = mean(cdMedian);
summaryMeasures.avgLlrMean = mean(llrMean);
summaryMeasures.avgLlrMedian = mean(llrMedian);

allMeasures.cdMean = cdMean;

```

```

    allMeasures.llrMean = llrMean;
end

```

Cepstral Distance

```

function [meanVal, medianVal] = cepstralDistance(x,y,fs)
    x = x/sqrt(sum(x.^2));
    y = y/sqrt(sum(y.^2));

    width = round(0.025*fs);
    shift = round(0.01*fs);

    nSamples = length(x);
    nFrames = floor((nSamples - width + shift)/shift);
    win = window(@hanning,width);

    winIndex = repmat((1:width)',1,nFrames) + repmat((0:nFrames - 1)*shift,width,1);

    xFrames = x(winIndex).*win;
    yFrames = y(winIndex).*win;

    xCeps = cepstralReal(xFrames,width);
    yCeps = cepstralReal(yFrames,width);

    dist = (xCeps - yCeps).^2;
    cepsD = 10/log(10)*sqrt(2*sum(dist(2:end,:),1) + dist(1,:));
    cepsD = max(min(cepsD,10),0);

    meanVal = mean(cepsD);
    medianVal = median(cepsD);
end

```

Real Cepstrum

```

function realC = cepstralReal(x,width)
    width2p = 2^nextpow2(width);
    powX = abs(fft(x,width2p));

    lowCutoff = max(powX(:))*10^-5;
    powX = max(powX,lowCutoff);

    realC = real(ifft(log(powX)));
    order = 24;
    realC = realC(1:order + 1,:);
    realC = realC - mean(realC,2);
end

```

LPC Log-Likelihood Ratio

```

function [meanLlr,medianLlr] = lpcLogLikelihoodRatio(x,y,fs)
    order = 12;
    width = round(0.025*fs);
    shift = round(0.01*fs);

    nSamples = length(x);
    nFrames = floor((nSamples - width + shift)/shift);
    win = window(@hanning,width);

    winIndex = repmat((1:width)',1,nFrames) + repmat((0:nFrames - 1)*shift,width,1);

```

```
xFrames = x(winIndex).*win;
yFrames = y(winIndex).*win;

lpcX = realLpc(xFrames,width,order);
[lpcY,realY] = realLpc(yFrames,width,order);

llr = zeros(nFrames,1);
for n = 1:nFrames
    R = toeplitz(realY(1:order+1,n));
    num = lpcX(:,n)'*R*lpcX(:,n);
    den = lpcY(:,n)'*R*lpcY(:,n);
    llr(n) = log(num/den);
end

llr = sort(llr);
llr = llr(1:ceil(nFrames*0.95));
llr = max(min(llr,2),0);

meanLlr = mean(llr);
medianLlr = median(llr);
end
```

Real Linear Precition Coefficients

```
function [lpcCoeffs, realX] = realLpc(xFrames,width,order)
width2p = 2^nextpow2(width);
X = fft(xFrames,width2p);

Rx = ifft(abs(X).^2);
Rx = Rx./width;
realX = real(Rx);

lpcX = levinson(realX,order);
lpcCoeffs = real(lpcX');
end
```

References

- [1] Ernst, O., Chazan, S.E., Gannot, S., & Goldberger, J. (2018). Speech Dereverberation Using Fully Convolutional Networks. *2018 26th European Signal Processing Conference (EUSIPCO)*, 390-394.
- [2] <https://datashare.is.ed.ac.uk/handle/10283/2031>
- [3] <https://datashare.is.ed.ac.uk/handle/10283/2791>
- [4] <https://github.com/MuSAELab/SRMRTtoolbox>

Speaker Identification Using Custom SincNet Layer and Deep Learning

In this example, you train three convolutional neural networks (CNNs) to perform speaker verification and then compare the performances of the architectures. The architectures of the three CNNs are all equivalent except for the first convolutional layer in each:

- 1 In the first architecture, the first convolutional layer is a "standard" convolutional layer, implemented using `convolution2dLayer`.
- 2 In the second architecture, the first convolutional layer is a constant sinc filterbank, implemented using a custom layer.
- 3 In the third architecture, the first convolutional layer is a trainable sinc filterbank, implemented using a custom layer. This architecture is referred to as *SincNet* [1] on page 1-736.

[1] on page 1-736 shows that replacing the standard convolutional layer with a filterbank layer leads to faster training convergence and higher accuracy. [1] on page 1-736 also shows that making the parameters of the filter bank learnable yields additional performance gains.

Introduction

Speaker identification is a prominent research area with a variety of applications including forensics and biometric authentication. Many speaker identification systems depend on precomputed features such as i-vectors or MFCCs, which are then fed into machine learning or deep learning networks for classification. Other deep learning speech systems bypass the feature extraction stage and feed the audio signal directly to the network. In such end-to-end systems, the network directly learns low-level audio signal characteristics.

In this example, you first train a traditional end-to-end speaker identification CNN. The filters learned tend to have random shapes that do not correspond to perceptual evidence or knowledge of how the human ear works, especially in scenarios where the amount of training data is limited [1] on page 1-736. You then replace the first convolutional layer in the network with a custom sinc filterbank layer that introduces structure and constraints based on perceptual evidence. Finally, you train the SincNet architecture, which adds learnability to the sinc filterbank parameters.

The three neural network architectures explored in the example are summarized as follows:

- 1 **Standard Convolutional Neural Network** - The input waveform is directly connected to a randomly initialized convolutional layer which attempts to learn features and capture characteristics from the raw audio frames.
- 2 **ConstantSincLayer** - The input waveform is convolved with a set of fixed-width sinc functions (bandpass filters) equally spaced on the mel scale.
- 3 **SincNetLayer** - The input waveform is convolved with a set of sinc functions whose parameters are learned by the network. In the SincNet architecture, the network tunes parameters of the sinc functions while training.

This example defines and trains the three neural networks proposed above and evaluates their performance on the LibriSpeech Dataset [2] on page 1-736.

Data Set

Download Dataset

In this example, you use a subset of the LibriSpeech Dataset [2] on page 1-736. The LibriSpeech Dataset is a large corpus of read English speech sampled at 16 kHz. The data is derived from audiobooks read from the LibriVox project.

```
dataFolder = tempdir;  
  
dataset = fullfile(dataFolder,"LibriSpeech","train-clean-100");  
if ~datasetExists(dataset)  
    filename = "train-clean-100.tar.gz";  
    url = "http://www.openslr.org/resources/12/" + filename;  
    gunzip(url,dataFolder);  
    unzippedFile = fullfile(dataset,filename);  
    untar(unzippedFile{1}(1:end-3),dataset);  
end
```

Create an `audioDatastore` object to access the LibriSpeech audio data.

```
ads = audioDatastore(dataset,IncludeSubfolders=true);
```

Extract the speaker label from the file path.

```
ads.Labels = categorical(extractBetween(ads.Files,fullfile(dataset,filesep),filesep));
```

The full `dev-train-100` dataset is around 6 GB of data. To run this example quickly, set `speedupExample` to `true`.

```
speedupExample = ;  
if speedupExample  
    allSpeakers = unique(ads.Labels);  
    subsetSpeakers = allSpeakers(1:50);  
    ads = subset(ads,ismember(ads.Labels,subsetSpeakers));  
    ads.Labels = removecats(ads.Labels);  
end  
ads = splitEachLabel(ads,0.1);
```

Split the audio files into training and test data. 80% of the audio files are assigned to the training set and 20% are assigned to the test set.

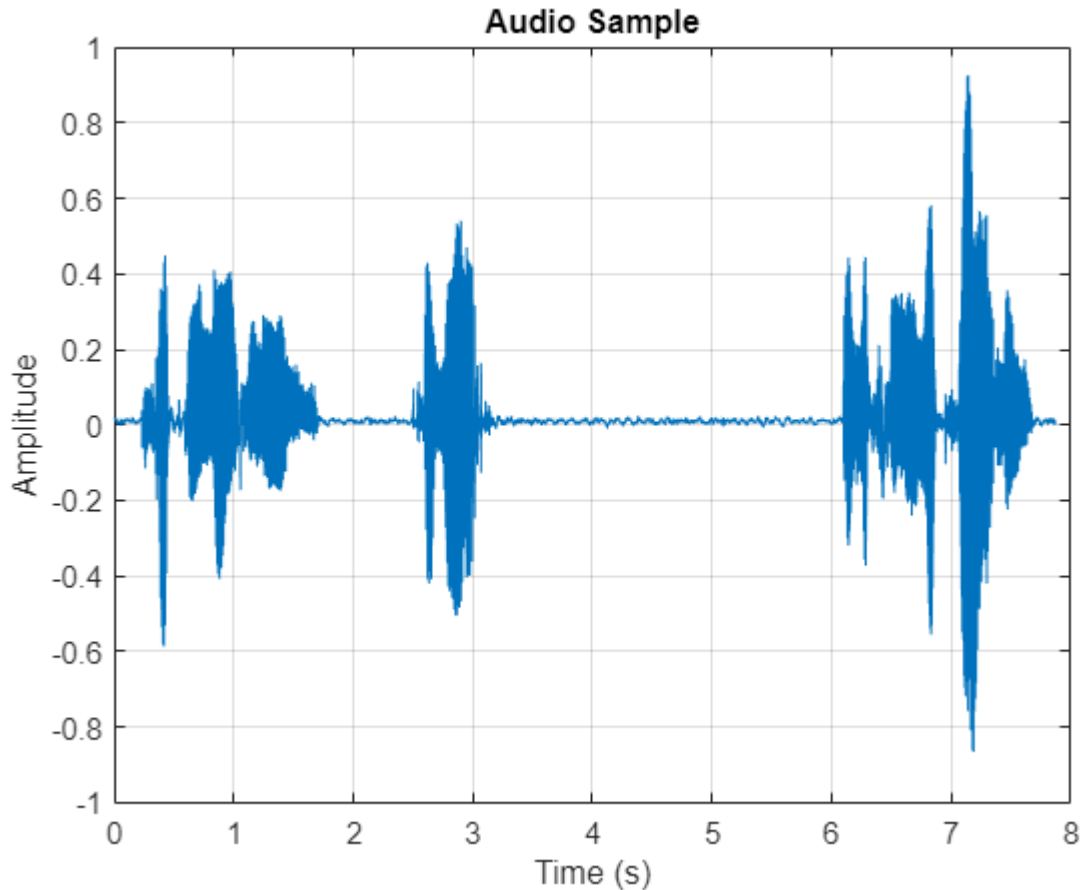
```
[adsTrain,adsTest] = splitEachLabel(ads,0.8);
```

Sample Speech Signal

Plot one of the audio files and listen to it.

```
[audioIn,dsInfo] = read(adsTrain);  
Fs = dsInfo.SampleRate;  
  
sound(audioIn,Fs)  
  
t = (1/Fs)*(0:length(audioIn)-1);  
  
plot(t,audioIn)  
title("Audio Sample")
```

```
xlabel("Time (s)")
ylabel("Amplitude")
grid on
```



Reset the training datastore.

```
reset(adsTrain)
```

Data Preprocessing

CNNs expect inputs to have consistent dimensions. You will preprocess the audio by removing regions of silence and then break the remaining speech into 200 ms frames with 40 ms overlap.

Set the parameters for preprocessing.

```
frameDuration = 200e-3;
overlapDuration = 40e-3;
frameLength = floor(Fs*frameDuration);
overlapLength = round(Fs*overlapDuration);
```

Use the supporting function, `preprocessAudioData` on page 1-736, to preprocess the training and test data. Define a transform on the audio datastores to perform the preprocessing, then use `readall` to preprocess the entire datasets and place the preprocessed data into memory. If you have Parallel Computing Toolbox™, you can spread the computational load across workers. `XTrain` and `XTest`

contain the train and test speech frames, respectively. TTrain and TTest contain the train and test labels, respectively.

```
pFlag = ~isempty(ver("parallel"));
```

```
adsTrainTransform = transform(adsTrain,@(x){preprocessAudioData(x,frameLength,overlapLength,Fs)});  
XTrain = readall(adsTrainTransform,UseParallel=pFlag);
```

Replicate the labels so that each 200 ms chunk has a corresponding label.

```
chunksPerFile = cellfun(@(x)size(x,4),XTrain);  
TTrain = repelem(adsTrain.Labels,chunksPerFile,1);
```

Concatenate the training set into an array.

```
XTrain = cat(4,XTrain{:});
```

Perform the same preprocessing steps to the test set.

```
adsTestTransform = transform(adsTest,@(x){preprocessAudioData(x,frameLength,overlapLength,Fs)});  
XTest = readall(adsTestTransform,UseParallel=true);  
chunksPerFile = cellfun(@(x)size(x,4),XTest);  
TTest = repelem(adsTest.Labels,chunksPerFile,1);  
XTest = cat(4,XTest{:});
```

Standard CNN

Define Layers

The standard CNN is inspired by the neural network architecture in [1] on page 1-736.

```
numFilters = 80;  
filterLength = 251;  
numSpeakers = numel(unique(removecats(ads.Labels)));
```

```
layers = [  
    imageInputLayer([1 frameLength 1])  
  
    % First convolutional layer  
  
    convolution2dLayer([1 filterLength],numFilters)  
    batchNormalizationLayer  
    leakyReluLayer(0.2)  
    maxPooling2dLayer([1 3])  
  
    % This layer is followed by 2 convolutional layers  
  
    convolution2dLayer([1 5],60)  
    batchNormalizationLayer  
    leakyReluLayer(0.2)  
    maxPooling2dLayer([1 3])  
  
    convolution2dLayer([1 5],60)  
    batchNormalizationLayer  
    leakyReluLayer(0.2)  
    maxPooling2dLayer([1 3])  
  
    % This is followed by 3 fully-connected layers
```



```

fullyConnectedLayer(256)
batchNormalizationLayer
leakyReluLayer(0.2)

fullyConnectedLayer(256)
batchNormalizationLayer
leakyReluLayer(0.2)

fullyConnectedLayer(256)
batchNormalizationLayer
leakyReluLayer(0.2)

fullyConnectedLayer(numSpeakers)
softmaxLayer
classificationLayer];

```

Analyze the layers of the neural network using the `analyzeNetwork` function

```
analyzeNetwork(layers)
```

Train Network

Train the neural network for 15 epochs using `adam` optimization. Shuffle the training data before every epoch. The training options for the neural network are set using `trainingOptions`. Use the test data as the validation data to observe how the network performance improves as training progresses.

```

numEpochs = 15;
miniBatchSize = 128;
validationFrequency = floor(numel(TTrain)/miniBatchSize);

options = trainingOptions("adam", ...
    Shuffle="every-epoch", ...
    MiniBatchSize=miniBatchSize, ...
    Plots="training-progress", ...
    Verbose=false,MaxEpochs=numEpochs, ...
    ValidationData={XTest,categorical(TTest)}, ...
    ValidationFrequency=validationFrequency);

```

To train the network, call `trainNetwork`.

```
[convNet,convNetInfo] = trainNetwork(XTrain,TTrain,layers,options);
```



Inspect Frequency Response of First Convolutional Layer

Plot the magnitude frequency response of nine filters learned from the standard CNN network. The shape of these filters is not intuitive and does not correspond to perceptual knowledge. The next section explores the effect of using constrained filter shapes.

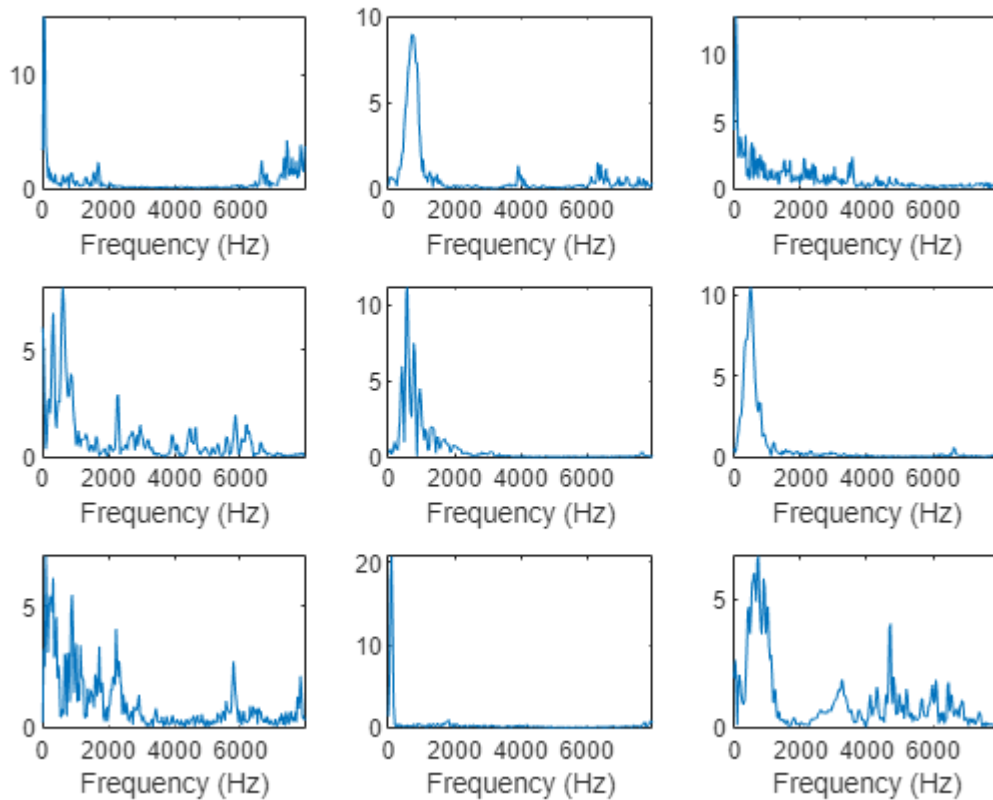
```
F = squeeze(convNet.Layers(2,1).Weights);
H = zeros(size(F));
Freq = zeros(size(F));

for ii = 1:size(F,2)
    [h,f] = freqz(F(:,ii),1,251,Fs);
    H(:,ii) = abs(h);
    Freq(:,ii) = f;
end

idx = linspace(1,size(F,2),9);
idx = round(idx);

figure
for jj = 1:9
    subplot(3,3,jj)
    plot(Freq(:,idx(jj)),H(:,idx(jj)))
    sgtitle("Frequency Response of Learned Standard CNN Filters")
    xlabel("Frequency (Hz)")
end
```

Frequency Response of Learned Standard CNN Filters



Constant Sinc Filterbank

In this section, you replace the first convolutional layer in the standard CNN with a constant sinc filterbank layer. The constant sinc filterbank layer convolves the input frames with a bank of fixed bandpass filters. The bandpass filters are a linear combination of two sinc filters in the time domain. The frequencies of the bandpass filters are spaced linearly on the mel scale.

Define Layers

The implementation for the constant sinc filterbank layer can be found in the `constantSincLayer.m` file (attached to this example). Define parameters for a `ConstantSincLayer`. Use 80 filters and a filter length of 251.

```
numFilters = 80;
filterLength = 251;
numChannels = 1;
name = "constant_sinc";
```

Change the first convolutional layer from the standard CNN to the `ConstantSincLayer` and keep the other layers unchanged.

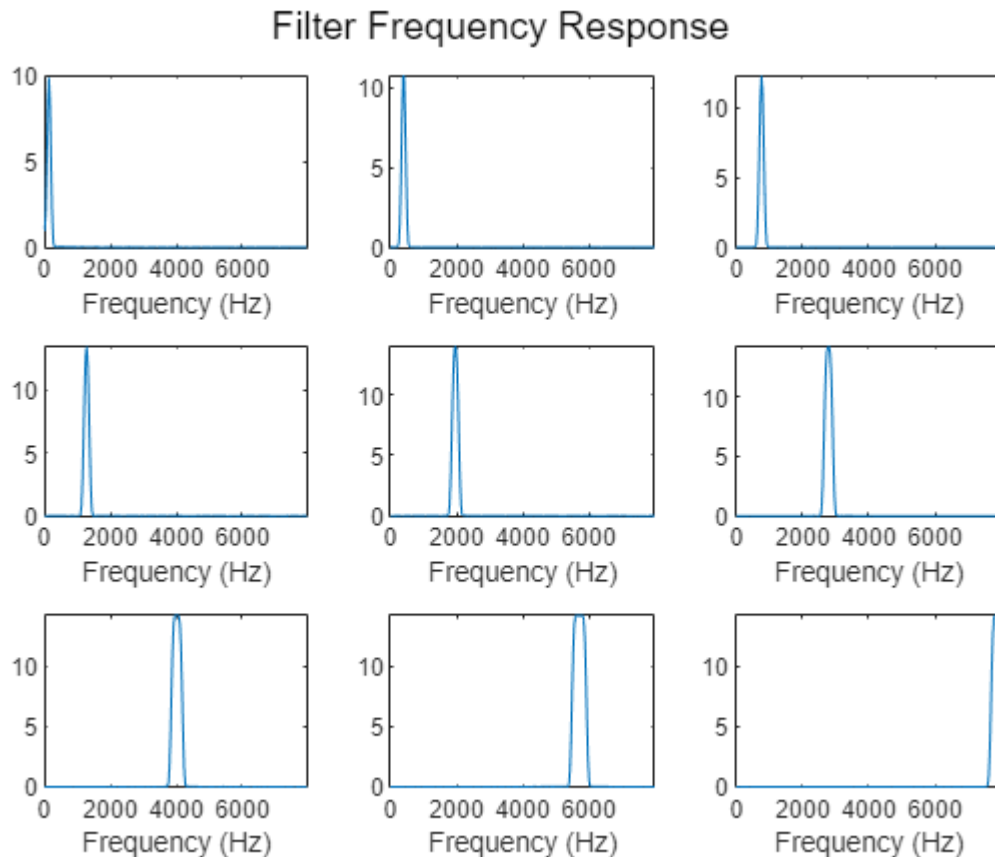
```
cSL = constantSincLayer(numFilters,filterLength,Fs,numChannels,name)
```

```
cSL =
    constantSincLayer with properties:
```


Inspect Frequency Response of First Convolutional Layer

The `plotNFilters` method plots the magnitude frequency response of `n` filters with equally spaced filter indices. Plot the magnitude frequency response of nine filters in the `ConstantSincLayer`.

```
figure
n = 9;
plotNFilters(constSincNet.Layers(2),n)
```



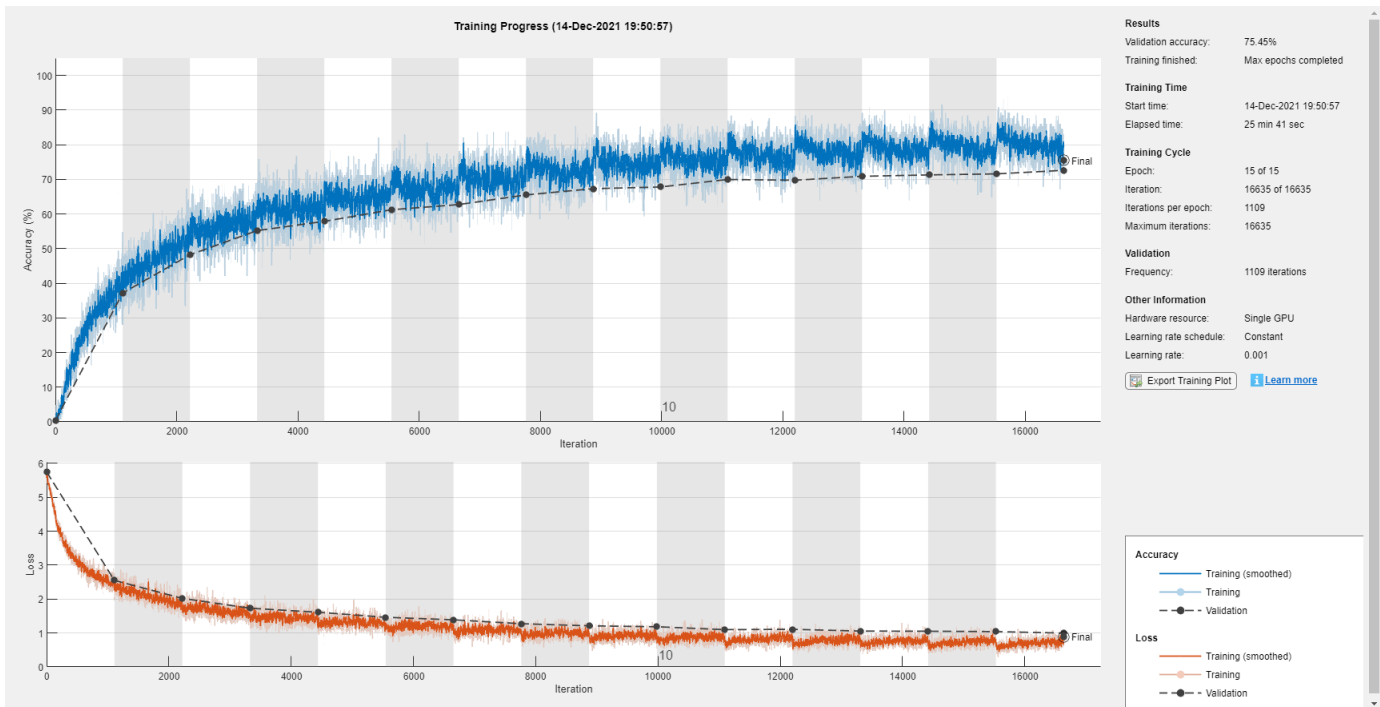
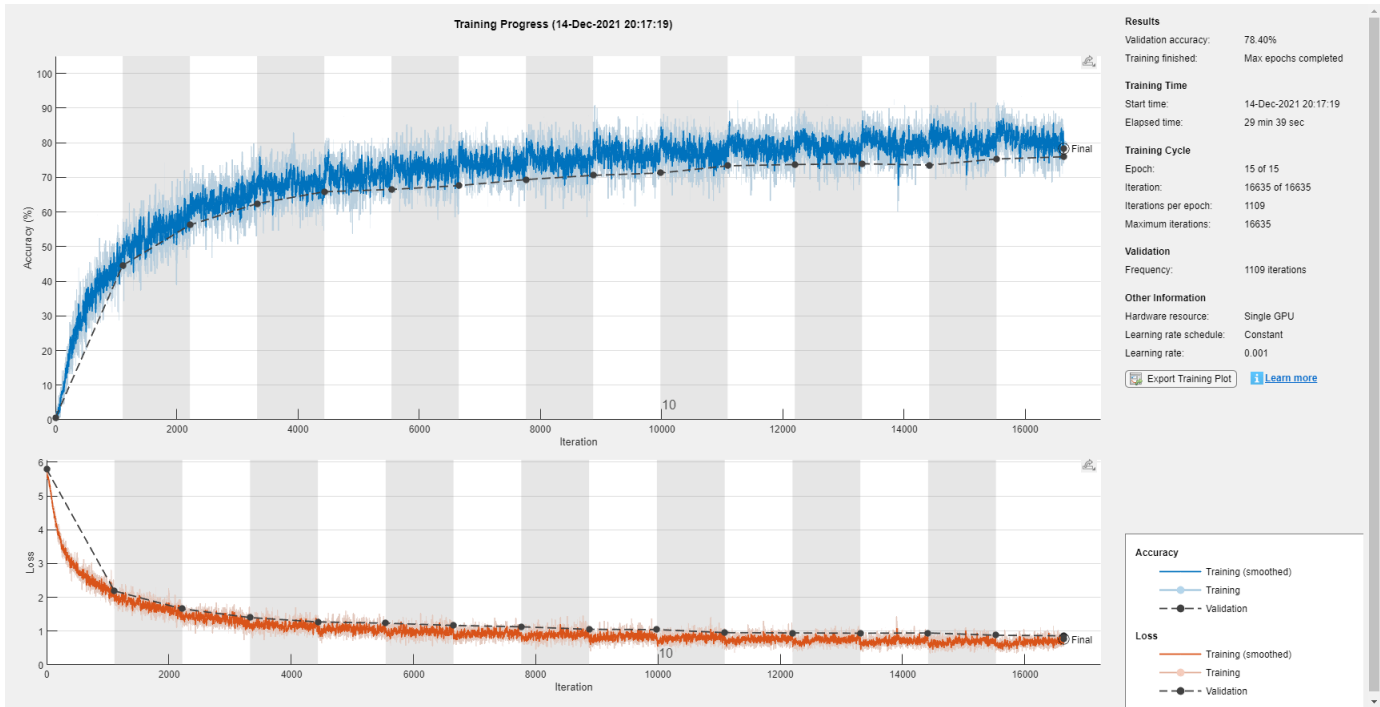
SincNet

In this section, you use a trainable SincNet layer as the first convolutional layer in your network. The SincNet layer convolves the input frames with a bank of bandpass filters. The bandwidth and the initial frequencies of the SincNet filters are initialized as equally spaced in the mel scale. The SincNet layer attempts to learn better parameters for these bandpass filters within the neural network framework.

Define Layers

The implementation for the SincNet layer filterbank layer can be found in the `sincNetLayer.m` file (attached to this example). Define parameters for a `SincNetLayer`. Use 80 filters and a filter length of 251.

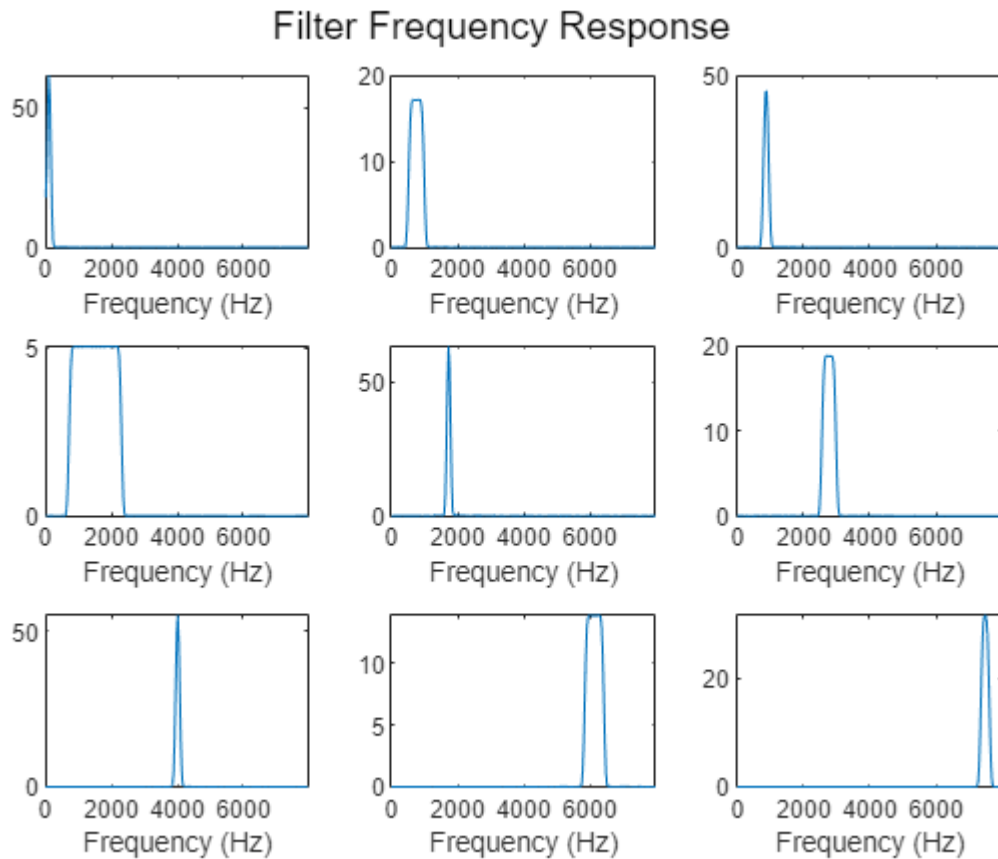
```
numFilters = 80;
filterLength = 251;
```

Inspect Frequency Response of First Convolutional Layer

Use the `plotNFilters` method of `SincNetLayer` to visualize the magnitude frequency response of nine filters with equally spaced indices learned by SincNet.

figure
`plotNFilters(sincNet.Layers(2),9)`



Results Summary

Accuracy

The table summarizes the frame accuracy for all three neural networks.

```
NetworkType = ["Standard CNN"; "Constant Sinc Layer"; "SincNet Layer"];
Accuracy = [convNetInfo.FinalValidationAccuracy; constSincInfo.FinalValidationAccuracy; sincNetInfo.FinalValidationAccuracy];
```

```
resultsSummary = table(NetworkType, Accuracy)
```

```
resultsSummary=3x2 table
    NetworkType      Accuracy
    _____      _____
    "Standard CNN"    71.202
    "Constant Sinc Layer" 75.455
    "SincNet Layer"   78.395
```

Performance with Respect to Epochs

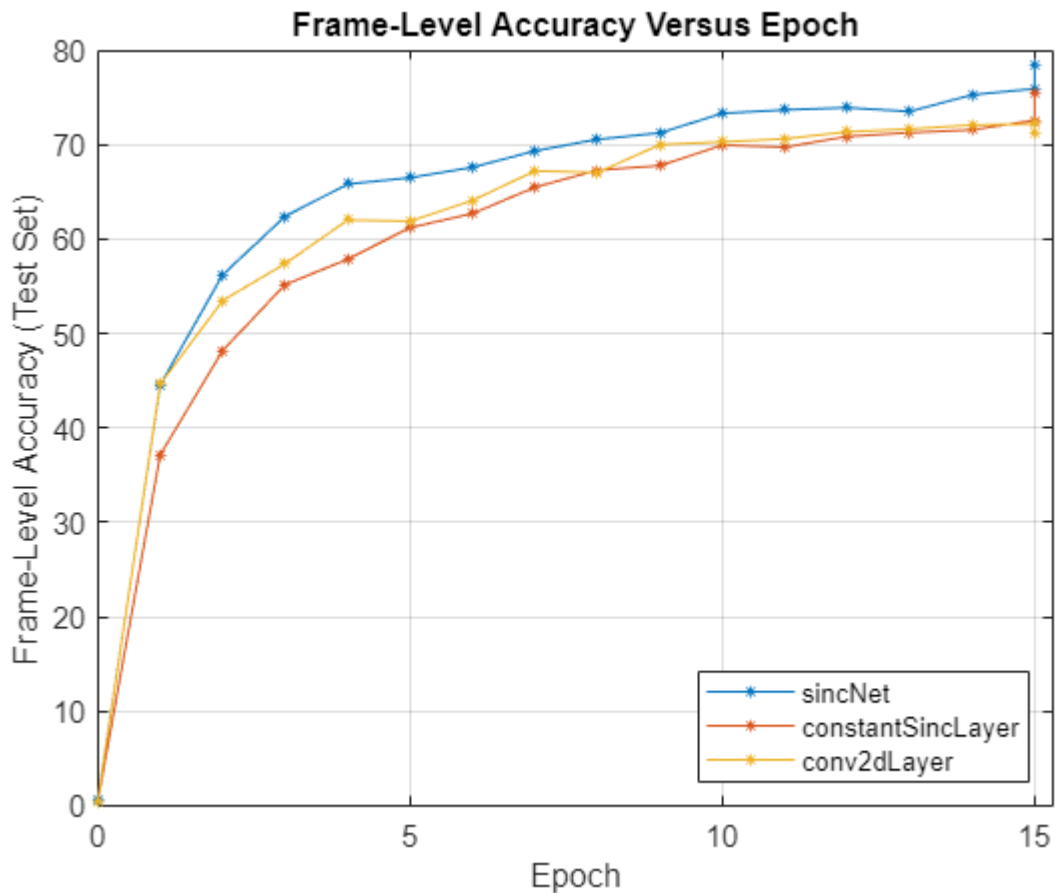
Plot the accuracy on the test set against the epoch number to see how well the networks learn as the number of epochs increase. SincNet outperforms the ConstantSincLayer network, especially during the early stages of training. This shows that updating the parameters of the bandpass filters

within the neural network framework leads to faster convergence. This behavior is only observed when the dataset is large enough, so it might not be seen when `speedupExample` is set to `true`.

```
epoch = linspace(0,numEpochs,numel(sincNetInfo.ValidationAccuracy(~isnan(sincNetInfo.ValidationAccuracy))),...
epoch = [epoch,numEpochs];
```

```
sinc_valAcc = [sincNetInfo.ValidationAccuracy(~isnan(sincNetInfo.ValidationAccuracy)),...
sincNetInfo.FinalValidationAccuracy];
const_sinc_valAcc = [constSincInfo.ValidationAccuracy(~isnan(constSincInfo.ValidationAccuracy)),...
constSincInfo.FinalValidationAccuracy];
conv_valAcc = [convNetInfo.ValidationAccuracy(~isnan(convNetInfo.ValidationAccuracy)),...
convNetInfo.FinalValidationAccuracy];
```

```
figure
plot(epoch,sinc_valAcc,"-*",MarkerSize=4)
hold on
plot(epoch,const_sinc_valAcc,"-*",MarkerSize=4)
plot(epoch,conv_valAcc,"-*",MarkerSize=4)
ylabel("Frame-Level Accuracy (Test Set)")
xlabel("Epoch")
xlim([0 numEpochs+0.3])
title("Frame-Level Accuracy Versus Epoch")
legend("sincNet","constantSincLayer","conv2dLayer",Location="southeast")
grid on
```



In the figure above, the final frame accuracy is a bit different from the frame accuracy that is computed in the last iteration. While training, the batch normalization layers perform normalization over mini-batches. However, at the end of training, the batch normalization layers normalize over the entire training data, which results in a slight change in performance.

Supporting Functions

```
function xp = preprocessAudioData(x, frameLength, overlapLength, Fs)

speechIdx = detectSpeech(x, Fs);
xp = zeros(1, frameLength, 1, 0);

for ii = 1:size(speechIdx, 1)
    % Isolate speech segment
    audioChunk = x(speechIdx(ii, 1):speechIdx(ii, 2));

    % Split into 200 ms chunks
    audioChunk = buffer(audioChunk, frameLength, overlapLength);
    audioChunk = reshape(audioChunk, 1, frameLength, 1, size(audioChunk, 2));

    % Concatenate with existing audio
    xp = cat(4, xp, audioChunk);
end
end
```

References

- [1] M. Ravanelli and Y. Bengio, "Speaker Recognition from Raw Waveform with SincNet," *2018 IEEE Spoken Language Technology Workshop (SLT)*, Athens, Greece, 2018, pp. 1021-1028, doi: 10.1109/SLT.2018.8639585.
- [2] V. Panayotov, G. Chen, D. Povey and S. Khudanpur, "Librispeech: An ASR corpus based on public domain audio books," *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Brisbane, QLD, 2015, pp. 5206-5210, doi: 10.1109/ICASSP.2015.7178964

Acoustics-Based Machine Fault Recognition

In this example, you develop a deep learning model to detect faults in an air compressor using acoustic measurements. After developing the model, you package the system so that you can recognize faults based on streaming input data.

Data Preparation

Download and unzip the air compressor data set [1] on page 1-756. This data set consists of recordings from air compressors in a healthy state or one of seven faulty states.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","AirCompressorDataset/AirC
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
dataset = fullfile(dataFolder,"AirCompressorDataset");
```

Create an `audioDatastore` object to manage the data and split it into training and validation sets. Call `countEachLabel` to inspect the distribution of labels in the train and validation sets.

```
ads = audioDatastore(dataset,IncludeSubfolders=true,LabelSource="foldernames");
```

```
[adsTrain,adsValidation] = splitEachLabel(ads,0.9,0.1);
```

```
countEachLabel(adsTrain)
```

```
ans=8x2 table
      Label      Count
      -----
      Bearing    203
      Flywheel   203
      Healthy    203
      LIV        203
      LOV        203
      NRV        203
      Piston     203
      Riderbelt  203
```

```
countEachLabel(adsValidation)
```

```
ans=8x2 table
      Label      Count
      -----
      Bearing    22
      Flywheel   22
      Healthy    22
      LIV        22
      LOV        22
      NRV        22
      Piston     22
      Riderbelt  22
```

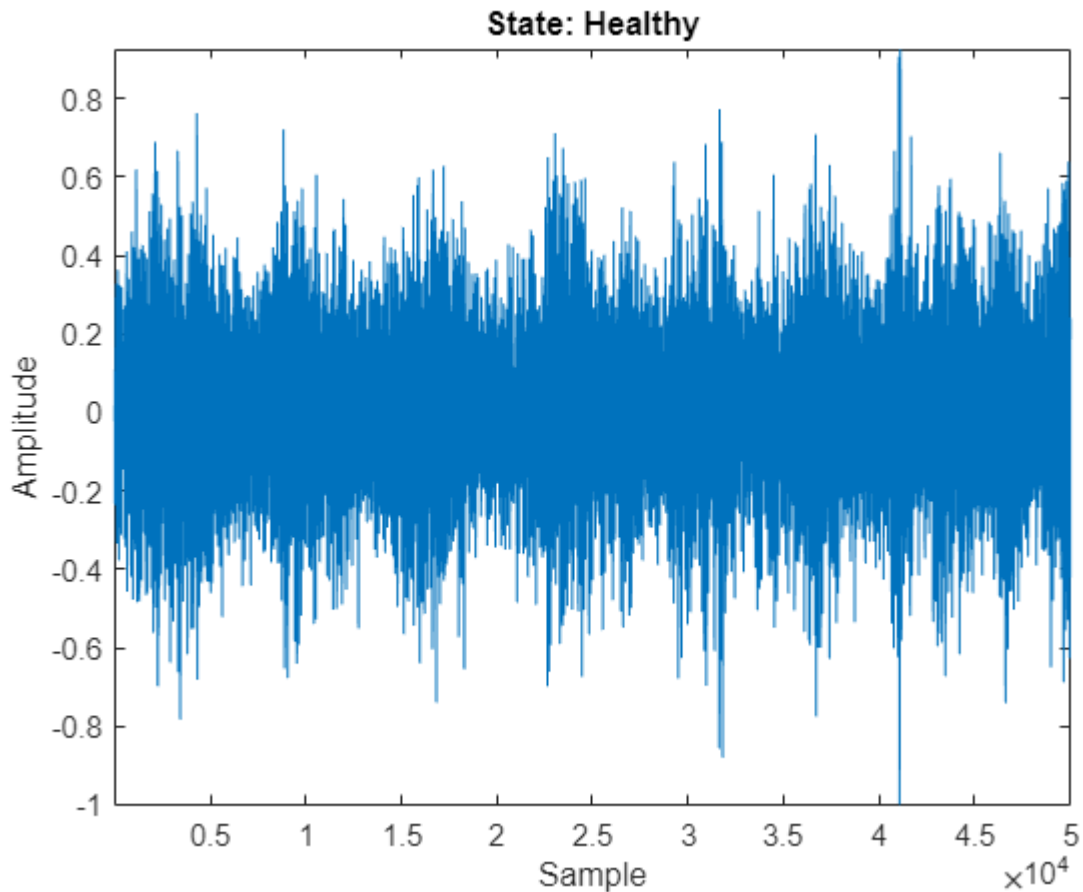
```
adsTrain = shuffle(adsTrain);
adsValidation = shuffle(adsValidation);
```

You can reduce the training data set 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 =  false ;  
if speedupExample  
    adsTrain = splitEachLabel(adsTrain,20);  
end
```

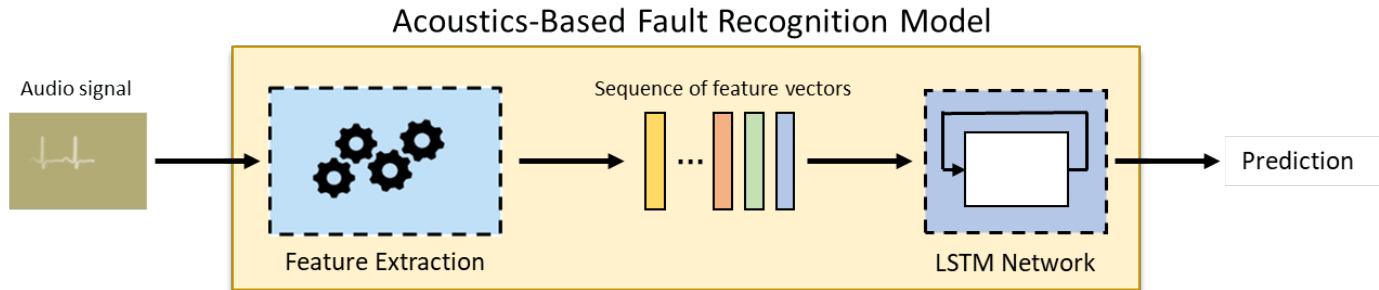
The data consists of time-series recordings of acoustics from faulty or healthy air compressors. As such, there are strong relationships between samples in time. Listen to a recording and plot the waveform.

```
[sampleData,sampleDataInfo] = read(adsTrain);  
fs = sampleDataInfo.SampleRate;  
  
soundsc(sampleData,fs)  
plot(sampleData)  
xlabel("Sample")  
ylabel("Amplitude")  
title("State: " + string(sampleDataInfo.Label))  
axis tight
```



Because the samples are related in time, you can use a recurrent neural network (RNN) to model the data. A long short-term memory (LSTM) network is a popular choice of RNN because it is designed to

avoid vanishing and exploding gradients. Before you can train the network, it's important to prepare the data adequately. Often, it is best to transform or extract features from 1-dimensional signal data in order to provide a richer set of features for the model to learn from.



Feature Engineering

The next step is to extract a set of acoustic features used as inputs to the network. Audio Toolbox™ enables you to extract spectral descriptors that are commonly used as inputs in machine learning tasks. You can extract the features using individual functions, or you can use `audioFeatureExtractor` to simplify the workflow and do it all at once.

```

trainFeatures = cell(1,numel(adsTrain.Files));
windowLength = 512;
overlapLength = 0;
  
```

```

aFE = audioFeatureExtractor(SampleRate=fs, ...
    Window=hamming(windowLength,"periodic"),...
    OverlapLength=overlapLength,...
    spectralCentroid=true, ...
    spectralCrest=true, ...
    spectralDecrease=true, ...
    spectralEntropy=true, ...
    spectralFlatness=true, ...
    spectralFlux=false, ...
    spectralKurtosis=true, ...
    spectralRolloffPoint=true, ...
    spectralSkewness=true, ...
    spectralSlope=true, ...
    spectralSpread=true);
  
```

```

reset(adsTrain)
tic
for index = 1:numel(adsTrain.Files)
    data = read(adsTrain);
    trainFeatures{index} = (extract(aFE,data))';
end
disp("Feature extraction of train set took " + toc + " seconds.");
  
```

```

Feature extraction of train set took 15.7192 seconds.
  
```

Data Augmentation

The training set contains a relatively small number of acoustic recordings for training a deep learning model. A popular method to enlarge the dataset is to use mixup. In mixup, you augment your dataset by mixing the features and labels from two different class instances. Mixup was reformulated by [2]

on page 1-756 as labels drawn from a probability distribution instead of mixed labels. The supporting function, `mixup` on page 1-755, takes the training features, associated labels, and the number of mixes per observation and then outputs the mixes and associated labels.

```
trainLabels = adsTrain.Labels;
numMixesPerInstance = 2 ;
tic
[augData,augLabels] = mixup(trainFeatures,trainLabels,numMixesPerInstance);
```

```
trainLabels = cat(1,trainLabels,augLabels);
trainFeatures = cat(2,trainFeatures,augData);
disp("Feature augmentation of train set took " + toc + " seconds.");
```

Feature augmentation of train set took 0.16065 seconds.

Generate Validation Features

Repeat the feature extraction for the validation features.

```
validationFeatures = cell(1,numel(adsValidation.Files));
reset(adsValidation)
tic
for index = 1:numel(adsValidation.Files)
    data = read(adsValidation);
    validationFeatures{index} = (extract(aFE,data))';
end
disp("Feature extraction of validation set took " + toc + " seconds.");
```

Feature extraction of validation set took 1.6419 seconds.

Train Model

Next, you define and train a network. To skip training the network, set `downloadPretrainedSystem` to `true`, then continue to the next section on page 1-742.

```
downloadPretrainedSystem = false ;
if downloadPretrainedSystem
    downloadFolder = matlab.internal.examples.downloadSupportFile("audio","AcousticsBasedMachineFaultRecognition");
    dataFolder = tempdir;
    unzip(downloadFolder,dataFolder)
    netFolder = fullfile(dataFolder,"AcousticsBasedMachineFaultRecognition");

    addpath(netFolder)
end
```

Define Network

An LSTM layer learns long-term dependencies between time steps of time series or sequence data. The first `lstmLayer` has 100 hidden units and outputs sequence data. Then a dropout layer is used to reduce overfitting. The second `lstmLayer` outputs the last step of the time sequence.

```
numHiddenUnits = 100 ;
dropProb = 0.2 ;
layers = [ ...
```

```

sequenceInputLayer(aFE.FeatureVectorLength,Normalization="zscore")
lstmLayer(numHiddenUnits,OutputMode="sequence")
dropoutLayer(dropProb)
lstmLayer(numHiddenUnits,OutputMode="last")
fullyConnectedLayer(numel(unique(adsTrain.Labels)))
softmaxLayer
classificationLayer];

```

Define Network Hyperparameters

To define hyperparameters for the network, use `trainingOptions`.

```

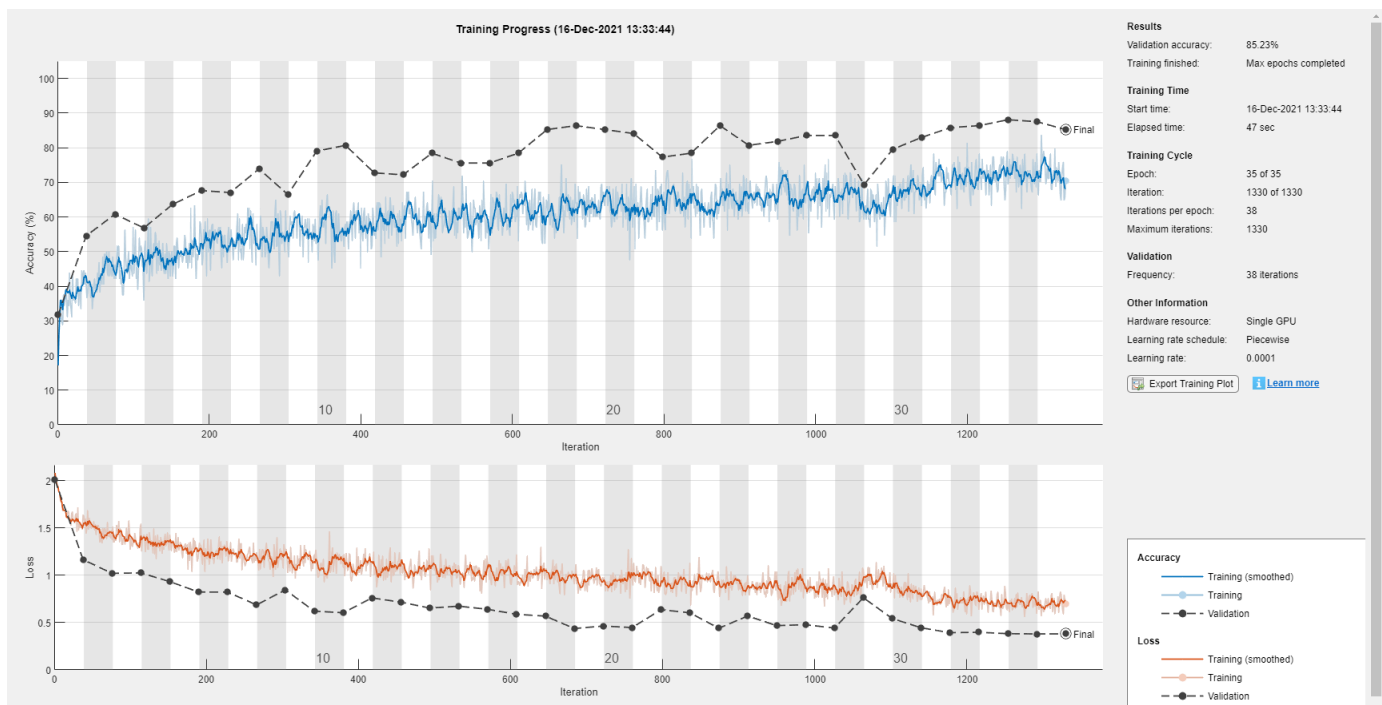
miniBatchSize = 128 ;
validationFrequency = floor(numel(trainFeatures)/miniBatchSize);
options = trainingOptions("adam", ...
    MiniBatchSize=miniBatchSize, ...
    MaxEpochs=35, ...
    Plots="training-progress", ...
    Verbose=false, ...
    Shuffle="every-epoch", ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropPeriod=30, ...
    LearnRateDropFactor=0.1, ...
    ValidationData={validationFeatures,adsValidation.Labels}, ...
    ValidationFrequency=validationFrequency);

```

Train Network

To train the network, use `trainNetwork`.

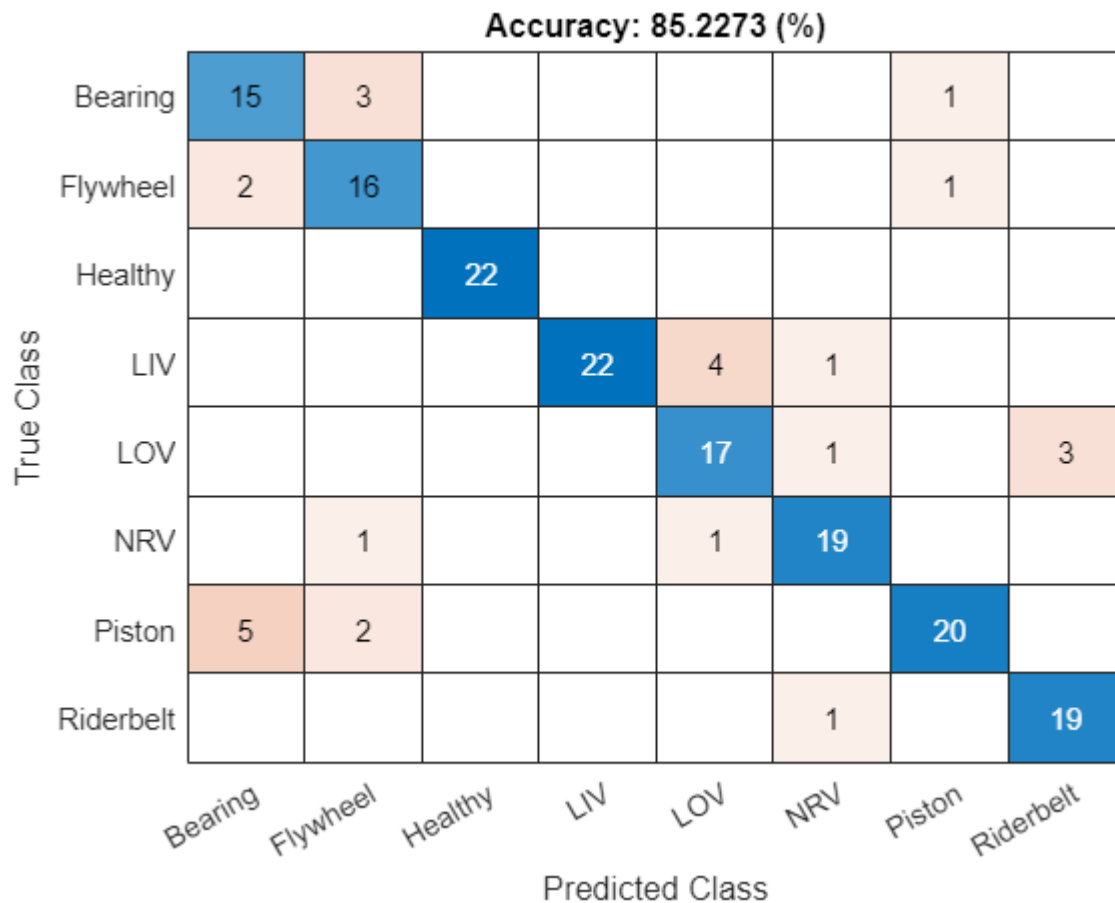
```
airCompNet = trainNetwork(trainFeatures,trainLabels,layers,options);
```



Evaluate Network

View the confusion chart for the validation data.

```
validationResults = classify(airCompNet,validationFeatures);
confusionchart(validationResults,adsValidation.Labels, ...
    Title="Accuracy: " + mean(validationResults == adsValidation.Labels)*100 + " (%)");
```



Model Streaming Detection

Create Functions to Process Data in a Streaming Loop

Once you have a trained network with satisfactory performance, you can apply the network to test data in a streaming fashion.

There are many additional considerations to take into account to make the system work in a real-world embedded system.

For example,

- The rate or interval at which classification can be performed with accurate results
- The size of the network in terms of generated code (program memory) and weights (data memory)
- The efficiency of the network in terms of computation speed

In MATLAB, you can mimic how the network is deployed and used in hardware on a real embedded system and begin to answer these important questions.

Create MATLAB Function Compatible with C/C++ Code Generation

Once you train your deep learning model, you will deploy it to an embedded target. That means you also need to deploy the code used to perform the feature extraction. Use the `generateMATLABFunction` method of `audioFeatureExtractor` to generate a MATLAB function compatible with C/C++ code generation. Specify `IsStreaming` as `true` so that the generated function is optimized for stream processing.

```
filename = fullfile(pwd, "extractAudioFeatures");
generateMATLABFunction(aFE, filename, IsStreaming=true);
```

Combine Streaming Feature Extraction and Classification

Save the trained network as a MAT file.

```
save("AirCompressorFaultRecognitionModel.mat", "airCompNet")
```

Create a function that combines the feature extraction and deep learning classification.

```
type recognizeAirCompressorFault.m

function scores = recognizeAirCompressorFault(audioIn, rs)
% This is a streaming classifier function

persistent airCompNet

if isempty(airCompNet)
    airCompNet = coder.loadDeepLearningNetwork('AirCompressorFaultRecognitionModel.mat');
end
if rs
    airCompNet = resetState(airCompNet);
end

% Extract features using function
features = extractAudioFeatures(audioIn);

% Classify
[airCompNet, scores] = predictAndUpdateState(airCompNet, features);

end
```

Test Streaming Loop

Next, you test the streaming classifier in MATLAB. Stream audio one frame at a time to represent a system as it would be deployed in a real-time embedded system. This enables you to measure and visualize the timing and accuracy of the streaming implementation.

Stream in several audio files and plot the output classification results for each frame of data. At a time interval equal to the length of each file, evaluate the output of the classifier.

```
reset(adsValidation)

N = 10;
labels = categories(ads.Labels);
numLabels = numel(labels);
```

```
% Create a dsp.AsyncBuffer to read audio in a streaming fashion
audioSource = dsp.AsyncBuffer;

% Create a dsp.AsyncBuffer to accumulate scores
scoreBuffer = dsp.AsyncBuffer;

% Create a dsp.AsyncBuffer to record execution time.
timingBuffer = dsp.AsyncBuffer;

% Pre-allocate array to store results
streamingResults = categorical(zeros(N,1));

% Loop over files
for fileIdx = 1:N

    % Read one audio file and put it in the source buffer
    [data,dataInfo] = read(adsValidation);
    write(audioSource,data);

    % Inner loop over frames
    rs = true;
    while audioSource.NumUnreadSamples >= windowLength

        % Get a frame of audio data
        x = read(audioSource>windowLength);

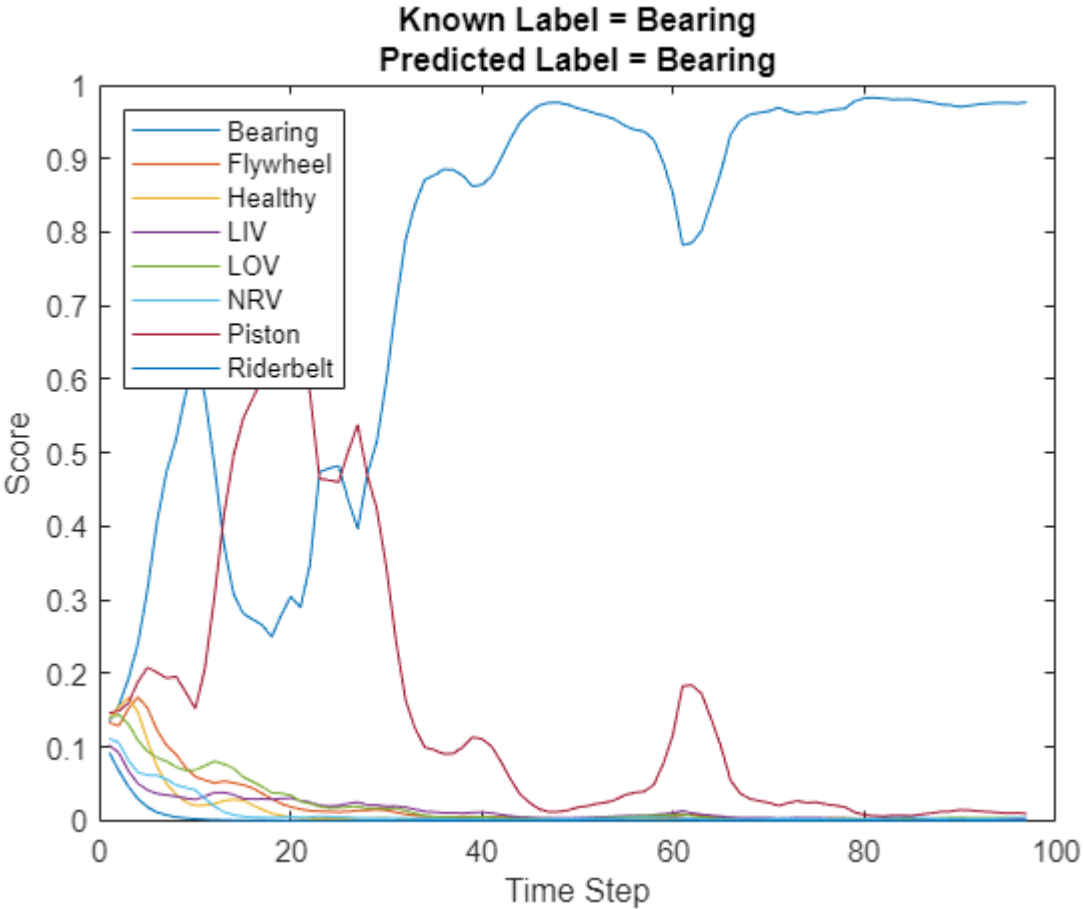
        % Apply streaming classifier function
        tic
        score = recognizeAirCompressorFault(x,rs);
        write(timingBuffer,toc);

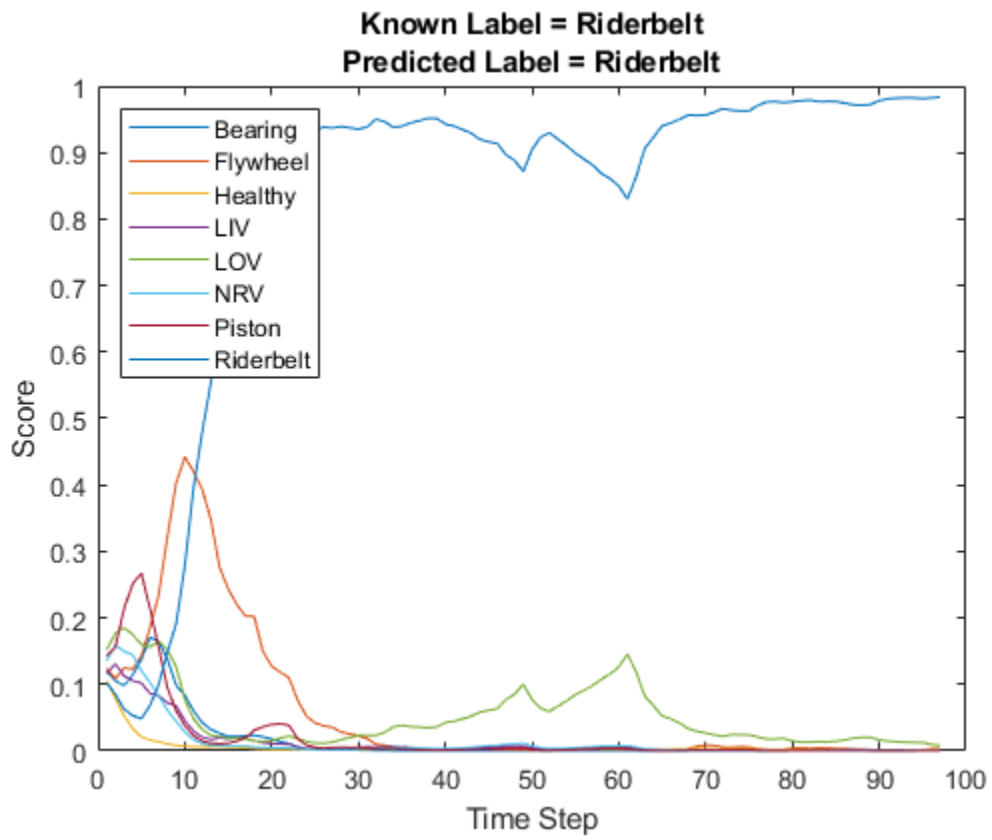
        % Store score for analysis
        write(scoreBuffer,score);

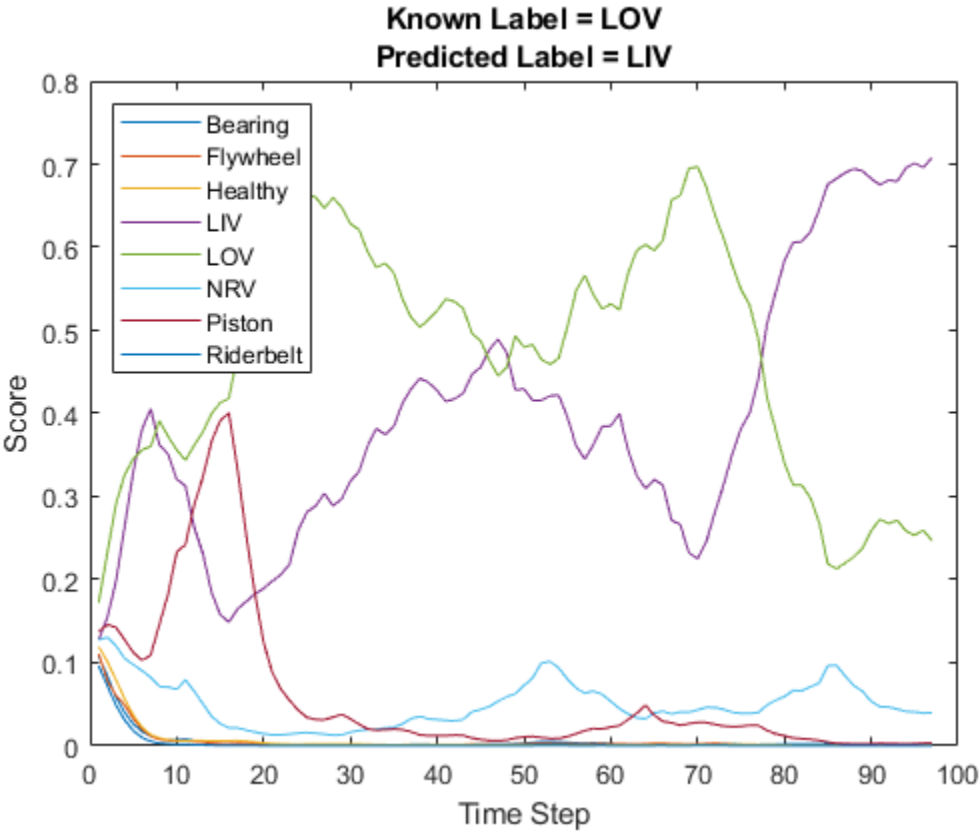
        rs = false;
    end
    reset(audioSource)

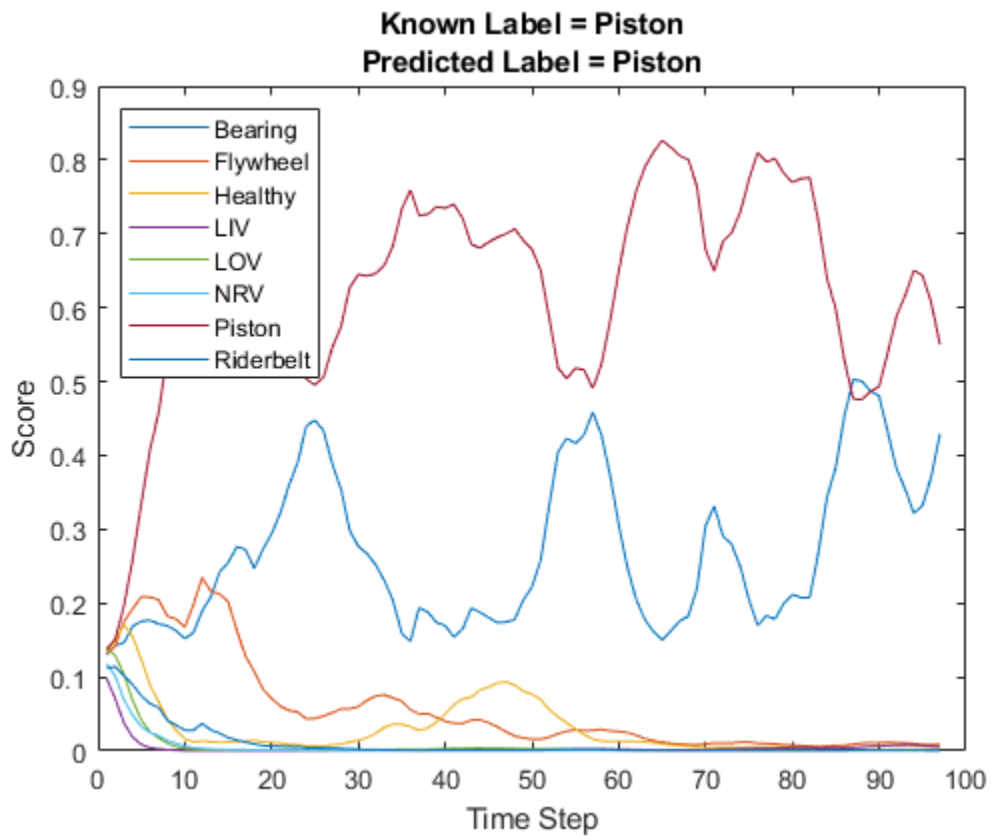
    % Store class result for that file
    scores = read(scoreBuffer);
    [~,result] = max(scores(end,:),[],2);
    streamingResults(fileIdx) = categorical(labels(result));

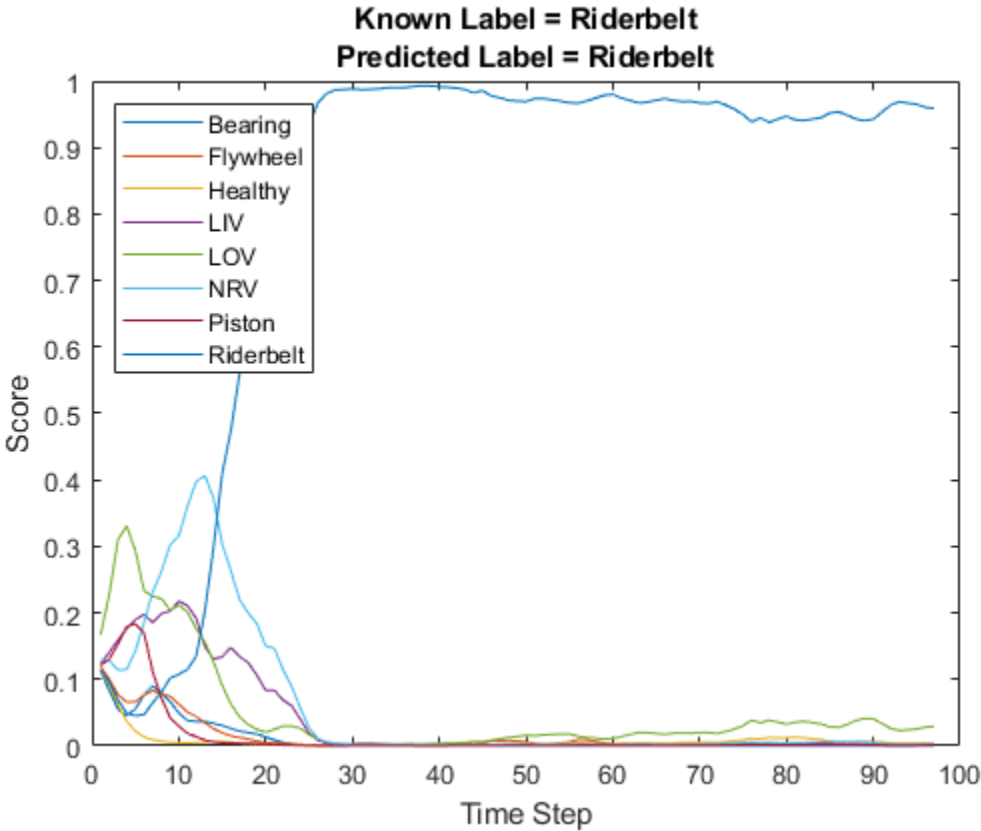
    % Plot scores to compare over time
    figure
    plot(scores) %#ok<*NASGU>
    legend(string(airCompNet.Layers(end).Classes),Location="northwest")
    xlabel("Time Step")
    ylabel("Score")
    title(["Known Label = " + string(dataInfo.Label),"Predicted Label = " + string(streamingResu
end
```

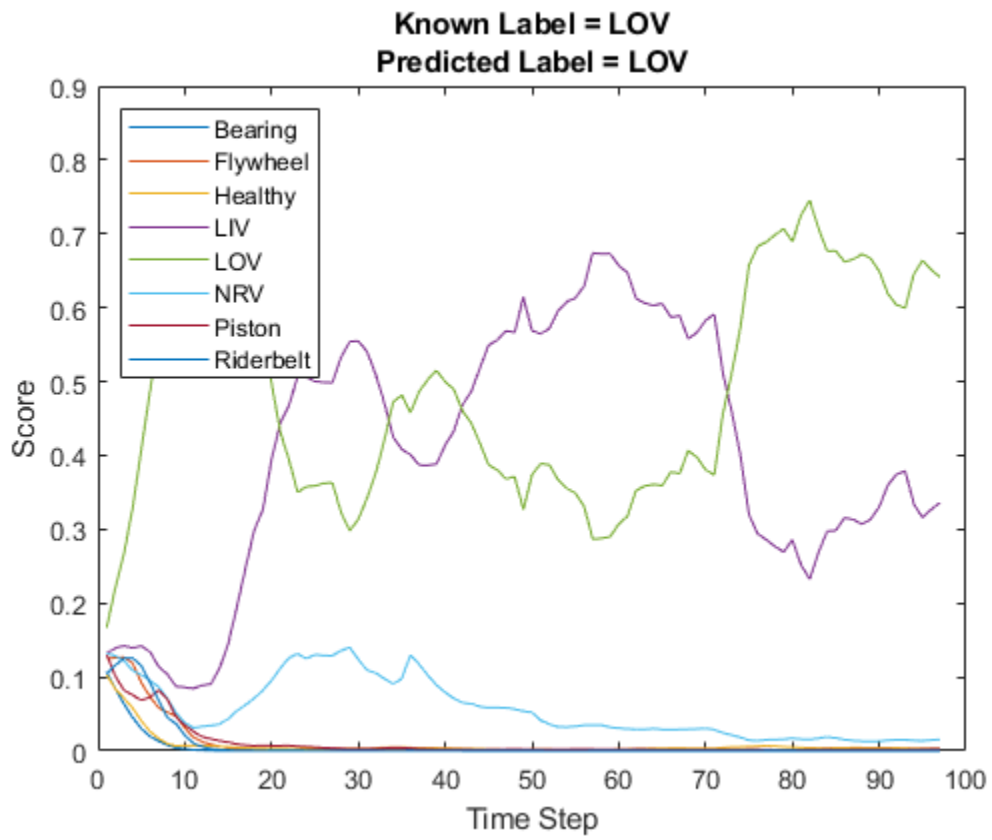


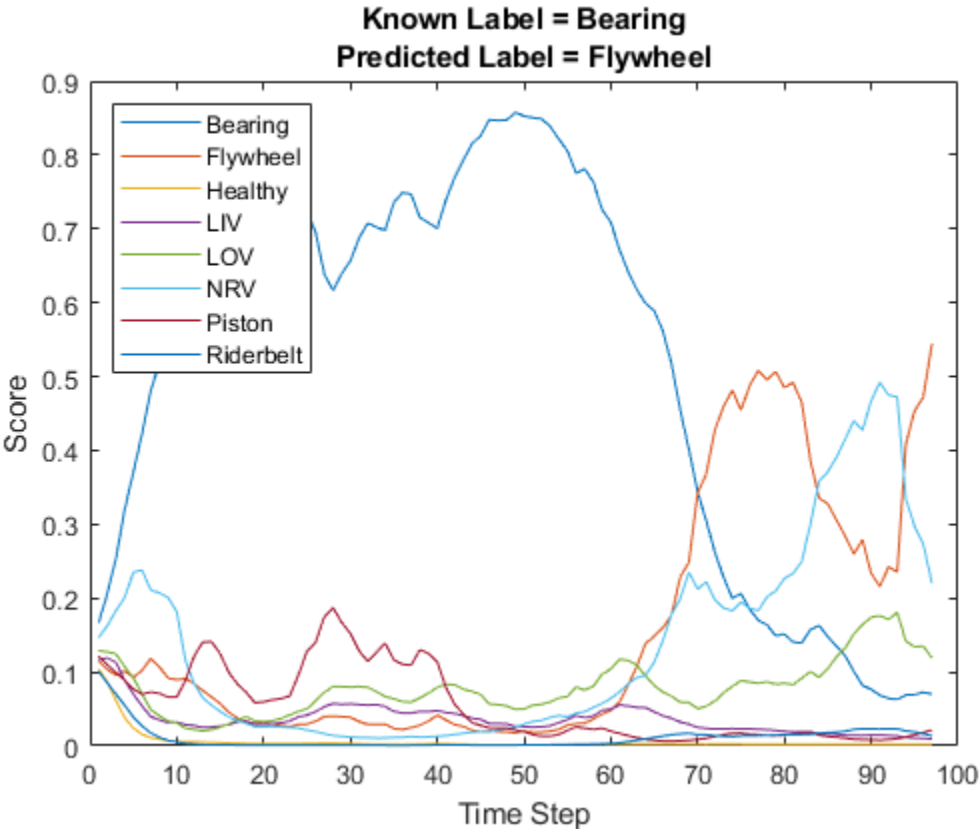


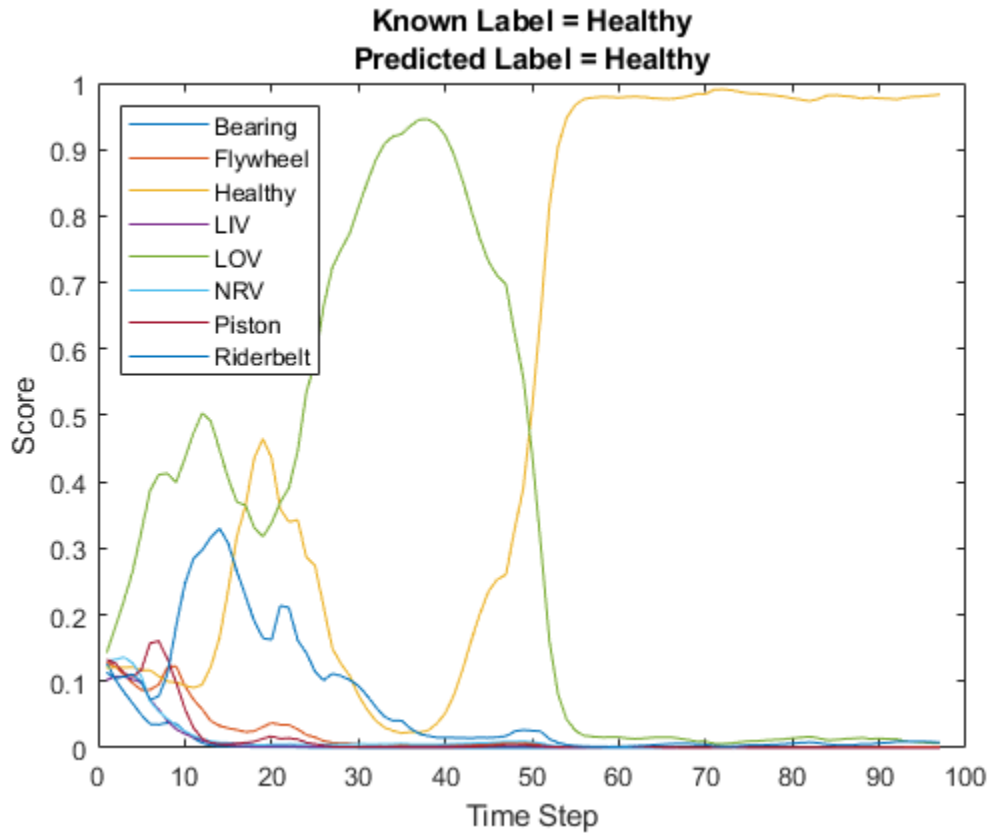


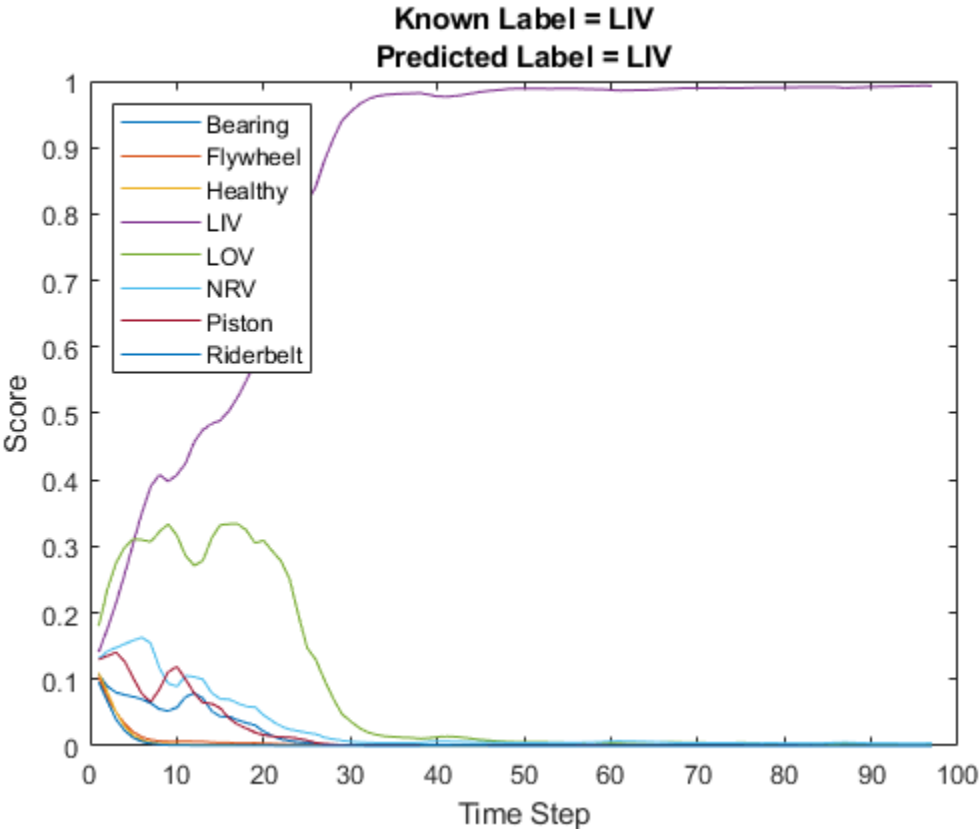


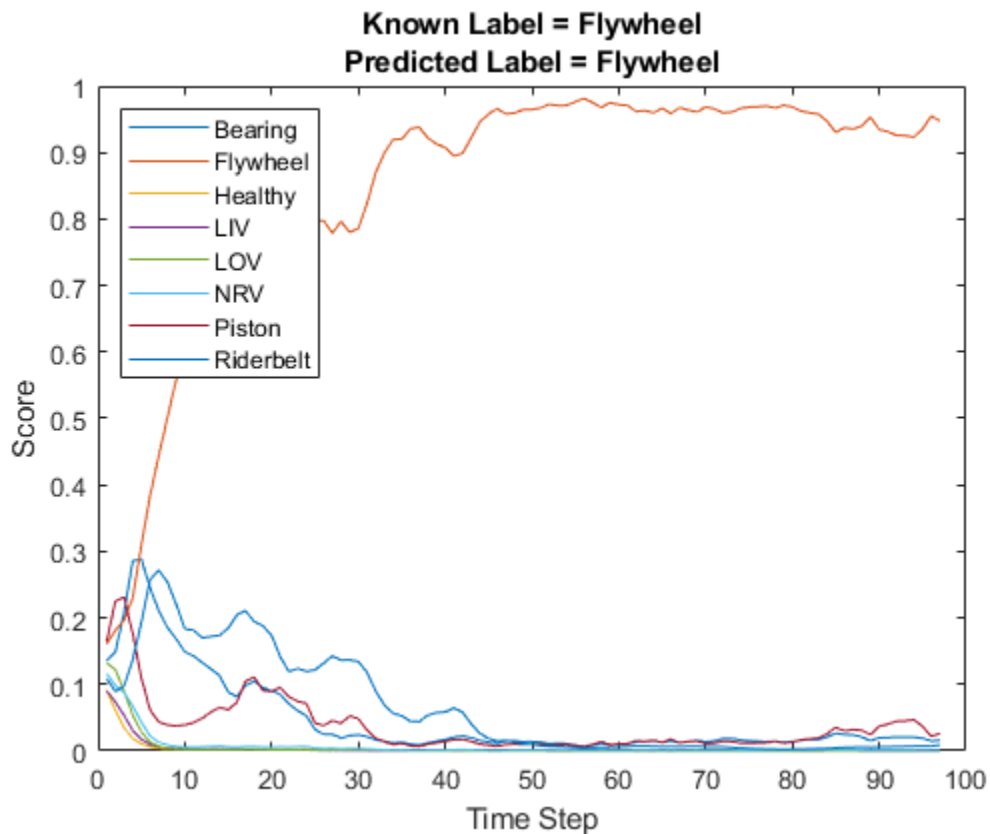












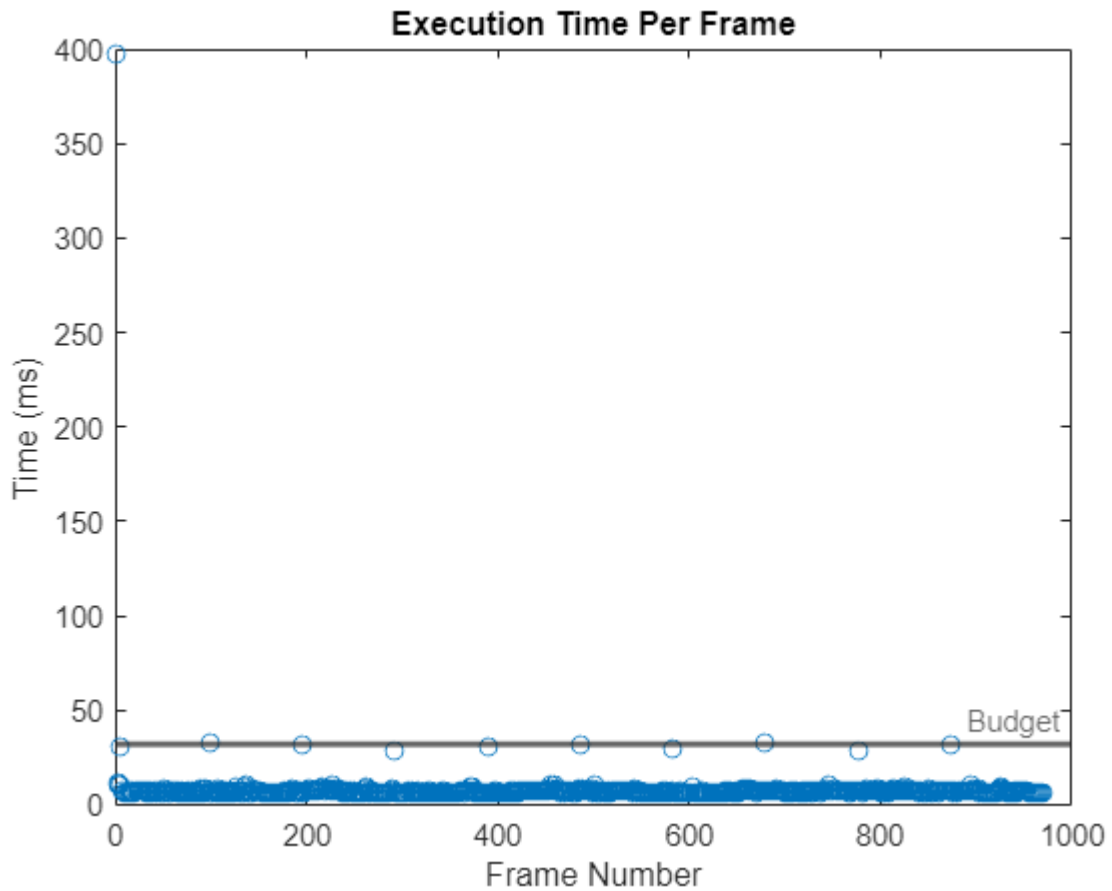
Compare the test results for the streaming version of the classifier and the non-streaming.

```
testError = mean(validationResults(1:N) ~= streamingResults);
disp("Error between streaming classifier and non-streaming: " + testError*100 + " (%)")
```

```
Error between streaming classifier and non-streaming: 0 (%)
```

Analyze the execution time. The execution time when state is reset is often above the 32 ms budget. However, in a real, deployed system, that initialization time will only be incurred once. The execution time of the main loop is around 10 ms, which is well below the 32 ms budget for real-time performance.

```
executionTime = read(timingBuffer)*1000;
budget = (windowLength/aFE.SampleRate)*1000;
plot(executionTime,"o")
title("Execution Time Per Frame")
xlabel("Frame Number")
ylabel("Time (ms)")
yline(budget,"-", "Budget", LineWidth=2)
```



Supporting Functions

```
function [augData,augLabels] = mixup(data,labels,numMixesPerInstance)
augData = cell(1,numel(data)*numMixesPerInstance);
augLabels = repelem(labels,numMixesPerInstance);

kk = 1;
for ii = 1:numel(data)
    for jj = 1:numMixesPerInstance
        lambda = max(min((randn./10)+0.5,1),0);

        % Find all available data with different labels.
        availableData = find(labels~=labels(ii));

        % Randomly choose one of the available data with a different label.
        numAvailableData = numel(availableData);
        idx = randi([1,numAvailableData]);

        % Mix.
        augData{kk} = lambda*data{ii} + (1-lambda)*data{availableData(idx)};

        % Specify the label as randomly set by lambda.
        if lambda < rand
```

```
        augLabels(kk) = labels(availableData(idx));  
    else  
        augLabels(kk) = labels(ii);  
    end  
    kk = kk + 1;  
end  
end  
end
```

References

[1] Verma, Nishchal K., et al. "Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors." *IEEE Transactions on Reliability*, vol. 65, no. 1, Mar. 2016, pp. 291-309. DOI.org (Crossref), doi:10.1109/TR.2015.2459684.

[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/>.

Acoustics-Based Machine Fault Recognition Code Generation with Intel MKL-DNN

This example demonstrates code generation for “Acoustics-Based Machine Fault Recognition” on page 1-737 using a long short-term memory (LSTM) network and spectral descriptors. This example uses MATLAB® Coder™ with deep learning support to generate a MEX (MATLAB executable) function that leverages performance of Intel® MKL-DNN library. The input data consists of acoustics time-series recordings from faulty or healthy air compressors and the output is the state of the mechanical machine predicted by the LSTM network. For details on audio preprocessing and network training, see “Acoustics-Based Machine Fault Recognition” on page 1-737.

Example Requirements

- The MATLAB Coder Interface for Deep Learning Support Package
- Intel processor with support for Intel Advanced Vector Extensions 2 (Intel AVX2)
- Intel Deep Neural Networks Library (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).

Prepare Input Dataset

Specify a sample rate `fs` of 16 kHz and a `windowLength` of 512 samples, as defined in “Acoustics-Based Machine Fault Recognition” on page 1-737. Set `numFrames` to 100.

```
fs = 16000;
windowLength = 512;
numFrames = 100;
```

To run the Example on a test signal, generate a pink noise signal. To test the performance of the system on a real dataset, download the air compressor dataset [1] on page 1-763.

```
downloadDataset = 

if ~downloadDataset
    pinkNoiseSignal = pinknoise(windowLength*numFrames);
else
    % Download AirCompressorDataset.zip
    component = 'audio';
    filename = 'AirCompressorDataset/AirCompressorDataset.zip';
    localfile = matlab.internal.examples.downloadSupportFile(component, filename);

    % Unzip the downloaded zip file to the downloadFolder
    downloadFolder = fileparts(localfile);
    if ~exist(fullfile(downloadFolder, 'AirCompressorDataset'), 'dir')
        unzip(localfile, downloadFolder)
    end

    % Create an audioDatastore object datastore, to manage, the data.
    datastore = audioDatastore(downloadFolder, 'IncludeSubfolders', true, 'LabelSource', 'foldernames');
```

```

    % Use countEachLabel to get the number of samples of each category in the dataset.
    countEachLabel(dataStore)
end

```

Recognize Machine Fault in MATLAB

To run the streaming classifier in MATLAB, download and unzip the system developed in “Acoustics-Based Machine Fault Recognition” on page 1-737.

```

component = 'audio';
filename = 'AcousticsBasedMachineFaultRecognition/AcousticsBasedMachineFaultRecognition.zip';
localfile = matlab.internal.examples.downloadSupportFile(component, filename);

downloadFolder = fullfile(fileparts(localfile), 'system');
if ~exist(downloadFolder, 'dir')
    unzip(localfile, downloadFolder)
end

```

To access the `recognizeAirCompressorFault` function of the system, add `downloadFolder` to the search path.

```
addpath(downloadFolder)
```

Create a `dsp.AsyncBuffer` object to read audio in a streaming fashion and a `dsp.AsyncBuffer` object to accumulate scores.

```

audioSource = dsp.AsyncBuffer;
scoreBuffer = dsp.AsyncBuffer;

```

Load the pretrained network and extract labels from the network.

```

airCompNet = coder.loadDeepLearningNetwork('AirCompressorFaultRecognitionModel.mat');
labels = string(airCompNet.Layers(end).Classes);

```

Initialize `signalToBeTested` to `pinkNoiseSignal` or select a signal from the drop-down list to test the file of your choice from the dataset.

```

if ~downloadDataset
    signalToBeTested = pinkNoiseSignal;
else
    [allFiles, ~] = splitEachLabel(dataStore, 1);
    allData = readall(allFiles);

    signalToBeTested = ;
    signalToBeTested = cell2mat(signalToBeTested);
end

```

Stream one audio frame at a time to represent the system as it would be deployed in a real-time embedded system. Use `recognizeAirCompressorFault` developed in “Acoustics-Based Machine Fault Recognition” on page 1-737 to compute audio features and perform deep learning classification.

```

write(audioSource, signalToBeTested);
resetNetworkState = true;

while audioSource.NumUnreadSamples >= windowLength

    % Get a frame of audio data
    x = read(audioSource, windowLength);

```



```

% Apply streaming classifier function
score = recognizeAirCompressorFault(x, resetNetworkState);

% Store score for analysis
write(scoreBuffer, score);

resetNetworkState = false;
end

```

Compute the recognized fault from scores and display it.

```

scores = read(scoreBuffer);
[~, labelIndex] = max(scores(end, :), [], 2);
detectedFault = labels(labelIndex)

```

```

detectedFault =
"Flywheel"

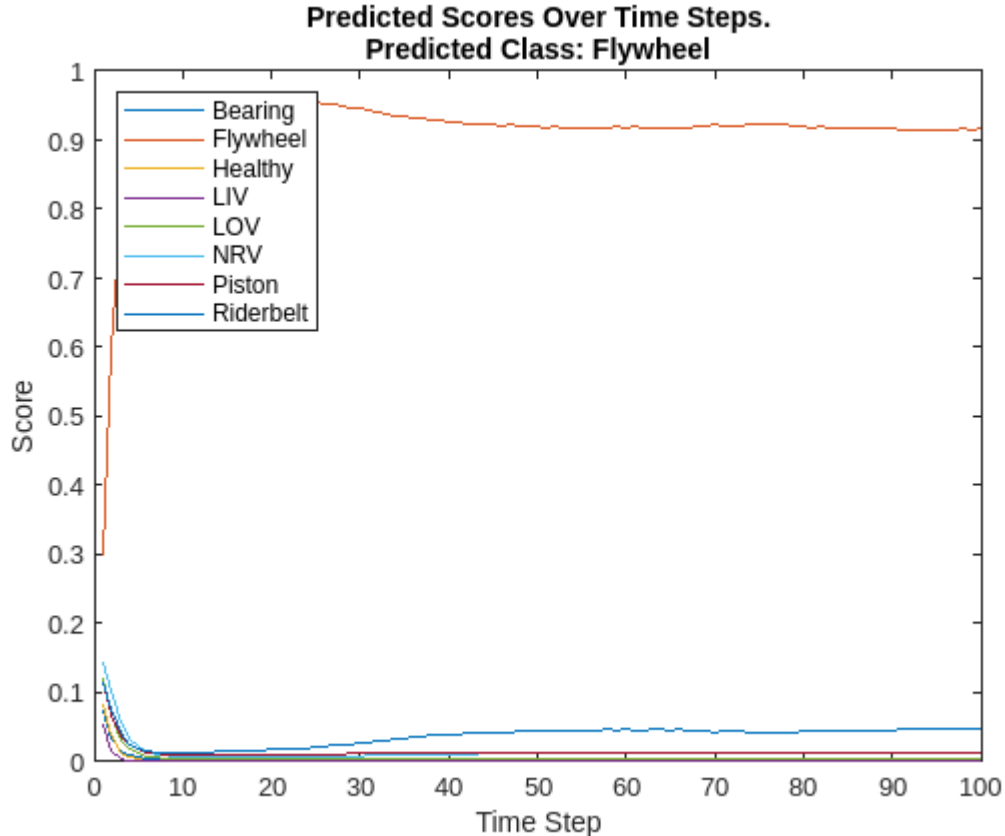
```

Plot the scores of each label for each frame.

```

plot(scores)
legend("" + labels, 'Location', 'northwest')
xlabel("Time Step")
ylabel("Score")
str = sprintf("Predicted Scores Over Time Steps.\nPredicted Class: %s", detectedFault);
title(str)

```



Generate MATLAB Executable

Create a code generation configuration object to generate an executable. 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 deep learning configuration object to the code generation configuration object.

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

Create an audio data frame of length `windowLength`.

```
audioFrame = ones(windowLength,1);
```

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

```
codegen -config cfg recognizeAirCompressorFault -args {audioFrame,resetNetworkState} -report
```

Code generation successful: [View report](#)

Perform Machine Fault Recognition Using MATLAB Executable

Initialize `signalToBeTested` to `pinkNoiseSignal` or select a signal from the drop-down list to test the file of your choice from the dataset.

```
if ~downloadDataset  
    signalToBeTested = pinkNoiseSignal;  
else  
    [allFiles,~] = splitEachLabel(dataStore,1);  
    allData = readall(allFiles);  
  
    signalToBeTested =  ;  
    signalToBeTested = cell2mat(signalToBeTested);  
end
```

Stream one audio frame at a time to represent the system as it would be deployed in a real-time embedded system. Use generated `recognizeAirCompressorFault_mex` to compute audio features and perform deep learning classification.

```
write(audioSource,signalToBeTested);  
resetNetworkState = true;
```

```
while audioSource.NumUnreadSamples >= windowLength  
  
    % Get a frame of audio data  
    x = read(audioSource>windowLength);  
  
    % Apply streaming classifier function  
    score = recognizeAirCompressorFault_mex(x,resetNetworkState);  
  
    % Store score for analysis
```

```

write(scoreBuffer,score);

resetNetworkState = false;
end

```

Compute the recognized fault from scores and display it.

```

scores = read(scoreBuffer);
[~,labelIndex] = max(scores(end,:),[],2);
detectedFault = labels(labelIndex)

```

```

detectedFault =
"Flywheel"

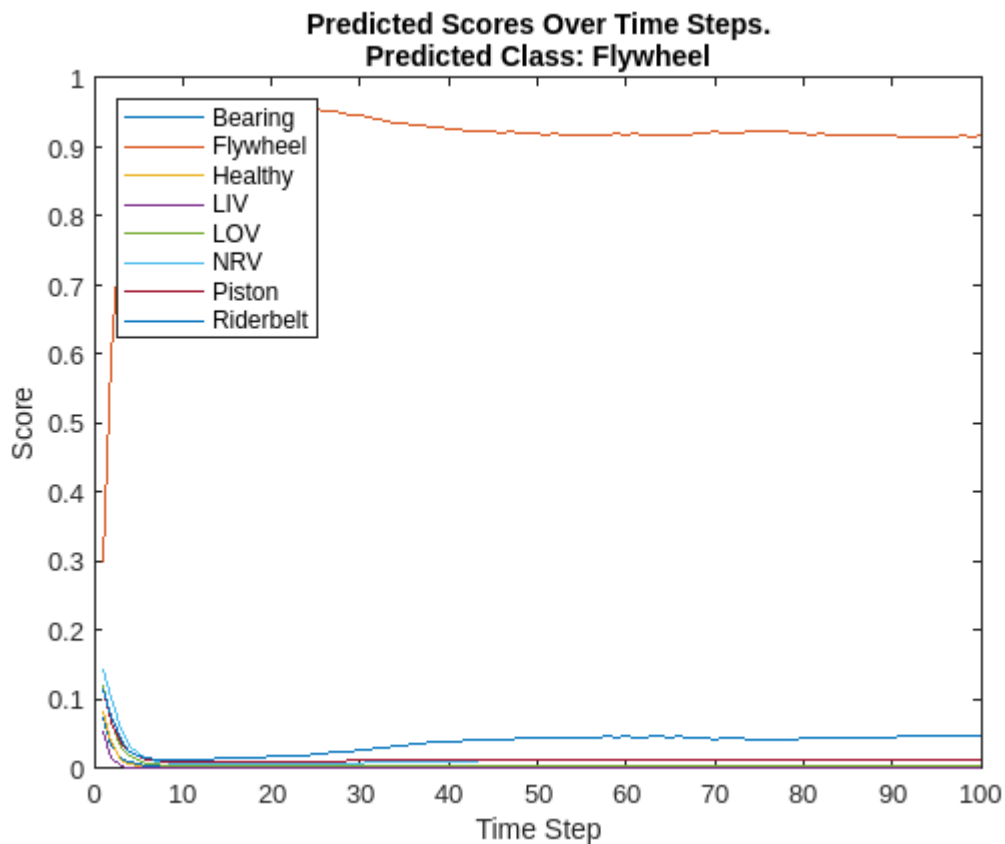
```

Plot the scores of each label for each frame.

```

plot(scores)
legend("" + labels,'Location','northwest')
xlabel("Time Step")
ylabel("Score")
str = sprintf("Predicted Scores Over Time Steps.\nPredicted Class: %s",detectedFault);
title(str)

```



Evaluate Execution Time of Alternative MEX Function Workflow

Use `tic` and `toc` to measure the execution time of MATLAB function `recognizeAirCompressorFault` and MATLAB executable (MEX) `recognizeAirCompressorFault_mex`.

Create a `dsp.AsyncBuffer` object to record execution time.

```
timingBufferMATLAB = dsp.AsyncBuffer;  
timingBufferMEX = dsp.AsyncBuffer;
```

Use same recording that you chose in previous section as input to `recognizeAirCompressorFault` function and its MEX equivalent `recognizeAirCompressorFault_mex`.

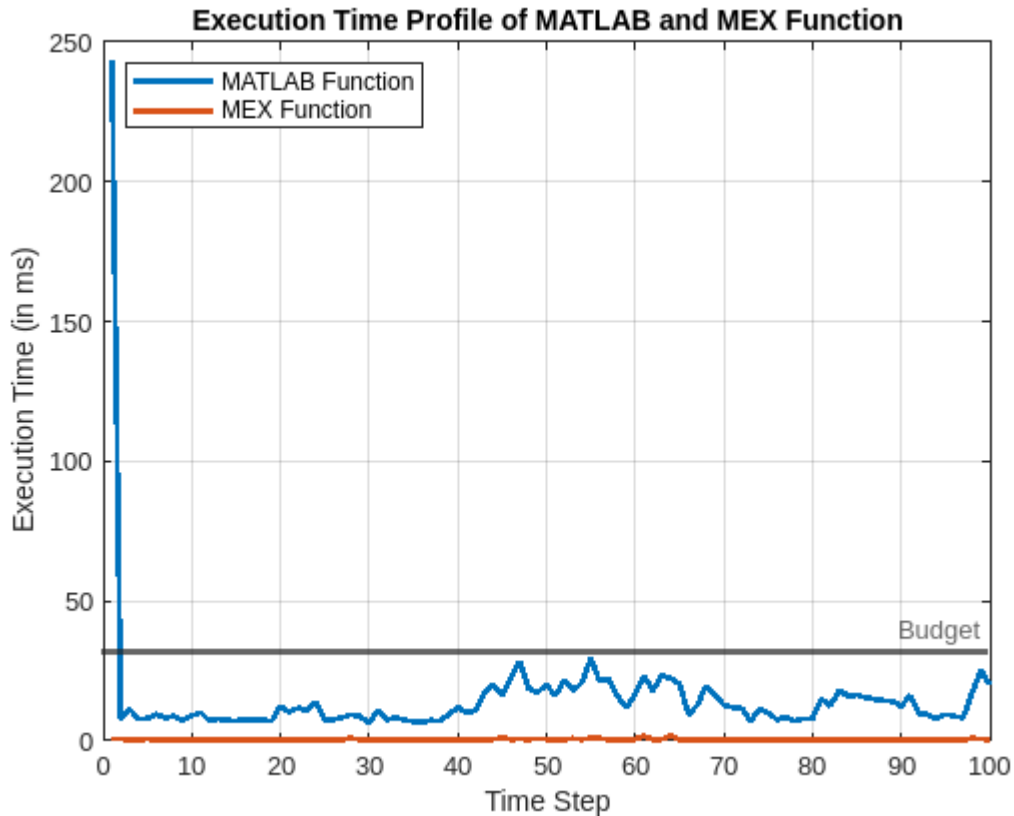
```
write(audioSource,signalToBeTested);
```

Measure the execution time of the MATLAB code.

```
resetNetworkState = true;  
while audioSource.NumUnreadSamples >= windowLength  
    % Get a frame of audio data  
    x = read(audioSource,windowLength);  
  
    % Apply streaming classifier function  
    tic  
    scoreMATLAB = recognizeAirCompressorFault(x,resetNetworkState);  
    write(timingBufferMATLAB,toc);  
  
    % Apply streaming classifier MEX function  
    tic  
    scoreMEX = recognizeAirCompressorFault_mex(x,resetNetworkState);  
    write(timingBufferMEX,toc);  
  
    resetNetworkState = false;  
  
end
```

Plot the execution time for each frame and analyze the profile. The first call of `recognizeAirCompressorFault_mex` consumes around four times of the budget as it includes loading of network and resetting of the states. However, in a real, deployed system, that initialization time is only incurred once. The execution time of the MATLAB function is around 10 ms and that of MEX function is ~1 ms, which is well below the 32 ms budget for real-time performance.

```
budget = (windowLength/fs)*1000;  
timingMATLAB = read(timingBufferMATLAB)*1000;  
timingMEX = read(timingBufferMEX)*1000;  
frameNumber = 1:numel(timingMATLAB);  
perfGain = timingMATLAB./timingMEX;  
plot(frameNumber,timingMATLAB,frameNumber,timingMEX,'LineWidth',2)  
grid on  
yline(budget,'',{'Budget'},'LineWidth',2)  
legend('MATLAB Function','MEX Function','Location','northwest')  
xlabel("Time Step")  
ylabel("Execution Time (in ms)")  
title("Execution Time Profile of MATLAB and MEX Function")
```



Compute the performance gain of MEX over MATLAB function excluding the first call. This performance test is done on a machine using an NVIDIA Quadro P620 (Version 26) GPU and an Intel® Xeon® W-2133 CPU running at 3.60 GHz.

```
PerformanceGain = sum(timingMATLAB(2:end))/sum(timingMEX(2:end))
```

```
PerformanceGain = 24.0484
```

This example ends here. For deploying machine fault recognition on Raspberry Pi, see “Acoustics-Based Machine Fault Recognition Code Generation on Raspberry Pi” on page 1-764.

References

[1] Verma, Nishchal K., et al. "Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors." *IEEE Transactions on Reliability*, vol. 65, no. 1, Mar. 2016, pp. 291-309. DOI.org (Crossref), doi:10.1109/TR.2015.2459684.

Acoustics-Based Machine Fault Recognition Code Generation on Raspberry Pi

This example demonstrates code generation for “Acoustics-Based Machine Fault Recognition” on page 1-737 using a long short-term memory (LSTM) network and spectral descriptors. This example uses MATLAB® Coder™, MATLAB Coder Interface for Deep Learning, MATLAB Support Package for Raspberry Pi™ Hardware to generate a standalone executable (.elf) file on a Raspberry Pi that leverages performance of the ARM® Compute Library. The input data consists of acoustics time-series recordings from faulty or healthy air compressors and the output is the state of the mechanical machine predicted by the LSTM network. This standalone executable on Raspberry Pi runs the streaming classifier on the input data received from MATLAB and sends the computed scores for each label to MATLAB. Interaction between MATLAB script and the executable on your Raspberry Pi is handled using the user datagram protocol (UDP). For more details on audio preprocessing and network training, see “Acoustics-Based Machine Fault Recognition” on page 1-737.

Example Requirements

- The MATLAB Coder Interface for Deep Learning Support Package
- ARM processor that supports the NEON extension
- ARM Compute Library version 20.02.1 (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)

Prepare Input Dataset

Specify a sample rate `fs` of 16 kHz and a `windowLength` of 512 samples, as defined in “Acoustics-Based Machine Fault Recognition” on page 1-737. Set `numFrames` to 100.

```
fs = 16000;
windowLength = 512;
numFrames = 100;
```

To run the Example on a test signal, generate a pink noise signal. To test the performance of the system on a real dataset, download the air compressor dataset [1] on page 1-774.

```
downloadDataset = 
if ~downloadDataset
    pinkNoiseSignal = pinknoise(windowLength*numFrames);
else
    % Download AirCompressorDataset.zip
    component = 'audio';
    filename = 'AirCompressorDataset/AirCompressorDataset.zip';
    localfile = matlab.internal.examples.downloadSupportFile(component, filename);

    % Unzip the downloaded zip file to the downloadFolder
    downloadFolder = fileparts(localfile);
    if ~exist(fullfile(downloadFolder, 'AirCompressorDataset'), 'dir')
        unzip(localfile, downloadFolder)
    end
end
```

```

% Create an audioDatastore object datastore, to manage, the data.
dataStore = audioDatastore(downloadFolder, 'IncludeSubfolders', true, 'LabelSource', 'foldernames');

% Use countEachLabel to get the number of samples of each category in the dataset.
countEachLabel(dataStore)
end

```

Recognize Machine Fault in MATLAB

To run the streaming classifier in MATLAB, download and unzip the system developed in “Acoustics-Based Machine Fault Recognition” on page 1-737.

```

component = 'audio';
filename = 'AcousticsBasedMachineFaultRecognition/AcousticsBasedMachineFaultRecognition.zip';
localfile = matlab.internal.examples.downloadSupportFile(component, filename);

downloadFolder = fullfile(fileparts(localfile), 'system');
if ~exist(downloadFolder, 'dir')
    unzip(localfile, downloadFolder)
end

```

To access the `recognizeAirCompressorFault` function of the system, add `downloadFolder` to the search path.

```
addpath(downloadFolder)
```

Create a `dsp.AsyncBuffer` object to read audio in a streaming fashion and a `dsp.AsyncBuffer` object to accumulate scores.

```
audioSource = dsp.AsyncBuffer;
scoreBuffer = dsp.AsyncBuffer;
```

Load the pretrained network and extract labels from the network.

```
airCompNet = coder.loadDeepLearningNetwork('AirCompressorFaultRecognitionModel.mat');
labels = string(airCompNet.Layers(end).Classes);
```

Initialize `signalToBeTested` to `pinkNoiseSignal` or select a signal from the drop-down list to test the file of your choice from the dataset.

```

if ~downloadDataset
    signalToBeTested = pinkNoiseSignal;
else
    [allFiles, ~] = splitEachLabel(dataStore, 1);
    allData = readall(allFiles);

    signalToBeTested =  ;
    signalToBeTested = cell2mat(signalToBeTested);
end

```

Stream one audio frame at a time to represent the system as it would be deployed in a real-time embedded system. Use `recognizeAirCompressorFault` developed in “Acoustics-Based Machine Fault Recognition” on page 1-737 to compute audio features and perform deep learning classification.

```
write(audioSource, signalToBeTested);
resetNetworkState = true;
```

```
while audioSource.NumUnreadSamples >= windowLength

    % Get a frame of audio data
    x = read(audioSource,windowLength);

    % Apply streaming classifier function
    score = recognizeAirCompressorFault(x,resetNetworkState);

    % Store score for analysis
    write(scoreBuffer,score);

    resetNetworkState = false;
end
```

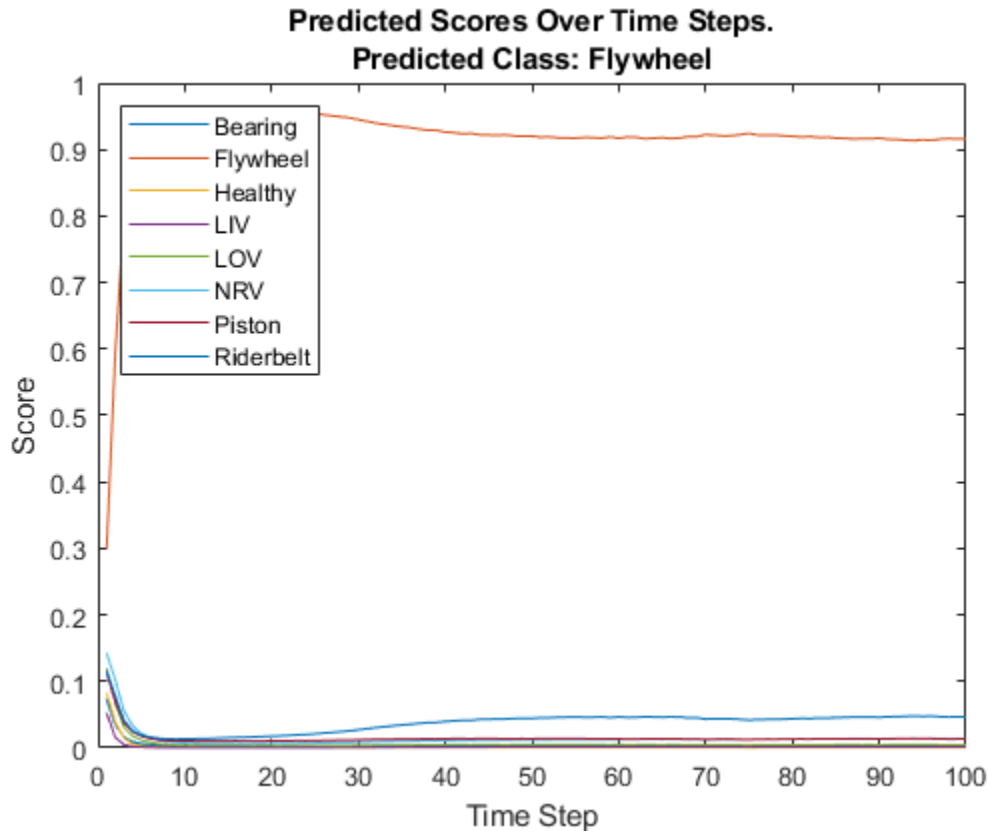
Compute the recognized fault from scores and display it.

```
scores = read(scoreBuffer);
[~,labelIndex] = max(scores(end,:),[],2);
detectedFault = labels(labelIndex)
```

```
detectedFault =
"Flywheel"
```

Plot the scores of each label for each frame.

```
plot(scores)
legend("" + labels,'Location','northwest')
xlabel("Time Step")
ylabel("Score")
str = sprintf("Predicted Scores Over Time Steps.\nPredicted Class: %s",detectedFault);
title(str)
```

Reset the asynchronous buffer audioSource.

```
reset(audioSource)
```

Prepare MATLAB Code For Deployment

This example uses the `dsp.UDPSender` System object to send the audio frame to the executable running on Raspberry Pi and the `dsp.UDPReceiver` System object to receive the score vector from the Raspberry Pi. Create a `dsp.UDPSender` system object to send audio captured in MATLAB to your Raspberry Pi. Set the `targetIPAddress` to the IP address of your Raspberry Pi. Set the `RemoteIPPort` to 25000. Raspberry Pi receives the input audio frame from the same port using the `dsp.UDPReceiver` system object.

```
targetIPAddress = '172.31.164.247';
UDPSend = dsp.UDPSender('RemoteIPPort',25000,'RemoteIPAddress',targetIPAddress);
```

Create a `dsp.UDPReceiver` system object to receive predicted scores from your Raspberry Pi. Each UDP packet received from the Raspberry Pi is a vector of scores and each vector element is a score for a state of the air compressor. The maximum message length for the `dsp.UDPReceiver` object is 65507 bytes. Calculate the buffer size to accommodate the maximum number of UDP packets.

```
sizeofDoubleInBytes = 8;
numScores = 8;
maxUDPMessageLength = floor(65507/sizeofDoubleInBytes);
numPackets = floor(maxUDPMessageLength/numScores);
bufferSize = numPackets*numScores*sizeofDoubleInBytes;
```

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

Create a supporting function, `recognizeAirCompressorFaultRaspi`, that receives an audio frame using `dsp.UDPReceiver` and applies the streaming classifier and sends the predicted score vector to MATLAB using `dsp.UDPSender`.

```
type recognizeAirCompressorFaultRaspi

function recognizeAirCompressorFaultRaspi(hostIPAddress)
% This function receives acoustic input using dsp.UDPReceiver and runs a
% streaming classifier by calling recognizeAirCompressorFault, developed in
% the Acoustics-Based Machine Fault Recognition - MATLAB Example.
% Computed scores are sent to MATLAB using dsp.UDPSender.
%#codegen

% Copyright 2021 The MathWorks, Inc.

frameLength = 512;

% Configure UDP Sender System Object
UDPSend = dsp.UDPSender('RemoteIPPort',21000,'RemoteIPAddress',hostIPAddress);

% Configure UDP Receiver system object
sizeofDoubleInBytes = 8;
maxUDPMessageLength = floor(65507/sizeofDoubleInBytes);
numPackets = floor(maxUDPMessageLength/frameLength);
bufferSize = numPackets*frameLength*sizeofDoubleInBytes;
UDPReceiveRaspi = dsp.UDPReceiver('LocalIPPort',25000, ...
    'MaximumMessageLength',frameLength, ...
    'ReceiveBufferSize',bufferSize, ...
    'MessageDataType','double');

% Reset network state for first call
resetNetworkState = true;

while true
    % Receive audio frame of size frameLength x 1
    x = UDPReceiveRaspi();

    if(~isempty(x))

        x = x(1:frameLength,1);

        % Apply streaming classifier function
        scores = recognizeAirCompressorFault(x,resetNetworkState);

        %Send output to the host machine
        UDPSend(scores);

        resetNetworkState = false;
    end
end
```

Generate Executable on Raspberry Pi

Replace the `hostIPAddress` with your machine's address. Your Raspberry Pi sends the predicted scores to the IP address you specify.

```
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 = '20.02.1';
cfg.DeepLearningConfig = dlcfg;
```

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

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

```
if (~exist('r','var'))
    r = raspi('raspiname','pi','password');
end
```

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 autogenerated C++ main file to generate a standalone executable.

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

Call the `codegen` (MATLAB Coder) function from MATLAB Coder to generate C++ code and the executable on your Raspberry Pi. By default, the Raspberry Pi executable has the same name as the MATLAB function. You get a warning in the code generation logs that you can disregard because `recognizeAirCompressorFaultRaspi` has an infinite loop that looks for an audio frame from MATLAB.

```
codegen -config cfg recognizeAirCompressorFaultRaspi -args {hostIPAddress} -report
```

```
    Deploying code. This may take a few minutes.
Warning: Function 'recognizeAirCompressorFaultRaspi' does not terminate due to an infinite loop.
```

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

Perform Machine Fault Recognition Using Deployed Code

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

```
applicationName = 'recognizeAirCompressorFaultRaspi';  
  
applicationDirPaths = raspi.utils.getRemoteBuildDirectory('applicationName',applicationName);  
targetDirPath = applicationDirPaths{1}.directory;  
  
exeName = strcat(applicationName, '.elf');  
command = ['cd ',targetDirPath,'; ./',exeName,' &> 1 &'];  
  
system(r,command);
```

Initialize `signalToBeTested` to `pinkNoiseSignal` or select a signal from the drop-down list to test the file of your choice from the dataset.

```
if ~downloadDataset  
    signalToBeTested = pinkNoiseSignal;  
else  
    [allFiles,~] = splitEachLabel(dataStore,1);  
    allData = readall(allFiles);  
  
    signalToBeTested =  ;  
    signalToBeTested = cell2mat(signalToBeTested);  
end
```

Stream one audio frame at a time to represent a system as it would be deployed in a real-time embedded system. Use the generated MEX file `recognizeAirCompressorFault_mex` to compute audio features and perform deep learning classification.

```
write(audioSource,signalToBeTested);  
  
while audioSource.NumUnreadSamples >= windowLength  
    x = read(audioSource,windowLength);  
    UDPSend(x);  
    score = UDPReceive();  
    if ~isempty(score)  
        write(scoreBuffer,score');  
    end  
end
```

Compute the recognized fault from scores and display it.

```
scores = read(scoreBuffer);  
[~,labelIndex] = max(scores(end,:), [],2);  
detectedFault = labels(labelIndex)  
  
detectedFault =  
"Flywheel"
```

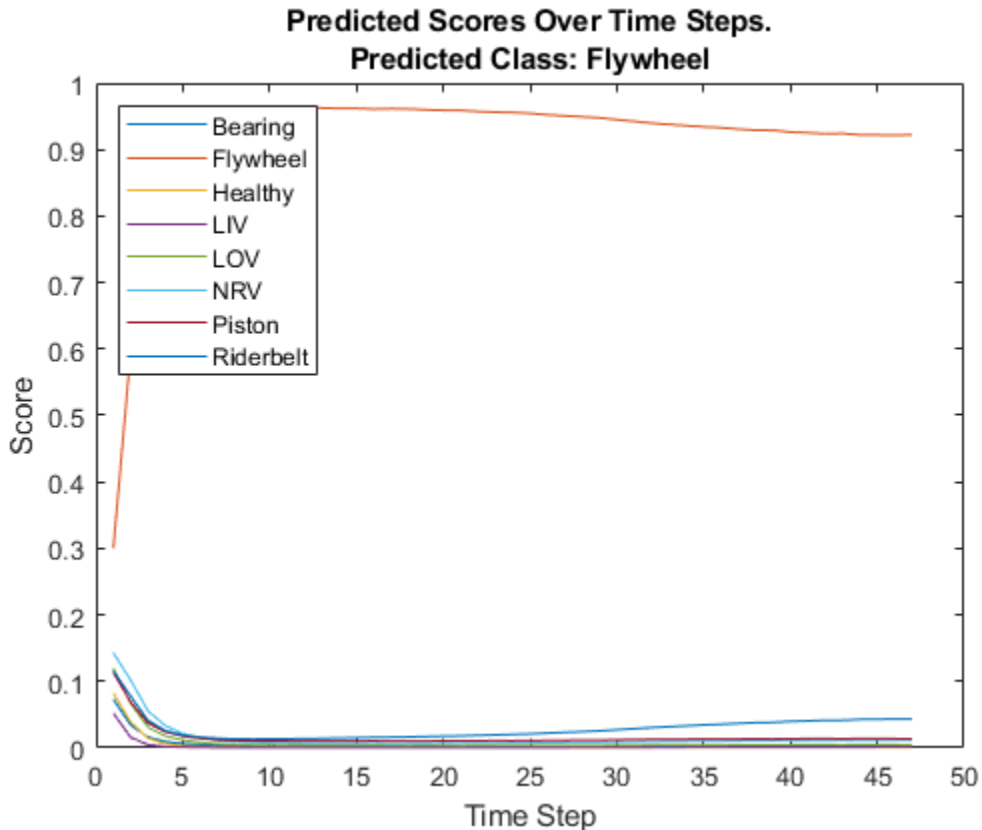
Plot the scores of each label for each frame.

```
plot(scores)  
legend("" + labels,'Location','northwest')
```

```

xlabel("Time Step")
ylabel("Score")
str = sprintf("Predicted Scores Over Time Steps.\nPredicted Class: %s",detectedFault);
title(str)

```



Terminate the standalone executable running on Raspberry Pi.

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

Evaluate Execution Time Using Alternative PIL Function Workflow

To evaluate execution time taken by standalone executable on Raspberry Pi, use a PIL (processor-in-loop) workflow. To perform PIL profiling, generate a PIL function for the supporting function `recognizeAirCompressorFault`.

Create a code generation configuration object to generate the PIL function.

```

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 = '20.02.1';

```

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 generate the PIL code. A MEX file named `recognizeAirCompressorFault_pil` is generated in your current folder.

```
cfg.CodeExecutionProfiling = true;
audioFrame = ones(windowLength,1);
resetNetworkStateFlag = true;
codegen -config cfg recognizeAirCompressorFault -args {audioFrame,resetNetworkStateFlag}
```

Deploying code. This may take a few minutes.

```
### Connectivity configuration for function 'recognizeAirCompressorFault': 'Raspberry Pi'
Location of the generated elf : /home/pi/remoteBuildDir/MATLAB_ws/R2021b/S/MATLAB/Examples/Examp
Code generation successful.
```

Call the generated PIL function 50 times to get the average execution time.

```
totalCalls = 50;

for k = 1:totalCalls
    x = pinknoise(windowLength,1);
    score = recognizeAirCompressorFault_pil(x,resetNetworkStateFlag);
    resetNetworkStateFlag = false;
end
```

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

Terminate the PIL execution.

```
clear recognizeAirCompressorFault_pil

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

### Connectivity configuration for function 'recognizeAirCompressorFault': 'Raspberry Pi'
Execution profiling report: report(getCoderExecutionProfile('recognizeAirCompressorFault'))
```

Generate an execution profile report to evaluate execution time.

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

Code Execution Profiling Report

Find: Match Case

Code Execution Profiling Report for recognizeAirCompressorFault

The code execution profiling report provides metrics based on data collected from a SIL or PIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. See [Code Execution Profiling](#) for more information.

1. Summary

Total time	42.3507
Unit of time	ms
Command	report(executionProfile, 'Units', 'seconds', 'ScaleFactor', '0.001', 'NumericFormat', '%0.4f');
Timer frequency (ticks per second)	1e+09
Profiling data created	08-Jun-2021 01:56:20

2. Profiled Sections of Code

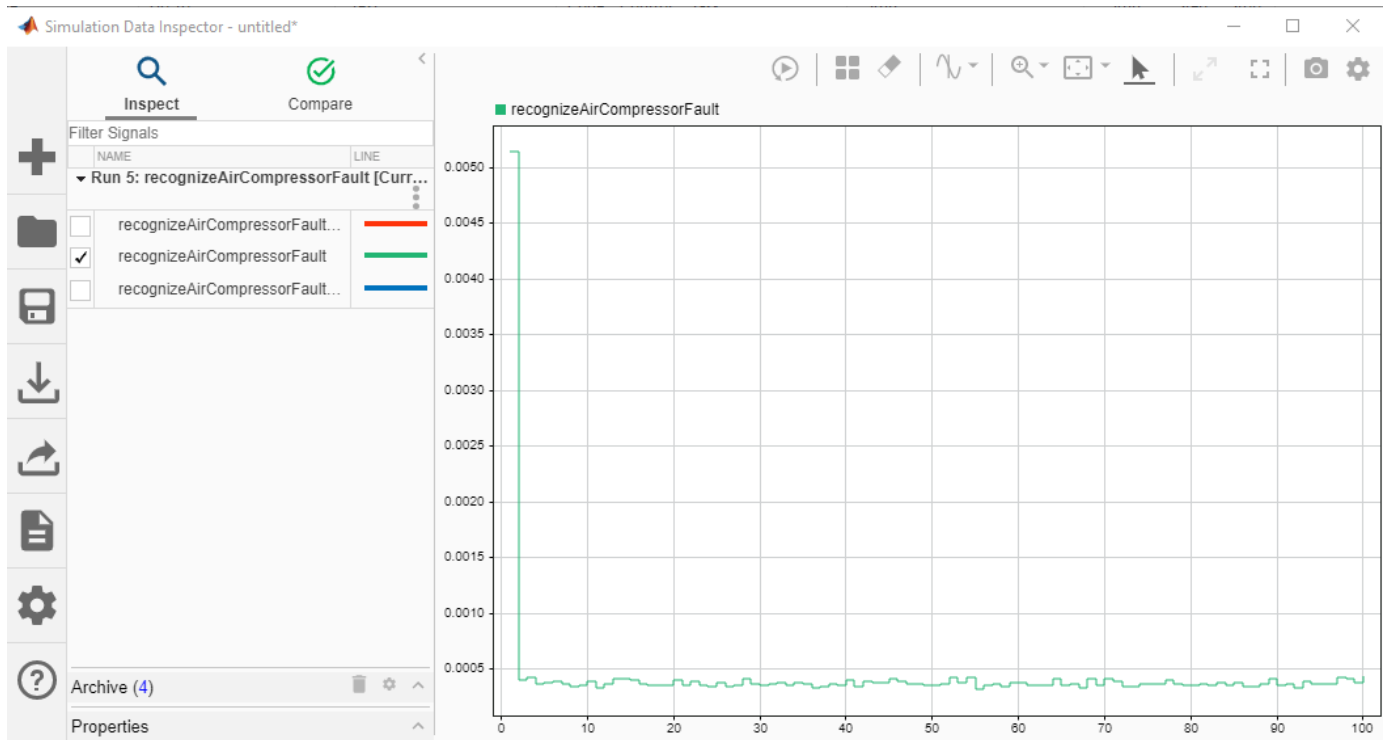
Section	Maximum Execution Time in ms	Average Execution Time in ms	Maximum Self Time in ms	Average Self Time in ms	Calls	
recognizeAirCompressorFault_initialize	0.0488	0.0488	0.0488	0.0488	1	
recognizeAirCompressorFault	5.1386	0.4230	5.1386	0.4230	100	
recognizeAirCompressorFault_terminate	0.0006	0.0006	0.0006	0.0006	1	

3. Definitions

Execution Time: Time between start and end of code section.

Self Time: Execution time, excluding time in child sections.

OK Help



The average execution time of `recognizeAirCompressorFault_pil` function is 0.423 ms, which is well below the 32 ms budget for real-time performance. The first call of `recognizeAirCompressorFault_pil` consumes around 12 times of the average execution time as it includes loading of network and resetting of the states. However, in a real, deployed system, that initialization time is incurred only once. This example ends here. For deploying machine fault recognition on desktops, see “Acoustics-Based Machine Fault Recognition Code Generation with Intel MKL-DNN” on page 1-757.

References

[1] Verma, Nishchal K., et al. "Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors." *IEEE Transactions on Reliability*, vol. 65, no. 1, Mar. 2016, pp. 291-309. DOI.org (Crossref), doi:10.1109/TR.2015.2459684.

audioDatastore Object Pointing to Audio Files

To create an `audioDatastore` object, first specify the file path to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot, 'toolbox', 'audio', 'samples');
```

Create an `audioDatastore` object that points to the specified folder of audio files.

```
ADS = audioDatastore(folder)
```

```
ADS =
```

```
audioDatastore with properties:
```

```

    Files: {
        'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
        'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs.flac'
        '...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5secs.wav'
        ... and 33 more
    }
    Folders: {
        'B:\matlab\toolbox\audio\samples'
    }
    AlternateFileSystemRoots: {}
    OutputDataType: 'double'
    Labels: {}
    SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
    DefaultOutputFormat: "wav"

```

Generate a subset of the audio datastore that only includes audio files containing 'Guitar' in the file name.

```
fileContainsGuitar = cellfun(@(c)contains(c, 'Guitar'), ADS.Files);
ADSsubset = subset(ADS, fileContainsGuitar)
```

```
ADSsubset =
```

```
audioDatastore with properties:
```

```

    Files: {
        'B:\matlab\toolbox\audio\samples\RockGuitar-16-44p1-stereo-72secs.flac'
        'B:\matlab\toolbox\audio\samples\RockGuitar-16-96-stereo-72secs.flac'
        'B:\matlab\toolbox\audio\samples\SoftGuitar-44p1_mono-10mins.ogg'
    }
    Folders: {
        'B:\matlab\toolbox\audio\samples'
    }
    AlternateFileSystemRoots: {}
    OutputDataType: 'double'
    Labels: {}
    SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
    DefaultOutputFormat: "wav"

```

Use the subset audio datastore as the source for a `labeledSignalSet` object.

```
audioLabSigSet = labeledSignalSet(ADSsubset)
```

```
audioLabSigSet =  
  labeledSignalSet with properties:  
  
      Source: {3x1 cell}  
      NumMembers: 3  
      TimeInformation: "inherent"  
      Labels: [3x0 table]  
      Description: ""
```

Use `labelDefinitionsHierarchy` to see a list of labels and sublabels.
Use `setLabelValue` to add data to the set.

Open **Signal Labeler** and use **Import From Workspace** to import the `labeledSignalSet`.

Accelerate Audio Deep Learning Using GPU-Based Feature Extraction

In this example, you leverage GPUs for feature extraction and augmentation to decrease the time required to train a deep learning model. The model you train is a convolutional neural network (CNN) for acoustic fault recognition.

Audio Toolbox™ includes `gpuArray` (Parallel Computing Toolbox) support for most feature extractors, including popular ones such as `melSpectrogram` and `mfcc`. For an overview of GPU support, see “Code Generation and GPU Support”.

Load Training Data

Download and unzip the air compressor data set [1] on page 1-787. This data set consists of recordings from air compressors in a healthy state or one of seven faulty states.

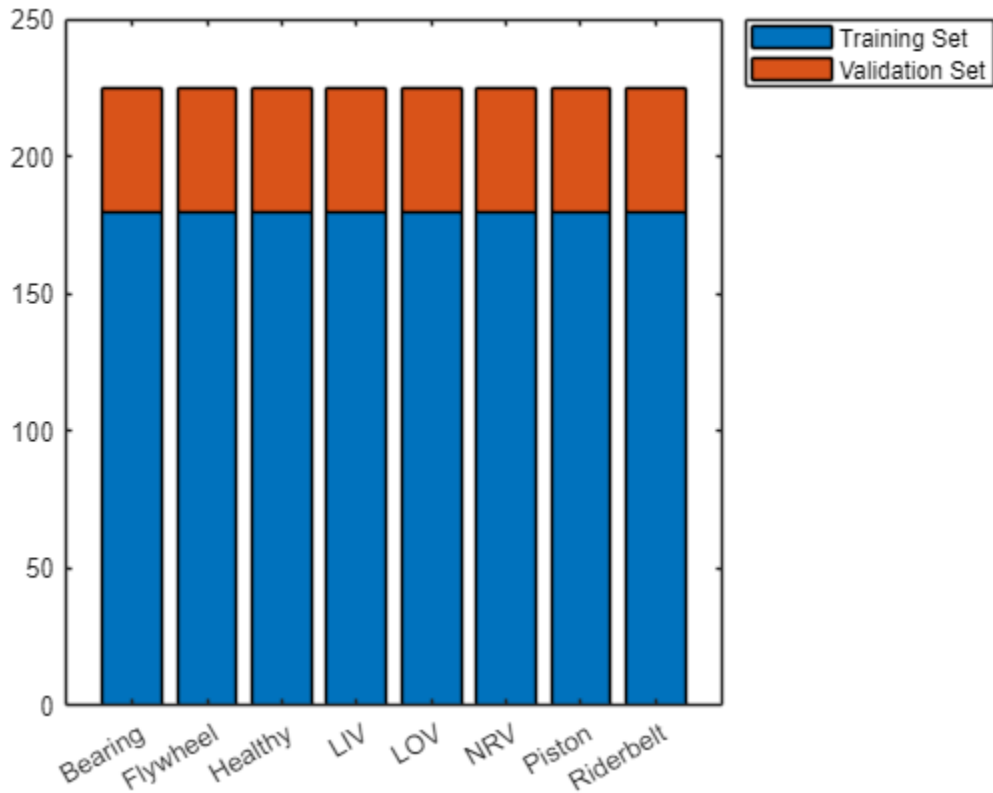
```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "AirCompressorDataset/AirC  
dataFolder = tempdir;  
unzip(downloadFolder, dataFolder)  
dataset = fullfile(dataFolder, "AirCompressorDataset");
```

Create an `audioDatastore` object to manage the data and split it into training and validation sets.

```
ads = audioDatastore(dataset, IncludeSubfolders=true, LabelSource="foldernames");  
rng default  
[adsTrain, adsValidation] = splitEachLabel(ads, 0.8);
```

Visualize the number of files in the training and validation sets.

```
uniqueLabels = unique(adsTrain.Labels);  
tblTrain = countEachLabel(adsTrain);  
tblValidation = countEachLabel(adsValidation);  
H = bar(uniqueLabels, [tblTrain.Count, tblValidation.Count], "stacked");  
legend(H, ["Training Set", "Validation Set"], Location="NorthEastOutside")
```



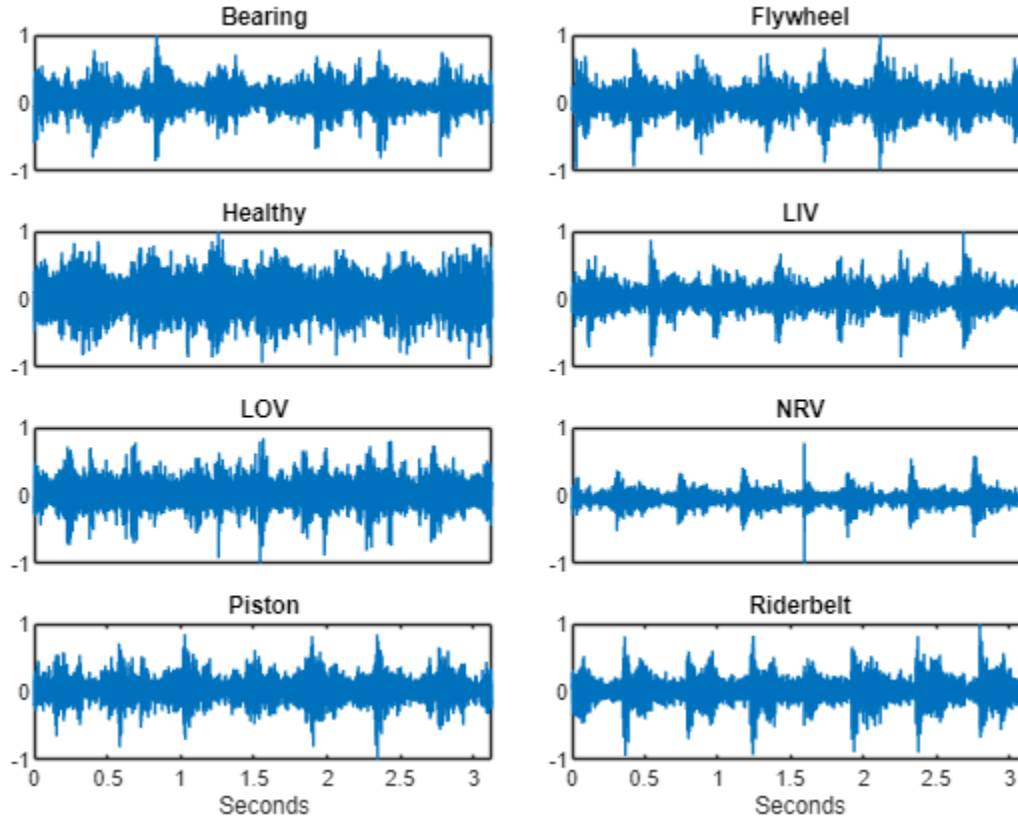
Select random examples from the training set for plotting. Each recording has 50,000 samples sampled at 16 kHz.

```

t = (0:5e4-1)/16e3;
tiledlayout(4,2,TileSpacing="compact",Padding="compact")
for n = 1:numel(uniqueLabels)
    idx = find(adsTrain.Labels==uniqueLabels(n));
    [x,fs] = audioread(adsTrain.Files{idx(randperm(numel(idx),1))});

    nexttile
    plotHandle = plot(t,x);
    if n == 7 || n == 8
        xlabel("Seconds");
    else
        set(gca,xtick=[])
    end
    title(string(uniqueLabels(n)));
end

```



Preprocess Data on CPU and GPU

In this example, you perform feature extraction and data augmentation while training the network. In this section, you define the feature extraction and augmentation pipeline and compare the speed of the pipeline executed on a CPU against the speed of the pipeline executed on a GPU. The output of this pipeline is the input to the CNN you train.

Create an `audioFeatureExtractor` object to extract mel spectrums using 200 ms mel windows with a 5 ms hop. The output from `extract` is a `numHops-by-128-by-1` array.

```
afe = audioFeatureExtractor(SampleRate=fs, ...
    FFTLength=4096, ...
    Window=hann(round(fs*0.2),"periodic"), ...
    OverlapLength=round(fs*0.195), ...
    melSpectrum=true);
setExtractorParameters(afe,"melSpectrum",NumBands=128);

featureVector = extract(afe,x);
[numHops,numFeatures,numChannels] = size(featureVector)

numHops = 586
numFeatures = 128
numChannels = 1
```

Deep learning methods are data-hungry, and the training dataset in this example is relatively small. Use the mixup [2] on page 1-787 augmentation technique to effectively enlarge the training set. In

mixup, you merge the features extracted from two audio signals as a weighted sum. The two signals have different labels, and the label assigned to the merged feature matrix is probabilistically assigned based on the mixing coefficient. The mixup augmentation is implemented in the supporting object, `Mixup` on page 1-786.

Create the pipeline to perform the following steps:

- 1 Extract the log-mel spectrogram.
- 2 Apply mixup to the feature matrices. The `Mixup` supporting object outputs a cell array containing the features and the label.

Create two versions of the pipeline for comparison: one that executes the pipeline on your CPU, and one that converts the raw audio signal to a `gpuArray` so that the pipeline is executed on your GPU.

```
offset = eps;

adsTrainCPU = transform(adsTrain,@(x)log10(extract(afe,x)+offset));
mixerCPU = Mixup(adsTrainCPU);
adsTrainCPU = transform(adsTrainCPU,@(x,info)mix(mixerCPU,x,info),IncludeInfo=true);

adsTrainGPU = transform(adsTrain,@gpuArray);
adsTrainGPU = transform(adsTrainGPU,@(x)log10(extract(afe,x)+offset));
mixerGPU = Mixup(adsTrainGPU);
adsTrainGPU = transform(adsTrainGPU,@(x,info)mix(mixerGPU,x,info),IncludeInfo=true);
```

For the validation set, apply the feature extraction pipeline but not the augmentation. Because you are not applying mixup, create a combined datastore to output a cell array containing the features and the label. Again, create one validation pipeline that executes on your GPU and one validation pipeline that executes on your CPU.

```
adsValidationGPU = transform(adsValidation,@gpuArray);
adsValidationGPU = transform(adsValidationGPU,@(x){log10(extract(afe,x)+offset)});
adsValidationGPU = combine(adsValidationGPU,arrayDatastore(adsValidation.Labels));

adsValidationCPU = transform(adsValidation,@(x){log10(extract(afe,x)+offset)});
adsValidationCPU = combine(adsValidationCPU,arrayDatastore(adsValidation.Labels));
```

Compare the time it takes for the CPU and a single GPU to extract features and perform data augmentation.

```
tic
for ii = 1:numel(adsTrain.Files)
    x = read(adsTrainCPU);
end
cpuPipeline = toc;
reset(adsTrainCPU)

tic
for ii = 1:numel(adsTrain.Files)
    x = read(adsTrainGPU);
end
wait(gpuDevice) % Ensure all calculations are completed
gpuPipeline = toc;
reset(adsTrainGPU)

disp(['Read, extract, and augment train set (CPU): '+cpuPipeline+' seconds'; ...
```

```

"Read, extract, and augment train set (GPU): "+gpuPipeline+" seconds"; ...
"Speedup (CPU time)/(GPU time): "+cpuPipeline/gpuPipeline]);

"Read, extract, and augment train set (CPU): 117.0887 seconds"
"Read, extract, and augment train set (GPU): 34.8972 seconds"
"Speedup (CPU time)/(GPU time): 3.3552"

```

Reading from the datastore contributes a significant amount of the overall time to the pipeline. A comparison of just extraction and augmentation shows an even greater speedup. Compare just feature extraction on the GPU versus on the CPU.

```

x = read(ads);

extract(afe,x); % Incur initialization cost outside timing loop
tic
for ii = 1:numel(adsTrain.Files)
    features = log10(extract(afe,x)+offset);
end
cpuFeatureExtraction = toc;

x = gpuArray(x); % Incur initialization cost outside timing loop
extract(afe,x);
tic
for ii = 1:numel(adsTrain.Files)
    features = log10(extract(afe,x)+offset);
end
wait(gpuDevice) % Ensure all calculations are completed
gpuFeatureExtraction = toc;

disp(["Extract features from train set (CPU): "+cpuFeatureExtraction+" seconds"; ...
"Extract features from train set (GPU): "+gpuFeatureExtraction+" seconds"; ...
"Speedup (CPU time)/(GPU time): "+cpuFeatureExtraction/gpuFeatureExtraction]);

"Extract features from train set (CPU): 52.7254 seconds"
"Extract features from train set (GPU): 1.2611 seconds"
"Speedup (CPU time)/(GPU time): 41.8096"

```

Define Network

Define a convolutional neural network that takes the augmented mel spectrogram as input. This network applies a single convolutional layer consisting of 48 filters with 3-by-3 kernels, followed by a batch normalization layer and a ReLU activation layer. The time dimension is then collapsed using a max pooling layer. Finally, the output of the pooling layer is reduced using a fully connected layer followed by softmax and classification layers. See “List of Deep Learning Layers” (Deep Learning Toolbox) for more information.

```

numClasses = numel(categories(adsTrain.Labels));
imageSize = [numHops,afe.FeatureVectorLength];

layers = [
    imageInputLayer(imageSize,Normalization="none")

    convolution2dLayer(3,48,Padding="same")
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer([numHops,1])

```

```
fullyConnectedLayer(numClasses)
softmaxLayer
classificationLayer(Classes=categories(adsTrain.Labels));
];
```

To define the training options, use `trainingOptions` (Deep Learning Toolbox). Set the `ExecutionEnvironment` to `multi-gpu` to leverage multiple GPUs, if available. Otherwise, you can set `ExecutionEnvironment` to `gpu`. The computer used in this example has access to four Titan V GPU devices. In this example, the network training always leverages GPUs.

```
miniBatchSize = 128;
options = trainingOptions("adam", ...
    Shuffle="every-epoch", ...
    MaxEpochs=40, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropPeriod=15, ...
    LearnRateDropFactor=0.2, ...
    MiniBatchSize=miniBatchSize, ...
    Plots="training-progress", ...
    Verbose=false, ...
    ValidationData=adsValidationCPU, ...
    ValidationFrequency=ceil(numel(adsTrain.Files)/miniBatchSize), ...
    ExecutionEnvironment="multi-gpu");
```

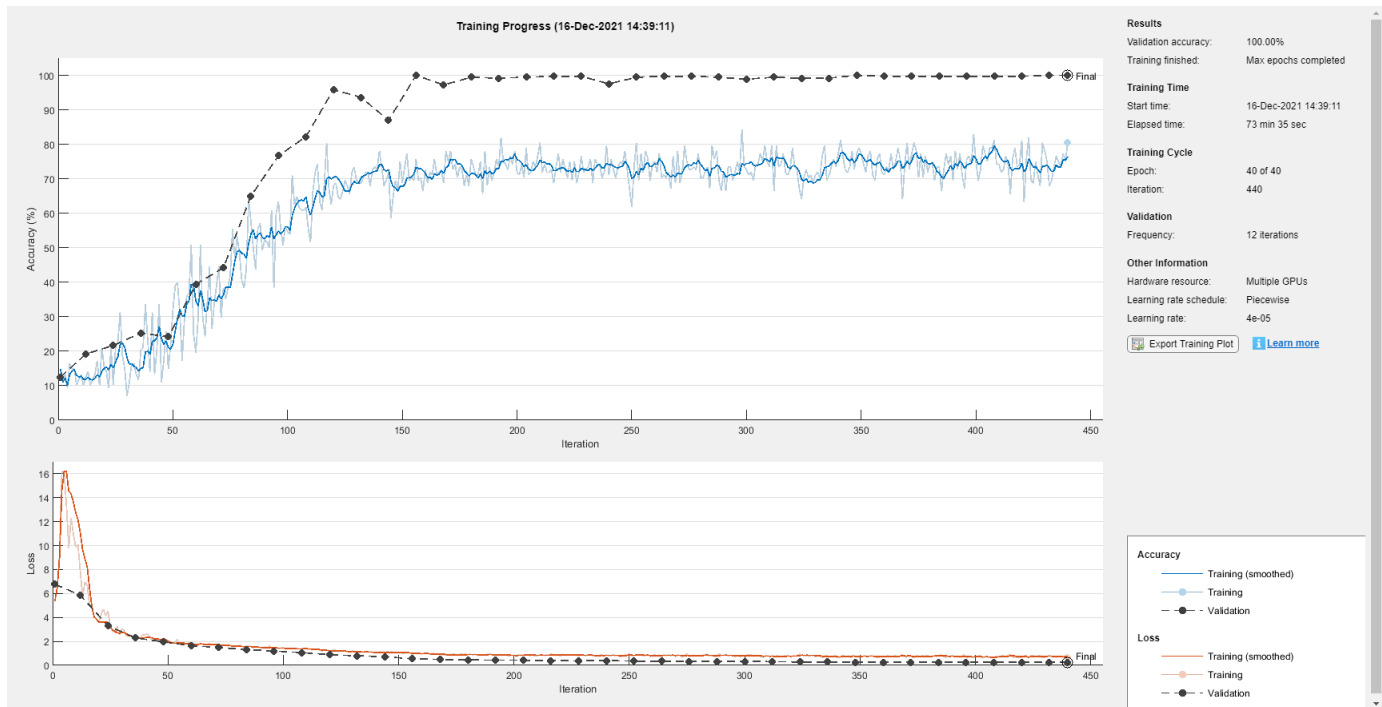
Train Network

Train Network Using CPU-Based Preprocessing

Call `trainNetwork` (Deep Learning Toolbox) to train the network using your CPU for the feature extraction pipeline. The execution environment for the network training is your GPU(s).

```
tic
net = trainNetwork(adsTrainCPU, layers, options);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 4).
```

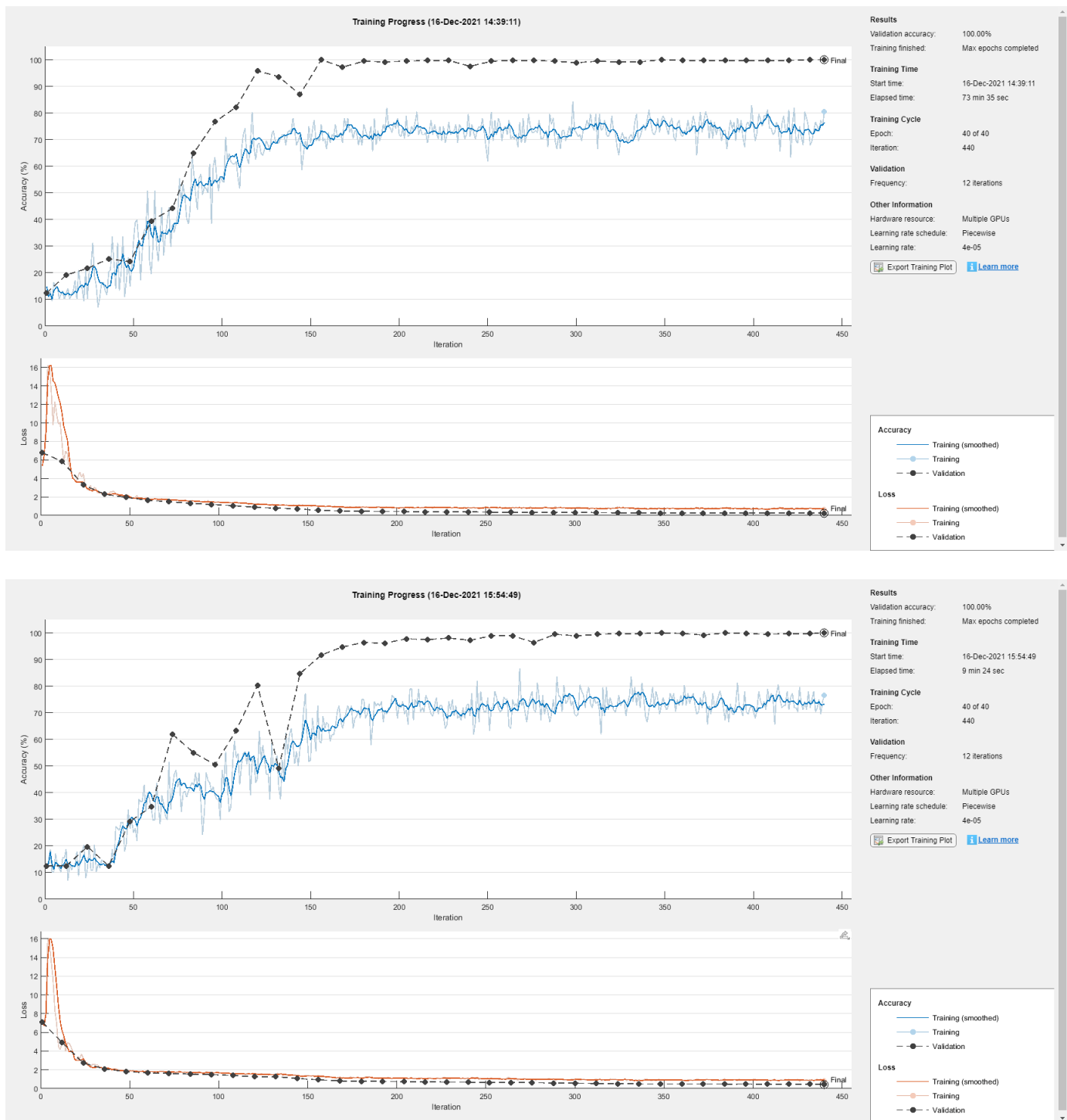



```
cpuTrainTime = toc;
```

Train Network Using GPU-Based Preprocessing

Replace the validation data in the training options with the GPU-based pipeline. Train the network using your GPU(s) for the feature extraction pipeline. The execution environment for the network training is your GPU(s).

```
options.ValidationData = adsValidationGPU;
tic
net = trainNetwork(adsTrainGPU, layers, options);
```



```
gpuTrainTime = toc;
```

Compare CPU- and GPU-based Preprocessing

Print the timing results for training using a CPU for feature extraction and augmentation, and training using GPU(s) for feature extraction and augmentation.

```

disp(["Training time (CPU): "+cpuTrainTime+" seconds";
      "Training time (GPU): "+gpuTrainTime+" seconds";
      "Speedup (CPU time)/(GPU time): "+cpuTrainTime/gpuTrainTime])

      "Training time (CPU): 4650.3639 seconds"
      "Training time (GPU): 599.1963 seconds"
      "Speedup (CPU time)/(GPU time): 7.761"

```

Compare CPU and GPU Inference Performance

Compare the time it takes to perform prediction on a single 3-second clip when feature extraction is performed on the GPU versus the CPU. In both cases, the network prediction happens on your GPU.

```

signalToClassify = read(ads);

gpuFeatureExtraction = gputimeit(@()predict(net,log10(extract(afe,gpuArray(signalToClassify))+offset)));
cpuFeatureExtraction = gputimeit(@()predict(net,log10(extract(afe,(signalToClassify))+offset)));

disp(["Prediction time for 3 s of data (feature extraction on CPU): "+cpuFeatureExtraction*1e3+" ms";
      "Prediction time for 3 s of data (feature extraction on GPU): "+gpuFeatureExtraction*1e3+" ms";
      "Speedup (CPU time)/(GPU time): "+cpuFeatureExtraction/gpuFeatureExtraction])

      "Prediction time for 3 s of data (feature extraction on CPU): 42.8014 ms"
      "Prediction time for 3 s of data (feature extraction on GPU): 4.0693 ms"
      "Speedup (CPU time)/(GPU time): 10.5182"

```

Compare the time it takes to perform prediction on a set of 3-second clips when feature extraction is performed on the GPU(s) versus the CPU. In both cases, the network prediction happens on your GPU(s).

```

adsValidationGPU = transform(adsValidation,@(x)gpuArray(x));
adsValidationGPU = transform(adsValidationGPU,@(x){log10(extract(afe,x)+offset)});
adsValidationCPU = transform(adsValidation,@(x){log10(extract(afe,x)+offset)});

gpuFeatureExtraction = gputimeit(@()predict(net,adsValidationGPU,ExecutionEnvironment="multi-gpu"));
cpuFeatureExtraction = gputimeit(@()predict(net,adsValidationCPU,ExecutionEnvironment="multi-gpu"));

disp(["Prediction time for validation set (feature extraction on CPU): "+cpuFeatureExtraction+" seconds";
      "Prediction time for validation set (feature extraction on GPU): "+gpuFeatureExtraction+" seconds";
      "Speedup (CPU time)/(GPU time): "+cpuFeatureExtraction/gpuFeatureExtraction])

      "Prediction time for validation set (feature extraction on CPU): 36.2089 seconds"
      "Prediction time for validation set (feature extraction on GPU): 4.1345 seconds"
      "Speedup (CPU time)/(GPU time): 8.7578"

```

Conclusion

It is well known that you can decrease the time it takes to train a network by leveraging GPU devices. This enables you to more quickly iterate and develop your final system. In many training setups, you can achieve additional performance gains by leveraging GPU devices for feature extraction and data augmentation. This example shows a significant decrease in the overall time it takes to train a CNN when leveraging GPU devices for feature extraction and data augmentation. Additionally, leveraging GPU devices for feature extraction at inference time, for both single-observations and data sets, achieves significant performance gains.

Supporting Functions

Mixup

The supporting object, `Mixup`, is placed in your current folder when you open this example.

type `Mixup`

```
classdef Mixup < handle
    %MIXUP Mixup data augmentation
    % mixer = Mixup(augDatastore) creates an object that can mix features
    % at a randomly set ratio and then probabilistically set the output
    % label as one of the two original signals.
    %
    % Mixup Properties:
    % MixProbability - Mix probability
    % AugDatastore - Augmentation datastore
    %
    % Mixup Methods:
    % mix - Apply mixup
    %
    % Copyright 2021 The MathWorks, Inc.

    properties (SetAccess=public,GetAccess=public)
        %MixProbability Mix probability
        % Specify the probability that mixing is applied as a scalar in the
        % range [0,1]. If unspecified, MixProbability defaults to 1/3.
        MixProbability (1,1) {mustBeNumeric} = 1/3;
    end
    properties (SetAccess=immutable,GetAccess=public)
        %AUGDATASTORE Augmentation datastore
        % Specify a datastore from which to get the mixing signals. The
        % datastore must contain a label in the info returned from reading.
        % This property is immutable, meaning it cannot be changed after
        % construction.
        AugDatastore
    end

    methods
        function obj = Mixup(augDatastore)
            obj.AugDatastore = augDatastore;
        end

        function [dataOut,infoOut] = mix(obj,x,infoIn)
            %MIX Apply mixup
            % [dataOut,infoOut] = mix(mixer,x,infoIn) probabilistically mix
            % the input, x, and its associated label contained in infoIn
            % with a signal randomly drawn from the augmentation datastore.
            % The output, dataOut, is a cell array with two columns. The
            % first column contains the features and the second column
            % contains the label.

            if rand > obj.MixProbability % Only mix ~1/3 the dataset

                % Randomly set mixing coefficient. Draw from a normal
                % distribution with mean 0.5 and contained within [0,1].
                lambda = max(min((randn./10)+0.5,1),0);
            end
        end
    end
end
```

```
% Read one file from the augmentation datastore.
subDS = subset(obj.AugDatastore,randi([1,numel(obj.AugDatastore.UnderlyingDatastore.Files),1]));
[y,yInfo] = read(subDS);

% Mix the features element-by-element according to lambda.
dataOut = lambda*x + (1-lambda)*y;

% Set the output label probabilistically based on the mixing coefficient.
if lambda < rand
    labelOut = yInfo.Label;
    infoOut.Label = labelOut;
else
    labelOut = infoIn.Label;
end
infoOut.Label = labelOut;

% Combine the output data and labels.
dataOut = [{dataOut},{labelOut}];

else % Do not apply mixing

    dataOut = [{x},{infoIn.Label}];
    infoOut = infoIn;

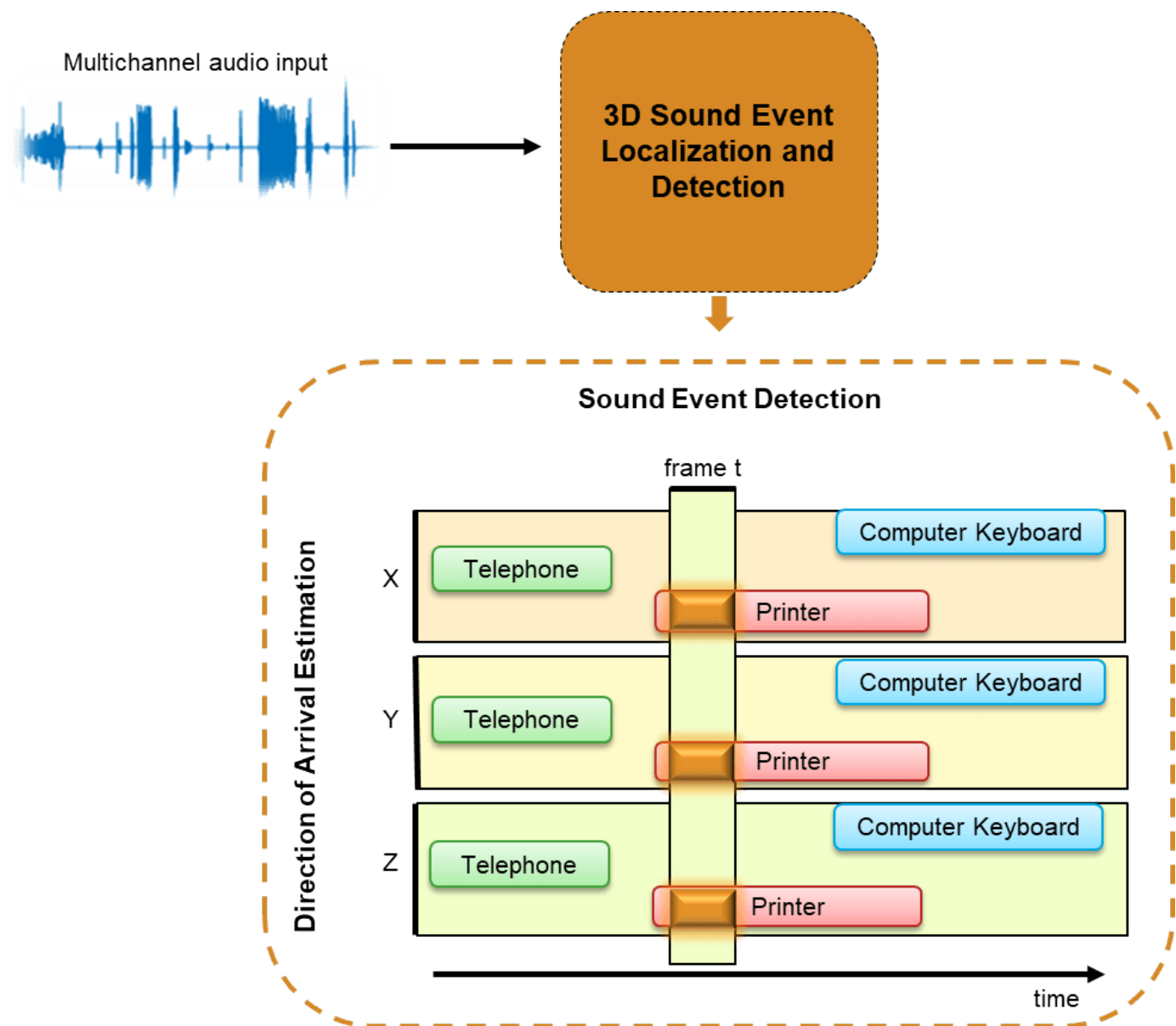
end
end
end
end
```

References

- [1] Verma, Nishchal K., et al. "Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors." *IEEE Transactions on Reliability*, vol. 65, no. 1, Mar. 2016, pp. 291-309. DOI.org (Crossref), doi:10.1109/TR.2015.2459684.
- [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/>.

Train 3-D Sound Event Localization and Detection (SELD) Using Deep Learning

In this example, you train a deep learning model to perform sound localization and event detection from ambisonic data. The model consists of two independently trained convolutional recurrent neural networks (CRNN) [1] on page 1-804: one for sound event detection (SED), and one for direction of arrival (DOA) estimation. To explore the models trained in this example, see “3-D Sound Event Localization and Detection Using Trained Recurrent Convolutional Neural Network” on page 1-815.



Introduction

Ambisonics is a popular 3-D sound format that has shown promise in tasks like sound source localization, speech enhancement, and source separation. Ambisonics is a full sphere surround sound format that contains a speaker-independent sound field representation (B-format). First order B-format ambisonic recordings contain components that correspond to the sound pressure captured by an omnidirectional microphone (W) and sound pressure gradients X , Y , and Z that correspond to front/back, left/right, and up/down captured by figure-of-eight capsules oriented along the three spatial axes. 3-D SELD has applications in virtual reality, robotics, smart homes, and defense.

You will train two separate models for the sound event detection task and the localization task. Both models are based on the convolutional recurrent neural network architecture described in [1] on page 1-804. The sound event detection task is formulated as a classification task. The sound event localization task estimates Cartesian coordinates of the sound source and is formulated as a regression task. You use the L3DAS21 data set [2] on page 1-805 to train and validate the networks. To explore the models trained in this example, see “3-D Sound Event Localization and Detection Using Trained Recurrent Convolutional Neural Network” on page 1-815.

Download and Prepare Data

This example uses a subset of the L3DAS21 Task 2 challenge data set [2] on page 1-805. The data set contains multiple-source and multiple-perspective (MSMP) B-format ambisonic audio recordings collected at a sampling rate of 32 kHz. The train and validation splits are provided with the data set. Each recording is one minute long and contains a simulated 3-D audio environment in which up to 3 simultaneous acoustic events may be active at the same time. In this example, you only use the data that contains non-overlapping sounds. The sound events belong to 14 sound classes. The labels are provided as csv files that contain the sound class, the Cartesian coordinates of the sound source, and the onset and offset time stamps.

Download the dataset.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "L3DAS21_ov1.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
dataset = fullfile(dataFolder, "L3DAS21_ov1");
```

Optionally Reduce Data Set

To train the networks with the entire data set and achieve a reasonable performance, set `speedupExample` to `false`. To run this example quickly, set `speedupExample` to `true`.

```
speedupExample =  ;
```

Create Datastores

Create `audioDatastore` objects to ingest the data. Each data point in the data set consists of two B-format ambisonic recordings that correspond to the two microphones (A and B). For each data folder (train and validation), use `subset` to create two subsets corresponding to the two microphones.

```
adsTrain = audioDatastore(fullfile(dataset, "train", "data"));
adsTrainA = subset(adsTrain, cellfun(@(c) endsWith(c, "A.wav"), adsTrain.Files));
adsTrainB = subset(adsTrain, cellfun(@(c) endsWith(c, "B.wav"), adsTrain.Files));

adsValidation = audioDatastore(fullfile(dataset, "validation", "data"));
```

```
adsValidationA = subset(adsValidation,cellfun(@(c)endsWith(c,"A.wav"),adsValidation.Files));  
adsValidationB = subset(adsValidation,cellfun(@(c)endsWith(c,"B.wav"),adsValidation.Files));
```

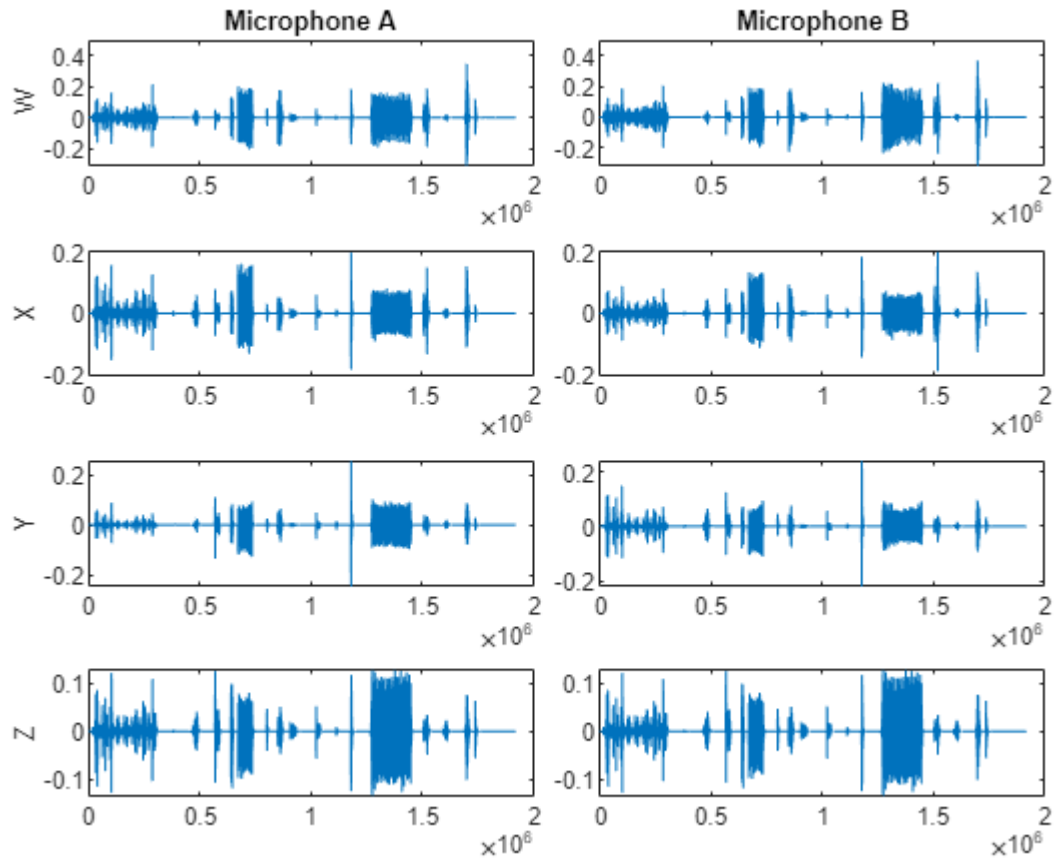
Reduce the data set if requested.

```
if speedupExample  
    adsTrainA = subset(adsTrainA,1:2);  
    adsTrainB = subset(adsTrainB,1:2);  
end
```

Inspect Data

Preview the ambisonic recordings and plot the data.

```
micA = preview(adsTrainA);  
micB = preview(adsTrainB);  
  
tiledlayout(4,2,TileSpacing="tight")  
  
nexttile  
plot(micA(:,1))  
title("Microphone A")  
ylabel("W")  
  
nexttile  
plot(micB(:,1))  
title("Microphone B")  
  
nexttile  
plot(micA(:,2))  
ylabel("X")  
  
nexttile  
plot(micB(:,2))  
  
nexttile  
plot(micA(:,3))  
ylabel("Y")  
  
nexttile  
plot(micB(:,3))  
  
nexttile  
plot(micB(:,4))  
ylabel("Z")  
  
nexttile  
plot(micB(:,4))
```

Listen to a section of the data.

```
microphone =  ;
channel =  ;
duration = 10  ;
fs = 32e3; % Known sampling rate of data.
```

```
s = [micA,micB];
data = s(1:round(duration*fs),channel + (microphone-1)*4);
sound(data,fs)
```

Create Targets

Each data point in the data set has a corresponding CSV file containing the sound event class, the start and end times of the sound, and the location of the sound. Create a container to map between the sound classes and integers.

```
keySet = ["Chink_and_clink","Computer_keyboard","Cupboard_open_or_close","Drawer_open_or_close",
"Female_speech_and_woman_speaking","Finger_snapping","Keys_jangling","Knock","Laughter", ...
"Male_speech_and_man_speaking","Printer","Scissors","Telephone","Writing"];
valueSet = {1,2,3,4,5,6,7,8,9,10,11,12,13,14};
params.SoundClasses = containers.Map(keySet,valueSet);
```

Create a `tabularTextDatastore` to ingest the train file labels. Make sure the label files are in the same order as the data files. Preview a label file from the datastore.

```
[folder,fn] = fileparts(adsTrainA.Files);
targetPath = fullfile(strrep(folder,filesep+"data",filesep+"labels"),"label_" + strrep(fn,"_A",""));
ttdsTrain = tabularTextDatastore(targetPath);
```

```
labelTable = preview(ttdsTrain)
```

```
labelTable=8x7 table
    File      Start      End      Class      X      Y      Z
    _____  _____  _____  _____  _____  _____  _____
    0          0.54784    9.6651    {'Writing'   }    0.5    -1.5    0.3
    0          11.521     12.534    {'Finger_snapping' }    0.75   1.25   -1
    0          14.255     16.064    {'Keys_jangling' }    0.5    -1.5    0.3
    0          17.728     18.878    {'Chink_and_clink' }    0.5    1      0
    0          19.95      20.4      {'Printer'   }    -1.5   -1.5   -0.6
    0          20.994     23.477    {'Cupboard_open_or_close' }    -0.5   0.75   0
    0          25.032     25.723    {'Chink_and_clink' }    -2     -0.5   -0.3
    0          26.547     27.491    {'Female_speech_and_woman_speaking' }    1     -1.5   0
```

The labels in the dataset are provided with time stamps in seconds. To create targets and train a network, you need to map the time stamps to frames. The total duration of each file is 60 seconds. You will divide each file into 600 frames for the target, meaning the model will make a prediction every 0.1 seconds.

```
params.Targets.TotalDuration = 60;
params.Targets.NumFrames = 600;
```

SED Targets

The supporting function, `extractSEDTTargets` on page 1-805, uses the label data to create an SED target. The target is a one-hot encoded matrix of size `numframes-by-numclasses`. Frames with no sounds present are encoded as all-zero vectors.

```
SEDTTargets = extractSEDTTargets(labelTable,params);
```

```
[numframes,numclasses] = size(SEDTTargets{1})
```

```
numframes = 600
```

```
numclasses = 14
```

Extract SED targets from the train and validation sets.

```
dsTTrain = transform(ttdsTrain,@(x)extractSEDTTargets(x,params));
sedTTrain = readall(dsTTrain);
```

```
[folder,fn] = fileparts(adsValidationA.Files);
targetPath = fullfile(strrep(folder,filesep+"data",filesep+"labels"),"label_" + strrep(fn,"_A",""));
```

```
ttdsValidation = tabularTextDatastore(targetPath);
dsTValidation = transform(ttdsValidation,@(x)extractSEDTTargets(x,params));
sedTValidation = readall(dsTValidation);
```

DOA Targets

The supporting function, `extractDOATargets` on page 1-805, uses the label data to create a DOA target. The target is a matrix of size `numframes-by-numaxis`. The axis values correspond to the sound source location in 3-D space. Frames with no sounds present are encoded as all-zero vectors.

First, define a parameter to scale the target axis values so that they are between -1 and 1. This scaling is necessary because the DOA network you define later uses `tanh` activation as its final layer.

```
params.DOA.ScaleFactor = 2;
DOATargets = extractDOATargets(labelTable,params);
```

```
[numframes,numaxis] = size(DOATargets{1})
```

```
numframes = 600
```

```
numaxis = 3
```

Extract DOA targets from the train and validation sets.

```
dsTTrain = transform(ttdsTrain,@(x)extractDOATargets(x,params));
doaTTrain = readall(dsTTrain);
```

```
[folder,fn] = fileparts(adsValidationA.Files);
```

```
targetPath = fullfile(strrep(folder,filesep+"data",filesep+"labels"),"label_" + strrep(fn,"_A","_B"));
```

```
ttdsValidation = tabularTextDatastore(targetPath);
```

```
dsTValidation = transform(ttdsValidation,@(x)extractDOATargets(x,params));
```

```
doaTValidation = readall(dsTValidation);
```

Sound Event Detection (SED)

Feature Extraction

The sound event detection model uses log-magnitude short-time Fourier transforms (STFT) as predictors to the system. Specify a 512-point periodic Hamming window and a hop length of 400 samples.

```
params.SED.SampleRate = 32e3;
params.SED.HopLength = 400;
params.SED.Window = hamming(512,"periodic");
```

The supporting function, `extractSTFT` on page 1-806, takes a cell array of microphone readings and extracts the half-sided centered log-magnitude STFTs. The STFT features corresponding to both microphones are stacked along the third dimension.

```
stftFeats = extractSTFT({micA,micB},params);
```

```
[numfeaturesSED,numframesSED,numchannelsSED] = size(stftFeats)
```

```
numfeaturesSED = 256
```

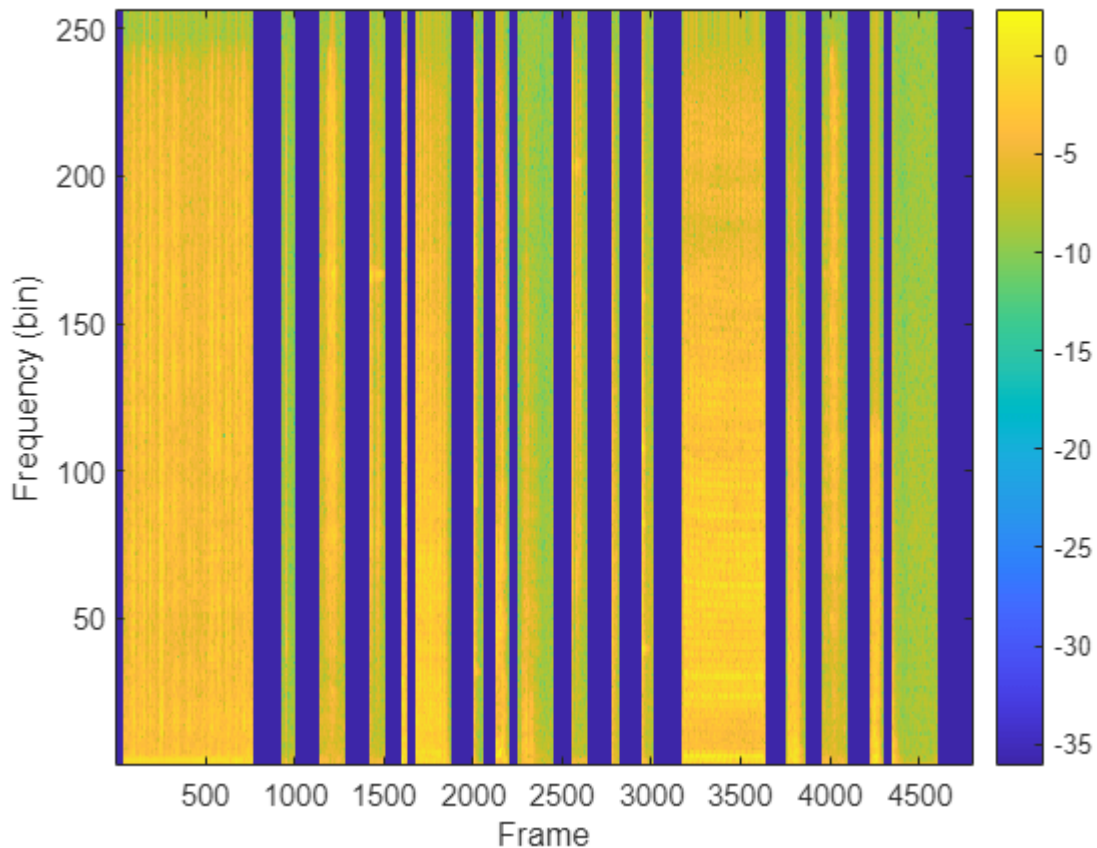
```
numframesSED = 4800
```

```
numchannelsSED = 8
```

Plot the STFT features of one channel.

```
channel =  ;
```

```
figure
imagesc(stftFeats(:,:,channel))
colorbar
xlabel("Frame")
ylabel("Frequency (bin)")
set(gca,YDir="normal")
```



Extract features from the entire train and validation sets. First, combine the datastores corresponding to microphones A and B. Then, define a transform on the datastore so that reading from it returns the STFT. If you have Parallel Computing Toolbox™, you can speed up processing using the `UseParallel` flag of `readall`.

```
pFlag = ~isempty(ver("parallel")) && ~speedupExample;

trainDS = combine(adsTrainA,adsTrainB);
trainDS_T = transform(trainDS,@(x){extractSTFT(x,params)},IncludeInfo=false);
XTrain = readall(trainDS_T,UseParallel=pFlag);
valDS = combine(adsValidationA,adsValidationB);
valDS_T = transform(valDS,@(x){extractSTFT(x,params)},IncludeInfo=false);
XValidation = readall(valDS_T,UseParallel=pFlag);
```

Combine the predictor arrays with the previously computed SED target arrays.

```
trainSedDS = combine(arrayDatastore(XTrain,OutputType="same"),arrayDatastore(sedTTrain,OutputType="same"));
valSedDS = combine(arrayDatastore(XValidation,OutputType="same"),arrayDatastore(sedTValidation,OutputType="same"));
```

Training Options

Define training parameters for Adam optimization.

```
trainOptionsSED = struct( ...
    MaxEpochs=300, ...
    MiniBatchSize=4, ...
    InitialLearnRate=1e-5, ...
    GradientDecayFactor=0.01, ...
    SquaredGradientDecayFactor=0.0, ...
    ValidationPatience=25, ...
    LearnRateDropPeriod=100, ...
    LearnRateDropFactor=1);
```

```
if speedupExample
    trainOptionsSED.MaxEpochs = 1;
end
```

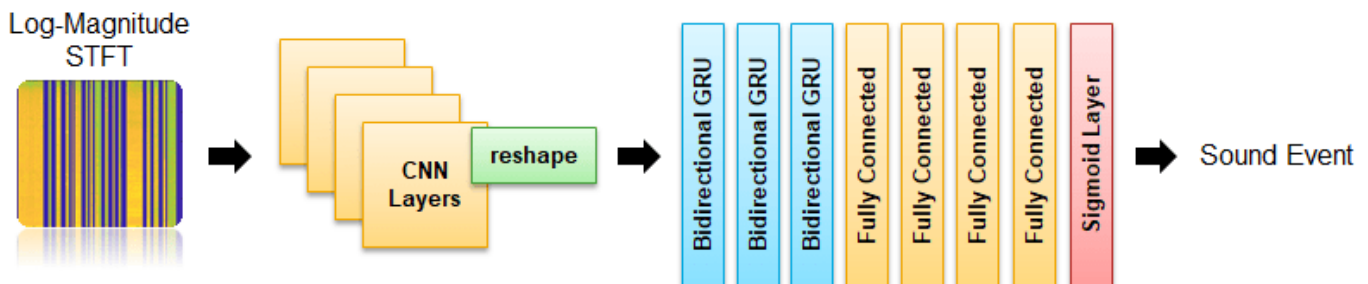
Create minibatchqueue (Deep Learning Toolbox) objects to read mini-batches from the train and validation datastores.

```
trainSEDbmq = minibatchqueue(trainSedDS, ...
    MiniBatchSize=trainOptionsSED.MiniBatchSize, ...
    OutputAsDlarray=[1,1], ...
    MiniBatchFormat=["SSCB","TCB"], ...
    OutputEnvironment=["auto","auto"]);
```

```
validationSEDbmq = minibatchqueue(valSedDS, ...
    MiniBatchSize=trainOptionsSED.MiniBatchSize, ...
    OutputAsDlarray=[1,1], ...
    MiniBatchFormat=["SSCB","TCB"], ...
    OutputEnvironment=["auto","auto"]);
```

Define Sound Event Detection (SED) Network

The network is implemented in two stages - Convolutional Neural Network (CNN) and Gated Recurrent Network (GRU). You will use a custom reshaping layer to recast the output of the CNN model into a sequence and pass that as the input to the RNN model. The custom reshaping layer is placed in your current folder when you open this example. The final output layer uses sigmoid activation.



Define the CNN layers for the SED model.

```
seldnetCNNLayers = [
    imageInputLayer([numfeaturesSED,numframesSED,numchannelsSED],Normalization="none",Name="input
```

```
convolution2dLayer([3,3],64,Padding="same",Name="conv1")
batchNormalizationLayer(Name="batchnorm1")
reluLayer(Name="relu1")
maxPooling2dLayer([8,2],Stride=[8,2],Padding="same",Name="maxpool1")

convolution2dLayer([3,3],128,Padding="same",Name="conv2")
batchNormalizationLayer(Name="batchnorm2")
reluLayer(Name="relu2")
maxPooling2dLayer([8,2],Stride=[8,2],Padding="same",Name="maxpool2")

convolution2dLayer([3,3],256,Padding="same",Name="conv3")
batchNormalizationLayer(Name="batchnorm3")
reluLayer(Name="relu3")
maxPooling2dLayer([2,2],Stride=[2,2],Padding="same",Name="maxpool3")

convolution2dLayer([3,3],512,Padding="same",Name="conv4")
batchNormalizationLayer(Name="batchnorm4")
reluLayer(Name="relu4")
maxPooling2dLayer([1,1],Stride=[1,1],Padding="same",Name="maxpool4")

reshapeLayer("reshape")
];
netCNN = dlnetwork(layerGraph(seldnetCNNLayers));
```

Define the RNN layers for the SED model.

```
seldnetGRULayers = [
    sequenceInputLayer(1024,Name="sequenceInputLayer")

    bigruLayer(1024,256,Name="gru1")
    bigruLayer(512,256,Name="gru2")
    bigruLayer(512,256,Name="gru3")

    fullyConnectedLayer(1024,Name="fc1")
    reluLayer(Name="relu1")
    fullyConnectedLayer(1024,Name="fc2")
    reluLayer(Name="relu2")
    fullyConnectedLayer(1024,Name="fc3")
    reluLayer(Name="relu3")

    fullyConnectedLayer(params.SoundClasses.Count,Name="fc4")
    sigmoidLayer(Name="output")
];

netRNN = dlnetwork(layerGraph(seldnetGRULayers));
```

Create a struct to contain both the CNN and RNN sections of the full model.

```
sedModel.CNN = netCNN;
sedModel.RNN = netRNN;
```

Train SED Network

Initialize variables to track the progress of the training.

```
iteration = 0;
averageGrad = [];
averageSqGrad = [];
```

```
epoch = 0;
bestLoss = Inf;
badEpochs = 0;
learnRate = trainOptionsSED.InitialLearnRate;
```

To display training progress, initialize the supporting object `progressPlotterSELD`. The supporting object, `progressPlotterSELD`, is placed in your current folder when you open this example.

```
pp = progressPlotterSELD();
```

Run the training loop.

```
rng(0)
while epoch < trainOptionsSED.MaxEpochs && badEpochs < trainOptionsSED.ValidationPatience
    epoch = epoch + 1;

    % Shuffle mini-batch queue.
    shuffle(trainSEDMbq)

    while hasdata(trainSEDMbq)

        % Update iteration counter.
        iteration = iteration + 1;

        % Read mini-batch of data.
        [X,T] = next(trainSEDMbq);

        % Evaluate the model gradients and loss using dlfeval and the modelLoss function.
        [loss,grad,state] = dlfeval(@modelLoss,sedModel,X,T);
        loss = loss/size(T,2);

        % Update state.
        sedModel.CNN.State = state.CNN;
        sedModel.RNN.State = state.RNN;

        % Update the network parameters using the Adam optimizer.
        [sedModel,averageGrad,averageSqGrad] = adamupdate(sedModel,grad,averageGrad, ...
            averageSqGrad,iteration,learnRate,trainOptionsSED.GradientDecayFactor,trainOptionsSELD);

        % Update the training progress plot.
        updateTrainingProgress(pp,Epoch=epoch,LearnRate=learnRate,Iteration=iteration,Loss=loss)
    end

    % Perform validation after each epoch.
    loss = predictBatch(sedModel,validationSEDMbq);

    % Update the training progress plot with validation results.
    updateValidation(pp,Loss=loss,Iteration=iteration)

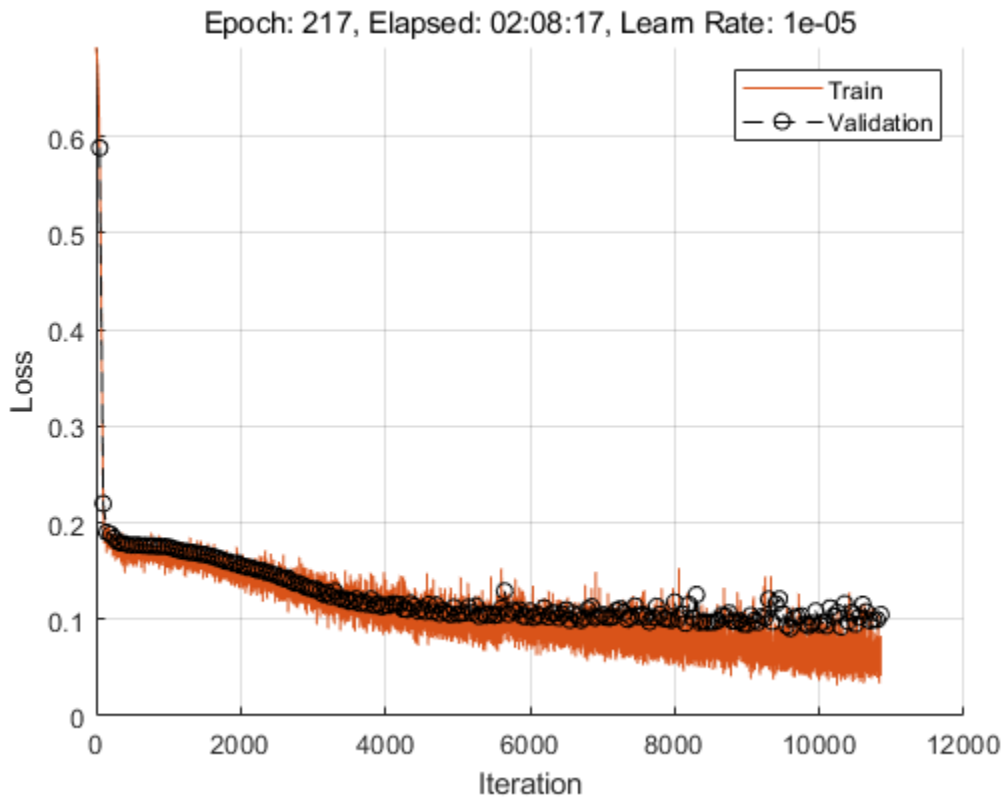
    % Create a checkpoint if the validation loss improved. If validation
    % loss did not improve, add to the number of bad epochs.
    if loss < bestLoss
        bestLoss = loss;
        badEpochs = 0;
        fileName = "SED-BestModel";
        save(fileName,"sedModel");
    else
```

```

        badEpochs = badEpochs + 1;
    end

    % Update learn rate
    if rem(epoch,trainOptionsSED.LearnRateDropPeriod)==0
        learnRate = learnRate*trainOptionsSED.LearnRateDropFactor;
    end
end
end

```



Direction of Arrival (DOA)

Feature Extraction

The direction of arrival estimation model uses generalized cross correlation phase transform (GCC-PHAT) as predictors to the system. Specify a 1024-point Hann window, a hop length of 400 samples, and the number of bands as 96.

```

params.DOA.SampleRate = 32e3;
params.DOA.Window = hann(1024);
params.DOA.NumBands = 96;
params.DOA.HopLength = 400;

```

Extract the GCC-PHAT features used as input predictors to the sound localization network. The GCC-PHAT algorithm measures the cross correlation between each pair of channels. The input signals have a total of 8 channels, so the output has a total of 28 measurements.

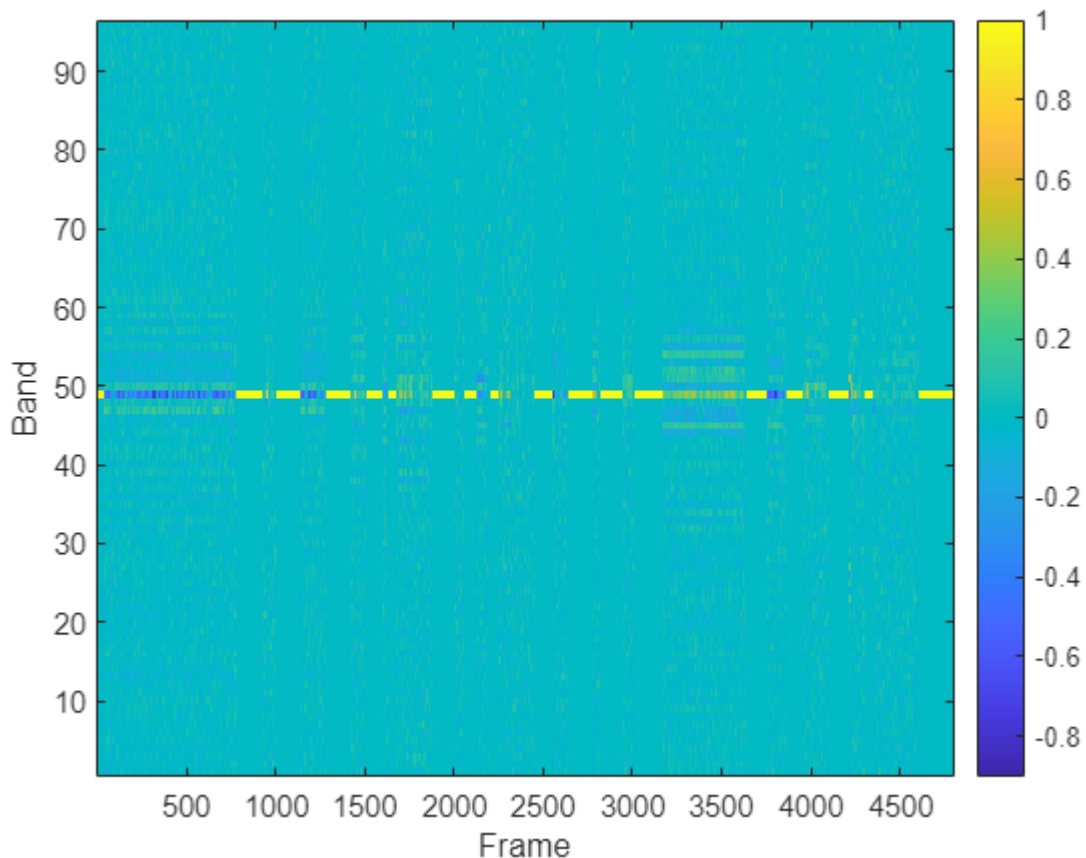

```
gccPhatFeats = extractGCCPHAT({micA,micB},params);
[numfeaturesDOA,timestepsDOA,numchannelsDOA] = size(gccPhatFeats)

numfeaturesDOA = 96
timestepsDOA = 4800
numchannelsDOA = 28
```

Plot the GCC-PHAT features of a channel pair.

```
channelpair = ;
```

```
figure
imagesc(gccPhatFeats(:,:,channelpair))
colorbar
xlabel("Frame")
ylabel("Band")
set(gca,YDir="normal")
```



Extract features from the entire train and validation sets. If you have Parallel Computing Toolbox™, you can speed up processing using the `UseParallel` flag of `readall`.

```
pFlag = ~isempty(ver("parallel")) && ~speedupExample;
```

```
trainDS = combine(adsTrainA,adsTrainB);
trainDS_T = transform(trainDS,@(x){extractGCCPHAT(x,params)},IncludeInfo=false);
XTrain = readall(trainDS_T,UseParallel=pFlag);
```

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

```
valDS = combine(adsValidationA,adsValidationB);
valDS_T = transform(valDS,@(x){extractGCCPHAT(x,params)},IncludeInfo=false);
XValidation = readall(valDS_T,UseParallel=pFlag);
```

Combine the predictor arrays with the previously compute DOA target arrays.

```
trainDOA = combine(arrayDatastore(XTrain,OutputType="same"),arrayDatastore(doaTTrain,OutputType="same"));
validationDOA = combine(arrayDatastore(XValidation,OutputType="same"),arrayDatastore(doaTValidation,OutputType="same"));
```

Training Options

Use the same train options you defined when training the SED network.

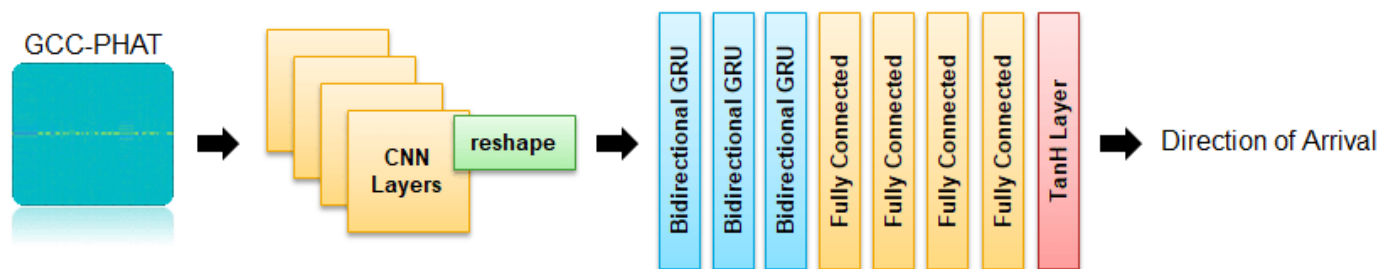
```
trainOptionsDOA = trainOptionsSED;
```

Create mini-batch queues for the train and validation sets.

```
trainDOAmbq = minibatchqueue(trainDOA, ...
    MiniBatchSize=trainOptionsDOA.MiniBatchSize, ...
    OutputAsDlarray=[1,1], ...
    MiniBatchFormat=["SSCB","TCB"], ...
    OutputEnvironment=["auto","auto"]);
validationDOAmbq = minibatchqueue(validationDOA, ...
    MiniBatchSize=trainOptionsDOA.MiniBatchSize, ...
    OutputAsDlarray=[1,1], ...
    MiniBatchFormat=["SSCB","TCB"], ...
    OutputEnvironment=["auto","auto"]);
```

Define Direction of Arrival (DOA) Network

The DOA network is very similar to the SED network defined earlier. The key differences are the size of the input layer and the final activation layer.



Update the SELDnet architecture used for the SED network for use with DOA estimation.

```
seldnetCNILayers(1) = imageInputLayer([numfeaturesDOA,timestepsDOA,numchannelsDOA],Normalization="none");
seldnetCNILayers(5) = maxPooling2dLayer([3,2],Stride=[3,2],Padding="same",Name="maxpool1");
netCNN = dlnetwork(layerGraph(seldnetCNILayers));
```

```
seldnetGRULayers(11) = fullyConnectedLayer(3,Name="fc4");
seldnetGRULayers(12) = tanhLayer(Name="output");
netRNN = dlnetwork(layerGraph(seldnetGRULayers));
```

Create a struct to contain both the CNN and RNN sections of the full model.

```
doaModel.CNN = netCNN;
doaModel.RNN = netRNN;
```

Train DOA Network

Initialize variables used in the training loop.

```
iteration = 0;
averageGrad = [];
averageSqGrad = [];
epoch = 0;
bestLoss = Inf;
badEpochs = 0;
learnRate = trainOptionsDOA.InitialLearnRate;
```

To display training progress, initialize the supporting object `progressPlotterSELD`. The supporting object, `progressPlotterSELD`, is placed in your current folder when you open this example.

```
pp = progressPlotterSELD();
```

Run the training loop.

```
rng(0)
while epoch < trainOptionsDOA.MaxEpochs && badEpochs < trainOptionsDOA.ValidationPatience
    epoch = epoch + 1;

    % Shuffle mini-batch queue.
    shuffle(trainDOAmbq)

    while hasdata(trainDOAmbq)

        % Update iteration counter.
        iteration = iteration + 1;

        % Read mini-batch of data.
        [X,T] = next(trainDOAmbq);

        % Evaluate the model gradients and loss using dlfeval and the modelLoss function.
        [loss,grad,state] = dlfeval(@modelLoss,doaModel,X,T);
        loss = loss/size(T,2);

        % Update state.
        doaModel.CNN.State = state.CNN;
        doModel.RNN.State = state.RNN;

        % Update the network parameters using the Adam optimizer.
        [doaModel,averageGrad,averageSqGrad] = adamupdate(doaModel,grad,averageGrad, ...
            averageSqGrad,iteration,learnRate,trainOptionsDOA.GradientDecayFactor,trainOptionsDOA);

        % Update the training progress plot
        updateTrainingProgress(pp,Epoch=epoch,LearnRate=learnRate,Iteration=iteration,Loss=loss)
    end

    % Perform validation after each epoch
    loss = predictBatch(doaModel,validationDOAmbq);
```

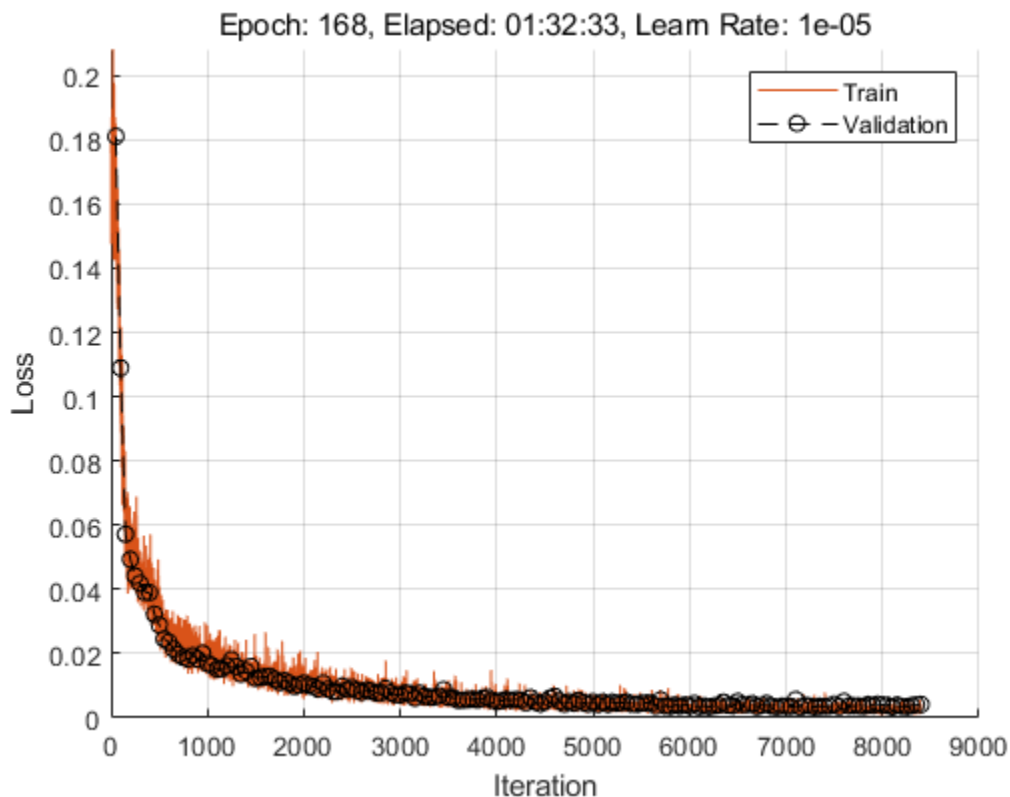
```

% Update the training progress plot with validation results.
updateValidation(pp, Loss=loss, Iteration=iteration)

% Create a checkpoint if the validation loss improved. If validation
% loss did not improve, add to the number of bad epochs.
if loss < bestLoss
    bestLoss = loss;
    badEpochs = 0;
    fileName = "DOA-BestModel";
    save(fileName, "doaModel");
else
    badEpochs = badEpochs + 1;
end

% Update learn rate
if rem(epoch, trainOptionsDOA.LearnRateDropPeriod)==0
    learnRate = learnRate*trainOptionsDOA.LearnRateDropFactor;
end
end
end

```



Evaluate System Performance

To evaluate your system's performance, use the location-sensitive detection error defined in [4] on page 1-805. Load the best-performing models.

```

sedModel = importdata("SED-BestModel.mat");
doaModel = importdata("DOA-BestModel.mat");

```

Location-sensitive detection is a joint metric that evaluates the results of both sound event detection and sound event localization tasks. In this type of evaluation, a true positive only occurs when the predicted label is correct, and the predicted location is within a predefined threshold of the true location. A threshold of 0.2 is used in this example which is about ~3% of the maximum possible error. To determine regions of silence in the prediction, set a confidence threshold on SED decisions. If the SED predictions are below that threshold, the frame is considered silence.

```
params.SpatialThreshold = 0.2;
params.SilenceThreshold = 0.1;
```

Compute the metrics for the validation data set using the `computeMetrics` on page 1-810 supporting function.

```
results = computeMetrics(sedModel,doaModel,validationSEDbq,validationDOAmbq,params);
results
```

```
results = struct with fields:
    precision: 0.4246
    recall: 0.4275
    f1Score: 0.4261
    avgErr: 0.1861
```

The `computeMetrics` supporting function can optionally smooth the decisions over time before evaluating the system. This option requires the Statistics and Machine Learning Toolbox™. Evaluate the system again, this time including the smoothing.

```
[results,cm] = computeMetrics(sedModel,doaModel,validationSEDbq,validationDOAmbq,params,ApplySmoothing);
results
```

```
results = struct with fields:
    precision: 0.5077
    recall: 0.5084
    f1Score: 0.5080
    avgErr: 0.1659
```

You can inspect the confusion matrix for SED predictions to get more insights on the prediction errors. The confusion matrix is only calculated over regions where there is an active sound source.

```
figure(Position=[100 100 800 800]);
confusionchart(cm,keys(params.SoundClasses))
```

True Class	Chink_and_clink	408					136	198		64		98	27		
	Computer_keyboard	9	1062	6	84		14			10	210	237		136	
	Cupboard_open_or_close		382	110	113	10			11	88		233			
	Drawer_open_or_close		133	12	1871				42		44	295		21	
	Female_speech_and_woman_speaking					641				56	13			30	
	Finger_snapping	16	75				924		3						
	Keys_jangling	159						1019					246	8	
	Knock	98		34			33		1105		18				
	Laughter					172			6	837	6			166	
	Male_speech_and_man_speaking			1		98			82	49	1220				
	Printer		139	17	370					10	6	739		69	
	Scissors		50	35	133		52	39	47			66	1902	225	
	Telephone		10	1	10	36			68	67	156			1953	
	Writing		228	20	126			20				30	310		519
			Chink_and_clink	Computer_keyboard	Cupboard_open_or_close	Drawer_open_or_close	Female_speech_and_woman_speaking	Finger_snapping	Keys_jangling	Knock	Laughter	Male_speech_and_man_speaking	Printer	Scissors	Telephone
		Predicted Class													

Conclusion

For next steps, you can download and try out the pretrained models from this example in this second example showing inference: “3-D Sound Event Localization and Detection Using Trained Recurrent Convolutional Neural Network” on page 1-815.

References

[1] Sharath Adavanne, Archontis Politis, Joonas Nikunen, and Tuomas Virtanen, "Sound event localization and detection of overlapping sources using convolutional recurrent neural networks," *IEEE J. Sel. Top. Signal Process.*, vol. 13, no. 1, pp. 34-48, 2019.

[2] Eric Guizzo, Riccardo F. Gramaccioni, Saeid Jamili, Christian Marinoni, Edoardo Massaro, Claudia Medaglia, Giuseppe Nachira, Leonardo Nucciarelli, Ludovica Paglialunga, Marco Pennese, Sveva Pepe, Enrico Rocchi, Aurelio Uncini, and Danilo Comminiello "L3DAS21 Challenge: Machine Learning for 3D Audio Signal Processing," 2021.

[3] Yin Cao, Qiuqiang Kong, Turab Iqbal, Fengyan An, Wenwu Wang, and Mark D. Plumbley, "Polyphonic sound event detection and localization using a two-stage strategy," arXiv preprint: arXiv:1905.00268v4, 2019.

[4] Mesaros, Annamaria, Sharath Adavanne, Archontis Politis, Toni Heittola, and Tuomas Virtanen. "Joint Measurement of Localization and Detection of Sound Events." *2019 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, 2019. <https://doi.org/10.1109/waspaa.2019.8937220>.

Supporting Functions

Extract Direction of Arrival (DOA) Targets

```
function T = extractDOATargets(csvFile,params)
%EXTRACTDOATARGETS Extract direction of arrival (DOA) targets
% T = extractDOATargets(fileName,params) parses the CSV file
% fileName and returns a matrix, T. The target matrix is an N-by-3
% matrix, where N corresponds to the number of frames and 3 corresponds to
% the 3 axes describing location in 3-D space.

% Preallocate target matrix. A frame of all zeros corresponds to no sound
% source.
T = zeros(params.Targets.NumFrames,3);

% Quantize the time stamps for sound sources into frames.
startendTime = [csvFile.Start,csvFile.End];
startendframe = time2frame(startendTime,params.Targets.TotalDuration,params.Targets.NumFrames);

% For each sound source, fill the target matrix sound source location for
% the appropriate number of frames.
for ii = 1:size(startendframe,1)
    idx = startendframe(ii,1):startendframe(ii,2)-1;
    T(idx,:) = repmat([csvFile.X(ii),csvFile.Y(ii),csvFile.Z(ii)],numel(idx),1);
end

% Scale the target so that it is between -1 and 1 (the bounds of the tanh
% activation layer). Wrap the target in a cell array for convenient batch
% processing.
T = {T/params.DOA.ScaleFactor};
end
```

Extract Sound Event Detection (SED) Targets

```
function T = extractSEDTargets(csvFile,params)
%EXTRACTSEDTARGETS Extract sound event detection (SED) targets
% T = extractSEDTargets(fileName,params) parses the CSV file
% fileName and returns a matrix of SED targets, T. The target matrix is an N-by-K
% matrix, where N corresponds to the number of frames and K corresponds to
% the number of sound classes.

% Preallocate target matrix. A frame of all zeros corresponds to no sound
% source.
T = zeros(params.Targets.NumFrames,params.SoundClasses.Count);

% Quantize the time stamps for sound sources into frames.
startendTime = [csvFile.Start,csvFile.End];
startendFrame = time2frame(startendTime,params.Targets.TotalDuration,params.Targets.NumFrames);

% For each sound source, fill the appropriate column of the target matrix
% with a 1, indicating that the sound class is present in that frame.
for ii = 1:size(startendFrame,1)
    classID = params.SoundClasses(csvFile.Class{ii});
    T(startendFrame(ii,1):startendFrame(ii,2)-1,classID) = 1;
end

% Wrap the target in a cell array for convenient batch processing.
T = {T};
end
```

Short-Time Fourier Transform (STFT)

```
function X = extractSTFT(s,params)
%EXTRACTSTFT Extract log-magnitude of centered STFT
% X = extractSTFT({s1,s2},params) concatenates s1 and s2 and then
% extracts the one-sided log-magnitude STFT. The signals are padded before
% the STFT so that the first window is centered on the first sample. The
% output is trimmed to remove the 1st (DC) coefficient and the last
% spectrum. The input params defines the STFT.

% Concatenate the signals along the second (channel) dimension.
audio = cat(2,s{:});

% Extract the centered STFT.
N = numel(params.SED.Window);
overlapLength = N - params.SED.HopLength;
S = centeredSTFT(audio,params.SED.Window,overlapLength,N);

% Trim the 1st coefficient from all spectrums and trim the last spectrum.
S = S(2:end,1:end-1,:);

% Convert to log-magnitude. Use an offset to protect against log of zero.
mag = log(abs(S) + eps);

% Cast output to single precision.
X = single(mag);
end
```


Generalized Cross Correlation with Phase Transform (GCC-PHAT)

```

function X = extractGCCPHAT(s,params)
%EXTRACTGCCPHAT Extract generalized cross correlation phase transform (GCC-PHAT) features
% X = extractGCCPHAT({s1,s2},params) concatenates s1 and s2 and then
% extracts the GCC-PHAT for all pairs of channels.

% Concatenate the signals corresponding to the two microphones.
audio = cat(2,s{:});

% Count the total number of input channels.
nChan = size(audio,2);

% Calculate the total number of output channels.
numOutputChannels = nchoosek(nChan,2);

% Preallocate a NumFeatures-by-NumFrames-by-NumChannels feature (predictor)
% matrix.
numFrames = size(audio,1)/params.DOA.HopLength;
X = zeros(params.DOA.NumBands,numFrames,numOutputChannels);

% -----
% Calculate GCC-PHAT for each pair of channels.
% Precompute STFT for each channel.
N = numel(params.DOA.Window);
overlapLength = N - params.DOA.HopLength;
micAB_stft = centeredSTFT(audio,params.DOA.Window,overlapLength,N);
conjmicAB_stft = conj(micAB_stft(:,:,2:end));
idx = 1;
for ii = 1:nChan - 1
    R = micAB_stft(:,:,ii).*conjmicAB_stft(:,:,ii:end);
    R = exp(1i .* angle(R));
    R = padarray(R, N/2 - 1,"post");
    gcc = fftshift(iff(R,[],1,"symmetric"),1);
    X(:,:,idx:idx+size(R,3)-1) = gcc(floor(N/2+1 - (params.DOA.NumBands-1)/2):floor(N/2+1 + (para

        idx = idx + size(R,3);
end
% -----

% Cast output to single precision.
X = single(X);

end

```

Centered Short-Time Fourier Transform (STFT)

```

function s = centeredSTFT(audio,win,overlapLength,fftLength)
%CENTEREDSTFT Centered STFT
% s = centeredSTFT(audioIn,win,overlapLength,fftLength) computes an STFT
% with the first window centered around the first sample. The two ends are
% padded with the reflected audio signal.

% Pad front and back of input signal.
firstR = flip(audio(1:fftLength/2,:),1);
lastR = flip(audio(end - fftLength/2 + 1:end,:),1);
sig = cat(1,firstR,audio,lastR);

```

```
% Perform STFT.
s = stft(sig,Window=win,OverlapLength=overlapLength,FFTLength=fftLength,FrequencyRange="onesided");
end
```

Convert Time Stamp to Frame Number

```
function fnum = time2frame(t,dur,numFrames)
%TIME2FRAME Convert time stamp to frame number
% fnum = time2frame(t,dur,numFrames) maps the times t, which exist in dur,
% to a frame number if dur is divided into numFrames.

stp = dur/numFrames;

qt = round(t./stp).*stp;

fnum = floor(qt*(numFrames - 1)/dur) + 1;
end
```

Forward Pass Through CNN and RNN Networks

```
function [loss,cnnState,rnnState,Y3] = forwardAll(model,X,T)
%FORWARDALL Forward pass of model through CNN and RNN networks
% [loss,cnnState,rnnState] = forwardAll(model,X,T) passes the predictors X
% through the model and returns the loss and the states of the networks in
% the model. The model is a struct containing a CNN network and an RNN
% network.
%
% [loss,cnnState,rnnState,Y] = forwardAll(model,X,T) also returns the final
% prediction of the model Y.

% Pass predictors through CNN.
[Y1,cnnState] = forward(model.CNN,X);

% Label the dimensions output from the CNN for consumption by the RNN.
Y2 = dldarray(Y1,"TCUB");

% Pass the predictors through the RNN.
[Y3,rnnState] = forward(model.RNN,Y2);

% Calculate the loss.
loss = seldNetLoss(Y3,T);
end
```

Full Model Prediction

```
function [loss,Y3] = predictAll(model,X,T)
%PREDICTALL Model prediction through CNN and RNN networks
% [loss,prediction] = predictAll(model,X,T) passes the predictors X through
% the model and returns the loss and the model prediction. The model is a
% struct containing a CNN network and an RNN network.

% Pass predictors through CNN.
Y1 = predict(model.CNN,X);

% Label the dimensions output from the CNN for consumption by the RNN.
Y2 = dldarray(Y1,"TCUB");
```

```
% Pass the predictors through the RNN.
Y3 = predict(model.RNN,Y2);
```

```
% Calculate the loss.
loss = seldNetLoss(Y3,T);
```

```
end
```

Predict Batch

```
function loss = predictBatch(model,mbq)
%PREDICTBATCH Calculate the loss of mini-batch queue
% loss = predictBatch(model,mbq) returns the total loss calculated by
% passing the entire contents of the mini-batch queue through the model.
```

```
% Reset mini-batch queue and initialize counters.
reset(mbq)
loss = 0;
n = 0;
```

```
while hasdata(mbq)
```

```
    % Read the predictors and targets from mini-batch queue.
    [X,T] = next(mbq);
```

```
    % Pass the mini-batch through the model and calculate the loss.
    lss = predictAll(model,X,T);
    lss = lss/size(T,2);
```

```
    % Update the total loss.
    loss = loss + lss;
```

```
    % Sum number of datapoints.
    n = n + 1;
```

```
end
```

```
% Divide the total loss accumulated by the number of mini-batches.
loss = loss/n;
```

```
end
```

Compute Model Loss, Gradients, and Network States

```
function [loss,gradients,state] = modelLoss(model,X,T)
%MODELLOSS Compute model loss, gradients, and network states
% [loss,gradients,state] = modelLoss(model,X,T) passes the
% predictors X through the model and returns the loss, the gradients, and
% the states of the networks in the model. The model is a struct containing
% a CNN network and an RNN network.
```

```
% Pass the predictors through the model.
[loss,cnnState,rnnState] = forwardAll(model,X,T);
```

```
% Isolate the learnables.
allGrad.CNN = model.CNN.Learnables;
allGrad.RNN = model.RNN.Learnables;
```

```
state.CNN = cnnState;
state.RNN = rnnState;

% Calculate the gradients.
gradients = dlgradient(loss,allGrad);

end
```

Loss Function of SELDnet

```
function loss = seldNetLoss(Y,T)
%SELDNETLOSS Compute the SELDnet loss function for DOA or SED models
% loss = seldNetLoss(Y,T) returns the SELDnet loss given predictions Y and
% targets T. The loss function depends on the network (DOA or SED). The
% network is inferred by the dimensions of the target. For the DOA network,
% the loss function is mean-squared error. For the SED network, the loss
% function is crossentropy.

% Determine whether the targets correspond to the DOA network or SED
% network.
isDOAModel = size(T,find(dims(T)=='C'))==3;

if isDOAModel
    % Calculate MSE loss.
    doaLoss = mse(Y,T);
    doaLossFactor = 2 / (size(Y,1) * size(Y,3));
    loss = doaLoss * doaLossFactor; % To align with the original implementation
else
    % Calculate cross-entropy loss.
    loss = crossentropy(Y,T,TargetCategories="independent",NormalizationFactor="all-elements");
end

loss = loss * size(T,2);

end
```

Compute Performance Metrics

```
function [r,cm] = computeMetrics(sedModel,doaModel,sedMBQ,doaMBQ,params,nvars)
%COMPUTEMETRICS Compute performance metrics
% [r,cm] = computeMetrics(sedModel,doaModel,sedMBQ,doaMBQ,params) returns
% a struct of performance metrics calculated over the SED and DOA
% validation mini-batch queues, and a confusion matrix cm valid SED
% regions.
arguments
    sedModel
    doaModel
    sedMBQ
    doaMBQ
    params
    nvars.ApplySmoothing = false;
end

% Initialize counters.
TP = 0;
FP = 0;
```

```

FN = 0;
it = 0;
ct = 0;
err = 0;

sedYAll = [];
sedTAll = [];

% Loop over all the data.
reset(sedMBQ)
reset(doaMBQ)
while hasdata(sedMBQ)

    % Get the predictors, targets, and predictions for the SED model.
    [sedXb,sedTb] = next(sedMBQ);
    [~,sedYb] = predictAll(sedModel,sedXb,sedTb);
    sedTb = extractdata(gather(sedTb));
    sedYb = extractdata(gather(sedYb));

    % Get the predictors, targets, and predictions for the DOA model.
    [doaXb,doaTb] = next(doaMBQ);
    [~,doaYb] = predictAll(doaModel,doaXb,doaTb);
    doaTb = extractdata(gather(doaTb));
    doaYb = extractdata(gather(doaYb));
    doaYb = doaYb*params.DOA.ScaleFactor;
    doaTb = doaTb*params.DOA.ScaleFactor;

    % Loop over the mini-batches.
    for batch = 1:size(sedYb,2)

        % Isolate the predictors and targets for current data point.
        sedY = squeeze(sedYb(:,batch,:));
        sedT = squeeze(sedTb(:,batch,:));
        doaY = squeeze(doaYb(:,batch,:));
        doaT = squeeze(doaTb(:,batch,:));

        % If the SED predictions of a frame are all made with low
        % confidence (beneath a threshold), assume that there is no sound
        % source present.
        isActive = ~(sum(double(sedY<params.SilenceThreshold),1)==size(sedY,1));

        % Convert the SED predictors and targets from one-hot vectors to
        % scalars.
        [~,sedY] = max(sedY,[],1);
        sedY = sedY.*isActive;

        [isActive,sedT] = max(sedT,[],1);
        sedT = sedT.*isActive;

        % Smooth outputs.
        if nvars.ApplySmoothing
            [doaY,sedY] = smoothOutputs(doaY,sedY,params);
        end

        % Perform location-sensitive detection.
        [tp,fp,fn,e,c] = locationSensitiveDetection(sedY,sedT,doaY,doaT,params);

        % Accumulate performance metrics.

```

```

        TP = TP + tp;
        FP = FP + fp;
        FN = FN + fn;
        err = err + e;
        ct = ct + c;

        sedYAll = [sedYAll sedY.*isActive]; %#ok<AGROW>
        sedTAll = [sedTAll sedT.*isActive]; %#ok<AGROW>
    end
    it = it + 1;
end

% Calculate performance metrics.
r.precision = TP/(TP + FP + eps);
r.recall = TP / (TP + FN + eps);
r.f1Score = 2*(r.precision*r.recall)/(r.precision + r.recall + eps);
r.avgErr = err/ct;

% Calculate confusion matrix.
confmat = confusionmat(sedTAll,single(sedYAll),Order=0:14);
cm = confmat(2:end,2:end); % Remove the silence from the confusion matrix.
end

```

Location Sensitive Detection

```

function [TP,FP,FN,totErr,ct] = locationSensitiveDetection(sedY,sedT,doaY,doaT,params)
%LOCATIONSENSITIVEDETECTION Location sensitive detection
% [TP,FP,FN,totErr,ct] =
% locationSensitiveDetection(sedY,sedT,doaY,doaT,params) calculates the
% true positive, false positive, false negative, DOA total error, and
% number of active targets. The definitions of each metric are provided in
% [4].

% Calculate distance.
dist = vecnorm(doaY-doaT);

% Determine if sounds active for reference and predictions.
isReferenceActive = sedT~=0;
isPredictedActive = sedY~=0;

% Calculate the total DOA error for reference-active sections.
totErr = sum(dist.*isReferenceActive);

% Count total number of active targets.
ct = sum(isReferenceActive);

% Determine if the DOA is within threshold per frame.
isDOAnear = dist < params.SpatialThreshold;

% True positive:
TP = sum(isDOAnear & isReferenceActive & isPredictedActive & (sedT==sedY));

% False positive:
FP1 = sum(~isReferenceActive & isPredictedActive);
FP2 = sum(isReferenceActive & isPredictedActive & (sedT~=sedY | ~isDOAnear));
FP = FP1 + FP2;

% False negative:

```

```

FN1 = sum(isReferenceActive & ~isPredictedActive);
FN2 = sum(isReferenceActive & (sedT~=sedY | ~isDOAnear));
FN = FN1 + FN2;

```

```
end
```

Smooth Outputs

```

function [doaYSmooth,sedYSmooth] = smoothOutputs(doaY,sedY,params)
%SMOOTHOUTPUTS Smooth DOA and SED predictions over time
% [doaYSmooth,sedYSmooth] = smoothOutputs(doaY,sedY,params) smooths the DOA
% and SED predictions over time.

% Preallocate smoothed outputs.
doaYSmooth = doaY;
sedYSmooth = sedY;

% Cluster the DOA predictions.
clusters = clusterdata(doaY',Criterion="distance",Cutoff=params.SpatialThreshold);
stt = 1;
enn = 1;

while enn <= params.Targets.NumFrames
    if clusters(stt) == clusters(enn)
        enn = enn + 1;
    else
        doaYSmooth(:,stt:enn-1) = smoothDOA(doaY(:,stt:enn-1));
        sedYSmooth(:,stt:enn-1) = smoothSED(sedY(:,stt:enn-1));
        stt = enn;
    end
end

end

doaYSmooth(:,stt:enn-1) = smoothDOA(doaY(:,stt:enn-1));
sedYSmooth(:,stt:enn-1) = smoothSED(sedY(:,stt:enn-1));

sedYSmooth = round(movmedian(sedYSmooth,5));

end

```

Smooth DOA Prediction

```

function smoothed = smoothDOA(chunk)
%SMOOTHDOA Smooth DOA prediction
% smoothed = smoothDOA(chunk) smooths DOA predictions by replacing the
% values of each axis with the mean of that axis in the chunk. The mean is
% calculated after discarding the lower and upper quarters of data.

% Determine the length of the chunk, and then indices to cut out the middle
% half of the data.
chlen = size(chunk,2);
st = max(round(chlen*1/4),1);
en = max(round(chlen*3/4),1);

% Sort the spatial axes (columns).
dim = sort(chunk,2);

```

```
% Take the mean of the inner half.  
smoothed = repmat(mean(dim(:,st:en),2),1,chlen);
```

```
end
```

Smooth SED Prediction

```
function smoothed = smoothSED(chunk)  
%SMOOTHSED Smooth SED prediction  
% smoothed = smoothSED(chunk) smooths SED predictions using the mode.
```

```
smoothed = repmat(mode(chunk),1,size(chunk,2));
```

```
end
```


3-D Sound Event Localization and Detection Using Trained Recurrent Convolutional Neural Network

In this example, you perform 3-D sound event localization and detection (SELD) using a pretrained deep learning model. For details about the model and how it was trained, see “Train 3-D Sound Event Localization and Detection (SELD) Using Deep Learning” on page 1-788. The SELD model uses two B-format ambisonic audio recordings to detect the presence and location of one of 14 sound classes commonly found in an office environment.

Download Pretrained Network

Download the pretrained SELD network, ambisonic test files, and labels. The model architecture is based on [1] on page 1-828 and [3] on page 1-828. The data the model was trained on, the labels, and the ambisonic test files, are provided as part of the L3DAS21 challenge [2] on page 1-828.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "SELDmodel.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
netFolder = fullfile(dataFolder, "SELDmodel");
addpath(netFolder)
```

Load and Inspect Data

Load the ambisonic data. First order B-format ambisonic recordings contain components that correspond to the sound pressure captured by an omnidirectional microphone (W) and sound pressure gradients X, Y, and Z that correspond to front/back, left/right, and up/down captured by figure-of-eight capsules oriented along the three spatial axes.

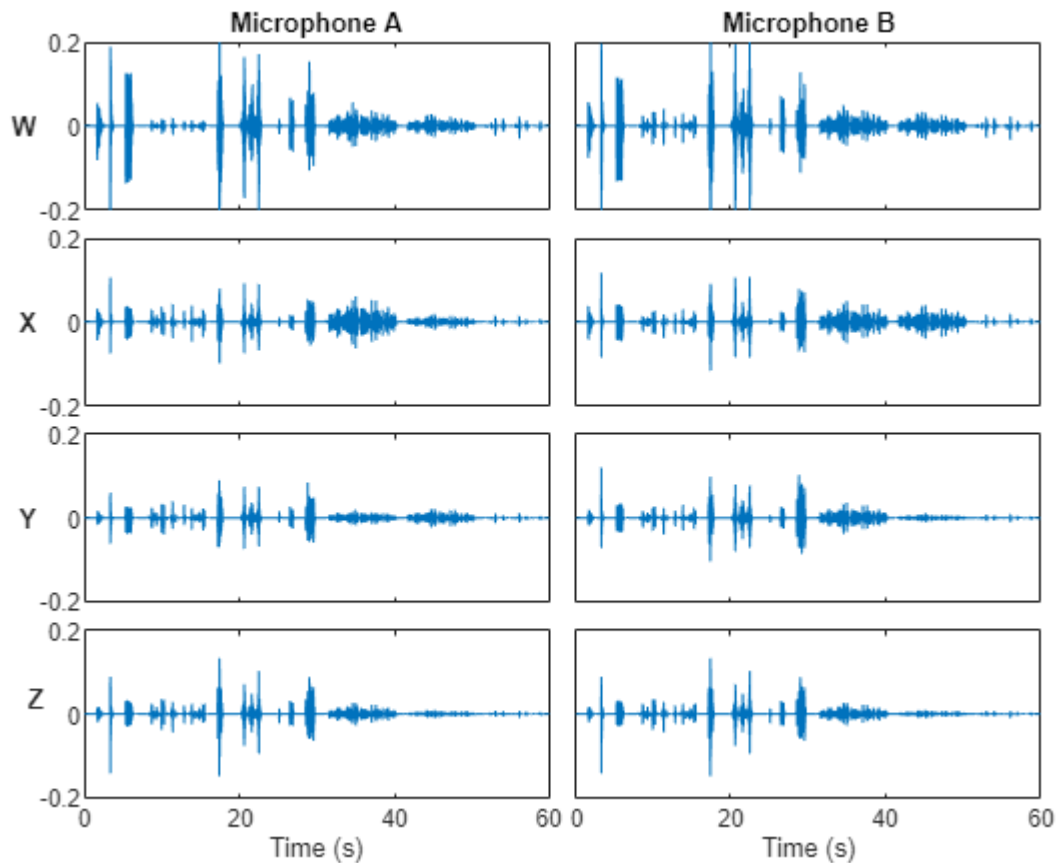
```
[micA, fs] = audioread("micA.wav");
micB = audioread("micB.wav");
```

Listen to a section of the data.

```
microphone = A ;
channel = X ;
start = 21.5 ;
stop = 39.7 ;
s = [micA, micB];
data = s(round(start*fs):round(stop*fs), channel+(microphone-1)*4);
sound(data, fs)
```

Plot the waveforms.

```
plotAmbisonics(micA, micB)
```



Use the supporting function, `getLabels`, to load the ground truth labels associated with the sound event detection (SED) and direction of arrival (DOA).

```
[sedLabels,doalabels] = getLabels();
```

`sedLabels` is a T-by-1 vector of keys over time, where the values map to one of 14 possible sound classes. A key of zero indicates a region of silence. The 14 possible sound classes are chink/clink, keyboard, cupboard, drawer, female speech, finger snapping, keys jangling, knock, laughter, male speech, printer, scissors, telephone, and writing.

`sedLabels`

`sedLabels` = 600×1 single column vector

```
0
0
0
0
0
0
0
0
0
0
0
```

```

:

soundClasses = getSoundClasses();
soundClasses(sedLabels+1)

ans = 1x600 categorical
      Silence      Silence      Silence      Silence      Silence      Silence

```

`doaLabels` is a T-by-3 matrix where T is the number of time steps and 3 corresponds to the X, Y, and Z axes in 3-D space.

```

doaLabels
doaLabels = 600x3

      0      0      0
      0      0      0
      0      0      0
      0      0      0
      0      0      0
      0      0      0
      0      0      0
      0      0      0
      0      0      0
      0      0      0
      :

```

In both cases, the 60-second ground truth has been discretized into 600 time steps.

Perform 3-D Sound Event Localization and Detection (SELD)

Use the supporting object, `seldModel`, to perform SELD. The object encapsulates the SELD model developed in “Train 3-D Sound Event Localization and Detection (SELD) Using Deep Learning” on page 1-788. Create the model, then call `seld` on the ambisonic data to detect and localize sound in time and space.

If you have Statistics and Machine Learning Toolbox™, the model applies smoothing to the decisions using moving averages and clustering.

```

model = seldModel();
[sed,doa] = seld(model,micA,micB);

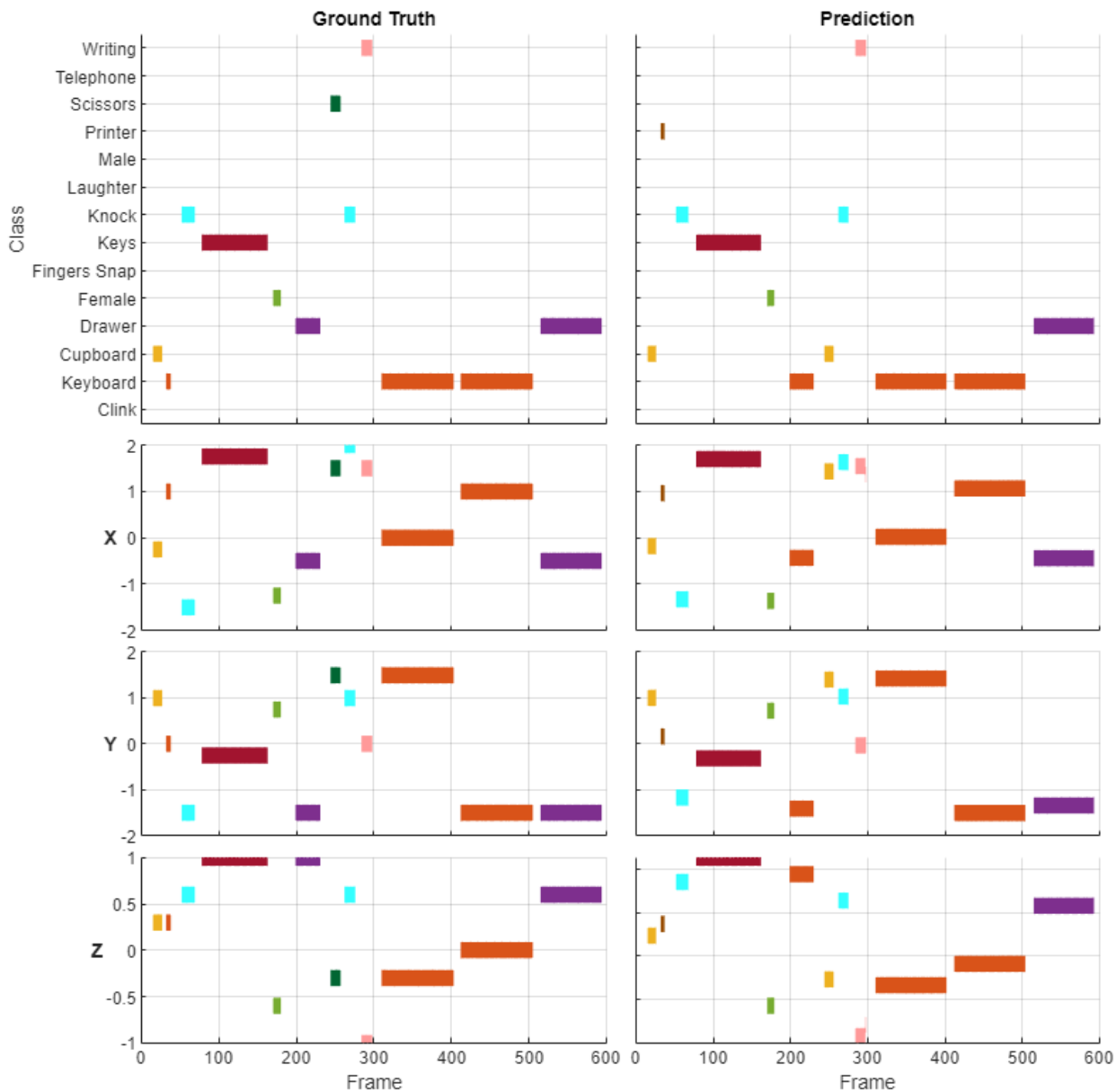
```

To visualize the system's performance over time, call the supporting function `plot2d` on page 1-822.

```

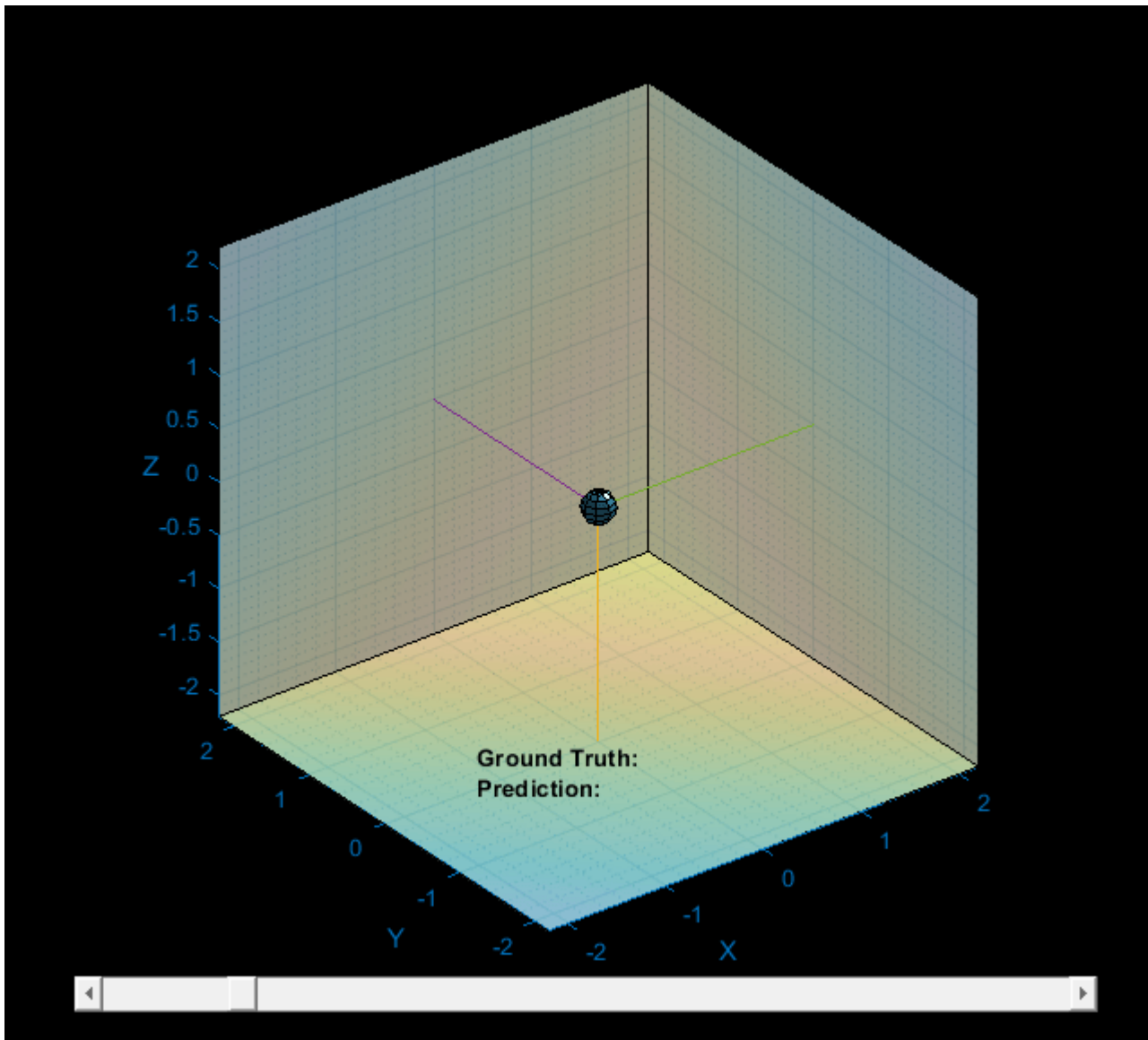
plot2d(sedLabels,doaLabels,sed,doa)

```



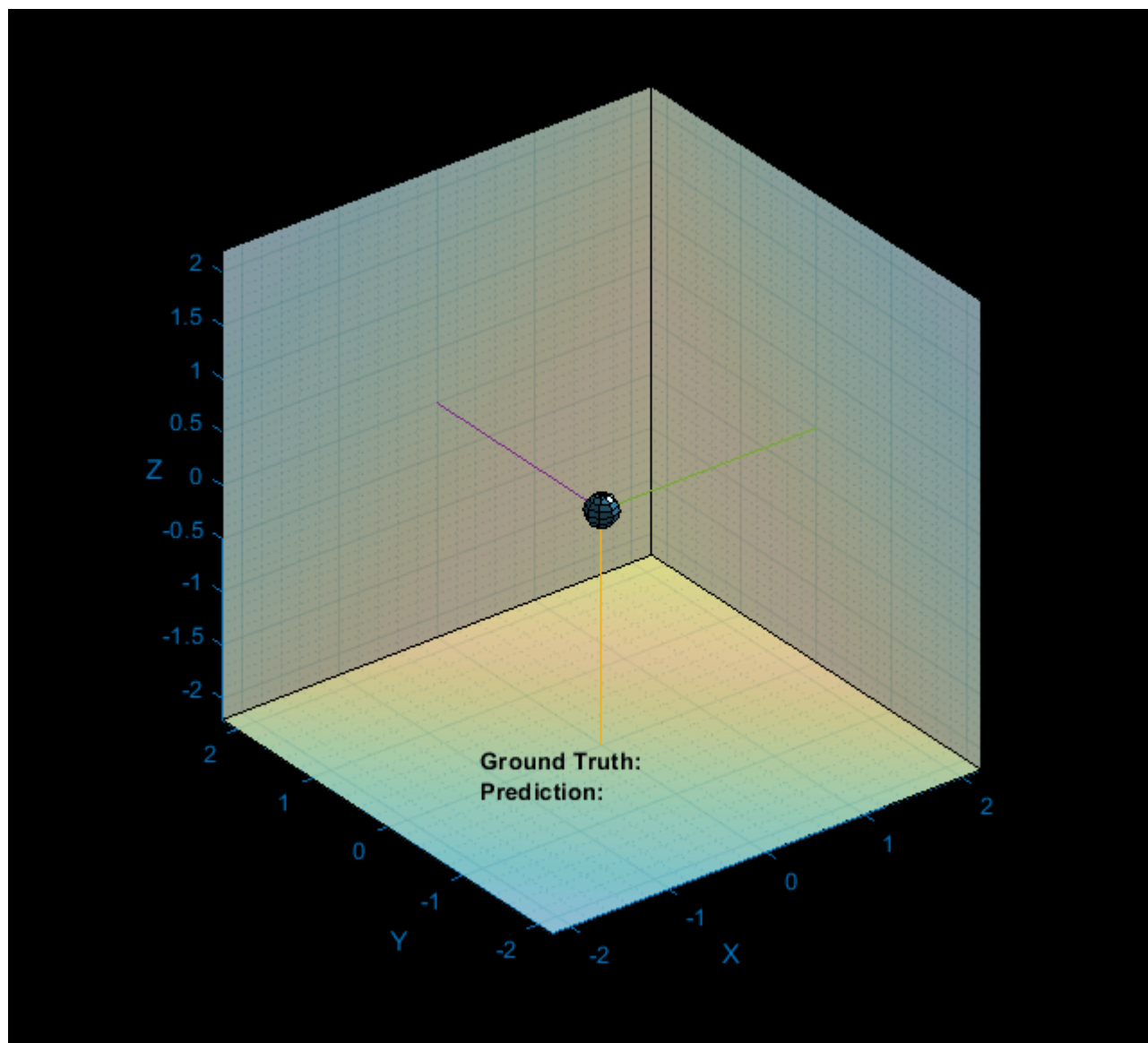
To visualize the system's performance in three spatial dimensions, call the supporting function `plot3d` on page 1-823. You can move the slider to visualize sound event locations detected at different times. The ground truth source location is identified by a semi-transparent sphere. The predicted source location is identified by a circle connected to the original by a dotted line.

```
plot3d(sedLabels, doaLabels, sed, doa);
```



SELD is a 4D problem, in that you are localizing the sound source in 3-D space and 1D time. To examine the system's performance in 4D, call the supporting function `plot4d` on page 1-827. The `plot4d` function plays the 3-D plot and corresponding ambisonic recording over time.

```
plot4d(micA(:,1),sedLabels,doaLabels,sed,doa)
```



Supporting Functions

Plot Ambisonics

```
function plotAmbisonics(micA,micB)
%PLOTAMBISONICS Plot B-format ambisonics over time
% plotAmbisonics(micA,micB) plots the ambisonic recordings collected from
% micA and micB. The channels are plotted along the rows of a 4-by-2 tiled
% layout (W,X,Y,Z). The first column of the plot corresponds to data from
% microphone A and the second column corresponds to data from microphone B.

figure(1)
tiledlayout(4,2,TileSpacing="tight")

t = linspace(0,60,size(micA,1));
```

```

nexttile
plot(t,micA(:,1))
title("Microphone A")
yL = ylabel("W",FontWeight="bold");
set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")
axis([t(1),t(end),-0.2,0.2])
set(gca,Xticklabel=[])

nexttile
plot(t,micB(:,1))
title("Microphone B")
axis([t(1),t(end),-0.2,0.2])
set(gca,Yticklabel=[],XtickLabel=[])

nexttile
plot(t,micA(:,2))
yL = ylabel("X",FontWeight="bold");
set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")
axis([t(1),t(end),-0.2,0.2])
set(gca,Xticklabel=[])

nexttile
plot(t,micB(:,2))
axis([t(1),t(end),-0.2,0.2])
set(gca,Yticklabel=[],XtickLabel=[])

nexttile
plot(t,micA(:,3))
yL = ylabel("Y",FontWeight="bold");
set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")
axis([t(1),t(end),-0.2,0.2])
set(gca,Xticklabel=[])

nexttile
plot(t,micB(:,3))
axis([t(1),t(end),-0.2,0.2])
set(gca,Yticklabel=[],XtickLabel=[])

nexttile
plot(t,micB(:,4))
yL = ylabel("Z",FontWeight="bold");
set(yL,Rotation=0)
axis([t(1),t(end),-0.2,0.2])
xlabel("Time (s)")

nexttile
plot(t,micB(:,4))
axis([t(1),t(end),-0.2,0.2])
set(gca,Yticklabel=[])
xlabel("Time (s)")
end

```

Plot Time Series

```

function plotTimeSeries(sed,values)
%PLOTTIMESERIES Plot time series
% plotTimeSeries(sed,values) is leveraged by plot2d to plot the color-coded
% SED or DOA estimation.

```

```
colors = getColors();
hold on
for ii = 1:numel(sed)
    cls = sed(ii);
    if cls > 0
        x = [ii-1,ii];
        y = repelem(values(ii),2);
        plot(x,y,Color=colors{cls},LineWidth=8)
    end
end
hold off
grid on
end
```

Plot 2D

```
function plot2d(sedLabels,doaLabels,sed,doa)
%PLOT2D Plot 2D
% plot2d(sedLabels,doaLabels,sed,doa) creates plots for SED, SED ground
% truth, DOA estimation, and DOA ground truth.

fh = figure(2);
set(fh,Position=[100 100 800 800])

SoundClasses = ["Click","Keyboard","Cupboard","Drawer","Female","Fingers Snap", ...
    "Keys","Knock","Laughter","Male","Printer","Scissors","Telephone","Writing"];

tiledlayout(5,2,TileSpacing="tight")

nexttile([2,1])
plotTimeSeries(sedLabels,sedLabels);
yticks(1:14)
ytickLabels(SoundClasses)
ylim([0.5,14.5])
ylabel("Class")
title("Ground Truth")
set(gca,XtickLabel=[])

nexttile([2,1])
plotTimeSeries(sed,sed);
yticks(1:14)
ylim([0.5,14.5])
title("Prediction")
set(gca,YtickLabel=[],XtickLabel=[])

nexttile
plotTimeSeries(sedLabels,doaLabels(:,1));
yL = ylabel("X",FontWeight="bold");
set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")
set(gca,XtickLabel=[])

nexttile
plotTimeSeries(sed,doa(:,1));
set(gca,YtickLabel=[],XtickLabel=[])
```



```

nexttile
plotTimeSeries(sedLabels,doaLabels(:,2));
yL = ylabel("Y",FontWeight="bold");
set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")
set(gca,Xticklabel=[])

nexttile
plotTimeSeries(sed,doa(:,2));
set(gca,Yticklabel=[],XtickLabel=[])

nexttile
plotTimeSeries(sedLabels,doaLabels(:,3));
xlabel("Frame")
yL = ylabel("Z",FontWeight="bold");
set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")

nexttile
plotTimeSeries(sed,doa(:,3));
xlabel("Frame")
set(gca,YtickLabel=[])

end

```

Plot 3-D

```

function data = plot3d(sedLabels,doaLabels,sed,doa,nvars)
%PLOT3D Plot 3-D
% plot3d(sedLabels,doaLabels,sed,doa) creates a 3-dimensional plot with a
% slider. Moving the slider moves the frame in the plot. The location of
% the recording is located at the origin. The location of a sound event is
% noted by a semi-transparent orb. The location of the estimated sound
% event is noted by a filled circle connected to the origin by a line. The
% estimated sound event class and the true sound event class for the
% current frame are displayed on the plot.
%
% plot3d(sedLabels,doaLabels,sed,doa,IncludeSlider=false) creates a 3-D plot
% but does not add the slider and associated callback. This format of the
% plot is leveraged by plot4d.

arguments
    sedLabels
    doaLabels
    sed
    doa
    nvars.IncludeSlider = true;
end

% Create data struct to contain plot information.
data.sedLabels = sedLabels;
data.doaLabels = doaLabels;
data.sed = sed;
data.doa = doa;
data.Colors = getColors();
data.SoundClasses = ["Clink","Keyboard","Cupboard","Drawer","Female","Fingers Snap", ...
    "Keys","Knock","Laughter","Male","Printer","Scissors","Telephone","Writing"];

```

```

% Create figure.
if nvars.IncludeSlider
    data.FigureHandle = figure(3);
else
    data.FigureHandle = figure(4);
end
set(data.FigureHandle,Position=[680,400,640,580],Color="k",MenuBar="none",ToolBar="none")

% Initialize plot.
data = initialize3DPlot(data);

% Add slider for 3-D plot.
if nvars.IncludeSlider
    N = numel(data.sedLabels);
    frame = 1/N;
    b = uicontrol(Parent=data.FigureHandle,Style="slider",Position=[40,20,570,20], ...
        value=frame*80,min=1/N,max=1,units="pixel", ...
        SliderStep=[1/N,20/N]);
    cbk = @(es,ed)update3DPlot(es.Value,data);
    addlistener(b,ContinuousValueChange=cbk);
end

end

```

Initialize 3-D Plot

```

function data = initialize3DPlot(data)
%INITIALIZE3DPLOT Initialize 3-D plot
% data = initialize3DPlot(data) creates the 3-D plot and initializes lines,
% dots, and surfaces that are included in the plot. data is appended to
% include handles for figure properties.

% Make sure the figure is visible.
data.FigureHandle.Visible = "on";

% Initialize the line plot that connects origin to predicted location.
data.YPlot = plot3([0,1],[0,1],[0,1],":",Color="k",LineWidth=1.5);
hold on
data.YPlot.Visible = "off";

% Initialize the dot plot that marks the predicted location.
data.YPlotDot = plot3(0,0,0,"o",MarkerSize=8,MarkerEdgeColor="k",LineWidth=2,MarkerFaceColor="k");
data.YPlotDot.Visible = "off";

% Initialize the sphere that marks the target location.
[x,y,z] = sphere;
data.TPlotDot = surf(x,y,z,FaceAlpha=0.2,EdgeColor="none",FaceColor="b");
data.TPlotDot.Visible = "off";

% Create a sphere to mark the origin. This is where the ambisonic
% microphones are located.
light
[X,Y,Z] = sphere(8);
surf(X*0.15,Y*0.15,Z*0.15,FaceColor=[0.3010 0.7450 0.9330],LineWidth=0.25);

% Create 'walls' on the 3-D plot to aid 3-D visualization.
patch([2.2,2.2,2.2,2.2],[2.2,2.2,-2.2,-2.2],[-2.2,2.2,2.2,-2.2],[3,2,1,2],FaceAlpha=0.5,FaceColor

```

```

patch([2.2,2.2,-2.2,-2.2],[2.2,2.2,2.2,2.2],[-2.2,2.2,2.2,-2.2],[3,2,1,2],FaceAlpha=0.5,FaceColor='k');
patch([2.2,2.2,-2.2,-2.2],[2.2,-2.2,-2.2,2.2],[-2.2,-2.2,-2.2,-2.2],[3,2,1,2],FaceAlpha=0.5,FaceColor='k');

% Create guidelines on the 3-D plot to aid 3-D visualization.
plot3([0,0],[0,0],[-2.2,0])
plot3([0,0],[2.2,0],[0,0])
plot3([2.2,0],[0,0],[0,0])

% Set axes limits.
xlim([-2,2])
ylim([-2,2])
zlim([-2,2])

% Add axis labels.
xlabel("X",Color=[0,0.4470,0.7410]);
ylabel("Y",Color=[0,0.4470,0.7410]);
zlabel("Z",Color=[0,0.4470,0.7410],Rotation=0);
set(gca,XColor=[0,0.4470,0.7410],YColor=[0,0.4470,0.7410],ZColor=[0,0.4470,0.7410])

% Initialize annotations for the ground truth and predicted labels.
annotation("textbox",[0.4,0.2,0.6,0.1],String="Ground Truth: ",FitBoxToText="on",Color="k",EdgeColor="k");
annotation("textbox",[0.4,0.17,0.4,0.1],String="Prediction: ",FitBoxToText="on",Color="k",EdgeColor="k");
data.GTAnnotation = annotation("textbox",[0.55,0.2,0.6,0.1],String=" ",FitBoxToText="on",Color="k",EdgeColor="k");
data.PredictedAnnotation = annotation("textbox",[0.55,0.17,0.4,0.1],String=" ",FitBoxToText="on",Color="k",EdgeColor="k");

grid on
grid minor
axis equal
hold off
end

```

Update 3-D Plot

```

function update3DPlot(timeFrame,data)
%UPDATE3DPLOT Update 3-D Plot
% update3DPlot(timeFrame,data) updates the 3-D plot to display data
% corresponding to the specified time frame.

timeFrame = round(timeFrame*numel(data.sedLabels));

if data.sedLabels(timeFrame) > 0
    % Turn plot visibility on.
    data.TPlotDot.Visible = "on";
    data.GTAnnotation.Visible = "on";

    % Get the current target sound class.
    gtClass = data.SoundClasses{data.sedLabels(timeFrame)};

    % Get current location coordinates and SED color code.
    doa = data.doaLabels(timeFrame,:);
    tcol = data.Colors{data.sedLabels(timeFrame)};

    % Update target sphere.
    [x,y,z] = sphere;
    r = 0.2;
    data.TPlotDot.XData = x*r + doa(1);
    data.TPlotDot.YData = y*r + doa(2);
    data.TPlotDot.ZData = z*r + doa(3);
end

```

```
    data.TPlotDot.FaceColor = tcol;
    data.TPlotDot.MarkerEdgeColor = tcol;
else
    % Turn plot visibility off.
    data.TPlotDot.Visible = "off";
    data.GTAnnotation.Visible = "off";

    % Set the current target sound class to silence and color-code as
    % black.
    gtClass = "Silence";
    tcol = "k";
end

if data.sed(timeFrame) > 0
    % Turn plot visibility on.
    data.PredictedAnnotation.Visible = "on";
    data.YPlot.Visible = "on";
    data.YPlotDot.Visible = "on";

    % Get the current predicted sound class.
    pClass = data.SoundClasses{data.sed(timeFrame)};

    % Get current location coordinates and SED color code.
    doa = data.doa(timeFrame,:);
    pcol = data.Colors{data.sed(timeFrame)};

    % Update prediction line.
    data.YPlot.XData = [0,doa(1)];
    data.YPlot.YData = [0,doa(2)];
    data.YPlot.ZData = [0,doa(3)];
    data.YPlot.Color = pcol;

    % Update the prediction dot.
    data.YPlotDot.XData = doa(1);
    data.YPlotDot.YData = doa(2);
    data.YPlotDot.ZData = doa(3);
    data.YPlotDot.Color = pcol;
    data.YPlotDot.MarkerEdgeColor = pcol;
    data.YPlotDot.MarkerFaceColor = pcol;
else
    % Turn plot visibility off.
    data.YPlot.Visible = "off";
    data.YPlotDot.Visible = "off";
    data.PredictedAnnotation.Visible = "off";

    % Set the current predicted sound class to silence and color-code as
    % black.
    pClass = "Silence";
    pcol = "k";
end

% Update the annotation strings and color code them.
if isequal(tcol,pcol)
    col = 'b';
else
    col = 'r';
end
data.GTAnnotation.String = gtClass;
```

```

data.GTAnnotation.Color = col;
data.PredictedAnnotation.String = pClass;
data.PredictedAnnotation.Color = col;

```

```

drawnow
end

```

Plot 4D

```

function plot4d(audioToPlay, sedLabels, doaLabels, sed, doa)
%PLOT4D Plot 4D
% plot4d(audioToPlay, sedLabels, doaLabels, sed, doa) creates a "movie" of
% ground truth and estimated sound events in a 3-D environment over time.
% The movie runs in real time and plays the audioToPlay to your default
% sound device.

% Create an audioDeviceWriter object to play streaming audio.
adw = audioDeviceWriter(SampleRate=32e3);

% Create and fill a dsp.AsyncBuffer to read chunks of audio data.
buff = dsp.AsyncBuffer(size(audioToPlay,1));
write(buff, audioToPlay);

% Create a 3-D plot without a slider.
data = plot3d(sedLabels, doaLabels, sed, doa, IncludeSlider=false);

drawnow

% The true and predicted label definitions have resolutions of 0.1 seconds. Create a
% labels vector to only update the 3-D plot when necessary.
changepoints = 0.1:0.1:60;

% Initialize counters.
idx = 1;
elapsedTime = 0;

% Loop while audio data is unread.
while buff.NumUnreadSamples~=0

    % Update a plot if a changepoint is reached.
    if elapsedTime>changepoints(idx)
        update3DPlot(idx/600, data)
        idx = idx+1;
    end

    % Write a chunk of data to your sound card.
    adw(read(buff, 400));

    % Push the elapsed time forward.
    elapsedTime = elapsedTime + 400/32e3;
end
end

```

Get Colors

```
function colors = getColors()
%GETCOLORS Get colors
% colors = getColors() returns a cell array of 14 unique colors.

% Define 14 colors to color-code the sound classes.
colors = {[0,0.4470,0.7410],[0.8500,0.3250,0.0980],[0.9290,0.6940,0.1250],[0.4940,0.1840,0.5560]
          [0.4660 0.6740 0.1880],[0.3010 0.7450 0.9330],[0.6350 0.0780 0.1840],[0.2,1,1],[0.6,0,0.6],
          [0.6,0.6,0],[0.6,0.3,0],[0,0.4,0.2],[0.2,0,0.4],[1,0.6,0.6]};
end
```

Get Sound Classes

```
function soundClasses = getSoundClasses()
%GETSOUNDCLASSES Get map between sound classes (keys) and values.

soundClasses = categorical(["Silence","Clink","Keyboard","Cupboard","Drawer","Female","Fingers S
                          "Keys","Knock","Laughter","Male","Printer","Scissors","Telephone","Writing"]);
end
```

References

[1] Sharath Adavanne, Archontis Politis, Joonas Nikunen, and Tuomas Virtanen, "Sound event localization and detection of overlapping sources using convolutional recurrent neural networks," *IEEE J. Sel. Top. Signal Process.*, vol. 13, no. 1, pp. 34-48, 2019.

[2] Eric Guizzo, Riccardo F. Gramaccioni, Saeid Jamili, Christian Marinoni, Edoardo Massaro, Claudia Medaglia, Giuseppe Nachira, Leonardo Nucciarelli, Ludovica Paglialunga, Marco Pennese, Sveva Pepe, Enrico Rocchi, Aurelio Uncini, and Danilo Comminiello "L3DAS21 Challenge: Machine Learning for 3D Audio Signal Processing," 2021.

[3] Yin Cao, Qiuqiang Kong, Turab Iqbal, Fengyan An, Wenwu Wang, and Mark D. Plumbley, "Polyphonic sound event detection and localization using a two-stage strategy," *arXiv preprint: arXiv:1905.00268v4*, 2019.

Import Audacity Labels to Signal Labeler

This example shows how to import labels created in Audacity™ into **Signal Labeler**.

Read Labels in MATLAB

You have an audio file consisting of a human voice uttering "Volume up" several times.

Load the audio file to the MATLAB® workspace.

```
audioFile = "speaker1.ogg";
[x,fs] = audioread(audioFile);
sound(x, fs)
```

You label the file in Audacity [1] and export the labels to `speaker.txt`.

Read the labels using `readtable`. The labels consist of three columns corresponding to the speech utterances along with their respective start and end times (in seconds).

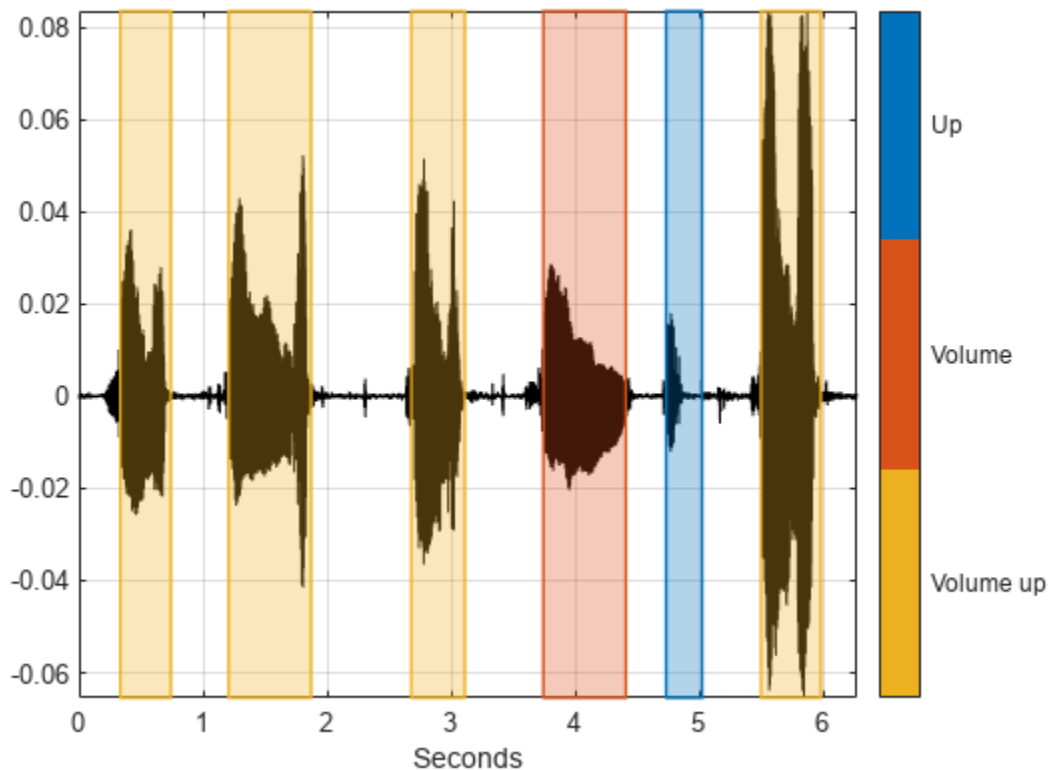
```
labelFile = "speaker1.txt";
roiTable = readtable(labelFile,Delimiter="tab");
roiTable.Properties.VariableNames = ["StartTime", "EndTime", "Value"]
```

```
roiTable=6x3 table
  StartTime      EndTime      Value
  _____  _____  _____
    0.33273      0.74033  {'Volume up'}
    1.2062       1.8716  {'Volume up'}
    2.6785       3.111   {'Volume up'}
    3.7432       4.4087  {'Volume'  }
    4.7331       5.0243  {'Up'      }
    5.4984       5.9809  {'Volume up'}
```

In order to gain insight into the labels, you plot the audio signal along with a mask corresponding to labeled regions of speech.

Convert the signal regions of interest to a binary mask.

```
mask=signalMask(table(roiTable{:,1:2},categorical(roiTable{:,3})),SampleRate=fs);
plotsigroi(mask, x, true);
```



Convert Labels to a Labeled Signal Set

Next, convert the label data to a `labeledSignalSet` that can be imported to **Signal Labeler**.

First, define the label type. Specify "roi" (Region of interest) for the label type, and "string" for the label datatype.

```
labelName = "Speech";
lblDef = signalLabelDefinition(string(labelName),...
                               LabelType="roi",...
                               LabelDataType="string");
```

Next, create a `labeledSignalSet` pointing to the labeled audio file. Add the label definition to the labeled signal set.

```
lss = labeledSignalSet(audioDatastore(audioFile),lblDef);
```

Set the label values.

```
roiLimits = [roiTable.StartTime roiTable.EndTime];
setLabelValue(lss,1,labelName,roiLimits,string(roiTable.Value));
```

Load Labels in Signal Labeler

You are now ready to read these labels into **Signal Labeler**.

1) Open `signalLabeler`

2) Click **Import**, then From Workspace, and import lss.

The audio signal is now available to you along with its labels.

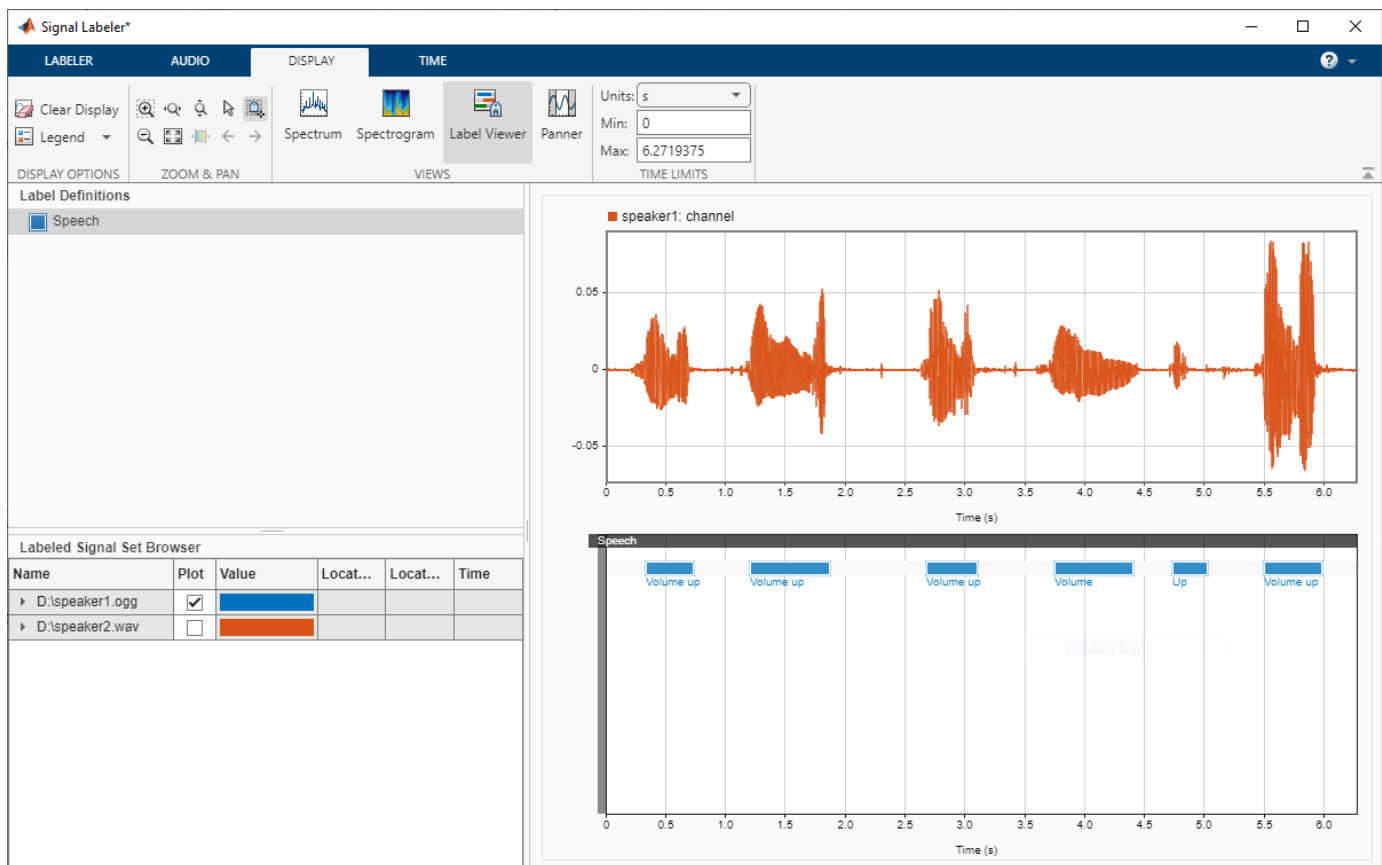
Import a Labeled Dataset

The helper function `importLabels` creates a labeled signal set corresponding to labels for multiple audio files. In this example, you work with two audio files with labels stored in text files.

Call `importLabels` to create a signal data set for multiple annotated files. Specify the label name as "Speech".

```
lss = importLabels("Speech");
```

You can now load `lss` in `SignalLabeler` by following the same steps from the previous section.



```
function lss = importLabels(labelName)
% IMPORTLABELS Import labels from multiple audio file
%
% The function assumes the labels are stored in *.txt files
% The function assumes the label file name is identical to the
% audio file name.
```

```
labelFiles = dir("*.txt");
labelFiles = {labelFiles.name};
numFiles = length(labelFiles);
```

```
% Create an audio datastore pointing to all audio files in the current
% folder
ads = audioDatastore(pwd);

lss = labeledSignalSet(ads);
lblDef = signalLabelDefinition(string(labelName),...
    LabelType="roi",...
    LabelDataType="string");
addLabelDefinitions(lss, lblDef)

for index0=1:numFiles
    filename = labelFiles{index0};
    roiTable = readtable(filename, Delimiter="tab");
    roiTable.Properties.VariableNames = ["StartTime", "EndTime", "Value"];
    roiLimits = [roiTable.StartTime roiTable.EndTime];
    setLabelValue(lss, index0, labelName, roiLimits, roiTable.Value);
end
end
```

References

[1] <https://www.audacityteam.org/>

Room Impulse Response Simulation with the Image-Source Method and HRTF Interpolation

Room impulse response simulation aims to model the reverberant properties of a space without having to perform acoustic measurements. Many geometric and wave-based room acoustic simulation methods exist in the literature [1] on page 1-843. The image-source method is a popular and relatively straightforward geometric method [2] on page 1-844. It models the specular reflections between a transmitter and a receiver.

This example showcases the image-source method for a simple "shoebox" (cuboid) room. The example also uses head-related transfer function (HRTF) interpolation to simulate the received sound at the ears of the listener.

Define Room Parameters

You simulate the impulse response of a shoebox empty room.

Define the room dimensions, in meters (width, length and height, respectively).

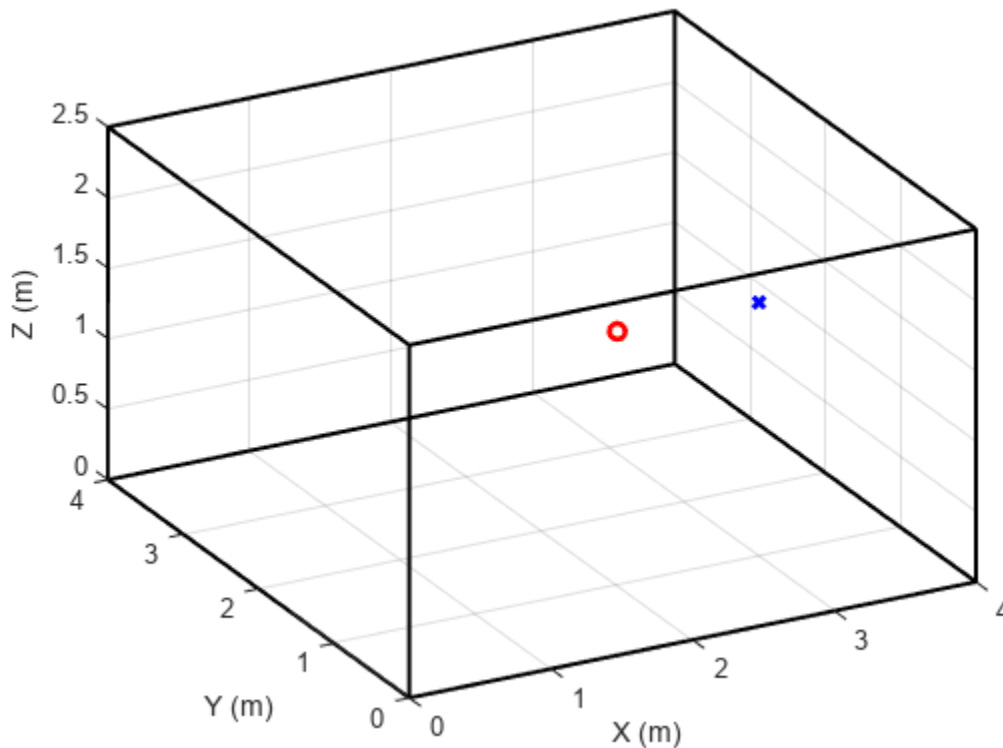
```
roomDimensions = [4 4 2.5];
```

You treat the receiver and transmitter as points within the space of the room. Define their coordinates, in meters.

```
receiverCoord = [2 1 1.8];  
sourceCoord = [3 1 1.8];
```

Plot the room space along with the receiver (red circle) and transmitter (blue x).

```
h = figure;  
plotRoom(roomDimensions, receiverCoord, sourceCoord, h)
```



The Image-Source Method

Image-source is a geometric simulation method that models specular sound reflection paths between the source and receiver. It assumes that sound travels in straight lines (rays) which undergo perfect reflections when they encounter an obstacle (in our case, one of the four walls, the floor, or the ceiling of the room).

When a sound ray hits a wall, it spawns a mirrored "image" source. The image source is the symmetrical reflection of the original source with respect to the encountered boundary. Higher-order reflections (rays that reach the receiver after bouncing off multiple obstacles) are modeled by repeating the mirroring process with respect to each encountered obstacle.

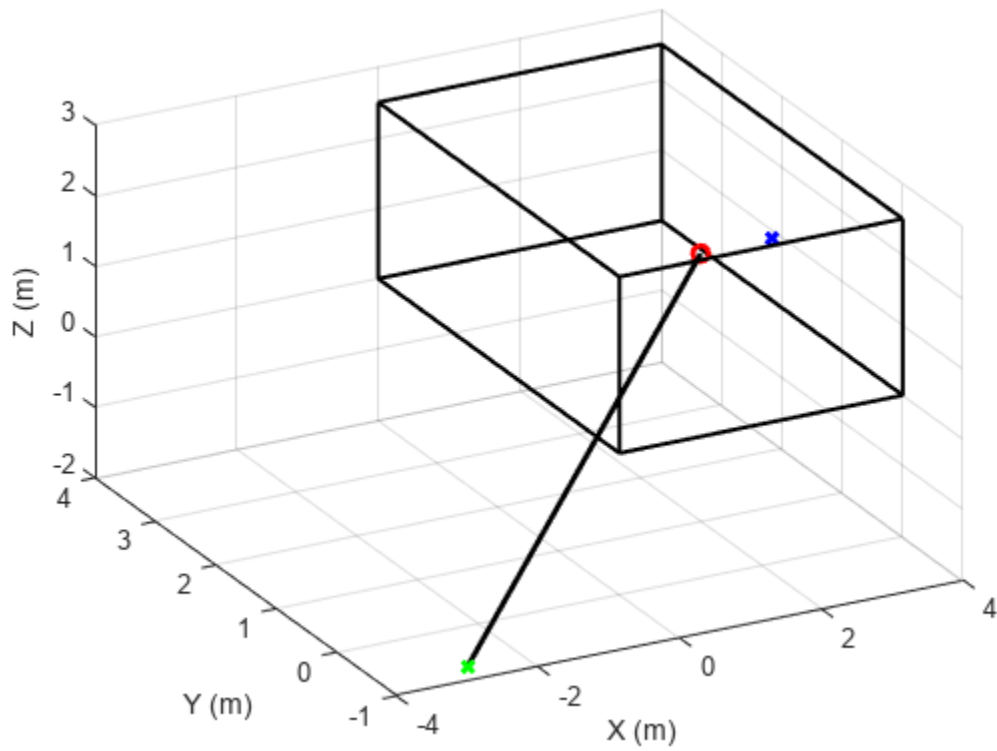
As an example, consider the ray that bounces off two walls and the floor before arriving at the receiver. Define the coordinates of the equivalent image for this ray.

```
imageSource = [-sourceCoord(1) -sourceCoord(2) -sourceCoord(3)];
```

The ray is modeled by the straight line connecting the image to the receiver. The length of this straight line is equal to the traveled distance from the original source to the receiver along the reflected ray.

Visualize the image-source and the resulting path.

```
plot3(imageSource(1),imageSource(2),imageSource(3),"gx",LineWidth=2)
plot3([imageSource(1) receiverCoord(1)], ...
      [imageSource(2) receiverCoord(2)], ...
      [imageSource(3) receiverCoord(3)],Color="k",LineWidth=2)
```



Visualize Multiple Images

To calculate the room impulse response, add the contributions of a large number of source images.

Extend the visible space around the room to ensure the images appear in the plot.

```
h2 = figure;
```

```
plotRoom(roomDimensions, receiverCoord, sourceCoord, h2)
```

```
Lx = roomDimensions(1);
```

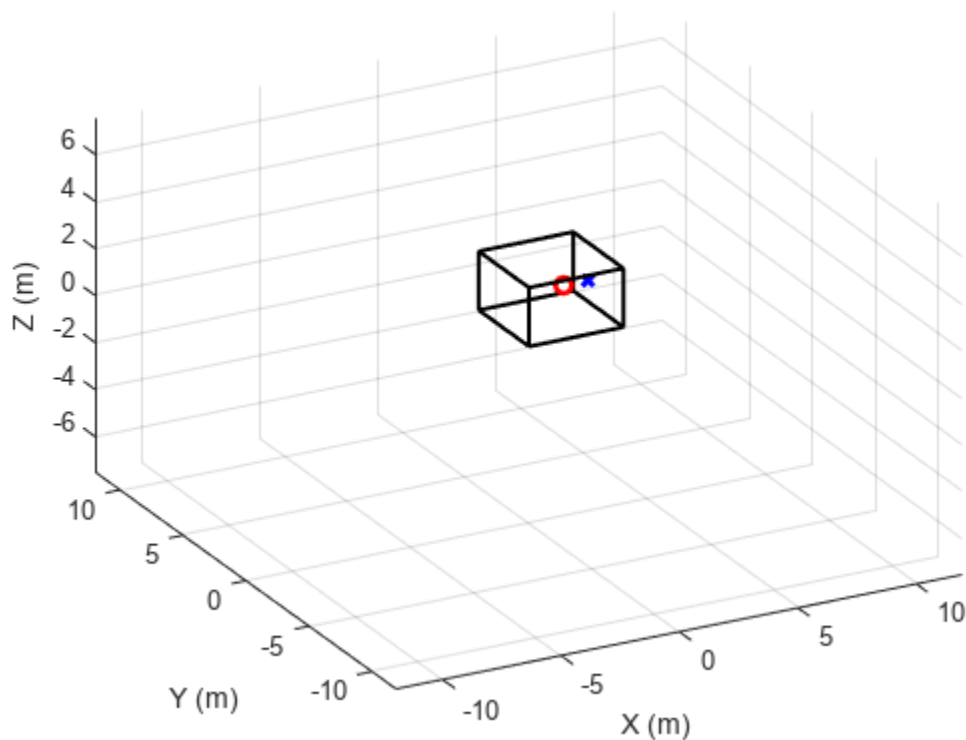
```
Ly = roomDimensions(2);
```

```
Lz = roomDimensions(3);
```

```
xlim([-3*Lx, 3*Lx]);
```

```
ylim([-3*Ly, 3*Ly]);
```

```
zlim([-3*Lz, 3*Lz]);
```



Visualize a subset of the source images. Compute the image coordinates based on equations 6 and 7 in [2].

Model the eight combinations stemming from possible reflections along the x -, y - and z - axes.

```
x = sourceCoord(1);
y = sourceCoord(2);
z = sourceCoord(3);
sourceXYZ = [-x -y -z;...
             -x -y  z;...
             -x  y -z;...
             -x  y  z;...
             x  -y -z;...
             x  -y  z;...
             x  y -z;...
             x  y  z].';
```

Model scenarios with multiple reflections by looping over the x -, y - and z - axes. These loops have infinite ranges in theory. You will see how to practically limit the ranges in the next section. For now, select arbitrary limits for the loops.

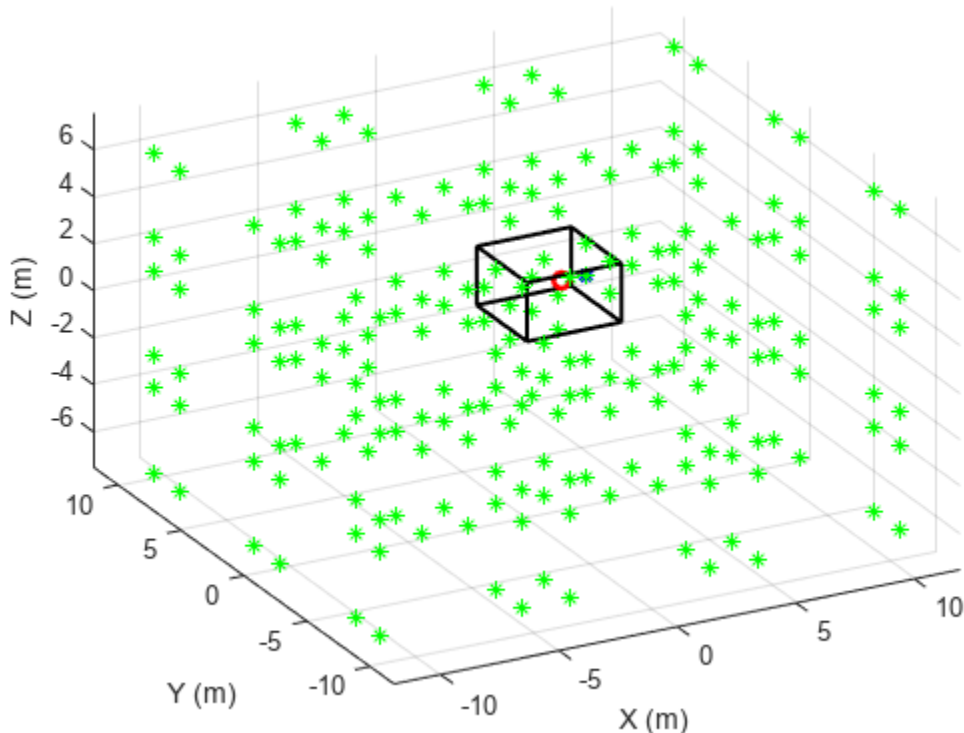
```
% Increase the range to plot more images
nVect = -2:2;
lVect = -2:2;
mVect = -2:2;

for n = nVect
```

```

for l = lVect
  for m = mVect
    xyz = [n*2*Lx; l*2*Ly; m*2*Lz];
    isourceCoords = xyz - sourceXYZ;
    for kk=1:8
      isourceCoord=isourceCoords(:,kk);
      plot3(isourceCoord(1),isourceCoord(2),isourceCoord(3),"g*")
    end
  end
end
end
end

```



Restricting the Number of Simulated Images

The number of images is theoretically infinite. Restrict the number of images by limiting the computed impulse response length to the time by which the reverberated sound pressure drops below a certain level. Here, you use the reverberation time RT_{60} [3] on page 1-844, which is the time by which the sound level has dropped by 60 dB.

You use Sabine's formula to calculate RT_{60} .

Define Wall Absorption Coefficients

First, define the absorption coefficients of the walls. The absorption coefficient is a measure of how much sound is absorbed (rather than reflected) when hitting a surface.

The absorption coefficients are frequency-dependent, and are defined at the frequencies defined in the variable $FVect$ [4] on page 1-844.

```
FVect = [125 250 500 1000 2000 4000];  
A = [0.10 0.20 0.40 0.60 0.50 0.60; ...  
     0.10 0.20 0.40 0.60 0.50 0.60; ...  
     0.10 0.20 0.40 0.60 0.50 0.60; ...  
     0.10 0.20 0.40 0.60 0.50 0.60; ...  
     0.02 0.03 0.03 0.03 0.04 0.07; ...  
     0.02 0.03 0.03 0.03 0.04 0.07].';
```

Estimate RT60

Compute RT60 based on Sabine's formula.

First, compute the room's volume.

```
V = Lx*Ly*Lz;
```

Next, compute the total wall surface area of the room.

```
WallXZ = Lx*Lz;  
WallYZ = Ly*Lz;  
WallXY = Lx*Ly;
```

Compute the frequency-dependent effective absorbing area of the room surfaces.

```
S = WallYZ*(A(:,1)+A(:,2))+WallXZ.*(A(:,3)+A(:,4))+WallXY.*(A(:,5)+A(:,6));
```

Compute the frequency-dependent RT60, in seconds, based on Sabine's equation. Notice that RT60 is frequency-dependent: There are 6 different RT60 values, one for each frequency band.

```
c = 343; % Speed of sound (m/s)  
RT60 = (55.25/c)*V./S
```

```
RT60 = 6×1
```

```
1.3886  
0.7191  
0.3799  
0.2581  
0.3028  
0.2455
```

Deduce the maximum impulse response length (in samples) based on the largest value in RT60. Assume a sample rate of 48 kHz.

```
fs = 48000;  
impResLength = fix(max(RT60)*fs)
```

```
impResLength = 66653
```

Express the maximum range of the impulse response in meters.

```
impResRange=c*(1/fs)*impResLength
```

```
impResRange = 476.2912
```

Use this value to limit the range over which to compute images. In this example, to limit the run time, you restrict the loop ranges to [-10;10].


```
nMax = min(ceil(impResRange./(2.*Lx)),10);
lMax = min(ceil(impResRange./(2.*Ly)),10);
mMax = min(ceil(impResRange./(2.*Lz)),10);
```

Derive Contribution of One Image

In this section, you derive the contribution of one image to the room impulse response.

You later obtain the full room impulse response by summing the contributions of all images under consideration.

Derive the pressure reflection coefficients from the absorption coefficients.

```
B=sqrt(1-A);
```

Store the reflection coefficients for each wall in a separate variable.

```
BX1=B(:,1);
BX2=B(:,2);
BY1=B(:,3);
BY2=B(:,4);
BZ1=B(:,5);
BZ2=B(:,6);
```

Model the eight permutations representing the absence or presence of reflection on the x-, y-, and z-axes.

```
surface_coeff=[0 0 0; 0 0 1; 0 1 0; 0 1 1; 1 0 0; 1 0 1; 1 1 0; 1 1 1];
q = surface_coeff(:,1).';
j = surface_coeff(:,2).';
k = surface_coeff(:,3).';
```

In this section, you focus on the contribution of a single image. Select index values corresponding to an arbitrary image.

```
n = 1;
l = 1;
m = 1;
p = 1;
```

Compute Image Delay

The contribution of each image is defined by two values:

- Delay: The time it takes the signal to reach the receiver.
- Power: The (frequency-dependent) energy level of the signal when it reaches the receiver.

You start by computing the image delay. The delay is related to the total distance traveled by the wave from the image to the receiver.

Get the coordinates of the image.

```
isourceCoord = [n*2*Lx; l*2*Ly; m*2*Lz] - sourceXYZ(:,p);
```

Calculate the delay (in samples) at which the contribution occurs.

```
dist = norm((isourceCoord(:)-receiverCoord(:)),2);
delay = (fs/c).*dist;
```

Compute Image Power

Now compute the frequency-dependent magnitude of the contribution.

```
ImagePower = BX1.^abs(n-q(p)).*BY1.^abs(l-j(p)).*BZ1.^abs(m-k(p)).*BX2.^abs(n).*(BY2.^abs(l)).*(BZ2.^abs(m)).*
```

Derive Image Contribution

The image power is only defined at 6 frequencies. Here, you first interpolate the response to the entire frequency (Nyquist) range, and then perform an inverse FFT operation to derive the image's contribution to the impulse response.

Extend the energy level to incorporate zero and Nyquist frequencies.

```
FVect2=[0 FVect fs/2]';  
ImagePower2 = [ImagePower(1); ImagePower(:); ImagePower(6)];
```

In this example, use an FFT length of 512.

```
FFTLength = 512;  
HalfLength = fix(FFTLength./2);  
OneSidedLength = HalfLength+1;
```

Interpolate the response to the entire frequency range.

```
ImagePower2 = interp1(FVect2./(fs/2),ImagePower2,linspace(0,1,257)).';
```

Convert the response to two-sided.

```
ImagePower2 = [ImagePower2; conj(ImagePower2(HalfLength:-1:2))];
```

Convert the frequency response to the time-based contribution of the image.

```
h_ImagePower = real(ifft(ImagePower2,FFTLength));
```

Smooth the response by applying a Hann window.

```
win = hann(FFTLength+1);  
h_ImagePower = win.*[h_ImagePower(OneSidedLength:FFTLength); ...  
h_ImagePower(1:OneSidedLength)];
```

HRTF Modeling

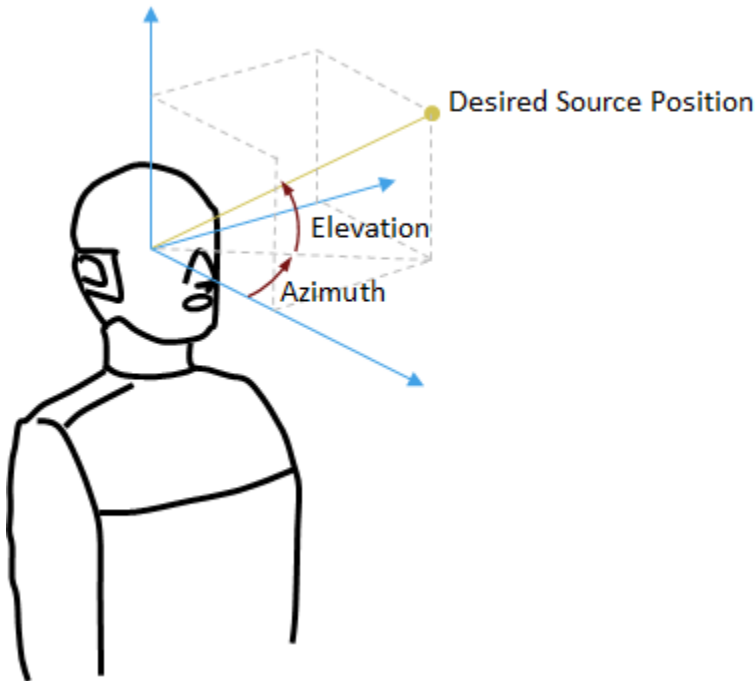
You have derived the image's contribution to the impulse response, where you assumed that the receiver is a point in space. Here, you derive the image's contribution at the ears of a listener located at the receiver coordinates by using 3-D head-related transfer function (HRTF) interpolation.

You use the ARI HRTF data set [5] on page 1-844. Load the data set.

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

Express the `hrtfData` as an array of size (Number of source measurements) \times 2 \times (Sample lengths).

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



Calculate the elevation and azimuth corresponding to the coordinates of the image source.

```
sensor_xyz=receiverCoord;
xyz=isourceCoord-sensor_xyz;
hyp = sqrt(xyz(1)^2+xyz(2)^2);
elevation = atan(xyz(3)./(hyp+eps));
azimuth = atan2(xyz(2),xyz(1));
```

The desired HRTF position is formed by the computed elevation and azimuth.

```
desiredPosition = [azimuth elevation]*180/pi;
```

Calculate the HRTF at the desired position.

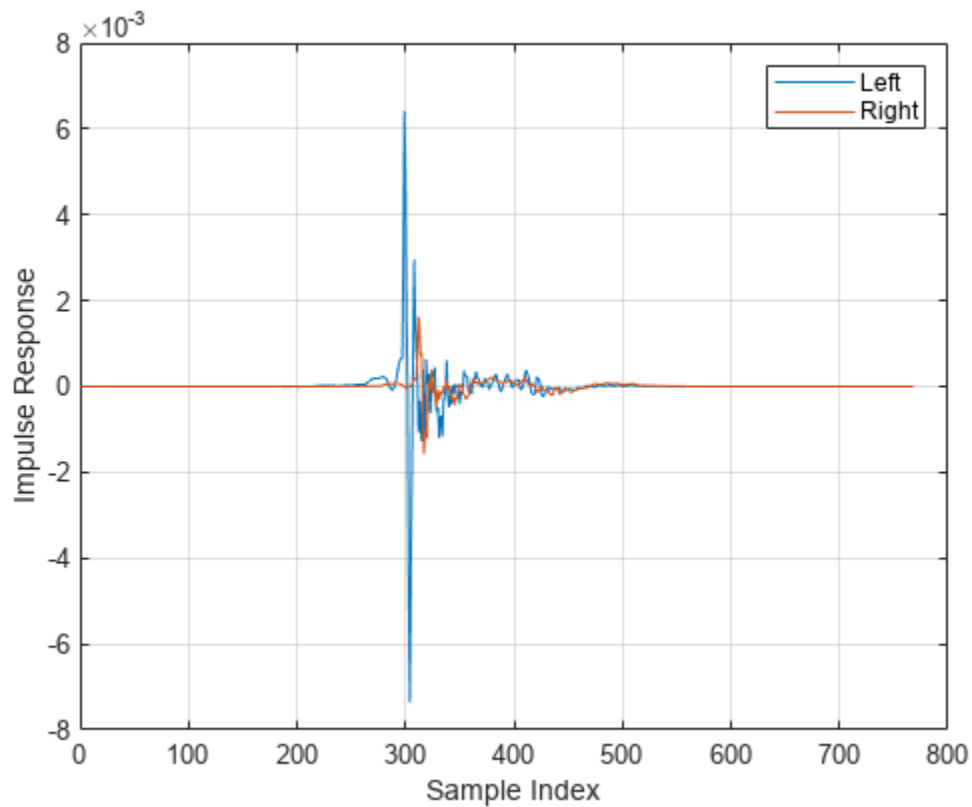
```
interpolatedIR = interpolateHRTF(hrtfData,sourcePosition,desiredPosition);
interpolatedIR = squeeze(permute(interpolatedIR,[3 2 1]));
```

Incorporate the HRTF into the response using convolution.

```
interpolatedIR = [interpolatedIR; zeros(512,2)];
h = [1];
h(:,1) = filter(h_ImagePower,1,interpolatedIR(:,1));
h(:,2) = filter(h_ImagePower,1,interpolatedIR(:,2));
```

Plot the overall contribution of the selected image. This contribution is added to the overall impulse response at the computed image delay.

```
figure
plot(1:size(h,1),h)
grid on
xlabel("Sample Index")
ylabel("Impulse Response")
legend("Left", "Right")
```



Compute The Impulse Response

In this section, you compute the overall impulse response by summing the contributions of individual images. The contribution of each image is computed exactly like in the previous section.

The helper function `HelperImageSource` encapsulates the steps you went over in the previous section. It computes the impulse response by summing image contributions.

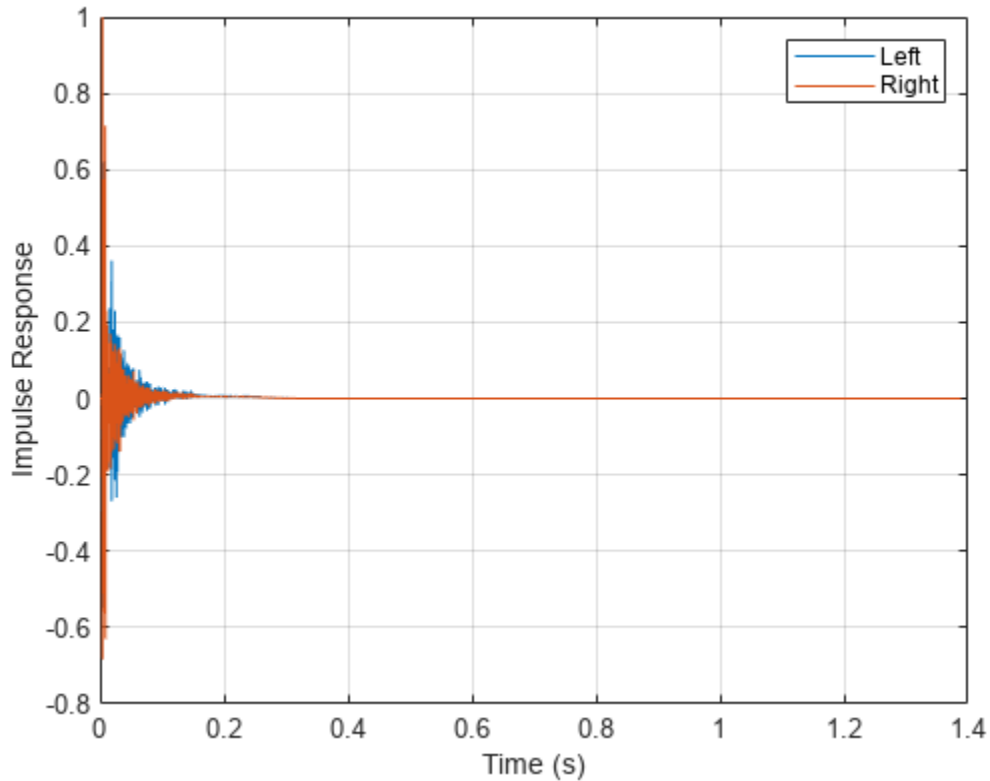
```
useHRTF = true;
h = HelperImageSource(roomDimensions,receiverCoord,sourceCoord,A,FVect,fs,useHRTF,hrtfData,sourceCoord);
```

```
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to parallel pool with 20 workers.
```

Visualize Impulse Response

Plot the impulse response.

```
figure
t= (1/fs)*(0:size(h,1)-1);
plot(t,h)
grid on
xlabel("Time (s)")
ylabel("Impulse Response")
legend("Left","Right")
```



Auralization

Apply the impulse response to an audio signal.

Load an audio signal.

```
[audioIn,fs] = audioread("FunkyDrums-44p1-stereo-25secs.mp3");
audioIn = audioIn(:,1);
```

Simulate the received audio by filtering with the impulse response.

```
y1 = filter(h(:,1),1,audioIn);
y2 = filter(h(:,2),1,audioIn);
y = [y1 y2];
```

Listen to a few seconds of the original audio.

```
T = 10;
sound(audioIn(1:fs*T), fs)
pause(T)
```

Listen to a few seconds of the received audio.

```
sound(y(1:fs*T), fs);
```

References

[1] "Overview of geometrical room acoustic modeling techniques", Lauri Savioja, Journal of the Acoustical Society of America 138, 708 (2015).

[2] Allen, J and Berkley, D. "Image Method for efficiently simulating small-room acoustics", The Journal of the Acoustical Society of America, Vol 65, No.4, pp. 943-950, 1978.

[3] <https://www.sciencedirect.com/topics/engineering/sabine-equation>

[4] https://www.acoustic.ua/st/web_absorption_data_eng.pdf

[5] HRTF Database

Helper Functions

```
function plotRoom(roomDimensions,receiverCoord,sourceCoord,figHandle)
% PLOTROOM Plot room, transmitter and receiver

figure(figHandle)
X = [0;roomDimensions(1);roomDimensions(1);0;0];
Y = [0;0;roomDimensions(2);roomDimensions(2);0];
Z = [0;0;0;0;0];
figure;
hold on;
plot3(X,Y,Z,"k","LineWidth",1.5); % draw a square in the xy plane with z = 0
plot3(X,Y,Z+roomDimensions(3),"k","LineWidth",1.5); % draw a square in the xy plane with z = 1
set(gca,"View",[-28,35]); % set the azimuth and elevation of the plot
for k=1:length(X)-1
    plot3([X(k);X(k)], [Y(k);Y(k)], [0;roomDimensions(3)], "k", "LineWidth", 1.5);
end
grid on
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
plot3(sourceCoord(1),sourceCoord(2),sourceCoord(3),"bx","LineWidth",2)
plot3(receiverCoord(1),receiverCoord(2),receiverCoord(3),"ro","LineWidth",2)
end

function h = HelperImageSource(roomDimensions,receiverCoord, ....
                               sourceCoord,A,FVect,fs,useHRTF,varargin)
% HELPERIMAGESOURCE Estimate impulse response of shoebox room
% roomDimensions: Room dimensions, specified as a row vector with three
%                 values.
% receiverCoord: Receiver coordinates, specified as a row vector with 3
%               values
% sourceCoord: Source coordinates, specified as a row vector with 3
%             values
% A: Wall absorption coefficient matrix, specified as a L-by-6 matrix,
%   where L is the number of frequency bands.
% FVect: Vector of frequencies, of length L.
% fs: Sampling rate, in Hertz
% useHRTF: Specify as true to use HRTF interpolation
% hrtfData: Specify is useHRTF is true
% sourcePosition: Specify is useHRTF is true
```

```

hrtfData = [];
sourcePosition = [];
if useHRTF
    hrtfData = varargin{1};
    sourcePosition = varargin{2};
end

x = sourceCoord(1);
y = sourceCoord(2);
z = sourceCoord(3);
sourceXYZ = [-x -y -z; ...
             -x -y  z; ...
             -x  y -z; ...
             -x  y  z; ...
              x -y -z; ...
              x -y  z; ...
              x  y -z; ...
              x  y  z].';

Lx=roomDimensions(1);
Ly=roomDimensions(2);
Lz=roomDimensions(3);

V = Lx*Ly*Lz;

WallXZ=Lx*Lz;
WallYZ=Ly*Lz;
WallXY=Lx*Ly;

S = WallYZ*(A(:,1)+A(:,2))+WallXZ.*(A(:,3)+A(:,4))+WallXY.*(A(:,5)+A(:,6));

c = 343; % Speed of sound (m/s)
RT60 = (55.25/c)*V./S;

impResLength = fix(max(RT60)*fs);

impResRange=c*(1/fs)*impResLength;

nMax = min(ceil(impResRange./(2.*Lx)),10);
lMax = min(ceil(impResRange./(2.*Ly)),10);
mMax = min(ceil(impResRange./(2.*Lz)),10);

B=sqrt(1-A);

BX1=B(:,1);
BX2=B(:,2);
BY1=B(:,3);
BY2=B(:,4);
BZ1=B(:,5);
BZ2=B(:,6);

surface_coeff=[0 0 0; 0 0 1; 0 1 0; 0 1 1; 1 0 0; 1 0 1; 1 1 0; 1 1 1];
q=surface_coeff(:,1).';
j=surface_coeff(:,2).';
k=surface_coeff(:,3).';

FFTLenght=512;

```

```
HalfLength=fix(FFTLength./2);
OneSidedLength = HalfLength+1;
win = hann(FFTLength+1);

FVect2=[0 FVect fs/2]';

h = zeros(impResLength,2);

for n=-nMax:nMax
    Lxn2=n*2*Lx;
    for l=-lMax:lMax
        Lyl2=l*2*Ly;

        if useHRTF
            imagesVals = zeros(FFTLength+size(hrtfData,3),2,2*lMax+1,8);
        else
            imagesVals = zeros(FFTLength+1,2,2*lMax+1,8);
        end

        Li = size(imagesVals,1);
        isDelayValid = zeros(2*lMax+1,8);
        start_index_HpV = zeros(2*lMax+1,8);
        stop_index_HpV = zeros(2*lMax+1,8);
        start_index_hV = zeros(2*lMax+1,8);

        parfor mInd=1:2*mMax+1

            m = mInd - mMax - 1;

            Lzm2=m*2*Lz;
            xyz = [Lxn2; Lyl2; Lzm2];

            isourceCoordV=xyz - sourceXYZ;
            xyzV = isourceCoordV - receiverCoord.';
            distV = sqrt(sum(xyzV.^2));
            delayV = (fs/c)*distV;

            ImagePower = BX1.^abs(n-q).*BY1.^abs(l-j).*BZ1.^abs(m-k).*BX2.^abs(n).*(BY2.^abs(l))
            ImagePower2 = [ImagePower(1,:); ImagePower; ImagePower(6,:)];

            ImagePower2 = ImagePower2./distV;

            validDelay = delayV<= impResLength;

            if sum(validDelay)==0
                continue;
            end

            isDelayValid(mInd,:) = validDelay;

            ImagePower2 = interp1(FVect2./(fs/2),ImagePower2,linspace(0,1,257));
            if isrow(ImagePower2)
                ImagePower2 = ImagePower2.';
            end
            ImagePower3 = [ImagePower2; conj(ImagePower2(HalfLength:-1:2,:))];

            h_ImagePower = real(ifft(ImagePower3,FFTLength));
            h_ImagePower = [h_ImagePower(OneSidedLength:FFTLength,:); h_ImagePower(1:OneSidedLength
```



```

h_ImagePower = win.*h_ImagePower;

if useHRTF
    hyp = sqrt(xyzV(1,:).^2+xyzV(2,:).^2);
    elevation = atan(xyzV(3,:)./(hyp+realmin));
    azimuth = atan2(xyzV(2,:),xyzV(1,:));

    desiredPosition = [azimuth.',elevation.']*180/pi;

    interpolatedIR = interpolateHRTF(hrtfData,sourcePosition,desiredPosition,"Algorithm");
    interpolatedIR = squeeze(permute(interpolatedIR,[3 2 1]));

    pad_ImagePower = zeros(512,2);

    for index=1:8
        hrir0 = interpolatedIR(:,:,index);
        hrir_ext=[hrir0; pad_ImagePower];
        for ear=1:2
            imagesVals(:,ear,mInd,index)=filter(h_ImagePower(:,index),1,hrir_ext(:,ear));
        end
    end
else
    for index=1:8
        for ear=1:2
            imagesVals(:,ear,mInd,index)=h_ImagePower(:,index);
        end
    end
end

adjust_delay = round(delayV) - (fix(FFTLength/2))+1;

len_h=Li;
start_index_HpV(mInd,:) = max(adjust_delay+1+(adjust_delay>=0),1);
stop_index_HpV(mInd,:) = min(adjust_delay+1+len_h,impResLength);
start_index_hV(mInd,:) = max(-adjust_delay,1);

end
stop_index_hV = start_index_hV + (stop_index_HpV - start_index_HpV);

for index2=1:size(imagesVals,3)
    for index3=1:8
        if isDelayValid(index2,index3)
            h(start_index_HpV(index2,index3):stop_index_HpV(index2,index3),:)= h(start_index_hV(index2,index3):stop_index_hV(index2,index3),:);
        end
    end
end

end
end

h = h./max(abs(h));
end

```

Room Impulse Response Simulation with Stochastic Ray Tracing

Room impulse response simulation aims to model the reverberant properties of a space without having to perform acoustic measurements. Many geometric and wave-based room acoustic simulation methods exist in the literature [1] on page 1-856.

The image-source method is a popular geometric method (for an example, see “Room Impulse Response Simulation with the Image-Source Method and HRTF Interpolation” on page 1-833). One drawback of the image-source method is that it only models specular reflections between a transmitter and a receiver. There are other geometric methods that address this limitation by also taking sound diffusion and diffraction into account. Stochastic ray tracing is one such method.

This example showcases a stochastic ray tracing method for a simple "shoebox" (cuboid) room.

Stochastic Ray Tracing Overview

Ray tracing assumes that sound energy travels around the room in rays. The rays start at the sound source, and are emitted in all directions, following a uniform random distribution. In this example, you follow (trace) rays as they bounce off obstacles (walls, floor and ceiling) in the room. At each ray reflection, you compute the measured ray energy at the receiver. You use the measured energy to update a frequency-dependent histogram. You then compute the room impulse response by weighting a Poisson random process by the histogram values [2] on page 1-856.

Define Room Parameters

Simulate the impulse response of the shoebox empty room.

Define the room dimensions, in meters (width, length, and height, respectively).

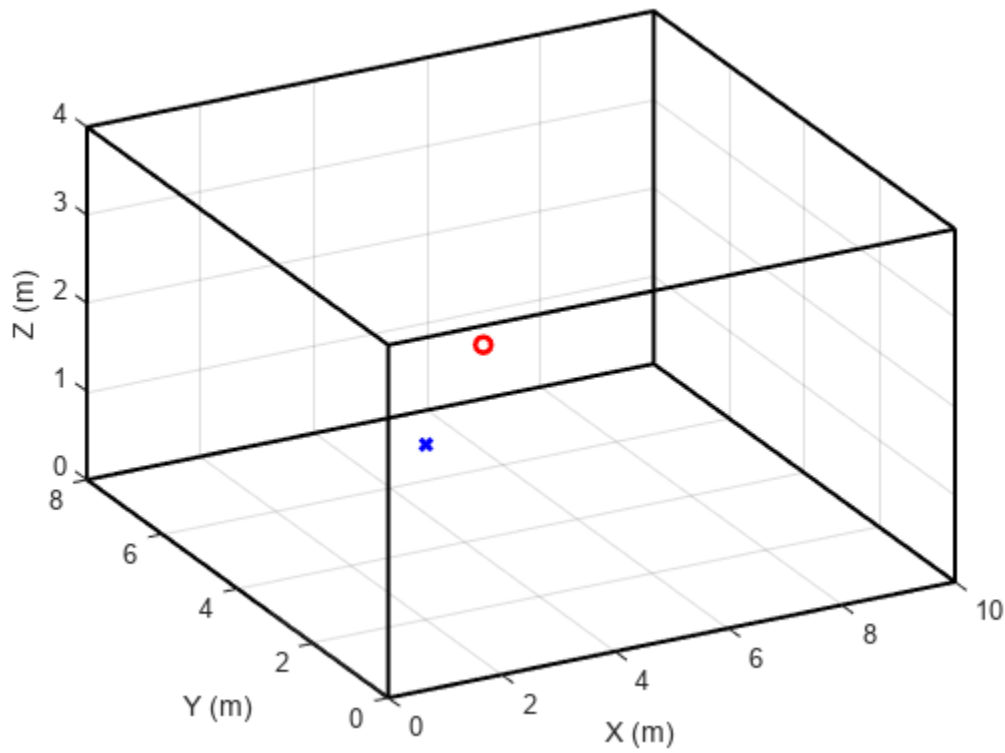
```
roomDimensions = [10 8 4];
```

Treat the transmitter as a point within the space of the room. Assume that the receiver is a sphere with radius 8.75 cm.

```
sourceCoord = [2 2 2];  
receiverCoord = [5 5 1.8];  
r = 0.0875;
```

Plot the room space along with the receiver (red circle) and transmitter (blue x).

```
h = figure;  
plotRoom(roomDimensions, receiverCoord, sourceCoord, h)
```



Generate Random Rays

First, generate rays emanating from the source in random directions.

Set the number of rays.

```
N = 5000;
```

Generate the rays using the helper function `RandSampleSphere`. `rays` is a N -by-3 matrix. Each row of `rays` holds the three-dimensional ray vector direction.

```
rng(0)
rays = RandSampleSphere(N);
size(rays)
```

```
ans = 1×2
```

```
5000    3
```

Define Reflection and Scattering Coefficients

A sound ray is reflected when it hits a surface. The reflection is a combination of a specular component and a diffused component. The relative strength of each component is determined by the reflection and scattering coefficients of the surfaces.

Define the absorption coefficients of the walls. The absorption coefficient is a measure of how much sound is absorbed (rather than reflected) when hitting a surface.

The frequency-dependent absorption coefficients are defined at the frequencies in the variable `FVect` [4] on page 1-857.

```
FVect = [125 250 500 1000 2000 4000];
```

```
A = [0.08 0.09 0.12 0.16 0.22 0.24;  
     0.08 0.09 0.12 0.16 0.22 0.24;  
     0.08 0.09 0.12 0.16 0.22 0.24;  
     0.08 0.09 0.12 0.16 0.22 0.24;  
     0.08 0.09 0.12 0.16 0.22 0.24;  
     0.08 0.09 0.12 0.16 0.22 0.24].';
```

Derive the reflection coefficients of the six surfaces from the absorption coefficients.

```
R = sqrt(1-A);
```

Define the frequency-dependent scattering coefficients [5] on page 1-857. The scattering coefficient is defined as one minus the ratio between the specularly reflected acoustic energy and the total reflected acoustic energy.

```
D = [0.05 0.3 0.7 0.9 0.92 0.94;  
     0.05 0.3 0.7 0.9 0.92 0.94;  
     0.05 0.3 0.7 0.9 0.92 0.94;  
     0.05 0.3 0.7 0.9 0.92 0.94;  
     0.01 0.05 0.1 0.2 0.3 0.5;  
     0.01 0.05 0.1 0.2 0.3 0.5];
```

Initialize Energy Histogram

As you trace the rays, you update a two-dimensional histogram of the energy detected at the receiver. The histogram records values along time and frequency.

Set the histogram time resolution, in seconds. The time resolution is typically much larger than the inverse of the audio sample rate.

```
histTimeStep = 0.0010;
```

Compute the number of histogram time bins. In this example, limit the impulse response length to one second.

```
impResTime = 1;  
nTBins = round(impResTime/histTimeStep);
```

The ray tracing algorithm is frequency-selective. In this example, focus on six frequency bands, centered around the frequencies in `FVect`.

The number of frequency bins in the histogram is equal to the number of frequency bands.

```
nFBins = length(FVect);
```

Initialize the histogram.

```
TFHist = zeros(nTBins,nFBins);
```

Perform Ray Tracing

Compute the received energy histogram by tracing the rays over each frequency band. When a ray hits a surface, record the amount of diffused ray energy seen at the receiver based on the diffused

rain model [2] on page 1-856. The new ray direction upon hitting a surface is a combination of a specular reflection and a random reflection. Continue tracing the ray until its travel time exceeds the impulse response duration.

```

for iBand = 1:nFBins
    % Perform ray tracing independently for each frequency band.
    for iRay = 1:size(rays,1)
        % Select ray direction
        ray = rays(iRay,:);
        % All rays start at the source/transmitter
        ray_xyz = sourceCoord;
        % Set initial ray direction. This direction changes as the ray is
        % reflected off surfaces.
        ray_dxyz = ray;
        % Initialize ray travel time. Ray tracing is terminated when the
        % travel time exceeds the impulse response length.
        ray_time = 0;
        % Initialize the ray energy to a normalized value of 1. Energy
        % decreases when the ray hits a surface.
        ray_energy = 1;

        while (ray_time <= impResTime)

            % Determine the surface that the ray encounters
            [surfaceofimpact,displacement] = getImpactWall(ray_xyz,...
                ray_dxyz,roomDimensions);

            % Determine the distance traveled by the ray
            distance = sqrt(sum(displacement.^2));

            % Determine the coordinates of the impact point
            impactCoord = ray_xyz+displacement;

            % Update ray location/source
            ray_xyz = impactCoord;

            % Update cumulative ray travel time
            c = 343; % speed of light (m/s)
            ray_time = ray_time+distance/c;

            % Apply surface reflection to ray's energy
            % This is the amount of energy that is not lost through
            % absorption.
            ray_energy = ray_energy*R(surfaceofimpact,iBand);

            % Apply diffuse reflection to ray energy
            % This is the fraction of energy used to determine what is
            % detected at the receiver
            rayrecv_energy = ray_energy*D(surfaceofimpact,iBand);

            % Determine impact point-to-receiver direction.
            rayrecvvector = receiverCoord-impactCoord;

            % Determine the ray's time of arrival at receiver.
            distance = sqrt(sum(rayrecvvector.*rayrecvvector));
            recv_timeofarrival = ray_time+distance/c;

            if recv_timeofarrival>impResTime

```

```

        break
    end

    % Determine amount of diffuse energy that reaches the receiver.
    % See (5.20) in [2].

    % Compute received energy
    N = getWallNormalVector(surfaceofimpact);
    cosTheta = sum(rayrecvvector.*N)/(sqrt(sum(rayrecvvector.^2)));
    cosAlpha = sqrt(sum(rayrecvvector.^2)-r^2)/sum(rayrecvvector.^2);
    E = (1-cosAlpha)*2*cosTheta*rayrecv_energy;

    % Update energy histogram
    tbin = floor(recv_timeofarrival/histTimeStep + 0.5);
    TFHist(tbin,iBand) = TFHist(tbin,iBand) + E;

    % Compute a new direction for the ray.
    % Pick a random direction that is in the hemisphere of the
    % normal to the impact surface.
    d = rand(1,3);
    d = d/norm(d);
    if sum(d.*N)<0
        d = -d;
    end

    % Derive the specular reflection with respect to the incident
    % wall
    ref = ray_dxyz-2*(sum(ray_dxyz.*N))*N;

    % Combine the specular and random components
    d = d/norm(d);
    ref = ref/norm(ref);
    ray_dxyz = D(surfaceofimpact,iBand)*d+(1-D(surfaceofimpact,iBand))*ref;
    ray_dxyz = ray_dxyz/norm(ray_dxyz);
end
end
end
end

```

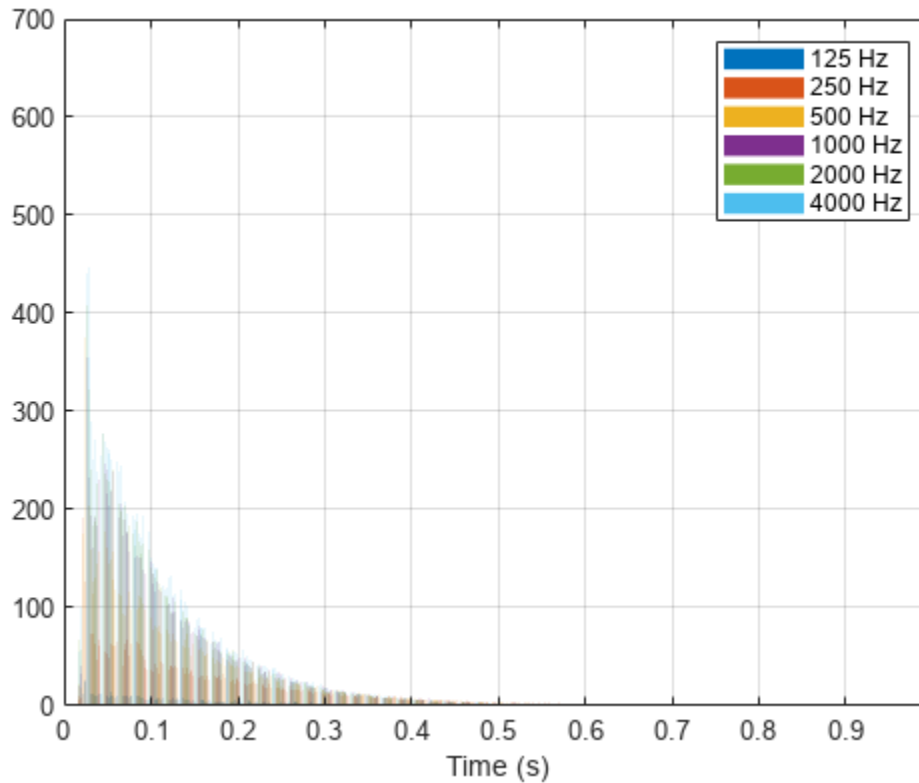
View Energy Histogram

Plot the computed frequency-dependent histogram.

```

figure
bar(histTimeStep*(0:size(TFHist)-1),TFHist)
grid on
xlabel("Time (s)")
legend(["125 Hz", "250 Hz", "500 Hz", "1000 Hz", "2000 Hz", "4000 Hz"])

```



Generate Room Impulse Response

The energy histogram represents the envelope of the room impulse response. Synthesize the impulse response using a Poisson-distributed noise process [2] on page 1-856.

Generate Poisson Random Process

Define the audio sample rate (in Hz).

```
fs = 44100;
```

You model sound reflections as a Poisson random process.

Define the start time of the process.

```
V = prod(roomDimensions);
t0 = ((2*V*log(2))/(4*pi*c^3))^(1/3); % eq 5.45 in [2]
```

Initialize the random process vector and the vector containing the times at which events occur.

```
poissonProcess = [];
timeValues = [];
```

Create the random process.

```
t = t0;
while (t < impResTime)
    timeValues = [timeValues t]; %#ok
```

```

% Determine polarity.
if (round(t*fs)-t*fs) < 0
    poissonProcess = [poissonProcess 1]; %#ok
else
    poissonProcess = [poissonProcess -1];%#ok
end
% Determine the mean event occurrence (eq 5.44 in [2])
mu = min(1e4,4*pi*c^3*t^2/V);
% Determine the interval size (eq. 5.44 in [2])
deltaTA = (1/mu)*log(1/rand); % eq. 5.43 in [2])
t = t+deltaTA;
end

```

Create a random process sampled at the specified sample rate.

```

randSeq = zeros(ceil(impResTime*fs),1);
for index=1:length(timeValues)
    randSeq(round(timeValues(index)*fs)) = poissonProcess(index);
end

```

Pass Poisson Process Through Bandpass Filters

You create the impulse response by passing the Poisson process through bandpass filters centered at the frequencies in `FVect`, and then weighting the filtered signals with the received energy envelope computed in the histogram.

Define the lower and upper cutoff frequencies of the bandpass filters.

```

flow = [115 225 450 900 1800 3600];
fhigh = [135 275 550 1100 2200 4400];

```

Set the FFT length.

```
NFFT = 8192;
```

Create the short-time Fourier transform and inverse short-time Fourier transform objects you will use to filter the Poisson process and reconstruct it. Use a Hann window with 50% overlap.

```

win = hann(882, "symmetric");
sfft = dsp.STFT(Window = win,OverlapLength=441,FFTLength=NFFT,FrequencyRange="onesided");
isfft = dsp.ISTFT(Window=win,OverlapLength=441,FrequencyRange="onesided");
F = sfft.getFrequencyVector(fs);

```

Create the bandpass filters (use equation 5.46 in [2] on page 1-856).

```

RCF = zeros(length(FVect),length(F));
for index0 = 1:length(FVect)
    for index=1:length(F)
        f = F(index);
        if f<FVect(index0) && f>=flow(index0)
            RCF(index0,index) = .5*(1+cos(2*pi*f/FVect(index0)));
        end
        if f<fhigh(index0) && f>=FVect(index0)
            RCF(index0,index) = .5*(1-cos(2*pi*f/(FVect(index0)+1)));
        end
    end
end
end

```


Filter the Poisson sequence through the six bandpass filters.

```
frameLength = 441;
numFrames = length(randSeq)/frameLength;
y = zeros(length(randSeq),6);
for index=1:numFrames
    x = randSeq((index-1)*frameLength+1:index*frameLength);
    X = sfft(x);
    X = X.*RCF.';
    y((index-1)*frameLength+1:index*frameLength,:) = isfft(X);
end
```

Combine the Filtered Sequences

Construct the impulse response by weighting the filtered random sequences sample-wise using the envelope (histogram) values.

Compute the times corresponding to the impulse response samples.

```
impTimes = (1/fs)*(0:size(y,1)-1);
```

Compute the times corresponding to the histogram bins.

```
hisTimes = histTimeStep/2 + histTimeStep*(0:nTBins);
```

Compute the weighting factors (equation 5.47 in [2] on page 1-856).

```
W = zeros(size(impTimes,2),numel(FVect));
BW = fhigh-flow;
for k=1:size(TFHist,1)
    gk0 = floor((k-1)*fs*histTimeStep)+1;
    gk1 = floor(k*fs*histTimeStep);
    yy = y(gk0:gk1,:).^2;
    val = sqrt(TFHist(k,:)./sum(yy,1)).*sqrt(BW/(fs/2));
    for iRay=gk0:gk1
        W(iRay,:)= val;
    end
end
```

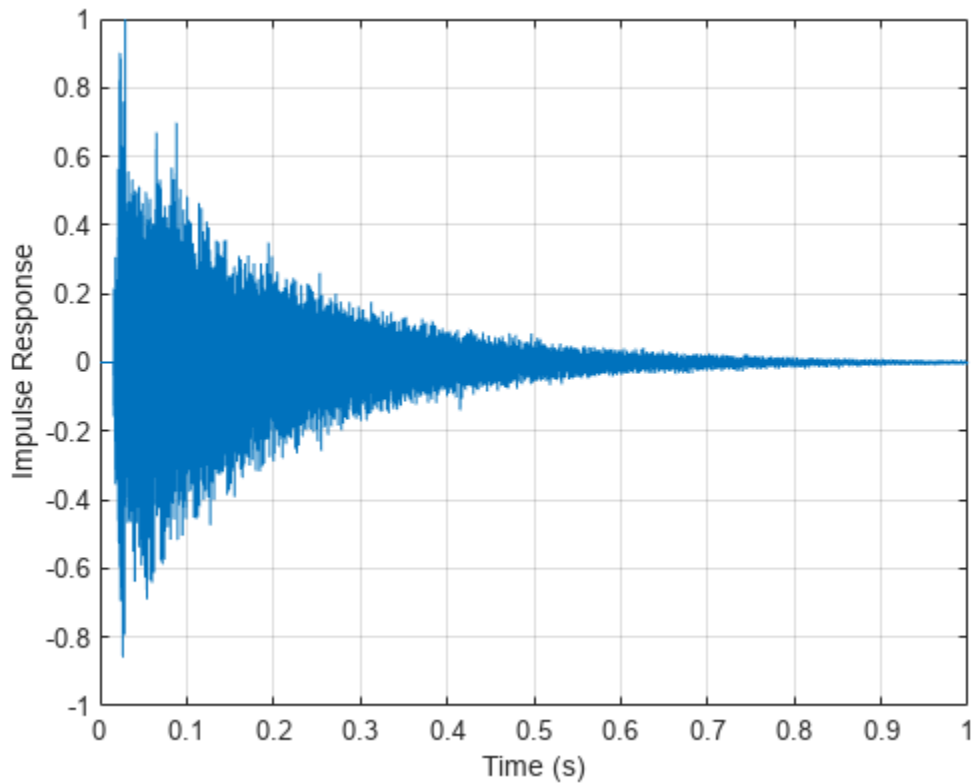
Create the impulse response.

```
y = y.*W;
ip = sum(y,2);
ip = ip./max(abs(ip));
```

Auralization

Plot the impulse response.

```
figure
plot((1/fs)*(0:numel(ip)-1),ip)
grid on
xlabel("Time (s)")
ylabel("Impulse Response")
```



Apply the impulse response to an audio signal.

```
[audioIn,fs] = audioread("FunkyDrums-44p1-stereo-25secs.mp3");  
audioIn = audioIn(:,1);
```

Simulate the received audio by filtering with the impulse response.

```
audioOut = filter(ip,1,audioIn);  
audioOut = audioOut/max(audioOut);
```

Listen to a few seconds of the original audio.

```
T = 10;  
sound(audioIn(1:T*fs), fs)  
pause(T)
```

Listen to a few seconds of the received audio.

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

References

[1] "Overview of geometrical room acoustic modeling techniques", Lauri Savioja, Journal of the Acoustical Society of America 138, 708 (2015).

[2] "*Physically Based Real-Time Auralization of Interactive Virtual Environments*", Dirk Schröder, Aachen, Techn. Hochsch., Diss., 2011.

[3] "*Auralization: Fundamentals of Acoustics, Modelling, Simulation, Algorithms and Acoustic Virtual Reality*", Michael Vorlander, Second Edition, Springer.

[4] https://www.acoustic.ua/st/web_absorption_data_eng.pdf

[5] "*Scattering in Room Acoustics and Related Activities in ISO and AES*", Jens Holger Rindel, 17th ICA Conference, Rome, Italy, September 2001.

Helper Functions

```
function plotRoom(roomDimensions, receiverCoord, sourceCoord, figHandle)
% PLOTROOM Helper function to plot 3D room with receiver/transmitter points
figure(figHandle)
X = [0;roomDimensions(1);roomDimensions(1);0;0];
Y = [0;0;roomDimensions(2);roomDimensions(2);0];
Z = [0;0;0;0;0];
figure;
hold on;
plot3(X,Y,Z,"k",LineWidth=1.5);
plot3(X,Y,Z+roomDimensions(3),"k",LineWidth=1.5);
set(gca,"View",[-28,35]);
for k=1:length(X)-1
    plot3([X(k);X(k)],[Y(k);Y(k)],[0;roomDimensions(3)],"k",LineWidth=1.5);
end
grid on
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
plot3(sourceCoord(1),sourceCoord(2),sourceCoord(3),"bx",LineWidth=2)
plot3(receiverCoord(1),receiverCoord(2),receiverCoord(3),"ro",LineWidth=2)
end

function X=RandSampleSphere(N)
% RANDSAMPLESPHERE Return random ray directions

% Sample the unfolded right cylinder
z = 2*rand(N,1)-1;
lon = 2*pi*rand(N,1);

% Convert z to latitude
z(z<-1) = -1;
z(z>1) = 1;
lat = acos(z);

% Convert spherical to rectangular co-ords
s = sin(lat);
x = cos(lon).*s;
y = sin(lon).*s;

X = [x y z];
end
```

```

function [surfaceofimpact,displacement] = getImpactWall(ray_xyz,ray_dxyz,roomDims)
% GETIMPACTWALL Determine which wall the ray encounters
surfaceofimpact = -1;
displacement = 1000;
% Compute time to intersection with x-surfaces
if (ray_dxyz(1) < 0)
    displacement = -ray_xyz(1) / ray_dxyz(1);
    if displacement==0
        displacement=1000;
    end
    surfaceofimpact = 0;
elseif (ray_dxyz(1) > 0)
    displacement = (roomDims(1) - ray_xyz(1)) / ray_dxyz(1);
    if displacement==0
        displacement=1000;
    end
    surfaceofimpact = 1;
end
% Compute time to intersection with y-surfaces
if ray_dxyz(2)<0
    t = -ray_xyz(2) / ray_dxyz(2);
    if (t<displacement) && t>0
        surfaceofimpact = 2;
        displacement = t;
    end
elseif ray_dxyz(2)>0
    t = (roomDims(2) - ray_xyz(2)) / ray_dxyz(2);
    if (t<displacement) && t>0
        surfaceofimpact = 3;
        displacement = t;
    end
end
% Compute time to intersection with z-surfaces
if ray_dxyz(3)<0
    t = -ray_xyz(3) / ray_dxyz(3);
    if (t<displacement) && t>0
        surfaceofimpact = 4;
        displacement = t;
    end
elseif ray_dxyz(3)>0
    t = (roomDims(3) - ray_xyz(3)) / ray_dxyz(3);
    if (t<displacement) && t>0
        surfaceofimpact = 5;
        displacement = t;
    end
end
surfaceofimpact = surfaceofimpact + 1;

displacement = displacement * ray_dxyz;

end

function N = getWallNormalVector(surfaceofimpact)
% GETWALLNORMALVECTOR Get the normal vector of a surface
switch surfaceofimpact
    case 1
        N = [1 0 0];
    case 2

```

```
        N = [-1 0 0];  
case 3  
        N = [0 1 0];  
case 4  
        N = [0 -1 0];  
case 5  
        N = [0 0 1];  
case 6  
        N = [0 0 -1];  
end  
end
```

Feature Selection for Audio Classification

Feature selection reduces the dimensionality of data by selecting a subset of measured features to create a model. Performing feature selection enables you to train smaller models quickly without sacrificing accuracy. For some tasks, properly selected features used with simple thresholding can provide adequate results, especially in situations where model size and complexity must be minimized.

In this example, you walk through a standard machine learning pipeline to develop an audio classification system. The pipeline has been abstracted so that you can apply the same steps to either speaker recognition or word recognition tasks.



Dataset Management and Labeling

Download the Free Spoken Digit Dataset (FSDD) [1] on page 1-871. FSDD consists of short audio files with spoken digits (0-9). The data is sampled at 8 kHz.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "FSDD.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
dataset = fullfile(dataFolder, "FSDD");
```

Create an `audioDatastore` to manage the audio dataset.

```
ads = audioDatastore(dataset, IncludeSubfolders=true);
```

Choose a task and set the `audioDatastore` labels accordingly.

```
task = ;
[~, filenames] = fileparts(ads.Files);
switch task
    case "speaker recognition"
        ads.Labels = extractBetween(filenames, "_", "_");
    case "word recognition"
        ads.Labels = extractBefore(filenames, "_");
end
```

Split data into train and test sets. Use 80% for training and 20% for testing.

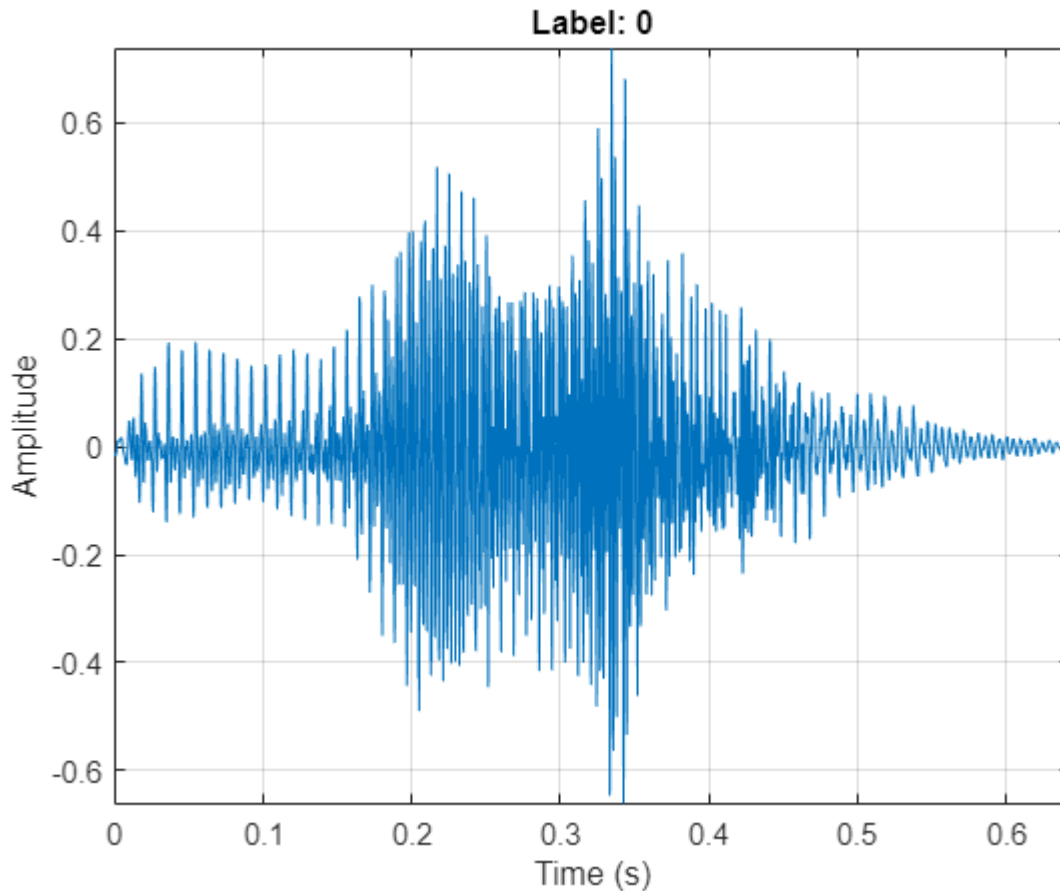
```
[adsTrain, adsTest] = splitEachLabel(ads, 0.8);
```

Listen to a sample from the training set. Plot the waveform and display the associated label.

```
[x, xinfo] = read(adsTrain);
sound(x, xinfo.SampleRate)

t = (0: numel(x)-1)/xinfo.SampleRate;
figure
plot(t, x)
```

```
title("Label: " + xinfo.Label)
grid on
axis tight
ylabel("Amplitude")
xlabel("Time (s)")
```



Feature Extraction Pipeline

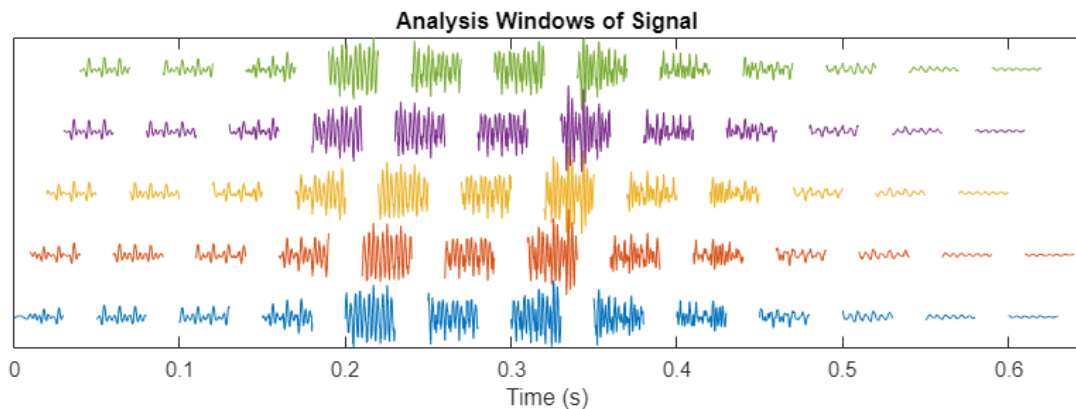
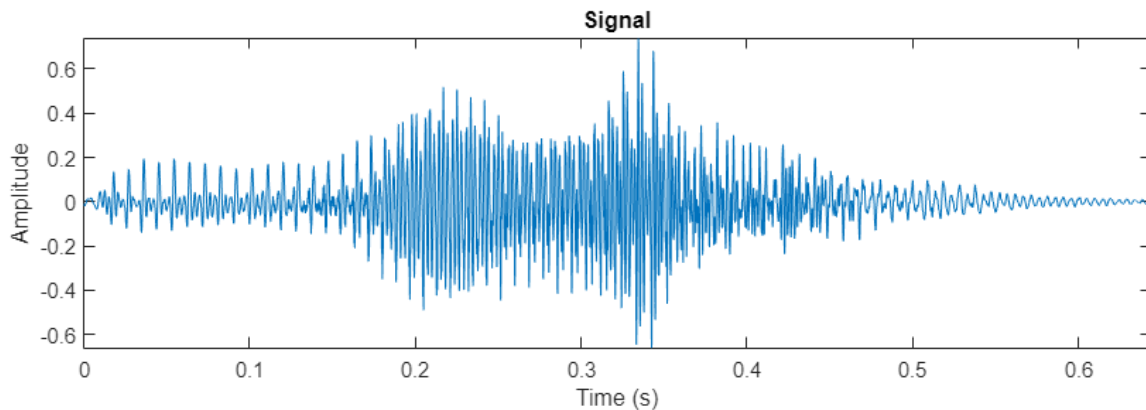
Audio signals can broadly be categorized as stationary or non-stationary. Stationary signals have spectrums that do not change over time, like pure tones. Non-stationary signals have spectrums that change over time, like speech signals. To make machine learning-based tasks tractable, non-stationary signals can be approximated as stationary when analyzed at appropriately small time scales. Generally, speech signals are considered stationary when viewed at time scales around 30 ms. Therefore, speech can be characterized by extracting features from 30 ms analysis windows over time.

Use the helper function, `helperVisualizeBuffer`, to visualize the analysis windows of an audio file. Specify a 30 ms analysis window with 20 ms overlap between adjacent windows. The overlap duration must be less than the window duration. The **Analysis Windows of Signal** plot shows the individual analysis windows from which features are extracted.

```

windowDuration = 0.03 ;
overlapDuration = 0.02 ;
helperVisualizeBuffer(x,xinfo.SampleRate,WindowDuration=windowDuration,OverlapDuration=overlapDuration);

```



Create an `audioFeatureExtractor` to extract features from 30 ms windows with 20 ms overlap between windows.

```

afe = audioFeatureExtractor(SampleRate=xinfo.SampleRate, ...
    Window=hann(round(windowDuration*xinfo.SampleRate),"periodic"), ...
    OverlapLength=round(overlapDuration*xinfo.SampleRate))

```

```

afe =
    audioFeatureExtractor with properties:

        Properties
            Window: [240x1 double]
            OverlapLength: 160
            SampleRate: 8000
            FFTLength: []
            SpectralDescriptorInput: 'linearSpectrum'
            FeatureVectorLength: 0

```

```

Enabled Features

```



```
none
```

Disabled Features

```
linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
mfccDeltaDelta, gtcc, gtccDelta, gtccDeltaDelta, spectralCentroid, spectralCrest
spectralDecrease, spectralEntropy, spectralFlatness, spectralFlux, spectralKurtosis, spectral
spectralSkewness, spectralSlope, spectralSpread, pitch, harmonicRatio, zerocrossrate
shortTimeEnergy
```

To extract a feature, set the corresponding property to true.
For example, `obj.mfcc = true`, adds mfcc to the list of enabled features.

Configure the `audioFeatureExtractor` to extract all features.

```
in = info(afe,"all");
featureSwitches = fields(in);
cellfun(@(x)afe.set(x,true),featureSwitches)
```

```
afe
```

```
afe =
  audioFeatureExtractor with properties:
```

Properties

```
          Window: [240x1 double]
      OverlapLength: 160
          SampleRate: 8000
           FFTLength: []
SpectralDescriptorInput: 'linearSpectrum'
      FeatureVectorLength: 306
```

Enabled Features

```
linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
mfccDeltaDelta, gtcc, gtccDelta, gtccDeltaDelta, spectralCentroid, spectralCrest
spectralDecrease, spectralEntropy, spectralFlatness, spectralFlux, spectralKurtosis, spectral
spectralSkewness, spectralSlope, spectralSpread, pitch, harmonicRatio, zerocrossrate
shortTimeEnergy
```

Disabled Features

```
none
```

To extract a feature, set the corresponding property to true.
For example, `obj.mfcc = true`, adds mfcc to the list of enabled features.

You can use the `extract` object function of `audioFeatureExtractor` to extract all the enabled features from an audio signal. The features are concatenated into a matrix with analysis windows along the rows and features along the columns.

```
featureMatrix = extract(afe,x);
[numWindows,numFeatures] = size(featureMatrix)
```

```
numWindows = 62
```

```
numFeatures = 306
```

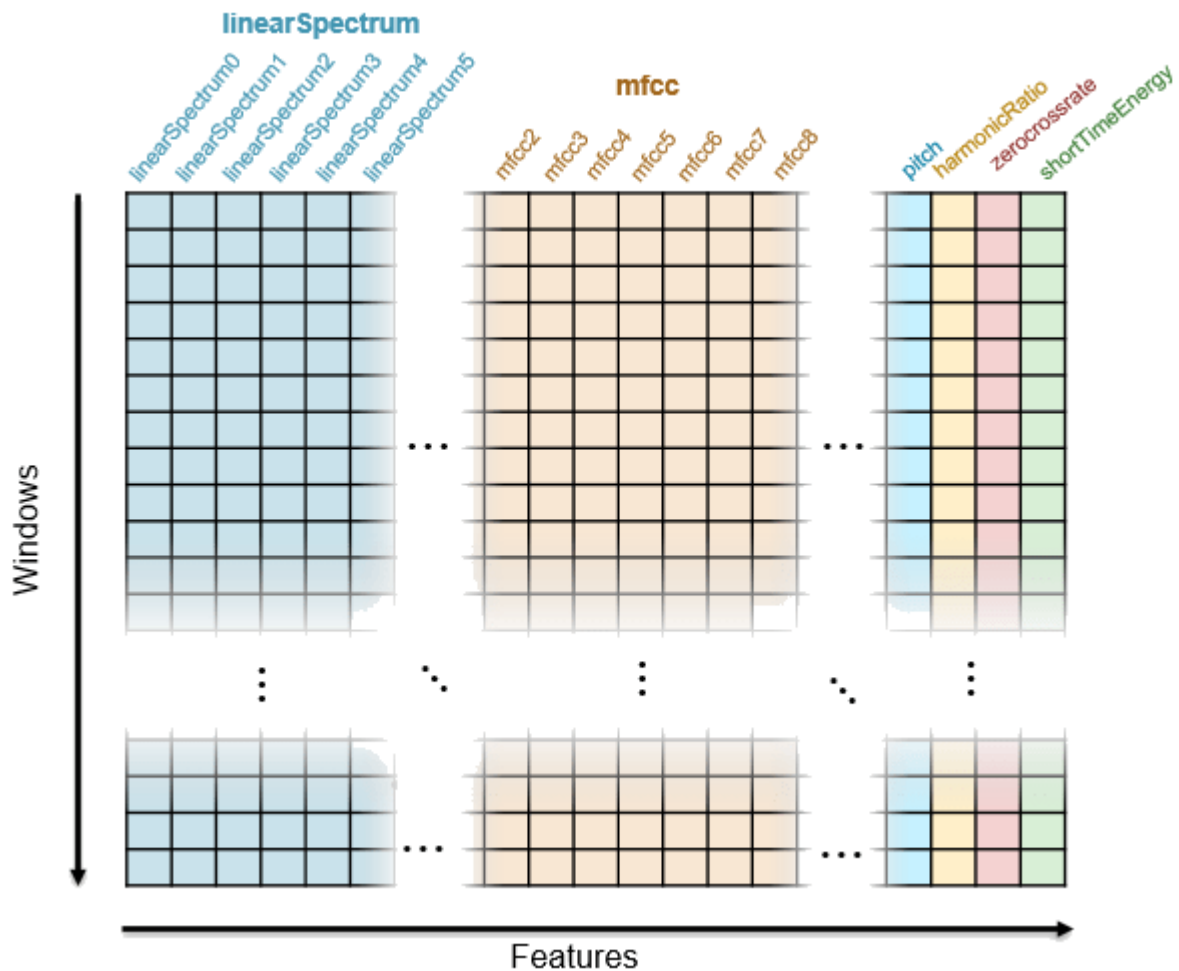
You can use `info` to get a mapping between the columns of the output matrix and the feature names. The term "features" is overloaded in the literature. *features* can refer to the feature group, such as "linearSpectrum", "mfcc", or "spectralCentroid", or the individual feature elements, such as the first element of the linear spectrum or the third element of the MFCC. The output map returned by `info` is a struct where each field corresponds to a feature group and the values correspond to which columns in the feature matrix the feature groups occupy.

```
outputMap = info(afe)
```

```
outputMap = struct with fields:
```

```
    linearSpectrum: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226]
      melSpectrum: [122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226]
      barkSpectrum: [154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226]
      erbSpectrum: [186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226]
      mfcc: [214 215 216 217 218 219 220 221 222 223 224 225 226]
      mfccDelta: [227 228 229 230 231 232 233 234 235 236 237 238 239]
      mfccDeltaDelta: [240 241 242 243 244 245 246 247 248 249 250 251 252]
      gtcc: [253 254 255 256 257 258 259 260 261 262 263 264 265]
      gtccDelta: [266 267 268 269 270 271 272 273 274 275 276 277 278]
      gtccDeltaDelta: [279 280 281 282 283 284 285 286 287 288 289 290 291]
      spectralCentroid: 292
      spectralCrest: 293
      spectralDecrease: 294
      spectralEntropy: 295
      spectralFlatness: 296
      spectralFlux: 297
      spectralKurtosis: 298
      spectralRolloffPoint: 299
      spectralSkewness: 300
      spectralSlope: 301
      spectralSpread: 302
      pitch: 303
      harmonicRatio: 304
      zerocrossrate: 305
      shortTimeEnergy: 306
```

This figure is intended to help you interpret the feature matrix returned from `extract`.



Use `transform` to create a transformed datastore that extracts features when reading from the audio datastore. The transform function is applied after calling `read` on the underlying datastore. Use `readall` to extract audio features from all audio files and place them into memory. If you have Parallel Computing Toolbox™, spread the computation across multiple workers.

The output is a *(Number of files)*-by-1 cell array. Each element of the cell array is a *(Number of hops)*-by-*(Number of features)* matrix, where the number of hops depends on the length of the audio file.

```
pFlag = ~isempty(ver("parallel"));
```

```
adsTrainT = transform(adsTrain,@(x){extract(afe,x)});
features = readall(adsTrainT,UseParallel=pFlag)
```

```
features=1600x1 cell array
    {62x306 double}
    {51x306 double}
    {66x306 double}
    {59x306 double}
    {49x306 double}
    {56x306 double}
    {60x306 double}
```

```
{61×306 double}  
{53×306 double}  
{50×306 double}  
{63×306 double}  
{61×306 double}  
{51×306 double}  
{60×306 double}  
{57×306 double}  
{58×306 double}  
:  
:
```

Feature/Label Correspondence

Once you have extracted features from approximately stationary windows in time, the next question is whether to feed the window-level features to your machine learning model or to combine the features into file-level representations. The choice of window-level or file-level features depends on your application and requirements. For file-level features, you will generally create summary statistics of the window-level features to collapse the time dimension. Common summary statistics include the mean and standard deviation. This example uses window-level features.

To train a machine learning model on window-level features, replicate the file-level labels so that they are in one-to-one correspondence with the features.

```
N = cellfun(@(x) size(x,1), features);  
T = repelem(adsTrain.Labels,N);
```

Concatenate the features into a single matrix for consumption by machine-learning tools.

```
X = cat(1, features{:});
```

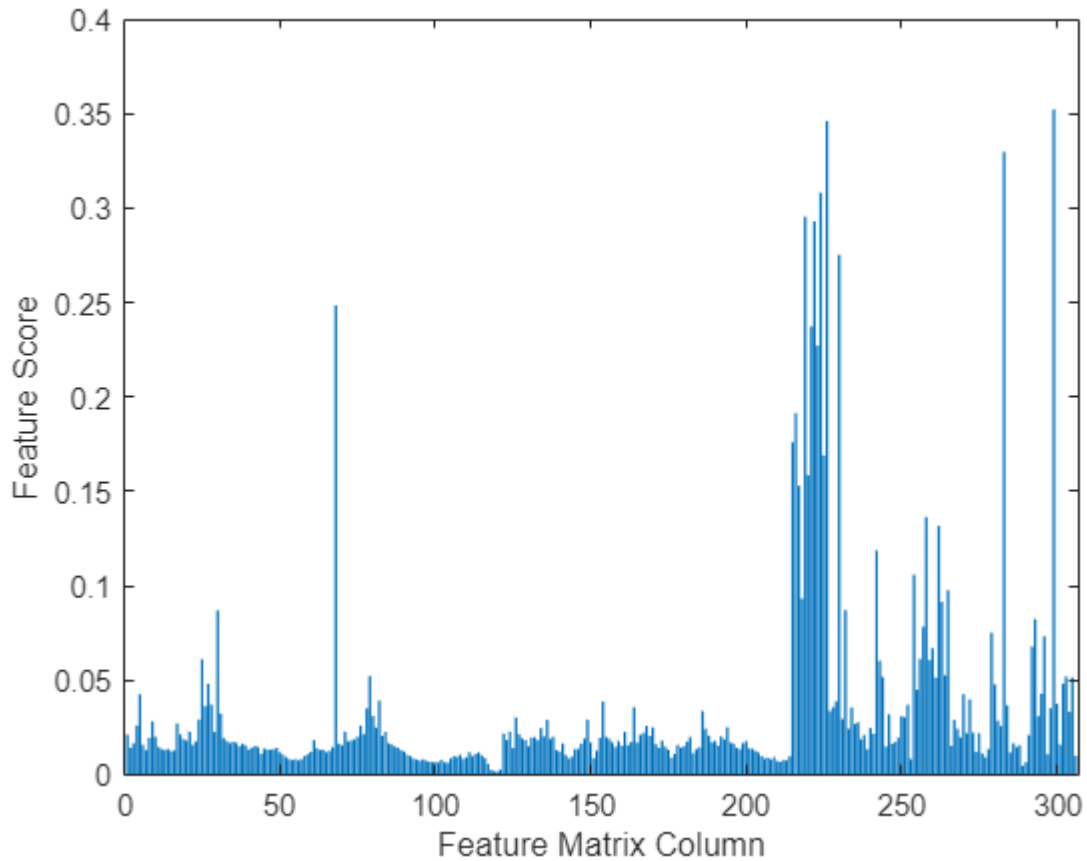
Feature Selection

Statistics and Machine Learning Toolbox™ provides several tools to aid in feature selection. The best feature selector will depend on your intended model. Use `fscmr` (Statistics and Machine Learning Toolbox) to rank features for classification using the minimum-redundancy/maximum-relevance (MRMR) algorithm. The MRMR is a sequential algorithm that finds an optimal set of features that is mutually and maximally dissimilar and can represent the response variable effectively.

```
rng("default") % for reproducibility  
[featureSelectionIdx, featureSelectionScores] = fscmr(X,T);
```

The `fscmr` function considers each column of the input feature matrix as a unique feature. Plot the scores of each scalar in the feature matrix returned by `audioFeatureExtractor`.

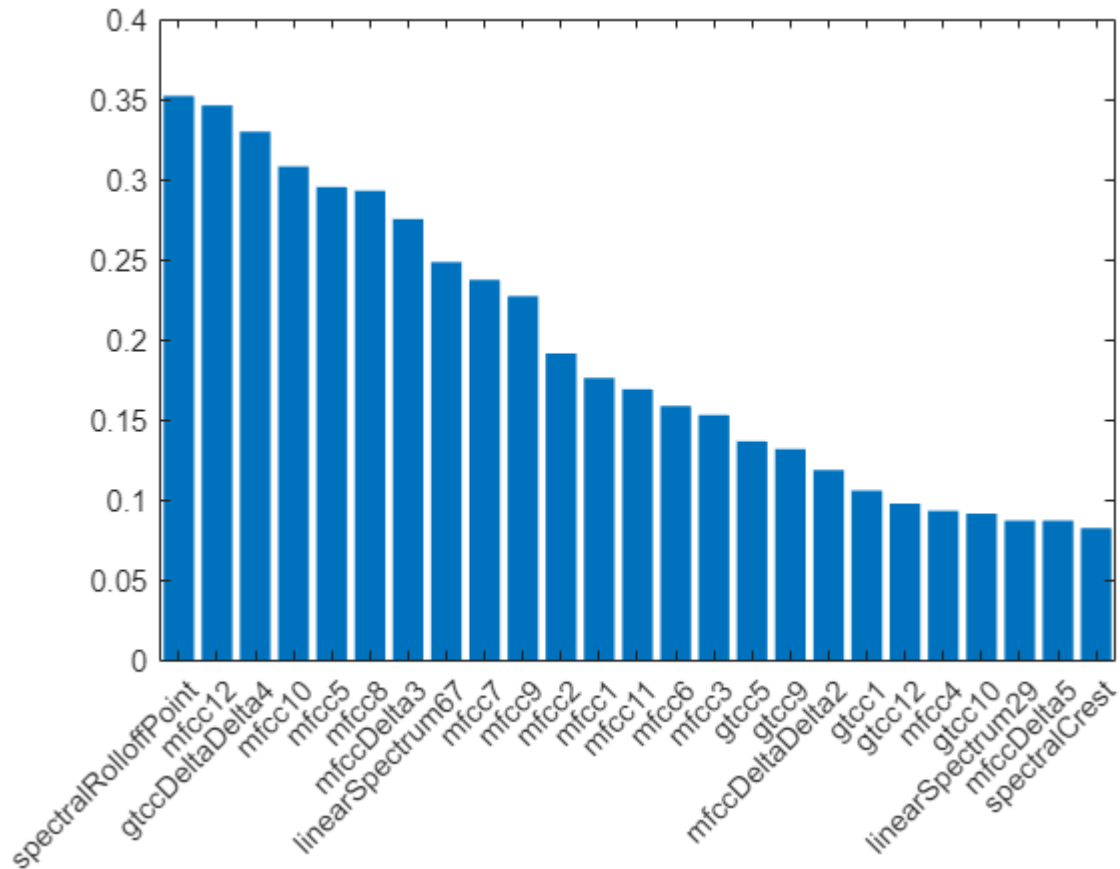
```
figure  
bar(featureSelectionScores)  
ylabel("Feature Score")  
xlabel("Feature Matrix Column")
```



The `audioFeatureExtractor` extracts feature groups with varying numbers of elements. For example, the default number of elements of the MFCC feature group is 13, while the spectral centroid feature always consists of 1 element. The output map returned by calling `info` on `audioFeatureExtractor` is a struct with fields equal to the feature group and values equal to the columns that feature group occupies in the matrix output by `extract`. Use the output map and the supporting function `uniqueFeatureName` on page 1-871 to create a unique name for each scalar feature, then plot the scores of the top 25 performing features.

```
featureNames = uniqueFeatureName(outputMap);

featureNamesSorted = featureNames(featureSelectionIdx);
figure
bar(reordercats(categorical(featureNames), featureNamesSorted), featureSelectionScores)
xlim([featureNamesSorted(1), featureNamesSorted(25)])
```



Depending on your application, you can approximate grouped feature selection by averaging the scores of feature groups. Using grouped features (for example, all MFCC) may help you deploy more efficient feature extraction. In this example, you use the top-performing feature scalars, regardless of which feature group they belong to.

Select some top scoring features. The number you select will depend on the model you are training and the final constraints of your application.

```
numFeatures = 30 ;
selectedFeatureIndex = featureSelectionIdx(1:numFeatures);
```

Train Model

To train a KNN model using your selected features, use `fitcknn` (Statistics and Machine Learning Toolbox). If you are unsure of which machine learning model you want to use, try `fitcauto` (Statistics and Machine Learning Toolbox) to automatically select a classification model with optimized parameters, or try the Classification Learner (Statistics and Machine Learning Toolbox).

```
Mdl = fitcknn(X(:,selectedFeatureIndex),T,Standardize=true);
```

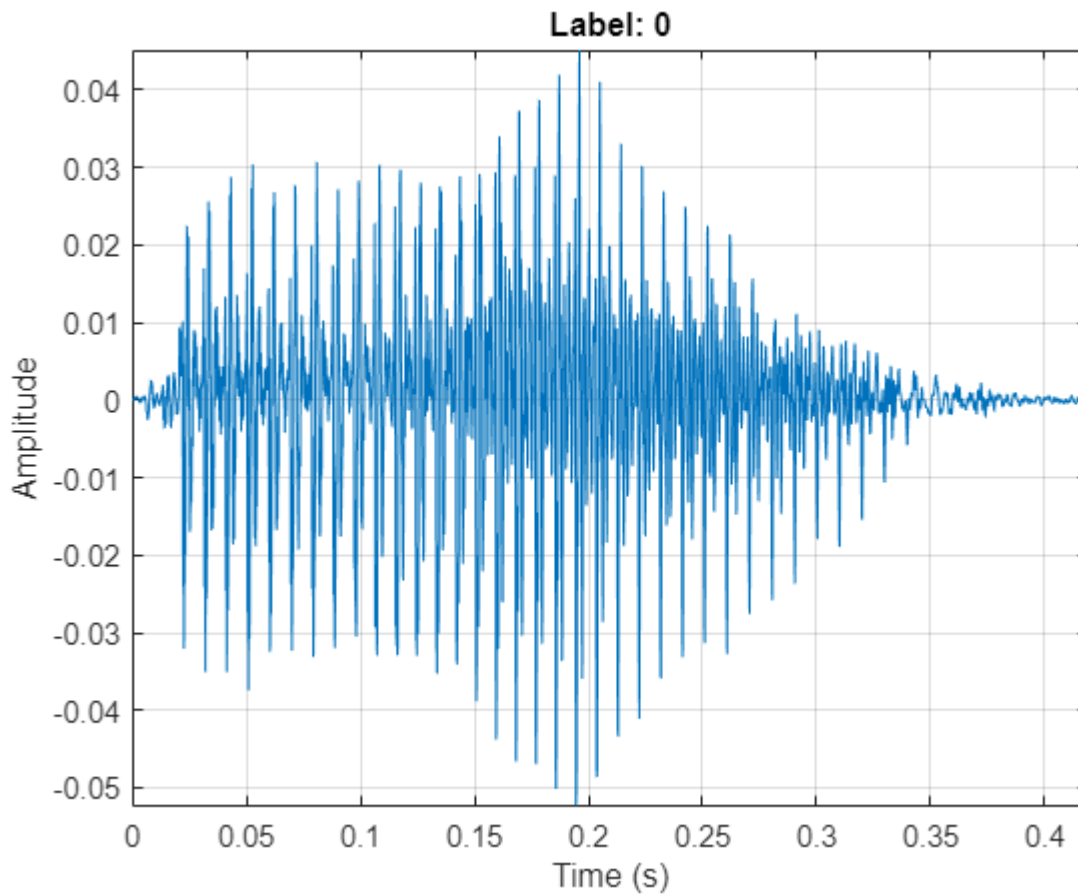
Evaluate Model

Spot-check the model's performance.

Read a sample from the test set. Listen to the sample and then plot its waveform and display the ground-truth label.

```
[x,xInfo] = read(adsTest);
sound(x,xInfo.SampleRate)

t = (0:numel(x)-1)/xInfo.SampleRate;
figure
plot(t,x)
title("Label: " + xInfo.Label)
grid on
axis tight
ylabel("Amplitude")
xlabel("Time (s)")
```



Extract features from analysis windows.

```
yPerWindow = extract(afe,x);
```

Predict the correct label per window.

```
t = predict(Mdl,yPerWindow(:,selectedFeatureIndex));
```

```
trueLabel = categorical(xInfo.Label)
```


You can apply a similar pattern as above to also select an optimal window, window length, window overlap, DFT length, and input to spectral descriptors.

Supporting Functions

```
function c = uniqueFeatureName(afeInfo)
%UNIQUEFEATURENAME Create unique feature names
%c = uniqueFeatureName(featureInfo) creates a unique set of feature names
%for each element of each feature described in the afeInfo struct. The
%afeInfo struct is returned by the info object function of
%audioFeatureExtractor.
a = repelem(fields(afeInfo),structfun(@numel,afeInfo));
b = matlab.lang.makeUniqueStrings(a);
d = find(endsWith(b,"_1"));
c = strrep(b,"_", "");
c(d-1) = strcat(c(d-1),"0");
end
```

References

[1] Jakobovski. "Jakobovski/Free-Spoken-Digit-Dataset." GitHub, May 30, 2019. <https://github.com/Jakobovski/free-spoken-digit-dataset>.

Transfer Learning with Pretrained Audio Networks in Deep Network Designer

This example shows how to interactively fine-tune a pretrained network to classify new audio signals using Deep Network Designer.

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.

This example retrains YAMNet, a pretrained convolutional neural network, to classify a new set of audio signals.

Load Data

Download and unzip the air compressor data set [1] on page 1-880. This data set consists of recordings from air compressors in a healthy state or one of seven faulty states.

```
zipFile = matlab.internal.examples.downloadSupportFile('audio','AirCompressorDataset/AirCompressorDataset.zip');
dataFolder = fileparts(zipFile);
unzip(zipFile,dataFolder);
```

Create an `audioDatastore` object to manage the data.

```
ads = audioDatastore(dataFolder,IncludeSubfolders=true,LabelSource="foldernames");
```

Split the data into training, validation, and test sets using the `splitEachLabel` function.

```
[adsTrain,adsValidation,adsTest] = splitEachLabel(ads,0.7,0.2,0.1);
```

Use the `transform` function to preprocess the data using the function `audioPreprocess`, found at the end of this example. For each signal:

- Use `yamnetPreprocess` to generate mel spectrograms suitable for training using YAMNet. Each audio signal produces multiple spectrograms.
- Duplicate the class label for each of the spectrograms.

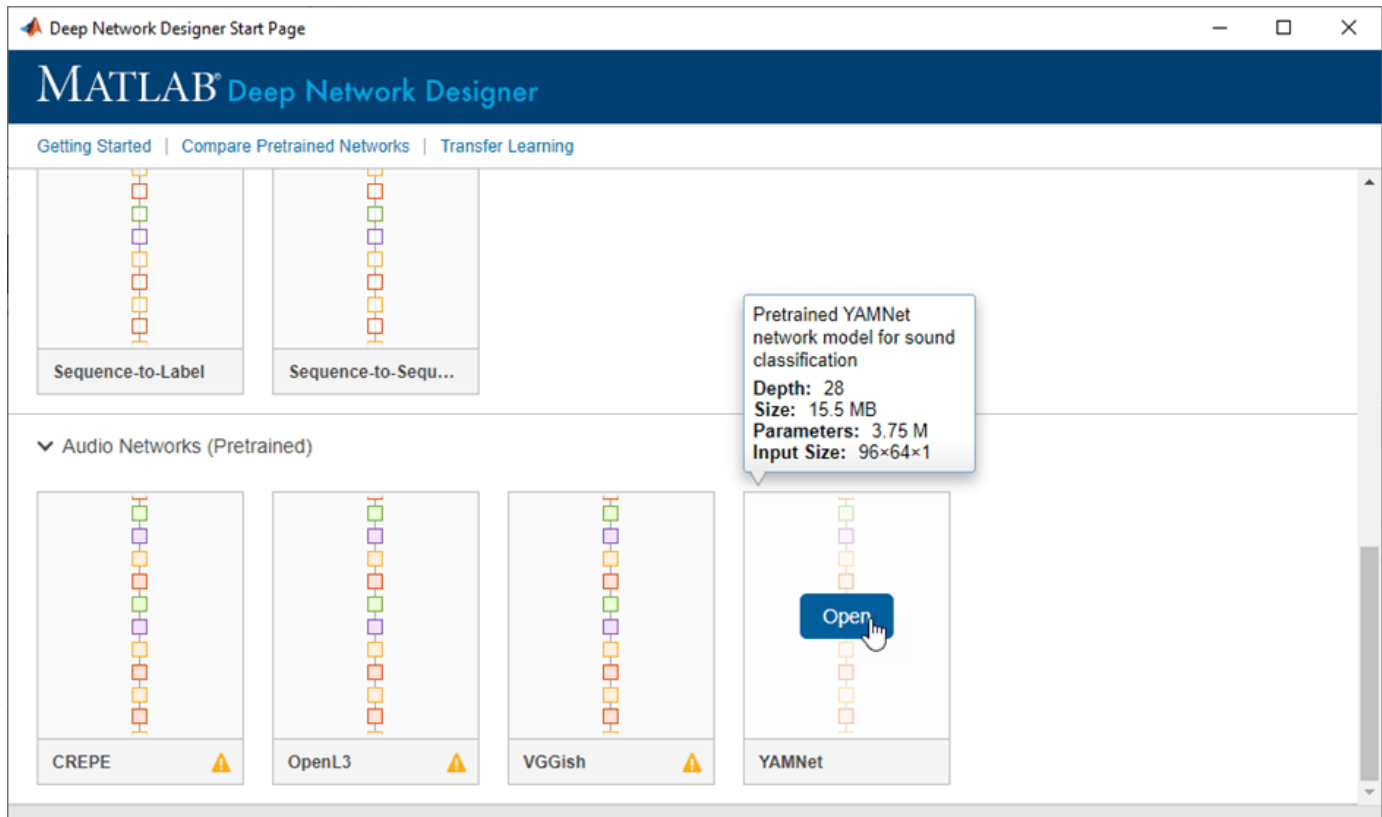
```
tdsTrain = transform(adsTrain,@audioPreprocess,IncludeInfo=true);
tdsValidation = transform(adsValidation,@audioPreprocess,IncludeInfo=true);
tdsTest = transform(adsTest,@audioPreprocess,IncludeInfo=true);
```

Select Pretrained Network

Prepare and train the network interactively using Deep Network Designer (Deep Learning Toolbox). To open Deep Network Designer, on the **Apps** tab, under **Machine Learning and Deep Learning**, click the app icon. Alternatively, you can open the app from the command line.

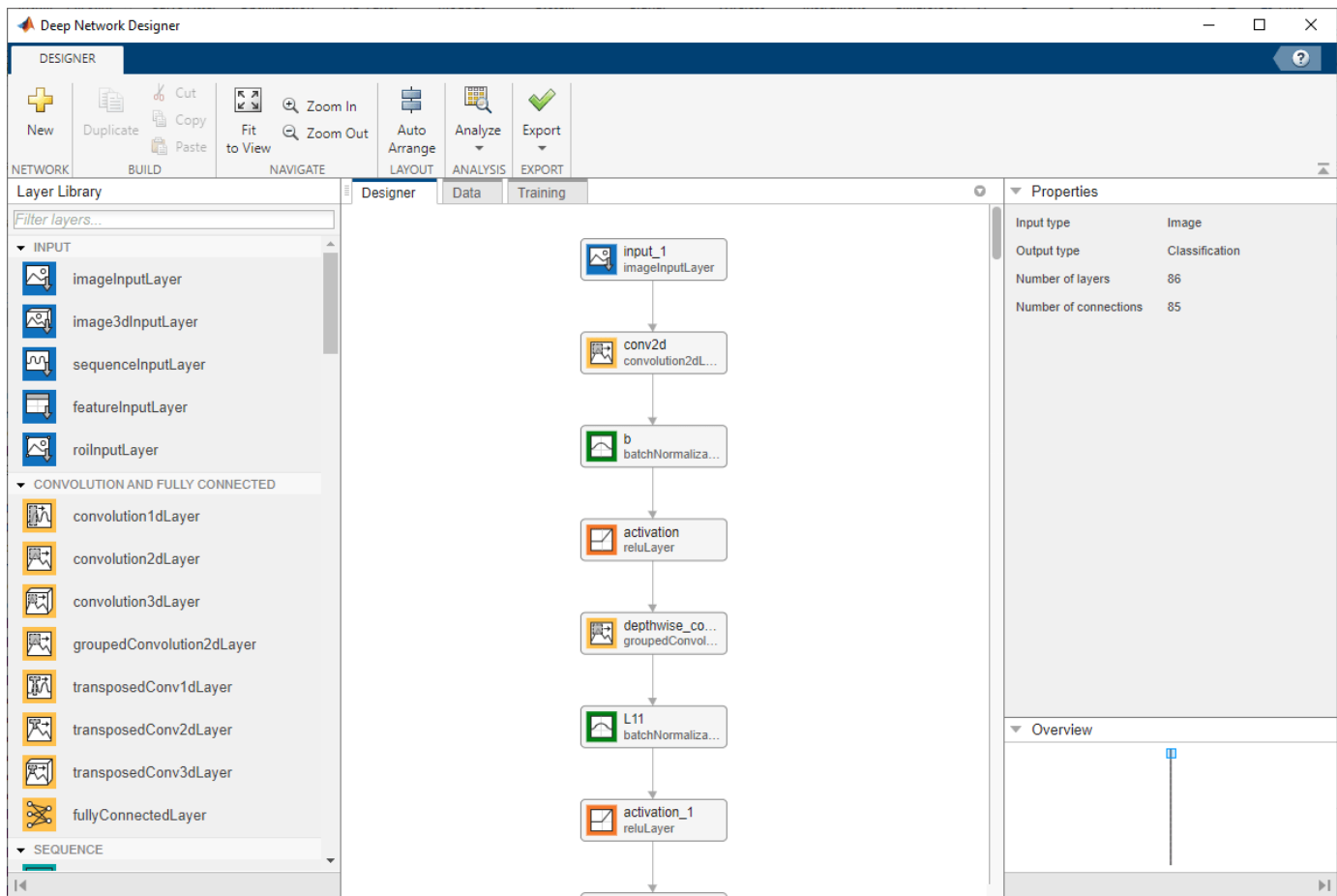
```
deepNetworkDesigner
```

Deep Network Designer provides a selection of pretrained audio classification networks. These models require both Audio Toolbox™ and Deep Learning Toolbox™.



Under **Audio Networks**, select **YAMNet** from the list of pretrained networks and click **Open**. If the Audio Toolbox model for YAMNet is not installed, click **Install** instead. Deep Network Designer provides a link to the location of the network weights. Unzip the file to a location on the MATLAB path. Now close the Deep Network Designer Start Page and reopen it. When the network is correctly installed and on the path, you can click the **Open** button on YAMNet. The YAMNet model can classify audio into one of 521 sound categories. For more information, see [yamnet](#).

Deep Network Designer displays a zoomed-out view of the whole network in the **Designer** pane. To zoom in with the mouse, use **Ctrl**+scroll wheel. To pan, use the arrow keys, or hold down the scroll wheel and drag the mouse. Select a layer to view its properties. Clear all layers to view the network summary in the **Properties** pane.



Prepare Network for Transfer Learning

To prepare the network for transfer learning, in the **Designer** pane, replace the last learnable layer and the final classification layer.

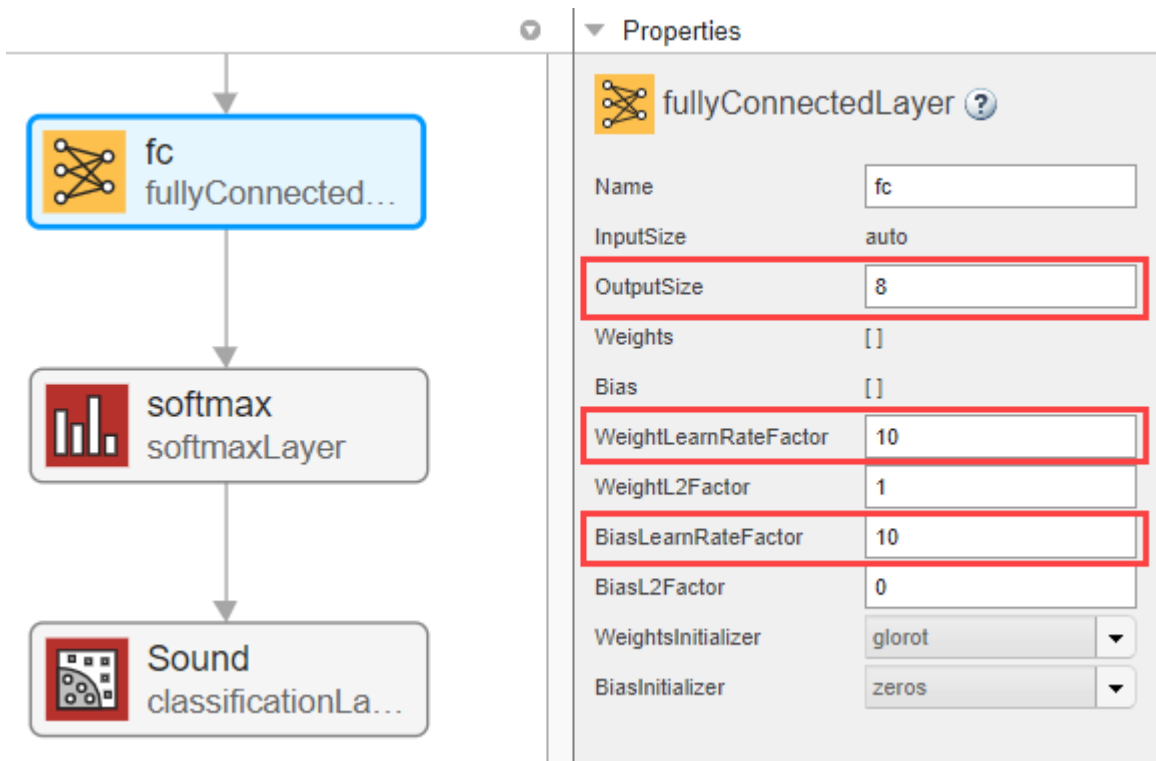
Replace Last Learnable Layer

To use a pretrained network for transfer learning, you must change the number of classes to match your new data set. First, find the last learnable layer in the network. For YAMNet, the last learnable layer is the last fully connected layer, **dense**.

Drag a new `fullyConnectedLayer` onto the canvas. The `OutputSize` property defines the number of classes for classification problems. Change `OutputSize` to the number of classes in the new data, in this example, 8.

Change the learning rates so that learning is faster in the new layer than in the transferred layers by setting `WeightLearnRateFactor` and `BiasLearnRateFactor` to 10.

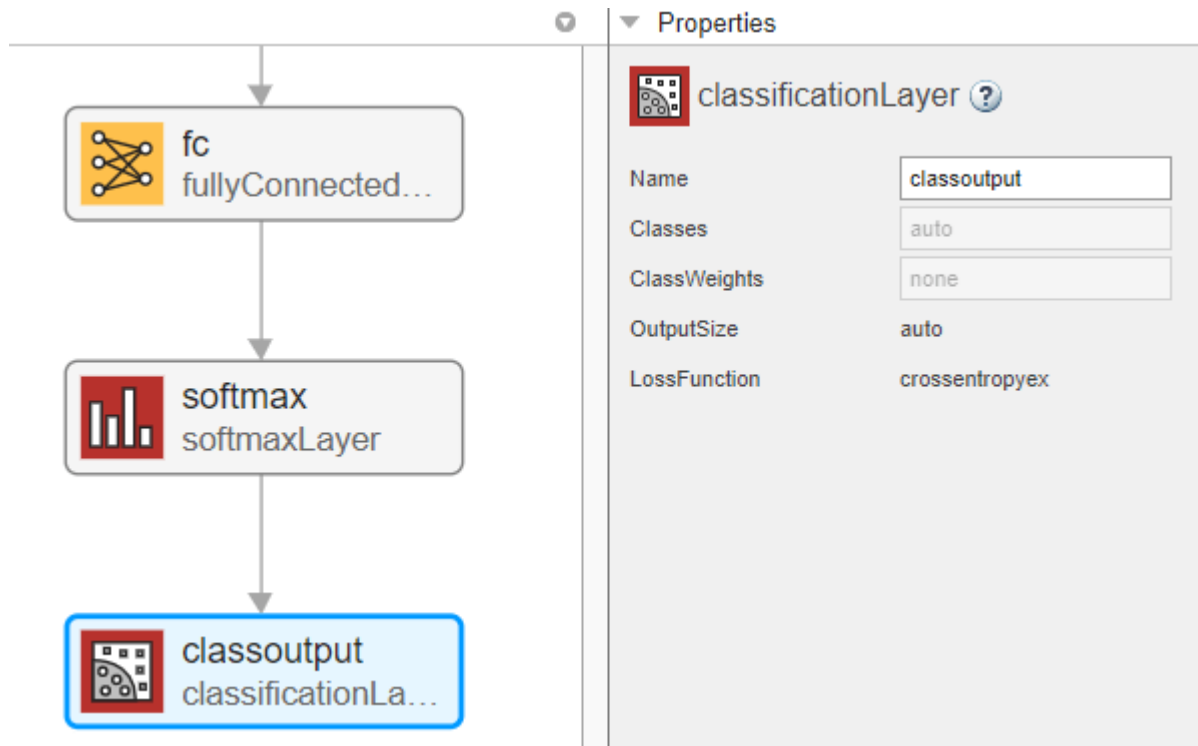
Delete the last fully connected layer and connect your new layer instead.



Replace Output Layer

For transfer learning, you need to replace the output layer. Scroll to the end of the **Layer Library** and drag a new `classificationLayer` onto the canvas. Delete the original classification layer and connect your new layer in its place.

For a new output layer, you do not need to set `OutputSize`. At training time, Deep Network Designer automatically sets the output classes of the layer from the data.



Check Network

To check that the network is ready for training, click **Analyze**. If the Deep Learning Network Analyzer reports zero errors, then the edited network is ready for training.

Deep Learning Network Analyzer

Analysis for training in Deep Network Designer

Name: Network from Deep Network Designer

Analysis date: 10-Jan-2022 09:37:19

3.2M total learnables

86 layers

0 warnings

0 errors

input_1

conv2d

b

activation

depthwise_con...

L11

activation_1

conv2d_1

L12

activation_2

depthwise_con...

L21

activation_3

conv2d_2

ANALYSIS RESULT

	Name	Type	Activations	Learnable Properties
1	input_1 60x64x1 images	Image Input	96(S) × 64(S) × 1(C) × 1(B)	-
2	conv2d 32 3x3x1 convolutions with stride [2 2] a...	Convolution	48(S) × 32(S) × 32(C) × 1(B)	Weights 3 × 3 × 1 × 32 Bias 1 × 1 × 32
3	b Batch normalization with 32 channels	Batch Normalization	48(S) × 32(S) × 32(C) × 1(B)	Offset 1 × 1 × 32 Scale 1 × 1 × 32
4	activation ReLU	ReLU	48(S) × 32(S) × 32(C) × 1(B)	-
5	depthwise_conv2d 32 groups of 1 3x3x1 convolutions with ...	Grouped Convolution	48(S) × 32(S) × 32(C) × 1(B)	Weigh... 3 × 3 × 1 × 1 ... Bias 1 × 1 × 1 × 32
6	L11 Batch normalization with 32 channels	Batch Normalization	48(S) × 32(S) × 32(C) × 1(B)	Offset 1 × 1 × 32 Scale 1 × 1 × 32
7	activation_1 ReLU	ReLU	48(S) × 32(S) × 32(C) × 1(B)	-
8	conv2d_1 64 1x1x32 convolutions with stride [1 1] ...	Convolution	48(S) × 32(S) × 64(C) × 1(B)	Weigh... 1 × 1 × 32 × ... Bias 1 × 1 × 64
9	L12 Batch normalization with 64 channels	Batch Normalization	48(S) × 32(S) × 64(C) × 1(B)	Offset 1 × 1 × 64 Scale 1 × 1 × 64
10	activation_2 ReLU	ReLU	48(S) × 32(S) × 64(C) × 1(B)	-
11	depthwise_conv2d_1 64 groups of 1 3x3x1 convolutions with ...	Grouped Convolution	24(S) × 16(S) × 64(C) × 1(B)	Weigh... 3 × 3 × 1 × 1 ... Bias 1 × 1 × 1 × 64
12	L21	Batch Normalization	24(S) × 16(S) × 64(C) × 1(B)	Offset 1 × 1 × 64

Import Data

To load the data into Deep Network Designer, on the **Data** tab, click **Import Data > Import Custom Data**. Select `tdsTrain` as the training data and `tdsValidation` as the validation data.

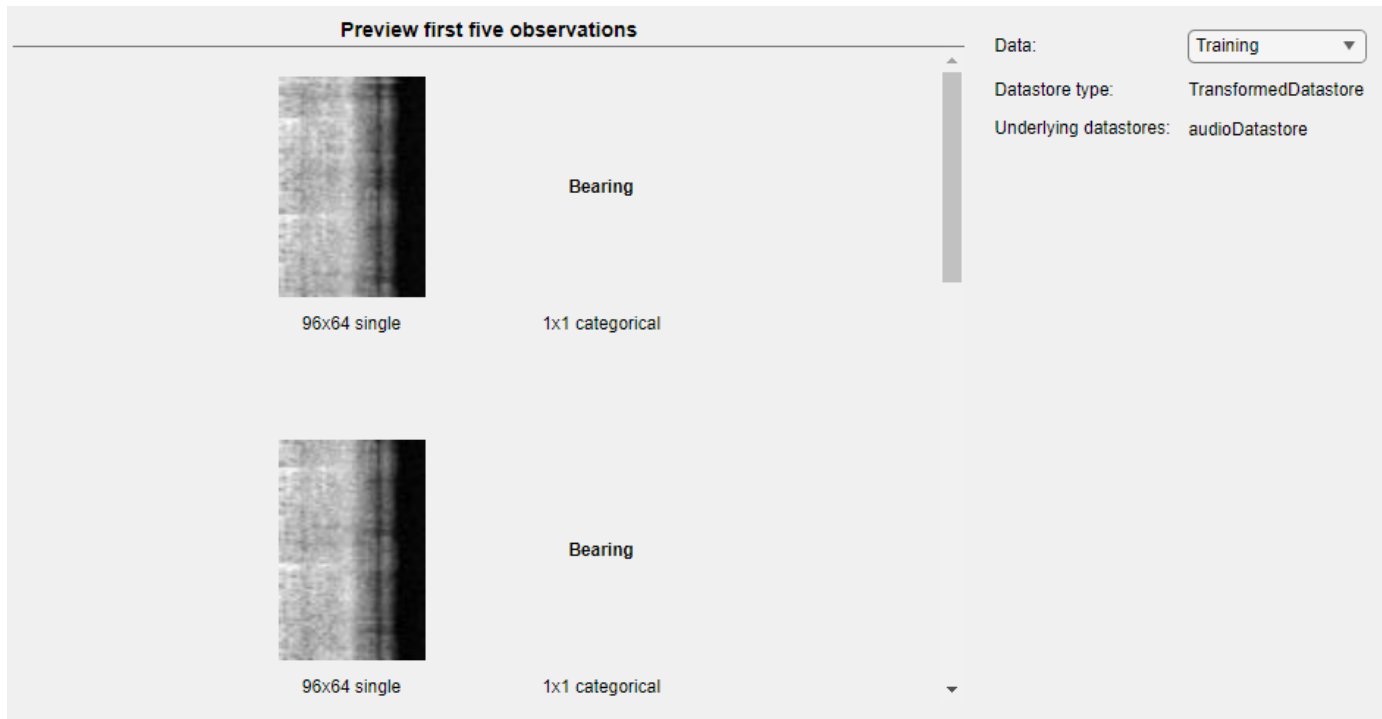
Import Custom Data

Training data: Refresh

Validation data: Refresh

Help Import Cancel

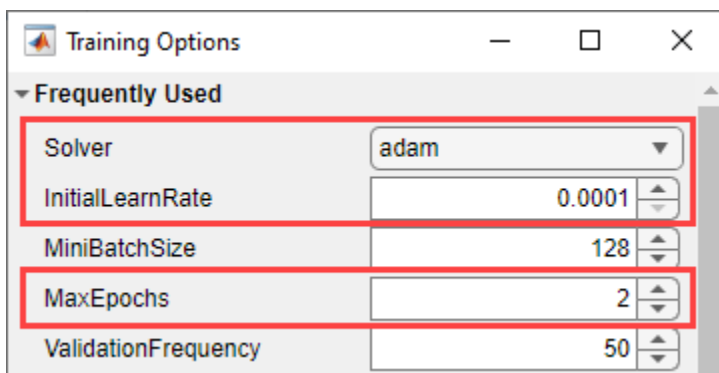
Using Deep Network Designer, you can inspect the training and validation data in the **Data** tab. You can see that the data is as expected prior to training.



Select Training Options

To specify the training options, select the **Training** tab and click **Training Options**. Set the initial learning rate to a small value to slow down learning in the transferred layers. In combination with the increased learning rate factors for the fully connected layer, learning is now fast only in the new layers and slower in the other layers.

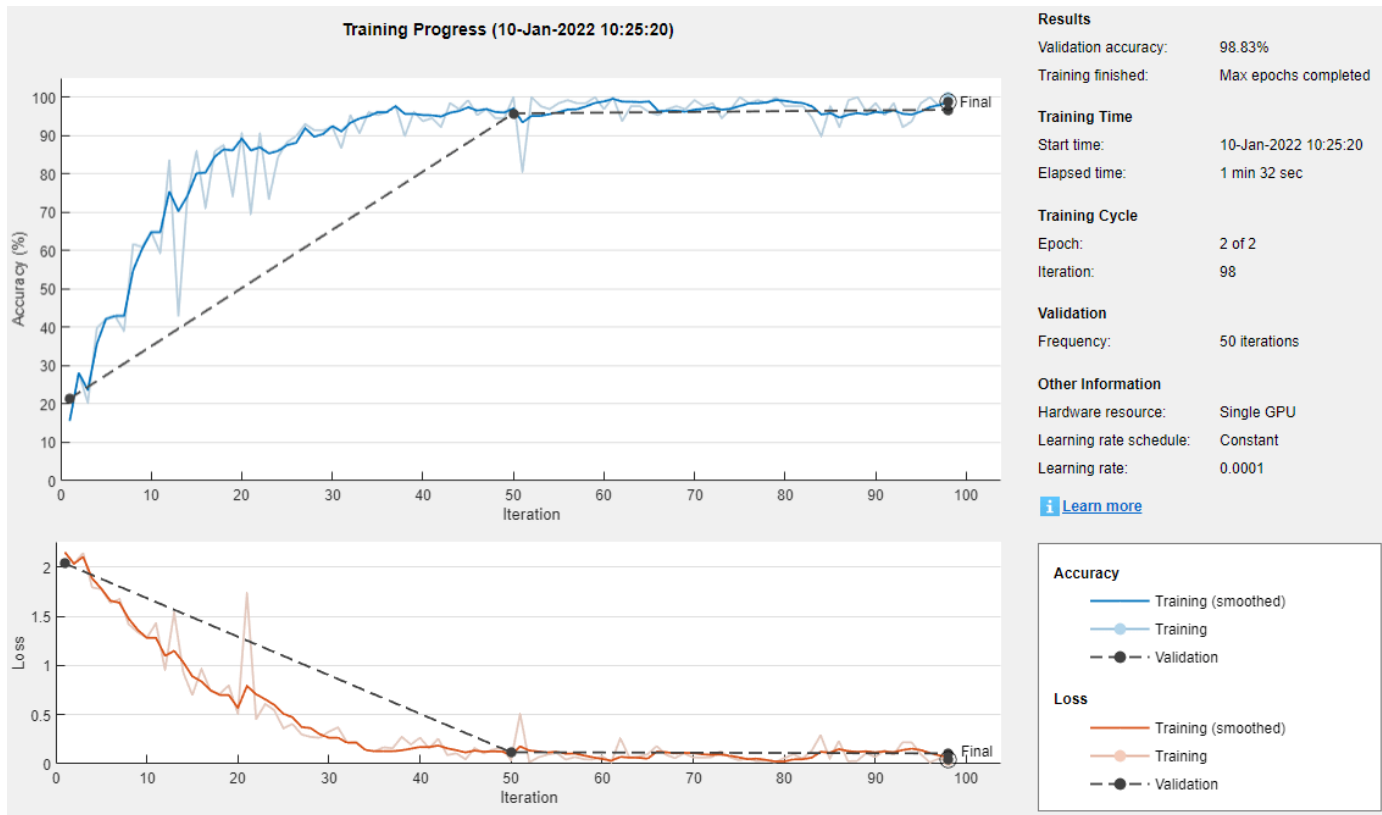
For this example, set **Solver** to adam, **InitialLearnRate** to 0.0001, and **MaxEpochs** to 2.



Train Network

To train the network with the specified training options, click **OK** and then click **Train**.

Deep Network Designer allows you to visualize and monitor the training progress. You can then edit the training options and retrain the network, if required. To find the optimal training options, create a deep learning experiment using Experiment Manager. You can create a deep learning experiment in Deep Network Designer by clicking **Export > Create Experiment**.



To export the results from training, on the **Training** tab, select **Export > Export Trained Network and Results**. Deep Network Designer exports the trained network as the variable `trainedNetwork_1` and the training info as the variable `trainInfoStruct_1`.

You can also generate MATLAB code, which recreates the network and the training options used. On the **Training** tab, select **Export > Generate Code for Training**. Examine the MATLAB code to learn how to programmatically prepare the data for training, create the network architecture, and train the network.

Test Network

Classify the test data using the exported network and the `classify` function.

```
data = readall(tdsTest);
YTest = [data{:,2}];
YPred = classify(trainedNetwork_1,tdsTest);

accuracy = sum(YPred == YTest')/numel(YTest)

accuracy = 0.9830
```

Supporting Function

The function `audioPreprocess` uses `yamnetPreprocess` to generate mel spectrograms from `audioIn` that you can feed to the YAMNet pretrained network. Each input signal generates multiple spectrograms, so the labels must be duplicated to create a one-to-one correspondence with the spectrograms.

```
function [data,info] = audioPreprocess(audioIn,info)
class = info.Label;
fs = info.SampleRate;
features = yamnetPreprocess(audioIn,fs);

numSpectrograms = size(features,4);

data = cell(numSpectrograms,2);
for index = 1:numSpectrograms
    data{index,1} = features(:,:,,index);
    data{index,2} = class;
end
end
```

References

[1] Verma, Nishchal K., Rahul Kumar Sevakula, Sonal Dixit, and Al Salour. "Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors." IEEE Transactions on Reliability 65, no. 1 (March 2016): 291-309. <https://doi.org/10.1109/TR.2015.2459684>.

Speech Command Recognition Code Generation with Intel MKL-DNN Using Simulink

This example demonstrates 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 Embedded Coder in Simulink® and the Intel® Math Kernel Library for Deep Neural Networks (MKL-DNN). In this example you generate **Software-in-the-loop (SIL)** code for a reference model which performs feature extraction and predicts the speech command. The generated SIL code is called in a Simulink model which displays the predicted speech command and predicted scores for the given inputs. For details about audio preprocessing and network training, see “Train Speech Command Recognition Model Using Deep Learning” on page 1-332.

Prerequisites

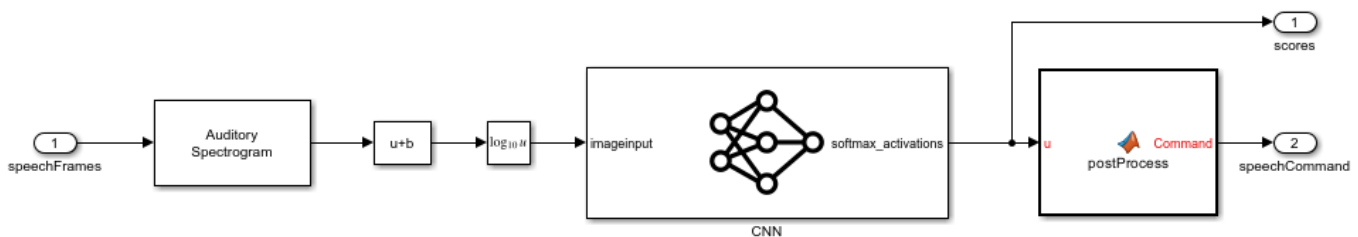
- The MATLAB® Coder Interface for Deep Learning Libraries
- Intel Processor with support for Advanced Vector Extension 2 (AVX2)
- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- Environment Variables for Intel MKL-DNN

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

Prepare Simulink Model to Deploy

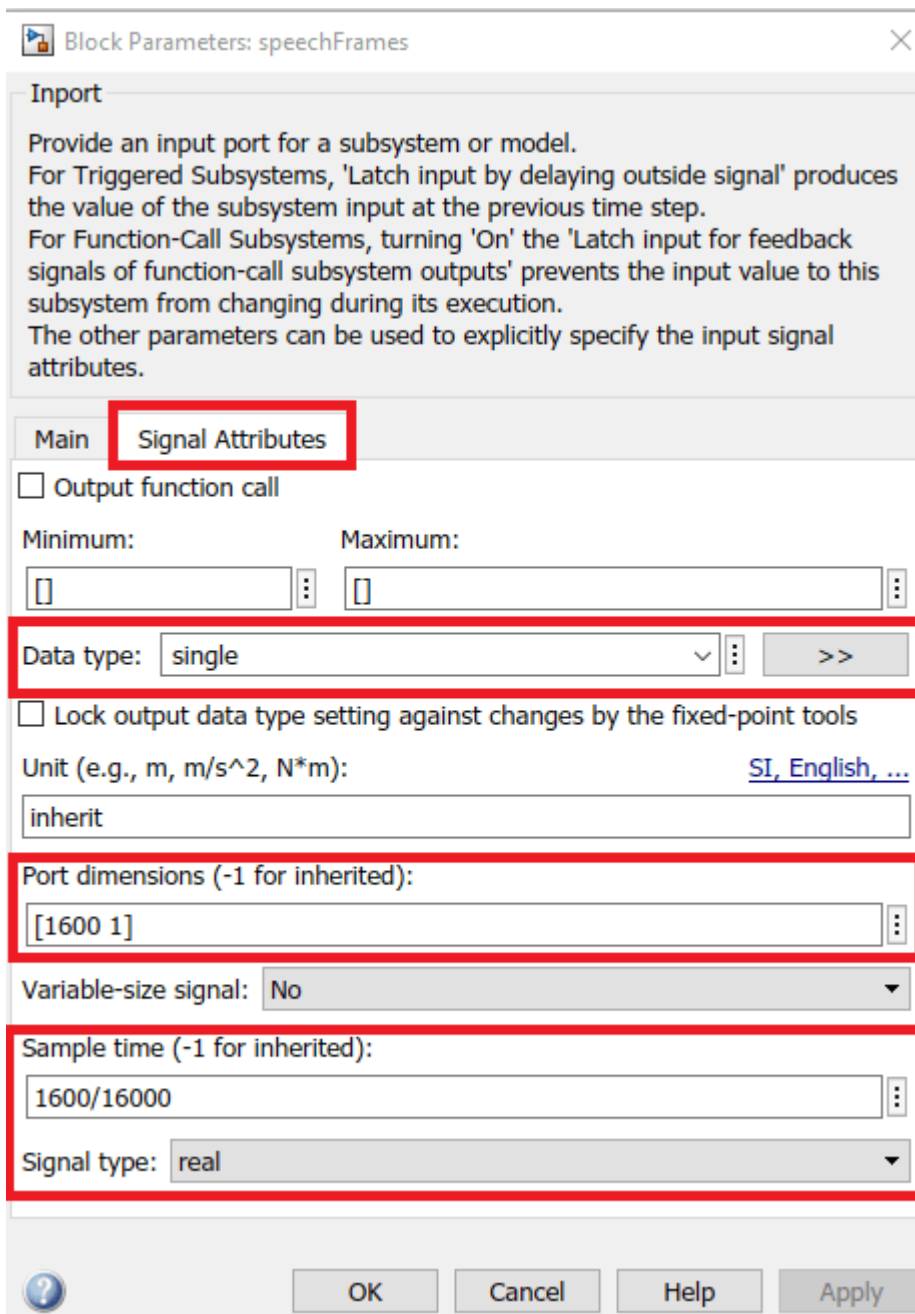
Create a Simulink model and capture the feature extraction, convolutional neural network and postprocessing as developed in “Speech Command Recognition in Simulink” on page 1-40. This model is shipped with this example. Open the shipped model to understand its configurations.

```
modelToDeploy = "recognizeSpeechCommand";
open_system(modelToDeploy)
```



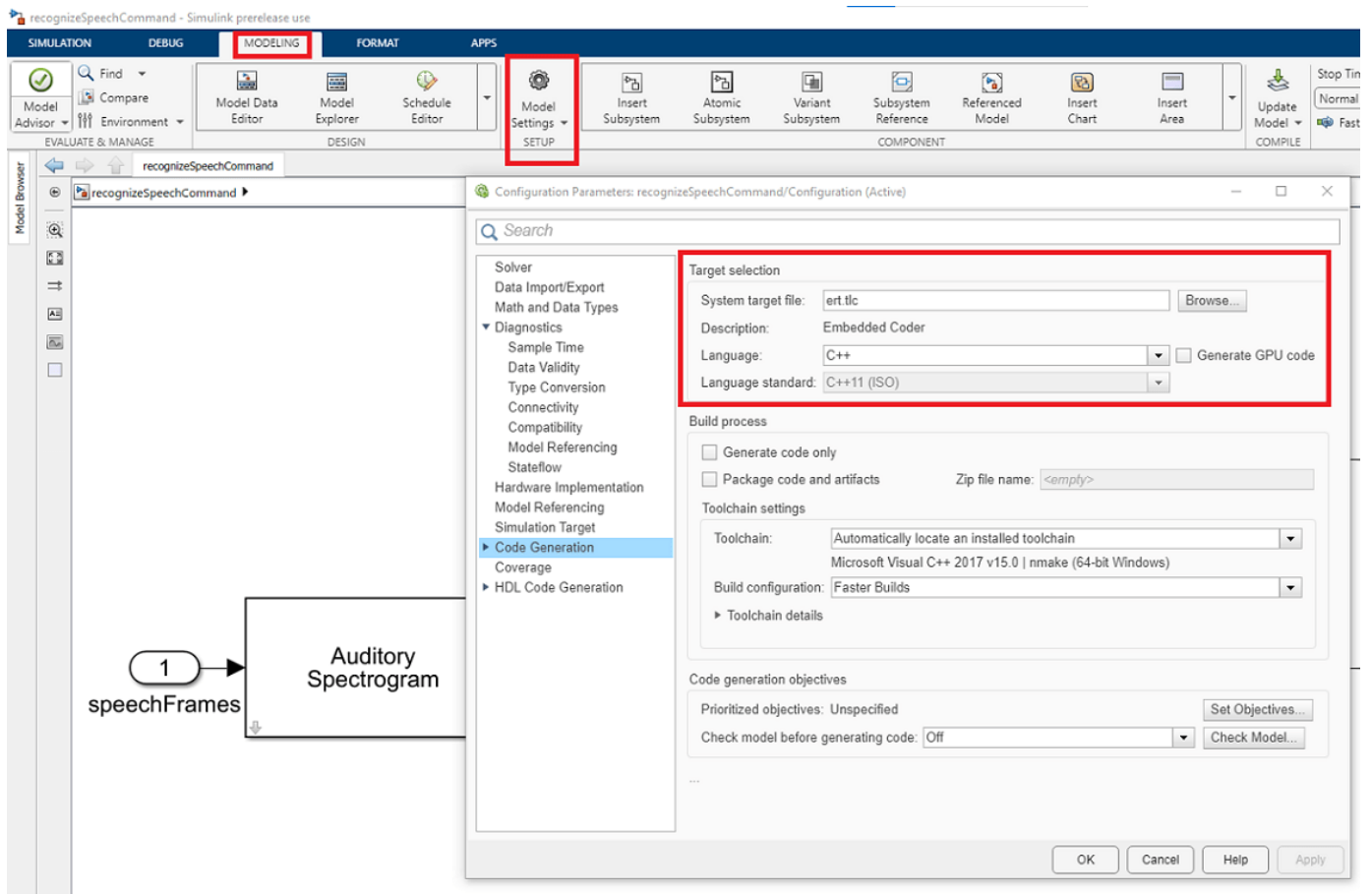
Copyright 2021, The MathWorks, Inc.

Set the **Data type**, **Port dimensions**, **Sample time**, and **Signal type** of the input port block as shown.



Configure Code Generation Settings

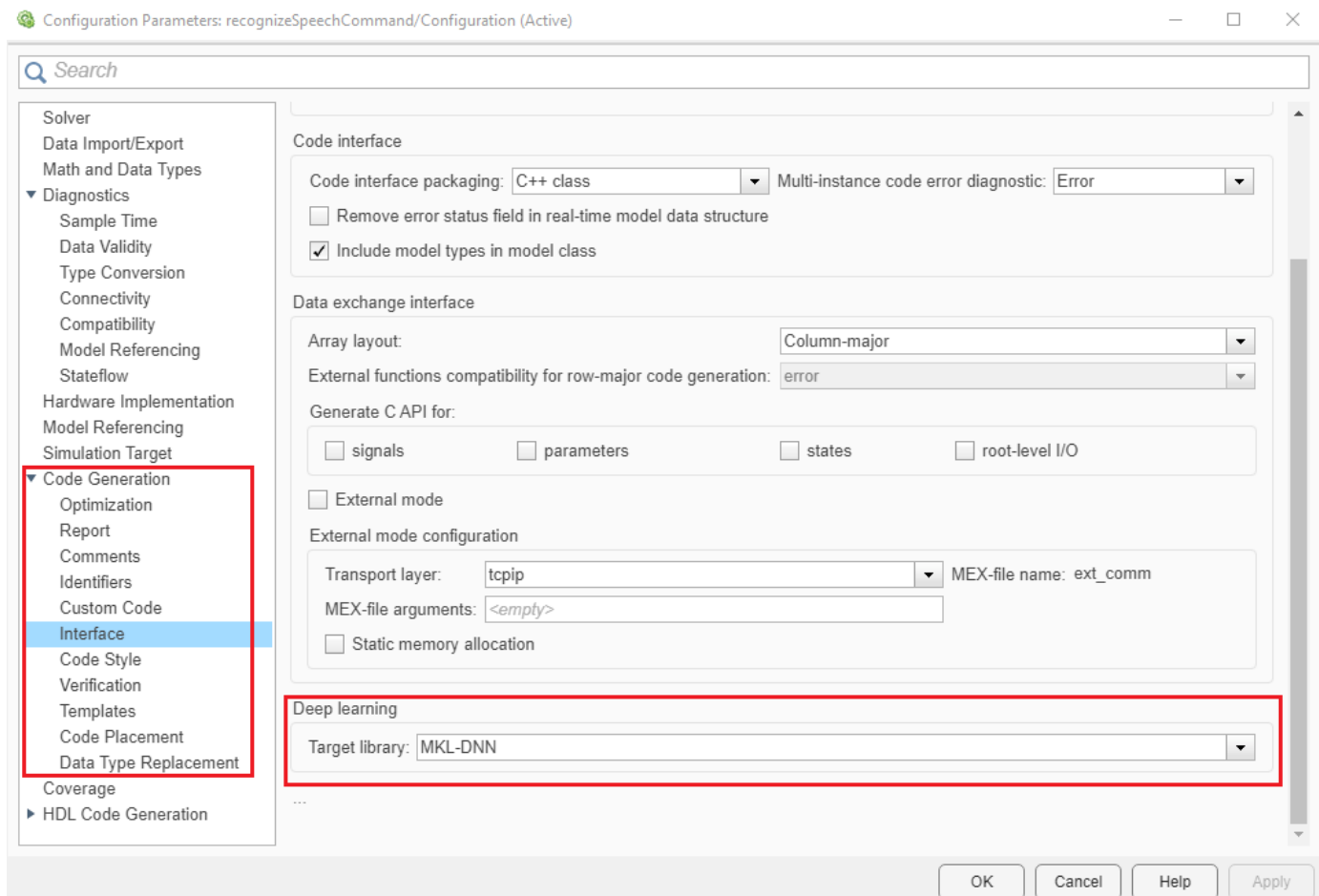
Open the `recognizeSpeechCommand` model. Go to the **MODELING** Tab and click on Model Settings or press **Ctrl+E**. Select **Code Generation** and set the **System Target File** to `ert.tlc` whose **Description** is Embedded Coder. Set the **Language** to C++, which will automatically set the **Language Standard** to C++11 (ISO).



Alternatively, use `set_param` to configure the settings programmatically,

```
set_param(modelToDeploy, SystemTargetFile="ert.tlc")
set_param(modelToDeploy, TargetLang="C++")
set_param(modelToDeploy, TargetLangStandard="C++11 (ISO)")
```

To set Intel MKL-DNN Deep Learning Config, expand **Code Generation** and select **Interface**. Now set the **Deep Learning Target Library** to MKL - DNN as shown.



Alternatively, use `set_param` to configure the Deep learning target library programmatically.

```
set_param(modelToDeploy,DLTargetLibrary="mkl-dnn")
```

Select a solver that supports code generation. Set **Solver** to `auto` (Automatic solver selection) and **Solver type** to `Fixed-step`.

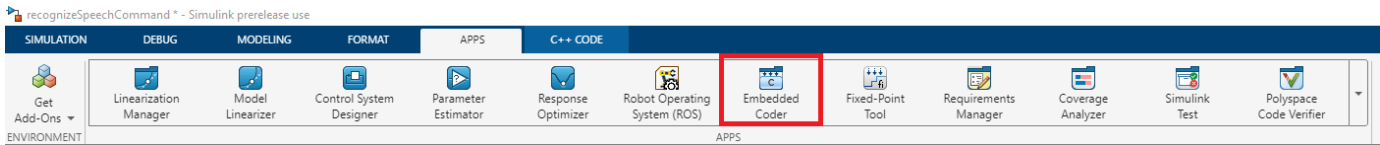
```
set_param(modelToDeploy,SolverName="FixedStepAuto")
set_param(modelToDeploy,SolverType="Fixed-step")
```

In **Configuration > Hardware Implementation**, set **Device vendor** to `Intel` and **Device type** to `x86-64 (Windows64)` or `x86-64 (Linux 64)` or `x86-64 (Mac OS X)` depending on your target system. Alternatively, use `set_param` to configure the settings programmatically.

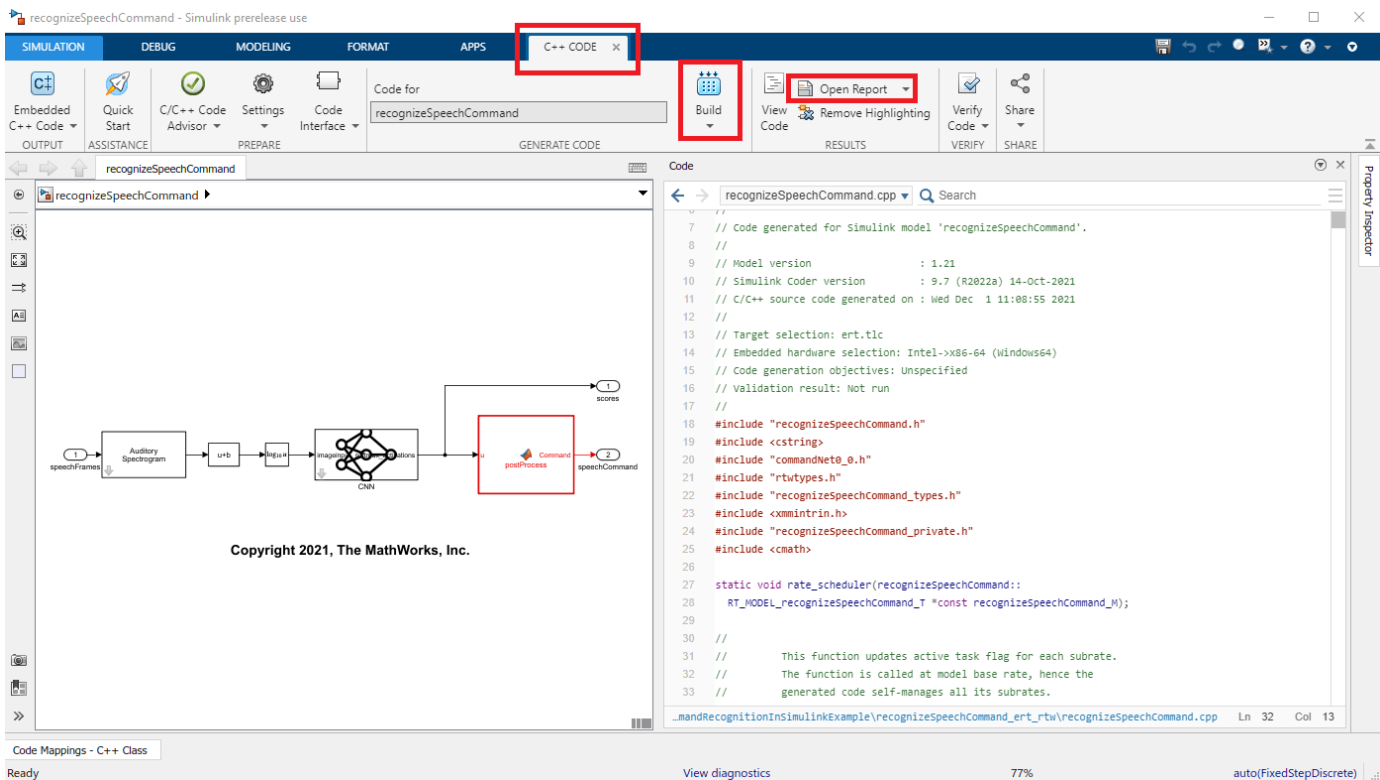
```
switch(computer("arch"))
    case "win64"
        ProdHWDeviceType = "Intel->x86-64 (Windows64)";
    case "lnxa64"
        ProdHWDeviceType = "Intel->x86-64 (Linux 64)";
    case "maci64"
        ProdHWDeviceType = "Intel->x86-64 (Mac OS X)";
end
set_param(modelToDeploy, "ProdHWDeviceType", ProdHWDeviceType)
```

To automate setting the **Device type**, add the above code in **Property Inspector > Properties > Callbacks > PreLoadFcn** of the recognizeSpeechCommand model.

Use Embedded Coder app to generate and build the code. Click on **APPS** tab and then click on **Embedded coder** as shown.



It will open a new **C++ CODE** tab, then click on **Build** to generate and build the code. It will generate the code in a folder named recognizeSpeechCommand_ert_rtw. After generating the code, you view the report by clicking on **Open Report**.



Alternatively, you can use `slbuild` to generate the code programatically.

```
slbuild(modelToDeploy);
```

```
### Starting build procedure for: recognizeSpeechCommand
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: W:\ExampleManager\sporwal.Bdoc23a.j2106495\deeplearning_sl
### Generated code for 'recognizeSpeechCommand' is up to date because no structural, parameter or
### Saving binary information cache.
### Skipping makefile generation and compilation because W:\ExampleManager\sporwal.Bdoc23a.j21064
### Successful completion of build procedure for: recognizeSpeechCommand
```

Build Summary

```
0 of 1 models built (1 models already up to date)
Build duration: 0h 0m 36.32s
```

Now close the recognizeSpeechCommand model.

```
save_system(modelToDeploy)
close_system(modelToDeploy)
```

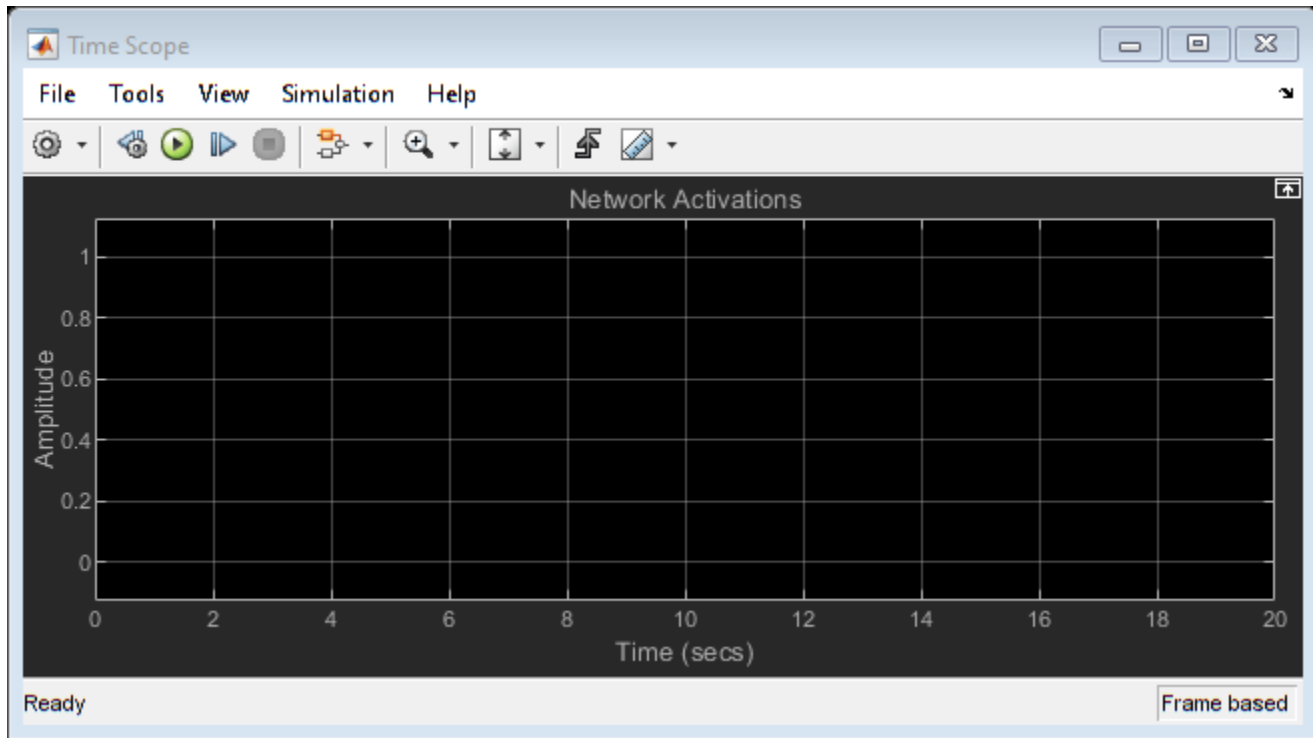
Create a Simulink Model that Calls recognizeSpeechCommand and Displays its Output

Create a new simulink model and add recognizeSpeechCommand as a model reference block to it. Add the same base workspace variables, source blocks, and sink blocks as developed in “Speech Command Recognition in Simulink” on page 1-40. Use a radio button group for selecting speech command files. For your reference, this model is shipped with this example. Open the same simulink model.

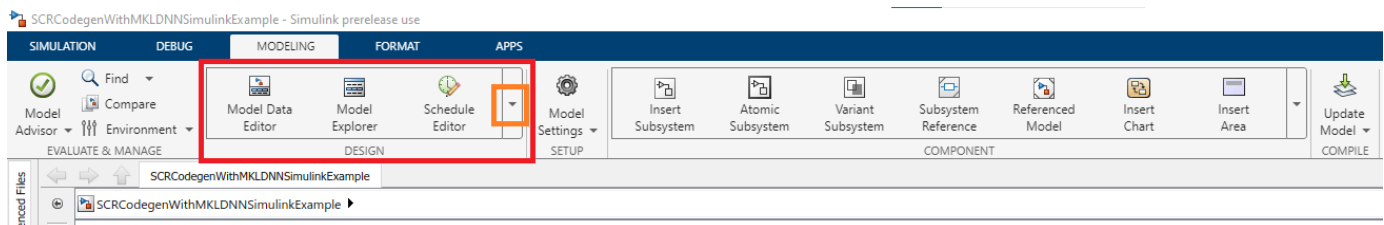
```
mainModel = "slexSpeechCommRecognitionCodegenWithMklDnnExample";
open_system(mainModel)
```



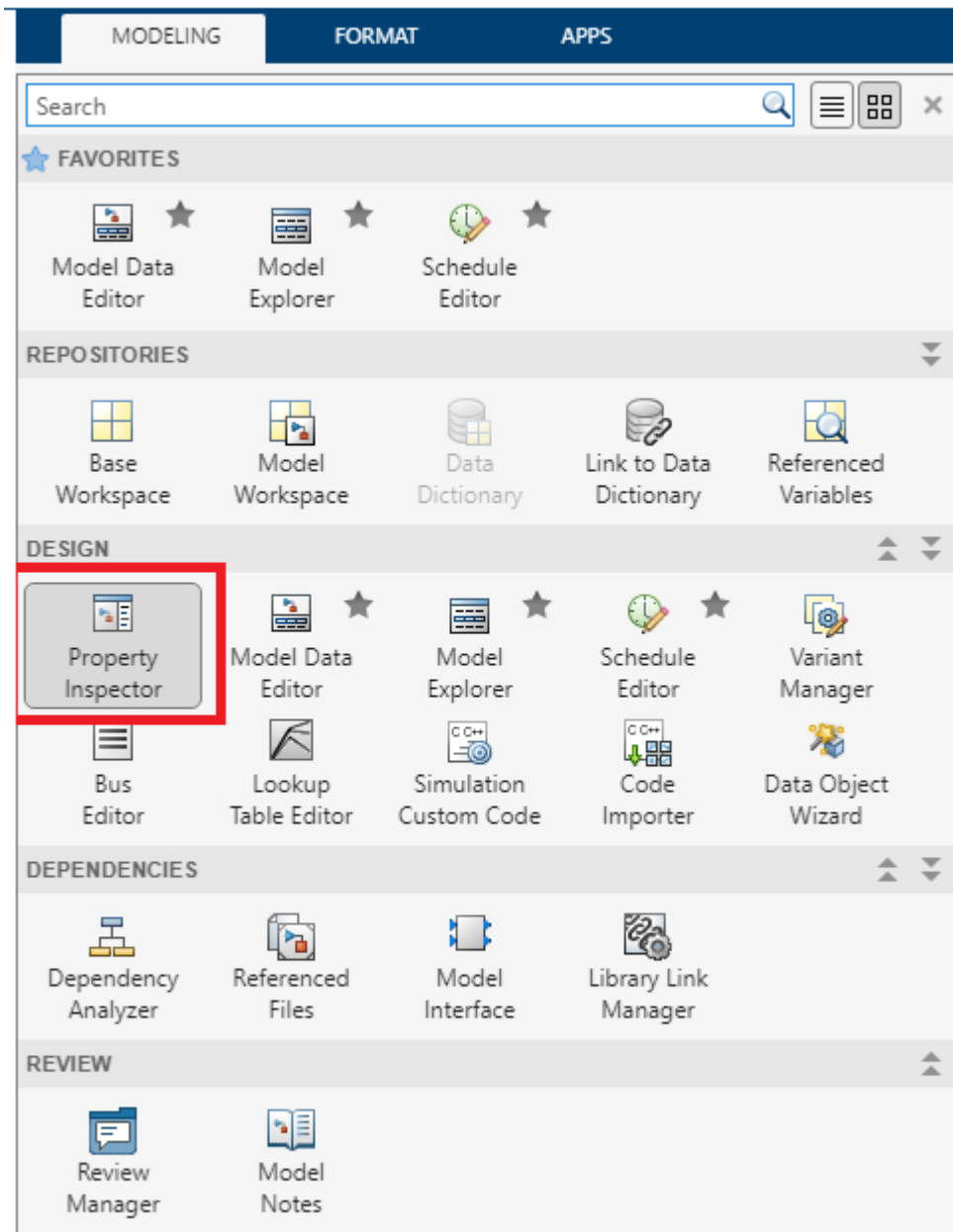
Copyright 2021, The MathWorks, Inc.



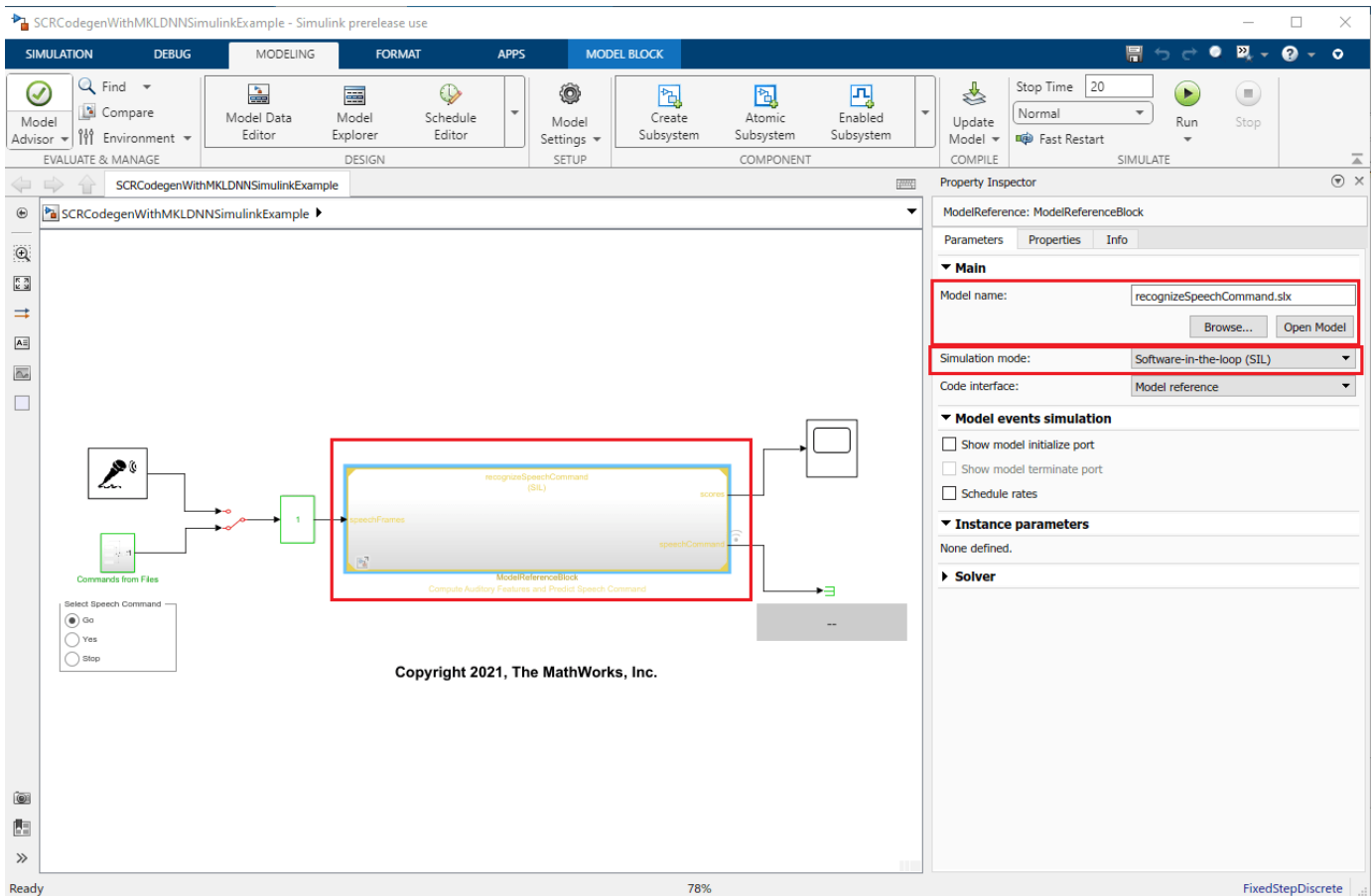
To set the Software-in-the-loop (SIL) simulation mode for the model reference block, click on **MODELING** tab.



Now click on the drop-down button as shown above, and it will open a window. Select **Property Inspector** as shown below.



You will get a **Property Inspector** window at the right of your model. Click on the Model block to get its **Property Inspector**. If the * Model name* is not set, browse for the recognizeSpeechCommand.slx and set the **Model name**. Now set **Simulation mode** to Software-in-the-loop (SIL) as shown.



Run the model to deploy the `recognizeSpeechCommand.slx` on your computer and perform speech command recognition.

```
set_param(mainModel, StopTime="20")
sim(mainModel)
```

```
### Starting serial model reference code generation build.
### Starting build procedure for: recognizeSpeechCommand
### Generating code and artifacts to 'Model specific' folder structure
### Code for the model reference code generation target for model recognizeSpeechCommand is up to date.
### Saving binary information cache.
### Model reference code generation target for recognizeSpeechCommand is up to date.
```

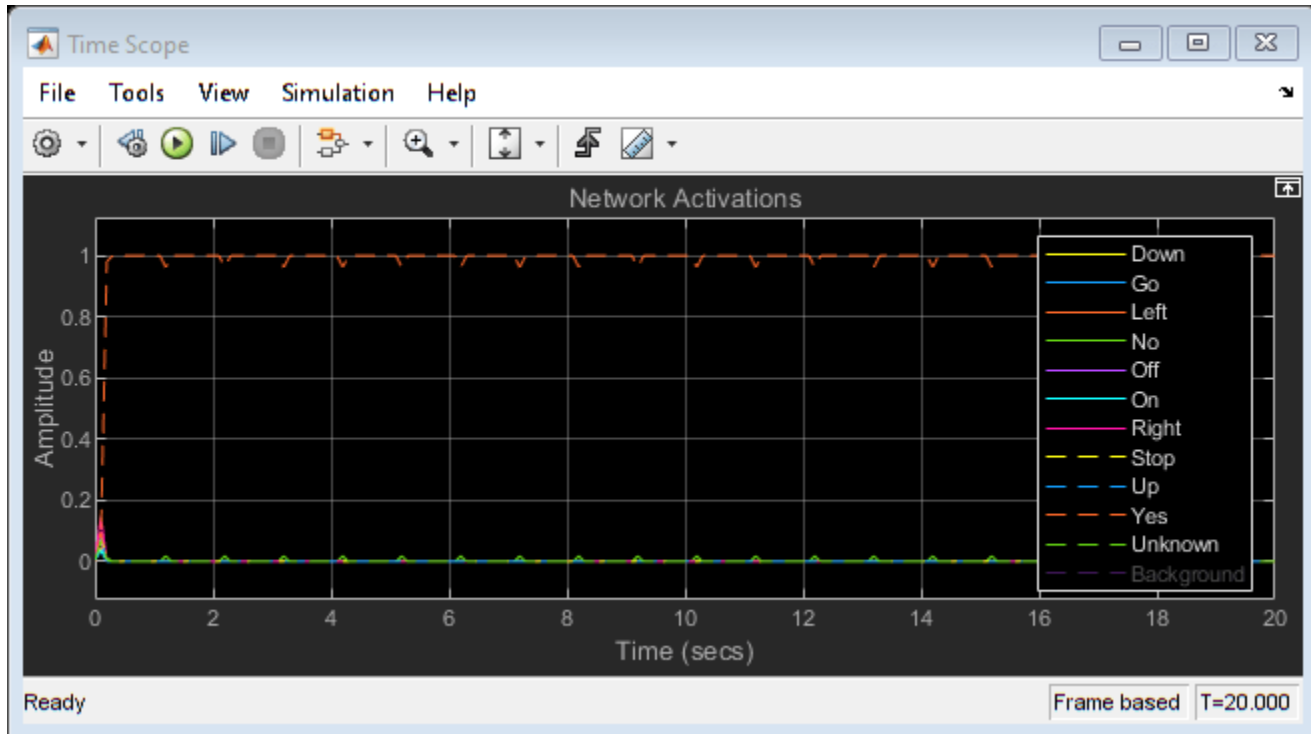
Build Summary

```
0 of 1 models built (1 models already up to date)
Build duration: 0h 0m 14.832s
### Preparing to start SIL simulation ...
### Skipping makefile generation and compilation because W:\ExampleManager\sporwal.Bdoc23a.j2106
### Starting SIL simulation for component: recognizeSpeechCommand
### Application stopped
### Stopping SIL simulation for component: recognizeSpeechCommand
```

ans =

```
Simulink.SimulationOutput:
```

```
SimulationMetadata: [1x1 Simulink.SimulationMetadata]
ErrorMessage: [0x0 char]
```



Now close the mainModel.

```
save_system(mainModel)
close_system(mainModel)
```

Other Things to Try

- Simulate "Speech Command Recognition in Simulink" on page 1-40 model using Intel® MKL - DNN library by setting the **Configuration > Simulation Target > Language** to C++.
- Compare the simulation speed of the "Speech Command Recognition in Simulink" on page 1-40 model with and without Intel® MKL - DNN library. Use Simulink Profiler (Simulink) to profile the model by setting the **Configuration > Simulation Target > Language** to C and C++.

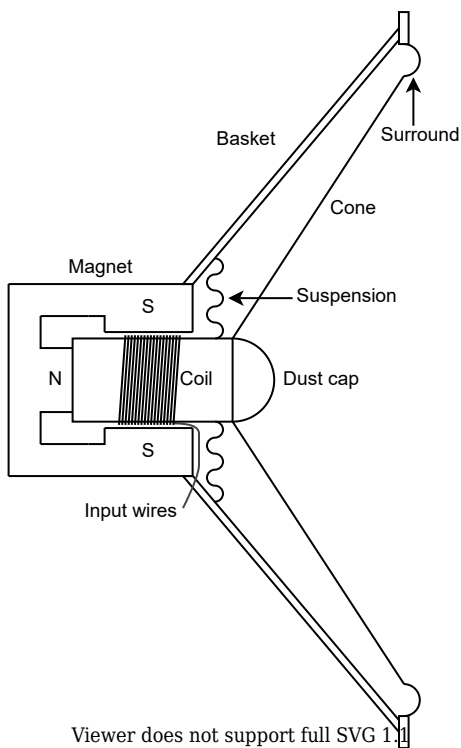
Copyright 2021-2022 The MathWorks, Inc.

Loudspeaker Modeling with Simscape™

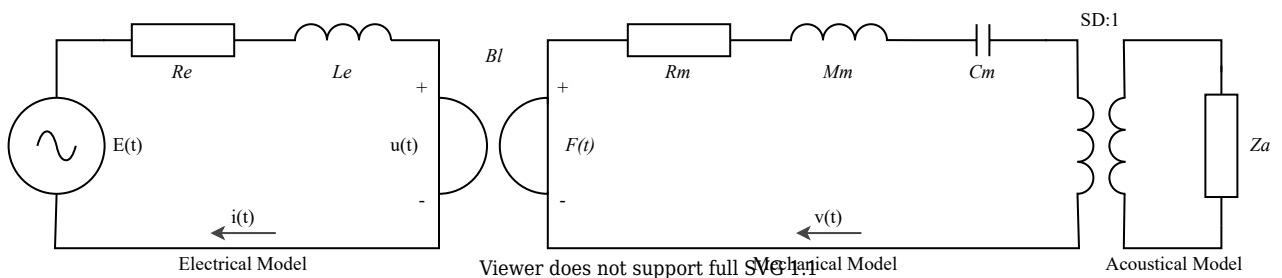
This example shows how to model a dynamic loudspeaker using linear and nonlinear lumped element models.

Introduction to Loudspeaker Modeling

Dynamic loudspeaker drivers convert electrical signals into acoustic waves using electromagnetic energy to produce mechanical movements in a cone-shaped diaphragm. Therefore, three main domains must be represented in the model: electrical, mechanical, and acoustical, in addition to the bidirectional energy conversions between these.



A common linear model for a loudspeaker is to represent it as an electrical circuit, which is known as a lumped element model. The mechanical and acoustic effects are represented by electrical circuits that are mathematically equivalent models.



For the electrical model, the motor is composed of the voice coil and the magnet. The voice coil is driven by a voltage $E(t)$ and has a resistance R_e and an inductance L_e . These two parameters depend on the wire material, diameter, length, turn radius, number of turns, and other physical properties. The magnet also has an impact on the coil inductance L_e because of the addition of a ferrous core.

The magnet creates a field in the gap with a flux density B . Multiplied by the wire length l , this is known as the force factor Bl . This is also the conversion factor between the electrical and mechanical domains, as there is a force $F(t) = Bl \cdot i(t)$ applied to the voice coil, where $i(t)$ is the electrical current applied at the input. Inversely, there is a voltage $u(t) = Bl \cdot v(t)$ generated by $v(t)$ which is analogous to the cone velocity. When converting the mechanical model to an electrical model, the coupling between them is represented by a gyrator, where velocity corresponds to the electrical current and force corresponds to the voltage.

For the mechanical model, lumped electrical components are used as analogues to mechanical properties such as mass and compliance. First, the inertia of the total moving mass (including the coil, cone, and dust cap) is analogous to the effect of an inductance M_m on varying electrical currents. Second, the stiffness of the suspension and spider is analogous to the effect of a capacitor C_m . Thirdly, the mechanical loss in the suspension system is analogous to a resistor R_m . This mechanical model forms a circuit with resonant frequency $f_s = 1/(2\pi \cdot \text{sqrt}(M_m C_m))$, which implies that the efficient frequency range of the driver depends on its mass.

For the acoustical model, the driver cone surface interface with the air is analogous to a transformer. The larger the cone is, the more mechanical energy is transformed to acoustic energy (at least for a given mass). An impedance Z_a (formed by R_a , C_a and M_a) models the radiation resistance for the front and the back of the cone. For a non-enclosed driver, this value is nonlinear but relatively small. An enclosed driver has a fixed amount of air, which creates a compliance modeled by a capacitor C_a , and any air leaks (including a vent) will contribute to the resistance R_a . For a vented enclosure, the mass of air moving in and out acts as an inductor M_a .

For simplicity, the remainder of this example assumes a loudspeaker in free space, i.e. no enclosure ($Z_a = 0$).

The equation for the electrical part of the model is:

$$E(t) = L_e \cdot \frac{di(t)}{dt} + R_e \cdot i(t) + u(t)$$

The equation for the mechanical part of the model is:

$$F(t) = M_m \cdot \frac{dv(t)}{dt} + R_m \cdot v(t) + \frac{1}{C_m} \cdot x(t)$$

Where: $v(t) = \frac{dx(t)}{dt}$

The equations for a gyrator are:

$$F(t) = Bl \cdot i(t)$$

$$u(t) = Bl \cdot v(t)$$

Linear Loudspeaker Models

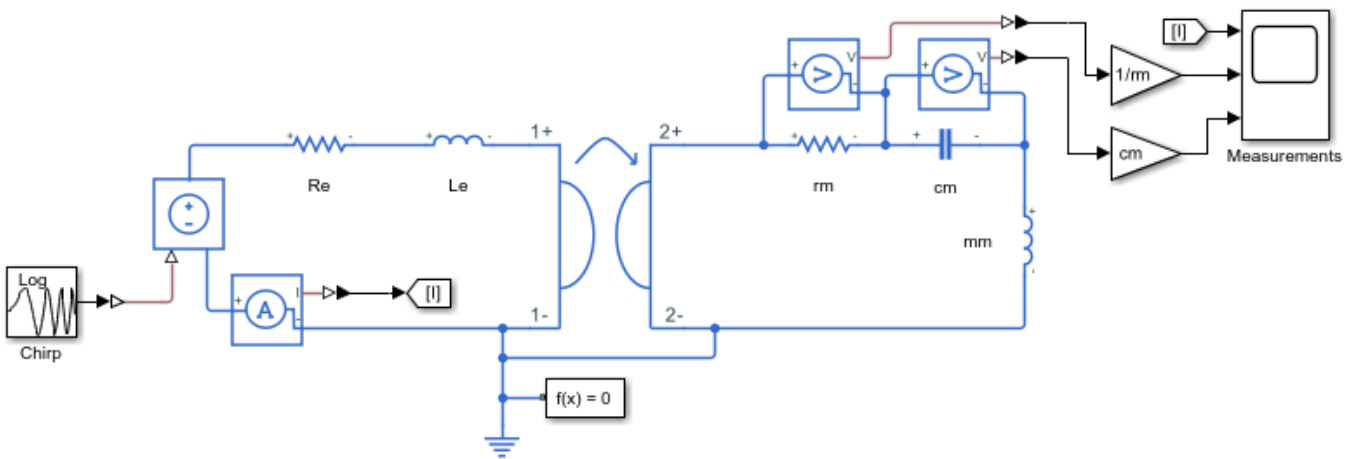
Using Simscape™, a loudspeaker can be modeled using a mixed-domain approach (mixing electrical and mechanical domains), or with a familiar model that converts the mechanical domain into the electrical domain.

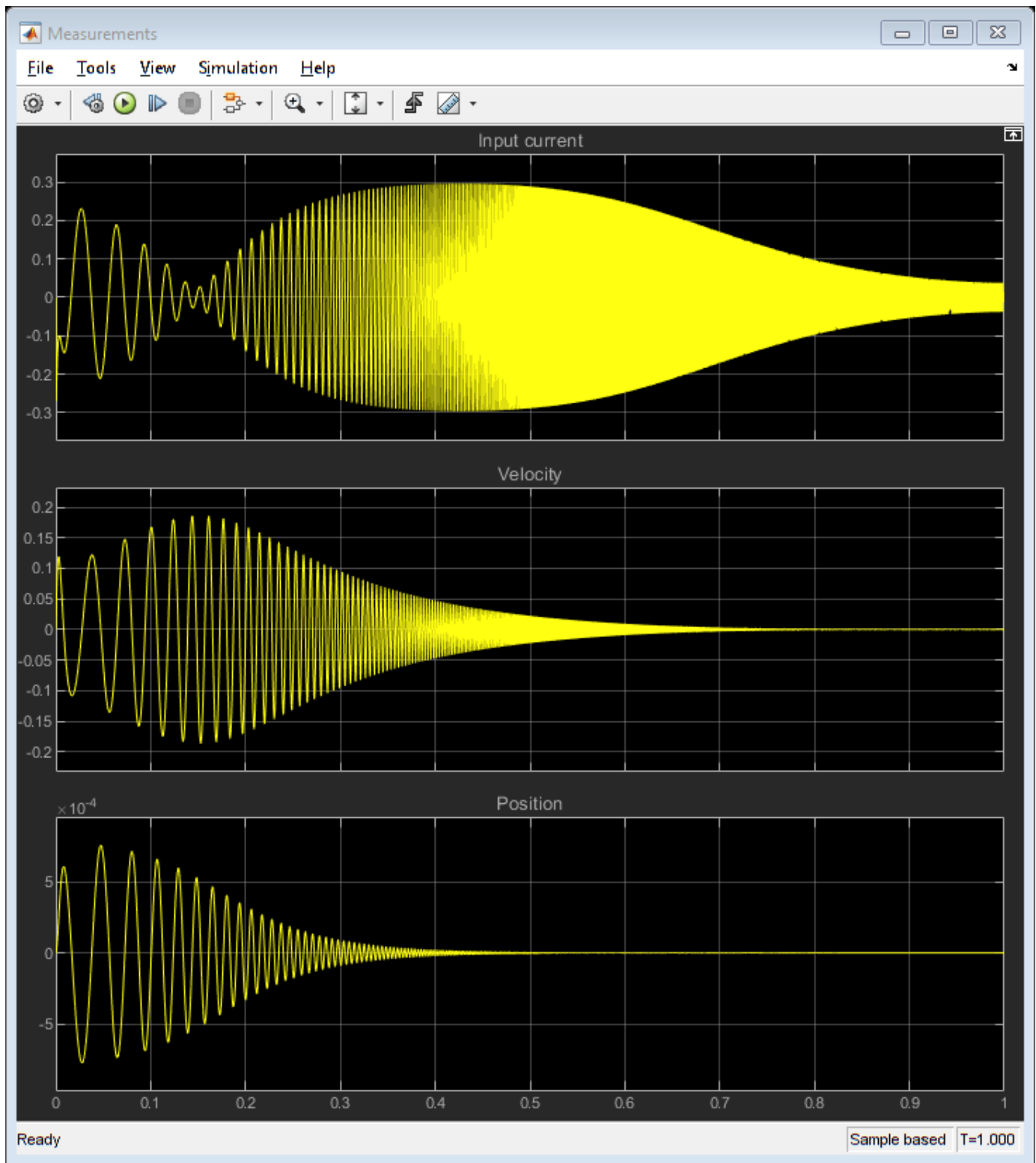
First, implement the electrical model shown above, using a gyrator directly.

```
model = 'LinearGyrator';
open_system(model);
sim(model,1);
```

Linear Gyrator Model

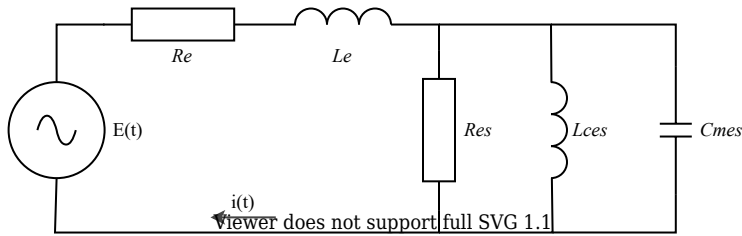
Copyright 2021 The MathWorks, Inc.





```
close_system(model,0)
```


For analysis, the gyrator is often removed by rearranging the circuit topology and the elements values.



It can be shown that this circuit behaves the same as above with the following components:

$$R_{es} = \frac{(Bl)^2}{R_{ms}}$$

$$C_{mes} = \frac{M_{ms}}{(Bl)^2}$$

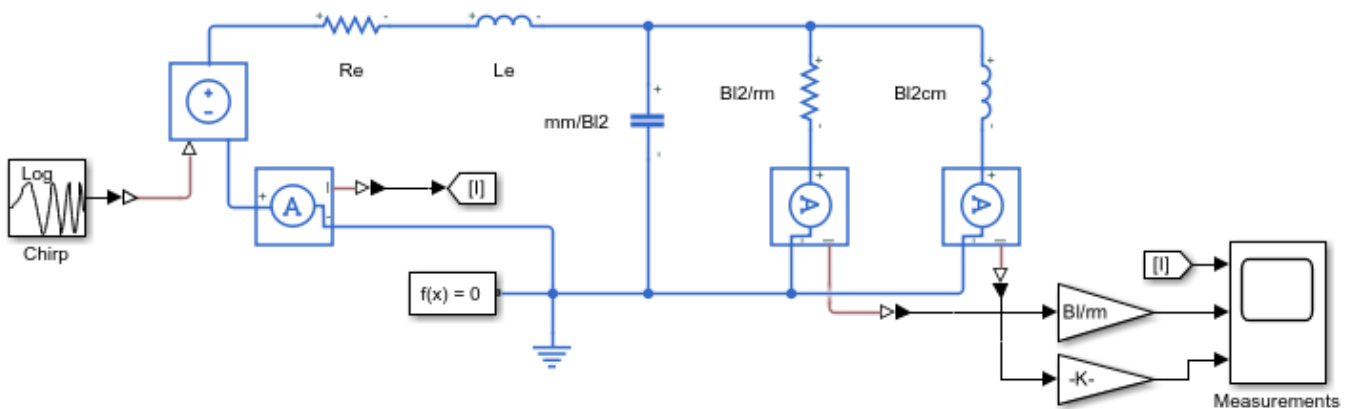
$$L_{ces} = (Bl)^2 \cdot C_{ms}$$

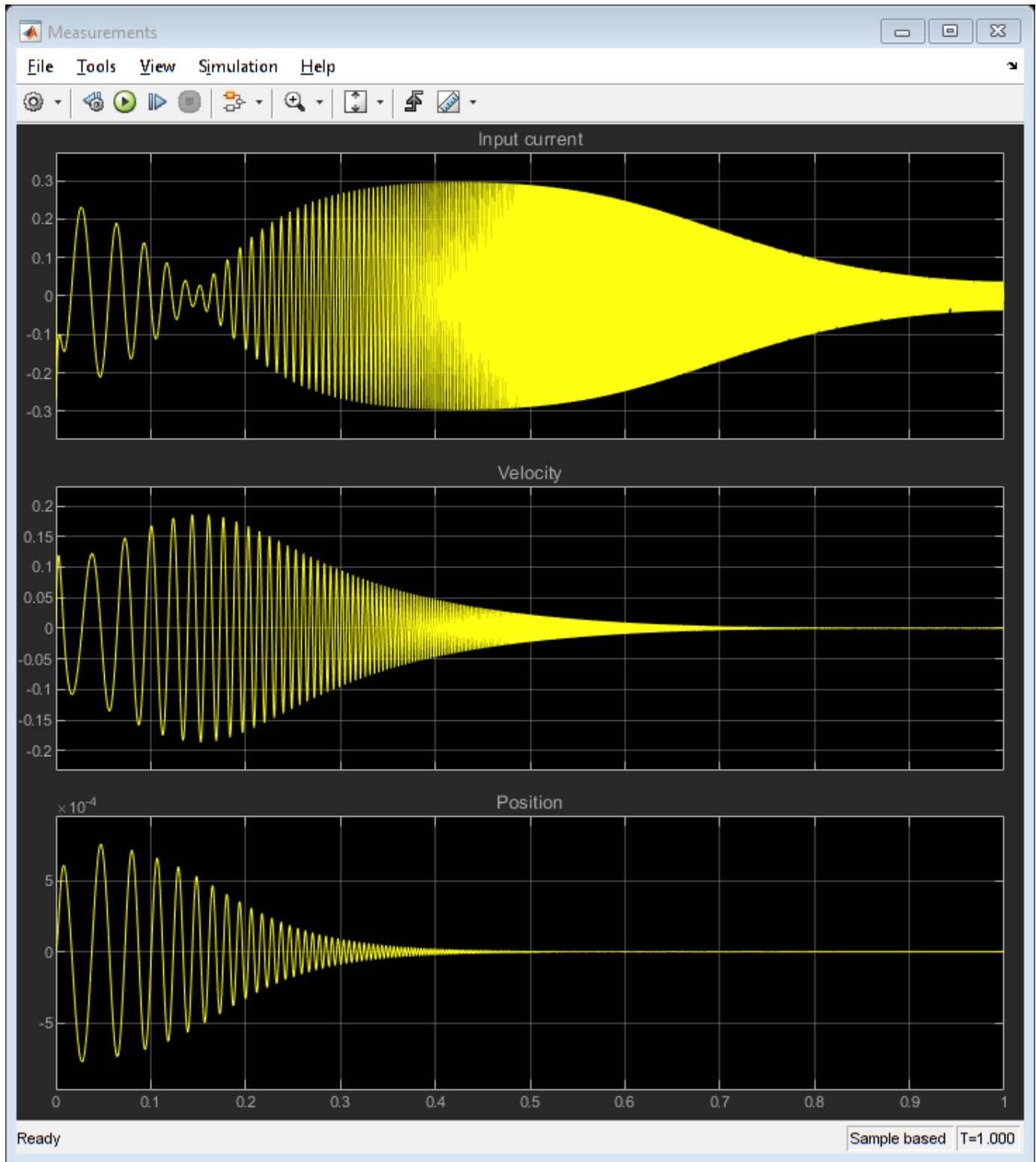
This simplified circuit produces the exact same results as before.

```
model = 'LinearCircuit';
open_system(model);
sim(model,1);
```

Linear Circuit Model

Copyright 2021 The MathWorks, Inc.





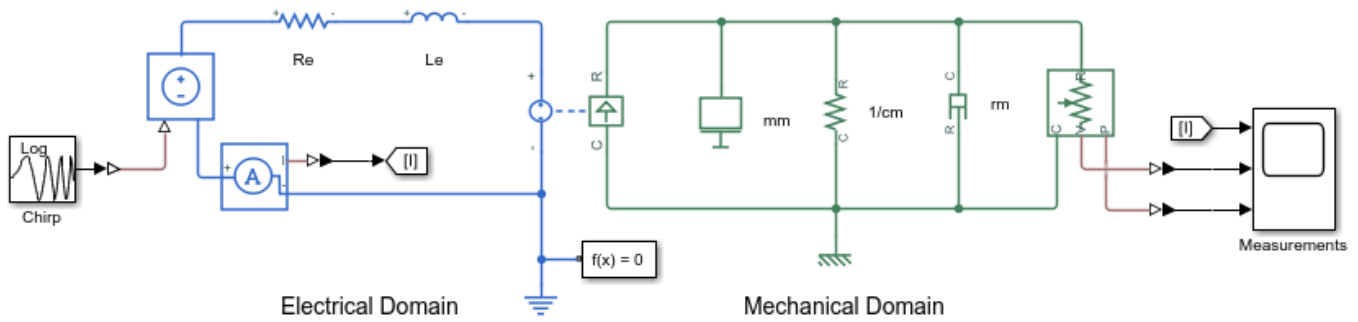
```
close_system(model,0)
```

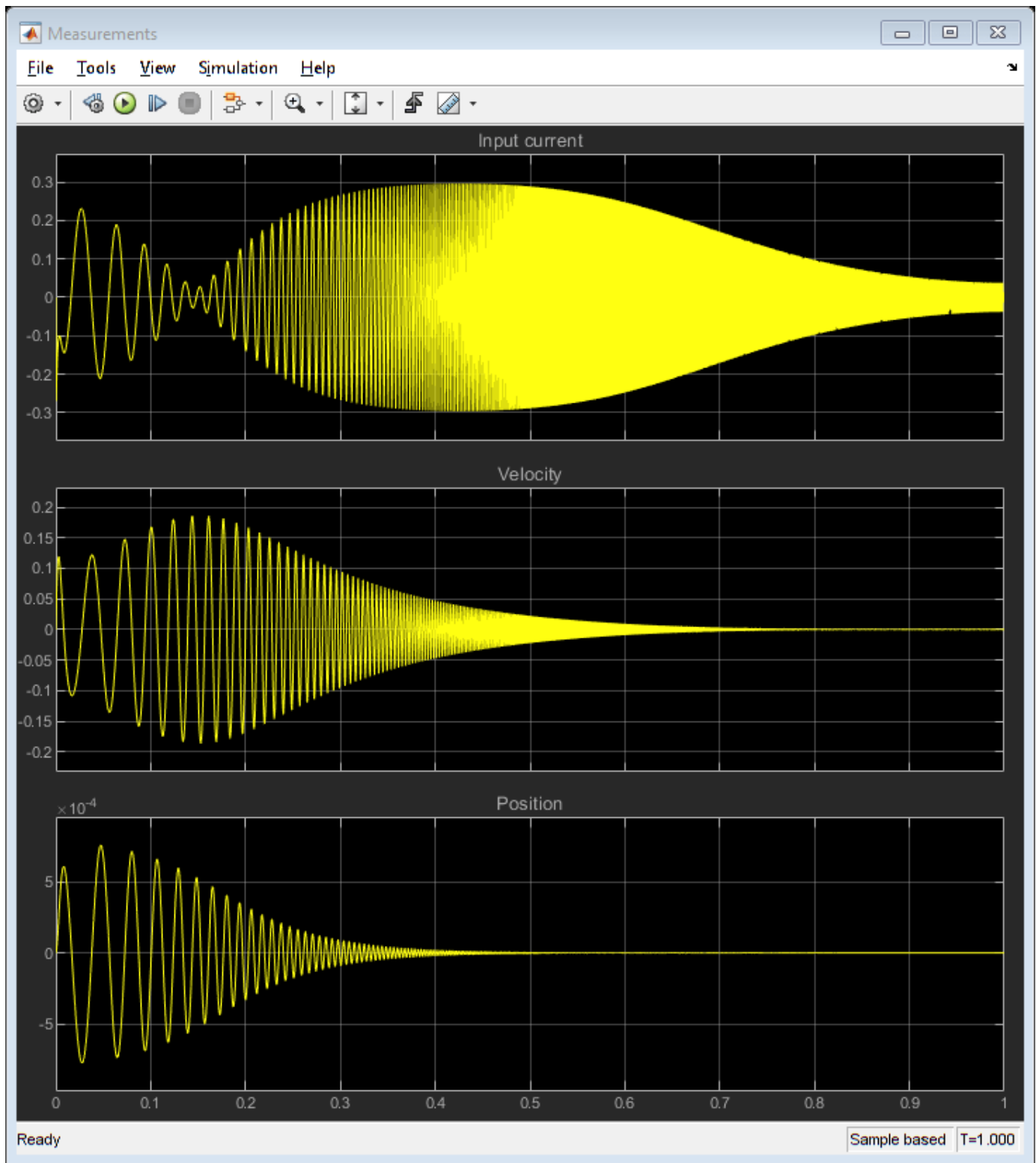
Simscape™ also allows mixing electrical and mechanical elements, so the loudspeaker model can be simulated without any physical domain conversions. Again, the same results are obtained.

```
model = 'LinearMixedDomain';
open_system(model);
sim(model,1);
```

Mixed-Domain Model

Copyright 2021 The MathWorks, Inc.





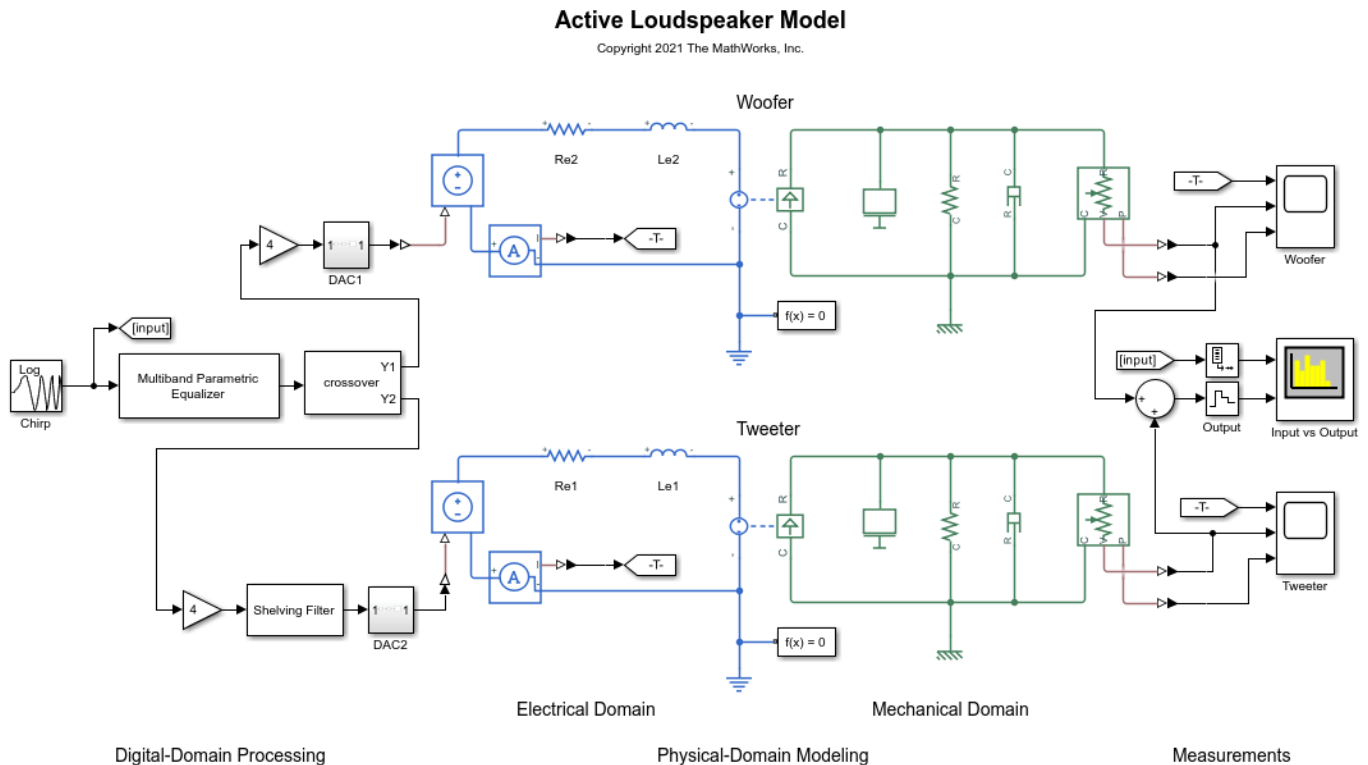
```
close_system(model,0)
```

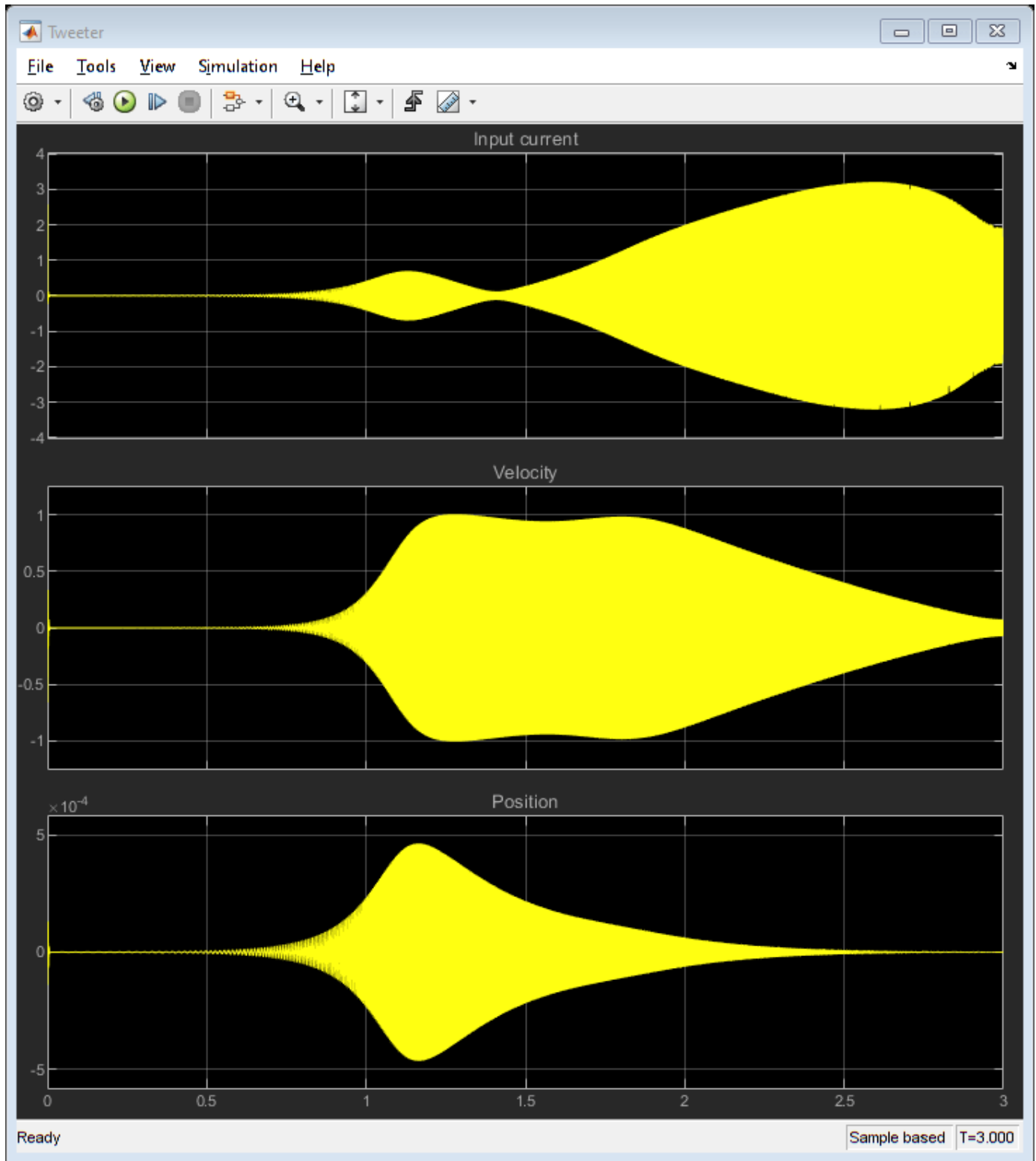
Other elements can easily be added to this circuit model to account for a closed or vented enclosure.

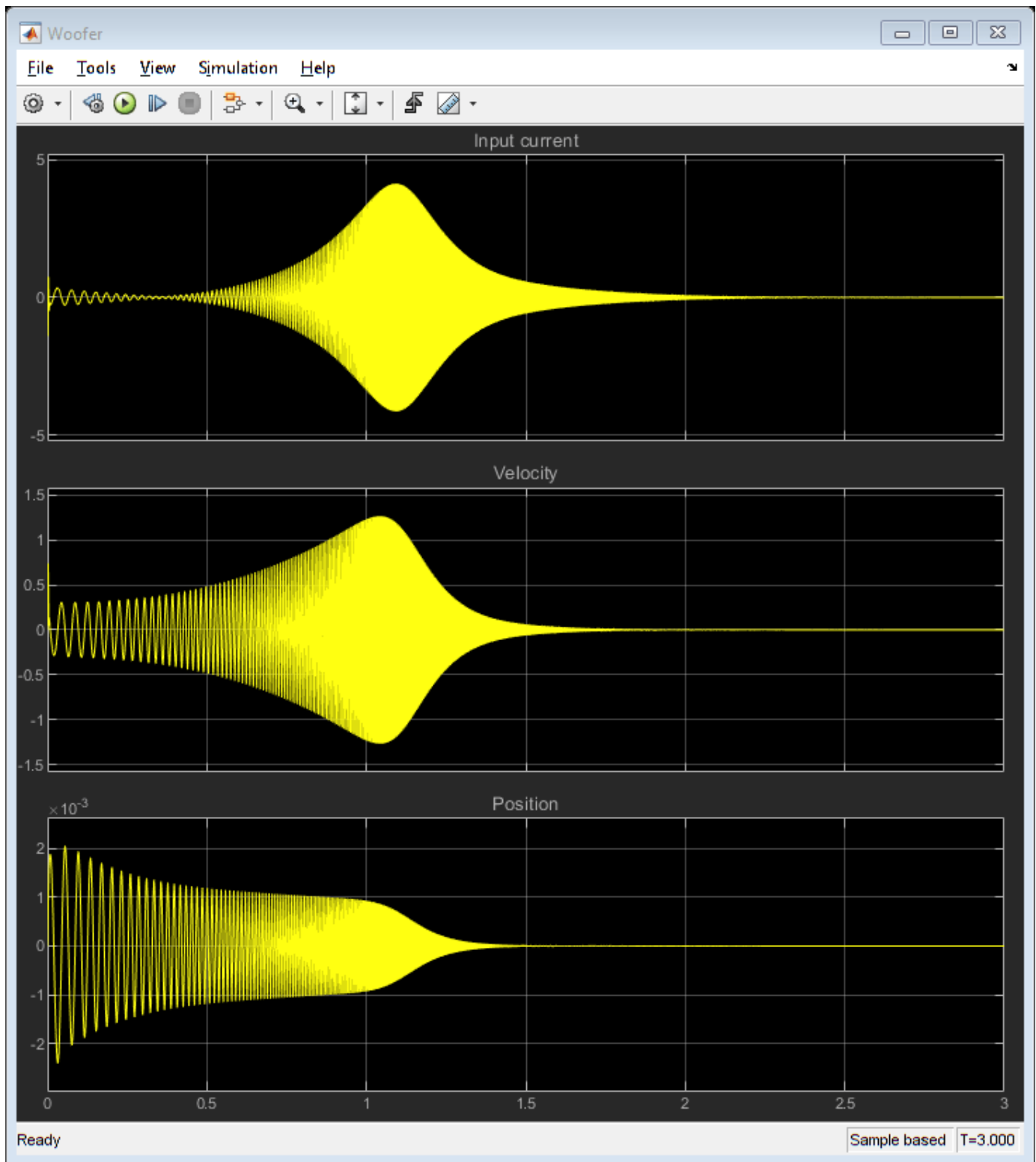
Adding DSP to a Physical Model

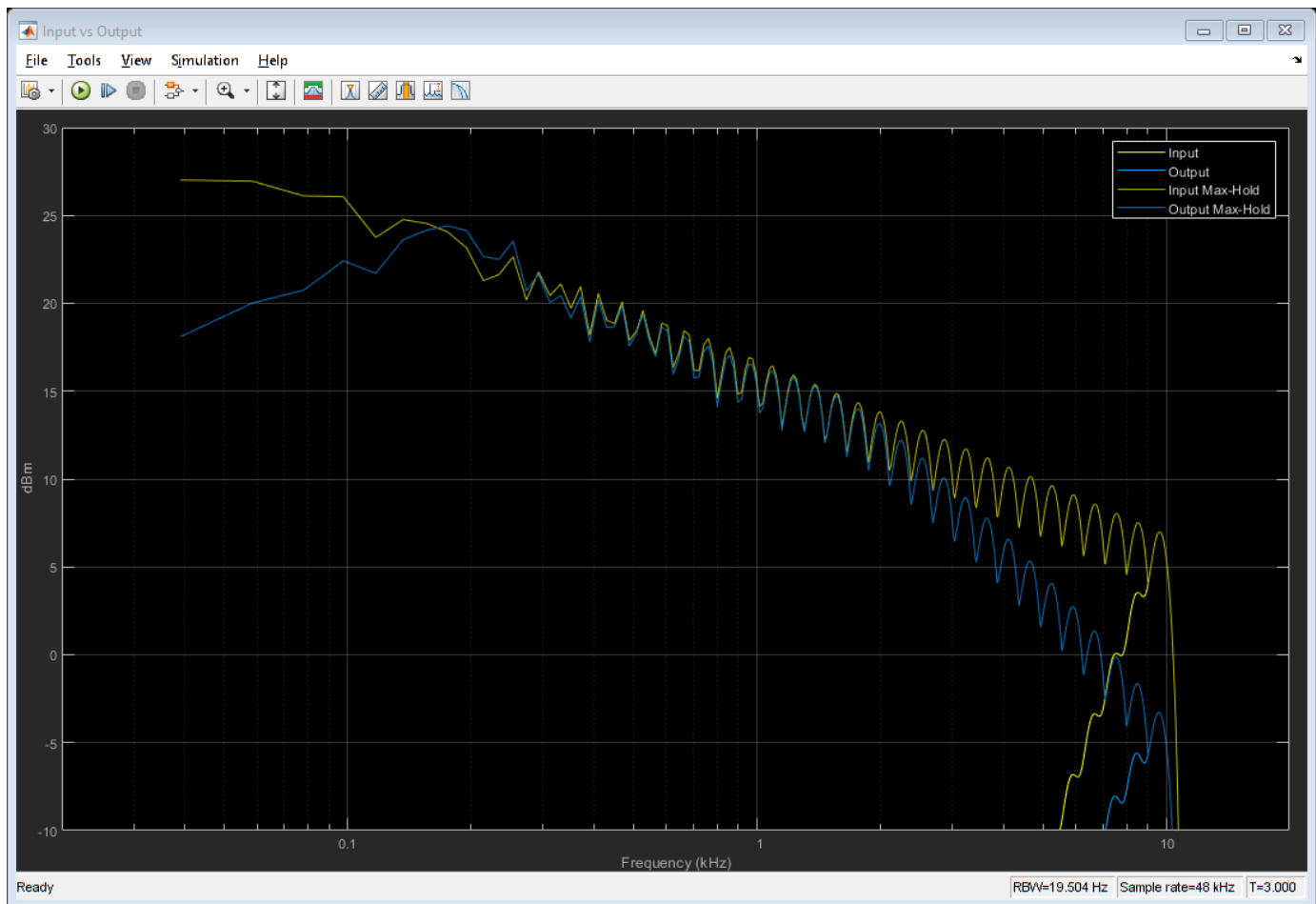
In addition to combining two physical domains in one simulation, digital signal processing algorithms can be included. The following model represents an active loudspeaker with a woofer and a tweeter. The crossover, parametric EQ and shelving filters are implemented in the digital domain, followed by an optimized power amplifier for each driver. The output of each driver is measured separately, and the combined output is compared to the log-chirp input in the frequency domain.

```
model = 'MixedModeling';
open_system(model);
sim(model,3);
```









```
close_system(model,0)
```

Modeling Nonlinear Elements

Several loudspeaker elements are nonlinear. For example, the voice coil force factor and inductance vary with its position in the magnet. Furthermore, the suspension spring rate changes at the extremities of its displacement range.

Linear components in the previous model can be replaced by custom versions that implement the nonlinearities that are required. For example, the spring component can define the spring rate K as $K = K_0 + K_1 \cdot x + K_2 \cdot x^2 + K_3 \cdot x^3 + K_4 \cdot x^4$, where x is the displacement and K_n are the polynomial coefficients (K_0 being the spring rate at displacement zero).

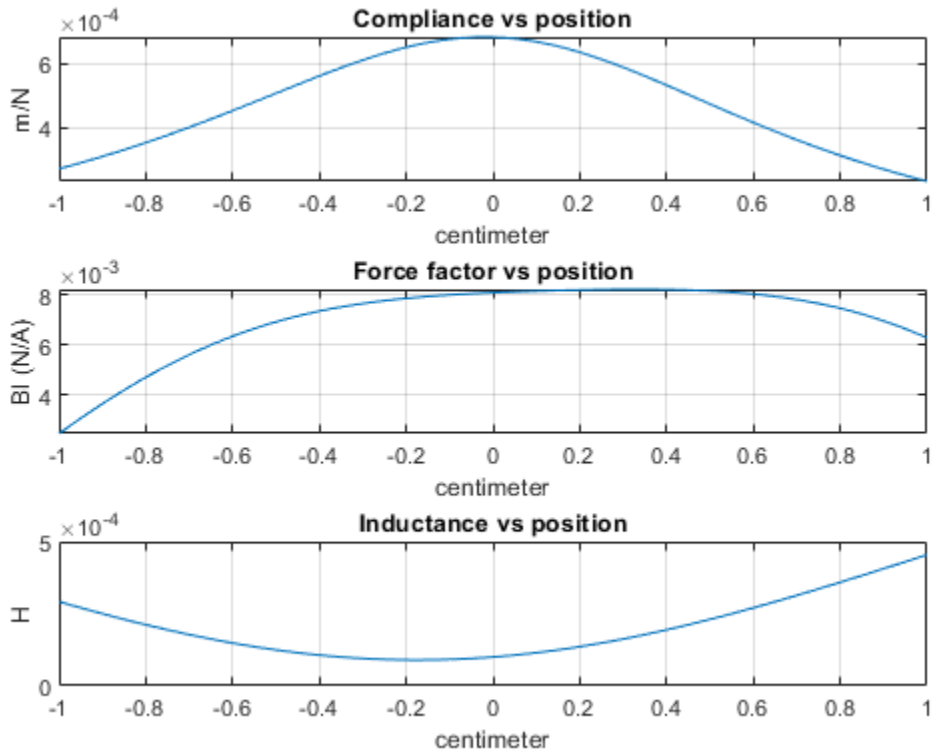
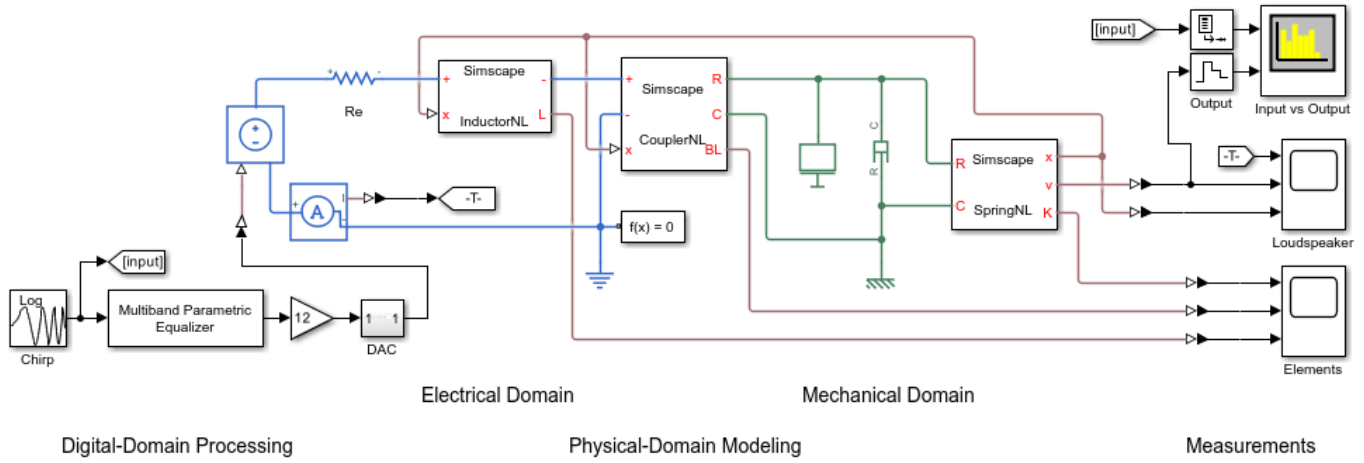
Plot sample values for compliance, force factor and inductance. Run model that implements the "woofer" driver of the previous model, but with three linear components replaced by nonlinear components.

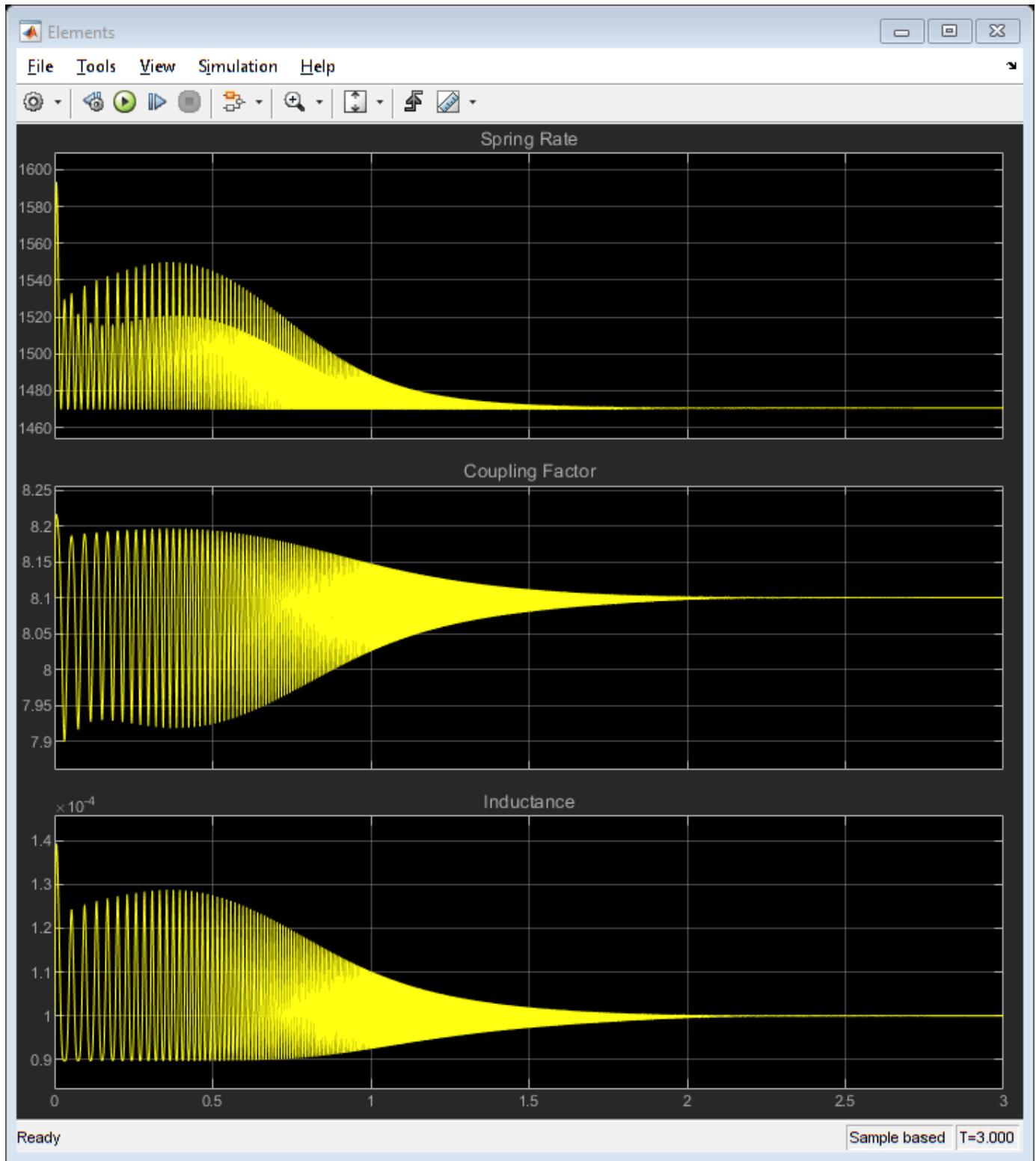
```
plotNonLinearBHK;

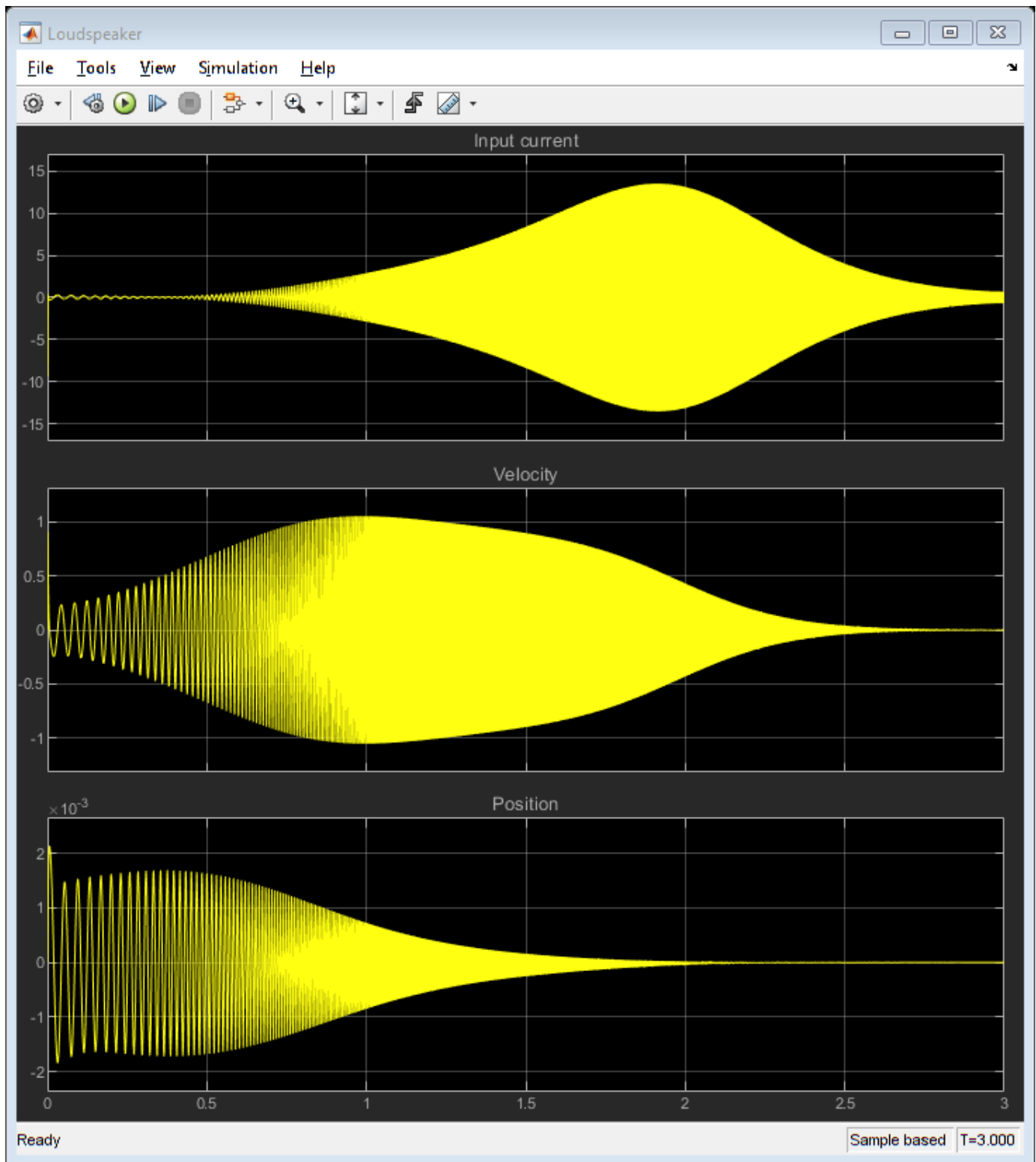
model = 'NonLinearBHK';
open_system(model);
sim(model,3);
```

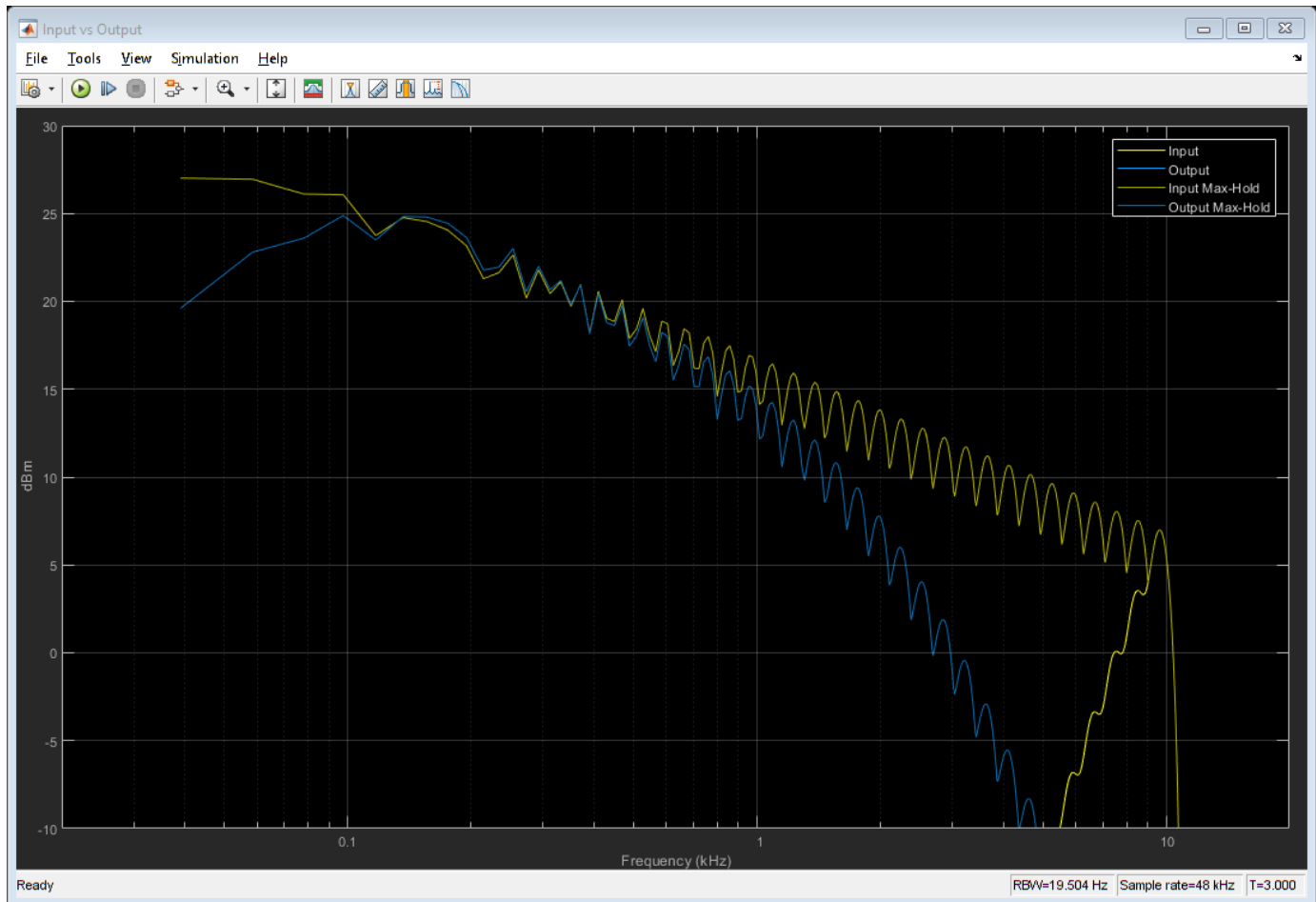

Non-Linear Loudspeaker Model

Copyright 2021 The MathWorks, Inc.









```
close_system(model,0)
```

Starting from these examples, add components to model an enclosure, or implement your own nonlinear elements. Simscape™ will allow you to do this using the domain of your choice (electrical, mechanical). You can also test any digital pre-processing that is required, all in one model.

Definitions

$E(t)$ input voltage

$i(t)$ input current

L_e voice coil inductance

R_e voice coil resistance

Bl force factor

$F(t)$ force applied to the diaphragm

$v(t)$ diaphragm velocity

$x(t)$ diaphragm displacement

M_m moving mass

R_m mechanical loss

C_m suspension compliance

Z_a acoustical impedance

Investigate Audio Classifications Using Deep Learning Interpretability Techniques

This example shows how to use interpretability techniques to investigate the predictions of a deep neural network trained to classify audio data.

Deep learning networks are often described as "black boxes" because why a network makes a certain decision is not always obvious. You can use interpretability techniques to translate network behavior into output that a person can interpret. This interpretable output can then answer questions about the predictions of a network. This example uses interpretability techniques that explain network predictions using visual representations of what a network is "looking" at. You can then use these visual representations to see which parts of the input images the network is using to make decisions.

This example uses transfer learning to retrain VGGish, a pretrained convolutional neural network, to classify a new set of audio signals.

Load Data

Download and unzip the environmental sound classification data set. This data set consists of recordings labeled as one of 10 different audio sound classes (ESC-10). Download the ESC-10.zip zip file from the MathWorks website, then unzip the file.

```
rng("default")
zipFile = matlab.internal.examples.downloadSupportFile("audio","ESC-10.zip");

filepath = fileparts(zipFile);
dataFolder = fullfile(filepath,"ESC-10");
unzip(zipFile,dataFolder)
```

Create an `audioDatastore` object to manage the data and split it into training and validation sets. Use `countEachLabel` to display the distribution of sound classes and the number of unique labels.

```
ads = audioDatastore(dataFolder,IncludeSubfolders=true,LabelSource="foldernames");
labelTable = countEachLabel(ads)
```

```
labelTable=10×2 table
      Label      Count
-----
chainsaw      40
clock_tick    40
crackling_fire 40
crying_baby   40
dog           40
helicopter    40
rain          40
rooster       38
sea_waves     40
sneezing      40
```

Determine the total number of classes.

```
classes = labelTable.Label;
numClasses = size(labelTable,1);
```

Use `splitEachLabel` to split the data set into training and validation sets. Use 80% of the data for training and 20% for validation.

```
[adsTrain,adsValidation] = splitEachLabel(ads,0.8,0.2);
```

The VGGish pretrained network requires preprocessing of the audio signals into log mel spectrograms. The supporting function `helperAudioPreprocess`, defined at the end of this example, takes as input an `audioDatastore` object and the overlap percentage between log mel spectrograms and returns matrices of predictors and responses suitable for input to the VGGish network. Each audio file is split into several segments to feed into the VGGish network.

```
overlapPercentage = 75;
```

```
[trainFeatures,trainLabels] = helperAudioPreprocess(adsTrain,overlapPercentage);
[validationFeatures,validationLabels,segmentsPerFile] = helperAudioPreprocess(adsValidation,over
```

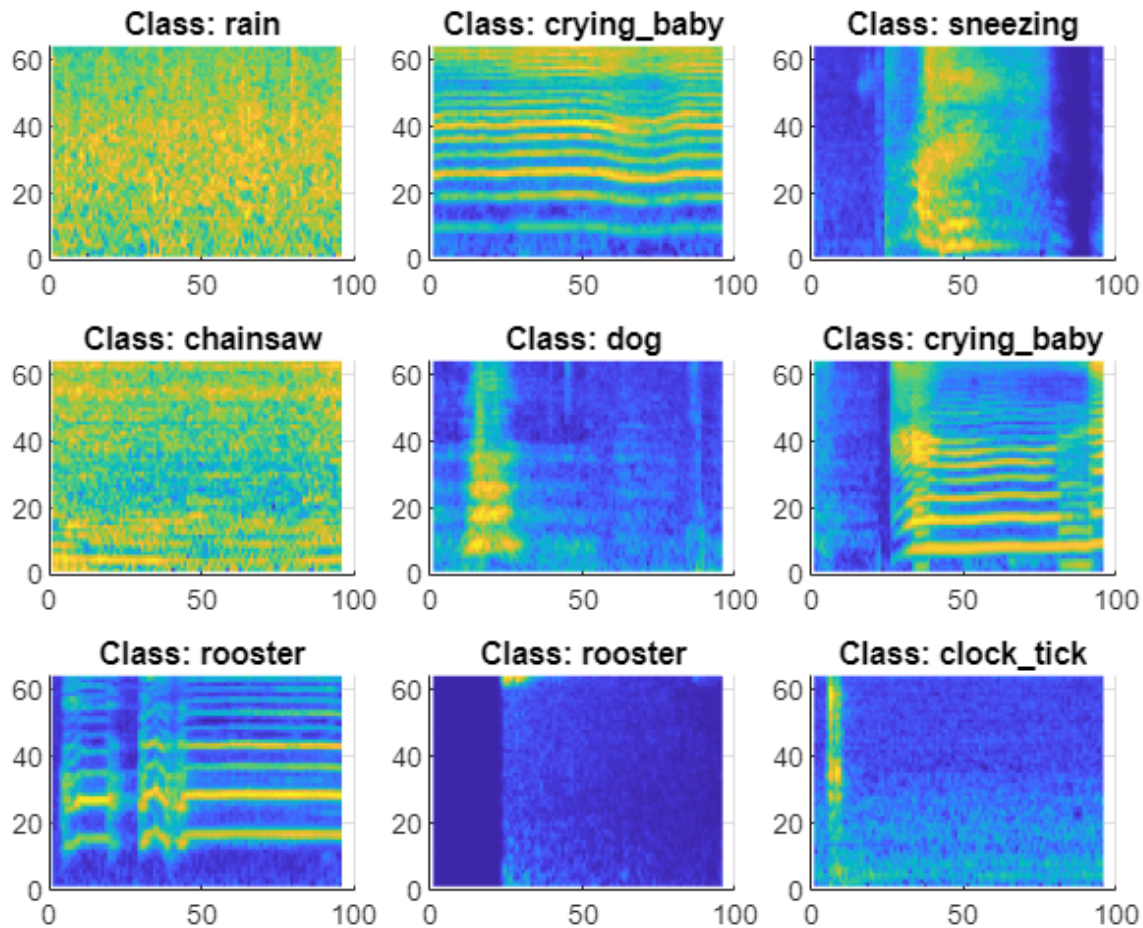
Visualize Data

View a random sample of the data.

```
numImages = 9;
idxSubset = randi(numel(trainLabels),1,numImages);
```

```
viewingAngle =  ;
```

```
figure
tiledlayout("flow",TileSpacing="compact");
for i = 1:numImages
    img = trainFeatures(:,:,,idxSubset(i));
    label = trainLabels(idxSubset(i));
    nexttile
    surf(img,EdgeColor="none")
    view(viewingAngle)
    title("Class: " + string(label),interpreter="none")
end
colormap parula
```



Build Network

This example uses transfer learning to retrain VGGish, a pretrained convolutional neural network, to classify a new set of audio signals.

Download VGGish Network

Download and unzip the Audio Toolbox™ model for VGGish.

Type `vggish` in the Command Window. If the Audio Toolbox model for VGGish 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.

Load the VGGish model and convert it to a `layerGraph` object.

```
pretrainedNetwork = vggish;
lgraph = layerGraph(pretrainedNetwork.Layers);
```


Prepare Network for Transfer Learning

Prepare the network for transfer learning by replacing the final layers with new layers suitable for the new data. You can adapt VGGish for the new data programmatically or interactively using Deep Network Designer. For an example showing how to use Deep Network Designer to perform transfer learning with an audio classification network, see “Transfer Learning with Pretrained Audio Networks in Deep Network Designer” (Deep Learning Toolbox).

Use `removeLayers` to remove the final regression output layer from the graph. After you remove the regression layer, the new final layer of the graph is a ReLU layer named `EmbeddingBatch`.

```
lgraph = removeLayers(lgraph, "regressionoutput");
lgraph.Layers(end)
```

```
ans =
    ReLULayer with properties:
```

```
    Name: 'EmbeddingBatch'
```

Use `addLayers` to add a `fullyConnectedLayer`, a `softmaxLayer`, and a `classificationLayer` to the layer graph.

```
lgraph = addLayers(lgraph, fullyConnectedLayer(numClasses, Name="FCFinal"));
lgraph = addLayers(lgraph, softmaxLayer(Name="softmax"));
lgraph = addLayers(lgraph, classificationLayer(Name="classOut"));
```

Use `connectLayers` to append the fully connected, softmax, and classification layers to the layer graph.

```
lgraph = connectLayers(lgraph, "EmbeddingBatch", "FCFinal");
lgraph = connectLayers(lgraph, "FCFinal", "softmax");
lgraph = connectLayers(lgraph, "softmax", "classOut");
```

Specify Training Options

To define the training options, use the `trainingOptions` function. Set the solver to "adam" and train for five epochs with a mini-batch size of 128. Specify an initial learning rate of 0.001 and drop the learning rate after two epochs by multiplying by a factor of 0.5. Monitor the network accuracy during training by specifying validation data and the validation frequency.

```
miniBatchSize = 128;
options = trainingOptions("adam", ...
    MaxEpochs=5, ...
    MiniBatchSize=miniBatchSize, ...
    InitialLearnRate = 0.001, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropPeriod=2, ...
    LearnRateDropFactor=0.5, ...
    ValidationData={validationFeatures, validationLabels}, ...
    ValidationFrequency=50, ...
    Shuffle="every-epoch");
```

Train Network

To train the network, use the `trainNetwork` function. By default, `trainNetwork` uses a GPU if one is available. Otherwise, it uses a CPU. Training on a GPU requires Parallel Computing Toolbox™ and a

supported GPU device. For information on supported devices, see “GPU Computing Requirements” (Parallel Computing Toolbox). You can also specify the execution environment by using the `ExecutionEnvironment` name-value argument of `trainingOptions`.

```
[net,netInfo] = trainNetwork(trainFeatures,trainLabels,lgraph,options);
```

Training on single GPU.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Validation Accuracy	Mini-batch Loss	Validation Loss
1	1	00:00:17	3.91%	20.07%	2.4103	2.7
2	50	00:00:22	96.88%	82.57%	0.1491	0.7
3	100	00:00:27	92.19%	83.75%	0.1730	0.7
4	150	00:00:32	94.53%	85.15%	0.1654	0.8
5	200	00:00:37	96.09%	85.96%	0.1747	0.8
5	210	00:00:38	93.75%	86.03%	0.1643	0.7

Training finished: Max epochs completed.

Test Network

Classify the validation mel spectrograms using the trained network.

```
[validationPredictions,validationScores] = classify(net,validationFeatures);
```

Each audio file produces multiple mel spectrograms. Combine the predictions for each audio file in the validation set using a majority-rule decision and calculate the classification accuracy.

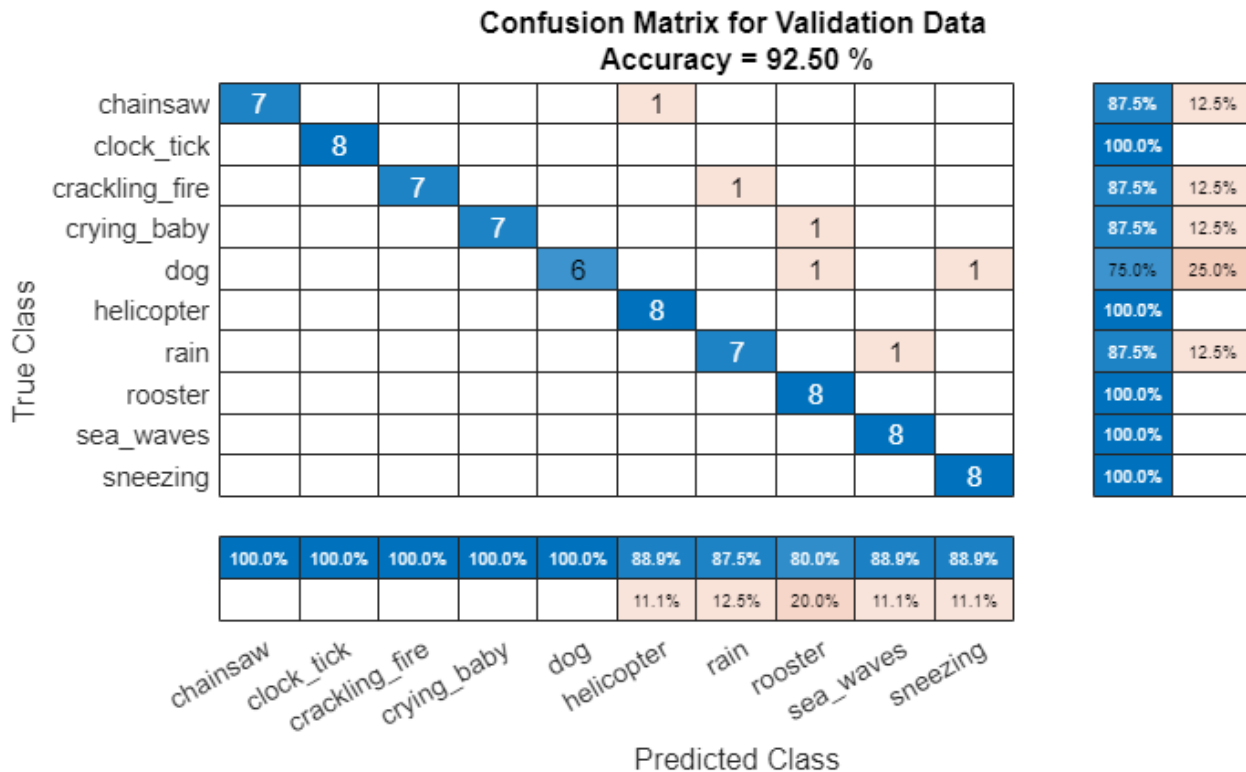
```
idx = 1;
validationPredictionsPerFile = categorical;
for ii = 1:numel(adsValidation.Files)
    validationPredictionsPerFile(ii,1) = mode(validationPredictions(idx:idx+segmentsPerFile(ii))-1);
    idx = idx + segmentsPerFile(ii);
end
```

```
accuracy = mean(validationPredictionsPerFile==adsValidation.Labels)*100
```

```
accuracy = 92.5000
```

Use `confusionchart` to evaluate the performance of the network on the validation set.

```
figure(Units="normalized",Position=[0.2 0.2 0.5 0.5]);
cm = confusionchart(adsValidation.Labels,validationPredictionsPerFile);
cm.Title = sprintf("Confusion Matrix for Validation Data \nAccuracy = %0.2f %%",accuracy);
cm.ColumnSummary = "column-normalized";
cm.RowSummary = "row-normalized";
```



Visualize Predictions

View a random sample of the input data with the true and predicted class labels.

```

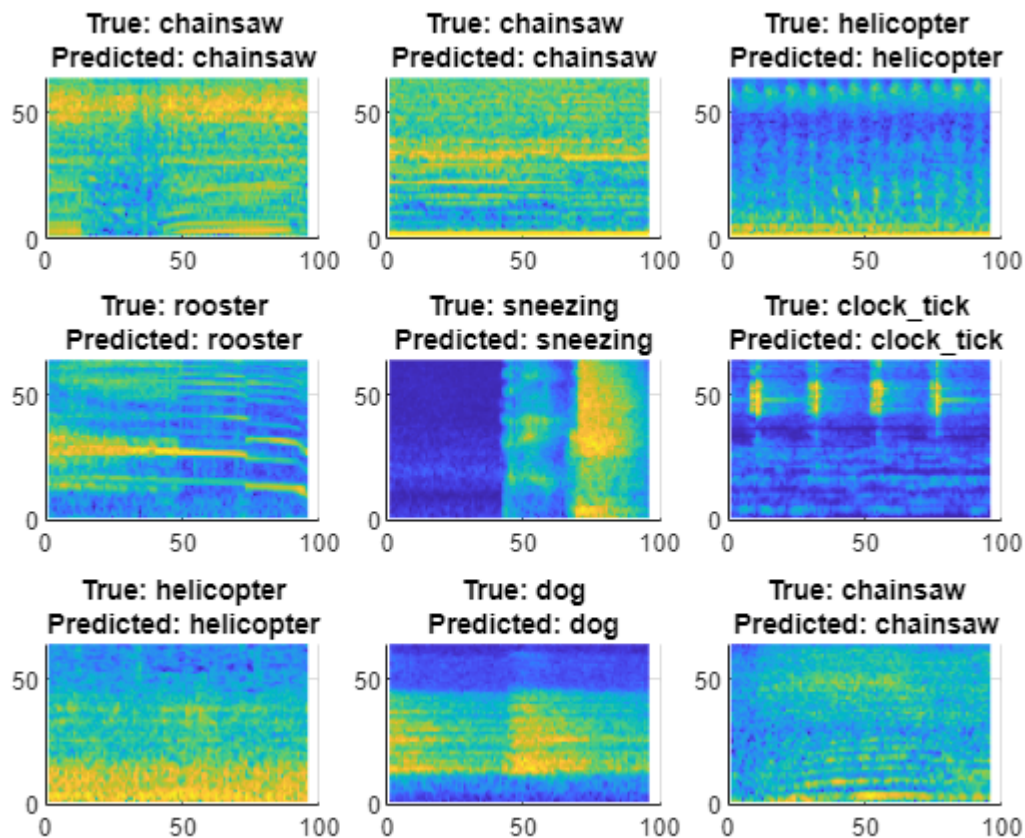
numImages = 9  ;
idxSubset = randi(numel(validationLabels),1,numImages);

viewingAngle =  ;

figure
t1 = tiledlayout("flow",TileSpacing="compact");
for i = 1:numImages
    img = validationFeatures(:,:,,idxSubset(i));
    YPred = validationPredictions(idxSubset(i));
    YTrue = validationLabels(idxSubset(i));

    nexttile
    surf(img,EdgeColor="none")
    view(viewingAngle)
    title({"True: " + string(YTrue),"Predicted: " + string(YPred)},interpreter= "none")
end
colormap parula

```



The x-axis represents time, the y-axis represents frequency, and the colormap represents decibels. For several of the classes, you can see interpretable features. For example, the spectrogram for the `clock_tick` class shows a repeating pattern through time representing the ticking of a clock. The first spectrogram from the `helicopter` class has the constant, loud, low-frequency sound of the helicopter engine and a repeating high-frequency sound representing the spinning of the helicopter blades.

As the network is a convolutional neural network with image input, the network might use these features when making classification decisions. You can investigate this hypothesis using deep learning interpretability techniques.

Investigate Predictions

Investigate the predictions of the validation mel spectrograms. For each input, generate the Grad-CAM (`gradCAM` (Deep Learning Toolbox)), LIME (`imageLIME` (Deep Learning Toolbox)), and occlusion sensitivity (`occlusionSensitivity` (Deep Learning Toolbox)) maps for the predicted classes. These methods take an input image and a class label and produce a map indicating the regions of the image that are important to the score for the specified class. Each visualization method has a specific approach that determines the output it produces.

- Grad-CAM — Use the gradient of the classification score with respect to the convolutional features determined by the network to understand which parts of the image are most important for

classification. The places where the gradient is large are the places where the final score depends most on the data.

- LIME — Approximate the classification behavior of a deep learning network using a simpler, more interpretable model, such as a linear model or a regression tree. The simple model determines the importance of features of the input data as a proxy for the importance of the features to the deep learning network.
- Occlusion sensitivity — Perturb small areas of the input by replacing them with an occluding mask, typically a gray square. As the mask moves across the image, the technique measures the change in probability score for a given class.

Comparing the results of different interpretability techniques is important for verifying the conclusions you make. For more information about these techniques, see “Deep Learning Visualization Methods” (Deep Learning Toolbox).

Using the supporting function `helperPlotMaps`, defined at the end of this example, plot the input log mel spectrogram and the three interpretability maps for a selection of images and their predicted classes.

```
viewingAngle =  ;
imgIdx = [250 500 750];
numImages = length(imgIdx);

figure
t2 = tiledlayout(numImages,4,TileSpacing="compact");
for i = 1:numImages

    img = validationFeatures(:,:,,imgIdx(i));
    YPred = validationPredictions(imgIdx(i));
    YTrue = validationLabels(imgIdx(i));

    mapClass = YPred;

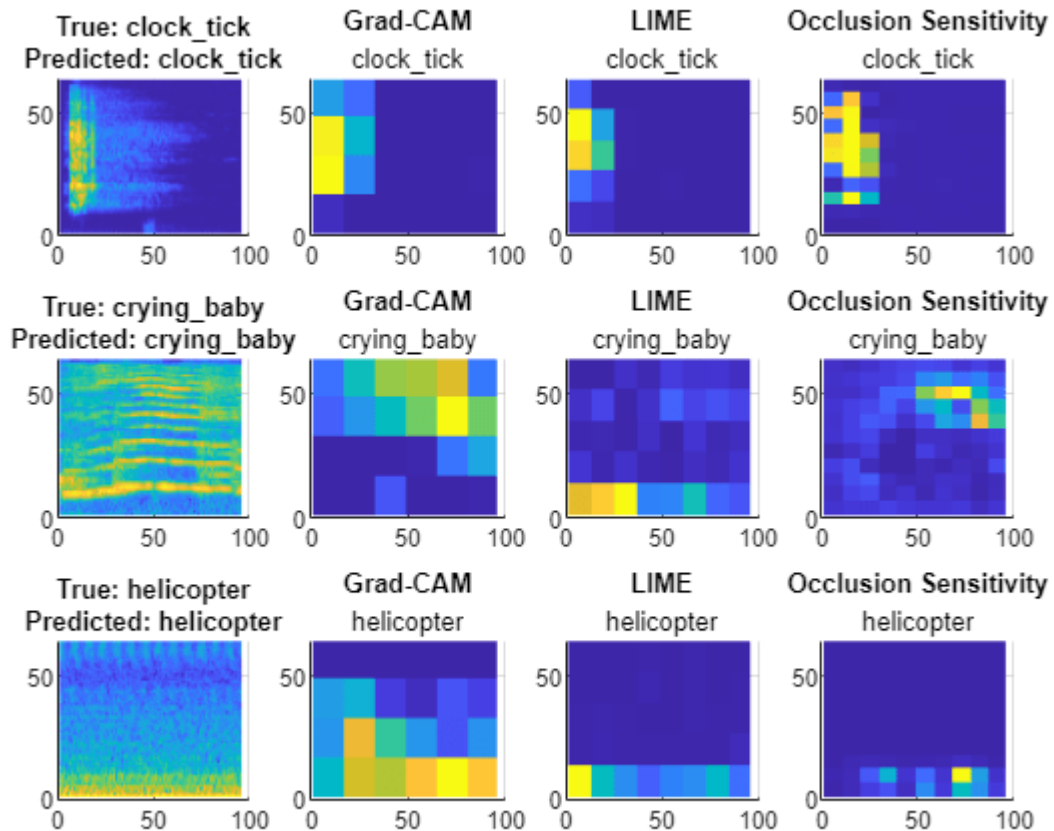
    mapGradCAM = gradCAM(net,img,mapClass, ...
        OutputUpsampling="nearest");

    mapLIME = imageLIME(net,img,mapClass, ...
        OutputUpsampling="nearest", ...
        Segmentation="grid");

    mapOcclusion = occlusionSensitivity(net,img,mapClass, ...
        OutputUpsampling="nearest");

    maps = {mapGradCAM,mapLIME,mapOcclusion};
    mapNames = ["Grad-CAM","LIME","Occlusion Sensitivity"];

    helperPlotMaps(img,YPred,YTrue,maps,mapNames,viewingAngle,mapClass)
end
```



The interpretability mappings highlight regions of interest for the predicted class label of each spectrogram.

- For the `clock_tick` class, all three methods focus on the same area of interest. The network uses the region corresponding to the ticking sound to make its prediction.
- For the `helicopter` class, all the three methods focus on the same region at the bottom of the spectrogram.
- For the `crying_baby` class, the three methods highlight different areas of the spectrogram, possibly because this spectrogram contains many small features. Methods like Grad-CAM, which produce lower resolution maps, might have difficulty picking out meaningful features. This example highlights the limits of using interpretability methods to understand individual network predictions.

As the results of training have an element of randomness, if you run this example again, you might see different results. Additionally, to produce interpretable output for different images, you might need to adjust the map parameters for the occlusion sensitivity and LIME maps. Grad-CAM does not require parameter tuning, but it can produce lower resolution maps than the other two methods.

Investigate Predictions for Specific Class

Investigate the interpretability maps for spectrograms from a particular class.

Find the spectrograms corresponding to the `helicopter` class.

```

classToInvestigate = ;
idxClass = find(classes == classToInvestigate);
idxSubset = validationLabels==classes(idxClass);

subsetLabels = validationLabels(idxSubset);
subsetImages = validationFeatures(:,:,,idxSubset);
subsetPredictions = validationPredictions(idxSubset);

imgIdx = [25 50 100];
numImages = length(imgIdx);

```

Generate and plot the interpretability maps using the input spectrograms and the predicted class labels.

```

viewingAngle = ;

figure
t3 = tiledlayout(numImages,4,"TileSpacing","compact");
for i = 1:numImages

    img = subsetImages(:,:,,imgIdx(i));
    YPred = subsetPredictions(imgIdx(i));
    YTrue = subsetLabels(imgIdx(i));

    mapClass = YPred;

    mapGradCAM = gradCAM(net,img,mapClass, ...
        OutputUpsampling="nearest");

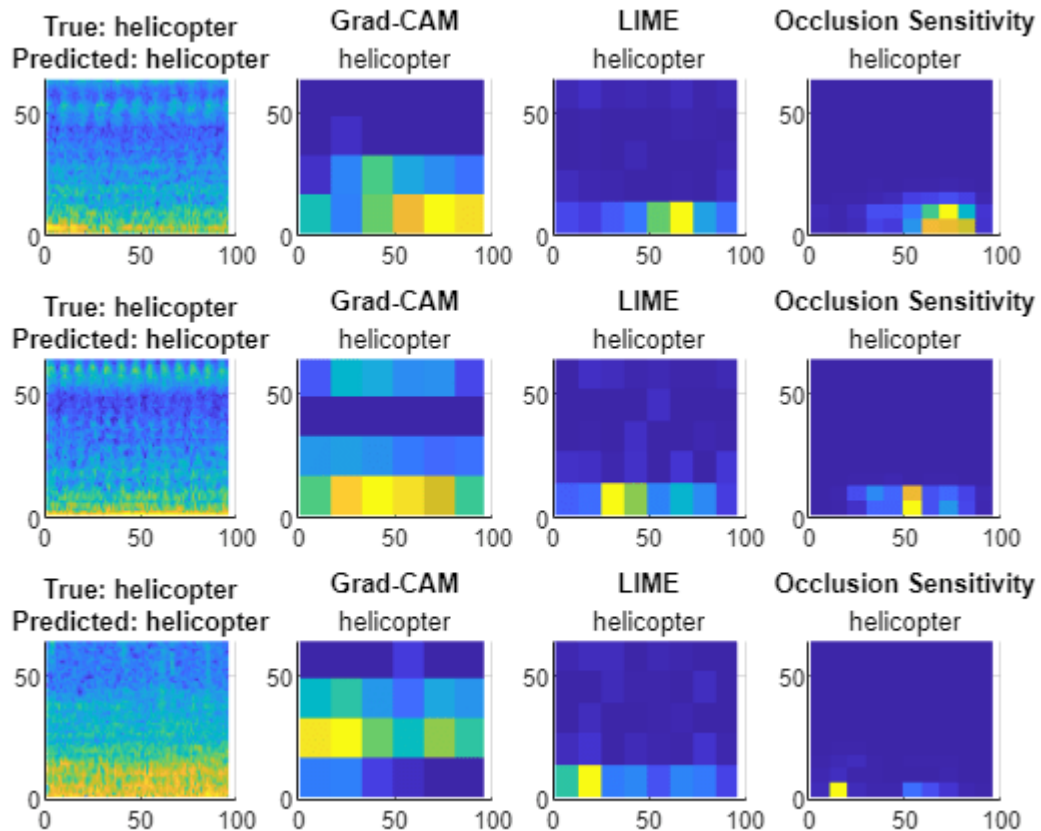
    mapLIME = imageLIME(net,img,mapClass, ...
        OutputUpsampling="nearest", ...
        Segmentation="grid");

    mapOcclusion = occlusionSensitivity(net,img,mapClass, ...
        OutputUpsampling="nearest");

    maps = {mapGradCAM,mapLIME,mapOcclusion};
    mapNames = ["Grad-CAM","LIME","Occlusion Sensitivity"];

    helperPlotMaps(img,YPred,YTrue,maps,mapNames,viewingAngle,mapClass)
end

```



The maps for each image show that the network is focusing on the area of high intensity and low frequency. The result is surprising as you might expect the network to also be interested in the high-frequency noise that repeats through time. Spotting patterns like this is important for understanding the features a network is using to make predictions.

Investigate Misclassifications

Use the interpretability maps to investigate misclassifications.

Investigate a spectrogram with the true class chainsaw but the predicted class helicopter.

```
trueClass = chainsaw ;
predictedClass = helicopter ;

incorrectIdx = find(validationPredictions == predictedClass & validationLabels' == trueClass);

idxToInvestigate = incorrectIdx(1);
YPred = validationPredictions(idxToInvestigate);
YTrue = validationLabels(idxToInvestigate);
```

Generate and plot the maps for both the true class (chainsaw) and the predicted class (helicopter).


```

figure
t4 = tiledlayout(2,4,"TileSpacing","compact");
img = validationFeatures(:,:,idxToInvestigate);

for mapClass = [YPred, YTrue]

    mapGradCAM = gradCAM(net,img,mapClass, ...
        OutputUpsampling="nearest");

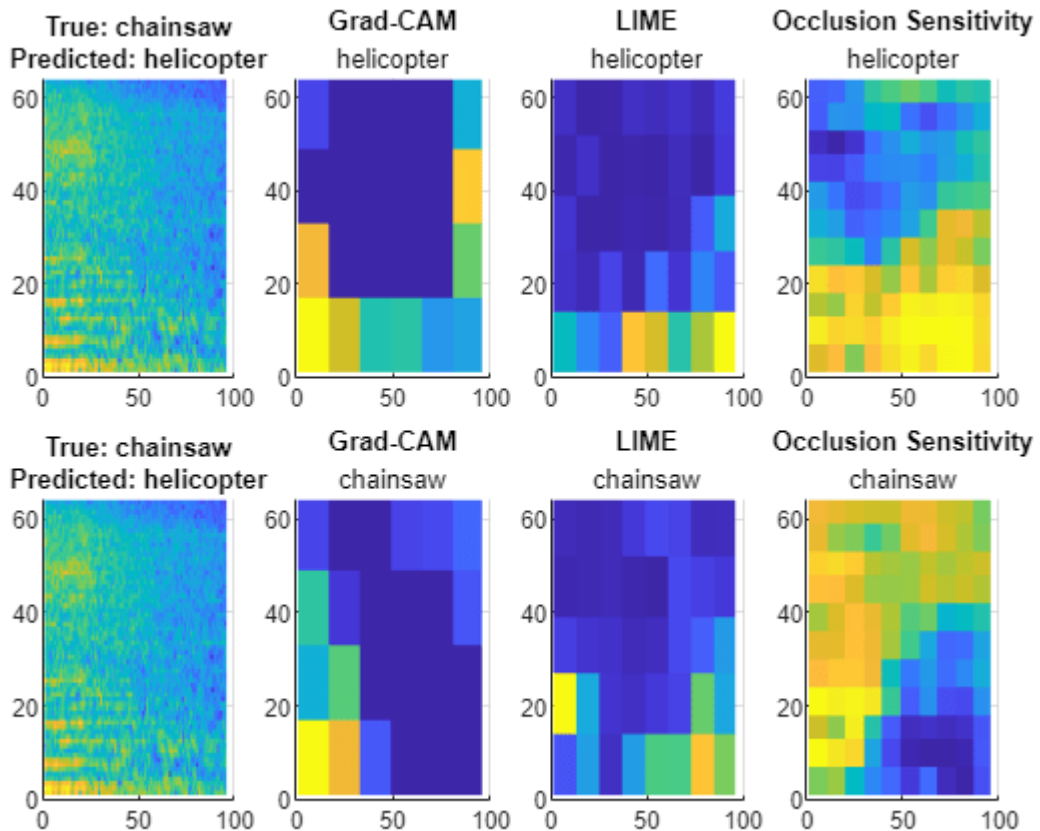
    mapLIME = imageLIME(net,img,mapClass, ...
        OutputUpsampling="nearest", ...
        Segmentation="grid");

    mapOcclusion = occlusionSensitivity(net,img,mapClass, ...
        OutputUpsampling="nearest");

    maps = {mapGradCAM,mapLIME,mapOcclusion};
    mapNames = ["Grad-CAM","LIME","Occlusion Sensitivity"];

    helperPlotMaps(img,YPred,YTrue,maps,mapNames,viewingAngle,mapClass)
end

```



The network focuses on the area of low frequency for the `helicopter` class. The result matches the interpretability maps generated for the `helicopter` class. Visual inspection is important for investigating what parts of an input the network is using to make its classification decisions.

Supporting Functions

`helperPlotMaps`

The supporting function `helperPlotMap` generates a plot of the input image and the specified interpretability maps.

```
function helperPlotMaps(img,YPred,YTrue,maps,mapNames,viewingAngle,mapClass)
nexttile
surf(img,EdgeColor="none")
view(viewingAngle)
title({"True: " + string(YTrue), "Predicted: " + string(YPred)}, ...
      interpreter="none")
colormap parula

numMaps = length(maps);
for i = 1:numMaps
    map = maps{i};
    mapName = mapNames(i);

    nexttile
    surf(map,EdgeColor="none")
    view(viewingAngle)
    title(mapName,mapClass,interpreter="none")
end
end
```

`helperAudioPreprocess`

The supporting function `helperAudioPreprocess` takes as input an `audioDatastore` object and the overlap percentage between log mel spectrograms and returns matrices of predictors and responses suitable for input to the VGGish network.

```
function [predictor,response,segmentsPerFile] = helperAudioPreprocess(ads,overlap)

numFiles = numel(ads.Files);

% Extract predictors and responses for each file
for ii = 1:numFiles
    [audioIn,info] = read(ads);

    fs = info.SampleRate;
    features = vggishPreprocess(audioIn,fs,OverlapPercentage=overlap);
    numSpectrograms = size(features,4);

    predictor{ii} = features;
    response{ii} = repelem(info.Label,numSpectrograms);
    segmentsPerFile(ii) = numSpectrograms;
end

% Concatenate predictors and responses into arrays
predictor = cat(4,predictor{:});
```

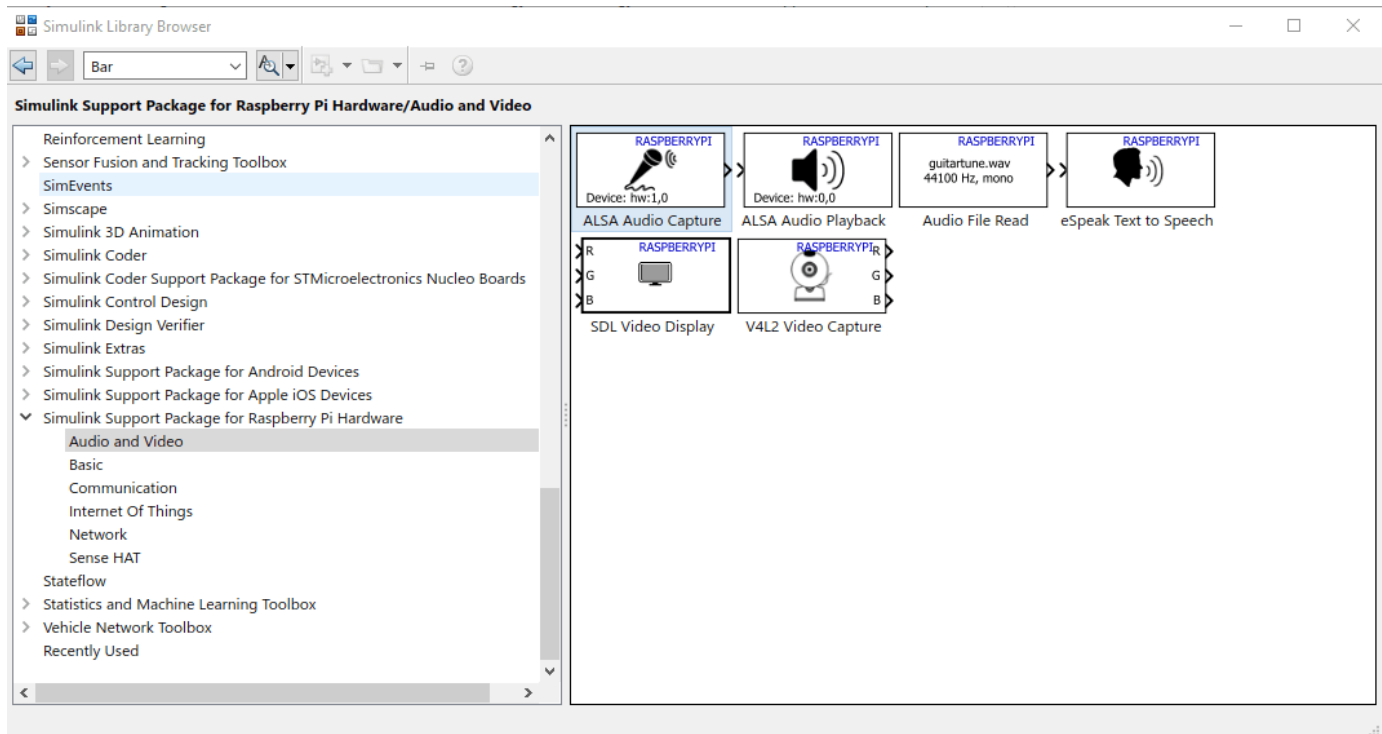
```
response = cat(2,response{:});  
end
```

Speech Command Recognition on Raspberry Pi Using Simulink

This example shows how to deploy feature extraction and a convolutional neural network (CNN) for speech command recognition on Raspberry Pi™. In this example you develop a simulink® model that captures audio from the microphone connected to the Raspberry Pi board and performs speech command recognition. You run the Simulink model on Raspberry Pi in External Mode and display the recognized speech command. For details about audio preprocessing and network training, see “Train Speech Command Recognition Model Using Deep Learning” on page 1-332.

Prepare Simulink Model

Create a Simulink model and capture the feature extraction, convolutional neural network and postprocessing as developed in “Speech Command Recognition in Simulink” on page 1-40. Add the ALSA Audio Capture (Simulink Support Package for Raspberry Pi Hardware) block from the **Simulink Support Package for Raspberry Pi Hardware** library as shown.



Connect a microphone to your Raspberry Pi board and use `listAudioDevices` (Simulink Support Package for Raspberry Pi Hardware) to list all the audio capture devices connected to your board.

```
r = raspi("raspiname", "pi", "password");
a = listAudioDevices(r, "capture");
a(1)
a(2)
```

```
ans =
```

```
struct with fields:
```

```

        Name: 'USB-Audio-LogitechUSBHeadsetH340-LogitechInc.LogitechUSBHeadsetH340atusb-0000:01:00.0-1.1, f
        Device: '2,0'
        Channels: {}
        BitDepth: {}
        SamplingRate: {}

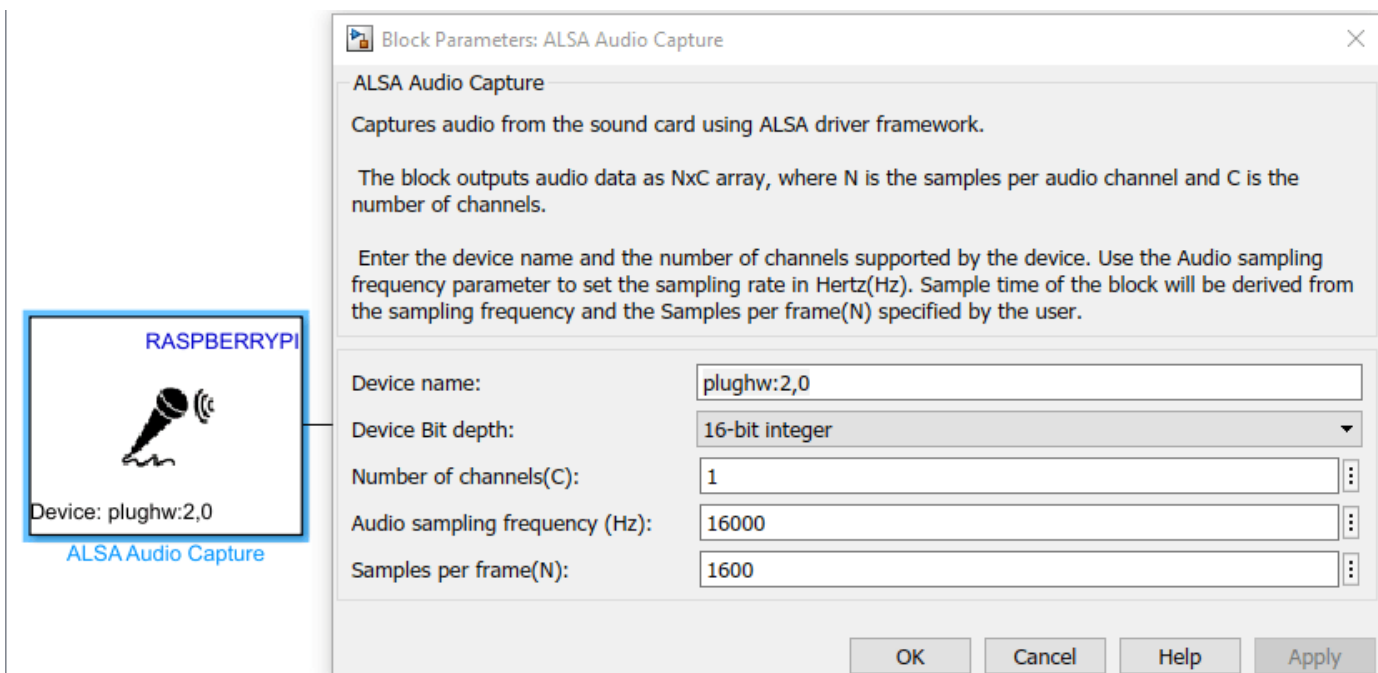
ans =

struct with fields:

        Name: 'USB-Audio-PlantronicsBT600-PlantronicsPlantronicsBT600atusb-0000:01:00.0-1.1, f
        Device: '3,0'
        Channels: {'1'}
        BitDepth: {'16-bit integer'}
        SamplingRate: {'16000'}

```

ALSA Audio Capture (Simulink Support Package for Raspberry Pi Hardware) block captures the audio signal from the default audio device on the Raspberry Pi hardware. You can also enter the name of an audio device such as `plughw:2,0` to capture audio from a device other than the default audio device. Double click on the ALSA Audio Capture (Simulink Support Package for Raspberry Pi Hardware) block and set **Device name** to `plughw:2,0`. Set the other parameters as shown.

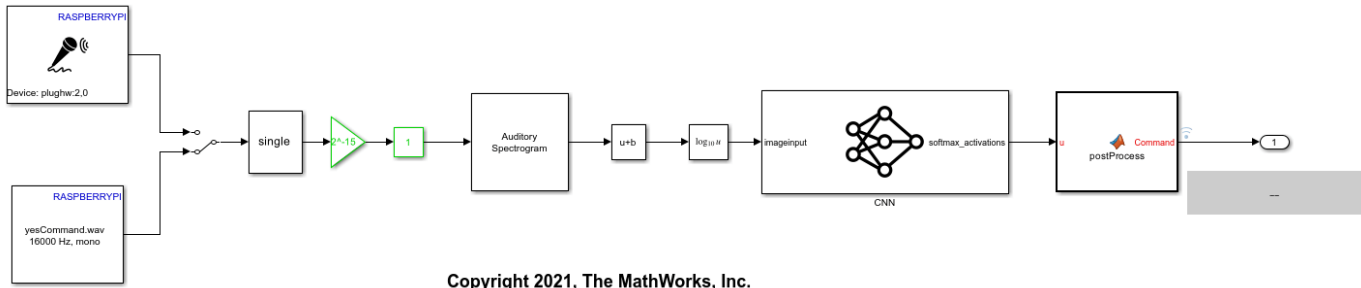


ALSA Audio Capture (Simulink Support Package for Raspberry Pi Hardware) outputs 16-bit fixed-point audio samples with values in the interval of $[-2^{15}, 2^{15} - 1]$. You cast the ALSA Audio Capture (Simulink Support Package for Raspberry Pi Hardware) output to single-precision data and multiply it by 2^{-15} to change the numerical range to $[-1, +1]$. Note that you are changing the numerical range because the subsequent blocks expect the audio in the range $[-1, +1]$. Use Audio File Read (Simulink Support Package for Raspberry Pi Hardware) block and a Manual Switch to switch the audio from the microphone to the audio file and back.

```

model = "slexSpeechCommandRecognitionRaspiExample";
open_system(model)

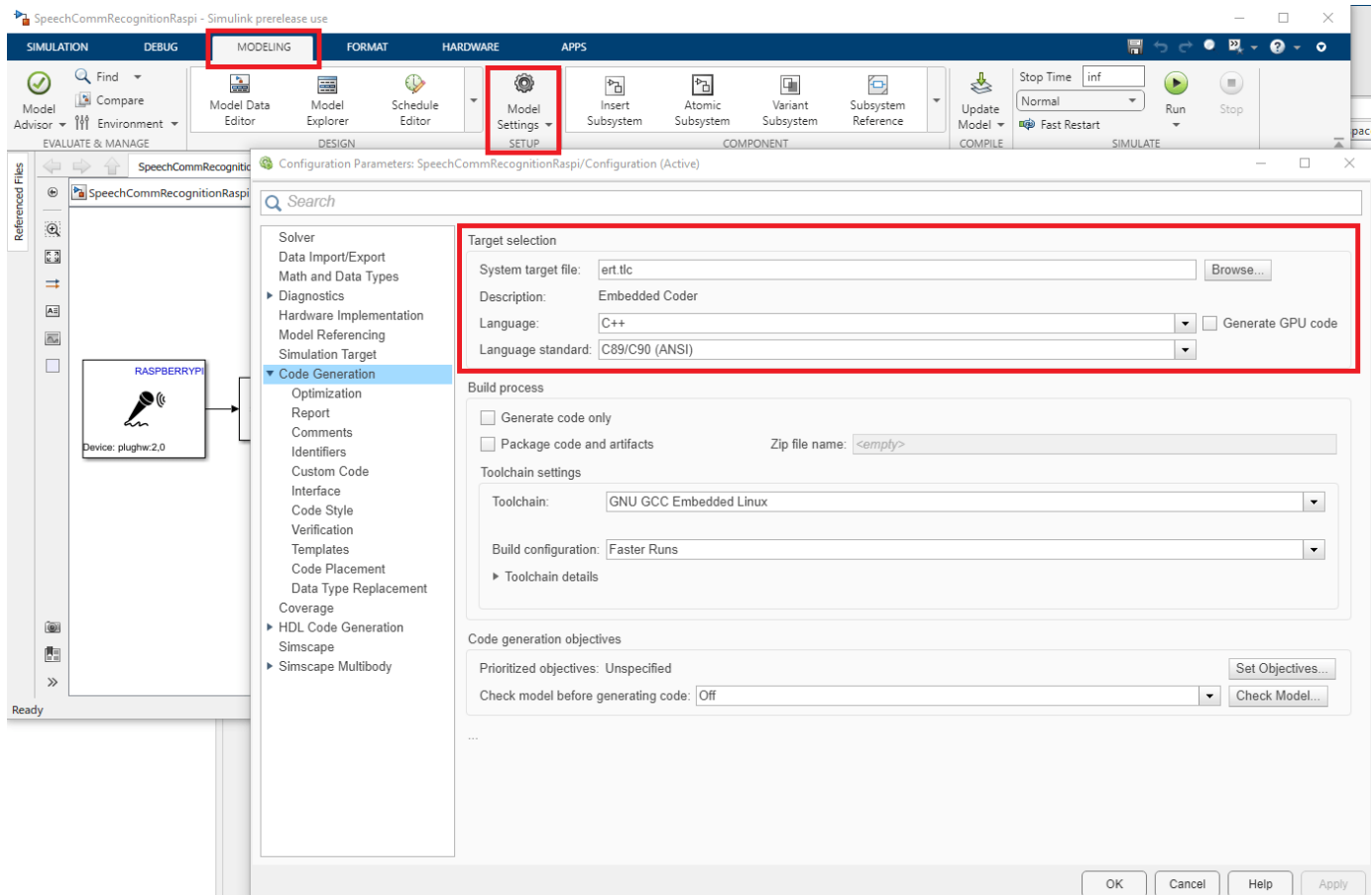
```



Copyright 2021, The MathWorks, Inc.

Configure Code Generation Settings

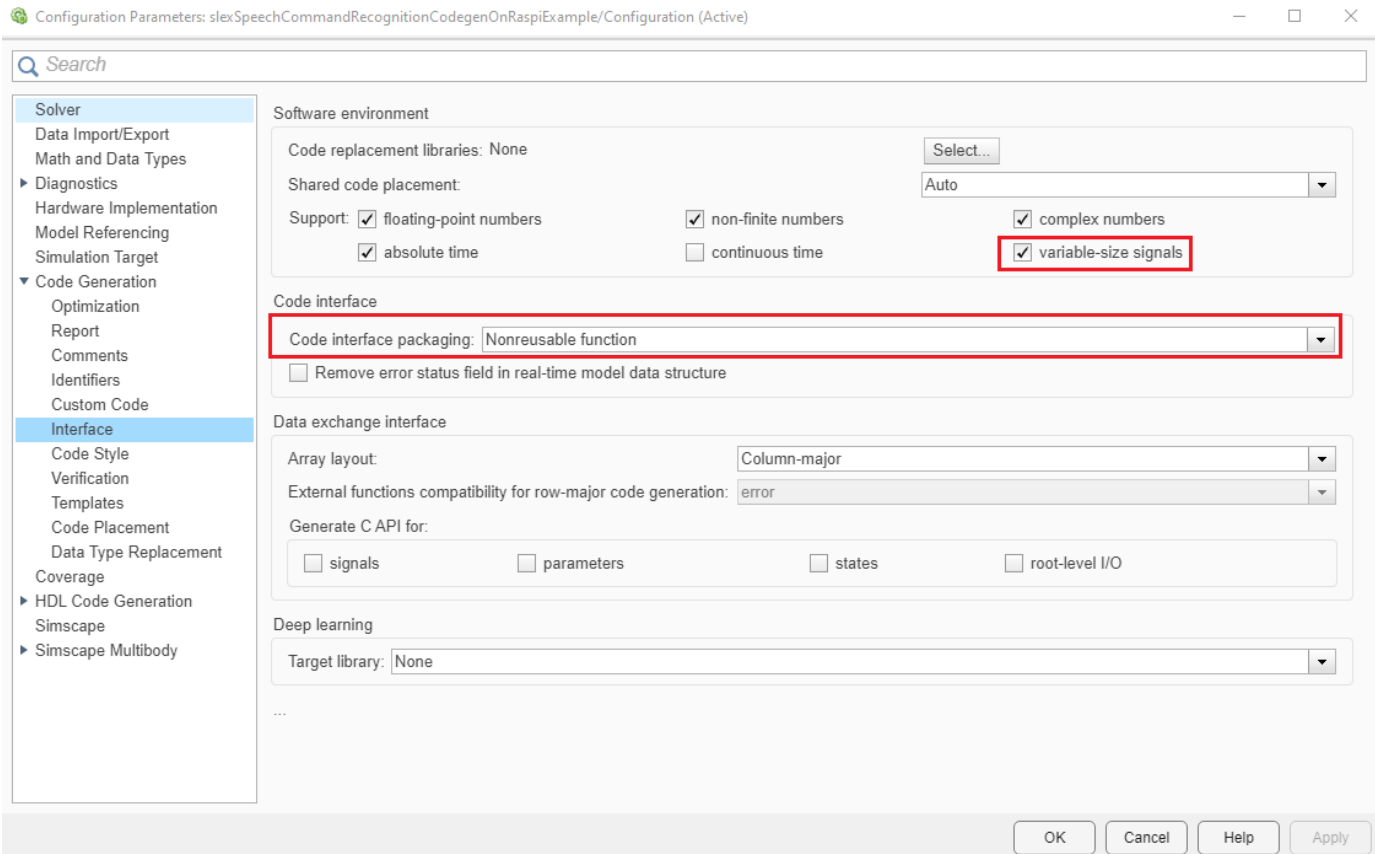
Open the `SpeechCommRecognitionRaspi` model, go to **MODELING** Tab and Click on **Model Settings** or press **Ctrl+E**. Select **Code Generation** and set the **System Target File** to `ert.tlc` whose **Description** is Embedded Coder. Set the **Language** to C++, which will automatically set the **Language Standard** to C++11 (ISO).



Alternatively, use `set_param` to configure the settings programmatically,

```
set_param(model, SystemTargetFile="ert.tlc")
set_param(model, TargetLang="C++")
set_param(model, TargetLangStandard="C++11 (ISO)")
```

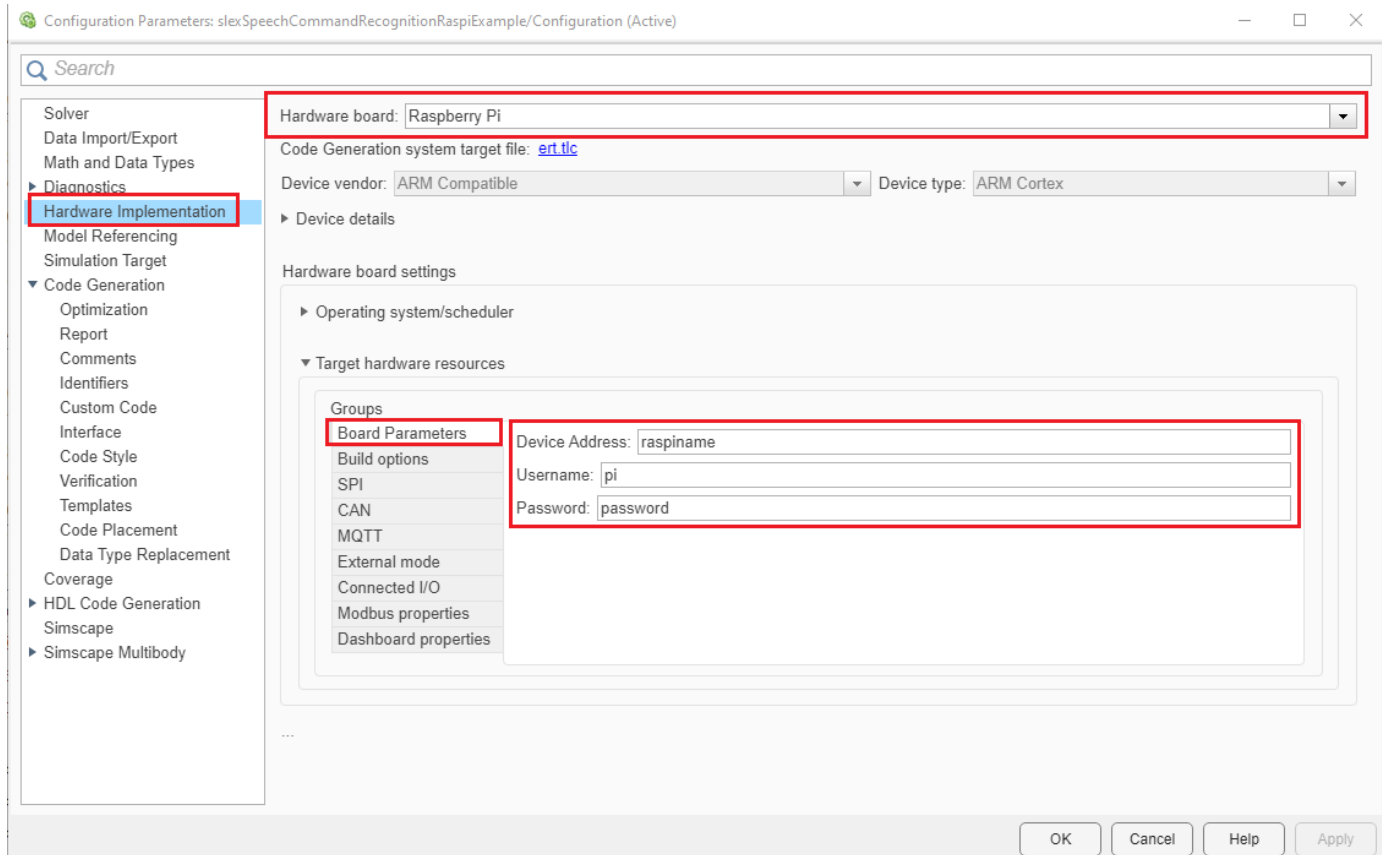
To run your model in External Mode, set **Code Interface packaging** to Nonreusable function and check **variable-size signals** in **Code Generation > Interface > Support** as shown.



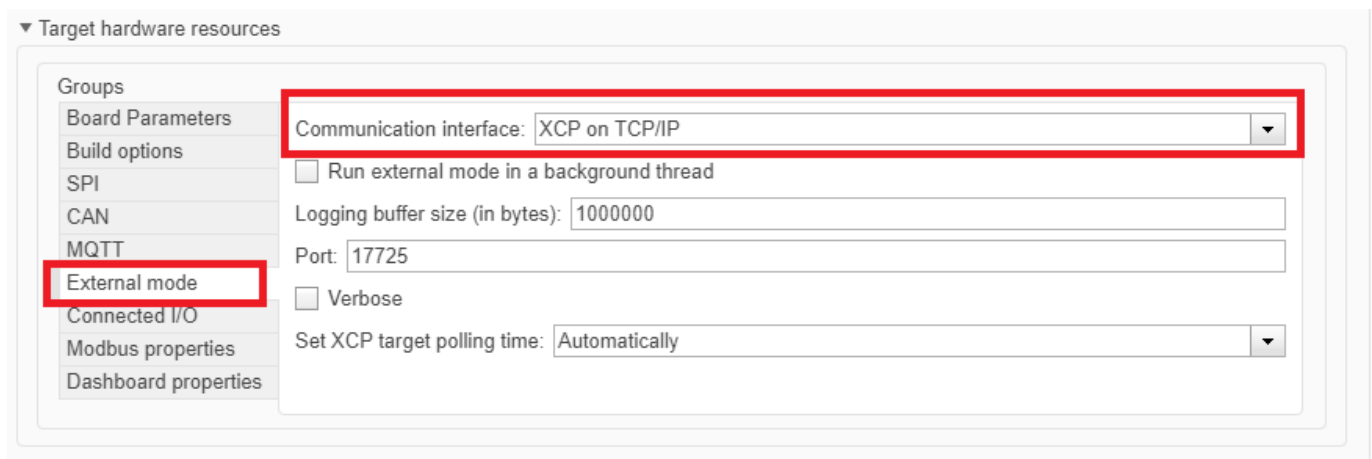
Select a solver that supports code generation. Set **Solver** to auto (Automatic solver selection) and **Solver type** to Fixed-step.

```
set_param(model,SolverName="FixedStepAuto")
set_param(model,SolverType="Fixed-step")
```

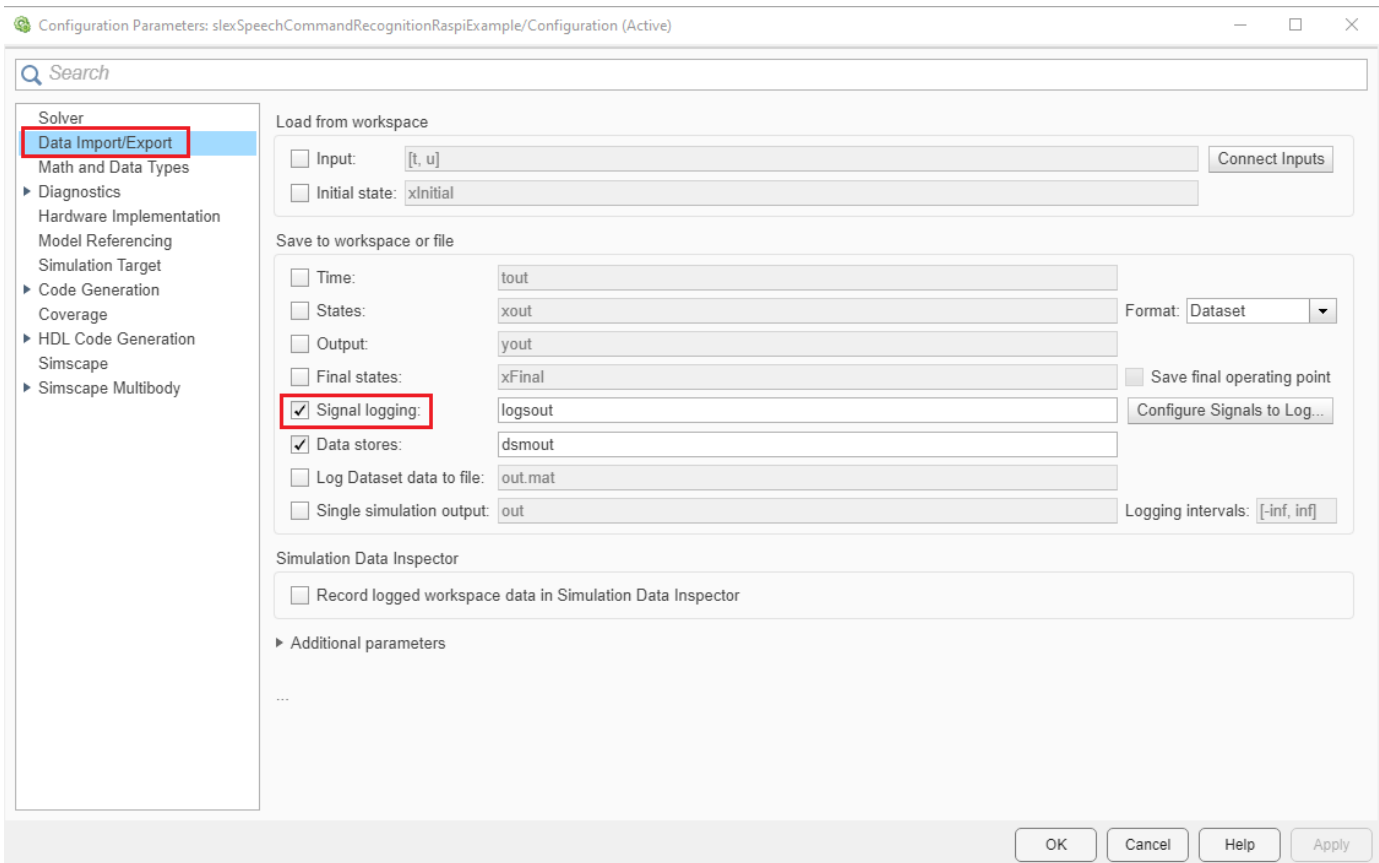
In **Configuration > Hardware Implementation**, set **Hardware board** to Raspberry Pi and enter your Raspberry Pi credentials in the **Board Parameters** as shown.



In the same window, set **External mode > Communication interface** to XCP on TCP/IP as shown.

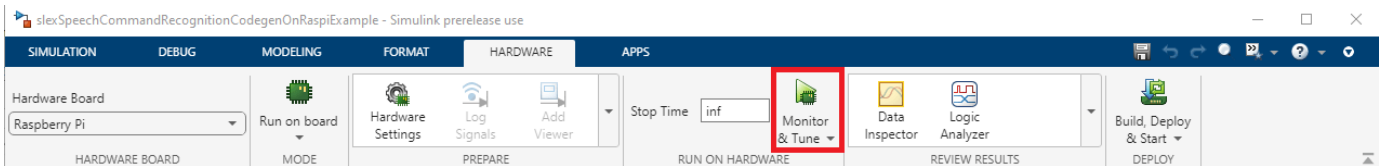


Check **Signal logging** in **Data Import/Export** to enable signal monitoring in External Mode.



Deploy the Model on Raspberry Pi and Perform Speech Command Recognition

Go to **Hardware** tab and click on **Monitor & Tune** as shown.



Now close the model.

```
save_system(model);
close_system(model);
```

```
Warning: Unable to resolve the name
'CloneDetector.ExclusionEditorUIService.getInstance'.
```

Other Things To Try

- Simulate “Speech Command Recognition Code Generation with Intel MKL-DNN Using Simulink” on page 1-881 Example in Processor-in-the-loop (PIL) mode on Raspberry Pi.

- Use LED (Simulink Support Package for Raspberry Pi Hardware) block of Simulink Support Package for Raspberry Pi Hardware and light it up for the Go speech command. Use **Deploy** pane in **Hardware** tab to deploy the standalone application on Raspberry Pi.

Speech Command Recognition Using Deep Learning

This example shows how to perform speech command recognition on streaming audio. The example uses a pretrained deep learning model. To learn how the deep learning model was trained, see “Train Speech Command Recognition Model Using Deep Learning” on page 1-332.

Load the pre-trained network. The network is trained to recognize the following speech commands: *yes*, *no*, *up*, *down*, *left*, *right*, *on*, *off*, *stop*, and *go*, and to otherwise classify audio as an unknown word or as background noise.

```
load("commandNet.mat")
labels = trainedNet.Layers(end).Classes'

labels = 1×12 categorical
      down      go      left      no      off      on      right      stop      up      yes
```

Load one of the following audio signals: noise, someone saying *stop*, or someone saying *play*. The word *stop* is recognized by the network as a command. The word *play* is an unknown word to the network. Listen to the signal.

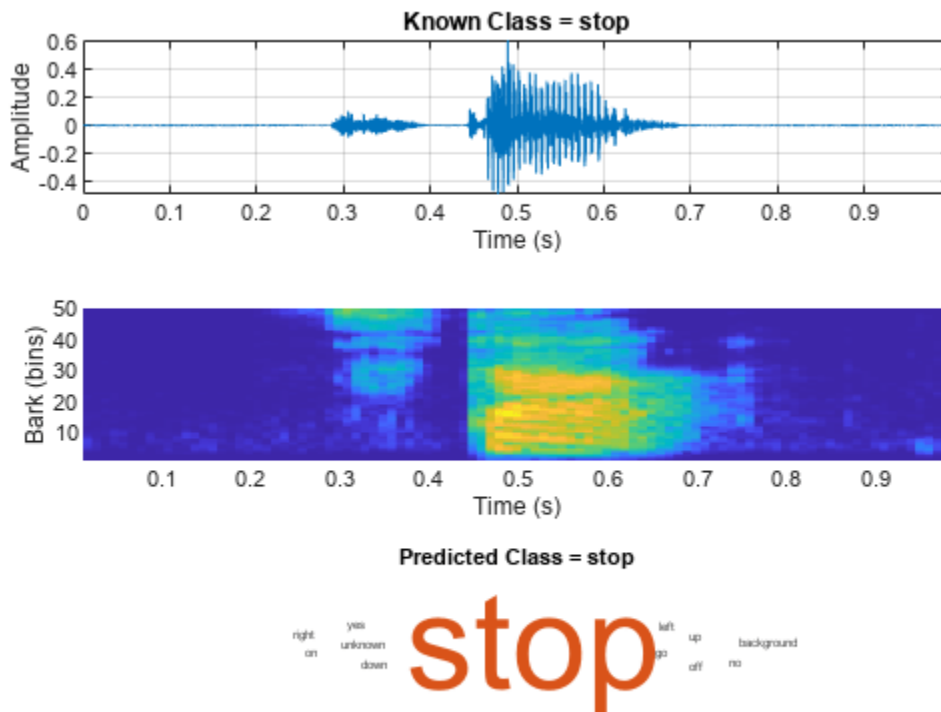
```
audioData = ;
sound(audioData{1},audioData{2})
```

The pre-trained network takes auditory-based spectrograms as inputs. Use the supporting function `extractAuditorySpectrogram` on page 1-933 to extract the spectrogram. Classify the audio based on its auditory spectrogram.

```
auditorySpectrogram = extractAuditorySpectrogram(audioData{1},audioData{2});
prediction = classify(trainedNet,auditorySpectrogram);
```

Use the supporting function `visualizeClassificationPipeline` on page 1-934, to plot the audio signal, the auditory spectrogram, the network prediction, and a word cloud indicating the prediction scores.

```
visualizeClassificationPipeline(audioData,trainedNet)
```



Detect Commands from Streaming Audio

The model was trained to classify auditory spectrograms that correspond to 1-second chunks of audio data. It has no concept of memory between classifications. To adapt this model for streaming applications, you can add logic to build up decision confidence over time.

Create a 9-second long audio clip of the background noise, the unknown word, and the known command.

```
fs = 16e3;
audioPlay = audioread("playCommand.flac");
audioStop = audioread("stopCommand.flac");
audioBackground = 0.02*pinknoise(fs);
audioIn = repmat([audioBackground;audioPlay;audioStop],3,1);
```

Specify the classification rate in hertz. The classification rate is the number of classifications per second. Every classification requires 1 second of audio data.

```
classificationRate = 20  ; % Hz
```

Specify the time window for decisions. Decisions are made by considering all individual classifications in a decision time window.

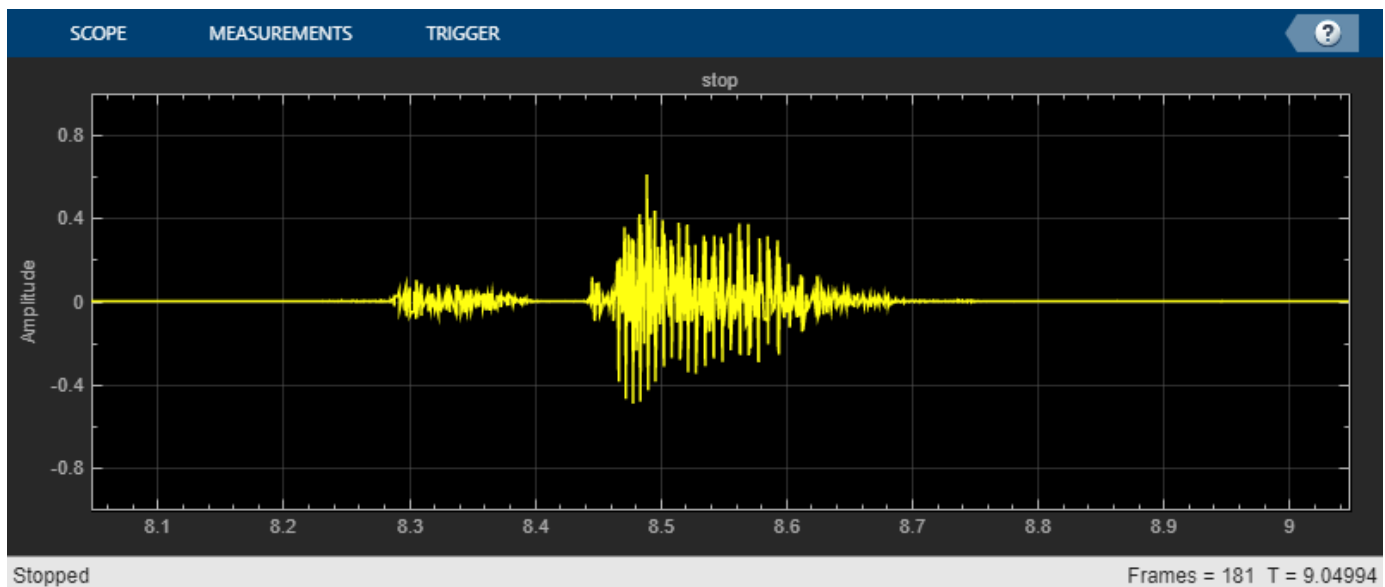
```
decisionTimeWindow = 1.5  ; % seconds
```

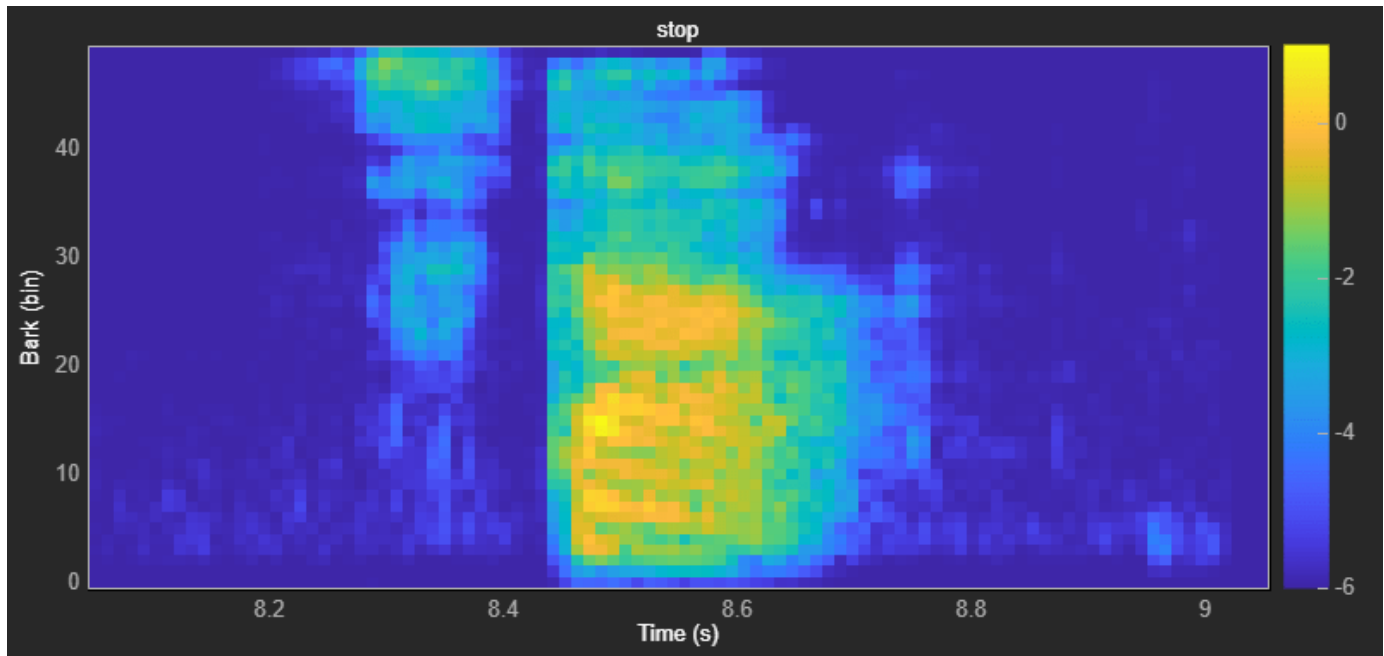
Specify thresholds for the decision logic. The `frameAgreementThreshold` is the percent of frames within a `decisionTimeWindow` that must agree to recognize a word. The `probabilityThreshold` is the threshold that at least one of the classification probabilities in the `decisionTimeWindow` must pass.

```
frameAgreementThreshold = 50  ; % percent
probabilityThreshold = 0.7  ;
```

Use the supporting function, `detectCommands` on page 1-936, to simulate streaming command detection. The function uses your default audio device to play the streaming audio.

```
detectCommands( ...
    Input=audioIn, ...
    SampleRate=fs, ...
    Network=trainedNet, ...
    ClassificationRate=classificationRate, ...
    DecisionTimeWindow=decisionTimeWindow, ...
    FrameAgreementThreshold=frameAgreementThreshold, ...
    ProbabilityThreshold=probabilityThreshold);
```



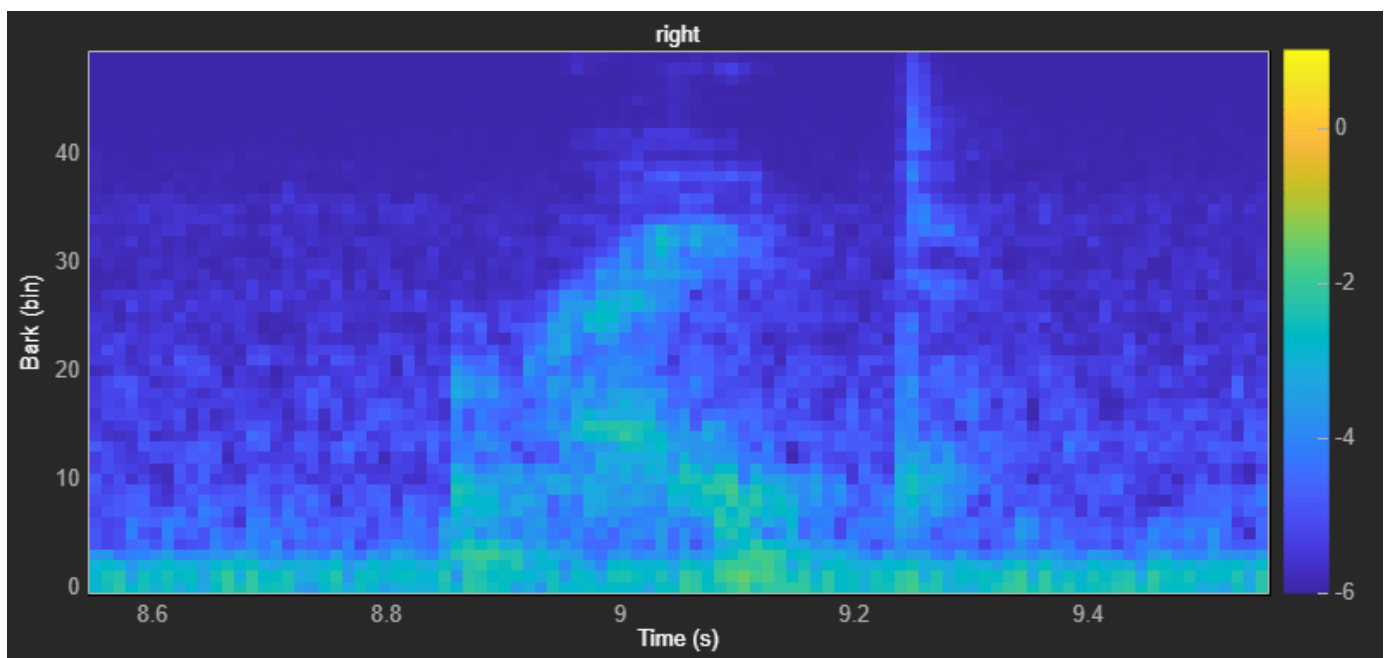
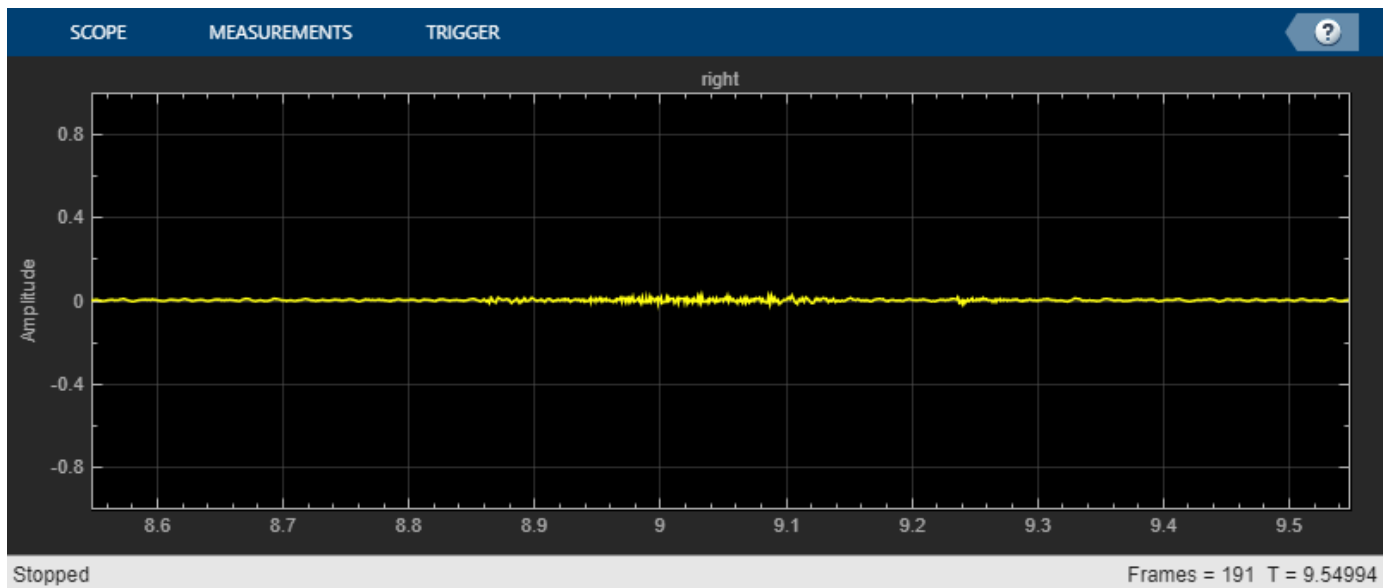


Detect Commands from Microphone Input

You can test the model by performing speech command recognition on data input from your microphone. In this case, audio is read from your default audio input device. The `TimeLimit` parameter controls the duration of the audio recording. You can end the recording early by closing the scopes.

The network is trained to recognize the following speech commands: *yes*, *no*, *up*, *down*, *left*, *right*, *on*, *off*, *stop*, and *go*, and to otherwise classify audio as an unknown word or as background noise.

```
detectCommands( ...
    SampleRate=fs, ...
    Network=trainedNet, ...
    ClassificationRate= 20 _____, ...
    DecisionTimeWindow= 1.5 _____, ...
    FrameAgreementThreshold= 50 _____, ...
    ProbabilityThreshold= 0.7 _____, ...
    TimeLimit= 10 _____ );
```



Supporting Functions

Extract Auditory Spectrogram

```
function features = extractAuditorySpectrogram(x,fs)
%extractAuditorySpectrogram Compute auditory spectrogram
%
% features = extractAuditorySpectrogram(x,fs) computes an auditory (Bark)
% spectrogram in the same way as done in the Train Speech Command
% Recognition Model Using Deep Learning example. Specify the audio input,
% x, as a mono audio signal with a 1 second duration.
```

```
% Design audioFeatureExtractor object
persistent afe segmentSamples
if isempty(afe)
    designFs = 16e3;
    segmentDuration = 1;
    frameDuration = 0.025;
    hopDuration = 0.01;

    numBands = 50;
    FFTLength = 512;

    segmentSamples = round(segmentDuration*designFs);
    frameSamples = round(frameDuration*designFs);
    hopSamples = round(hopDuration*designFs);
    overlapSamples = frameSamples - hopSamples;

    afe = audioFeatureExtractor( ...
        SampleRate=designFs, ...
        FFTLength=FFTLength, ...
        Window=hann(frameSamples,"periodic"), ...
        OverlapLength=overlapSamples, ...
        barkSpectrum=true);
    setExtractorParams(afe,"barkSpectrum",NumBands=numBands,WindowNormalization=false);
end

% Resample to 16 kHz if necessary
if double(fs)~=16e3
    x = cast(resample(double(x),16e3,double(fs)),like=x);
end

% Ensure the input is equal to 1 second of data at 16 kHz.
x = trimOrPad(x,segmentSamples);

% Extract features
features = extract(afe,x);

% Apply logarithm
features = log10(features + 1e-6);

end
```

Visualize Classification Pipeline

```
function visualizeClassificationPipeline(audioData,trainedNet)
%visualizeClassificationPipeline Visualize classification pipeline
%
% visualizeClassificationPipeline(audioData,trainedNet) creates a tiled
% layout of the audio data, the extracted auditory spectrogram, and a word
% cloud indicating the relative prediction probability of each class.

% Unpack audio data
audio = audioData{1};
fs = audioData{2};
label = audioData{3};
```



```

% Create tiled layout
tiledlayout(3,1)

% Plot audio signal in first tile
nexttile
plotAudio(audio,fs)
title("Known Class = "+label)

% Plot auditory spectrogram in second tile
nexttile
auditorySpectrogram = extractAuditorySpectrogram(audio,fs);
plotAuditorySpectrogram(auditorySpectrogram)

% Plot network predictions as word cloud in third tile
nexttile
[prediction,scores] = classify(trainedNet,auditorySpectrogram);
wordcloud(trainedNet.Layers(end).Classes,scores)
title("Predicted Class = "+string(prediction))

function plotAuditorySpectrogram(auditorySpectrogram)
    %plotAuditorySpectrogram Plot auditory spectrogram

    % extratAuditorySpectrogram uses 25 ms windows with 10 ms hops.
    % Create a time vector with instants corresponding to the center of
    % the windows
    t = 0.0125:0.01:(1-0.0125);

    bins = 1:size(auditorySpectrogram,2);

    pcolor(t,bins,auditorySpectrogram')
    shading flat
    xlabel("Time (s)")
    ylabel("Bark (bins)")

end
function plotAudio(audioIn,fs)
    %plotAudio Plot audio

    t = (0:size(audioIn,1)-1)/fs;
    plot(t,audioIn)
    xlabel("Time (s)")
    ylabel("Amplitude")
    grid on
    axis tight

end
end

```

Trim or Pad

```

function y = trimOrPad(x,n)
%trimOrPad Trim or pad audio
%
% y = trimOrPad(x,n) trims or pads the input x to n samples along the first
% dimension. 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.

```

```
a = size(x,1);
if a < n
    frontPad = floor((n-a)/2);
    backPad = n - a - frontPad;
    y = [zeros(frontPad,size(x,2),like=x);x;zeros(backPad,size(x,2),like=x)];
elseif a > n
    frontTrim = floor((a-n)/2) + 1;
    backTrim = a - n - frontTrim;
    y = x(frontTrim:end-backTrim,:);
else
    y = x;
end

end
```

Plot Streaming Features

```
function detectCommands(nvars)
%detectCommand Detect commands
%
% detectCommand(SampleRate=fs,Network=net,ClassificationRate=cr, ...
%   DecisionTimeWindow=dtw,FrameAgreementThreshold=fat,ProbabilityThreshold=pt, ...
%   Input=audioIn)
% opens a timescope to visualize streaming audio and a dsp.MatrixViewer to
% visualize auditory spectrograms extracted from a simulation of streaming
% audioIn. The scopes display the detected speech command after it has been
% processed by the streaming algorithm. The streaming audio is played to
% your default audio output device.
%
% detectCommand(SampleRate=fs,Network=net,ClassificationRate=cr, ...
%   DecisionTimeWindow=dtw,FrameAgreementThreshold=fat,ProbabilityThreshold=pt, ...
%   TimeLimit=tl)
% opens a timescope to visualize streaming audio and a dsp.MatrixViewer to
% visualize auditory spectrograms extracted from audio streaming from your
% default audio input device. The scopes display the detected speech
% command after it has been processed by the streaming algorithm.

arguments
    nvars.SampleRate
    nvars.Network
    nvars.ClassificationRate
    nvars.DecisionTimeWindow
    nvars.FrameAgreementThreshold
    nvars.ProbabilityThreshold
    nvars.Input = []
    nvars.TimeLimit = inf;
end

% Isolate the labels
labels = nvars.Network.Layers(end).Classes;

if isempty(nvars.Input)
    % Create an audioDeviceReader to read audio from your microphone.
    adr = audioDeviceReader(SampleRate=nvars.SampleRate,SamplesPerFrame=floor(nvars.SampleRate,
```

```

    % Create a dsp.AsyncBuffer to buffer the audio streaming from your
    % microphone into overlapping segments.
    audioBuffer = dsp.AsyncBuffer(nvars.SampleRate);
else
    % Create a dsp.AsyncBuffer object. Write the audio to the buffer so that
    % you can read from it in a streaming fashion.
    audioBuffer = dsp.AsyncBuffer(size(nvars.Input,1));
    write(audioBuffer,nvars.Input);

    % Create an audioDeviceWriter object to write the audio to your default
    % speakers in a streaming loop.
    adw = audioDeviceWriter(SampleRate=nvars.SampleRate);
end

newSamplesPerUpdate = floor(nvars.SampleRate/nvars.ClassificationRate);

% Convert the requested decision time window to the number of analysis frames.
numAnalysisFrame = round((nvars.DecisionTimeWindow-1)*(nvars.ClassificationRate) + 1);

% Convert the requested frame agreement threshold in percent to the number of frames that must agree.
countThreshold = round(nvars.FrameAgreementThreshold/100*numAnalysisFrame);

% Initialize buffers for the classification decisions and probabilities of the streaming audio.
YBuffer = repmat(categorical("background"),numAnalysisFrame,1);
probBuffer = zeros(numel(labels),numAnalysisFrame,"single");

% Create a timescope object to visualize the audio processed in the
% streaming loop. Create a dsp.MatrixViewer object to visualize the
% auditory spectrogram used to make predictions.
wavePlotter = timescope( ...
    SampleRate=nvars.SampleRate, ...
    Title="...", ...
    TimeSpanSource="property", ...
    TimeSpan=1, ...
    YLimits=[-1,1], ...
    Position=[600,640,800,340], ...
    TimeAxisLabels="none", ...
    AxesScaling="manual");
show(wavePlotter)
specPlotter = dsp.MatrixViewer( ...
    XDataMode="Custom", ...
    AxisOrigin="Lower left corner", ...
    Position=[600,220,800,380], ...
    ShowGrid=false, ...
    Title="...", ...
    XLabel="Time (s)", ...
    YLabel="Bark (bin)");
show(specPlotter)

% Initialize variables for plotting
currentTime = 0;
colorLimits = [-1,1];

% Run the streaming loop.
loopTimer = tic;
while whileCriteria(loopTimer,nvars.TimeLimit,wavePlotter,specPlotter,nvars.Input,audioBuffer)

    if isempty(nvars.Input)

```

```

        % Extract audio samples from the audio device and add the samples to
        % the buffer.
        audioIn = adr();
        write(audioBuffer,audioIn);
    end

    % Read samples from the buffer
    y = read(audioBuffer,nvars.SampleRate,nvars.SampleRate - newSamplesPerUpdate);

    % Extract an auditory spectrogram from the audio
    spec = extractAuditorySpectrogram(y,nvars.SampleRate);

    % Classify the current spectrogram, save the label to the label buffer,
    % and save the predicted probabilities to the probability buffer.
    [YPredicted,probs] = classify(nvars.Network,spec);
    YBuffer = [YBuffer(2:end);YPredicted];
    probBuffer = [probBuffer(:,2:end),probs(:)];

    % Plot the current waveform and spectrogram.
    ynew = y(end-newSamplesPerUpdate+1:end);
    wavePlotter(ynew)
    specPlotter(spec')

    % Declare a detection and display it in the figure if 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,:));
    if YMode == "background" || count < countThreshold || maxProb < nvars.ProbabilityThreshold
        wavePlotter.Title = "...";
        specPlotter.Title = "...";
    else
        wavePlotter.Title = string(YMode);
        specPlotter.Title = string(YMode);
    end
end

% Update variables for plotting
currentTime = currentTime + newSamplesPerUpdate/nvars.SampleRate;
colorLimits = [min([colorLimits(1),min(spec,[],"all")]),max([colorLimits(2),max(spec,[],"all")])];
specPlotter.CustomXData = [currentTime-1,currentTime];
specPlotter.ColorLimits = colorLimits;

if ~isempty(nvars.Input)
    % Write the new audio to your audio output device.
    adw(ynew);
end
end
release(wavePlotter)
release(specPlotter)

function tf = whileCriteria(loopTimer,timeLimit,wavePlotter,specPlotter,Input,audioBuffer)
    if isempty(Input)
        tf = toc(loopTimer)<timeLimit && isVisible(wavePlotter) && isVisible(specPlotter);
    else
        tf = audioBuffer.NumUnreadSamples > 0;
    end
end

```

end
end

See Also

Related Examples

- “Train Speech Command Recognition Model Using Deep Learning” on page 1-332

Keyword Spotting in Simulink

This example shows a Simulink® model that identifies a keyword in speech using a pretrained deep learning model. This model was trained to identify the keyword "yes". To learn about the model architecture and training, see “Keyword Spotting in Noise Using MFCC and LSTM Networks” on page 1-501.

Download Pretrained Keyword Spotting Network

Download and unzip the pretrained network and the standardization factors. The standardization factors are the global mean and standard deviation of the features used to train the model.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","KeywordSpotting.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
netFolder = fullfile(dataFolder,"KeywordSpotting");
addpath(netFolder)
```

Model Description

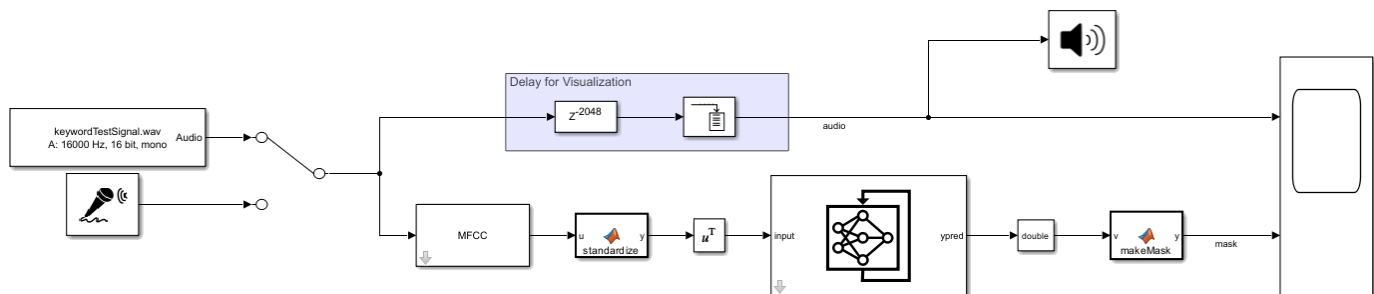
The deep learning network was trained on mel-frequency cepstral coefficients (MFCC) computed using an `audioFeatureExtractor`. The MFCC block in the model has been configured to extract the same features that the network was trained on.

The MFCC block extracts feature vectors from the audio stream using 512-point analysis windows with 384-point overlap and then applies a buffer to output 16 feature vectors consisting of 39 features each. Buffering the feature vectors enables vectorized computations on the `Stateful Classify` block, which enables the system to keep pace with real time (given a short time delay).

After the MFCC block, the features are standardized using precomputed coefficients and then transposed so that time is along the second dimension.

The `Stateful Classify` block outputs a binary decision for each feature vector. The decisions are converted to doubles and then upsampled to create a decision mask the same length as the corresponding audio.

```
open_system("keywordSpotting.slx")
```

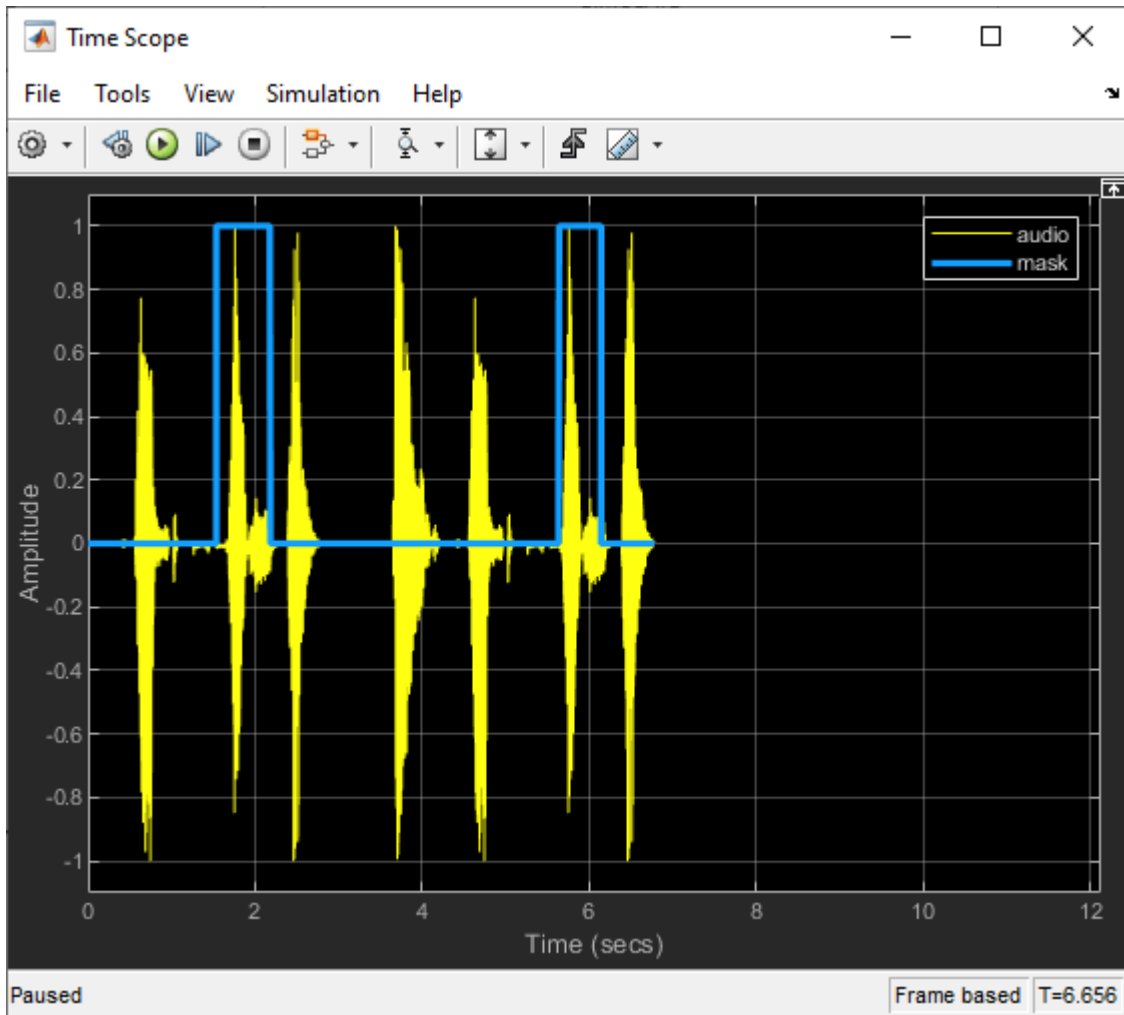


Copyright 2022 The MathWorks, Inc.

Run Model

Use the `Manual Switch` block to select either a live stream from your microphone or a test signal from an audio file.

```
sim("keywordSpotting.slx");
```



Close the model and remove the path to the pretrained network.

```
close_system("keywordSpotting.slx",0);  
rmpath(netFolder)
```

Audio Transfer Learning Using Experiment Manager

This example shows how to configure an experiment that compares the performance of multiple pretrained networks when applied to a speech command recognition task using transfer learning. It highlights Experiment Manager (Deep Learning Toolbox)'s capability to tune hyperparameters and easily compare results between the different pretrained networks using both built-in and user-defined metrics.

Audio Toolbox™ provides a variety of pretrained networks for audio processing, and each consists of a different architecture that requires different data pre-processing. These differences result in tradeoffs between the accuracy, speed, and size of the various networks. Experiment Manager organizes the results of training experiments to highlight the strengths and weaknesses of each individual network so you can select the network that best fits your constraints.

The example compares the performance of the YAMNet and VGGish pretrained networks, as well as a custom-designed network that is trained from scratch. See Deep Network Designer (Deep Learning Toolbox) to explore other pretrained network options supported by Audio Toolbox™.

In this example you will download the Google Speech Commands Dataset [1] and the pretrained networks and store them in your temp directory if they are not already present. The dataset takes up 1.96 GB of disk space and the networks in total take up 470 MB.

Open Experiment Manager

Load the example by clicking the **Open Example** button. This opens the project in Experiment Manager in your MATLAB editor.

The screenshot shows the Experiment Manager interface with the following configuration for the 'CompareNetworks' experiment:

- Mode:** Sequential
- Cluster:** (empty)
- Pool Size:** 0
- Run:** (Run button)

Description:

Compare different networks for speech command recognition:
 * Custom network (see the Speech Command Recognition Using Deep Learning example for details)
 * YAMNet transfer learning
 * VGGish transfer learning

Hyperparameters:

Strategy: Exhaustive Sweep

In the setup and metric functions, access hyperparameter values by using dot notation.

Name	Values
Network	["vggish", "yamnet", "custom"]

Buttons: + Add, Delete

Setup Function:

compareNetSetup

Buttons: + New, Edit

Metrics:

Standard training and validation metrics (such as accuracy, RMSE, and loss) are computed by default.

Custom Metrics
sizeMB
numLearnableParams
numIters

Built-in training experiments consist of a description, a table of hyperparameters, a setup function, and a collection of metric functions to evaluate the results of the experiment. For more information, see “Configure Built-In Training Experiment” (Deep Learning Toolbox).

The **Description** field contains a textual description of the experiment.

The **Hyperparameters** section specifies the strategy (Exhaustive Sweep) and hyperparameter values to use for the experiment. When you run the experiment, Experiment Manager trains the network using every combination of hyperparameter values specified in the hyperparameter table. This example demonstrates how to test the different network types. Define one hyperparameter, **Network**, to represent the network names stored as strings.

The **Setup Function** field contains the name of the main function that configures the training data, network architecture, and training options for the experiment. The input to the setup function is a structure with fields from the hyperparameter table. The setup function returns the training data, network architecture, and training parameters as outputs. This has already been implemented for you.

The **Metrics** list enables you to define your own custom metrics to compare across different trials of the training experiment. A couple of example custom metric functions are defined for you later in this example. Experiment Manager runs each of the listed metrics against the networks trained in each trial. The metrics defined for you in this example are listed here. Any additional custom metric you intend to use must be listed in this section.

Define Setup Function

In this example, the **Setup Function** downloads the dataset, selects the desired network, performs the requisite data pre-processing, and sets the network training options. The input to this function is a structure with fields for each of the hyperparameters defined in the Experiment Manager interface. In the **Setup Function** for this example the input variable is named `params` and the output variables are named `trainingData`, `layers`, and `options` representing the training data, network structure, and training parameters, respectively. The key steps of the **Setup Function** for this example are explained below. Open the example in MATLAB to see the full definition of `compareNetSetup`, the name of the **Setup Function** used in this example.

Download and Extract Data

To speed up the example, open `compareNetSetup` and toggle the `speedUp` flag to `true`. This reduces the size of the dataset to quickly test the basic functionality of the experiment.

```
speedUp = false;
```

The helper function `setupDatastores` downloads the Google Speech Commands Dataset [1], selects the commands for networks to recognize, and randomly partitions the data into training and validation datastores.

```
[adsTrain,adsValidation] = setupDatastores(speedUp);
```

Select the Desired Network and Preprocess Data

Initially transform the datastores based on the preprocessing required by the network type defined in the hyperparameter table, which is accessed as `params.Network`. The helper function `extractSpectrogram` processes the input data to the format expected by each respective network type. The helper function `getLayers` returns a `layerGraph` (Deep Learning Toolbox) object that represents the architecture of the desired network.

```
tdsTrain = transform(adsTrain,@(x)extractSpectrogram(x,params.Network));  
tdsValidation = transform(adsValidation,@(x)extractSpectrogram(x,params.Network));  
  
layers = getLayers(classes,classWeights,numClasses,netName);
```

Now that the datastores are properly set up, read the data into the `trainingData` and `validationData` variables.

```
trainingData = readall(tdsTrain,UseParallel=canUseParallelPool);  
validationData = readall(tdsValidation,UseParallel=canUseParallelPool);  
  
validationData = table(validationData(:,1),adsValidation.Labels);  
trainingData = table(trainingData(:,1),adsTrain.Labels);
```

Set the Training Options

Set the training parameters by assigning a `trainingOptions` (Deep Learning Toolbox) object into the `options` output variable. Train the networks for a maximum of 30 epochs with a patience of 8

epochs using the Adam optimizer. Set the `ExecutionEnvironment` field to "auto" to use a GPU if available. Without using a GPU, training may be very time consuming.

```
maxEpochs = 30;
miniBatchSize = 256;
validationFrequency = floor(numel(TTrain)/miniBatchSize);
options = trainingOptions("adam", ...
    GradientDecayFactor=0.7, ...
    InitialLearnRate=params.LearnRate, ...
    MaxEpochs=maxEpochs, ...
    MiniBatchSize=miniBatchSize, ...
    Shuffle="every-epoch", ...
    Plots="training-progress", ...
    Verbose=false, ...
    ValidationData=validationData, ...
    ValidationFrequency=validationFrequency, ...
    ValidationPatience=10, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropFactor=0.2, ...
    LearnRateDropPeriod=round(maxEpochs/3), ...
    ExecutionEnvironment="auto");
```

Define Custom Metrics

Experiment Manager enables you to define custom metric functions to evaluate the performance of the networks trained in each trial. Basic metrics like accuracy and loss are computed by default. In this example you compare the size of each of the models as memory usage is an important metric when deploying deep neural networks to real-world applications.

Custom metric functions must take one input argument `trialInfo` which is a structure containing the fields `trainedNetwork`, `trainingInfo`, and `parameters`.

- `trainedNetwork` is the `SeriesNetwork` (Deep Learning Toolbox) object or `DAGNetwork` (Deep Learning Toolbox) object returned by the `trainNetwork` (Deep Learning Toolbox) function.
- `trainingInfo` is a struct containing the training information returned by the `trainNetwork` (Deep Learning Toolbox) function.
- `parameters` is a struct with fields from the hyperparameter table

The metric functions must return a scalar number, logical output, or string which gets displayed in the results table. The custom metrics defined for you in this experiment are listed below:

- `sizeMB` computes the memory allocated to store the networks in megabytes
- `numLearnableParams` counts the number of learnable parameters within each model
- `numIters` computes the number of mini-batches each network trained on before hitting either `MaxEpochs` or violating the `ValidationPatience` parameter in the `trainingOptions` object.

Run Experiment

Press 'Run' in the top pane of the Experiment Manager app to run the experiment. You can select to either run each trial sequentially, simultaneously, or in batches by toggling the mode option. For this experiment, the trials were run sequentially.

Evaluate Results

When the experiment finishes, the results for each trial appear and the metrics are displayed in tabular format. The progress bar shows how many epochs each network trained for before violating the patience parameter in terms of the percentage of MaxEpochs.

Trial	Status	Actions	Progress	Elapsed Time	Network	Training Accu...	Training Loss	Validation Acc...	Validation Loss	sizeMB	numLearnabl...	numIters
1	Complete (...)		<div style="width: 40%;"></div> 40.0%	0 hr 4 min 0 sec	vggish	100.0000	0.0107	94.4574	0.2540	275.2578	72142603.0000	1176.0000
2	Complete (...)		<div style="width: 60%;"></div> 60.0%	0 hr 5 min 21 sec	yamnet	99.6094	0.0120	95.3617	0.2600	12.6763	3228619.0000	1764.0000
3	Complete (...)		<div style="width: 100%;"></div> 100.0%	0 hr 3 min 10 sec	custom	93.7500	0.1264	91.5403	0.1988	0.2799	58787.0000	2940.0000

The table can be sorted by entries in each column by hovering over the right side of the column name cell and clicking the arrow that appears. Click the table icon on the top right to select which columns to show or hide. To first compare the networks by accuracy, sort the table over the Validation Accuracy in descending order.

Trial	Status	Actions	Progress	Elapsed Time	Network	Training Accu...	Training Loss	Validation Acc...	Validation Loss	sizeMB	numLearnabl...	numIters
2	Complete (...)		<div style="width: 60%;"></div> 60.0%	0 hr 5 min 21 sec	yamnet	99.6094	0.0120	95.3617	0.2600	12.6763	3228619.0000	1764.0000
1	Complete (...)		<div style="width: 40%;"></div> 40.0%	0 hr 4 min 0 sec	vggish	100.0000	0.0107	94.4574	0.2540	275.2578	72142603.0000	1176.0000
3	Complete (...)		<div style="width: 100%;"></div> 100.0%	0 hr 3 min 10 sec	custom	93.7500	0.1264	91.5403	0.1988	0.2799	58787.0000	2940.0000

In terms of accuracy, the Yamnet network performs the best followed by VGGish, and lastly the custom network. However, the Elapsed Time column shows that Yamnet takes the longest to train. To compare the size of these networks, sort the table by the sizeMB column.

Trial	Status	Actions	Progress	Elapsed Time	Network	Training Accu...	Training Loss	Validation Acc...	Validation Loss	sizeMB	numLearnabl...	numIters
3	Complete (...)		<div style="width: 100%;"></div> 100.0%	0 hr 3 min 10 sec	custom	93.7500	0.1264	91.5403	0.1988	0.2799	58787.0000	2940.0000
2	Complete (...)		<div style="width: 60%;"></div> 60.0%	0 hr 5 min 21 sec	yamnet	99.6094	0.0120	95.3617	0.2600	12.6763	3228619.0000	1764.0000
1	Complete (...)		<div style="width: 40%;"></div> 40.0%	0 hr 4 min 0 sec	vggish	100.0000	0.0107	94.4574	0.2540	275.2578	72142603.0000	1176.0000

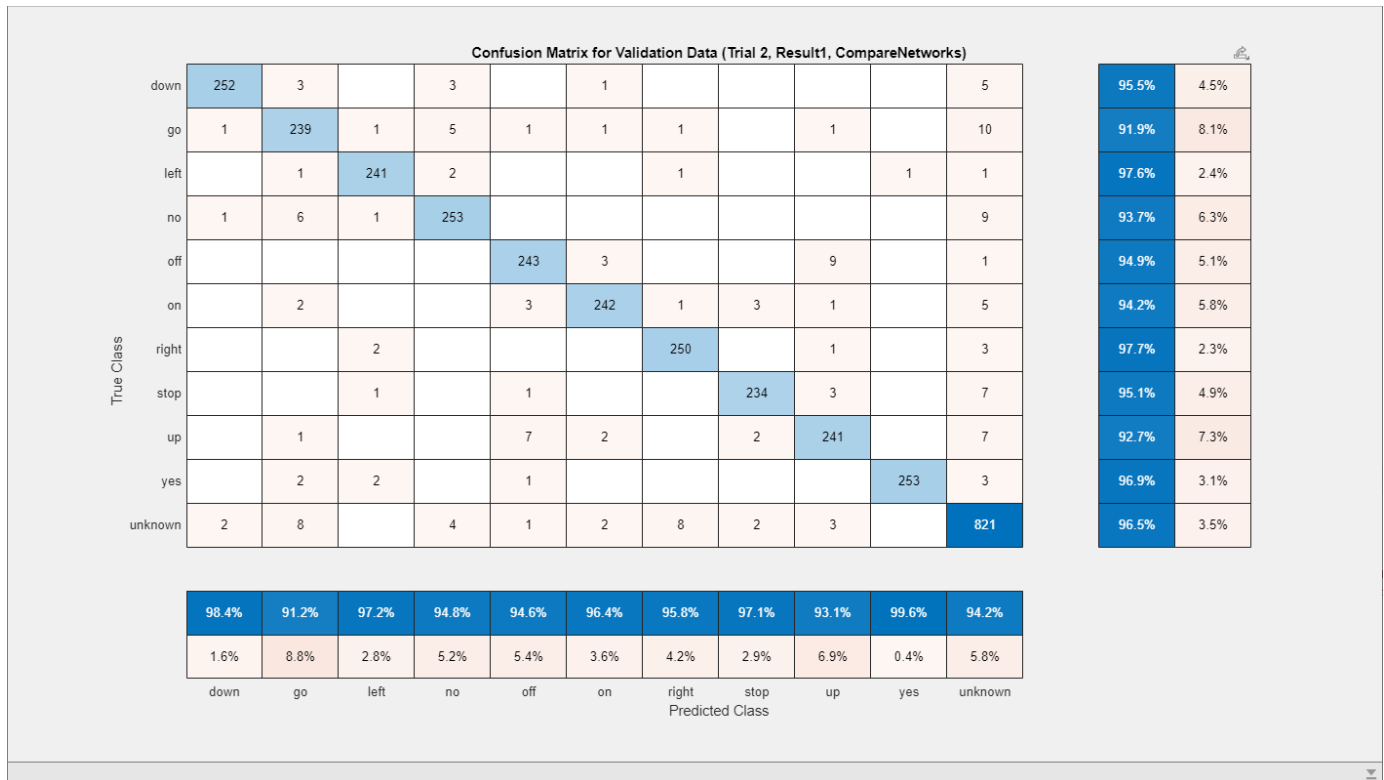
The custom network is the smallest, Yamnet is a few orders of magnitude larger, and VGGish is the largest.

These results highlight the tradeoffs between the different network designs. The Yamnet network performs the best at the classification task at the cost of more training time and a moderately large memory consumption. The VGGish network performs slightly worse in terms of accuracy but requires over 20 times more memory than YAMNet. Lastly, the custom network has the worst accuracy by a small margin but also uses the least memory.

Notice that even though Yamnet and VGGish are pretrained networks, the custom network converges the fastest. Looking at the NumIters column, the custom network takes the most batch iterations to converge because it is learning from scratch. But, since the custom network is much smaller and shallower than the deep pretrained models, each of these batch updates are processed much faster so the overall training time is reduced.

To save one of the trained networks from any of the trials, right click on the corresponding row in the results table and select **Export Trained Network**.

To further analyze any of the individual trials, single click on the corresponding row, and under the **Review Results** tab in the top pane, you can choose to bring up a plot of the training progress or a confusion matrix of the resulting trained model. Below shows the confusion matrix for the Yamnet model from trial 2 of the experiment.



The model struggles most at differentiating between the pair of commands "off" and "up" as well as the pair "no" and "go", although the accuracy is generally uniform across all classes. Further, the model is very confident in predicting the "yes" command as the false positive rate for that class is only .4%.

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>.

Model Smart Speaker in Simulink

This example shows how to model a smart speaker system in Simulink. The smart speaker incorporates voice command recognition and operates in real time.

Introduction

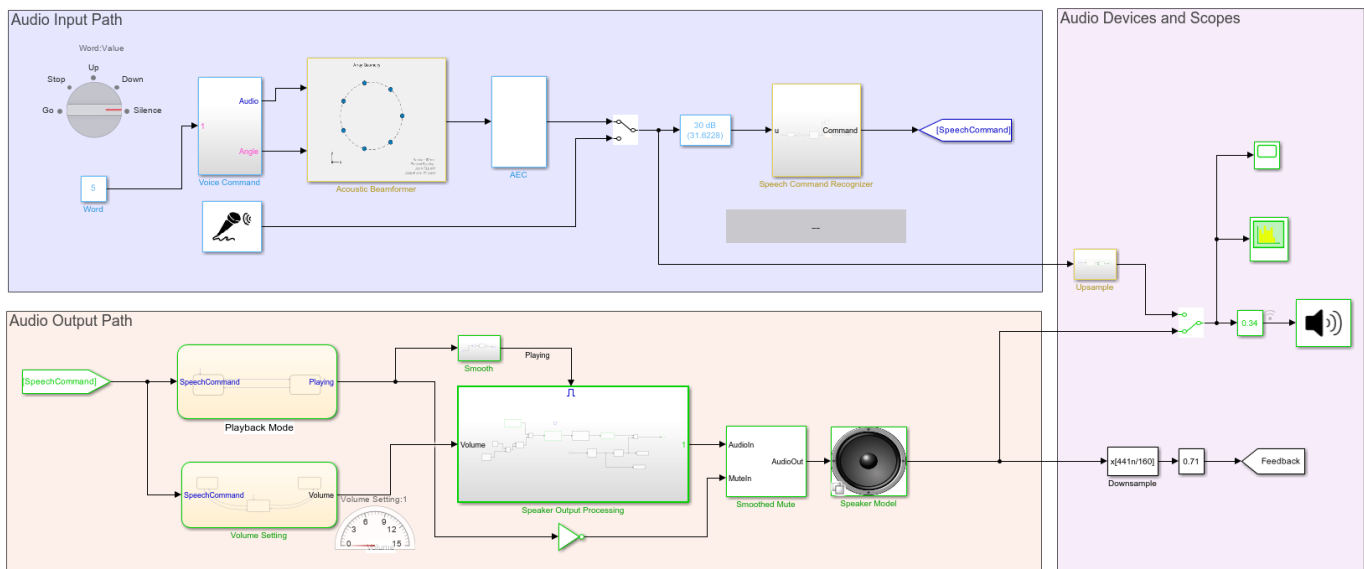
A smart speaker is a speaker that can be controlled by your voice. This example shows a smart speaker model that responds to a number of voice commands. You make the smart speaker play music with the command "Go". You make it stop playing music by saying "Stop". You increase or decrease the music volume with the commands "Up" and "Down", respectively.

Model Summary

The model comprises three main parts:

- 1 An audio input path, representing microphone preprocessing.
- 2 An audio output path, representing loudspeaker output processing.
- 3 Audio devices and scopes, including components to monitor the audio and plot output signals in the time and frequency domains.

```
open_system("audioSmartSpeaker");
```



Copyright 2022 The MathWorks, Inc.

Voice Command Source

You can drive the smart speaker in two ways:

- 1 You can specify commands directly as the model is running through a microphone. Set up your microphone through the dialog of the Audio Device Reader block.
- 2 You can also simulate the reception of signals into a microphone array. In this case, the source of the voice commands is a set of audio files containing prerecorded commands.

Select the voice command source by toggling the manual switch in the Audio Input Path section of the model.

Acoustic Beamforming

You apply acoustic beamforming when you simulate a microphone array. In this case, you model three sound sources in the Voice Command subsystem (the voice command, plus two background noise sources). The Acoustic Beamformer subsystem processes the different sound signals to isolate and enhance the voice command.

Acoustic Echo Cancellation

When you utter commands as music is playing, the music is picked up by the microphone after it reverberates around the room, creating an undesired feedback effect.

The Acoustic Echo Cancellation (AEC) subsystem removes the playback audio from the input signal by using a Normalized LMS adaptive filter. This component applies only when you simulate a microphone array using acoustic beamforming.

You can include or exclude the AEC component by changing the value of the check box on its mask dialog.

To hear the effect of AEC on the input audio signal, flip the manual switch in the Audio Devices and Scopes section of the model.

Speech Command Recognition

You pass the preprocessed speech command to the Speech Command Recognition subsystem. Speech command recognition is based on a pretrained deep learning convolutional network identical to the one in the "Train Speech Command Recognition Model Using Deep Learning" on page 1-332 example.

You extract auditory (Bark) spectrograms from the input audio, which you feed to the pretrained network. The network outputs the predicted command. You use this command to drive the Audio Output Path section of the model.

Control Audio Output Path with State Charts

The decoded speech command goes into two different state charts:

- 1 The first chart controls playback. Music starts playing when the "Go" command is received, and stops playing when "Stop" is received.
- 2 The second chart controls the playback volume by reacting to the commands "Up" and "Down".

Speaker Output Processing

When playback is triggered, the Speaker Output Processing subsystem is enabled. This subsystem contains blocks commonly used to tune audio, such as a Graphic EQ, a multiband parametric equalizer, and a dynamic range controller (limiter). You can tune your system sound as the model is playing by opening the mask dialog of these blocks and changing the values of parameters (for example, the frequencies of the Graphic EQ).

Smoothed Mute

When the music stops playing, it fades smoothly rather than stopping suddenly. This is achieved by the Smoothed Mute block which applies a time-varying gain on the audio signal. This block is based on the System object SmoothedMute.

Speaker Modeling

After Speaker Output Processing and Smoothed Mute, the signal goes into a Speaker Model subsystem. This subsystem allows you to control how the loudspeaker is modeled:

- 1 You can choose a behavioral model which implements the speaker model using basic mathematics Simulink blocks (such as sum, delay, integrator, and gain).
- 2 You can choose a circuit model which implements the speaker model using SimScape components.
- 3 You may also bypass these models if you are using a real, physical loudspeaker to listen to the audio.

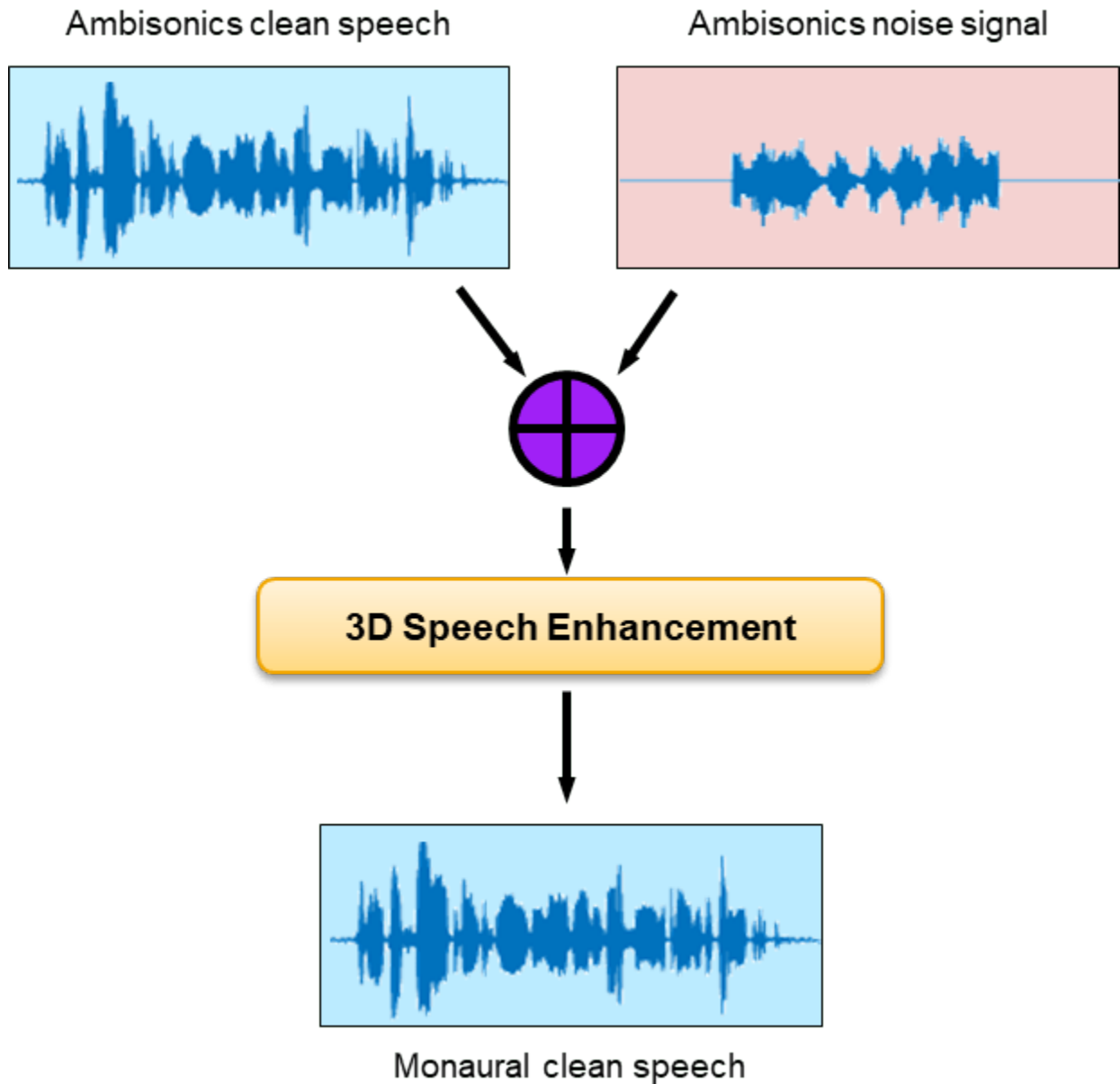
Change the value of the variable `speakerMode` in the base workspace to select one of bypass (`speakerMode=0`), behavioral (`speakerMode=1`), or circuit (`speakerMode=2`).

Audio Devices and Scopes

The model uses a Spectrum Analyzer block to plot the audio signal in the frequency domain, and a time scope to visualize the streaming time-domain audio.

Train 3-D Speech Enhancement Network Using Deep Learning

In this example, you train a filter and sum network (FaSNet) [1] on page 1-964 to perform speech enhancement (SE) using ambisonic data. The model has been updated to use stacked dual-path recurrent neural networks (DPRNNs) which enable memory-efficient joint modeling of short- and long-term sequences [4] on page 1-965. To explore the model trained in this example, see “3-D Speech Enhancement Using Trained Filter and Sum Network” on page 1-973.



Introduction

The aim of speech enhancement (SE) is to suppress the noise in a noisy speech signal. The SE system may be used as a front end in teleconferencing systems, where intelligibility and listening experience are important metrics, or a speech-to-text system, where the word error rate of the downstream speech-to-text system is the important metric.

In this example, you use the L3DAS 2021 Task 1 dataset [2] on page 1-965 to train and evaluate a model that uses B-format ambisonic data to perform speech enhancement. The enhanced speech is output as a mono audio signal. To explore the model trained in this example, see “3-D Speech Enhancement Using Trained Filter and Sum Network” on page 1-973.

Optionally Reduce Data Set

To train the network with the entire data set, set `speedupExample` to `false`. To run this example quickly, set `speedupExample` to `true`. This network requires a large amount of data to achieve reasonable results.

```
speedupExample =  ;
```

Download and Prepare Data

This example uses the L3DAS21 task 1 challenge data set [2] on page 1-965. The train data sets contains 2 multiple-source and multiple-perspective (MSMP) B-format ambisonic recordings collected at a sampling rate of 16 kHz. The two microphones are labeled as "A" and "B". In this example, you discard recordings from microphone B. Including microphone B data in the training should improve the final performance. The train and validation splits are provided with the data set. The 3-D speech enhancement data set contains more than 30,000 virtual 3-D audio environments with a duration up to 10 seconds. Each sample contains a spoken voice and other office-like background noises. The target data is the clean monophonic voice signal. The dev dataset is 2.6 GB, the `train100` dataset is 7.6 GB, and the `train360` dataset is 28.6 GB.

Download the data set and point to it using `audioDatastore`.

```
downloadLocation = tempdir;

datasetLocationDev = fullfile(downloadLocation, "L3DAS_Task1_dev");
datasetLocationTrain100 = fullfile(downloadLocation, "L3DAS_Task1_train100");
datasetLocationTrain360 = fullfile(downloadLocation, "L3DAS_Task1_train360");
if speedupExample
    if ~datasetExists(datasetLocationDev)
        urlDev = "https://zenodo.org/record/4642005/files/L3DAS_Task1_dev.zip";
        unzip(urlDev, downloadLocation)
    end

    ads = audioDatastore(fullfile(downloadLocation, "L3DAS_Task1_dev"), IncludeSubfolders=true);
else
    if ~datasetExists(datasetLocationDev)
        urlDev = "https://zenodo.org/record/4642005/files/L3DAS_Task1_dev.zip";
        unzip(urlDev, downloadLocation)
    end
    if ~datasetExists(datasetLocationTrain100)
        urlTrain100 = "https://zenodo.org/record/4642005/files/L3DAS_Task1_train100.zip";
        unzip(urlTrain100, downloadLocation)
    end
end
```

```

if ~datasetExists(datasetLocationTrain360)
    urlTrain360 = "https://zenodo.org/record/4642005/files/L3DAS_Task1_train360.zip";
    unzip(urlTrain360,downloadLocation)
end
adsValidation = audioDatastore(fullfile(downloadLocation,"L3DAS_Task1_dev"),IncludeSubfolders);
adsTrain = audioDatastore([fullfile(downloadLocation,"L3DAS_Task1_train100"), ...
    fullfile(downloadLocation,"L3DAS_Task1_train360")],IncludeSubfolders=true);
end

```

To subset the datastores into targets and predictors, use `subset`. Only use microphone A predictors. Using both microphones should increase model performance at the cost of more training time.

```

if speedupExample
    [~,fileNames] = fileparts(ads.Files);
    targetFiles = ~endsWith(fileNames,["A","B"]);
    micAFiles = endsWith(fileNames,"A");
    T = subset(ads,targetFiles);
    X = subset(ads,micAFiles);
    XTrain = subset(X,1:40);
    TTrain = subset(T,1:40);
    XValidation = subset(X,41:50);
    TValidation = subset(T,41:50);
else
    [~,fileNames] = fileparts(adsTrain.Files);
    targetFiles = ~endsWith(fileNames,["A","B"]);
    micAFiles = endsWith(fileNames,"A");
    TTrain = subset(adsTrain,targetFiles);
    XTrain = subset(adsTrain,micAFiles);

    [~,fileNames] = fileparts(adsValidation.Files);
    targetFiles = ~endsWith(fileNames,["A","B"]);
    micAFiles = endsWith(fileNames,"A");
    TValidation = subset(adsValidation,targetFiles);
    XValidation = subset(adsValidation,micAFiles);
end

```

Remove any files that do not overlap between targets and predictors.

```

[~,hFiles] = fileparts(TTrain.Files);
[~,kFiles] = fileparts(XTrain.Files);
kFiles = erase(kFiles,"_A");
validFiles = intersect(kFiles,hFiles);
targetValidFiles = ismember(validFiles,kFiles);
predictorsValidFiles = ismember(kFiles,validFiles);
TTrain = subset(TTrain,targetValidFiles);
XTrain = subset(XTrain,predictorsValidFiles);

[~,hFiles] = fileparts(TValidation.Files);
[~,kFiles] = fileparts(XValidation.Files);
kFiles = erase(kFiles,"_A");
validFiles = intersect(kFiles,hFiles);
targetValidFiles = ismember(validFiles,kFiles);
predictorsValidFiles = ismember(kFiles,validFiles);
TValidation = subset(TValidation,targetValidFiles);
XValidation = subset(XValidation,predictorsValidFiles);

```

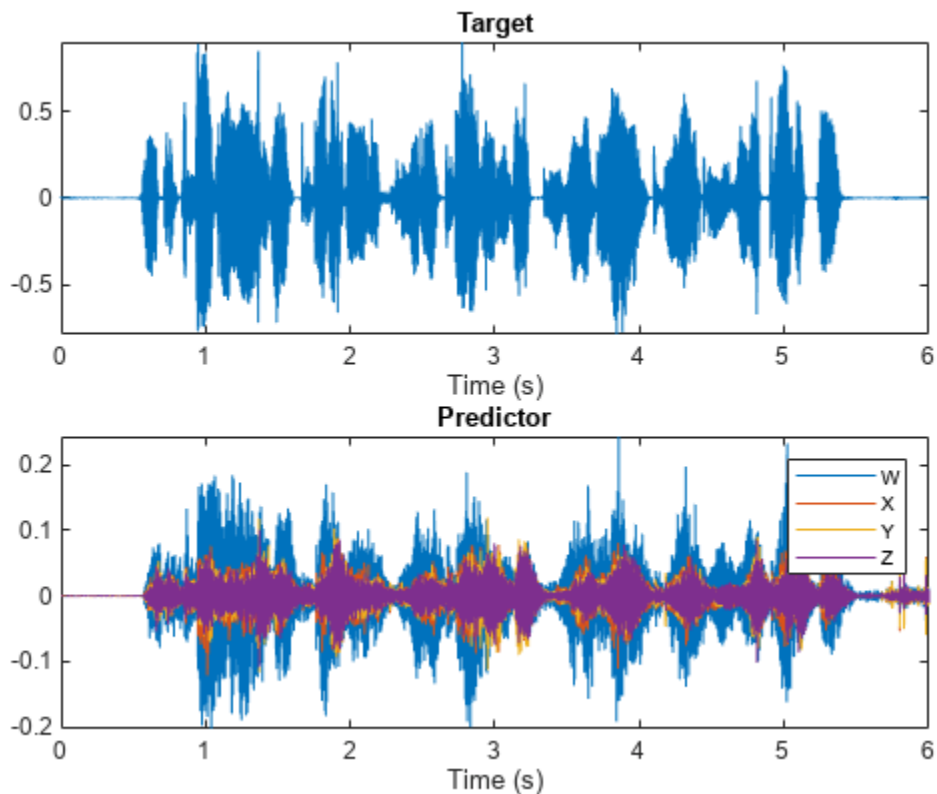
To combine the predictor and target datastores so that reading from the combined datastore returns the predictors and associated target, use `combine`.

```
dsTrain = combine(XTrain,TTrain);  
dsValidation = combine(XValidation,TValidation);
```

Inspect Data

Preview the ambisonic recordings and plot the data.

```
predictor = preview(XTrain);  
target = preview(TTrain);  
  
fs = 16e3; % Known sampling rate of data.  
t = (0:size(target,1)-1)/fs;  
  
tiledlayout(2,1,TileSpacing="tight")  
  
nexttile  
plot(t,target)  
title("Target")  
xlabel("Time (s)")  
axis tight  
  
nexttile  
plot(t,predictor)  
title("Predictor")  
xlabel("Time (s)")  
legend(["W","X","Y","Z"])  
axis tight
```



Listen to the target data, the mean of the ambisonic channels, or one of the ambisonic channels individually.

```
soundSource =  ;
soundsc(soundSource, fs)
```

Word Error Rate (WER)

Choosing an appropriate metric to evaluate a SE system performance depends on the final task of the system. For speech-to-text applications, evaluating the word error rate (WER) using the target speech-to-text system is a common approach. For teleconferencing applications, the short-time objective intelligibility measure (STOI) is a common approach. Similarly, the choice of loss function should depend on the final application of the speech enhancement system. In this example, you attempt to optimize the system to reduce WER for a downstream speech-to-text system. One option for the loss function is to use the WER directly, however this can be prohibitively time-consuming for training, and couples the speech enhancement module tightly with the speech-to-text module. Another approach is to use an auditory-based representation of the targets and predictors and calculate the mean square error between them. This example takes the second approach. To get a baseline for performance analysis, calculate the WER of the target (clean) signal, and the noisy signal using a naive approach to SE (mean over channels). The supporting function, `wordErrorRate` on page 1-965, uses the `wav2vec2.0` option of the `speech2text` functionality. If you have not downloaded the pretrained `wav2vec 2.0` model, the function throws an error with a link to the download. The WER is calculated using Text Analytics Toolbox™.

```
tds = fileDatastore(datasetLocationDev, ...
    ReadFcn=@(x)string(fileread(x)), ...
    IncludeSubfolders=true,FileExtensions=".txt");
[~,tdsFiles] = fileparts(tds.Files);
[~,TValidationFiles] = fileparts(TValidation.Files);
validFiles = ismember(tdsFiles,TValidationFiles);
tds = subset(tds,validFiles);
dsWER = combine(XValidation,TValidation,tds);
```

```
WERa = wordErrorRate(dsWER,TargetWER=true,BaselineWER=true);
```

```
progress = 1.2.3.4.5.6.7.8.9.10.11.12.13.14.15.16.17.18.19.20.21.22.23.24.25.26.27.28.29.30.31.32.33.34.35.36.37.38.39.40.41.42.43.44.45.46.47.48.49.50.51.52.53.54.55.56.57.58.59.60.61.62.63.64.65.66.67.68.69.70.71.72.73.74.75.76.77.78.79.80.81.82.83.84.85.86.87.88.89.90.91.92.93.94.95.96.97.98.99.100
```

```
WERa.Target
```

```
ans = 0.0296
```

```
WERa.Baseline
```

```
ans = 0.4001
```

Filter and Sum Network (FaSNet)

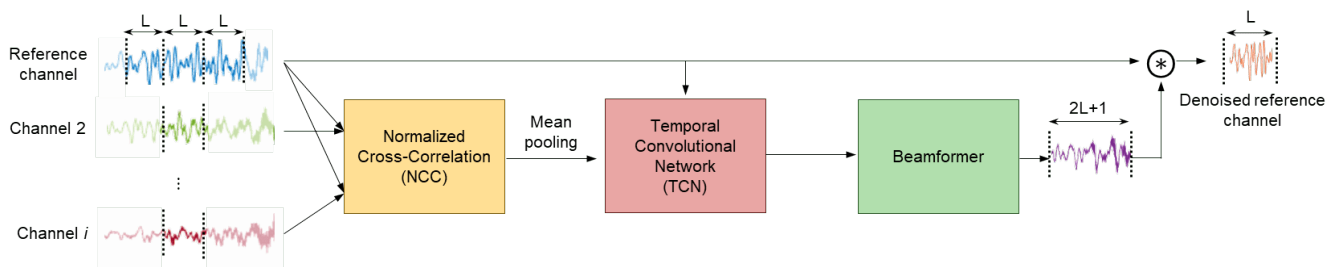
This example uses the filter and sum network (FaSNet) architecture with dual-path recurrent neural networks (DPRNN). FaSNet is a time-domain adaptive beamforming framework consisting of two stages:

- 1 Estimate the beamforming filter for selected reference channel, and then denoise the reference signal.
- 2 Beamform remaining channels using the denoised reference channel.

The FaSNet using DPRNN architecture is implemented in the supporting function FaSNet, which is in the current folder when you open this example.

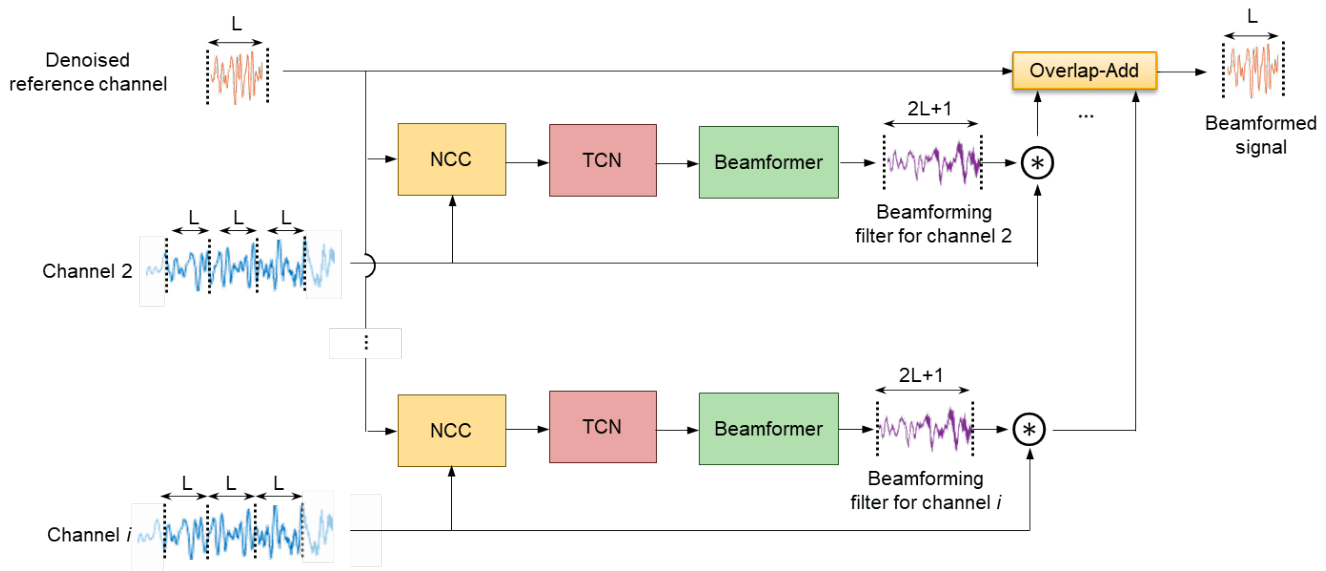
Stage 1: Denoise Reference Mic

In stage one, a normalized cross correlation (NCC) metric is computed between the windows of the reference channel with context and windows of the remaining channels. This example uses cosine similarity as the correlation metric. The metric is pooled across the channels, passed through a temporal convolutional network (TCN), and then through the beamforming filter learner. The output from the beamformer module blocks is then used to filter the reference channel.



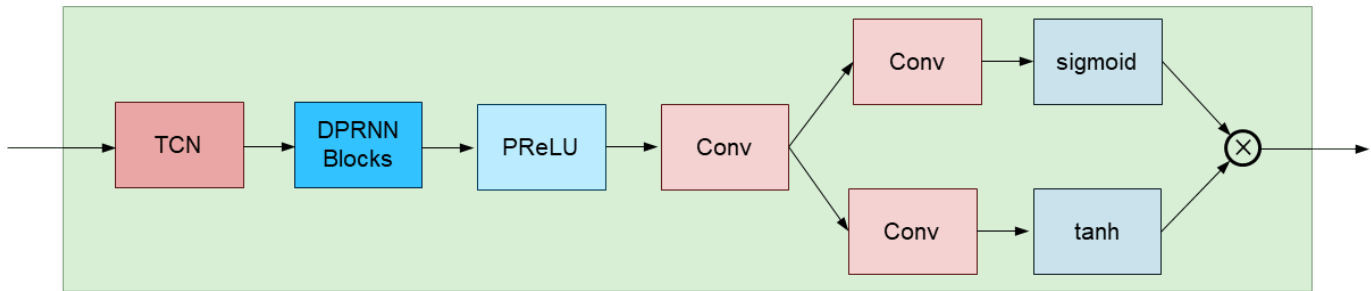
Stage 2: Create Beamformed Signal

In stage two, a NCC metric is computed between the denoised windows of the reference channel and windows of the remaining channels with context. A beamforming filter is learned for each of the remaining channels. Each channel is separately denoised, and then the channels are summed to create the beamformed final signal.



Beamformer

The beamformer module follows the design of [1] on page 1-964 except replaces the stacked TCN blocks with stacked DPRNN blocks.

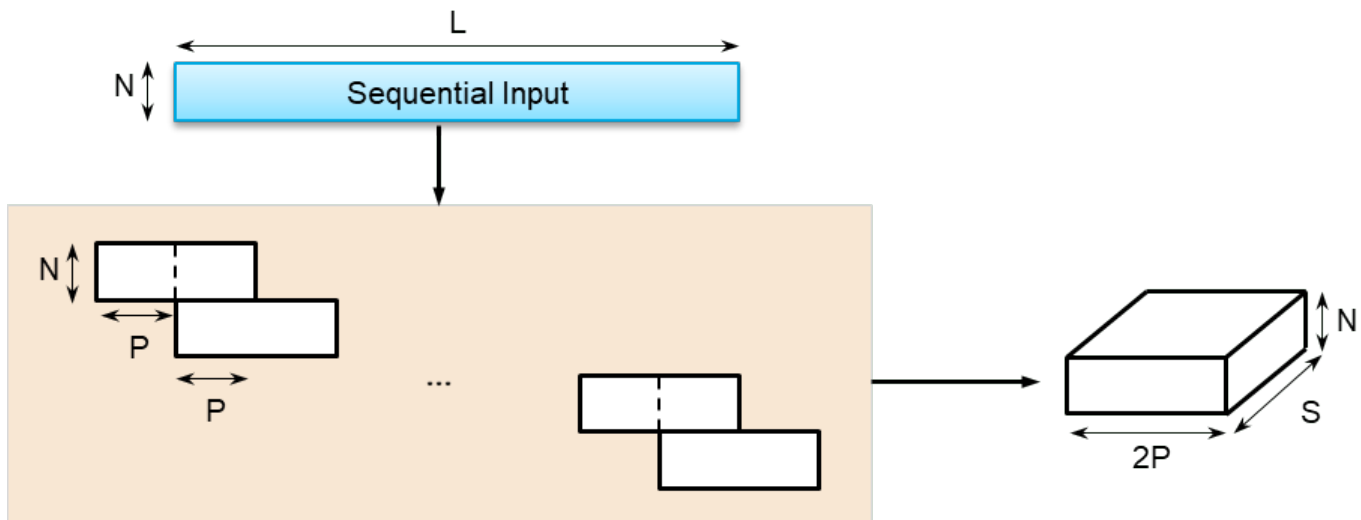


Dual-Path Recurrent Neural Network

Dual-path recurrent neural networks (DPRNN) were introduced in [4] on page 1-965 as a method of organizing RNN layers in a deep structure to model extremely long sequences. DPRNN splits sequential input into chunks and then applies intra- and inter-chunk operations iteratively. The approach has been shown to perform as well or better than 1-D CNN architectures with a significantly smaller model size. The DPRNN model consists of three stages: segmentation, DPRNN blocks (which may be stacked), and then overlap-add reconstruction.

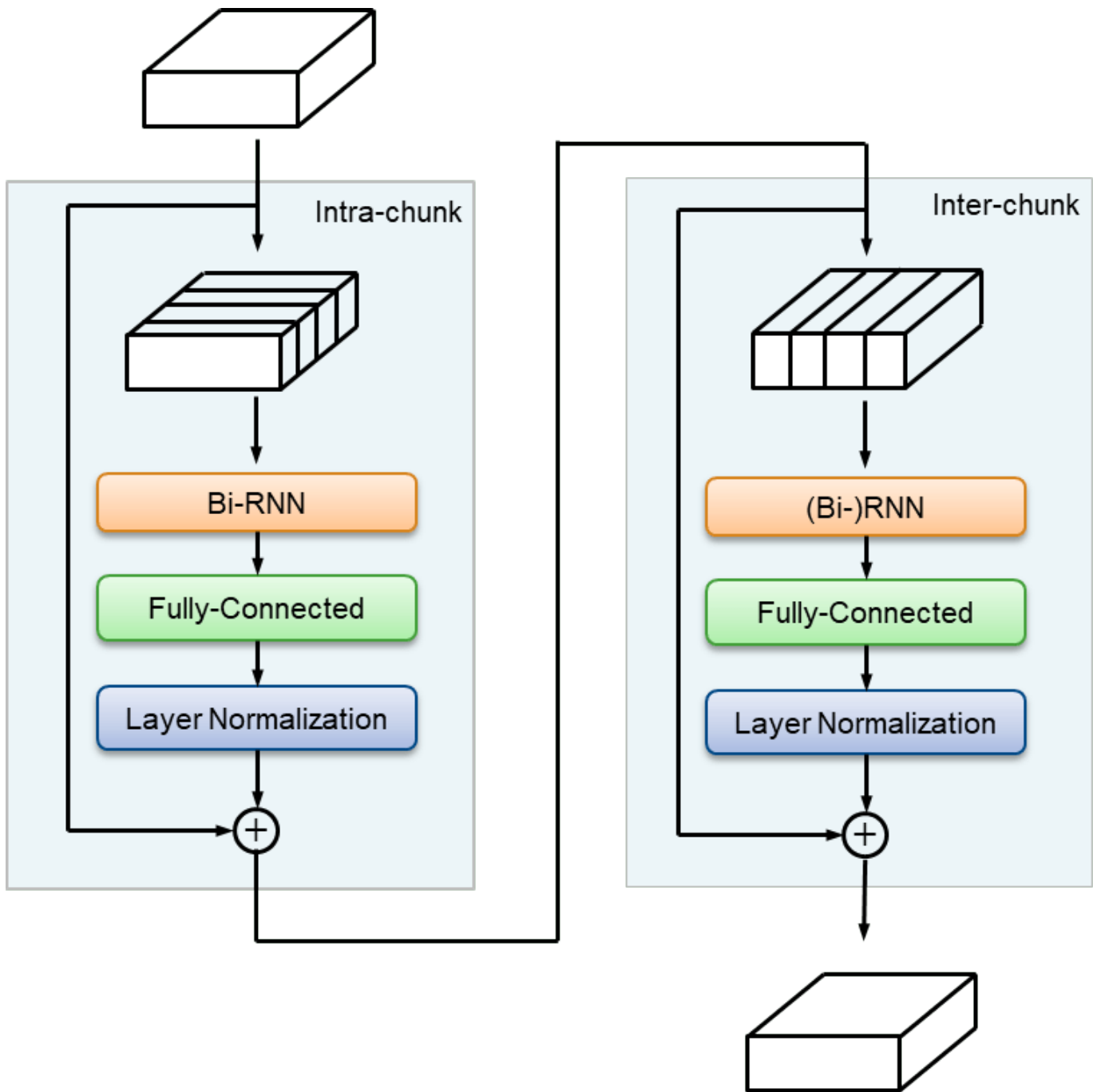
Segmentation

The sequence is split into S segments of length K with overlap P . In this example, $K = 2P$.



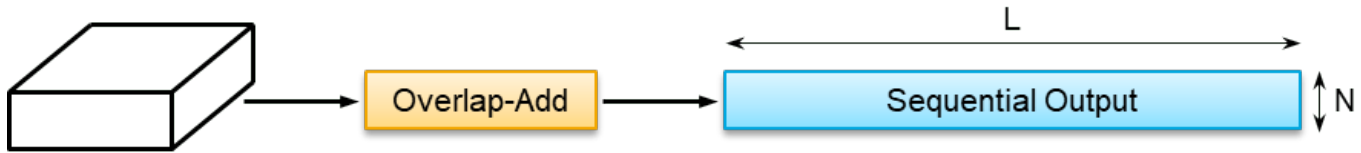
DPRNN Block

The segmented signal passes through B DPRNN blocks. In this example, B is set to 6. Each block contains two sub-modules corresponding to intra- and inter-chunk processing. The intra-chunk RNN is always bi-directional. The intra-chunk RNN processes each segment individually. The inter-chunk RNN may be uni- or bi-directional, depending on latency requirements of your system. In this example, the inter-chunk RNN is bi-directional. The inter-chunk RNN processes along the stacked dimension of length S . The output of each DPRNN block is the same size as the input.



Overlap-Add

The output from the stacked DPRNN blocks is overlapped and added to reconstruct the sequence data.



Define Parameters

Define system-level, FaSNet-level, and DPRNN-level parameters.

```
% System-level parameters
parameters.SampleRate = fs;
parameters.AnalysisLength = 2*parameters.SampleRate;

% FaSNet-level parameters
parameters.WindowLength = 256; % L in FaSNet
parameters.EncoderDimension = 64; % Number filters in TCN
parameters.NumDPRNNBlocks = 6; % Number of stacked DPRNN blocks

% DPRNN-level parameters
parameters.FeatureDimension = 64; % Number of filters in convolutional blocks
parameters.SegmentSize = 24; % 2P
parameters.HiddenDimension = 128; % RNN size
```

Initialize Network Learnables

Use the supporting function, `intitializeLearnables` on page 1-971, to initialize the FaSNet architecture for the specified parameters.

```
learnables = initializeLearnables(parameters);
```

Input Pipeline

Define the mini-batch size. Create `minibatchqueue` (Deep Learning Toolbox) objects to read mini-batches from the training data set. The supporting function `preprocessMiniBatch` on page 1-968 randomly selects a single clip of the specified `parameters.AnalysisLength` on page 1-959 from each audio file in the mini-batch. This approach avoids the need to buffer and save individual audio files, which reduces disk space requirements. The approach has the added benefit of changing the exact sequences seen between epochs. However, this approach puts more emphasis on shorter files in the training data.

```
miniBatchSize = 32 ;

mbqTrain = minibatchqueue(dsTrain, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFcn=@(x,t)preprocessMiniBatch(x,t,parameters.AnalysisLength), ...
    DispatchInBackground=canUseParallelPool);

mbqValidation = minibatchqueue(dsValidation, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFcn=@(x,t)preprocessMiniBatch(x,t,parameters.AnalysisLength), ...
    DispatchInBackground=canUseParallelPool);
```






Training Options

Choose a loss metric as `auditory-mse`, `sample-mse`, or `sample-sisdr`.

- `auditory-mse`: Use the mean-square-error (MSE) between a mel spectrogram computed from the target and a mel spectrogram computed from the prediction.
- `sample-mse`: Use the sample-level MSE between the target and predictor.
- `sample-sisdr`: Use the sample-level scale-invariant signal-to-distortion ratio defined in [3] on page 1-965.

```
lossType =  ;
```

Define the maximum number of epochs, the initial learn rate, and piece-wise learning parameters such as validation patience, learn rate drop factor, and minimum learn rate. The default settings correspond to those reported in [4] on page 1-965 for the task of speaker separation.

```
maxEpochs = 100  ;
initialLearnRate = 0.001  ;
validationPatience = 10  ;
learnRateDropFactor = 0.98  ;
learnRateDropPeriod = 2  ;

if speedupExample
    maxEpochs = 1;
end
```

Initialize parameters required for the training loop.

```
iteration = 0;
bestLoss = inf;
averageGrad = [];
averageSqGrad = [];
learnRate = initialLearnRate;
```

Train Network

Create a `trainingProgressMonitor` to monitor the training loss and validation loss while training.

```
monitor = trainingProgressMonitor( ...
    Metrics=["TrainingLoss","ValidationLoss"], ...
    Info=["Epoch","LearnRate"]);
groupSubPlot(monitor,"Loss",["TrainingLoss","ValidationLoss"])
```

Record the loss for the untrained network.

```
validationLoss = mbqLoss(mbqValidation,learnables,parameters,lossType);
recordMetrics(monitor,0,ValidationLoss=validationLoss)
```

Run the training loop.

```
for epoch = 1:maxEpochs

    % Update plot info
    updateInfo(monitor,Epoch=epoch,LearnRate=learnRate)

    % Shuffle dataset each epoch
```

```

shuffle(mbqTrain)

while hasdata(mbqTrain)
    iteration = iteration + 1;

    % Get next mini batch
    [X,T] = next(mbqTrain);

    % Pass the predictors through the network and return the loss and
    % gradients.
    [loss,gradients] = dlfeval(@modelLoss,learnables,parameters,X,T,lossType);

    % Update the network parameters using the ADAM optimizer.
    [learnables,averageGrad,averageSqGrad] = adamupdate(learnables,gradients, ...
        averageGrad,averageSqGrad,iteration,learnRate);

    % Update training progress visualization
    loss = gather(extractdata(loss));
    recordMetrics(monitor,iteration,TrainingLoss=loss)

    if monitor.Stop
        break
    end
end
if monitor.Stop
    break
end

% Compute validation loss
validationLoss = mbqLoss(mbqValidation,learnables,parameters,lossType);

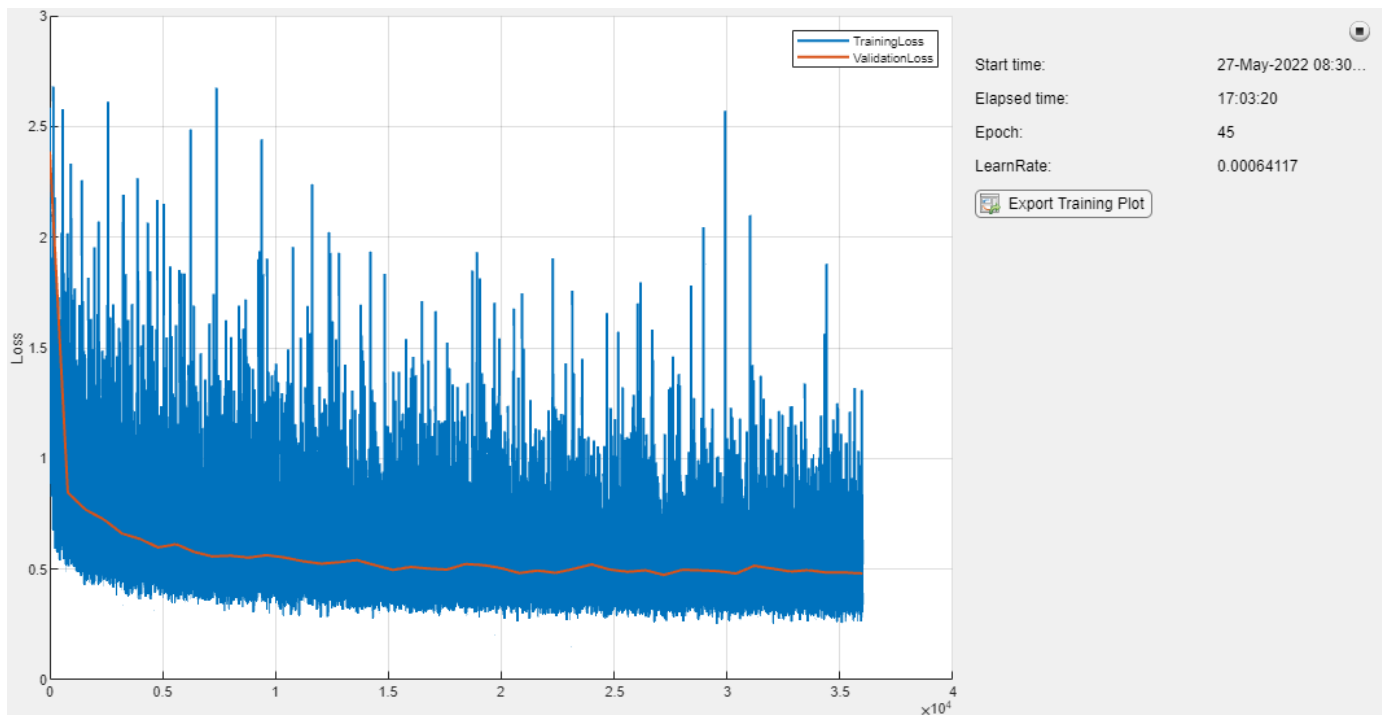
% Update validation progress visualization
recordMetrics(monitor,iteration,ValidationLoss=validationLoss)

% Checkpoint
if validationLoss < bestLoss
    bestLoss = validationLoss;
    bestLossEpoch = epoch;
    save("CheckPoint.mat","bestLoss","learnables","epoch", ...
        "averageGrad","averageSqGrad","iteration","learnRate")
end

if (epoch - bestLossEpoch) > validationPatience
    display("Validation loss did not improve for "+validationPatience+" epochs.")
    break
end

% Reduce the learning rate according to schedule
if rem(epoch,learnRateDropPeriod)==0
    learnRate = learnRate*learnRateDropFactor;
end
end
end

```



"Validation loss did not improve for 10 epochs."

Evaluate System

Load the best performing model.

```
load("Checkpoint.mat")
```

Spot Check Performance

Compare the results of the baseline speech enhancement approach against the FaSNet approach using listening tests and common metrics.

```
dsValidation = shuffle(dsValidation);
[x,t] = read(dsValidation);
predictor = x{1};
target = x{2};
```

As a baseline speech enhancement system, simply take the mean of the predictors across the channels.

```
yBaseline = mean(predictor,2);
```

Pass the noisy speech through the network. The network was trained to process data in 2-second segments. The architecture does accept longer and shorter segments, but performs best on inputs of the same size as it was trained on. Use the `preprocessSignal` on page 1-968 supporting function to split the audio input into the same segment length as your model was trained on. Pass the segments through the FaSNet model. Treat each segment individually by placing the segment dimension along the third dimension, which the FaSNet model recognizes as the batch dimension.

```
y = preprocessSignal(predictor,parameters.AnalysisLength);
```

```

y = FaSNet(dlarray(y),parameters,learnables);

y = gather(extractdata(y)); % Convert to regular array
y = y(:); % Concatenate the segments
y = y(1:size(predictor,1)); % Trim off any zero-padding used to make complete segments

```

Listen to the clean, baseline speech enhanced, and FaSNet speech enhanced signals.

```

dur = size(target,1)/fs;
soundsc(target,fs),pause(dur+1)
soundsc(yBaseline,fs),pause(dur+1)
soundsc(y,fs),pause(dur+1)

```

Compute the baseline and FaSNet sample MSE, auditory-based MSE, and SISDR. Another common metric not implemented in this example is short-time objective intelligibility (STOI) [5] on page 1-965, which is often used both as a training loss function and for system evaluation.

```

yBaselineMSE = 2*mse(yBaseline,target,DataFormat="TB")/size(target,1);
yMSE = 2*mse(y,target,DataFormat="TB")/size(target,1);

yABaseline = extractdata(dlmelspectrogram(yBaseline,parameters.SampleRate));
yA = extractdata(dlmelspectrogram(y,parameters.SampleRate));
targetA = extractdata(dlmelspectrogram(target,parameters.SampleRate));
yBaselineAMSE = mse(yABaseline,targetA,DataFormat="CTB")/(size(targetA,1)*size(targetA,2));
yAMSE = mse(yA,targetA,DataFormat="CTB")/(size(targetA,1)*size(targetA,2));

yBaselineSISDR = sisdr(yBaseline,target);
ySISDR = sisdr(y,target);

```

Plot the target signal, the baseline SE result, and the FaSNet SE result. Display performance metrics in the plot titles.

```

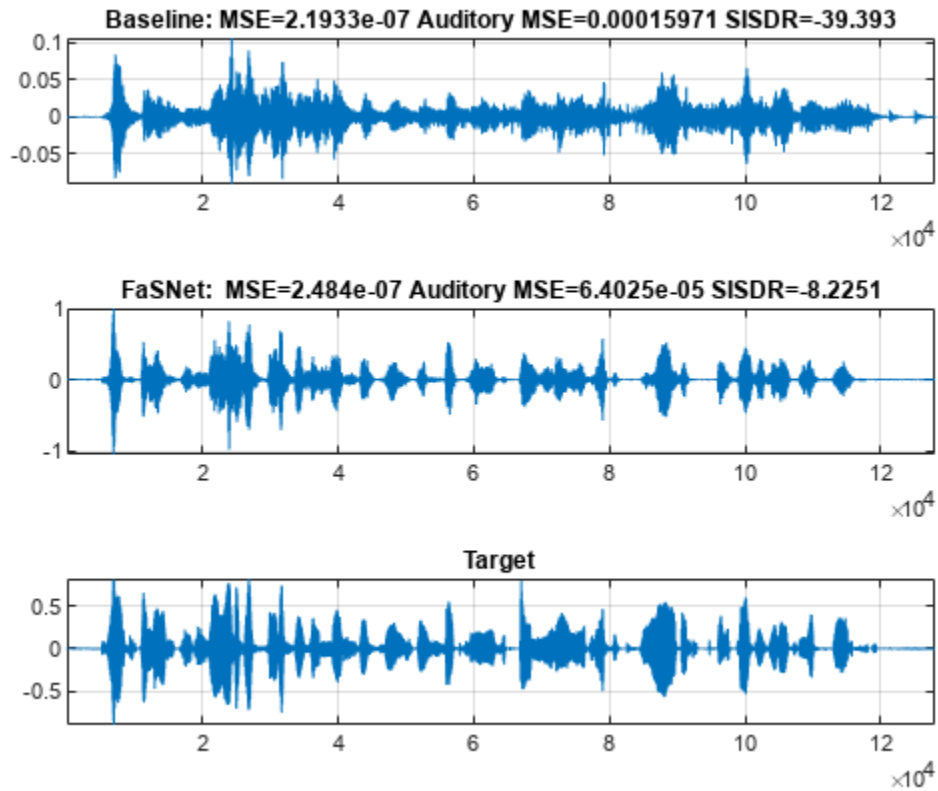
tiledlayout(3,1)

nexttile
plot(yBaseline)
title("Baseline: "+" MSE="+yBaselineMSE+" Auditory MSE="+yBaselineAMSE+" SISDR="+yBaselineSISDR)
grid on
axis tight

nexttile
plot(y)
title("FaSNet: "+" MSE="+yMSE+" Auditory MSE="+yAMSE+" SISDR="+ySISDR)
grid on
axis tight

nexttile
plot(target)
grid on
title("Target")
axis tight

```



Word Error Rate

Evaluate the word error rate after FaSNet processing and compare to the target (clean) signal and the baseline approach.

```
WER = wordErrorRate(dsWER,parameters,learnables,FaSNetWER=true);
```

```
progress = 1.2.3.4.5.6.7.8.9.10.11.12.13.14.15.16.17.18.19.20.21.22.23.24.25.26.27.28.29.30.31.32.
```

```
WERa.Baseline
```

```
ans = 0.4001
```

```
WER.FaSNet
```

```
ans = 0.2760
```

```
WERa.Target
```

```
ans = 0.0296
```

References

[1] Luo, Yi, Cong Han, Nima Mesgarani, Enea Ceolini, and Shih-Chii Liu. "FaSNet: Low-Latency Adaptive Beamforming for Multi-Microphone Audio Processing." In 2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU), 260-67. SG, Singapore: IEEE, 2019. <https://doi.org/10.1109/ASRU46091.2019.9003849>.

[2] Guizzo, Eric, Riccardo F. Gramaccioni, Saeid Jamili, Christian Marinoni, Edoardo Massaro, Claudia Medaglia, Giuseppe Nachira, et al. "L3DAS21 Challenge: Machine Learning for 3D Audio Signal Processing." In 2021 IEEE 31st International Workshop on Machine Learning for Signal Processing (MLSP), 1-6. Gold Coast, Australia: IEEE, 2021. <https://doi.org/10.1109/MLSP52302.2021.9596248>.

[3] Roux, Jonathan Le, et al. "SDR - Half-Baked or Well Done?" *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2019, pp. 626-30. *DOI.org (Crossref)*, <https://doi.org/10.1109/ICASSP.2019.8683855>.

[4] Luo, Yi, et al. "Dual-Path RNN: Efficient Long Sequence Modeling for Time-Domain Single-Channel Speech Separation." *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2020, pp. 46-50. *DOI.org (Crossref)*, <https://doi.org/10.1109/ICASSP40776.2020.9054266>.

[5] Taal, Cees H., Richard C. Hendriks, Richard Heusdens, and Jesper Jensen. "An Algorithm for Intelligibility Prediction of Time-Frequency Weighted Noisy Speech." *IEEE Transactions on Audio, Speech, and Language Processing* 19, no. 7 (September 2011): 2125-36. <https://doi.org/10.1109/TASL.2011.2114881>.

Supporting Functions

Word Error Rate (WER)

```
function out = wordErrorRate(ds,parameters,learnables,nvars)
%wordErrorRate Word error rate (WER)
% wordErrorRate(ds,parameters,learnables) calculates the word error rate
% over all files in the datastore. Specify ds as a combined datastore that
% outputs the predictors and targets and also the text labels.
%
% wordErrorRate(ds,net,TargetWER=TF1,BaselineWER=TF2,FaSNetWER=TF2)
% specifies which signals to calculate the word error rate for. Choose any
% combination of target (the clean monoaural signal), baseline (the noisy
% ambisonic signal converted to monoaural through channel mean) and FaSNet
% (the beamform output from the FaSNet model). By default, WER is computed
% for all options.
%
% This function requires Text Analytics Toolbox(TM).

arguments
    ds
    parameters = [];
    learnables = [];
    nvars.TargetWER = false;
    nvars.BaselineWER = false;
    nvars.FaSNetWER = false;
    nvars.Verbose = true;
end

% Create a speech client object to perform transcription.
transcriber = speechClient("wav2vec2.0",Segmentation="none");
```

```
% Initialize counters
editDistanceTotal_t = 0;
editDistanceTotal_b = 0;
editDistanceTotal_y = 0;
numWordsTotal = 0;
p = 0;

% Reset the datastore
reset(ds)
fprintf("progress = ")
while hasdata(ds)

    % Read from datastore and unpack.
    [data, audioInfo] = read(ds);
    predictors = data{1};
    targets = data{2};
    txt = lower(data{3});
    fs = audioInfo{1}.SampleRate;

    % Put data on GPU if available
    if canUseGPU && nvars.TargetWER
        targets = gpuArray(targets);
    end
    if canUseGPU && (nvars.BaselineWER || nvars.FaSNetWER)
        predictors = gpuArray(predictors);
    end

    % Update the total number of words.
    numWordsTotal = numWordsTotal + numel(split(txt));

    % Tokenize the text.
    tokenizedGroundTruth = tokenizedDocument(txt);
    tokenizedGroundTruth = correctSpelling(tokenizedGroundTruth);

    % Update the total edit distance by passing the signal through
    % speech-to-text, tokenizing the document, and then computing the edit
    % distance against the ground truth text.
    if nvars.TargetWER
        targetsText = speech2text(transcriber, targets, fs);
        T = tokenizedDocument(targetsText);
        T = correctSpelling(T);
        editDistanceTotal_t = editDistanceTotal_t + editDistance(T, tokenizedGroundTruth);
    end
    if nvars.BaselineWER
        predictorsTextBaseline = speech2text(transcriber, mean(predictors, 2), fs);
        B = tokenizedDocument(predictorsTextBaseline);
        B = correctSpelling(B);
        editDistanceTotal_b = editDistanceTotal_b + editDistance(B, tokenizedGroundTruth);
    end
    if nvars.FaSNetWER
        x = preprocessSignal(predictors, parameters.AnalysisLength);
        y = FaSNet(dllarray(x), parameters, learnables);
        y = y.extractdata();
        y = y(:);
        predictorsText = speech2text(transcriber, y, fs);
        Y = tokenizedDocument(predictorsText);
        Y = correctSpelling(Y);
    end
end
```



```

        editDistanceTotal_y = editDistanceTotal_y + editDistance(Y,tokenizedGroundTruth);
    end

    % Print status
    if nvars.Verbose && (100*progress(ds))>p+1
        p = round(100*progress(ds));
        fprintf(string(p)+".")
    end

end
fprintf("...complete.\n")

% Output the results as a struct.
out = struct();
if nvars.FaSNetWER
    out.FaSNet = editDistanceTotal_y/numWordsTotal;
end
if nvars.BaselineWER
    out.Baseline = editDistanceTotal_b/numWordsTotal;
end
if nvars.TargetWER
    out.Target = editDistanceTotal_t/numWordsTotal;
end

end
end

```

Model Loss

```

function [loss,gradients] = modelLoss(learnables,parameters,X,T,lossType)
%modelLoss Model loss for FaSNet
% loss = modelLoss(learnables,parameters,X,T,lossType) calculates the
% FaSNet model loss using the specified loss type. Specify learnables and
% parameters as the learnables and parameters of the FaSNet model. X and T
% are the predictors and targets, respectively. lossType is "sample-mse",
% "sample-sisdr", or "auditory-mse".
%
% [loss,gradients] = modelLoss(...) also calculates the gradients when
% training a model.

% Beamform ambisonic data using FaSNet
Y = FaSNet(X,parameters,learnables);

% Compute specified loss type
switch lossType
    case "sample-sisdr"
        loss = -sisdr(Y,T);
        loss = sum(loss)/size(T,2);
    case "sample-mse"
        loss = 2*mse(Y,T,DataFormat="TB")/size(T,1);
    case "auditory-mse"
        Ym = dlmelspectrogram(Y,parameters.SampleRate);
        Tm = dlmelspectrogram(T,parameters.SampleRate);
        loss = mse(Ym,Tm,DataFormat="CTB")./(size(Tm,1)*size(Tm,2));
end

% If gradients requested, compute them
if nargout==2

```

```
    gradients = dlgradient(loss,learnables);  
end  
  
end
```

Preprocess Mini Batch

```
function [X,T] = preprocessMiniBatch(Xcell,Tcell,N)  
%preprocessMiniBatch Preprocess mini batch  
% [X,T] = preprocessMiniBatch(Xcell,Tcell,N) takes the mini-batch of data  
% read from the combined datastore and preprocesses the data using the  
% preprocessSignalTrain supporting function.  
  
for ii = 1:numel(Xcell)  
    [Xcell{ii},idx] = preprocessSignalTrain(Xcell{ii},Samples=N);  
    Tcell{ii} = preprocessSignalTrain(Tcell{ii},Samples=N,Index=idx);  
end  
  
X = cat(3,Xcell{:});  
T = cat(2,Tcell{:});  
  
end
```

Preprocess Signal for FaSNet

```
function y = preprocessSignal(x,L)  
%preprocessSignal Preprocess signal for FaSNet  
% y = preprocessSignal(x,L) splits the multi-channel  
% signal x into analysis frames of length L and hop L. The output is a  
% L-by-size(x,2)-by-numHop array, where the number of hops depends on the  
% input signal length and L.  
  
% Cast the input to single precision  
x = single(x);  
  
% Get the input dimensions  
N = size(x,1);  
nchan = size(x,2);  
  
% Pad as necessary.  
if N<L  
    numToPad = L-N;  
    x = cat(1,x,zeros(numToPad,size(x,2),like=x));  
else  
    numHops = floor((N-L)/L) + 1;  
    numSamplesUsed = L+(L*(numHops-1));  
    if numSamplesUsed < N  
        numSamplesUnused = N-numSamplesUsed;  
        numToPad = L - numSamplesUnused;  
        x = cat(1,x,zeros(numToPad,nchan,like=x));  
    end  
end  
  
% Buffer the input signal  
x = audio.internal.buffer(x,L,L);
```

```

% Reshape the signal to Time-Channel-Hop.
numHops = size(x,2)/nchan;
x = reshape(x,L,numHops,nchan);
y = permute(x,[1,3,2]);
end

```

Mel Spectrogram Compatible with dLarray

```

function y = dlmelspectrogram(x,fs)
%dlmelspectrogram Mel spectrogram compatible with dLarray
% y = dlmelspectrogram(x,fs) computes a mel spectrogram from the audio
% input.

persistent win overlap fftLength filterBank
if isempty(filterBank)
    win = hann(round(0.03*fs),"periodic");
    overlap = round(0.02*fs);
    fftLength = numel(win);
    filterBank = designAuditoryFilterBank(fs,FFTLength=fftLength);
end

% Short-time Fourier transform
[yr,yi] = dlstft(x,DataFormat="TBC", ...
    Window=win,OverlapLength=overlap,FFTLength=fftLength);

% Power spectrum
y = abs(yr).^2 + abs(yi).^2;

% Apply filter bank
y = permute(y,[1,4,3,2]); % FFTLength-by-NumHops-by-BatchSize
y = pagemtimes(filterBank,y); % NumBins-by-NumHops-by-BatchSize

% Apply log10.
y = log(y+eps)/log(10);
end

```

Scale-Invariant Signal-to-Distortion Ratio (SDR)

```

function metric = sisdr(y,t)
%sisdr Scale-Invariant Signal-to-Distortion Ratio (SDR)
% metric = sisdr(estimate,target) calculates the scale-invariant SDR
% described in [1].
%
% [1] Roux, Jonathan Le, et al. "SDR – Half-Baked or Well Done?" ICASSP 2019 -
% 2019 IEEE International Conference on Acoustics, Speech and Signal
% Processing (ICASSP), IEEE, 2019, pp. 626–30. DOI.org (Crossref),
% https://doi.org/10.1109/ICASSP.2019.8683855.

y = y - mean(y,1);
t = t - mean(t,1);

alpha = sum(y.*t,1)/(sum(t.^2,1) + eps);

etarget = alpha.*t;
eres = y - etarget;

top = sum(etarget.^2);

```

```
bottom = sum(eres.^2);  
metric = 10*log(top./(bottom+eps))/log(10);
```

```
end
```

Preprocess Signal for Training

```
function [y,idx] = preprocessSignalTrain(x,options)  
%preprocessSignalTrain Preprocess signal for training  
% y = preprocessSignalTrain(x) clips out 32000 contiguous samples from x  
% and returns as y. The clip starting point is determined randomly. If x is  
% less than 32000, the signal is padded to 32000.  
%  
% y = preprocessSignalTrain(x,Samples=N) specifies the number of samples to  
% clip as N. If unspecified, Samples defaults to 32000.  
%  
% y = preprocessSignalTrain(...,Index=K) specifies the starting index for  
% clipping. If unspecified, Index is selected randomly with the condition  
% that there are N samples in the clip.
```

arguments

```
x  
options.Samples = 32000  
options.Index = []
```

```
end
```

```
numSamples = size(x,1);  
numChannels = size(x,2);
```

```
% If signal shorter than requested number of samples, pad it.
```

```
if numSamples < options.Samples  
    x = cat(1,x,zeros(options.Samples - numSamples,numChannels,like=x));  
    numSamples = options.Samples;
```

```
end
```

```
% Choose a random starting index in the signal, then clip a segment out of  
% the signal.
```

```
if isempty(options.Index)  
    idx = randi(numSamples-options.Samples+1);
```

```
else
```

```
    idx = options.Index;
```

```
end
```

```
y = x(idx:idx+options.Samples-1,:);
```

```
end
```

Calculate Loss Over Mini-Batch Queue

```
function loss = mbqLoss(mbq,learnables,parameters,lossType)
```

```
%mbqLoss Mini-batch queue loss
```

```
% loss = mbqLoss(mbq,learnables,parameters) calculates the total loss over  
% the mini-batch queue.
```

```
numMiniBatch = 0;  
validationLoss = 0;
```

```
reset(mbq)
```

```
while hasdata(mbq)
```

```

[X,T] = next(mbq);
numMiniBatch = numMiniBatch + 1;
validationLoss = validationLoss + modelLoss(learnables,parameters,X,T,lossType);
end

```

```
loss = validationLoss/numMiniBatch;
```

```
end
```

Initialize FaSNet Learnables

```
function learnables = initializeLearnables(parameters)
```

```
%initializeLearnables Initialize FaSNet learnables
```

```
% learnables = initializeLearnables(parameters) creates a structure
```

```
% containing the randomly initialized learnable weights of FaSNet.
```

```
validateattributes(parameters.SegmentSize,["single","double"],["even","positive"],"initialzeLearnables",1);
```

```
validateattributes(parameters.WindowLength,["single","double"],["even","positive"],"initialzeLearnables",2);
```

```
filterDimension = 2*parameters.WindowLength+1;
```

```
learnables.TCN.conv.weight = dlarray(permute(initializeGlorot(1,parameters.EncoderDimension,3*parameters.WindowLength,filterDimension)));
```

```
learnables.TCN.norm.offset = dlarray(zeros(parameters.EncoderDimension,1,"single"));
```

```
learnables.TCN.norm.scaleFactor = dlarray(ones(parameters.EncoderDimension,1,"single"));
```

```
for jj = 1:2 % Loop over reference mic and other mics
```

```
learnables.("Beamformer"+jj).BN.conv.weight = dlarray(squeeze(initializeGlorot(1,parameters.EncoderDimension,3*parameters.WindowLength,filterDimension)));
```

```
for ii = 1:parameters.NumDPRNNBlocks % Loop over DPRNN blocks
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+1).rnn.forward.weights = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+1).rnn.forward.recurrentWeights = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+1).rnn.forward.bias = dlarray(permute(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension)));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+1).rnn.reverse.weights = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+1).rnn.reverse.recurrentWeights = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+1).rnn.reverse.bias = dlarray(permute(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension)));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+1).projection.weights = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+1).projection.bias = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+1).norm.offset = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+1).norm.scaleFactor = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+2).rnn.weights = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+2).rnn.recurrentWeights = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+2).rnn.bias = dlarray(permute(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension)));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+2).rnn.reverse.weights = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+2).rnn.reverse.recurrentWeights = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+2).rnn.reverse.bias = dlarray(permute(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension)));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+2).projection.weights = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+2).projection.bias = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+2).norm.offset = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
learnables.("Beamformer"+jj).("DPRNN_" + ii).("pass"+2).norm.scaleFactor = dlarray(initializeGlorot(1,parameters.EncoderDimension,parameters.EncoderDimension,parameters.EncoderDimension));
```

```
end
```

```
learnables.("Beamformer"+jj).Output.prelu.alpha = dlarray(0.25);
```

```
learnables.("Beamformer"+jj).Output.conv.weight = dlarray(initializeGlorot(parameters.FeatureDimension,parameters.FeatureDimension,parameters.FeatureDimension,parameters.FeatureDimension));
```

```
learnables.("Beamformer"+jj).Output.conv.bias = dlarray(initializeZeros([1,parameters.FeatureSize,1]));

learnables.("Beamformer"+jj).GenerateFilter.X1.weight = dlarray(permute(initializeGlorot(parameters.FeatureSize,parameters.FeatureSize,parameters.FeatureSize)));
learnables.("Beamformer"+jj).GenerateFilter.X1.bias = dlarray(initializeZeros([1,filterDimensions,1]));

learnables.("Beamformer"+jj).GenerateFilter.X2.weight = dlarray(permute(initializeGlorot(parameters.FeatureSize,parameters.FeatureSize,parameters.FeatureSize)));
learnables.("Beamformer"+jj).GenerateFilter.X2.bias = dlarray(initializeZeros([1,filterDimensions,1]));

end
function weights = initializeGlorot(filterSize,numChannels,numFilters)
    sz = [filterSize,numChannels,numFilters];
    numOut = prod(filterSize)*numFilters;
    numIn = prod(filterSize)*numFilters;

    Z = 2*rand(sz,"single") - 1;
    bound = sqrt(6/(numIn + numOut));

    weights = bound*Z;
    weights = dlarray(weights);

end
function parameter = initializeOrthogonal(numHiddenUnits)
    sz = [4*numHiddenUnits,numHiddenUnits];
    Z = randn(sz,"single");
    [Q,R] = qr(Z,0);
    D = diag(R);
    Q = Q * diag(D./abs(D));
    parameter = dlarray(Q);

end
function bias = initializeUnitForgetGate(numHiddenUnits)
    bias = zeros(4*numHiddenUnits,1,"single");
    idx = numHiddenUnits+1:2*numHiddenUnits;
    bias(idx) = 1;
    bias = dlarray(bias);

end
function parameter = initializeZeros(sz)
    parameter = zeros(sz,"single");
    parameter = dlarray(parameter);

end
function parameter = initializeOnes(sz)
    parameter = ones(sz,"single");
    parameter = dlarray(parameter);

end
end
```

3-D Speech Enhancement Using Trained Filter and Sum Network

In this example, you perform speech enhancement using a pretrained deep learning model. For details about the model and how it was trained, see “Train 3-D Speech Enhancement Network Using Deep Learning” on page 1-951. The speech enhancement model is an end-to-end deep beamformer that takes B-format ambisonic audio recordings and outputs enhanced mono speech signals.

Download Pretrained Network

Download the pretrained speech enhancement (SE) network, ambisonic test files, and labels. The model architecture is based on [1] on page 1-977 and [4] on page 1-977, as implemented in the baseline system for the L3DAS21 challenge task 1 [2] on page 1-977. The data the model was trained on and the ambisonic test files are provided as part of [2] on page 1-977.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "speechEnhancement/FaSNet...");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
netFolder = fullfile(dataFolder, "speechEnhancement");
addpath(netFolder)
```

Load and Inspect Data

Load the clean speech and listen to it.

```
[cleanSpeech, fs] = audioread("cleanSpeech.wav");
soundsc(cleanSpeech, fs)
```

In the L3DAS21 challenge, "clean" speech files were taken from the LibriSpeech dataset and augmented to obtain synthetic tridimensional acoustic scenes containing a randomly placed speaker and other sound sources typical of background noise in an office environment. The data is encoded as B-format ambisonics. Load the ambisonic data. First order B-format ambisonic channels correspond to the sound pressure captured by an omnidirectional microphone (W) and sound pressure gradients X, Y, and Z that correspond to front/back, left/right, and up/down captured by figure-of-eight capsules oriented along the three spatial axes.

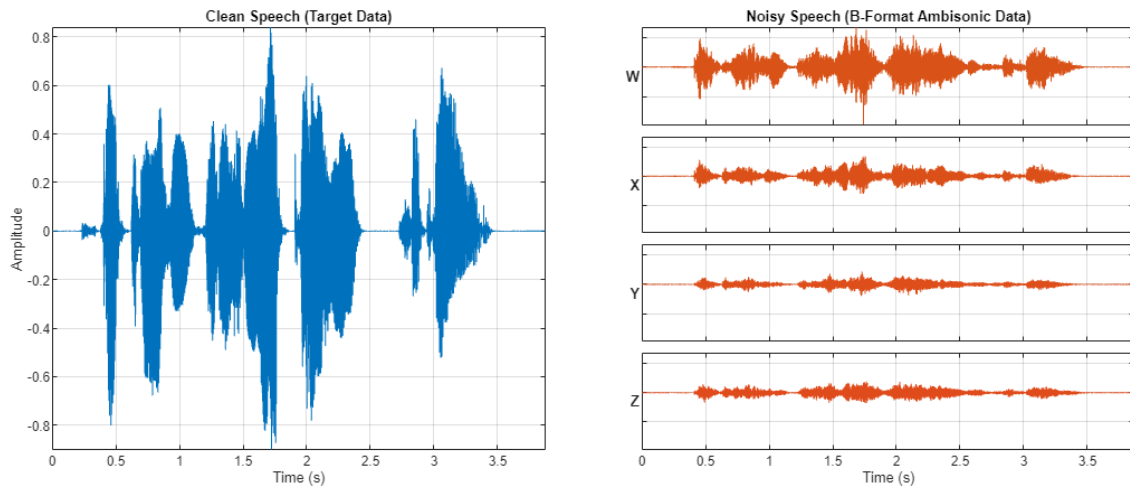
```
[ambisonicData, fs] = audioread("ambisonicRecording.wav");
```

Listen to a channel of the ambisonic data.

```
channel = ;
soundsc(ambisonicData(:, channel), fs)
```

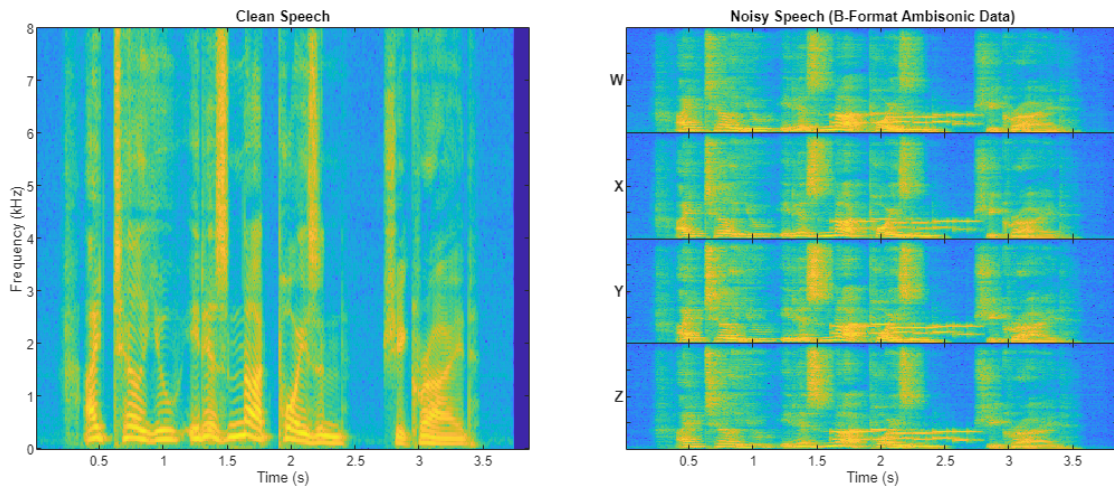
To plot the clean speech and the noisy ambisonic data, use the supporting function `compareAudio` on page 1-977.

```
compareAudio(cleanSpeech, ambisonicData, SampleRate=fs)
```



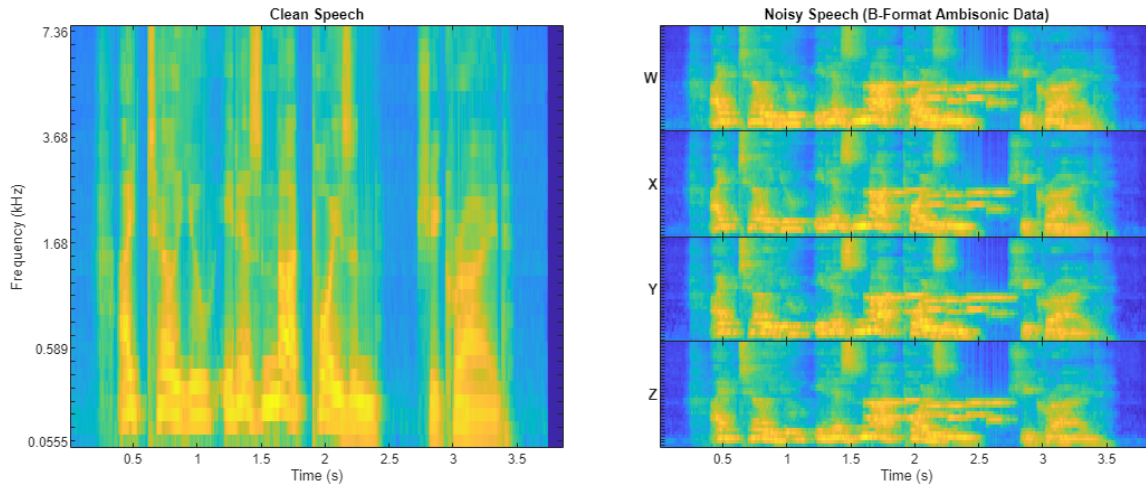
To visualize the spectrograms of the clean speech and the noisy ambisonic data, use the supporting function `compareSpectrograms` on page 1-979.

```
compareSpectrograms(cleanSpeech, ambisonicData)
```



Mel spectrograms are auditory-inspired transformations of spectrograms that emphasize, de-emphasize, and blur frequencies similar to how the auditory system does. To visualize the mel spectrograms of the clean speech and the noisy ambisonic data, use the supporting function `compareSpectrograms` on page 1-979 and set `Warp` to `mel`.

```
compareSpectrograms(cleanSpeech, ambisonicData, Warp="mel")
```

Perform 3-D Speech Enhancement

Use the supporting object, `seModel`, to perform speech enhancement. The `seModel` class definition is in the current folder when you open this example. The object encapsulates the SE model developed in “Train 3-D Speech Enhancement Network Using Deep Learning” on page 1-951. Create the model, then call `enhanceSpeech` on the ambisonic data to perform speech enhancement.

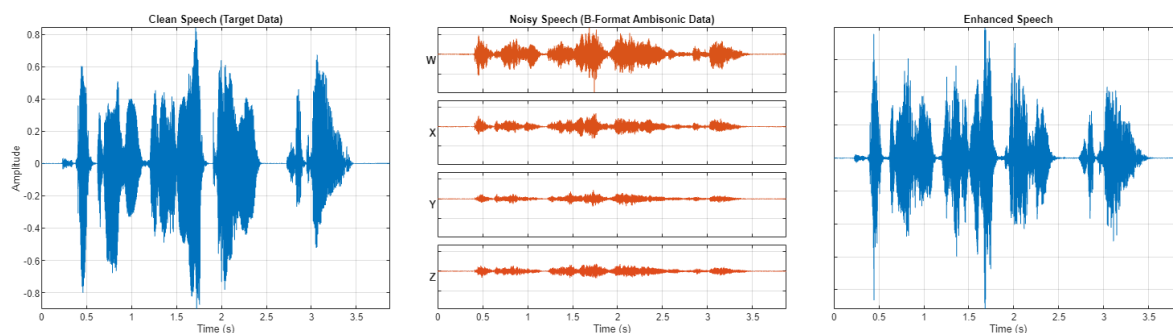
```
model = seModel(netFolder);
enhancedSpeech = enhanceSpeech(model,ambisonicData);
```

Listen to the enhanced speech. You can compare the enhanced speech listening experience with the clean speech or noisy ambisonic data by selecting the desired sound source from the dropdown.

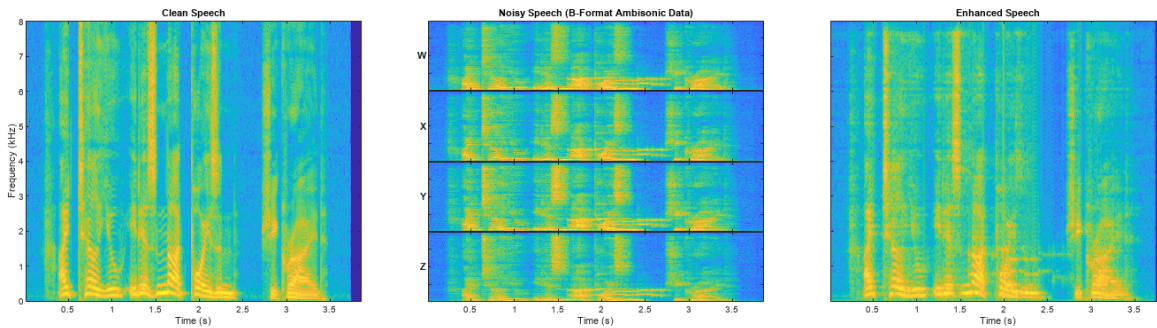
```
soundSource =  ;
soundsc(soundSource, fs)
```

Compare the clean speech, noisy speech, and enhanced speech in the time domain, as spectrograms, and as mel spectrograms.

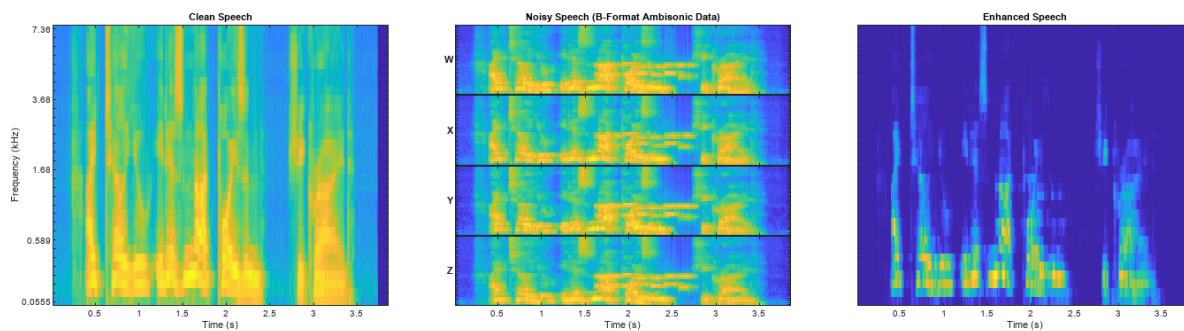
```
compareAudio(cleanSpeech,ambisonicData,enhancedSpeech)
```



```
compareSpectrograms(cleanSpeech,ambisonicData,enhancedSpeech)
```



```
compareSpectrograms(cleanSpeech,ambisonicData,enhancedSpeech,Warp="mel")
```



Speech Enhancement for Speech-to-Text Applications

Compare the performance of the speech enhancement system on a downstream speech-to-text system. Use the wav2vec 2.0 speech-to-text model. This model requires a one-time download of pretrained weights to run. If you have not downloaded the wav2vec weights, the first call to `speechClient` will provide a download link.

Create the wav2vec 2.0 speech client to perform transcription.

```
transcriber = speechClient("wav2vec2.0",segmentation="none");
```

Perform speech-to-text transcription using the clean speech, the ambisonic data, and the enhanced speech.

```
cleanSpeechResults = speech2text(transcriber,cleanSpeech,fs)
```

```
cleanSpeechResults =  
"i tell you it is not poison she cried"
```

```
noisySpeechResults = speech2text(transcriber,ambisonicData(:,channel),fs)
```

```
noisySpeechResults =  
"i tell you it is not parzona she cried"
```

```
enhancedSpeechResults = speech2text(transcriber,enhancedSpeech,fs)
```

```
enhancedSpeechResults =  
"i tell you it is not poison she cried"
```

Speech Enhancement for Telecommunications Applications

Compare the performance of the speech enhancement system using the short-time objective intelligibility (STOI) measurement [5]. STOI has been shown to have a high correlation with the intelligibility of noisy speech and is commonly used to evaluate speech enhancement systems.

Calculate STOI for the omnidirectional channel of the ambisonics, and for the enhanced speech. Perfect intelligibility has a score of 1.

```
stoi(cleanSpeech,ambisonicData(:,channel),fs)
```

```
ans = 0.6950
```

```
stoi(cleanSpeech,enhancedSpeech,fs)
```

```
ans = 0.8393
```

References

[1] Luo, Yi, Cong Han, Nima Mesgarani, Enea Ceolini, and Shih-Chii Liu. "FaSNet: Low-Latency Adaptive Beamforming for Multi-Microphone Audio Processing." In 2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU), 260-67. SG, Singapore: IEEE, 2019. <https://doi.org/10.1109/ASRU46091.2019.9003849>.

[2] Guizzo, Eric, Riccardo F. Gramaccioni, Saeid Jamili, Christian Marinoni, Edoardo Massaro, Claudia Medaglia, Giuseppe Nachira, et al. "L3DAS21 Challenge: Machine Learning for 3D Audio Signal Processing." In 2021 IEEE 31st International Workshop on Machine Learning for Signal Processing (MLSP), 1-6. Gold Coast, Australia: IEEE, 2021. <https://doi.org/10.1109/MLSP52302.2021.9596248>.

[3] Roux, Jonathan Le, et al. "SDR - Half-Baked or Well Done?" *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2019, pp. 626-30. *DOI.org (Crossref)*, <https://doi.org/10.1109/ICASSP.2019.8683855>.

[4] Luo, Yi, et al. "Dual-Path RNN: Efficient Long Sequence Modeling for Time-Domain Single-Channel Speech Separation." *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2020, pp. 46-50. *DOI.org (Crossref)*, <https://doi.org/10.1109/ICASSP40776.2020.9054266>.

[5] Taal, Cees H., Richard C. Hendriks, Richard Heusdens, and Jesper Jensen. "An Algorithm for Intelligibility Prediction of Time-Frequency Weighted Noisy Speech." *IEEE Transactions on Audio, Speech, and Language Processing* 19, no. 7 (September 2011): 2125-36. <https://doi.org/10.1109/TASL.2011.2114881>.

Supporting Functions

Compare Audio

```
function compareAudio(target,x,y,parameters)
%compareAudio Plot clean speech, B-format ambisonics, and predicted speech
```

```
% over time

arguments
    target
    x
    y = []
    parameters.SampleRate = 16e3
end

numToPlot = 2 + ~isempty(y);

f = figure;
tiledlayout(4,numToPlot,TileSpacing="compact",TileIndexing="columnmajor")
f.Position = [f.Position(1),f.Position(2),f.Position(3)*numToPlot,f.Position(4)];

t = (0:(size(x,1)-1))/parameters.SampleRate;

xmax = max(x(:));
xmin = min(x(:));

nexttile(1,[4,1])
plot(t,target,Color=[0 0.4470 0.7410])
axis tight
ylabel("Amplitude")
xlabel("Time (s)")
title("Clean Speech (Target Data)")
grid on

nexttile(5)
plot(t,x(:,1),Color=[0.8500 0.3250 0.0980])
title("Noisy Speech (B-Format Ambisonic Data)")
axis([t(1),t(end),xmin,xmax])
set(gca,Xticklabel=[],YtickLabel=[])
grid on
yL = ylabel("W",FontWeight="bold");
set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")

nexttile(6)
plot(t,x(:,2),Color=[0.8600 0.3150 0.0990])
axis([t(1),t(end),xmin,xmax])
set(gca,Xticklabel=[],YtickLabel=[])
grid on
yL = ylabel("X",FontWeight="bold");
set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")

nexttile(7)
plot(t,x(:,3),Color=[0.8700 0.3050 0.1000])
axis([t(1),t(end),xmin,xmax])
set(gca,Xticklabel=[],YtickLabel=[])
grid on
yL = ylabel("Y",FontWeight="bold");
set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")

nexttile(8)
plot(t,x(:,4),Color=[0.8800 0.2950 0.1100])
axis([t(1),t(end),xmin,xmax])
xlabel("Time (s)")
set(gca,YtickLabel=[])
```

```

grid on
yL = ylabel("Z",FontWeight="bold");
set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")

if numToPlot==3
    nexttile(9,[4,1])
    plot(t,y,Color=[0 0.4470 0.7410])
    axis tight
    xlabel("Time (s)")
    title("Enhanced Speech")
    grid on
    set(gca,YtickLabel=[])
end

end

```

Compare Spectrograms

```

function compareSpectrograms(target,x,y,parameters)
%compareSpectrograms Plot spectrograms of clean speech, B-format
% ambisonics, and predicted speech over time

arguments
    target
    x
    y = []
    parameters.SampleRate = 16e3
    parameters.Warp = "linear"
end
fs = parameters.SampleRate;

switch parameters.Warp
    case "linear"
        fn = @(x)spectrogram(x,hann(round(0.03*fs),"periodic"),round(0.02*fs),round(0.03*fs),fs);
    case "mel"
        fn = @(x)melSpectrogram(x,fs);
end

numToPlot = 2 + ~isempty(y);

f = figure;
tiledlayout(4,numToPlot,TileSpacing="tight",TileIndexing="columnmajor")
f.Position = [f.Position(1),f.Position(2),f.Position(3)*numToPlot,f.Position(4)];

nexttile(1,[4,1])
fn(target)
fh = gcf;
fh.Children(1).Children(1).Visible="off";
title("Clean Speech")

nexttile(5)
fn(x(:,1))
fh = gcf;
fh.Children(1).Children(1).Visible="off";
set(gca,YtickLabel=[],XtickLabel=[],Xlabel=[])
yL = ylabel("W",FontWeight="bold");

```

```

set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")
title("Noisy Speech (B-Format Ambisonic Data)")

nexttile(6)
fn(x(:,2))
fh = gcf;
fh.Children(1).Children(1).Visible="off";
set(gca,Yticklabel=[],XtickLabel=[],Xlabel=[])
yL = ylabel("X",FontWeight="bold");
set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")

nexttile(7)
fn(x(:,3))
fh = gcf;
fh.Children(1).Children(1).Visible="off";
set(gca,Yticklabel=[],XtickLabel=[],Xlabel=[])
yL = ylabel("Y",FontWeight="bold");
set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")

nexttile(8)
fn(x(:,4))
fh = gcf;
fh.Children(1).Children(1).Visible="off";
set(gca,Yticklabel=[])
yL = ylabel("Z",FontWeight="bold");
set(yL,Rotation=0,VerticalAlignment="middle",HorizontalAlignment="right")

if numToPlot==3
    nexttile(9,[4,1])
    fn(y)
    fh = gcf;
    fh.Children(1).Children(1).Visible="off";
    set(gca,Yticklabel=[],Ylabel=[])
    title("Enhanced Speech")
end
end

```

Short-Time Objective Intelligibility (STOI) Measure

```

function metric = stoi(t,y,fs)
%STOI Short-time objective intelligibility measure (STOI)
% metric = stoid(t,y,fs) returns the short-time objective intelligibility
% measurement (STOI) defined in [1] and [2]. t is the clean speech signal,
% y is the predicted speech signal, and fs is the sample rate.
%
% References
% [1] C. H. Taal, R. C. Hendriks, R. Heusdens, and J. Jensen. A Short-Time
% Objective Intelligibility Measure for Time-Frequency Weighted Noisy
% Speech. In Acoustics Speech and Signal Processing (ICASSP), pages
% 4214-4217. IEEE, 2010.
%
% [2] C. H. Taal, R. C. Hendriks, R. Heusdens, and J. Jensen. An Algorithm
% for Intelligibility Prediction of Time-Frequency Weighted Noisy Speech.
% IEEE Transactions on Audio, Speech and Language Processing,
% 19(7):2125-2136, 2011.

% Define parameters
designFs = 10e3;

```

```

windowLength = 256;
fttLength = 512;
numBands = 15;
cf1 = 150; % Center frequency of first 1/3 octave band in Hz.
N = 30; % Number of frames for intermediate intelligibility
Beta = -15; % Lower bound of signal to distortion ratio (SDR)
clipConstant = 10^(-Beta/20);
energyThreshold = 40; % Speech dynamic range

if fs ~= designFs
    t = cast(resample(double(t),designFs,fs),like=t);
    y = cast(resample(double(y),designFs,fs),like=y);
end

% Remove silent frames
[t,y] = removeSilentFrames(t,y,energyThreshold>windowLength);

% Compute magnitude short-time Fourier transform
T = stft(t,FFTLength=fttLength,Window=hann(windowLength), ...
    OverlapLength=windowLength/2,FrequencyRange="onesided");
Y = stft(y,FFTLength=fttLength,Window=hann(windowLength), ...
    OverlapLength=windowLength/2,FrequencyRange="onesided");
T = abs(T);
Y = abs(Y);

% Design frequency-domain octave filter bank
fb = designOctaveFilterBank(designFs,fttLength,numBands,cf1);

% Apply octave filter bank
T = fb*T;
Y = fb*Y;

% Compute intelligibility measurement
djm = zeros(numBands,length(N:size(T,2)));
for m = N:size(T,2)
    % Isolate region of N consecutive TF-units
    Tj = T(:,(m-N+1):m);
    Yj = Y(:,(m-N+1):m);

    % Calculate alpha
    alpha = sqrt(sum(Tj.^2,2)./sum(Yj.^2,2));

    alphaYj = Yj.*alpha;
    Ypj = min(alphaYj,Tj+Tj*clipConstant); % Eq 3 from [1]

    % Eq 4 from [1]
    yn = Ypj - mean(Ypj,2);
    xn = Tj - mean(Tj,2);
    djm(:,m-N+1) = dot(xn./vecnorm(xn,2,2),yn./vecnorm(yn,2,2),2);
end

% Average intermediate intelligibility over all bands and frames (eq 5 in
% [1])
metric = mean(djm(:));

% Remove Silent Frames
function [tS,yS] = removeSilentFrames(t,y,eThreshold,N)

```

```
win = hanning(N);

tb = buffer(t,N,N/2,"nodelay");
tbwin = tb.*win;

frameEnergy = 20*log10(vecnorm(tbwin)./sqrt(N));

mask = (frameEnergy-max(frameEnergy)+eThreshold)>0;

tS = tbwin;
tS(:,~mask) = [];
tS = tS(1:N/2,2:end) + tS(N/2+1:end,1:end-1);
tS = [tb(1:N/2,1);tS(:);tb(N/2+1:end,end)];

yb = buffer(y,N,N/2,"nodelay");
ybwin = yb.*win;
yS = ybwin;
yS(:,~mask) = [];
yS = yS(1:N/2,2:end) + yS(N/2+1:end,1:end-1);
yS = [yb(1:N/2,1);yS(:);yb(N/2+1:end,end)];

end

% Design Octave Filter Bank
function fb = designOctaveFilterBank(fs,fftLength,numBands,cf1)
    f = linspace(0,fs,fftLength+1);
    f = f(1:(fftLength/2+1));
    k = 0:(numBands-1);

    cf = 2.^(k/3)*cf1;
    fl = sqrt((2.^(k/3)*cf1).*2.^((k-1)/3)*cf1);
    fr = sqrt((2.^(k/3)*cf1).*2.^((k+1)/3)*cf1);

    fb = zeros(numBands,numel(f));

    fp = f';
    [~,bandLow] = min((fp-fl).^2);
    [~,bandHigh] = min((fp-fr).^2);

    for ii = 1:numel(cf)
        fb(ii,bandLow(ii):(bandHigh(ii)-1)) = 1;
    end
end
end
```


Automated Design of Audio Filters for Room Equalization

This example combines Optimization Toolbox™ and Audio Toolbox™ to develop an algorithm that automatically tunes a set of filter parameters.

There are many audio applications where it is desirable to compute parametric equalizer parameters to fit an arbitrary frequency response. For instance, one could fit a filter response to a measured impulse response (IR) to obtain a lower-order implementation of the same filter. Alternatively, one can apply correction to a measured loudspeaker response (anechoic or in-room) to smooth out any imperfections and create a perceptually flat frequency response. The latter is demonstrated here by designing an algorithm that automatically tunes the parameters of N parametric equalizers such that when the resulting EQ is applied to the speaker, the frequency response is perceived as flat in the room.

This example goes over the following steps:

- Measure the in-room response of a loudspeaker using the Impulse Response Measurer
- Compute a fractional-octave smoothed response
- Take into account the microphone calibration data
- Compute a target response that is perceptually optimal for the given loudspeaker system and room configuration
- Optimize a set of filter parameters that modifies the response to better fit the target
- Produce an audio filter or audio files to evaluate the results using headphones or listening room

In-Room Measurement

The first step is to obtain measurements for the system that needs to be improved.

Set up a full duplex sound interface so that it can both play on the loudspeaker, and record with a calibration microphone (such as a Behringer ECM8000 connected to a sound interface capable of supplying phantom power). Place the microphone on a stand so that you can move it into the listening position(s). You can start with the microphone centered in the listener's position, but measuring at several positions will help reduce the chances of overcorrecting for issues like a high frequency dip in the response that could only be present in a region smaller than a listener's head.

Start the **Impulse Response Measurer** app by entering `impulseResponseMeasurer` at the command prompt. You can also click the app icon on the **Apps** tab of the MATLAB® Toolstrip.

Verify that the correct audio device is selected. Change the sample rate if desired (this example uses 96 kHz). Set the player and recorder channels to the loudspeaker and microphone, respectively.

Select the Swept Sine method. Set the number of runs to average several measurement together. This example uses five. Set the duration per run to have time for a long enough swept sine and a period of silence that is long enough for the reverberation to completely die off, which can be several seconds in a typical room.

In the advanced settings, set a pause between runs that allows time to move the microphone around the initial "center" position. You must keep silent during the "silence" part of the measurement, but you can move the microphone and make noise during this pause. The start frequency should be set below the range of the loudspeaker (10 Hz might be a good starting point). The stop frequency can be set to half the sampling rate, unless measuring a subwoofer with limited high-frequency range. Set

the sweep duration to a few seconds, and make sure there is enough duration left for silence at the end.

You may test levels with the level meter or try a first capture with 1 short run. Set playback level loud enough to hide any background noise in the room and set the microphone level so that it is high but does not overload/clip.

Now you can capture and save export the data to a MAT file. The rest of this example uses a file provided here.

Import the Measurement

Import the **last** measurement that was exported by the application (by addressing with `end`). The data used here is in a similar (but compatible) format.

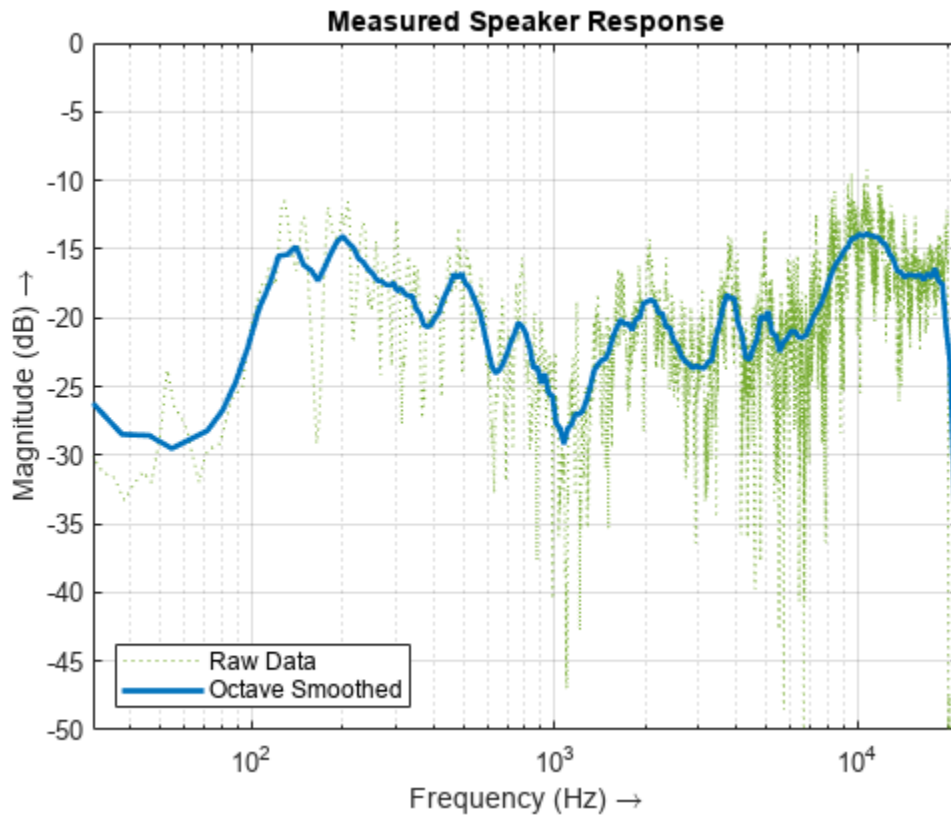
```
load('measured_ir_data.mat','measurementData');
Fs = measurementData.SampleRate(end);
ir = measurementData.ImpulseResponse(end).Amplitude;
frequency = measurementData.MagnitudeResponse(end).Frequency;
if isfield(measurementData.MagnitudeResponse,'PowerDb')
    magnitudeDB = measurementData.MagnitudeResponse(end).PowerDb;
else % Version R2022b of IRM has renamed this field to Magnitude (dB)
    magnitudeDB = measurementData.MagnitudeResponse(end).MagnitudeDB;
end
```

Using a helper function provided here, smooth the response by 1/24-octave sections. Since `powerDB` is a measurement, add extra smoothing (last argument set to `true`).

```
fullRange = [10,Fs/2]; % Full audio range (for the plots)
[powerFR,cfFR] = octaveAverage(frequency,db2mag(magnitudeDB),24,fullRange,true);
pdbFR = 20*log10(powerFR);
```

Plot the measurement and the smoothed response.

```
semilogx(frequency,magnitudeDB,':',Color="#77AC30")
hold on
plot(cfFR,pdbFR,'b',LineWidth=2,Color="#0072BD")
title('Measured Speaker Response')
legend('Raw Data','Octave Smoothed',Location='southwest')
xlabel('Frequency (Hz) \rightarrow')
ylabel('Magnitude (dB) \rightarrow')
yrange = [-50 0];
axis([30 22e3 yrange])
hold off
grid on
```

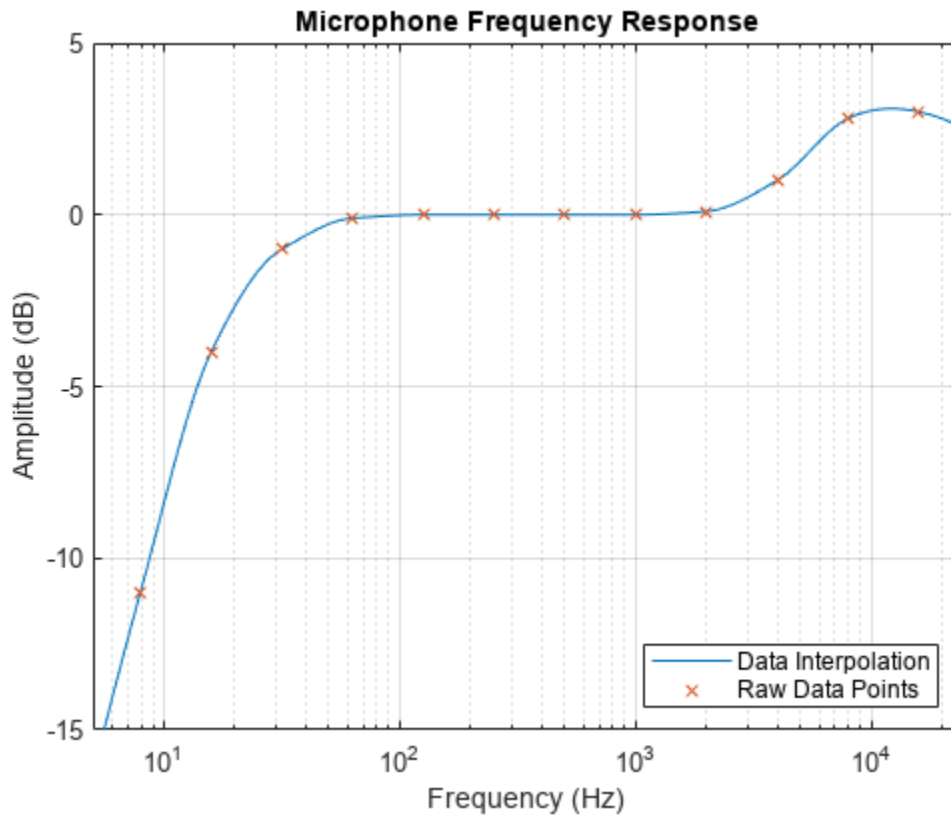


Microphone Calibration

If there is calibration data available for the microphone, subtract it from the measurement.

In this case, generic calibration data for the Behringer ECM8000 microphone is used. Calling the helper function without capturing the output produces a convenience plot.

```
getMicCalibration()
```



Subtract the microphone calibration data from the measured response, using the same frequency values.

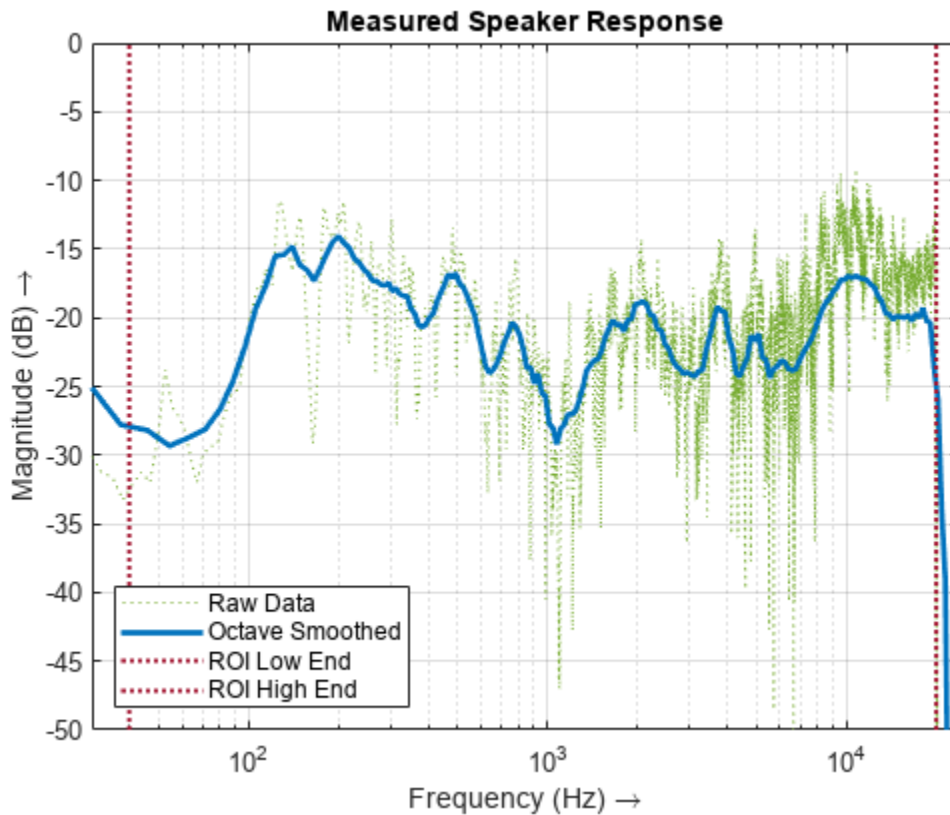
```
micGainDB = getMicCalibration(cfFR);
pdbFRmic = pdbFR - micGainDB;
```

Determine a "range of interest" (ROI) for the optimization based on the measurement above and the manufacturer specifications for our loudspeaker. The bookshelf speaker measured above has a range of 60 Hz to 22 kHz according to the manufacturer. Set the ROI to a range of 40 Hz to 20 kHz to slightly increase the low end and to take into account the steep decline above 20 kHz.

```
ROI = [40 20e3]; % specify the region of interest for the optimization
```

Plot the corrected response and the ROI region.

```
semilogx(frequency,magnitudeDB, ':',Color="#77AC30")
hold on
plot(cfFR,pdbFRmic,Color="#0072BD",LineWidth=2)
plot([ROI(1) ROI(1)],yrange, ':',Color="#A2142F",LineWidth=1.5);
plot([ROI(2) ROI(2)],yrange, ':',Color="#A2142F",LineWidth=1.5);
title('Measured Speaker Response')
legend('Raw Data','Octave Smoothed',...
       'ROI Low End','ROI High End',Location='southwest')
xlabel('Frequency (Hz) \rightarrow')
ylabel('Magnitude (dB) \rightarrow')
axis([30 24e3 -50 0])
hold off
grid on
```



Compute a Target Response

The next step is to determine a suitable target response for the system in the ROI. The main goal is to provide a response that is *perceived* as "flat", and potentially extend the low frequencies (within reasonable limits).

It is important to consider whether the response being optimized is a loudspeaker measurement in an anechoic chamber, or a loudspeaker in a room measured from a listening position. In the former case, the target could be a constant level (with a roll off outside of the loudspeaker's range). In the later case, which is the case for this example, the response at the listening position also depends on the room. The loudspeaker is perceived as "flat" if it exhibits a constant slope down. The degree of that slope depends on factors such as the distance of the listener. The perceived quality also depends on the low frequency cutoff point (approximately -10 dB), so the target curve can be used to extend the low frequencies if the boost remains moderate. Keep in mind that loudspeaker distortion increases in the lower frequencies, and the overall limits of the amplifier and loudspeaker should not be exceeded.

To compute a target response, fit a straight line (in the log-frequency domain) onto the frequency response (in dB) for a subset of the ROI. Add a roll off to the lower frequencies.

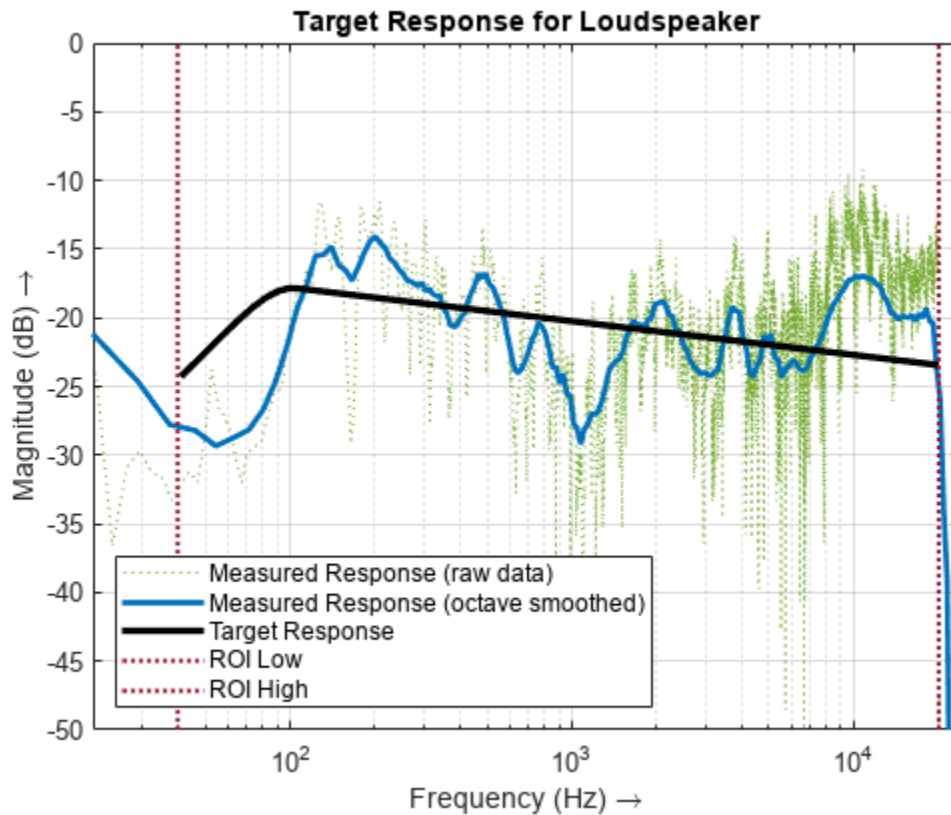
```
% Compute the octave average (only for the region of interest)
[power,cf] = octaveAverage(frequency,db2mag(magnitudeDB),24,ROI,true);
pdb = 20*log10(power) - getMicCalibration(cf);
```

```
% Set the range of the linear fit
lfCutOff = 2.5*ROI(1); % lowest frequency to linearize
hfMaxFit = 0.6*ROI(2); % max frequency to fit for
```

```
% Fit a straight line in the log-frequency domain
linIdx = cf>=lfCutOff & cf<=hfMaxFit;
if license('test','statistics_toolbox')
    pfit = robustfit(log(cf(linIdx)),pdb(linIdx));
else % if this toolbox is not available, use a simple linear regression
    pfit = [ones(nnz(linIdx),1) log(cf(linIdx))] \ pdb(linIdx);
end
targetResp = pfit(1)+pfit(2)*log(cf);

% Roll off the low frequencies, starting slightly above the linear range above
lfcutoff = 1.05*lfCutOff;
idx = cf<lfcutoff;
targetResp(idx) = targetResp(idx) - min(30,ROI(1)/2)*((lfcutoff-cf(idx))/lfcutoff).^2;

% Plot the target response
semilogx(frequency,magnitudeDB,':',Color="#77AC30");
hold on
axis([20 24e3 yrange]);
plot(cfFR,pdbFRmic,Color="#0072BD",LineWidth=2)
plot(cf,targetResp,Color="#000000",LineWidth=2.5);
plot([ROI(1) ROI(1)],yrange,':',Color="#A2142F",LineWidth=1.5);
plot([ROI(2) ROI(2)],yrange,':',Color="#A2142F",LineWidth=1.5);
title('Target Response for Loudspeaker');
legend('Measured Response (raw data)', 'Measured Response (octave smoothed)', ...
       'Target Response', 'ROI Low', 'ROI High', Location='southwest')
xlabel('Frequency (Hz) \rightarrow')
ylabel('Magnitude (dB) \rightarrow')
grid on
hold off
```



The target response (in black) has a slight downward tilt and a roll off for the low frequencies that allows for some boost (10 to 12 dB).

Parametric Filter Overview

The variables being tuned by the optimization algorithm are typical audio parametric EQ parameters: Center Frequency, Filter Bandwidth, and Peak Gain.

Use the response to produce settings for a 12-band parametric filter (10 peak filters and 2 shelf filters).

Use the `designParamEQ` function from Audio Toolbox to design the filter. Use the `lsqnonlin` (Optimization Toolbox) function to perform the fit by tuning the parameters of the EQ bands until the speaker response is as flat as possible.

Before configuring the optimization algorithm, you can look at what a manual filter design looks like for 2 filters. Use the following controls to manually tune the filter parameters and observe the output response using `fvtool`. This allows us to visualize the parameters that the optimization algorithm automatically tunes.

```
gain = [ 4  , ...
        -6.6  ];

centerFreq = [ 3520  , ...
```

```

7230  _____ ]
bandwidth = [ 1920  _____ , ...
              4110  _____ ]

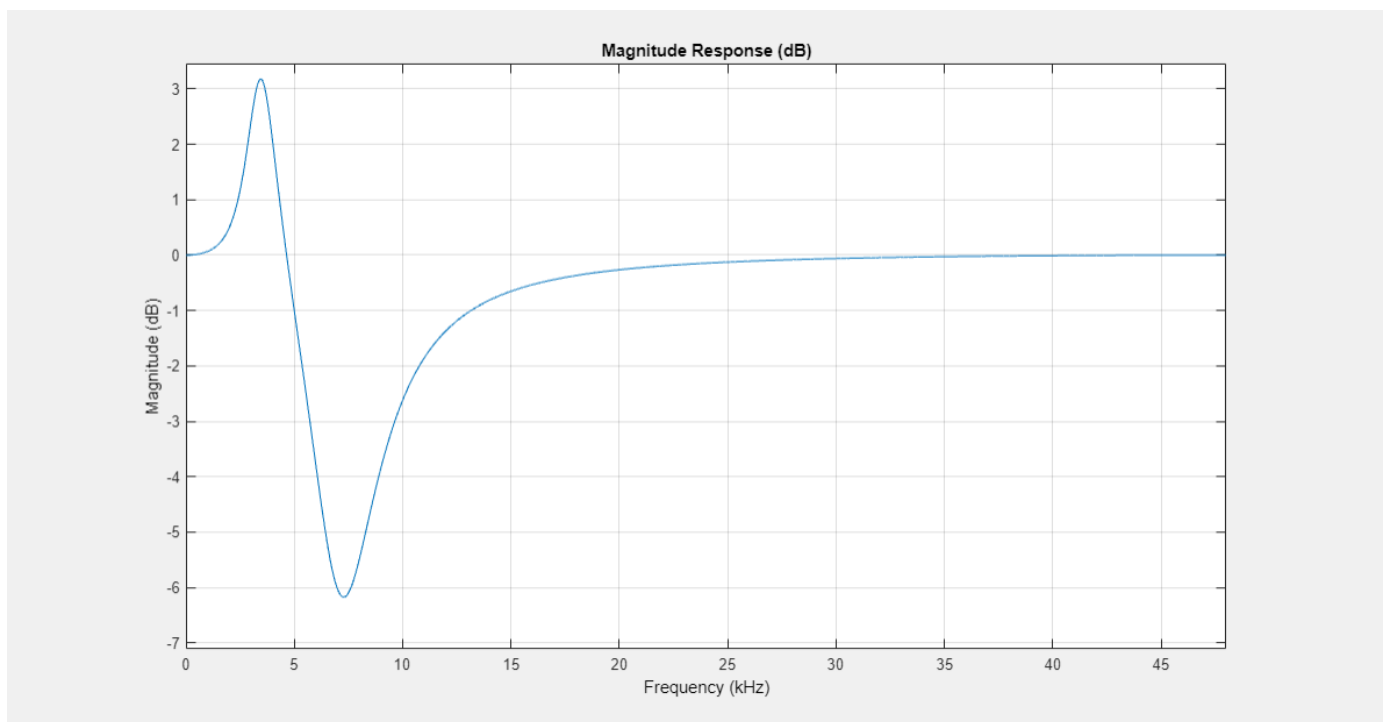
```

Next, generate the filter coefficients using the specified parameters by calling `designParamEQ`:

```
[B,A] = designParamEQ(2,gain,(centerFreq/(Fs/2)),(bandwidth/(Fs/2)),Orientation='row');
```

Visualize the filter design.

```
fvtool([B,A], 'Fs',Fs)
```



Parametric Filter Optimization

To produce the optimized parametric filters, call a helper function that sets starting values and limits for every filter parameter, then calls `lsqnonlin`. The optimizer uses the `eqObjectiveFct` function that computes the response of the given EQ and compares it to the desired response. The `lsqnonlin` optimizer attempts to minimize that error on every iteration.

```

% Run the optimization with a selected number of filters
numFilters = 10+2; % 10 peak filters, one low shelf, and one high shelf
[EQ,outputResp] = eqOptimizer(numFilters,frequency,pdb,targetResp,ROI,Fs);

```

Iteration	Func-count	Resnorm	Norm of step	First-order optimality
0	37	0.162731		0.425
1	74	0.0545925	10	0.524
2	111	0.0289868	20	0.108

3	148	0.0272225	27.8857	0.215
4	185	0.0187429	6.97142	0.188
5	222	0.0154992	0.345245	0.174
6	259	0.014701	13.9428	0.302
7	296	0.0105981	3.48571	0.195
8	333	0.00902063	6.97142	0.0335
9	370	0.00788068	6.97142	0.00907
10	407	0.00635659	13.9428	0.359
11	444	0.00635659	30.7718	0.359
12	481	0.0055785	6.97142	0.0129
13	518	0.00492435	13.9428	0.00426
14	555	0.00452565	27.8857	0.0686
15	592	0.00425527	12.6583	0.0446
16	629	0.00425527	16.7565	0.0446
17	666	0.0040533	4.18912	0.0339
18	703	0.00384572	8.37823	0.00953
19	740	0.00384572	4.72211	0.00953
20	777	0.00374876	1.18053	0.0102
21	814	0.00357736	2.36106	0.00508
22	851	0.00357736	4.41566	0.00508
23	888	0.00351695	1.10391	0.0078
24	925	0.00334974	2.20783	0.012
25	962	0.00334974	4.41566	0.012
26	999	0.00323091	1.10391	0.00812
27	1036	0.00309264	2.20783	0.00466
28	1073	0.00285452	2.20783	0.0865
29	1110	0.00262239	4.01131	0.0678
30	1147	0.00262239	4.33015	0.0678
31	1184	0.00255327	1.08254	0.0425
32	1221	0.00245483	2.40821	0.0338
33	1258	0.00243857	4.33015	0.00585
34	1295	0.00243857	3.19387	0.00585
35	1332	0.0024362	0.2198	0.00533
36	1369	0.00243225	0.798467	0.00404
37	1406	0.00243037	0.798467	0.00285
38	1443	0.00242846	1.59693	0.00203
39	1480	0.00242656	1.59693	0.00116
40	1517	0.00242605	1.59693	0.000984
41	1554	0.00242549	0.864994	0.000378
42	1591	0.00242537	1.26553	0.00064
43	1628	0.00242512	0.795897	0.000308
44	1665	0.00242512	1.5253	0.000308
45	1702	0.00242505	0.381326	0.000439
46	1739	0.00242496	0.762652	0.000113
47	1776	0.00242496	1.5253	0.000113
48	1813	0.00242492	0.381326	0.000196
49	1850	0.00242484	0.762652	9.39e-05
50	1887	0.00242484	1.7062	9.39e-05
51	1924	0.0024248	0.381326	0.000186
52	1961	0.00242474	0.762652	0.000158
53	1998	0.00242474	1.2928	0.000158
54	2035	0.00242471	0.323201	0.000187
55	2072	0.00242466	0.646402	0.000105
56	2109	0.00242466	1.1841	0.000105
57	2146	0.00242464	0.296026	0.000128
58	2183	0.0024246	0.592051	8.08e-05
59	2220	0.00242459	0.973815	0.000327
60	2257	0.00242454	0.243454	0.000198

```
61      2294      0.00242453      0.486907      8.3e-05
62      2331      0.00242452      0.372546      4.24e-05
```

Optimization stopped because the relative sum of squares (r) is changing by less than options.FunctionTolerance = 1.000000e-08.

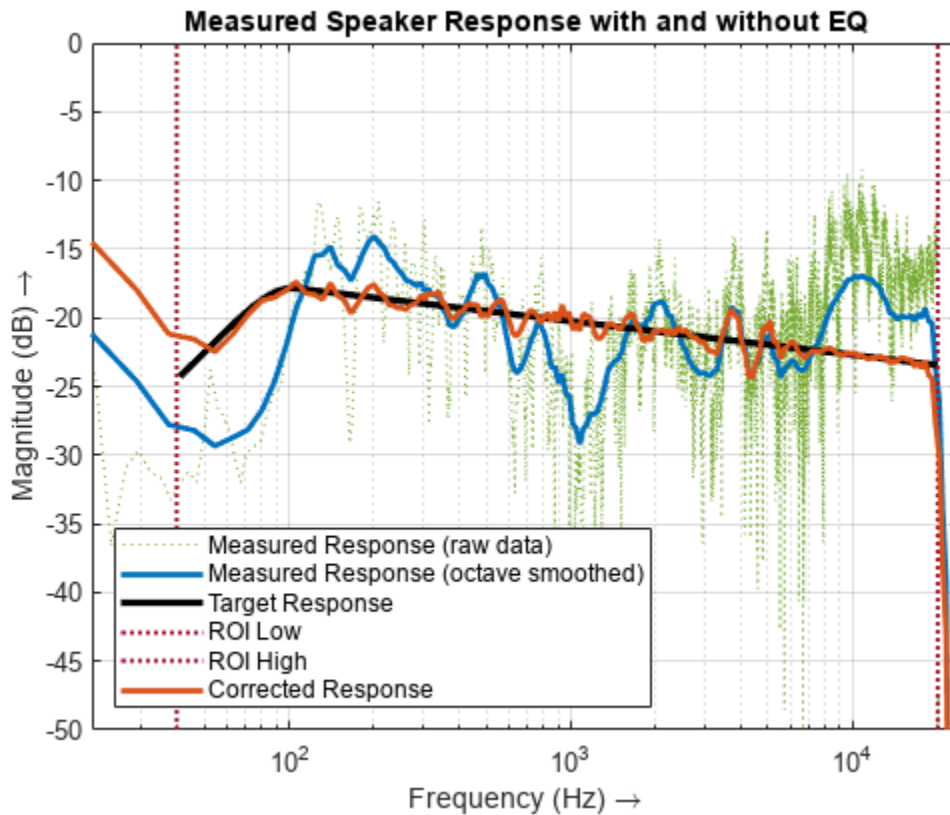
Results

Examine the results. Start by computing the filter frequency response over a larger frequency range.

```
freqzResp = eqFreqz(EQ,frequency,Fs);
pFR = octaveAverage(frequency,abs(freqzResp),24,fullRange,false);
filtRespFR = 20*log10(pFR);
outputRespFR = pdbFRmic + filtRespFR;
```

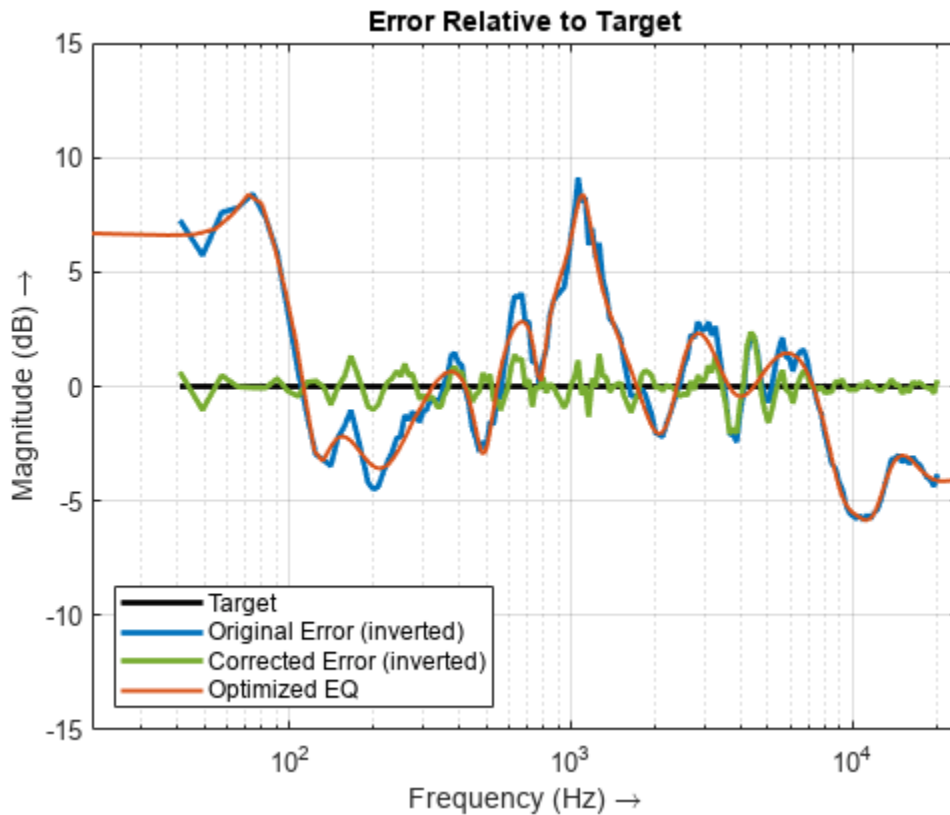
Plot the system response.

```
% Plot the system response
semilogx(frequency,magnitudeDB,':',Color="#77AC30");
hold on
axis([20 24e3 yrange]);
plot(cfFR,pdbFRmic,Color="#0072BD",LineWidth=2)
plot(cf,targetResp,Color="#000000",LineWidth=2.5);
plot([ROI(1) ROI(1)],yrange,':',Color="#A2142F",LineWidth=1.5);
plot([ROI(2) ROI(2)],yrange,':',Color="#A2142F",LineWidth=1.5);
plot(cfFR,outputRespFR,Color="#D95319",LineWidth=2);
title('Measured Speaker Response with and without EQ');
legend('Measured Response (raw data)',...
       'Measured Response (octave smoothed)',...
       'Target Response','ROI Low','ROI High',...
       'Corrected Response',Location='southwest')
xlabel('Frequency (Hz) \rightarrow')
ylabel('Magnitude (dB) \rightarrow')
grid on
hold off
```



Also plot error relative to target, but invert the error curves to make it easier to compare them with the optimized EQ.

```
% Plot error relative to target
errorOld = targetResp - pdb;
errorNew = targetResp - outputResp;
semilogx(cf([1,end]), [0,0], Color="#000000", LineWidth=2);
hold on;
plot(cf, errorOld, Color="#0072BD", LineWidth=2);
plot(cf, errorNew, Color="#77AC30", LineWidth=2);
plot(cfFR, filtRespFR, Color="#D95319", LineWidth=1.5);
hold off; grid on;
axis([20 24e3 -15 15])
title('Error Relative to Target');
xlabel('Frequency (Hz) \rightarrow');
ylabel('Magnitude (dB) \rightarrow');
legend('Target', 'Original Error (inverted)', ...
       'Corrected Error (inverted)', ...
       'Optimized EQ', Location='southwest');
```



Sort peak filters by center frequency and convert to table format.

```
EQpk = EQ(1:end-2,:);
[~,idx] = sort(EQpk(:,3));
EQ(1:end-2,:) = EQpk(idx,:);
```

```
% Create a table with all the EQ settings
filterType = [repmat("PK",numFilters-2,1);"LS";"HS"];
EQt = table(filterType,EQ(:,3),EQ(:,1),EQ(:,2),VariableNames=...
            {'Type','Frequency (Hz)','Gain (dB)','Q/S'})
```

EQt=12x4 table

Type	Frequency (Hz)	Gain (dB)	Q/S
"PK"	72.52	3.9074	2.1516
"PK"	122.84	-5.8932	3.0223
"PK"	211.61	-9.727	0.92132
"PK"	492.18	-6.61	3.7217
"PK"	779	-4.2525	7.8339
"PK"	1102.3	5.1899	4.358
"PK"	2127.6	-15.942	1.1218
"PK"	4842.4	8.0688	0.48624
"PK"	9175.5	-6.1896	2.0737
"PK"	11710	-6.4538	2.1912
"LS"	3194.1	6.7385	5
"HS"	15580	-3.5357	3.4374

Compute an overall gain that avoids any tones increasing in amplitude. This is not a guarantee that signals cannot clip but is generally more than sufficient in practice.

```
gainDB = -max(filtRespFR) - .1;
```

Instantiate a `multibandParametricEQ` object with the EQ settings. You can use this object to visualize the filter, create an audio plugin, or load either the object or plugin into the Audio Test Bench.

`% Create EQ object`

```
mbpeq = multibandParametricEQ(HasLowShelfFilter=true, HasHighShelfFilter=true, ...
    NumEQBands=numFilters-2, EQOrder=2, SampleRate=Fs, ...
    Frequencies=EQ(1:end-2,3)', QualityFactors=EQ(1:end-2,2)', PeakGain=...
    LowShelfCutoff=EQ(end-1,3), LowShelfSlope=EQ(end-1,2), LowShelfGain=...
    HighShelfCutoff=EQ(end,3), HighShelfSlope=EQ(end,2), HighShelfGain=...);
```

Produce output files to subjectively evaluate the results, either with headphones or in the actual listening room. The IR is included for headphone evaluation but should be omitted when testing in the actual room (which produces that response itself).

```
[rock,fileFs] = audioread('RockDrums-44p1-stereo-11secs.mp3');
```

`% Resample the test file to match the sample rate of the IR measurement`

```
rock = resample(rock,Fs,fileFs,100);
```

`% Convolve the IR with the test audio. This will simulate the effect of the room when evaluating the EQ using headphones.`

```
rockIR = conv(rock(:,1),ir);
```

```
rockIR(:,2) = conv(rock(:,2),ir);
```

```
rockIR = rockIR*.97/max(abs(rockIR),[],'all');
```

```
audiowrite('RockDrums-with-IR.wav',rockIR,Fs,Comment=...
```

```
    'Convolution of rock drums with impulse response (for simulated evaluation on headphones)');
```

Type `audioTestBench(mbpeq)` at the command prompt to try the plugin in the Audio Test Bench app.

In the Audio Test Bench, click the "Audio File Reader" button with the gear icon and select 'RockDrums-with-IR.wav' if using headphones, or simply use 'RockDrums-44p1-stereo-11secs.mp3' if playing back over the same system that was used to measure the impulse response. With the Audio Test Bench, you can toggle the EQ on and off, and you can even further tune the EQ settings to your liking.

You can also process files with the EQ to listen outside of the Audio Test Bench.

`% Apply the EQ to the original file (for use in the measured room).`

`% Use the gain that was computed to avoid clipping.`

```
rockEQ = db2mag(gainDB)*mbpeq(rock);
```

```
reset(mbpeq); % reset the EQ before processing another file
```

```
audiowrite('RockDrums-with-Correction-only.wav',rockEQ,Fs,Comment=...
```

```
    ['Convolution of rock drums with correction (for listening '...'
```

```
    'in the same environment the IR was measured in)']);
```

`% Apply the EQ to the IR-processed file (for use with headphones).`

`% Use an output level that avoids clipping.`

```
rockIREQ = mbpeq(rockIR);
```

```
reset(mbpeq); % reset the EQ so it is free to process another file
```

```
rockIREQ = rockIREQ/max(abs(rockIREQ),[],'all')*.97;
```

```
audiowrite('RockDrums-with-IR-and-Correction.wav',rockIREQ,Fs,Comment=...  
    ['Convolution of rock drums with impulse response and '...  
    'correction (IR simulation for evaluation over headphones)']);
```

Alternatively, to apply the EQ to any playback on a selected device, Windows users can export the EQ settings in a Room EQ Wizard (REW) compatible format and load it in Equalizer APO.

```
eqExport2APO("myroomeq.txt",EQ,gainDB);  
type("myroomeq.txt")
```

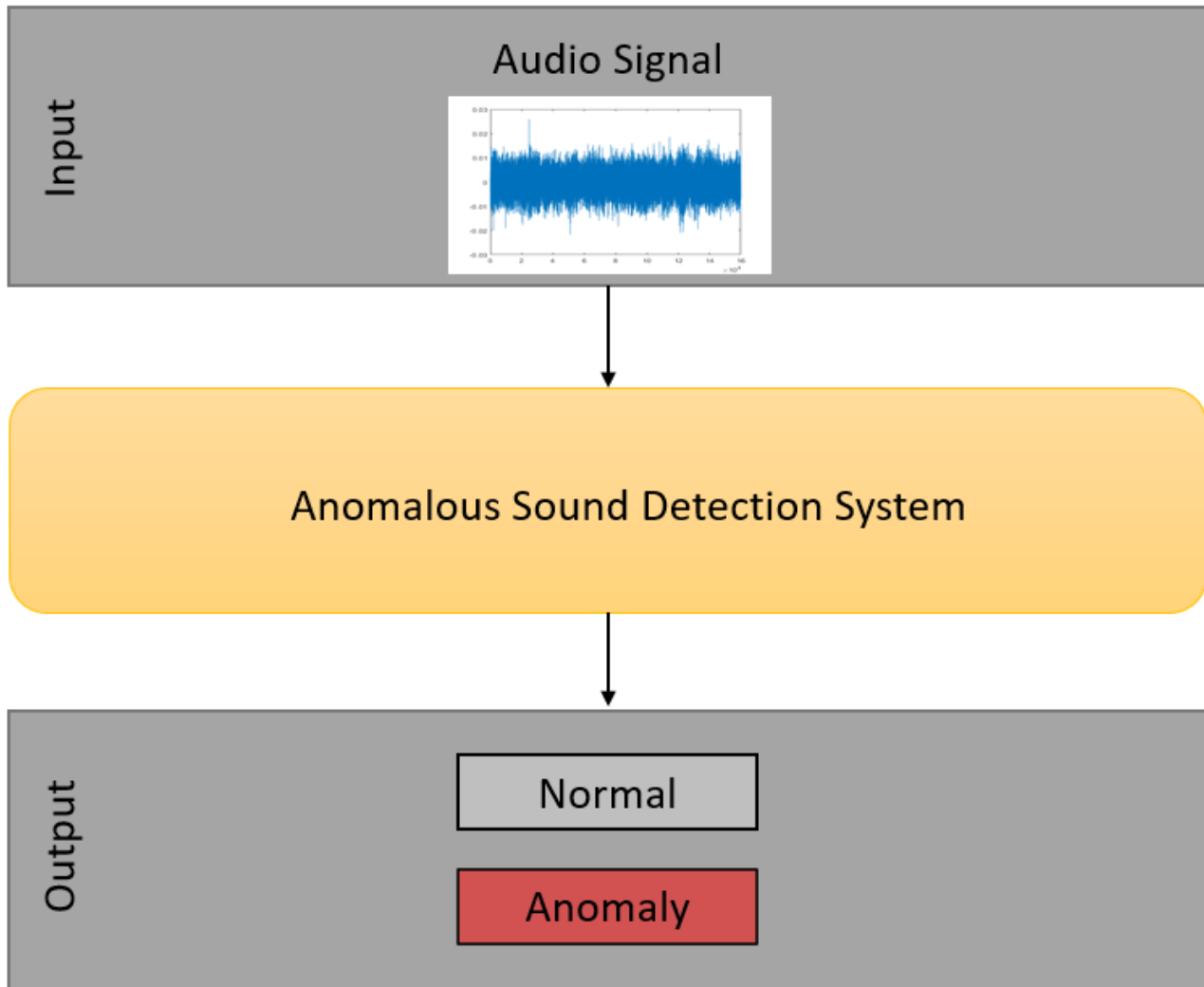
Filter Settings file

MATLAB Filter Export
Dated: 21-Dec-2022 18:26:34

Notes: Generated using MATLAB example

```
Equalizer: Generic  
Preamp: -8.5 dB  
Filter: ON PK Fc 72.52 Hz Gain 3.91 dB Q 2.15162  
Filter: ON PK Fc 122.84 Hz Gain -5.89 dB Q 3.02225  
Filter: ON PK Fc 211.61 Hz Gain -9.73 dB Q 0.92132  
Filter: ON PK Fc 492.18 Hz Gain -6.61 dB Q 3.72172  
Filter: ON PK Fc 779.00 Hz Gain -4.25 dB Q 7.83389  
Filter: ON PK Fc 1102.35 Hz Gain 5.19 dB Q 4.35797  
Filter: ON PK Fc 2127.60 Hz Gain -15.94 dB Q 1.12182  
Filter: ON PK Fc 4842.40 Hz Gain 8.07 dB Q 0.48624  
Filter: ON PK Fc 9175.50 Hz Gain -6.19 dB Q 2.07367  
Filter: ON PK Fc 11710.49 Hz Gain -6.45 dB Q 2.19118  
Filter: ON LS Fc 3194.10 Hz Gain 6.74 dB Q 1.89623  
Filter: ON HS Fc 15580.23 Hz Gain -3.54 dB Q 1.34551
```

Audio-Based Anomaly Detection for Machine Health Monitoring



This example shows how to design an autoencoder neural network to perform anomaly detection for machine sounds using unsupervised learning. In this example you will download and process the data using a log-mel spectrogram, design and train an autoencoder network, and make out-of-sample predictions by applying a statistical model to the trained network output.

Audio-based anomaly detection is the process of identifying whether the sound generated by an object is abnormal. This is applicable to the automatic detection of industrial component failures, as a machine that emits an abnormal sound is likely malfunctioning.

The problem of classifying sounds as either normal or abnormal can be viewed as a standard supervised learning task, where a model is trained on samples of both sound types and learns to discriminate between them. However, in practice, a data set of abnormal sounds is generally not available because machine malfunctions do not occur frequently enough or for long enough duration

to be properly recorded. Also, it would be impossible to create a data set representative of every type of anomaly, as a machine could malfunction for a diverse set of reasons.

Autoencoders are useful for anomaly detection tasks because they train solely on the normal samples. Autoencoder networks perform the unsupervised learning task of finding both a low-dimensional encoding of the input as well as a rule to accurately reconstruct the input from its low-dimensional representation. This forces the autoencoder to learn a process specifically for compressing and decompressing normal samples. The motivating principle is that when an abnormal sample is fed into the autoencoder, the reconstruction error will be much larger than expected from the training set because the signal compression and decompression scheme learned by the network is only expected to work well for normal samples. To make predictions on unseen samples, an error threshold is picked based off the expected distribution of reconstruction errors for normal samples, and any input with an error larger than the threshold is classified as an anomaly.

In this example, the autoencoder first passes the input through an encoding section of fully-connected layers using a number of nodes on the same order of magnitude as the input dimension. The data then feeds into a bottleneck layer with a number of nodes much smaller than the input size which forces the network to compress the input signal into the lower-dimensional representation. This compressed representation feeds into a decoding section that generally mirrors the same architecture as the encoder section in order to recreate the input signal. Lastly, the decoder output is passed into a final output layer with the same number of dimensions as the input. The network loss is taken as the regression error between the original input and the reconstructed signal.

Download Data

This example applies to the second task of the Detection and Classification of Acoustic Scenes and Events (DCASE) 2022 challenge [1] on page 1-1009. The example uses a subset of the public data set from Sound Dataset for Malfunctioning Industrial Machine Investigation and Inspection [2] on page 1-1009 to train and evaluate the autoencoder. It implements ideas from the preprocessing steps and network designs of both the autoencoder baseline system in [1] on page 1-1009 and the proposed network in [2] on page 1-1009 and uses the performance metrics devised in [1] on page 1-1009 to analyze the testing results.

Download a subset of the data set in [2] on page 1-1009 that contains recorded audio files of 4 different fan types, labelled by ID number. There are both normal and abnormal recordings for each fan type. These files contain 1 channel sampled at 16 kHz and are 10 seconds long. The samples are recordings of operating fans with background noise with a signal to noise ratio of 6 dB. A full explanation of the data collection process is available in [2] on page 1-1009.

```
dataFolder = tempdir;  
dataset = fullfile(dataFolder, "fan6db");  
supportFileLoc = "mimii/mono/fan6db.zip";  
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", supportFileLoc);  
unzip(downloadFolder, dataFolder)
```

Investigate Data

To briefly examine the data set and the differences between the normal and abnormal recordings, select one recording of each type from the ID 00 fan data set and play the first two seconds over your speaker.

```
[normalSample, fs] = audioread(fullfile(dataset, "id_00", "normal_00", "00000000.wav"));  
abnormalSample = audioread(fullfile(dataset, "id_00", "abnormal_00", "00000000.wav"));  
  
numSamples = 10*fs;
```



```
sound(normalSample(1:numSamples/5), fs)
pause(3)
sound(abnormalSample(1:numSamples/5), fs)
```

Both recordings are dominated by a single tone, and this tone is clearly higher pitched in the abnormal sample.

Preprocess Data

You can optionally set the `speedUp` flag to `true` to reduce the size of the data set used in the example. If you set this to `true` you can quickly verify that the script runs as expected, but the results will be skewed.

```
speedUp =  ;
```

Separate the data set into two `audioDatastore` objects, one with the normal samples and one with the abnormal samples. Since the autoencoder only trains on the normal samples, hold out the abnormal samples to be included in the test set.

```
ads = audioDatastore(dataset, ...
    IncludeSubfolders=true, ...
    LabelSource="foldernames", ...
    FileExtensions=".wav");

normalLabels = categorical(["normal_00", "normal_02", "normal_04", "normal_06"]);
abnormalLabels = categorical(["abnormal_00", "abnormal_02", "abnormal_04", "abnormal_06"]);

isNormal = ismember(ads.Labels, normalLabels);
isAbnormal = ~isNormal;
adsNormal = subset(ads, isNormal);
adsTestAbnormal = subset(ads, isAbnormal);
rng(3);
if speedUp
    c = cvpartition(adsTestAbnormal.Labels, kFold=8, Stratify=true);
    adsTestAbnormal = subset(adsTestAbnormal, c.test(1));
end
```

Divide the normal samples into training, validation, and test sets, stratified by ID number. Then concatenate the normal test set with the abnormal samples to form the full test set.

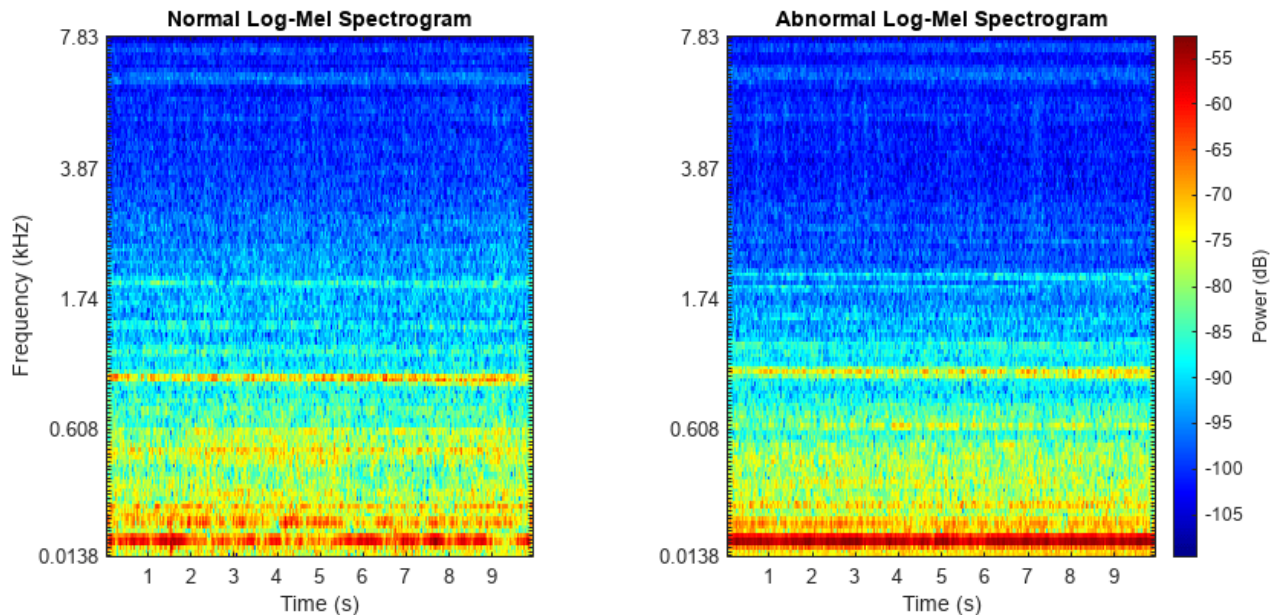
```
c = cvpartition(adsNormal.Labels, kFold=8, Stratify=true);
if speedUp
    trainInd = c.test(3);
else
    trainInd = ~boolean(c.test(1)+c.test(2));
end
valInd = c.test(1);
testInd = c.test(2);

adsTrain = subset(adsNormal, trainInd);
adsVal = subset(adsNormal, valInd);
adsTestNormal = subset(adsNormal, testInd);
```

Transform each of the datastores by applying an STFT with frame length of 64 ms and hop length of 32 ms, find the log-mel energies for 128 frequency bands, and then concatenate these frames into overlapping, consecutive groups of 5 to form a context window. It is common to use log-mel energies

as inputs to audio deep learning tasks as they represent the spectrum of tones on a scale similar to how humans perceive sound. Visualize the log-mel spectrograms of the two clips played previously using the `plotLogMelSpect` supporting function.

```
plotLogMelSpect(normalSample,abnormalSample);
```



Use the `processData` supporting function to perform the data transformation.

```
tdsTrain = transform(adsTrain,@processData);
tdsVal = transform(adsVal,@processData);
tdsTestNormal = transform(adsTestNormal,@processData);
tdsTestAbnormal = transform(adsTestAbnormal,@processData);
```

Read the data into arrays where each column represents an input sample. Do this in parallel if you have enabled Parallel Computing Toolbox™. Then combine the normal test set and abnormal data set into the full test set, and label the samples accordingly.

```
trainingData = readall(tdsTrain,UseParallel=canUseParallelPool);
valData = readall(tdsVal,UseParallel=canUseParallelPool);
```

```
normalTestData = readall(tdsTestNormal,UseParallel=canUseParallelPool);
abnormalTestData = readall(tdsTestAbnormal,UseParallel=canUseParallelPool);
testLabels = categorical([zeros(length(adsTestNormal.Labels),1);ones(length(adsTestAbnormal.Labels),1)]);
testData = [normalTestData;abnormalTestData];
```

Network Architecture

The encoder section consists of 2 fully connected layers with output sizes of 128. The bottleneck layer constrains the network to an 8-dimensional representation of the original 640-dimensional input. The decoder section mirrors the encoder architecture as the input is reconstructed and fed into the output layer. Use half-mean-squared-error as the loss function to train the network and quantify the reconstruction error.

```
layers = [
```

```

featureInputLayer(640)

fullyConnectedLayer(128,Name="Encoder1")
batchNormalizationLayer
reluLayer

fullyConnectedLayer(128,Name="Encoder2")
batchNormalizationLayer
reluLayer

fullyConnectedLayer(8,Name="Bottleneck")
batchNormalizationLayer
reluLayer

fullyConnectedLayer(128,Name="Decoder1")
batchNormalizationLayer
reluLayer

fullyConnectedLayer(128,Name="Decoder2")
batchNormalizationLayer
reluLayer

fullyConnectedLayer(640,Name="Output")
regressionLayer];

```

Train Network

Train the network using an ADAM optimizer for 40 epochs. Shuffle the mini-batches each epoch, and set the ExecutionEnvironment field to "auto" so that a GPU is used instead of the CPU if available. If using a GPU with limited memory, you may need to decrease the value of the miniBatchSize field. The training parameter settings were found empirically to optimize convergence speed. This may take 10-15 minutes depending on your hardware.

```

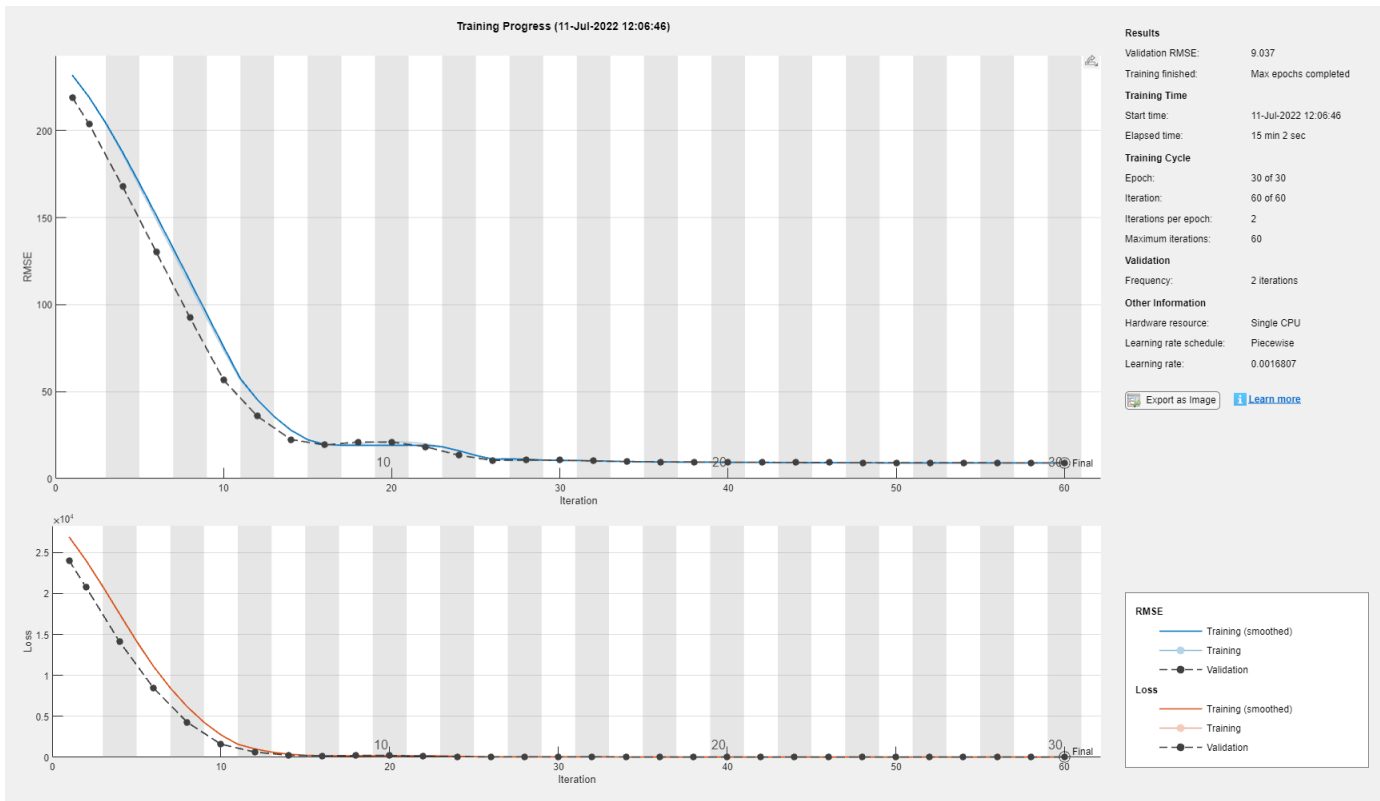
batchSize = length(trainingData)/2;
if speedUp
    batchSize = 2*batchSize;
end

options = trainingOptions("adam", ...
    MaxEpochs=30, ...
    InitialLearnRate=1e-2, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropPeriod=5, ...
    LearnRateDropFactor=.7, ...
    GradientDecayFactor=.8, ...
    miniBatchSize=batchSize, ...
    Shuffle="every-epoch", ...
    ExecutionEnvironment="auto", ...
    ValidationData={valData,valData}, ...
    ValidationFrequency=2, ...
    Verbose=0, ...
    Plots="training-progress");

```

trainingData is both the input and the target output as the network attempts to regress the training data on itself with the low-dimensional encoding constraint. Your results should look similar to the training plots below.

```
[net,info] = trainNetwork(trainingData,trainingData,layers,options);
```



Evaluate Performance

For each input to the network, the autoencoder outputs an attempted reconstruction. However, each network input is only one context window from a larger audio sample. For each of these network inputs, the error is defined as the squared L-2 norm of the difference between the original input and the network output. To calculate a decision metric for each entire audio sample, the errors for each context window associated with that audio sample are added together, and this sum is divided by the product of the network input dimension and the number of context groups per audio sample. For an audio sample X , the decision function metric is denoted $A(X)$ and defined $A(X) = \sum_{i=1}^n \frac{\|f(x_i) - x_i\|^2}{n * \dim(x_i)}$

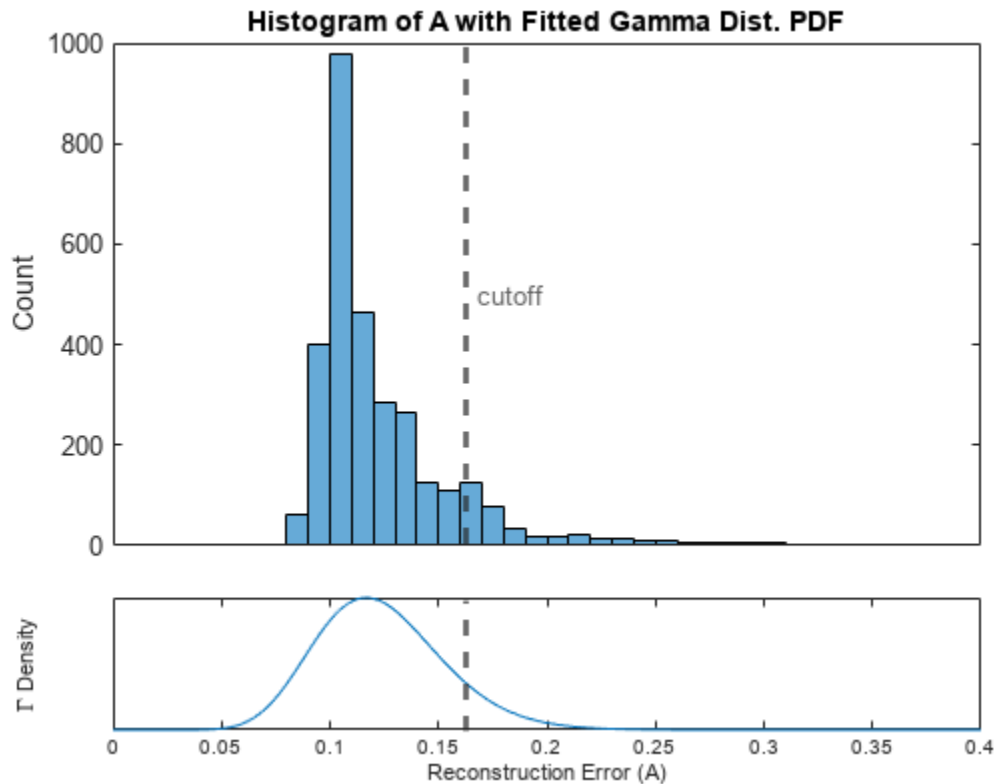
where n is the number of context groups per sample, x_i is the i^{th} context group constructed from X , and $f(x_i)$ is the network output for x_i . $A(X)$ represents the mean squared reconstruction error across each vector component of all context windows associated with an audio sample X .

For each input X , $A(X)$ can also be interpreted as a relative measure of the network's confidence that X is abnormal, with higher values indicating larger confidence. To deploy this model and make predictions on new data, you must select a decision boundary on the values of A to separate positive and negative predictions. Model $A(X)$ for normal samples as a gamma distribution. Gamma distributions are commonly used to model autoencoder reconstruction errors since the errors are usually skewed right with a heavy tail, which is the natural shape of a gamma distribution. In this example, the decision boundary is selected as the point that corresponds to an expected false positive rate (FPR) $p = 0.1$. This decision boundary attempts to capture all truly abnormal samples while tolerating the expectation that 10% of normal samples will be falsely predicted as abnormal. You can choose a specific value of p to fit your individual system constraints.

$p = .1;$

Compute the values of A over the training set and store them in the variable A_train using the helper function `getScore`. Then solve for the maximum likelihood estimate for the gamma distribution parameters, select the cutoff point from the inverse gamma cumulative distribution function, and plot the fitted distribution with the histogram of A using the `getCutoff` helper function.

```
trainRecons = predict(net,trainingData,MiniBatchSize=length(trainingData));
A_train = getScore(trainingData,trainRecons);
cutoff = getCutoff(A_train,p);
```

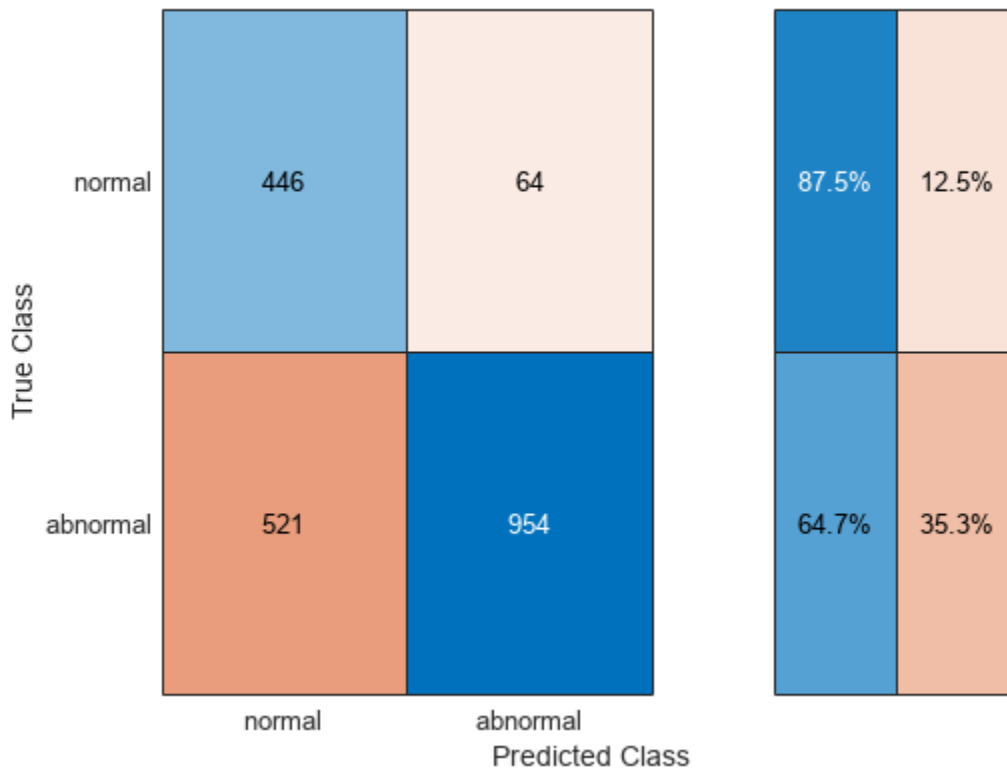


Verify that this cutoff point roughly corresponds to an FPR of 0.1 on the training set:

```
sum(A_train > cutoff) / length(A_train)
ans = 0.1086
```

Test the classification accuracy of this system with the chosen cutoff point on the holdout test set.

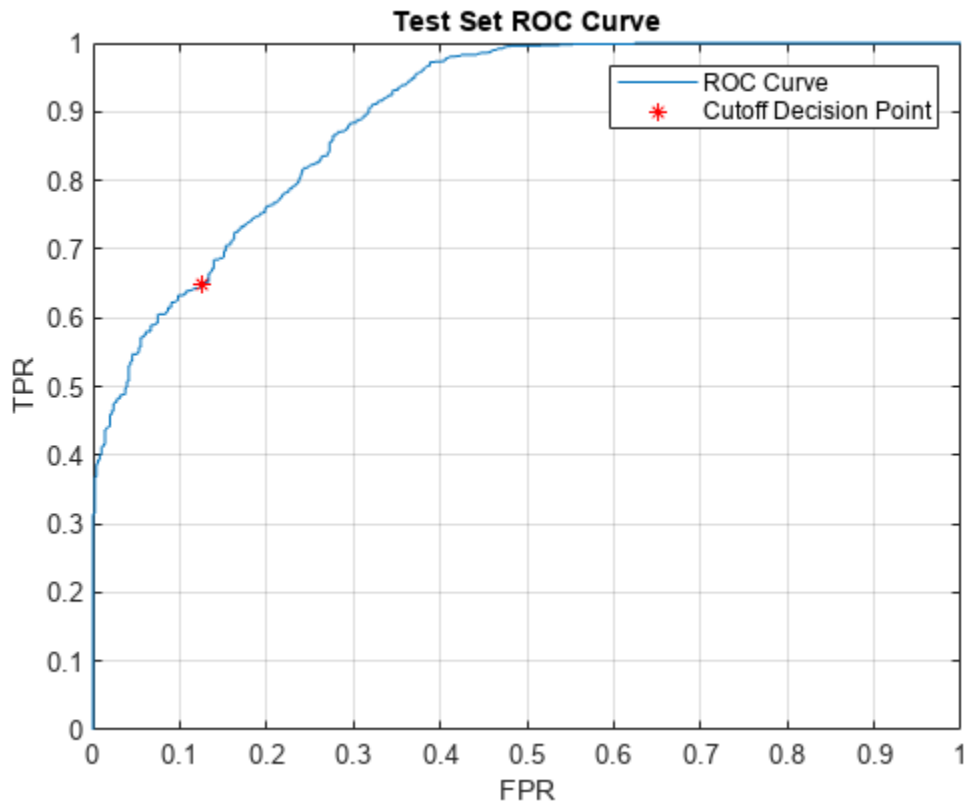
```
testRecons = predict(net,testData,MiniBatchSize=length(testData));
A_test = getScore(testData,testRecons);
testPreds = categorical(A_test > cutoff,[false,true],["normal","abnormal"]);
figure
cm = confusionchart(testLabels,testPreds);
cm.RowSummary="row-normalized";
```



Using this cutoff point, the model achieves a true positive rate (TPR) of 0.647 at the cost of an FPR of 0.125.

To evaluate the accuracy of the network over a range of decision boundaries, measure the overall performance on the test set by the area under the receiver operating characteristic curve (AUC). Use both the full AUC and the partial AUC (pAUC) to analyze the network performance. pAUC is the AUC on the subdomain where the FPR is on the interval $[0, p]$ divided by the maximum possible area in the interval, which is p . It is important to consider pAUC since anomaly detection systems need to be able to achieve high TPR while keeping the FPR to a minimum, as a system with frequent false alarms is untrustworthy and unusable. Compute the AUC using the `perfcurve` function from Statistics and Machine Learning Toolbox™.

```
[X,Y,T,AUC] = perfcurve(testLabels,A_test,categorical("abnormal"));
[~,cutoffIdx] = min(abs(T-cutoff));
figure
plot(X,Y);
xlabel("FPR");
ylabel("TPR");
title("Test Set ROC Curve");
hold on
plot(X(cutoffIdx),Y(cutoffIdx),'r*');
hold off
legend("ROC Curve","Cutoff Decision Point");
grid on
```



AUC

```
AUC = single
0.8957
```

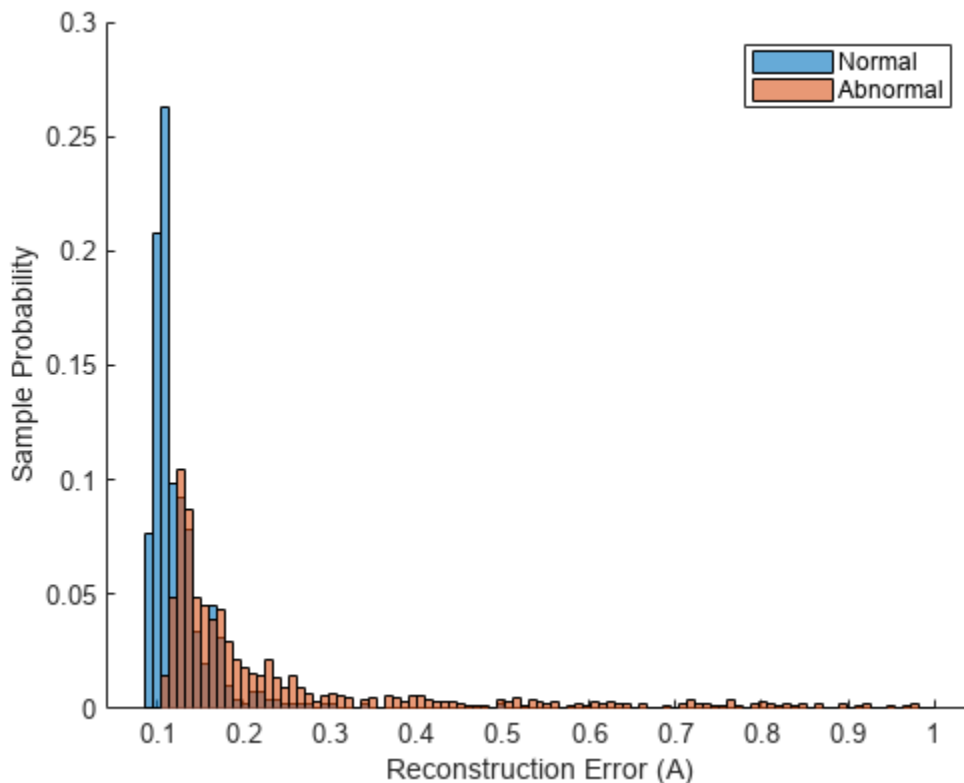
To calculate the pAUC, approximate the area under the curve in the first tenth of the FPR domain using `trapz`. For reference, the expected value of the pAUC of a random classifier is 0.05.

```
pX = X(X <= p);
pY = Y(X <= p);
pAUC = trapz(pX,pY)/p
```

```
pAUC = single
0.5161
```

The network separates the normal and abnormal test samples fairly well and is able to learn a single encoding across multiple fan IDs. Visualize the difference in reconstruction errors between the normal and abnormal groups by their histograms.

```
figure
hold on
edges = linspace(min(A_test),1,100);
histogram(A_test(testLabels == categorical("normal")),edges,Normalization="probability");
histogram(A_test(testLabels == categorical("abnormal")),edges,Normalization="probability");
ylabel("Sample Probability");
xlabel("Reconstruction Error (A)");
legend("Normal", "Abnormal");
```



Although there is some overlap, the distribution of reconstruction errors for the abnormal samples is offset further to the right and contains a much heavier tail than the distribution of reconstruction errors over the normal samples.

Lastly, evaluate the model's performance on each fan ID individually to reveal any imbalance between the fan types and check if the model is able to predict universally well over all IDs.

```

IDs = [0;2;4;6];
AUCs = zeros(4,1);
pAUCs = AUCs;
A_testNormal = A_test(1:sum(testInd));
A_testAbnormal = A_test(sum(testInd)+1:end);
for i = 1:4
    normalMask = adsTestNormal.Labels == normalLabels(i);
    abnormalMask = adsTestAbnormal.Labels == abnormalLabels(i);
    A_testByID = [A_testNormal(normalMask) A_testAbnormal(abnormalMask)];
    testLabelsByID = [adsTestNormal.Labels(normalMask);adsTestAbnormal.Labels(abnormalMask)];
    [X_ID,Y_ID,T_ID,AUC_ID] = perfcurve(testLabelsByID,A_testByID,abnormalLabels(i));
    AUCs(i) = AUC_ID;
    pX_ID = X_ID(X_ID <= p);
    pY_ID = Y_ID(X_ID <= p);
    pAUCs(i) = trapz(pX_ID,pY_ID)/p;
end
disp(table(IDs,AUCs,pAUCs));

```

IDs	AUCs	pAUCs
_____	_____	_____

0	0.72147	0.19134
2	0.97739	0.73454
4	0.8761	0.42547
6	0.96017	0.6742

The results show the model performance significantly varies by fan type. This result is important to note as this network is relatively small and simple compared to the top performing DCASE challenge submissions in [3] on page 1-1009. To generalize better across fan types and to different domains, a more complex model is needed. However, if you know the exact fan type that you are deploying an anomaly detector for, a very light-weight model like the one in this example may suffice.

Supporting Functions

```
function plotLogMelSpect(normalSample,abnormalSample)
%PLOTLOGMELSPECT plots the log-mel spectrogram of the normal and abnormal
% plotLogMelSpect(normalSample,abnormalSample) plots the log-mel
% spectrogram of the two inputs side by side, with parameters consistent
% with the data preprocessing transformation used to prepare the signals
% to be fed into the autoencoder.
f = figure;
f.Position(3) = 900;
samples = {normalSample,abnormalSample};
fs = 16e3;
winDur = 64e-3;
winLen = winDur * fs;
numMelBands = 128;
tiledlayout(1,2)
for i = 1:2
    nexttile
    x = samples{i};
    melSpectrogram(x, fs, Window=hamming(winLen, "periodic"), FFTLength=winLen, OverlapLength=winLen/2,
    xticks(1:10);
    xticklabels(string(1:10));
    colormap("jet");
    if i == 2
        cbar = colorbar;
        cbar.Label.String = "Power (dB)";
        title("Abnormal Log-Mel Spectrogram");
        ylabel([]);
    else
        colorbar off
        title("Normal Log-Mel Spectrogram");
    end
end
end
function features = processData(x)
%PROCESSDATA transforms an audio file input x into the autoencoder network
%input format
% features = processData(x) takes the STFT of audio data x, transforms
% the STFT into the log-mel spectrogram, and then constructs context
% groups of consecutive mel-spectrogram frames. The function returns the
% features as a numContextGroupsPerSample-by-contextGroupSize matrix. For
% this data set, numContextGroupsPerSample = 309 and contextGroupSize =
% 640 = 128*5 (since there are 128 mel bands per frame and 5 frames are
% concatenated for each context group)

fs = 16e3;
```

```
winDur = 64e-3;
winLen = winDur*fs;
numMelBands = 128;
afe = audioFeatureExtractor(...
    Window=hamming(winLen,"periodic"), ...
    FFTLength=winLen, ...
    OverlapLength=winLen/2, ...
    SampleRate=fs, ...
    melSpectrum=true);

setExtractorParameters(afe,"melSpectrum",numBands=numMelBands);

% Zero pad
numSamples = length(x);
numPad = winLen - mod(numSamples,winLen);
numToPadFront = floor(numPad/2);
numToPadBack = ceil(numPad/2);

xPadded = [zeros(numToPadFront,1,like=x);x;zeros(numToPadBack,1,like=x)];
% Extract
features = extract(afe,xPadded);
features = {log10(features)};
features = cellfun(@groupSTFT,features,UniformOutput=false);
features = vertcat(features{:});
end

function groups = groupSTFT(x)
%GROUPSTFT transforms an STFT x into context groups of size 5
% groups = groupSTFT(x) transforms the STFT x by grouping each STFT frame
% with the following 4 frames to form context groups of size 5. This
% creates multiple network inputs out of each audio sample, each of size
% contextLen*numMelBands = 5*128 = 640. Each of these context groups are
% treated as individual 640-dimensional vectors for the purpose of the
% autoencoder.
contextLen = 5;
numMelBands = 128;
x_flat = reshape(x',1,[]);
groups = buffer(x_flat,contextLen*numMelBands,numMelBands*(contextLen-1),"nodelay");
end

function A = getScore(data,preds)
%GETSCORE returns the reconstruction error for each sample in data
% A = getScore(data,preds) returns A(X) for each X in the set of samples
% transformed into network input data.
err = sum((preds-data).^2,2);
numSTFTFrames = 313;
contextWin = 5;
numMelFilters = 128;
numContextGroupsPerSample = numSTFTFrames-contextWin+1;
numSamples = length(err)/numContextGroupsPerSample;
A_total = reshape(err,[numContextGroupsPerSample,numSamples]); %Each column contains reconstruct
A = sum(A_total)/(numMelFilters*contextWin*numSTFTFrames); %Each entry is a reconstruction error
end

function cutoff = getCutoff(A,p)
%GETCUTOFF fits a gamma distribution to A and returns the cutoff as the inverse cdf of 1-p
% cutoff = getCutoff(A,p) fits a gamma distribution to the reconstruction
% error array A, solves for the cutoff point as the inverse gamma cdf
```

```

% evaluated at 1-p, and plots the fitted distribution along with the
% histogram of A and the calculated cutoff point. A is expected as
% one-by-numSamples array where numSamples is the number of audio samples
% used to compute the reconstruction error values of A.
gammaParams = gamfit(A);
a = gammaParams(1);
b = gammaParams(2);
cutoff = gaminv(1-p,a,b);
figure
ax1 = subplot(4,1,1:3);
histogram(A);
xticks([]);
title("Histogram of A with Fitted Gamma Dist. PDF");
ylTop = ylabel("Count");
xline(cutoff,"--",LineWidth=2,Label="cutoff",LabelOrientation="horizontal",LabelVerticalAlignment="top");
ax2 = subplot(4,1,4);
t = linspace(0, max(A), 1000);
y = gampdf(t,a,b);
plot(t,y);
xline(cutoff,"--",LineWidth=2);
ylBottom = ylabel("\Gamma Density");
yticks([]);
linkaxes([ax1 ax2],"x");
ylBottom.Position(1) = ylTop.Position(1);
xlabel("Reconstruction Error (A)");
xlim([0 .4]);
ylBottom.Position(1) = ylTop.Position(1);
end

```

References

[1] "Unsupervised anomalous sound detection for machine condition monitoring applying domain generalization techniques," *DCASE 2022*. [Online]. Available: <https://dcase.community/challenge2022/task-unsupervised-anomalous-sound-detection-for-machine-condition-monitoring>. [Accessed: 08-Jun-2022].

[2] "Purohit, Harsh, Tanabe, Ryo, Ichige, Kenji, Endo, Takashi, Nikaido, Yuki, Suefusa, Kaori, & Kawaguchi, Yohei. (2019). MIMII Dataset: Sound Dataset for Malfunctioning Industrial Machine Investigation and Inspection (public 1.0) [Data set]. 4th Workshop on Detection and Classification of Acoustic Scenes and Events (DCASE 2019 Workshop), New York, USA. Zenodo. <https://doi.org/10.5281/zenodo.3384388>. Dataset is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License available at <https://creativecommons.org/licenses/by-sa/4.0/>

[3] "Unsupervised detection of anomalous sounds for Machine Condition Monitoring," *DCASE 2020*. [Online]. Available: <https://dcase.community/challenge2020/task-unsupervised-detection-of-anomalous-sounds-results#Giri2020>. [Accessed: 08-Jun-2022].

Use Datasets to Manage Audio Data Sets

Deep learning and machine learning models are popular for processing audio signals for various tasks. Training these models requires working with large data sets containing both audio data and labeling information. For example, when training a model to identify spoken commands, the data can be a collection of audio files and the labels in this case are the ground truth commands for each file. Datasets are useful for working with large collections of data, and the `audioDataset` object allows you to manage collections of audio files.

This example shows you how to use datasets to manage three different audio data sets. The first data set uses the names of the folders containing the audio files as labels, the second data set uses the file names as labels, and the third data set contains labels in a metadata file. You can then use these datasets to train machine learning or deep learning models on the audio data.

Data With Folder Name Labels

The Google Speech Commands data set [1] on page 1-1013 contains files with spoken command words stored in folders whose names are the word labels. Download and extract the data set.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "google_speech.zip");
dataFolder = tempdir;
unzip(downloadFolder, dataFolder)
dataset = fullfile(dataFolder, "google_speech");
```

Create an `audioDataset` that points to the training data.

```
ads = audioDataset(fullfile(dataset, "train"), IncludeSubfolders=true);
```

Extract the labels for each file from the folder names using the `folders2labels` function. Use `countlabels` to view the distribution of labels.

```
labels = folders2labels(ads);
countlabels(labels)
```

```
ans=30x3 table
   Label      Count      Percent
   _____  _____  _____
   bed         1340         2.6229
   bird        1411         2.7619
   cat         1399         2.7384
   dog         1396         2.7325
   down        1842         3.6055
   eight       1852         3.6251
   five        1844         3.6095
   four        1839         3.5997
   go          1861         3.6427
   happy       1373         2.6875
   house       1427         2.7932
   left        1839         3.5997
   marvin      1424         2.7873
   nine        1875         3.6701
   no          1853         3.6271
   off         1839         3.5997
   :
```

Use `combine` to create a `CombinedDatastore` object from the audio data and the labels. Each call to `read` on the datastore returns one of the audio signals and its label.

```
lds = arrayDatastore(labels);
cds = combine(ads,lds);
```

You can create a separate datastore for validation data by repeating the same steps after creating an `audioDatastore` that instead points to the `validation` subfolder of the data set. Alternatively, you can use `splitlabels` to separate an existing datastore into training and validation sets. Specify `UnderlyingDatastoreIndex` to indicate which of the underlying datastores in the combined datastore contains the labels.

```
idxs = splitlabels(cds,0.8,"randomized",UnderlyingDatastoreIndex=2);
trainDs = subset(cds,idxs{1});
valDs = subset(cds,idxs{2});
```

Call `read` on the train datastore. The function returns both the audio signal and the label in a cell array.

```
read(trainDs)

ans=1x2 cell array
    {14861x1 double}    {[bed]}
```

Data With File Name Labels

The Free Spoken Digit Dataset (FSDD) [2] on page 1-1013 contains recordings of spoken digits in files whose names contain the digit labels as well as speaker labels. Download the data set and create an `audioDatastore` that points to the data.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","FSDD.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
dataset = fullfile(dataFolder,"FSDD","recordings");
ads = audioDatastore(dataset);
```

Select a random file from the data set and display its name. The file name is formatted as `digitLabel_speakerName_index`.

```
[~,name] = fileparts(ads.Files{randi(length(ads.Files))})

name =
'1_jackson_45'
```

Use `filenames2labels` to extract the digit labels from the file names. Combine the labels with the audio into a `CombinedDatastore` and see the label distribution of the data set.

```
labels = filenames2labels(ads,ExtractBefore="_");
lds = arrayDatastore(labels);
cds = combine(ads,lds);
```

```
countLabels(cds,UnderlyingDatastoreIndex=2)
```

```
ans=10x3 table
    Label    Count    Percent
    _____
```

```
0      200      10
1      200      10
2      200      10
3      200      10
4      200      10
5      200      10
6      200      10
7      200      10
8      200      10
9      200      10
```

Data With Metadata File

The Mozilla Common Voice data set [3] on page 1-1013 contains recordings of subjects speaking short sentences. The data set has a metadata file with various labels including sentence transcriptions and speaker IDs. Download the data set and create an `audioDatastore` that points to the training data.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","commonvoice.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
dataset = fullfile(dataFolder,"commonvoice","train");
ads = audioDatastore(fullfile(dataset,"clips"));
```

Read the metadata file into a table.

```
metadata = readtable(fullfile(dataset,"train.tsv"),FileType="text");
```

Assert that the order of the files in the datastore matches the table. This ensures you can easily associate the metadata information with the datastore.

```
[~,adsFileNames,~] = fileparts(ads.Files);
assert(length(adsFileNames)==length(metadata.path))
assert(all(strcmp(adsFileNames,metadata.path)))
```

Create a `CombinedDatastore` using the transcribed sentences as labels.

```
sentences = arrayDatastore(string(metadata.sentence));
transcriptDs = combine(ads,sentences);
```

Create another `CombinedDatastore` with speaker IDs as labels. Rename the speaker ID labels to natural numbers for simplicity.

```
speakerLabels = categorical(metadata.client_id);
speakerIDs = string(1:length(categories(speakerLabels)));
speakerLabels = renamecats(speakerLabels,speakerIDs);
```

```
labelsDs = arrayDatastore(speakerLabels);
speakerDs = combine(ads,labelsDs);
countLabels(speakerDs,UnderlyingDatastoreIndex=2)
```

```
ans=595x3 table
   Label   Count   Percent
   ----   -
   1         1     0.05
  10         1     0.05
```

100	3	0.15
101	4	0.2
102	36	1.8
103	4	0.2
104	1	0.05
105	2	0.1
106	4	0.2
107	1	0.05
108	1	0.05
109	1	0.05
11	4	0.2
110	1	0.05
111	1	0.05
112	10	0.5
:		

Next Steps

You can now use the data sets to train deep learning or machine learning models, and you can use `read` and `readall` to access the data and labels. You can also perform feature extraction on the data and use `transform` to create a new datastore that extracts features from the audio data.

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] Zohar Jackson, César Souza, Jason Flaks, Yuxin Pan, Hereman Nicolas, and Adhish Thite. "Jakobovski/free-spoken-digit-dataset: V1.0.8". Zenodo, August 9, 2018. <https://doi.org/10.5281/zenodo.1342401>.

[3] Mozilla Common Voice. <https://commonvoice.mozilla.org/en>.

See Also

Objects

`audioDatastore`

Functions

`folders2labels` | `filenames2labels` | `countlabels`

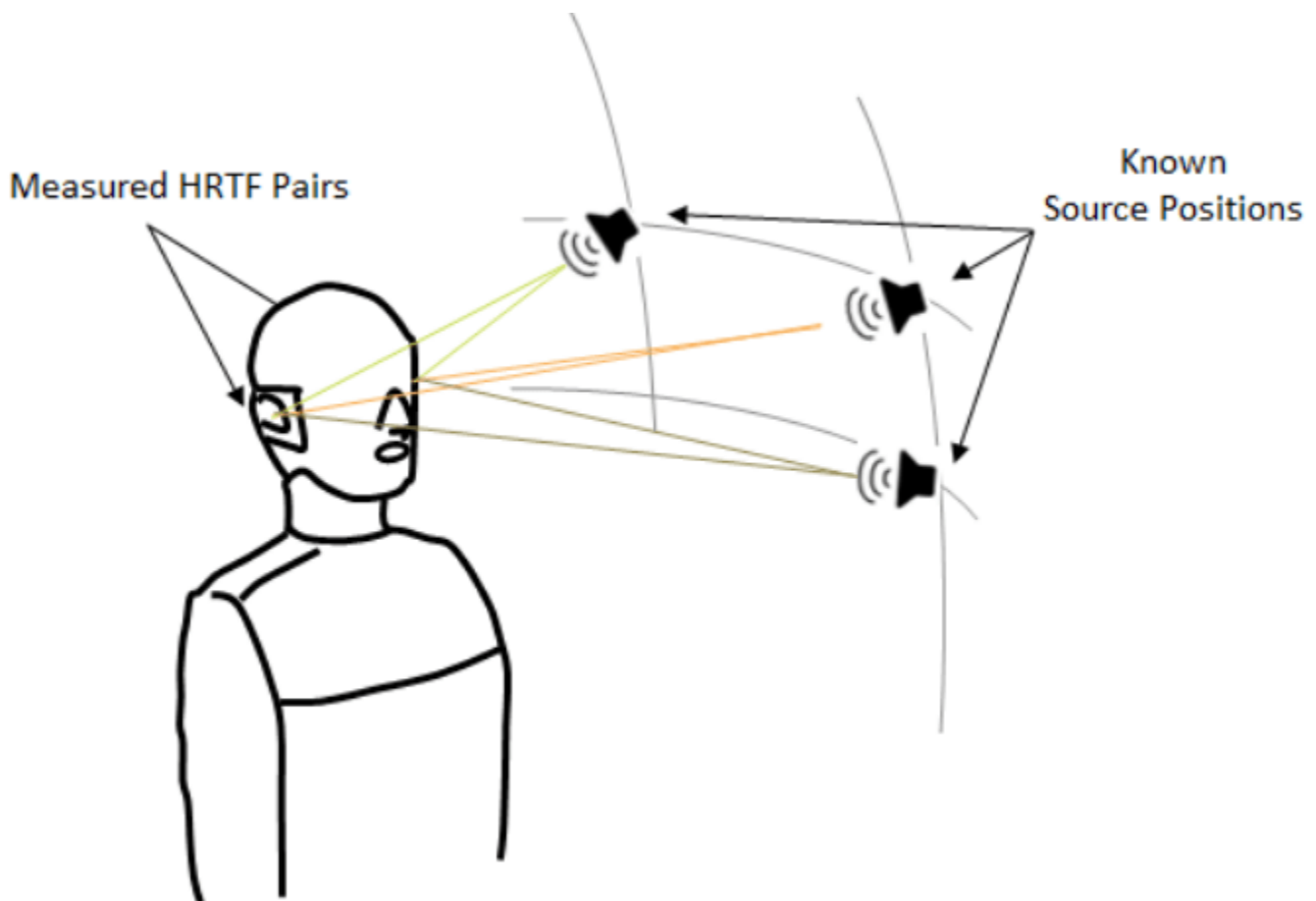
Read, Analyze and Process SOFA Files

SOFA (Spatially-Oriented Format for Acoustics) [1] on page 1-1032 is a file format for storing spatially oriented acoustic data like head-related transfer functions (HRTF) and binaural or spatial room impulse responses. SOFA has been standardized by the Audio Engineering Society (AES) as AES69-2015.

In this example, you load a SOFA file containing HRTF measurements for a single subject in MATLAB. You then analyze the HRTF measurements in the time domain and the frequency domain. Finally, you use the HRTF impulse responses to spatialize an audio signal in real time by modeling a moving source based on desired azimuth and elevation values.

Load a SOFA File in MATLAB

You use a SOFA file from the SADIE II database [2] on page 1-1032. The file corresponds to spatially discrete free-field in-the-ear HRTF measurements for a single subject. The measurements characterize how each ear receives a sound from a point in space.



Download the SOFA file.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "SOFA/SOFA.zip");  
dataFolder = tempdir;
```



```

unzip(downloadFolder,dataFolder)
netFolder = fullfile(dataFolder,"SOFA");
addpath(netFolder)
filename = "H10_48K_24bit_256tap_FIR_SOFA.sofa";

```

Display SOFA File Contents

SOFA files consist of binary data stored in the netCDF-4 format. You can use MATLAB to read and write netCDF files.

Display the contents of the SOFA file using `ncdisp` (execute `ncdisp(filename)`).

The file contents consist of multiple fields corresponding to different aspects of the measurements, such as the (fixed) listener position, the varying source position, the coordinate system used to capture the data, general metadata related to the measurement, as well as the measured impulse responses.

NetCDF is a "self-describing" file format, where data is stored along with attributes that can be used to assist in its interpretation. Consider the display snippet corresponding to the source position for example:

```

SourcePosition
  Size:      3x2114
  Dimensions: C,M
  Datatype:  double
  Attributes:
    Type = 'spherical'
    Units = 'degree, degree, metre'

```

`SourcePosition` contains the coordinates for the varying source position used in the measurements (here, there are 2114 separate positions). The file also contains attributes (Type, Units) describing the coordinate system used to store the positions (here, spherical), as well as information about the dimensions of the data (C,M). The dimensions are defined in the AES69 standard [3] on page 1-1032:

Dimension	Description
M	Number of measurements; shall be integer greater than zero.
R	Number of receivers or harmonic coefficients describing receivers; shall be integer greater than zero.
E	Number of emitters or harmonic coefficients describing emitters; shall be integer greater than zero.
N	Number of data samples describing one measurement; shall be integer greater than zero.
S	Number of characters in a string; shall be integer equal or greater than zero.
I	Singleton dimension, always one (1); defines a scalar value.
C	Coordinate triplet, always three (3); the coordinate type defines the meaning of this dimension.

For the file in this example:

- M = 2114 (the total number of measurements, each corresponding to a unique source position).
- R = 2 (corresponding to the two ears).
- E = 1 (one emitter or sound source per measurement).
- N = 256 (the length of each recorded impulse response).

Read SOFA File Information

Use `ncinfo` to get information about the SOFA file.

```
SOFAMInfo = ncinfo(filename);
```

The fields of the structure `SOFAMInfo` hold information related to the file's dimensions, variables and attributes.

Read a Variable from the SOFA File

Use `ncread` to read the value of a variable in the SOFA file.

Read the variable corresponding to the measured impulse responses.

```
ir = ncread(filename, "Data.IR");  
size(ir)  
  
ans = 1×3  
  
256 2 2114
```

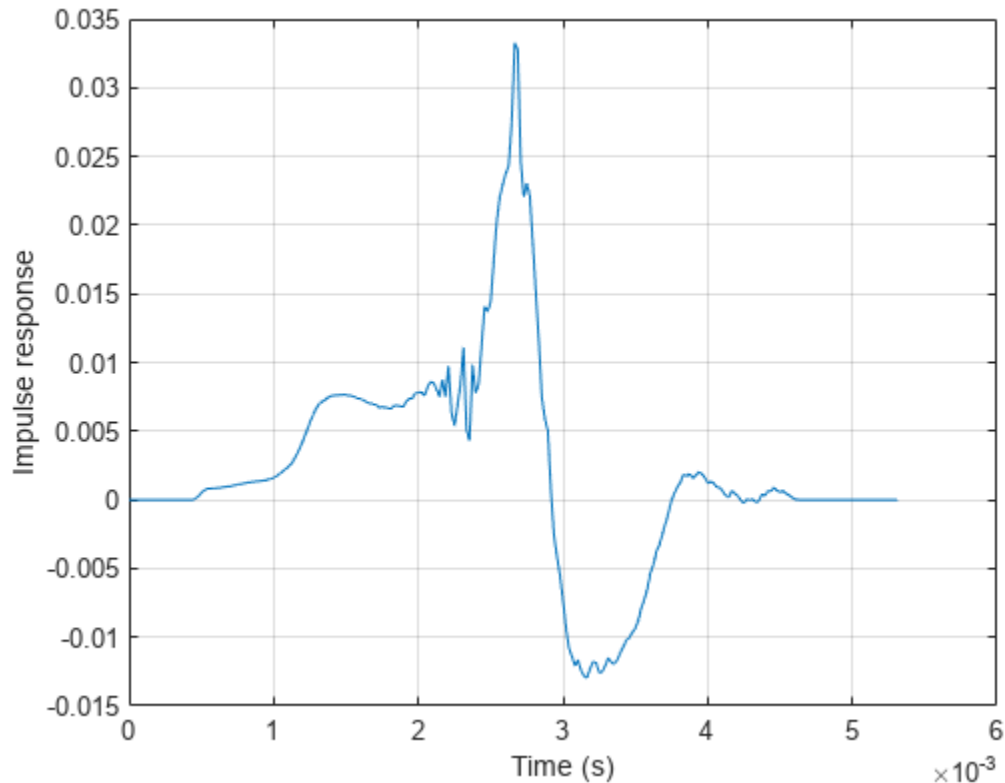
This variable holds impulse responses for the left and right ear for 2114 independent measurements. Each impulse response is of length 256.

Read the sampling rate of the measurements.

```
fs = ncread(filename, "Data.SamplingRate")  
  
fs = 48000
```

Plot the first measured impulse response.

```
figure;  
t = (0:size(ir,1)-1)/fs;  
plot(t,ir(:,1,1))  
grid on  
xlabel("Time (s)")  
ylabel("Impulse response")
```



The SOFA Object

It is possible to read and analyze the contents of the SOFA file using a combination of `ncinfo` and `ncread`. However, the process can be cumbersome and time consuming.

This example introduces a new object, `sofa`, that simplifies this operation. Use `sofa` to read the contents of a SOFA file, analyze the HRTF measurements in the time domain and the frequency domain, and spatialize audio signals based on desired source positions.

Use the `sofa` object to read the contents of the SOFA file.

```
s = sofa(filename)

s =
  sofa with properties:
      IR: [2114x2x256 double]
      SamplingRate: 48000
      SamplingRateUnits: 'hertz'
      Delay: [0 0]

  Show all properties
```

Click "Show all properties" in the display above to see the rest of the properties from the SOFA file.

You can easily access the measured impulses responses on the `sofa` object.

```
ir = s.IR;
size(ir)

ans = 1×3

      2114      2      256
```

You access other variables in a similar fashion. For example, read the source positions along with the coordinate system used to express them:

```
srcPositions = s.SourcePosition;
size(srcPositions)

ans = 1×2

      2114      3
```

```
srcPositions(1,:)

ans = 1×3

      0  -90.0000  1.2000
```

```
s.SourcePositionType
```

```
ans =
'spherical'
```

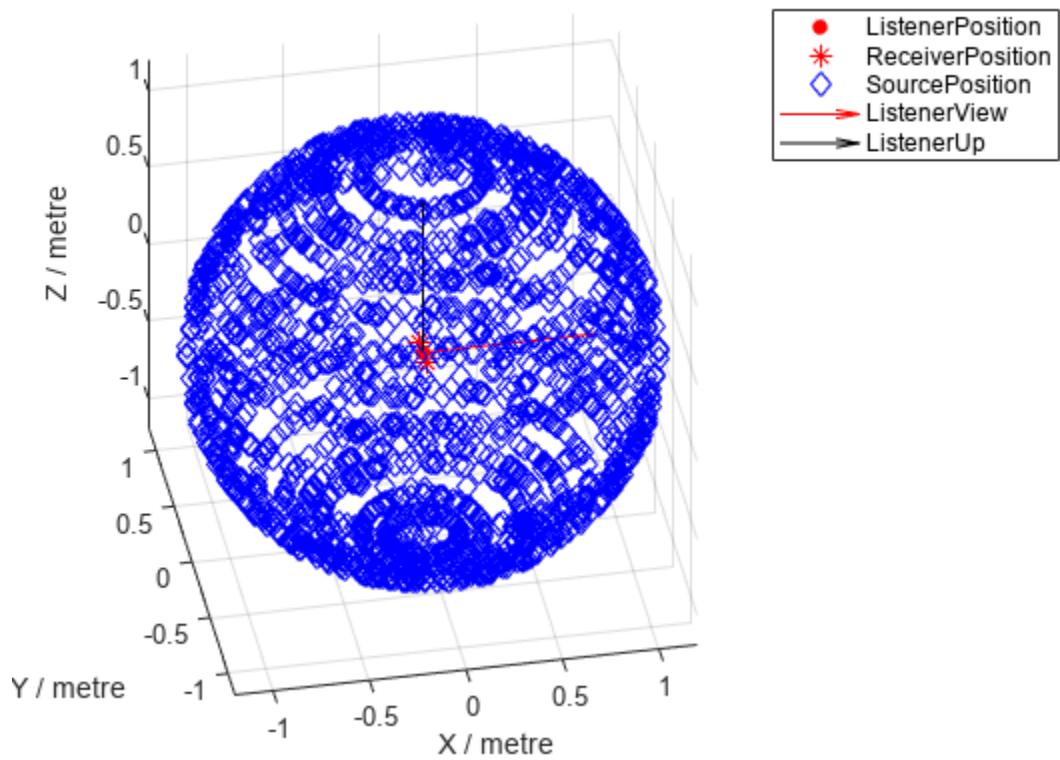
```
s.SourcePositionUnits
```

```
ans =
'degree, degree, metre'
```

View the Measurement Geometry

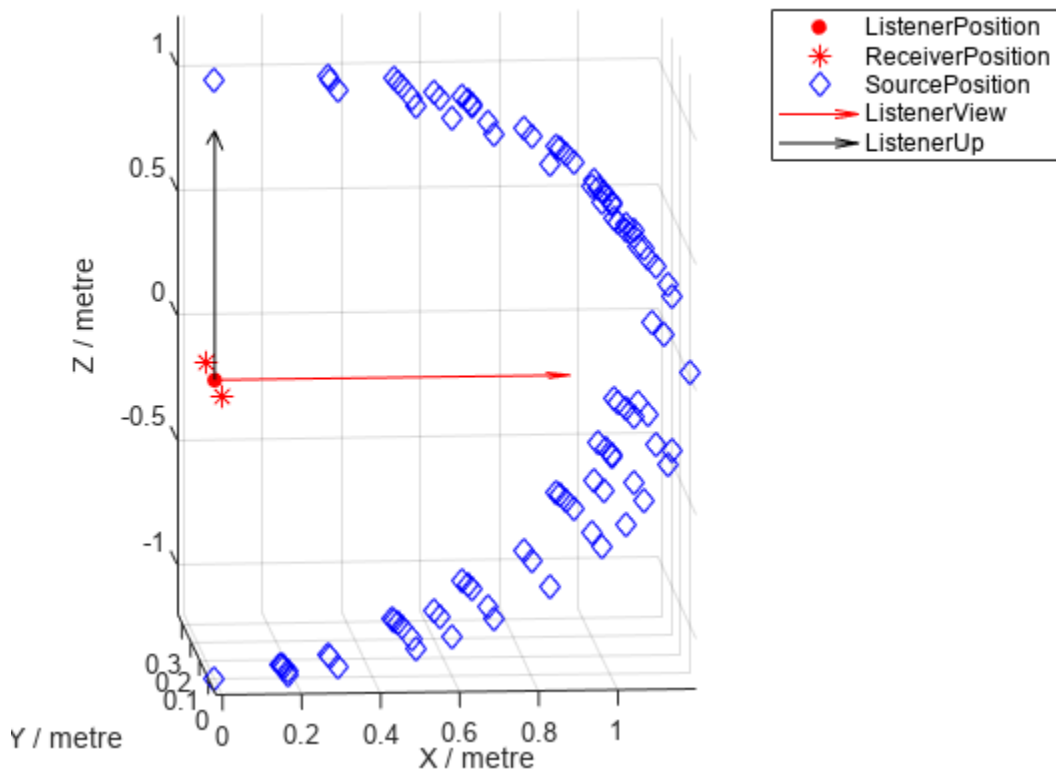
Call `plotGeometry` to view the general geometric setup of the measurements.

```
figure;
plotGeometry(s)
a = gca;
a.CameraPosition = [-3 -16 12];
```



Alternatively, specify input indices to restrict the plot to desired source locations.

```
figure;  
plotGeometry(s,1:100)  
a = gca;  
a.CameraPosition = [-3 -16 12];
```



Compute HRTF Frequency Responses

The file in this example uses the `SimpleFreeFieldHRIR` convention, which stores impulse response measurements in the time domain as FIR filters.

```
s.SOFAConventions
```

```
ans =  
'SimpleFreeFieldHRIR'
```

```
s.DataType
```

```
ans =  
'FIR'
```

It is straightforward to compute and plot the frequency response of the impulse responses using `freqz`.

As an illustration, assume you want to plot the frequency response of measurements with an azimuth value in the range of 30 degrees to 32 degrees.

First, inspect the type of the source position data.

```
s.SourcePositionType
```

```
ans =  
'spherical'
```

```
s.SourcePositionUnits
```

```
ans =
'degree, degree, metre'
```

The coordinates are spherical, which is convenient for your purpose. Also, note that the units are in degrees, so conversion from radians per second to degrees is not necessary.

Per the SOFA standard [3] on page 1-1032, the first angle value corresponds to the azimuth.

```
az = s.SourcePosition(:,1);
```

Find the values corresponding to the desired azimuth range.

```
indices = find(az>30 & az<32);
```

You are interested in the impulse responses corresponding to the first receiver (ear) only. Read the corresponding impulse responses.

```
receiver = 1;
IR = s.IR(indices,receiver,:);
IR = permute(IR,[3 1 2]);
IR = squeeze(IR);
size(IR)
```

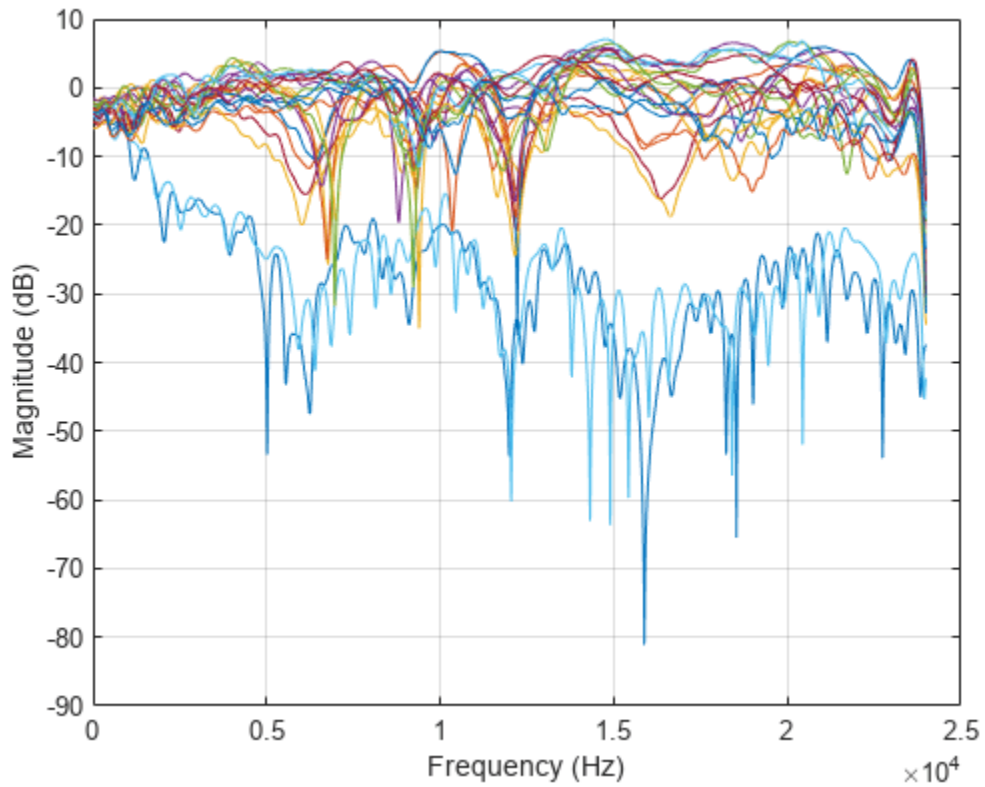
```
ans = 1×2
      256      22
```

Use `freqz` to compute the frequency response of each FIR filter. Specify a 4096-point frequency response.

```
N = 4096;
H = zeros(N,length(indices));
for index = 1:length(indices)
    [H(:,index),F] = freqz(IR(:,index),1, N ,s.SamplingRate);
end
```

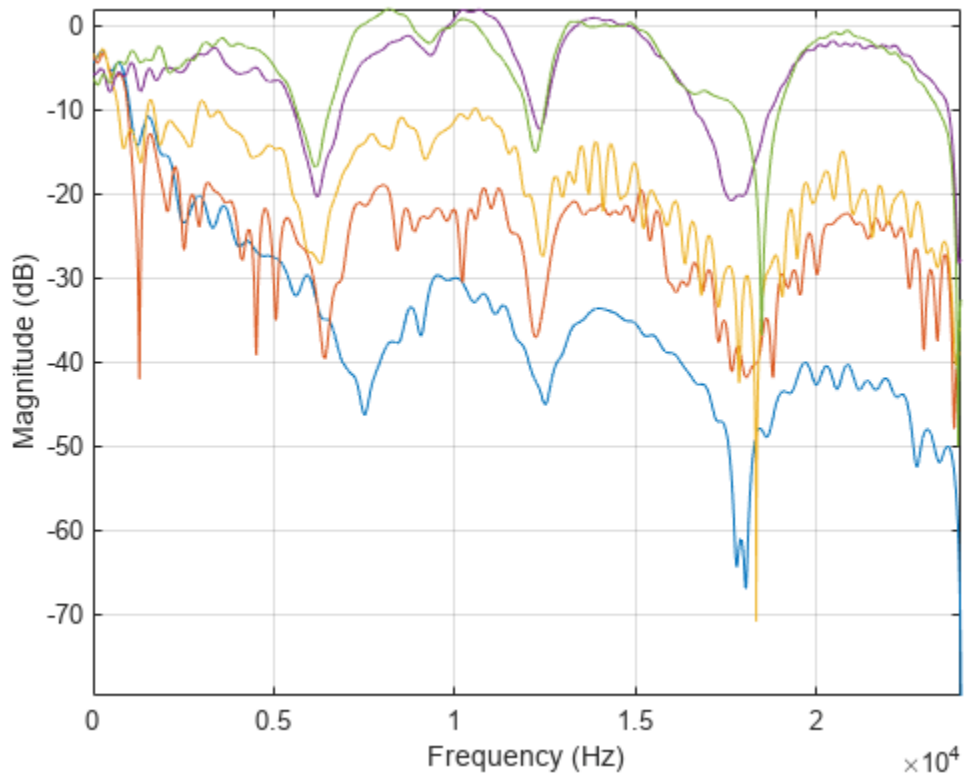
Plot the magnitude responses.

```
figure
for index=1:length(indices)
    plot(F,20*log10(abs(H(:,index))));
    hold on
end
grid on
ylabel("Magnitude (dB)");
xlabel("Frequency (Hz)");
```



You can also accomplish this task by calling `freqz` on the `sofa` object directly. As an example, plot the frequency responses of the first 5 impulse responses of the second receiver.

```
figure  
freqz(s,Receiver=2,Indices=1:5)
```

Compute HRTF Spectra

It is often useful to compute and visualize the magnitude spectra of HRTF data in a specific plane in space.

For example, compute the spectrum in the horizontal plane (corresponding to an elevation angle equal to zero) for the first receiver.

Find measurements with an elevation angle within 2 degrees of zero.

```
threshold = 2;
ele = s.SourcePosition(:,2);
indices = find(abs(ele)<threshold);
```

Read the corresponding impulse responses for one ear.

```
receiver = 1;
IR = s.IR(indices,receiver,:);
IR = permute(IR,[3 1 2]);
IR = squeeze(IR);
size(IR)
```

```
ans = 1x2
```

```
256 96
```

Use `freqz` to compute the frequency response of each FIR filter. Specify a 4096-point frequency response.

```
N = 4096;
H = zeros(N,length(indices));
for index = 1:length(indices)
    [H(:,index),F] = freqz(IR(:,index),1, N ,s.SamplingRate);
end
```

Convert to log scale.

```
H = 20*log10(abs(H)).';
```

Eliminate small values by using a noise floor.

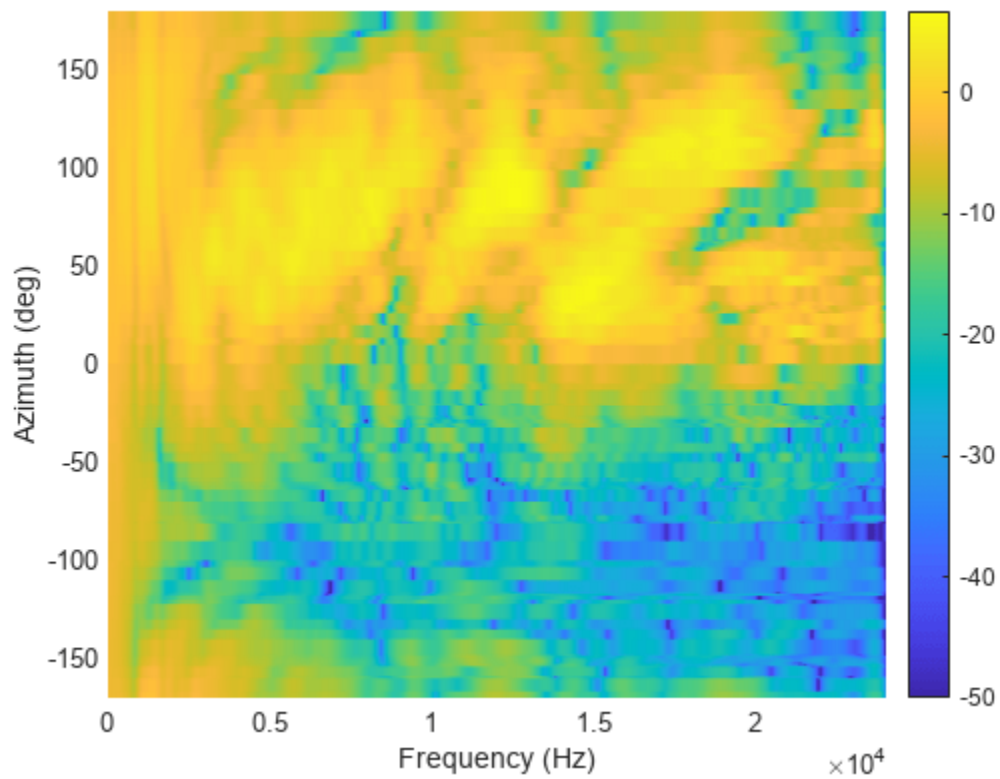
```
noiseFloor = -50;
H(H<noiseFloor) = noiseFloor;
```

Sort data by azimuth values. Note that the SOFA standard uses an azimuth range of [0, 360] degrees. Convert it to [-180,180] degrees.

```
azi = s.SourcePosition(indices,1);
azi(azi>180)=azi(azi>180)-360;
[azi,ind]=sort(azi);
H = H(ind,:);
```

Plot the horizontal plane spectrum.

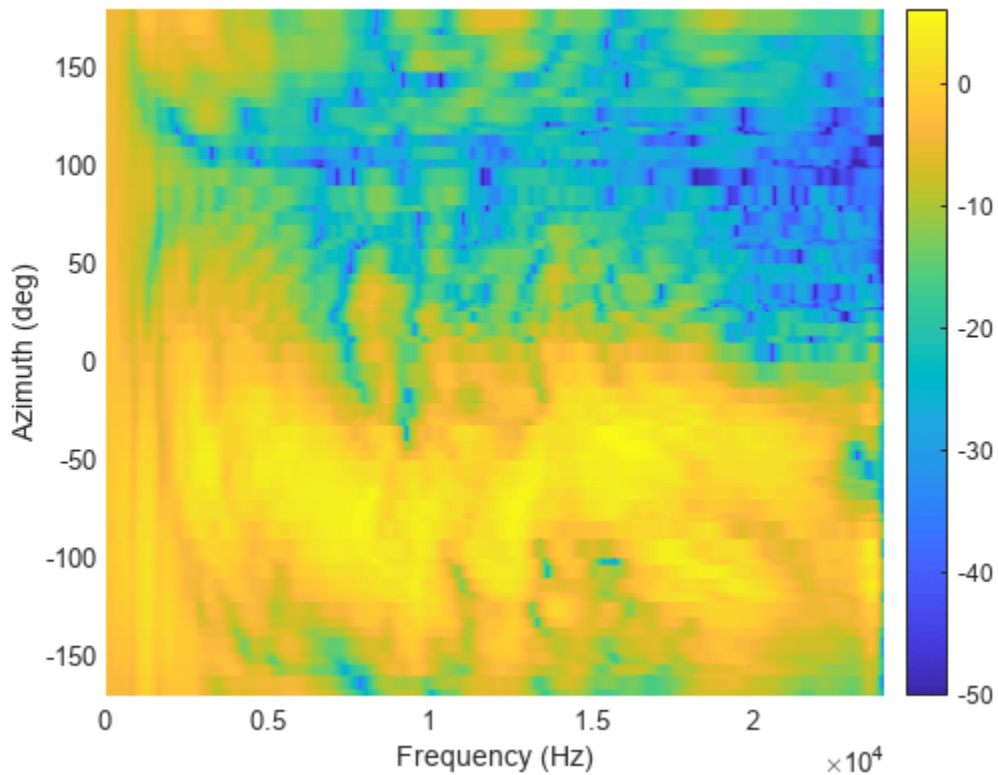
```
figure
surface(F.',azi,H(:,:,));
shading flat
xlabel("Frequency (Hz)");
ylabel("Azimuth (deg)");
colorbar;
axis tight
```



You can also plot the horizontal spectrum by directly using the `spectrum` method of the `sofa` object.

Use the `spectrum` method to plot the spectrum in the horizontal plane for the second receiver.

```
figure;  
spectrum(s, Receiver=2)
```



Compute Energy-Time Curve

It is often useful to visualize the decay of the HRTF responses over time using an energy-time curve (ETC).

In this section, you measure and plot the ETC in the horizontal plane. You use the same impulse responses you used to compute the magnitude spectrum in the horizontal plane in the previous section.

Convert the impulse response values to the log domain.

```
IRLog = 20*log10(abs(IR));
```

Similar to the previous section, sort the data by azimuth values.

```
IRLog = IRLog(:,ind);
```

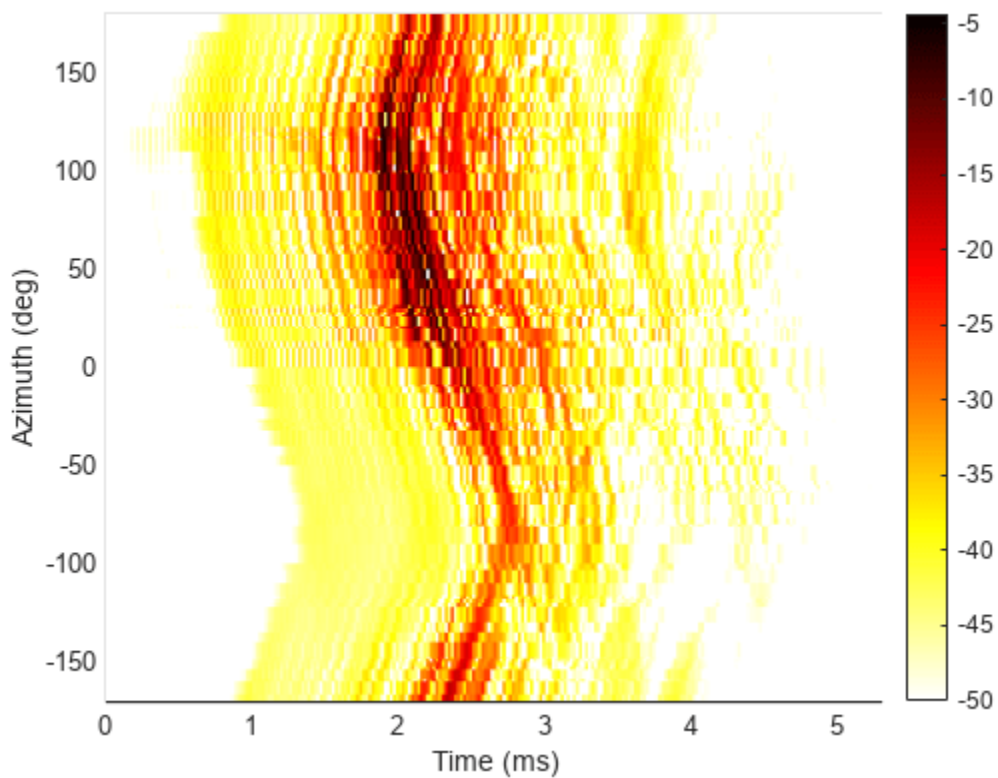
Eliminate values smaller than the noise floor.

```
noiseFloor=-50;
IRLog(IRLog<=noiseFloor)=noiseFloor;
```

Plot the ETC.

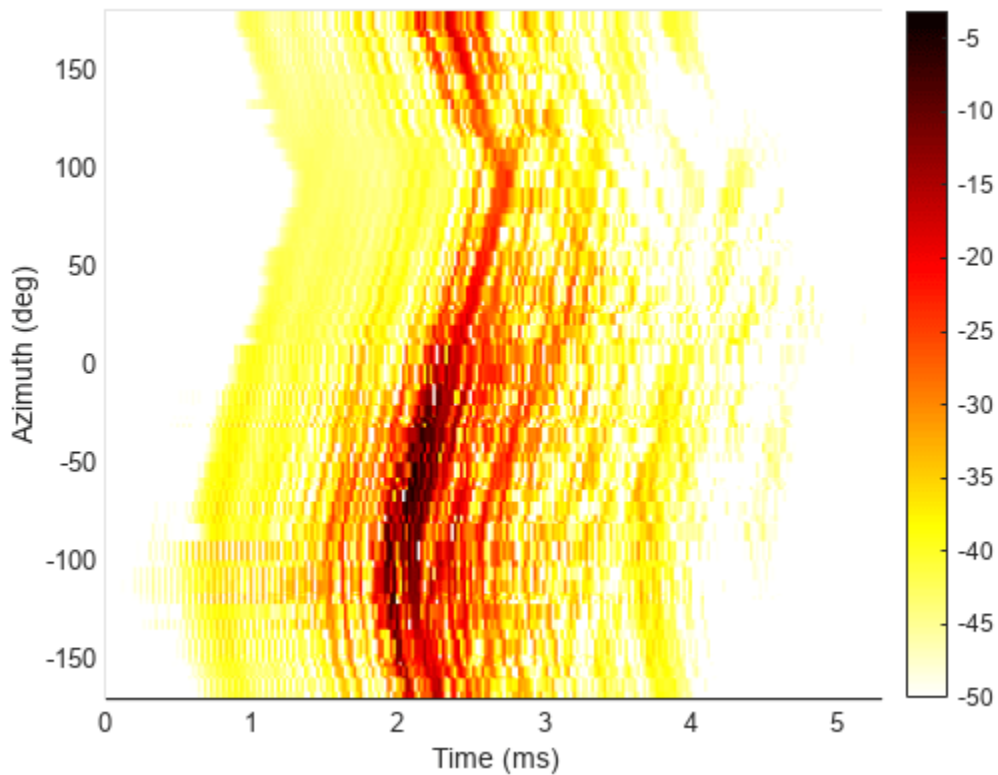
```
fs = s.SamplingRate;
t = 0:1/fs*1000:(size(IRLog,1)-1)/fs*1000;
figure
surface(t,azi,IRLog.');
```

```
set(gca,FontName="Arial",FontSize=10);  
set(gca, TickLength=[0.02 0.05]);  
set(gca,LineWidth=1);  
cmap=colormap(hot);  
cmap=flipud(cmap);  
shading flat  
colormap(cmap);  
box on;  
colorbar;  
xlabel("Time (ms)");  
ylabel("Azimuth (deg)");  
axis tight  
grid on
```



Use the `energyTimeCurve` method of the `sofa` object to generate such ETC visualizations. For example, visualize the ETC in the horizontal plane for the second receiver.

```
figure;  
energyTimeCurve(s, Receiver=2)
```



Compute Interaural Time Delay

Interaural time delay (ITD) is the difference in arrival time of a sound between two ears. It is an important binaural cue for sound source localization. There are many ITD estimation methods (see [4] on page 1-1032). Here, you compute the ITD using a simple cross-correlation technique.

First, pass the impulse responses through a lowpass filter.

Design the lowpass filter.

```
cutOffFreq = 3000;
fs = s.SamplingRate;
cutOffFreqNorm = cutOffFreq/(fs/2);
[b,a] = butter(5,cutOffFreqNorm);
```

Filter the impulse responses.

```
ir = filter(b,a,s.IR,[],3);
```

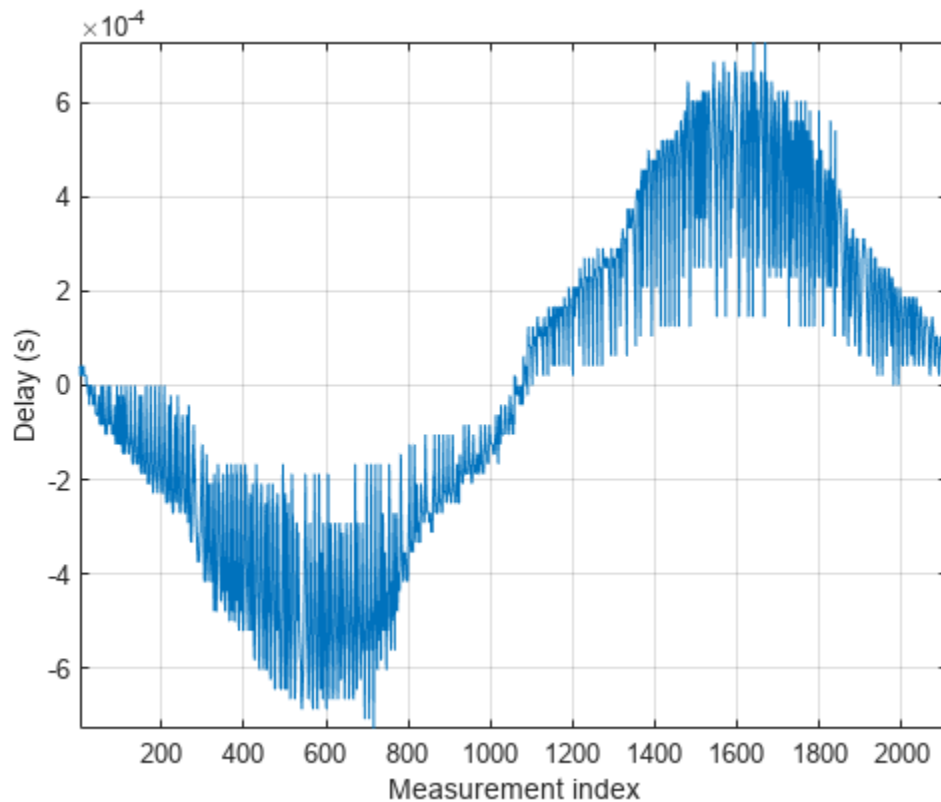
Estimate the delay between the left and right ear responses using a cross-correlation metric.

```
pos = size(ir,1);
toa_diff = zeros(1,pos);
N = size(ir,3);
for ii=1:pos
    cc = xcorr(squeeze(ir(ii,1,:)),squeeze(ir(ii,2,:)));
    [~,idx_lag] = max(abs(cc));
    toa_diff(ii) = idx_lag - N;
```

```
end  
toa_diff = toa_diff/fs;
```

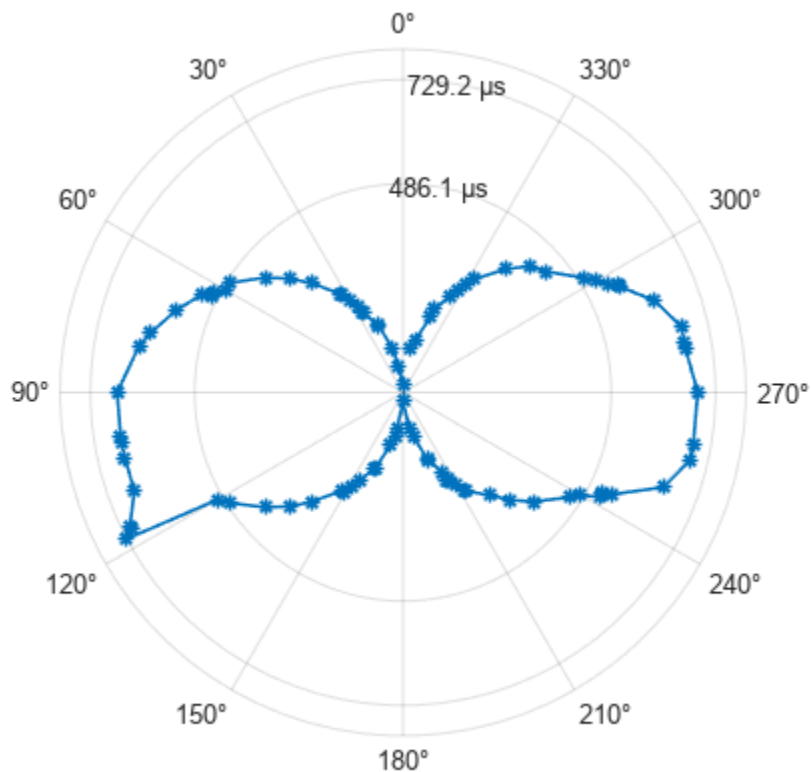
You can perform the same task by calling the `interauralTimeDelay` method of the `sofa` object. To get a utility plot displaying ITD for all measurements, call the method with no output arguments.

```
interauralTimeDelay(s)
```



Plot ITD versus the azimuth angle in the horizontal plane using the method `horizontalITD`. The method leverages `polarplot`.

```
horizontalITD(s);
```



Interpolate HRTF Measurements

The HRTF measurements in the SOFA file correspond to a finite number of azimuth/elevation angle combinations. It is possible to interpolate the data to any desired spatial location using 3-D HRTF interpolation with `interpolateHRTF`.

Specify the desired source position (in degrees).

```
desiredAz = [-120;-60;0;60;120;0;-120;120];
desiredEl = [-90;90;45;0;-45;0;45;45];
desiredPosition = [desiredAz,desiredEl];
```

Read the measured source positions stored in the SOFA file.

```
pos = s.SourcePosition;
sourcePosition = pos(:,1:2);
```

Per the SOFA standard, azimuth values are stored in the range zero to 360 degrees. Since the desired azimuth values contain negative angles, switch the stored values to the range -180 to 180 degrees.

```
azi = sourcePosition(:,1);
azi=mod(azi,360);
idx=find(azi>180 & azi<=360);
azi(idx) = -(360-azi(idx));
sourcePosition(:,1) = azi;
```

Read the measured impulse responses.


```
hrtfData = s.IR;
```

Calculate the head-related impulse response (HRIR) using the vector base amplitude panning interpolation (VBAP) algorithm at a desired source position.

```
interpolatedIR = interpolateHRTF(hrtfData, ...
    sourcePosition, ...
    desiredPosition);
```

The above steps break down how to use `interpolateHRTF` with the SOFA data. You can accomplish this task easily by directly calling the `interpolateHRTF` method of the `sofa` object:

```
interpolatedIR2 = interpolateHRTF(s,desiredPosition);
```

Separate the output into the impulse responses of the left and right ears.

```
leftIR = squeeze(interpolatedIR(:,1,:));
rightIR = squeeze(interpolatedIR(:,2,:));
```

Model Moving Source Using HRIR Filtering

Filter a mono input through the interpolated impulse responses to model a moving source.

Create an audio file sampled at 48 kHz for compatibility with the HRTF dataset.

```
desiredFs = s.SamplingRate;
[audio,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
audiowrite("Counting-16-48-mono-15secs.wav",audio,desiredFs);
```

Create a `dsp.AudioFileReader` object to read in a file frame by frame. Create an `audioDeviceWriter` object to play audio to your sound card frame by frame. Create two `dsp.FIRFilter` objects with `NumeratorSource` set to `Input port`. Setting `NumeratorSource` to `Input port` enables you to modify the filter coefficients while streaming.

```
fileReader = dsp.AudioFileReader("Counting-16-48-mono-15secs.wav");
deviceWriter = audioDeviceWriter(SampleRate=fileReader.SampleRate);
```

```
leftFilter = dsp.FIRFilter(NumeratorSource="Input port");
rightFilter = dsp.FIRFilter(NumeratorSource="Input port");
```

In an audio stream loop:

- 1 Read in a frame of audio data.
- 2 Feed the audio data through the left and right HRIR filters.
- 3 Concatenate the left and right channels and write the audio to your output device. If you have a stereo output hardware, such as headphones, you can hear the source shifting position over time.
- 4 Modify the desired source position in 2-second intervals by updating the filter coefficients.

```
durationPerPosition = 2;
samplesPerPosition = durationPerPosition*fileReader.SampleRate;
samplesPerPosition = samplesPerPosition - rem(samplesPerPosition,fileReader.SamplesPerFrame);

sourcePositionIndex = 1;
samplesRead = 0;
while ~isDone(fileReader)
    audioIn = fileReader();
    samplesRead = samplesRead + fileReader.SamplesPerFrame;
```

```
leftChannel = leftFilter(audioIn,leftIR(sourcePositionIndex,:));
rightChannel = rightFilter(audioIn,rightIR(sourcePositionIndex,:));

deviceWriter([leftChannel,rightChannel]);

if mod(samplesRead,samplesPerPosition) == 0
    sourcePositionIndex = sourcePositionIndex + 1;
end
end
```

As a best practice, release your System objects when complete.

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

Spatialize Audio in Real Time

You can use the `sofa` object directly to simulate a sound source moving in space.

Simulate a sound source moving in the horizontal plane, with an initial azimuth of -90 degrees. Gradually increase the azimuth in the loop. Pass the audio frames to the `sofa` object along with the desired source location.

```
index = 1;
loc = [-90 0];
while ~isDone(fileReader)
    index=index+1;
    frame = fileReader();
    frame = frame(:,1);
    y = s(frame,loc);
    deviceWriter(y);
    if mod(index,100)==0
        loc(1)=loc(1)+30;
    end
end

release(deviceWriter)
release(fileReader)
```

References

[1] SOFA (Spatially Oriented Format for Acoustics). <https://www.sofaconventions.org>

[2] <https://www.york.ac.uk/sadie-project/database.html>

[3] AES69-2022: AES standard for file exchange - Spatial acoustic data file format

[4] Andreopoulou A, Katz BFG. Identification of perceptually relevant methods of inter-aural time difference estimation. J Acoust Soc Am. 2017 Aug;142(2):588. doi: 10.1121/1.4996457. PMID: 28863557.

See Also

interpolateHRTF

Related Examples

- “Room Impulse Response Simulation with the Image-Source Method and HRTF Interpolation” on page 1-833
- “Binaural Audio Rendering Using Head Tracking” on page 1-65

Impulse Response Measurement Using a NI USB-4431 Device

This example shows how to measure an impulse response using a National Instruments (NI) USB-4431 sound and vibration device. You set up the device, create an excitation signal to play, and simultaneously record the response. Finally, you compute the response from the recording and plot the results.

For this example, you need Audio Toolbox™ and Data Acquisition Toolbox™. You also need to have installed the NI drivers (recommended) or the MATLAB NI support package.

Device Setup

Verify if the NI USB-4431 is connected and the proper drivers are installed using the `daqlist` command.

```
daqlist("ni")
```

```
ans=1x4 table
```

DeviceID	Description	Model	DeviceInfo
"Dev1"	"National Instruments(TM) USB-4431"	"USB-4431"	1x1 daq.ni.DeviceSpecializa

Next, knowing that this device only supports clocked operations, you can disable the warning.

```
ws = warning("off", "daq:Session:clockedOnlyChannelsAdded");
% restore warning state when cleared
oc = onCleanup(@() warning(ws));
```

Setup the NI device with a sampling rate of 48 kHz, one input, and one output.

To do so, create a `daq` object for NI devices and set `Rate` to 48 kHz.

```
FS = 48e3;
dq = daq("ni");
dq.Rate = FS;
dev = "Dev1";
```

Add one input and one output. In this example, the microphone has its own pre-amplifier, so set the measurement type to `Voltage`. The output is connected to a powered loudspeaker and the measurement type is set to `Voltage`.

```
addinput(dq, dev, "ai0", "Voltage");
addoutput(dq, dev, "ao0", "Voltage");
```

Stimulus Signal

Create an exponential swept sine going from 20 Hz to 24 kHz over a period of 3 seconds, followed by one second of silence. This limits the maximum impulse response length to one second. You can also set the output level, in this case -18 dB.

```
irDur = 1;
sweepDur = 3;
outputLevel = -18;
sweepRange = [20 24000];
```

```
exc = sweeptone( ...  
    sweepDur, irDur, FS, ...  
    ExcitationLevel=outputLevel, ...  
    SweepFrequencyRange=sweepRange);
```

Response Measurement

Using the `readwrite` method of the `daq` object (`dq`), play the swept sine stimulus (`exc`) and simultaneously record the response.

```
data = readwrite(dq,exc);  
dq = []; % release
```

Compute the Impulse Response

Use the `impzest` function to compute the impulse response (`ir`) from the stimulus (`exc`) and its response.

```
response = data.Variables;  
ir = impzest(exc,response);
```

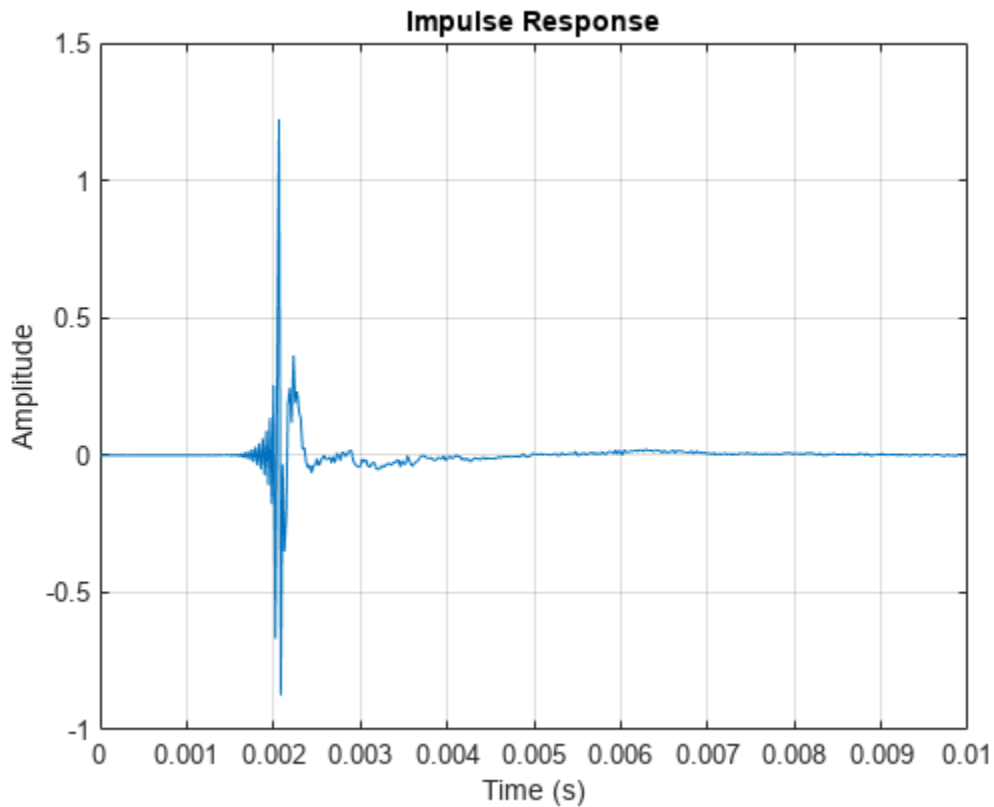
Create the corresponding time vector.

```
t_sec = (0:length(ir)-1) ./ FS;
```

Display the Results

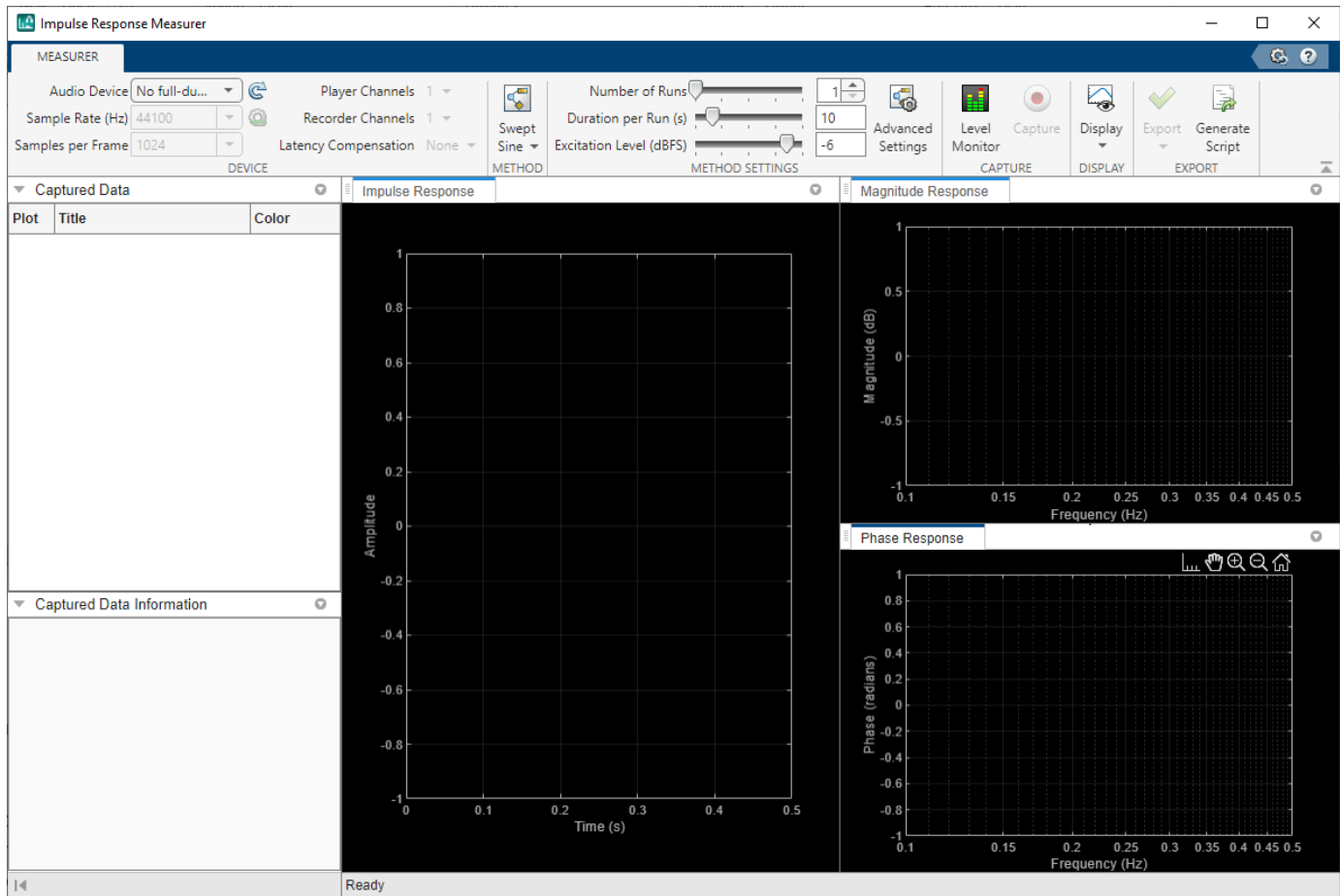
Plot the first 10 milliseconds, in this case, 480 samples.

```
totalDur = sweepDur*FS;  
n = 480;  
plot(t_sec(1:n), ir(1:n))  
title("Impulse Response")  
ylabel("Amplitude")  
xlabel("Time (s)")  
grid on
```



Impulse Response Measurer App

Another way of writing the code above is to start from a script generated from the **Impulse Response Measurer** app. Start the app by entering `impulseResponseMeasurer` at the command prompt. You can also click the app icon on the **Apps** tab of the MATLAB® Toolstrip. Even without a supported device connected, you can set the **Method Settings**, **Display Settings** and a linear or log scale for magnitude and phase responses (using the toolbar that appears when hovering the mouse over these plots). Then, click **Generate Script**. A new document will appear in the editor with code for an audio device, that you will modify to use the NI USB-4431.



Script Customization

First, you may want to make this a function by adding `function capture = irm_usb4431` to the top of the file, and an end statement before the first embedded function.

The function to interface with the NI device can take a complete signal, without proceeding frame by frame. Consequently, remove the code for "Allocate the input/output buffers" and "Copy the excitation to the output buffer".

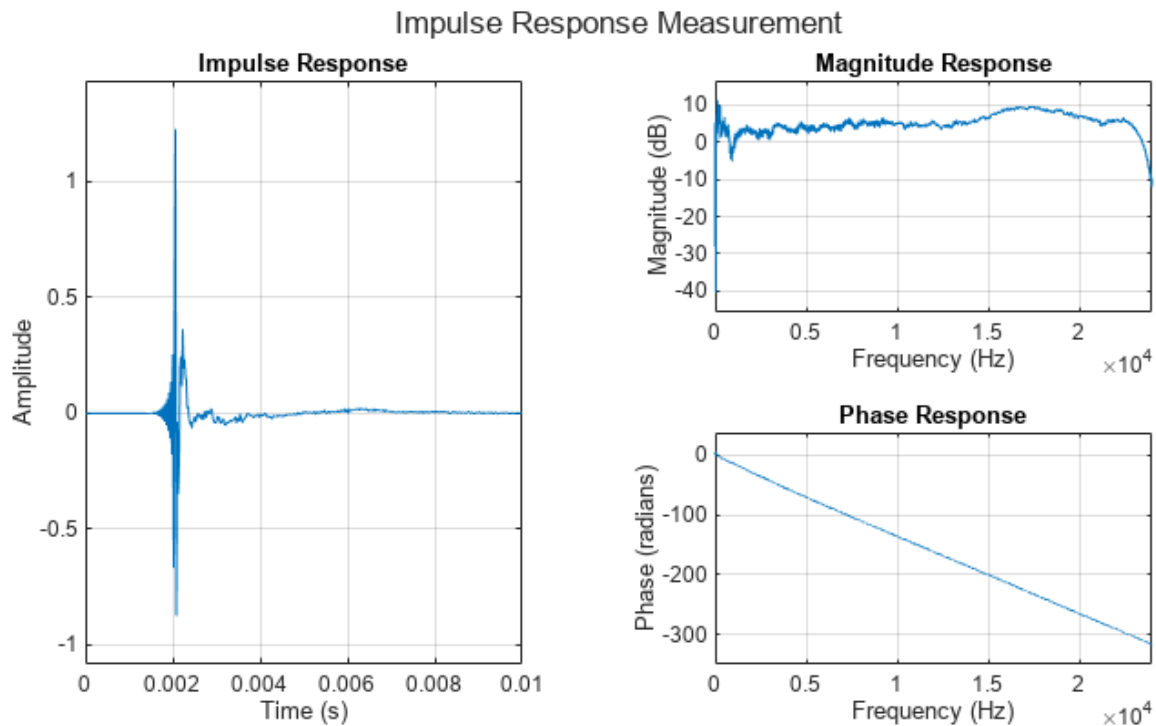
The code required to "Setup the audio device" can be replaced by code to setup the NI USB-4431 device. Set the sample rate of the device and add an output and at least one input.

```
dq = daq("ni");
dq.Rate = 48e3; % Sampling rate
addinput(dq,"Dev1","ai0","Voltage");
addoutput(dq,"Dev1","ao0","Voltage");
```

The playback and capture loop can be replaced by a `readwrite` statement. Get the results from `data.Variables` instead of the buffer. Also get the handle from the first figure to facilitate zooming in.

Now run the customized function `irm_usb4431` to obtain the capture data and plot the results and zoom in.

```
[capture,ax1] = irm_usb4431();
xlim(ax1,[0 10e-3]);
```



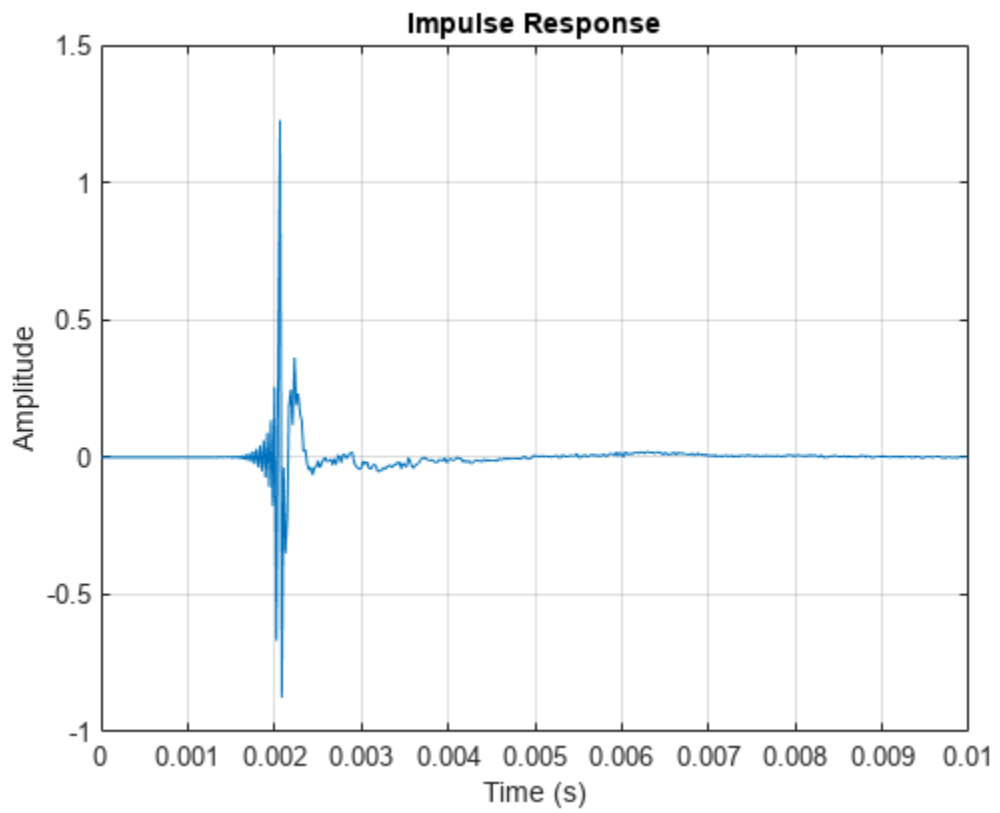
The data is also available programmatically.

capture

```
capture = struct with fields:
    ImpulseResponse: [1x1 struct]
    MagnitudeResponse: [1x1 struct]
    PhaseResponse: [1x1 struct]
```

For example, plot the first 480 samples of the impulse response.

```
figure % new figure
plot(capture.ImpulseResponse.Time(1:480), capture.ImpulseResponse.Amplitude(1:480))
title("Impulse Response")
ylabel("Amplitude")
xlabel("Time (s)")
grid on
```

See Also

Impulse Response Measurer

Plot Large Audio Files

This example shows how to plot large audio files in MATLAB. The first section shows a simple way to read and plot all the data in an audio file. The next two sections show how to read and plot only the envelope of an audio file without loading the entire audio file into memory.

Plot Short Audio Files

Use the `audioread` function to read an 11 second MP3 audio file. The `audioread` function can support other file formats. For a full list of viable formats, see “Supported File Formats for Import and Export”.

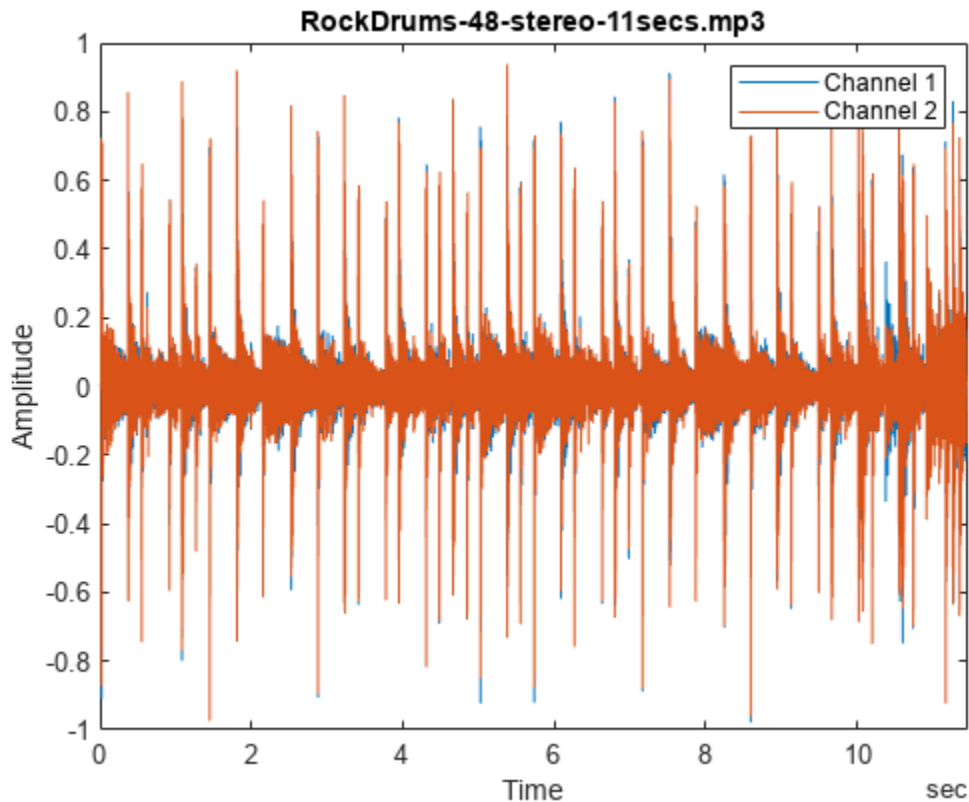
```
filename = "RockDrums-48-stereo-11secs.mp3";  
[y,fs] = audioread(filename);
```

Using the sample rate `fs` returned by `audioread`, create a duration vector `t` the same length as `y` to represent elapsed time.

```
t = seconds(0:1/fs:(size(y,1)-1)/fs);
```

The audio file contains a stereo signal. Plot the two channels of audio data `y` as a function of time `t`.

```
plot(t,y)  
title(filename)  
xlabel("Time")  
ylabel("Amplitude")  
legend("Channel 1", "Channel 2")  
xlim("tight")  
ylim([-1 1])
```



Plot Large Audio Files Using Audio Envelope

When the audio file is very long (hours or even several minutes), reading and plotting all the data in MATLAB might take significant time and memory resources. In such cases, you might not want to read all the data in MATLAB, if the only purpose is to visualize the waveform. You can use the `audioEnvelope` function to read an envelope of the audio file and plot only the overall envelope of the audio waveform.

```
filename = "SoftGuitar-44p1_mono-10mins.ogg";
auInfo = audioinfo(filename)

auInfo = struct with fields:
    Filename: 'B:\matlab\toolbox\audio\samples\SoftGuitar-44p1_mono-10mins.ogg'
    CompressionMethod: 'Vorbis'
    NumChannels: 1
    SampleRate: 44100
    TotalSamples: 26300000
    Duration: 596.3719
    Title: []
    Comment: []
    Artist: []
```

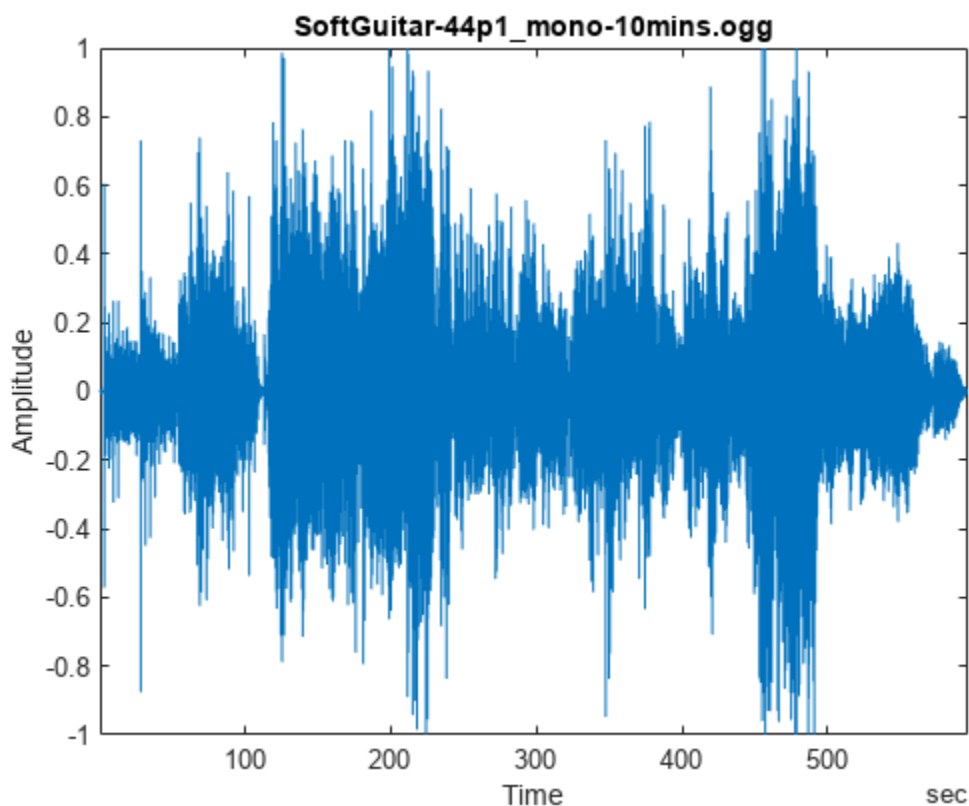
`SoftGuitar-44p1_mono-10mins.ogg` is approximately 10 minutes long, recorded at 44100 Hz, and contains 26.3 million audio samples.

Read the envelope of the audio signal in `SoftGuitar-44p1_mono-10mins.ogg` using the `audioEnvelope` function.

```
[envMin,envMax,loc] = audioEnvelope(filename,NumPoints=2000);
```

`audioEnvelope` returns `envMin` and `envMax` containing the minimum and maximum sample values over frames of length equal to `floor(L/numPoints)`, where `L` is the length of the audio signal and `numPoints` is the number of points returned by `audioEnvelope`. Connect `envMin` and `envMax` at each point and plot them as a function of time `t`.

```
nChans = size(envMin,2);
envbars = [shiftdim(envMin,-1);
           shiftdim(envMax,-1);
           shiftdim(NaN(size(envMin)),-1)];
ybars = reshape(envbars,[],nChans);
t = seconds(loc/auInfo.SampleRate);
tbars = reshape(repmat(t,3,1),[],1);
plot(tbars,ybars);
title(filename,Interpreter="none")
xlabel("Time")
ylabel("Amplitude")
xlim("tight")
ylim([-1 1])
```



Plot Large Audio Files Using a Custom Chart

In the previous section, you plot the audio envelope of a 10-minute audio file using 2000 points. You can zoom and pan the plot above, but when you zoom in, it does not fetch more data.

This section introduces a new custom chart, `audioplot`, which plots any audio file using the audio envelope technique and also makes it interactive so that when you zoom or pan, the plot fetches more

data from the audio file and updates the visual as needed. The custom chart `audioplot` is a subclass of the `ChartContainer` base class. By inheriting from the `ChartContainer` base class, instances of `audioplot` are members of the graphics object hierarchy and can be embedded in any MATLAB figure alongside other graphics objects. For more information, see “Chart Development Overview”.

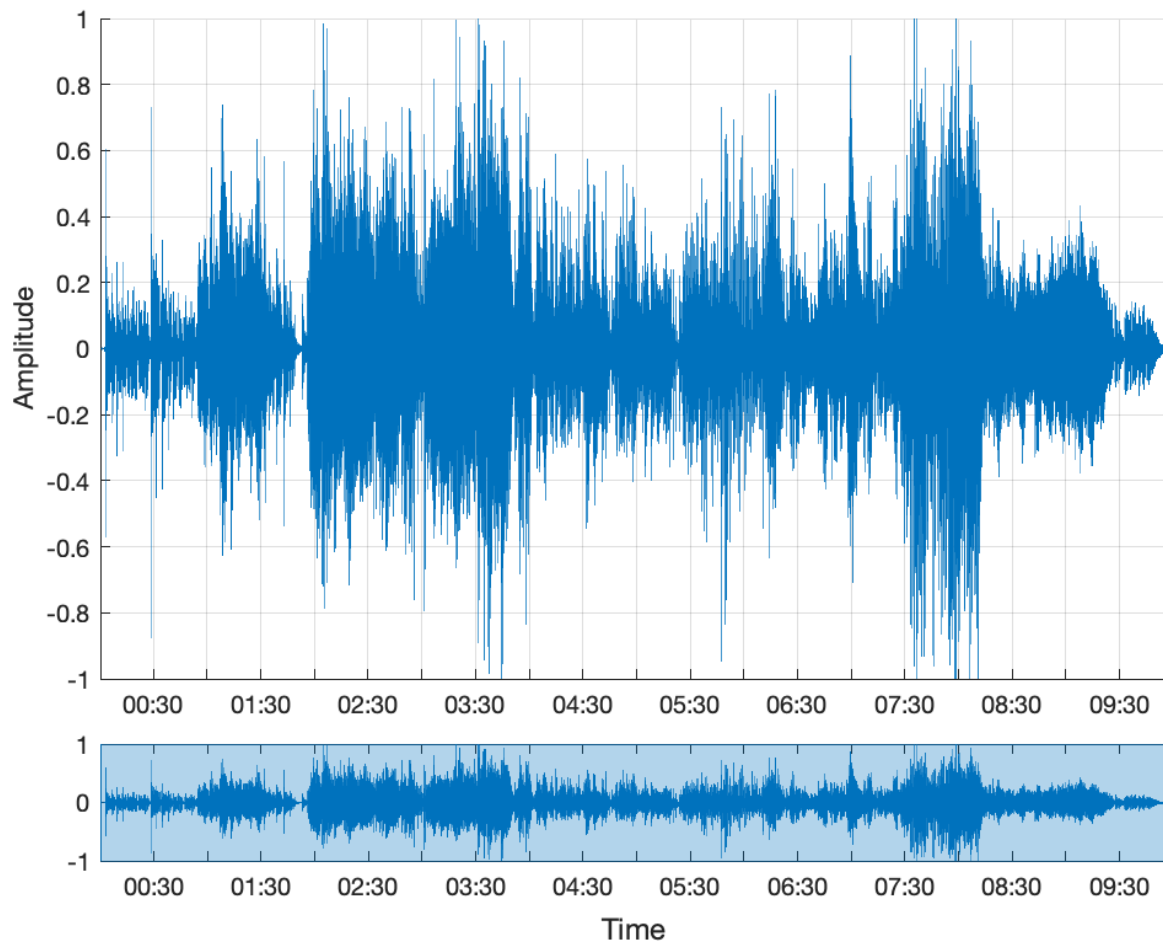
`audioplot` displays audio data using two axes with interactive features. The top axes has panning and zooming enabled along the x dimension to help examine a region of interest. The bottom axes displays a plot over the entire time range along with a light blue time window, which indicates the display range in the top axes.

`audioplot` defines the following public properties:

- `AudioSource` - A public and dependent property that stores the audio file name or a numeric array representing audio data.
- `SampleRate` - A public and dependent property that stores the sampling rate of the audio signal in hertz. This property is read-only when the `AudioSource` property is an audio file name.
- `DisplayLimits` - A public property that sets the limits of the top axes and the width of the time window in the bottom axes.
- `WaveformAxes` and `PannerAxes` - Read-only properties that store the axes objects.

Use `audioplot` to plot the 10-minute long audio file `SoftGuitar-44p1_mono-10mins.ogg`.

```
filename = "SoftGuitar-44p1_mono-10mins.ogg";  
ap = audioplot(filename);
```



See Also
audioEnvelope

Audio Event Classification Using TensorFlow Lite on Raspberry Pi

This example demonstrates audio event classification using a pretrained deep neural network, YAMNet, from TensorFlow™ Lite library on Raspberry Pi™. You load the TensorFlow Lite model and predict the class for the given audio frame on Raspberry Pi using a processor-in-the-loop (PIL) workflow. To generate code on Raspberry Pi, you use Embedded Coder®, MATLAB® Support Package for Raspberry Pi Hardware and Deep Learning Toolbox Interface for TensorFlow Lite. Refer to Audio Classification and yamnet classification for more details on the YAMNet model description.

Third-Party Prerequisites

- Raspberry Pi hardware
- TensorFlow Lite library (on the target ARM® hardware)
- Pretrained TensorFlow Lite Model

Download YAMNet

Download and unzip the yamnet.

```
component = "audio";
filename = "yamnet.zip";
localfile = matlab.internal.examples.downloadSupportFile(component, filename);
downloadFolder = fileparts(localfile);
if exist(fullfile(downloadFolder, "yamnet"), "dir") ~= 7
    unzip(localfile, downloadFolder)
end
addpath(fullfile(downloadFolder, "yamnet"))
```

Read Audio Data and Classify the Sounds

Use `audioread` to read the audio file data and listen to it using `sound` function.

```
[audioIn, fs] = audioread("multipleSounds-16-16-mono-18secs.wav");
sound(audioIn, fs)
```

Call `classifySound` to detect the different sounds present in the given audio.

```
detectedSounds = classifySound(audioIn, fs)

detectedSounds = 1x5 string
    "Stream"    "Machine gun"    "Snoring"    "Bark"    "Meow"
```

You detected the different sounds in the pre-recorded audio in offline mode. The later sections of this example demonstrates the audio event classification in the real-time scenario where you process one audio frame at a time.

Load TensorFlow Lite Model and Audio Event Classes

You load the TFLite YAMNet using `loadTFLiteModel` (Deep Learning Toolbox). As mentioned in `TFLiteModel` (Deep Learning Toolbox) page, you set the `Mean` and `Variance` parameter of the TFLite model to 0 and 1, respectively, because the input to YAMNet is not already normalized.

```
modelName = "lite-model_yamnet_classification_tflite_1.tflite";
modelFullPath = fullfile(downloadFolder, "yamnet", modelName);
TFLiteYAMNet = loadTFLiteModel(modelFullPath);
TFLiteYAMNet.Mean = 0;
TFLiteYAMNet.StandardDeviation = 1;
```

Use `yamnetGraph` to load all the audio event classes supported by YAMNet, as an array of strings.

```
[~, audioEventClasses] = yamnetGraph;
```

Set the sample rate (in Hertz), the length of input audio frame and the frame duration in seconds, supported by YAMNet.

```
modelSamplingRate = 16000;
frameDimension = TFLiteYAMNet.InputSize{1};
frameLength = frameDimension(2);
frameDuration = frameLength/modelSamplingRate;
```

Set the `classificationRate` i.e. the number of classifications per second. As the number of hops per second must be equal to the classification rate, set the `hopDuration` to the reciprocal of `classificationRate`.

```
classificationRate = 10;
hopDuration = 1/classificationRate;
hopLength = floor(modelSamplingRate*hopDuration);
overlapLength = frameLength - hopLength;
```

Read Input Audio

You use dropdown control to list the different input audio files. Use `dsp.AudioFileReader` to read the audio file data.

```
afr = dsp.AudioFileReader();
audioInSamplingRate = afr.SampleRate;
audioFileInfo = audioinfo(afr.FileName);
```

Set the `SamplesPerFrame` corresponding to one hop.

```
audioInFrameLength = floor(audioInSamplingRate*hopDuration);
afr.SamplesPerFrame = audioInFrameLength;
```

Setup the FIFO Buffers

Create two `dsp.AsyncBuffer` objects `audioBufferYamnet` and `audioClassBuffer` to buffer the resampled audio samples and the indices of predicted audio classes. You set the length of the `audioClassBuffer` corresponding to `predictedAudioClassesDuration` seconds. You initialize the `audioClassBuffer` with the index corresponding to the `Silence` audio class.

```
predictedAudioClassesDuration = 1;
audioClassBufferLength = floor(predictedAudioClassesDuration*classificationRate);
audioClassBuffer = dsp.AsyncBuffer(audioClassBufferLength);
audioBufferYamnet = dsp.AsyncBuffer(2*frameLength);
indexOfSilenceAudioClass = find(audioEventClasses == "Silence");
write(audioClassBuffer, ones(audioClassBufferLength, 1)*indexOfSilenceAudioClass);
```

Create a `timescope` object to visualize the audio.


```
timeScope = timescope("SampleRate", modelSamplingRate, ...
    "YLimits", [-1 1], ...
    "Name", "Audio Event Classification Using TensorFlow Lite YAMNet", ...
    "TimeSpanSource", "Property", ...
    "TimeSpan", audioFileInfo.Duration);
```

Run TFLite YAMNet in MATLAB to Perform Audio Event Classification

Setup a `dsp.SampleRateConverter` system object to convert the sampling rate of the input audio to 16000 Hz, as YAMNet is trained using audio signals sampled at 16000 Hz sampling rate.

```
src = dsp.SampleRateConverter('InputSampleRate', audioInSamplingRate, ...
    'OutputSampleRate', modelSamplingRate, ...
    'Bandwidth', 10000);
```

You feed one audio frame at a time to represent the system as it would be deployed in a real-time embedded system. In the streaming loop, you first load one hop of audio samples and fed them to the `dsp.SampleRateConverter` to convert the sampling rate to 16000 Hz. The resampled frame is written in a FIFO buffer, `audioBufferYamnet`, you load the overlapping frames of length `frameLength` from this buffer and fed it to the YAMNet. The TensorFlow Lite YAMNet model outputs the predicted score vector that contains a score for each audio event class. You calculate the index of the maximum score in the score vector and write it in the FIFO buffer, `audioClassBuffer`. The predicted index is the statistical mode of the contents of the `audioClassBuffer`. The predicted audio event class is the value of `audioEventClasses` array at the predicted index. You visualize the resampled audio frame in the time scope and print the predicted audio event class as the title of the time scope.

```
while ~isDone(afr)
    audioInFrame = afr();
    resampledAudioInFrame = src(audioInFrame);
    write(audioBufferYamnet, resampledAudioInFrame);
    audioInYamnetFrame = read(audioBufferYamnet, frameLength, overlapLength);
    scoresTFLite = TFLiteYAMNet.predict(audioInYamnetFrame');
    [~, audioClassIndex] = max(scoresTFLite);
    write(audioClassBuffer, audioClassIndex);
    predictedSoundClass = audioEventClasses(mode(audioClassBuffer.peak(audioClassBufferLength)));
    timeScope(resampledAudioInFrame);
    timeScope.Title = char(predictedSoundClass);
    drawnow
end
hide(timeScope)
reset(timeScope)
reset(afr)
```

Prepare MATLAB Code for Deployment

You prepare a MATLAB function `predictAudioClassUsingYAMNET` that performs audio class prediction for the input audio frames. It buffers the indices of the predicted audio class in a FIFO buffer. The predicted audio class index is the statistical mode of the contents of this FIFO buffer.

```
type predictAudioClassUsingYAMNET.m
```

```
function predictedAudioClassIndex = predictAudioClassUsingYAMNET(audioIn, audioClassHistoryBuffer)
% predictAudioClassUsingYAMNET Predicts the audio class of input audio by
% using a pre-trained TensorFlow Lite YAMNET model.
%
% Input Arguments:
```

```
% audioIn - Audio frame of length 1x15600 with
%          sampling rate of 16000 samples per
%          second
% audioClassHistoryBufferLength - Length of the audio class FIFO buffer
%          to contain predicted audio class
%          indices. The index of the predicted
%          audio class is the statistical mode
%          of the contents of this buffer.
%
% Output Arguments:
% predictedAudioClassIndex - Index of the predicted audio class.
%
% Copyright 2022 The MathWorks, Inc.

%#codegen

persistent TFLiteYAMNETModel AudioClassBuffer

if isempty(TFLiteYAMNETModel)
    TFLiteYAMNETModel = loadTFLiteModel("lite-model_yamnet_classification_tflite_1.tflite");
    TFLiteYAMNETModel.NumThreads = 4;
    TFLiteYAMNETModel.Mean = 0;
    TFLiteYAMNETModel.StandardDeviation = 1;

    % Create and initialize a FIFO buffer with index of the 'Silence'
    AudioClassBuffer = dsp.AsyncBuffer(audioClassHistoryBufferLength);
    write(AudioClassBuffer,ones(audioClassHistoryBufferLength,1)*indexSilenceAudioClass);
end

scores = predict(TFLiteYAMNETModel, audioIn);
[~, audioClassIndex] = max(scores);
write(AudioClassBuffer, audioClassIndex);
predictedAudioClassHistory = peek(AudioClassBuffer, audioClassHistoryBufferLength);
predictedAudioClassIndex = mode(predictedAudioClassHistory);
end
```

Generate Code for Audio Event Classifier on Raspberry Pi

Create Code Generation Configuration

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

Set Up Connection with Raspberry Pi

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

```
if ~(exist("r", "var"))
    r = raspi("raspiname", "pi", "password");
end
```

Configure Code Generation Hardware Parameters for Raspberry Pi

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 Raspberry Pi.

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

Copy TensorFlow Lite Model to the Target Hardware and the Current Directory

Copy the TensorFlow Lite model to the Raspberry Pi board. On the hardware board, set the environment variable `TFLITE_MODEL_PATH` to the the location of the TensorFlow Lite model. For more information on setting environment variables, see “Prerequisites for Deep Learning with TensorFlow Lite Models” (Deep Learning Toolbox).

Use `putFile` method of the `raspi` object to copy the TFLite model to Raspberry Pi.

```
putFile(r,char(modelFullPath), '/home/pi')
```

Copy the model to the current directory as it is required by `codegen` (MATLAB Coder) during code generation.

```
copyfile(modelFullPath)
```

Generate PIL MEX

You use `coder.Constant` (MATLAB Coder) to make the constant input arguments, compile time constants in the generated code. Run the `codegen` (MATLAB Coder) command to generate a PIL MEX function `predictAudioClassUsingYAMNET_pil`.

```
codegen -config cfg predictAudioClassUsingYAMNET -args {ones(1,15600,"single"), coder.Constant(a
```

```
### Connectivity configuration for function 'predictAudioClassUsingYAMNET': 'Raspberry Pi'
```

Predict Audio Class on Raspberry Pi Using PIL Workflow

You call the generated PIL function `predictAudioClassUsingYAMNET_pil` to stream one audio frame at a time to represent the system as it would be deployed in a real-time embedded system.

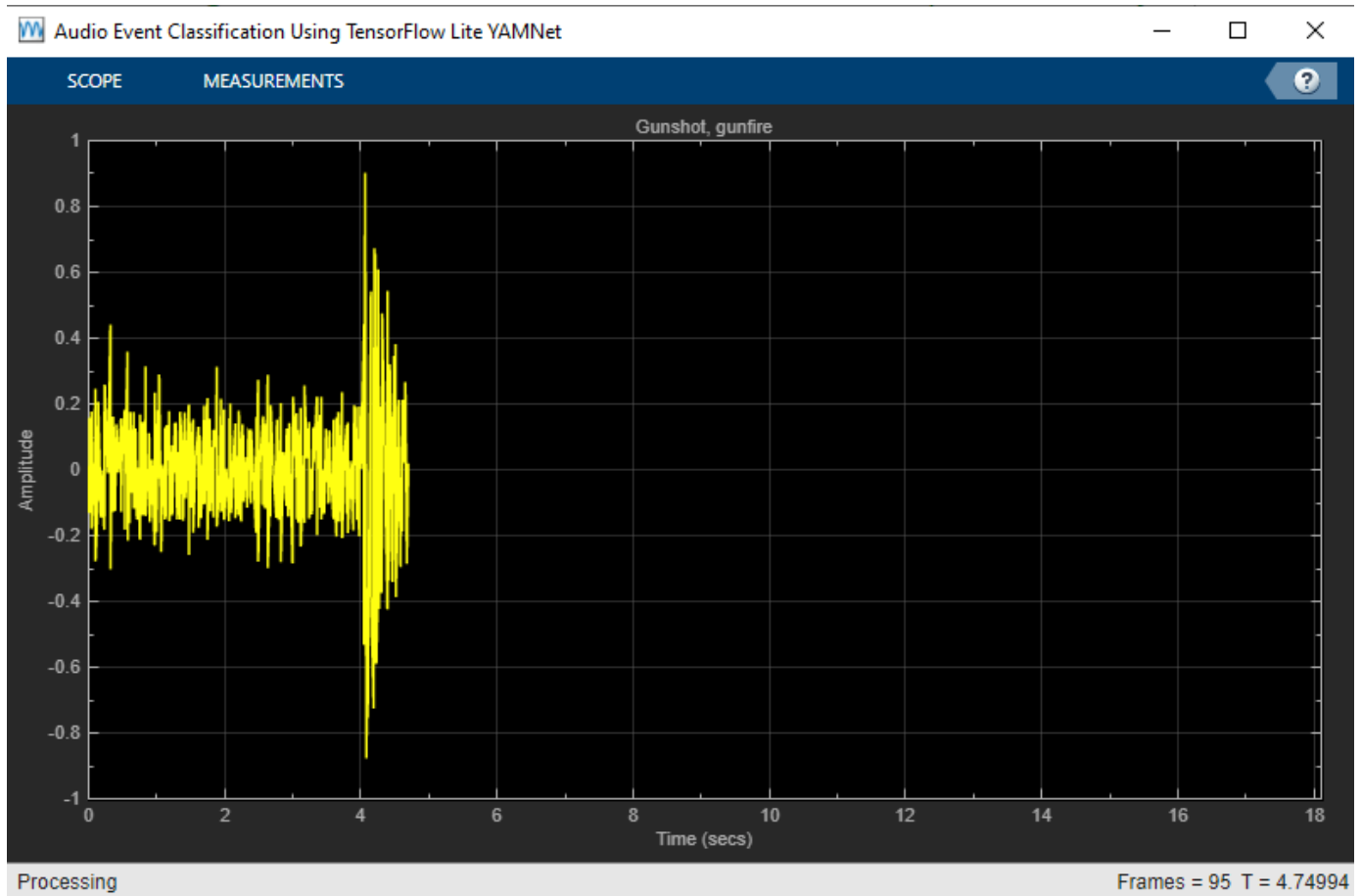
```
show(timeScope)
while ~isDone(afr)
    audioInFrame = afr();
    resampledAudioInFrame = src(audioInFrame);
    write(audioBufferYamnet,resampledAudioInFrame);
    audioInYamnetFrame = read(audioBufferYamnet,frameLength,overlapLength);
    predictedSoundClassIndex = predictAudioClassUsingYAMNET_pil(single(audioInYamnetFrame'),audi
    preditedSoundClass = audioEventClasses(predictedSoundClassIndex);
    timeScope(resampledAudioInFrame)
    timeScope.Title = char(preditedSoundClass);
    drawnow
end
```

```

### Starting application: 'codegen\lib\predictAudioClassUsingYAMNET\pil\predictAudioClassUsingYAMNET.pil'
To terminate execution: clear predictAudioClassUsingYAMNET_pil
### Launching application predictAudioClassUsingYAMNET.elf...

hide(timeScope)

```



Terminate the PIL execution

```
clear predictAudioClassUsingYAMNET_pil
```

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

Evaluate Raspberry Pi Execution Time

You use PIL workflow to profile the `predictAudioClassUsingYAMNET` function. You enable profiling in the code generation configuration and generate the PIL function that keeps a log of execution profile.

```

cfg.CodeExecutionProfiling = true;
codegen -config cfg predictAudioClassUsingYAMNET -args {ones(1,15600,"single"), coder.Constant(a
### Connectivity configuration for function 'predictAudioClassUsingYAMNET': 'Raspberry Pi'

```

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

```

numCalls = 100;
for k = 1:numCalls

```

```

x = pinknoise(1,15600,"single");
scores = predictAudioClassUsingYAMNET_pil(x, audioClassBufferLength, indexOfSilenceAudioClass)
end

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

```

Terminate the PIL execution.

```

clear predictAudioClassUsingYAMNET_pil

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

Execution profiling report: coder.profile.show(getCoderExecutionProfile('predictAudioClassUsingYAMNET_pil'));

```

Generate an execution profile report to evaluate execution time.

```

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

```

Code Execution Profiling Report for predictAudioClassUsingYAMNET

The code execution profiling report provides metrics based on data collected from a SIL or PIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. See [Code Execution Profiling](#) for more information.

1. Summary

Total time	2430.9361
Unit of time	ms
Command	report(executionProfile, 'Units', 'seconds', 'ScaleFactor', '0.001', 'NumericFormat', '%0.4f');
Timer frequency (ticks per second)	1e+09
Profiling data created	29-Dec-2022 15:18:12

2. Profiled Sections of Code

Section	Maximum Execution Time in ms	Average Execution Time in ms	Maximum Self Time in ms	Average Self Time in ms	Calls
predictAudioClassUsingYAMNET_initialize	0.0008	0.0008	0.0008	0.0008	1
predictAudioClassUsingYAMNET	32.9552	24.2937	32.9552	24.2937	100
predictAudioClassUsingYAMNET_terminate	1.5684	1.5684	1.5684	1.5684	1

3. Execution Times in Percentages [\[hide\]](#)

Section	Self Time / Caller Function	Self Time / Task	Self Time / Simulation	Function / Simulation
predictAudioClassUsingYAMNET_initialize	100%	100%	3.2128e-05%	3.2128e-05%
predictAudioClassUsingYAMNET	100%	100%	99.935%	99.935%
predictAudioClassUsingYAMNET_terminate	100%	100%	0.064518%	0.064518%

4. Definitions

Execution Time: Time between start and end of code section.

Self Time: Execution time, excluding time in child sections.

In the code execution profiling report, you find that the average execution time taken by `predictAudioClassUsingYAMNET` is 24.29 ms which is within the budget of 100 ms. You calculate the budget as the reciprocal of the classification rate. The performance is measured on Raspberry Pi 3 Model B Plus Rev 1.2.

Release buffers, timescope and other system objects used in the example.

```
release(audioBufferYamnet)
release(audioClassBuffer)
release(timeScope)
release(src)
release(afr)
```

Deploy Smart Speaker System on Raspberry Pi Using Simulink

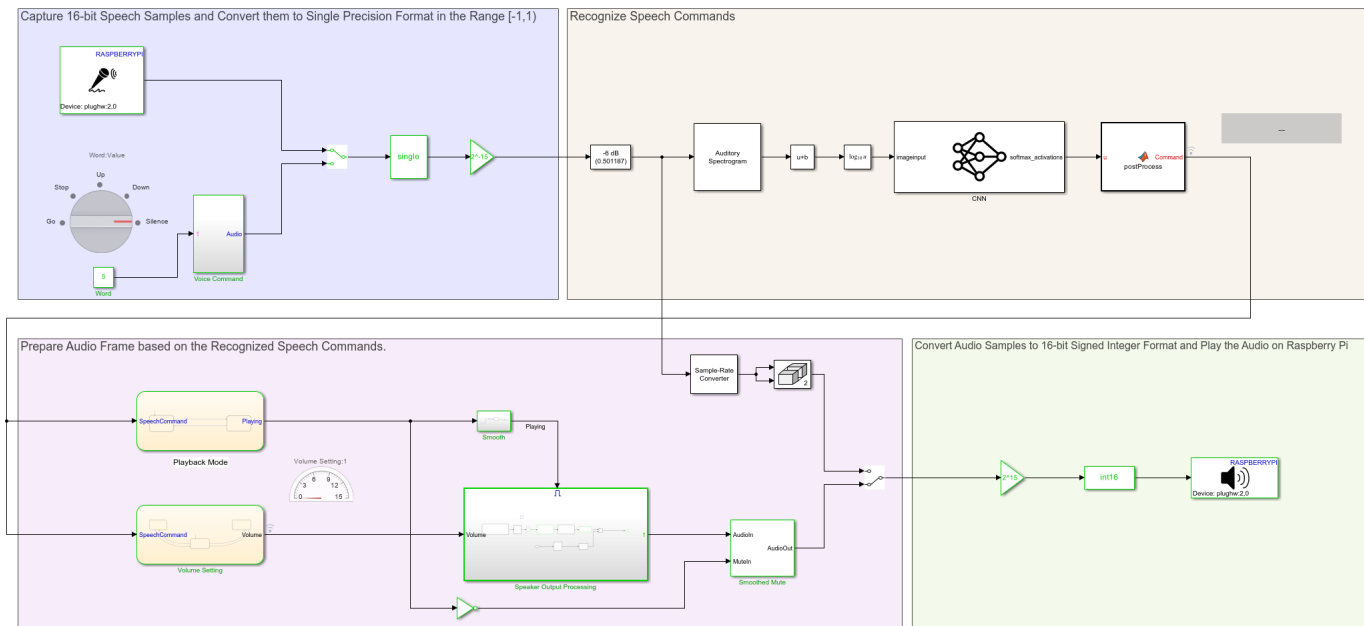
This example demonstrates how to deploy a smart speaker system on Raspberry Pi™ using Simulink®. A smart speaker is a speaker that can be controlled by your voice. You run the smart speaker Simulink model on Raspberry Pi in External Mode. The voice commands are captured through the USB microphone connected to your Raspberry Pi board. You can optionally input voice commands through the pre-recorded files. The smart speaker model plays the audio on the speaker connected to the Raspberry Pi. You make the smart speaker play music with the command "Go". You make it stop playing music by saying "Stop". You increase or decrease the music volume with the commands "Up" and "Down", respectively. For details about modeling the various modules used in the smart speaker model, see "Model Smart Speaker in Simulink" on page 1-948.

Smart Speaker Model

The model can be divided into four sub-modules that perform four sub-tasks

- 1 Capture 16-bit speech samples and convert them to single precision format in the range [-1,1)
- 2 Recognize speech commands
- 3 Prepare audio frame based on the recognized speech commands
- 4 Convert audio samples to 16-bit signed integer format and play the audio on Raspberry Pi

```
modelName = "AudioSmartSpeakerOnRaspberryPi";
open_system(modelName)
```



Copyright 2022, The MathWorks, Inc.

Configure Audio I/O Blocks

The smart speaker model uses the ALSA Audio Capture (Simulink Support Package for Raspberry Pi Hardware) block to capture the voice commands from a microphone connected to your Raspberry Pi board. The model uses the ALSA Audio Playback (Simulink Support Package for Raspberry Pi

Hardware) block to play the audio on a speaker connected to your Raspberry Pi board. The ALSA Audio IO blocks come with **Simulink Support Package for Raspberry Pi Hardware**. After connecting the microphone and speaker to your Raspberry Pi board, you list the audio capture and audio playback devices using `listAudioDevices` (Simulink Support Package for Raspberry Pi Hardware).

```
r = raspi("raspiname","pi","password");
audioCaptureDevicesList = listAudioDevices(r,"capture");
audioPlaybackDevicesList = listAudioDevices(r,"playback");
```

You set the **Device name** in the **ALSA Audio Capture:Block Parameters** dialog to the device of your choice from `audioCaptureDevicesList`. Similarly, you configure the **Device name** in the **ALSA Audio Playback:Block Parameters** dialog to the playback device of your choice from `audioPlaybackDevicesList`.

Display the details of an audio capture and audio playback device from `audioCaptureDevicesList` and `audioPlaybackDevicesList`.

```
audioCaptureDevicesList(1)
```

```
ans =
```

```
    struct with fields:
```

```
        Name: 'USB-Audio-LogitechUSBHeadsetH340-LogitechInc.LogitechUSBHeadsetH340atusb-0000'
        Device: '2,0'
        Channels: {'2'}
        BitDepth: {'16-bit integer'}
        SamplingRate: {'44100'}
```

```
audioPlaybackDevicesList(3)
```

```
ans =
```

```
    struct with fields:
```

```
        Name: 'USB-Audio-LogitechUSBHeadsetH340-LogitechInc.LogitechUSBHeadsetH340atusb-0000'
        Device: '2,0'
        Channels: {'2'}
        BitDepth: {'16-bit integer'}
        SamplingRate: {'44100'}
```

To use the above devices, you set the **Device name** in the **ALSA Audio Capture:Block Parameters** and **ALSA Audio Playback:Block Parameters** dialog to `plughw:2,0`. You set the **Audio sampling frequency (Hz)** to `16000` as the subsequent convolutional neural network (CNN) used to recognize voice commands was trained on a `16000` Hz sampling frequency.

The model provides a manual switch to switch audio from microphone to the pre-recorded audio files. You select the voice commands using the Rotary switch. The model uses four Audio File Read (Simulink Support Package for Raspberry Pi Hardware) blocks to read the audio files `go.wav`, `stop.wav`, `up.wav`, and `down.wav`. Note that Audio File Read (Simulink Support Package for Raspberry Pi Hardware) block is included in **Simulink Support Package for Raspberry Pi Hardware**.

Modify the Data Type of the Audio Samples

ALSA Audio Capture (Simulink Support Package for Raspberry Pi Hardware) and Audio File Read (Simulink Support Package for Raspberry Pi Hardware) blocks outputs 16-bit signed integers audio

samples with values in the interval of $[-2^{15}, 2^{15} - 1]$. You cast the output of these blocks output to single-precision data and multiply it by 2^{-15} to change the numerical range to $[-1, +1)$. Note that you are changing the numerical range because the subsequent blocks expect the audio in the range $[-1, +1)$.

The ALSA Audio Playback (Simulink Support Package for Raspberry Pi Hardware) block expects 16-bit signed integers as input, hence the output of the preceding block that prepares audio frame must be converted to 16-bit signed integers. The range of the floating-point audio frame samples is $[-1, +1)$. You multiply the floating-point audio frame samples by 2^{15} to bring the range to $[-2^{15}, 2^{15} - 1]$. After multiplying, you typecast the product to `int16` data type. These `int16` audio frame samples can be fed to ALSA Audio Playback (Simulink Support Package for Raspberry Pi Hardware) block. The `AudioSmartSpeakerOnRaspberryPi` model uses Gain (Simulink) block to multiply the audio samples by the constants 2^{-15} or 2^{15} . It uses Data Type Conversion (Simulink) block to typecast the audio samples to `single` or `int16`.

Configure Smart Speaker Model Settings and Run the Model in External Mode

Open the `AudioSmartSpeakerOnRaspberryPi` model, go to **MODELING** Tab and Click on **Model Settings** or press **Ctrl+E** to open the configuration parameters dialog.

- Select a solver that supports code generation. Set **Solver** to `auto` (Automatic solver selection) and **Solver type** to `Fixed-step`.
- Select **Code Generation** and set the **System Target File** to `ert.tlc` whose **Description** is `Embedded Coder`.
- Set the **Language** to `C++`, which will automatically set the **Language Standard** to `C++11 (ISO)`.
- In **Configuration > Hardware Implementation**, set the **Hardware board** to `Raspberry Pi` and enter your Raspberry Pi credentials in the **Board Parameters**.
- In the same window, set **External mode > Communication interface** to `XCP on TCP/IP`.
- Check **Signal logging** in **Configuration > Data Import/Export** to enable signal monitoring in External Mode.
- Go to the **Hardware** tab and click on **Monitor & Tune** to run the model in external mode.

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
end
```

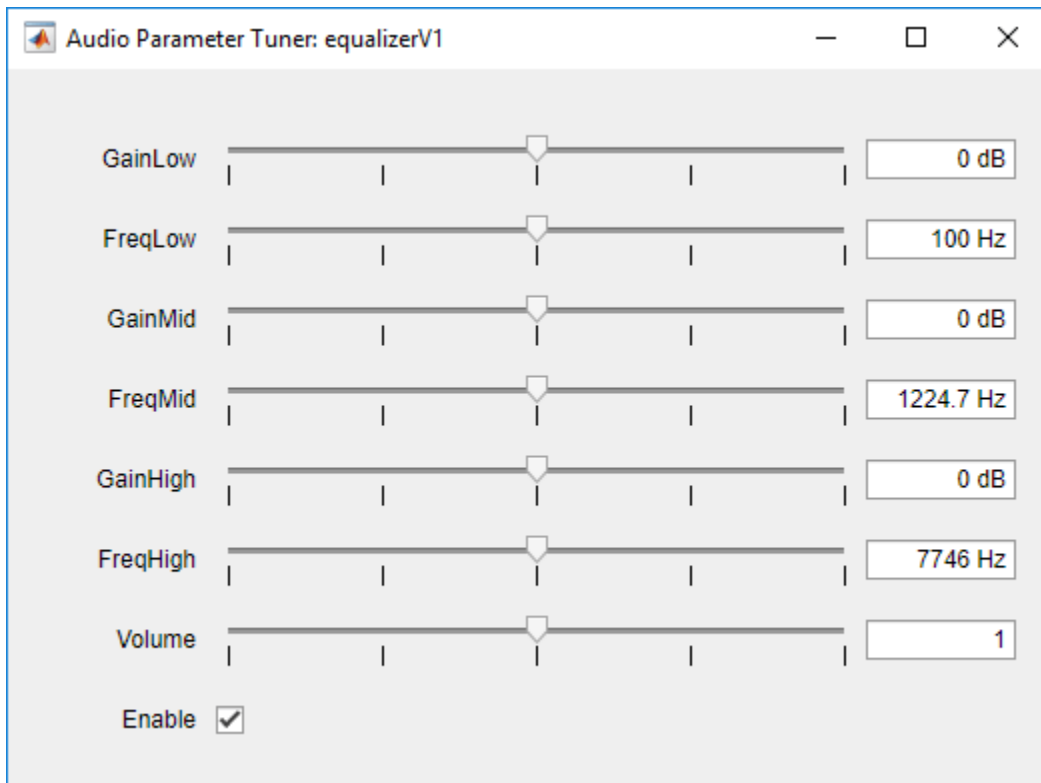
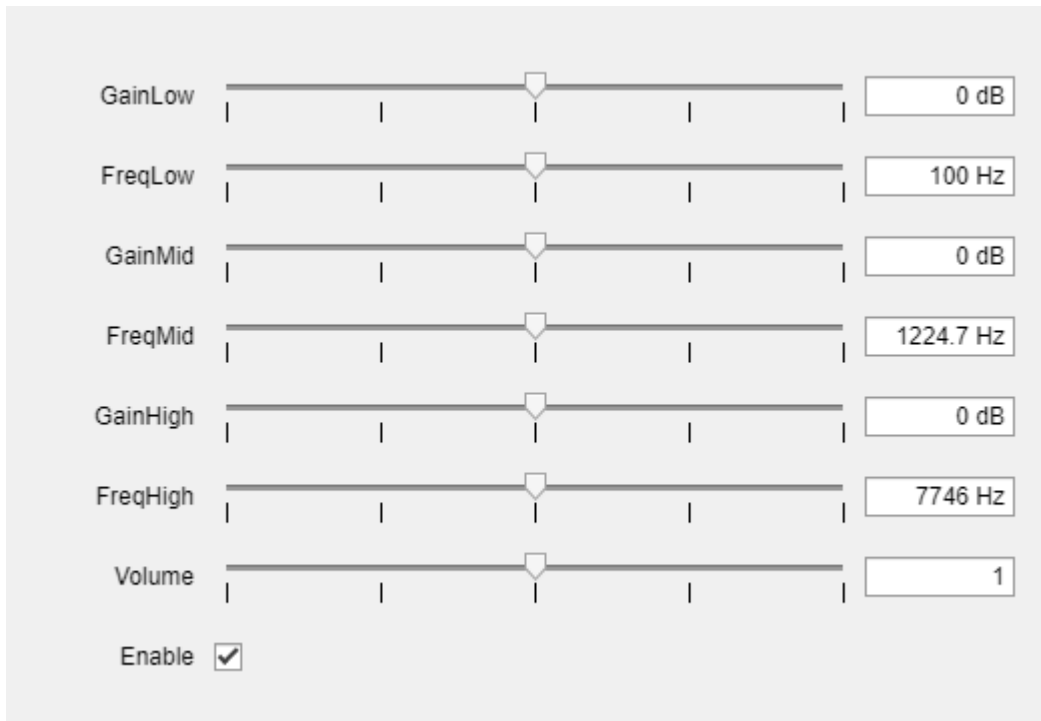
```

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)
```



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`.

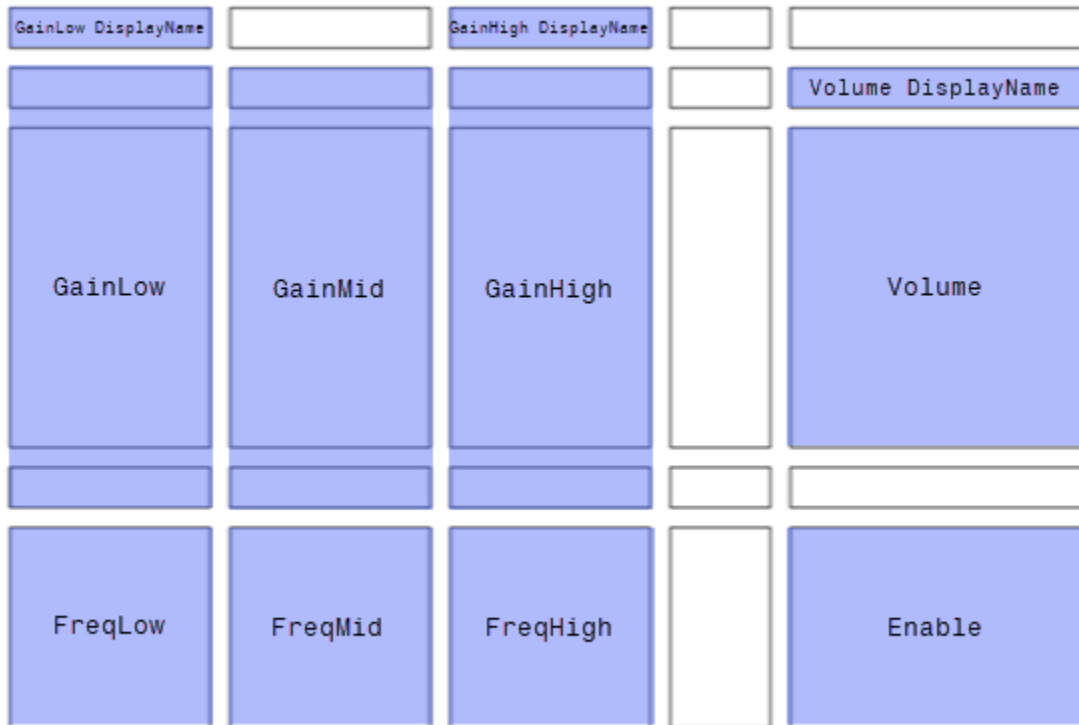
```
classdef equalizerV2 < audioPlugin
    ... % omitted for example purposes
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('GainLow', ...
                'Label','dB', ...
                'Mapping',{'lin',-20,20}, ...
                'Style','vslider', ...           %<--
                '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

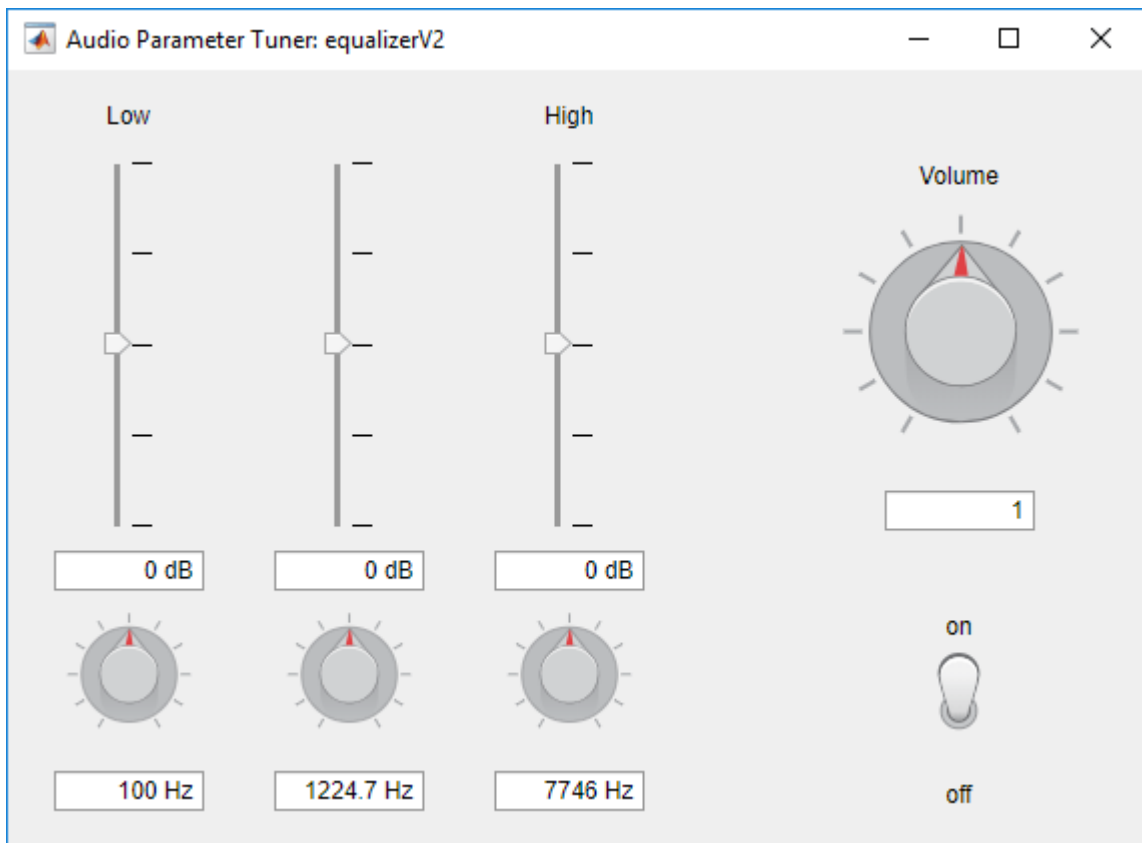
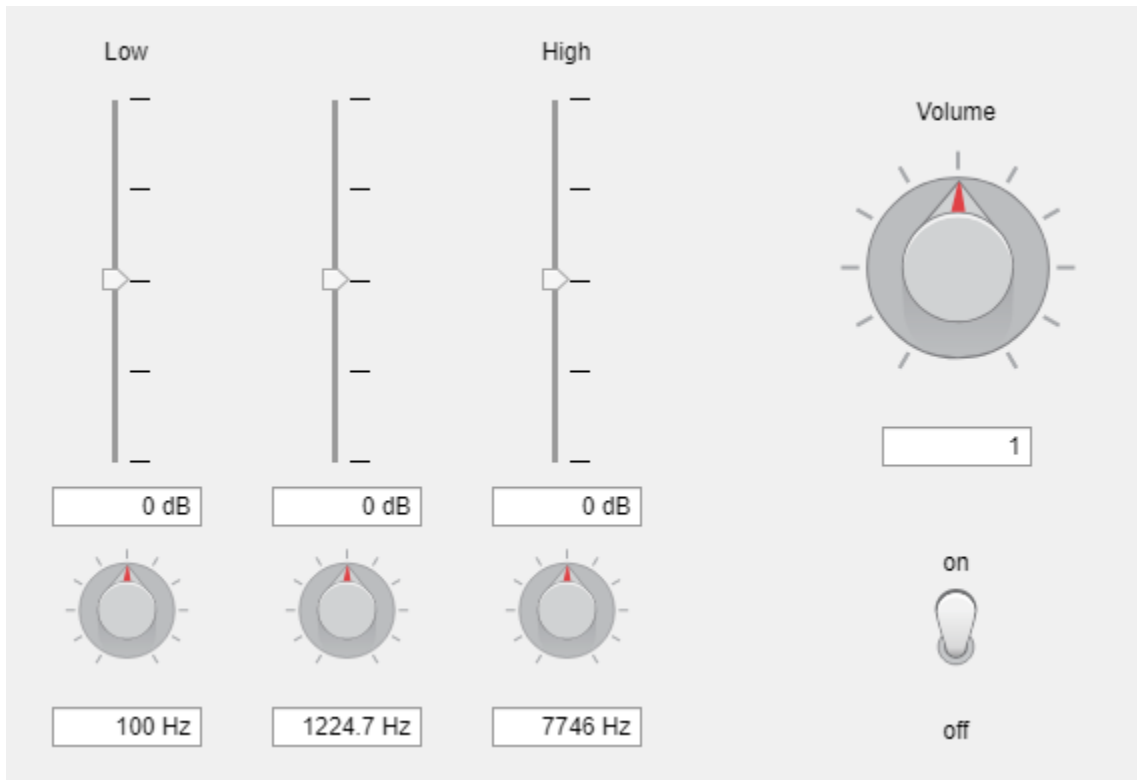
```

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)
```



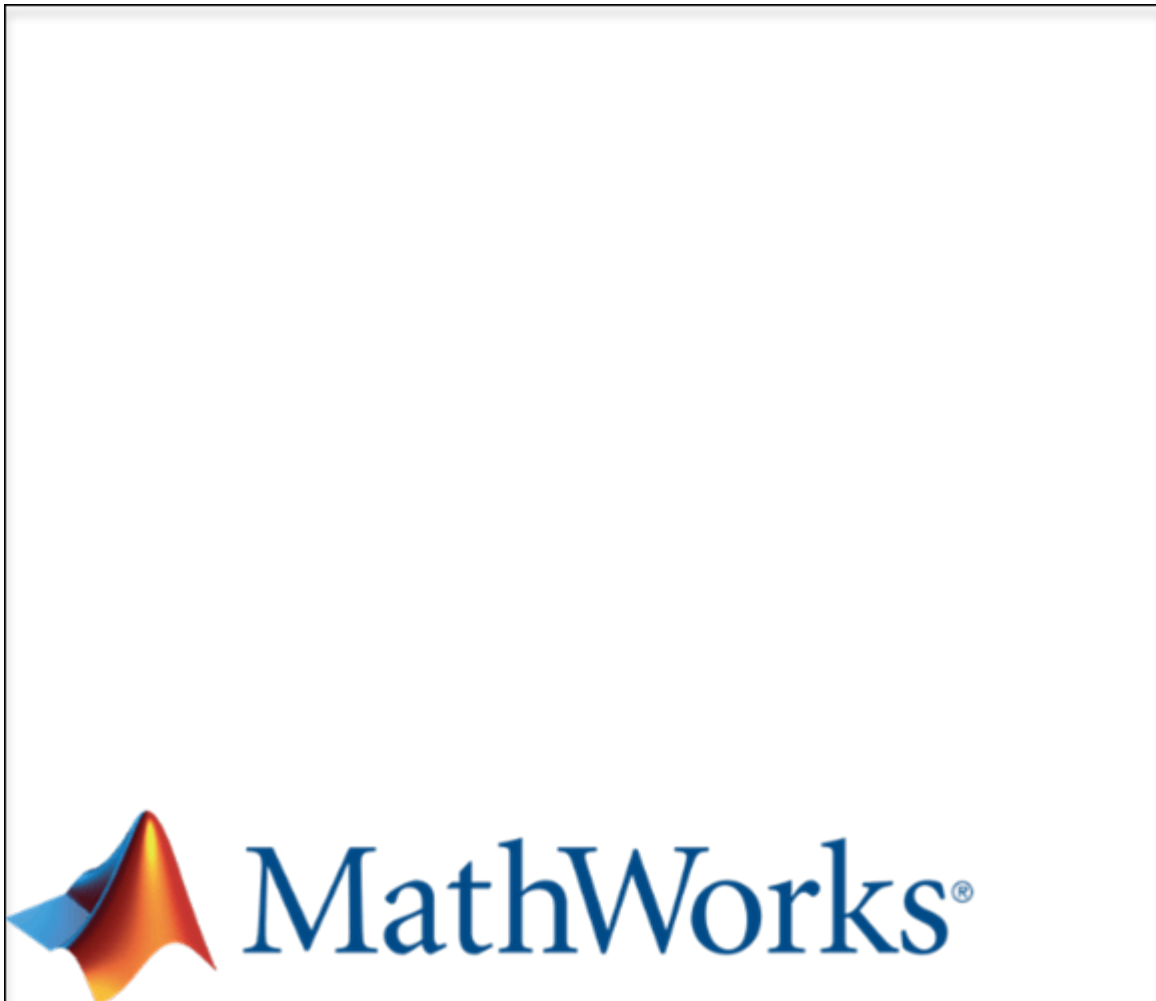
Background Image and Color

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. 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'), ...
```

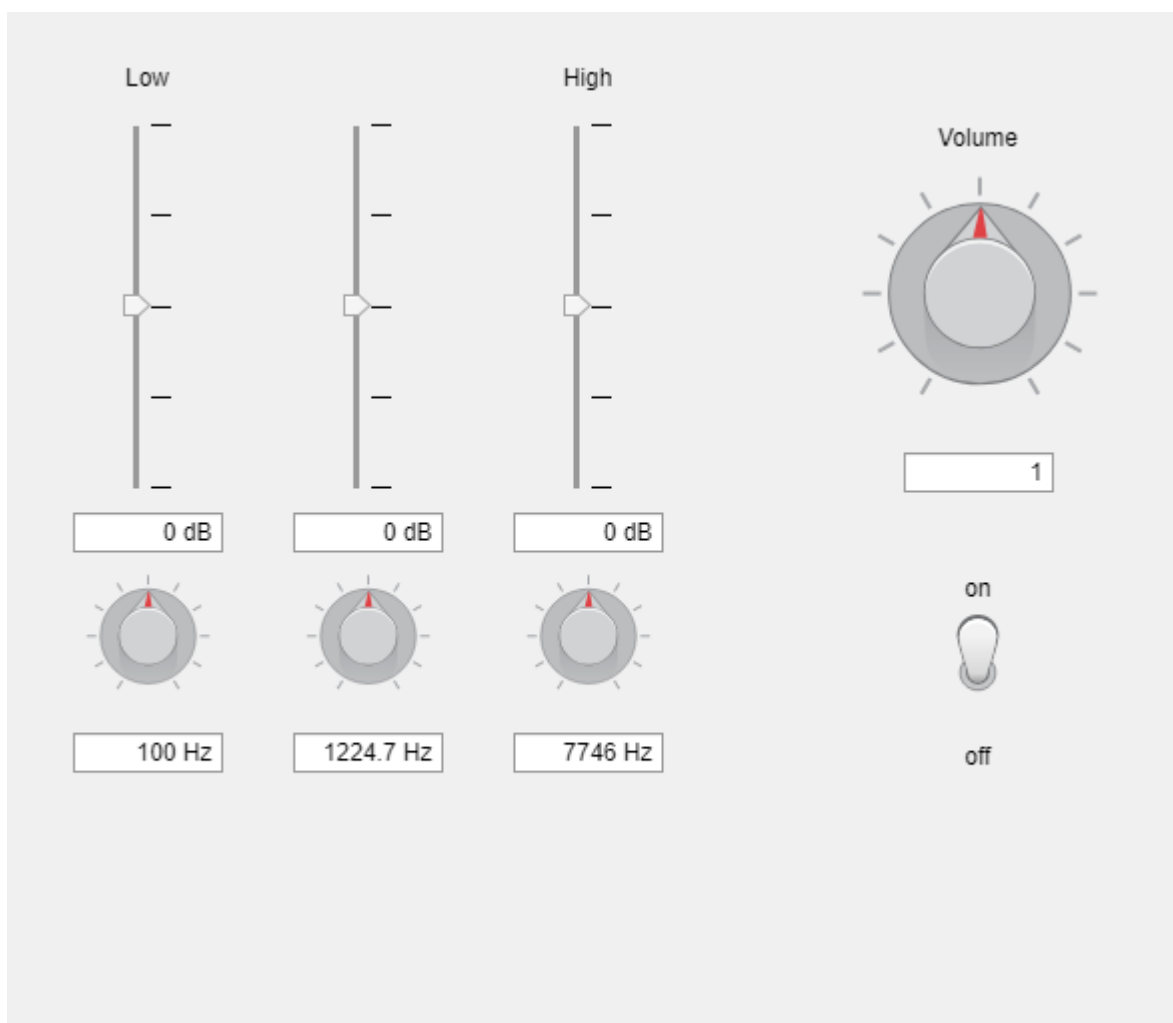
```

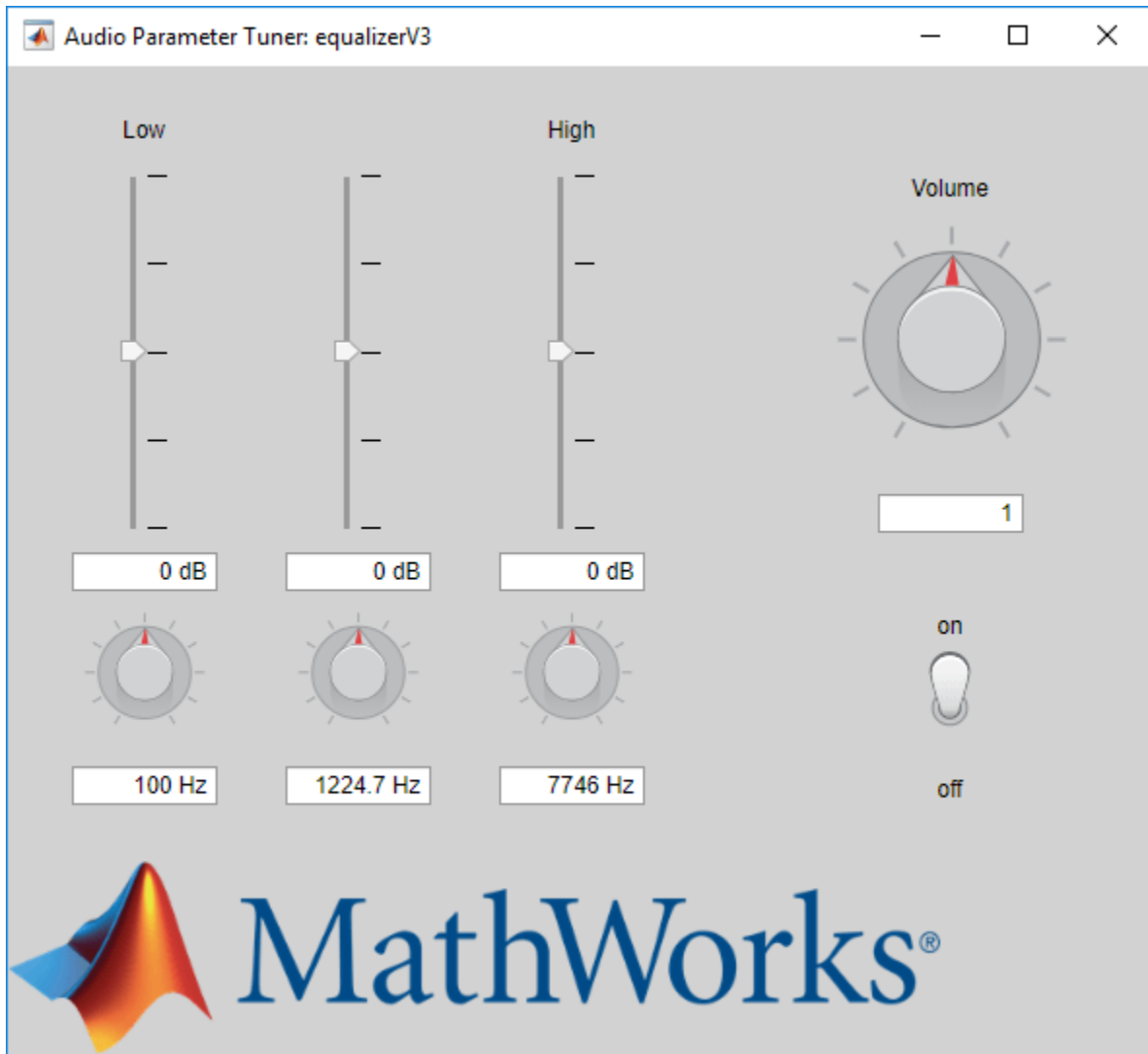
audioPluginParameter('FreqMid', ...
    'Label', 'Hz', ...
    'Mapping', {'log', 500, 3e3}, ...
    'Style', 'rotaryknob', ...
    'Layout', [5,2], ...
    'DisplayNameLocation', 'None'), ...
audioPluginParameter('GainHigh', ...
    'Label', 'dB', ...
    'Mapping', {'lin', -20, 20}, ...
    'Style', '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

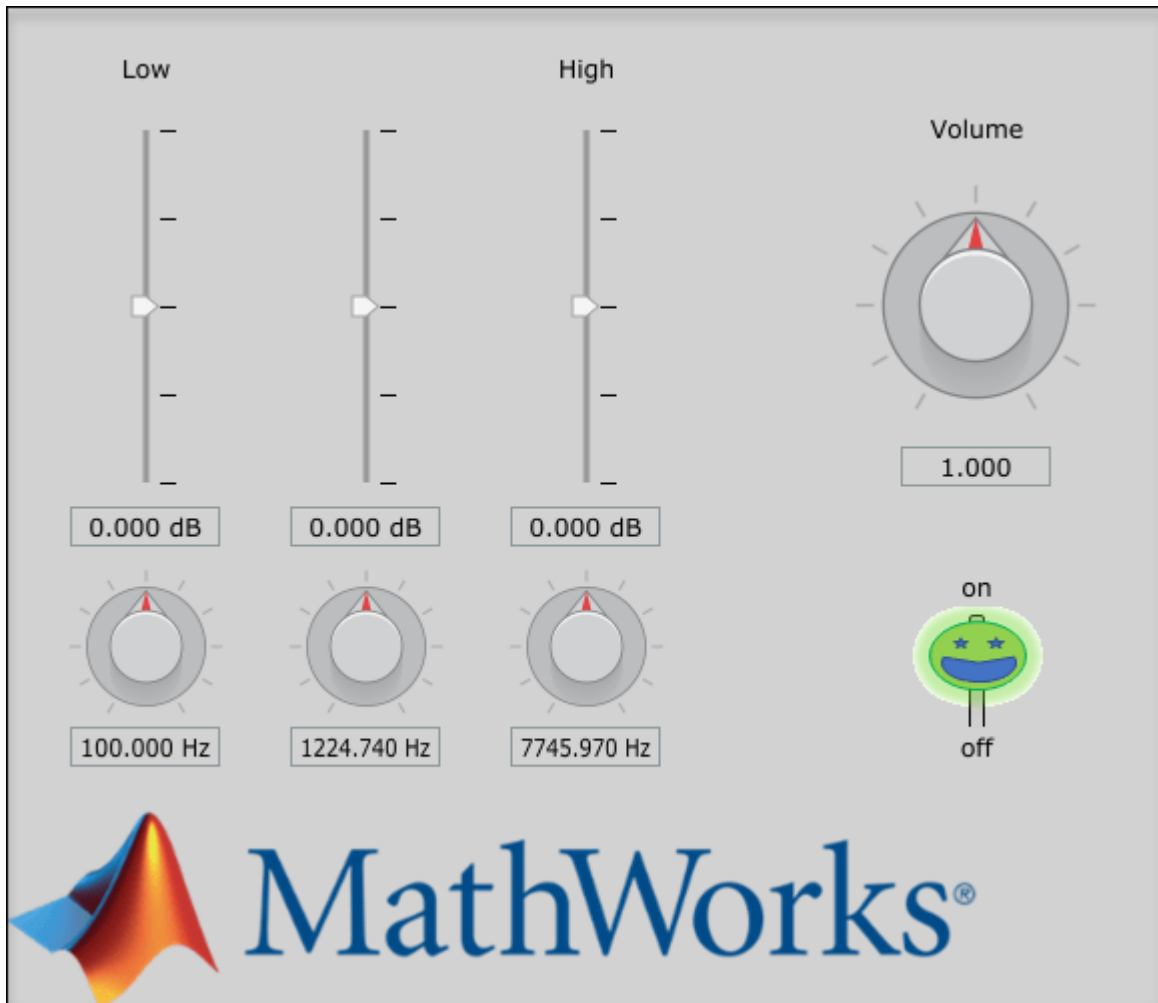
```

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” on page 3-2
- “Choose an App to Label Ground Truth Data” on page 3-13

Label Audio Using Audio Labeler

Note **Audio Labeler** will be removed in a future release. Use **Signal Labeler** instead.

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

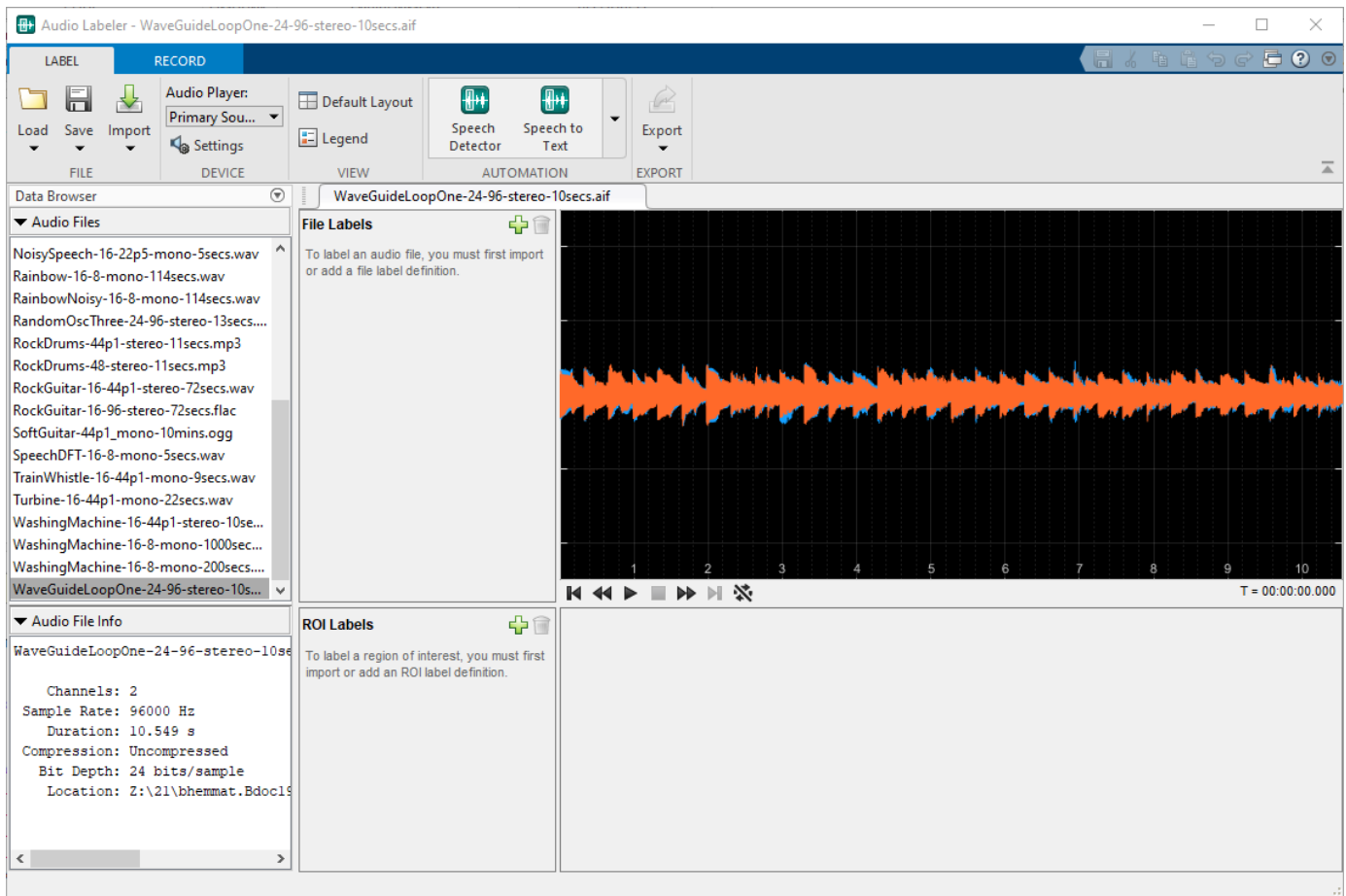
- 1 To open the **Audio Labeler**, at the MATLAB command prompt, enter:

```
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')
```

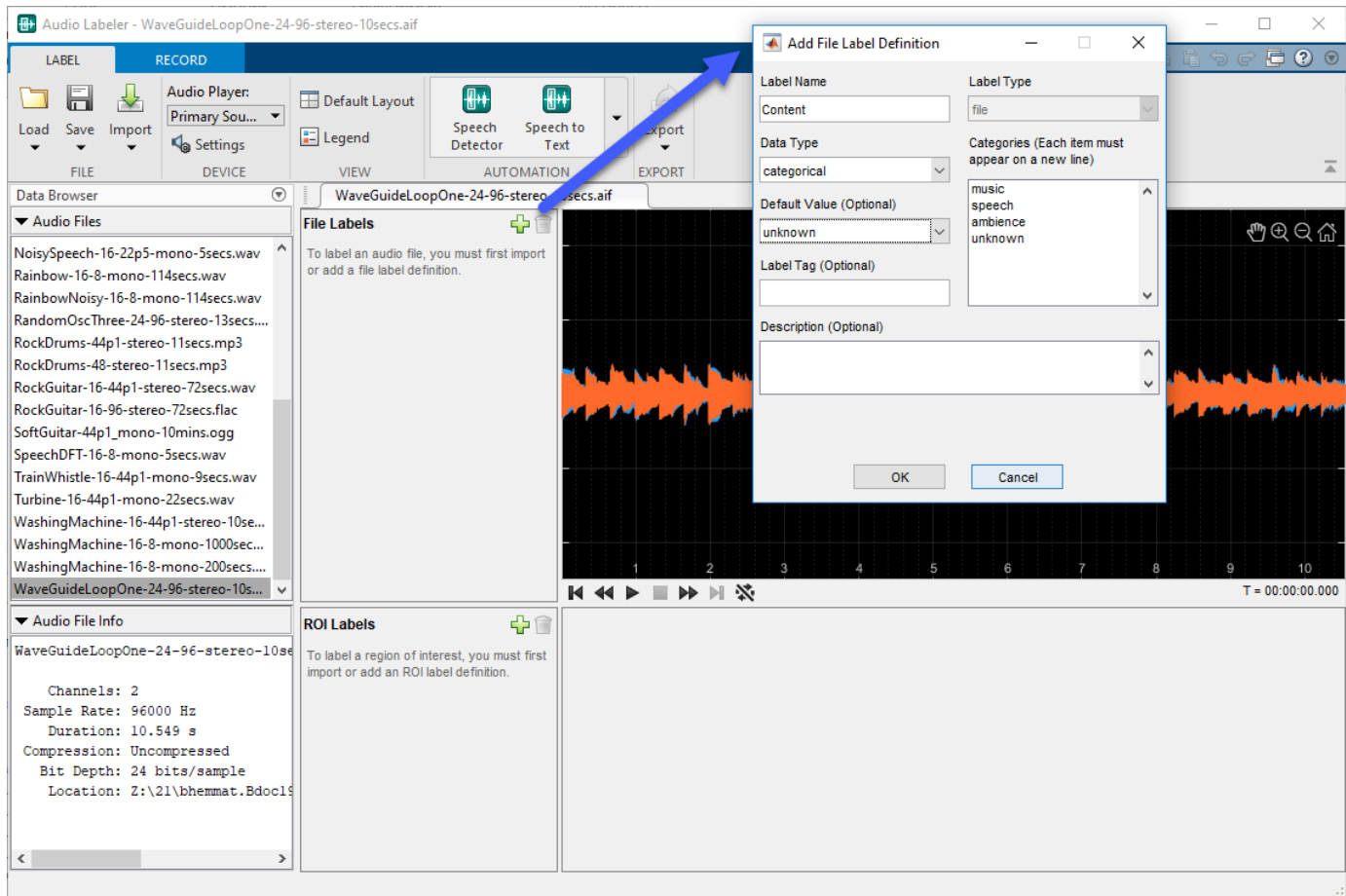
To load audio from a file, click **Load** > **Audio Folders** and select the folder containing audio files you want to label.



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.

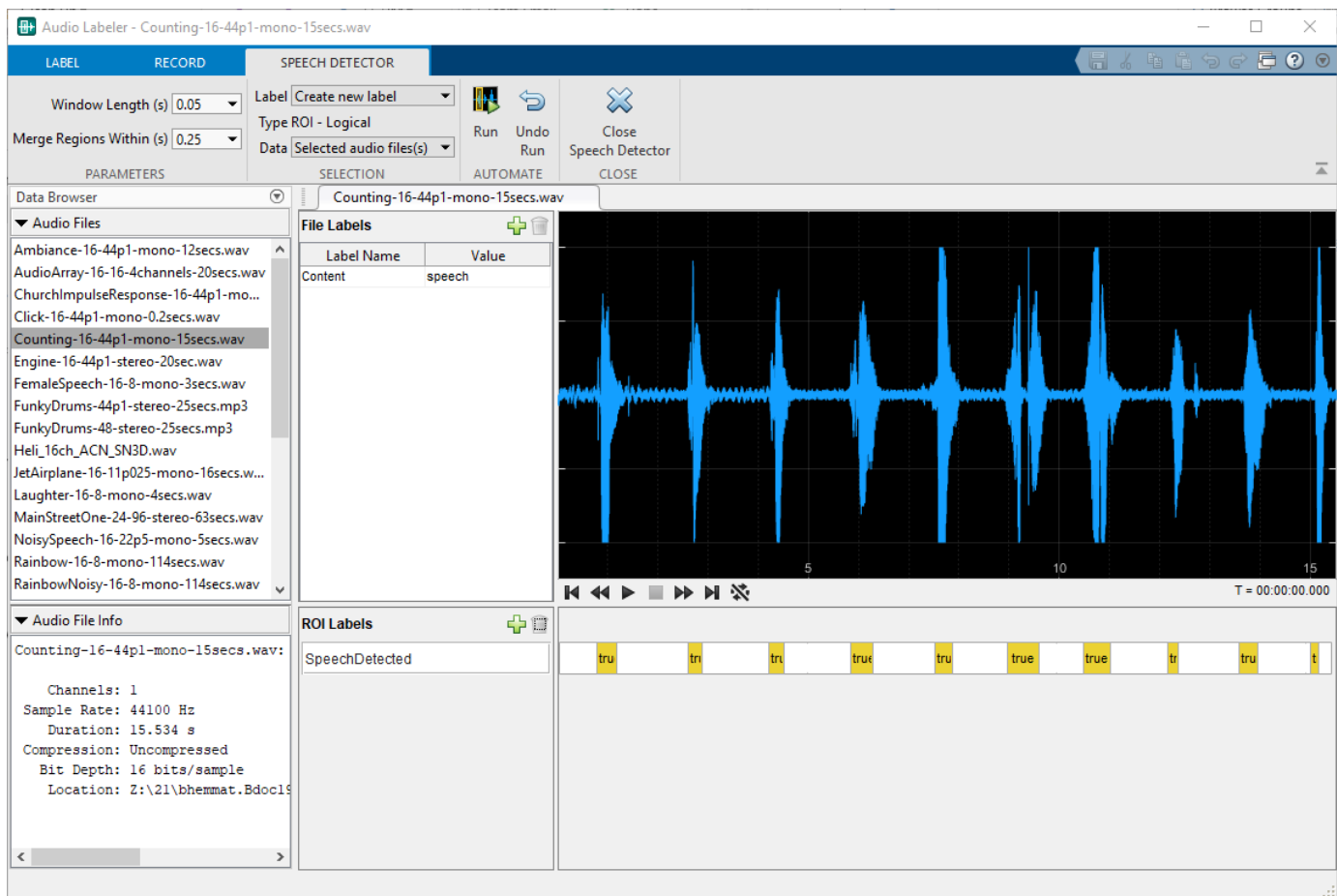
Region-Level Labels


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.

The screenshot displays the Audio Labeler application window. The interface is divided into several sections:

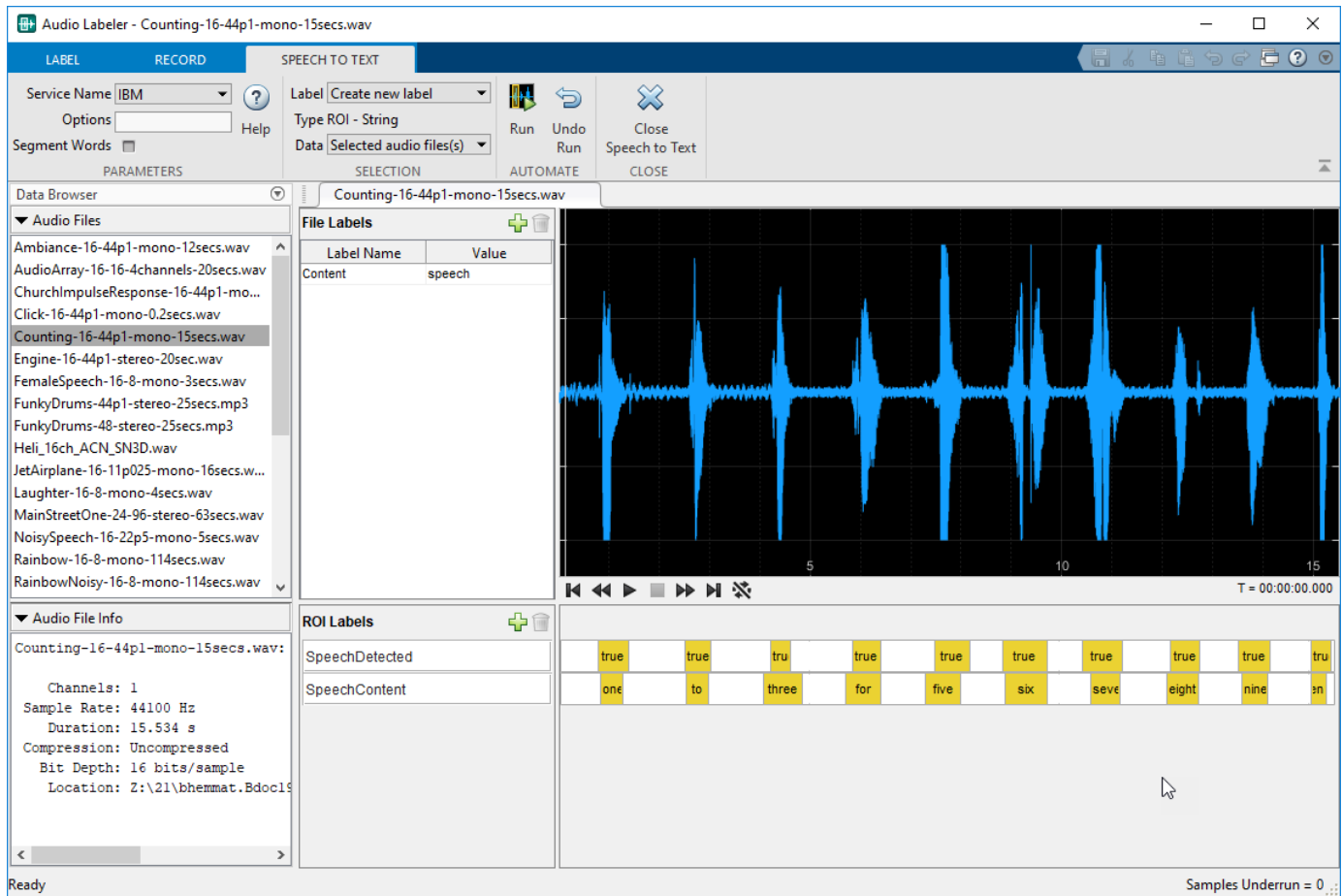
- Top Bar:** Includes tabs for 'LABEL' and 'RECORD', and a menu bar with options like 'Load', 'Save', 'Import', 'Settings', 'Default Layout', 'Legend', 'Speech Detector', 'Speech to Text', and 'Export'.
- Data Browser:** A list of audio files on the left, with 'Counting-16-44p1-mono-15secs.wav' selected.
- File Labels:** A table showing the current file's label:

Label Name	Value
Content	speech
- Audio Waveform:** A central visualization of the audio signal with a blue waveform and a yellow vertical line indicating the current playback position. Time markers are visible at 2.5988, 3.1336, 5, 10, and 15 seconds.
- Audio File Info:** Metadata for the selected file:



```
Counting-16-44p1-mono-15secs.wav:
  Channels: 1
  Sample Rate: 44100 Hz
  Duration: 15.534 s
  Compression: Uncompressed
  Bit Depth: 16 bits/sample
  Location: Z:\21\bhemmat.Bdoc15
```
- ROI Labels:** A row of labels indicating detected speech segments, shown as 'true' values in yellow boxes:

SpeechDetected	true	true	tru	true	tru	true	true	tru	tru	t
----------------	------	------	-----	------	-----	------	------	-----	-----	---

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.



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.

The screenshot shows the Audio Labeler software interface. The main window is titled "Audio Labeler - Counting-16-44p1-mono-15secs.wav". The interface is divided into several sections:

- Top Bar:** Includes "LABEL", "RECORD", and "SPEECH TO TEXT" tabs. Below these are buttons for "Run", "Undo Run", "Close Speech to Text", and "Close".
- Left Panel:** Contains a "Data Browser" with a list of audio files, including "Counting-16-44p1-mono-15secs.wav" which is selected. Below it is "Audio File Info" for the selected file.
- Center:** A waveform viewer showing a blue waveform on a black background. A red ROI bar is visible. A blue arrow points from the "Add ROI Label Definition" dialog box to the waveform.
- Right Panel:** A "File Labels" table with columns "Label Name" and "Value". Below it is an "ROI Labels" table.
- Bottom Right:** A "Ready" status bar and "Samples Underrun = 0".

The "Add ROI Label Definition" dialog box is open, showing the following fields:

- Label Name: VUV
- Label Type: roi
- Data Type: categorical
- Categories (Each item must appear on a new line):

voiced
unvoiced
- Default Value (Optional):
- Label Tag (Optional):
- Description (Optional):

The "ROI Labels" table shows the following data:

Label Name	Value
SpeechDetected	true true tru true true true true true true tru
SpeechContent	one two thr four five six seven eight nine ten

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. The main window displays an audio waveform for the file "Counting-16-44p1-mono-15secs.wav". The waveform is blue on a black background, with time markers at 0.76075, 1.3992, 2, 2.5, 3, 3.5, 4, and 4.5 seconds. The interface is divided into several panels:

- Top Panel:** Includes tabs for LABEL, RECORD, and SPEECH TO TEXT. It features a Service Name dropdown (set to IBM), a Label dropdown (set to "Create new label"), and a Type ROI dropdown (set to "String"). There are also buttons for Run, Undo Run, Close, and Speech to Text.
- Left Panel:** A Data Browser showing a list of audio files. The file "Counting-16-44p1-mono-15secs.wav" is selected. Below it is the "Audio File Info" section, which displays metadata 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.
- Right Panel:** A "File Labels" table with columns "Label Name" and "Value". It contains one row: "Content" with the value "speech". Below this is the "ROI Labels" section, which shows a timeline with three yellow boxes labeled "one", "two", and "three". A blue bar labeled "VUV" spans the duration of these three words. A dropdown menu is open over the "VUV" bar, showing "voiced" (highlighted in blue) and "unvoiced" options.

At the bottom of the window, the status bar shows "Ready" on the left and "Samples Underrun = 0" on the right.

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.

The screenshot shows the Audio Labeler software interface. The main window displays an audio waveform for the file "Counting-16-44p1-mono-15secs.wav". The waveform is overlaid with a blue signal and a yellow vertical bar indicating a detected speech segment. The time axis ranges from 0.5 to 4.6655 seconds.

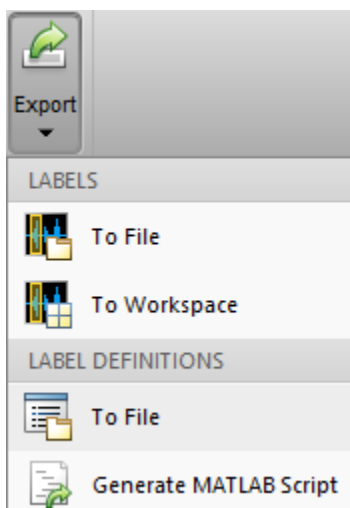
Below the waveform, there is a table of ROI Labels:

Label Name	Value
SpeechDetected	true
SpeechContent	one
VUV	voiced

The interface also includes a Data Browser on the left with a list of audio files, and a File Labels section with a table of labels and values.

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'  
        ... 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 =
```

```
4x2 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:

- “Train Speech Command Recognition Model Using Deep Learning” on page 1-332
- “Speaker Identification Using Pitch and MFCC” on page 1-237
- “Denoise Speech Using Deep Learning Networks” on page 1-312

See Also

Apps

Signal Labeler

Objects

`signalLabelDefinition` | `labeledSignalSet` | `audioDatastore` | `audioDeviceReader` | `audioDeviceWriter`

Choose an App to Label Ground Truth Data

You can use Computer Vision Toolbox™, Automated Driving Toolbox™, Lidar Toolbox™, Audio Toolbox, Signal Processing Toolbox™, and Medical Imaging Toolbox™ apps to label ground truth data. Use this labeled data to validate or train algorithms such as image classifiers, object detectors, semantic segmentation networks, instance segmentation networks, and deep learning applications. The choice of labeling app depends on several factors, including the supported data sources, labels, and types of automation.

One key consideration is the type of data that you want to label.

- If your data is an image collection, use the **Image Labeler** app. An image collection is an unordered set of images that can vary in size. For example, you can use the app to label images of books for training a classifier. The **Image Labeler** can also handle very large images (at least one dimension >8K).
- If your data is a single video or image sequence, use the **Video Labeler** app. An image sequence is an ordered set of images that resembles a video. For example, you can use this app to label a video or image sequence of cars driving on a highway for training an object detector.
- If your data includes multiple time-overlapped signals, such as videos, image sequences, or lidar signals, use the **Ground Truth Labeler** app. For example, you can label data for a single scene captured by multiple sensors mounted on a vehicle.
- If your data is only a lidar signal, use the **Lidar Labeler**. For example, you can use this app to label data captured from a point cloud sensor.
- If your data consists of single-channel or multichannel one-dimensional signals, use the **Signal Labeler**. For example, you can label biomedical, speech, communications, or vibration data. You can also use **Signal Labeler** to perform audio-specific tasks, such as speech detection and speech-to-text transcription.
- If your data is a 2-D medical image or image series, or a 3-D medical image volume, use the **Medical Image Labeler**. For example, you can label computed tomography (CT) image volumes of the chest to train a semantic segmentation network.

This table summarizes the key features of the labeling apps.

Labeling App	Data Sources	Label Support	Automation	Additional Features
Image Labeler	<ul style="list-style-type: none"> • Image collections • Very large images (at least one dimension >8K) 	<ul style="list-style-type: none"> • Rectangle regions of interest (ROIs) • Projected cuboid (ROIs) • Line ROIs • Pixel ROIs • Polygon ROIs • Point ROIs • Sublabels • Attributes • Scenes 	<ul style="list-style-type: none"> • Built-in automation algorithms • Custom automation algorithms • Blocked image automation algorithms 	<ul style="list-style-type: none"> • View visual summary of labeled data

Labeling App	Data Sources	Label Support	Automation	Additional Features
Video Labeler	<ul style="list-style-type: none"> • Videos • Image sequences • Custom image data sources 	<ul style="list-style-type: none"> • Rectangle ROIs • Projected cuboid (ROIs) • Line ROIs • Pixel ROIs • Polygon ROIs • Point ROIs • Sublabels • Attributes • Scenes 	<ul style="list-style-type: none"> • Built-in automation algorithms • Custom automation algorithms • Temporal automation algorithms 	<ul style="list-style-type: none"> • View visual summary of labeled data
Ground Truth Labeler	<ul style="list-style-type: none"> • Videos • Image sequences • Custom image data sources • Point cloud sequences (PCD or PLY files) • Velodyne® lidar files • Rosbags (requires ROS Toolbox) 	<ul style="list-style-type: none"> • Rectangle ROIs • Projected cuboid (ROIs) • Cuboid ROIs • Line ROIs • Pixel ROIs • Polygon ROIs • Point ROIs • Sublabels • Attributes • Scenes 	<ul style="list-style-type: none"> • Built-in automation algorithms, including vehicle and lane detection algorithms and a point cloud temporal interpolation algorithm • Custom automation algorithms • Temporal automation algorithms • Multisignal automation 	<ul style="list-style-type: none"> • View visual summary of labeled data • Connect external tool to app for displaying time-synchronized signals, such as lidar or CAN bus data • Customize loading interface to support additional data sources

Labeling App	Data Sources	Label Support	Automation	Additional Features
Lidar Labeler	<ul style="list-style-type: none"> Point cloud sequences (PCD or PLY files) Velodyne lidar files LAS/LAZ file sequences Rosbags (requires ROS Toolbox) 	<ul style="list-style-type: none"> Cuboid ROIs Attributes Scenes 	<ul style="list-style-type: none"> Built-in automation algorithms, including a lidar object tracker and point cloud temporal interpolator Custom automation algorithms Temporal automation algorithms 	<ul style="list-style-type: none"> View the cuboid labels in top, side, and front views Save and reuse custom camera views Connect to external tool to display time-synchronized signals for ease of labeling, such as videos, to use as a reference while labeling
Signal Labeler	<ul style="list-style-type: none"> Numeric arrays, MATLAB timetables, and labeledSignalSet objects in the MATLAB workspace MAT-files and CSV files Audio files (WAVE, OGG, FLAC, AU, AIFF, AIFC, MP3, MPEG-4 AAC) 	<ul style="list-style-type: none"> Time-based ROIs Time-based ROI features Time-based points Attributes Attribute features File-level labels Sublabels 	<ul style="list-style-type: none"> Built-in peak labeling Built-in feature extraction Custom automation algorithms Speech detection Speech-to-text transcription (requires Audio Toolbox extended functionality for speech2text) 	<ul style="list-style-type: none"> Expand, collapse, and browse details of labeled data View signal spectra and spectrograms Label ROIs and points using the spectrogram Label signals in bulk Use Label Viewer to view and compare labels Audio playback Inspect audio file information Export extracted features to Classification Learner

Labeling App	Data Sources	Label Support	Automation	Additional Features
Medical Image Labeler	<ul style="list-style-type: none"> • 2-D medical images and image series (DICOM or NIFTI files) • 3-D medical image volume (DICOM, NIFTI, or NRRD files) 	<ul style="list-style-type: none"> • Pixel ROIs 	<ul style="list-style-type: none"> • Built-in automation algorithms • Custom automation algorithms 	<ul style="list-style-type: none"> • View 3-D medical images in the coronal, sagittal, and transverse anatomical planes • View 3-D medical images using customizable volume rendering • Label multiple related images or image volumes in one app session

See Also

More About

- “Get Started with the Image Labeler” (Computer Vision Toolbox)
- “Get Started with the Video Labeler” (Computer Vision Toolbox)
- “Get Started with Ground Truth Labelling” (Automated Driving Toolbox)
- “Get Started with the Lidar Labeler” (Lidar Toolbox)
- “Using Signal Labeler App”
- “Label Spoken Words in Audio Signals”
- “Get Started with Medical Image Labeler” (Medical Imaging Toolbox)

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 **Signal Labeler** app.

See Also

Signal 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

Signal 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

Measure and Manage Impulse Responses


In this tutorial, explore key functionality of the **Impulse Response Mesurer**. The **Impulse Response Mesurer** app enables you to

- Configure your audio I/O system.
- Acquire impulse response (IR) measurements using either the exponential swept sine (ESS) or maximum length sequences (MLS) methods.
- View and manage captured IR data.
- Export the data to a file, workspace, or other app for further study.
- Generate a script that performs IR measurements according to the current settings.

To begin, open the **Impulse Response Mesurer** app by selecting the  icon from the app gallery.

Configure Audio I/O System

The **Impulse Response Mesurer** app enables you to specify an audio device, sample rate, samples per frame, 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:

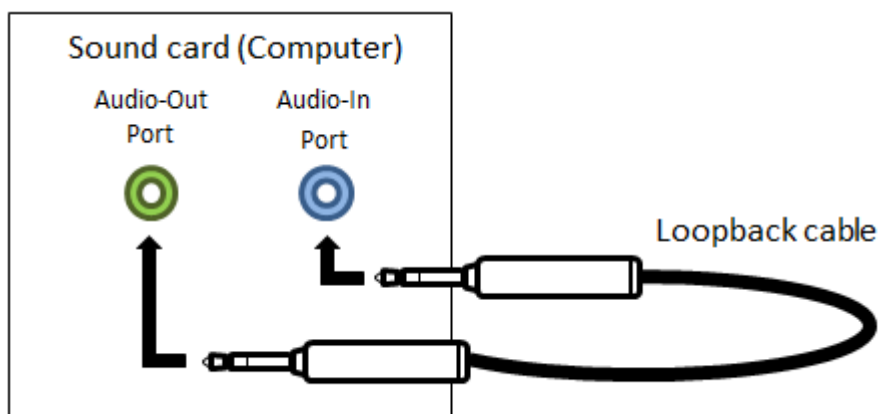
- Windows® -- ASIO™: Click the  button to open the settings panel for the ASIO driver.
- Mac -- CoreAudio
- Linux® -- ALSA

Valid values for sample rate and number of samples per frame depend on your specified audio device.

You can use the level monitor to verify the configuration of your audio I/O system.

Loopback Cable for Latency Measurement

To measure the audio device latency and remove it from captured measurements, you must use a loopback cable to connect one of the device player channels directly to one of the recorder channels.



To enable latency measurement and removal, click the **Latency Compensation** drop-down list, select **Loopback Measurement**, and then set the **Loopback Channels** to the player and recorder channels that are connected by the loopback cable.

This example sets **Latency Compensation** to **None**, so it does not measure the audio device latency.

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 (**Swept Sine**)

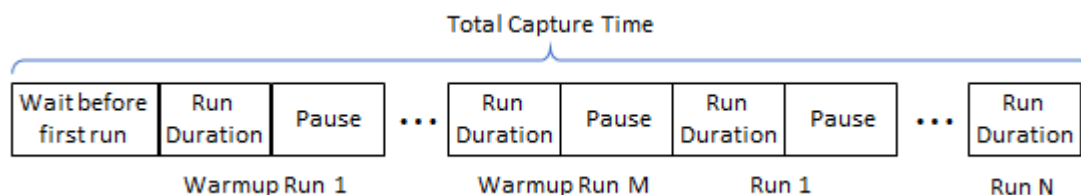
Both methods for IR acquisition have the same basic settings, including:

- **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 the pause between runs. During a pause, the excitation signal is not sent, and audio is not recorded. When using the **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.

The total capture time is a sum of run durations, pauses, and the initial wait.



The **Swept Sine** method has additional **Advanced Settings** to control the excitation signal, including:

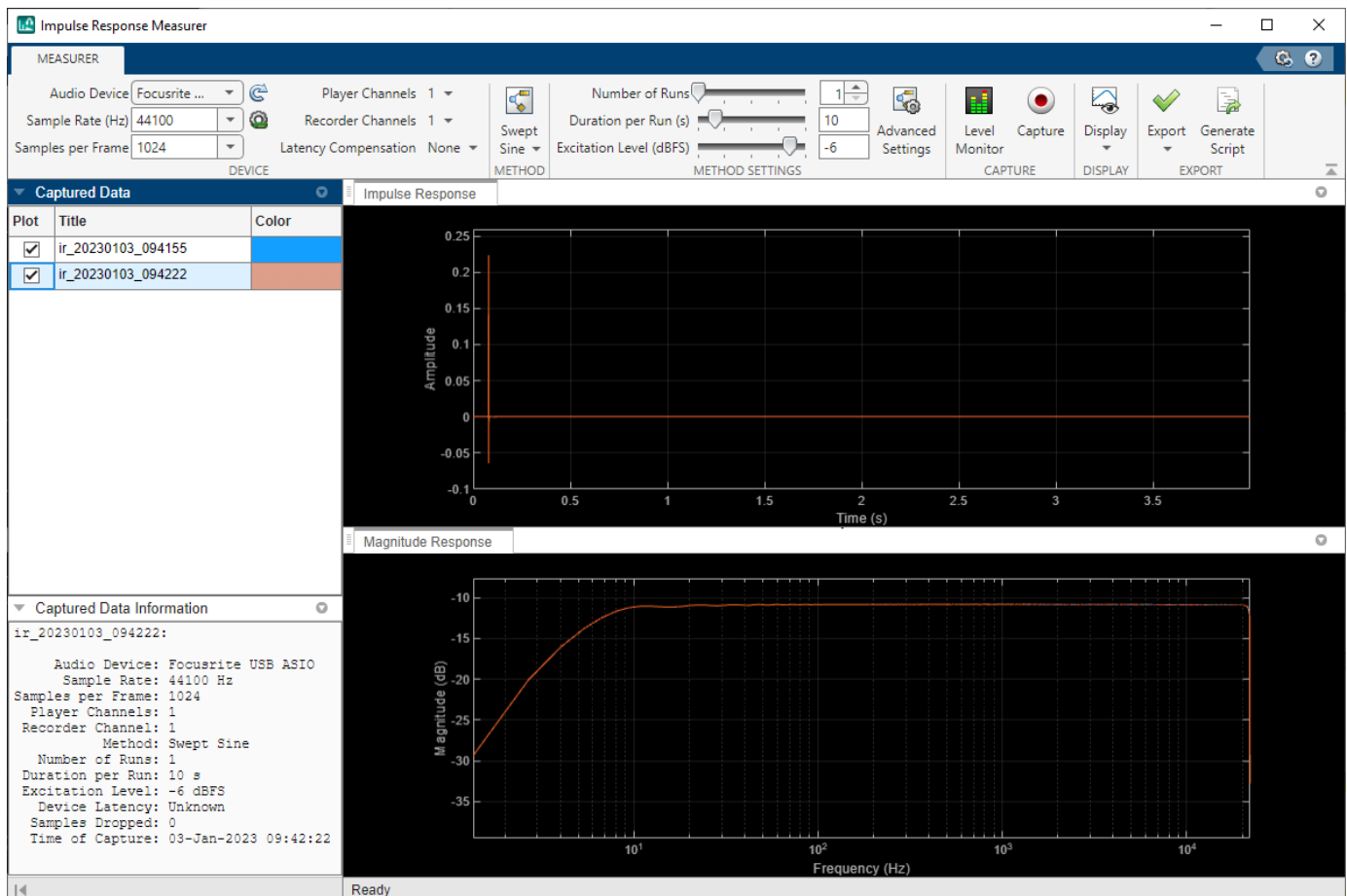
- **Sweep start frequency**
- **Sweep stop frequency**
- **Sweep duration**
- **End silence duration**

When using the **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.

Starting in R2022a, you can automatically save device, method, and advanced settings and use them in future measurement sessions.

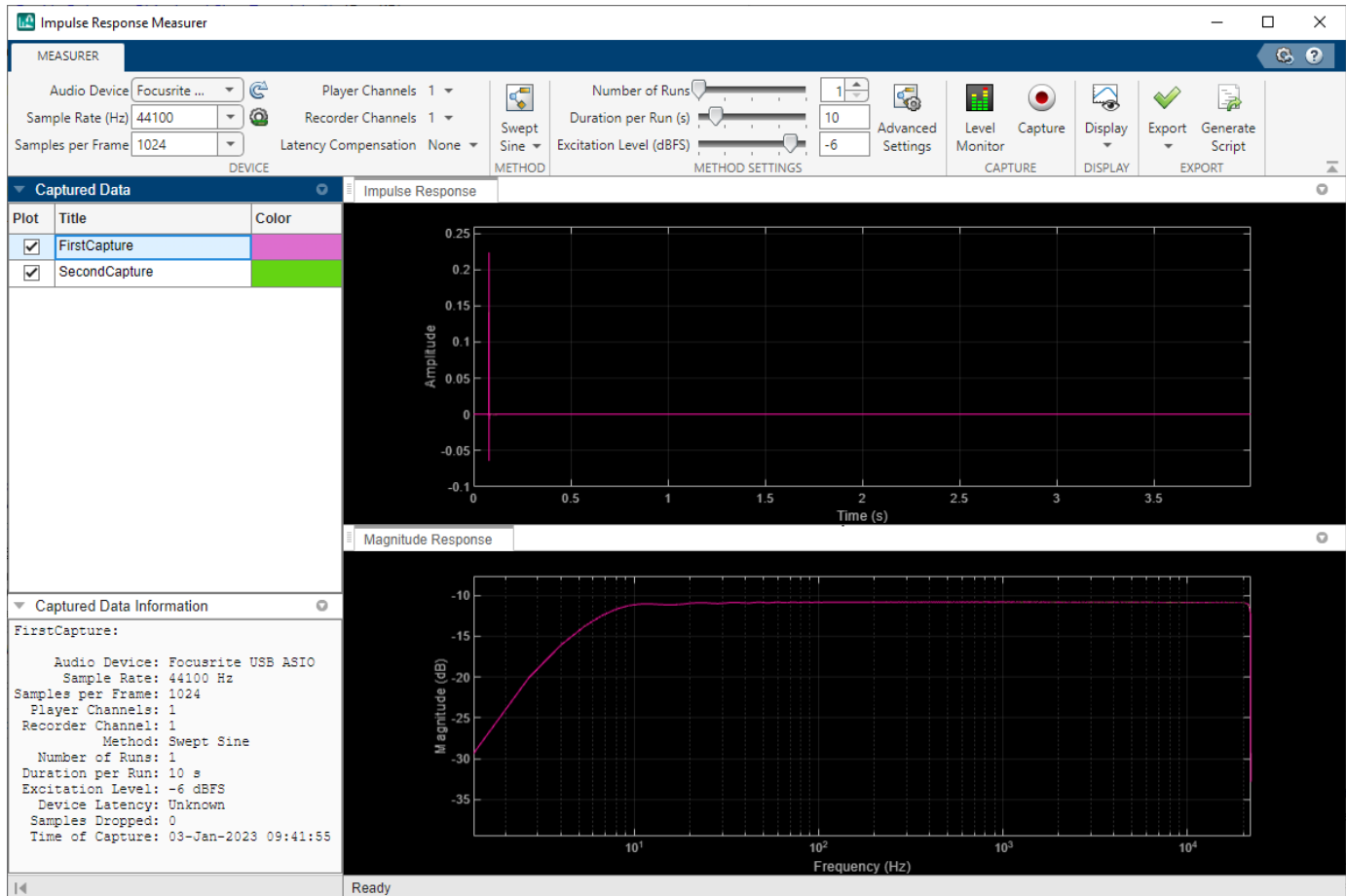
Acquire IR Measurements

For this example, use the **Swept Sine** 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. Capture IR measurements twice.



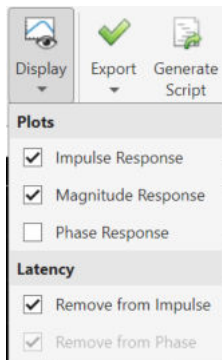
Analyze and Manage IR Measurements

After the capture, the **Impulse Response Measurer** app stores the captured data locally. The **Captured Data** panel 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 the color 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 under **Captured Data**. Select the capture you relabeled **FirstCapture**.

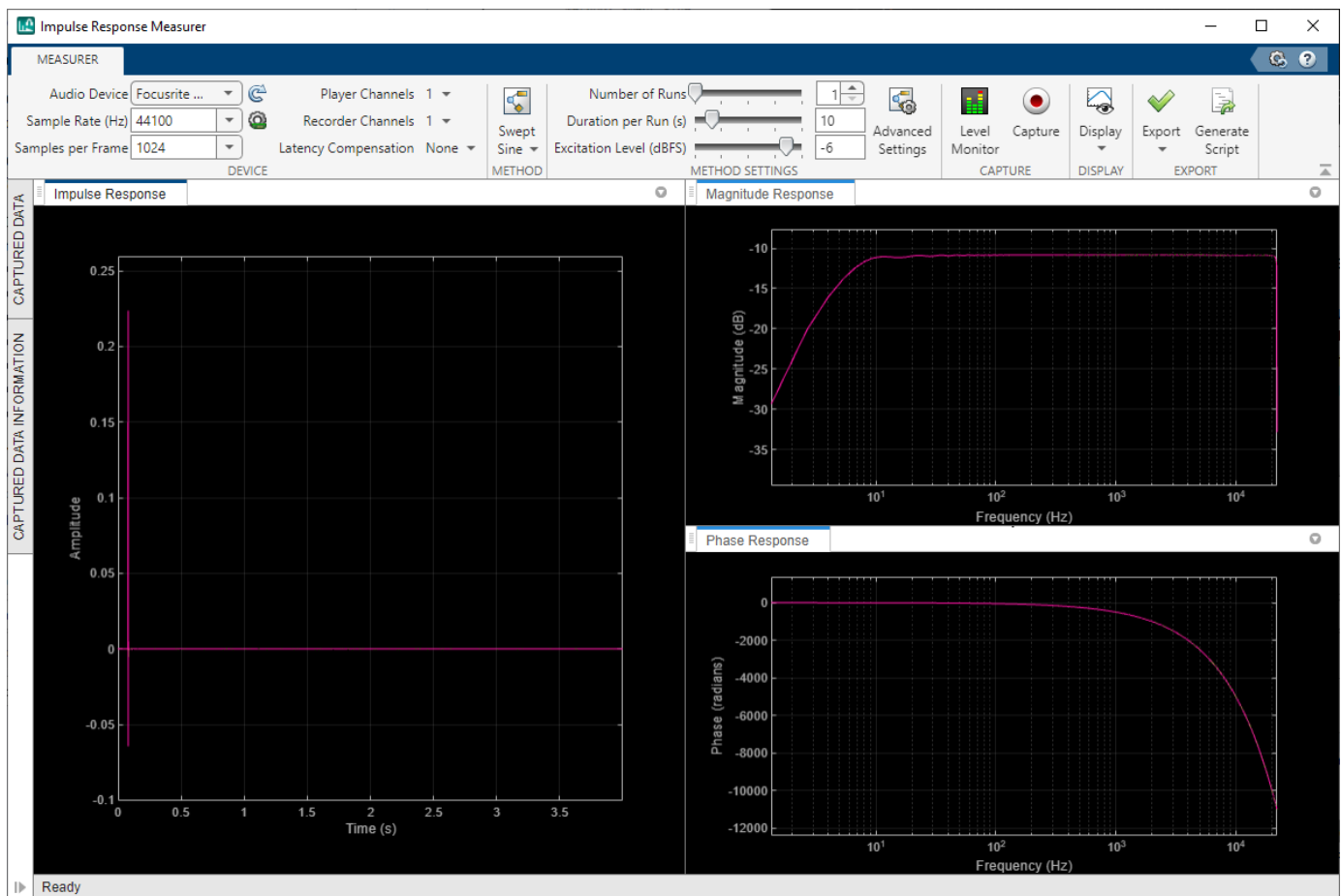


By default, the impulse response and magnitude response are plotted. You can view any combination of the impulse, magnitude, and phase response using the **Display** button. Here you can also remove the measured audio device latency from the plotted impulse response and phase response.

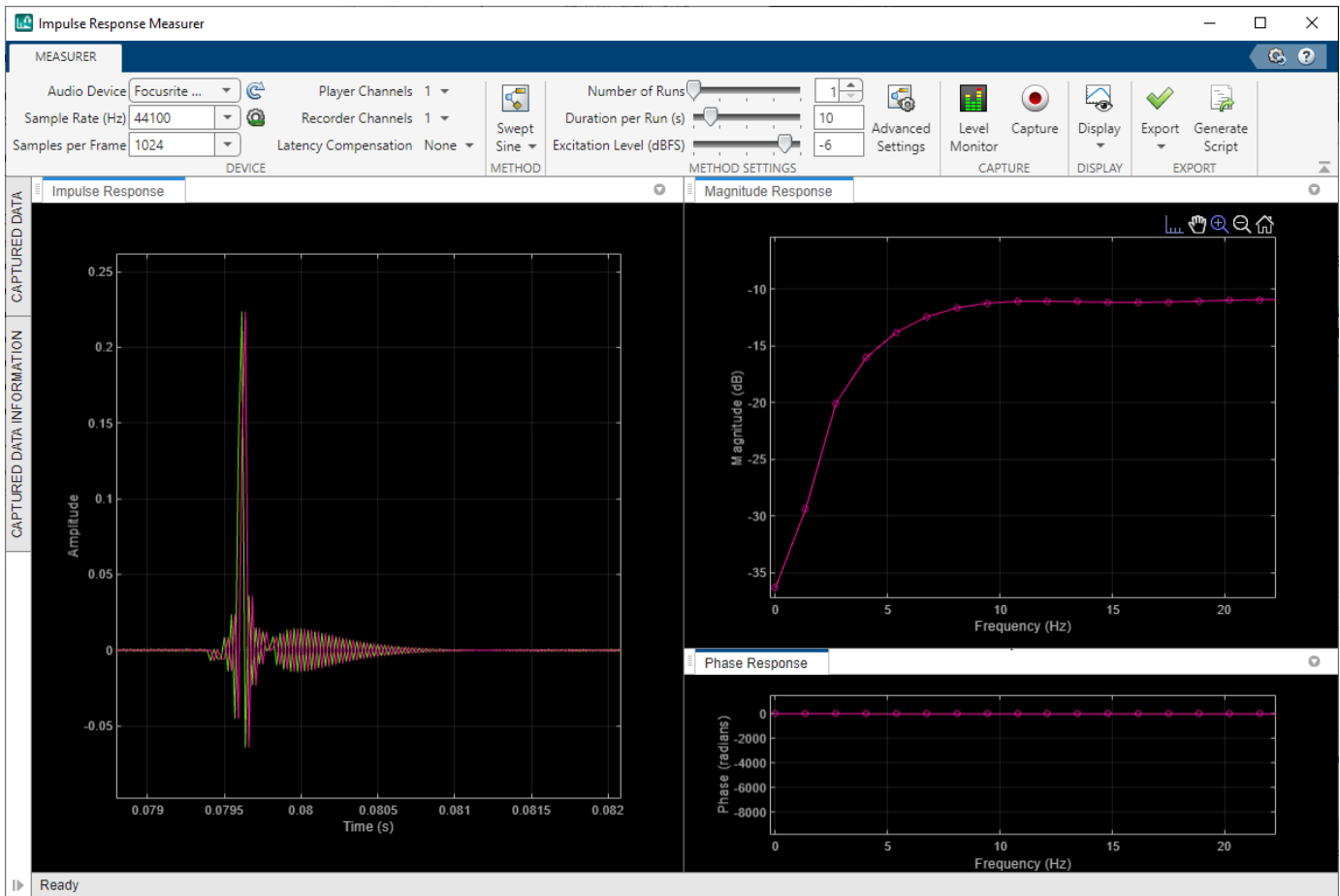
5 Measure Impulse Response of an Audio System



Minimize **Captured Data** and **Captured Data Information**, then select the **Phase Response**.

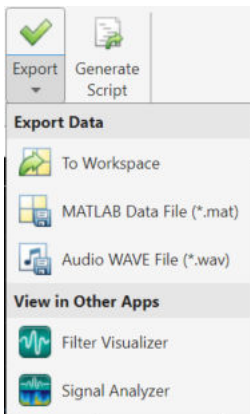


You can toggle the relative size of the plot by moving the dividers. You can zoom in and out or toggle between linear and logarithmic frequency axes by selecting the icons that appear when your pointer is over the plot. Updating either the magnitude response or the phase response updates the other. Zoom in on the impulse response plot and in the range 0–20 Hz of your frequency response plots. Zooming in, you can see the small delay between **FirstCapture** and **SecondCapture**. When the zoom level is high enough, line markers automatically appear.



Export IR Measurements

To view export options for further analysis or use, click **Export**.



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_160957`

```
irdata_160957 =
```

```
2×17 table
```

	TimeOfCapture	ImpulseResponse	Device
FirstCapture	30-Jun-2022 15:59:42 UTC-04:00	1×1 struct	"Focusrite USB ASIO"
SecondCapture	30-Jun-2022 15:59:54 UTC-04:00	1×1 struct	"Focusrite USB ASIO"

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 is 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 **FVTool** or **Signal Analyzer**.

Generate MATLAB Code

You can generate MATLAB code that measures impulse responses using the current settings of the app. To open an untitled script in the MATLAB editor containing the code, click **Generate Script** in the **Export** section of the toolstrip.

```
untitled * x +
1 % Generated by Impulse Response Measurer on 28-Dec-2022 14:36:18 UTC-05:00.
2
3 %% Device Settings
4 device = "Focusrite USB ASIO"; % Audio device name
5 fs = 44100; % Sample rate (Hz)
6 L = 1024; % Samples per frame
7 recChMap = 1; % Recorder channel mapping
8 playChMap = 1; % Player channel mapping
9 nbPlayCh = 1; % Total number of playback channels
10 nbRecCh = 1; % Total number of recorder channels
11
12 %% Method Settings (Exponential Swept Sine)
13 durationPerRun = 10; % Duration per Run (s)
14 outputLevel = -6; % Excitation Level (dBFS)
15 % Advanced Run Settings
16 nbWarmUps = 0; % Number of warm-up runs
17 % Advanced Excitation Settings
18 sweepRange = [10 22000]; % Sweep start/stop frequency (Hz)
19 sweepDur = 6; % Sweep Duration (s)
20 % IR measurement duration corresponds to the silent time after the sweep
21 irDur = durationPerRun - sweepDur;
22
23 %% Create the excitation signal
24 % Use swepttone to generate one run of the excitation
25 exc = swepttone( ...
26 sweepDur, irDur, fs, ...
```

Run the script to measure the impulse response and store the captured data in the capture structure.

```
capture
```

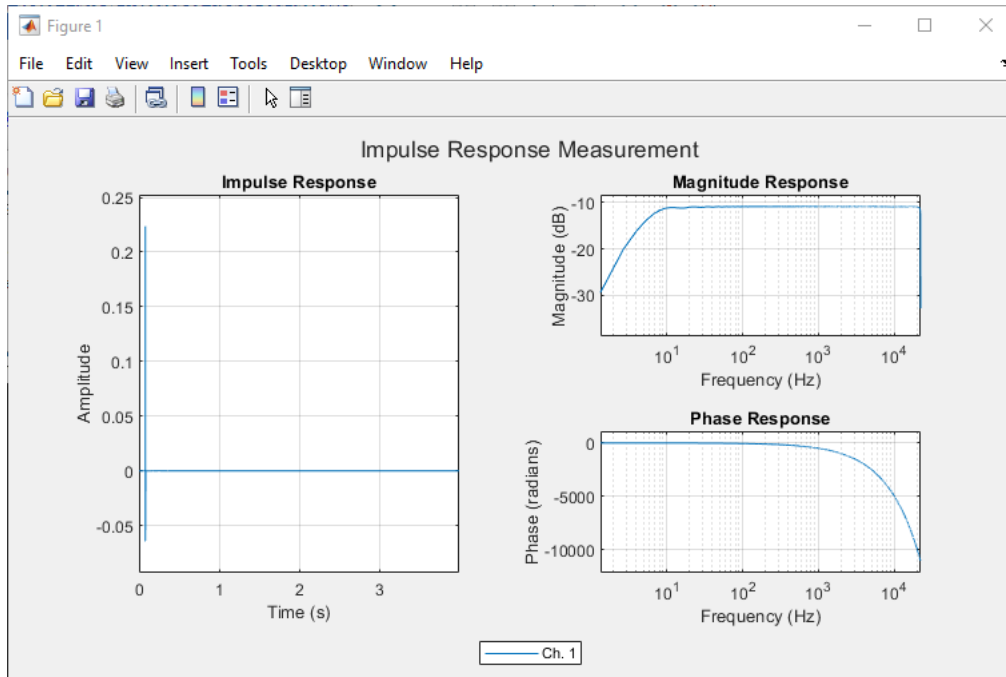
```
capture =
```

```
struct with fields:
```

```
ImpulseResponse: [1×1 struct]
```

```
MagnitudeResponse: [1x1 struct]
PhaseResponse: [1x1 struct]
```

The script optionally plots the impulse, magnitude, and phase response according to the **Display** settings in the app.



You can examine the generated code to understand how the app performs the measurement, and you can edit the script for customization.

See Also

Impulse Response Measurer | [audioPlayerRecorder](#) | [splMeter](#) | [reverberator](#)

Related Examples

- “Measure Impulse Response of an Audio System” on page 1-262
- “Measure Frequency Response of an Audio Device” on page 1-267

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
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`

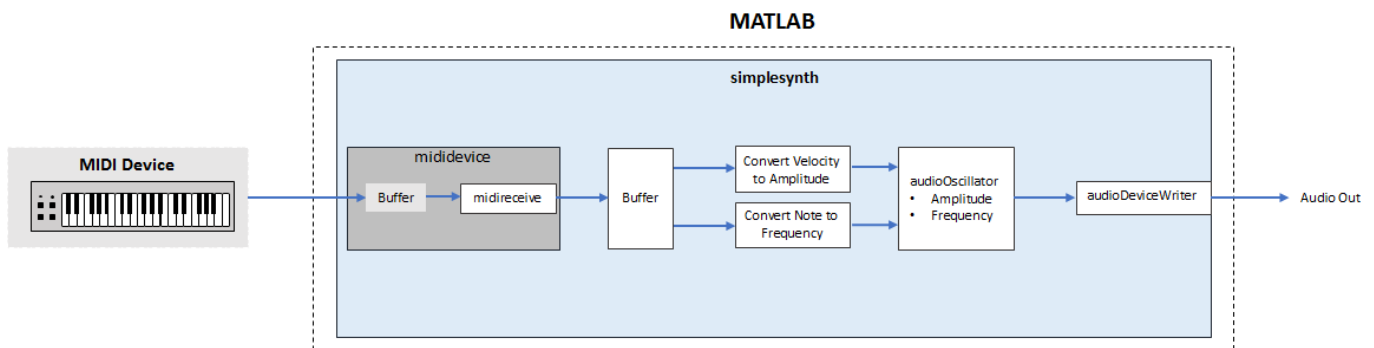
```

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

midimsg | mididevice

Functions

midisend | midireceive | mididevinfo

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:
```

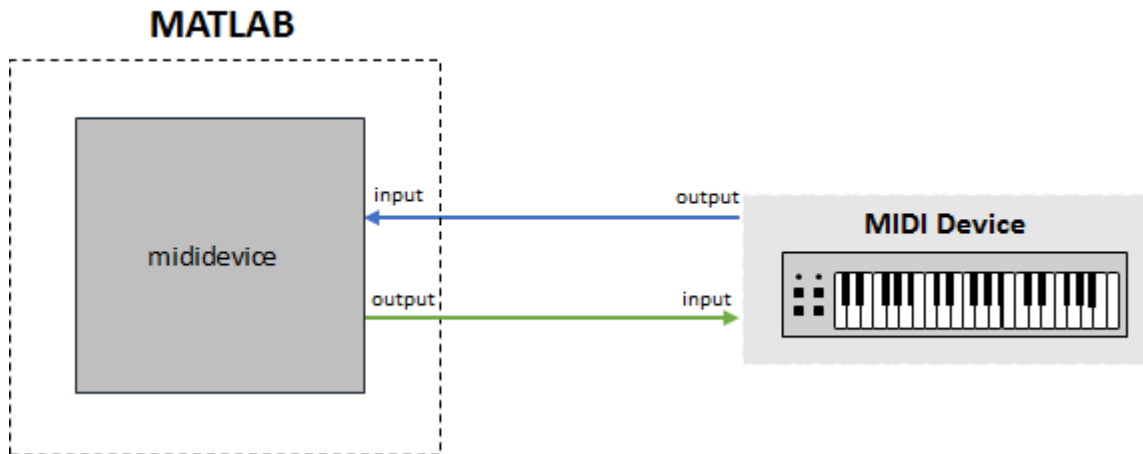
ID	Direction	Interface	Name
0	output	MMSystem	'Microsoft MIDI Mapper'
1	input	MMSystem	'USB MIDI Interface '
2	output	MMSystem	'Microsoft GS Wavetable Synth'
3	output	MMSystem	'USB MIDI Interface '

```
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.

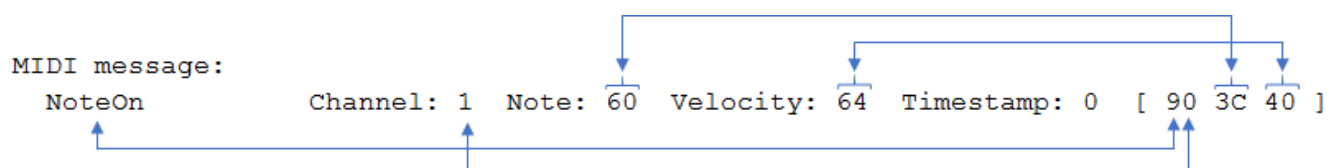
In MATLAB, a MIDI message is packaged as a `midimsg` object and can be manipulated as scalars or arrays. To create a MIDI message, call `midimsg` with a message type and then specify the required parameters for the specific message type. For example, to create a note on message, specify the `midimsg` 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 =
```

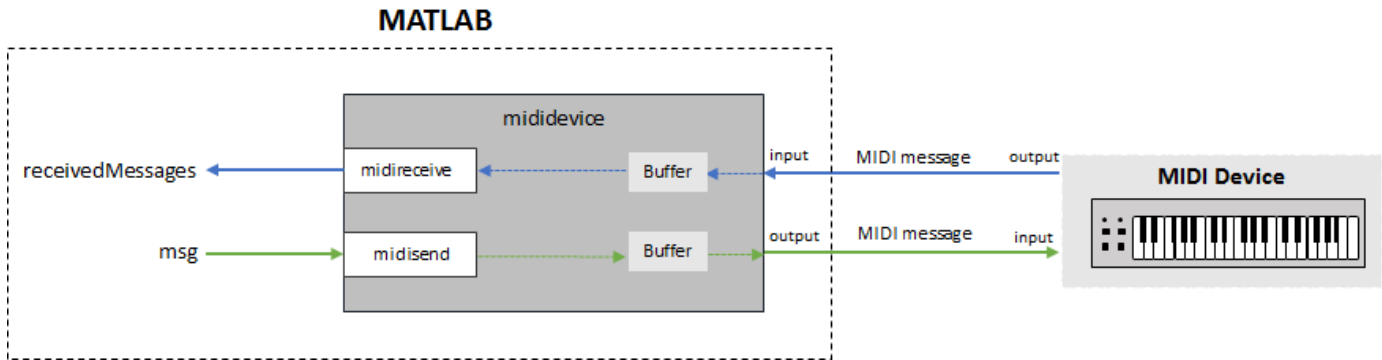
```
MIDI message:
  NoteOn      Channel: 1  Note: 60  Velocity: 64  Timestamp: 0  [ 90 3C 40 ]
```

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

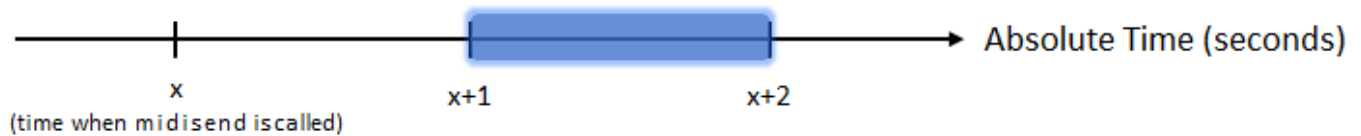
The MIDI protocol does not define message timing and assumes that messages are acted on immediately. Many applications require timing information for queuing and batch processing. For convenience, the Audio Toolbox packages timing information with MIDI messages into a single `midimsg` object. All `midimsg` 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.

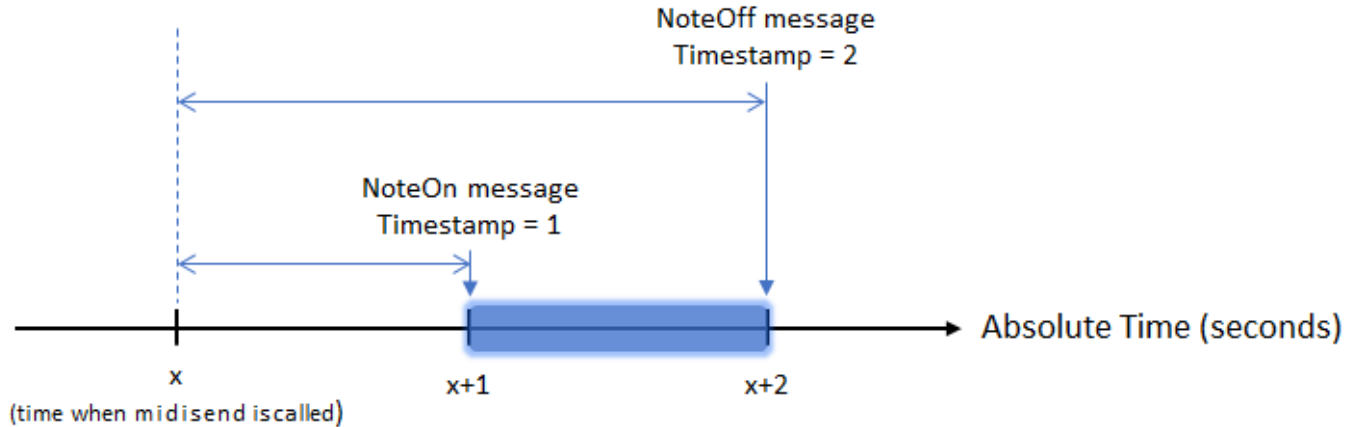


Create Note On and Note Off messages. To create the Note Off message, use the 'NoteOn' MIDI message type and specify zero velocity. (If you want to specify a velocity, use the 'NoteOff' message type.) For more information, see `midimsg`.

```
OnMsg = midimsg('NoteOn',1,59,64);
OffMsg = midimsg('NoteOn',1,59,0);
```

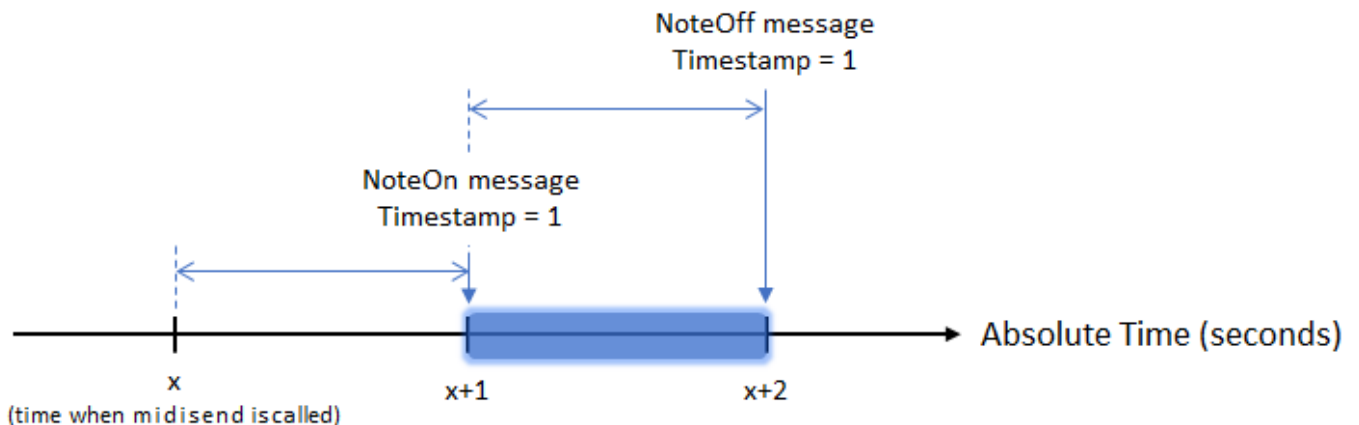
To send on and off messages using a single call to `midisend`, specify the timestamps of the messages relative to the same start time.

```
OnMsg.Timestamp = 1;
OffMsg.Timestamp = 2;
midisend(device,[OnMsg;OffMsg])
```



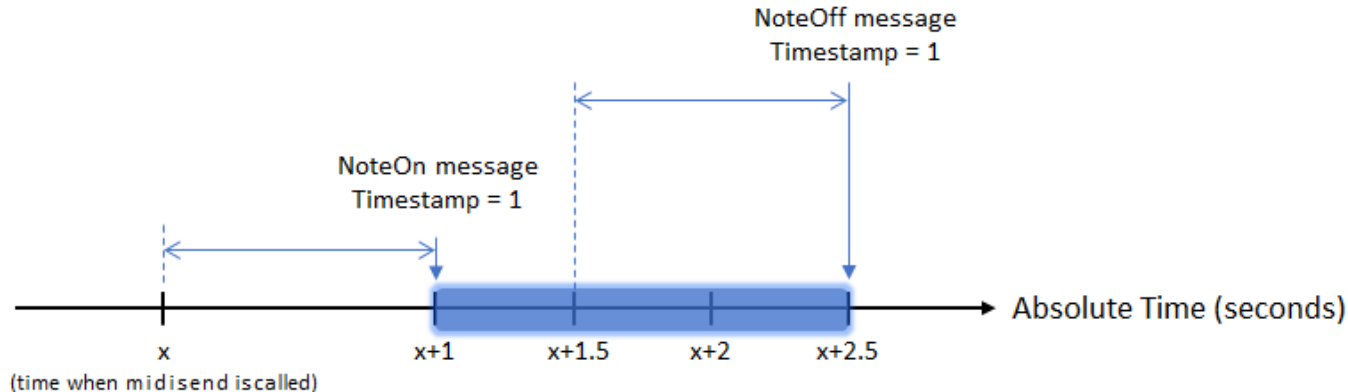
To send the Note Off message separately, specify the timestamp of the Note Off message relative to the largest timestamp of the previous call to midisend.

```
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

`midimsg` | `mididevice`

Functions

`midisend` | `midireceive` | `mididevinfo`

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:

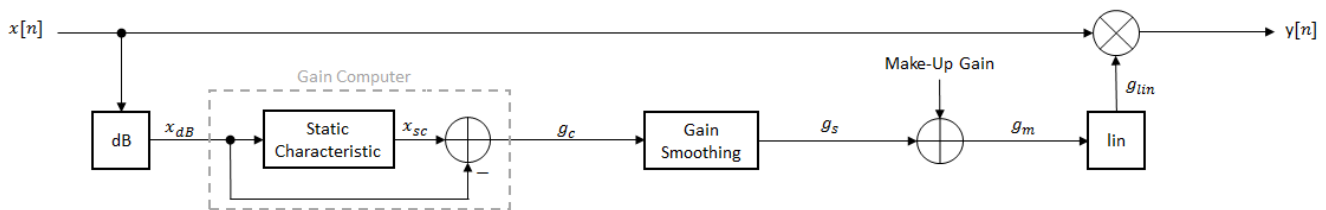
- Match an audio signal level to its environment
- Protect AD converters from overload
- Optimize information
- Suppress low-level noise

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: $y = g_{lin} \times x$

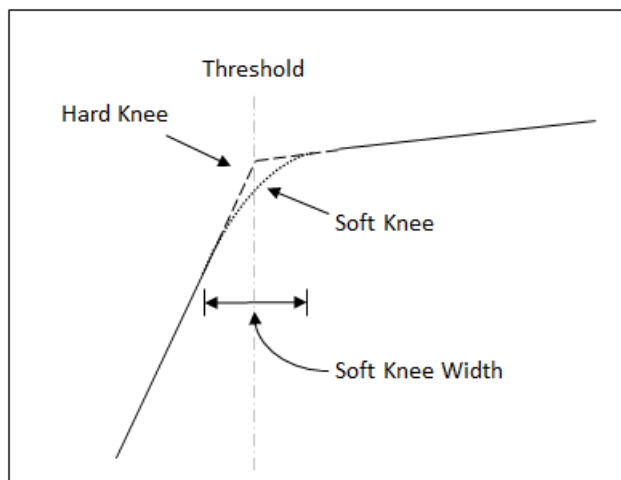
Linear to dB Conversion

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.

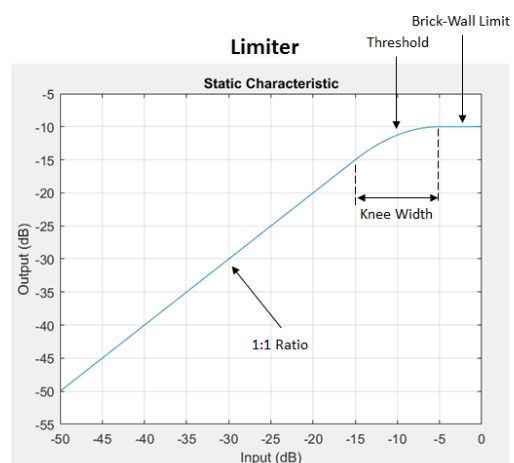
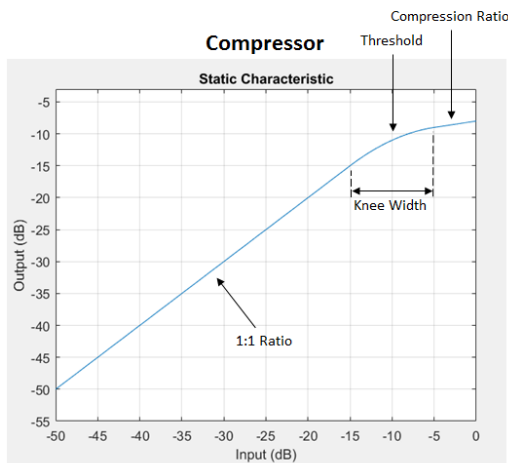
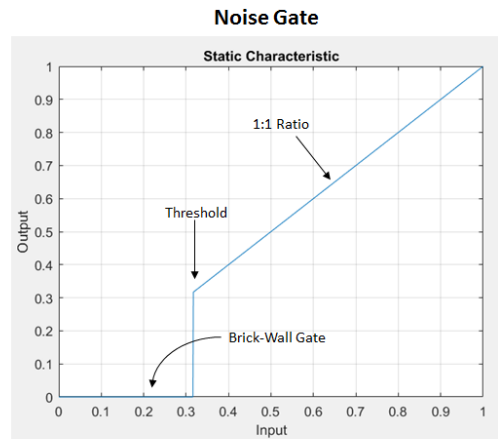
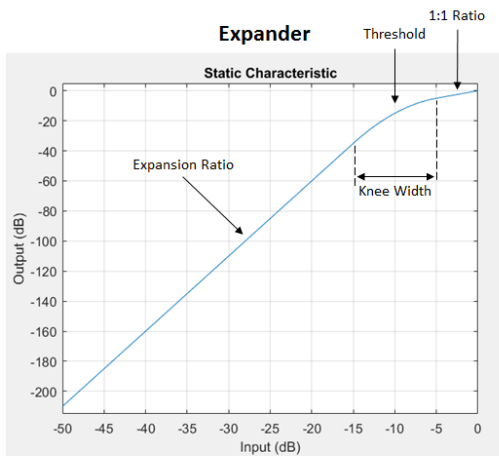
Gain Computer

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

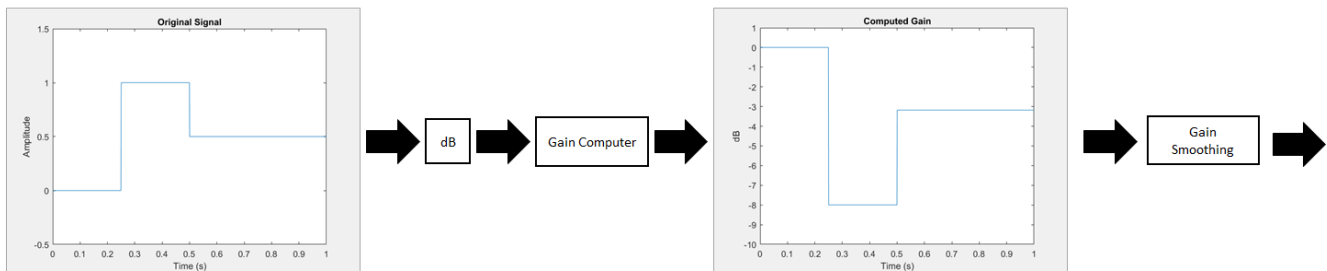
$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) g_c[n] & \text{if } g_c[n] \leq g_s[n-1] \\ \alpha_R g_s[n-1] + (1 - \alpha_R) g_c[n] & \text{if } g_c[n] > g_s[n-1] \end{cases}$$

- α_A and α_R are determined by the sample rate and specified attack and release time:

$$\alpha_A = \exp\left(\frac{-\log(9)}{F_S \times T_A}\right), \quad \alpha_R = \exp\left(\frac{-\log(9)}{F_S \times T_R}\right)$$

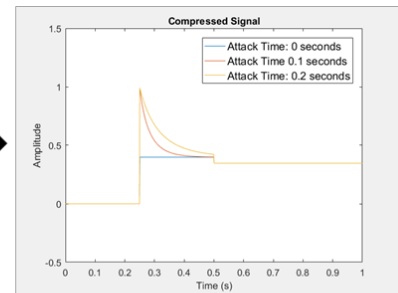
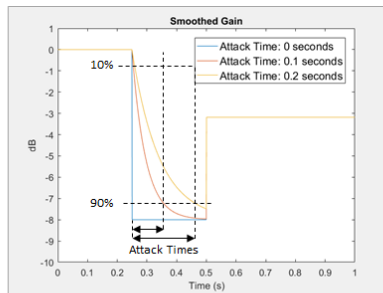
Gain Smoothing Example

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.

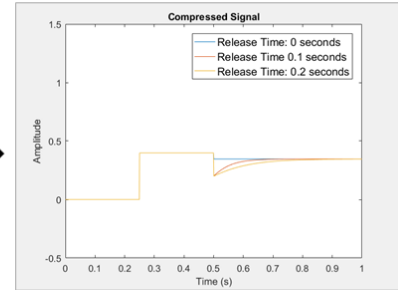
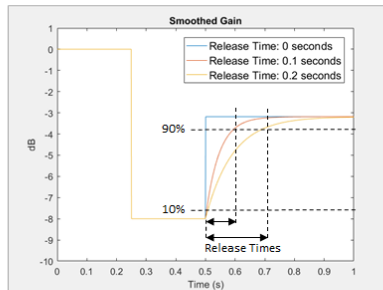


Several variations of gain smoothing are shown. On the top, a smoothed gain curve is shown for different attack time values, with release time set to zero seconds. In the middle, release time is varied and attack time is held constant at zero seconds. On the bottom, both attack and release time are specified by nonzero values.

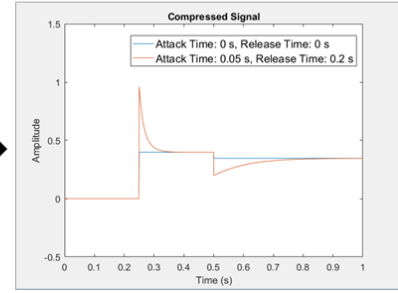
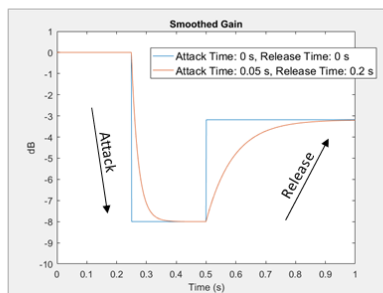
Vary Attack Time
(Release Time = 0 s)



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:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{\left(\frac{1}{R} - 1\right)\left(x_{dB} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \leq x_{dB} \leq \left(T + \frac{W}{2}\right) \\ T + \frac{\left(x_{dB} - T\right)}{R} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases}$$

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:

$$\text{MAKE-UP GAIN} = -x_{sc}(0) = \begin{cases} 0 & \frac{W}{2} < T \\ -\frac{(\frac{1}{R} - 1)(T - \frac{W}{2})^2}{2W} & -\frac{W}{2} \leq T \leq \frac{W}{2} \\ -T + \frac{T}{R} & -\frac{W}{2} > T \end{cases}$$

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

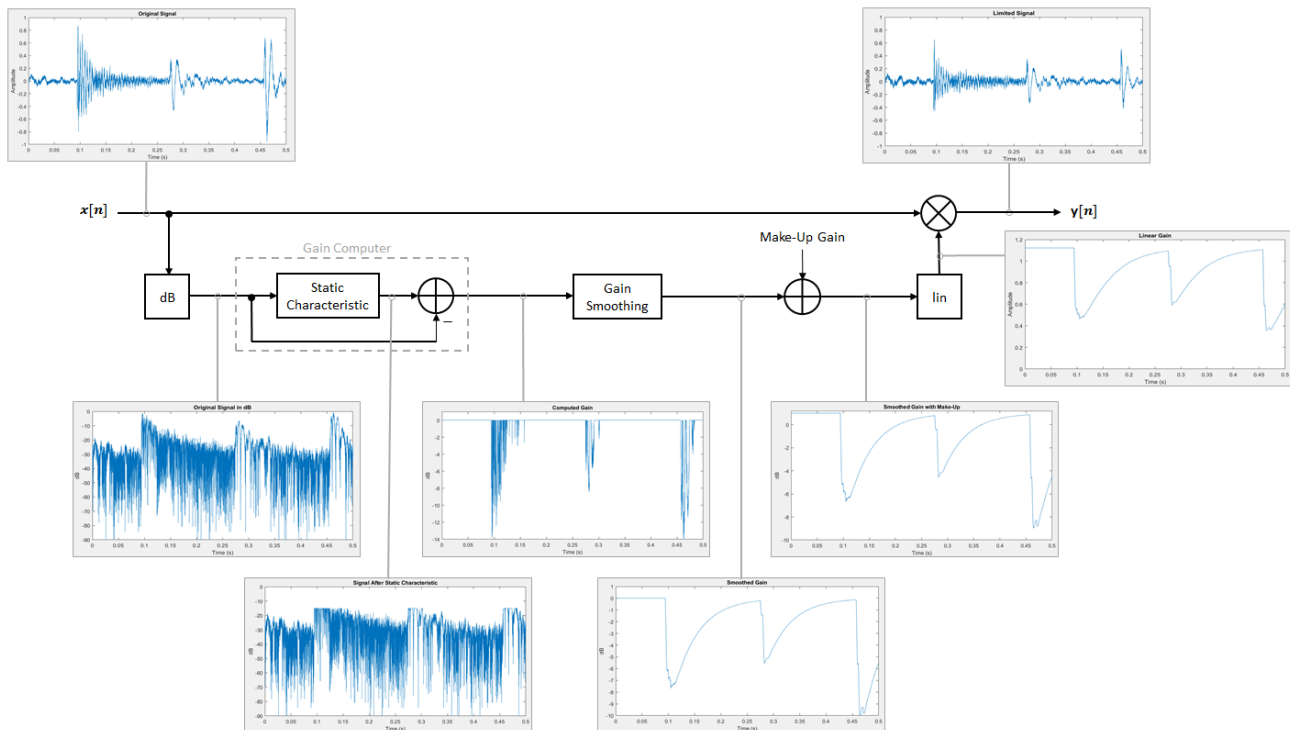
The audio signal described in this example is a 0.5 second interval of a drum track. The limiter properties are:

- 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:

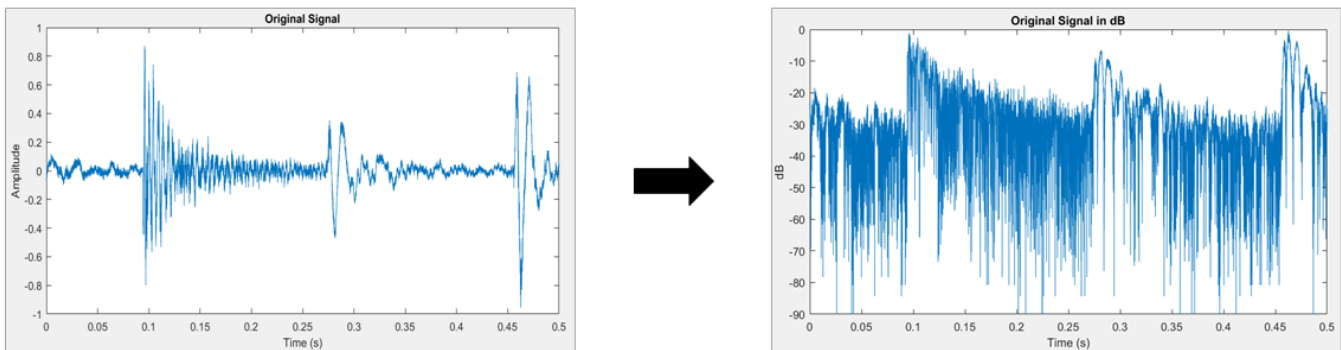
```
dRL = limiter('Threshold',-15,...
             'KneeWidth',0,...
             'AttackTime',0.004,...
             'ReleaseTime',0.1,...
             'MakeUpGainMode','property',...
             'MakeUpGain',1);
```

This example provides a visual walkthrough of the various stages of the dynamic range limiter system.



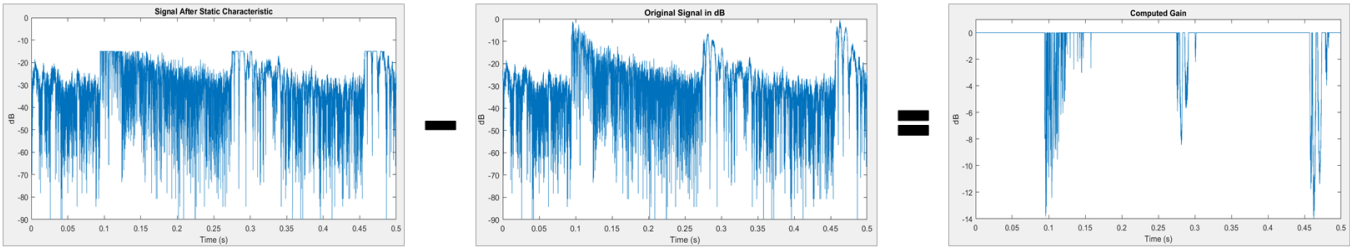
Linear to dB Conversion

The input signal is converted to a dB scale element by element.



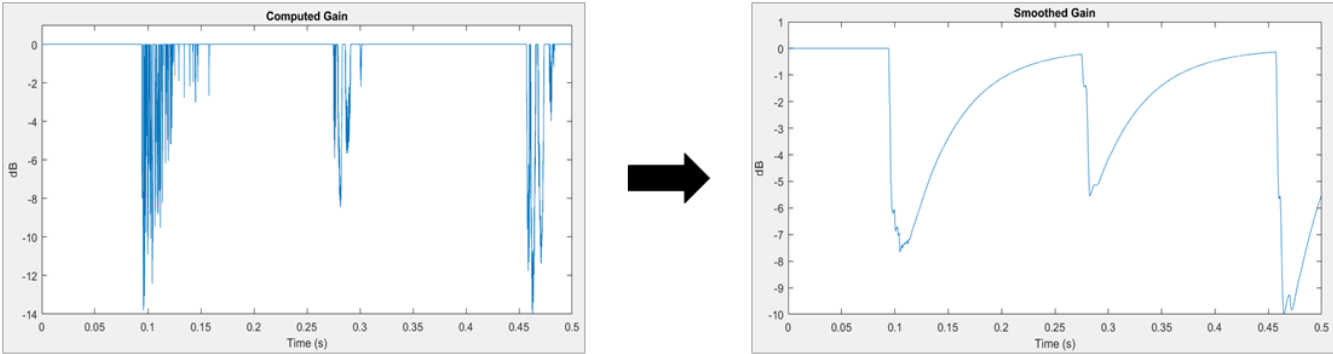
Gain Computer

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.



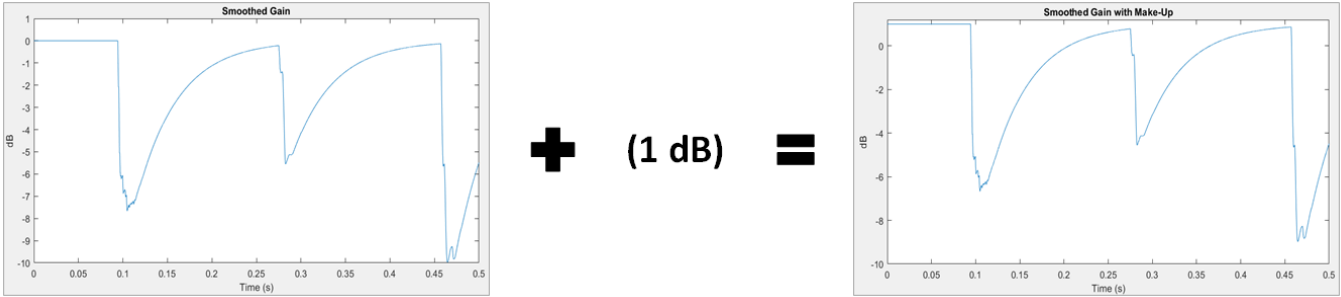
Gain Smoothing

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.



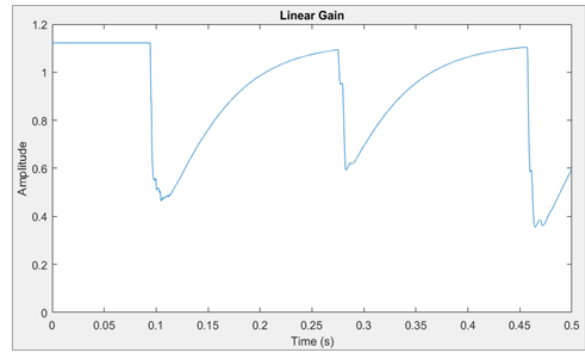
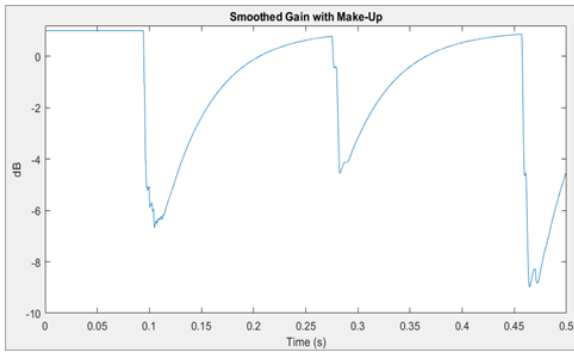
Make-Up Gain

Assume a limiter with a 1 dB make-up gain value. The make-up gain is added to the smoothed gain signal.



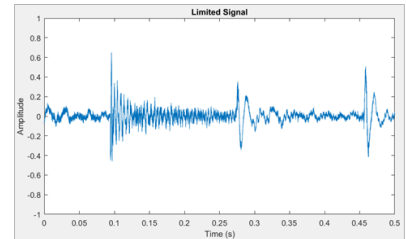
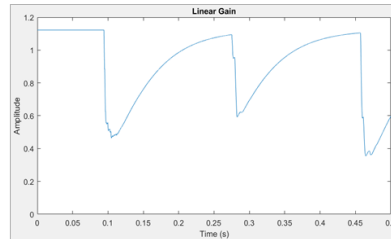
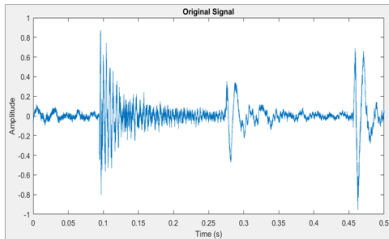
dB to Linear Conversion

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 1-169

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”.

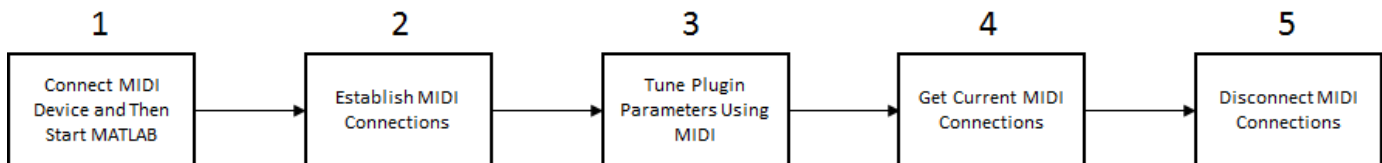
Use MIDI with MATLAB Plugins

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

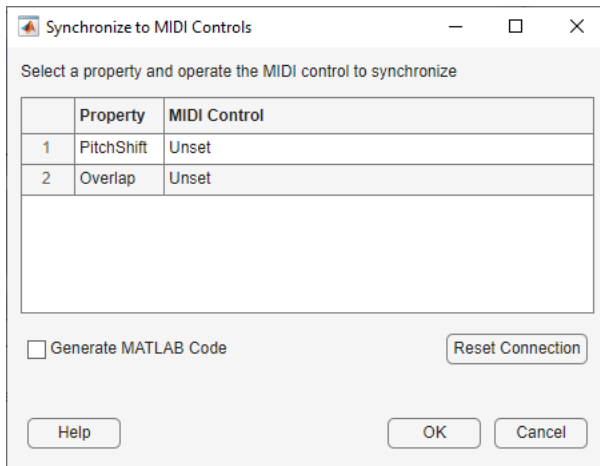
Before starting MATLAB, connect your MIDI control surface to your computer and turn it on. For connection instructions, see the instructions for your MIDI device. If you start MATLAB before connecting your device, MATLAB might not recognize your device when you connect it. To correct the problem, restart 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 select a property and operate a MIDI control, its identification is entered into the **MIDI control** column. 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.

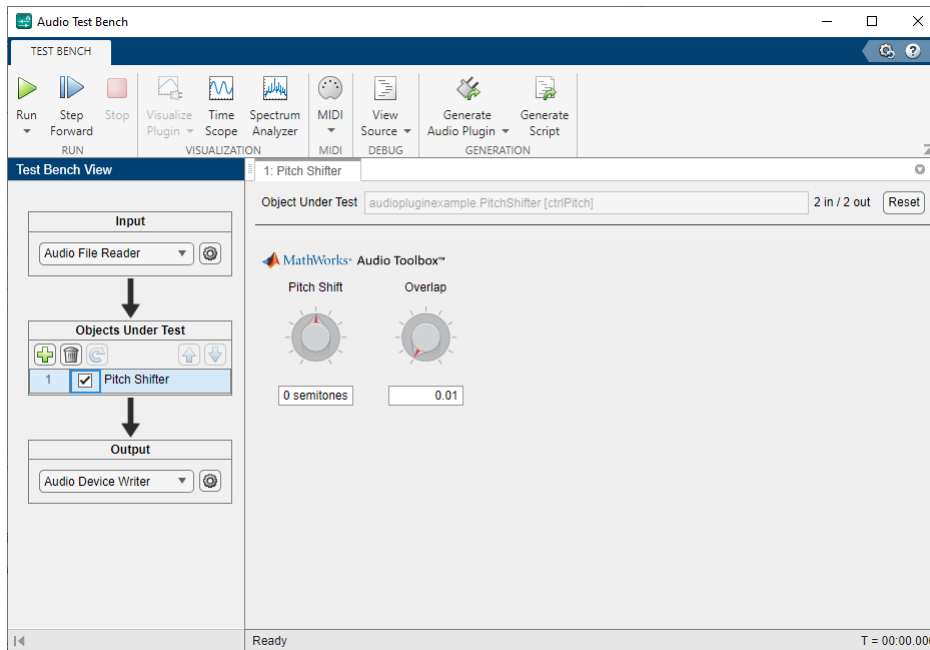
3. Tune Plugin Parameters Using MIDI

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 dials 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: [1x1 struct]
        Overlap: [1x1 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

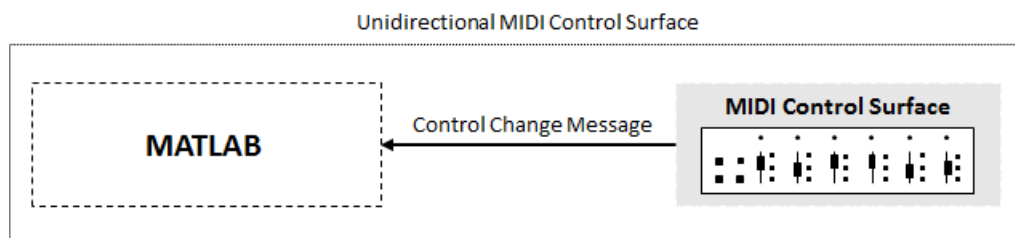
About MIDI

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.

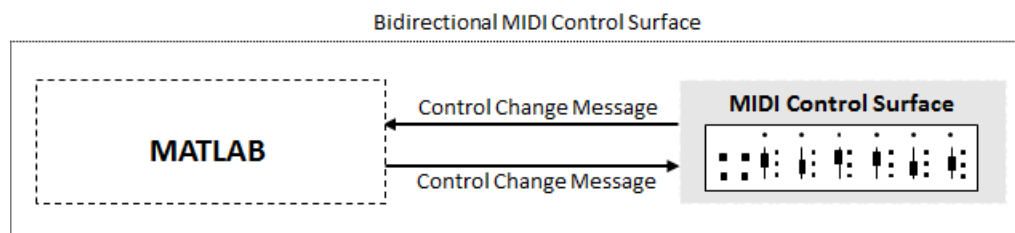
MIDI Control Surfaces

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.

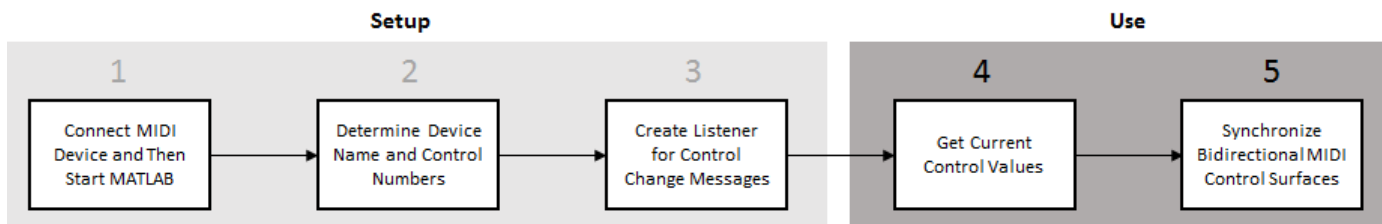


Use MIDI Control Surfaces with MATLAB and Simulink

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.



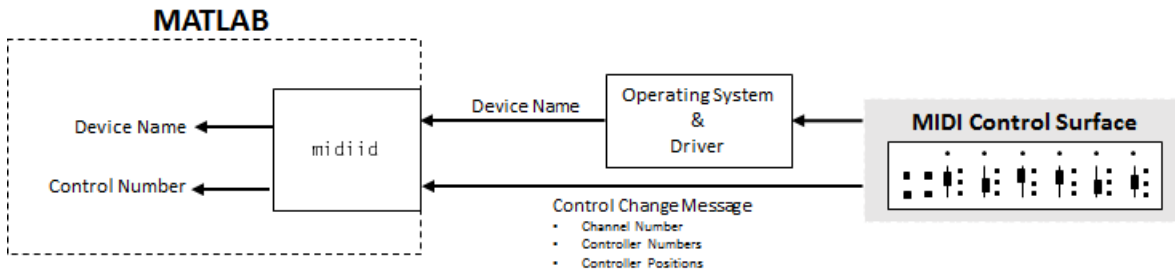
1. Connect MIDI Device and Then Start MATLAB

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.



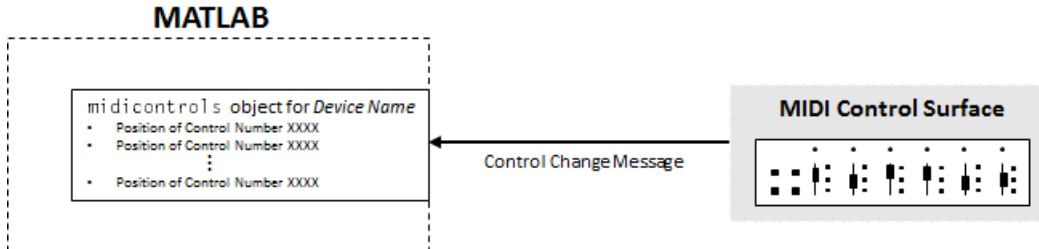
View Example

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

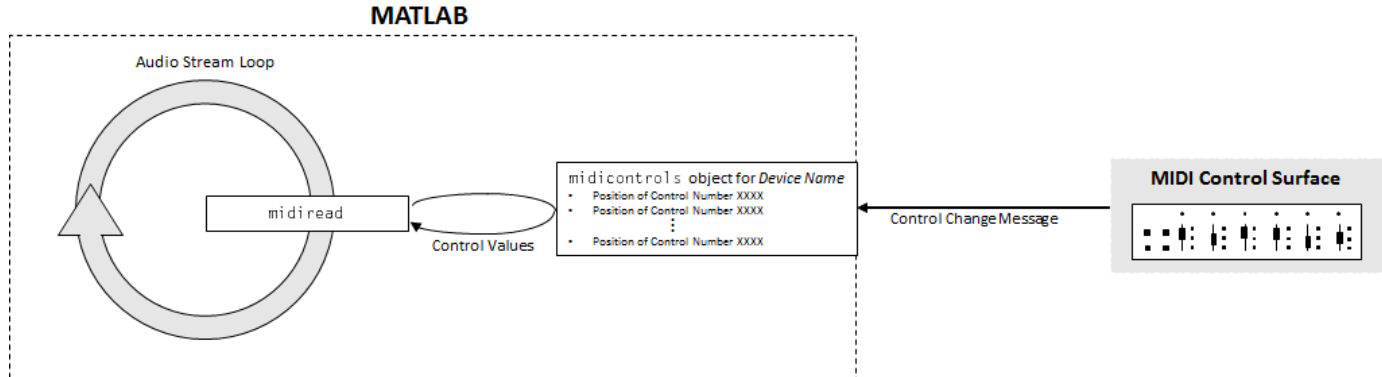
Identify two control numbers on your MIDI control surface. Choose initial control values for the controls you identified. Create a `midicontrols` object that listens to Control Change messages that arrive from the controllers you identified on the device you identified. When you create your `midicontrols` 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, ...
    'MIDIDevice',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,'MIDIDevice',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.

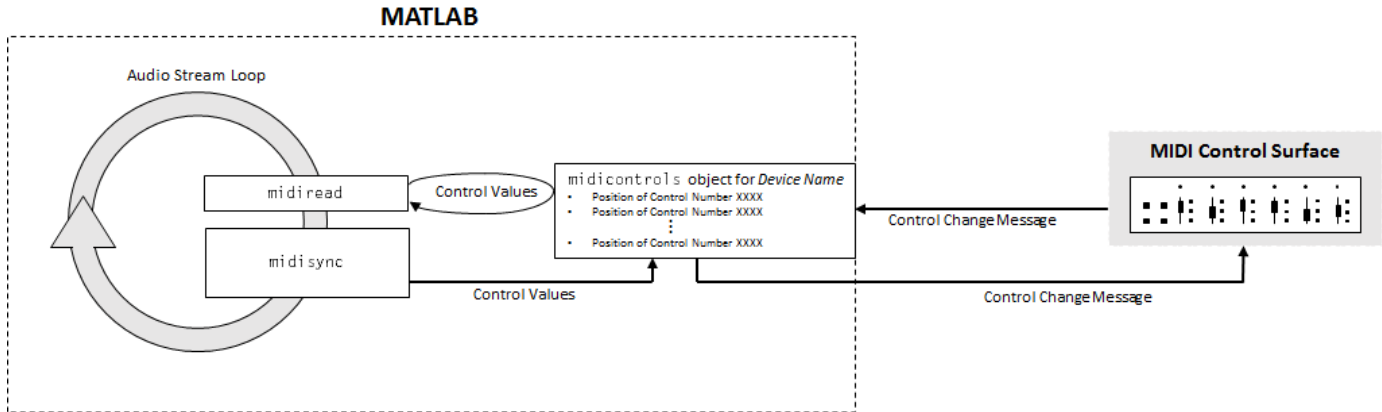
View Example

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,'MIDIDevice',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

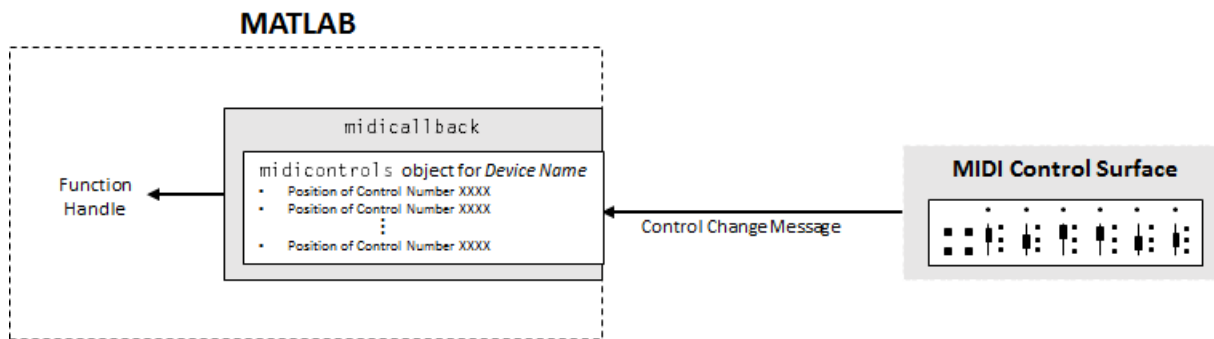
    play(deviceWriter,audioDataWithGain);
end

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` | `midisync` | `midiread` | `midicontrols` | `midiid`

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

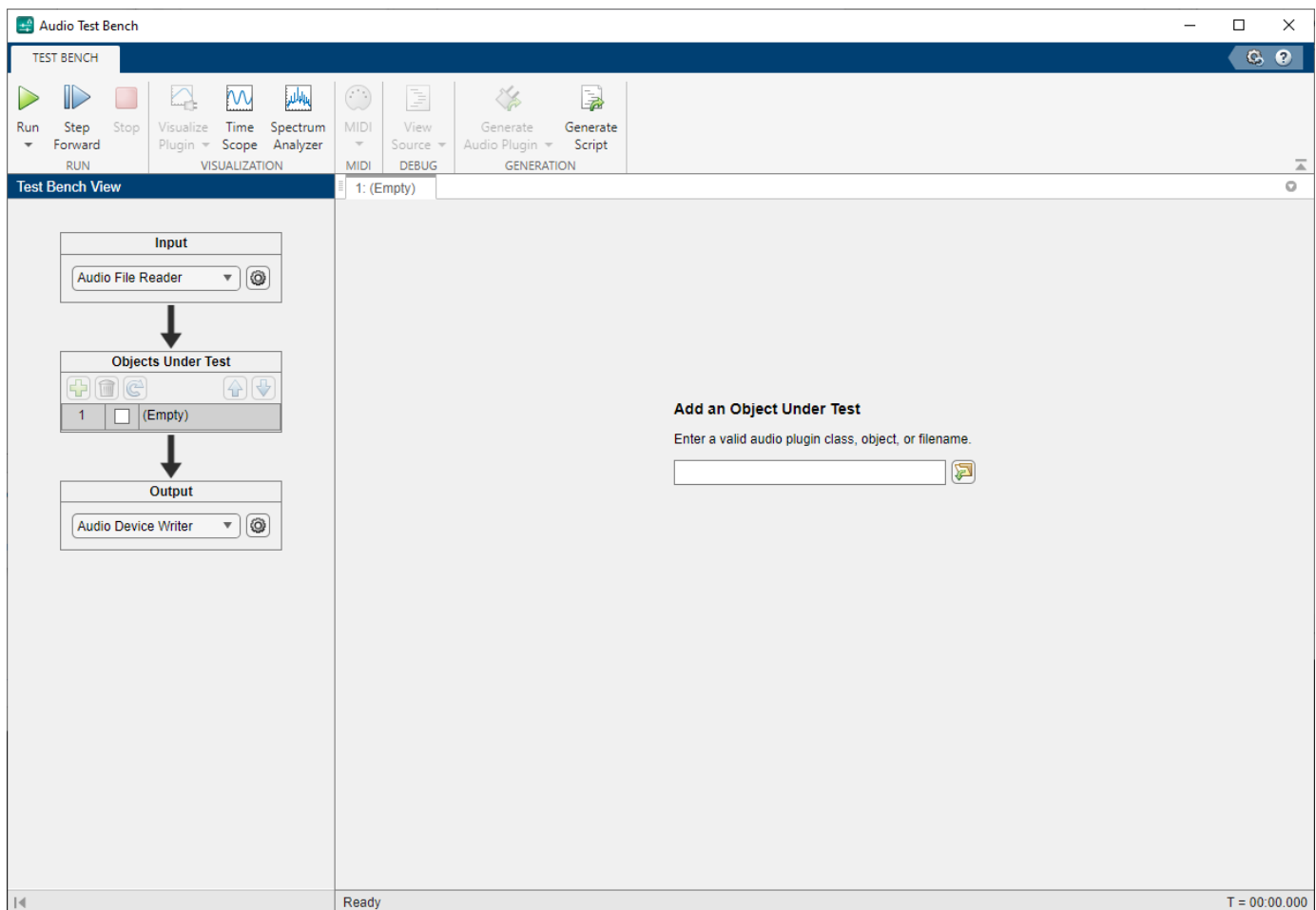
Develop, Analyze, and Debug Plugins In Audio Test Bench

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 Objects Under Test

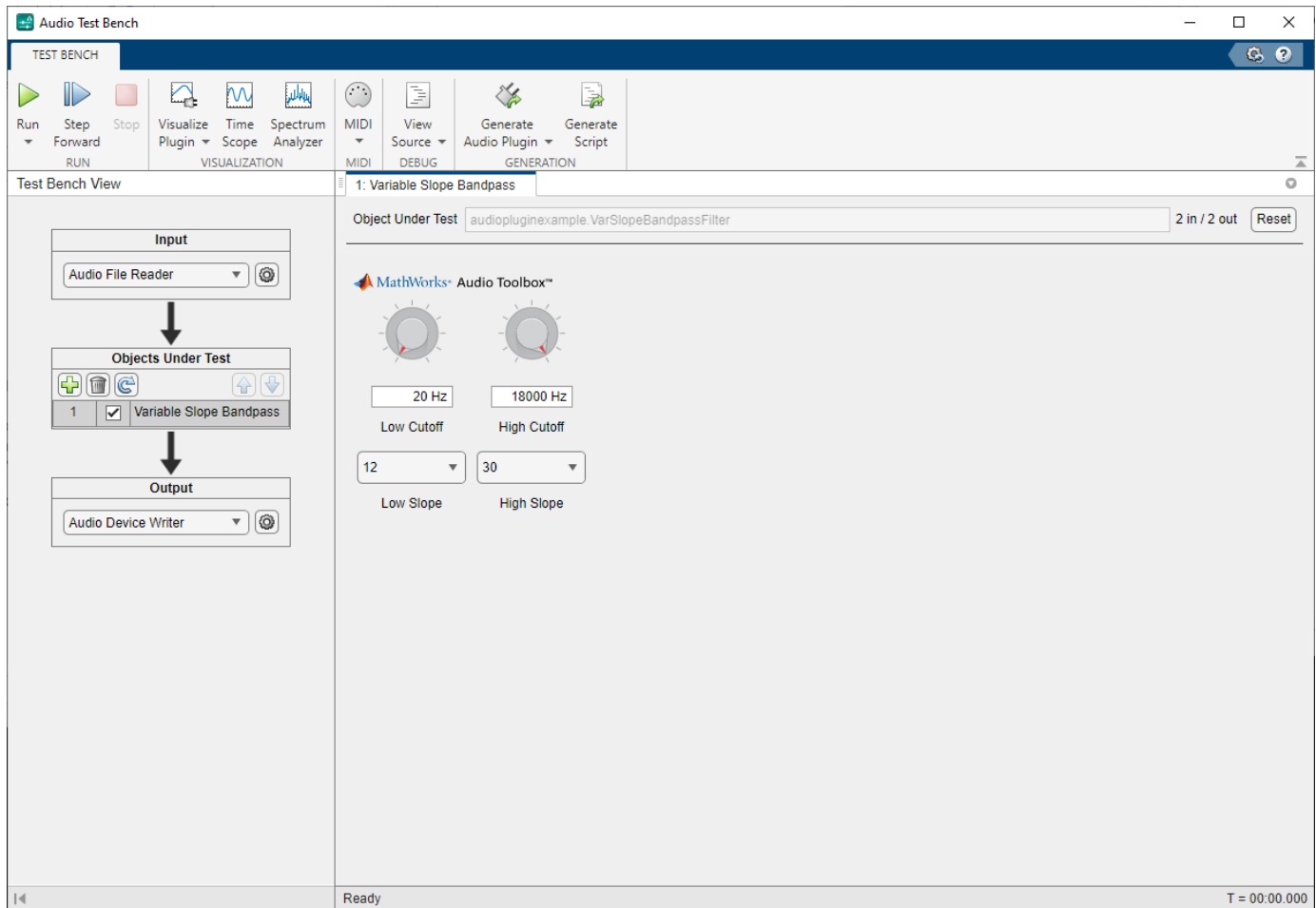
To open the **Audio Test Bench**, at the MATLAB command prompt, enter this command.

```
audioTestBench
```




In the **Add an Object Under Test** field, enter `audiopluginexample.VarSlopeBandpassFilter`

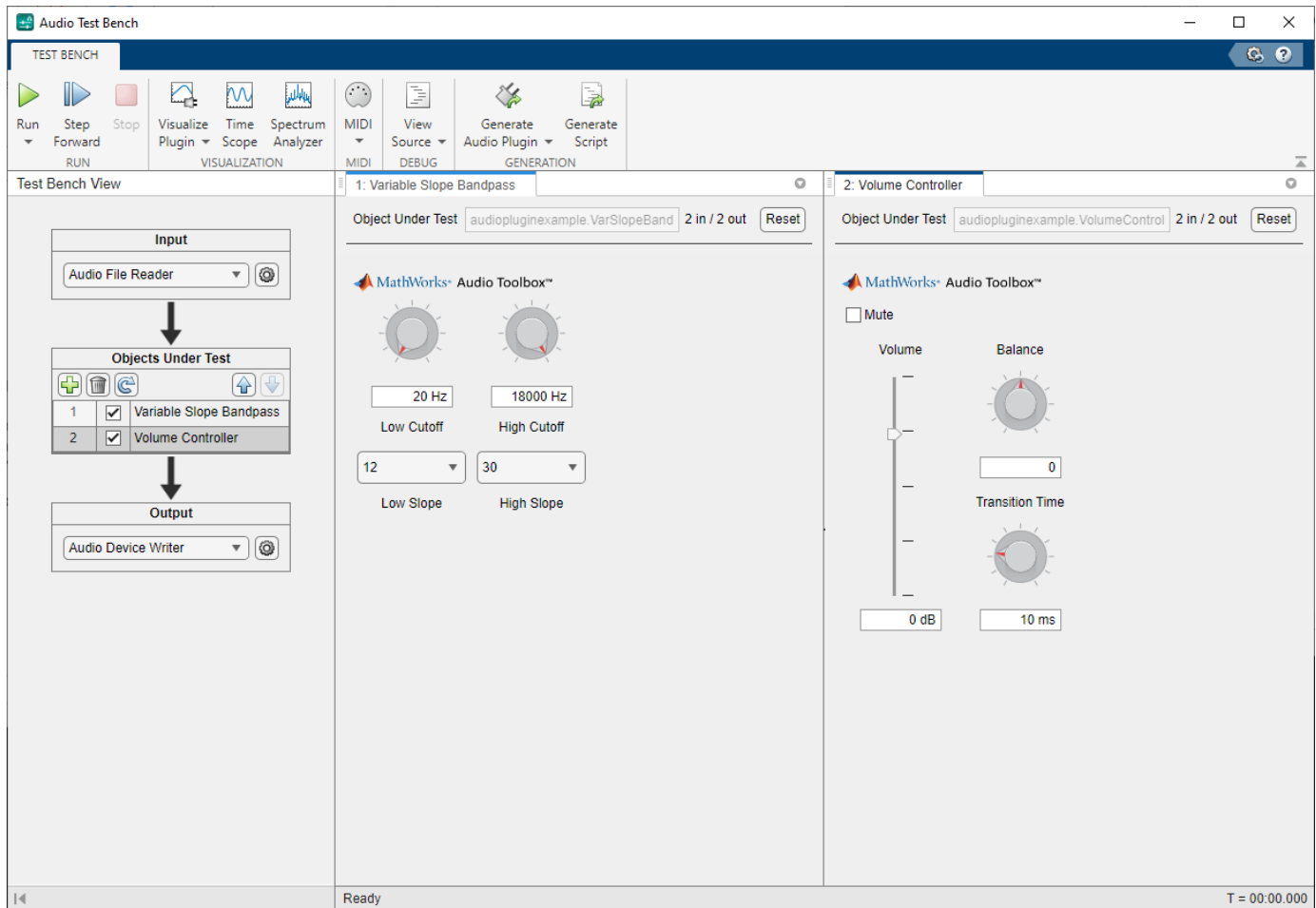
and press **Enter**. Alternatively, you can click the  button to browse for a file containing a plugin class definition or an external plugin binary. The **Audio Test Bench** automatically displays the tunable parameters of the `audiopluginexample.VarSlopeBandpassFilter` audio plugin.






The mapping between the tunable parameters of your object and the UI controls on the **Audio Test Bench** is determined by `audioPluginInterface` and `audioPluginParameter` in the class definition of your object.


Under **Objects Under Test** in the **Test Bench View**, click  and add the `audiopluginexample.VolumeController` plugin. The two plugins are now connected in a cascade where the `audiopluginexample.VarSlopeBandpassFilter` plugin processes the input signal, and its output is then processed by the `audiopluginexample.VolumeController` plugin.

Right-click the **Volume Controller** tab and select the Left/Right configuration under Tile All to view the tunable parameters of both plugins side by side.




You can change the order of the plugins in the cascade by selecting a plugin from the **Objects Under Test** and clicking  or  to move it up or down. You can also remove a plugin by selecting it and clicking .

Run Audio Test Bench

To run the **Audio Test Bench** and stream audio through your plugins, click . Use the UI controls to tune the plugin 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 UI controls to their initial positions, select the plugin from the **Objects Under Test** and 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  and select the plugin from the dropdown list. For this example, select `Volume Controller`.

You can inspect the source code of your audio plugin, set breakpoints on it, and modify the code. Set a breakpoint in the `set.TransitionTime` function and then click  on the **Audio Test Bench**.



```

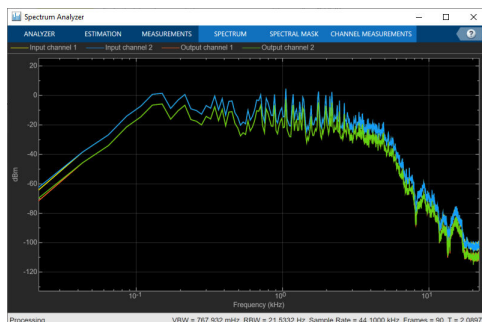
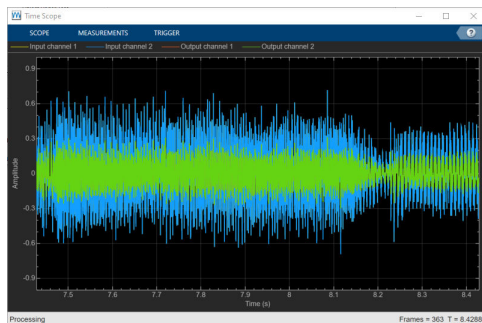
142
143
144
145
146
...
function set.TransitionTime(plugin,period)
    plugin.TransitionTime = period;
    plugin.DampedGain.TransitionTime = period/1000;
    plugin.DampedBalance.TransitionTime = period/1000;
end


```

The **Audio Test Bench** runs your plugin until it reaches the breakpoint. To reach the breakpoint, move the **Transition Time** dial. To stop debugging, remove the breakpoint and click **Continue** in the MATLAB editor.

Open Scopes

To open a time scope to visualize the time-domain input and output, click . To open a spectrum analyzer to visualize the frequency-domain input and output, click . The scopes display input to the **Audio Test Bench** and the output of the last plugin in the cascade.




To release objects and stop the audio stream loop, click .

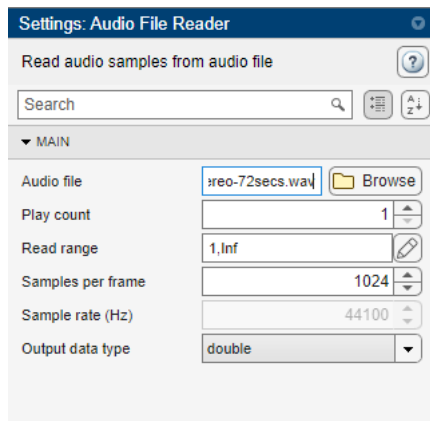
Configure Input to Audio Test Bench

The **Input** list contains these options corresponding to System objects:

- Audio File Reader -- `dsp.AudioFileReader`
- Audio Device Reader -- `audioDeviceReader`
- Audio Oscillator -- `audioOscillator`
- Wavetable Synthesizer -- `wavetableSynthesizer`
- Chirp Signal -- `dsp.Chirp`
- Colored Noise -- `dsp.ColoredNoise`



The Audio Device Reader option is not supported in MATLAB Online.

- 1 Select Audio File Reader.
- 2 Click  to open the settings panel 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`

Click  again to close the settings panel. To run the audio test bench with your new input, click .


To release your output object and stop the audio stream loop, click .

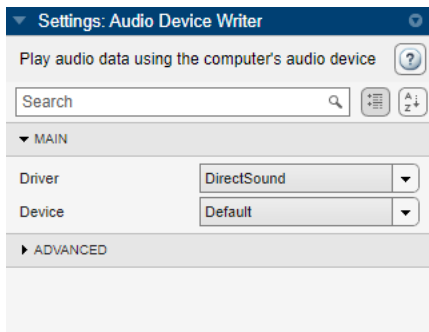
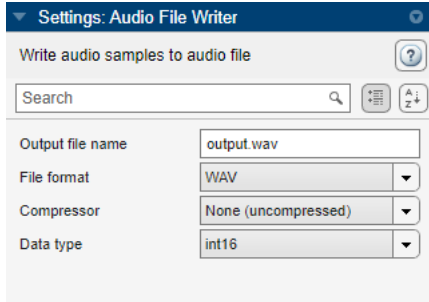
Configure Output from Audio Test Bench

The **Output** list contains these options corresponding to System objects:

- 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.


The Audio Device Writer and Both options are not supported in MATLAB Online.

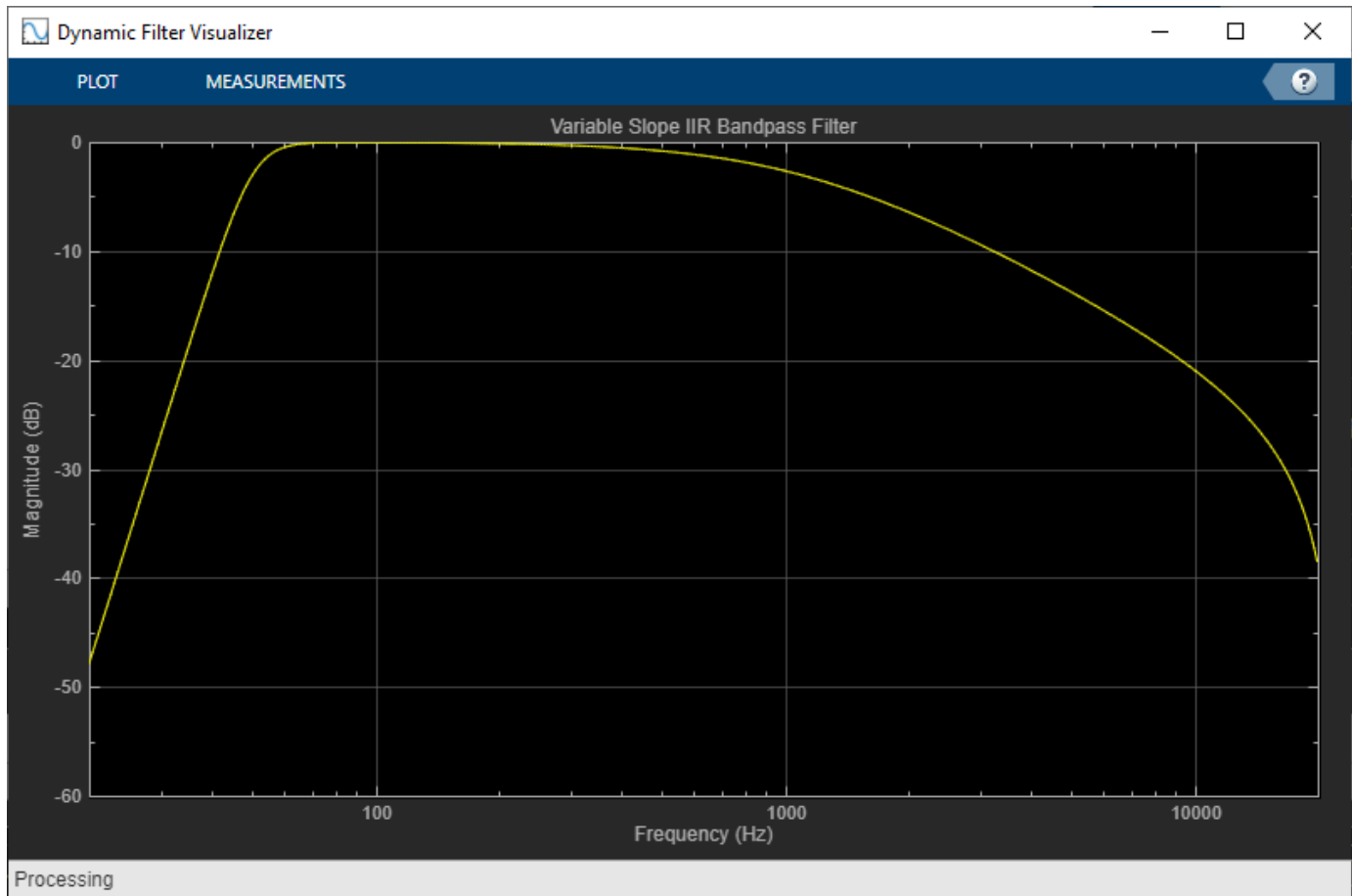
- 1 Choose to output to device and file by selecting Both from the **Output** menu.
- 2 To open settings panels for Audio Device Writer and Audio File Writer configuration, click .



Call Custom Visualization of Audio Plugin


If your audio plugin has a custom visualization method, you can view the visualization in the **Audio Test Bench**. To open the custom visualization of

`audiopluginexample.VarSlopeBandpassFilter`, click  and select Variable Slope Bandpass. 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 available only in MATLAB and not in 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  and select **Variable Slope Bandpass**. Synchronize the **LowCutoff** and **HighCutoff** properties with MIDI controls you choose. Click **OK**.


See `configureMIDI` for more information.

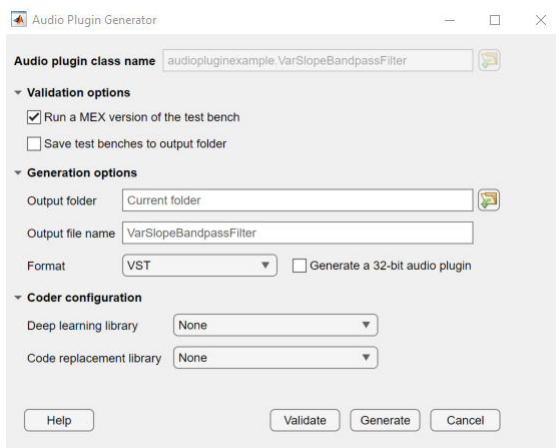
Play the Audio and Save the Output File

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 according to the **Audio File Writer** settings you configured for the **Output**.

Audio playback is not supported in MATLAB Online.

Validate and Generate Audio Plugin


To open the validation and generation dialog box, click  and select Variable Slope Bandpass.



You can validate your MATLAB audio plugin code and generate audio plugin binaries. In the **Coder configuration** section, you can specify libraries for deep learning and code replacement when generating plugins. See `generateAudioPlugin`, `validateAudioPlugin`, and `audioPluginConfig` for more information.

Plugin generation is not supported in MATLAB Online.

Generate MATLAB Script

To generate a MATLAB script that implements a test bench with the current app settings for the cascade of plugins, click .

```

untitled * x +
1 % Test bench script for: Variable Slope Bandpass, Volume Controller.
2 % Generated by Audio Test Bench on 19-Jan-2023 09:11:06 UTC-05:00.
3
4 % Create test bench input and output
5 fileReader = dsp.AudioFileReader('Filename','C:\Program Files\MATLAB\R2023a
6 deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
7
8 % Create scopes
9 timeScope = timescope(SampleRate=fileReader.SampleRate, ...
10     TimeSpanSource="property", ...
11     TimeSpan=1, ...
12     AxesScaling="manual", ...
13     BufferLength=176400, ...
14     ChannelNames=["Input channel 1","Output channel 1"], ...
15     Position=[2933 549 800 500], ...
16     YLimits=[-1 1], ...
17     ShowLegend=true);
18
19 % Create the objects under test
20 sut1 = audiopluginexample.VarSlopeBandpassFilter;
21 setSampleRate(sut1,fileReader.SampleRate);
22 sut1.LowCutoff = 45.7776;
23 sut1.HighCutoff = 10205.7;
24 sut1.LowSlope = '30';
25 sut1.HighSlope = '12';
26 sut2 = audiopluginexample.VolumeController;
27 setSampleRate(sut2,fileReader.SampleRate);
28 sut2.Gain = -5.26474;
29
30 % Open visualizer
31 % visualize(sut1);
32
33 % Open parameter tuners for interactive tuning during simulation
34 tuner1 = parameterTuner(sut1);
35 tuner2 = parameterTuner(sut2);
36 drawnow
37
38 % Stream processing loop
39 nUnderruns = 0;
40 while ~isDone(fileReader)
41     % Read from input, process, and write to output
42     in = fileReader();
43     out1 = sut1(in);
44     out2 = process(sut2,out1);
45     nUnderruns = nUnderruns + deviceWriter(out2);
46
47     % Visualize input and output data in scopes
48     timeScope(in,out2);
49     specScope(in,out2);
50
51     % Process parameterTuner callbacks
52     drawnow limitrate
53 end

```


You can modify the code for complete control over the test bench environment.

See Also

Audio Test Bench | `validateAudioPlugin` | `generateAudioPlugin` | `audioPlugin`

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

Deep Learning

Audio Plugin Examples

For a list of available audio plugins, see the online documentation.
--

See Also

Audio Test Bench | `audioPlugin` | `audioPluginSource` | `audioPluginInterface` | `audioPluginParameter`

More About

- “Develop, Analyze, and Debug Plugins In Audio Test Bench” 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 1-391.

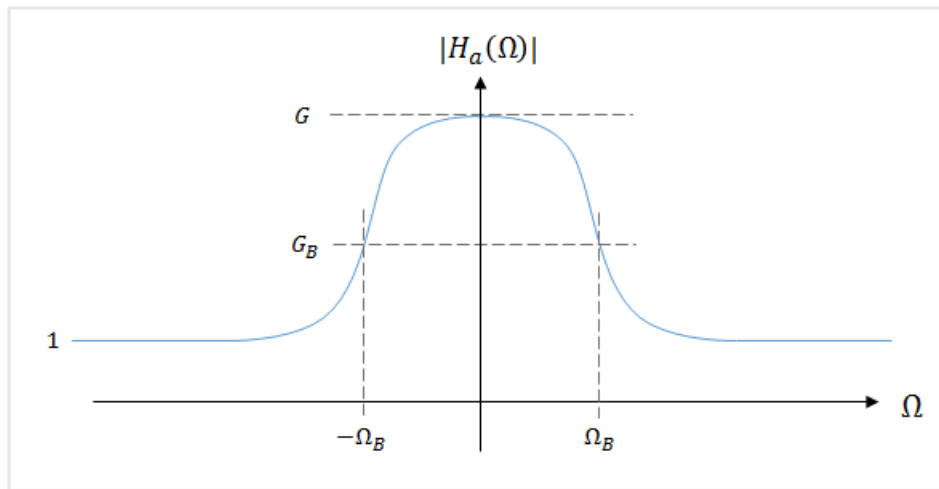
Equalization Design Using Audio Toolbox

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
- $r = \begin{cases} 0 & N \text{ even} \\ 1 & N \text{ odd} \end{cases}$
- $g = G^{1/N}$
- $\beta = \Omega_B \times \left(\sqrt{\frac{G^2 - G_B^2}{G_B^2 - 1}} \right)^{-1/N} = \tan\left(\frac{\pi\Delta\omega}{2}\right) \times \left(\sqrt{\frac{G^2 - G_B^2}{G_B^2 - 1}} \right)^{-1/N}$, 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
$D_0 = \beta + 1$	$D_i = \beta^2 + 2s_i\beta + 1$
$b_{00} = (1 + g\beta)/D_0$	$b_{i0} = (g^2\beta^2 + 2gs_i\beta + 1)/D_i$
$b_{01} = -2\cos(\omega_0)/D_0$	$b_{i1} = -4c_0(1 + gs_i\beta)/D_i$
$b_{02} = (1 - g\beta)/D_0$	$b_{i2} = 2(1 + 2\cos^2(\omega_0) - g^2\beta^2)/D_i$
$a_{01} = -2\cos(\omega_0)/D_0$	$b_{i3} = -4c_0(1 - gs_i\beta)/D_i$
$a_{02} = (1 - \beta)/D_0$	$b_{i4} = (g^2\beta^2 - 2gs_i\beta + 1)/D_i$
	$a_{i1} = -4c_0(1 + s_i\beta)/D_i$
	$a_{i2} = 2(1 + 2\cos^2(\omega_0) - \beta^2)/D_i$
	$a_{i3} = -4\cos(\omega_0)(1 - s_i\beta)/D_i$
	$a_{i4} = (\beta^2 - 2s_i\beta + 1)/D_i$

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 $A = \text{sqrt}(G)$, $\Omega_B \cong \alpha$ 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]:

$$B = \frac{\omega_0}{\sin\omega_0} \times BW$$

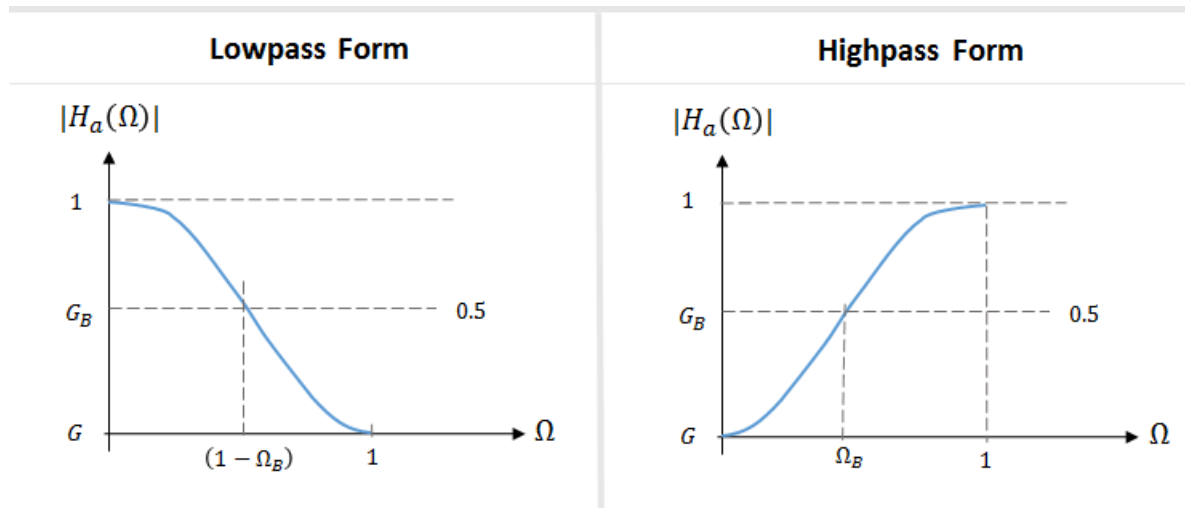
Substituting the approximation for B into the Ω_B equation yields the definition of α in [2]:

$$\alpha = \sin(\omega_0) \sinh\left(\frac{\ln 2}{2} \times \frac{\omega_0}{\sin\omega_0} \times BW\right)$$

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
$D_0 = 1 + \tan\left(\pi\frac{\Delta\omega}{2}\right)$	$D_i = \tan^2\left(\pi\frac{\Delta\omega}{2}\right) + 2s_i\tan\left(\pi\frac{\Delta\omega}{2}\right) + 1$
$b_{00} = 1/D_0$	$b_{i0} = 1/D_i$
$b_{01} = -2\cos(\omega_0)/D_0$	$b_{i1} = -4\cos(\omega_0)/D_i$
$b_{02} = 1/D_0$	$b_{i2} = 2(1 + 2\cos^2(\omega_0))/D_i$
$a_{01} = -2\cos(\omega_0)/D_0$	$b_{i3} = -4\cos(\omega_0)/D_i$
$a_{02} = \left(1 - \tan\left(\pi\frac{\Delta\omega}{2}\right)\right)/D_0$	$b_{i4} = 1/D_i$
	$a_{i1} = -4\cos(\omega_0)\left(1 + s_i\tan\left(\pi\frac{\Delta\omega}{2}\right)\right)/D_i$
	$a_{i2} = 2\left(1 + 2\cos^2(\omega_0) - \tan^2\left(\pi\frac{\Delta\omega}{2}\right)\right)/D_i$
	$a_{i3} = -4\cos(\omega_0)\left(1 - s_i\tan\left(\pi\frac{\Delta\omega}{2}\right)\right)/D_i$
	$a_{i4} = \left(\tan^2\left(\pi\frac{\Delta\omega}{2}\right) - 2s_i\tan\left(\pi\frac{\Delta\omega}{2}\right) + 1\right)/D_i$

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	$\alpha = \frac{\sin(\omega_0)}{2} \sqrt{\left(A + \frac{1}{A}\right) \left(\frac{1}{\text{slope}} - 1\right) + 2A}$ $\omega_0 = 2\pi \frac{\text{Cutoff Frequency}}{F_s}$

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

designVarSlopeFilter | designParamEQ | designShelvingEQ | multibandParametricEQ | graphicEQ

More About

- "Parametric Equalizer Design" on page 1-391
- "Graphic Equalization" on page 1-189
- "Octave-Band and Fractional Octave-Band Filters" on page 1-399
- "Audio Weighting Filters" on page 1-199

Deployment

- “Desktop Real-Time Audio Acceleration with MATLAB Coder” on page 14-2
- “What is C Code Generation from MATLAB?” on page 14-5

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.

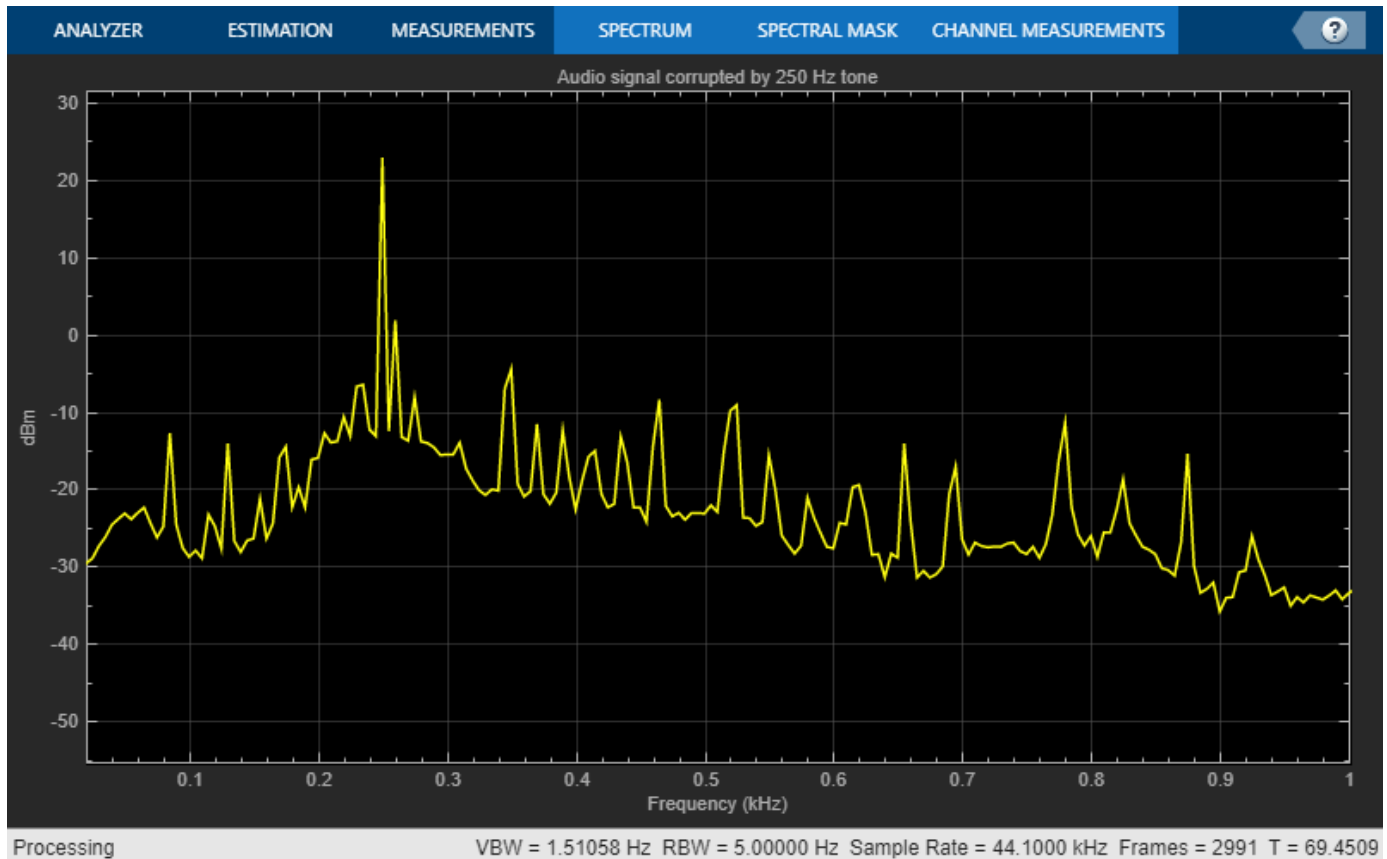
The helper function used in this example is `helperAudioToneRemoval`. The function reads an audio signal corrupted by a 250 Hz sinusoidal tone from a file. `helperAudioToneRemoval` uses a notch filter to remove the interfering tone and writes the filtered signal to a file.

You can visualize the corrupted audio signal using a spectrum analyzer.

```
reader = dsp.AudioFileReader("guitar_plus_tone.ogg");

scope = spectrumAnalyzer(SampleRate=reader.SampleRate, ...
    RBWSource="property",RBW=5, ...
    PlotAsTwoSidedSpectrum=false, ...
    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: 2.732519e+00

Next, generate a MEX-function from `helperAudioToneRemoval` using the MATLAB Coder function, `codegen` (MATLAB Coder).

```
codegen helperAudioToneRemoval
```

Code generation successful.

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: 9.064711e-01
```

```
fprintf("Speedup factor: %6.2f\n",t1/t2)
```

```
Speedup factor: 3.01
```

See Also

Related Examples

- “Generate Standalone Executable for Parametric Audio Equalizer” on page 1-272
- “Deploy Audio Applications with MATLAB Compiler” on page 1-275

What is C Code Generation from MATLAB?

You can use Audio Toolbox together with MATLAB Coder to:

- Create a MEX file to speed up your MATLAB application.
- Generate ANSI®/ISO® compliant C/C++ source code that implements your MATLAB functions and models.
- Generate a standalone executable that runs independently of MATLAB on your computer or another platform.

In general, the code you generate using the toolbox is portable ANSI C code. In order to use code generation, you need a MATLAB Coder license. For more information, see “Get Started with MATLAB Coder” (MATLAB Coder).

Using MATLAB Coder

Creating a MATLAB Coder MEX file can substantially accelerate your MATLAB code. It is also a convenient first step in a workflow that ultimately leads to completely standalone code. When you create a MEX file, it runs in the MATLAB environment. Its inputs and outputs are available for inspection just like any other MATLAB variable. You can then use MATLAB tools for visualization, verification, and analysis.

The simplest way to generate MEX files from your MATLAB code is by using the `codegen` function at the command line. For example, if you have an existing function, `myfunction.m`, you can type the commands at the command line to compile and run the MEX function. `codegen` adds a platform-specific extension to this name. In this case, the “mex” suffix is added.

```
codegen myfunction.m  
myfunction_mex;
```

Within your code, you can run specific commands either as generated C code or by using the MATLAB engine. In cases where an isolated command does not yet have code generation support, you can use the `coder.extrinsic` command to embed the command in your code. This means that the generated code reenters the MATLAB environment when it needs to run that particular command. This is also useful if you want to embed commands that cannot generate code (such as plotting functions).

To generate standalone executables that run independently of the MATLAB environment, create a MATLAB Coder project inside the MATLAB Coder Integrated Development Environment (IDE). Alternatively, you can call the `codegen` command in the command line environment with appropriate configuration parameters. A standalone executable requires you to write your own `main.c` or `main.cpp` function. See “Generating Standalone C/C++ Executables from MATLAB Code” (MATLAB Coder) for more information.

C/C++ Compiler Setup

Before using `codegen` to compile your code, you must set up your C/C++ compiler. For 32-bit Windows platforms, MathWorks® supplies a default compiler with MATLAB. If your installation does not include a default compiler, you can supply your own compiler. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks website. Install a compiler that is suitable for your platform, then read “Setting Up the C or C++ Compiler” (MATLAB Coder).

After installation, at the MATLAB command prompt, run `mex -setup`. You can then use the `codegen` function to compile your code.

Functions and System Objects That Support Code Generation

All Audio Toolbox functions and System objects support code generation.

See Also

Functions

`codegen` | `mex`

More About

- “Code Generation Workflow” (MATLAB Coder)
- Generate C Code from MATLAB Code Video

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

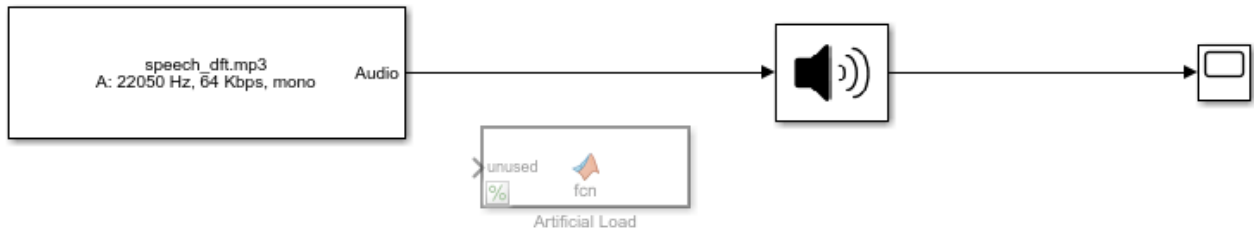
More About

- “MATLAB Programming for Code Generation” (MATLAB Coder)

Block Example Repository

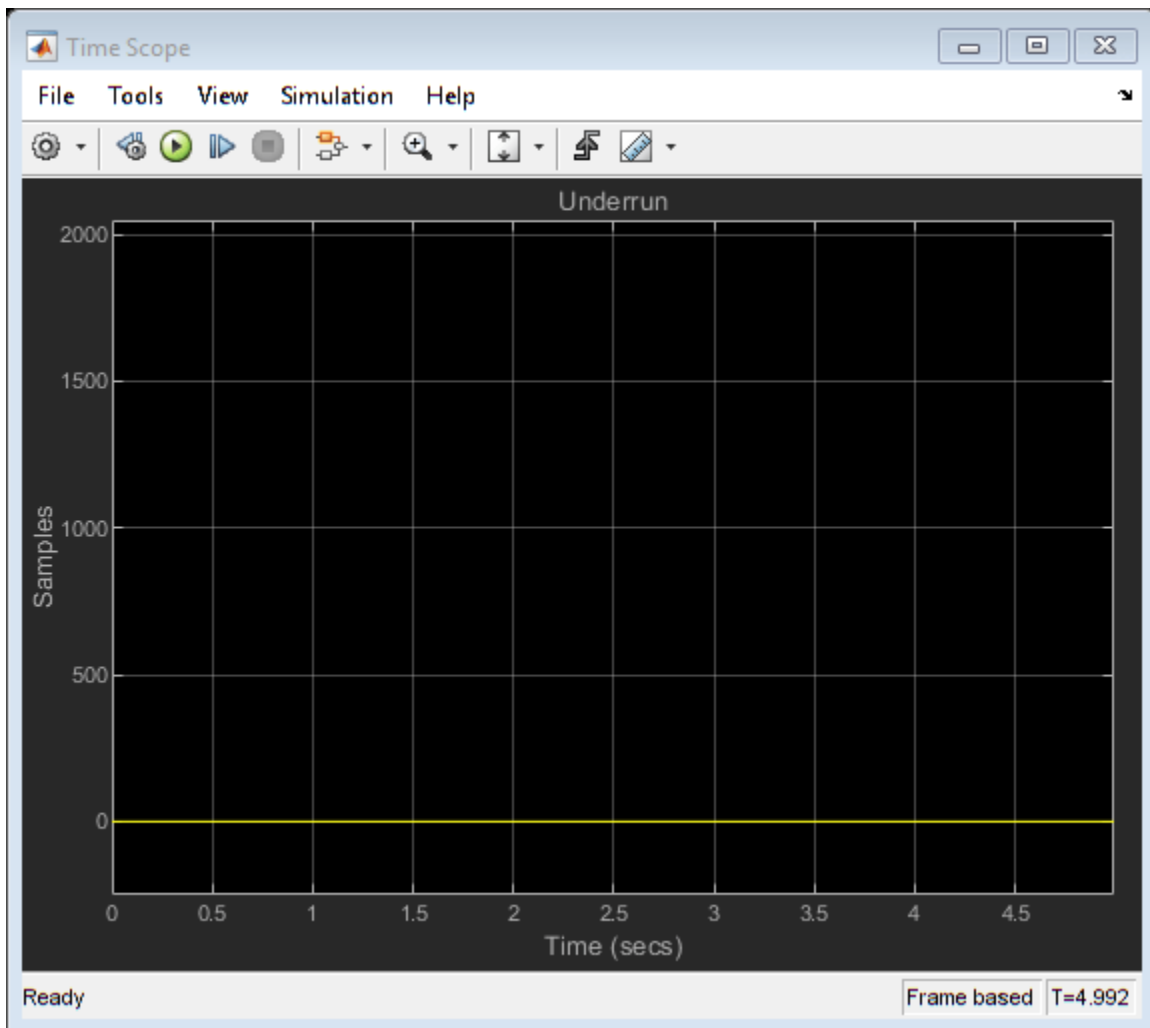
Decrease Underrun

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

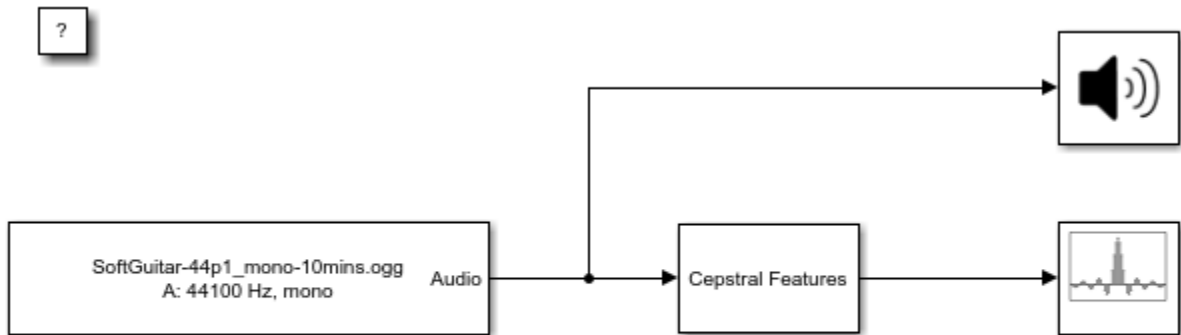
Block Example Repository

- “Extract Cepstral Coefficients” on page 17-3
- “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
- “Use Octave Filter Bank to Create Flanging Chorus Effect for Guitar Layers (Overdubs)” on page 17-50
- “Decompose Signal using Gammatone Filter Bank Block” on page 17-52
- “Visualize Filter Response of Multiband Parametric Equalizer Block” on page 17-54
- “Detect Music in Simulink Using YAMNet” on page 17-57
- “Compare Sound Classifier block with Equivalent YAMNet blocks” on page 17-60
- “Detect Air Compressor Sounds in Simulink Using YAMNet” on page 17-62
- “Design Auditory Filter Bank” on page 17-66
- “Design Mel Filter Bank” on page 17-67

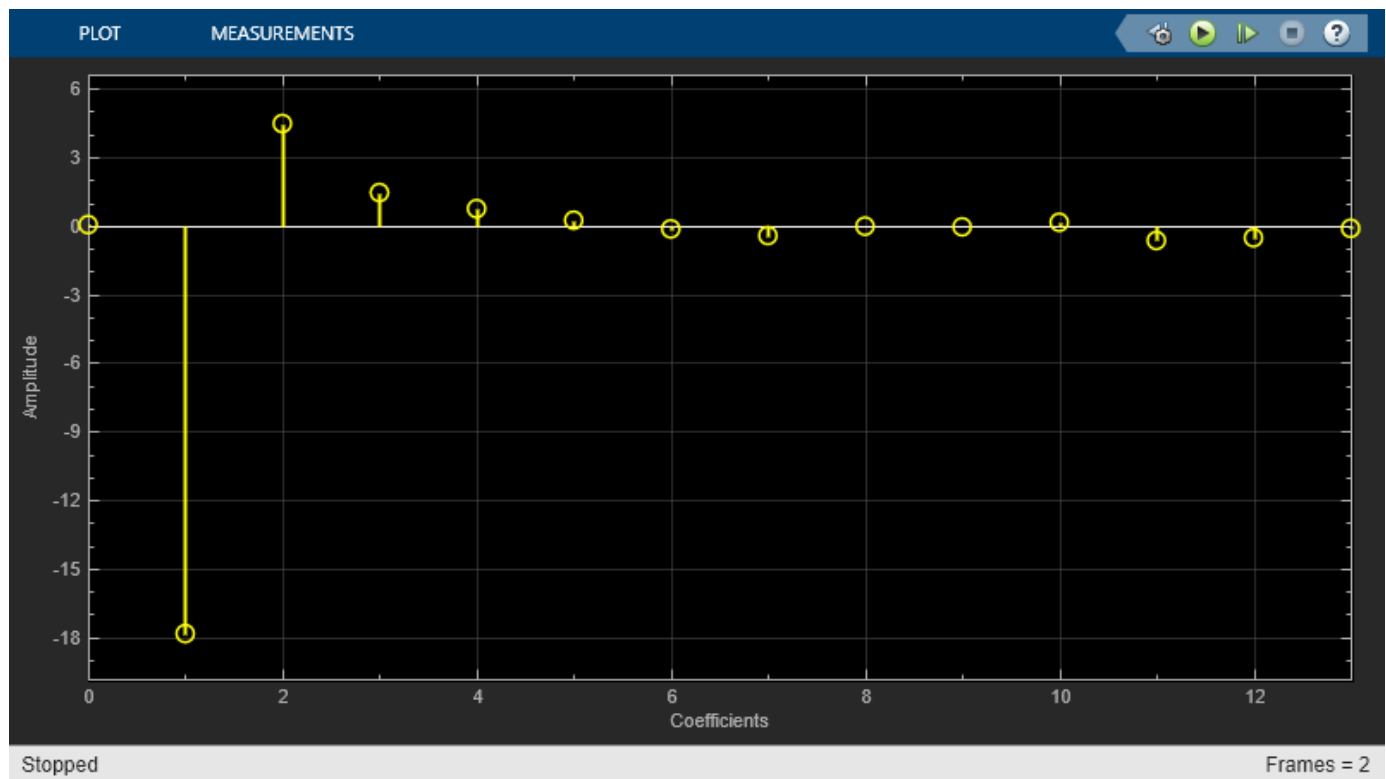
- “Extract Auditory Spectrogram” on page 17-68
- “Extract Mel Spectrogram” on page 17-69
- “Filter Audio Using Shelving Filter Block” on page 17-71
- “Compare VGGish Embeddings Block with Equivalent VGGish Blocks” on page 17-72
- “Extract GTCC from Audio in Simulink” on page 17-74
- “Include an Audio Plugin in Simulink” on page 17-75
- “Use VGGish Embeddings for Deep Learning in Simulink” on page 17-78

Extract Cepstral Coefficients

Use the Cepstral Feature Extractor block to extract and visualize cepstral coefficients from an audio file.



Copyright 2018 The MathWorks, Inc.

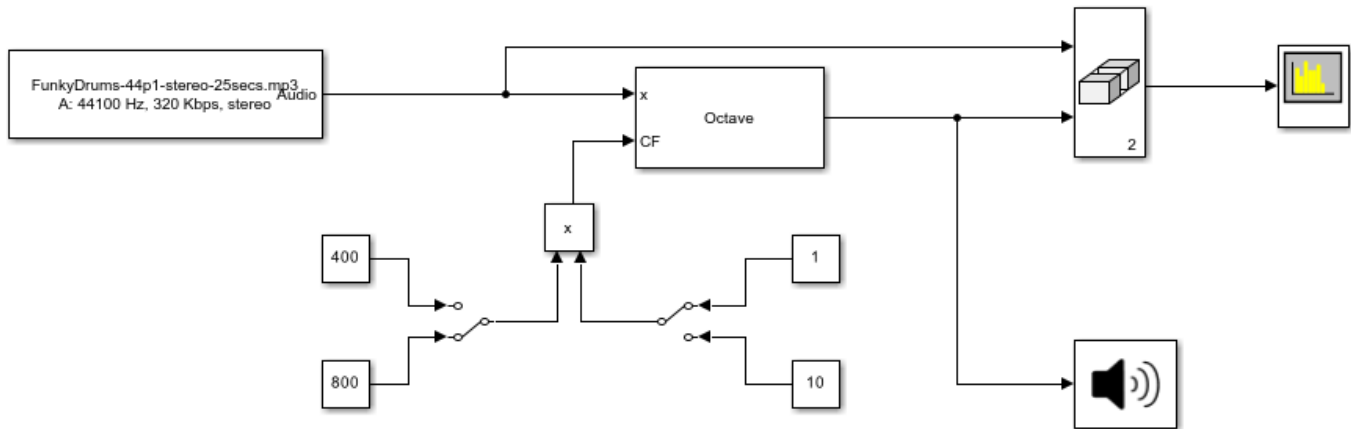


See Also

Cepstral Feature Extractor | mfcc | gtcc | Audio Device Writer | From Multimedia File | Array Plot

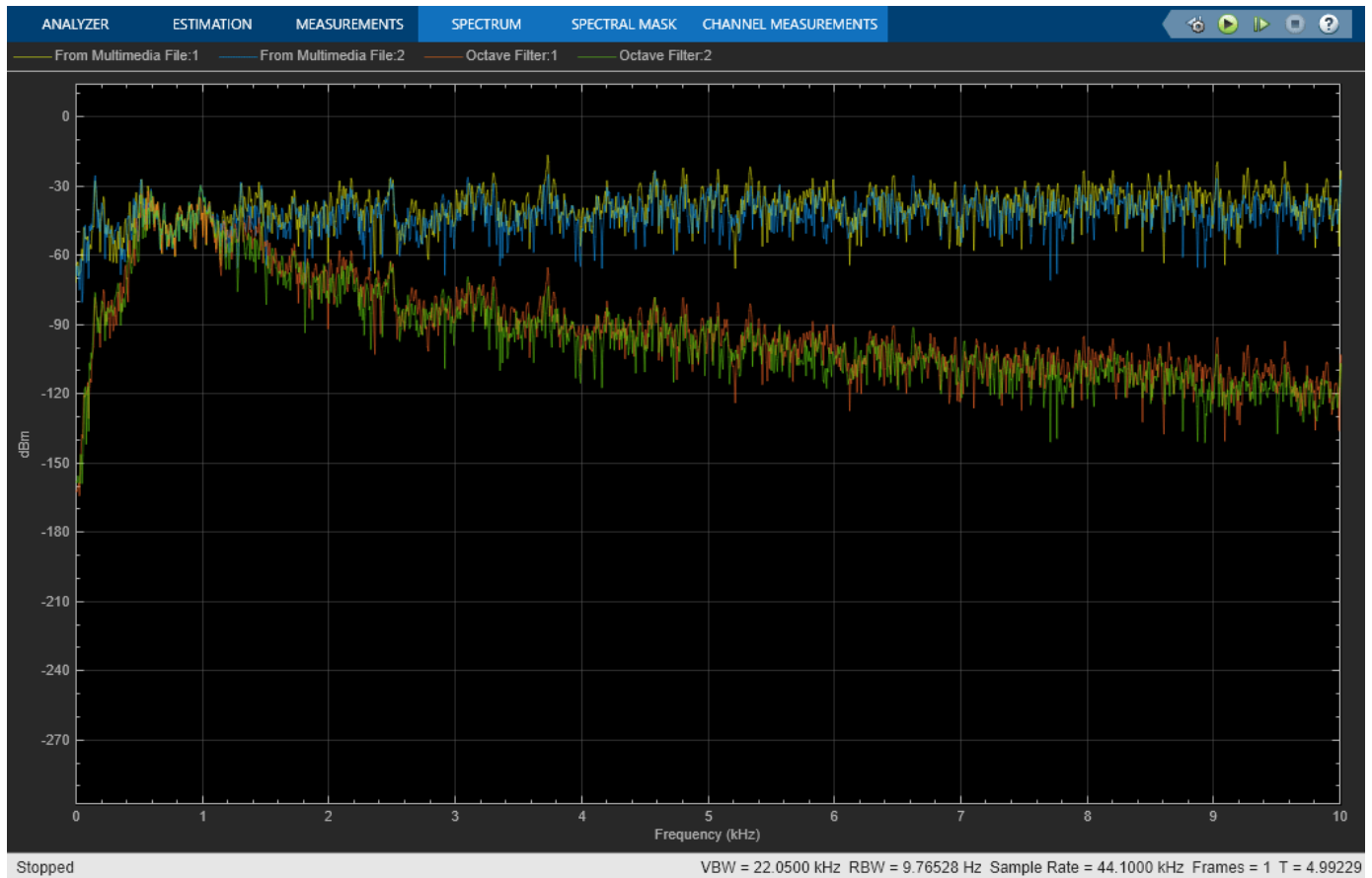
Tune Center Frequency Using Input Port

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.



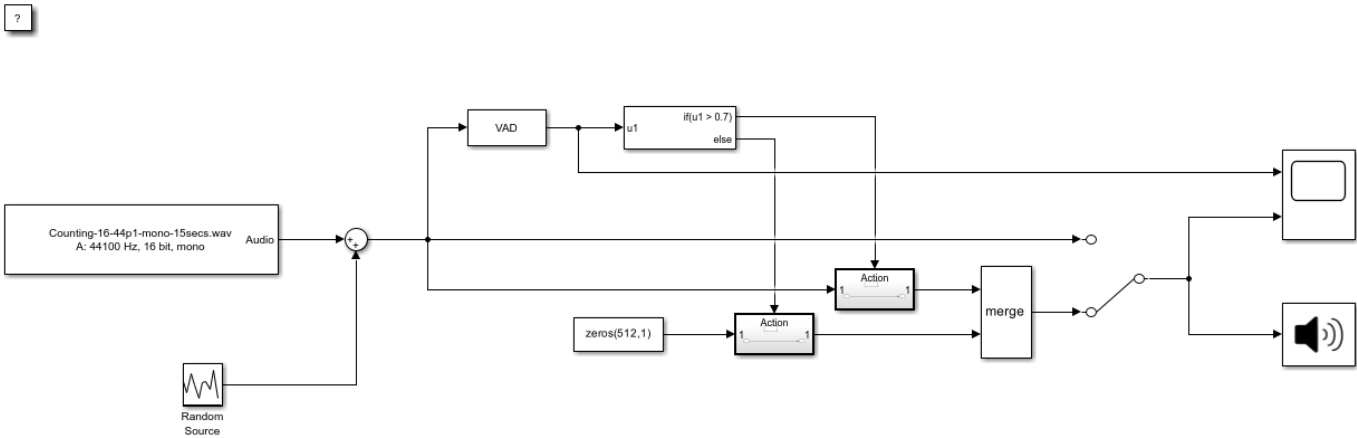
See Also

Audio Device Writer | From Multimedia File | Time Scope | Manual Switch | Octave Filter

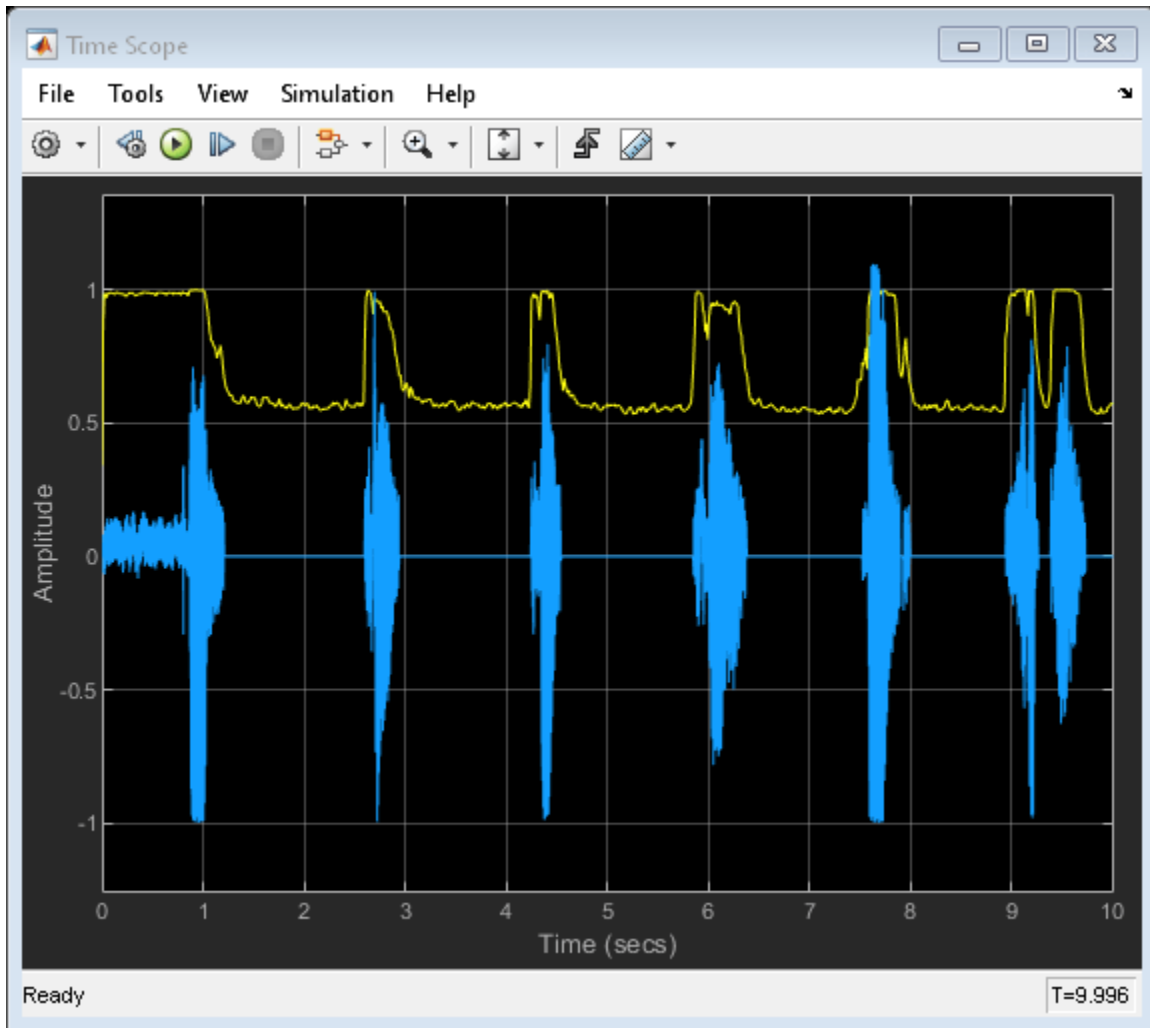
Gate Audio Signal Using VAD

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

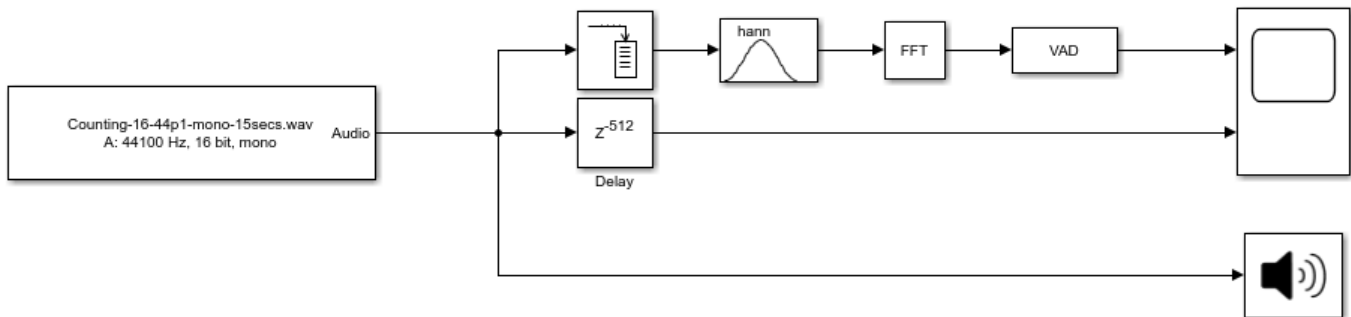
Voice Activity Detector | Audio Device Writer | From Multimedia File | Time Scope | Random Source | Manual Switch | If | If Action Subsystem

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.



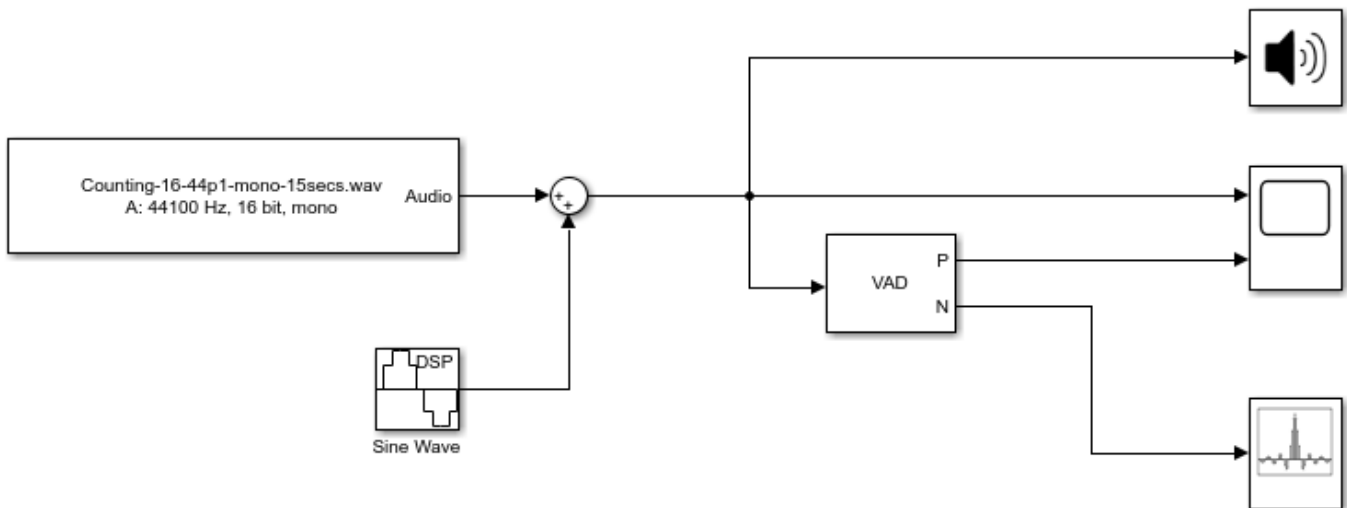
Copyright 2017 The MathWorks, Inc.

See Also

Voice Activity Detector | Audio Device Writer | From Multimedia File | Time Scope | Window Function | Buffer | Delay | FFT

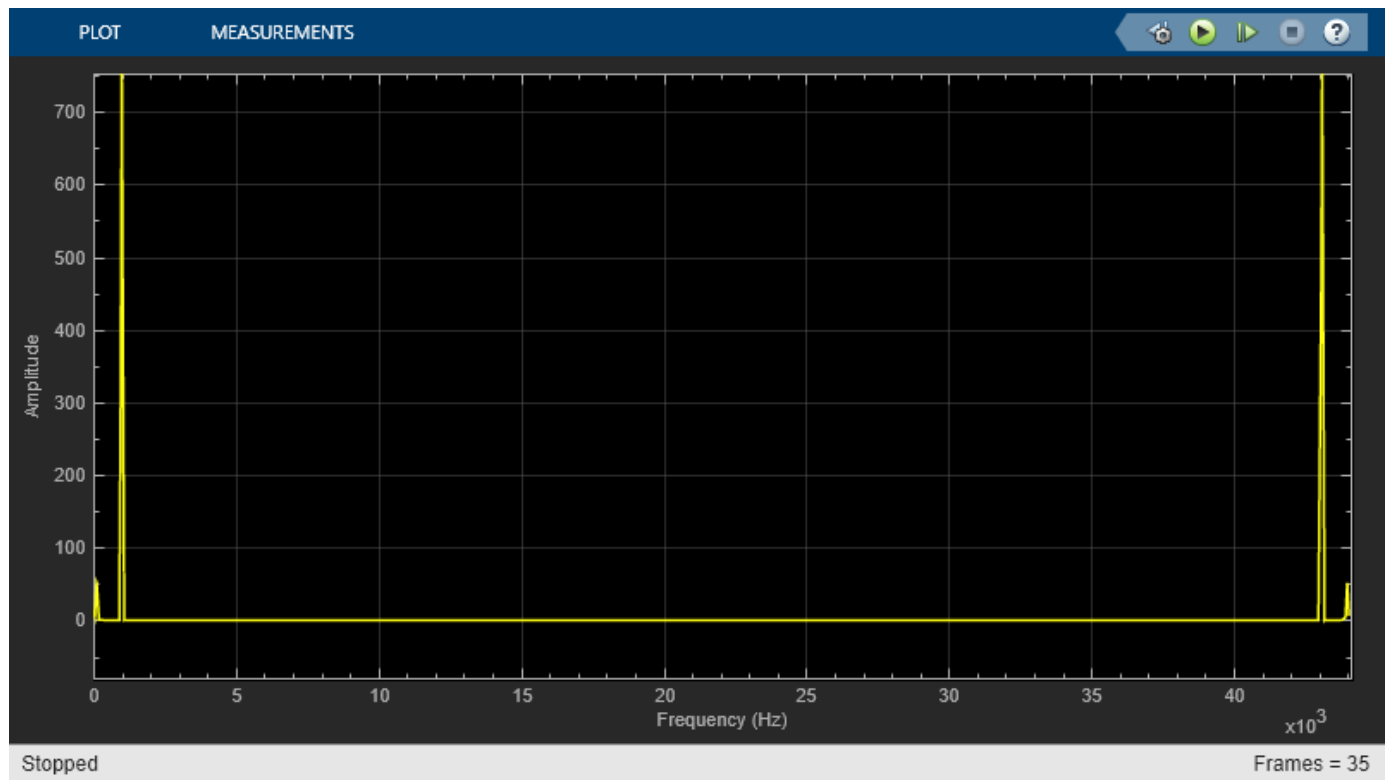
Visualize Noise Power

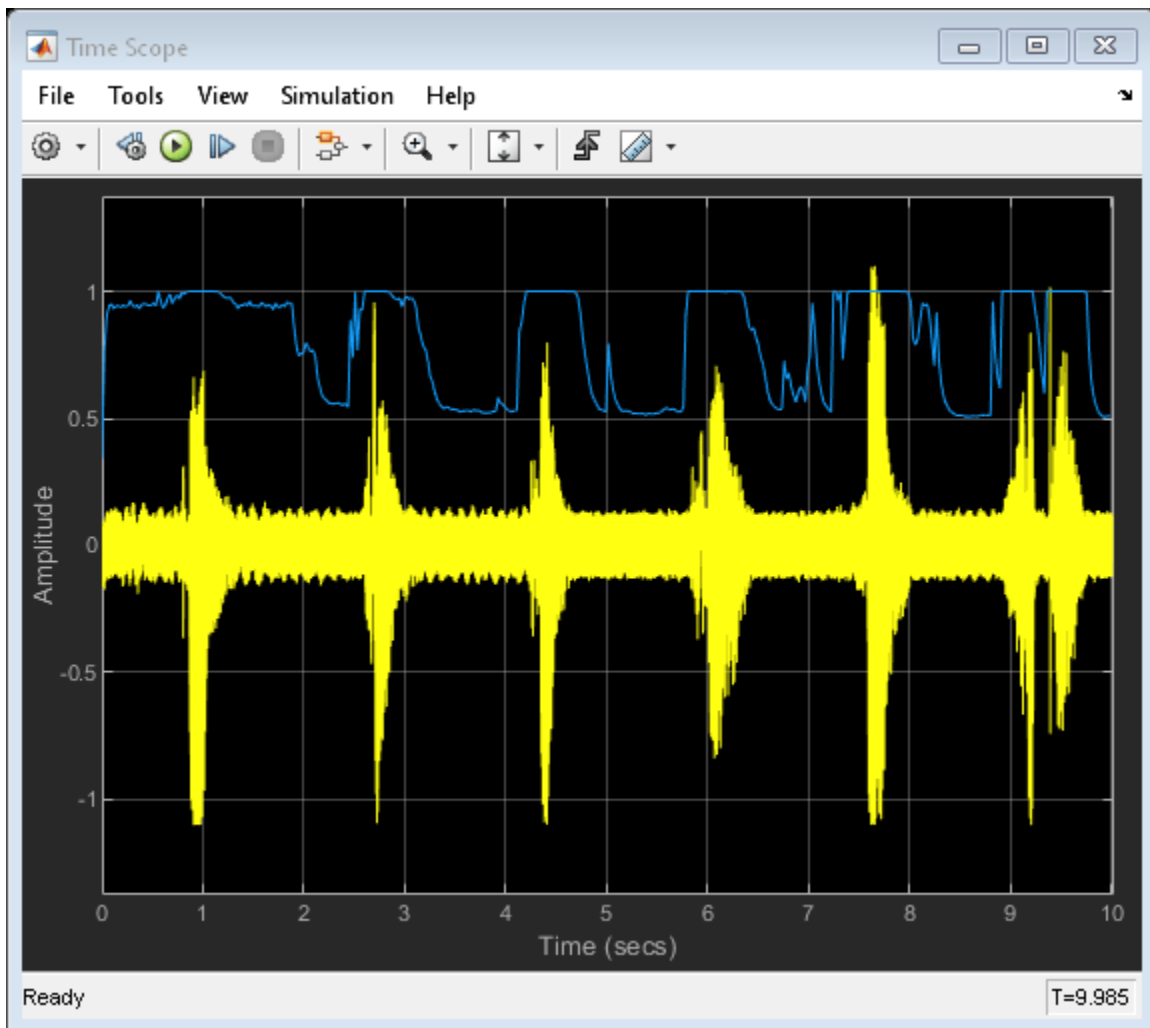
This model plots the noise power estimated by the Voice Activity Detector.



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.

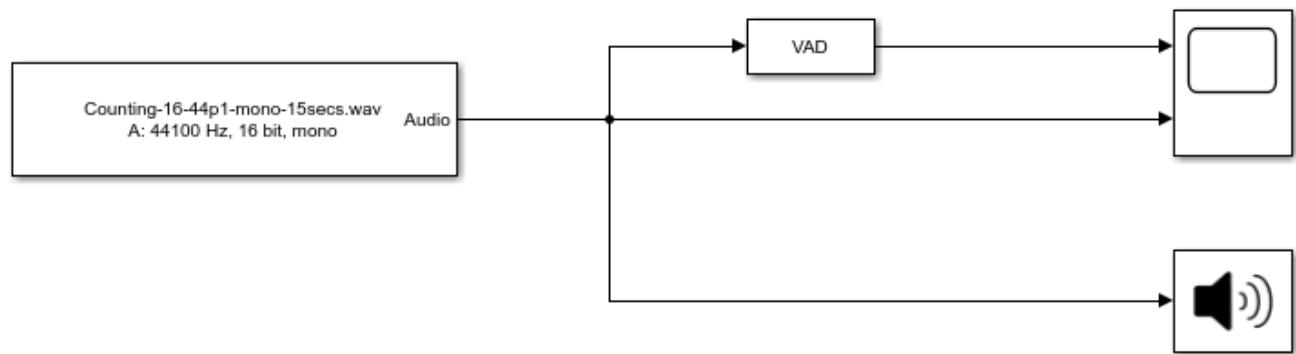
See Also

[Voice Activity Detector](#) | [Audio Device Writer](#) | [From Multimedia File](#) | [Time Scope](#) | [Array Plot](#) | [Sine Wave](#)

Detect Presence of Speech

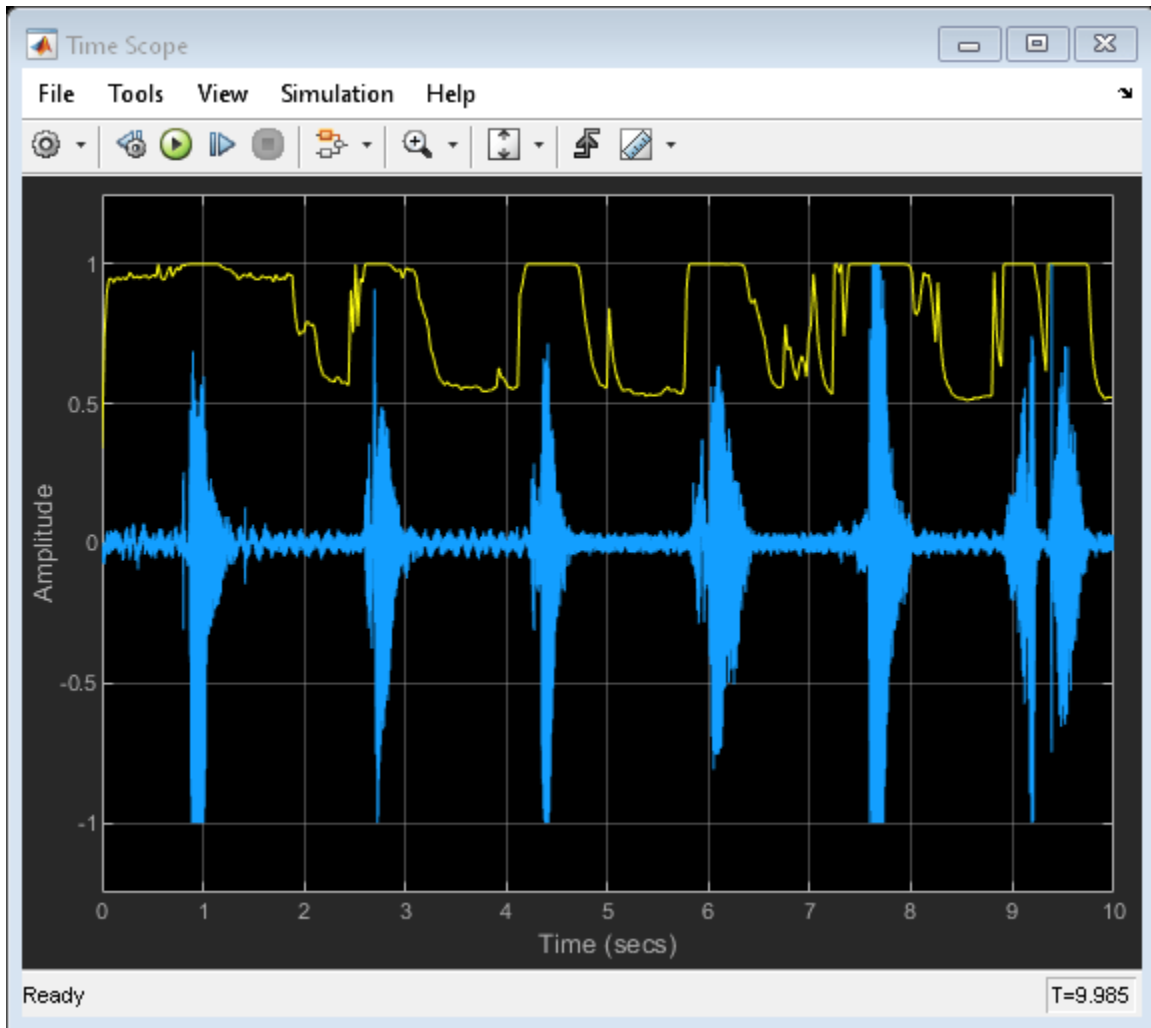
This model uses the Voice Activity Detector block to visualize the probability of speech presence in an audio signal.

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.

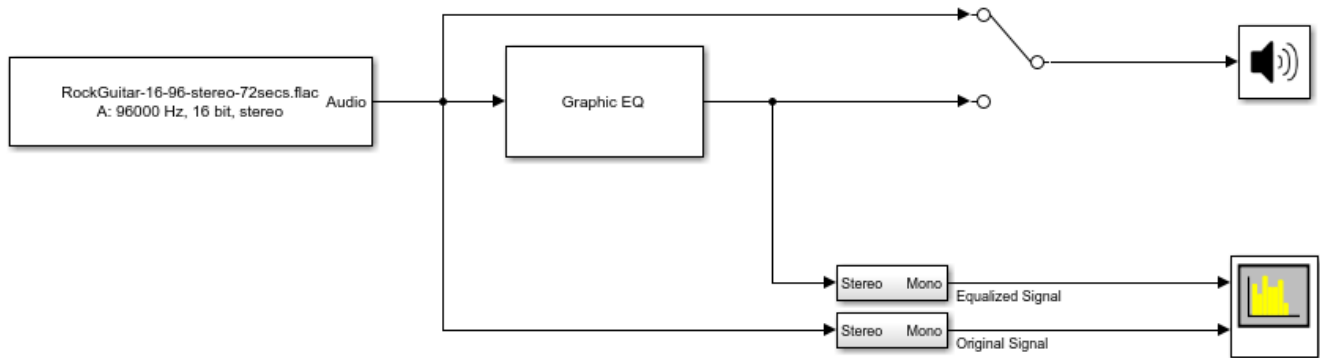


See Also

Voice Activity Detector | Audio Device Writer | From Multimedia File | Time Scope

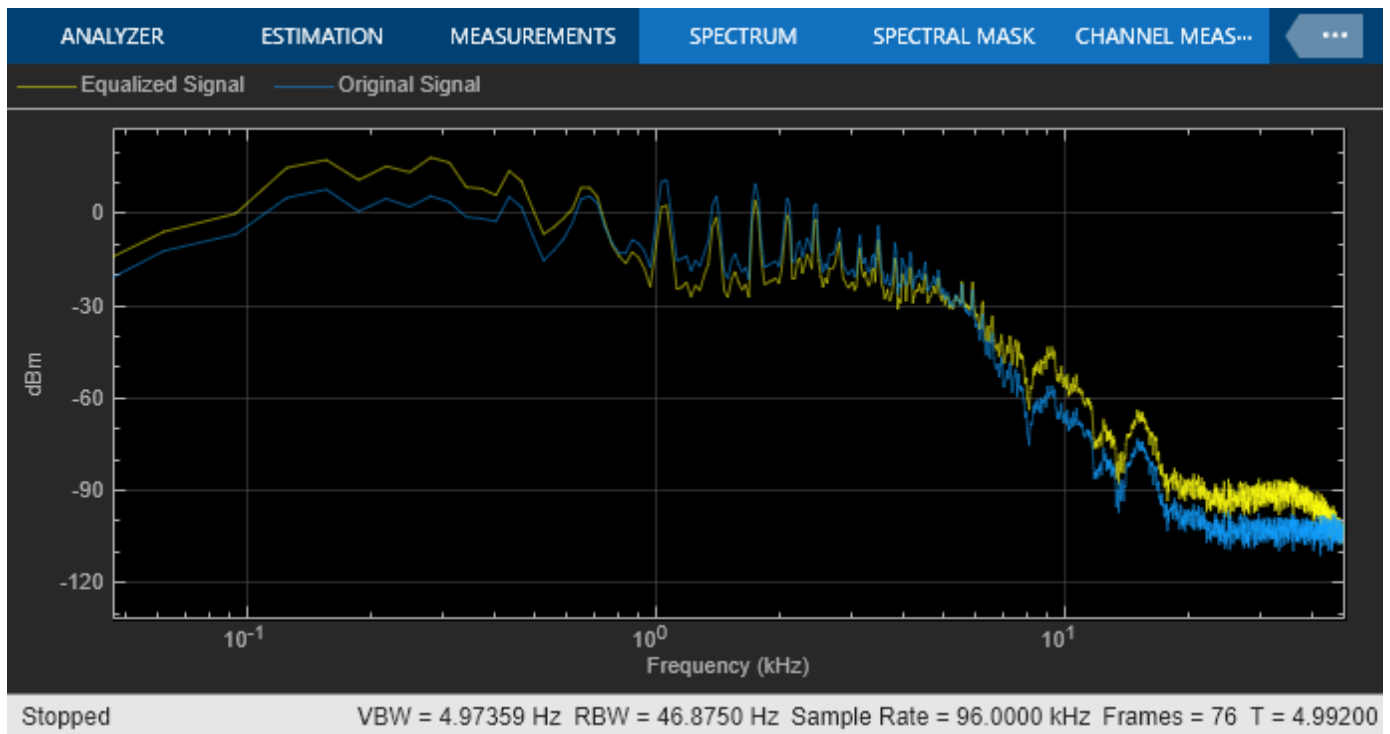
Perform Graphic Equalization

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

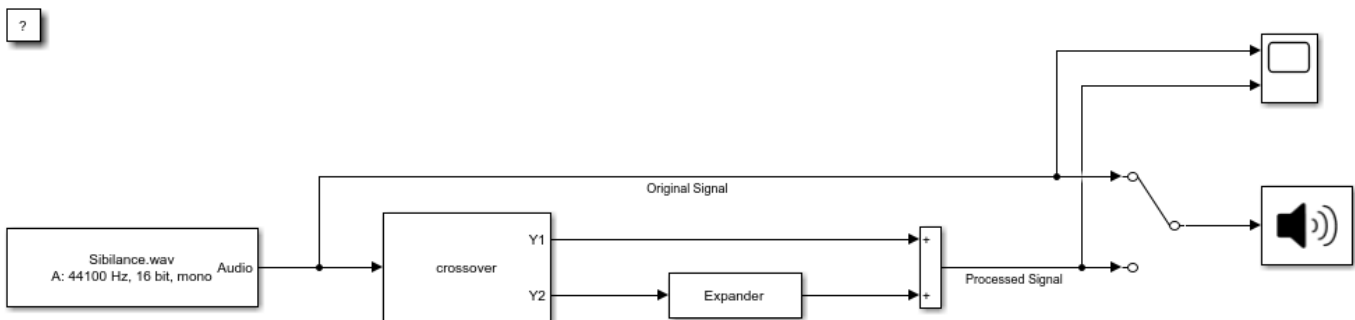
Graphic EQ | Audio Device Writer | From Multimedia File | Spectrum Analyzer

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](#) | [Time Scope](#) | [Expander](#) | [From Multimedia File](#) | [Crossover Filter](#)

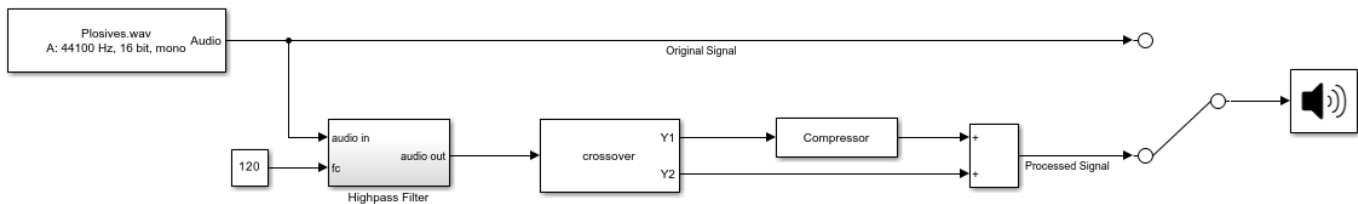
Diminish Plosives from Speech

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 displeasing 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.

?



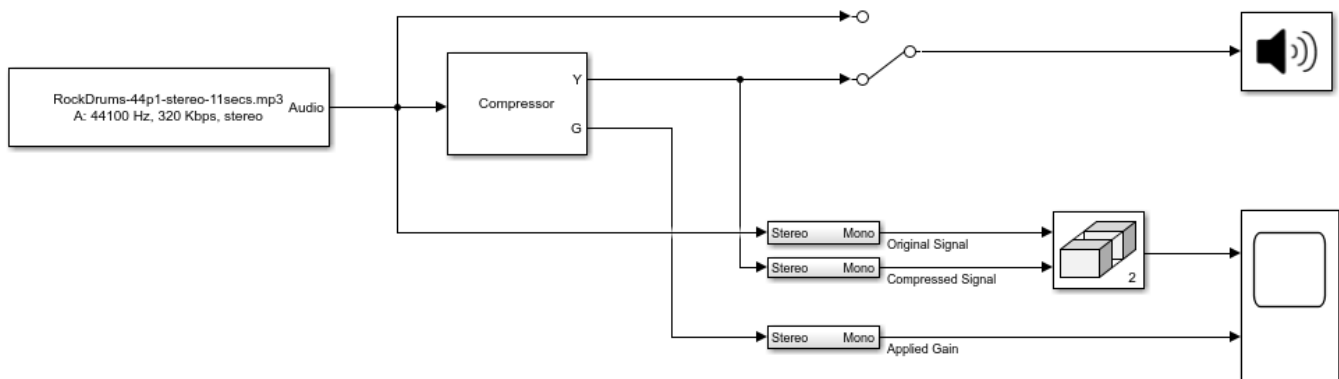
Copyright 2017 The MathWorks, Inc.

See Also

Audio Device Writer | Compressor | From Multimedia File | Crossover Filter

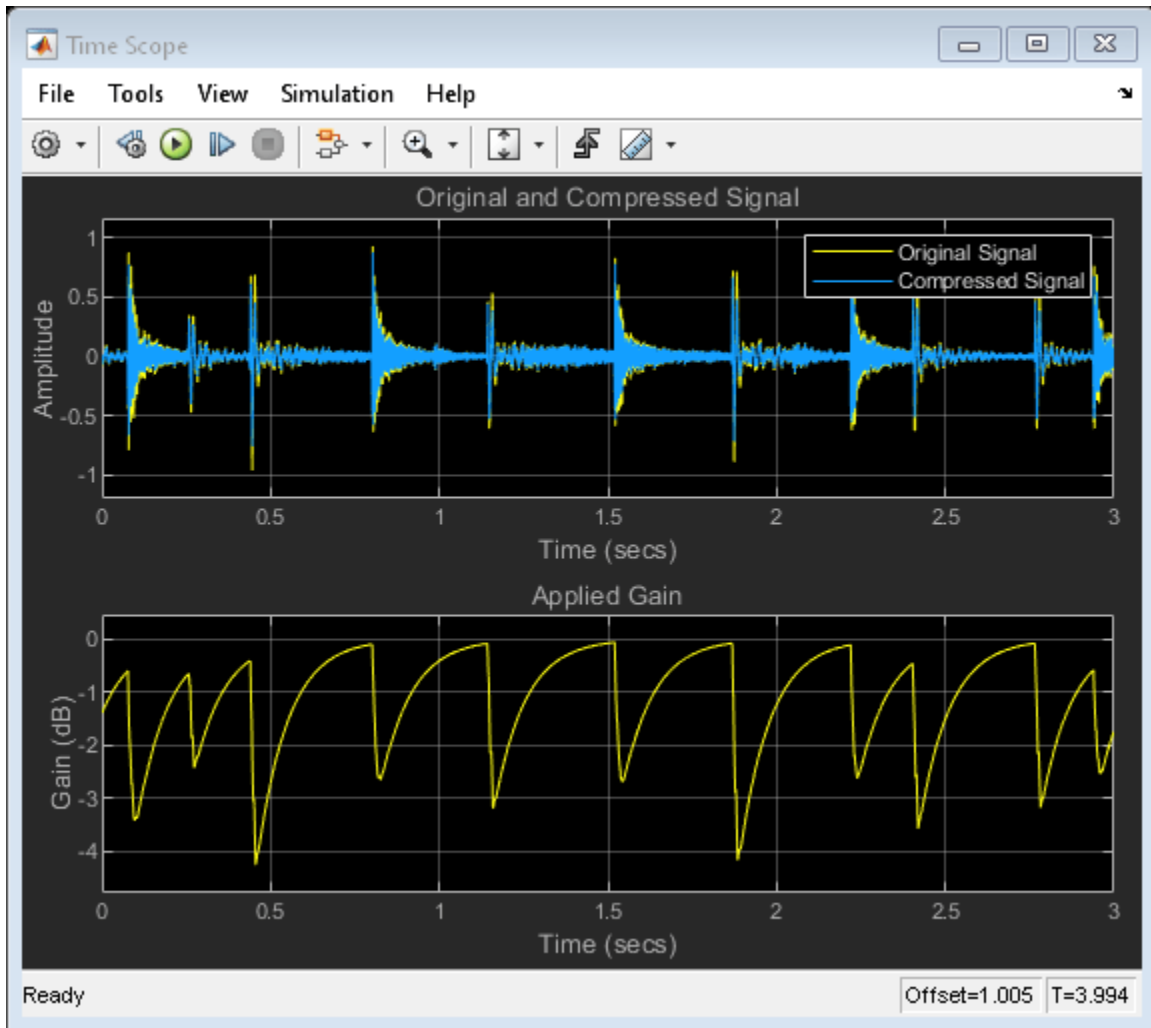
Suppress Loud Sounds

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

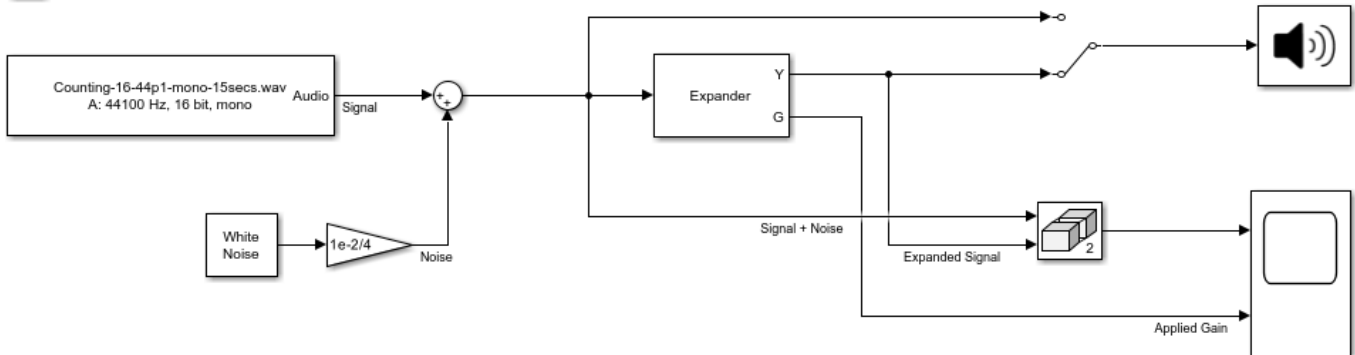
Audio Device Writer | Time Scope | From Multimedia File | Matrix Concatenate | Compressor

More About

- “Dynamic Range Control” on page 8-2

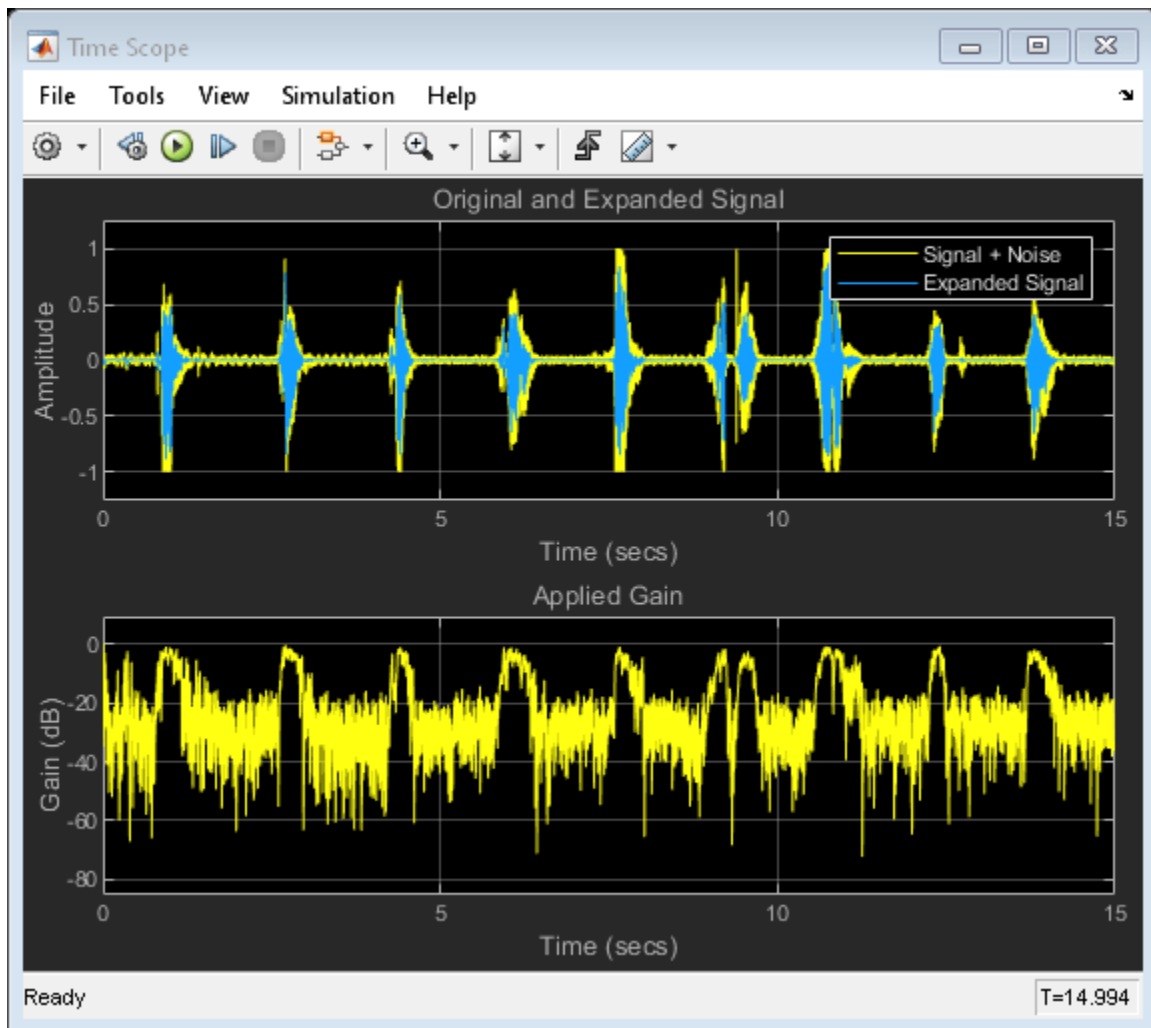
Attenuate Low-Level Noise

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

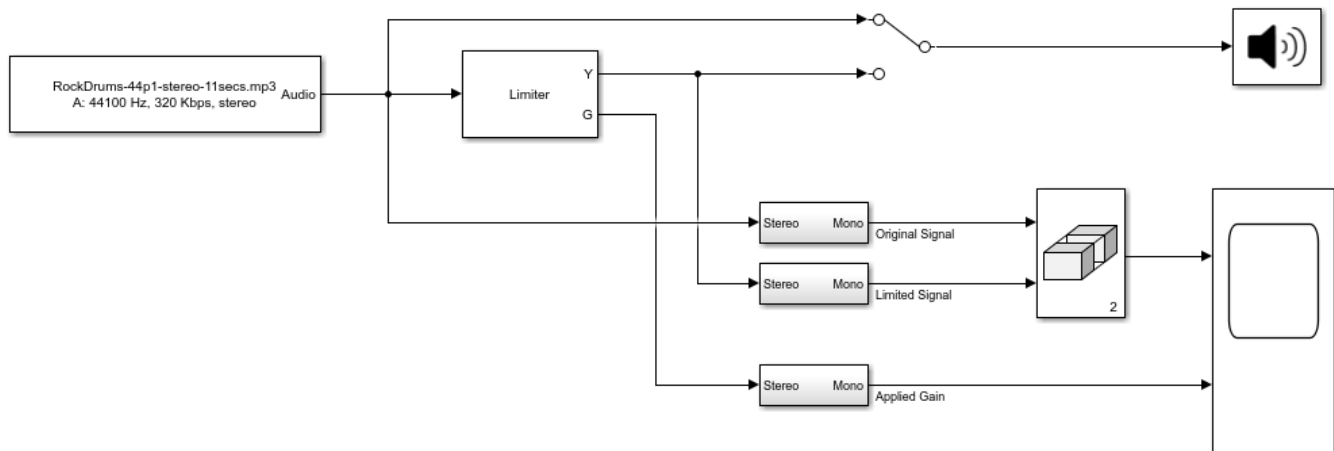
Audio Device Writer | Time Scope | From Multimedia File | Matrix Concatenate | Colored Noise | Expander

More About

- "Dynamic Range Control" on page 8-2

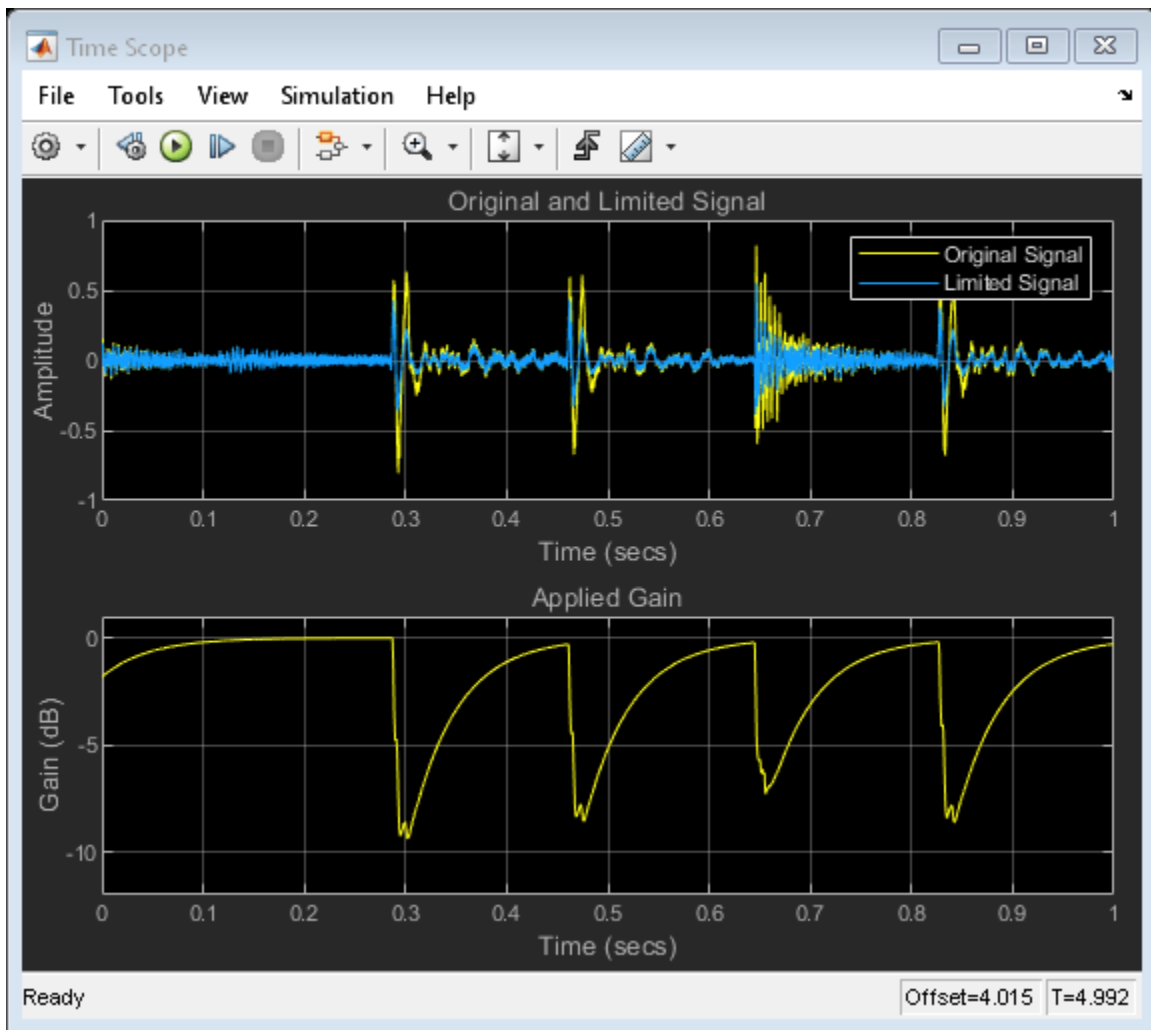
Suppress Volume of Loud Sounds

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

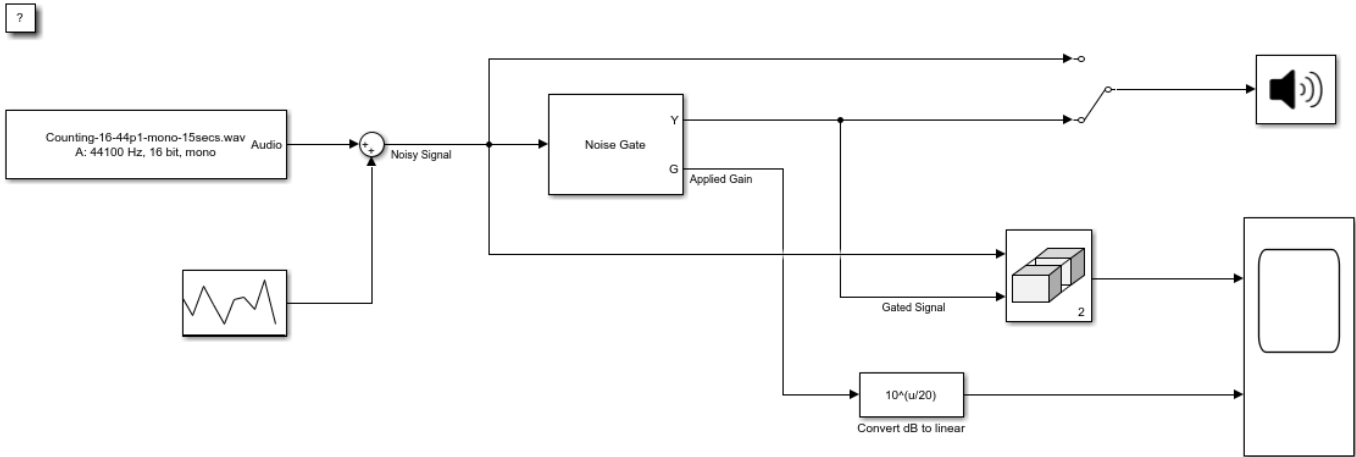
Audio Device Writer | Time Scope | From Multimedia File | Matrix Concatenate | Limiter

More About

- "Dynamic Range Control" on page 8-2

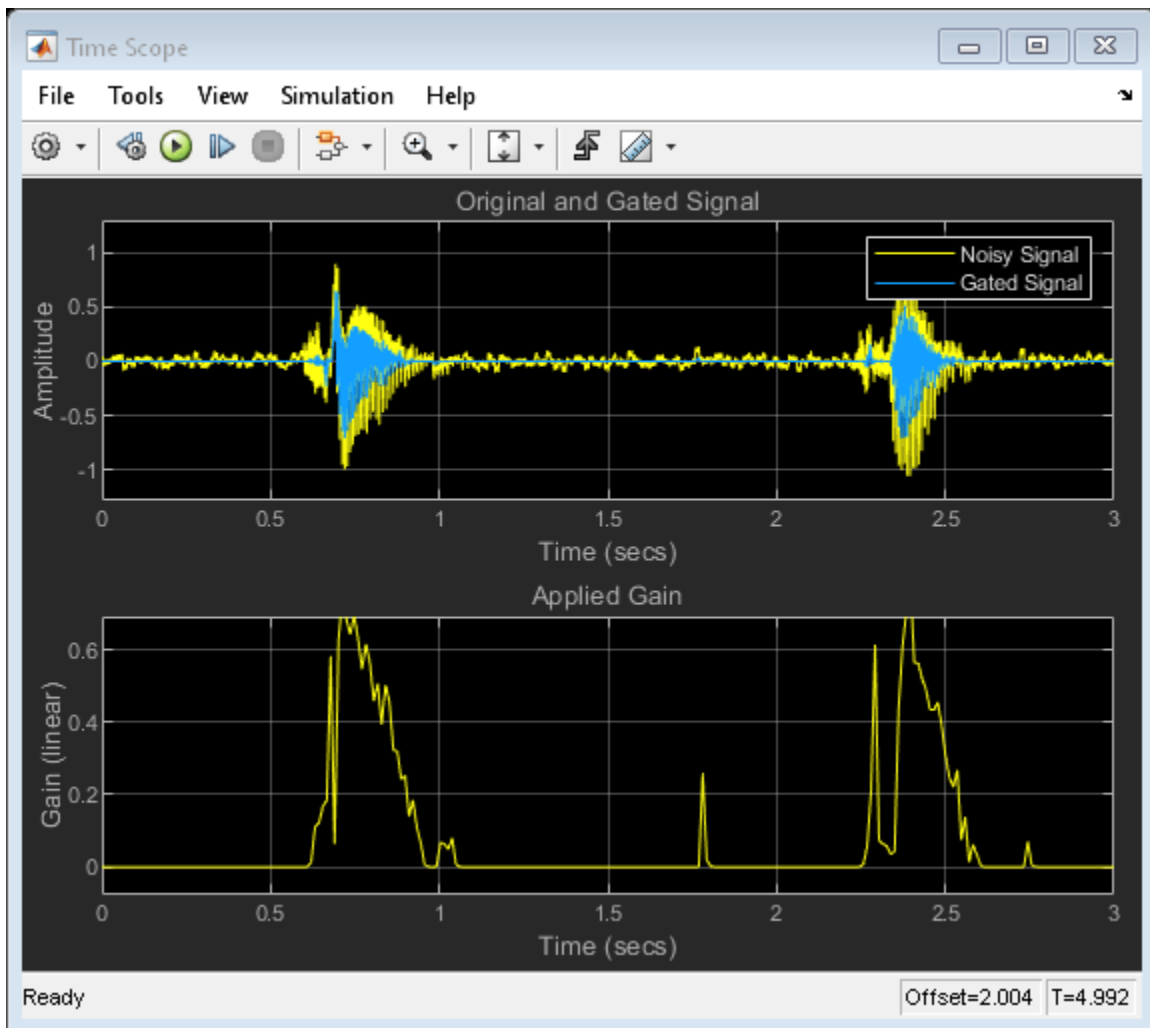
Gate Background Noise

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 | Time Scope | From Multimedia File | Matrix Concatenate | Random Source | Noise Gate

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 `midid` 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 `midid`.
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.

The screenshot shows the parameter configuration window for the MIDI Controls block. The parameters are as follows:

- MIDI device:** Specify other (dropdown menu)
- MIDI device name:** 'BCF2000' (text input field)
- MIDI controls:** Respond to specified controls (dropdown menu)
- MIDI control numbers:** [1081,1083] (text input field)
- Initial values:** [0.1,0.9] (text input field)
- Send initial values to device at start (checkbox)
- Output mode:** Normalized (0 - 1) (dropdown menu)

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

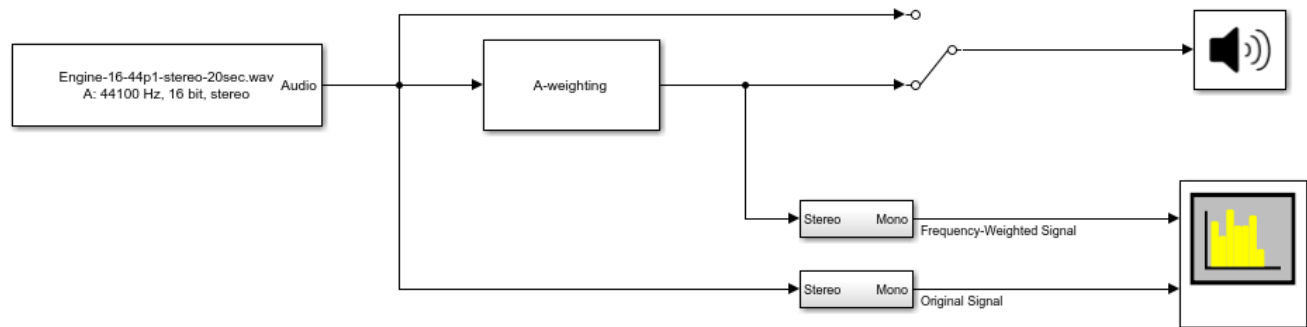
Audio Device Writer | Time Scope | From Multimedia File | MIDI Controls

More About

- “MIDI Control Surface Interface” on page 10-2

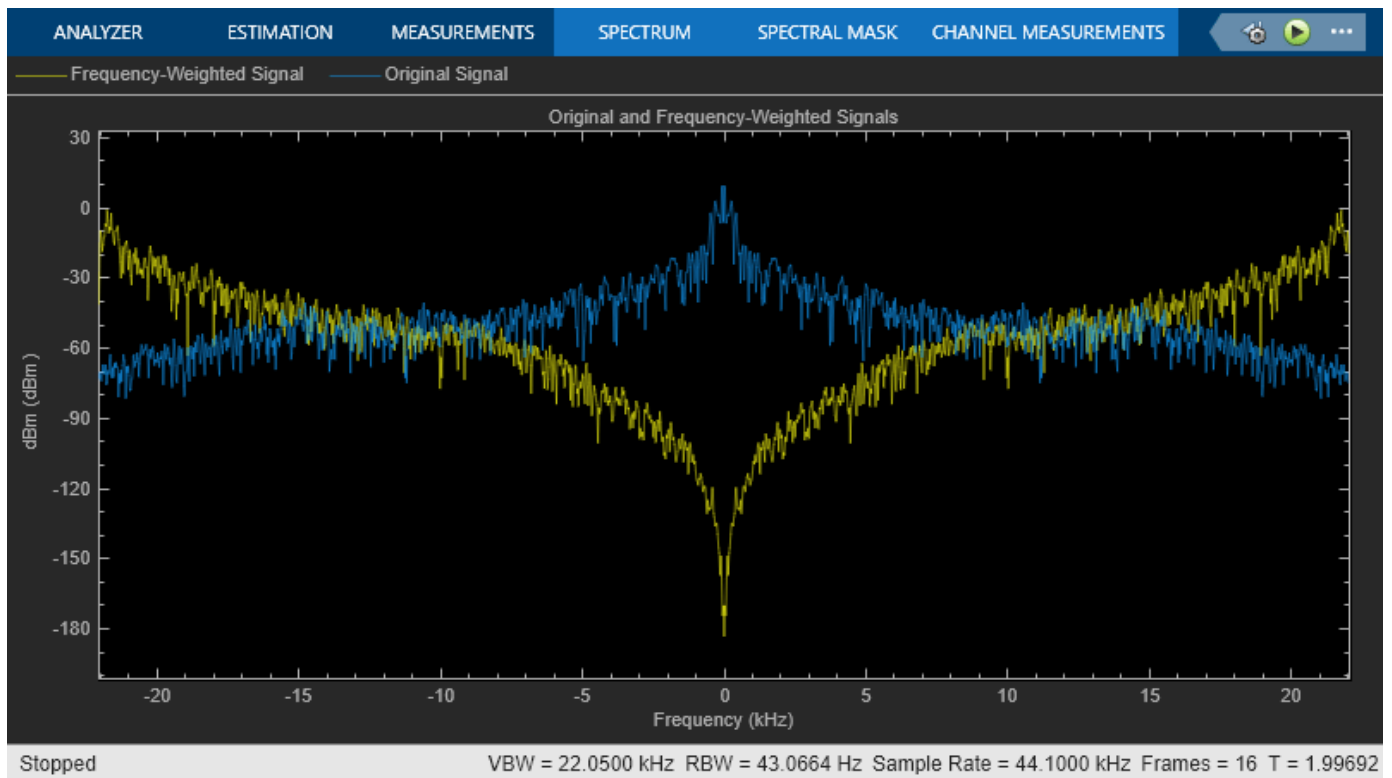
Apply Frequency Weighting

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.

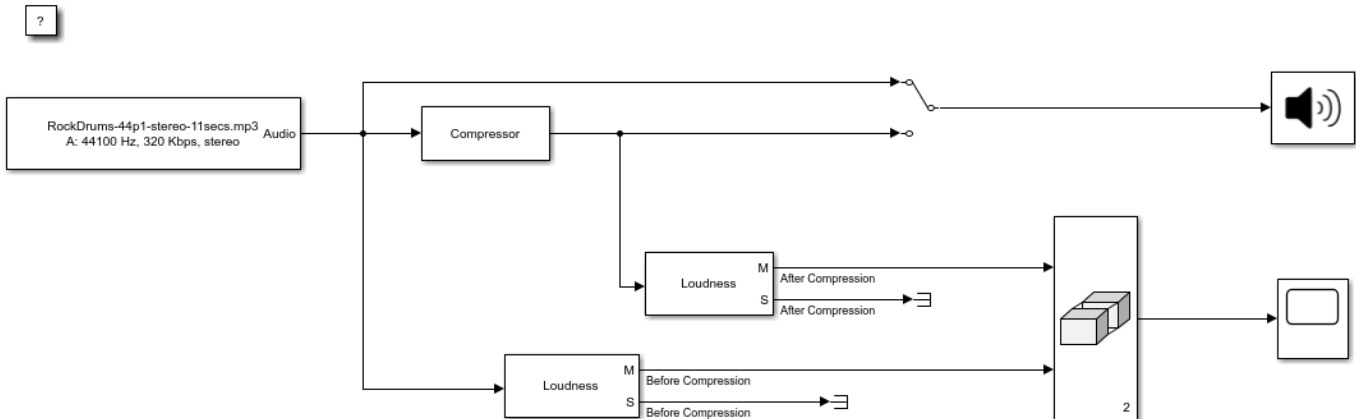


See Also

[Audio Device Writer](#) | [Spectrum Analyzer](#) | [From Multimedia File](#) | [Weighting Filter](#)

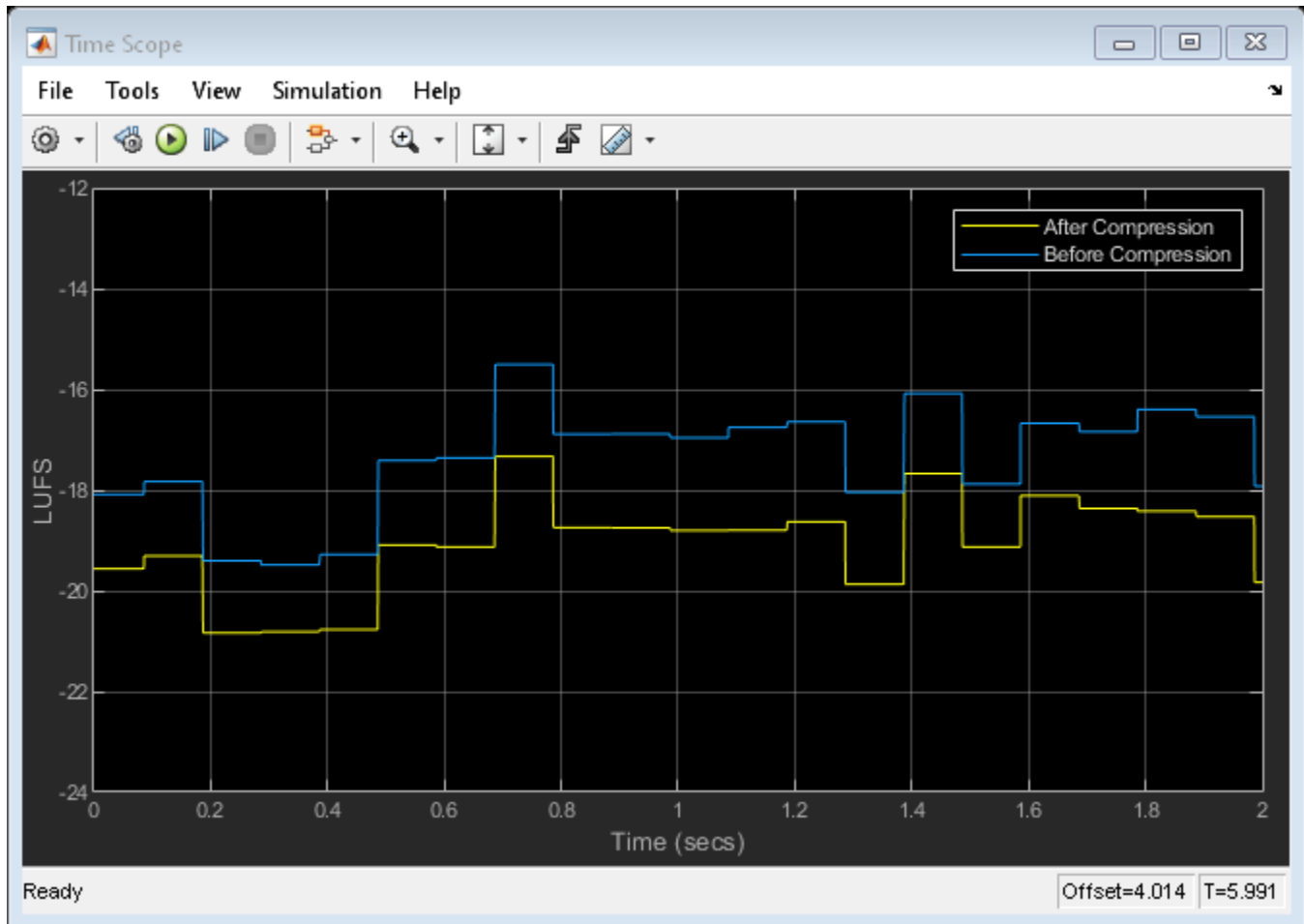
Compare Loudness Before and After Audio Processing

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.



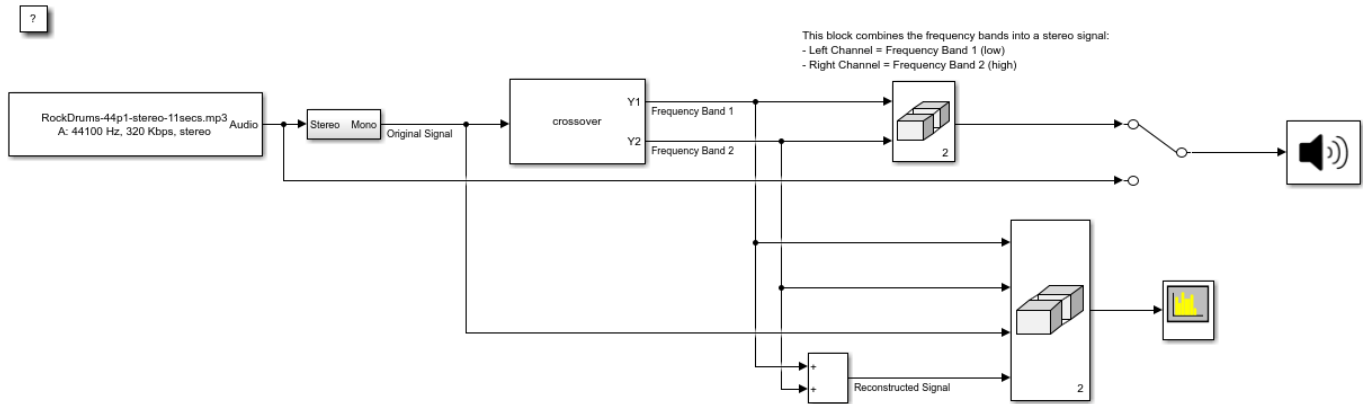
4. Stop the model. For both Loudness blocks, replace momentary loudness with short-term loudness as input to the Matrix Concatenate block. Run the model again and observe the effect of compression on short-term loudness.

See Also

Audio Device Writer | Time Scope | From Multimedia File | Matrix Concatenate | Compressor | Loudness Meter

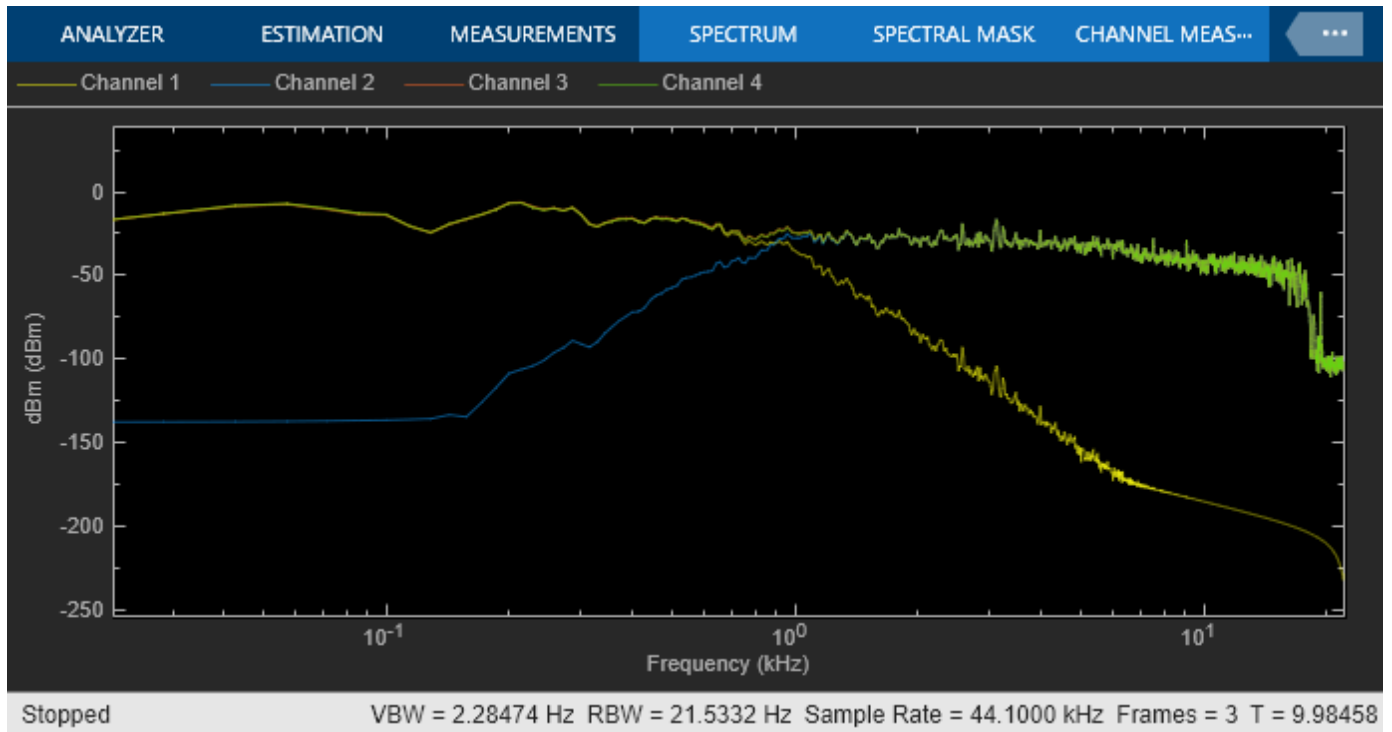
Two-Band Crossover Filtering for a Stereo Speaker System

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.



Copyright 2016 The MathWorks, Inc.

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.

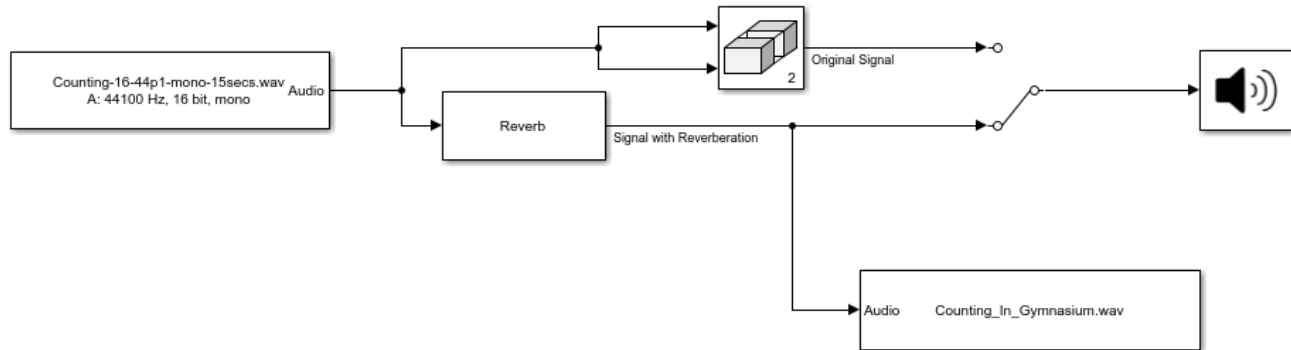


See Also

Audio Device Writer | Spectrum Analyzer | From Multimedia File | Matrix Concatenate | Crossover Filter

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.

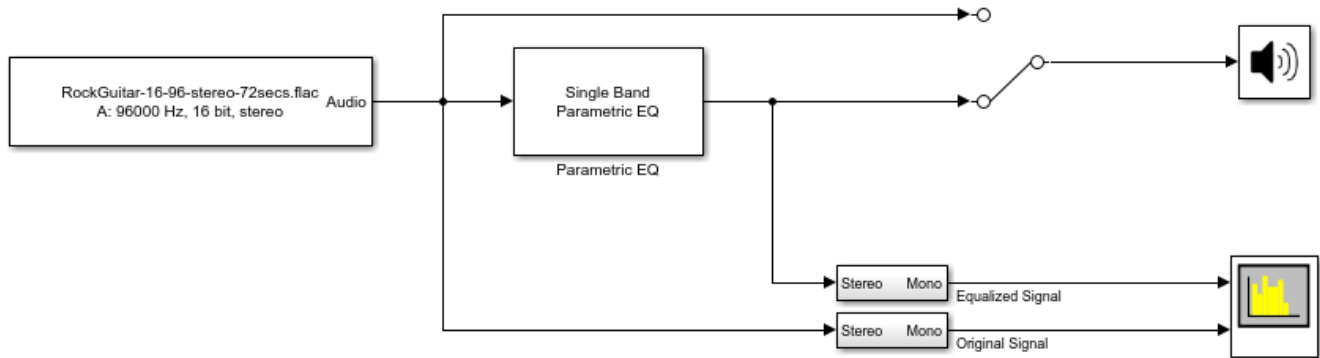
1. Run the simulation. Listen to the audio signal with and without reverberation by double-clicking the Manual Switch block.
2. Stop the simulation.
3. Disconnect the To Multimedia File block so that you can run the model without recording.
4. Open the Reverberator block.
5. Run the simulation and tune 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 | To Multimedia File | From Multimedia File | Matrix Concatenate | Reverberator

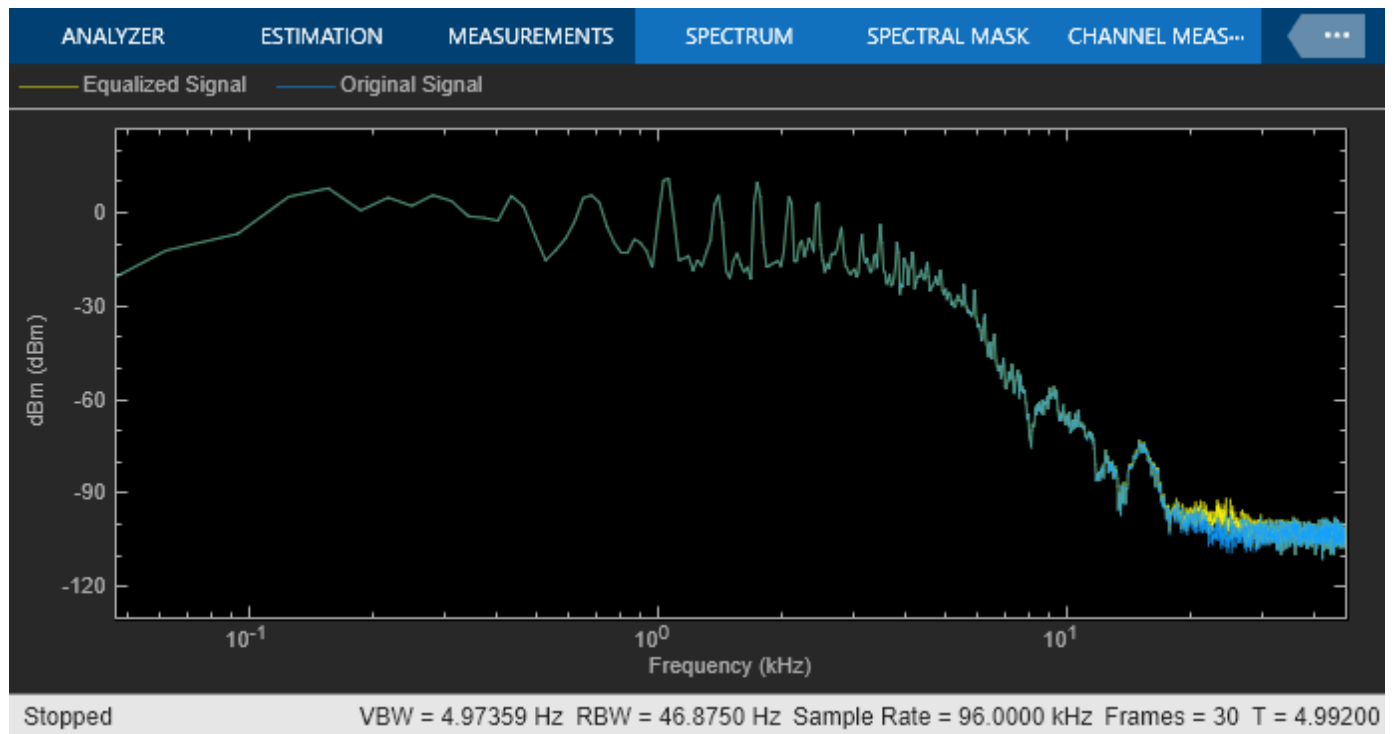
Perform Parametric Equalization

Examine the Single-Band 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 Single-Band 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 Single-Band 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.

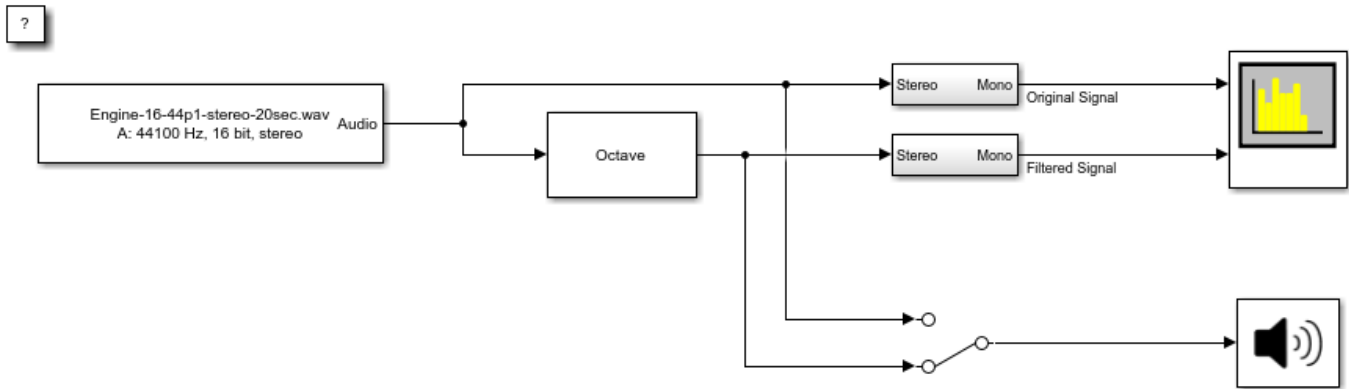


See Also

Audio Device Writer | Spectrum Analyzer | From Multimedia File | Single-Band Parametric EQ

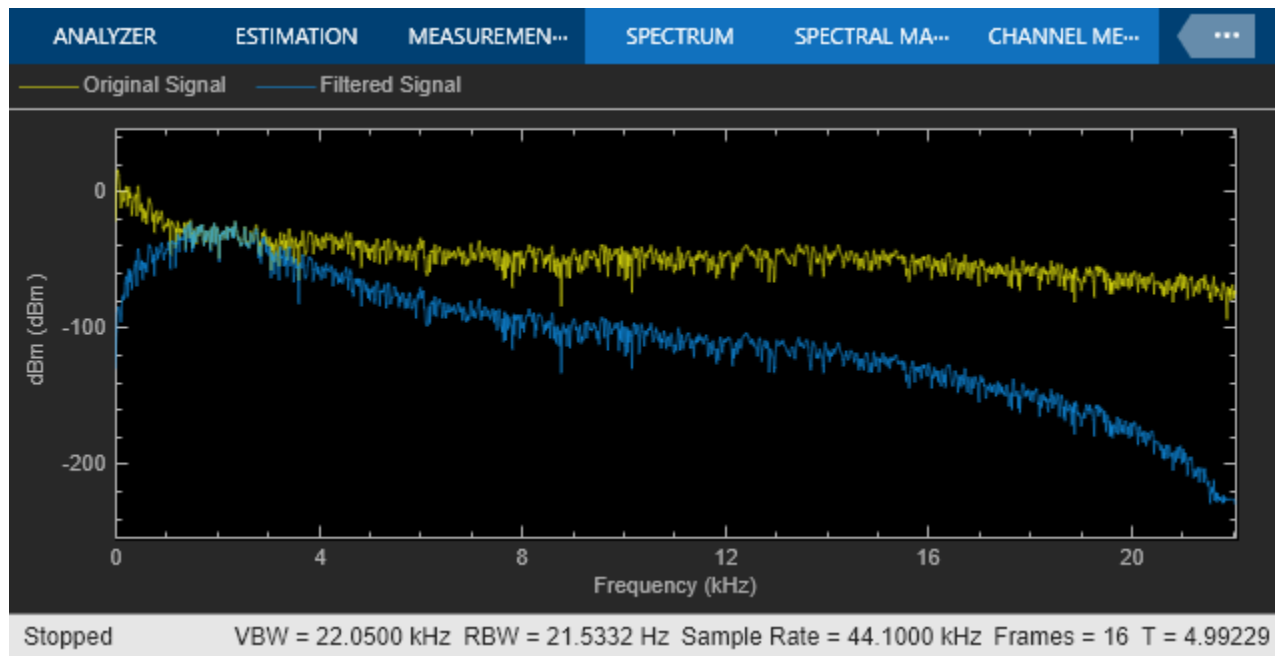
Perform Octave Filtering

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.

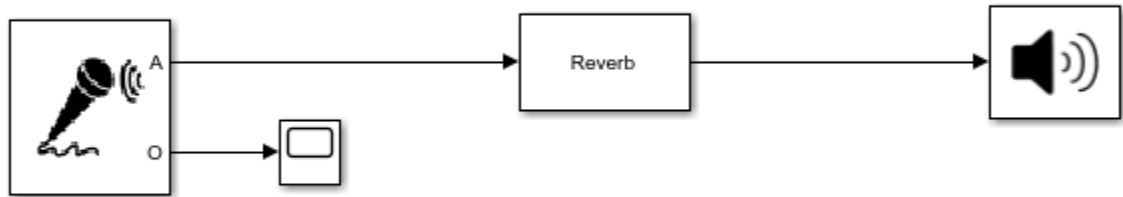


See Also

Audio Device Writer | Spectrum Analyzer | From Multimedia File | Octave Filter

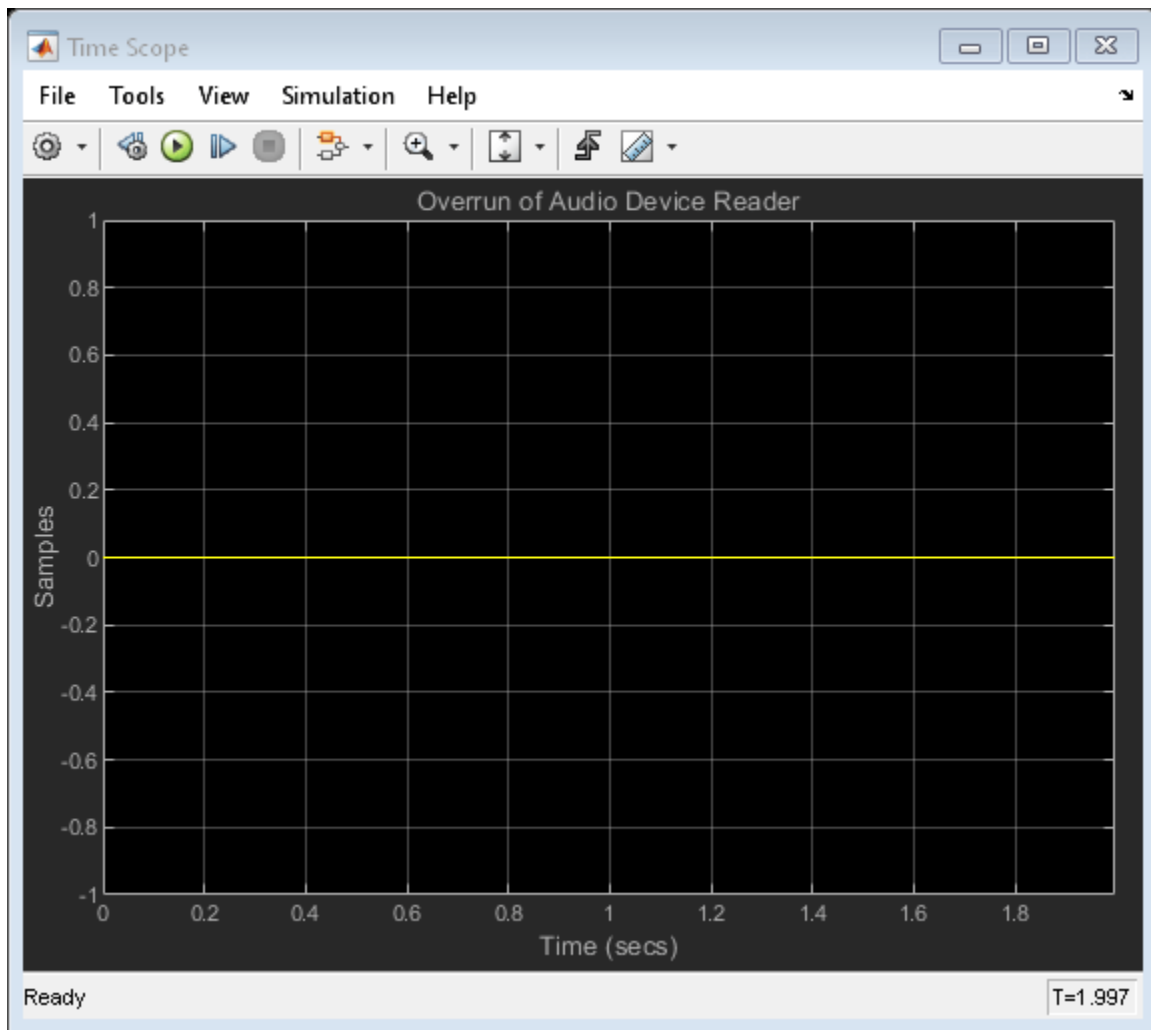
Read from Microphone and Write to Speaker

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

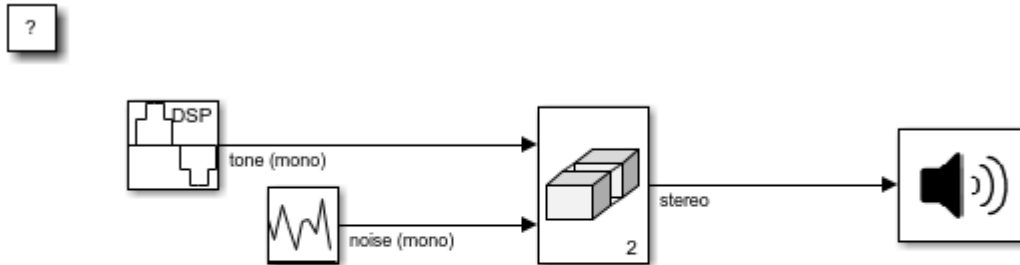
Audio Device Writer | Audio Device Reader | Time Scope | Reverberator

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

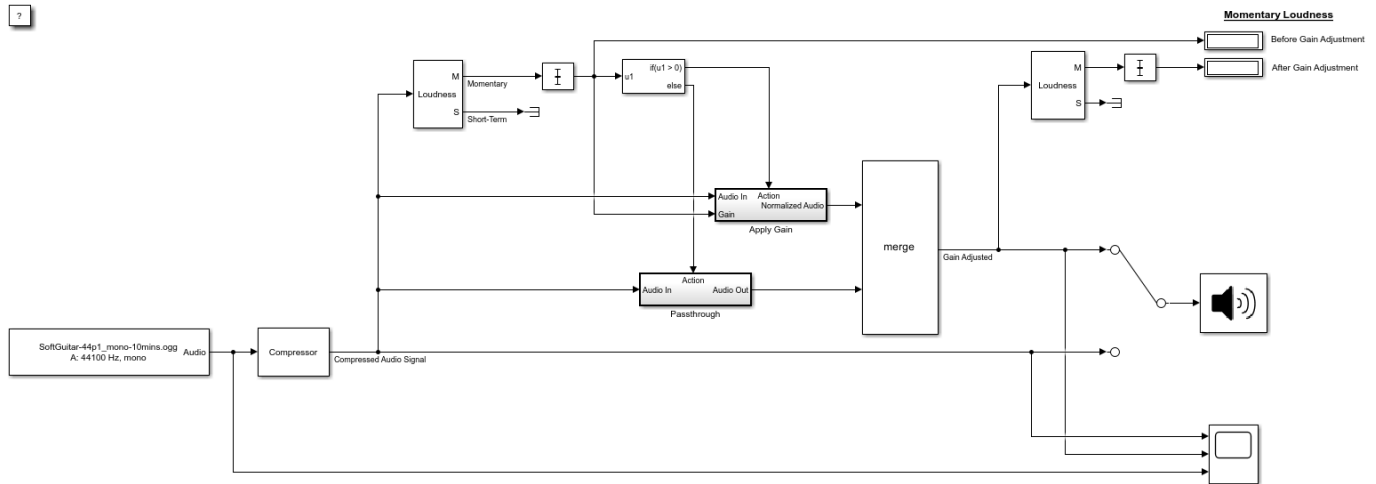
Audio Device Writer | Random Source | Sine Wave | Matrix Concatenate

More About

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

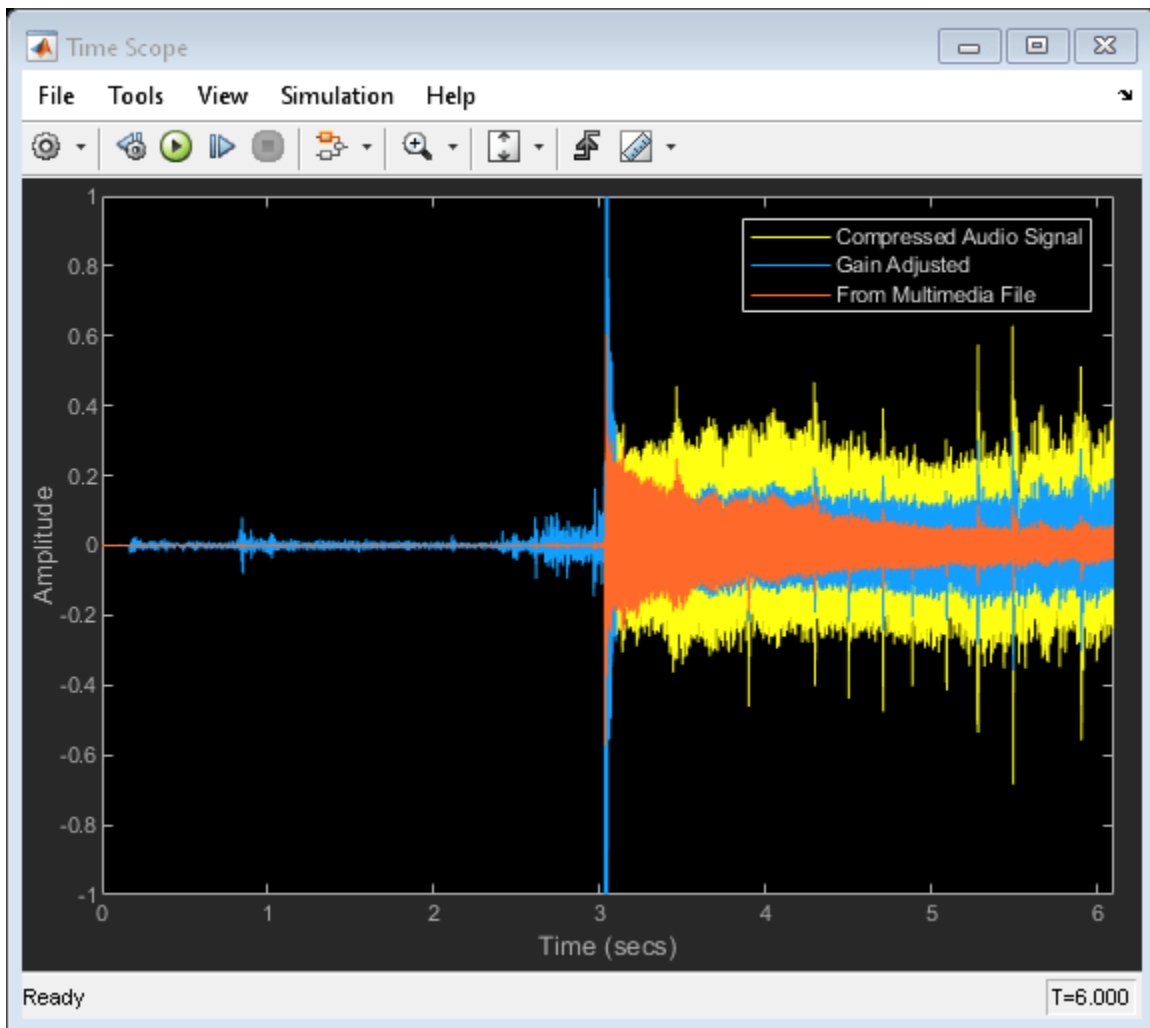
Trigger Gain Control Based on Loudness Measurement

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 | Time Scope | From Multimedia File | Compressor | Loudness Meter

Objects

loudnessMeter

Functions

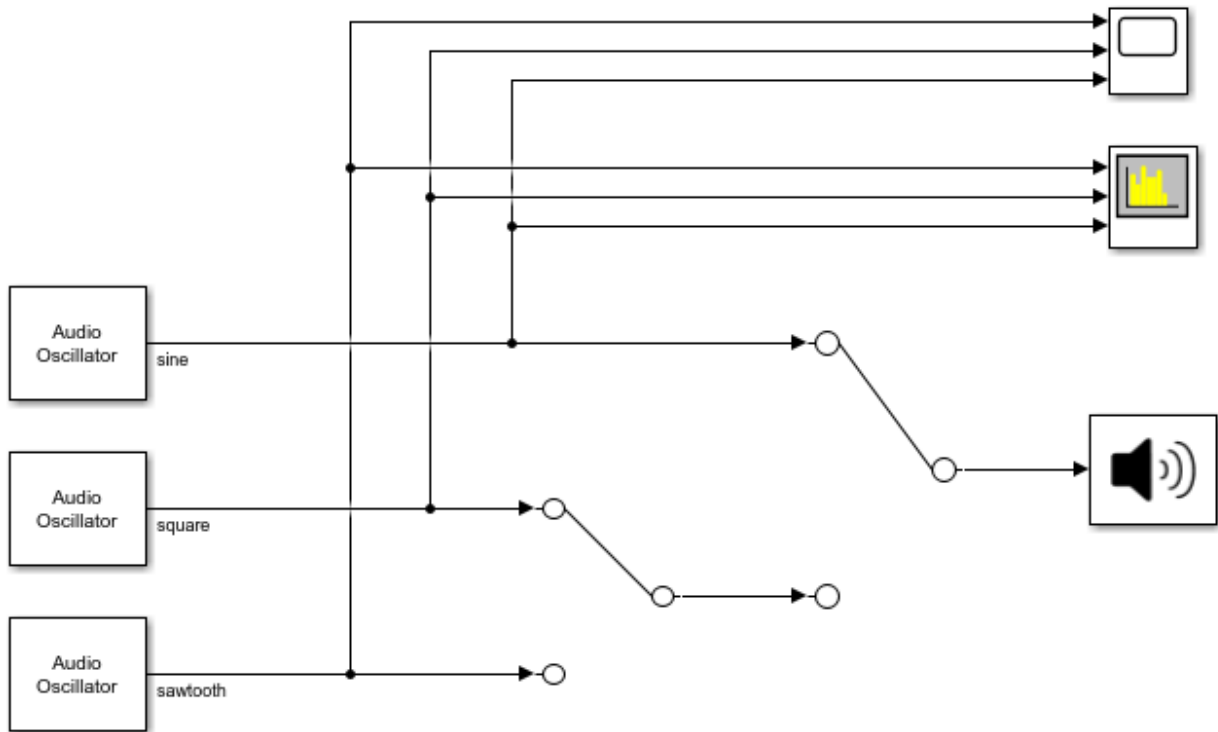
integratedLoudness

More About

- “Loudness Normalization in Accordance with EBU R 128 Standard” on page 1-178

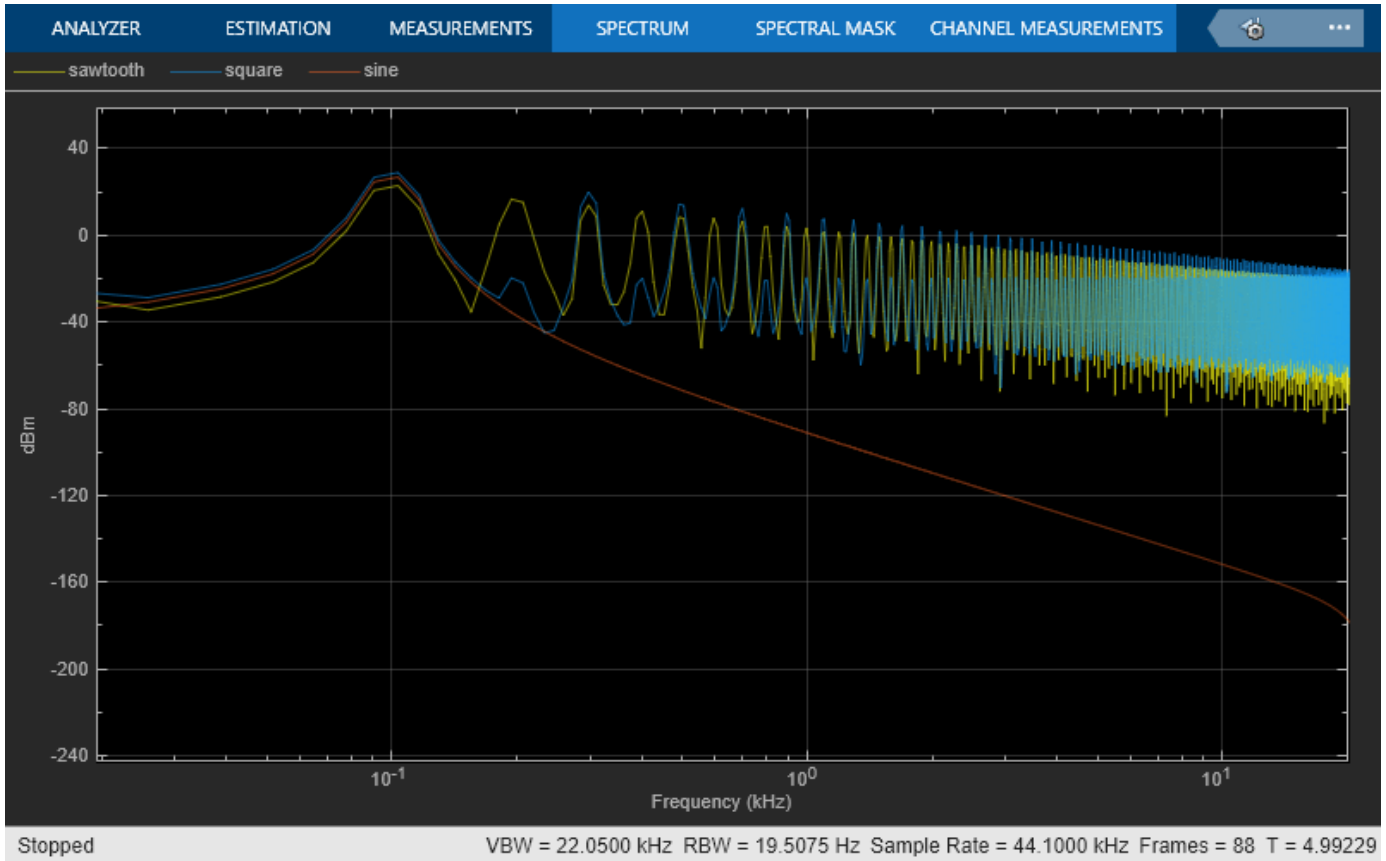
Generate Variable-Frequency Tones in Simulink

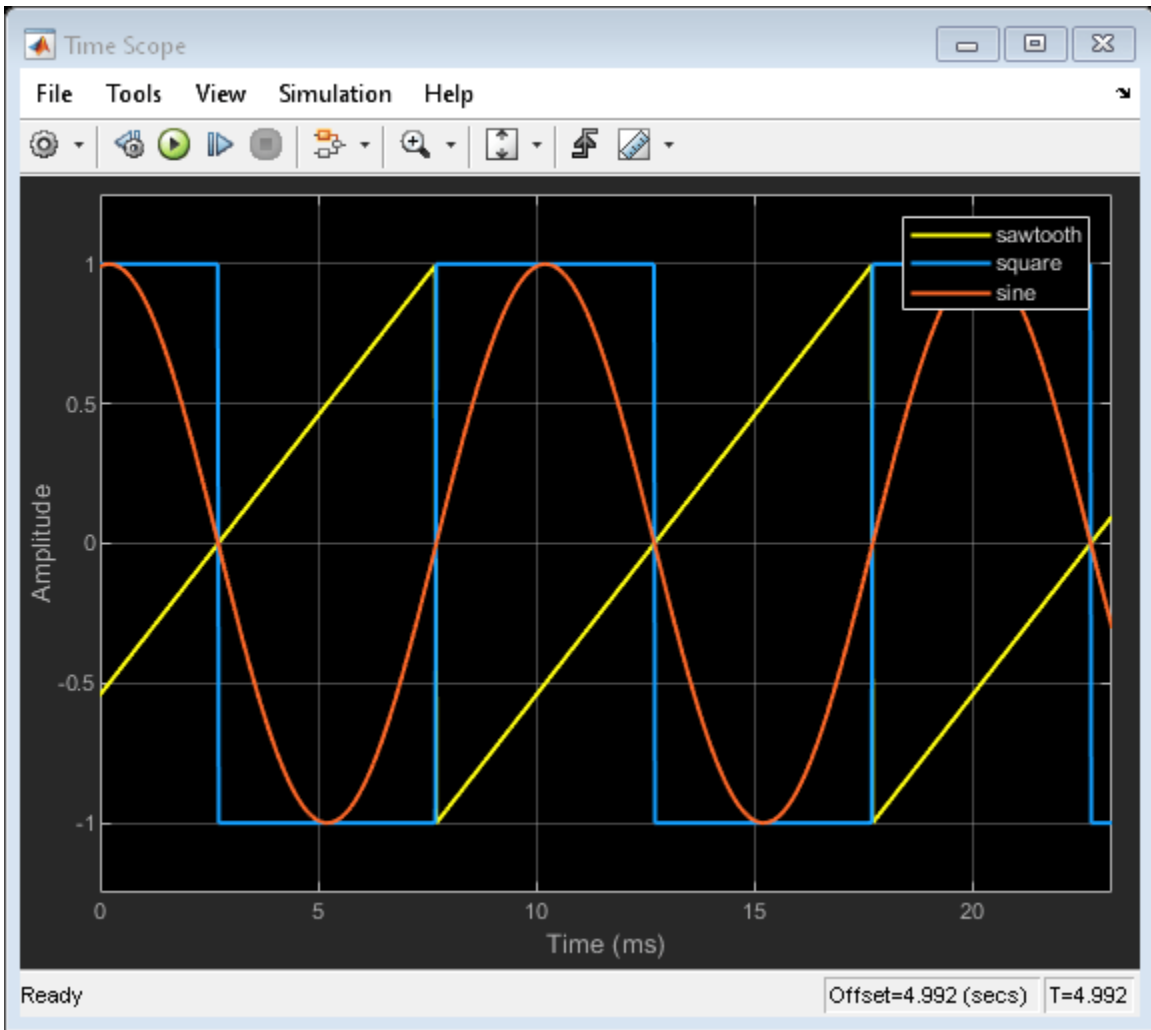
Examine the Audio Oscillator block in a Simulink® model and tune the parameters.



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.

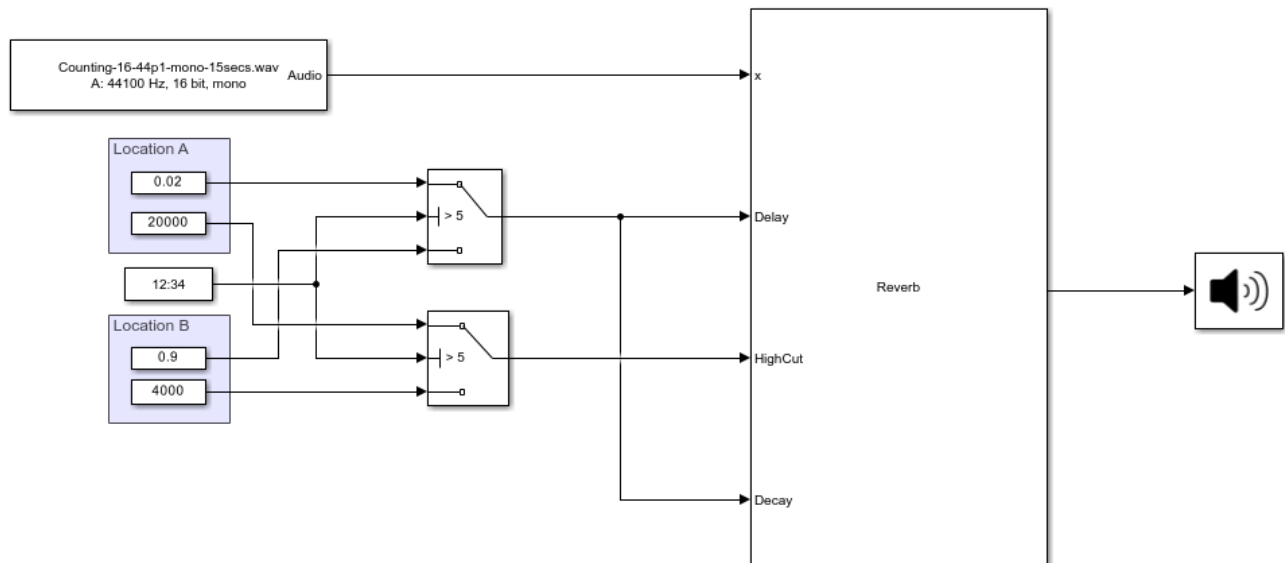




Trigger Reverberation Parameters

Examine the Reverberator block in a Simulink® model where the reverberation parameters are triggered by time.

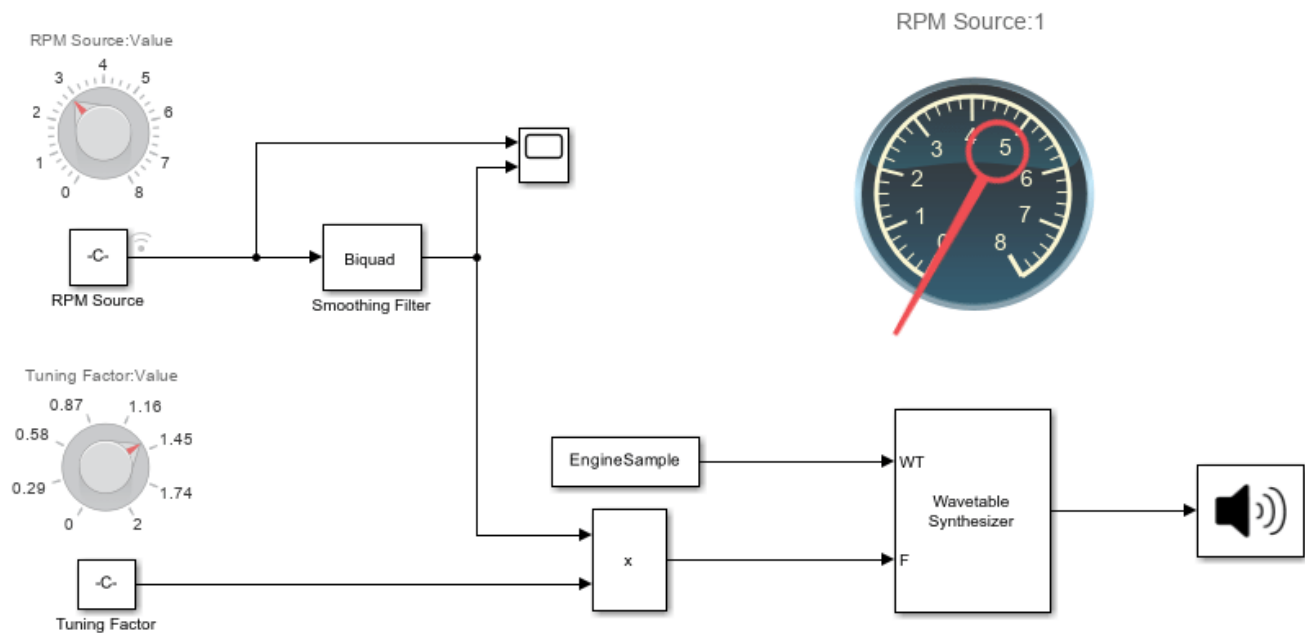
Run the simulation. Listen to the audio signal with the reverberation parameters set to Location A. After 5 seconds, the switches change to the reverberation parameters of Location B.



Copyright 2019 The MathWorks, Inc.

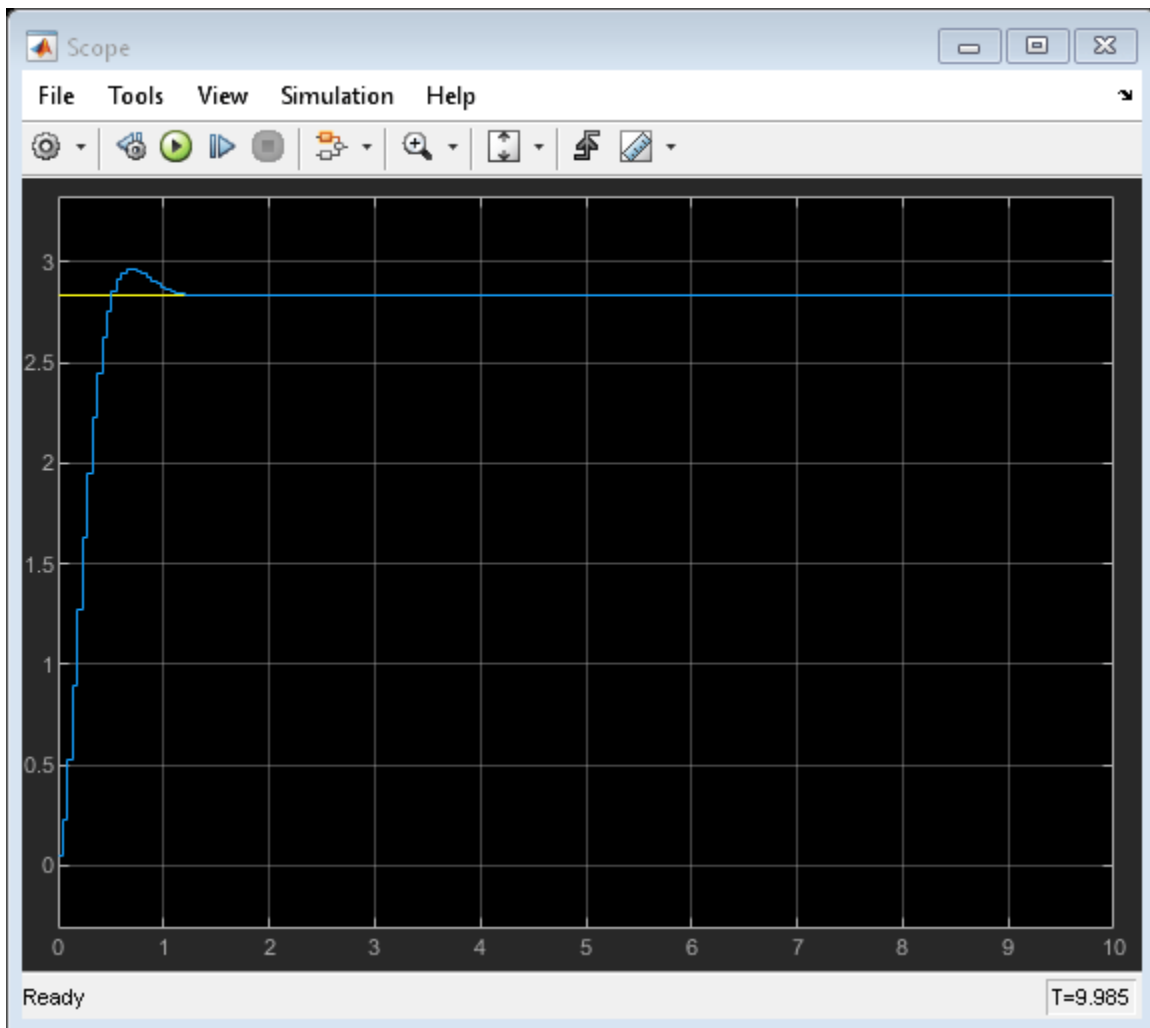
Model Engine Noise

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.

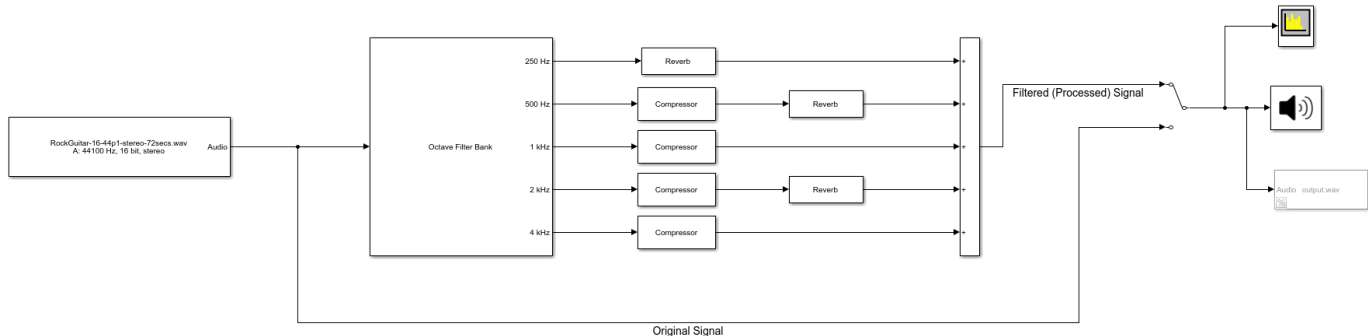
1. Run the simulation. Listen to the engine sound output from the Wavetable Synthesizer.
2. Tune the RPM source to adjust the perceived RPM of the generated engine sound. The RPM source is lowpass smoothed using a Biquad filter, so that the engine sound ramps in a realistic fashion. Visualize the RPM 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.

Use Octave Filter Bank to Create Flanging Chorus Effect for Guitar Layers (Overdubs)

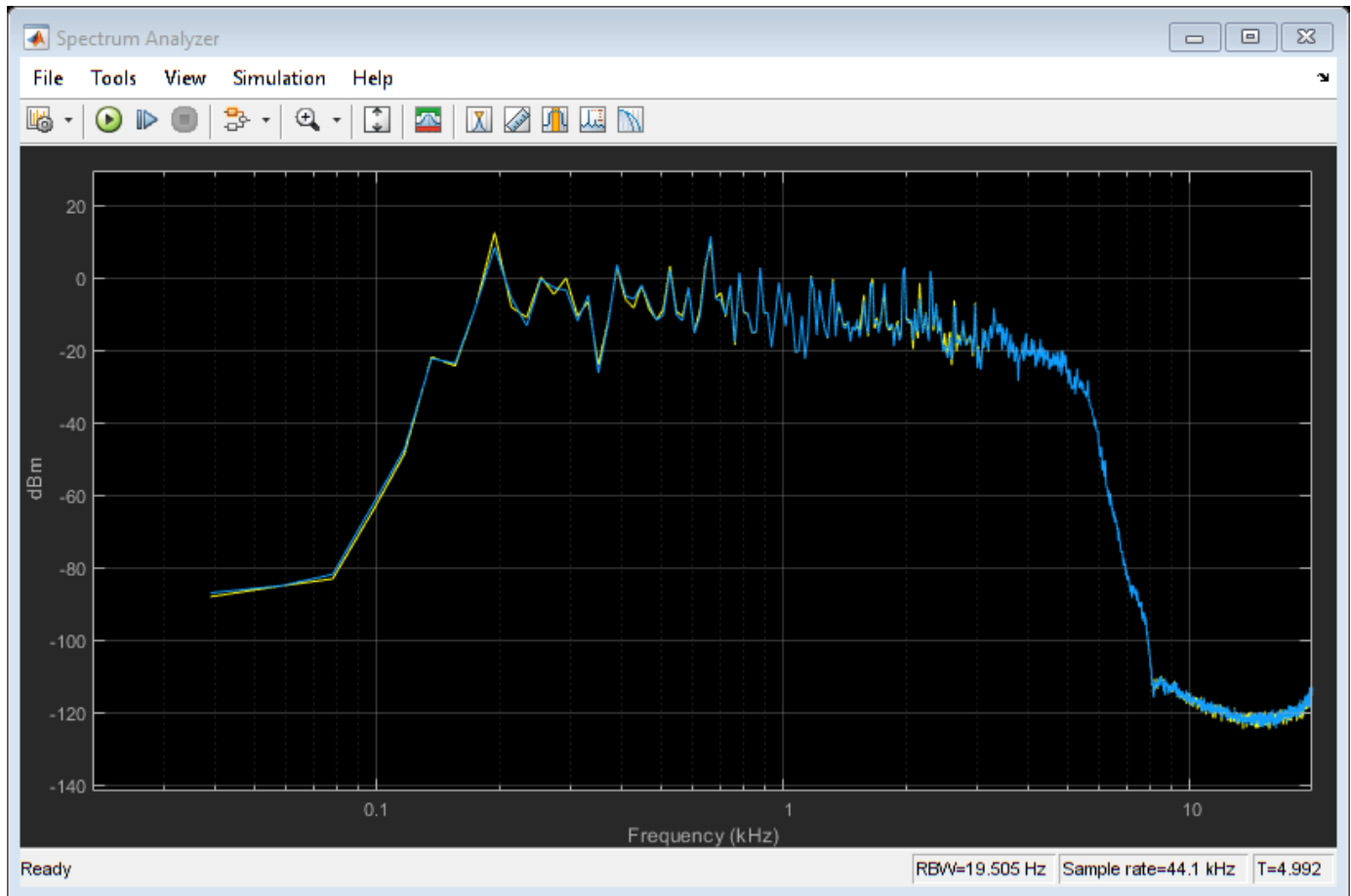
Examine the Octave Filter Bank block in a Simulink® model. Apply octave band compression and reverb to create a flanging chorus effect on the guitar signal. Use the processed signal as an overdub layer to enhance your guitar recordings.



Copyright 2021 The MathWorks Inc.

Using the Octave Filter Bank block allows you to separate the audio signal into multiple frequency bands and process each band individually. In this model example, you split a guitar recording into 5 octave bands and apply compression and reverb to each band separately to create a flanging chorus overdub layer for your recording project.

1. Double-click the Octave Filter Bank block to view its parameters. Notice the **Bands as separate output ports** box is checked. This creates a direct output on the block for each filter in the bank. The **Octave ratio** is also set to **Base two (musical scale)**. To see the magnitude response of the filters in the bank, click the **View Filter Response** button.
2. Run the model.
3. Tune parameters on the Reverberator and Compressor blocks to hear the effects on your audio device and see the effect on the Spectrum Analyzer display. Switch between listening to the **Original Signal** and the **Filtered (Processed) Signal** by double-clicking the Manual Switch (Simulink) block.
4. This Simulink® model can be used to provide overdub guitar layers in your digital audio workstation (DAW) recording projects. Uncomment the **To Multimedia File** block to save your **Filtered (Processed) Signal** audio to a file. In your DAW session, pan the **Original Signal** to the left side of the stereo-field and pan the **Filtered (Processed) Signal** to the right side of the stereo-field. This creates a wide, lush stereo image and adds depth and warmth to your guitar track.



Decompose Signal using Gammatone Filter Bank Block

Use the Gammatone Filter Bank block to decompose a signal by passing it through a bank of gammatone filters.

Connect the blocks as shown in the model.

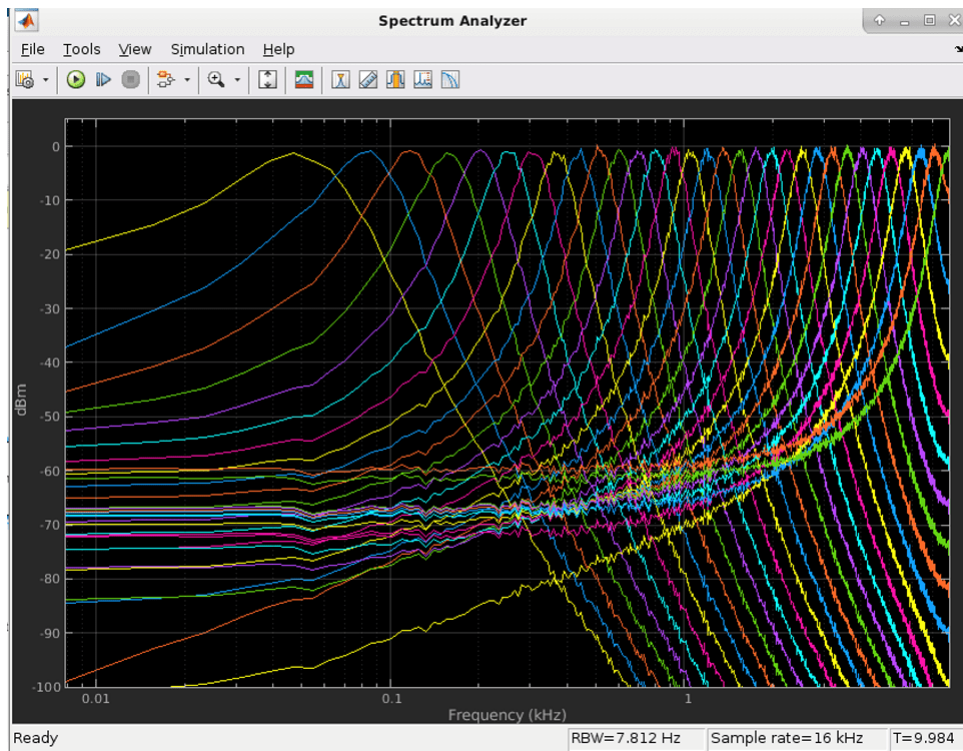


Copyright 2021 The MathWorks, Inc.

Use the Random Source block to generate the signal and observe the output of the Gammatone Filter Bank block using the Spectrum Analyzer Block. Configure the Gammatone Filter Bank by setting the block parameter as:

- **Frequency range (Hz)** — [50 8000]
- **Number of filters** — 32
- **Inherit sample rate from input** — off
- **Input sample rate (Hz)** — 16000
- **Bands as separate output ports** — off

Run the model and select the Spectrum Analyzer Block to view the output of the Gammatone Filter Bank block.



See Also

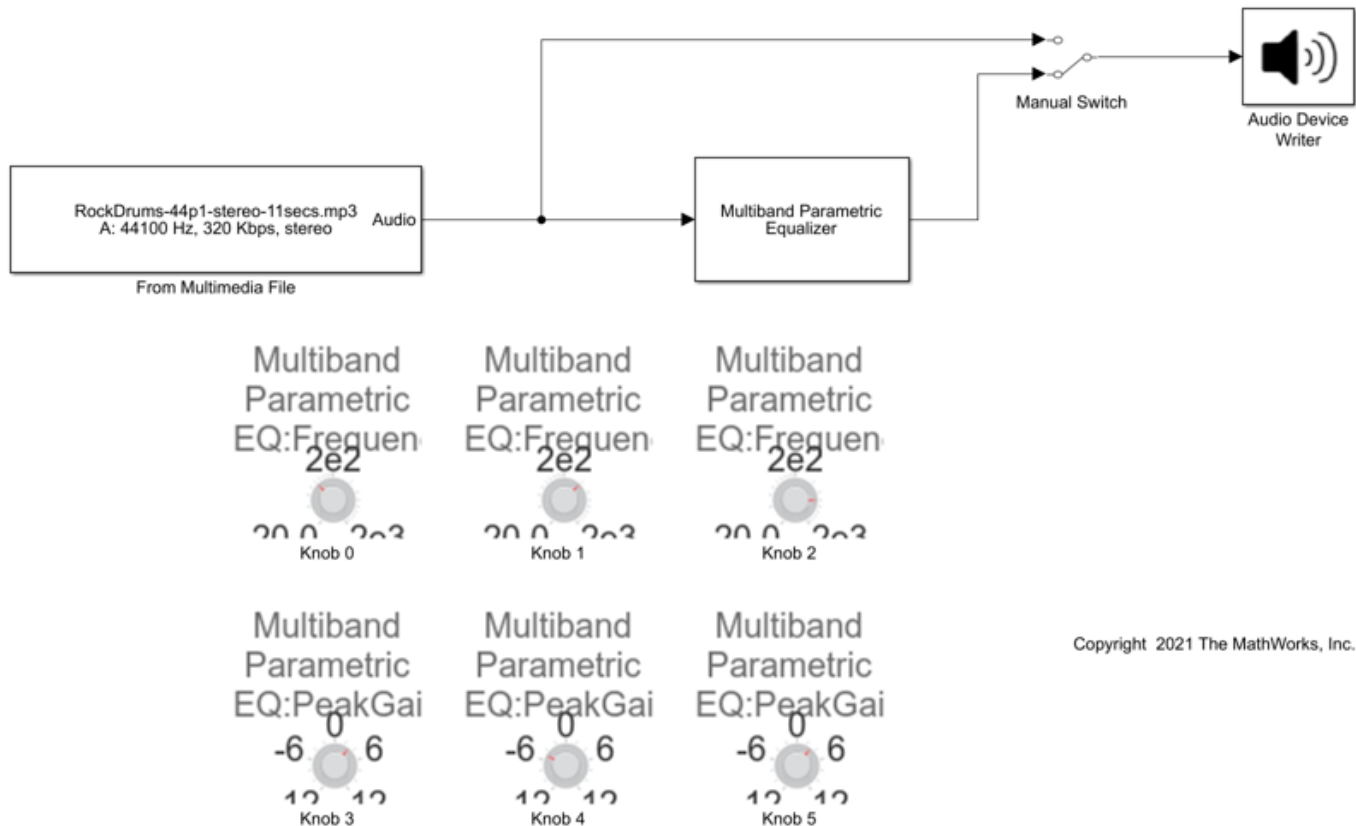
Related Examples

- “Visualize Filter Response of Multiband Parametric Equalizer Block” on page 17-54

Visualize Filter Response of Multiband Parametric Equalizer Block

Perform multiband parametric equalization independently across each channel of an input using specified center frequencies, gains, and quality factors.

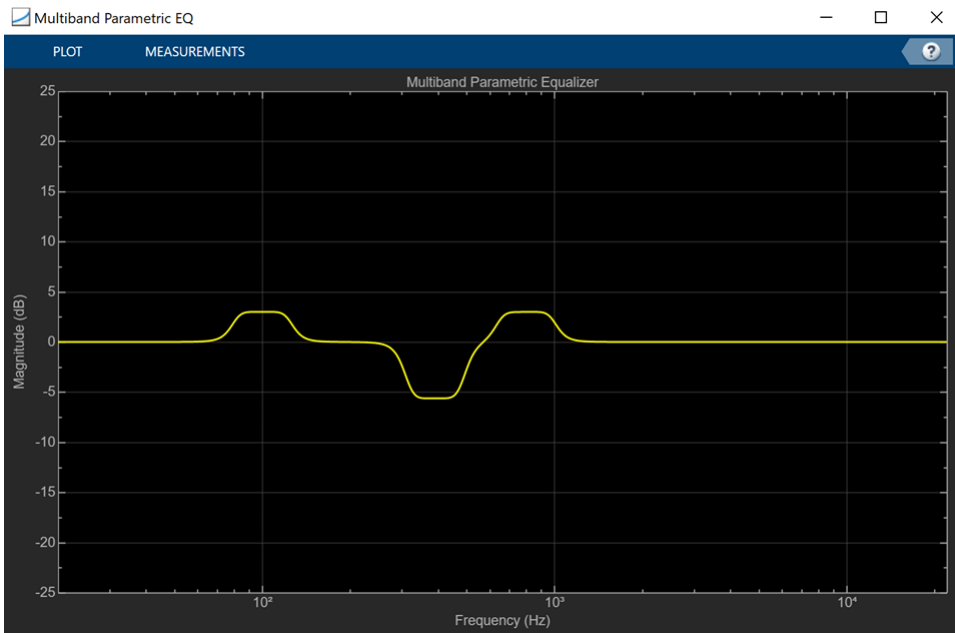
Connect the Multiband Parametric EQ block to an audio input as shown in this model.



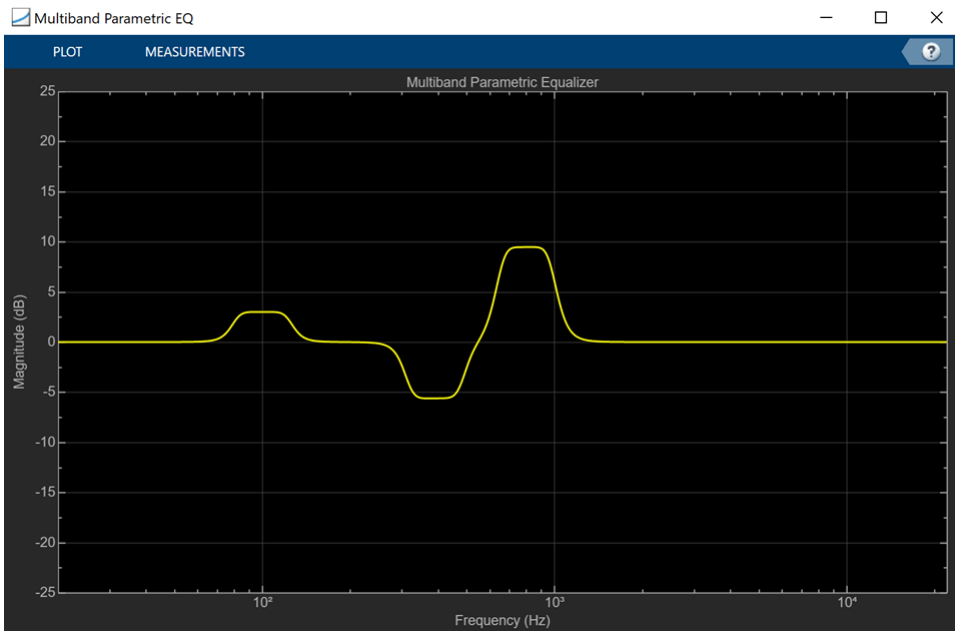
Configure the Multiband Parametric Equalizer block by setting its parameters as:

- **EQ order** — 6
- **Number of bands** — 3
- **Frequencies (Hz)** — [100 390 800]
- **Peak gains (dB)** — [3 -5 3]
- **Quality factors** — [2 2 2]
- **Input sample rate (Hz)** — 44100

Run the model and click the **Visualize filter response** button to plot the filter response in magnitude (dB) vs. frequency (Hz).



Use the knobs to change frequency and gain and observe the changing response. For instance, change Knob 5 to set the peak gain of the third frequency to 9 dB and observe the filter response.



You can also toggle the switch to listen to either the original or the filtered signal.

See Also

Multiband Parametric EQ

Related Examples

- “Decompose Signal using Gammatone Filter Bank Block” on page 17-52

Detect Music in Simulink Using YAMNet

The YAMNet network requires you to preprocess and extract features from audio signals by converting them to the sample rate the network was trained on (16e3 Hz), and then extracting overlapping mel spectrograms. The Sound Classifier block does the required preprocessing and feature extraction that is necessary to match the preprocessing and feature extraction used to train YAMNet.

To use YAMNet, a pretrained YAMNet network must be installed in a location on the MATLAB® path. If a pretrained network is not installed, run the `yamnetGraph` function and the software provides a download link. Click the link and unzip the file to a location on the MATLAB path.

Alternatively, execute the following commands to download and unzip the YAMNet model to your temporary directory.

```
downloadFolder = fullfile(tempdir, 'YAMNetDownload');
loc = websave(downloadFolder, 'https://ssd.mathworks.com/supportfiles/audio/yamnet.zip');
YAMNetLocation = tempdir;
unzip(loc, YAMNetLocation)
addpath(fullfile(YAMNetLocation, 'yamnet'))
```

Get all music sounds in the AudioSet ontology. The ontology covers a wide range of everyday sounds, from human and animal sounds to natural and environmental sounds and to musical and miscellaneous sounds. Use the `yamnetGraph` function to obtain a graph of the AudioSet ontology and a list of all sounds supported by YAMNet. The `dfsearch` function returns a vector of 'Music' sounds in the order of their discovery using depth-first search.

```
[ygraph, allSounds] = yamnetGraph;
musicSounds = dfsearch(ygraph, "Music");
```

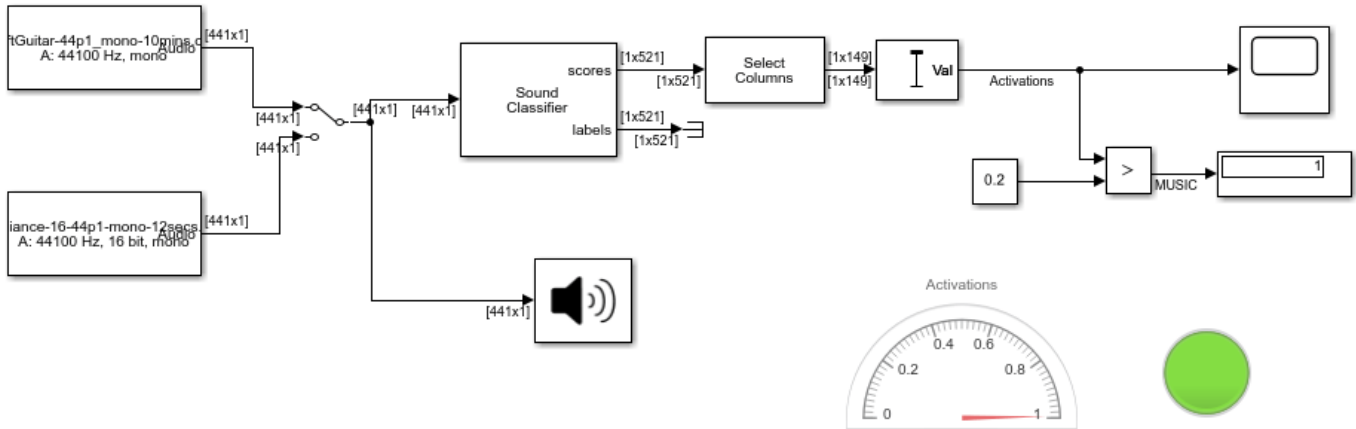
Find the location of these musical sounds in the list of supported sounds.

```
[~, musicIndices] = intersect(allSounds, musicSounds);
```

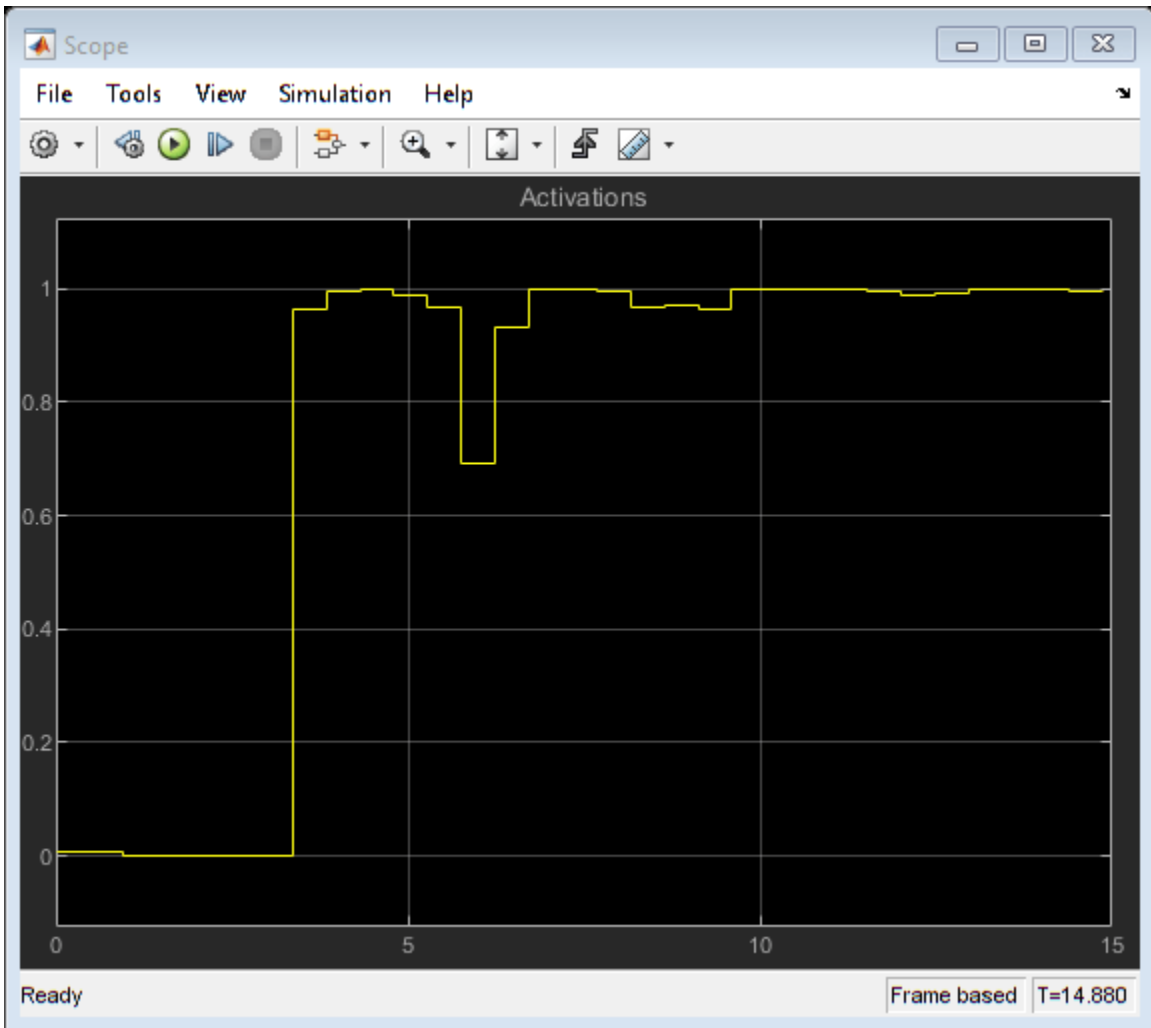
The `detectMusic` model detects the musical sounds in input audio. Open and run the model. The model starts by reading in an audio signal to classify using two From Multimedia File blocks. The first block reads in a musical sound signal and the second block reads in an ambiance signal that is not music. Both signals have a sample rate of 44100 Hz and contain 441 samples per channel. Using the Manual Switch (Simulink) block, you can choose one of the two signals.

The Sound Classifier block in the model detects the scores and labels of the input audio. The Selector (Simulink) block in the model picks the scores related to music using the vector of indices given by `musicIndices`. If the maximum value of these scores is greater than 0.2, then the score is related to music. The Scope (Simulink) block plots the maximum value of the score. The Activation dial in the model shows this value as well. Using the Audio Device Writer block, confirm that you hear music when the plot shows a score greater than 0.2

```
open_system("detectMusic.slx")
sim("detectMusic.slx")
```



Copyright 2021 The MathWorks Inc.



```
close_system("detectMusic.slx",0)
```

See Also

Functions

yamnetGraph | dfsearch

Blocks

Sound Classifier | From Multimedia File | Manual Switch | Selector | Scope | Audio Device Writer

Related Examples

- “Detect Air Compressor Sounds in Simulink Using YAMNet” on page 17-62
- “Compare Sound Classifier block with Equivalent YAMNet blocks” on page 17-60

Compare Sound Classifier block with Equivalent YAMNet blocks

The Sound Classifier block is equivalent to the cascade of the YAMNet Preprocess block and YAMNet block. The model in this example compares the two implementations and shows their equivalence.

The input to the model is a single-channel audio signal. The signal has a sample rate of 44100 Hz and contains 441 samples per channel. The first branch of the model contains the Sound Classifier block. The second branch of the model contains the YAMNet Preprocess block followed by the YAMNet block.

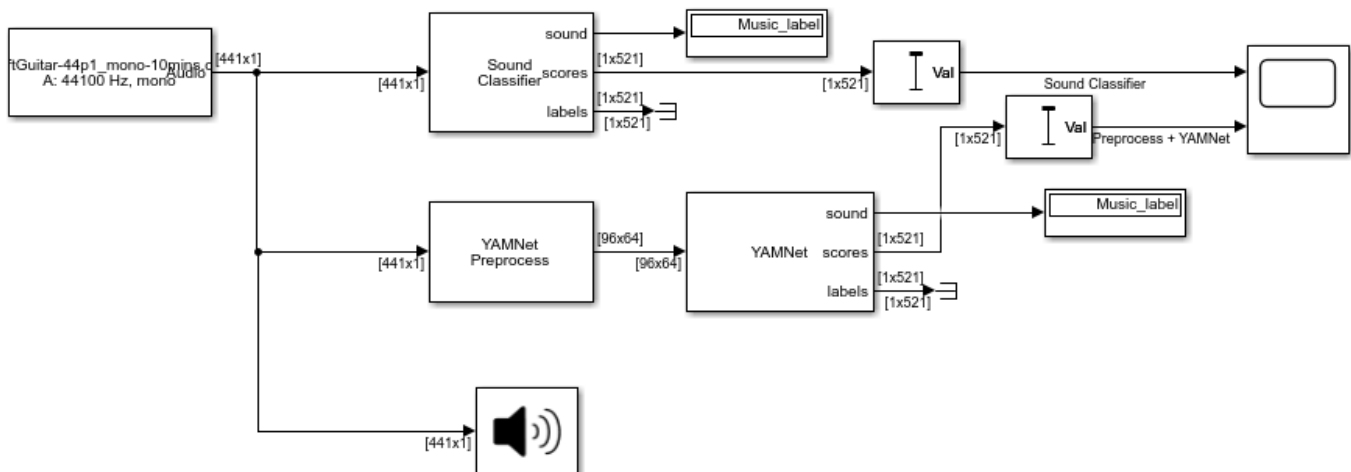
To use these blocks, a YAMNet pretrained network must be installed in a location on the MATLAB® path. If a pretrained network is not installed, then open and run the model. The software provides a download link. To download the network, click the link and unzip the file to a location on the MATLAB path.

Alternatively, execute the following commands to download and unzip the YAMNet model to your temporary directory.

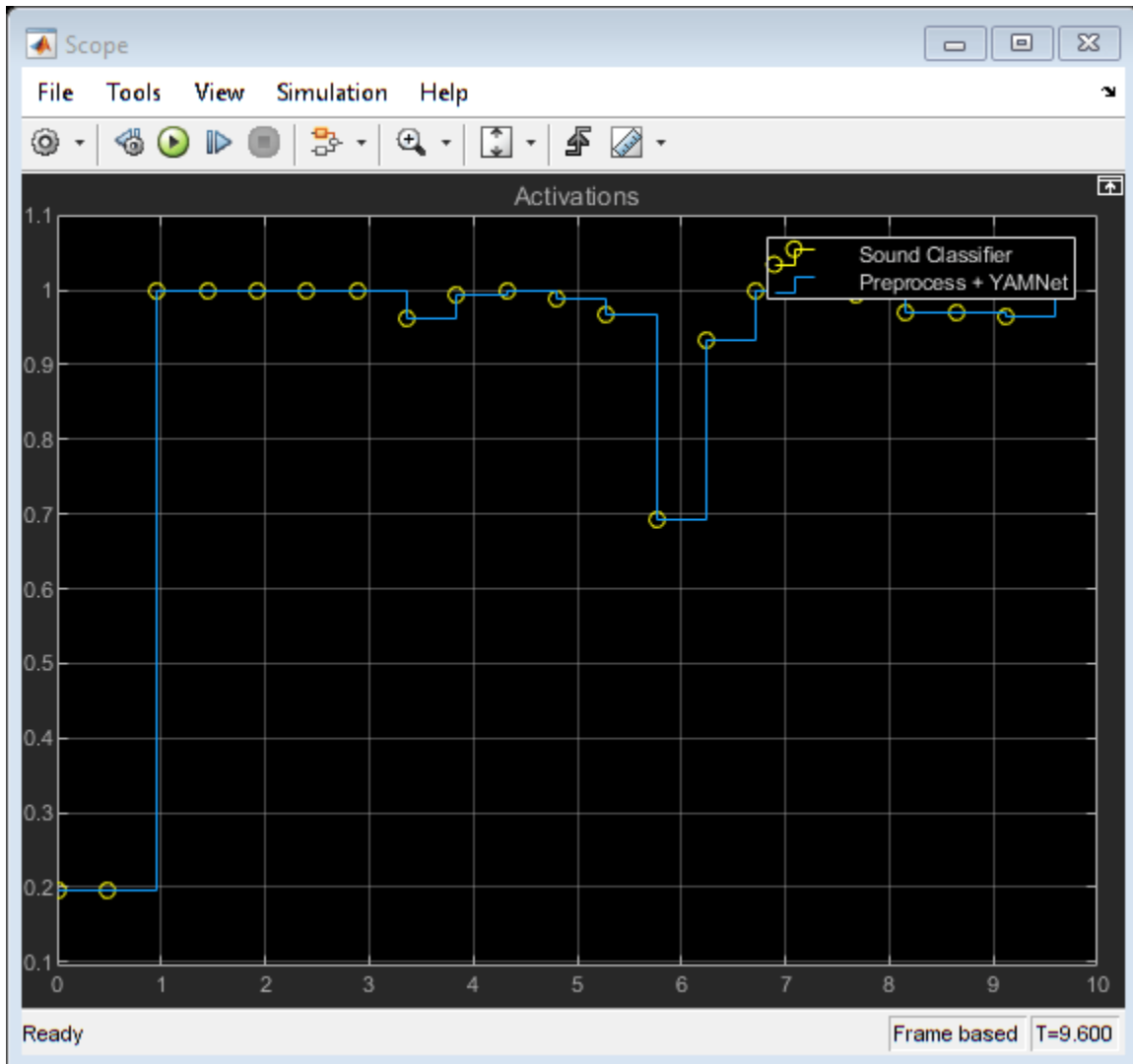
```
downloadFolder = fullfile(tempdir, 'YAMNetDownload');
loc = websave(downloadFolder, 'https://ssd.mathworks.com/supportfiles/audio/yamnet.zip');
YAMNetLocation = tempdir;
unzip(loc, YAMNetLocation)
addpath(fullfile(YAMNetLocation, 'yamnet'))
```

Open and run the model. The Maximum block on each branch computes the maximum value of the vector of music scores predicted on each branch. Plot these maximum values on the Scope block and confirm if they match. Similarly, confirm the equivalence in sound labels shown by the Display blocks.

```
open_system("compareblocks.slx")
sim("compareblocks.slx")
```



Copyright 2021 The MathWorks Inc.



```
close_system("compareblocks.slx",0)
```

See Also

Blocks

Sound Classifier | YAMNet Preprocess | YAMNet | From Multimedia File | Maximum | Scope | Audio Device Writer

Related Examples

- "Detect Music in Simulink Using YAMNet" on page 17-57
- "Detect Air Compressor Sounds in Simulink Using YAMNet" on page 17-62

Detect Air Compressor Sounds in Simulink Using YAMNet

This example shows how to use a pretrained network obtained from transfer learning within a Simulink® model to classify audio signals obtained from an air compressor.

The network is pretrained using a data set that contains recordings from air compressors. The data set is classified into one healthy state and seven faulty states, for a total of eight classes. For more information on training, see “Transfer Learning Using YAMNet”.

To download this pretrained network and a set of air compressor sounds to detect, run the following commands. These commands download and unzip the files to a location on the MATLAB® path. The `airCompressorNet.mat` file stores the pretrained network.

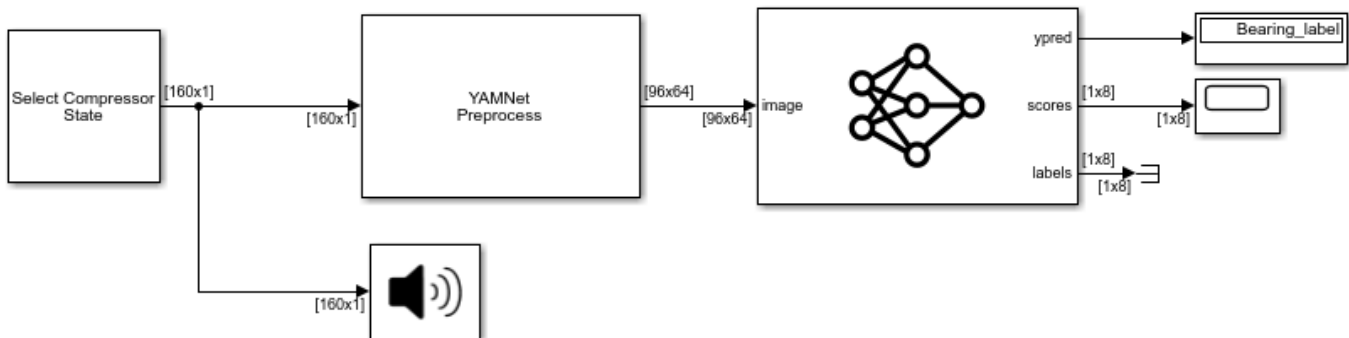
```
url = 'https://ssd.mathworks.com/supportfiles/audio/YAMNetTransferLearning.zip';
AirCompressorLocation = tempdir;
dataFolder = fullfile(AirCompressorLocation, 'YAMNetTransferLearning');

if ~exist(dataFolder, 'dir')
    disp('Downloading pretrained network ...')
    unzip(url, AirCompressorLocation)
end
addpath(fullfile(AirCompressorLocation, 'YAMNetTransferLearning'))
```

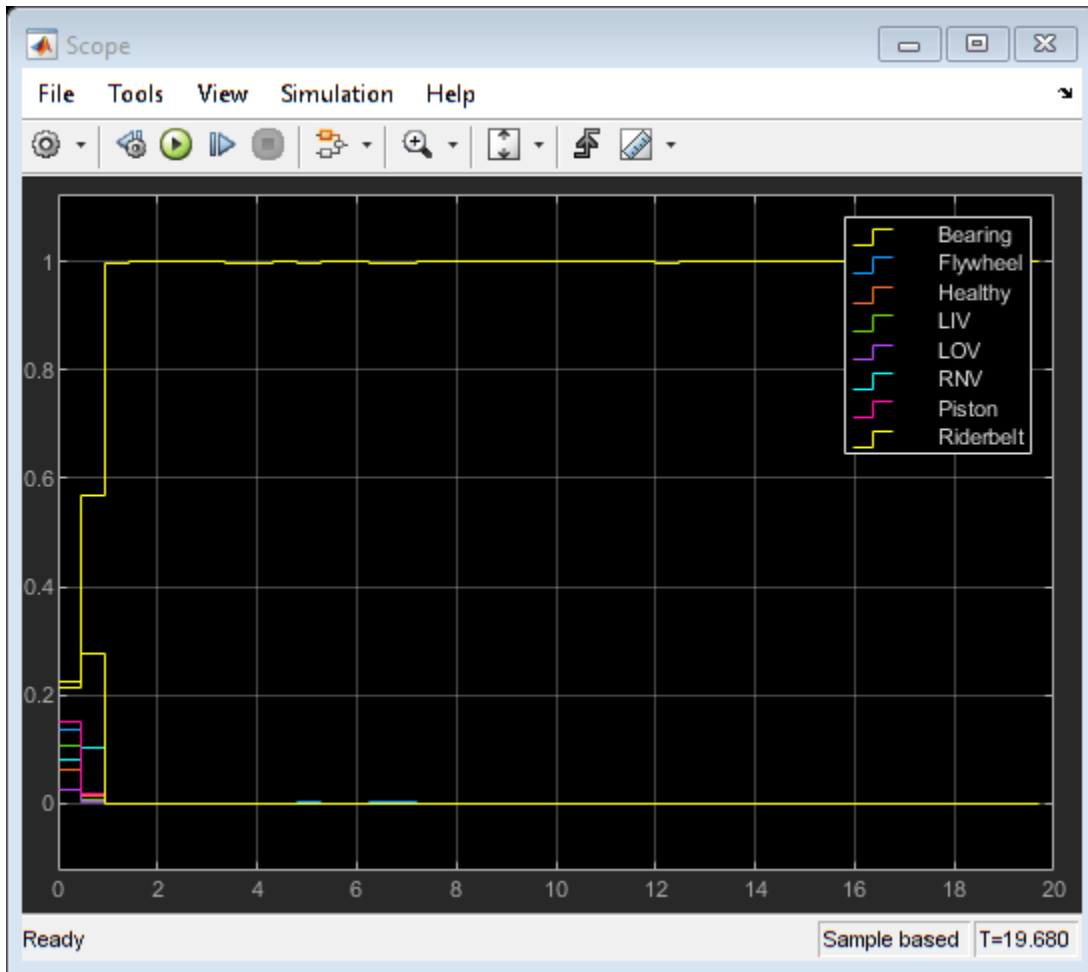
Open the `detectsound.slx` model. Click the `Select Compressor State` block. The default type of sound is set to 'Bearing'. The model contains a YAMNet Preprocess block followed by an Image Classifier (Deep Learning Toolbox) block.

Run the model. The YAMNet Preprocess block generates 96-by-64 sized mel spectrograms from the input audio. The Image Classifier block uses the `airCompressorNet.mat` file and classifies the signal into one of the eight classes the model is trained on. The label of the predicted class is displayed using the Display block. The scope shows the score of the predicted class and the other classes.

```
open_system("detectsound.slx")
sim("detectsound.slx")
```

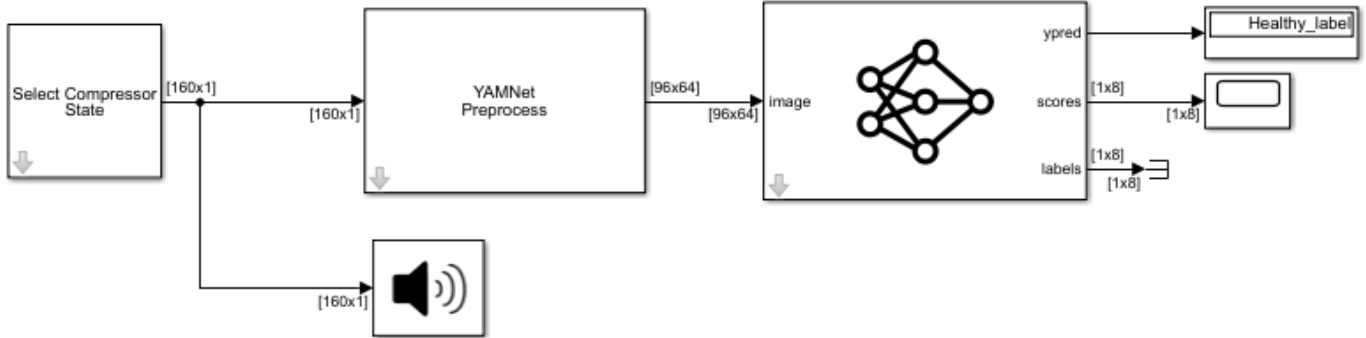


Copyright 2021 The MathWorks Inc.

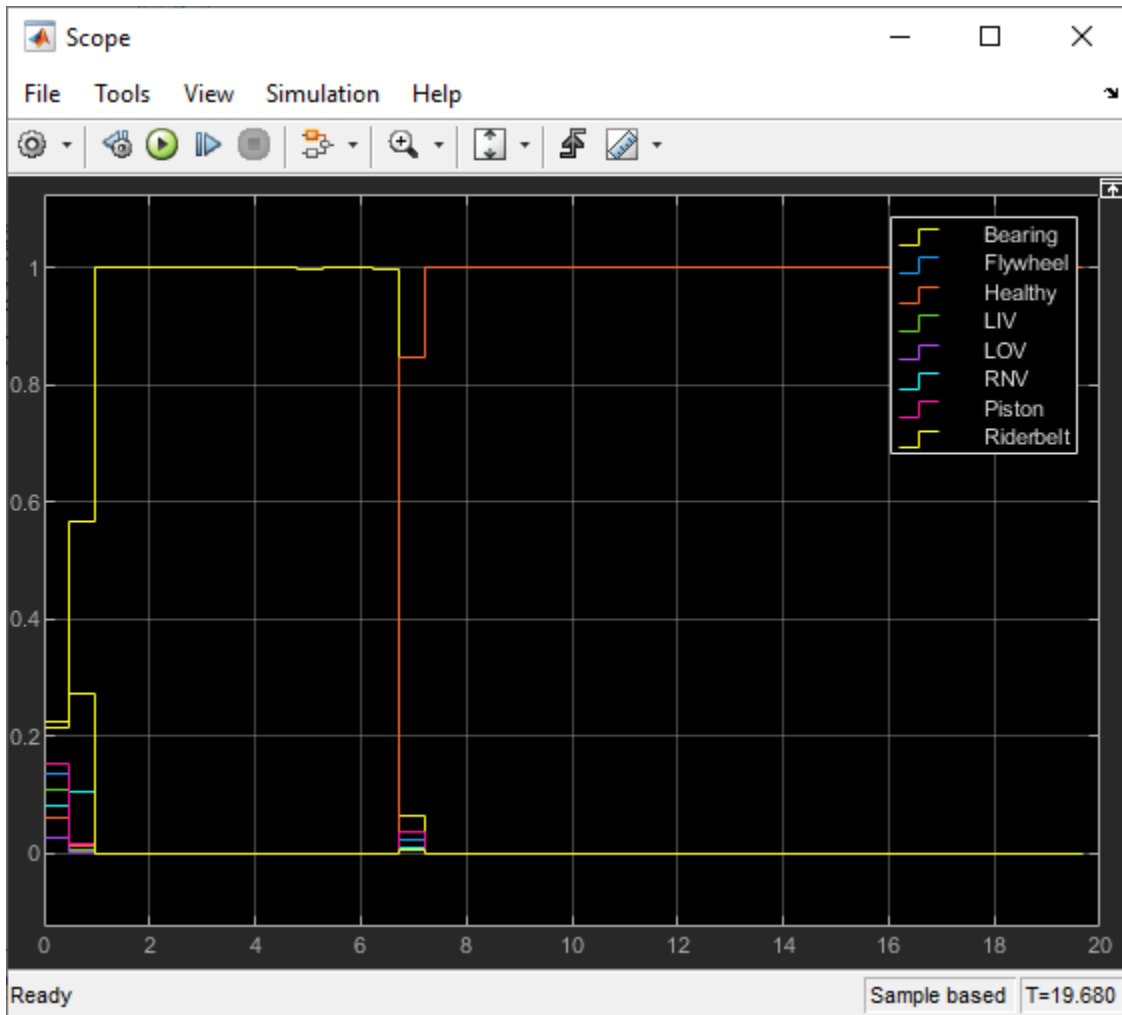


While the simulation is running, you can change the input sound by double clicking the Select Compressor State block and choosing a type of sound from the drop-down menu.

Select 'Healthy' while the simulation is running. The Display block updates the predicted label and the Scope block shows the new scores.



Copyright 2021 The MathWorks Inc.



See Also

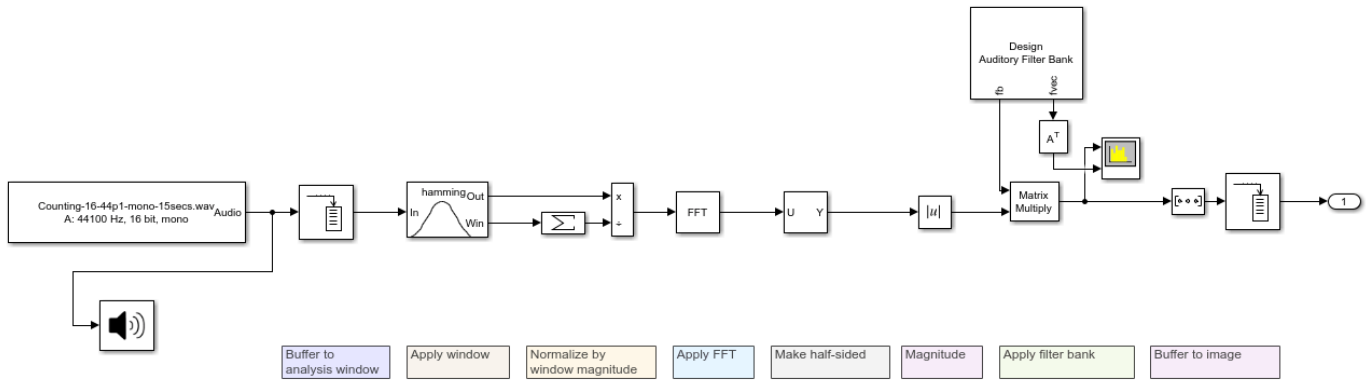
Image Classifier | YAMNet Preprocess

Related Examples

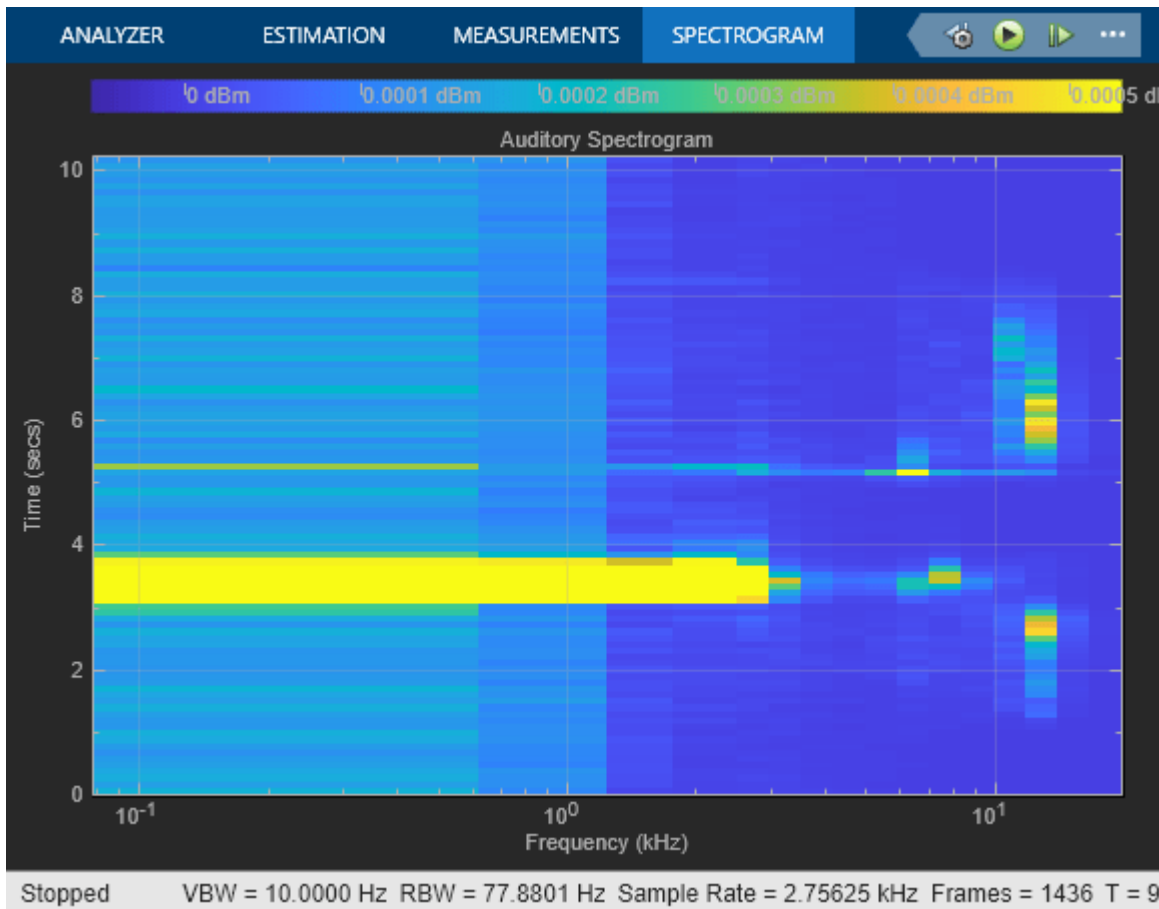
- “Detect Music in Simulink Using YAMNet” on page 17-57
- “Compare Sound Classifier block with Equivalent YAMNet blocks” on page 17-60

Design Auditory Filter Bank

Create an auditory filter bank and apply it to a signal in the frequency domain using the Design Auditory Filter Bank block in Simulink.

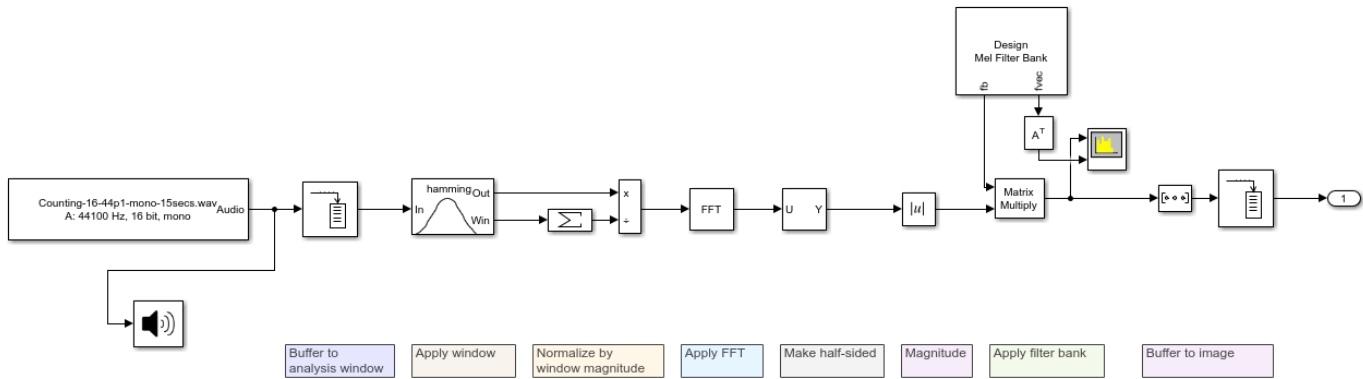


Copyright 2021 The MathWorks, Inc.

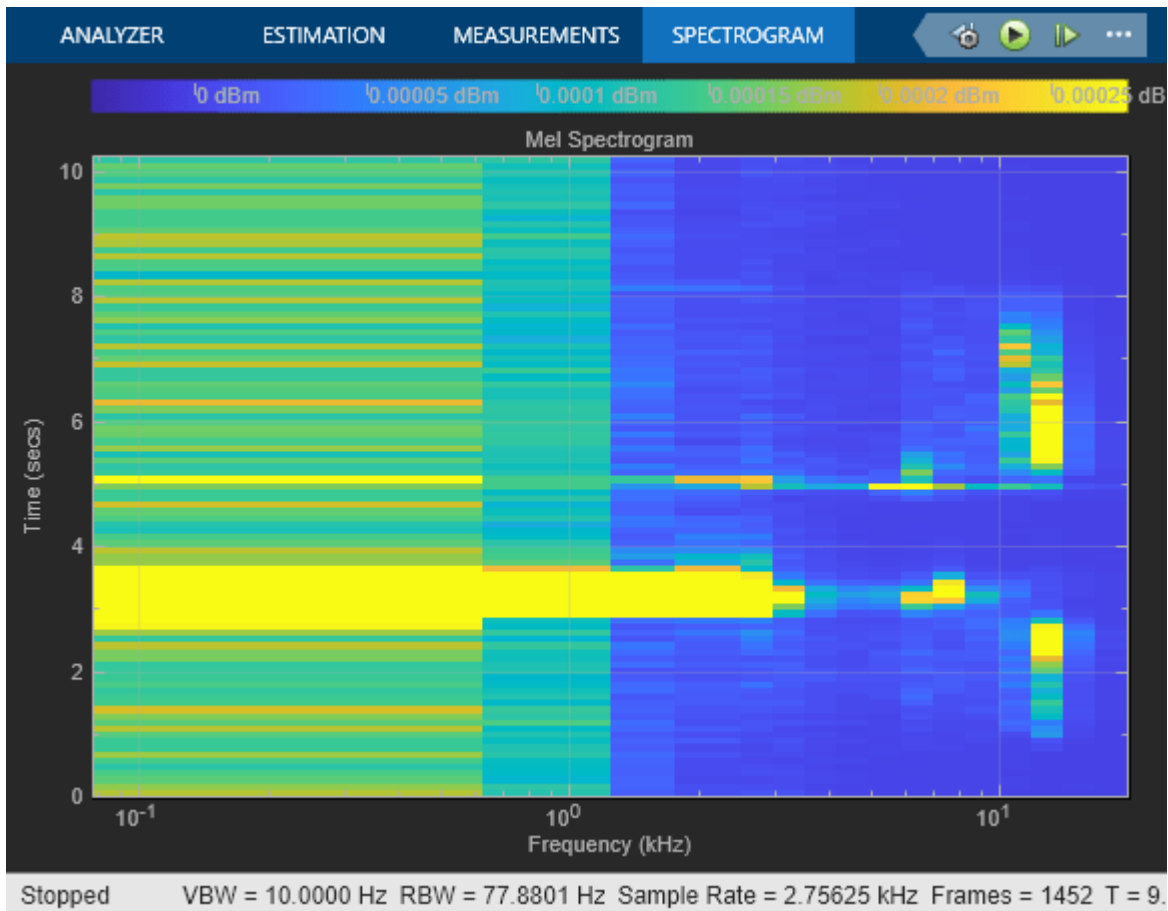


Design Mel Filter Bank

Create a mel filter bank and apply it to a signal in the frequency domain using the Design Mel Filter Bank block in Simulink.

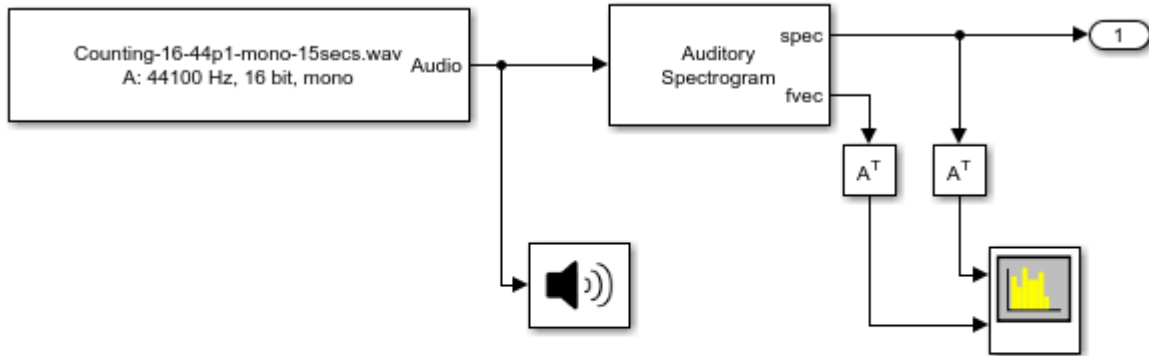


Copyright 2021 The MathWorks, Inc.

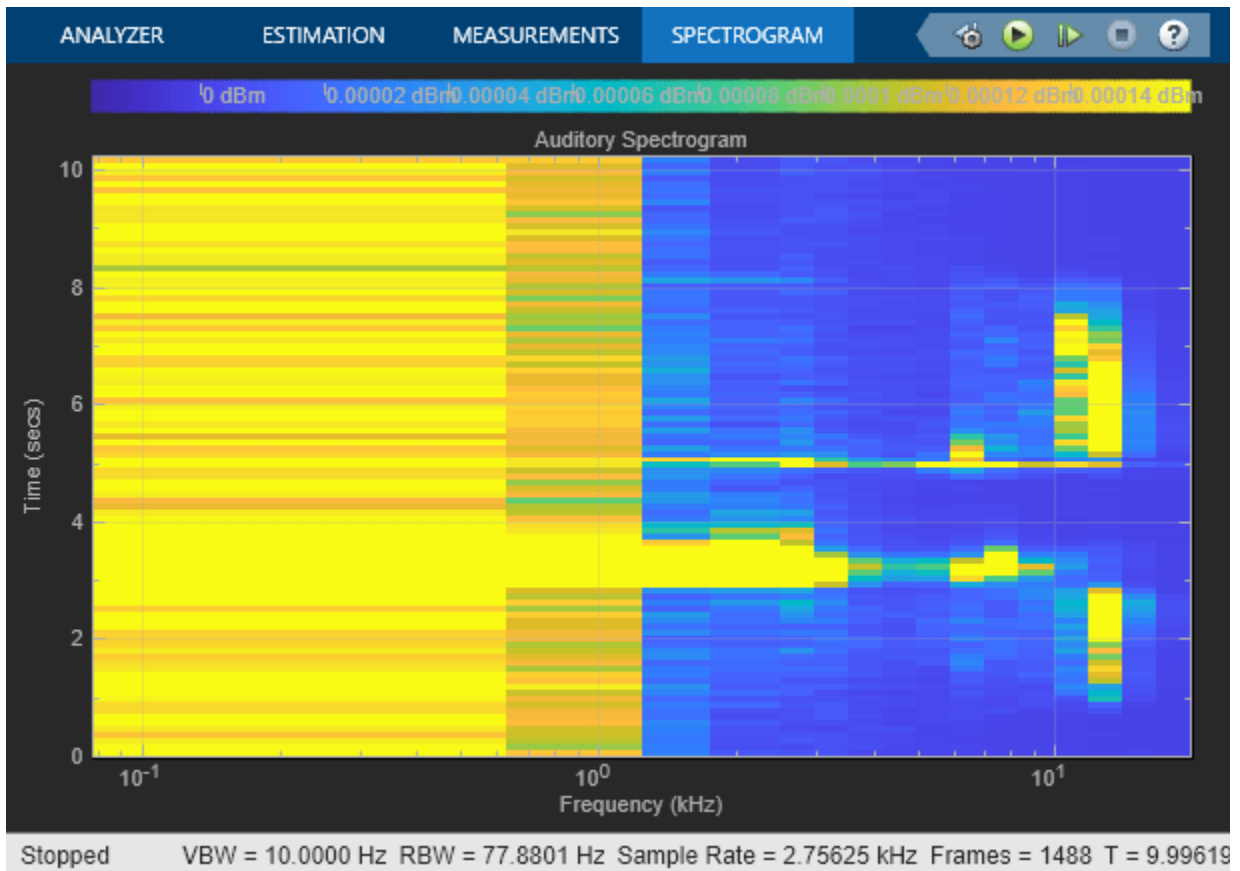


Extract Auditory Spectrogram

Extract an auditory spectrogram from a signal using the Auditory Spectrogram block in Simulink.

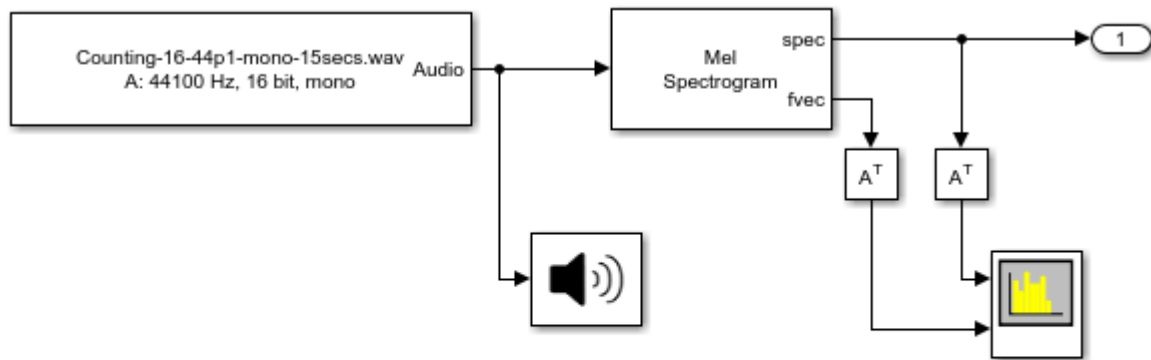


Copyright 2021 The MathWorks, Inc.

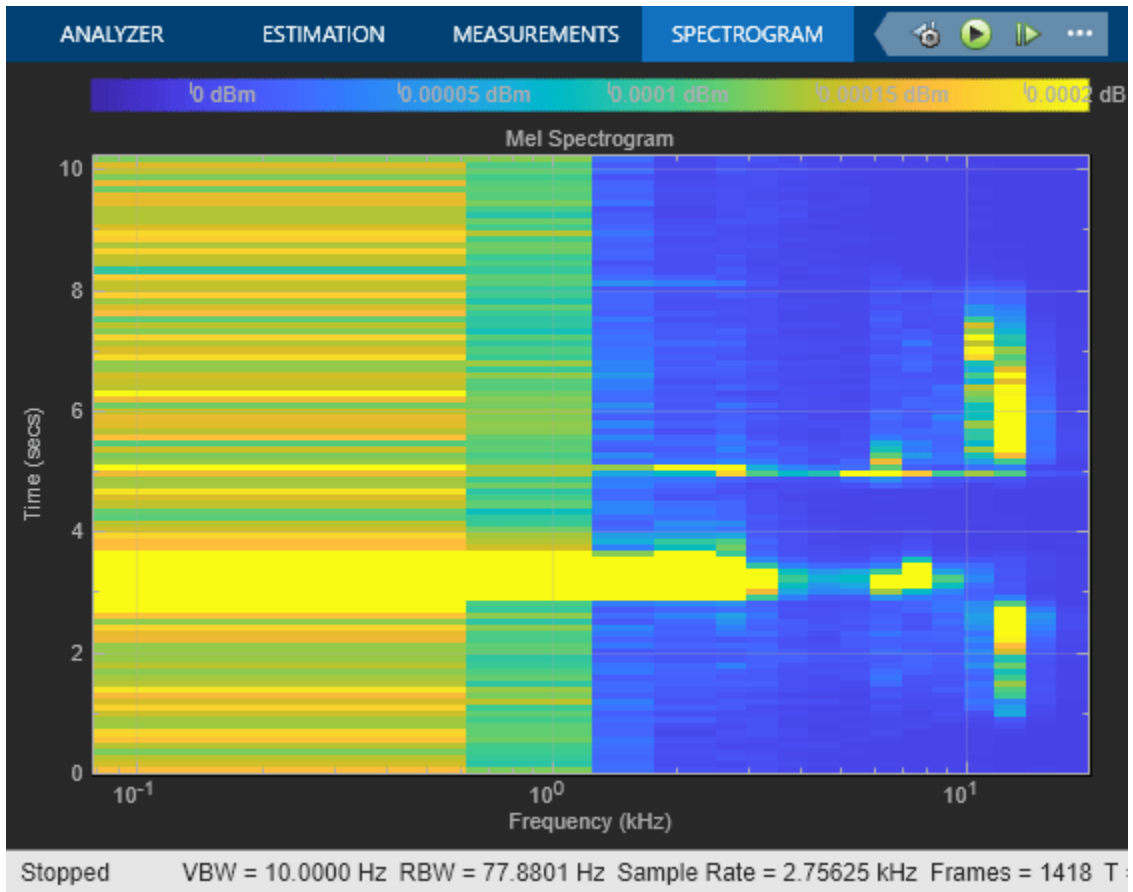


Extract Mel Spectrogram

Extract the mel spectrogram from an audio signal using the Mel Spectrogram block in Simulink.

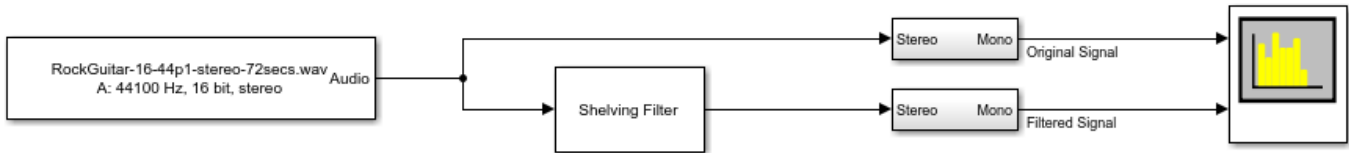


Copyright 2021 The MathWorks, Inc.

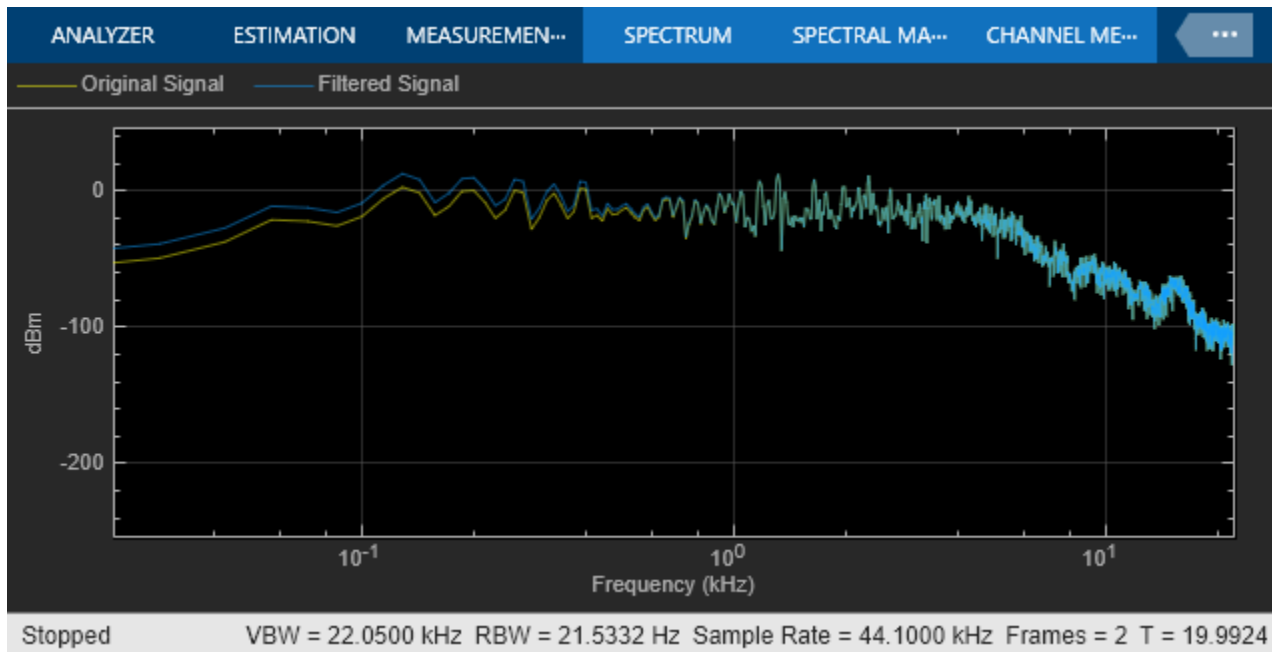


Filter Audio Using Shelving Filter Block

Use the Shelving Filter block to filter an audio signal in Simulink.



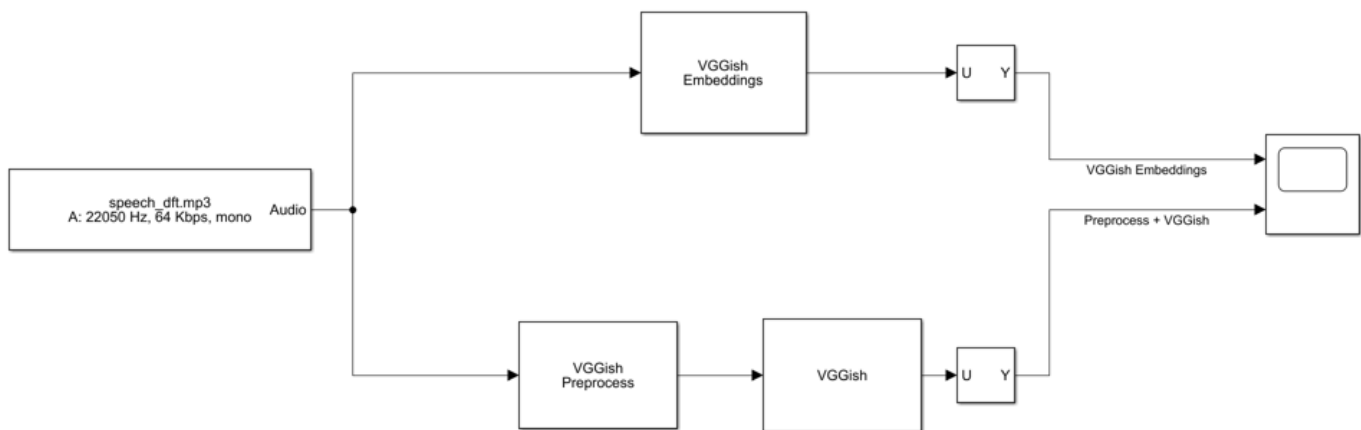
Copyright 2022 The MathWorks, Inc.



Compare VGGish Embeddings Block with Equivalent VGGish Blocks

The VGGish Embeddings block is equivalent to the cascade of the VGGish Preprocess block and VGGish block. The model in this example compares the two implementations and shows their equivalence.

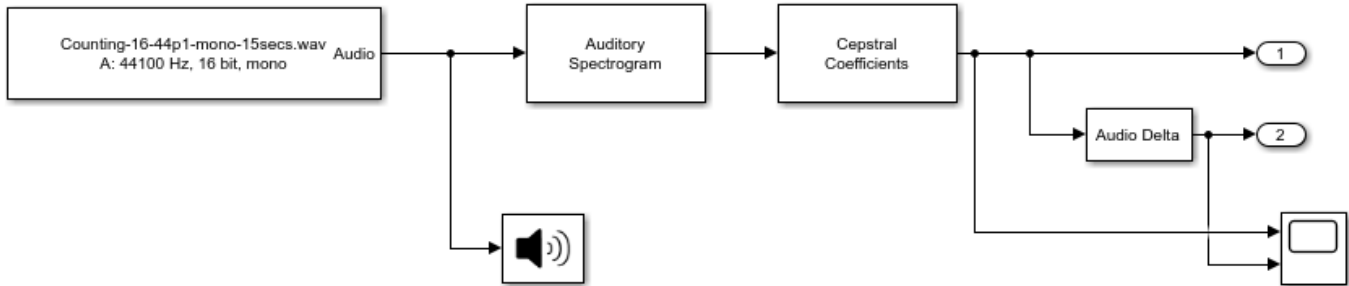
To use these blocks, a VGGish pretrained network must be installed in a location on the MATLAB® path. If a pretrained network is not installed, then open and run the model. The software provides a download link. To download the network, click the link and unzip the file to a location on the MATLAB path.



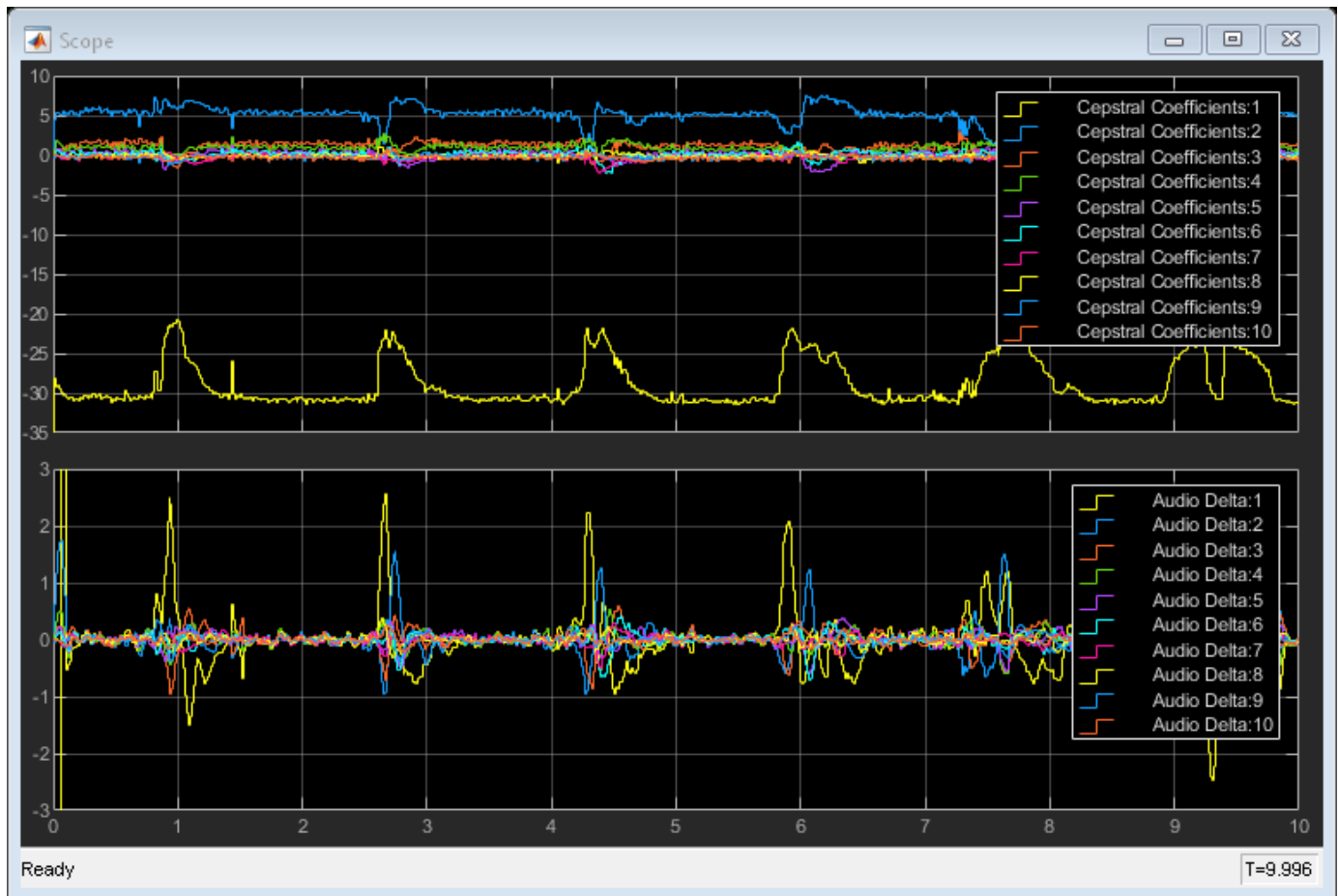
Copyright 2022 The MathWorks Inc.

Extract GTCC from Audio in Simulink

Extract gammatone cepstral coefficients and their delta features using the Auditory Spectrogram, Cepstral Coefficients, and Audio Delta blocks in Simulink.



Copyright 2022 The MathWorks, Inc.



Include an Audio Plugin in Simulink

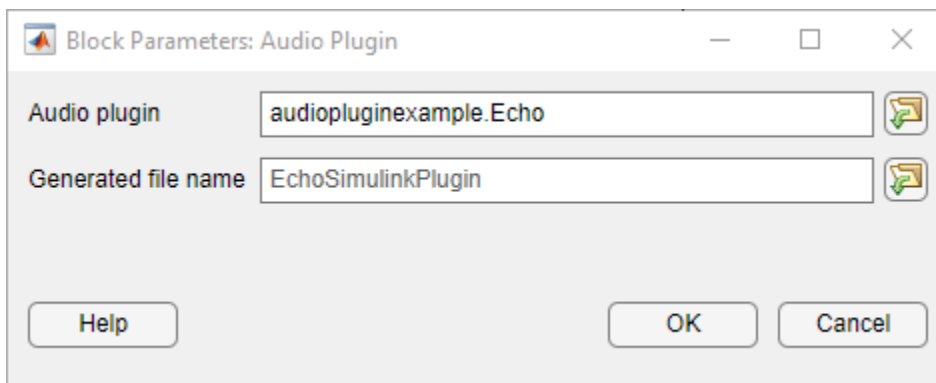
This example shows how to use the Audio Plugin block to include an audio plugin in a Simulink model.

Generate Block from Plugin

Place the Audio Plugin block in a Simulink model. You can find the block in the Audio Toolbox / User-Defined Functions library.



Double-click the block to open the dialog box. In the **Audio plugin** field, enter the plugin `audiopluginexample.Echo`. To inspect the source code for this plugin, enter `edit audiopluginexample.Echo` in the command line. Optionally, specify the name and location of the generated System object class file using the **Generated file name** field.



Click **OK** to generate a block with the same functionality as the plugin. This also generates the System object class file and places it in the current directory by default. The file must be on the MATLAB path for the generated block to work.



Double-clicking the new block opens the parameter dialog box, where you can view and edit the plugin parameters. You can also choose to specify the tunable parameters through additional input ports on the block.

Block Parameters: Audio Plugin

EchoSimulinkPlugin

This block is based on the System object EchoSimulinkPlugin, generated from the plugin audiopluginexample.Echo.

[Source code](#)

Parameters

Specify Base delay from input port

Base delay (Plugin range [0 1] s): 0.5

Specify Gain from input port

Gain (Plugin range [0 1]): 0.5

Specify Feedback from input port

Feedback (Plugin range [0 0.5]): 0.35

Specify Wet/dry mix from input port

Wet/dry mix (Plugin range [0 1]): 0.5

Inherit sample rate from input

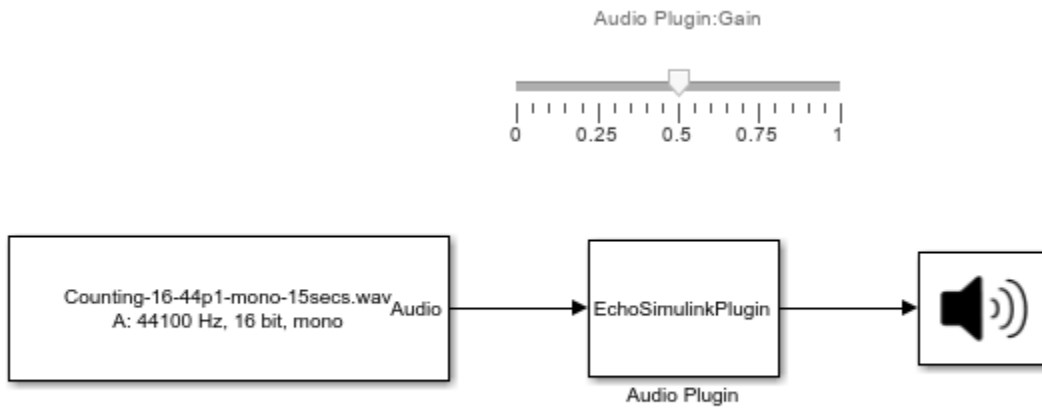
Input sample rate (Hz): 44100

Simulate using: Code generation

OK Cancel Help Apply

Use Generated Audio Plugin Block in Model

Use the plugin in a model to process an audio signal and listen to the results. Add a Slider (Simulink) block to the model to tune the gain parameter of the plugin during simulation.



Copyright 2022 The MathWorks, Inc.

See Also

Audio Plugin

More About

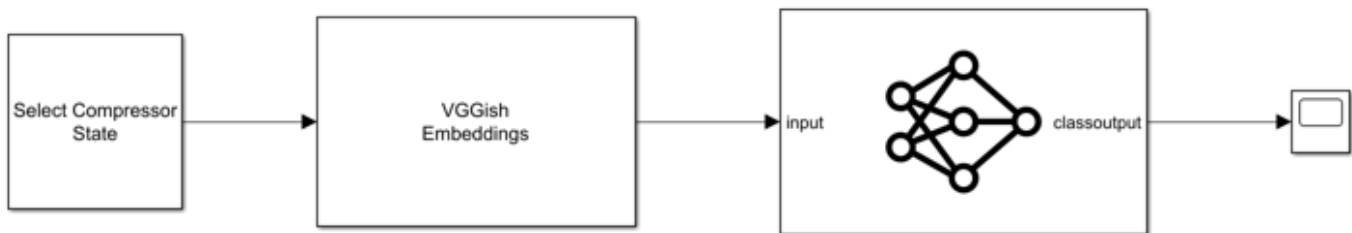
- “Audio Plugins in MATLAB”

Use VGGish Embeddings for Deep Learning in Simulink

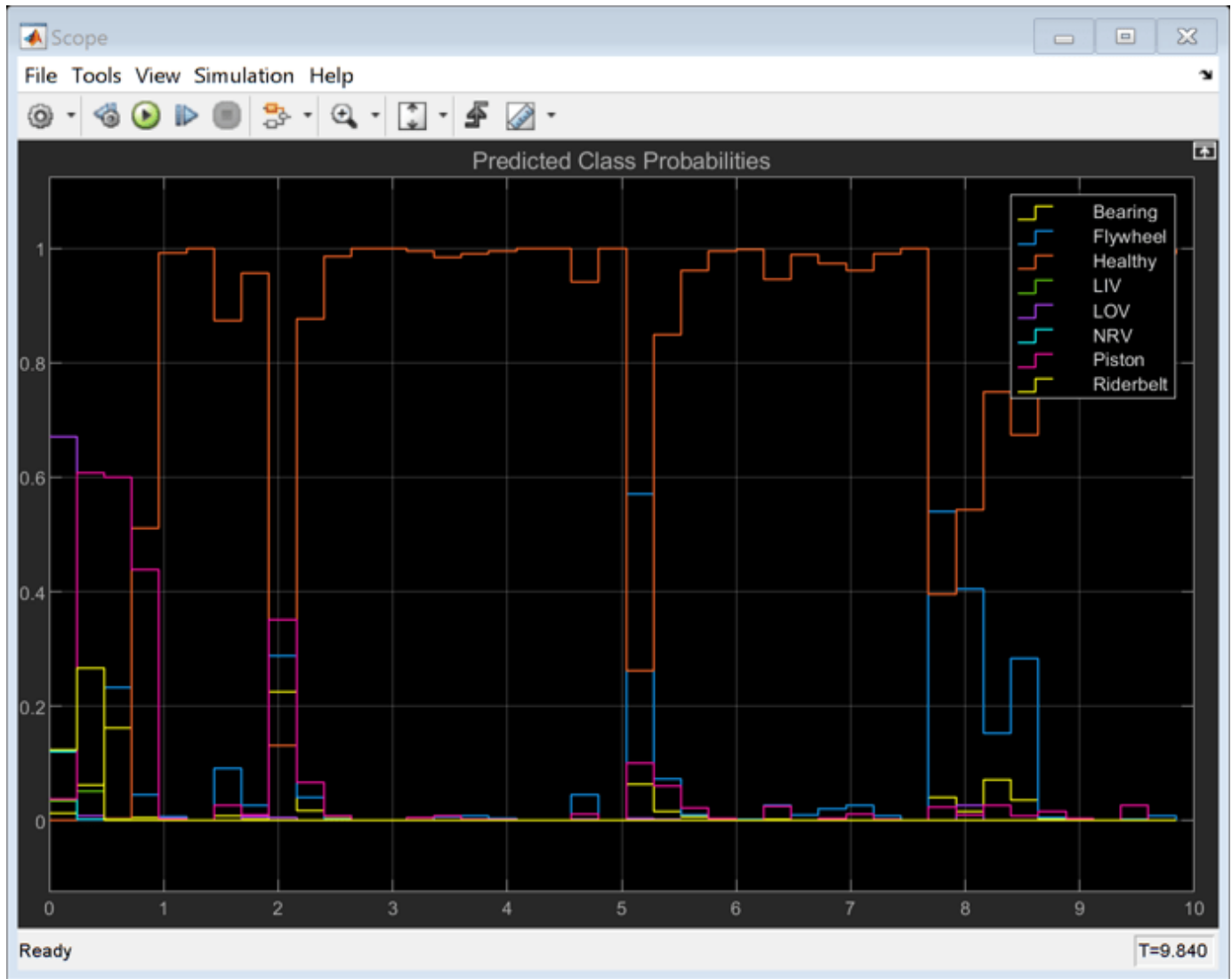
This example shows how to use a simple neural network in Simulink® to classify audio signals from their VGGish feature embeddings using the VGGish Embeddings and Predict (Deep Learning Toolbox) blocks.

The network is a small fully connected network that was trained on VGGish feature embeddings extracted from air compressor audio signals. The air compressor data set consists of recordings from air compressors in a healthy state or one of seven faulty states. For information on how the network was trained, see “Use VGGish Embeddings for Deep Learning”.

While the simulation is running, you can change the input sound by double clicking the Select Compressor State block and choosing a type of sound from the drop-down menu. After you change the air compressor sound, see how the predicted class probabilities change.



Copyright 2022 The MathWorks, Inc.



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.

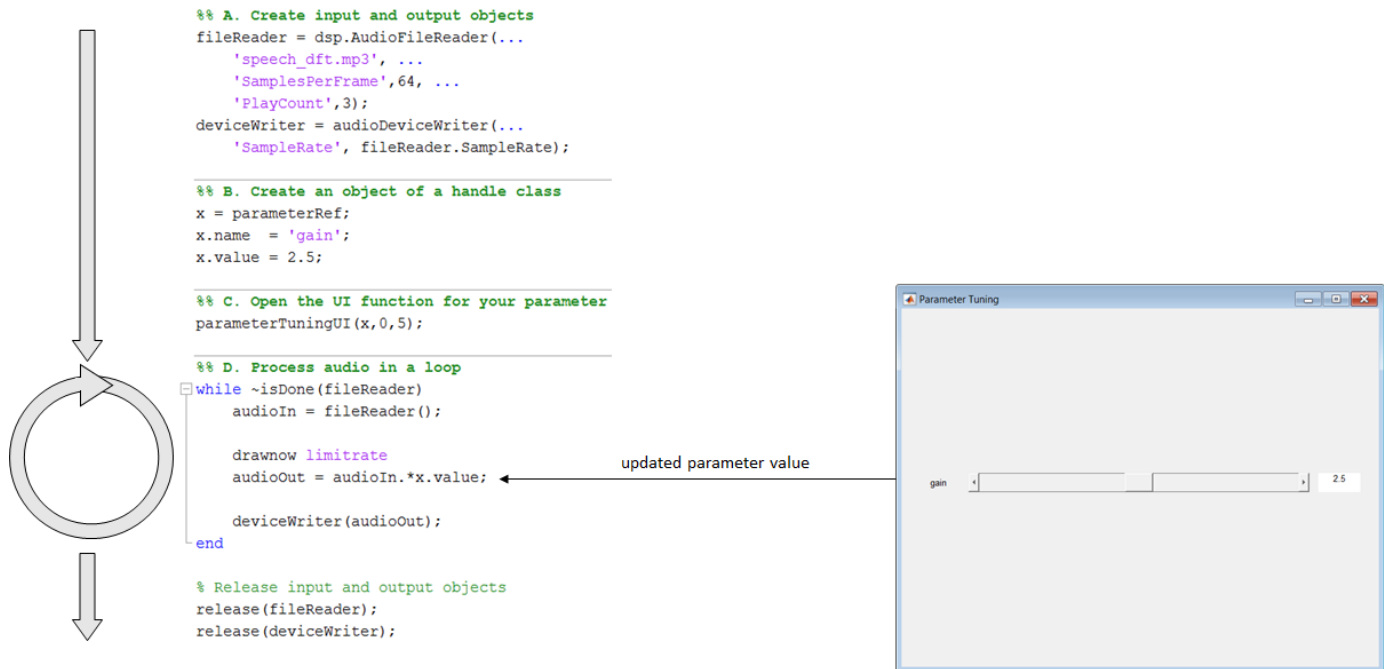
Programmatic Parameter Tuning

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.

Open the `parameterRef` class definition file.

```

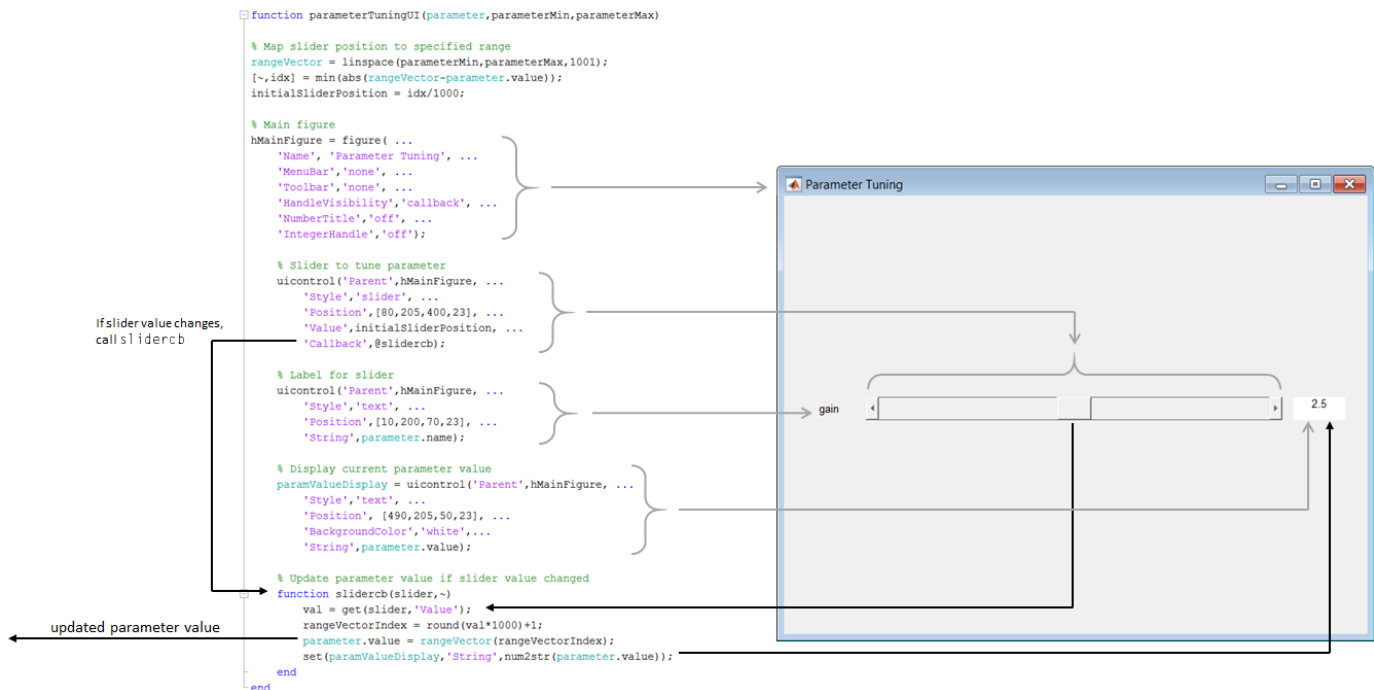
classdef parameterRef < handle
    properties
        name
        value
    end
end

```

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.



parameterTuningUI

Open parameterTuningUI.

```
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`, 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”
- “Develop, Analyze, and Debug Plugins In Audio Test Bench” 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.

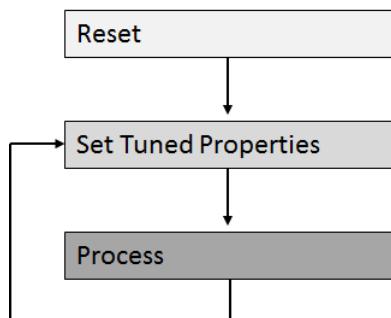
To learn more about audio plugins in general, see “Audio Plugins in MATLAB”.

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)

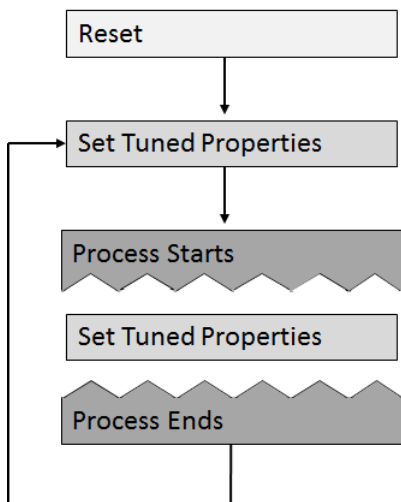
        L = plugin.Gain*in(:,1);
        drawnow

        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
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
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), ...
        'TimeSpanOvverrunAction','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
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)
    ol = process(plugin, in(:,1));

Error in validateAudioPlugin
```

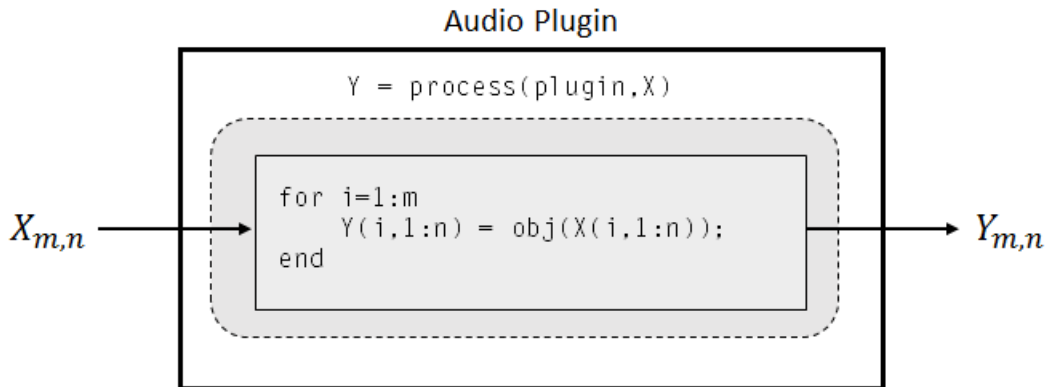
See BrokenAnalyticSignalTransformer Code

```
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

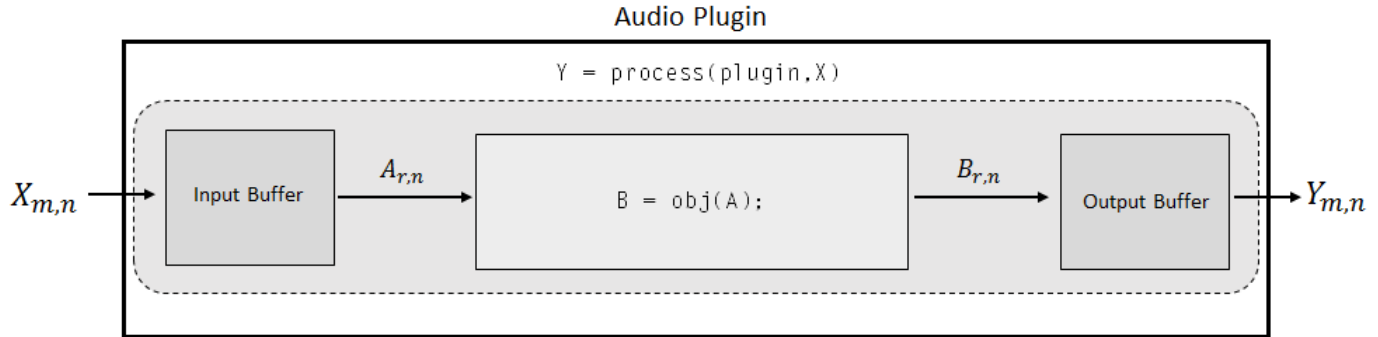
```

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
  
```

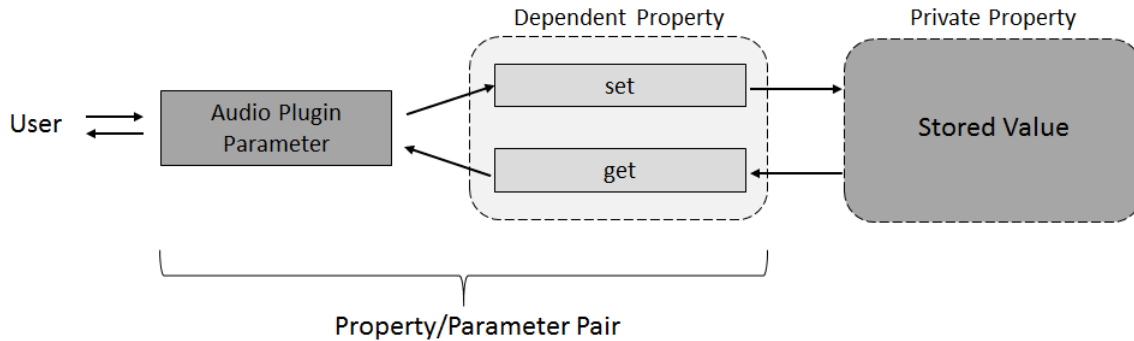
Note Use of the asynchronous buffering object forces a minimum latency of your specified frame size.

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 20-25]
- Acoustic scene recognition [11 on page 20-24] [17 on page 20-24]
- Instrument recognition [22 on page 20-25]
- Music genre classification [16 on page 20-24] [18 on page 20-24]
- Mood recognition [19 on page 20-25] [20 on page 20-25]
- Voice activity detection [5 on page 20-23] [7 on page 20-24] [8 on page 20-24] [10 on page 20-24] [12 on page 20-24] [13 on page 20-24]

Spectral Centroid

The spectral centroid (`spectralCentroid`) is the frequency-weighted sum normalized by the unweighted sum [1 on page 20-23]:

$$\mu_1 = \frac{\sum_{k=b_1}^{b_2} f_k s_k}{\sum_{k=b_1}^{b_2} s_k}$$

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 20-23] 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')
```

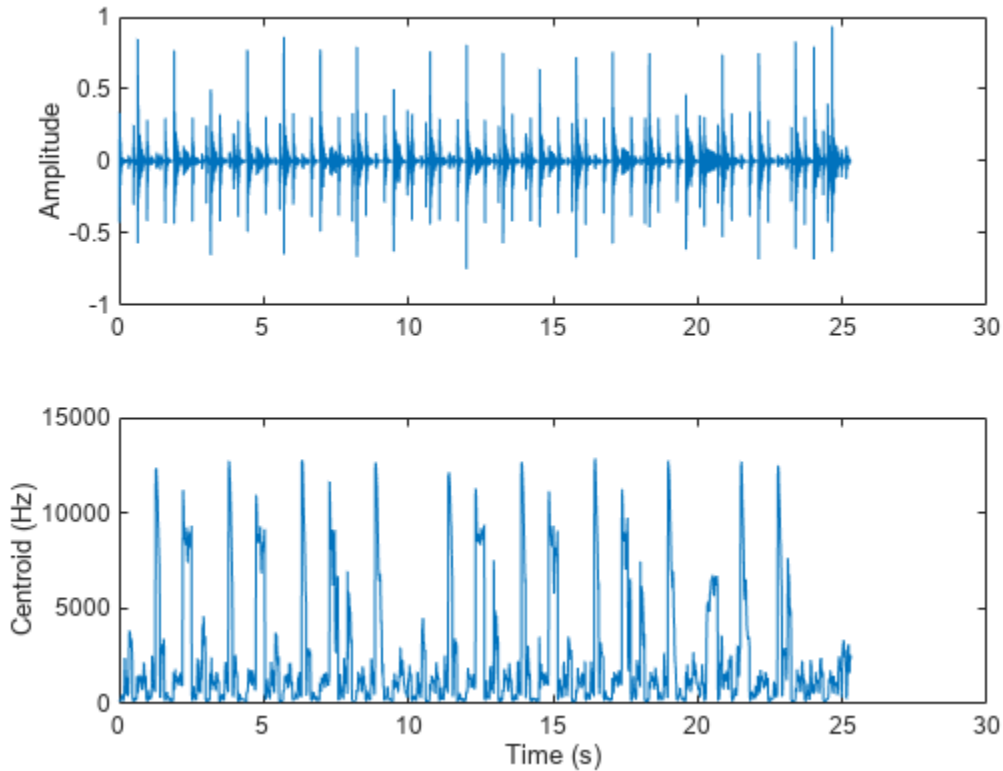
```
subplot(2,1,2)
t = linspace(0,size(audio,1)/fs,size(audio,1));
```



```

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 20-23]. For example, the centroid jumps in regions of unvoiced speech.

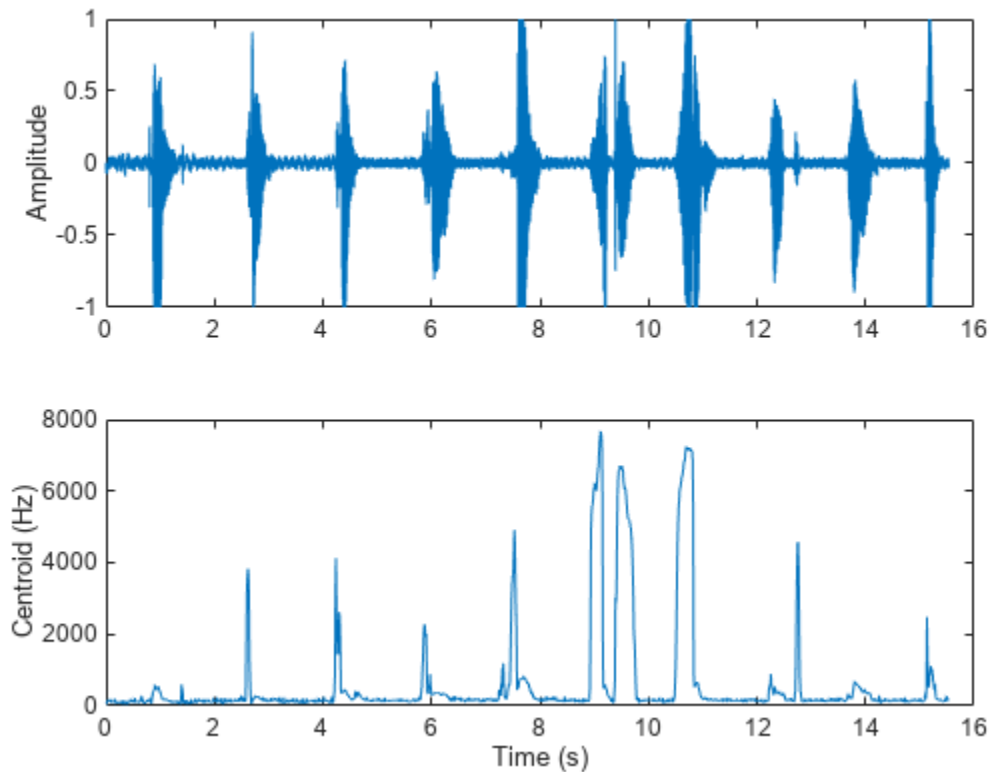
```

[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 20-23]:

$$\mu_2 = \sqrt{\frac{\sum_{k=b_1}^{b_2} (f_k - \mu_1)^2 s_k}{\sum_{k=b_1}^{b_2} s_k}}$$

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;
```

```

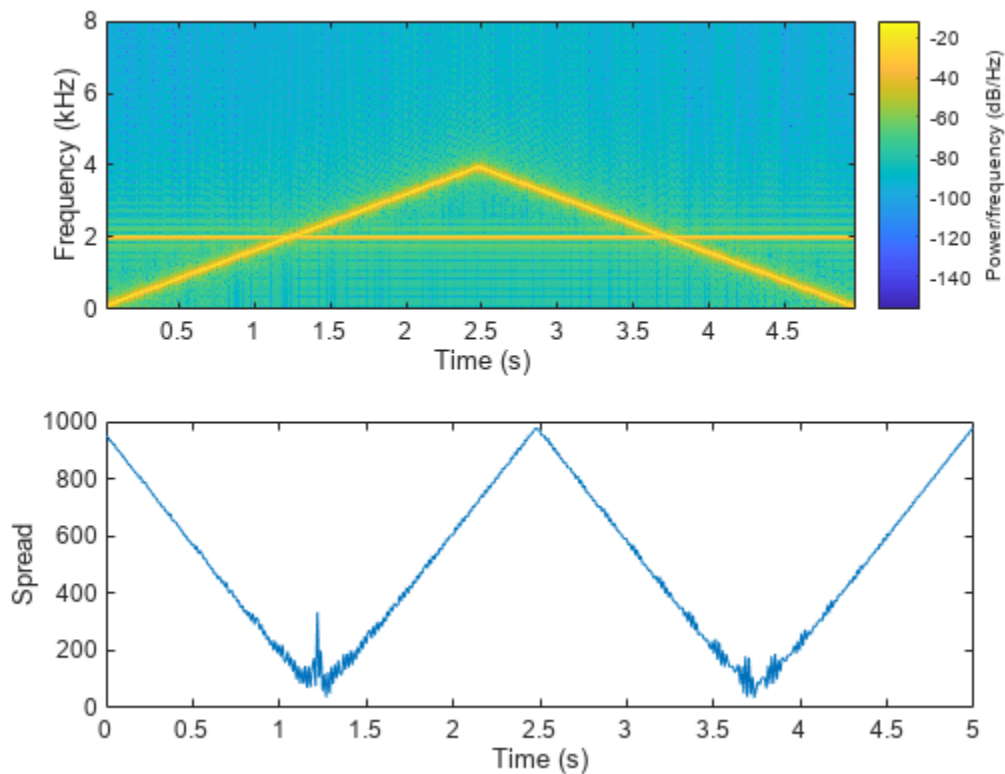
numLoops = floor(duration*fs/tone.SamplesPerFrame);
signal = [];
for i = 1:numLoops
    signal = [signal;tone()];
    if i<numLoops/2
        tone.Frequency = tone.Frequency + [0,50];
    else
        tone.Frequency = tone.Frequency - [0,50];
    end
end

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 20-23]:

$$\mu_3 = \frac{\sum_{k=b_1}^{b_2} (f_k - \mu_1)^3 s_k}{(\mu_2)^3 \sum_{k=b_1}^{b_2} s_k}$$

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 20-23]. 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.

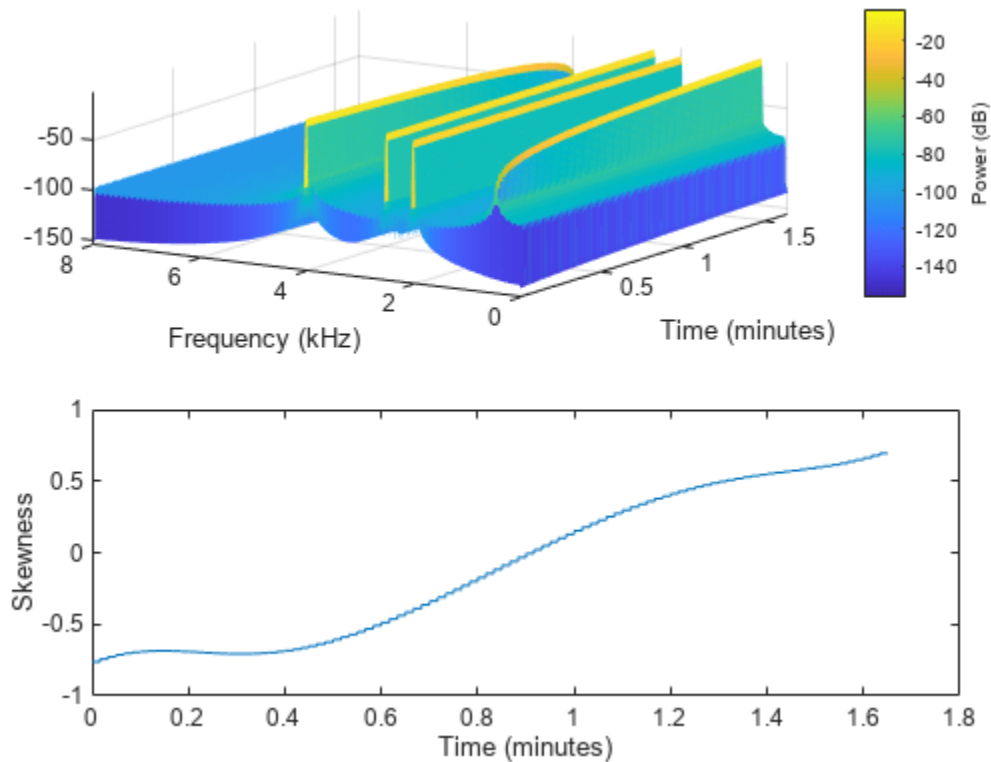
```
fs = 16e3;
duration = 99;
tone = audioOscillator('SampleRate', fs, 'NumTones', 4, 'SamplesPerFrame', fs, 'Frequency', [500, 2000, 2000, 500]);

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 20-23]:

$$\mu_4 = \frac{\sum_{k=b_1}^{b_2} (f_k - \mu_1)^4 s_k}{(\mu_2)^4 \sum_{k=b_1}^{b_2} s_k}$$

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));

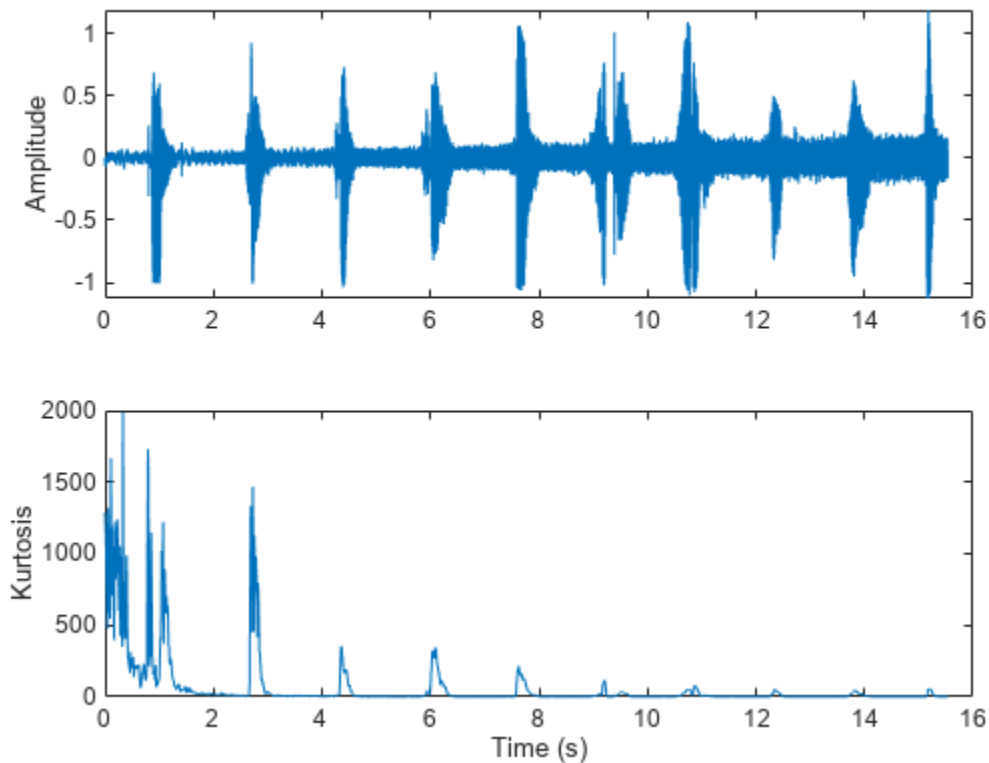
noise = noiseGenerator();
noise = noise/max(abs(noise));
ramp = linspace(0,.25,numel(noise))';
noise = noise.*ramp;

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 20-23]:

$$\text{entropy} = \frac{-\sum_{k=b_1}^{b_2} s_k \log(s_k)}{\log(b_2 - b_1)}$$

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.

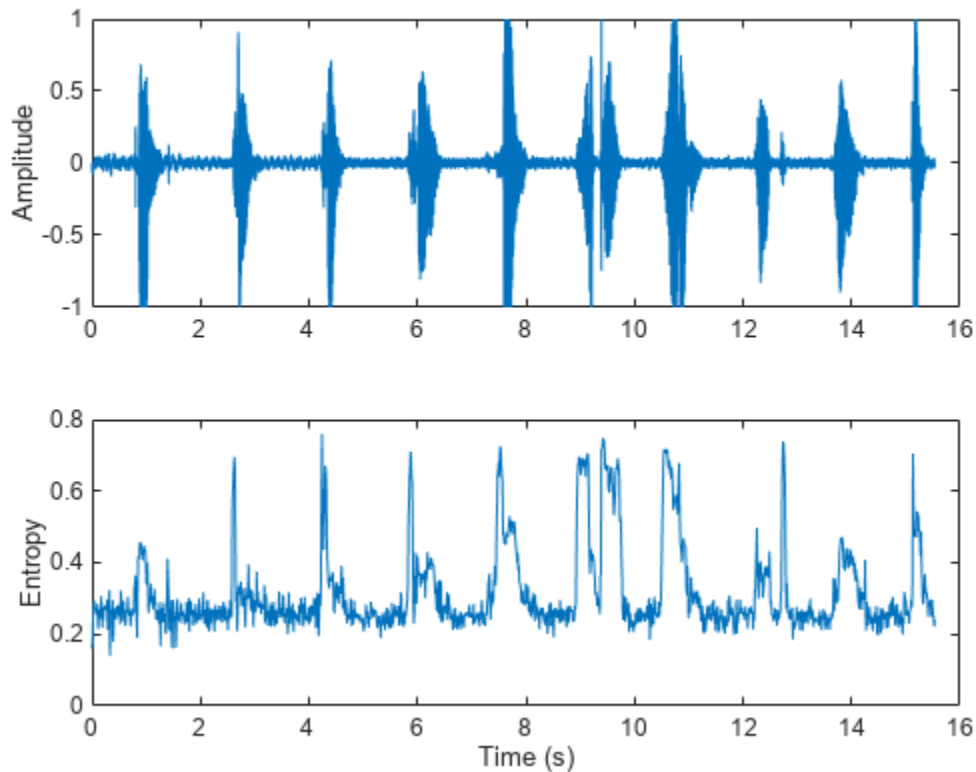
Spectral entropy has been used successfully in voiced/unvoiced decisions for automatic speech recognition [6 on page 20-23]. Because entropy is a measure of disorder, regions of voiced speech have lower entropy compared to regions of unvoiced speech.

```
[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 20-24] [8 on page 20-24]. 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);

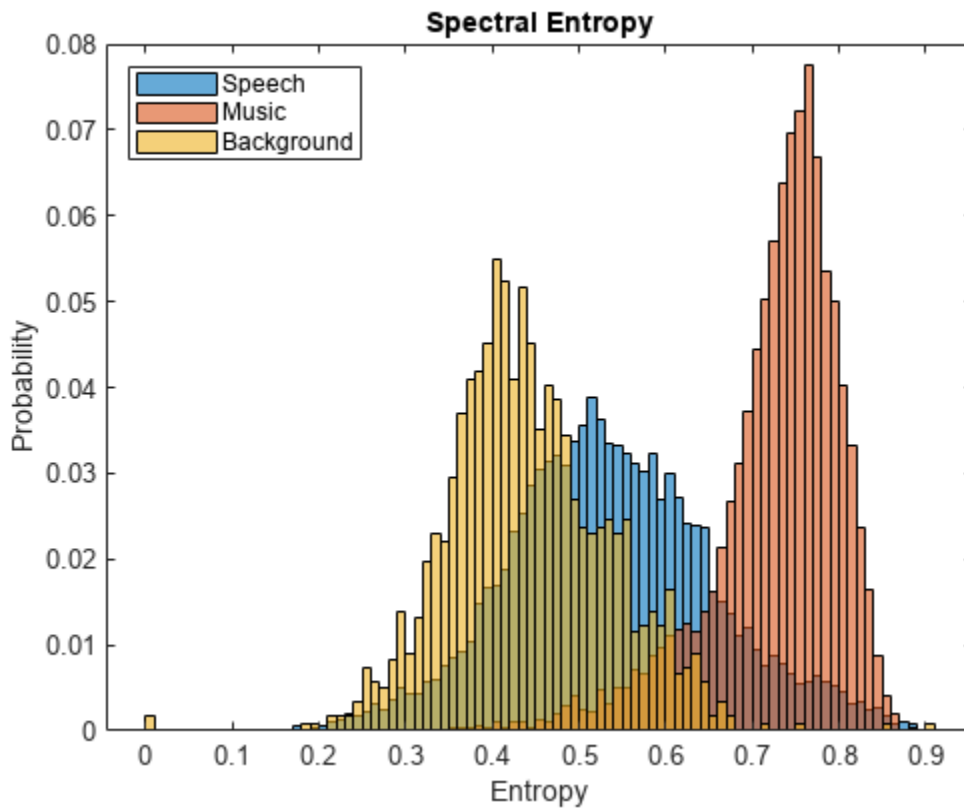
```



```

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 20-24]:

$$\text{flatness} = \frac{\left(\prod_{k=b_1}^{b_2} s_k \right)^{\frac{1}{b_2 - b_1}}}{\frac{1}{b_2 - b_1} \sum_{k=b_1}^{b_2} s_k}$$

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;

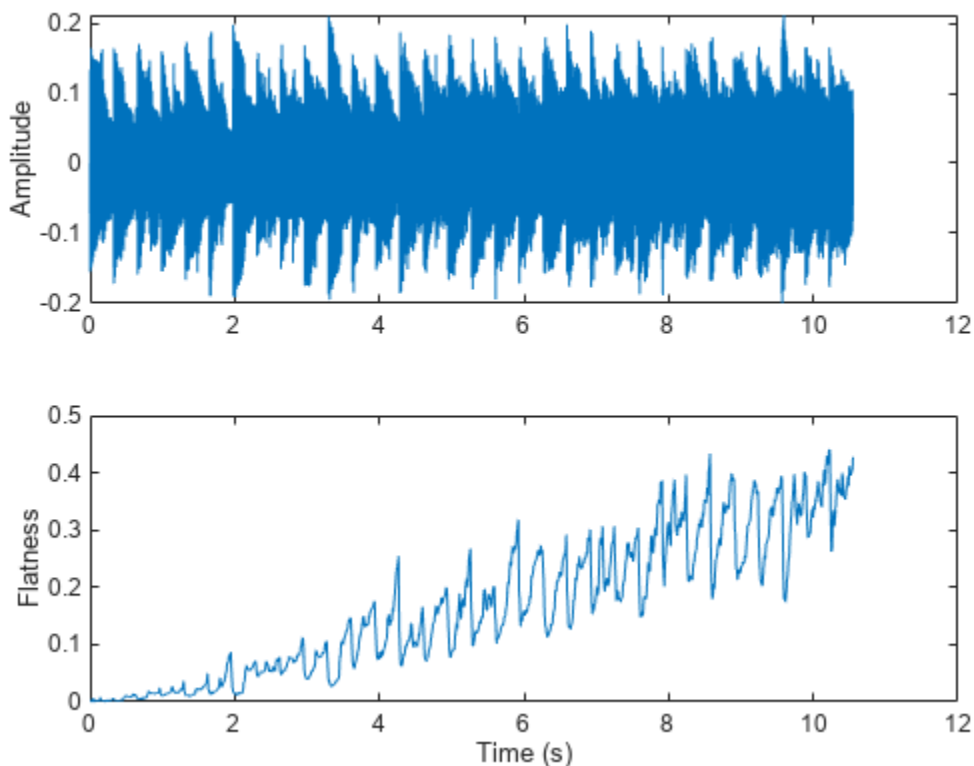
noise = (2*rand(numel(audio),1)-1).*linspace(0,0.05,numel(audio))';

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 20-24] and to audio scene recognition [11 on page 20-24].

Spectral Crest

Spectral crest (`spectralCrest`) measures the ratio of the maximum of the spectrum to the arithmetic mean of the spectrum [1 on page 20-23]:

$$\text{crest} = \frac{\max(s_{k \in [b_1, b_2]})}{\frac{1}{b_2 - b_1} \sum_{k=b_1}^{b_2} s_k}$$

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;

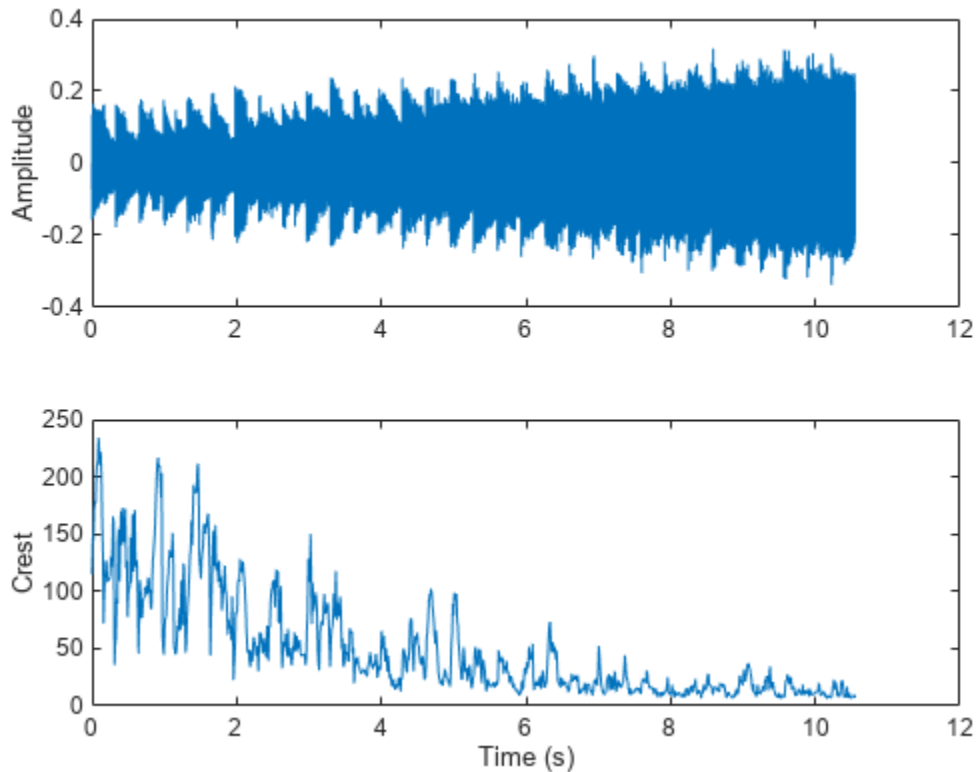
noise = (2*rand(numel(audio),1)-1).*linspace(0,0.2,numel(audio))';

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 20-24]:

$$\text{flux}(t) = \left(\sum_{k=b_1}^{b_2} |s_k(t) - s_k(t-1)|^p \right)^{\frac{1}{p}}$$

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 20-24] and audio segmentation [14 on page 20-24]. 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);
```

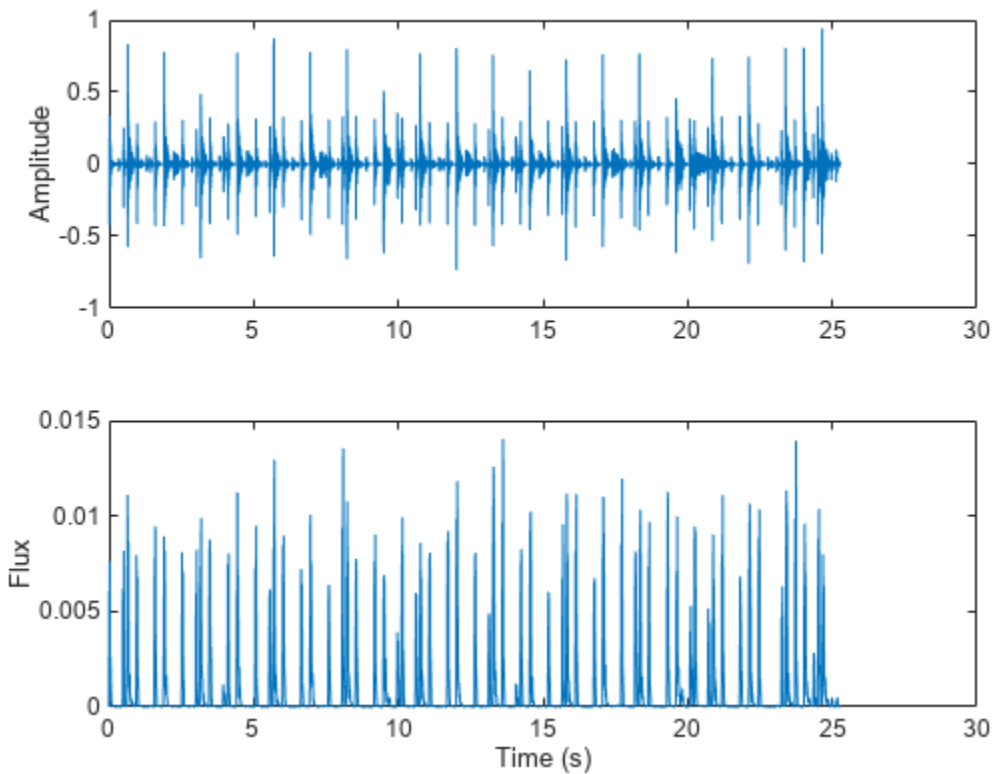
```
subplot(2,1,1)
```

```

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 20-24]:

$$\text{slope} = \frac{\sum_{k=b_1}^{b_2} (f_k - \mu_f)(s_k - \mu_s)}{\sum_{k=b_1}^{b_2} (f_k - \mu_f)^2}$$

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.

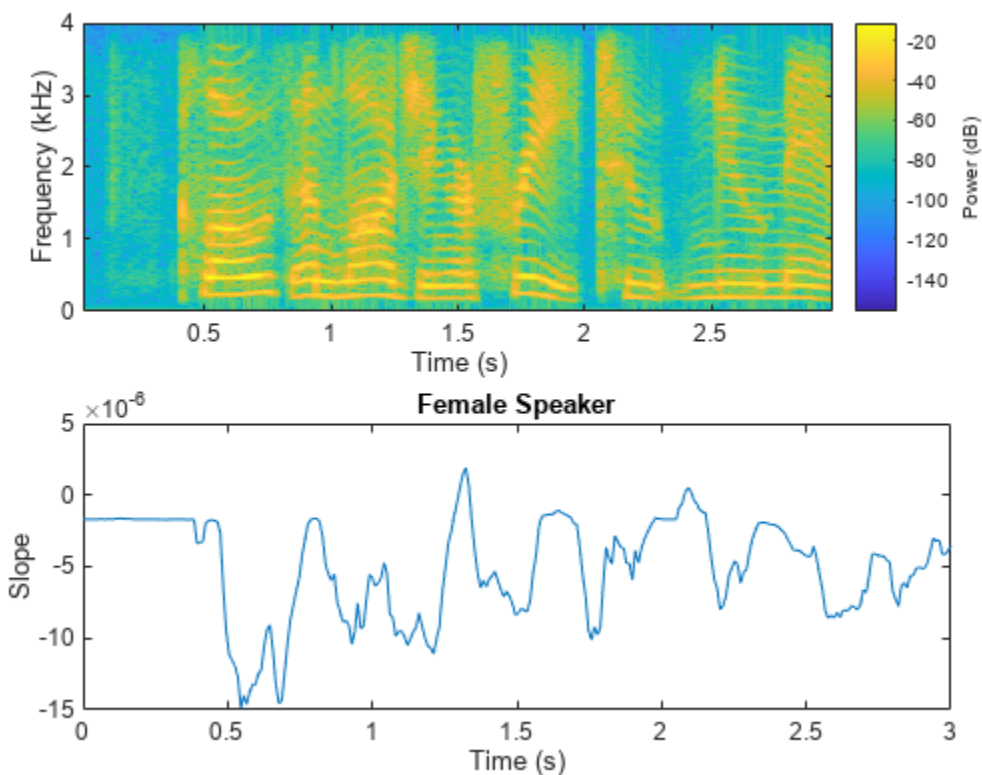
- μ_s is the mean spectral value.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral slope.

Spectral slope has been used extensively in speech analysis, particularly in modeling speaker stress [19 on page 20-25]. The slope is directly related to the resonant characteristics of the vocal folds and has also been applied to speaker identification [21 on page 20-25]. Spectral slope is a socially important aspect of timbre. Spectral slope discrimination has been shown to occur in early childhood development [20 on page 20-25]. 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 20-23]:

$$\text{decrease} = \frac{\sum_{k=b_1+1}^{b_2} \frac{s_k - s_{b_1}}{k-1}}{\sum_{k=b_1+1}^{b_2} s_k}$$

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 20-25].

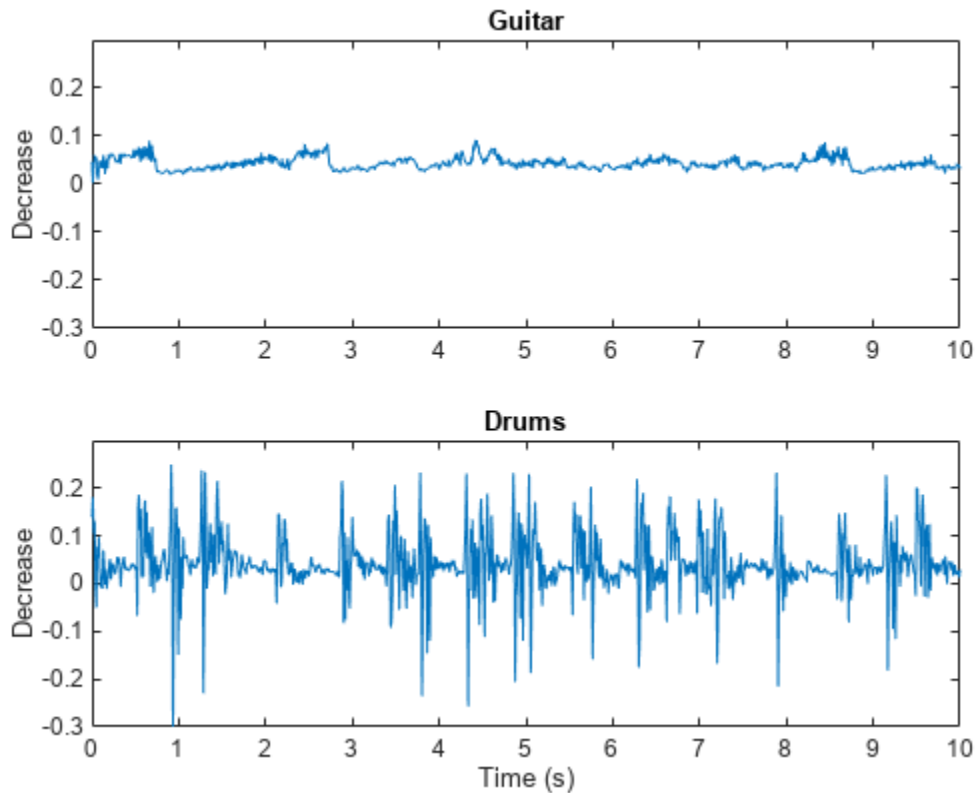
```
[guitar,guitarFs] = audioread('RockGuitar-16-44p1-stereo-72secs.wav');
guitar = mean(guitar,2);
[drums,drumsFs] = audioread('RockDrums-44p1-stereo-11secs.mp3');
drums = mean(drums,2);
```

```
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 20-24]:

$$\text{Rolloff Point} = i \quad \text{such that} \quad \sum_{k=b_1}^i |s_k| = \kappa \sum_{k=b_1}^{b_2} s_k$$

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 20-24], music genre classification [16 on page 20-24], acoustic scene recognition [17 on page 20-24], and music mood classification [18 on page 20-24]. 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);

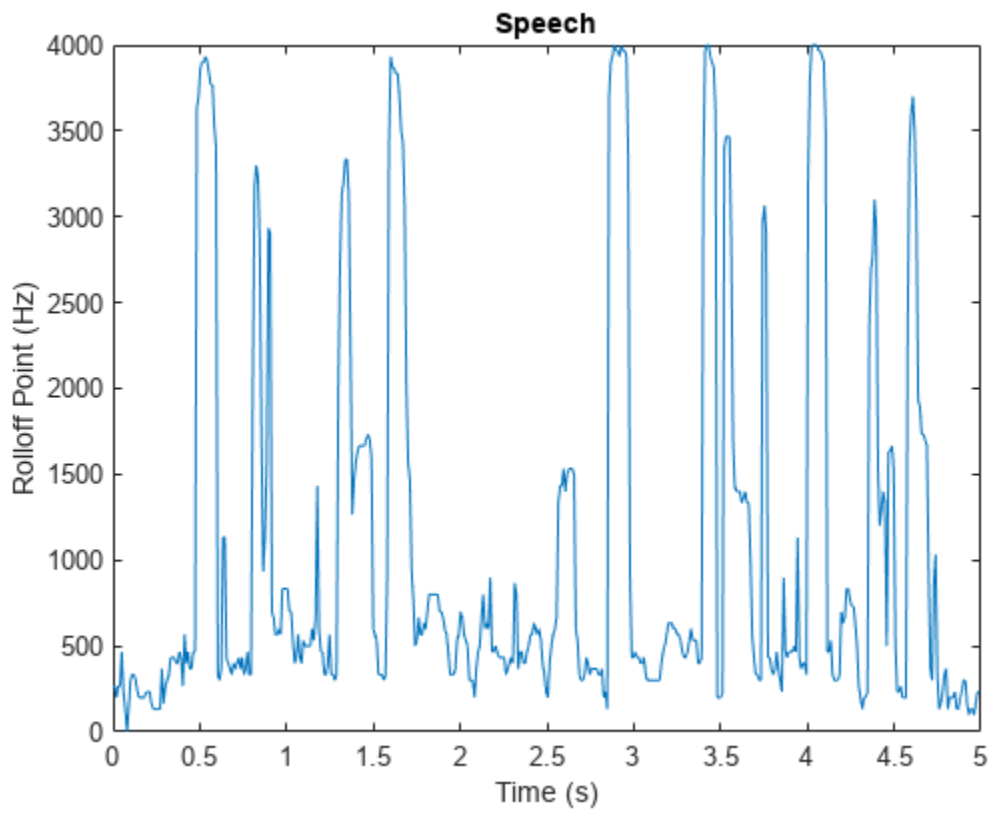
[acousticGuitar,fs3] = audioread('SoftGuitar-44p1_mono-10mins.ogg');
acousticGuitar = acousticGuitar(1:fs3*dur);

[acousticScene,fs4] = audioread('MainStreetOne-16-16-mono-12secs.wav');
acousticScene = acousticScene(1:fs4*dur);

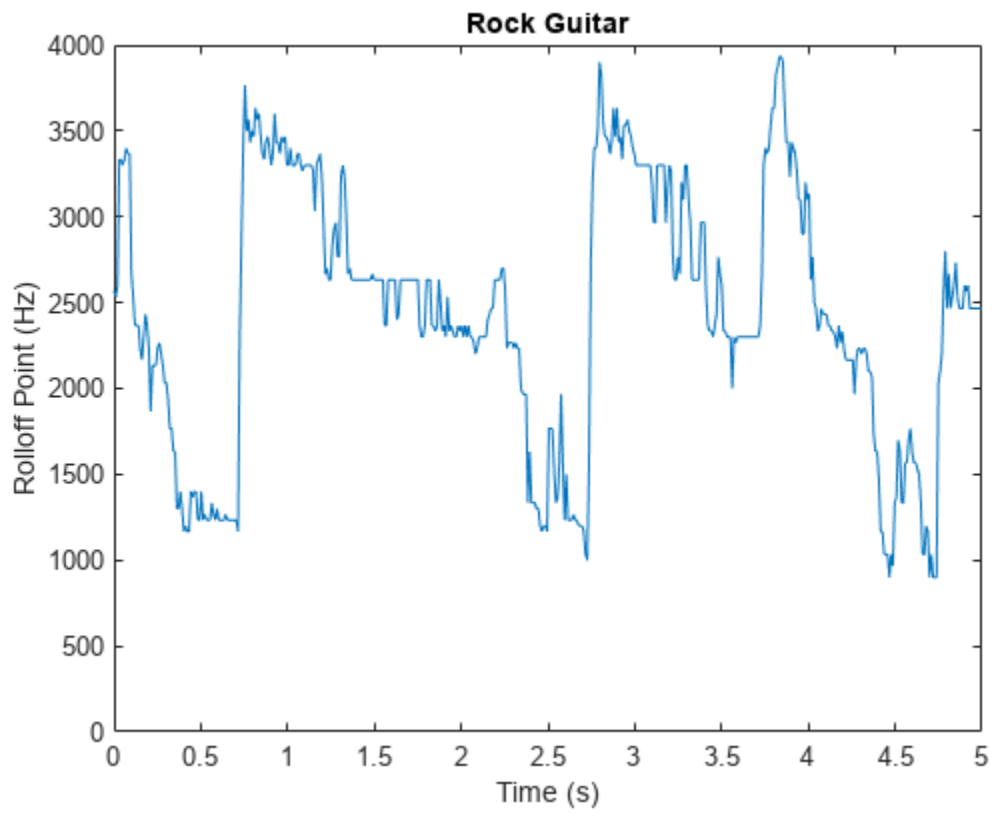
r1 = spectralRolloffPoint(speech,fs1);
r2 = spectralRolloffPoint(electricGuitar,fs2);
r3 = spectralRolloffPoint(acousticGuitar,fs3);
r4 = spectralRolloffPoint(acousticScene,fs4);

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));

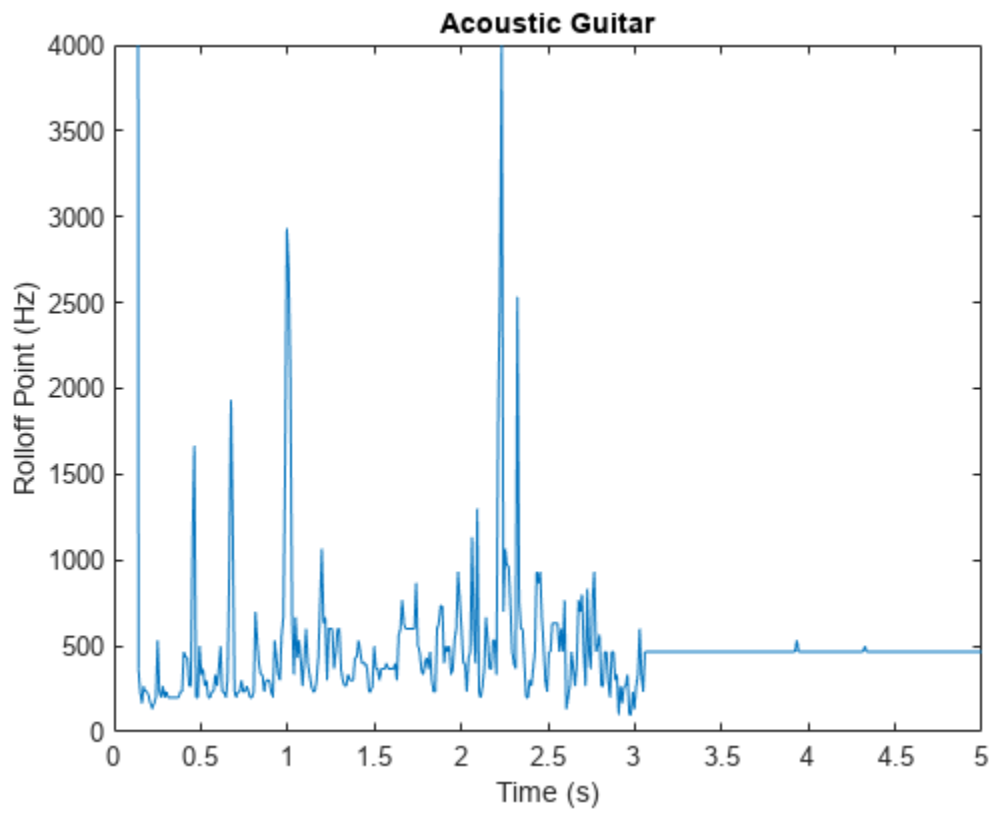
figure
plot(t1,r1)
title('Speech')
ylabel('Rolloff Point (Hz)')
xlabel('Time (s)')
axis([0 5 0 4000])
```



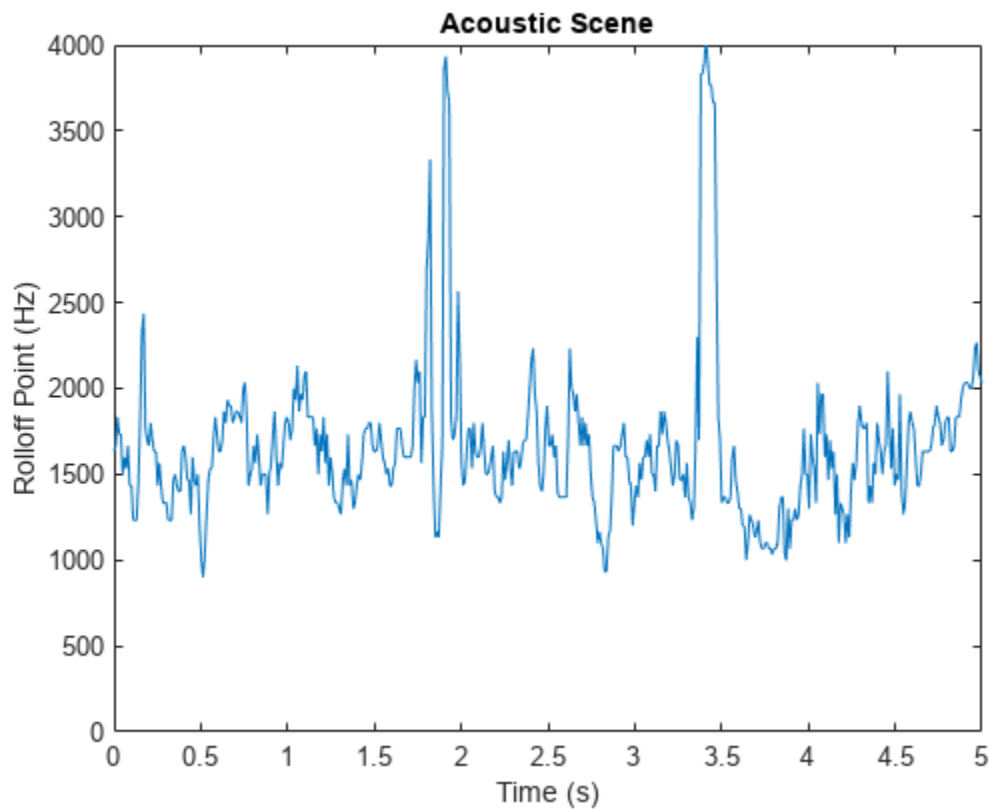
```
figure
plot(t2,r2)
title('Rock Guitar')
ylabel('Rolloff Point (Hz)')
xlabel('Time (s)')
axis([0 5 0 4000])
```



```
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. Iqbal, 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., V.t. 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.

