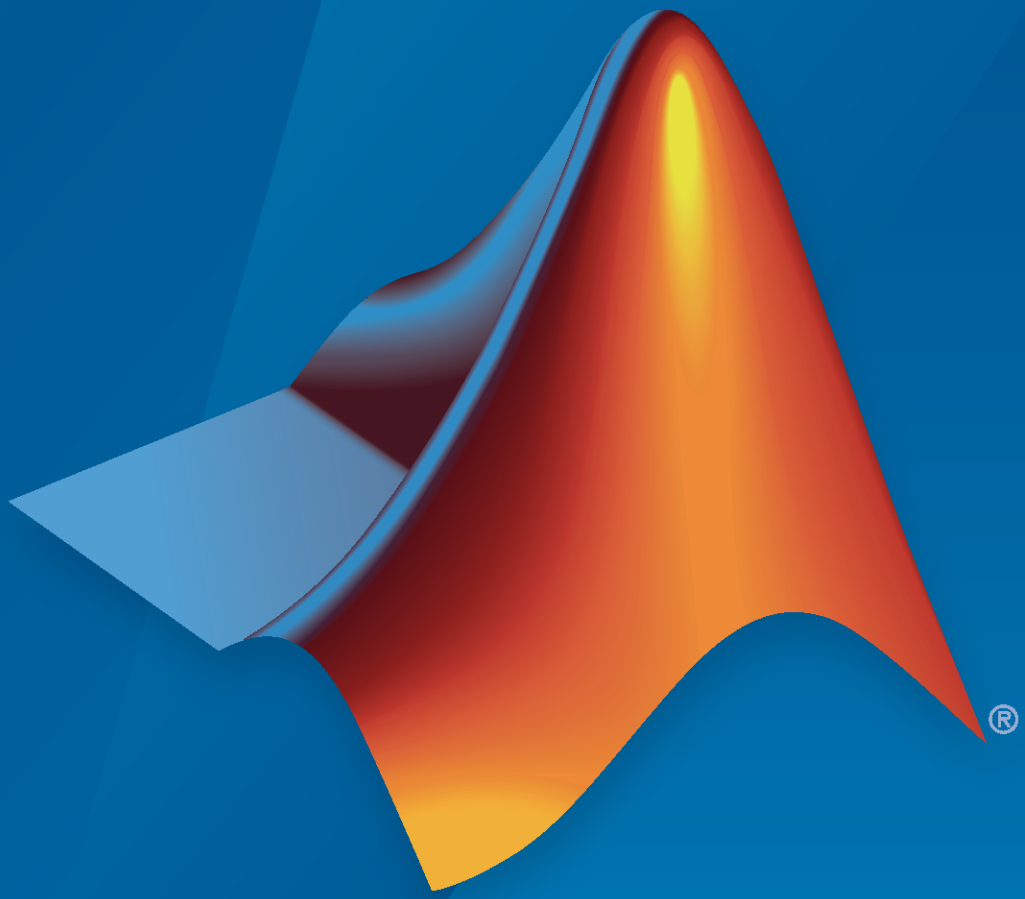
# Signal Processing Toolbox™

User's Guide



MATLAB®

R2020b

MathWorks®

# 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

*Signal Processing Toolbox™ User's Guide*

© COPYRIGHT 1988–2020 by The MathWorks, Inc.

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

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

**Trademarks**

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

**Patents**

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

**Revision History**

| | | |
|---|---|---|
| 1988 | First printing | New |
| November 1997 | Second printing | Revised |
| January 1998 | Third printing | Revised |
| September 2000 | Fourth printing | Revised for Version 5.0 (Release 12) |
| July 2002 | Fifth printing | Revised for Version 6.0 (Release 13) |
| December 2002 | Online only | Revised for Version 6.1 (Release 13+) |
| June 2004 | Online only | Revised for Version 6.2 (Release 14) |
| October 2004 | Online only | Revised for Version 6.2.1 (Release 14SP1) |
| March 2005 | Online only | Revised for Version 6.2.1 (Release 14SP2) |
| September 2005 | Online only | Revised for Version 6.4 (Release 14SP3) |
| March 2006 | Sixth printing | Revised for Version 6.5 (Release 2006a) |
| September 2006 | Online only | Revised for Version 6.6 (Release 2006b) |
| March 2007 | Online only | Revised for Version 6.7 (Release 2007a) |
| September 2007 | Online only | Revised for Version 6.8 (Release 2007b) |
| March 2008 | Online only | Revised for Version 6.9 (Release 2008a) |
| October 2008 | Online only | Revised for Version 6.10 (Release 2008b) |
| March 2009 | Online only | Revised for Version 6.11 (Release 2009a) |
| September 2009 | Online only | Revised for Version 6.12 (Release 2009b) |
| March 2010 | Online only | Revised for Version 6.13 (Release 2010a) |
| September 2010 | Online only | Revised for Version 6.14 (Release 2010b) |
| April 2011 | Online only | Revised for Version 6.15 (Release 2011a) |
| September 2011 | Online only | Revised for Version 6.16 (Release 2011b) |
| March 2012 | Online only | Revised for Version 6.17 (Release 2012a) |
| September 2012 | Online only | Revised for Version 6.18 (Release 2012b) |
| March 2013 | Online only | Revised for Version 6.19 (Release 2013a) |
| September 2013 | Online only | Revised for Version 6.20 (Release 2013b) |
| March 2014 | Online only | Revised for Version 6.21 (Release 2014a) |
| October 2014 | Online only | Revised for Version 6.22 (Release 2014b) |
| March 2015 | Online only | Revised for Version 7.0 (Release 2015a) |
| September 2015 | Online only | Revised for Version 7.1 (Release 2015b) |
| March 2016 | Online only | Revised for Version 7.2 (Release 2016a) |
| September 2016 | Online only | Revised for Version 7.3 (Release 2016b) |
| March 2017 | Online only | Revised for Version 7.4 (Release 2017a) |
| September 2017 | Online only | Revised for Version 7.5 (Release 2017b) |
| March 2018 | Online only | Revised for Version 8.0 (Release 2018a) |
| September 2018 | Online only | Revised for Version 8.1 (Release 2018b) |
| March 2019 | Online only | Revised for Version 8.2 (Release 2019a) |
| September 2019 | Online only | Revised for Version 8.3 (Release 2019b) |
| March 2020 | Online only | Revised for Version 8.4 (Release 2020a) |
| September 2020 | Online only | Revised for Version 8.5 (Release 2020b) |

# Contents

## 1 Filtering, Linear Systems and Transforms Overview


**Filter Implementation** . . . . . **1-2**
- Convolution and Filtering . . . . . **1-2**
- Filters and Transfer Functions . . . . . **1-2**
- Filtering with the filter Function . . . . . **1-3**

**The filter Function** . . . . . **1-5**

**Multirate Filter Bank Implementation** . . . . . **1-6**

**Frequency Domain Filter Implementation** . . . . . **1-7**

**Anti-Causal, Zero-Phase Filter Implementation** . . . . . **1-8**

**Impulse Response** . . . . . **1-10**

**Frequency Response** . . . . . **1-13**
- Digital Domain . . . . . **1-13**
- Analog Domain . . . . . **1-18**

**Phase Response** . . . . . **1-21**

**Group Delay and Phase Delay** . . . . . **1-25**

**Zero-Pole Analysis** . . . . . **1-28**

**Discrete-Time System Models** . . . . . **1-32**
- Transfer Function . . . . . **1-32**
- Zero-Pole-Gain . . . . . **1-32**
- State Space . . . . . **1-33**
- Partial Fraction Expansion (Residue Form) . . . . . **1-33**
- Second-Order Sections (SOS) . . . . . **1-34**
- Lattice Structure . . . . . **1-35**
- Convolution Matrix . . . . . **1-36**

**Continuous-Time System Models** . . . . . **1-38**

**Linear System Transformations** . . . . . **1-39**

**Discrete Fourier Transform** . . . . . **1-40**

## 2 Filter Design and Implementation


**Filter Requirements and Specification** . . . . . **2-2**

**IIR Filter Design** . . . . . **2-4**
IIR vs. FIR Filters . . . . . **2-4**
Classical IIR Filters . . . . . **2-4**
Other IIR Filters . . . . . **2-4**
IIR Filter Method Summary . . . . . **2-4**
Classical IIR Filter Design Using Analog Prototyping . . . . . **2-5**
Comparison of Classical IIR Filter Types . . . . . **2-7**

**FIR Filter Design** . . . . . **2-16**
FIR vs. IIR Filters . . . . . **2-16**
FIR Filter Summary . . . . . **2-16**
Linear Phase Filters . . . . . **2-17**
Windowing Method . . . . . **2-17**
Multiband FIR Filter Design with Transition Bands . . . . . **2-20**
Constrained Least Squares FIR Filter Design . . . . . **2-24**
Arbitrary-Response Filter Design . . . . . **2-28**

**Special Topics in IIR Filter Design** . . . . . **2-33**
Classic IIR Filter Design . . . . . **2-33**
Analog Prototype Design . . . . . **2-33**
Frequency Transformation . . . . . **2-34**
Filter Discretization . . . . . **2-35**

**Filtering Data With Signal Processing Toolbox Software** . . . . . **2-39**

**Selected Bibliography** . . . . . **2-55**


## 3 Designing a Filter in fdesign — Process Overview


**Process Flow Diagram and Filter Design Methodology** . . . . . **3-2**
Exploring the Process Flow Diagram . . . . . **3-2**
Selecting a Response . . . . . **3-4**
Selecting a Specification . . . . . **3-4**
Selecting an Algorithm . . . . . **3-5**
Customizing the Algorithm . . . . . **3-6**
Designing the Filter . . . . . **3-6**
Design Analysis . . . . . **3-7**
Realize or Apply the Filter to Input Data . . . . . **3-7**

## 4 Designing a Filter in the Filter Builder GUI


**Filter Builder Design Process** . . . . . **4-2**
Introduction to Filter Builder . . . . . **4-2**
Design a Filter Using Filter Builder . . . . . **4-2**
Select a Response . . . . . **4-2**
Select a Specification . . . . . **4-4**
Select an Algorithm . . . . . **4-5**
Customize the Algorithm . . . . . **4-5**
Analyze the Design . . . . . **4-6**
Realize or Apply the Filter to Input Data . . . . . **4-7**

**Designing a FIR Filter Using filterBuilder** . . . . . **4-9**
FIR Filter Design . . . . . **4-9**

**Compensate for Delay and Distortion Introduced by Filters** . . . . . **4-12**

**Comparison of Analog IIR Lowpass Filters** . . . . . **4-19**

**Frequency Response of Lowpass Bessel Filter** . . . . . **4-21**

**Speaker Crossover Filters** . . . . . **4-23**


## 5 Filter Designer: A Filter Design and Analysis App


**Filter Design Methods** . . . . . **5-2**
Advanced Filter Design Methods . . . . . **5-2**

**Using the Filter Designer App** . . . . . **5-4**

**Analyzing Filter Responses** . . . . . **5-5**

**Filter Designer App Panels** . . . . . **5-6**

**Getting Help** . . . . . **5-7**

**Getting Started with Filter Designer** . . . . . **5-8**
Choosing a Response Type . . . . . **5-9**
Choosing a Filter Design Method . . . . . **5-9**
Setting the Filter Design Specifications . . . . . **5-9**
Computing the Filter Coefficients . . . . . **5-11**
Analyzing the Filter . . . . . **5-11**
Editing the Filter Using the Pole-Zero Editor . . . . . **5-13**
Converting the Filter Structure . . . . . **5-14**
Exporting a Filter Design . . . . . **5-15**
Generating a C Header File . . . . . **5-18**
Generating MATLAB Code . . . . . **5-18**
Managing Filters in the Current Session . . . . . **5-19**
Saving and Opening Filter Design Sessions . . . . . **5-20**

**Importing a Filter Design** . . . . . **5-21**
Import Filter Panel . . . . . **5-21**
Filter Structures . . . . . **5-21**

**FIR Bandpass Filter with Asymmetric Attenuation** . . . . . **5-24**

**Arbitrary Magnitude Filter** . . . . . **5-26**

## 6 Filter Visualization Tool

**Modifying the Axes** . . . . . **6-2**

**Modifying the Plot** . . . . . **6-4**

**Controlling FVTool from the MATLAB Command Line** . . . . . **6-6**

## 7 Statistical Signal Processing

**Correlation and Covariance** . . . . . **7-2**
Background Information . . . . . **7-2**
Using xcorr and xcov Functions . . . . . **7-2**
Bias and Normalization . . . . . **7-3**
Multiple Channels . . . . . **7-3**

**Spectral Analysis** . . . . . **7-5**
Background Information . . . . . **7-5**
Spectral Estimation Method . . . . . **7-6**

**Nonparametric Methods** . . . . . **7-8**
Periodogram . . . . . **7-8**
Performance of the Periodogram . . . . . **7-9**
The Modified Periodogram . . . . . **7-15**
Welch's Method . . . . . **7-17**
Bias and Normalization in Welch's Method . . . . . **7-20**
Multitaper Method . . . . . **7-20**
Cross-Spectral Density Function . . . . . **7-23**
Transfer Function Estimate . . . . . **7-24**
Coherence Function . . . . . **7-25**

**Parametric Methods** . . . . . **7-27**
Yule-Walker AR Method . . . . . **7-28**
Burg Method . . . . . **7-30**
Covariance and Modified Covariance Methods . . . . . **7-34**

**MUSIC and Eigenvector Analysis Methods** . . . . . **7-37**
Eigenanalysis Overview . . . . . **7-37**
Frequency Estimator Functions . . . . . **7-37**

**Selected Bibliography** . . . . . **7-39**

# 8 Special Topics

**Windows** . . . . . **8-2**
Why Use Windows? . . . . . **8-2**
Available Window Functions . . . . . **8-2**
Graphical User Interface Tools . . . . . **8-2**
Basic Shapes . . . . . **8-3**

**Get Started with Window Designer** . . . . . **8-6**
Window Parameters . . . . . **8-7**
Window Designer Menus . . . . . **8-7**

**Generalized Cosine Windows** . . . . . **8-9**

**Kaiser Window** . . . . . **8-11**
Kaiser Windows in FIR Design . . . . . **8-15**

**Chebyshev Window** . . . . . **8-17**

**Parametric Modeling** . . . . . **8-18**
What is Parametric Modeling . . . . . **8-18**
Available Parametric Modeling Functions . . . . . **8-18**
Time-Domain Based Modeling . . . . . **8-19**
Frequency-Domain Based Modeling . . . . . **8-21**

**Resampling** . . . . . **8-24**
resample Function . . . . . **8-24**
decimate and interp Functions . . . . . **8-25**
upfirdn Function . . . . . **8-25**
spline Function . . . . . **8-25**

**Cepstrum Analysis** . . . . . **8-26**

**FFT-Based Time-Frequency Analysis** . . . . . **8-29**

**Cross-Spectrogram of Complex Signals** . . . . . **8-31**

**Median Filtering** . . . . . **8-33**

**Communications Applications** . . . . . **8-34**
Modulation . . . . . **8-34**
Demodulation . . . . . **8-35**
Voltage Controlled Oscillator . . . . . **8-36**

**Deconvolution** . . . . . **8-38**

**Chirp Z-Transform** . . . . . **8-39**

**Discrete Cosine Transform** . . . . . **8-41**

**Hilbert Transform** . . . . . **8-44**

**Walsh-Hadamard Transform** . . . . . **8-46**

**Walsh-Hadamard Transform for Spectral Analysis and Compression of ECG Signals** . . . . . **8-48**

**Eliminate Outliers Using Hampel Identifier** . . . . . **8-51**

**Selected Bibliography** . . . . . **8-53**

## 9 SPTool: A Signal Processing GUI Suite

**SPTool: An Interactive Signal Processing Environment** . . . . . **9-2**
SPTool Overview . . . . . **9-2**
SPTool Data Structures . . . . . **9-2**

**Opening SPTool** . . . . . **9-3**

**Getting Context-Sensitive Help** . . . . . **9-4**

**Signal Browser** . . . . . **9-5**
Overview of the Signal Browser . . . . . **9-5**
Opening the Signal Browser . . . . . **9-5**

**Filter Visualization Tool** . . . . . **9-7**
Connection between FVTool and SPTool . . . . . **9-7**
Opening the Filter Visualization Tool . . . . . **9-7**
Analysis Parameters . . . . . **9-8**

**Spectrum Viewer** . . . . . **9-9**
Spectrum Viewer Overview . . . . . **9-9**
Opening the Spectrum Viewer . . . . . **9-9**

**Filtering and Analysis of Noise** . . . . . **9-11**
Overview . . . . . **9-11**
Importing a Signal into SPTool . . . . . **9-11**
Designing a Filter . . . . . **9-12**
Applying a Filter to a Signal . . . . . **9-14**
Analyzing a Signal . . . . . **9-15**
Spectral Analysis in the Spectrum Viewer . . . . . **9-17**

**Exporting Signals, Filters, and Spectra** . . . . . **9-19**
Opening the Export Dialog Box . . . . . **9-19**
Exporting a Filter to the MATLAB Workspace . . . . . **9-19**

**Accessing Filter Parameters** . . . . . **9-20**
Accessing Filter Parameters in a Saved Filter . . . . . **9-20**
Accessing Parameters in a Saved Spectrum . . . . . **9-20**

**Importing Filters and Spectra** . . . . . **9-22**
Similarities to Other Procedures . . . . . **9-22**
Importing Filters . . . . . **9-22**
Importing Spectra . . . . . **9-23**

**Loading Variables from the Disk** . . . . . **9-25**

**Saving and Loading Sessions** . . . . . **9-26**
SPTool Sessions . . . . . **9-26**
Filter Formats . . . . . **9-26**

**Selecting Signals, Filters, and Spectra** . . . . . **9-28**

**Editing Signals, Filters, or Spectra** . . . . . **9-29**

**Making Signal Measurements with Markers** . . . . . **9-30**

**Setting Preferences** . . . . . **9-32**
Overview of Setting Preferences . . . . . **9-32**
Summary of Settable Preferences . . . . . **9-32**

## 10 Code Generation from MATLAB Support in Signal Processing Toolbox

**List of Signal Processing Toolbox Functions that Support Code Generation** . . . . . **10-2**

**Specifying Inputs in Code Generation from MATLAB** . . . . . **10-9**
Defining Input Size and Type . . . . . **10-9**
Inputs must be Constants . . . . . **10-10**

**Apply Lowpass Filter to Input Signal** . . . . . **10-12**

**Zero-Phase Filtering** . . . . . **10-14**

**Compute Modified Periodogram Using Generated C Code** . . . . . **10-16**

## 11 Convolution and Correlation

**Linear and Circular Convolution** . . . . . **11-2**

**Confidence Intervals for Sample Autocorrelation** . . . . . **11-4**

**Residual Analysis with Autocorrelation** . . . . . **11-6**

**Autocorrelation of Moving Average Process** . . . . . **11-12**

**Cross-Correlation of Two Moving Average Processes** . . . . . **11-15**

**Cross-Correlation of Delayed Signal in Noise** . . . . . **11-17**

**Cross-Correlation of Phase-Lagged Sine Wave** . . . . . **11-19**

## 12 Multirate Signal Processing

**Downsampling — Signal Phases** . . . . . **12-2**

**Downsampling — Aliasing** . . . . . **12-5**

**Filtering Before Downsampling** . . . . . **12-9**

**Upsampling — Imaging Artifacts** . . . . . **12-11**

**Filtering After Upsampling — Interpolation** . . . . . **12-13**

**Simulate a Sample-and-Hold System** . . . . . **12-15**

**Change Signal Sample Rate** . . . . . **12-20**

## 13 Spectral Analysis

**Power Spectral Density Estimates Using FFT** . . . . . **13-2**

**Bias and Variability in the Periodogram** . . . . . **13-9**

**Cross Spectrum and Magnitude-Squared Coherence** . . . . . **13-17**

**Amplitude Estimation and Zero Padding** . . . . . **13-20**

**Significance Testing for Periodic Component** . . . . . **13-23**

**Frequency Estimation by Subspace Methods** . . . . . **13-25**

**Frequency-Domain Linear Regression** . . . . . **13-27**

**Measure Total Harmonic Distortion** . . . . . **13-36**

**Measure Mean Frequency, Power, Bandwidth** . . . . . **13-38**

**Periodogram of Data Set with Missing Samples** . . . . . **13-43**

**Welch Spectrum Estimates** . . . . . **13-46**

## 14 Time-Frequency Analysis


**Time-Frequency Gallery** . . . . . **14-2**
Short-Time Fourier Transform (Spectrogram) . . . . . **14-3**
Continuous Wavelet Transform (Scalogram) . . . . . **14-7**
Wigner-Ville Distribution . . . . . **14-9**
Reassignment and Synchrosqueezing . . . . . **14-11**
Constant-Q Gabor Transform . . . . . **14-17**
Empirical Mode Decomposition and Hilbert-Huang Transform . . . . . **14-18**


## 15 Linear Prediction


**Prediction Polynomial** . . . . . **15-2**

**Formant Estimation with LPC Coefficients** . . . . . **15-4**

**AR Order Selection with Partial Autocorrelation Sequence** . . . . . **15-7**


## 16 Transforms


**Complex Cepstrum — Fundamental Frequency Estimation** . . . . . **16-2**

**Analytic Signal for Cosine** . . . . . **16-5**

**Envelope Extraction** . . . . . **16-7**

**Analytic Signal and Hilbert Transform** . . . . . **16-13**

**Hilbert Transform and Instantaneous Frequency** . . . . . **16-18**

**Detect Closely Spaced Sinusoids** . . . . . **16-25**

**Instantaneous Frequency of Complex Chirp** . . . . . **16-32**

**Single-Sideband Amplitude Modulation** . . . . . **16-35**

**DCT for Speech Signal Compression** . . . . . **16-42**

## 17 Signal Generation


**Display Time-Domain Data in Signal Browser** . . . . . **17-2**
Import and Display Signals . . . . . **17-2**
Configure the Signal Browser Properties . . . . . **17-5**
Modify the Signal Browser Display . . . . . **17-7**
Inspect Your Data (Scaling the Axes and Zooming) . . . . . **17-8**


## 18 Signal Measurement


**RMS Value of Periodic Waveforms** . . . . . **18-2**

**Slew Rate of Triangular Waveform** . . . . . **18-5**

**Duty Cycle of Rectangular Pulse Waveform** . . . . . **18-8**

**Estimate State for Digital Clock** . . . . . **18-11**

**Calculate Settling Time with Signal Browser** . . . . . **18-14**

**Find Peak Amplitudes in Signal Browser** . . . . . **18-16**

**Distortion Measurements** . . . . . **18-18**

**Prominence** . . . . . **18-22**

**Determine Peak Widths** . . . . . **18-24**


## 19 Spectrum Object to Function Replacement


**Nonparametric Spectrum Object to Function Replacement** . . . . . **19-2**
Periodogram PSD Object to Function Replacement Syntax . . . . . **19-2**
Periodogram MSSPECTRUM Object to Function Replacement Syntax . . . **19-3**
Welch PSD Object to Function Replacement Syntax . . . . . **19-4**
Welch MSSPECTRUM Object to Function Replacement Syntax . . . . . **19-5**
Multitaper PSD Object to Function Replacement Syntax . . . . . **19-7**

**Autoregressive PSD Object to Function Replacement Syntax** . . . . . **19-9**

**Subspace Pseudospectrum Object to Function Replacement Syntax** . . **19-10**

# 20 Vibration Analysis

**Modal Parameters of MIMO System** . . . . . **20-2**

**Compute and Display Order-RPM Map** . . . . . **20-5**

**MIMO Stabilization Diagram** . . . . . **20-8**

**Modal Analysis of Identified Models** . . . . . **20-12**

# 21 Signal Analyzer App

**Using Signal Analyzer App** . . . . . **21-2**
App Workflow . . . . . **21-2**
Example: Extract Regions of Interest from Whale Song . . . . . **21-2**

**Select Signals to Analyze** . . . . . **21-9**
Select Signals from the Workspace Browser . . . . . **21-9**
Filter Signals in the Signal Table . . . . . **21-10**
Next Step . . . . . **21-12**

**Preprocess Signals** . . . . . **21-13**
Duplicate and Rename Signals . . . . . **21-13**
Filter Signals . . . . . **21-14**
Smooth Signals . . . . . **21-14**
Resample Signals . . . . . **21-14**
Detrend Signals . . . . . **21-15**
Compute Signal Envelopes . . . . . **21-15**
Add Custom Preprocessing Functions . . . . . **21-15**
Previous Step . . . . . **21-18**
Next Step . . . . . **21-18**

**Explore Signals** . . . . . **21-19**
Plot Signals . . . . . **21-19**
View Signals on Multiple Plots . . . . . **21-19**
Move Signals Between Displays . . . . . **21-19**
Visualize Signal Spectra . . . . . **21-19**
Visualize Persistence Spectra . . . . . **21-20**
Visualize Signal Spectrograms . . . . . **21-20**
Visualize Signal Scalograms . . . . . **21-21**
Zoom and Pan Through Signals . . . . . **21-22**
Edit Time Information and Link Displays in Time . . . . . **21-22**
Measure Signal, Spectrum, and Time-Frequency Data . . . . . **21-23**
Extract Signal Regions of Interest . . . . . **21-24**
Previous Step . . . . . **21-24**
Next Step . . . . . **21-24**

**Share Analysis** . . . . . **21-26**
Copy Displays . . . . . **21-26**

Export Signals . . . . . 21-26
Generate MATLAB Scripts and Functions . . . . . 21-28
Save and Load Signal Analyzer Sessions . . . . . 21-29
Previous Step . . . . . 21-29

**Find Delay Between Correlated Signals** . . . . . 21-30

**Plot Signals from the Command Line** . . . . . 21-35

**Resolve Tones by Varying Window Leakage** . . . . . 21-39

**Resolve Tones by Varying Window Leakage** . . . . . 21-43

**Analyze Signals with Inherent Time Information** . . . . . 21-45

**Spectrogram View of Dial Tone Signal** . . . . . 21-48

**Find Interference Using Persistence Spectrum** . . . . . 21-51

**Extract Regions of Interest from Whale Song** . . . . . 21-56

**Modulation and Demodulation Using Complex Envelope** . . . . . 21-61

**Find and Track Ridges Using Reassigned Spectrogram** . . . . . 21-70

**Scalogram of Hyperbolic Chirp** . . . . . 21-75

**Extract Voices from Music Signal** . . . . . 21-78

**Resample and Filter a Nonuniformly Sampled Signal** . . . . . 21-84

**Declip Saturated Signals Using Your Own Function** . . . . . 21-91

**Compute Envelope Spectrum of Vibration Signal** . . . . . 21-96

**Edit Sample Rate and Other Time Information** . . . . . 21-103

**Data Types Supported by Signal Analyzer** . . . . . 21-106
Numeric Data . . . . . 21-106
MATLAB Timetables . . . . . 21-106
timeseries Objects . . . . . 21-106
Nonuniformly Sampled Signals . . . . . 21-107
Labeled Signal Sets . . . . . 21-107

**Spectrum Computation in Signal Analyzer** . . . . . 21-109
Spectral Windowing . . . . . 21-109
Parameter and Algorithm Selection . . . . . 21-110
Zooming . . . . . 21-111

**Persistence Spectrum in Signal Analyzer** . . . . . 21-113

**Spectrogram Computation in Signal Analyzer** . . . . . 21-115
Divide Signal into Segments . . . . . 21-116
Window the Segments and Compute Spectra . . . . . 21-118

Display Spectrum Power . . . . . 21-119

**Scalogram Computation in Signal Analyzer** . . . . . 21-120
Divide the Signal into Segments . . . . . 21-120
Compute the Continuous Wavelet Transform . . . . . 21-121
Display the Scalogram . . . . . 21-122

**Keyboard Shortcuts for Signal Analyzer** . . . . . 21-124
General Actions . . . . . 21-124
Multichannel Signals . . . . . 21-124
Zooming . . . . . 21-124
Data Cursors . . . . . 21-124

**Signal Analyzer Tips and Limitations** . . . . . 21-126
Select Signals to Analyze . . . . . 21-126
Preprocess Signals . . . . . 21-127
Explore Signals . . . . . 21-127
Share or Reuse Analysis . . . . . 21-128
Troubleshooting . . . . . 21-128

**Customize Signal Analyzer** . . . . . 21-130
Specify Line Color and Style . . . . . 21-130
Add or Remove Columns in the Signal Table . . . . . 21-130
Modify Signal Analyzer Displays . . . . . 21-131
Signal Analyzer Preferences . . . . . 21-133

## 22 Signal Labeler

**Using Signal Labeler App** . . . . . 22-2
App Workflow . . . . . 22-2
Example: Label Points and Regions of Interest in Signal . . . . . 22-2

**Import Data into Signal Labeler** . . . . . 22-6
Supported Signal Types . . . . . 22-6
Specify Time Information . . . . . 22-7
Import Signals from the MATLAB Workspace . . . . . 22-8
Import Signals from Files . . . . . 22-10

**Create or Import Signal Label Definitions** . . . . . 22-13
Import Signal Label Definitions . . . . . 22-13
Create Label Definitions . . . . . 22-14
Create Sublabel Definitions . . . . . 22-15
Edit Label or Sublabel Definitions . . . . . 22-15
Delete Label or Sublabel Definitions . . . . . 22-15

**Label Signals Interactively or Automatically** . . . . . 22-17
Fast Navigation Mode . . . . . 22-17
Label Signals Manually . . . . . 22-17
Label Signals Automatically . . . . . 22-19
Label Signal Peaks Automatically Using Peak Labeler . . . . . 22-22

**Custom Labeling Functions** . . . . . **22-25**
Create Custom Labeling Functions . . . . . **22-25**
Add Custom Labeling Functions to the Gallery . . . . . **22-28**
Manage Custom Labeling Functions in Gallery . . . . . **22-28**

**Customize Labeling View** . . . . . **22-30**
Visualize Signal Spectra and Spectrograms . . . . . **22-30**
Use Spectrogram to Aid Labeling . . . . . **22-31**

**Export Labeled Signal Sets and Signal Label Definitions** . . . . . **22-36**
Export Label Definitions . . . . . **22-36**
Export Labeled Signal Sets . . . . . **22-36**

**Signal Labeler Usage Tips** . . . . . **22-38**
Keyboard Shortcuts . . . . . **22-38**
Troubleshooting . . . . . **22-39**

**Label Signal Attributes, Regions of Interest, and Points** . . . . . **22-40**

**Examine Labeled Signal Set** . . . . . **22-46**

**Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network** . . . . . **22-51**

**Label Spoken Words in Audio Signals Using External API** . . . . . **22-60**

**Choose an App to Label Ground Truth Data** . . . . . **22-65**

## 23 Common Applications

**Create Uniform and Nonuniform Time Vectors** . . . . . **23-2**

**Remove Trends from Data** . . . . . **23-4**

**Remove the 60 Hz Hum from a Signal** . . . . . **23-6**

**Remove Spikes from a Signal** . . . . . **23-10**

**Process a Signal with Missing Samples** . . . . . **23-13**

**Reconstruct a Signal from Irregularly Sampled Data** . . . . . **23-18**

**Align Signals with Different Start Times** . . . . . **23-21**

**Align Signals Using Cross-Correlation** . . . . . **23-25**

**Align Two Simple Signals** . . . . . **23-29**

**Find Peaks in Data** . . . . . **23-33**

**Find a Signal in a Measurement** . . . . . **23-37**

**Find Periodicity Using Autocorrelation** . . . . . **23-45**

**Extract Features of a Clock Signal** . . . . . **23-48**

**Find Periodicity in a Categorical Time Series** . . . . . **23-54**

**Compensate for the Delay Introduced by an FIR Filter** . . . . . **23-59**

**Compensate for the Delay Introduced by an IIR Filter** . . . . . **23-63**

**Take Derivatives of a Signal** . . . . . **23-67**

**Find Periodicity Using Frequency Analysis** . . . . . **23-73**

**Detect a Distorted Signal in Noise** . . . . . **23-75**

**Measure the Power of a Signal** . . . . . **23-81**

**Compare the Frequency Content of Two Signals** . . . . . **23-85**

**Detect Periodicity in a Signal with Missing Samples** . . . . . **23-88**

**Echo Cancelation** . . . . . **23-91**

**Cross-Correlation with Multichannel Input** . . . . . **23-95**

**Autocorrelation Function of Exponential Sequence** . . . . . **23-99**

**Cross-Correlation of Two Exponential Sequences** . . . . . **23-104**

## 24 Featured Examples

**Signal Generation and Visualization** . . . . . **24-2**

**Signal Smoothing** . . . . . **24-10**

**Reconstructing Missing Data** . . . . . **24-27**

**Resampling Uniformly Sampled Signals** . . . . . **24-38**

**Resampling Nonuniformly Sampled Signals** . . . . . **24-46**

**Peak Analysis** . . . . . **24-60**

**Measuring Signal Similarities** . . . . . **24-71**

**Measurement of Pulse and Transition Characteristics** . . . . . **24-82**

**Analyzing Harmonic Distortion** . . . . . **24-91**

**Spurious-Free Dynamic Range (SFDR) Measurement** . . . . . **24-104**

**Extracting Classification Features from Physiological Signals** . . . . . **24-113**

**Detecting Outbreaks and Significant Changes in Signals** . . . . . **24-119**

**Finding a Signal in Data** . . . . . **24-132**

**Filter Design Gallery** . . . . . **24-141**

**Practical Introduction to Digital Filter Design** . . . . . **24-161**

**Practical Introduction to Digital Filtering** . . . . . **24-174**

**Introduction to Filter Designer** . . . . . **24-192**

**Filter Analysis using FVTool** . . . . . **24-202**

**FIR Gaussian Pulse-shaping Filter Design** . . . . . **24-214**

**Generating Guitar Chords Using the Karplus-Strong Algorithm** . . . . **24-221**

**DFT Estimation with the Goertzel Algorithm** . . . . . **24-227**

**Discrete Walsh-Hadamard Transform** . . . . . **24-231**

**Single Sideband Modulation via the Hilbert Transform** . . . . . **24-239**

**Practical Introduction to Frequency-Domain Analysis** . . . . . **24-254**

**Practical Introduction to Time-Frequency Analysis** . . . . . **24-267**

**Measure Power of Deterministic Periodic Signals** . . . . . **24-289**

**Spectral Analysis of Nonuniformly Sampled Signals** . . . . . **24-302**

**Linear Prediction and Autoregressive Modeling** . . . . . **24-309**

**Classify ECG Signals Using Long Short-Term Memory Networks** . . . . **24-313**

**Waveform Segmentation Using Deep Learning** . . . . . **24-330**

**Deploy Signal Segmentation Deep Network on Raspberry Pi** . . . . . **24-350**

**Iterative Approach for Creating Labeled Signal Sets with Reduced Human Effort** . . . . . **24-360**

**Generate Synthetic Signals Using Conditional Generative Adversarial Network** . . . . . **24-371**

**Classify Time Series Using Wavelet Analysis and Deep Learning** . . . . **24-386**

**Denoise Speech Using Deep Learning Networks** . . . . . **24-404**

**Order Analysis of a Vibration Signal** . . . . . **24-424**

**Vibration Analysis of Rotating Machinery** . . . . . **24-435**

**Modal Analysis of a Simulated System and a Wind Turbine Blade** . . . **24-456**

**Accelerating Correlation with GPUs** . . . . . **24-474**

# Filtering, Linear Systems and Transforms Overview


- “Filter Implementation” on page 1-2
- “The filter Function” on page 1-5
- “Multirate Filter Bank Implementation” on page 1-6
- “Frequency Domain Filter Implementation” on page 1-7
- “Anti-Causal, Zero-Phase Filter Implementation” on page 1-8
- “Impulse Response” on page 1-10
- “Frequency Response” on page 1-13
- “Phase Response” on page 1-21
- “Group Delay and Phase Delay” on page 1-25
- “Zero-Pole Analysis” on page 1-28
- “Discrete-Time System Models” on page 1-32
- “Continuous-Time System Models” on page 1-38
- “Linear System Transformations” on page 1-39
- “Discrete Fourier Transform” on page 1-40

# Filter Implementation

| In this section... |
| --- |
| “Convolution and Filtering” on page 1-2 |
| “Filters and Transfer Functions” on page 1-2 |
| “Filtering with the filter Function” on page 1-3 |

## Convolution and Filtering

The mathematical foundation of filtering is convolution. For a finite impulse response (FIR) filter, the output $y(k)$ of a filtering operation is the convolution of the input signal $x(k)$ with the impulse response $h(k)$:

$$y(k) = \sum_{l=-\infty}^{\infty} h(l)\, x(k-l)\,.$$

If the input signal is also of finite length, you can implement the filtering operation using the MATLAB® `conv` function. For example, to filter a five-sample random vector with a third-order averaging filter, you can store $x(k)$ in a vector `x`, $h(k)$ in a vector `h`, and convolve the two:

```
x = randn(5,1);
h = [1 1 1 1]/4;   % A third-order filter has length 4
y = conv(h,x)
```

```
y =
   -0.3375
    0.4213
    0.6026
    0.5868
    1.1030
    0.3443
    0.1629
    0.1787
```

The length of `y` is one less than the sum of the lengths of `x` and `h`.

## Filters and Transfer Functions

The transfer function of a filter is the Z-transform of its impulse response. For an FIR filter, the Z-transform of the output $y$, $Y(z)$, is the product of the transfer function and $X(z)$, the Z-transform of the input $x$:

$$Y(z) = H(z)X(z) = \left(h(1) + h(2)z^{-1} + \cdots + h(n+1)z^{-n}\right)X(z)\,.$$

The polynomial coefficients $h(1)$, $h(2)$, ..., $h(n+1)$ correspond to the coefficients of the impulse response of an $n$th-order filter.

**Note** The filter coefficient indices run from 1 to $(n+1)$, rather than from 0 to $n$. This reflects the standard indexing scheme used for MATLAB vectors.

FIR filters are also called all-zero, nonrecursive, or moving-average (MA) filters.

For an infinite impulse response (IIR) filter, the transfer function is not a polynomial, but a rational function. The Z-transforms of the input and output signals are related by

$$Y(z) = H(z)X(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \ldots + a(m+1)z^{-m}}X(z),$$

where $b(i)$ and $a(i)$ are the filter coefficients. In this case, the order of the filter is the maximum of $n$ and $m$. IIR filters with $n = 0$ are also called all-pole, recursive, or autoregressive (AR) filters. IIR filters with both $n$ and $m$ greater than zero are also called pole-zero, recursive, or autoregressive moving-average (ARMA) filters. The acronyms AR, MA, and ARMA are usually applied to filters associated with filtered stochastic processes.

## Filtering with the filter Function

For IIR filters, the filtering operation is described not by a simple convolution, but by a difference equation that can be found from the transfer-function relation. Assume that $a(1) = 1$, move the denominator to the left side, and take the inverse Z-transform to obtain

$$y(k) + a(2)\,y(k-1) + \ldots + a(m+1)\,y(k-m) = b(1)\,x(k) + b(2)\,x(k-1) + \cdots + b(n+1)\,x(k-n)\,.$$

In terms of current and past inputs, and past outputs, $y(k)$ is

$$y(k) = b(1)\,x(k) + b(2)\,x(k-1) + \cdots + b(n+1)\,x(k-n) - a(2)\,y(k-1) - \cdots - a(m+1)\,y(k-m),$$

which is the standard time-domain representation of a digital filter. Starting with $y(1)$ and assuming a causal system with zero initial conditions, the representation is equivalent to

$$\begin{aligned}
y(1) &= b(1)\,x(1)\\
y(2) &= b(1)\,x(2) + b(2)\,x(1) - a(2)\,y(1)\\
y(3) &= b(1)\,x(3) + b(2)\,x(2) + b(3)\,x(1) - a(2)\,y(2) - a(3)\,y(1)\\
&\vdots\\
y(n) &= b(1)\,x(n) + \cdots + b(n)\,x(1) - a(2)\,y(n-1) - \cdots - a(n)\,y(1)\,.
\end{aligned}$$

To implement this filtering operation, you can use the MATLAB `filter` function. `filter` stores the coefficients in two row vectors, one for the numerator and one for the denominator. For example, to solve the difference equation

$$y(n) - 0.9y(n-1) = x(n) \quad \Rightarrow \quad Y(z) = \frac{1}{1 - 0.9\,z^{-1}}X(z) = H(z)\,X(z),$$

you can use

```
b = 1;
a = [1 -0.9];
y = filter(b,a,x);
```

`filter` gives you as many output samples as there are input samples, that is, the length of `y` is the same as the length of `x`. If the first element of $a$ is not 1, then `filter` divides the coefficients by $a(1)$ before implementing the difference equation.

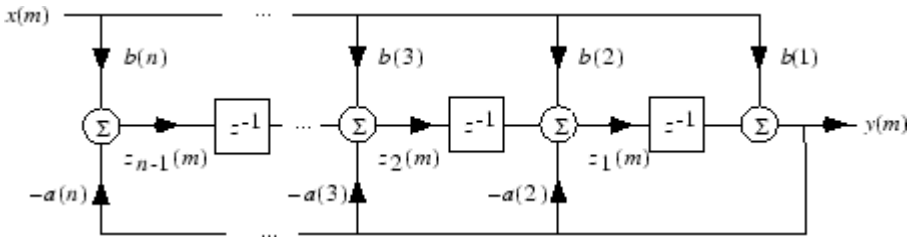
## See Also

**Apps**
**Filter Designer**

**Functions**
`conv` | `designfilt` | `filter`

# The filter Function

`filter` is implemented as the transposed direct-form II structure, where $n$–1 is the filter order. This is a canonical form that has the minimum number of delay elements.




At sample $m$, `filter` computes the difference equations

$$\begin{aligned} y(m) &= b(1)x(m) + z_1(m-1) \\ z_1(m) &= b(2)x(m) + z_2(m-1) - a(2)y(m) \\ &\vdots \quad = \quad \vdots \\ z_{n-2}(m) &= b(n-1)x(m) + z_{n-1}(m-1) - a(n-1)y(m) \\ z_{n-1}(m) &= b(n)x(m) - a(n)y(m) \end{aligned}$$

In its most basic form, `filter` initializes the delay outputs $z_i(1)$, $i = 1, \ldots, n\text{-}1$ to 0. This is equivalent to assuming both past inputs and outputs are zero. Set the initial delay outputs using a fourth input parameter to `filter`, or access the final delay outputs using a second output parameter:

```
[y,zf] = filter(b,a,x,zi)
```

Access to initial and final conditions is useful for filtering data in sections, especially if memory limitations are a consideration. Suppose you have collected data in two segments of 5000 points each:

```
x1 = randn(5000,1);  % Generate two random data sequences.
x2 = randn(5000,1);
```

Perhaps the first sequence, `x1`, corresponds to the first 10 minutes of data and the second, `x2`, to an additional 10 minutes. The whole sequence is `x = [x1;x2]`. If there is not sufficient memory to hold the combined sequence, filter the subsequences `x1` and `x2` one at a time. To ensure continuity of the filtered sequences, use the final conditions from `x1` as initial conditions to filter `x2`:

```
[y1,zf] = filter(b,a,x1);
y2 = filter(b,a,x2,zf);
```

The `filtic` function generates initial conditions for `filter`. `filtic` computes the delay vector to make the behavior of the filter reflect past inputs and outputs that you specify. To obtain the same output delay values `zf` as above using `filtic`, use

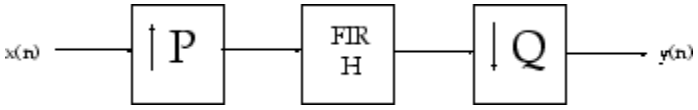
```
zf = filtic(b,a,flipud(y1),flipud(x1));
```

This can be useful when filtering short data sequences, as appropriate initial conditions help reduce transient startup effects.

# Multirate Filter Bank Implementation

The `upfirdn` function alters the sampling rate of a signal by an integer ratio P/Q. It computes the result of a cascade of three systems that performs the following tasks:

- Upsampling (zero insertion) by integer factor `p`
- Filtering by FIR filter `h`
- Downsampling by integer factor `q`




For example, to change the sample rate of a signal from 44.1 kHz to 48 kHz, we first find the smallest integer conversion ratio `p/q`. Set

```
d = gcd(48000,44100);
p = 48000/d;
q = 44100/d;
```

In this example, `p = 160` and `q = 147`. Sample rate conversion is then accomplished by typing

```
y = upfirdn(x,h,p,q)
```

This cascade of operations is implemented in an efficient manner using polyphase filtering techniques, and it is a central concept of multirate filtering. Note that the quality of the resampling result relies on the quality of the FIR filter `h`.

Filter banks may be implemented using `upfirdn` by allowing the filter `h` to be a matrix, with one FIR filter per column. A signal vector is passed independently through each FIR filter, resulting in a matrix of output signals.

Other functions that perform multirate filtering (with fixed filter) include `resample`, `interp`, and `decimate`.

# Frequency Domain Filter Implementation

Duality between the time domain and the frequency domain makes it possible to perform any operation in either domain. Usually one domain or the other is more convenient for a particular operation, but you can always accomplish a given operation in either domain.

To implement general IIR filtering in the frequency domain, multiply the discrete Fourier transform (DFT) of the input sequence with the quotient of the DFT of the filter:

```
n = length(x);
y = ifft(fft(x).*fft(b,n)./fft(a,n));
```

This computes results that are identical to `filter`, but with different startup transients (edge effects). For long sequences, this computation is very inefficient because of the large zero-padded FFT operations on the filter coefficients, and because the FFT algorithm becomes less efficient as the number of points `n` increases.

For FIR filters, however, it is possible to break longer sequences into shorter, computationally efficient FFT lengths. The function

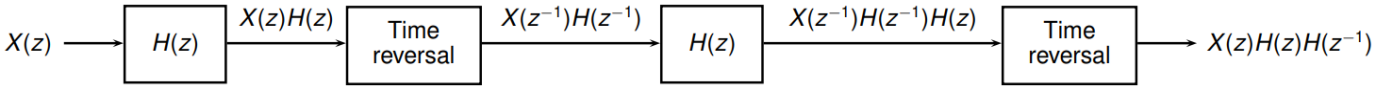
```
y = fftfilt(b,x)
```

uses the overlap add method to filter a long sequence with multiple medium-length FFTs. Its output is equivalent to `filter(b,1,x)`.

# Anti-Causal, Zero-Phase Filter Implementation

In the case of FIR filters, it is possible to design linear phase filters that, when applied to data (using `filter` or `conv`), simply delay the output by a fixed number of samples. For IIR filters, however, the phase distortion is usually highly nonlinear. The `filtfilt` function uses the information in the signal at points before and after the current point, in essence "looking into the future," to eliminate phase distortion.

To see how `filtfilt` does this, recall that if the Z-transform of a real sequence $x(n)$ is $X(z)$, then the Z-transform of the time-reversed sequence $x(-n)$ is $X(z^{-1})$. Consider the following processing scheme:



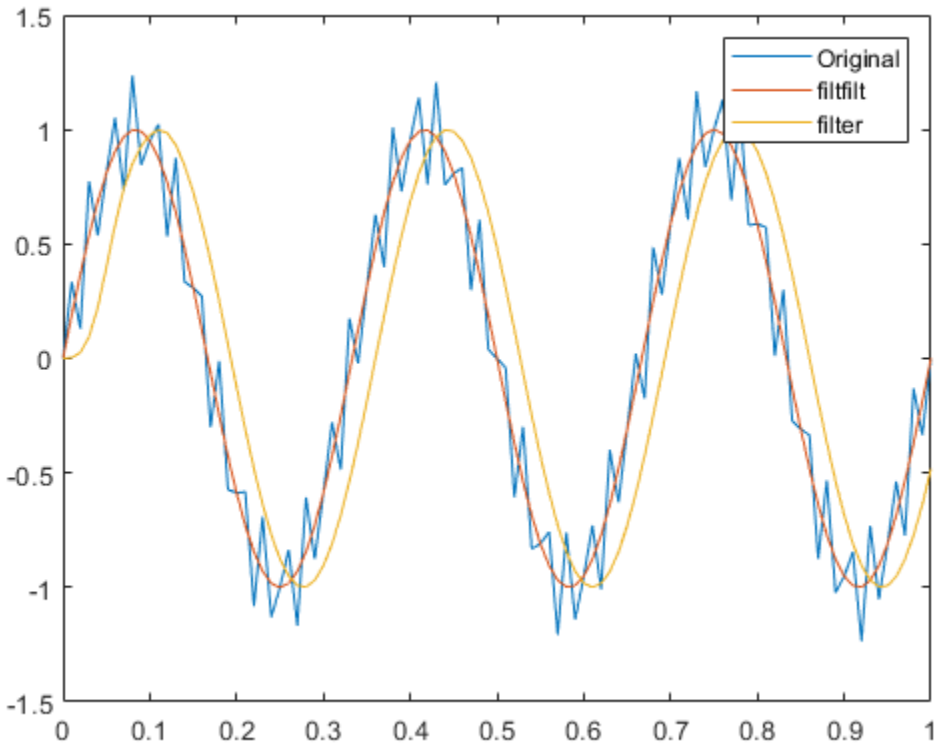

When $|z| = 1$, that is $z = e^{j\omega}$, the output reduces to $X(e^{j\omega})\left|H(e^{j\omega})\right|^2$. Given all the samples of the sequence $x(n)$, a doubly filtered version of $x$ that has zero-phase distortion is possible.

For example, a 1-second duration signal sampled at 100 Hz, composed of two sinusoidal components at 3 Hz and 40 Hz, is

```
fs = 100;
t = 0:1/fs:1;
x = sin(2*pi*t*3)+.25*sin(2*pi*t*40);
```

Now create a 6th-order Butterworth lowpass filter to filter out the high-frequency sinusoid. Filter `x` using both `filter` and `filtfilt` for comparison:

```
[b,a] = butter(6,20/(fs/2));

y = filtfilt(b,a,x);
yy = filter(b,a,x);

plot(t,x,t,y,t,yy)
legend('Original','filtfilt','filter')
```




Both filtered versions eliminate the 40 Hz sinusoid evident in the original signal. The plot also shows how `filter` and `filtfilt` differ. The `filtfilt` line is in phase with the original 3 Hz sinusoid, while the `filter` line is delayed. The `filter` line shows a transient at early times. `filtfilt` reduces filter startup transients by carefully choosing initial conditions, and by prepending onto the input sequence a short, reflected piece of the input sequence.

For best results, make sure the sequence you are filtering has length at least three times the filter order and tapers to zero on both edges.

## See Also

`conv` | `filter` | `filtfilt`

# Impulse Response

The impulse response of a digital filter is the output arising from the unit impulse sequence defined as

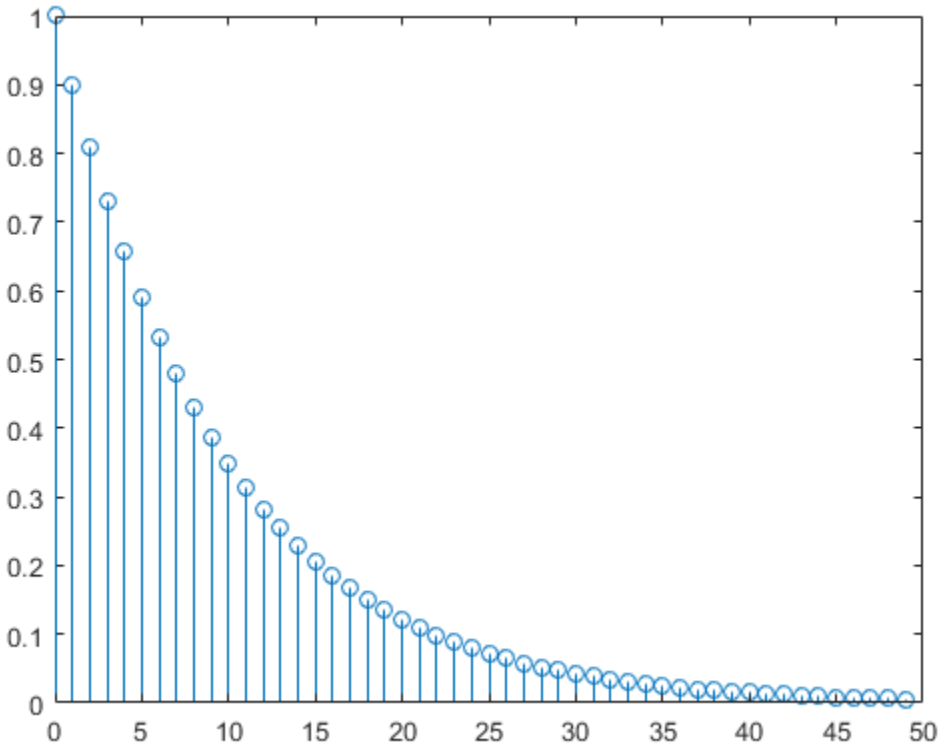
$$\delta(n) = \begin{cases} 1, & n = 0, \\ 0, & n \neq 0. \end{cases}$$

You can generate an impulse sequence a number of ways; one straightforward way is

```
imp = [1; zeros(49,1)];
```

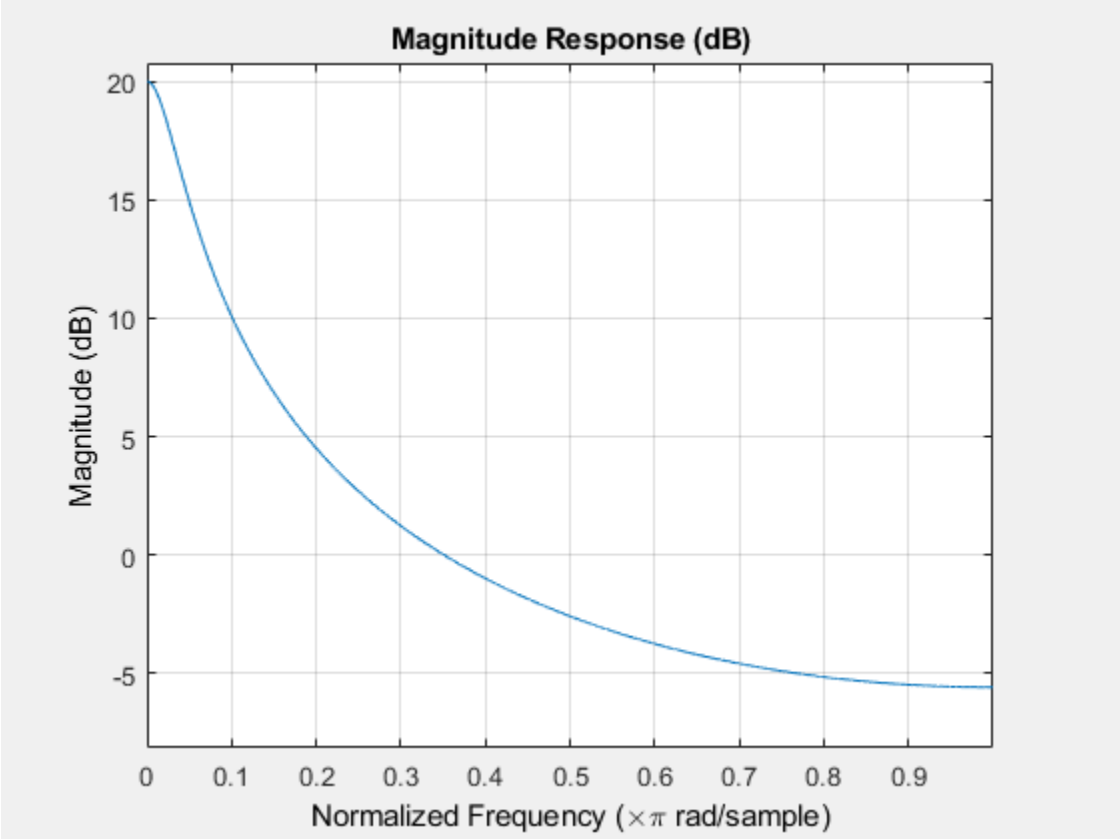
The impulse response of the simple filter with $b = 1$ and $a = [1 \; -0.9]$ is $h(n) = 0.9^n$, which decays exponentially.

```
b = 1;
a = [1 -0.9];

h = filter(b,a,imp);

stem(0:49,h)
```



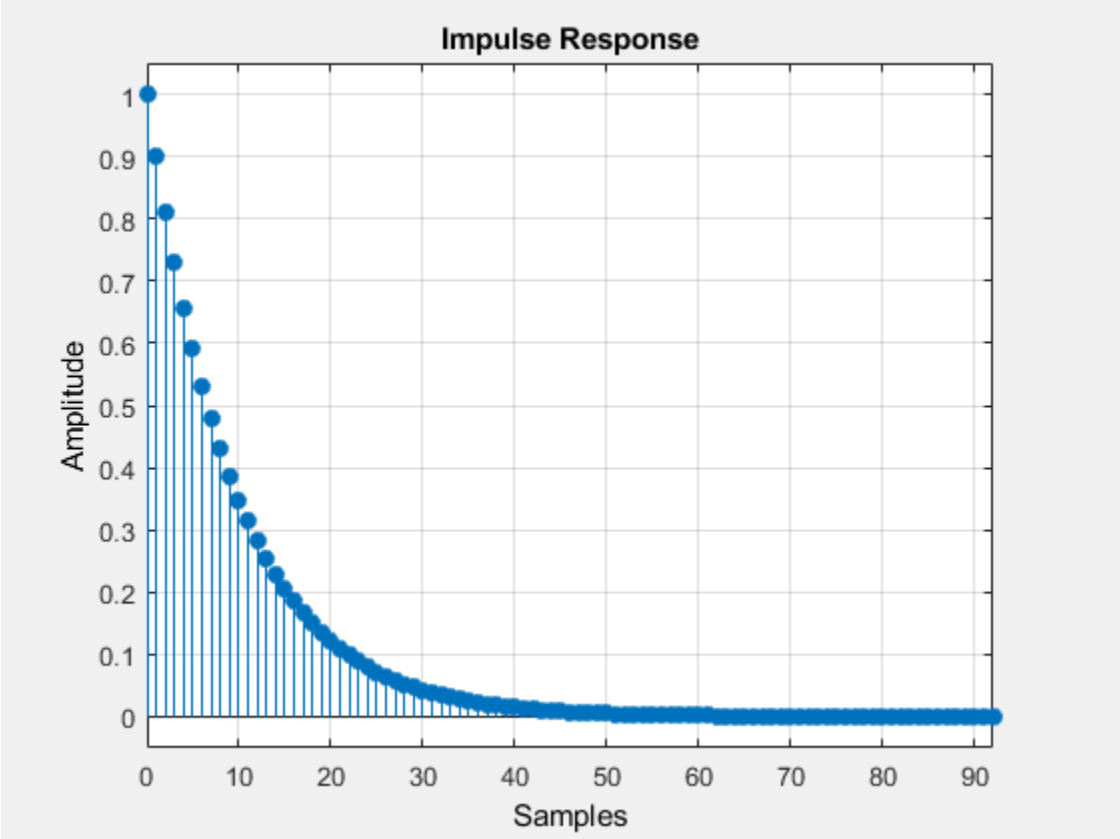

A simple way to display the impulse response is with the Filter Visualization Tool, `fvtool`.

```
fvtool(b,a)
```




Click the **Impulse Response** button, [ ↑ ], on the toolbar, select **Analysis > Impulse Response** from the menu, or type the following code to obtain the exponential decay of the single-pole system.

```
fvtool(b,a,'Analysis','impulse')
```


Impulse Response
Amplitude
Samples
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
10
20
30
40
50
60
70
80
90

# Frequency Response

| In this section... |
| --- |
| “Digital Domain” on page 1-13 |
| “Analog Domain” on page 1-18 |

## Digital Domain

`freqz` uses an FFT-based algorithm to calculate the Z-transform frequency response of a digital filter. Specifically, the statement

```
[h,w] = freqz(b,a,p)
```

returns the $p$-point complex frequency response, $H(e^{j\omega})$, of the digital filter.

$$H(e^{j\omega}) = \frac{b(1) + b(2)e^{-j\omega} + \ldots + b(n+1)e^{-j\omega n}}{a(1) + a(2)e^{-j\omega} + \ldots + a(m+1)e^{-j\omega m}}$$

In its simplest form, `freqz` accepts the filter coefficient vectors `b` and `a`, and an integer `p` specifying the number of points at which to calculate the frequency response. `freqz` returns the complex frequency response in vector `h`, and the actual frequency points in vector `w` in rad/s.

`freqz` can accept other parameters, such as a sampling frequency or a vector of arbitrary frequency points. The example below finds the 256-point frequency response for a 12th-order Chebyshev Type I filter. The call to `freqz` specifies a sampling frequency `fs` of 1000 Hz:

```
[b,a] = cheby1(12,0.5,200/500);
[h,f] = freqz(b,a,256,1000);
```

Because the parameter list includes a sampling frequency, `freqz` returns a vector `f` that contains the 256 frequency points between 0 and `fs/2` used in the frequency response calculation.

---

**Note** This toolbox uses the convention that unit frequency is the Nyquist frequency, defined as half the sampling frequency. The cutoff frequency parameter for all basic filter design functions is normalized by the Nyquist frequency. For a system with a 1000 Hz sampling frequency, for example, 300 Hz is 300/500 = 0.6. To convert normalized frequency to angular frequency around the unit circle, multiply by $\pi$. To convert normalized frequency back to hertz, multiply by half the sample frequency.

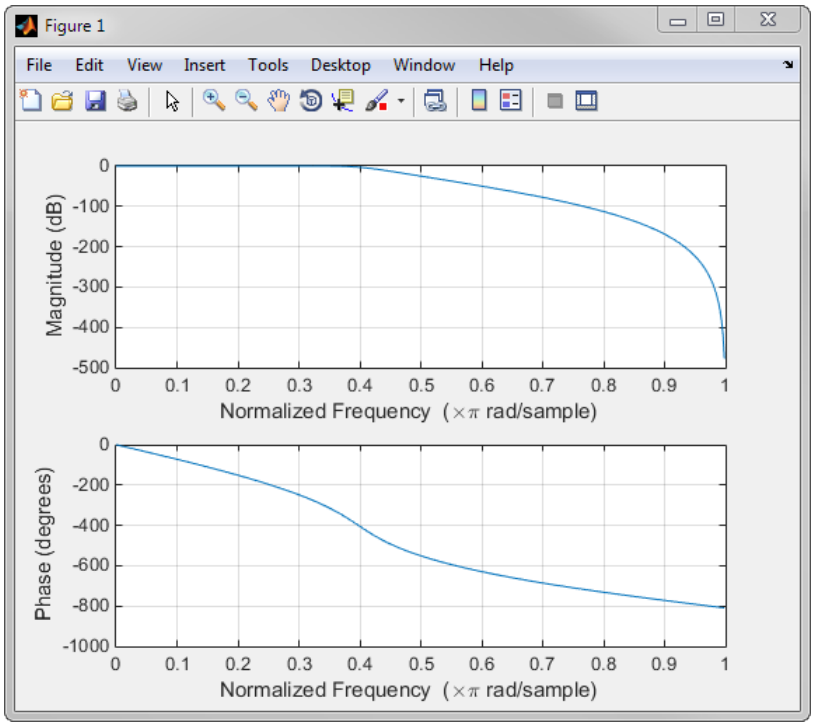
---

If you call `freqz` with no output arguments, it plots both magnitude versus frequency and phase versus frequency. For example, a ninth-order Butterworth lowpass filter with a cutoff frequency of 400 Hz, based on a 2000 Hz sampling frequency, is

```
[b,a] = butter(9,400/1000);
```

To calculate the 256-point complex frequency response for this filter, and plot the magnitude and phase with `freqz`, use

```
freqz(b,a,256,2000)
```




`freqz` can also accept a vector of arbitrary frequency points for use in the frequency response calculation. For example,

```
w = linspace(0,pi);
h = freqz(b,a,w);
```

calculates the complex frequency response at the frequency points in `w` for the filter defined by vectors `b` and `a`. The frequency points can range from 0 to $2\pi$. To specify a frequency vector that ranges from zero to your sampling frequency, include both the frequency vector and the sampling frequency value in the parameter list.

These examples show how to compute and display digital frequency responses.

### Frequency Response from Transfer Function

Compute and display the magnitude response of the third-order IIR lowpass filter described by the following transfer function:

$$H(z) = \frac{0.05634(1+z^{-1})(1-1.0166z^{-1}+z^{-2})}{(1-0.683z^{-1})(1-1.4461z^{-1}+0.7957z^{-2})}.$$

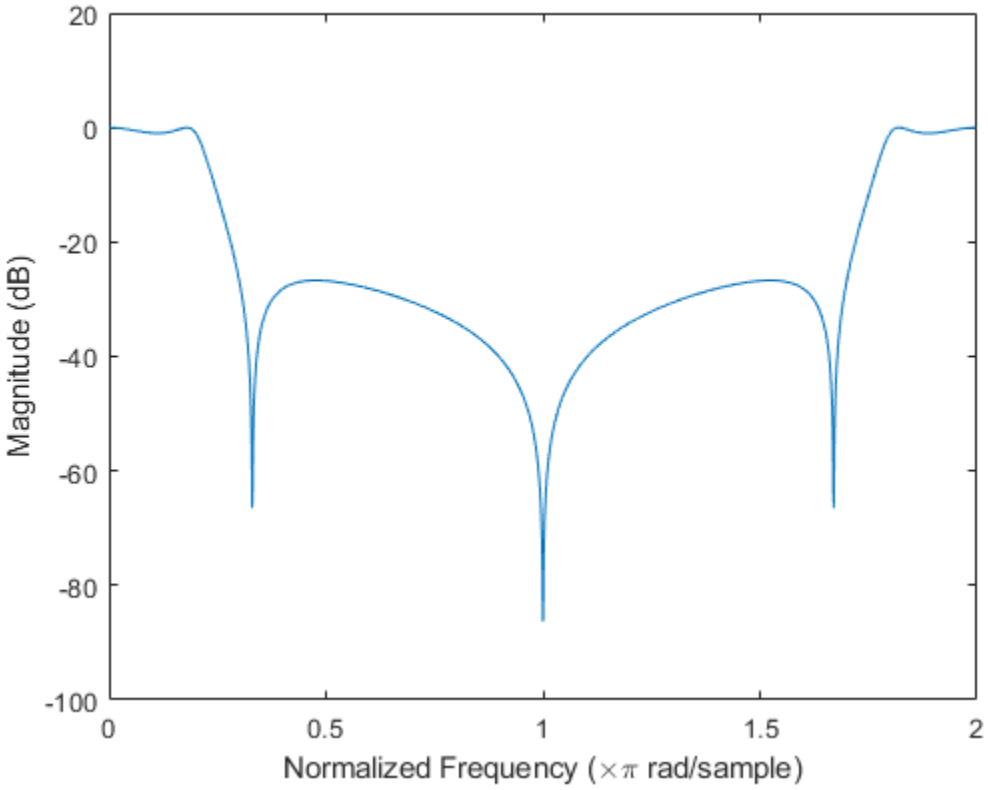
Express the numerator and denominator as polynomial convolutions. Find the frequency response at 2001 points spanning the complete unit circle.

```
b0 = 0.05634;
b1 = [1  1];
b2 = [1 -1.0166 1];
a1 = [1 -0.683];
a2 = [1 -1.4461 0.7957];

b = b0*conv(b1,b2);
```

```
a = conv(a1,a2);

[h,w] = freqz(b,a,'whole',2001);
```

Plot the magnitude response expressed in decibels.

```
plot(w/pi,20*log10(abs(h)))
ax = gca;
ax.YLim = [-100 20];
ax.XTick = 0:.5:2;
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Magnitude (dB)')
```

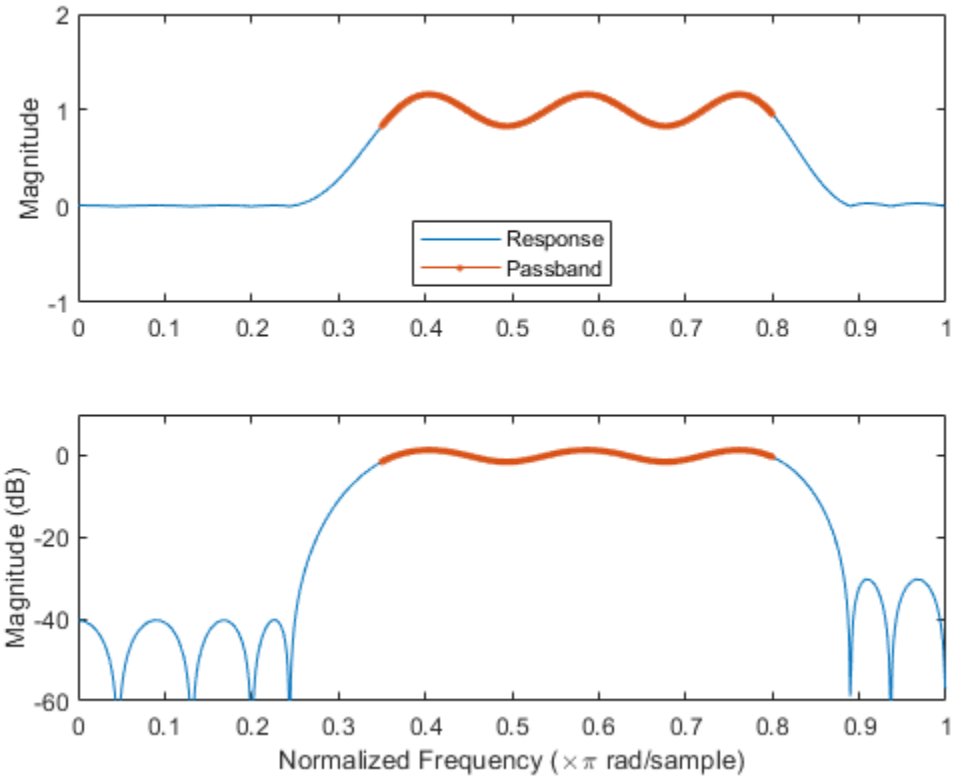



### Frequency Response of an FIR Bandpass Filter

Design an FIR bandpass filter with passband between $0.35\pi$ and $0.8\pi$ rad/sample and 3 dB of ripple. The first stopband goes from 0 to $0.1\pi$ rad/sample and has an attenuation of 40 dB. The second stopband goes from $0.9\pi$ rad/sample to the Nyquist frequency and has an attenuation of 30 dB. Compute the frequency response. Plot its magnitude in both linear units and decibels. Highlight the passband.

```
sf1 = 0.1;
pf1 = 0.35;
pf2 = 0.8;
sf2 = 0.9;
pb = linspace(pf1,pf2,1e3)*pi;
```

```
bp = designfilt('bandpassfir', ...
    'StopbandAttenuation1',40, 'StopbandFrequency1',sf1,...
    'PassbandFrequency1',pf1,'PassbandRipple',3,'PassbandFrequency2',pf2, ...
    'StopbandFrequency2',sf2,'StopbandAttenuation2',30);

[h,w] = freqz(bp,1024);
hpb = freqz(bp,pb);

subplot(2,1,1)
plot(w/pi,abs(h),pb/pi,abs(hpb),'.-')
axis([0 1 -1 2])
legend('Response','Passband','Location','South')
ylabel('Magnitude')

subplot(2,1,2)
plot(w/pi,db(h),pb/pi,db(hpb),'.-')
axis([0 1 -60 10])
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Magnitude (dB)')
```

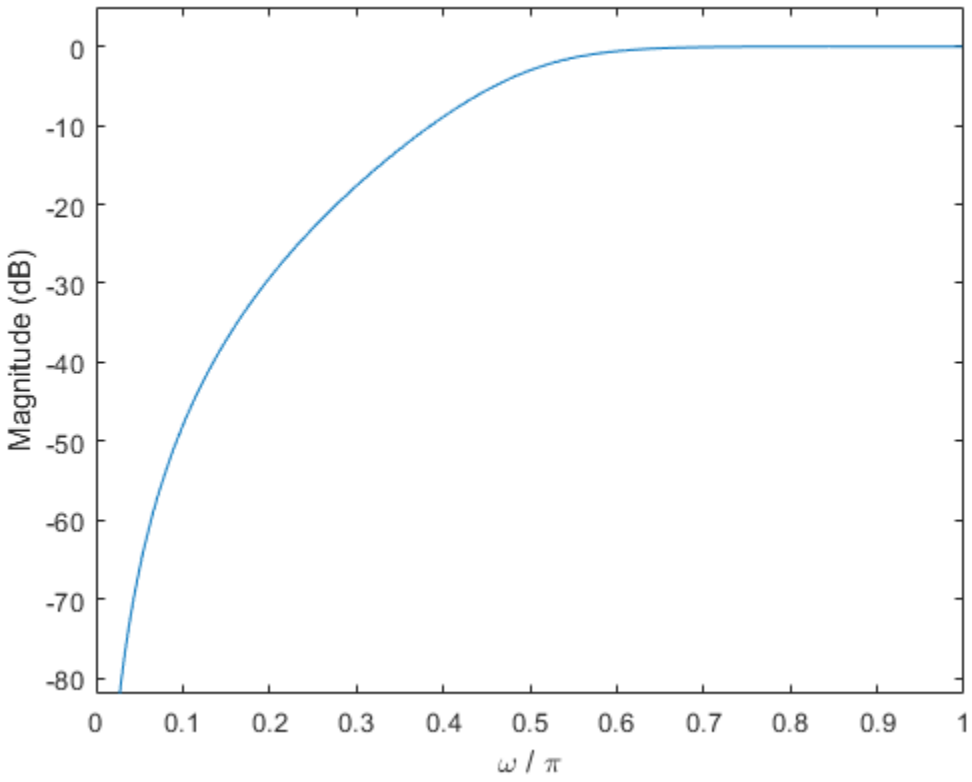



### Magnitude Response of a Highpass Filter

Design a 3rd-order highpass Butterworth filter having a normalized 3-dB frequency of $0.5\pi$ rad/sample. Compute its frequency response. Express the magnitude response in decibels and plot it.

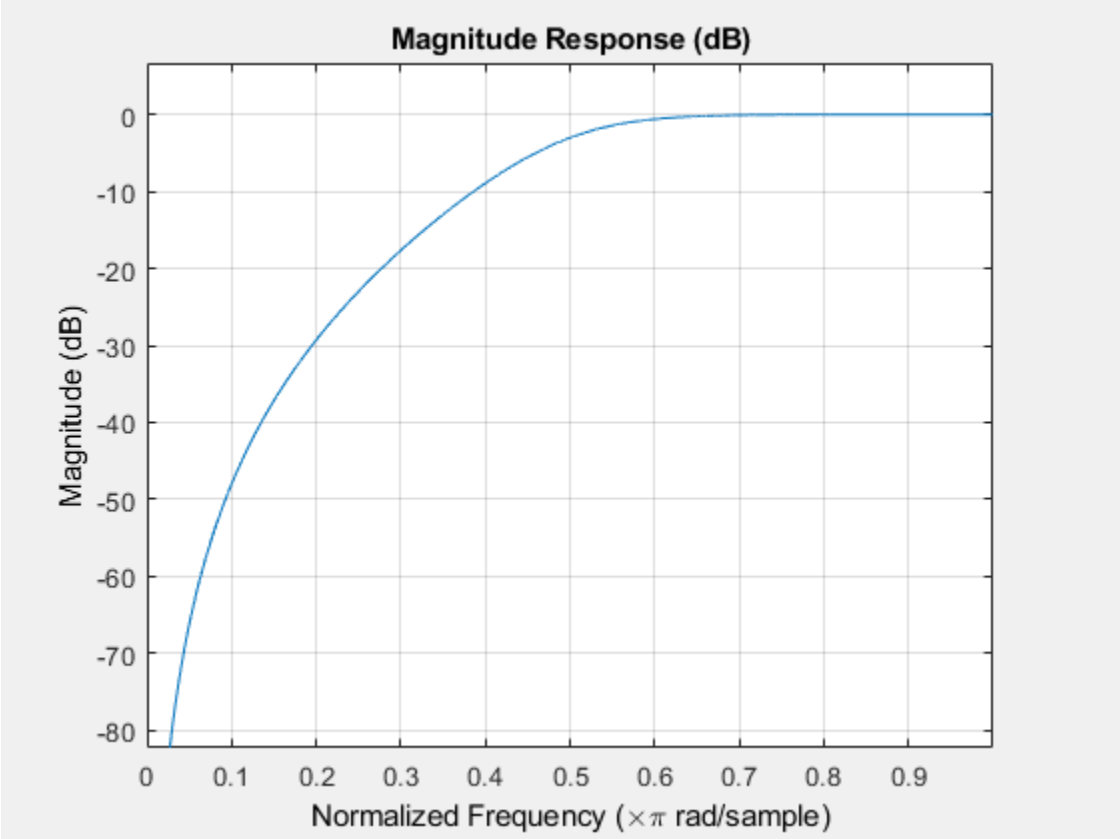
```
[b,a] = butter(3,0.5,'high');
[h,w] = freqz(b,a);
```

```
dB = mag2db(abs(h));

plot(w/pi,dB)
xlabel('\omega / \pi')
ylabel('Magnitude (dB)')
ylim([-82 5])
```




Repeat the computation using `fvtool`.

```
fvtool(b,a)
```




## Analog Domain

`freqs` evaluates frequency response for an analog filter defined by two input coefficient vectors, `b` and `a`. Its operation is similar to that of `freqz`; you can specify a number of frequency points to use, supply a vector of arbitrary frequency points, and plot the magnitude and phase response of the filter. This example shows how to compute and display analog frequency responses.

### Comparison of Analog IIR Lowpass Filters

Design a 5th-order analog Butterworth lowpass filter with a cutoff frequency of 2 GHz. Multiply by $2\pi$ to convert the frequency to radians per second. Compute the frequency response of the filter at 4096 points.

```
n = 5;
f = 2e9;

[zb,pb,kb] = butter(n,2*pi*f,'s');
[bb,ab] = zp2tf(zb,pb,kb);
[hb,wb] = freqs(bb,ab,4096);
```

Design a 5th-order Chebyshev Type I filter with the same edge frequency and 3 dB of passband ripple. Compute its frequency response.

```
[z1,p1,k1] = cheby1(n,3,2*pi*f,'s');
[b1,a1] = zp2tf(z1,p1,k1);
[h1,w1] = freqs(b1,a1,4096);
```

Design a 5th-order Chebyshev Type II filter with the same edge frequency and 30 dB of stopband attenuation. Compute its frequency response.

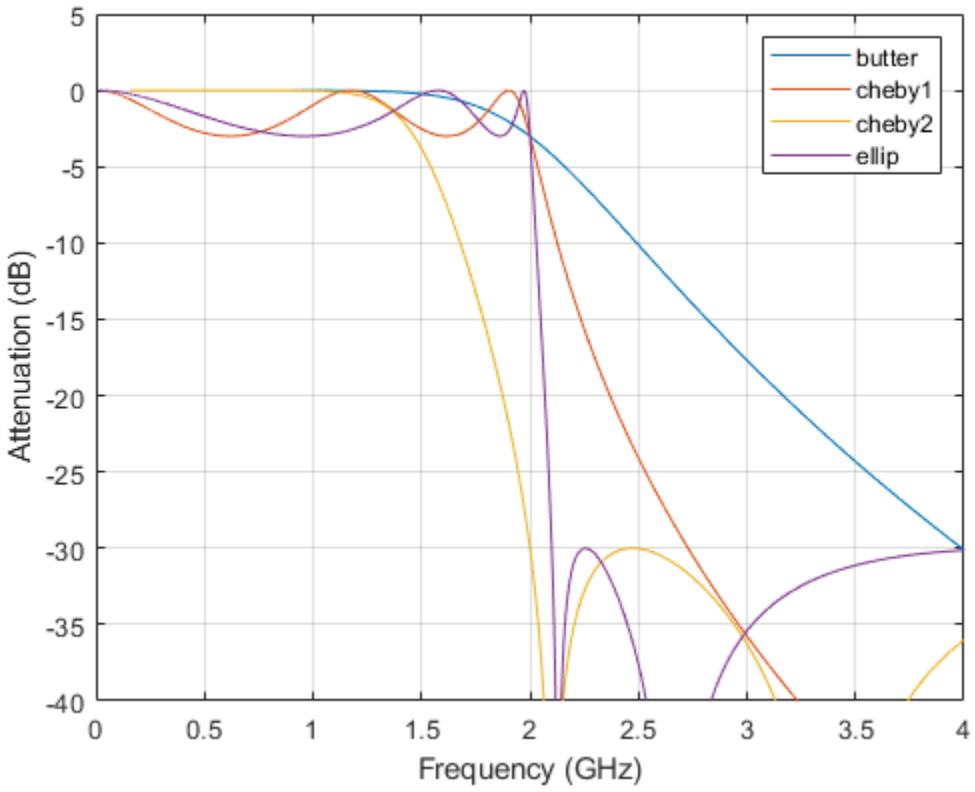
```
[z2,p2,k2] = cheby2(n,30,2*pi*f,'s');
[b2,a2] = zp2tf(z2,p2,k2);
[h2,w2] = freqs(b2,a2,4096);
```

Design a 5th-order elliptic filter with the same edge frequency, 3 dB of passband ripple, and 30 dB of stopband attenuation. Compute its frequency response.

```
[ze,pe,ke] = ellip(n,3,30,2*pi*f,'s');
[be,ae] = zp2tf(ze,pe,ke);
[he,we] = freqs(be,ae,4096);
```

Plot the attenuation in decibels. Express the frequency in gigahertz. Compare the filters.

```
plot(wb/(2e9*pi),mag2db(abs(hb)))
hold on
plot(w1/(2e9*pi),mag2db(abs(h1)))
plot(w2/(2e9*pi),mag2db(abs(h2)))
plot(we/(2e9*pi),mag2db(abs(he)))
axis([0 4 -40 5])
grid
xlabel('Frequency (GHz)')
ylabel('Attenuation (dB)')
legend('butter','cheby1','cheby2','ellip')
```

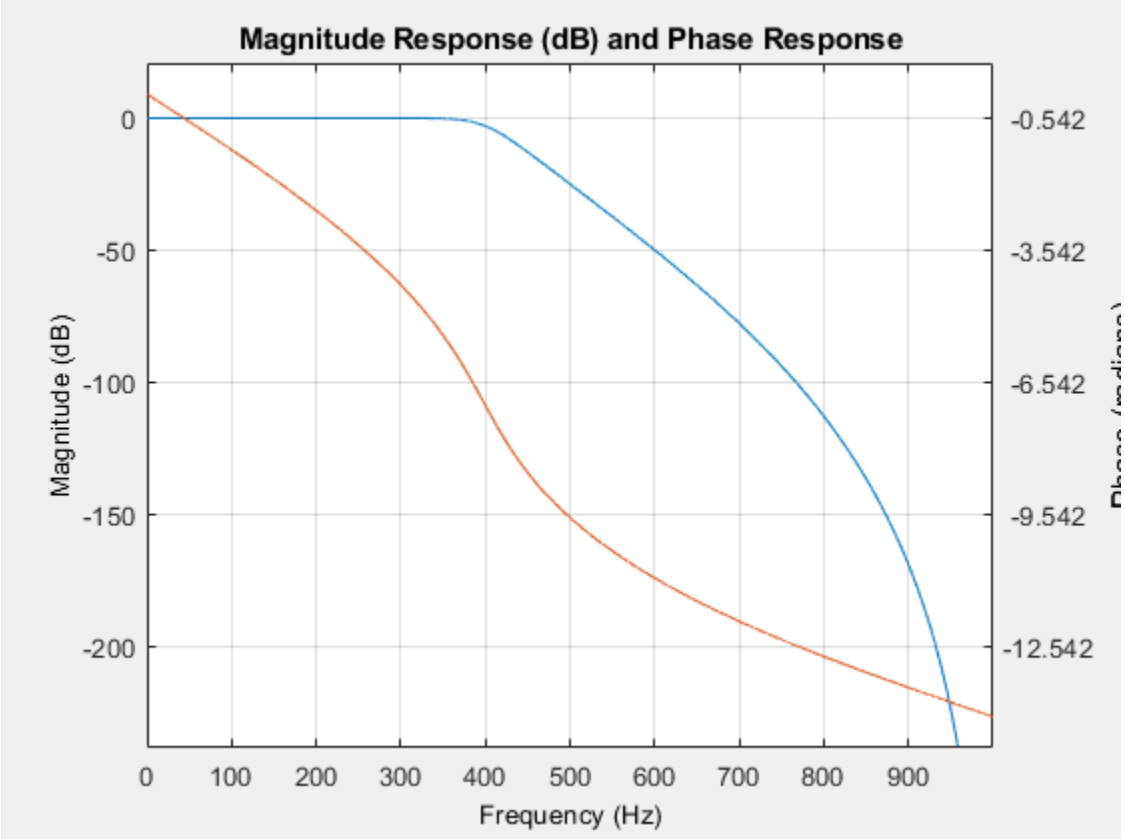


The Butterworth and Chebyshev Type II filters have flat passbands and wide transition bands. The Chebyshev Type I and elliptic filters roll off faster but have passband ripple. The frequency input to the Chebyshev Type II design function sets the beginning of the stopband rather than the end of the passband.

# Phase Response

MATLAB® functions are available to extract the phase response of a filter. Given a frequency response, the function `abs` returns the magnitude and `angle` returns the phase angle in radians. To view the magnitude and phase of a Butterworth filter using `fvtool`:

```
d = designfilt('lowpassiir','FilterOrder',9, ...
    'HalfPowerFrequency',400,'SampleRate',2000);
fvtool(d,'Analysis','freq')
```



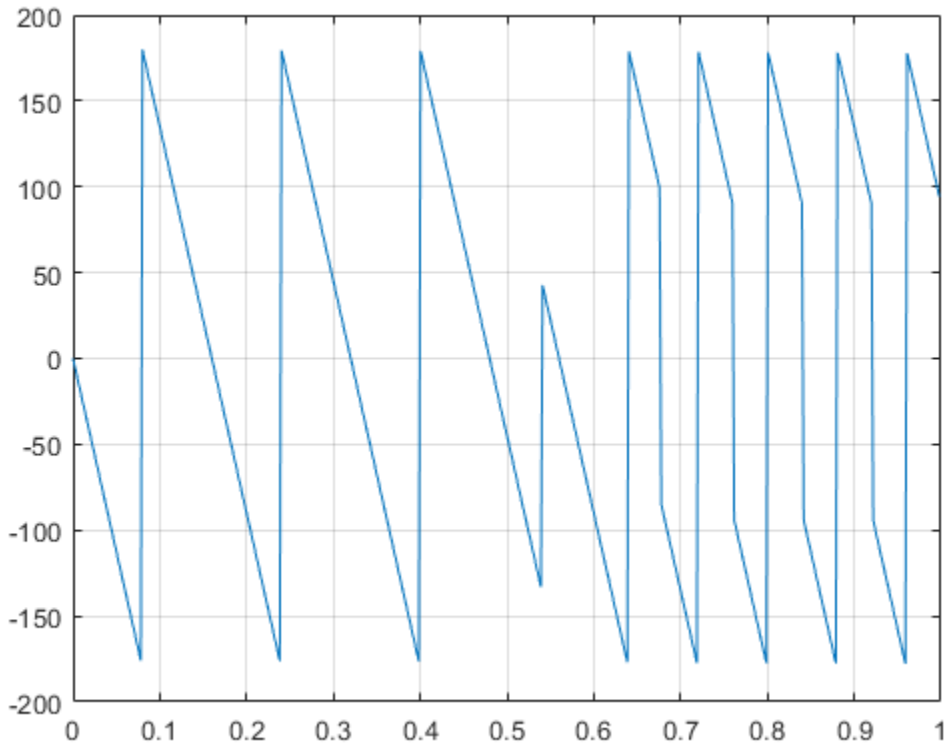

You can also click the **Magnitude and Phase Response** button on the toolbar or select **Analysis** > **Magnitude and Phase Response** to display the plot.

The `unwrap` function is also useful in frequency analysis. `unwrap` unwraps the phase to make it continuous across 360° phase discontinuities by adding multiples of ±360°, as needed. To see how `unwrap` is useful, design a 25th-order lowpass FIR filter:

```
h = fir1(25,0.4);
```

Obtain the frequency response with `freqz` and plot the phase in degrees:

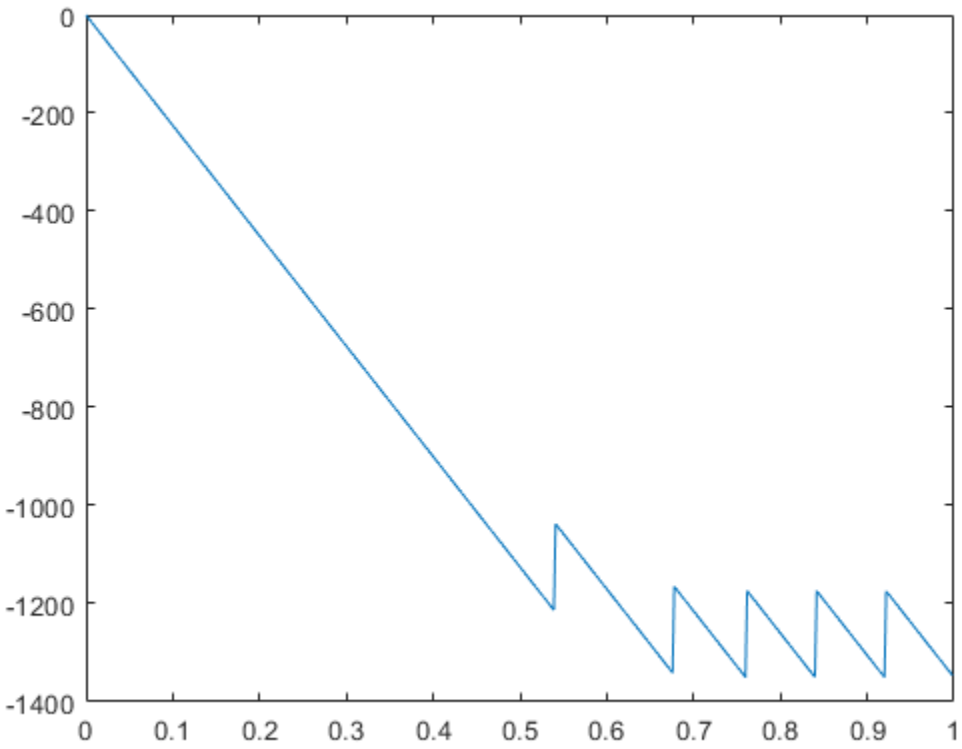
```
[H,f] = freqz(h,1,512,2);
plot(f,angle(H)*180/pi)
grid
```




It is difficult to distinguish the 360° jumps (an artifact of the arctangent function inside `angle`) from the 180° jumps that signify zeros in the frequency response.

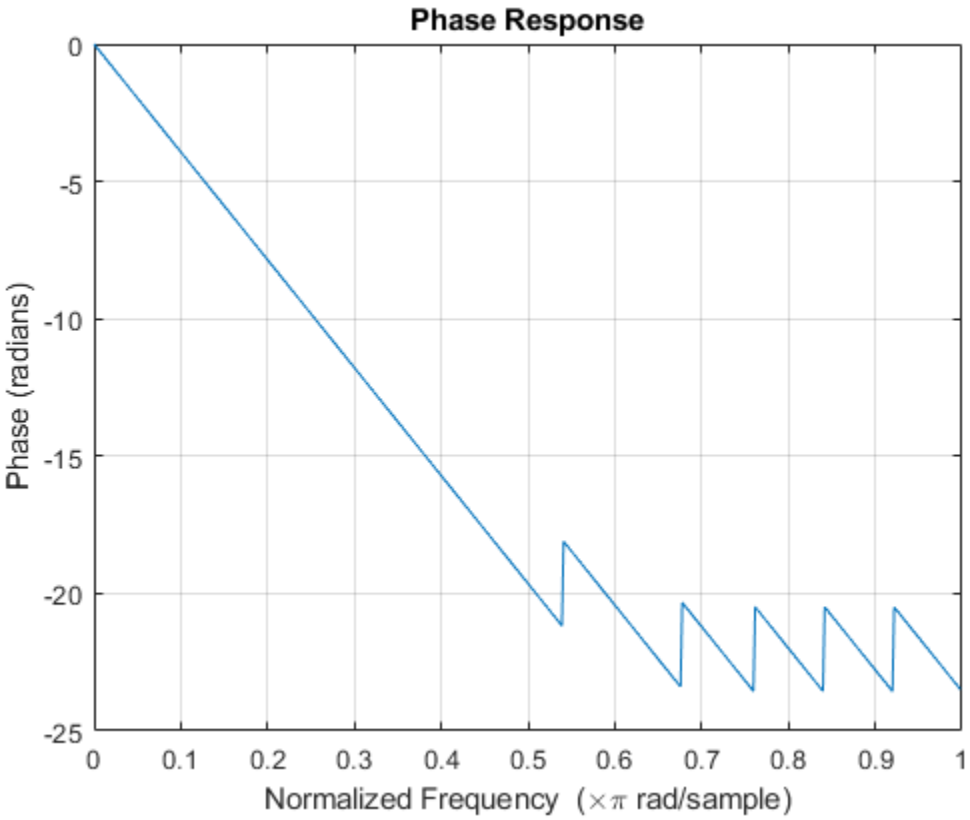
`unwrap` eliminates the 360° jumps:

```
plot(f,unwrap(angle(H))*180/pi)
```




Alternatively, you can use `phasez` to see the unwrapped phase:

```
phasez(h,1)
```




## See Also

**FVTool** | `abs` | `angle` | `freqz` | `phasez` | `unwrap`

# Group Delay and Phase Delay

The *group delay* of a filter is a measure of the average time delay of the filter as a function of frequency. The group delay is defined as the negative first derivative of the filter's phase response. If the complex frequency response of a filter is $H\left(e^{j\omega}\right)$, then the group delay is

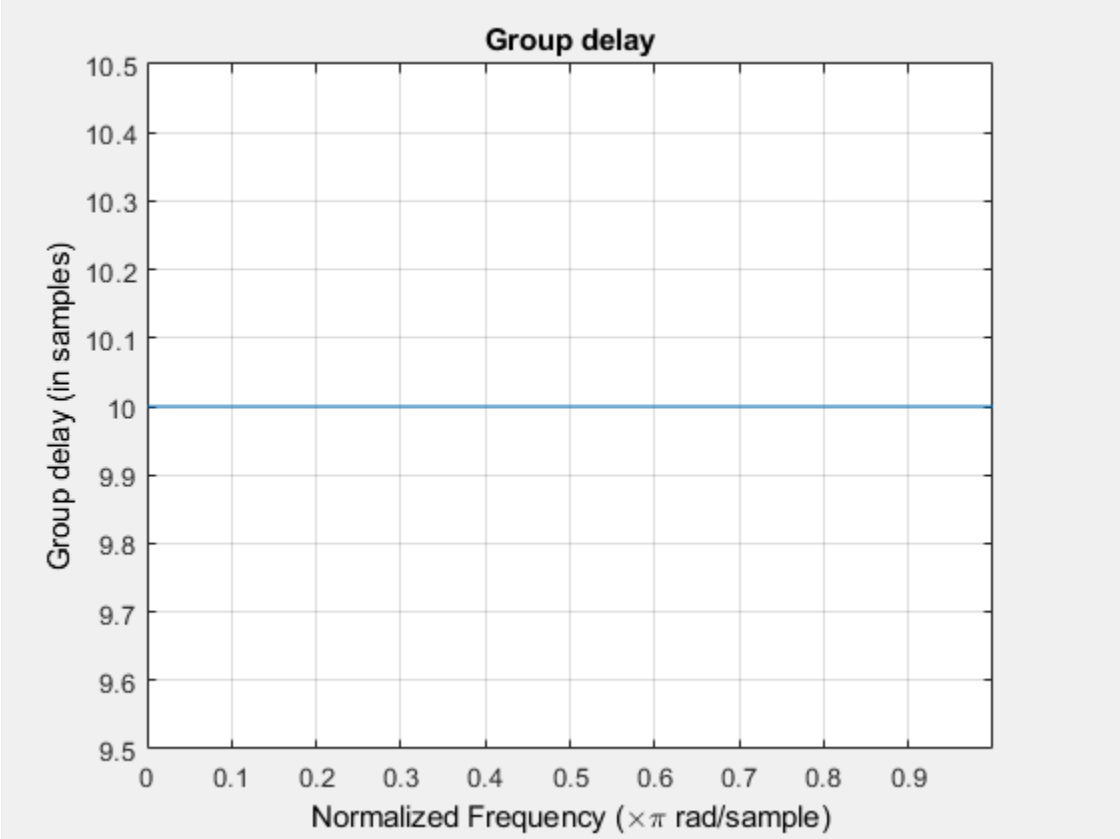
$$\tau_g(\omega) = -\frac{d\theta(\omega)}{d\omega},$$

where $\theta(\omega)$ is the phase, or argument, of $H\left(e^{j\omega}\right)$. Use the `grpdelay` function to compute group delay of a filter. For example, verify that, for a linear-phase FIR filter, the group delay is one-half the filter order.

```
d = designfilt('lowpassfir','FilterOrder',20,'CutoffFrequency',200,'SampleRate',2000);
islinphase(d)
```

```
ans = logical
   1
```

```
grpdelay(d)
```

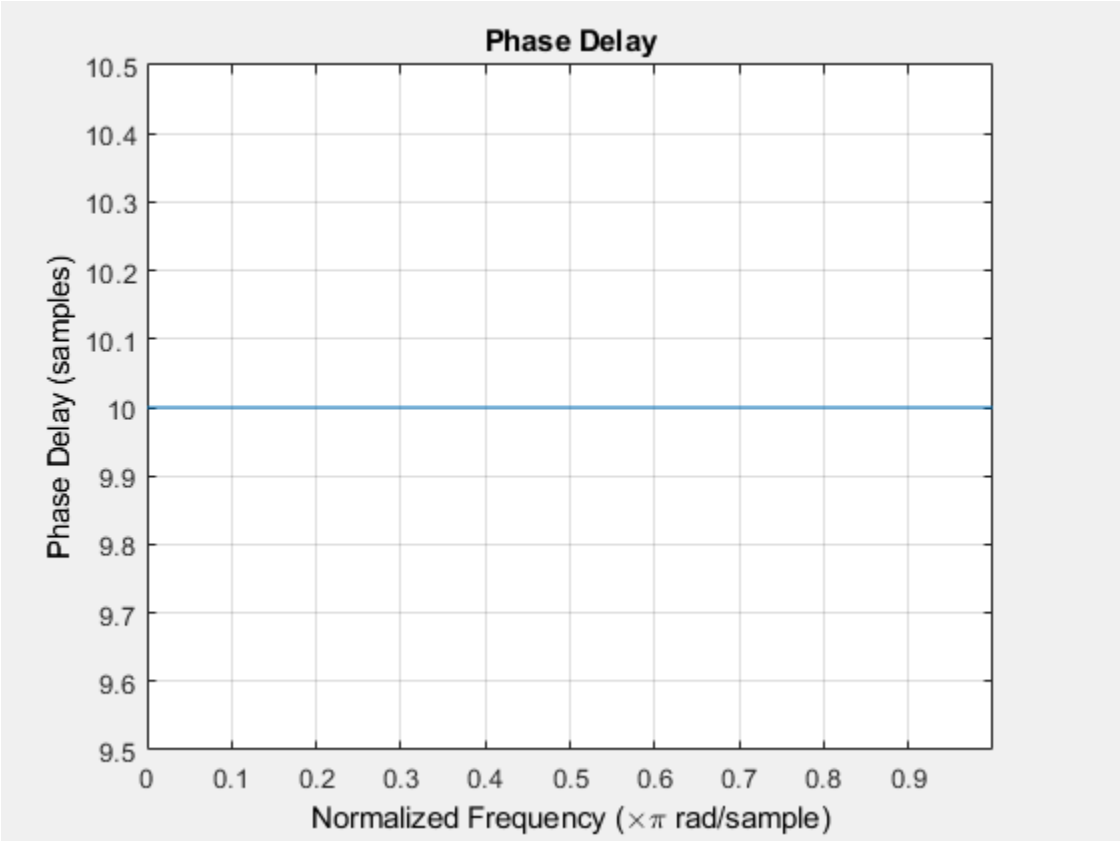



The *phase delay* of a filter is defined as the negative of the phase divided by the frequency:

$$\tau_p(\omega) = -\frac{\theta(\omega)}{\omega}.$$

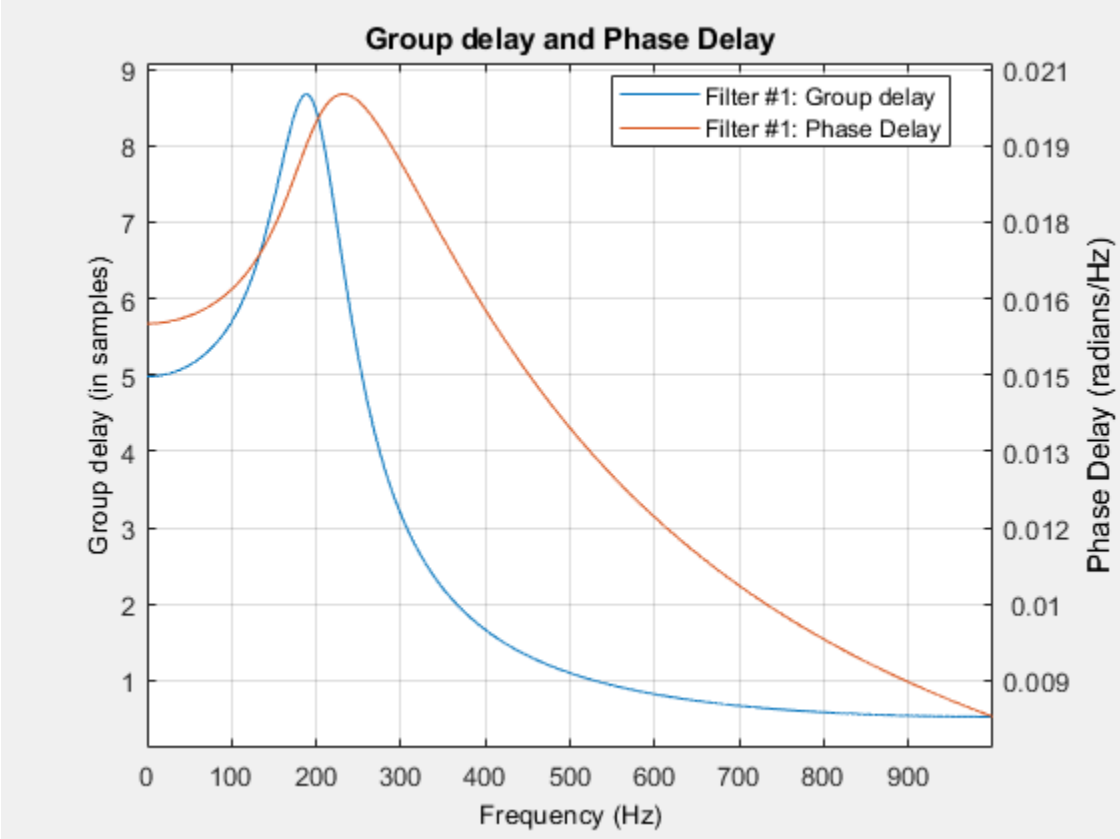
Use the `phasedelay` function to compute the phase delay of a filter. For the linear-phase FIR filter of the previous example, the phase delay is equal to the group delay.

```
phasedelay(d)
```




Use FVTool to plot both the group delay and the phase delay of a system on the same graph. For example, plot the delays of a fifth-order Butterworth lowpass filter.

```
d = designfilt('lowpassiir','FilterOrder',5,'HalfPowerFrequency',200,'SampleRate',2000);

fvtool(d,'Analysis','grpdelay','OverlayedAnalysis','phasedelay','Legend','on')
```




## See Also

**FVTool** | `grpdelay` | `phasedelay`

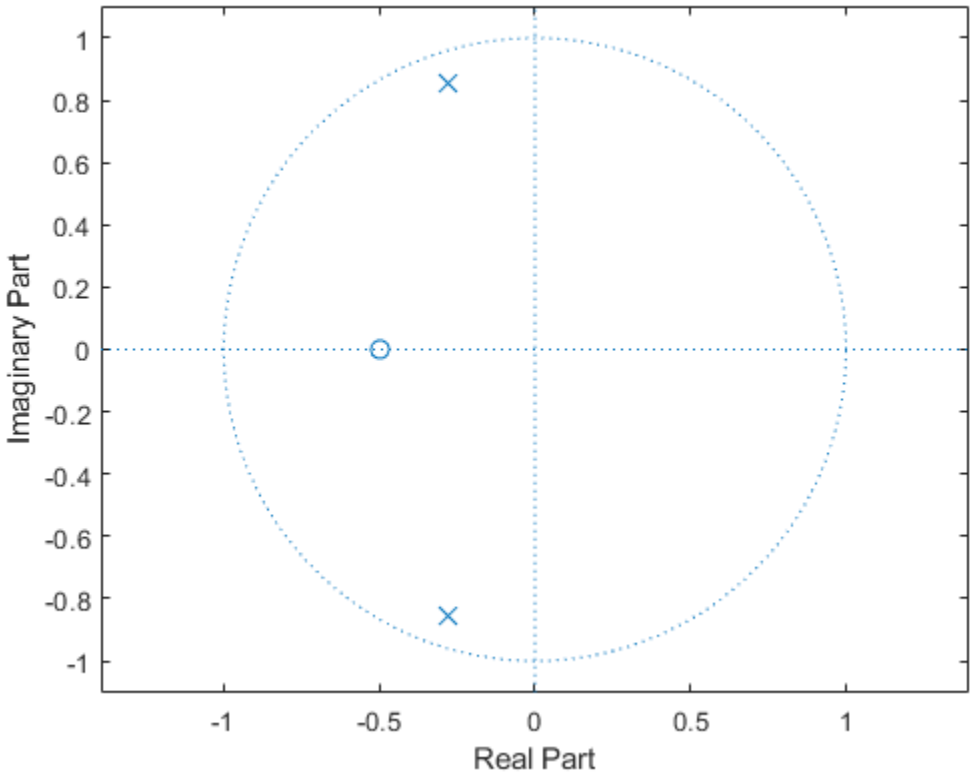
# Zero-Pole Analysis

The `zplane` function plots poles and zeros of a linear system. For example, a simple filter with a zero at -1/2 and a complex pole pair at $0.9e^{-j2\pi 0.3}$ and $0.9e^{j2\pi 0.3}$ is

```
zer = -0.5;
pol = 0.9*exp(j*2*pi*[-0.3 0.3]');
```

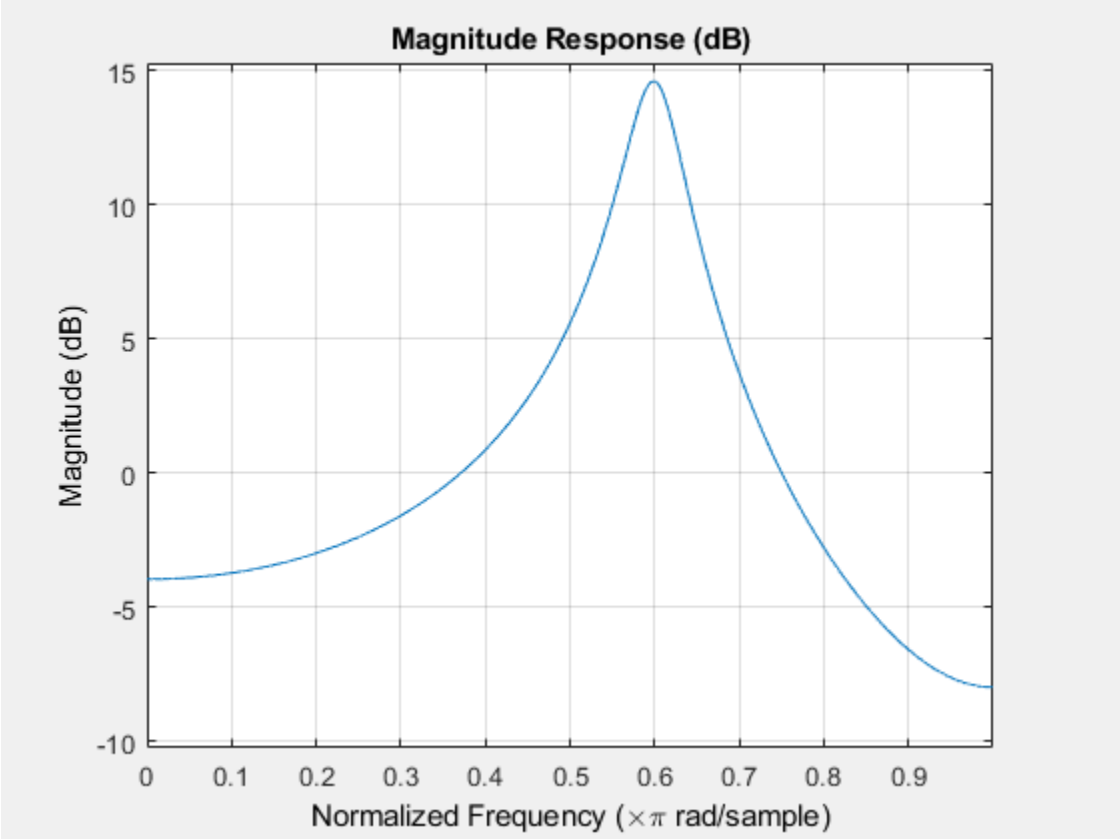
To view the pole-zero plot for this filter you can use `zplane`. Supply column vector arguments when the system is in pole-zero form.

```
zplane(zer,pol)
```



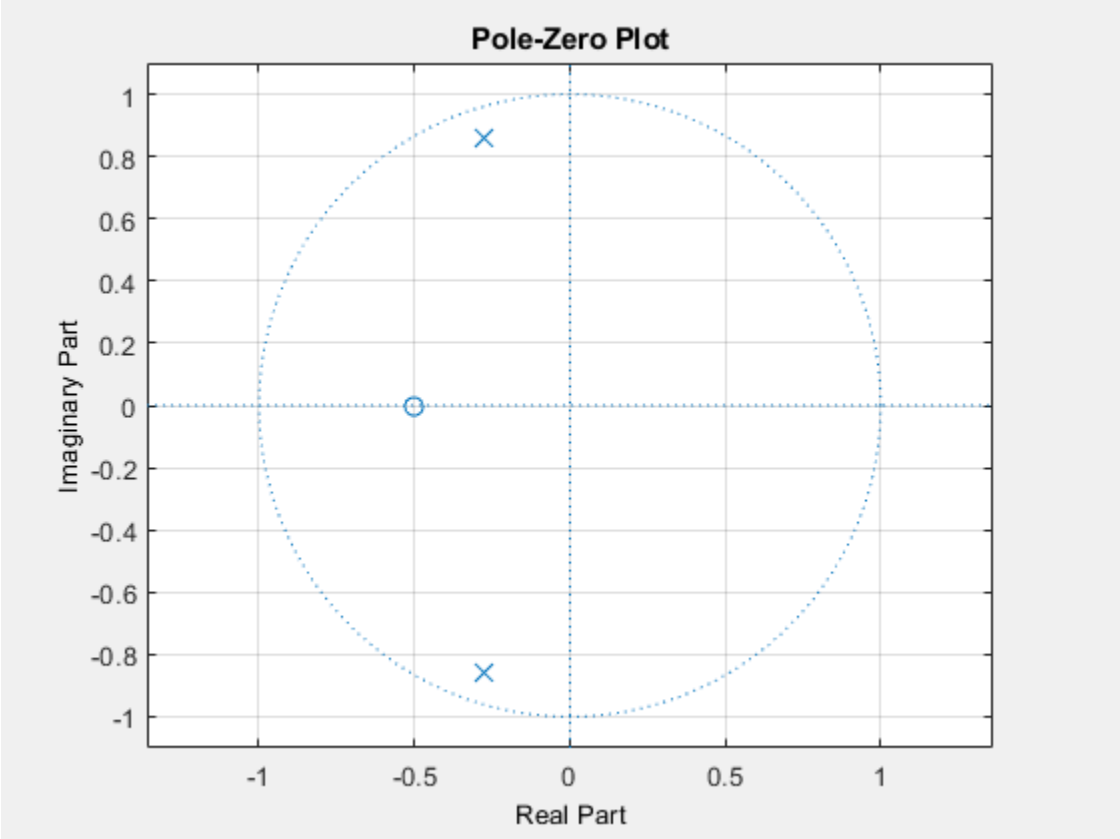

For access to additional tools, use `fvtool`. First convert the poles and zeros to transfer function form, then call `fvtool`.

```
[b,a] = zp2tf(zer,pol,1);
fvtool(b,a)
```



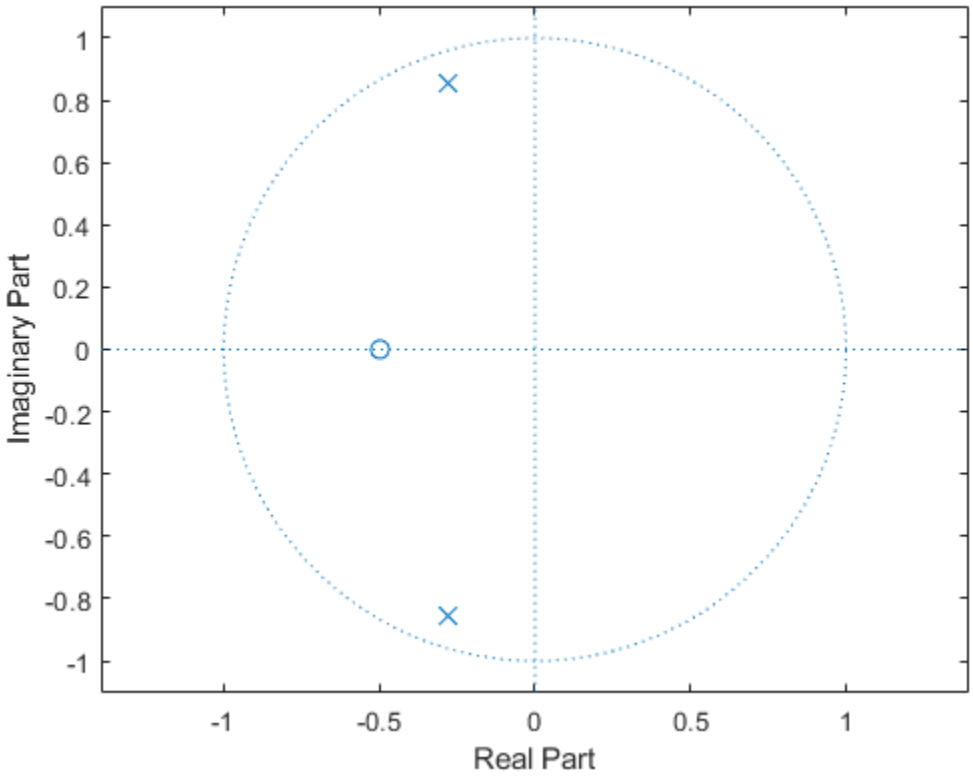

Click the **Pole/Zero Plot** toolbar button, select **Analysis** > **Pole/Zero Plot** from the menu, or type the following code to see the plot.

```
fvtool(b,a,'Analysis','polezero')
```




To use `zplane` for a system in transfer function form, supply row vector arguments. In this case, `zplane` finds the roots of the numerator and denominator using the `roots` function and plots the resulting zeros and poles.

```
zplane(b,a)
```




See “Discrete-Time System Models” on page 1-32 for details on zero-pole and transfer function representation of systems.

## See Also

**FVTool** | `zp2tf` | `zplane`

# Discrete-Time System Models

The discrete-time system models are representational schemes for digital filters. The MATLAB technical computing environment supports several discrete-time system models, which are described in the following sections:


- “Transfer Function” on page 1-32
- “Zero-Pole-Gain” on page 1-32
- “State Space” on page 1-33
- “Partial Fraction Expansion (Residue Form)” on page 1-33
- “Second-Order Sections (SOS)” on page 1-34
- “Lattice Structure” on page 1-35
- “Convolution Matrix” on page 1-36


## Transfer Function

The transfer function is a basic Z-domain representation of a digital filter, expressing the filter as a ratio of two polynomials. It is the principal discrete-time model for this toolbox. The transfer function model description for the Z-transform of a digital filter's difference equation is

$$Y(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \ldots + a(m+1)z^{-m}} X(z) .$$

Here, the constants $b(i)$ and $a(i)$ are the filter coefficients, and the order of the filter is the maximum of $n$ and $m$. In the MATLAB environment, you store these coefficients in two vectors (row vectors by convention), one row vector for the numerator and one for the denominator. See “Filters and Transfer Functions” on page 1-2 for more details on the transfer function form.

## Zero-Pole-Gain

The factored or zero-pole-gain form of a transfer function is

$$H(z) = \frac{q(z)}{p(z)} = k\frac{(z - q(1))(z - q(2))\ldots(z - q(n))}{(z - p(1))(z - p(2))\ldots(z - p(n))} .$$

By convention, polynomial coefficients are stored in row vectors and polynomial roots in column vectors. In zero-pole-gain form, therefore, the zero and pole locations for the numerator and denominator of a transfer function reside in column vectors. The factored transfer function gain $k$ is a MATLAB scalar.

The `poly` and `roots` functions convert between polynomial and zero-pole-gain representations. For example, a simple IIR filter is

```
b = [2 3 4];
a = [1 3 3 1];
```

The zeros and poles of this filter are

```
q = roots(b)
p = roots(a)
```

```
% Gain factor
k = b(1)/a(1)
```

Returning to the original polynomials,

```
bb = k*poly(q)
aa = poly(p)
```

Note that `b` and `a` in this case represent the transfer function:

$$H(z) = \frac{2 + 3z^{-1} + 4z^{-2}}{1 + 3z^{-1} + 3z^{-2} + z^{-3}} = \frac{z\,(2z^2 + 3z + 4)}{z^3 + 3z^2 + 3z + 1}.$$

For `b = [2 3 4]`, the `roots` function misses the zero for $z$ equal to 0. In fact, the function misses poles and zeros for $z$ equal to 0 whenever the input transfer function has more poles than zeros, or vice versa. This is acceptable in most cases. To circumvent the problem, however, simply append zeros to make the vectors the same length before using the `roots` function; for example, `b = [b 0]`.

## State Space

It is always possible to represent a digital filter, or a system of difference equations, as a set of first-order difference equations. In matrix or state-space form, you can write the equations as

$$\begin{aligned} x(n+1) &= Ax(n) + Bu(n) \\ y(n) &= Cx(n) + Du(n), \end{aligned}$$

where `u` is the input, `x` is the state vector, and `y` is the output. For single-channel systems, `A` is an `m`-by-`m` matrix where `m` is the order of the filter, `B` is a column vector, `C` is a row vector, and `D` is a scalar. State-space notation is especially convenient for multichannel systems where input `u` and output `y` become vectors, and `B`, `C`, and `D` become matrices.

State-space representation extends easily to the MATLAB environment. `A`, `B`, `C`, and `D` are rectangular arrays; MATLAB functions treat them as individual variables.

Taking the Z-transform of the state-space equations and combining them shows the equivalence of state-space and transfer function forms:

$$Y(z) = H(z)U(z), \text{ where } H(z) = C(zI - A)^{-1}B + D$$

Don't be concerned if you are not familiar with the state-space representation of linear systems. Some of the filter design algorithms use state-space form internally but do not require any knowledge of state-space concepts to use them successfully. If your applications use state-space based signal processing extensively, however, see the Control System Toolbox™ product for a comprehensive library of state-space tools.

## Partial Fraction Expansion (Residue Form)

Each transfer function also has a corresponding partial fraction expansion or *residue* form representation, given by

$$\frac{b(z)}{a(z)} = \frac{r(1)}{1 - p(1)z^{-1}} + \ldots + \frac{r(n)}{1 - p(n)z^{-1}} + k(1) + k(2)z^{-1} + \ldots + k(m - n + 1)z^{-(m-n)}$$

provided $H(z)$ has no repeated poles. Here, $n$ is the degree of the denominator polynomial of the rational transfer function $b(z)/a(z)$. If $r$ is a pole of multiplicity $s_r$, then $H(z)$ has terms of the form:

$$\frac{r(j)}{1-p(j)z^{-1}}+\frac{r(j+1)}{(1-p(j)z^{-1})^2}\ldots+\frac{r(j+s_r-1)}{(1-p(j)z^{-1})^{s_r}}$$

The Signal Processing Toolbox `residuez` function in converts transfer functions to and from the partial fraction expansion form. The "`z`" on the end of `residuez` stands for $z$-domain, or discrete domain. `residuez` returns the poles in a column vector `p`, the residues corresponding to the poles in a column vector `r`, and any improper part of the original transfer function in a row vector `k`. `residuez` determines that two poles are the same if the magnitude of their difference is smaller than 0.1 percent of either of the poles' magnitudes.

Partial fraction expansion arises in signal processing as one method of finding the inverse Z-transform of a transfer function. For example, the partial fraction expansion of

$$H(z)=\frac{-4+8z^{-1}}{1+6z^{-1}+8z^{-2}}$$

is

```
b = [-4 8];
a = [1 6 8];
[r,p,k] = residuez(b,a)
```

which corresponds to

$$H(z)=\frac{-12}{1+4z^{-1}}+\frac{8}{1+2z^{-1}}$$

To find the inverse Z-transform of $H(z)$, find the sum of the inverse Z-transforms of the two addends of $H(z)$, giving the causal impulse response:

$$h(n)=-12(-4)^n+8(-2)^n,\quad n=0,1,2,\ldots$$

To verify this in the MATLAB environment, type

```
imp = [1 0 0 0 0];
resptf = filter(b,a,imp)
respres = filter(r(1),[1 -p(1)],imp)+...
  filter(r(2),[1 -p(2)],imp)
```

## Second-Order Sections (SOS)

Any transfer function $H(z)$ has a second-order sections representation

$$H(z)=\prod_{k=1}^{L}H_k(z)=\prod_{k=1}^{L}\frac{b_{0k}+b_{1k}z^{-1}+b_{2k}z^{-2}}{a_{0k}+a_{1k}z^{-1}+a_{2k}z^{-2}}$$

where $L$ is the number of second-order sections that describe the system. The MATLAB environment represents the second-order section form of a discrete-time system as an $L$-by-6 array `sos`. Each row of `sos` contains a single second-order section, where the row elements are the three numerator and three denominator coefficients that describe the second-order section.

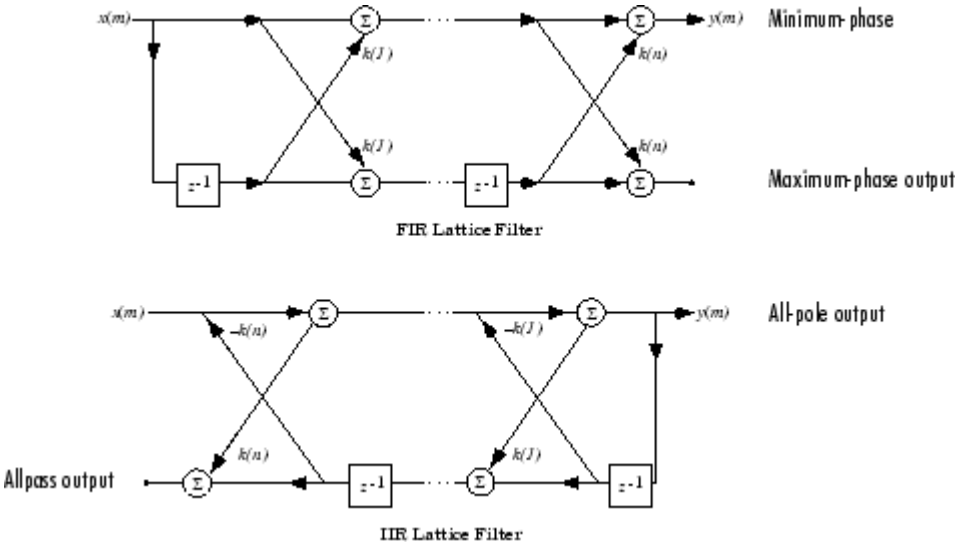
$$\text{sos} = \begin{pmatrix} b_{01} & b_{11} & b_{21} & a_{01} & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & a_{02} & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & a_{0L} & a_{1L} & a_{2L} \end{pmatrix}$$

There are many ways to represent a filter in second-order section form. Through careful pairing of the pole and zero pairs, ordering of the sections in the cascade, and multiplicative scaling of the sections, it is possible to reduce quantization noise gain and avoid overflow in some fixed-point filter implementations. The functions `zp2sos` and `ss2sos`, described in “Linear System Transformations” on page 1-39, perform pole-zero pairing, section scaling, and section ordering.

**Note** All Signal Processing Toolbox second-order section transformations apply only to digital filters.

## Lattice Structure

For a discrete $N$th order all-pole or all-zero filter described by the polynomial coefficients $a(n)$, $n = 1, 2, \ldots, N+1$, there are $N$ corresponding lattice structure coefficients $k(n)$, $n = 1, 2, \ldots, N$. The parameters $k(n)$ are also called the *reflection coefficients* of the filter. Given these reflection coefficients, you can implement a discrete filter as shown below.




**FIR and IIR Lattice Filter structure diagrams**

For a general pole-zero IIR filter described by polynomial coefficients $a$ and $b$, there are both lattice coefficients $k(n)$ for the denominator $a$ and ladder coefficients $v(n)$ for the numerator $b$. The lattice/ladder filter may be implemented as

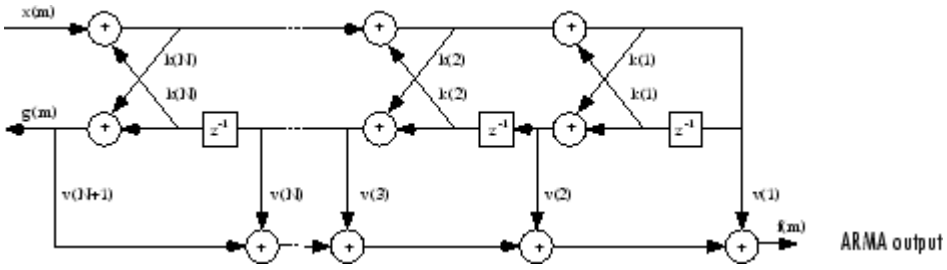



**Diagram of lattice/ladder filter**

The toolbox function `tf2latc` accepts an FIR or IIR filter in polynomial form and returns the corresponding reflection coefficients. An example FIR filter in polynomial form is

```
b = [1.0000   0.6149   0.9899   0.0000   0.0031  -0.0082];
```

This filter's lattice (reflection coefficient) representation is

```
k = tf2latc(b)
```

For IIR filters, the magnitude of the reflection coefficients provides an easy stability check. If all the reflection coefficients corresponding to a polynomial have magnitude less than 1, all of that polynomial's roots are inside the unit circle. For example, consider an IIR filter with numerator polynomial `b` from above and denominator polynomial:

```
a = [1 1/2 1/3];
```

The filter's lattice representation is

```
[k,v] = tf2latc(b,a);
```

Because `abs(k) < 1` for all reflection coefficients in `k`, the filter is stable.

The function `latc2tf` calculates the polynomial coefficients for a filter from its lattice (reflection) coefficients. Given the reflection coefficient vector `k`, the corresponding polynomial form is

```
b = latc2tf(k);
```

The lattice or lattice/ladder coefficients can be used to implement the filter using the function `latcfilt`.

## Convolution Matrix

In signal processing, convolving two vectors or matrices is equivalent to filtering one of the input operands by the other. This relationship permits the representation of a digital filter as a convolution matrix.

Given any vector, the toolbox function `convmtx` generates a matrix whose inner product with another vector is equivalent to the convolution of the two vectors. The generated matrix represents a digital filter that you can apply to any vector of appropriate length; the inner dimension of the operands must agree to compute the inner product.

The convolution matrix for a vector `b`, representing the numerator coefficients for a digital filter, is

```
b = [1 2 3];
x = randn(3,1);
C = convmtx(b',3);
```

Two equivalent ways to convolve b with x are as follows.

```
y1 = C*x;
y2 = conv(b,x);
```

# Continuous-Time System Models

The continuous-time system models are representational schemes for analog filters. Many of the discrete-time system models described earlier are also appropriate for the representation of continuous-time systems:

- State-space form
- Partial fraction expansion
- Transfer function
- Zero-pole-gain form

It is possible to represent any system of linear time-invariant differential equations as a set of first-order differential equations. In matrix or *state-space* form, you can express the equations as

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

where $u$ is a vector of $nu$ inputs, $x$ is an $nx$-element state vector, and $y$ is a vector of $ny$ outputs. In the MATLAB environment, `A`, `B`, `C`, and `D` are stored in separate rectangular arrays.

An equivalent representation of the state-space system is the Laplace transform transfer function description

$$Y(s) = H(s)U(s)$$

where

$$H(s) = C(sI - A)^{-1}B + D$$

For single-input, single-output systems, this form is given by

$$H(s) = \frac{b(s)}{a(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \ldots + b(n+1)}{a(1)s^m + a(2)s^{m-1} + \ldots + a(m+1)}$$

Given the coefficients of a Laplace transform transfer function, `residue` determines the partial fraction expansion of the system. See the description of `residue` for details.

The factored zero-pole-gain form is

$$H(s) = \frac{z(s)}{p(s)} = k\frac{(s - z(1))(s - z(2))\ldots(s - z(n))}{(s - p(1))(s - p(2))\ldots(s - p(m))}$$

As in the discrete-time case, the MATLAB environment stores polynomial coefficients in row vectors in descending powers of $s$. It stores polynomial roots, or zeros and poles, in column vectors.

# Linear System Transformations

A number of Signal Processing Toolbox functions are provided to convert between the various linear system models. You can use the following chart to find an appropriate transfer function: find the row of the model to convert *from* on the left side of the chart and the column of the model to convert *to* on the top of the chart and read the function name(s) at the intersection of the row and column. Note that some cells of this table are empty.

| **To →**<br><br>**From ↓** | **Transfer Function** | **State-Space** | **Zero- Pole-Gain** | **Partial Fraction** | **Lattice Filter** | **Second-Order Sections** | **Convolution Matrix** |
|---|---|---|---|---|---|---|---|
| **Transfer Function** | | `tf2ss` | `tf2zp`<br>`roots` | `residuez` | `tf2latc` | none | `convmtx` |
| **State-Space** | `ss2tf` | | `ss2zp` | none | none | `ss2sos` | none |
| **Zero-Pole-Gain** | `zp2tf` `poly` | `zp2ss` | | none | none | `zp2sos` | none |
| **Partial Fraction** | `residuez` | none | none | | none | none | none |
| **Lattice Filter** | `latc2tf` | none | none | none | | none | none |
| **SOS** | `sos2tf` | `sos2ss` | `sos2zp` | none | none | | none |

**Note** Converting from one filter structure or model to another may produce a result with different characteristics than the original. This is due to the computer's finite-precision arithmetic and the variations in the conversion's round-off computations.

Many of the toolbox filter design functions use these functions internally. For example, the `zp2ss` function converts the poles and zeros of an analog prototype into the state-space form required for creation of a Butterworth, Chebyshev, or elliptic filter. Once in state-space form, the filter design function performs any required frequency transformation, that is, it transforms the initial lowpass design into a bandpass, highpass, or bandstop filter, or a lowpass filter with the desired cutoff frequency.

**Note** All Signal Processing Toolbox second-order section transformations apply only to digital filters.

# Discrete Fourier Transform

The discrete Fourier transform, or DFT, is the primary tool of digital signal processing. The foundation of the product is the fast Fourier transform (FFT), a method for computing the DFT with reduced execution time. Many of the toolbox functions (including *Z*-domain frequency response, spectrum and cepstrum analysis, and some filter design and implementation functions) incorporate the FFT.

The MATLAB® environment provides the functions `fft` and `ifft` to compute the discrete Fourier transform and its inverse, respectively. For the input sequence $x$ and its transformed version $X$ (the discrete-time Fourier transform at equally spaced frequencies around the unit circle), the two functions implement the relationships

$$X(k+1) = \sum_{n=0}^{N-1} x(n+1) W_N^{kn}$$

and

$$x(n+1) = \frac{1}{N} \sum_{k=0}^{N-1} X(k+1) W_N^{-kn}.$$

In these equations, the series subscripts begin with 1 instead of 0 because of the MATLAB vector indexing scheme, and

$$W_N = e^{-j2\pi/N}.$$

**Note** The MATLAB convention is to use a negative $j$ for the `fft` function. This is an engineering convention; physics and pure mathematics typically use a positive $j$.

`fft`, with a single input argument, `x`, computes the DFT of the input vector or matrix. If `x` is a vector, `fft` computes the DFT of the vector; if `x` is a rectangular array, `fft` computes the DFT of each array column.

For example, create a time vector and signal:

```
t = 0:1/100:10-1/100;                    % Time vector
x = sin(2*pi*15*t) + sin(2*pi*40*t);     % Signal
```

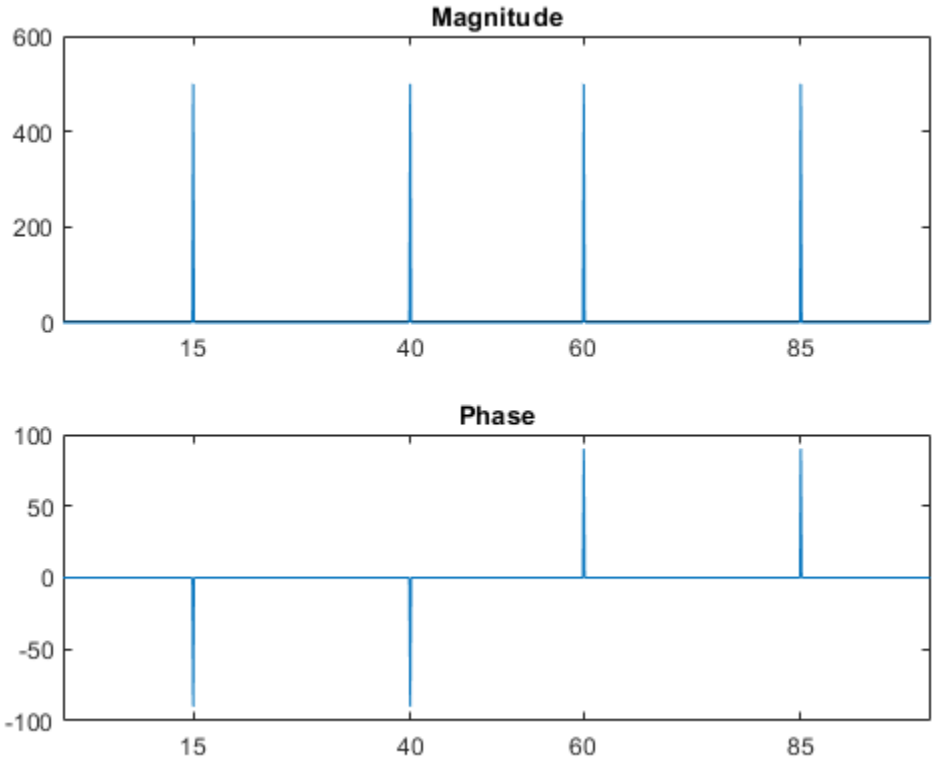
Compute the DFT of the signal and the magnitude and phase of the transformed sequence. Decrease round-off error when computing the phase by setting small-magnitude transform values to zero.

```
y = fft(x);                              % Compute DFT of x
m = abs(y);                              % Magnitude
y(m<1e-6) = 0;
p = unwrap(angle(y));                    % Phase
```

To plot the magnitude and phase in degrees, type the following commands:

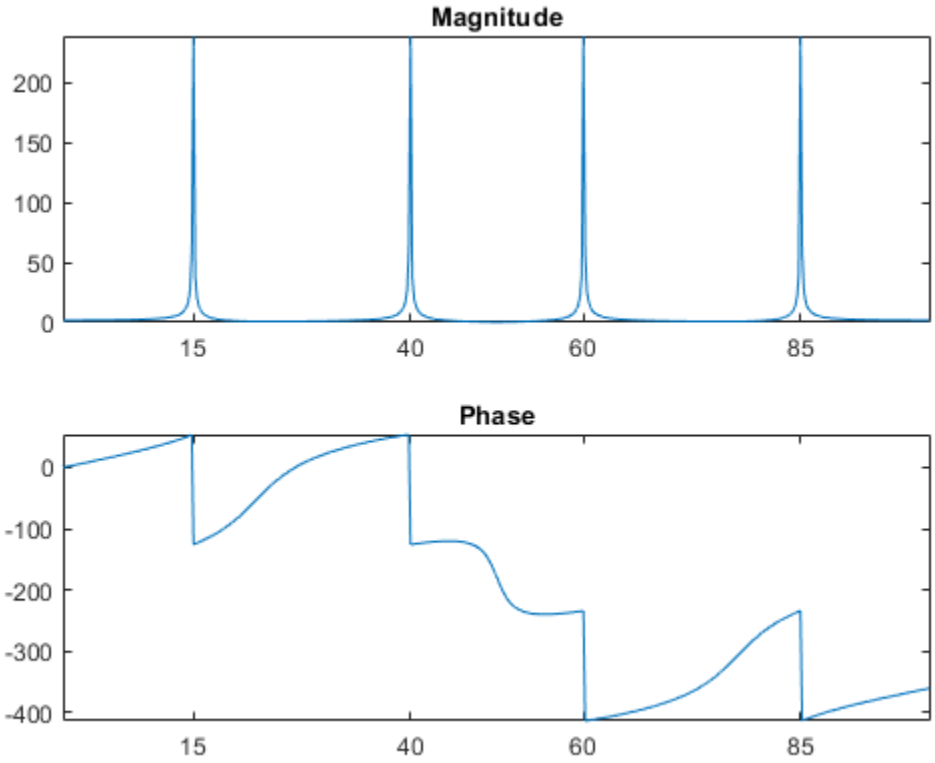
```
f = (0:length(y)-1)*100/length(y);       % Frequency vector

subplot(2,1,1)
plot(f,m)
title('Magnitude')
ax = gca;
```

```
ax.XTick = [15 40 60 85];

subplot(2,1,2)
plot(f,p*180/pi)
title('Phase')
ax = gca;
ax.XTick = [15 40 60 85];
```




A second argument to `fft` specifies a number of points `n` for the transform, representing DFT length:

```
n = 512;
y = fft(x,n);
m = abs(y);
p = unwrap(angle(y));
f = (0:length(y)-1)*100/length(y);

subplot(2,1,1)
plot(f,m)
title('Magnitude')
ax = gca;
ax.XTick = [15 40 60 85];

subplot(2,1,2)
plot(f,p*180/pi)
title('Phase')
ax = gca;
ax.XTick = [15 40 60 85];
```

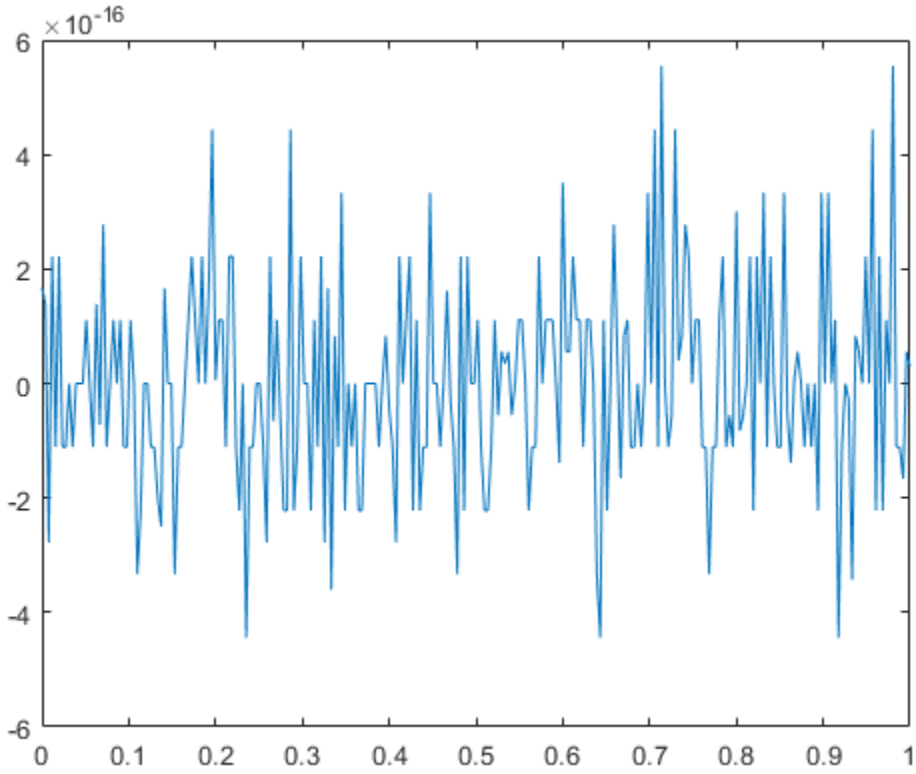



In this case, `fft` pads the input sequence with zeros if it is shorter than `n`, or truncates the sequence if it is longer than `n`. If `n` is not specified, it defaults to the length of the input sequence. Execution time for `fft` depends on the length, `n`, of the DFT it performs; see the `fft` reference page for details about the algorithm.

**Note** The resulting FFT amplitude is `A*n/2`, where `A` is the original amplitude and `n` is the number of FFT points. This is true only if the number of FFT points is greater than or equal to the number of data samples. If the number of FFT points is less, the FFT amplitude is lower than the original amplitude by the above amount.

The inverse discrete Fourier transform function `ifft` also accepts an input sequence and, optionally, the number of desired points for the transform. Try the example below; the original sequence `x` and the reconstructed sequence are identical (within rounding error).

```
t = 0:1/255:1;
x = sin(2*pi*120*t);
y = real(ifft(fft(x)));

figure
plot(t,x-y)
```




This toolbox also includes functions for the two-dimensional FFT and its inverse, `fft2` and `ifft2`. These functions are useful for two-dimensional signal or image processing. The goertzel function, which is another algorithm to compute the DFT, also is included in the toolbox. This function is efficient for computing the DFT of a portion of a long signal.

It is sometimes convenient to rearrange the output of the `fft` or `fft2` function so the zero frequency component is at the center of the sequence. The function `fftshift` moves the zero frequency component to the center of a vector or matrix.

## See Also

`fft` | `fft2` | `fftshift` | `goertzel` | `ifft` | `ifft2`

# Filter Design and Implementation


- “Filter Requirements and Specification” on page 2-2
- “IIR Filter Design” on page 2-4
- “FIR Filter Design” on page 2-16
- “Special Topics in IIR Filter Design” on page 2-33
- “Filtering Data With Signal Processing Toolbox Software” on page 2-39
- “Selected Bibliography” on page 2-55

# Filter Requirements and Specification

Filter design is the process of creating the filter coefficients to meet specific filtering requirements. Filter implementation involves choosing and applying a particular filter structure to those coefficients. Only after both design and implementation have been performed can data be filtered. The following chapter describes filter design and implementation in Signal Processing Toolbox™ software.

The goal of filter design is to perform frequency dependent alteration of a data sequence. A possible requirement might be to remove noise above 200 Hz from a data sequence sampled at 1000 Hz. A more rigorous specification might call for a specific amount of passband ripple, stopband attenuation, or transition width. A very precise specification could ask to achieve the performance goals with the minimum filter order, or it could call for an arbitrary magnitude shape, or it might require an FIR filter. Filter design methods differ primarily in how performance is specified.

To design a filter, the Signal Processing Toolbox software offers two approaches. The first approach uses the `designfilt` function. As an example, design and implement a 5th-order lowpass Butterworth filter with a 3-dB frequency of 200 Hz. Assume a sample rate of 1 kHz. Apply the filter to input data.

```
Fs = 1000;
fc = 200;
time = 0:1/Fs:1;
x = cos(2*pi*60*time)+sin(2*pi*120*time)+randn(size(time));

d = designfilt('lowpassiir','FilterOrder',5, ...
    'HalfPowerFrequency',fc,'DesignMethod','butter', ...
    'SampleRate',Fs);
yd = filter(d,x);
```

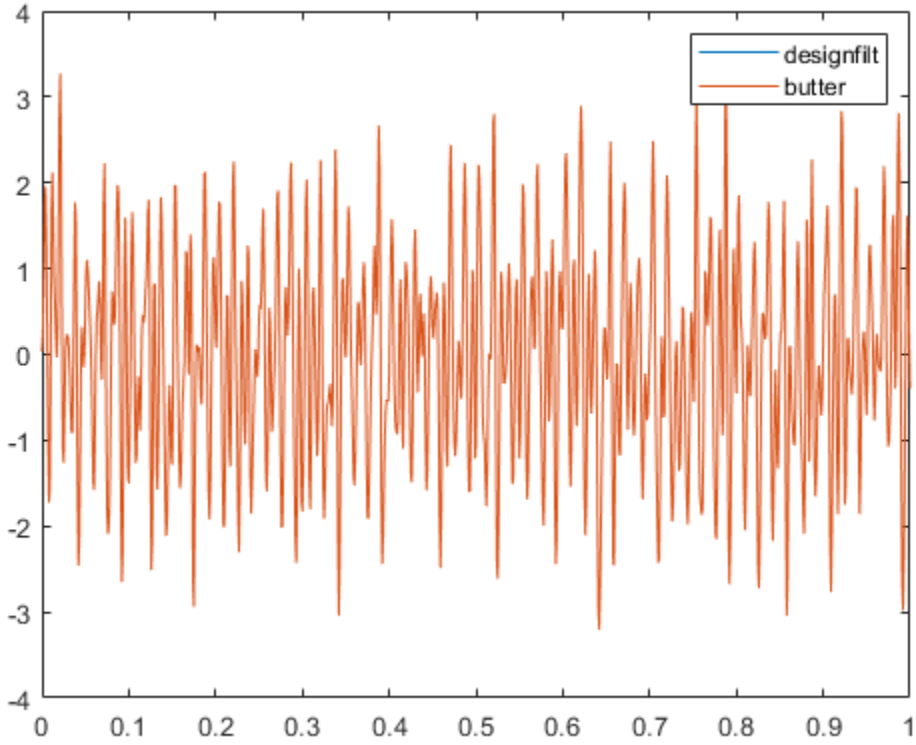
The other approach implements the filter using a function such as `butter` or `firpm`. All of these "classic" filter design functions operate with normalized frequencies. Convert frequency specifications in Hz to normalized frequency to use these functions. The Signal Processing Toolbox software defines normalized frequency to be in the closed interval [0,1], with 1 denoting $\pi$ rad/sample. For example, to specify a normalized frequency of $\pi/2$ rad/sample, enter 0.5.

To convert from Hz to normalized frequency, multiply the frequency in Hz by two and divide by the sampling frequency. For example, design a 5th-order lowpass Butterworth filter with a 3-dB frequency of 200 Hz using `butter`.

```
Wn = fc/(Fs/2);

[b,a] = butter(5,Wn,'low');
yb = filter(b,a,x);
```

Plot the two filtered signals.

```
plot(time,yd,time,yb)
legend('designfilt','butter')
```




## See Also

`butter` | `designfilt` | `filter`

# IIR Filter Design

| In this section... |
| --- |
| “IIR vs. FIR Filters” on page 2-4 |
| “Classical IIR Filters” on page 2-4 |
| “Other IIR Filters” on page 2-4 |
| “IIR Filter Method Summary” on page 2-4 |
| “Classical IIR Filter Design Using Analog Prototyping” on page 2-5 |
| “Comparison of Classical IIR Filter Types” on page 2-7 |

## IIR vs. FIR Filters

The primary advantage of IIR filters over FIR filters is that they typically meet a given set of specifications with a much lower filter order than a corresponding FIR filter. Although IIR filters have nonlinear phase, data processing within MATLAB software is commonly performed “offline,” that is, the entire data sequence is available prior to filtering. This allows for a noncausal, zero-phase filtering approach (via the `filtfilt` function), which eliminates the nonlinear phase distortion of an IIR filter.

## Classical IIR Filters

The classical IIR filters, Butterworth, Chebyshev Types I and II, elliptic, and Bessel, all approximate the ideal “brick wall” filter in different ways.

This toolbox provides functions to create all these types of classical IIR filters in both the analog and digital domains (except Bessel, for which only the analog case is supported), and in lowpass, highpass, bandpass, and bandstop configurations. For most filter types, you can also find the lowest filter order that fits a given filter specification in terms of passband and stopband attenuation, and transition width(s).

## Other IIR Filters

The direct filter design function `yulewalk` finds a filter with magnitude response approximating a specified frequency-response function. This is one way to create a multiband bandpass filter.

You can also use the parametric modeling or system identification functions to design IIR filters. These functions are discussed in “Parametric Modeling” on page 8-18.

The generalized Butterworth design function `maxflat` is discussed in the section “Generalized Butterworth Filter Design” on page 2-14.

## IIR Filter Method Summary

The following table summarizes the various filter methods in the toolbox and lists the functions available to implement these methods.

**Toolbox Filters Methods and Available Functions**

| Filter Method | Description | Filter Functions |
|---|---|---|
| Analog Prototyping | Using the poles and zeros of a classical lowpass prototype filter in the continuous (Laplace) domain, obtain a digital filter through frequency transformation and filter discretization. | Complete design functions:<br>`besself`, `butter`, `cheby1`, `cheby2`, `ellip`<br><br>Order estimation functions:<br>`buttord`, `cheb1ord`, `cheb2ord`, `ellipord`<br><br>Lowpass analog prototype functions:<br>`besselap`, `buttap`, `cheb1ap`, `cheb2ap`, `ellipap`<br><br>Frequency transformation functions:<br>`lp2bp`, `lp2bs`, `lp2hp`, `lp2lp`<br><br>Filter discretization functions:<br>`bilinear`, `impinvar` |
| Direct Design | Design digital filter directly in the discrete time-domain by approximating a piecewise linear magnitude response. | `yulewalk` |
| Generalized Butterworth Design | Design lowpass Butterworth filters with more zeros than poles. | `maxflat` |
| Parametric Modeling | Find a digital filter that approximates a prescribed time or frequency domain response. (See System Identification Toolbox™ documentation for an extensive collection of parametric modeling tools.) | Time-domain modeling functions:<br>`lpc`, `prony`, `stmcb`<br><br>Frequency-domain modeling functions:<br>`invfreqs`, `invfreqz` |

## Classical IIR Filter Design Using Analog Prototyping

The principal IIR digital filter design technique this toolbox provides is based on the conversion of classical lowpass analog filters to their digital equivalents. The following sections describe how to design filters and summarize the characteristics of the supported filter types. See “Special Topics in IIR Filter Design” on page 2-33 for detailed steps on the filter design process.

### Complete Classical IIR Filter Design

You can easily create a filter of any order with a lowpass, highpass, bandpass, or bandstop configuration using the filter design functions.

### Filter Design Functions

| Filter Type | Design Function |
|---|---|
| Bessel (analog only) | `[b,a] = besself(n,Wn,options)`<br>`[z,p,k] = besself(n,Wn,options)`<br>`[A,B,C,D] = besself(n,Wn,options)` |
| Butterworth | `[b,a] = butter(n,Wn,options)`<br>`[z,p,k] = butter(n,Wn,options)`<br>`[A,B,C,D] = butter(n,Wn,options)` |
| Chebyshev Type I | `[b,a] = cheby1(n,Rp,Wn,options)`<br>`[z,p,k] = cheby1(n,Rp,Wn,options)`<br>`[A,B,C,D] = cheby1(n,Rp,Wn,options)` |
| Chebyshev Type II | `[b,a] = cheby2(n,Rs,Wn,options)`<br>`[z,p,k] = cheby2(n,Rs,Wn,options)`<br>`[A,B,C,D] = cheby2(n,Rs,Wn,options)` |
| Elliptic | `[b,a] = ellip(n,Rp,Rs,Wn,options)`<br>`[z,p,k] = ellip(n,Rp,Rs,Wn,options)`<br>`[A,B,C,D] = ellip(n,Rp,Rs,Wn,options)` |

By default, each of these functions returns a lowpass filter; you need to specify only the cutoff frequency that you want, `Wn`, in normalized units such that the Nyquist frequency is 1 Hz). For a highpass filter, append `'high'` to the function's parameter list. For a bandpass or bandstop filter, specify `Wn` as a two-element vector containing the passband edge frequencies. Append `'stop'` for the bandstop configuration.

Here are some example digital filters:

```
[b,a] = butter(5,0.4);                    % Lowpass Butterworth
[b,a] = cheby1(4,1,[0.4 0.7]);            % Bandpass Chebyshev Type I
[b,a] = cheby2(6,60,0.8,'high');          % Highpass Chebyshev Type II
[b,a] = ellip(3,1,60,[0.4 0.7],'stop');   % Bandstop elliptic
```

To design an analog filter, perhaps for simulation, use a trailing `'s'` and specify cutoff frequencies in rad/s:

```
[b,a] = butter(5,0.4,'s');      % Analog Butterworth filter
```

All filter design functions return a filter in the transfer function, zero-pole-gain, or state-space linear system model representation, depending on how many output arguments are present. In general, you should avoid using the transfer function form because numerical problems caused by round-off errors can occur. Instead, use the zero-pole-gain form which you can convert to a second-order section (SOS) form using `zp2sos` and then use the SOS form to analyze or implement your filter.

**Note** All classical IIR lowpass filters are ill-conditioned for extremely low cutoff frequencies. Therefore, instead of designing a lowpass IIR filter with a very narrow passband, it can be better to design a wider passband and decimate the input signal.

### Designing IIR Filters to Frequency Domain Specifications

This toolbox provides order selection functions that calculate the minimum filter order that meets a given set of requirements.

| Filter Type | Order Estimation Function |
|---|---|
| Butterworth | `[n,Wn] = buttord(Wp,Ws,Rp,Rs)` |
| Chebyshev Type I | `[n,Wn] = cheb1ord(Wp,Ws,Rp,Rs)` |
| Chebyshev Type II | `[n,Wn] = cheb2ord(Wp,Ws,Rp,Rs)` |
| Elliptic | `[n,Wn] = ellipord(Wp,Ws,Rp,Rs)` |

These are useful in conjunction with the filter design functions. Suppose you want a bandpass filter with a passband from 1000 to 2000 Hz, stopbands starting 500 Hz away on either side, a 10 kHz sampling frequency, at most 1 dB of passband ripple, and at least 60 dB of stopband attenuation. You can meet these specifications by using the `butter` function as follows.

```
[n,Wn] = buttord([1000 2000]/5000,[500 2500]/5000,1,60)
[b,a] = butter(n,Wn);

n =
    12
Wn =
    0.1951    0.4080
```

An elliptic filter that meets the same requirements is given by

```
[n,Wn] = ellipord([1000 2000]/5000,[500 2500]/5000,1,60)
[b,a] = ellip(n,1,60,Wn);

n =
    5
Wn =
    0.2000    0.4000
```

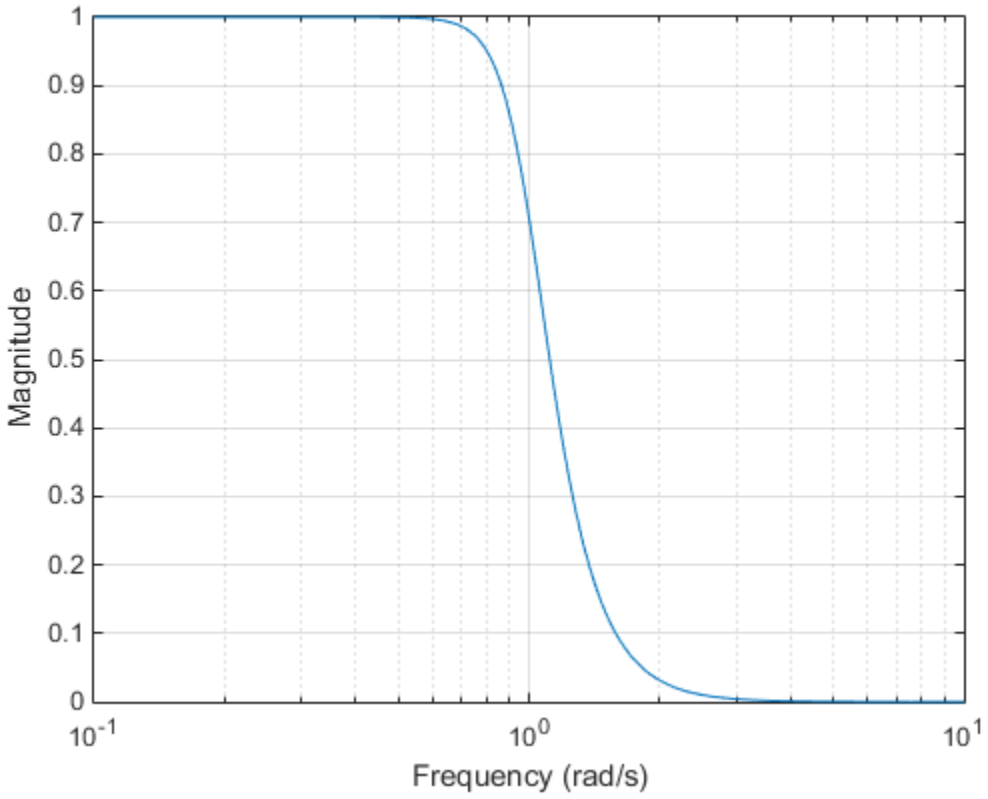
These functions also work with the other standard band configurations, as well as for analog filters.

## Comparison of Classical IIR Filter Types

The toolbox provides five different types of classical IIR filters, each optimal in some way. This section shows the basic analog prototype form for each and summarizes major characteristics.

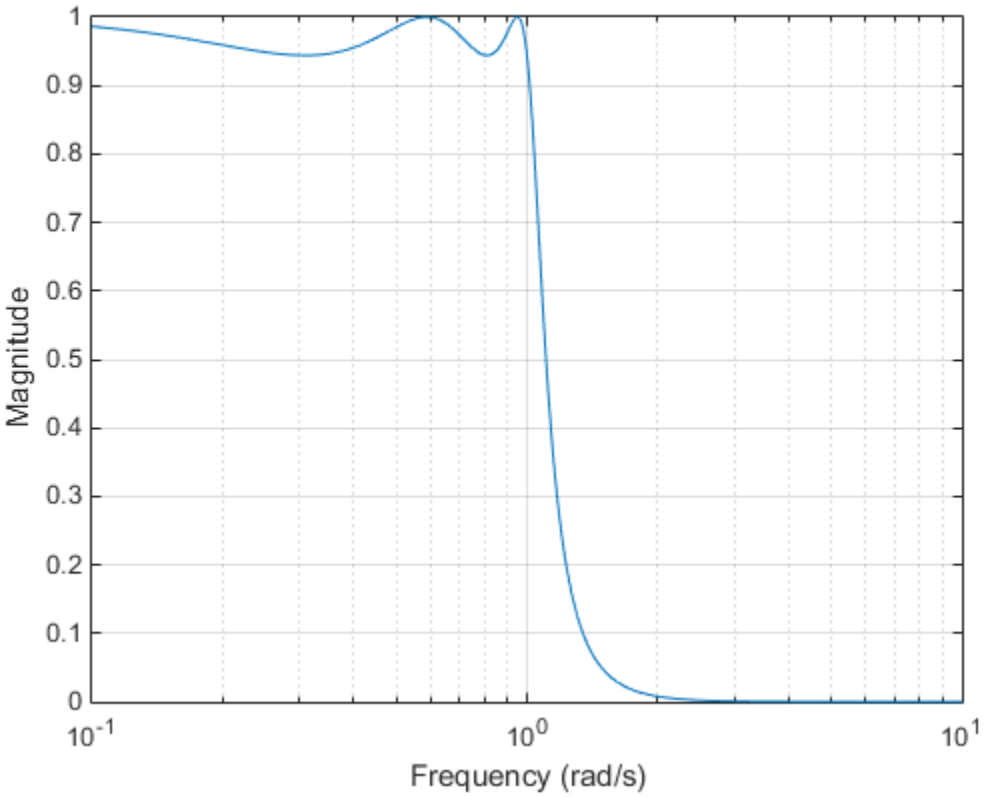
### Butterworth Filter

The Butterworth filter provides the best Taylor series approximation to the ideal lowpass filter response at analog frequencies $\Omega = 0$ and $\Omega = \infty$; for any order $N$, the magnitude squared response has $2N - 1$ zero derivatives at these locations (*maximally flat* at $\Omega = 0$ and $\Omega = \infty$). Response is monotonic overall, decreasing smoothly from $\Omega = 0$ to $\Omega = \infty$. $|H(j\Omega)| = 1/\sqrt{2}$ at $\Omega = 1$.




### Chebyshev Type I Filter

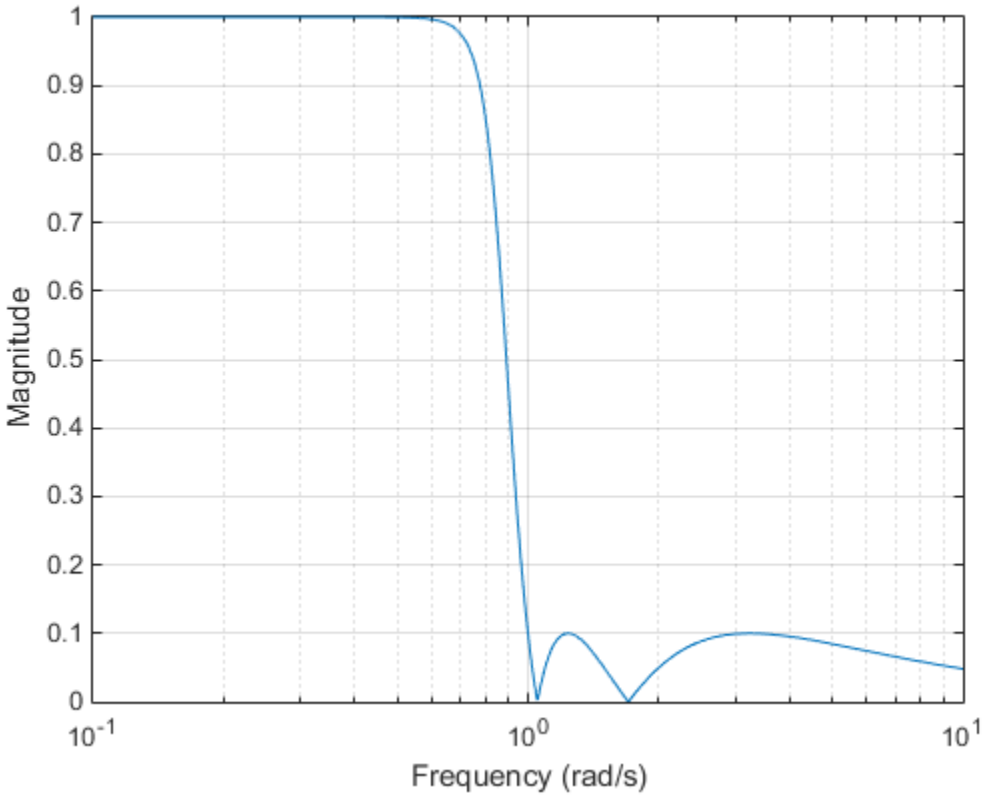
The Chebyshev Type I filter minimizes the absolute difference between the ideal and actual frequency response over the entire passband by incorporating an equal ripple of Rp dB in the passband. Stopband response is maximally flat. The transition from passband to stopband is more rapid than for the Butterworth filter. $|H(j\Omega)| = 10^{-\mathrm{Rp}/20}$ at $\Omega = 1$.




### Chebyshev Type II Filter

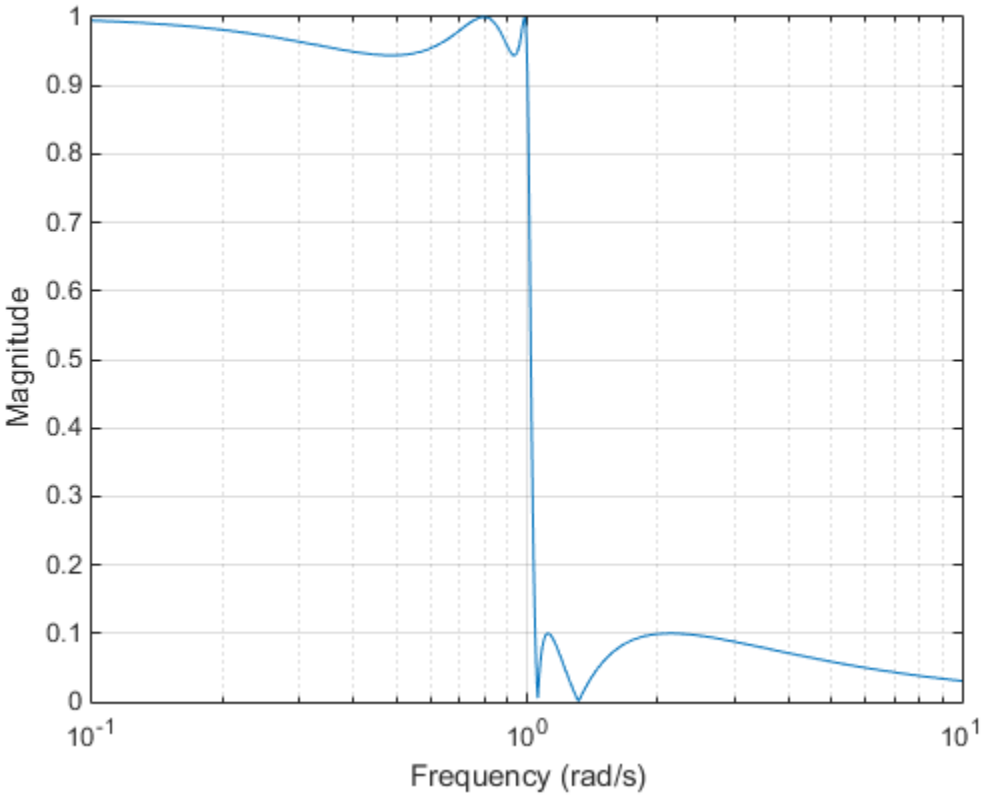
The Chebyshev Type II filter minimizes the absolute difference between the ideal and actual frequency response over the entire stopband by incorporating an equal ripple of `Rs` dB in the stopband. Passband response is maximally flat.

The stopband does not approach zero as quickly as the type I filter (and does not approach zero at all for even-valued filter order $n$). The absence of ripple in the passband, however, is often an important advantage. $|H(j\Omega)| = 10^{-\mathrm{Rs}/20}$ at $\Omega = 1$.




**Elliptic Filter**

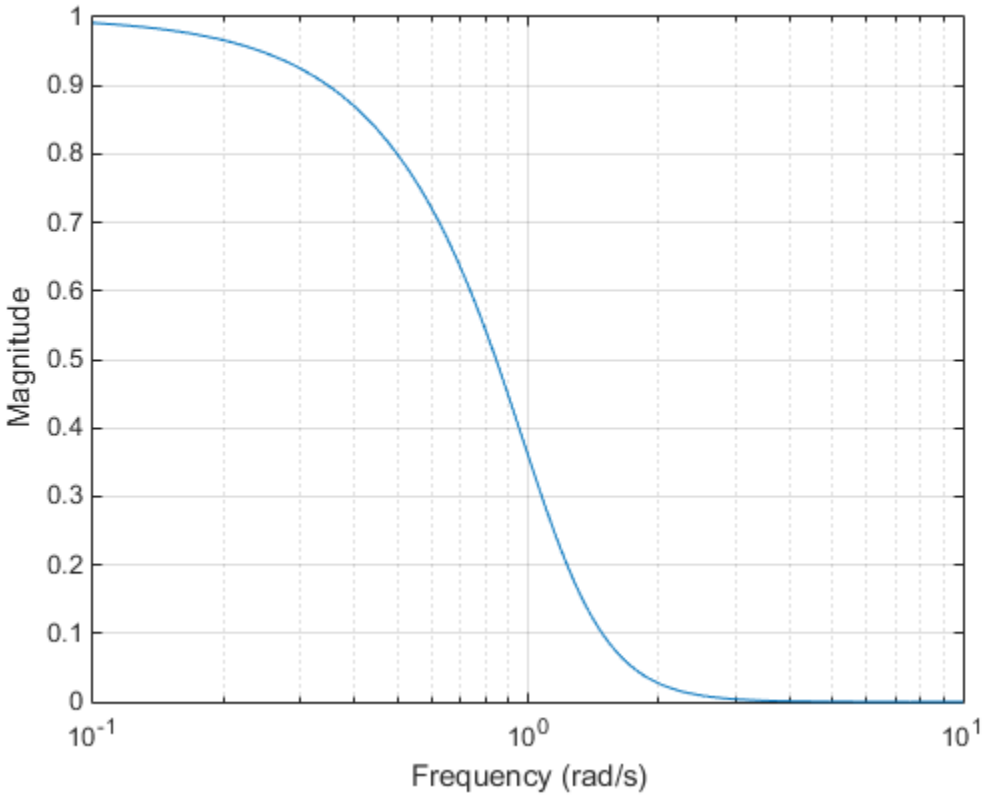
Elliptic filters are equiripple in both the passband and stopband. They generally meet filter requirements with the lowest order of any supported filter type. Given a filter order $n$, passband ripple `Rp` in decibels, and stopband ripple `Rs` in decibels, elliptic filters minimize transition width. $|H(j\Omega)| = 10^{-\mathrm{Rp}/20}$ at $\Omega = 1$.




**Bessel Filter**

Analog Bessel lowpass filters have maximally flat group delay at zero frequency and retain nearly constant group delay across the entire passband. Filtered signals therefore maintain their waveshapes in the passband frequency range. When an analog Bessel lowpass filter is converted to a digital one through frequency mapping, it no longer has this maximally flat property. Signal Processing Toolbox supports only the analog case for the complete Bessel filter design function.

Bessel filters generally require a higher filter order than other filters for satisfactory stopband attenuation. $|H(j\Omega)| < 1/\sqrt{2}$ at $\Omega = 1$ and decreases as filter order $n$ increases.




**Note** The lowpass filters shown above were created with the analog prototype functions `besselap`, `buttap`, `cheb1ap`, `cheb2ap`, and `ellipap`. These functions find the zeros, poles, and gain of an nth-order analog filter of the appropriate type with a cutoff frequency of 1 rad/s. The complete filter design functions (`besself`, `butter`, `cheby1`, `cheby2`, and `ellip`) call the prototyping functions as a first step in the design process. See “Special Topics in IIR Filter Design” on page 2-33 for details.

To create similar plots, use `n = 5` and, as needed, `Rp = 0.5` and `Rs = 20`. For example, to create the elliptic filter plot:

```
[z,p,k] = ellipap(5,0.5,20);
w = logspace(-1,1,1000);
h = freqs(k*poly(z),poly(p),w);
semilogx(w,abs(h)), grid
xlabel('Frequency (rad/s)')
ylabel('Magnitude')
```

### Direct IIR Filter Design

This toolbox uses the term *direct methods* to describe techniques for IIR design that find a filter based on specifications in the discrete domain. Unlike the analog prototyping method, direct design methods are not constrained to the standard lowpass, highpass, bandpass, or bandstop configurations. Rather, these functions design filters with an arbitrary, perhaps multiband, frequency response. This section discusses the `yulewalk` function, which is intended specifically for filter design; “Parametric Modeling” on page 8-18 discusses other methods that may also be considered

direct, such as Prony's method, Linear Prediction, the Steiglitz-McBride method, and inverse frequency design.

The `yulewalk` function designs recursive IIR digital filters by fitting a specified frequency response. `yulewalk`'s name reflects its method for finding the filter's denominator coefficients: it finds the inverse FFT of the ideal specified magnitude-squared response and solves the modified Yule-Walker equations using the resulting autocorrelation function samples. The statement

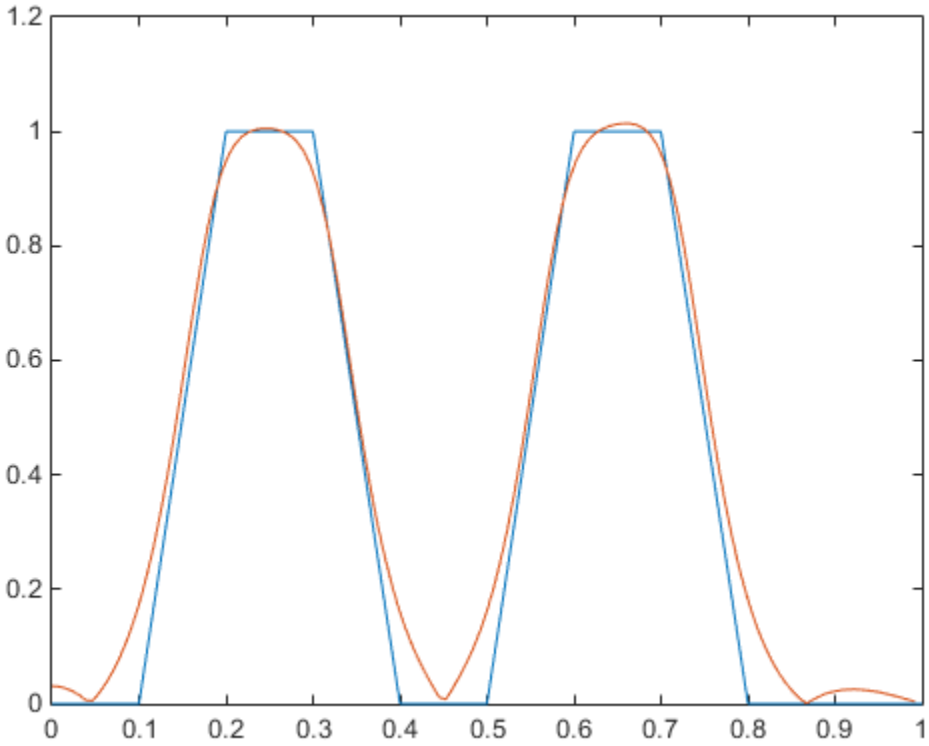
```
[b,a] = yulewalk(n,f,m)
```

returns row vectors `b` and `a` containing the `n+1` numerator and denominator coefficients of the `n`th-order IIR filter whose frequency-magnitude characteristics approximate those given in vectors `f` and `m`. `f` is a vector of frequency points ranging from 0 to 1, where 1 represents the Nyquist frequency. `m` is a vector containing the specified magnitude response at the points in `f`. `f` and `m` can describe any piecewise linear shape magnitude response, including a multiband response. The FIR counterpart of this function is `fir2`, which also designs a filter based on an arbitrary piecewise linear magnitude response. See “FIR Filter Design” on page 2-16 for details.

Note that `yulewalk` does not accept phase information, and no statements are made about the optimality of the resulting filter.

Design a multiband filter with `yulewalk` and plot the specified and actual frequency response:

```
m = [0   0   1   1   0   0   1   1   0 0];
f = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 1];
[b,a] = yulewalk(10,f,m);
[h,w] = freqz(b,a,128)
plot(f,m,w/pi,abs(h))
```




### Generalized Butterworth Filter Design

The toolbox function `maxflat` enables you to design generalized Butterworth filters, that is, Butterworth filters with differing numbers of zeros and poles. This is desirable in some implementations where poles are more expensive computationally than zeros. `maxflat` is just like the `butter` function, except that it you can specify *two* orders (one for the numerator and one for the denominator) instead of just one. These filters are *maximally flat*. This means that the resulting filter is optimal for any numerator and denominator orders, with the maximum number of derivatives at 0 and the Nyquist frequency $\omega = \pi$ both set to 0.

For example, when the two orders are the same, `maxflat` is the same as `butter`:

```
[b,a] = maxflat(3,3,0.25)
```

```
b =
    0.0317    0.0951    0.0951    0.0317
a =
    1.0000   -1.4590    0.9104   -0.1978
```

```
[b,a] = butter(3,0.25)
```

```
b =
    0.0317    0.0951    0.0951    0.0317
a =
    1.0000   -1.4590    0.9104   -0.1978
```

However, `maxflat` is more versatile because it allows you to design a filter with more zeros than poles:

```
[b,a] = maxflat(3,1,0.25)
```

```
b =
    0.0950    0.2849    0.2849    0.0950
a =
    1.0000   -0.2402
```

The third input to `maxflat` is the *half-power frequency*, a frequency between 0 and 1 with a magnitude response of $1/\sqrt{2}$.

You can also design linear phase filters that have the maximally flat property using the `'sym'` option:

```
maxflat(4,'sym',0.3)
```

```
ans =
    0.0331    0.2500    0.4337    0.2500    0.0331
```

For complete details of the `maxflat` algorithm, see Selesnick and Burrus [2].

# FIR Filter Design

| In this section... |
| --- |
| “FIR vs. IIR Filters” on page 2-16 |
| “FIR Filter Summary” on page 2-16 |
| “Linear Phase Filters” on page 2-17 |
| “Windowing Method” on page 2-17 |
| “Multiband FIR Filter Design with Transition Bands” on page 2-20 |
| “Constrained Least Squares FIR Filter Design” on page 2-24 |
| “Arbitrary-Response Filter Design” on page 2-28 |

## FIR vs. IIR Filters

Digital filters with finite-duration impulse response (all-zero, or FIR filters) have both advantages and disadvantages compared to infinite-duration impulse response (IIR) filters.

FIR filters have the following primary advantages:

- They can have exactly linear phase.
- They are always stable.
- The design methods are generally linear.
- They can be realized efficiently in hardware.
- The filter startup transients have finite duration.

The primary disadvantage of FIR filters is that they often require a much higher filter order than IIR filters to achieve a given level of performance. Correspondingly, the delay of these filters is often much greater than for an equal performance IIR filter.

## FIR Filter Summary

### FIR Filters

| Filter Design Method | Description | Filter Functions |
| --- | --- | --- |
| Windowing | Apply window to truncated inverse Fourier transform of specified "brick wall" filter | `fir1`, `fir2`, `kaiserord` |
| Multiband with Transition Bands | Equiripple or least squares approach over sub-bands of the frequency range | `firls`, `firpm`, `firpmord` |
| Constrained Least Squares | Minimize squared integral error over entire frequency range subject to maximum error constraints | `fircls`, `fircls1` |
| Arbitrary Response | Arbitrary responses, including nonlinear phase and complex filters | `cfirpm` |
| Raised Cosine | Lowpass response with smooth, sinusoidal transition | `rcosdesign` |

## Linear Phase Filters

Except for `cfirpm`, all of the FIR filter design functions design linear phase filters only. The filter coefficients, or "taps," of such filters obey either an even or odd symmetry relation. Depending on this symmetry, and on whether the order $n$ of the filter is even or odd, a linear phase filter (stored in length $n$+1 vector `b`) has certain inherent restrictions on its frequency response.

| Linear Phase Filter Type | Filter Order | Symmetry of Coefficients | Response H(f), f = 0 | Response H(f), f = 1 (Nyquist) |
|---|---|---|---|---|
| Type I | Even | even: $b(k) = b(n+2-k), \quad k = 1, \ldots, n+1$ | No restriction | No restriction |
| Type II | Odd | even: $b(k) = b(n+2-k), \quad k = 1, \ldots, n+1$ | No restriction | $H(1) = 0$ |
| Type III | Even | odd: $b(k) = -b(n+2-k), \quad k = 1, \ldots, n+1$ | $H(0) = 0$ | $H(1) = 0$ |
| Type IV | Odd | odd: $b(k) = -b(n+2-k), \quad k = 1, \ldots, n+1$ | $H(0) = 0$ | No restriction |

The phase delay and group delay of linear phase FIR filters are equal and constant over the frequency band. For an order $n$ linear phase FIR filter, the group delay is $n/2$, and the filtered signal is simply delayed by $n/2$ time steps (and the magnitude of its Fourier transform is scaled by the filter's magnitude response). This property preserves the wave shape of signals in the passband; that is, there is no phase distortion.

The functions `fir1`, `fir2`, `firls`, `firpm`, `fircls`, and `fircls1` all design type I and II linear phase FIR filters by default. `rcosdesign` designs only type I filters. Both `firls` and `firpm` design type III and IV linear phase FIR filters given a `'hilbert'` or `'differentiator'` flag. `cfirpm` can design any type of linear phase filter, and nonlinear phase filters as well.

**Note** Because the frequency response of a type II filter is zero at the Nyquist frequency ("high" frequency), `fir1` does not design type II highpass and bandstop filters. For odd-valued `n` in these cases, `fir1` adds 1 to the order and returns a type I filter.

## Windowing Method

Consider the ideal, or "brick wall," digital lowpass filter with a cutoff frequency of $\omega_0$ rad/s. This filter has magnitude 1 at all frequencies with magnitude less than $\omega_0$, and magnitude 0 at frequencies with magnitude between $\omega_0$ and π. Its impulse response sequence $h(n)$ is

$$h(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H(\omega)e^{j\omega n}d\omega = \frac{1}{2\pi}\int_{-\omega_0}^{\omega_0} e^{j\omega n}d\omega = \frac{\sin\omega_0 n}{\pi n}$$

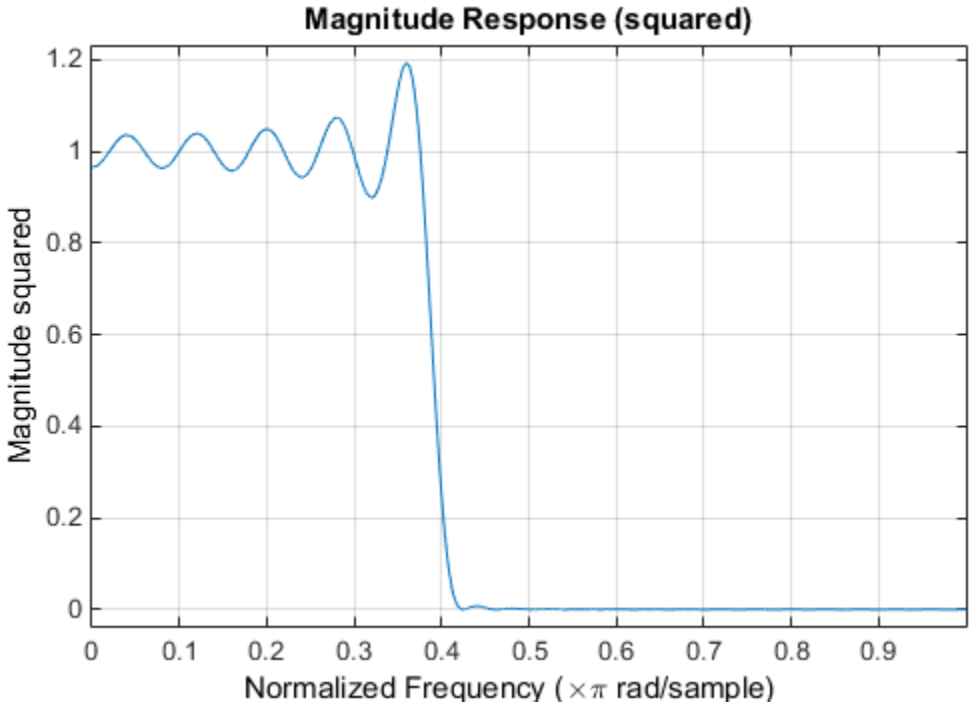
This filter is not implementable since its impulse response is infinite and noncausal. To create a finite-duration impulse response, truncate it by applying a window. By retaining the central section of impulse response in this truncation, you obtain a linear phase FIR filter. For example, a length 51 filter with a lowpass cutoff frequency $\omega_0$ of 0.4 π rad/s is

```
b = 0.4*sinc(0.4*(-25:25));
```

The window applied here is a simple rectangular window. By Parseval's theorem, this is the length 51 filter that best approximates the ideal lowpass filter, in the integrated least squares sense. The following command displays the filter's frequency response in FVTool:

```
fvtool(b,1)
```

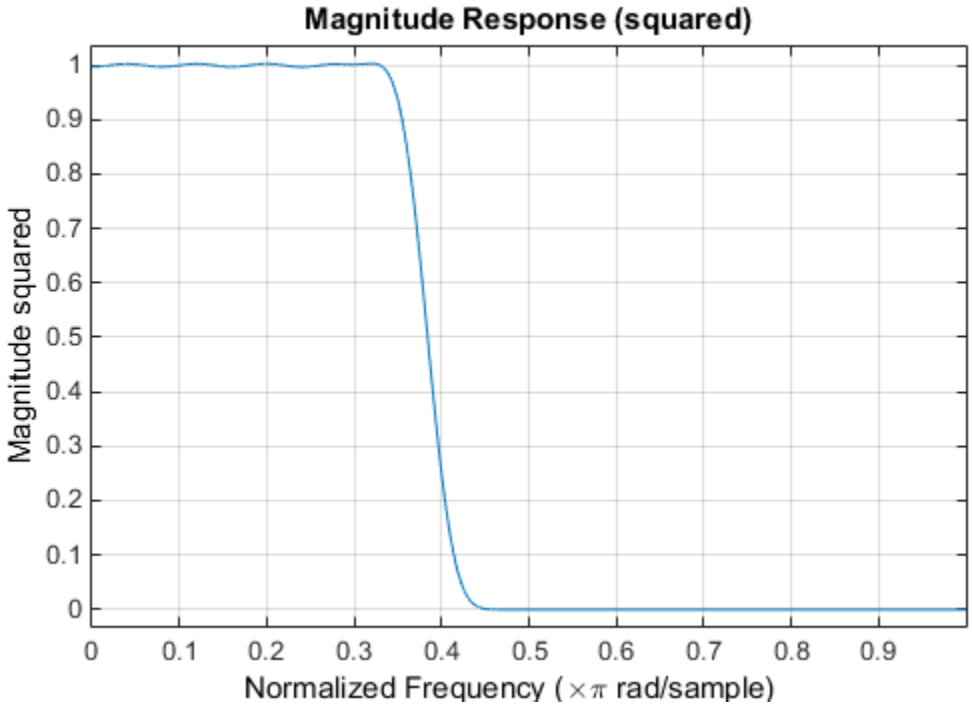
Note that the *y*-axis shown in the figure below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.




Ringing and ripples occur in the response, especially near the band edge. This "Gibbs effect" does not vanish as the filter length increases, but a nonrectangular window reduces its magnitude. Multiplication by a window in the time domain causes a convolution or smoothing in the frequency domain. Apply a length 51 Hamming window to the filter and display the result using FVTool:

```
b = 0.4*sinc(0.4*(-25:25));
b = b.*hamming(51)';
fvtool(b,1)
```

Note that the *y*-axis shown in the figure below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.




Using a Hamming window greatly reduces the ringing. This improvement is at the expense of transition width (the windowed version takes longer to ramp from passband to stopband) and optimality (the windowed version does not minimize the integrated squared error).

### Standard Band FIR Filter Design: fir1

`fir1` uses a least-squares approximation to compute filter coefficients and then smooths the impulse response with a window. For an overview of windows and their properties, see “Windows” on page 8-2. `fir1` resembles the IIR filter design functions in that it is formulated to design filters in standard band configurations: lowpass, bandpass, highpass, and bandstop.

The statements

```
n = 50;
Wn = 0.4;
b = fir1(n,Wn);
```

create row vector `b` containing the coefficients of the order `n` Hamming-windowed filter. This is a lowpass, linear phase FIR filter with cutoff frequency `Wn`. `Wn` is a number between 0 and 1, where 1 corresponds to the Nyquist frequency, half the sampling frequency. (Unlike other methods, here `Wn` corresponds to the 6 dB point.) For a highpass filter, simply append `'high'` to the function's parameter list. For a bandpass or bandstop filter, specify `Wn` as a two-element vector containing the passband edge frequencies. Append `'stop'` for the bandstop configuration.

`b = fir1(n,Wn,window)` uses the window specified in column vector `window` for the design. The vector `window` must be `n+1` elements long. If you do not specify a window, `fir1` applies a Hamming window.

#### Kaiser Window Order Estimation

The `kaiserord` function estimates the filter order, cutoff frequency, and Kaiser window beta parameter needed to meet a given set of specifications. Given a vector of frequency band edges and a

corresponding vector of magnitudes, as well as maximum allowable ripple, `kaiserord` returns appropriate input parameters for the `fir1` function.

#### Multiband FIR Filter Design: fir2

The `fir2` function also designs windowed FIR filters, but with an arbitrarily shaped piecewise linear frequency response. This is in contrast to `fir1`, which only designs filters in standard lowpass, highpass, bandpass, and bandstop configurations.

The commands

```
n = 50;
f = [0 .4 .5 1];
m = [1  1  0 0];
b = fir2(n,f,m);
```

return row vector `b` containing the `n+1` coefficients of the order `n` FIR filter whose frequency-magnitude characteristics match those given by vectors `f` and `m`. `f` is a vector of frequency points ranging from 0 to 1, where 1 represents the Nyquist frequency. `m` is a vector containing the specified magnitude response at the points specified in `f`. (The IIR counterpart of this function is `yulewalk`, which also designs filters based on arbitrary piecewise linear magnitude responses. See “IIR Filter Design” on page 2-4 for details.)

## Multiband FIR Filter Design with Transition Bands

The `firls` and `firpm` functions provide a more general means of specifying the ideal specified filter than the `fir1` and `fir2` functions. These functions design Hilbert transformers, differentiators, and other filters with odd symmetric coefficients (type III and type IV linear phase). They also let you include transition or “don't care” regions in which the error is not minimized, and perform band dependent weighting of the minimization.

The `firls` function is an extension of the `fir1` and `fir2` functions in that it minimizes the integral of the square of the error between the specified frequency response and the actual frequency response.

The `firpm` function implements the Parks-McClellan algorithm, which uses the Remez exchange algorithm and Chebyshev approximation theory to design filters with optimal fits between the specified and actual frequency responses. The filters are optimal in the sense that they minimize the maximum error between the specified frequency response and the actual frequency response; they are sometimes called *minimax* filters. Filters designed in this way exhibit an equiripple behavior in their frequency response, and hence are also known as *equiripple* filters. The Parks-McClellan FIR filter design algorithm is perhaps the most popular and widely used FIR filter design methodology.

The syntax for `firls` and `firpm` is the same; the only difference is their minimization schemes. The next example shows how filters designed with `firls` and `firpm` reflect these different schemes.

#### Basic Configurations

The default mode of operation of `firls` and `firpm` is to design type I or type II linear phase filters, depending on whether the order you want is even or odd, respectively. A lowpass example with approximate amplitude 1 from 0 to 0.4 Hz, and approximate amplitude 0 from 0.5 to 1.0 Hz is

```
n = 20;                    % Filter order
f = [0 0.4 0.5 1];         % Frequency band edges
```

```
a = [1  1  0 0];           % Amplitudes
b = firpm(n,f,a);
```

From 0.4 to 0.5 Hz, `firpm` performs no error minimization; this is a transition band or “don't care” region. A transition band minimizes the error more in the bands that you do care about, at the expense of a slower transition rate. In this way, these types of filters have an inherent trade-off similar to FIR design by windowing.

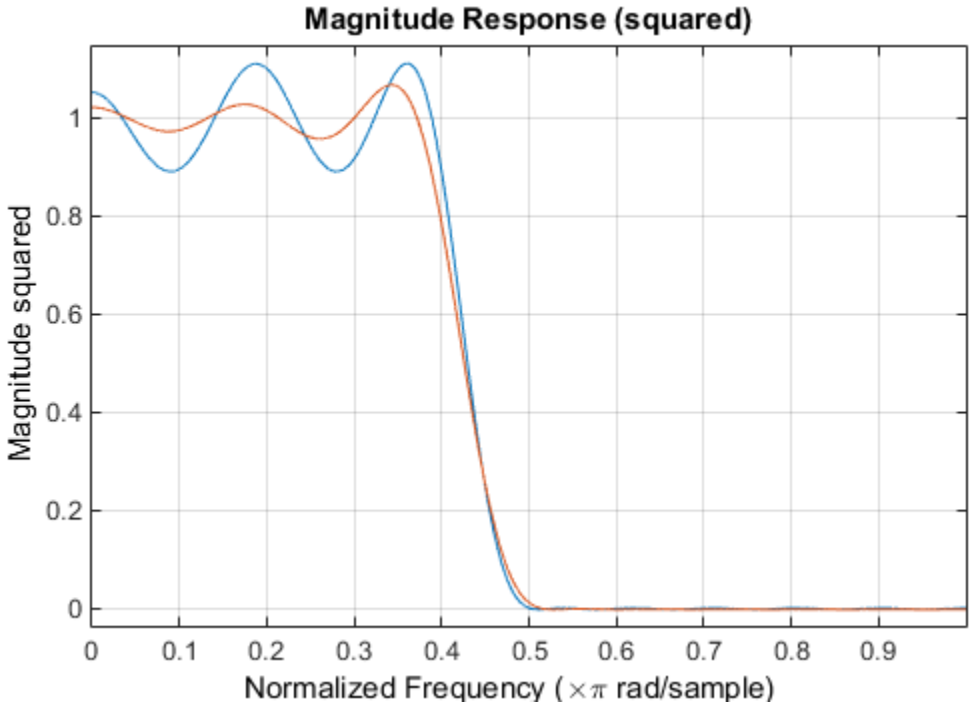
To compare least squares to equiripple filter design, use `firls` to create a similar filter. Type

```
bb = firls(n,f,a);
```

and compare their frequency responses using FVTool:

```
fvtool(b,1,bb,1)
```

Note that the *y*-axis shown in the figure below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.




The filter designed with `firpm` exhibits equiripple behavior. Also note that the `firls` filter has a better response over most of the passband and stopband, but at the band edges (`f = 0.4` and `f = 0.5`), the response is further away from the ideal than the `firpm` filter. This shows that the `firpm` filter's *maximum* error over the passband and stopband is smaller and, in fact, it is the smallest possible for this band edge configuration and filter length.

Think of frequency bands as lines over short frequency intervals. `firpm` and `firls` use this scheme to represent any piecewise linear frequency-response function with any transition bands. `firls` and `firpm` design lowpass, highpass, bandpass, and bandstop filters; a bandpass example is

```
f = [0 0.3  0.4  0.7  0.8  1];   % Band edges in pairs
a = [0  0    1    1    0   0];   % Bandpass filter amplitude
```

Technically, these `f` and `a` vectors define five bands:

- Two stopbands, from 0.0 to 0.3 and from 0.8 to 1.0
- A passband from 0.4 to 0.7
- Two transition bands, from 0.3 to 0.4 and from 0.7 to 0.8

Example highpass and bandstop filters are

```
f = [0 0.7  0.8  1];            % Band edges in pairs
a = [0  0    1   1];            % Highpass filter amplitude
f = [0 0.3  0.4  0.5  0.8  1];  % Band edges in pairs
a = [1  1    0    0    1   1];  % Bandstop filter amplitude
```

An example multiband bandpass filter is

```
f = [0 0.1 0.15 0.25 0.3 0.4 0.45 0.55 0.6 0.7 0.75 0.85 0.9 1];
a = [1  1   0    0    1   1   0    0    1   1   0    0    1  1];
```

Another possibility is a filter that has as a transition region the line connecting the passband with the stopband; this can help control “runaway” magnitude response in wide transition regions:

```
f = [0 0.4 0.42 0.48 0.5  1];
a = [1 1 0.8 0.2 0 0];   % Passband, linear transition,
                         %  stopband
```

### The Weight Vector

Both `firls` and `firpm` allow you to place more or less emphasis on minimizing the error in certain frequency bands relative to others. To do this, specify a weight vector following the frequency and amplitude vectors. An example lowpass equiripple filter with 10 times less ripple in the stopband than the passband is

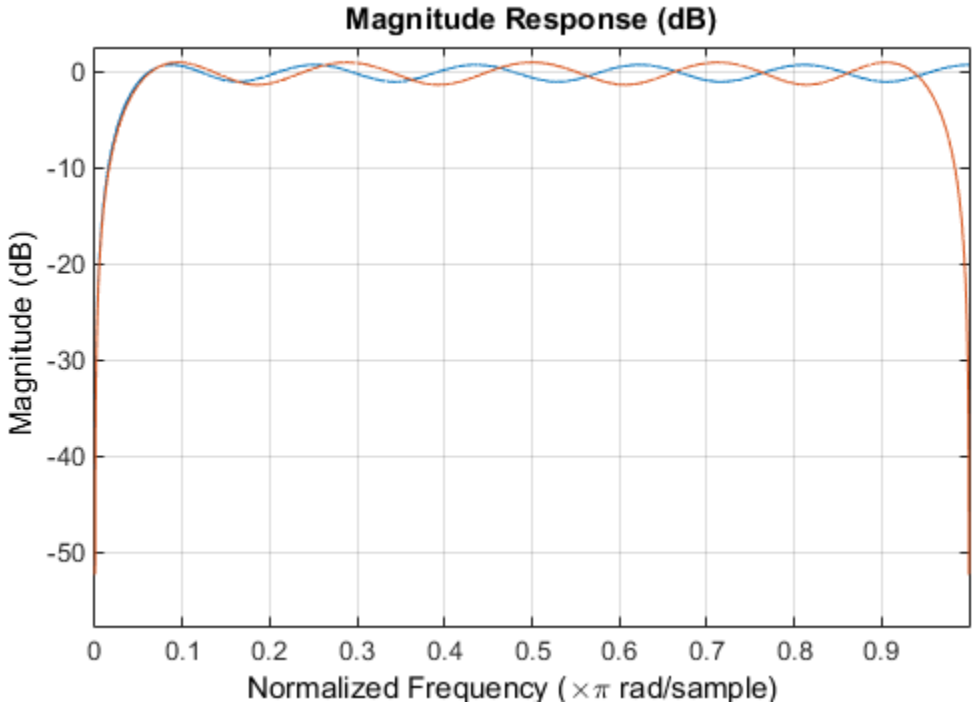
```
n = 20;              % Filter order
f = [0 0.4 0.5 1];   % Frequency band edges
a = [1  1   0  0];   % Amplitudes
w = [1 10];          % Weight vector
b = firpm(n,f,a,w);
```

A legal weight vector is always half the length of the `f` and `a` vectors; there must be exactly one weight per band.

### Anti-Symmetric Filters / Hilbert Transformers

When called with a trailing `'h'` or `'Hilbert'` option, `firpm` and `firls` design FIR filters with odd symmetry, that is, type III (for even order) or type IV (for odd order) linear phase filters. An ideal Hilbert transformer has this anti-symmetry property and an amplitude of 1 across the entire frequency range. Try the following approximate Hilbert transformers and plot them using FVTool:

```
b = firpm(21,[0.05 1],[1 1],'h');       % Highpass Hilbert
bb = firpm(20,[0.05 0.95],[1 1],'h');   % Bandpass Hilbert
fvtool(b,1,bb,1)
```




You can find the delayed Hilbert transform of a signal `x` by passing it through these filters.

```
fs = 1000;            % Sampling frequency
t = (0:1/fs:2)';      % Two second time vector
x = sin(2*pi*300*t);  % 300 Hz sine wave example signal
xh = filter(bb,1,x);  % Hilbert transform of x
```

The analytic signal corresponding to `x` is the complex signal that has `x` as its real part and the Hilbert transform of `x` as its imaginary part. For this FIR method (an alternative to the `hilbert` function), you must delay `x` by half the filter order to create the analytic signal:

```
xd = [zeros(10,1); x(1:length(x)-10)];        % Delay 10 samples
xa = xd + j*xh;                              % Analytic signal
```

This method does not work directly for filters of odd order, which require a noninteger delay. In this case, the `hilbert` function, described in “Hilbert Transform” on page 8-44, estimates the analytic signal. Alternatively, use the `resample` function to delay the signal by a noninteger number of samples.

### Differentiators

Differentiation of a signal in the time domain is equivalent to multiplication of the signal's Fourier transform by an imaginary ramp function. That is, to differentiate a signal, pass it through a filter that has a response $H(\omega) = j\omega$. Approximate the ideal differentiator (with a delay) using `firpm` or `firls` with a `'d'` or `'differentiator'` option:

```
b = firpm(21,[0 1],[0 pi],'d');
```

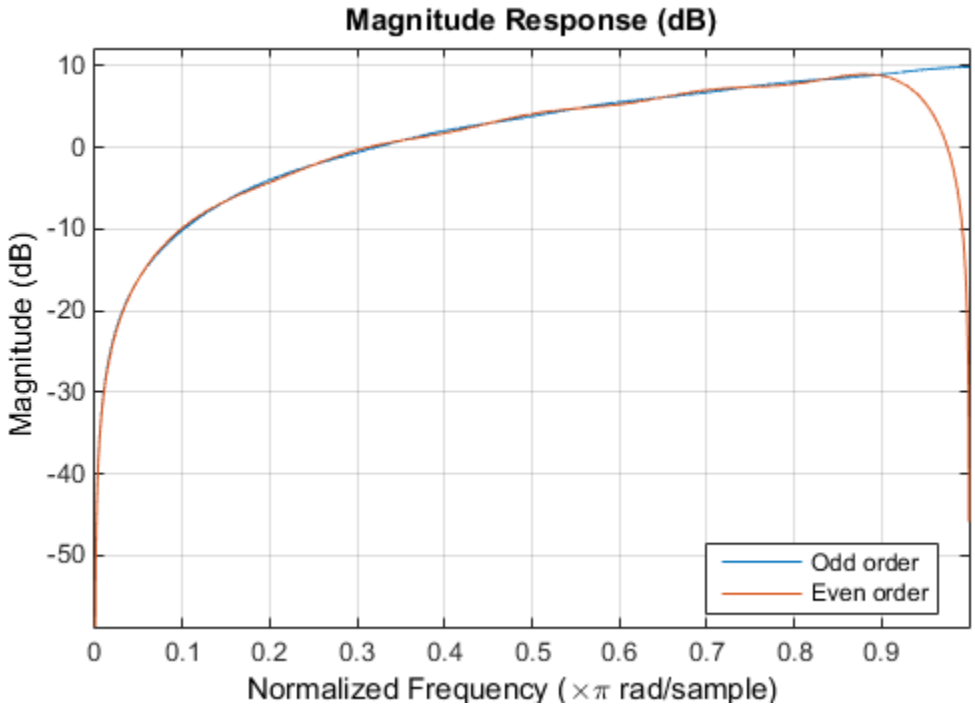
For a type III filter, the differentiation band should stop short of the Nyquist frequency, and the amplitude vector must reflect that change to ensure the correct slope:

```
bb = firpm(20,[0 0.9],[0 0.9*pi],'d');
```

In the `'d'` mode, `firpm` weights the error by 1/ω in nonzero amplitude bands to minimize the maximum *relative* error. `firls` weights the error by $(1/\omega)^2$ in nonzero amplitude bands in the `'d'` mode.

The following plots show the magnitude responses for the differentiators above.

```
fvtool(b,1,bb,1)
legend('Odd order','Even order','Location','best')
```




## Constrained Least Squares FIR Filter Design

The Constrained Least Squares (CLS) FIR filter design functions implement a technique that enables you to design FIR filters without explicitly defining the transition bands for the magnitude response. The ability to omit the specification of transition bands is useful in several situations. For example, it may not be clear where a rigidly defined transition band should appear if noise and signal information appear together in the same frequency band. Similarly, it may make sense to omit the specification of transition bands if they appear only to control the results of Gibbs phenomena that appear in the filter's response. See Selesnick, Lang, and Burrus [2] for discussion of this method.

Instead of defining passbands, stopbands, and transition regions, the CLS method accepts a cutoff frequency (for the highpass, lowpass, bandpass, or bandstop cases), or passband and stopband edges (for multiband cases), for the response you specify. In this way, the CLS method defines transition regions implicitly, rather than explicitly.

The key feature of the CLS method is that it enables you to define upper and lower thresholds that contain the maximum allowable ripple in the magnitude response. Given this constraint, the technique applies the least square error minimization technique over the frequency range of the filter's response, instead of over specific bands. The error minimization includes any areas of discontinuity in the ideal, "brick wall" response. An additional benefit is that the technique enables you to specify arbitrarily small peaks resulting from the Gibbs phenomenon.

There are two toolbox functions that implement this design technique.

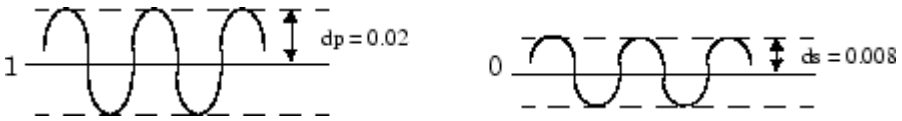
| Description | Function |
|---|---|
| Constrained least square multiband FIR filter design | `fircls` |
| Constrained least square filter design for lowpass and highpass linear phase filters | `fircls1` |

For details on the calling syntax for these functions, see their reference descriptions in the Function Reference.

### Basic Lowpass and Highpass CLS Filter Design

The most basic of the CLS design functions, `fircls1`, uses this technique to design lowpass and highpass FIR filters. As an example, consider designing a filter with order 61 impulse response and cutoff frequency of 0.3 (normalized). Further, define the upper and lower bounds that constrain the design process as:

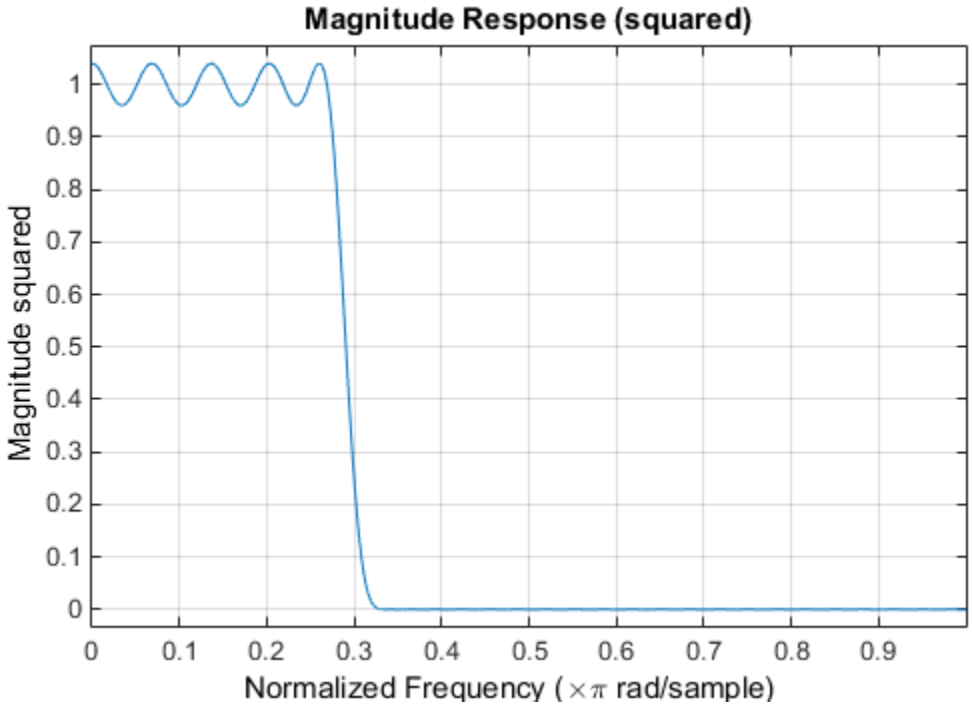
- Maximum passband deviation from 1 (passband ripple) of 0.02.
- Maximum stopband deviation from 0 (stopband ripple) of 0.008.




To approach this design problem using `fircls1`, use the following commands:

```
n = 61;
wo = 0.3;
dp = 0.02;
ds = 0.008;
h = fircls1(n,wo,dp,ds);
fvtool(h,1)
```

Note that the *y*-axis shown below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.




### Multiband CLS Filter Design

`fircls` uses the same technique to design FIR filters with a specified piecewise constant magnitude response. In this case, you can specify a vector of band edges and a corresponding vector of band amplitudes. In addition, you can specify the maximum amount of ripple for each band.

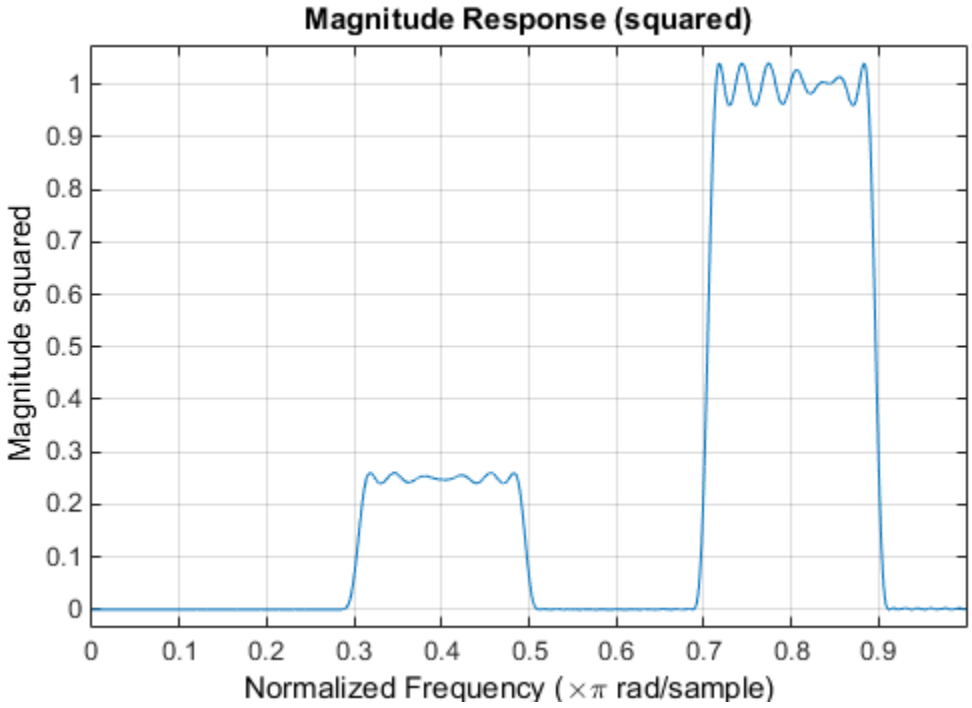
For example, assume the specifications for a filter call for:

- From 0 to 0.3 (normalized): amplitude 0, upper bound 0.005, lower bound –0.005
- From 0.3 to 0.5: amplitude 0.5, upper bound 0.51, lower bound 0.49
- From 0.5 to 0.7: amplitude 0, upper bound 0.03, lower bound –0.03
- From 0.7 to 0.9: amplitude 1, upper bound 1.02, lower bound 0.98
- From 0.9 to 1: amplitude 0, upper bound 0.05, lower bound –0.05

Design a CLS filter with impulse response order 129 that meets these specifications:

```
n = 129;
f = [0 0.3 0.5 0.7 0.9 1];
a = [0 0.5 0 1 0];
up = [0.005 0.51 0.03 1.02 0.05];
lo = [-0.005 0.49 -0.03 0.98 -0.05];
h = fircls(n,f,a,up,lo);
fvtool(h,1)
```

Note that the *y*-axis shown below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.




### Weighted CLS Filter Design

Weighted CLS filter design lets you design lowpass or highpass FIR filters with relative weighting of the error minimization in each band. The `fircls1` function enables you to specify the passband and stopband edges for the least squares weighting function, as well as a constant `k` that specifies the ratio of the stopband to passband weighting.

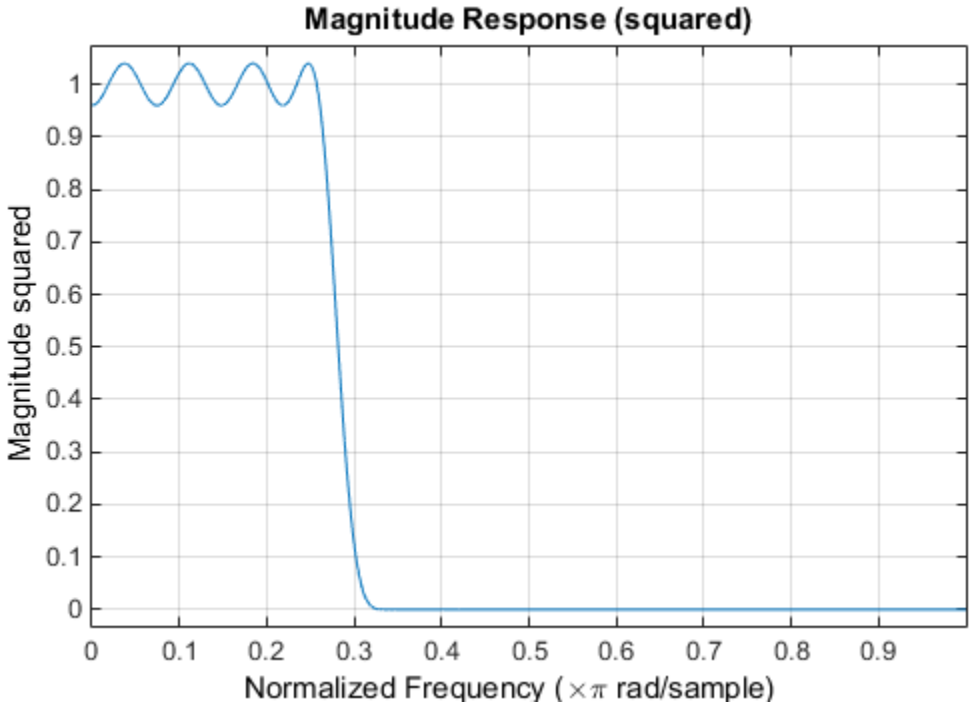
For example, consider specifications that call for an FIR filter with impulse response order of 55 and cutoff frequency of 0.3 (normalized). Also assume maximum allowable passband ripple of 0.02 and maximum allowable stopband ripple of 0.004. In addition, add weighting requirements:

- Passband edge for the weight function of 0.28 (normalized)
- Stopband edge for the weight function of 0.32
- Weight error minimization 10 times as much in the stopband as in the passband

To approach this using `fircls1`, type

```
n = 55;
wo = 0.3;
dp = 0.02;
ds = 0.004;
wp = 0.28;
ws = 0.32;
k = 10;
h = fircls1(n,wo,dp,ds,wp,ws,k);
fvtool(h,1)
```

Note that the *y*-axis shown below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.




## Arbitrary-Response Filter Design

The `cfirpm` filter design function provides a tool for designing FIR filters with arbitrary complex responses. It differs from the other filter design functions in how the frequency response of the filter is specified: it accepts the name of a function which returns the filter response calculated over a grid of frequencies. This capability makes `cfirpm` a highly versatile and powerful technique for filter design.

This design technique may be used to produce nonlinear-phase FIR filters, asymmetric frequency-response filters (with complex coefficients), or more symmetric filters with custom frequency responses.

The design algorithm optimizes the Chebyshev (or minimax) error using an extended Remez-exchange algorithm for an initial estimate. If this exchange method fails to obtain the optimal filter, the algorithm switches to an ascent-descent algorithm that takes over to finish the convergence to the optimal solution.

### Multiband Filter Design

Consider a multiband filter with the following special frequency-domain characteristics.

| Band | Amplitude | Optimization Weighting |
|---|---|---|
| [–1 –0.5] | [5 1] | 1 |
| [–0.4 +0.3] | [2 2] | 10 |
| [+0.4 +0.8] | [2 1] | 5 |

A linear-phase multiband filter may be designed using the predefined frequency-response function `multiband`, as follows:

```
b = cfirpm(38, [-1 -0.5 -0.4 0.3 0.4 0.8], ...
               {'multiband', [5 1 2 2 2 1]}, [1 10 5]);
```

For the specific case of a multiband filter, we can use a shorthand filter design notation similar to the syntax for `firpm`:

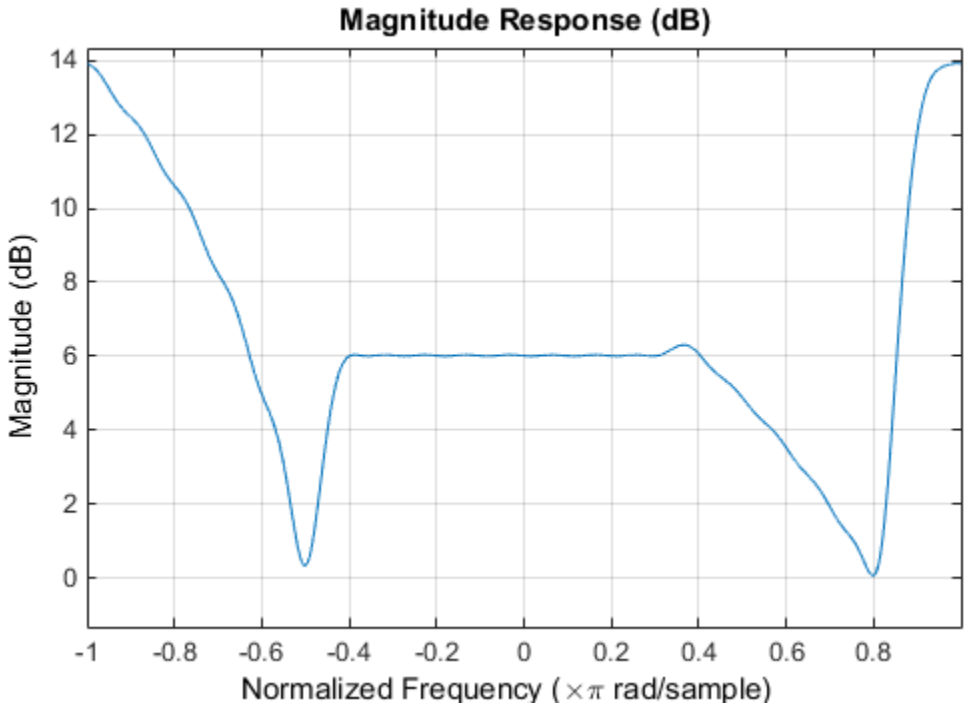
```
b = cfirpm(38,[-1 -0.5 -0.4 0.3 0.4 0.8], ...
              [5 1 2 2 2 1], [1 10 5]);
```

As with `firpm`, a vector of band edges is passed to `cfirpm`. This vector defines the frequency bands over which optimization is performed; note that there are two transition bands, from –0.5 to –0.4 and from 0.3 to 0.4.

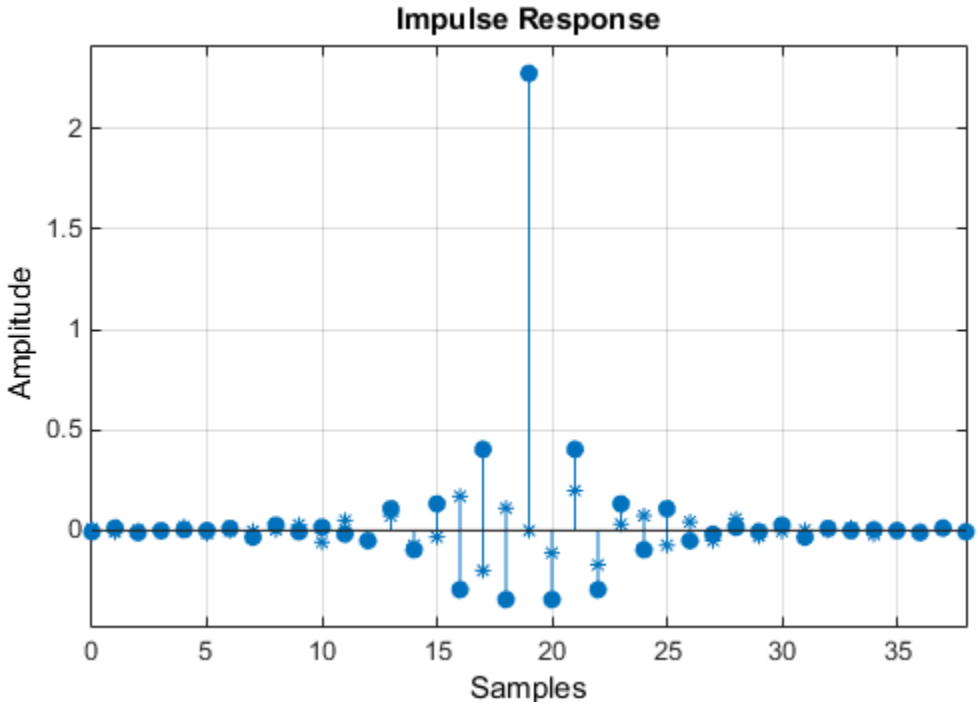
In either case, the frequency response is obtained and plotted using linear scale in FVTool:

```
fvtool(b,1)
```

Note that the range of data shown below is `(-pi,pi)`.




The filter response for this multiband filter is complex, which is expected because of the asymmetry in the frequency domain. The impulse response, which you can select from the FVTool toolbar, is shown below.




### Filter Design with Reduced Delay

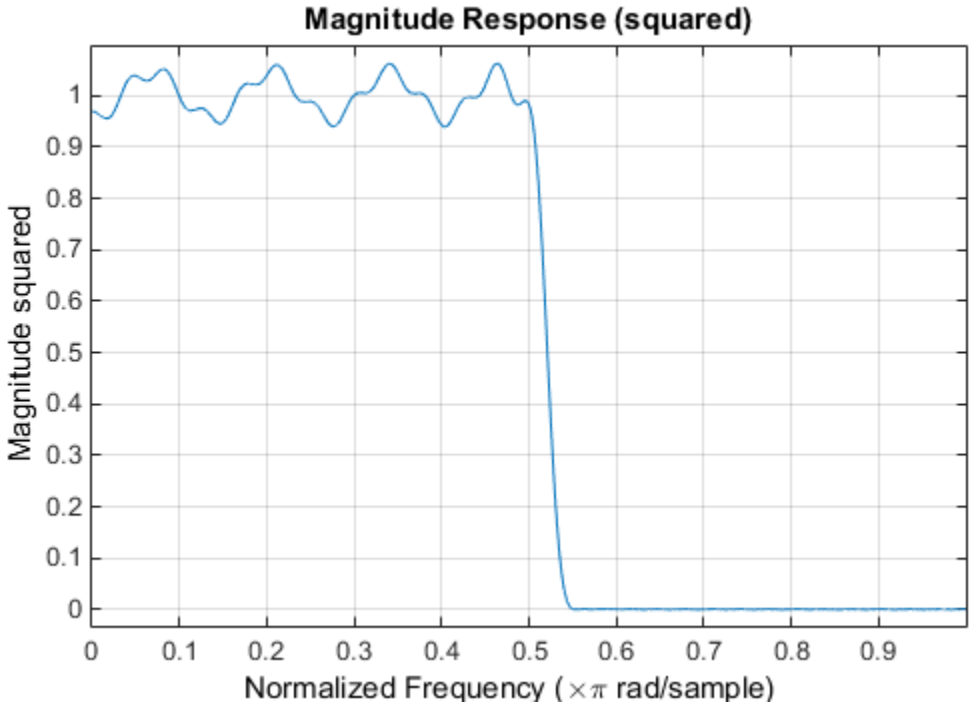
Consider the design of a 62-tap lowpass filter with a half-Nyquist cutoff. If we specify a negative offset value to the `lowpass` filter design function, the group delay offset for the design is significantly less than that obtained for a standard linear-phase design. This filter design may be computed as follows:

```
b = cfirpm(61,[0 0.5 0.55 1],{'lowpass',-16});
```

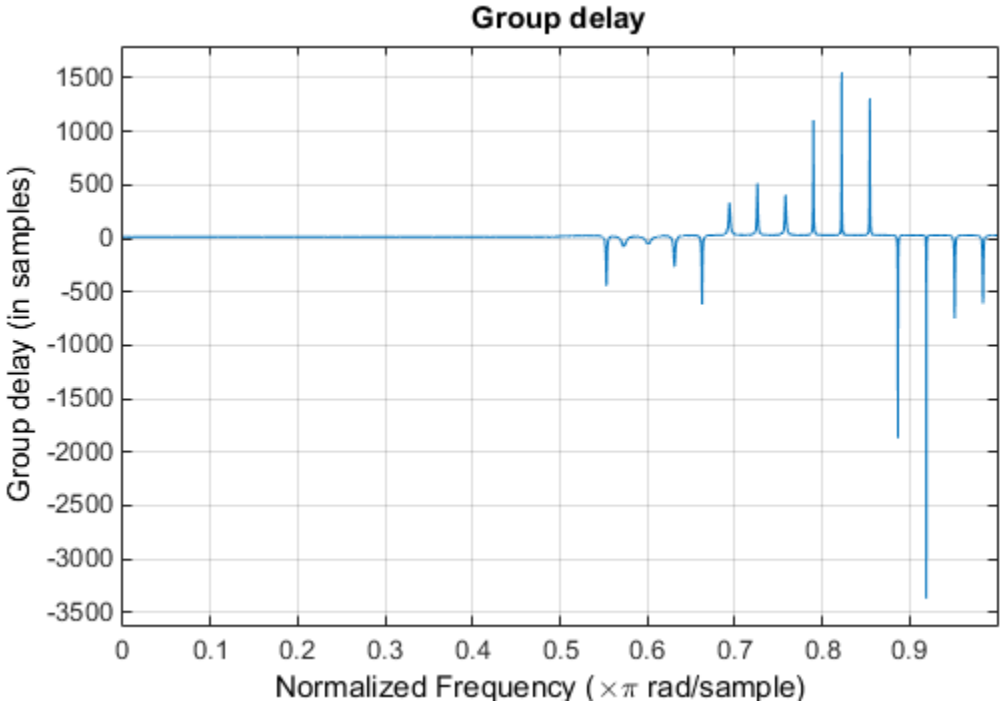
The resulting magnitude response is

```
fvtool(b,1)
```

The *y*-axis is in Magnitude Squared, which you can set by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.




The group delay of the filter reveals that the offset has been reduced from `N/2` to `N/2-16` (i.e., from `30.5` to `14.5`). Now, however, the group delay is no longer flat in the passband region. To create this plot, click the **Group Delay Response** button on the toolbar.




If we compare this nonlinear-phase filter to a linear-phase filter that has exactly 14.5 samples of group delay, the resulting filter is of order 2*14.5, or 29. Using `b = cfirpm(29,[0 0.5 0.55`

`1],'lowpass')`, the passband and stopband ripple is much greater for the order 29 filter. These comparisons can assist you in deciding which filter is more appropriate for a specific application.

# Special Topics in IIR Filter Design

| In this section... |
|---|
| “Classic IIR Filter Design” on page 2-33 |
| “Analog Prototype Design” on page 2-33 |
| “Frequency Transformation” on page 2-34 |
| “Filter Discretization” on page 2-35 |

## Classic IIR Filter Design

The classic IIR filter design technique includes the following steps.

1. Find an analog lowpass filter with cutoff frequency of 1 and translate this prototype filter to the specified band configuration
2. Transform the filter to the digital domain.
3. Discretize the filter.

The toolbox provides functions for each of these steps.

| Design Task | Available functions |
|---|---|
| Analog lowpass prototype | `buttap`, `cheb1ap`, `besselap`, `ellipap`, `cheb2ap` |
| Frequency transformation | `lp2lp`, `lp2hp`, `lp2bp`, `lp2bs` |
| Discretization | `bilinear`, `impinvar` |

Alternatively, the `butter`, `cheby1`, `cheb2ord`, `ellip`, and `besself` functions perform all steps of the filter design and the `buttord`, `cheb1ord`, `cheb2ord`, and `ellipord` functions provide minimum order computation for IIR filters. These functions are sufficient for many design problems, and the lower level functions are generally not needed. But if you do have an application where you need to transform the band edges of an analog filter, or discretize a rational transfer function, this section describes the tools with which to do so.

## Analog Prototype Design

This toolbox provides a number of functions to create lowpass analog prototype filters with cutoff frequency of 1, the first step in the classical approach to IIR filter design.

The table below summarizes the analog prototype design functions for each supported filter type; plots for each type are shown in “IIR Filter Design” on page 2-4.

| Filter Type | Analog Prototype Function |
|---|---|
| Bessel | `[z,p,k] = besselap(n)` |
| Butterworth | `[z,p,k] = buttap(n)` |
| Chebyshev Type I | `[z,p,k] = cheb1ap(n,Rp)` |
| Chebyshev Type II | `[z,p,k] = cheb2ap(n,Rs)` |
| Elliptic | `[z,p,k] = ellipap(n,Rp,Rs)` |

## Frequency Transformation

The second step in the analog prototyping design technique is the frequency transformation of a lowpass prototype. The toolbox provides a set of functions to transform analog lowpass prototypes (with cutoff frequency of 1 rad/s) into bandpass, highpass, bandstop, and lowpass filters with the specified cutoff frequency.

| Frequency Transformation | Transformation Function |
|---|---|
| Lowpass to lowpass<br>$s' = s/\omega_0$ | `[numt,dent]   = lp2lp (num,den,Wo)`<br>`[At,Bt,Ct,Dt] = lp2lp (A,B,C,D,Wo)` |
| Lowpass to highpass<br>$s' = \frac{\omega_0}{s}$ | `[numt,dent]   = lp2hp (num,den,Wo)`<br>`[At,Bt,Ct,Dt] = lp2hp (A,B,C,D,Wo)` |
| Lowpass to bandpass<br>$s' = \frac{\omega_0}{B_\omega}\frac{(s/\omega_0)^2 + 1}{s/\omega_0}$ | `[numt,dent]   = lp2bp (num,den,Wo,Bw)`<br>`[At,Bt,Ct,Dt] = lp2bp (A,B,C,D,Wo,Bw)` |
| Lowpass to bandstop<br>$s' = \frac{B_\omega}{\omega_0}\frac{s/\omega_0}{(s/\omega_0)^2 + 1}$ | `[numt,dent]   = lp2bs (num,den,Wo,Bw)`<br>`[At,Bt,Ct,Dt] = lp2bs( A,B,C,D,Wo,Bw)` |

As shown, all of the frequency transformation functions can accept two linear system models: transfer function and state-space form. For the bandpass and bandstop cases

$$\omega_0 = \sqrt{\omega_1 \omega_2}$$

and

$$B_\omega = \omega_2 - \omega_1$$

where $\omega_1$ is the lower band edge and $\omega_2$ is the upper band edge.

The frequency transformation functions perform frequency variable substitution. In the case of `lp2bp` and `lp2bs`, this is a second-order substitution, so the output filter is twice the order of the input. For `lp2lp` and `lp2hp`, the output filter is the same order as the input.

To begin designing an order 10 bandpass Chebyshev Type I filter with a value of 3 dB for passband ripple, enter

```
[z,p,k] = cheb1ap(10,3);
```

Outputs `z`, `p`, and `k` contain the zeros, poles, and gain of a lowpass analog filter with cutoff frequency $\Omega_c$ equal to 1 rad/s. Use the function to transform this lowpass prototype to a bandpass analog filter with band edges $\Omega_1 = \pi/5$ and $\Omega_2 = \pi$. First, convert the filter to state-space form so the `lp2bp` function can accept it:

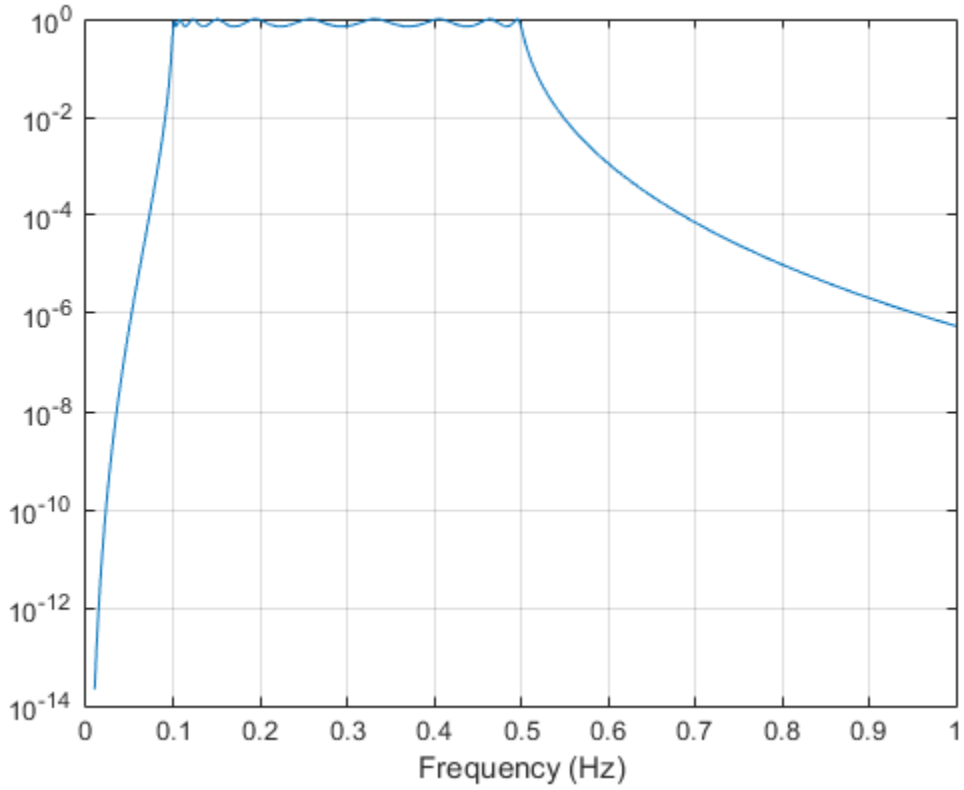
```
[A,B,C,D] = zp2ss(z,p,k);   % Convert to state-space form.
```

Now, find the bandwidth and center frequency, and call `lp2bp`:

```
u1 = 0.1*2*pi;
u2 = 0.5*2*pi;   % In radians per second
Bw = u2-u1;
Wo = sqrt(u1*u2);
[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw);
```

Finally, calculate the frequency response and plot its magnitude:

```
[b,a] = ss2tf(At,Bt,Ct,Dt);         % Convert to TF form
w = linspace(0.01,1,500)*2*pi;      % Generate frequency vector
h = freqs(b,a,w);                   % Compute frequency response
semilogy(w/2/pi,abs(h))             % Plot log magnitude vs. freq
xlabel('Frequency (Hz)')
grid
```




## Filter Discretization

The third step in the analog prototyping technique is the transformation of the filter to the discrete-time domain. The toolbox provides two methods for this: the impulse invariant and bilinear transformations. The filter design functions `butter`, `cheby1`, `cheby2`, and `ellip` use the bilinear transformation for discretization in this step.

| Analog to Digital Transformation | Transformation Function |
|---|---|
| Impulse invariance | `[numd,dend] = impinvar (num,den,fs)` |

| Analog to Digital Transformation | Transformation Function |
|---|---|
| Bilinear transform | `[zd,pd,kd] = bilinear (z,p,k,fs,Fp)`<br>`[numd,dend] = bilinear (num,den,fs,Fp)`<br>`[Ad,Bd,Cd,Dd] = bilinear (At,Bt,Ct,Dt,fs,Fp)` |

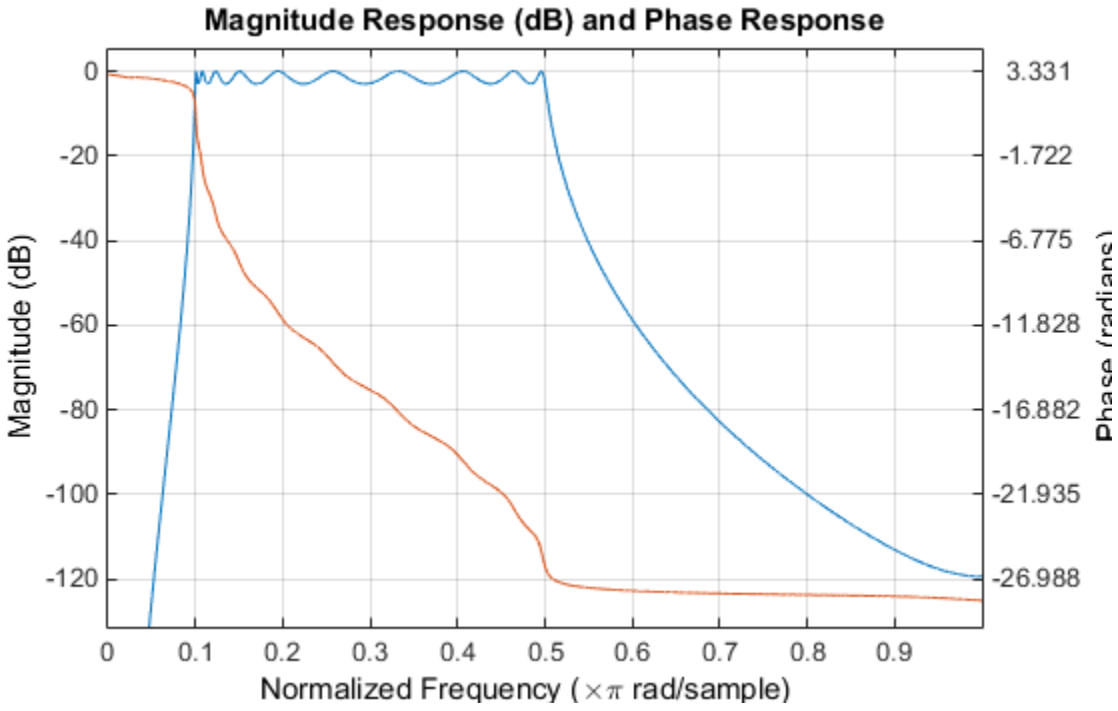
### Impulse Invariance

The toolbox function `impinvar` creates a digital filter whose impulse response is the samples of the continuous impulse response of an analog filter. This function works only on filters in transfer function form. For best results, the analog filter should have negligible frequency content above half the sampling frequency, because such high-frequency content is aliased into lower bands upon sampling. Impulse invariance works for some lowpass and bandpass filters, but is not appropriate for highpass and bandstop filters.

Design a Chebyshev Type I filter and plot its frequency and phase response using FVTool:

```
[bz,az] = impinvar(b,a,2);
fvtool(bz,az)
```

Click the **Magnitude and Phase Response** toolbar button.




Impulse invariance retains the cutoff frequencies of 0.1 Hz and 0.5 Hz.

### Bilinear Transformation

The bilinear transformation is a nonlinear mapping of the continuous domain to the discrete domain; it maps the *s*-plane into the *z*-plane by

$$H(z) = H(s)\Big|_{s = k\frac{z-1}{z+1}}$$

Bilinear transformation maps the $j\Omega$-axis of the continuous domain to the unit circle of the discrete domain according to

$$\omega = 2\tan^{-1}\left(\frac{\Omega}{k}\right)$$

The toolbox function `bilinear` implements this operation, where the frequency warping constant $k$ is equal to twice the sampling frequency (`2*fs`) by default, and equal to $2\pi f_p/\tan(\pi f_p/f_s)$if you give `bilinear` a trailing argument that represents a "match" frequency `Fp`. If a match frequency `Fp` (in hertz) is present, `bilinear` maps the frequency $\Omega = 2\pi f_p$ (in rad/s) to the same frequency in the discrete domain, normalized to the sampling rate: $\omega = 2\pi f_p/f_s$ (in rad/sample).

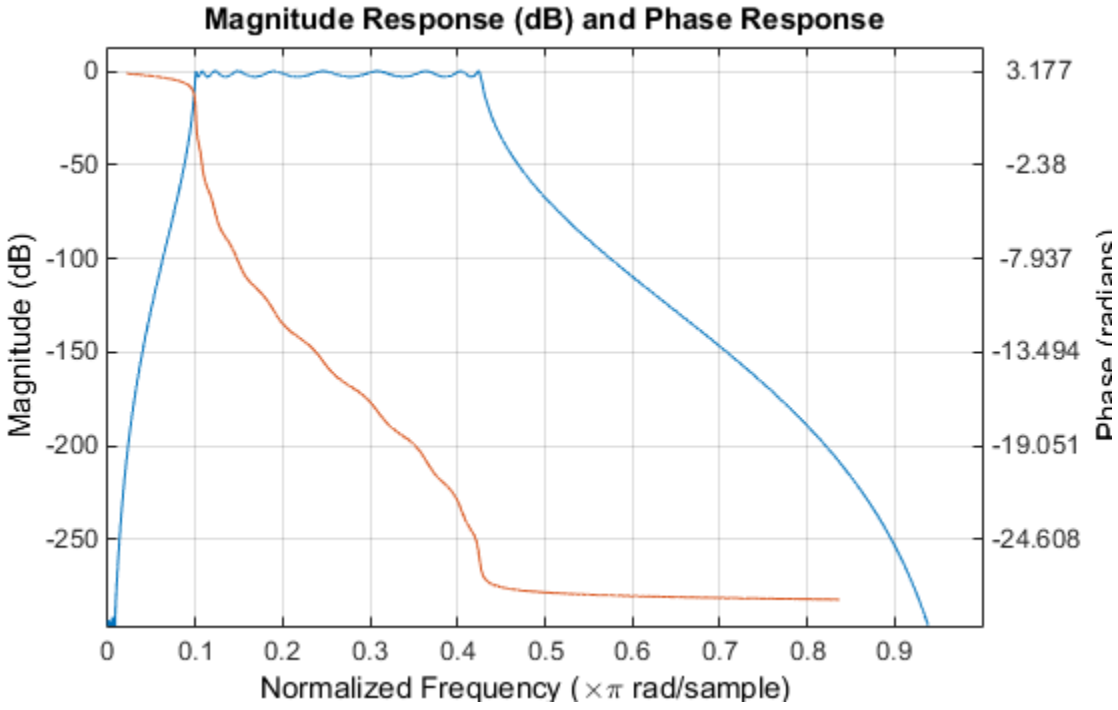
The `bilinear` function can perform this transformation on three different linear system representations: zero-pole-gain, transfer function, and state-space form. Try calling `bilinear` with the state-space matrices that describe the Chebyshev Type I filter from the previous section, using a sampling frequency of 2 Hz, and retaining the lower band edge of 0.1 Hz:

```
[Ad,Bd,Cd,Dd] = bilinear(At,Bt,Ct,Dt,2,0.1);
```

The frequency response of the resulting digital filter is

```
[bz,az] = ss2tf(Ad,Bd,Cd,Dd);       % Convert to TF
fvtool(bz,az)
```

Click the **Magnitude and Phase Response** toolbar button.




The lower band edge is at 0.1 Hz as expected. Notice, however, that the upper band edge is slightly less than 0.5 Hz, although in the analog domain it was exactly 0.5 Hz. This illustrates the nonlinear

nature of the bilinear transformation. To counteract this nonlinearity, it is necessary to create analog domain filters with “prewarped” band edges, which map to the correct locations upon bilinear transformation. Here the prewarped frequencies `u1` and `u2` generate `Bw` and `Wo` for the `lp2bp` function:

```
fs = 2;                          % Sampling frequency (hertz)
u1 = 2*fs*tan(0.1*(2*pi/fs)/2);  % Lower band edge (rad/s)
u2 = 2*fs*tan(0.5*(2*pi/fs)/2);  % Upper band edge (rad/s)
Bw = u2 - u1;                    % Bandwidth
Wo = sqrt(u1*u2);                % Center frequency
[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw);
```

A digital bandpass filter with correct band edges 0.1 and 0.5 times the Nyquist frequency is

```
[Ad,Bd,Cd,Dd] = bilinear(At,Bt,Ct,Dt,fs);
```

The example bandpass filters from the last two sections could also be created in one statement using the complete IIR design function `cheby1`. For instance, an analog version of the example Chebyshev filter is

```
[b,a] = cheby1(5,3,[0.1 0.5]*2*pi,'s');
```

Note that the band edges are in rad/s for analog filters, whereas for the digital case, frequency is normalized:

```
[bz,az] = cheby1(5,3,[0.1 0.5]);
```

All of the complete design functions call `bilinear` internally. They prewarp the band edges as needed to obtain the correct digital filter.

# Filtering Data With Signal Processing Toolbox Software

### Lowpass FIR Filter – Window Method

This example shows how to design and implement an FIR filter using two command line functions, `fir1` and `designfilt`, and the interactive **Filter Designer** app.

Create a signal to use in the examples. The signal is a 100 Hz sine wave in additive $N(0, 1/4)$ white Gaussian noise. Set the random number generator to the default state for reproducible results.

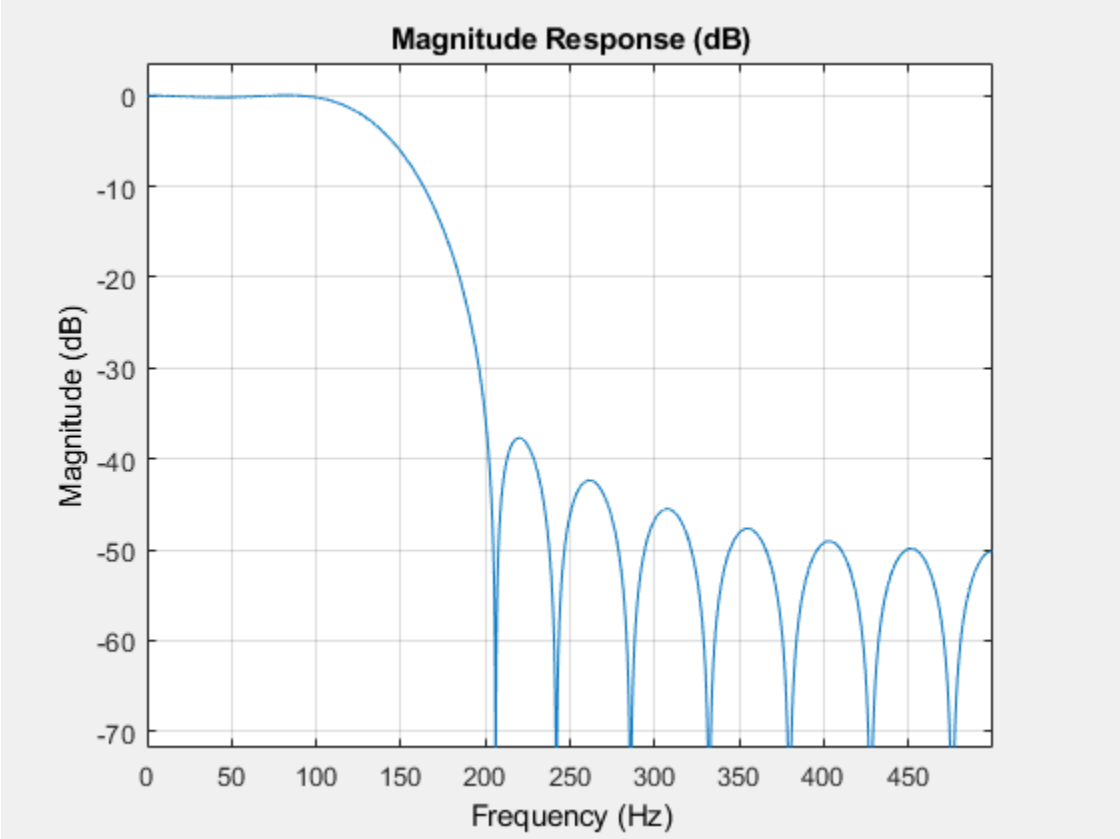
```
rng default

Fs = 1000;
t = linspace(0,1,Fs);
x = cos(2*pi*100*t)+0.5*randn(size(t));
```

The filter design is an FIR lowpass filter with order equal to 20 and a cutoff frequency of 150 Hz. Use a Kaiser window with length one sample greater than the filter order and $\beta = 3$. See `kaiser` for details on the Kaiser window.

Use `fir1` to design the filter. `fir1` requires normalized frequencies in the interval [0,1], where 1 corresponds to $\pi$ rad/sample. To use `fir1`, you must convert all frequency specifications to normalized frequencies.

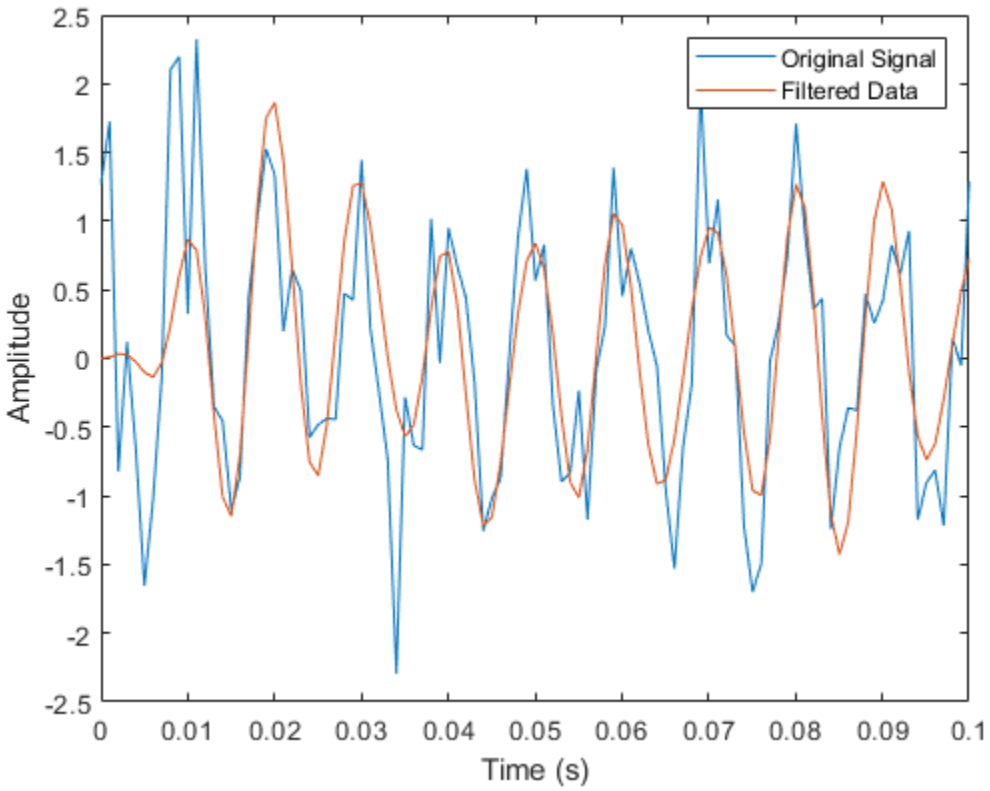
Design the filter and view the filter's magnitude response.

```
fc = 150;
Wn = (2/Fs)*fc;
b = fir1(20,Wn,'low',kaiser(21,3));

fvtool(b,1,'Fs',Fs)
```




Apply the filter to the signal and plot the result for the first ten periods of the 100 Hz sinusoid.

```
y = filter(b,1,x);

plot(t,x,t,y)
xlim([0 0.1])

xlabel('Time (s)')
ylabel('Amplitude')
legend('Original Signal','Filtered Data')
```

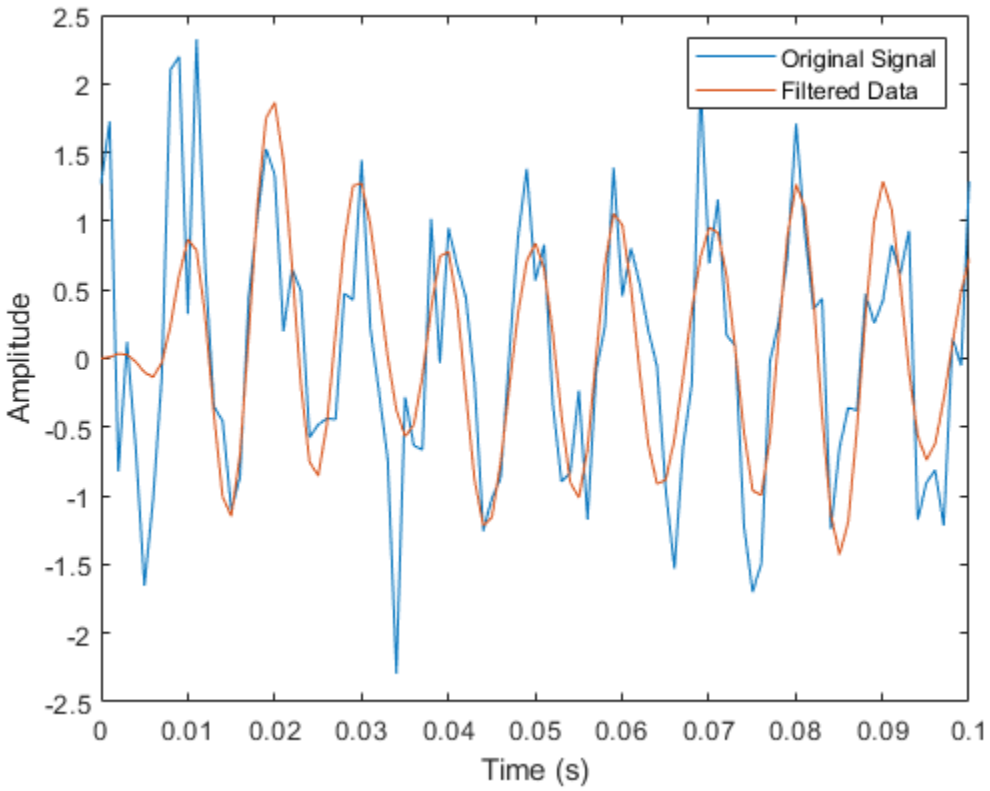



Design the same filter using `designfilt`. Set the filter response to `'lowpassfir'` and input the specifications as `Name,Value` pairs. With `designfilt`, you can specify your filter design in Hz.

```
Fs = 1000;
Hd = designfilt('lowpassfir','FilterOrder',20,'CutoffFrequency',150, ...
       'DesignMethod','window','Window',{@kaiser,3},'SampleRate',Fs);
```

Filter the data and plot the result.

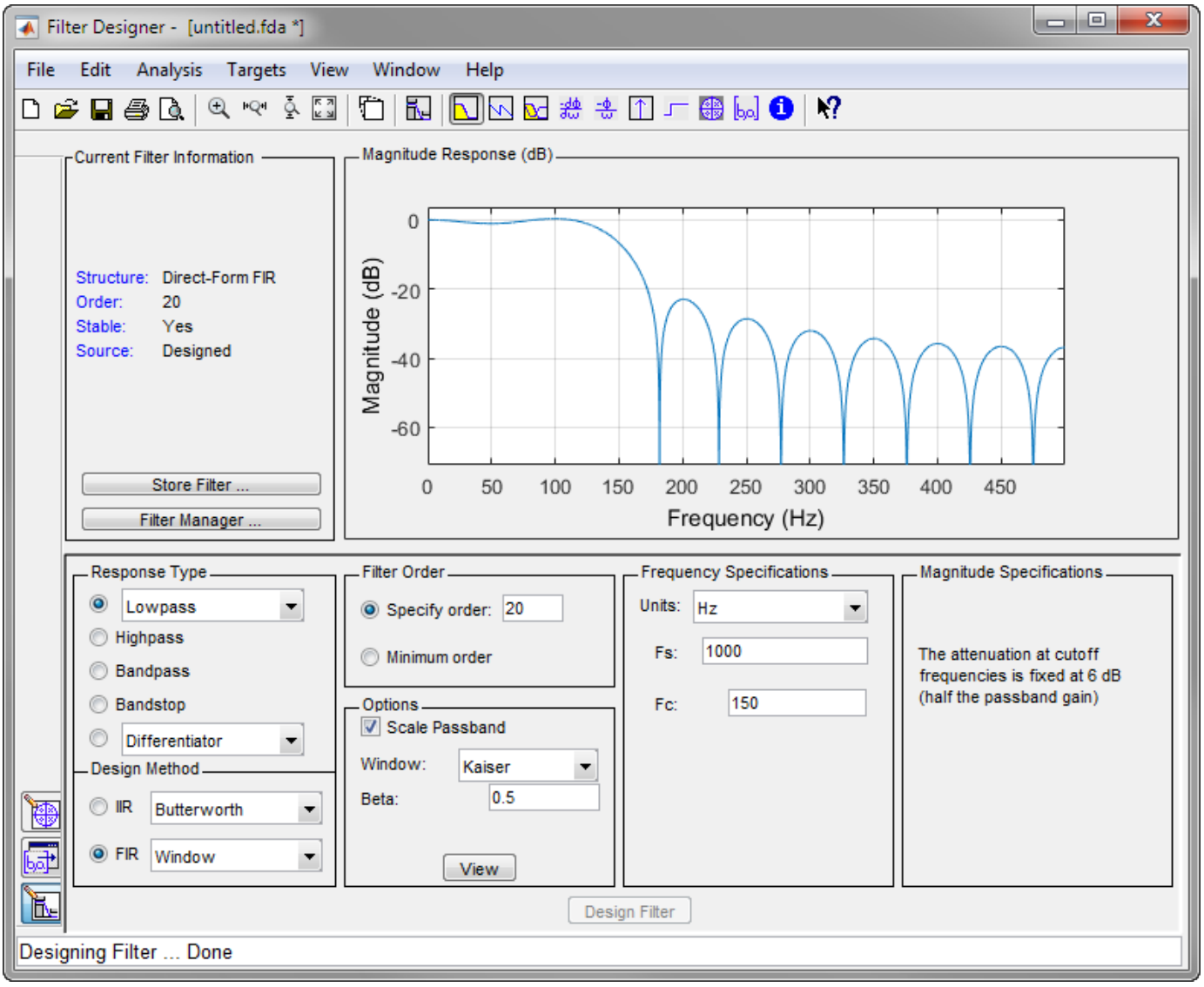
```
y1 = filter(Hd,x);

plot(t,x,t,y1)
xlim([0 0.1])

xlabel('Time (s)')
ylabel('Amplitude')
legend('Original Signal','Filtered Data')
```



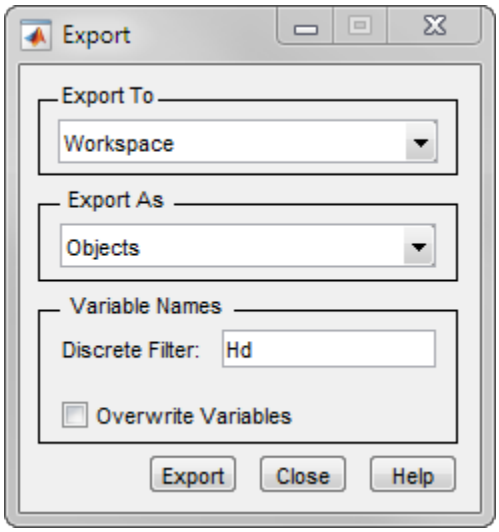

### Lowpass FIR Filter with Filter Designer

This example shows how to design and implement a lowpass FIR filter using the window method with the interactive **Filter Designer** app.

- Start the app by entering `filterDesigner` at the command line.
- Set the **Response Type** to **Lowpass**.
- Set the **Design Method** to **FIR** and select the **Window** method.
- Under **Filter Order**, select **Specify order**. Set the order to 20.
- Under **Frequency Specifications**, set **Units** to **Hz**, **Fs** to 1000, and **Fc** to 150.

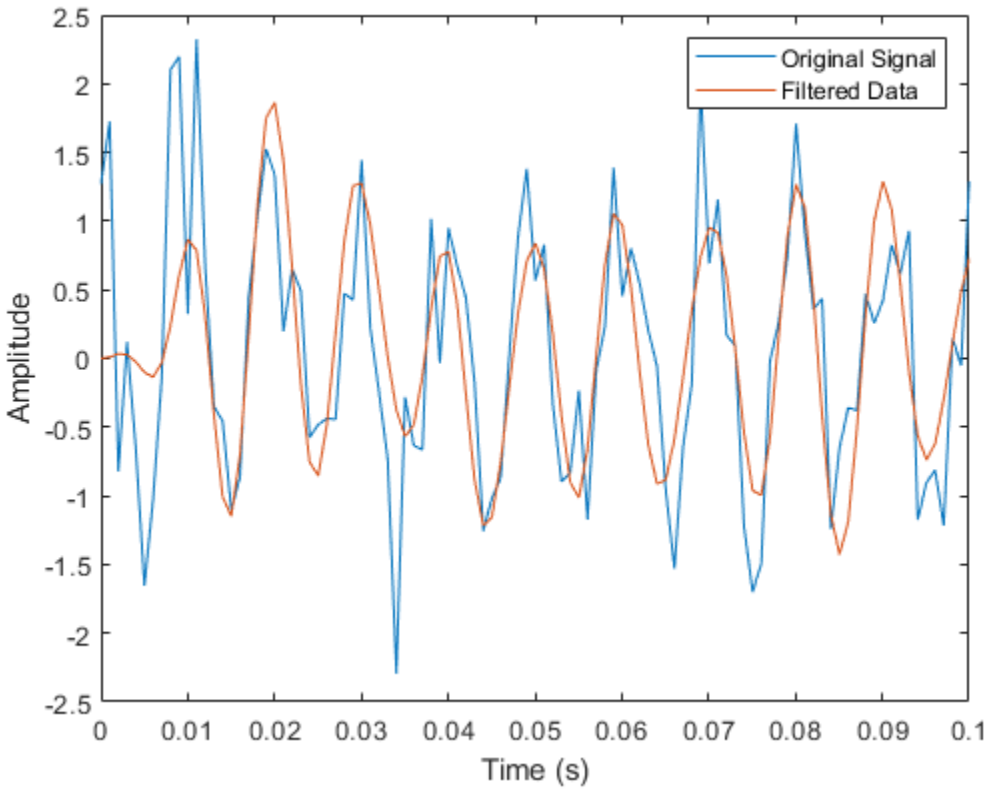



- Click **Design Filter**.
- Select **File > Export...** to export your FIR filter to the MATLAB® workspace as coefficients or a filter object. In this example, export the filter as an object. Specify the variable name as `Hd`.




- Click **Export**.
- Filter the input signal in the command window with the exported filter object. Plot the result for the first ten periods of the 100 Hz sinusoid.

```
y2 = filter(Hd,x);

plot(t,x,t,y2)
xlim([0 0.1])

xlabel('Time (s)')
ylabel('Amplitude')
legend('Original Signal','Filtered Data')
```




- Select **File > Generate MATLAB Code > Filter Design Function** to generate a MATLAB function to create a filter object using your specifications.

You can also use the interactive tool `filterBuilder` to design your filter.

**Bandpass Filters – Minimum-Order FIR and IIR Systems**

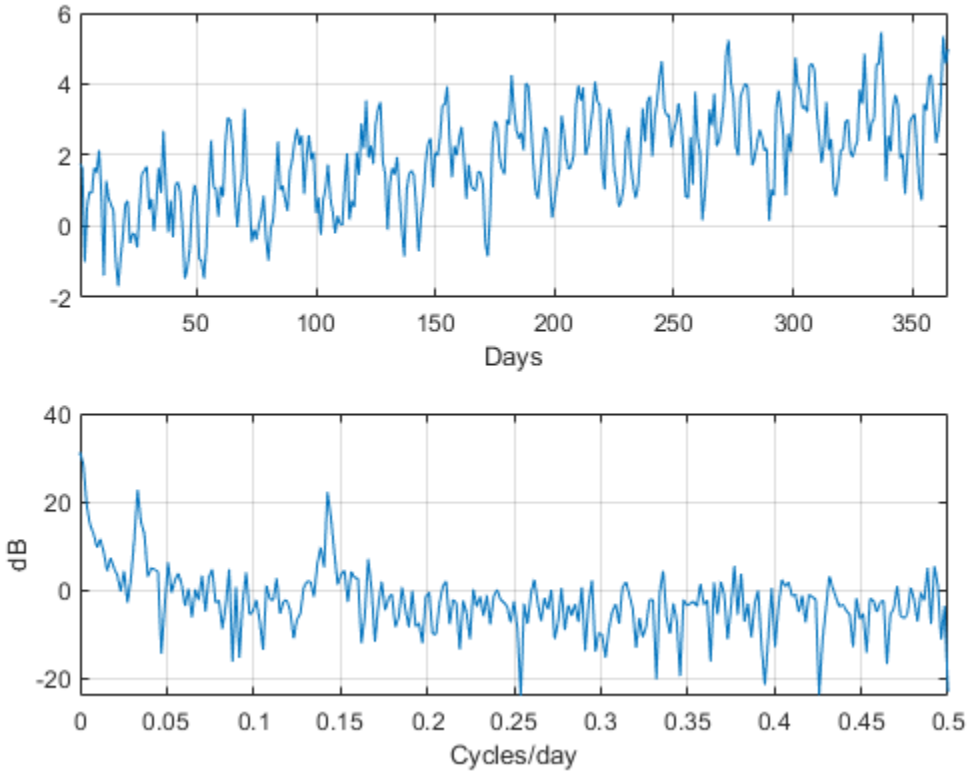
This example shows how to design a bandpass filter and filter data with minimum-order FIR equiripple and IIR Butterworth filters. You can model many real-world signals as a superposition of oscillating components, a low-frequency trend, and additive noise. For example, economic data often contain oscillations, which represent cycles superimposed on a slowly varying upward or downward trend. In addition, there is an additive noise component, which is a combination of measurement error and the inherent random fluctuations in the process.

In these examples, assume you sample some process every day for one year. Assume the process has oscillations on approximately one-week and one-month scales. In addition, there is a low-frequency upward trend in the data and additive $N(0, 1/4)$ white Gaussian noise.

Create the signal as a superposition of two sine waves with frequencies of 1/7 and 1/30 cycles/day. Add a low-frequency increasing trend term and $N(0, 1/4)$ white Gaussian noise. Reset the random number generator for reproducible results. The data is sampled at 1 sample/day. Plot the resulting signal and the power spectral density (PSD) estimate.

```
rng default

Fs = 1;
```

```
n = 1:365;

x = cos(2*pi*(1/7)*n)+cos(2*pi*(1/30)*n-pi/4);
trend = 3*sin(2*pi*(1/1480)*n);

y = x+trend+0.5*randn(size(n));

[pxx,f] = periodogram(y,[],[],Fs);

subplot(2,1,1)
plot(n,y)
xlim([1 365])
xlabel('Days')
grid

subplot(2,1,2)
plot(f,10*log10(pxx))
xlabel('Cycles/day')
ylabel('dB')
grid
```




The low-frequency trend appears in the power spectral density estimate as increased low-frequency power. The low-frequency power appears approximately 10 dB above the oscillation at 1/30 cycles/day. Use this information in the specifications for the filter stopbands.

Design minimum-order FIR equiripple and IIR Butterworth filters with the following specifications: passband from [1/40,1/4] cycles/day and stopbands from [0,1/60] and [1/4,1/2] cycles/day. Set both stopband attenuations to 10 dB and the passband ripple tolerance to 1 dB.

```
Hd1 = designfilt('bandpassfir', ...
    'StopbandFrequency1',1/60,'PassbandFrequency1',1/40, ...
    'PassbandFrequency2',1/4 ,'StopbandFrequency2',1/2 , ...
    'StopbandAttenuation1',10,'PassbandRipple',1, ...
    'StopbandAttenuation2',10,'DesignMethod','equiripple','SampleRate',Fs);
Hd2 = designfilt('bandpassiir', ...
    'StopbandFrequency1',1/60,'PassbandFrequency1',1/40, ...
    'PassbandFrequency2',1/4 ,'StopbandFrequency2',1/2 , ...
    'StopbandAttenuation1',10,'PassbandRipple',1, ...
    'StopbandAttenuation2',10,'DesignMethod','butter','SampleRate',Fs);
```

Compare the order of the FIR and IIR filters and the unwrapped phase responses.

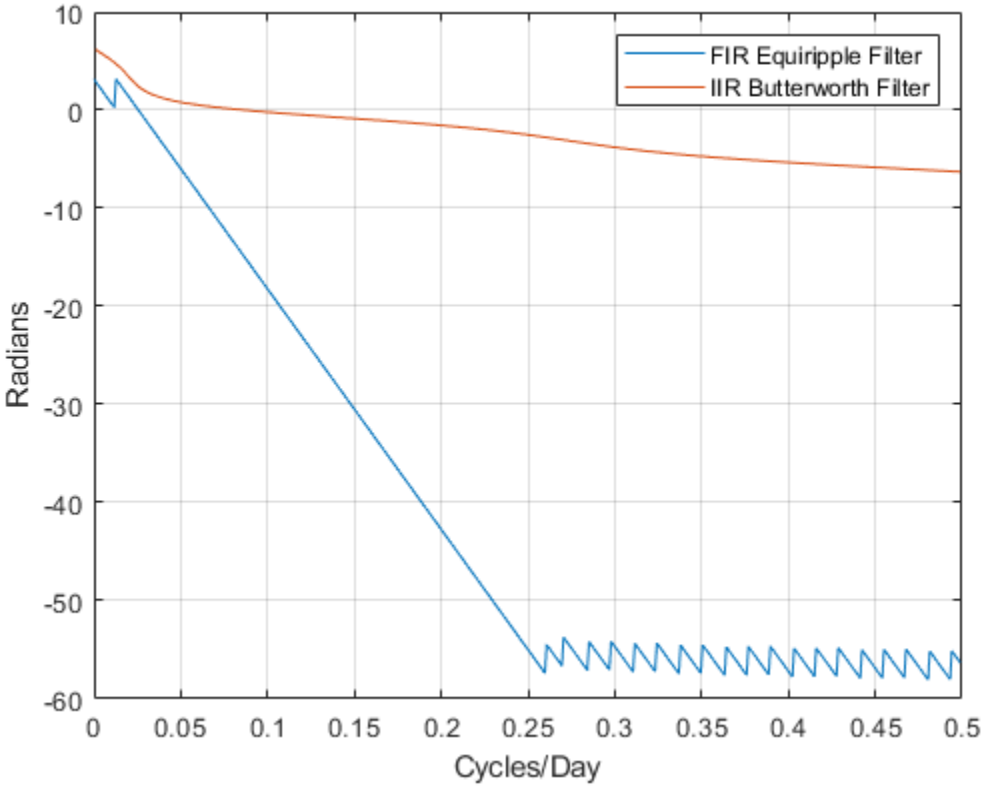
```
fprintf('The order of the FIR filter is %d\n',filtord(Hd1))
```

```
The order of the FIR filter is 78
```

```
fprintf('The order of the IIR filter is %d\n',filtord(Hd2))
```

```
The order of the IIR filter is 8
```

```
[phifir,w] = phasez(Hd1,[],1);
[phiiir,w] = phasez(Hd2,[],1);

figure
plot(w,unwrap(phifir))
hold on
plot(w,unwrap(phiiir))
hold off

xlabel('Cycles/Day')
ylabel('Radians')
legend('FIR Equiripple Filter','IIR Butterworth Filter')
grid
```




The IIR filter has a much lower order that the FIR filter. However, the FIR filter has a linear phase response over the passband, while the IIR filter does not. The FIR filter delays all frequencies in the filter passband equally, while the IIR filter does not.

Additionally, the rate of change of the phase per unit of frequency is greater in the FIR filter than in the IIR filter.

Design a lowpass FIR equiripple filter for comparison. The lowpass filter specifications are: passband [0,1/4] cycles/day, stopband attenuation equal to 10 dB, and the passband ripple tolerance set to 1 dB.

```
Hdlow = designfilt('lowpassfir', ...
    'PassbandFrequency',1/4,'StopbandFrequency',1/2, ...
    'PassbandRipple',1,'StopbandAttenuation',10, ...
    'DesignMethod','equiripple','SampleRate',1);
```

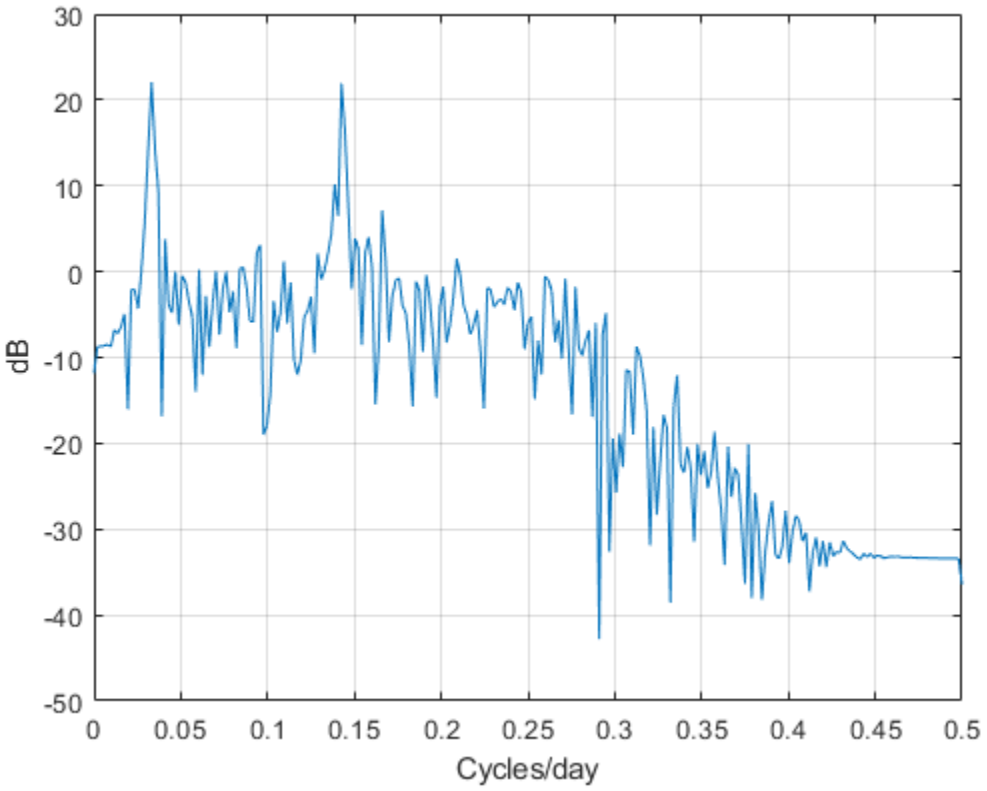
Filter the data with the bandpass and lowpass filters.

```
yfir = filter(Hd1,y);
yiir = filter(Hd2,y);
ylow = filter(Hdlow,y);
```

Plot the PSD estimate of the bandpass IIR filter output. You can replace `yiir` with `yfir` in the following code to view the PSD estimate of the FIR bandpass filter output.

```
[pxx,f] = periodogram(yiir,[],[],Fs);
```

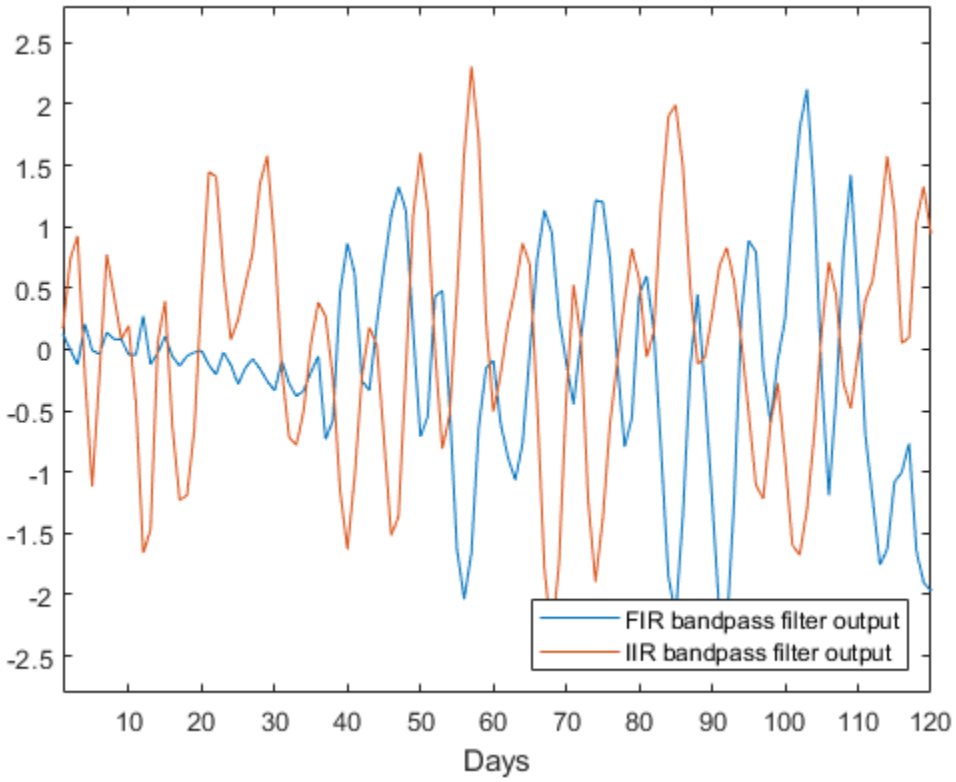
```
plot(f,10*log10(pxx))

xlabel('Cycles/day')
ylabel('dB')
grid
```




The PSD estimate shows the bandpass filter attenuates the low-frequency trend and high-frequency noise.

Plot the first 120 days of FIR and IIR filter output.

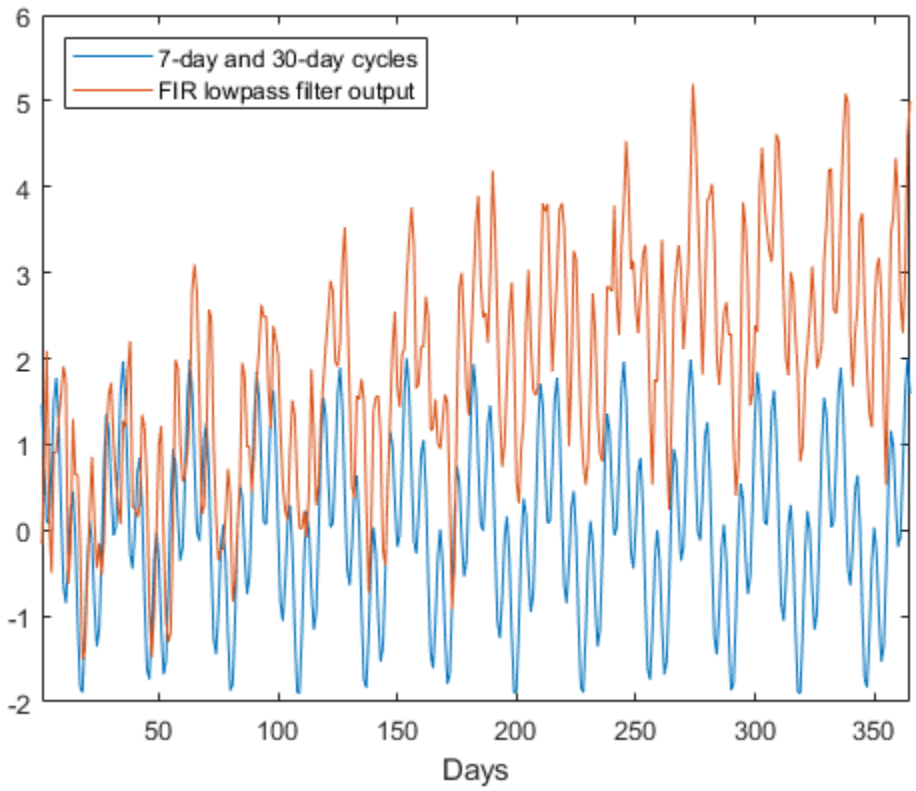
```
plot(n,yfir,n,yiir)

axis([1 120 -2.8 2.8])
xlabel('Days')
legend('FIR bandpass filter output','IIR bandpass filter output', ...
    'Location','SouthEast')
```




The increased phase delay in the FIR filter is evident in the filter output.

Plot the lowpass FIR filter output superimposed on the superposition of the 7-day and 30-day cycles for comparison.

```
plot(n,x,n,ylow)

xlim([1 365])
xlabel('Days')
legend('7-day and 30-day cycles','FIR lowpass filter output', ...
    'Location','NorthWest')
```




You can see in the preceding plot that the low-frequency trend is evident in the lowpass filter output. While the lowpass filter preserves the 7-day and 30-day cycles, the bandpass filters perform better in this example because the bandpass filters also remove the low-frequency trend.

### Zero-Phase Filtering

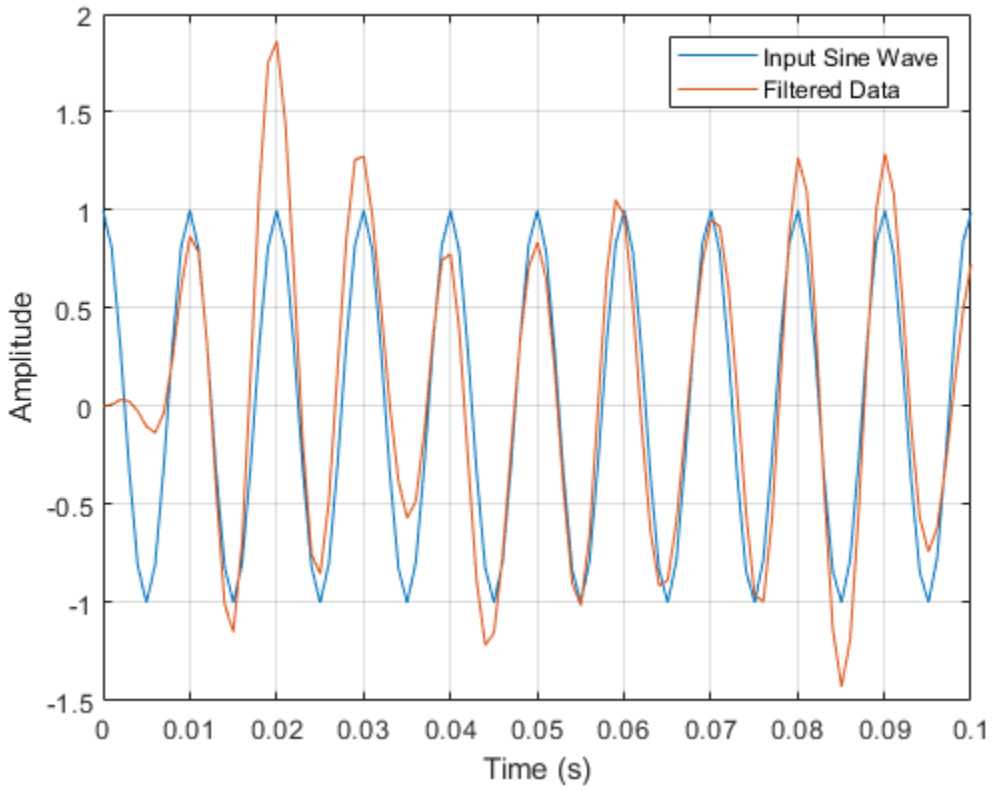
This example shows how to perform zero-phase filtering.

Repeat the signal generation and lowpass filter design with `fir1` and `designfilt`. You do not have to execute the following code if you already have these variables in your workspace.

```
rng default

Fs = 1000;
t = linspace(0,1,Fs);
x = cos(2*pi*100*t)+0.5*randn(size(t));

% Using fir1
fc = 150;
Wn = (2/Fs)*fc;
b = fir1(20,Wn,'low',kaiser(21,3));

% Using designfilt
Hd = designfilt('lowpassfir','FilterOrder',20,'CutoffFrequency',150, ...
       'DesignMethod','window','Window',{@kaiser,3},'SampleRate',Fs);
```

Filter the data using `filter`. Plot the first 100 points of the filter output along with a superimposed sinusoid with the same amplitude and initial phase as the input signal.

```
yout = filter(Hd,x);
xin = cos(2*pi*100*t);

plot(t,xin,t,yout)
xlim([0 0.1])

xlabel('Time (s)')
ylabel('Amplitude')
legend('Input Sine Wave','Filtered Data')
grid
```

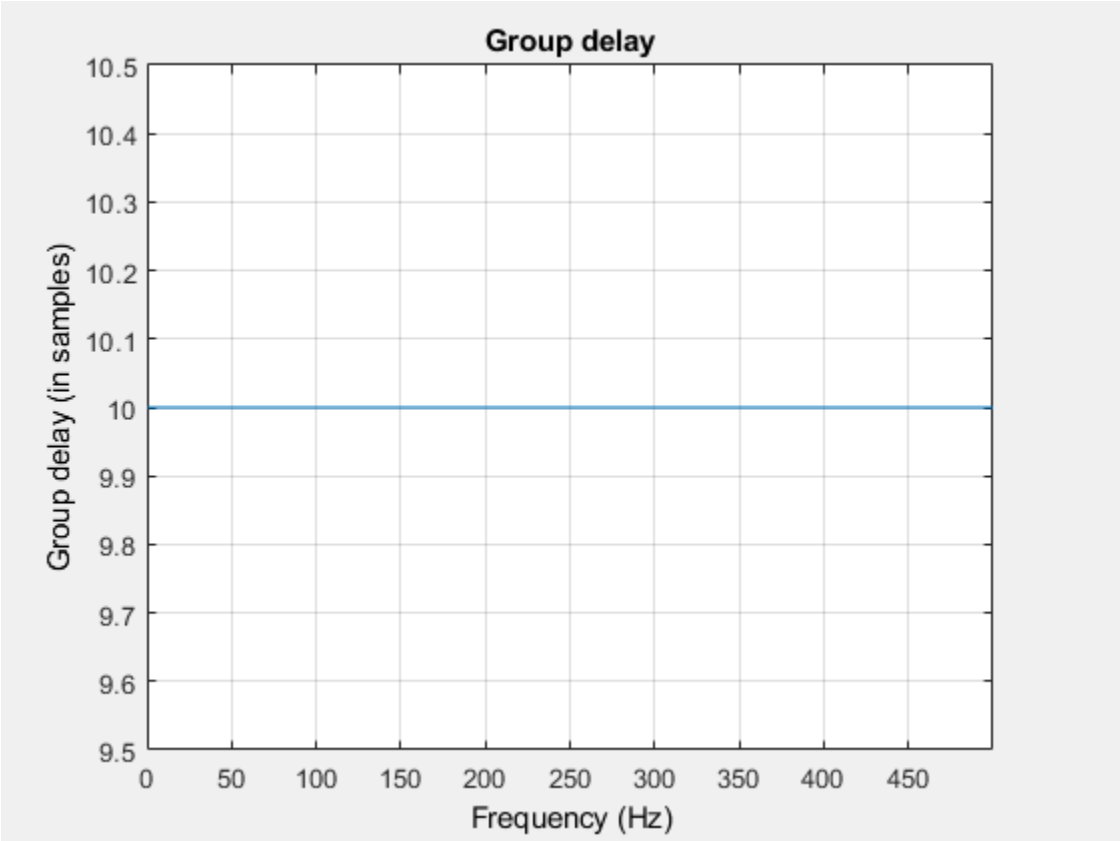



Looking at the initial 0.01 seconds of the filtered data, you see that the output is delayed with respect to the input. The delay appears to be approximately 0.01 seconds, which is almost 1/2 the length of the FIR filter in samples ($10 \times 0.001$).

This delay is due to the filter's phase response. The FIR filter in these examples is a type I linear-phase filter. The group delay of the filter is 10 samples.

Plot the group delay using `fvtool`.

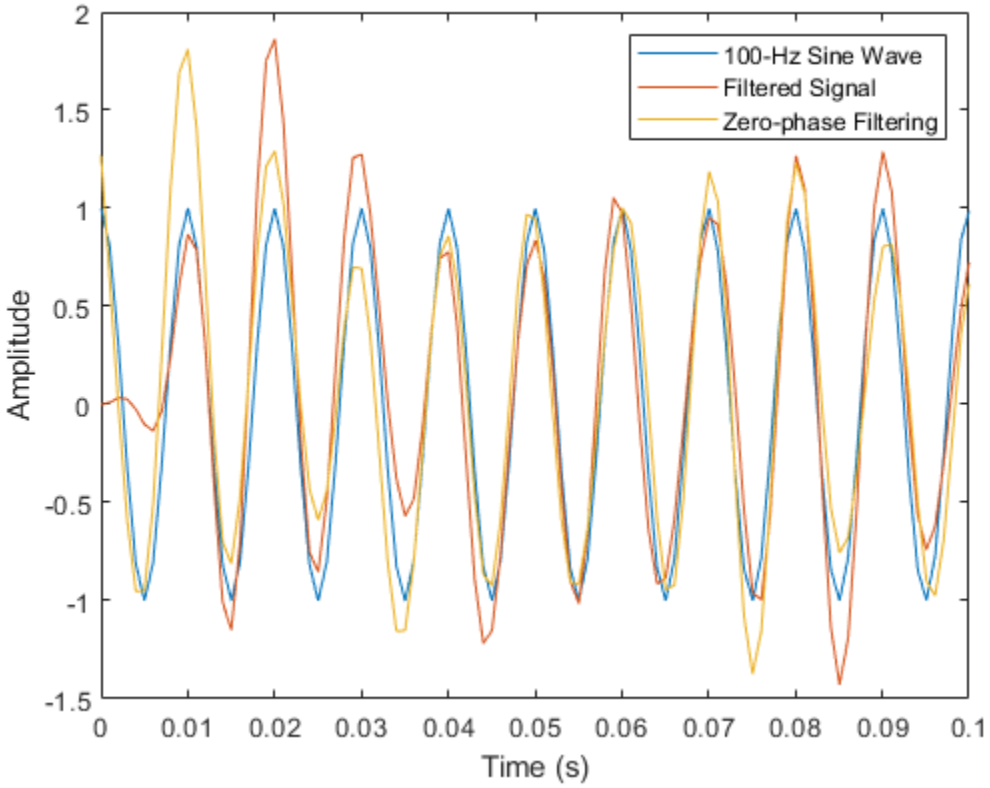
```
fvtool(Hd,'Analysis','grpdelay')
```




In many applications, phase distortion is acceptable. This is particularly true when phase response is linear. In other applications, it is desirable to have a filter with a zero-phase response. A zero-phase response is not technically possibly in a noncausal filter. However, you can implement zero-phase filtering using a causal filter with `filtfilt`.

Filter the input signal using `filtfilt`. Plot the responses to compare the filter outputs obtained with `filter` and `filtfilt`.

```
yzp = filtfilt(Hd,x);

plot(t,xin,t,yout,t,yzp)

xlim([0 0.1])
xlabel('Time (s)')
ylabel('Amplitude')
legend('100-Hz Sine Wave','Filtered Signal','Zero-phase Filtering',...
    'Location','NorthEast')
```




In the preceding figure, you can see that the output of `filtfilt` does not exhibit the delay due to the phase response of the FIR filter.

# Selected Bibliography

[1] Karam, Lina J., and James H. McClellan. “Complex Chebyshev Approximation for FIR Filter Design.” *IEEE® Transactions on Circuits and Systems II: Analog and Digital Signal Processing*. Vol. 42, March 1995, pp. 207–216.

[2] Selesnick, Ivan W., and C. Sidney Burrus. “Generalized Digital Butterworth Filter Design.” *IEEE Transactions on Signal Processing*. Vol. 46, June 1998, pp. 1688–1694.

[3] Selesnick, Ivan W., Markus Lang, and C. Sidney Burrus. “Constrained Least Square Design of FIR Filters without Specified Transition Bands.” *IEEE Transactions on Signal Processing*. Vol. 44, August 1996, pp. 1879–1892.

# Designing a Filter in `fdesign` — Process Overview

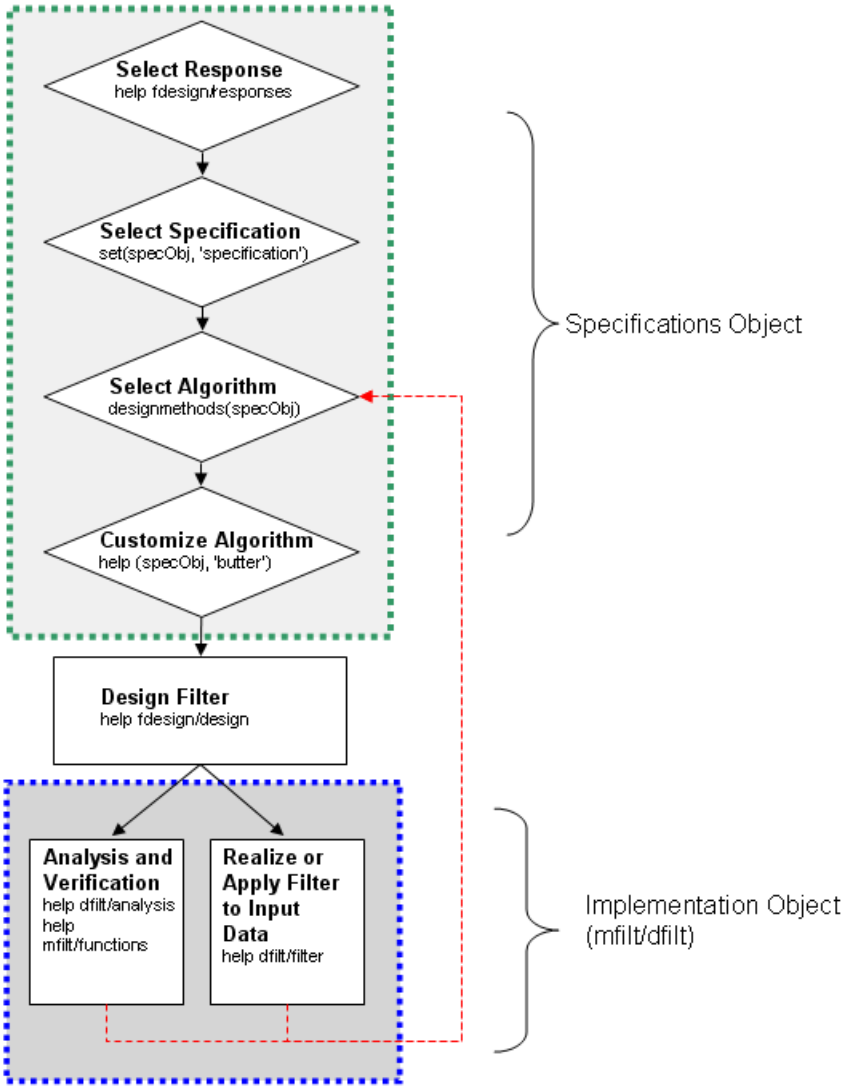
# Process Flow Diagram and Filter Design Methodology

| In this section... |
| --- |
| “Exploring the Process Flow Diagram” on page 3-2<br>“Selecting a Response” on page 3-4<br>“Selecting a Specification” on page 3-4<br>“Selecting an Algorithm” on page 3-5<br>“Customizing the Algorithm” on page 3-6<br>“Designing the Filter” on page 3-6<br>“Design Analysis” on page 3-7<br>“Realize or Apply the Filter to Input Data” on page 3-7 |

**Note** You must minimally have the Signal Processing Toolbox installed to use `fdesign` and `design`. Some of the features described below may be unavailable if your installation does not additionally include the DSP System Toolbox™ license. The DSP System Toolbox significantly expands the functionality available for the specification, design, and analysis of filters. You can verify the presence of both toolboxes by typing `ver` at the command prompt.

## Exploring the Process Flow Diagram

The process flow diagram shown in the following figure lists the steps and shows the order of the filter design process.




The first four steps of the filter design process relate to the filter Specifications Object, while the last two steps involve the filter Implementation Object. Both of these objects are discussed in more detail in the following sections. Step 5 - the design of the filter, is the transition step from the filter Specifications Object to the Implementation object. The analysis and verification step is completely optional. It provides methods for the filter designer to ensure that the filter complies with all design criteria. Depending on the results of this verification, you can loop back to steps 3 and 4, to either choose a different algorithm, or to customize the current one. You may also wish to go back to steps 3 or 4 after you filter the input data with the designed filter (step 7), and find that you wish to tweak the filter or change it further.

The diagram shows the help command for each step. Enter the help line at the MATLAB command prompt to receive instructions and further documentation links for the particular step. Not all of the steps have to be executed explicitly. For example, you could go from step 1 directly to step 5, and the interim three steps are done for you by the software.

The following are the details for each of the steps shown above.

## Selecting a Response

If you type:

```
help fdesign/responses
```

at the MATLAB command prompt, you see a list of all available filter responses. The responses marked with an asterisk require the DSP System Toolbox.

You must select a response to initiate the filter. In this example, a bandpass filter Specifications Object is created by typing the following:

```
d = fdesign.bandpass
```

## Selecting a Specification

A *specification* is an array of design parameters for a given filter. The specification is a property of the Specifications Object.

---

**Note** A specification is not the same as the Specifications Object. A Specifications Object contains a specification as one of its properties.

---

When you select a filter response, there are a number of different specifications available. Each one contains a different combination of design parameters. After you create a filter Specifications Object, you can query the available specifications for that response. Specifications marked with an asterisk require the DSP System Toolbox.

```
d = fdesign.bandpass;
set(d,'specification')

ans =

    'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
    'N,F3dB1,F3dB2'
    'N,F3dB1,F3dB2,Ap'
    'N,F3dB1,F3dB2,Ast'
    'N,F3dB1,F3dB2,Ast1,Ap,Ast2'
    'N,F3dB1,F3dB2,BWp'
    'N,F3dB1,F3dB2,BWst'
    'N,Fc1,Fc2'
    'N,Fp1,Fp2,Ap'
    'N,Fp1,Fp2,Ast1,Ap,Ast2'
    'N,Fst1,Fp1,Fp2,Fst2'
    'N,Fst1,Fp1,Fp2,Fst2,Ap'
    'N,Fst1,Fst2,Ast'
    'Nb,Na,Fst1,Fp1,Fp2,Fst2'

d = fdesign.arbmag;
set(d,'specification')

ans =

    'N,F,A'
    'N,B,F,A'
```

The `set` command can be used to select one of the available specifications as follows:

```
d = fdesign.lowpass;
set(d,'specification', 'N,Fc')
```

If you do not perform this step explicitly, `fdesign` returns the default specification for the response you chose in “Select a Response” on page 4-2, and provides default values for all design parameters included in the specification.

## Selecting an Algorithm

The availability of algorithms depends the chosen filter response, the design parameters, and the availability of the DSP System Toolbox. In other words, for the same lowpass filter, changing the specification also changes the available algorithms. In the following example, for a lowpass filter and a specification of `'N, Fc'`, only one algorithm is available—`window`.

```
set (d, 'specification', 'N,Fc')
designmethods (d) %step3: get available algorithms

Design Methods for class fdesign.lowpass (N,Fc):


window
```

However, for a specification of `'Fp,Fst,Ap,Ast'`, a number of algorithms are available. If the user has only the Signal Processing Toolbox installed, the following algorithms are available:

```
set(d,'specification','Fp,Fst,Ap,Ast')
designmethods(d)

Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):

butter
cheby1
cheby2
ellip
equiripple
kaiserwin
```

If the user additionally has the DSP System Toolbox installed, the number of available algorithms for this response and specification increases:

```
set(d,'specification','Fp,Fst,Ap,Ast')
designmethods(d)

Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):

butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

The user chooses a particular algorithm and implements the filter with the `design` function.

```
Hd=design(d,'butter');
```

The preceding code actually creates the filter. If you do not perform this step explicitly, `design` automatically selects the optimum algorithm for the chosen response and specification.

## Customizing the Algorithm

The customization options available for any given algorithm depend not only on the algorithm itself, selected in "Selecting an Algorithm" on page 3-5, but also on the specification selected in "Selecting a Specification" on page 3-4. To explore all the available options, type the following at the MATLAB command prompt:

```
help(d,'algorithm-name')
```

where `d` is the Filter Specification Object, and `algorithm-name` is the name of the algorithm in single quotes, such as `'butter'` or `'cheby1'`.

The application of these customization options takes place while "Designing the Filter" on page 3-6, because these options are the properties of the filter Implementation Object, not the Specification Object.

If you do not perform this step explicitly, the optimum algorithm structure is selected.

## Designing the Filter

To create a filter, use the `design` command:

```
Hd = design(d);
```

where `d` is the Specifications Object. This code creates a filter without specifying the algorithm. When the algorithm is not specified, the software selects the best available one.

To apply the algorithm chosen in "Selecting an Algorithm" on page 3-5, use the same `design` command, but specify the Butterworth algorithm as follows:

```
Hd = design(d,'butter');
```

To obtain help and see all the available options, type:

```
help fdesign/design
```

This help command describes not only the options for the `design` command itself, but also options that pertain to the method or the algorithm. If you are customizing the algorithm, you apply these options in this step. In the following example, you design a bandpass filter, and then modify the filter structure:

```
Hd = design(d,'butter','FilterStructure','df2sos')

Hd =

          FilterStructure: 'Direct-Form II, Second-Order Sections'
               Arithmetic: 'double'
                sosMatrix: [13x6 double]
              ScaleValues: [14x1 double]
     OptimizeScaleValues: true
        PersistentMemory: false
```

The filter design step, just like the first task of choosing a response, must be performed explicitly. The filter is created only when `design` is called.

## Design Analysis

After the filter is designed you may wish to analyze it to determine if the filter satisfies the design criteria. Filter analysis is broken into three main sections:

- Frequency domain analysis — Includes the magnitude response, group delay, and pole-zero plots.
- Time domain analysis — Includes impulse and step response
- Implementation analysis — Includes quantization noise and cost

To display help for analysis of a discrete-time filter, type:

```
>> help dfilt/analysis
```

To display help for analysis of a farrow filter, type:

```
>> help farrow/functions
```

To analyze your filter, you must explicitly perform this step.

## Realize or Apply the Filter to Input Data

After the filter is designed and optimized, it can be used to filter actual input data. The basic filter command takes input data `x`, filters it through the Filter Object, and produces output `y`:

```
>> y = filter (FilterObj, x)
```

This step is never automatically performed for you. To filter your data, you must explicitly execute this step. To understand how the filtering commands work, type:

```
>> help dfilt/filter
```

---

**Note** If you have Simulink®, you have the option of exporting this filter to a Simulink block using the `realizemdl` command. To get help on this command, type:

```
>> help realizemdl
```

---

# Designing a Filter in the Filter Builder GUI


- “Filter Builder Design Process” on page 4-2
- “Designing a FIR Filter Using filterBuilder” on page 4-9
- “Compensate for Delay and Distortion Introduced by Filters” on page 4-12
- “Comparison of Analog IIR Lowpass Filters” on page 4-19
- “Frequency Response of Lowpass Bessel Filter” on page 4-21
- “Speaker Crossover Filters” on page 4-23

# Filter Builder Design Process

| In this section... |
| --- |
| “Introduction to Filter Builder” on page 4-2 |
| “Design a Filter Using Filter Builder” on page 4-2 |
| “Select a Response” on page 4-2 |
| “Select a Specification” on page 4-4 |
| “Select an Algorithm” on page 4-5 |
| “Customize the Algorithm” on page 4-5 |
| “Analyze the Design” on page 4-6 |
| “Realize or Apply the Filter to Input Data” on page 4-7 |

## Introduction to Filter Builder

The `filterBuilder` function provides a graphical interface to the `fdesign` object-oriented filter design paradigm and is intended to reduce development time during the filter design process. `filterBuilder` uses a specification-centered approach to find the best algorithm for the desired response.

**Note** `filterBuilder` requires the Signal Processing Toolbox. The DSP System Toolbox product greatly expands the functionality of `filterBuilder`. Many of the features described or displayed on this page are only available if the DSP System Toolbox is installed. You may verify your installation by typing `ver` at the command prompt.

## Design a Filter Using Filter Builder

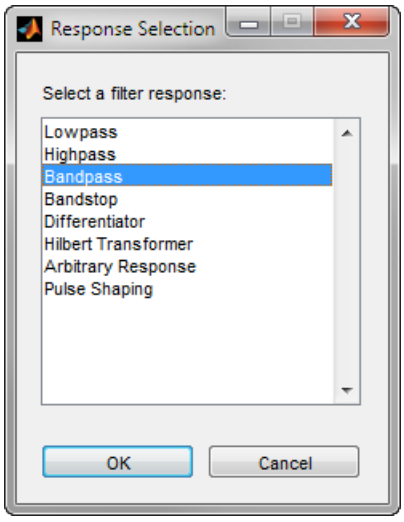
The basic workflow in using `filterBuilder` is to choose the constraints and specifications of the filter, and to use those constraints as a starting point in the design. Postponing the choice of algorithm for the filter allows the best design method to be determined automatically, based on the desired performance criteria. The following are the details of each of the steps for designing a filter with `filterBuilder`.

## Select a Response

When you open the `filterBuilder` tool by typing:

```
filterBuilder
```

at the MATLAB command prompt, the **Response Selection** dialog box appears, listing all possible filter responses available in DSP System Toolbox.

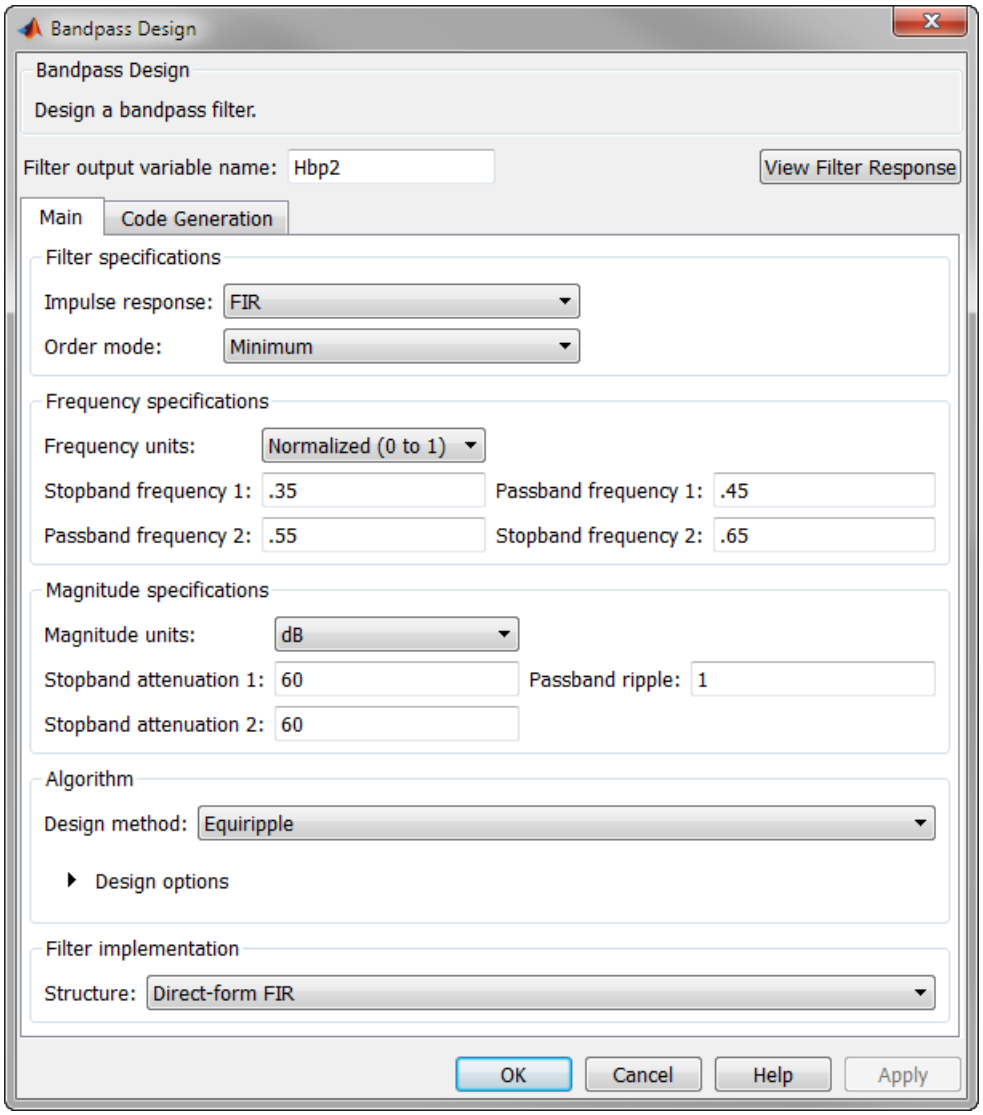



**Note** This step cannot be skipped because it is not automatically completed for you by the software. You must select a response to initiate the filter design process.

After you choose a response, say bandpass, you start the design of the Specifications Object, and the Bandpass Design dialog box appears. This dialog box contains a **Main** pane, a **Data Types** pane, and a **Code Generation** pane. The specifications of your filter are generally set in the **Main** pane of the dialog box.

The **Data Types** pane provides settings for precision and data types, and the **Code Generation** pane contains options for various implementations of the completed filter design.

For the initial design of your filter, you mostly use the **Main** pane.




The **Bandpass Design** dialog box contains all the parameters necessary to determine the specifications of a bandpass filter. The parameters listed in the **Main** pane depend upon the type of filter you are designing. However, no matter what type of filter you have chosen in the **Response Selection** dialog box, the filter design dialog box contains the **Main**, **Data Types**, and **Code Generation** panes.

## Select a Specification

To choose the specification for the bandpass filter, you can begin by selecting an **Impulse Response**, **Order Mode**, and **Filter Type** in the **Filter Specifications** frame of the **Main Pane**. You can further specify the response of your filter by setting frequency and magnitude specifications in the appropriate frames on the **Main Pane**.

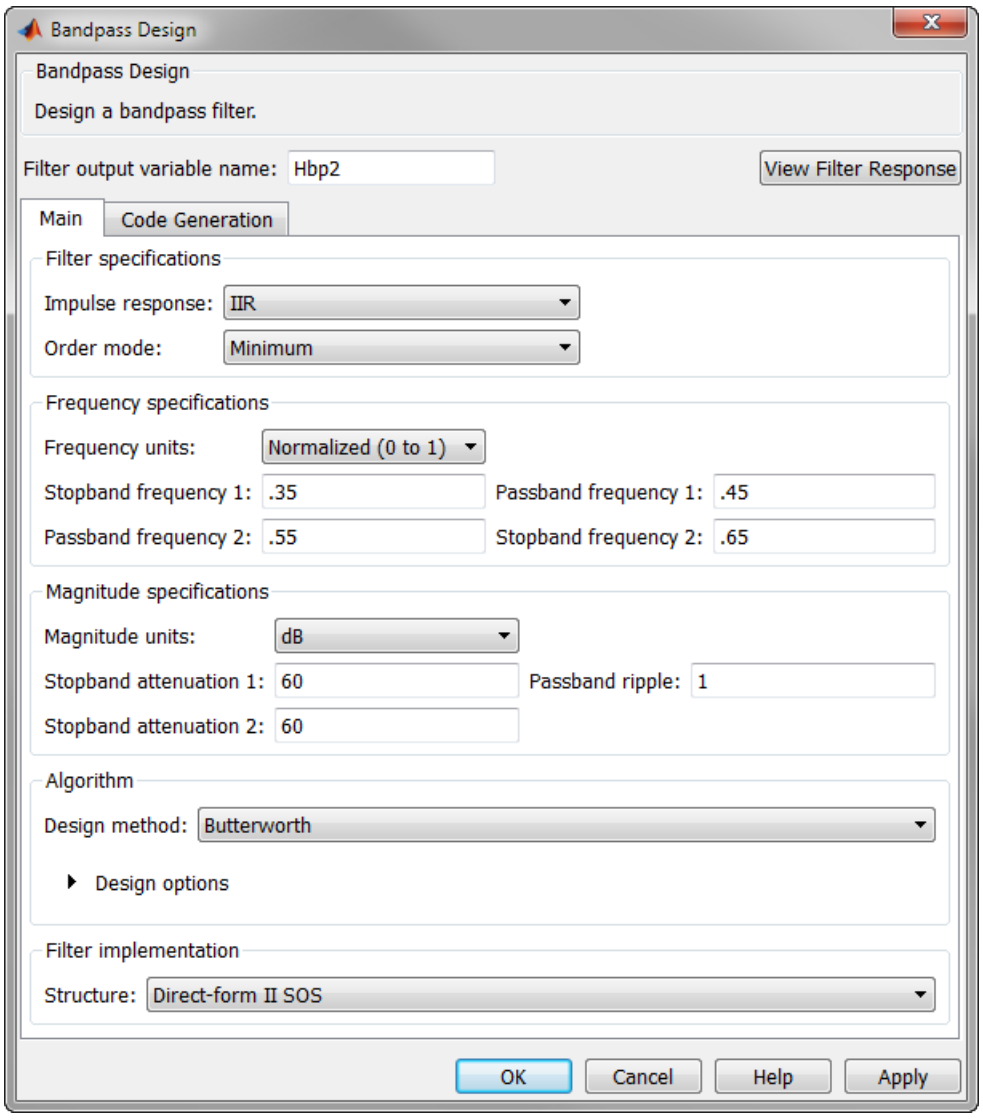
---

**Note** **Frequency**, **Magnitude**, and **Algorithm** specifications are interdependent and might change based on your **Filter Specifications** selections. When choosing specifications for your filter, select

your Filter Specifications first and work your way down the dialog box. This approach ensures that the best settings for dependent specifications display as available in the dialog box.

## Select an Algorithm

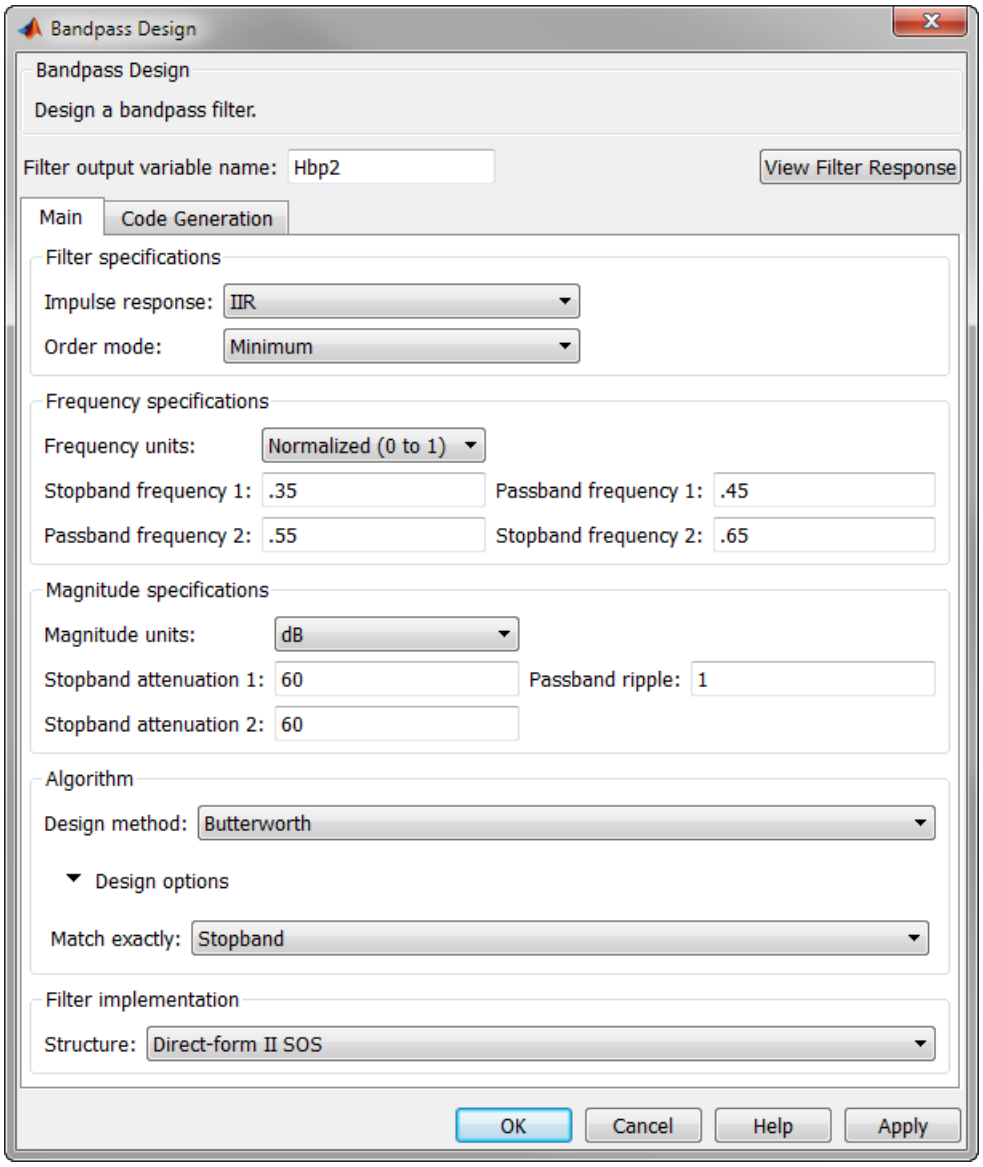
The algorithms available for your filter depend upon the filter response and design parameters you have selected in the previous steps. For example, in the case of a bandpass filter, if the impulse response selected is IIR and the **Order Mode** field is set to `Minimum`, the design methods available are `Butterworth`, `Chebyshev` type I or II, or `Elliptic`. If the **Order Mode** field is set to `Specify`, the design method available is `IIR least p-norm`.




## Customize the Algorithm

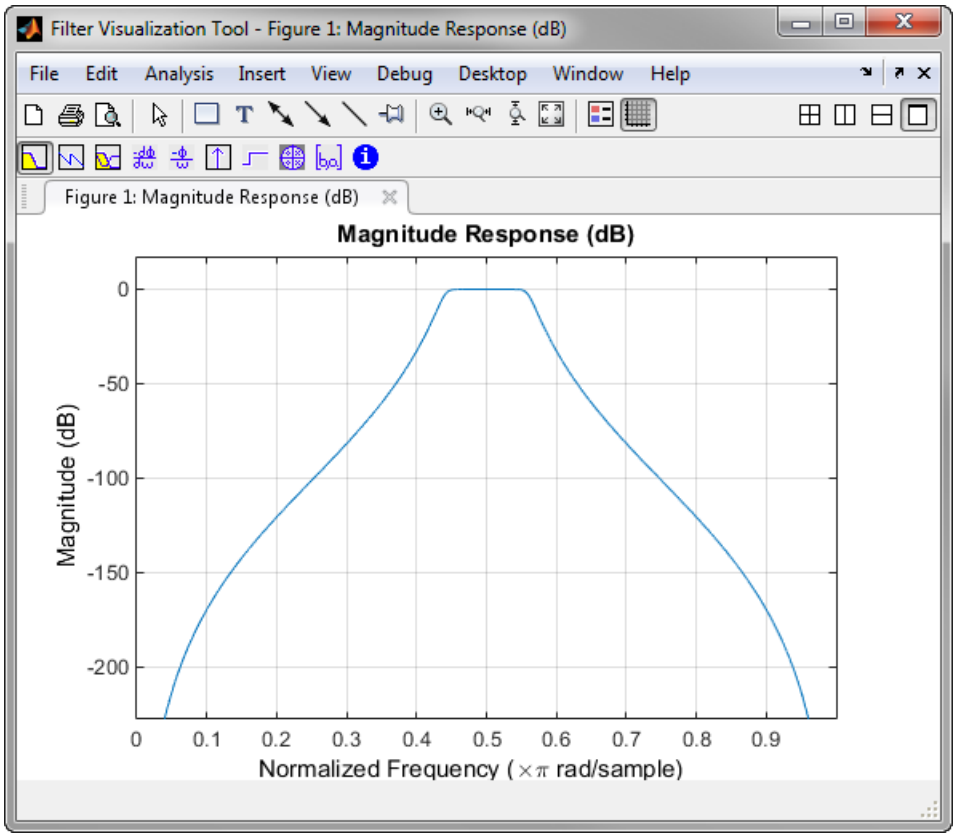
By expanding the **Design options** section of the **Algorithm** frame, you can further customize the algorithm specified. The options available depend upon the algorithm and settings that have already

been selected in the dialog box. In the case of a bandpass IIR filter using the `Butterworth` method, design options such as **Match Exactly** are available, as shown in the following figure.




## Analyze the Design

To analyze the filter response, click the View Filter Response button. The Filter Visualization Tool (**FVTool**) opens displaying the magnitude plot of the filter response.




## Realize or Apply the Filter to Input Data

When you have achieved the desired filter response through design iterations and analysis using the **Filter Visualization Tool**, apply the filter to the input data. Again, this step is never automatically performed for you by the software. To filter your data, you must explicitly execute this step. In the **Bandpass Design** dialog box, click OK and the Signal Processing Toolbox software creates the filter coefficients and exports it to the MATLAB workspace.

The filter is then ready to be used to filter actual input data. The basic filter command takes input data `x`, filters it through the Filter Object, and produces output `y`:

```
y = filter(Hbs,x)
```

To understand how the filtering command works, type:

```
help dfilt/filter
```

---

**Tip** If you have Simulink, you have the option of exporting this filter to a Simulink block using the `realizemdl` command. To get help on this command, type:

```
help realizemdl
```

---

## See Also

**FVTool**

# Designing a FIR Filter Using filterBuilder

## FIR Filter Design

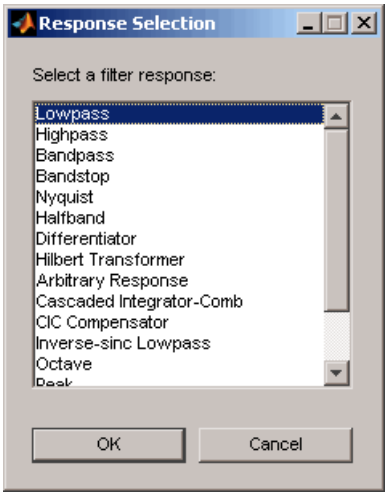
### Example 4.1. Example – Using Filter Builder to Design an FIR Filter

To design a lowpass finite impulse response (FIR) filter using `filterBuilder`:

1 Open the Filter Builder GUI by typing the following at the MATLAB prompt:

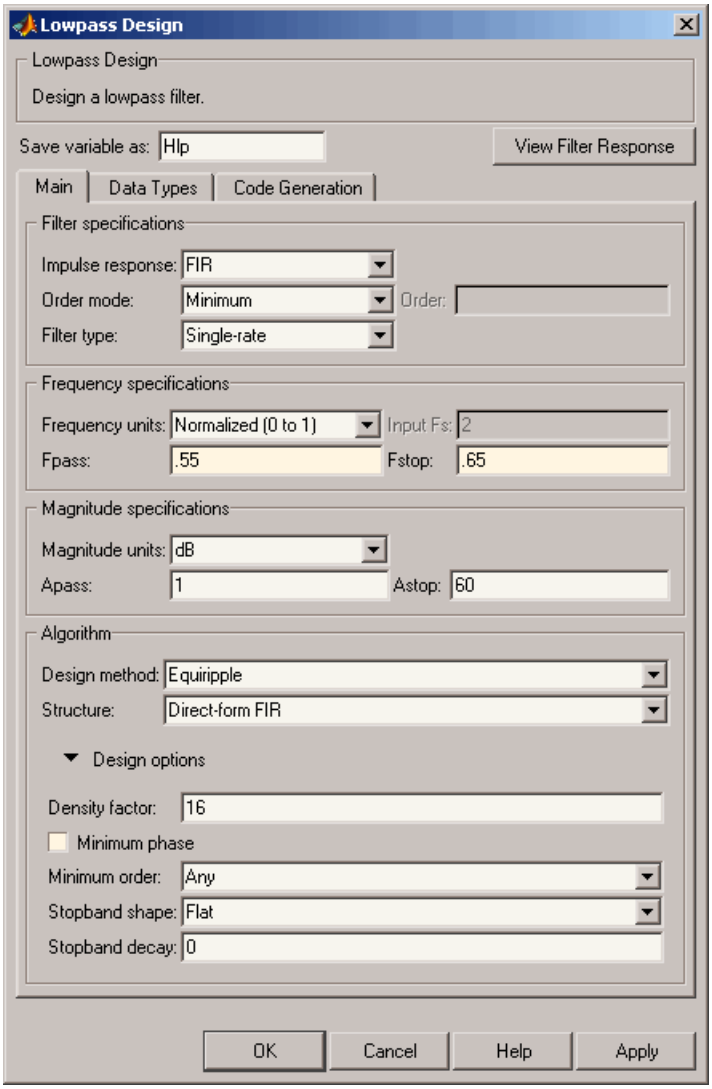
```
filterBuilder
```

The **Response Selection** dialog box appears. In this dialog box, you can select from a list of filter response types. Select `Lowpass` in the list box.




2 Hit the **OK** button. The **Lowpass Design** dialog box opens. Here you can specify the writable parameters of the Lowpass filter object. The components of the **Main** frame of this dialog box are described in the section titled *Lowpass Filter Design Dialog Box — Main Pane*. In the dialog box, make the following changes:

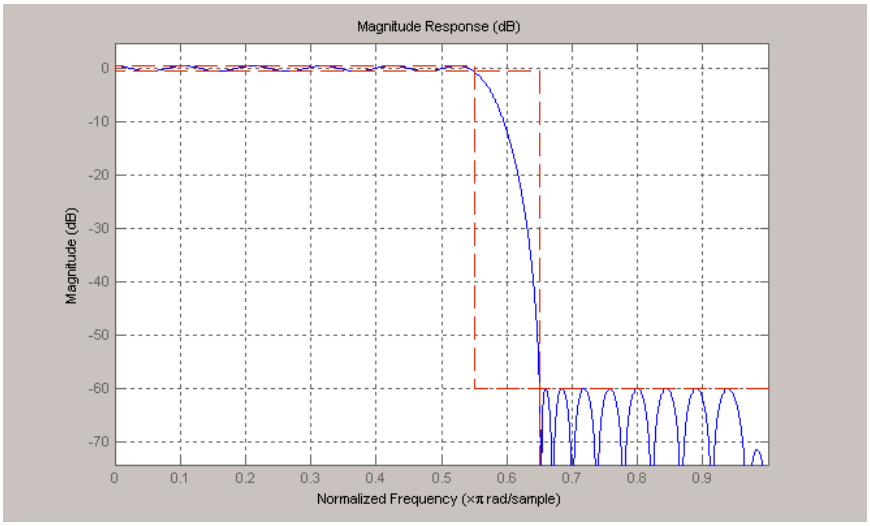
- Enter a Fpass value of 0.55.
- Enter a Fstop value of 0.65.




3 Click **Apply**, and the following message appears at the MATLAB prompt:

```
The variable 'Hlp' has been exported to the command window.
```

4 To check your design, click **View Filter Response**. The Filter Visualization tool appears, showing a plot of the magnitude response of the filter.




You can change the design and click **Apply**, followed by **View Filter Response**, as many times as needed until your design specifications are met.

# Compensate for Delay and Distortion Introduced by Filters

Filtering a signal introduces a delay. This means that the output signal is shifted in time with respect to the input.

When the shift is constant, you can correct for the delay by shifting the signal in time.

Sometimes the filter delays some frequency components more than others. This phenomenon is called phase distortion. To compensate for this effect, you can perform zero-phase filtering using the `filtfilt` function.

Take an electrocardiogram reading sampled at 500 Hz for 1 s. Add random noise. Reset the random number generator for reproducible results

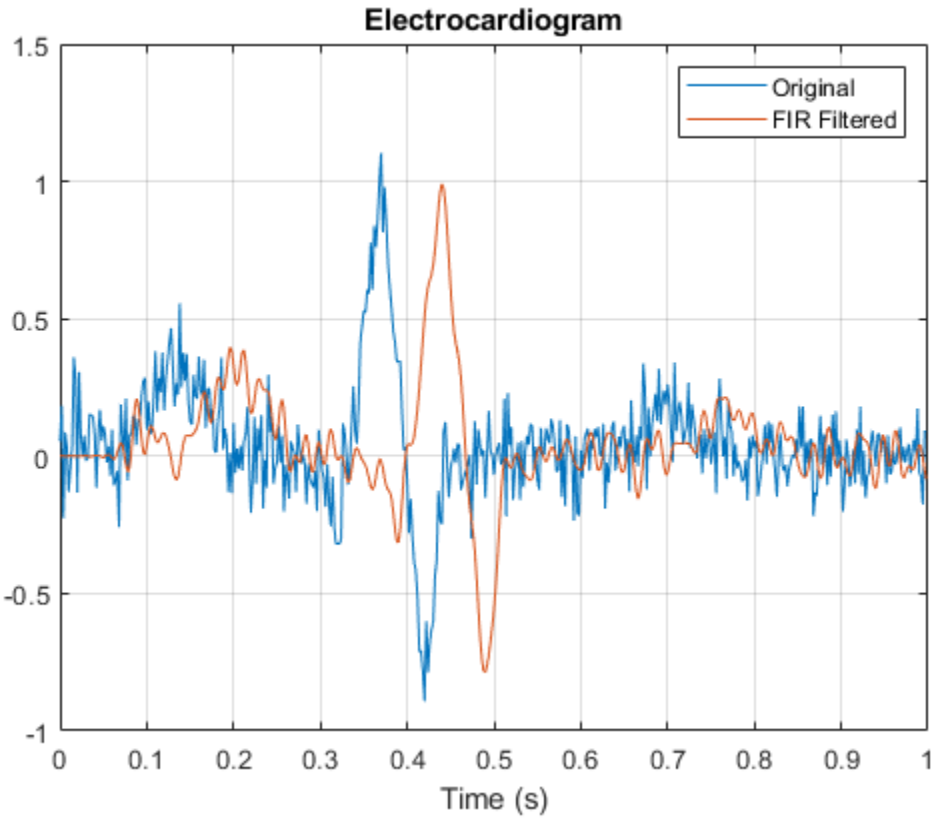
```
Fs = 500;
N = 500;

rng default

xn = ecg(N)+0.1*randn([1 N]);
tn = (0:N-1)/Fs;
```

Remove some of the noise using a filter that stops frequencies above 75 Hz. Use `designfilt` to design an FIR filter of order 70.

```
Nfir = 70;
Fst = 75;

firf = designfilt('lowpassfir','FilterOrder',Nfir, ...
    'CutoffFrequency',Fst,'SampleRate',Fs);
```

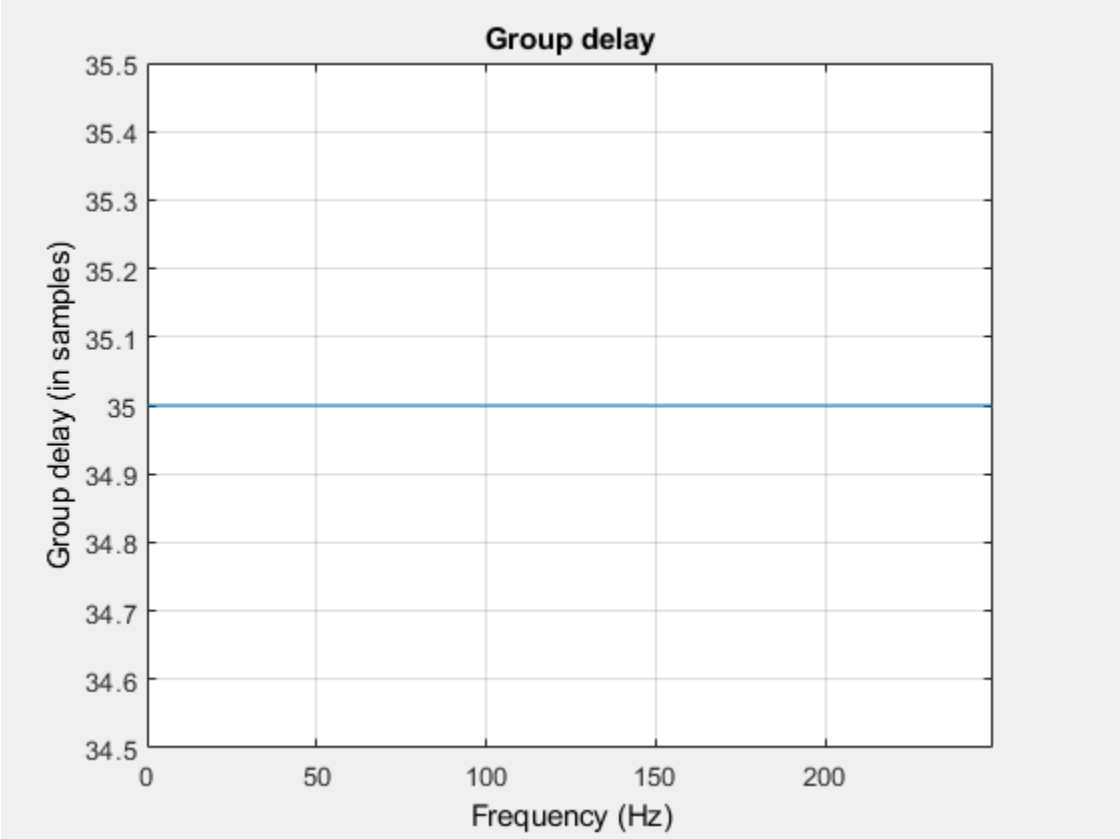
Filter the signal and plot it. The result is smoother than the original, but lags behind it.

```
xf = filter(firf,xn);

plot(tn,xn,tn,xf)
title 'Electrocardiogram'
xlabel 'Time (s)'
legend('Original','FIR Filtered')
grid
```




Use `grpdelay` to check that the delay caused by the filter equals half the filter order.

```
grpdelay(firf,N,Fs)
```




```
delay = mean(grpdelay(firf))
```

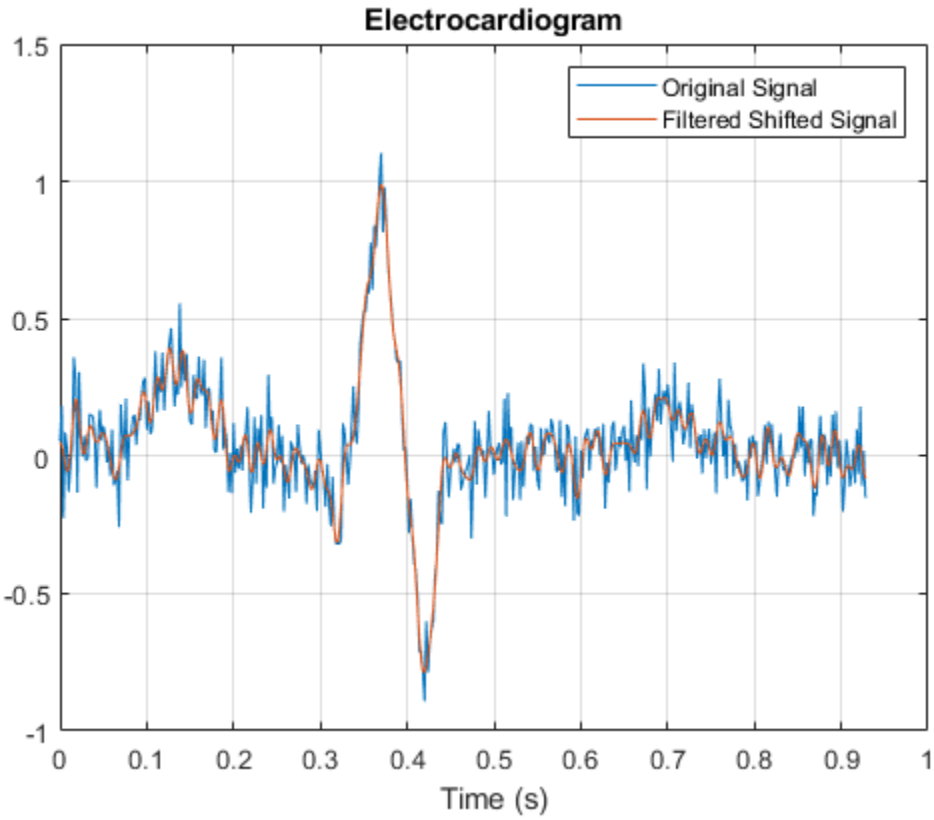
```
delay = 35
```

Line up the data. Shift the filtered signal by removing its first `delay` samples. Remove the last `delay` samples of the original and of the time vector.

```
tt = tn(1:end-delay);
sn = xn(1:end-delay);

sf = xf;
sf(1:delay) = [];
```

Plot the signals and verify that they are aligned.

```
plot(tt,sn,tt,sf)
title 'Electrocardiogram'
xlabel('Time (s)')
legend('Original Signal','Filtered Shifted Signal')
grid
```

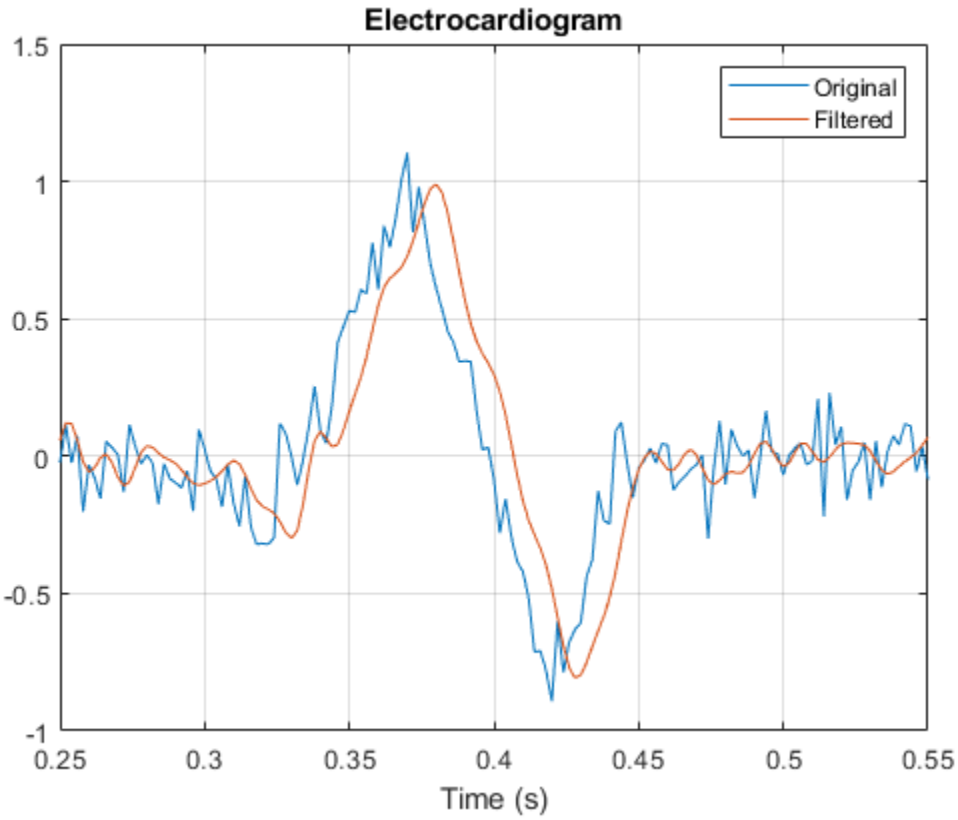



Repeat the computation using a 7th-order IIR filter.

```
Niir = 7;

iir = designfilt('lowpassiir','FilterOrder',Niir, ...
    'HalfPowerFrequency',Fst,'SampleRate',Fs);
```

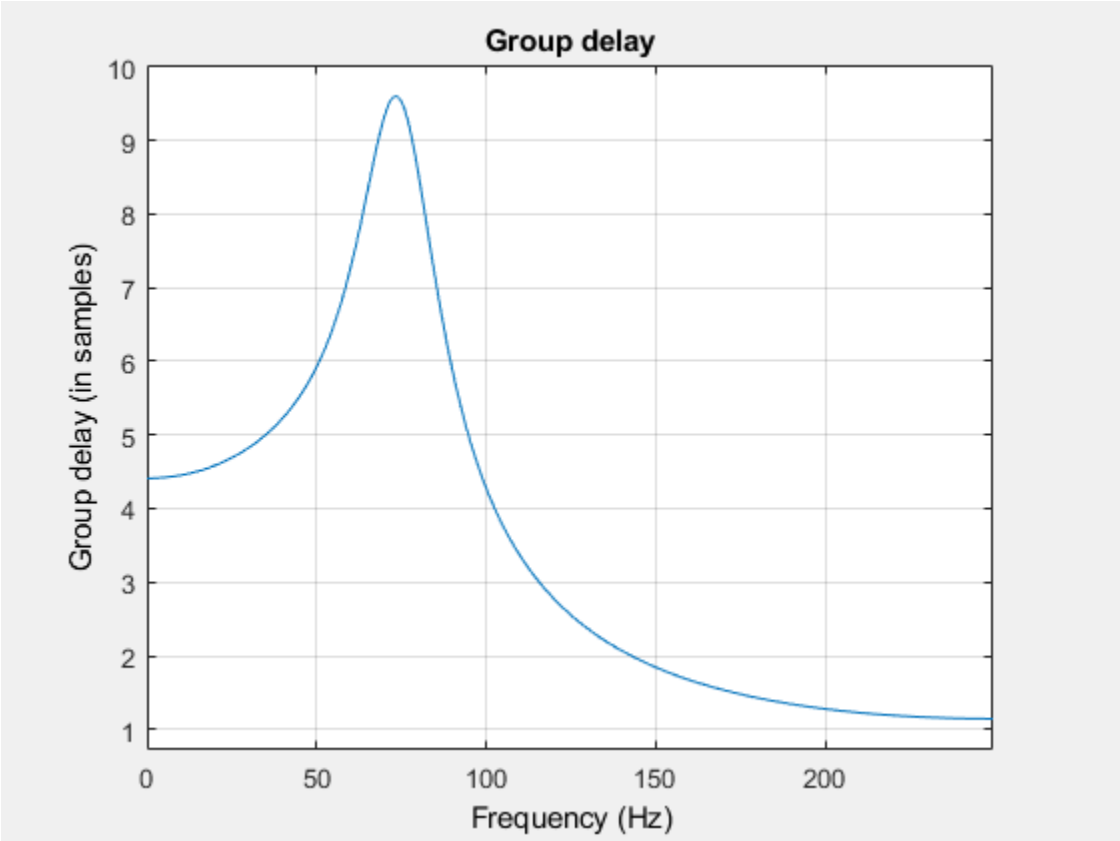
Filter the signal. The filtered signal is cleaner than the original, but lags in time with respect to it. It is also distorted due to the nonlinear phase of the filter.

```
xfilter = filter(iir,xn);

plot(tn,xn,tn,xfilter)

title 'Electrocardiogram'
xlabel 'Time (s)'
legend('Original','Filtered')
axis([0.25 0.55 -1 1.5])
grid
```



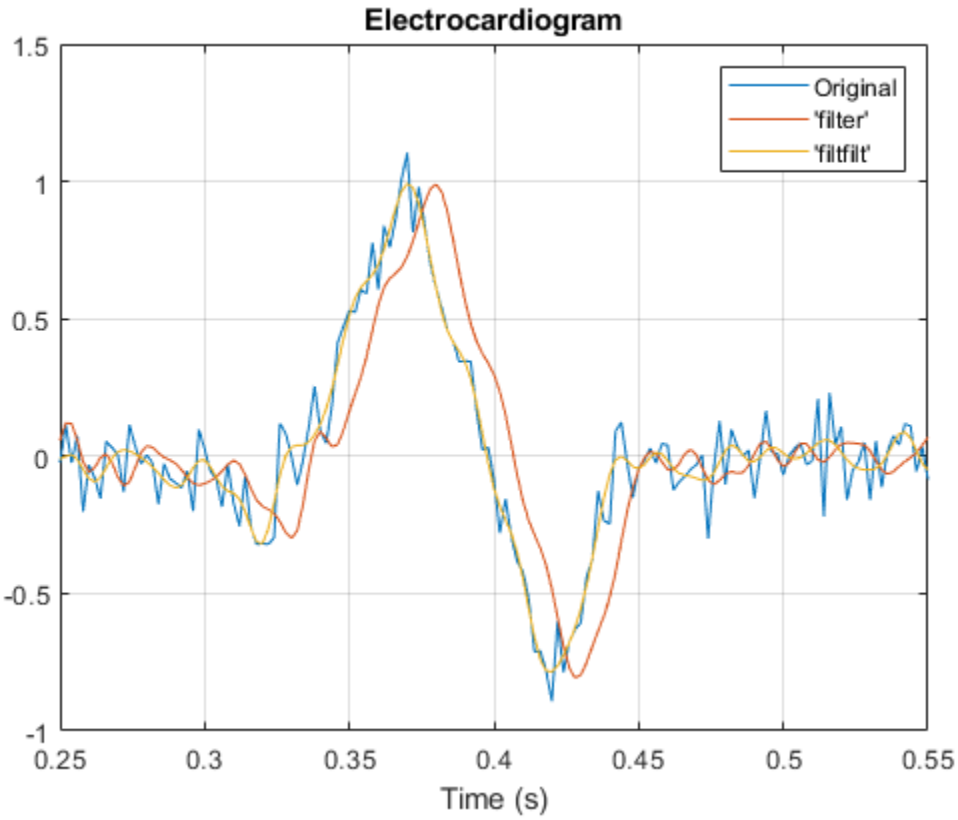

A look at the group delay introduced by the filter shows that the delay is frequency-dependent.

```
grpdelay(iir,N,Fs)
```




Filter the signal using `filtfilt`. The delay and distortion have been effectively removed. Use `filtfilt` when it is critical to keep the phase information of a signal intact.

```
xfiltfilt = filtfilt(iir,xn);

plot(tn,xn)
hold on
plot(tn,xfilter)
plot(tn,xfiltfilt)

title 'Electrocardiogram'
xlabel 'Time (s)'
legend('Original','''filter''','''filtfilt''')
axis([0.25 0.55 -1 1.5])
grid
```


Electrocardiogram
Original
'filter'
'filtfilt'
1.5
1
0.5
0
-0.5
-1
0.25
0.3
0.35
0.4
0.45
0.5
0.55
Time (s)

# Comparison of Analog IIR Lowpass Filters

Design a 5th-order analog Butterworth lowpass filter with a cutoff frequency of 2 GHz. Multiply by $2\pi$ to convert the frequency to radians per second. Compute the frequency response of the filter at 4096 points.

```
n = 5;
f = 2e9;

[zb,pb,kb] = butter(n,2*pi*f,'s');
[bb,ab] = zp2tf(zb,pb,kb);
[hb,wb] = freqs(bb,ab,4096);
```

Design a 5th-order Chebyshev Type I filter with the same edge frequency and 3 dB of passband ripple. Compute its frequency response.

```
[z1,p1,k1] = cheby1(n,3,2*pi*f,'s');
[b1,a1] = zp2tf(z1,p1,k1);
[h1,w1] = freqs(b1,a1,4096);
```

Design a 5th-order Chebyshev Type II filter with the same edge frequency and 30 dB of stopband attenuation. Compute its frequency response.

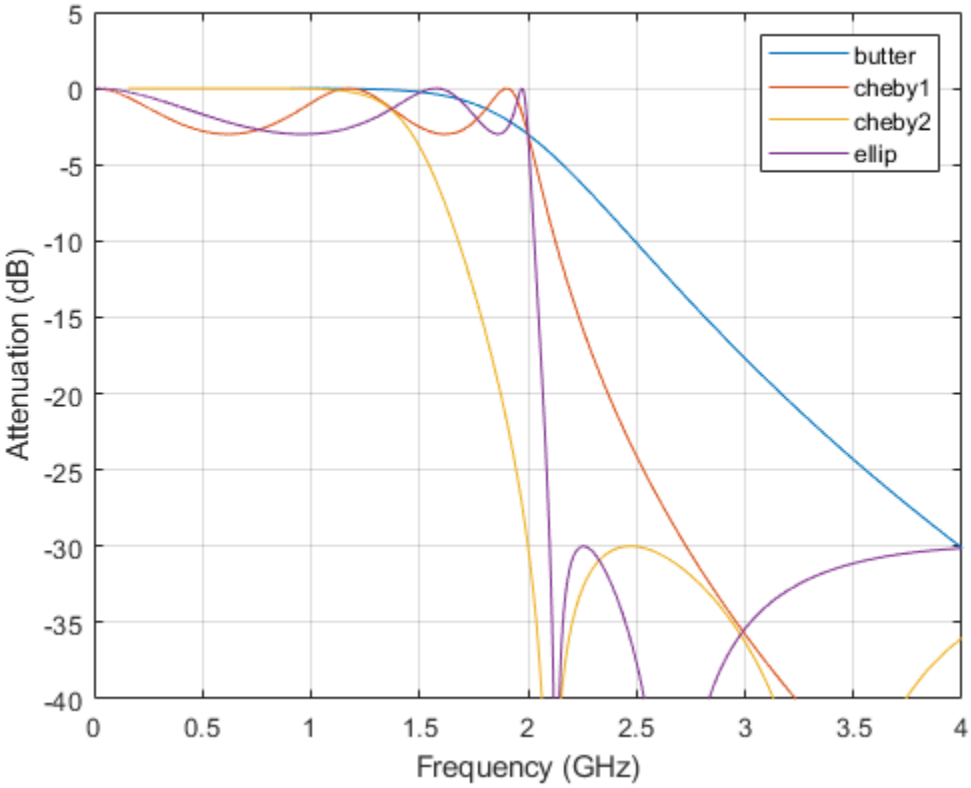
```
[z2,p2,k2] = cheby2(n,30,2*pi*f,'s');
[b2,a2] = zp2tf(z2,p2,k2);
[h2,w2] = freqs(b2,a2,4096);
```

Design a 5th-order elliptic filter with the same edge frequency, 3 dB of passband ripple, and 30 dB of stopband attenuation. Compute its frequency response.

```
[ze,pe,ke] = ellip(n,3,30,2*pi*f,'s');
[be,ae] = zp2tf(ze,pe,ke);
[he,we] = freqs(be,ae,4096);
```

Plot the attenuation in decibels. Express the frequency in gigahertz. Compare the filters.

```
plot(wb/(2e9*pi),mag2db(abs(hb)))
hold on
plot(w1/(2e9*pi),mag2db(abs(h1)))
plot(w2/(2e9*pi),mag2db(abs(h2)))
plot(we/(2e9*pi),mag2db(abs(he)))
axis([0 4 -40 5])
grid
xlabel('Frequency (GHz)')
ylabel('Attenuation (dB)')
legend('butter','cheby1','cheby2','ellip')
```




The Butterworth and Chebyshev Type II filters have flat passbands and wide transition bands. The Chebyshev Type I and elliptic filters roll off faster but have passband ripple. The frequency input to the Chebyshev Type II design function sets the beginning of the stopband rather than the end of the passband.

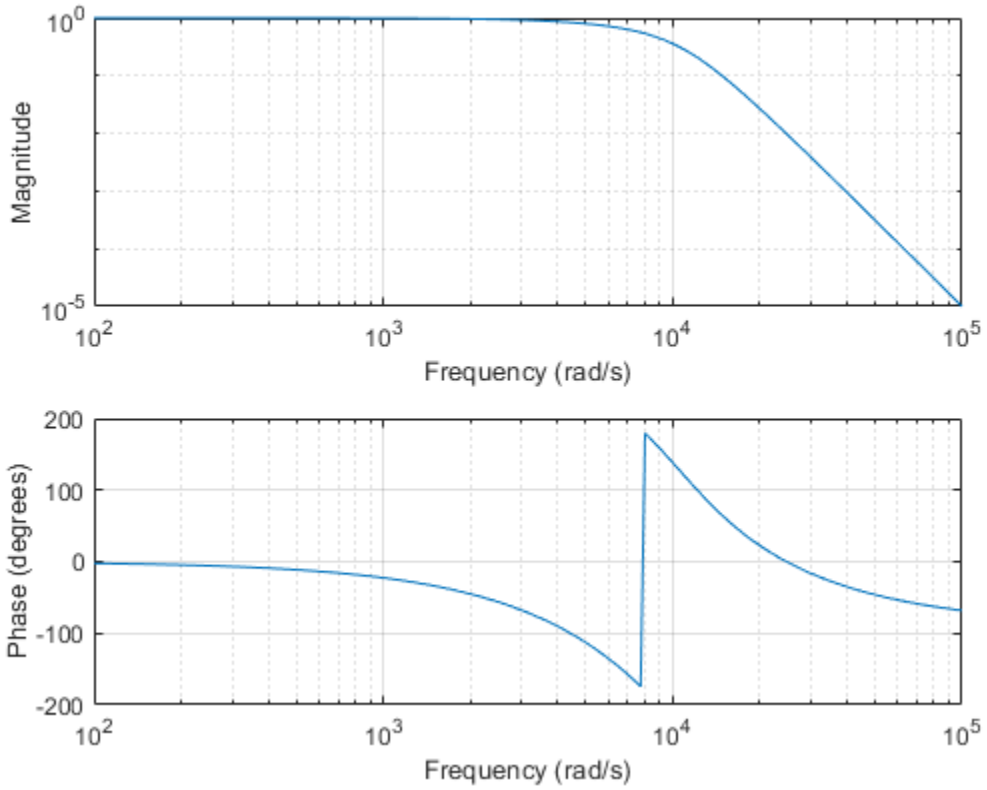
## See Also

`butter` | `cheby1` | `cheby2` | `ellip` | `freqs` | `zp2tf`

# Frequency Response of Lowpass Bessel Filter

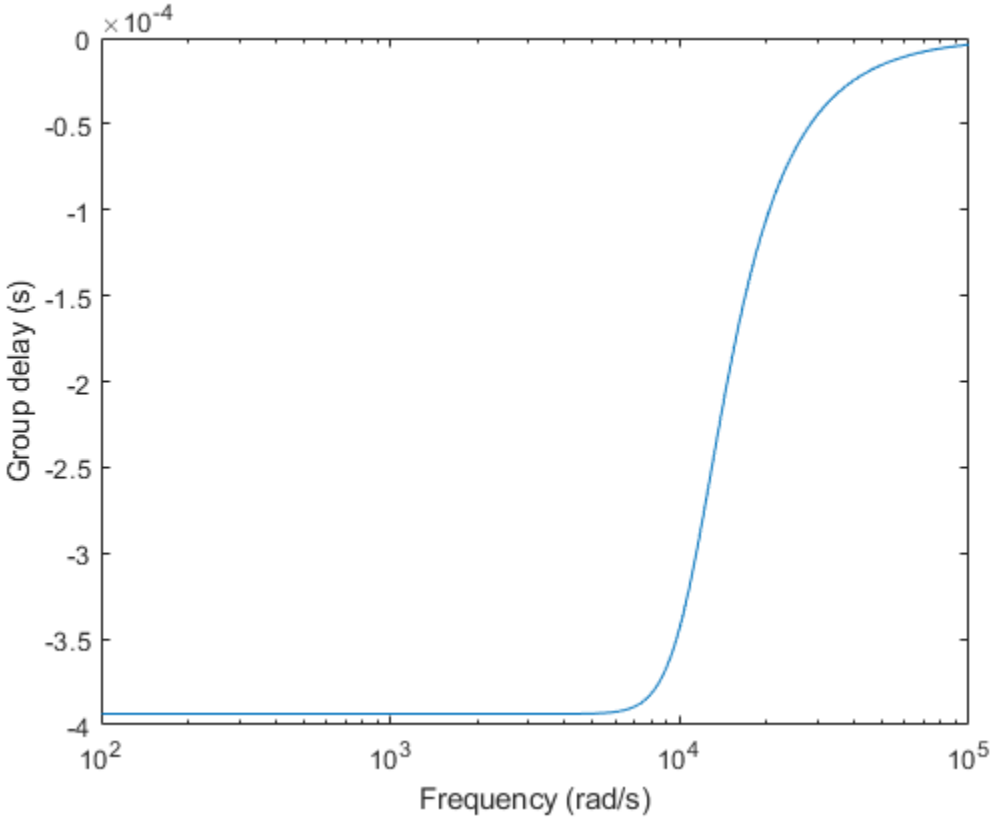
Design a fifth-order analog lowpass Bessel filter with approximately constant group delay up to $10^4$ rad/second. Plot the magnitude and phase responses of the filter using `freqs`.

```
[b,a] = besself(5,10000);
freqs(b,a)
```




Compute the group delay response of the filter as the derivative of the unwrapped phase response. Plot the group delay to verify that it is approximately constant up to the cutoff frequency.

```
[h,w] = freqs(b,a,1000);
grpdel = diff(unwrap(angle(h)))./diff(w);

clf
semilogx(w(2:end),grpdel)
xlabel('Frequency (rad/s)')
ylabel('Group delay (s)')
```




## See Also

`besself` | `freqs`

# Speaker Crossover Filters

This example shows how to devise a simple model of a digital three-way loudspeaker. The system splits the audio input into low-, mid-, and high-frequency bands that correspond respectively to the woofer, the midrange driver, and the tweeter. Typical values for the normalized crossover frequencies that delimit the bands are $0.136\pi$ rad/sample and $0.317\pi$ rad/sample.

Create lowpass, bandpass, and highpass filters to generate the low-frequency, mid-frequency, and high-frequency bands. Specify the frequencies.

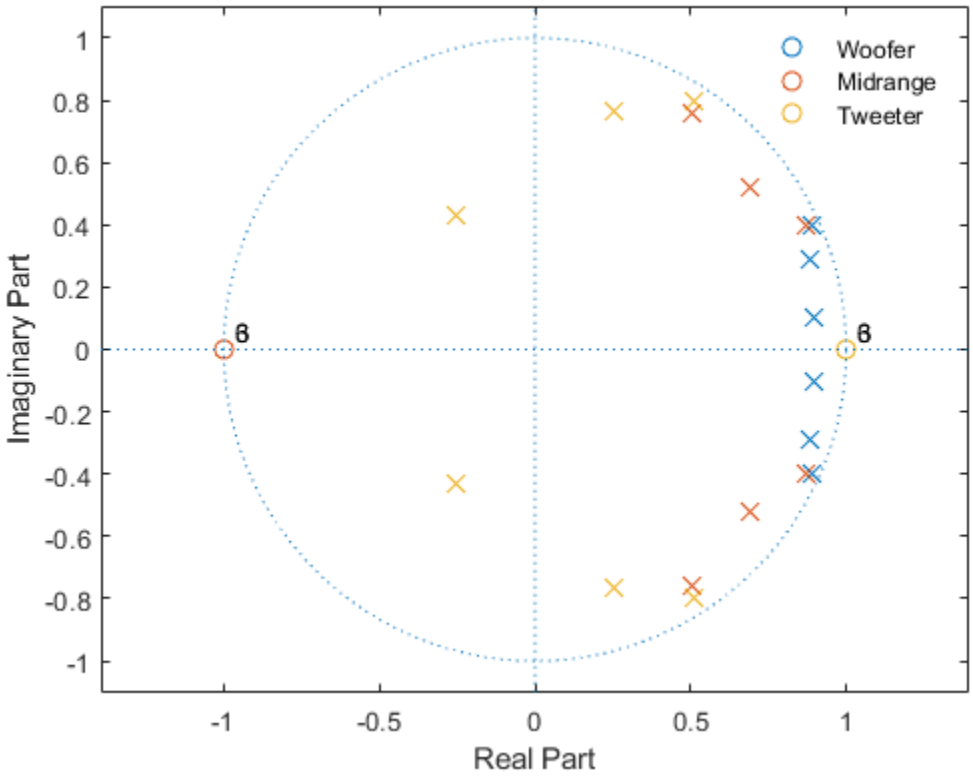
```
lo = 0.136;
hi = 0.317;
```

Use a 6th-order Chebyshev Type I design for each filter. Specify a passband ripple of 1 dB, larger than the value for real speakers. The `cheby1` function doubles the order of bandpass designs. Make all filters have the same order by halving the order of the bandpass filter. Return the zeros, poles, and gain of each filter.

```
ord = 6;
rip = 1;

[zw,pw,kw] = cheby1(ord,rip,lo);
[zm,pm,km] = cheby1(ord/2,rip,[lo hi]);
[zt,pt,kt] = cheby1(ord,rip,hi,'high');
```

Visualize the zeros and poles of the filters.

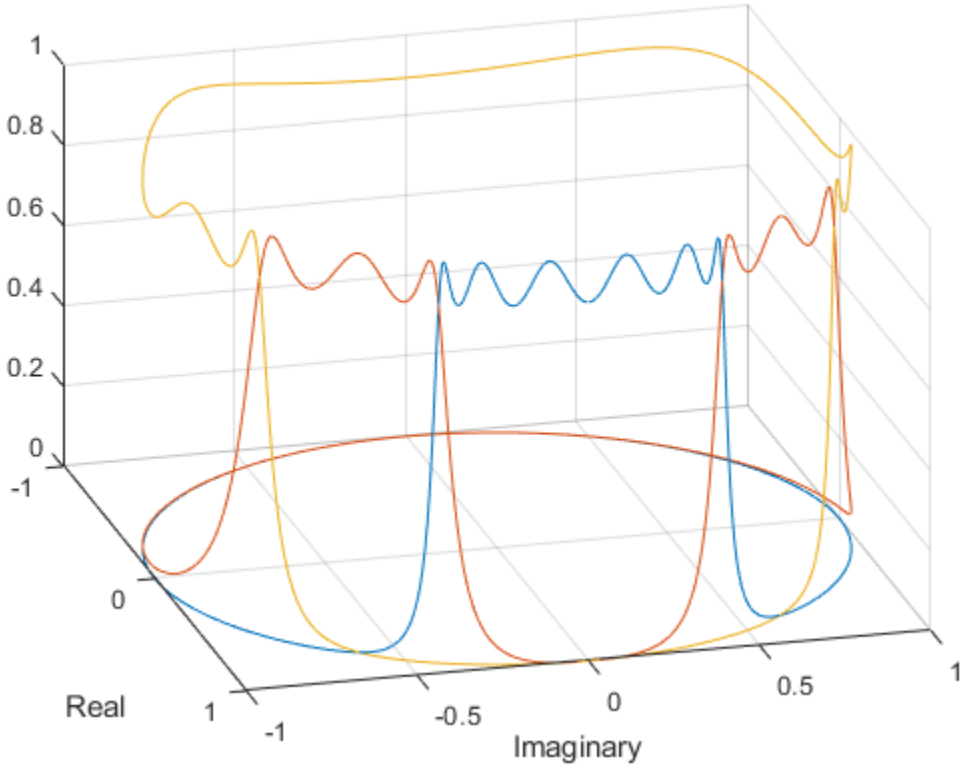
```
zplane([zw zm zt],[pw pm pt])
lg = legend('Woofer','Midrange','Tweeter');
lg.Box = 'off';
```




- *Woofer:* The zeros at $z = -1$ suppress high frequencies. The poles enhance the magnitude response between 0 and the lower crossover frequency.
- *Midrange:* The zeros at $z = 0$ and $z = 1$ suppress high and low frequencies. The poles enhance the magnitude response between the lower and higher crossover frequencies.
- *Tweeter:* The zeros at $z = 1$ suppress low frequencies. The poles enhance the magnitude response between the higher crossover frequency and $\pi$.

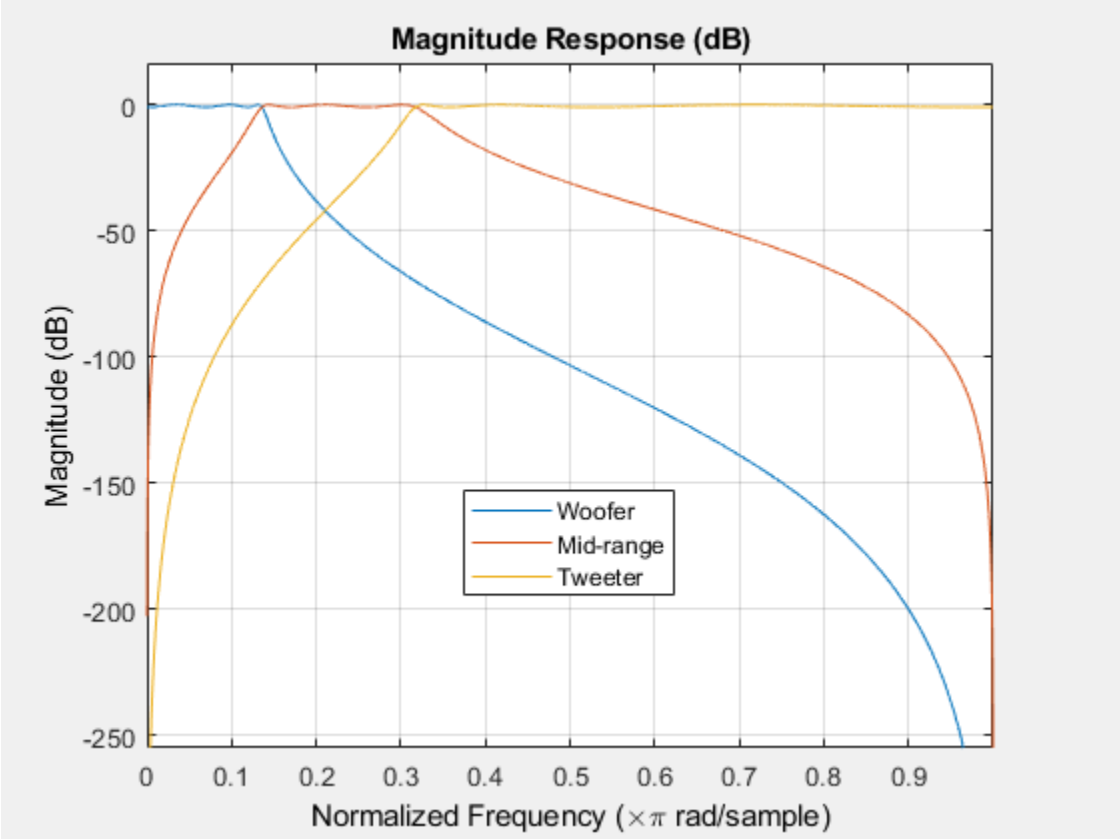
Plot the magnitude responses on the unit circle to see the effect of the different poles and zeros. Use linear units. Represent the filters as second-order sections.

```
sw = zp2sos(zw,pw,kw);
sm = zp2sos(zm,pm,km);
st = zp2sos(zt,pt,kt);

nf = 1024;
[hw,fw] = freqz(sw,nf,'whole');
hm = freqz(sm,nf,'whole');
ht = freqz(st,nf,'whole');

plot3(cos(fw),sin(fw),[abs(hw) abs(hm) abs(ht)])
xlabel('Real')
ylabel('Imaginary')
view(75,30)
grid
```



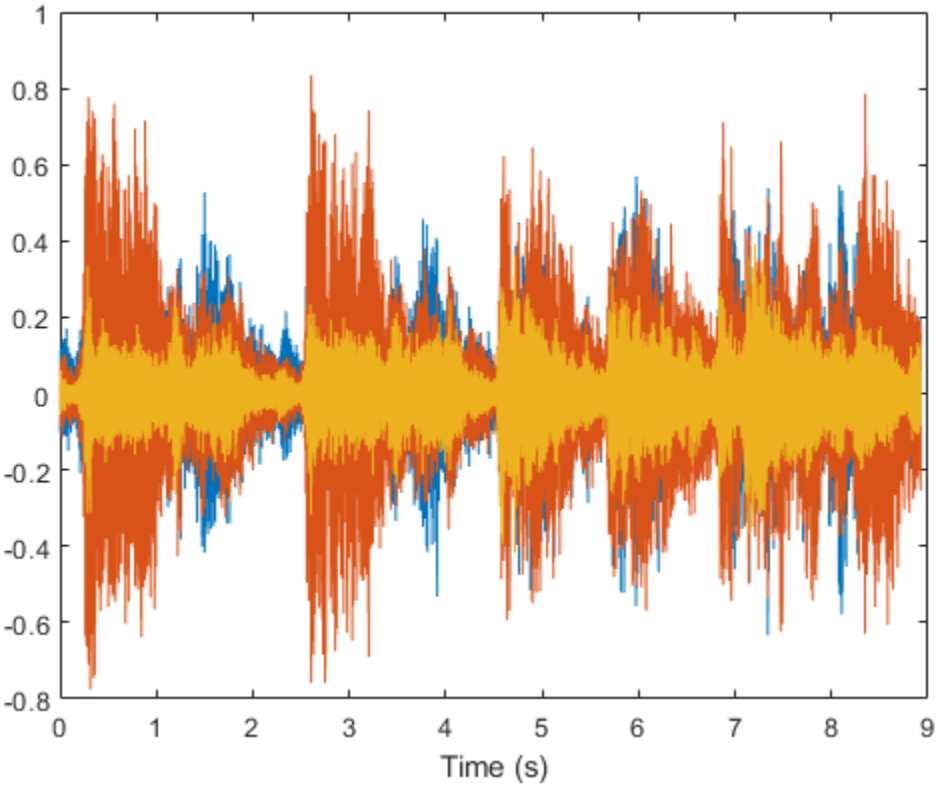

Plot the magnitude responses in dB using `fvtool`.

```
hfvt = fvtool(sw,sm,st);
legend(hfvt,'Woofer','Mid-range','Tweeter')
```




Load an audio file containing a fragment of Handel's "Hallelujah Chorus" sampled at 8192 Hz. Split the signal into three frequency bands by filtering. Plot the bands.

```
load handel                 % To hear, type soundsc(y,Fs)

yw = sosfilt(sw,y);         % To hear, type soundsc(yw,Fs)
ym = sosfilt(sm,y);         % To hear, type soundsc(ym,Fs)
yt = sosfilt(st,y);         % To hear, type soundsc(yt,Fs)

plot((0:length(y)-1)/Fs,[yw ym yt])
xlabel('Time (s)')
```




```
% To hear all the frequency ranges, type soundsc(yw+ym+yt,Fs)
```

**References**

Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1996.

## See Also

**FVTool** | `cheby1` | `freqz` | `sosfilt` | `zp2sos` | `zplane`

# Filter Designer: A Filter Design and Analysis App


- “Filter Design Methods” on page 5-2
- “Using the Filter Designer App” on page 5-4
- “Analyzing Filter Responses” on page 5-5
- “Filter Designer App Panels” on page 5-6
- “Getting Help” on page 5-7
- “Getting Started with Filter Designer” on page 5-8
- “Importing a Filter Design” on page 5-21
- “FIR Bandpass Filter with Asymmetric Attenuation” on page 5-24
- “Arbitrary Magnitude Filter” on page 5-26

# Filter Design Methods

The **Filter Designer** app is a user interface for designing and analyzing filters quickly. The app enables you to design digital FIR or IIR filters by setting filter specifications, by importing filters from your MATLAB workspace, or by adding, moving or deleting poles and zeros. It also provides tools for analyzing filters, such as magnitude and phase response and pole-zero plots.

The **Filter Designer** app gives you access to the following Signal Processing Toolbox filter design methods.

| Design Method | Function |
|---|---|
| Butterworth | `butter` |
| Chebyshev Type I | `cheby1` |
| Chebyshev Type II | `cheby2` |
| Elliptic | `ellip` |
| Maximally Flat | `maxflat` |
| Equiripple | `firpm` |
| Least-squares | `firls` |
| Constrained least-squares | `fircls` |
| Complex equiripple | `cfirpm` |
| Window | `fir1` |

When using the `window` method, all Signal Processing Toolbox window functions are available, and you can specify a user-defined window by entering its function name and input parameter.

## Advanced Filter Design Methods

The following advanced filter design methods are available if you have DSP System Toolbox software.

| Design Method | Function |
|---|---|
| Constrained equiripple FIR | `firceqrip` |
| Constrained-band equiripple FIR | `fircband` |
| Generalized remez FIR | `firgr` |
| Equiripple halfband FIR | `firhalfband` |
| Least P-norm optimal FIR | `firlpnorm` |
| Equiripple Nyquist FIR | `firnyquist` |
| Interpolated FIR | `ifir` |
| IIR comb notching or peaking | `iircomb` |
| Allpass filter (given group delay) | `iirgrpdelay` |
| Least P-norm optimal IIR | `iirlpnorm` |
| Constrained least P-norm IIR | `iirlpnormc` |
| Second-order IIR notch | `iirnotch` |

| Design Method | Function |
| --- | --- |
| Second-order IIR peaking (resonator) | `iirpeak` |

# Using the Filter Designer App

There are different ways that you can design filters using the **Filter Designer** app. For example:

- You can first choose a response type, such as bandpass, and then choose from the available FIR or IIR filter design methods.
- You can specify the filter by its type alone, along with certain frequency- or time-domain specifications such as passband frequencies and stopband frequencies. The filter you design is then computed using the default filter design method and filter order.

# Analyzing Filter Responses

Once you have designed your filter, you can display the filter coefficients and detailed filter information, export the coefficients to the MATLAB workspace, or create a C header file containing the coefficients.

You also can analyze different filter responses in the app or in a separate Filter Visualization Tool (**FVTool**). The following filter responses are available:

- Magnitude response (`freqz`)
- Phase response (`phasez`)
- Group delay (`grpdelay`)
- Phase delay (`phasedelay`)
- Impulse response (`impz`)
- Step response (`stepz`)
- Pole-zero plots (`zplane`)
- Zero-phase response (`zerophase`)

# Filter Designer App Panels

The **Filter Designer** app has sidebar buttons that display particular panels in the lower half. The panels are:

- Design Filter. See “Choosing a Filter Design Method” on page 5-9 for more information. You use this panel to
  - Design filters from scratch.
  - Modify existing filters designed with the app.
  - Analyze filters.
- Import filter. You use this panel to
  - Import previously saved filters or filter coefficients that you have stored in the MATLAB workspace.
  - Analyze imported filters.
- Pole-Zero Editor. See “Editing the Filter Using the Pole-Zero Editor” on page 5-13. You use this panel to add, delete, and move poles and zeros in your filter design.

If you also have DSP System Toolbox product installed, additional panels are available:

- Set quantization parameters — Use this panel to quantize double-precision filters that you design with **Filter Designer**, quantize double-precision filters that you import into the app, and analyze quantized filters.
- Transform filter — Use this panel to change a filter from one response type to another.
- Multirate filter design — Use this panel to create a multirate filter from your existing FIR design, create CIC filters, and linear and hold interpolators.

If you have Simulink installed, this panel is available:

- Realize Model — Use this panel to create a Simulink block containing the filter structure.

# Getting Help

At any time, you can right-click or click the **What's this?** button, , to get information. You can also use the **Help** menu to see complete Help information.

# Getting Started with Filter Designer

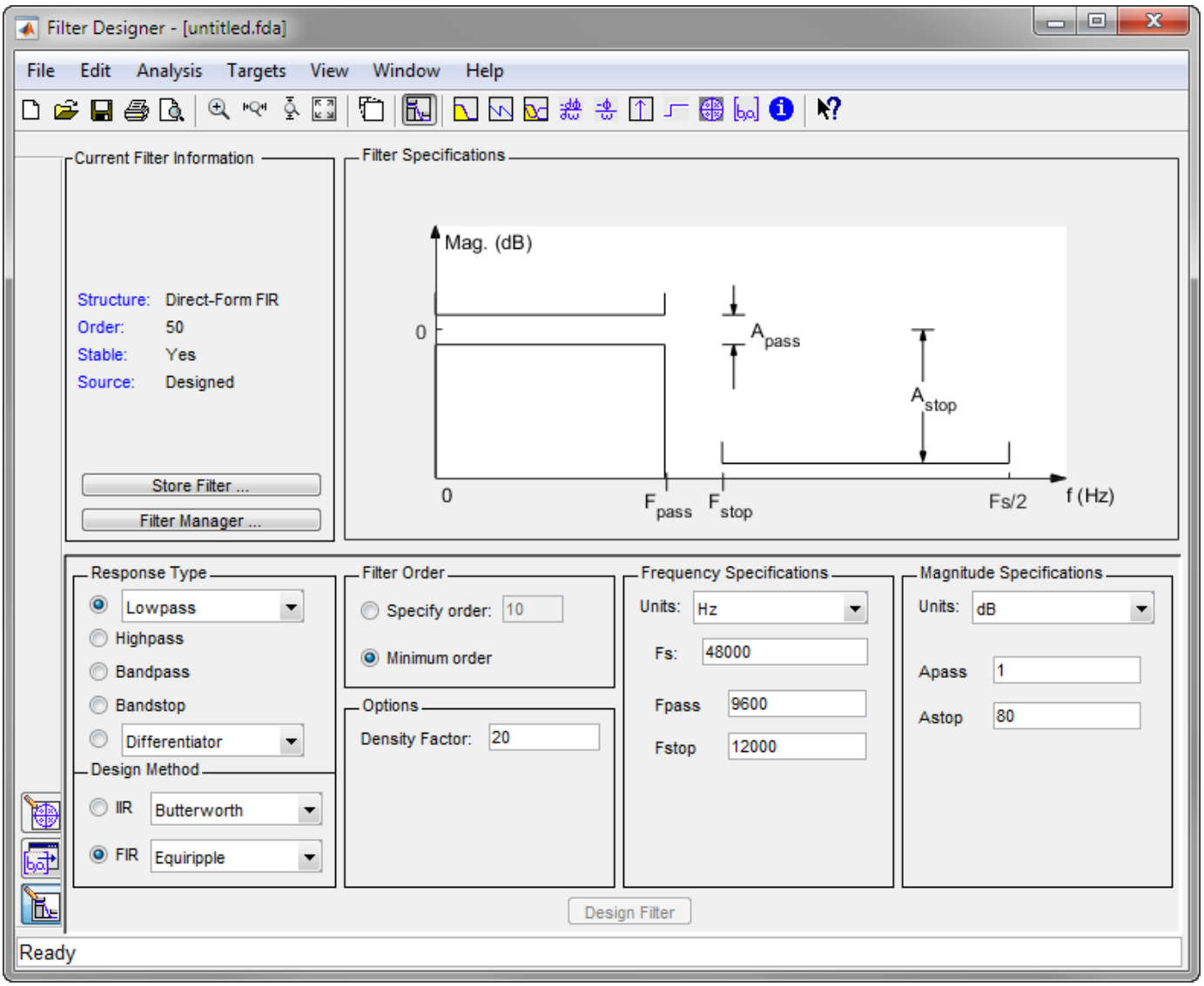
The **Filter Designer** app enables you to design and analyze digital filters. You can also import and modify existing filter designs.

To open the **Filter Designer** app, type

```
filterDesigner
```

at the MATLAB command prompt.

The **Filter Designer** app opens with the Design Filter panel displayed.




Note that when you open **Filter Designer**, **Design Filter** is not enabled. You must make a change to the default filter design in order to enable **Design Filter**. This is true each time you want to change the filter design. Changes to radio button items or drop down menu items such as those under

**Response Type** or **Filter Order** enable **Design Filter** immediately. Changes to specifications in text boxes such as **Fs**, **Fpass**, and **Fstop** require you to click outside the text box to enable **Design Filter**.

## Choosing a Response Type

You can choose from several response types:

- Lowpass
- Raised cosine
- Highpass
- Bandpass
- Bandstop
- Differentiator
- Multiband
- Hilbert transformer
- Arbitrary magnitude

Additional response types are available if you have DSP System Toolbox software installed.

---

**Note** Not all filter design methods are available for all response types. Once you choose your response type, this may restrict the filter design methods available to you. Filter design methods that are not available for a selected response type are removed from the Design Method region of the app.

---

## Choosing a Filter Design Method

You can use the default filter design method for the response type that you've selected, or you can select a filter design method from the available FIR and IIR methods listed in the app.

To select the Remez algorithm to compute FIR filter coefficients, select the **FIR** radio button and choose `Equiripple` from the list of methods.

## Setting the Filter Design Specifications

### Viewing Filter Specifications

The filter design specifications that you can set vary according to response type and design method. The display region illustrates filter specifications when you select **Analysis > Filter Specifications** or when you click the **Filter Specifications** toolbar button.

You can also view the filter specifications on the Magnitude plot of a designed filter by selecting **View > Specification Mask**.

### Filter Order

You have two mutually exclusive options for determining the filter order when you design an equiripple filter:

- **Specify order**: You enter the filter order in a text box.

- **Minimum order**: The filter design method determines the minimum order filter.

Note that filter order specification options depend on the filter design method you choose. Some filter methods may not have both options available.

#### Options

The available options depend on the selected filter design method. Only the FIR Equiripple and FIR Window design methods have settable options. For FIR Equiripple, the option is a **Density Factor**. See `firpm` for more information. For FIR Window the options are **Scale Passband**, **Window** selection, and for the following windows, a settable parameter:

| Window | Parameter |
|---|---|
| Chebyshev (`chebwin`) | Sidelobe attenuation |
| Gaussian (`gausswin`) | Alpha |
| Kaiser (`kaiser`) | Beta |
| Taylor (`taylorwin`) | Nbar and Sidelobe level |
| Tukey (`tukeywin`) | Alpha |
| User Defined | Function Name, Parameter |

You can view the window in the Window Visualization Tool (**WVTool**) by clicking the **View** button.

#### Bandpass Filter Frequency Specifications

For a bandpass filter, you can set

- Units of frequency:
  - Hz
  - kHz
  - MHz
  - Normalized (0 to 1)
- Sample rate
- Passband frequencies
- Stopband frequencies

You specify the passband with two frequencies. The first frequency determines the lower edge of the passband, and the second frequency determines the upper edge of the passband.

Similarly, you specify the stopband with two frequencies. The first frequency determines the upper edge of the first stopband, and the second frequency determines the lower edge of the second stopband.

#### Bandpass Filter Magnitude Specifications

For a bandpass filter, you can specify the following magnitude response characteristics:

- Units for the magnitude response (dB or linear)
- Passband ripple

- Stopband attenuation

## Computing the Filter Coefficients

Now that you've specified the filter design, click the **Design Filter** button to compute the filter coefficients.

**Note** The **Design Filter** button is disabled once you've computed the coefficients for your filter design. This button is enabled again once you make any changes to the filter specifications.

## Analyzing the Filter

### Displaying Filter Responses

You can view the following filter response characteristics in the display region or in a separate window.

- Magnitude response
- Phase response
- Magnitude and Phase responses
- Group delay response
- Phase delay response
- Impulse response
- Step response
- Pole-zero plot
- Zero-phase response — available from the *y*-axis context menu in a Magnitude or Magnitude and Phase response plot.

**Note** If you have DSP System Toolbox product installed, two other analyses are available: magnitude response estimate and round-off noise power. These two analyses are the only ones that use filter internals.

For descriptions of the above responses and their associated toolbar buttons and other **Filter Designer** toolbar buttons, see **FVTool**.

You can display two responses in the same plot by selecting **Analysis > Overlay Analysis** and selecting an available response. A second *y*-axis is added to the right side of the response plot. (Note that not all responses can be overlaid on each other.)

You can also display the filter coefficients and detailed filter information in this region.

For all the analysis methods, except zero-phase response, you can access them from the **Analysis** menu, the Analysis Parameters dialog box from the context menu, or by using the toolbar buttons. For zero-phase, right-click the *y*-axis of the plot and select **Zero-phase** from the context menu.

You can overlay the filter specifications on the Magnitude plot by selecting **View > Specification Mask**.

### Using Data Tips

You can click the response to add plot data tips that display information about particular points on the response.

For information on using data tips, see “Interactively Explore Plotted Data”.

### Drawing Spectral Masks

To add spectral masks or rejection area lines to your magnitude plot, click **View > User-defined Spectral Mask**.

The mask is defined by a frequency vector and a magnitude vector. These vectors must be the same length.

- **Enable Mask** — Select to turn on the mask display.
- **Normalized Frequency** — Select to normalize the frequency between 0 and 1 across the displayed frequency range.
- **Frequency Vector** — Enter a vector of *x*-axis frequency values.
- **Magnitude Units** — Select the desired magnitude units. These units should match the units used in the magnitude plot.
- **Magnitude Vector** — Enter a vector of *y*-axis magnitude values.

### Changing the Sample Rate

To change the sample rate of your filter, right-click any filter response plot and select **Sampling Frequency** from the context menu.

To change the filter name, type the new name in **Filter name**. (In **FVTool**, if you have multiple filters, select the desired filter and then enter the new name.)

To change the sample rate, select the desired unit from **Units** and enter the sample rate in **Fs**. (For each filter in **FVTool**, you can specify a different sample rate or you can apply the sample rate to all filters.)

To save the displayed parameters as the default values to use when **Filter Designer** or **FVTool** is opened, click **Save as Default**.

To restore the default values, click **Restore Original Defaults**.

### Displaying the Response in FVTool

To display the filter response characteristics in a separate window, select **View > Filter Visualization Tool** (available if any analysis, except the filter specifications, is in the display region) or click the **Full View Analysis** button. This starts the Filter Visualization Tool (**FVTool**).

---

**Note** If Filter Specifications are shown in the display region, clicking the **Full View Analysis** toolbar button launches a MATLAB figure window instead of **FVTool**. The associated menu item is **Print to Figure**, which is enabled only if the filter specifications are displayed.

---

You can use this tool to annotate your design, view other filter characteristics, and print your filter response. You can link **Filter Designer** and **FVTool** so that changes made in **Filter Designer** are immediately reflected in **FVTool**. See **FVTool** for more information.

## Editing the Filter Using the Pole-Zero Editor

### Displaying the Pole-Zero Plot

You can edit a designed or imported filter's coefficients by moving, deleting, or adding poles or zeros or both using the Pole-Zero Editor panel.

---

**Note** You cannot generate MATLAB code (**File > Generate MATLAB code**) if your filter was designed or edited with the Pole-Zero Editor.

You cannot move quantized poles and zeros. You can only move the reference poles and zeros.

---

Click the **Pole-Zero Editor** button in the sidebar or select **Edit > Pole-Zero Editor** to display the Pole-Zero Editor panel.

Poles are shown using "`x`" symbols and zeros are shown using "`o`" symbols.

### Changing the Pole-Zero Plot

Plot mode buttons are located to the left of the pole-zero plot. Select one of the buttons to change the mode of the pole-zero plot. The Pole-Zero Editor has these buttons from left to right: **Move Pole-Zero**, **Add Pole**, **Add Zero**, and **Delete Pole-Zero**.

---

**Note** For filters with orders larger than approximately 100, the Pole-Zero Editor might encounter numerical problems when computing transfer function polynomials. As a result, the displayed filter responses might be different than expected. To inspect poles and zeros without attempting to compute high-order polynomials, select **Analysis > Pole-Zero Plot**. You cannot edit a filter in this view.

---

The following plot parameters and controls are located to the left of the pole-zero plot and below the plot mode buttons.

- **Filter gain** — factor to compensate for the filter's pole(s) and zero(s) gains
- **Coordinates** — units (`Polar` or `Rectangular`) of the selected pole or zero
- **Magnitude** — if polar coordinates is selected, magnitude of the selected pole or zero
- **Angle** — if polar coordinates is selected, angle of selected pole(s) or zero(s)
- **Real** — if rectangular coordinates is selected, real component of selected pole(s) or zero(s)
- **Imaginary** — if rectangular coordinates is selected, imaginary component of selected pole or zero
- **Section** — for multisection filters, number of the current section
- **Conjugate** — creates a corresponding conjugate pole or zero or automatically selects the conjugate pole or zero if it already exists.
- **Auto update** — immediately updates the displayed magnitude response when poles or zeros are added, moved, or deleted.

The **Edit > Pole-Zero Editor** has items for selecting multiple poles or zeros, for inverting and mirroring poles or zeros, and for deleting, scaling and rotating poles or zeros.

- When you select a pole or zero from a conjugate pair, the **Conjugate** check box and the conjugate are automatically selected.

## Converting the Filter Structure

### Converting to a New Structure

You can use **Edit > Convert Structure** to convert the current filter to a new structure. All filters can be converted to the following representations:

- Direct-form I
- Direct-form II
- Direct-form I transposed
- Direct-form II transposed
- Lattice ARMA

**Note** If you have DSP System Toolbox product installed, you will see additional structures in the Convert structure dialog box.

In addition, the following conversions are available for particular classes of filters:

- Minimum phase FIR filters can be converted to Lattice minimum phase
- Maximum phase FIR filters can be converted to Lattice maximum phase
- Allpass filters can be converted to Lattice allpass
- IIR filters can be converted to Lattice ARMA

**Note** Converting from one filter structure to another might produce a result with different characteristics than the original. This is due to the computer's finite-precision arithmetic and the variations in the conversion's round-off computations.

For example:

- Select **Edit > Convert Structure** to open the Convert structure dialog box.
- Select `Direct-form I` in the list of filter structures.

### Converting to Second-Order Sections

You can use **Edit > Convert to Second-Order Sections** to store the converted filter structure as a collection of second-order sections rather than as a monolithic higher-order structure.

**Note** The following options are also used for **Edit > Reorder and Scale Second-Order Sections**, which you use to modify an SOS filter structure.

The following **Scale** options are available when converting a direct-form II structure only:

- `None` (default)
- `L-2` ($L^2$ norm)
- `L-infinity` ($L^\infty$ norm)

The **Direction** (`Up` or `Down`) determines the ordering of the second-order sections. The optimal ordering changes depending on the **Scale** option selected.

For example:

- Select **Edit > Convert to Second-Order Sections** to open the Convert to SOS dialog box.
- Select `L-infinity` from the **Scale** menu for $L^\infty$ norm scaling.
- Leave `Up` as the **Direction** option.

**Note** To convert from second-order sections back to a single section, use **Edit > Convert to Single Section**.

## Exporting a Filter Design

### Exporting Coefficients or Objects to the Workspace

You can save the filter either as filter coefficients variables or as a filter object variable. To save the filter to the MATLAB workspace:

1. Select **File > Export**. The Export dialog box appears.
2. Select `Workspace` from the **Export To** menu.
3. Select `Coefficients` from the **Export As** menu to save the filter coefficients or select `Objects` to save the filter in a filter object.
4. For coefficients, assign variable names using the **Numerator** (for FIR filters) or **Numerator** and **Denominator** (for IIR filters), or **SOS Matrix** and **Scale Values** (for IIR filters in second-order section form) text boxes in the Variable Names region.

   For objects, assign the variable name in the **Discrete Filter** text box. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** check box.
5. Click the **Export** button.

### Exporting Coefficients to an ASCII File

To save filter coefficients to a text file,

1. Select **File > Export**. The Export dialog box appears.
2. Select `Coefficients File (ASCII)` from the **Export To** menu.
3. Click the **Export** button. The Export Filter Coefficients to FCF File dialog box appears.
4. Choose or enter a file name and click the **Save** button.

The coefficients are saved in the text file that you specified, and the MATLAB Editor opens to display the file. The text file also contains comments with the MATLAB version number, the Signal Processing Toolbox version number, and filter information.

### Exporting Coefficients or Objects to a MAT-File

To save filter coefficients or a filter object as variables in a MAT-file:

1. Select **File** > **Export**. The Export dialog box appears.
2. Select `MAT-file` from the **Export To** menu.
3. Select `Coefficients` from the **Export As** menu to save the filter coefficients or select `Objects` to save the filter in a filter object.
4. For coefficients, assign variable names using the **Numerator** (for FIR filters) or **Numerator** and **Denominator** (for IIR filters), or **SOS Matrix** and **Scale Values** (for IIR filters in second-order section form) text boxes in the Variable Names region.

   For objects, assign the variable name in the **Discrete Filter (**or **Quantized Filter)** text box. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** check box.
5. Click the **Export** button. The Export to a MAT-File dialog box appears.
6. Choose or enter a file name and click the **Save** button.

### Exporting to a Simulink Model

If you have the Simulink product installed, you can export a Simulink block of your filter design and insert it into a new or existing Simulink model.

You can export a filter designed using any filter design method available in **Filter Designer**.

---

**Note** If you have the DSP System Toolbox and Fixed-Point Designer™ installed, you can export a CIC filter to a Simulink model.

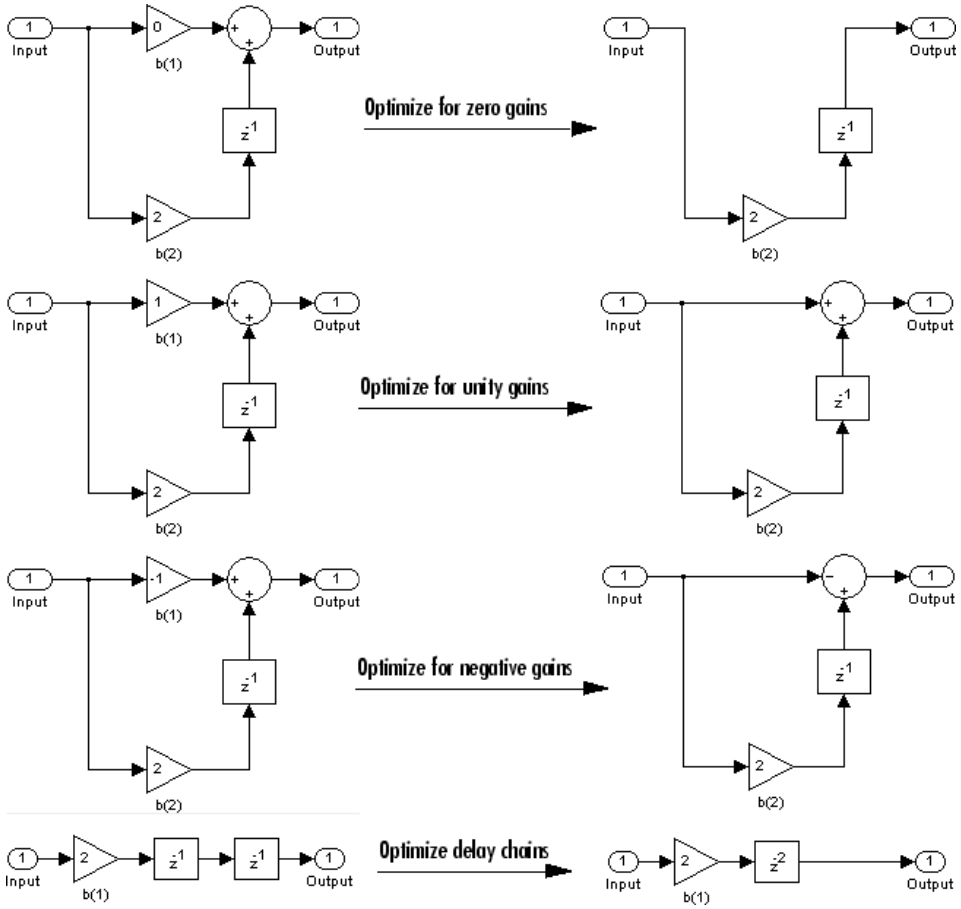
---

1. After designing your filter, click the **Realize Model** sidebar button or select **File > Export to Simulink Model**. The Realize Model panel is displayed.
2. Specify the name to use for your block in **Block name**.
3. To insert the block into the current (most recently selected) Simulink model, set the **Destination** to `Current`. To inset the block into a new model, select `New`. To insert the block into a user-defined subsystem, select `User defined`.
4. If you want to overwrite a block previously created from this panel, check **Overwrite generated `Filter' block**.
5. If you select the **Build model using basic elements** check box, your filter is created as a subsystem (Simulink) block, which uses separate sub-elements. In this mode, the following optimization(s) are available:

   - `Optimize for zero gains` — Removes zero-valued gain paths from the filter structure.
   - `Optimize for unity gains` — Substitutes a wire (short circuit) for gains equal to 1 in the filter structure.
   - `Optimize for negative gains` — Substitutes a wire (short circuit) for gains equal to -1 and changes corresponding additions to subtractions in the filter structure.
   - `Optimize delay chains` — Substitutes delay chains composed of `n` unit delays with a single delay of `n`.

- `Optimize for unity scale values` — Removes multiplications for scale values equal to 1 from the filter structure.

The following illustration shows the effects of some of the optimizations:




**Note** The **Build model using basic elements** check box is enabled only when you have a DSP System Toolbox license and your filter can be designed using a Biquad Filter block or a Discrete FIR Filter block. For more information, see the Filter Realization Wizard topic in the DSP System Toolbox documentation.

6 Set the **Input processing** parameter to specify whether the generated filter performs sample- or frame-based processing on the input. Depending on the type of filter you design, one or both of the following options may be available:

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

7 Click the **Realize Model** button to create the filter block. When the **Build model using basic elements** check box is selected, **Filter Designer** implements the filter as a subsystem block using Add, Gain, and Delay blocks.

If you double-click the Simulink Filter block, the filter structure is displayed.

## Generating a C Header File

You may want to include filter information in an external C program. To create a C header file with variables that contain filter parameter data, follow this procedure:

1. Select **Targets > Generate C Header**. The Generate C Header dialog box appears.
2. Enter the variable names to be used in the C header file. The particular filter structure determines the variables that are created in the file.

| Filter Structure | Variable Parameter |
|---|---|
| Direct-form I<br>Direct-form II<br>Direct-form I transposed<br>Direct-form II transposed | **Numerator**, **Numerator length**, **Denominator**, **Denominator length** |
| Lattice ARMA | **Lattice coeff.**, **Lattice coeff. length**, **Ladder coeff.**, **Ladder coeff. length** |
| Lattice MA | **Lattice coeff.**, **Lattice coeff. length**, and **Number of sections** (inactive if filter has only one section) |
| Direct-form FIR Direct-form FIR transposed | **Numerator**, **Numerator length**, and **Number of sections** (inactive if filter has only one section) |

   Length variables contain the total number of coefficients of that type.

   **Note** Variable names cannot be C language reserved words, such as `for`.

3. Select **Export Suggested** to use the suggested data type or select **Export As** and select the desired data type from the pull-down.

   **Note** If you do not have DSP System Toolbox software installed, selecting any data type other than double-precision floating point results in a filter that does not exactly match the one you designed in the **Filter Designer**. This is due to rounding and truncating differences.

4. Click **Generate** to save the file and leave the dialog box open for additional C header file definitions. To close the dialog box, click **Close**.

## Generating MATLAB Code

You can generate MATLAB code that constructs the filter you designed in **Filter Designer** from the command line. Select **File > Generate MATLAB Code > Filter Design Function** and specify the file name in the Generate MATLAB code dialog box.

**Note** You cannot generate MATLAB code (**File > Generate MATLAB Code > Filter Design Function**) if your filter was designed or edited with the Pole-Zero Editor.

The following is generated MATLAB code for the default lowpass filter in **Filter Designer**.

```
function Hd = ExFilter
%EXFILTER Returns a discrete-time filter object.

%
% MATLAB Code
% Generated by MATLAB(R) 7.11 and the Signal Processing Toolbox 6.14.
%
% Generated on: 17-Feb-2010 14:15:37
%

% Equiripple Lowpass filter designed using the FIRPM function.

% All frequency values are in Hz.
Fs = 48000;  % Sample Rate

Fpass = 9600;            % Passband Frequency
Fstop = 12000;           % Stopband Frequency
Dpass = 0.057501127785;  % Passband Ripple
Dstop = 0.0001;          % Stopband Attenuation
dens  = 20;              % Density Factor

% Calculate the order from the parameters using FIRPMORD.
[N, Fo, Ao, W] = firpmord([Fpass, Fstop]/(Fs/2), [1 0], [Dpass, Dstop]);

% Calculate the coefficients using the FIRPM function.
b  = firpm(N, Fo, Ao, W, {dens});
Hd = dfilt.dffir(b);

% [EOF]
```

## Managing Filters in the Current Session

You can store filters designed in the current **Filter Designer** session for cascading together, exporting to **FVTool** or for recalling later in the same or future **Filter Designer** sessions.

You store and access saved filters with the **Store Filter** and **Filter Manager** buttons, respectively, in the **Current Filter Information** pane.

**Store Filter** — Displays the Store Filter dialog box in which you specify the filter name to use when storing the filter in the Filter Manager. The default name is the type of the filter.

**Filter Manager** — Opens the Filter Manager.

The current filter is listed below the list box. To change the current filter, highlight the desired filter. If you select **Edit current filter**, **Filter Designer** displays the currently selected filter specifications. If you make any changes to the specifications, the stored filter is updated immediately.

To cascade two or more filters, highlight the desired filters and press **Cascade**. A new cascaded filter is added to the Filter Manager.

To change the name of a stored filter, press **Rename**. The Rename filter dialog box is displayed.

To remove a stored filter from the Filter Manager, press **Delete**.

To export one or more filters to **FVTool**, highlight the filter(s) and press **FVTool**.

## Saving and Opening Filter Design Sessions

You can save your filter design session as a MAT-file and return to the same session another time.

Select the **Save Session** button to save your session as a MAT-file. The first time you save a session, a Save Filter Design Session browser opens, prompting you for a session name.

The `.fda` extension is added automatically to all filter design sessions you save.

---

**Note** You can also use **File > Save Session** and **File > Save Session As** to save a session.

---

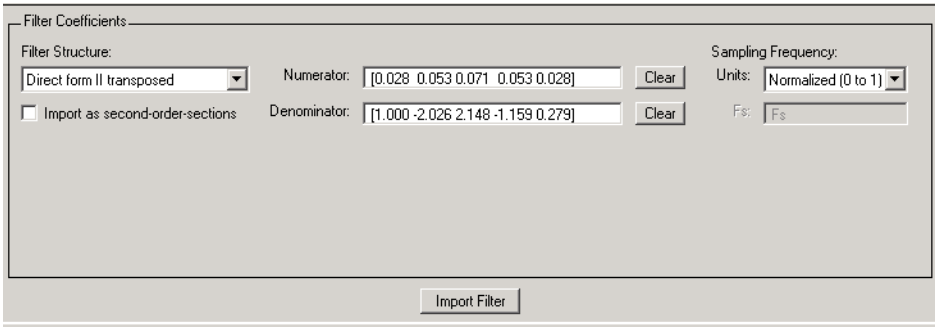
You can load existing sessions into **Filter Designer** by selecting the **Open Session** button or **File > Open Session** . A Load Filter Design Session browser opens that allows you to select from your previously saved filter design sessions.

# Importing a Filter Design

| In this section... |
| --- |
| “Import Filter Panel” on page 5-21 |
| “Filter Structures” on page 5-21 |

## Import Filter Panel

The Import Filter panel allows you to import a filter. You can access this region by clicking the **Import Filter** button in the sidebar.




The imported filter can be in any of the representations listed in the **Filter Structure** pull-down menu. You can import a filter as second-order sections by selecting the check box.

Specify the filter coefficients in **Numerator** and **Denominator**, either by entering them explicitly or by referring to variables in the MATLAB workspace.

Select the frequency units from the following options in the **Units** menu, and for any frequency unit other than Normalized, specify the value or MATLAB workspace variable of the sample rate in the **Fs** field.

To import the filter, click the **Import Filter** button. The display region is automatically updated when the new filter has been imported.

You can edit the imported filter using the Pole-Zero Editor panel.

## Filter Structures

The available filter structures are:

- Direct Form, which includes direct-form I, direct-form II, direct-form I transposed, direct-form II transposed, and direct-form FIR
- Lattice, which includes lattice allpass, lattice MA min phase, lattice MA max phase, and lattice ARMA
- Discrete-time Filter (`dfilt` object)

The structure that you choose determines the type of coefficients that you need to specify in the text fields to the right.

### Direct-form

For direct-form I, direct-form II, direct-form I transposed, and direct-form II transposed, specify the filter by its transfer function representation

$$H(z) = \frac{b(1) + b(2)z^{-1} + b(3)z^{-2} + ...b(m+1)z^{-m}}{a(1) + a(2)z^{-1} + a(3)Z^{-3} + ...a(n+1)z^{-n}}$$

- The **Numerator** field specifies a variable name or value for the numerator coefficient vector `b`, which contains `m+1` coefficients in descending powers of $z$.
- The **Denominator** field specifies a variable name or value for the denominator coefficient vector `a`, which contains `n+1` coefficients in descending powers of $z$. For FIR filters, the **Denominator** is `1`.

Filters in transfer function form can be produced by all of the Signal Processing Toolbox filter design functions (such as `fir1`, `fir2`, `firpm`, `butter`, `yulewalk`). See “Transfer Function” on page 1-32 for more information.

#### Importing as second-order sections

For all direct-form structures, except direct-form FIR, you can import the filter in its second-order section representation:

$$H(z) = G \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

The **Gain** field specifies a variable name or a value for the gain $G$, and the **SOS Matrix** field specifies a variable name or a value for the $L$-by-6 SOS matrix

$$SOS = \begin{pmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{22} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{pmatrix}$$

whose rows contain the numerator and denominator coefficients $b_{ik}$ and $a_{ik}$ of the second-order sections of $H(z)$.

Filters in second-order section form can be produced by functions such as `tf2sos`, `zp2sos`, `ss2sos`, and `sosfilt`. See “Second-Order Sections (SOS)” on page 1-34 for more information.

### Lattice

For lattice allpass, lattice minimum and maximum phase, and lattice ARMA filters, specify the filter by its lattice representation:

- For lattice allpass, the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, where `N` is the filter order.
- For lattice MA (minimum or maximum phase), the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, where `N` is the filter order.

- For lattice ARMA, the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, and the **Ladder coeff** field specifies the ladder coefficients, `v(1)` to `v(N+1)`, where `N` is the filter order.

Filters in lattice form can be produced by `tf2latc`. See “Lattice Structure” on page 1-35 for more information.

#### Discrete-time Filter (dfilt object)

For Discrete-time filter, specify the name of the `dfilt` object.

# FIR Bandpass Filter with Asymmetric Attenuation

Use the **Filter Designer** app to create a 50th-order equiripple FIR bandpass filter to be used with signals sampled at 1 kHz.

```
N = 50;
Fs = 1e3;
```

Specify that the passband spans frequencies of 200–300 Hz and that the transition region on either side has a width of 50 Hz.

```
Fstop1 = 150;
Fpass1 = 200;
Fpass2 = 300;
Fstop2 = 350;
```

Specify weights for the optimization fit:

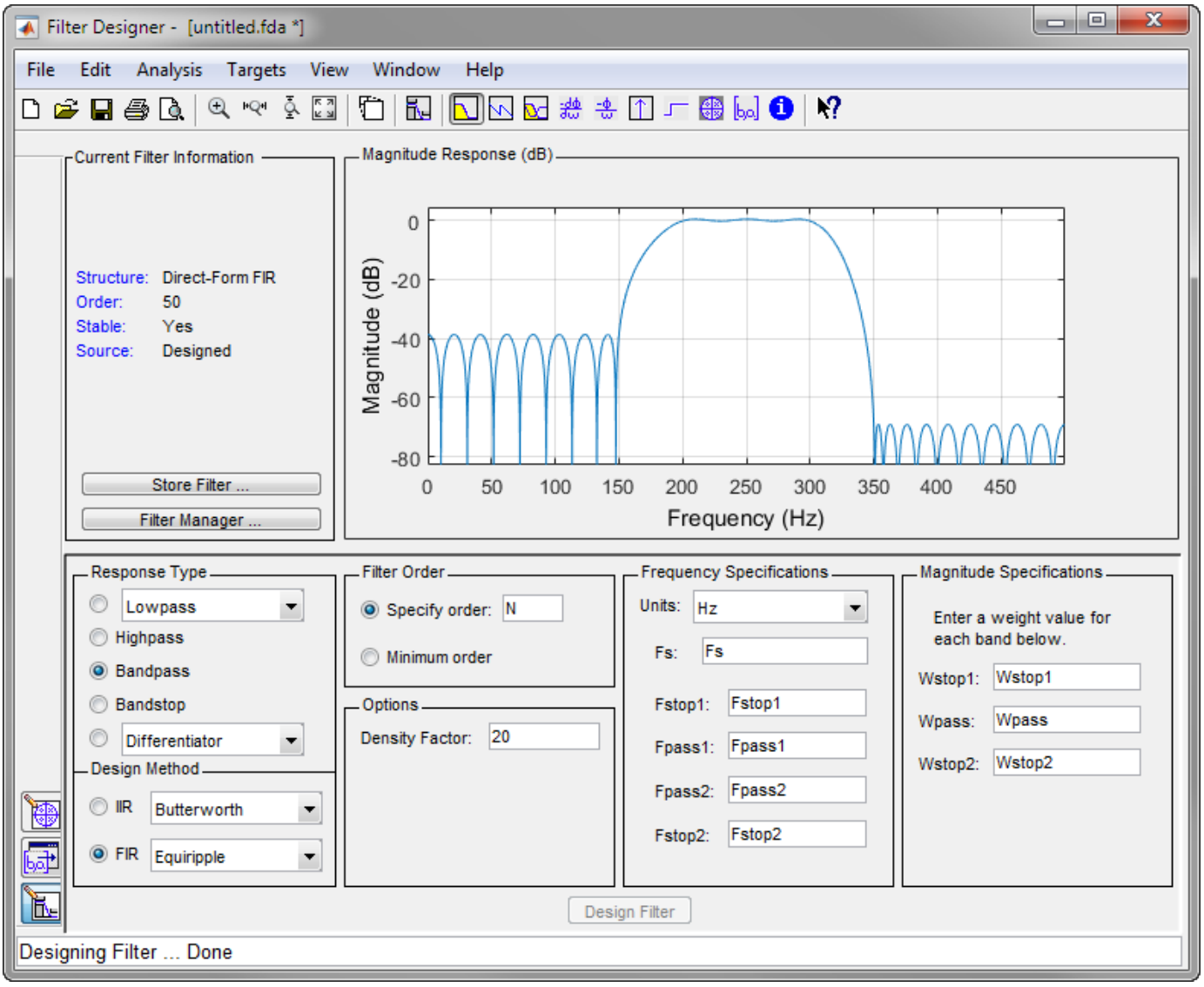
- 3 for the low-frequency stopband
- 1 for the passband
- 100 for the high-frequency stopband

Open the **Filter Designer** app.

```
Wstop1 = 3;
Wpass = 1;
Wstop2 = 100;

filterDesigner
```

Use the app to design the rest of the filter. To specify the frequency constraints and magnitude specifications, use the variables you created.

1. Set **Response Type** to `Bandpass`.
2. Set **Design Method** to `FIR`. From the drop-down list, select `Equiripple`.
3. Under **Filter Order**, specify the order as `N`.
4. Under **Frequency Specifications**, specify **Fs** as `Fs`.
5. Click **Design Filter**.




## See Also

**Apps**
**Filter Designer**

**Functions**
`designfilt`

# Arbitrary Magnitude Filter

Design an FIR filter with the following piecewise frequency response:

- A sinusoid between 0 and 0.19$\pi$ rad/sample.

  ```
  F1 = 0:0.01:0.19;
  A1 = 0.5+sin(2*pi*7.5*F1)/4;
  ```

- A piecewise linear section between 0.2$\pi$ rad/sample and 0.78$\pi$ rad/sample.

  ```
  F2 = [0.2 0.38 0.4 0.55 0.562 0.585 0.6 0.78];
  A2 = [0.5 2.3 1 1 -0.2 -0.2 1 1];
  ```

- A quadratic section between 0.79$\pi$ rad/sample and the Nyquist frequency.

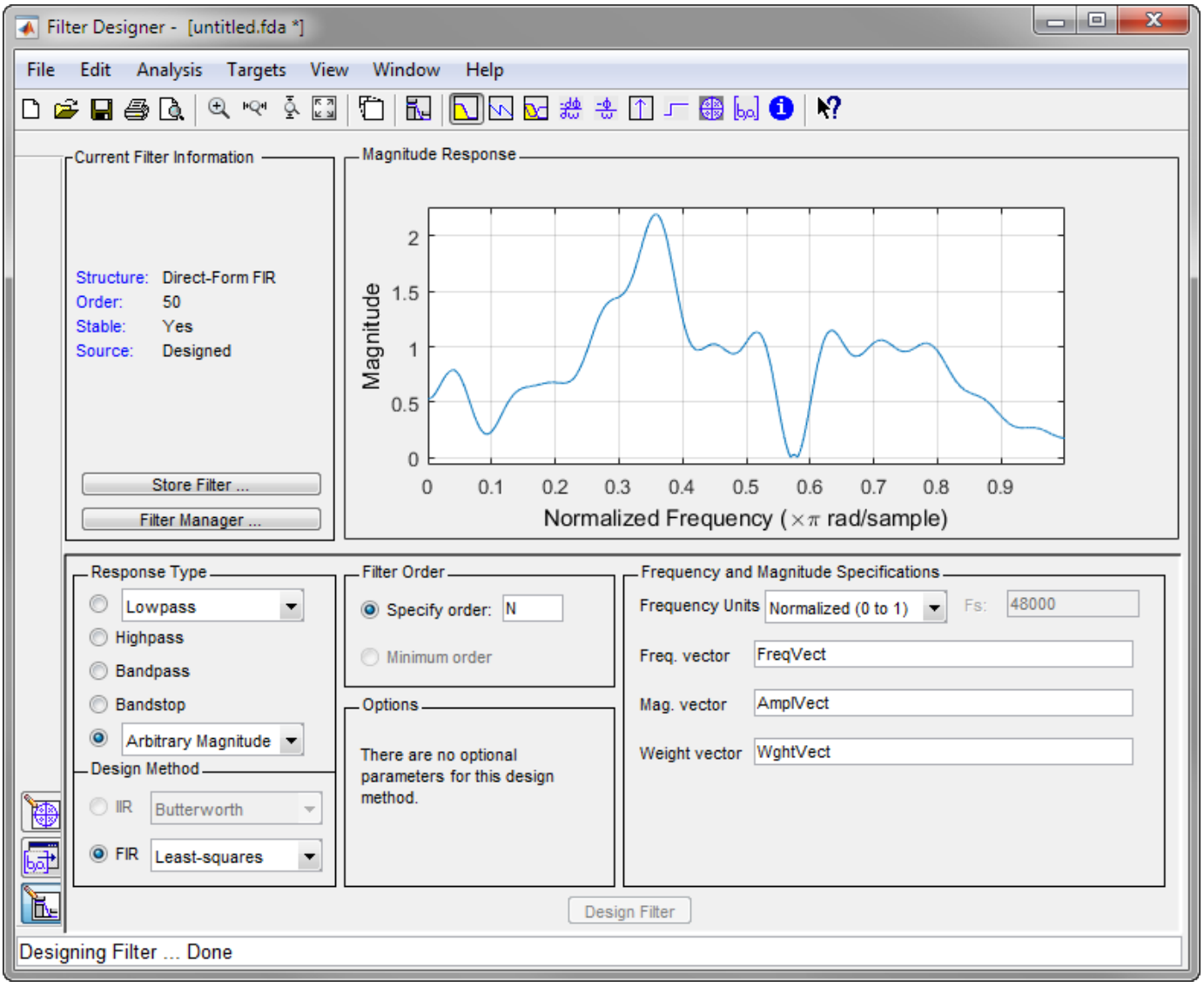
  ```
  F3 = 0.79:0.01:1;
  A3 = 0.2+18*(1-F3).^2;
  ```

Specify a filter order of 50. Consolidate the frequency and amplitude vectors. To give all bands equal weights during the optimization fit, specify a weight vector of all ones. Open the **Filter Designer** app.

```
N = 50;

FreqVect = [F1 F2 F3];
AmplVect = [A1 A2 A3];
WghtVect = ones(1,N/2);

filterDesigner
```

Use the app to design the filter.

1. Under **Response Type**, select the button next to `Differentiator`. From the drop-down list, choose `Arbitrary Magnitude`.
2. Set **Design Method** to `FIR`. From the drop-down list, select `Least-squares`.
3. Under **Filter Order**, specify the order as the variable `N`.
4. Under **Frequency and Magnitude Specifications**, specify the variables you created:
   - **Freq. vector** — `FreqVect`.
   - **Mag. vector** — `AmplVect`.
   - **Weight vector** — `WghtVect`.
5. Click **Design Filter**.
6. Right-click the *y*-axis of the plot and select **Magnitude** to express the magnitude response in linear units.




## See Also

**Apps**
**Filter Designer**

**Functions**
`designfilt`

# Filter Visualization Tool


- “Modifying the Axes” on page 6-2
- “Modifying the Plot” on page 6-4
- “Controlling FVTool from the MATLAB Command Line” on page 6-6

# Modifying the Axes

You can change the *x*- or *y*-axis units by right-clicking the mouse on the axis label or by right-clicking on the plot and selecting **Analysis Parameters**. Available options for the axes units are as follows.

| Plot | X-Axis Units | Y-Axis Units |
|---|---|---|
| Magnitude | Normalized Frequency<br>Linear Frequency | Magnitude<br>Magnitude (dB)<br>Magnitude squared<br>Zero-Phase |
| Phase | Normalized Frequency<br>Linear Frequency | Phase<br>Continuous Phase<br>Degrees<br>Radians |
| Magnitude and Phase | Normalized Frequency<br>Linear Frequency | (*y*-axis on left side)<br>Magnitude<br>Magnitude (dB)<br>Magnitude squared<br>Zero-Phase<br><br>(*y*-axis on right side)<br>Phase<br>Continuous Phase<br>Degrees<br>Radians |
| Group Delay | Normalized Frequency<br>Linear Frequency | Samples<br>Time |
| Phase Delay | Normalized Frequency<br>Linear Frequency | Degrees<br>Radians |
| Impulse Response | Samples<br>Time | Amplitude |
| Step Response | Samples<br>Time | Amplitude |
| Pole-Zero | Real Part | Imaginary Part |

## See Also

**Apps**
**Filter Designer | Signal Analyzer**

**Functions**
`designfilt` | `digitalFilter`

## Related Examples

- “Filter Analysis using FVTool” on page 24-202

## More About

- “Modifying the Plot” on page 6-4
- “Controlling FVTool from the MATLAB Command Line” on page 6-6

# Modifying the Plot

You can use any of the plot editing toolbar buttons to change the properties of your plot.

**Analysis Parameters** are parameters that apply to the displayed analyses. To display them, right-click in the plot area and select **Analysis Parameters** from the menu. (Note that you can access the menu only if the **Edit Plot** button is inactive.) The following analysis parameters are displayed. (If more than one response is displayed, parameters applicable to each plot are displayed.) Not all of these analysis fields are displayed for all types of plots:

- **Normalized Frequency** — if checked, frequency is normalized between 0 and 1, or if not checked, frequency is in Hz
- **Frequency Scale** — *y*-axis scale (`Linear` or `Log`)
- **Frequency Range** — range of the frequency axis or `Specify freq. vector`
- **Number of Points** — number of samples used to compute the response
- **Frequency Vector** — vector to use for plotting, if `Specify freq. vector` is selected in **Frequency Range**.
- **Magnitude Display** — *y*-axis units (`Magnitude`, `Magnitude (dB)`, `Magnitude squared`, or `Zero-Phase`)
- **Phase Units** — *y*-axis units (`Degrees` or `Radians`)
- **Phase Display** — type of phase plot (`Phase` or `Continuous Phase`)
- **Group Delay Units** — *y*-axis units (`Samples` or `Time`)
- **Specify Length** — length type of impulse or step response (`Default` or `Specified`)
- **Length** — number of points to use for the impulse or step response

In addition to the above analysis parameters, you can change the plot type for Impulse and Step Response plots by right-clicking and selecting **Line with Marker**, **Stem** or **Line** from the context menu. You can change the *x*-axis units by right-clicking the *x*-axis label and selecting `Samples` or `Time`.

To save the displayed parameters as the default values to use when **Filter Designer** or FVTool is opened, click **Save as default**.

To restore the default values, click **Restore original defaults**.

Data tips display information about a particular point in the plot. See “Interactively Explore Plotted Data” for information on data tips.

If you have the DSP System Toolbox software, FVTool displays a specification mask along with your designed filter on a magnitude plot.

**Note** To use **View > Passband zoom**, your filter must have been designed using `fdesign` or **Filter Designer**. Passband zoom is not provided for cascaded integrator-comb (CIC) filters because CICs do not have conventional passbands.

## See Also

**Apps**
**Filter Designer | Signal Analyzer**

**Functions**
`designfilt` | `digitalFilter`

## Related Examples

- “Filter Analysis using FVTool” on page 24-202

## More About

- “Modifying the Axes” on page 6-2
- “Controlling FVTool from the MATLAB Command Line” on page 6-6

# Controlling FVTool from the MATLAB Command Line

After you obtain the handle for FVTool, you can control some aspects of FVTool from the command line. In addition to the standard Handle Graphics® properties (see Handle Graphics in the MATLAB documentation), FVTool has the following properties:

- `'Analysis'` — displays the specified type of analysis plot. The following table lists all analysis types and how to invoke them. Note that the only analyses that use filter internals are magnitude response estimate and round-off noise power, which are available only with the DSP System Toolbox product.

| Analysis Type | Analysis Option |
|---|---|
| Magnitude plot | `'magnitude'` |
| Phase plot | `'phase'` |
| Magnitude and phase plot | `` `freq' `` |
| Group delay plot | `'grpdelay'` |
| Phase delay plot | `` `phasedelay' `` |
| Impulse response plot | `'impulse'` |
| Step response plot | `'step'` |
| Pole-zero plot | `'polezero'` |
| Filter coefficients | `'coefficients'` |
| Filter information | `'info'` |
| Magnitude response estimate<br><br>(available only with the DSP System Toolbox product, see `freqrespest` for more information) | `'magestimate'` |
| Round-off noise power<br><br>(available only with the DSP System Toolbox product, see `noisepsd` for more information) | `'noisepower'` |

- `'Grid'` — controls whether the grid is `'on'` or `'off'`
- `'Legend'` — controls whether the legend is `'on'` or `'off'`
- `'Fs'` — controls the sampling frequency of filters in FVTool. The sampling frequency vector must be of the same length as the number of filters or a scalar value. If it is a vector, each value is applied to its corresponding filter. If it is a scalar, the same value is applied to all filters.
- `SosViewSettings` — (This option is available only if you have the DSP System Toolbox product.) For second-order sections filters, this controls how the filter is displayed. The `SOSViewSettings` property contains an object so you must use this syntax to set it: `set(h.SOSViewSettings,'View',`*`viewtype`*`)`, where *`viewtype`* is one of the following:
  - `'Complete'` — Displays the complete response of the overall filter
  - `'Individual'` — Displays the response of each section separately
  - 'Cumulative' — Displays the response for each section accumulated with each prior section. If your filter has three sections, the first plot shows section one, the second plot shows the

accumulation of sections one and two, and the third plot show the accumulation of all three sections.

You can also define whether to use `SecondaryScaling`, which determines where the sections should be split. The secondary scaling points are the scaling locations between the recursive and the nonrecursive parts of the section. The default value is `false`, which does not use secondary scaling. To turn on secondary scaling, use this syntax: `set(h.SOSViewSettings,'View','Cumulative',true)`

- `'UserDefined'` — Allows you to define which sections to display and the order in which to display them. Enter a cell array where each section is represented by its index. If you enter one index, only that section is plotted. If you enter a range of indices, the combined response of that range of sections is plotted. For example, if your filter has four sections, entering `{1:4}` plots the combined response for all four sections, and entering `{1,2,3,4}` plots the response for each section individually.

**Note** You can change other properties of FVTool from the command line using the `set` function. Use `get(h)` to view property tags and current property settings.

You can use the following methods with the FVTool handle.

`addfilter(h,filtobj)` adds a new filter to FVTool. The new filter, `filtobj`, must be a `dfilt` filter object. You can specify the sampling frequency of the new filter with `addfilter(h,filtobj,'Fs',10)`.

`setfilter(h,filtobj)` replaces the filter in FVTool with the filter specified in `filtobj`. You can set the sampling frequency as described above.

`deletefilter(h, index)` deletes the filter at the FVTool cell array `index` location.

`legend(h,str1,str2,...)` creates a legend in FVTool by associating `str1` with filter 1, `str2` with filter 2, etc. See `legend`.

## See Also

**Apps**
**Filter Designer** | **Signal Analyzer**

**Functions**
`designfilt` | `digitalFilter`

## Related Examples

- “Filter Analysis using FVTool” on page 24-202

## More About

- “Modifying the Axes” on page 6-2
- “Modifying the Plot” on page 6-4

# Statistical Signal Processing

The following chapter discusses statistical signal processing tools and applications, including correlations, covariance, and spectral estimation.


- “Correlation and Covariance” on page 7-2
- “Spectral Analysis” on page 7-5
- “Nonparametric Methods” on page 7-8
- “Parametric Methods” on page 7-27
- “MUSIC and Eigenvector Analysis Methods” on page 7-37
- “Selected Bibliography” on page 7-39

# Correlation and Covariance

| In this section... |
| --- |
| “Background Information” on page 7-2 |
| “Using xcorr and xcov Functions” on page 7-2 |
| “Bias and Normalization” on page 7-3 |
| “Multiple Channels” on page 7-3 |

## Background Information

The cross-correlation sequence for two wide-sense stationary random process, $x(n)$ and $y(n)$ is

$$R_{xy}(m) = E\{x(n+m)y^*(n)\},$$

where the asterisk denotes the complex conjugate and the expectation is over the ensemble of realizations that constitute the random processes.

Note that cross-correlation is not commutative, but a Hermitian (conjugate) symmetry property holds such that:

$$R_{xy}(m) = R_{yx}^*(-m).$$

The cross-covariance between $x(n)$ and $y(n)$ is:

$$C_{xy}(m) = E\{(x(n+m) - \mu_x)(y(n) - \mu_y)^*\} = R_{xy}(m) - \mu_x\mu_y^*.$$

For zero-mean wide-sense stationary random processes, the cross-correlation and cross-covariance are equivalent.

In practice, you must estimate these sequences, because it is possible to access only a finite segment of the infinite-length random processes. Further, it is often necessary to estimate ensemble moments based on time averages because only a single realization of the random processes are available. A common estimate based on $N$ samples of $x(n)$ and $y(n)$ is the deterministic cross-correlation sequence (also called the time-ambiguity function)

$$\widehat{R}_{xy}(m) = \begin{cases} \sum_{n=0}^{N-m-1} x(n+m)y^*(n), & m \geq 0, \\ \widehat{R}_{yx}^*(-m), & m < 0. \end{cases}$$

where we assume for this discussion that $x(n)$ and $y(n)$ are indexed from 0 to $N$ – 1, and $\widehat{R}_{xy}(m)$ from –($N$ – 1) to $N$ – 1.

## Using xcorr and xcov Functions

The functions `xcorr` and `xcov` estimate the cross-correlation and cross-covariance sequences of random processes. They also handle autocorrelation and autocovariance as special cases. The `xcorr` function evaluates the sum shown above with an efficient FFT-based algorithm, given inputs $x(n)$ and $y(n)$ stored in length $N$ vectors `x` and `y`. Its operation is equivalent to convolution with one of the two subsequences reversed in time.

For example:

```
x = [1 1 1 1 1]';
y = x;
xyc = xcorr(x,y)
```

Notice that the resulting sequence length is one less than twice the length of the input sequence. Thus, the $N$th element is the correlation at lag 0. Also notice the triangular pulse of the output that results when convolving two square pulses.

The `xcov` function estimates autocovariance and cross-covariance sequences. This function has the same options and evaluates the same sum as `xcorr`, but first removes the means of `x` and `y`.

## Bias and Normalization

An estimate of a quantity is *biased* if its expected value is not equal to the quantity it estimates. The expected value of the output of `xcorr` is

$$E\left\{\widehat{R}_{xy}(m)\right\} = (N - |m|)R_{xy}(m)\,.$$

`xcorr` provides the unbiased estimate, dividing by $N$ – $|m|$ when you specify an `'unbiased'` flag after the input sequences.

```
xcorr(x,y,'unbiased')
```

Although this estimate is unbiased, the end points (near –($N$ – 1) and $N$ – 1) suffer from large variance because `xcorr` computes them using only a few data points. A possible trade-off is to simply divide by $N$ using the `'biased'` flag:

```
xcorr(x,y,'biased')
```

With this scheme, only the sample of the correlation at zero lag (the $N$th output element) is unbiased. This estimate is often more desirable than the unbiased one because it avoids random large variations at the end points of the correlation sequence.

`xcorr` provides one other normalization scheme. The syntax

```
xcorr(x,y,'coeff')
```

divides the output by `norm(x)*norm(y)` so that, for autocorrelations, the sample at zero lag is 1.

## Multiple Channels

For a multichannel signal, `xcorr` and `xcov` estimate the autocorrelation and cross-correlation and covariance sequences for all of the channels at once. If `S` is an $M$-by-$N$ signal matrix representing $N$ channels in its columns, `xcorr(S)` returns a (2$M$ – 1)-by-$N^2$ matrix with the autocorrelations and cross-correlations of the channels of `S` in its $N^2$ columns. If `S` is a three-channel signal

```
S = [s1 s2 s3]
```

then the result of `xcorr(S)` is organized as

```
R = [Rs1s1 Rs1s2 Rs1s3 Rs2s1 Rs2s2 Rs2s3 Rs3s1 Rs3s2 Rs3s3]
```

Two related functions, `cov` and `corrcoef`, are available in the standard MATLAB environment. They estimate covariance and normalized covariance respectively between the different channels at lag 0 and arrange them in a square matrix.

# Spectral Analysis

| In this section... |
| --- |
| “Background Information” on page 7-5<br>“Spectral Estimation Method” on page 7-6 |

## Background Information

The goal of *spectral estimation* is to describe the distribution (over frequency) of the power contained in a signal, based on a finite set of data. Estimation of power spectra is useful in a variety of applications, including the detection of signals buried in wideband noise.

The *power spectral density* (PSD) of a stationary random process $x(n)$ is mathematically related to the autocorrelation sequence by the discrete-time Fourier transform. In terms of normalized frequency, this is given by

$$P_{xx}(\omega) = \frac{1}{2\pi} \sum_{m=-\infty}^{\infty} R_{xx}(m) e^{-j\omega m}.$$

This can be written as a function of physical frequency $f$ (for example, in hertz) by using the relation $\omega = 2\pi f / f_s$, where $f_s$ is the sampling frequency:

$$P_{xx}(f) = \frac{1}{f_s} \sum_{m=-\infty}^{\infty} R_{xx}(m) e^{-j2\pi m f / f_s}.$$

The correlation sequence can be derived from the PSD by use of the inverse discrete-time Fourier transform:

$$R_{xx}(m) = \int_{-\pi}^{\pi} P_{xx}(\omega)\, e^{j\omega m} d\omega = \int_{-f_s/2}^{f_s/2} P_{xx}(f)\, e^{j2\pi m f / f_s} df.$$

The average power of the sequence $x(n)$ over the entire Nyquist interval is represented by

$$R_{xx}(0) = \int_{-\pi}^{\pi} P_{xx}(\omega) d\omega = \int_{-f_s/2}^{f_s/2} P_{xx}(f) df.$$

The average power of a signal over a particular frequency band $[\omega_1, \omega_2]$, $0 \le \omega_1 \le \omega_2 \le \pi$, can be found by integrating the PSD over that band:

$$\bar{P}_{[\omega_1, \omega_2]} = \int_{\omega_1}^{\omega_2} P_{xx}(\omega)\, d\omega = \int_{-\omega_2}^{-\omega_1} P_{xx}(\omega)\, d\omega.$$

You can see from the above expression that $P_{xx}(\omega)$ represents the power content of a signal in an *infinitesimal* frequency band, which is why it is called the power spectral *density*.

The units of the PSD are power (e.g., watts) per unit of frequency. In the case of $P_{xx}(\omega)$, this is watts/radian/sample or simply watts/radian. In the case of $P_{xx}(f)$, the units are watts/hertz. Integration of the PSD with respect to frequency yields units of watts, as expected for the average power.

For real–valued signals, the PSD is symmetric about DC, and thus $P_{xx}(\omega)$ for $0 \le \omega \le \pi$ is sufficient to completely characterize the PSD. However, to obtain the average power over the entire Nyquist interval, it is necessary to introduce the concept of the *one-sided* PSD.

The one-sided PSD is given by

$$P_{\text{one-sided}}(\omega) = \begin{cases} 0, & -\pi \le \omega < 0, \\ 2P_{xx}(\omega), & 0 \le \omega \le \pi. \end{cases}$$

The average power of a signal over the frequency band, $[\omega_1,\omega_2]$ with $0 \le \omega_1 \le \omega_2 \le \pi$, can be computed using the one-sided PSD as

$$\bar{P}_{[\omega_1,\omega_2]} = \int_{\omega_1}^{\omega_2} P_{\text{one-sided}}(\omega) d\omega .$$

## Spectral Estimation Method

The various methods of spectrum estimation available in the toolbox are categorized as follows:

- Nonparametric methods
- Parametric methods
- Subspace methods

*Nonparametric methods* are those in which the PSD is estimated directly from the signal itself. The simplest such method is the *periodogram*. Other nonparametric techniques such as *Welch's method* [8], the *multitaper method* (*MTM*) reduce the variance of the periodogram.

*Parametric methods* are those in which the PSD is estimated from a signal that is assumed to be output of a linear system driven by white noise. Examples are the *Yule-Walker autoregressive* (*AR*) *method* and the *Burg method*. These methods estimate the PSD by first estimating the parameters (coefficients) of the linear system that hypothetically generates the signal. They tend to produce better results than classical nonparametric methods when the data length of the available signal is relatively short. Parametric methods also produce smoother estimates of the PSD than nonparametric methods, but are subject to error from model misspecification.

*Subspace methods*, also known as *high-resolution methods* or *super-resolution methods*, generate frequency component estimates for a signal based on an eigenanalysis or eigendecomposition of the autocorrelation matrix. Examples are the multiple signal classification (*MUSIC*) *method* or the eigenvector (*EV*) *method*. These methods are best suited for line spectra — that is, spectra of sinusoidal signals — and are effective in the detection of sinusoids buried in noise, especially when the signal to noise ratios are low. The subspace methods do not yield true PSD estimates: they do not preserve process power between the time and frequency domains, and the autocorrelation sequence cannot be recovered by taking the inverse Fourier transform of the frequency estimate.

All three categories of methods are listed in the table below with the corresponding toolbox function names. More information about each function is on the corresponding function reference page. See “Parametric Modeling” on page 8-18 for details about `lpc` and other parametric estimation functions.

**Spectral Estimation Methods/Functions**

| Method | Description | Functions |
|---|---|---|
| Periodogram | Power spectral density estimate | `periodogram` |
| Welch | Averaged periodograms of overlapped, windowed signal sections | `pwelch`, `cpsd`, `tfestimate`, `mscohere` |
| Multitaper | Spectral estimate from combination of multiple orthogonal windows (or “tapers”) | `pmtm` |
| Yule-Walker AR | Autoregressive (AR) spectral estimate of a time-series from its estimated autocorrelation function | `pyulear` |
| Burg | Autoregressive (AR) spectral estimation of a time-series by minimization of linear prediction errors | `pburg` |
| Covariance | Autoregressive (AR) spectral estimation of a time-series by minimization of the forward prediction errors | `pcov` |
| Modified Covariance | Autoregressive (AR) spectral estimation of a time-series by minimization of the forward and backward prediction errors | `pmcov` |
| MUSIC | Multiple signal classification | `pmusic` |
| Eigenvector | Pseudospectrum estimate | `peig` |

# Nonparametric Methods

The following sections discuss the periodogram on page 7-8, modified periodogram on page 7-15, Welch on page 7-17, and multitaper on page 7-20 methods of nonparametric estimation, along with the related CPSD function on page 7-23, transfer function estimate on page 7-24, and coherence function on page 7-25.

## Periodogram

In general terms, one way of estimating the PSD of a process is to simply find the discrete-time Fourier transform of the samples of the process (usually done on a grid with an FFT) and appropriately scale the magnitude squared of the result. This estimate is called the *periodogram*.

The periodogram estimate of the PSD of a signal $x_L(n)$ of length $L$ is

$$P_{xx}(f) = \frac{1}{LF_s}\left|\sum_{n=0}^{L-1} x_L(n)e^{-j2\pi fn/F_s}\right|^2,$$

where Fs is the sampling frequency.

In practice, the actual computation of $P_{xx}(f)$ can be performed only at a finite number of frequency points, and usually employs an FFT. Most implementations of the periodogram method compute the $N$-point PSD estimate at the frequencies

$$f_k = \frac{kF_s}{N}, \quad k = 0, 1, ..., N-1\,.$$

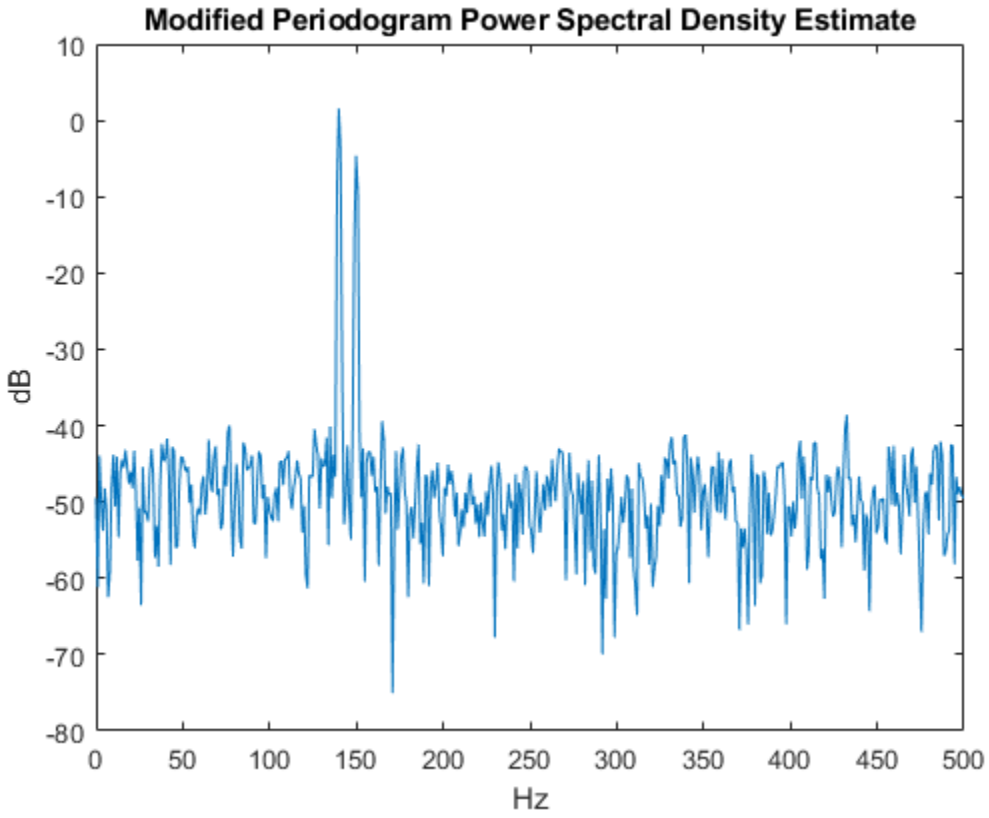
In some cases, the computation of the periodogram via an FFT algorithm is more efficient if the number of frequencies is a power of two. Therefore it is not uncommon to pad the input signal with zeros to extend its length to a power of two.

As an example of the periodogram, consider the following 1001-element signal `xn`, which consists of two sinusoids plus noise:

```
fs = 1000;                % Sampling frequency
t = (0:fs)/fs;            % One second worth of samples
A = [1 2];                % Sinusoid amplitudes (row vector)
f = [150;140];            % Sinusoid frequencies (column vector)
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
% The three last lines are equivalent to
% xn = sin(2*pi*150*t) + 2*sin(2*pi*140*t) + 0.1*randn(size(t));
```

The periodogram estimate of the PSD can be computed using `periodogram`. In this case, the data vector is multiplied by a Hamming window to produce a modified periodogram.

```
[Pxx,F] = periodogram(xn,hamming(length(xn)),length(xn),fs);
plot(F,10*log10(Pxx))
xlabel('Hz')
ylabel('dB')
title('Modified Periodogram Power Spectral Density Estimate')
```




### Algorithm

Periodogram computes and scales the output of the FFT to produce the power vs. frequency plot as follows.

1. If the input signal is real-valued, the magnitude of the resulting FFT is symmetric with respect to zero frequency (DC). For an even-length FFT, only the first (1 + `nfft`/2) points are unique. Determine the number of unique values and keep only those unique points.
2. Take the squared magnitudes of the unique FFT values. Scale the squared magnitudes (except for DC) by $2/(F_s N)$, where $N$ is the length of signal prior to any zero padding. Scale the DC value by $1/(F_s N)$.
3. Create a frequency vector from the number of unique points, the nfft and the sampling frequency.
4. Plot the resulting magnitude squared FFT against the frequency.

## Performance of the Periodogram

The following sections discuss the performance of the periodogram with regard to the issues of leakage, resolution, bias, and variance.

### Spectral Leakage

Consider the PSD of a finite-length (length $L$) signal $x_L(n)$. It is frequently useful to interpret $x_L(n)$ as the result of multiplying an infinite signal, $x(n)$, by a finite-length rectangular window, $w_R(n)$:

$$x_L(n) = x(n)w_R(n)\,.$$

Because multiplication in the time domain corresponds to convolution in the frequency domain, the expected value of the periodogram in the frequency domain is

$$E\{\widehat{P}_{xx}(f)\} = \frac{1}{F_s}\int_{-F_s/2}^{F_s/2}\frac{\sin^2(L\pi(f-f')/F_s)}{L\sin^2(\pi(f-f')/F_s)}P_{xx}(f')\,df',$$

showing that the expected value of the periodogram is the convolution of the true PSD with the square of the Dirichlet kernel.

The effect of the convolution is best understood for sinusoidal data. Suppose that $x(n)$ is composed of a sum of $M$ complex sinusoids:

$$x(n) = \sum_{k=1}^{N} A_k e^{j\omega_k n}\,.$$

Its spectrum is

$$X(\omega) = \sum_{k=1}^{N} A_k\delta(\omega-\omega_k),$$

which for a finite-length sequence becomes

$$X(\omega) = \int_{-\pi}^{\pi}\left(\sum_{k=1}^{N} A_k\delta(\varepsilon-\omega_k)\right)W_R(\omega-\varepsilon)\,d\varepsilon\,.$$

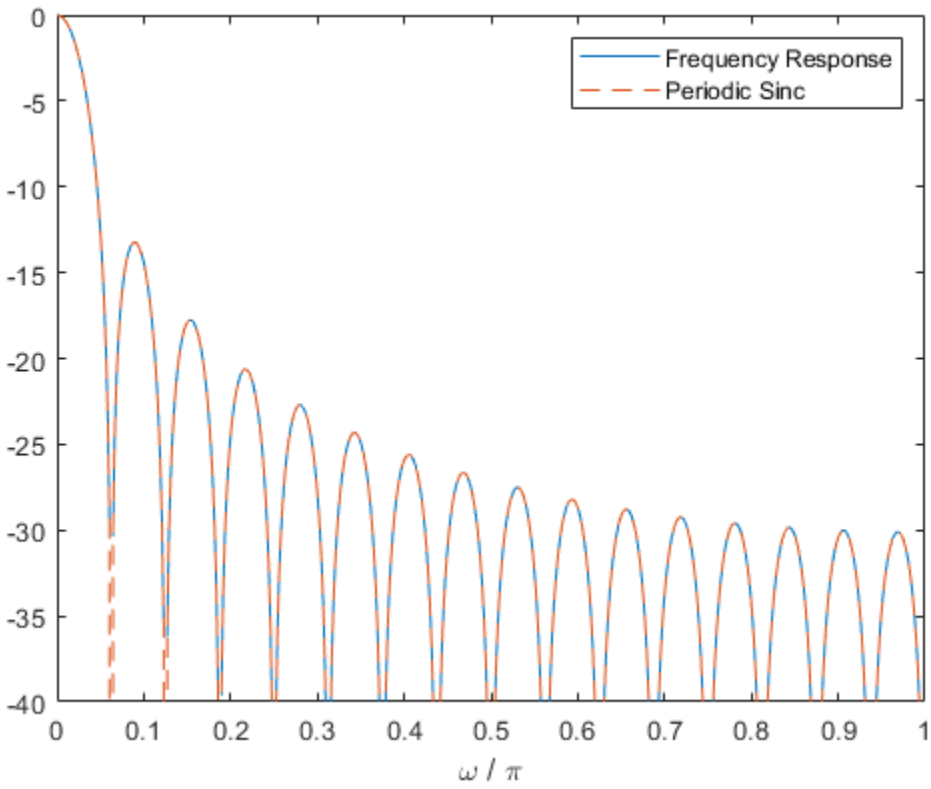
The preceding equation is equal to

$$X(\omega) = \sum_{k=1}^{N} A_k W_R(\omega-\omega_k)\,.$$

So in the spectrum of the finite-length signal, the Dirac deltas have been replaced by terms of the form $W_R(\omega-\omega_k)$, which corresponds to the frequency response of a rectangular window centered on the frequency $\omega_k$.

The frequency response of a rectangular window has the shape of a periodic sinc:

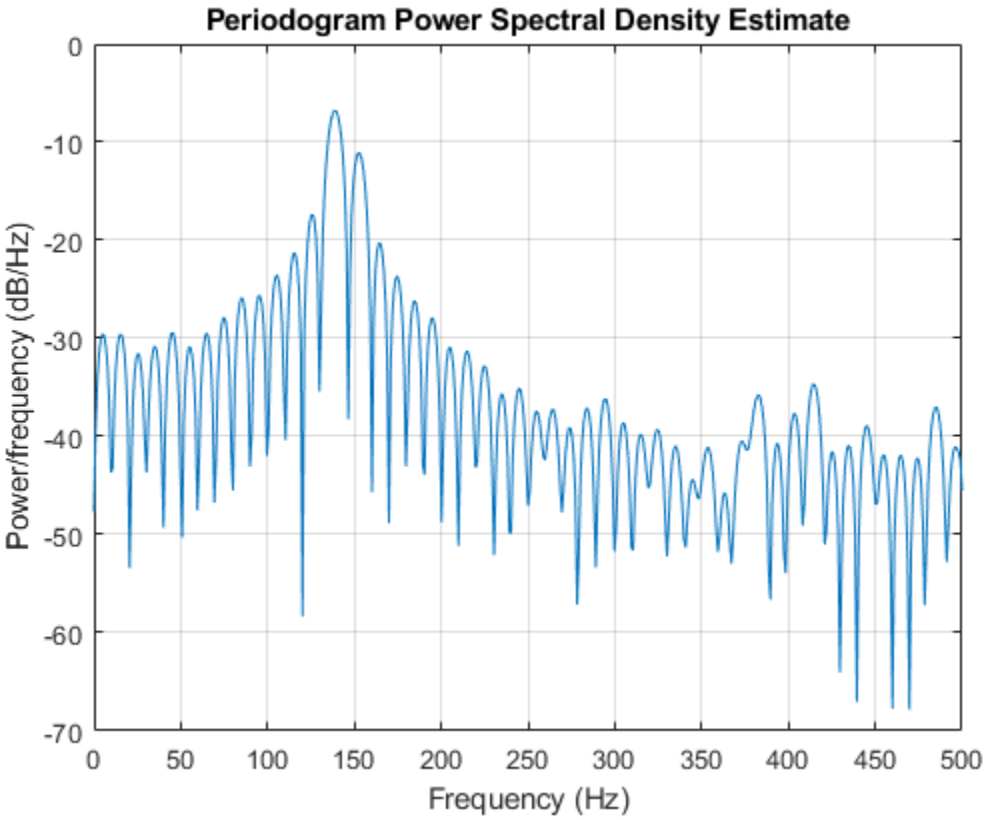
```
L = 32;
[h,w] = freqz(rectwin(L)/L,1);
y = diric(w,L);

plot(w/pi,20*log10(abs(h)))
hold on
plot(w/pi,20*log10(abs(y)),'--')
hold off
ylim([-40,0])
legend('Frequency Response','Periodic Sinc')
xlabel('\omega / \pi')
```




The plot displays a mainlobe and several sidelobes, the largest of which is approximately 13.5 dB below the mainlobe peak. These lobes account for the effect known as spectral leakage. While the infinite-length signal has its power concentrated exactly at the discrete frequency points $f_k$, the windowed (or truncated) signal has a continuum of power "leaked" around the discrete frequency points $f_k$.

Because the frequency response of a short rectangular window is a much poorer approximation to the Dirac delta function than that of a longer window, spectral leakage is especially evident when data records are short. Consider the following sequence of 100 samples:

```
fs = 1000;                  % Sampling frequency
t = (0:fs/10)/fs;           % One-tenth second worth of samples
A = [1 2];                  % Sinusoid amplitudes
f = [150;140];              % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
periodogram(xn,rectwin(length(xn)),1024,fs)
```




It is important to note that the effect of spectral leakage is contingent solely on the length of the data record. It is not a consequence of the fact that the periodogram is computed at a finite number of frequency samples.

**Resolution**

*Resolution* refers to the ability to discriminate spectral features, and is a key concept on the analysis of spectral estimator performance.

In order to resolve two sinusoids that are relatively close together in frequency, it is necessary for the difference between the two frequencies to be greater than the width of the mainlobe of the leaked spectra for either one of these sinusoids. The mainlobe width is defined to be the width of the mainlobe at the point where the power is half the peak mainlobe power (i.e., 3 dB width). This width is approximately equal to $f_s/L$.

In other words, for two sinusoids of frequencies $f_1$ and $f_2$, the resolvability condition requires that

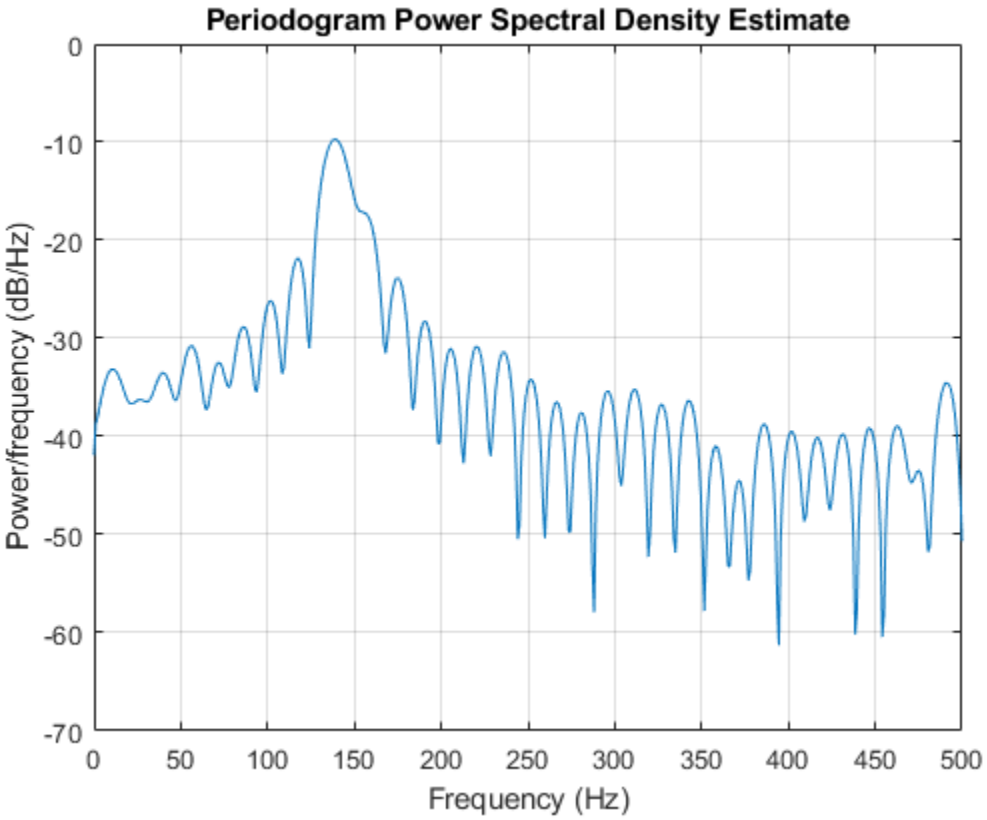
$$f_2 - f_1 > \frac{F_s}{L}.$$

In the example above, where two sinusoids are separated by only 10 Hz, the data record must be greater than 100 samples to allow resolution of two distinct sinusoids by a periodogram.

Consider a case where this criterion is not met, as for the sequence of 67 samples below:

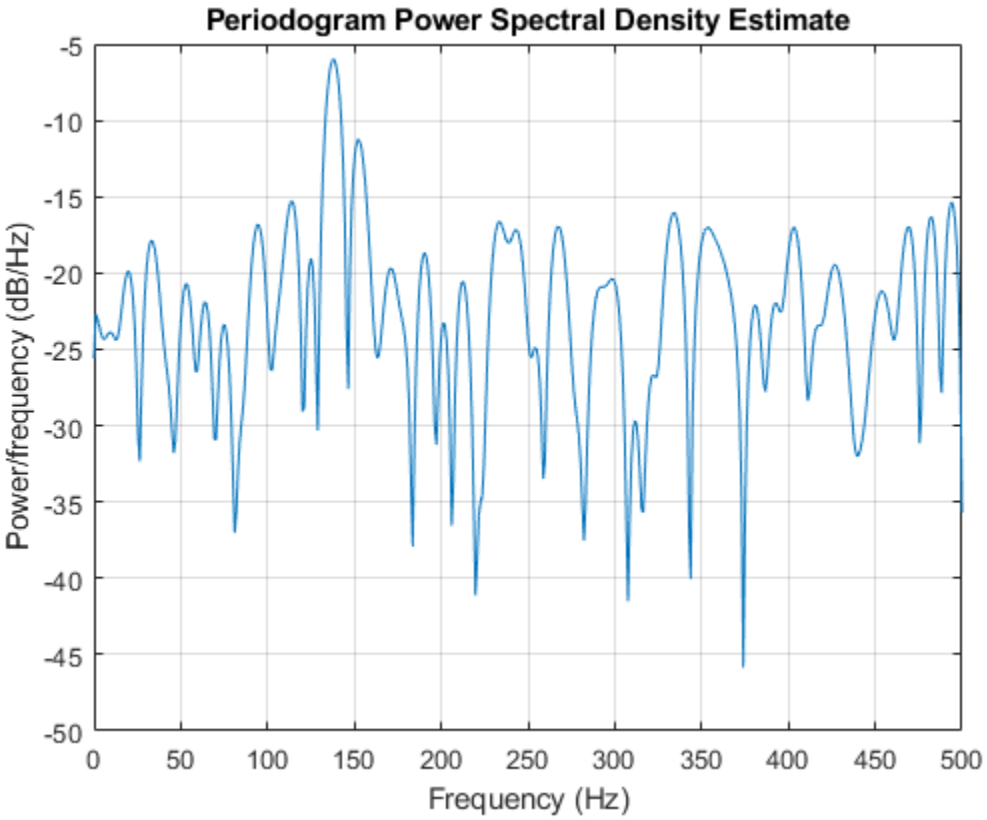
```
fs = 1000;                  % Sampling frequency
t = (0:fs/15)/fs;           % 67 samples
```

```
A = [1 2];                  % Sinusoid amplitudes
f = [150;140];              % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
periodogram(xn,rectwin(length(xn)),1024,fs)
```




The above discussion about resolution did not consider the effects of noise since the signal-to-noise ratio (SNR) has been relatively high thus far. When the SNR is low, true spectral features are much harder to distinguish, and noise artifacts appear in spectral estimates based on the periodogram. The example below illustrates this:

```
fs = 1000;                  % Sampling frequency
t = (0:fs/10)/fs;           % One-tenth second worth of samples
A = [1 2];                  % Sinusoid amplitudes
f = [150;140];              % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 2*randn(size(t));
periodogram(xn,rectwin(length(xn)),1024,fs)
```




**Bias of the Periodogram**

The periodogram is a biased estimator of the PSD. Its expected value was previously shown to be

$$E\{\widehat{P}_{xx}(f)\} = \frac{1}{F_s}\int_{-F_s/2}^{F_s/2} \frac{\sin^2(L\pi(f-f')/F_s)}{L\sin^2(\pi(f-f')/F_s)} P_{xx}(f')\,df'\,.$$

The periodogram is asymptotically unbiased, which is evident from the earlier observation that as the data record length tends to infinity, the frequency response of the rectangular window more closely approximates the Dirac delta function. However, in some cases the periodogram is a poor estimator of the PSD even when the data record is long. This is due to the variance of the periodogram, as explained below.

**Variance of the Periodogram**

The variance of the periodogram can be shown to be

$$\mathrm{Var}(\widehat{P}_{xx}(f)) = \begin{cases} P_{xx}^2(f), & 0 < f < F_s/2, \\ 2P_{xx}^2(f), & f = 0,\ F_s/2, \end{cases}$$

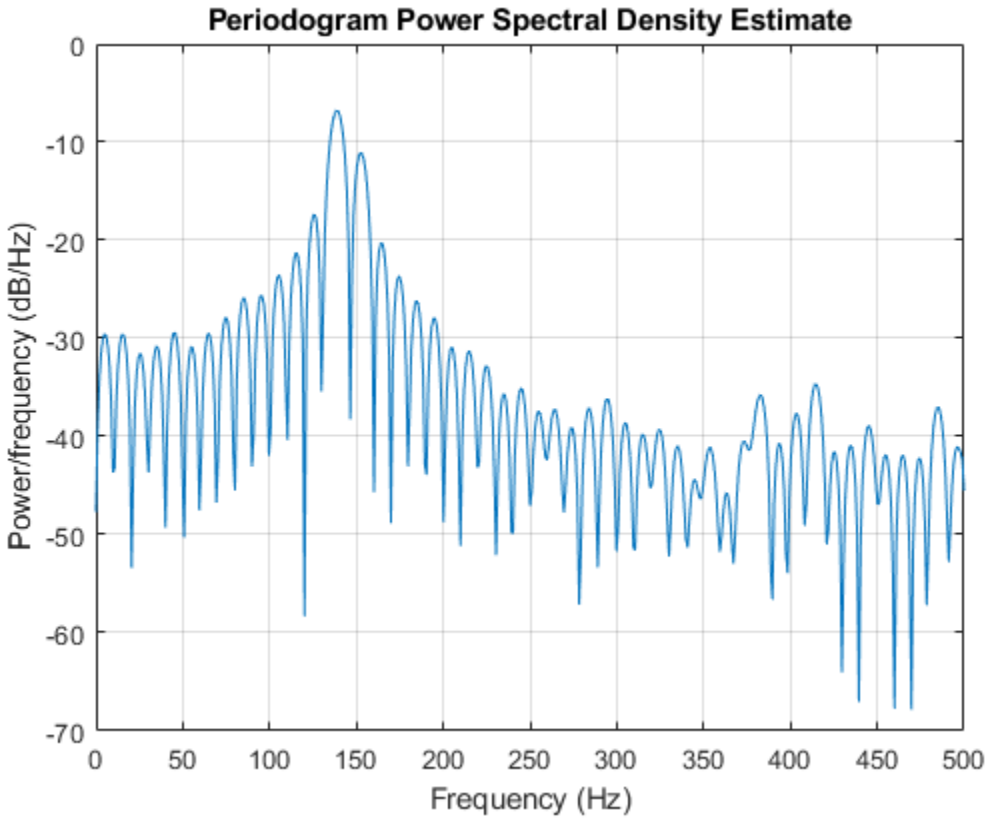
which indicates that the variance does not tend to zero as the data length $L$ tends to infinity. In statistical terms, the periodogram is not a consistent estimator of the PSD. Nevertheless, the periodogram can be a useful tool for spectral estimation in situations where the SNR is high, and especially if the data record is long.

## The Modified Periodogram

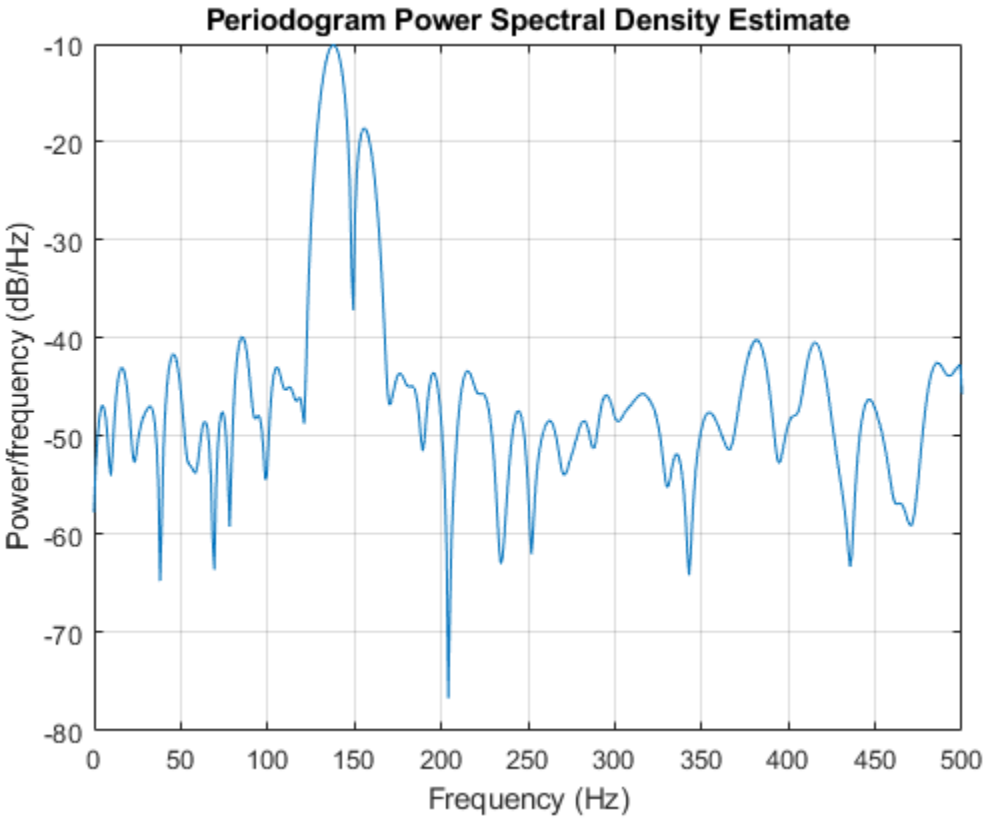
The *modified periodogram* windows the time-domain signal prior to computing the DFT in order to smooth the edges of the signal. This has the effect of reducing the height of the sidelobes or spectral leakage. This phenomenon gives rise to the interpretation of sidelobes as spurious frequencies introduced into the signal by the abrupt truncation that occurs when a rectangular window is used. For nonrectangular windows, the end points of the truncated signal are attenuated smoothly, and hence the spurious frequencies introduced are much less severe. On the other hand, nonrectangular windows also broaden the mainlobe, which results in a reduction of resolution.

The `periodogram` allows you to compute a modified periodogram by specifying the window to be used on the data. For example, compare a default rectangular window and a Hamming window. Specify the same number of DFT points in both cases.

```
fs = 1000;                  % Sampling frequency
t = (0:fs/10)/fs;           % One-tenth second worth of samples
A = [1 2];                  % Sinusoid amplitudes
f = [150;140];              % Sinusoid frequencies
nfft = 1024;

xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
periodogram(xn,rectwin(length(xn)),nfft,fs)
```




```
periodogram(xn,hamming(length(xn)),nfft,fs)
```




You can verify that although the sidelobes are much less evident in the Hamming-windowed periodogram, the two main peaks are wider. In fact, the 3 dB width of the mainlobe corresponding to a Hamming window is approximately twice that of a rectangular window. Hence, for a fixed data length, the PSD resolution attainable with a Hamming window is approximately half that attainable with a rectangular window. The competing interests of mainlobe width and sidelobe height can be resolved to some extent by using variable windows such as the Kaiser window.

Nonrectangular windowing affects the average power of a signal because some of the time samples are attenuated when multiplied by the window. To compensate for this, `periodogram` and `pwelch` normalize the window to have an average power of unity. This ensures that the measured average power is generally independent of window choice. If the frequency components are not well resolved by the PSD estimators, the window choice does affect the average power.

The modified periodogram estimate of the PSD is

$$\widehat{P}_{xx}(f) = \frac{|X(f)|^2}{F_s L U},$$

where $U$ is the window normalization constant:

$$U = \frac{1}{L}\sum_{n=0}^{N-1} |w(n)|^2 .$$

For large values of `L`, `U` tends to become independent of window length. The addition of `U` as a normalization constant ensures that the modified periodogram is asymptotically unbiased.

## Welch's Method

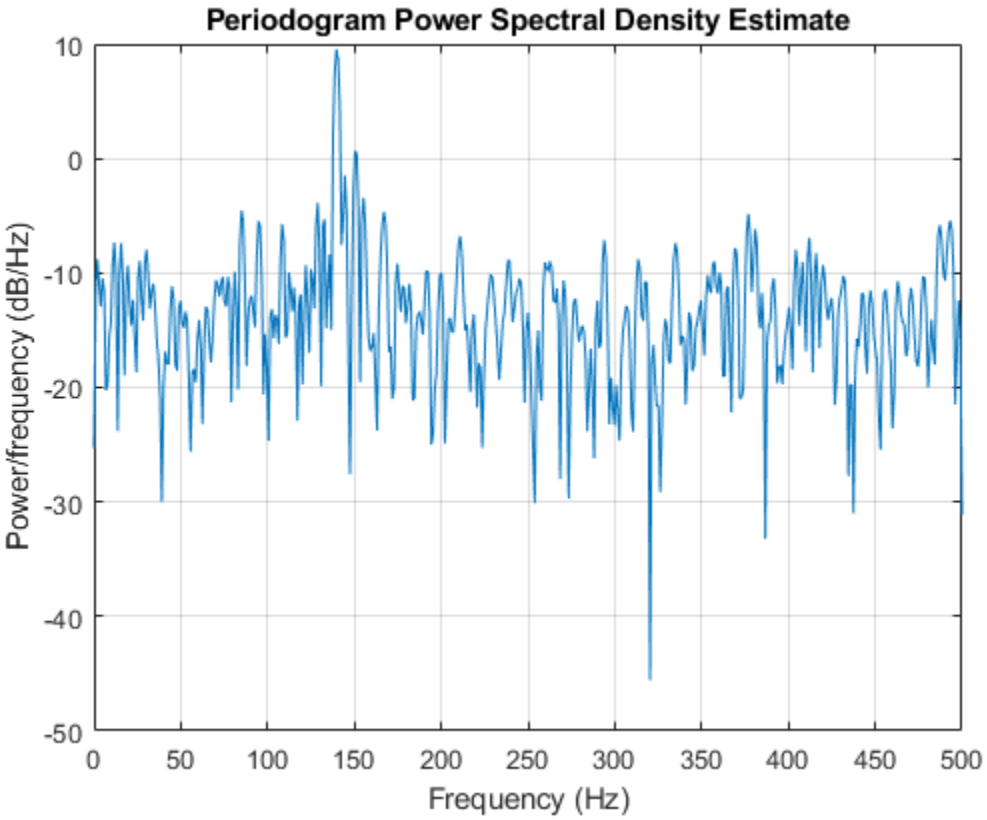
An improved estimator of the PSD is the one proposed by Welch. The method consists of dividing the time series data into (possibly overlapping) segments, computing a modified periodogram of each segment, and then averaging the PSD estimates. The result is Welch's PSD estimate. The toolbox function `pwelch` implements Welch's method.

The averaging of modified periodograms tends to decrease the variance of the estimate relative to a single periodogram estimate of the entire data record. Although overlap between segments introduces redundant information, this effect is diminished by the use of a nonrectangular window, which reduces the importance or *weight* given to the end samples of segments (the samples that overlap).

However, as mentioned above, the combined use of short data records and nonrectangular windows results in reduced resolution of the estimator. In summary, there is a tradeoff between variance reduction and resolution. One can manipulate the parameters in Welch's method to obtain improved estimates relative to the periodogram, especially when the SNR is low. This is illustrated in the following example.

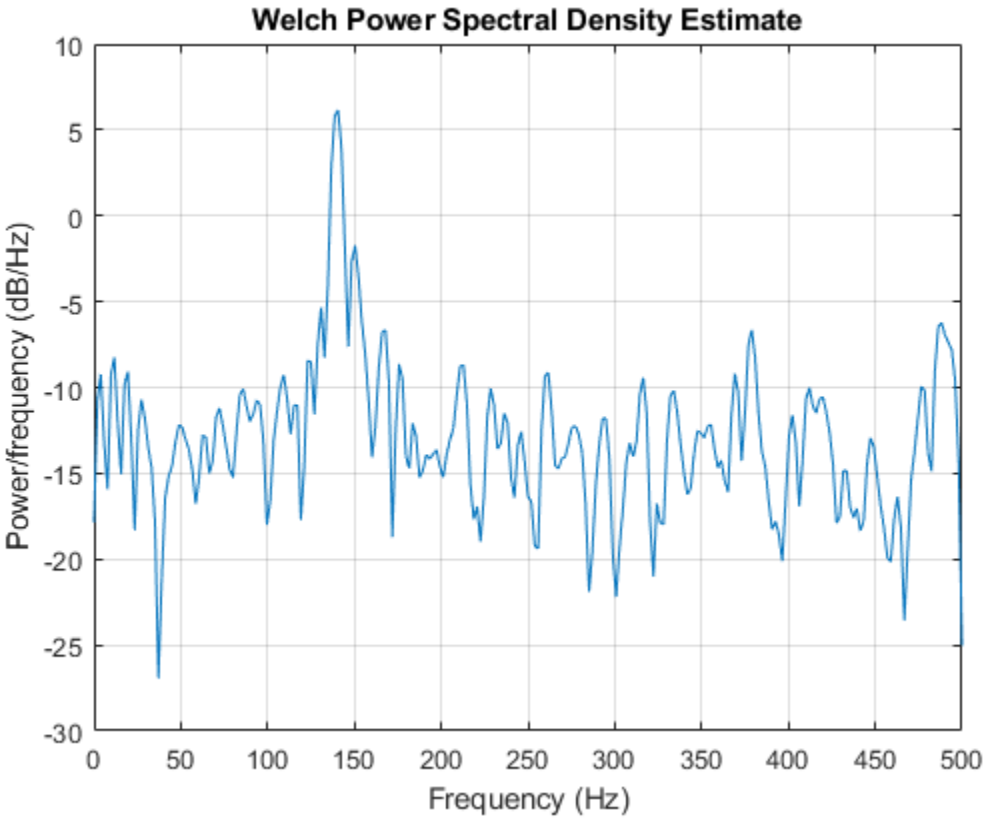
Consider a signal consisting of 301 samples:

```
fs = 1000;              % Sampling frequency
t = (0:0.3*fs)/fs;      % 301 samples
A = [2 8];              % Sinusoid amplitudes (row vector)
f = [150;140];          % Sinusoid frequencies (column vector)

xn = A*sin(2*pi*f*t) + 5*randn(size(t));
periodogram(xn,rectwin(length(xn)),1024,fs)
```




We can obtain Welch's spectral estimate for 3 segments with 50% overlap using a rectangular window.

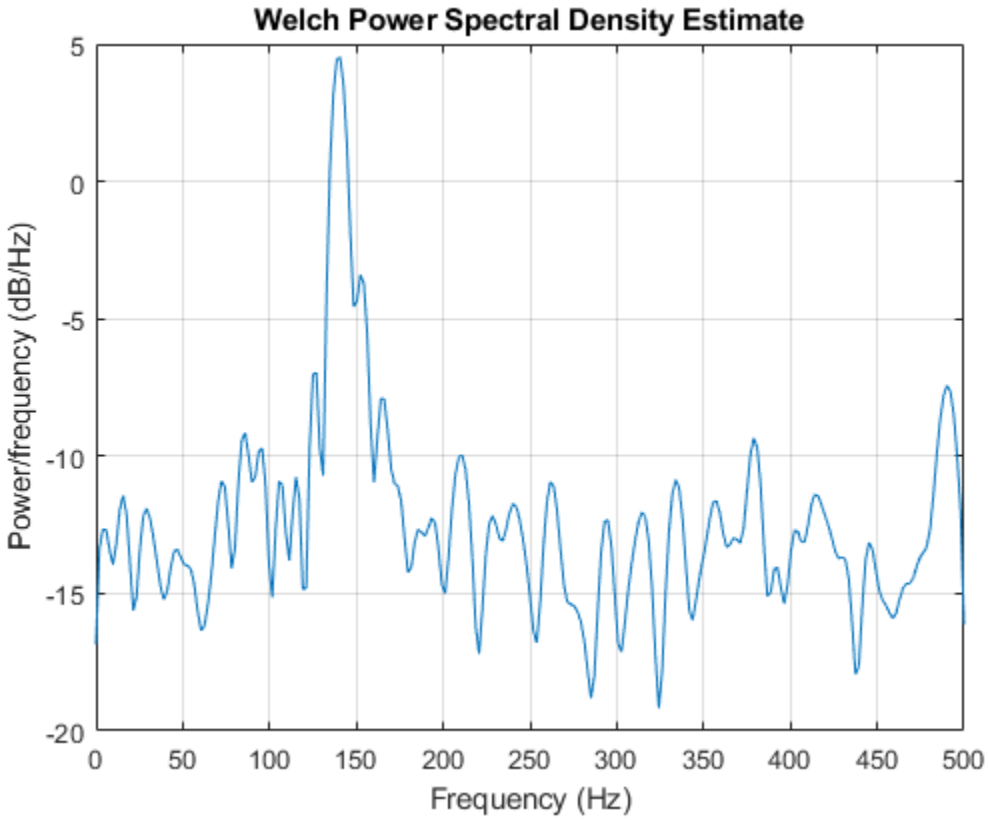
```
pwelch(xn,rectwin(150),50,512,fs)
```




In the periodogram above, noise and the leakage make one of the sinusoids essentially indistinguishable from the artificial peaks. In contrast, although the PSD produced by Welch's method has wider peaks, you can still distinguish the two sinusoids, which stand out from the "noise floor."

However, if we try to reduce the variance further, the loss of resolution causes one of the sinusoids to be lost altogether.

```
pwelch(xn,rectwin(100),75,512,fs)
```




## Bias and Normalization in Welch's Method

Welch's method yields a biased estimator of the PSD. The expected value of the PSD estimate is:

$$E\left\{P_{\text{Welch}}(f)\right\} = \frac{1}{F_s LU}\int_{-F_s/2}^{F_s/2} |W(f-f')|^2 P_{xx}(f')\,df',$$

where $L$ is the length of the data segments, $U$ is the same normalization constant present in the definition of the modified periodogram, and $W(f)$ is the Fourier transform of the window function. As is the case for all periodograms, Welch's estimator is asymptotically unbiased. For a fixed length data record, the bias of Welch's estimate is larger than that of the periodogram because the length of the segments is less than the length of the entire data sample.

The variance of Welch's estimator is difficult to compute because it depends on both the window used and the amount of overlap between segments. Basically, the variance is inversely proportional to the number of segments whose modified periodograms are being averaged.

## Multitaper Method

The periodogram can be interpreted as filtering a length $L$ signal, $x_L(n)$, through a filter bank (a set of filters in parallel) of $L$ FIR bandpass filters. The 3 dB bandwidth of each of these bandpass filters can be shown to be approximately equal to $f_s/L$. The magnitude response of each one of these bandpass filters resembles that of a rectangular window. The periodogram can thus be viewed as a computation

of the power of each filtered signal (i.e., the output of each bandpass filter) that uses just one sample of each filtered signal and assumes that the PSD of $x_L(n)$ is constant over the bandwidth of each bandpass filter.

As the length of the signal increases, the bandwidth of each bandpass filter decreases, making it a more selective filter, and improving the approximation of constant PSD over the bandwidth of the filter. This provides another interpretation of why the PSD estimate of the periodogram improves as the length of the signal increases. However, there are two factors apparent from this standpoint that compromise the accuracy of the periodogram estimate. First, the rectangular window yields a poor bandpass filter. Second, the computation of the power at the output of each bandpass filter relies on a single sample of the output signal, producing a very crude approximation.

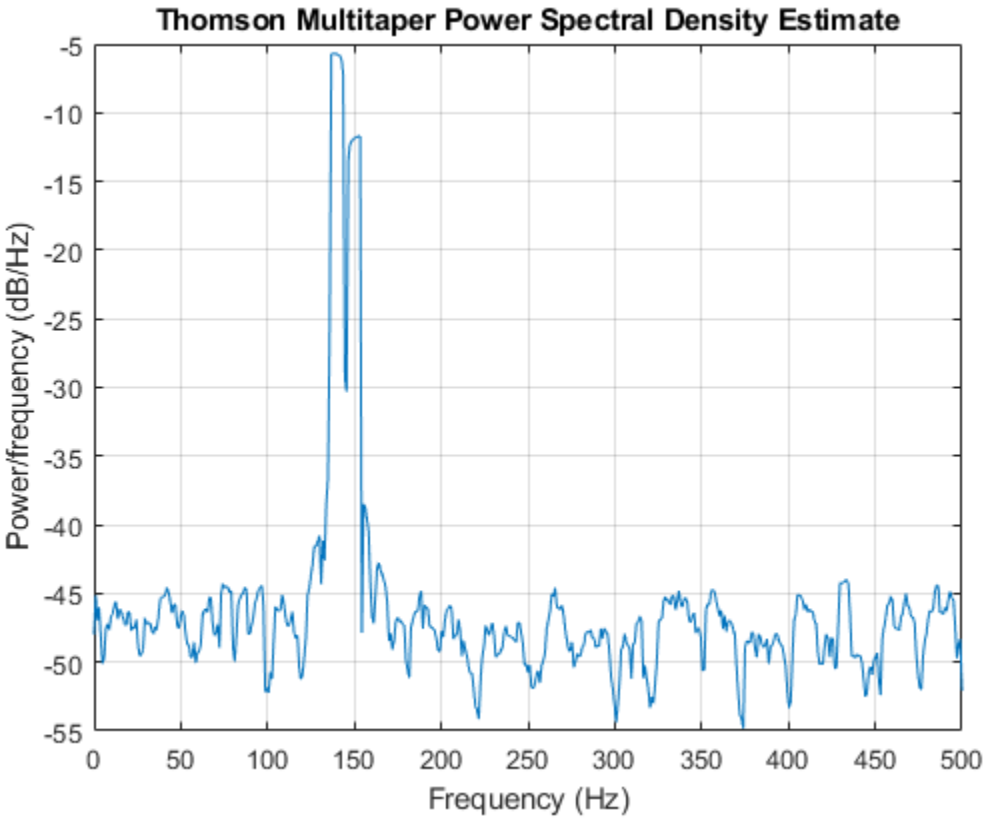
Welch's method can be given a similar interpretation in terms of a filter bank. In Welch's implementation, several samples are used to compute the output power, resulting in reduced variance of the estimate. On the other hand, the bandwidth of each bandpass filter is larger than that corresponding to the periodogram method, which results in a loss of resolution. The filter bank model thus provides a new interpretation of the compromise between variance and resolution.

Thompson's *multitaper method* (MTM) builds on these results to provide an improved PSD estimate. Instead of using bandpass filters that are essentially rectangular windows (as in the periodogram method), the MTM method uses a bank of optimal bandpass filters to compute the estimate. These optimal FIR filters are derived from a set of sequences known as discrete prolate spheroidal sequences (DPSSs, also known as *Slepian sequences*).

In addition, the MTM method provides a time-bandwidth parameter with which to balance the variance and resolution. This parameter is given by the time-bandwidth product, $NW$ and it is directly related to the number of tapers used to compute the spectrum. There are always $2NW - 1$ tapers used to form the estimate. This means that, as $NW$ increases, there are more estimates of the power spectrum, and the variance of the estimate decreases. However, the bandwidth of each taper is also proportional to $NW$, so as $NW$ increases, each estimate exhibits more spectral leakage (i.e., wider peaks) and the overall spectral estimate is more biased. For each data set, there is usually a value for $NW$ that allows an optimal trade-off between bias and variance.

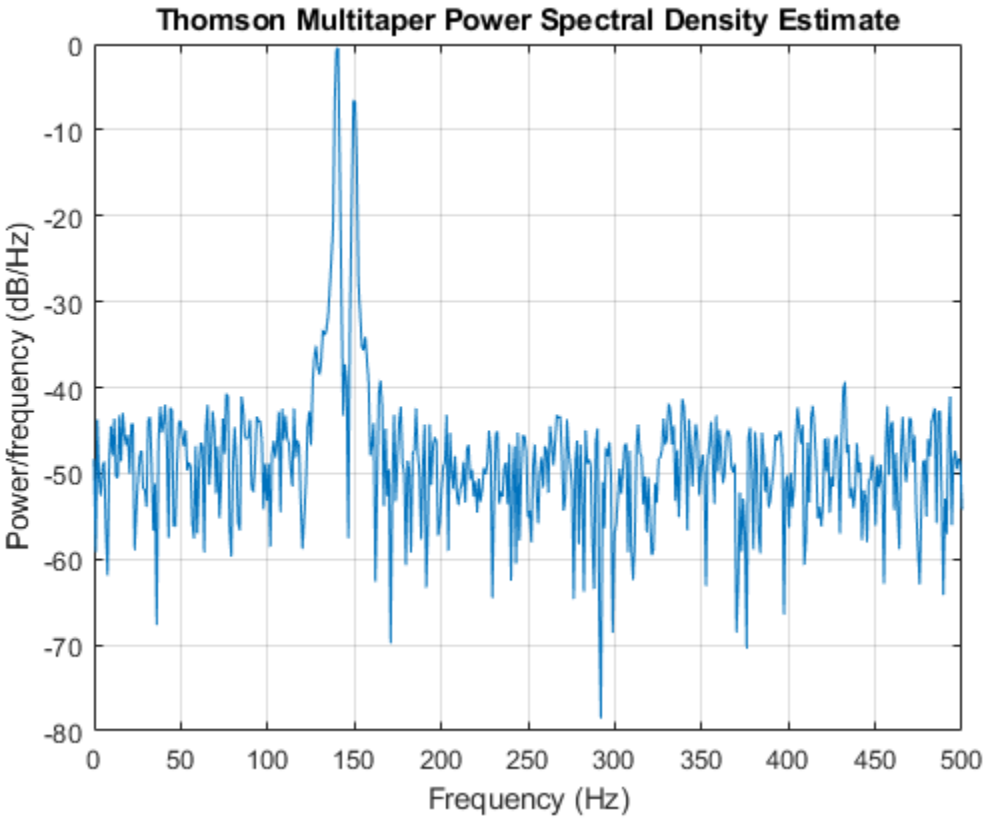
The Signal Processing Toolbox™ function that implements the MTM method is `pmtm`. Use `pmtm` to compute the PSD of a signal.

```
fs = 1000;                % Sampling frequency
t = (0:fs)/fs;            % One second worth of samples
A = [1 2];                % Sinusoid amplitudes
f = [150;140];            % Sinusoid frequencies

xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
pmtm(xn,4,[],fs)
```




By lowering the time-bandwidth product, you can increase the resolution at the expense of larger variance.

```
pmtm(xn,1.5,[],fs)
```




This method is more computationally expensive than Welch's method due to the cost of computing the discrete prolate spheroidal sequences. For long data series (10,000 points or more), it is useful to compute the DPSSs once and save them in a MAT-file. `dpsssave`, `dpssload`, `dpssdir`, and `dpssclear` are provided to keep a database of saved DPSSs in the MAT-file `dpss.mat`.

## Cross-Spectral Density Function

The PSD is a special case of the *cross spectral density* (CPSD) function, defined between two signals $x(n)$ and $y(n)$ as

$$P_{xy}(\omega) = \frac{1}{2\pi} \sum_{m=-\infty}^{\infty} R_{xy}(m) e^{-j\omega m}.$$

As is the case for the correlation and covariance sequences, the toolbox *estimates* the PSD and CPSD because signal lengths are finite.

To estimate the cross-spectral density of two equal length signals `x` and `y` using Welch's method, the `cpsd` function forms the periodogram as the product of the FFT of `x` and the conjugate of the FFT of `y`. Unlike the real-valued PSD, the CPSD is a complex function. `cpsd` handles the sectioning and windowing of `x` and `y` in the same way as the `pwelch` function:

```
Sxy = cpsd(x,y,nwin,noverlap,nfft,fs)
```

## Transfer Function Estimate

One application of Welch's method is nonparametric system identification. Assume that $H$ is a linear, time invariant system, and $x(n)$ and $y(n)$ are the input to and output of $H$, respectively. Then the power spectrum of $x(n)$ is related to the CPSD of $x(n)$ and $y(n)$ by

$$P_{yx}(\omega) = H(\omega)P_{xx}(\omega)\,.$$

An estimate of the transfer function between $x(n)$ and $y(n)$ is

$$\widehat{H}(\omega) = \frac{\widehat{P}_{yx}(\omega)}{\widehat{P}_{xx}(\omega)}\,.$$

This method estimates both magnitude and phase information. The `tfestimate` function uses Welch's method to compute the CPSD and power spectrum, and then forms their quotient for the transfer function estimate. Use `tfestimate` the same way that you use the `cpsd` function.

Generate a signal consisting of two sinusoids embedded in white Gaussian noise.

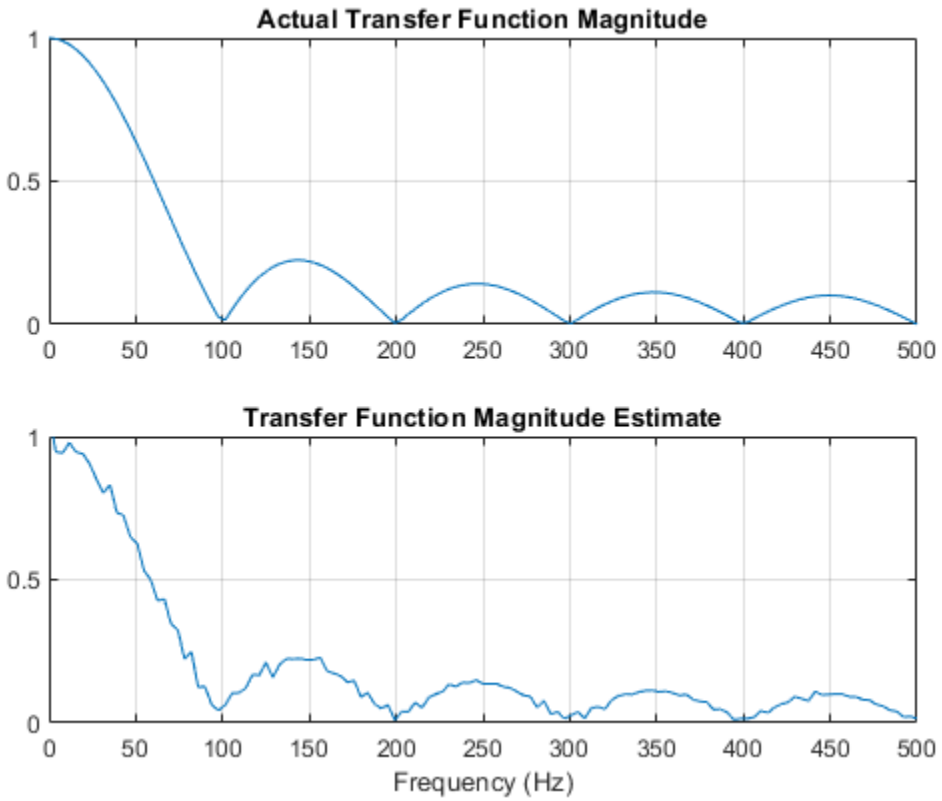
```
rng('default')

fs = 1000;                % Sampling frequency
t = (0:fs)/fs;            % One second worth of samples
A = [1 2];                % Sinusoid amplitudes
f = [150;140];            % Sinusoid frequencies

xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
```

Filter the signal `xn` with an FIR moving-average filter. Compute the actual magnitude response and the estimated response.

```
h = ones(1,10)/10;                % Moving-average filter
yn = filter(h,1,xn);

[HEST,f] = tfestimate(xn,yn,256,128,256,fs);
H = freqz(h,1,f,fs);
```

Plot the results.

```
subplot(2,1,1)
plot(f,abs(H))
title('Actual Transfer Function Magnitude')
yl = ylim;
grid
subplot(2,1,2)
plot(f,abs(HEST))
title('Transfer Function Magnitude Estimate')
xlabel('Frequency (Hz)')
ylim(yl)
grid
```




## Coherence Function

The magnitude-squared coherence between two signals $x(n)$ and $y(n)$ is

$$C_{xy}(\omega) = \frac{\left|P_{xy}(\omega)\right|^2}{P_{xx}(\omega)P_{yy}(\omega)}.$$

This quotient is a real number between 0 and 1 that measures the correlation between $x(n)$ and $y(n)$ at the frequency $\omega$.

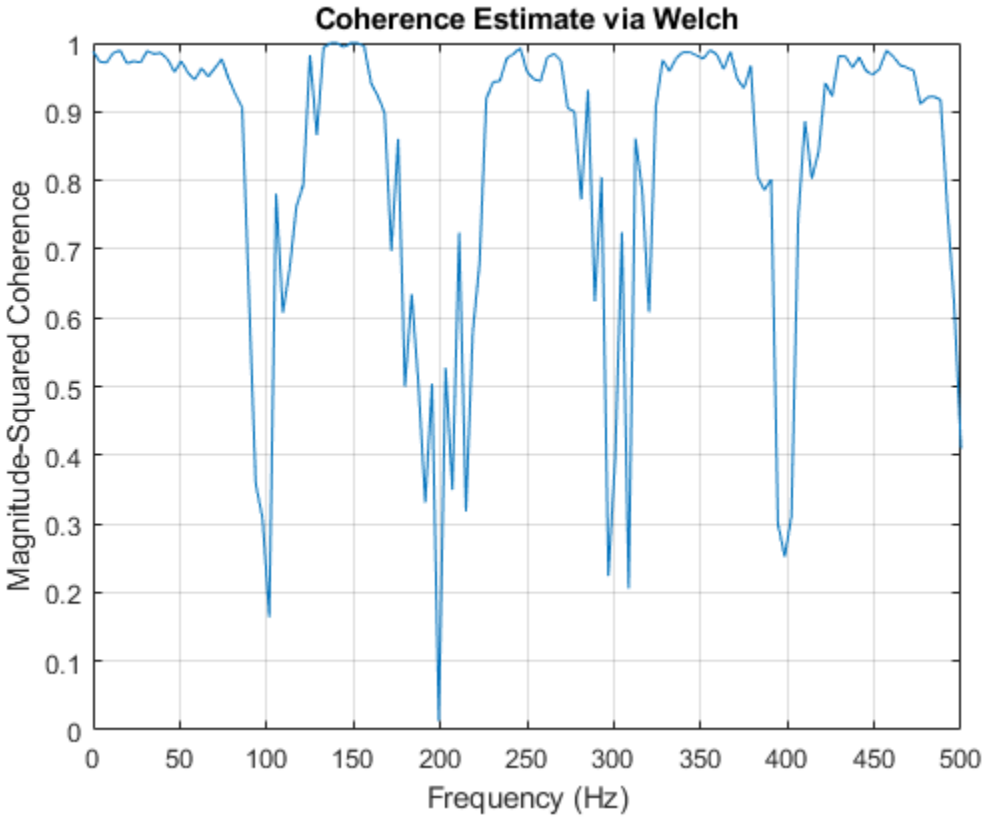
The `mscohere` function takes sequences `xn` and `yn`, computes their power spectra and CPSD, and returns the quotient of the magnitude squared of the CPSD and the product of the power spectra. Its options and operation are similar to the `cpsd` and `tfestimate` functions.

Generate a signal consisting of two sinusoids embedded in white Gaussian noise. The signal is sampled at 1 kHz for 1 second.

```
rng('default')

fs = 1000;
t = (0:fs)/fs;
A = [1 2];                  % Sinusoid amplitudes
f = [150;140];              % Sinusoid frequencies

xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
```

Filter the signal `xn` with an FIR moving-average filter. Compute and plot the coherence function of `xn` and the filter output `yn` as a function of frequency.

```
h = ones(1,10)/10;
yn = filter(h,1,xn);

mscohere(xn,yn,256,128,256,fs)
```




If the input sequence length, window length, and number of overlapping data points in a window are such that `mscohere` operates on only a single record, the function returns all ones. This is because the coherence function for linearly dependent data is one.

## See Also

**Apps**
**Signal Analyzer**

**Functions**
`cpsd` | `mscohere` | `periodogram` | `pmtm` | `pwelch` | `tfestimate`

# Parametric Methods

Parametric methods can yield higher resolutions than nonparametric methods in cases when the signal length is short. These methods use a different approach to spectral estimation; instead of trying to estimate the PSD directly from the data, they *model* the data as the output of a linear system driven by white noise, and then attempt to estimate the parameters of that linear system.

The most commonly used linear system model is the *all-pole model*, a filter with all of its zeroes at the origin in the *z*-plane. The output of such a filter for white noise input is an autoregressive (AR) process. For this reason, these methods are sometimes referred to as *AR methods* of spectral estimation.

The AR methods tend to adequately describe spectra of data that is “peaky,” that is, data whose PSD is large at certain frequencies. The data in many practical applications (such as speech) tends to have “peaky spectra” so that AR models are often useful. In addition, the AR models lead to a system of linear equations which is relatively simple to solve.

Signal Processing Toolbox AR methods for spectral estimation include:

- Yule-Walker AR method (autocorrelation method) on page 7-28
- Burg method on page 7-30
- Covariance method on page 7-34
- Modified covariance method on page 7-34

All AR methods yield a PSD estimate given by

$$\widehat{P}(f) = \frac{1}{F_s}\frac{\varepsilon_p}{\left|1 - \sum_{k=1}^{p} \widehat{a}_p(k) e^{-j2\pi kf/F_s}\right|^2}.$$

The different AR methods estimate the parameters slightly differently, yielding different PSD estimates. The following table provides a summary of the different AR methods.

**AR Methods**

| | Burg | Covariance | Modified Covariance | Yule-Walker |
|---|---|---|---|---|
| **Characteristics** | Does not apply window to data | Does not apply window to data | Does not apply window to data | Applies window to data |
| | Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion | Minimizes the forward prediction error in the least squares sense | Minimizes the forward and backward prediction errors in the least squares sense | Minimizes the forward prediction error in the least squares sense<br><br>(also called "Autocorrelation method") |
| **Advantages** | High resolution for short data records | Better resolution than Y-W for short data records (more accurate estimates) | High resolution for short data records | Performs as well as other methods for large data records |
| | Always produces a stable model | Able to extract frequencies from data consisting of $p$ or more pure sinusoids | Able to extract frequencies from data consisting of $p$ or more pure sinusoids | Always produces a stable model |
| | | | Does not suffer spectral line-splitting | |
| **Disadvantages** | Peak locations highly dependent on initial phase | May produce unstable models | May produce unstable models | Performs relatively poorly for short data records |
| | May suffer spectral line-splitting for sinusoids in noise, or when order is very large | Frequency bias for estimates of sinusoids in noise | Peak locations slightly dependent on initial phase | Frequency bias for estimates of sinusoids in noise |
| | Frequency bias for estimates of sinusoids in noise | | Minor frequency bias for estimates of sinusoids in noise | |
| **Conditions for Nonsingularity** | | Order must be less than or equal to half the input frame size | Order must be less than or equal to 2/3 the input frame size | Because of the biased estimate, the autocorrelation matrix is guaranteed to positive-definite, hence nonsingular |

## Yule-Walker AR Method

The *Yule-Walker AR method* of spectral estimation computes the AR parameters by solving the following linear system, which give the Yule-Walker equations in matrix form:

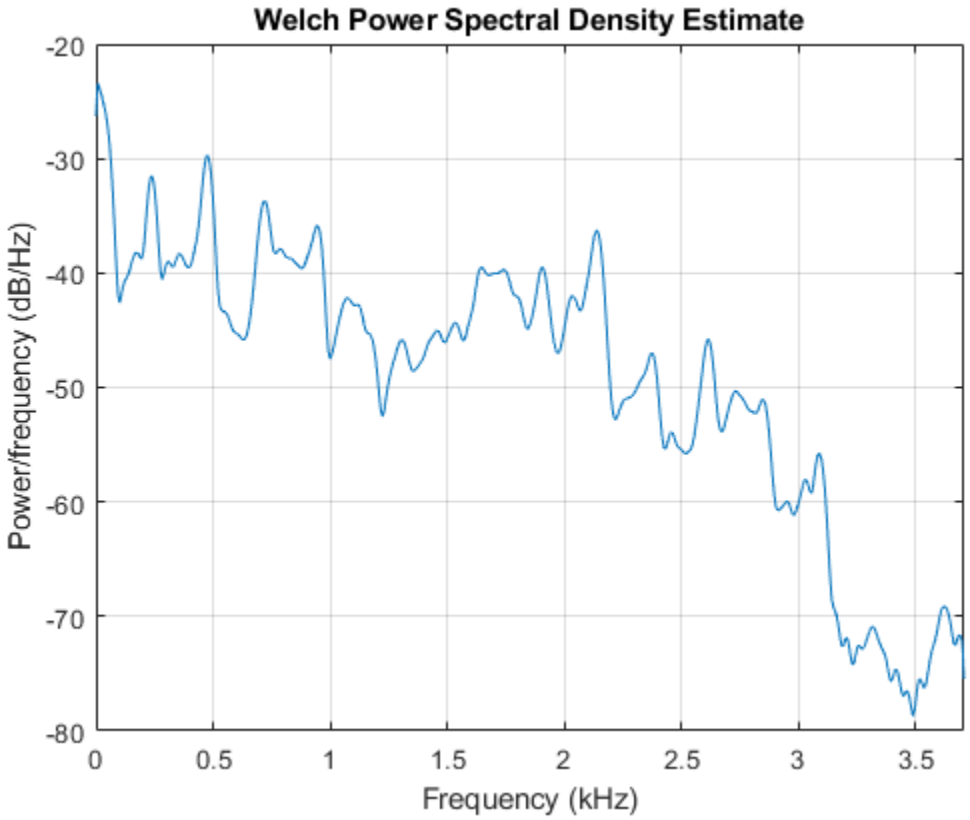
$$\begin{bmatrix} r(0) & r(1) & \cdots & r(p-1) \\ r(1) & r(0) & \cdots & r(p-2) \\ \vdots & \vdots & \ddots & \vdots \\ r(p-1) & r(p-2) & \cdots & r(0) \end{bmatrix} \begin{bmatrix} a(1) \\ a(2) \\ \vdots \\ a(p) \end{bmatrix} = \begin{bmatrix} r(1) \\ r(2) \\ \vdots \\ r(p) \end{bmatrix}.$$

In practice, the biased estimate of the autocorrelation is used for the unknown true autocorrelation. The Yule-Walker AR method produces the same results as a maximum entropy estimator.

The use of a biased estimate of the autocorrelation function ensures that the autocorrelation matrix above is positive definite. Hence, the matrix is invertible and a solution is guaranteed to exist. Moreover, the AR parameters thus computed always result in a stable all-pole model. The Yule-Walker equations can be solved efficiently using Levinson’s algorithm, which takes advantage of the Hermitian Toeplitz structure of the autocorrelation matrix.

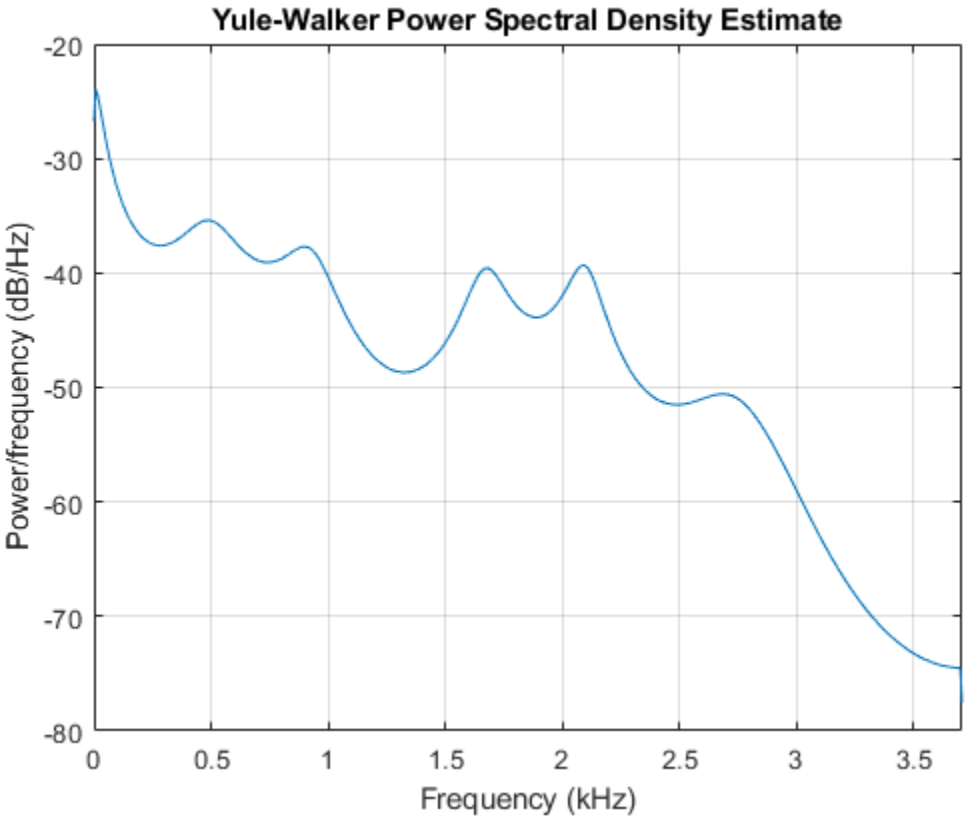
The toolbox function `pyulear` implements the Yule-Walker AR method. For example, compare the spectrum of a speech signal using Welch's method and the Yule-Walker AR method. Initially compute and plot the Welch periodogram.

```
load mtlb
pwelch(mtlb,hamming(256),128,1024,Fs)
```




The Yule-Walker AR spectrum is smoother than the periodogram because of the simple underlying all-pole model.

```
order = 14;
pyulear(mtlb,order,1024,Fs)
```




## Burg Method

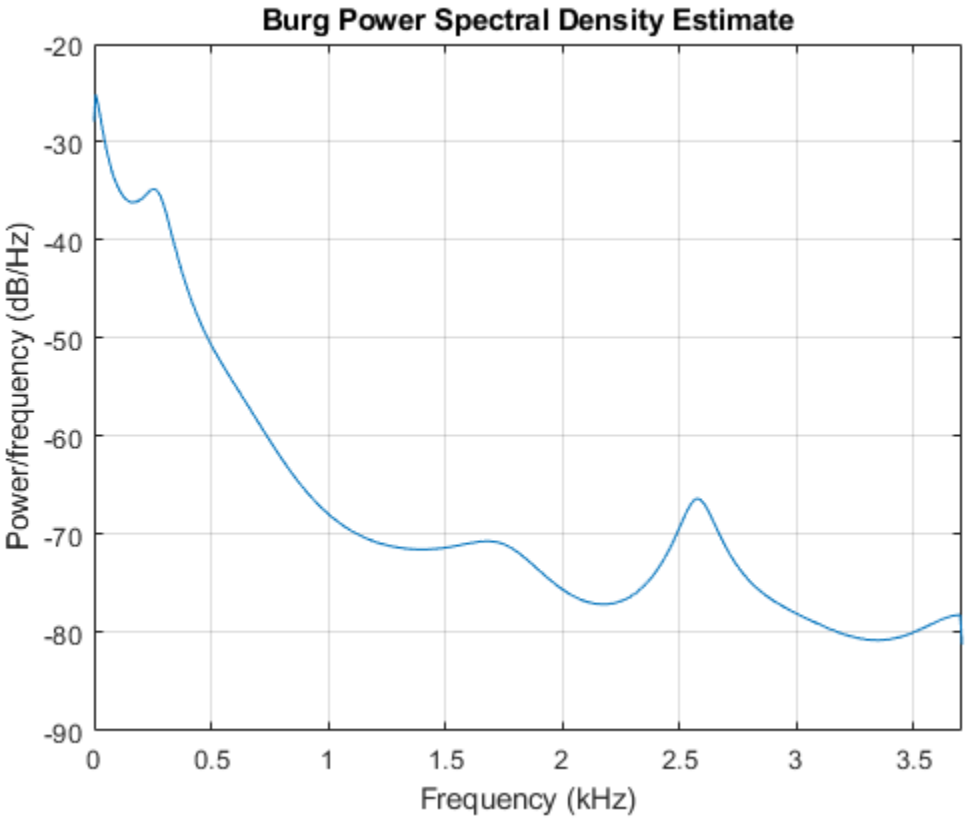
The Burg method for AR spectral estimation is based on minimizing the forward and backward prediction errors while satisfying the Levinson-Durbin recursion. In contrast to other AR estimation methods, the Burg method avoids calculating the autocorrelation function, and instead estimates the reflection coefficients directly.

The primary advantages of the Burg method are resolving closely spaced sinusoids in signals with low noise levels, and estimating short data records, in which case the AR power spectral density estimates are very close to the true values. In addition, the Burg method ensures a stable AR model and is computationally efficient.

The accuracy of the Burg method is lower for high-order models, long data records, and high signal-to-noise ratios (which can cause *line splitting*, or the generation of extraneous peaks in the spectrum estimate). The spectral density estimate computed by the Burg method is also susceptible to frequency shifts (relative to the true frequency) resulting from the initial phase of noisy sinusoidal signals. This effect is magnified when analyzing short data sequences.

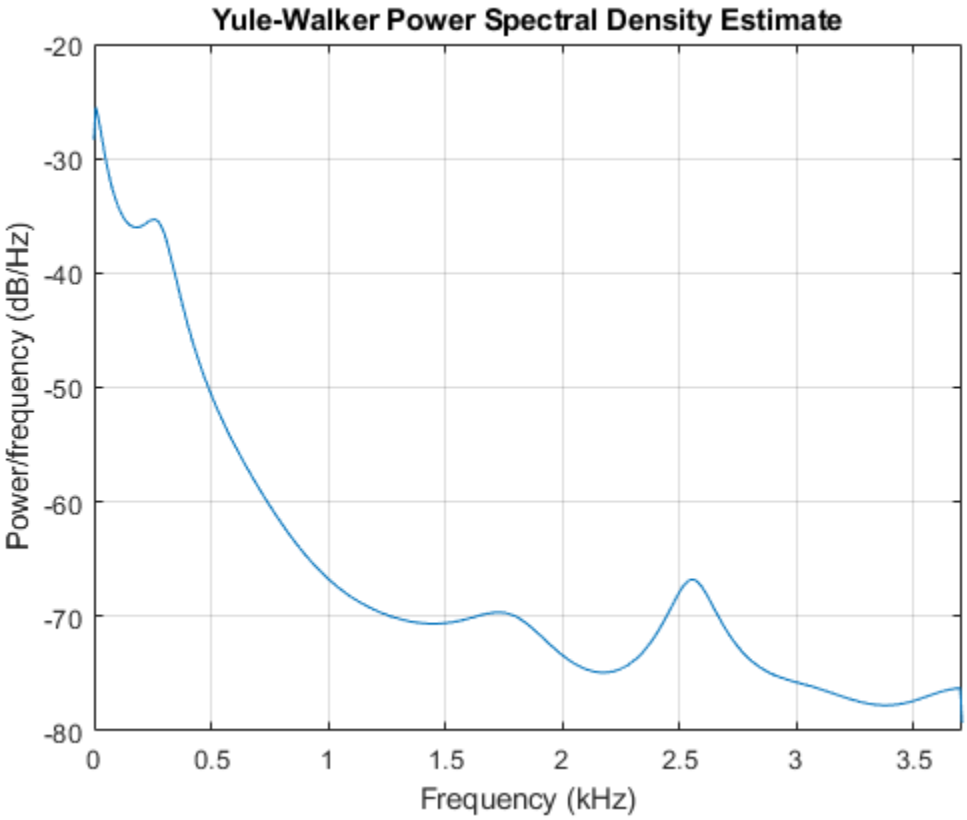
The toolbox function `pburg` implements the Burg method. Compare the spectrum estimates of a speech signal generated by both the Burg method and the Yule-Walker AR method. Initially compute and plot the Burg estimate.

```
load mtlb
order = 14;
pburg(mtlb(1:512),order,1024,Fs)
```



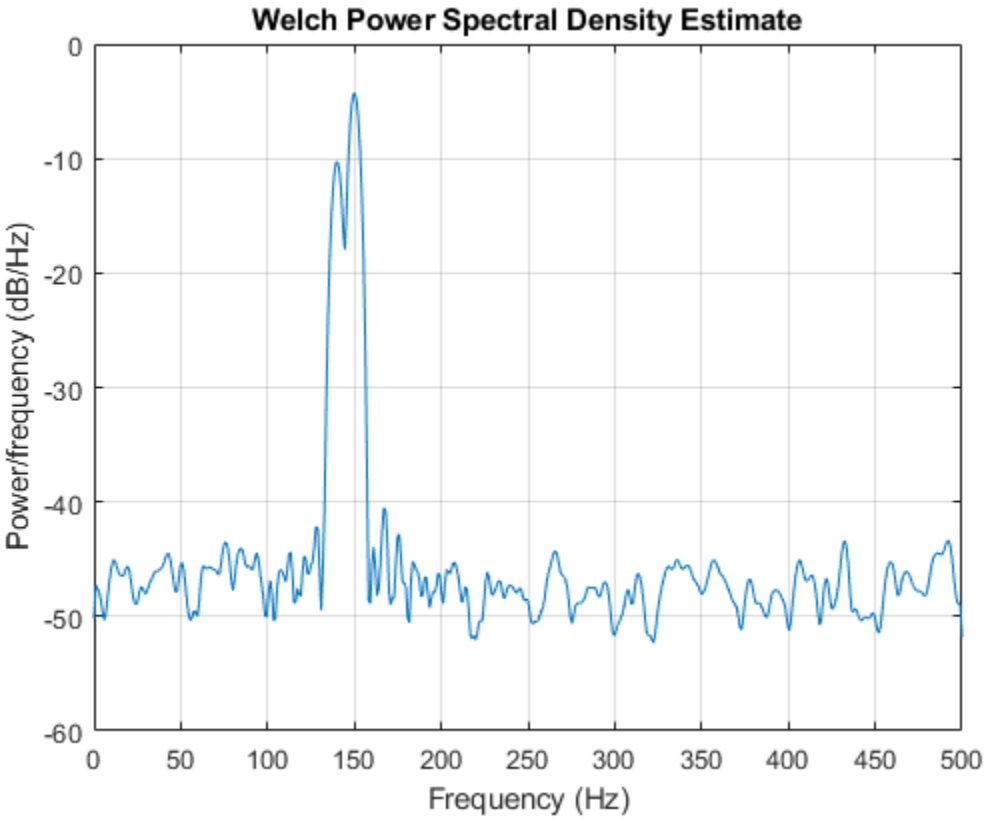

The Yule-Walker estimate is very similar if the signal is long enough.

```
pyulear(mtlb(1:512),order,1024,Fs)
```



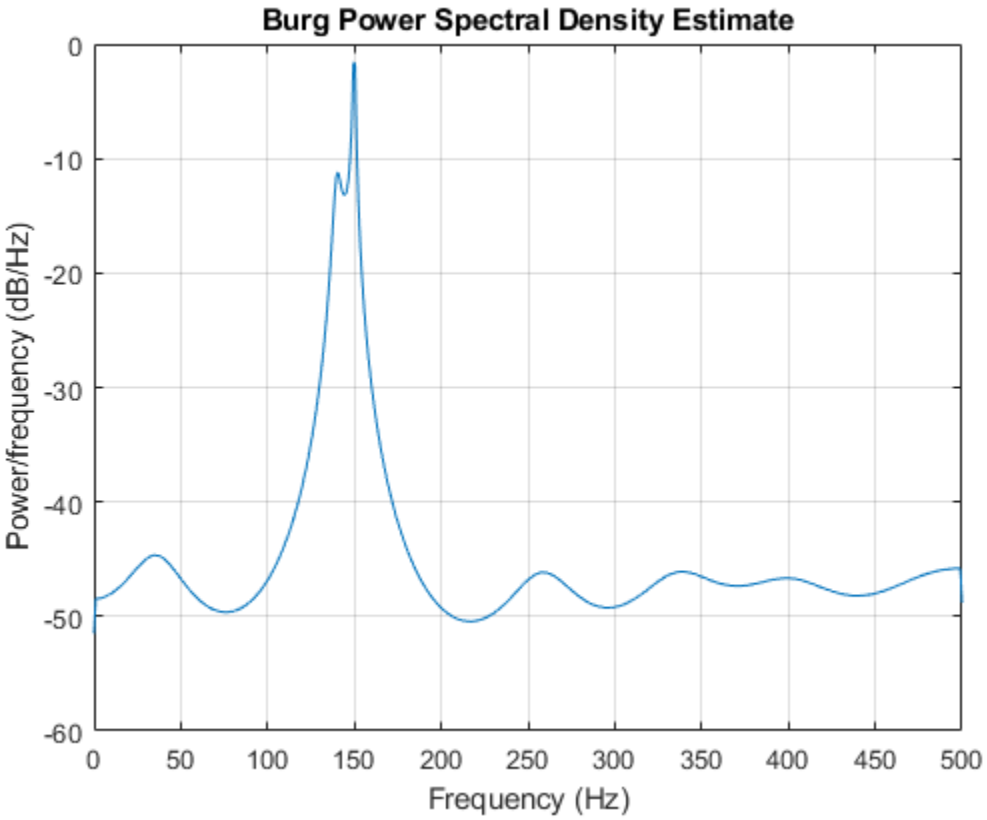

Compare the spectrum of a noisy signal computed using the Burg method and the Welch method. Create a two-component sinusoidal signal with frequencies 140 Hz and 150 Hz embedded in white Gaussian noise of variance $0.1^2$. The second component has twice the amplitude of the first component. The signal is sampled at 1 kHz for 1 second. Initially compute and plot the Welch spectrum estimate.

```
fs = 1000;
t = (0:fs)/fs;
A = [1 2];
f = [140;150];
xn = A*cos(2*pi*f*t) + 0.1*randn(size(t));

pwelch(xn,hamming(256),128,1024,fs)
```




Compute and plot the Burg estimate using a model of order 14.

```
pburg(xn,14,1024,fs)
```

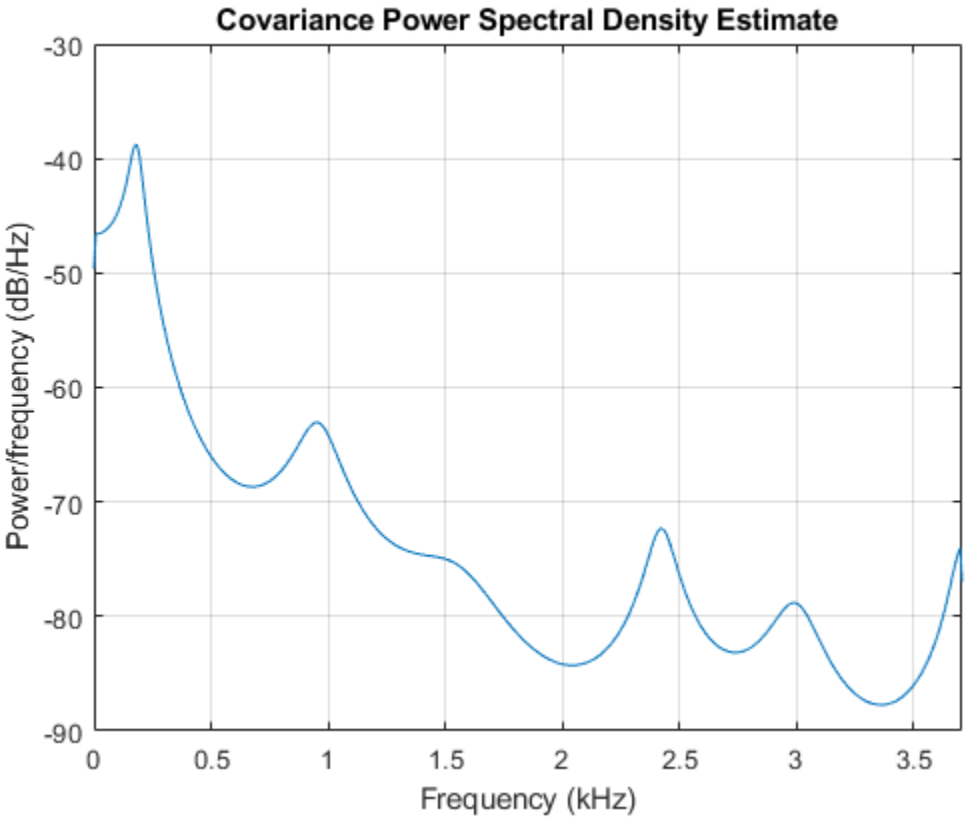



## Covariance and Modified Covariance Methods

The covariance method for AR spectral estimation is based on minimizing the forward prediction error. The modified covariance method is based on minimizing the forward and backward prediction errors. The toolbox functions `pcov` and `pmcov` implement the respective methods.

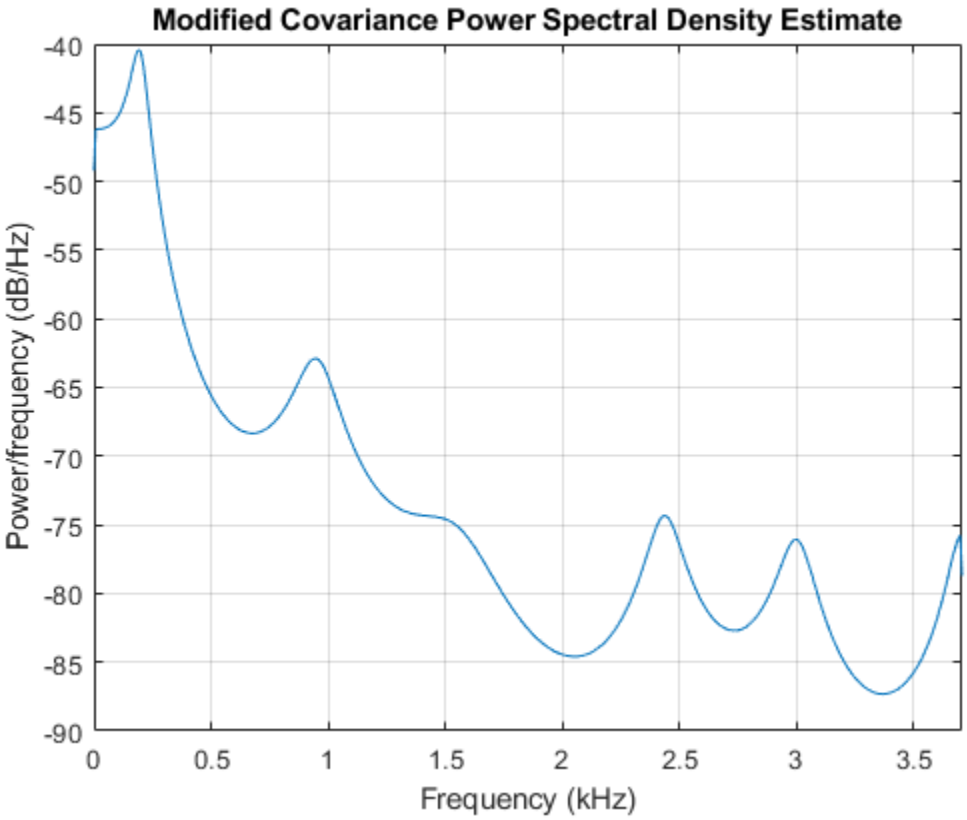
Compare the spectrum of a speech signal generated by both the covariance method and the modified covariance method. First compute and plot the covariance method estimate.

```
load mtlb
pcov(mtlb(1:64),14,1024,Fs)
```




The modified covariance method estimate is nearly identical, even for a short signal length.

```
pmcov(mtlb(1:64),14,1024,Fs)
```




## See Also

**Functions**
`pburg` | `pcov` | `pmcov` | `pyulear`

# MUSIC and Eigenvector Analysis Methods

The `pmusic` and `peig` functions provide two related spectral analysis methods:

- `pmusic` provides the multiple signal classification (MUSIC) method developed by Schmidt.
- `peig` provides the eigenvector (EV) method developed by Johnson.

Both of these methods are frequency estimator techniques based on eigenanalysis of the autocorrelation matrix. This type of spectral analysis categorizes the information in a correlation or data matrix, assigning information to either a signal subspace or a noise subspace.

## Eigenanalysis Overview

Consider a number of complex sinusoids embedded in white noise. You can write the autocorrelation matrix $R$ for this system as the sum of the signal autocorrelation matrix ($S$) and the noise autocorrelation matrix ($W$): $R = S + W$. There is a close relationship between the eigenvectors of the signal autocorrelation matrix and the signal and noise subspaces. The eigenvectors $v$ of $S$ span the same signal subspace as the signal vectors. If the system contains $M$ complex sinusoids and the order of the autocorrelation matrix is $p$, eigenvectors $v_{M+1}$ through $v_{p+1}$ span the noise subspace of the autocorrelation matrix.

## Frequency Estimator Functions

To generate their frequency estimates, eigenanalysis methods calculate functions of the vectors in the signal and noise subspaces. Both the MUSIC and EV techniques choose a function that goes to infinity (denominator goes to zero) at one of the sinusoidal frequencies in the input signal. Using digital technology, the resulting estimate has sharp peaks at the frequencies of interest; this means that there might not be infinity values in the vectors.

The MUSIC estimate is given by the formula

$$\widehat{P}_{\mathrm{MUSIC}}(f) = \frac{1}{\sum_{k=p+1}^{M} \left| v_k^H e(f) \right|^2},$$

where the $v_k$ are the eigenvectors of the noise subspace and $e(f)$ is a vector of complex sinusoids:

$$e(f) = [1 \;\; e^{j2\pi f} \;\; e^{j4\pi f} \;\; \ldots \;\; e^{j2(M-1)\pi f}]^T .$$

Here $v$ represents the eigenvectors of the input signal's correlation matrix; $v_k$ is the $k$th eigenvector. $H$ is the conjugate transpose operator. The eigenvectors used in the sum correspond to the smallest eigenvalues and span the noise subspace ($p$ is the size of the signal subspace).

The expression ${v_k}^H e(f)$ is equivalent to a Fourier transform (the vector $e(f)$ consists of complex exponentials). This form is useful for numeric computation because the FFT can be computed for each $v_k$ and then the squared magnitudes can be summed.

The EV method weights the summation by the eigenvalues of the correlation matrix:

$$\widehat{P}_{\mathrm{EV}}(f) = \frac{1}{\sum_{k=p+1}^{M} \frac{1}{\lambda_k} \left| v_k^H e(f) \right|^2} .$$

The `pmusic` and `peig` functions interpret their first input either as a signal matrix or as a correlation matrix (if the `'corr'` input flag is set). In the former case, the singular value decomposition of the signal matrix is used to determine the signal and noise subspaces. In the latter case, the eigenvalue decomposition of the correlation matrix is used to determine the signal and noise subspaces.

## See Also

**Functions**
`peig` | `pmusic`

# Selected Bibliography

[1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

[2] Kay, Steven M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[3] Marple, S. Lawrence *Digital Spectral Analysis*. Englewood Cliffs, NJ: Prentice Hall, 1987.

[4] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1996.

[5] Percival, D. B., and A. T. Walden. *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques*. Cambridge: Cambridge University Press, 1993.

[6] Proakis, John G., and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1996.

[7] Stoica, Petre, and Randolph Moses. *Spectral Analysis of Signals*. Upper Saddle River, NJ: Prentice Hall, 1997.

[8] Welch, Peter D. “The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms.” *IEEE Trans. Audio Electroacoust.*. Vol. AU-15, 1967, pp. 70–73.

# Special Topics


- “Windows” on page 8-2
- “Get Started with Window Designer” on page 8-6
- “Generalized Cosine Windows” on page 8-9
- “Kaiser Window” on page 8-11
- “Chebyshev Window” on page 8-17
- “Parametric Modeling” on page 8-18
- “Resampling” on page 8-24
- “Cepstrum Analysis” on page 8-26
- “FFT-Based Time-Frequency Analysis” on page 8-29
- “Cross-Spectrogram of Complex Signals” on page 8-31
- “Median Filtering” on page 8-33
- “Communications Applications” on page 8-34
- “Deconvolution” on page 8-38
- “Chirp Z-Transform” on page 8-39
- “Discrete Cosine Transform” on page 8-41
- “Hilbert Transform” on page 8-44
- “Walsh-Hadamard Transform” on page 8-46
- “Walsh-Hadamard Transform for Spectral Analysis and Compression of ECG Signals” on page 8-48
- “Eliminate Outliers Using Hampel Identifier” on page 8-51
- “Selected Bibliography” on page 8-53

# Windows

| In this section... |
| --- |
| “Why Use Windows?” on page 8-2<br>“Available Window Functions” on page 8-2<br>“Graphical User Interface Tools” on page 8-2<br>“Basic Shapes” on page 8-3 |

## Why Use Windows?

In both digital filter design and spectral estimation, the choice of a windowing function can play an important role in determining the quality of overall results. The main role of the window is to damp out the effects of the Gibbs phenomenon that results from truncation of an infinite series.

## Available Window Functions

| Window | Function |
| --- | --- |
| Bartlett-Hann window | `barthannwin` |
| Bartlett window | `bartlett` |
| Blackman window | `blackman` |
| Blackman-Harris window | `blackmanharris` |
| Bohman window | `bohmanwin` |
| Chebyshev window | `chebwin` |
| Flat Top window | `flattopwin` |
| Gaussian window | `gausswin` |
| Hamming window | `hamming` |
| Hann window | `hann` |
| Kaiser window | `kaiser` |
| Nuttall's Blackman-Harris window | `nuttallwin` |
| Parzen (de la Vallée-Poussin) window | `parzenwin` |
| Rectangular window | `rectwin` |
| Tapered cosine window | `tukeywin` |
| Triangular window | `triang` |

## Graphical User Interface Tools

Two graphical user interface tools are provided for working with windows in the Signal Processing Toolbox product:

- **Window Designer** app
- Window Visualization Tool (**WVTool**)

Refer to the reference pages for detailed information.

## Basic Shapes

The basic window is the rectangular window, a vector of ones of the appropriate length. A rectangular window of length 50 is

```
n = 50;
w = rectwin(n);
```

Signal Processing Toolbox stores windows in column vectors by convention, so an equivalent expression is

```
w = ones(50,1);
```

To use the **Window Designer** app to create this window, type

```
windowDesigner
```

The app opens with a default Hamming window. To visualize the rectangular window, set **Type = Rectangular** and **Length = 50** in the Current Window Information panel and then press **Apply**.

The *Bartlett* (or triangular) *window* is the convolution of two rectangular windows. The functions `bartlett` and `triang` compute similar triangular windows, with three important differences. The `bartlett` function always returns a window with two zeros on the ends of the sequence, so that for `n` odd, the center section of `bartlett(n+2)` is equivalent to `triang(n)`:

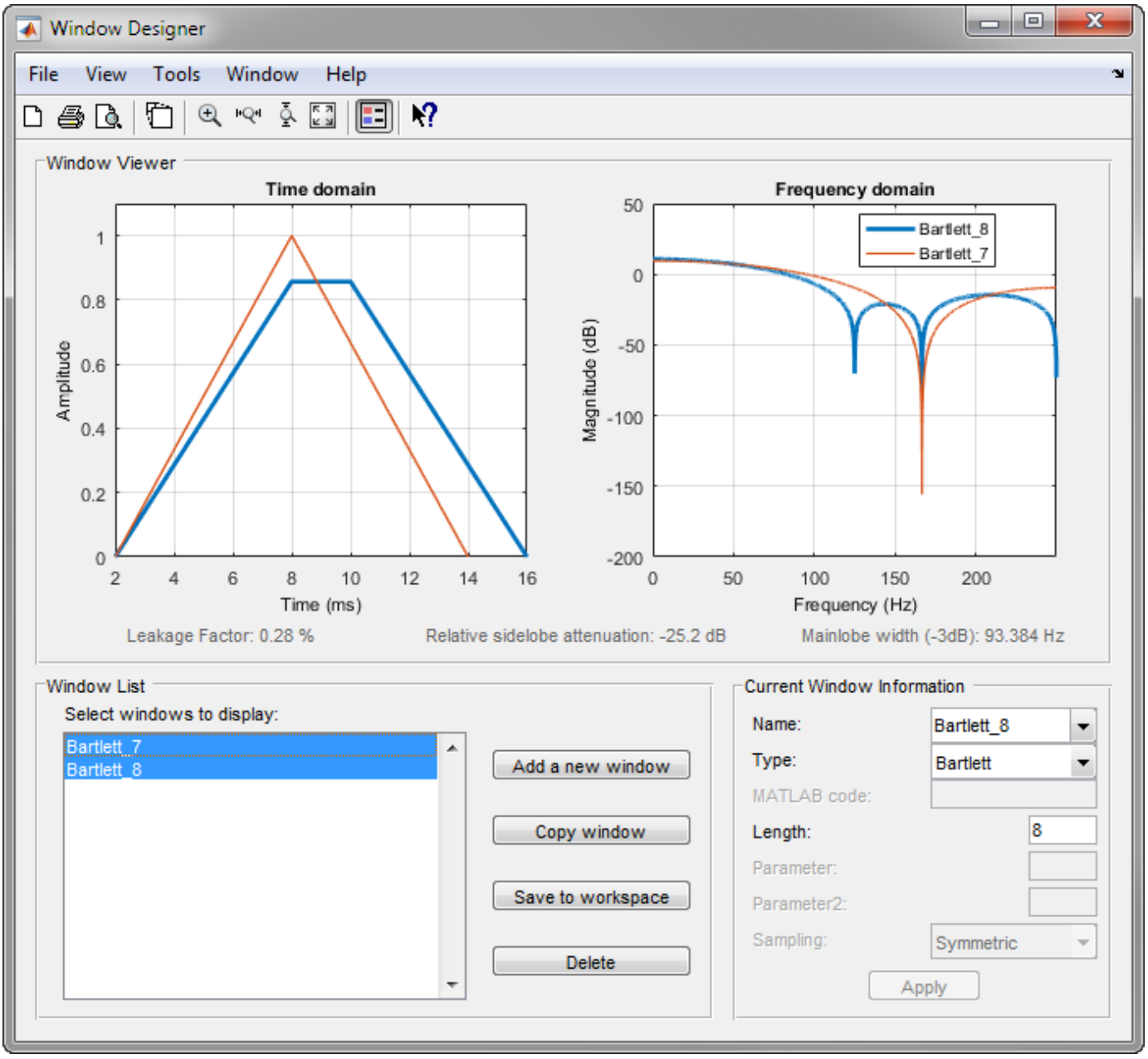
```
Bartlett = bartlett(7);
isequal(Bartlett(2:end-1),triang(5))
```

```
ans =
     1
```

For `n` even, `bartlett` is still the convolution of two rectangular sequences. There is no standard definition for the triangular window for `n` even; the slopes of the line segments of the `triang` result are slightly steeper than those of `bartlett` in this case:

```
w = bartlett(8);
[w(2:7) triang(6)]
```

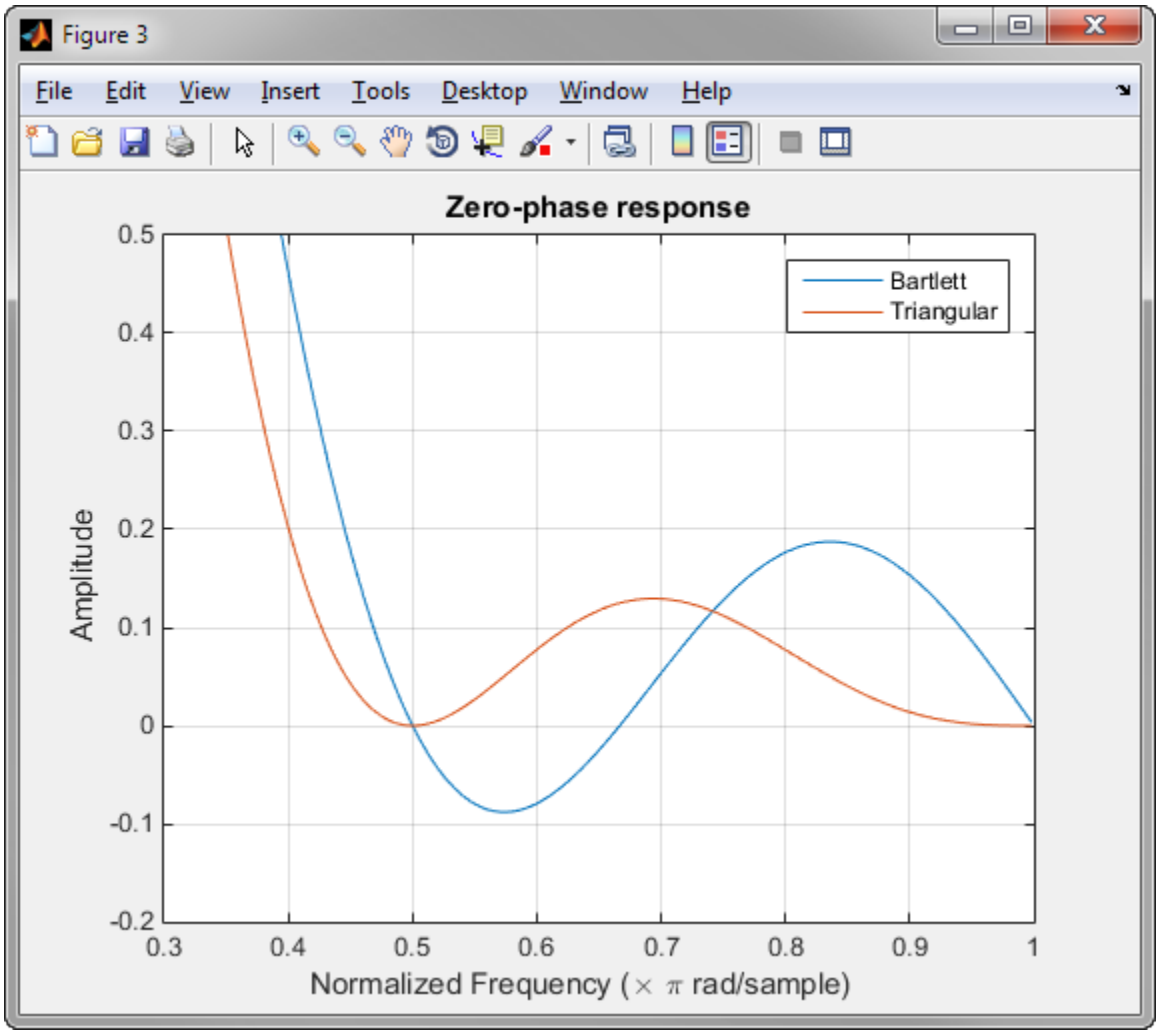
You can see the difference between odd and even Bartlett windows in **Window Designer**.




The final difference between the Bartlett and triangular windows is evident in the Fourier transforms of these functions. The Fourier transform of a Bartlett window is negative for n even. The Fourier transform of a triangular window, however, is always nonnegative.

The following figure, which plots the zero-phase responses of 8-point Bartlett and Triangular windows, illustrates the difference.

```
zerophase(bartlett(8))
hold on
zerophase(triang(8))
legend('Bartlett','Triangular')
axis([0.3 1 -0.2 0.5])
```




This difference can be important when choosing a window for some spectral estimation techniques, such as the Blackman-Tukey method. Blackman-Tukey forms the spectral estimate by calculating the Fourier transform of the autocorrelation sequence. The resulting estimate might be negative at some frequencies if the window's Fourier transform is negative.

## See Also

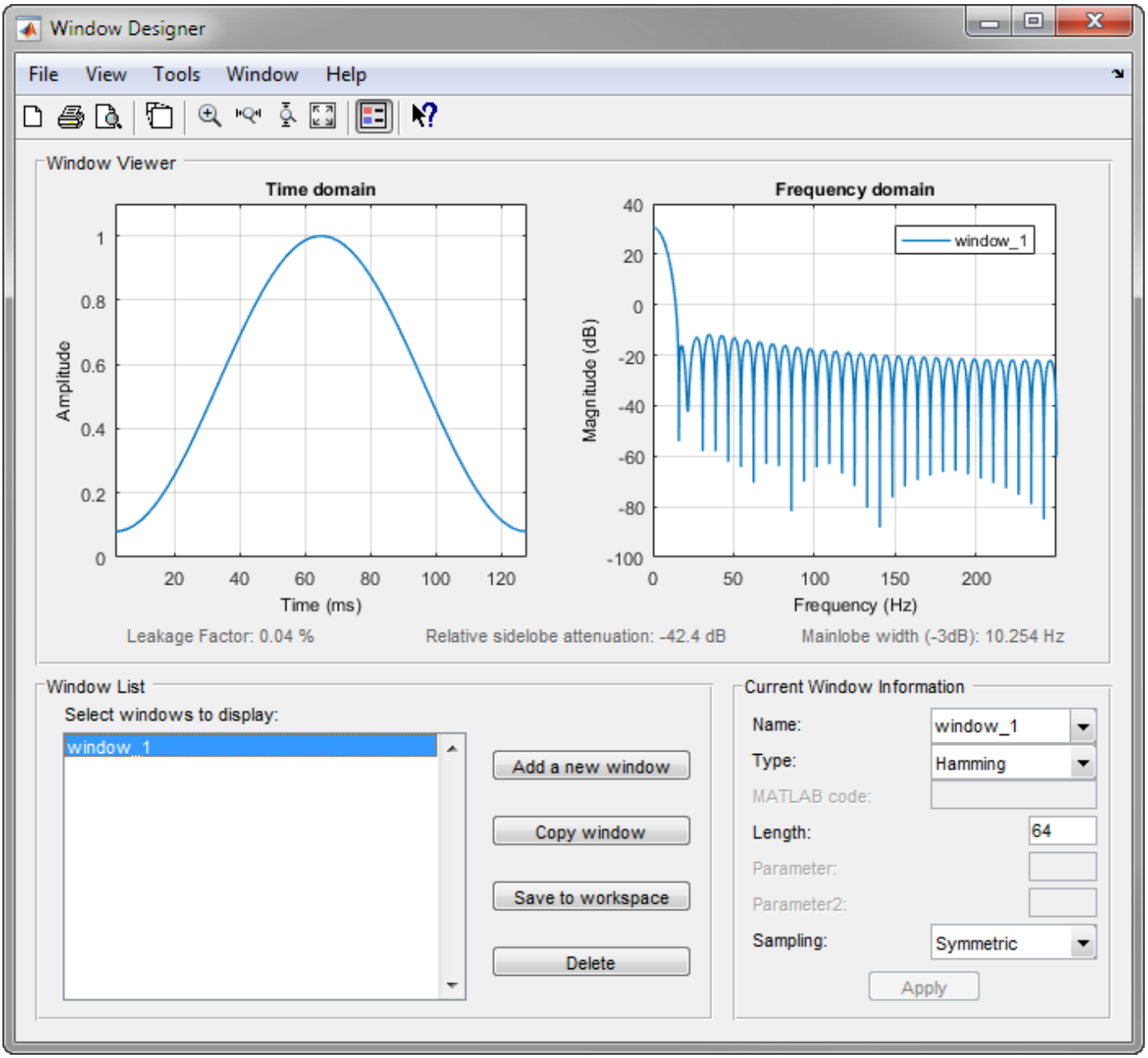
**Apps**
**Window Designer**

**Functions**
**WVTool** | `bartlett` | `rectwin` | `triang`

# Get Started with Window Designer

Typing `windowDesigner` at the command line opens the **Window Designer** app for designing and analyzing spectral windows. The app opens with a default 64-point Hamming window.

**Note** A related tool, **WVTool**, is available for displaying, annotating, or printing windows.




The app has three panels:

- Window Viewer displays the time domain and frequency domain representations of the selected window(s). The currently active window is shown in bold. Three window measurements are shown below the plots.
  - Leakage factor — ratio of power in the sidelobes to the total window power
  - Relative sidelobe attenuation — difference in height from the mainlobe peak to the highest sidelobe peak
  - Mainlobe width (–3dB) — width of the mainlobe at 3 dB below the mainlobe peak
- Window List lists the windows available for display in the Window Viewer. Highlight one or more windows to display them. The Window List buttons are:
  - **Add a new window** — Adds a default Hamming window with length 64 and symmetric sampling. You can change the information for this window by applying changes made in the **Current Window Information** panel.
  - **Copy window** — Copies the selected window(s).
  - **Save to workspace** — Saves the selected window(s) as vector(s) to the MATLAB workspace. The name of the window is used as the vector name.
  - **Delete** — Removes the selected window(s) from the window list.
- *Current Window Information* displays information about the currently active window. The active window name is shown in the `Name` field. To make another window active, select its name from the **Name** menu.

## Window Parameters

Each window is defined by the parameters in the Current Window Information panel. You can change the current window's characteristics by changing its parameters and clicking **Apply**. The parameters of the current window are

- **Name** — Name of the window. The name is used for the legend in the Window Viewer, in the Window List, and for the vector saved to the workspace. You can either select a name from the menu or type the desired name in the edit box.
- **Type** — Algorithm for the window. Select the type from the menu. All Signal Processing Toolbox windows are available.
- **MATLAB code** — Any valid MATLAB expression that returns a vector defining the window if `Type = User Defined`.
- **Length** — Number of samples.
- **Parameter** — Additional parameter for windows that require it, such as Chebyshev, which requires you to specify the sidelobe attenuation. Note that the title “Parameter” changes to the appropriate parameter name.
- **Sampling** — Type of sampling to use for generalized cosine windows (Hamming, Hann, and Blackman) — `Periodic` or `Symmetric`. `Periodic` computes a length `n+1` window and returns the first `n` points, and `Symmetric` computes and returns the `n` points specified in `Length`.

## Window Designer Menus

In addition to the usual menu items, **Window Designer** contains these menu commands:

**File** menu:

- **Export** — Exports window coefficient vectors to the MATLAB workspace, a text file, or a MAT-file.

  In the **Window List** in, highlight the window(s) you want to export and then select **File > Export**. For exporting to the workspace or a MAT-file, specify the variable name for each set of window coefficients. To overwrite variables in the workspace, select the Overwrite variables check box.
- **Full View Analysis** — Copies the windows shown in both plots to a separate **WVTool** figure window. This is useful for printing and annotating. This option is also available with the Full View Analysis toolbar button.

**View** menu:

- **Time domain** — Select to show the time domain plot in the Window Viewer panel.
- **Frequency domain** — Select to show the frequency domain plot in the Window Viewer panel.
- **Legend** — Toggles the window name legend on and off. This option is also available with the Legend toolbar button.
- **Analysis Parameters** — Controls the response plot parameters, including number of points, range, *x*- and *y*-axis units, sampling frequency, and normalized magnitude.

  You can also access the Analysis Parameters by right-clicking the *x*-axis label of a plot in the Window Viewer panel. The *x*-axis units for the time domain plot depend on the selected `Sampling Frequency` units.

| Frequency Domain | Time Domain |
|---|---|
| Hz | s |
| kHz | ms |
| MHz | µs |
| GHz | ns |

**Tools** menu:

- **Zoom In** — Zooms in along both *x*- and *y*-axes.
- **Zoom X** — Zooms in along the *x*-axis only. Drag the mouse in the *x* direction to select the zoom area.
- **Zoom Y** — Zooms in along the *y*-axis only. Drag the mouse in the *y* direction to select the zoom area.
- **Full View** — Returns to full view.

## See Also

**Functions**
**WVTool**

# Generalized Cosine Windows

Blackman, flat top, Hamming, Hann, and rectangular windows are all special cases of the *generalized cosine window*. These windows are combinations of sinusoidal sequences with frequencies that are multiples of $2\pi/(N-1)$, where $N$ is the window length. One special case is the Blackman window:

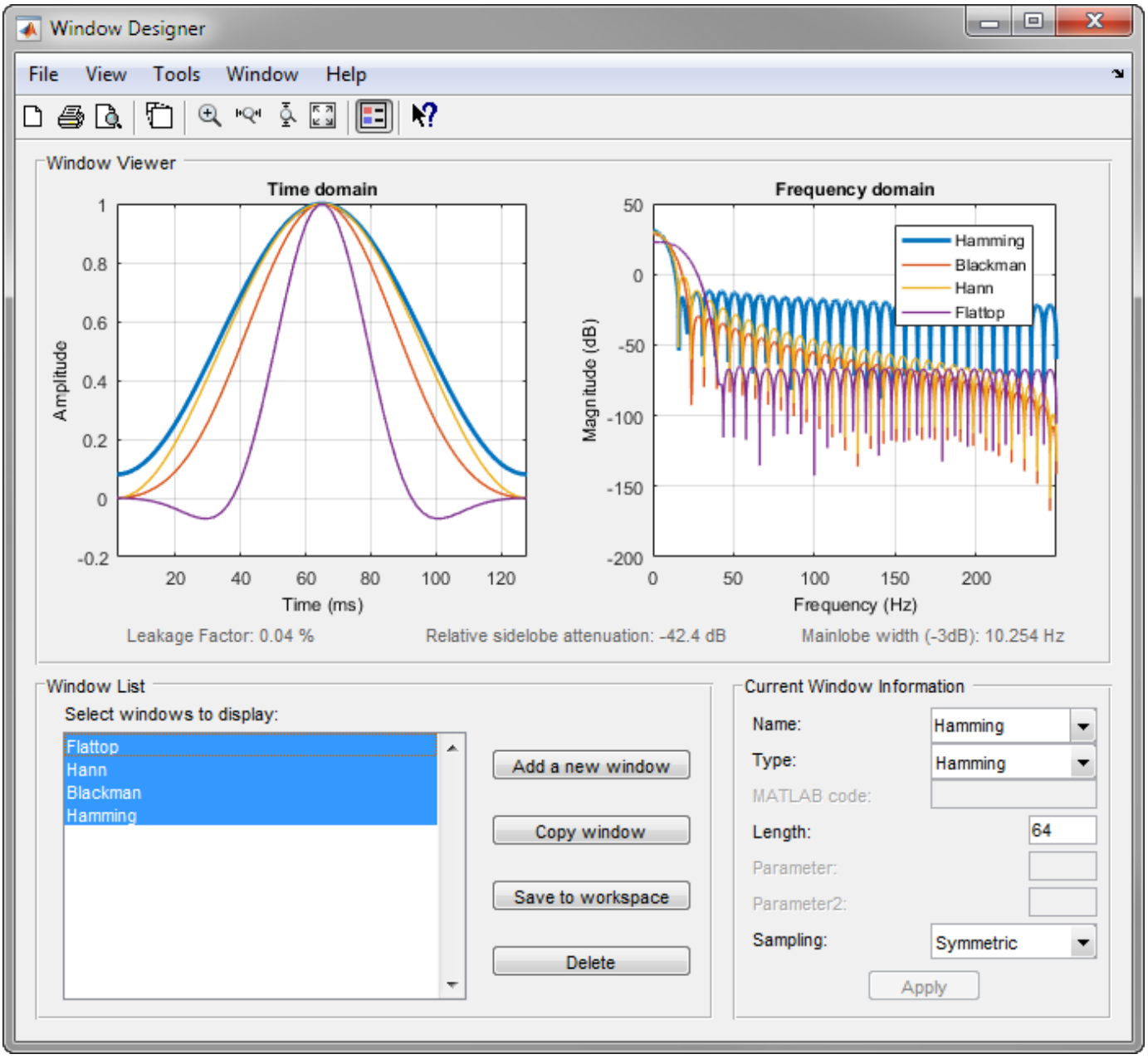
```
N = 128;
A = 0.42;
B = 0.5;
C = 0.08;
ind = (0:N-1)'*2*pi/(N-1);
w = A - B*cos(ind) + C*cos(2*ind);
```

Changing the values of the constants `A`, `B`, and `C` in the previous expression generates different generalized cosine windows like the Hamming and Hann windows. Adding additional cosine terms of higher frequency generates the flat top window. The concept behind these windows is that by summing the individual terms to form the window, the low frequency peaks in the frequency domain combine in such a way as to decrease sidelobe height. This has the side effect of increasing the mainlobe width.

The Hamming and Hann windows are two-term generalized cosine windows, given by `A = 0.54`, `B = 0.46` for the Hamming and `A = 0.5`, `B = 0.5` for the Hann.

Note that the definition of the generalized cosine window shown in the earlier MATLAB code yields zeros at samples 1 and `n` for `A = 0.5` and `B = 0.5`.

This **Window Designer** screen shot compares Blackman, Hamming, Hann, and Flat Top windows.




## See Also

**Apps**
**Window Designer**

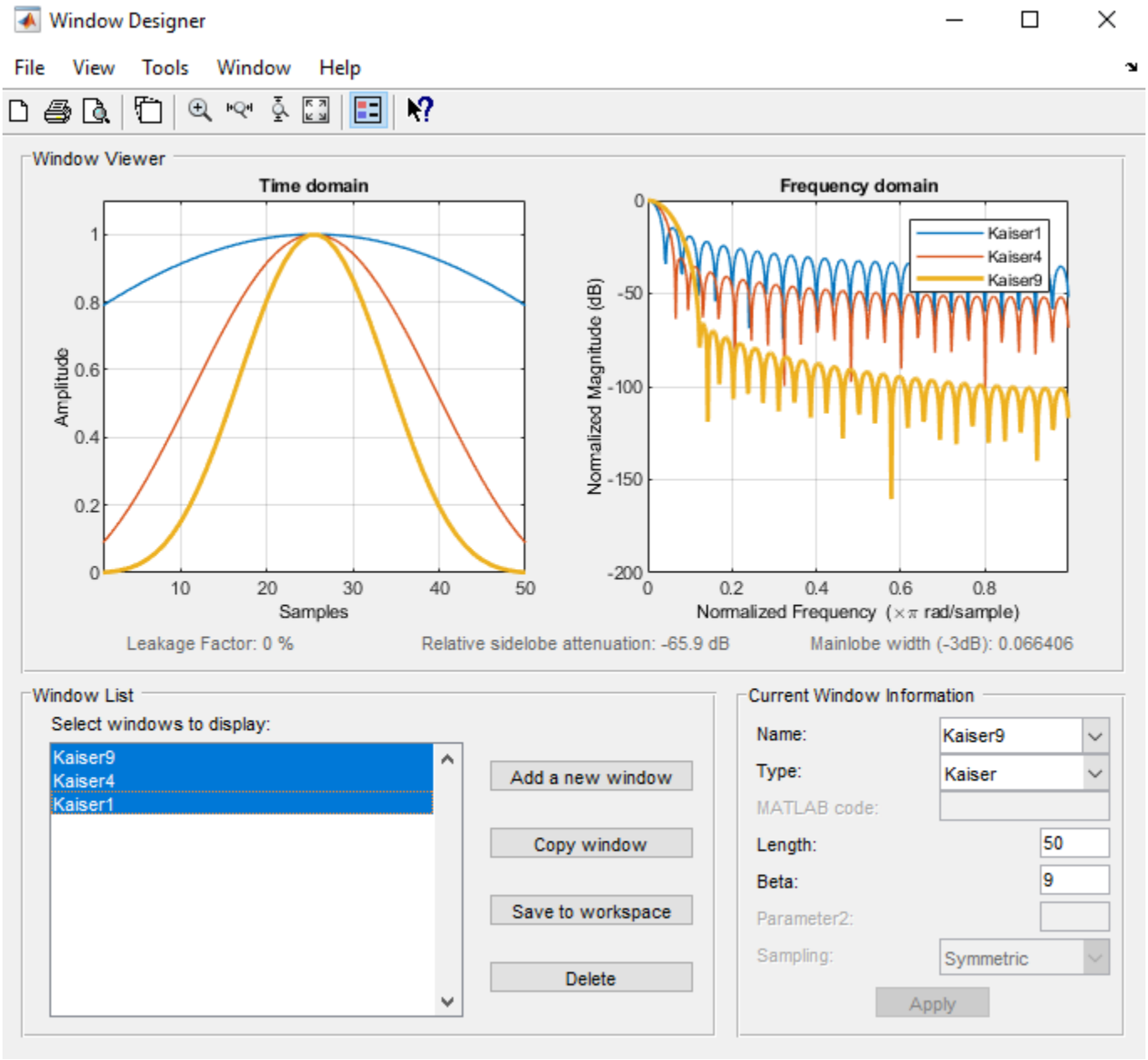
**Functions**
**WVTool** | `blackman` | `flattopwin` | `hamming` | `hann`

# Kaiser Window

The Kaiser window is an approximation to the prolate spheroidal window, for which the ratio of the mainlobe energy to the sidelobe energy is maximized. For a Kaiser window of a particular length, the parameter $\beta$ controls the relative sidelobe attenuation. For a given $\beta$, the relative sidelobe attenuation is fixed with respect to window length. The statement `kaiser(n,beta)` computes a length `n` Kaiser window with parameter `beta`.

As $\beta$ increases, the relative sidelobe attenuation decreases and the mainlobe width increases. This screen shot shows how the relative sidelobe attenuation stays approximately the same for a fixed $\beta$ parameter as the length is varied.

Examples of Kaiser windows with length 50 and $\beta$ parameters of 1, 4, and 9 are shown in this example.




To create these Kaiser windows using the MATLAB command line, type the following:

```
n = 50;
w1 = kaiser(n,1);
w2 = kaiser(n,4);
w3 = kaiser(n,9);
[W1,f] = freqz(w1/sum(w1),1,512,2);
[W2,f] = freqz(w2/sum(w2),1,512,2);
[W3,f] = freqz(w3/sum(w3),1,512,2);
plot(f,20*log10(abs([W1 W2 W3])))
grid
legend('\beta = 1','\beta = 4','\beta = 9')
```

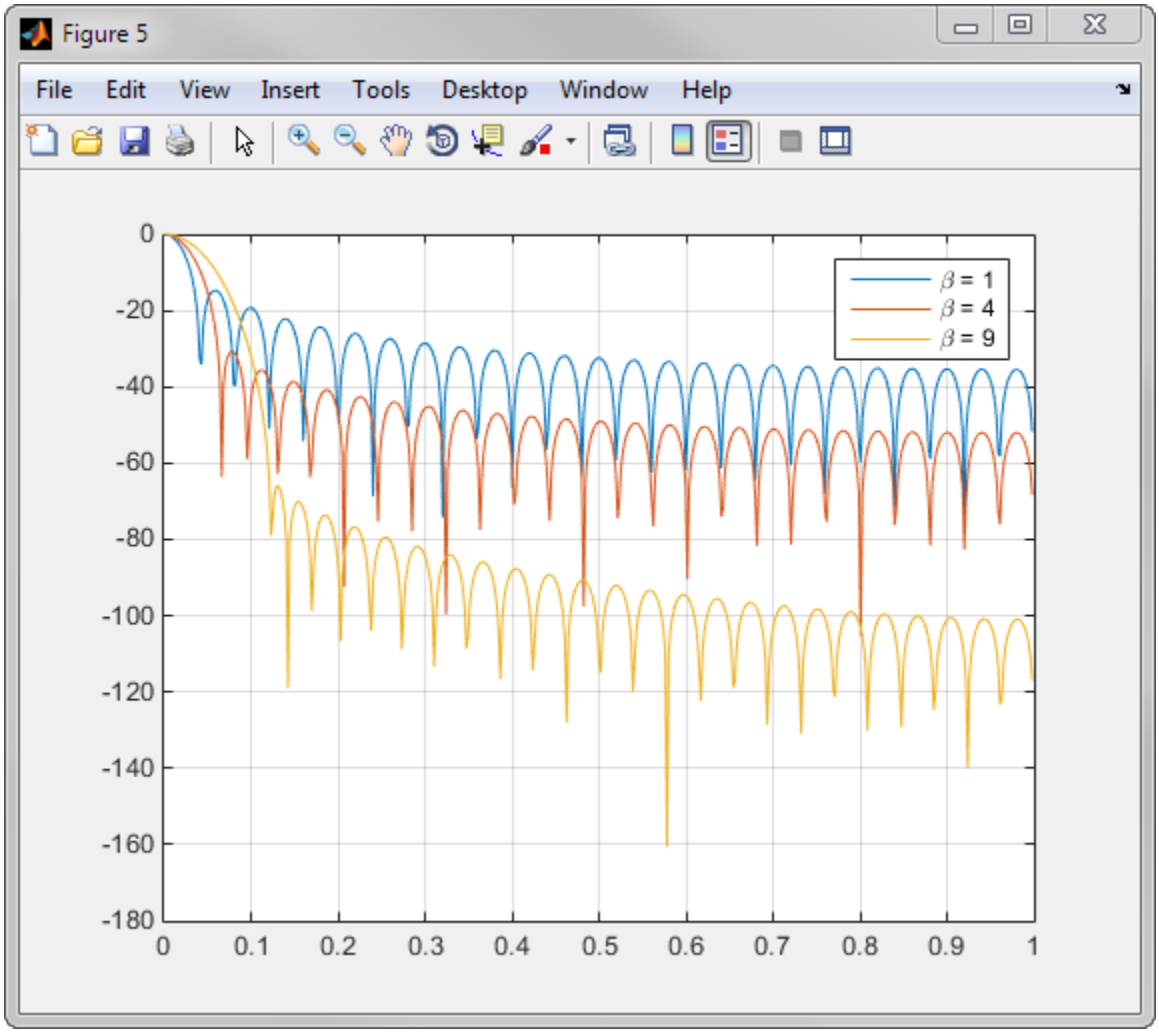
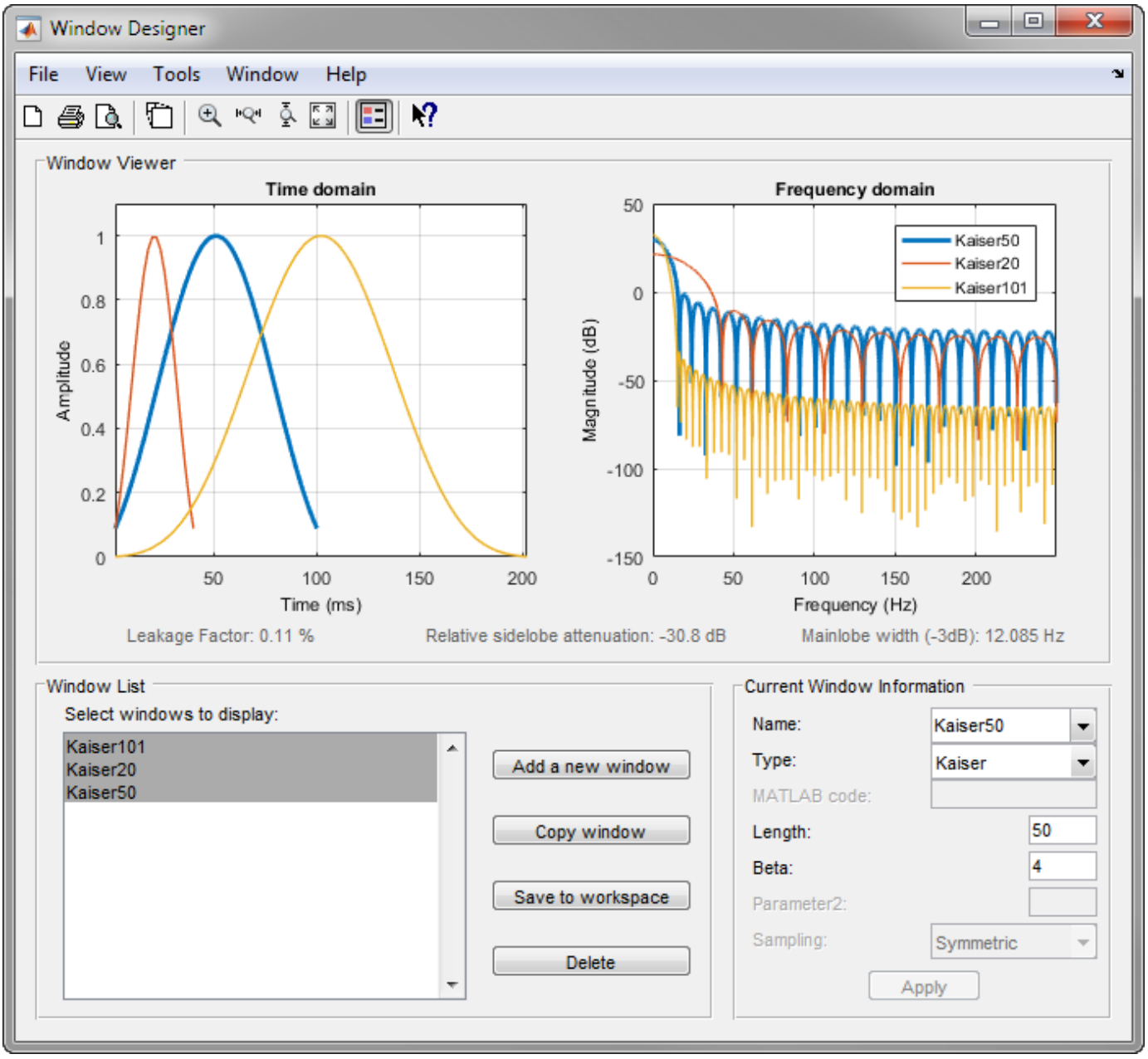
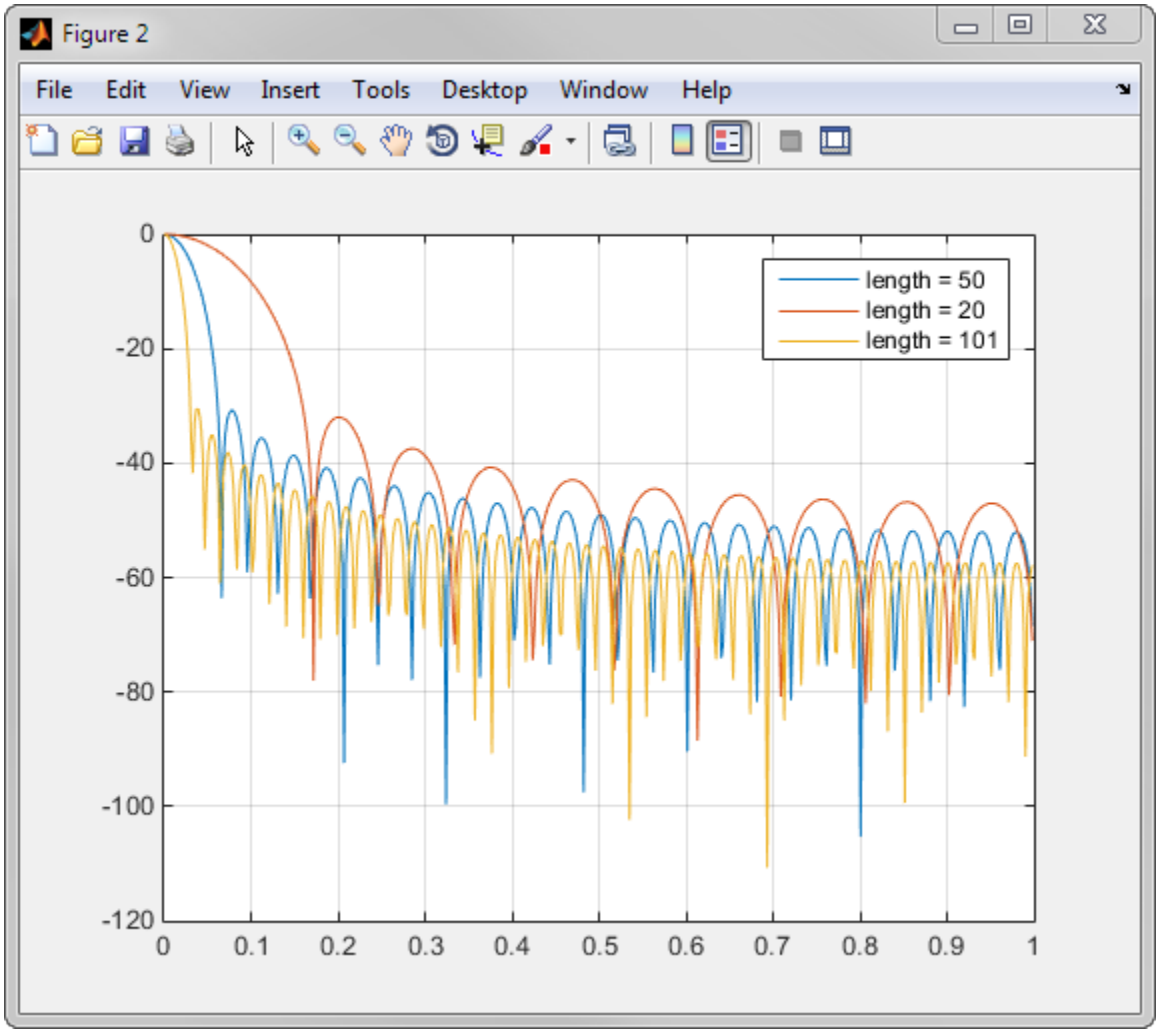


Figure 5
File Edit View Insert Tools Desktop Window Help
0
-20
-40
-60
-80
-100
-120
-140
-160
-180
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
β = 1
β = 4
β = 9




To create these Kaiser windows using the MATLAB command line, type the following:

```
w1 = kaiser(50,4);
w2 = kaiser(20,4);
w3 = kaiser(101,4);
[W1,f] = freqz(w1/sum(w1),1,512,2);
[W2,f] = freqz(w2/sum(w2),1,512,2);
[W3,f] = freqz(w3/sum(w3),1,512,2);
plot(f,20*log10(abs([W1 W2 W3])))
grid
legend('length = 50','length = 20','length = 101')
```




## Kaiser Windows in FIR Design

There are two design formulas that can help you design FIR filters to meet a set of filter specifications using a Kaiser window. To achieve a relative sidelobe attenuation of –$\alpha$ dB, the $\beta$ (`beta`) parameter is

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50, \\ 0.5842(\alpha - 21)^{0.4} + 0.7886(\alpha - 21), & 50 \geq \alpha \geq 21, \\ 0, & \alpha < 21. \end{cases}$$

For a transition width of $\Delta\omega$ rad/sample, use the length

$$n = \frac{\alpha - 8}{2.285\Delta\omega} + 1.$$

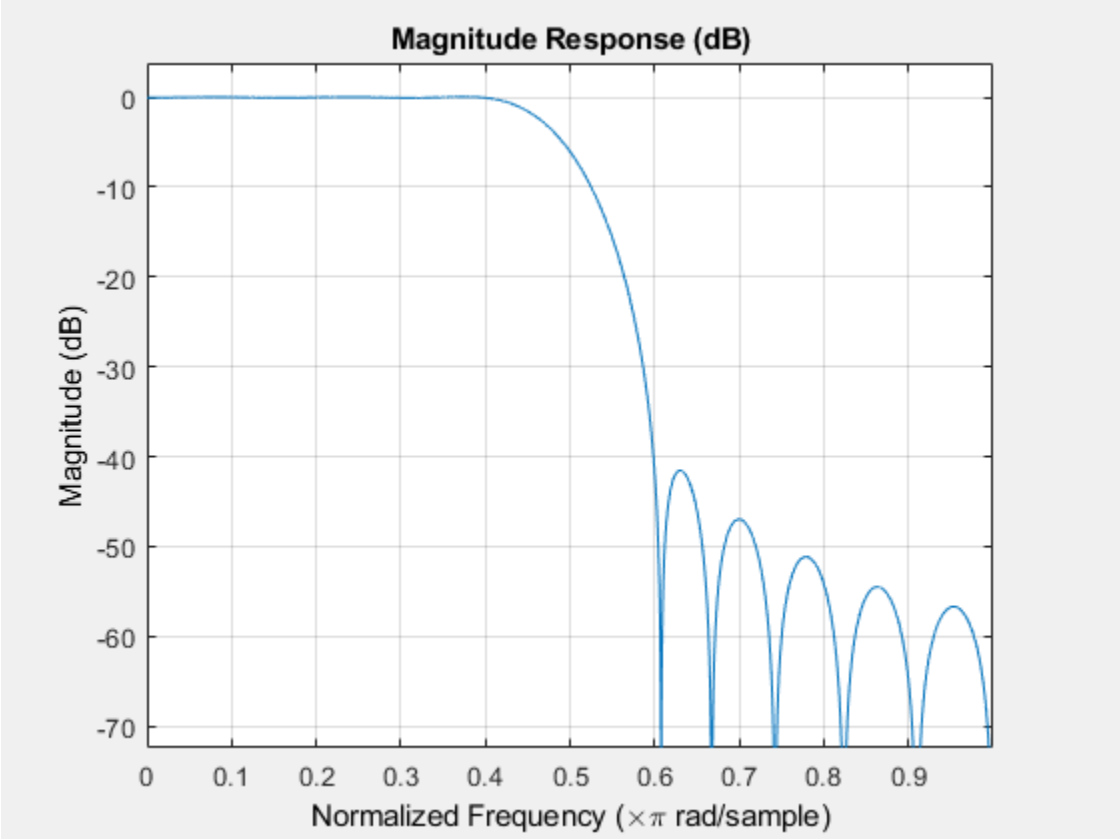
Filters designed using these heuristics will meet the specifications approximately, but you should verify this. To design a lowpass filter with cutoff frequency $0.5\pi$ rad/sample, transition width 0.2 $\pi$ rad/sample, and 40 dB of attenuation in the stopband, try

```
[n,wn,beta] = kaiserord([0.4 0.6]*pi,[1 0],[0.01 0.01],2*pi);
h = fir1(n,wn,kaiser(n+1,beta),'noscale');
```

The `kaiserord` function estimates the filter order, cutoff frequency, and Kaiser window beta parameter needed to meet a given set of frequency domain specifications.

The ripple in the passband is roughly the same as the ripple in the stopband. As you can see from the frequency response, this filter nearly meets the specifications:

```
fvtool(h,1)
```




## See Also

**Apps**
**Window Designer**

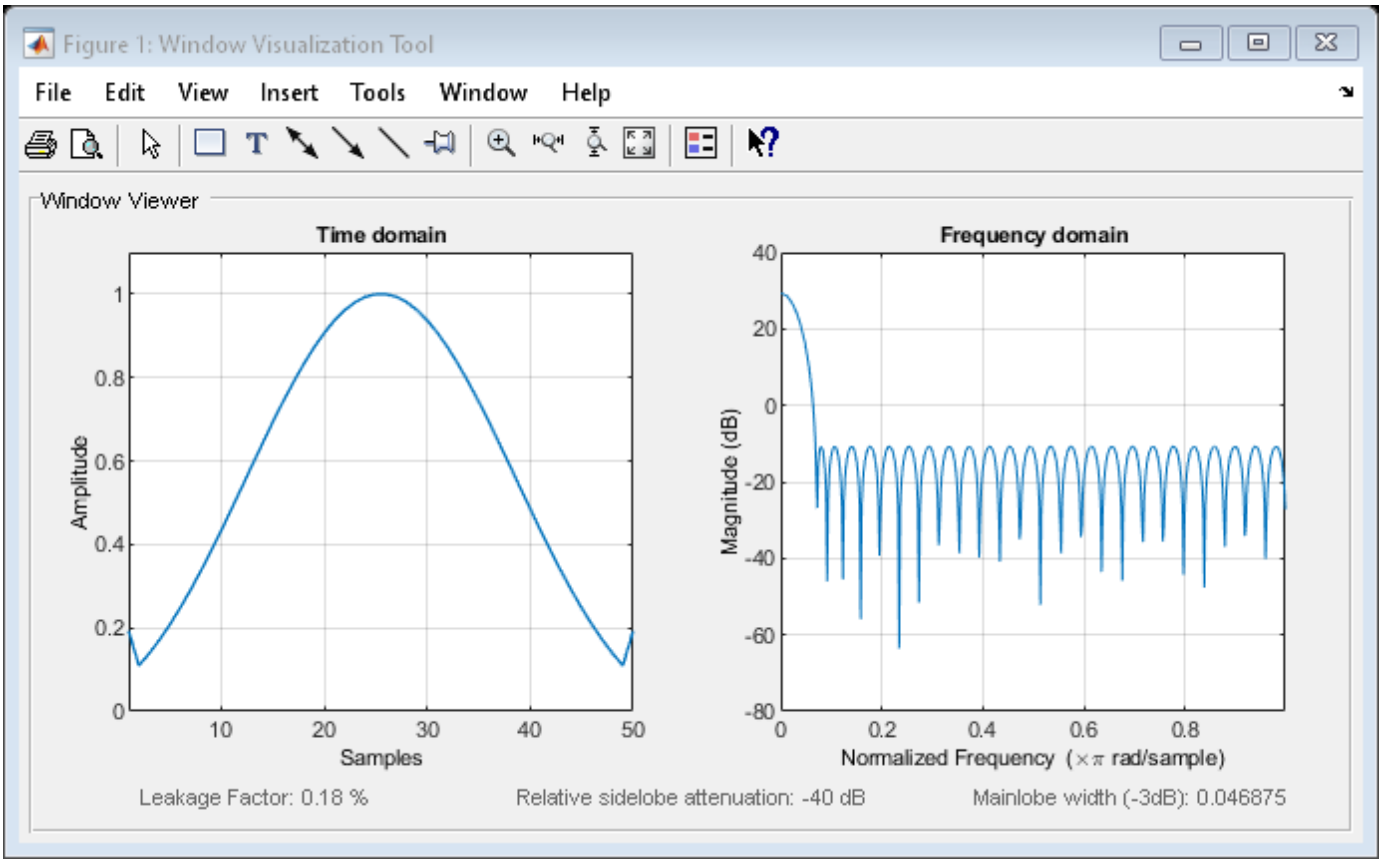
**Functions**
**WVTool** | `freqz` | `kaiser` | `kaiserord`

# Chebyshev Window

The Chebyshev window minimizes the mainlobe width, given a particular sidelobe height. It is characterized by an equiripple behavior. Its sidelobes all have the same height.

Generate and display a 50-point Chebyshev window with a sidelobe attenuation of 40 dB.

```
w = chebwin(50,40);
wvtool(w)
```




As shown in the time-domain plot, the Chebyshev window has large spikes at its outer samples.

## See Also

**Apps**
**Window Designer**

**Functions**
**WVTool** | `chebwin`

# Parametric Modeling

| In this section... |
| --- |
| “What is Parametric Modeling” on page 8-18 |
| “Available Parametric Modeling Functions” on page 8-18 |
| “Time-Domain Based Modeling” on page 8-19 |
| “Frequency-Domain Based Modeling” on page 8-21 |

## What is Parametric Modeling

Parametric modeling techniques find the parameters for a mathematical model describing a signal, system, or process. These techniques use known information about the system to determine the model. Applications for parametric modeling include speech and music synthesis, data compression, high-resolution spectral estimation, communications, manufacturing, and simulation.

## Available Parametric Modeling Functions

The toolbox parametric modeling functions operate with the rational transfer function model. Given appropriate information about an unknown system (impulse or frequency response data, or input and output sequences), these functions find the coefficients of a linear system that models the system.

One important application of the parametric modeling functions is in the design of filters that have a prescribed time or frequency response.

Here is a summary of the parametric modeling functions in this toolbox.

| Domain | Functions | Description |
| --- | --- | --- |
| Time | `arburg` | Generate all-pole filter coefficients that model an input data sequence using the Levinson-Durbin algorithm. |
| | `arcov` | Generate all-pole filter coefficients that model an input data sequence by minimizing the forward prediction error. |
| | `armcov` | Generate all-pole filter coefficients that model an input data sequence by minimizing the forward and backward prediction errors. |
| | `aryule` | Generate all-pole filter coefficients that model an input data sequence using an estimate of the autocorrelation function. |
| | `lpc`, `levinson` | Linear Predictive Coding. Generate all-pole recursive filter whose impulse response matches a given sequence. |
| | `prony` | Generate IIR filter whose impulse response matches a given sequence. |
| | `stmcb` | Find IIR filter whose output, given a specified input sequence, matches a given output sequence. |
| Frequency | `invfreqz`, `invfreqs` | Generate digital or analog filter coefficients given complex frequency response data. |

## Time-Domain Based Modeling

The `lpc`, `prony`, and `stmcb` functions find the coefficients of a digital rational transfer function that approximates a given time-domain impulse response. The algorithms differ in complexity and accuracy of the resulting model.

### Linear Prediction

Linear prediction modeling assumes that each output sample of a signal, `x(k)`, is a linear combination of the past `n` outputs (that is, it can be linearly predicted from these outputs), and that the coefficients are constant from sample to sample:

$$x(k) = -a(2)x(k-1) - a(3)x(k-2) - \cdots - a(n+1)x(k-n)\,.$$

An `n`th-order all-pole model of a signal `x` is

```
a = lpc(x,n)
```

To illustrate `lpc`, create a sample signal that is the impulse response of an all-pole filter with additive white noise:

```
x = impz(1,[1 0.1 0.1 0.1 0.1],10) + randn(10,1)/10;
```

The coefficients for a fourth-order all-pole filter that models the system are

```
a = lpc(x,4)
```

`lpc` first calls `xcorr` to find a biased estimate of the correlation function of `x`, and then uses the Levinson-Durbin recursion, implemented in the `levinson` function, to find the model coefficients `a`. The Levinson-Durbin recursion is a fast algorithm for solving a system of symmetric Toeplitz linear equations. `lpc`'s entire algorithm for `n = 4` is

```
r = xcorr(x);
r(1:length(x)-1) = [];      % Remove corr. at negative lags
a = levinson(r,4)
```

You could form the linear prediction coefficients with other assumptions by passing a different correlation estimate to `levinson`, such as the biased correlation estimate:

```
r = xcorr(x,'biased');
r(1:length(x)-1) = [];      % Remove corr. at negative lags
a = levinson(r,4)
```

### Prony's Method (ARMA Modeling)

The `prony` function models a signal using a specified number of poles and zeros. Given a sequence `x` and numerator and denominator orders `n` and `m`, respectively, the statement

```
[b,a] = prony(x,n,m)
```

finds the numerator and denominator coefficients of an IIR filter whose impulse response approximates the sequence `x`.

The `prony` function implements the method described in [4] Parks and Burrus. This method uses a variation of the covariance method of AR modeling to find the denominator coefficients `a`, and then finds the numerator coefficients `b` for which the resulting filter's impulse response matches exactly the first `n + 1` samples of `x`. The filter is not necessarily stable, but it can potentially recover the

coefficients exactly if the data sequence is truly an autoregressive moving-average (ARMA) process of the correct order.

**Note** The functions `prony` and `stmcb` (described next) are more accurately described as ARX models in system identification terminology. ARMA modeling assumes noise only at the inputs, while ARX assumes an external input. `prony` and `stmcb` know the input signal: it is an impulse for `prony` and is arbitrary for `stmcb`.

A model for the test sequence `x` (from the earlier `lpc` example) using a third-order IIR filter is

```
[b,a] = prony(x,3,3)
```

The `impz` command shows how well this filter's impulse response matches the original sequence:

```
format long
[x impz(b,a,10)]
```

Notice that the first four samples match exactly. For an example of exact recovery, recover the coefficients of a Butterworth filter from its impulse response:

```
[b,a] = butter(4,.2);
h = impz(b,a,26);
[bb,aa] = prony(h,4,4);
```

Try this example; you'll see that `bb` and `aa` match the original filter coefficients to within a tolerance of $10^{-13}$.

### Steiglitz-McBride Method (ARMA Modeling)

The `stmcb` function determines the coefficients for the system $b(z)/a(z)$ given an approximate impulse response `x`, as well as the desired number of zeros and poles. This function identifies an unknown system based on both input and output sequences that describe the system's behavior, or just the impulse response of the system. In its default mode, `stmcb` works like `prony`.

```
[b,a] = stmcb(x,3,3)
```

`stmcb` also finds systems that match given input and output sequences:

```
y = filter(1,[1 1],x);     % Create an output signal.
[b,a] = stmcb(y,x,0,1)
```

In this example, `stmcb` correctly identifies the system used to create `y` from `x`.

The Steiglitz-McBride method is a fast iterative algorithm that solves for the numerator and denominator coefficients simultaneously in an attempt to minimize the signal error between the filter output and the given output signal. This algorithm usually converges rapidly, but might not converge if the model order is too large. As for `prony`, `stmcb`'s resulting filter is not necessarily stable due to its exact modeling approach.

`stmcb` provides control over several important algorithmic parameters; modify these parameters if you are having trouble modeling the data. To change the number of iterations from the default of five and provide an initial estimate for the denominator coefficients:

```
n = 10;             % Number of iterations
a = lpc(x,3);       % Initial estimates for denominator
[b,a] = stmcb(x,3,3,n,a);
```

The function uses an all-pole model created with `prony` as an initial estimate when you do not provide one of your own.

To compare the functions `lpc`, `prony`, and `stmcb`, compute the signal error in each case:

```
a1 = lpc(x,3);
[b2,a2] = prony(x,3,3);
[b3,a3] = stmcb(x,3,3);
[x-impz(1,a1,10)  x-impz(b2,a2,10)  x-impz(b3,a3,10)]
```

In comparing modeling capabilities for a given order IIR model, the last result shows that for this example, `stmcb` performs best, followed by `prony`, then `lpc`. This relative performance is typical of the modeling functions.

## Frequency-Domain Based Modeling

The `invfreqs` and `invfreqz` functions implement the inverse operations of `freqs` and `freqz`; they find an analog or digital transfer function of a specified order that matches a given complex frequency response. Though the following examples demonstrate `invfreqz`, the discussion also applies to `invfreqs`.

To recover the original filter coefficients from the frequency response of a simple digital filter:

```
[b,a] = butter(4,0.4)          % Design Butterworth lowpass
[h,w] = freqz(b,a,64);         % Compute frequency response
[b4,a4] = invfreqz(h,w,4,4)    % Model: n = 4, m = 4
```

The vector of frequencies `w` has the units in rad/sample, and the frequencies need not be equally spaced. `invfreqz` finds a filter of any order to fit the frequency data; a third-order example is

```
[b4,a4] = invfreqz(h,w,3,3)    % Find third-order IIR
```

Both `invfreqs` and `invfreqz` design filters with real coefficients; for a data point at positive frequency `f`, the functions fit the frequency response at both `f` and `-f`.

By default `invfreqz` uses an equation error method to identify the best model from the data. This finds $b$ and $a$ in

$$\min_{b,a} \sum_{k=1}^{n} wt(k)\left|h(k)A(w(k)) - B(w(k))\right|^2$$

by creating a system of linear equations and solving them with the MATLAB \ operator. Here $A(w(k))$ and $B(w(k))$ are the Fourier transforms of the polynomials `a` and `b` respectively at the frequency $w(k)$, and $n$ is the number of frequency points (the length of `h` and `w`). $wt(k)$ weights the error relative to the error at different frequencies. The syntax

```
invfreqz(h,w,n,m,wt)
```

includes a weighting vector. In this mode, the filter resulting from `invfreqz` is not guaranteed to be stable.

`invfreqz` provides a superior ("output-error") algorithm that solves the direct problem of minimizing the weighted sum of the squared error between the actual frequency response points and the desired response

$$\min_{b,a} \sum_{k=1}^{n} wt(k) \left| h(k) - \frac{B(w(k))}{A(w(k))} \right|^2$$

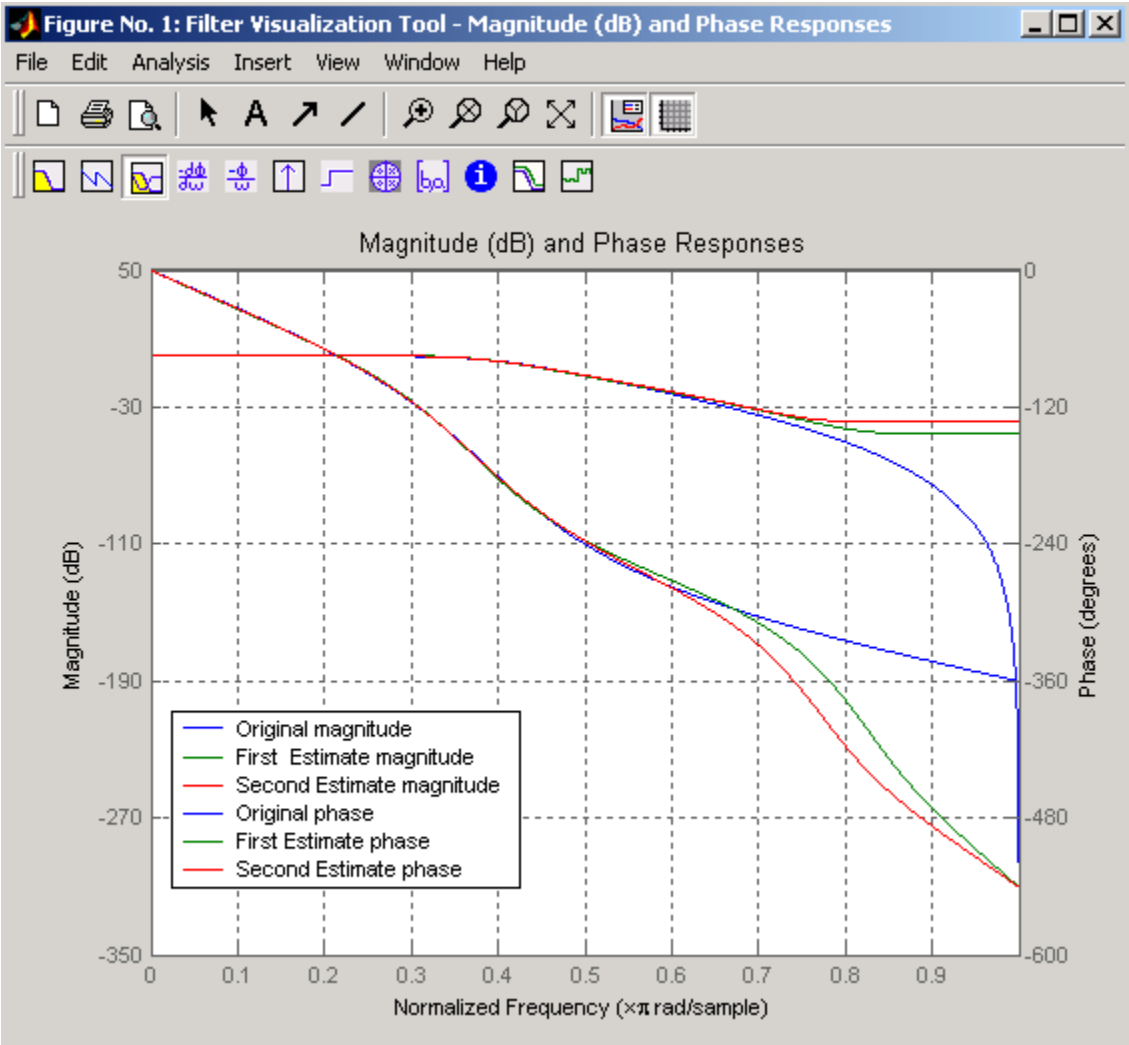
To use this algorithm, specify a parameter for the iteration count after the weight vector parameter:

```
wt = ones(size(w));   % Create unit weighting vector
[b30,a30] = invfreqz(h,w,3,3,wt,30)  % 30 iterations
```

The resulting filter is always stable.

Graphically compare the results of the first and second algorithms to the original Butterworth filter with FVTool (and select the Magnitude and Phase Responses):

```
fvtool(b,a,b4,a4,b30,a30)
```




To verify the superiority of the fit numerically, type

```
sum(abs(h-freqz(b4,a4,w)).^2)    % Total error, algorithm 1
sum(abs(h-freqz(b30,a30,w)).^2)  % Total error, algorithm 2
```

# Resampling

| In this section... |
| --- |
| “resample Function” on page 8-24<br>“decimate and interp Functions” on page 8-25<br>“upfirdn Function” on page 8-25<br>“spline Function” on page 8-25 |

Signal Processing Toolbox provides a number of functions that resample a signal at a higher or lower rate.

| Operation | Function |
| --- | --- |
| Apply FIR filter with resampling | `upfirdn` |
| Cubic spline interpolation | `spline` |
| Decimation | `decimate` |
| Interpolation | `interp` |
| Other 1-D interpolation | `interp1` |
| Resample at new rate | `resample` |

For examples, see

- “Reconstructing Missing Data” on page 24-27
- “Resampling Uniformly Sampled Signals” on page 24-38
- “Resampling Nonuniformly Sampled Signals” on page 24-46
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84

## resample Function

The `resample` function changes the sample rate for a sequence to any rate that is proportional to the original by a ratio of two integers. The basic syntax for `resample` is

```
y = resample(x,p,q)
```

where the function resamples the sequence `x` at `p/q` times the original sample rate. The length of the result `y` is `p/q` times the length of `x`.

One resampling application is the conversion of digitized audio signals from one sample rate to another, such as from 48 kHz (the digital audio tape standard) to 44.1 kHz (the compact disc standard). See “Convert from DAT Rate to CD Sample Rate” for an example.

`resample` applies a lowpass filter to the input sequence to prevent aliasing during resampling. The function designs this filter using the `firls` function with a Kaiser window. You can control the filter length and the beta parameter of the Kaiser window. Alternatively, you can use the function `intfilt` to design an interpolation filter.

## decimate and interp Functions

The `decimate` and `interp` functions are equivalent to `resample` with `p = 1` and `q = 1`, respectively. These functions provide different antialiasing filtering options, and they incur a slight signal delay due to filtering.

## upfirdn Function

The toolbox also contains a function, `upfirdn`, that applies an FIR filter to an input sequence and outputs the filtered sequence at a sample rate different than its original. See “Multirate Filter Bank Implementation” on page 1-6.

## spline Function

The standard MATLAB environment contains a function, `spline`, that works with irregularly spaced data. The function `interp1` performs interpolation, or table lookup, using various methods including linear and cubic interpolation.

## See Also

**Apps**
**Signal Analyzer**

**Functions**
`decimate` | `interp` | `interp1` | `resample` | `spline` | `upfirdn`

# Cepstrum Analysis

### What Is a Cepstrum?

Cepstrum analysis is a nonlinear signal processing technique with a variety of applications in areas such as speech and image processing.

The *complex cepstrum* of a sequence $x$ is calculated by finding the complex natural logarithm of the Fourier transform of $x$, then the inverse Fourier transform of the resulting sequence:

$$\hat{x} = \frac{1}{2\pi}\int_{-\pi}^{\pi} \log[X(e^{j\omega})]e^{j\omega n}\, d\omega\,.$$

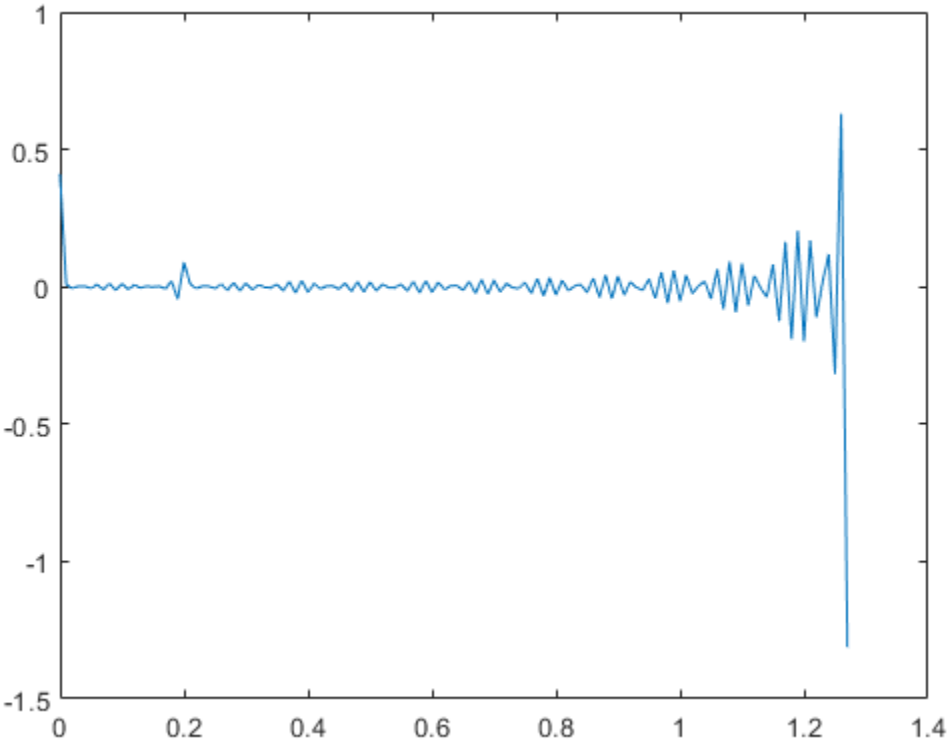
The toolbox function `cceps` performs this operation, estimating the complex cepstrum for an input sequence. It returns a real sequence the same size as the input sequence.

Try using cceps in an echo detection application. First, create a 45 Hz sine wave sampled at 100 Hz. Add an echo of the signal, with half the amplitude, 0.2 seconds after the beginning of the signal.

```
t = 0:0.01:1.27;
s1 = sin(2*pi*45*t);
s2 = s1 + 0.5*[zeros(1,20) s1(1:108)];
```

Compute and plot the complex cepstrum of the new signal.

```
c = cceps(s2);
plot(t,c)
```



The complex cepstrum shows a peak at 0.2 seconds, indicating the echo.

The *real cepstrum* of a signal $x$, sometimes called simply the cepstrum, is calculated by determining the natural logarithm of magnitude of the Fourier transform of $x$, then obtaining the inverse Fourier transform of the resulting sequence:

$$c_x = \frac{1}{2\pi}\int_{-\pi}^{\pi} \log\left|X(e^{j\omega})\right| e^{j\omega n}\, d\omega\,.$$

The toolbox function `rceps` performs this operation, returning the real cepstrum for a sequence. The returned sequence is a real-valued vector the same size as the input vector.

The `rceps` function also returns a unique minimum-phase sequence that has the same real cepstrum as the input. To obtain both the real cepstrum and the minimum-phase reconstruction for a sequence, use `[y,ym] = rceps(x)`, where `y` is the real cepstrum and `ym` is the minimum phase reconstruction of `x`. The following example shows that one output of `rceps` is a unique minimum-phase sequence with the same real cepstrum as `x`.

```
y = [4 1 5];                  % Non-minimum phase sequence
[xhat,yhat] = rceps(y);
xhat2 = rceps(yhat);
[xhat' xhat2']
```

```
ans = 3×2

    1.6225    1.6225
    0.3400    0.3400
    0.3400    0.3400
```

### Inverse Complex Cepstrum

To invert the complex cepstrum, use the `icceps` function. Inversion is complicated by the fact that the `cceps` function performs a data-dependent phase modification so that the unwrapped phase of its input is continuous at zero frequency. The phase modification is equivalent to an integer delay. This delay term is returned by `cceps` if you ask for a second output:

```
x = 1:10;
[xhat,delay] = cceps(x)
```

```
xhat = 1×10

    2.2428   -0.0420   -0.0210    0.0045    0.0366    0.0788    0.1386    0.2327    0.4114    0.9


delay = 1
```

To invert the complex cepstrum, use `icceps` with the original delay parameter:

```
icc = icceps(xhat,2)
```

```
icc = 1×10

    2.0000    3.0000    4.0000    5.0000    6.0000    7.0000    8.0000    9.0000   10.0000    1.0
```

As shown in the above example, with any modification of the complex cepstrum, the original delay term may no longer be valid. You will not be able to invert the complex cepstrum exactly.

## See Also

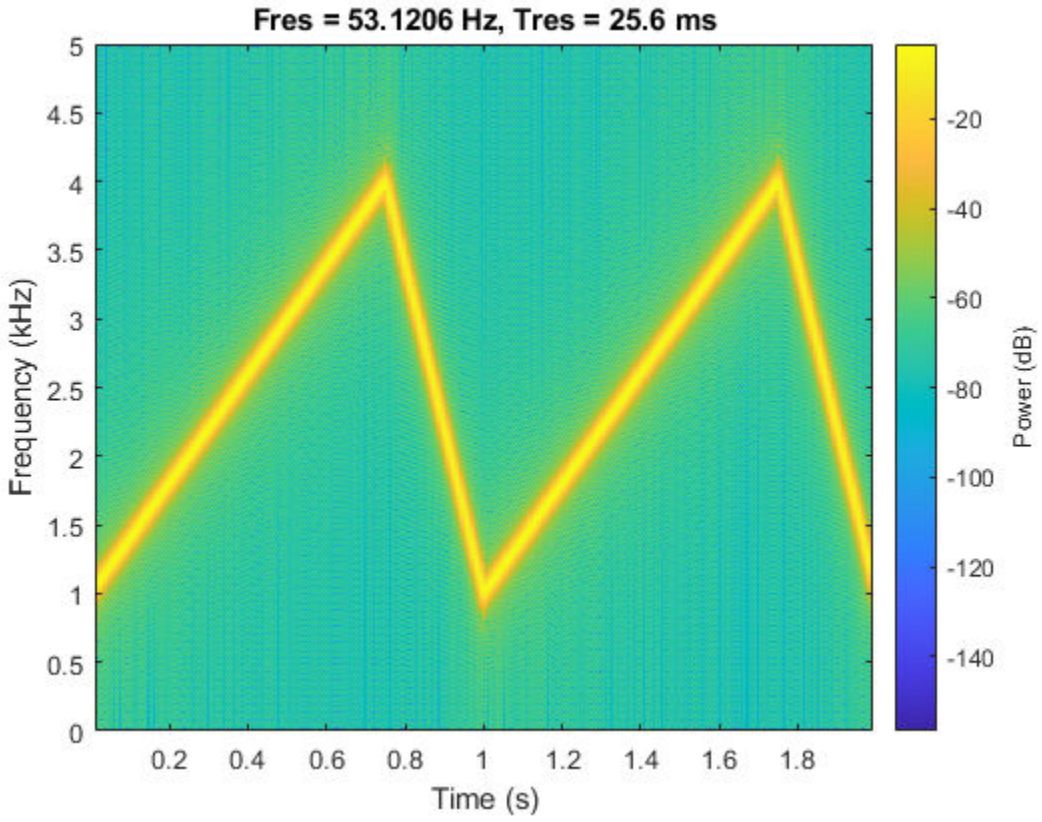
`cceps` | `icceps` | `rceps`

# FFT-Based Time-Frequency Analysis

The Signal Processing Toolbox™ product provides functions that return the time-dependent Fourier transform for a sequence, or displays this information as a spectrogram. The *time-dependent Fourier transform* is the discrete-time Fourier transform for a sequence, computed using a sliding window. This form of the Fourier transform, also known as the short-time Fourier transform (STFT), has numerous applications in speech, sonar, and radar processing. The *spectrogram* of a sequence is the magnitude of the time-dependent Fourier transform versus time.

To display the spectrogram of a signal, you can use the `pspectrum` function. For example, generate two seconds of a signal sampled at 10 kHz. Specify the instantaneous frequency of the signal as a triangular function of time. To compute the spectrogram, divide the signal into segments of duration 0.0256 second and specify 86% segment-to-segment overlap. The *leakage* measures the ability of the sliding window to detect a weak tone from noise in the presence of a neighboring strong tone. Specify a leakage of 0.875.

```
fs = 10000;
t = 0:1/fs:2;
x = vco(sawtooth(2*pi*t,0.75),[0.1 0.4]*fs,fs);

pspectrum(x,fs,'spectrogram', ...
    'TimeResolution',0.0256,'Overlap',86,'Leakage',0.875)
```

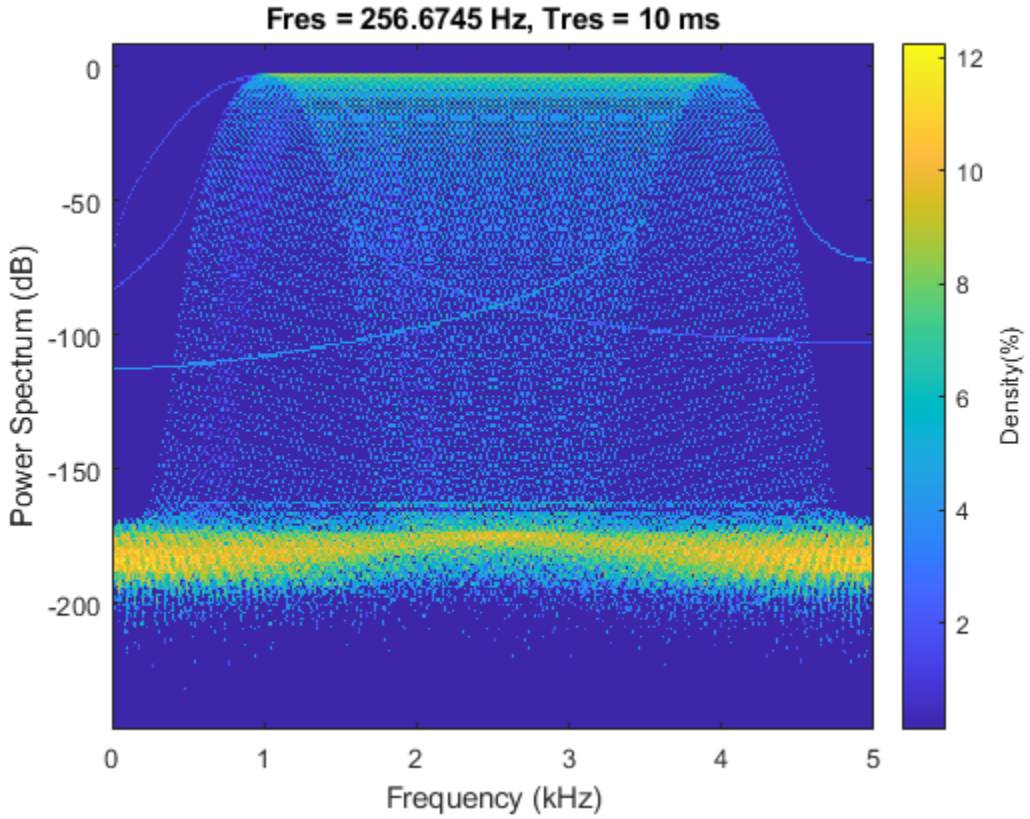



The *persistence spectrum* of a signal is a time-frequency view that shows the percentage of the time that a given frequency is present in a signal. The persistence spectrum is a histogram in power-

frequency space. The longer a particular frequency persists in a signal as the signal evolves, the higher its time percentage and thus the brighter or "hotter" its color in the display.

Compute and display the persistence spectrum of the signal. Specify a time resolution of 0.01 second, 50% overlap between adjoining segments, and a leakage of 0.5.

```
pspectrum(x,fs,'persistence', ...
    'TimeResolution',0.01,'Overlap',50,'Leakage',0.5)
```




## See Also

**Apps**
**Signal Analyzer**

**Functions**
`fsst` | `ifsst` | `pspectrum` | `spectrogram` | `tfridge` | `xspectrogram`

## Related Examples

- “Practical Introduction to Time-Frequency Analysis” on page 24-267
- “Detect Closely Spaced Sinusoids” on page 16-25
- “Hilbert Transform and Instantaneous Frequency” on page 16-18

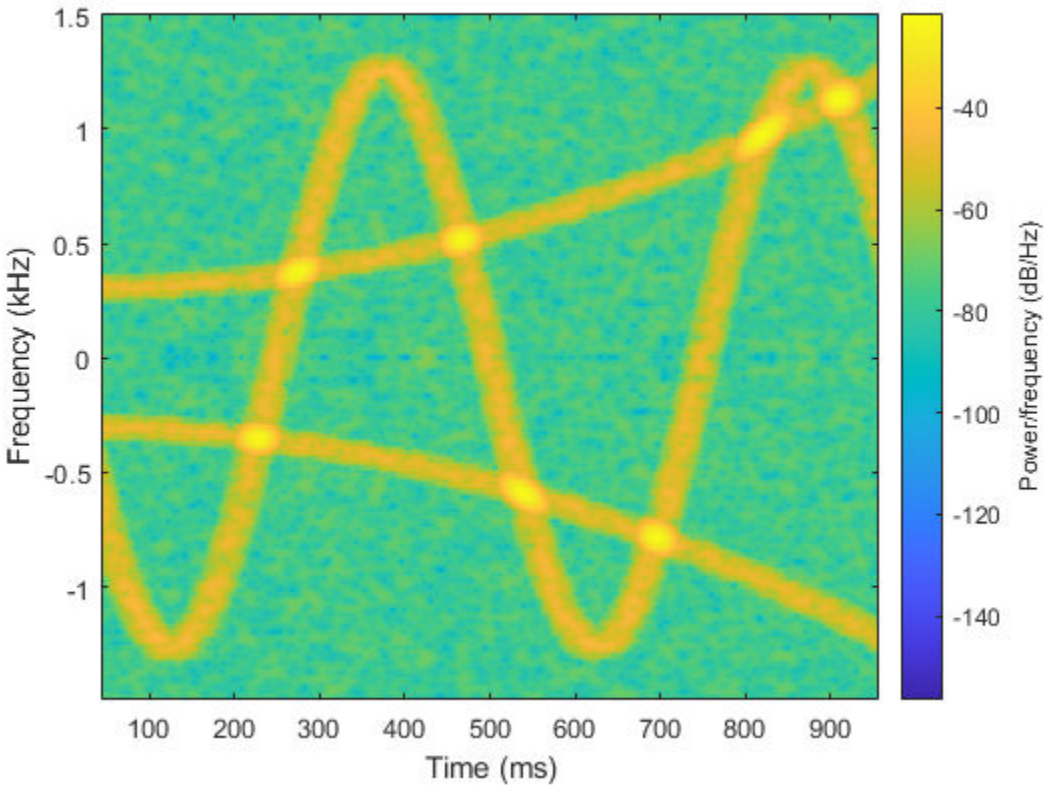
# Cross-Spectrogram of Complex Signals

Generate two signals, each sampled at 3 kHz for 1 second. The first signal is a quadratic chirp whose frequency increases from 300 Hz to 1300 Hz during the measurement. The chirp is embedded in white Gaussian noise. The second signal, also embedded in white noise, is a chirp with sinusoidally varying frequency content.

```
fs = 3000;
t = 0:1/fs:1-1/fs;

x1 = chirp(t,300,t(end),1300,'quadratic')+randn(size(t))/100;

x2 = exp(2j*pi*100*cos(2*pi*2*t))+randn(size(t))/100;
```

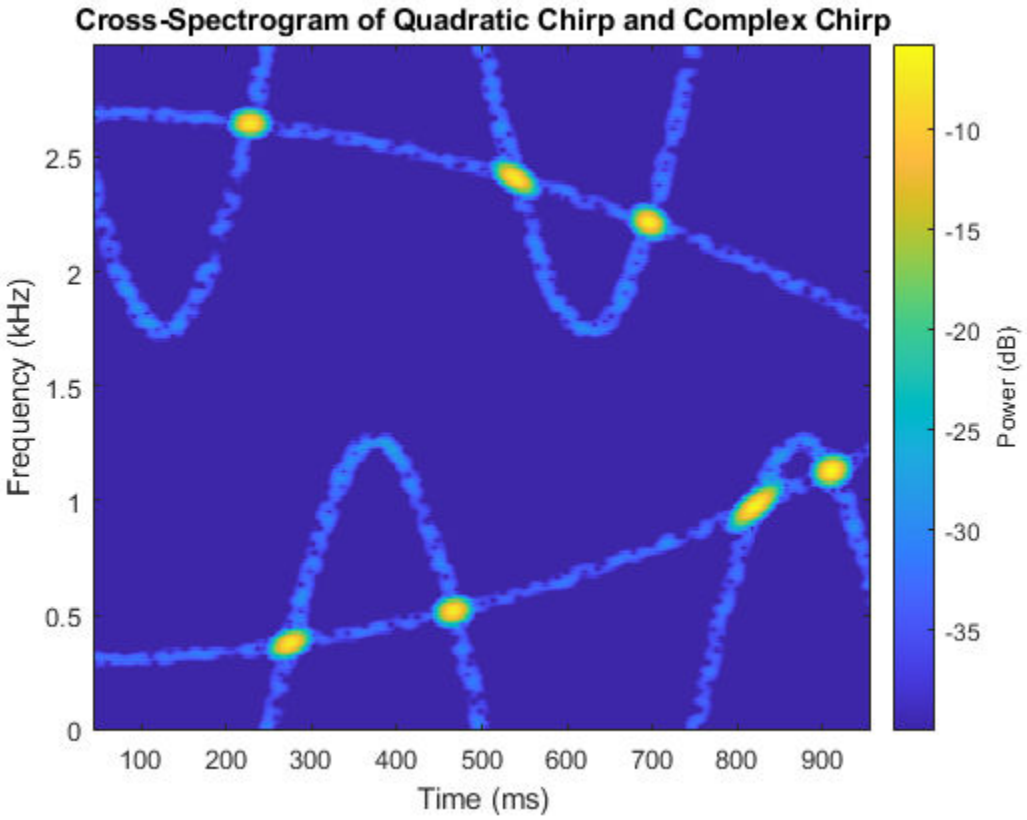
Compute and plot the cross-spectrogram of the two signals. Divide the signals into 256-sample segments with 255 samples of overlap between adjoining segments. Use a Kaiser window with shape factor $\beta = 30$ to window the segments. Use the default number of DFT points. Center the cross-spectrogram at zero frequency.

```
nwin = 256;

xspectrogram(x1,x2,kaiser(nwin,30),nwin-1,[],fs,'centered','yaxis')
```




Compute the power spectrum instead of the power spectral density. Set to zero the values smaller than –40 dB. Center the plot at the Nyquist frequency.

```
xspectrogram(x1,x2,kaiser(nwin,30),nwin-1,[],fs, ...
    'power','MinThreshold',-40,'yaxis')
title('Cross-Spectrogram of Quadratic Chirp and Complex Chirp')
```




The thresholding further highlights the regions of common frequency.

## See Also

`spectrogram` | `xspectrogram`

# Median Filtering

The function `medfilt1` implements one-dimensional median filtering, a nonlinear technique that applies a sliding window to a sequence. The median filter replaces the center value in the window with the median value of all the points within the window [5]. In computing this median, `medfilt1` assumes zeros beyond the input points.

When the number of elements `n` in the window is even, `medfilt1` sorts the numbers, then takes the average of the `n/2` and `n/2 + 1` elements.

Two simple examples with fourth- and third-order median filters are

```
medfilt1([4 3 5 2 8 9 1],4)
ans =
    1.500 3.500 3.500 4.000 6.500 5.000 4.500
medfilt1([4 3 5 2 8 9 1],3)
ans =
     3     4     3     5     8     8     1
```

See the `medfilt2` function in the Image Processing Toolbox™ for information on two-dimensional median filtering.

# Communications Applications

| In this section... |
| --- |
| “Modulation” on page 8-34<br>“Demodulation” on page 8-35<br>“Voltage Controlled Oscillator” on page 8-36 |

## Modulation

*Modulation* varies the amplitude, phase, or frequency of a *carrier signal* with reference to a *message signal*. The `modulate` function modulates a message signal with a specified modulation method.

The basic syntax for the `modulate` function is

```
y = modulate(x,fc,fs,'method',opt)
```

where:

- `x` is the message signal.
- `fc` is the carrier frequency.
- `fs` is the sampling frequency.
- *`method`* is a flag for the desired modulation method.
- `opt` is any additional argument that the method requires. (Not all modulation methods require an option argument.)

The table below summarizes the modulation methods provided; see the documentation for `modulate`, `demod`, and `vco` for complete details on each.

| Method | Description |
| --- | --- |
| `amdsb-sc` or `am` | Amplitude modulation, double sideband, suppressed carrier |
| `amdsb-tc` | Amplitude modulation, double sideband, transmitted carrier |
| `amssb` | Amplitude modulation, single sideband |
| `fm` | Frequency modulation |
| `pm` | Phase modulation |
| `ppm` | Pulse position modulation |
| `pwm` | Pulse width modulation |
| `qam` | Quadrature amplitude modulation |

If the input `x` is an array rather than a vector, `modulate` modulates each column of the array.

To obtain the time vector that `modulate` uses to compute the modulated signal, specify a second output parameter:

```
[y,t] = modulate(x,fc,fs,'method',opt)
```

## Demodulation

The `demod` function performs *demodulation*, that is, it obtains the original message signal from the modulated signal:

The syntax for `demod` is

```
x = demod(y,fc,fs,'method',opt)
```

`demod` uses any of the methods shown for `modulate`, but the syntax for quadrature amplitude demodulation requires two output parameters:

```
[X1,X2] = demod(y,fc,fs,'qam')
```

If the input `y` is an array, `demod` demodulates all columns.

Try modulating and demodulating a signal. A 50 Hz sine wave sampled at 1000 Hz is

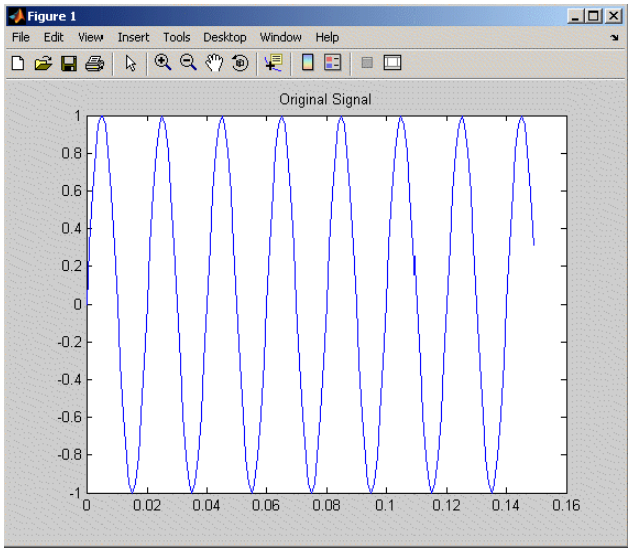
```
t = (0:1/1000:2);
x = sin(2*pi*50*t);
```

With a carrier frequency of 200 Hz, the modulated and demodulated versions of this signal are

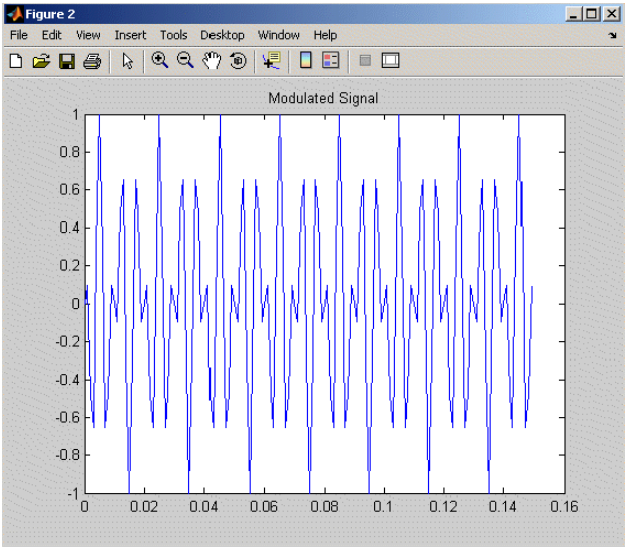
```
y = modulate(x,200,1000,'am');
z = demod(y,200,1000,'am');
```

To plot portions of the original, modulated, and demodulated signal:

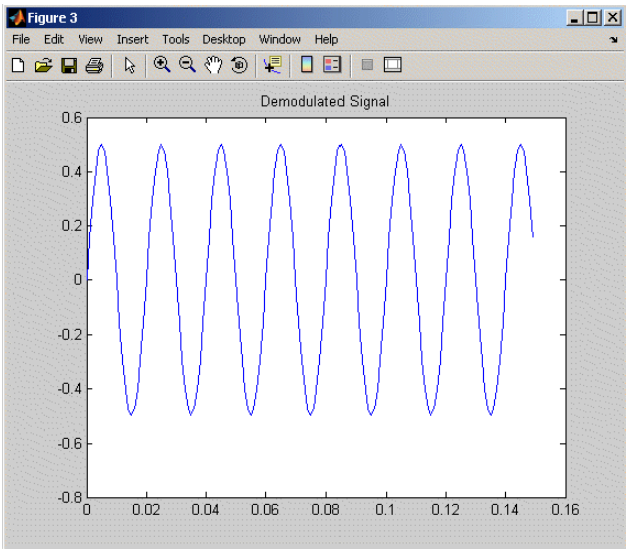
```
figure; plot(t(1:150),x(1:150)); title('Original Signal');
figure; plot(t(1:150),y(1:150)); title('Modulated Signal');
figure; plot(t(1:150),z(1:150)); title('Demodulated Signal');
```




**Original Signal**




**Modulated Signal**




**Demodulated Signal**

**Note** The demodulated signal is attenuated because demodulation includes two steps: multiplication and lowpass filtering. The multiplication produces a component with frequency centered at 0 Hz and a component with frequency at twice the carrier frequency. The filtering removes the higher frequency component of the signal, producing the attenuated result.

## Voltage Controlled Oscillator

The voltage controlled oscillator function `vco` creates a signal that oscillates at a frequency determined by the input vector. The basic syntax for `vco` is

```
y = vco(x,fc,fs)
```

where `fc` is the carrier frequency and `fs` is the sampling frequency.

To scale the frequency modulation range, use

```
y = vco(x,[Fmin Fmax],fs)
```

In this case, `vco` scales the frequency modulation range so values of `x` on the interval `[-1 1]` map to oscillations of frequency on `[Fmin Fmax]`.

If the input `x` is an array, `vco` produces an array whose columns oscillate according to the columns of `x`.

See “FFT-Based Time-Frequency Analysis” on page 8-29 for an example using the `vco` function.

# Deconvolution

Deconvolution, or polynomial division, is the inverse operation of convolution. Deconvolution is useful in recovering the input to a known filter, given the filtered output. This method is very sensitive to noise in the coefficients, however, so use caution in applying it.

The syntax for `deconv` is

```
[q,r] = deconv(b,a)
```

where `b` is the polynomial dividend, `a` is the divisor, `q` is the quotient, and `r` is the remainder.

To try `deconv`, first convolve two simple vectors `a` and `b`.

```
a = [1 2 3];
b = [4 5 6];
c = conv(a,b)

c =
     4    13    28    27    18
```

Now use `deconv` to deconvolve `b` from `c`:

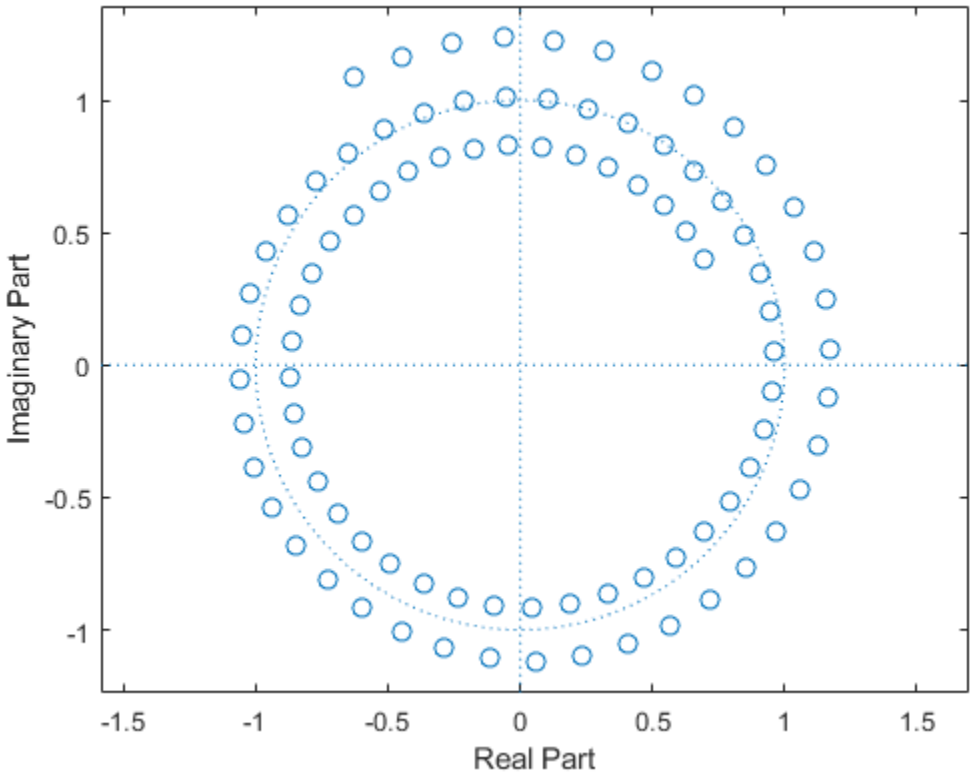
```
[q,r] = deconv(c,a)

q =
     4     5     6
r =
     0     0     0     0     0
```

# Chirp Z-Transform

The chirp Z-transform (CZT) is useful in evaluating the Z-transform along contours other than the unit circle. The chirp Z-transform is also more efficient than the DFT algorithm for the computation of prime-length transforms, and it is useful in computing a subset of the DFT for a sequence. The chirp Z-transform, or CZT, computes the Z-transform along spiral contours in the $z$-plane for an input sequence. Unlike the DFT, the CZT is not constrained to operate along the unit circle, but can evaluate the Z-transform along contours described by $z_\ell = AW^{-\ell}, \quad \ell = 0, \cdots, M-1$, where $A$ is the complex starting point, $W$ is a complex scalar describing the complex ratio between points on the contour, and $M$ is the length of the transform.

One possible spiral is

```
a = 0.8*exp(1j*pi/6);
w = 0.995*exp(-1j*pi*.05);
m = 91;
z = a*(w.^(-(0:m-1)'));
zplane(z)
```

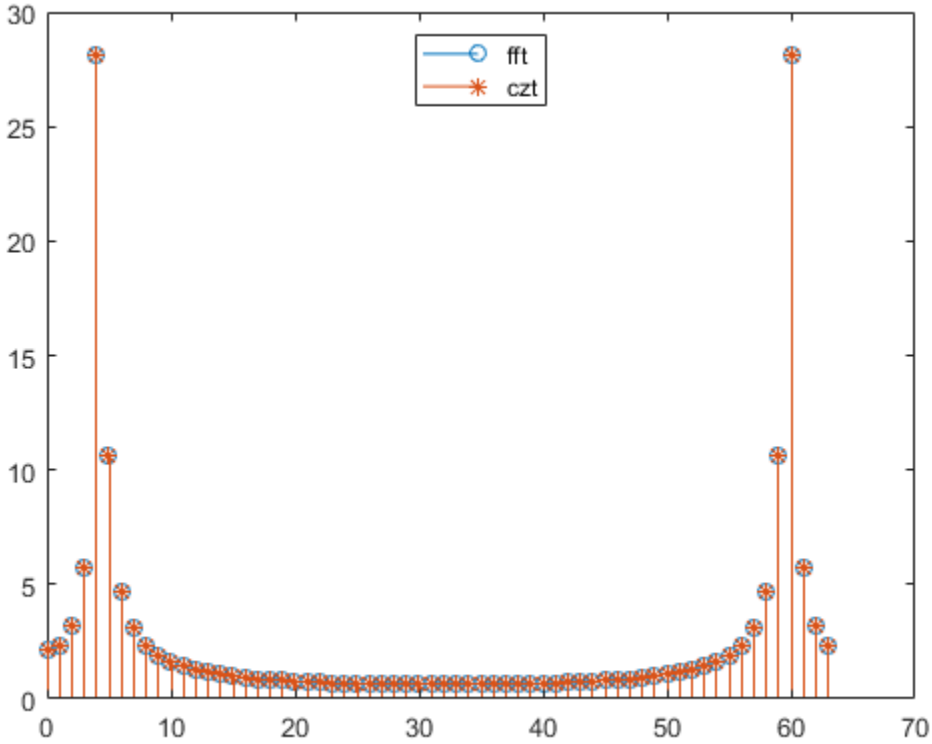



`czt(x,m,w,a)` computes the Z-transform of `x` on these points.

An interesting and useful spiral set is $m$ evenly spaced samples around the unit circle, parameterized by $A = 1$ and $W = \exp(-j\pi/M)$. The Z-transform on this contour is simply the DFT, obtained by `czt`:

```
M = 64;
m = 0:M-1;
```

```
x = sin(2*pi*m/15);
FFT = fft(x);
CZT = czt(x,M,exp(-2j*pi/M),1);

stem(m,abs(FFT))
hold on
stem(m,abs(CZT),'*')
hold off
legend('fft','czt','Location','north')
```




`czt` may be faster than the `fft` function for computing the DFT of sequences with certain odd lengths, particularly long prime-length sequences.

## See Also

`czt` | `fft`

# Discrete Cosine Transform

The discrete cosine transform (DCT) is closely related to the discrete Fourier transform (DFT). The DFT is actually one step in the computation of the DCT for a sequence. The DCT, however, has better *energy compaction* than the DFT, with just a few of the transform coefficients representing the majority of the energy in the sequence. This property of the DCT makes it useful in applications such as data communications and signal coding.

### DCT Variants

The DCT has four standard variants. For a signal $x$ of length $N$, and with $\delta_{k\ell}$ the Kronecker delta, the transforms are defined by:

- DCT-1:

$$y(k) = \sqrt{\frac{2}{N-1}} \sum_{n=1}^{N} x(n) \frac{1}{\sqrt{1+\delta_{n1}+\delta_{nN}}} \frac{1}{\sqrt{1+\delta_{k1}+\delta_{kN}}} \cos\left(\frac{\pi}{N-1}(n-1)(k-1)\right)$$

- DCT-2:

$$y(k) = \sqrt{\frac{2}{N}} \sum_{n=1}^{N} x(n) \frac{1}{\sqrt{1+\delta_{k1}}} \cos\left(\frac{\pi}{2N}(2n-1)(k-1)\right)$$

- DCT-3:

$$y(k) = \sqrt{\frac{2}{N}} \sum_{n=1}^{N} x(n) \frac{1}{\sqrt{1+\delta_{n1}}} \cos\left(\frac{\pi}{2N}(n-1)(2k-1)\right)$$

- DCT-4:

$$y(k) = \sqrt{\frac{2}{N}} \sum_{n=1}^{N} x(n) \cos\left(\frac{\pi}{4N}(2n-1)(2k-1)\right)$$

The Signal Processing Toolbox function `dct` computes the unitary DCT of an input array.

### Inverse DCT Variants

All variants of the DCT are *unitary* (or, equivalently, *orthogonal*): To find their inverses, switch $k$ and $n$ in each definition. DCT-1 and DCT-4 are their own inverses. DCT-2 and DCT-3 are inverses of each other:

- Inverse of DCT-1:

$$x(n) = \sqrt{\frac{2}{N-1}} \sum_{k=1}^{N} y(k) \frac{1}{\sqrt{1+\delta_{k1}+\delta_{kN}}} \frac{1}{\sqrt{1+\delta_{n1}+\delta_{nN}}} \cos\left(\frac{\pi}{N-1}(k-1)(n-1)\right)$$

- Inverse of DCT-2:

$$x(n) = \sqrt{\frac{2}{N}} \sum_{k=1}^{N} y(k) \frac{1}{\sqrt{1+\delta_{k1}}} \cos\left(\frac{\pi}{2N}(k-1)(2n-1)\right)$$

- Inverse of DCT-3:

$$x(n) = \sqrt{\frac{2}{N}} \sum_{k=1}^{N} y(k) \frac{1}{\sqrt{1+\delta_{n1}}} \cos\left(\frac{\pi}{2N}(2k-1)(n-1)\right)$$

- Inverse of DCT-4:

$$x(n) = \sqrt{\frac{2}{N}} \sum_{k=1}^{N} y(k) \cos\left(\frac{\pi}{4N}(2k-1)(2n-1)\right)$$

The function `idct` computes the inverse DCT for an input sequence, reconstructing a signal from a complete or partial set of DCT coefficients.

**Signal Reconstruction Using DCT**

Because of the energy compaction property of the DCT, you can reconstruct a signal from only a fraction of its DCT coefficients. For example, generate a 25 Hz sinusoidal sequence sampled at 1000 Hz.

```
t = 0:1/1000:1;
x = sin(2*pi*25*t);
```

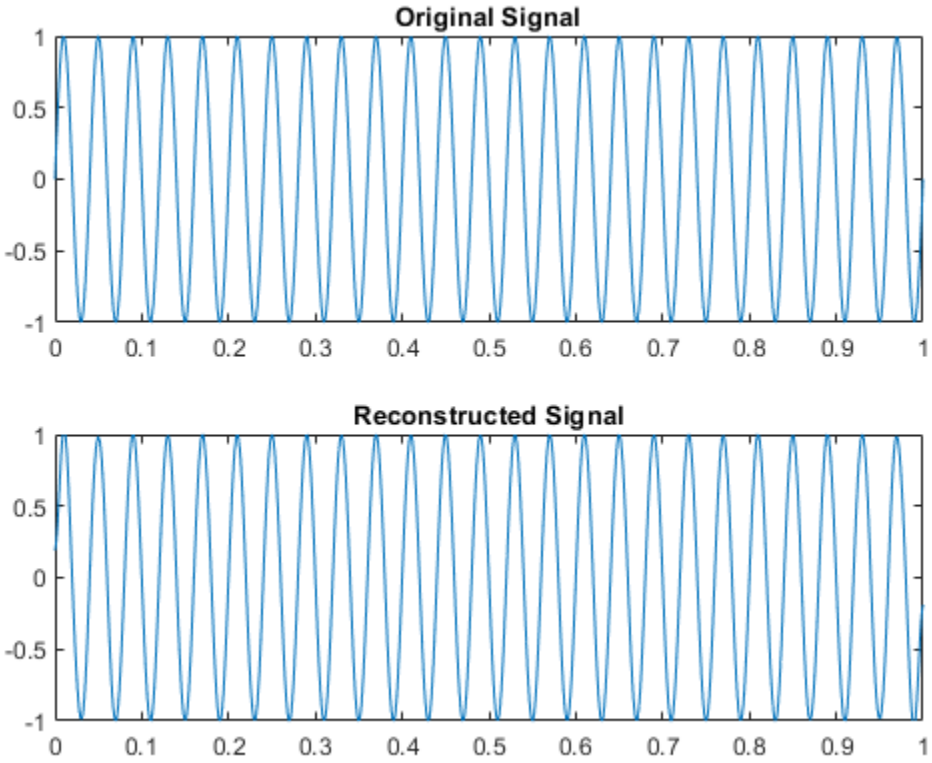
Compute the DCT of this sequence and reconstruct the signal using only those components with value greater than 0.1. Determine how many coefficients out of the original 1000 satisfy the requirement.

```
y = dct(x);
y2 = find(abs(y) < 0.1);
y(y2) = zeros(size(y2));
z = idct(y);

howmany = length(find(y))
```

```
howmany = 64
```

Plot the original and reconstructed sequences.

```
subplot(2,1,1)
plot(t,x)
ax = axis;
title('Original Signal')

subplot(2,1,2)
plot(t,z)
axis(ax)
title('Reconstructed Signal')
```




One measure of the accuracy of the reconstruction is the norm of the difference between the original and reconstructed signals, divided by the norm of the original signal. Compute this estimate and express it as a percentage.

```
norm(x-z)/norm(x)*100
```

```
ans = 1.9437
```

The reconstructed signal retains approximately 98% of the energy in the original signal.

## See Also

`dct` | `idct`

## Related Examples

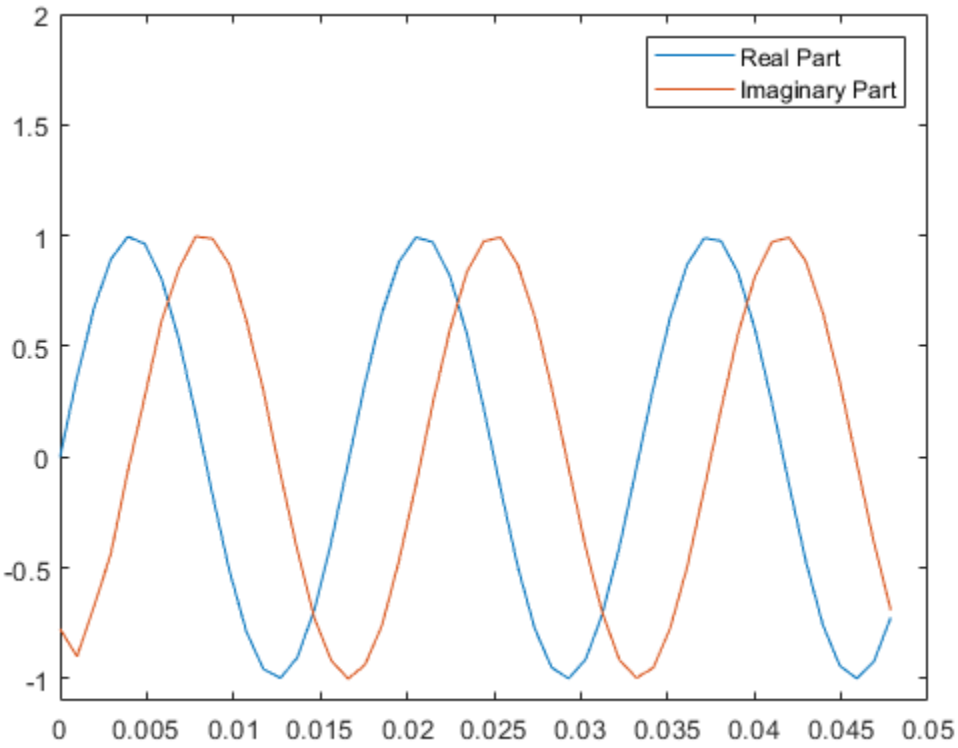
- “DCT for Speech Signal Compression” on page 16-42

# Hilbert Transform

The Hilbert transform facilitates the formation of the analytic signal. The analytic signal is useful in the area of communications, particularly in bandpass signal processing. The toolbox function `hilbert` computes the Hilbert transform for a real input sequence `x` and returns a complex result of the same length, `y = hilbert(x)`, where the real part of `y` is the original real data and the imaginary part is the actual Hilbert transform. `y` is sometimes called the *analytic signal*, in reference to the continuous-time analytic signal. A key property of the discrete-time analytic signal is that its Z-transform is 0 on the lower half of the unit circle. Many applications of the analytic signal are related to this property; for example, the analytic signal is useful in avoiding aliasing effects for bandpass sampling operations. The magnitude of the analytic signal is the complex envelope of the original signal.

The Hilbert transform is related to the actual data by a 90-degree phase shift; sines become cosines and vice versa. To plot a portion of data and its Hilbert transform, use

```
t = 0:1/1024:1;
x = sin(2*pi*60*t);
y = hilbert(x);

plot(t(1:50),real(y(1:50)))
hold on
plot(t(1:50),imag(y(1:50)))
hold off
axis([0 0.05 -1.1 2])
legend('Real Part','Imaginary Part')
```



The analytic signal is useful in calculating instantaneous attributes of a time series, the attributes of the series at any point in time. The procedure requires that the signal be monocomponent.

## See Also

`hilbert`

## Related Examples

- “Analytic Signal for Cosine” on page 16-5
- “Envelope Extraction” on page 16-7
- “Analytic Signal and Hilbert Transform” on page 16-13
- “Hilbert Transform and Instantaneous Frequency” on page 16-18

# Walsh-Hadamard Transform

The Walsh-Hadamard transform is a non-sinusoidal, orthogonal transformation technique that decomposes a signal into a set of basis functions. These basis functions are Walsh functions, which are rectangular or square waves with values of +1 or –1. Walsh-Hadamard transforms are also known as Hadamard (see the `hadamard` function in the MATLAB software), Walsh, or Walsh-Fourier transforms.

The first eight Walsh functions have these values:

| Index | Walsh Function Values |
|---|---|
| 0 | 1 1 1 1 1 1 1 1 |
| 1 | 1 1 1 1 -1 -1 -1 -1 |
| 2 | 1 1 -1 -1 -1 -1 1 1 |
| 3 | 1 1 -1 -1 1 1 -1 -1 |
| 4 | 1 -1 -1 1 1 -1 -1 1 |
| 5 | 1 -1 -1 1 -1 1 1 -1 |
| 6 | 1 -1 1 -1 -1 1 -1 1 |
| 7 | 1 -1 1 -1 1 -1 1 -1 |

The Walsh-Hadamard transform returns sequency values. Sequency is a more generalized notion of frequency and is defined as one half of the average number of zero-crossings per unit time interval. Each Walsh function has a unique sequency value. You can use the returned sequency values to estimate the signal frequencies in the original signal.

Three different ordering schemes are used to store Walsh functions: sequency, Hadamard, and dyadic. Sequency ordering, which is used in signal processing applications, has the Walsh functions in the order shown in the table above. Hadamard ordering, which is used in controls applications, arranges them as 0, 4, 6, 2, 3, 7, 5, 1. Dyadic or gray code ordering, which is used in mathematics, arranges them as 0, 1, 3, 2, 6, 7, 5, 4.

The Walsh-Hadamard transform is used in a number of applications, such as image processing, speech processing, filtering, and power spectrum analysis. It is very useful for reducing bandwidth storage requirements and spread-spectrum analysis. Like the FFT, the Walsh-Hadamard transform has a fast version, the fast Walsh-Hadamard transform (`fwht`). Compared to the FFT, the FWHT requires less storage space and is faster to calculate because it uses only real additions and subtractions, while the FFT requires complex values. The FWHT is able to represent signals with sharp discontinuities more accurately using fewer coefficients than the FFT. Both the FWHT and the inverse FWHT (`ifwht`) are symmetric and thus, use identical calculation processes. The FWHT and IFWHT for a signal $x(t)$ of length $N$ are defined as:

$$y_n = \frac{1}{N}\sum_{i=0}^{N-1} x_i \mathrm{WAL}(n,i),$$

$$x_i = \sum_{i=0}^{N-1} y_n \mathrm{WAL}(n,i),$$

where $i = 0,1, \ldots, N - 1$ and WAL($n$,$i$) are Walsh functions. Similar to the Cooley-Tukey algorithm for the FFT, the $N$ elements are decomposed into two sets of $N/2$ elements, which are then combined

using a butterfly structure to form the FWHT. For images, where the input is typically a 2-D signal, the FWHT coefficients are calculated by first evaluating across the rows and then evaluating down the columns.

For the following simple signal, the resulting FWHT shows that `x` was created using Walsh functions with sequency values of 0, 1, 3, and 6, which are the nonzero indices of the transformed `x`. The inverse FWHT recreates the original signal.

```
x = [4 2 2 0 0 2 -2 0]
y = fwht(x)

x =

     4     2     2     0     0     2    -2     0

y =

     1     1     0     1     0     0     1     0

x1 = ifwht(y)


x1 =

     4     2     2     0     0     2    -2     0
```

## See Also

`fwht` | `ifwht`

## Related Examples

- “Walsh-Hadamard Transform for Spectral Analysis and Compression of ECG Signals” on page 8-48

# Walsh-Hadamard Transform for Spectral Analysis and Compression of ECG Signals

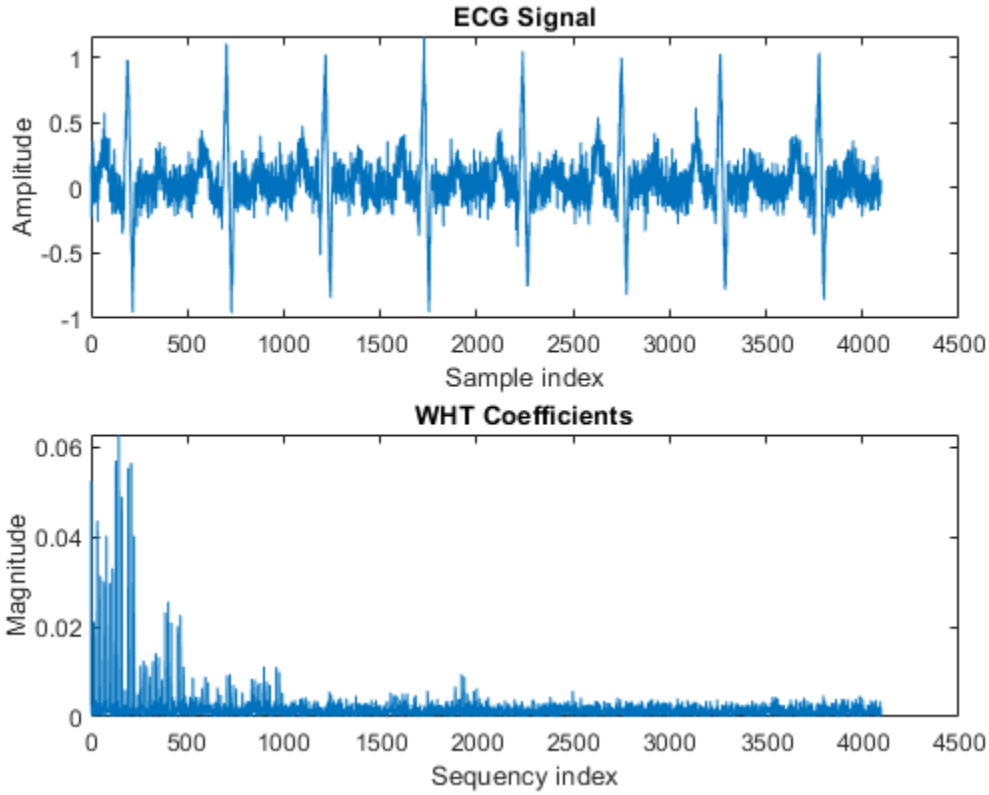
Use an electrocardiogram (ECG) signal to illustrate working with the Walsh-Hadamard transform. ECG signals typically are very large and need to be stored for analysis and retrieval at a future time. Walsh-Hadamard transforms are particularly well-suited to this application because they provide compression and thus require less storage space. They also provide rapid signal reconstruction.

Start with an ECG signal. Replicate it to create a longer signal and insert some additional random noise.

```
xe = ecg(512);
xr = repmat(xe,1,8);
x = xr + 0.1.*randn(1,length(xr));
```

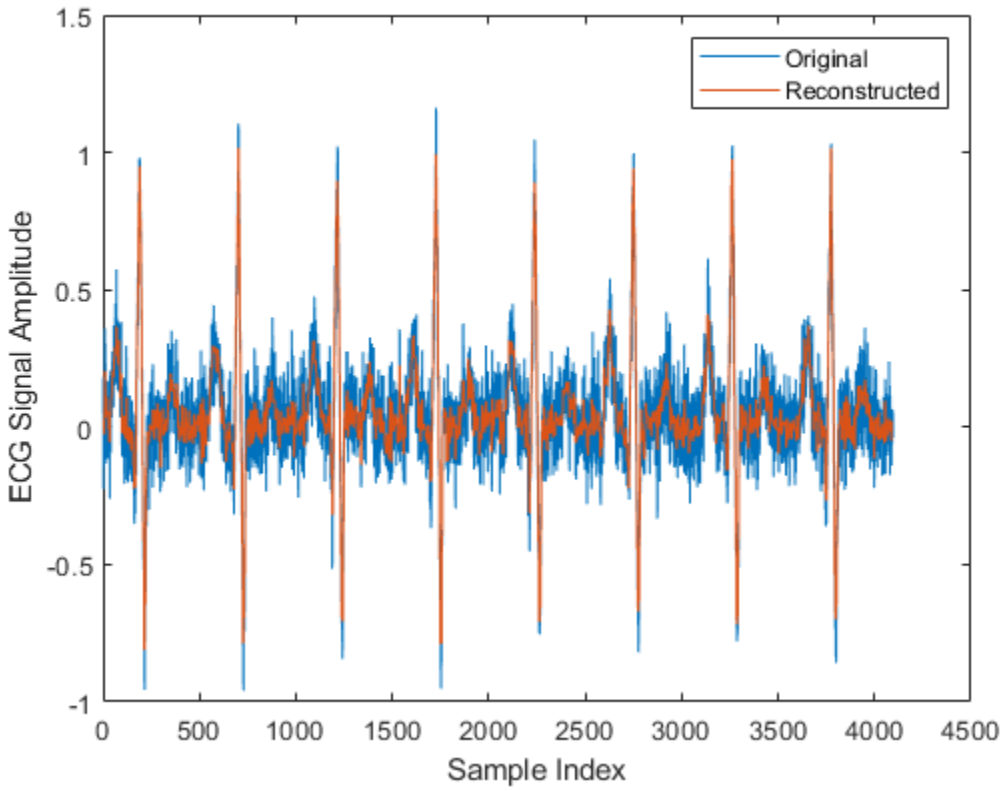
Transform the signal using the fast Walsh-Hadamard transform. Plot the original signal and the transformed signal.

```
y = fwht(x);

subplot(2,1,1)
plot(x)
xlabel('Sample index')
ylabel('Amplitude')
title('ECG Signal')

subplot(2,1,2)
plot(abs(y))
xlabel('Sequency index')
ylabel('Magnitude')
title('WHT Coefficients')
```




The plot shows that most of the signal energy is in the lower sequency values, below approximately 1100. Store only the first 1024 coefficients (out of 4096). Try to reconstruct the signal accurately from only these stored coefficients.

```
y(1025:length(x)) = 0;
xHat = ifwht(y);

figure
plot(x)
hold on
plot(xHat)
xlabel('Sample Index')
ylabel('ECG Signal Amplitude')
legend('Original','Reconstructed')
```




The reproduced signal is very close to the original but has been compressed to a quarter of the size. Storing more coefficients is a tradeoff between increased resolution and increased noise, while storing fewer coefficients can cause loss of peaks.

## See Also

`fwht` | `ifwht`

# Eliminate Outliers Using Hampel Identifier

This example shows a naive implementation of the procedure used by `hampel` to detect and remove outliers. The actual function is much faster.

Generate a random signal, `x`, containing 24 samples. Reset the random number generator for reproducible results.

```
rng default

lx = 24;
x = randn(1,lx);
```

Generate an observation window around each element of `x`. Take `k` = 2 neighbors at either side of the sample. The moving window that results has a length of 2 × 2 + 1 = 5 samples.

```
k = 2;

iLo = (1:lx)-k;
iHi = (1:lx)+k;
```

Truncate the window so that the function computes medians of smaller segments as it reaches the signal edges.

```
iLo(iLo<1) = 1;
iHi(iHi>lx) = lx;
```

Record the median of each surrounding window. Find the median of the absolute deviation of each element with respect to the window median.

```
for j = 1:lx
    w = x(iLo(j):iHi(j));
    medj = median(w);
    mmed(j) = medj;
    mmad(j) = median(abs(w-medj));
end
```

Scale the median absolute deviation with

$$\frac{1}{\sqrt{2}\,\mathrm{erf}^{-1}(1/2)} \approx 1.4826$$

to obtain an estimate of the standard deviation of a normal distribution.

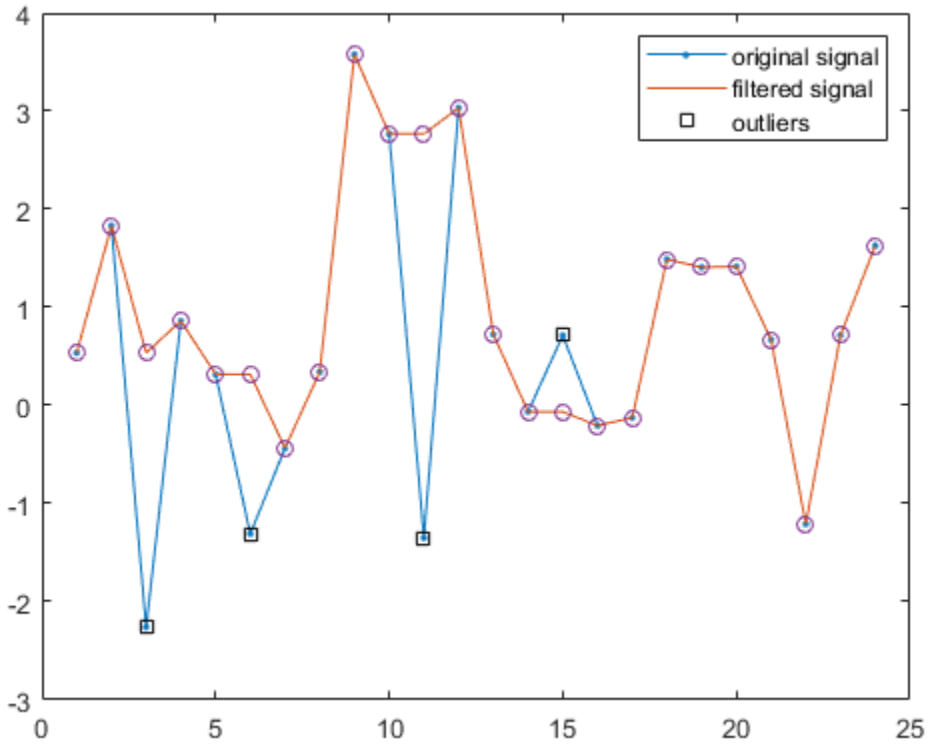
```
sd = mmad/(erfinv(1/2)*sqrt(2));
```

Find the samples that differ from the median by more than `nd` = 2 standard deviations. Replace each of those outliers by the value of the median of its surrounding window. This is the essence of the Hampel algorithm.

```
nd = 2;
ki = abs(x-mmed) > nd*sd;

yu = x;
yu(ki) = mmed(ki);
```

Use the `hampel` function to compute the filtered signal and annotate the outliers. Overlay the filtered values computed in this example.

```
hampel(x,k,nd)

hold on
plot(yu,'o','HandleVisibility','off')
hold off
```




## See Also

`hampel`

# Selected Bibliography

[1] Kay, Steven M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[2] Oppenheim, Alan V., and Ronald W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[3] Oppenheim, Alan V., and Ronald W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975.

[4] Parks, Thomas W., and C. Sidney Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987.

[5] Pratt, W. K. *Digital Image Processing*. New York: John Wiley & Sons, 1991.

# SPTool: A Signal Processing GUI Suite


- “SPTool: An Interactive Signal Processing Environment” on page 9-2
- “Opening SPTool” on page 9-3
- “Getting Context-Sensitive Help” on page 9-4
- “Signal Browser” on page 9-5
- “Filter Visualization Tool” on page 9-7
- “Spectrum Viewer” on page 9-9
- “Filtering and Analysis of Noise” on page 9-11
- “Exporting Signals, Filters, and Spectra” on page 9-19
- “Accessing Filter Parameters” on page 9-20
- “Importing Filters and Spectra” on page 9-22
- “Loading Variables from the Disk” on page 9-25
- “Saving and Loading Sessions” on page 9-26
- “Selecting Signals, Filters, and Spectra” on page 9-28
- “Editing Signals, Filters, or Spectra” on page 9-29
- “Making Signal Measurements with Markers” on page 9-30
- “Setting Preferences” on page 9-32

# SPTool: An Interactive Signal Processing Environment

| In this section... |
| --- |
| “SPTool Overview” on page 9-2<br>“SPTool Data Structures” on page 9-2 |

## SPTool Overview

SPTool is an interactive GUI for digital signal processing used to

- Analyze signals
- Design filters
- Analyze (view) filters
- Filter signals
- Analyze signal spectra

You can accomplish these tasks using four GUIs that you access from within SPTool:

- The “Signal Browser” on page 9-5 is for analyzing signals. You can also play signals using your computer's audio hardware.
- **Filter Designer** is available for designing or editing FIR and IIR digital filters. Most Signal Processing Toolbox filter design methods available at the command line are also available in **Filter Designer**.
- The “Filter Visualization Tool” on page 9-7 (FVTool) is for analyzing filter characteristics.
- The “Spectrum Viewer” on page 9-9 is for spectral analysis. You can use Signal Processing Toolbox spectral estimation methods to estimate the power spectral density of a signal.

## SPTool Data Structures

You can use SPTool to analyze signals, filters, or spectra that you create at the MATLAB command line.

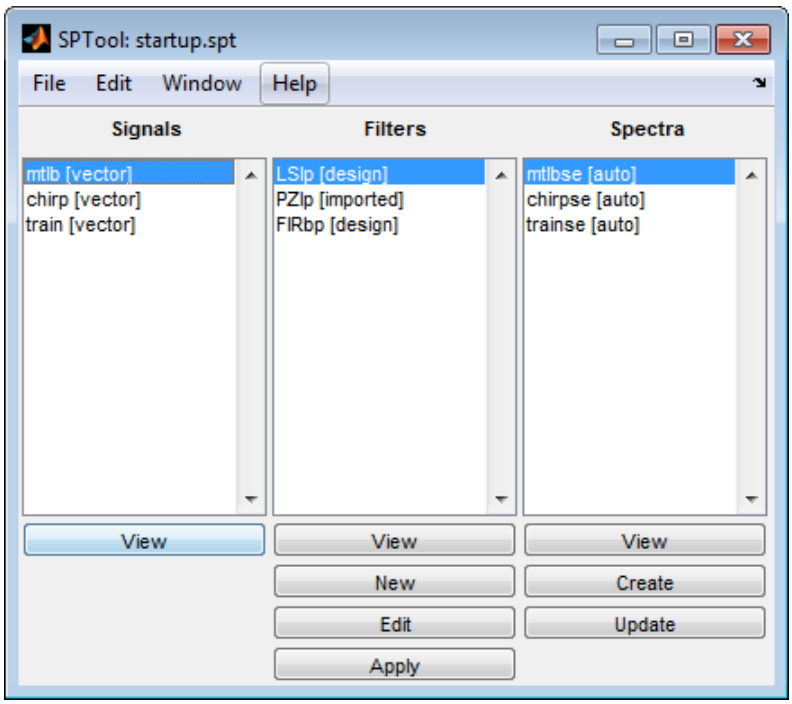
You can bring signals, filters, or spectra from the MATLAB workspace into the SPTool workspace using **File > Import**. For more information, see “Importing Filters and Spectra” on page 9-22. Signals, filters, or spectra that you create in (or import into) the SPTool workspace exist as MATLAB structures. See the MATLAB documentation for more information on MATLAB structures.

When you use **File > Export** to save signals, filters, and spectra that you create or modify in SPTool, these are also saved as MATLAB structures. For more information on exporting, see “Exporting Signals, Filters, and Spectra” on page 9-19.

# Opening SPTool

To open SPTool, type

```
sptool
```




When you first open SPTool, it contains a collection of default signals, filters, and spectra. To specify your own preferences for what signals, filters, and spectra to see when SPTool opens see “Setting Preferences” on page 9-32.

You can access these three GUIs from SPTool by selecting a signal, filter, or spectrum and clicking the appropriate **View** button:

- Signal Browser on page 9-5
- Filter Visualization Tool on page 9-7
- Spectrum Viewer on page 9-9

You can access **Filter Designer** by clicking **New** to create a new filter or `Edit` to edit a selected filter. Clicking **Apply** applies a selected filter to a selected signal.

**Create** opens the Spectrum Viewer and creates the power spectral density of the selected signal. **Update** opens the Spectrum Viewer for the selected spectrum.

# Getting Context-Sensitive Help

To find information on a particular feature or setting of the “Signal Browser” on page 9-5:

- In any Measurements panel, right-click anywhere on the panel and select **What's this?**.
- In any dialog box where you see the icon in the lower left corner, right-click on any parameter and select **What's this?**.

To find information on a particular region of **Filter Designer** or “Spectrum Viewer” on page 9-9:

1. Click **What's this?** .
2. Click on the region of the GUI you want information on.

You can also use **Help > What's This?** to launch context-sensitive help.

# Signal Browser

| In this section... |
| --- |
| “Overview of the Signal Browser” on page 9-5<br>“Opening the Signal Browser” on page 9-5 |

## Overview of the Signal Browser

You can use the Signal Browser to display and analyze signals listed in the **Signals** list box in SPTool.

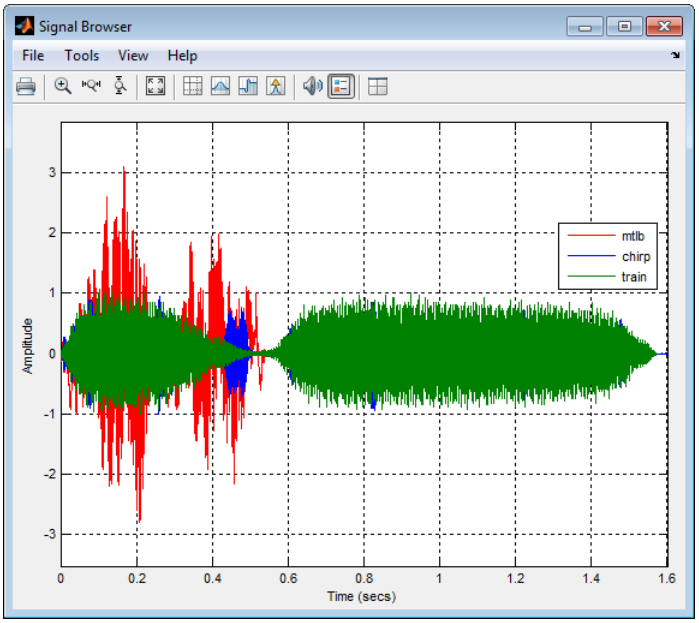
Using the Signal Browser, you can:

- Analyze and compare vector or array (matrix) signals.
- Zoom in on portions of signal data.
- Measure a variety of characteristics of signal data.
- Compare multiple signals.
- Play portions of signal data on audio hardware.
- Print signal plots.

## Opening the Signal Browser

To open the Signal Browser from SPTool:

1. Select one or more signals in the **Signals** list in SPTool.
2. Click **View** under the **Signals** list.




The Signal Browser has the following components:

- A display region for analyzing signals
- A panels section on the right side of the scope window, which shows statistics and information about your signals
- A toolbar with buttons for convenient access to frequently used functions

For more information on the Signal Browser, see the `sptool` function reference page.

# Filter Visualization Tool

| In this section... |
| --- |
| “Connection between FVTool and SPTool” on page 9-7<br>“Opening the Filter Visualization Tool” on page 9-7<br>“Analysis Parameters” on page 9-8 |

## Connection between FVTool and SPTool

You can use the Filter Visualization Tool to analyze response characteristics of the selected filter(s). See **FVTool** for detailed information about FVTool.

If you start FVTool by clicking the SPTool **Filter View** button, that FVTool is linked to SPTool. Any changes made in SPTool to the filter are immediately reflected in FVTool. The FVTool title bar includes "SPTool" to indicate the link.

If you start an FVTool by clicking the **New** button or by selecting **File > New** from within FVTool, that FVTool is a standalone version and is not linked to SPTool.

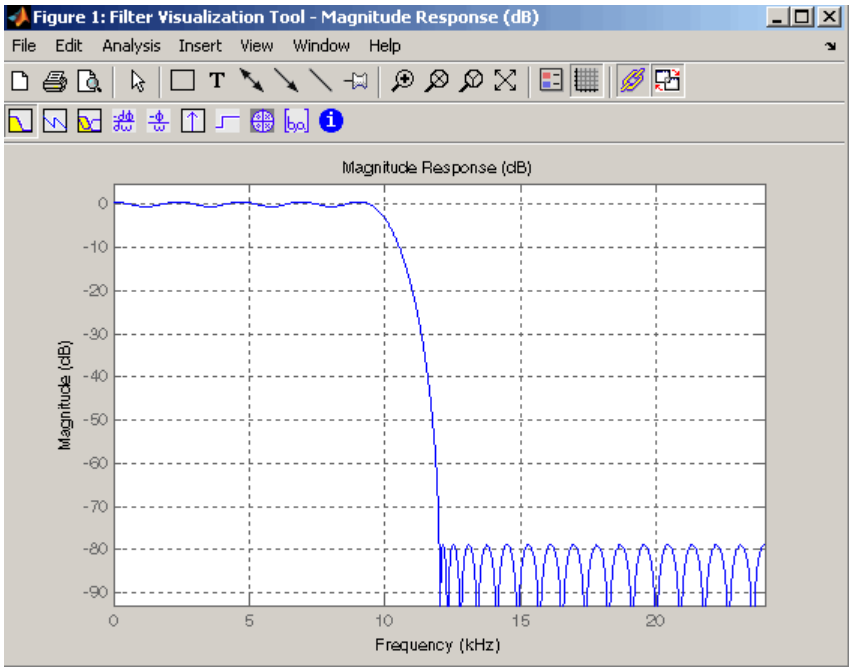
**Note** Every time you click the **Filter View** button a new, linked FVTool starts. This allows you to view multiple analyses simultaneously.

## Opening the Filter Visualization Tool

You open FVTool from SPTool as follows.

1. Select one or more filters in the **Filters** list in SPTool.
2. Click the **View** button under the **Filters** list.

When you first open FVTool, it displays the selected filter's magnitude plot.




## Analysis Parameters

In the plot area of any filter response plot, right-click and select **Analysis Parameters** to display details about the displayed plot. See “Analysis Parameters” in the **Filter Designer** online help for more information.

You can change any parameter in a linked FVTool, except the sampling frequency. You can only change the sampling frequency using the SPTool **Edit > Sampling Frequency** or the SPTool **Filters Edit** button.

# Spectrum Viewer

| In this section... |
| --- |
| “Spectrum Viewer Overview” on page 9-9<br>“Opening the Spectrum Viewer” on page 9-9 |

## Spectrum Viewer Overview

You can use the Spectrum Viewer for estimating and analyzing a signal's power spectral density (PSD). You can use the PSD estimates to understand a signal's frequency content.

The Spectrum Viewer provides the following functionality.

- Analyze and compare spectral density plots.
- Use different spectral estimation methods to create spectra:
  - Burg (`pburg`)
  - Covariance (`pcov`)
  - FFT (`fft`)
  - Modified covariance (`pmcov`)
  - MTM (multitaper method) (`pmtm`)
  - MUSIC (`pmusic`)
  - Welch (`pwelch`)
  - Yule-Walker AR (`pyulear`)
- Modify power spectral density parameters such as FFT length, window type, and sample frequency.
- Print spectral plots.

## Opening the Spectrum Viewer

To open the Spectrum Viewer and create a PSD estimate from SPTool:

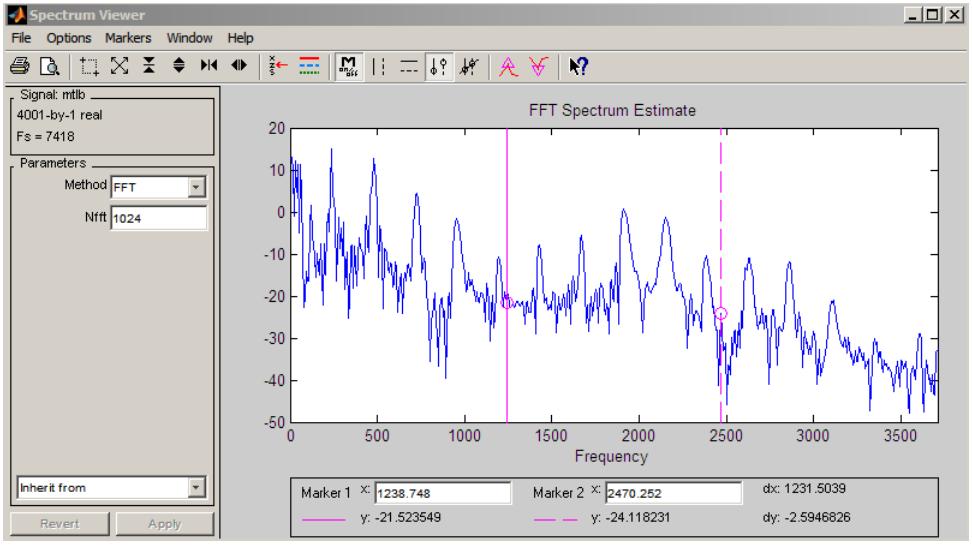
1. Select a signal from the **Signal** list box in SPTool.
2. Click **Create** in the **Spectra** list.
3. Click **Apply** in the Spectrum Viewer.

To open the Spectrum Viewer with a PSD estimate already listed in SPTool:

1. Select a PSD estimate from the **Spectra** list box in SPTool.
2. Click **View** in the Spectra list.

For example:

1. Select `mtlb` in the default **Signals** list in SPTool.
2. Click **Create** in SPTool to open the Spectrum Viewer.
3. Click **Apply** in the Spectrum Viewer to plot the spectrum.




The Spectrum Viewer has the following components:

- A signal identification region that provides information about the signal whose power spectral density estimate is displayed
- A Parameters region for modifying the PSD parameters
- A display region for analyzing spectra and an **Options** menu for modifying display characteristics
- Spectrum management controls
  - **Inherit from** menu to inherit PSD specifications from another PSD object listed in the menu
  - Revert button to revert to the named PSD's original specifications
  - **Apply** button for creating or updating PSD estimates
- A toolbar with buttons for convenient access to frequently used functions

| Icon | Description |
|---|---|
| | Print and print preview |
| | Zoom the signal in and out |
| | Select one of several loaded signals |
| | Set the display color and line style of a signal |
| | Toggle the markers on and off |
| | Set marker types |
| | Turn on the What's This help |

# Filtering and Analysis of Noise

| In this section... |
| --- |
| “Overview” on page 9-11 |
| “Importing a Signal into SPTool” on page 9-11 |
| “Designing a Filter” on page 9-12 |
| “Applying a Filter to a Signal” on page 9-14 |
| “Analyzing a Signal” on page 9-15 |
| “Spectral Analysis in the Spectrum Viewer” on page 9-17 |

## Overview

The following sections provide an example of using the GUI-based interactive tools to:

- Design and implement an FIR bandpass digital filter
- Apply the filter to a noisy signal
- Analyze signals and their spectra

The steps include:

1. “Importing a Signal into SPTool” on page 9-11
2. Designing a bandpass filter using Filter Designer on page 9-12
3. Applying the filter to the original noise signal to create a bandlimited noise signal on page 9-14
4. Comparing the time domain information of the original and filtered signals using the Signal Browser on page 9-15
5. Comparing the spectra of both signals using the Spectrum Viewer on page 9-17

## Importing a Signal into SPTool

To import a signal into SPTool from the workspace or disk, the signal must be either:

- A special MATLAB signal structure, such as that saved from a previous SPTool session
- A signal created as a variable (vector or matrix) in the MATLAB workspace

For this example, create a new signal at the command line and then import it as a structure into SPTool:

1. Create a random signal in the MATLAB workspace by typing

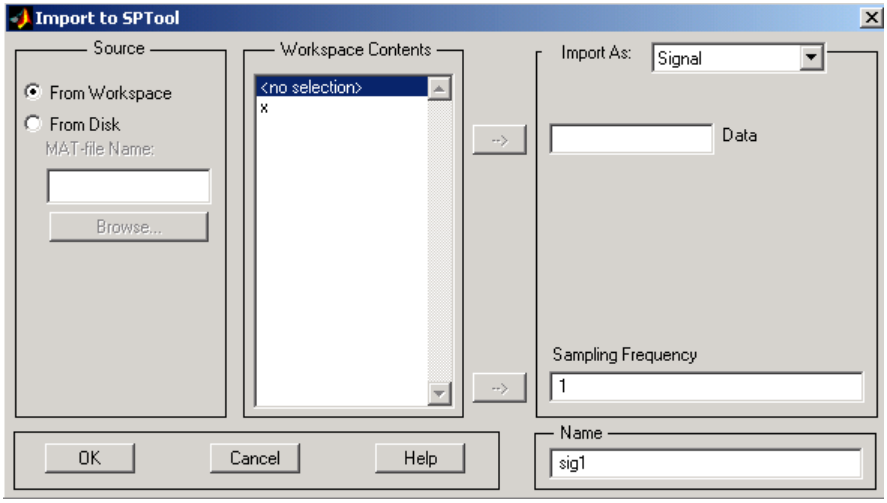
   ```
   x = randn(5000,1);
   ```

2. If SPTool is not already open, open SPTool by typing

   ```
   sptool
   ```

   The SPTool window is displayed.

3. Select **File > Import**. The Import to SPTool dialog opens.




The variable `x` is displayed in the **Workspace Contents** list. (If it is not, select the **From Workspace** radio button to display the contents of the workspace.)

4 Select the signal and import it into the **Data** field:

   a Select the signal variable `x` in the **Workspace Contents** list.

   b Make sure that `Signal` is selected in the **Import As** pull-down menu.

   c Click on the arrow to the left of the **Data** field or type `x` in the **Data** field.

   d Type `5000` in the **Sampling Frequency** field.

   e Name the signal by typing `noise` in the **Name** field.

   f Click **OK**.

   The signal `noise[vector]` appears and is selected in SPTool's **Signals** list.

---

**Note** You can import filters on page 9-22 and spectra on page 9-23 into SPTool in much the same way as you import signals. See “Importing Filters and Spectra” on page 9-22 for specific details.

You can also import signals from MAT-files on your disk, rather than from the workspace. See “Loading Variables from the Disk” on page 9-25 for more information.

Type `help sptool` for information about importing from the command line.

---

## Designing a Filter

You can import an existing filter into SPTool, or you can design and edit a new filter using **Filter Designer**.

In this example, you

1 Open a default filter in **Filter Designer**.

2 Specify an equiripple bandpass FIR filter.

### Opening Filter Designer

To open **Filter Designer**, click **New** in SPTool. **Filter Designer** opens with a default filter named `filt1`.

### Specifying the Bandpass Filter

Design an equiripple bandpass FIR filter with the following characteristics:

- Sampling frequency of 5000 Hz
- Stopband frequency ranges of [0 500] Hz and [1500 2500] Hz
- Passband frequency range of [750 1250] Hz
- Ripple in the passband of 0.01 dB
- Stopband attenuation of 75 dB

To modify the filter in **Filter Designer** to meet these specifications, you need to

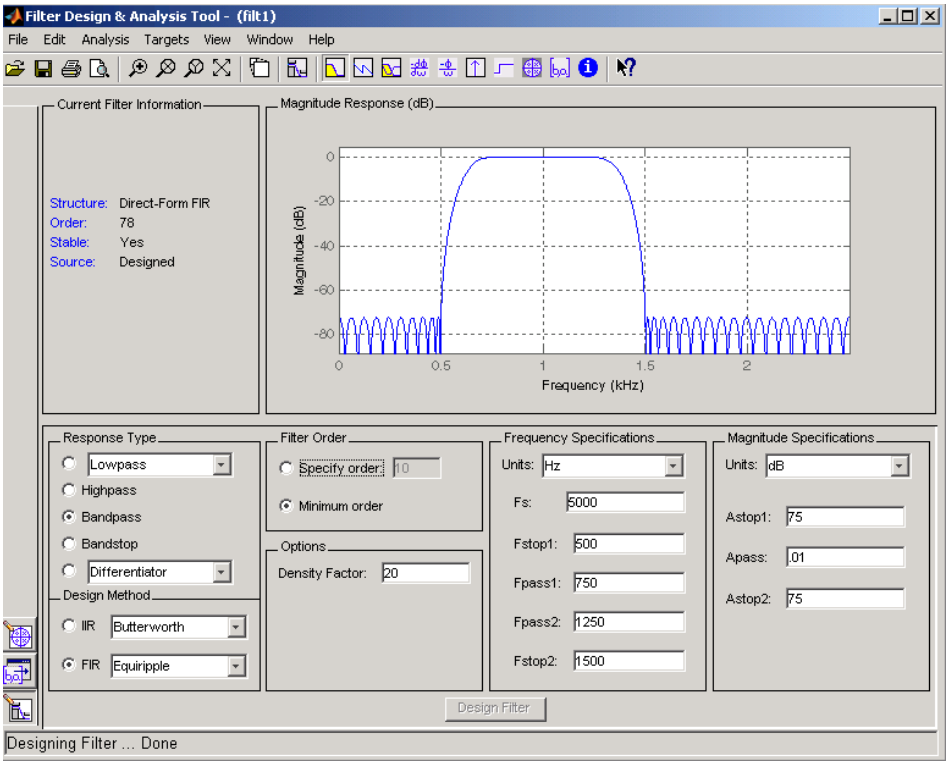
1. Select `Bandpass` from the **Response Type** list.
2. Verify that **FIR Equiripple** is selected as the **Design Method**.
3. Verify that **Minimum order** is selected as the **Filter Order** and that the **Density Factor** is set to `20`.
4. Under **Frequency Specifications**, set the sampling frequency (**Fs**) and the passband (**Fpass1**, **Fpass2**) and stopband (**Fstop1**, **Fstop2**) edges:

| **Units** | **Hz** |
|---|---|
| **Fs** | `5000` |
| **Fstop1** | `500` |
| **Fpass1** | `750` |
| **Fpass2** | `1250` |
| **Fstop2** | `1500` |

5. Under **Magnitude Specifications**, set the stopband attenuation (**Astop1**, **Astop2**) and the maximum passband ripple (**Apass**):

| **Units** | **dB** |
|---|---|
| **Astop1** | `75` |
| **Apass** | `0.01` |
| **Astop2** | `75` |

6. Click **Design Filter** to design the new filter. When the new filter is designed, the magnitude response of the filter is displayed.



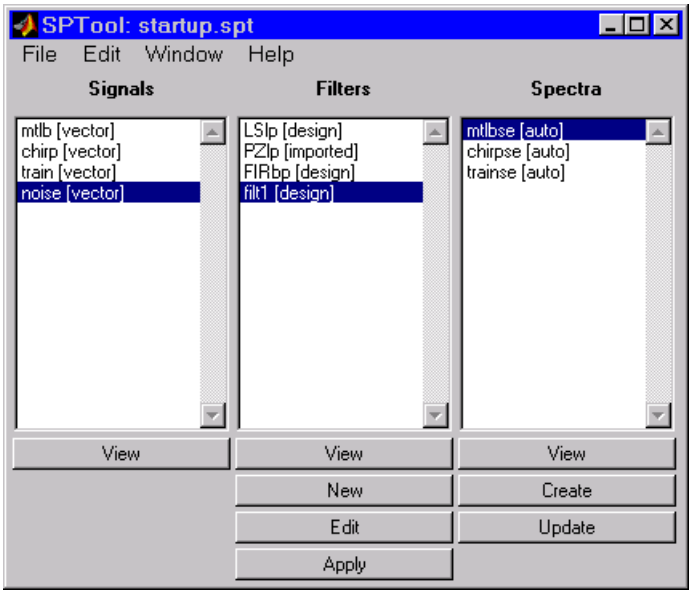

The resulting filter is an order-78 bandpass equiripple filter.

## Applying a Filter to a Signal

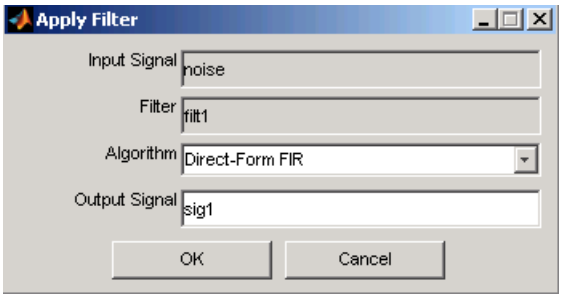
When you apply a filter to a signal, you create a new signal in SPTool representing the filtered signal.

To apply the filter `filt1` you just created to the signal `noise`,

1. In SPTool, select the signal `noise[vector]` from the **Signals** list and select the filter (named `filt1[design]`) from the **Filters** list.




2 Click **Apply** under the **Filters** list.




3 Leave the **Algorithm** as `Direct-Form FIR`.

---

**Note** You can apply one of two filtering algorithms to FIR filters. The default algorithm is specific to the filter structure, which is shown in the **Filter Designer** Current Filter Info frame. Alternately for FIR filters, `FFT based FIR (fftfilt)` uses the algorithm described in `fftfilt`.

For IIR filters, the alternate algorithm is a zero-phase IIR that uses the algorithm described in `filtfilt`.

---

4 Enter `blnoise` as the **Output Signal** name.

5 Click **OK** to close the Apply Filter dialog box.

The filter is applied to the selected signal, and the filtered signal `blnoise[vector]` is listed in the **Signals** list in SPTool.

## Analyzing a Signal

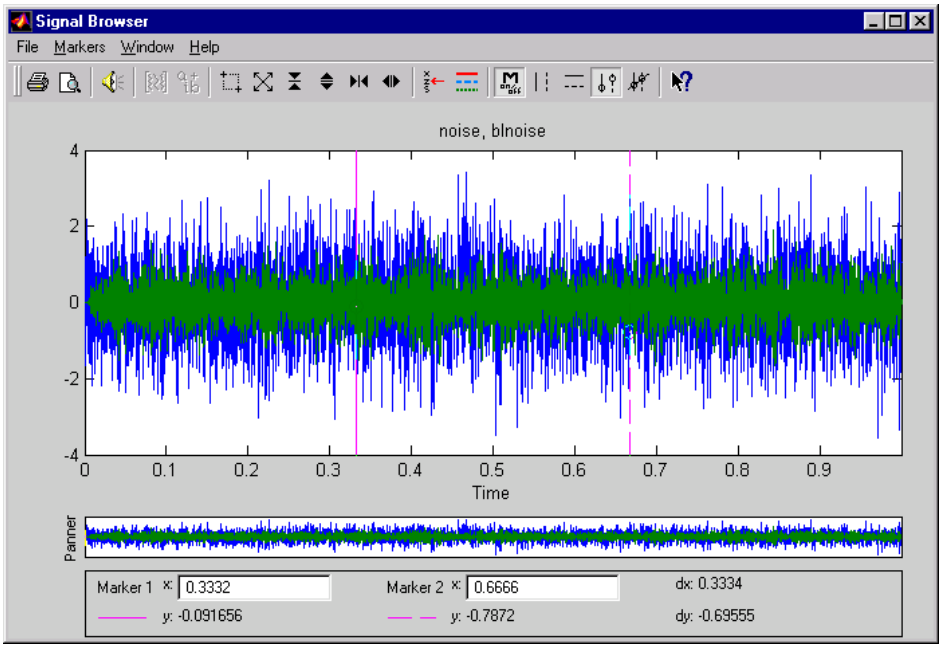
You can analyze and print signals using the Signal Browser. You can also play the signals if your computer has audio output capabilities.

For example, compare the signal `noise` on page 9-11 to the filtered signal `blnoise` on page 9-14:

1. **Shift**+click on the `noise` and `blnoise` signals in the **Signals** list of SPTool to select both signals.
2. Click **View** under the **Signals** list.

   The Signal Browser is activated, and both signals are displayed in the display region. (The names of both signals are shown above the display region.) Initially, the original `noise` signal covers up the bandlimited `blnoise` signal.
3. Push the selection button on the toolbar to select the `blnoise` signal.

   The display area is updated. Now you can see the `blnoise` signal superimposed on top of the `noise` signal. The signals are displayed in different colors in both the display region and the panner. You can change the color of the selected signal using the **Line Properties** button on the toolbar, .




### Playing a Signal

When you click **Play** in the Signal Browser toolbar, , the active signal is played on the computer's audio hardware.

1. To hear a portion of the active (selected) signal
   a. Use the vertical markers on page 9-30to select a portion of the signal you want to play.

      Vertical markers are enabled by the and buttons.
   b. Click **Play**.
2. To hear the other signal

a Select the signal using the selection button on the toolbar. You can also select the signal directly in the display region.

b Click **Play** again.

#### Printing a Signal

You can print from the Signal Browser using the **Print** button, .

You can use the line display buttons to maximize the visual contrast between the signals by setting the line color for noise on page 9-11 to gray and the line color for blnoise on page 9-11 to white. Do this before printing two signals together.

---

**Note** You can follow the same rules to print spectra, but you can't print filter responses directly from SPTool.

---

Use the Signal Browser region in the Preferences dialog box in SPTool to suppress printing of both the panner and the marker settings.

To print both signals, click **Print** in the Signal Browser toolbar.

## Spectral Analysis in the Spectrum Viewer

You can analyze the frequency content of a signal using the Spectrum Viewer, which estimates and displays a signal's power spectral density.

For example, to analyze and compare the spectra of `noise` and `blnoise:`

1. Create a power spectral density (PSD) object on page 9-17, `spect1`, that is associated with the signal `noise`, and a second PSD object, spect2, that is associated with the signal `blnoise`.
2. Open the Spectrum Viewer to analyze both of these spectra on page 9-17.
3. Print both spectra on page 9-17.

#### Creating a PSD Object From a Signal

1. Click on SPTool, or select **Window > SPTool** in any active open GUI. SPTool is now the active window.
2. Select the `noise[vector]` on page 9-11 signal in the **Signals** list of SPTool.
3. Click **Create** in the **Spectra** list.

   The Spectrum Viewer is activated, and a PSD (`spect1`) corresponding to the `noise` signal is created in the **Spectra** list. The PSD is not computed or displayed yet.
4. Click **Apply** in the Spectrum Viewer to compute and display the PSD estimate `spect1` using the default parameters.

   The PSD of the `noise` signal is displayed in the display region. The identifying information for the PSD's associated signal (`noise`) is displayed above the Parameters region.

   The PSD estimate `spect1` is within 2 or 3 dB of 0, so the noise has a fairly "flat" power spectral density.

5 Follow steps 1 through 4 for the bandlimited noise signal blnoise on page 9-14 to create a second PSD estimate `spect2`.

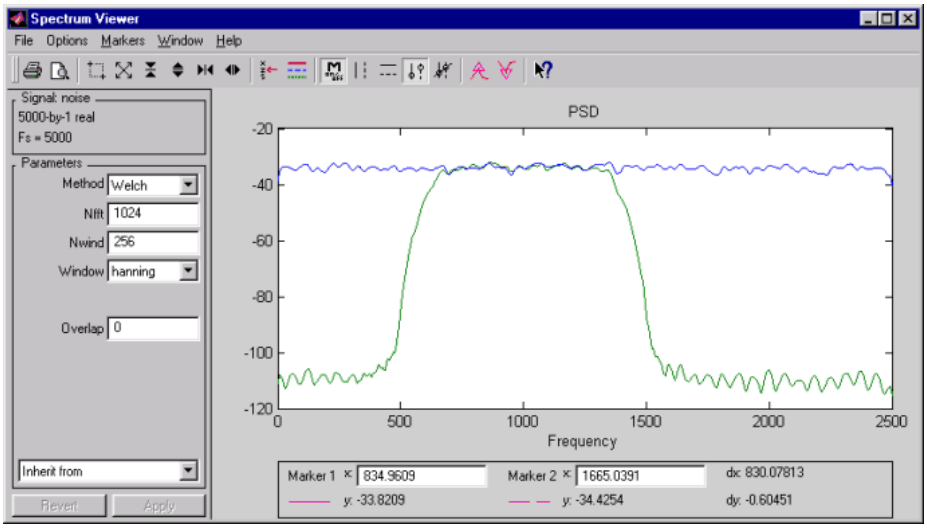
The PSD estimate `spect2` is flat between 750 and 1250 Hz and has 75 dB less power in the stopband regions of `filt1`.

### Opening the Spectrum Viewer with Two Spectra

1 Reactivate SPTool again, as in step 1 above.

2 **Shift**+click on `spect1` and `spect2` in the **Spectra** list to select them both.

3 Click **View** in the **Spectra** list to reactivate the Spectrum Viewer and display both spectra together.




### Printing the Spectra

Before printing the two spectra together, use the color and line style selection button, , to differentiate the two plots by line style, rather than by color.

To print both spectra:

1 Click **Print Preview** in the toolbar on the Spectrum Viewer.

2 From the Spectrum Viewer Print Preview window, drag the legend out of the display region so that it doesn't obscure part of the plot.

3 Click **Print** in the Spectrum Viewer Print Preview window.

# Exporting Signals, Filters, and Spectra

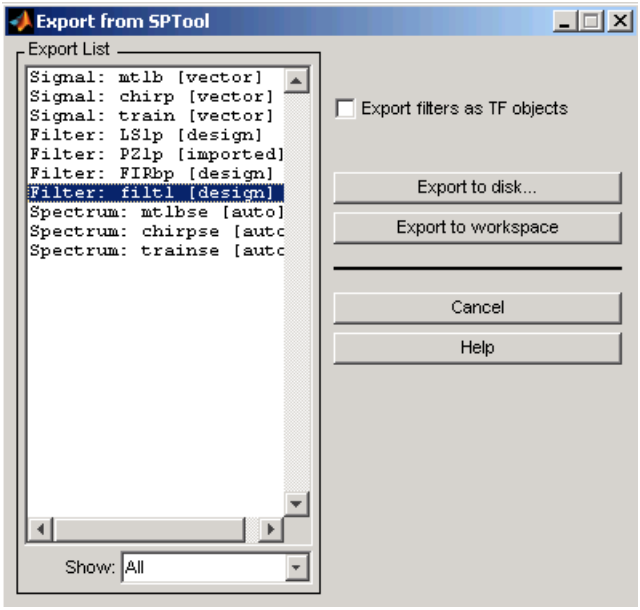
| In this section... |
| --- |
| “Opening the Export Dialog Box” on page 9-19<br>“Exporting a Filter to the MATLAB Workspace” on page 9-19 |

## Opening the Export Dialog Box

To save the filter `filt1` you just created in this example, open the Export dialog box with `filt1` preselected:

1. Select `filt1` in the SPTool **Filters** list.
2. Select **File > Export**.

The Export dialog box opens with `filt1` preselected.




## Exporting a Filter to the MATLAB Workspace

To export the filter `filt1` to the MATLAB workspace:

1. Select `filt1` from the **Export List** and deselect all other items using **Ctrl**+click.
2. Click **Export to Workspace**.

# Accessing Filter Parameters

| In this section... |
| --- |
| “Accessing Filter Parameters in a Saved Filter” on page 9-20 |
| “Accessing Parameters in a Saved Spectrum” on page 9-20 |

## Accessing Filter Parameters in a Saved Filter

The MATLAB structures created by SPTool have several associated fields, many of which are also MATLAB structures. See the MATLAB documentation for general information about MATLAB structures.

For example, after exporting on page 9-19 a filter filt1 to the MATLAB workspace, type

```
filt1
```

to display the fields of the MATLAB filter structure. The `tf` field of the structure contains information that describes the filter.

### The tf Field: Accessing Filter Coefficients

The `tf` field is a structure containing the transfer function representation of the filter. Use this field to obtain the filter coefficients;

- `filt1.tf.num` contains the numerator coefficients.
- `filt1.tf.den` contains the denominator coefficients.

The vectors contained in these structures represent polynomials in descending powers of $z$. The numerator and denominator polynomials are used to specify the transfer function

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(nb+1)z^{-m}}{a(1) + a(2)z^{-1} + \cdots + a(na+1)z^{-n}}$$

where:

- $b$ is a vector containing the coefficients from the `tf.num` field.
- $a$ is a vector containing the coefficients from the `tf.den` field.
- $m$ is the numerator order.
- $n$ is the denominator order.

You can change the filter representation from the default transfer function to another form by using the `tf2ss` or `tf2zp` functions.

**Note** The `FDAspecs` field of your filter contains internal information about **Filter Designer** and should not be changed.

## Accessing Parameters in a Saved Spectrum

The following structure fields describe the spectra saved by SPTool.

| Field | Description |
|---|---|
| `P` | The spectral power vector. |
| `f` | The spectral frequency vector. |
| `confid` | A structure containing the confidence intervals data<br>• The `confid.level` field contains the chosen confidence level.<br>• The `confid.Pc` field contains the spectral power data for the confidence intervals.<br>• The `confid.enable` field contains a `1` if confidence levels are enabled for the power spectral density. |
| `signalLabel` | The name of the signal from which the power spectral density was generated. |
| `Fs` | The associated signal's sample rate. |

You can access the information in these fields as you do with every MATLAB structure.

For example, if you export an SPTool PSD estimate `spect1` to the workspace, type

```
spect1.P
```

to obtain the vector of associated power values.

# Importing Filters and Spectra

| In this section... |
| --- |
| “Similarities to Other Procedures” on page 9-22<br>“Importing Filters” on page 9-22<br>“Importing Spectra” on page 9-23 |

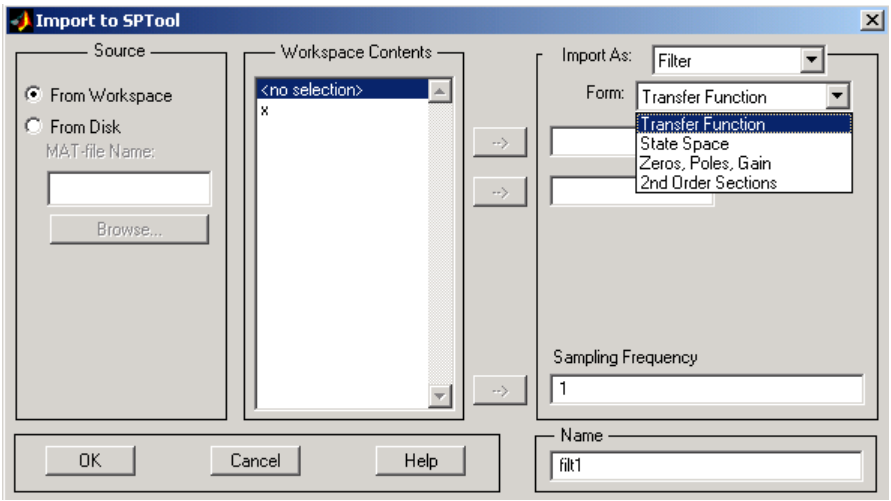
## Similarities to Other Procedures

The procedures are very similar to those explained in

- “Importing a Signal into SPTool” on page 9-11 for loading variables from the workspace
- “Loading Variables from the Disk” on page 9-25 for loading variables from your disk

## Importing Filters

When you import filters, first select the appropriate filter form from the **Form** list. SPTool does not currently support the import of filter objects.




For every filter you specify a variable name or a value for the filter's sampling frequency in the **Sampling Frequency** field. Each filter form requires different variables.

### Transfer Function

For `Transfer Function`, you specify the filter by its transfer function representation:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(m+1)z^{-m}}{a(1) + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}$$

- The **Numerator** field specifies a variable name or value for the numerator coefficient vector $b$, which contains $m$+1 coefficients in descending powers of $z$.
- The **Denominator** field specifies a variable name or value for the denominator coefficient vector $a$, which contains $n$+1 coefficients in descending powers of $z$.

#### State Space

For `State Space`, you specify the filter by its state-space representation:

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

The **A-Matrix**, **B-Matrix**, **C-Matrix**, and **D-Matrix** fields specify a variable name or a value for each matrix in this system.

#### Zeros, Poles, Gain

For `Zeros`, `Poles`, `Gain`, you specify the filter by its zero-pole-gain representation:

$$H(z) = \frac{Z(z)}{P(z)} = k\frac{(z - z(1))(z - z(2))\cdots(z - z(m))}{(z - p(1))(z - p(2))\cdots(z - p(n))}$$

- The **Zeros** field specifies a variable name or value for the zeros vector $z$, which contains the locations of $m$ zeros.
- The **Poles** field specifies a variable name or value for the zeros vector $p$, which contains the locations of $n$ poles.
- The **Gain** field specifies a variable name or value for the gain $k$.

#### Second Order Sections

For `2nd Order Sections` you specify the filter by its second-order section representation:

$$H(z) = \prod_{k=1}^{L} H_k(z) = \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

The **SOS Matrix** field specifies a variable name or a value for the $L$-by-6 SOS matrix

$$\text{sos} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

whose rows contain the numerator and denominator coefficients $b_{ik}$ and $a_{ik}$ of the second-order sections of $H(z)$.

---

**Note** If you import a filter that was not created in SPTool, you can only edit that filter using the Pole-Zero Editor.

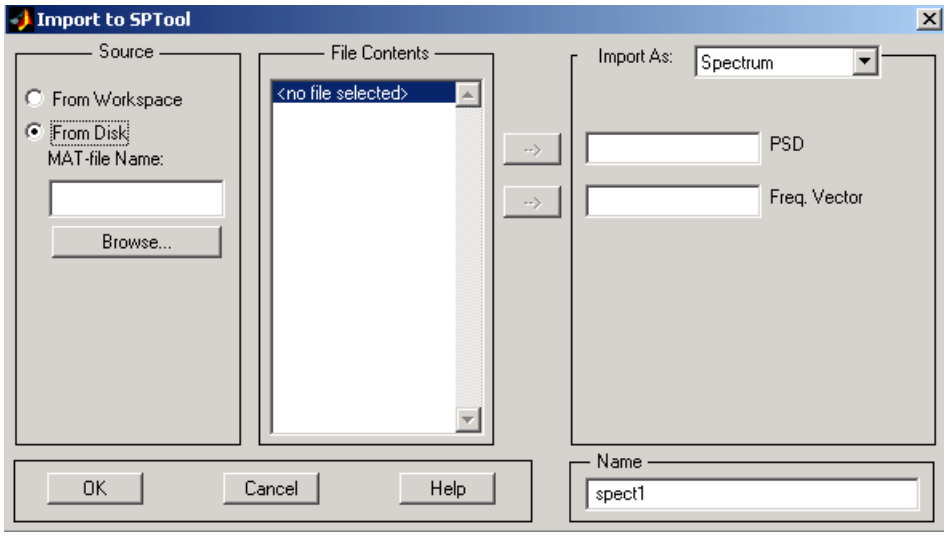
---

## Importing Spectra

When you import a power spectral density (PSD), you specify:

- A variable name or a value for the PSD vector in the **PSD** field
- A variable name or a value for the frequency vector in the **Freq. Vector** field

The PSD values in the **PSD** vector correspond to the frequencies contained in the **Freq. Vector** vector; the two vectors must have the same length.



# Loading Variables from the Disk

To import variables representing signals, filters, or spectra from a MAT-file on your disk;

1. Select the **From Disk** radio button and do either of the following:

   - Type the name of the file you want to import into the **MAT-file Name** field and press either the **Tab** or the **Enter** key on your keyboard.
   - Select **Browse**, and then find and select the file you want to import using **Select** > **File to Open**. Click **OK** to close that dialog.

   In either case, all variables in the MAT-file you selected are displayed in the **File Contents** list.
2. Select the variables to be imported into SPTool.

You can now import one or more variables from the **File Contents** list into SPTool, as long as these variables are scalars, vectors, or matrices.

# Saving and Loading Sessions

| In this section... |
| --- |
| “SPTool Sessions” on page 9-26<br>“Filter Formats” on page 9-26 |

## SPTool Sessions

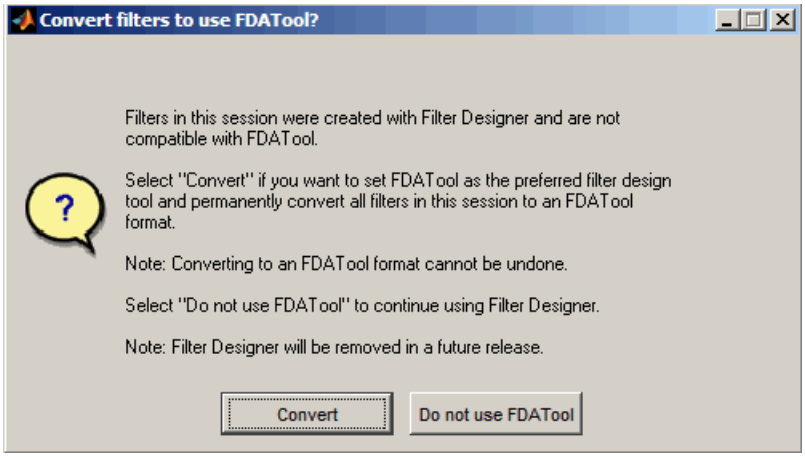
When you start SPTool, the default `startup.spt` session is loaded. To save your work in the startup SPTool session, use **File > Save Session** or to specify a session name, use **File > Save Session As**.

To recall a previously saved session, use **File > Open Session**.

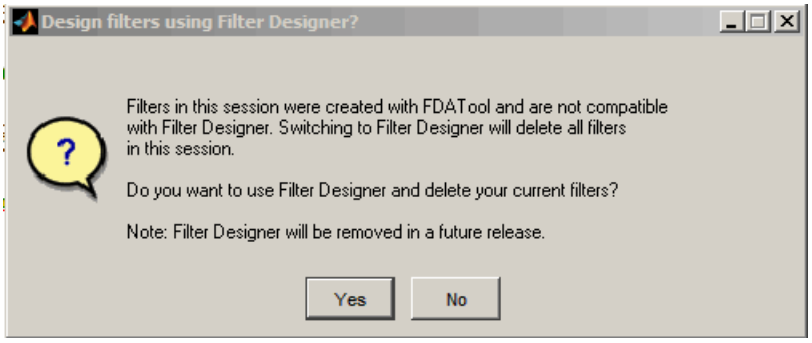
## Filter Formats

When you start SPTool or open a session, the current filter design format preference is compared to the filter formats in the session. See “Setting Preferences” on page 9-32.

- If the formats match, the session opens.
- If the filter preference is **Filter Designer**, but the session contains Filter Designer filters, this warning displays:




  Click **Convert** to convert the filters to **Filter Designer** format. Click **Don't Use Filter Designer** to leave the filters in **Filter Designer** format and change the preference to **Use Filter Designer**.
- If the filter preference is **Use Filter Designer**, but the session contains **Filter Designer** filters, this warning displays:




Click **Yes** to remove the current filters. Click **No** to leave the filters in **Filter Designer**.

# Selecting Signals, Filters, and Spectra

All signals, filters, or spectra listed in SPTool exist as special MATLAB structures. You can bring data representing signals, filters, or spectra into SPTool from the MATLAB workspace. In general, you can select one or several items in a given list box. An item is selected when it is highlighted.

The **Signals** list shows all vector and array signals in the current SPTool session.

The **Filters** list shows all designed and imported filters in the current SPTool session.

The **Spectra** list shows all spectra in the current SPTool session.

You can select a single data object in a list, a range of data objects in a list, or multiple separate data objects in a list. You can also have data objects simultaneously selected in different lists:

- To select a single item, click it. All other items in that list box become deselected.
- To add or remove a range of items, **Shift**+click on the items at the top and bottom of the section of the list that you want to add. You can also drag your mouse pointer to select these items.
- To add a single data object to a selection or remove a single data object from a multiple selection, **Ctrl**+click on the object.

# Editing Signals, Filters, or Spectra

You can edit selected items in SPTool by

1. Selecting the names of the signals, filters, or spectra you want to edit.
2. Selecting the appropriate **Edit** menu item:

   - **Duplicate** to copy an item in an SPTool list
   - **Clear** to delete an item in an SPTool list
   - **Name** to rename an item in an SPTool list
   - **Sampling Frequency** to modify the sampling frequency associated with either a signal (and its associated spectra) or filter in an SPTool list

   The pull-down menu next to each menu item shows the names of all selected items.

You can also edit the following signal characteristics by right-clicking in the display region of the Signal Browser, the Filter Visualization Tool, or the Spectrum Viewer:

- The signal name
- The sampling frequency
- The line style properties

---

**Note** If you modify the sample rate associated with a signal's spectrum using the right-click menu on the Spectrum Viewer display region, the sample rate of the associated signal is automatically updated.

---

# Making Signal Measurements with Markers

You can use the markers on the Signal Browser on page 9-5 or the Spectrum Viewer on page 9-5 to make measurements on either of the following:

- A signal in the Signal Browser
- A power spectral density plotted in the Spectrum Viewer

The following marker buttons are included

| Icon | Description |
|---|---|
| | Toggle markers on/off |
| | Vertical markers |
| | Horizontal markers |
| | Vertical markers with tracking |
| | Vertical markers with tracking and slope |
| | Display peaks (local maxima)<br><br>You can find peaks in a signal from the command line with `findpeaks` |
| | Display valleys (local minima) |

To make a measurement:

1. Select a line to measure (or play, if you are in the Signal Browser).
2. Select one of the marker buttons to apply a marker to the displayed signal.
3. Position a marker in the main display area by grabbing it with your mouse and dragging:

    a. Select a marker setting. If you choose the **Vertical**, **Track**, or **Slope** buttons, you can drag a marker to the right or left. If you choose the **Horizontal** button, you can drag a marker up or down.

    b. Move the mouse over the marker (1 or 2) that you want to drag.

    The hand cursor with the marker number inside it is displayed when your mouse passes over a marker.

    c. Drag the marker to where you want it on the signal

    As you drag a marker, the bottom of the Signal Browser shows the current position of both markers. Depending on which marker setting you select, some or all of the following fields are displayed — **x1**, **y1**, **x2**, **y2**, **dx**, **dy**, **m**. These fields are also displayed when you print from the Signal Browser, unless you suppress them.

You can also position a marker by typing its **x1** and **x2** or **y1** and **y2** values in the region at the bottom.

# Setting Preferences

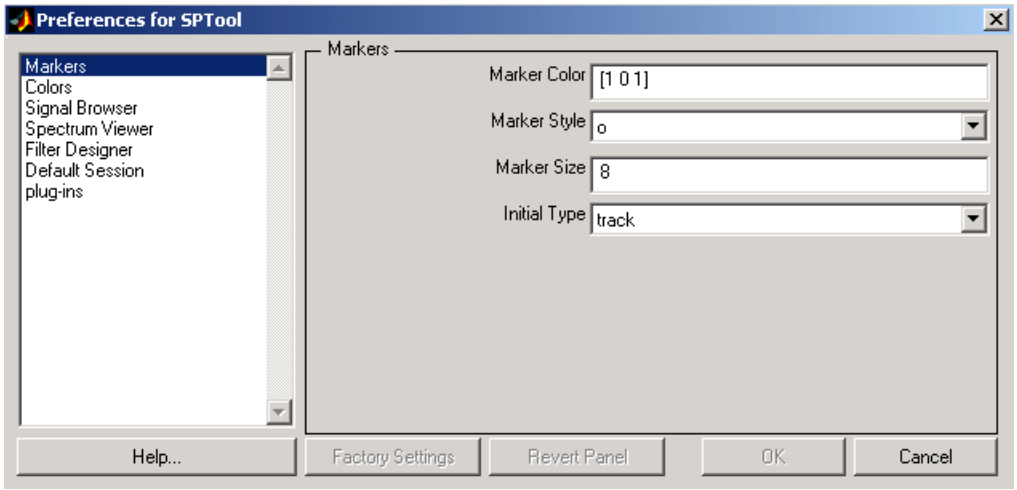
| In this section... |
| --- |
| “Overview of Setting Preferences” on page 9-32<br>“Summary of Settable Preferences” on page 9-32 |

## Overview of Setting Preferences

Use **File > Preferences** to customize displays and certain parameters for SPTool and its four component GUIs. If you change any preferences, a dialog box displays when you close SPTool asking if you want to save those changes. If you click **Yes**, the new settings are saved on disk and are used when you restart SPTool from the MATLAB workspace.

**Note** You can set MATLAB preferences that affect the Filter Visualization Tool only from within FVTool by selecting **File > Preferences**. You can set FVTool-specific preferences using **Analysis > Analysis Parameters**.

When you first select **Preferences**, the Preferences dialog box opens with **Markers** selected by default.




Change any marker settings, if desired. To change settings for another category, click its name in the category list to display its settings. Most of the fields are self-explanatory. Details of the Filter Design options are described below.

## Summary of Settable Preferences

In the Preferences regions, you can

- Select colors and markers for all displays.
- Select colors and line styles for displayed signals.
- Configure labels, and enable/disable markers, panner, and zoom in the Signal Browser on page 9-5.

- Configure display parameters, and enable/disable markers and zoom in the Spectrum Viewer on page 9-9.
- Enable/disable use of a default session file.
- Export filters for use with Control System Toolbox software.
- Enable/disable search for plug-ins at startup.

# Code Generation from MATLAB Support in Signal Processing Toolbox


- “List of Signal Processing Toolbox Functions that Support Code Generation” on page 10-2
- “Specifying Inputs in Code Generation from MATLAB” on page 10-9
- “Apply Lowpass Filter to Input Signal” on page 10-12
- “Zero-Phase Filtering” on page 10-14
- “Compute Modified Periodogram Using Generated C Code” on page 10-16

# List of Signal Processing Toolbox Functions that Support Code Generation

Code generation from MATLAB is a restricted subset of the MATLAB language that provides optimizations for:

- Generating efficient, production-quality C/C++ code and MEX files for deployment in desktop and embedded applications. For embedded targets, the subset restricts MATLAB semantics to meet the memory and data type requirements of the target environments.

Depending on which feature you wish to use, there are additional required products. For a comprehensive list, see “Installing Prerequisite Products” (MATLAB Coder).

Code generation from MATLAB supports Signal Processing Toolbox functions listed in the table. To generate C code, you must have the MATLAB Coder™ software. If you have the Fixed-Point Designer software, you can use `fiaccel` to generate MEX code for fixed-point applications.

To follow the examples in this documentation:

- To generate C/C++ code and MEX files with `codegen`, install the MATLAB Coder software, the Signal Processing Toolbox, and a C compiler. For the Windows® platform, MATLAB supplies a default C compiler. Run `mex -setup` at the MATLAB command prompt to set up the C compiler.
- Change to a folder where you have write permission.

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

| | |
|---|---|
| `abs` | Absolute value and complex magnitude |
| `alignsignals` | Align two signals by delaying earliest signal |
| `angle` | Phase angle |
| `bandpower*` | Band power |
| `barthannwin` | Modified Bartlett-Hann window |
| `bartlett` | Bartlett window |
| `besselap*` | Bessel analog lowpass filter prototype |
| `bilinear` | Bilinear transformation method for analog-to-digital filter conversion |
| `binmask2sigroi` | Convert binary mask to matrix of ROI limits |
| `bitrevorder` | Permute data into bit-reversed order |
| `blackman` | Blackman window |
| `blackmanharris` | Minimum four-term Blackman-Harris window |
| `bohmanwin` | Bohman window |
| `buttap` | Butterworth filter prototype |
| `butter` | Butterworth filter design |
| `buttord*` | Butterworth filter order and cutoff frequency |
| `cceps` | Complex cepstral analysis |
| `cconv` | Modulo-n circular convolution |

| | |
|---|---|
| `cfirpm*` | Complex and nonlinear-phase equiripple FIR filter design |
| `cheb1ap*` | Chebyshev Type I analog lowpass filter prototype |
| `cheb1ord*` | Chebyshev Type I filter order |
| `cheb2ap*` | Chebyshev Type II analog lowpass filter prototype |
| `cheb2ord*` | Chebyshev Type II filter order |
| `chebwin` | Chebyshev window |
| `cheby1*` | Chebyshev Type I filter design |
| `cheby2*` | Chebyshev Type II filter design |
| `chirp` | Swept-frequency cosine |
| `conv*` | Convolution and polynomial multiplication |
| `conv2` | 2-D convolution |
| `convmtx` | Convolution matrix |
| `corrcoef*` | Correlation coefficients |
| `corrmtx` | Data matrix for autocorrelation matrix estimation |
| `cov*` | Covariance |
| `cpsd` | Cross power spectral density |
| `cummax` | Cumulative maximum |
| `cummin` | Cumulative minimum |
| `cusum*` | Detect small changes in mean using cumulative sum |
| `czt*` | Chirp Z-transform |
| `db2pow` | Convert decibels to power |
| `dct*` | Discrete cosine transform |
| `deconv*` | Deconvolution and polynomial division |
| `detrend*` | Remove polynomial trend |
| `dftmtx` | Discrete Fourier transform matrix |
| `diric` | Dirichlet or periodic sinc function |
| `downsample` | Decrease sample rate by integer factor |
| `dpss*` | Discrete prolate spheroidal (Slepian) sequences |
| `dtw` | Distance between signals using dynamic time warping |
| `edr` | Edit distance on real signals |
| `ellip*` | Elliptic filter design |
| `ellipap*` | Elliptic analog lowpass filter prototype |
| `ellipord*` | Minimum order for elliptic filters |
| `emd*` | Empirical mode decomposition |
| `enbw` | Equivalent noise bandwidth |
| `envelope*` | Signal envelope |
| `envspectrum*` | Envelope spectrum for machinery diagnosis |

| | |
|---|---|
| eqtflength | Equalize lengths of transfer function's numerator and denominator |
| extendsigroi | Extend signal regions of interest to left and right |
| extractsigroi* | Extract signal regions of interest |
| falltime* | Fall time of negative-going bilevel waveform transitions |
| fft* | Fast Fourier transform |
| fft2* | 2-D fast Fourier transform |
| fftfilt* | FFT-based FIR filtering using overlap-add method |
| fftshift | Shift zero-frequency component to center of spectrum |
| filloutliers* | Detect and replace outliers in data |
| filter* | 1-D digital filter |
| filter2 | 2-D digital filter |
| filtfilt* | Zero-phase digital filtering |
| filtord | Filter order |
| finddelay | Estimate delay(s) between signals |
| findpeaks | Find local maxima |
| findsignal | Find signal location using similarity search |
| fir1 | Window-based FIR filter design |
| fir2* | Frequency sampling-based FIR filter design |
| fircls* | Constrained-least-squares FIR multiband filter design |
| fircls1* | Constrained-least-squares linear-phase FIR lowpass and highpass filter design |
| firls | Least-squares linear-phase FIR filter design |
| firpm* | Parks-McClellan optimal FIR filter design |
| firpmord* | Parks-McClellan optimal FIR filter order estimation |
| flattopwin | Flat top weighted window |
| freqspace | Frequency spacing for frequency response |
| freqz* | Frequency response of digital filter |
| fsst* | Fourier synchrosqueezed transform |
| fwht | Fast Walsh-Hadamard transform |
| gauspuls | Gaussian-modulated sinusoidal RF pulse |
| gausswin | Gaussian window |
| gmonopuls | Gaussian monopulse |
| goertzel* | Discrete Fourier transform with second-order Goertzel algorithm |
| hamming | Hamming window |
| hann | Hann (Hanning) window |
| hht* | Hilbert-Huang transform |
| hilbert | Discrete-time analytic signal using Hilbert transform |
| icceps | Inverse complex cepstrum |

| | |
|---|---|
| idct* | Inverse discrete cosine transform |
| ifft* | Inverse fast Fourier transform |
| ifft2* | 2-D inverse fast Fourier transform |
| ifftshift | Inverse zero-frequency shift |
| ifsst | Inverse Fourier synchrosqueezed transform |
| ifwht | Inverse Fast Walsh-Hadamard transform |
| instfreq* | Estimate instantaneous frequency |
| interp1* | 1-D data interpolation (table lookup) |
| intfilt* | Interpolation FIR filter design |
| iscola | Determine whether window-overlap combination is COLA compliant |
| isoutlier* | Find outliers in data |
| istft* | Inverse short-time Fourier transform |
| kaiser | Kaiser window |
| kaiserord | Kaiser window FIR filter design estimation parameters |
| kurtogram* | Visualize spectral kurtosis |
| levinson* | Levinson-Durbin recursion |
| lp2bp | Transform lowpass analog filters to bandpass |
| lp2bs | Transform lowpass analog filters to bandstop |
| lp2hp | Transform lowpass analog filters to highpass |
| lp2lp | Change cutoff frequency for lowpass analog filter |
| lsf2poly | Convert line spectral frequencies to prediction filter coefficients |
| max* | Maximum elements of an array |
| maxflat* | Generalized digital Butterworth filter design |
| mean* | Average or mean value of array |
| meanfreq | Mean frequency |
| medfreq | Median frequency |
| median* | Median value of array |
| mergesigroi | Merge signal regions of interest |
| min* | Minimum elements of an array |
| movmad* | Moving median absolute deviation |
| movmedian | Moving median |
| mscohere | Magnitude-squared coherence |
| nuttallwin | Nuttall-defined minimum 4-term Blackman-Harris window |
| obw | Occupied bandwidth |
| orderspectrum | Average spectrum versus order for vibration signal |
| ordertrack* | Track and extract order magnitudes from vibration signal |
| orderwaveform | Extract time-domain order waveforms from vibration signal |

| | |
|---|---|
| parzenwin | Parzen (de la Vallée Poussin) window |
| pchip* | Piecewise Cubic Hermite Interpolating Polynomial (PCHIP) |
| peak2peak | Maximum-to-minimum difference |
| peak2rms | Peak-magnitude-to-RMS ratio |
| pentropy* | Spectral entropy of signal |
| periodogram | Periodogram power spectral density estimate |
| pkurtosis* | Spectral kurtosis from signal or spectrogram |
| plomb* | Lomb-Scargle periodogram |
| poly2ac* | Convert prediction filter polynomial to autocorrelation sequence |
| poly2lsf* | Convert prediction filter coefficients to line spectral frequencies |
| poly2rc* | Convert prediction filter polynomial to reflection coefficients |
| pow2db | Convert power to decibels |
| powerbw | Power bandwidth |
| pspectrum* | Analyze signals in the frequency and time-frequency domains |
| pulstran* | Pulse train |
| pwelch | Welch's power spectral density estimate |
| rainflow* | Rainflow counts for fatigue analysis |
| randn* | Normally distributed random numbers |
| rc2ac* | Convert reflection coefficients to autocorrelation sequence |
| rc2poly* | Convert reflection coefficients to prediction filter polynomial |
| rceps | Real cepstrum and minimum-phase reconstruction |
| rcosdesign* | Raised cosine FIR pulse-shaping filter design |
| rectpuls | Sampled aperiodic rectangle |
| rectwin | Rectangular window |
| removesigroi | Remove signal regions of interest |
| resample* | Resample uniform or nonuniform data to new fixed rate |
| risetime* | Rise time of positive-going bilevel waveform transitions |
| rlevinson* | Reverse Levinson-Durbin recursion |
| rms | Root-mean-square level |
| rpmfreqmap | Frequency-RPM map for order analysis |
| rpmordermap | Order-RPM map for order analysis |
| rssq* | Root-sum-of-squares level |
| sawtooth | Sawtooth or triangle wave |
| sfdr* | Spurious free dynamic range |
| sgolay | Savitzky-Golay filter design |
| sgolayfilt | Savitzky-Golay filtering |
| shortensigroi | Shorten signal regions of interest from left and right |

| | |
|---|---|
| sigroi2binmask | Convert matrix of ROI limits to binary mask |
| sin | Sine of argument in radians |
| sinad* | Signal to noise and distortion ratio |
| sinc | Sinc function |
| smoothdata* | Smooth noisy data |
| snr* | Signal-to-noise ratio |
| sos2tf | Convert digital filter second-order section data to transfer function form |
| sosfilt | Second-order (biquadratic) IIR digital filtering |
| spectrogram* | Spectrogram using short-time Fourier transform |
| spline* | Cubic spline data interpolation |
| square | Square wave |
| statelevels | State-level estimation for bilevel waveform with histogram method |
| std* | Standard deviation |
| stft | Short-time Fourier transform |
| stftmag2sig | Signal reconstruction from STFT magnitude |
| tachorpm* | Extract RPM signal from tachometer pulses |
| taylorwin | Taylor window |
| tf2ss | Convert transfer function filter parameters to state-space form |
| tfestimate* | Transfer function estimate |
| tfridge* | Time-frequency ridges |
| thd* | Total harmonic distortion |
| toi* | Third-order intercept point |
| triang | Triangular window |
| tripuls | Sampled aperiodic triangle |
| tsa* | Time-synchronous signal average |
| tukeywin | Tukey (tapered cosine) window |
| unwrap* | Shift phase angles |
| upfirdn | Upsample, apply FIR filter, and downsample |
| upsample | Increase sample rate by integer factor |
| var* | Variance |
| vmd* | Variational mode decomposition |
| wvd* | Wigner-Ville distribution and smoothed pseudo Wigner-Ville distribution |
| xcorr* | Cross-correlation |
| xcorr2 | 2-D cross-correlation |
| xcov | Cross-covariance |
| xspectrogram* | Cross-spectrogram using short-time Fourier transforms |
| xwvd* | Cross Wigner-Ville distribution and cross smoothed pseudo Wigner-Ville distribution |

| | |
|---|---|
| `yulewalk*` | Recursive digital filter design |
| `zp2ss` | Convert zero-pole-gain filter parameters to state-space form |
| `zp2tf` | Convert zero-pole-gain filter parameters to transfer function form |

# Specifying Inputs in Code Generation from MATLAB

| In this section... |
| --- |
| “Defining Input Size and Type” on page 10-9 |
| “Inputs must be Constants” on page 10-10 |

## Defining Input Size and Type

When you use Signal Processing Toolbox functions for code generation, you must define the size and type of the function inputs. One way to do this is with the `-args` compilation option. The size and type of inputs must be defined because C is a statically typed language. To illustrate the need to define input size and type, consider the simplest call to `xcorr` requiring two input arguments. The following demonstrates the differences in the use of `xcorr` in MATLAB and in Code Generation from MATLAB.

Cross correlate two white noise vectors in MATLAB:

```
x = randn(512,1); %real valued white noise
y = randn(512,1); %real valued white noise
[C,lags] = xcorr(x,y);
x_circ = randn(256,1)+1j*randn(256,1); %circular white noise
y_circ = randn(256,1)+1j*randn(256,1); %circular white noise
[C1,lags1] = xcorr(x_circ,y_circ);
```

`xcorr` does not require the size and type of the input arguments. `xcorr` obtains this information at runtime. Contrast this behavior with a MEX-file created with `codegen`. Create the file `myxcorr.m` in a folder where you have read and write permission. Ensure that this folder is in the MATLAB search path. Copy and paste the following two lines of code into `myxcorr.m` and save the file. The compiler tag `%#codegen` must be included in the file.

```
function [C,Lags]=myxcorr(x,y)  %#codegen
[C,Lags]=xcorr(x,y);
```

Enter the following command at the MATLAB command prompt:

```
codegen myxcorr -args {zeros(512,1),zeros(512,1)} -o myxcorr
```

Run the MEX-file:

```
x = randn(512,1); %real valued white noise
y = randn(512,1); %real valued white noise
[C,Lags] = myxcorr(x,y);
```

Define two new inputs `x1` and `y1` by transposing `x` and `y`.

```
x1 = x'; %x1 is 1x512
y1 = y'; %y1 is 1x512
```

Attempt to rerun the MEX-file with the transposed inputs.

```
[C,Lags] = myxcorr(x1,y1); %Errors
```

The preceding program errors with the message `??? MATLAB expression 'x' is not of the correct size: expected [512x1] found [1x512]`.

The error results because the inputs are specified to be `512x1` real-valued column vectors at compilation. For complex-valued inputs, you must specify that the input is complex valued. For example:

```
codegen myxcorr -o ComplexXcorr ...
-args {complex(zeros(512,1)),complex(zeros(512,1))}
```

Run the MEX-file at the MATLAB command prompt with complex-valued inputs of the correct size:

```
x_circ = randn(512,1)+1j*randn(512,1); %circular white noise
y_circ = randn(512,1)+1j*randn(512,1); %circular white noise
[C,Lags] = ComplexXcorr(x_circ,y_circ);
```

Attempting to run `ComplexXcorr` with real valued inputs results in the error: `??? MATLAB expression 'x' is not of the correct complexness`.

## Inputs must be Constants

For a number of supported Signal Processing Toolbox functions, the inputs or a subset of the inputs must be specified as constants at compilation time. Use `coder.Type` with the `-args` compilation option, or enter the constants directly in the source code.

Specifying inputs as constants at compilation time results in significant advantages in the speed and efficiency of the generated code. For example, storing filter coefficients or window function values as vectors in the C source code improves performance by avoiding costly computation at runtime. Because a primary purpose of Code Generation from MATLAB is to generate optimized C code for desktop and embedded systems, emphasis is placed on providing the user with computational savings at runtime whenever possible.

To illustrate the constant input requirement with `ellip`, create the file `myLowpassFilter.m` in a folder where you have read and write permission. Ensure that this folder is in the MATLAB search path. Copy and paste the following lines of code into `myLowpassFilter.m` and save the file.

```
function output = myLowpassFilter(input,N,Wn) %#codegen
[B,A] = ellip(N,Wn,'low');
output = filter(B,A,input);
```

If you have the MATLAB Coder software, enter the following command at the MATLAB command prompt:

```
codegen myLowpassFilter -o myLowpassFilter ...
-args {zeros(512,1),coder.newtype('constant',5),coder.newtype('constant',0.1)} -report
```

Once the program compiles successfully, the following message appears in the command window: `Code generation successful: View report`.

Click on `View report`. Click on the `C code` tab on the top left and open the target source file `myLowpassFilter.c`. The source code includes the numerator and denominator filter coefficients.

Run the MEX-file without entering the constants:

```
output = myLowpassFilter(randn(512,1));
```

If you attempt to run the MEX-file by inputting the constants, you receive the error `??? Error using ==> myLowpassFilter 1 input required for entry-point 'myLowpassFilter'`.

You may also enter the constants in the MATLAB source code directly. Edit the `myLowPassFilter.m` file and replace the MATLAB code with the lines:

```
function output = myLowpassFilter(input) %#codegen
[B,A] = ellip(5,0.1,'low');
output = filter(B,A,input);
```

Enter the following command at the MATLAB command prompt:

```
codegen myLowpassFilter -args {zeros(512,1)} -o myLowpassFilter
```

Run the MEX-file by entering the following at the MATLAB command prompt:

```
output = myLowpassFilter(randn(512,1));
```

# Apply Lowpass Filter to Input Signal

Assuming a sample rate of 20 kHz, create a fourth-order Butterworth filter with a 3-dB frequency of 2.5 kHz. Filter coefficients for `butter` must be constants for code generation.

```
type ButterFilt
```

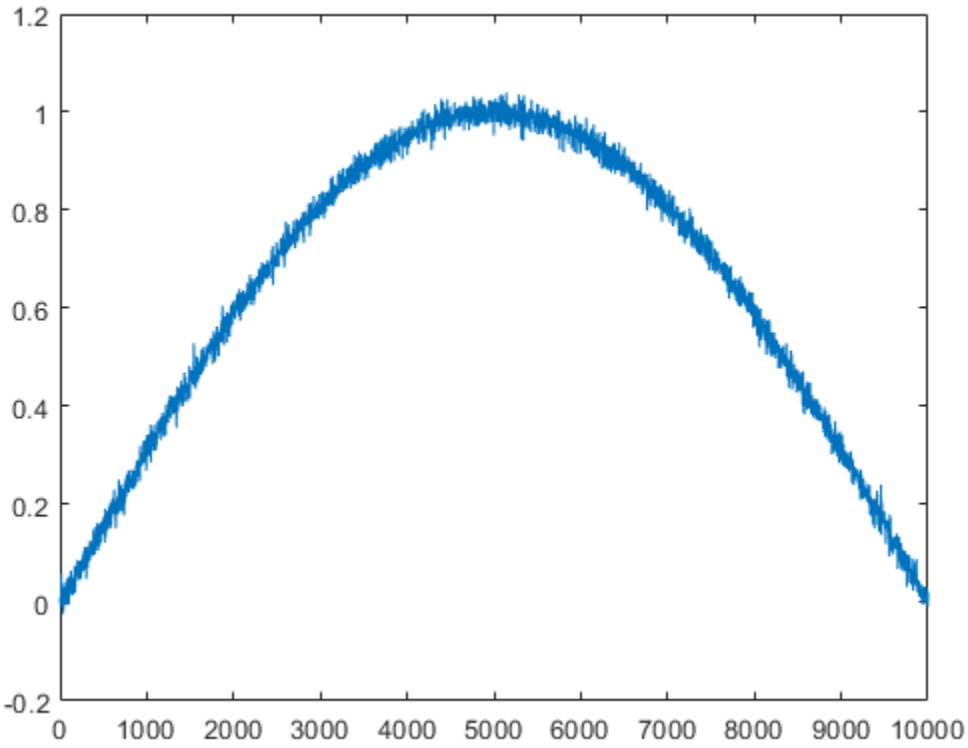
```
function output_data=ButterFilt(input_data) %#codegen
[b,a]=butter(4,0.25);
output_data=filter(b,a,input_data);
end
```

Use the Butterworth filter to lowpass-filter a noisy sine wave. Plot the result.

```
t = transpose(linspace(0,pi,10000));
x = sin(t) + 0.03*randn(numel(t),1);
```

Filter the noisy sine wave using the Butterworth filter.

```
fx = ButterFilt(x);
plot(fx)
```



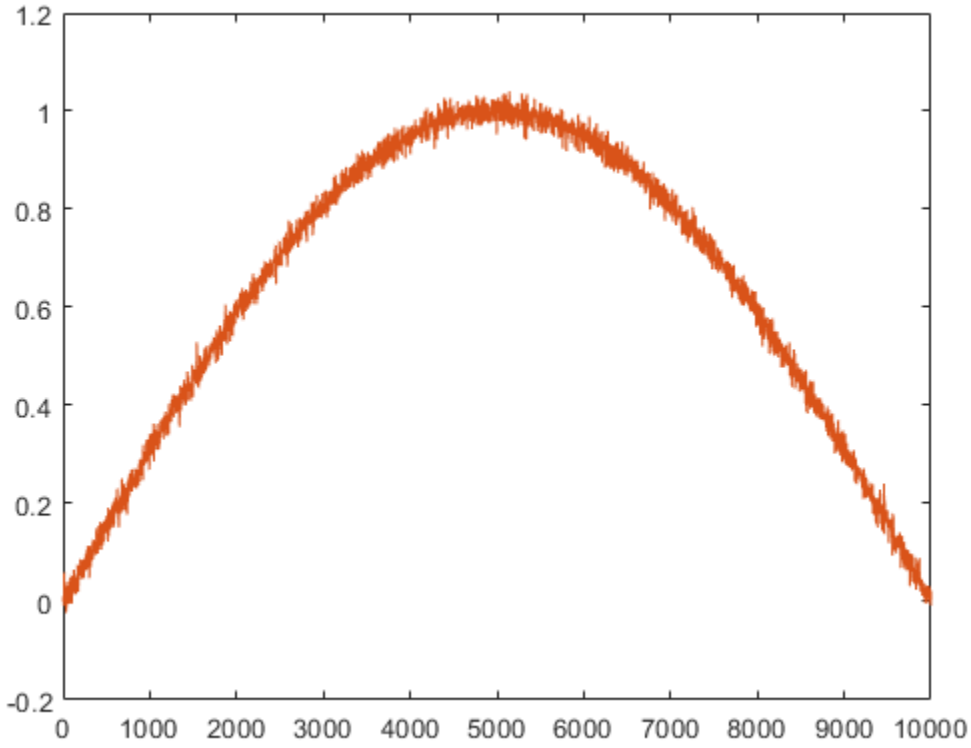

Run the `codegen` command to obtain the C source code `ButterFilt.c` and MEX file:

```
codegen ButterFilt -args {zeros(10000,1)} -o ButterFilt_mex -report
```

```
Code generation successful: To view the report, open('codegen\mex\ButterFilt\html\report.mldatx'
```

The C source code includes the five numerator and denominator coefficients of the fourth-order Butterworth filter as static constants. Apply the filter using the MEX-file:

```
output_data = ButterFilt_mex(x);
hold on
plot(output_data)
hold off
```



# Zero-Phase Filtering

Design a lowpass Butterworth filter with a 1 kHz 3-dB frequency to implement zero-phase filtering on data sampled at a rate of 20 kHz.

```
type myZerophaseFilt.m
```

```
function output  = myZerophaseFilt(input) %#codegen

[B,A] = butter(20,0.314);
output = filtfilt(B,A,input);

end
```

Use codegen to create the MEX file for `myZerophaseFilt.m`.

```
codegen myZerophaseFilt -args {zeros(1,20001)} -o myZerophaseFilt_mex -report
```

```
Code generation successful: To view the report, open('codegen\mex\myZerophaseFilt\html\report.ml
```

Generate a noisy sinusoid signal as input to the filter.

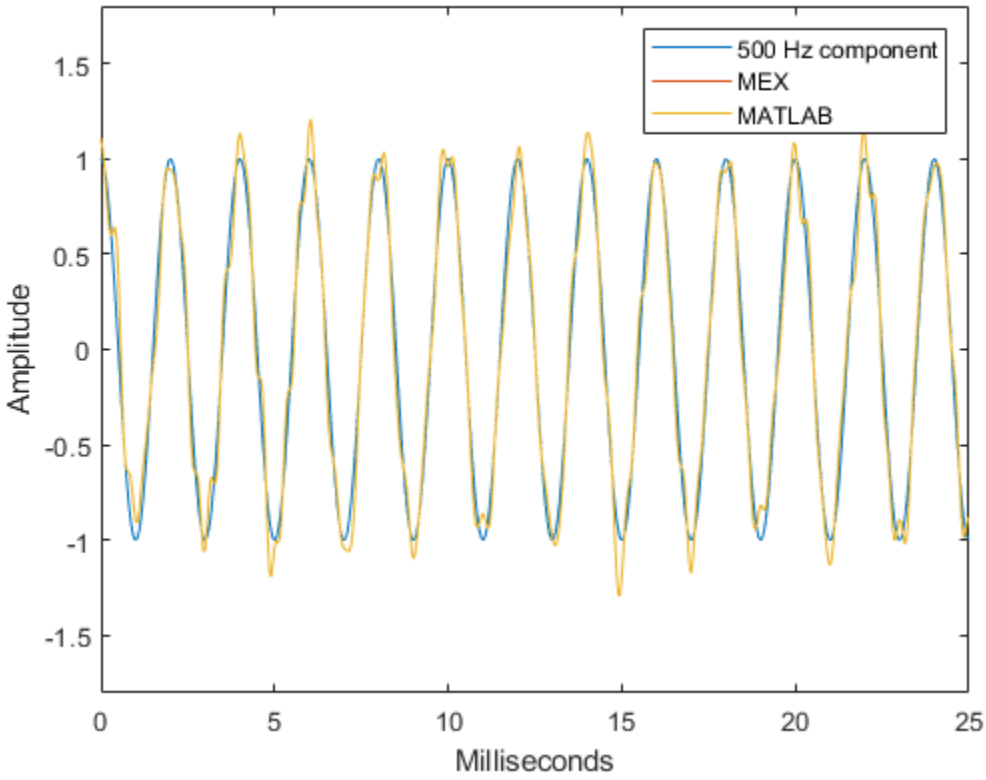
```
Fs = 20000;
t = 0:1/Fs:1;
comp500Hz = cos(2*pi*500*t);
signal = comp500Hz + sin(2*pi*4000*t) + 0.2*randn(size(t));
```

Filter input data using both MATLAB® and MEX functions.

```
FilteredData = myZerophaseFilt(signal);
MexFilteredData = myZerophaseFilt_mex(signal);
```

Plot the 500 Hz component and the filtered data.

```
tms = t*1000;
plot(tms,comp500Hz)
hold on
plot(tms,MexFilteredData)
plot(tms,FilteredData)
hold off

xlabel('Milliseconds')
ylabel('Amplitude')
axis([0 25 -1.8 1.8])
legend('500 Hz component','MEX','MATLAB')
```


500 Hz component
MEX
MATLAB
Amplitude
Milliseconds
1.5
1
0.5
0
-0.5
-1
-1.5
0
5
10
15
20
25

# Compute Modified Periodogram Using Generated C Code

Create a function `periodogram_data.m` that returns the modified periodogram power spectral density (PSD) estimate of an input signal using a window. The function specifies a number of discrete Fourier transform points equal to the length of the input signal.

```
type periodogram_data
```

```
function [pxx,f] = periodogram_data(inputData,window)
%#codegen
nfft = length(inputData);
[pxx,f] = periodogram(inputData,window,nfft);
end
```

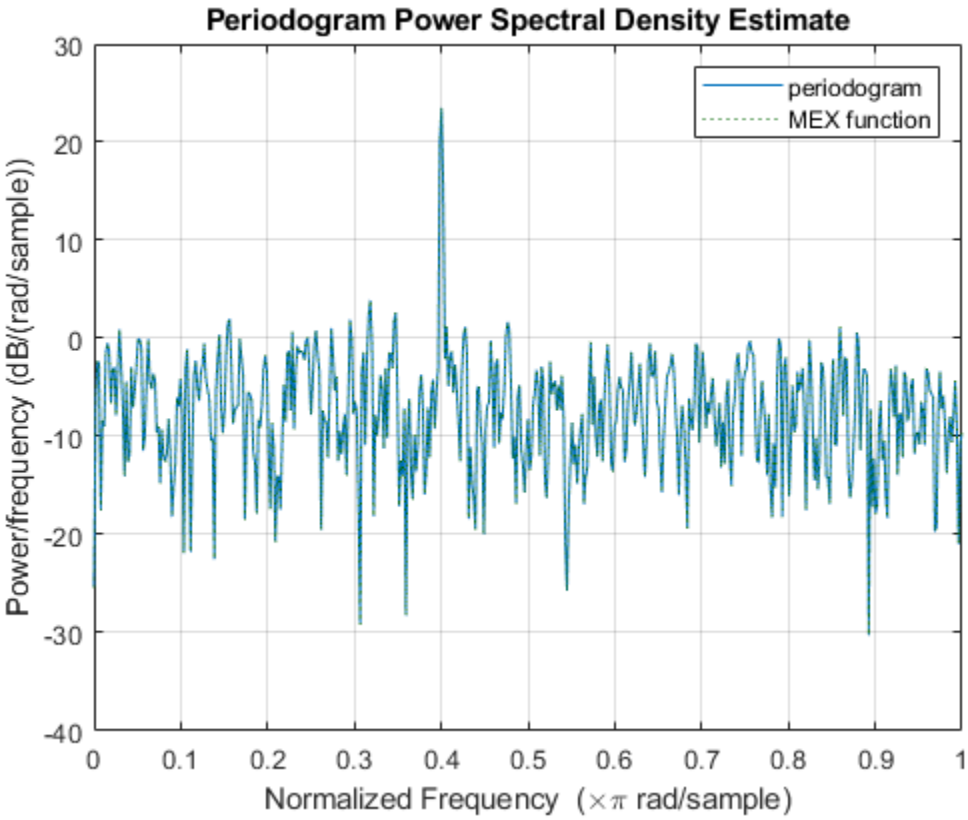
Use `codegen` (MATLAB Coder) to generate a MEX function.

- The `%#codegen` directive in the function indicates that the MATLAB® code is intended for code generation.
- The `-args` option specifies example arguments that define the size, class, and complexity of the inputs to the MEX-file. For this example, specify `inputData` as a 1024-by-1 double precision random vector and `window` as a Hamming window of length 1024. In subsequent calls to the MEX function, use 1024-sample input signals and windows.
- If you want the MEX function to have a different name, use the `-o` option.
- If you want to view a code generation report, add the `-report` option at the end of the `codegen` command.

```
codegen periodogram_data -args {randn(1024,1),hamming(1024)}
```

Compute the PSD estimate of a 1024-sample noisy sinusoid using the periodogram function and the MEX function you generated. Specify a sinusoid normalized frequency of $2\pi/5$ rad/sample and a Hann window. Plot the two estimates to verify they coincide.

```
N = 1024;
x = 2*cos(2*pi/5*(0:N-1)') + randn(N,1);
periodogram(x,hann(N))
[pxMex,fMex] = periodogram_data(x,hann(N));
hold on
plot(fMex/pi,pow2db(pxMex),':','Color',[0 0.4 0])
hold off
grid on
legend('periodogram','MEX function')
```


Periodogram Power Spectral Density Estimate
periodogram
MEX function
Power/frequency (dB/(rad/sample))
Normalized Frequency (×π rad/sample)
30
20
10
0
-10
-20
-30
-40
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

# Convolution and Correlation


- “Linear and Circular Convolution” on page 11-2
- “Confidence Intervals for Sample Autocorrelation” on page 11-4
- “Residual Analysis with Autocorrelation” on page 11-6
- “Autocorrelation of Moving Average Process” on page 11-12
- “Cross-Correlation of Two Moving Average Processes” on page 11-15
- “Cross-Correlation of Delayed Signal in Noise” on page 11-17
- “Cross-Correlation of Phase-Lagged Sine Wave” on page 11-19

# Linear and Circular Convolution

This example shows how to establish an equivalence between linear and circular convolution.

Linear and circular convolution are fundamentally different operations. However, there are conditions under which linear and circular convolution are equivalent. Establishing this equivalence has important implications. For two vectors, `x` and `y`, the circular convolution is equal to the inverse discrete Fourier transform (DFT) of the product of the vectors' DFTs. Knowing the conditions under which linear and circular convolution are equivalent allows you to use the DFT to efficiently compute linear convolutions.

The linear convolution of an $N$-point vector, `x`, and an $L$-point vector, `y`, has length $N + L - 1$.

For the circular convolution of $x$ and $y$ to be equivalent, you must pad the vectors with zeros to length at least $N + L - 1$ before you take the DFT. After you invert the product of the DFTs, retain only the first $N + L - 1$ elements.

Create two vectors, `x` and `y`, and compute the linear convolution of the two vectors.

```
x = [2 1 2 1];
y = [1 2 3];
clin = conv(x,y);
```

The output has length 4+3-1.

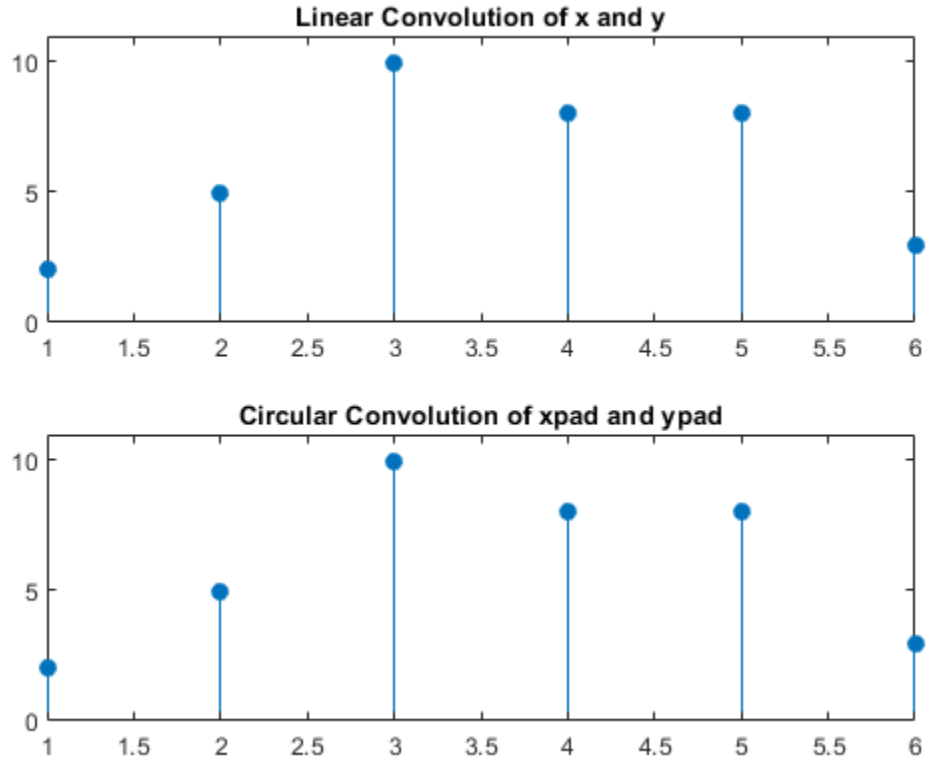
Pad both vectors with zeros to length 4+3-1. Obtain the DFT of both vectors, multiply the DFTs, and obtain the inverse DFT of the product.

```
xpad = [x zeros(1,6-length(x))];
ypad = [y zeros(1,6-length(y))];
ccirc = ifft(fft(xpad).*fft(ypad));
```

The circular convolution of the zero-padded vectors, `xpad` and `ypad`, is equivalent to the linear convolution of `x` and `y`. You retain all the elements of `ccirc` because the output has length 4+3-1.

Plot the output of linear convolution and the inverse of the DFT product to show the equivalence.

```
subplot(2,1,1)
stem(clin,'filled')
ylim([0 11])
title('Linear Convolution of x and y')

subplot(2,1,2)
stem(ccirc,'filled')
ylim([0 11])
title('Circular Convolution of xpad and ypad')
```




Pad the vectors to length 12 and obtain the circular convolution using the inverse DFT of the product of the DFTs. Retain only the first 4+3-1 elements to produce an equivalent result to linear convolution.

```
N = length(x)+length(y)-1;
xpad = [x zeros(1,12-length(x))];
ypad = [y zeros(1,12-length(y))];
ccirc = ifft(fft(xpad).*fft(ypad));
ccirc = ccirc(1:N);
```

The Signal Processing Toolbox™ software has a function, `cconv`, that returns the circular convolution of two vectors. You can obtain the linear convolution of `x` and `y` using circular convolution with the following code.

```
ccirc2 = cconv(x,y,6);
```

`cconv` internally uses the same DFT-based procedure illustrated in the previous example.

# Confidence Intervals for Sample Autocorrelation

This example shows how to create confidence intervals for the autocorrelation sequence of a white noise process. Create a realization of a white noise process with length $L = 1000$ samples. Compute the sample autocorrelation to lag 20. Plot the sample autocorrelation along with the approximate 95%-confidence intervals for a white noise process.

Create the white noise random vector. Set the random number generator to the default settings for reproducible results. Obtain the normalized sampled autocorrelation to lag 20.

```
rng default
L = 1000;
x = randn(L,1);
[xc,lags] = xcorr(x,20,'coeff');
```

Create the lower and upper 95% confidence bounds for the normal distribution $N(0, 1/L)$, whose standard deviation is $1/\sqrt{L}$. For a 95%-confidence interval, the critical value is $\sqrt{2}\,\mathrm{erf}^{-1}(0.95) \approx 1.96$ and the confidence interval is

$$\Delta = 0 \pm \frac{1.96}{\sqrt{L}}.$$

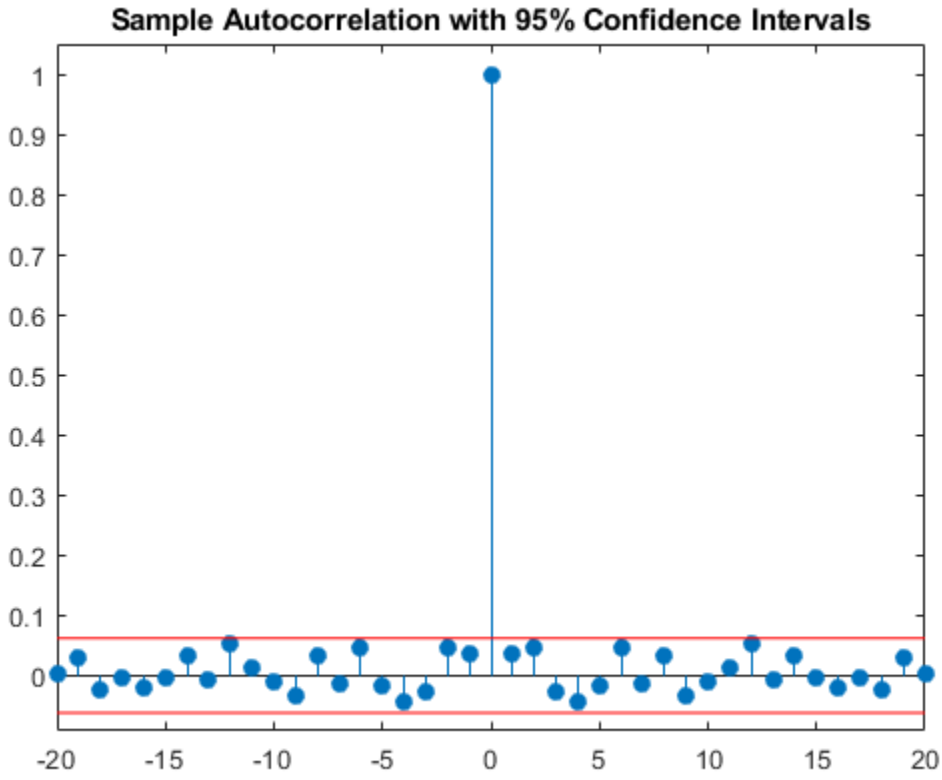
```
vcrit = sqrt(2)*erfinv(0.95)
```

```
vcrit = 1.9600
```

```
lconf = -vcrit/sqrt(L);
upconf = vcrit/sqrt(L);
```

Plot the sample autocorrelation along with the 95%-confidence interval.

```
stem(lags,xc,'filled')
hold on
plot(lags,[lconf;upconf]*ones(size(lags)),'r')
hold off
ylim([lconf-0.03 1.05])
title('Sample Autocorrelation with 95% Confidence Intervals')
```



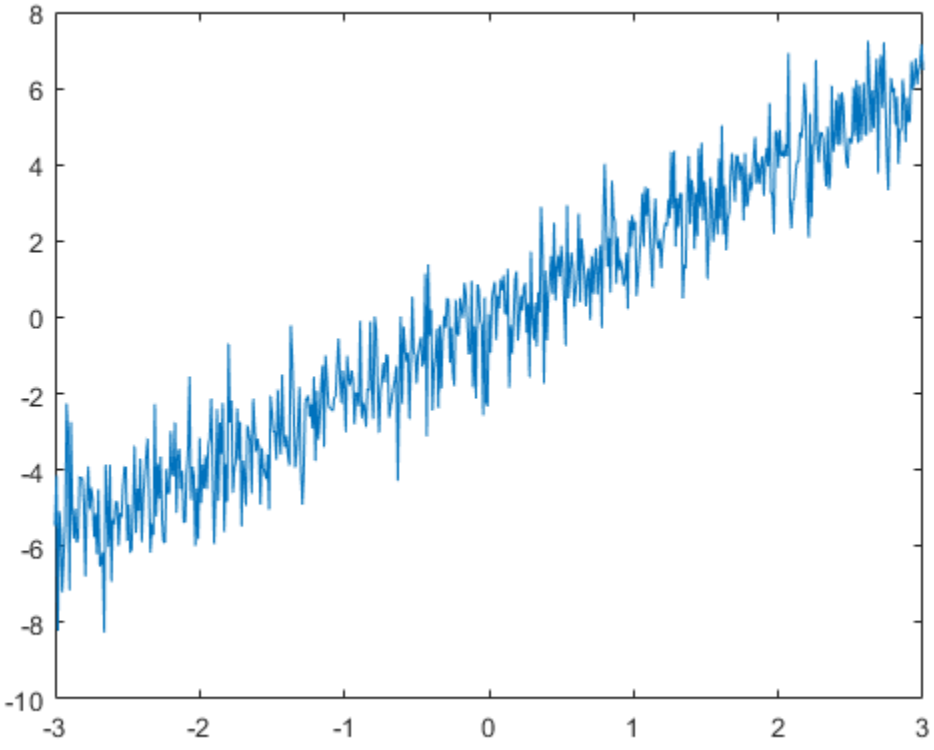

You see in the above figure that the only autocorrelation value outside of the 95%-confidence interval occurs at lag 0 as expected for a white noise process. Based on this result, you can conclude that the data are a realization of a white noise process.

# Residual Analysis with Autocorrelation

This example shows how to use autocorrelation with a confidence interval to analyze the residuals of a least-squares fit to noisy data. The residuals are the differences between the fitted model and the data. In a signal-plus-white noise model, if you have a good fit for the signal, the residuals should be white noise.

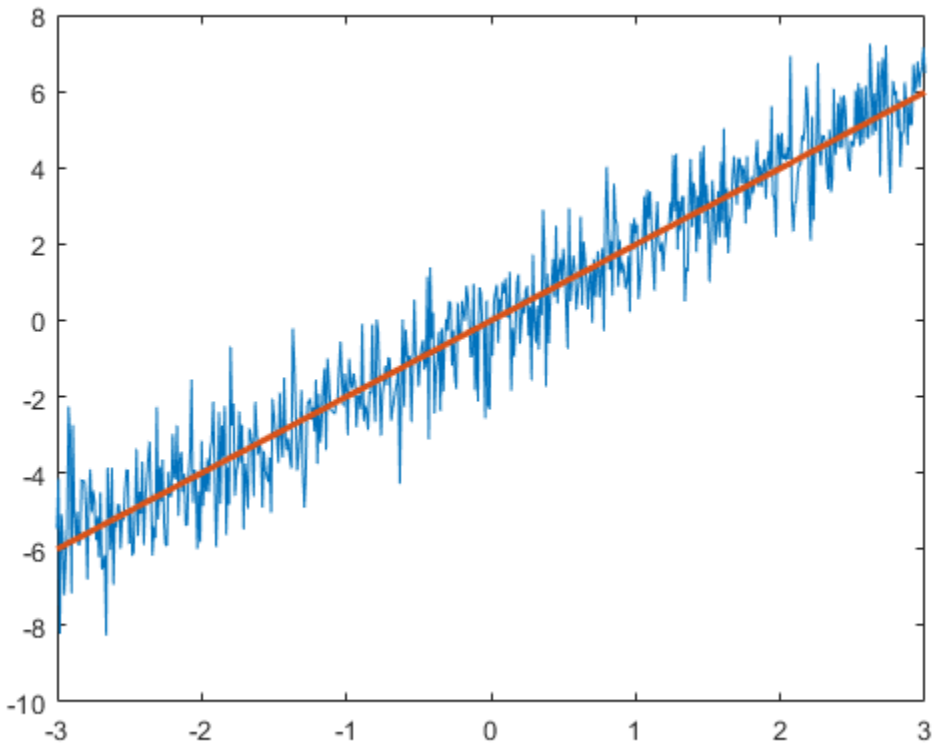
Create a noisy data set consisting of a 1st-order polynomial (straight line) in additive white Gaussian noise. The additive noise is a sequence of uncorrelated random variables following a $N(0,1)$ distribution. This means that all the random variables have mean zero and unit variance. Set the random number generator to the default settings for reproducible results.

```
x = -3:0.01:3;
rng default
y = 2*x+randn(size(x));
plot(x,y)
```




Use `polyfit` to find the least-squares line for the noisy data. Plot the original data along with the least-squares fit.

```
coeffs = polyfit(x,y,1);
yfit = coeffs(2)+coeffs(1)*x;

plot(x,y)
hold on
plot(x,yfit,'linewidth',2)
```




Find the residuals. Obtain the autocorrelation sequence of the residuals to lag 50.

```
residuals = y - yfit;
[xc,lags] = xcorr(residuals,50,'coeff');
```

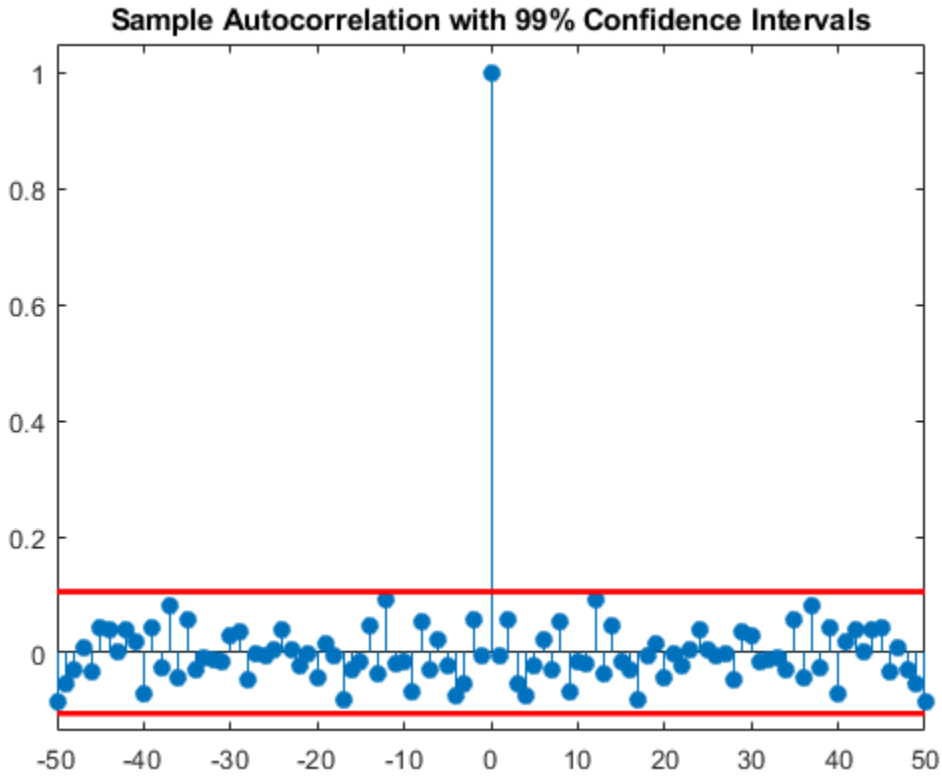
When you inspect the autocorrelation sequence, you want to determine whether or not there is evidence of autocorrelation. In other words, you want to determine whether the sample autocorrelation sequence looks like the autocorrelation sequence of white noise. If the autocorrelation sequence of the residuals looks like the autocorrelation of a white noise process, you are confident that none of the signal has escaped your fit and ended up in the residuals. In this example, use a 99%-confidence interval. To construct the confidence interval, you need to know the distribution of the sample autocorrelation values. You also need to find the critical values on the appropriate distribution between which lie 0.99 of the probability. Because the distribution in this case is Gaussian, you can use complementary inverse error function, `erfcinv`. The relationship between this function and the inverse of the Gaussian cumulative distribution function is described on the reference page for `erfcinv`.

Find the critical value for the 99%-confidence interval. Use the critical value to construct the lower and upper confidence bounds.

```
conf99 = sqrt(2)*erfcinv(2*.01/2);
lconf = -conf99/sqrt(length(x));
upconf = conf99/sqrt(length(x));
```

Plot the autocorrelation sequence along with the 99%-confidence intervals.

```
figure

stem(lags,xc,'filled')
ylim([lconf-0.03 1.05])
hold on
plot(lags,lconf*ones(size(lags)),'r','linewidth',2)
plot(lags,upconf*ones(size(lags)),'r','linewidth',2)
title('Sample Autocorrelation with 99% Confidence Intervals')
```



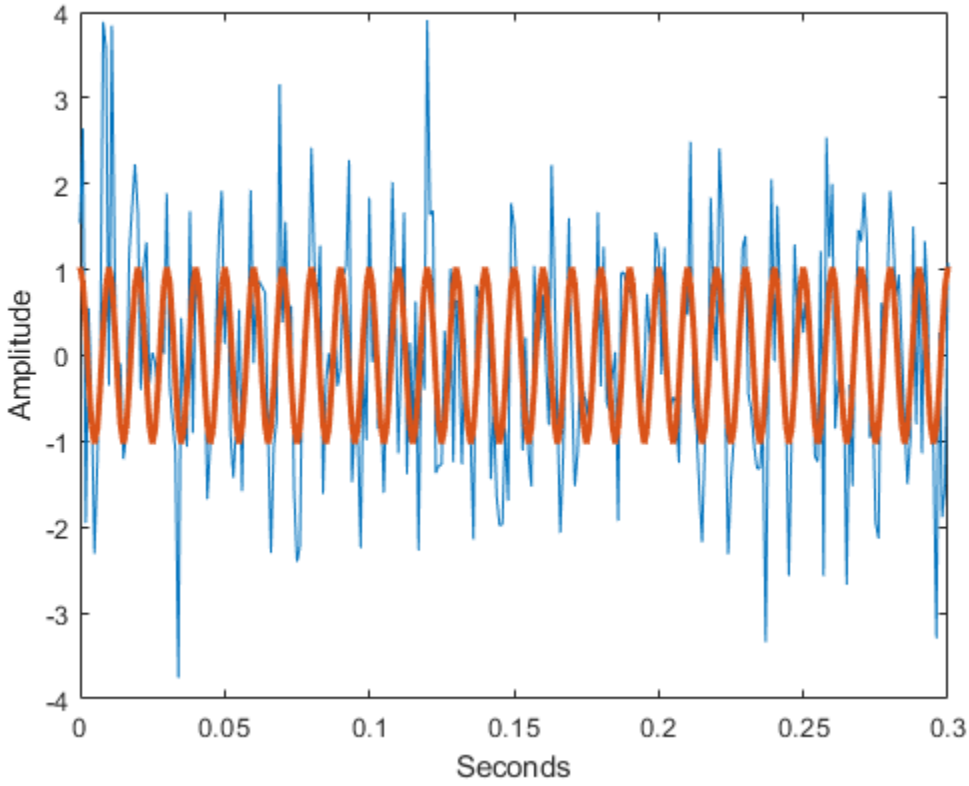

Except at zero lag, the sample autocorrelation values lie within the 99%-confidence bounds for the autocorrelation of a white noise sequence. From this, you can conclude that the residuals are white noise. More specifically, you cannot reject that the residuals are a realization of a white noise process.

Create a signal consisting of a sine wave plus noise. The data are sampled at 1 kHz. The frequency of the sine wave is 100 Hz. Set the random number generator to the default settings for reproducible results.

```
Fs = 1000;
t = 0:1/Fs:1-1/Fs;
rng default
x = cos(2*pi*100*t)+randn(size(t));
```

Use the discrete Fourier transform (DFT) to obtain the least-squares fit to the sine wave at 100 Hz. The least-squares estimate of the amplitude is 2 / $N$ times the DFT coefficient corresponding to 100 Hz, where $N$ is the length of the signal. The real part is the amplitude of a cosine at 100 Hz and the imaginary part is the amplitude of a sine at 100 Hz. The least-squares fit is the sum of the cosine and sine with the correct amplitude. In this example, DFT bin 101 corresponds to 100 Hz.

```
xdft = fft(x);
ampest = 2/length(x)*xdft(101);
xfit = real(ampest)*cos(2*pi*100*t)+imag(ampest)*sin(2*pi*100*t);

figure

plot(t,x)
hold on
plot(t,xfit,'linewidth',2)
axis([0 0.30 -4 4])
xlabel('Seconds')
ylabel('Amplitude')
```

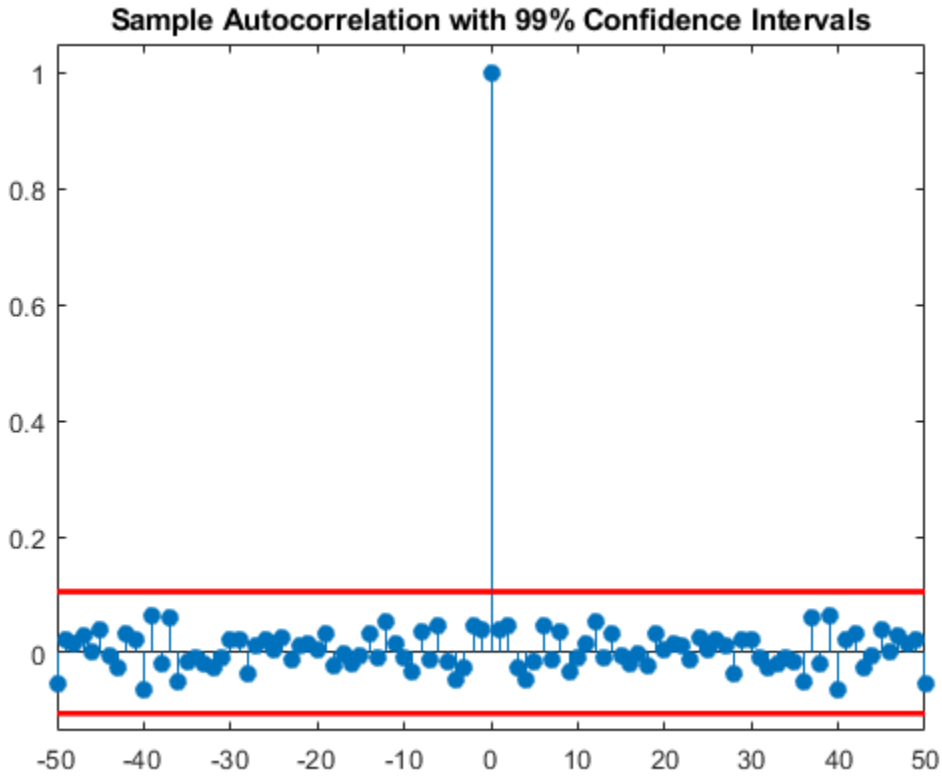



Find the residuals and determine the sample autocorrelation sequence to lag 50.

```
residuals = x-xfit;
[xc,lags] = xcorr(residuals,50,'coeff');
```

Plot the autocorrelation sequence with the 99%-confidence intervals.

```
figure

stem(lags,xc,'filled')
ylim([lconf-0.03 1.05])
hold on
plot(lags,lconf*ones(size(lags)),'r','linewidth',2)
plot(lags,upconf*ones(size(lags)),'r','linewidth',2)
title('Sample Autocorrelation with 99% Confidence Intervals')
```



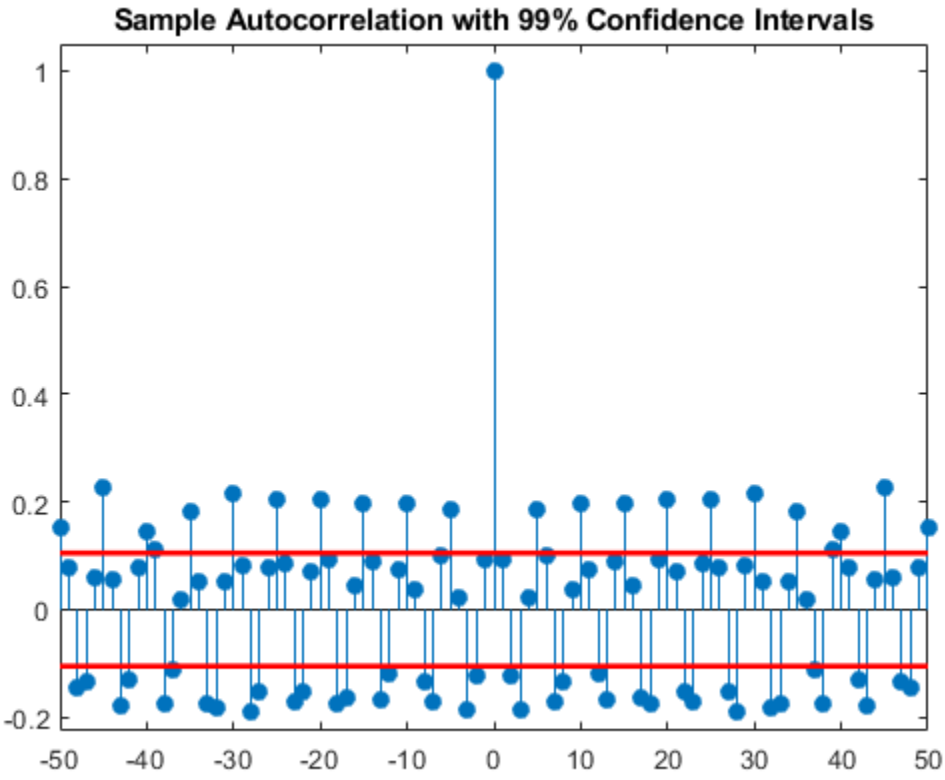

Again, you see that except at zero lag, the sample autocorrelation values lie within the 99%-confidence bounds for the autocorrelation of a white noise sequence. From this, you can conclude that the residuals are white noise. More specifically, you cannot reject that the residuals are a realization of a white noise process.

Finally, add another sine wave with a frequency of 200 Hz and an amplitude of 3/4. Fit only the sine wave at 100 Hz and find the sample autocorrelation of the residuals.

```
x = x+3/4*sin(2*pi*200*t);
xdft = fft(x);
ampest = 2/length(x)*xdft(101);
xfit = real(ampest)*cos(2*pi*100*t)+imag(ampest)*sin(2*pi*100*t);
residuals = x-xfit;
[xc,lags] = xcorr(residuals,50,'coeff');
```

Plot the sample autocorrelation along with the 99%-confidence intervals.

```
figure

stem(lags,xc,'filled')
ylim([lconf-0.12 1.05])
hold on
plot(lags,lconf*ones(size(lags)),'r','linewidth',2)
plot(lags,upconf*ones(size(lags)),'r','linewidth',2)
title('Sample Autocorrelation with 99% Confidence Intervals')
```




In this case, the autocorrelation values clearly exceed the 99%-confidence bounds for a white noise autocorrelation at many lags. Here you can reject the hypothesis that the residuals are a white noise sequence. The implication is that the model has not accounted for all the signal and therefore the residuals consist of signal plus noise.

# Autocorrelation of Moving Average Process

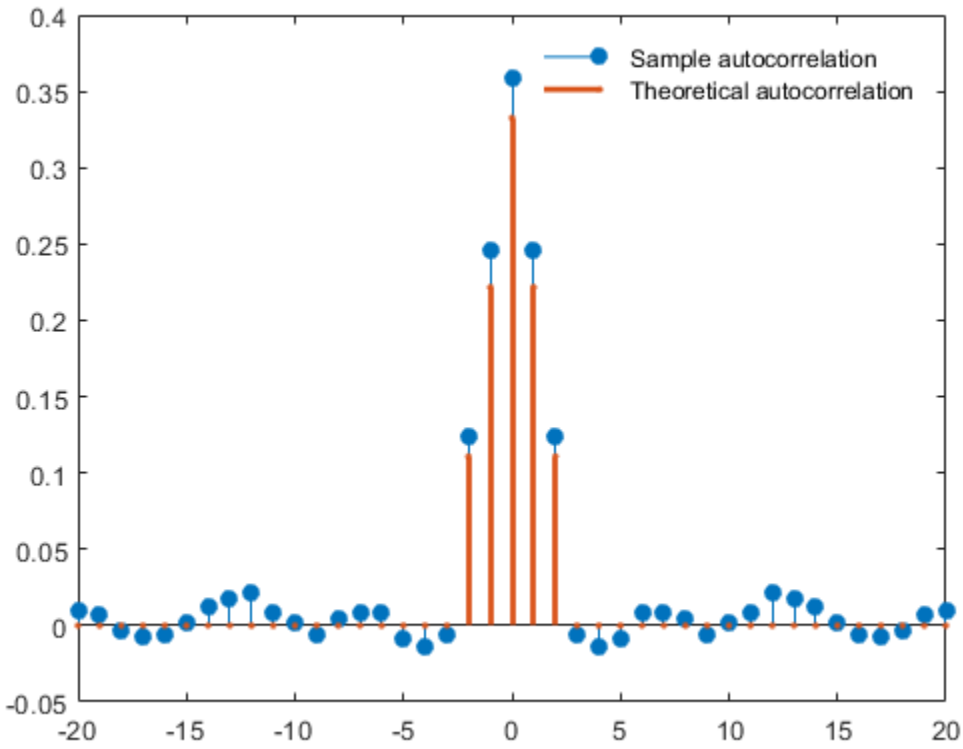
This example shows how to introduce autocorrelation into a white noise process by filtering. When we introduce autocorrelation into a random signal, we manipulate its frequency content. A moving average filter attenuates the high-frequency components of the signal, effectively smoothing it.

Create the impulse response for a 3-point moving average filter. Filter an $N(0,1)$ white noise sequence with the filter. Set the random number generator to the default settings for reproducible results.

```
h = 1/3*ones(3,1);
rng default
x = randn(1000,1);
y = filter(h,1,x);
```

Obtain the biased sample autocorrelation out to 20 lags. Plot the sample autocorrelation along with the theoretical autocorrelation.

```
[xc,lags] = xcorr(y,20,'biased');

Xc = zeros(size(xc));
Xc(19:23) = [1 2 3 2 1]/9*var(x);

stem(lags,xc,'filled')
hold on
stem(lags,Xc,'.','linewidth',2)

lg = legend('Sample autocorrelation','Theoretical autocorrelation');
lg.Location = 'NorthEast';
lg.Box = 'off';
```

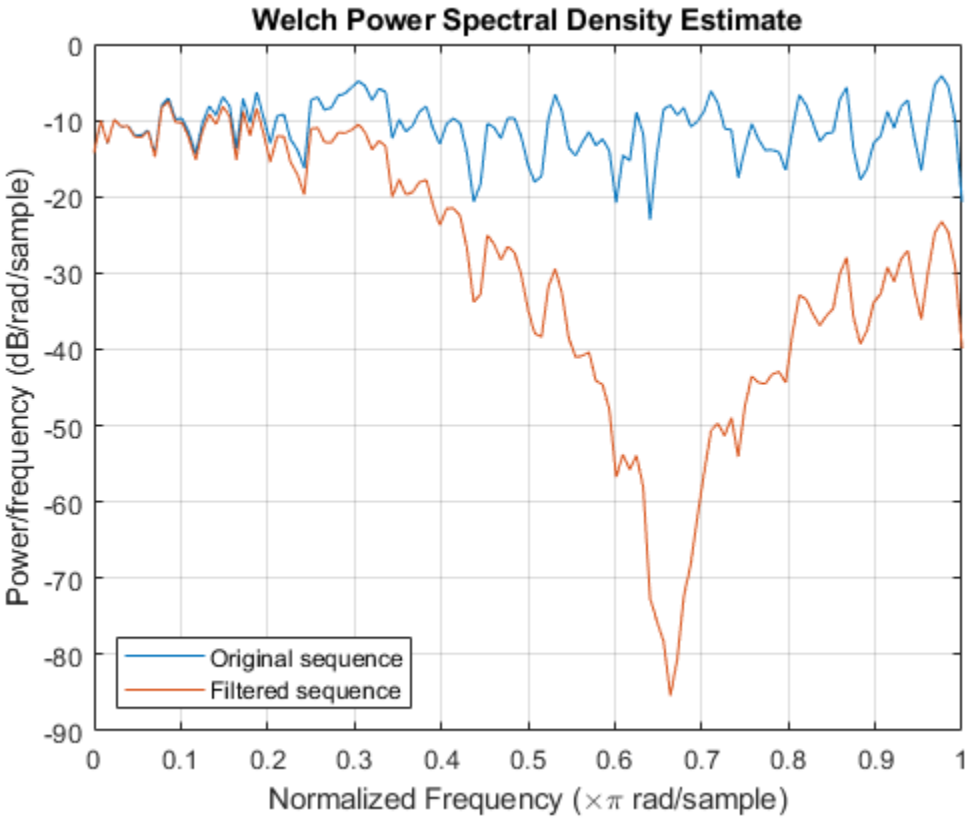



The sample autocorrelation captures the general form of the theoretical autocorrelation, even though the two sequences do not agree in detail.

In this case, it is clear that the filter has introduced significant autocorrelation only over lags [-2,2]. The absolute value of the sequence decays quickly to zero outside of that range.

To see that the frequency content has been affected, plot Welch estimates of the power spectral densities of the original and filtered signals.

```
[pxx,wx] = pwelch(x);
[pyy,wy] = pwelch(y);

figure
plot(wx/pi,20*log10(pxx),wy/pi,20*log10(pyy))

lg = legend('Original sequence','Filtered sequence');
lg.Location = 'SouthWest';

xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Power/frequency (dB/rad/sample)')
title('Welch Power Spectral Density Estimate')
grid
```




The white noise has been "colored" by the moving average filter.

## See Also

## External Websites

- Ellis, Dan. *About Colored Noise*. https://www.ee.columbia.edu/~dpwe/noise/

# Cross-Correlation of Two Moving Average Processes

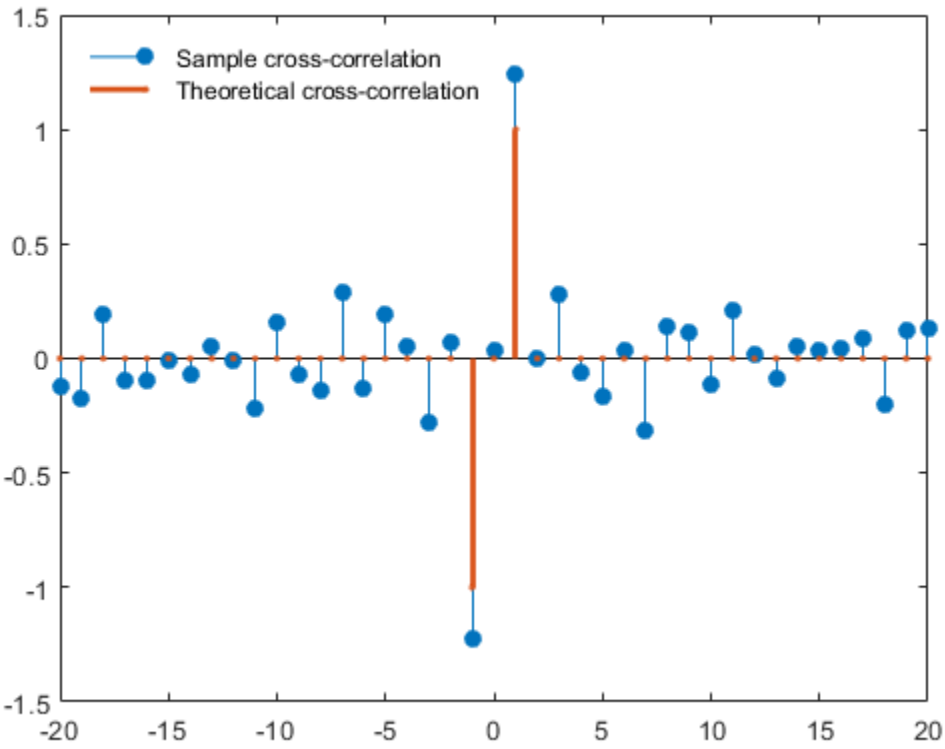
This example shows how to find and plot the cross-correlation sequence between two moving average processes. The example compares the sample cross-correlation with the theoretical cross-correlation. Filter an $N(0,1)$ white noise input with two different moving average filters. Plot the sample and theoretical cross-correlation sequences.

Create an $N(0,1)$ white noise sequence. Set the random number generator to the default settings for reproducible results. Create two moving average filters. One filter has impulse response $\delta(n)+\delta(n-1)$. The other filter has impulse response $\delta(n)-\delta(n-1)$.

```
rng default

w = randn(100,1);
x = filter([1 1],1,w);
y = filter([1 -1],1,w);
```

Obtain the sample cross-correlation sequence up to lag 20. Plot the sample cross-correlation along with the theoretical cross-correlation.

```
[xc,lags] = xcorr(x,y,20,'biased');

Xc = zeros(size(xc));
Xc(20) = -1;
Xc(22) = 1;

stem(lags,xc,'filled')
hold on
stem(lags,Xc,'.','linewidth',2)

q = legend('Sample cross-correlation','Theoretical cross-correlation');
q.Location = 'NorthWest';
q.FontSize = 9;
q.Box = 'off';
```




The theoretical cross-correlation is −1 at lag −1, 1 at lag 1, and zero at all other lags. The sample cross-correlation sequence approximates the theoretical cross-correlation.

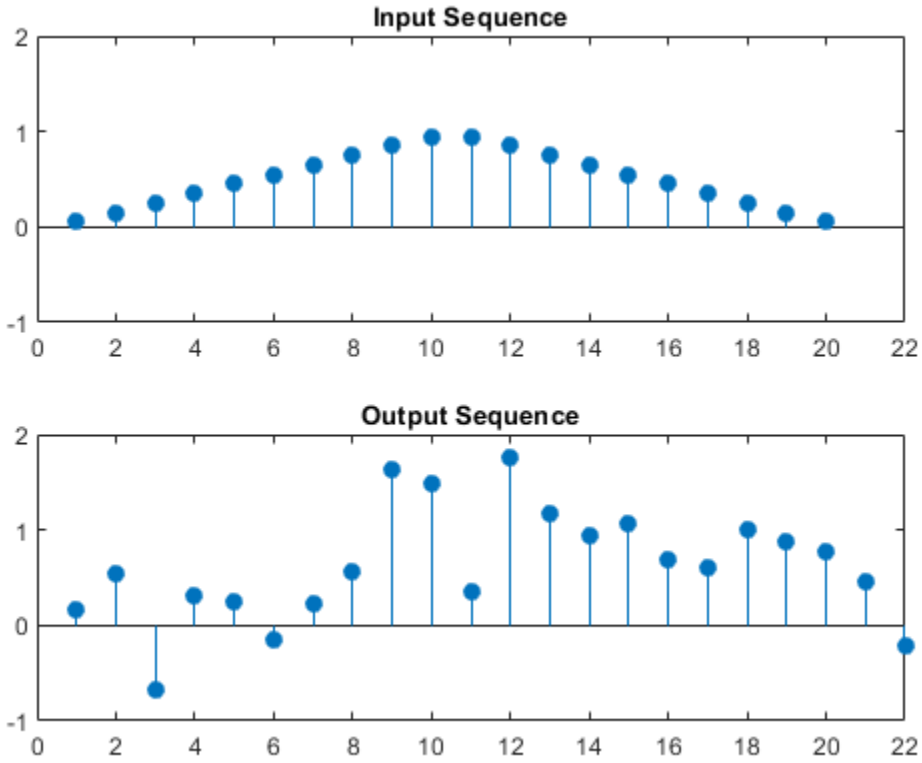
As expected, there is not perfect agreement between the theoretical cross-correlation and sample cross-correlation. The sample cross-correlation does accurately represent both the sign and magnitude of the theoretical cross-correlation sequence values at lag −1 and lag 1.

# Cross-Correlation of Delayed Signal in Noise

This example shows how to use the cross-correlation sequence to detect the time delay in a noise-corrupted sequence. The output sequence is a delayed version of the input sequence with additive white Gaussian noise. Create two sequences. One sequence is a delayed version of the other. The delay is 3 samples. Add $N(0, 0.3^2)$ white noise to the delayed signal. Use the sample cross-correlation sequence to detect the lag.

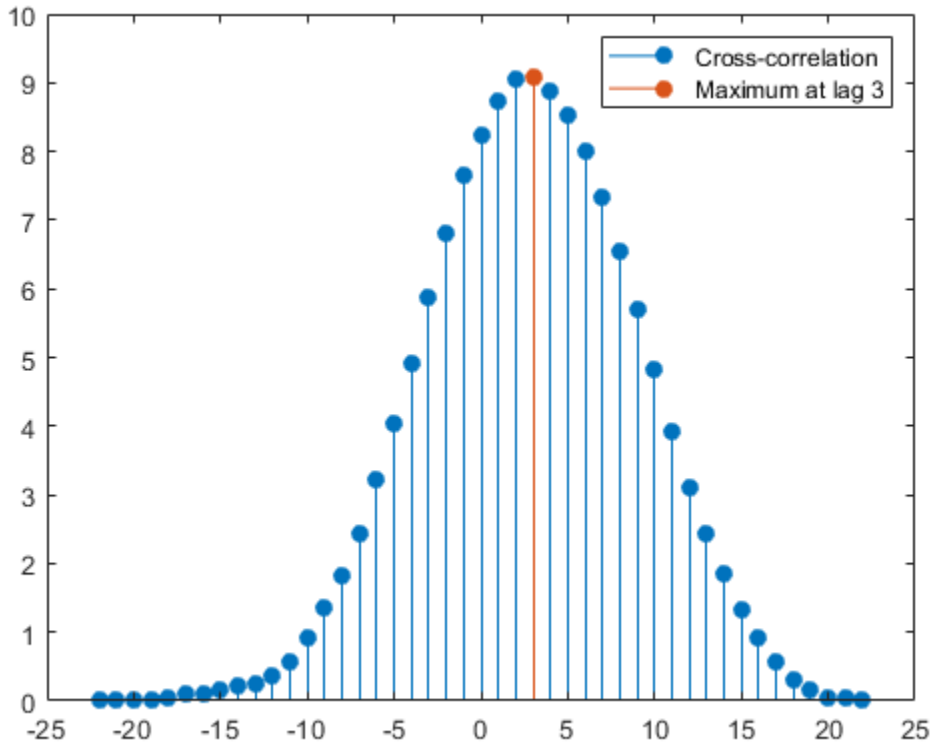
Create and plot the signals. Set the random number generator to the default settings for reproducible results.

```
rng default

x = triang(20);
y = [zeros(3,1);x]+0.3*randn(length(x)+3,1);

subplot(2,1,1)
stem(x,'filled')
axis([0 22 -1 2])
title('Input Sequence')

subplot(2,1,2)
stem(y,'filled')
axis([0 22 -1 2])
title('Output Sequence')
```



Obtain the sample cross-correlation sequence and use the maximum absolute value to estimate the lag. Plot the sample cross-correlation sequence. The maximum cross correlation sequence value occurs at lag 3, as expected.

```
[xc,lags] = xcorr(y,x);
[~,I] = max(abs(xc));

figure
stem(lags,xc,'filled')
hold on
stem(lags(I),xc(I),'filled')
hold off
legend(["Cross-correlation",sprintf('Maximum at lag %d',lags(I))])
```




Confirm the result using the `finddelay` function.

```
finddelay(x,y)
```

```
ans = 3
```

## See Also

`finddelay` | `xcorr`

# Cross-Correlation of Phase-Lagged Sine Wave

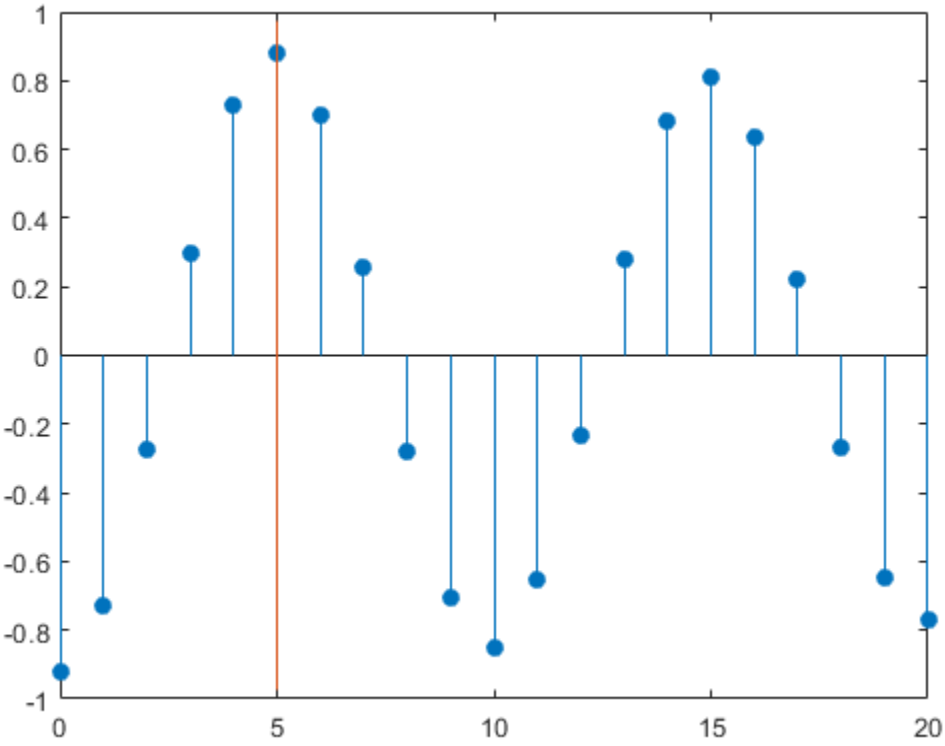
This example shows how to use the cross-correlation sequence to estimate the phase lag between two sine waves. The theoretical cross-correlation sequence of two sine waves at the same frequency also oscillates at that frequency. Because the sample cross-correlation sequence uses fewer and fewer samples at larger lags, the sample cross-correlation sequence also oscillates at the same frequency, but the amplitude decays as the lag increases.

Create two sine waves with frequencies of $2\pi/10$ rad/sample. The starting phase of one sine wave is 0, while the starting phase of the other sine wave is $-\pi$ radians. Add $N(0, 0.25^2)$ white noise to the sine wave with the phase lag of $\pi$ radians. Set the random number generator to the default settings for reproducible results.

```
rng default

t = 0:99;
x = cos(2*pi*1/10*t);
y = cos(2*pi*1/10*t-pi)+0.25*randn(size(t));
```

Obtain the sample cross-correlation sequence for two periods of the sine wave (10 samples). Plot the cross-correlation sequence and mark the known lag between the two sine waves (5 samples).

```
[xc,lags] = xcorr(y,x,20,'coeff');

stem(lags(21:end),xc(21:end),'filled')

hold on
plot([5 5],[-1 1])

ax = gca;
ax.XTick = 0:5:20;
```




You see that the cross-correlation sequence peaks at lag 5 as expected and oscillates with a period of 10 samples.

## See Also

`xcorr`

# Multirate Signal Processing


- “Downsampling — Signal Phases” on page 12-2
- “Downsampling — Aliasing” on page 12-5
- “Filtering Before Downsampling” on page 12-9
- “Upsampling — Imaging Artifacts” on page 12-11
- “Filtering After Upsampling — Interpolation” on page 12-13
- “Simulate a Sample-and-Hold System” on page 12-15
- “Change Signal Sample Rate” on page 12-20

# Downsampling — Signal Phases

This example shows how to use `downsample` to obtain the *phases* of a signal. Downsampling a signal by $M$ can produce $M$ unique phases. For example, if you have a discrete-time signal, $x$, with $x(0)$ $x(1)$ $x(2)$ $x(3)$, ..., the $M$ phases of $x$ are $x(nM + k)$ with $k = 0,1, ..., M-1$.

The $M$ signals are referred to as the *polyphase* components of $x$.

Create a white noise vector and obtain the 3 polyphase components associated with downsampling by 3.

Reset the random number generator to the default settings to produce a repeatable result. Generate a white noise random vector and obtain the 3 polyphase components associated with downsampling by 3.

```
rng default
x = randn(36,1);
x0 = downsample(x,3,0);
x1 = downsample(x,3,1);
x2 = downsample(x,3,2);
```

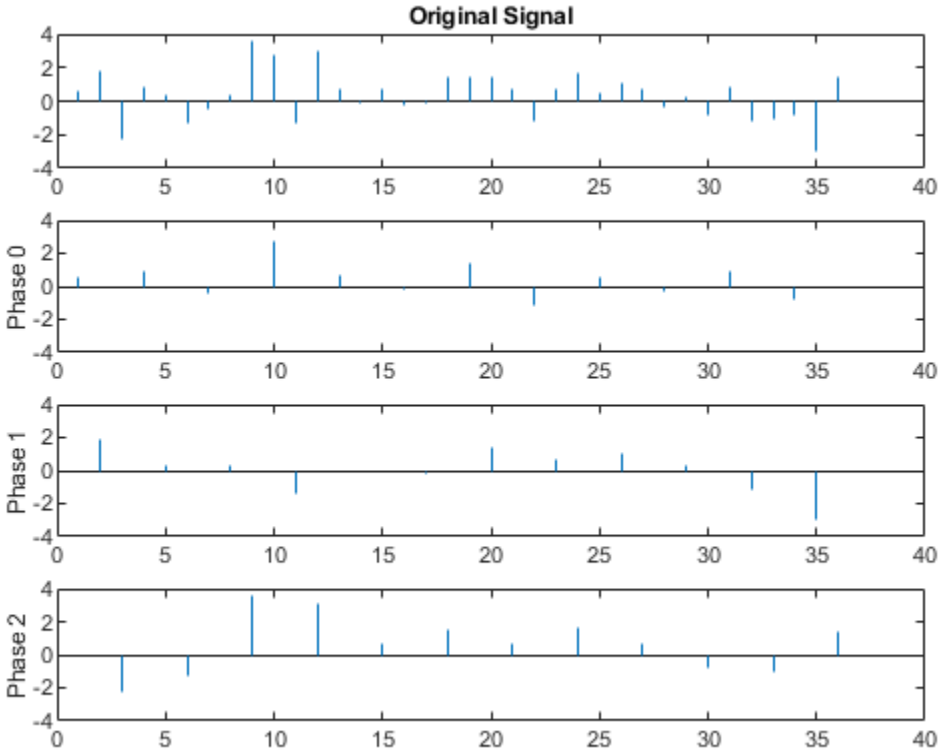
The polyphase components have length equal to 1/3 the original signal.

Upsample the polyphase components by 3 using `upsample`.

```
y0 = upsample(x0,3,0);
y1 = upsample(x1,3,1);
y2 = upsample(x2,3,2);
```

Plot the result.

```
subplot(4,1,1)
stem(x,'Marker','none')
title('Original Signal')
ylim([-4 4])

subplot(4,1,2)
stem(y0,'Marker','none')
ylabel('Phase 0')
ylim([-4 4])

subplot(4,1,3)
stem(y1,'Marker','none')
ylabel('Phase 1')
ylim([-4 4])

subplot(4,1,4)
stem(y2,'Marker','none')
ylabel('Phase 2')
ylim([-4 4])
```




If you sum the upsampled polyphase components you obtain the original signal.

Create a discrete-time sinusoid and obtain the 2 polyphase components associated with downsampling by 2.

Create a discrete-time sine wave with an angular frequency of $\pi/4$ rad/sample. Add a DC offset of 2 to the sine wave to help with visualization of the polyphase components. Downsample the sine wave by 2 to obtain the even and odd polyphase components.

```
n = 0:127;
x = 2+cos(pi/4*n);
x0 = downsample(x,2,0);
x1 = downsample(x,2,1);
```

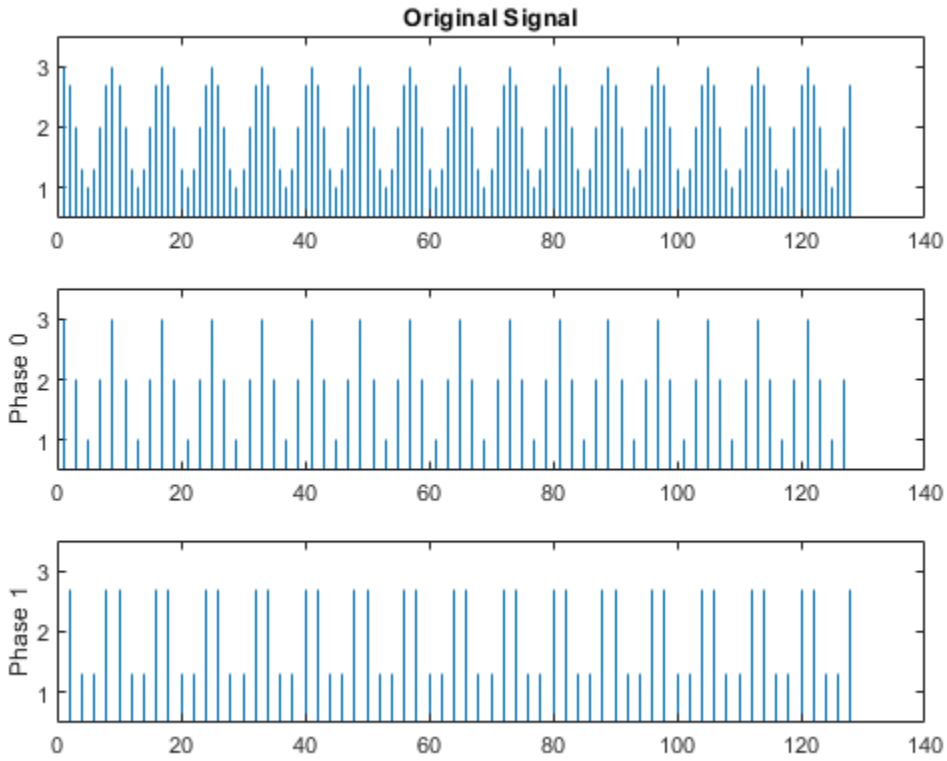
Upsample the two polyphase components.

```
y0 = upsample(x0,2,0);
y1 = upsample(x1,2,1);
```

Plot the upsampled polyphase components along with the original signal for comparison.

```
subplot(3,1,1)
stem(x,'Marker','none')
ylim([0.5 3.5])
title('Original Signal')

subplot(3,1,2)
```

```
stem(y0,'Marker','none')
ylim([0.5 3.5])
ylabel('Phase 0')

subplot(3,1,3)
stem(y1,'Marker','none')
ylim([0.5 3.5])
ylabel('Phase 1')
```




If you sum the two upsampled polyphase components (Phase 0 and Phase 1), you obtain the original sine wave.

## See Also

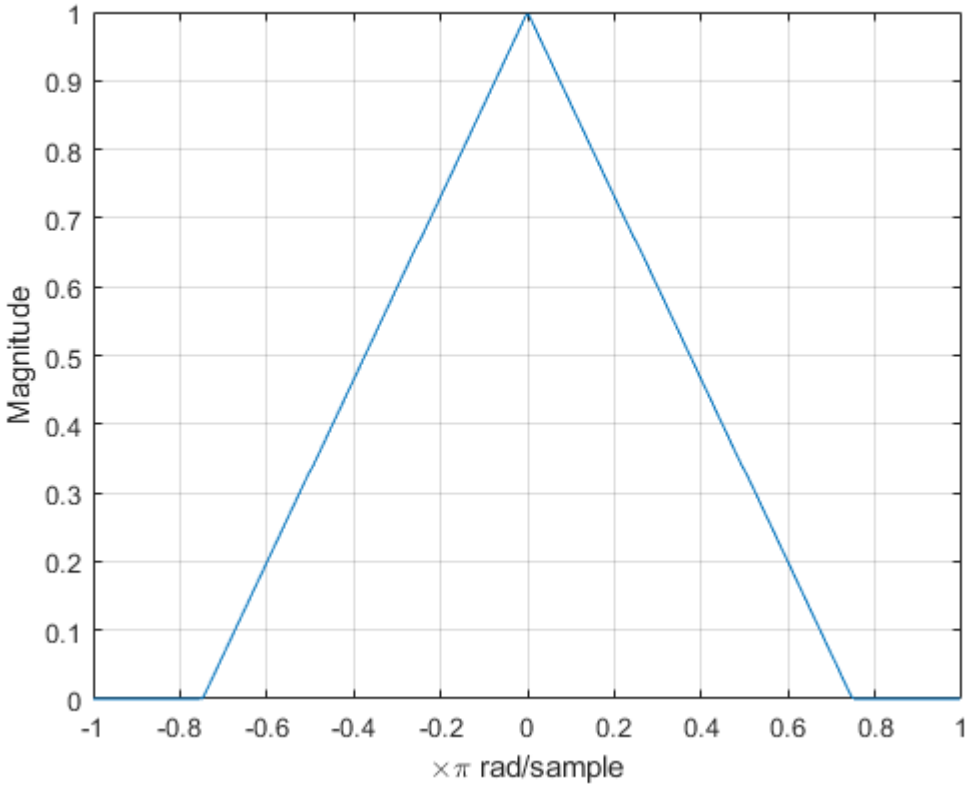
`downsample` | `upsample`

# Downsampling — Aliasing

This example shows how to avoid aliasing when downsampling a signal. If a discrete-time signal's baseband spectral support is not limited to an interval of width $2\pi/M$ radians, downsampling by $M$ results in aliasing. Aliasing is the distortion that occurs when overlapping copies of the signal's spectrum are added together. The more the signal's baseband spectral support exceeds $2\pi/M$ radians, the more severe the aliasing. Demonstrate aliasing in a signal downsampled by two. The signal's baseband spectral support exceed $\pi$ radians in width.

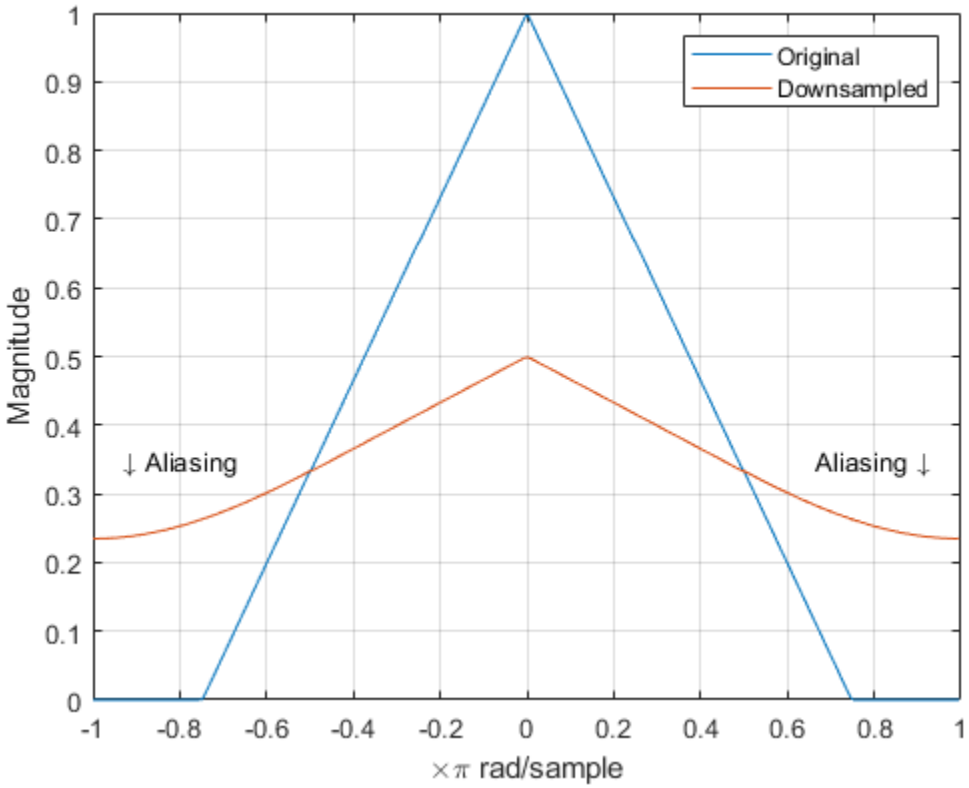
Create a signal with baseband spectral support equal to $3\pi/2$ radians. Use `fir2` to design the signal. Plot the signal's spectrum. The signal's baseband spectral support exceeds $[-\pi/2, \pi/2]$.

```
f = [0 0.2500 0.5000 0.7500 1.0000];
a = [1.00 0.6667 0.3333 0 0];

nf = 512;
b1 = fir2(nf-1,f,a);
Hx = fftshift(freqz(b1,1,nf,'whole'));

omega = -pi:2*pi/nf:pi-2*pi/nf;
plot(omega/pi,abs(Hx))
grid
xlabel('\times\pi rad/sample')
ylabel('Magnitude')
```



Downsample the signal by a factor of 2 and plot the downsampled signal's spectrum with the spectrum of the original signal. In addition to an amplitude scaling of the spectrum, the superposition of overlapping spectral replicas causes distortion of the original spectrum for $|\omega| > \pi/2$.

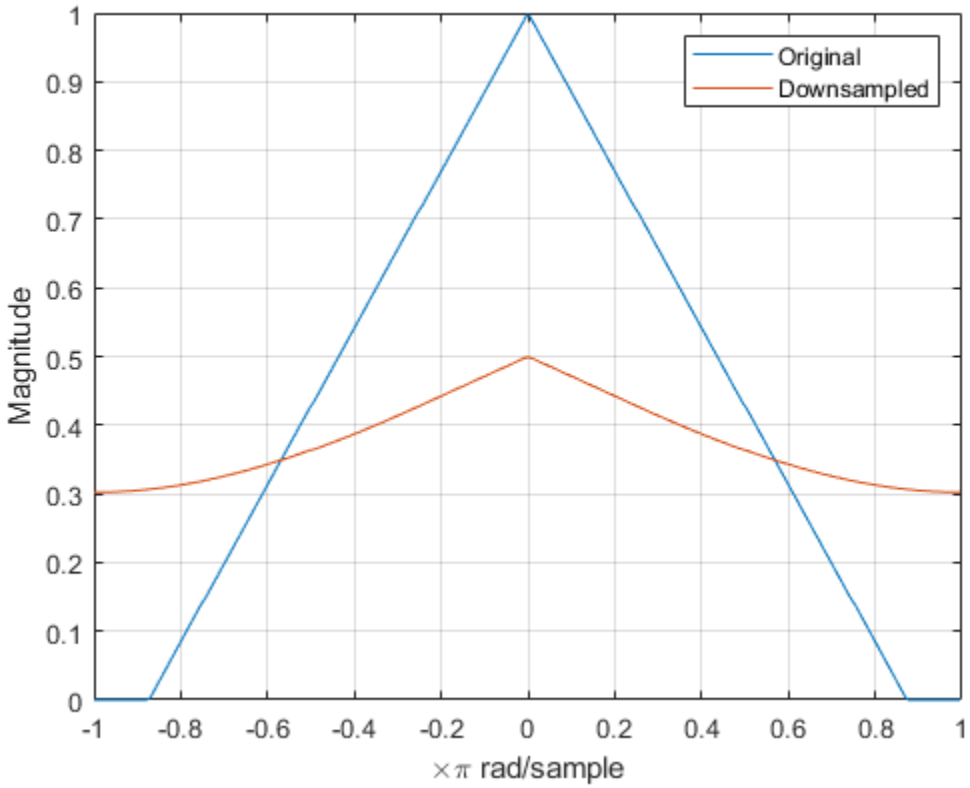
```
y = downsample(b1,2,0);
Hy = fftshift(freqz(y,1,nf,'whole'));

hold on
plot(omega/pi,abs(Hy))
hold off
legend('Original','Downsampled')
text(2.5/pi*[-1 1],0.35*[1 1],{'\downarrow Aliasing','Aliasing \downarrow'}, ...
    'HorizontalAlignment','center')
```




Increase the baseband spectral support of the signal to $[-7\pi/8, 7\pi/8]$ and downsample the signal by 2. Plot the original spectrum along with the spectrum of the downsampled signal. The increased spectral width results in more pronounced aliasing in the spectrum of the downsampled signal because more signal energy is outside $[-\pi/2, \pi/2]$.

```
f = [0 0.2500 0.5000 0.7500 7/8 1.0000];
a = [1.00 0.7143 0.4286 0.1429 0 0];

b2 = fir2(nf-1,f,a);
Hx = fftshift(freqz(b2,1,nf,'whole'));

plot(omega/pi,abs(Hx))
grid
```

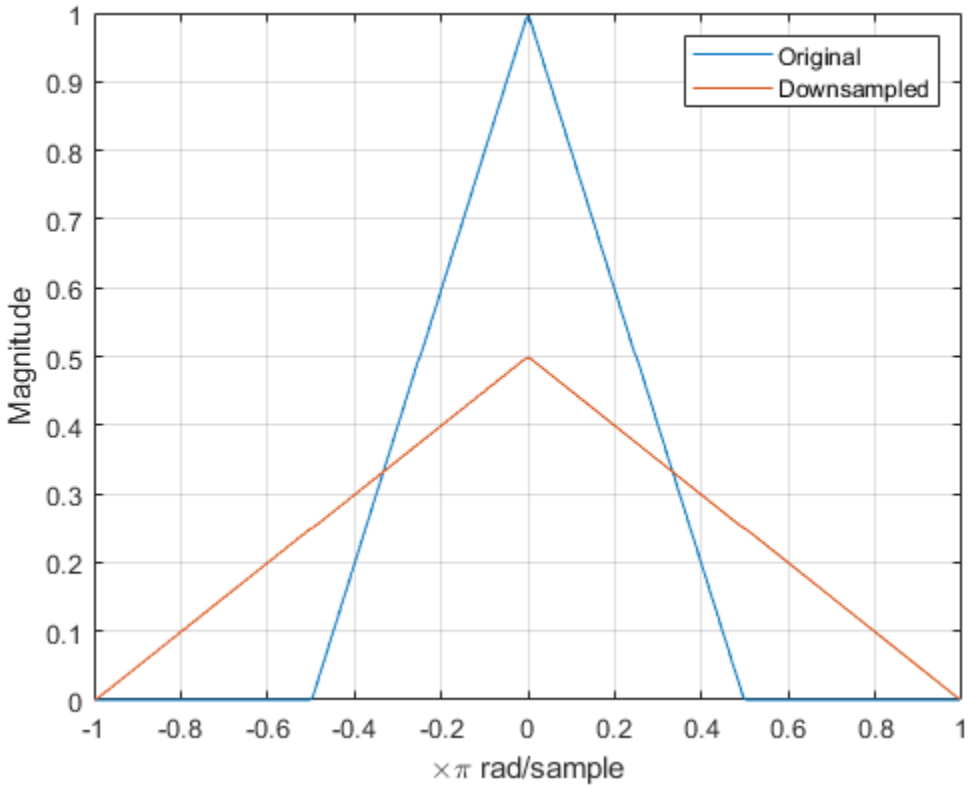
```
xlabel('\times\pi rad/sample')
ylabel('Magnitude')

y = downsample(b2,2,0);
Hy = fftshift(freqz(y,1,nf,'whole'));

hold on
plot(omega/pi,abs(Hy))
hold off
legend('Original','Downsampled')
```




Finally, construct a signal with baseband spectral support limited to $[-\pi/2, \pi/2]$. Downsample the signal by a factor of 2 and plot the spectrum of the original and downsampled signals. The downsampled signal is full band. The spectrum of the downsampled signal is a stretched and scaled version of the original spectrum, but the shape is preserved because the spectral copies do not overlap. There is no aliasing.

```
f = [0 0.250 0.500 0.7500 1];
a = [1.0000 0.5000 0 0 0];

b3 = fir2(nf-1,f,a);
Hx = fftshift(freqz(b3,1,nf,'whole'));

plot(omega/pi,abs(Hx))
grid
xlabel('\times\pi rad/sample')
ylabel('Magnitude')
```

```
y = downsample(b3,2,0);
Hy = fftshift(freqz(y,1,nf,'whole'));

hold on
plot(omega/pi,abs(Hy))
hold off
legend('Original','Downsampled')
```




## See Also

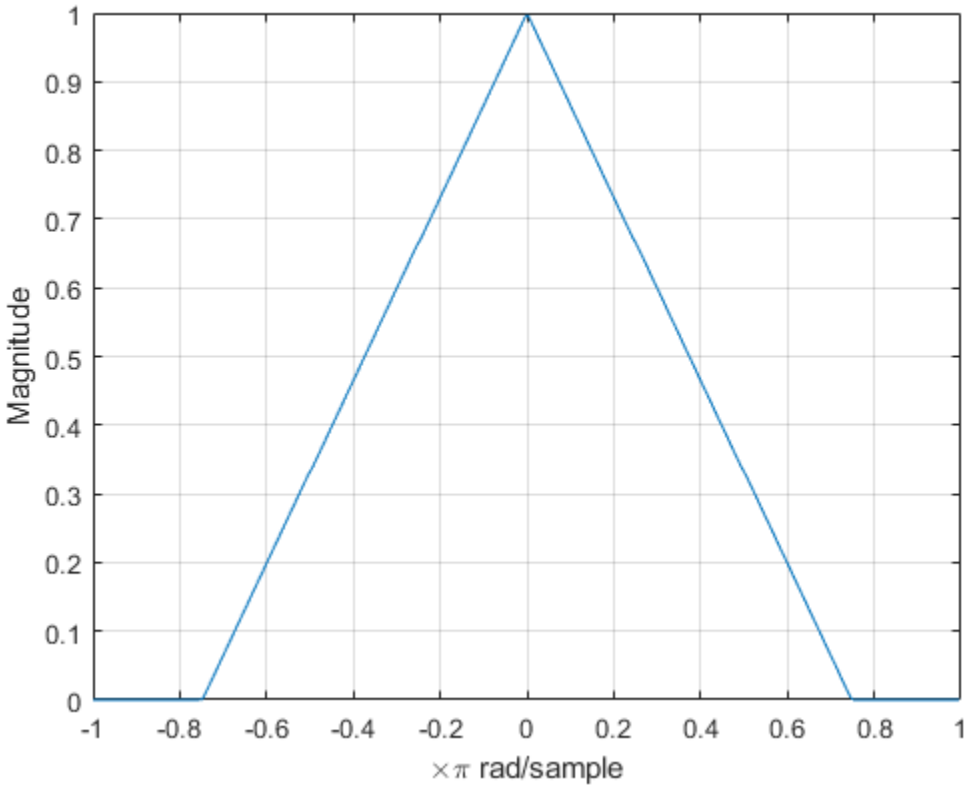
downsample | fir2 | freqz

# Filtering Before Downsampling

This example shows how to filter before downsampling to mitigate the distortion caused by aliasing. You can use `decimate` or `resample` to filter and downsample with one function. Alternatively, you can lowpass filter your data and then use `downsample`. Create a signal with baseband spectral support greater than $\pi$ radians. Use `decimate` to filter the signal with a 10th-order Chebyshev type I lowpass filter prior to downsampling.

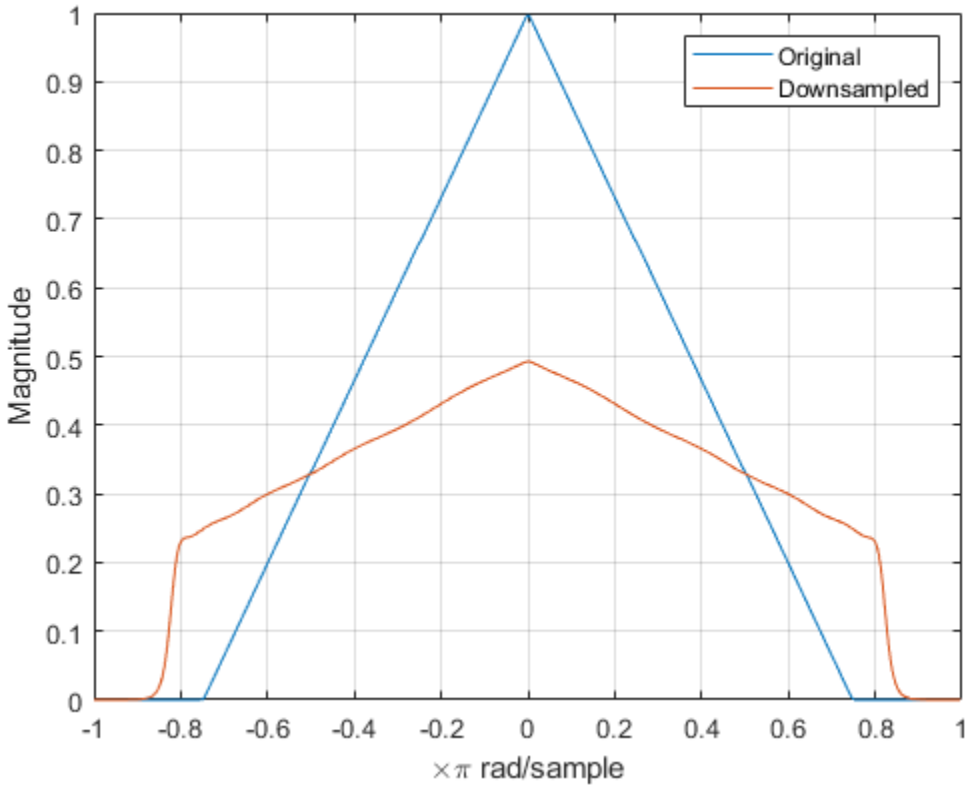
Create the signal and plot the magnitude spectrum.

```
f = [0 0.2500 0.5000 0.7500 1.0000];
a = [1.00 0.6667 0.3333 0 0];

nf = 512;
b = fir2(nf-1,f,a);
Hx = fftshift(freqz(b,1,nf,'whole'));

omega = -pi:2*pi/nf:pi-2*pi/nf;
plot(omega/pi,abs(Hx))
grid
xlabel('\times\pi rad/sample')
ylabel('Magnitude')
```




Filter the signal with a 10th-order type I Chebyshev lowpass filter and downsample by 2. Plot the magnitude spectra of the original signal along with the filtered and downsampled signal. The lowpass filter reduces the amount of aliasing distortion outside the interval $[-\pi/2, \pi/2]$.

```
y = decimate(b,2,10);
Hy = fftshift(freqz(y,1,nf,'whole'));

hold on
plot(omega/pi,abs(Hy))
legend('Original','Downsampled')
```




## See Also

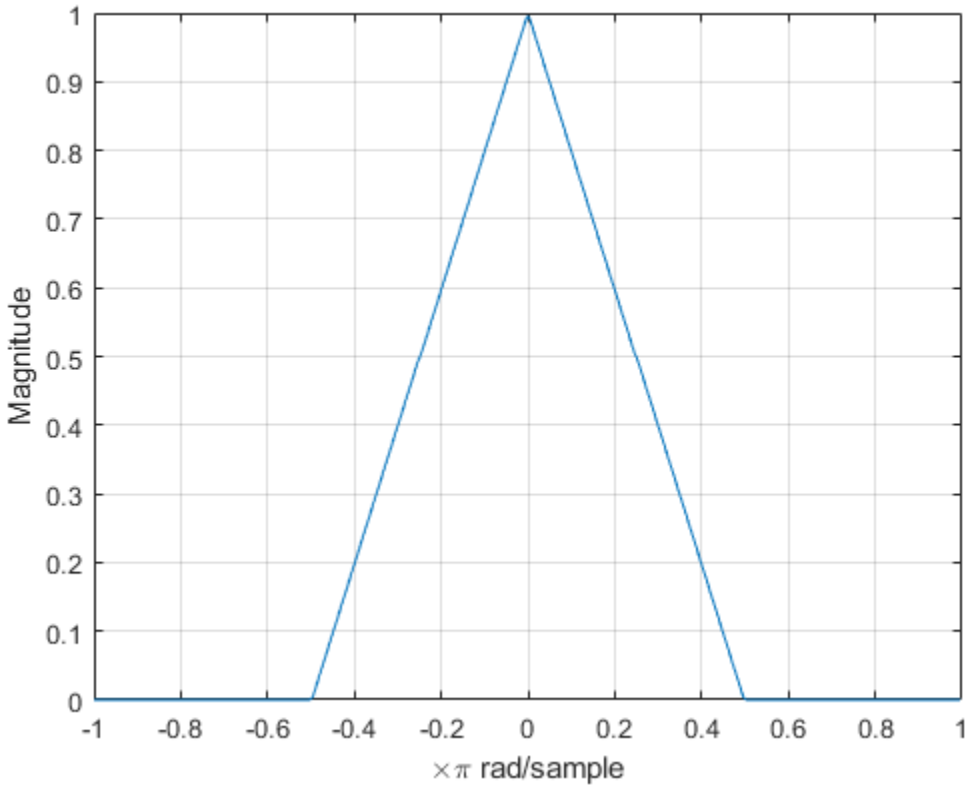
`decimate` | `fir2` | `freqz`

# Upsampling — Imaging Artifacts

This example shows how to upsample a signal and how upsampling can result in images. Upsampling a signal contracts the spectrum. For example, upsampling a signal by 2 results in a contraction of the spectrum by a factor of 2. Because the spectrum of a discrete-time signal is $2\pi$-periodic, contraction can cause replicas of the spectrum normally outside of the baseband to appear inside the interval $[-\pi, \pi]$.

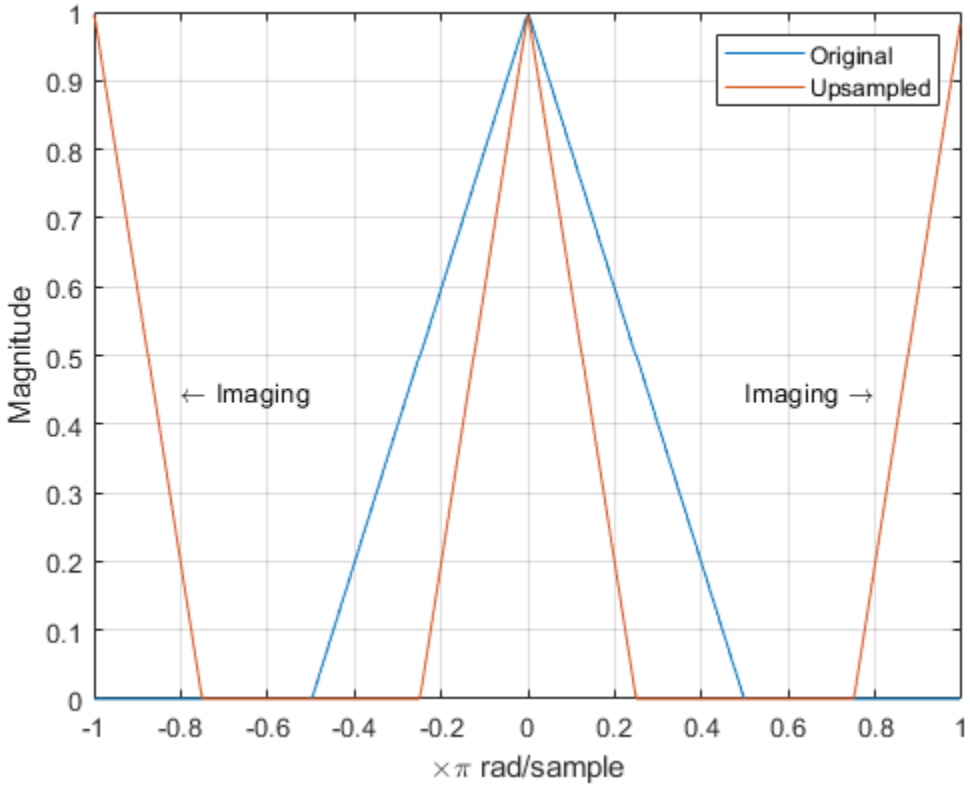
Create a discrete-time signal whose baseband spectral support is $[-\pi, \pi]$. Plot the magnitude spectrum.

```
f = [0 0.250 0.500 0.7500 1];
a = [1.0000 0.5000 0 0 0];

nf = 512;
b = fir2(nf-1,f,a);
Hx = fftshift(freqz(b,1,nf,'whole'));

omega = -pi:2*pi/nf:pi-2*pi/nf;
plot(omega/pi,abs(Hx))
grid
xlabel('\times\pi rad/sample')
ylabel('Magnitude')
```




Upsample the signal by 2. Plot the spectrum of the upsampled signal. The contraction of the spectrum has drawn subsequent periods of the spectrum into the interval $[-\pi, \pi]$.

```
y = upsample(b,2);
Hy = fftshift(freqz(y,1,nf,'whole'));

hold on
plot(omega/pi,abs(Hy))
hold off
legend('Original','Upsampled')
text(0.65*[-1 1],0.45*[1 1],["\leftarrow Imaging" "Imaging \rightarrow"], ...
    'HorizontalAlignment','center')
```




## See Also

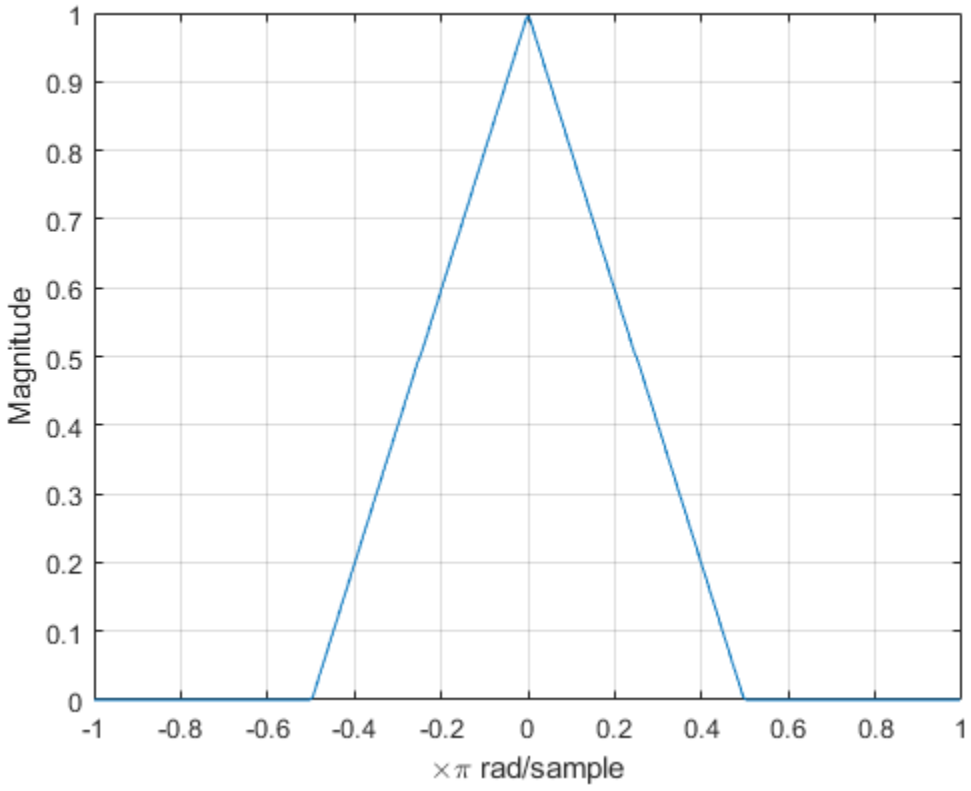
`fir2` | `freqz` | `upsample`

# Filtering After Upsampling — Interpolation

This example shows how to upsample a signal and apply a lowpass interpolation filter with `interp`. Upsampling by $L$ inserts $L$ – 1 zeros between every element of the original signal. Upsampling can create imaging artifacts. Lowpass filtering following upsampling can remove these imaging artifacts. In the time domain, lowpass filtering interpolates the zeros inserted by upsampling.

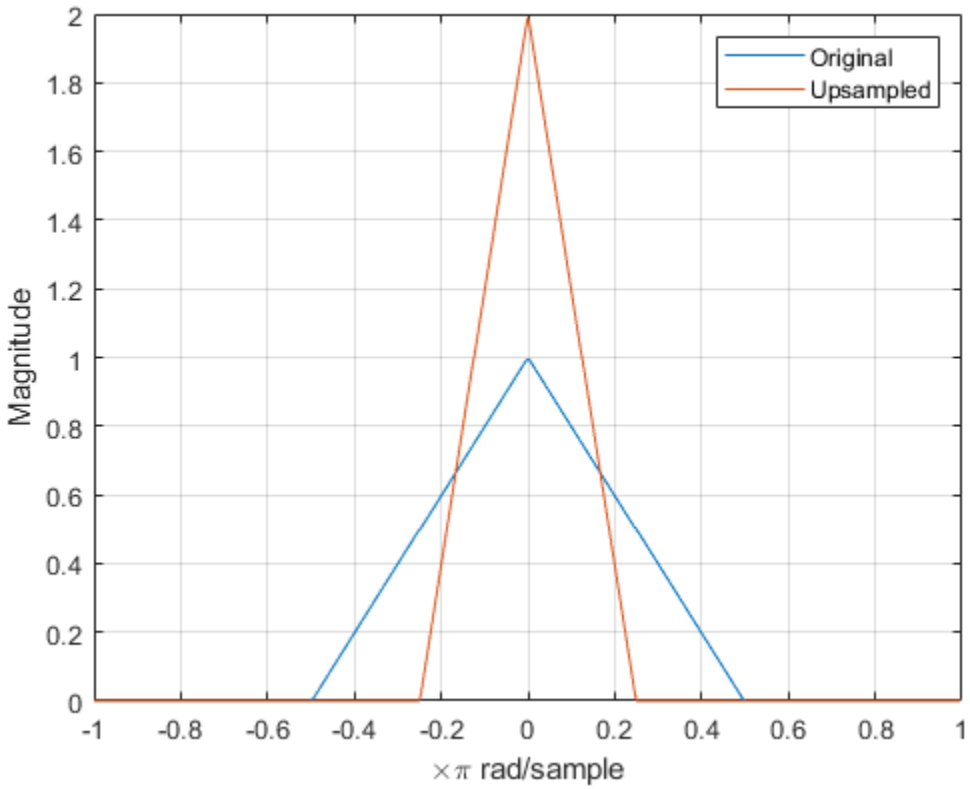
Create a discrete-time signal whose baseband spectral support is $[-\pi/2, \pi/2]$. Plot the magnitude spectrum.

```
f = [0 0.250 0.500 0.7500 1];
a = [1.0000 0.5000 0 0 0];

nf = 512;
b = fir2(nf-1,f,a);
Hx = fftshift(freqz(b,1,nf,'whole'));

omega = -pi:2*pi/nf:pi-2*pi/nf;
plot(omega/pi,abs(Hx))
grid
xlabel('\times\pi rad/sample')
ylabel('Magnitude')
```




Upsample the signal and apply a lowpass filter to remove the imaging artifacts. Plot the magnitude spectrum. Upsampling still contracts the spectrum, but the imaging artifacts are removed by the lowpass filter.

```
y = interp(b,2);
Hy = fftshift(freqz(y,1,nf,'whole'));

hold on
plot(omega/pi,abs(Hy))
hold off
legend('Original','Upsampled')
```




## See Also

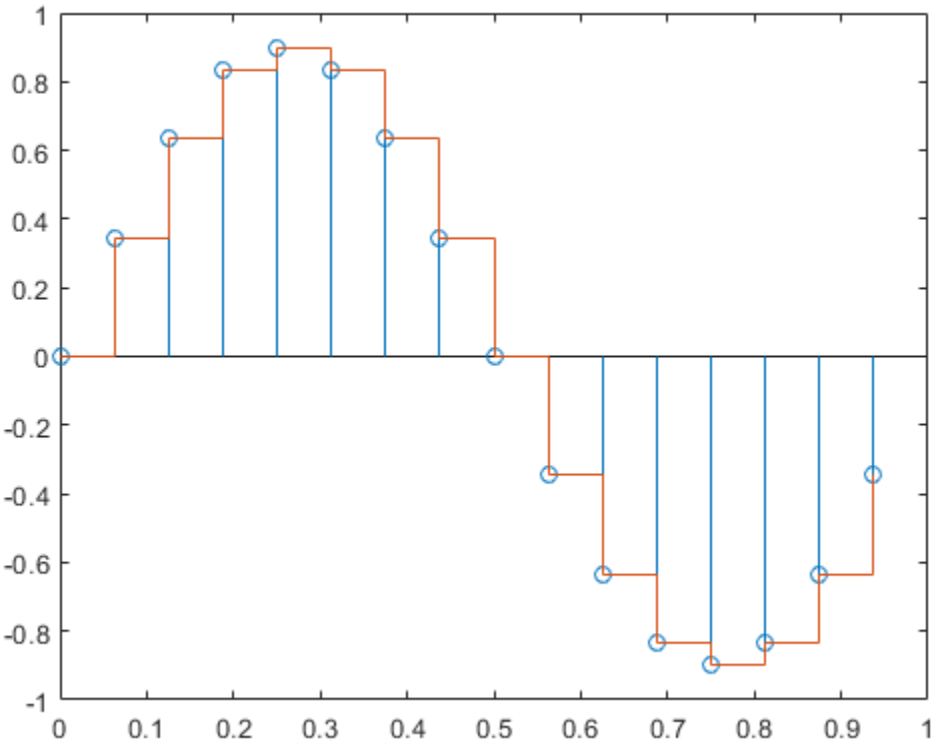
`fir2` | `freqz` | `interp`

# Simulate a Sample-and-Hold System

This example shows several ways to simulate the output of a sample-and-hold system by upsampling and filtering a signal.

Construct a sinusoidal signal. Specify a sample rate such that 16 samples correspond to exactly one signal period. Draw a stem plot of the signal. Overlay a stairstep graph for sample-and-hold visualization.

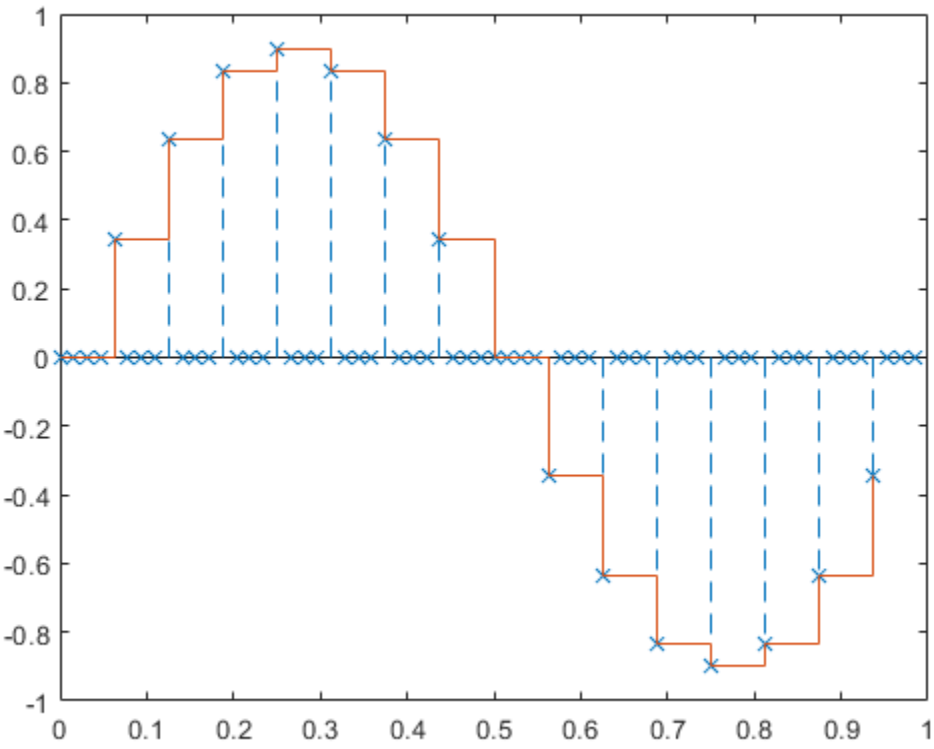
```
fs = 16;
t = 0:1/fs:1-1/fs;

x = .9*sin(2*pi*t);

stem(t,x)
hold on
stairs(t,x)
hold off
```




Upsample the signal by a factor of four. Plot the result alongside the original signal. `upsample` increases the sample rate of the signal by adding zeros between the existing samples.

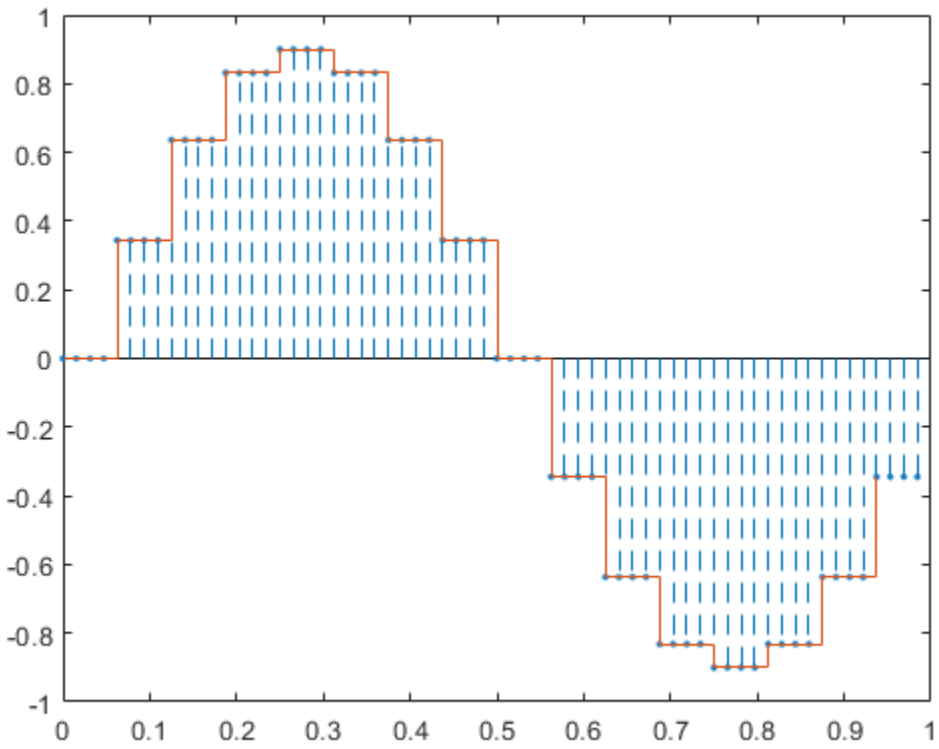
```
ups = 4;

fu = fs*ups;
tu = 0:1/fu:1-1/fu;
```

```
y = upsample(x,ups);

stem(tu,y,'--x')

hold on
stairs(t,x)
hold off
```



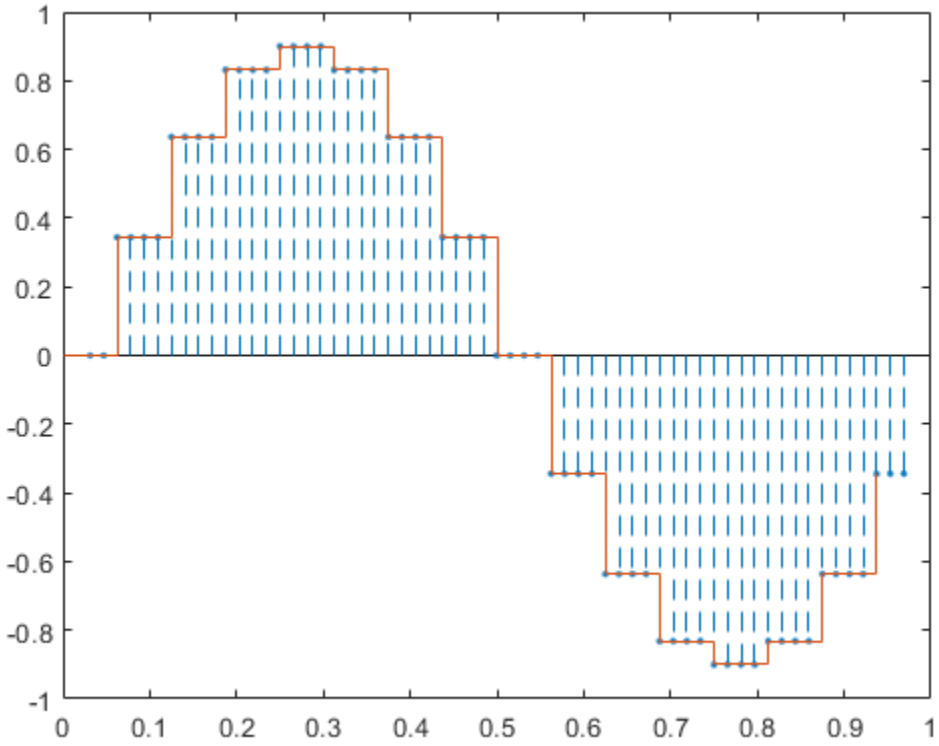

Filter with a moving-average FIR filter to fill in the zeros with sample-and-hold values.

```
h = ones(ups,1);

z = filter(h,1,y);

stem(tu,z,'--.')
hold on
stairs(t,x)
hold off
```



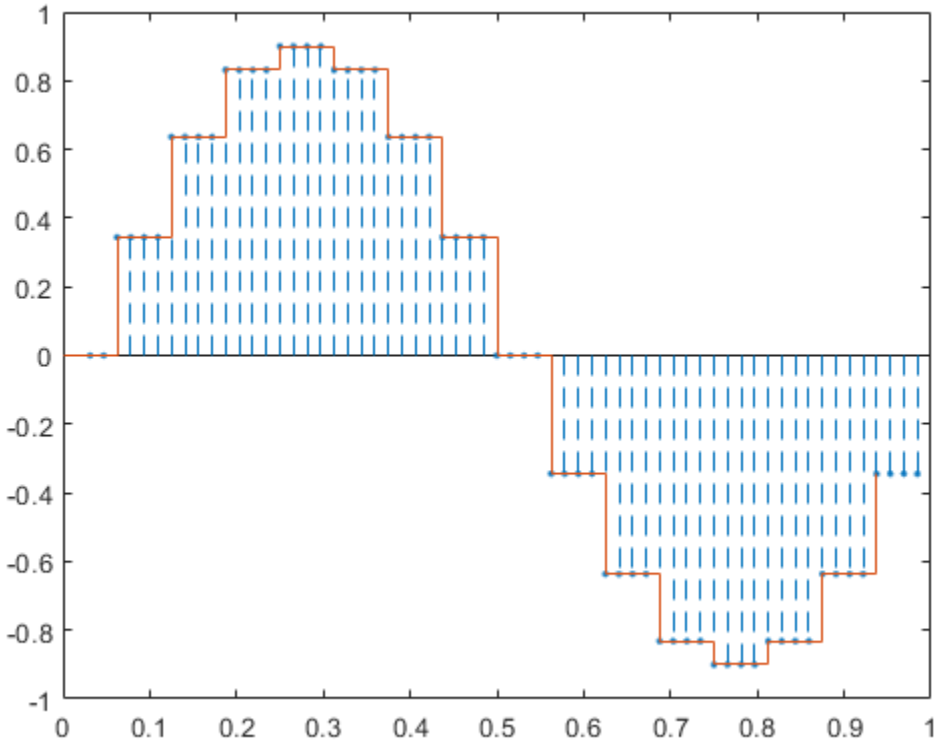

You can obtain the same behavior using the MATLAB® function `interp1` with nearest-neighbor interpolation. In that case, you must shift the origin to line up the sequence.

```
zi = interp1(t,x,tu,'nearest');

dl = floor(ups/2);

stem(tu(1+dl:end),zi(1:end-dl),'--.')
hold on
stairs(t,x)
hold off
```




The function `resample` produces the same result when you set the last input argument to zero.

```
q = resample(x,ups,1,0);

stem(tu(1+dl:end),q(1:end-dl),'--.')
hold on
stairs(t,x)
hold off
```




## See Also

`resample` | `upsample`

# Change Signal Sample Rate

This example shows how to change the sample rate of a signal. The example has two parts. Part one changes the sample rate of a sinusoidal input from 44.1 kHz to 48 kHz. This workflow is common in audio processing. The sample rate used on compact discs is 44.1 kHz, while the sample rate used on digital audio tape is 48 kHz. Part two changes the sample rate of a recorded speech sample from 7418 Hz to 8192 Hz.

Create an input signal consisting of a sum of sine waves sampled at 44.1 kHz. The sine waves have frequencies of 2, 4, and 8 kHz.

```
Fs = 44.1e3;
t = 0:1/Fs:1-1/Fs;
x = cos(2*pi*2000*t) + 1/2*sin(2*pi*4000*(t-pi/4)) + ...
    1/4*cos(2*pi*8000*t);
```

To change the sample rate from 44.1 to 48 kHz, you have to determine a rational number (ratio of integers), `P/Q`, such that `P/Q` times the original sample rate, 44100, is equal to 48000 within some specified tolerance.

To determine these factors, use `rat`. Input the ratio of the new sample rate, 48000, to the original sample rate, 44100.

```
[P,Q] = rat(48e3/Fs);
abs(P/Q*Fs-48000)
```

```
ans = 7.2760e-12
```

You see that `P/Q*Fs` only differs from the desired sample rate, 48000, on the order of $10^{-12}$.

Use the numerator and denominator factors obtained with `rat` as inputs to `resample` to output a waveform sampled at 48 kHz.

```
xnew = resample(x,P,Q);
```

If your computer can play audio, you can play the two waveforms. Set the volume to a comfortable level before you play the signals. Execute the `sound` commands separately so that you can hear the signal with the two different sample rates.

```
% sound(x,44100)
% sound(xnew,48000)
```

Change the sample rate of a speech sample from 7418 Hz to 8192 Hz. The speech signal is a recording of a speaker saying "MATLAB®".

Load the speech sample.

```
load mtlb
```

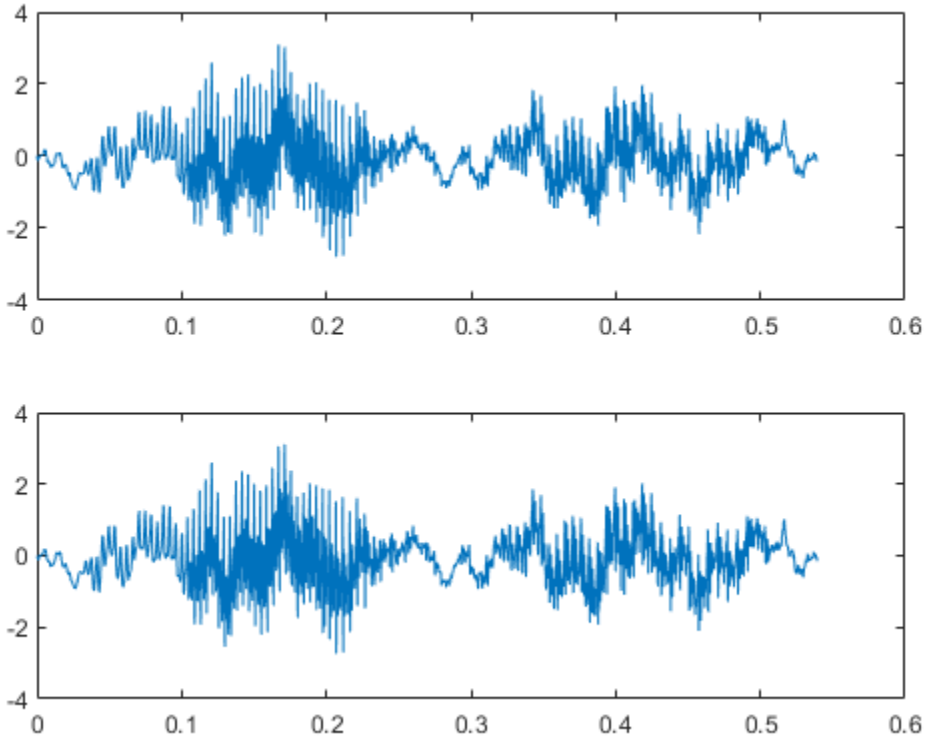
Loading the file `mtlb.mat` brings the speech signal, `mtlb`, and the sample rate, `Fs`, into the MATLAB workspace.

Determine a rational approximation to the ratio of the new sample rate, 8192, to the original sample rate. Use `rat` to determine the approximation.

```
[P,Q] = rat(8192/Fs);
```

Resample the speech sample at the new sample rate. Plot the two signals.

```
mtlb_new = resample(mtlb,P,Q);

subplot(2,1,1)
plot((0:length(mtlb)-1)/Fs,mtlb)
subplot(2,1,2)
plot((0:length(mtlb_new)-1)/(P/Q*Fs),mtlb_new)
```




If your computer has audio output capability, you can play the two waveforms at their respective sample rates for comparison. Set the volume on your computer to a comfortable listening level before playing the sounds. Execute the `sound` commands separately to compare the speech samples at the different sample rates.

```
% sound(mtlb,Fs)
% sound(mtlb_new,8192)
```

## See Also

`resample`

# Spectral Analysis


- “Power Spectral Density Estimates Using FFT” on page 13-2
- “Bias and Variability in the Periodogram” on page 13-9
- “Cross Spectrum and Magnitude-Squared Coherence” on page 13-17
- “Amplitude Estimation and Zero Padding” on page 13-20
- “Significance Testing for Periodic Component” on page 13-23
- “Frequency Estimation by Subspace Methods” on page 13-25
- “Frequency-Domain Linear Regression” on page 13-27
- “Measure Total Harmonic Distortion” on page 13-36
- “Measure Mean Frequency, Power, Bandwidth” on page 13-38
- “Periodogram of Data Set with Missing Samples” on page 13-43
- “Welch Spectrum Estimates” on page 13-46

# Power Spectral Density Estimates Using FFT

This example shows how to obtain nonparametric power spectral density (PSD) estimates equivalent to the periodogram using `fft`. The examples show you how to properly scale the output of `fft` for even-length inputs, for normalized frequency and hertz, and for one- and two-sided PSD estimates.

### Even-Length Input with Sample Rate

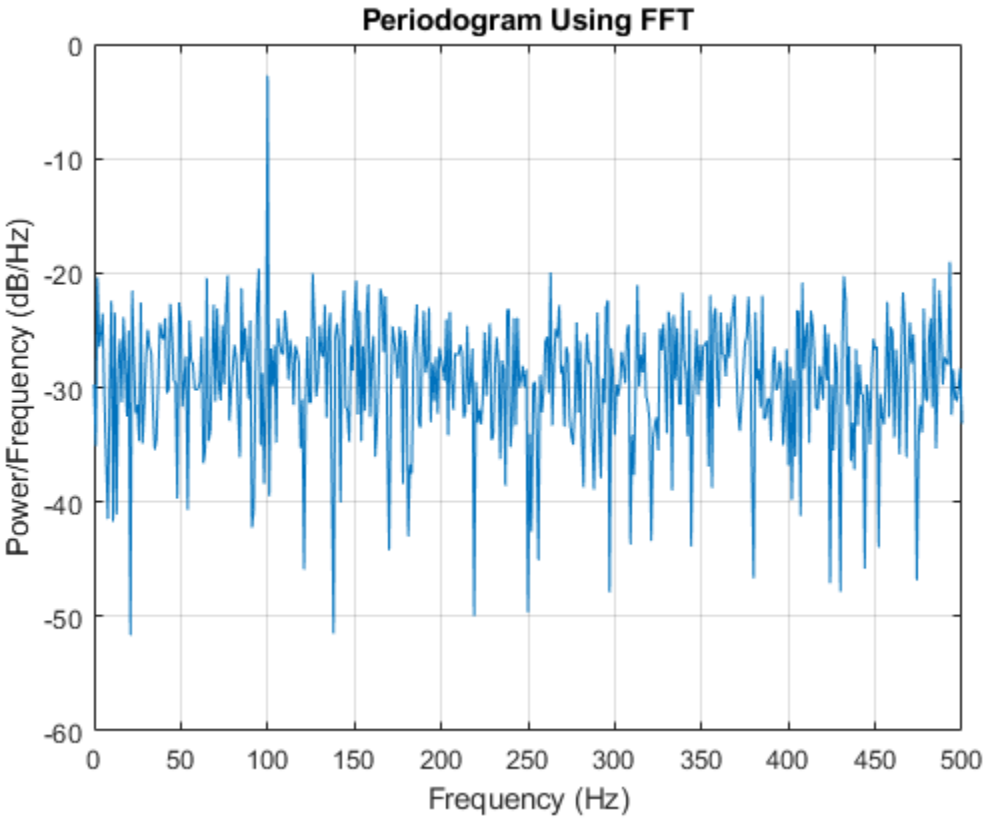
Obtain the periodogram for an even-length signal sampled at 1 kHz using both `fft` and `periodogram`. Compare the results.

Create a signal consisting of a 100 Hz sine wave in $N(0,1)$ additive noise. The sampling frequency is 1 kHz. The signal length is 1000 samples. Use the default settings of the random number generator for reproducible results.

```
rng default
Fs = 1000;
t = 0:1/Fs:1-1/Fs;
x = cos(2*pi*100*t) + randn(size(t));
```

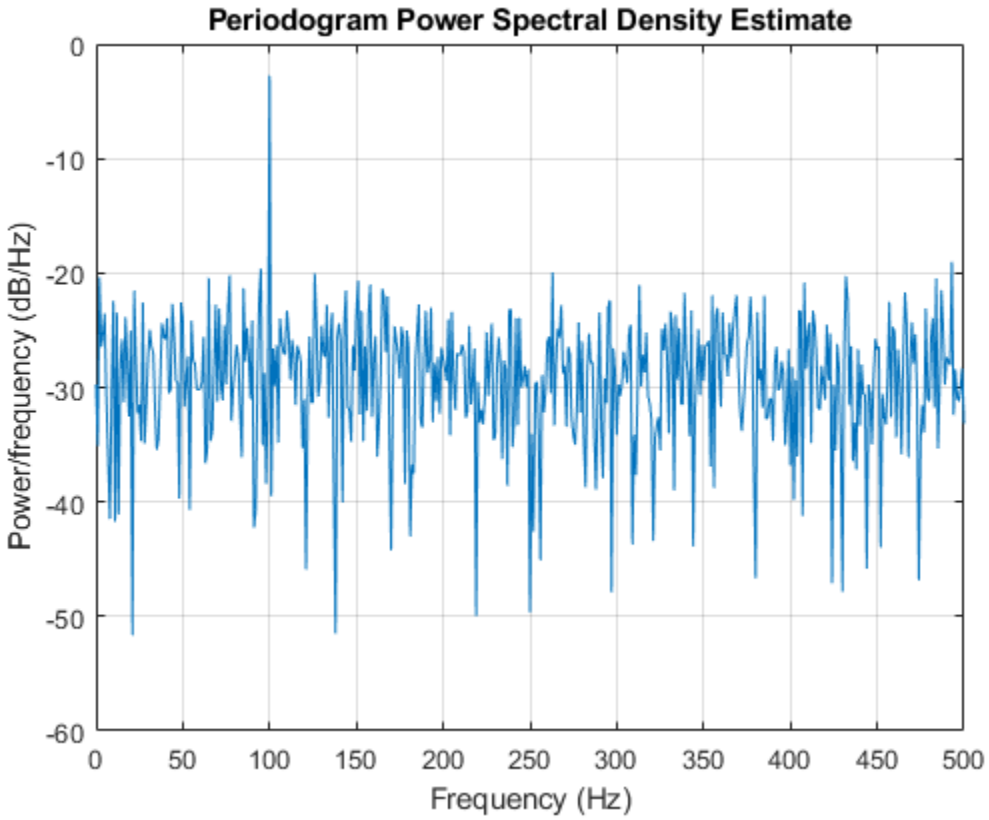
Obtain the periodogram using `fft`. The signal is real-valued and has even length. Because the signal is real-valued, you only need power estimates for the positive or negative frequencies. In order to conserve the total power, multiply all frequencies that occur in both sets — the positive and negative frequencies — by a factor of 2. Zero frequency (DC) and the Nyquist frequency do not occur twice. Plot the result.

```
N = length(x);
xdft = fft(x);
xdft = xdft(1:N/2+1);
psdx = (1/(Fs*N)) * abs(xdft).^2;
psdx(2:end-1) = 2*psdx(2:end-1);
freq = 0:Fs/length(x):Fs/2;

plot(freq,10*log10(psdx))
grid on
title('Periodogram Using FFT')
xlabel('Frequency (Hz)')
ylabel('Power/Frequency (dB/Hz)')
```




Compute and plot the periodogram using `periodogram`. Show that the two results are identical.

```
periodogram(x,rectwin(length(x)),length(x),Fs)
```




```
mxerr = max(psdx'-periodogram(x,rectwin(length(x)),length(x),Fs))
```

```
mxerr = 3.4694e-18
```

**Input with Normalized Frequency**

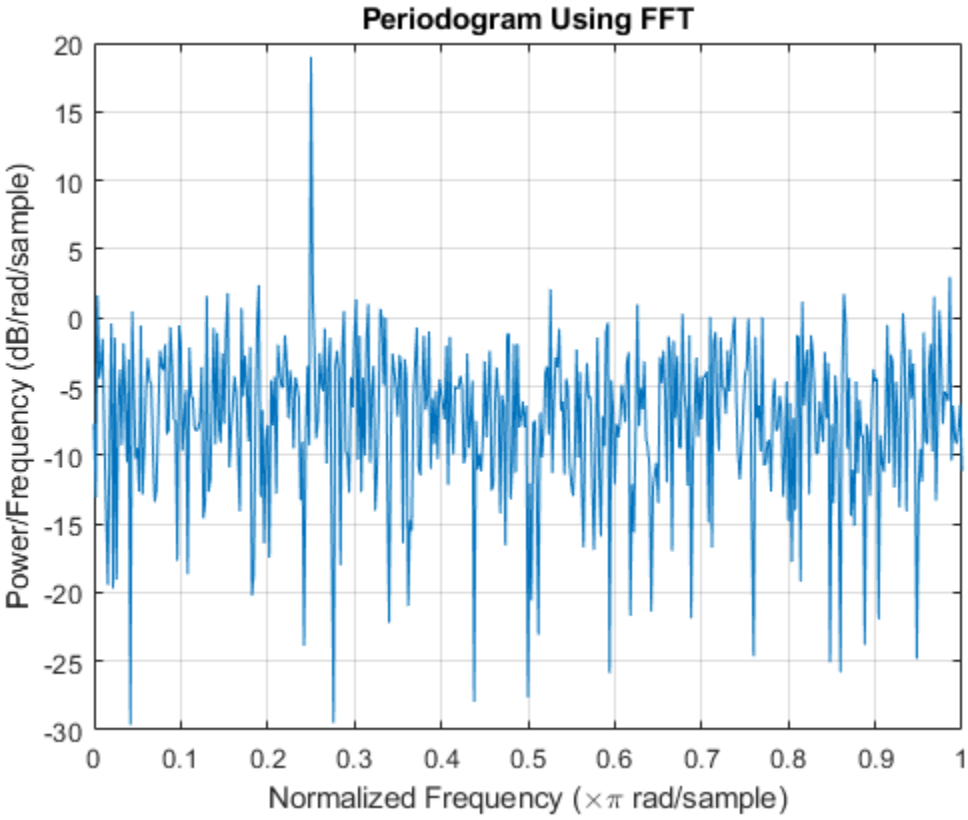
Use `fft` to produce a periodogram for an input using normalized frequency. Create a signal consisting of a sine wave in $N(0,1)$ additive noise. The sine wave has an angular frequency of $\pi/4$ rad/sample. Use the default settings of the random number generator for reproducible results.

```
rng default
n = 0:999;
x = cos(pi/4*n) + randn(size(n));
```

Obtain the periodogram using `fft`. The signal is real-valued and has even length. Because the signal is real-valued, you only need power estimates for the positive or negative frequencies. In order to conserve the total power, multiply all frequencies that occur in both sets — the positive and negative frequencies — by a factor of 2. Zero frequency (DC) and the Nyquist frequency do not occur twice. Plot the result.

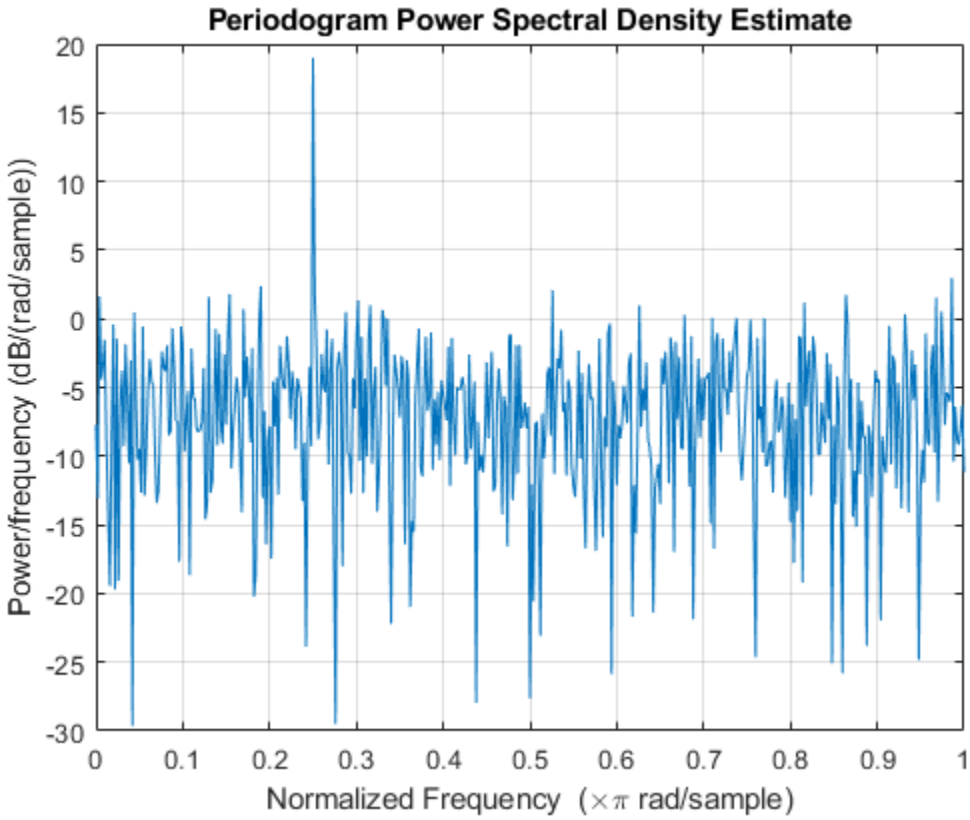
```
N = length(x);
xdft = fft(x);
xdft = xdft(1:N/2+1);
psdx = (1/(2*pi*N)) * abs(xdft).^2;
psdx(2:end-1) = 2*psdx(2:end-1);
freq = 0:(2*pi)/N:pi;
```

```
plot(freq/pi,10*log10(psdx))
grid on
title('Periodogram Using FFT')
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Power/Frequency (dB/rad/sample)')
```




Compute and plot the periodogram using `periodogram`. Show that the two results are identical.

```
periodogram(x,rectwin(length(x)),length(x))
```




```
mxerr = max(psdx'-periodogram(x,rectwin(length(x)),length(x)))
```

```
mxerr = 1.4211e-14
```

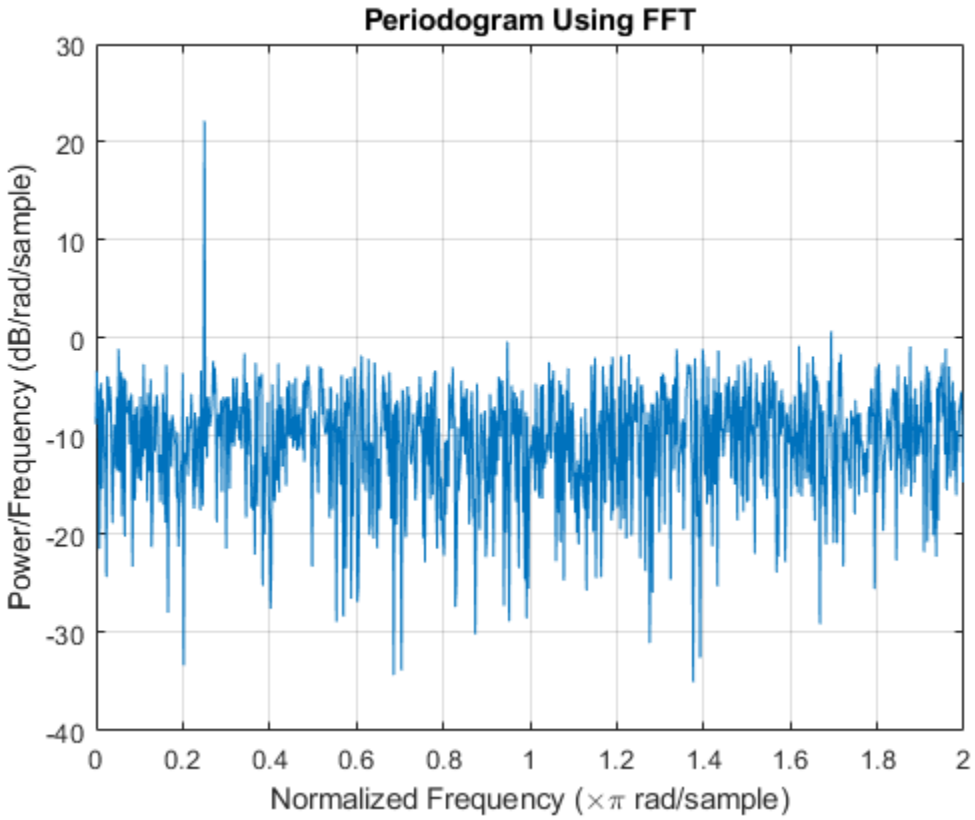
### Complex-Valued Input with Normalized Frequency

Use `fft` to produce a periodogram for a complex-valued input with normalized frequency. The signal is a complex exponential with an angular frequency of $\pi/4$ rad/sample in complex-valued $N(0,1)$ noise. Set the random number generator to the default settings for reproducible results.

```
rng default
n = 0:999;
x = exp(1j*pi/4*n) + [1 1j]*randn(2,length(n))/sqrt(2);
```

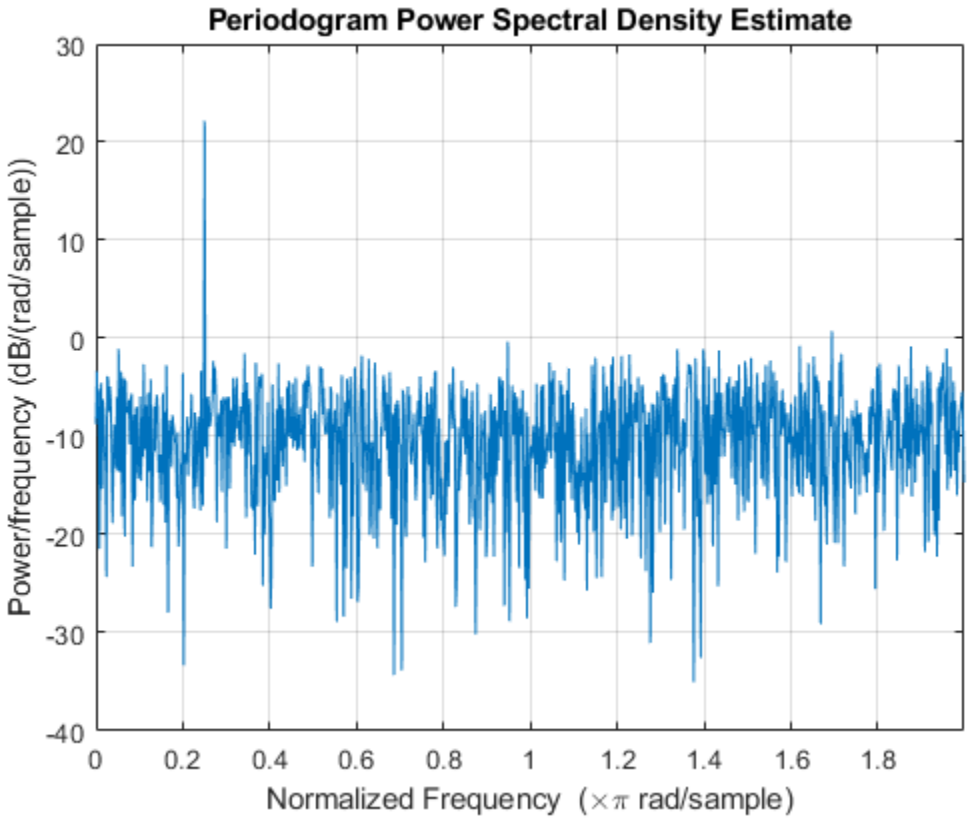
Use `fft` to obtain the periodogram. Because the input is complex-valued, obtain the periodogram from $[0, 2\pi)$ rad/sample. Plot the result.

```
N = length(x);
xdft = fft(x);
psdx = (1/(2*pi*N)) * abs(xdft).^2;
freq = 0:(2*pi)/N:2*pi-(2*pi)/N;

plot(freq/pi,10*log10(psdx))
grid on
title('Periodogram Using FFT')
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Power/Frequency (dB/rad/sample)')
```




Use `periodogram` to obtain and plot the periodogram. Compare the PSD estimates.

```
periodogram(x,rectwin(length(x)),length(x),'twosided')
```




```
mxerr = max(psdx'-periodogram(x,rectwin(length(x)),length(x),'twosided'))
```

```
mxerr = 4.4409e-16
```

## See Also

**Apps**
**Signal Analyzer**

**Functions**
`fft` | `periodogram` | `pspectrum`

# Bias and Variability in the Periodogram

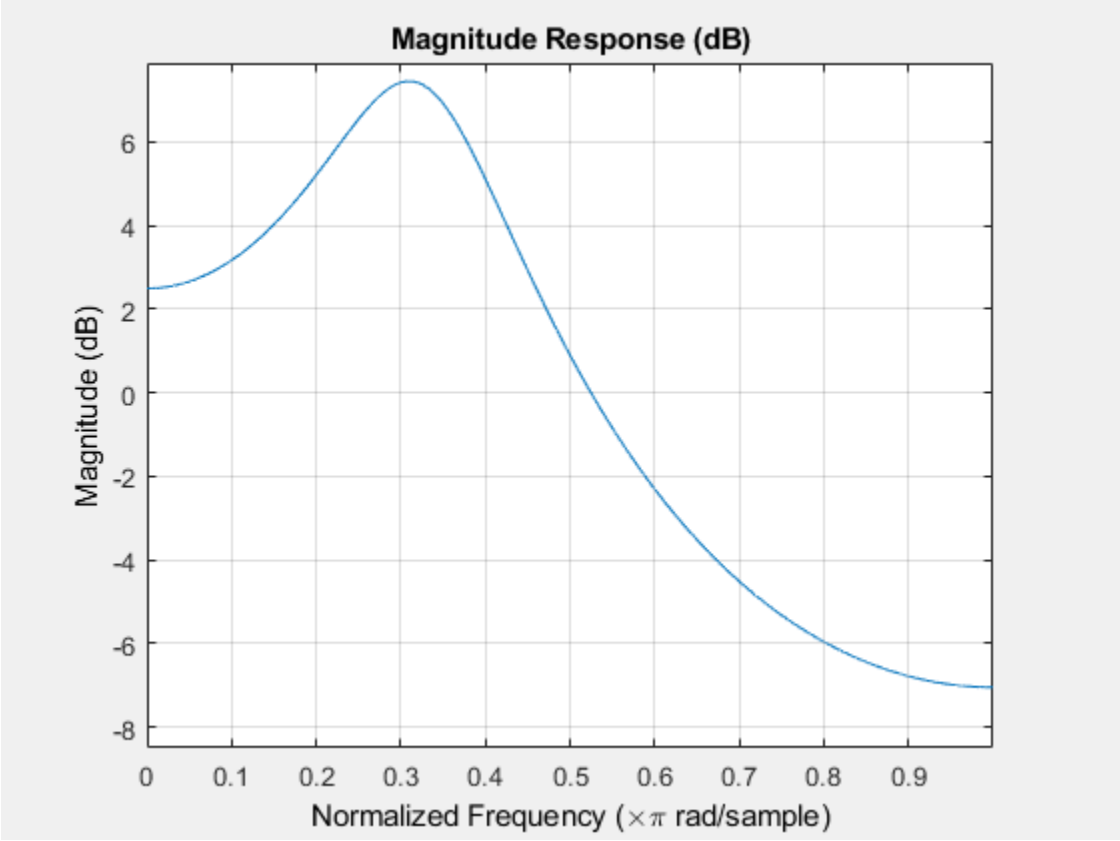
This example shows how to reduce bias and variability in the periodogram. Using a window can reduce the bias in the periodogram, and using windows with averaging can reduce variability.

Use wide-sense stationary autoregressive (AR) processes to show the effects of bias and variability in the periodogram. AR processes present a convenient model because their PSDs have closed-form expressions. Create an AR(2) model of the following form:

$$y(n) - 0.75y(n-1) + 0.5y(n-2) = \varepsilon(n),$$

where $\varepsilon(n)$ is a zero mean white noise sequence with some specified variance. In this example, assume the variance and the sampling period to be 1. To simulate the preceding AR(2) process, create an all-pole (IIR) filter. View the filter's magnitude response.

```
B2 = 1;
A2 = [1 -0.75 0.5];
fvtool(B2,A2)
```



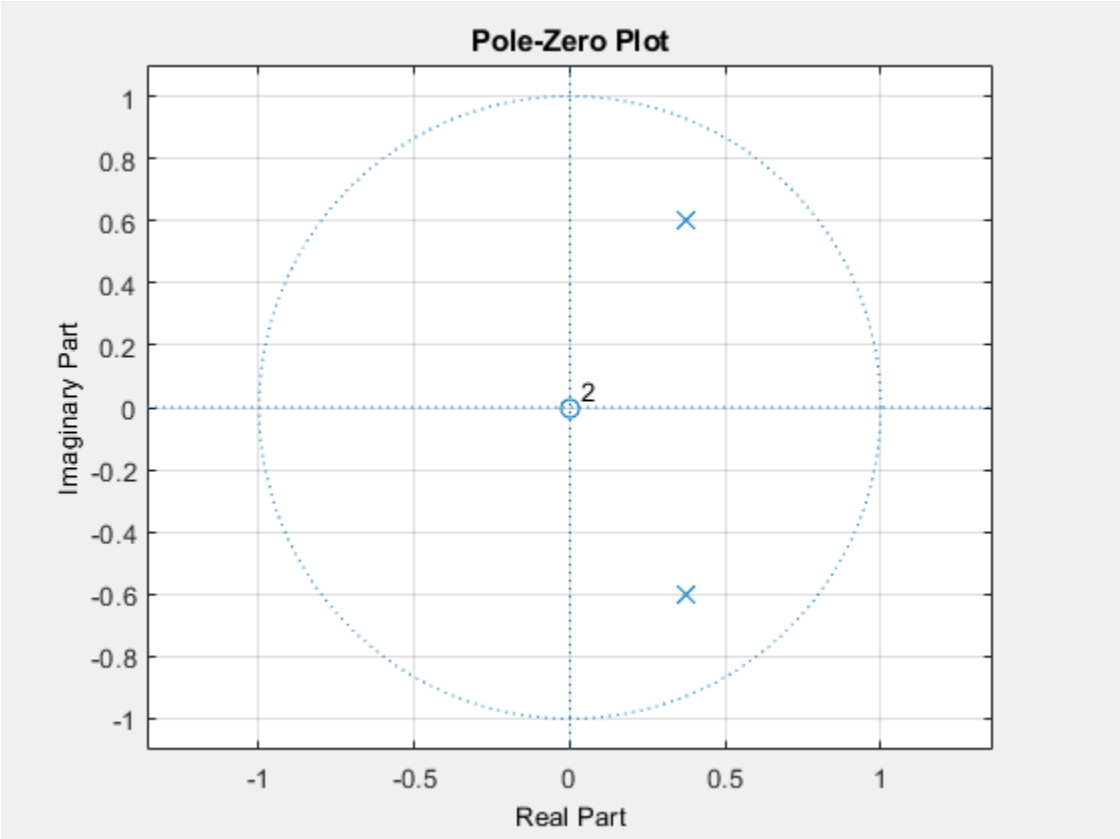

This process is bandpass. The dynamic range of the PSD is approximately 14.5 dB, as you can determine with the following code.

```
[H2,W2] = freqz(B2,A2,1e3,1);
dr2 = max(20*log10(abs(H2)))-min(20*log10(abs(H2)))
```

```
dr2 = 14.4984
```

By examining the placement of the poles, you see that this AR(2) process is stable. The two poles are inside the unit circle.

```
fvtool(B2,A2,'Analysis','polezero')
```

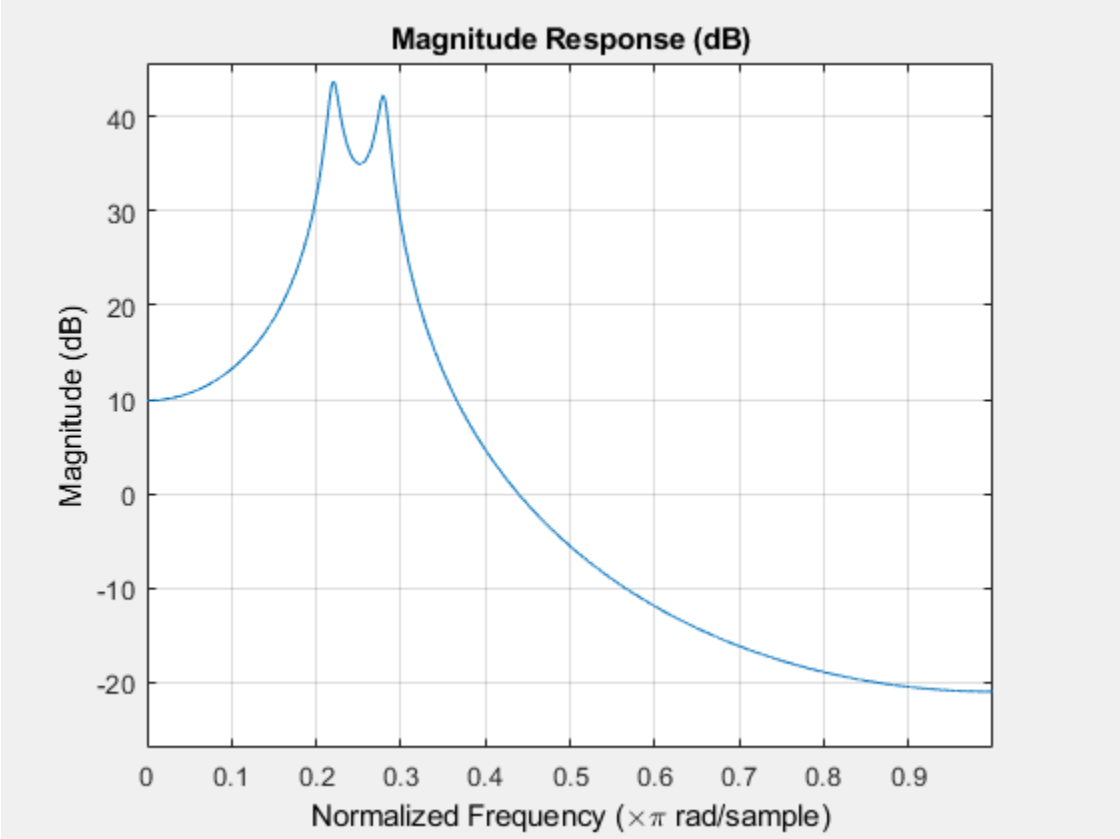



Next, create an AR(4) process described by the following equation:

$$y(n) - 2.7607y(n-1) + 3.8106y(n-2) - 2.6535y(n-3) + 0.9238y(n-4) = \varepsilon(n).$$

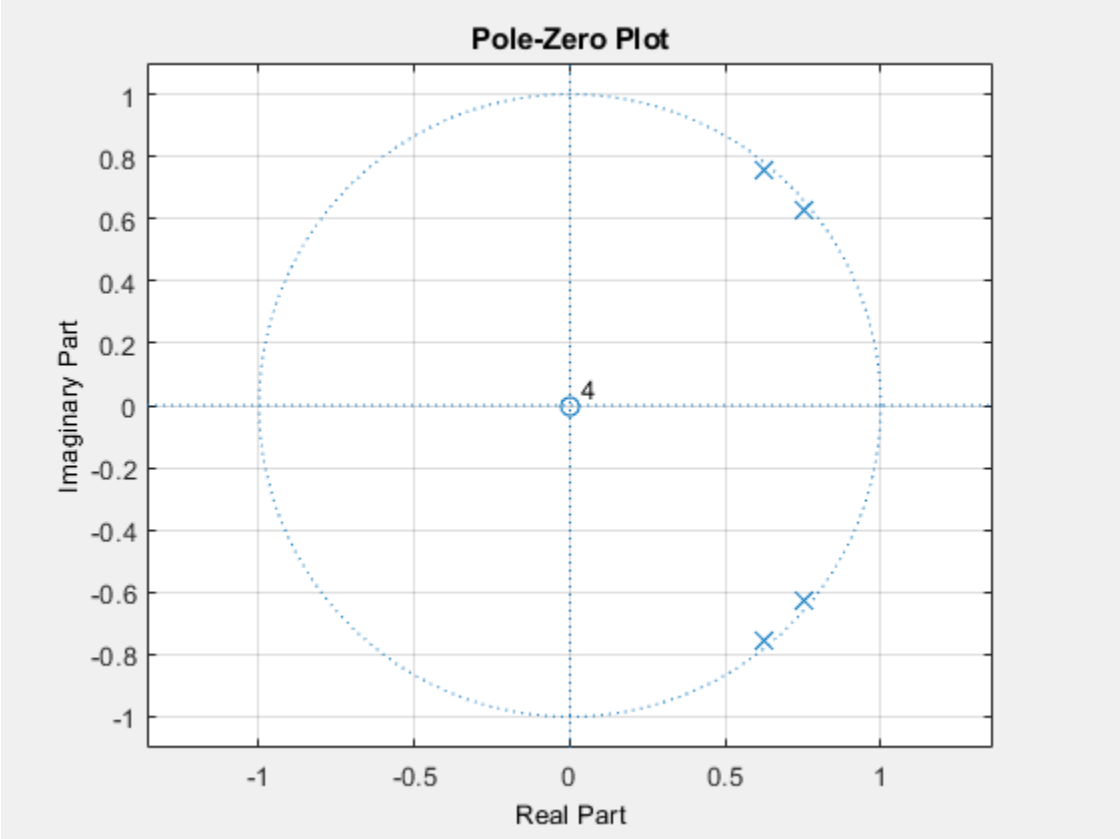
Use the following code to view the magnitude response of this IIR system.

```
B4 = 1;
A4 = [1 -2.7607 3.8106 -2.6535 0.9238];
fvtool(B4,A4)
```




Examining the placement of the poles, you can see this AR(4) process is also stable. The four poles are inside the unit circle.

```
fvtool(B4,A4,'Analysis','polezero')
```



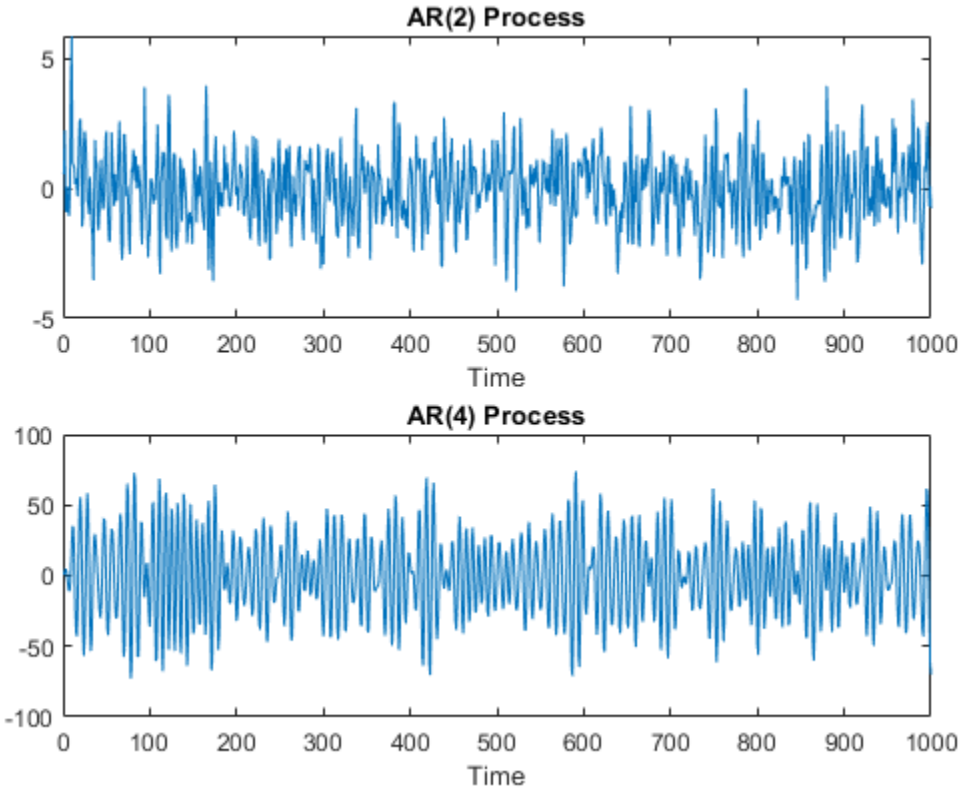

The dynamic range of this PSD is approximately 65 dB, much larger than the AR(2) model.

```
[H4,W4] = freqz(B4,A4,1e3,1);
dr4 = max(20*log10(abs(H4)))-min(20*log10(abs(H4)))
```

```
dr4 = 64.6213
```

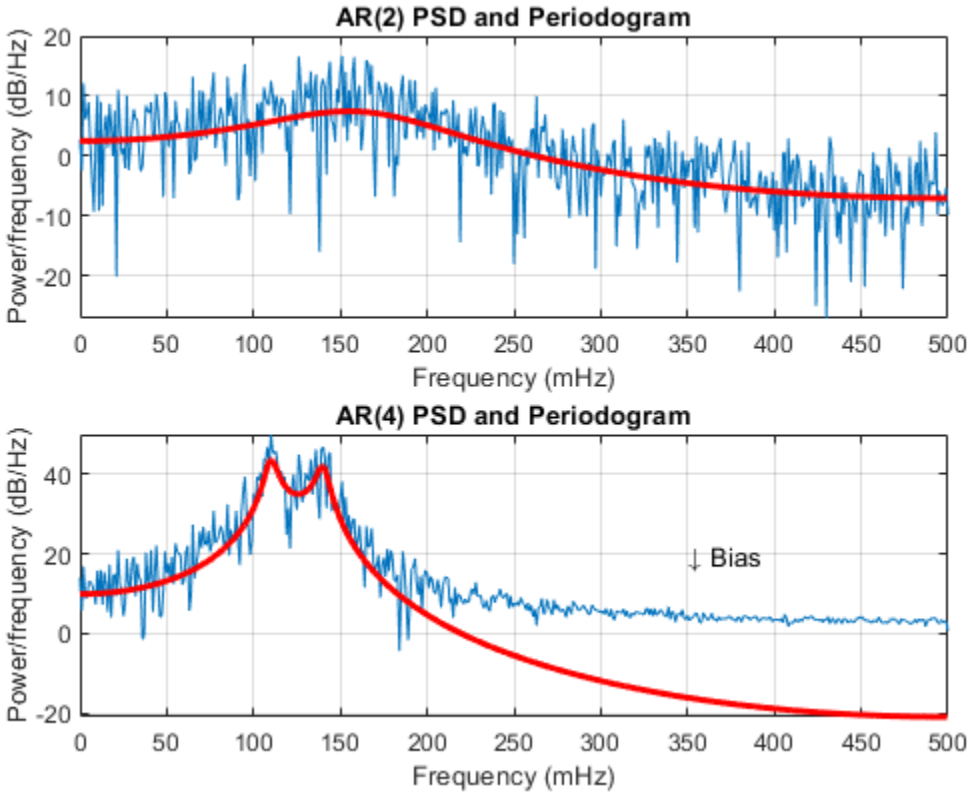
To simulate realizations from these AR(*p*) processes, use `randn` and `filter`. Set the random number generator to the default settings to produce repeatable results. Plot the realizations.

```
rng default
x = randn(1e3,1);
y2 = filter(B2,A2,x);
y4 = filter(B4,A4,x);

subplot(2,1,1)
plot(y2)
title('AR(2) Process')
xlabel('Time')

subplot(2,1,2)
plot(y4)
title('AR(4) Process')
xlabel('Time')
```




Compute and plot the periodograms of the AR(2) and AR(4) realizations. Compare the results against the true PSD. Note that `periodogram` converts the frequencies to millihertz for plotting.

```
Fs = 1;
NFFT = length(y2);

subplot(2,1,1)
periodogram(y2,rectwin(NFFT),NFFT,Fs)
hold on
plot(1000*W2,20*log10(abs(H2)),'r','linewidth',2)
title('AR(2) PSD and Periodogram')

subplot(2,1,2)
periodogram(y4,rectwin(NFFT),NFFT,Fs)
hold on
plot(1000*W4,20*log10(abs(H4)),'r','linewidth',2)
title('AR(4) PSD and Periodogram')
text(350,20,'\downarrow Bias')
```

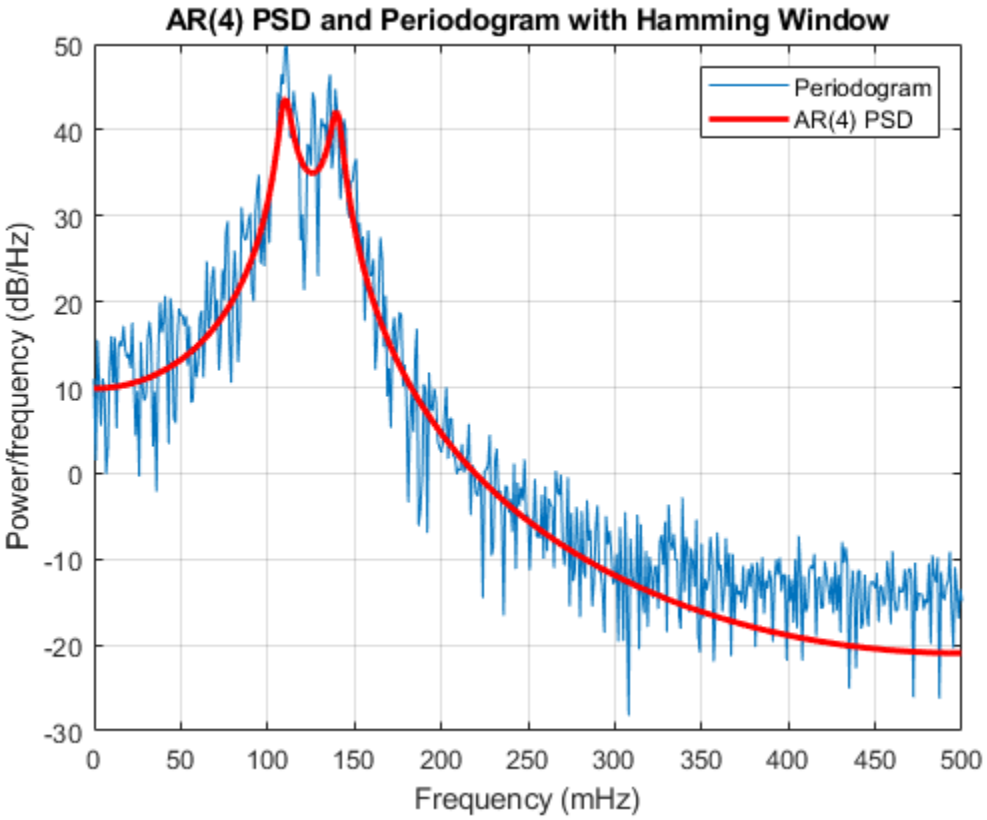



In the case of the AR(2) process, the periodogram estimate follows the shape of the true PSD but exhibits considerable variability. This is due to the low degrees of freedom. The pronounced negative deflections (in dB) in the periodogram are explained by taking the log of a chi-square random variable with two degrees of freedom.

In the case of the AR(4) process, the periodogram follows the shape of the true PSD at low frequencies but deviates from the PSD in the high frequencies. This is the effect of the convolution with Féjer's kernel. The large dynamic range of the AR(4) process compared to the AR(2) process is what makes the bias more pronounced.

Mitigate the bias demonstrated in the AR(4) process by using a taper, or window. In this example, use a Hamming window to taper the AR(4) realization before obtaining the periodogram.

```
figure
periodogram(y4,hamming(length(y4)),NFFT,Fs)
hold on
plot(1000*W4,20*log10(abs(H4)),'r','linewidth',2)
title('AR(4) PSD and Periodogram with Hamming Window')
legend('Periodogram','AR(4) PSD')
```

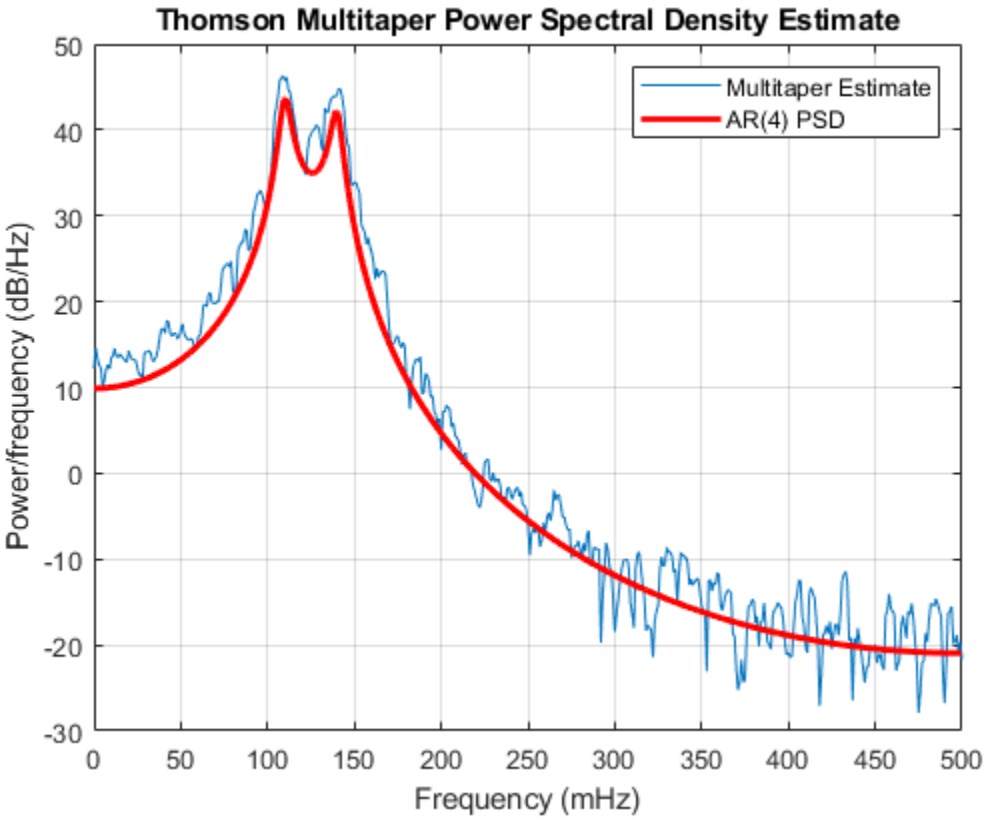



Note that the periodogram estimate now follows the true AR(4) PSD over the entire Nyquist frequency range. The periodogram estimates still only have two degrees of freedom so the use of a window does not reduce the variability of periodogram, but it does address bias.

In nonparametric spectral estimation, two methods for increasing the degrees of freedom and reducing the variability of the periodogram are Welch's overlapped segment averaging and multitaper spectral estimation.

Obtain a multitaper estimate of the AR(4) time series using a time half bandwidth product of 3.5. Plot the result.

```
NW = 3.5;

figure
pmtm(y4,NW,NFFT,Fs)
hold on
plot(1000*W4,20*log10(abs(H4)),'r','linewidth',2)
legend('Multitaper Estimate','AR(4) PSD')
```




The multitaper method produces a PSD estimate with significantly less variability than the periodogram. Because the multitaper method also uses windows, you see that the bias of the periodogram is also addressed.

## See Also

`periodogram` | `pmtm`

# Cross Spectrum and Magnitude-Squared Coherence

This example shows how to use the cross spectrum to obtain the phase lag between sinusoidal components in a bivariate time series. The example also uses the magnitude-squared coherence to identify significant frequency-domain correlation at the sine wave frequencies.

Create the bivariate time series. The individual series consist of two sine waves with frequencies of 100 and 200 Hz. The series are embedded in additive white Gaussian noise and sampled at 1 kHz. The sine waves in the *x*-series both have amplitudes equal to 1. The 100 Hz sine wave in the *y*-series has amplitude 0.5, and the 200 Hz sine wave in the *y*-series has amplitude 0.35. The 100 Hz and 200 Hz sine waves in the *y*-series are phase-lagged by $\pi/4$ radians and $\pi/2$ radians, respectively. You can think of the *y*-series as the noise-corrupted output of a linear system with input *x*. Set the random number generator to the default settings for reproducible results.

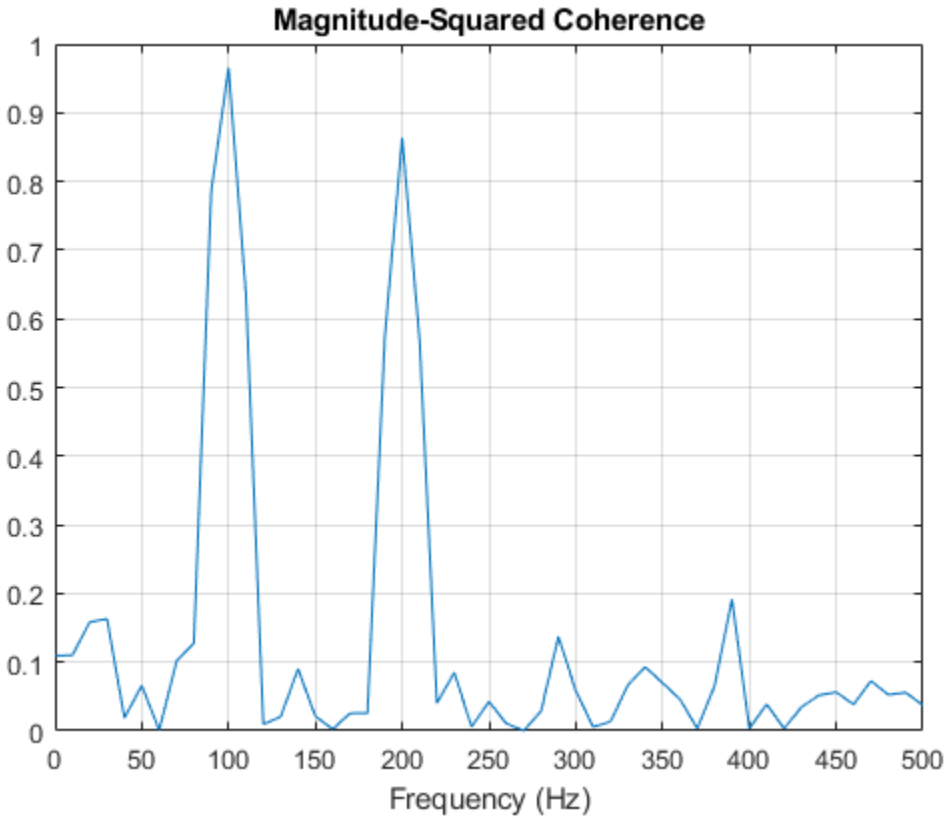
```
rng default

Fs = 1000;
t = 0:1/Fs:1-1/Fs;

x = cos(2*pi*100*t) + sin(2*pi*200*t) + 0.5*randn(size(t));
y = 0.5*cos(2*pi*100*t - pi/4) + 0.35*sin(2*pi*200*t - pi/2) + 0.5*randn(size(t));
```

Obtain the magnitude-squared coherence estimate for the bivariate time series. The magnitude-squared coherence enables you to identify significant frequency-domain correlation between the two time series. Phase estimates in the cross spectrum are only useful where significant frequency-domain correlation exists.

To prevent obtaining a magnitude-squared coherence estimate that is identically 1 for all frequencies, you must use an averaged coherence estimator. Both Welch's overlapped segment averaging (WOSA) and multitaper techniques are appropriate. `mscohere` implements a WOSA estimator.

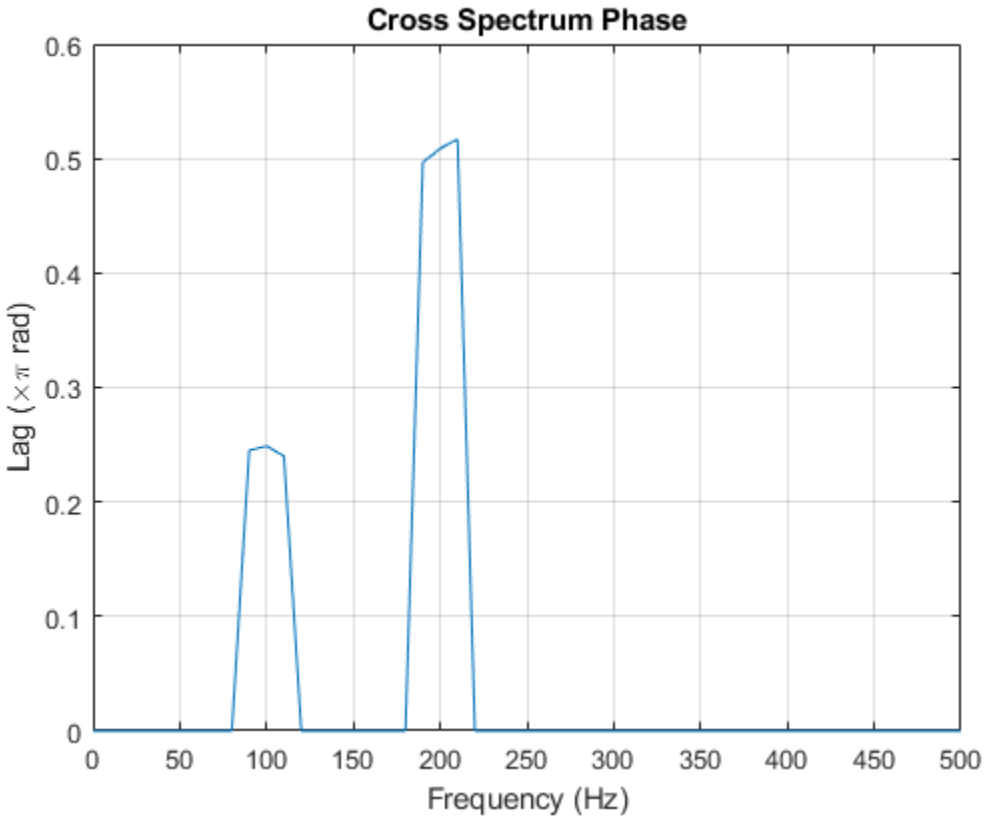
Set the window length to 100 samples. This window length contains 10 periods of the 100 Hz sine wave and 20 periods of the 200 Hz sine wave. Use an overlap of 80 samples with the default Hamming window. Input the sample rate explicitly to get the output frequencies in Hz. Plot the magnitude-squared coherence. The magnitude-squared coherence is greater than 0.8 at 100 and 200 Hz.

```
[Cxy,F] = mscohere(x,y,hamming(100),80,100,Fs);

plot(F,Cxy)
title('Magnitude-Squared Coherence')
xlabel('Frequency (Hz)')
grid
```




Obtain the cross spectrum of *x* and *y* using `cpsd`. Use the same parameters to obtain the cross spectrum that you used in the coherence estimate. Neglect the cross spectrum when the coherence is small. Plot the phase of the cross spectrum and indicate the frequencies with significant coherence between the two times. Mark the known phase lags between the sinusoidal components. At 100 Hz and 200 Hz, the phase lags estimated from the cross spectrum are close to the true values.

```
[Pxy,F] = cpsd(x,y,hamming(100),80,100,Fs);

Pxy(Cxy < 0.2) = 0;

plot(F,angle(Pxy)/pi)
title('Cross Spectrum Phase')
xlabel('Frequency (Hz)')
ylabel('Lag (\times\pi rad)')
grid
```




## See Also

`cpsd` | `mscohere` | `pwelch`

# Amplitude Estimation and Zero Padding

This example shows how to use zero padding to obtain an accurate estimate of the amplitude of a sinusoidal signal. Frequencies in the discrete Fourier transform (DFT) are spaced at intervals of $F_s/N$, where $F_s$ is the sample rate and $N$ is the length of the input time series. Attempting to estimate the amplitude of a sinusoid with a frequency that does not correspond to a DFT *bin* can result in an inaccurate estimate. Zero padding the data before computing the DFT often helps to improve the accuracy of amplitude estimates.

Create a signal consisting of two sine waves. The two sine waves have frequencies of 100 and 202.5 Hz. The sample rate is 1000 Hz and the signal is 1000 samples in length.

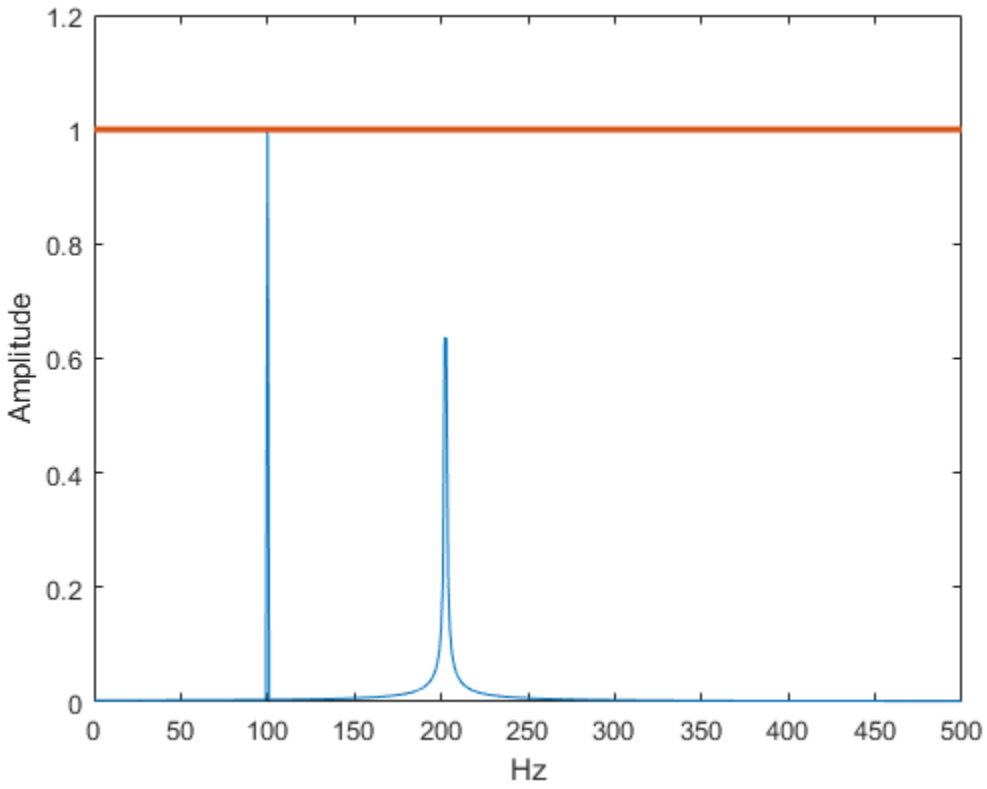
```
Fs = 1e3;
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)+sin(2*pi*202.5*t);
```

Obtain the DFT of the signal. The DFT bins are spaced at 1 Hz. Accordingly, the 100 Hz sine wave corresponds to a DFT bin, but the 202.5 Hz sine wave does not.

Because the signal is real-valued, use only the positive frequencies from the DFT to estimate the amplitude. Scale the DFT by the length of the input signal and multiply all frequencies except 0 and the Nyquist by 2.

Plot the result with the known amplitudes for comparison.

```
xdft = fft(x);
xdft = xdft(1:length(x)/2+1);
xdft = xdft/length(x);
xdft(2:end-1) = 2*xdft(2:end-1);
freq = 0:Fs/length(x):Fs/2;

plot(freq,abs(xdft))
hold on
plot(freq,ones(length(x)/2+1,1),'LineWidth',2)
xlabel('Hz')
ylabel('Amplitude')
hold off
```

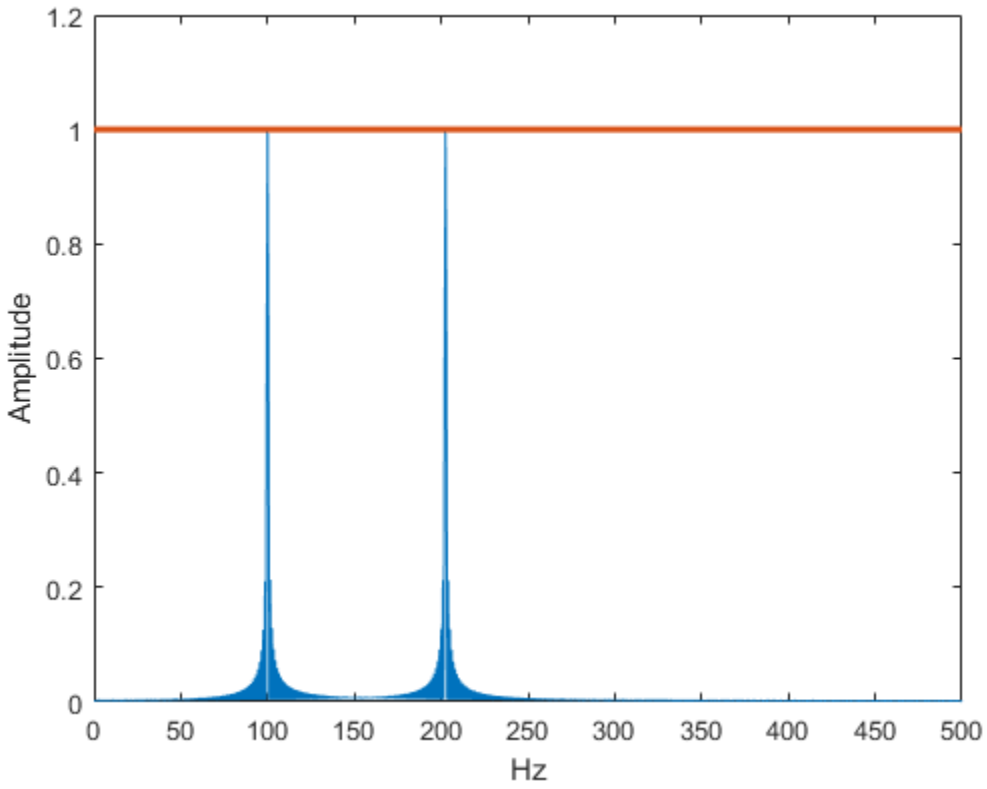



The amplitude estimate at 100 Hz is accurate because that frequency corresponds to a DFT bin. However, the amplitude estimate at 202.5 Hz is not accurate because that frequency does not correspond to a DFT bin.

You can interpolate the DFT by zero padding. Zero padding enables you to obtain more accurate amplitude estimates of resolvable signal components. On the other hand, zero padding does not improve the spectral (frequency) resolution of the DFT. The resolution is determined by the number of samples and the sample rate.

Pad the DFT out to 2000, or twice the original length of `x`. With this length, the spacing between DFT bins is $F_s/2000 = 0.5$ Hz. In this case, the energy from the 202.5 Hz sine wave falls directly in a DFT bin. Obtain the DFT and plot the amplitude estimates. Use zero padding out to 2000 samples.

```
lpad = 2*length(x);
xdft = fft(x,lpad);
xdft = xdft(1:lpad/2+1);
xdft = xdft/length(x);
xdft(2:end-1) = 2*xdft(2:end-1);
freq = 0:Fs/lpad:Fs/2;

plot(freq,abs(xdft))
hold on
plot(freq,ones(2*length(x)/2+1,1),'LineWidth',2)
xlabel('Hz')
ylabel('Amplitude')
hold off
```




The use of zero padding enables you to estimate the amplitudes of both frequencies correctly.

## See Also

`fft`

# Significance Testing for Periodic Component

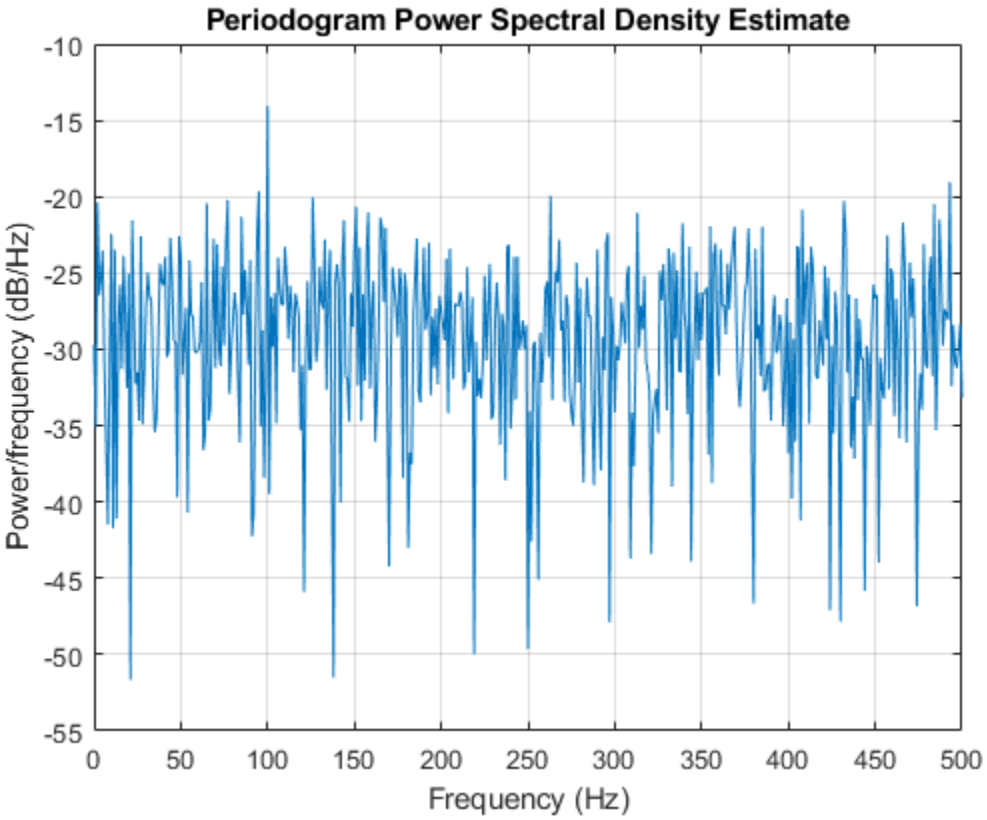
This example shows how to assess the significance of a sinusoidal component in white noise using Fisher's *g*-statistic. Fisher's *g*-statistic is the ratio of the largest periodogram value to the sum of all the periodogram values over 1/2 of the frequency interval, (0, `Fs`/2). A detailed description of the *g*-statistic and exact distribution can be found in the references.

Create a signal consisting of a 100 Hz sine wave in white Gaussian noise with zero mean and variance 1. The amplitude of the sine wave is 0.25. The sample rate is 1 kHz. Set the random number generator to the default settings for reproducible results.

```
rng default

Fs = 1e3;
t = 0:1/Fs:1-1/Fs;
x = 0.25*cos(2*pi*100*t)+randn(size(t));
```

Obtain the periodogram of the signal using `periodogram`. Exclude 0 and the Nyquist frequency (`Fs`/2). Plot the periodogram.

```
[Pxx,F] = periodogram(x,rectwin(length(x)),length(x),Fs);
Pxx = Pxx(2:length(x)/2);

periodogram(x,rectwin(length(x)),length(x),Fs)
```




Find the maximum value of the periodogram. Fisher's *g*-statistic is the ratio of the maximum periodogram value to the sum of all periodogram values.

```
[maxval,index] = max(Pxx);
fisher_g = Pxx(index)/sum(Pxx)
```

```
fisher_g = 0.0381
```

The maximum periodogram value occurs at 100 Hz, which you can verify by finding the frequency corresponding to the index of the maximum periodogram value.

```
F = F(2:end-1);
F(index)
```

```
ans = 100
```

Use the distributional results detailed in the references to determine the significance level, `pval`, of Fisher's *g*-statistic. The following MATLAB® code implements equation (6) of [2]. Use the logarithm of the gamma function to avoid overflows when computing binomial coefficients.

```
N = length(Pxx);
nn = 1:floor(1/fisher_g);

I = (-1).^(nn-1).*exp(gammaln(N+1)-gammaln(nn+1)-gammaln(N-nn+1)).*(1-nn*fisher_g).^(N-1);

pval = sum(I)
```

```
pval = 2.0163e-06
```

The *p*-value is less than 0.00001, which indicates a significant periodic component at 100 Hz. The interpretation of Fisher's *g*-statistic is complicated by the presence of other periodicities. See [1] for a modification when multiple periodicities may be present.

**References**

[1] Percival, Donald B. and Andrew T. Walden. *Spectral Analysis for Physical Applications*. Cambridge, UK: Cambridge University Press, 1993.

[2] Wichert, Sofia, Konstantinos Fokianos, and Korbinian Strimmer. "Identifying Periodically Expressed Transcripts in Microarray Time Series Data." *Bioinformatics*. Vol. 20, 2004, pp. 5-20.

## See Also

`nchoosek` | `periodogram`

# Frequency Estimation by Subspace Methods

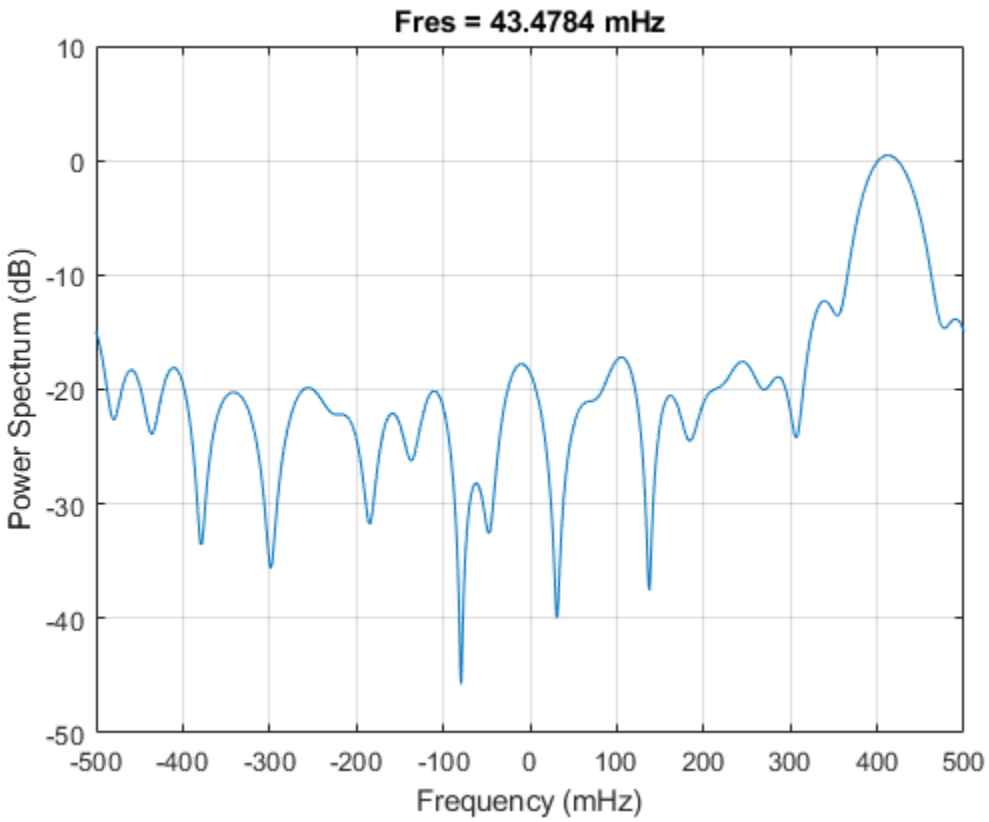
This example shows how to resolve closely spaced sine waves using subspace methods. Subspace methods assume a harmonic model consisting of a sum of sine waves, possibly complex, in additive noise. In a complex-valued harmonic model, the noise is also complex-valued.

Create a complex-valued signal 24 samples in length. The signal consists of two complex exponentials (sine waves) with frequencies of 0.4 Hz and 0.425 Hz and additive complex white Gaussian noise. The noise has zero mean and variance $0.2^2$. In complex white noise, both the real and imaginary parts have variance equal to one-half the overall variance.

```
n = 0:23;
x = exp(1j*2*pi*0.4*n) + exp(1j*2*pi*0.425*n)+ ...
    0.2/sqrt(2)*(randn(size(n))+1j*randn(size(n)));
```

Attempt to resolve the two sine waves using the power spectrum of the signal. Set the leakage to the maximum value for best results.

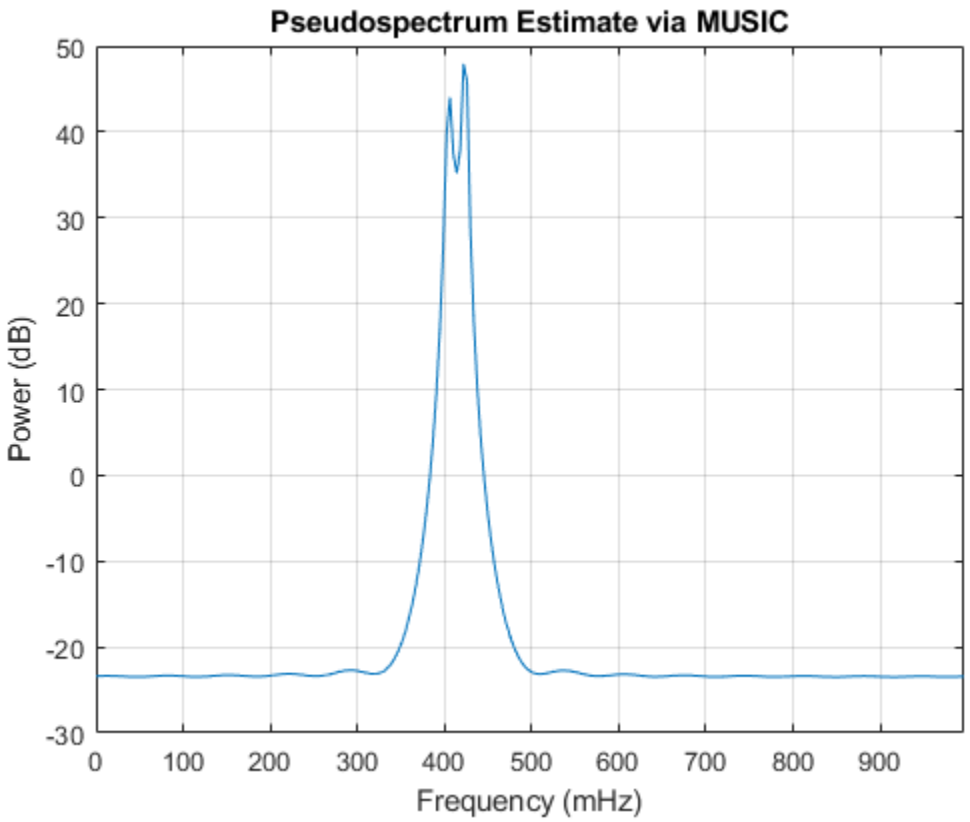
```
pspectrum(x,n,'Leakage',1)
```




The periodogram shows a broad peak near 0.4 Hz. You cannot resolve the two separate sine waves because the frequency resolution of the periodogram is 1/*N*, where *N* is the length of the signal. In this case, 1/*N* is greater than the separation of the two sine waves. Zero padding does not help to resolve two separate peaks.

Use a subspace method to resolve the two closely spaced peaks. In this example, use the MUSIC method. Estimate the autocorrelation matrix and input the autocorrelation matrix into `pmusic`. Specify a model with two sinusoidal components. Plot the result.

```
[X,R] = corrmtx(x,14,'mod');

pmusic(R,2,[],1,'corr')
```




The MUSIC method is able to separate the two peaks at 0.4 Hz and 0.425 Hz. However, subspace methods do not produce power estimates like power spectral density estimates. Subspace methods are most useful for frequency identification and can be sensitive to model-order misspecification.

## See Also

`corrmtx` | `pmusic` | `pspectrum`

# Frequency-Domain Linear Regression

This example shows how to use the discrete Fourier transform to construct a linear regression model for a time series. The time series used in this example is the monthly number of accidental deaths in the United States from 1973 to 1979. The data are published in Brockwell and Davis (2006). The original source is the U. S. National Safety Council.

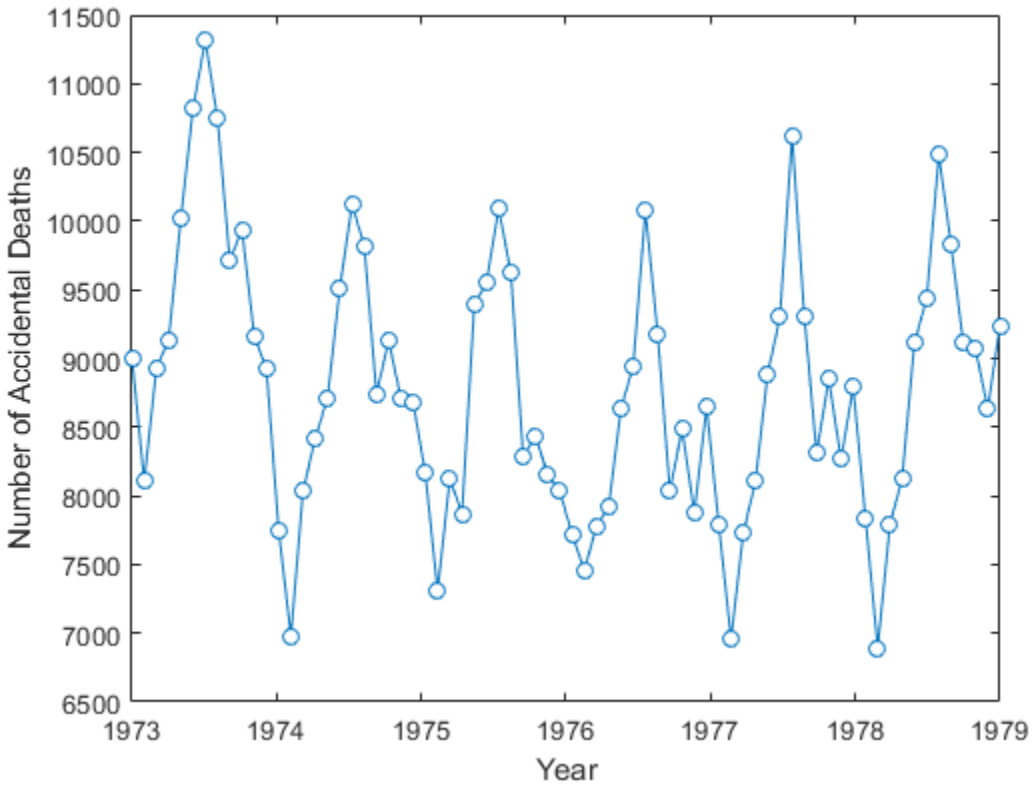
Enter the data. Copy the `exdata` matrix into the MATLAB® workspace.

```
exdata = [
        9007        7750        8162        7717        7792        7836
        8106        6981        7306        7461        6957        6892
        8928        8038        8124        7776        7726        7791
        9137        8422        7870        7925        8106        8129
       10017        8714        9387        8634        8890        9115
       10826        9512        9556        8945        9299        9434
       11317       10120       10093       10078       10625       10484
       10744        9823        9620        9179        9302        9827
        9713        8743        8285        8037        8314        9110
        9938        9129        8433        8488        8850        9070
        9161        8710        8160        7874        8265        8633
        8927        8680        8034        8647        8796        9240];
```

`exdata` is a 12-by-6 matrix. Each column of `exdata` contains 12 months of data. The first row of each column contains the number of U.S. accidental deaths for January of the corresponding year. The last row of each column contains the number of U.S. accidental deaths for December of the corresponding year.

Reshape the data matrix into a 72-by-1 time series and plot the data for the years 1973 to 1978.

```
ts = reshape(exdata,72,1);
years = linspace(1973,1979,72);

plot(years,ts,'o-','MarkerFaceColor','auto')
xlabel('Year')
ylabel('Number of Accidental Deaths')
```




A visual inspection of the data indicates that number of accidental deaths varies in a periodic manner. The period of the oscillation appears to be roughly 1 year (12 months). The periodic nature of the data suggests that an appropriate model may be

$$X(n) = \mu + \sum_k \left( A_k \cos \frac{2\pi k n}{N} + B_k \cos \frac{2\pi k n}{N} \right) + \varepsilon(n),$$

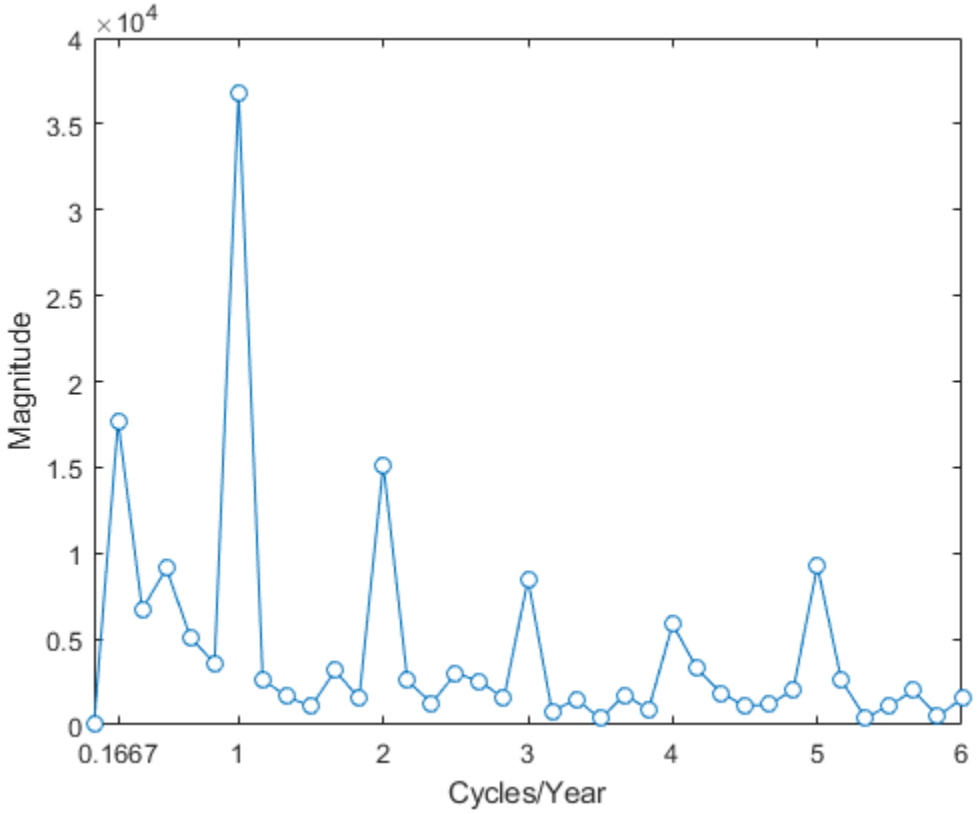
where $\mu$ is the overall mean, $N$ is the length of the time series, and $\varepsilon(n)$ is a white noise sequence of independent and identically-distributed (IID) Gaussian random variables with zero mean and some variance. The additive noise term accounts for the randomness inherent in the data. The parameters of the model are the overall mean and the amplitudes of the cosines and sines. The model is linear in the parameters.

To construct a linear regression model in the time domain, you have to specify which frequencies to use for the cosines and sines, form the design matrix, and solve the normal equations in order to obtain the least-squares estimates of the model parameters. In this case, it is easier to use the discrete Fourier transform to detect the periodicities, retain only a subset of the Fourier coefficients, and invert the transform to obtain the fitted time series.

Perform a spectral analysis of the data to reveal which frequencies contribute significantly to the variability in the data. Because the overall mean of the signal is approximately 9,000 and is proportional to the Fourier transform at 0 frequency, subtract the mean prior to the spectral analysis. This reduces the large magnitude Fourier coefficient at 0 frequency and makes any significant oscillations easier to detect. The frequencies in the Fourier transform are spaced at an interval that is the reciprocal of the time series length, 1/72. Sampling the data monthly, the highest frequency in the

spectral analysis is 1 cycle/2 months. In this case, it is convenient to look at the spectral analysis in terms of cycles/year so scale the frequencies accordingly for visualization.

```
tsdft = fft(ts-mean(ts));
freq = 0:1/72:1/2;

plot(freq.*12,abs(tsdft(1:length(ts)/2+1)),'o-', ...
    'MarkerFaceColor','auto')
xlabel('Cycles/Year')
ylabel('Magnitude')
ax = gca;
ax.XTick = [1/6 1 2 3 4 5 6];
```




Based on the magnitudes, the frequency of 1 cycle/12 months is the most significant oscillation in the data. The magnitude at 1 cycle/12 months is more than twice as large as any other magnitude. However, the spectral analysis reveals that there are also other periodic components in the data. For example, there appears to be periodic components at harmonics (integer multiples) of 1 cycle/12 months. There also appears to be a periodic component with a period of 1 cycle/72 months.

Based on the spectral analysis of the data, fit a simple linear regression model using a cosine and sine term with a frequency of the most significant component: 1 cycle/year (1 cycle/12 months).

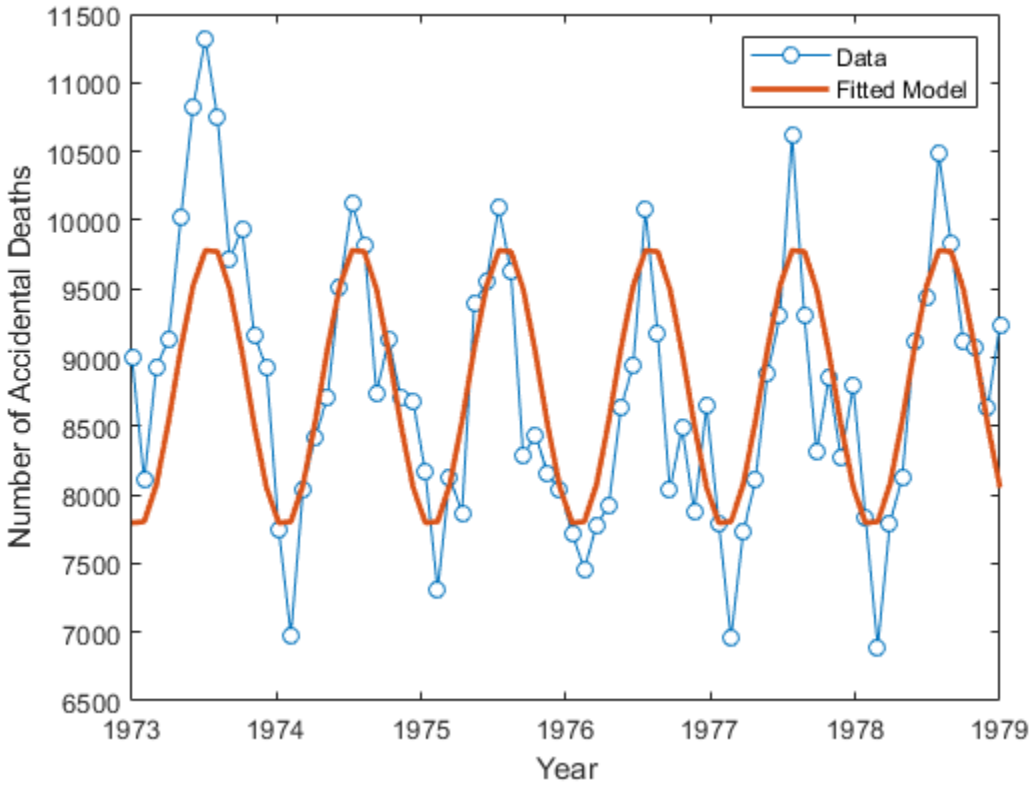
Determine the frequency bin in the discrete Fourier transform that corresponds to 1 cycle/12 months. Because the frequencies are spaced at 1/72 and the first bin corresponds to 0 frequency, the correct bin is 72/12+1. This is the frequency bin of the *positive* frequency. You must also include the frequency bin corresponding to the *negative* frequency: -1 cycle/12 months. With MATLAB indexing, the frequency bin of the negative frequency is 72-72/12+1.

Create a 72-by-1 vector of zeros. Fill the appropriate elements of the vector with the Fourier coefficients corresponding to a positive and negative frequency of 1 cycle/12 months. Invert the Fourier transform and add the overall mean to obtain a fit to the accidental death data.

```
freqbin = 72/12;
freqbins = [freqbin 72-freqbin]+1;
tsfit = zeros(72,1);
tsfit(freqbins) = tsdft(freqbins);
tsfit = ifft(tsfit);
mu = mean(ts);
tsfit = mu+tsfit;
```

Plot the original data along with the fitted series using two Fourier coefficients.

```
plot(years,ts,'o-','MarkerFaceColor','auto')
xlabel('Year')
ylabel('Number of Accidental Deaths')
hold on
plot(years,tsfit,'linewidth',2)
legend('Data','Fitted Model')
hold off
```

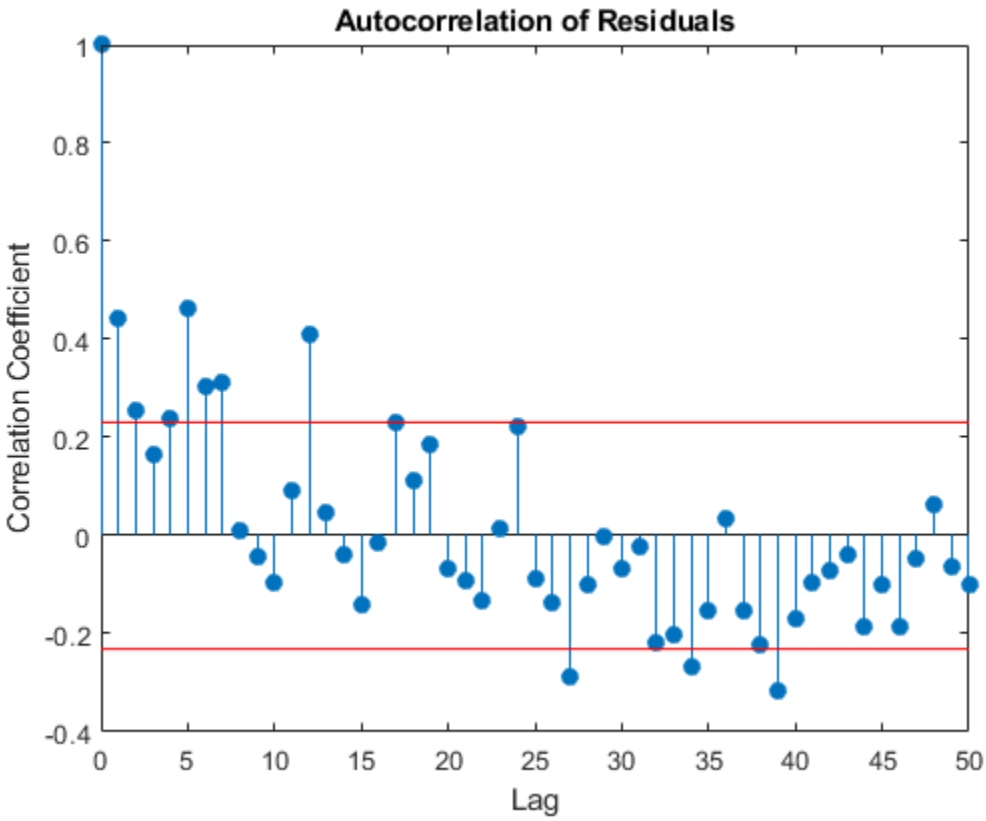



The fitted model appears to capture the general periodic nature of the data and supports the initial conclusion that data oscillate with a cycle of 1 year.

To assess how adequately the single frequency of 1 cycle/12 months accounts for the observed time series, form the residuals. If the residuals resemble a white noise sequence, the simple linear model with one frequency has adequately modeled the time series.

To assess the residuals, use the autocorrelation sequence with 95%-confidence intervals for a white noise.

```
resid = ts-tsfit;
[xc,lags] = xcorr(resid,50,'coeff');

stem(lags(51:end),xc(51:end),'filled')
hold on
lconf = -1.96*ones(51,1)/sqrt(72);
uconf = 1.96*ones(51,1)/sqrt(72);
plot(lags(51:end),lconf,'r')
plot(lags(51:end),uconf,'r')
xlabel('Lag')
ylabel('Correlation Coefficient')
title('Autocorrelation of Residuals')
hold off
```




The autocorrelation values fall outside the 95% confidence bounds at a number of lags. It does not appear that the residuals are white noise. The conclusion is that the simple linear model with one sinusoidal component does not account for all the oscillations in the number of accidental deaths. This is expected because the spectral analysis revealed additional periodic components in addition to the dominant oscillation. Creating a model that incorporates additional periodic terms indicated by the spectral analysis will improve the fit and whiten the residuals.

Fit a model which consists of the three largest Fourier coefficient magnitudes. Because you have to retain the Fourier coefficients corresponding to both negative and positive frequencies, retain the largest 6 indices.

```
tsfit2dft = zeros(72,1);
[Y,I] = sort(abs(tsdft),'descend');
indices = I(1:6);
tsfit2dft(indices) = tsdft(indices);
```

Demonstrate that preserving only 6 of the 72 Fourier coefficients (3 frequencies) retains most of the signal's energy. First, demonstrate that retaining all the Fourier coefficients yields energy equivalence between the original signal and the Fourier transform.

```
norm(1/sqrt(72)*tsdft,2)/norm(ts-mean(ts),2)
```

```
ans = 1.0000
```

The ratio is 1. Now, examine the energy ratio where only 3 frequencies are retained.

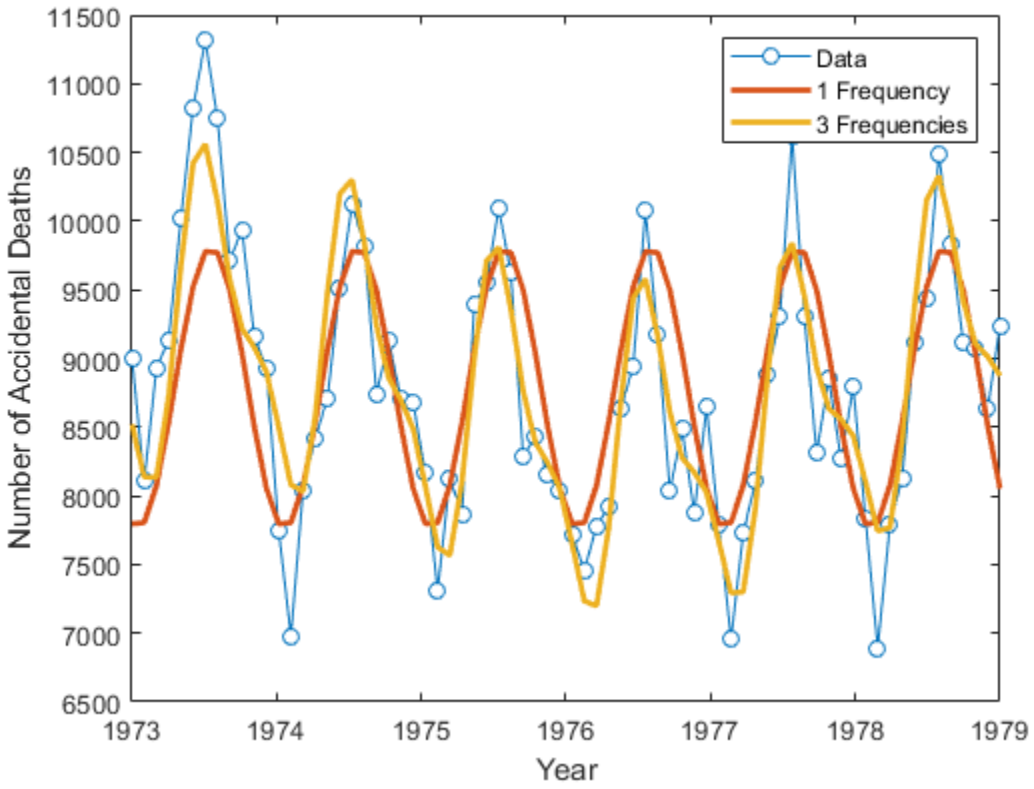
```
norm(1/sqrt(72)*tsfit2dft,2)/norm(ts-mean(ts),2)
```

```
ans = 0.8991
```

Almost 90% of the energy is retained. Equivalently, 90% of the variance of the time series is accounted for by 3 frequency components.

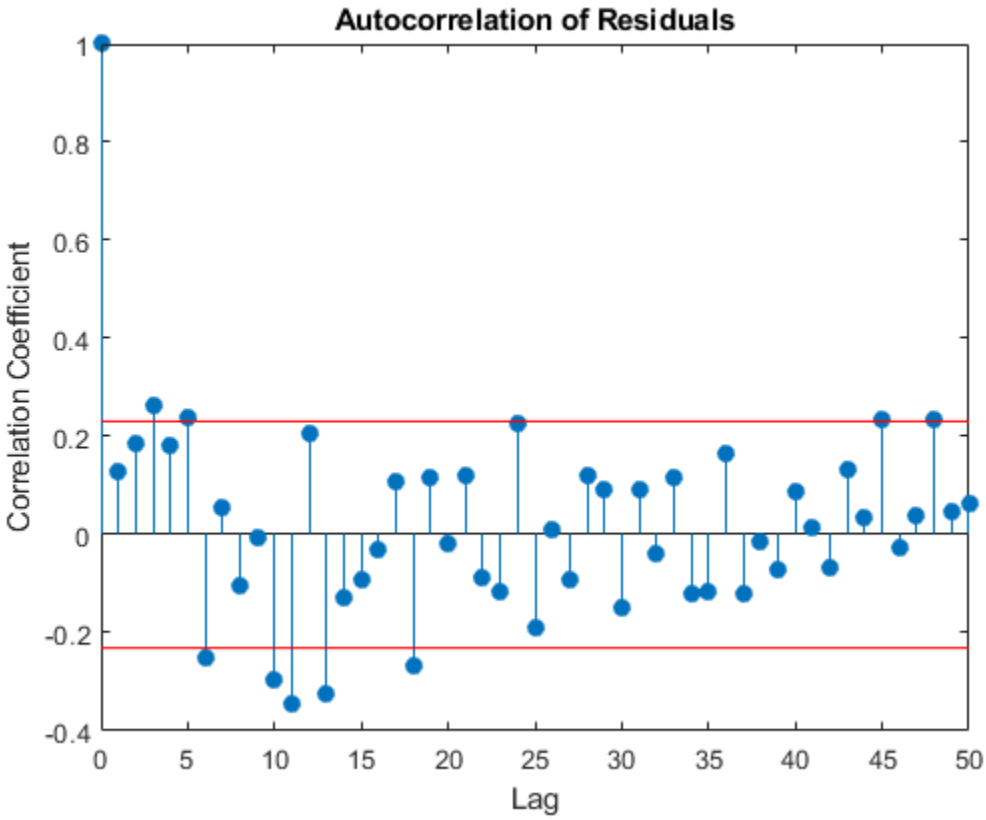
Form an estimate of the data based on 3 frequency components. Compare the original data, the model with one frequency, and the model with 3 frequencies.

```
tsfit2 = mu+ifft(tsfit2dft,'symmetric');

plot(years,ts,'o-','markerfacecolor','auto')
xlabel('Year')
ylabel('Number of Accidental Deaths')
hold on
plot(years,tsfit,'linewidth',2)
plot(years,tsfit2,'linewidth',2)
legend('Data','1 Frequency','3 Frequencies')
hold off
```




Using 3 frequencies has improved the fit to the original signal. You can see this by examining the autocorrelation of the residuals from the 3-frequency model.

```
resid = ts-tsfit2;
[xc,lags] = xcorr(resid,50,'coeff');

stem(lags(51:end),xc(51:end),'filled')
hold on
lconf = -1.96*ones(51,1)/sqrt(72);
uconf = 1.96*ones(51,1)/sqrt(72);
plot(lags(51:end),lconf,'r')
plot(lags(51:end),uconf,'r')
xlabel('Lag')
ylabel('Correlation Coefficient')
title('Autocorrelation of Residuals')
hold off
```




Using 3 frequencies has resulted in residuals that more closely approximate a white noise process.

Demonstrate that the parameter values obtained from the Fourier transform are equivalent to a time-domain linear regression model. Find the least-squares estimates for the overall mean, the cosine amplitudes, and the sine amplitudes for the three frequencies by forming the design matrix and solving the normal equations. Compare the fitted time series with that obtained from the Fourier transform.

```
X = ones(72,7);
X(:,2) = cos(2*pi/72*(0:71))';
X(:,3) = sin(2*pi/72*(0:71))';
X(:,4) = cos(2*pi*6/72*(0:71))';
X(:,5) = sin(2*pi*6/72*(0:71))';
X(:,6) = cos(2*pi*12/72*(0:71))';
X(:,7) = sin(2*pi*12/72*(0:71))';
beta = X\ts;
tsfit_lm = X*beta;
max(abs(tsfit_lm-tsfit2))
```

```
ans = 7.2760e-12
```

The two methods yield identical results. The maximum absolute value of the difference between the two waveforms is on the order of 10-12. In this case, the frequency-domain approach was easier than the equivalent time-domain approach. You naturally use a spectral analysis to visually inspect which oscillations are present in the data. From that step, it is simple to use the Fourier coefficients to construct a model for the signal consisting of a sum cosines and sines.

For more details on spectral analysis in time series and the equivalence with time-domain regression see (Shumway and Stoffer, 2006).

While spectral analysis can answer which periodic components contribute significantly to the variability of the data, it does not explain why those components are present. If you examine these data closely, you see that the minimum values in the 12-month cycle tend to occur in February, while the maximum values occur in July. A plausible explanation for these data is that people are naturally more active in summer than in the winter. Unfortunately, as a result of this increased activity, there is an increased probability of the occurrence of fatal accidents.

References

Brockwell, Peter J., and Richard A. Davis. *Time Series: Theory and Methods*. New York: Springer, 2006.

Shumway, Robert H., and David S. Stoffer. *Time Series Analysis and Its Applications with R Examples*. New York: Springer, 2006.

## See Also

`fft` | `ifft` | `xcorr`

# Measure Total Harmonic Distortion

This example shows how to measure the total harmonic distortion (THD) of a sinusoidal signal. The example uses the following scenario: A manufacturer of audio speakers claims the model A speaker produces less than 0.09% harmonic distortion at 1 kHz with a 1 volt input. The harmonic distortion is measured with respect to the fundamental (THD-F).

Assume you record the following data obtained by driving the speaker with a 1 kHz tone at 1 volt. The data is sampled at 44.1 kHz for analysis.

```
Fs = 44.1e3;
t = 0:1/Fs:1;
x = cos(2*pi*1000*t)+8e-4*sin(2*pi*2000*t)+2e-5*cos(2*pi*3000*t-pi/4)+...
    8e-6*sin(2*pi*4000*t);
```

Obtain the total harmonic distortion of the input signal in dB. Specify that six harmonics are used in calculating the THD. This includes the fundamental frequency of 1 kHz. Input the sampling frequency of 44.1 kHz. Determine the frequencies of the harmonics and their power estimates.

```
nharm = 6;
[thd_db,harmpow,harmfreq] = thd(x,Fs,nharm);
```

The function `thd` outputs the total harmonic distortion in dB. Convert the measurement from dB to a percentage to compare the value against the manufacturer's claims.

```
percent_thd = 100*(10^(thd_db/20))
```

```
percent_thd = 0.0800
```

The value you obtain indicates that the manufacturer's claims about the THD for speaker model A are correct.

You can obtain further insight by examining the power (dB) of the individual harmonics.

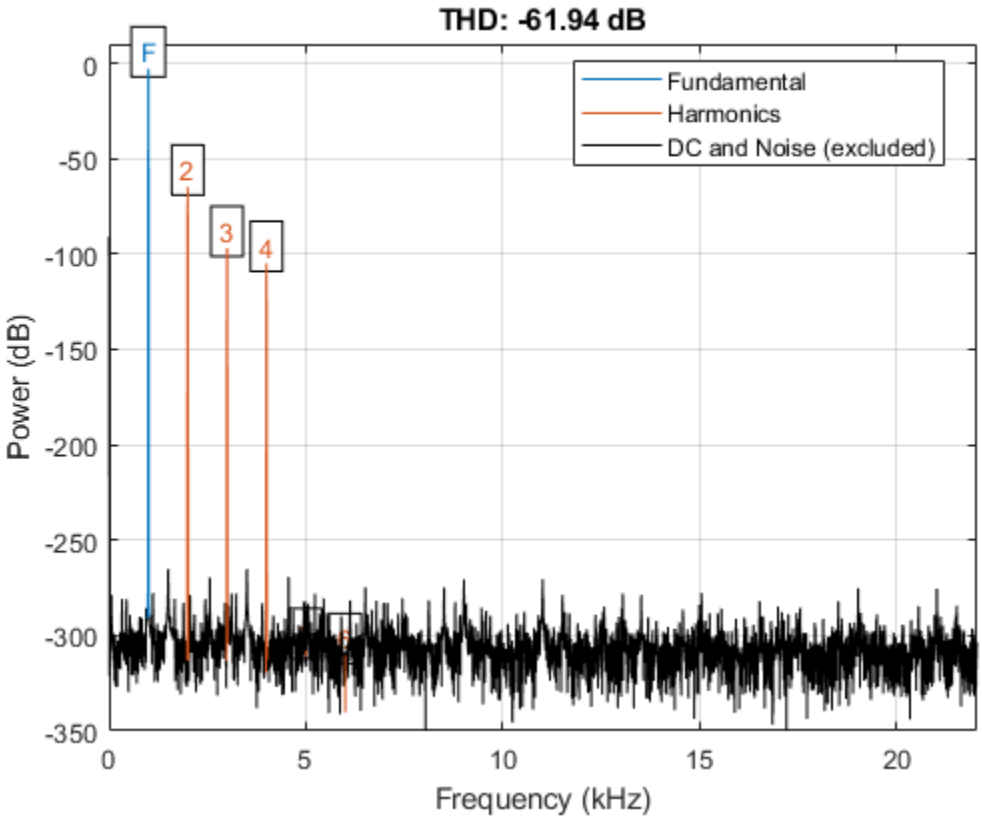
```
T = table(harmfreq,harmpow,'VariableNames',{'Frequency','Power'})
```

```
T=6×2 table
    Frequency     Power
    _________    _______

       1000      -3.0103
       2000      -64.949
       3000       -96.99
       4000      -104.95
     4997.9      -306.09
     5998.9      -310.51
```

The total harmonic distortion is approximately −62 dB. If you examine the power of the individual harmonics, you see that the major contribution comes from the harmonic at 2 kHz. The power at 2 kHz is approximately 62 dB below the power of the fundamental. The remaining harmonics do not contribute significantly to the total harmonic distortion. Additionally, the synthesized signal contains only four harmonics, including the fundamental. This is confirmed by the table, which shows a large power reduction after 4 kHz. Therefore, repeating the calculation with only four harmonics does not change the total harmonic distortion significantly.

Plot the signal spectrum, display the total harmonic distortion on the figure title, and annotate the harmonics.

```
thd(x,Fs,nharm);
```




## See Also

`thd`

## Related Examples

- “Analyzing Harmonic Distortion” on page 24-91

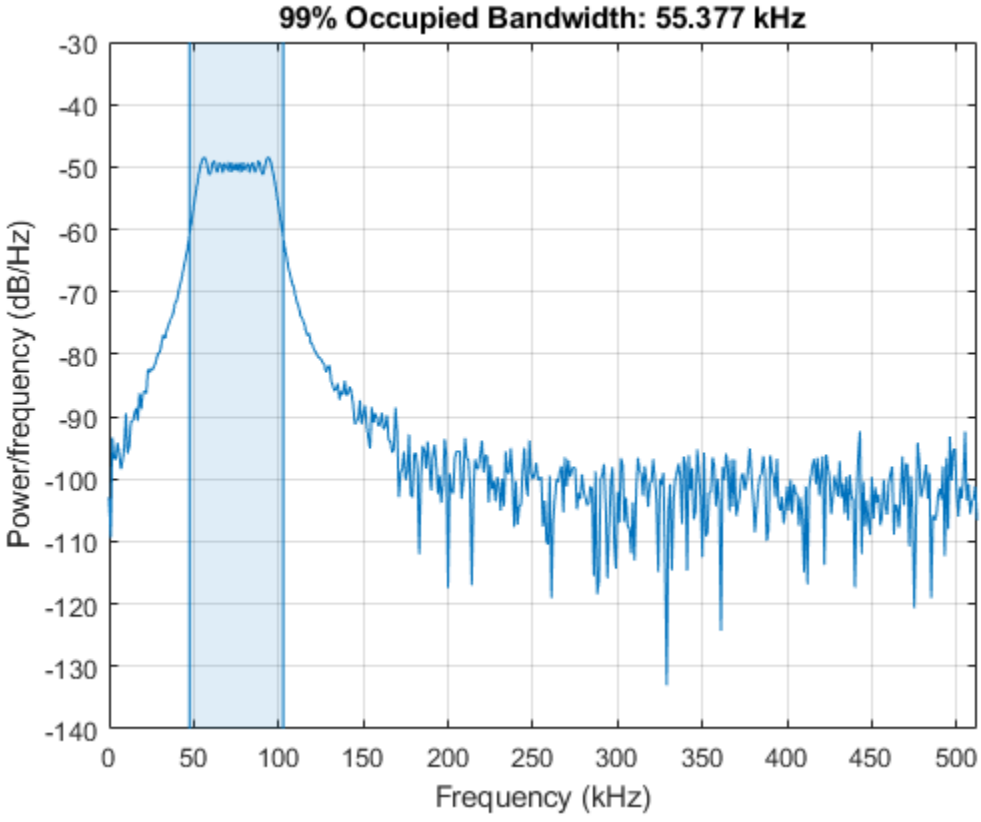
# Measure Mean Frequency, Power, Bandwidth

Generate 1024 samples of a chirp sampled at 1024 kHz. The chirp has an initial frequency of 50 kHz and reaches 100 kHz at the end of the sampling. Add white Gaussian noise such that the signal-to-noise ratio is 40 dB.

```
nSamp = 1024;
Fs = 1024e3;
SNR = 40;

t = (0:nSamp-1)'/Fs;

x = chirp(t,50e3,nSamp/Fs,100e3);
x = x+randn(size(x))*std(x)/db2mag(SNR);
```

Estimate the 99% occupied bandwidth of the signal and annotate it on a plot of the power spectral density (PSD).

```
obw(x,Fs);
```




Compute the power in the band and verify that it is 99% of the total.

```
[bw,flo,fhi,powr] = obw(x,Fs);

pcent = powr/bandpower(x)*100
```

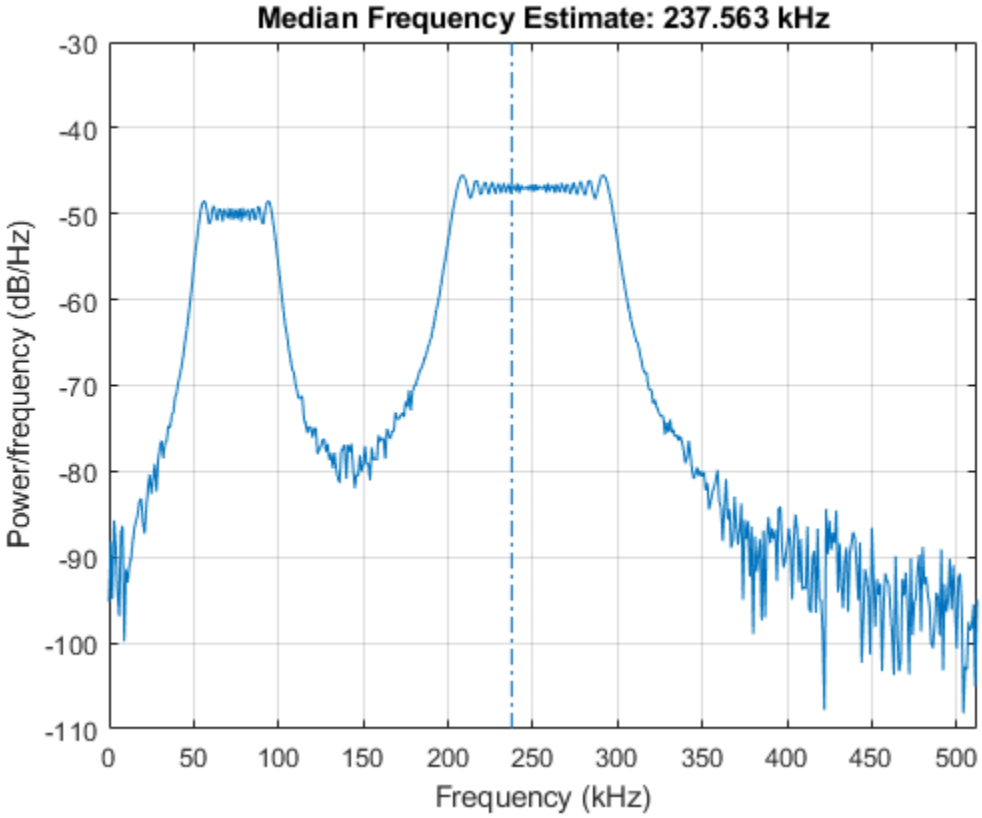
```
pcent = 99.0000
```

Generate another chirp. Specify an initial frequency of 200 kHz, a final frequency of 300 kHz, and an amplitude that is twice that of the first signal. Add white Gaussian noise.

```
x2 = 2*chirp(t,200e3,nSamp/Fs,300e3);
x2 = x2+randn(size(x2))*std(x2)/db2mag(SNR);
```

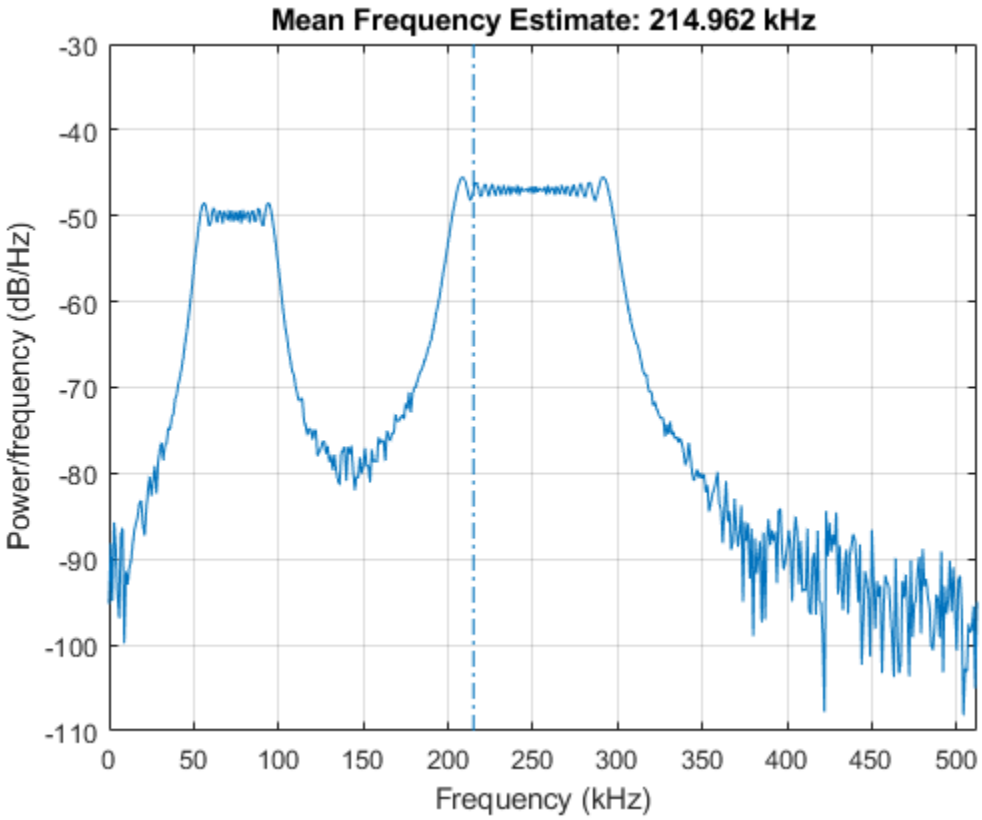
Add the two chirps to form a new signal. Plot the PSD of the signal and annotate its median frequency.

```
medfreq([x+x2],Fs);
```



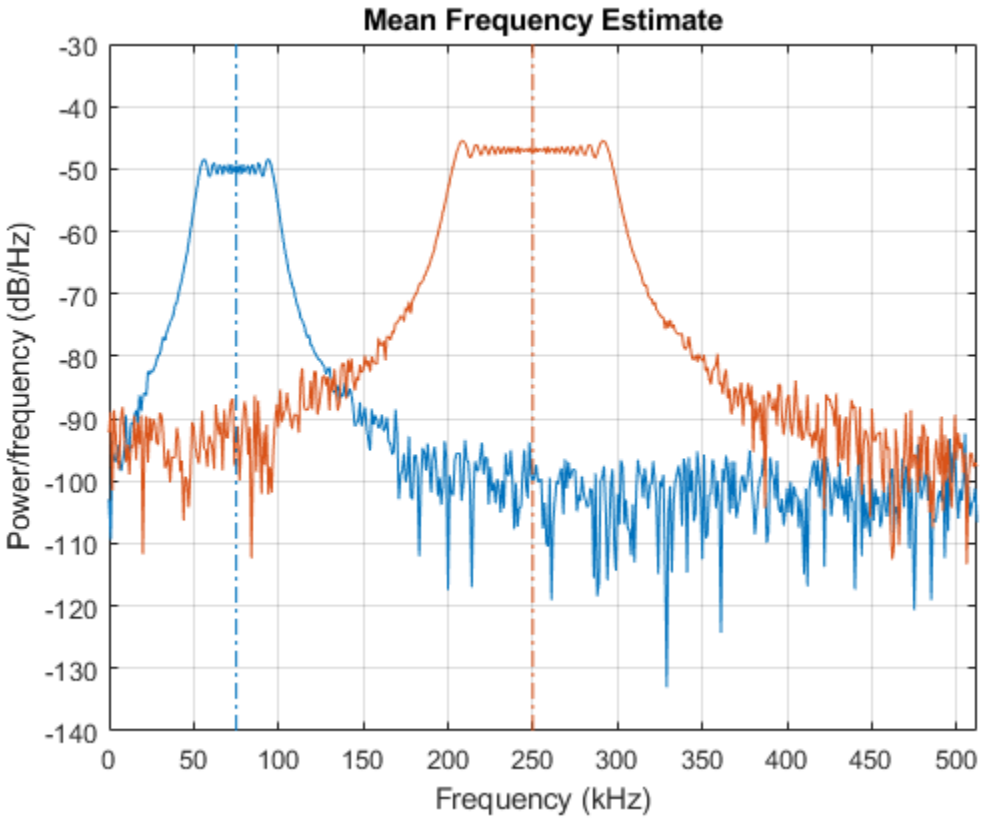

Plot the PSD and annotate the mean frequency.

```
meanfreq([x+x2],Fs);
```




Now consider each chirp to represent a separate channel. Estimate the mean frequency of each channel. Annotate the mean frequencies on a plot of the PSDs.

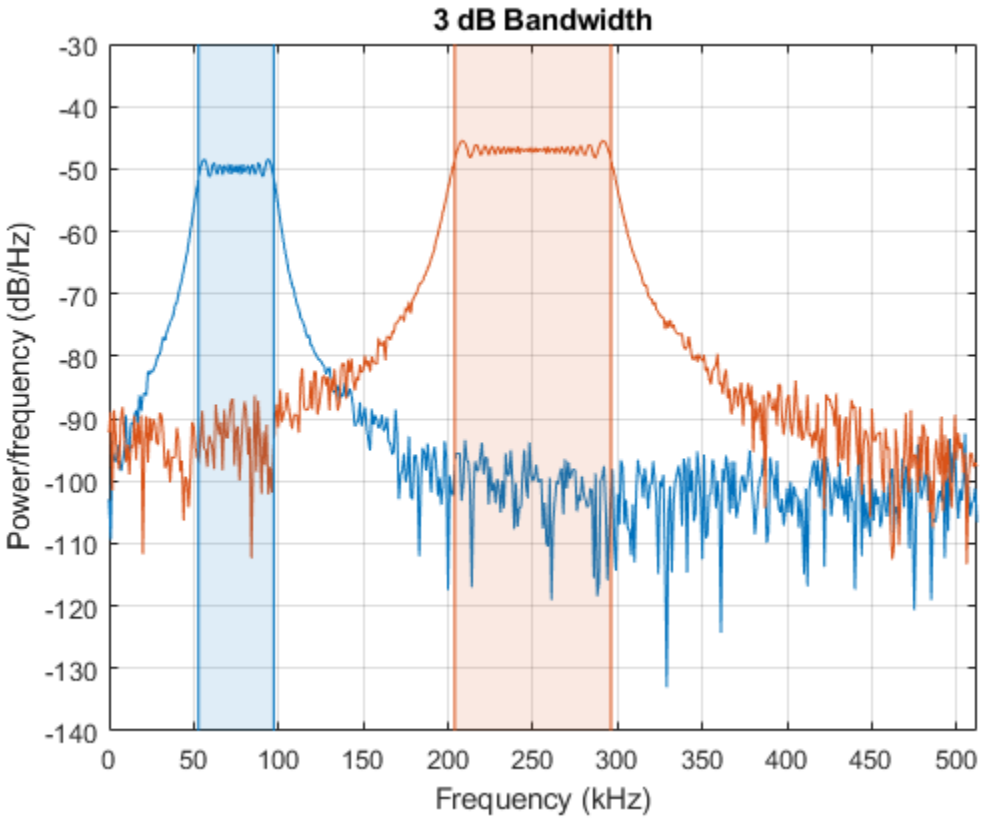
```
meanfreq([x x2],Fs)
```




```
ans = 1×2
10^5 ×

    0.7503    2.4999
```

Estimate the half-power bandwidth of each channel. Annotate the 3-dB bandwidths on a plot of the PSDs.

```
powerbw([x x2],Fs)
```




```
ans = 1×2
10^4 ×

    4.4386    9.2208
```

## See Also

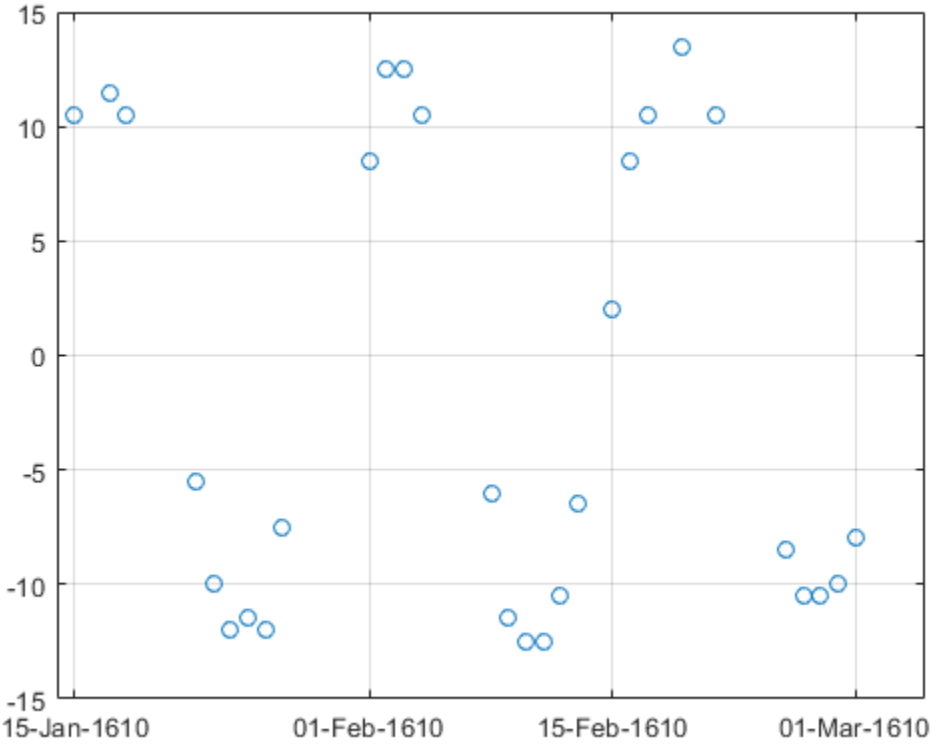
`bandpower` | `meanfreq` | `medfreq` | `obw` | `powerbw`

# Periodogram of Data Set with Missing Samples

Galileo Galilei observed the motion of Jupiter's four largest satellites during the winter of 1610. When the weather allowed, Galileo recorded the satellites' locations. Use his observations to estimate the orbital period of one of the satellites, Callisto.

Callisto's angular position is measured in minutes of arc. Missing data due to cloudy conditions are specified using NaNs. The first observation is dated January 15. Generate a `datetime` array of observation times.

```
yg = [10.5 NaN 11.5 10.5 NaN NaN NaN -5.5 -10.0 -12.0 -11.5 -12.0 -7.5 ...
    NaN NaN NaN NaN 8.5 12.5 12.5 10.5 NaN NaN NaN -6.0 -11.5 -12.5 ...
    -12.5 -10.5 -6.5 NaN 2.0 8.5 10.5 NaN 13.5 NaN 10.5 NaN NaN NaN ...
    -8.5 -10.5 -10.5 -10.0 -8.0]';

obsv = datetime(1610,1,14+(1:length(yg)));

plot(yg,'o')

ax = gca;
nights = [1 18 32 46];
ax.XTick = nights;
ax.XTickLabel = char(obsv(nights));
grid
```

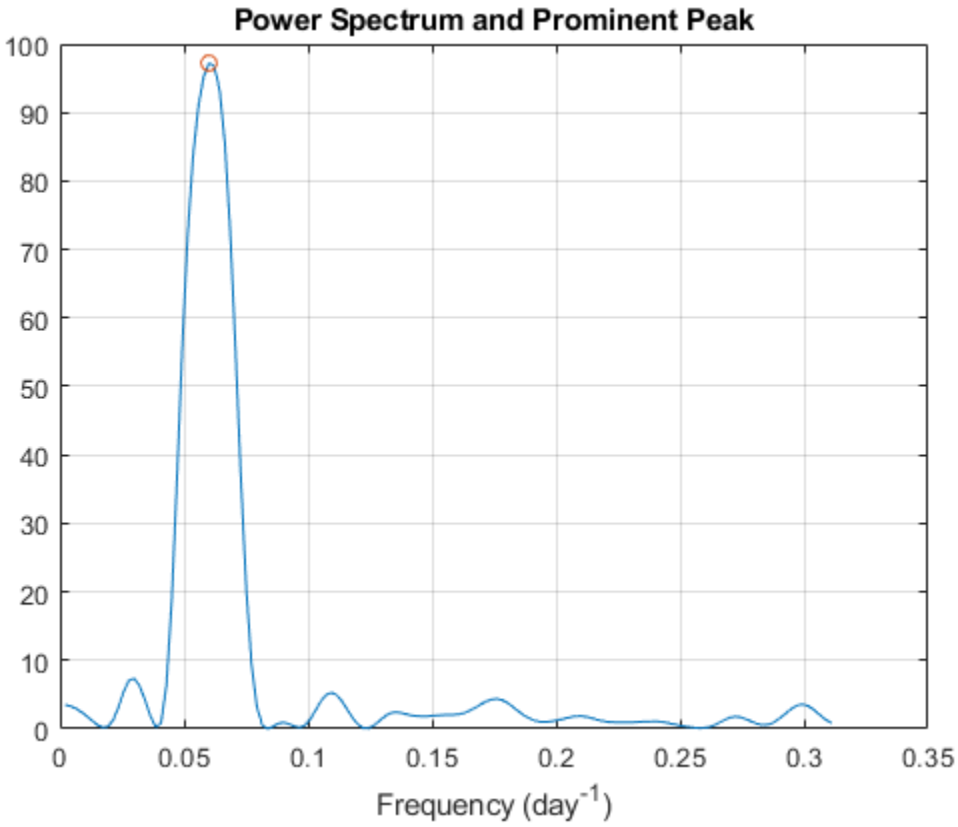



Estimate the power spectrum of the data using `plomb`. Specify an oversampling factor of 10. Express the resulting frequencies in inverse days.

```
[pxx,f] = plomb(yg,obsv,[],10,'power');
f = f*86400;
```

Use `findpeaks` to determine the location of the only prominent peak of the spectrum. Plot the power spectrum and show the peak.

```
[pk,f0] = findpeaks(pxx,f,'MinPeakHeight',10);

plot(f,pxx,f0,pk,'o')
xlabel('Frequency (day^{-1})')
title('Power Spectrum and Prominent Peak')
grid
```




Determine Callisto's orbital period (in days) as the inverse of the frequency of maximum energy. The result differs by less than 1% from the value published by NASA.

```
Period = 1/f0
```

```
Period = 16.6454
```

```
NASA = 16.6890184;
PercentDiscrep = (Period-NASA)/NASA*100
```

```
PercentDiscrep = -0.2613
```

## See Also

`findpeaks` | `plomb`

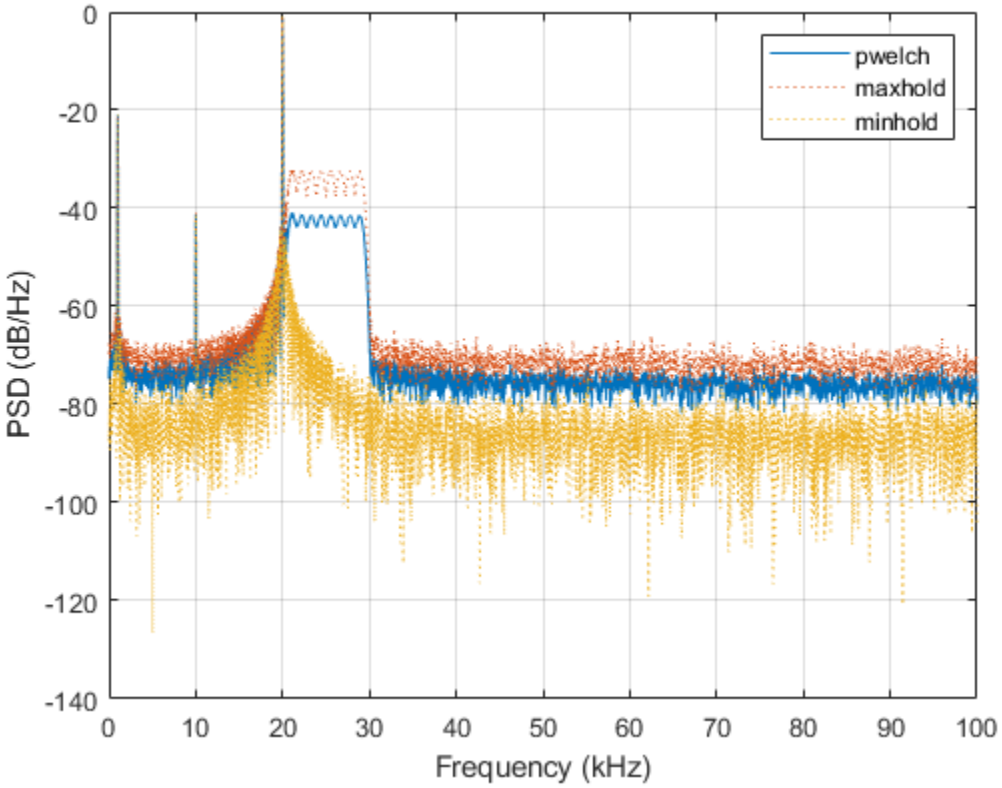
# Welch Spectrum Estimates

Create a signal consisting of three noisy sinusoids and a chirp, sampled at 200 kHz for 0.1 second. The frequencies of the sinusoids are 1 kHz, 10 kHz, and 20 kHz. The sinusoids have different amplitudes and noise levels. The noiseless chirp has a frequency that starts at 20 kHz and increases linearly to 30 kHz during the sampling.

```
Fs = 200e3;
Fc = [1 10 20]'*1e3;
Ns = 0.1*Fs;

t = (0:Ns-1)/Fs;
x = [1 1/10 10]*sin(2*pi*Fc*t)+[1/200 1/2000 1/20]*randn(3,Ns);
x = x+chirp(t,20e3,t(end),30e3);
```

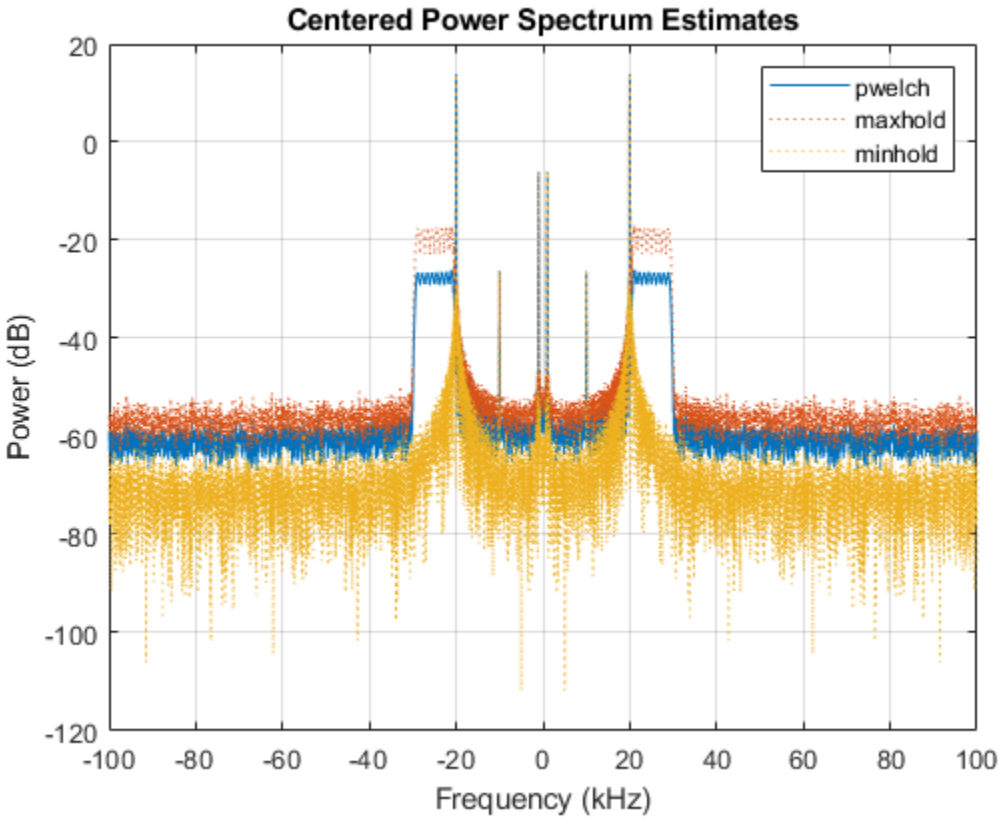
Compute the Welch PSD estimate and the maximum-hold and minimum-hold spectra of the signal. Plot the results.

```
[pxx,f] = pwelch(x,[],[],[],Fs);
pmax = pwelch(x,[],[],[],Fs,'maxhold');
pmin = pwelch(x,[],[],[],Fs,'minhold');

plot(f/1000,pow2db(pxx))
hold on
plot(f/1000,pow2db([pmax pmin]),':')
hold off
xlabel('Frequency (kHz)')
ylabel('PSD (dB/Hz)')
legend('pwelch','maxhold','minhold')
grid
```




Repeat the procedure, this time computing centered power spectrum estimates.

```
[pxx,f] = pwelch(x,[],[],[],Fs,'centered','power');
pmax = pwelch(x,[],[],[],Fs,'maxhold','centered','power');
pmin = pwelch(x,[],[],[],Fs,'minhold','centered','power');

plot(f/1000,pow2db(pxx))
hold on
plot(f/1000,pow2db([pmax pmin]),':')
hold off
xlabel('Frequency (kHz)')
ylabel('Power (dB)')
legend('pwelch','maxhold','minhold')
title('Centered Power Spectrum Estimates')
grid
```




## See Also

`chirp` | `pow2db` | `pwelch`

14

# Time-Frequency Analysis

# Time-Frequency Gallery

This gallery provides you with an overview of the time-frequency analysis features available in Signal Processing Toolbox and Wavelet Toolbox™. The descriptions and usage examples present various methods that you can use for your signal analysis.

| Method | Features | Invertible | Examples |
|---|---|---|---|
| "Short-Time Fourier Transform (Spectrogram)" on page 14-3 | • The short-time Fourier transform (STFT) has fixed time-frequency resolution.<br>• The spectrogram is the magnitude squared of STFT. | • `stft`: Yes<br>• `spectrogram`: No | "Example: Whale Song" on page 14-5 |
| "Continuous Wavelet Transform (Scalogram)" on page 14-7 | • The continuous wavelet transform (CWT) has a variable time-frequency resolution.<br>• The CWT preserves time shifts and time scalings. | Yes | "Example: ECG Signal" on page 14-8 |
| "Wigner-Ville Distribution" on page 14-9 | • The Wigner-Ville distribution (WVD) is always real.<br>• Time and frequency marginal densities correspond to instantaneous power and spectral energy density, respectively.<br>• The time resolution of the WVD is equal to the number of input samples. | No | "Example: Otoacoustic Emission" on page 14-10 |
| "Reassignment and Synchrosqueezing" on page 14-11 | • Reassignment sharpens localization of spectral estimates.<br>• Synchrosqueezing "condenses" time-frequency maps around curves of instantaneous frequency.<br>• Both methods are especially suited to track and extract time-frequency ridges | • `pspectrum`: No<br>• `fsst`, `wsst`: Yes | "Example: Echolocation Pulse" on page 14-11 |

| Method | Features | Invertible | Examples |
|---|---|---|---|
| "Constant-Q Gabor Transform" on page 14-17 | • The constant-*Q* Gabor transform (CQT) tiles the time-frequency plane with variable-sized windows.<br>• The windows have adaptable bandwidth and sampling density.<br>• The ratio of center frequency to bandwidth (*Q*-factor) for all windows is constant. | Yes | "Example: Rock Music" on page 14-18 |
| "Empirical Mode Decomposition and Hilbert-Huang Transform" on page 14-18 | • Empirical mode decomposition (EMD) decomposes signals into intrinsic mode functions.<br>• The Hilbert-Huang transform (HHT) computes the instantaneous frequency of each empirical mode. | No | "Example: Bearing Vibration" on page 14-19 |

## Short-Time Fourier Transform (Spectrogram)

### Description

- The short-time Fourier transform is a linear time-frequency representation useful in the analysis of nonstationary multicomponent signals.
- The short-time Fourier transform is invertible.
- The spectrogram is the magnitude squared of the STFT.
- You can compute the cross-spectrogram of two signals to look for similarities in time-frequency space.
- The *persistence spectrum* of a signal is a time-frequency view that shows the percentage of the time that a given frequency is present in a signal. The persistence spectrum is a histogram in power-frequency space. The longer a particular frequency persists in a signal as the signal evolves, the higher its time percentage and thus the brighter or "hotter" its color in the display.

### Potential Applications

The applications of this time-frequency method include, but are not limited to:

- *Audio signal processing*: Fundamental frequency estimation, cross synthesis, spectral envelope extraction, time-scale modification, time-stretching, and pitch shifting. (See "Phase Vocoder with Different Synthesis and Analysis Windows" for more details.)
- *Crack detection*: Detect cracks in aluminum plates using dispersion curves of ultrasonic Lamb waves.
- *Sensor array processing*: Sonar exploration, geophysical exploration, and beamforming.
- *Digital communications*: Detection of frequency hopping signal.

### How to Use

- `stft` computes the short-time Fourier transform. To invert the short-time Fourier transform, use the `istft` function.
- `pspectrum` or `spectrogram` computes the spectrogram.
- `xspectrogram` computes the cross-spectrogram of two signals.
- You can also use the spectrogram view in **Signal Analyzer** to view the spectrogram of a signal.
- Use the persistence spectrum option in `pspectrum` or **Signal Analyzer** to identify signals hidden in other signals.

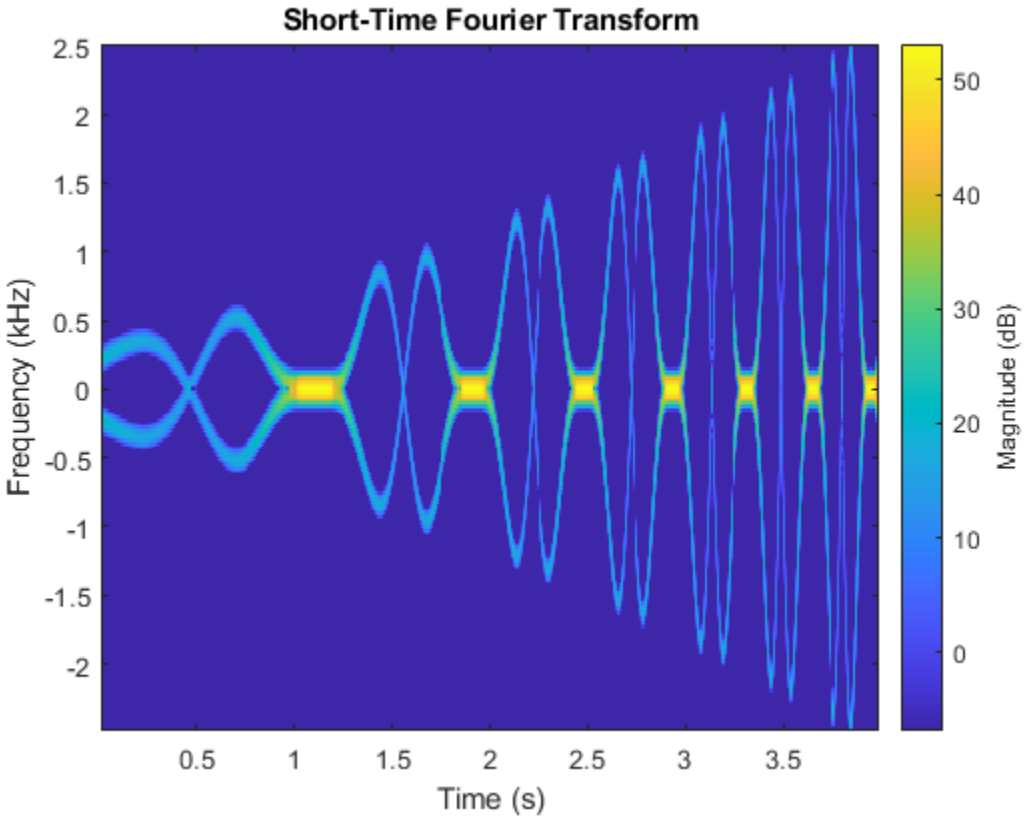
### Example: Pulses and Oscillations

Generate a signal sampled at 5 kHz for 4 seconds. The signal consists of a set of pulses of decreasing duration separated by regions of oscillating amplitude and fluctuating frequency with an increasing trend.

```
fs = 5000;
t = 0:1/fs:4-1/fs;

x = 10*besselj(0,1000*(sin(2*pi*(t+2).^3/60).^5));
```

Compute and plot the short-time Fourier transform of the signal. Window the signal with a 200-sample Kaiser window with shape factor $\beta = 30$.

```
stft(x,fs,'Window',kaiser(200,30))
```



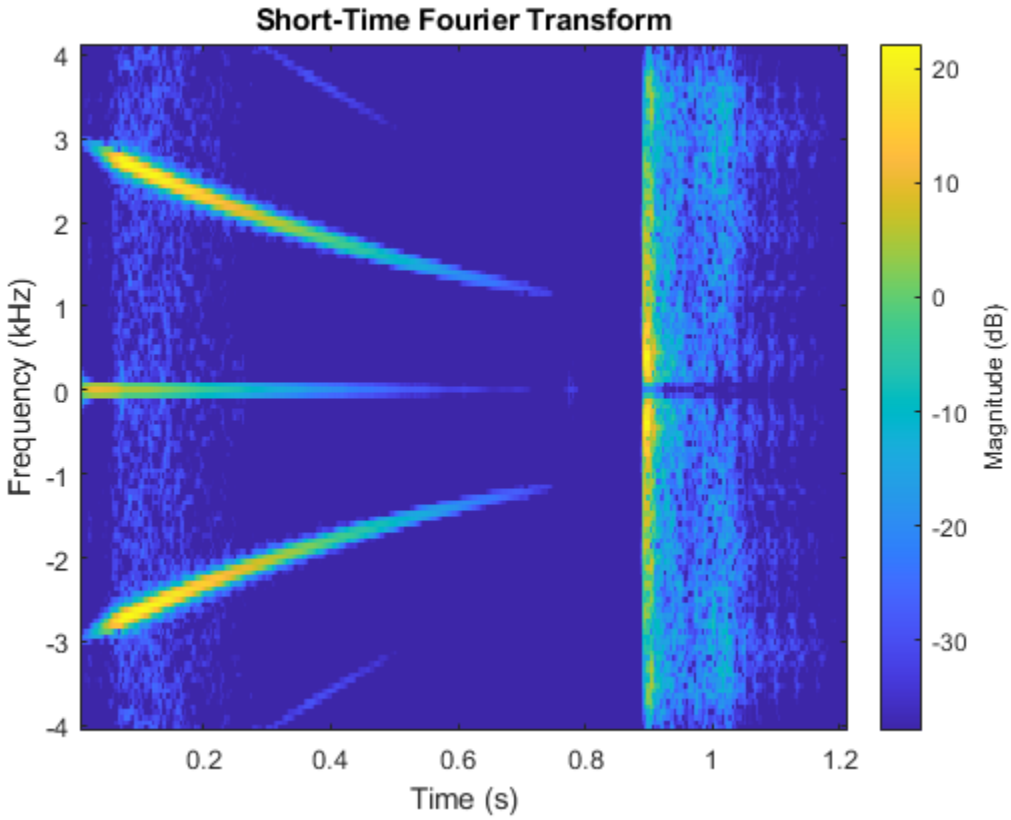
### Example: Audio Signal with Decreasing Chirps

Load an audio signal that contains two decreasing chirps and a wideband splatter sound.

```
load splat
```

Set the overlap length to 96 samples. Plot the short-time Fourier transform.

```
stft(y,Fs,'OverlapLength',96)
```



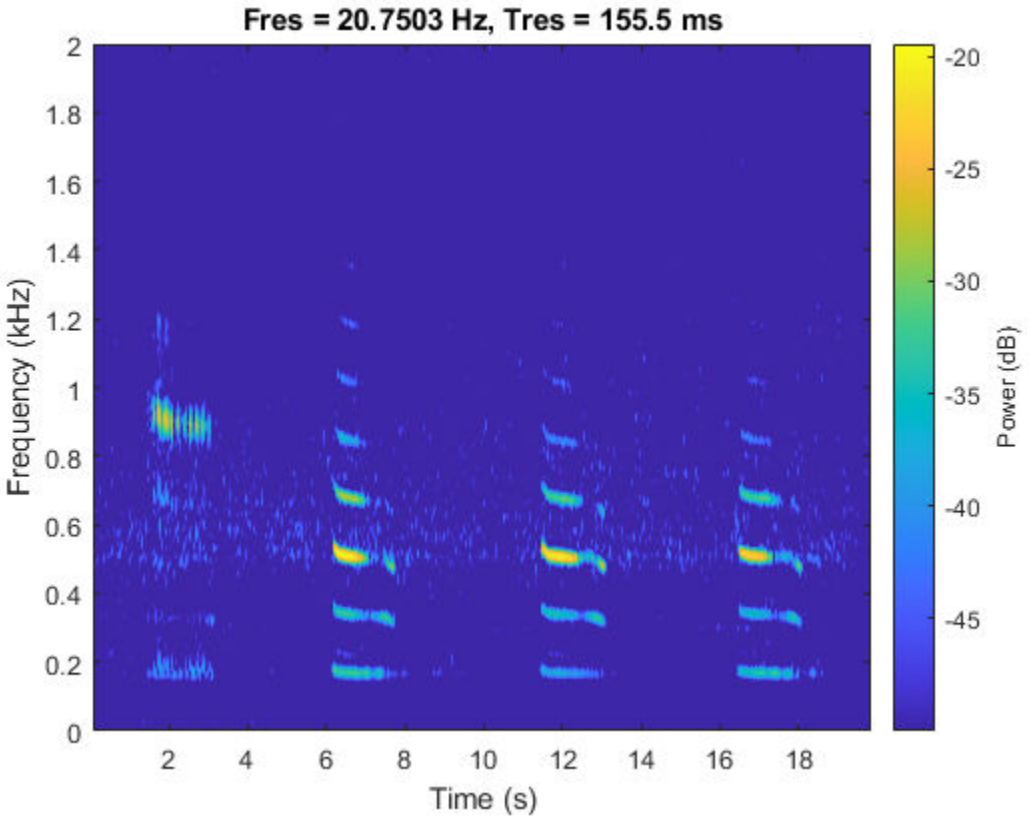

### Example: Whale Song

Load a file that contains audio data from a Pacific blue whale, sampled at 4 kHz. The file is from the library of animal vocalizations maintained by the Cornell University Bioacoustics Research Program. The time scale in the data is compressed by a factor of 10 to raise the pitch and make the calls more audible.

```
whaleFile = fullfile(matlabroot,'examples','matlab','data','bluewhale.au');
[w,fs] = audioread(whaleFile);
```

Compute the spectrogram of the whale song with an overlap percentage equal to eighty percent. Set the minimum threshold for the spectrogram to `-50` dB.

```
pspectrum(w,fs,'spectrogram','Leakage',0.2,'OverlapPercent',80,'MinThreshold',-50)
```



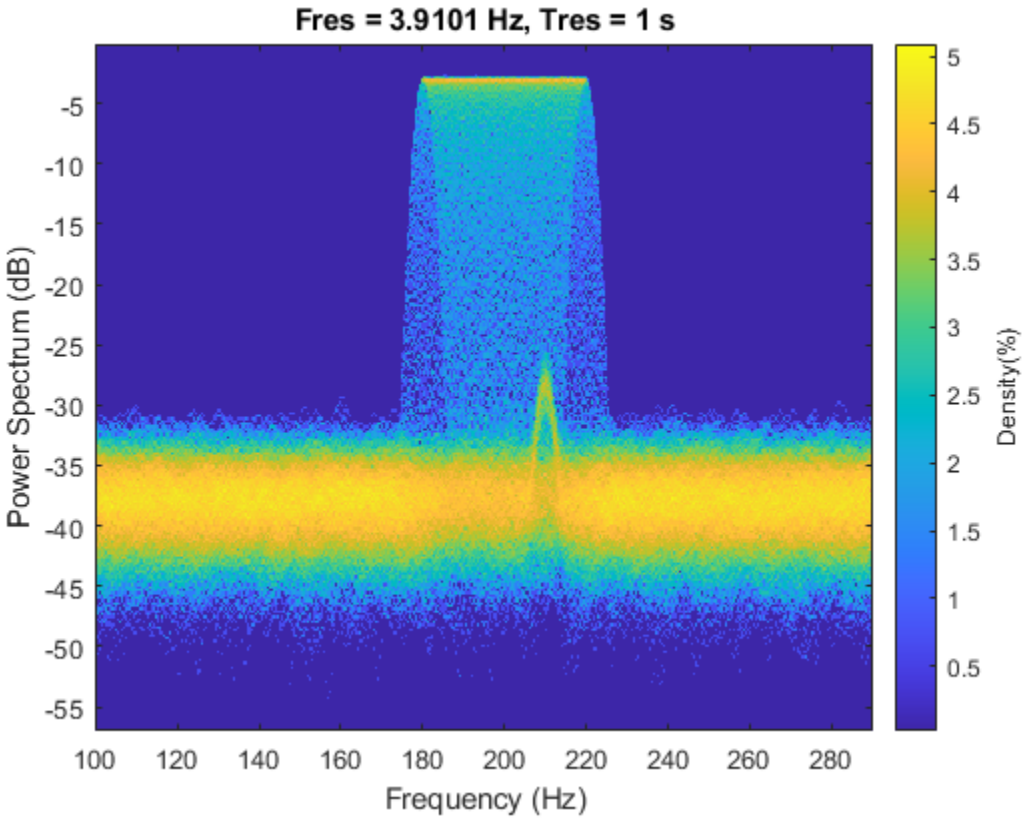

**Example: Persistence Spectrum of Transient Signal**

Load an interference narrowband signal embedded within a broadband signal.

```
load TransientSig
```

Compute the persistence spectrum of the signal. Both signal components are clearly visible.

```
pspectrum(x,fs,'persistence', ...
    'FrequencyLimits',[100 290],'TimeResolution',1)
```




## Continuous Wavelet Transform (Scalogram)

### Description

- The wavelet transform is a linear time-frequency representation that preserves time shifts and time scalings.
- The continuous wavelet transform is good at detecting transients in nonstationary signals, and for signals in which instantaneous frequency grows rapidly.
- The CWT is invertible.
- The CWT tiles the time-frequency plane with variable-sized windows. The window automatically widens in time, making it suitable for low-frequency phenomena, and narrows for high frequency phenomena.

### Potential Applications

The applications of this time-frequency method include, but are not limited to:

- *Electrocardiograms (ECG)*: The most clinically useful information of the ECG signal is found in the time intervals between its consecutive waves and amplitudes defined by its features. The wavelet transform breaks down the ECG signal into scales, making it easier to analyze the ECG signal in different frequency ranges easier to analyze.
- *Electroencephalogram (EEG)*: Raw EEG signals suffer from poor spatial resolution, low signal-to-noise ratio, and artifacts. Continuous wavelet decomposition of a noisy signal concentrates

intrinsic signal information in a few wavelet coefficients having large absolute values without modifying the random distribution of noise. Therefore, denoising can be achieved by thresholding the wavelet coefficients.

- *Signal demodulation*: Demodulate extended binary phase shift keying (EBPSK) using an adaptive wavelet construction method.
- *Deep learning*: The CWT can be used to create time-frequency representations that can be used to train a convolutional neural network. “Classify Time Series Using Wavelet Analysis and Deep Learning” (Wavelet Toolbox) shows how to classify ECG signals using scalograms and transfer learning.

### How to Use

- `cwt` computes the continuous wavelet transform and displays the scalogram. Alternatively, create a CWT filter bank using `cwtfilterbank` and apply the `wt` function. Use this method to run in parallel applications or when computing the transform for several functions in a loop.
- `icwt` inverts the continuous wavelet transform.
- **Signal Analyzer** has a scalogram view to visualize the CWT of a time series.

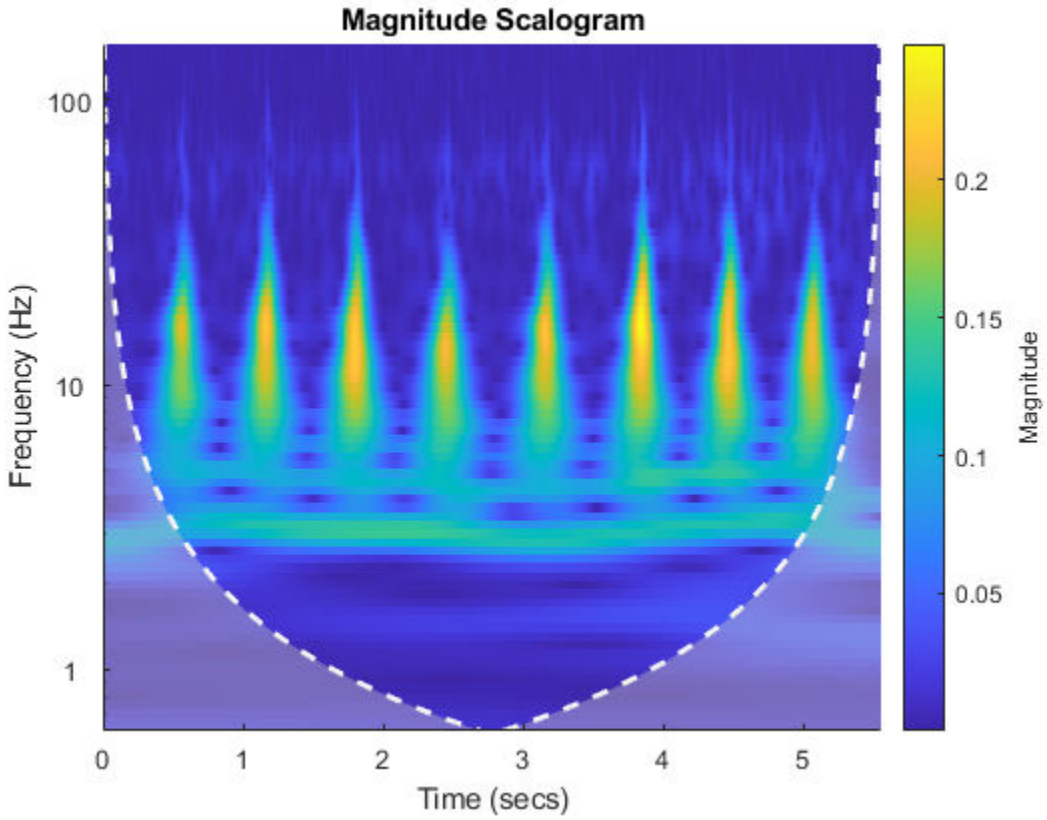
### Example: ECG Signal

Load a noisy ECG waveform sampled at 360 Hz.

```
load ecg
Fs = 360;
```

Compute the continuous wavelet transform.

```
cwt(ecg,Fs)
```




The ECG data is taken from the MIT-BIH Arrhythmia Database [2].

## Wigner-Ville Distribution

### Description

- The Wigner-Ville distribution (WVD) is a quadratic energy density computed by correlating the signal with a time and frequency translated and complex-conjugated version of itself.
- The Wigner-Ville distribution is always real even if the signal is complex.
- Time and frequency marginal densities correspond to instantaneous power and spectral energy density, respectively.
- The instantaneous frequency and group delay can be evaluated using local first-order moments of the Wigner distribution.
- The time resolution of the WVD is equal to the number of input samples.
- The Wigner distribution can locally assume negative values.

### Potential Applications

The applications of this time-frequency method include, but are not limited to:

- *Otoacoustic emissions (OAEs)*: OAEs are narrowband oscillatory signals emitted by the cochlea (inner ear), and their presence is indicative of normal hearing.

- *Quantum mechanics*: Quantum corrections to classical statistical mechanics, model electron transport, and calculate static and dynamic properties of many-body quantum systems.

### How to Use

- `wvd` computes the Wigner-Ville distribution.
- `xwvd` computes the cross Wigner-Ville distribution of two signals. See “Use Cross Wigner-Ville Distribution to Estimate Instantaneous Frequency” for more details.

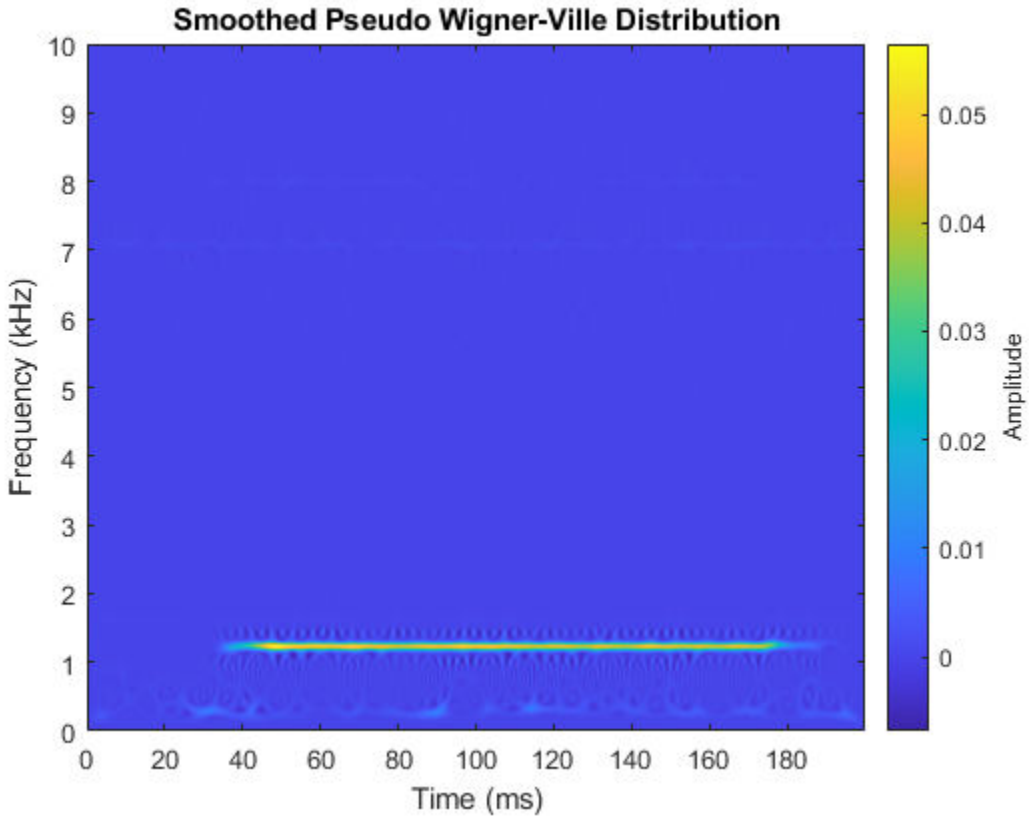
### Example: Otoacoustic Emission

Load a data file containing otoacoustic emission data sampled at 20 kHz. The emission is produced by a stimulus beginning at 25 milliseconds and ending at 175 milliseconds.

```
load dpoae
Fs = 20e3;
```

Compute the smoothed-pseudo Wigner Ville distribution of the otoacoustic data. The convenience plot isolates the emission frequency at roughly the expected value 1.2 kHz.

```
wvd(dpoaets,Fs,'smoothedPseudo',kaiser(511,10),kaiser(511,10),'NumFrequencyPoints',4000,'NumTimeI
```




For more details on otoacoustic emissions, see "Determining Exact Frequency Through the Analytic CWT" in “CWT-Based Time-Frequency Analysis” (Wavelet Toolbox).

## Reassignment and Synchrosqueezing

### Description

- Reassignment sharpens the localization of spectral estimates and produces spectrograms that are easier to read and interpret. The technique relocates each spectral estimate to the center of energy of its bin instead of the bin's geometric center. It provides exact localization for chirps and impulses.
- The Fourier synchrosqueezed transform starts from the short-time Fourier transform and "squeezes" its values so that they concentrate around curves of instantaneous frequency in the time-frequency plane.
- The wavelet synchrosqueezed transform reassigns the signal energy in frequency.
- Both the Fourier synchrosqueezed transform and the wavelet synchrosqueezed transform are invertible.
- The reassigned and synchrosqueezing methods are especially suited to track and extract time-frequency ridges.

### Potential Applications

The applications of this time-frequency method include, but are not limited to:

- *Audio signal processing*: Synchrosqueezing transform (SST) was originally introduced in the context of audio signal analysis.
- *Seismic data*: Analysis of seismic data to find oil and gas traps. Synchrosqueezing can also detect deep-layer weak signals that are usually smeared in seismic data.
- *Oscillations in power systems*: A steam turbine and electric generator can have mechanical subsynchronous oscillation (SSO) modes between the various turbine stages and the generator. The frequency of the SSO is generally between 5 Hz and 45 Hz, and the mode frequencies are often close to each other. The antinoise ability and time-frequency resolution of WSST improves the readability of the time-frequency view.
- *Deep learning*: Synchrosqueezed transforms can be used to extract time-frequency features and fed into a network that classifies time-series data. “Waveform Segmentation Using Deep Learning” on page 24-330 shows how `fsst` outputs can be fed into an LSTM network that classifies ECG signals.

### How to Use

- Use the `'reassigned'` option in `spectrogram`, set the `'Reassigned'` argument to `true` in `pspectrum`, or check the **Reassign** box in the spectrogram view of **Signal Analyzer** to compute reassigned spectrograms.
- `fsst` computes the Fourier synchrosqueezed transform. Use the `ifsst` function to invert the Fourier synchrosqueezed transform. (See “Fourier Synchrosqueezed Transform of Speech Signal” for reconstruction of speech signals using `ifsst`.)
- `wsst` computes the wavelet synchrosqueezed transform. Use the `iwsst` function to invert the wavelet synchrosqueezed transform. (See “Inverse Synchrosqueezed Transform of Chirp” (Wavelet Toolbox) for reconstruction of a quadratic chirp using `iwsst`.)

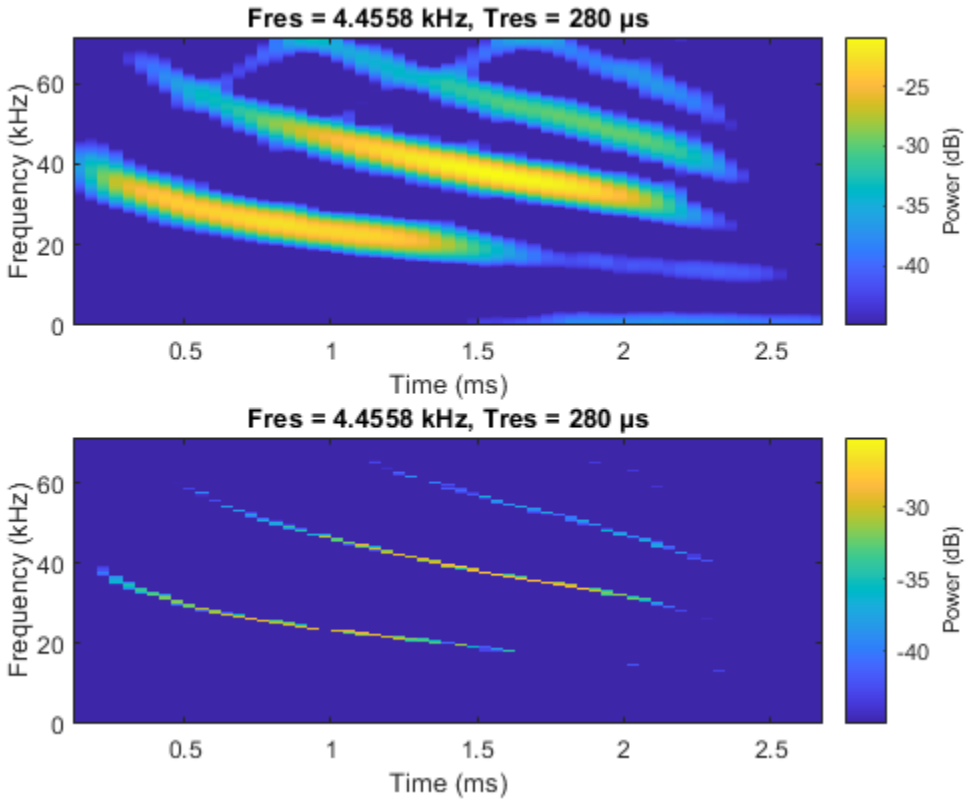
### Example: Echolocation Pulse

Load an echolocation pulse emitted by a big brown bat (Eptesicus Fuscus). The sampling interval is 7 microseconds.

```
load batsignal
Fs = 1/DT;
```

Compute the reassigned spectrogram of the signal.

```
subplot(2,1,1)
pspectrum(batsignal,Fs,'spectrogram','TimeResolution',280e-6, ...
    'OverlapPercent',85,'MinThreshold',-45,'Leakage',0.9)
subplot(2,1,2)
pspectrum(batsignal,Fs,'spectrogram','TimeResolution',280e-6, ...
    'OverlapPercent',85,'MinThreshold',-45,'Leakage',0.9,'Reassign',true)
```




Thanks to Curtis Condon, Ken White, and Al Feng of the Beckman Center at the University of Illinois for the bat data and permission to use it in this example [3].

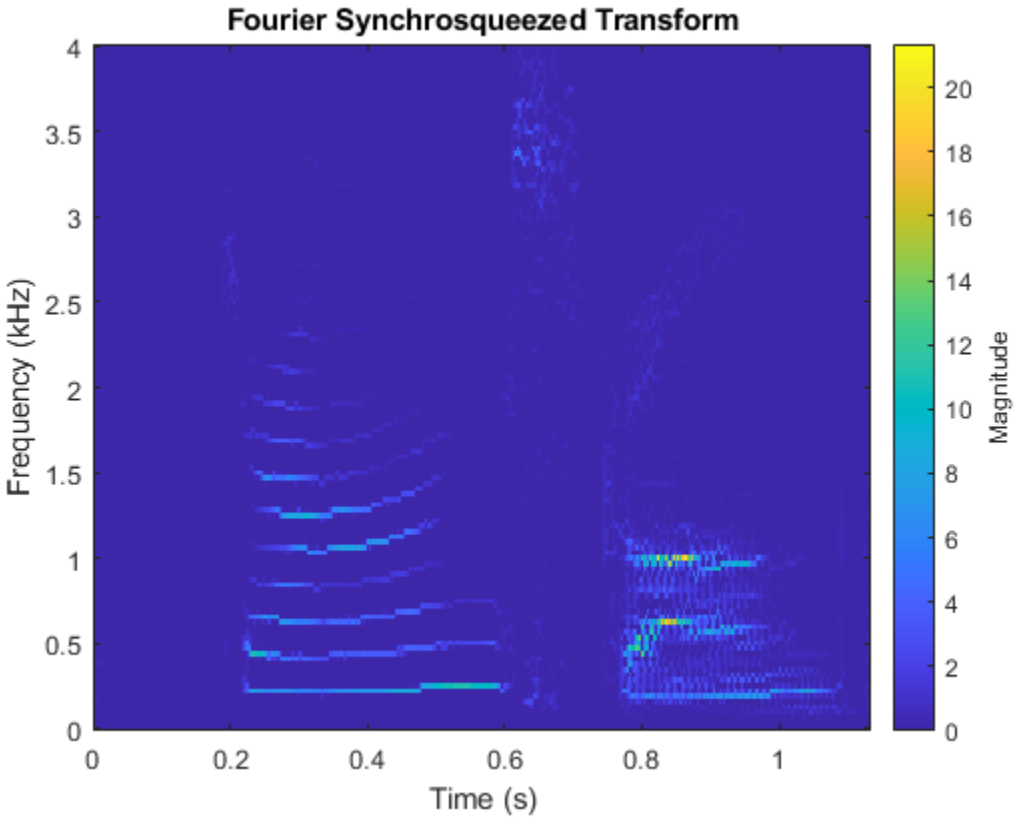
### Example: Speech Signals

Load a file containing the word "strong," spoken by a woman and by a man. The signals are sampled at 8 kHz. Concatenate them into a single signal.

```
load Strong
x = [her' him'];
```

Compute the synchrosqueezed Fourier transform of the signal. Window the signal using a Kaiser window with shape factor $\beta = 20$.

```
fsst(x,Fs,kaiser(256,20),'yaxis')
```



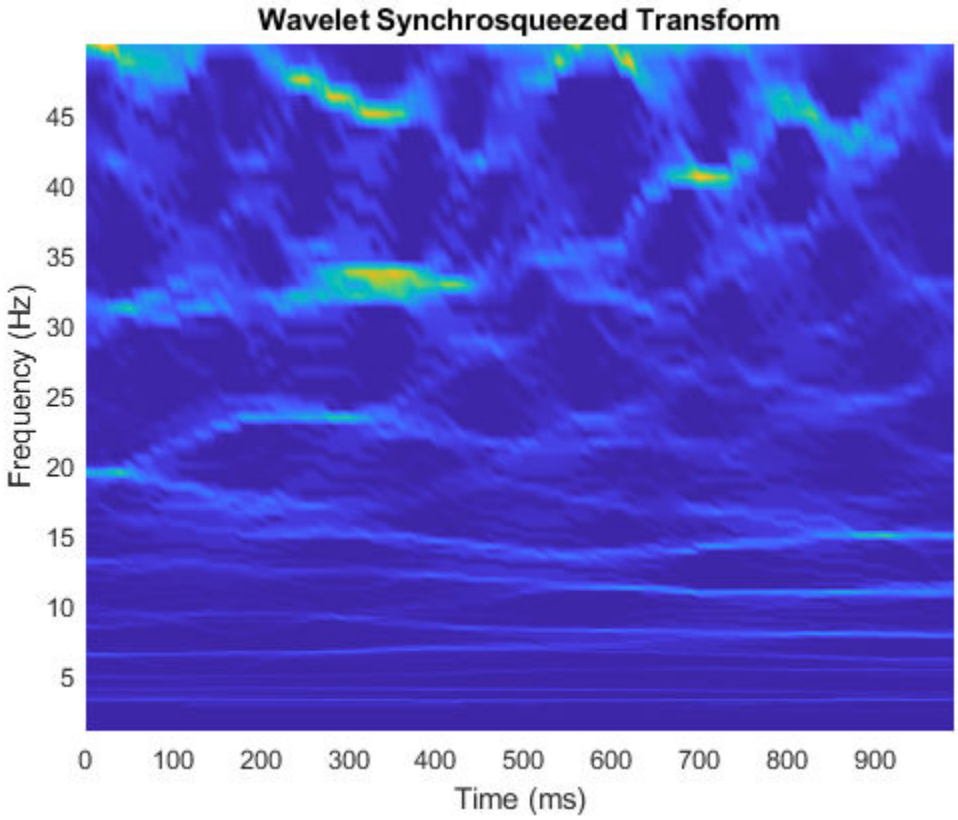

**Example: Synthetic Seismic Data**

Load the synthetic seismic data sampled at 100 Hz for 1 second.

```
load SyntheticSeismicData
```

Compute the wavelet synchrosqueezed transform of the seismic data using the bump wavelet and 30 voices per octave.

```
wsst(x,Fs,'bump','VoicesPerOctave',30,'ExtendSignal',true)
```




The seismic signal is generated using the two sinusoids mentioned in "Time-Frequency Analysis of Seismic Data Using Synchrosqueezing Transform" by Ping Wang, Jinghuai Gao, and Zhiguo Wang [4].

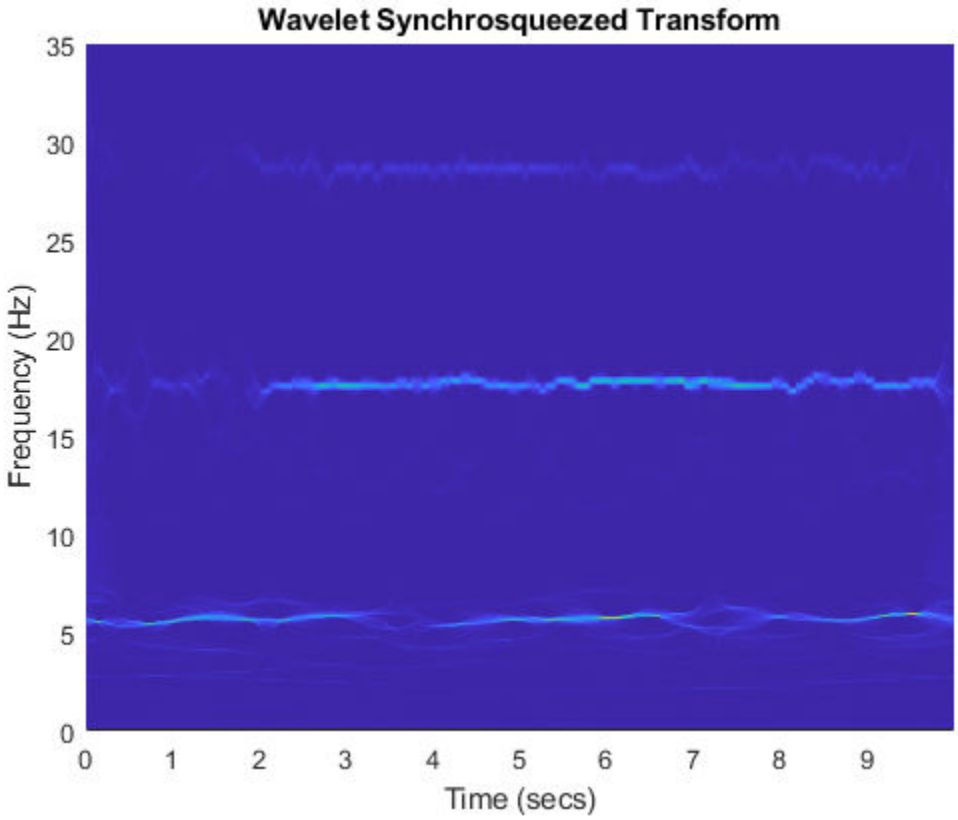
### Example: Earthquake Vibration

Load acceleration measurements recorded on the first floor of a three story test structure under earthquake conditions. The measurements are sampled at 1 kHz.

```
load quakevib
Fs = 1e3;
```

Compute the wavelet synchrosqueezed transform of the acceleration measurements. You are analyzing vibration data that exhibit a cyclic behavior. The synchrosqueezed transform allows you to isolate the three frequency components, separated by roughly 11 Hz. The main vibration frequency is at 5.86 Hz, and the equispaced frequency peaks suggest that they are harmonically related. The cyclic behavior of the vibrations is also visible.

```
wsst(gfloor1OL,Fs,'bump','VoicesPerOctave',48)
ylim([0 35])
```



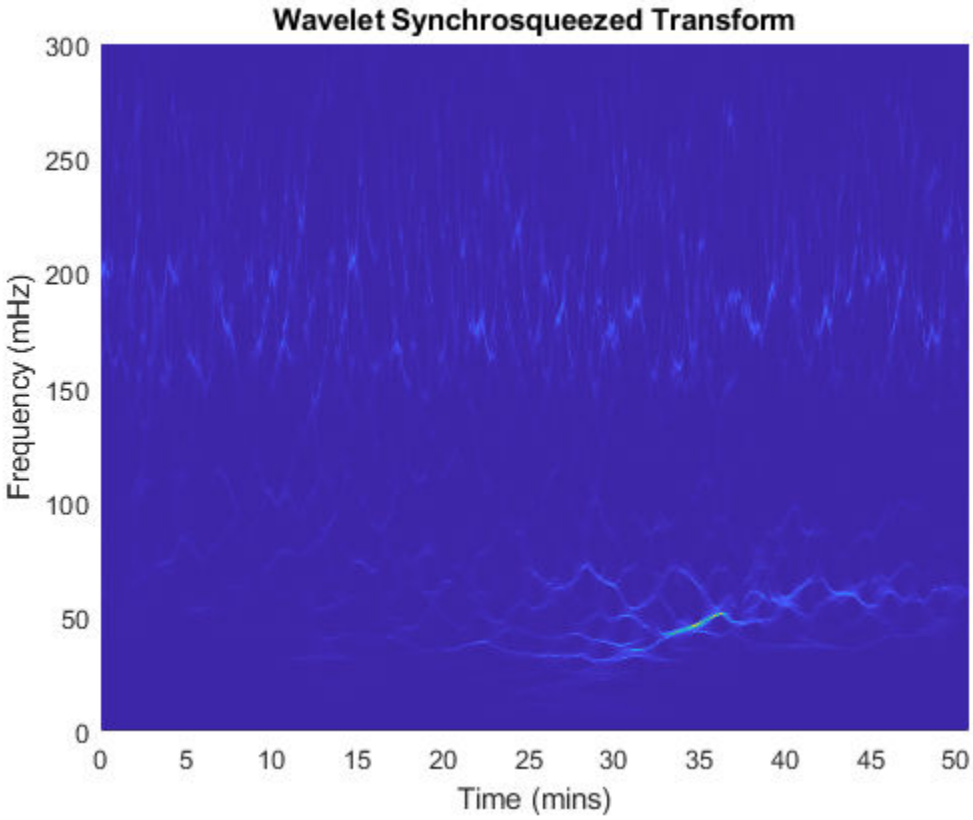

**Example: Kobe Earthquake Data**

Load seismograph data recorded during the 1995 Kobe earthquake. The data has a sample rate of 1 Hz.

```
load kobe
Fs = 1;
```

Compute the wavelet synchrosqueezed transform that isolates the different frequency components of the seismic data.

```
wsst(kobe,Fs,'bump','VoicesPerOctave',48)
ylim([0 300])
```




The data are seismograph (vertical acceleration, nm/sq.sec) measurements recorded at Tasmania University, Hobart, Australia on 16 January 1995 beginning at 20:56:51 (GMT) and continuing for 51 minutes at 1 second intervals [5].

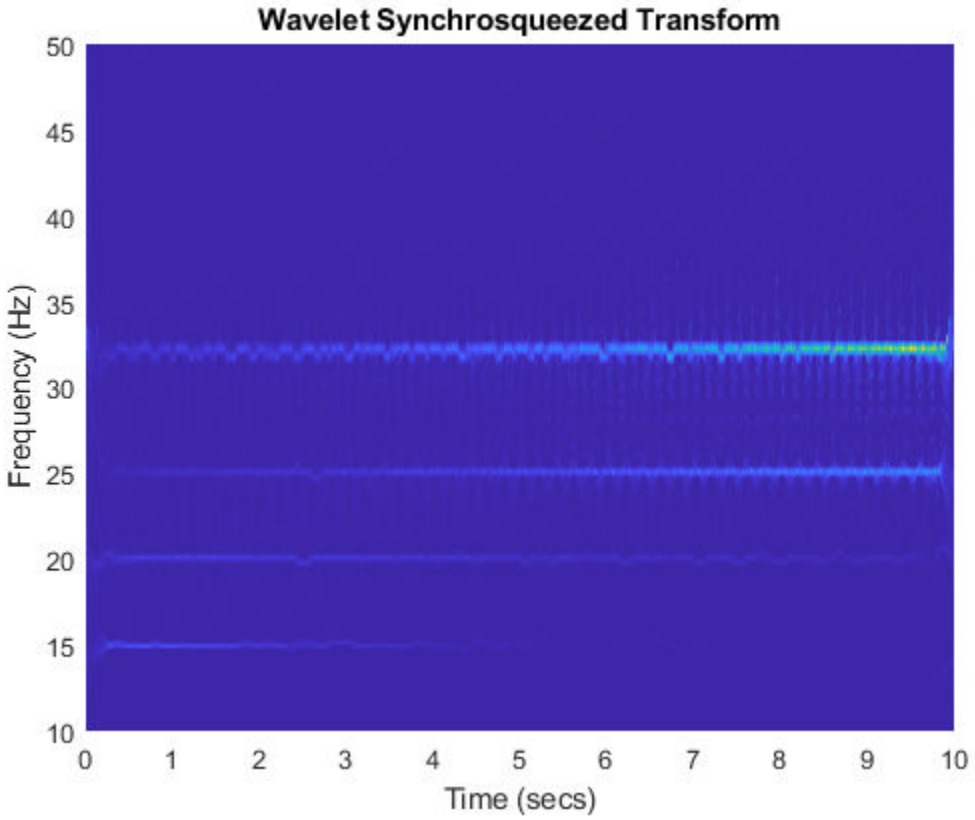
### Example: Subsynchronous Oscillation in Power Systems

Load the subsynchronous oscillation data of a Power System.

```
load OscillationData
```

Compute the wavelet synchrosqueezed transform using the bump wavelet and 48 voices per octave. The four mode frequencies are at 15 Hz, 20 Hz, 25 Hz and 32 Hz. Notice that the energies of the modes at 15 Hz and 20 Hz decrease with time, whereas the energy of the modes at 25 Hz and 32 Hz increase gradually over time.

```
wsst(x,Fs,'bump','VoicesPerOctave',48)
ylim([10 50])
```




This synthetic subsynchronous oscillation data was generated using the equation defined by Zhao et al in "Application of Synchrosqueezed Wavelet Transforms for Extraction of the Oscillatory Parameters of Subsynchronous Oscillation in Power Systems" [6].

## Constant-Q Gabor Transform

### Description

- The constant-*Q* nonstationary Gabor transform uses windows with different center frequencies and bandwidths such that the ratio of center frequency to bandwidth, the *Q* factor, remains constant.
- The constant-*Q* Gabor transform enables the construction of stable inverses, yielding perfect signal reconstruction.
- In frequency space, the windows are centered at logarithmically spaced center frequencies.

### Potential Applications

The applications of this time-frequency method include, but are not limited to:

*Audio signal processing*: The fundamental frequencies of the tones in music are geometrically spaced. The frequency resolution of the human auditory system is approximately constant-*Q*, making this technique appropriate for music signal processing.

### How to Use

- `cqt` computes the constant-*Q* Gabor transform.
- `icqt` inverts the constant-*Q* Gabor transform.

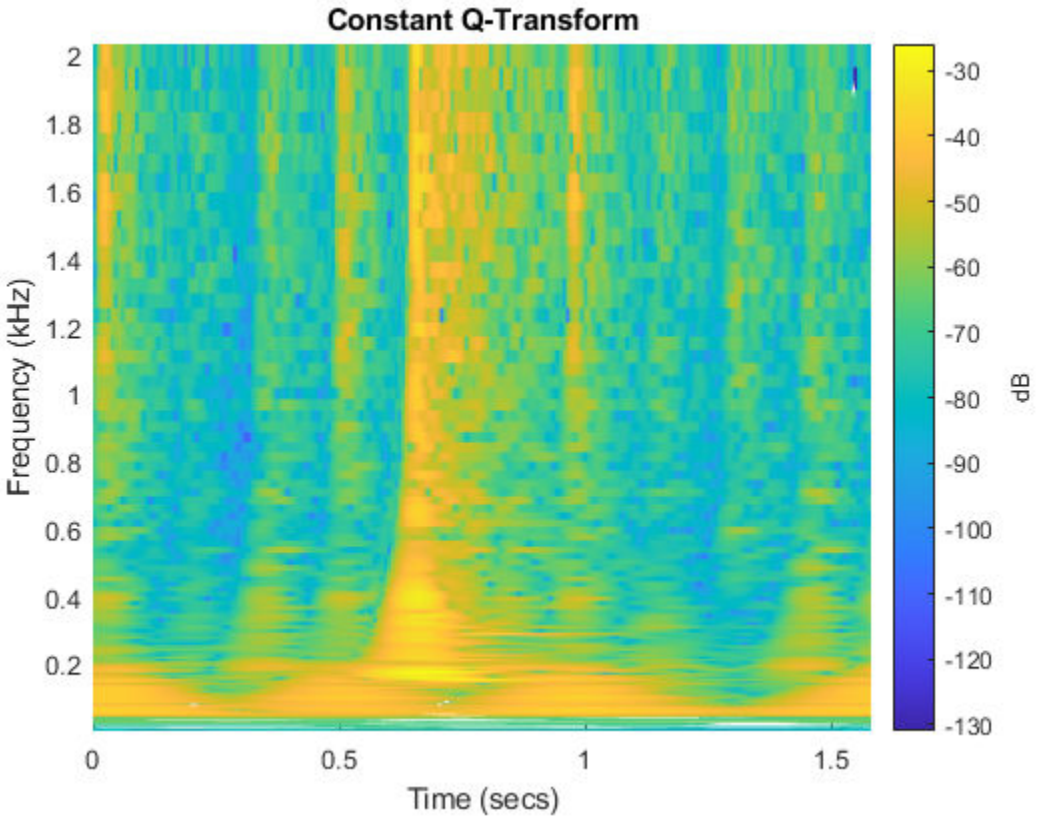
### Example: Rock Music

Load an audio file containing a fragment of Rock music with vocals, drums, and guitar. The signal has a sample rate of 44.1 kHz.

```
load drums
```

Set the frequency range over which the CQT has a logarithmic frequency response to be the minimum allowable frequency to 2 kHz. Perform the CQT of the signal using 20 bins per octave.

```
minFreq = fs/length(audio);
maxFreq = 2000;
cqt(audio,'SamplingFrequency',fs,'BinsPerOctave',20,'FrequencyLimits',[minFreq maxFreq])
```




## Empirical Mode Decomposition and Hilbert-Huang Transform

### Description

- The empirical mode decomposition decomposes the signals into intrinsic mode functions which form a complete and nearly orthogonal basis for the original signal.

- The Hilbert-Huang transform computes the instantaneous frequency of each intrinsic mode function.
- These two methods combined are useful for analyzing nonlinear and nonstationary signals.

### Potential Applications

The applications of this time-frequency method include, but are not limited to:

- *Physiological signal processing*: Analyze human EEG response to transcranial magnetic stimulation (TMS) of the brain cortex.
- *Structural applications*: Locate anomalies that appear as cracks, delamination, or stiffness loss in beams and plates.
- *System identification*: Isolate modal damping ratios of structures with closely spaced modal frequencies.
- *Ocean engineering*: Identify transient electromagnetic disturbances caused by humans in underwater electromagnetic environments.
- *Solar physics*: Extract periodic components of sunspot data.
- *Atmospheric turbulence*: Observe stable boundary layer to separate turbulent and nonturbulent motions.
- *Epidemiology*: Assess traveling speed of communicative diseases such as Dengue fever.

### How to Use

- `emd` computes the empirical mode decomposition.
- `hht` computes the Hilbert Huang spectrum of an empirical mode decomposition.

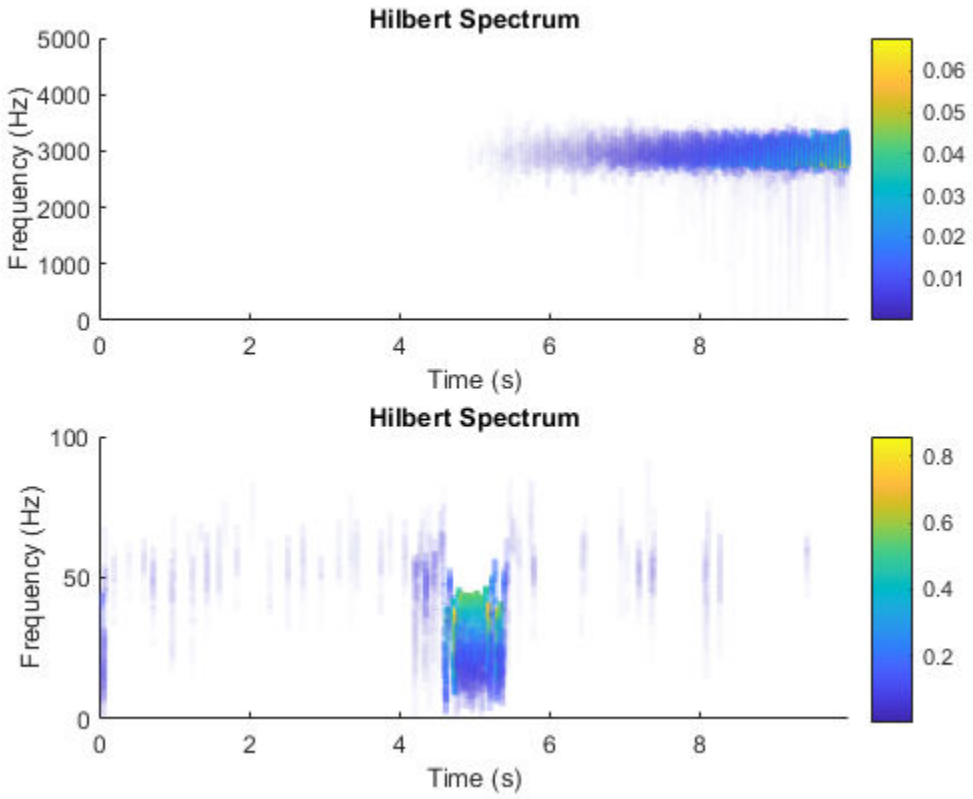
### Example: Bearing Vibration

Load the vibration signal from a defective bearing generated in the “Compute Hilbert Spectrum of Vibration Signal” example. The signal is sampled at a rate 10 kHz.

```
load bearingVibration
```

Compute the first five intrinsic mode functions (IMFs) of the signal. Plot the Hilbert spectrum of the first and third empirical modes. The first mode reveals increasing wear due to high-frequency impacts on the bearing's outer race. The third mode shows a resonance occurring halfway through the measurement process that caused the defect in the bearing.

```
imf = emd(y,'MaxNumIMF',5,'Display',0);
subplot(2,1,1)
hht(imf(:,1),fs)
subplot(2,1,2)
hht(imf(:,3),fs,'FrequencyLimits',[0 100])
```




## References

[1] The Pacific blue whale file is obtained from the library of animal vocalizations maintained by the Cornell University Bioacoustics Research Program.

[2] Moody G. B, Mark R. G. *The impact of the MIT-BIH Arrhythmia Database*. IEEE Eng in Med and Biol 20(3):45-50 (May-June 2001). (PMID: 11446209)

[3] Thanks to Curtis Condon, Ken White, and Al Feng of the Beckman Center at the University of Illinois for the bat echolocation data.

[4] Wang, Ping, Gao, J., and Wang, Z. *Time-Frequency Analysis of Seismic Data Using Synchrosqueezing Transform*, IEEE Geoscience and Remote Sensing Letters, Vol 12, Issue 11, Dec. 2014.

[5] Seismograph (vertical acceleration, nm/sq.sec) of the Kobe earthquake, recorded at Tasmania University, Hobart, Australia on 16 January 1995 beginning at 20:56:51 (GMTRUE) and continuing for 51 minutes at 1 second intervals.

[6] Zhao et al. *Application of Synchrosqueezed Wavelet Transforms for Extraction of the Oscillatory Parameters of Subsynchronous Oscillation in Power Systems* MDPI Energies; Published 12 June 2018.

[7] Boashash, Boualem. *Time-Frequency Signal Analysis and Processing: A Comprehensive Reference* Elsevier, 2016.

## See Also

**Apps**
**Signal Analyzer** | **Signal Multiresolution Analyzer**

**Functions**
`cqt` | `cwt` | `cwtfilterbank` | `emd` | `fsst` | `hht` | `icqt` | `icwt` | `ifsst` | `istft` | `iwsst` | `kurtogram` | `pkurtosis` | `pspectrum` | `spectrogram` | `stft` | `wsst` | `wt` | `wvd` | `xspectrogram` | `xwvd`

# Linear Prediction


- “Prediction Polynomial” on page 15-2
- “Formant Estimation with LPC Coefficients” on page 15-4
- “AR Order Selection with Partial Autocorrelation Sequence” on page 15-7

# Prediction Polynomial

This example shows how to obtain the prediction polynomial from an autocorrelation sequence. The example also shows that the resulting prediction polynomial has an inverse that produces a stable all-pole filter. You can use the all-pole filter to filter a wide-sense stationary white noise sequence to produce a wide-sense stationary autoregressive process.

Create an autocorrelation sequence defined by

$$r(k) = \frac{24}{5}2^{-|k|} - \frac{27}{10}3^{-|k|}, \quad k = 0, 1, 2\,.$$

```
k = 0:2;
rk = (24/5)*2.^(-k)-(27/10)*3.^(-k);
```

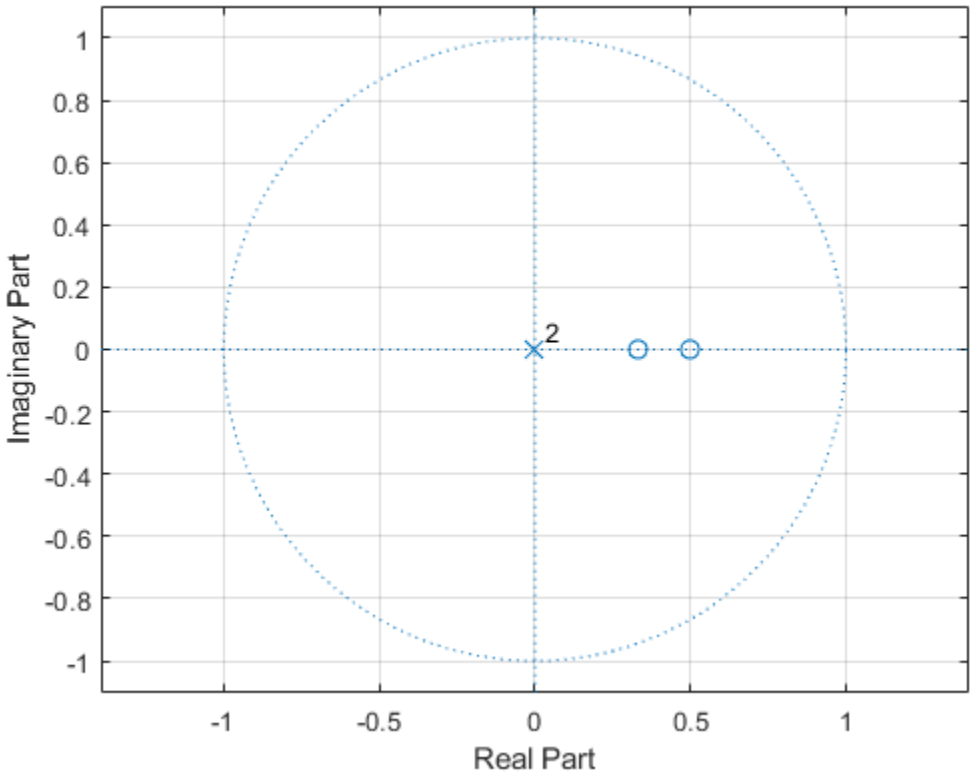
Use `ac2poly` to obtain the prediction polynomial of order 2, which is

$$A(z) = 1 - \frac{5}{6}z^{-1} + \frac{1}{6}z^{-2}\,.$$

```
A = ac2poly(rk);
```

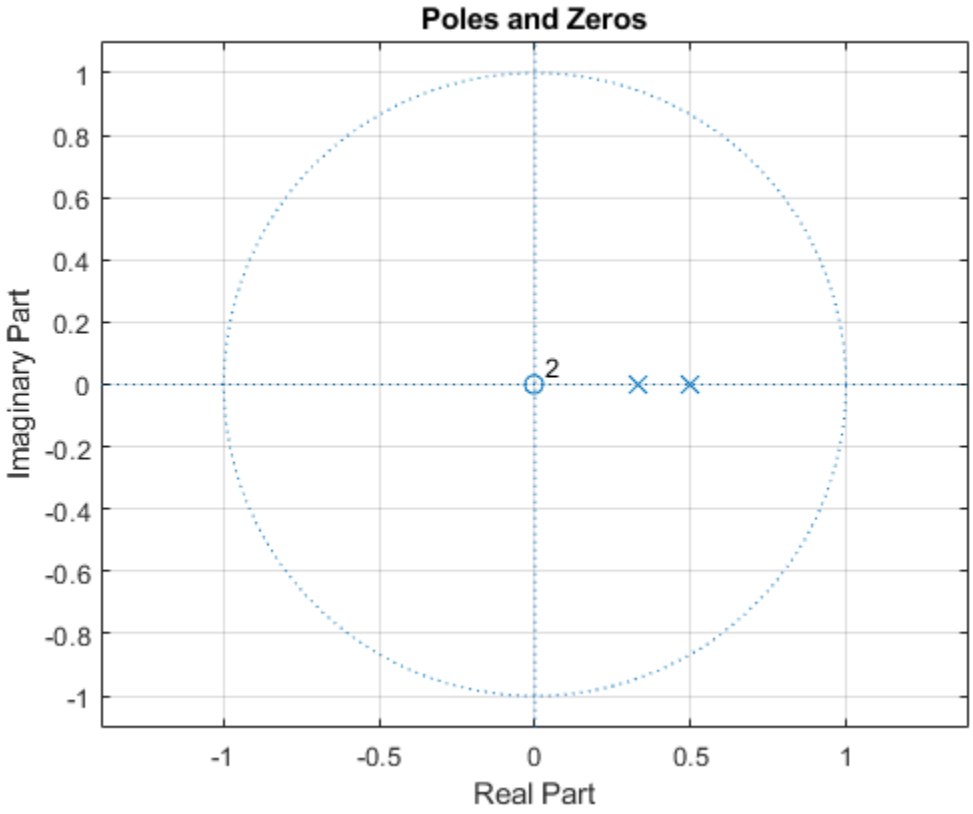
Examine the pole-zero plot of the FIR filter to see that the zeros are inside the unit circle.

```
zplane(A,1)
grid
```



The inverse all-pole filter is stable with poles inside the unit circle.

```
zplane(1,A)
grid
title('Poles and Zeros')
```




Use the all-pole filter to produce a realization of a wide-sense stationary AR(2) process from a white-noise sequence. Set the random number generator to the default settings for reproducible results.

```
rng default

x = randn(1000,1);
y = filter(1,A,x);
```

Compute the sample autocorrelation of the AR(2) realization and show that the sample autocorrelation is close to the true autocorrelation.

```
[xc,lags] = xcorr(y,2,'biased');
[xc(3:end) rk']
```

```
ans = 3×2

    2.2401    2.1000
    1.6419    1.5000
    0.9980    0.9000
```

# Formant Estimation with LPC Coefficients

This example shows how to estimate vowel formant frequencies using linear predictive coding (LPC). The formant frequencies are obtained by finding the roots of the prediction polynomial.

This example uses the speech sample `mtlb.mat`, which is part of Signal Processing Toolbox™. The speech is lowpass-filtered. Because of the low sampling frequency, this speech sample is not optimal for this example. The low sampling frequency limits the order of the autoregressive model you can fit to the data. In spite of this limitation, the example illustrates the technique for using LPC coefficients to determine vowel formants.

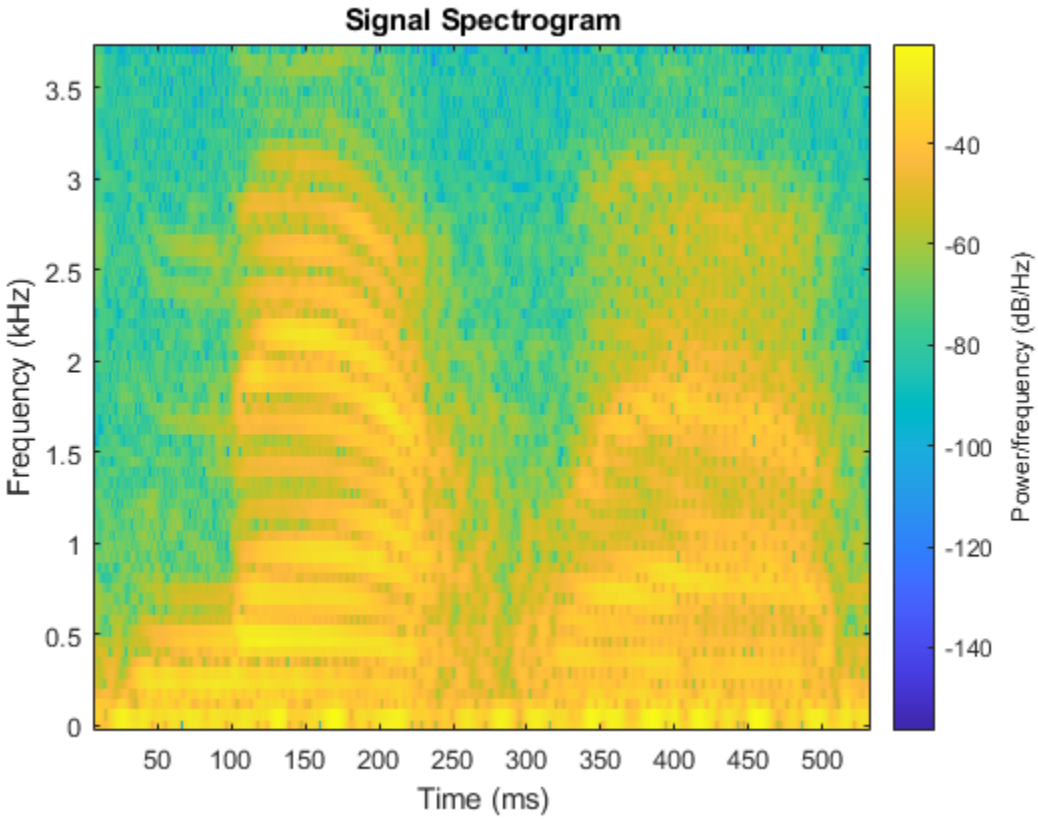
Load the speech signal. The recording is a woman saying "MATLAB®". The sampling frequency is 7418 Hz.

```
load mtlb
```

The MAT-file contains the speech waveform, `mtlb`, and the sampling frequency, `Fs`.

Use the `spectrogram` function to identify a voiced segment for analysis.

```
segmentlen = 100;
noverlap = 90;
NFFT = 128;

spectrogram(mtlb,segmentlen,noverlap,NFFT,Fs,'yaxis')
title('Signal Spectrogram')
```



Extract the segment from 0.1 to 0.25 seconds for analysis. The extracted segment corresponds roughly to the first vowel, /ae/, in "MATLAB".

```
dt = 1/Fs;
I0 = round(0.1/dt);
Iend = round(0.25/dt);
x = mtlb(I0:Iend);
```

Two common preprocessing steps applied to speech waveforms before linear predictive coding are windowing and pre-emphasis (highpass) filtering.

Window the speech segment using a Hamming window.

```
x1 = x.*hamming(length(x));
```

Apply a pre-emphasis filter. The pre-emphasis filter is a highpass all-pole (AR(1)) filter.

```
preemph = [1 0.63];
x1 = filter(1,preemph,x1);
```

Obtain the linear prediction coefficients. To specify the model order, use the general rule that the order is two times the expected number of formants plus 2. In the frequency range, [0,|Fs|/2], you expect three formants. Therefore, set the model order equal to 8. Find the roots of the prediction polynomial returned by `lpc`.

```
A = lpc(x1,8);
rts = roots(A);
```

Because the LPC coefficients are real-valued, the roots occur in complex conjugate pairs. Retain only the roots with one sign for the imaginary part and determine the angles corresponding to the roots.

```
rts = rts(imag(rts)>=0);
angz = atan2(imag(rts),real(rts));
```

Convert the angular frequencies in rad/sample represented by the angles to hertz and calculate the bandwidths of the formants.

The bandwidths of the formants are represented by the distance of the prediction polynomial zeros from the unit circle.

```
[frqs,indices] = sort(angz.*(Fs/(2*pi)));
bw = -1/2*(Fs/(2*pi))*log(abs(rts(indices)));
```

Use the criterion that formant frequencies should be greater than 90 Hz with bandwidths less than 400 Hz to determine the formants.

```
nn = 1;
for kk = 1:length(frqs)
    if (frqs(kk) > 90 && bw(kk) <400)
        formants(nn) = frqs(kk);
        nn = nn+1;
    end
end
formants
```

```
formants = 1×3
10^3 ×
```

```
    0.8697    2.0265    2.7380
```

The first three formants are 869.70, 2026.49, and 2737.95 Hz.

**References**

[1] Snell, Roy C., and Fausto Milinazzo. "Formant location from LPC analysis data." IEEE® Transactions on Speech and Audio Processing. Vol. 1, Number 2, 1993, pp. 129-134.

[2] Loizou, Philipos C. "COLEA: A MATLAB Software Tool for Speech Analysis."

# AR Order Selection with Partial Autocorrelation Sequence

This example shows how to assess the order of an autoregressive model using the partial autocorrelation sequence. For a stationary time series with values $X(1), X(2), X(3), \ldots, X(k+1)$, the partial autocorrelation sequence at lag $k$ is the correlation between $X(1)$ and $X(k+1)$ after regressing $X(1)$ and $X(k+1)$ on the intervening observations, $X(2), X(3), X(4), \ldots, X(k)$. For a moving average process, you can use the autocorrelation sequence to assess the order. However, for an autoregressive (AR) or autoregressive moving average (ARMA) process, the autocorrelation sequence does not help in order selection. This example uses the following workflow for model order selection in an AR process:

- Simulates a realization of the AR(2) process.
- Graphically explores the correlation between lagged values of the time series.
- Examines the sample autocorrelation sequence of the time series.
- Fits an AR(15) model to the time series by solving the Yule-Walker equations (`aryule`).
- Uses the reflection coefficients returned by `aryule` to compute the partial autocorrelation sequence.
- Examines the partial autocorrelation sequence to select the model order.

Consider the AR(2) process defined by

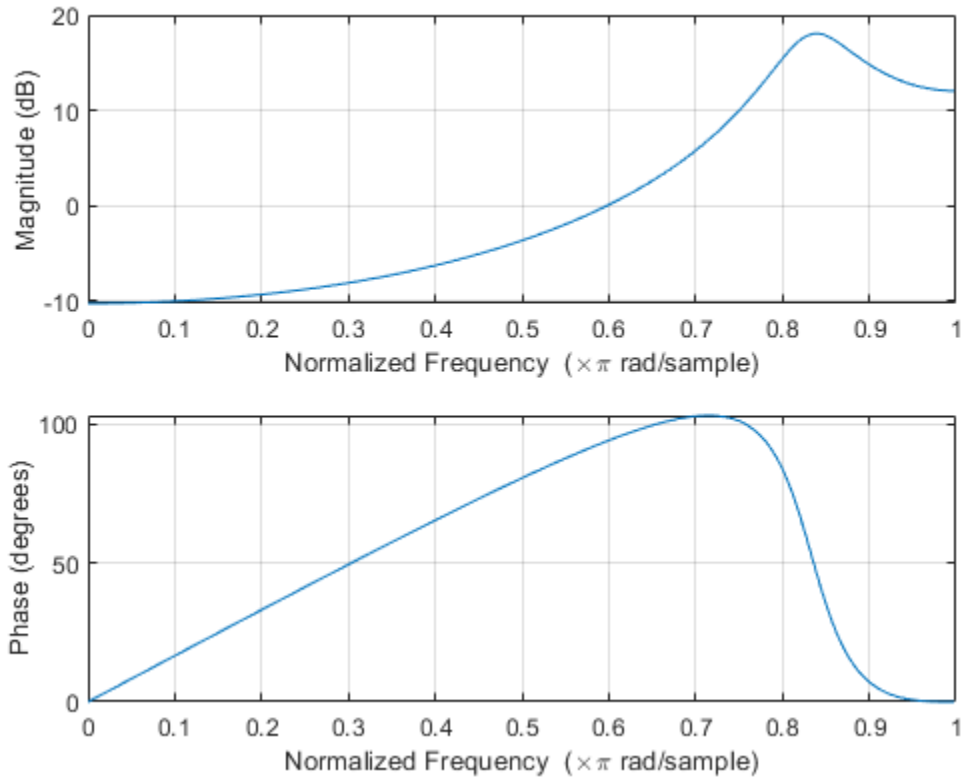
$$X(n) + 1.5X(n-1) + 0.75X(n-2) = \varepsilon(n),$$

where $\varepsilon(n)$ is an $N(0,1)$ Gaussian white noise process. Simulate a 1000-sample time series from the AR(2) process defined by the difference equation. Set the random number generator to the default settings for reproducible results.

```
A = [1 1.5 0.75];
rng default
x = filter(1,A,randn(1000,1));
```

View the frequency response of the AR(2) process.

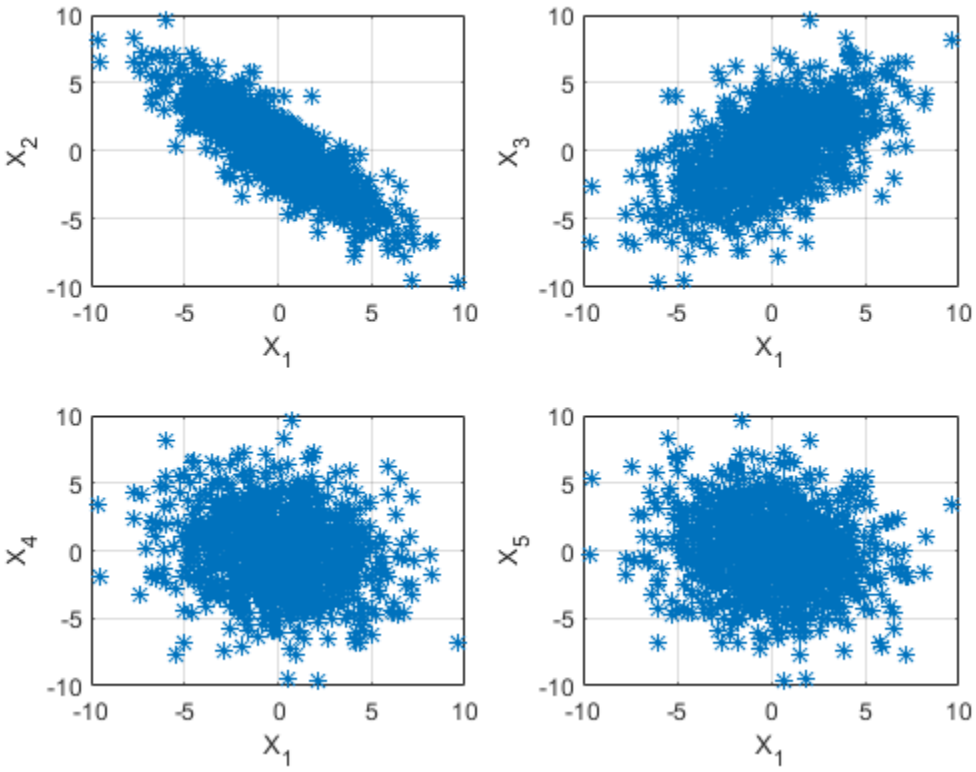
```
freqz(1,A)
```




The AR(2) process acts like a highpass filter in this case.

Graphically examine the correlation in x by producing scatter plots of $X(n+1)$ vs. $X(1)$ for $n = 2, 3, 4, 5$.

```
figure
for k = 1:4
    subplot(2,2,k)
    plot(x(1:end-k),x(k+1:end),'*')
    xlabel('X_1')
    ylabel(['X_' int2str(k+1)])
    grid
end
```



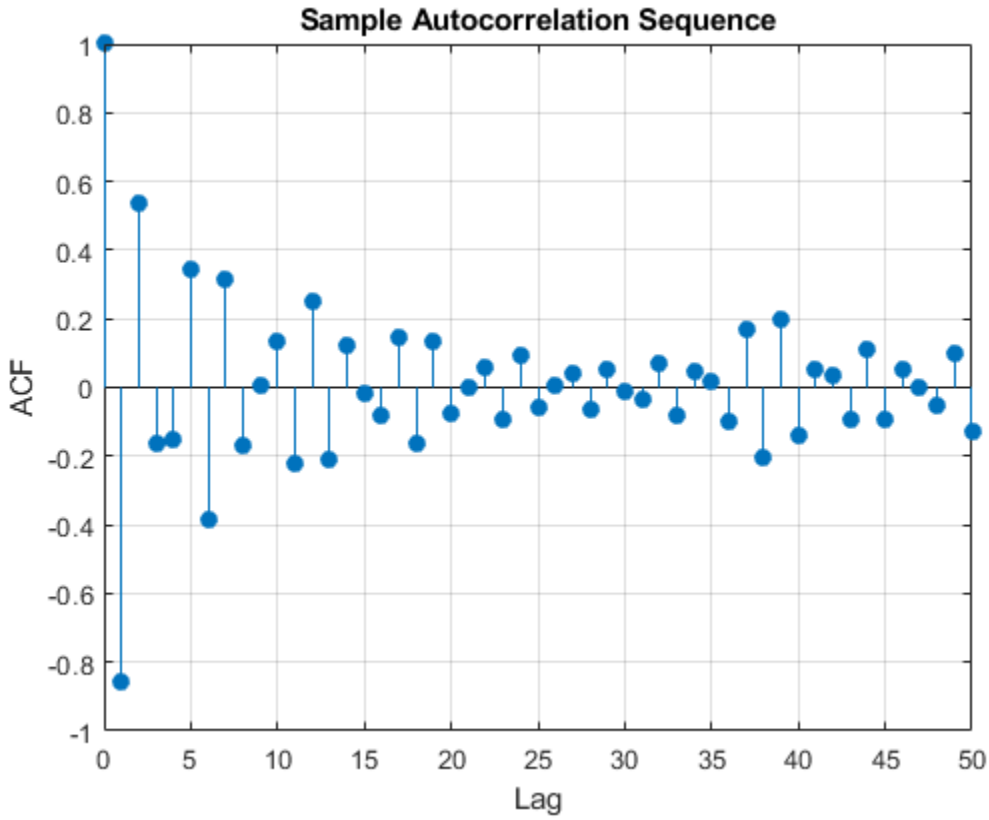

In the scatter plot, you see a linear relationship between $X(1)$ and $X(2)$ and between $X(1)$ and $X(3)$, but not between $X(1)$ and either $X(4)$ or $X(5)$.

The points in the top row scatter plots fall approximately on a line with a negative slope in the top left panel and positive slope in the top right panel. The scatter plots in the bottom two panels do not show any apparent linear relationship.

The negative correlation between $X(1)$ and $X(2)$ and the positive correlation between $X(1)$ and $X(3)$ are explained by the highpass-filter behavior of the AR(2) process.

Find the sample autocorrelation sequence up to lag 50 and plot the result.

```
[xc,lags] = xcorr(x,50,'coeff');

figure
stem(lags(51:end),xc(51:end),'filled')
xlabel('Lag')
ylabel('ACF')
title('Sample Autocorrelation Sequence')
grid
```



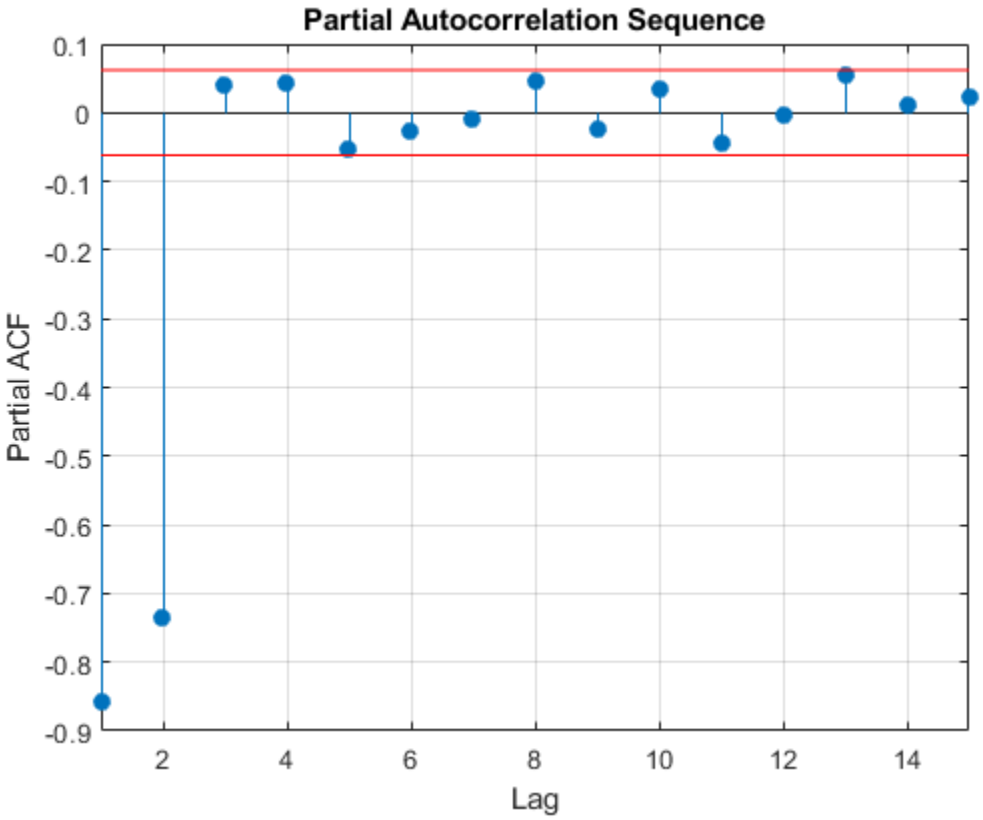

The sample autocorrelation sequence shows a negative value at lag 1 and a positive value at lag 2. Based on the scatter plot, this result is expected. However, you cannot determine the appropriate order for the AR model from the sample autocorrelation sequence.

Fit an AR(15) model using `aryule`. Return the sequence of reflection coefficients, whose negative is the partial autocorrelation sequence.

```
[arcoefs,E,K] = aryule(x,15);
pacf = -K;
```

Plot the partial autocorrelation sequence along with the large-sample 95% confidence intervals. If the data are generated by an autoregressive process of order $p$, the values of the sample partial autocorrelation sequence for lags greater than $p$ follow a $N(0, 1/N)$ distribution, where $N$ is the length of the time series. For a 95% confidence interval, the critical value is $\sqrt{2}\mathrm{erf}^{-1}(0.95) \approx 1.96$ and the confidence interval is $\Delta = 0 \pm 1.96/\sqrt{N}$.

```
stem(pacf,'filled')
xlabel('Lag')
ylabel('Partial ACF')
title('Partial Autocorrelation Sequence')
xlim([1 15])

conf = sqrt(2)*erfinv(0.95)/sqrt(1000);
hold on
plot(xlim,[1 1]'*[-conf conf],'r')
hold off
grid
```




The only values of the partial autocorrelation sequence outside the 95% confidence bounds occur at lags 1 and 2. This indicates that the correct model order for the AR process is 2.

In this example, you generated the time series to simulate an AR(2) process. The partial autocorrelation sequence only confirms that result. In practice, you have only the observed time series without any prior information about model order. In a realistic scenario, the partial autocorrelation sequence is an important tool for appropriate model order selection in stationary autoregressive time series.

## See Also

`aryule` | `xcorr`

# Transforms


- “Complex Cepstrum — Fundamental Frequency Estimation” on page 16-2
- “Analytic Signal for Cosine” on page 16-5
- “Envelope Extraction” on page 16-7
- “Analytic Signal and Hilbert Transform” on page 16-13
- “Hilbert Transform and Instantaneous Frequency” on page 16-18
- “Detect Closely Spaced Sinusoids” on page 16-25
- “Instantaneous Frequency of Complex Chirp” on page 16-32
- “Single-Sideband Amplitude Modulation” on page 16-35
- “DCT for Speech Signal Compression” on page 16-42

# Complex Cepstrum — Fundamental Frequency Estimation

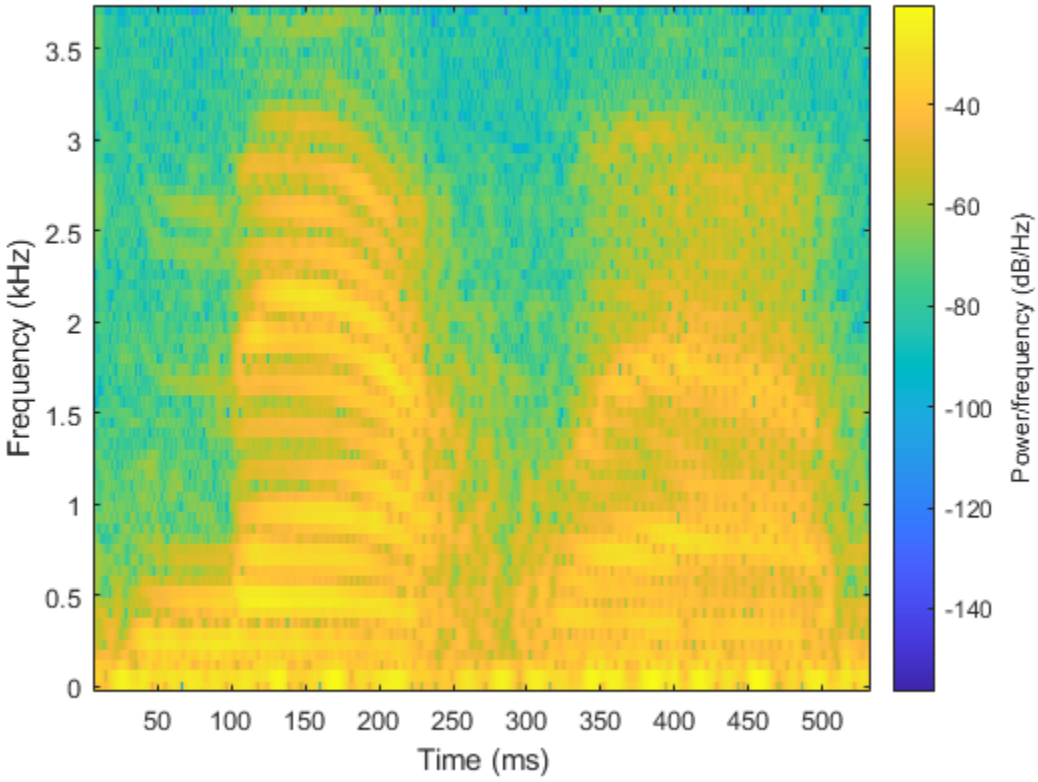
This example shows how to estimate a speaker's fundamental frequency using the complex cepstrum. The example also estimates the fundamental frequency using a zero-crossing method and compares the results.

Load the speech signal. The recording is of a woman saying "MATLAB". The sampling frequency is 7418 Hz. The following code loads the speech waveform, `mtlb`, and the sampling frequency, `Fs`, into the MATLAB® workspace.

```
load mtlb
```

Use the spectrogram to identify a voiced segment for analysis.

```
segmentlen = 100;
noverlap = 90;
NFFT = 128;

spectrogram(mtlb,segmentlen,noverlap,NFFT,Fs,'yaxis')
```




Extract the segment from 0.1 to 0.25 seconds for analysis. The extracted segment corresponds roughly to the first vowel, /æ/, in "MATLAB".

```
dt = 1/Fs;
I0 = round(0.1/dt);
Iend = round(0.25/dt);
x = mtlb(I0:Iend);
```

Obtain the complex cepstrum.

```
c = cceps(x);
```

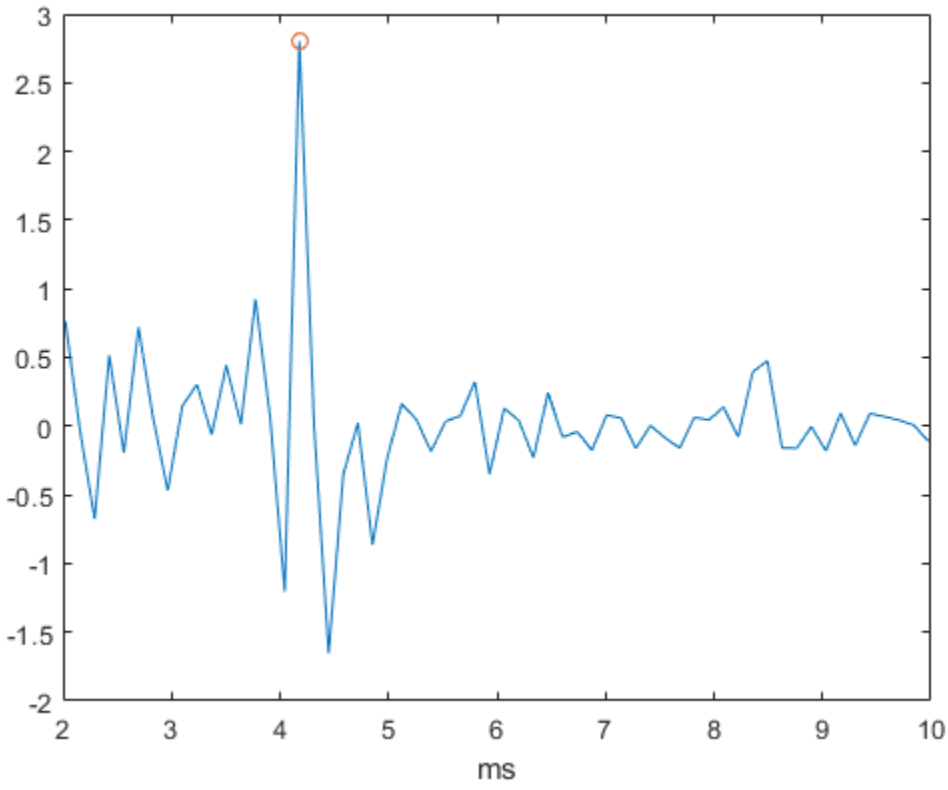
Select a time range between 2 and 10 ms, corresponding to a frequency range of approximately 100 to 500 Hz. Identify the tallest peak of the cepstrum in the selected range. Find the frequency corresponding to the peak. Use the peak as the estimate of the fundamental frequency.

```
t = 0:dt:length(x)*dt-dt;

trng = t(t>=2e-3 & t<=10e-3);
crng = c(t>=2e-3 & t<=10e-3);

[~,I] = max(crng);

fprintf('Complex cepstrum F0 estimate is %3.2f Hz.\n',1/trng(I))
```

```
Complex cepstrum F0 estimate is 239.29 Hz.
```

Plot the cepstrum in the selected time range and overlay the peak.

```
plot(trng*1e3,crng)
xlabel('ms')

hold on
plot(trng(I)*1e3,crng(I),'o')
hold off
```



Use a zero-crossing detector on a lowpass-filtered and rectified form of the vowel to estimate the fundamental frequency.

```
[b0,a0] = butter(2,325/(Fs/2));
xin = abs(x);
xin = filter(b0,a0,xin);
xin = xin-mean(xin);
x2 = zeros(length(xin),1);
x2(1:length(x)-1) = xin(2:length(x));
zc = length(find((xin>0 & x2<0) | (xin<0 & x2>0)));
F0 = 0.5*Fs*zc/length(x);
fprintf('Zero-crossing F0 estimate is %3.2f Hz.\n',F0)
```

```
Zero-crossing F0 estimate is 233.27 Hz.
```

The estimate of the fundamental frequency obtained with the complex cepstrum is 239.29 Hz and the estimate with the zero-crossing detector is 233.27 Hz.

## See Also

`cceps` | `icceps` | `rceps`

# Analytic Signal for Cosine

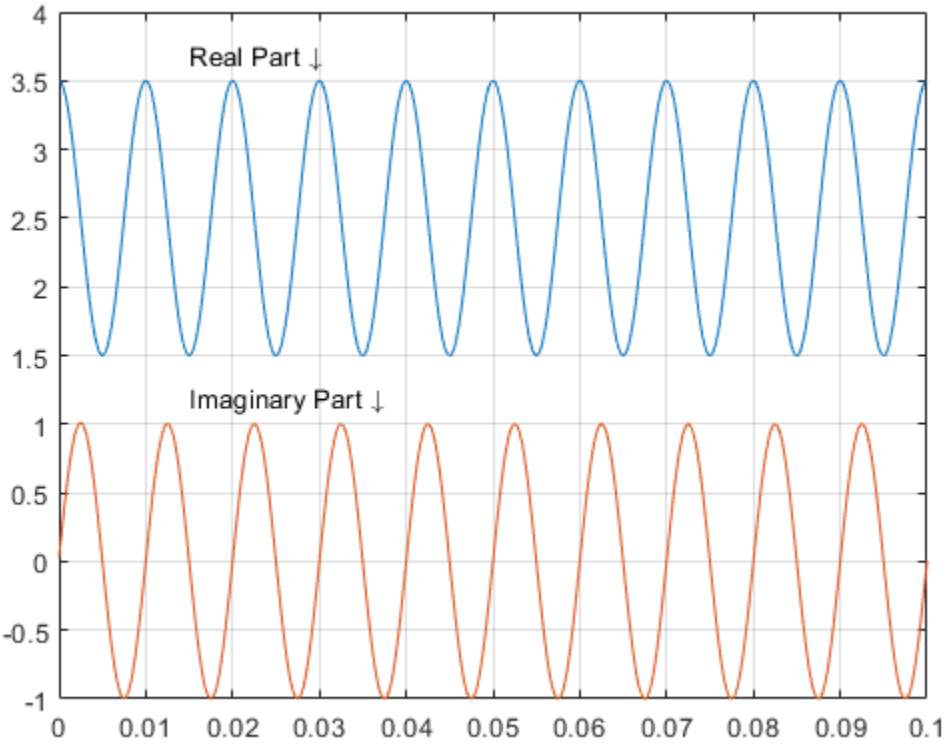
This example shows how to determine the analytic signal. The example also demonstrates that the imaginary part of the analytic signal corresponding to a cosine is a sine with the same frequency. If the cosine has a nonzero mean (DC shift), then the real part of the analytic signal is the original cosine with the same mean, but the imaginary part has zero mean.

Create a cosine with a frequency of 100 Hz. The sample rate is 10 kHz. Add a DC offset of 2.5 to the cosine.

```
t = 0:1e-4:1;
x = 2.5 + cos(2*pi*100*t);
```

Use the `hilbert` function to obtain the analytic signal. The real part is equal to the original signal. The imaginary part is the Hilbert transform of the original signal. Plot the real and imaginary parts for comparison.

```
y = hilbert(x);

plot(t,real(y))
hold on
plot(t,imag(y))
xlim([0 0.1])
grid on
text([0.015 0.015],[3.7 1.2], ...
    {'Real Part \downarrow';'Imaginary Part \downarrow'})
```



You see that the imaginary part is a sine with the same frequency as the cosine real part. However, the imaginary part has a mean of zero, while the real part has a mean of 2.5.

The original signal is

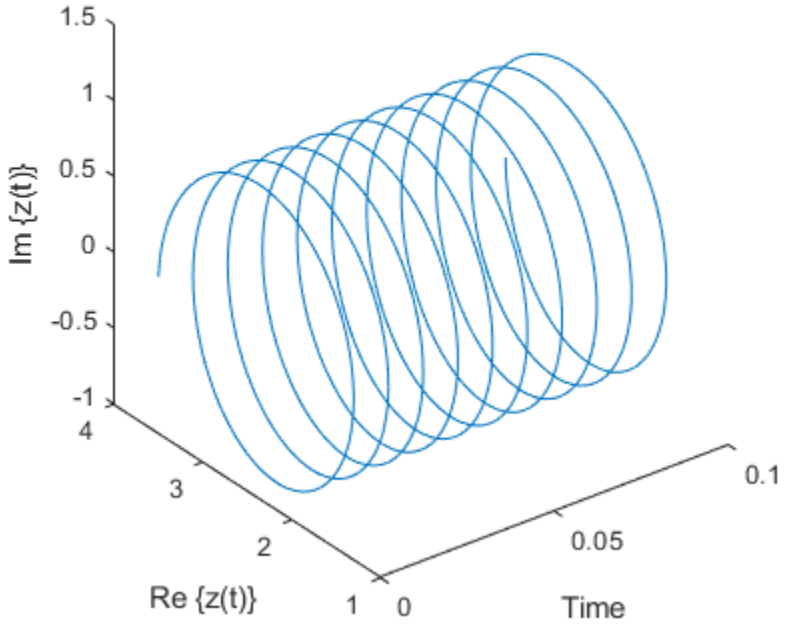
$$x(t) = 2.5 + \cos(2\pi 1000t).$$

The resulting analytic signal is

$$z(t) = 2.5 + e^{j2\pi 1000t}.$$

Plot 10 periods of the complex-valued analytic signal.

```
prds = 1:1000;

figure
plot3(t(prds),real(y(prds)),imag(y(prds)))

xlabel('Time')
ylabel('Re \{z(t)\}')
zlabel('Im \{z(t)\}')
axis square
```




## See Also

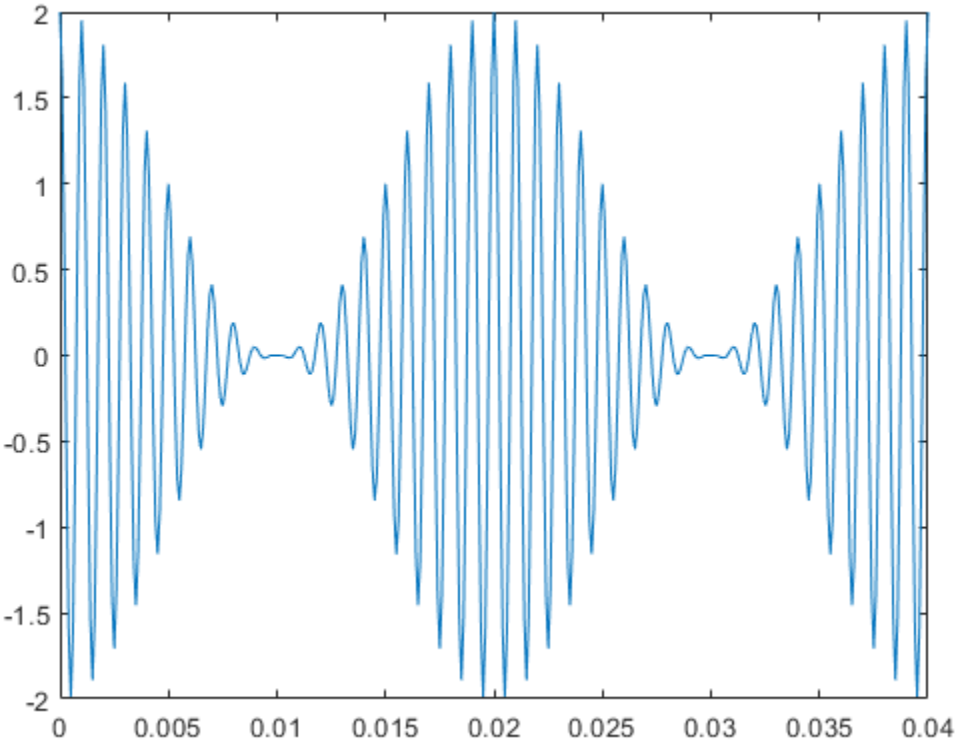
`hilbert`

# Envelope Extraction

This example shows how to extract the envelope of a signal.

Create a double sideband amplitude-modulated signal. The carrier frequency is 1 kHz. The modulation frequency is 50 Hz. The modulation depth is 100%. The sample rate is 10 kHz.

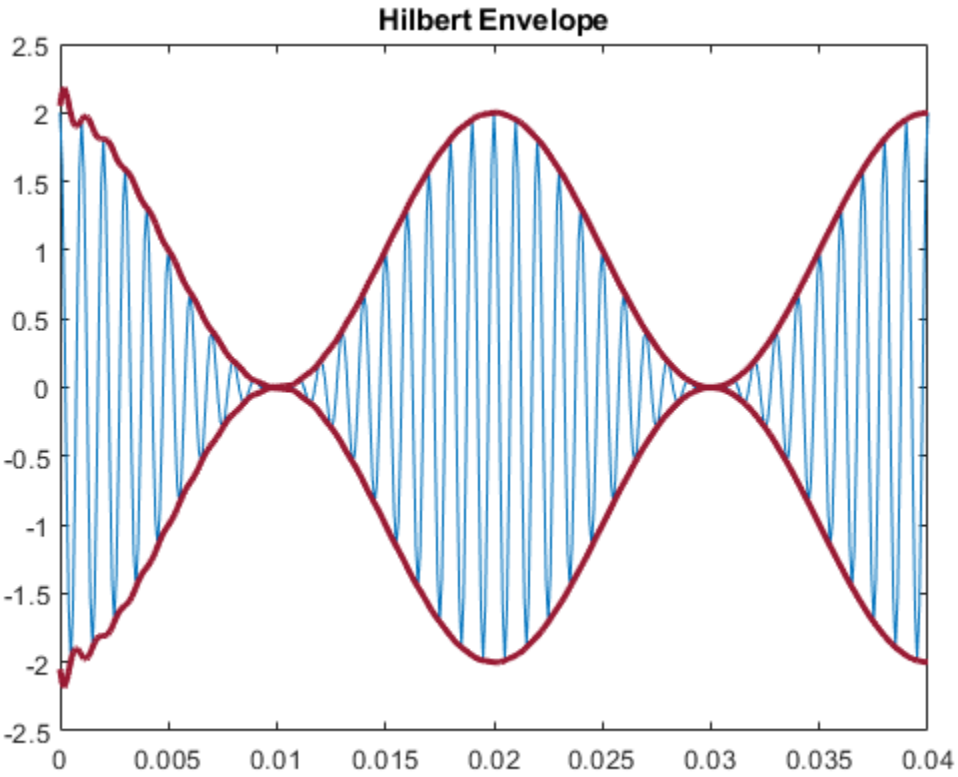
```
t = 0:1e-4:0.1;
x = (1+cos(2*pi*50*t)).*cos(2*pi*1000*t);

plot(t,x)
xlim([0 0.04])
```




Extract the envelope using the `hilbert` function. The envelope is the magnitude of the analytic signal computed by `hilbert`. Plot the envelope along with the original signal. Store the name-value pair arguments of the `plot` function in a cell array for later use. The magnitude of the analytic signal captures the slowly varying features of the signal, while the phase contains the high-frequency information.

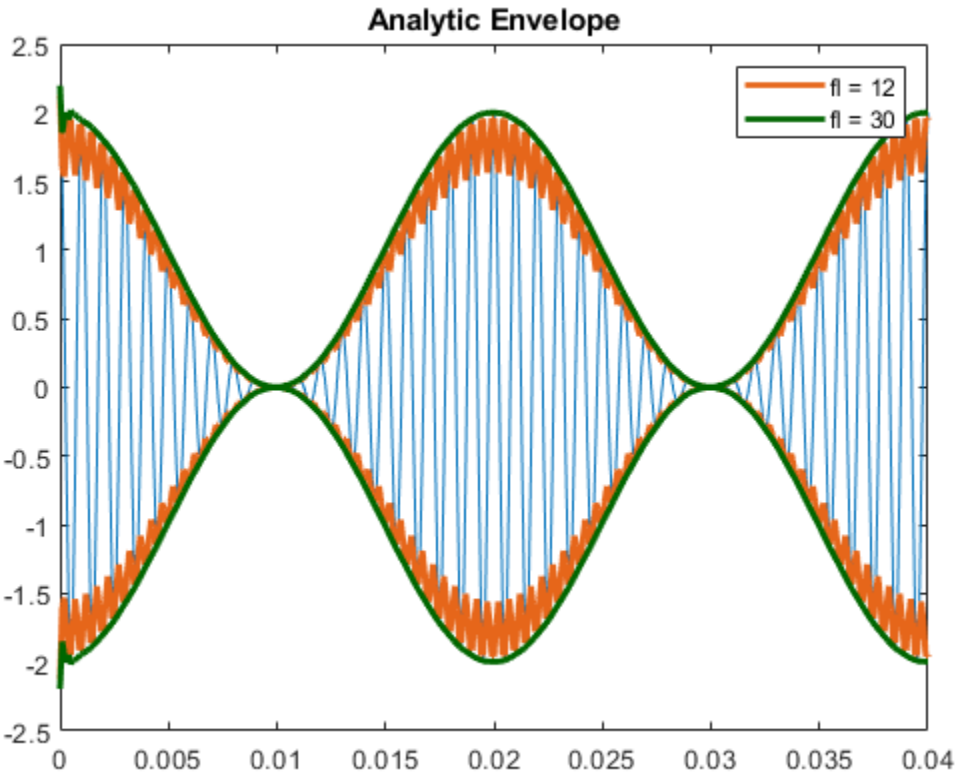
```
y = hilbert(x);
env = abs(y);
plot_param = {'Color', [0.6 0.1 0.2],'Linewidth',2};

plot(t,x)
hold on
plot(t,[-1;1]*env,plot_param{:})
```

```
hold off
xlim([0 0.04])
title('Hilbert Envelope')
```



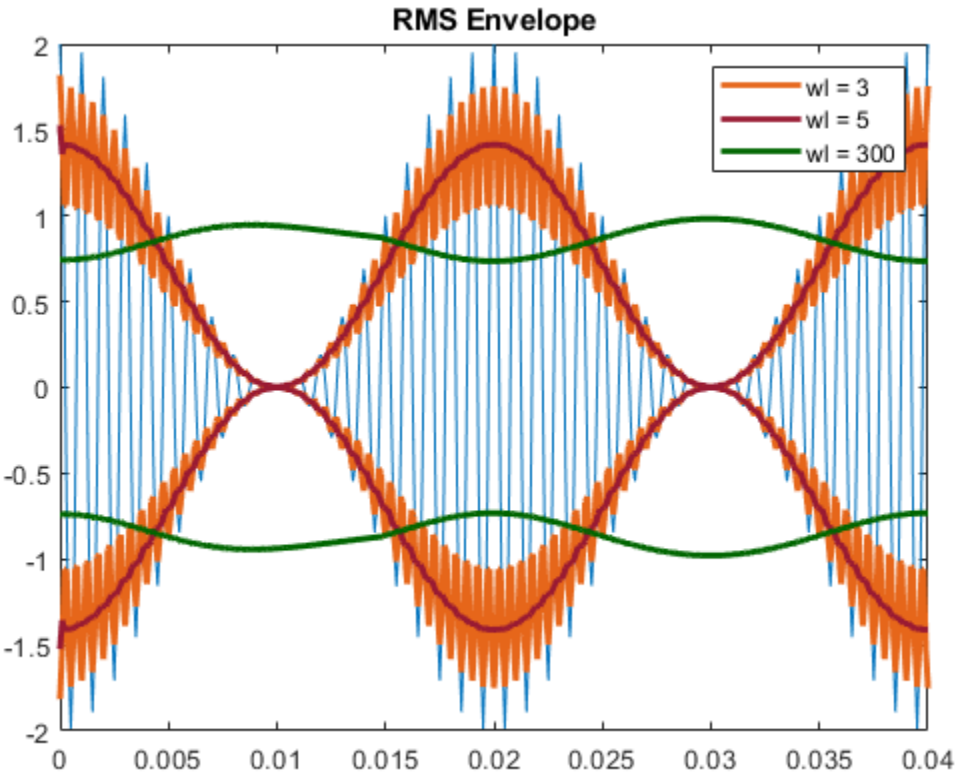

You can also use the `envelope` function to generate the signal envelope directly and modify the way it is computed. For example, you can adjust the length of the Hilbert filter used to find the analytic envelope. Using a filter length that is too small results in a distorted envelope.

```
fl1 = 12;
[up1,lo1] = envelope(x,fl1,'analytic');
fl2 = 30;
[up2,lo2] = envelope(x,fl2,'analytic');
param_small = {'Color',[0.9 0.4 0.1],'Linewidth',2};
param_large = {'Color',[0 0.4 0],'Linewidth',2};

plot(t,x)
hold on
p1 = plot(t,up1,param_small{:});
plot(t,lo1,param_small{:});
p2 = plot(t,up2,param_large{:});
plot(t,lo2,param_large{:});
hold off

legend([p1 p2],'fl = 12','fl = 30')
xlim([0 0.04])
title('Analytic Envelope')
```



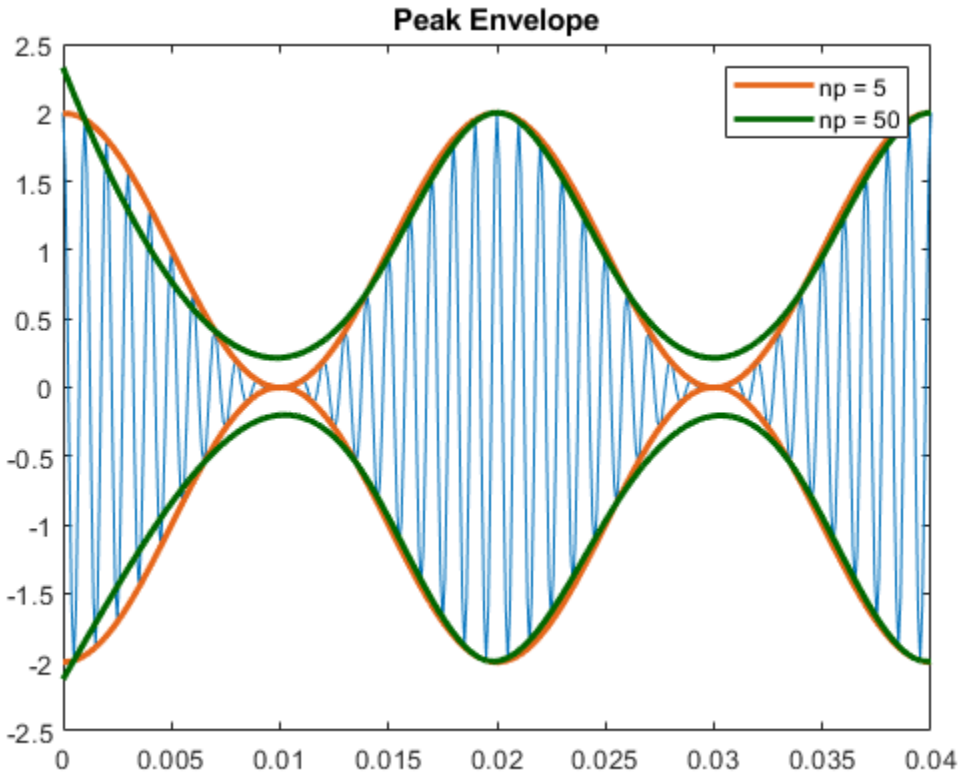

You can generate moving RMS envelopes by using a sliding window. Using a window length that is too small results in a distorted envelope. Using a window length that is too large smooths out the envelope.

```
wl1 = 3;
[up1,lo1] = envelope(x,wl1,'rms');
wl2 = 5;
[up2,lo2] = envelope(x,wl2,'rms');
wl3 = 300;
[up3,lo3] = envelope(x,wl3,'rms');

plot(t,x)
hold on
p1 = plot(t,up1,param_small{:});
plot(t,lo1,param_small{:});
p2 = plot(t,up2,plot_param{:});
plot(t,lo2,plot_param{:});
p3 = plot(t,up3,param_large{:});
plot(t,lo3,param_large{:})
hold off

legend([p1 p2 p3],'wl = 3','wl = 5','wl = 300')
xlim([0 0.04])
title('RMS Envelope')
```



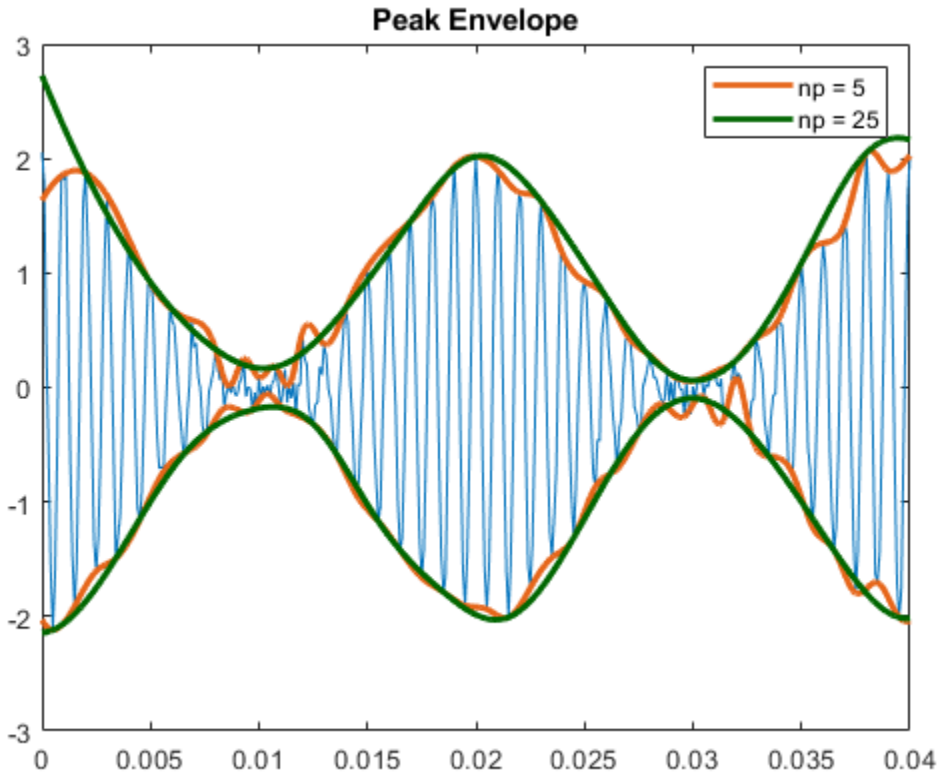

You can generate peak envelopes by using spline interpolation over local maxima separated by an adjustable number of samples. Spreading out the samples too much smooths the envelope.

```
np1 = 5;
[up1,lo1] = envelope(x,np1,'peak');
np2 = 50;
[up2,lo2] = envelope(x,np2,'peak');

plot(t,x)
hold on
p1 = plot(t,up1,param_small{:});
plot(t,lo1,param_small{:})
p2 = plot(t,up2,param_large{:});
plot(t,lo2,param_large{:})
hold off

legend([p1 p2],'np = 5','np = 50')
xlim([0 0.04])
title('Peak Envelope')
```




Increasing the peak separation parameter can decrease the effect of spurious peaks due to noise. Introduce random noise to the signal. Use a 5-sample interval to see the effect of noise on the peak envelope. Repeat the exercise using a 25-sample interval.

```
rng default
q = x + randn(size(x))/10;
np1 = 5;
[up1,lo1] = envelope(q,np1,'peak');
np2 = 25;
[up2,lo2] = envelope(q,np2,'peak');

plot(t,q)
hold on
p1 = plot(t,up1,param_small{:});
plot(t,lo1,param_small{:})
p2 = plot(t,up2,param_large{:});
plot(t,lo2,param_large{:})
hold off

legend([p1 p2],'np = 5','np = 25')
xlim([0 0.04])
title('Peak Envelope')
```




## See Also

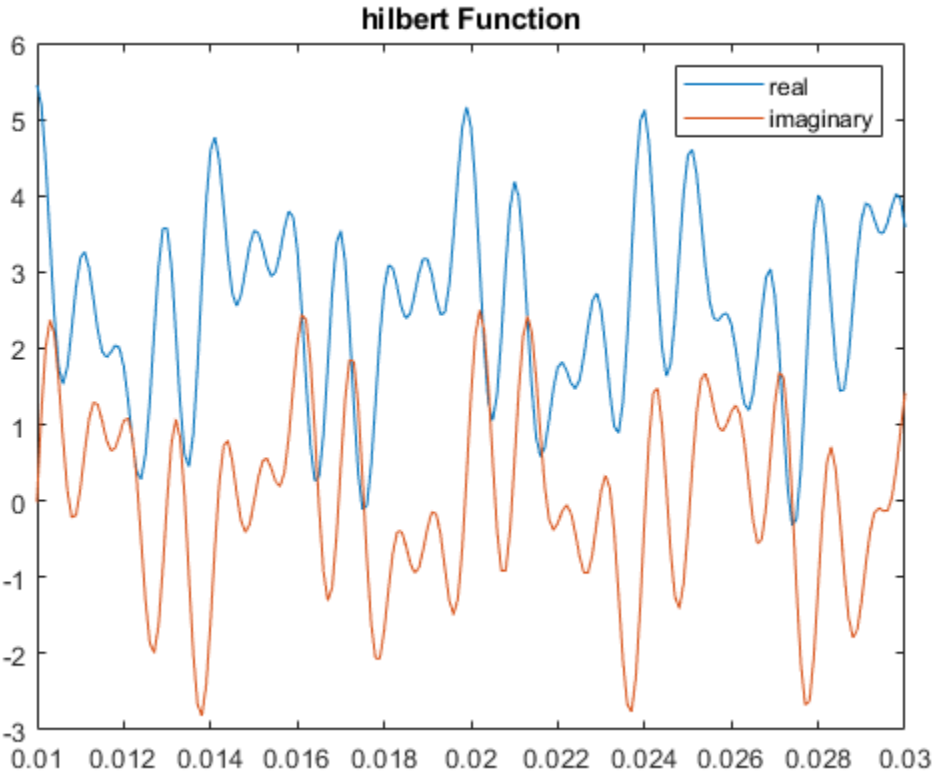
`envelope` | `hilbert`

# Analytic Signal and Hilbert Transform

The `hilbert` function finds the exact analytic signal for a finite block of data. You can also generate the analytic signal by using an finite impulse response (FIR) Hilbert transformer filter to compute an approximation to the imaginary part.

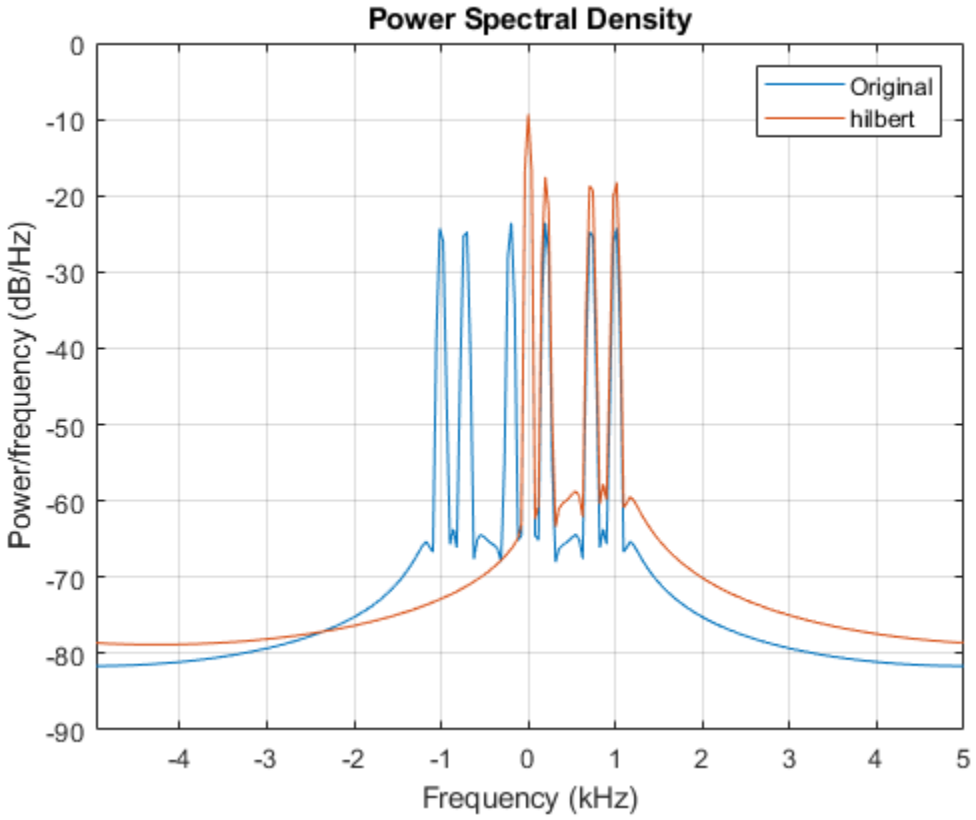
Generate a sequence composed of three sinusoids with frequencies 203, 721, and 1001 Hz. The sequence is sampled at 10 kHz for about 1 second. Use the `hilbert` function to compute the analytic signal. Plot it between 0.01 seconds and 0.03 seconds.

```
fs = 1e4;
t = 0:1/fs:1;

x = 2.5 + cos(2*pi*203*t) + sin(2*pi*721*t) + cos(2*pi*1001*t);

y = hilbert(x);

plot(t,real(y),t,imag(y))
xlim([0.01 0.03])
legend('real','imaginary')
title('hilbert Function')
```



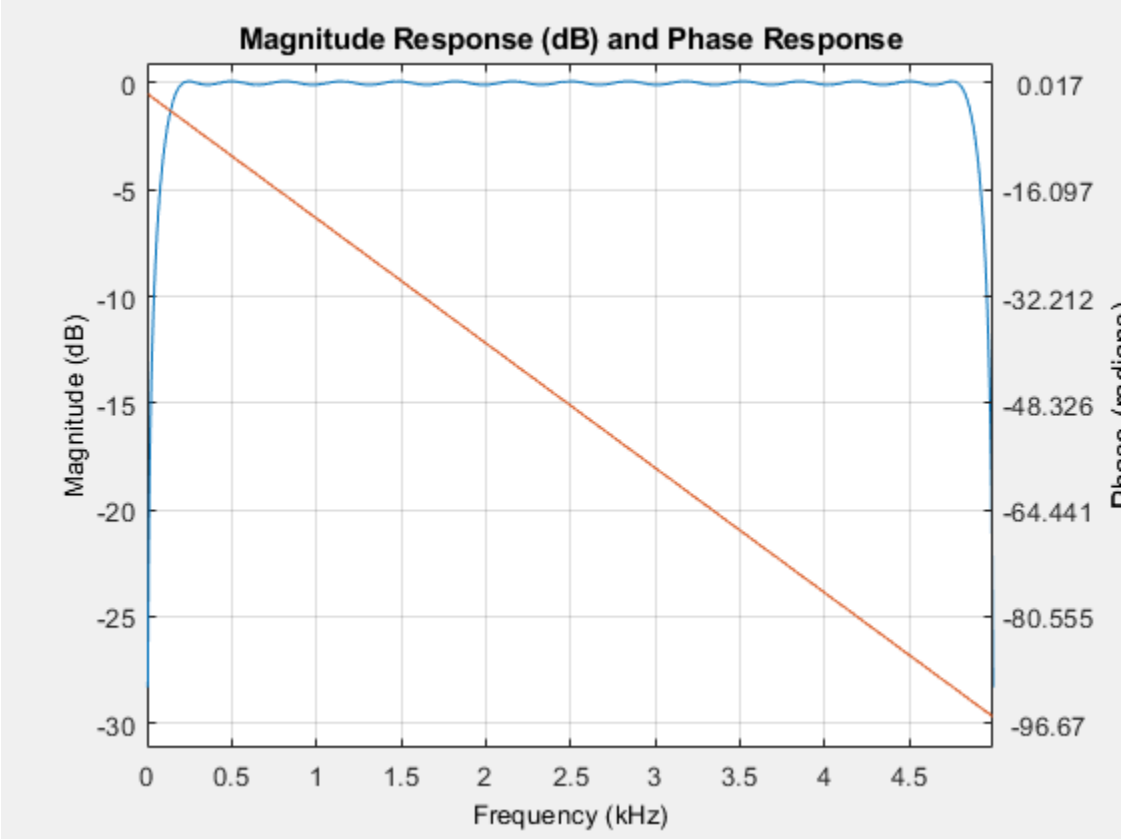

Compute Welch estimates of the power spectral densities of the original sequence and the analytic signal. Divide the sequences into Hamming-windowed, nonoverlapping sections of length 256. Verify that the analytic signal has no power at negative frequencies.

```
pwelch([x;y].',256,0,[],fs,'centered')
legend('Original','hilbert')
```




Use the `designfilt` function to design a 60th-order Hilbert transformer FIR filter. Specify a transition width of 400 Hz. Visualize the frequency response of the filter.

```
fo = 60;

d = designfilt('hilbertfir','FilterOrder',fo, ...
       'TransitionWidth',400,'SampleRate',fs);

freqz(d,1024,fs)
```

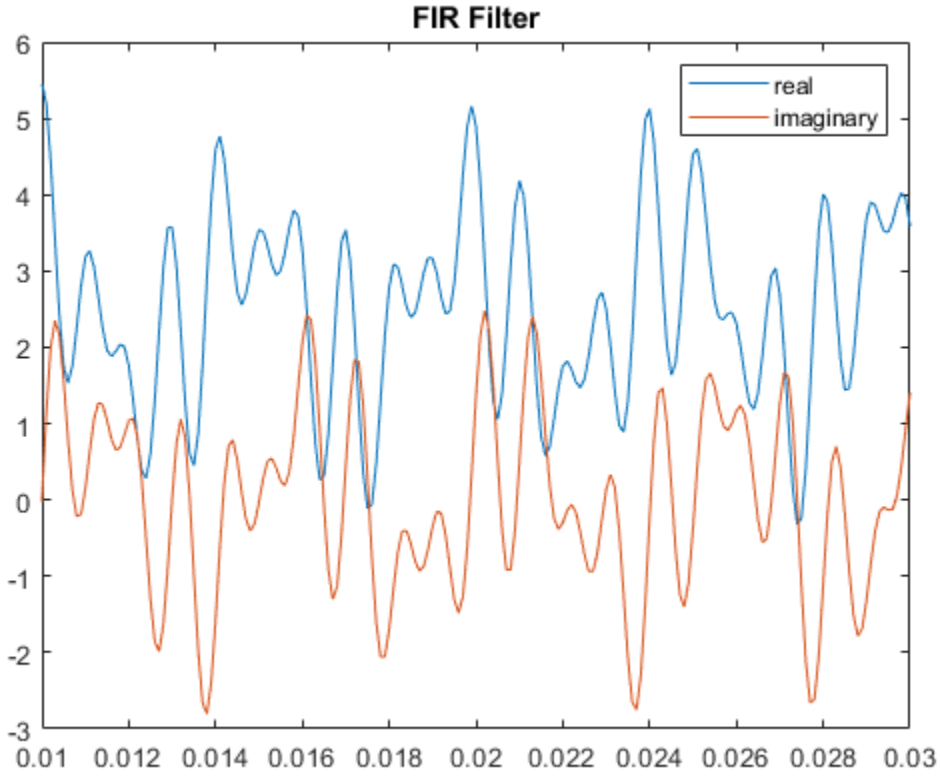



Filter the sinusoidal sequence to approximate the imaginary part of the analytic signal.

```
hb = filter(d,x);
```

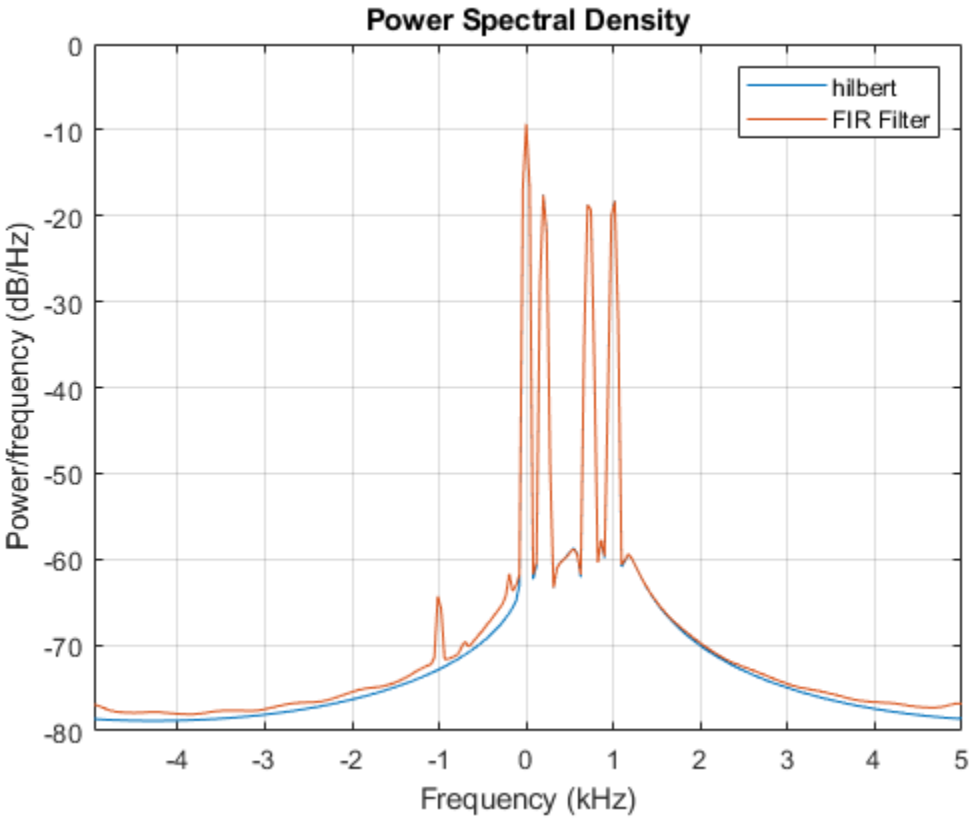
The group delay of the filter, `grd`, is equal to one-half the filter order. Compensate for this delay. Remove the first `grd` samples of the imaginary part and the last `grd` samples of the real part and the time vector. Plot the result between 0.01 seconds and 0.03 seconds.

```
grd = fo/2;

y2 = x(1:end-grd) + 1j*hb(grd+1:end);
t2 = t(1:end-grd);

plot(t2,real(y2),t2,imag(y2))
xlim([0.01 0.03])
legend('real','imaginary')
title('FIR Filter')
```




Estimate the power spectral density (PSD) of the approximate analytic signal and compare it to the `hilbert` result.

```
pwelch([y;[y2 zeros(1,grd)]].',256,0,[],fs,'centered')
legend('hilbert','FIR Filter')
```




## See Also

`designfilt` | `hilbert`

# Hilbert Transform and Instantaneous Frequency

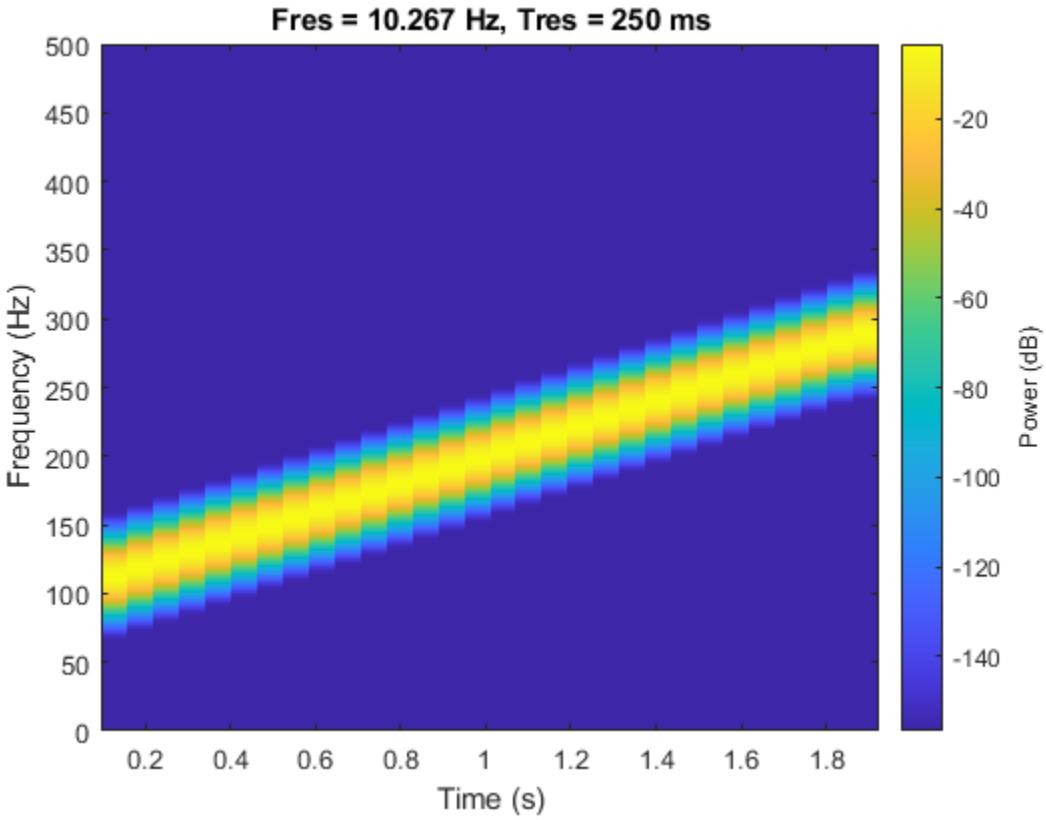
The Hilbert transform estimates the instantaneous frequency of a signal for monocomponent signals only. A monocomponent signal is described in the time-frequency plane by a single "ridge." The set of monocomponent signals includes single sinusoids and signals like chirps.

Generate a chirp sampled at 1 kHz for two seconds. Specify the chirp so its frequency is initially 100 Hz and increases to 200 Hz after one second.

```
fs = 1000;
t = 0:1/fs:2-1/fs;
y = chirp(t,100,1,200);
```

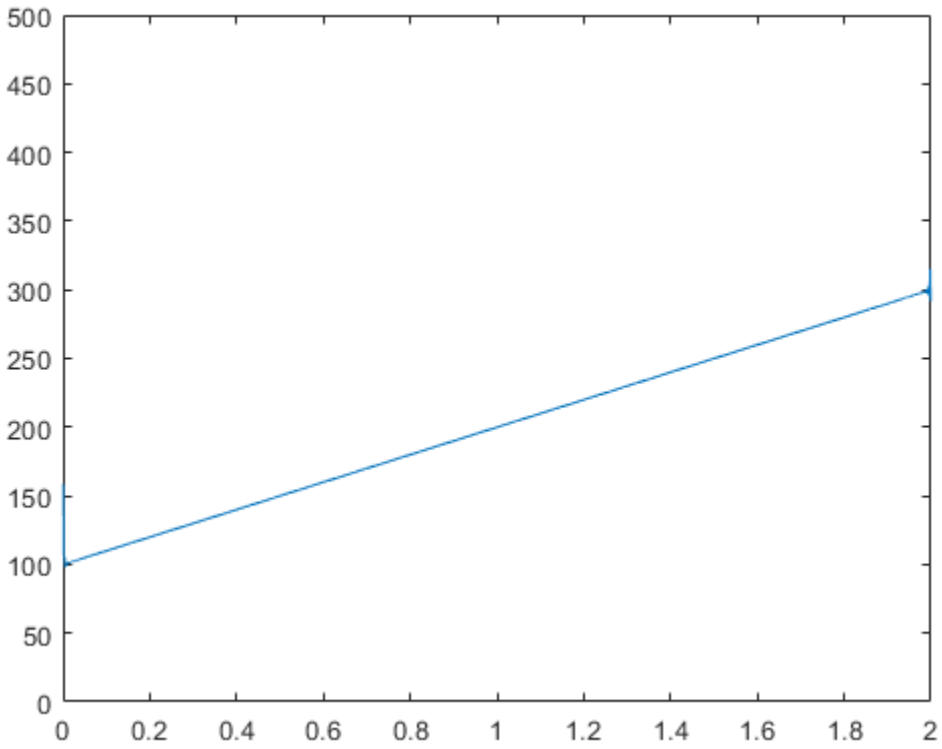
Estimate the spectrogram of the chirp using the short-time Fourier transform implemented in the `pspectrum` function. The signal is well described by a single peak frequency at each point in time.

```
pspectrum(y,fs,'spectrogram')
```



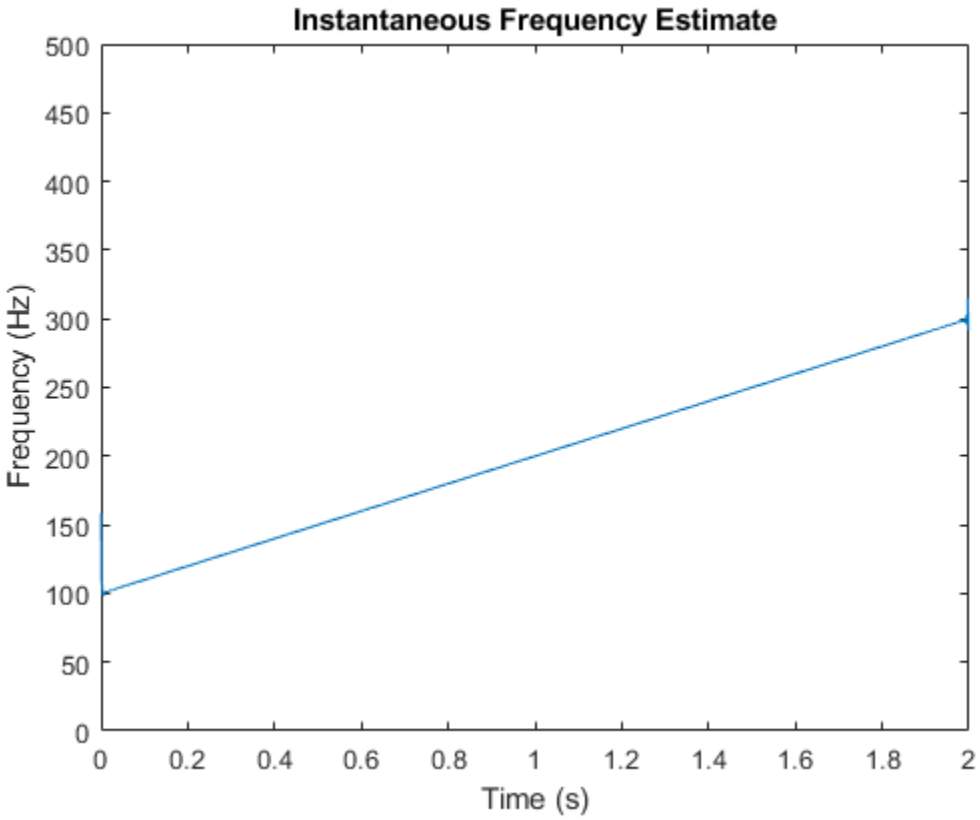

Compute the analytic signal and differentiate its phase to measure the instantaneous frequency. The scaled derivative yields a meaningful estimate.

```
z = hilbert(y);
instfrq = fs/(2*pi)*diff(unwrap(angle(z)));

clf
plot(t(2:end),instfrq)
ylim([0 fs/2])
```




The `instfreq` function computes and displays the instantaneous frequency in one step.

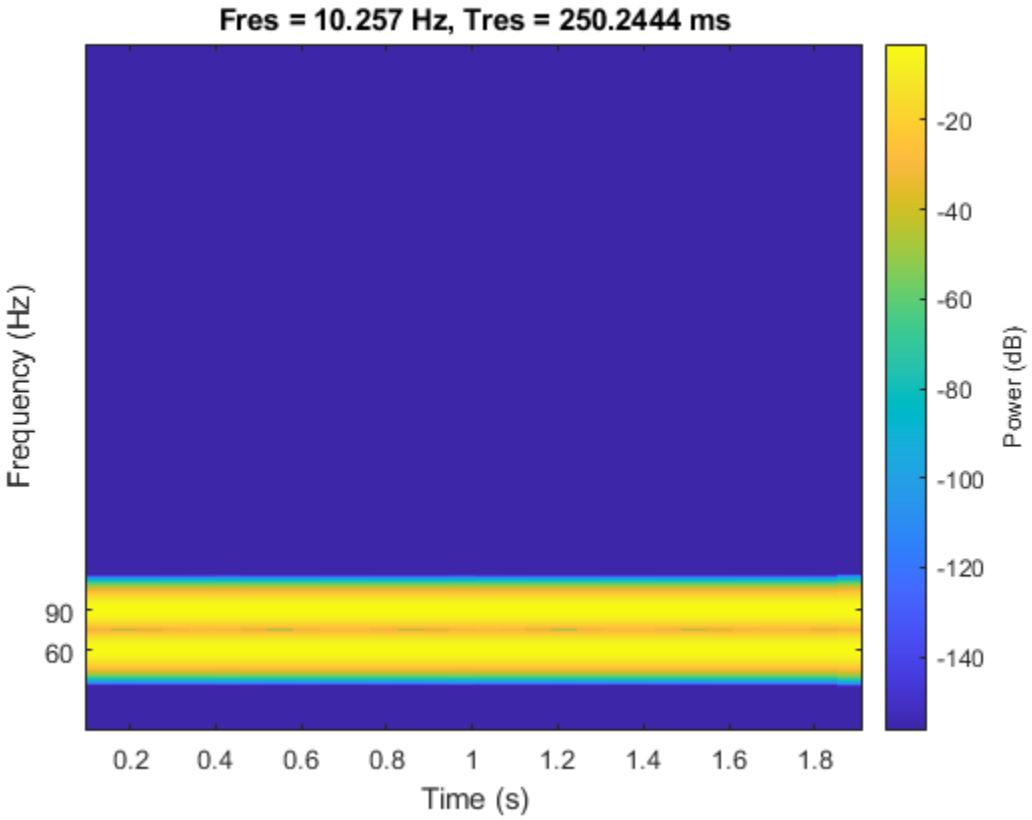
```
instfreq(y,fs,'Method','hilbert')
```




The method fails when the signal is not monocomponent.

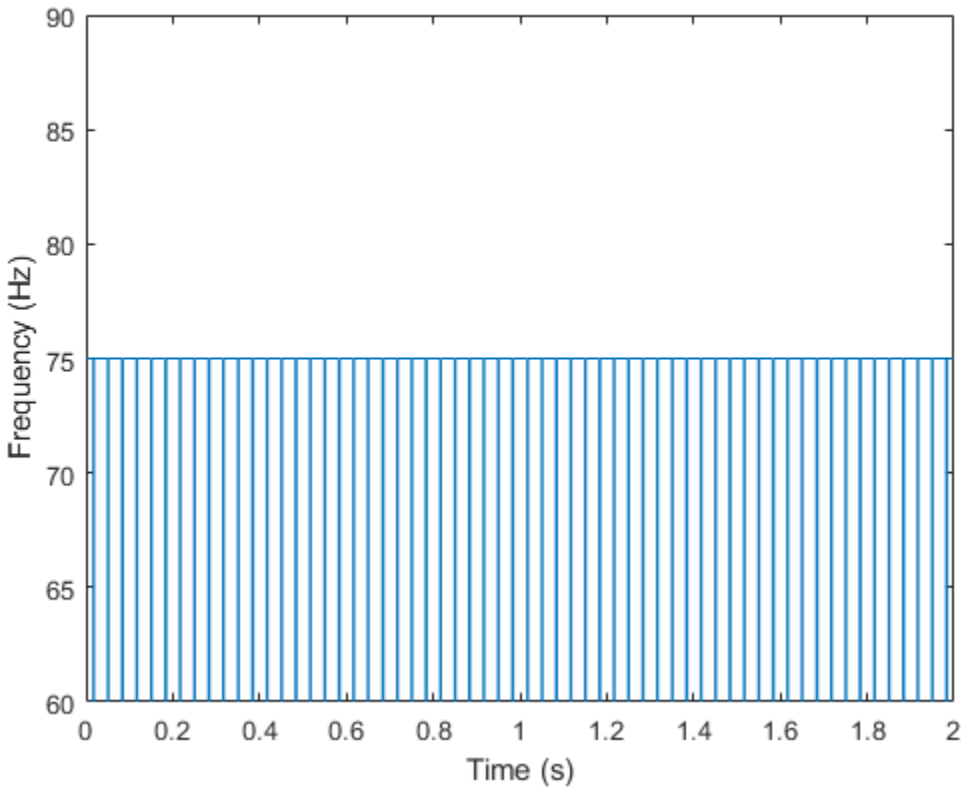
Generate a sum of two sinusoids of frequencies 60 Hz and 90 Hz, sampled at 1023 Hz for two seconds. Compute and plot the spectrogram. Each time point shows the presence of the two components.

```
fs = 1023;
t = 0:1/fs:2-1/fs;
x = sin(2*pi*60*t)+sin(2*pi*90*t);

pspectrum(x,fs,'spectrogram')
yticks([60 90])
```



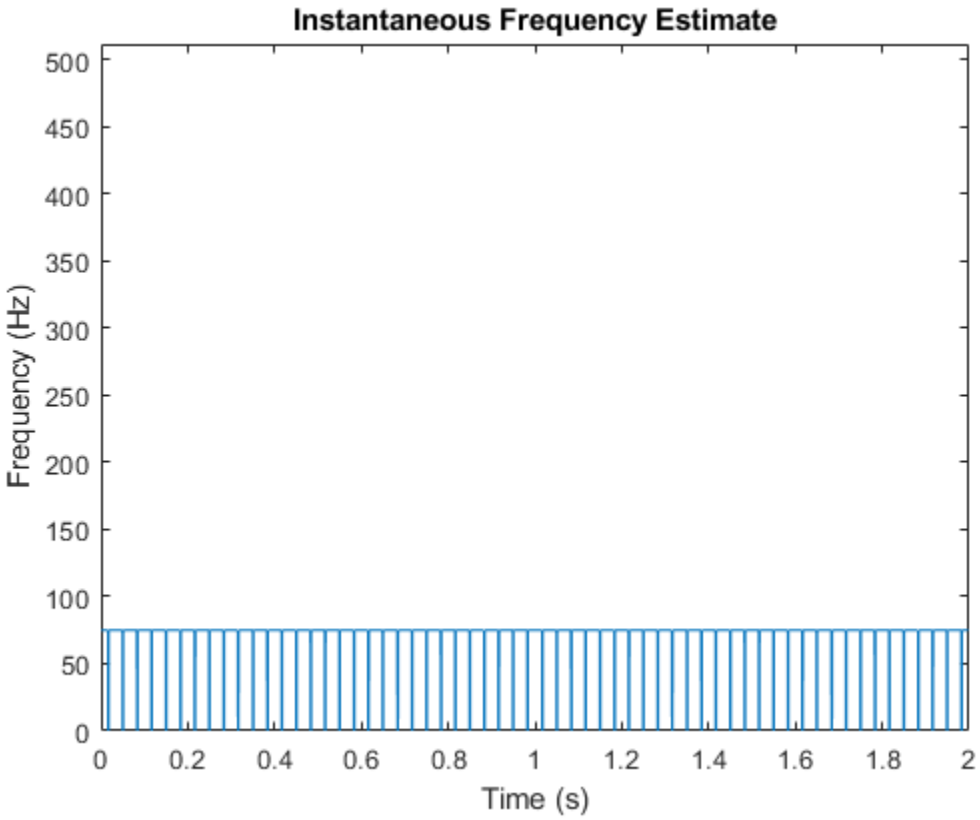

Compute the analytic signal and differentiate its phase. Zoom in on the region enclosing the frequencies of the sinusoids. The analytic signal predicts an instantaneous frequency that is the average of the sinusoid frequencies.

```
z = hilbert(x);
instfrq = fs/(2*pi)*diff(unwrap(angle(z)));

plot(t(2:end),instfrq)
ylim([60 90])
xlabel('Time (s)')
ylabel('Frequency (Hz)')
```



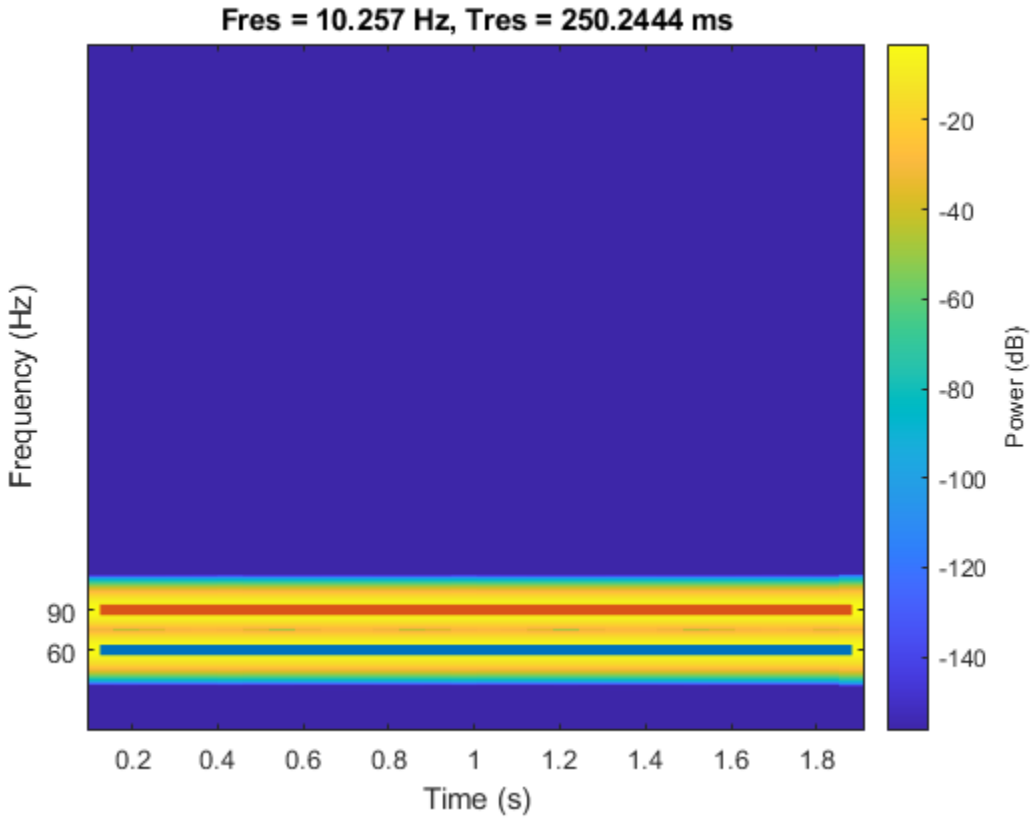

The `instfreq` function also estimates the average.

```
instfreq(x,fs,'Method','hilbert')
```




To estimate both frequencies as functions of time, use `spectrogram` to find the power spectral density and `tfridge` to track the two ridges. In `tfridge`, specify the penalty for changing frequency as 0.1.

```
[s,f,tt] = pspectrum(x,fs,'spectrogram');

numcomp = 2;
[fridge,~,lr] = tfridge(s,f,0.1,'NumRidges',numcomp);

pspectrum(x,fs,'spectrogram')
hold on
plot3(tt,fridge,abs(s(lr)),'LineWidth',4)
hold off
yticks([60 90])
```




## See Also

`hilbert` | `spectrogram`

## Related Examples

- “Detect Closely Spaced Sinusoids” on page 16-25

# Detect Closely Spaced Sinusoids

Consider a sinusoid, $f(x) = e^{j2\pi\nu x}$, windowed with a Gaussian window, $g(t) = e^{-\pi t^2}$. The short-time transform is

$$V_g f(t,\eta) = e^{j2\pi\nu t}\int_{\infty}^{\infty} e^{-\pi(x-t)^2} e^{-j2\pi(x-t)(\eta-\nu)}\,dx = e^{-\pi(\eta-\nu)^2} e^{j2\pi\nu t}.$$

When viewed as a function of frequency, the transform combines a constant (in time) oscillation at $\nu$ with Gaussian decay away from it. The synchrosqueezing estimate of the instantaneous frequency,

$$\Omega_g f(t,\eta) = \frac{1}{j2\pi}\frac{e^{-\pi(\eta-\nu)^2}\frac{\partial}{\partial t}e^{j2\pi\nu t}}{e^{-\pi(\eta-\nu)^2}e^{j2\pi\nu t}} = \nu,$$

equals the value obtained by using the standard definition, $(2\pi)^{-1} d\arg f(x)/dx$. For a superposition of sinusoids,

$$f(x) = \sum_{k=1}^{K} A_k e^{j2\pi\nu_k x},$$

the short-time transform becomes

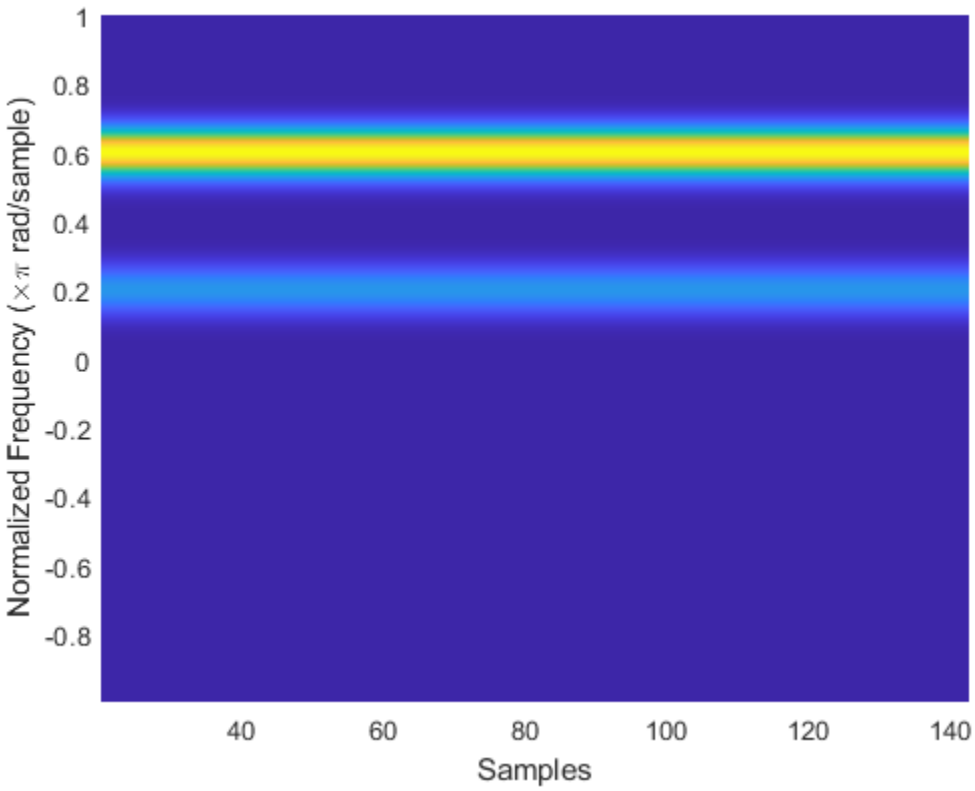
$$V_g f(t,\eta) = \sum_{k=1}^{K} A_k e^{-\pi(\eta-\nu_k)^2} e^{j2\pi\nu_k t}.$$

Create 1024 samples of a signal consisting of two sinusoids. One sinusoid has a normalized frequency of $\omega_0 = \pi/5$ rad/sample. The other sinusoid has three times the frequency and three times the amplitude.

```
N = 1024;
n = 0:N-1;

w0 = pi/5;
x = exp(1j*w0*n)+3*exp(1j*3*w0*n);
```

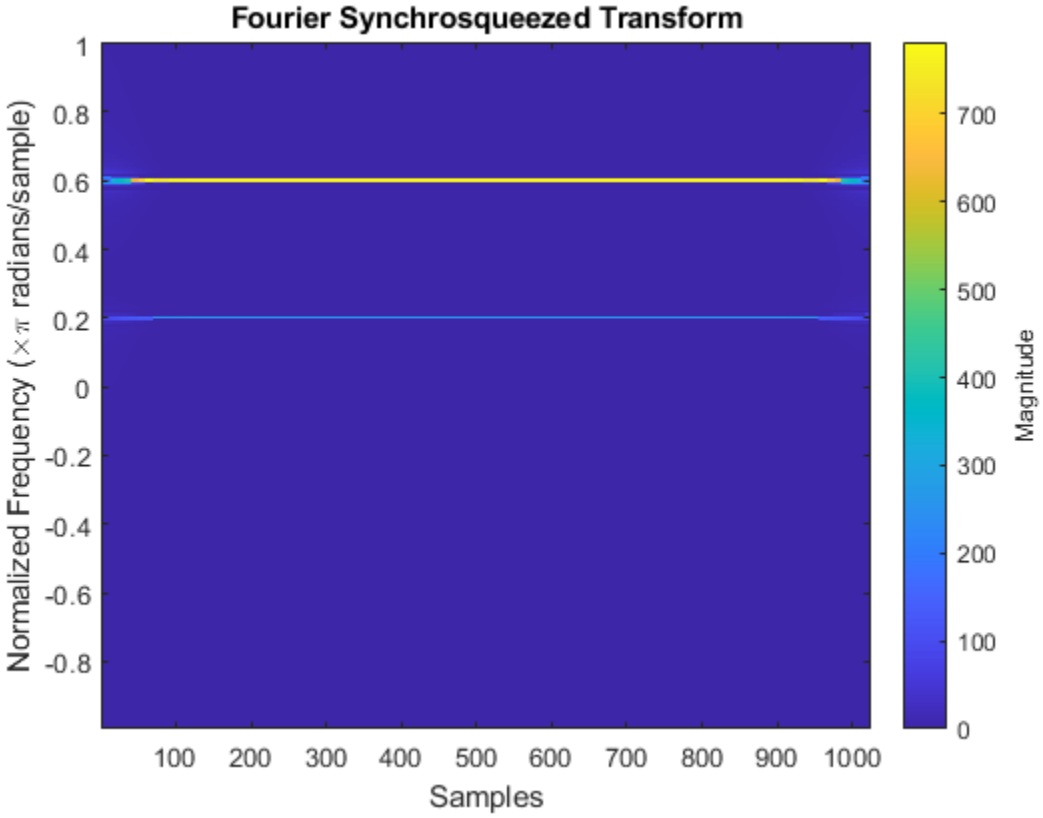
Compute the short-time Fourier transform of the signal. Use a 256-sample Gaussian window with $\alpha = 20$, 255 samples of overlap between adjoining sections, and 1024 DFT points. Plot the absolute value of the transform.

```
Nw = 256;
nfft = 1024;
alpha = 20;

[s,w,t] = spectrogram(x,gausswin(Nw,alpha),Nw-1,nfft,'centered');

surf(t,w/pi,abs(s),'EdgeColor','none')
view(2)
axis tight
xlabel('Samples')
ylabel('Normalized Frequency (\times\pi rad/sample)')
```



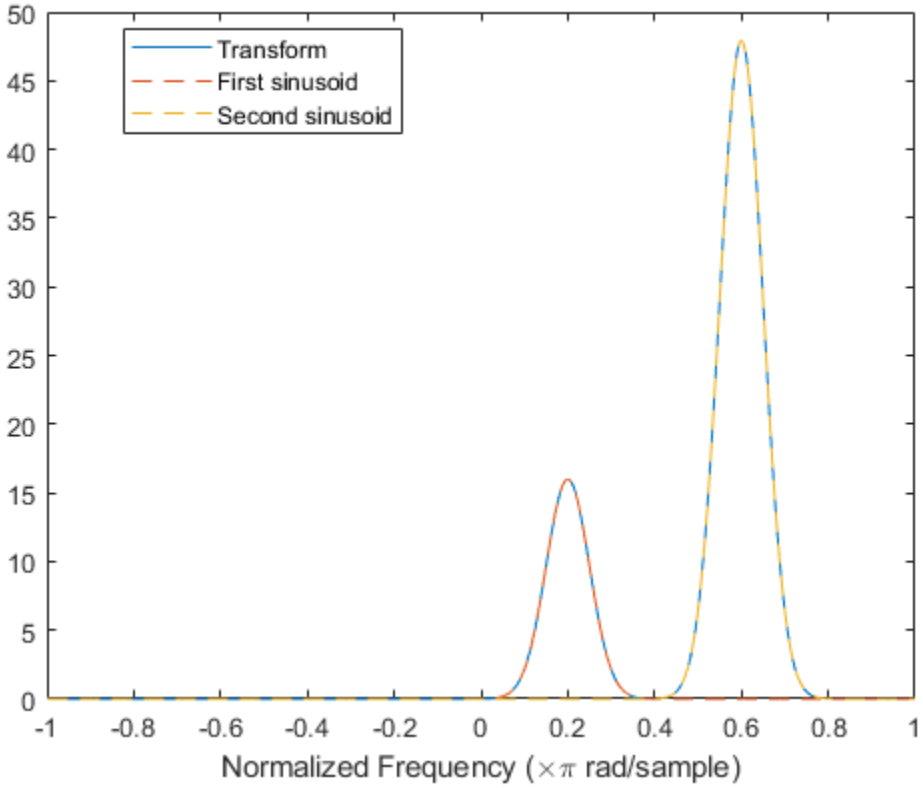

The Fourier synchrosqueezed transform results in a sharper, better localized estimate of the spectrum.

```
[ss,sw,st] = fsst(x,[],gausswin(Nw,alpha));

fsst(x,'yaxis')
```



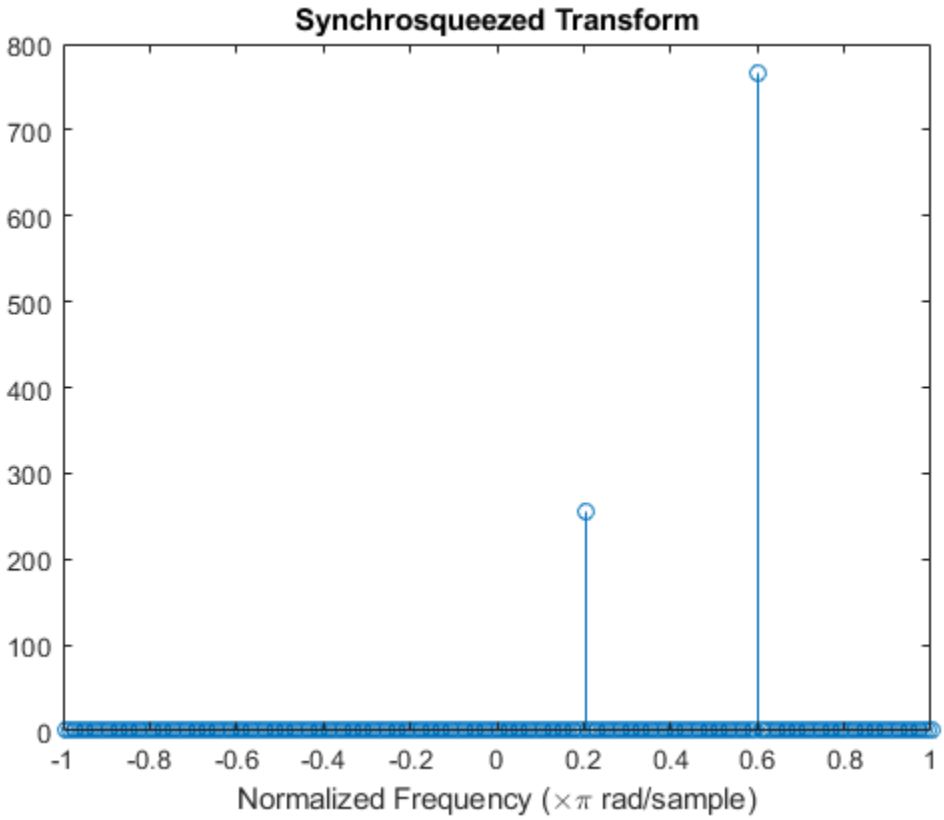

The sinusoids are visible as constant oscillations at the expected frequency values. To see that the decay away from the ridges is Gaussian, plot an instantaneous value of the transform and overlay two instances of a Gaussian. Express the Gaussian amplitude and standard deviation in terms of $\alpha$ and the window length. Recall that the standard deviation of the frequency-domain window is the reciprocal of the standard deviation of the time-domain window.

```
rstdev = (Nw-1)/(2*alpha);
amp = rstdev*sqrt(2*pi);

instransf = abs(s(:,128));

plot(w/pi,instransf)
hold on
plot(w/pi,[1 3]*amp.*exp(-rstdev^2/2*(w-[1 3]*w0).^2),'--')
hold off
xlabel('Normalized Frequency (\times\pi rad/sample)')
lg = legend('Transform','First sinusoid','Second sinusoid');
lg.Location = 'best';
```




The Fourier synchrosqueezed transform concentrates the energy content of the signal at the estimated instantaneous frequencies.

```
stem(sw/pi,abs(ss(:,128)))
xlabel('Normalized Frequency (\times\pi rad/sample)')
title('Synchrosqueezed Transform')
```



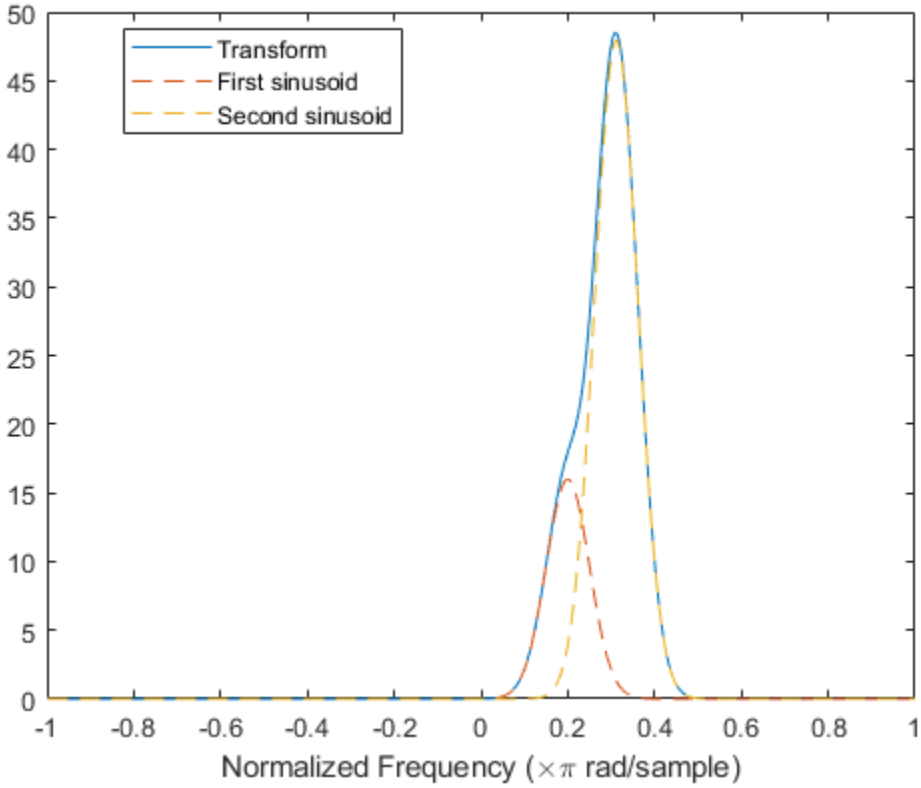

The synchrosqueezed estimates of instantaneous frequency are valid only if the sinusoids are separated by more than $2\Delta$, where

$$\Delta = \frac{1}{\sigma}\sqrt{2\log 2}$$

for a Gaussian window and $\sigma$ is the standard deviation.

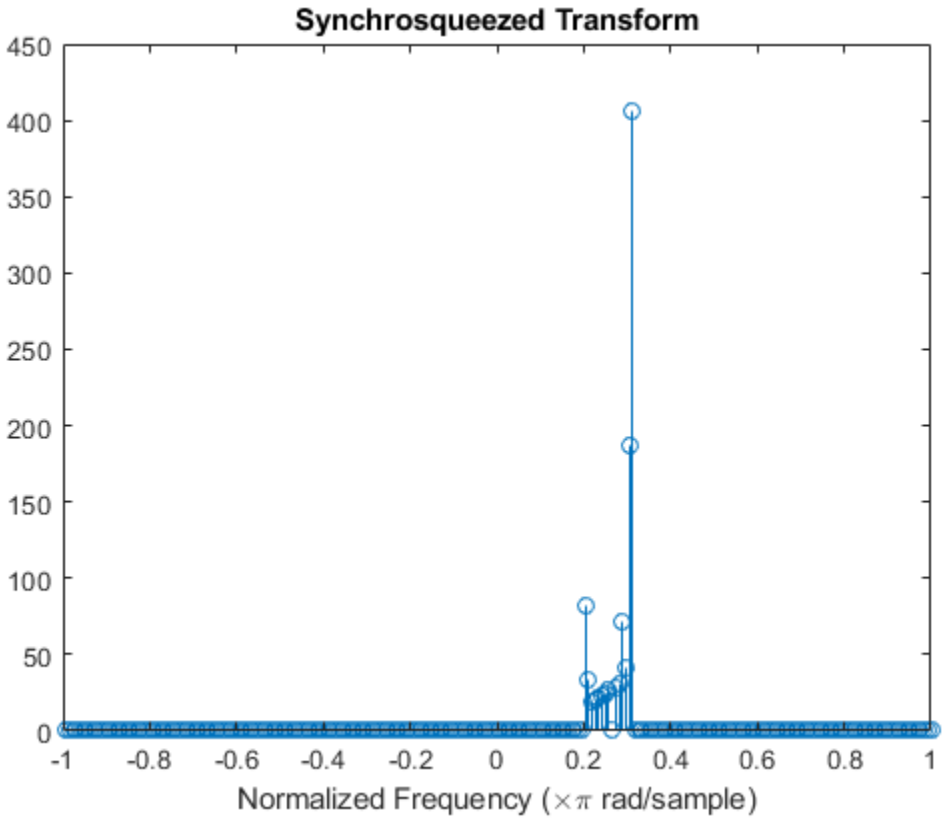
Repeat the previous calculation, but now specify that the second sinusoid has a normalized frequency of $\omega_0 + 1.9\Delta$ rad/sample.

```
D = sqrt(2*log(2))/rstdev;

w1 = w0+1.9*D;

x = exp(1j*w0*n)+3*exp(1j*w1*n);

[s,w,t] = spectrogram(x,gausswin(Nw,alpha),Nw-1,nfft,'centered');
instransf = abs(s(:,20));

plot(w/pi,instransf)
hold on
plot(w/pi,[1 3]*amp.*exp(-rstdev^2/2*(w-[w0 w1]).^2),'--')
hold off
xlabel('Normalized Frequency (\times\pi rad/sample)')
lg = legend('Transform','First sinusoid','Second sinusoid');
lg.Location = 'best';
```




The Fourier synchrosqueezed transform cannot resolve the sinusoids well because $|\omega_1 - \omega_0| < 2\Delta$. The spectral estimates decrease significantly in value.

```
[ss,sw,st] = fsst(x,[],gausswin(Nw,alpha));

stem(sw/pi,abs(ss(:,128)))
xlabel('Normalized Frequency (\times\pi rad/sample)')
title('Synchrosqueezed Transform')
```




## See Also

`fsst` | `gausswin` | `ifsst` | `spectrogram`

## Related Examples

- “Hilbert Transform and Instantaneous Frequency” on page 16-18
- “Instantaneous Frequency of Complex Chirp” on page 16-32
- “Practical Introduction to Time-Frequency Analysis” on page 24-267

# Instantaneous Frequency of Complex Chirp

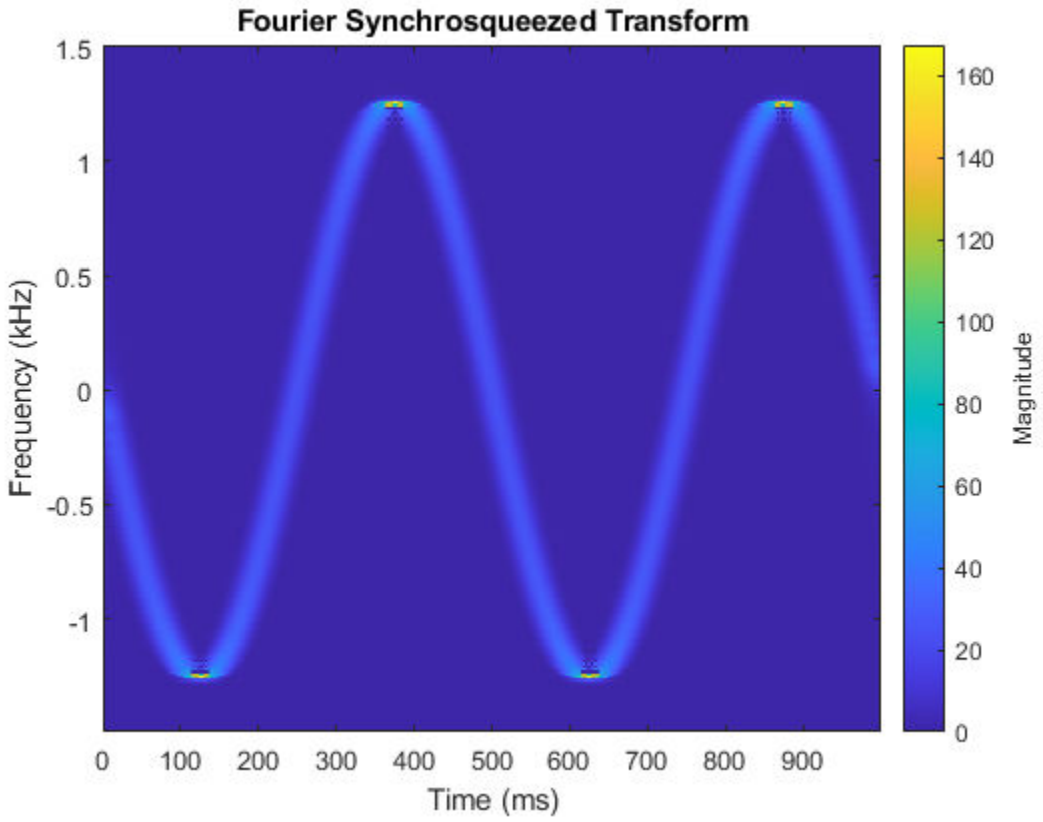
This example shows how to compute the instantaneous frequency of a signal using the Fourier synchrosqueezed transform.

Generate a chirp with sinusoidally varying frequency content. The signal is embedded in white Gaussian noise and sampled at 3 kHz for 1 second.

```
fs = 3000;
t = 0:1/fs:1-1/fs;

x = exp(2j*pi*100*cos(2*pi*2*t)) + randn(size(t))/100;
```

Compute and plot the Fourier synchrosqueezed transform of the signal. Display the time on the *x*-axis and the frequency on the *y*-axis.

```
fsst(x,fs,'yaxis')
```

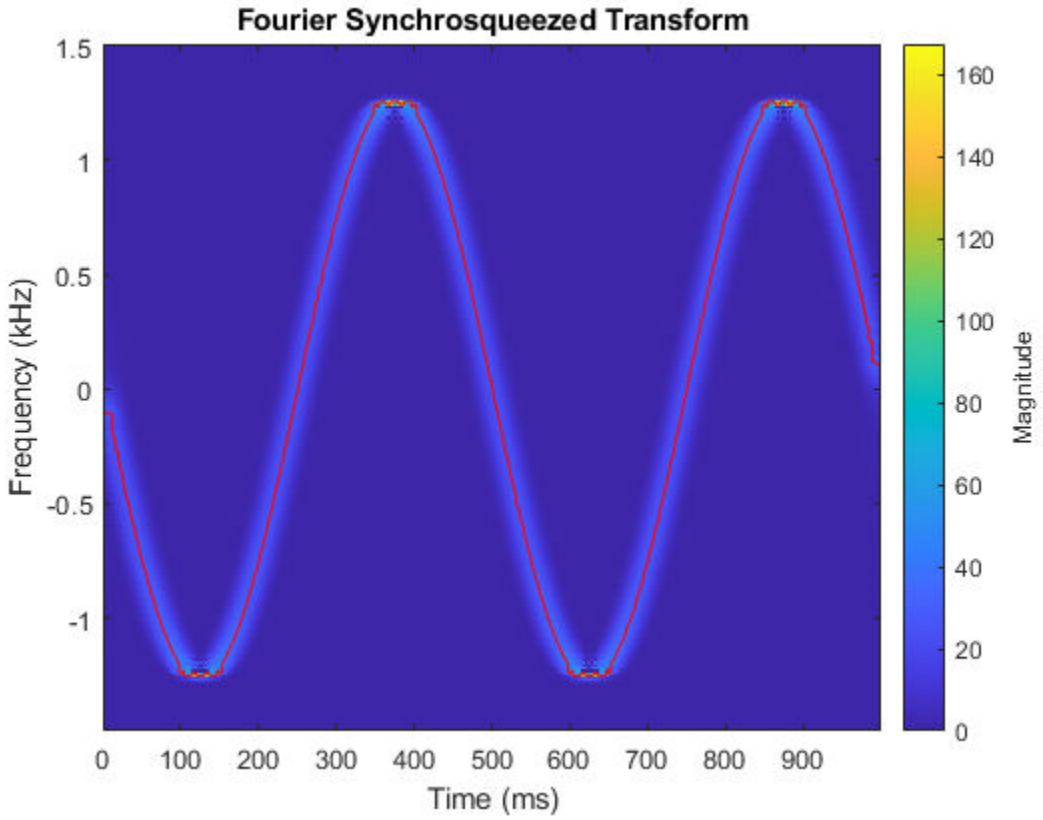



Find the instantaneous frequency of the signal by extracting the maximum-energy time-frequency ridge of the Fourier Synchrosqueezed transform.

```
[sst,f,tfs] = fsst(x,fs);

fridge = tfridge(sst,f);
```

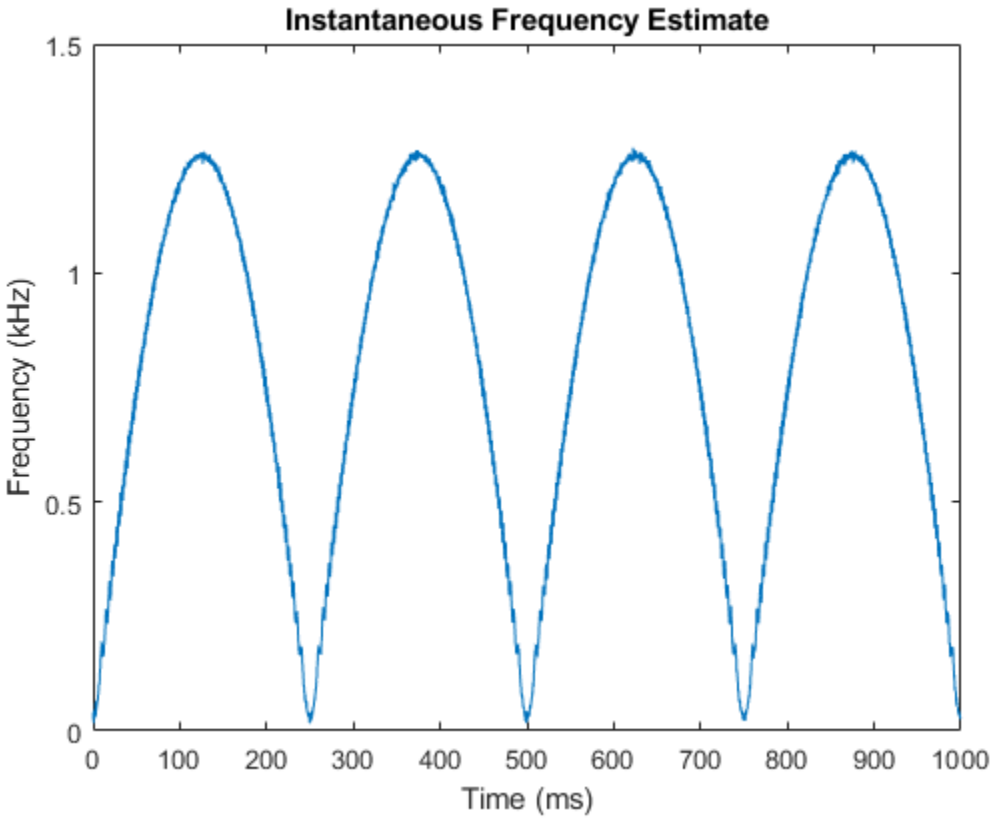
Overlay the ridge on the transform plot. Convert time to milliseconds and frequency to kHz.

```
hold on
plot(t*1000,fridge/1000,'r')
hold off
```




For a real signal, you can find the instantaneous frequency more easily using the `instfreq` function. For example, display the instantaneous frequency of the real part of the complex chirp by computing the analytic signal and differentiating its phase.

```
ax = real(x);

instfreq(ax,fs,'Method','hilbert')
```




## See Also

`fsst` | `ifsst` | `instfreq` | `pspectrum` | `spectrogram` | `tfridge`

## Related Examples

- “Hilbert Transform and Instantaneous Frequency” on page 16-18
- “Detect Closely Spaced Sinusoids” on page 16-25
- “Practical Introduction to Time-Frequency Analysis” on page 24-267

# Single-Sideband Amplitude Modulation

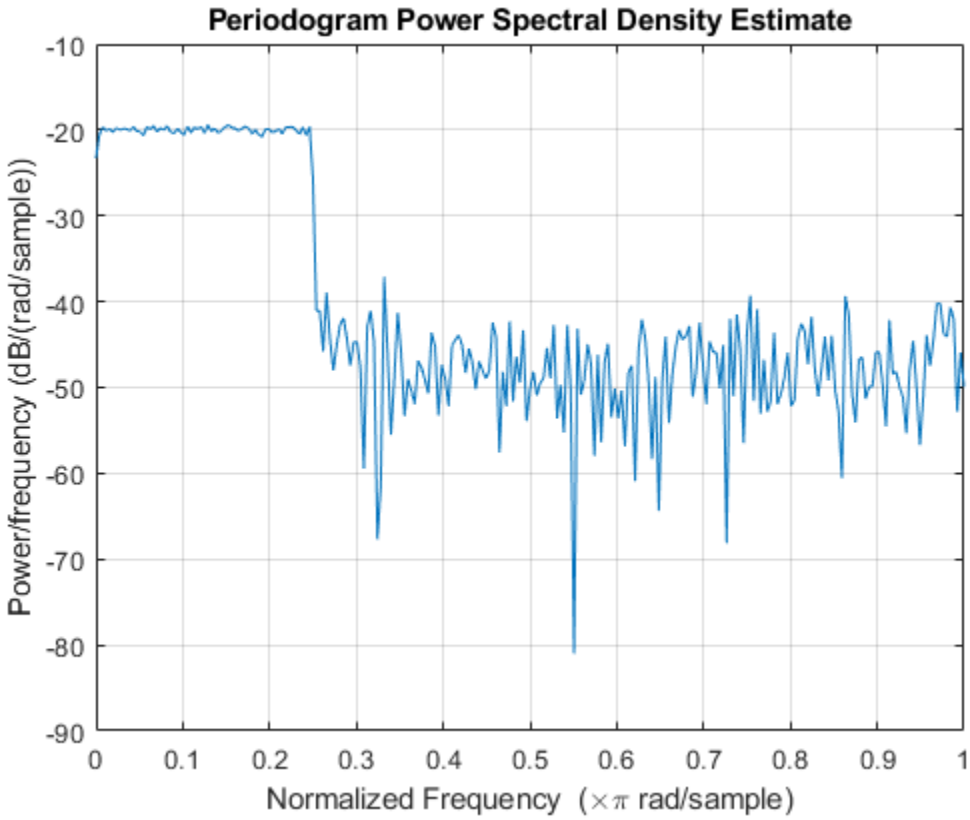
This example shows how to use the Hilbert transform to carry out single-sideband (SSB) amplitude modulation (AM) of a signal. Single-sideband AM signals have less bandwidth than normal AM signals.

Generate 512 samples of a simulated broadband signal using the `sinc` function. Specify a bandwidth of $\pi/4$ rad/sample.

```
N = 512;
n = 0:N-1;

bw = 1/4;
x = sinc((n-N/2)*bw);
```

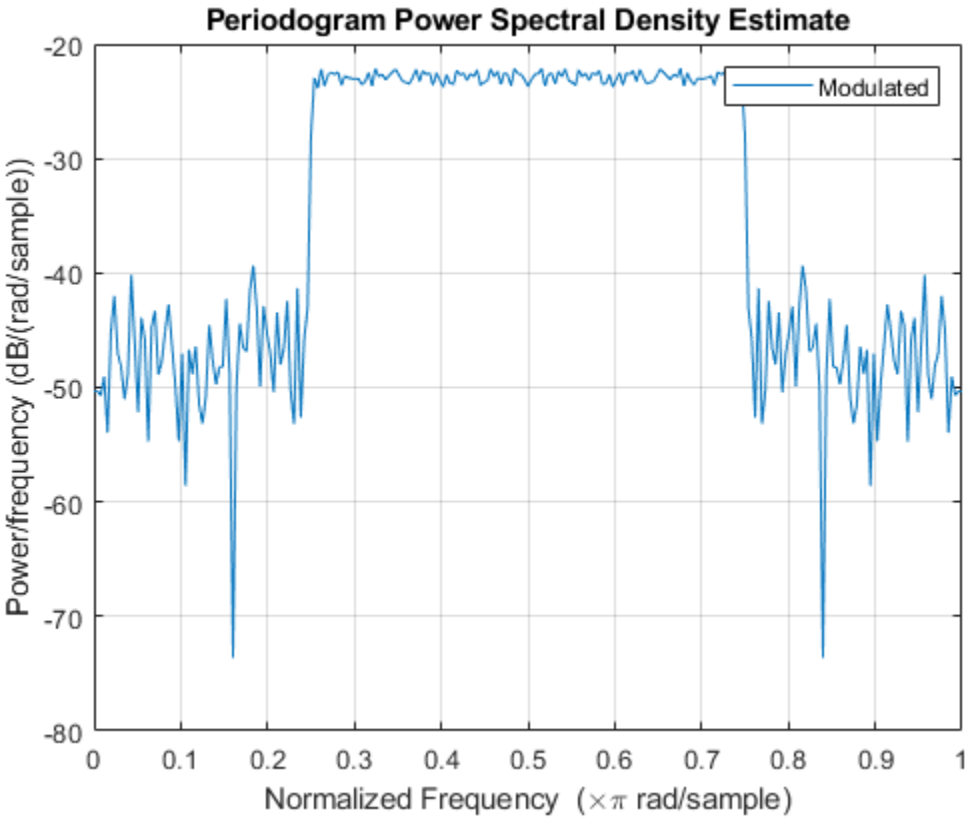
Add white Gaussian noise such that the signal-to-noise ratio is 20 dB. Reset the random number generator for reproducible results. Use the `periodogram` function to estimate the power spectral density (PSD) of the signal.

```
rng default

SNR = 20;
noise = randn(size(x))*std(x)/db2mag(SNR);
x = x + noise;

periodogram(x)
```

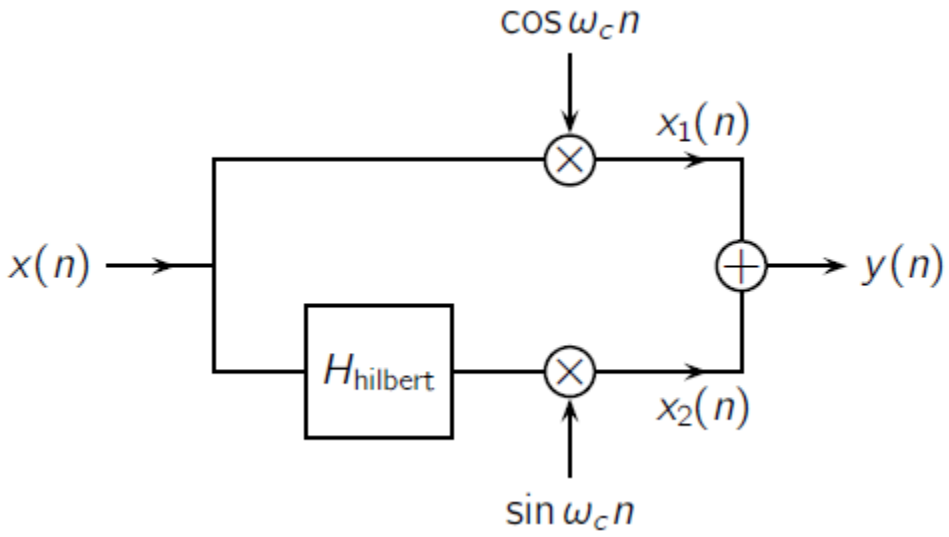


Amplitude modulate the signal using a cosine of carrier frequency $\omega_c = \pi/2$. Multiply by $\sqrt{2}$ so that the power of the modulated signal equals the power of the original signal. Estimate the PSD.

```
wc = pi/2;

x1 = x.*cos(wc*n)*sqrt(2);

periodogram(x1)
legend('Modulated')
```




SSB amplitude modulation reduces the bandwidth of the signal by half. To carry out SSB amplitude modulation, you must first compute the Hilbert transform of the signal. Then, amplitude modulate the signal using a sine with the same carrier frequency, $\omega_c$, as before, and add it to the previous signal.

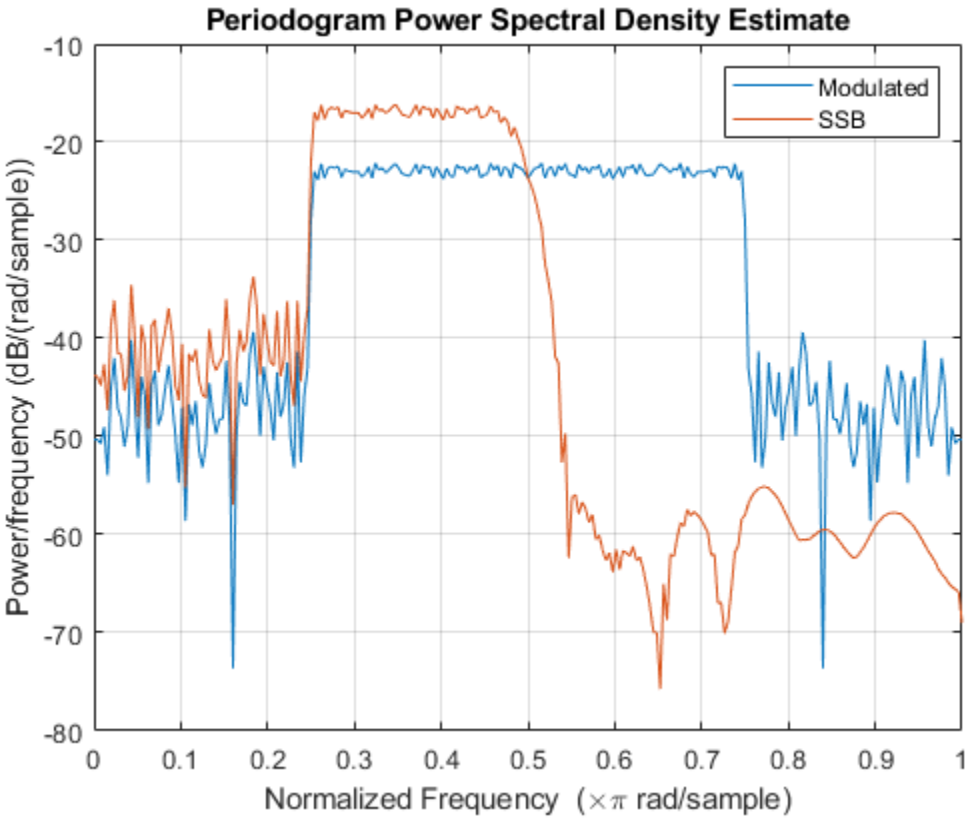



Design a Hilbert transformer using the `designfilt` function. Specify a filter order of 64 and a transition width of 0.1. Filter the signal.

```
Hhilbert = designfilt('hilbertfir','FilterOrder',64, ...
    'TransitionWidth',0.1);

xh = filter(Hhilbert,x);
```

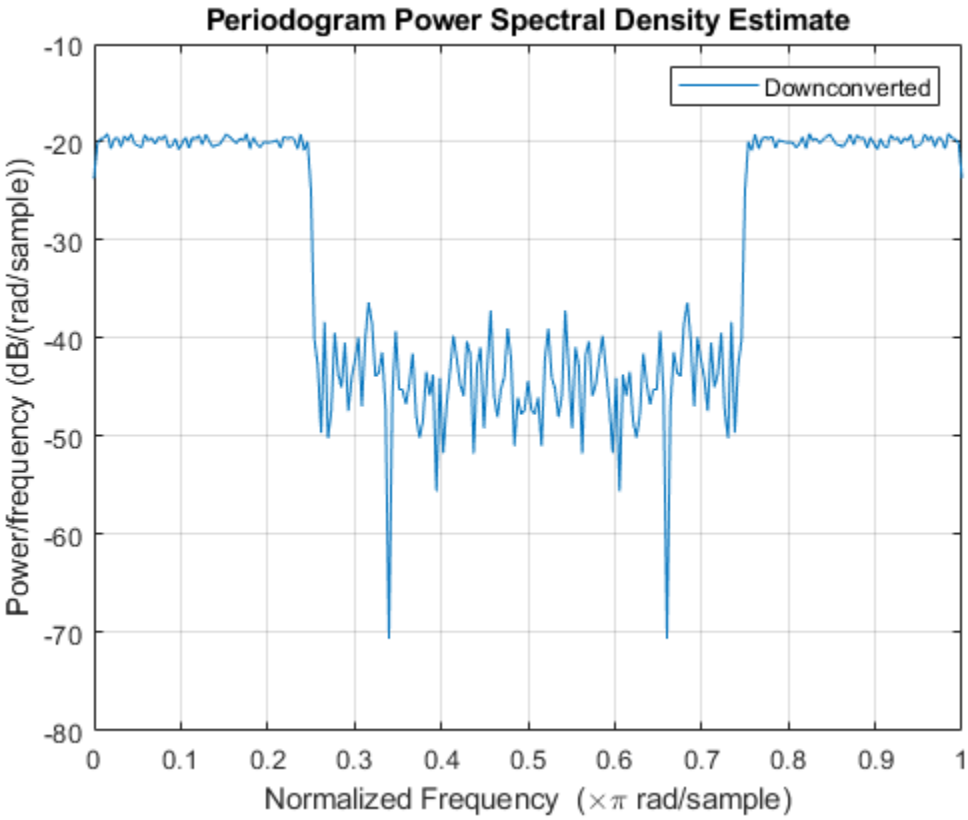
Use the `grpdelay` function to determine the delay, `gd`, introduced by the filter. Compensate for the delay by discarding the first `gd` points of the filtered signal and padding with zeros at the end. Amplitude modulate the result and add it to the original. Compare the PSDs.

```
gd = mean(grpdelay(Hhilbert));
xh = xh(gd+1:end);
eh = zeros(size(x));
eh(1:length(xh)) = xh;

x2 = eh.*sin(wc*n)*sqrt(2);

y = x1+x2;

periodogram([x1;y]')
legend('Modulated','SSB')
```




Downconvert the signal and estimate the PSD.

```
ym = y.*cos(wc*n)*sqrt(2);

periodogram(ym)
legend('Downconverted')
```

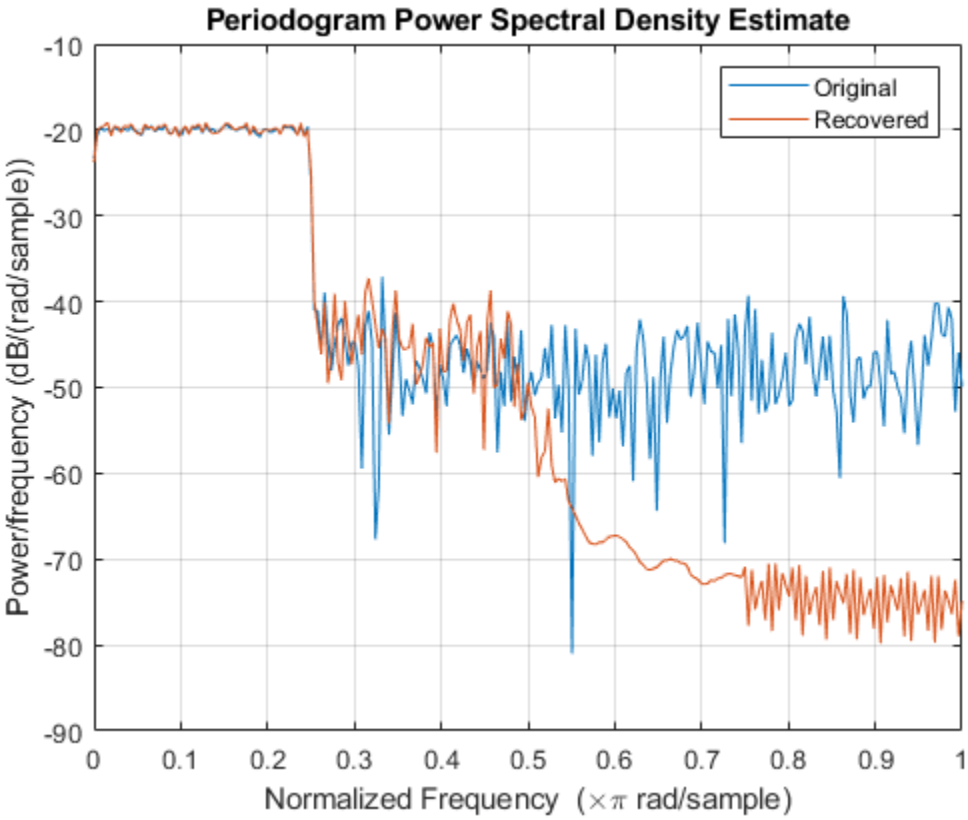



Lowpass filter the modulated signal to recover the original. Specify a 64th-order FIR lowpass filter with a cutoff frequency of $\pi/2$. Compensate for the delay introduced by the filter.

```
d = designfilt('lowpassfir','FilterOrder',64, ...
    'CutoffFrequency',0.5);
dem = filter(d,ym);

gd = mean(grpdelay(d));
dem = dem(gd+1:end);

dm = zeros(size(x));
dm(1:length(dem)) = dem;
```

Estimate the PSD of the filtered signal and compare it to that of the original.

```
periodogram([x;dm]')
legend('Original','Recovered')
```




Use the `snr` function to compare the signal-to-noise ratios of the two signals. Plot the two signals in the time domain.

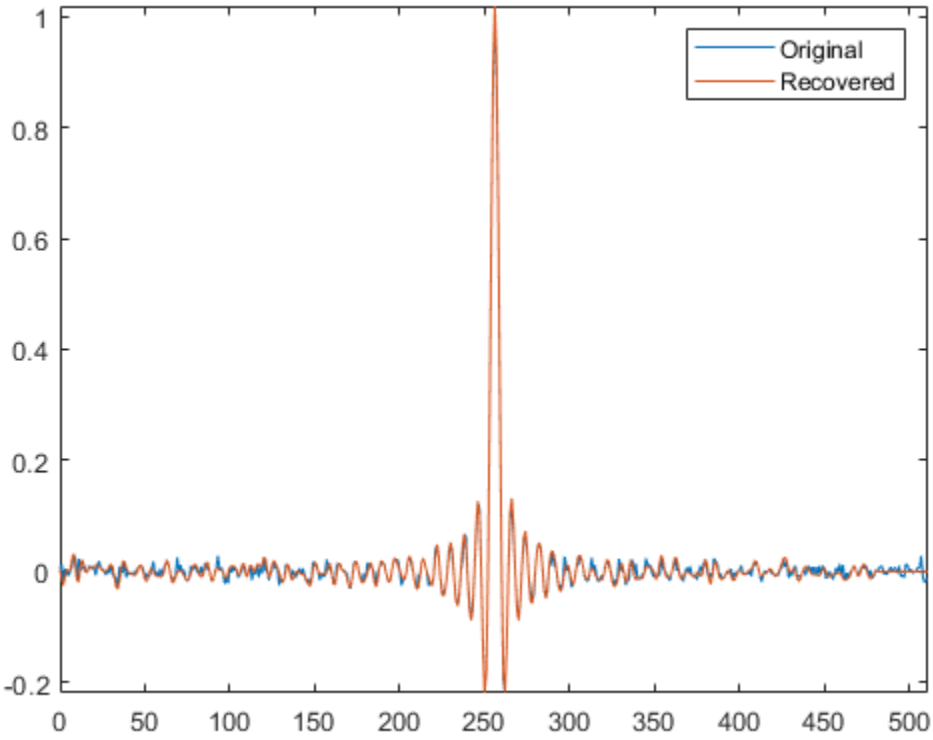
```
snrOrig = snr(x,noise)
```

```
snrOrig = 20.0259
```

```
snrRecv = snr(dm,noise)
```

```
snrRecv = 20.1373
```

```
plot(n,[x;dm]')
legend('Original','Recovered')
axis tight
```




**References**

Buck, John R., Michael M. Daniel, and Andrew C. Singer. *Computer Explorations in Signals and Systems Using MATLAB*. 2nd Edition. Upper Saddle River, NJ: Prentice Hall, 2002.

## See Also

`designfilt` | `periodogram` | `snr`

# DCT for Speech Signal Compression

This example shows how to compress a speech signal using the discrete cosine transform (DCT).

Load a file containing the word "strong," spoken by a woman and by a man. The signals are sampled at 8 kHz.

```
load('strong.mat')

% To hear, type soundsc(her,fs), pause(1), soundsc(him,fs)
```

Use the discrete cosine transform to compress the female voice signal. Decompose the signal into DCT basis vectors. There are as many terms in the decomposition as there are samples in the signal. The expansion coefficients in vector X measure how much energy is stored in each of the components. Sort the coefficients from largest to smallest.

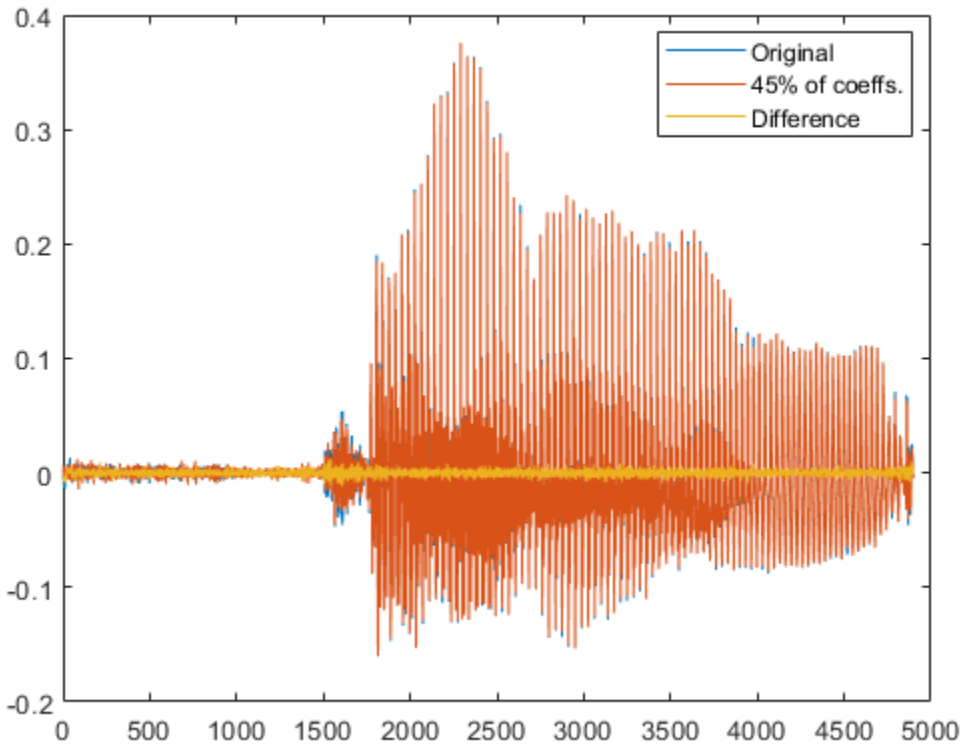
```
x = her';

X = dct(x);

[XX,ind] = sort(abs(X),'descend');
```

Find how many DCT coefficients represent 99.9% of the energy in the signal. Express the number as a percentage of the total.

```
need = 1;
while norm(X(ind(1:need)))/norm(X)<0.999
   need = need+1;
end

xpc = need/length(X)*100;
```

Set to zero the coefficients that contain the remaining 0.1% of the energy. Reconstruct the signal from the compressed representation. Plot the original signal, its reconstruction, and the difference between the two.

```
X(ind(need+1:end)) = 0;
xx = idct(X);

plot([x;xx;x-xx]')
legend('Original',[int2str(xpc) '% of coeffs.'],'Difference', ...
       'Location','best')
```



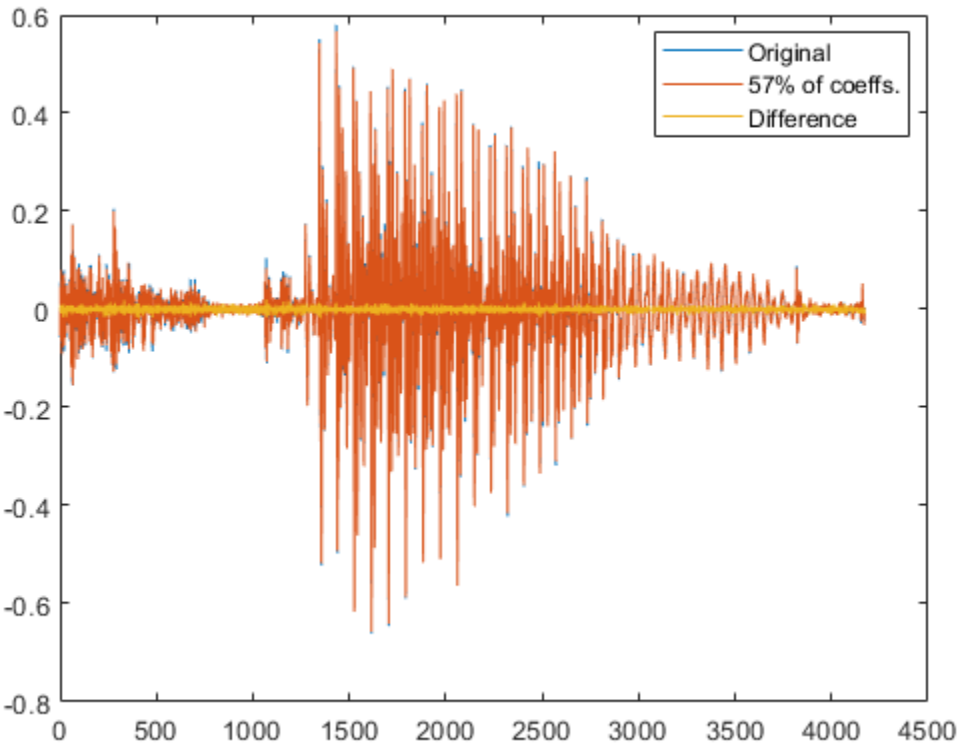

```
% To hear, type soundsc(x,fs), pause(1), soundsc(xx,fs)
```

Repeat the analysis for the male voice. Find how many DCT coefficients represent 99.9% of the energy and express the number as a percentage of the total.

```
y = him';
Y = dct(y);

[YY,ind] = sort(abs(Y),'descend');

need = 1;
while norm(Y(ind(1:need)))/norm(Y)<0.999
   need = need+1;
end

ypc = need/length(Y)*100;
```

Set the rest of the coefficients to zero and reconstruct the signal from the compressed version. Plot the original signal, its reconstruction, and the difference between the two.

```
Y(ind(need+1:end)) = 0;
yy = idct(Y);

plot([y;yy;y-yy]')
legend('Original',[int2str(ypc) '% of coeffs.'],'Difference', ...
       'Location','best')
```




```
% To hear, type soundsc(y,fs), pause(1), soundsc(yy,fs)
```

In both cases, about half of the DCT coefficients suffice to reconstruct the speech signal reasonably. If the required energy fraction is 99%, the number of necessary coefficients reduces to about 20% of the total. The resulting reconstruction is inferior but still intelligible.

Analysis of these and other samples suggests that more coefficients are needed to characterize the man's voice than the woman's.

## See Also

`dct` | `idct`

# Signal Generation

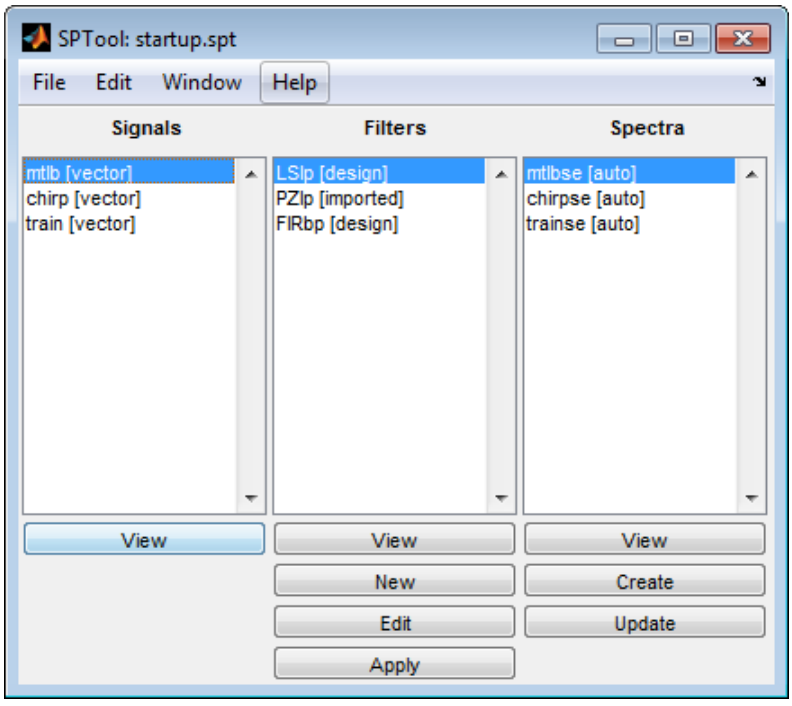
# Display Time-Domain Data in Signal Browser

| In this section... |
| --- |
| “Import and Display Signals” on page 17-2 |
| “Configure the Signal Browser Properties” on page 17-5 |
| “Modify the Signal Browser Display” on page 17-7 |
| “Inspect Your Data (Scaling the Axes and Zooming)” on page 17-8 |

This example shows how to use and configure the SPTool Signal Browser to display time-domain signals. First, open SPTool by typing

```
sptool
```

at the MATLAB command line. SPTool opens.

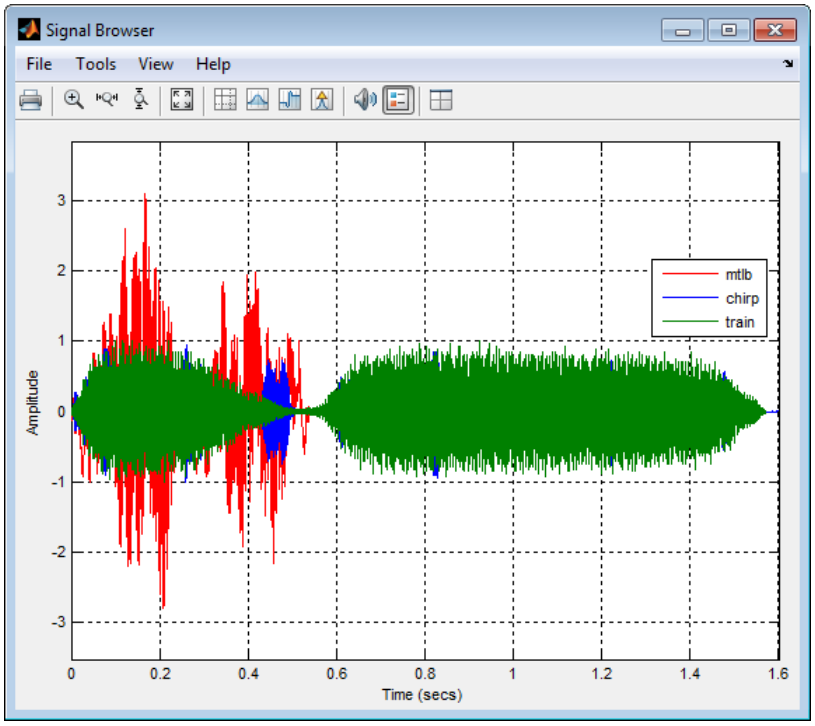



## Import and Display Signals

### Displaying Multiple Signals

You can display multiple signals in the Signal Browser by selecting more than one entry in the SPTool **Signals** list. In the **Signals** list, first select `mtlb [vector]`. Next, press the **Ctrl** key, and select `chirp [vector]`. Finally, click the **View** button. The Signal Browser opens, displaying both signals.

Any signals that you select in SPTool are now visible in the Signal Browser. You can also select multiple entries using the **Shift** key. In the SPTool **Signals** list, first select `mtlb [vector]`. Next, press the **Shift** key, and select `train [vector]`. The Signal Browser automatically updates to display all three signals, as shown in the following figure.

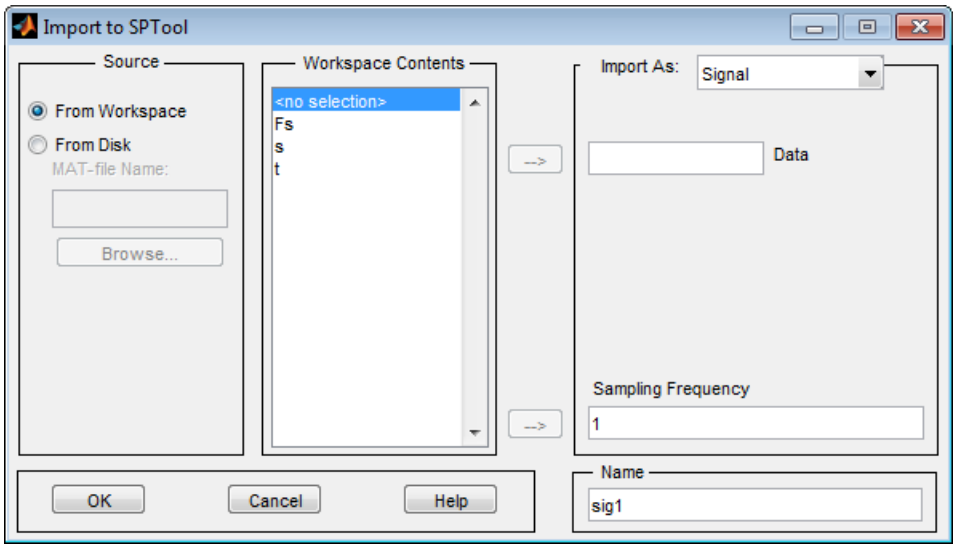



### Importing New Signals

Using SPTool, you can import signals from variables in the MATLAB workspace. First, create a 3-second signal, sampled at 10 kHz, that is the sum of two sine waves. At the MATLAB command line, enter the following commands:

```
Fs = 1e4;
t = 0:1/Fs:3;
s = sum(sin(2*pi*[350;440]*t));
```

To import the signal from these variables, in the SPTool menu, select **File** > **Import**. Alternatively, you can press the **Ctrl+I** keyboard shortcut. The Import to SPTool dialog box opens.



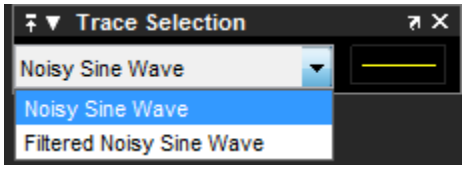
From the **Workspace Contents** list, select `s`. Click the right arrow (-->) button to the left of the **Data** box. Next, from the **Workspace Contents** list, select `Fs`. Click the right arrow (-->) button to the left of the **Sampling Frequency** box. You can assign the signal a name in the **Name** box, but since you will rename the signal later in this example, leave it as `sig1` for now.

Click **OK**. The SPTool **Signals** list now contains a signal named `sig1 [vector]`.

### Selecting a Signal and Playing Audio

In the SPTool **Signals** list, select `sig1 [vector]`, and click the **View** button. The Signal Browser reappears in front. To play audio for the signal, click the Play selected signal ( ) button. Signal Browser sends the audio signal to the speaker. For more information, see `sound`.

If you have multiple signals selected in SPTool, you can use the **Trace Selection** panel to choose which signal to make active. In the SPTool **Signals** list, press the **Ctrl** key, and select `mtlb [vector]`. The Signal Browser now displays two signals. In the Signal Browser menu, select **Tools** > **Measurements** > **Trace Selection**. The **Trace Selection** panel appears as shown in the following figure.




In the **Trace Selection** panel drop-down list, select `sig1`. Then, click the Play selected signal ( ) button. Signal Browser sends the audio signal to the speaker.

---

**Note** To hear audio when you click the Play selected signal ( ) button, your computer sound card must be able to support the sample rate of the signal. In this example, the sample rate of the signal, `sig1 [vector]`, is 10 kHz. If your sound card supports this or a greater sample rate, such as 44.1 kHz, then you can hear the audio on your speaker. For more information, see `sound`.

---

### Change Signal Names from the Legend

When multiple signals are displayed, Signal Browser shows a legend by default. To turn off the legend, click the Show all legends ( ) button. Click the button again to turn the legend back on. You can modify the names of the signal directly in the legend. To do so, when the legend is visible, click and drag it to any location on the display.

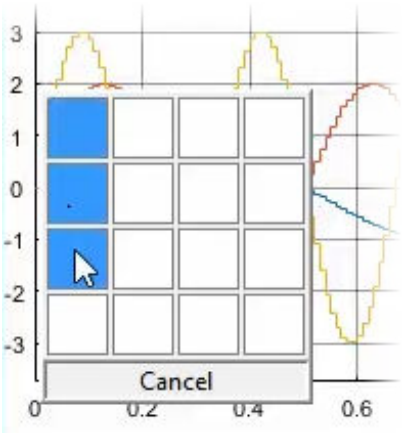
You can change the name of any signal directly within the legend. In the legend, double-click the signal name `sig1`. A cursor appears, indicating that you can now change this name. Highlight the text, and type `Dial Tone`. The legend now shows `Dial Tone` as the name of that signal. In the SPTool **Signals** list, the last item is now also named `Dial Tone`.

## Configure the Signal Browser Properties

First, configure the appearance of the Signal Browser window. In the SPTool **Signals** list, first select `mtlb [vector]`. Next, press the **Shift** key, and select `train [vector]`. The Signal Browser automatically updates to display all three signals.

### Multiple Displays

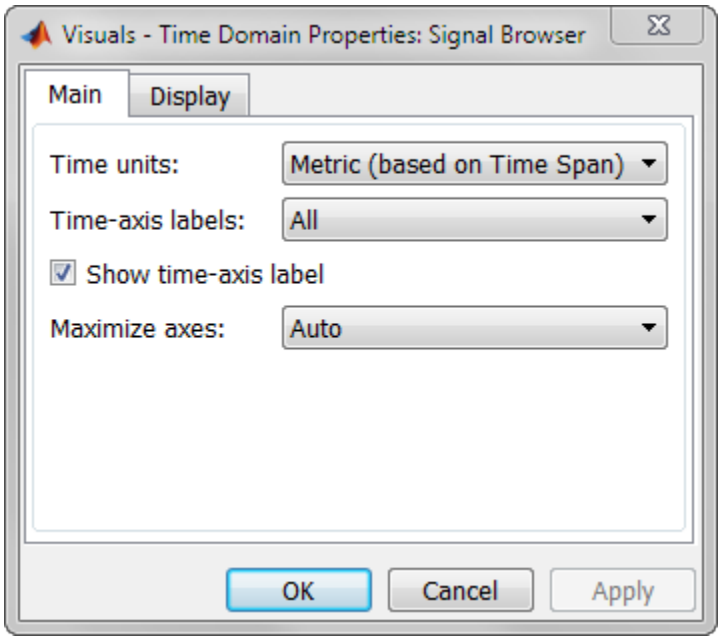
You can display multiple signals on different displays in the Signal Browser window. In the SPTool **Signals** list, first ensure all three signals are selected. In the Signal Browser toolbar, click the Layout ( ) button. Select row 3, column 1, as shown in the following figure.




After you make this selection, the Signal Browser is separated into three displays.

### Configure Appearance

In the Signal Browser menu, select **View > Properties**. The Visuals:Time Domain Options dialog box opens, as shown in the following figure.




In the Visuals:Time Domain Options dialog box, click the **Main** tab. Choose the appropriate parameter settings for the **Main** tab, as shown in the following table.

| Parameter | Setting |
|---|---|
| **Time units** | `Metric (based on Time Span)` |
| **Show time-axis labels** | `Bottom Displays Only` |
| **Maximize axes** | `On` |

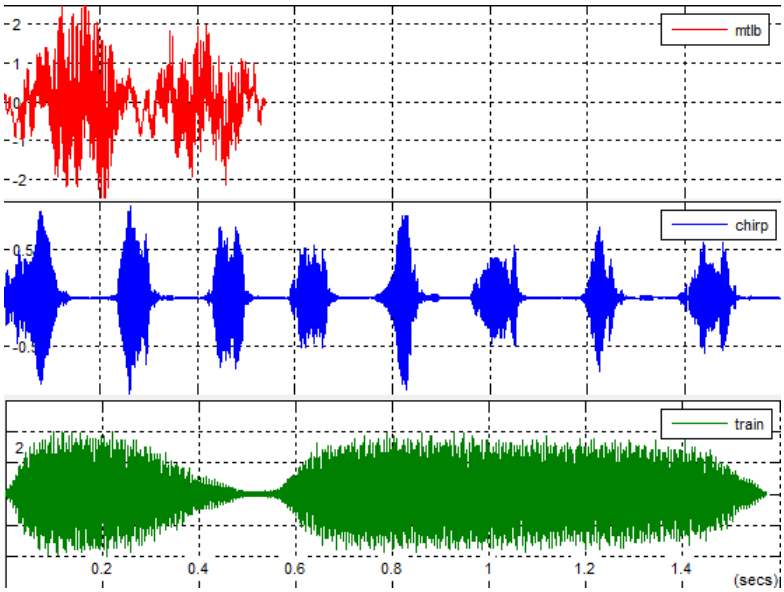
When you change the **Maximize axes** parameter to `On`, the axes are expanded to fill the entire display. To conserve space, titles and axis labels are not shown in each display. Click **Apply**.

### Set Display Properties

In the Visuals:Time Domain Options dialog box, click the **Display** tab. You can change the value of the **Select display** parameter to make different settings for each display. Set the parameters to the values shown in the following table.

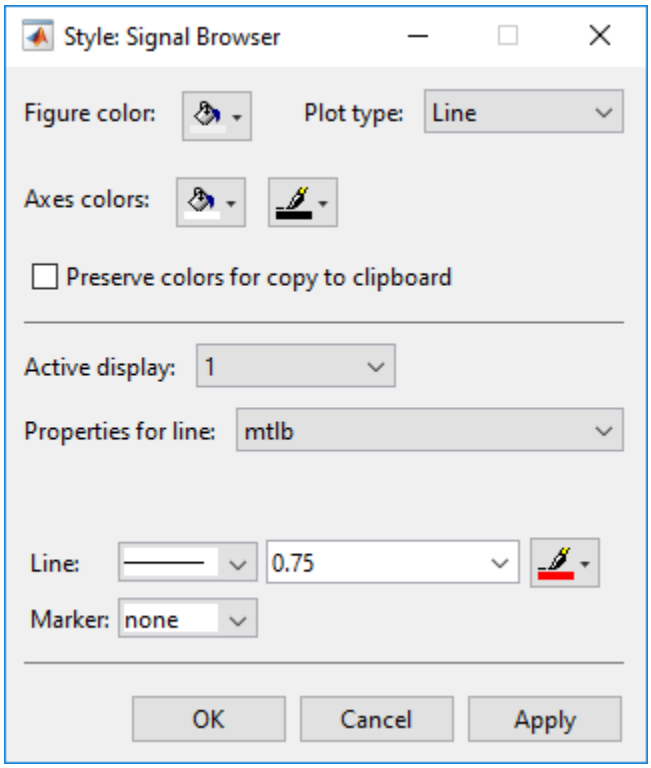
| Parameter | Display 1 Setting | Display 2 Setting | Display 3 Setting |
|---|---|---|---|
| **Select display** | 1 | 2 | 3 |
| **Title** | `mtlb` | `chirp` | `train` |
| **Show legend** | Selected | Selected | Selected |
| **Show grid** | Selected | Selected | Selected |
| **Plot signal(s) as magnitude and phase** | Cleared | Cleared | Cleared |
| **Minimum Y-limit** | `-2.5` | `-1` | `-1.5` |
| **Maximum Y-limit** | `2.5` | `1` | `1.5` |
| **Y-axis label** | `Amplitude` | `Amplitude` | `Amplitude` |

Click **OK** to save your changes and close the Visuals:Time Domain Options dialog box. The Signal Browser appears as shown in the following figure.




## Modify the Signal Browser Display

Use the Style dialog box to modify the appearance of the axes and the lines for each of the selected signals in SPTool. In the Signal Browser menu, select **View > Style**. The Style dialog box opens, as shown in the following figure.

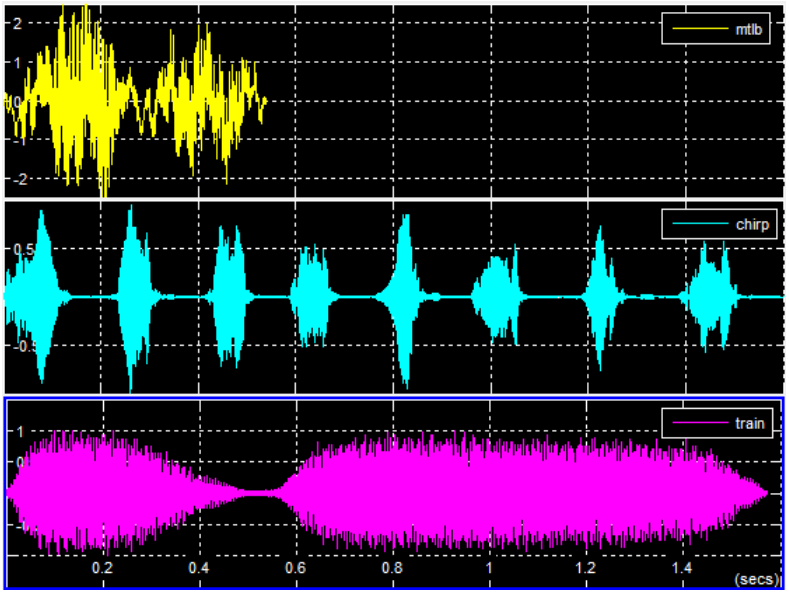


### Modify Axes Colors and Line Properties

You can change the value of the **Select display** parameter to make different settings for each display. Set the parameters to the values shown in the following table.

| Parameter | Display 1 Setting | Display 2 Setting | Display 3 Setting |
|---|---|---|---|
| **Select display** | 1 | 2 | 3 |
| **Axes background color** | Black | Black | Black |
| **Ticks, labels, and grid colors** | White | White | White |
| **Line color** | Yellow | Cyan | Magenta |

These settings enable the Signal Browser to display line colors in the same manner as the Simulink Scope block. Click **OK** to save your changes and close the Style dialog box. The Signal Browser now appears as shown in the following figure.




### Show and Hide Toolbar

To hide the toolbar, from the Signal Browser menu, select **View > Toolbar**. Doing so removes the toolbar from the Signal Browser window and also removes the check mark beside the **Toolbar** option in the **View** menu. You can choose to show the toolbar again at any time by selecting **View > Toolbar**.

## Inspect Your Data (Scaling the Axes and Zooming)

So far, you have manually set the *y*-axis limits. Use one of the following options to let Signal Browser scale the axes:

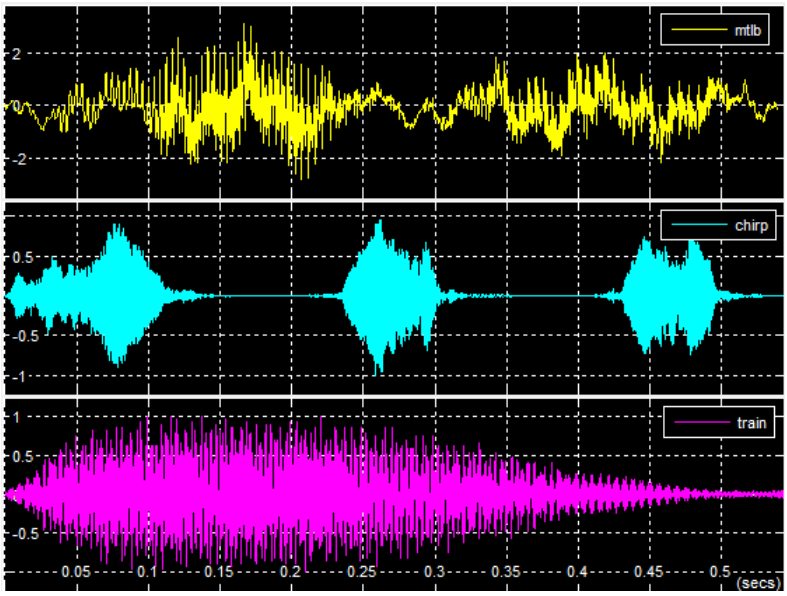
- From the Signal Browser menu, select **Tools > Scale Axes Limits**.
- From the Signal Browser toolbar, click the Scale Axes Limits ( ) button.

- With the Signal Browser as your active window, press **Ctrl + A**.

### Use the Zoom Tools

The zoom tools allow you to zoom in simultaneously in the directions of both the *x*- and *y*-axes , or in either direction individually. For example, to zoom in on the signal between 0 and 0.5 seconds, you can use the **Zoom X** option.

- To activate the **Zoom X** tool, select **Tools > Zoom X**, or press the corresponding toolbar button ( ). The Signal Browser indicates that the **Zoom X** tool is active by indenting the toolbar button and placing a check mark next to the **Tools > Zoom X** menu option.
- Next, zoom in on the region between 0 and 0.5 seconds. In the Signal Browser window, click on the 0-second mark and drag to the 0.5-second mark. All three displays reflect this new *x*-axis setting, as shown in the following figure.




- To zoom out of the Signal Browser window, right-click inside the window, and select **Zoom Out**. Alternatively, you can return to the original view of your signal by right-clicking inside the Signal Browser window and selecting **Reset to Original View**.

# Signal Measurement


- “RMS Value of Periodic Waveforms” on page 18-2
- “Slew Rate of Triangular Waveform” on page 18-5
- “Duty Cycle of Rectangular Pulse Waveform” on page 18-8
- “Estimate State for Digital Clock” on page 18-11
- “Calculate Settling Time with Signal Browser” on page 18-14
- “Find Peak Amplitudes in Signal Browser” on page 18-16
- “Distortion Measurements” on page 18-18
- “Prominence” on page 18-22
- “Determine Peak Widths” on page 18-24

# RMS Value of Periodic Waveforms

This example shows how to find the root mean square (RMS) value of a sine wave, a square wave, and a rectangular pulse train using `rms`. The waveforms in this example are discrete-time versions of their continuous-time counterparts.

Create a sine wave with a frequency of $\pi/4$ rad/sample. The length of the signal is 16 samples, which equals two periods of the sine wave.

```
n = 0:15;
x = cos(pi/4*n);
```

Compute the RMS value of the sine wave.

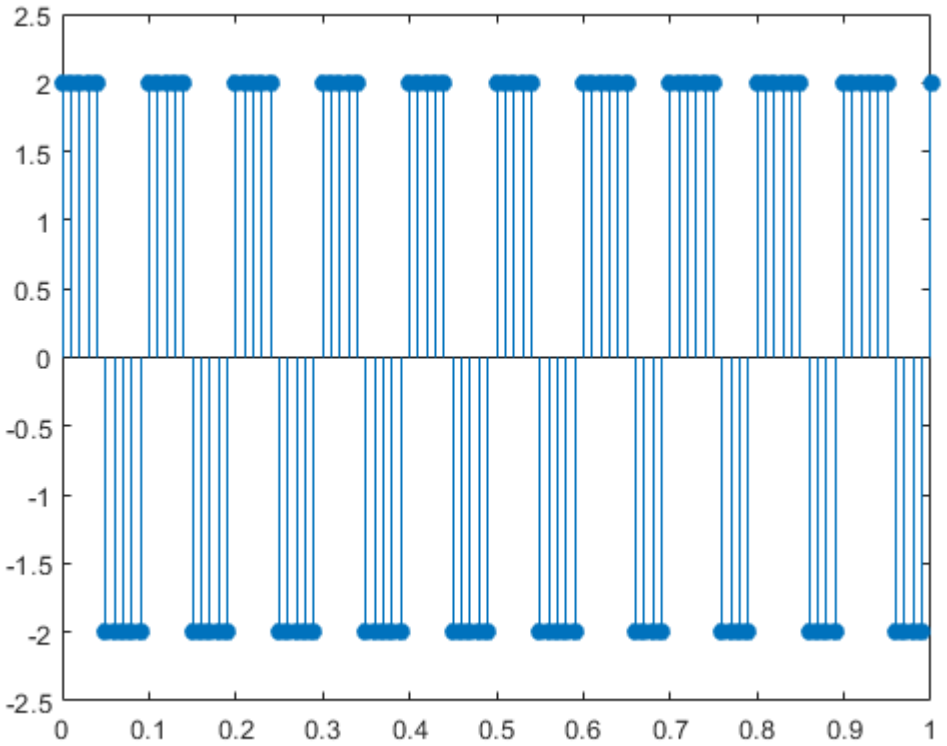
```
rmsval = rms(x)
```

```
rmsval = 0.7071
```

The RMS value is equal to $1/\sqrt{2}$, as expected.

Create a periodic square wave with a period of 0.1 seconds. The square wave values oscillate between $-2$ and 2.

```
t = 0:0.01:1;
x = 2*square(2*pi*10*t);

stem(t,x,'filled')
axis([0 1 -2.5 2.5])
```



Find the RMS value.

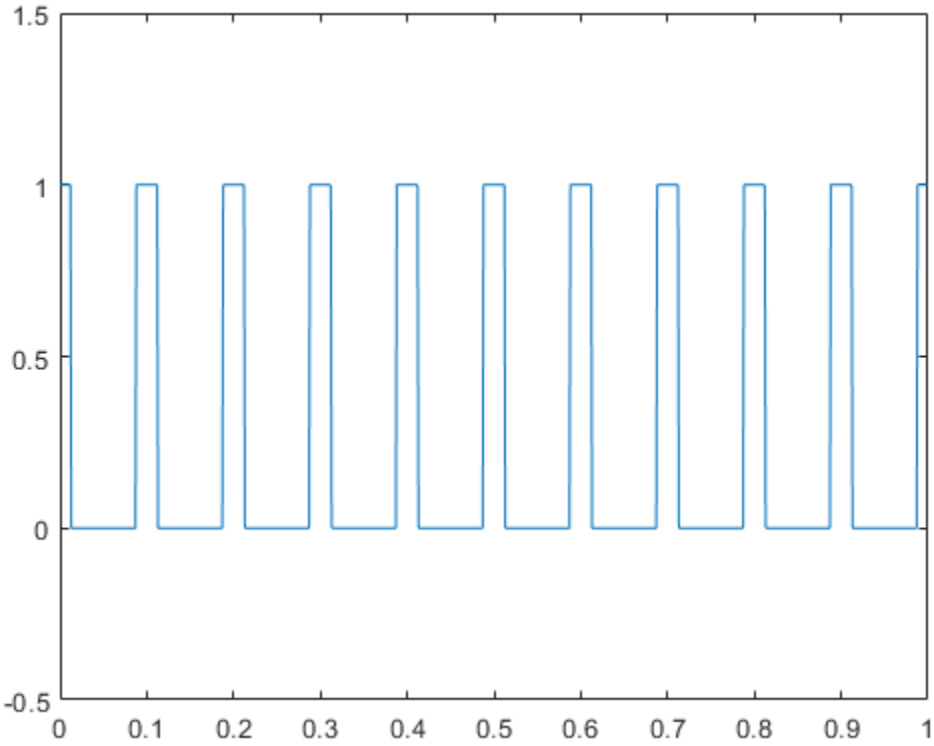
```
rmsval = rms(x)
```

```
rmsval = 2
```

The RMS value agrees with the theoretical value of 2.

Create a rectangular pulse train sampled at 1 kHz with the following parameters: the pulse is on, or equal to 1, for 0.025 seconds, and off, or equal to 0, for 0.075 seconds in each 0.1 second interval. This means the pulse period is 0.1 seconds and the pulse is on for 1/4 of that interval. This is referred to as the *duty cycle*. Use `pulstran` to create the rectangular pulse train.

```
t = 0:0.001:(10*0.1);
pulsewidth = 0.025;
pulseperiods = [0:10]*0.1;
x = pulstran(t,pulseperiods,@rectpuls,pulsewidth);

plot(t,x)
axis([0 1 -0.5 1.5])
```




Find the RMS value and compare it to the RMS of a continuous-time rectangular pulse waveform with duty cycle 1/4 and peak amplitude 1.

```
rmsval = rms(x)
```

```
rmsval = 0.5007
```

```
thrms = sqrt(1/4)
```

```
thrms = 0.5000
```

The observed RMS value and the RMS value for a continuous-time rectangular pulse waveform are in good agreement.

# Slew Rate of Triangular Waveform

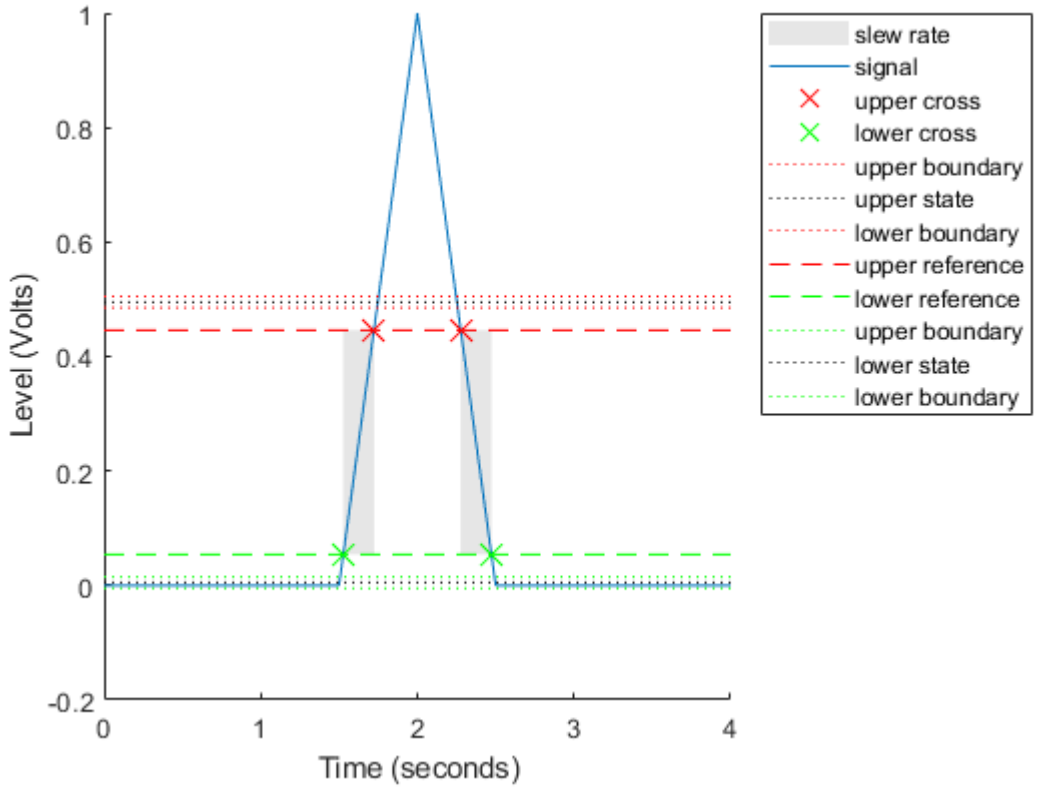
This example shows how to use the slew rate as an estimate of the rising and falling slopes of a triangular waveform. Create three triangular waveforms. One waveform has rising-falling slopes of $\pm 2$, one waveform has rising-falling slopes of $\pm \frac{1}{2}$, and one waveform has a rising slope of $+2$ and a falling slope of $-\frac{1}{2}$. Use `slewrate` to find the slopes of the waveforms.

Use `tripuls` to create a triangular waveform with rising-falling slopes of $\pm 2$. Set the sampling interval to 0.01 seconds, which corresponds to a sample rate of 100 hertz.

```
dt = 0.01;
t = -2:dt:2;

x = tripuls(t);
```

Compute and plot the slew rate for the triangular waveform. Input the sample rate (100 Hz) to obtain the correct positive and negative slope values.

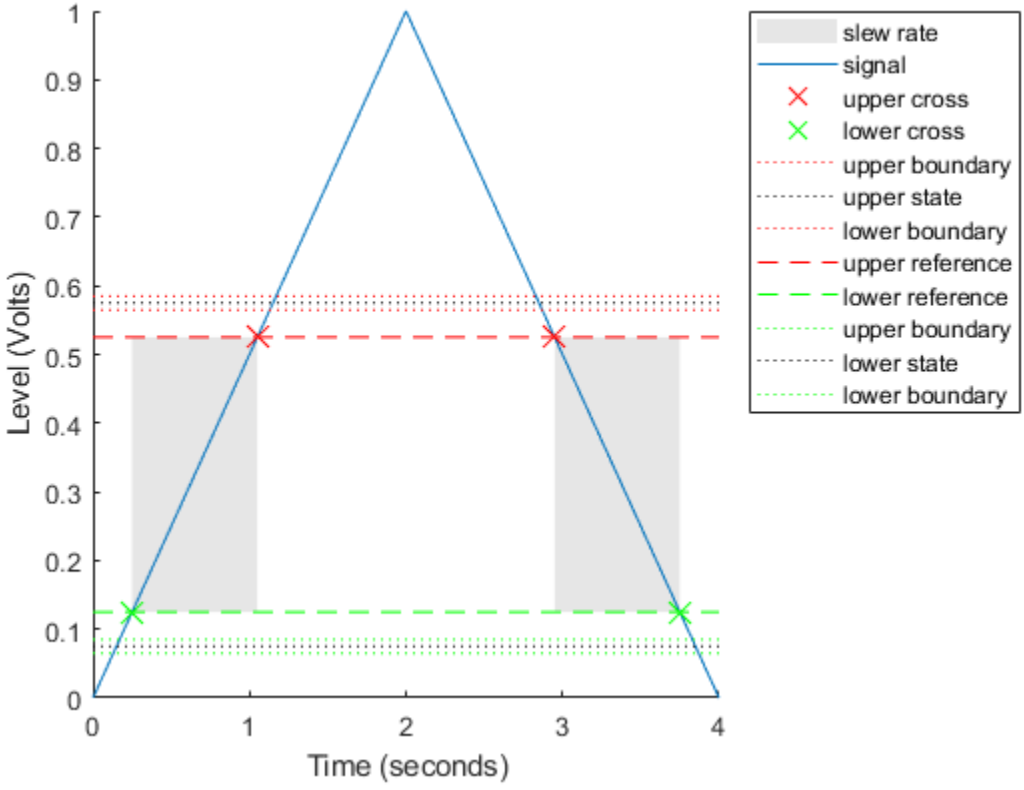
```
slewrate(x,1/dt)
```




```
ans = 1×2

    2.0000   -2.0000
```

Change the width of the triangular waveform so it has slopes of $\pm\frac{1}{2}$. Compute and plot the slew rate.

```
x = tripuls(t,4);
slewrate(x,1/dt)
```




```
ans = 1×2

    0.5000   -0.5000
```

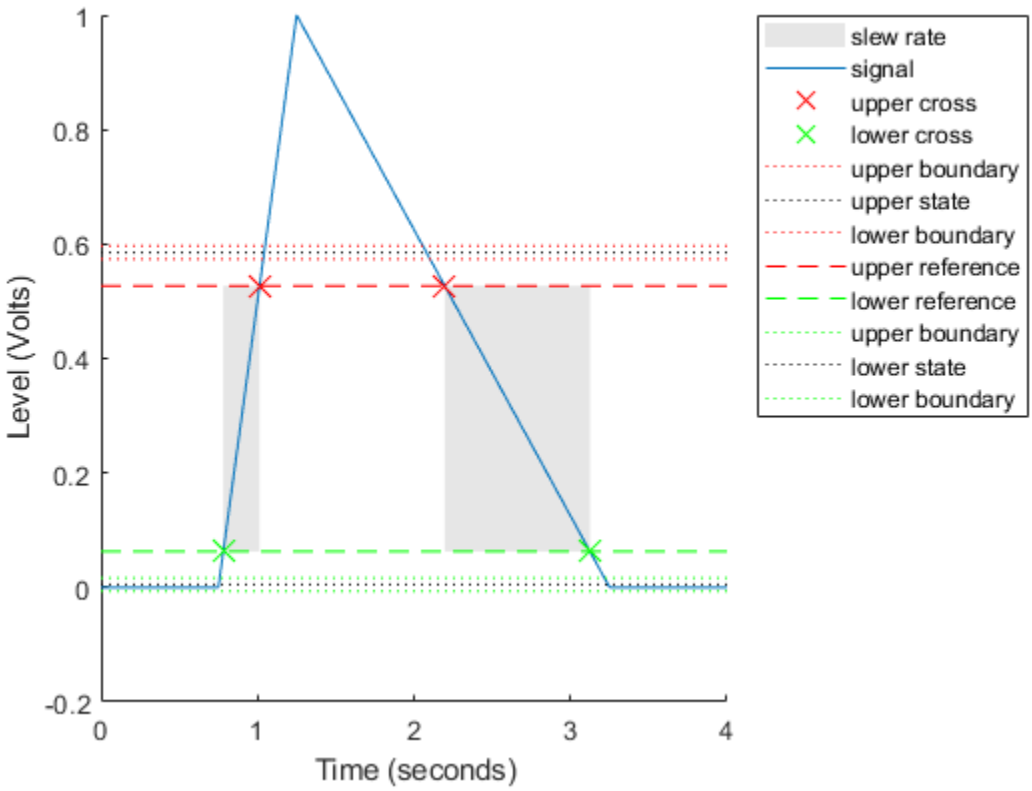
Create a triangular waveform with a rising slope of +2 and a falling slope of $-\frac{1}{2}$. Compute the slew rate.

```
x = tripuls(t,5/2,-3/5);
s = slewrate(x,1/dt)
```

```
s = 1×2

    2.0000   -0.5000
```

The first element of `s` is the rising slope and the second element is the falling slope. Plot the result.

```
slewrate(x,1/dt);
```




## See Also

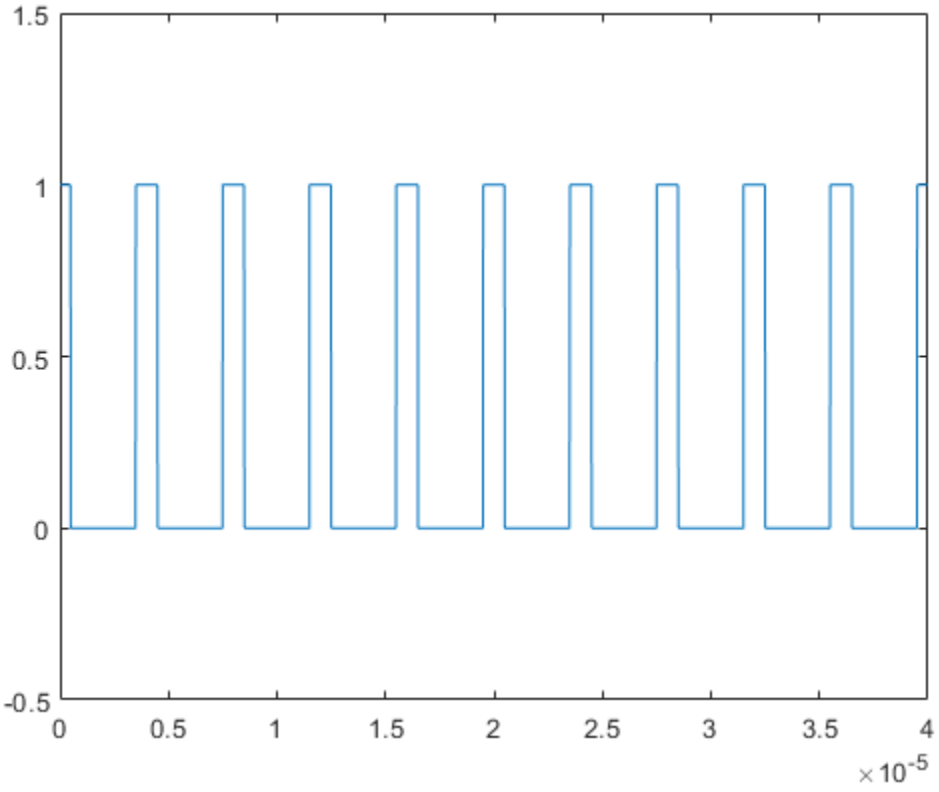
`slewrate` | `tripuls`

# Duty Cycle of Rectangular Pulse Waveform

This example shows how to create a rectangular pulse waveform and measure its duty cycle. You can think of a rectangular pulse waveform as a sequence of *on* and *off* states. One pulse period is the total duration of an *on* and *off* state. The pulse width is the duration of the *on* state. The duty cycle is the ratio of the pulse width to the pulse period. The duty cycle for a rectangular pulse describes the fraction of time that the pulse is *on* in one pulse period.

Create a rectangular pulse sampled at 1 gigahertz. The pulse is *on*, or equal to 1, for a duration of 1 microsecond. The pulse if *off*, or equal to 0, for a duration of 3 microseconds. The pulse period is 4 microseconds. Plot the waveform.

```
Fs = 1e9;
t = 0:1/Fs:(10*4e-6);

pulsewidth = 1e-6;
pulseperiods = [0:10]*4e-6;

x = pulstran(t,pulseperiods,@rectpuls,pulsewidth);

plot(t,x)
axis([0 4e-5 -0.5 1.5])
```




Determine the duty cycle of the waveform using `dutycycle`. Input both the pulse waveform and the sample rate to output the duty cycle. `dutycycle` outputs a duty cycle value for each detected pulse.

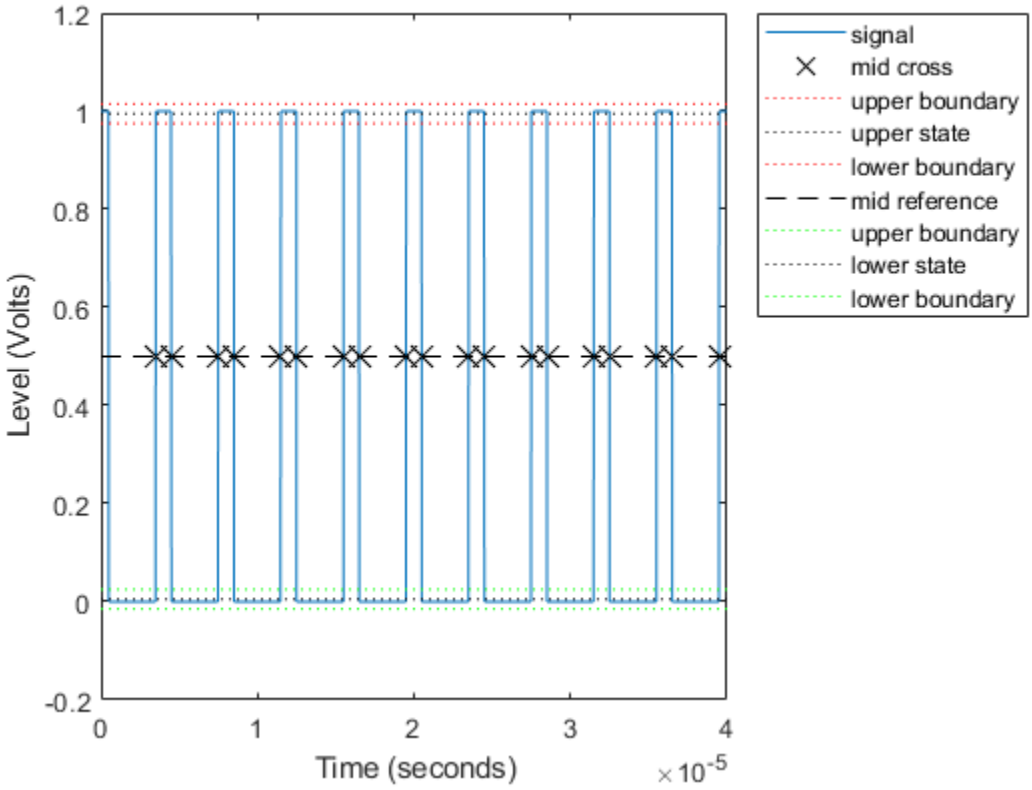
```
D = dutycycle(x,Fs)
```

```
D = 1×9

    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500    0.2500
```

In this example, the duty cycle for each of the detected pulses is identical and equal to 0.25. This is the expected duty cycle because the pulse is on for 1 microsecond and off for 3 microseconds in each 4 microsecond period. Therefore, the pulse is on for 1/4 of each period. Expressed as a percentage, this is equal to a duty cycle of 25%.

Calling `dutycycle` with no output arguments produces a plot with all the detected pulse widths marked.

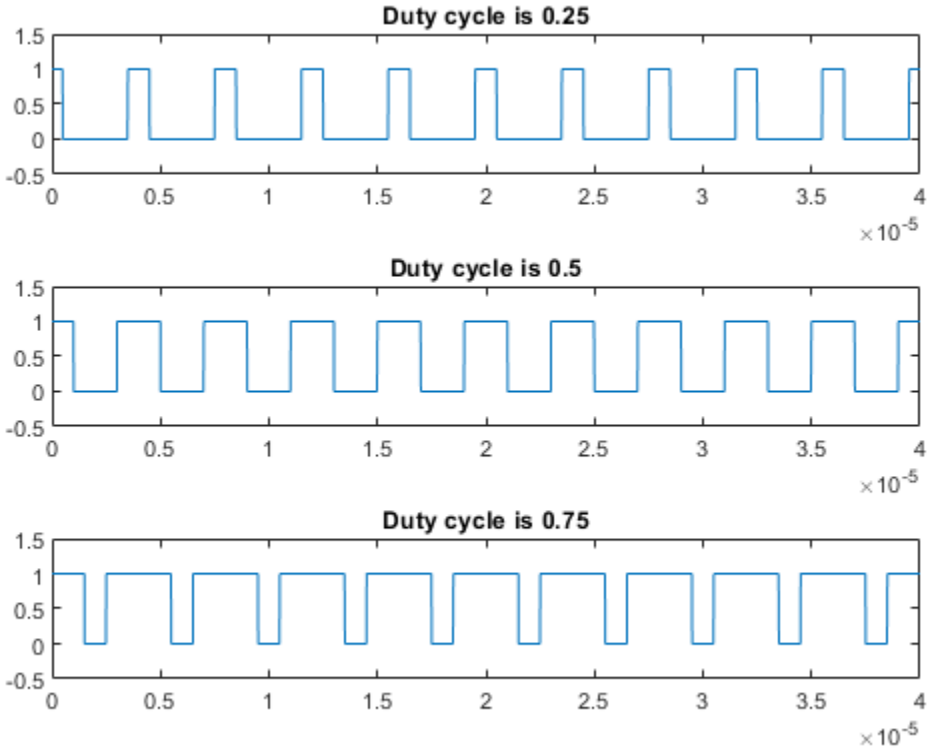
```
dutycycle(x,Fs);
```




Using the same sample rate and pulse period, vary the pulse on time (pulse width) from 1 to 3 microseconds in a loop and calculate the duty cycle. Plot the pulse waveforms and display the duty cycle value in the plot title for each step through the loop. The duty cycle increases from 0.25 (1/4) to 0.75 (3/4) as the pulse width increases.

```
nwid = 3;

for nn = 1:nwid
    x = pulstran(t,pulseperiods,@rectpuls,nn*pulsewidth);

    subplot(nwid,1,nn)
    plot(t,x)
```

```
    axis([0 4e-5 -0.5 1.5])

    D = dutycycle(x,Fs);
    title(['Duty cycle is ' num2str(mean(D))])
end
```




## See Also

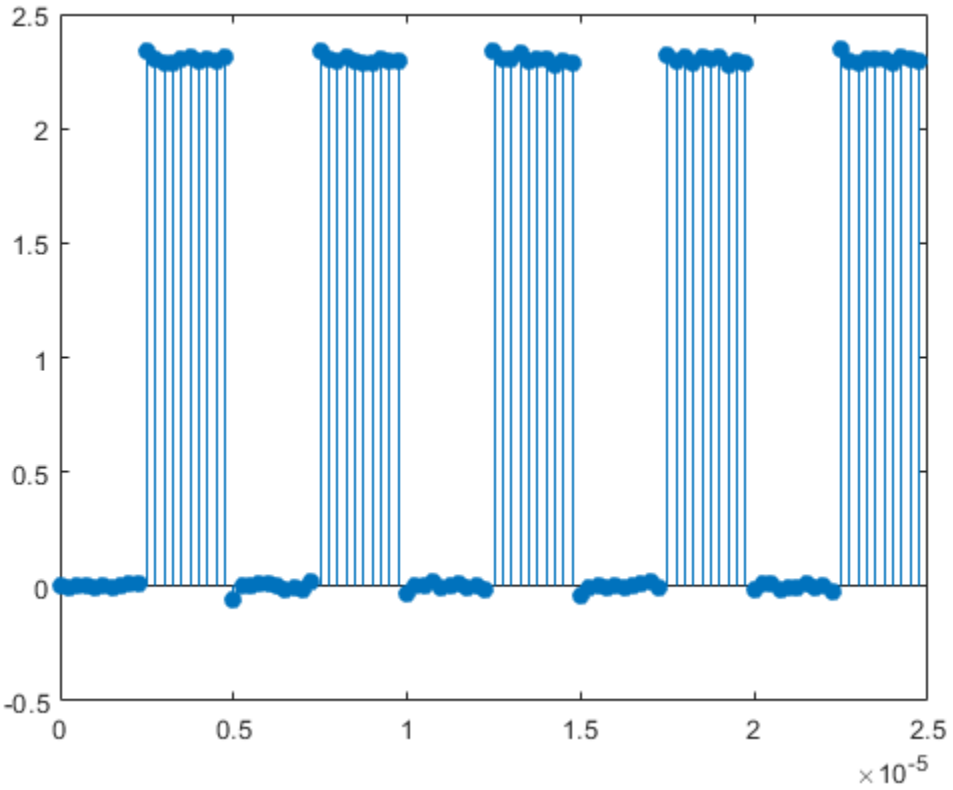
`dutycycle` | `pulstran`

# Estimate State for Digital Clock

This example shows how to estimate the high and low state levels for digital clock data. In contrast to analog voltage signals, signals in digital circuits have only two states: HIGH and LOW. Information is conveyed by the pattern of high and low state levels.

Load `clockex.mat` into the MATLAB® workspace. `clockex.mat` contains a 2.3 volt digital clock waveform sampled at 4 megahertz. Load the clock data into the variable `x` and the vector of sampling times in the variable `t`. Plot the data.

```
load('clockex.mat','x','t')
stem(t,x,'filled')
```




Determine the high and low state levels for the clock data using `statelevels`.

```
levels = statelevels(x)
```

```
levels = 1×2

    0.0027    2.3068
```

This is the expected result for the 2.3 volt clock data, where the noise-free low-state level is 0 volts and the noise-free high-state level is 2.3 volts.

Use the estimated state levels to convert the voltages into a sequence of zeros and ones. The sequence of zeros and ones is a binary waveform representation of the two states. To make the assignment, use the following decision rule:

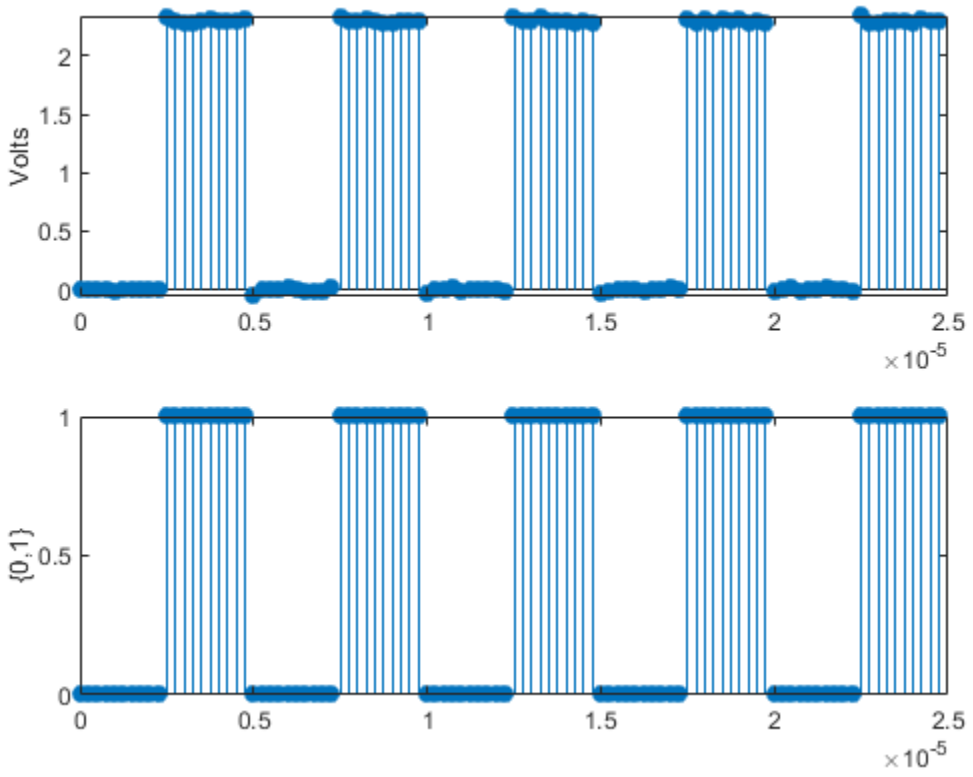
- Assign any voltage within a 3%-tolerance region of the low-state level the value 0.
- Assign any voltage within a 3%-tolerance region of the high-state level the value 1.

Determine the widths of the 3%-tolerance regions around the low- and high-state levels.

```
tolwd = 3/100*diff(levels);
```

Use logical indexing to determine the voltages within a 3%-tolerance region of the low-state level and the voltages within a 3%-tolerance region of the high-state level. Assign the value 0 to the voltages within the tolerance region of the low-state level and 1 to the voltages within the tolerance region of the high-state level. Plot the result.

```
y = zeros(size(x));
y(abs(x-min(levels))<=tolwd) = 0;
y(abs(x-max(levels))<=tolwd) = 1;

subplot(2,1,1)
stem(t,x,'filled')
ylabel('Volts')
subplot(2,1,2)
stem(t,y,'filled')
ylabel('\{0,1\}')
```



The decision rule has assigned all the voltages to the correct state.

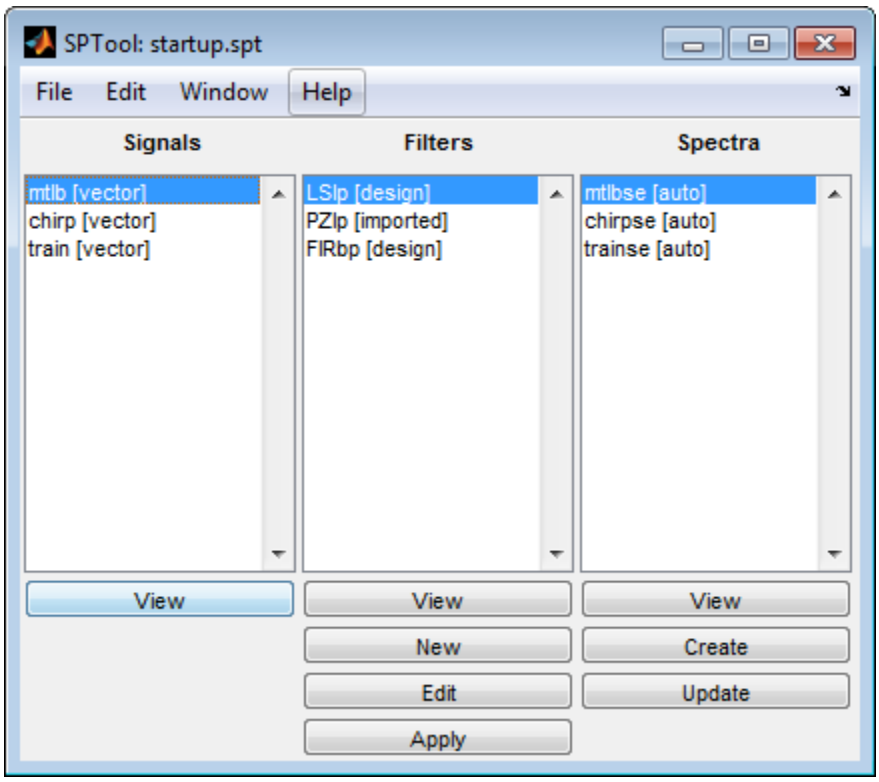
# Calculate Settling Time with Signal Browser

This example shows how to use the Bilevel Measurements panel in the SPTool Signal Browser to find the settling time of a clock signal.

First, open SPTool by typing the following at the MATLAB command line.

```
sptool
```

SPTool opens.




In this example, you import a clock signal from the MAT-file named `clockex.mat`. In the SPTool menu, select **File** > **Import**. Alternatively, you can press the **Ctrl+I** keyboard shortcut. The Import to SPTool dialog box opens.

1. Under **Source**, click **From Disk**.
2. In the **MAT-file Name** box, type `clockex`, and press **Enter**. The variables `x` and `t` appear under **File Contents**.
3. Under **File Contents**, click `x`. Click the right arrow ( --> ) button to the left of the **Data** box.
4. In the **Sampling Frequency** box, type `4000000`.
5. In the **Name** box, type `Clock`.
6. Click **OK**. The SPTool **Signals** list now contains a signal named `Clock [vector]`.
7. In SPTool, in the **Signals** list, select `Clock [vector]`, and click the **View** button. The Signal Browser appears and displays the clock signal.

Because you selected only one signal, the legend is not needed. Turn off the legend by clicking the

Show All Legends ( ) button. Then, use the Style dialog box to modify the appearance of the axes and the lines for the signal. In the Signal Browser menu, select **View > Style**.

| Parameter | Display 1 Setting |
|---|---|
| **Axes background color** | Black |
| **Ticks, labels, and grid colors** | Dark Gray |
| **Line color** | Yellow |

To show the **Bilevel Measurements** panel, in the Signal Browser menu, select **Tools > Measurements > Bilevel Measurements**. To collapse the **Transitions** pane, click the pane collapse button ( ) next to that label. To expand the **Settings** pane and the **Overshoots / Undershoots** pane, click the pane expand button ( ) next to each label.

The value for the rising edge **Settling Time** parameter does not appear in the **Overshoots / Undershoots** pane because the **Settle Seek** parameter is too large. The **Settle Seek** value is longer than the entire simulation duration. Enter a value for settle seek of `2e-6`, and press **Enter**. Signal Browser now displays a rising edge settling time value of `118.392 ns`.

This settling time value displayed is actually the statistical average of the settling times for all five rising edges. To display the settling time for only one rising edge, you can zoom in on that transition.

In the Signal Browser toolbar, click the Zoom X button ( ). Click the display near a value of 2 microseconds on the *time*-axis. Drag to the right, and release near a value of 4 microseconds on the *time*-axis. Signal Browser updates the rising edge **Settling Time** value to reflect the new time window, as shown in the following figure.

## See Also

`falltime` | `overshoot` | `risetime` | `settlingtime` | `slewrate` | `undershoot`

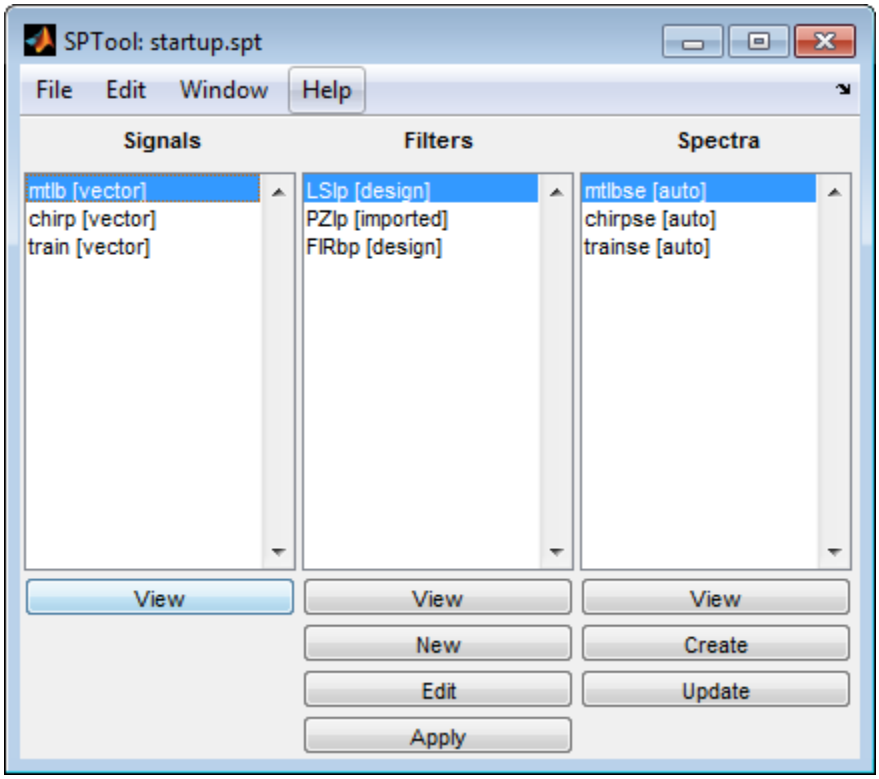
# Find Peak Amplitudes in Signal Browser

This example shows how to use the Peak Finder panel in the SPTool Signal Browser to find heart rate, given an electrocardiogram (ECG) signal.

First, open SPTool by typing the following at the MATLAB command line.

```
sptool
```

SPTool opens.




Using SPTool, you can import signals from variables in the MATLAB workspace. First, create an electrocardiogram (ECG) signal, sampled at 4 kHz. To create the signal, save this function definition in a file called `ecg.m`:

```
function x = ecg(L)
a0 = [0,1,40,1,0,-34,118,-99,0,2,21,2,0,0,0];
d0 = [0,27,59,91,131,141,163,185,195,275,307,339,357,390,440];
a = a0 / max(a0);
d = round(d0 * L / d0(15));
d(15) = L;
for i = 1:14,
    m = d(i) : d(i+1) - 1
    slope = (a(i+1) - a(i)) / (d(i+1) - d(i))
    x(m+1) = a(i) + slope * (m - d(i))
end
```

Now apply the Savitzky-Golay filter to the ECG signal. At the MATLAB command line, enter the following commands:

```
x1 = 3.5*ecg(2700).';
y1 = sgolayfilt(kron(ones(1,13),x1),0,21);
n = (1:30000)';
del = round(2700*rand(1));
mhb = y1(n + del);
ts = 0.00025;
Fs = 1/ts;
```

To import the signal from these variables, in the SPTool menu, select **File > Import**. Alternatively, you can press the **Ctrl+I** keyboard shortcut. The Import to SPTool dialog box appears.

1. Under **Workspace Contents**, click `mhb`. Click the right arrow ( --> ) button to the left of the **Data** box.
2. Under **Workspace Contents**, click `Fs`. Click the right arrow ( --> ) button to the left of the **Sampling Frequency** box.
3. In the **Name** box, type `ECG`.
4. Click **OK**. The SPTool **Signals** list now contains a signal named `ECG [vector]`.
5. In SPTool, in the **Signals** list, select `ECG [vector]`, and click the **View** button. The Signal Browser opens and displays the ECG signal.

Because you only selected one signal, the legend is not needed. Turn off the legend by clicking the

Show All Legends ( ) button. Then, use the Style dialog box to modify the appearance of the axes and the lines for the signal. In the Signal Browser menu, select **View > Style**.

| Parameter | Display 1 Setting |
|---|---|
| **Axes background color** | Black |
| **Ticks, labels, and grid colors** | Dark Gray |
| **Line color** | Yellow |

To show the **Peak Finder** panel, in the Signal Browser menu, select **Tools > Measurements > Peak Finder**. To expand the **Settings** pane, click the pane expand button ( ▶ ) next to that label. In the **Max Num of Peaks** box, type `10` and press the **Enter** key. Signal Browser now displays in the **Peaks** pane a list of 10 peak amplitude values, and the times at which they occur.

From the list of peak values, there is a constant time difference of 0.675 seconds between each heartbeat. Therefore, the heart rate detected by the ECG signal is given by the following equation.

$$\frac{60\frac{\text{sec}}{\text{min}}}{0.675\frac{\text{sec}}{\text{beat}}} = 88.89\frac{\text{beats}}{\text{min}}(\text{bpm})$$

## See Also

`findpeaks` | `sgolay` | `sgolayfilt`

# Distortion Measurements

Generate 2048 samples of a sinusoid of frequency 2.5 kHz sampled at 50 kHz. Add white Gaussian noise such that the signal-to-noise ratio (SNR) is 80 dB.

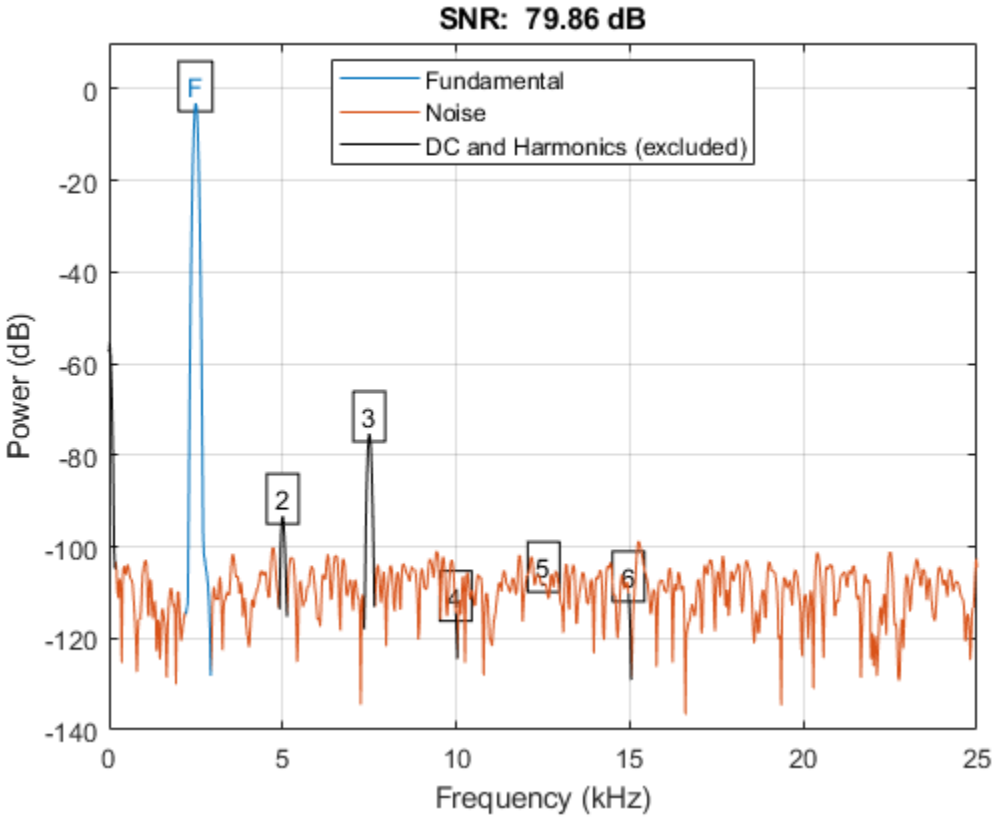
```
Fs = 5e4;
f0 = 2.5e3;
N = 2048;
t = (0:N-1)/Fs;
SNR = 80;

x = cos(2*pi*f0*t);
x = x+randn(size(x))*std(x)/db2mag(SNR);
```

Pass the result through a weakly nonlinear amplifier represented by a polynomial. The amplifier introduces spurious tones at the frequencies of the harmonics.

```
amp = [1e-5 5e-6 -1e-3 6e-5 1 25e-3];
x = polyval(amp,x);
```

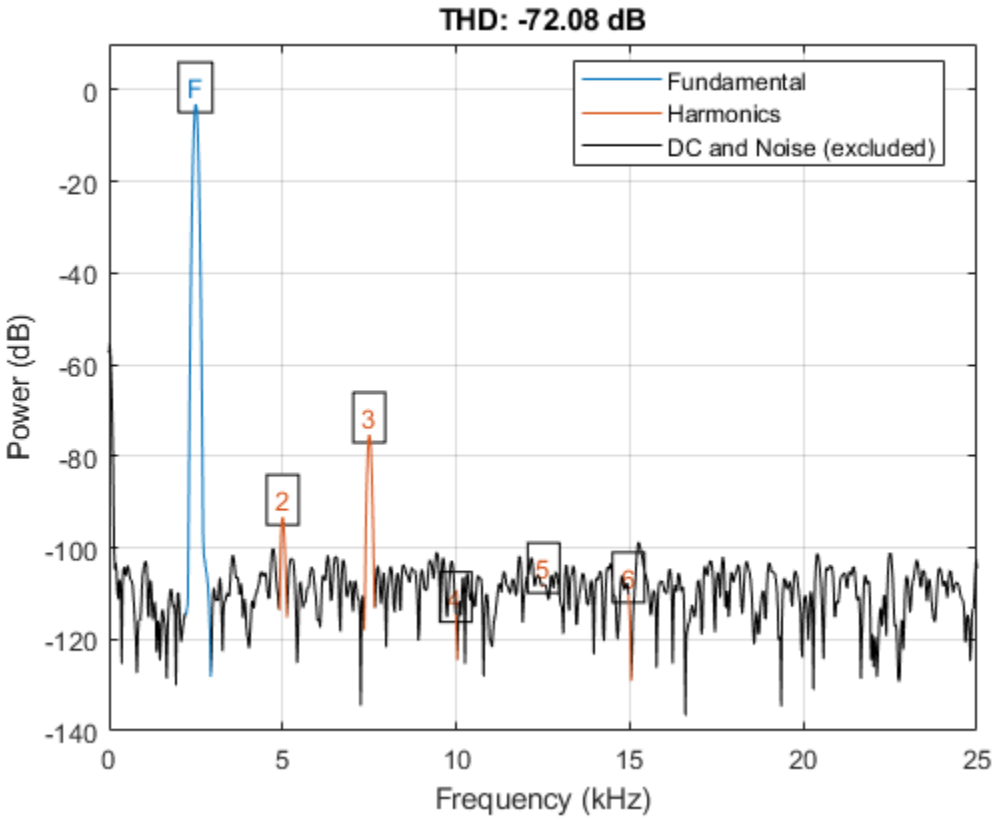
Plot the signal spectrum and annotate the SNR, verifying that it has the expected value. The `snr` function computes the power ratio of the fundamental to the noise floor and ignores the DC component and the harmonics.

```
snr(x,Fs);
```



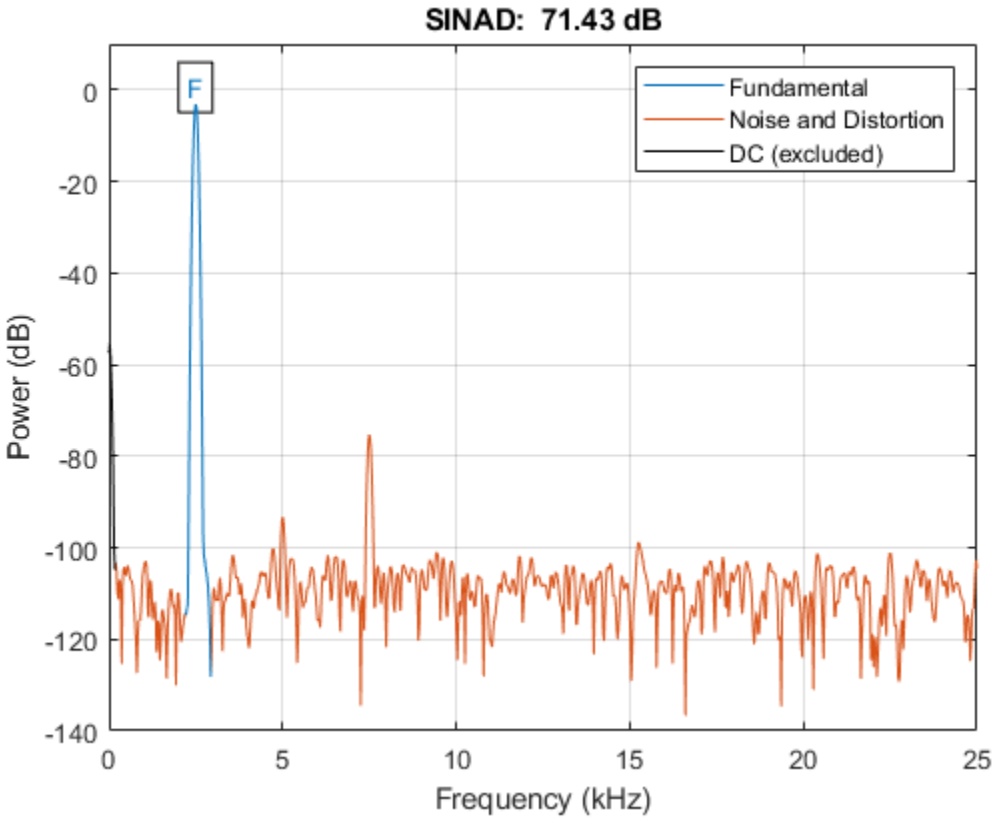
Plot the signal spectrum and annotate the total harmonic distortion (THD). The `thd` function computes the power ratio of the harmonics to the fundamental and ignores the DC component and the noise floor.

```
thd(x,Fs);
```




Plot the signal spectrum and annotate the signal to noise and distortion ratio (SINAD). The `sinad` function computes the power ratio of the fundamental to the harmonics and the noise floor. It ignores only the DC component.

```
sinad(x,Fs);
```




Verify that the SNR, THD, and SINAD obey the equation

$$10^{-\mathrm{SNR}/10} + 10^{\mathrm{THD}/10} = 10^{-\mathrm{SINAD}/10}.$$

```
lhs = 10^(-snr(x,Fs)/10)+10^(thd(x,Fs)/10)
```

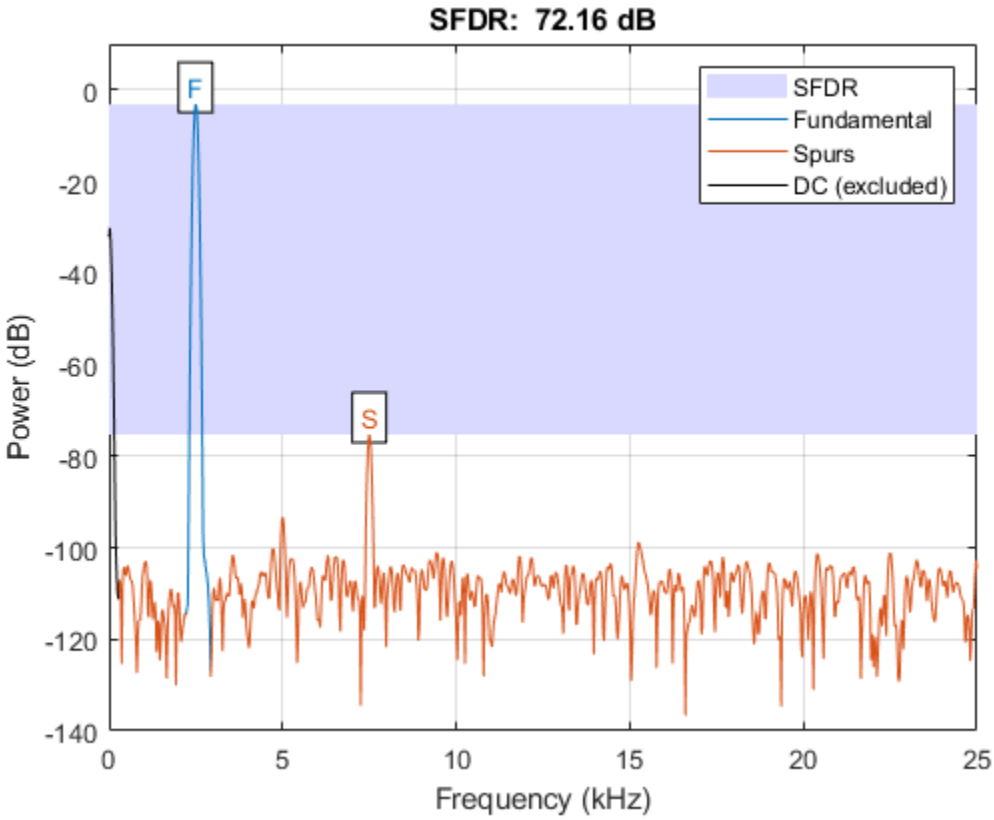
```
lhs = 7.2203e-08
```

```
rhs = 10^(-sinad(x,Fs)/10)
```

```
rhs = 7.1997e-08
```

Plot the signal spectrum and annotate the spurious-free dynamic range (SFDR). The SFDR is the power ratio of the fundamental to the strongest spurious component ("spur"). In this case, the spur corresponds to the third harmonic.

```
sfdr(x,Fs);
```




## See Also

`sfdr` | `sinad` | `snr` | `thd`

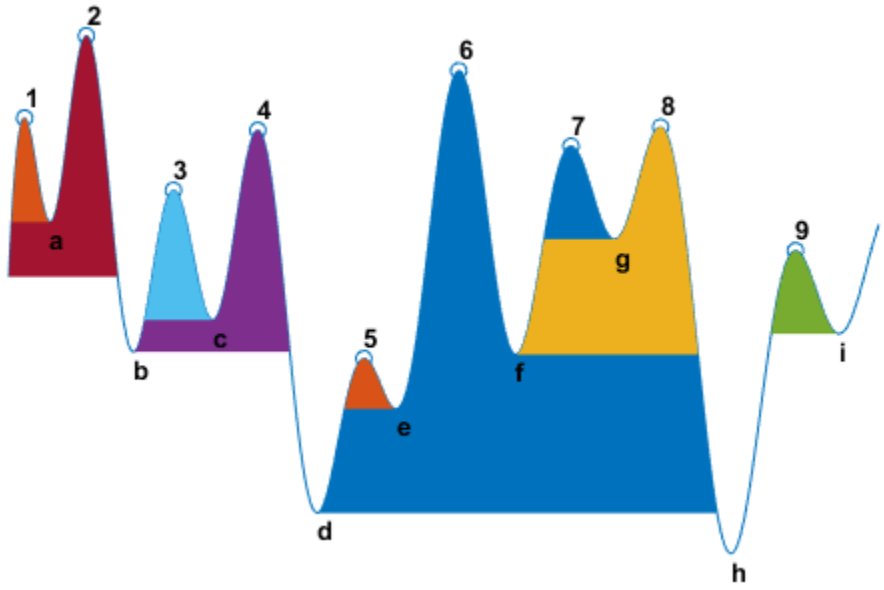
# Prominence

The *prominence* of a peak measures how much the peak stands out due to its intrinsic height and its location relative to other peaks. A low isolated peak can be more prominent than one that is higher but is an otherwise unremarkable member of a tall range.

To measure the prominence of a peak:

1. Place a marker on the peak.
2. Extend a horizontal line from the peak to the left and right until the line does one of the following:
   - Crosses the signal because there is a higher peak
   - Reaches the left or right end of the signal
3. Find the minimum of the signal in each of the two intervals defined in Step 2. This point is either a valley or one of the signal endpoints.
4. The higher of the two interval minima specifies the reference level. The height of the peak above this level is its prominence.

`findpeaks` makes no assumption about the behavior of the signal beyond its endpoints, whatever their height. This is reflected in Steps 2 and 4 and often affects the value of the reference level. Consider for example the peaks of this signal:




| Peak Number | Left Interval Lies Between Peak and | Right Interval Lies Between Peak and | Lowest Point on the Left Interval | Lowest Point on the Right Interval | Reference Level (Highest Minimum) |
|---|---|---|---|---|---|
| **1** | Left end | Crossing due to peak **2** | Left endpoint | **a** | **a** |
| **2** | Left end | Right end | Left endpoint | **h** | Left endpoint |

| Peak Number | Left Interval Lies Between Peak and | Right Interval Lies Between Peak and | Lowest Point on the Left Interval | Lowest Point on the Right Interval | Reference Level (Highest Minimum) |
|---|---|---|---|---|---|
| **3** | Crossing due to peak **2** | Crossing due to peak **4** | **b** | **c** | **c** |
| **4** | Crossing due to peak **2** | Crossing due to peak **6** | **b** | **d** | **b** |
| **5** | Crossing due to peak **4** | Crossing due to peak **6** | **d** | **e** | **e** |
| **6** | Crossing due to peak **2** | Right end | **d** | **h** | **d** |
| **7** | Crossing due to peak **6** | Crossing due to peak **8** | **f** | **g** | **g** |
| **8** | Crossing due to peak **6** | Right end | **f** | **h** | **f** |
| **9** | Crossing due to peak **8** | Crossing due to right endpoint | **h** | **i** | **i** |

## See Also

`findpeaks` | `islocalmax` | `islocalmin`

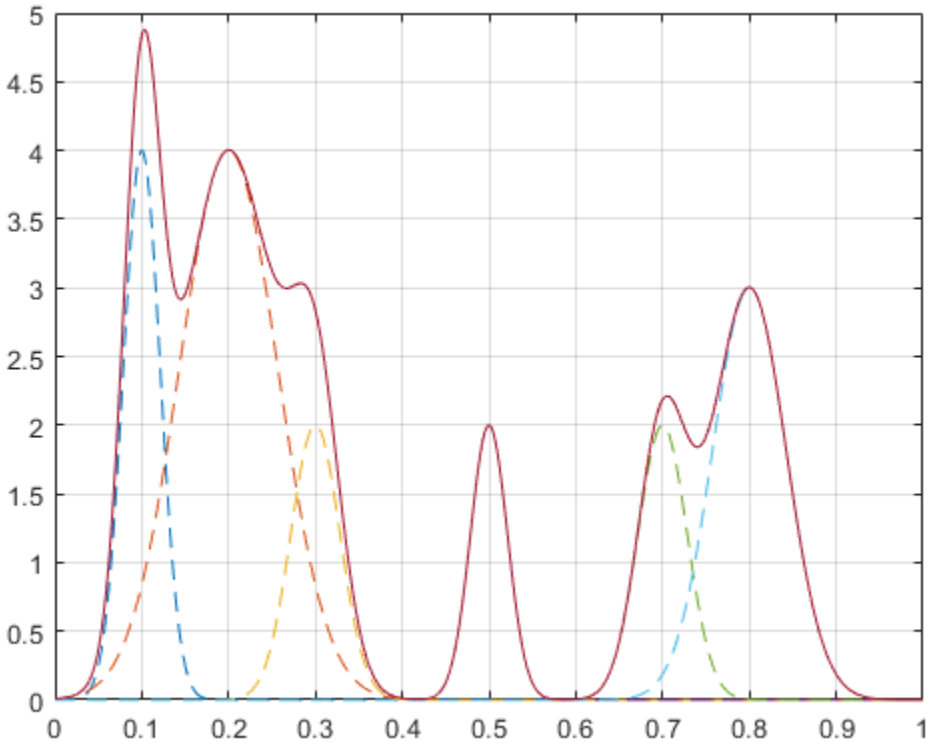
# Determine Peak Widths

Create a signal that consists of a sum of bell curves. Specify the location, height, and width of each curve.

```
x = linspace(0,1,1000);

Pos = [1 2 3 5 7 8]/10;
Hgt = [4 4 2 2 2 3];
Wdt = [3 8 4 3 4 6]/100;

for n = 1:length(Pos)
    Gauss(n,:) =  Hgt(n)*exp(-((x - Pos(n))/Wdt(n)).^2);
end

PeakSig = sum(Gauss);
```

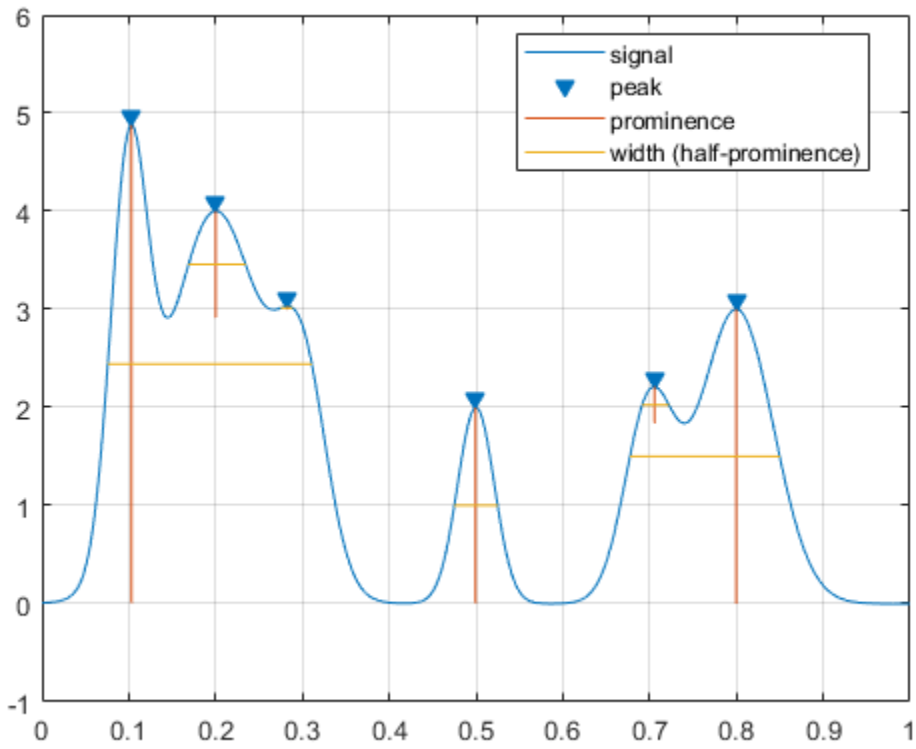
Plot the individual curves and their sum.

```
plot(x,Gauss,'--',x,PeakSig)
grid
```



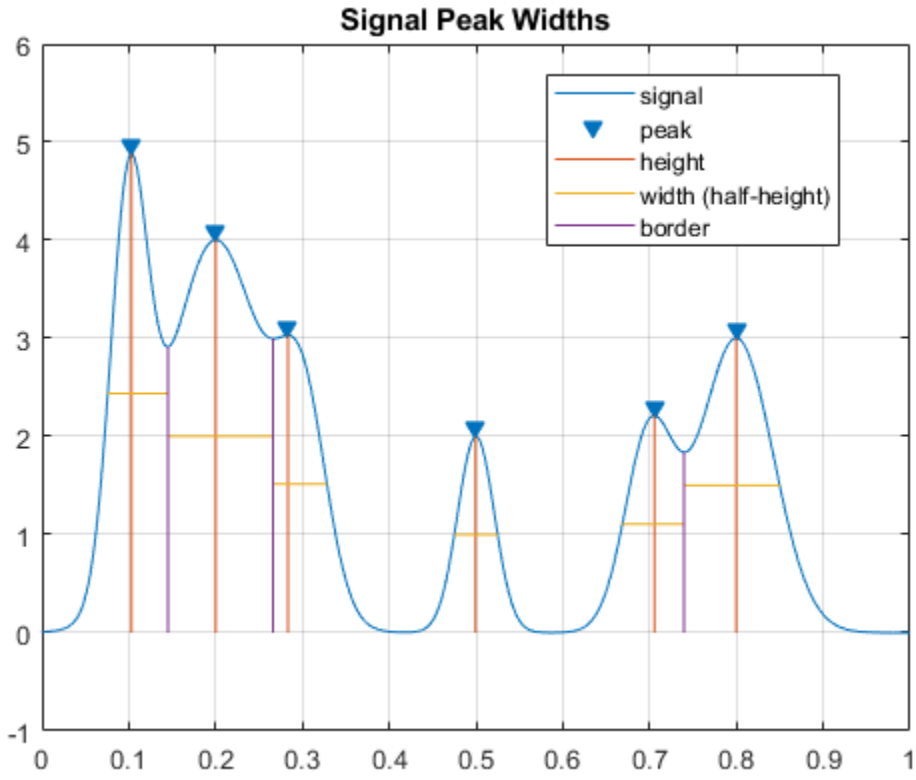

Measure the widths of the peaks using the half prominence as reference.

```
findpeaks(PeakSig,x,'Annotate','extents')
```




Measure the widths again, this time using the half height as reference.

```
findpeaks(PeakSig,x,'Annotate','extents','WidthReference','halfheight')
title('Signal Peak Widths')
```


Signal Peak Widths
signal
peak
height
width (half-height)
border
6
5
4
3
2
1
0
-1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

# Spectrum Object to Function Replacement


- “Nonparametric Spectrum Object to Function Replacement” on page 19-2
- “Autoregressive PSD Object to Function Replacement Syntax” on page 19-9
- “Subspace Pseudospectrum Object to Function Replacement Syntax” on page 19-10

# Nonparametric Spectrum Object to Function Replacement

| In this section... |
|---|
| “Periodogram PSD Object to Function Replacement Syntax” on page 19-2 |
| “Periodogram MSSPECTRUM Object to Function Replacement Syntax” on page 19-3 |
| “Welch PSD Object to Function Replacement Syntax” on page 19-4 |
| “Welch MSSPECTRUM Object to Function Replacement Syntax” on page 19-5 |
| “Multitaper PSD Object to Function Replacement Syntax” on page 19-7 |

## Periodogram PSD Object to Function Replacement Syntax

The `spectrum.periodogram` object syntax will be removed in the future. The following table gives the equivalent recommended function syntax for `periodogram`. In the modified periodogram, you use a window other than the default rectangular window. To illustrate modified periodogram syntaxes, the table uses a specific window. In each example, `x` is the input signal.

| Deprecated Syntax | Replacement Syntax |
|---|---|
| `h = spectrum.periodogram;`<br>`psd(h,x);` | `periodogram(x);` |
| `% Modified periodogram with window function`<br>`h = spectrum.periodogram('hamming');`<br>`psd(h,x);` | `win = hamming(length(x));`<br>`periodogram(x,win);` |
| `% Window function and optional input arguments to window function`<br>`h = spectrum.periodogram({'Hamming','periodic'});`<br>`psd(h,x);` | `win = hamming(length(x),'periodic');`<br>`periodogram(x,win);` |
| `% Taylor window and multiple optional input arguments`<br>`nbar = 4;`<br>`sll = 30;`<br>`h = spectrum.periodogram({'Taylor',nbar,sll});`<br>`psd(h,x,'Fs',fs,'centerdc',true);` | `nbar = 4;`<br>`sll = -30;`<br>`win = taylorwin(length(x),nbar,sll);`<br>`periodogram(x,win,[],fs,'centered');` |
| `h = spectrum.periodogram(...);`<br>`psd(h,x,'NFFT',nfft);` | `win = ...`<br>`periodogram(x,win,nfft);` |
| `h = spectrum.periodogram(...);`<br>`psd(h,x,'Fs',fs);` | `win = ...`<br>`periodogram(x,win,[],fs);` |
| `h = spectrum.periodogram(...);`<br>`psd(h,x,'NFFT',nfft,'Fs',fs);` | `win = ...`<br>`periodogram(x,win,nfft,fs);` |
| `h = spectrum.periodogram(...);`<br>`psd(h,x,...,'FreqPoints','User Defined',...`<br>`'FrequencyVector',w);` | `win = ...`<br>`periodogram(x,win,w);` |
| `h = spectrum.periodogram(...);`<br>`psd(h,x,'FreqPoints','User Defined',...`<br>`'FrequencyVector',f,'Fs',fs);` | `win = ...`<br>`periodogram(x,win,f,fs);` |
| `% Two-sided spectrum of a real signal`<br>`h = spectrum.periodogram(...);`<br>`psd(h,x,...,'SpectrumType','TwoSided');` | `win = ...`<br>`periodogram(x,win,...,'twosided');` |
| `% Two-sided spectrum with DC (0 frequency) in the center`<br>`h = spectrum.periodogram(...);`<br>`psd(h,x,...,'CenterDC',true);` | `win = ...`<br>`periodogram(x,win,...,'centered');` |

| Deprecated Syntax | Replacement Syntax |
|---|---|
| `h = spectrum.periodogram(...);`<br>`psd(h,x,...,'ConfLevel',p);` | `win = ...`<br>`periodogram(x,win,...,'ConfidenceLevel',p);` |
| `h = spectrum.periodogram(...);`<br>`hPSD = psd(h,x,...);`<br>`Pxx = hPSD.Data;`<br>`F = hPSD.Frequencies;` | `win = ...`<br>`[Pxx,F] = periodogram(x,win,...);` |
| `h = spectrum.periodogram(...);`<br>`hPSD = psd(h,x,...,'ConfLevel',p);`<br>`Pxx = hPSD.Data;`<br>`F = hPSD.Frequencies;`<br>`Pxxc = hPSD.ConfInterval;` | `win = ...`<br>`[Pxx,F,Pxxc] = periodogram(x,win,...);` |

## Periodogram MSSPECTRUM Object to Function Replacement Syntax

The `spectrum.periodogram` MSSPECTRUM object syntax will be removed in the future. The following table gives the equivalent recommended function syntax for `periodogram`. In the modified periodogram, you use a window other than the default rectangular window. To illustrate modified periodogram syntaxes, the table uses a specific window. In each example, `x` is the input signal.

| Deprecated Syntax | Recommended Syntax |
|---|---|
| `h = spectrum.periodogram;`<br>`msspectrum(h,x);` | `periodogram(x,'power');` |
| `h = spectrum.periodogram('Hamming');`<br>`msspectrum(h,x);` | `win = hamming(length(x));`<br>`periodogram(x,win,'power');` |
| `h = spectrum.periodogram({'Hamming','periodic'});`<br>`msspectrum(h,x);` | `win = hamming(length(x),'periodic');`<br>`periodogram(x,win,'power');` |
| `nbar = 4;`<br>`sll = 30;`<br>`h = spectrum.periodogram({'Taylor',nbar,sll});`<br>`msspectrum(h,x);` | `nbar = 4;`<br>`sll = -30;`<br>`win = taylorwin(length(x),nbar,sll);`<br>`periodogram(x,win,'power');` |
| `h = spectrum.periodogram(...);`<br>`msspectrum(h,x,'NFFT',nfft);` | `win= ...`<br>`periodogram(x,win,nfft,'power');` |
| `h = spectrum.periodogram(...);`<br>`msspectrum(h,x,'Fs',fs);` | `win = ...`<br>`periodogram(x,win,[],fs,'power');` |
| `h = spectrum.periodogram(...);`<br>`msspectrum(h,x,'NFFT',nfft,'Fs',fs);` | `win = ...`<br>`periodogram(x,win,nfft,fs,'power');` |
| `h = spectrum.periodogram(...);`<br>`msspectrum(h,x,...,'SpectrumType','TwoSided');` | `win = ...`<br>`periodogram(x,win,..., 'twosided','power');` |
| `h = spectrum.periodogram(...);`<br>`msspectrum(h, x,...,'CenterDC',true);` | `win = ...`<br>`periodogram(x,win,...,'centered','power');` |
| `h = spectrum.periodogram(...);`<br>`msspectrum(h,x,...,'ConfLevel',p);` | `win = ...`<br>`periodogram(x,win,...,'ConfidenceLevel', p,...'power');` |
| `h = spectrum.periodogram(...);`<br>`hMS = msspectrum(h,x,...);`<br>`Sxx = hMS.Data;`<br>`F = hMS.Frequencies;` | `win = ...`<br>`[Sxx,F] = periodogram(x,win,...,'power');` |

| Deprecated Syntax | Recommended Syntax |
|---|---|
| `h = spectrum.periodogram(...);`<br>`hMS = msspectrum(h,x,...,'ConfLevel',p);`<br>`Sxx = hMS.Data;`<br>`F = hMS.Frequencies;`<br>`Sxxc = hMS.ConfInterval;` | `win = ...`<br>`[Sxx,F,Sxxc] = periodogram(x,win,...,'power');` |

## Welch PSD Object to Function Replacement Syntax

The `spectrum.welch` object syntax will be removed in the future. The following table gives the equivalent recommended function syntax for `pwelch`. To illustrate modified periodogram syntaxes, the table uses a specific window. In each example, `x` is the input signal.

| Deprecated Syntax | Replacement Syntax |
|---|---|
| `h = spectrum.welch;`<br>`psd(h,x);` | `pwelch(x);` |
| `h = spectrum.welch('Gaussian');`<br>`psd(h,x);` | `win = gausswin(64);`<br>`pwelch(x,win);` |
| `% Welch estimate with window function and optional input arguments`<br>`h = spectrum.welch({'Hamming','periodic'});`<br>`psd(h,x);` | `win = hamming(64,'periodic');`<br>`pwelch(x,win);` |
| `% Taylor window and multiple optional input arguments`<br>`nbar = 4;`<br>`sll = 30;`<br>`h = spectrum.welch({'Taylor', nbar, sll});`<br>`psd(h,x);` | `nbar = 4;`<br>`sll = -30;`<br>`win = taylorwin(64,nbar,sll);`<br>`pwelch(x,win);` |
| `h = spectrum.welch('Hamming',segLen);`<br>`psd(h,x);` | `win = hamming(segLen);`<br>`pwelch(x,win);` |
| `h = spectrum.welch({'Hamming','periodic'},...`<br>`segLen);`<br>`psd(h,x);` | `win = hamming(segLen,'periodic');`<br>`pwelch(x,win);` |
| `nbar = 4;`<br>`sll = 30;`<br>`h = spectrum.welch({'Taylor',nbar,sll},...`<br>`segLen);`<br>`psd(h,x);` | `nbar = 4;`<br>`sll = -30;`<br>`win = taylorwin(segLen,nbar,sll);`<br>`pwelch(x,win);` |
| `h = spectrum.welch('Hamming',segLen,ovlpPct);`<br>`psd(h,x);` | `win = hamming(segLen);`<br>`Noverlap = ceil((ovlpPct/100)*segLen);`<br>`pwelch(x,win,Noverlap);` |
| `h = spectrum.welch({'Hamming','periodic'},...`<br>`segLen,ovlpPct);`<br>`psd(h,x);` | `win = hamming(segLen,'periodic');`<br>`Noverlap = ceil((ovlpPct/100)*segLen);`<br>`pwelch(x,win,Noverlap);` |
| `nbar = 4;`<br>`sll = 30;`<br>`h = spectrum.welch({'Taylor',nbar,sll},...`<br>`segLen,ovlpPct);`<br>`psd(h,x);` | `nbar = 4;`<br>`sll = -30;`<br>`win = taylorwin(segLen,nbar,sll);`<br>`Noverlap = ceil((ovlpPct/100)*segLen);`<br>`pwelch(x,win,Noverlap);` |
| `h = spectrum.welch(...);`<br>`psd(h,x,'NFFT',nfft);` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,nfft);` |

| Deprecated Syntax | Replacement Syntax |
|---|---|
| `h = spectrum.welch(...);`<br>`psd(h,x,'Fs',fs);` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,[],fs);` |
| `h = spectrum.welch(...);`<br>`psd(h,x,'NFFT',nfft,'Fs',fs);` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,nfft,fs);` |
| `h = spectrum.welch(...);`<br>`psd(h,x,...,'FreqPoints','User Defined',...`<br>`'FrequencyVector',w);` | `win = ...`<br>`periodogram(x,win,w);` |
| `h = spectrum.periodogram(...);`<br>`psd(h,x,'FreqPoints','User Defined',...`<br>`'FrequencyVector',f,'Fs',fs);` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,f,fs);` |
| `% Two-sided spectrum of a real signal`<br>`h = spectrum.welch(...);`<br>`psd(h,x,...,'SpectrumType','TwoSided');` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,...,'twosided');` |
| `% Two-sided spectrum with DC (0 frequency) in the center`<br>`h = spectrum.welch(...);`<br>`psd(h,x,...,'CenterDC',true);` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,...,'centered');` |
| `h = spectrum.welch(...);`<br>`psd(h,x,...,'ConfLevel',p);` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,...'ConfidenceLevel',p);` |
| `h = spectrum.welch(...);`<br>`hPSD = psd(h,x,...);`<br>`Pxx = hPSD.Data;`<br>`F = hPSD.Frequencies;` | `win = ...`<br>`Noverlap = ...`<br>`[Pxx,F] = pwelch(x,win,Noverlap,...);` |
| `h = spectrum.periodogram(...);`<br>`hPSD = psd(h,x,...,'ConfLevel',p);`<br>`Pxx = hPSD.Data;`<br>`F = hPSD.Frequencies;`<br>`Pxxc = hPSD.ConfInterval;` | `win = ...`<br>`Noverlap = ...`<br>`[Pxx,F,Pxxc] = pwelch(x,win,Noverlap,...`<br>`'ConfidenceLevel',p);` |

## Welch MSSPECTRUM Object to Function Replacement Syntax

The `spectrum.welch` MSSPECTRUM object syntax will be removed in the future. The following table gives the equivalent recommended function syntax for `pwelch`. In the modified periodogram, you use a window other than the default rectangular window. To illustrate modified periodogram syntaxes, the table uses a specific window. In each example, `x` is the input signal.

| Deprecated Syntax | Recommended Syntax |
|---|---|
| `h = spectrum.welch`<br>`msspectrum(h,x);` | `win = hamming(64);`<br>`pwelch(x,win,[],'power');` |
| `h = spectrum.welch('Gaussian');`<br>`msspectrum(h,x);` | `win = gausswin(64);`<br>`pwelch(x,win,[],'power');` |
| `h = spectrum.welch({'Hamming','periodic'});`<br>`msspectrum(h,x);` | `win = hamming(64,'periodic');`<br>`pwelch(x,win,[],'power');` |
| `nbar = 4;`<br>`sll = 30;`<br>`h = spectrum.welch({'Taylor',nbar,sll});`<br>`msspectrum(h,x);` | `nbar = 4;`<br>`sll = -30;`<br>`win = taylorwin(64,nbar,sll);`<br>`pwelch(x,win,[],'power');` |

| Deprecated Syntax | Recommended Syntax |
|---|---|
| `segLen = 128;`<br>`h = spectrum.welch('Hamming',segLen);`<br>`msspectrum(h,x);` | `win = hamming(128);`<br>`pwelch(x,win,[],'power');` |
| `segLen = 128;`<br>`h = spectrum.welch({'Hamming','periodic'},...`<br>`segLen);`<br>`msspectrum(h,x);` | `win = hamming(128,'periodic');`<br>`pwelch(x,win,[],'power');` |
| `nbar = 4;`<br>`sll = 30;`<br>`segLen = 128;`<br>`h = spectrum.welch({'Taylor',nbar,sll},segLen);`<br>`msspectrum(h,x);` | `nbar = 4;`<br>`sll = -30;`<br>`segLen = 128;`<br>`win = taylorwin(segLen,nbar,sll);`<br>`pwelch(x,win,[],'power');` |
| `segLen = 128;`<br>`ovlpPct = 50;`<br>`h = spectrum.welch('Hamming',segLen,ovlpPct);`<br>`msspectrum(h,x);` | `segLen = 128;`<br>`win = hamming(segLen);`<br>`ovlpPct = 50;`<br>`Noverlap = ceil((ovlpPct/100)*segLen);`<br>`pwelch(x,win,Noverlap,'power');` |
| `segLen = 128;`<br>`ovlpPct = 50;`<br>`h = spectrum.welch({'Hamming','periodic'},...`<br>`segLen,ovlpPct);`<br>`msspectrum(h,x);` | `segLen = 128;`<br>`ovlpPct = 50;`<br>`win = hamming(segLen,'periodic');`<br>`Noverlap = ceil((ovlpPct/100)*segLen);`<br>`pwelch(x,win,Noverlap,'power');` |
| `nbar = 4;`<br>`sll = 30;`<br>`segLen = 128;`<br>`ovlpPct = 50;`<br>`h = spectrum.welch({'Taylor',nbar,sll},...`<br>`segLen,ovlpPct);`<br>`msspectrum(h,x);` | `nbar = 4;`<br>`sll = -30;`<br>`segLen = 128;`<br>`win = taylorwin(segLen,nbar,sll);`<br>`ovlpPct = 50;`<br>`Noverlap = ceil((ovlpPct/100)*segLen);`<br>`pwelch(x,win,Noverlap,'power');` |
| `h = spectrum.welch(...);`<br>`msspectrum(h,x,'NFFT',nfft);` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,nfft,'power');` |
| `h = spectrum.welch(...);`<br>`msspectrum(h,x,'Fs',fs);` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,[],fs,'power');` |
| `h = spectrum.welch(...);`<br>`msspectrum(h,x,'NFFT',nfft,'Fs',fs);` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,nfft,fs,'power');` |
| `h = spectrum.welch(...);`<br>`msspectrum(h, x,...,'FreqPoints','User Defined',`<br>`'FrequencyVector',w);` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,f,fs,'power');` |
| `h = spectrum.welch(...);`<br>`msspectrum(h,x,...,'SpectrumType','TwoSided');` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,...,'twosided','power');` |
| `h = spectrum.welch(...);`<br>`msspectrum(h,x,...,'CenterDC',true);` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,...,'centered','power');` |
| `h = spectrum.welch(...);`<br>`msspectrum(h,x,...,'ConfLevel',p);` | `win = ...`<br>`Noverlap = ...`<br>`pwelch(x,win,Noverlap,...,'ConfidenceLevel',p,'power');` |

| Deprecated Syntax | Recommended Syntax |
|---|---|
| `h = spectrum.welch(...);`<br>`hMS = msspectrum(h,x,...);`<br>`Sxx = hMS.Data;`<br>`F = hMS.Frequencies;` | `[Sxx,F] = pwelch(...,'power');` |
| `h = spectrum.welch(...);`<br>`hMS = msspectrum(h, x, …, 'ConfLevel', p);`<br>`Sxx = hMS.Data;`<br>`F = hMS.Frequencies;`<br>`Sxxc = hMS.ConfInterval;` | `[Sxx,F,Sxxc] = pwelch(...,'ConfidenceLevel',p,'power');` |

## Multitaper PSD Object to Function Replacement Syntax

The `spectrum.mtm` object syntax will be removed in the future. The following table gives the equivalent recommended function syntax for `pmtm`. In each example, `x` is the input signal.

| Deprecated Syntax | Recommended Syntax |
|---|---|
| `hMTM = spectrum.mtm;`<br>`psd(hMTM,x);` | `pmtm(x,4);` |
| `hMTM = spectrum.mtm(NW);`<br>`psd(hMTM,x);` | `pmtm(x,NW);` |
| `[E,V] = dpss(length(x),NW);`<br>`hMTM = spectrum.mtm(E,V);`<br>`psd(hMTM,x);` | `[E,V] = dpss(length(x),NW);`<br>`pmtm(x,E,V);` |
| `hMTM = spectrum.mtm(NW);`<br>`psd(hMTM,x,'Fs',fs);` | `pmtm(x,NW,fs);` |
| `hMTM = spectrum.mtm(E,V);`<br>`psd(hMTM,x,'Fs',fs);` | `pmtm(x,E,V,fs);` |
| `hMTM = spectrum.mtm(NW);`<br>`psd(hMTM,x,'Fs',fs,'NFFT',nfft);` | `pmtm(x,NW,nfft,fs);` |
| `hMTM = spectrum.mtm(E,V);`<br>`psd(hMTM,x,'Fs',fs,'NFFT',nfft);` | `pmtm(x,E,V,nfft,fs);` |
| `hMTM = spectrum.mtm(NW);`<br>`psd(hMTM,x,'FreqPoints','User Defined',...`<br>`'FrequencyVector',w);` | `pmtm(x,NW,w);` |
| `hMTM = spectrum.mtm(E,V);`<br>`psd(hMTM,x,'FreqPoints','User Defined',...`<br>`'FrequencyVector',w);` | `pmtm(x,E,V,w);` |
| `hMTM = spectrum.mtm(NW);`<br>`psd(hMTM,x,'FreqPoints','User Defined',...`<br>`'FrequencyVector',f,'Fs',fs);` | `pmtm(x,E,V,f,fs);` |
| `hMTM = spectrum.mtm(E,V);`<br>`psd(hMTM,x,'FreqPoints','User Defined',...`<br>`'FrequencyVector',f,'Fs',fs);` | `pmtm(x,E,V,f,fs);` |
| `hMTM = spectrum.mtm(...,'Adaptive');`<br>`psd(hMTM,...);` | `pmtm(...,'adapt');` |
| `hMTM = spectrum.mtm(...,'Eigenvalue');`<br>`psd(hMTM,...);` | `pmtm(...,'eigen');` |

| Deprecated Syntax | Recommended Syntax |
|---|---|
| `hMTM = spectrum.mtm(...,'Unity');`<br>`psd(hMTM,...);` | `pmtm(...,'unity');` |
| `hMTM = spectrum.mtm(...);`<br>`psd(hMTM,...,'SpectrumType','twosided');` | `pmtm(...,'twosided');` |
| `hMTM = spectrum.mtm(...);`<br>`psd(hMTM,...,'SpectrumType','twosided',...`<br>`'CenterDC',true);` | `pmtm(...,'centered');` |
| `hMTM = spectrum.mtm(...);`<br>`psd(hMTM,...,'ConfLevel',p);` | `pmtm(...,'ConfidenceLevel',p);` |
| `hMTM = spectrum.mtm(...);`<br>`hPSD = psd(hMTM,...);`<br>`Pxx = hPSD.Data;`<br>`F = hPSD.Frequencies;` | `[Pxx,F] = pmtm(...);` |
| `hMTM = spectrum.mtm(...);`<br>`hPSD = psd(hMTM,x,'ConfLevel',p);`<br>`Pxx = hPSD.Data;`<br>`F = hPSD.Frequencies;`<br>`Pxxc = hPSD.ConfInterval;` | `[Pxx,F,Pxxc] = pmtm(x,'ConfidenceLevel',p);` |

# Autoregressive PSD Object to Function Replacement Syntax

The AR PSD object syntax will be removed in the future. The following table gives the equivalent recommended function syntax. The table uses `spectrum.burg` and `pburg` as examples, but the object-to-function replacement syntaxes are valid for all the AR spectral estimators with the appropriate substitution: `spectrum.burg` to `pburg`, `spectrum.cov` to `pcov`, `spectrum.mcov` to `pmcov`, and `spectrum.yulear` to `pyulear`. In each example, `x` is the input signal.

| Deprecated Syntax | Replacement Syntax |
|---|---|
| `hBurg = spectrum.burg;`<br>`psd(hBurg,x);` | `pburg(x,4);` |
| `hBurg = spectrum.burg(order);`<br>`psd(hBurg,x);` | `pburg(x,order);` |
| `hBurg = spectrum.burg(order);`<br>`psd(hBurg,x,'NFFT',nfft);` | `pburg(x,order,nfft);` |
| `hBurg = spectrum.burg(order);`<br>`psd(hBurg,x,'Fs',fs);` | `pburg(x,order,[],fs);` |
| `hBurg = spectrum.burg(order);`<br>`psd(hBurg,x,'NFFT',nfft,'Fs',fs);` | `pburg(x,order,nfft,fs);` |
| `hBurg = spectrum.burg(order);`<br>`psd(hBurg, x,...,'FreqPoints','User Defined',...`<br>`'FrequencyVector',w);` | `pburg(x,order,w);` |
| `hBurg = spectrum.burg(order);`<br>`psd(hBurg,x,'FreqPoints','User Defined',...`<br>`'FrequencyVector',f,'Fs',fs);` | `pburg(x,order,f,fs);` |
| `hBurg = spectrum.burg`<br>`psd(...,'SpectrumType','TwoSided');` | `pburg(...,'twosided');` |
| `hBurg = spectrum.burg;`<br>`psd(...,'CenterDC',true);` | `pburg(x,...,'centered');` |
| `hBurg = spectrum.burg;`<br>`psd(...,'ConfLevel',p);` | `pburg(x,...,'ConfidenceLevel',p);` |
| `hBurg = spectrum.burg;`<br>`hPSD = psd(...);`<br>`Pxx = hPSD.Data;`<br>`F = hPSD.Frequencies;` | `[Pxx,F] = pburg(...);` |
| `hBurg = spectrum.burg;`<br>`hPSD = psd(...,'ConfLevel',p);`<br>`Pxx = hPSD.Data;`<br>`F = hPSD.Frequencies;`<br>`Pxxc = hPSD.ConfInterval;` | `[Pxx,F,Pxxc] = pburg(...);` |

# Subspace Pseudospectrum Object to Function Replacement Syntax

The pseudospectrum object syntax will be removed in the future. The following table gives the equivalent recommended function syntax. The table uses `spectrum.music` and the functional equivalent, `pmusic`, but the syntax replacements are also valid for `spectrum.eigenvector` to `peig`. In each example, `x` is the input signal.

| Deprecated Syntax | Replacement Syntax |
|---|---|
| `h = spectrum.music(nsinusoids);`<br>`pseudospectrum(h,x);` | `pmusic(x,nsinusoids)` |
| `h = spectrum.music(nsinusoids);`<br>`pseudospectrum(h,x,'Fs',fs)` | `pmusic(x,nsinusoids,[],fs);` |
| `h = spectrum.music(nsinusoids,segLen,ovlpPct,...`<br>`'Hamming');`<br>`pseudospectrum(h,x)` | `win = hamming(segLen)`<br>`Noverlap = ceil(ovlpPct/100*segLen);`<br>`P = nsinusoids;`<br>`Fs = 2*pi;`<br>`pmusic(x,P,[],Fs,win,Noverlap);` |
| `h = spectrum.music(nsinusoids,segLen,ovlpPct,...`<br>`winName,thresh);`<br>`pseudospectrum(h,x)` | `win = winfunc(segLen)`<br>`Noverlap = ceil(ovlpPct/100*segLen);`<br>`P = [nsinusoids thresh];`<br>`Fs = 2 *pi;`<br>`pmusic(x,P,[],Fs,win,Noverlap);` |
| `h = spectrum.music(nsinusoids,segLen,...`<br>`ovlpPct,winName,thresh);`<br>`pseudospectrum(h,x,'Fs',fs)` | `win = hamming(segLen)`<br>`nfft = max(256,2^nexpow2(segLen));`<br>`Noverlap = ceil(ovlpPct/100*segLen);`<br>`P = [nsinusoids thresh];`<br>`pmusic(x,P,[],Fs,win,Noverlap);` |
| `h = spectrum.music(nsinusoids,segLen,...`<br>`ovlpPct,winName,thresh);`<br>`pseudospectrum(h,x,'Fs',fs,'SpectrumRange',range)` | `win = hamming(segLen)`<br>`Noverlap = ceil(ovlpPct/100*segLen);`<br>`P = [nsinusoids thresh];`<br>`pmusic(x,P,[],Fs,range,win,Noverlap);` |
| `h = spectrum.music(nsinusoids,segLen,...`<br>`ovlpPct,winName,thresh);`<br>`pseudospectrum(h,x,'Fs',fs,'SpectrumRange',range,'NFFT',nfft)` | `win = hamming(segLen)`<br>`Noverlap = ceil(ovlpPct/100*segLen);`<br>`P = [nsinusoids thresh];`<br>`pmusic(x,P,nfft,Fs,range,win,Noverlap);` |
| `h = spectrum.music(nsinusoids,segLen,...`<br>`ovlpPct,winName,thresh);`<br>`pseudospectrum(h,x,...,'FreqPoints','User Defined',...`<br>`'Frequency Vector',fVec)` | `win = hamming(segLen)`<br>`Noverlap = ceil(ovlpPct/100*segLen);`<br>`P = [nsinusoids thresh];`<br>`pmusic(x,P,fVec,Fs,range,win,Noverlap);` |
| `h = spectrum.music(...,'DataMatrix');`<br>`pseudospectrum(...)` | `nfft = min(256,2^nextpow2(size(x,1)));`<br>`pmusic(x,P,nfft,Fs,range,win)` |
| `h = spectrum.music(...,'CorrelationMatrix');`<br>`pseudospectrum(...)` | `pmusic(x,P,'corr',nfft,Fs,range,win,Noverlap);`<br>`% or equivalently`<br>`pmusic(x,P,'corr',fVec,Fs,range,win,Noverlap);` |
| `h = spectrum.music(...);`<br>`pseudospectrum(...,'CenterDC',true)` | `pmusic(...,'centered');` |
| `[Spec,F] = pseudospectrum(...)` | `[Spec,F] = pmusic(...);` |

# Vibration Analysis

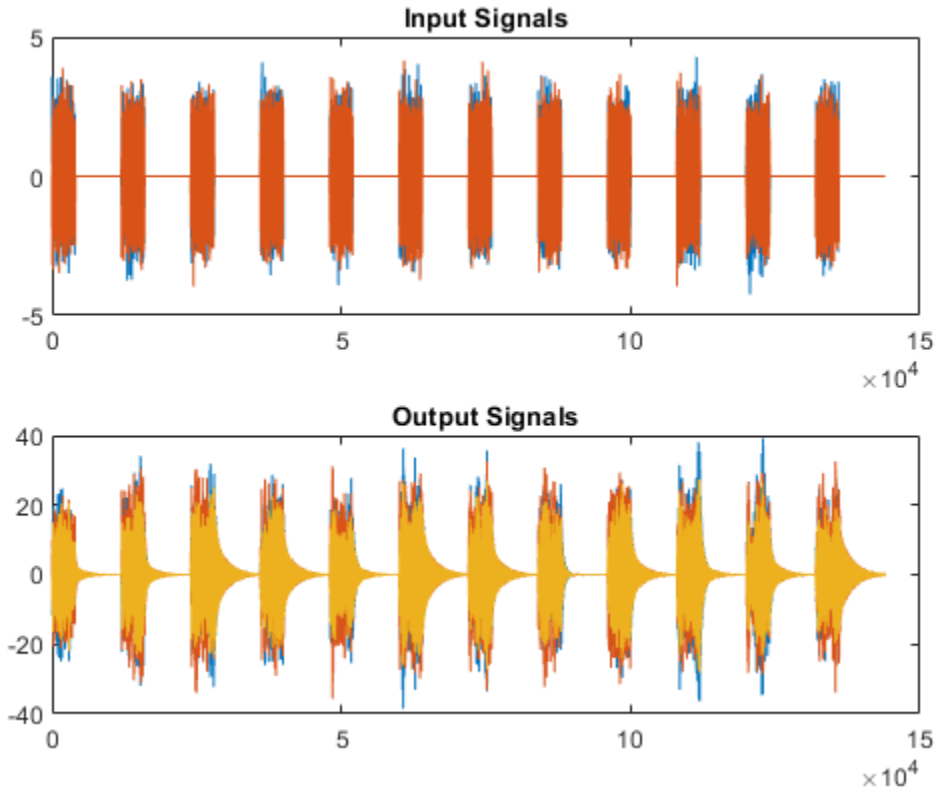

- “Modal Parameters of MIMO System” on page 20-2
- “Compute and Display Order-RPM Map” on page 20-5
- “MIMO Stabilization Diagram” on page 20-8
- “Modal Analysis of Identified Models” on page 20-12

# Modal Parameters of MIMO System

Compute the natural frequencies, the damping ratios, and the mode shapes for a two-input/three-output system excited by several bursts of random noise. Each burst lasts for 1 second, and there are 2 seconds between the end of each burst and the start of the next. The data are sampled at 4 kHz.

Load the data file. Plot the input signals and the output signals.

```
load modaldata

subplot(2,1,1)
plot(Xburst)
title('Input Signals')
subplot(2,1,2)
plot(Yburst)
title('Output Signals')
```

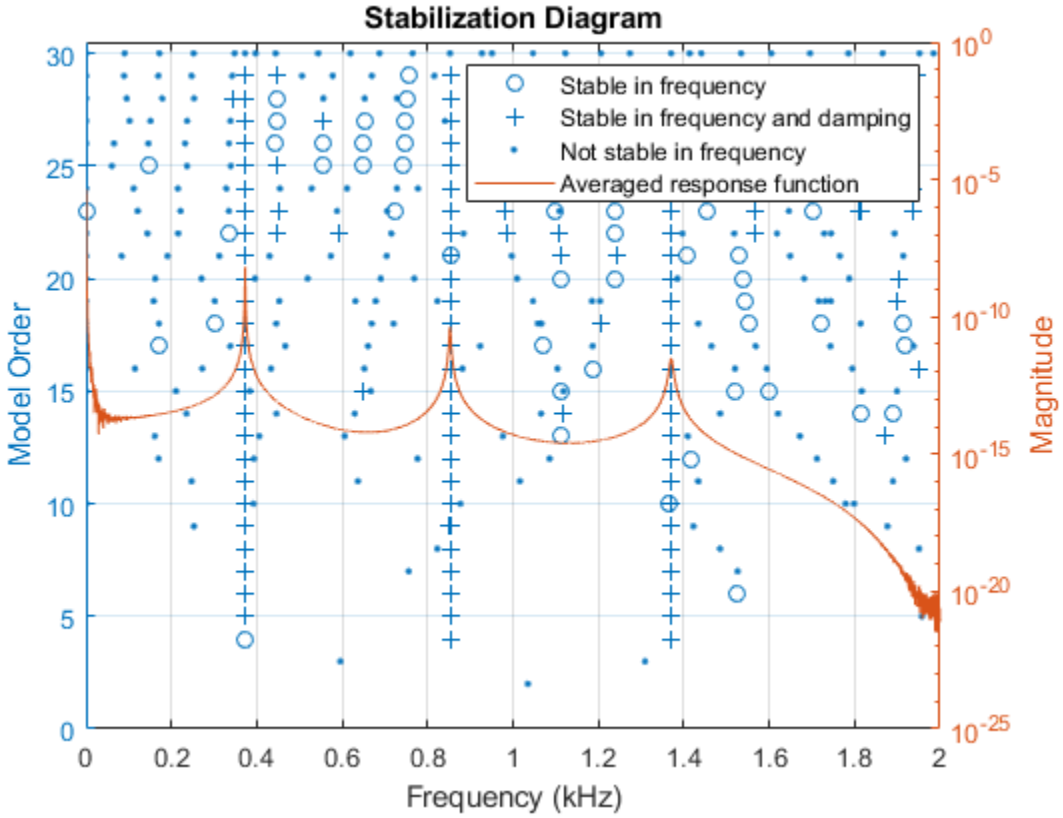



Compute the frequency-response functions. Specify a rectangular window with length equal to the burst period and no overlap between adjoining segments.

```
burstLen = 12000;
[frf,f] = modalfrf(Xburst,Yburst,fs,burstLen);
```

Visualize a stabilization diagram and return the stable natural frequencies. Specify a maximum model order of 30 modes.

```
figure
modalsd(frf,f,fs,'MaxModes',30);
```




Zoom in on the plot. The averaged response function has maxima at 373 Hz, 852 Hz, and 1371 Hz, which correspond to the physical frequencies of the system. Save the maxima to a variable.

```
phfr = [373 852 1371];
```

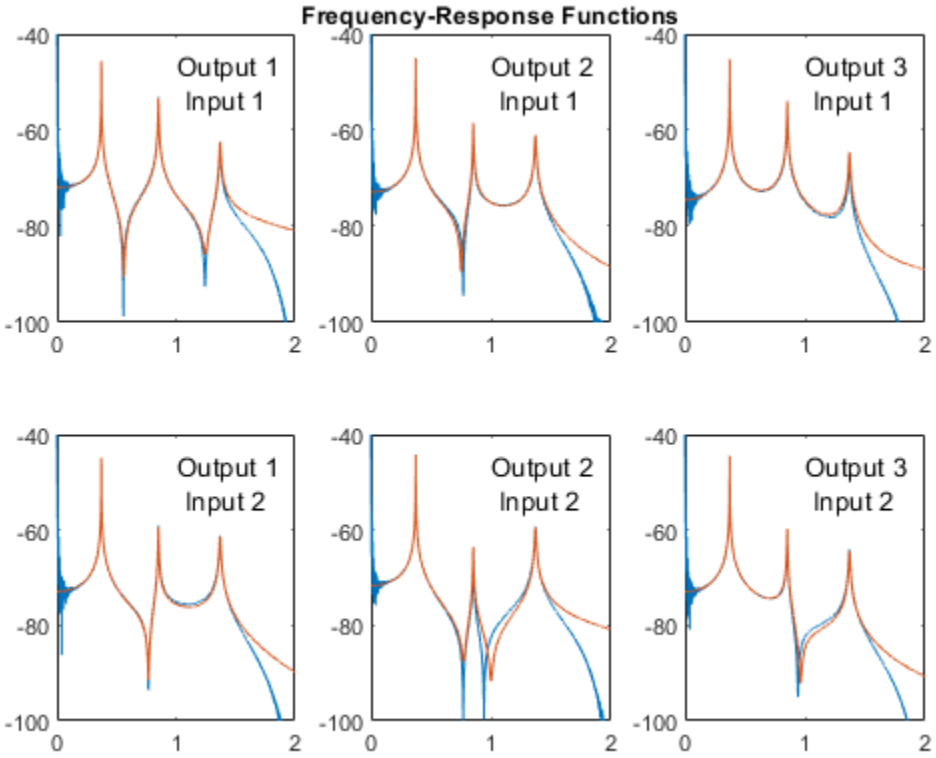
Compute the modal parameters using the least-squares complex exponential (LSCE) algorithm. Specify a model order of 6 modes and specify physical frequencies for the 3 modes determined from the stabilization diagram. The function generates one set of natural frequencies and damping ratios for each input reference.

```
[fn,dr,ms,ofrf] = modalfit(frf,f,fs,6,'PhysFreq',phfr);
```

Plot the reconstructed frequency-response functions and compare them to the original ones.

```
for k = 1:2
    for m = 1:3
        subplot(2,3,m+3*(k-1))
        plot(f/1000,10*log10(abs(frf(:,m,k))))
        hold on
        plot(f/1000,10*log10(abs(ofrf(:,m,k))))
        hold off
        text(1,-50,[['Output ';' Input '] num2str([m k]')])
        ylim([-100 -40])
    end
end
```

```
subplot(2,3,2)
title('Frequency-Response Functions')
```



# Compute and Display Order-RPM Map

Generate a signal that consists of two linear chirps and a quadratic chirp, all sampled at 600 Hz for 5 seconds. The system that produces the signal increases its rotational speed from 10 to 40 revolutions per second during the testing period.

Generate the tachometer readings.

```
fs = 600;
t1 = 5;
t = 0:1/fs:t1;
f0 = 10;
f1 = 40;
rpm = 60*linspace(f0,f1,length(t));
```

The linear chirps have orders 1 and 2.5. The component with order 1 has twice the amplitude of the other. The quadratic chirp starts at order 6 and returns to this order at the end of the measurement. Its amplitude is 0.8. Create the signal using this information.

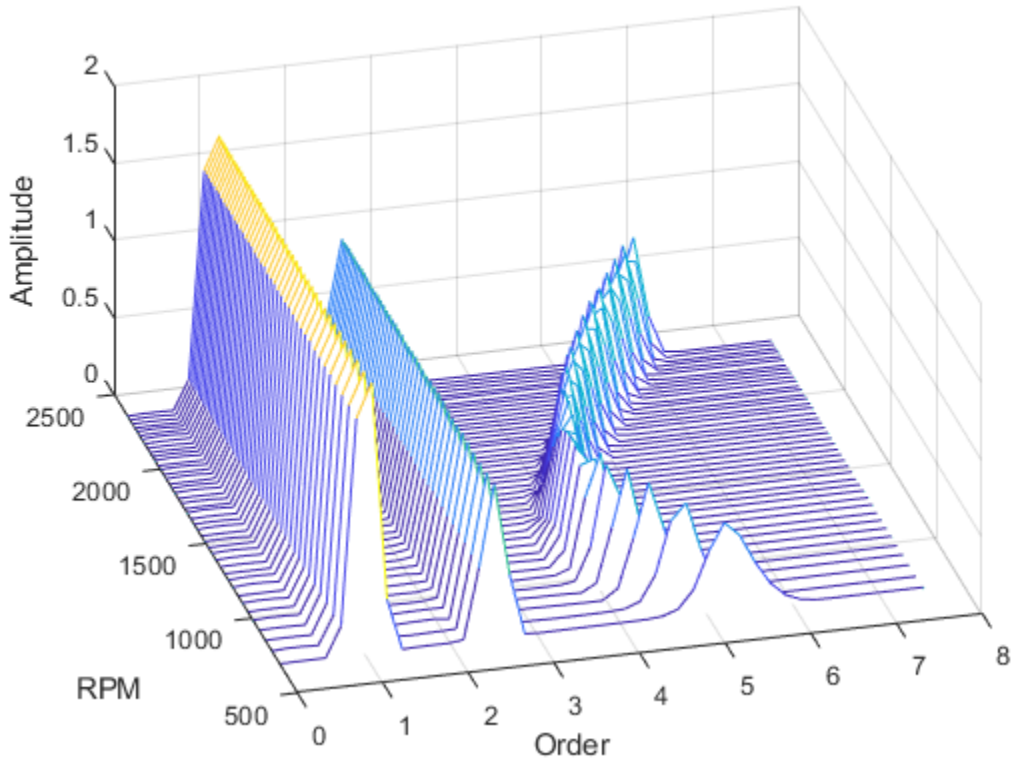
```
o1 = 1;
o2 = 2.5;
o6 = 6;
x = 2*chirp(t,o1*f0,t1,o1*f1)+chirp(t,o2*f0,t1,o2*f1) + ...
    0.8*chirp(t,o6*f0,t1,o6*f1,'quadratic');
```

Compute the order-RPM map of the signal. Use the peak amplitude at each measurement cell. Specify a resolution of 0.25 orders. Window the data with a Chebyshev window whose sidelobe attenuation is 50 dB.

```
[map,or,rp] = rpmordermap(x,fs,rpm,0.25, ...
    'Amplitude','peak','Window',{'chebwin',50});
```

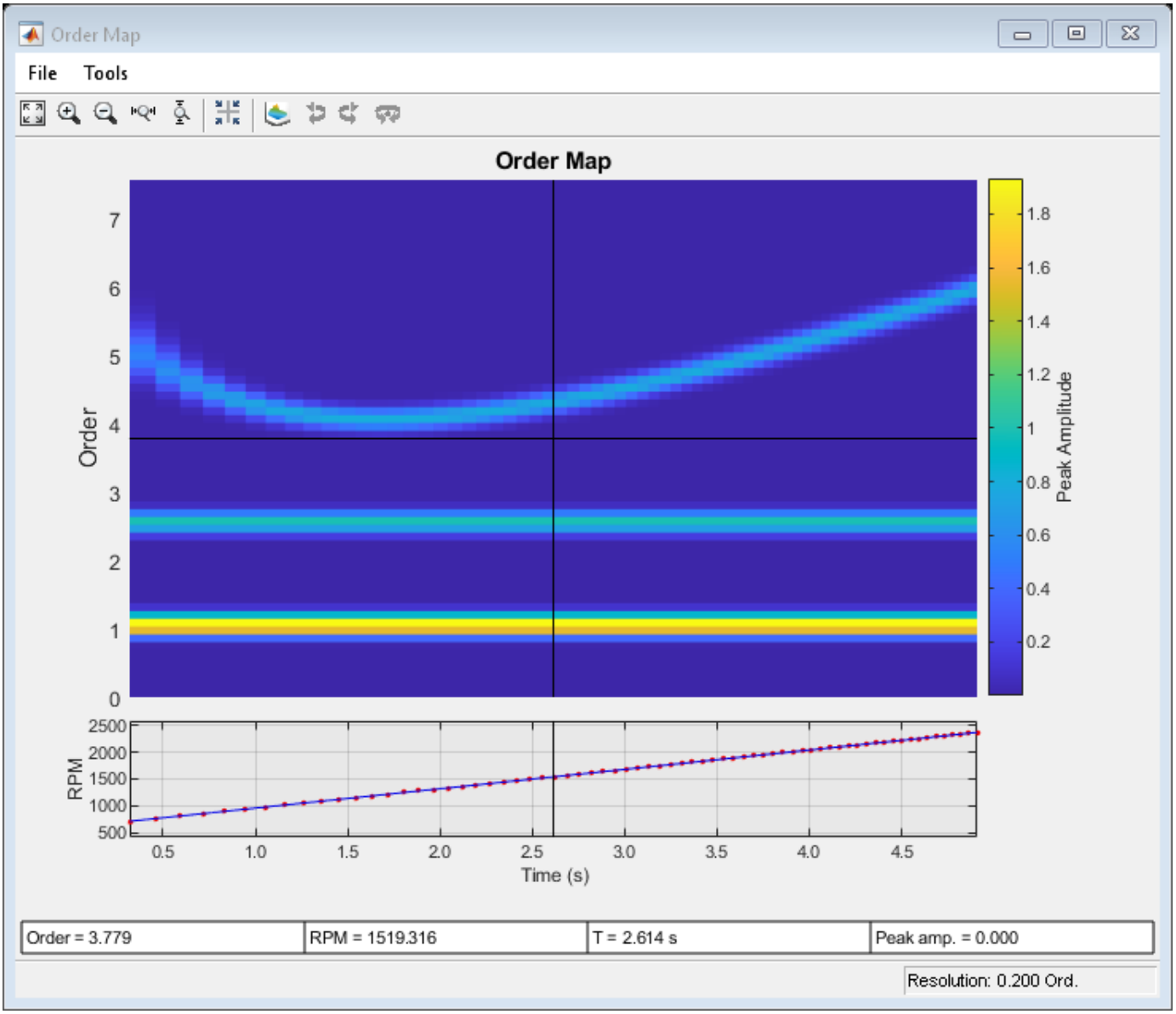
Draw the order-RPM map as a waterfall plot.

```
[OR,RP] = meshgrid(or,rp);
waterfall(OR,RP,map')
view(-15,45)
xlabel('Order')
ylabel('RPM')
zlabel('Amplitude')
```




Use `rpmordermap` with no output arguments to display the map. Specify a resolution of 0.2 orders and 80% overlap between adjoining segments. Set the sidelobe attenuation of the Chebyshev window to 80 dB.

```
rpmordermap(x,fs,rpm,0.2, ...
    'Amplitude','peak','OverlapPercent',80,'Window',{'chebwin',80})
```




## See Also

`orderspectrum` | `ordertrack` | `orderwaveform` | `rpmfreqmap` | `rpmordermap` | `tachorpm`

# MIMO Stabilization Diagram

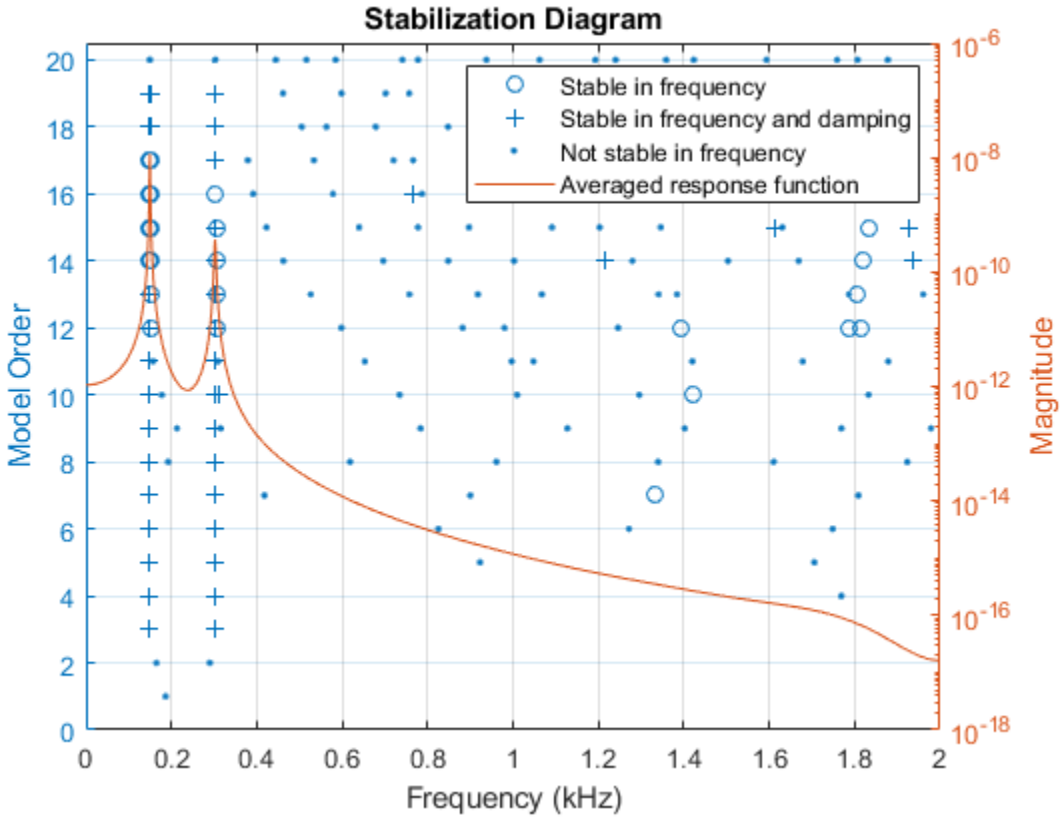
Compute the frequency-response functions for a two-input/two-output system excited by random noise.

Load the data file. Compute the frequency-response functions using a 5000-sample Hann window and 50% overlap between adjoining data segments. Specify that the output measurements are displacements.

```
load modaldata
winlen = 5000;

[frf,f] = modalfrf(Xrand,Yrand,fs,hann(winlen),0.5*winlen,'Sensor','dis');
```

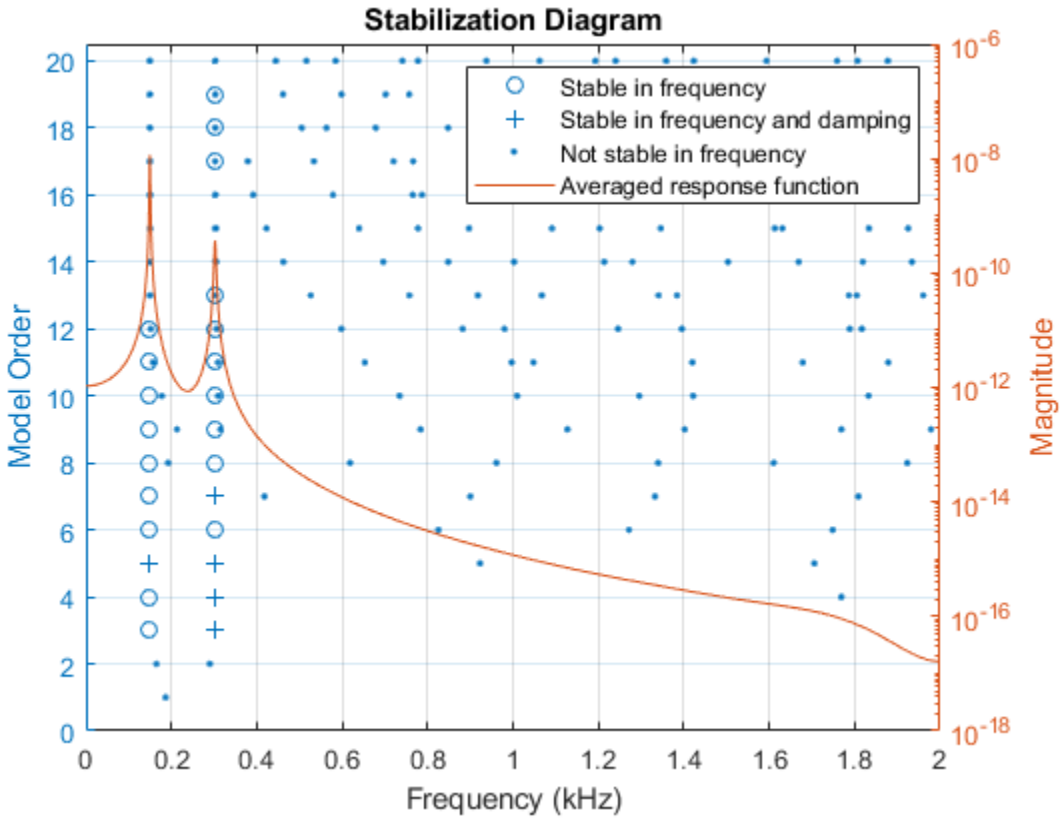
Generate a stabilization diagram to identify up to 20 physical modes.

```
modalsd(frf,f,fs,'MaxModes',20)
```



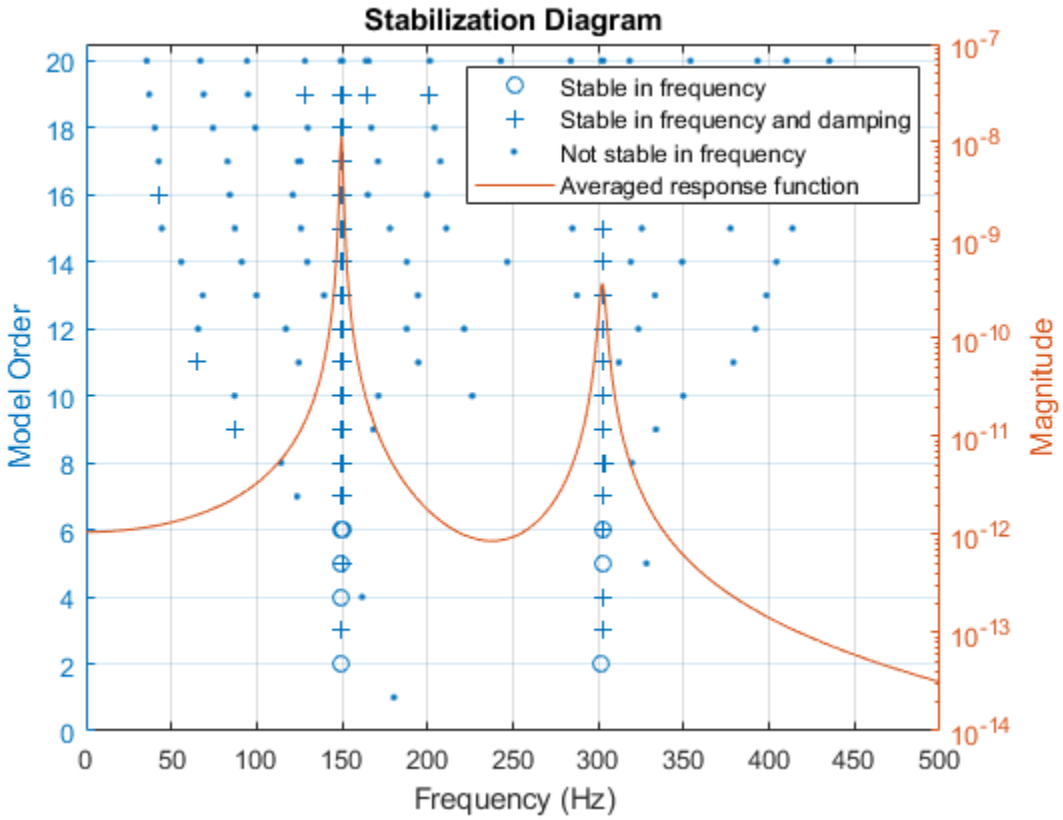

Repeat the computation, but now tighten the criteria for stability. Classify a given pole as stable in frequency if its natural frequency changes by less than 0.01% as the model order increases. Classify a given pole as stable in damping if the damping ratio estimate changes by less than 0.2% as the model order increases.

```
modalsd(frf,f,fs,'MaxModes',20,'SCriteria',[1e-4 0.002])
```



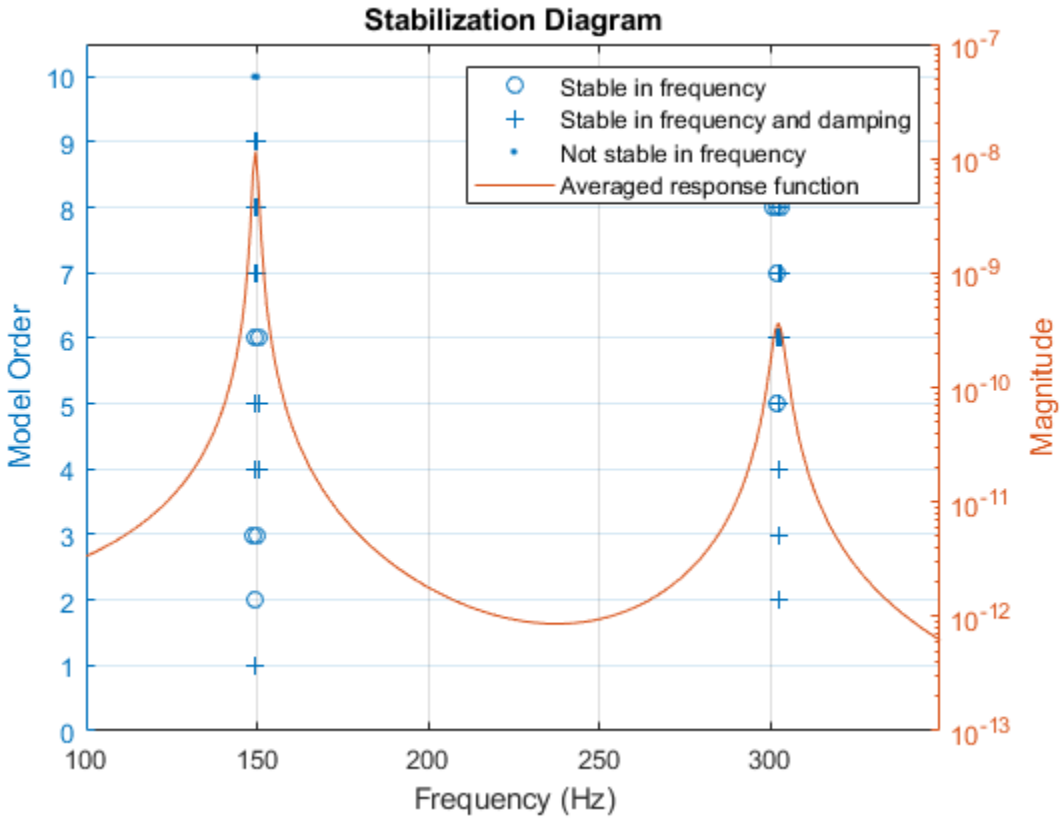

Restrict the frequency range to between 0 and 500 Hz. Relax the stability criteria to 0.5% for frequency and 10% for damping.

```
modalsd(frf,f,fs,'MaxModes',20,'SCriteria',[5e-3 0.1],'FreqRange',[0 500])
```




Repeat the computation using the least-squares rational function algorithm. Restrict the frequency range from 100 Hz to 350 Hz and identify up to 10 physical modes.

```
modalsd(frf,f,fs,'MaxModes',10,'FreqRange',[100 350],'FitMethod','lsrf')
```




## See Also

`modalfit` | `modalfrf` | `modalsd`

## Related Examples

- “Order Analysis of a Vibration Signal” on page 24-424

# Modal Analysis of Identified Models

Identify state-space models of systems. Use the models to compute frequency-response functions and modal parameters. This example requires a System Identification Toolbox™ license.

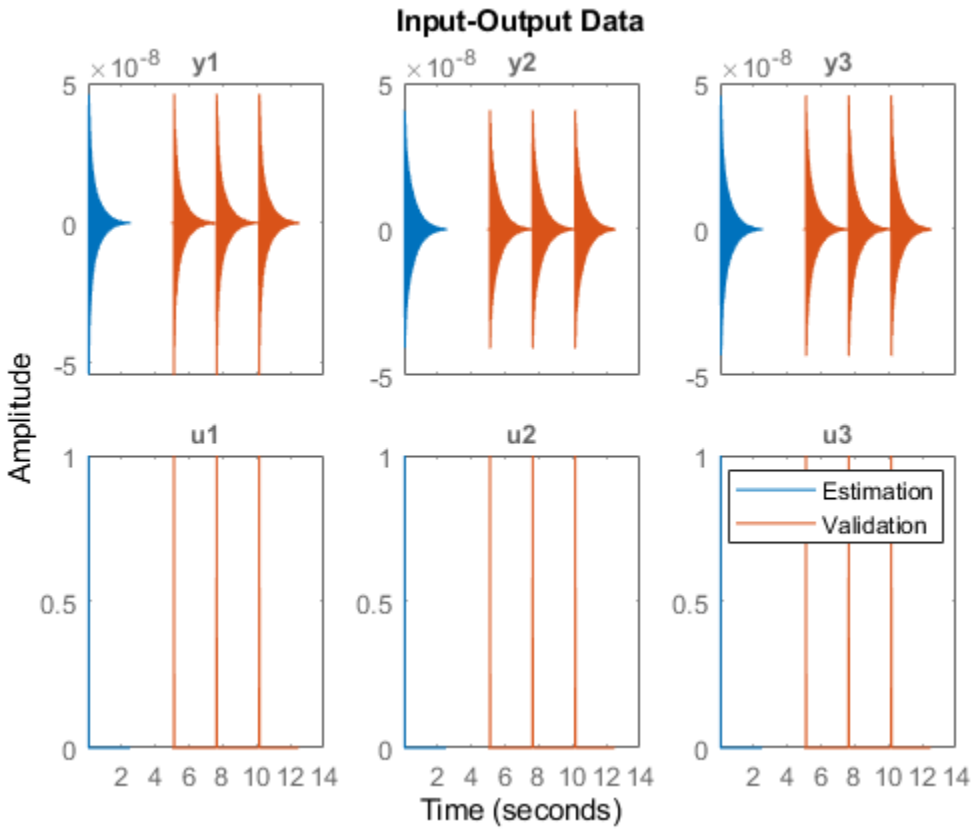
### Hammer Excitation

Load a file containing three-input/three-output hammer excitation data sampled at 4 kHz. Use the first $10^4$ samples for estimation and samples $2 \times 10^4$ to $5 \times 10^4$ for model quality validation. Specify the sample time as the inverse of the sample rate. Store the data as `@iddata` objects.

```
load modaldata XhammerMISO1 YhammerMISO1 fs

rest = 1:1e4;
rval = 2e4:5e4;
Ts = 1/fs;

Estimation = iddata(YhammerMISO1(rest,:),XhammerMISO1(rest,:),Ts);
Validation = iddata(YhammerMISO1(rval,:),XhammerMISO1(rval,:),Ts,'Tstart',rval(1)*Ts);
```

Plot the estimation data and the validation data.

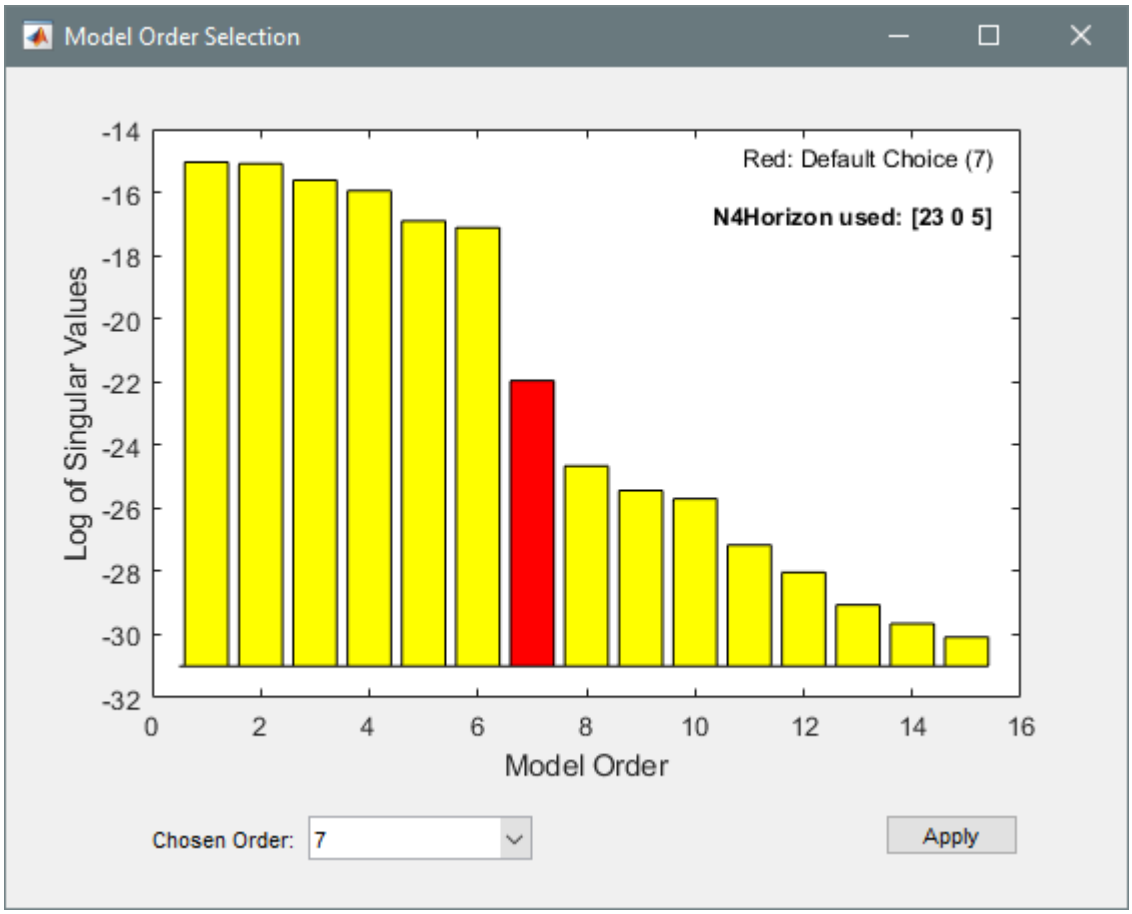
```
plot(Estimation,Validation)
legend(gca,'show')
```



Use the `ssest` function to estimate a 7th-order state-space model of the system that minimizes the simulation error between the measured outputs and the model outputs. Specify that the state-space model has feedthrough.

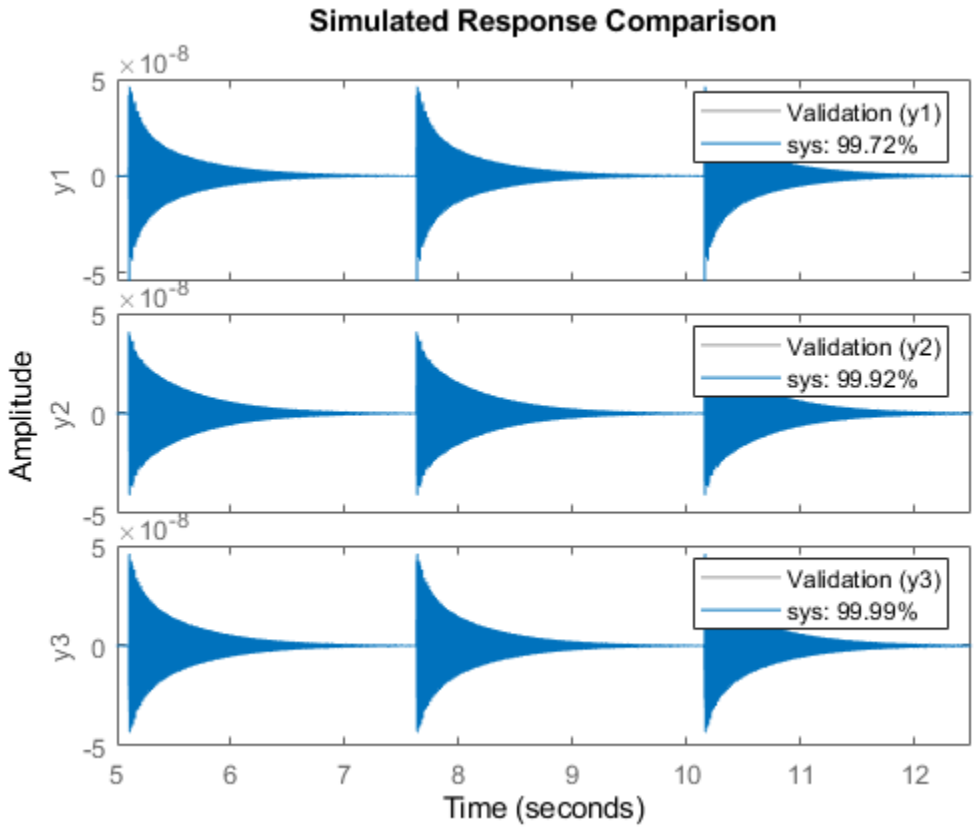
```
Orders = 7;
opt = ssestOptions('Focus','simulation');

sys = ssest(Estimation,Orders,'Feedthrough',true,'Ts',Ts,opt);
```

(To find the model order that gives the best tradeoff between accuracy and complexity, set `Orders` to `1:15` in the previous code. `ssest` outputs a log plot of singular values that lets you specify the order interactively. The function also recommends a model order of 7.)



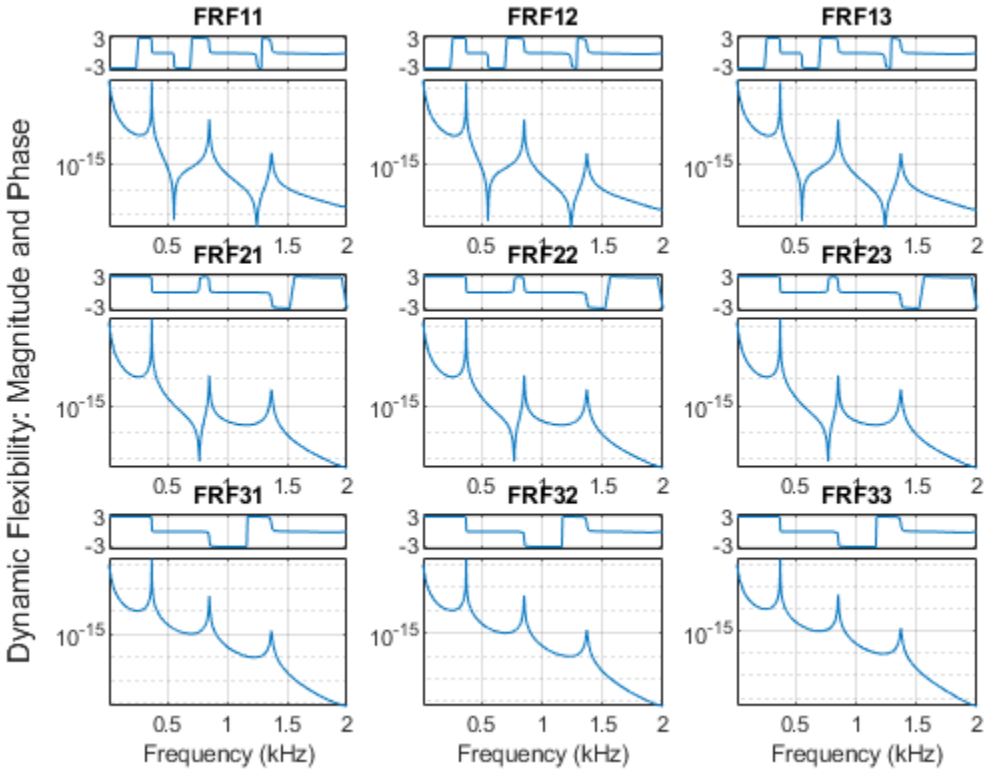

Validate the model quality on the validation dataset. Plot the normalized root mean square error (NRMSE) measure of goodness-of-fit. The model describes accurately the output signals of the validation data.

```
compare(Validation,sys)
```




Estimate the frequency-response functions of the model. Display the functions using `modalfrf` without output arguments.

```
[frf,f] = modalfrf(sys);
modalfrf(sys)
```



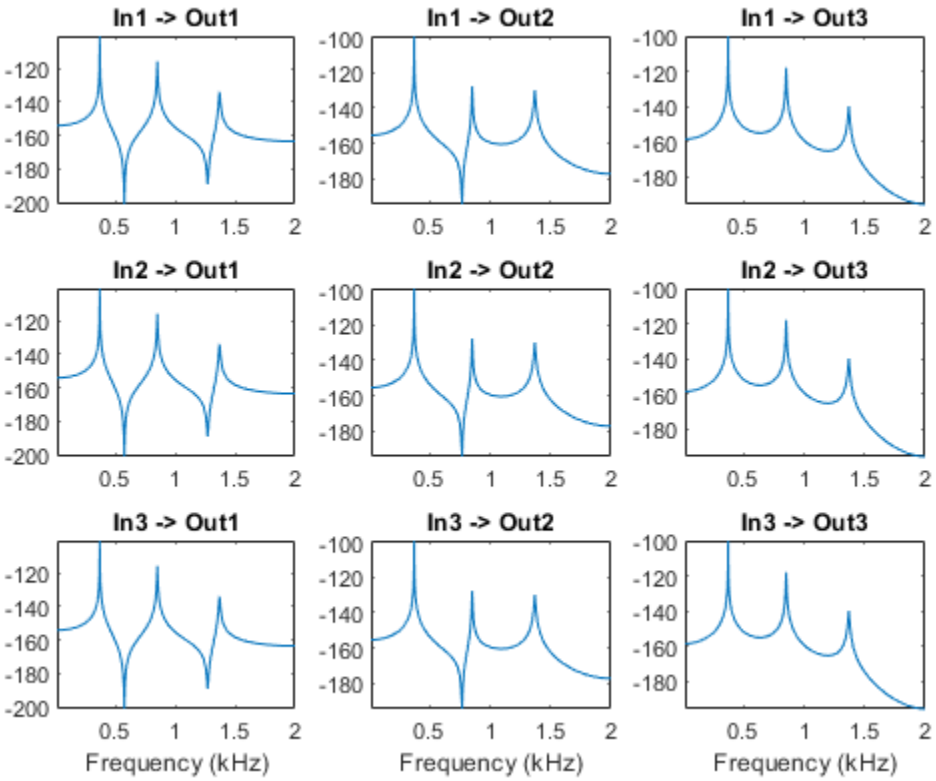

Assume that the system is well described using three modes. Compute the natural frequencies, damping ratios, and mode-shape vectors of the three modes.

```
Modes = 3;
[fn,dr,ms] = modalfit(sys,f,Modes)
```

```
fn = 3×1
10^3 ×

    0.3727
    0.8525
    1.3706


dr = 3×1

    0.0008
    0.0018
    0.0029


ms = 3×3 complex

   0.0036 - 0.0019i   0.0039 - 0.0005i   0.0021 + 0.0006i
   0.0043 - 0.0023i   0.0010 - 0.0001i  -0.0033 - 0.0010i
   0.0040 - 0.0021i  -0.0031 + 0.0004i   0.0011 + 0.0003i
```

Compute and display the reconstructed frequency-response functions. Express the magnitudes in decibels.

```
[~,~,~,ofrf] = modalfit(sys,f,Modes);

clf
for ij = 1:3
    for ji = 1:3
        subplot(3,3,3*(ij-1)+ji)
        plot(f/1000,20*log10(abs(ofrf(:,ji,ij))))
        axis tight
        title(sprintf('In%d -> Out%d',ij,ji))
        if ij==3
            xlabel('Frequency (kHz)')
        end
    end
end
```

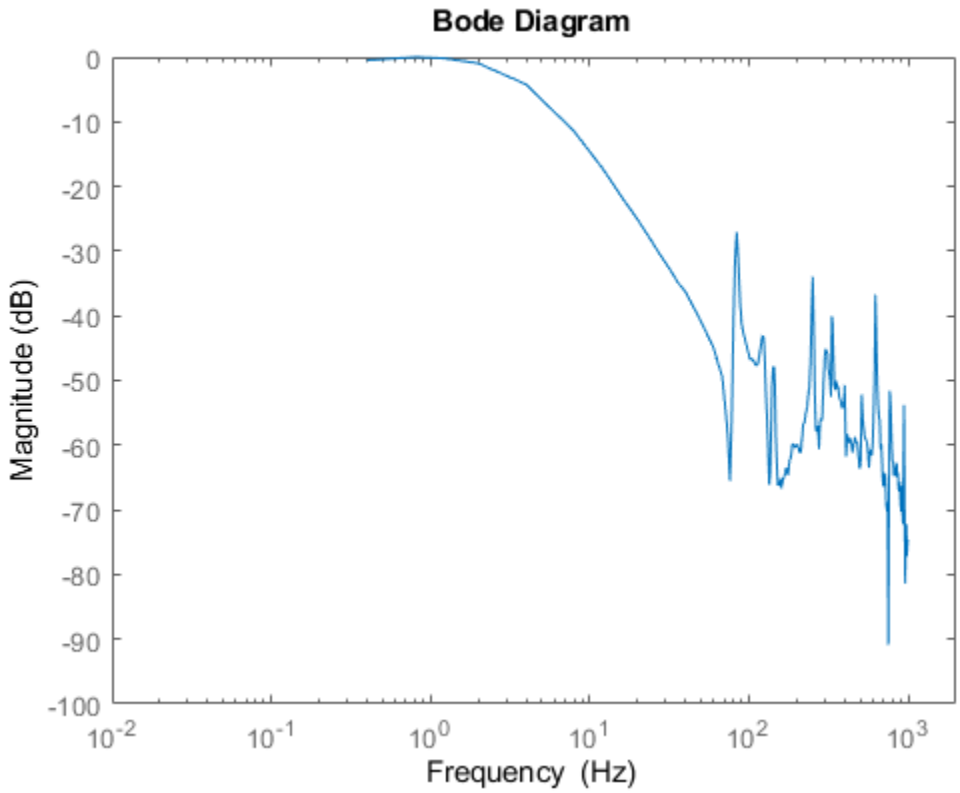



### Controlled Unstable Process

Load a file containing a high modal density frequency-response measurement. The data corresponds to an unstable process maintained at equilibrium using feedback control. Store the data as an `idfrd` object for identification. Plot the Bode diagram.

```
load HighModalDensData FRF f

G = idfrd(permute(FRF,[2 3 1]),f,0,'FrequencyUnit','Hz');
figure
```

```
bodemag(G)
xlim([0.01,2e3])
```

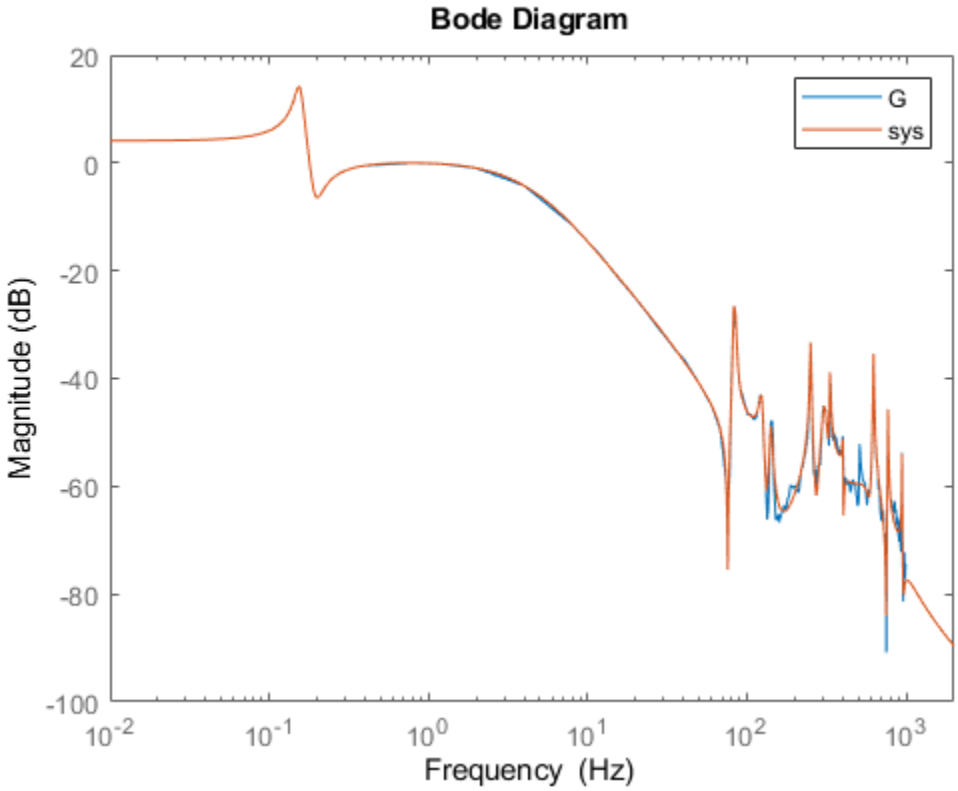



Identify a transfer function with 32 poles and 32 zeros.

```
sys = tfest(G,32,32);
```

Compare the frequency response of the model with the measured response.

```
bodemag(G,sys)
xlim([0.01,2e3])
legend(gca,'show')
```




Extract the natural frequencies and damping ratios of the first 10 least-damped oscillatory modes. Store the results in a table.

```
[fn,dr] = modalfit(sys,[],10);
T = table((1:10)',fn,dr,'VariableNames',{'Mode','Frequency','Damping'})
```

```
T=10×3 table
    Mode    Frequency     Damping
    ____    _________    _________

      1      82.764       0.011304
      2      85.013       0.015632
      3      124.04       0.025252
      4      142.04       0.017687
      5      251.46      0.0062182
      6      332.79      0.0058266
      7      401.21      0.0043645
      8      625.14      0.0039247
      9      770.49       0.002795
     10      943.64      0.0019943
```

## See Also

`modalfit` | `modalfrf` | `ssest`

# Signal Analyzer App


- “Using Signal Analyzer App” on page 21-2
- “Select Signals to Analyze” on page 21-9
- “Preprocess Signals” on page 21-13
- “Explore Signals” on page 21-19
- “Share Analysis” on page 21-26
- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Resolve Tones by Varying Window Leakage” on page 21-43
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Extract Regions of Interest from Whale Song” on page 21-56
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126
- “Customize Signal Analyzer” on page 21-130

# Using Signal Analyzer App

## App Workflow

A typical workflow for inspecting and comparing signals using the **Signal Analyzer** app is:

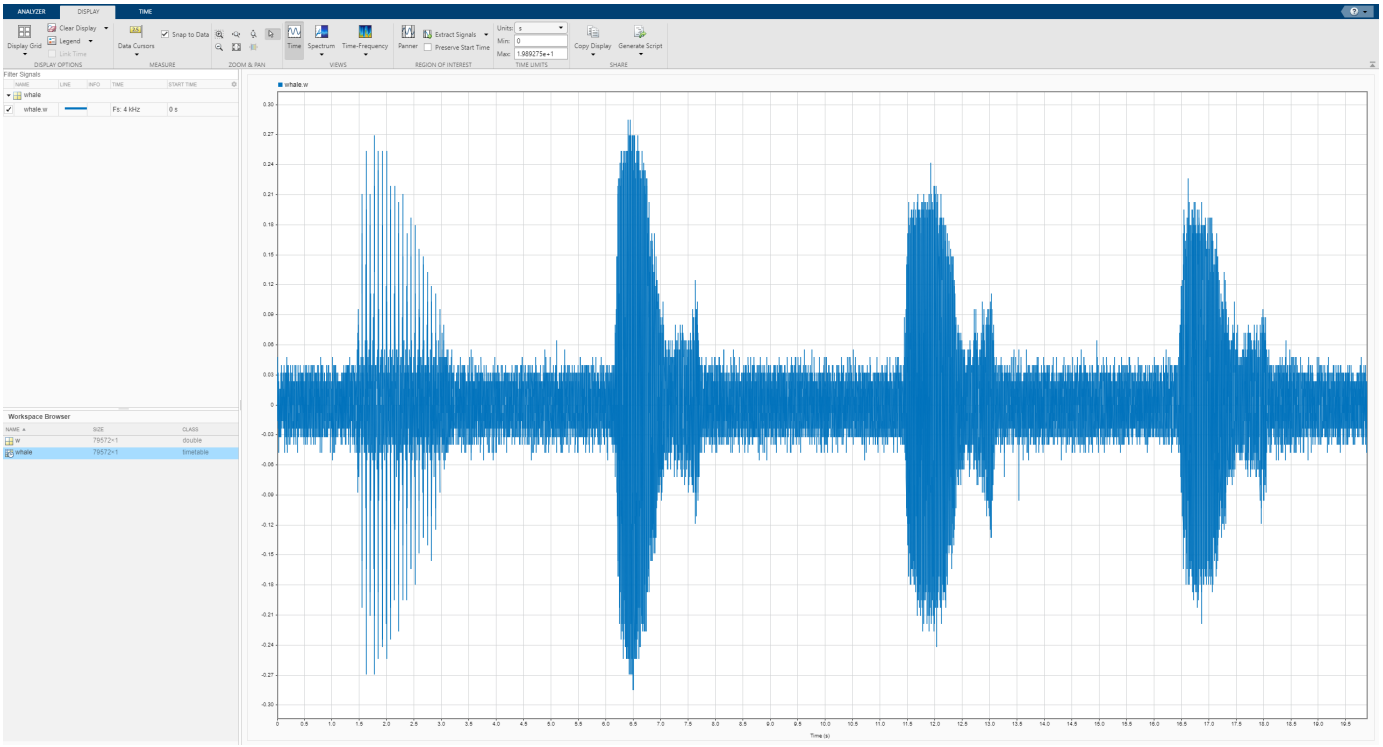
1. Select Signals to Analyze on page 21-9 — Select any signal available in the MATLAB workspace. The app accepts numeric arrays and signals with inherent time information, such as MATLAB `timetable` arrays, `timeseries` objects, and `labeledSignalSet` objects. See “Data Types Supported by Signal Analyzer” on page 21-106 for more information.
2. “Preprocess Signals” on page 21-13 — Lowpass, highpass, bandpass, or bandstop filter signals. Remove trends and compute signal envelopes. Smooth signals using moving averages, regression, Savitzky-Golay filters, or other methods. Change sample rates of signals or interpolate nonuniformly sampled signals onto uniform grids. Preprocess signals using your own custom functions. Generate MATLAB functions to automate preprocessing operations.
3. Explore Signals on page 21-19 — Add time information to signals using sample rates, numeric vectors, `duration` arrays, or MATLAB expressions. Plot, measure, and compare data, their spectra, their spectrograms, or their scalograms. Look for features and patterns in the time domain, in the frequency domain, and in the time-frequency domain. Compute persistence spectra to analyze sporadic signals and sharpen spectrogram estimates using reassignment. Extract regions of interest from signals.
4. Share Analysis on page 21-26 — Copy displays from the app to the clipboard as images. Export signals to the MATLAB workspace or save them to MAT-files. Generate MATLAB scripts to automate the computation of power spectrum, spectrogram, or persistence spectrum estimates and the extraction of regions of interest. Save **Signal Analyzer** sessions to resume your analysis later or on another machine.

## Example: Extract Regions of Interest from Whale Song

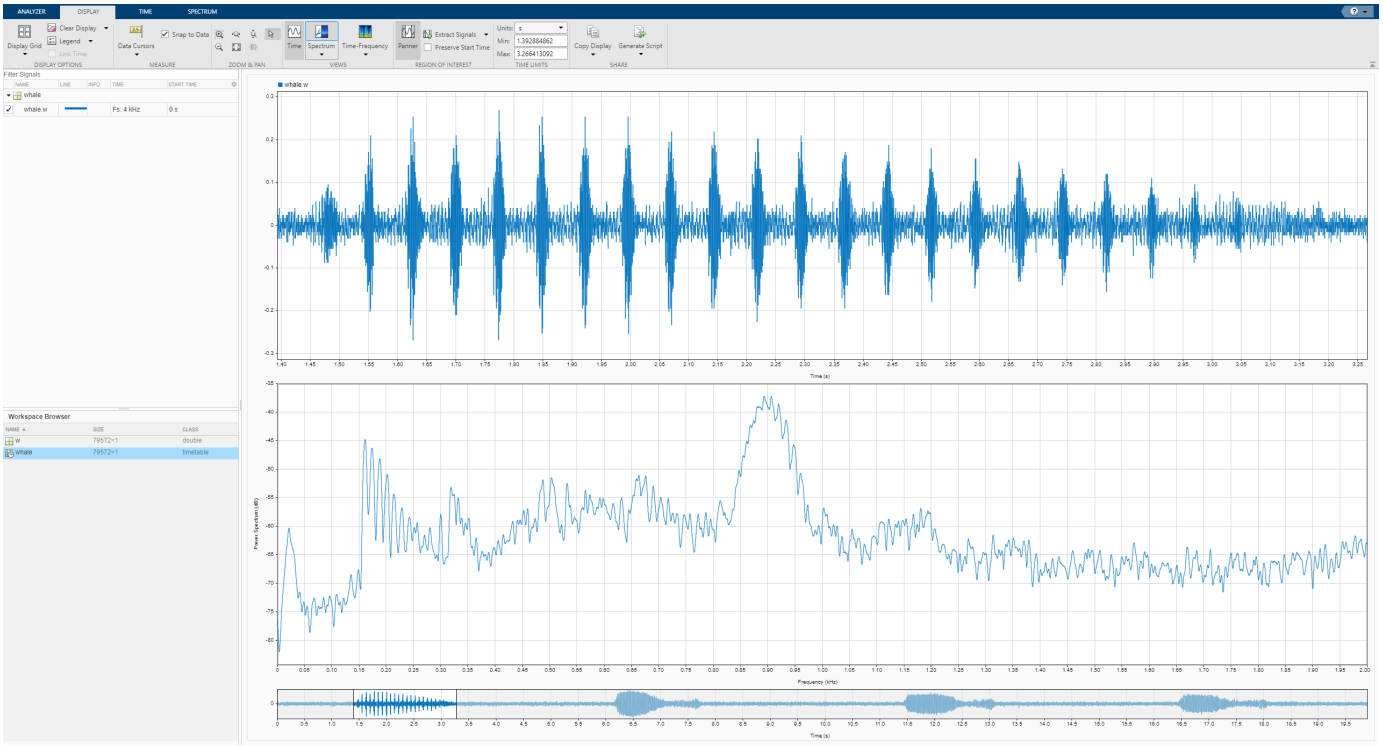
Load a file that contains audio data from a Pacific blue whale, sampled at 4 kHz. The file is from the library of animal vocalizations maintained by the Cornell University Bioacoustics Research Program. The time scale in the data is compressed by a factor of 10 to raise the pitch and make the calls more audible. Convert the signal to a MATLAB® timetable.

```
whaleFile = fullfile(matlabroot,'examples','matlab','data','bluewhale.au');
[w,fs] = audioread(whaleFile);

whale = timetable(seconds((0:length(w)-1)'/fs),w);

% To hear, type soundsc(w,fs)
```

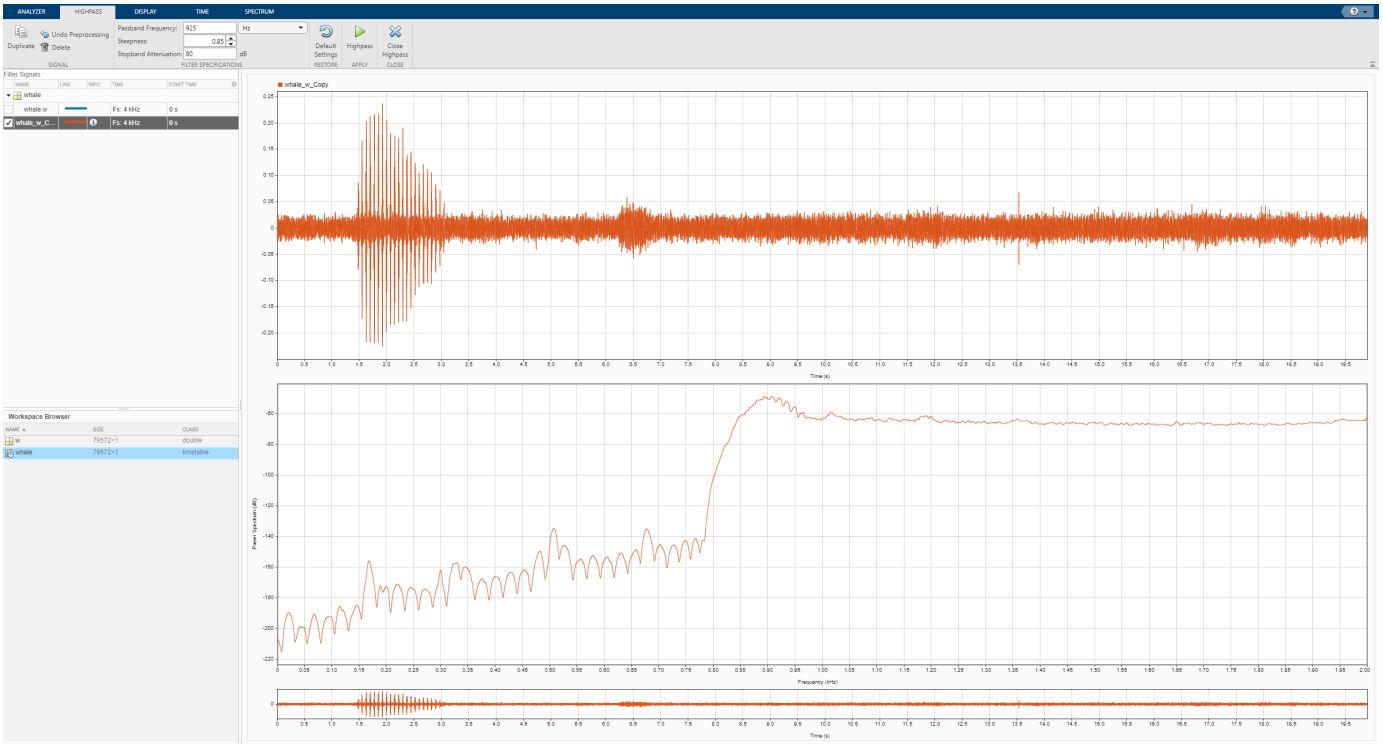
Open **Signal Analyzer** and drag the timetable to a display. Four features stand out from the noise. The first is known as a *trill*, and the other three are known as *moans*.



On the **Display** tab, click **Spectrum** to open a spectrum view and click **Panner** to activate the panner. Use the panner to create a zoom window with a width of about 2 seconds. Drag the zoom window so that it is centered on the trill. The spectrum shows a noticeable peak at around 900 Hz.

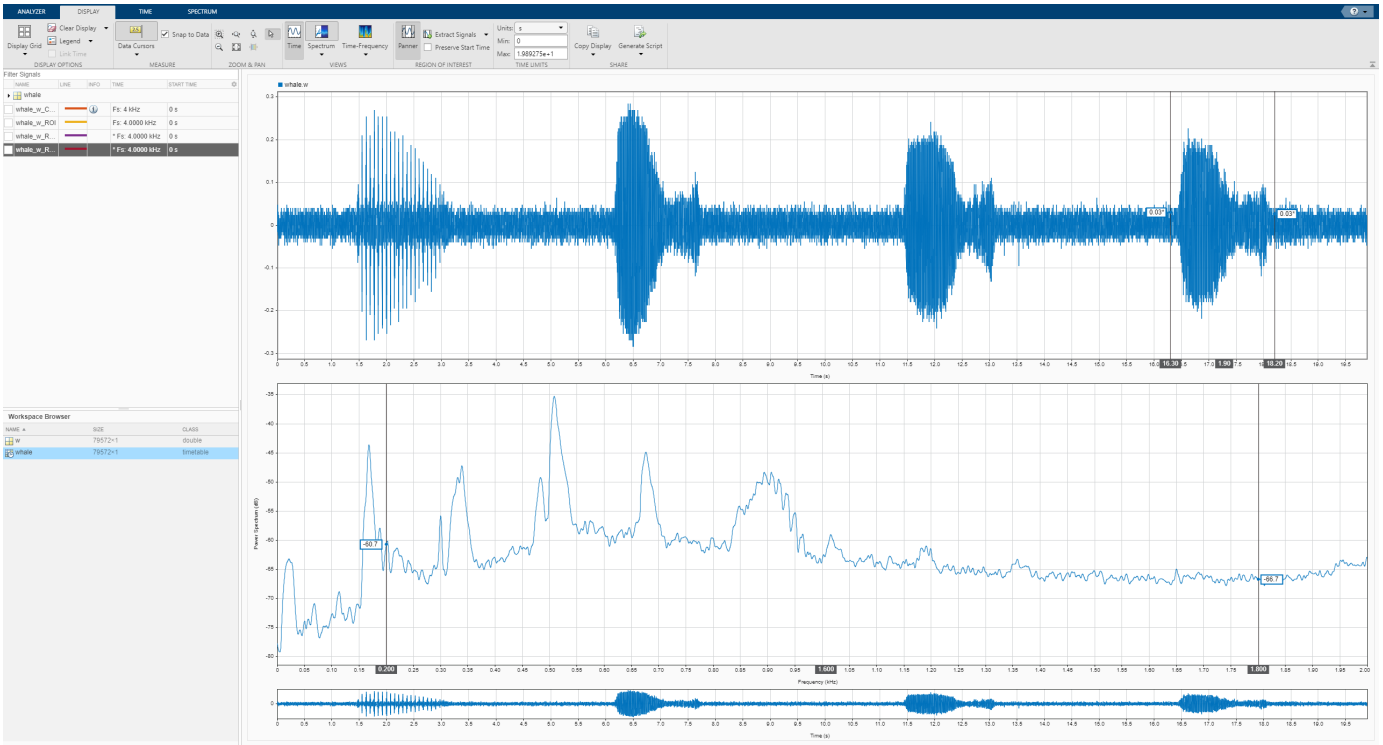


Isolate the single trill by highpass filtering. Right-click the signal in the Signal table and select **Duplicate** to create a copy of the whale song. Remove the original signal from the display by clearing the check box next to its name in the Signal table. On the **Analyzer** tab, click **Preprocessing ▼** and select **Highpass**. Set the passband frequency to 925 Hz and the stopband attenuation at 80 dB. Use the default value for the steepness.

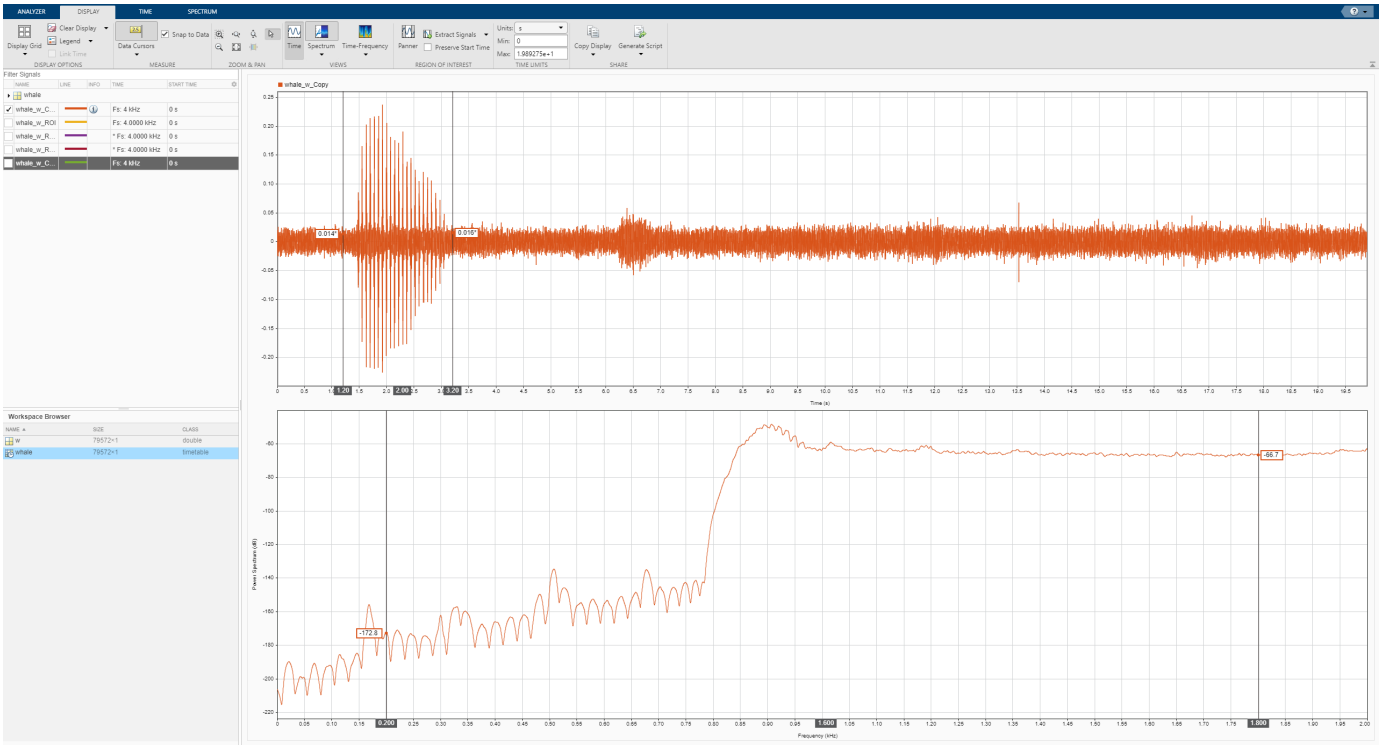


Clear the display and select the original signal. Extract the three moans to compare their spectra:

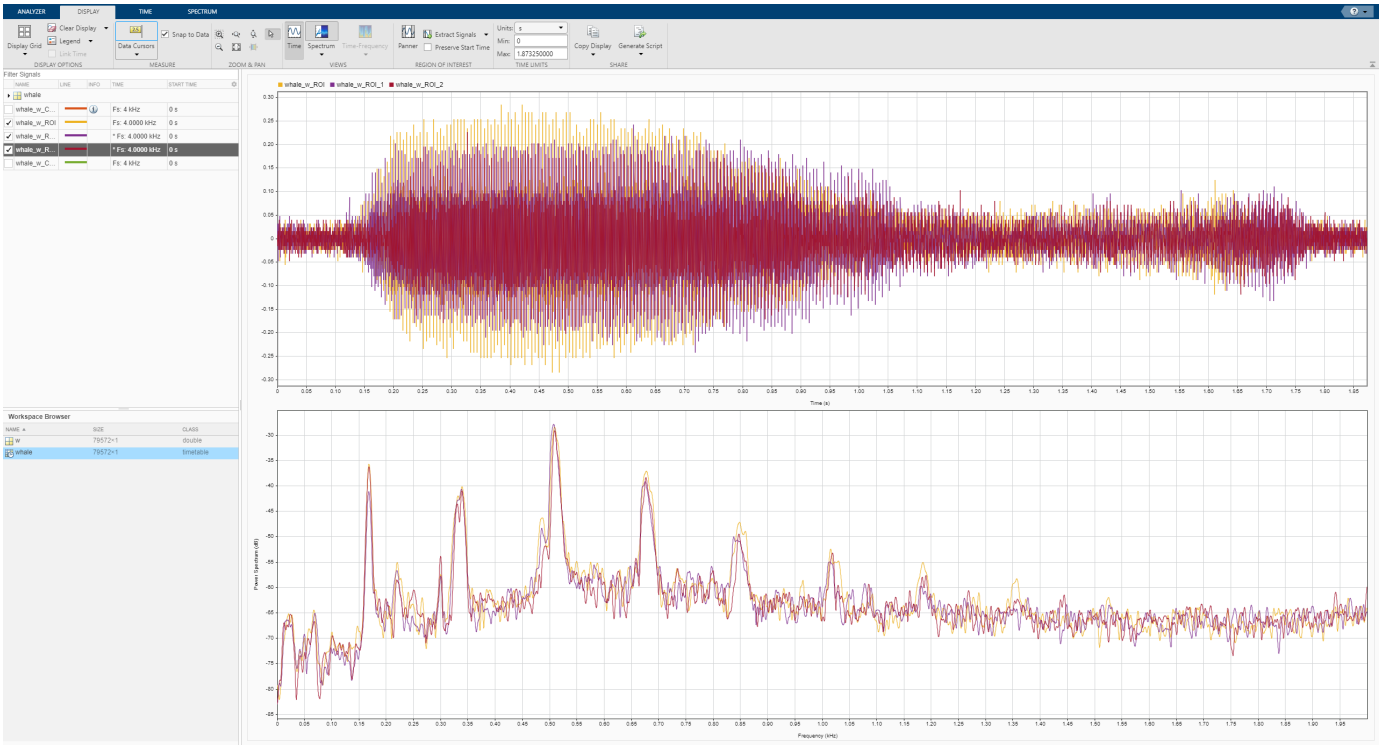
1. Center the panner zoom window on the first moan. The spectrum has eight clearly defined peaks, located very close to multiples of 170 Hz. Click **Extract Signals ▼** and select `Between Time Limits`.
2. Click **Panner** to hide the panner. Press the space bar to see the full signal. Click **Zoom in X** and zoom in on a 2-second interval of the time view centered on the second moan. The spectrum again has peaks at multiples of 170 Hz. Click **Extract Signals ▼** and select `Between Time Limits`.
3. Press the space bar to see the full signal. Click **Data Cursors ▼** and select `Two`. Place the time domain cursors in a 2-second interval around the third moan. Again, there are peaks at multiples of 170 Hz. Click **Extract Signals ▼** and select `Between Time Cursors`.



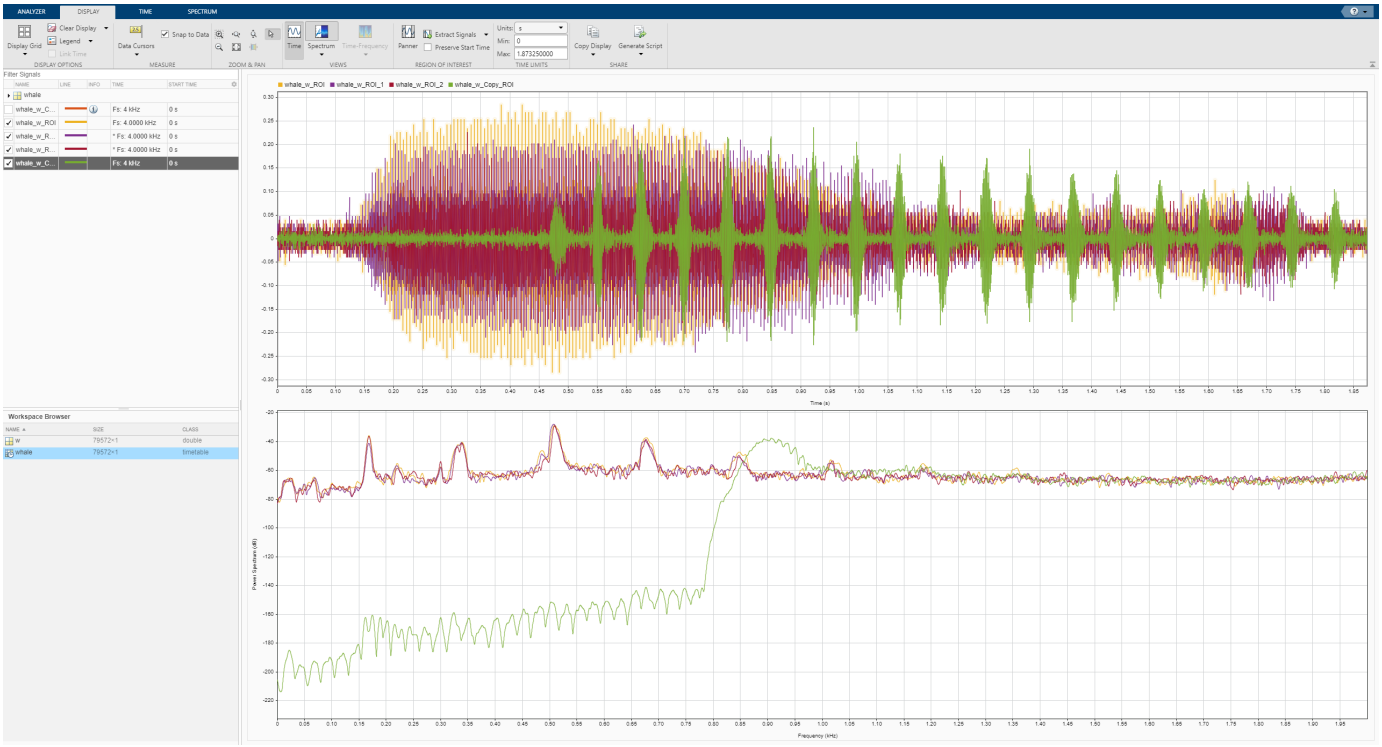
Plot the highpass-filtered signal and place the two data cursors at 1 second and 3.5 seconds. Extract the region containing the trill.



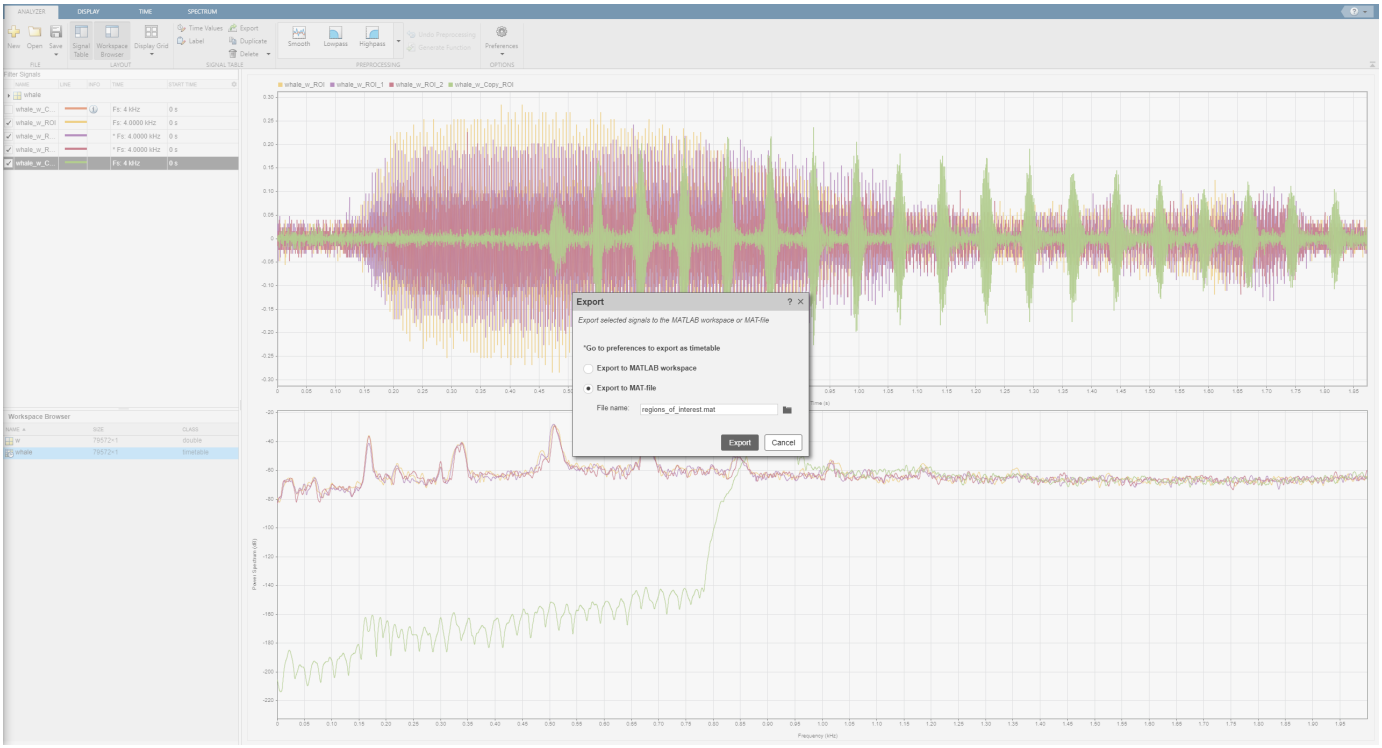
Remove the original signal from the display by clearing the check box next to its name in the Signal table. Display the three regions of interest you just extracted. Their spectra lie approximately on top of each other.



On the same display, plot the region of interest containing the trill that you extracted. The trill and moan spectra are noticeably different.



Click on **Export** on the **Analyzer** tab to export the four regions of interest in a MAT-file.



## See Also

**Signal Analyzer | Signal Labeler**

## Related Examples


- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

# Select Signals to Analyze

The **Signal Analyzer** app works with vectors, matrices, MATLAB timetables, `timeseries` objects, or `labeledSignalSet` objects in the MATLAB workspace. When you start the app, all usable signals in the workspace appear in the Workspace browser at the bottom-left corner. See “Data Types Supported by Signal Analyzer” on page 21-106 for more information.

## Select Signals from the Workspace Browser

Select signals from the Workspace browser by clicking their names and dragging them to the Signal table at the top-left corner. To plot a signal, drag it to a display. If you select the check box next to the name of a signal in the Signal table, the signal is plotted in the selected display. You can also drag signals directly from the Workspace browser to a display. The dragged signals are plotted in the display and listed in the Signal table.

---

**Note** **Signal Analyzer** does not support matrices, time series, timetables, or labeled signal sets with more than 8000 channels.

---

There are two different ways to choose signals in the Signal table. Each way gives you access to a different set of operations.

- Selecting the signal by clicking the **Name**, **Info**, **Time**, or **Start Time** column in the Signal table enables you to perform all the operations in the **Analyzer** tab. You can change the time information, preprocess the signals, or duplicate them. You do not need to plot a signal to preprocess it.
- Selecting the check box to the left of the signal name plots the signal in the currently selected display and enables you to perform all the operations in the **Display** tab. You can display the signal in the frequency domain or the time-frequency domain, or you can measure the signal using cursors.

---

**Note** If you attempt to import signals with more than 100 columns, the app displays a warning. The matrix you are trying to import might be the transpose of a multichannel signal that you want to analyze. In that case, click **No** in the warning dialog box and transpose the matrix in the workspace. If you do want to import the columns as separate signals, click **Yes**. If you drag the matrix to a display and click **Yes** in the warning dialog box, then the app plots only the first 10 columns of the matrix but imports all the columns. To plot signal columns beyond the 10th, drag them to the display. Alternatively, in the Signal table, select the check boxes next to the names of the signals you want to plot.

---

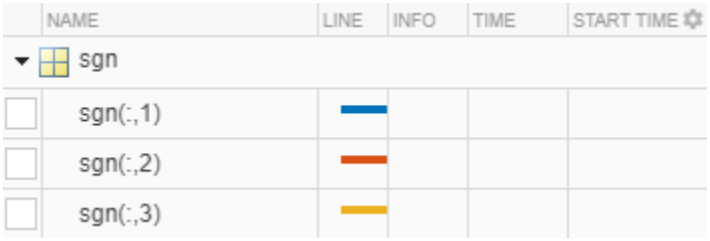
If you modify a signal in the MATLAB workspace, the Workspace browser updates automatically. However, the app does not recognize the changes until you reimport the signal by dragging it again to the Signal table or to a display.

If you add or remove matrix columns, the app deletes the signals, clears any plots of them, and creates new signals with the modified matrix dimensions.

Matrices, timetables, time series, and labeled signal sets containing nested channels in a hierarchical structure are shown in a tree view that displays the hierarchy explicitly.

**Note** **Signal Analyzer** treats timetables as multichannel signals, even if they have only one channel.

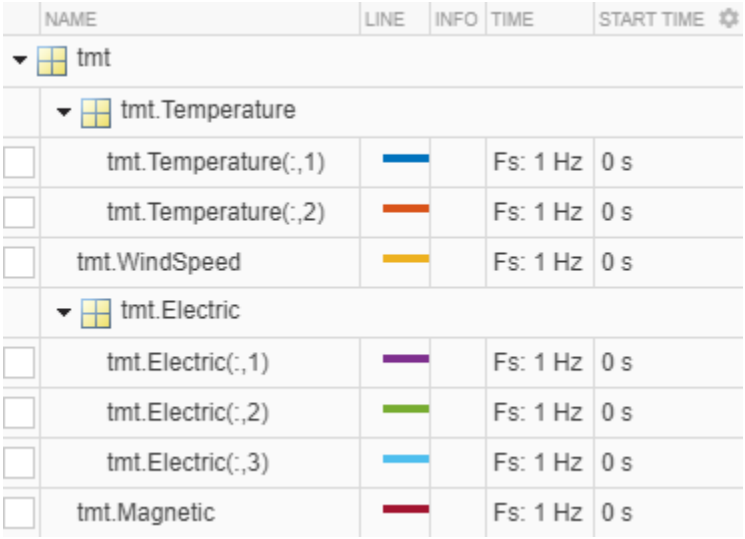
- **Example:** A 100-by-3 matrix called `sgn` appears in the Signal table as `sgn`. If you expand the tree view, you can see the three individual columns, labeled `sgn(:,1)`, `sgn(:,2)`, and `sgn(:,3)`.




- **Example:** Create a timetable with four variables. `"Temperature"` has two channels, `"WindSpeed"` has one channel, `"Electric"` has three channels, and `"Magnetic"` has one channel.

```
tmt = timetable(seconds(0:99)', ...
    randn(100,2),randn(100,1),randn(100,3),randn(100,1));
tmt.Properties.VariableNames = ...
    ["Temperature" "WindSpeed" "Electric" "Magnetic"];
```

Drag the timetable to the Signal table. Expand the tree view to see the individual channels.

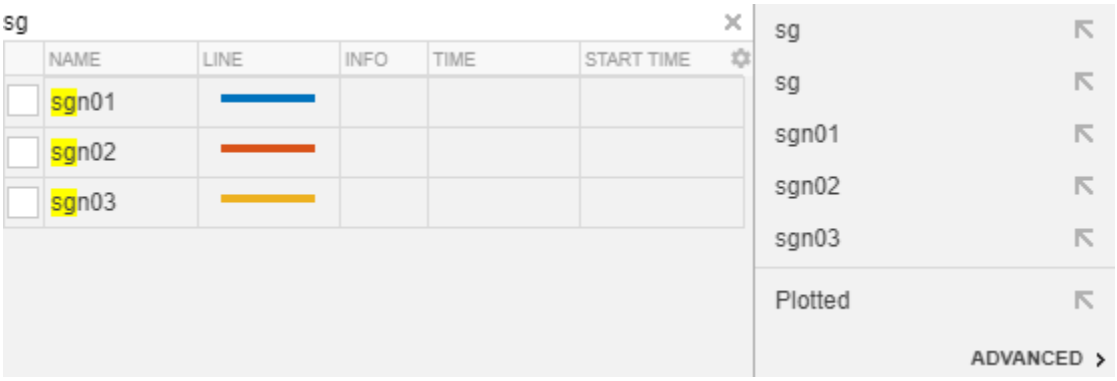



## Filter Signals in the Signal Table

To help search through a large amount of data in the Signal table, you can filter signals. The filter criteria can be any text that is contained in the signal name or in other columns.

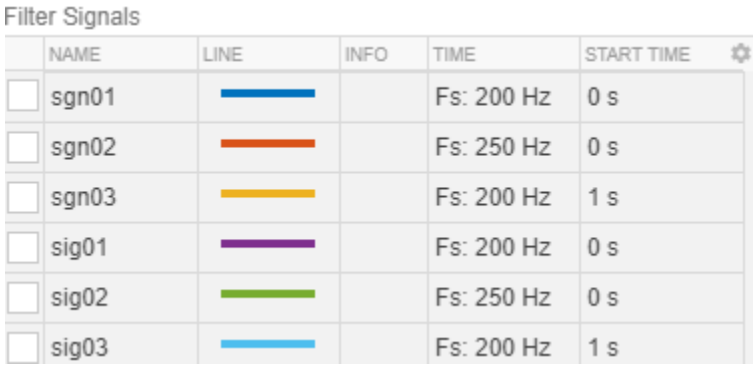
- To show signals with a given name, enter a search phrase into the **Filter Signals** text box. The matches are highlighted in the filter results.

  Suppose that you have three *sig* signals, `sig01`, `sig02`, and `sig03`, and three *sgn* signals, `sgn01`, `sgn02`, and `sgn03`. You can enter `sg` to show the three *sgn* signals, or enter `2` to show `sig02` and `sgn02`.




- You can also filter signals according to their time information. To access this functionality, click inside the search results box, and then click **Advanced**. For details on entering time information, see Edit Sample Rate and Other Time Information on page 21-103.

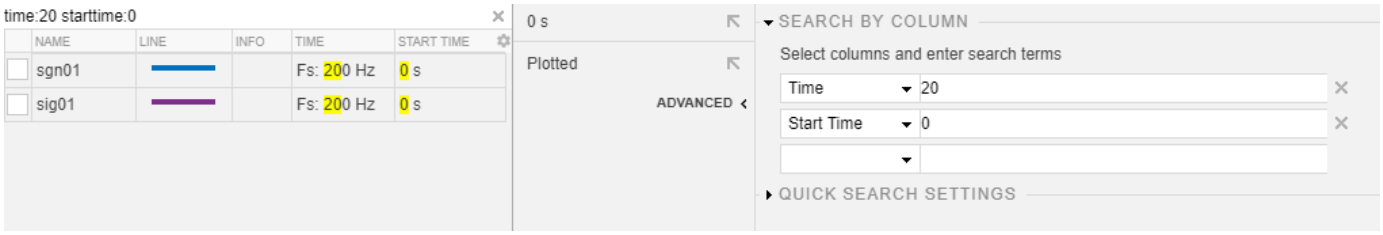
  Suppose that you have six signals with these sample times and start times:

  The **Advanced** menu lets you search signals by `Name`, `Samples`, `Start Time`, or `Time` in terms of sample rate or sample time.

  If you select the `Time` option and enter `20`, the app finds the four signals sampled at 200 Hz. If you also select the `Start Time` option in the second text box and enter `0`, the app finds `sgn01` and `sig01`.

  **Note** The filter matches values as text, not numbers. For instance, if you choose the `Start Time` option and enter `00`, then the filter does not return any results.




- You can save and store a filter for future use. From the **Advanced** menu of the search results box, click **Quick Search Settings**. Enter a name in the **Save Search As** box, and click **Save**.

## Next Step

“Preprocess Signals” on page 21-13

## See Also

**Signal Analyzer**

## Related Examples

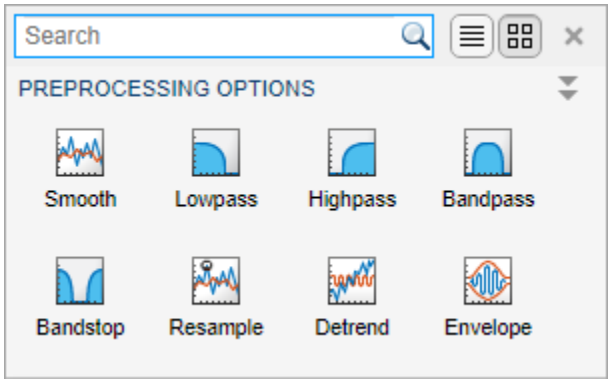
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Spectrum Computation in Signal Analyzer” on page 21-109

# Preprocess Signals

You can use the **Signal Analyzer** app to perform several signal preprocessing tasks. The different processing options appear in the **Analyzer** tab:




Preprocessing operations, undo operations, and function generation apply to all signals currently selected in the Signal table. To select signals, click their **Name**, **Info**, **Time**, or **Start Time** column in the Signal table.

**Note**

- Preprocessing is not supported for labeled signal sets. If you want to preprocess a signal that belongs to a labeled signal set, you must first extract the signal from the set. See “Extract Signal Regions of Interest” on page 21-24 for more information.
- Preprocessing operations overwrite the signal on which they work. If you want to keep the original signal, duplicate it and operate on the duplicate.

You can perform preprocessing actions any number of times and in any order. The **Info** column in the Signal table includes an icon that indicates if any preprocessing has been performed on a signal. Clicking the icon enumerates the actions and the order in which they were performed. Preprocessing steps can be undone by clicking **Undo Preprocessing** on the **Analyzer** tab or on any tab arising from a preprocessing action. The steps are undone one at a time, starting with the most recent.

**Tip** To see a full summary of the preprocessing steps you took, including all settings you chose, click **Generate Function** on the **Analyzer** tab.

You can preprocess individual channels of a multichannel signal. If you select a multichannel signal and one of its channels for preprocessing, the app preprocesses the individual channel only once.

## Duplicate and Rename Signals

**Signal Analyzer** enables you to duplicate and rename signals that you can then preprocess or export for further analysis.

To duplicate a signal, use the **Duplicate** button on the **Analyzer** tab or on any tab arising from a preprocessing action. Alternatively, right-click the signal in the Signal table and select **Duplicate**. The duplicate has the same name as the original signal with `_copy` appended.

If you select a signal and one of its channels for duplication, the app creates a duplicate of the signal and an independent duplicate of the selected channel.

To rename a signal, double-click the signal name in the Signal table and change the name. Alternatively, right-click the signal in the Signal table and select **Rename**.

**Note** You cannot rename individual channels of a multichannel signal.

## Filter Signals

To filter one or more selected signals, on the **Analyzer** tab, click the **Lowpass**, **Highpass**, **Bandpass**, or **Bandstop** icon in the **Preprocessing** gallery. The app uses the `lowpass`, `highpass`, `bandpass`, and `bandstop` functions to perform the filtering. You can control the stopband attenuation, the passband frequencies, and the widths of the transition regions. See the function reference pages for additional information. Filtering does not support nonuniformly sampled signals.

## Smooth Signals

To smooth one or more selected signals, on the **Analyzer** tab, click the **Smooth** icon in the **Preprocessing** gallery. The app uses the MATLAB function `smoothdata` to perform the smoothing. The following smoothing methods are available:

- Moving mean
- Moving median
- Gaussian
- Linear regression
- Quadratic regression
- Robust linear regression
- Robust quadratic regression
- Savitzky-Golay filtering

## Resample Signals

To resample one or more selected signals, on the **Analyzer** tab, expand the **Preprocessing** gallery and click the **Resample** icon. **Signal Analyzer** uses the Signal Processing Toolbox function `resample` to perform the resampling. The following options are available:

- When your signal is nonuniformly sampled, you can use the app to interpolate it onto a uniform grid. You can specify the interpolation method and the sample rate at which you want the signal to be sampled. The following interpolation methods are available:
  - Linear interpolation
  - Shape-preserving piecewise cubic interpolation

  - Cubic spline interpolation using not-a-knot end conditions

  See the `interp1` reference page for more information.
- When your signal is uniformly sampled, you can use the app to change its sample rate. You can specify either the desired sample rate or the factor by which you want to upsample or downsample the signal. In this case, the interpolation panel in the **Resample** tab is disabled because the interpolation operation does not make sense with uniformly sampled signals.

The resampling operation requires time information. If you try to resample a signal in samples, the app issues a warning.

## Detrend Signals

To detrend one or more selected signals, on the **Analyzer** tab, expand the **Preprocessing** gallery and click the **Detrend** icon. **Signal Analyzer** uses the MATLAB function `detrend` to perform the detrending. The app can remove the following trends from signals:

- Constant trends.
- Linear trends.
- Piecewise linear trends. To remove a piecewise linear trend, specify the breakpoints as a comma-separated list.

## Compute Signal Envelopes

To compute the envelope of one or more selected signals, on the **Analyzer** tab, expand the **Preprocessing** gallery and click the **Envelope** icon. **Signal Analyzer** uses the Signal Processing Toolbox function `envelope` to estimate envelopes. You can compute the upper envelope or the lower envelope of each signal. The following envelope estimation algorithms are available:

- `Hilbert` — The app computes the signal envelope as the magnitude of the analytic signal found using the discrete Fourier transform as implemented in `hilbert`.
- `FIR` — The app computes the signal envelope by filtering the signal with a Hilbert FIR filter of adjustable size and using the result as the imaginary part of the analytic signal.
- `RMS` — The app computes the signal envelope by connecting RMS values computed using a moving window of adjustable length.
- `Peak` — The app computes the signal envelope by using spline interpolation over local maxima separated by an adjustable number of samples.

**Note** Envelope computation does not support complex signals.

## Add Custom Preprocessing Functions

To add a custom preprocessing function, on the **Analyzer** tab, click the arrow next to the **Preprocessing** gallery and then select **Add Custom Function**. The app prompts you to enter the function name and a brief description:

- If you have already written a preprocessing function, and the function is in the current folder or in the MATLAB path, the app incorporates it to the gallery. You can use tab completion to search for the function name.

- If you have not written the function yet, the app opens a blank template in the Editor.

Custom preprocessing functions have mandatory and optional arguments:

- The first input argument, `x`, is the input signal. This argument must be a vector and is treated as a single channel.
- The second input argument, `tIn`, is a vector of time values. The vector must have the same length as the signal. If the input signal has no time information, the function reads this argument as an empty array.
- Use `varargin` to specify additional input arguments. If you do not have additional input arguments, you can omit `varargin`. Enter the additional arguments as an ordered comma-separated list in the **Preprocess** tab.
- The first output argument, `y`, is the preprocessed signal.
- The second output argument, `tOut`, is a vector of output time values. If the input signal has no time information, `tOut` is returned as an empty array.
- To implement your algorithm, you can use any MATLAB or Signal Processing Toolbox function.

See “Declip Saturated Signals Using Your Own Function” on page 21-91 for more details.

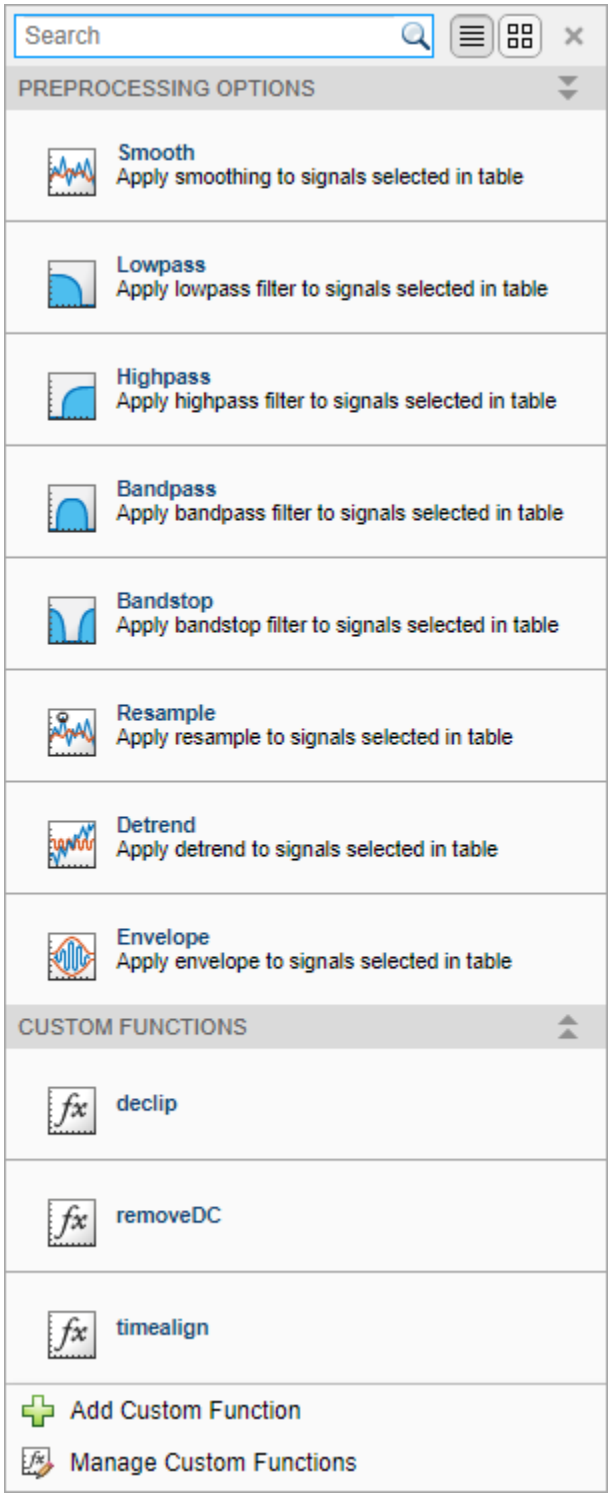
**Example:** This function removes the DC value of a signal by subtracting its mean.

```
function [y,tOut] = removeDC(x,tIn)
% Remove the DC value of a signal by subtracting its mean
   y = x - mean(x);
   tOut = tIn;
end
```

**Example:** This function changes the starting time of a signal to a specified value.

```
function [y,tOut] = timealign(x,tIn,startTime)
% Change the starting time of a signal
   y = x;
   t = tIn;
   if ~isempty(t)
       t = t - t(1) + startTime;
   end
   tOut = t;
end
```

At any time, you can edit functions, edit their descriptions, or remove them, using the **Manage Custom Functions** option in the gallery.




**Note** Custom preprocessing functions must not change the complexity of the input signal.

## Previous Step

“Select Signals to Analyze” on page 21-9

## Next Step

“Explore Signals” on page 21-19

## See Also

**Signal Analyzer**

## More About


- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126
- “Customize Signal Analyzer” on page 21-130

# Explore Signals

## Plot Signals

Select a signal by clicking its name in the Workspace browser or the Signal table. Then plot your selection by dragging it to a display. This action also selects the check box to the left of the signal **Name** on the Signal table. You can also plot a signal by selecting this check box. The app displays a set of axes with the time-domain waveform and a **Time** tab with options to control the view.

If you drag a matrix from the Workspace browser to a display, the app automatically plots each column as a separate signal, up to a maximum of 10 columns. The app creates signals in the Signal table for the remaining columns, but you must drag the additional signals to the display.

**Note** Signals with no time information are plotted in units of samples on the *x*-axis. Signals with time information are plotted in units of time on the *x*-axis. To plot several signals on the same display, ensure that they all have time information or are all in samples. Otherwise, you get a warning.

## View Signals on Multiple Plots

Click **Display Grid** to create or remove displays.

## Move Signals Between Displays

To move a signal from one display to another, click the plotted line or select its name on its **Legend**, for example, ■ sgn01. Click the resulting thicker line and drag it to the target display.

**Note** If you move the real part or the imaginary part of a complex signal from one display to another, the app moves both parts of the signal.

## Visualize Signal Spectra

Use the **Signal Analyzer** app to analyze signals in the frequency domain. To activate the frequency-domain view of a signal, click **Spectrum** ▼ on the **Display** tab and select `Spectrum`. The app displays a set of axes with the signal spectrum, and a **Spectrum** tab with options to control the view.

- If the panner is activated and is zoomed in on a particular region of interest, the spectrum in the display corresponds to the region of interest, not the whole signal.
- If you zoom in on a region of the signal in the time plot using one of the zoom actions on the **Display** tab, the spectrum in the display corresponds to the region of interest, not the whole signal.
- You cannot zoom out in frequency beyond the Nyquist range.
- To see a time plot and a spectrum plot of the same signal side-by-side, use different displays. Drag the signal to two displays. Click **Time** or **Spectrum** on the **Display** tab to control what is plotted on each display.

**Signal Analyzer** scales the spectrum so that, if the frequency content of a signal falls exactly within a bin, its amplitude in that bin is the true average power of the signal. For example, the average power of a sinusoid is one-half the square of the sinusoid amplitude. For more details, see “Measure Power of Deterministic Periodic Signals” on page 24-289. For more information on how **Signal Analyzer** computes spectra, see “Spectrum Computation in Signal Analyzer” on page 21-109.

**Note** When displaying a spectrum, **Signal Analyzer** converts the power to dB using $10\log_{10}(\text{Power})$.

If any complex signals are plotted, **Signal Analyzer** displays centered two-sided spectra.

If a signal is nonuniformly sampled, then **Signal Analyzer** interpolates the signal to a uniform grid to compute spectral estimates. The app uses linear interpolation and assumes a sample time equal to the median of the differences between adjacent time points. For a nonuniformly sampled signal to be supported, the median time interval and the mean time interval must obey

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

## Visualize Persistence Spectra

Use the **Signal Analyzer** app to visualize the persistence spectrum of a signal: The persistence spectrum contains time-dependent probabilities of occurrence of signals at given frequency locations and power levels. This type of spectrum is useful for detecting brief events.

To activate the persistence spectrum, click **Spectrum** ▼ on the **Display** tab and select `Persistence Spectrum`. The app displays a set of axes with the persistence spectrum, and a **Persistence Spectrum** tab with options to control the view. You cannot zoom out in frequency beyond the Nyquist range.

**Note** You can plot the persistence spectrum of only one signal per display.

For more information on how **Signal Analyzer** computes persistence spectra, see “Persistence Spectrum in Signal Analyzer” on page 21-113.

For complex input signals, **Signal Analyzer** displays centered two-sided persistence spectra.

## Visualize Signal Spectrograms

Use the **Signal Analyzer** app to analyze a signal in the time-frequency domain. To activate the spectrogram view of a signal, click **Time-Frequency** ▼ on the **Display** tab and select `Spectrogram`. The app displays a set of axes with the signal spectrogram, and a **Spectrogram** tab with options to control the view.

**Note** You can plot the spectrogram of only one signal per display.

- If the panner is activated and is zoomed in on a particular region of interest, the spectrogram in the display corresponds to the region of interest, not the whole signal.

- If you zoom in on a region of the signal in the time plot using one of the zoom actions on the **Display** tab, the spectrogram in the display corresponds to the region of interest, not the whole signal.
- You cannot zoom out in frequency beyond the Nyquist range.
- To see a time plot and a spectrogram plot of the same signal side-by-side, use different displays.

  Drag the signal to two displays. Click **Time** or **Time-Frequency** on the **Display** tab to control what is plotted on each display.

For more information on how **Signal Analyzer** computes spectrograms, see “Spectrogram Computation in Signal Analyzer” on page 21-115.

The reassignment technique sharpens the time and frequency localization of spectrograms by reassigning each power spectrum estimate to the location of its center of energy. If your signal contains well-localized temporal or spectral components, then this option generates a spectrogram that is easier to read and interpret. To apply reassignment to a spectrogram, check **Reassign** in the **Spectrogram** tab.

If a signal is nonuniformly sampled, then **Signal Analyzer** interpolates the signal to a uniform grid to compute spectral estimates. The app uses linear interpolation and assumes a sample time equal to the median of the differences between adjacent time points. For a nonuniformly sampled signal to be supported, the median time interval and the mean time interval must obey

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

For complex input signals, **Signal Analyzer** displays centered two-sided spectrograms.

## Visualize Signal Scalograms

Use the **Signal Analyzer** app to visualize the scalogram of a signal. The scalogram is useful for identifying signals with low-frequency components and for analyzing signals whose frequency content changes rapidly with time. You need a Wavelet Toolbox license to use the scalogram view.

To activate the scalogram view of a signal, click **Time-Frequency** ▼ on the **Display** tab and select `Scalogram`. The app displays a set of axes with the signal scalogram and a **Scalogram** tab with options to control the view.

**Note** You can plot the scalogram of only one signal per display.

- If the panner is activated and is zoomed in on a particular region of interest, the scalogram in the display corresponds to the whole signal, not just the region of interest. **Signal Analyzer** performs an optical zooming, using interpolation to display a smooth curve.
- If you zoom in on a region of the signal in the time plot using one of the zoom actions on the **Display** tab, the scalogram in the display corresponds to the whole signal, not just the region of interest. **Signal Analyzer** performs an optical zooming, using interpolation to display a smooth curve.
- To see a time plot and a scalogram plot of the same signal side by side, use different displays. On the **Display** tab, click **Display Grid**, create a side-by-side pair of displays, and drag-and-drop the

signal on both displays. Click **Time** or **Time-Frequency** on the **Display** tab to control what is plotted on each display.

**Note**

- Scalogram view does not support complex signals.
- Scalogram view does not support nonuniformly sampled signals.

For more information on how **Signal Analyzer** computes scalograms, see “Scalogram Computation in Signal Analyzer” on page 21-120.

## Zoom and Pan Through Signals

The **Signal Analyzer** app features a panner that enables you to zoom in on and navigate through signals to see how they change in frequency and time. To activate the panner, on the **Display** tab, click **Panner** .

The panner renders signals in their entire duration. To select a region of interest, click the panner and drag to create a zoom window. Use the mouse to resize or slide the zoom window along the length of the signal.

- If the spectrum of the signal is plotted, it corresponds to the region of interest, not the whole signal. For more details, see “Spectrum Computation in Signal Analyzer” on page 21-109.
- If the persistence spectrum of the signal is plotted, it corresponds to the region of interest, not the whole signal. For more details, see “Persistence Spectrum in Signal Analyzer” on page 21-113.
- If the spectrogram of the signal is plotted, it corresponds to the region of interest, not the whole signal. For more details, see “Spectrogram Computation in Signal Analyzer” on page 21-115.
- If the scalogram of the signal is plotted, it corresponds to the whole signal, not the region of interest. **Signal Analyzer** performs an optical zooming, using interpolation to display a smooth curve. For more details, see “Scalogram Computation in Signal Analyzer” on page 21-120.
- You cannot zoom out in frequency beyond the Nyquist range.

## Edit Time Information and Link Displays in Time

Use the **Signal Analyzer** app to add time information to signals. In the Signal table, select the signals whose time information you want to add or modify. Add time information to the signals by clicking **Time Values** in the **Analyzer** tab.

**Note**

- You cannot edit the time information of a timetable or time series with inherent time information.
- You cannot edit the time information of a labeled signal set.
- You cannot edit the time information for individual channels of a multichannel signal. You must edit the time information for the whole signal.

You can express the time information in terms of a sample rate or sample time, and a start time. You can also add explicit time values using a numeric vector, a `duration` array, or a MATLAB expression. Time values must be unique and cannot be `NaN`, but they need not be uniformly spaced. The app derives a sample rate from the time values and displays it in the **Time** column of the Signal table. For more details, see “Edit Sample Rate and Other Time Information” on page 21-103.

**Note** Filtering and scalogram view do not support nonuniformly sampled signals.

- If a signal is nonuniformly sampled, then **Signal Analyzer** interpolates the signal to a uniform grid to compute spectral estimates. The app uses linear interpolation and assumes a sample time equal to the median of the differences between adjacent time points. The derived sample rate in the Signal table has an asterisk to indicate that the signal is nonuniformly sampled. For a nonuniformly sampled signal to be supported, the median time interval and the mean time interval must obey

  $$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

  **Note** The interpolation is used only to compute spectral estimates. Time plots are not resampled.

- You can link display time spans so that plot responses are synchronized when you pan and zoom horizontally. The signals in the displays you want to link must contain time information. To link the time span of a display to the time spans of the displays linked already, select the display and, on the **Display** tab, select **Link Time**. To unlink a display, select it and clear **Link Time**.

  **Note** Selecting **Link Time** links the selected display to the complete collection of displays that have already been linked.

  Displays with linked time spans have the following operations synchronized:

  - Panning by selecting and dragging the plot or by using the display panner.
  - Zooming in, zooming out, or zooming on the time axis. Zooming in or out on one display affects only the time axis in the remaining linked displays.
  - Fitting data to view. The app stretches the common time axis so that it shows the span from the earliest to the latest time among all signals in the linked displays.
  - If the axes of two displays are linked in time, then the time cursors in the displays are linked.

  The time axis of a linked display might update as you add or remove signals.

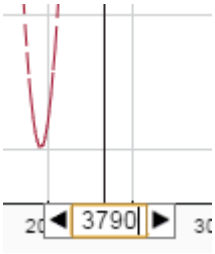
**Note** Frequency axes are never linked between displays.

## Measure Signal, Spectrum, and Time-Frequency Data

Measure your data using data cursors:

1. On the **Display** tab, click **Data Cursors ▼** to add one or two data cursors to all the displays. Time-domain and frequency-domain cursors are not linked and can be moved independently.

2. The persistence spectrum, spectrogram, and scalogram views display two-dimensional crosshair cursors.
3. To move a data cursor, drag it left, right, up, or down to a point of interest. To move the cursor sample-by-sample, click the time or frequency field and use the arrow keys.
4. You can move a data cursor to a specific point without dragging it. Click the data cursor time or frequency field and enter a value.




   If the signal was not sampled at a point of interest, then the app linearly interpolates the value. If the value is interpolated, an asterisk appears in the data cursor label.
5. By default, cursors snap to the nearest data point. To change this behavior, clear the **Snap to Data** check box on the **Display** tab.
6. To toggle the cursors, click **Data Cursors**.

## Extract Signal Regions of Interest

The **Signal Analyzer** app enables you to extract regions of interest from the signals you are studying and export them for further analysis. To extract regions of interest, select the display that has them. On the **Display** tab, click **Extract Signals**, or right-click the display and select **Extract Signals**.

- Select `Between Time Limits` to extract a region of interest defined by the time limits of the selected display. To change the time limits, you can use the panner, select one of the zoom actions on the **Display** tab, or change the limit values on the **Display**, **Time**, **Spectrogram**, or **Scalogram** tabs.
- Select `Between Time Cursors` to extract a region of interest defined by the locations of the time-domain cursors in the selected display.
- If a signal has time information, you can preserve the start time of the region of interest by checking **Preserve Start Time**.

The extracted regions of interest are added at the bottom of the Signal table.

## Previous Step

“Preprocess Signals” on page 21-13

## Next Step

“Share Analysis” on page 21-26

## See Also

**Signal Analyzer**

## Related Examples

- “Spectrogram View of Dial Tone Signal” on page 21-48

## More About

- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126
- “Nonparametric Methods” on page 7-8
- “Customize Signal Analyzer” on page 21-130

# Share Analysis

## Copy Displays

You can share the plots that you have produced using the **Signal Analyzer** app by copying one or more displays to the clipboard as images and pasting them into another application.

To copy displays to the clipboard, on the **Display** tab, click **Copy All Displays ▼**. You can then copy either the selected display or the complete display layout.

To copy a single display to the clipboard, you can also right-click the display and select **Copy Display**.

## Export Signals

You can export any signals in the **Signal Analyzer** Signal table to the MATLAB workspace or to a MAT-file.

To export signals:

1. Select one or more signals from the Signal table.
2. On the **Analyzer** tab, click **Export** .
3. Choose whether you want to export the selected signals to the MATLAB workspace or save them to a MAT-file. If you choose to save the signals, browse to where you want to save the file, name the file, and click **Save**.

You can also select the signals, right-click, and select **Export**.

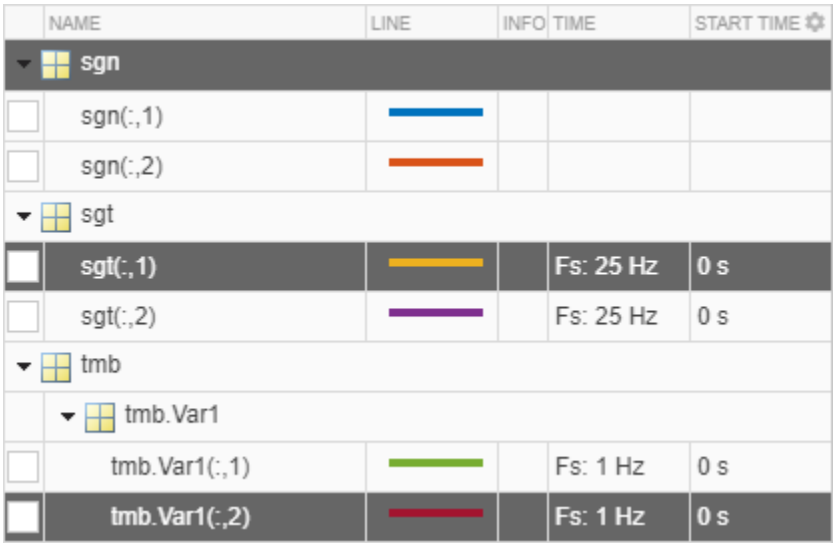
Signals are exported differently, depending on their type:

- Signals with no time information are exported or saved as numeric vectors.
- Signals stored as timetables are exported or saved as timetables.
- Signals that have time information but are not stored as timetables are exported or saved as numeric vectors. If you want to preserve the time information, you can save the signals as timetables. On the **Analyzer** tab, click **Preferences** and check `Always use timetables when signals have time information`.
- The export behavior for multichannel signals depends on the signals and channels that you select and on the preferences you have set.
  - Whenever possible, the app exports signals of the same name and type (numeric or timetable) as the originals.
  - If you select a signal with several channels, the app exports it as a single matrix or timetable if the individual channels have the same length and time information.
  - If you select a signal with several channels that have different lengths or different time information, the app exports them as independent signals.
  - If you select a signal and one or more of its channels at the same time, the app exports a copy of the whole signal and independent variables corresponding to the selected channels.

  **Example:** Create three two-channel signals. Each channel of `sgn` has 100 samples. Each channel of `sgt` has 200 samples. The timetable `tmb` has two 20-sample channels sampled at 1 Hz.

```
sgn = randn(100,2);
sgt = randn(200,2);
tmb = timetable(seconds(0:19)',randn(20,2));
```

Drag the signals to the Signal table. Expand the tree view to see the individual channels. Select `sgt` and, on the **Analyzer** tab, click **Time Values**. Select `Sample Rate and Start Time` and specify a sample rate of 25 Hz. Select `sgn`, the first channel of `sgt`, and the second channel of the only variable of `tmb`.




On the **Analyzer** tab, click **Export** to export the selected signals to a MAT-file. Use the default file name. Load the file into the MATLAB workspace.

```
load New_Export
whos
```

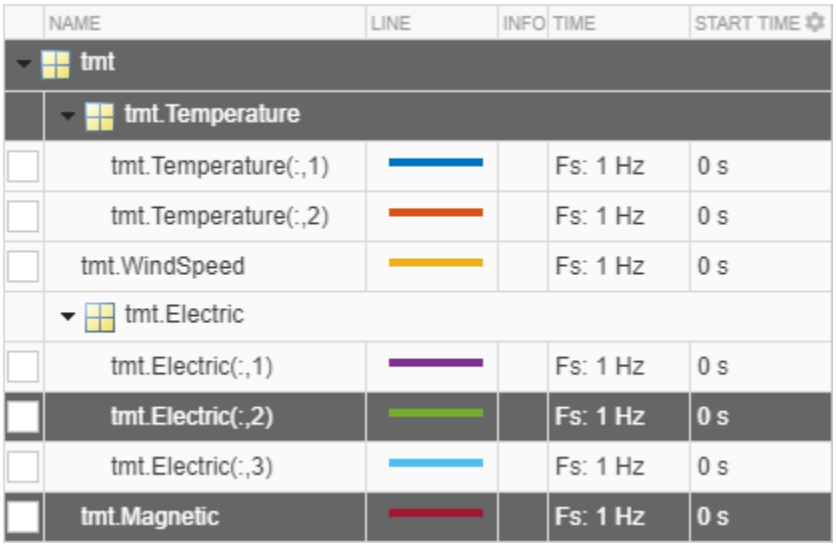
```
  Name              Size            Bytes  Class

  sgn             100x2              1600  double
  sgt_1           200x1              1600  double
  tmb_Var1_2       20x1              1368  timetable
```

The app exported `sgt_1` as a vector, even though it has time information. On the **Analyzer** tab, click **Preferences** and check `Always use timetables when signals have time information`. Export the signals again. `sgt_1` becomes a timetable.

**Example:** Create a timetable with four variables. `"Temperature"` has two channels, `"WindSpeed"` has one channel, `"Electric"` has three channels, and `"Magnetic"` has one channel.

```
tmt = timetable(seconds(0:99)', ...
    randn(100,2),randn(100,1),randn(100,3),randn(100,1));
tmt.Properties.VariableNames = ...
    ["Temperature" "WindSpeed" "Electric" "Magnetic"];
```

Drag the timetable to the Signal table. Expand the tree view to see the individual channels. Select `tmt`, `tmt.Temperature`, the second channel of `tmt.Electric`, and `tmt.Magnetic`.




On the **Analyzer** tab, click **Export** to export the selected signals to a MAT-file. Use the default file name. Load the file into the MATLAB workspace.

```
load New_Export
whos
```

```
  Name                  Size            Bytes  Class

  tmt                 100x4               8180  timetable
  tmt_Electric_2      100x1               2656  timetable
  tmt_Magnetic        100x1               2652  timetable
  tmt_Temperature     100x1               3458  timetable
```

The app exported `tmt` as a four-variable timetable, `tmt_Temperature` as a timetable with a two-channel variable, and the two single-variable, single-channel timetables `tmt_Electric_2` and `tmt_Magnetic`.

## Generate MATLAB Scripts and Functions

You can generate MATLAB scripts to extract signal regions of interest or automate the computation of power spectrum, persistence spectrum, spectrogram, or scalogram estimates obtained with the **Signal Analyzer** app.

To generate a MATLAB script, on the **Display** tab, click **Generate Script**. The generated script opens in the Editor.

- Select `ROI Script Between Time Limits` to generate a MATLAB script that extracts a region of interest defined by the time limits of the selected display. Depending on the preferences, the regions of interest are saved as numeric vectors or as a timetable.
- Select `ROI Script Between Time Cursors` to generate a MATLAB script that extracts a region of interest defined by the locations of the time-domain cursors in the selected display. Depending on the preferences, the regions of interest are saved as numeric vectors or as a timetable.
- Select `Spectrum Script` to generate a MATLAB script that computes the power spectrum appearing in the spectrum view of the selected display, including all current settings.

- Select `Persistence Spectrum Script` to generate a MATLAB script that computes the persistence spectrum appearing in the spectrum view of the selected display, including all current settings.
- Select `Spectrogram Script` to generate a MATLAB script that computes the spectrogram appearing in the spectrogram view of the selected display, including all current settings.
- Select `Scalogram Script` to generate a MATLAB script that computes the scalogram appearing in the scalogram view of the selected display, including all current settings. You need a Wavelet Toolbox license to use the scalogram view.

You can generate MATLAB functions to automate signal preprocessing steps performed with the **Signal Analyzer** app.

To generate a MATLAB preprocessing function, on the **Analyzer** tab, click **Generate Function**. The generated function opens in the Editor.

## Save and Load Signal Analyzer Sessions

If you want to share session snapshots or archive them to view later, save the **Signal Analyzer** session to a MAT-file or MLDATX-file. Using MLDATX-files results in faster save and load times.

To save a session to a MAT-file or MLDATX-file:

1. On the **Analyzer** tab, click **Save ▼** and select `Save`.
2. Browse to where you want to save the file, name the file, choose the format, and click **Save**.

If you want to update the file, click **Save**. If you want to save the session to a different file, click **Save ▼** and select `Save as`.

To load a saved session:

1. On the **Analyzer** tab, click **Open**.
2. Browse to the MAT-file or MLDATX-file saved from a previous session, select it, and click **Open**. The signal data and properties appear as they were when the file was last saved.

To start a new session, on the **Analyzer** tab, click **New**.

## Previous Step

“Explore Signals” on page 21-19

## See Also

**Signal Analyzer**

## Related Examples

- “Compute Envelope Spectrum of Vibration Signal” on page 21-96

## More About

- “Signal Analyzer Tips and Limitations” on page 21-126

# Find Delay Between Correlated Signals

Three sensors at different locations measure vibrations caused by a car as it crosses a bridge. The signals they produce arrive at the analysis station at different times. The sample rate is 11,025 Hz. Use the **Signal Analyzer** app to determine the delays between the signals.

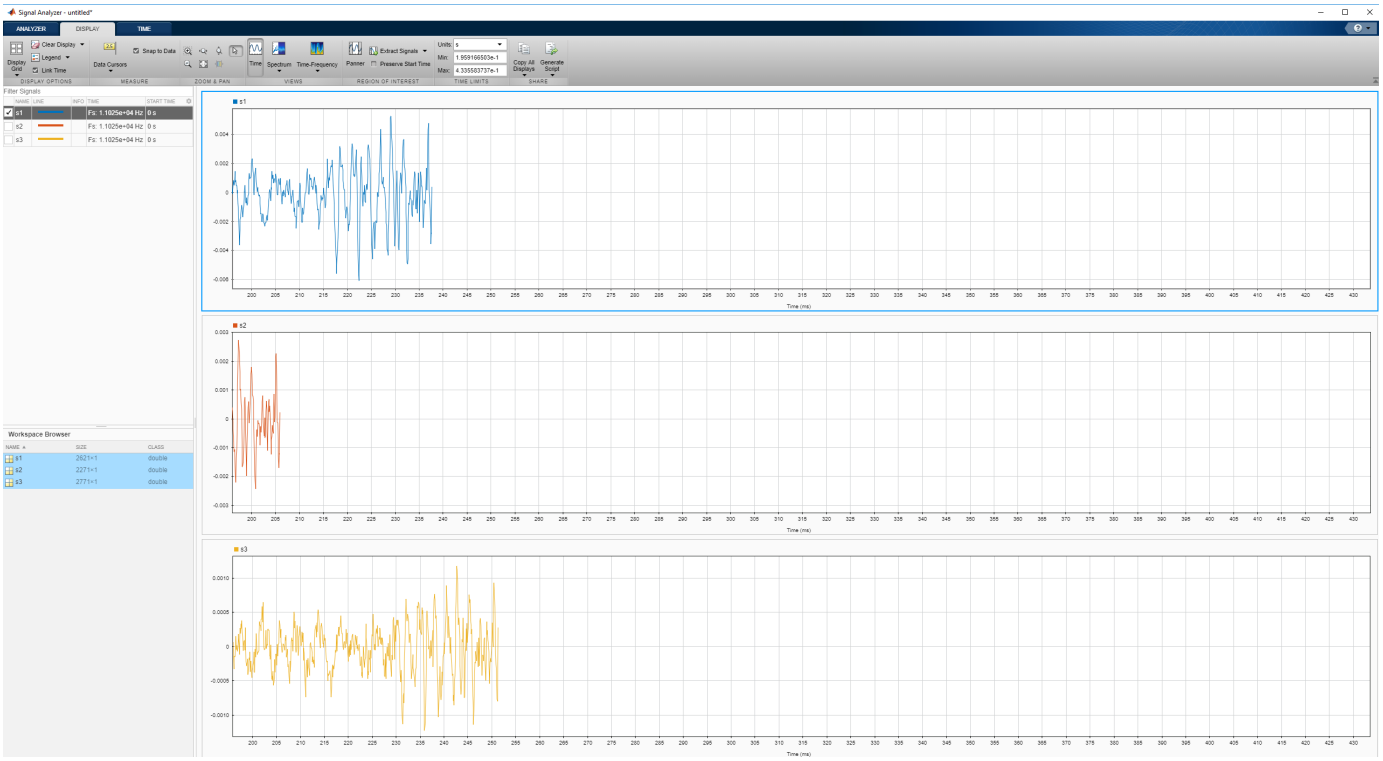
Load the signals into the MATLAB® workspace and start the app. The name of each signal includes the number of the sensor that took it. Create three displays. Drag each signal from the Workspace browser to its own display. The signal from Sensor 2 arrives earlier than the signal from Sensor 1. The signal from Sensor 1 arrives earlier than the signal from Sensor 3.

```
load sensorData
```

Add time information. Select the three signals in the Signal table and click the Time Values button on the Analyzer tab. Select the `Sample Rate and Start Time` option and enter the sample rate of 11,025 Hz. For more information, see Edit Sample Rate and Other Time Information.

The signals share a common time axis. Link their time spans by selecting each display and selecting **Link Time** on the **Display** tab.

To estimate the delays between the signals, pan them horizontally and line up a salient feature to the end of the time axis. From the **Time** tab, read the time from the lower limit of the time-axis. Choose a region where the signal-to-noise ratio is high, such as the signal maximum toward the end of each signal. In the signal from Sensor 2, that feature occurs about 0.197 second after the clock starts.



Similarly, the signal from Sensor 1 has that feature about 0.229 second after the start, and the signal from Sensor 3 has it about 0.243 second after the start. Thus, the delays are approximately 0.032 second and 0.014 second long.

You can also use data cursors to find the delays. Press the space bar to reset the view. On the **Display** tab, click the arrow under the **Data Cursors** ▼ and select Two. Place a cursor on the maximum of each of the top two signals. You can read the lag of approximately 0.032 second directly from the app.

Similarly, the lag between the top and bottom signals is 0.014 second.

You can get similar results with the `finddelay` and `xcorr` functions.

## See Also

### Apps
**Signal Analyzer**

### Functions
`alignsignals` | `finddelay` | `xcorr`

## Related Examples

- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78

- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

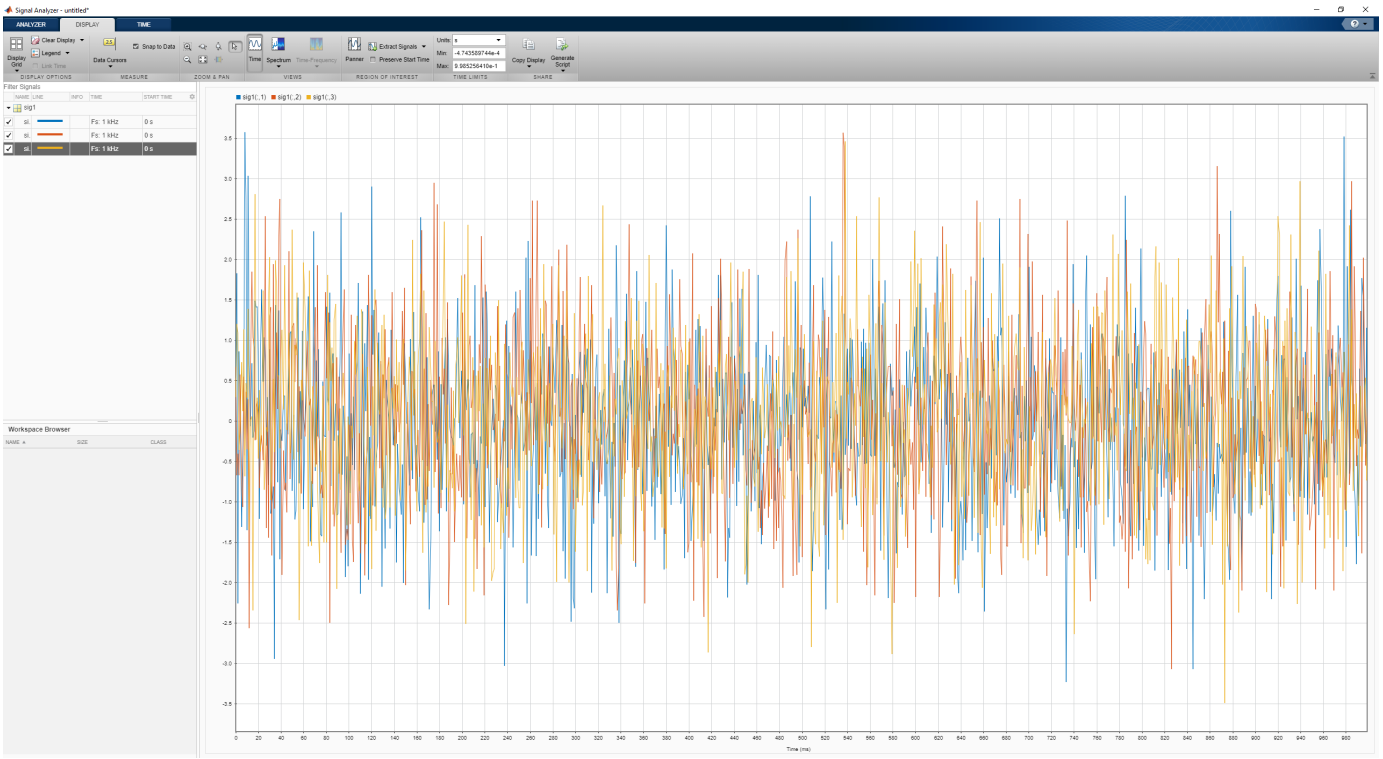
- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Plot Signals from the Command Line

You can plot signals in the **Signal Analyzer** app by calling the app in the command line.

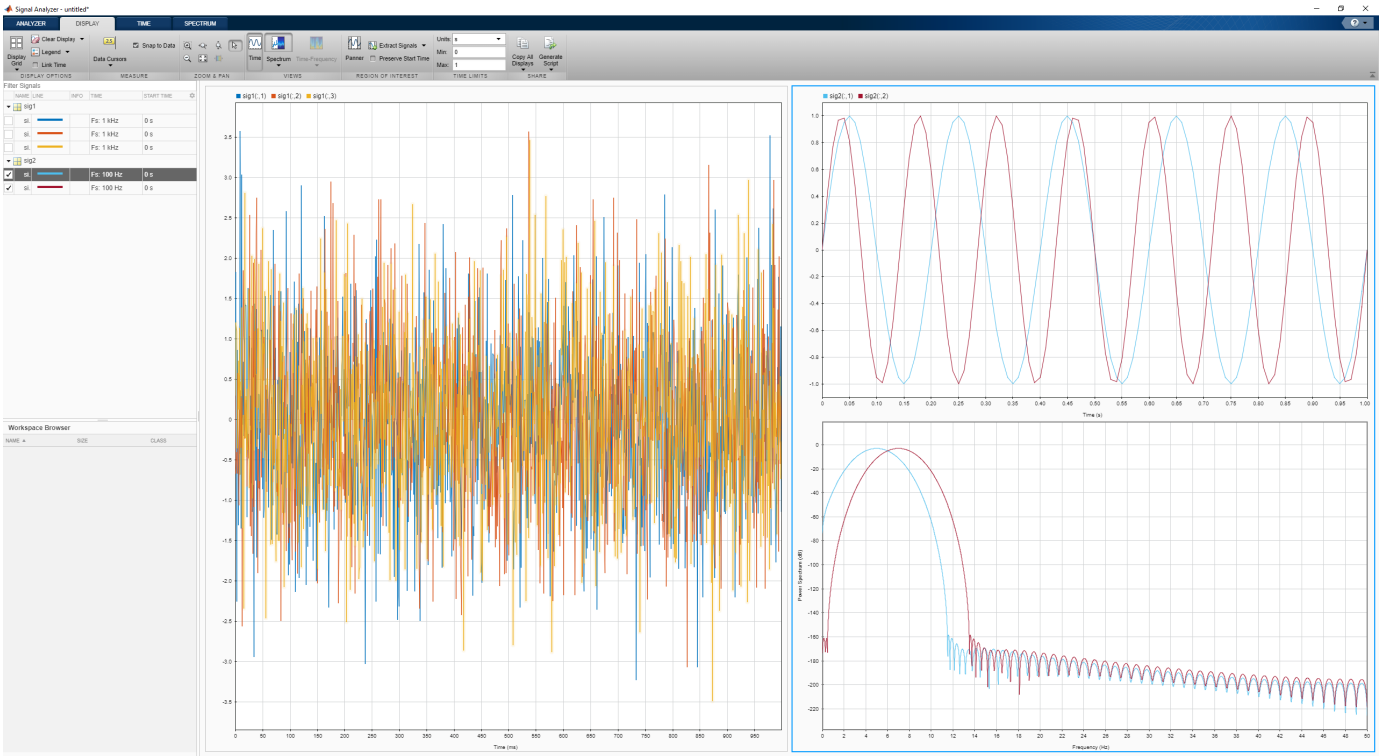
Create a three-channel random signal sampled for 1 second at 1 kHz. Plot the signal in **Signal Analyzer**.

```
signalAnalyzer(randn(1000,3),'SampleRate',1e3)
```



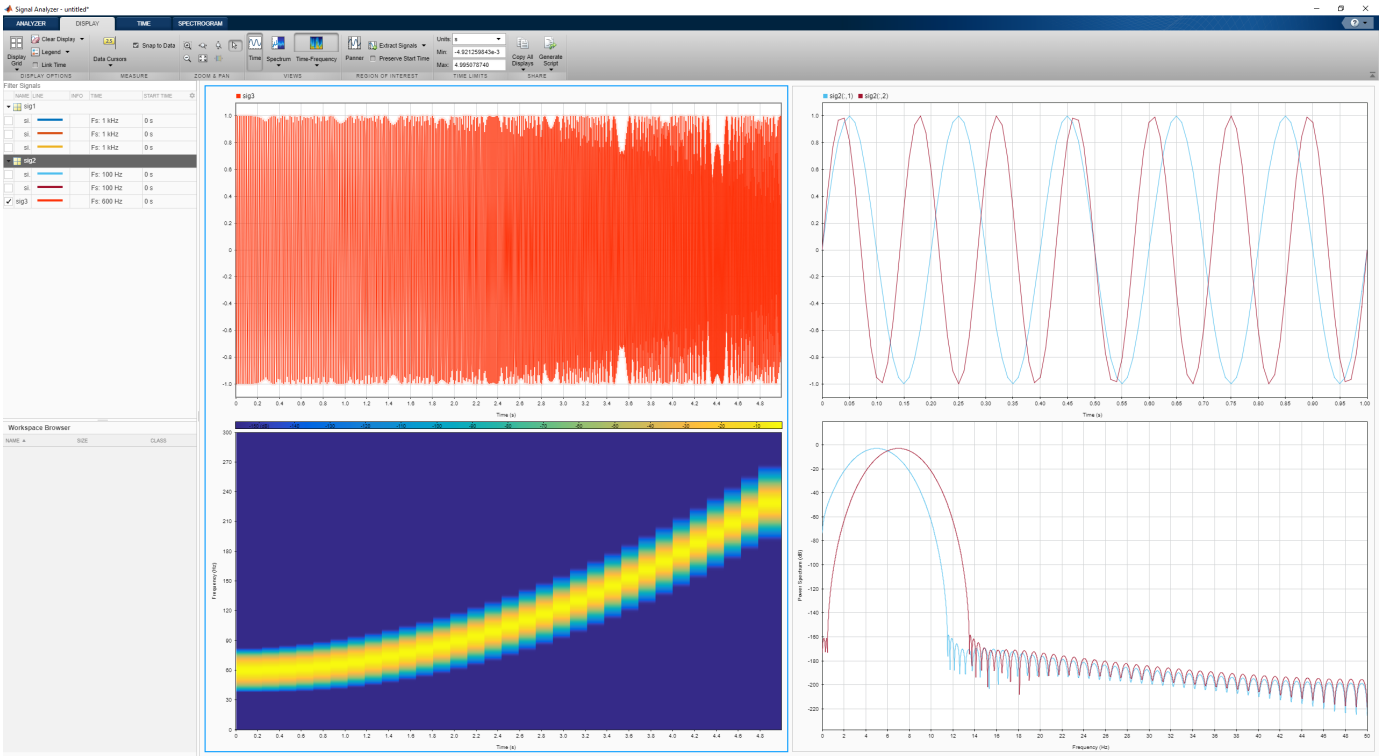
Use **Display Grid** on the **Display** tab to open an empty display. Select the new display and click **Spectrum**. Create a two-channel sinusoidal signal sampled for 1 second at 100 Hz. The sinusoids have frequencies of 5 Hz and 7 Hz. Plot the signal in the new display.

```
signalAnalyzer(sin(2*pi*[5 7].*(0:100)'/100),'SampleRate',100)
```



Select the display with the three random signals. Remove the random signals from the display by clearing the check boxes next to their names. Click **Time-Frequency**. Create a quadratic chirp sampled for 5 seconds at 600 Hz. The chirp has an initial frequency of 60 Hz that increases to 240 Hz by the end of the measurement. Plot the chirp.

```
signalAnalyzer(chirp(0:1/600:5,60,5,240,'quadratic'),'SampleRate',600)
```



## See Also

### Apps
**Signal Analyzer**

### Functions
`chirp`

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96

- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Resolve Tones by Varying Window Leakage

You can adjust the spectral leakage of the analysis window to resolve sinusoids in **Signal Analyzer**.

Generate a two-channel signal sampled at 100 Hz for 2 seconds.

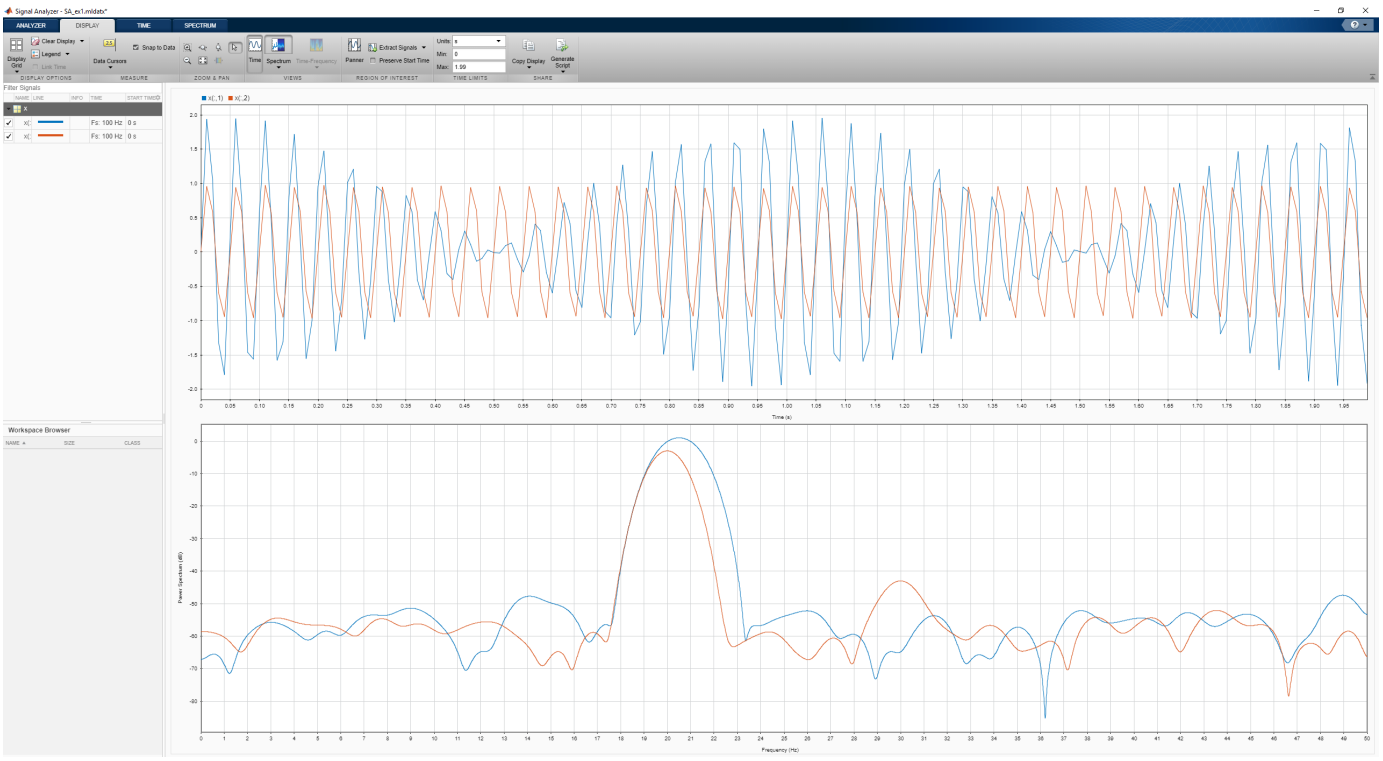
1. The first channel consists of a 20 Hz tone and a 21 Hz tone. Both tones have unit amplitude.
2. The second channel also has two tones. One tone has unit amplitude and a frequency of 20 Hz. The other tone has an amplitude of 1/100 and a frequency of 30 Hz.

```
fs = 100;

t = (0:1/fs:2-1/fs)';

x = sin(2*pi*[20 20].*t)+[1 1/100].*sin(2*pi*[21 30].*t);
```

Embed the signal in white noise. Specify a signal-to-noise ratio of 40 dB. Open **Signal Analyzer** and plot the signal using the sample rate `fs`.

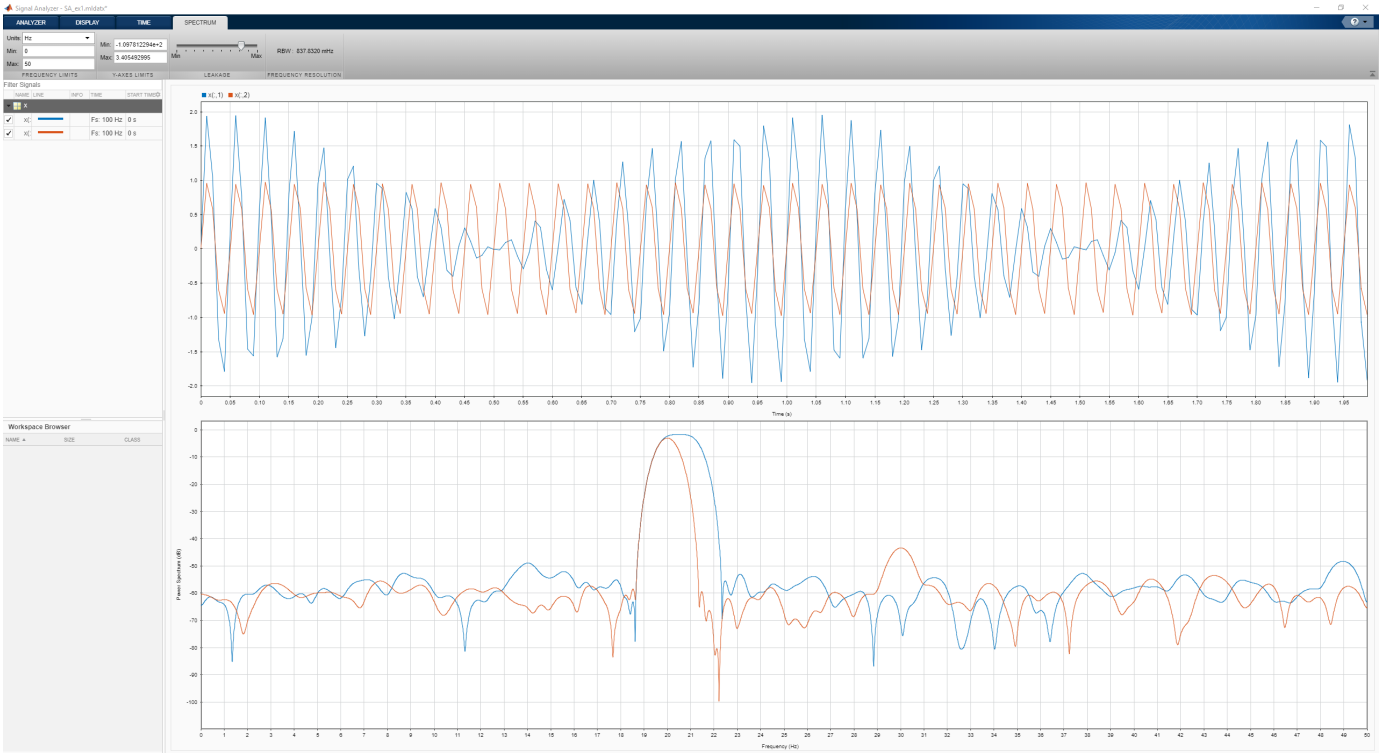
```
x = x + randn(size(x)).*std(x)/db2mag(40);
```

Click **Spectrum** to add spectral plots to the display. Click the **Spectrum** tab that appears.

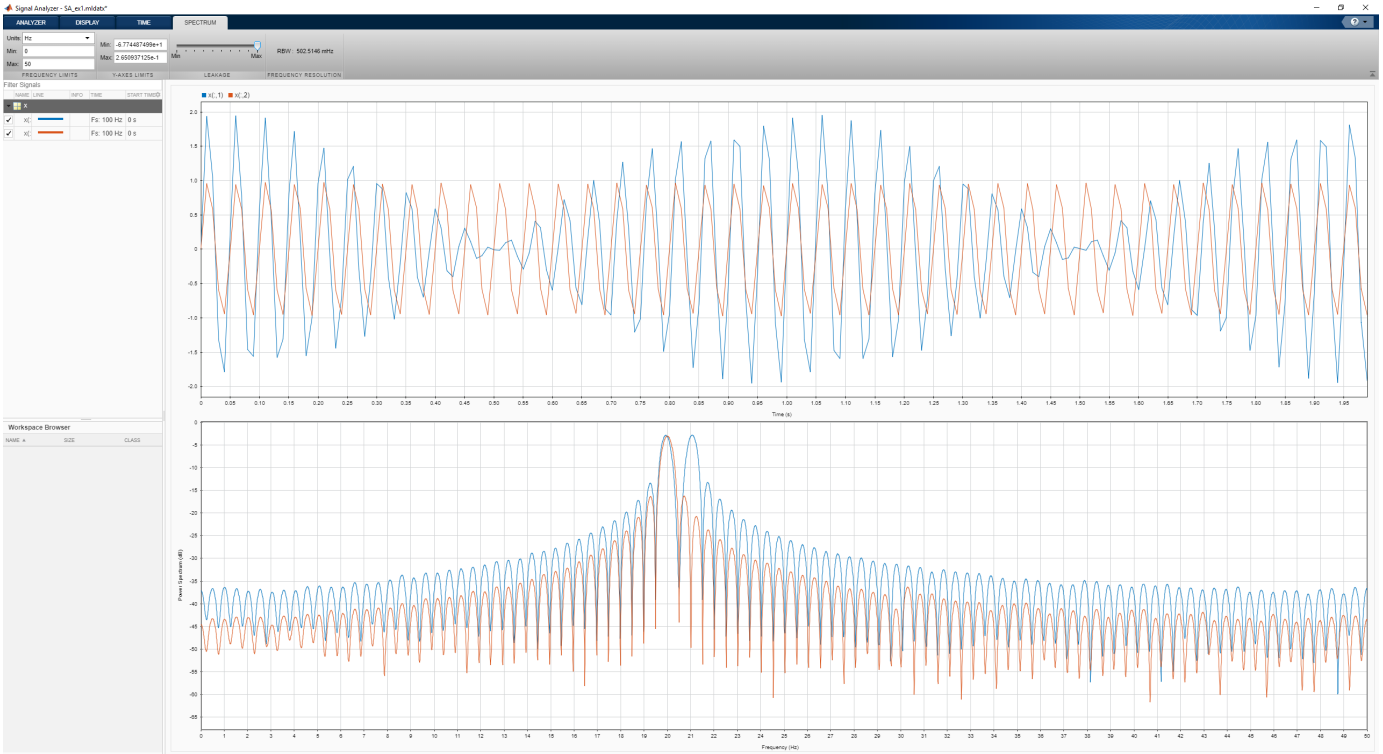


The slider that controls the spectral leakage is in the middle position, corresponding to a resolution bandwidth of about 1.29 Hz. The two tones in the first channel are not resolved. The 30 Hz tone in the second channel is visible, despite being much weaker than the other one.

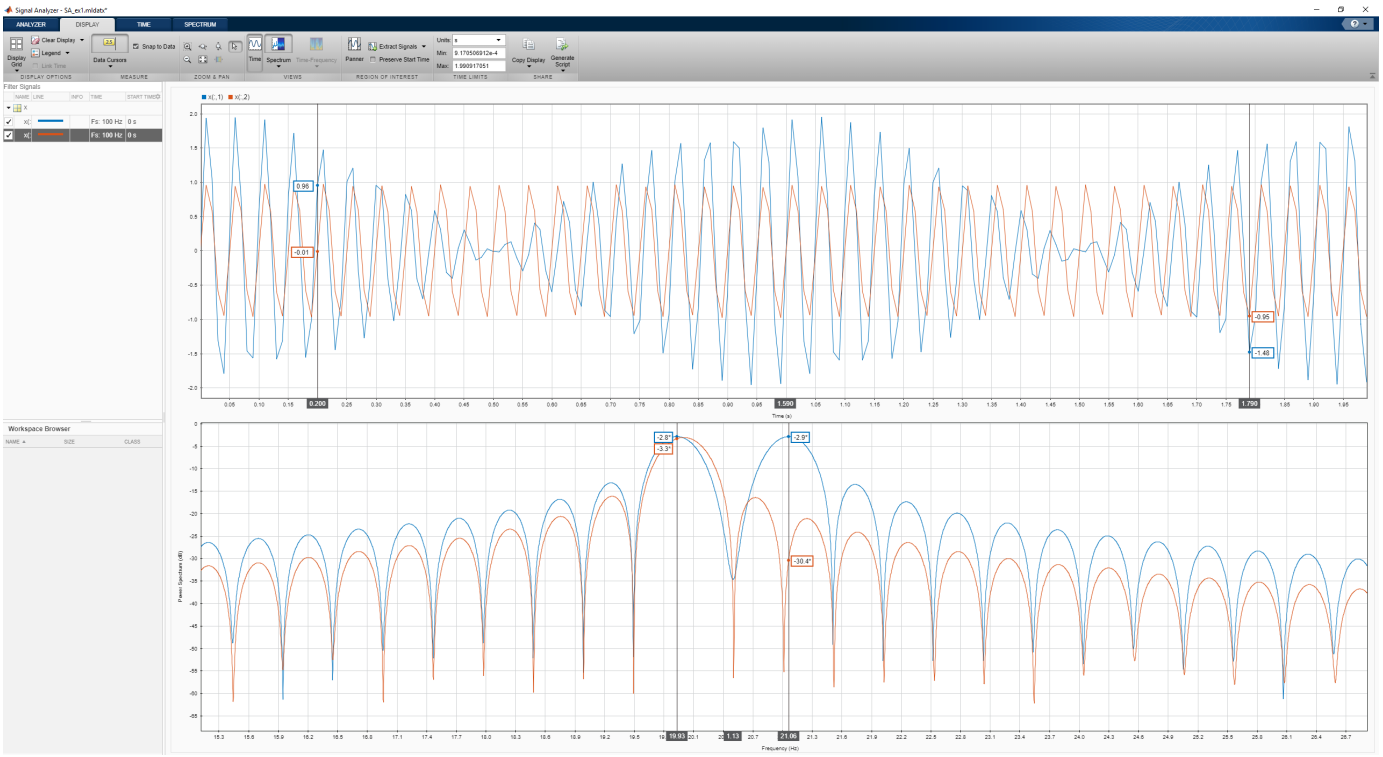
Increase the leakage so that the resolution bandwidth is approximately 0.84 Hz. The weak tone in the second channel is clearly resolved.



Move the slider to the maximum value. The resolution bandwidth is approximately 0.5 Hz. The two tones in the first channel are resolved. The weak tone in the second channel is masked by the large window sidelobes.



Click the **Display** tab. Use the horizontal zoom to magnify the frequency axis. Add frequency-domain cursors to estimate the frequencies of the tones.



## See Also

### Apps
**Signal Analyzer**

### Functions
`enbw` | `pspectrum` | `pwelch`

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84

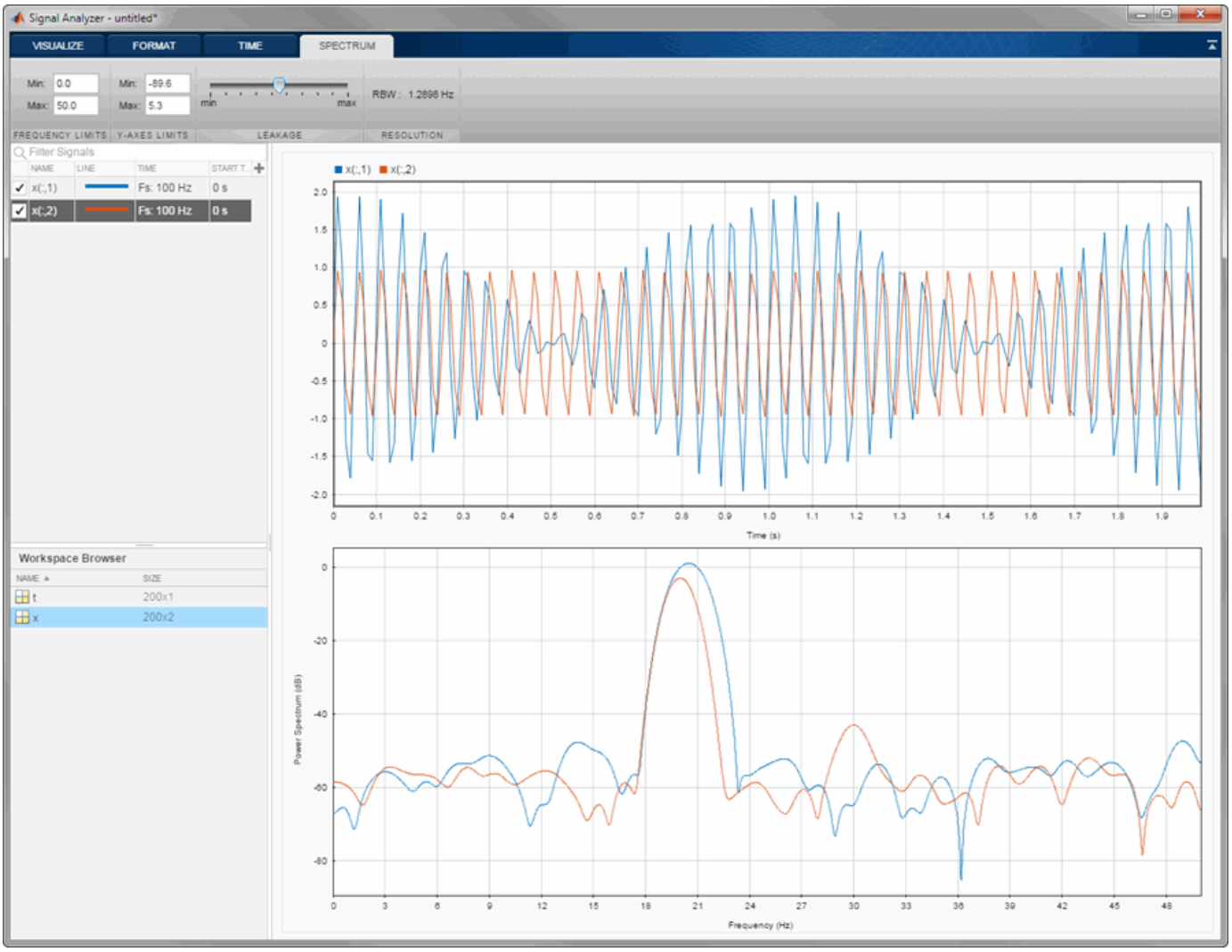
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Resolve Tones by Varying Window Leakage

Go back to example on page 21-39




## See Also

### Apps

**Signal Analyzer**

### Functions

`enbw` | `pspectrum` | `pwelch`

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35

- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Analyze Signals with Inherent Time Information

Load a file that contains measurements of the international normalized ratio (INR) performed on a patient over a five-year period. The INR measures the effect of warfarin, a blood thinner used to treat people predisposed to clotting.

The file includes a `datetime` array with the date and time of each measurement, and a vector of INR readings. Display the first five time points.

```
load INR

Date(1:5)
```

```
ans = 5x1 datetime
   05/15/09 11:28 AM
   06/16/09 09:05 AM
   07/02/09 09:50 AM
   07/16/09 11:20 AM
   07/31/09 12:05 PM
```

Convert the time readings to a `duration` array by subtracting the first time point. Convert the `duration` array to seconds.

```
t = Date-Date(1);
s = seconds(t);
```

Create a timetable with the `duration` array and the INR readings.

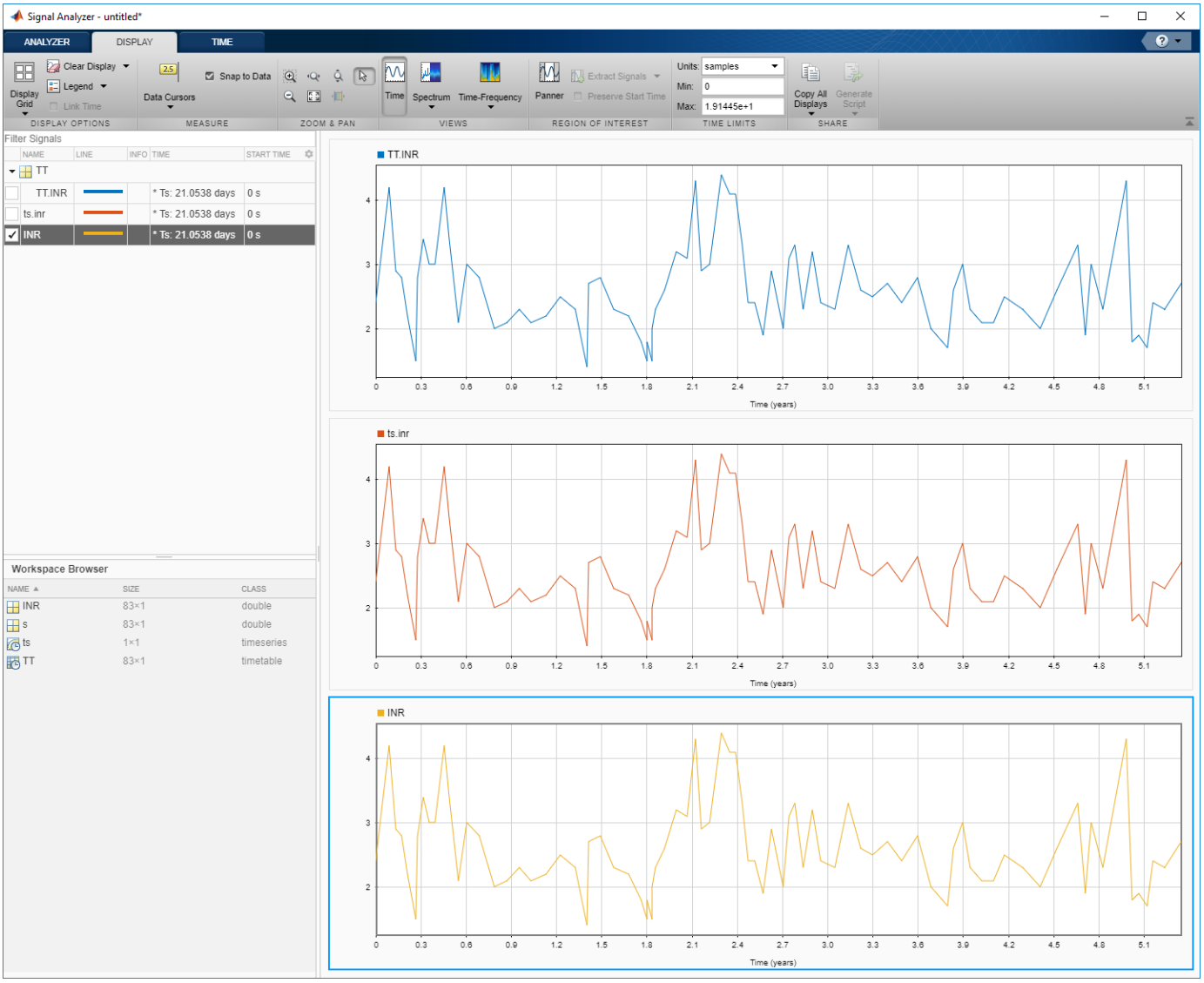
```
TT = timetable(t,INR);
```

Create a `timeseries` object with the time in seconds and the INR readings.

```
ts = timeseries(INR,s,'Name','inr');
```

Open **Signal Analyzer**. Under **Display Grid**, select three displays.

1. Drag the timetable to the first display. The app calls this signal `TT.INR`.
2. Drag the `timeseries` object to the second display. The app calls this signal `ts.inr`.
3. Drag the array of `INR` readings to the third display. Add time information. Select the signal by clicking its legend. From the **Analyzer** tab, click **Time Values**. Select the `Time Values` option and in the **Time Values** field, enter `t`.




## See Also

### Apps
**Signal Analyzer**

### Functions
`timeseries` | `timetable`

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39

- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

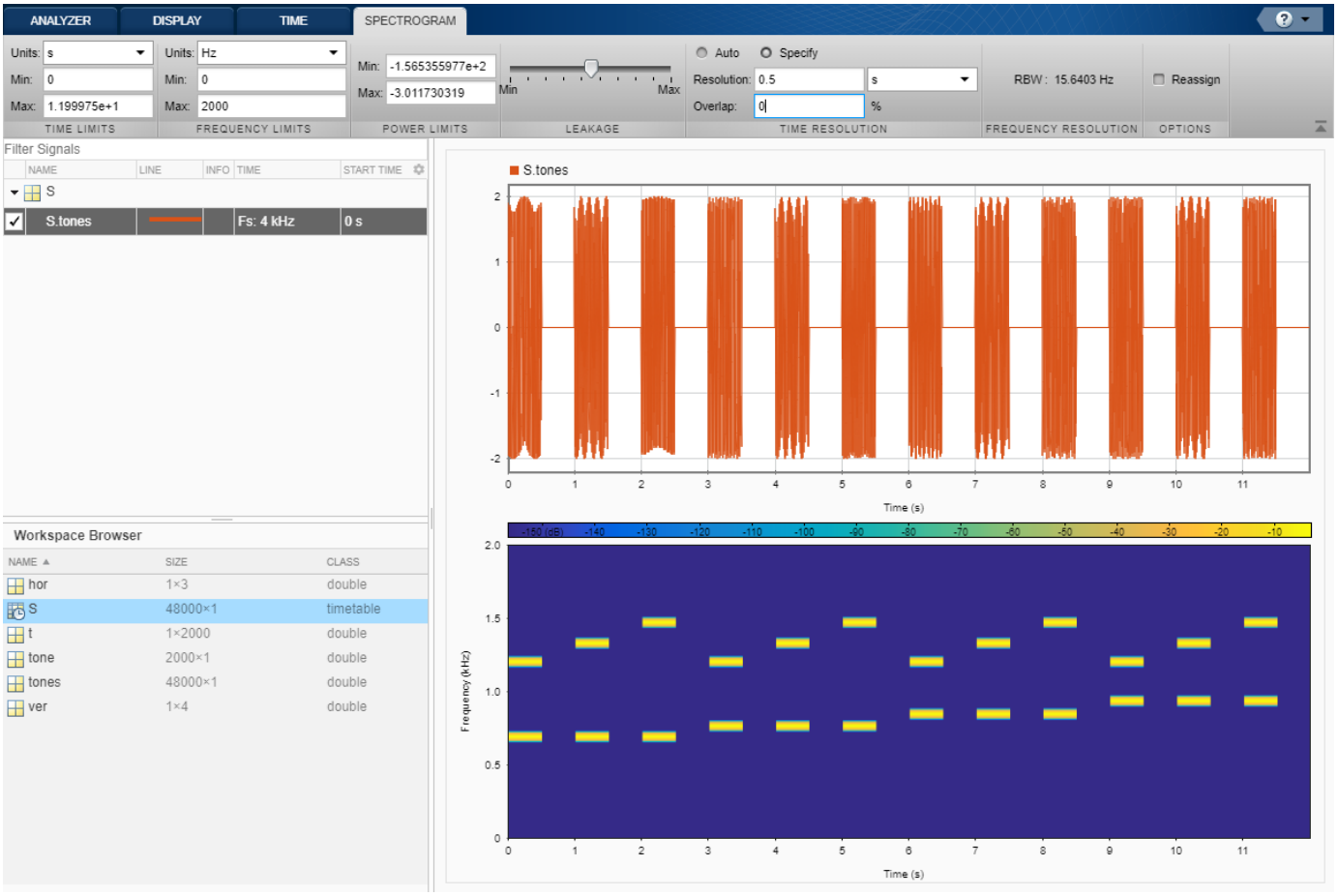
- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Spectrogram View of Dial Tone Signal

Create a signal, sampled at 4 kHz, that resembles dialing all the keys of a digital telephone. Save the signal as a MATLAB® timetable.

```
fs = 4e3;
t = 0:1/fs:0.5-1/fs;

ver = [697 770 852 941];
hor = [1209 1336 1477];

tones = [];

for k = 1:length(ver)
    for l = 1:length(hor)
        tone = sum(sin(2*pi*[ver(k);hor(l)].*t))';
        tones = [tones;tone;zeros(size(tone))];
    end
end

% To hear, type soundsc(tones,fs)

S = timetable(seconds(0:length(tones)-1)'/fs,tones);
```

Open **Signal Analyzer** and drag the timetable to a display. Click **Time-Frequency** to add a spectrogram view. On the **Spectrogram** tab, under **Time Resolution**, select **Specify**. Enter a time resolution of 0.5 second and zero overlap between adjoining segments.




The spectrogram view shows that each key is dialed for half a second, with half-second silent pauses between keys. The first tone has its frequency content concentrated around 697 Hz and 1209 Hz, corresponding to the digit `'1'` in the DTMF standard.

## See Also

### Apps

**Signal Analyzer**

### Functions

`pspectrum` | `spectrogram` | `timetable`

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Find Interference Using Persistence Spectrum” on page 21-51

- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Find Interference Using Persistence Spectrum

Visualize an interference narrowband signal embedded in a broadband signal.

Generate a chirp sampled at 1 kHz for 500 seconds. The frequency of the chirp increases from 180 Hz to 220 Hz during the measurement.

```
fs = 1000;
t = (0:1/fs:500)';
x = chirp(t,180,t(end),220) + 0.15*randn(size(t));
```

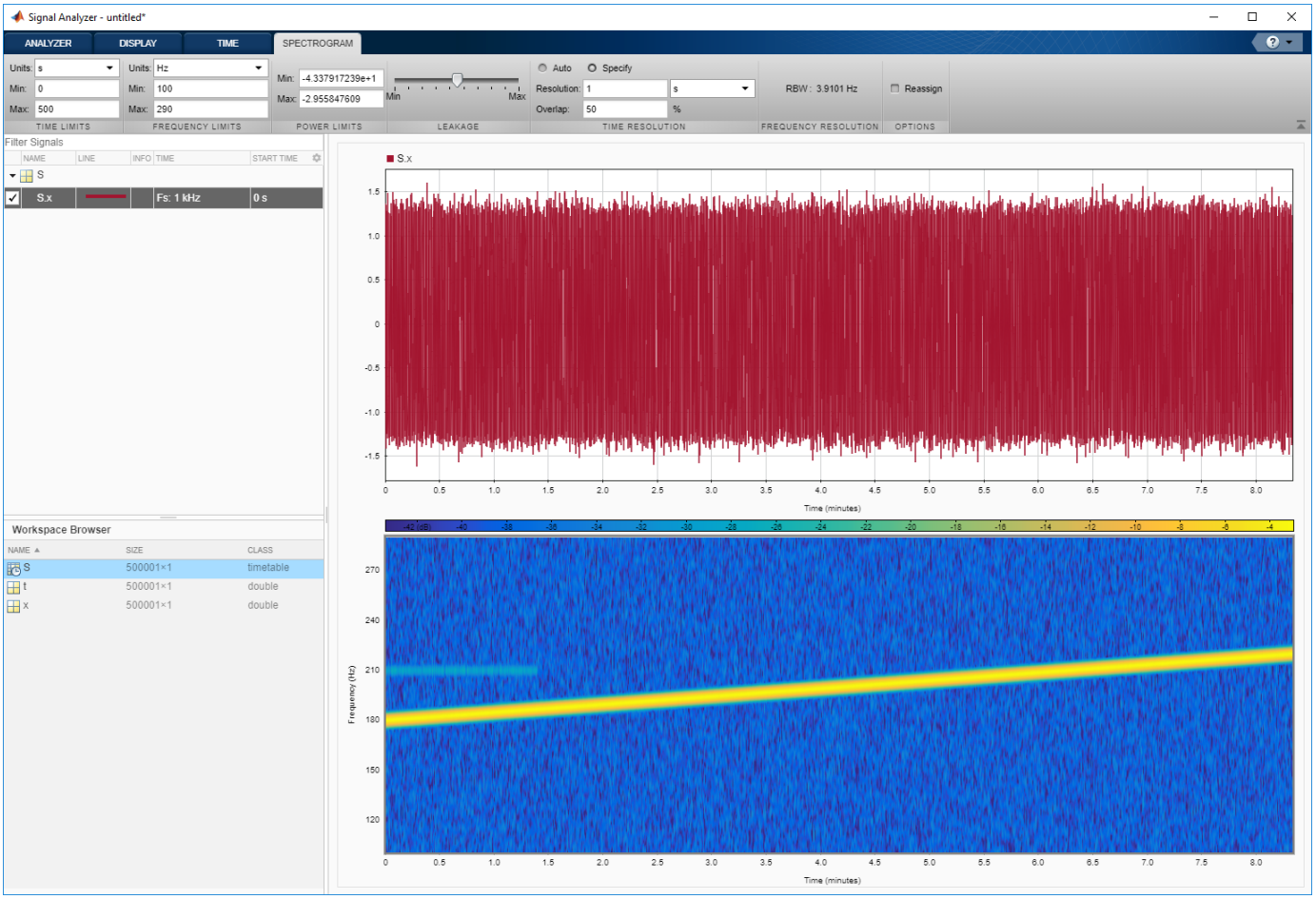
The signal also contains a 210 Hz sinusoid. The sinusoid has an amplitude of 0.05 and is present only for 1/6 of the total signal duration.

```
idx = floor(length(x)/6);
x(1:idx) = x(1:idx) + 0.05*cos(2*pi*t(1:idx)*210);
```

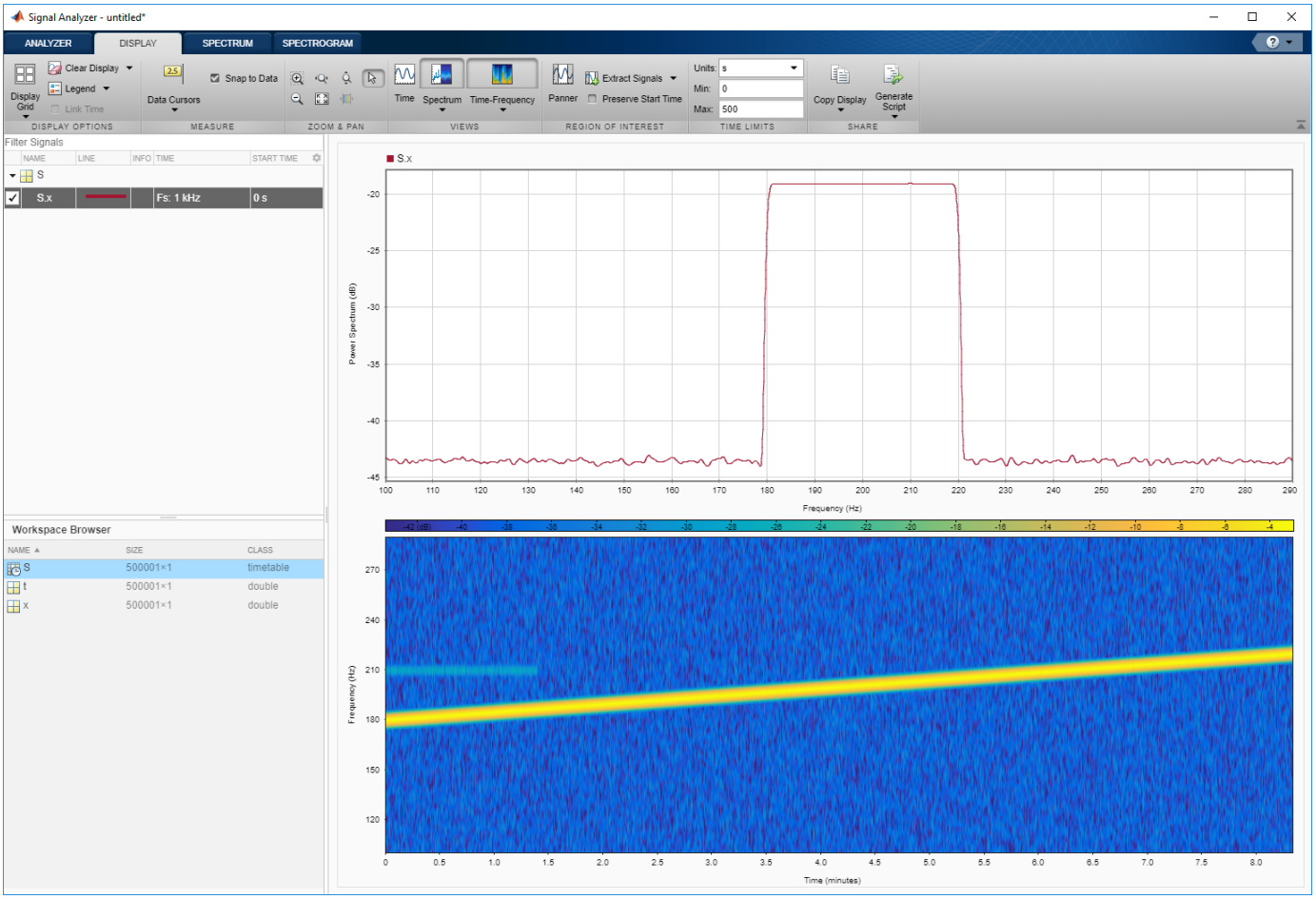
Save the signal as a MATLAB® timetable.

```
S = timetable(seconds(t),x);
```

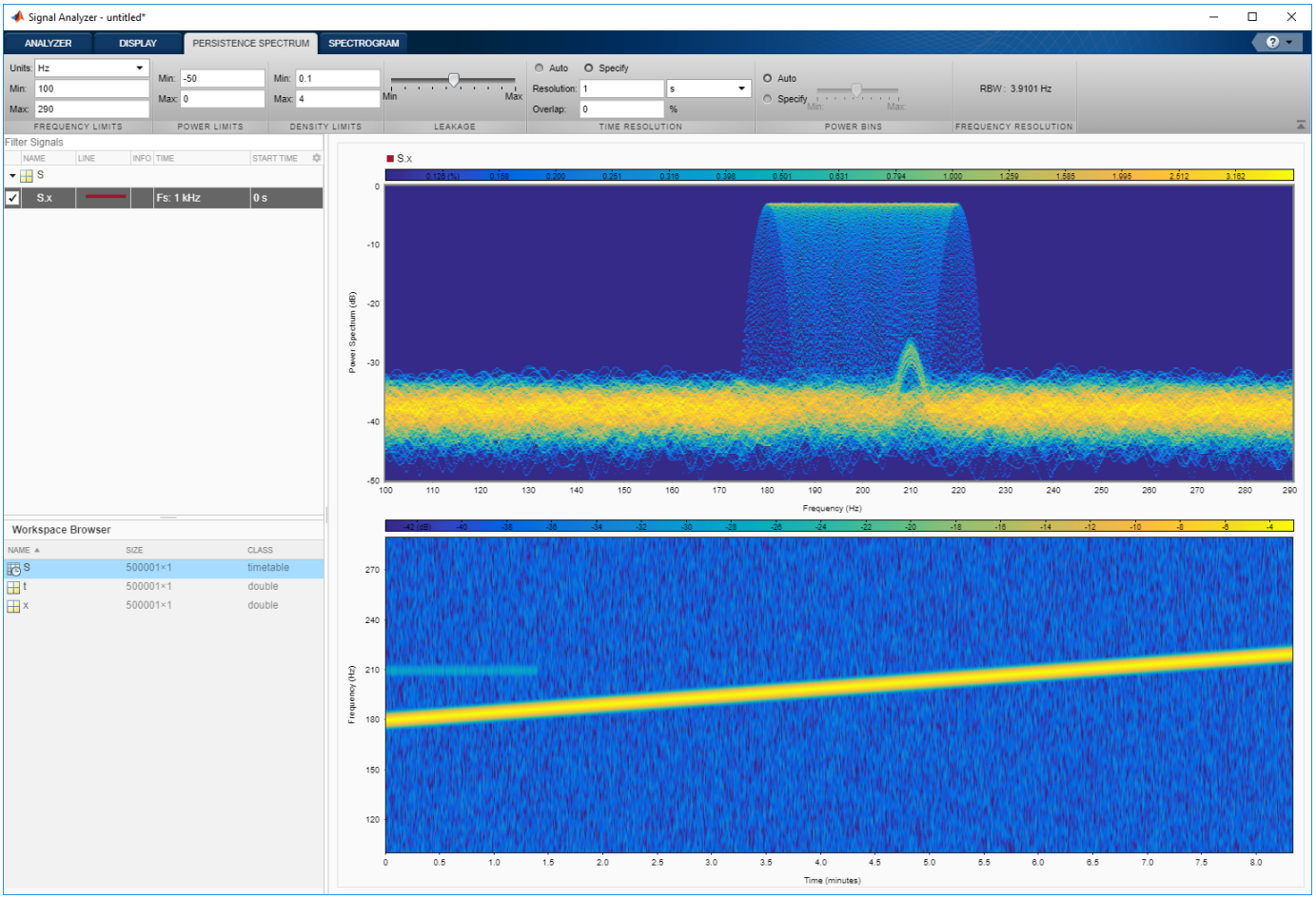
Open **Signal Analyzer** and drag the timetable from the Workspace browser to a display. Click the **Time-Frequency** button to add a spectrogram view. On the **Spectrogram** tab, under **Time Resolution**, select **Specify** and enter a time resolution of 1 second. Set the **Frequency Limits** to 100 Hz and 290 Hz. Both signal components are visible.




Go back to the **Display** tab. Click the **Time** button to remove the time view and click the **Spectrum** button to add a power spectrum view. The frequency range continues to be from 100 Hz to 290 Hz. The weak sinusoid is obscured by the chirp.




Click the **Spectrum ▼** button to change the `Spectrum` view to a `Persistence Spectrum` view. On the **Persistence Spectrum** tab, under **Time Resolution**, select **Specify** and enter a time resolution of 1 second. Specify zero overlap between adjoining segments. Set the **Power Limits** to –50 dB and 0 dB and the **Density Limits** to 0.1 and 4. Now both signal components are clearly visible.




On the **Display** tab, under **Share**, click **Generate Script ▼** and select `Persistence Spectrum Script`. The script appears in the MATLAB Editor.

```
% Compute persistence spectrum

% Generated by MATLAB(R) 9.7 and Signal Processing Toolbox 8.2.
% Generated on: 26-Dec-2018 16:07:45

% Parameters
timeLimits = seconds([0 500]); % seconds
frequencyLimits = [100 290]; % Hz
timeResolution = 1; % seconds
overlapPercent = 0;

%%
% Index into signal time region of interest
S_x_ROI = S(:,'x');
S_x_ROI = S_x_ROI(timerange(timeLimits(1),timeLimits(2),'closed'),1);

% Compute spectral estimate
% Run the function call below without output arguments to plot the results
[P,F,PWR] = pspectrum(S_x_ROI, ...
    'persistence', ...
    'FrequencyLimits',frequencyLimits, ...
```

```
    'TimeResolution',timeResolution, ...
    'OverlapPercent',overlapPercent);
```

## See Also

**Apps**
**Signal Analyzer**

**Functions**
`pspectrum` | `timetable`

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

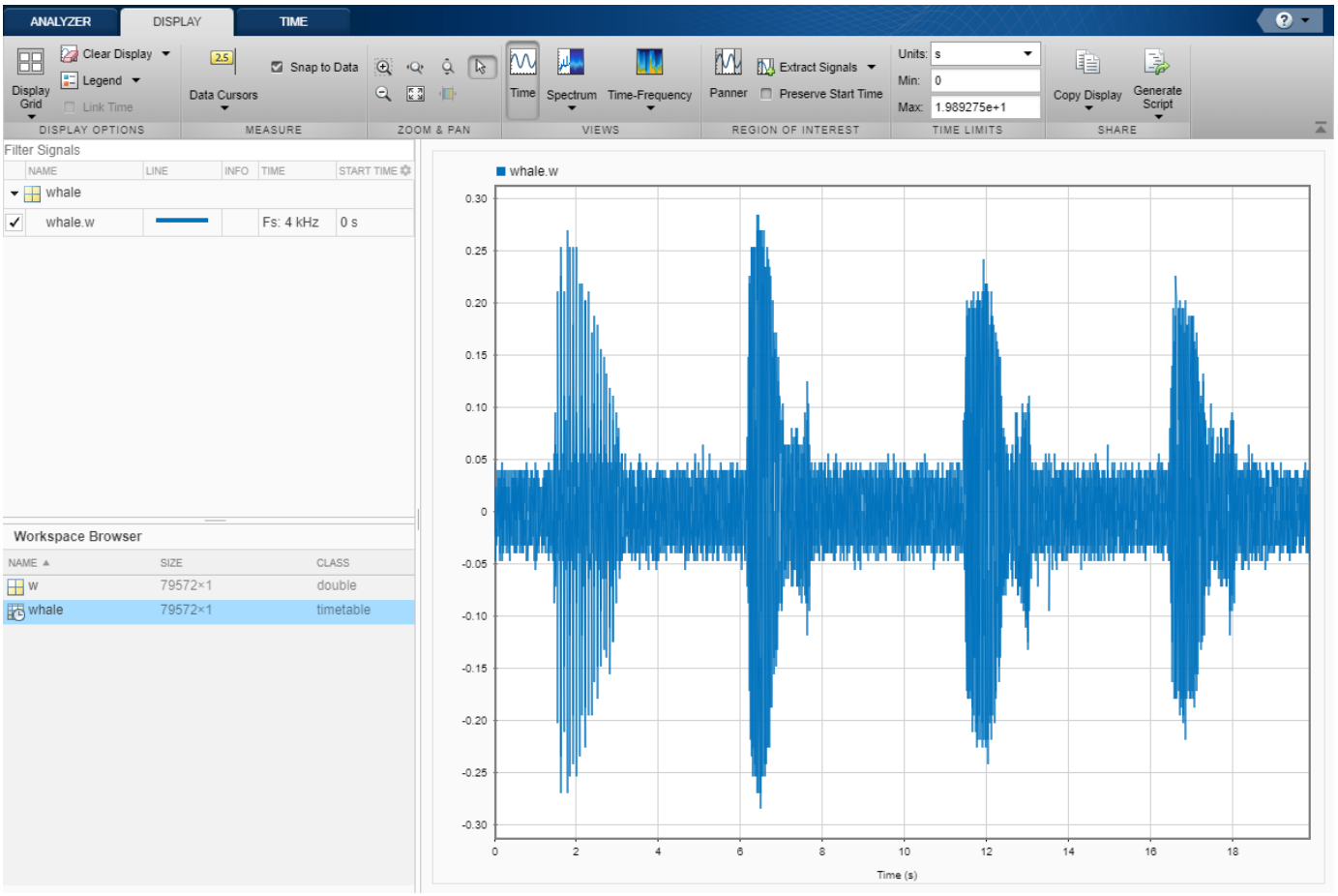
- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Extract Regions of Interest from Whale Song

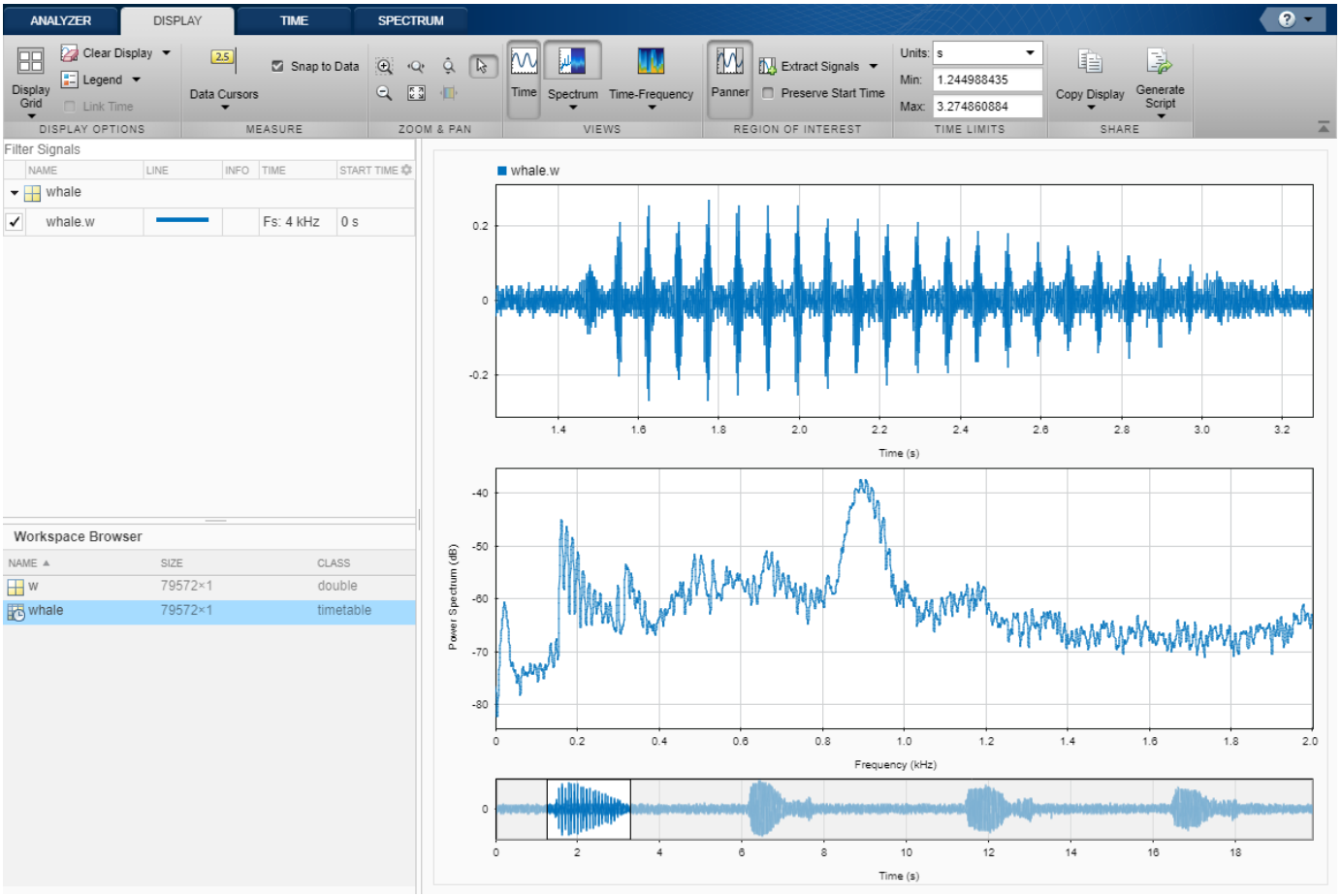
Load a file that contains audio data from a Pacific blue whale, sampled at 4 kHz. The file is from the library of animal vocalizations maintained by the Cornell University Bioacoustics Research Program. The time scale in the data is compressed by a factor of 10 to raise the pitch and make the calls more audible. Convert the signal to a MATLAB® timetable.

```
whaleFile = fullfile(matlabroot,'examples','matlab','data','bluewhale.au');
[w,fs] = audioread(whaleFile);

whale = timetable(seconds((0:length(w)-1)'/fs),w);

% To hear, type soundsc(w,fs)
```

Open **Signal Analyzer** and drag the timetable to a display. Four features stand out from the noise. The first is known as a *trill*, and the other three are known as *moans*.

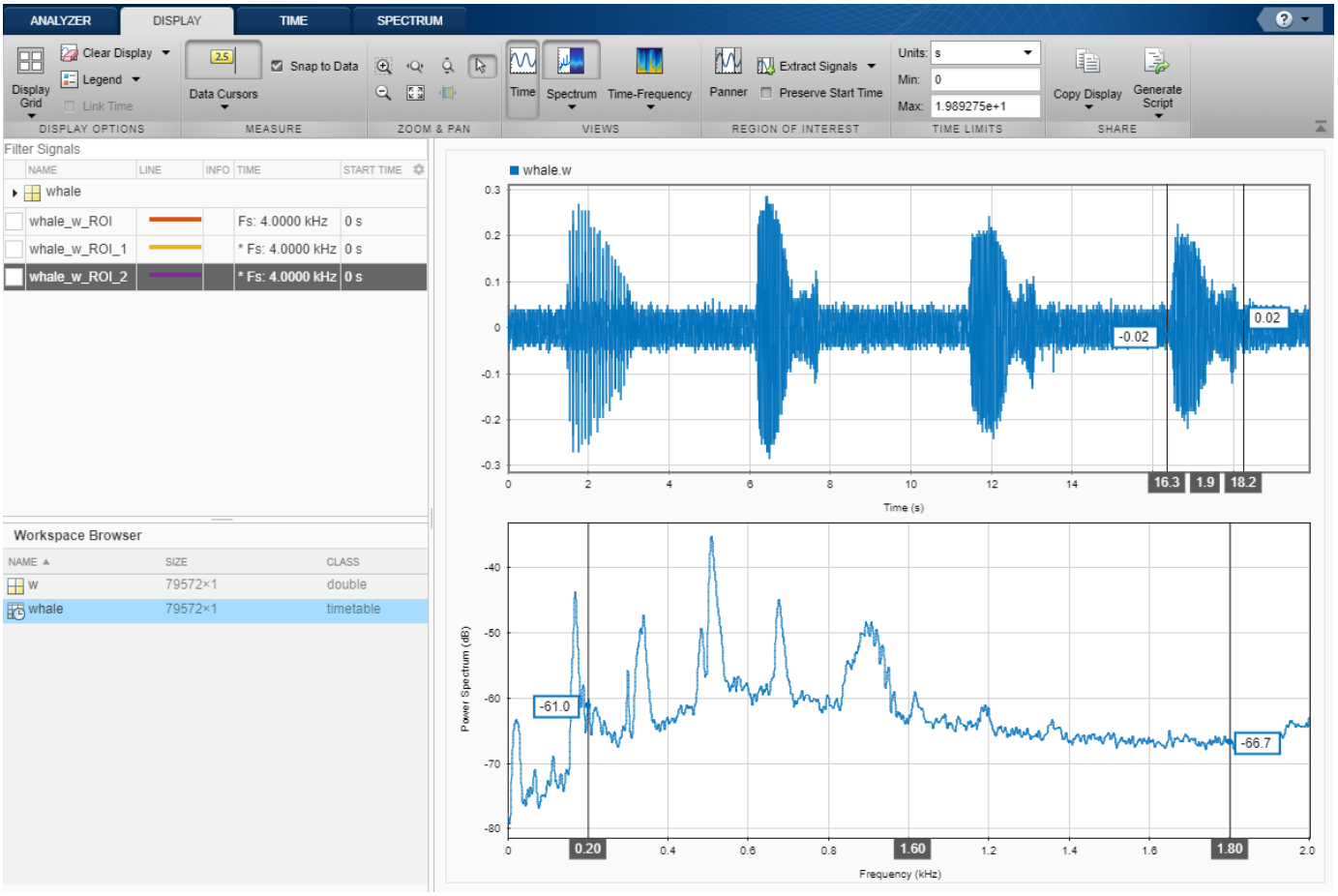



On the **Display** tab, click **Spectrum** to open a spectrum view and click **Panner** to activate the panner. Use the panner to create a zoom window with a width of about 2 seconds. Drag the zoom window so that it is centered on the trill. The spectrum shows a noticeable peak at around 900 Hz.

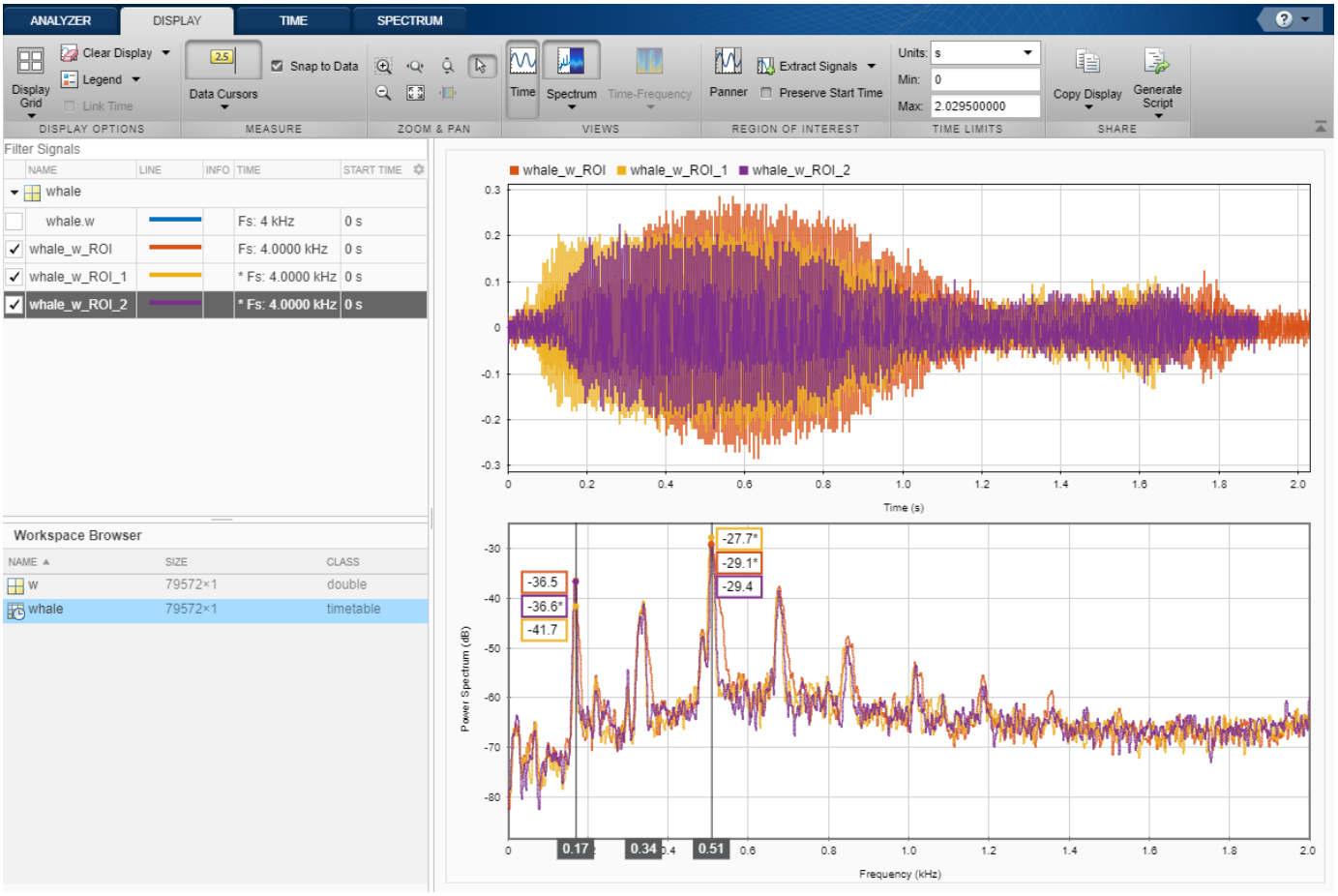



Extract the three moans to compare their spectra:

1. Center the panner zoom window on the first moan. The spectrum has eight clearly defined peaks, located very close to multiples of 170 Hz. Click **Extract Signals** ▼ and select `Between Time Limits`.
2. Click **Panner** to hide the panner. Press the space bar to see the full signal. Click **Zoom in X** and zoom in on a 2-second interval of the time view centered on the second moan. The spectrum again has peaks at multiples of 170 Hz. Click **Extract Signals** ▼ and select `Between Time Limits`.
3. Press the space bar to see the full signal. Click **Data Cursors** ▼ and select `Two`. Place the time-domain cursors in a 2-second interval around the third moan. Again, there are peaks at multiples of 170 Hz. Click **Extract Signals** ▼ and select `Between Time Cursors`.




Remove the original signal from the display by clearing the check box next to its name in the Signal table. Display the three regions of interest you just extracted. Their spectra lie approximately on top of each other. Move the frequency-domain cursors to the locations of the first and third spectral peaks. Asterisks in cursor labels indicate interpolated signal values.




## See Also

### Apps

**Signal Analyzer**

### Functions

`pspectrum` | `timetable`

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70

- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96

## More About

- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Modulation and Demodulation Using Complex Envelope

This example simulates the different steps of a basic communication process. Communication systems work by modulating chunks of information into a higher *carrier frequency*, transmitting the modulated signals through a noisy physical channel, receiving the noisy waveforms, and demodulating the received signals to reconstruct the initial information.

All the information carried in a real-valued signal $s(t)$ can be represented by a corresponding lowpass *complex envelope*:

$$s(t) = \mathrm{Re}\left\{g(t)e^{j2\pi f_c t}\right\} = i(t)\cos 2\pi f_c t + q(t)\sin 2\pi f_c t.$$

In this equation:

- $f_c$ is the carrier frequency.
- Re represents the real part of a complex-valued quantity.
- $g(t) = i(t) + jq(t)$is the complex envelope of $s(t)$.
- $i(t)$ is the *inphase* component of the complex envelope.
- $q(t)$ is the *quadrature* component of the complex envelope.

The complex envelope is modulated to the carrier frequency and sent through the channel. At the receiver, the noisy waveform is demodulated using the carrier frequency.

The phase variation due to the carrier frequency is predictable and thus does not convey any information. The complex envelope does not include the phase variation and can be sampled at a lower rate.

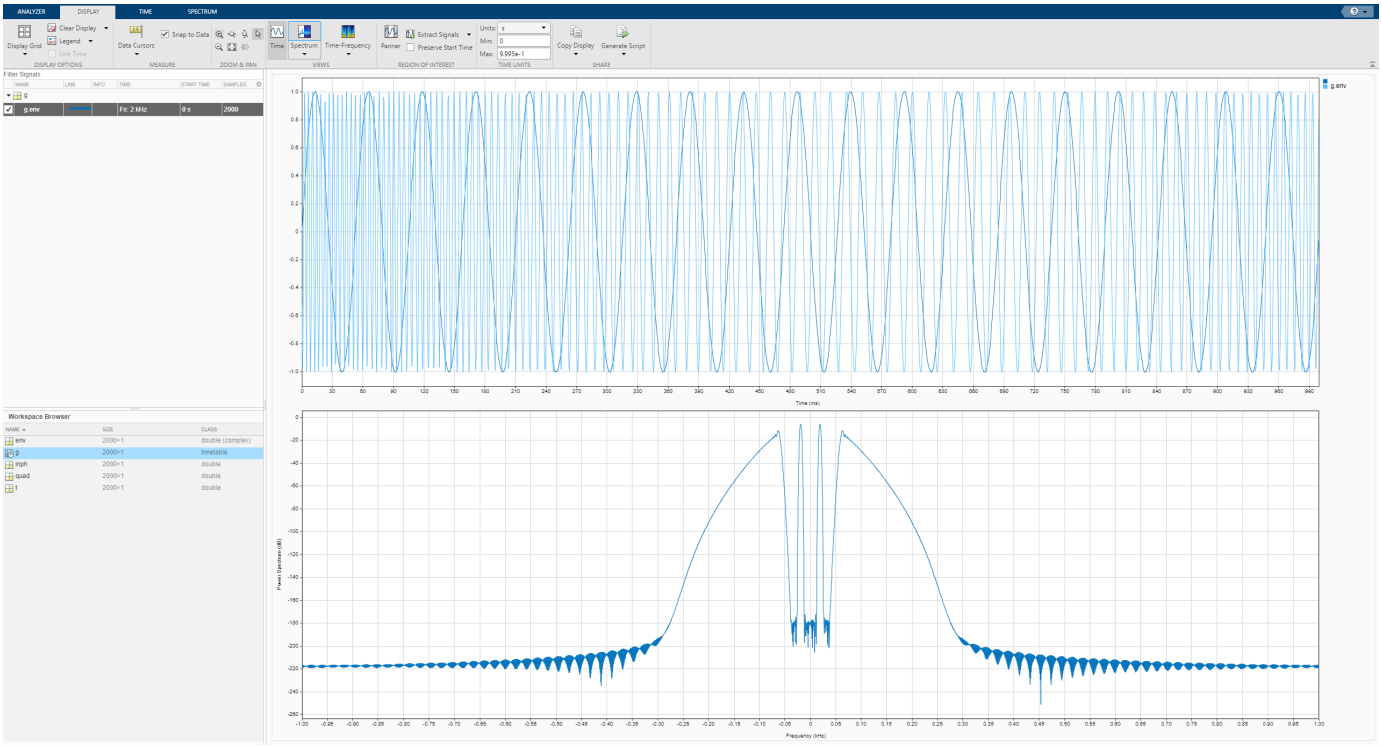
Generate a signal whose complex envelope consists of a sinusoid and a chirp. The inphase component is a sinusoid with a frequency of 19 Hz. The quadrature component is a quadratic chirp whose frequency ranges from 61 Hz to 603 Hz. The signal is sampled at 2 kHz for 1 second.

```
fs = 2e3;
t = (0:1/fs:1-1/fs)';
inph = sin(2*pi*19*t);
quad = chirp(t-0.6,61,t(end),603,'quadratic');
```

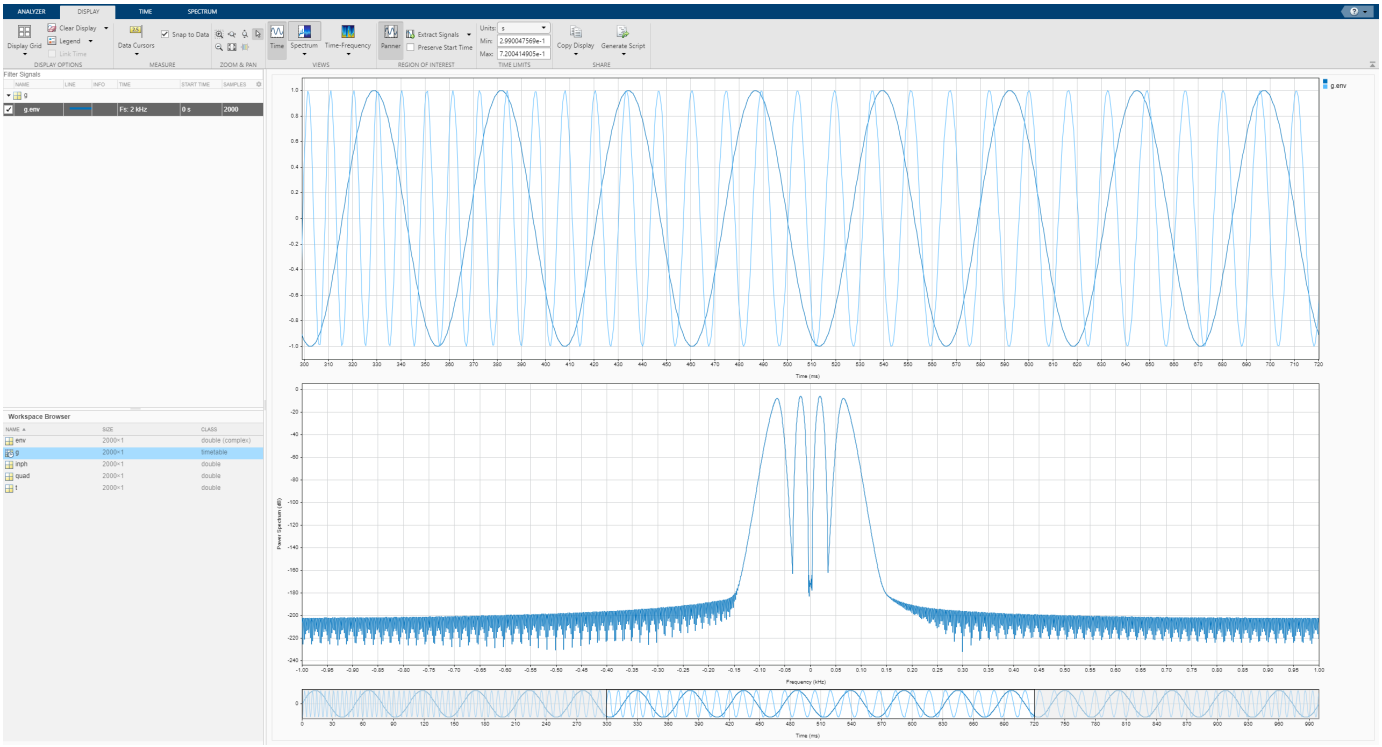
Compute the complex envelope and store it as a MATLAB® timetable of sample rate `fs`.

```
env = inph + 1j*quad;
g = timetable(env,'SampleRate',fs);
```

Open **Signal Analyzer** and drag the complex envelope from the Workspace browser to the Signal table. The display shows the inphase and quadrature components of the envelope as lines of the same hue and saturation, but different luminosity. The first line color represents the inphase component and the second line color represents the quadrature component. Click **Spectrum ▼** on the **Display** tab and select `Spectrum`. The app displays a set of axes with the signal spectrum. The complex envelope has a two-sided spectrum, displayed as a line of the same color of the inphase component of the complex envelope.



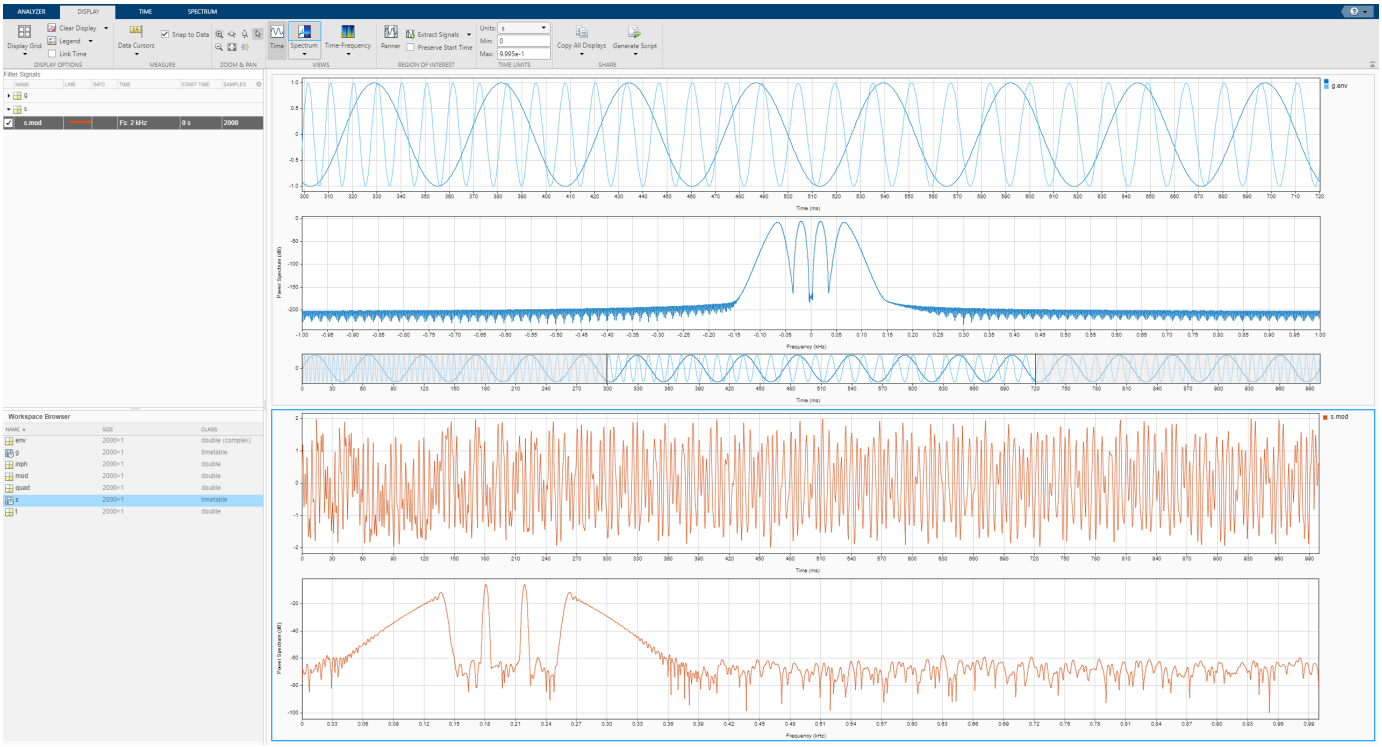
On the **Display** tab, click **Panner** to activate the panner. Use the panner to create a zoom window between 300 ms and 720 ms. Drag the zoom window so that it is centered at 0 Hz. The spectrum has an impulse at 0.19 kHz and a wider tapering profile at higher frequencies. The negative-frequency region of the spectrum is a mirror image of the positive-frequency region.



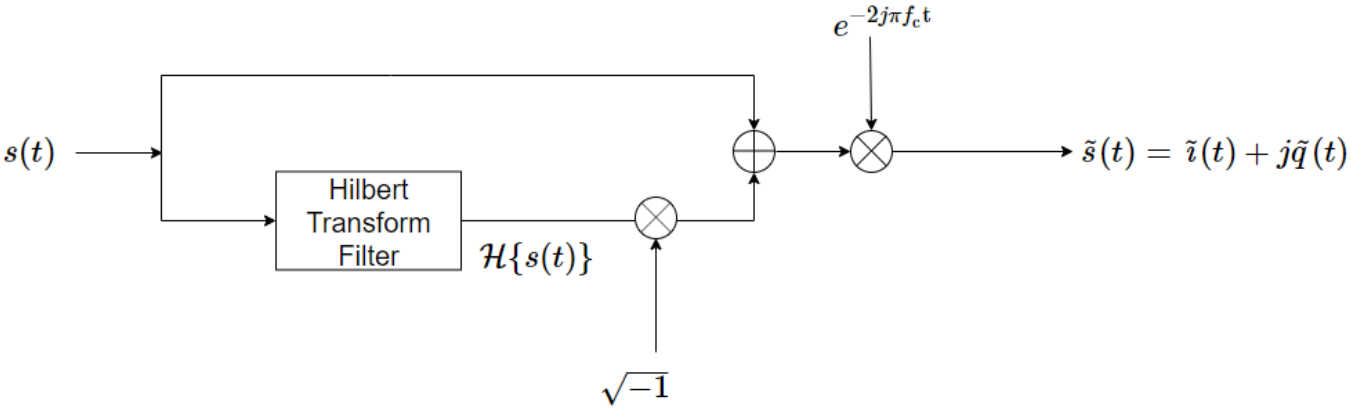
Modulate the signal using a carrier frequency of 200 Hz. Multiply by $\sqrt{2}$ so that the power of the modulated signal equals the power of the original signal. Add white Gaussian noise such that the signal-to-noise ratio is 40 dB.

```
fc = 200;
mod = sqrt(2)*real(env.*exp(2j*pi*fc*t));

SNR = 40;
mod = mod + randn(size(mod))*std(mod)/db2mag(SNR);
s = timetable(mod,'SampleRate',fs);
```

Click **Display Grid ▼** to add a second display. Drag the modulated signal to the Signal table and enter the time information. The modulation has moved the spectrum to positive frequencies centered on the carrier frequency.

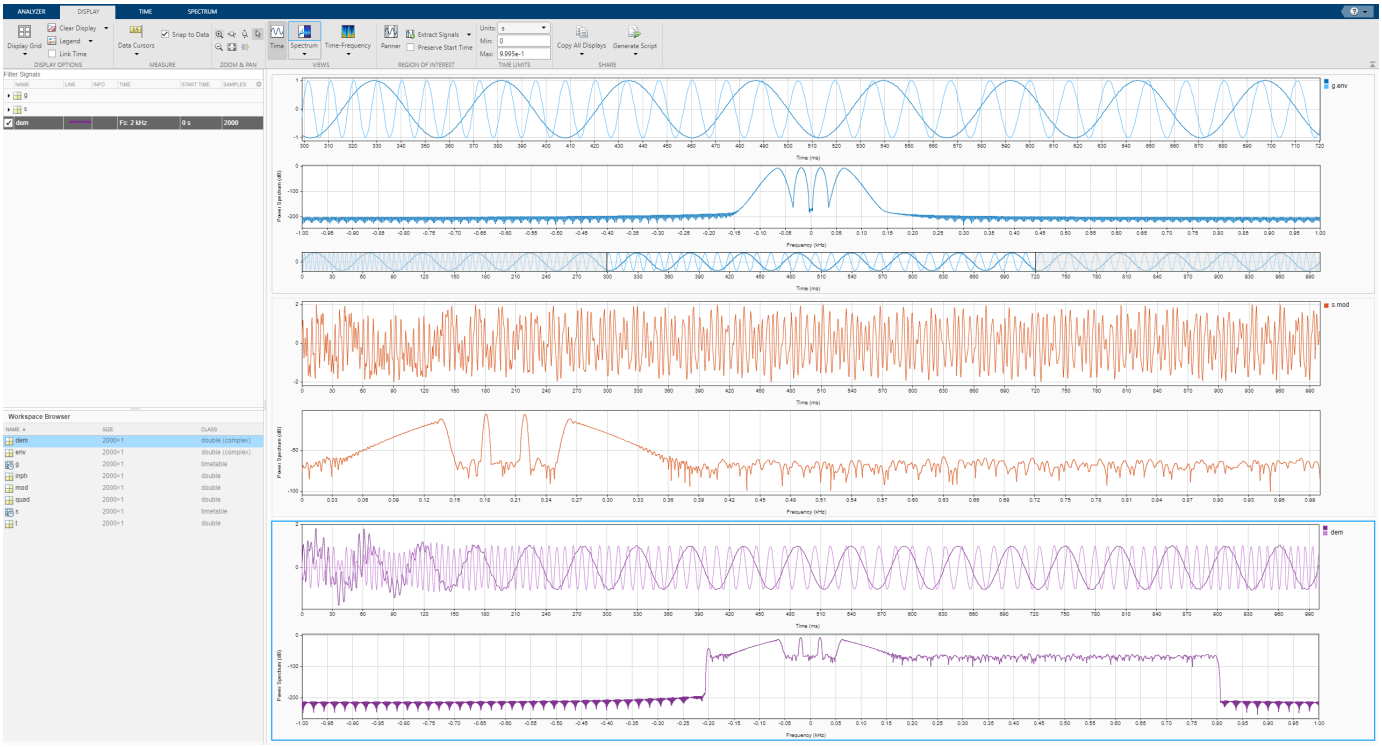


Calculate the analytic signal and demodulate the signal by multiplying the analytic signal with a complex-valued negative exponential of frequency 200 Hz.

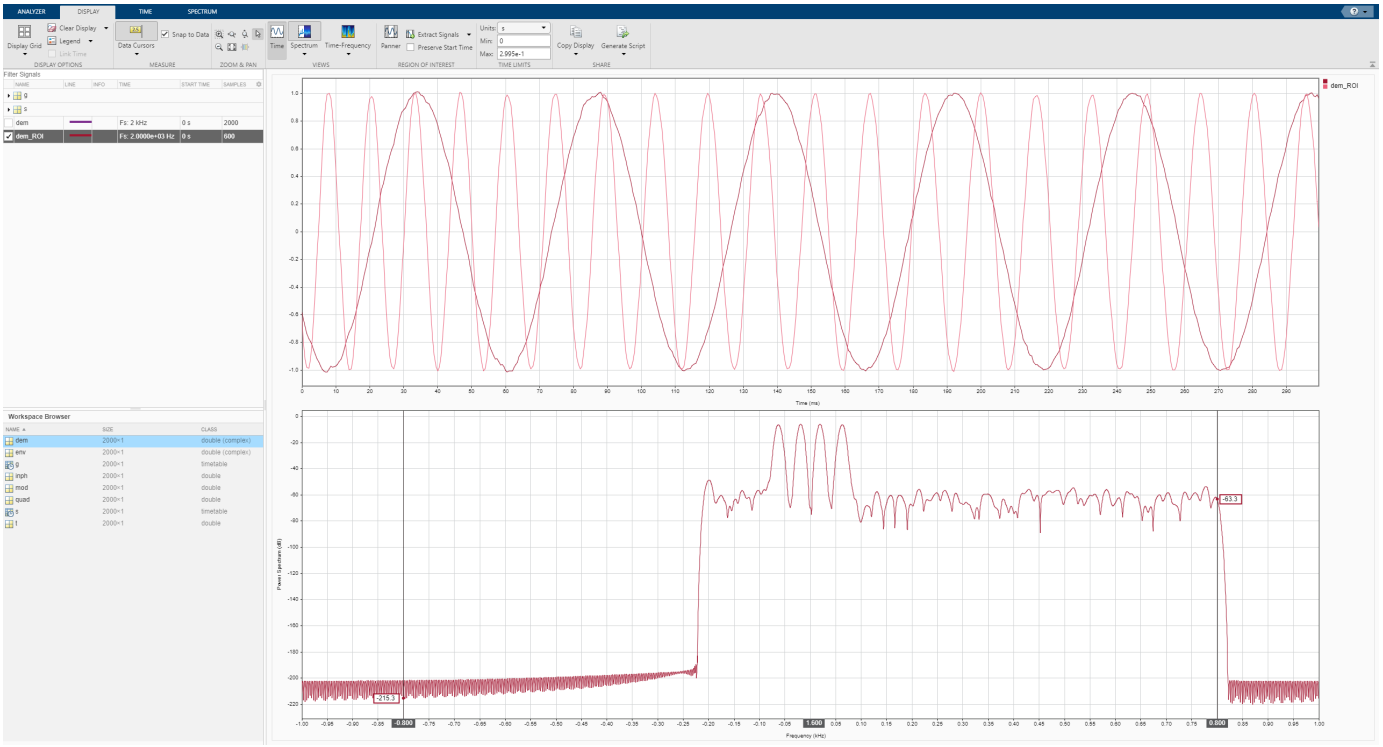



```
dem = hilbert(mod).*exp(-2j*pi*fc*t)/sqrt(2);
```

Click **Display Grid ▼** to create a three-by-one grid of displays. Drag the demodulated signal to the Signal table. Add time information to the complex envelope by clicking **Time Values** in the **Analyzer** tab. The two-sided spectrum shows the recovered inphase and quadrature components of the baseband signal.



Click **Display Grid ▼** to create a one-by-one grid of displays and plot the demodulated signal. Click **Data Cursors ▼** and select `Two`. Place the time-domain cursors at 300 ms and 900 ms, so they enclose the spectral peaks. Click **Extract Signals ▼** and select `Between Time Cursors`. Check the **Preserve Start Time** box. Clear the display and plot the extracted signal. The app extracts both inphase and quadrature components of the demodulated signal in the region of interest. Select the extracted signal by clicking its **Name** column in the Signal table. On the **Analyzer** tab, click **Export** and save the signal to a MAT-file called `dem_ROI.mat`.

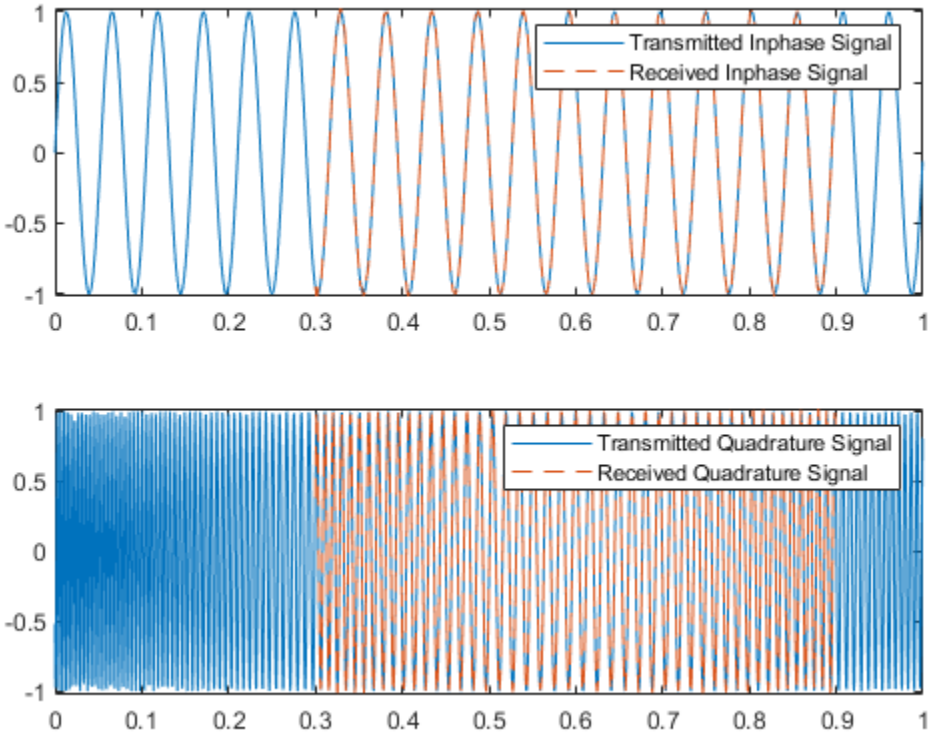


Load the `dem_ROI` file to the workspace. Compute the demodulated inphase and quadrature components by taking the real and imaginary parts of the extracted signal. Store the time information of the extracted signal in a time variable `t_dem`.

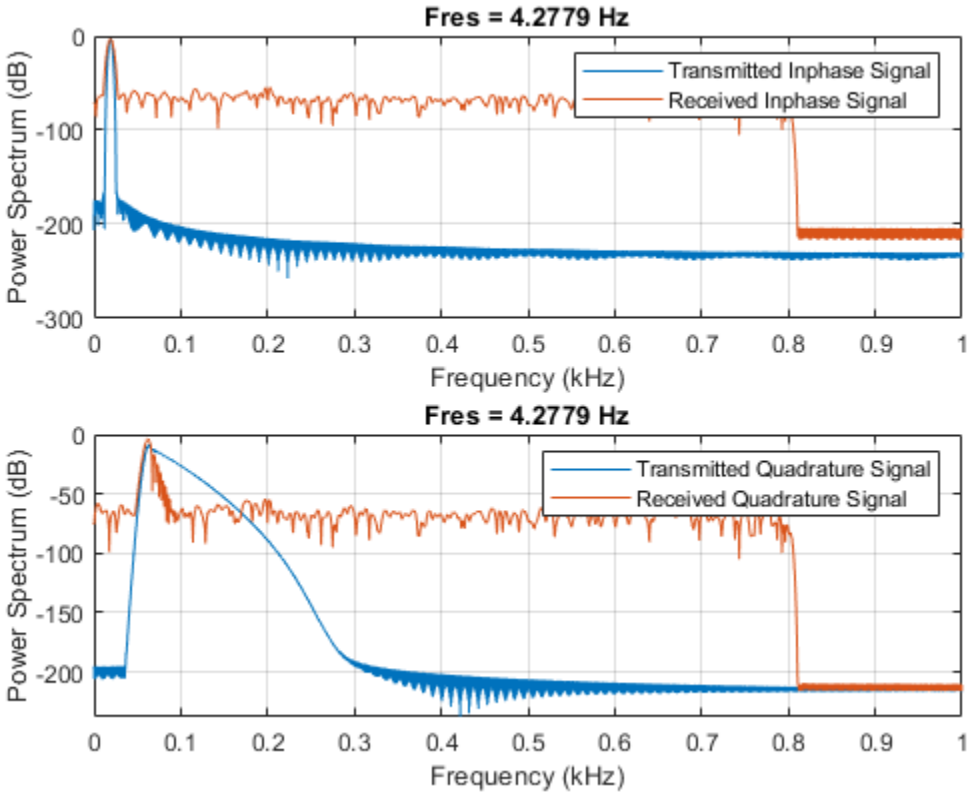
```
load dem_ROI
inph_dem = real(dem_ROI);
quad_dem = imag(dem_ROI);
t_dem = 0.3+(0:length(dem_ROI)-1)/fs;
```

Compare the transmitted waveforms and the extracted regions of interest. Also compare their spectra.

```
subplot(2,1,1)
plot(t,inph,t_dem,inph_dem,'--')
legend('Transmitted Inphase Signal','Received Inphase Signal')

subplot(2,1,2)
plot(t,quad,t_dem,quad_dem,'--')
legend('Transmitted Quadrature Signal','Received Quadrature Signal')
```




```
figure
subplot(2,1,1)
pspectrum(inph,fs)
hold on
pspectrum(inph_dem,fs)
legend('Transmitted Inphase Signal','Received Inphase Signal')
hold off

subplot(2,1,2)
pspectrum(quad,fs)
hold on
pspectrum(quad_dem,fs)
legend('Transmitted Quadrature Signal','Received Quadrature Signal')
hold off
```




## See Also

**Apps**
**Signal Analyzer**

**Functions**

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91

- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

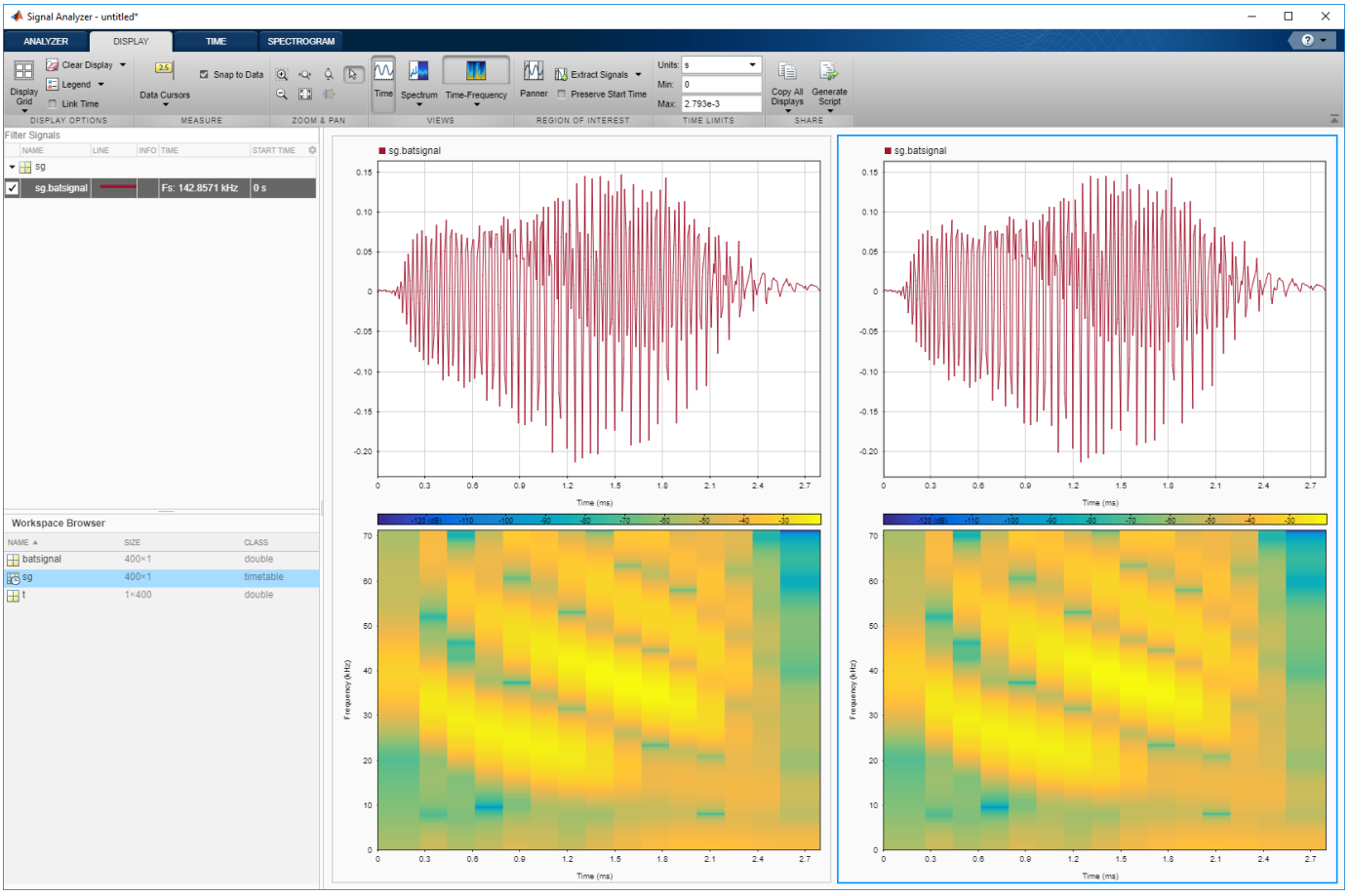
# Find and Track Ridges Using Reassigned Spectrogram

Load a datafile containing an echolocation pulse emitted by a big brown bat (*Eptesicus fuscus*) and measured with a sampling interval of 7 microseconds. Create a MATLAB® timetable using the signal and the time information.

```
load batsignal

t = (0:length(batsignal)-1)*DT;
sg = timetable(seconds(t)',batsignal);
```

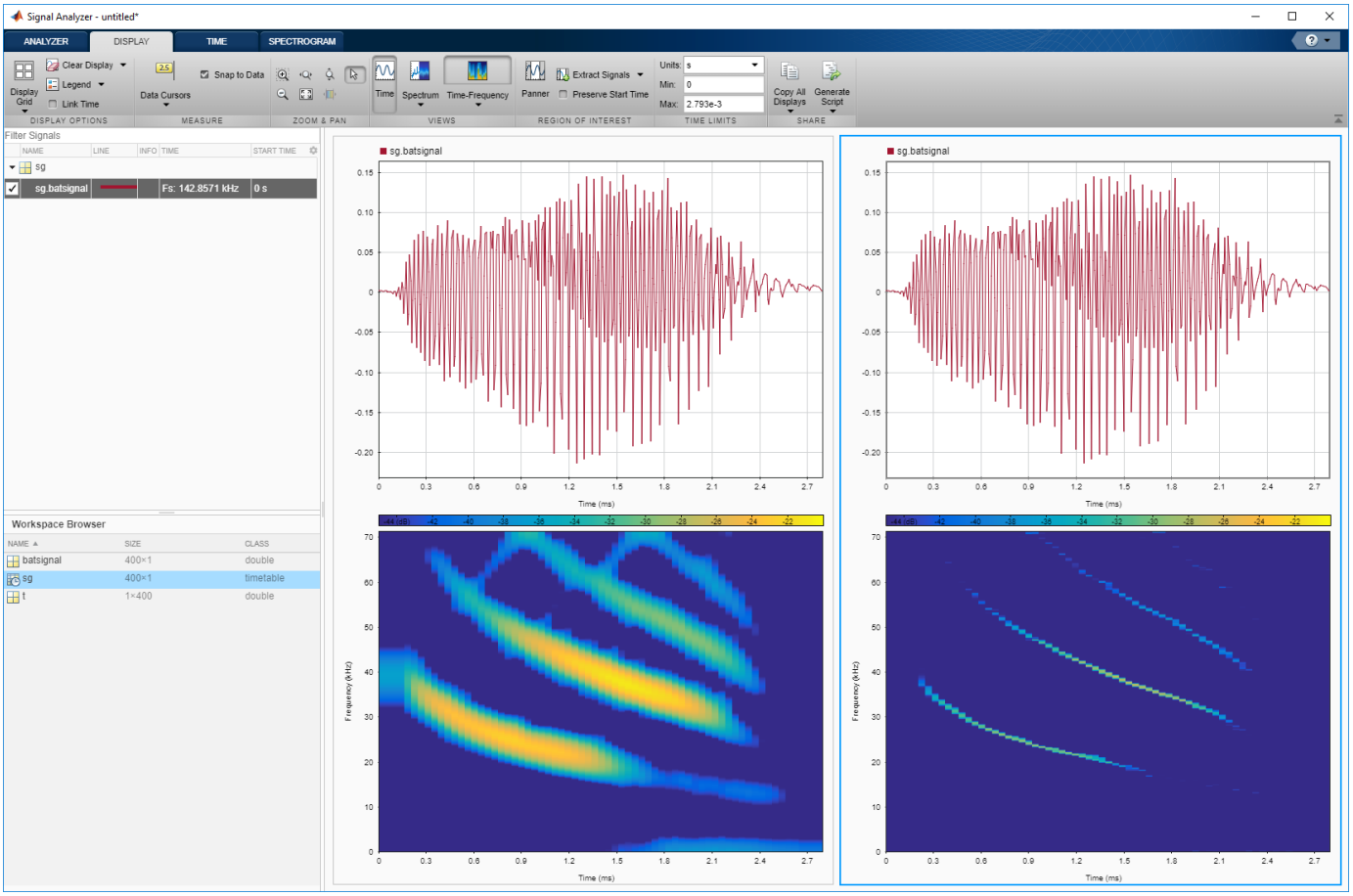
Open **Signal Analyzer** and drag the timetable from the Workspace browser to the Signal table. Click **Display Grid ▼** to create two side-by-side displays. Select each display and click the **Time-Frequency** button to add a spectrogram view.

Drag the timetable to both displays.




Select the **Spectrogram** tab. On the display at right, check **Reassign**. For each display:

- Set the time resolution to 280 microseconds and specify 85% overlap between adjoining segments.
- Use the **Leakage** slider to increase the leakage until the RBW is about 4.5 kHz.
- Set the power limits to –45 dB and –20 dB.



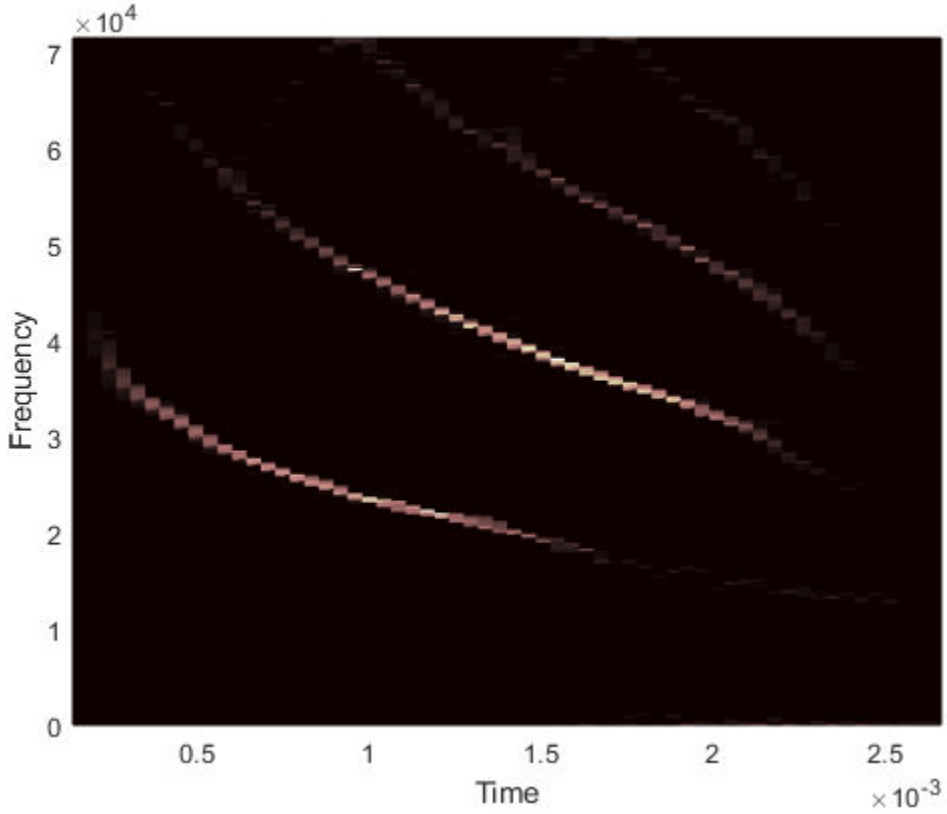

The reassigned spectrogram clearly shows three time-frequency ridges. To track the ridges, select the display at right. On the **Display** tab, click **Generate Script** and select `Spectrogram Script`. The script appears in the Editor.

```
% Compute spectrogram

% Generated by MATLAB(R) 9.7 and Signal Processing Toolbox 8.2.
% Generated on: 26-Dec-2018 17:21:44

% Parameters
timeLimits = seconds([0 0.002793]); % seconds
frequencyLimits = [0 71428.57]; % Hz
leakage = 0.9;
timeResolution = 0.00028; % seconds
overlapPercent = 85;
reassignFlag = true;

%%
% Index into signal time region of interest
sg_batsignal_ROI = sg(:,'batsignal');
sg_batsignal_ROI = sg_batsignal_ROI(timerange(timeLimits(1),timeLimits(2),'closed'),1);

% Compute spectral estimate
% Run the function call below without output arguments to plot the results
[P,F,T] = pspectrum(sg_batsignal_ROI, ...
```

```
    'spectrogram', ...
    'FrequencyLimits',frequencyLimits, ...
    'Leakage',leakage, ...
    'TimeResolution',timeResolution, ...
    'OverlapPercent',overlapPercent, ...
    'Reassign',reassignFlag);
```

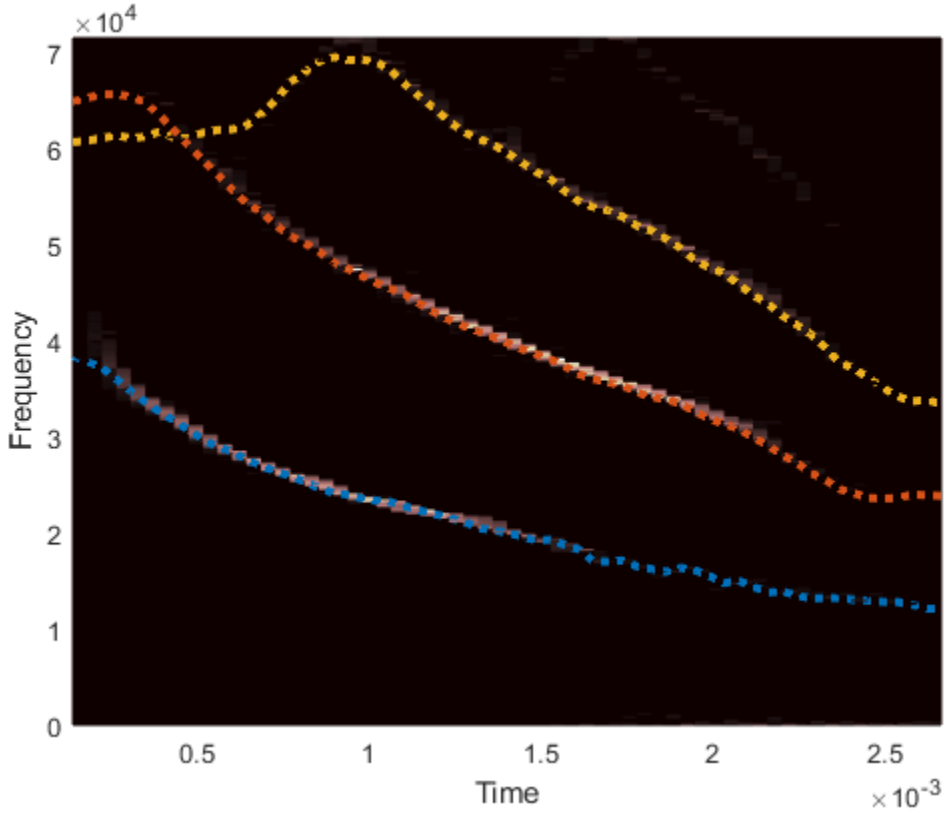
Run the script. Plot the reassigned spectrogram.

```
mesh(seconds(T),F,P)
xlabel('Time')
ylabel('Frequency')
axis tight
view(2)
colormap pink
```




Use the `tfridge` function to track the ridges.

```
[fridge,~,lridge] = tfridge(P,F,0.01,'NumRidges',3,'NumFrequencyBins',10);

hold on
plot3(seconds(T),fridge,P(lridge),':','linewidth',3)
hold off
```




Thanks to Curtis Condon, Ken White, and Al Feng of the Beckman Center at the University of Illinois for the bat data and permission to use it in this example.

## See Also

**Apps**
**Signal Analyzer**

**Functions**
`pspectrum` | `tfridge` | `timetable`

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Scalogram of Hyperbolic Chirp” on page 21-75

- “Extract Voices from Music Signal” on page 21-78
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Scalogram of Hyperbolic Chirp

Use **Signal Analyzer** to display a hyperbolic chirp and its scalogram. This example requires a Wavelet Toolbox license.

Generate a signal sampled at 1 kHz for 1 second. The signal consists of two hyperbolic chirps. One of the chirps is active between 0.1 second and 0.68 second. The other chirp has twice the amplitude and one-third of the instantaneous frequency of the first chirp. The second chirp is active between 0.1 second and 0.75 second.

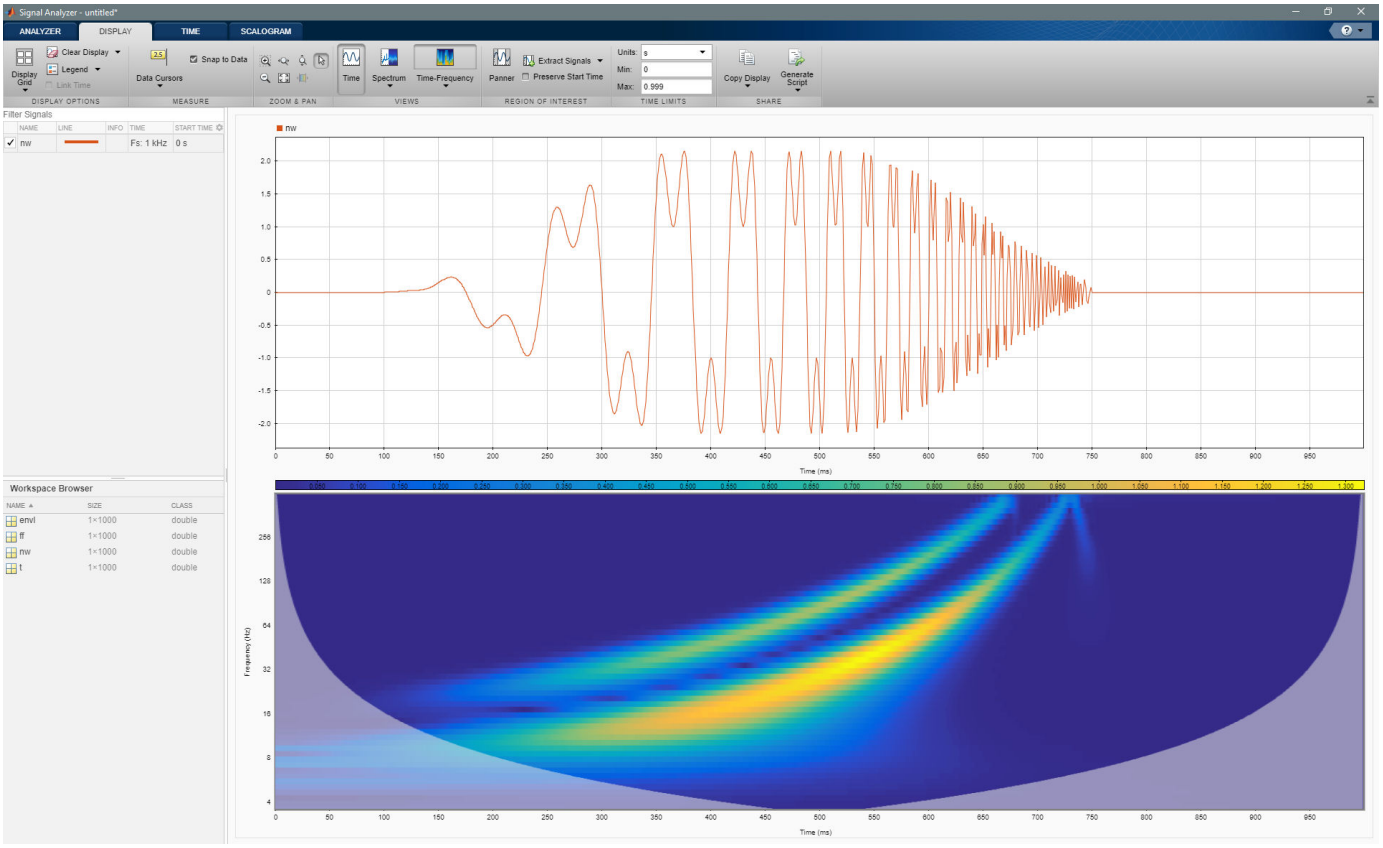
```
n = 1000;
t = (0:n-1)/n;

ff = [1 2]*(sin(pi*[15 5]'./(t-0.8001)).*(t>0.1 & t<[0.68 0.75]'));
```

The chirp is modulated by an envelope that starts at 0.1 second and is nonzero for 0.7 second.

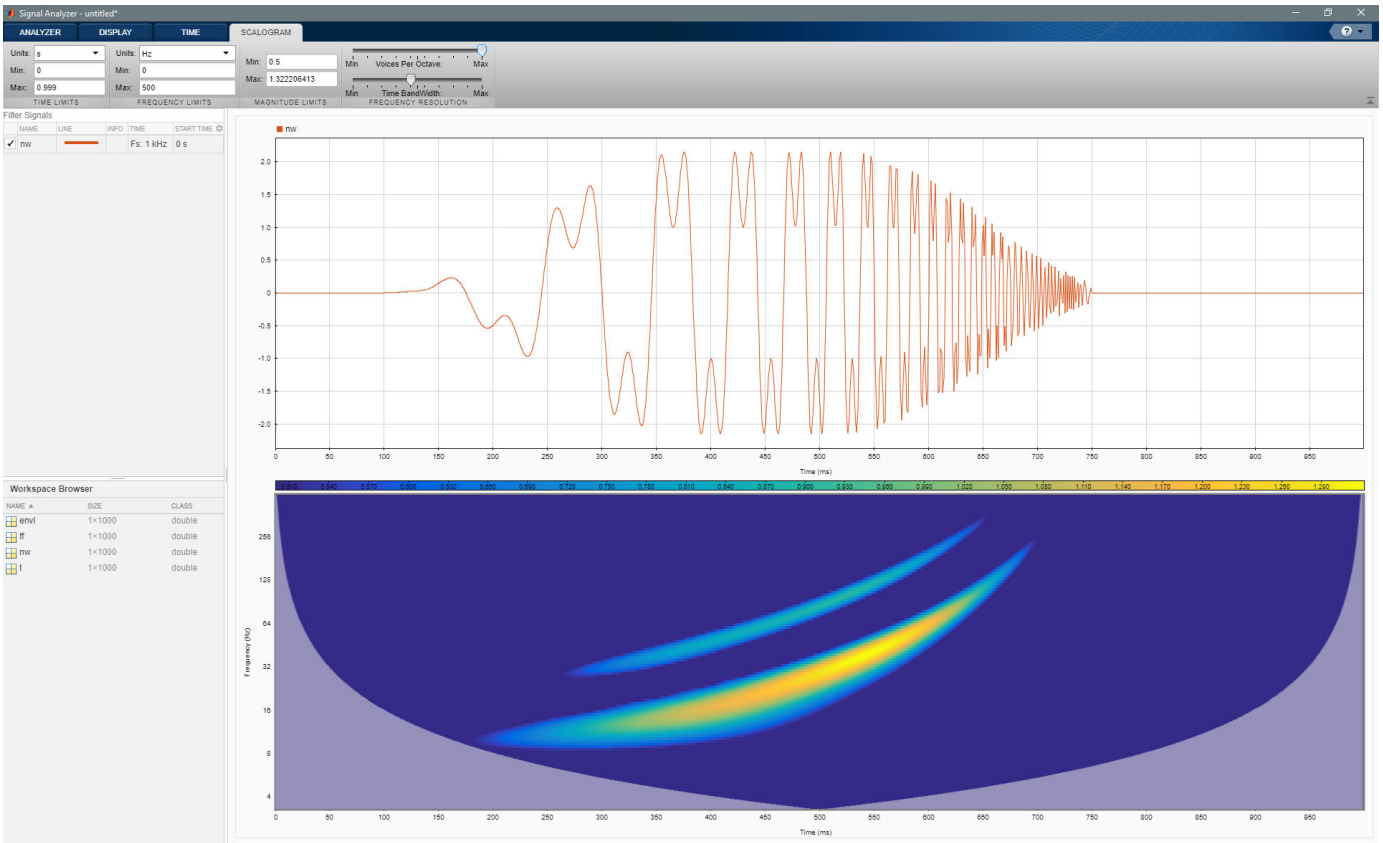
```
envl = zeros(size(t));
M = 0.7*n;
envl(n/10+(1:M)) = tukeywin(M,0.8);

nw = ff.*envl;
```

Display the signal in **Signal Analyzer**. Click **Time-Frequency** ▼ on the **Display** tab and select `Scalogram`.

```
signalAnalyzer(nw,'TimeValues',t)
```



On the **Scalogram** tab, set the **Voices Per Octave** to the maximum value. Set the lower magnitude limit to 0.5.




## See Also

### Apps

**Signal Analyzer**

### Functions

`cwt` | `cwtfilterbank` | `pspectrum`

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Extract Voices from Music Signal” on page 21-78

- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

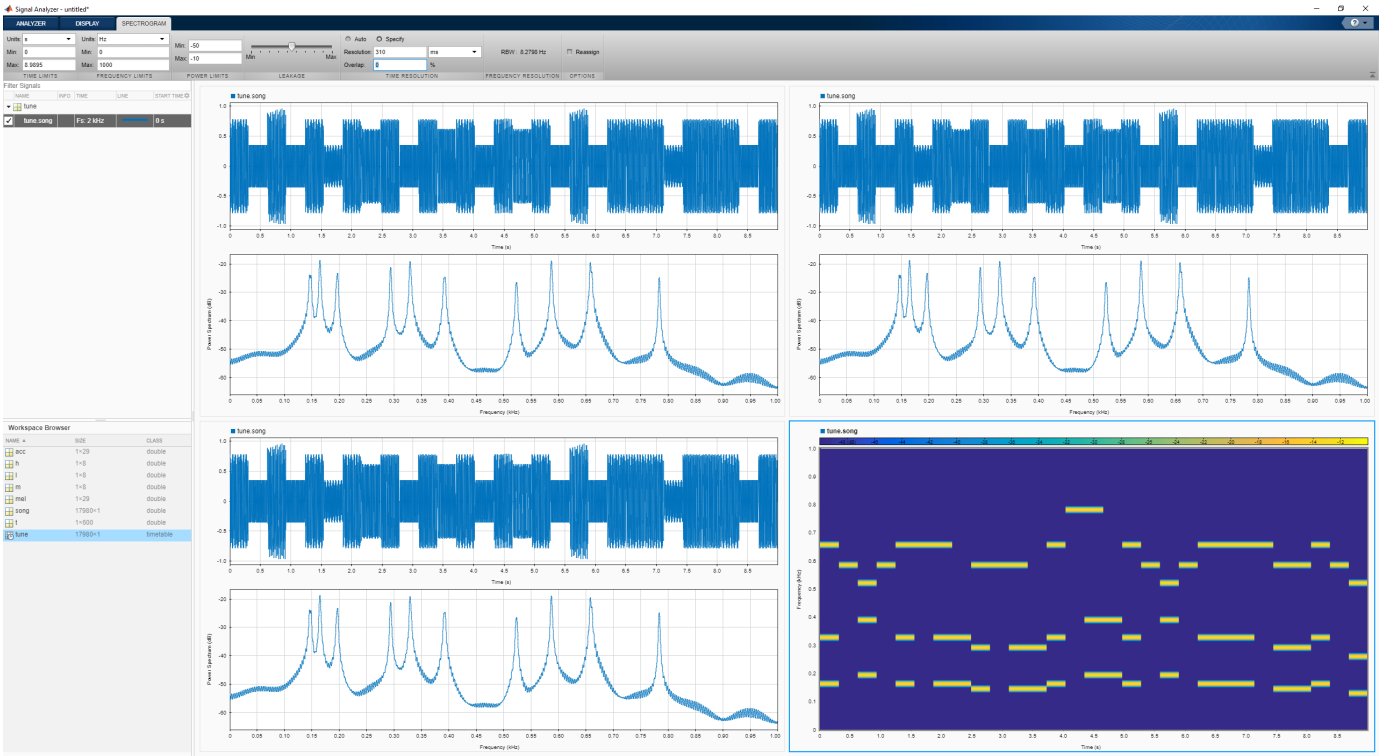
- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Extract Voices from Music Signal

Implement a basic digital music synthesizer and use it to play a traditional song in a three-voice arrangement. Specify a sample rate of 2 kHz. Save the song as a MATLAB® timetable.

```
fs = 2e3;
t = 0:1/fs:0.3-1/fs;

l = [0 130.81 146.83 164.81 174.61 196.00 220 246.94];
m = [0 261.63 293.66 329.63 349.23 392.00 440 493.88];
h = [0 523.25 587.33 659.25 698.46 783.99 880 987.77];
note = @(f,g) [1 1 1]*sin(2*pi*[l(g) m(g) h(f)]'.*t);

mel = [3 2 1 2 3 3 3 0 2 2 2 0 3 5 5 0 3 2 1 2 3 3 3 3 2 2 3 2 1]+1;
acc = [3 0 5 0 3 0 3 3 2 0 2 2 3 0 5 5 3 0 5 0 3 3 3 0 2 2 3 0 1]+1;

song = [];
for kj = 1:length(mel)
    song = [song note(mel(kj),acc(kj)) zeros(1,0.01*fs)];
end
song = song'/(max(abs(song))+0.1);

% To hear, type sound(song,fs)

tune = timetable(seconds((0:length(song)-1)'/fs),song);
```

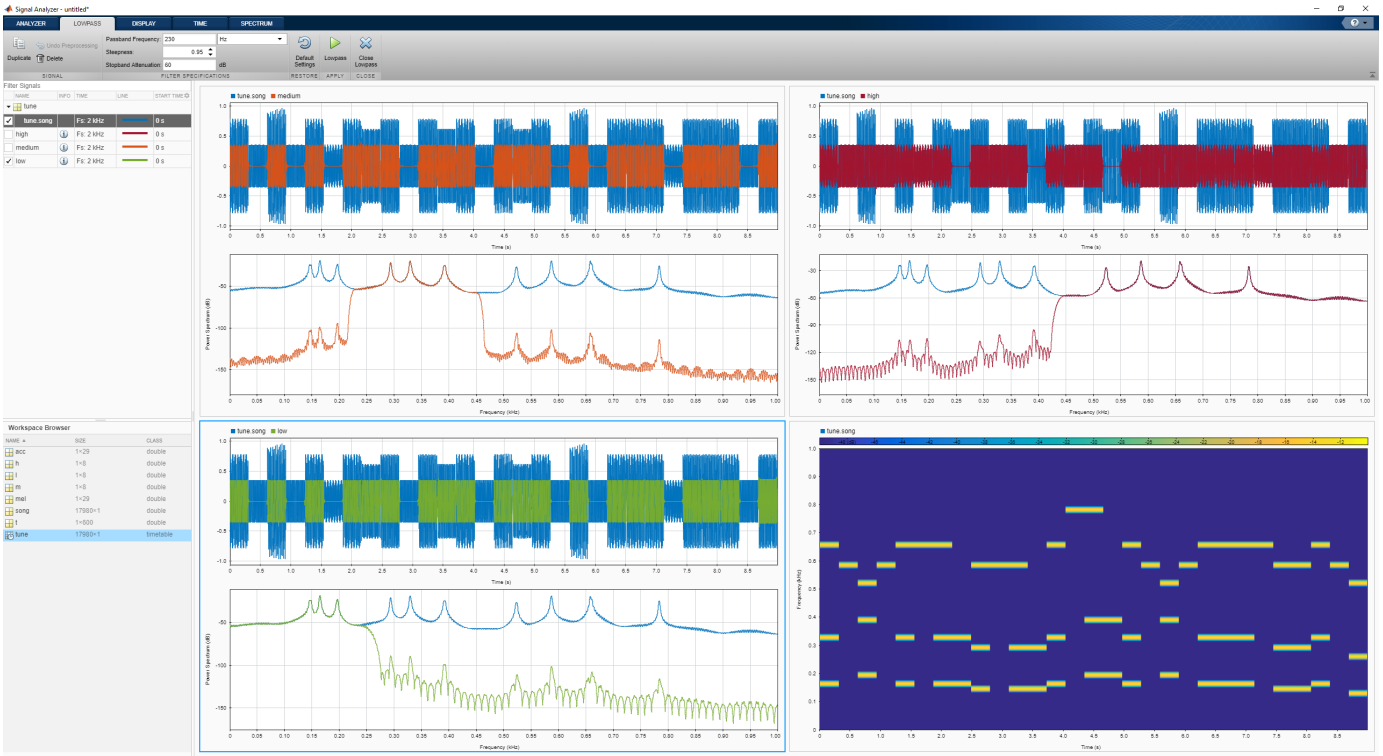
Open **Signal Analyzer** and drag the timetable from the Workspace browser to the Signal table. Click **Display Grid ▼** to create a two-by-two grid of displays. Select the top two displays and the lower left display and click the **Spectrum** button to add a spectrum view. Select the lower right display, click **Time-Frequency** to add a spectrogram view, and click **Time** to remove the time view. Drag the song to all four displays. Select the lower right display, and in the **Spectrogram** tab, specify a time resolution of 0.31 second (310 ms) and 0% overlap between adjoining segments. Set the **Power Limits** to −50 dB and −10 dB.



On the **Analyzer** tab, click **Duplicate** three times to create three copies of the song. Rename the copies as `high`, `medium`, and `low` by double-clicking the **Name** column in the Signal table. Move the copies to the top two and lower left displays.

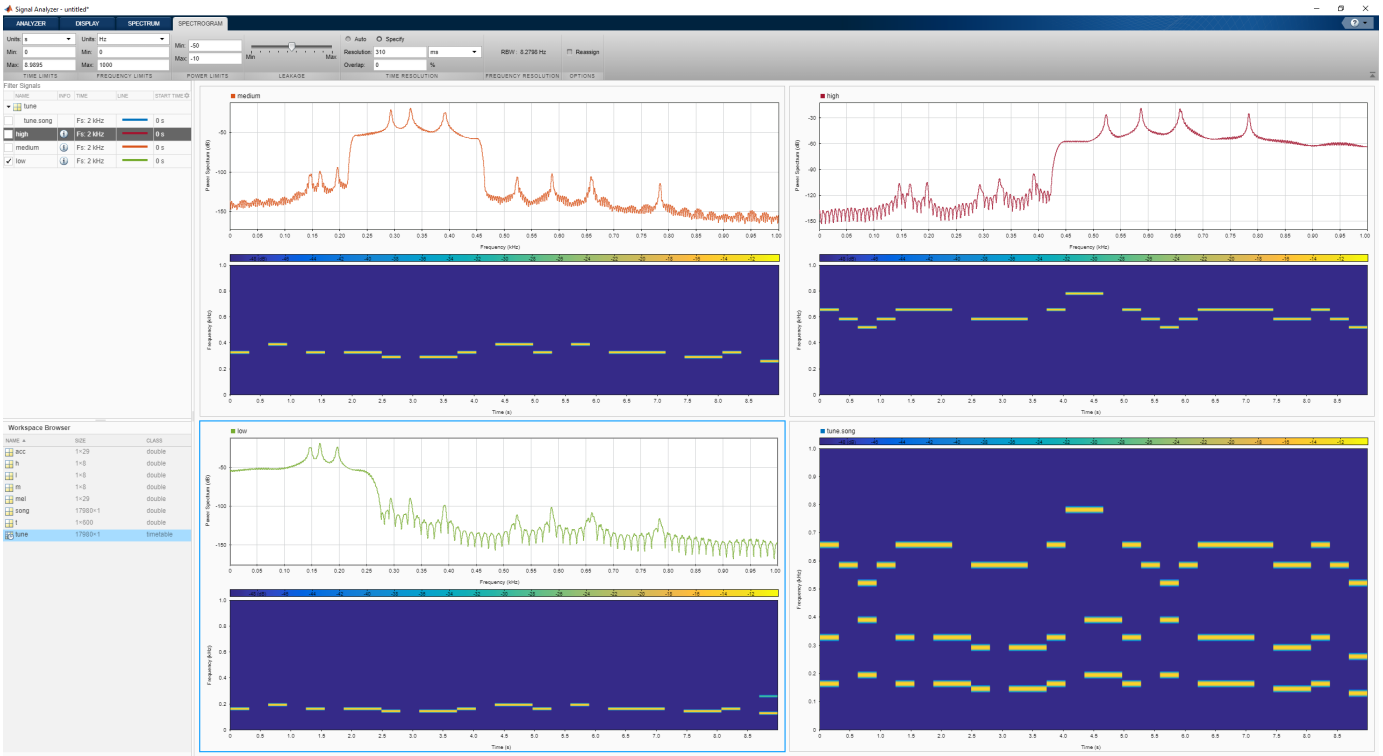
Preprocess the duplicate signals using filters.

1. Select the `high` signal by clicking its name in the Signal table. On the **Analyzer** tab, click **Highpass**. On the **Highpass** tab that appears, enter a passband frequency of 450 Hz and increase the steepness to 0.95. Click **Highpass**.
2. Select the `medium` signal by clicking its name in the Signal table. On the **Analyzer** tab, click **Preprocessing ▼** and select **Bandpass**. On the **Bandpass** tab that appears, enter 230 Hz and 450 Hz as the lower and upper passband frequencies, respectively. Increase the steepness to 0.95. Click **Bandpass**.
3. Select the `low` signal by clicking its name in the Signal table. On the **Analyzer** tab, click **Lowpass**. On the **Lowpass** tab that appears, enter a passband frequency of 230 Hz and increase the steepness to 0.95. Click **Lowpass**.



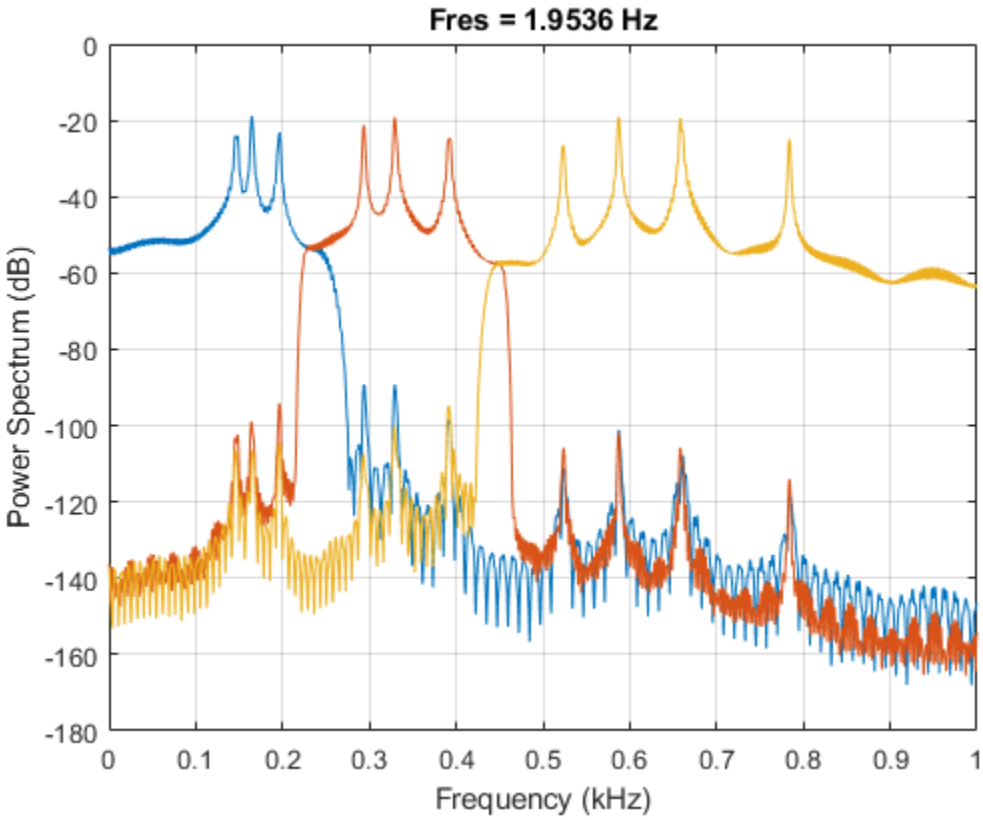
On each of the three displays containing filtered signals:

1. Remove the original signal by clearing the check box next to its name.
2. On the **Display** tab, click **Time-Frequency** to add a spectrogram view and click **Time** to remove the time view.
3. On the **Spectrogram** tab, specify a time resolution of 0.31 second and 0% overlap between adjoining segments. Set the **Power Limits** to −50 dB and −10 dB.



Select the three filtered signals by clicking their **Name** column in the Signal table. On the **Analyzer** tab, click **Export** and save the signals to a MAT-file called `music.mat`. In MATLAB, load the file to the workspace. Plot the spectra of the three signals.

```
load music

pspectrum(low)
hold on
pspectrum(medium)
pspectrum(high)
hold off
```




```
% To hear the different voices, type
% sound(low.low,fs), pause(5), sound(medium.medium,fs), pause(5), sound(high.high,fs)
```

## See Also

**Apps**
**Signal Analyzer**

**Functions**
`bandpass` | `bandstop` | `highpass` | `lowpass` | `pspectrum`

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70

- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Resample and Filter a Nonuniformly Sampled Signal

A person recorded their weight in pounds during the leap year 2012. The person did not record their weight every day, so the data are nonuniform. Use the **Signal Analyzer** app to preprocess and study the recorded weight. The app enables you to fill in the missing data points by interpolating the signal to a uniform grid. (This procedure gives the best results if the signal has only small gaps.)

Load the data and convert the measurements to kilograms. The data file has the missing readings set to `NaN`. There are 27 data points missing, most of them during a two-week stretch in August.

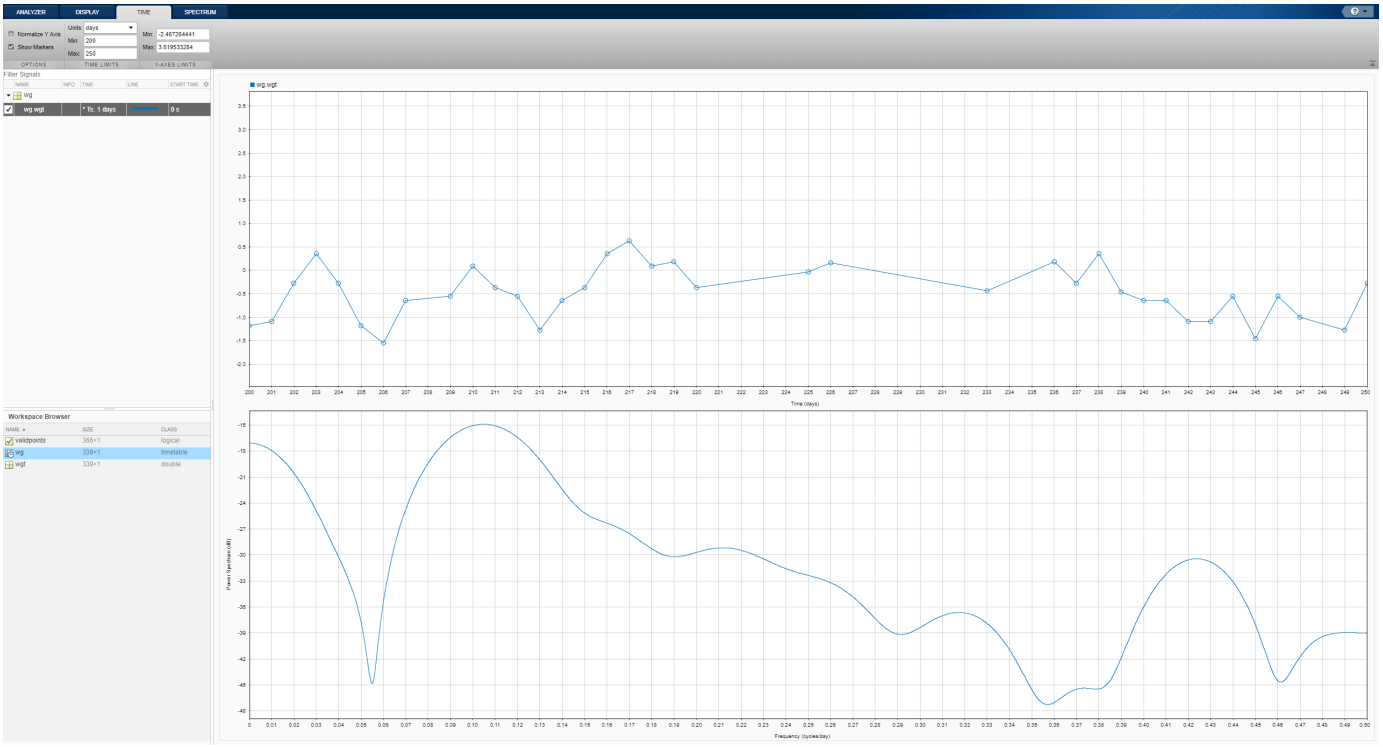
```
wt = datetime(2012,1,1:366)';

load weight2012.dat
wgt = weight2012(:,2)/2.20462;

validpoints = ~isnan(wgt);
missing = wt(~validpoints);
missing(15:26)
```

```
ans = 12x1 datetime
   09-Aug-2012
   10-Aug-2012
   11-Aug-2012
   12-Aug-2012
   15-Aug-2012
   16-Aug-2012
   17-Aug-2012
   18-Aug-2012
   19-Aug-2012
   20-Aug-2012
   22-Aug-2012
   23-Aug-2012
```

Store the data in a MATLAB® timetable. Remove the missing points. Remove the DC value to concentrate on fluctuations. Convert the time information to a `duration` array by subtracting the first time point. For more details, see “Data Types Supported by Signal Analyzer” on page 21-106.

```
wgt = wgt(validpoints);
wgt = wgt - mean(wgt);

wt = wt(validpoints);
wt = wt - wt(1);

wg = timetable(wt,wgt);
```

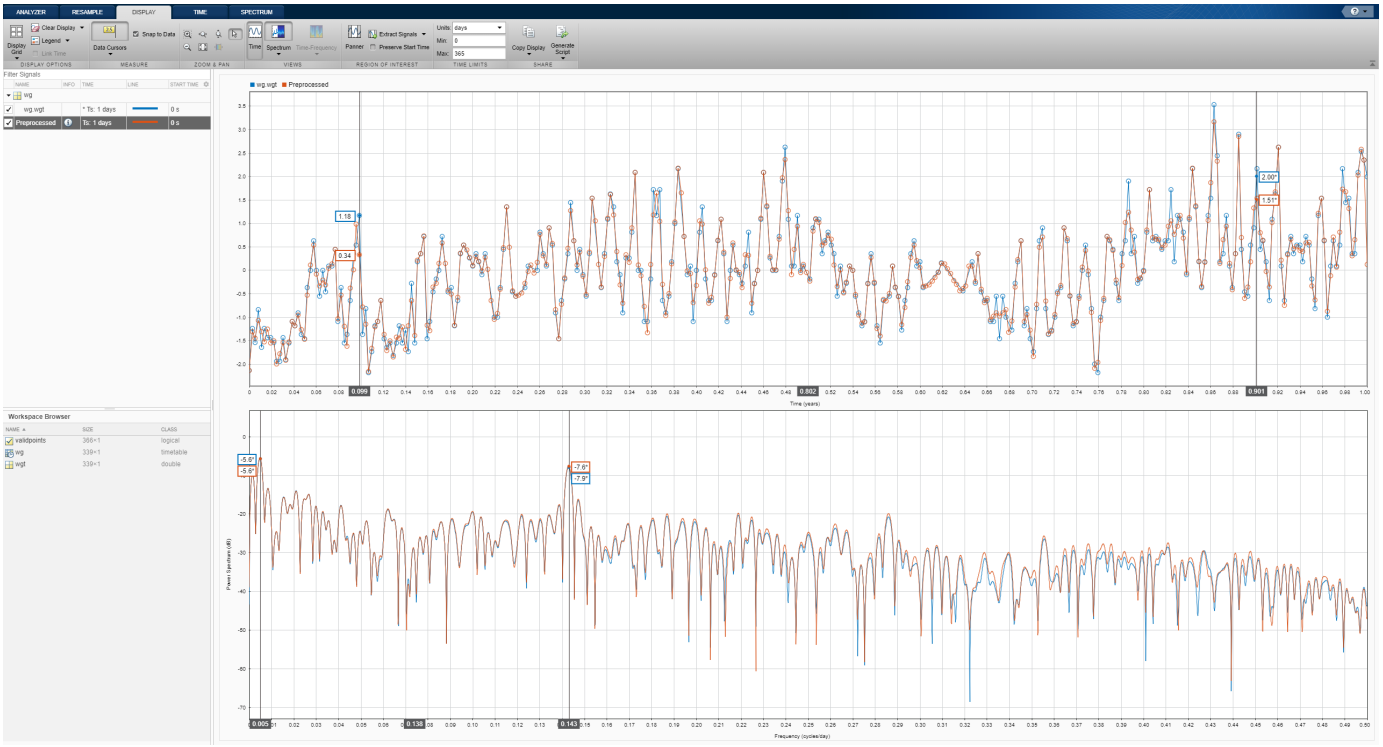
Open **Signal Analyzer** and drag the timetable to a display. On the **Display** tab, click **Spectrum** to open a spectrum view. On the **Time** tab, select **Show Markers**. Zoom into the missing stretch by setting the **Time Limits** to 200 and 250 days.



Right-click the signal in the Signal table and select `Duplicate`. Rename the copy as `Preprocessed` by double-clicking the **Name** column in the Signal table. Leave the `Preprocessed` signal selected. On the **Analyzer** tab, click **Preprocessing ▼** and select **Resample**. On the **Resample** tab that appears, enter a sample rate of `1 cycles/day` and select the `Shape Preserving Cubic` method. Click **Resample**. Overlay the resampled signal on the display by selecting the check box next to its name.

Zoom out to reveal the data for the whole year. On the **Spectrum** tab, set the leakage to the maximum value. The spectra of the original and resampled signals agree well for most frequencies. The spectrum shows two noticeable peaks, one at around 0.14 cycles/day and the other at very low frequencies. To locate the peaks better, click **Data Cursors** ▼ and select Two. Place the cursors on the peaks. Hover over the frequency field of each cursor to get a more precise value of its location.

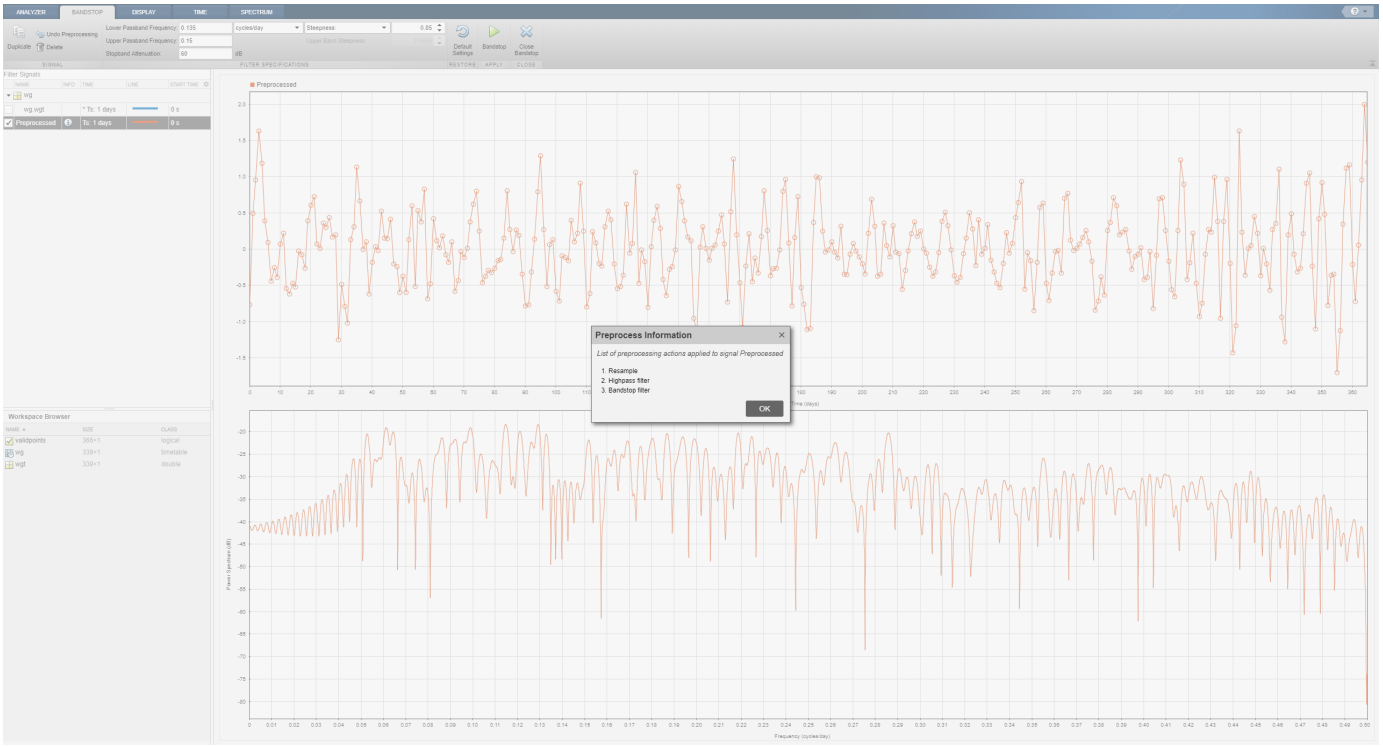
- The medium-frequency peak is at 0.142857 = 1/7 cycles/day, which corresponds to a one-week cycle.
- The low-frequency peak is at 0.004762 cycles/day, which corresponds to a 210-day cycle.



Remove the cursors by clicking the **Data Cursors** icon. Remove the original signal from the display. Filter the `Preprocessed` signal to remove the effects of the cycles.

1. To remove the low-frequency cycle, highpass-filter the signal. On the **Analyzer** tab, select **Highpass**. On the **Highpass** tab that appears, enter a passband frequency of `0.05 cycles/day`. Use the default values of the other parameters. Click **Highpass**.
2. To remove the weekly cycle, bandstop-filter the signal. On the **Analyzer** tab, click **Preprocessing ▼** and select **Bandstop**. On the **Bandstop** tab that replaces the **Highpass** tab, enter a lower passband frequency of `0.135 cycles/day` and a higher passband frequency of `0.15 cycles/day`. Use the default values of the other parameters. Click **Bandstop**.

The preprocessed signal shows less fluctuation than the original. The shape of the signal suggests the person's weight varies less in the summer months than in winter, but that may be an artifact of the resampling. Click the icon on the **Info** column in the Signal table entry for the `Preprocessed` signal to see the preprocessing steps performed on it.




To see a full summary of the preprocessing steps, including all the settings you chose, click **Generate Function** on the **Analyzer** tab. The generated function appears in the MATLAB® Editor.

```
function [y,ty] = preprocess(x,tx)
%  Preprocess input x
%    This function expects an input vector x and a vector of time values
%    tx. tx is a numeric vector in units of seconds.
%    Follow the timetable documentation (type 'doc timetable' in
%    command line) to learn how to index into a table variable and its time
%    values so that you can pass them into this function.

% Generated by MATLAB(R) 9.5 and Signal Processing Toolbox 8.1.
% Generated on: 08-Jun-2018 14:35:38

targetSampleRate = 1.1574074074074073e-05;
[y,ty] = resample(x,tx,targetSampleRate,'pchip');
Fs = 1/mean(diff(ty)); % Average sample rate
y = highpass(y,5.787e-07,Fs,'Steepness',0.85,'StopbandAttenuation',60);
y = bandstop(y,[1.5625e-06 1.73611111111111e-06],Fs,'Steepness',0.85,'StopbandAttenuation',60);
end
```

## See Also

**Apps**
**Signal Analyzer**

**Functions**
`bandstop` | `highpass` | `resample`

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

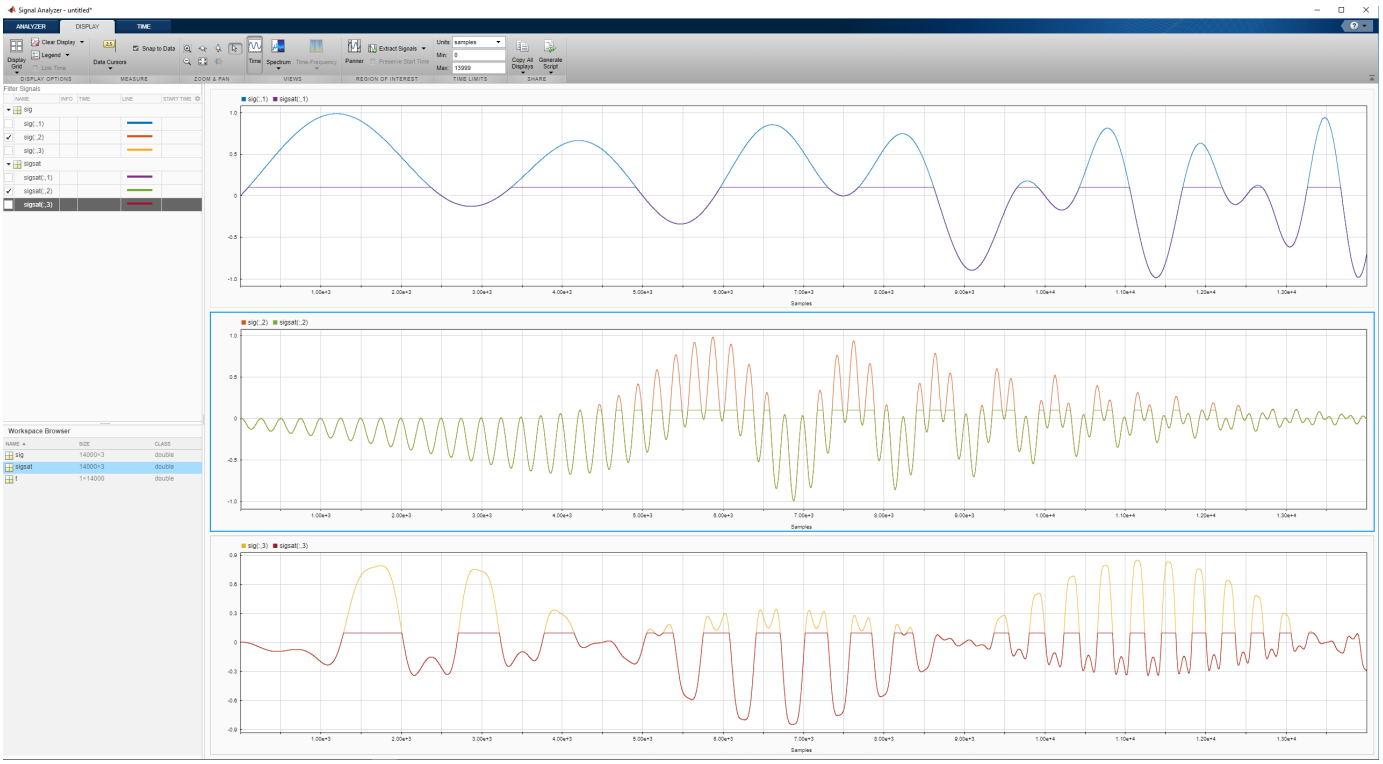
# Declip Saturated Signals Using Your Own Function

Sensors can return clipped readings if the data are larger than a given saturation point. To reconstruct the readings, you can fit a polynomial through the points adjacent to the saturated intervals. Write a function that performs the reconstruction and integrate it into **Signal Analyzer**.

Generate a three-channel signal sampled at 1 kHz for 14 seconds. The signal has several peaks of varying sizes and shapes. A sensor that reads the signal saturates at 0.1 V.

```
fs = 1000;
t = 0:1/fs:14-1/fs;

sig = [chirp(t-1,0.1,17,2,'quadratic',1).*sin(2*pi*t/5);
    chirp(t-2,2,2,2.1,'quadratic',100).*exp(-(t-6.5).^2/20).*sin(2*pi*t*2);
    0.85*besselj(0,5*(sin(2*pi*(t+1.5).^2/20).^2)).*sin(2*pi*t/9)]';

sigsat = sig;
stv = 0.1;
sigsat(sigsat >= stv) = stv;
```

Open **Signal Analyzer** and drag the original signal and the saturated signal to the Signal table. Drag each original and saturated channel to its own display.



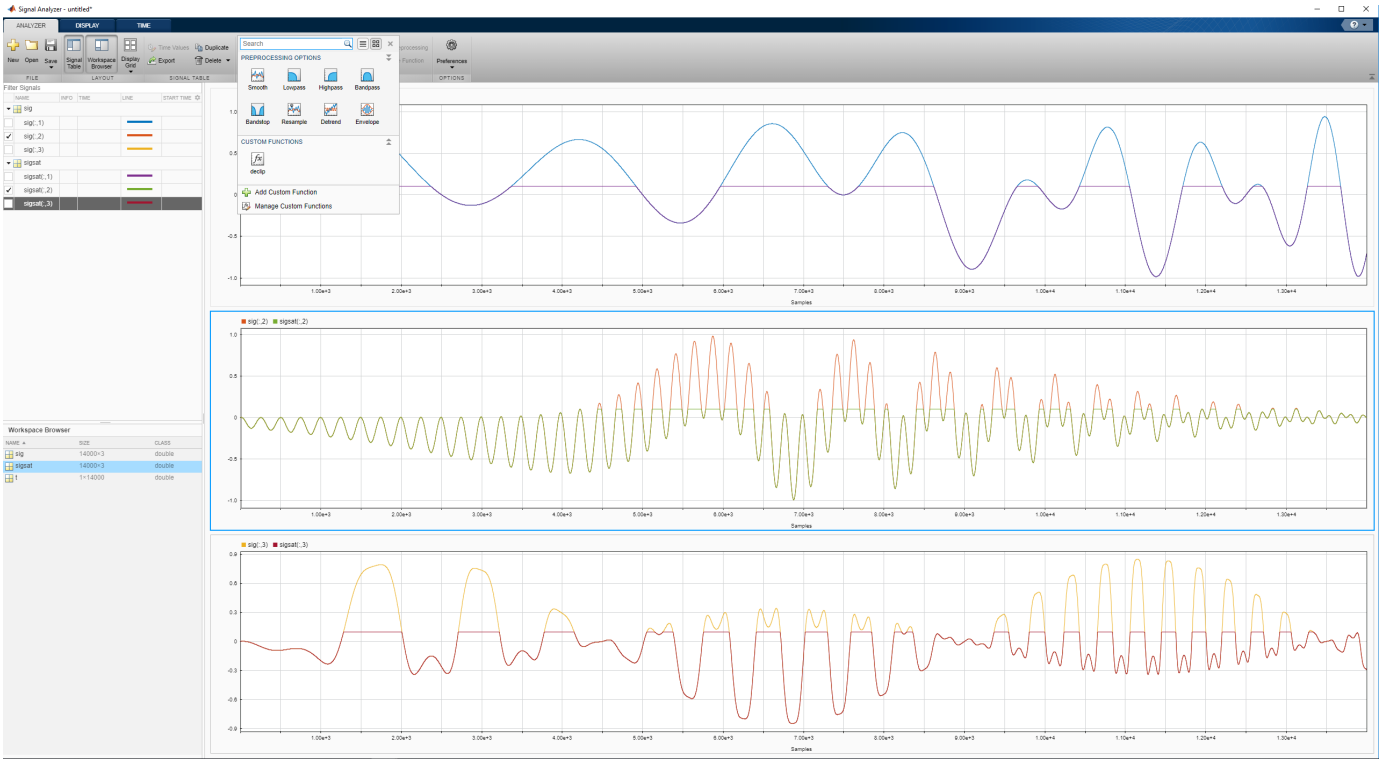
Write a function that uses a polynomial to reconstruct the signal peaks:

- The first input argument, `x`, is the input signal. This argument must be a vector and is treated as a single channel.

- The second input argument, `tIn`, is a vector of time values. The vector must have the same length as the signal. If the input signal has no time information, the function reads this argument as an empty array.
- Use `varargin` to specify additional input arguments. If you do not have additional input arguments, you can omit `varargin`. Enter the additional arguments as an ordered comma-separated list in the **Preprocess** tab.
- The first output argument, `y`, is the preprocessed signal.
- The second output argument, `tOut`, is a vector of output time values. If the input signal has no time information, `tOut` is returned as an empty array.
- To implement your algorithm, you can use any MATLAB® or Signal Processing Toolbox™ function.

```
function [y,tOut] = declip(x,tIn,varargin)
% Declip saturated signal by fitting a polynomial

    % Initialize the output signal

    y = x;

    % For signals with no time information, use sample numbers as abscissas

    if isempty(tIn)
        tOut = [];
        t = (1:length(x))';
    else
        t = tIn;
        tOut = t;
    end

    % Specify the degree of the polynomial as an optional input argument
    % and provide a default value of 4

    if nargin<3
        ndx = 4;
    else
        ndx = varargin{1};
    end

    % To implement your algorithm, you can use any MATLAB or Signal
    % Processing Toolbox function

    % Find the intervals where the signal is saturated and generate an
    % array containing the interval endpoints
    idx = find(x==max(x));
    fir = [true;diff(idx)~=1];
    ide = [idx(fir) idx(fir([2:end 1]))];
    % For each interval, fit a polynomial of degree ndx over the ndx+1 points
    % before the interval and the ndx+1 points after the interval
    for k = 1:size(ide,1)
        bef = ide(k,1); aft = ide(k,2);
        intv = [bef-1+(-ndx:0) aft+1+(0:ndx)];
        [pp,~,mu] = polyfit(t(intv),x(intv),ndx);
        y(bef:aft) = polyval(pp,t(bef:aft),[],mu);
    end

end
```

Add the function to **Signal Analyzer** as a custom preprocessing function. On the **Analyzer** tab, click **Preprocessing ▼** and select **Add Custom Function**. Input the function name and description. Paste the text of your function in the editor window that appears. Save the file. The function appears in the preprocessing gallery.




Demonstrate that the function you created reconstructs the saturated regions.

1. Select the first channel of the saturated signal in the Signal table.
2. On the **Analyzer** tab, click **Preprocessing ▼** and select **declip**.
3. On the **Preprocessing** tab that appears, click **Preprocess**.

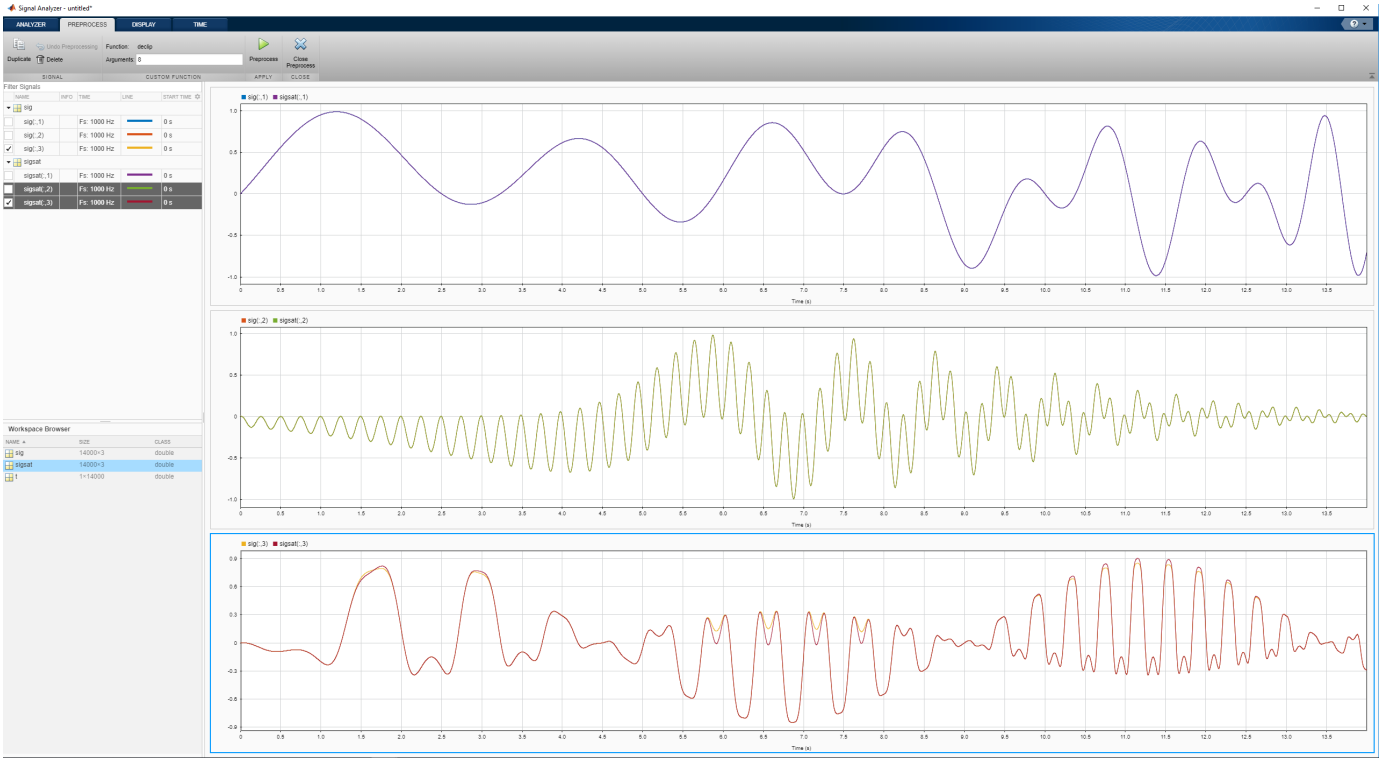
Verify that the preprocessing function works when the signals have time information.

1. Select `sig` and `sigsat` in the Signal table. Do not select individual channels.
2. On the **Analyzer** tab, click **Time Values**, select `Sample Rate and Start Time`, and specify `fs` as the sample rate.
3. The signal in the top panel, including the reconstructed regions, has time information.

| | NAME | INFO | TIME | LINE | START TIME |
|---|---|---|---|---|---|
| ▾ | sig | | | | |
| ☐ | sig(:,1) | | Fs: 1000 Hz | | 0 s |
| ☐ | sig(:,2) | | Fs: 1000 Hz | | 0 s |
| ☑ | sig(:,3) | | Fs: 1000 Hz | | 0 s |
| ▾ | sigsat | | | | |
| ☐ | sigsat(:,1) | ⓘ | Fs: 1000 Hz | | 0 s |
| ☐ | sigsat(:,2) | ⓘ | Fs: 1000 Hz | | 0 s |
| ☑ | sigsat(:,3) | ⓘ | Fs: 1000 Hz | | 0 s |

Check that the function works when you specify optional inputs.

1. Select the second and third channels of the saturated signal in the Signal table.
2. On the **Preprocessing** tab, enter 8 in the **Arguments** field and click **Preprocess**. The preprocessing function uses a polynomial of degree 8 to reconstruct the saturated regions.



## See Also

**Apps**
**Signal Analyzer**

**Functions**
`polyfit` | `polyval` | `varargin`

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

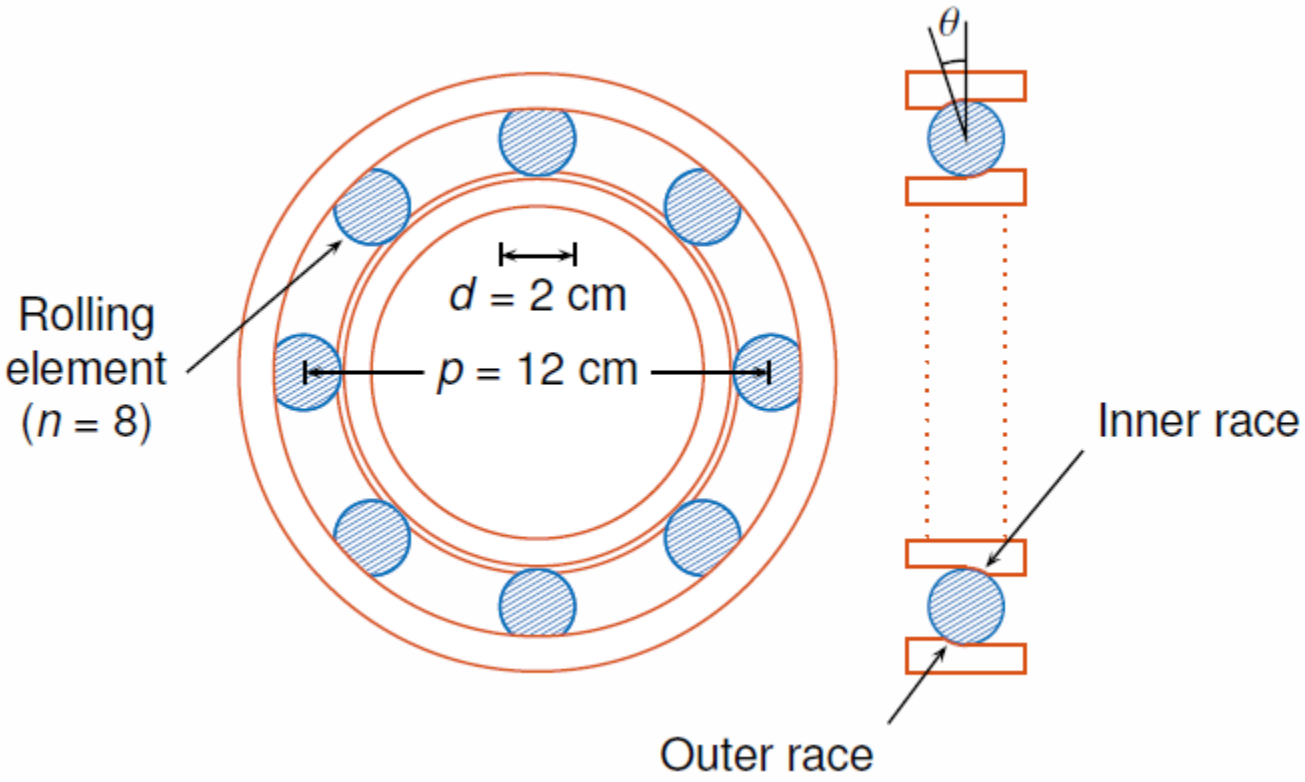
- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Compute Envelope Spectrum of Vibration Signal

Use **Signal Analyzer** to compute the envelope spectrum of a bearing vibration signal and look for defects. Generate MATLAB® scripts and functions to automate the analysis.

### Generate Bearing Vibration Data

A bearing with the dimensions shown in the figure is driven at $f_0$ = 25 cycles per second. An accelerometer samples the bearing vibrations at 10 kHz.



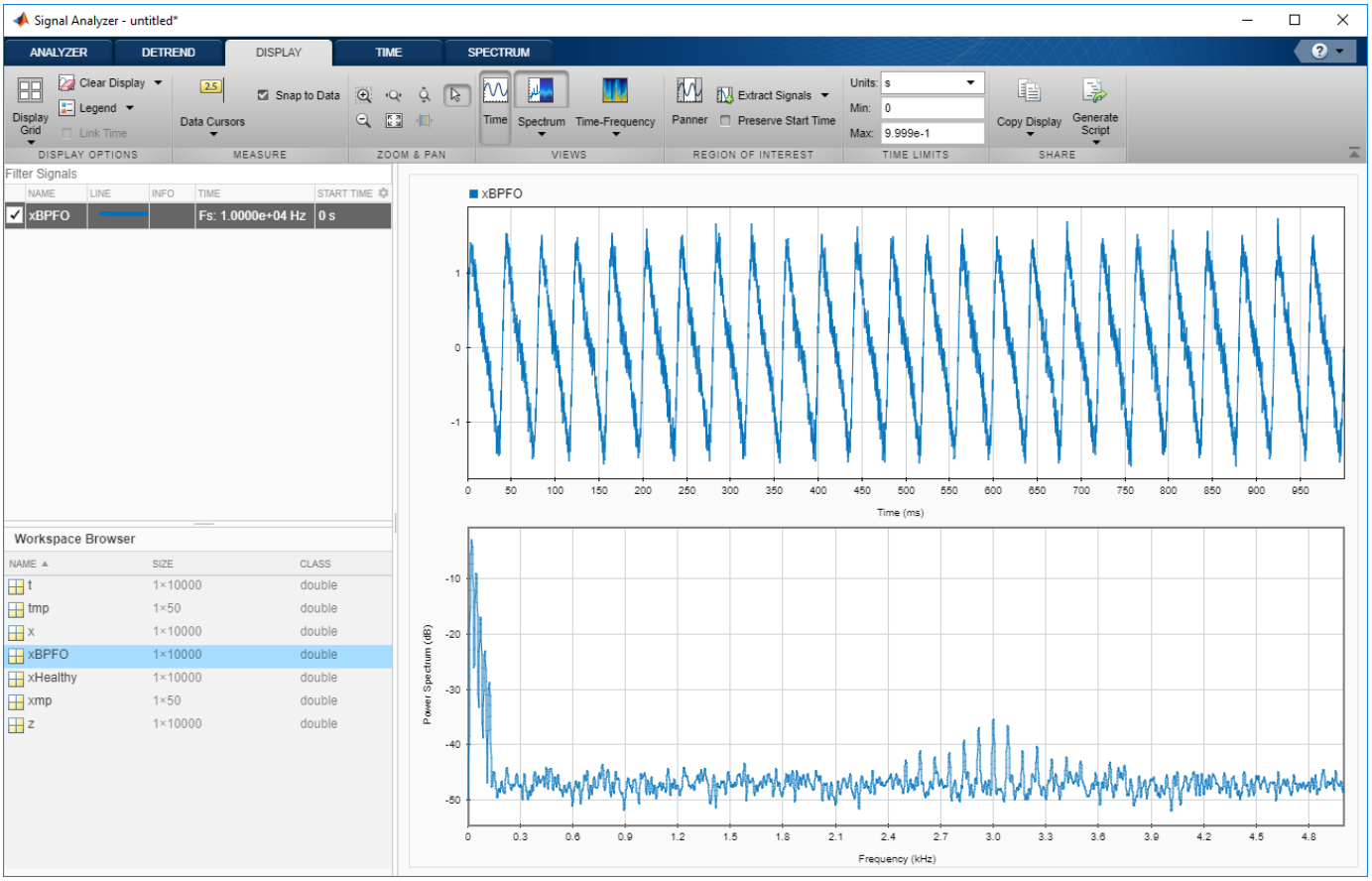

Generate vibration signals from two defective bearings using the `bearingdata` on page 21-0 function at the end of the example. In one of the signals, `xBPFO`, the bearing has a defect in the outer race. In the other signal, `xBPFI`, the bearing has a defect in the inner race. For more details on modeling and diagnosing defects in bearings, see “Vibration Analysis of Rotating Machinery” on page 24-435 and `envspectrum`.

```
[t,xBPFO,xBPFI,bpfi] = bearingdata;
```

### Compute Envelope Spectrum Using Signal Analyzer

Open **Signal Analyzer** and drag the BPFO signal to a display. Add time information to the signal by selecting it in the Signal table and clicking the **Time Values** button on the **Analyzer** tab. Select the `Sample Rate and Start Time` option and enter the 10 kHz sample rate.

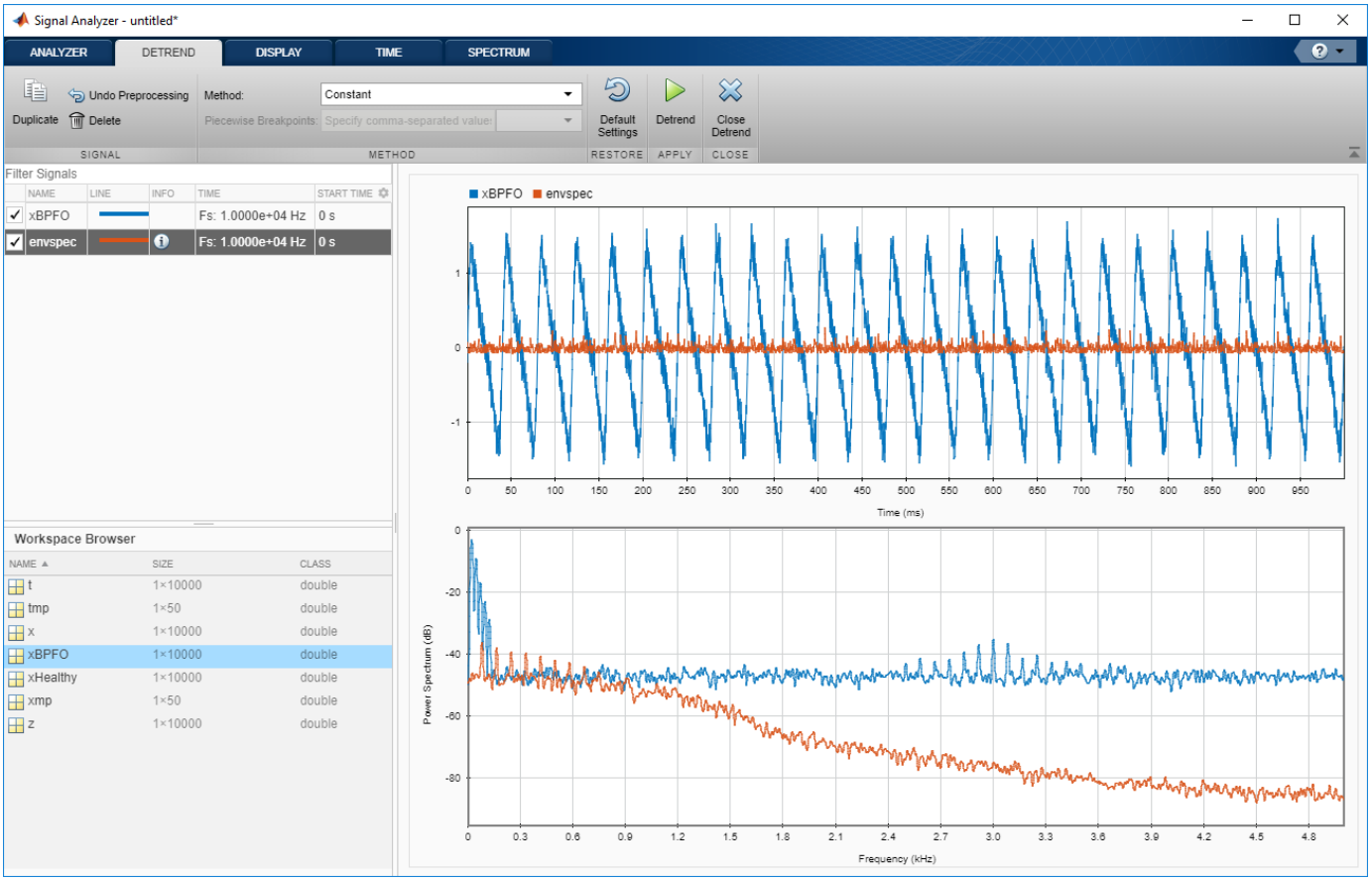
On the **Display** tab, click **Spectrum** to open a spectrum view. The spectrum of the vibration signal shows BPFO harmonics modulated by the 3 kHz impact frequency. At the low end of the spectrum, the driving frequency and its orders obscure other features.




Select the signal and, on the **Analyzer** tab, click **Duplicate** to generate a copy of it. Give the new signal the name `envspec` and drag it to the display. Compute the envelope spectrum of the signal using the Hilbert transform:

1. Remove the DC value of the signal. On the **Analyzer** tab, click **Preprocessing ▼** and select **Detrend**. On the **Detrend** tab that appears, select `Constant` as the method. Click **Detrend**.
2. Bandpass-filter the detrended signal. On the **Analyzer** tab, click **Preprocessing ▼** and select **Bandpass**. On the **Bandpass** tab that appears, enter 2250 Hz and 3750 Hz as the lower and upper passband frequencies, respectively. Click **Bandpass**.
3. Compute the envelope of the filtered signal. On the **Analyzer** tab, click **Preprocessing ▼** and select **Envelope**. On the **Envelope** tab that appears, select `Hilbert` as the method. Click **Envelope**.
4. Remove the DC value of the envelope using **Detrend**.

The envelope spectrum appears in the spectrum view of the display. The envelope spectrum clearly displays the BPFO harmonics.




### Steps to Create an Integrated Analysis Script

The computation of the envelope spectrum can get tedious if it has to be repeated for many different bearings. **Signal Analyzer** can generate MATLAB® scripts and functions to help you automate the computation.

As an exercise, repeat the previous analysis for the BPFI signal. **Signal Analyzer** generates two components useful for the automation:

1. `preprocess` on page 21-0 , a function that preprocesses the signal by detrending it, filtering it, and computing its envelope
2. `Compute power spectrum` on page 21-0 , a script that computes the envelope spectrum

To create the integrated analysis script, put the preprocessing function and the plotting script together unchanged in a single file. (Alternatively, you can save functions in separate files.)

- If you save the script and the function in a single MATLAB® script, keep in mind that functions must appear at the end.
- You must add the keyword `end` at the end of each function.

#### 1. Create Preprocessing Function

Initially, create the function that reproduces the preprocessing steps. Select the `envspec` signal. On the **Analyzer** tab, click **Generate Function**. The function, called `preprocess` by default, appears in

the Editor. Save the generated function at the end of your integrated analysis script. The function expects a second argument specifying the time information. Preprocess the BPFI signal using the function.

```
envspec = preprocess(xBPFI,t);
```

### 2. Create Spectrum Script

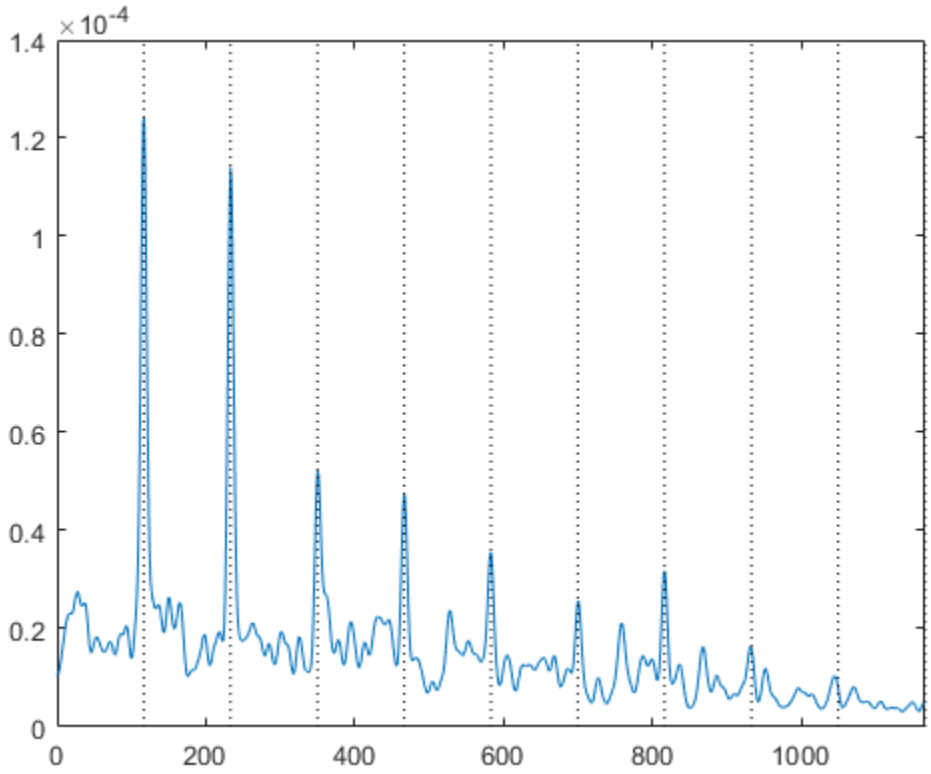
In the app, remove the unprocessed signal from the display by clearing the check box next to its name. On the **Display** tab, click **Generate Script ▼** and select `Spectrum Script`. The script appears in the Editor. Include the generated code in your integrated analysis script. When you run the analysis script, the generated spectrum script computes the envelope spectrum of the preprocessed BPFI signal.

```
% Compute power spectrum

% Generated by MATLAB(R) 9.6 and Signal Processing Toolbox 8.2.
% Generated on: 12-Nov-2018 15:13:34

% Parameters
timeLimits = [0 0.9999]; % seconds
frequencyLimits = [0 5000]; % Hz

%%
% Index into signal time region of interest
envspec_ROI = envspec(:);
sampleRate = 10000; % Hz
startTime = 0; % seconds
minIdx = ceil(max((timeLimits(1)-startTime)*sampleRate,0))+1;
maxIdx = floor(min((timeLimits(2)-startTime)*sampleRate,length(envspec_ROI)-1))+1;
envspec_ROI = envspec_ROI(minIdx:maxIdx);

% Compute spectral estimate
% Run the function call below without output arguments to plot the results
[Penvspec_ROI, Fenvspec_ROI] = pspectrum(envspec_ROI,sampleRate, ...
    'FrequencyLimits',frequencyLimits);
```

### 3. Plot Envelope Spectrum

Plot the envelope spectrum. Compare the peak locations to the frequencies of the first ten BPFI harmonics. You can also plot the envelope spectrum using the `pspectrum` command with no output arguments.

```
plot(Fenvspec_ROI,(Penvspec_ROI))
hold on
[X,Y] = meshgrid((1:10)*bpfi,ylim);
plot(X,Y,':k')
hold off
xlim([0 10*bpfi])
```




### Function Code

#### Signal Preprocessing Function

The signal preprocessing function generated by the app combines detrending, bandpass filtering, and envelope computation.

```
function y = preprocess(x,tx)
%  Preprocess input x
%    This function expects an input vector x and a vector of time values
%    tx. tx is a numeric vector in units of seconds.

% Generated by MATLAB(R) 9.6 and Signal Processing Toolbox 8.2.
% Generated on: 12-Nov-2018 15:09:44

y = detrend(x,'constant');
Fs = 1/mean(diff(tx)); % Average sample rate
y = bandpass(y,[2250 3750],Fs,'Steepness',0.85,'StopbandAttenuation',60);
[y,~] = envelope(y);
y = detrend(y,'constant');
end
```

#### Bearing Data Generating Function

The bearing has pitch diameter $p = 12$ cm and a bearing contact angle $\theta = 0$. Each of the $n = 8$ rolling elements has a diameter $d = 2$ cm. The outer race remains stationary as the inner race is driven at $f_0 = 25$ cycles per second. An accelerometer samples the bearing vibrations at 10 kHz.

```
function [t,xBPFO,xBPFI,bpfi] = bearingdata

p = 0.12;
d = 0.02;
n = 8;
th = 0;
f0 = 25;
fs = 10000;
```

For a healthy bearing, the vibration signal is a superposition of several orders of the driving frequency, embedded in white Gaussian noise.

```
t = 0:1/fs:1-1/fs;
z = [1 0.5 0.2 0.1 0.05]*sin(2*pi*f0*[1 2 3 4 5]'.*t);

xHealthy = z + randn(size(z))/10;
```

A defect in the outer race causes a series of 5 millisecond impacts that over time result in bearing wear. The impacts occur at the ball pass frequency outer race (BPFO) of the bearing,

$$\text{BPFO} = \frac{1}{2}nf_0\left[1 - \frac{d}{p}\cos\theta\right].$$

Model the impacts as a periodic train of 3 kHz exponentially damped sinusoids. Add the impacts to the healthy signal to generate the BPFO vibration signal.

```
bpfo = n*f0/2*(1-d/p*cos(th));

tmp = 0:1/fs:5e-3-1/fs;
xmp = sin(2*pi*3000*tmp).*exp(-1000*tmp);

xBPFO = xHealthy + pulstran(t,0:1/bpfo:1,xmp,fs)/4;
```

If the defect is instead in the inner race, the impacts occur at a frequency

$$\text{BPFI} = \frac{1}{2}nf_0\left[1 + \frac{d}{p}\cos\theta\right].$$

Generate the BPFI vibration signal by adding the impacts to the healthy signals.

```
bpfi = n*f0/2*(1+d/p*cos(th));

xBPFI = xHealthy + pulstran(t,0:1/bpfi:1,xmp,fs)/4;

end
```

## See Also

**Apps**
**Signal Analyzer**

**Functions**
`polyfit` | `polyval` | `varargin`

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

- “Using Signal Analyzer App” on page 21-2
- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Edit Sample Rate and Other Time Information

You can add and edit the time information in the **Signal Analyzer** app for any signal that is not a `timetable` or a `timeseries` object. Select one or more signals with no inherent time information in the Signal table and on the **Analyzer** tab, click **Time Values**.

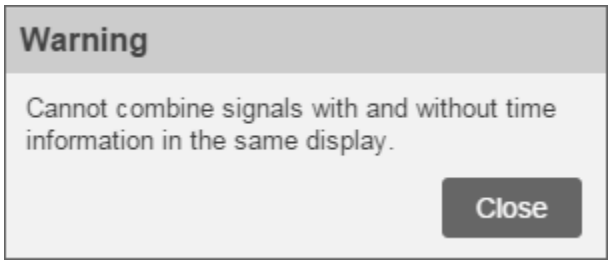
**Note** Select a signal in the Signal table by clicking its **Name** column. The complete row is highlighted, indicating that the signal is selected. The check box next to the name of the signal indicates whether or not the signal is plotted in the active display.

In the **Time Values** dialog box, select a **Time Specification** option.

| Time Specification Option | Description |
|---|---|
| `Work in Samples` (default) | This option enables you to explore signals without the need to specify a sample rate or a sample time. It is equivalent to plotting the signal in MATLAB without *x*-axis information. |
| `Sample Rate and Start Time` | Use this option when you know the rate at which the signal has been sampled. Specify the sample rate and the instant corresponding to the first sample.<br><br>The **Sample Rate** can be expressed in Hz, kHz, MHz, or GHz.<br><br>The **Start Time** can be expressed in seconds, years, days, hours, minutes, milliseconds, microseconds, or nanoseconds.<br><br>Set the sample rate so that the signal is plotted in units of time on a display. |
| `Sample Time and Start Time` | Use this option when you know the time interval between samples. Specify the sample time and the instant corresponding to the first sample.<br><br>The **Sample Time** and **Start Time** each can be expressed in seconds, years, days, hours, minutes, milliseconds, microseconds, or nanoseconds.<br><br>Set the sample time so that the signal is plotted in units of time on a display. |

| Time Specification Option | Description |
| --- | --- |
| `Time Values` | Use this option when you know the time value corresponding to each sample. Specify the time values using a MATLAB expression or the name of a variable in the MATLAB workspace.<br><br>The **Time Values** can be stored in a numeric vector with real time values expressed in seconds. The values must be unique and cannot be `NaN`, but they need not be uniformly spaced. The vector must have the same length as the signal.<br><br>The time values can also be stored in a `duration` array. The values must be unique and cannot be `NaN`, but they need not be uniformly spaced. The array must have the same length as the signal.<br><br>The time values can also be entered as a MATLAB expression. The expression must specify an array with the same length as the signal. The values must be unique and cannot be `NaN`, but they need not be uniformly spaced. Valid examples include:<br><br>• `(0:length(s)-1)'/Fs`, where `s` is the signal and `Fs` is a scalar in the workspace representing a sample rate.<br>• `linspace(2,2.5,length(s))'`, where `s` is the signal.<br>• `minutes(0:15)'`, equivalent to taking measurements every minute for 15 minutes.<br>• `[0:10 20:30]`, equivalent to taking two sets of measurements at 1 Hz with a long pause between the sets.<br><br>In all cases, the app derives a sample rate from the time values and displays it in the **Time** column of the Signal table. An asterisk preceding the sample rate indicates that the signal is nonuniformly sampled. |

**Note** Signals with no time information are plotted in units of samples on the *x*-axis. Signals with time information are plotted in units of time on the *x*-axis. To plot several signals on the same display, ensure that they all have time information or are all in samples. Otherwise, you get a warning:




- If a signal has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.
- If a signal is nonuniformly sampled, then **Signal Analyzer** interpolates the signal to a uniform grid to compute spectral estimates. The app uses linear interpolation and assumes a sample time equal to the median difference between adjacent time points. The derived sample rate in the Signal table has an asterisk to indicate that the signal is nonuniformly sampled.

  **Note** The interpolation is used only to compute spectral estimates. Time plots are not resampled.

  For a nonuniformly sampled signal to be supported, the median time interval and the mean time interval must obey

  $$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$
- Filtering and scalogram view do not support nonuniformly sampled signals.
- The app does not support adding time information to `labeledSignalSet` objects or editing the time information of `labeledSignalSet` objects.

## See Also

**Signal Analyzer**

## Related Examples

- “Find Delay Between Correlated Signals” on page 21-30
- “Plot Signals from the Command Line” on page 21-35
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Extract Voices from Music Signal” on page 21-78
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

- “Data Types Supported by Signal Analyzer” on page 21-106
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Data Types Supported by Signal Analyzer

## Numeric Data

- Numeric vectors and matrices with finite elements are supported.

  **Example:** `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal consisting of sinusoids embedded in white noise.
- Scalars, empty arrays, multidimensional arrays, and the `ans` variable are not supported.

## MATLAB Timetables

- Timetables with one or more variables are supported. Each variable can be a vector or a matrix. **Signal Analyzer** supports timetable inputs only when the time values are increasing and finite. Signals with missing, nonfinite, or duplicate time points are not imported. For some timetables, this restriction might mean that the app imports some signals but does not import others. To make sure that all signals are imported, you can fix them using the tips in "Clean Timetable with Missing, Duplicate, or Nonuniform Times".

  **Example:** `timetable(seconds(0:4)',rand(5,2))` and `timetable(seconds(0:4)',rand(5,1),rand(5,1))` both specify a two-channel random variable sampled at 1 Hz for 4 seconds.
- Empty timetables and timetables with row times specified as `datetime` arrays are not supported.

**Tip** To analyze timetables with time values stored as a `datetime` array, convert the array to a `duration` array by subtracting the first time point, and then convert the `duration` array to seconds. See "Analyze Signals with Inherent Time Information" on page 21-45 for an example.

## timeseries Objects

- Single-channel and multichannel `timeseries` objects are supported. To be supported, a `timeseries` object must have its `DataInfo.Interpolation` property set to `'linear'`. Use `setinterpmethod` to change the property.

  **Example:** `timeseries(rand(5,2))` and `timeseries(rand(5,2),0:4)` both specify a two-channel random variable sampled at 1 Hz for 4 seconds.
- **Signal Analyzer** supports `timeseries` inputs only when the time values are increasing and finite. Signals with missing, nonfinite, or duplicate time points are not imported. For some `timeseries` objects, this restriction might mean that the app imports some signals but does not import others. To make sure that all signals are imported, you can fix them using the tips in "Time Series Objects and Collections".
- Empty `timeseries` objects, `timeseries` objects with time vectors specified as MATLAB date strings, and `timeseries` objects whose `Name` property is not a valid MATLAB variable name are not supported. See `isvarname` for more information on valid variable names.

**Note** **Signal Analyzer** does not support matrices, time series, timetables, or labeled signal sets with more than 8000 channels.

## Nonuniformly Sampled Signals

- Filtering and scalogram view do not support nonuniformly sampled signals.
- If a signal is nonuniformly sampled, then **Signal Analyzer** interpolates the signal to a uniform grid to compute spectral estimates. The app uses linear interpolation and assumes a sample time equal to the median difference between adjacent time points. The derived sample rate in the Signal table has an asterisk to indicate that the signal is nonuniformly sampled. For a nonuniformly sampled signal to be supported, the median time interval and the mean time interval must obey

  $$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

  **Note** The interpolation is used only to compute spectral estimates. Time plots are not resampled.

## Labeled Signal Sets

- `labeledSignalSet` objects are supported.

  **Example:** The code

  ```
  lbs = labeledSignalSet({randn(100,2) randn(200,3)},'SampleRate',400);
  setMemberNames(lbs,["Water" "Earth"]);
  addMembers(lbs,{randn(120,1) randn(300,2)},100,["Air" "Fire"]);
  ```

  specifies a labeled signal set with four members. Each member has a different length and a different number of channels. Two members, `"Water"` and `"Earth"`, are sampled at 400 Hz. The other two members, `"Air"` and `"Fire"`, are sampled at 100 Hz.
- Preprocessing is not supported for labeled signal sets.
- The app does not support adding time information to `labeledSignalSet` objects or editing the time information of `labeledSignalSet` objects.

## See Also

**Apps**
**Signal Analyzer**

**Functions**
`labeledSignalSet` | `timeseries` | `timetable`

## Related Examples

- “Plot Signals from the Command Line” on page 21-35
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

- “Edit Sample Rate and Other Time Information” on page 21-103
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126
- “Customize Signal Analyzer” on page 21-130

# Spectrum Computation in Signal Analyzer

To compute signal spectra, **Signal Analyzer** finds a compromise between the spectral resolution achievable with the entire length of the signal and the performance limitations that result from computing large FFTs.

- If the resolution resulting from analyzing the full signal is achievable, the app computes a single modified periodogram of the whole signal using an adjustable Kaiser window.
- If the resolution resulting from analyzing the full signal is not achievable, the app computes a Welch periodogram: It divides the signal into overlapping segments, windows each segment using a Kaiser window, and averages the periodograms of the segments.

## Spectral Windowing

Any real-world signal is measurable only for a finite length of time. This fact introduces nonnegligible effects into Fourier analysis, which assumes that signals are either periodic or infinitely long. Spectral windowing, which consists of assigning different weights to different signal samples, deals systematically with finite-size effects.

The simplest way to window a signal is to assume that it is identically zero outside of the measurement interval and that all samples are equally significant. This “rectangular window” has discontinuous jumps at both ends that result in spectral ringing. All other spectral windows taper at both ends to lessen this effect by assigning smaller weights to samples close to the signal edges.

The windowing process always involves a compromise between conflicting aims: improving resolution and decreasing leakage.

- Resolution is the ability to know precisely how the signal energy is distributed in the frequency space. A spectrum analyzer with ideal resolution can distinguish two different tones (pure sinusoids) present in the signal, no matter how close in frequency. Quantitatively, this ability relates to the mainlobe width of the transform of the window.
- Leakage is the fact that, in a finite signal, every frequency component projects energy content throughout the complete frequency span. The amount of leakage in a spectrum can be measured by the ability to detect a weak tone from noise in the presence of a neighboring strong tone. Quantitatively, this ability relates to the sidelobe level of the frequency transform of the window.

The better the resolution, the higher the leakage, and vice versa. At one end of the range, a rectangular window has the narrowest possible mainlobe and the highest sidelobes. This window can resolve closely spaced tones if they have similar energy content, but it fails to find the weaker one if they do not. At the other end, a window with high sidelobe suppression has a wide mainlobe in which close frequencies are smeared together.

**Signal Analyzer** uses Kaiser windows to carry out windowing. For Kaiser windows, the fraction of the signal energy captured by the mainlobe depends most importantly on an adjustable shape factor, $\beta$. The shape factor ranges from $\beta = 0$, which corresponds to a rectangular window, to $\beta = 40$, where a wide mainlobe captures essentially all the spectral energy representable in double precision. An intermediate value of $\beta \approx 6$ approximates a Hann window closely. To control $\beta$, use the **Leakage** slider on the **Spectrum** and **Spectrogram** tabs. If you set the leakage to $\ell$ using the slider, then $\ell$ and $\beta$ are related by $\beta = 40(1 - \ell)$. See `kaiser` for more details.

## Parameter and Algorithm Selection

To compute the spectra of the signals appearing on a given display, **Signal Analyzer** initially determines the resolution bandwidth, which measures how close two tones can be and still be resolved. The resolution bandwidth has a theoretical value of

$$\mathrm{RBW}_{\mathrm{theory}} = \frac{\mathrm{ENBW}}{t_{\max} - t_{\min}}.$$

- $t_{\max} - t_{\min}$, the record length, is the time-domain duration of the selected signal region.

  Use the panner to select and adjust the record length or region of interest. Equivalently, you can zoom in on the time-domain plot or change the limits on the **Time** tab.
- ENBW is the equivalent noise bandwidth of the spectral window. See `enbw` for more details.

  Use the **Leakage** slider in the **Spectrum** tab to control the ENBW. The minimum value in the slider range corresponds to a Kaiser window with $\beta = 40$. The maximum value corresponds to a Kaiser window with $\beta = 0$.

In practice, however, the app might lower the resolution. Lowering the resolution makes it possible to compute the spectrum in a reasonable amount of time and to display it with a finite number of pixels. For these practical reasons, the lowest resolution bandwidth the app can use is

$$\mathrm{RBW}_{\mathrm{performance}} = \frac{f_{\mathrm{span}}}{4096 - 1},$$

where $f_{\mathrm{span}}$ is the width of the frequency range specified by setting **Frequency Limits** values on the **Spectrum** tab. If you do not specify a frequency range, the app uses as $f_{\mathrm{span}}$ the maximum sample rate among all the signals in the display. $\mathrm{RBW}_{\mathrm{performance}}$ cannot be adjusted.

To compute the spectrum of a signal, the app chooses the larger of the two values:

$$\mathrm{RBW} = \max(\mathrm{RBW}_{\mathrm{theory}}, \mathrm{RBW}_{\mathrm{performance}}).$$

This target resolution bandwidth is displayed on the **Spectrum** tab.

- If the resolution bandwidth is $\mathrm{RBW}_{\mathrm{theory}}$, then **Signal Analyzer** computes a single modified periodogram for the whole signal. The app uses a Kaiser window with the slider-controlled shape factor and applies zero-padding when the time limits on the axes exceed the signal duration. See `periodogram` for more details.
- If the resolution bandwidth is $\mathrm{RBW}_{\mathrm{performance}}$, then **Signal Analyzer** computes a Welch periodogram for the signal. The app:

  1. Divides the signals into overlapping segments.
  2. Windows each segment separately using a Kaiser window with the specified shape factor.
  3. Averages the periodograms of all the segments.

  Welch's procedure is designed to reduce the variance of the spectrum estimate by averaging different "realizations" of the signals, given by the overlapping sections, and using the window to remove redundant data. See `pwelch` for more details.

  - The length of each segment (or, equivalently, of the window) is computed using

$$\text{Segment length} = \frac{\max(f_{\mathrm{Nyquist}}) \times \mathrm{ENBW}}{\mathrm{RBW}},$$

where $\max(f_{\text{Nyquist}})$ is the highest Nyquist frequency among all the signals in the display. (If there is no aliasing, the Nyquist frequency is one-half the sample rate.)

- The stride length is found by adjusting an initial estimate,

$$\text{Stride length} \equiv \text{Segment length} - \text{Overlap} = \frac{\text{Segment length}}{2 \times \text{ENBW} - 1},$$

so that the first window starts exactly on the first sample of the first segment and the last window ends exactly on the last sample of the last segment.

## Zooming

If you zoom in on a region of a signal spectrum using one of the zoom actions on the **Display** tab, the app does not change the resolution bandwidth. Instead, **Signal Analyzer** performs an optical zooming, using bandlimited interpolation to display a smooth spectral curve.

Zooming in on a time-domain region of a signal is equivalent to setting the record length or region of interest with the panner.

If the selected time interval extends beyond the ends of a signal, the app zero-pads the signal. If a signal has no samples within the selected time interval, the app displays nothing.

## References

[1] harris, fredric j. “On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform.” *Proceedings of the IEEE*. Vol. 66, January 1978, pp. 51–83.

[2] Welch, Peter D. “The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms.” *IEEE Transactions on Audio and Electroacoustics*. Vol. 15, June 1967, pp. 70–73.

## See Also

**Apps**
**Signal Analyzer**

**Functions**
`enbw` | `kaiser` | `periodogram` | `pspectrum` | `pwelch`

## Related Examples

- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Modulation and Demodulation Using Complex Envelope” on page 21-61
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Extract Voices from Music Signal” on page 21-78
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96

## More About

- “Edit Sample Rate and Other Time Information” on page 21-103
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Nonparametric Methods” on page 7-8

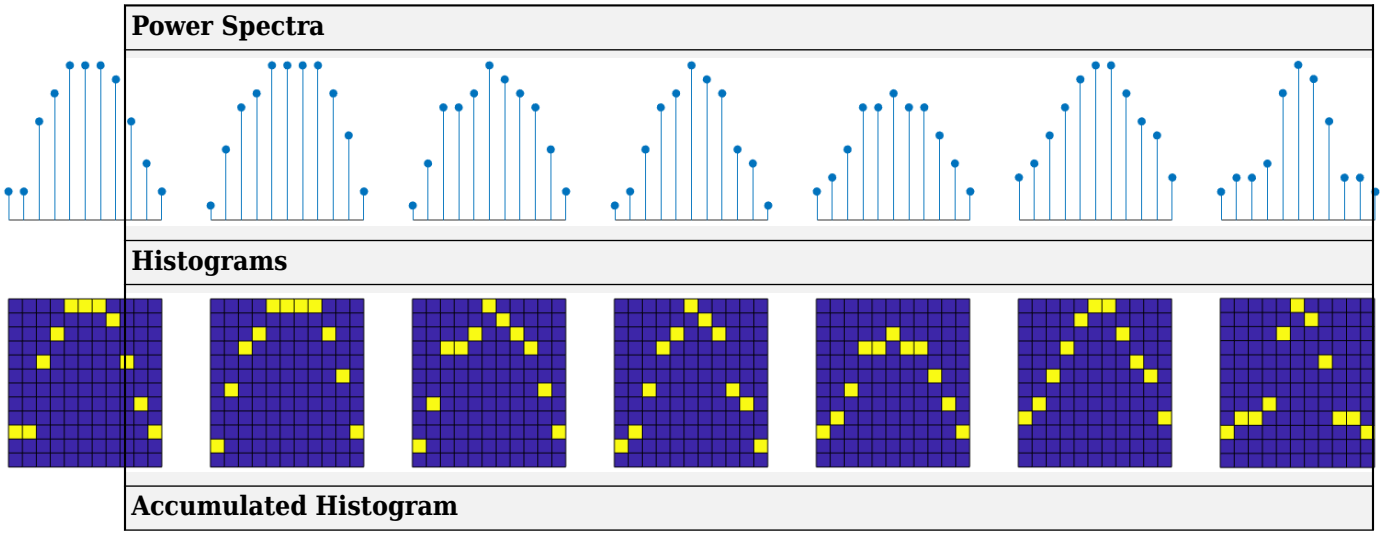
# Persistence Spectrum in Signal Analyzer

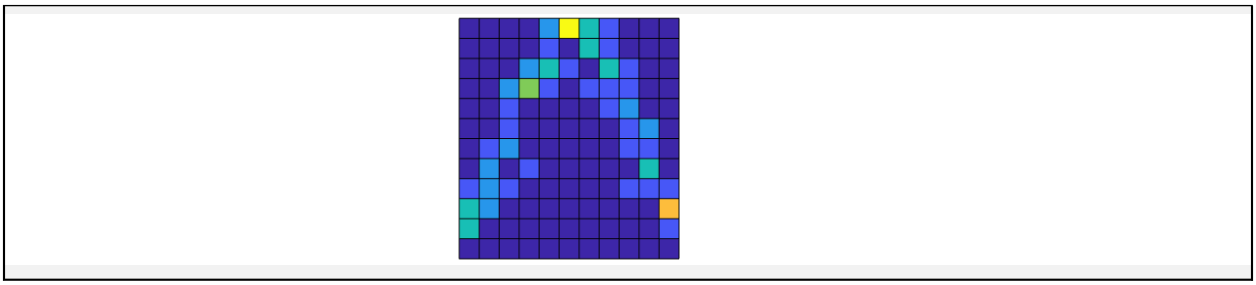
The persistence spectrum of a signal is a time-frequency view that shows the percentage of the time that a given frequency is present in a signal. The persistence spectrum is a histogram in power-frequency space. The longer a particular frequency persists in a signal as the signal evolves, the higher its time percentage and thus the brighter or "hotter" its color in the display. Use the persistence spectrum to identify signals hidden in other signals.

To compute the persistence spectrum, **Signal Analyzer** performs these steps:

1. Compute the spectrogram using the specified leakage, time resolution, and overlap. (When you zoom in time using the panner or a zoom button, the app computes and displays the persistence spectrum using the segments that fall within the visible zoomed-in region of interest, including those segments that are only partially visible. See “Spectrogram Computation in Signal Analyzer” on page 21-115 for more details.)
2. Partition the power and frequency values into 2-D bins. (To adjust the power or the frequency limits, enter the minimum and maximum **Power Limits** or **Frequency Limits** values on the **Persistence Spectrum** tab.)
3. For each time value, compute a bivariate histogram of the logarithm of the power spectrum. For every power-frequency bin where there is signal energy at that instant, increase the corresponding matrix element by 1. Sum the histograms for all the time values.
4. Plot the accumulated histogram against the power and the frequency, with the color proportional to the logarithm of the histogram counts expressed as normalized percentages. To represent zero values, use one-half of the smallest possible magnitude.

   (To adjust the range of histogram counts represented in the colormap, enter the minimum and maximum **Density Limits** on the **Persistence Spectrum** tab. To fit the colormap to the current density limits, click the **Fit colormap** button on the **Display** tab.)





## See Also

**Apps**
**Signal Analyzer**

**Functions**
`histcounts2` | `pspectrum`

## Related Examples

- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96

## More About

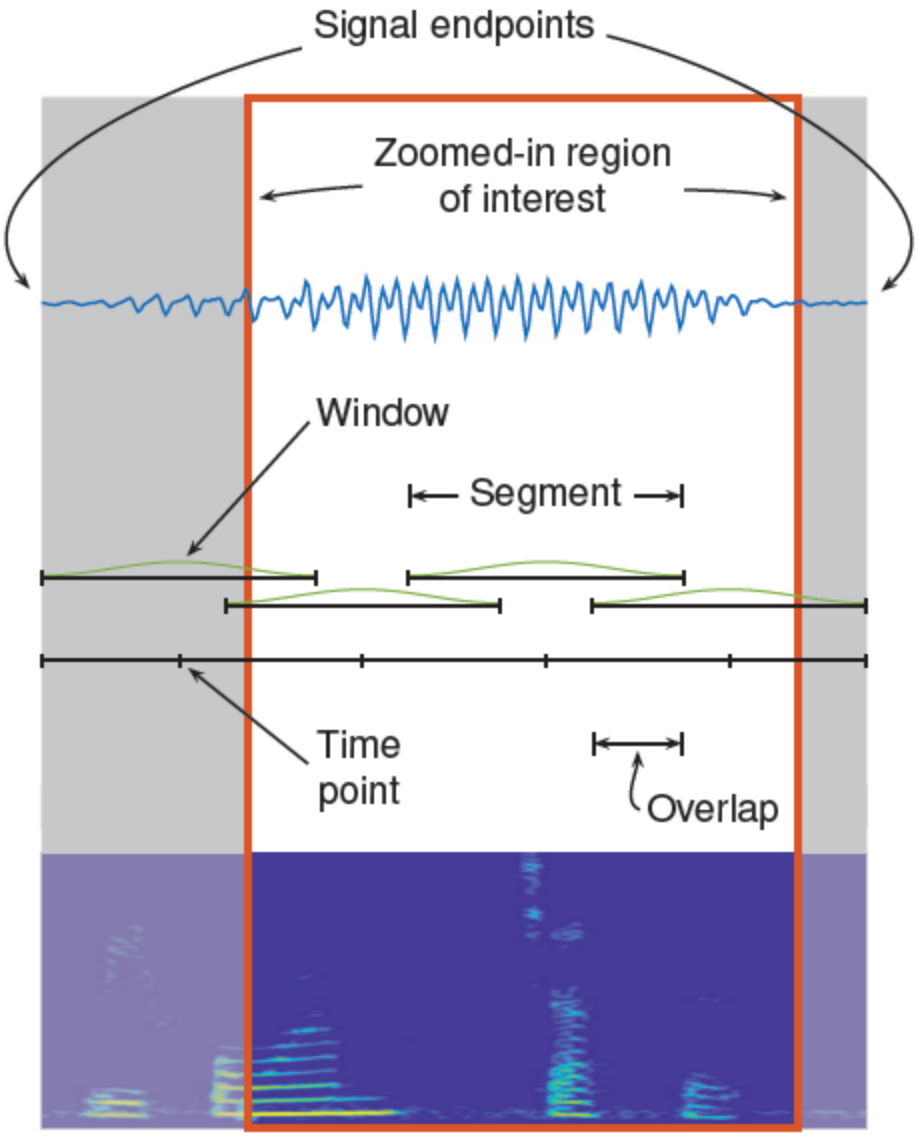
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Spectrogram Computation in Signal Analyzer” on page 21-115
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124

# Spectrogram Computation in Signal Analyzer

A nonstationary signal is a signal whose frequency content changes with time. The spectrogram of a nonstationary signal is an estimate of the time evolution of its frequency content. To construct the spectrogram of a nonstationary signal, **Signal Analyzer** follows these steps:

1. Divide the signal into equal-length segments. The segments must be short enough that the frequency content of the signal does not change appreciably within a segment. The segments may or may not overlap.
2. Window each segment and compute its spectrum to get the short-time Fourier transform.
3. Display segment-by-segment the power of each spectrum in decibels. Depict the magnitudes side-by-side as an image with magnitude-dependent colormap.

The spectrogram view is available in displays that contain only one signal.



## Divide Signal into Segments

To construct a spectrogram, first divide the signal into possibly overlapping segments. In **Signal Analyzer**, you can control the length of the segments and the amount of overlap between adjoining segments using **Time Resolution** and **Overlap**. If you do not specify the length and overlap, **Signal Analyzer** chooses a length based on the entire length of the signal, and 50% overlap. The app aligns the time axis of the spectrogram with the axis of the time-domain plot.

### Specified Time Resolution

On the **Spectrogram** tab, in the **Time Resolution** section, click **Specify**.

- If the signal does not have time information, specify the time resolution (segment length) in samples. The time resolution must be an integer greater than or equal to 1 and smaller than or equal to the signal length.

  If the signal has time information, specify the time resolution in seconds. The app converts the result into a number of samples and rounds it to the nearest integer that is less than or equal to the number but not smaller than 1. The time resolution must be smaller than or equal to the signal duration.
- Specify the overlap as a percentage of the segment length. The app converts the result into a number of samples and rounds it to the nearest integer that is less than or equal to the number.

### Default Time Resolution

If you select **Auto** for the time resolution computation, then **Signal Analyzer** uses the length of the entire signal to choose the length of the segments. The app sets the time resolution as $\lceil N/d \rceil$ samples, where the brackets denote the ceiling function, $N$ is the length of the signal, and $d$ is a divisor that depends on $N$:

| Signal Length ($N$) | Divisor ($d$) | Segment Length |
|---|---|---|
| 2 samples – 63 samples | 2 | 1 sample – 32 samples |
| 64 samples – 255 samples | 8 | 8 samples – 32 samples |
| 256 samples – 2047 samples | 8 | 32 samples – 256 samples |
| 2048 samples – 4095 samples | 16 | 128 samples – 256 samples |
| 4096 samples – 8191 samples | 32 | 128 samples – 256 samples |
| 8192 samples – 16383 samples | 64 | 128 samples – 256 samples |
| 16384 samples – $N$ samples | 128 | 128 samples – $\lceil N$ / 128$\rceil$ samples |

You can still specify the overlap between adjoining segments. Specifying the overlap changes the number of segments. Segments that extend beyond the signal endpoint are zero-padded.

Consider the seven-sample signal `[s0 s1 s2 s3 s4 s5 s6]`. Because $\lceil 7/2 \rceil = \lceil 3.5 \rceil = 4$, the app divides the signal into two segments of length four when there is no overlap. The number of segments changes as the overlap increases.

| Number of Overlapping Samples | Resulting Segments |
|---|---|
| 0 | `s0 s1 s2 s3`<br>`s4 s5 s6 0` |

| Number of Overlapping Samples | Resulting Segments |
|---|---|
| 1 | `s0 s1 s2 s3`<br>`  s3 s4 s5 s6` |
| 2 | `s0 s1 s2 s3`<br>`  s2 s3 s4 s5`<br>`  s4 s5 s6 0` |
| 3 | `s0 s1 s2 s3`<br>`  s1 s2 s3 s4`<br>`  s2 s3 s4 s5`<br>`  s3 s4 s5 s6` |

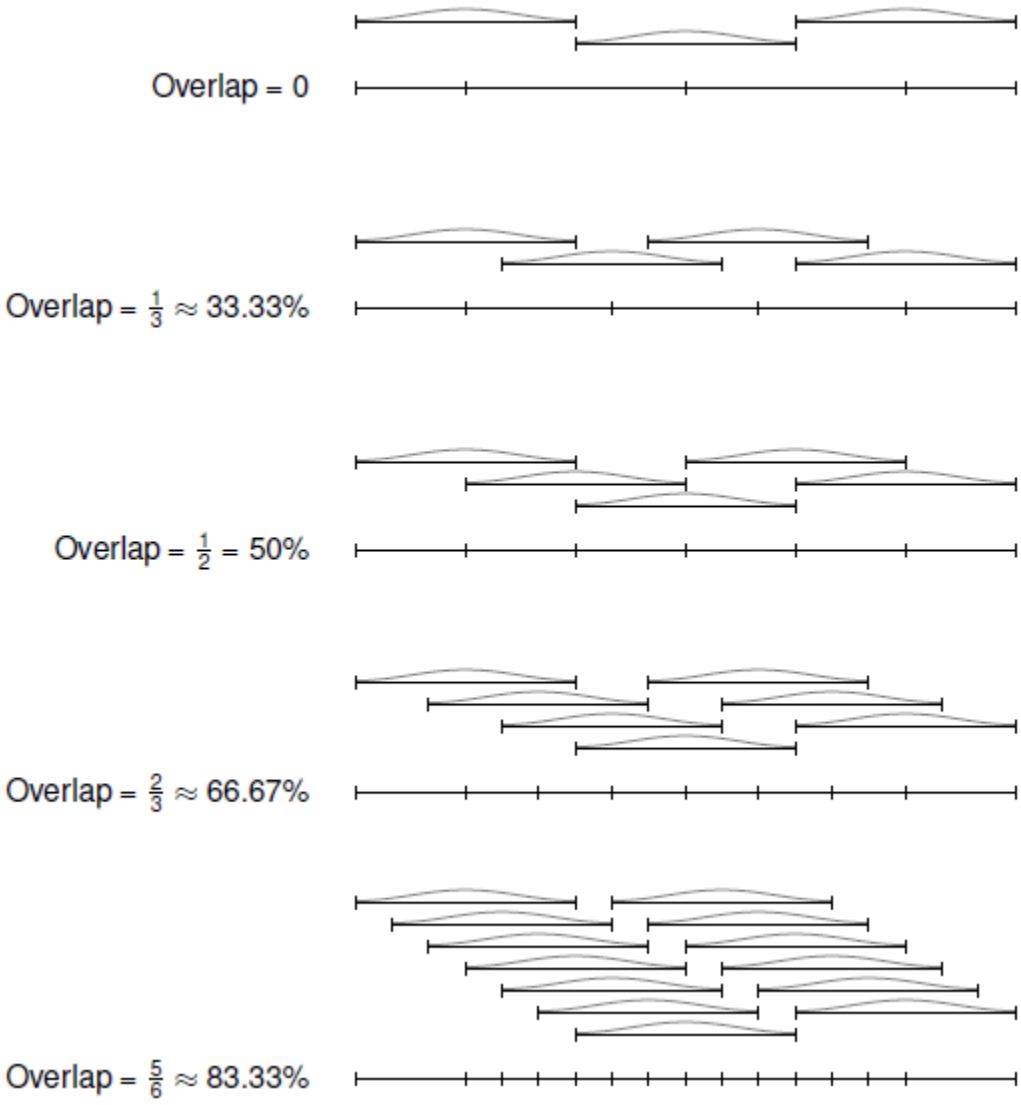
### Time Alignment

Once the segment length and overlap are set, the number of segments and their edge locations stay fixed and are independent of any zooming or panning. When you zoom and pan, the app computes and displays the spectrogram using the segments that fall within the visible zoomed-in region of interest.

The app:

- Aligns the time axis of the spectrogram with the axis of the corresponding time-domain plot. That way, the spectral content at a given time aligns with its occurrence.
- For nonzero overlap, extends the first and last segments to the signal endpoints.
- Zero-pads the signal if the last segment extends beyond the signal endpoint.

When the segments have 0% overlap, each segment is centered at the actual time of occurrence. When the overlap is nonzero, the alignment of the spectrogram time axis with the time-domain axis has the effect that the first and last time intervals are elongated. All other time intervals are of the same length. In other words, the center of each segment, except for the first and last, corresponds to the actual time of occurrence. Consider this example:




## Window the Segments and Compute Spectra

After **Signal Analyzer** divides the signal into overlapping segments, the app windows each segment with a Kaiser window. The shape factor $\beta$ of the window, and therefore the leakage, is adjustable.

**Note** The leakage used to compute the signal spectrum and the leakage used to window the spectrogram segments are independent of each other. You can adjust them separately.

The app then computes the spectrum of each segment, following the procedure outlined in “Spectrum Computation in Signal Analyzer” on page 21-109, except that the lower limit of the resolution bandwidth is

$$\mathrm{RBW}_{\mathrm{performance}} = \frac{f_{\mathrm{span}}}{1024 - 1}.$$

In summary, **Signal Analyzer** finds a compromise between the spectral resolution achievable with the entire length of the segment and the performance limitations that result from computing large FFTs.

- If the resolution resulting from analyzing the full segment is achievable, the app computes a single modified periodogram of the whole segment using a Kaiser window with the specified shape factor.
- If the resolution resulting from analyzing the full segment is not achievable, the app computes a Welch periodogram: It divides the segment into overlapping subsegments, windows each subsegment, and averages the periodograms of the subsegments. The app chooses the subsegment size, the window, and the overlap so that the composite periodogram is equivalent to a modified periodogram of the whole segment with the specified Kaiser window.

## Display Spectrum Power

The app displays the power of the short-time Fourier transform in decibels, using a color bar with the default MATLAB colormap. The color bar comprises the full power range of the spectrogram and does not change if you zoom or pan.

You can change the magnitude levels represented by a given color range. On the **Spectrogram** tab, change the minimum and maximum power values to display. You can also set the colormap so that it comprises the full power range of the zoomed-in section of the spectrogram. On the **Display** tab, click the scale color button .

## See Also

**Apps**
**Signal Analyzer**

**Functions**
`pspectrum` | `spectrogram` | `xspectrogram`

## Related Examples

- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96

## More About

- “Edit Sample Rate and Other Time Information” on page 21-103
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Scalogram Computation in Signal Analyzer” on page 21-120
- “Nonparametric Methods” on page 7-8

# Scalogram Computation in Signal Analyzer

The scalogram is the absolute value of the continuous wavelet transform (CWT) of a signal, plotted as a function of time and frequency. The scalogram can be more useful than the spectrogram for analyzing real-world signals with features occurring at different scales — for example, signals with slowly varying events punctuated by abrupt transients. Use the scalogram when you want better time localization for short-duration, high-frequency events, and better frequency localization for low-frequency, longer-duration events.

**Note** You need a Wavelet Toolbox license to use the scalogram view.

The spectrogram is obtained by windowing the input signal with a *window* of constant length (duration) that is shifted in time and frequency. (See “Spectrogram Computation in Signal Analyzer” on page 21-115 for more information.) The window used in the spectrogram is even, real-valued, and does not oscillate. Because the spectrogram uses a constant window, the time-frequency resolution of the spectrogram is fixed.

By contrast, the CWT is obtained by windowing the signal with a wavelet that is scaled and shifted in time. The wavelet oscillates and can be complex-valued. The scaling and shifting operations are applied to a prototype wavelet. The scaling used in the CWT both shrinks and stretches the prototype wavelet. Shrinking the prototype wavelet yields short duration, high-frequency wavelets that are good at detecting transient events. Stretching the prototype wavelet yields long duration, low-frequency wavelets which are good at isolating long-duration, low frequency events.

To compute the scalogram, **Signal Analyzer** performs these steps:

1. If the signal has more than 1 million samples, divide the signal into overlapping segments.
2. Compute the CWT of each segment to get its scalogram.
3. Display the scalogram segment by segment.

As implemented, the CWT uses $L^1$ normalization. Therefore, the amplitudes of the oscillatory components in a signal agree with the amplitudes of the corresponding wavelet coefficients.

**Tip**

- Scalogram view does not support complex signals.
- Scalogram view does not support nonuniformly sampled signals. To compute the scalogram of a nonuniformly sampled signal, resample your signal to a uniform grid by using the `resample` function.
- Scalogram view is available in displays that contain only one signal. To compare scalograms of different signals, open separate displays and drag each signal to its own display.

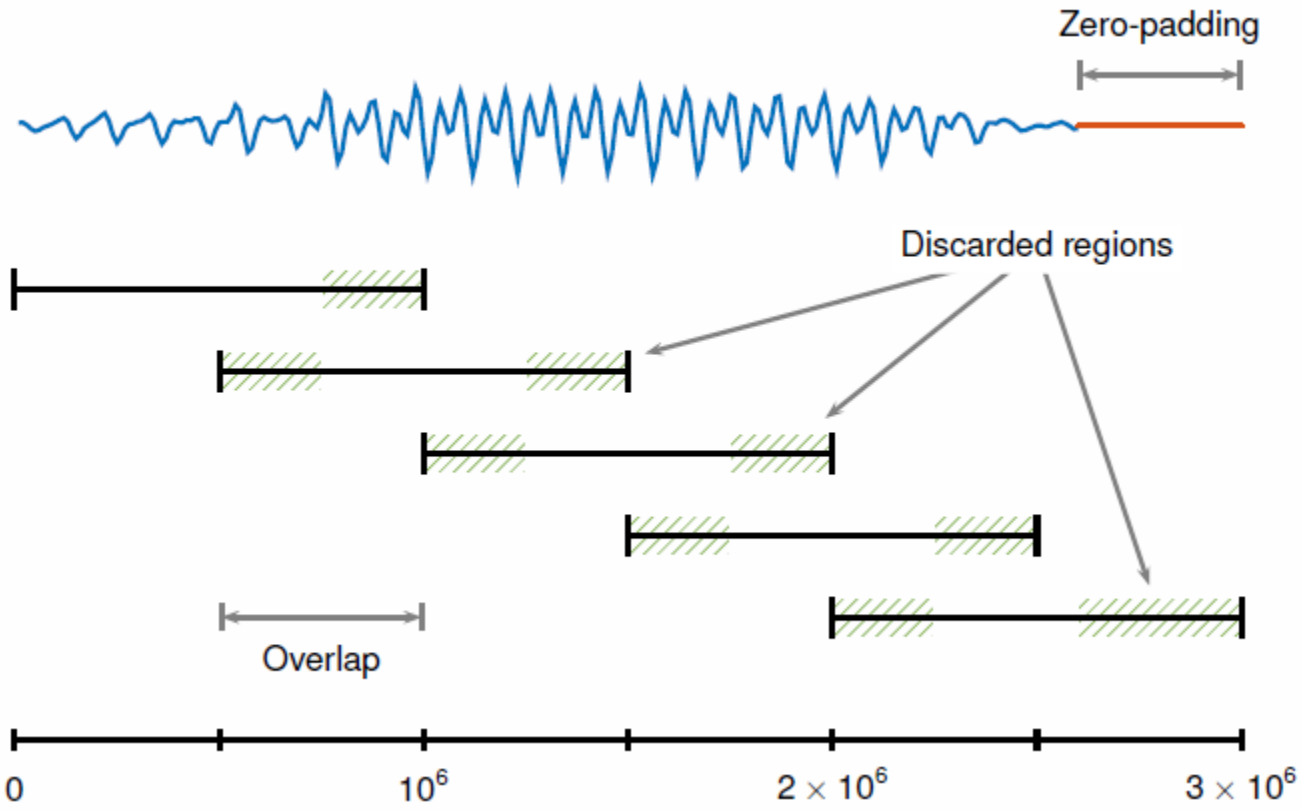
## Divide the Signal into Segments

If the input signal has 1 million samples or less, **Signal Analyzer** uses the `cwt` function directly. If the signal has more than 1 million samples, the app performs these steps:

1. Divide the signal into segments of 1 million samples, with 50% overlap between adjoining segments.

2. If the last segment extends beyond the signal endpoint, zero-pad the signal until the last segment has 1 million samples.
3. After computing the scalogram of each segment, remove edge effects:
   - Discard the first 250,000 and the last 250,000 scalogram samples of all segments except the first and the last.
   - Discard the last 250,000 scalogram samples of the first segment.
   - In the last segment, discard the first 250,000 scalogram samples and the samples corresponding to the zero-padded region.

Consider, for example, a signal with $2.6 \times 10^6$ samples:



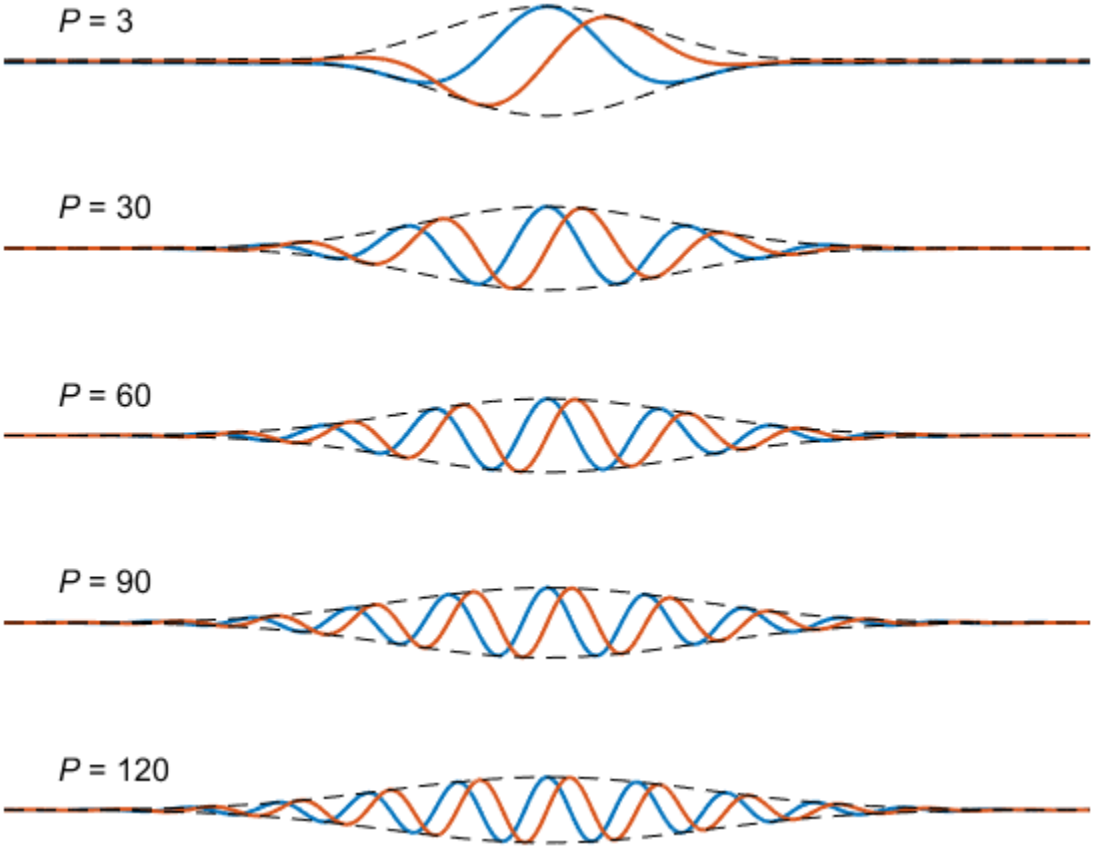

## Compute the Continuous Wavelet Transform

**Signal Analyzer** computes the CWT using the default settings of the `cwt` function. The app uses generalized analytic Morse wavelets with gamma factor $\gamma = 3$. See “Morse Wavelets” (Wavelet Toolbox) for more information.

**Signal Analyzer** provides two separate controls for frequency resolution.

- The **Time-Bandwidth** slider controls the time-bandwidth product, which is proportional to the wavelet duration in the time domain. Increasing the time-bandwidth product results in wavelets with more oscillations in their central portions, larger spreads in time, and narrower spreads in frequency. The slider moves in the range from 3 to 120. The default value is 60. The figure shows

some Morse wavelets with varying time-bandwidth product *P*. The real part is in blue, the imaginary part is in red, and the absolute value is in black.




- The **Voices Per Octave** slider controls the number of scales per octave used to discretize the CWT. As the number of voices per octave increases, the scale resolution becomes finer. The slider moves in steps of multiples of 4 in the range from 4 to 16. The default value is 8.

## Display the Scalogram

**Signal Analyzer** plots the absolute value of the CWT coefficients as a function of time and frequency. If the signal was divided into segments, the app concatenates portions of the scalograms of the individual segments and displays them. The app also plots the cone of influence, which shows where edge effects become significant. See “Boundary Effects and the Cone of Influence” (Wavelet Toolbox) for more information.

## See Also

**Apps**
**Signal Analyzer**

**Functions**
`cwt` | `cwtfilterbank` | `pspectrum`

## Related Examples

- “Spectrogram View of Dial Tone Signal” on page 21-48
- “Find Interference Using Persistence Spectrum” on page 21-51
- “Find and Track Ridges Using Reassigned Spectrogram” on page 21-70
- “Scalogram of Hyperbolic Chirp” on page 21-75
- “Compute Envelope Spectrum of Vibration Signal” on page 21-96

## More About

- “Continuous and Discrete Wavelet Transforms” (Wavelet Toolbox)
- “Spectrum Computation in Signal Analyzer” on page 21-109
- “Persistence Spectrum in Signal Analyzer” on page 21-113
- “Spectrogram Computation in Signal Analyzer” on page 21-115

# Keyboard Shortcuts for Signal Analyzer

You can use several keyboard shortcuts to facilitate working with the **Signal Analyzer** app.

**Note** On Macintosh platforms, use the **Command** key instead of **Ctrl**.

## General Actions

| Task | Shortcut |
|---|---|
| Start a new session | **Ctrl+N** |
| Open a session | **Ctrl+O** |
| Save a session | **Ctrl+S** |
| Link or unlink a display | **Ctrl+U** |
| Delete a signal | **Del** |
| Copy a display to the clipboard | **Ctrl+C** |

## Multichannel Signals

| Task | Shortcut |
|---|---|
| Expand signal hierarchy | **Ctrl+Shift+=** |
| Collapse signal hierarchy | **Ctrl+=** |

## Zooming

| Task | Shortcut |
|---|---|
| Zoom in X-axis (time or frequency) | **Ctrl+Shift+T** |
| Zoom in Y-axis | **Ctrl+Shift+Y** |
| Zoom in X and Y | **Ctrl++** (numeric keypad only) |
| Zoom out | **Ctrl+-** (numeric keypad only) |
| Fit to view | **Spacebar** |
| Fit colormap to current power limits | **Ctrl+Spacebar** |
| Cancel zoom operation or signal dragging | **Esc** |

## Data Cursors

| Task | Shortcut |
|---|---|
| Show a data cursor | **Ctrl+I** |
| Hide all data cursors | **Shift+Del** |
| Move a selected data cursor to the next data point | **Right arrow** |

| Task | Shortcut |
| --- | --- |
| Move a selected data cursor to the previous data point | **Left arrow** |
| Activate first (left) cursor | **Ctrl+1** |
| Activate second (right) cursor | **Ctrl+2** |

## See Also

**Signal Analyzer**

## Related Examples

- “Analyze Signals with Inherent Time Information” on page 21-45
- “Extract Voices from Music Signal” on page 21-78
- “Declip Saturated Signals Using Your Own Function” on page 21-91
- “Extract Regions of Interest from Whale Song” on page 21-56

## More About

- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Signal Analyzer Tips and Limitations” on page 21-126
- “Customize Signal Analyzer” on page 21-130

# Signal Analyzer Tips and Limitations

Frequently asked questions and current limitations of the **Signal Analyzer** app.

## Select Signals to Analyze

1. "I dragged a 512-by-24 matrix into a display, but the app plotted only 10 of the 24 signals. How do I plot the others?"

   By default, **Signal Analyzer** imports all the columns of a multichannel signal but plots only the first 10 columns. To plot signal columns beyond the 10th, drag them to the display. Alternatively, on the Signal table, select the check boxes next to the names of the signals you want to plot.

2. "My data is saved in structures. How can I analyze them in **Signal Analyzer**?"

   To study a structure in **Signal Analyzer**, convert it to a timetable. The easiest way to do the conversion is to convert the structure to a table and then convert the table to a timetable. The second step involves converting the time variable to a duration array. The following example creates a structure with three fields, one of them containing the time values, and converts the structure to a timetable readable by **Signal Analyzer**.

   ```
   str.st = (0:999)'/1000;
   str.s1 = randn(1000,1);
   str.s2 = sin(2*pi*20*str.st);

   T = struct2table(str);
   T.st = seconds(T.st);
   TT = table2timetable(T,'RowTimes','st');
   ```

   If your structure does not have time information, you can use other MATLAB functions. The following function takes a structure as input, extracts from it the arrays of signal values, and calls **Signal Analyzer** to plot the signals.

   ```
   function structSig(x)
      names = fieldnames(x);
      for i = 1:length(names)
         signalAnalyzer(getfield(x,names{i}))
      end
   end
   ```

3. "What does it mean when a row in the Signal table is highlighted in gray and what does the check box mean?"

   There are two different ways to choose signals in the Signal table. Each way gives you access to a different set of operations.

   - Selecting the signal by clicking the **Name**, **Info**, **Time**, or **Start Time** column in the Signal table enables you to perform all the operations in the **Analyzer** tab. You can change the time information and smooth, filter, or duplicate the signals. You can run preprocessing operations on a signal without plotting the signal.
   - Selecting the check box to the left of the signal name plots the signal in the currently selected display and enables you to perform all the operations in the **Display** tab. You can display the signal in the frequency domain or the time-frequency domain, or you can measure the signal using cursors.

4 "I use timetables with the time values stored as `datetime` arrays. How can I analyze them?"

To analyze timetables with time values stored as a `datetime` array, convert the array to a relative `duration` array by subtracting the first element from all the others. The following example creates a timetable with `datetime` row times and converts it to a timetable readable by **Signal Analyzer**.

```
tt = timetable(datetime(2016,11,9,2,30,1:10)',randn(10,1));
dt = tt.Time-tt.Time(1);
tn = timetable(dt,tt.Var1);
```

See “Analyze Signals with Inherent Time Information” on page 21-45 for another example.

5 "I have a timetable but only some of its variables were imported. How can I import them?"

**Signal Analyzer** lists only the variables that it can display and process. If some variables of a timetable are not being imported, they probably are complex or have NaNs. To be able to import them to the app, you must fix them in MATLAB first. To fix timetables, you can use the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

6 "I changed a variable in the MATLAB workspace. Why is there no change in the **Signal Analyzer** display?"

If you modify a signal in the MATLAB workspace, the Workspace browser updates automatically. To have the app recognize the changes, reimport the signal by dragging it again to the Signal table or to a display.

## Preprocess Signals

1 "How do I apply a lowpass filter to a signal that is not uniformly sampled?"

The filtering functionality of **Signal Analyzer** supports only uniformly sampled signals. You can resample your signal to a uniform grid by using **Signal Analyzer**'s resampling functionality, which you can find in the **Preprocessing** gallery on the **Analyzer** tab. Alternatively, you can use the Signal Processing Toolbox `resample` function.

2 "How do I know what parameters were used for a preprocessing operation?"

To see a full summary of the preprocessing steps you took, including all settings you chose, click **Generate Function** on the **Analyzer** tab.

## Explore Signals

1 "I want to view a scalogram of my signal, but I get a warning saying that I have to create a uniformly sampled signal. How do I resample my signal?"

You can resample your signal to a uniform grid by using **Signal Analyzer**'s resampling functionality, which you can find in the **Preprocessing** gallery on the **Analyzer** tab. Alternatively, you can use the Signal Processing Toolbox `resample` function.

2 "Why can I not zoom out beyond the Nyquist range of a scalogram?"

If a real signal is sampled properly, then all of its frequency information is contained within the Nyquist range.

3 "How can I compare spectrograms of 10 different signals?"

The time-frequency views of **Signal Analyzer** support only one signal per display. To compare spectrograms of 10 different signals, open 10 displays and drag each signal to its own display. You can use the same procedure for persistence spectra and scalograms.

## Share or Reuse Analysis

1. "I generated a script that does not run because the variable it uses does not exist. Why?"

   If you extract, duplicate, or rename a signal in **Signal Analyzer** and generate a MATLAB script without exporting the modified signal, the script will throw an error because the variables do not exist in the MATLAB workspace. Remember to export any signals used by generated scripts.

2. "How do I reproduce a **Signal Analyzer** spectrum, persistence spectrum, spectrogram, or scalogram in MATLAB?"

   Click **Spectrum** or **Spectrogram** on the **Display** tab to compute and display the spectrum, persistence spectrum, spectrogram, or scalogram of a plotted signal. When you have the optimal settings for your signal, click **Generate Script** and select `Spectrum Script`, `Persistence Spectrum Script`, `Spectrogram Script`, or `Scalogram Script` to generate a script that you can use in MATLAB.

3. "How can I automate computation using Signal Analyzer generated MATLAB scripts and functions?"

   **Signal Analyzer** can generate MATLAB functions that reproduce any combination of preprocessing steps performed on a signal. The app can also generate MATLAB scripts for extracting regions of interest or for computing the spectrum, spectrogram, persistence spectrum, or scalogram of a signal. You can combine scripts and functions to automate your analysis. See “Compute Envelope Spectrum of Vibration Signal” on page 21-96 for an example.

## Troubleshooting

1. "I cannot get **Signal Analyzer** to start."

   - **Signal Analyzer** can fail to start if MATLAB is using a software implementation of OpenGL®. To solve the problem, upgrade your graphics hardware driver or use `opengl` to switch to a hardware-accelerated implementation of OpenGL. See “Resolving Low-Level Graphics Issues” for more information.
   - Attempting to start **Signal Analyzer** can cause JavaScript® support for WebGL™ to fail. To solve the problem, update your graphics hardware driver.
   - **Signal Analyzer** can fail to start due to a network error. Check your organization's proxy settings and, if possible, disable the proxy that is interfering with the app startup process.

2. "When I start **Signal Analyzer**, I get an error saying it is unable write the file `temp_signalAnalyzer_datarepository.mat`."

   You cannot launch **Signal Analyzer** from multiple instances of MATLAB running on the same computer.

## See Also

**Signal Analyzer**

## Related Examples

- “Plot Signals from the Command Line” on page 21-35
- “Resolve Tones by Varying Window Leakage” on page 21-39
- “Analyze Signals with Inherent Time Information” on page 21-45
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91

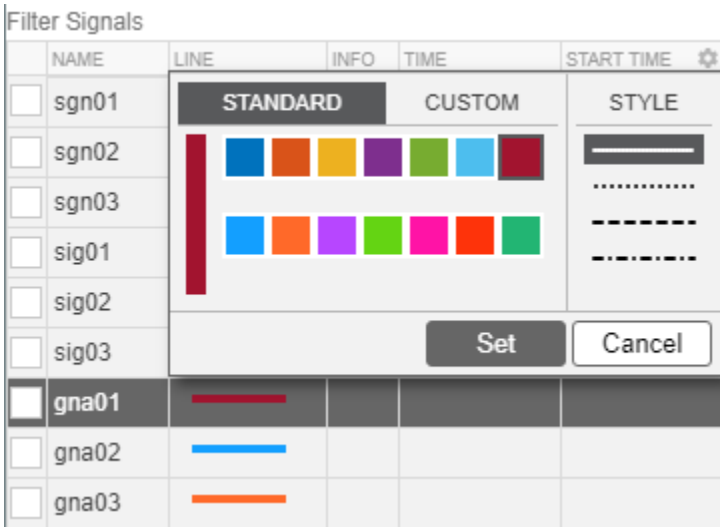
## More About

- “Edit Sample Rate and Other Time Information” on page 21-103
- “Data Types Supported by Signal Analyzer” on page 21-106
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Customize Signal Analyzer” on page 21-130

# Customize Signal Analyzer

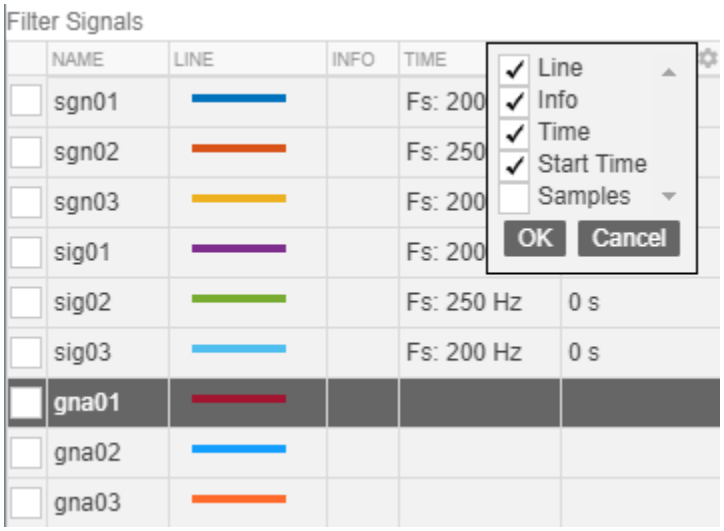
## Specify Line Color and Style

To specify the line style and color, click in the **Line** column of a signal. If the line column is not shown, add the column using the column selector button. Select a color from the palette and a line style. Click **Custom** to choose custom colors for your signals. You can specify custom colors as RGB triplets or as hexadecimal codes. For complex signals, the color that you set corresponds to the real part. The color of the imaginary part has the same hue and saturation with a different value of luminosity.




## Add or Remove Columns in the Signal Table

Columns in the Signal table display the plot configuration and signal properties. To add or remove a column, click the column selector button. From the list, select the columns that you want to display and click **OK**. After you select a column, the new column is added to the table in the order that it appears in the column selection list.




## Modify Signal Analyzer Displays

| Goal | Action |
|---|---|
| Hide the Workspace browser or the Signal table to enlarge the display area. | On the **Analyzer** tab, click one of the layout buttons.<br> |
| Zoom and pan to inspect the data. | On the **Display** tab, select one of the zoom actions.<br> <br>Alternatively, activate the panner by clicking **Panner** . |
| Fit spectrogram, scalogram, or persistence spectrum colormap to current power or density limits. | On the **Display** tab, click the **Fit Colormap** button . |

<table>
<tr><th>Goal</th><th>Action</th></tr>
<tr><td>Set the minimum and maximum values of the plot axes.</td><td>On the Time, Spectrum, Persistence Spectrum, Spectrogram, or Scalogram tab, enter the axes limit values. You can also change the minimum and maximum time values on the Display tab.<br>
<br>When setting axes for a display, you can specify time or frequency units before specifying limit values. Several engineering units are available:<br><table><tr><th>Quantity</th><th>Units</th></tr><tr><td>Time</td><td>ps, ns, µs, ms, seconds, minutes, hours, days, years</td></tr><tr><td>Frequency</td><td>cycles/year, cycles/day, cycles/hour, cycles/minute, mHz, Hz, kHz, MHz, GHz, THz</td></tr></table><br>Note Selecting different time or frequency units for axes limit values does not change any plots.</td></tr>
<tr><td>Show or hide legends identifying plotted signals.</td><td>On the Display tab, click Legend .<br><br>Each display gets its own legend. The legends appear either at the top of the display or to the right of the display.<br><br>For each signal on a display, the legend shows the signal name and signal color. For complex signals, the first signal color represents the real part, and the second signal color represents the imaginary part.</td></tr>
<tr><td>Link or unlink a display.</td><td>Select a display. On the Display tab, select Link Time. Link Time is enabled only when there are two or more displays and at least one signal contains time information.<br><br>To unlink a display, select it and clear Link Time.<br><br>Frequency axes are never linked between displays.</td></tr>
</table>

| Goal | Action |
| --- | --- |
| Normalize the data for each signal from 0 to 1 along the *y*-axis of a time plot. | On the **Time** tab, select **Normalize Y Axis**. |
| Show markers at each sample point in a time plot of a signal. | On the **Time** tab, select **Show Markers**. |

## Signal Analyzer Preferences

- If you export or save signals that have time information but are not stored as timetables, the time information by default is not saved. If you want to preserve the time information by saving the signals as timetables, on the **Analyzer** tab, click **Preferences** and check `Always use timetables when signals have time information`.
- If you generate scripts involving signals that have time information but are not stored as timetables, the time information by default is not saved. If you want to preserve the time information by generating scripts that treat signals as timetables, on the **Analyzer** tab, click **Preferences** and check `Always use timetables when signals have time information`.

## See Also

**Apps**
**Signal Analyzer**

## Related Examples

- “Analyze Signals with Inherent Time Information” on page 21-45
- “Resample and Filter a Nonuniformly Sampled Signal” on page 21-84
- “Declip Saturated Signals Using Your Own Function” on page 21-91

## More About

- “Edit Sample Rate and Other Time Information” on page 21-103
- “Keyboard Shortcuts for Signal Analyzer” on page 21-124
- “Signal Analyzer Tips and Limitations” on page 21-126

# Signal Labeler


- “Using Signal Labeler App” on page 22-2
- “Import Data into Signal Labeler” on page 22-6
- “Create or Import Signal Label Definitions” on page 22-13
- “Label Signals Interactively or Automatically” on page 22-17
- “Custom Labeling Functions” on page 22-25
- “Customize Labeling View” on page 22-30
- “Export Labeled Signal Sets and Signal Label Definitions” on page 22-36
- “Signal Labeler Usage Tips” on page 22-38
- “Label Signal Attributes, Regions of Interest, and Points” on page 22-40
- “Examine Labeled Signal Set” on page 22-46
- “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51
- “Label Spoken Words in Audio Signals Using External API” on page 22-60
- “Choose an App to Label Ground Truth Data” on page 22-65

# Using Signal Labeler App

## App Workflow

A typical workflow for labeling signals using the **Signal Labeler** app is:

1. "Import Data into Signal Labeler" on page 22-6 — Select any real signal available in the MATLAB Workspace. The app accepts numeric arrays, MATLAB timetables, and `labeledSignalSet` objects. Read data from files or use `signalDatastore` objects as input.
2. "Create or Import Signal Label Definitions" on page 22-13 — Define labels to annotate signal attributes, regions, or points of interest quickly and consistently using logical, categorical, numerical, or string values. You can also import signal label definitions stored in MAT-files.
3. "Label Signals Interactively or Automatically" on page 22-17 — Label signals interactively. Automatically label signal peaks or use your own "Custom Labeling Functions" on page 22-25. Label several signals at once or use the autolabeling mode of the app to inspect labeling results before committing them.
4. "Customize Labeling View" on page 22-30 — Use spectrum and spectrogram to aid labeling and show or hide the label viewer.
5. "Export Labeled Signal Sets and Signal Label Definitions" on page 22-36 — Export labeled signal sets and label signal definitions to the MATLAB Workspace or to MAT-files.

## Example: Label Points and Regions of Interest in Signal

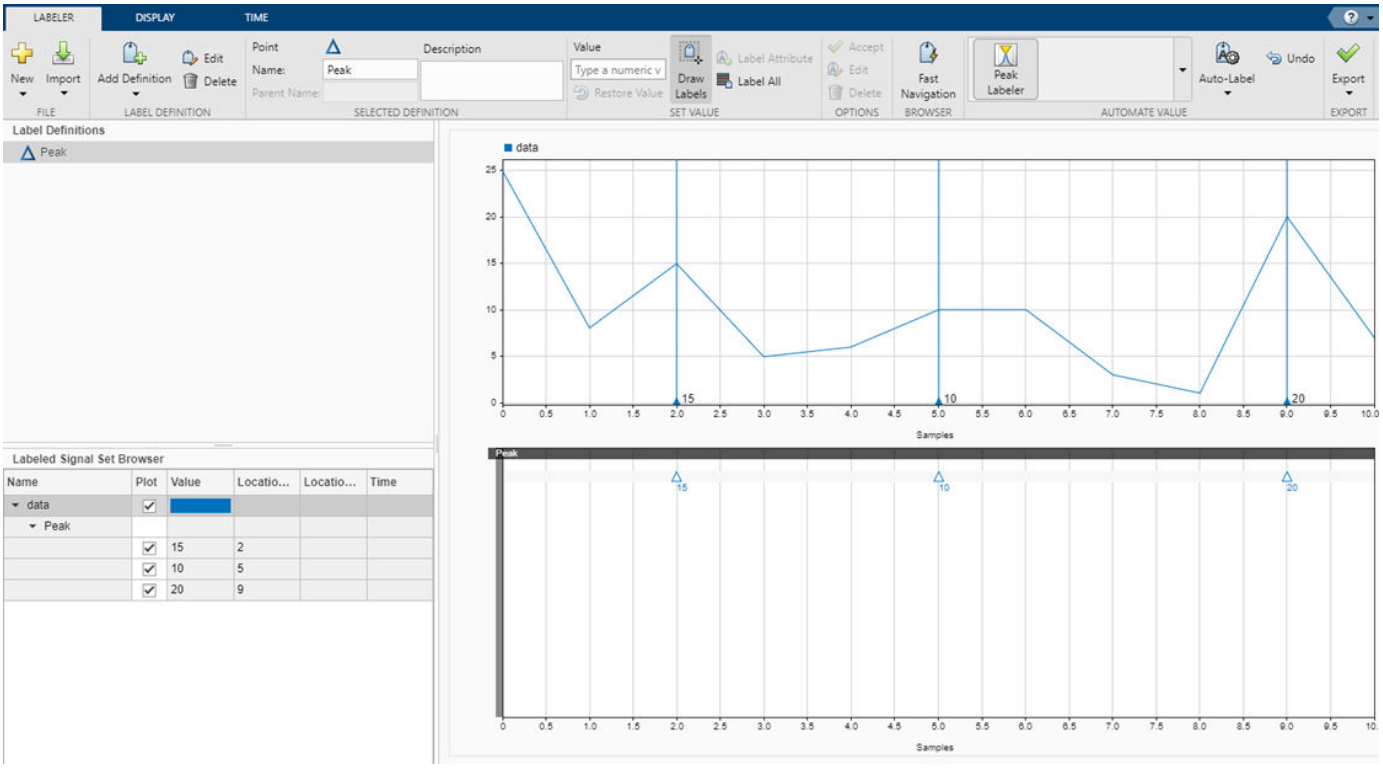
Define a vector with two acute peaks and one flat peak. Use **Signal Labeler** to label the peaks and mark the flat peak as different from the others.

```
data = [25 8 15 5 6 10 10 3 1 20 7];
```

Open Signal Labeler. Import the data vector. On the **Labeler** tab, click **Import ▼**, select `Members from workspace`, select the `data` signal in the dialog box that appears, click **Import**, and close the dialog box. Select the check box next to the signal name in the **Labeled Signal Set Browser** to display the signal in the time plot.

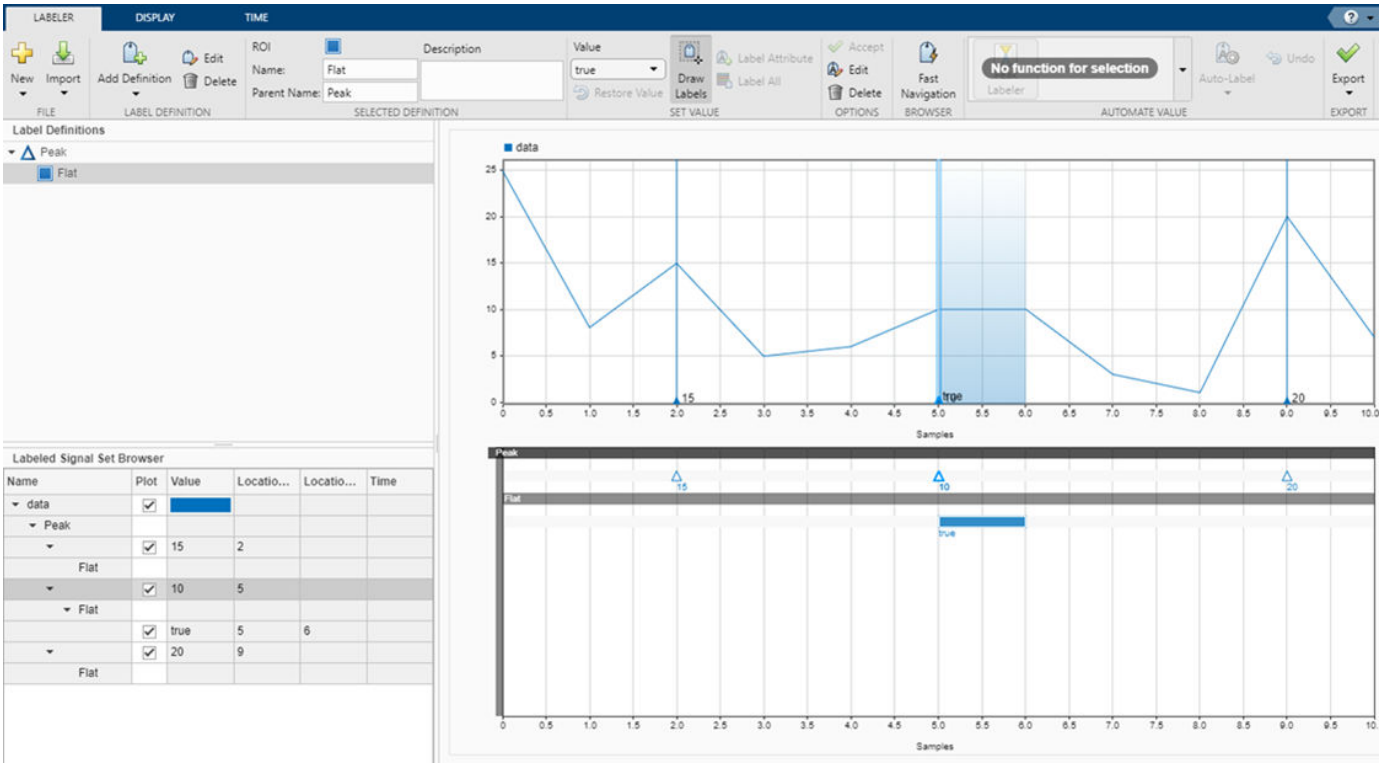
Label the signal peaks. Start by creating a signal label definition. Click **Add Definition ▼**. In the dialog box, specify **Label Name** as `Peak`, **Label Type** as `Point`, and **Data Type** as `numeric`.

With the `Peak` definition highlighted in the **Label Definitions** browser, select **Peak Labeler** in the **Automate Value** gallery. Click **Auto-Label ▼** and select `Auto-Label Signals`. Click **OK** in the dialog box that appears. **Signal Labeler** labels the three peaks and annotates its locations.




Create a sublabel for `Peak` to annotate the flat peak, which is the second of the three. With `Peak` still selected in the **Label Definitions** browser, click **Add Definition ▼** and select `Add sublabel definition`. Specify **Label Name** as `Flat`, **Label Type** as `ROI`, and **Data Type** as `logical`.

Select the point label for the flat peak. You can select the label by clicking it in the **Labeled Signal Set Browser**, in the time plot, or in the label viewer. Selecting the label highlights it in all three places. Select `Flat` in the **Label Definitions** browser. A shaded region appears on the signal plot. Move and resize the region until it encloses the flat peak. To accept a label, click the **Accept** check mark next to the **Label** button, press **Enter**, or double-click the shaded region.




Export the labeled signal. Click **Export ▼** and select `Labeled Signal Set To File`. Name the file `peaks.mat`. Click **Export**.

Inspect the labeled signal set you created. Load `peaks.mat` into the MATLAB® Workspace. The `labeledSignalSet` object is called `ls`. Verify that the data source is the vector you created at the beginning. Inspect the signal and subsignal label definitions.

```
load peaks
src = getSignal(ls,1)'
```

```
src = 1×11

    25     8    15     5     6    10    10     3     1    20     7
```

```
lbl = getLabelDefinitions(ls)
```

```
lbl =
  signalLabelDefinition with properties:

                     Name: "Peak"
                LabelType: "point"
            LabelDataType: "numeric"
       ValidationFunction: []
    PointLocationsDataType: "double"
             DefaultValue: []
                Sublabels: [1x1 signalLabelDefinition]
                      Tag: ""
              Description: ""

 Use labeledSignalSet to create a labeled signal set.
```

## See Also

**Apps**
**Signal Analyzer** | **Signal Labeler**

**Functions**
`labeledSignalSet` | `signalLabelDefinition`

## Related Examples

- “Label Signal Attributes, Regions of Interest, and Points” on page 22-40
- “Examine Labeled Signal Set” on page 22-46
- “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51
- “Label Spoken Words in Audio Signals Using External API” on page 22-60

## More About

- “Import Data into Signal Labeler” on page 22-6
- “Create or Import Signal Label Definitions” on page 22-13
- “Label Signals Interactively or Automatically” on page 22-17
- “Custom Labeling Functions” on page 22-25
- “Customize Labeling View” on page 22-30
- “Export Labeled Signal Sets and Signal Label Definitions” on page 22-36
- “Signal Labeler Usage Tips” on page 22-38

# Import Data into Signal Labeler

## Supported Signal Types

The **Signal Labeler** app works with real-valued vectors, matrices, MATLAB timetables, and `labeledSignalSet` objects. The app also supports MAT-files and CSV files.

---

**Note** **Signal Labeler** does not support complex-valued signals, signals with `Inf` or `NaN` values, multidimensional arrays, or sparse matrices.

---

- **Example: Numeric Arrays**

  ```
  num = cos(pi./[4;2]*(0:159))'+randn(160,2);
  ```

  specifies a two-channel signal consisting of sinusoids embedded in white noise. The signal does not contain time information unless you specify it. In **Signal Labeler**, you can import the signal in samples, or you can add time information when you import it.
- **Example: MATLAB Timetables**

  ```
  tt1 = timetable(num,'SampleRate',100);
  tt2 = timetable(seconds((0:159)'/100),num);
  ```

  both specify that the noisy two-channel sinusoid is sampled at 100 Hz. For more information, see the `timetable` documentation.
- **Example: `labeledSignalSet` Objects**

  ```
  lss = labeledSignalSet(num);
  ```

  specifies that the noisy sinusoid is in samples.
- **Example: `labeledSignalSet` Objects with Time Information**

  ```
  lst1 = labeledSignalSet(num,'SampleRate',100);
  lst2 = labeledSignalSet(timetable(seconds((0:159)'/100),num));
  ```

  both specify that the noisy sinusoid is sampled at 100 Hz.
- **Example: Multisignal Members**

  ```
  msn = labeledSignalSet({randn(10,3),randn(17,9)});
  ```

  has two members. The first member contains three 10-sample signals. The second member contains nine 17-sample signals.

  ```
  mst = labeledSignalSet({{timetable(seconds(1:10)',randn(10,3))}, ...
      {timetable(seconds(1:7)',randn(7,2)), ...
      timetable(randn(30,1),'SampleRate',100)}});
  ```

  has two members. The first member contains three signals sampled at 1 Hz for 10 seconds. The second member contains two signals sampled at 1 Hz for 7 seconds and one 30-sample signal sampled at 100 Hz.
- **Example: `signalDatastore` Object Pointing to Files**

  Specify the path to a set of sample sound signals included as MAT-files with MATLAB®. Each file contains a signal variable and a sample rate. List the names of the files.

```
folder = fullfile(matlabroot,"toolbox","matlab","audiovideo");
lst = dir(append(folder,"/*.mat"));
nms = {lst(:).name}'
```

```
nms = 7x1 cell
    {'chirp.mat'   }
    {'gong.mat'    }
    {'handel.mat'  }
    {'laughter.mat'}
    {'mtlb.mat'    }
    {'splat.mat'   }
    {'train.mat'   }
```

Create a signal datastore that points to the specified folder. Set the sample rate variable name to `Fs`, which is common to all files. Generate a subset of the datastore that excludes the file `mtlb.mat`, which differs from the other files in that the signal variable is not called `y`.

```
sds = signalDatastore(folder,"SampleRateVariableName","Fs");
sdss = subset(sds,~strcmp(nms,"mtlb.mat"));
```

Use the subset datastore as the source for a `labeledSignalSet` object.

```
lss = labeledSignalSet(sdss)
```

```
lss =
  labeledSignalSet with properties:

             Source: [1x1 signalDatastore]
         NumMembers: 6
    TimeInformation: "inherent"
             Labels: [6x0 table]
        Description: ""

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

## Specify Time Information

The signals you import into **Signal Labeler** can be labeled in samples or in time. This specification stays fixed to ensure consistent labeling. You cannot mix signals in samples and signals with time information in the same session.

When specifying the time information for a set of signals that do not have it, select a time specification option in the import dialog box.

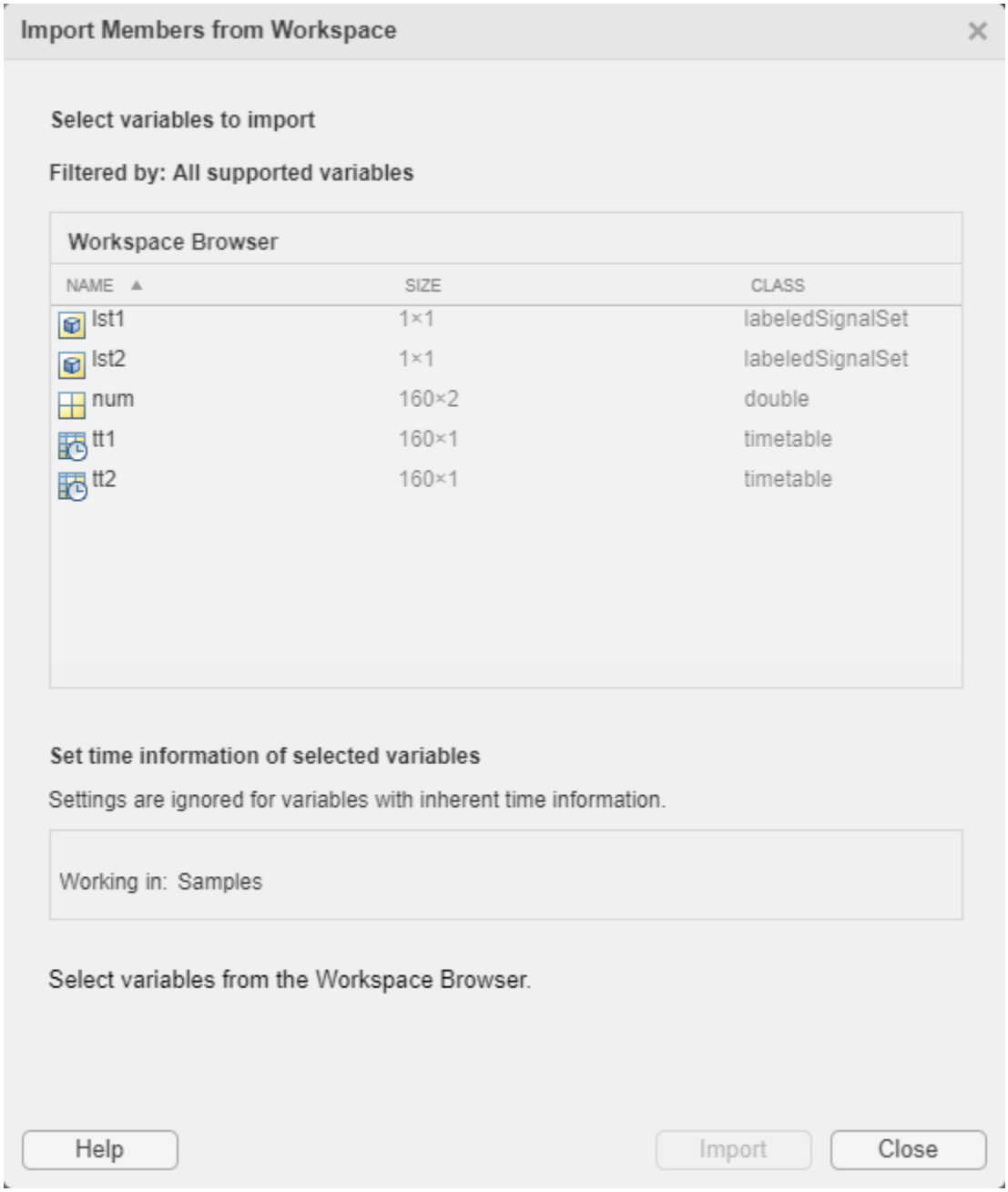
| Time Specification Option | Description |
|---|---|
| `Samples` | This option enables you to explore and label signals without the need to specify a sample rate or a sample time. It is equivalent to plotting the signal in MATLAB without *x*-axis information. |

| Time Specification Option | Description |
|---|---|
| `Sample Rate` | Use this option when you know the rate at which the signal has been sampled. The sample rate can be expressed in Hz, kHz, MHz, or GHz. To specify the sample rate, you can use a numeric value, the name of a scalar variable in the MATLAB Workspace, or any valid MATLAB expression. Set the sample rate so that the members are plotted in units of time. |
| `Sample Rate Variable From File` | Use this option when the sample rate is saved as a variable in the file being imported. |
| `Sample Time` | Use this option when you know the time interval between samples. The sample time can be expressed in seconds, years, days, hours, minutes, milliseconds, microseconds, or nanoseconds. To specify the sample time, you can use a numeric value, the name of a scalar variable in the MATLAB Workspace, or any valid MATLAB expression. Set the sample time so that the members are plotted in units of time. |
| `Sample Time Variable From File` | Use this option when the sample time is saved as a variable in the file being imported. |
| `Time Values` | Use this option when you know the time value corresponding to each sample. Specify the time values using a valid MATLAB expression or the name of a variable in the MATLAB Workspace. The time values can be stored in a numeric or `duration` vector with real time values expressed in seconds. The values must be unique and cannot be `NaN`, but need not be uniformly spaced. The time array must have the same length as the members.<br><br>In all cases, the app derives a sample rate from the time values and displays it in the **Time** column of the **Labeled Signal Set Browser**. An asterisk preceding the sample rate indicates that the members are nonuniformly sampled. |
| `Time Values Variable From File` | Use this option when the time values are saved as a variable in the file being imported. |

Once a signal or set of signals has been imported into **Signal Labeler**, the chosen time specification stays fixed throughout the labeling session.

## Import Signals from the MATLAB Workspace

To import signals to **Signal Labeler** from the MATLAB Workspace, on the **Labeler** tab, click **Import** ▼ and select `From workspace` in the **Members** list. In the dialog box, select the signals you want to import.




- If you initially imported a numeric array and specified it in samples, or if you initially imported a `labeledSignalSet` object in samples, you can subsequently choose only signals in samples. If you choose a numeric array, **Signal Labeler** imports it and treats it in samples.
- If you initially imported a numeric array and specified its time information, or if you initially imported a MATLAB timetable or a `labeledSignalSet` object with time information, you can subsequently choose only signals with time information. If you choose a numeric array, you must set its time information when importing it.

**Note** You cannot modify the time information of a `labeledSignalSet` object from within **Signal Labeler**. If the labeled signal set has no time information, the app treats its members as being in samples. If the labeled signal set has time information, the app incorporates this information when it imports the signals. For more information, see the `labeledSignalSet` documentation.

To be imported successfully, labeled signal sets must obey these additional rules:

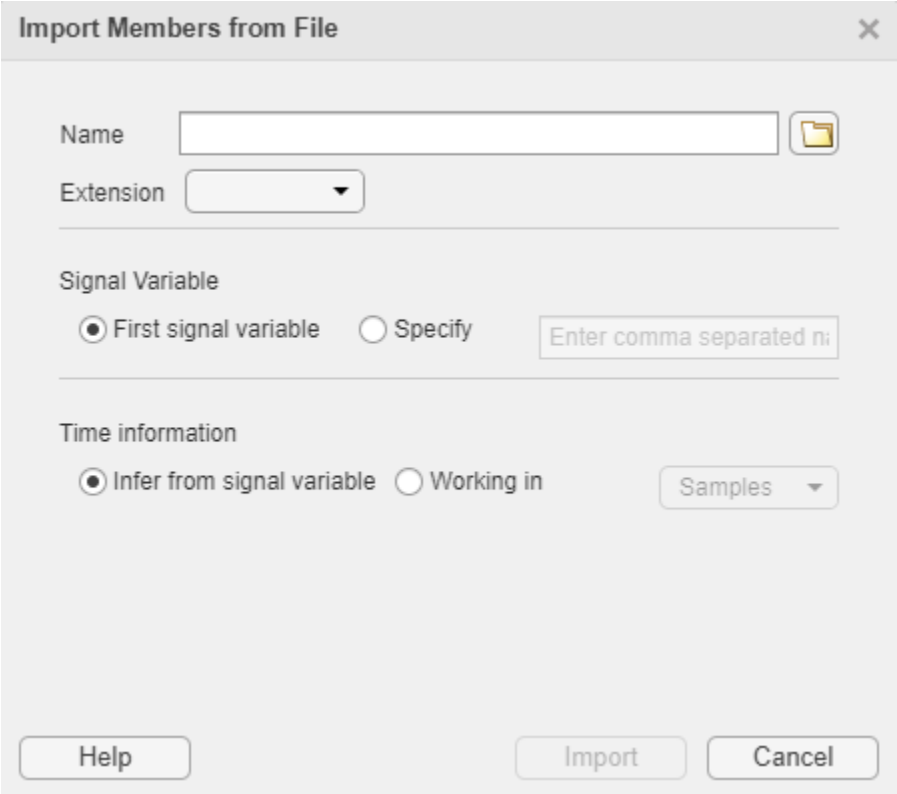
- If the selection includes two or more labeled signal sets, the labeled signal sets must have unique signal label definitions. If two or more sets share a label definition, the definition must have the same type and data type for all sets. For more information, see “Create or Import Signal Label Definitions” on page 22-13.
- If the selection includes two or more labeled signal sets, the labeled signal sets must have unique member names. You cannot change member names from within **Signal Labeler**. To change the name of a member of a labeled signal set, use `setMemberNames` at the command line.
- If you select two or more `labeledSignalSet` objects for labeling, **Signal Labeler** merges them and creates a single labeled signal set containing all the members and label values of the input sets. This action is equivalent to using `merge` at the command line.

## Import Signals from Files

To import signals to **Signal Labeler** from files, on the **Labeler** tab, click **Import** ▼ and select `From files` in the **Members** list. In the dialog box, browse to select the signals you want to import.

**Note**

- **Signal Labeler** supports MAT-files and CSV files only. All values in a CSV file other than headers must be numeric.
- Importing `labeledSignalSet` objects from files is not supported. To import a `labeledSignalSet` object, load it into the MATLAB Workspace and import it from there.




- All members to import must have the same extension and the same variables.
- **Signal Labeler** does not support working simultaneously with in-memory data and data from files.
  - If you initially imported in-memory members from the MATLAB Workspace, the `From files` option is disabled from the **Import** menu of the **Labeler** tab.
  - If you initially imported data from files, the only workspace variables you can then import from the MATLAB Workspace are `labeledSignalSet` objects whose input data sources are `signalDatastore` objects pointing to files. For an example, see “Supported Signal Types” on page 22-6.

By default, **Signal Labeler** reads the first signal variable of each file. To determine the name of the first variable in a file, `signalDatastore` follows these steps:

- For MAT-files:

```
s = load(fileName);
varNames = fieldnames(s);
firstVar = s.(varNames{1});
```

- For CSV files:

```
opts = detectImportOptions(fileName,'PreserveVariableNames',true);
varNames = opts.VariableNames;
firstVar = string(varNames{1});
```

To specify the signal variables that you want to read, click **Specify** and enter a comma-separated list of signal variable names.

**Tip** If a CSV file does not have variable names specified in a header line, then the variables are called `Var1` for the first column, `Var2` for the second column, and so on.

## See Also

**Apps**
**Signal Analyzer** | **Signal Labeler**

**Functions**
`labeledSignalSet` | `signalLabelDefinition`

## Related Examples

- “Label Signal Attributes, Regions of Interest, and Points” on page 22-40
- “Examine Labeled Signal Set” on page 22-46
- “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51
- “Label Spoken Words in Audio Signals Using External API” on page 22-60

## More About

- “Using Signal Labeler App” on page 22-2
- “Create or Import Signal Label Definitions” on page 22-13
- “Label Signals Interactively or Automatically” on page 22-17
- “Custom Labeling Functions” on page 22-25
- “Customize Labeling View” on page 22-30
- “Export Labeled Signal Sets and Signal Label Definitions” on page 22-36
- “Signal Labeler Usage Tips” on page 22-38

# Create or Import Signal Label Definitions

Use signal label definitions to define labels for signals. Labels can be of three types:

- Attribute labels define signal characteristics.
- Region-of-interest (ROI) labels define signal characteristics over regions of interest.
- Point labels define signal characteristics at points of interest.

Each label can have one of four data types:

- Logical labels can be either `true` or `false`.
- Categorical labels can belong to any one of a set of categories that you specify.
- Numeric labels can have any numeric value.
- String labels can have any value represented by a string.

A label can have any number of sublabels. Sublabels themselves cannot have sublabels.

In **Signal Labeler**, you can import already existing `signalLabelDefinition` objects stored in MAT-files, or you can create signal label definitions directly in the app.

- **Example:** Create a definition to label a signal with its mean RMS value as a numeric attribute.
  - At the command line, the code

    ```
    lblRMS = signalLabelDefinition("MeanRMSattr", ...
    'LabelType','attribute','LabelDataType','numeric');
    save('MeanRMSdef','lblRMS')
    ```

    creates a file, `MeanRMSdef.mat`, that you can load into **Signal Labeler** to import a label definition called `MeanRMSattr`.
  - In **Signal Labeler**, click **Add Definition ▼** on the **Labeler** tab and select `Add label definition`. In the dialog box, specify **Label Name** as `MeanRMS`, **Label Type** as `Attribute`, and **Data Type** as `numeric`.
- **Example:** Create a definition to label the zero crossings of a signal as `"rising"` for positive-going transitions and `"falling"` for negative-going transitions.
  - At the command line, the code

    ```
    ldf = signalLabelDefinition("Crosses",'LabelType','point', ...
    'LabelDataType','categorical','Categories',["rising","falling"]);
    save('CrossDef','ldf')
    ```

    creates a file, `CrossDef.mat`, that you can load into **Signal Labeler** to import a label definition called `Crosses`.
  - In **Signal Labeler**, click **Add Definition ▼** on the **Labeler** tab and select `Add label definition`. In the dialog box, specify **Label Name** as `crossings`, **Label Type** as `Point`, **Data Type** as `categorical`, and categories as `rising` and `falling`, with each category on a new line.

## Import Signal Label Definitions

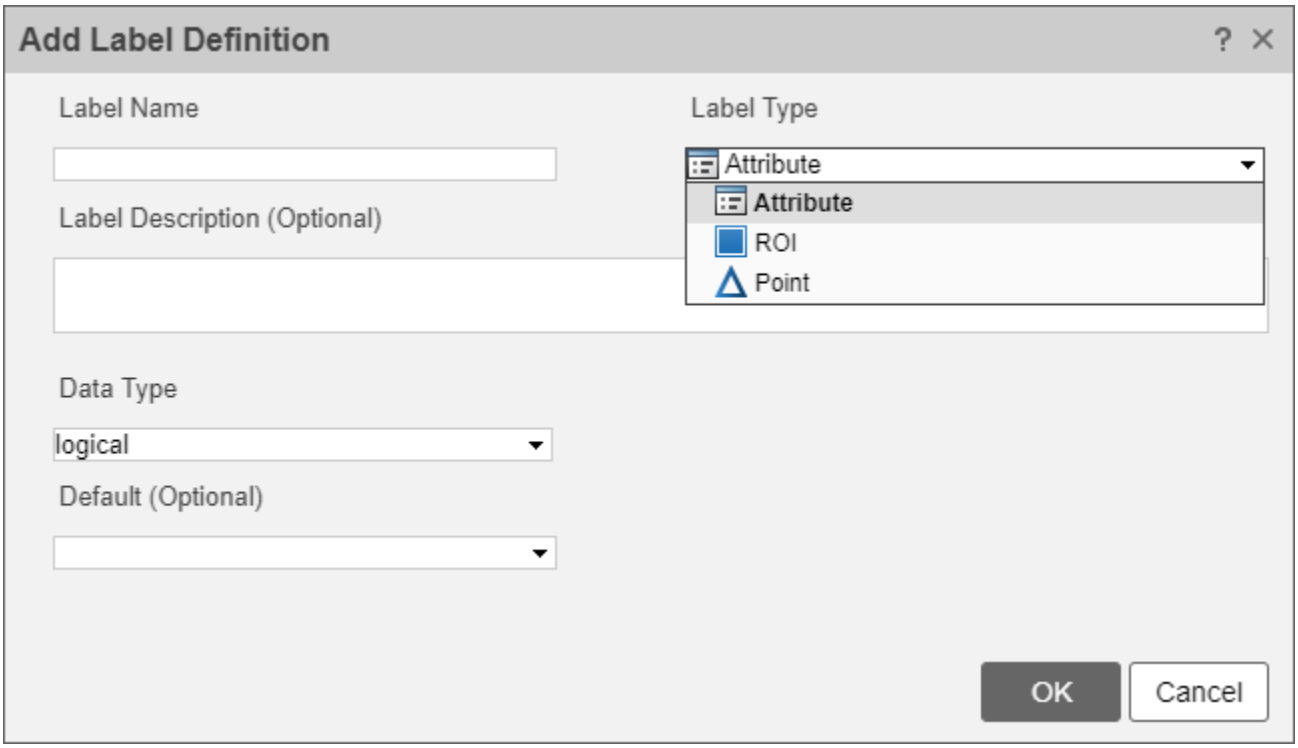
To import existing signal label definitions, click **Import ▼** on the **Labeler** tab and select `Label Definitions From file`. In the dialog box, specify the name of the MAT-file that contains the label

definitions you want to import. The MAT-file must contain only one vector of `signalLabelDefinition` objects.

## Create Label Definitions

To add a signal label definition to your labeled signal set, click **Add Definition ▼** on the **Labeler** tab and select `Add label definition`. In the dialog box, specify the following fields:

- **Label Name** — Specify the name in the text box.
- **Label Type** — Select one of `Attribute`, `ROI`, or `Point`.
- **Label Description** (optional) — Specify the description in the text box.
- **Data Type** — Select one of `string`, `numeric`, `logical` (the default), or `categorical`.
- **Categories** — This field appears if you specify **Data Type** as `categorical`. Enter each category on a new line.
- **Default** (optional) — Specify a default value for the signal label. For `logical` labels, select either `true` or `false`. For `categorical` labels, select any of the categories you specified.




This action is equivalent to using `addLabelDefinitions` at the command line.

**Note** If you want to reuse the signal label definitions that you created during a **Signal Labeler** session, you must export the definitions to a MAT-file and import them in a subsequent session.

## Create Sublabel Definitions

To add a sublabel definition, select the definition in the **Label Definitions** browser, click **Add Definition ▼** on the **Labeler** tab, and select `Add sublabel definition`. The top of the dialog box shows, as **Parent Name**, the name of the label to which you are adding the sublabel.

This action is equivalent to using `addLabelDefinitions` at the command line.

## Edit Label or Sublabel Definitions

To edit a label or sublabel definition, select the definition in the **Label Definitions** browser and click the **Edit** button. In the dialog box, specify the following fields:

- **Label Name** — Specify the value in the text box.
- **Label Description** — Specify the value in the text box.
- **Categories** — This field appears if you specify **Data Type** as `categorical`. You can add categories, but you cannot remove any existing categories. Enter each new category on a new line.
- **Default** — Specify a default value for the signal label. For `logical` labels, select either `true` or `false`. For `categorical` labels, select any of the categories you specified.

  Editing the default value does not affect existing labels. The new default value applies only to new members, new regions, or new points.

You cannot modify the **Label Type** or **Data Type** fields. To change the label type or the data type of a label definition, remove the definition and add a definition with the desired properties.

This action is equivalent to using `editLabelDefinition` at the command line.

## Delete Label or Sublabel Definitions

To delete a label or sublabel definition, select the definition in the **Label Definitions** browser and click the **Delete** button on the toolstrip.

This action is equivalent to using `removeLabelDefinition` at the command line.

## See Also

**Apps**
**Signal Analyzer** | **Signal Labeler**

**Objects**
`labeledSignalSet` | `signalLabelDefinition` | `signalMask`

## Related Examples

- “Label Signal Attributes, Regions of Interest, and Points” on page 22-40
- “Examine Labeled Signal Set” on page 22-46
- “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51

- “Label Spoken Words in Audio Signals Using External API” on page 22-60

## More About

- “Using Signal Labeler App” on page 22-2
- “Import Data into Signal Labeler” on page 22-6
- “Label Signals Interactively or Automatically” on page 22-17
- “Custom Labeling Functions” on page 22-25
- “Customize Labeling View” on page 22-30
- “Export Labeled Signal Sets and Signal Label Definitions” on page 22-36
- “Signal Labeler Usage Tips” on page 22-38

# Label Signals Interactively or Automatically

You can label signals with the **Signal Labeler** app in four different ways:

- "Bulk Manual Labeling" on page 22-17 — Apply a label to one or more signals by specifying its value and, when necessary, its location.
- "Interactive Manual Labeling" on page 22-18 — Plot one or more signals from the **Labeled Signal Set Browser** and interactively apply a label only to the plotted signals. Use "Fast Navigation Mode" on page 22-17 to speed up the labeling process.
- "Bulk Autolabeling" on page 22-19 — Automatically apply labels to one or more signals using **Peak Labeler** or "Custom Labeling Functions" on page 22-25.
- "Autolabeling Mode" on page 22-20 — Plot one or more signals from the **Labeled Signal Set Browser** and automatically label them using **Peak Labeler** or "Custom Labeling Functions" on page 22-25. Inspect the labeling, debug the autolabeling function, modify parameters, edit labels, and save the labeling when it is satisfactory.

## Fast Navigation Mode

To quickly navigate through channels or label members, click **Fast Navigation** in the **Browser** section of the toolstrip. Choose the channel that you want to plot in the **Signal Selection** section. The **Signal Selector** option is available when **Plot Selected** is active. In the dialog box, select the check box next to the channels you want to plot in each member. Select **Plot Previous** or **Plot Next** in the **Browser** section to navigate through members.

**Note** The **Signal Selection** window displays only as many channels as present in the member with the smallest number of channels.

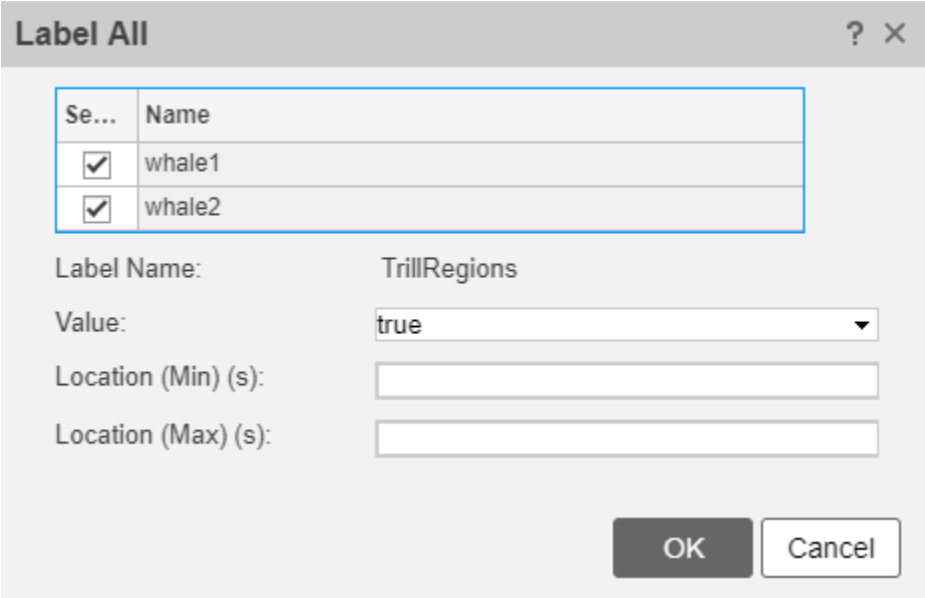
To apply a label to plotted signals in **Fast Navigation** mode, proceed as described in "Interactive Manual Labeling" on page 22-18.

## Label Signals Manually

### Bulk Manual Labeling

Sometimes an attribute applies to several signals, or an event occurs at the same time for several signals in your data set. To label a set of signals quickly, first select the label definition you want to apply in the **Label Definitions** browser. On the **Labeler** tab, in the **Set Value** section, specify the value that you want to assign to the label. Click **Label All** in the **Set Value** section. In the dialog box:

- Specify the signals you want to label by selecting the check boxes next to their names.
- Edit the label value if necessary.
- For region-of-interest (ROI) labels, specify the region endpoints using the **Location (Min)** and **Location (Max)** fields.
- For point labels, specify the point location using the **Location** field.




This action is equivalent to using `setLabelValue` at the command line.

**Note** Bulk manual labeling is not supported for sublabels.

### Interactive Manual Labeling

To label one or more signals interactively, first choose the signals that you want to plot by selecting the check boxes next to their names in the **Labeled Signal Set Browser**. Select the label definition you want to apply in the **Label Definitions** browser. **Draw Labels** is automatically activated when a signal and an ROI or point label are selected. Specify the label value in the **Set Value** section of the **Labeler** tab, or assign the value after the label is drawn. Click the time plot to add the label:

- For ROI labels, a thick animated dashed line appears that expands into a shaded region when you click and drag. (The animated frame indicates that the region is active.) Move and resize the active region until it encloses the ROI.
- For point labels, an animated dashed (active) line appears for the point being labeled. Move the active line until it crosses the signal at the point of your choice.

To accept a label, click the **Accept** check mark next to the **Label** button, press **Enter**, or double-click the active region or line. The app automatically accepts a label when the next label is drawn. A label is added to all channels within a member.

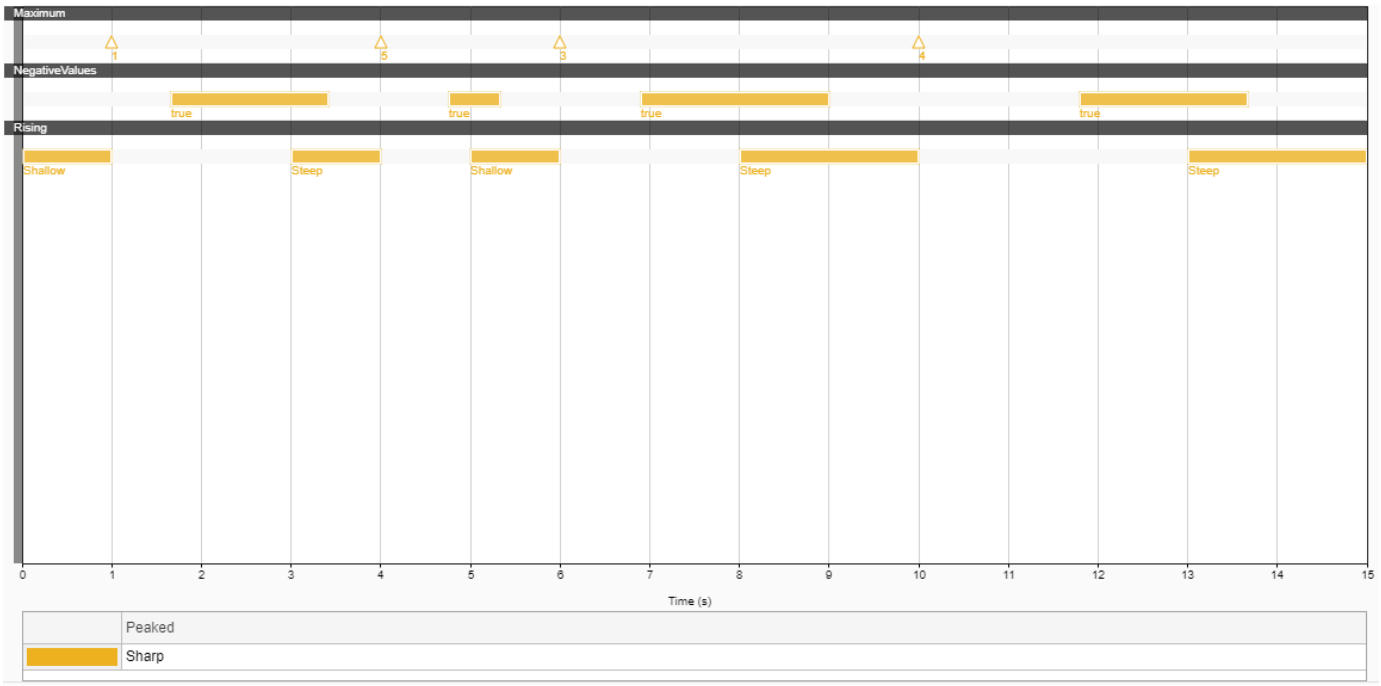
**Tip** To improve label placement, you can go to the **Display** tab and choose a zoom action or activate the panner.

For an example of interactive manual labeling, see “Label Signal Attributes, Regions of Interest, and Points” on page 22-40.

### Edit Labels

To edit an existing signal label, you can do any of these:

- In the **Labeled Signal Set Browser**, select the label, right-click, and select **Edit**. Input the new values in the dialog box.
- To edit an attribute label, double-click in a cell of the label viewer attribute table.
- To edit an ROI or point label, select the label in the **Labeled Signal Set Browser**, the time plot, or the label viewer. The chosen label is highlighted in all three regions and in the spectrogram. Click the label in the time plot, spectrogram, or label viewer to make it active. You can then edit its value and location. To accept the changes, click the **Accept** check mark next to the **Label** button, press **Enter**, or double-click.




This action is equivalent to using `setLabelValue` at the command line.

**Note**

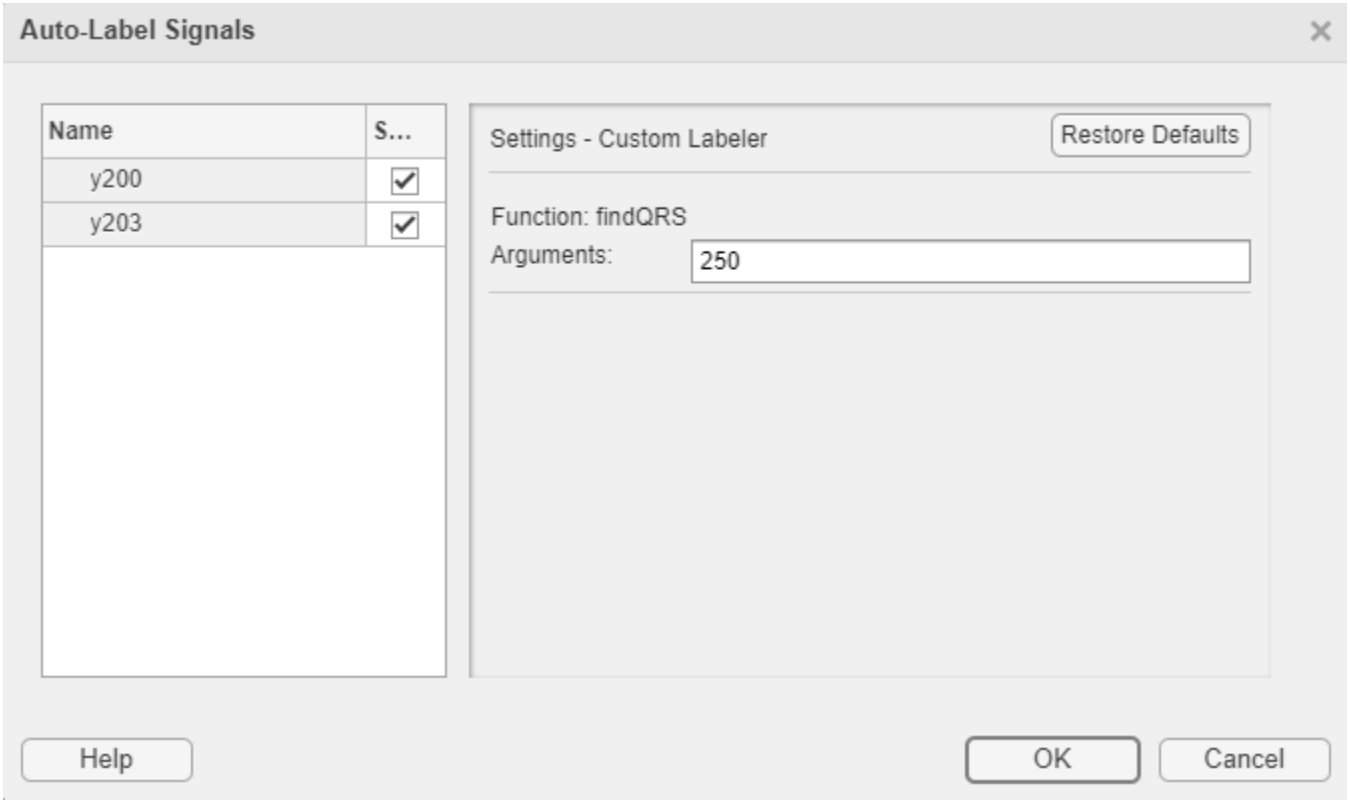
- You must select a parent label before you can label an ROI or point sublabel manually.
- To label an attribute sublabel manually, use the **Labeled Signal Set Browser** or the label viewer.

## Label Signals Automatically

### Bulk Autolabeling

To autolabel a set of signals using **Peak Labeler** or a custom function, start by creating a signal label definition that you want to apply or by selecting one from the **Label Definitions** browser. The **Automate Value** gallery shows the "Custom Labeling Functions" on page 22-25 that you can use with the definition. Choose a function from the gallery, click **Auto-Label ▼**, and select `Auto-Label Signals`. **Signal Labeler** prompts you to select the members you want to label and specify any optional input arguments to the labeling function.

For examples of autolabeling functions, see "Custom Labeling Functions" on page 22-25.

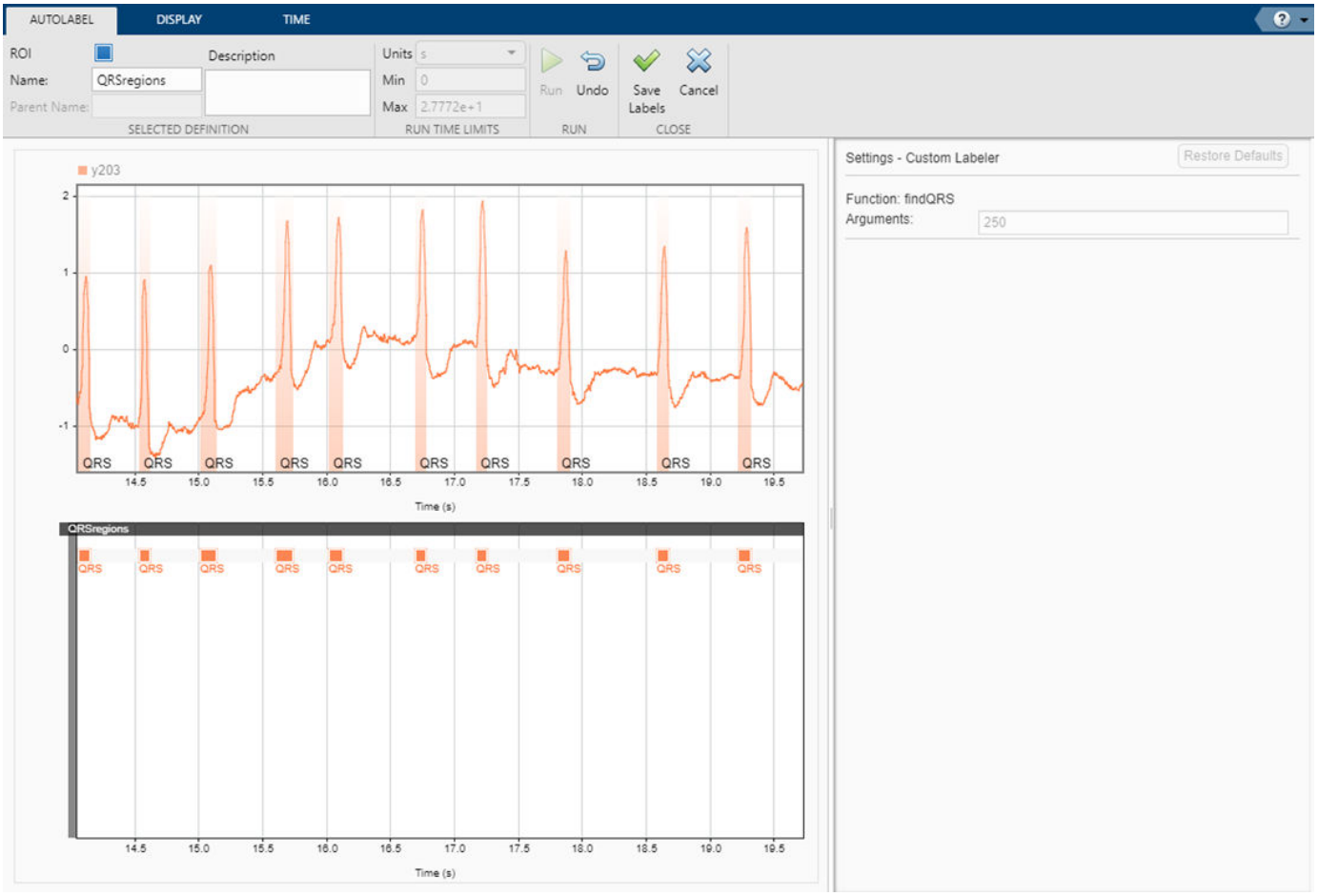



**Note** You can undo the last automated custom labeling you performed. However, you lose this ability once you add, modify, or delete any labels or label definitions.

For examples of bulk autolabeling, see “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51 and “Label Spoken Words in Audio Signals Using External API” on page 22-60.

### Autolabeling Mode

To label one or more signals interactively, first choose the signals that you want to plot by selecting the check boxes next to their names in the **Labeled Signal Set Browser**. Create a definition for the label you want to apply or select one from the **Label Definitions** browser. The **Automate Value** gallery shows the “Custom Labeling Functions” on page 22-25 that you can use with the definition. Choose a function from the gallery, click **Auto-Label ▼**, and select `Auto-Label and Inspect Plotted`. This action takes you to the **Signal Labeler** autolabeling mode.




When in the autolabeling mode, you can run the selected function on the whole signal or run it within a time range of your choice.

**Tip** The **Run Time Limits** section of the **Autolabel** tab shows the time range used by the autolabeling function and defaults to the entire signal. The range values change if you zoom in or specify values before running the autolabeling function. To reset the range to the entire signal, press the spacebar or click the **Fit to View** button on the **Display** tab.

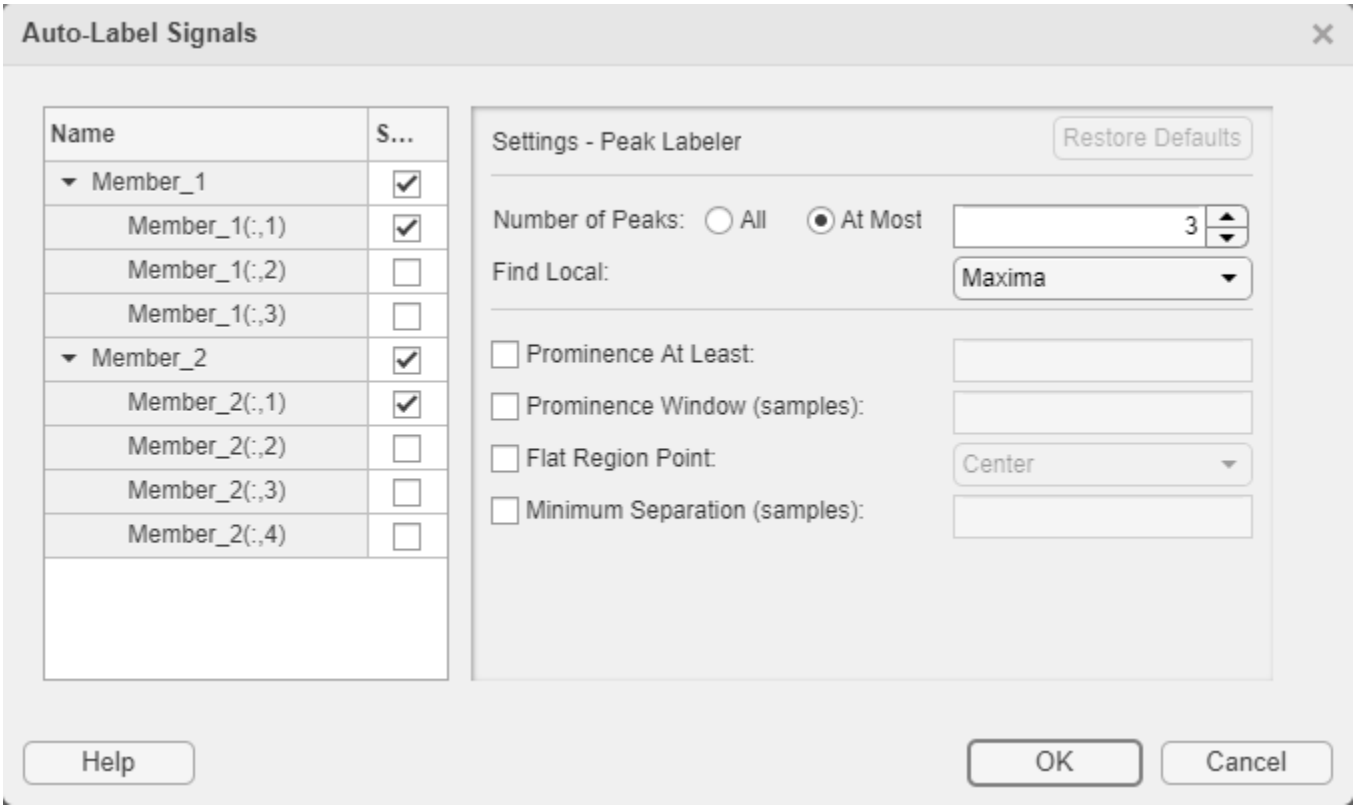
After running the function, you can inspect the labeling results. If the results are not satisfactory, you can undo the labeling action, modify the autolabeling function, and label the signal again. You can edit label values and locations either on the time plot or on the label viewer. You can also remove labels, but if you remove labels you lose the ability to undo the previous labeling action. When you are satisfied with the results, click **Save Labels** to save the labeling and exit the autolabeling mode.

**Note** For easier label inspection, the autolabeling mode displays only the labels generated during the current function call.

## Label Signal Peaks Automatically Using Peak Labeler

To autolabel signal peaks using **Peak Labeler**, start by selecting or creating a numeric point label definition. In the **Automate Value** gallery, select **Peak Labeler** and click **Auto-Label ▼**. You can label peaks in bulk or interactively. **Signal Labeler** uses the MATLAB functions `islocalmax` and `islocalmin` to find and label local maxima and minima. **Peak Labeler** returns the location of each peak and the numeric value of its amplitude.

**Note** You can select multiple members for peak labeling, but you can label only one channel of each member at a time. By default, **Signal Labeler** chooses the first channel of each selected member, but you can select which signal of the member you want to use for labeling.




**Note** You can undo the last automated peak labeling action. However, you lose this ability once you add, modify, or delete any labels or label definitions.

- You can choose to label all the peaks or valleys in the selected signal that satisfy the specified conditions. Alternatively, you can label a specific number of peaks or valleys that satisfy the specified conditions, sorted in order of descending prominence. See “Prominence” on page 18-22 for more information about peak prominence. **Peak Labeler** by default labels three peaks.
- If a local maximum or minimum value is repeated consecutively, the peak or valley belongs to a flat region. For a signal with flat peak or valley regions, you can choose to label the center of the region, the first point of the region, the last point of the region, or all points in the region.

- You can choose to label only those peaks or valleys with prominence larger than a specified value. You can also specify the width of the window centered on a peak or valley that is used to measure its prominence.
  - If you do not specify a window width, the **Peak Labeler** algorithm uses the entire signal as the window.
  - For a flat peak or valley region, the window is centered at the midpoint of the region.
- You can select to label only those peaks separated by a specified distance. The **Peak Labeler** algorithm:
  1. Chooses the most prominent peak in the signal and ignores all peaks within the specified distance.
  2. Repeats the procedure for the most prominent remaining peak.
  3. Iterates until it runs out of peaks to consider.

---

**Tip** If you label peaks in a signal using **Peak Labeler** and then move one of the labels, **Signal Labeler** still shows the amplitude value returned by **Peak Labeler**. To update the amplitude:

1. Read the new value on the data cursor you used to move the point label.
2. Edit the label.
3. Enter the new value in the **Value** field of the dialog box that appears.

---

For an example that uses **Peak Labeler**, see “Example: Label Points and Regions of Interest in Signal” on page 22-2.

## See Also

### Apps

**Signal Analyzer** | **Signal Labeler**

### Functions

`labeledSignalSet` | `signalLabelDefinition`

## Related Examples

- “Label Signal Attributes, Regions of Interest, and Points” on page 22-40
- “Examine Labeled Signal Set” on page 22-46
- “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51
- “Label Spoken Words in Audio Signals Using External API” on page 22-60

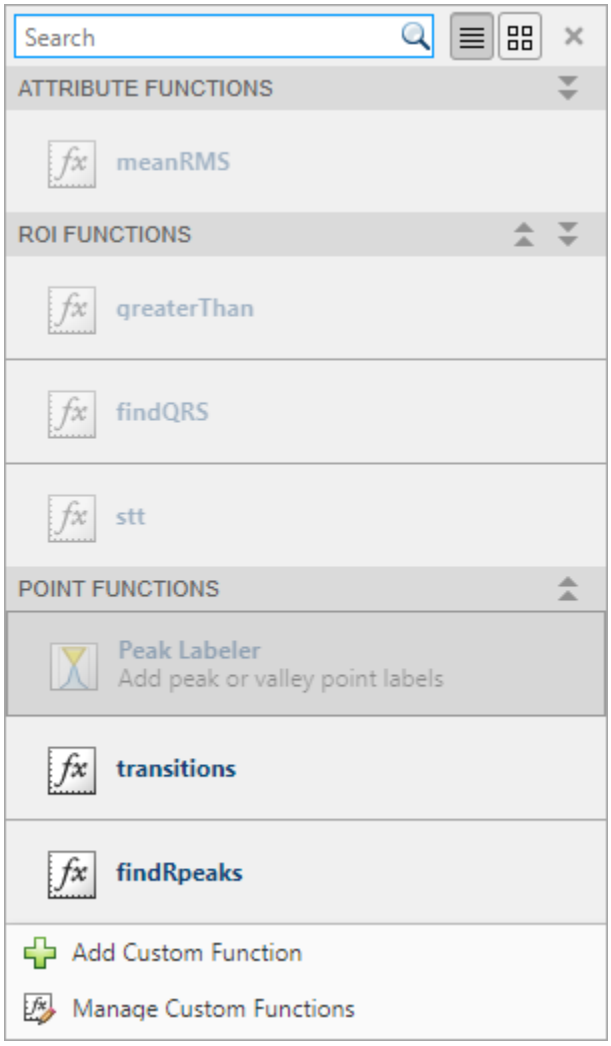
## More About

- “Using Signal Labeler App” on page 22-2
- “Import Data into Signal Labeler” on page 22-6
- “Create or Import Signal Label Definitions” on page 22-13

- “Custom Labeling Functions” on page 22-25
- “Customize Labeling View” on page 22-30
- “Export Labeled Signal Sets and Signal Label Definitions” on page 22-36
- “Signal Labeler Usage Tips” on page 22-38

# Custom Labeling Functions

You can use **Signal Labeler** to perform automated labeling tasks. Apart from **Peak Labeler**, the **Automate Value** gallery on the **Label** tab contains functions that you can use to label signals automatically.




## Create Custom Labeling Functions

The first line in every custom autolabeling function consists of a definition statement of the form

```
function [labelVals,labelLocs] = fx(x,t,parentLabelVal,parentLabelLoc,varargin)
```

The definition statement contains the function name and a set of mandatory and optional arguments:

- The first input argument, `x`, is the input signal. When writing the function, expect `x` to be a matrix where each column contains data corresponding to a channel. If the channels have different lengths, then expect `x` to be a cell array of column vectors.

- The second input argument, `t`, stores the time values. When writing the function, expect `t` to be a matrix where each column contains time information corresponding to a channel. If the channels have different lengths, then expect `t` to be a cell array of column vectors.

  **Note**

  - For single-channel members, custom autolabeling functions get data and time values as double precision vectors.
  - For multichannel members, custom autolabeling functions get data and time values as matrices or cell arrays.
  - Custom autolabeling functions get all the channels of a member as input, but they do not have to operate on all. You can choose which channels you want the function to operate on.

- The third input argument, `parentLabelVal`, is the parent label value associated with the output sublabel and contains a numeric, logical, or string scalar. This argument is passed in only for functions that automate the labeling of sublabels. If the function is for a parent label, expect `parentLabelVal` to be empty.
- The fourth input argument, `parentLabelLoc`, contains:

  - An empty vector when the parent label is an attribute
  - A two-element numeric row vector of region of interest (ROI) limits when the parent label is an ROI
  - A numeric scalar representing a point location when the parent label is a point

  This argument is passed in only for functions that automate the labeling of sublabels. If the function is for a parent label, expect `parentLabelLoc` to be empty.

  **Note**

  - For parent labels, the autolabeling function operates on each selected member.
  - For sublabels, the autolabeling function operates on one parent label instance at a time for each selected member.

- Use `varargin` to specify additional input arguments. If you do not have additional input arguments, you can omit `varargin`. Enter the additional arguments as an ordered comma-separated list in the dialog box that appears when you click the **Auto-Label** button.
- The first output argument, `labelVals`, contains the label values. `labelVals` must be:

  - A numeric, logical, or string scalar when the output labels are attributes
  - A column vector with numeric, logical, or string values when the output labels are ROIs or points
- The second output argument, `labelLocs`, contains the label locations. `labelLocs` must be:

  - An empty vector when the output labels are attributes
  - A two-column matrix of ROI limits when the output labels are ROIs
  - A column vector of point locations when the output labels are points
- To implement your algorithm, you can use any function from MATLAB or from any toolbox installed in your system.

For more details, see “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51 and “Label Spoken Words in Audio Signals Using External API” on page 22-60.

### Example: Mean RMS Value

This function computes the mean RMS value of a signal and labels the signal with the value as a numeric attribute. If a member has more than one channel, the function computes the RMS value of each channel and averages the values. The channels can have different lengths.

```
function [labelVals,labelLocs] = meanRMS(x,t,parentLabelVal,parentLabelLoc,varargin)
% Label signal with its mean RMS value as attribute

if iscell(x)
    labelVals = mean(cellfun(@rms,x))
else
    labelVals = mean(rms(x));
end
labelLocs = [];

end
```

### Example: Zero Crossings

This function finds the zero crossings of a signal and labels them as `"rising"` for positive-going transitions and `"falling"` for negative-going transitions.

```
function [labelVals,labelLocs] = transitions(x,t,parentLabelVal,parentLabelLoc,varargin)
% Label zero crossings as "rising" or "falling"

nchan = size(x,2);
tt = t(:,1);

labelVals = cell(nchan,1);
labelLocs = cell(nchan,1);

for kj = 1:nchan

    [~,ltup,utup] = risetime(x(:,kj),tt,'StateLevels',0.1*[-1 1]);

    ups = (utup+ltup)/2;
    upc = repmat("rising",length(ups),1);

    [~,ltdn,utdn] = falltime(x(:,kj),tt,'StateLevels',0.1*[-1 1]);

    dns = (utdn+ltdn)/2;
    dnc = repmat("falling",length(dns),1);

    labelVals{kj} = [upc;dnc];
    labelLocs{kj} = [ups;dns];

end

labelVals = cat(1,labelVals{:});
labelLocs = cell2mat(labelLocs);

end
```

#### Example: Multichannel Labeling

This logical function labels as true those regions of a multichannel signal where:

- The amplitude of the first channel is negative.
- The amplitude of the third channel is larger than a user-specified value, `mx`. If not specified, `mx` defaults to 0.1.

```
function [labelVals,labelLocs] = greaterThan(x,t,parentLabelVal,parentLabelLoc,varargin)
% Label regions with negative first channel and third channel larger than a given value

if nargin<5
    mx = 0.1;
else
    mx = varargin{1};
end

xr = x(:,1);
xx = x(:,3);
tt = t(:,1);

ss = signalMask(xx >= mx & xr < 0);
x = roimask(ss);

labelLocs = tt(x.ROILimits);
labelVals = logical(double(x.Value));

end
```

## Add Custom Labeling Functions to the Gallery

To add a custom autolabeling function, click the arrow next to the **Automate Value** gallery and then select **Add Custom Function**. In the dialog box, specify these fields:

- **Name** — Specify the name of the function you want to add.
- **Description** — Add a short description of what the function does and describe the optional input arguments.
- **Label Type** — Specify the type of label that the function generates. Select `Attribute` (the default), `ROI`, or `Point`.

> **Note** Based on the **Label Type** you specify, **Signal Labeler** places the function in the appropriate category in the **Automate Value** gallery. When you select a label definition, the gallery enables only those functions that can be used with that definition type.

If you have already written a function, and the function is in the current folder or in the MATLAB path, **Signal Labeler** incorporates it in the gallery. If you have not written the function yet, **Signal Labeler** opens a blank template in the Editor.

## Manage Custom Labeling Functions in Gallery

At any time, you can edit functions, edit function descriptions, or remove functions using the **Manage Custom Functions** option in the **Automate Value** gallery.

**Note** Using the **Manage Custom Functions** option changes only the function descriptions displayed in the **Automate Value** gallery. If you want to change the description in the file that contains the function, you must edit the file.

## See Also

### Apps

**Signal Analyzer** | **Signal Labeler**

### Functions

`labeledSignalSet` | `signalLabelDefinition`

## Related Examples

- “Label Signal Attributes, Regions of Interest, and Points” on page 22-40
- “Examine Labeled Signal Set” on page 22-46
- “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51
- “Label Spoken Words in Audio Signals Using External API” on page 22-60

## More About

- “Using Signal Labeler App” on page 22-2
- “Import Data into Signal Labeler” on page 22-6
- “Create or Import Signal Label Definitions” on page 22-13
- “Label Signals Interactively or Automatically” on page 22-17
- “Customize Labeling View” on page 22-30
- “Export Labeled Signal Sets and Signal Label Definitions” on page 22-36
- “Signal Labeler Usage Tips” on page 22-38

# Customize Labeling View

**Tip** To toggle the label viewer, click the **Display** tab and select **Label Viewer** in the **Views** section.

## Visualize Signal Spectra and Spectrograms

### Spectrum View

To activate the frequency-domain view of a signal, click the **Display** tab and select **Spectrum** in the **Views** section. The app displays a set of axes with the signal power spectrum and a **Spectrum** tab with options to control the view.

**Signal Labeler** scales the spectrum so that, if the frequency content of a signal falls exactly within a bin, its amplitude in that bin is the true average power of the signal. For example, the average power of a sinusoid is one-half the square of the sinusoid amplitude. For more details, see “Measure Power of Deterministic Periodic Signals” on page 24-289. **Signal Labeler** computes spectra using the same steps as **Signal Analyzer**. For more information, see “Spectrum Computation in Signal Analyzer” on page 21-109.

**Note** When displaying a spectrum, **Signal Labeler** converts the power to dB using $10\log_{10}(\text{Power})$.

### Spectrogram View

To activate the time-frequency view of a signal, click the **Display** tab and select **Spectrogram** in the **Views** section. The app displays a set of axes with the signal spectrogram and a **Spectrogram** tab with options to control the view.

To apply reassignment to a spectrogram, check **Reassign** in the **Spectrogram** tab. The reassignment technique sharpens the time and frequency localization of spectrograms by reassigning each power spectrum estimate to the location of its center of energy. If your signal contains well-localized temporal or spectral components, then this option generates a spectrogram that is easier to read and interpret.

**Signal Labeler** computes spectrograms using the same steps as **Signal Analyzer**. For more information, see “Spectrogram Computation in Signal Analyzer” on page 21-115.

**Note** You can plot the spectrogram of only one signal per display.

### Settings

- If the panner is activated and is zoomed in on a particular region of interest (ROI), the display corresponds to the region of interest, not the entire signal.
- If you zoom in on a region of the signal in the time plot using one of the zoom actions on the **Display** tab, the display corresponds to the region of interest, not the entire signal.
- You cannot zoom out in frequency beyond the Nyquist range.

If a signal is nonuniformly sampled, then **Signal Labeler** interpolates the signal to a uniform grid to compute spectral estimates. The app uses linear interpolation and assumes a sample time equal to

the median of the differences between adjacent time points. For a nonuniformly sampled signal to be supported, the median time interval and the mean time interval must obey

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

## Use Spectrogram to Aid Labeling

To label a signal using the spectrogram view, select the check box next to the signal name in the **Labeled Signal Set Browser** and activate the spectrogram view. Select the label definition you want to apply on the **Label Definitions** browser. **Draw Labels** is automatically activated when the label definition is an ROI or a point. Specify the label value in the **Set Value** section of the **Labeler** tab, or assign the value after the label is drawn. Click on the spectrogram to add the label:

- For ROI labels, click and drag the animated dashed line to create a shaded region. Move and resize the active region until it encloses the region of interest.
- For point labels, move the animated dashed line until it crosses the signal at the point of your choice.

To accept a label, click the **Accept** check mark in the **Options** section, press **Enter**, or double-click the active region or line. A label is automatically accepted when a subsequent label is drawn.

---

**Tip** The time-frequency view can be used to identify and label transient narrowband signals embedded in broadband signals. For more information, see “Find Interference Using Persistence Spectrum” on page 21-51.

---

### Example: Label a Signal in Spectrogram View

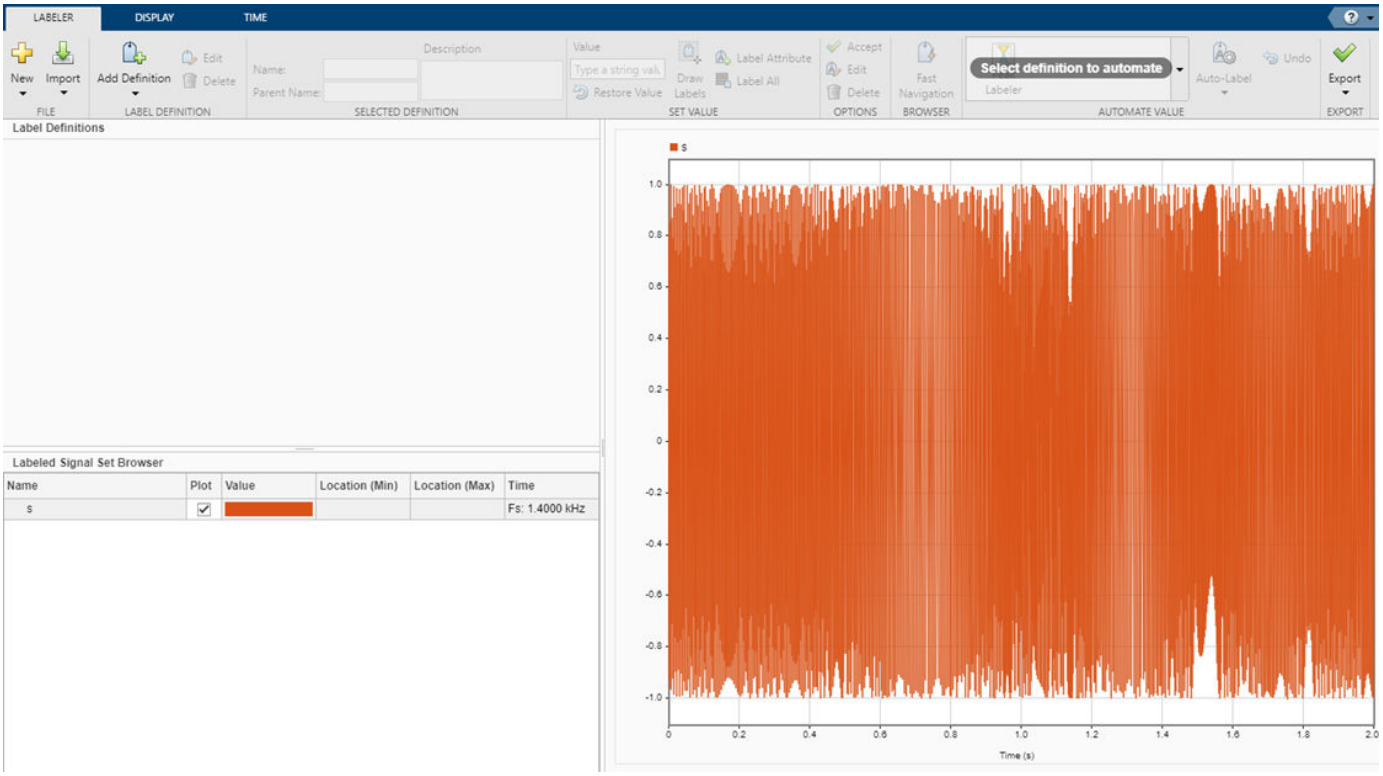
Generate a signal composed of a voltage-controlled oscillator and four Gaussian atoms. The signal is sampled at 1.4 kHz. Use **Spectrogram** in Signal Labeler to label the Gaussian atoms in the signal.

```
fs = 1400;
t = (0:1/fs:2)';

gaussFun = @(A,x,mu,f) exp(-(x-mu).^2/(2*0.01^2)).*sin(2*pi*f.*x)*A';
s = gaussFun([1 1 1 1],t,[0.2 0.5 1 1.75],[2 6 2 5]*100)/10;
x = vco(chirp(t+.1,0,t(end),3).*exp(-2*(t-1).^2),[0.1 0.4]*fs,fs);

s = s/10+x;
```

Open **Signal Labeler** and import the signal:

1. On the **Labeler** tab, click **Import ▼**, select `From workspace` in the **Members** list, and select the `s` variable in the dialog box that appears.
2. Set the time information. Select `Time` from the **Working in** drop-down list, select `Sample Rate`, and specify the sample rate as `1.4 kHz`.
3. Click **Import ▼** and close the dialog box.
4. Select the check box next to the signal name in the **Labeled Signal Set Browser** to display the signal in the time plot.

The oscillating chirp dominates the signal and obscures the atoms in the time plot.

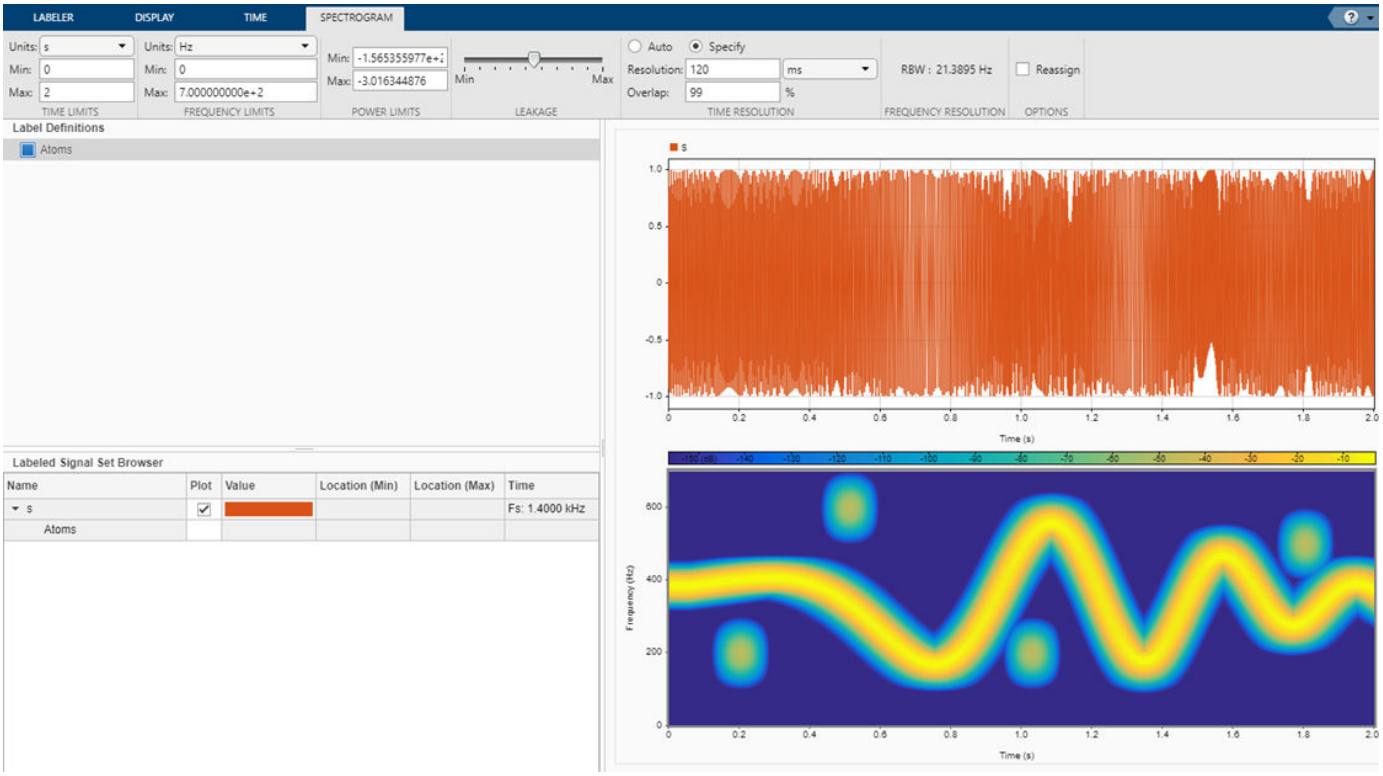



Create a signal label definition:

1. Click **Add Definition** ▼ and select `Add label definition`.
2. In the dialog box, specify **Label Name** as `Atoms`, **Label Type** as `ROI`, and **Data Type** as `logical`.
3. Click **OK**.

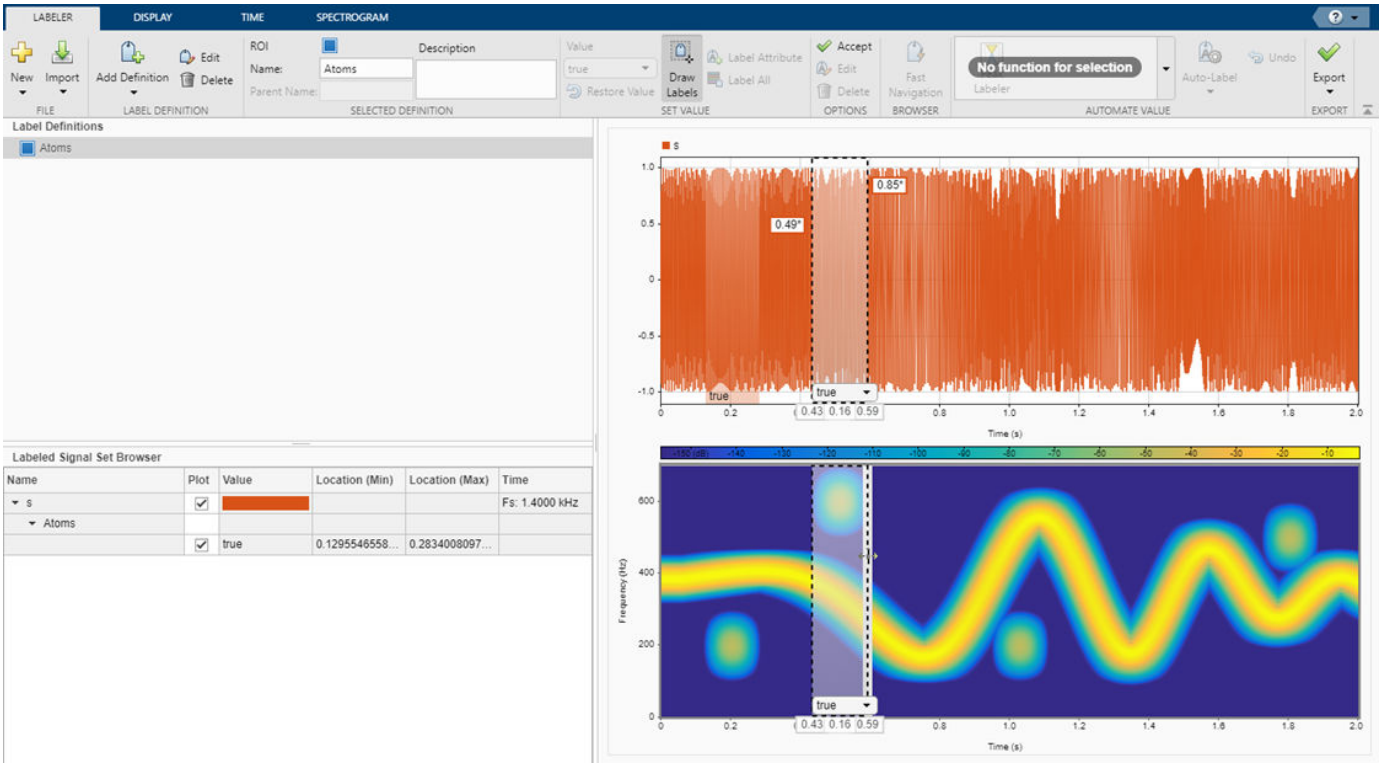
On the **Display** tab, click `Spectrogram` in the **Views** section to activate the spectrogram view of the signal. Adjust the spectrogram settings in the **Spectrogram** tab. In the **Time Resolution** section, **Specify** a **Resolution** of `120 ms` and an **Overlap** of 99%.

The locations of the atoms are visible in the spectrogram.




Label the atoms:

1. In the **Labeler** tab, select the label definition in the **Label Definitions** browser.
2. Specify the label value as `true` in the **Set Value** section.
3. Click the spectrogram plot to draw the label. Click and drag the animated dashed line to create a shaded region. Move and resize the active region until it encloses the first atom.
4. Click the spectrogram plot again to draw the label for a different atom. The app automatically accepts the previous label.
5. Repeat the procedure to label the other two atoms.




## See Also

### Apps

**Signal Analyzer** | **Signal Labeler**

### Functions

`labeledSignalSet` | `signalLabelDefinition`

## Related Examples

- “Label Signal Attributes, Regions of Interest, and Points” on page 22-40
- “Examine Labeled Signal Set” on page 22-46
- “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51
- “Label Spoken Words in Audio Signals Using External API” on page 22-60

## More About

- “Using Signal Labeler App” on page 22-2
- “Import Data into Signal Labeler” on page 22-6
- “Create or Import Signal Label Definitions” on page 22-13
- “Label Signals Interactively or Automatically” on page 22-17
- “Custom Labeling Functions” on page 22-25

- “Export Labeled Signal Sets and Signal Label Definitions” on page 22-36
- “Signal Labeler Usage Tips” on page 22-38

# Export Labeled Signal Sets and Signal Label Definitions

At the end of a labeling session, you can export any signal sets that you labeled or any label definitions that you created. You can export labeled signal sets and definitions either to MAT-files or to the MATLAB Workspace. **Signal Labeler** exports labeled signals as `labeledSignalSet` objects.

## Export Label Definitions

To export signal label definitions, click **Export ▼** on the **Labeler** tab and select `To Workspace` or `To File` under `Label Definitions`. In the dialog box, specify the name of a variable or MAT-file. The specified file or variable contains an array of `signalLabelDefinition` objects with all the definitions you created, imported, and modified in the current **Signal Labeler** session.

**Note** If you want to reuse signal label definitions created during a **Signal Labeler** session, you must export the definitions to a MAT-file and import them in a subsequent session.

## Export Labeled Signal Sets

To export new labeled signal sets, click **Export ▼** on the **Labeler** tab and select `To Workspace` or `To File` under `Labeled Signal Sets`. In the dialog box, specify the name of a variable or MAT-file and an optional brief description. The variable or file contains a `labeledSignalSet` object that merges all the signals, label definitions, and label values that you created, imported, and modified in the current **Signal Labeler** session. All signals that contain time information are converted to MATLAB timetables.

**Note** When exporting labeled signal sets, **Signal Labeler** converts all signals with time information to timetables. This conversion results in a deeper hierarchy of nested channels in the exported `labeledSignalSet` object. You can see the deeper hierarchy if you import the labeled signal set again into the app.

## See Also

**Apps**
**Signal Analyzer** | **Signal Labeler**

**Objects**
`labeledSignalSet` | `signalLabelDefinition` | `signalMask`

## Related Examples

- “Label Signal Attributes, Regions of Interest, and Points” on page 22-40
- “Examine Labeled Signal Set” on page 22-46
- “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51
- “Label Spoken Words in Audio Signals Using External API” on page 22-60

## More About

- “Using Signal Labeler App” on page 22-2
- “Import Data into Signal Labeler” on page 22-6
- “Create or Import Signal Label Definitions” on page 22-13
- “Label Signals Interactively or Automatically” on page 22-17
- “Custom Labeling Functions” on page 22-25
- “Customize Labeling View” on page 22-30
- “Signal Labeler Usage Tips” on page 22-38

# Signal Labeler Usage Tips

## Keyboard Shortcuts

**Note** On Macintosh platforms, use the **Command** key instead of **Ctrl**.

### Labeling

| Task | Shortcut |
|---|---|
| Accept labeling | **Enter** |
| Delete label | **Del** |
| Cancel labeling | **Esc** |
| Activate **Draw Labels** mode | **Ctrl+L** |
| Plot previous channel | **Ctrl+Up arrow** |
| Plot next channel | **Ctrl+Down arrow** |

### Zooming

| Task | Shortcut |
|---|---|
| Zoom in X-axis | **Ctrl+Shift+T** |
| Zoom in Y-axis | **Ctrl+Shift+Y** |
| Zoom in X and Y | **Ctrl++** (numeric keypad only) |
| Zoom out | **Ctrl+-** (numeric keypad only) |
| Fit to view | **Spacebar** |
| Cancel zoom operation | **Esc** |

**Tip** If you label peaks in a signal using **Peak Labeler** and then move one of the labels, **Signal Labeler** still shows the amplitude value returned by **Peak Labeler**. To update the amplitude:

1. Read the new value on the data cursor you used to move the point label.
2. Edit the label.
3. Enter the new value in the **Value** field of the dialog box that appears.

**Note**

- You must select a parent label before you can label an ROI or point sublabel manually.
- To label an attribute sublabel manually, use the **Labeled Signal Set** browser or the label viewer.

## Troubleshooting

- When you reduce the width of the **Signal Labeler** window, the attribute table under the time plot shows a horizontal scrollbar. In Apple macOS systems, the scrollbar is hidden by default. If you have a trackpad and scroll horizontally, the scrollbar appears. If you use a mouse and want the scrollbar to be visible always, you can modify the system behavior in the System Preferences.
- **Signal Labeler** can fail to start if MATLAB is using a software implementation of OpenGL. To solve the problem, upgrade your graphics hardware driver or use `opengl` to switch to a hardware-accelerated implementation of OpenGL. See “Resolving Low-Level Graphics Issues” for more information.
- Attempting to start **Signal Labeler** can cause JavaScript support for WebGL to fail. To solve the problem, update your graphics hardware driver.
- **Signal Labeler** can fail to start due to a network error. Check your organization's proxy settings and, if possible, disable the proxy that is interfering with the app startup process.

## See Also

### Apps

**Signal Analyzer** | **Signal Labeler**

### Functions

`labeledSignalSet` | `signalLabelDefinition`

## Related Examples

- “Label Signal Attributes, Regions of Interest, and Points” on page 22-40
- “Examine Labeled Signal Set” on page 22-46
- “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51
- “Label Spoken Words in Audio Signals Using External API” on page 22-60

## More About

- “Using Signal Labeler App” on page 22-2
- “Import Data into Signal Labeler” on page 22-6
- “Create or Import Signal Label Definitions” on page 22-13
- “Label Signals Interactively or Automatically” on page 22-17
- “Custom Labeling Functions” on page 22-25
- “Customize Labeling View” on page 22-30
- “Export Labeled Signal Sets and Signal Label Definitions” on page 22-36

# Label Signal Attributes, Regions of Interest, and Points

Recordings of whale songs contain trills and moans. *Trills* sound like series of clicks. *Moans* are low-frequency cries similar to the sound made by a ship's horn. You want to look at each signal and label it to identify the whale type, the trill regions, and the moan regions. For each trill region, you also want to label a few selected signal peaks.

### Load Unlabeled Data

Start by loading a data set that includes two recordings of whale songs. The signals are called `whale1` and `whale2` and are sampled at 4 kHz. `whale1` consists of a trill followed by three moans. `whale2` consists of two moans, a trill, and another moan.

```
load labelwhalesignals

% To hear, type soundsc(whale1,Fs), pause(22), soundsc(whale2,Fs)
```

Bring the signals into **Signal Labeler**:

1. Open Signal Labeler. On the **Labeler** tab, click **Import** ▼ and select `From workspace` in the **Members** list.
2. In the dialog box, select the signals. Add time information: Select `Time` from the drop-down list and enter the sample rate, `Fs`, which is measured in kHz. Click **Import** to import the signals and then click **Close** to close the dialog box. The signals appear in the **Labeled Signal Set Browser**. To plot any of the signals, you can select the check box next to its name in the browser.

### Add Signal Label Definitions

Define labels to attach to the signals. For more information about the kinds of labels you can define, see “Create or Import Signal Label Definitions” on page 22-13.

For the whale song signals:

1. Define a categorical attribute label to store whale types. Call it `WhaleType`. The possible categories are blue whale, humpback whale, and white whale.
2. Define a logical region-of-interest (ROI) label that is true for moan regions. Call it `MoanRegions`.
3. Define a logical ROI label that is true for trill regions. Call it `TrillRegions`.
4. Define a numeric point label to capture trill peaks. Call it `TrillPeaks`. Set this label as a sublabel of the `TrillRegions` label.

To define each label, click **Add Definition** ▼ on the **Labeler** tab. To define the sublabel, select the `TrillRegions` label in the **Label Definitions** browser, click **Add Definition** ▼, and select `Add sublabel definition`.

Enter the following values in the fields in the dialog box that appears for each signal label or sublabel definition. Leave the **Default** field empty in each case.

| Label Name | Label Type | Label Description | Data Type | Categories |
|---|---|---|---|---|
| WhaleType | Attribute | Whale type | categorical | blue<br>humpback<br>white |
| MoanRegions | ROI | Regions where moans occur | logical | – – – |
| TrillRegions | ROI | Regions where trills occur | logical | – – – |
| TrillPeaks | Point | Trill peaks | numeric | – – – |

You can export the signal definitions you created to a MAT-file or to the MATLAB® Workspace by clicking **Export ▼**. A dialog box appears that prompts you for a file name. At any point you can import signal definitions stored in a MAT-file by clicking **Import ▼**.

### Label Signal Attributes

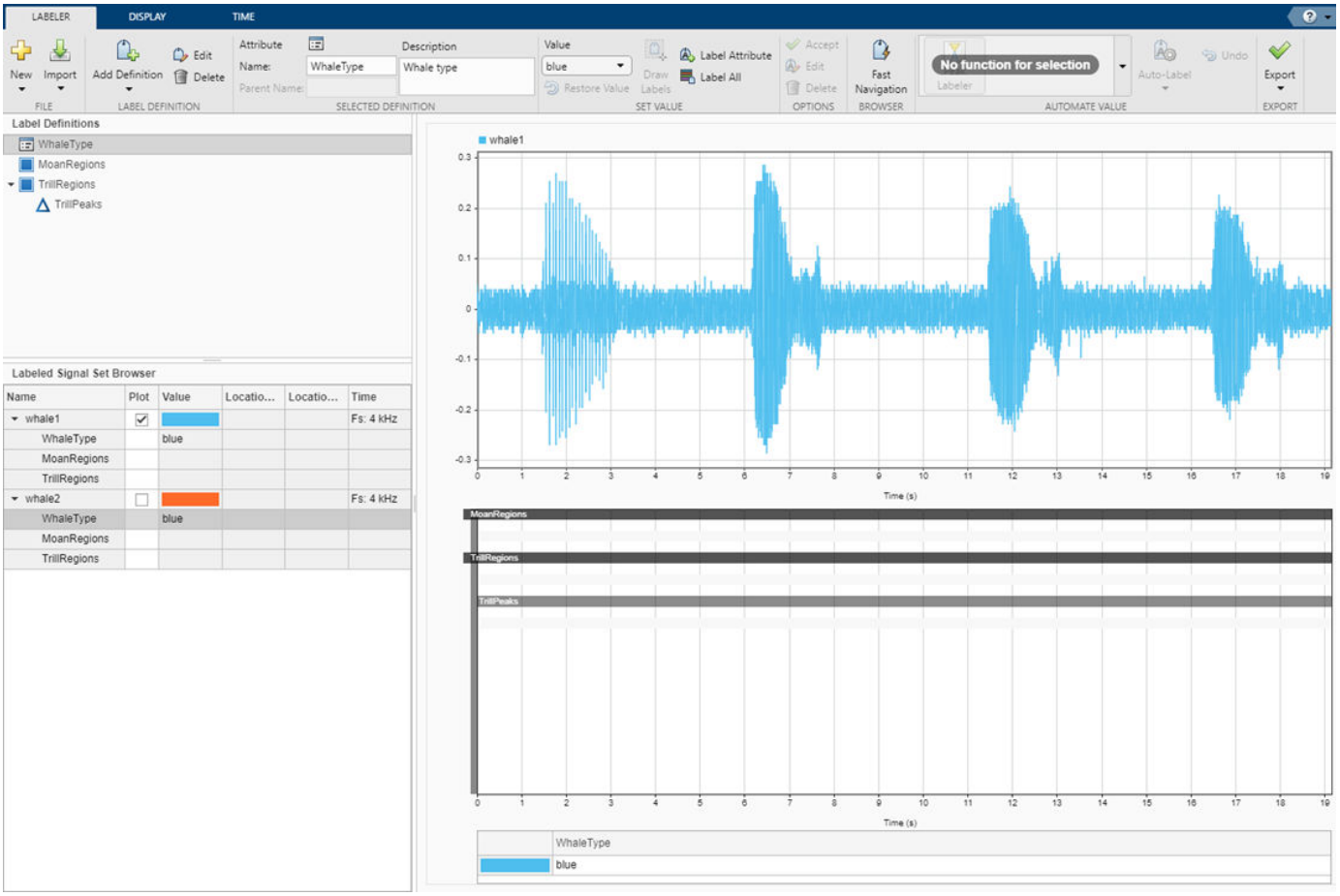
The songs in the data are from two blue whales. Set the `WhaleType` values for both signals:

1. Select `WhaleType` on the **Label Definitions** browser.
2. Click **Label All**.
3. In the dialog box that appears, verify that both `whale1` and `whale2` are selected and that the **Value** field is set to `blue`. (If you do not specify a default value in a categorical signal label definition, **Signal Labeler** sets the label to the first category specified in the definition.)
4. Click **OK**.

Set the `WhaleType` value for one signal at a time:

1. Check the box next to the signal name in the **Labeled Signal Set Browser**.
2. Select `WhaleType` on the **Label Definitions** browser.
3. Click **Label Attribute**. The value for `WhaleType` appears in the **Labeled Signal Set Browser**.
4. Select the value for `WhaleType` from the **Value** drop-down list.

Plot the `whale1` signal by selecting the check box next to its name. Signal attributes appear both in the **Labeled Signal Set Browser** and under the time plot.




### Label Signal Regions

Visualize the whale songs and label the trill and moan regions.

- Trill regions have distinct bursts of sound punctuated by silence. `whale1` has a trill centered at about 2 seconds.
- Moan regions are sustained low-frequency wails. `whale1` has moans centered at about 7 seconds, 12 seconds, and 17 seconds.

Label the signals one at a time:

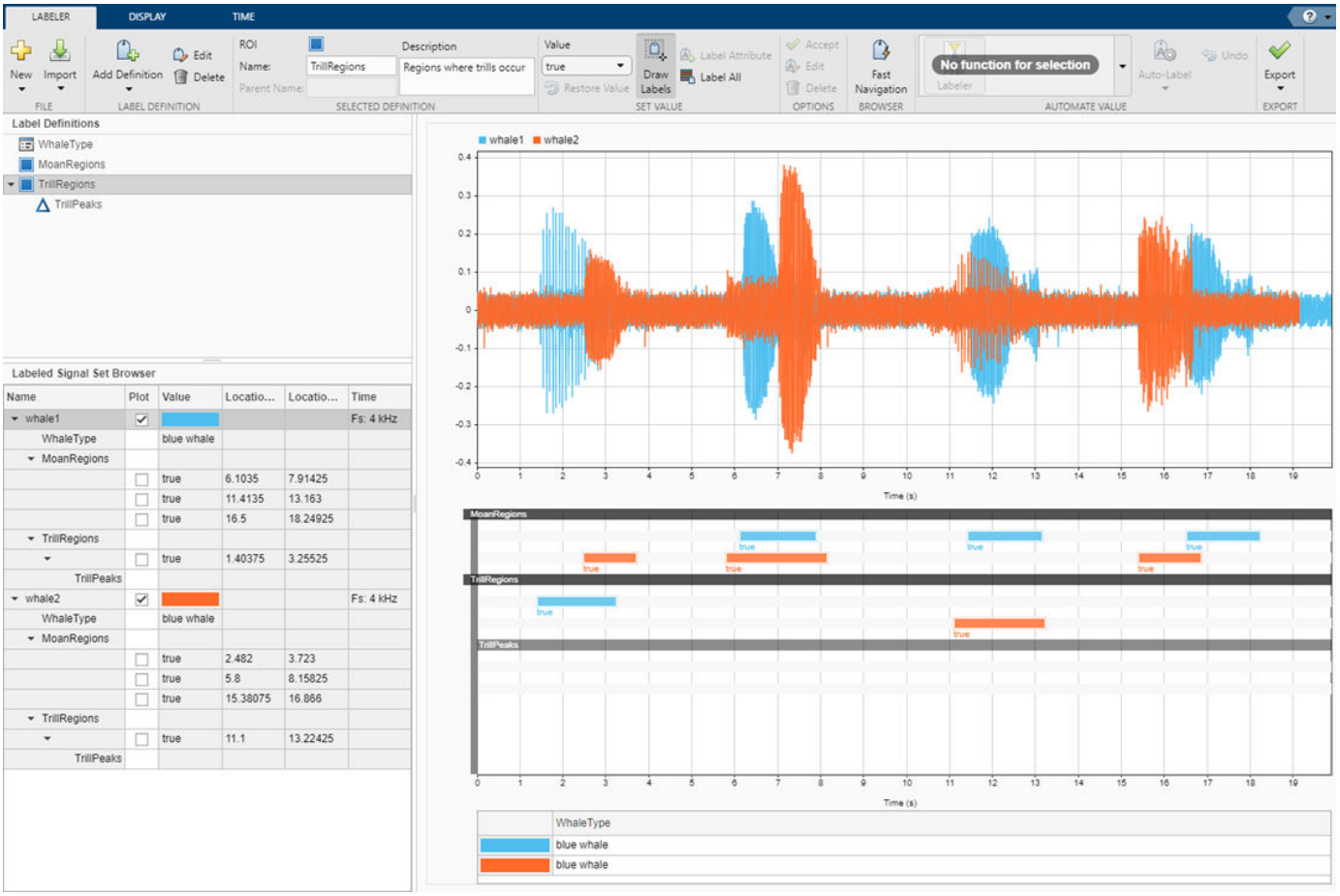
1. On the **Plot** column of the **Labeled Signal Set Browser**, check the box next to the signal name to plot the signal.
2. To label a moan, on the **Label Definitions** browser, select the `MoanRegions` label definition. **Draw Labels** is automatically activated.
3. Click the time plot. A thick animated dashed line appears that expands into a shaded region when you click and drag.
4. Move and resize the active region until it encloses a moan region. For better label placement, you can go to the **Display** tab and choose a zoom action or activate the panner.
5. Click the **Accept** check mark next to the **Label** button, press **Enter**, or double-click to label the ROI. The region changes to a gradient of the signal color. If you do not specify a default value in a logical label definition, **Signal Labeler** sets the label to `true`.

6 Repeat the procedure for the other two moans.

7 To label a trill, on the **Label Definitions** browser, select the `TrillRegions` label definition. Label the trill region using steps 3 and 4.

8 Before labeling the second signal, remove the first signal from the plot by clearing the check box next to its name in the **Labeled Signal Set Browser**. If you have the two signals plotted when you label a region or point, **Signal Labeler** associates the label with both signals.

Plot the two signals. The label viewer axes show the locations and widths of the regions of interest. They also show the value assigned to each region.




### Label Signal Points

Trill regions have distinct peaks that correspond to bursts of sound. Label three peaks in each trill region. Because trill peaks are sublabels, each one must be associated with a particular `TrillRegions` label.

Label the signals one at a time:

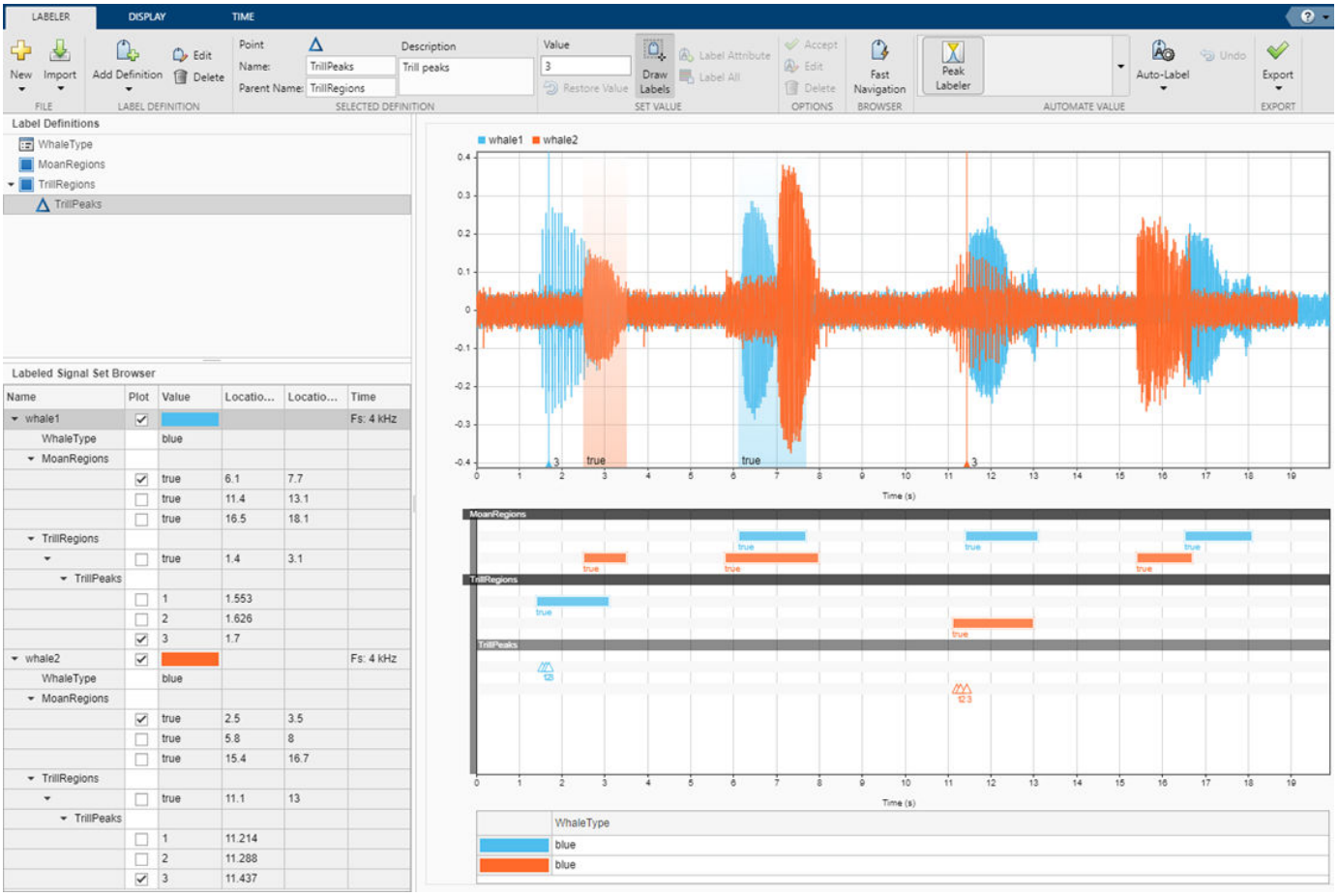
1 On the **Plot** column of the **Labeled Signal Set Browser**, check the box next to the signal name to plot the signal. Also check the box corresponding to the trill region whose peaks you want to label.

2 On the **Label Definitions** browser, select `TrillPeaks`. **Draw Labels** is automatically selected.

3. On the toolstrip, under **Value**, enter 1, corresponding to the first peak.
4. On the **Labeled Signal Set Browser**, select the trill region.
5. Click the point on the time plot. The trill region is framed by a solid line, and an animated dashed (active) line appears for the point being labeled.
6. Move the active line until it crosses the signal at a peak of your choice. For better label placement, you can go to the **Display** tab and choose a zoom action or activate the panner.
7. Click the check mark next to the **Label** button, press **Enter**, or double-click to label the peak. The dashed line changes to a solid line of the same color as the signal.
8. Repeat for two more peaks, entering 2 and 3 under **Value** to identify them.
9. Before labeling trill peaks for the second signal, remove the first signal from the plot by clearing the check box next to its name in the **Labeled Signal Set Browser**.

The label viewer axes show the locations of the points of interest and the value assigned to each point.

Plot the two signals to see a summary of their labels in the **Label Viewer**. Expand the labeled signal set hierarchy in the **Labeled Signal Set Browser** to see details for all the labels. (To expand the hierarchy, right-click any signal in the browser and select `Expand All`.) For each signal, plot the first moan region and the third trill peak that you labeled.



#### Edit Signal Label Values

At any point, you can edit any signal label using the **Labeled Signal Set Browser**, the time plot, or the label viewer. For more information, see “Edit Labels” on page 22-18.

#### Export Labeled Signal Set

Export the labeled signals as a new `labeledSignalSet` object. Click the **Export ▼** button on the **Labeler** tab of the toolstrip. You can export the labeled signal set to the MATLAB Workspace or to a MAT-file. For this example, choose a MAT-file. In the dialog box that appears, give the name `Whale_Songs.mat` to the labeled signal set, add an optional short description, and click **Export**. See “Export Labeled Signal Sets and Signal Label Definitions” on page 22-36 for more information on how **Signal Labeler** exports labeled signal sets.

## See Also

#### Apps
**Signal Analyzer** | **Signal Labeler**

#### Objects
`labeledSignalSet` | `signalLabelDefinition` | `signalMask`

## Related Examples

- “Examine Labeled Signal Set” on page 22-46
- “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51
- “Label Spoken Words in Audio Signals Using External API” on page 22-60

## More About

- “Using Signal Labeler App” on page 22-2
- “Import Data into Signal Labeler” on page 22-6
- “Create or Import Signal Label Definitions” on page 22-13
- “Label Signals Interactively or Automatically” on page 22-17
- “Custom Labeling Functions” on page 22-25
- “Customize Labeling View” on page 22-30
- “Export Labeled Signal Sets and Signal Label Definitions” on page 22-36
- “Signal Labeler Usage Tips” on page 22-38

# Examine Labeled Signal Set

Load into the MATLAB® Workspace the MAT-file you created in the “Label Signal Attributes, Regions of Interest, and Points” on page 22-40 example. Verify that the labeled signal set contains the definitions that you added using **Signal Labeler**.

```
load Whale_Songs

labelDefinitionsSummary(whalesongs)
```

```
ans=3×9 table
      LabelName        LabelType     LabelDataType     Categories     ValidationFunction    Defau
    _____________    ____________    _____________    ____________    __________________    _____

    "WhaleType"      "attribute"     "categorical"    {3x1 string}       {["N/A"   ]}       {0x0
    "MoanRegions"    "roi"           "logical"        {["N/A"   ]}       {0x0 double}       {0x0
    "TrillRegions"   "roi"           "logical"        {["N/A"   ]}       {0x0 double}       {0x0
```

Verify that `TrillPeaks` is a sublabel of `TrillRegions`.

```
labelDefinitionsHierarchy(whalesongs)
```

```
ans =
    'WhaleType
       Sublabels: []
     MoanRegions
       Sublabels: []
     TrillRegions
       Sublabels: TrillPeaks
     '
```

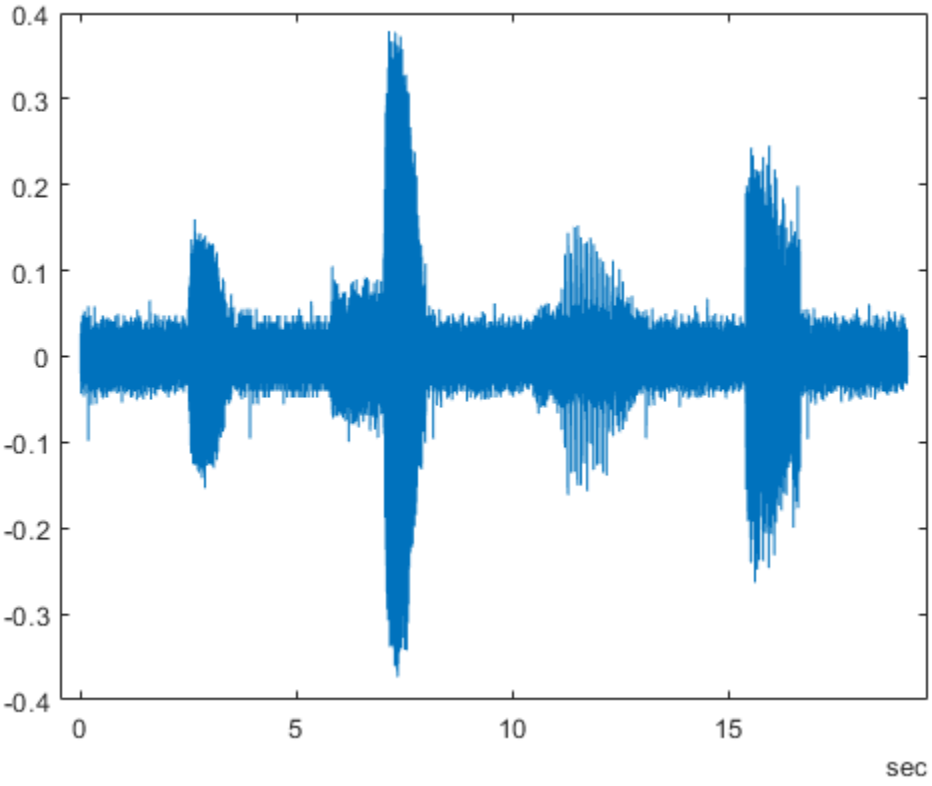
Retrieve the second member of the set. Retrieve the names of the timetable variables.

```
song = getSignal(whalesongs,2);

summary(song)
```

```
RowTimes:

    Time: 76579x1 duration
        Values:
            Min           0 sec
            Median        9.5722 sec
            Max           19.144 sec
            TimeStep      0.00025 sec

Variables:

    whale2: 76579x1 double

        Values:

            Min       -0.37326
            Median           0
            Max        0.37914
```

Plot the signal.

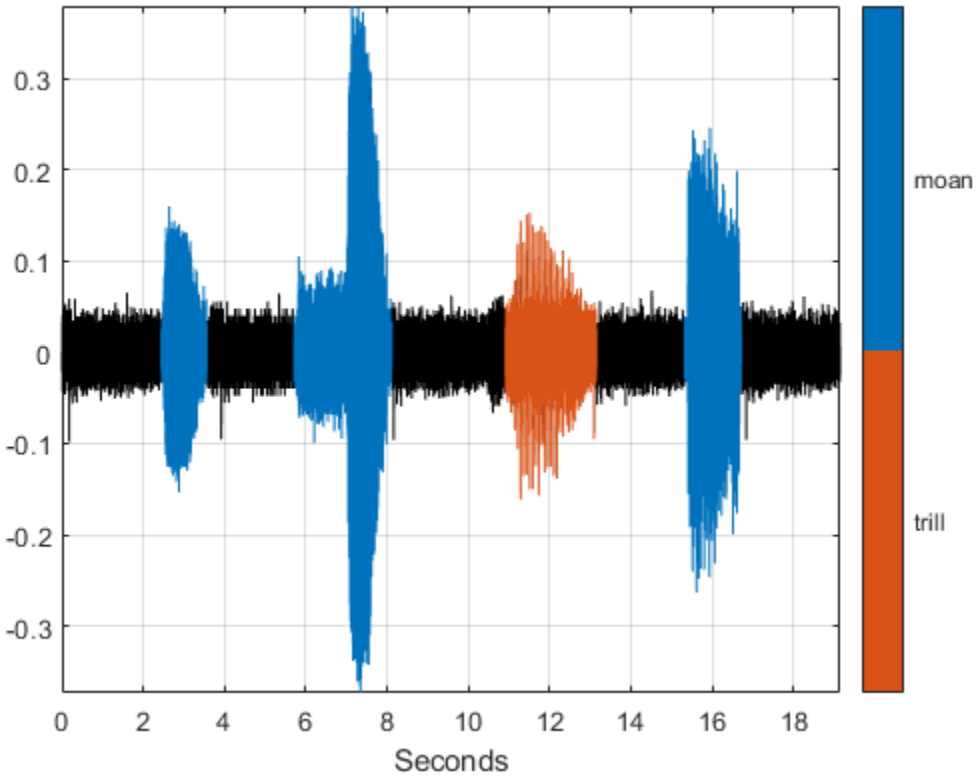
```
t = song.Time;
sng = song.whale2;

plot(t,sng)
```




### Visualize Labeled Regions

Use a `signalMask` object to display and identify the regions of interest that you labeled. For better display, change the label values from logical to categorical.

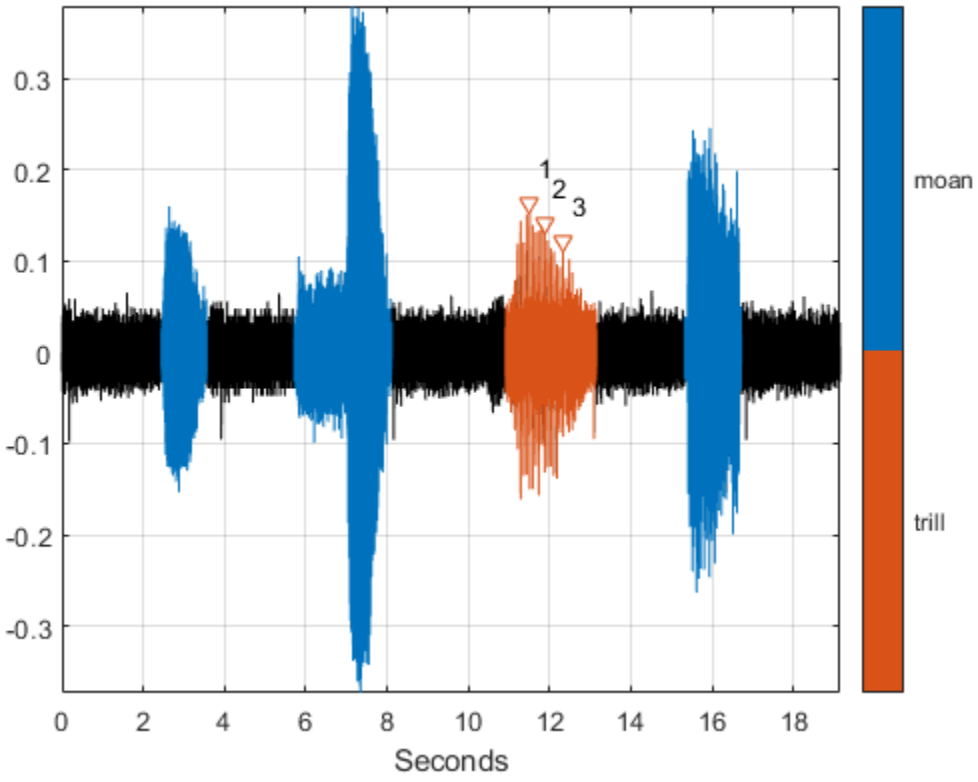
```
mvals = getLabelValues(whalesongs,2,'MoanRegions');
mvals.Value = categorical(repmat("moan",size(mvals,1),1));

tvals = getLabelValues(whalesongs,2,'TrillRegions');
tvals.Value = categorical(repmat("trill",size(tvals,1),1));

msk = signalMask([mvals;tvals],'SampleRate',1/seconds(t(2)-t(1)));

plotsigroi(msk,sng)
```




### Visualize Labeled Points

Display and identify the trill peaks that you labeled.

```
pk = getLabelValues(whalesongs,2,{'TrillRegions','TrillPeaks'});

locs  = zeros(size(pk,1),1);
for kj = 1:length(locs)
    locs(kj) = find(seconds(t) == pk.Location(kj));
end

hold on
plot(seconds(t(locs)),sng(locs)+0.01,'v')
text(seconds(t(locs))+0.2,sng(locs)+0.05,int2str(cell2mat(pk.Value)))
hold off
```




## See Also

**Apps**
**Signal Analyzer** | **Signal Labeler**

**Objects**
`labeledSignalSet` | `signalLabelDefinition` | `signalMask`

## Related Examples

- “Label Signal Attributes, Regions of Interest, and Points” on page 22-40
- “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51
- “Label Spoken Words in Audio Signals Using External API” on page 22-60

## More About

- “Using Signal Labeler App” on page 22-2
- “Import Data into Signal Labeler” on page 22-6
- “Create or Import Signal Label Definitions” on page 22-13
- “Label Signals Interactively or Automatically” on page 22-17
- “Custom Labeling Functions” on page 22-25

- “Customize Labeling View” on page 22-30
- “Export Labeled Signal Sets and Signal Label Definitions” on page 22-36
- “Signal Labeler Usage Tips” on page 22-38

# Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network

This example shows how to use custom autolabeling functions in **Signal Labeler** to label QRS complexes and R peaks of electrocardiogram (ECG) signals. One custom function uses a previously trained recurrent deep learning network to identify and locate the QRS complexes. Another custom function uses a simple peak finder to locate the R peaks. In this example, the network labels the QRS complexes of two signals that are completely independent of the network training and testing process.

The QRS complex, which consists of three deflections in the ECG waveform, reflects the depolarization of the right and left ventricles of the heart. The QRS is also the highest-amplitude segment of the human heartbeat. Study of the QRS complex can help assess the overall health of a person's heart and the presence of abnormalities [1 on page 22-0 ]. In particular, by locating R peaks within the QRS complexes and looking at the time intervals between consecutive peaks, a diagnostician can compute the heart-rate variability of a patient and detect cardiac arrhythmia.

The deep learning network in this example was introduced in “Waveform Segmentation Using Deep Learning” on page 24-330, where it was trained using ECG signals from the publicly available QT Database [2 on page 22-0 ] [3 on page 22-0 ]. The data consists of roughly 15 minutes of ECG recordings from a total of 105 patients, sampled at 250 Hz. To obtain each recording, the examiners placed two electrodes on different locations on a patient's chest, which resulted in a two-channel signal. The database provides signal region labels generated by an automated expert system [1 on page 22-0 ]. The added labels make it possible to use the data to train a deep learning network.

### Load, Resample, and Import Data into Signal Labeler

The signals labeled in this example are from the MIT-BIH Arrhythmia Database [4 on page 22-0 ]. Each signal in the database was irregularly sampled at a mean rate of 360 Hz and was annotated by two cardiologists, allowing for verification of the results.

Load two MIT database signals, corresponding to records 200 and 203. Resample the signals to a uniform grid with a sample time of 1/250 second, which corresponds to the nominal sample rate of the QT Database data.

```
load mit200
y200 = resample(ecgsig,tm,250);

load mit203
y203 = resample(ecgsig,tm,250);
```

Open Signal Labeler. On the **Labeler** tab, click **Import** ▼ and select `From workspace` in the **Members** list. In the dialog box, select the signals `y200` and `y203`. Add time information: Select `Time` from the drop-down list and specify a sample rate of `250` Hz. Click **Import** and close the dialog box. The signals appear in the **Labeled Signal Set Browser**. Plot the signals by selecting the check boxes next to their names.

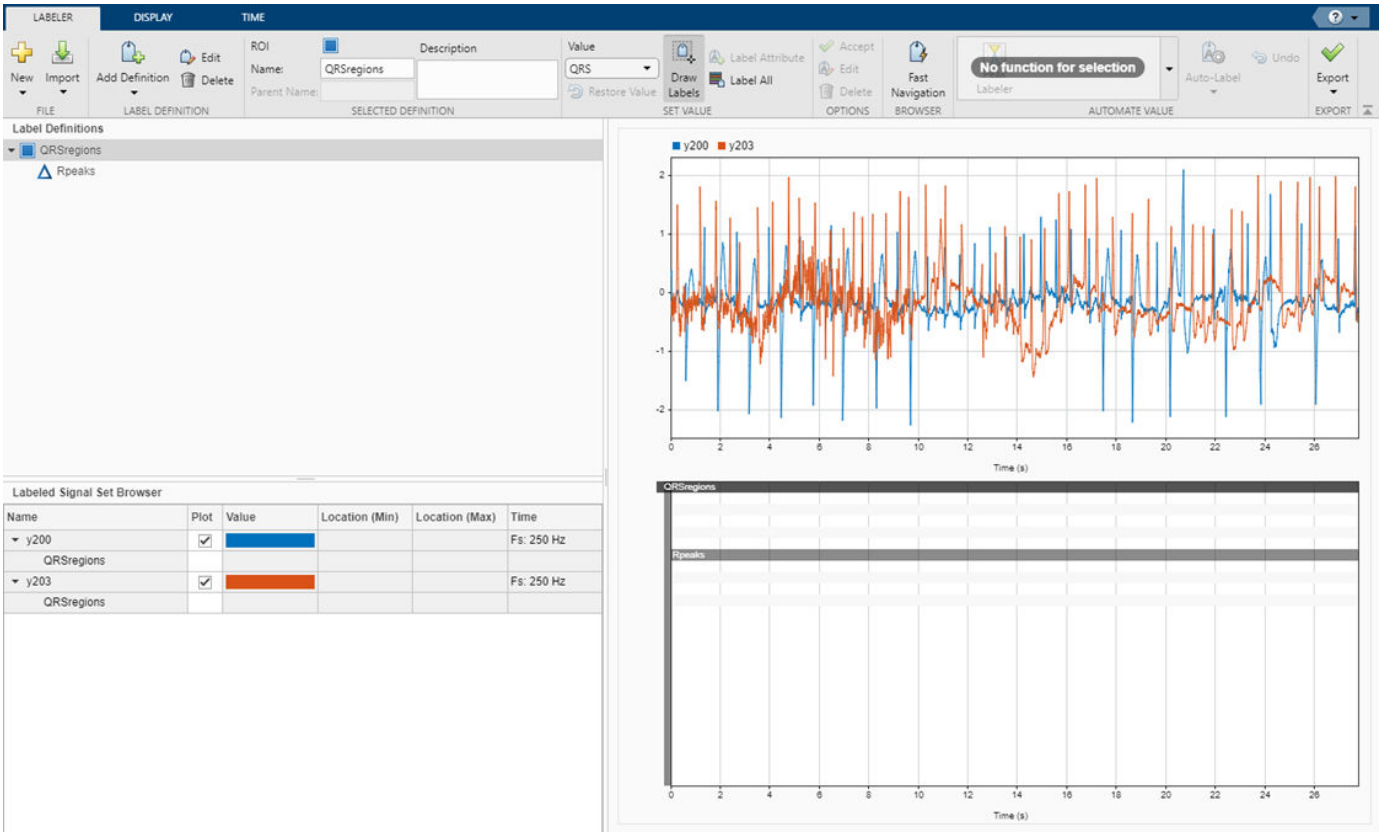
### Define Labels

Define labels to attach to the signals.

1. Define a categorical region-of-interest (ROI) label for the QRS complexes. Click **Add Definition** ▼ on the **Labeler** tab. Specify the **Label Name** as `QRSregions`, select a **Label Type** of `ROI`,

enter the **Data Type** as `categorical`, and add two **Categories,** `QRS` and `n/a`, each on its own line.

2. Define a sublabel of `QRSregions` as a numerical point label for the R peaks. Click `QRSregions` in the **Label Definitions** browser to select it. Click **Add Definition ▼** and select `Add sublabel definition`. Specify the **Label Name** as `Rpeaks`, select a **LabelType** of `Point`, and enter the **Data Type** as `numeric`.

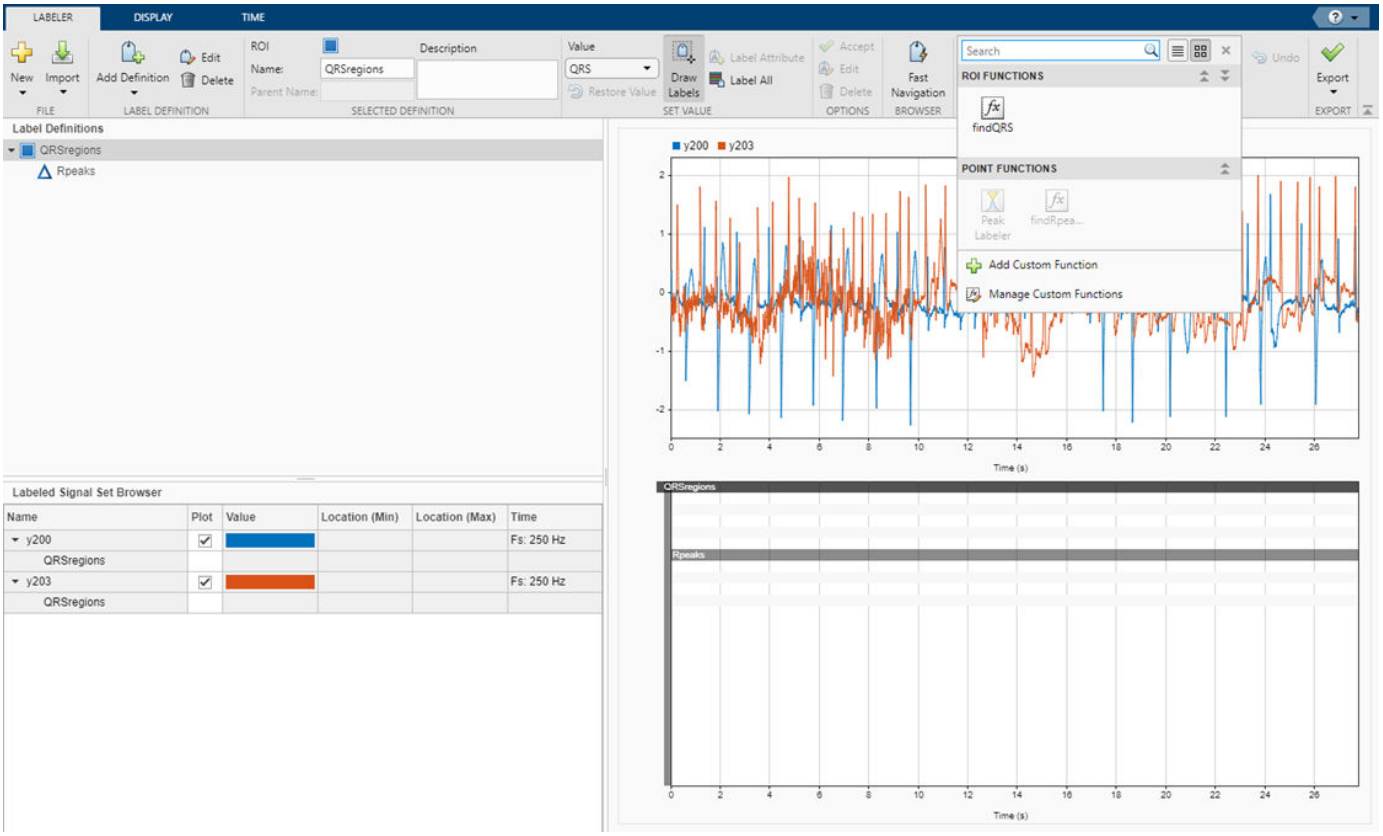



### Create Custom Autolabeling Functions

Create two “Custom Labeling Functions” on page 22-25, one to locate and label the QRS complexes and another to locate and label the R peak within each QRS complex. (Code for the `findQRS` on page 22-0 and `findRpeaks` on page 22-0 functions appears later in the example.) To create each function, in the **Labeler** tab, click **Automate Value ▼** and select **Add Custom Function**. **Signal Labeler** shows a dialog box asking for the name, description, and label type of the function.

1. For the function that locates the QRS complexes, enter `findQRS` in the **Name** field and select `ROI` as the **Label Type**. You can leave the **Description** field empty or you can enter your own description.
2. For the function that locates the R peaks, enter `findRpeaks` in the **Name** field and select `Point` as the **Label Type**. You can leave the **Description** field empty or you can enter your own description.

If you already have written the functions, and the functions are in the current folder or in the MATLAB® path, **Signal Labeler** adds the functions to the gallery. If you have not written the functions, **Signal Labeler** opens blank templates in the Editor for you to type or paste the code. Save the files. Once you save the files, the functions appear in the gallery.

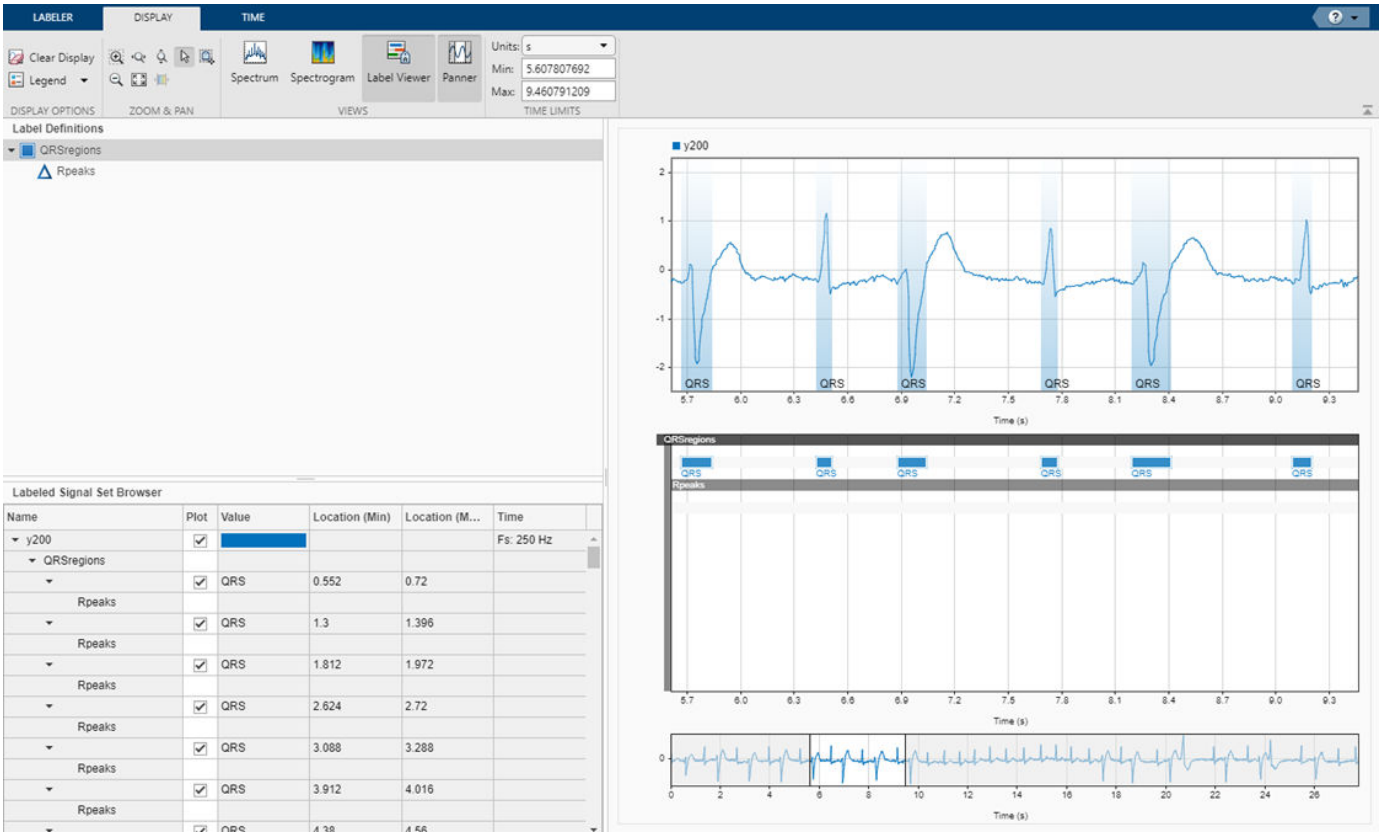



### Label QRS Complexes and R Peaks

Find and label the QRS complexes of the input signals.

1. In the **Labeled Signal Set Browser**, select the check box next to `y200`.
2. Select `QRSregions` in the **Label Definitions** browser.
3. In the **Automate Value** ▼ gallery, select `findQRS`.
4. Click **Auto-Label** ▼ and select `Auto-Label Signals`. In the dialog box that appears, enter the `250` Hz sample rate in the **Arguments** field and click **OK**.

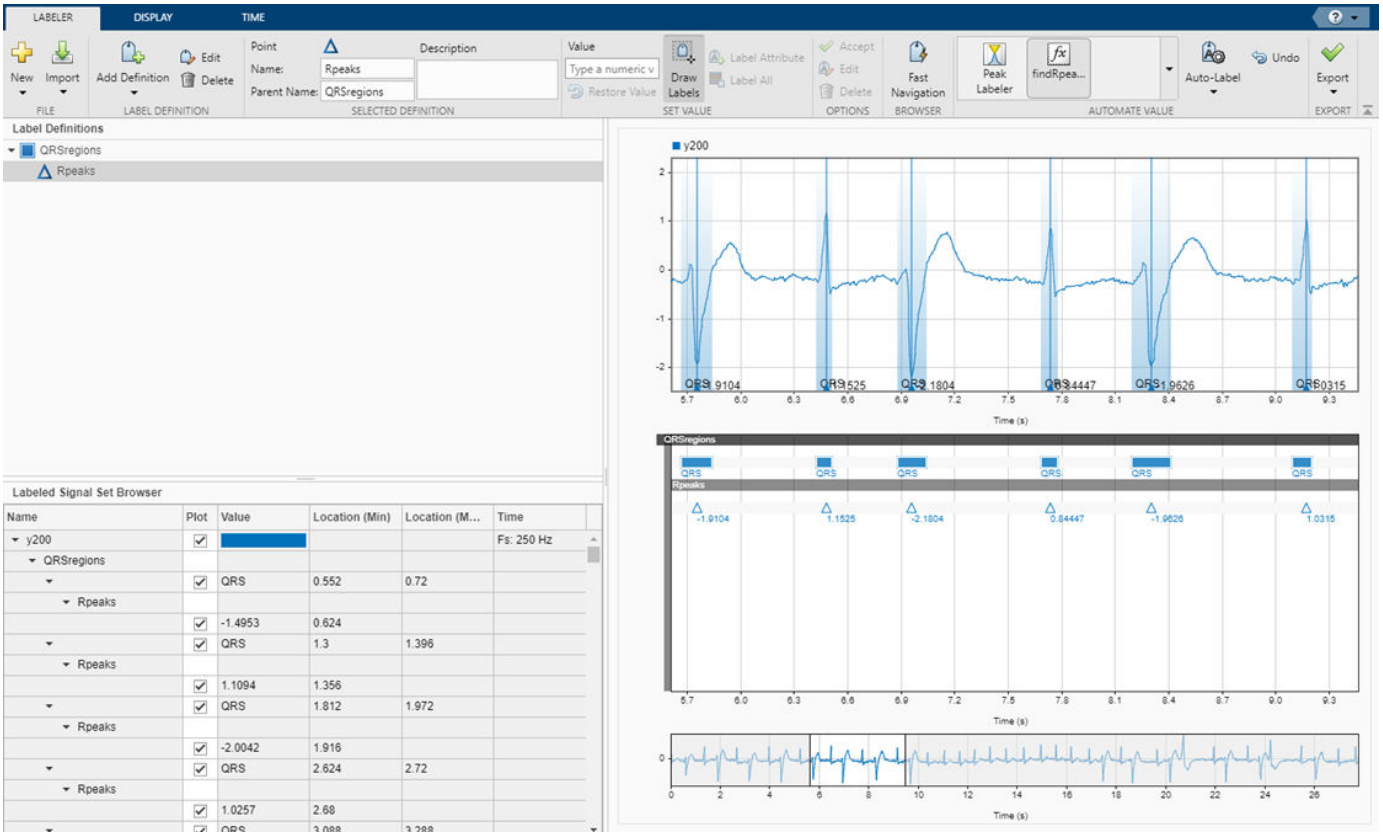
**Signal Labeler** locates and labels the QRS complexes for all signals, but displays labels only for the signals whose check boxes are selected. The QRS complexes appear as shaded regions in the plot and in the label viewer axes. Activate the panner by clicking **Panner** on the **Display** tab and zoom in on a region of the labeled signal.




Find and label the R peaks corresponding to the QRS complexes.

1. Select `Rpeaks` in the **Label Definitions** browser.
2. Go back to the **Labeler** tab. In the **Automate Value** ▼ gallery, select `findRpeaks`.
3. Click **Auto-Label** ▼ and select `Auto-Label Signals`. Click **OK** in the dialog box that appears.

The labels and their numeric values appear in the plot and in the label viewer axes.

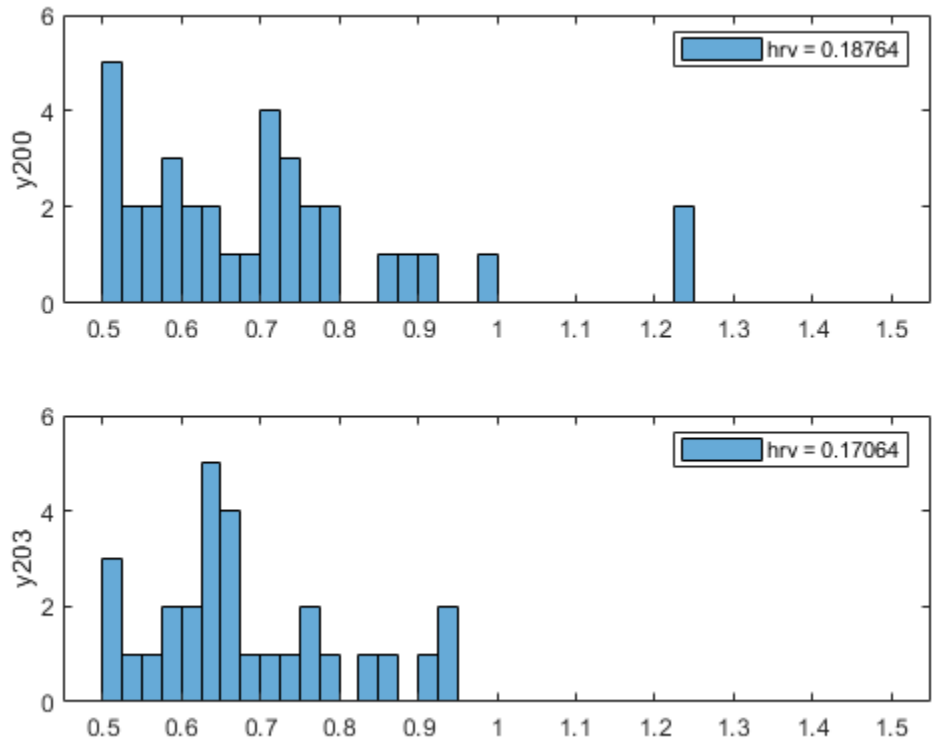



### Export Labeled Signals and Compute Heart-Rate Variability

Export the labeled signals to compare the heart-rate variability for each patient. On the **Labeler** tab, click **Export** ▼ and select `Labeled Signal Set To File`. In the dialog box that appears, give the name `HeartRates.mat` to the labeled signal set and add an optional short description. Click **Export**.

Go back to the MATLAB® Command Window. Load the labeled signal set. For each signal in the set, compute the heart-rate variability as the standard deviation of the time differences between consecutive heartbeats. Plot a histogram of the differences and display the heart-rate variability.

```
load HeartRates

nms = getMemberNames(heartrates);

for k = 1:heartrates.NumMembers

    v = getLabelValues(heartrates,k,{'QRSregions','Rpeaks'});

    hr = diff(cellfun(@(x)x.Location,v));

    subplot(2,1,k)
    histogram(hr,0.5:0.025:1.5)
    legend(['hrv = ' num2str(std(hr))])
    ylabel(nms(k))
    ylim([0 6])

end
```




### `findQRS` Function: Find QRS Complexes

The `findQRS` function finds and labels the QRS complexes of the input signals.

The function uses an auxiliary function, `computeFSST` on page 22-0 , to reshape the input data and compute the Fourier synchrosqueezed transform (FSST). You can either store `computeFSST` in a separate file in the same directory or nest it inside `findQRS` by inserting it before the final `end` statement.

`findQRS` uses the `classify` (Deep Learning Toolbox) function and the trained deep learning network `net` to identify the QRS regions. The deep learning network outputs a categorical array that labels every point of the input signal as belonging to a P region, a QRS complex, a T region, or to none of these. This function converts the point labels corresponding to a QRS complex to QRS region-of-interest labels using `signalMask` and discards the rest. The `df` parameter selects as regions of interest only those QRS complexes whose duration is greater than 20 samples. If you do not specify a sample rate, the function uses a default value of 250 Hz.

```
function [labelVals,labelLocs] = findQRS(x,t,parentLabelVal,parentLabelLoc,varargin)
% This is a template for creating a custom function for automated labeling
%
%  x is a matrix where each column contains data corresponding to a
%  channel. If the channels have different lengths, then x is a cell array
%  of column vectors.
%
%  t is a matrix where each column contains time corresponding to a
%  channel. If the channels have different lengths, then t is a cell array
%  of column vectors.
```

```
%
%  parentLabelVal is the parent label value associated with the output
%  sublabel or empty when output is not a sublabel.
%  parentLabelLoc contains an empty vector when the parent label is an
%  attribute, a vector of ROI limits when parent label is an ROI or a point
%  location when parent label is a point.
%
%  labelVals must be a column vector with numeric, logical or string output
%  values.
%  labelLocs must be an empty vector when output labels are attributes, a
%  two column matrix of ROI limits when output labels are ROIs, or a column
%  vector of point locations when output labels are points.

labelVals = cell(2,1);
labelLocs = cell(2,1);

if nargin<5
    Fs = 250;
else
    Fs = varargin{1};
end

df = 20;

load('trainedQTSegmentationNetwork','net')

for kj = 1:size(x,2)

    sig = x(:,kj);

    % Reshape input and compute Fourier synchrosqueezed transforms

    mitFSST = computeFSST(sig,Fs);

    % Use trained network to predict which points belong to QRS regions

    netPreds = classify(net,mitFSST,'MiniBatchSize',50);

    % Create a signal mask for QRS regions and specify minimum sequence length

    QRS = categorical([netPreds{1} netPreds{2}]',"QRS");
    msk = signalMask(QRS,"MinLength",df,"SampleRate",Fs);
    r = roimask(msk);

    % Label QRS complexes as regions of interest

    labelVals{kj} = r.Value;
    labelLocs{kj} = r.ROILimits;

end

labelVals = vertcat(labelVals{:});
labelLocs = cell2mat(labelLocs);

% Insert computeFSST here if you want to nest it inside findQRS.

end
```

### `computeFSST` Function: Reshape Input and Compute Fourier Synchrosqueezed Transforms

This function uses the `fsst` function to compute the Fourier synchrosqueezed transform (FSST) of the input. As discussed in “Waveform Segmentation Using Deep Learning” on page 24-330, the network performs best when given as input a time-frequency map of each training or testing signal. The FSST results in a set of features particularly useful for recurrent networks because the transform has the same time resolution as the original input. The function:

- Pads the the input data with random numbers and reshapes it into the 2-by-5000 cell array stack expected by `net`.
- Specifies a Kaiser window of length 128 to provide adequate frequency resolution.
- Extracts data over the frequency range from 0.5 Hz to 40 Hz.
- Treats the real and imaginary parts of the FSST as separate features.
- Normalizes the data by subtracting the mean and dividing by the standard deviation.

```
function signalsFsst = computeFSST(xd,Fs)

xd = reshape([xd;randn(10000-length(xd),1)/100],5000,2);
signalsFsst = cell(1,2);

for k = 1:2
    [ss,ff] = fsst(xd(:,k),Fs,kaiser(128));
    sp = ss(ff>0.5 & ff<40,:);
    signalsFsst{k} = normalize([real(sp);imag(sp)],2);
end

end
```

### `findRpeaks` Function: Find R Peaks

This function locates the most prominent peak of the QRS regions of interest found by `findQRS`. The function applies the MATLAB® `islocalmax` function to the absolute value of the signal in the intervals located by `findQRS`.

```
function [labelVals,labelLocs] = findRpeaks(x,t,parentLabelVal,parentLabelLoc,varargin)

labelVals = zeros(size(parentLabelLoc,1),1);
labelLocs = zeros(size(parentLabelLoc,1),1);

for kj = 1:size(parentLabelLoc,1)
    tvals = t>=parentLabelLoc(kj,1) & t<=parentLabelLoc(kj,2);
    ti = t(tvals);
    xi = x(tvals);
    lc = islocalmax(abs(xi),'MaxNumExtrema',1);
    labelVals(kj) = xi(lc);
    labelLocs(kj) = ti(lc);
end

end
```

### References

[1] Laguna, Pablo, Raimon Jané, and Pere Caminal. "Automatic Detection of Wave Boundaries in Multilead ECG Signals: Validation with the CSE Database." *Computers and Biomedical Research*. Vol. 27, No. 1, 1994, pp. 45–60.

[2] Goldberger, Ary L., Luis A. N. Amaral, Leon Glass, Jeffery M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H. Eugene Stanley. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." *Circulation.* Vol. 101, No. 23, 2000, pp. e215–e220. [Circulation Electronic Pages: http://circ.ahajournals.org/content/101/23/e215.full].

[3] Laguna, Pablo, Roger G. Mark, Ary L. Goldberger, and George B. Moody. "A Database for Evaluation of Algorithms for Measurement of QT and Other Waveform Intervals in the ECG." *Computers in Cardiology.* Vol. 24, 1997, pp. 673–676.

[4] Moody, George B., and Roger G. Mark. "The impact of the MIT-BIH Arrhythmia Database." *IEEE Engineering in Medicine and Biology Magazine*. Vol. 20, No. 3, May–June 2001, pp. 45–50.

## See Also

**Apps**
**Signal Analyzer** | **Signal Labeler**

**Objects**
`labeledSignalSet` | `signalLabelDefinition` | `signalMask`

## Related Examples

- “Label Signal Attributes, Regions of Interest, and Points” on page 22-40
- “Examine Labeled Signal Set” on page 22-46
- “Label Spoken Words in Audio Signals Using External API” on page 22-60

## More About

- “Using Signal Labeler App” on page 22-2
- “Import Data into Signal Labeler” on page 22-6
- “Create or Import Signal Label Definitions” on page 22-13
- “Label Signals Interactively or Automatically” on page 22-17
- “Custom Labeling Functions” on page 22-25
- “Customize Labeling View” on page 22-30
- “Export Labeled Signal Sets and Signal Label Definitions” on page 22-36
- “Signal Labeler Usage Tips” on page 22-38

# Label Spoken Words in Audio Signals Using External API

This example shows how to label spoken words in **Signal Labeler**. The example uses the IBM® Watson Speech to Text API and Audio Toolbox™ software. See “Speech-to-Text Transcription” (Audio Toolbox) for instructions about:

1. Downloading the Audio Toolbox `speech2text` extended functionality, available from MATLAB® Central.
2. Setting up the IBM Watson Speech API, offered through IBM Cloud Services. You must create an IBM Cloud account, a Speech to Text service instance, and go to the service dashboard and copy your credentials – API Key and URL values. See the Getting Started Tutorial in the IBM documentation for more details.

### Load Speech Data

Load an audio data file containing the sentence "Oak is strong, and also gives shade" spoken by a male voice. The signal is sampled at 44.1 kHz.

```
[y,fs] = audioread('oak.m4a');

% To hear, type soundsc(y,fs)
```

1. Open Signal Labeler. On the **Labeler** tab, click **Import** ▼ and select `From workspace` in the **Members** list. In the dialog box, select the signal, `y`.
2. Add time information: Select `Time` from the drop-down list and specify `fs` as the sample rate, which is measured in kHz.
3. Close the dialog box. The signal appears in the **Labeled Signal Set Browser**.

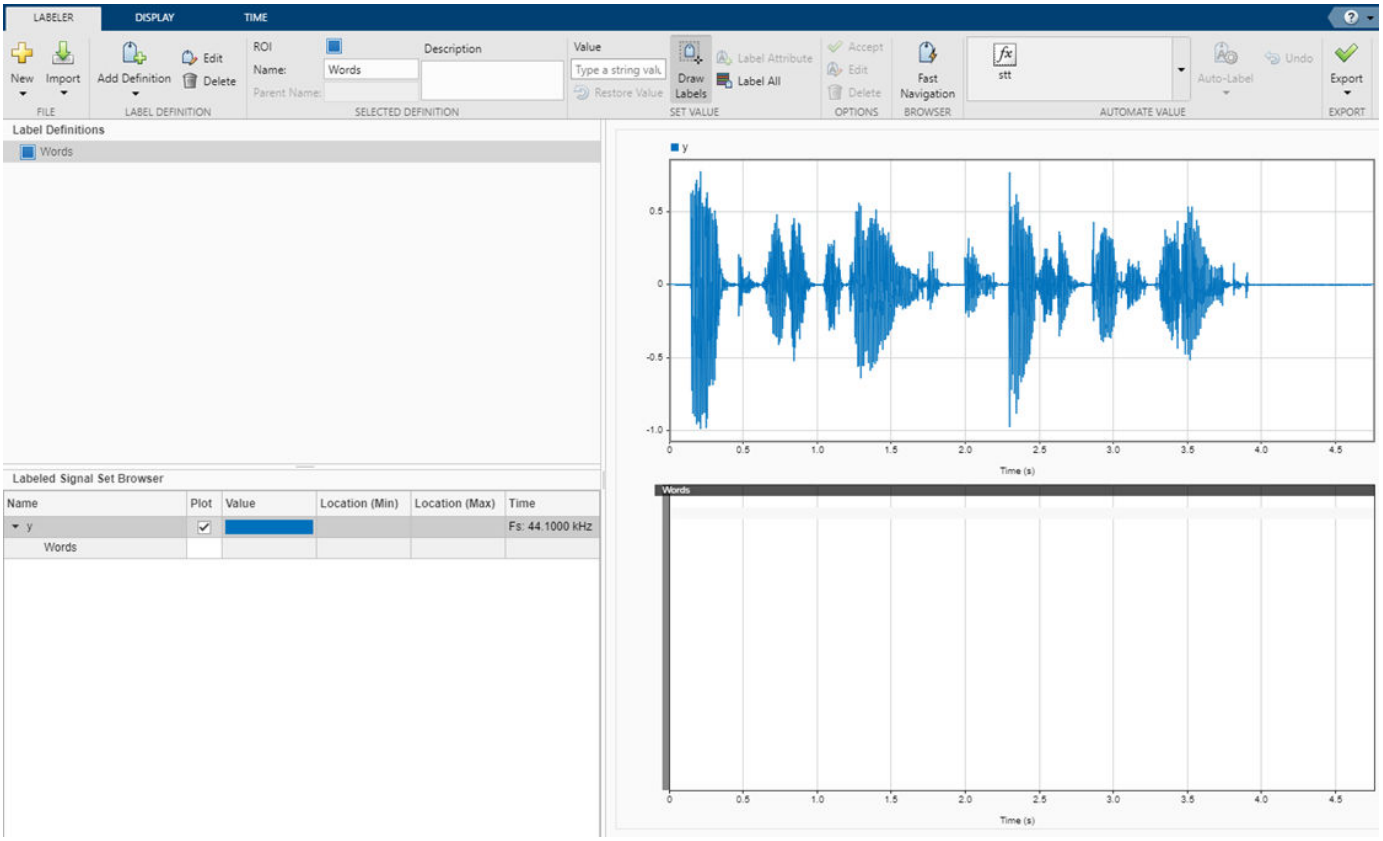
### Define Label

Define a label to attach to the signal. Click **Add Definition** ▼ on the **Labeler** tab. Specify the **Label Name** as `Words`, select a **Label Type** of `ROI`, and enter the **Data Type** as `string`.

### Create Custom Autolabeling Function

Create a custom function to label the words spoken in the audio file. (Code for the `stt` on page 22-0 function appears later in the example.) See “Custom Labeling Functions” on page 22-25 for more information.

1. Go to the directory where you have stored the `speech2text` P-code files and the JSON file that stores your IBM Cloud credentials.
2. To create the function, in the **Labeler** tab, click **Automate Value** ▼ and select **Add Custom Function**. **Signal Labeler** shows a dialog box asking you to enter the name, description, and label type of the function you are adding. Enter `stt` in the **Name** field and select `ROI` as the **Label Type**. You can leave the **Description** field empty or you can enter your own description.
3. Copy the function code and paste it in the empty template that appears. Save the file. The function appears in the gallery.

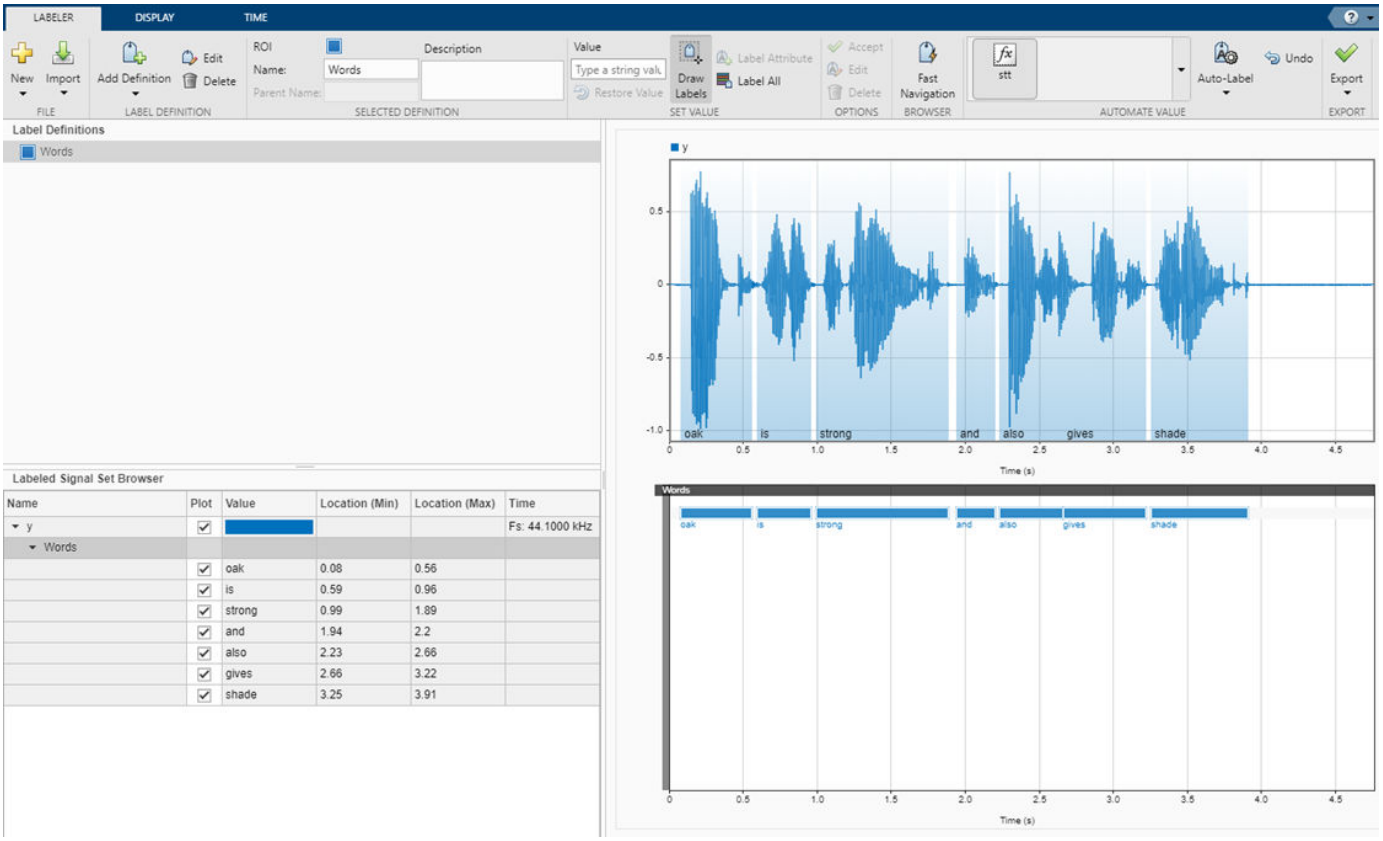



### Locate and Identify Spoken Words

Locate and identify the words spoken in the input signal.

1. In the **Labeled Signal Set Browser**, select the check box next to `y`.
2. Select `Words` in the **Label Definitions** browser.
3. On the **Automated Value** gallery, select `stt`.
4. Click **Auto-Label** ▼ and select `Auto-Label Signals`. Click **OK** in the dialog box that appears.

**Signal Labeler** locates and labels the spoken words.




### Export Labeled Signal

Export the labeled signal. On the **Labeler** tab, click **Export** ▼ and select `Labeled Signal Set To File`. In the dialog box that appears, give the name `Transcription.mat` to the labeled signal set and add an optional short description. Click **Export**.

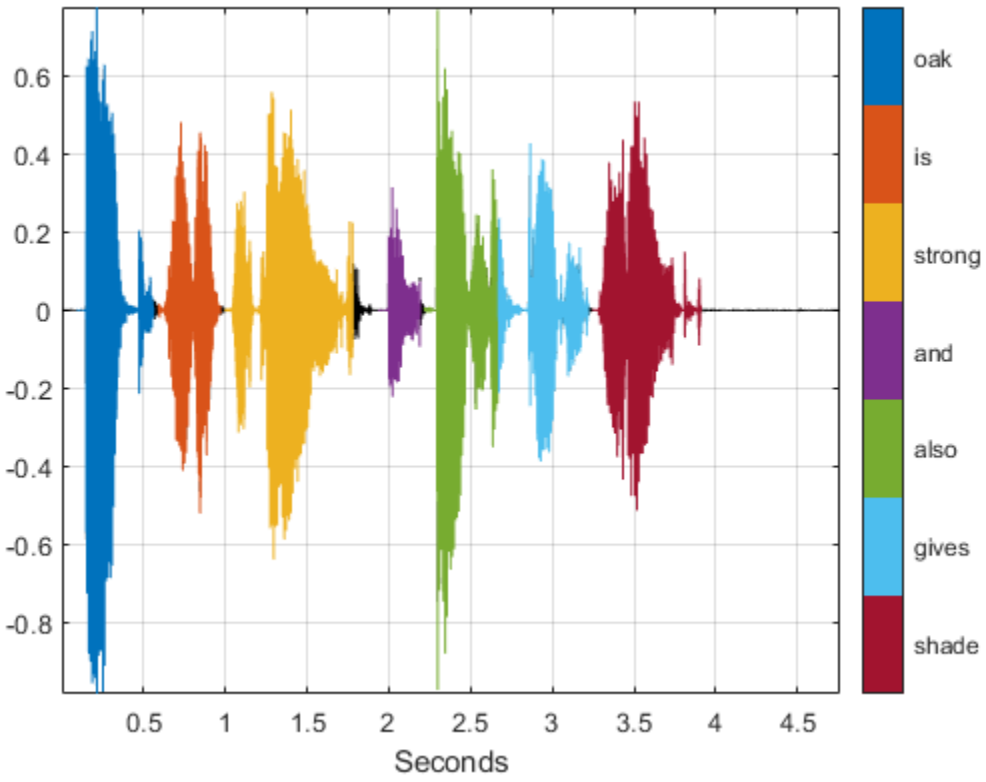
Go back to the MATLAB® Command Window. Load the labeled signal set. The set has only one member. Get the names of the labels, and use the name to obtain and display the transcribed words.

```
load Transcription

ln = getLabelNames(transcribedAudio);

v = getLabelValues(transcribedAudio,1,ln)
```

```
v=7×2 table
     ROILimits       Value
    ____________    _______

    0.09    0.56    "oak"
    0.59    0.97    "is"
       1    1.78    "strong"
    1.94    2.19    "and"
    2.22    2.67    "also"
    2.67    3.22    "gives"
    3.25    3.91    "shade"
```

Change the label values from strings to categories. Use a `signalMask` object to plot the signal using a different color for each word.

```
v.Value = categorical(v.Value,v.Value);

msk = signalMask(v,'SampleRate',fs);

s = getSignal(transcribedAudio,1);

plotsigroi(msk,s.y)
```




### `stt` Function: Locate and Identify Spoken Words

This function uses the IBM Watson Speech API and the Audio Toolbox `speech2text` extended functionality to extract spoken words from an audio file.

```
function [labelVals,labelLocs] = stt(x,t,parentLabelVal,parentLabelLoc,varargin)

aspeechObjectIBM = speechClient('IBM','timestamps',true);

fs = 1/(t(2)-t(1));

tixt = speech2text(aspeechObjectIBM,x,fs);

numLabels = numel(tixt.TimeStamps{:});
labelVals = strings(numLabels,1);
labelLocs = zeros(numLabels,2);
```

```
for idx = 1:numLabels
    labelVals(idx) = tixt.TimeStamps{:}{idx}{1};
    labelLocs(idx,1) = tixt.TimeStamps{:}{idx}{2};
    labelLocs(idx,2) = tixt.TimeStamps{:}{idx}{3};
end

end
```

## See Also

**Apps**
**Signal Analyzer** | **Signal Labeler**

**Objects**
`labeledSignalSet` | `signalLabelDefinition` | `signalMask`

## Related Examples

- “Label Signal Attributes, Regions of Interest, and Points” on page 22-40
- “Examine Labeled Signal Set” on page 22-46
- “Label QRS Complexes and R Peaks of ECG Signals Using Deep Learning Network” on page 22-51

## More About

- “Using Signal Labeler App” on page 22-2
- “Import Data into Signal Labeler” on page 22-6
- “Create or Import Signal Label Definitions” on page 22-13
- “Label Signals Interactively or Automatically” on page 22-17
- “Custom Labeling Functions” on page 22-25
- “Customize Labeling View” on page 22-30
- “Export Labeled Signal Sets and Signal Label Definitions” on page 22-36
- “Signal Labeler Usage Tips” on page 22-38

# Choose an App to Label Ground Truth Data

You can use Computer Vision Toolbox™, Automated Driving Toolbox™, Lidar Toolbox™, and Signal Processing 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, 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.
- 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. To perform audio-specific tasks, such as speech detection, speech-to-text transcription, and recording new audio, use the **Audio Labeler** app.

This table summarizes the key features of the labeling apps.

| Labeling App | Data Sources | Label Support | Automation | Additional Features |
|---|---|---|---|---|
| **Image Labeler** | • Image collections | • Rectangle regions of interest (ROIs)<br>• Projected cuboid (ROIs)<br>• Line ROIs<br>• Pixel ROIs<br>• Sublabels<br>• Attributes<br>• Scenes | • Built-in automation algorithms<br>• Custom automation algorithms | • View visual summary of labeled data |

| Labeling App | Data Sources | Label Support | Automation | Additional Features |
|---|---|---|---|---|
| **Video Labeler** | • Videos<br>• Image sequences<br>• Custom image data sources | • Rectangle ROIs<br>• Projected cuboid (ROIs)<br>• Line ROIs<br>• Pixel ROIs<br>• Sublabels<br>• Attributes<br>• Scenes | • Built-in automation algorithms<br>• Custom automation algorithms<br>• Temporal automation algorithms | • View visual summary of labeled data |
| **Ground Truth Labeler** | • Videos<br>• Image sequences<br>• Custom image data sources<br>• Point cloud sequences (PCD or PLY files)<br>• Velodyne® lidar files<br>• Rosbags (requires ROS Toolbox) | • Rectangle ROIs<br>• Projected cuboid (ROIs)<br>• Cuboid ROIs<br>• Line ROIs<br>• Pixel ROIs<br>• Sublabels<br>• Attributes<br>• Scenes | • Built-in automation algorithms, including vehicle and lane detection algorithms and a point cloud temporal interpolation algorithm<br>• Custom automation algorithms<br>• Temporal automation algorithms | • View visual summary of labeled data<br>• Connect external tool to app for displaying time-synchronized signals, such as lidar or CAN bus data<br>• Customize loading interface to support additional data sources |
| **Lidar Labeler** | • Point cloud sequences (PCD or PLY files)<br>• Velodyne lidar files<br>• LAS/LAZ file sequences<br>• Rosbags (requires ROS Toolbox) | • Cuboid ROIs<br>• Attributes<br>• Scenes | • Built-in automation algorithms, including a lidar object tracker and point cloud temporal interpolator<br>• Custom automation algorithms<br>• Temporal automation algorithms | • View the cuboid labels in top, side, and front views<br>• Save and reuse custom camera views<br>• Connect to external tool to display time-synchronized signals for ease of labeling, such as videos, to use as a reference while labeling |

| Labeling App | Data Sources | Label Support | Automation | Additional Features |
|---|---|---|---|---|
| **Signal Labeler** | • Numeric arrays, MATLAB timetables, and `labeledSignalSet` objects in the MATLAB workspace<br>• MAT-files and CSV files | • Time-based ROIs<br>• Time-based points<br>• Attributes<br>• Sublabels | • Built-in peak labeling<br>• Custom automation algorithms | • Expand, collapse, and browse details of labeled data<br>• View signal spectra and spectrograms<br>• Label ROIs and points using the spectrogram<br>• Label signals in bulk<br>• Use Label Viewer to view and compare labels |
| **Audio Labeler** | • Audio files (WAVE, OGG, FLAC, AU, AIFF, AIFC, MP3, MPEG-4 AAC)<br>• `labeledSignalSet` objects in the MATLAB workspace or in MAT-files | • Time-based ROIs<br>• File-level labels | • Speech detection<br>• Speech-to-text transcription (requires Audio Toolbox™ extended functionality for speech2text) | • Audio playback<br>• Audio recording<br>• Inspect audio file information |

## 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 the Ground Truth Labeler” (Automated Driving Toolbox)
- “Get Started with the Lidar Labeler” (Lidar Toolbox)
- “Using Signal Labeler App” on page 22-2
- “Label Audio Using Audio Labeler” (Audio Toolbox)

# Common Applications

# Create Uniform and Nonuniform Time Vectors

You can create uniform and nonuniform time vectors for use in computations involving time series.

Use the colon operator if you know the sampling frequency. If your system samples time at a rate of 15 Hz during one second, you get 16 readings, including the one at zero.

```
Fs = 15;
Ts = 1/Fs;
ts = 0:Ts:1;
```

Use `linspace` if you know the beginning and end of the time interval and the number of samples. Suppose you start a stopwatch and stop it one second later. If you know your instrument took 15 readings, you can generate the time vector.

```
tl = linspace(0,1,15);
```

You can compute the sample rate directly from the samples and use it to reconstruct the time vector.

```
sf = 1/(tl(2)-tl(1));

TL = (0:length(tl)-1)/sf;

ErrorTL = max(abs(tl-TL))
```

```
ErrorTL = 0
```

You can also reconstruct `ts` using `linspace`.

```
lts = length(ts);
TS = linspace(ts(1),ts(lts),lts);

ErrorTS = max(abs(ts-TS))
```

```
ErrorTS = 1.1102e-16
```

`linspace` and the colon operator create row vectors by default. Transpose them to obtain column vectors.

```
tcol = tl';
ttrans = ts';
```

Combine `linspace` and the colon operator to generate nonuniform time vectors of arbitrary characteristics.

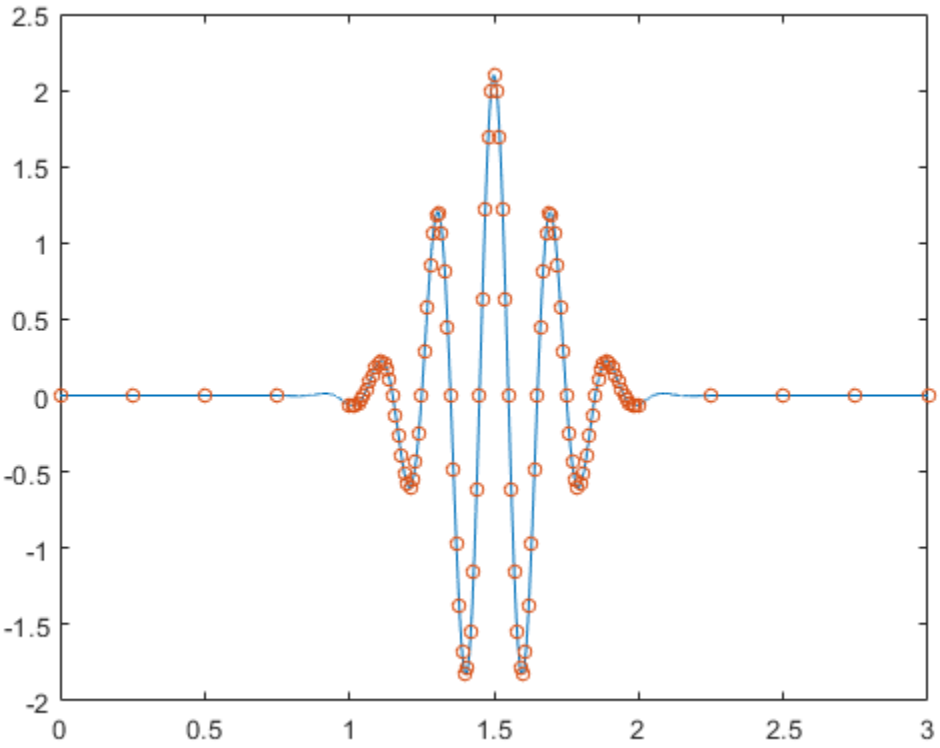
Suppose you have a Gaussian-modulated sinusoidal pulse that you must sample. The pulse changes rapidly during a one-second interval but slowly during the preceding and following seconds.

Sample the region of interest at 100 Hz and take only five samples before and after. Concatenate the vectors using square brackets.

```
gpl = @(x) 2.1*gauspuls(x-1.5,5,0.4);

Ffast = 100;
Tf = 1/Ffast;
Nslow = 5;
tdisc = [linspace(0,1,Nslow) 1+Tf:Tf:2-Tf linspace(2,3,Nslow)];
```

Generate 20001 samples of the function to simulate the continuous-time pulse. Overlay a plot of the samples defined by `tsf`.

```
Tcont = linspace(0,3,20001)';

plot(Tcont,gpl(Tcont),tdisc,gpl(tdisc),'o','markersize',5)
```




## See Also

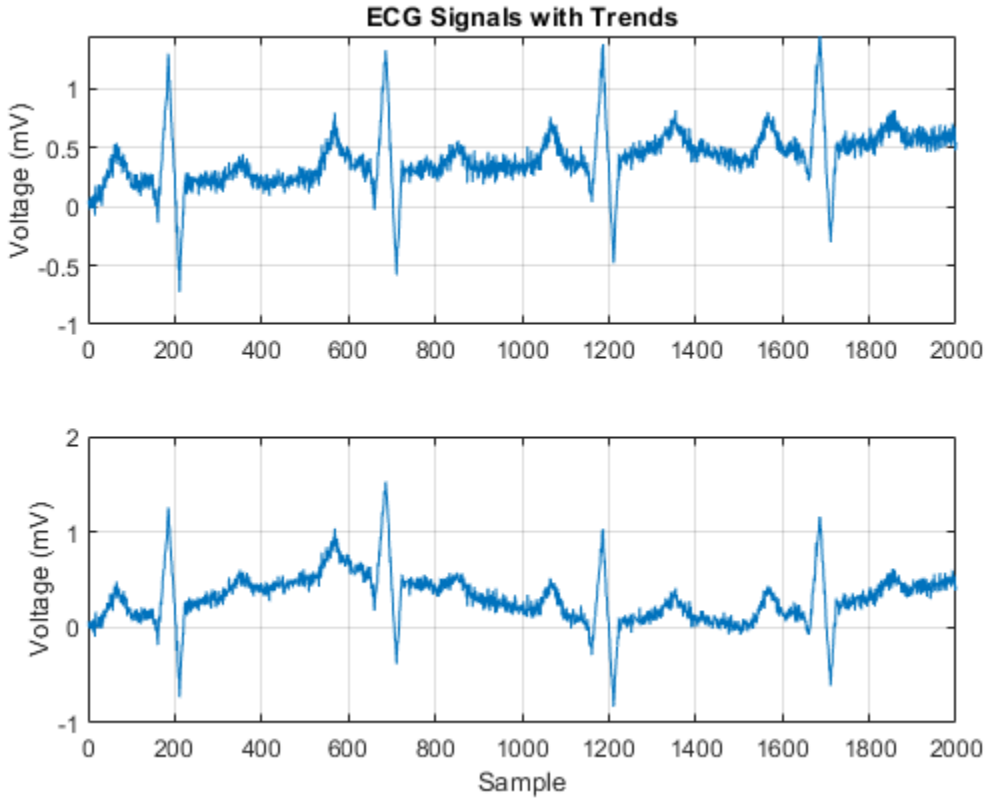
`gauspuls`

# Remove Trends from Data

Measured signals can show overall patterns that are not intrinsic to the data. These trends can sometimes hinder the data analysis and must be removed.

Consider two electrocardiogram (ECG) signals with different trends. ECG signals are sensitive to disturbances such as power source interference. Load the signals and plot them.

```
load('ecgSignals.mat')

t = (1:length(ecgl))';

subplot(2,1,1)
plot(t,ecgl), grid
title 'ECG Signals with Trends', ylabel 'Voltage (mV)'

subplot(2,1,2)
plot(t,ecgnl), grid
xlabel Sample, ylabel 'Voltage (mV)'
```

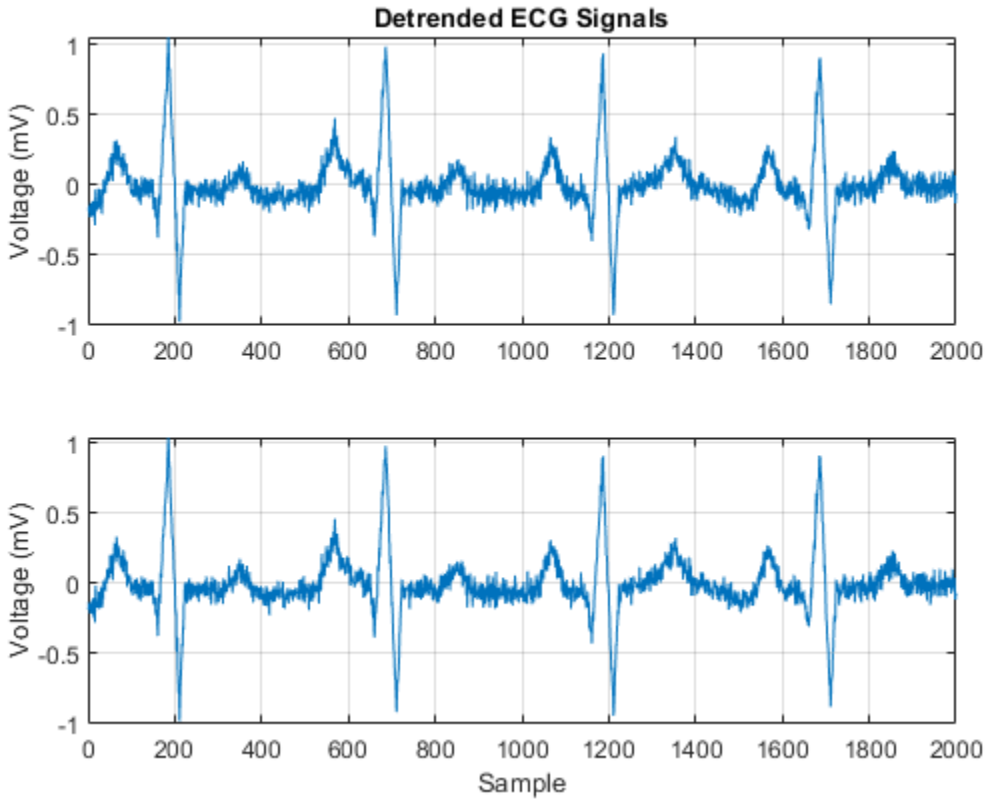



The signal on the first plot shows a linear trend. The trend on the second signal is nonlinear. To eliminate the linear trend, use the MATLAB® function `detrend`.

```
dt_ecgl = detrend(ecgl);
```

To eliminate the nonlinear trend, fit a low-order polynomial to the signal and subtract it. In this case, the polynomial is of order 6. Plot the two new signals.

```
opol = 6;
[p,s,mu] = polyfit(t,ecgnl,opol);
f_y = polyval(p,t,[],mu);

dt_ecgnl = ecgnl - f_y;

subplot(2,1,1)
plot(t,dt_ecgl), grid
title 'Detrended ECG Signals', ylabel 'Voltage (mV)'

subplot(2,1,2)
plot(t,dt_ecgnl), grid
xlabel Sample, ylabel 'Voltage (mV)'
```




The trends have been effectively removed. Observe how the signals do not show a baseline shift anymore. They are ready for further processing.

## See Also

`detrend` | `polyfit` | `polyval`

## Related Examples

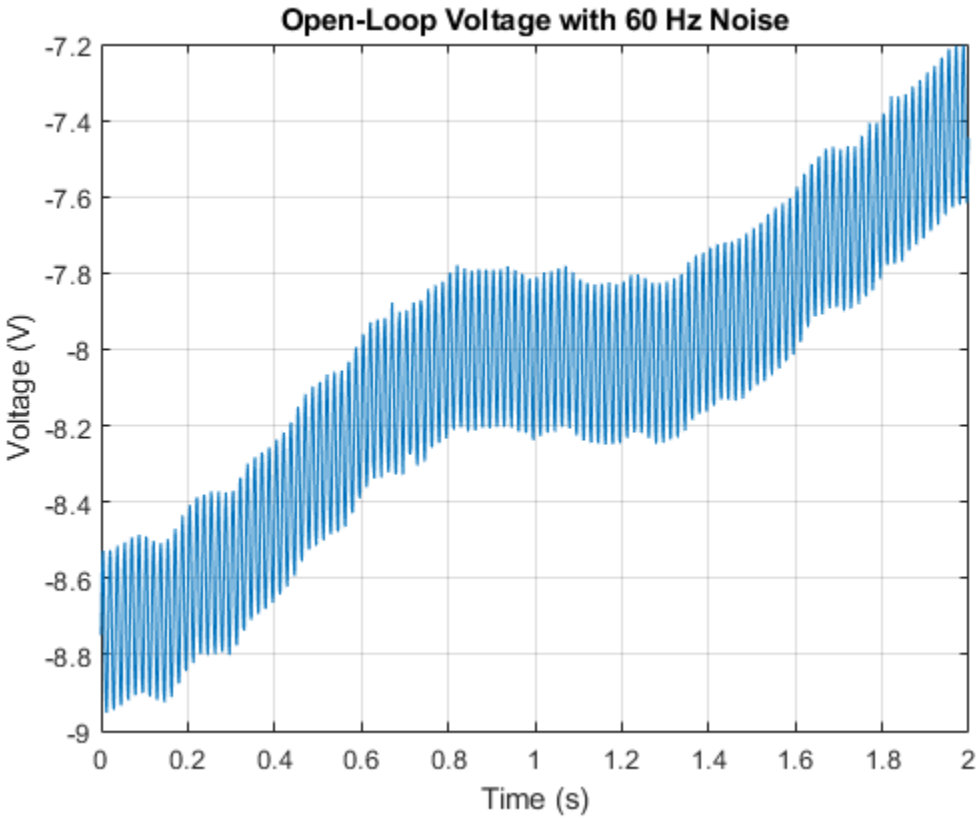
- “Peak Analysis” on page 24-60

# Remove the 60 Hz Hum from a Signal

Alternating current in the United States and several other countries oscillates at a frequency of 60 Hz. Those oscillations often corrupt measurements and have to be subtracted.

Study the open-loop voltage across the input of an analog instrument in the presence of 60 Hz power-line noise. The voltage is sampled at 1 kHz.

```
load openloop60hertz, openLoop = openLoopVoltage;

Fs = 1000;
t = (0:length(openLoop)-1)/Fs;

plot(t,openLoop)
ylabel('Voltage (V)')
xlabel('Time (s)')
title('Open-Loop Voltage with 60 Hz Noise')
grid
```

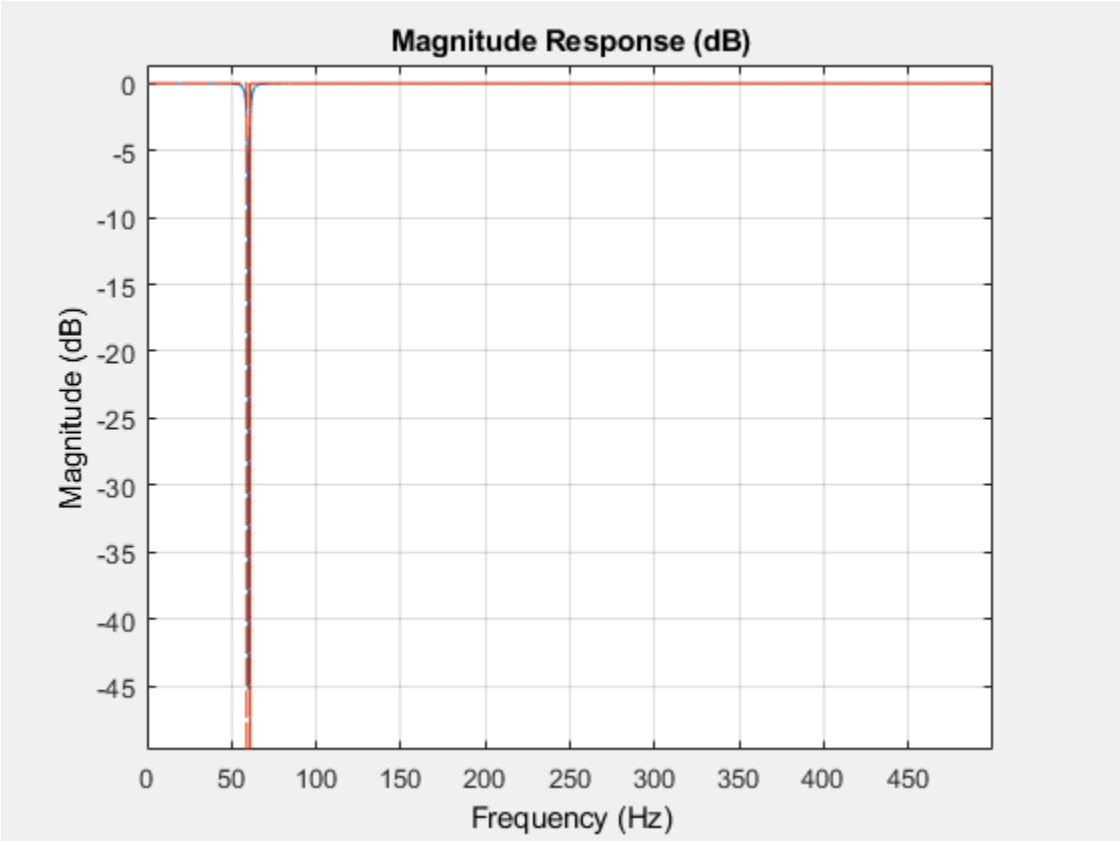



Eliminate the 60 Hz noise with a Butterworth notch filter. Use `designfilt` to design it. The width of the notch is defined by the 59 to 61 Hz frequency interval. The filter removes at least half the power of the frequency components lying in that range.

```
d = designfilt('bandstopiir','FilterOrder',2, ...
               'HalfPowerFrequency1',59,'HalfPowerFrequency2',61, ...
               'DesignMethod','butter','SampleRate',Fs);
```

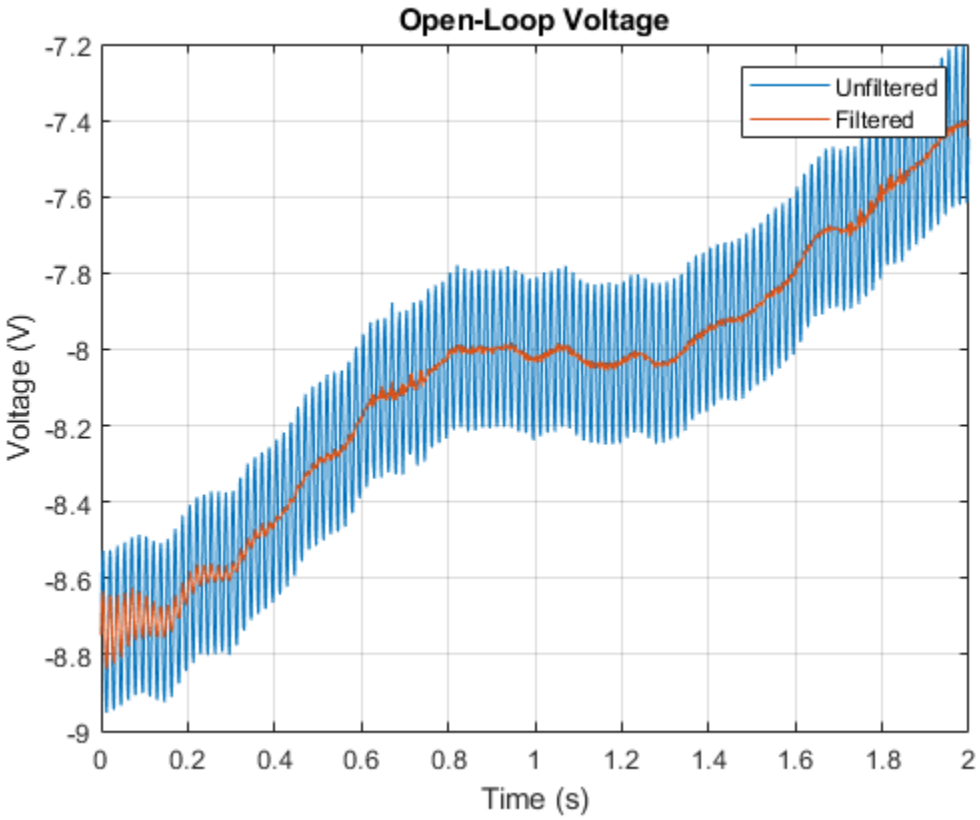
Plot the frequency response of the filter. Note that this notch filter provides up to 45 dB of attenuation.

```
fvtool(d,'Fs',Fs)
```



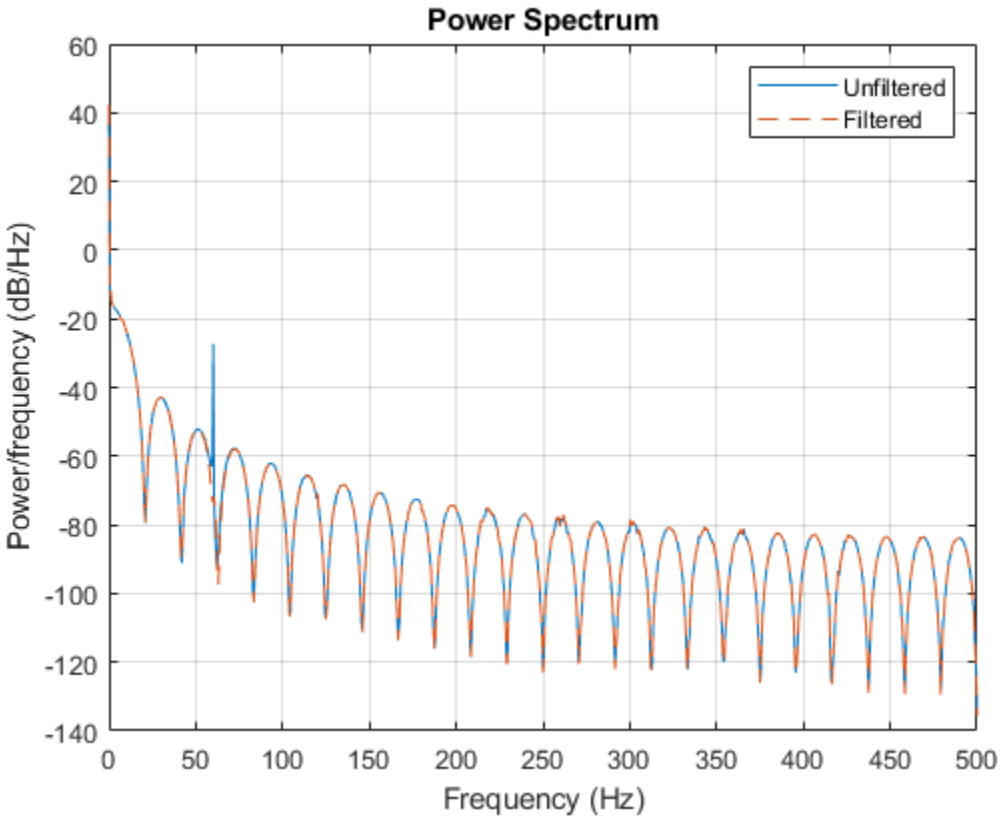

Filter the signal with `filtfilt` to compensate for filter delay. Note how the oscillations decrease significantly.

```
buttLoop = filtfilt(d,openLoop);

plot(t,openLoop,t,buttLoop)
ylabel('Voltage (V)')
xlabel('Time (s)')
title('Open-Loop Voltage')
legend('Unfiltered','Filtered')
grid
```




Use the periodogram to see that the "spike" at 60 Hz has been eliminated.

```
[popen,fopen] = periodogram(openLoop,[],[],Fs);
[pbutt,fbutt] = periodogram(buttLoop,[],[],Fs);

plot(fopen,20*log10(abs(popen)),fbutt,20*log10(abs(pbutt)),'--')
ylabel('Power/frequency (dB/Hz)')
xlabel('Frequency (Hz)')
title('Power Spectrum')
legend('Unfiltered','Filtered')
grid
```




## See Also

**FVTool** | `designfilt` | `filtfilt` | `periodogram`

## Related Examples

- “Signal Smoothing” on page 24-10

# Remove Spikes from a Signal

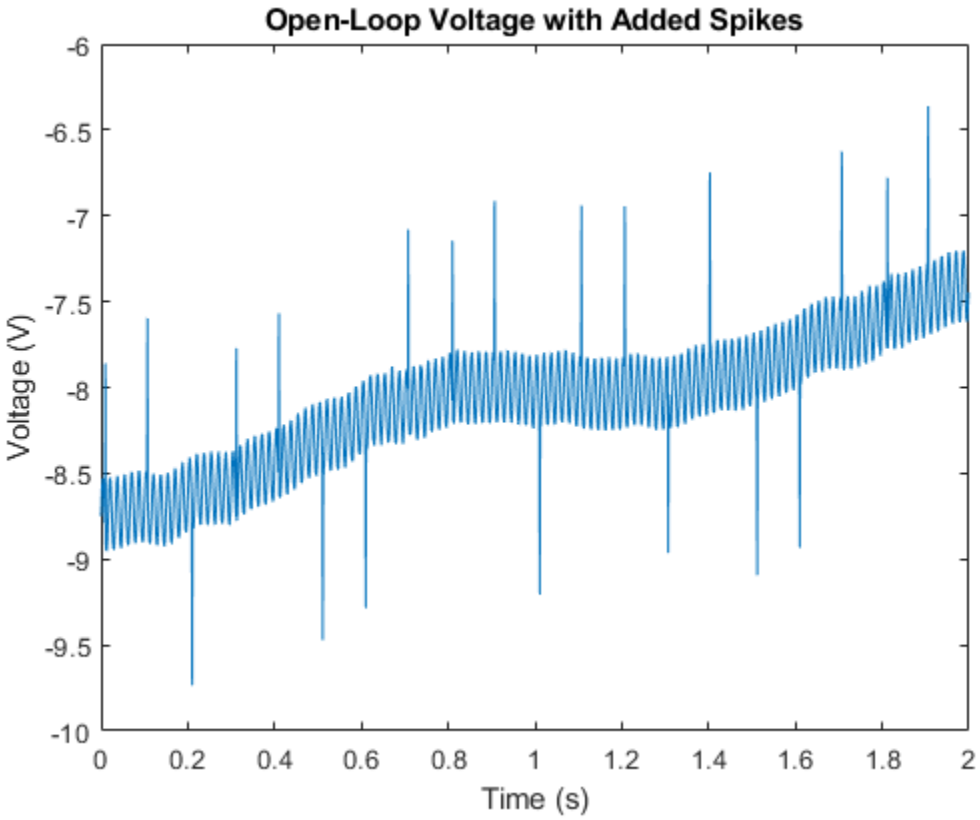
Sometimes data exhibit unwanted transients, or spikes. Median filtering is a natural way to eliminate them.

Consider the open-loop voltage across the input of an analog instrument in the presence of 60 Hz power-line noise. The sample rate is 1 kHz.

```
load openloop60hertz

fs = 1000;
t = (0:numel(openLoopVoltage) - 1)/fs;
```

Corrupt the signal by adding transients with random signs at random points. Reset the random number generator for reproducibility.

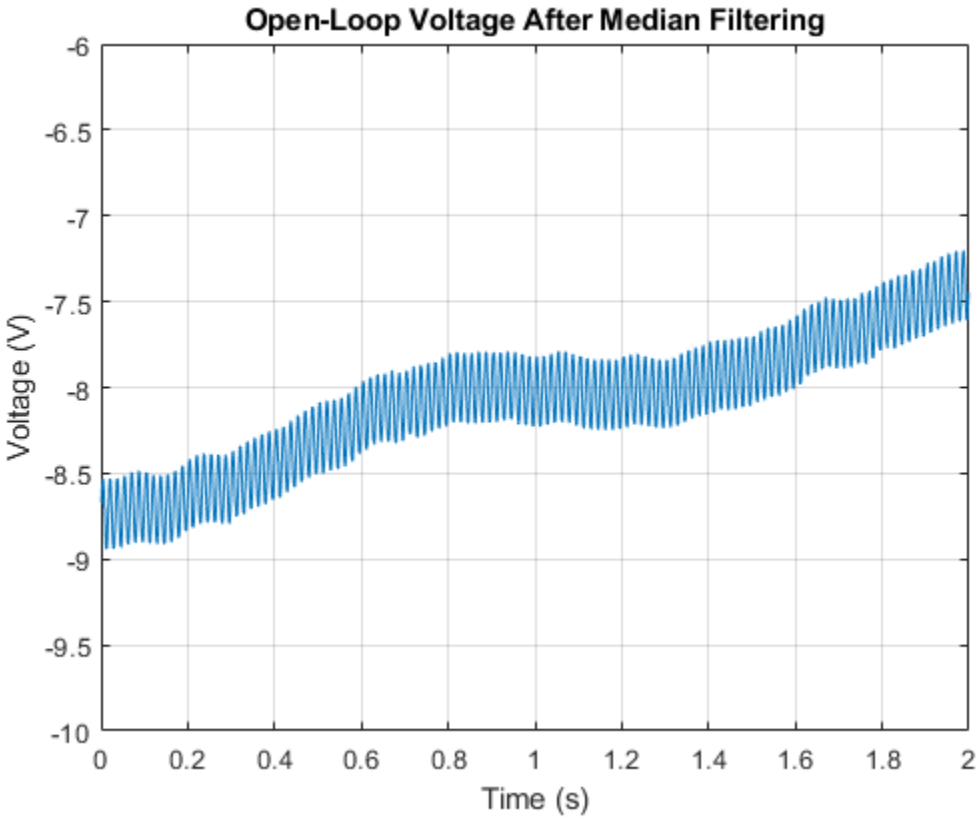
```
rng default

spikeSignal = zeros(size(openLoopVoltage));
spks = 10:100:1990;
spikeSignal(spks+round(2*randn(size(spks)))) = sign(randn(size(spks)));

noisyLoopVoltage = openLoopVoltage + spikeSignal;

plot(t,noisyLoopVoltage)

xlabel('Time (s)')
ylabel('Voltage (V)')
title('Open-Loop Voltage with Added Spikes')
```




```
yax = ylim;
```

The function `medfilt1` replaces every point of a signal by the median of that point and a specified number of neighboring points. Accordingly, median filtering discards points that differ considerably from their surroundings. Filter the signal, using sets of three neighboring points to compute the medians. Note how the spikes vanish.

```
medfiltLoopVoltage = medfilt1(noisyLoopVoltage,3);

plot(t,medfiltLoopVoltage)

xlabel('Time (s)')
ylabel('Voltage (V)')
title('Open-Loop Voltage After Median Filtering')
ylim(yax)
grid
```




## See Also

`medfilt1`

## Related Examples

- “Signal Smoothing” on page 24-10

# Process a Signal with Missing Samples

Consider the weight of a person as recorded (in pounds) during the leap year 2012. The person did not record their weight every day. You would like to study the periodicity of the signal, but before you can do so you must take care of the missing data.

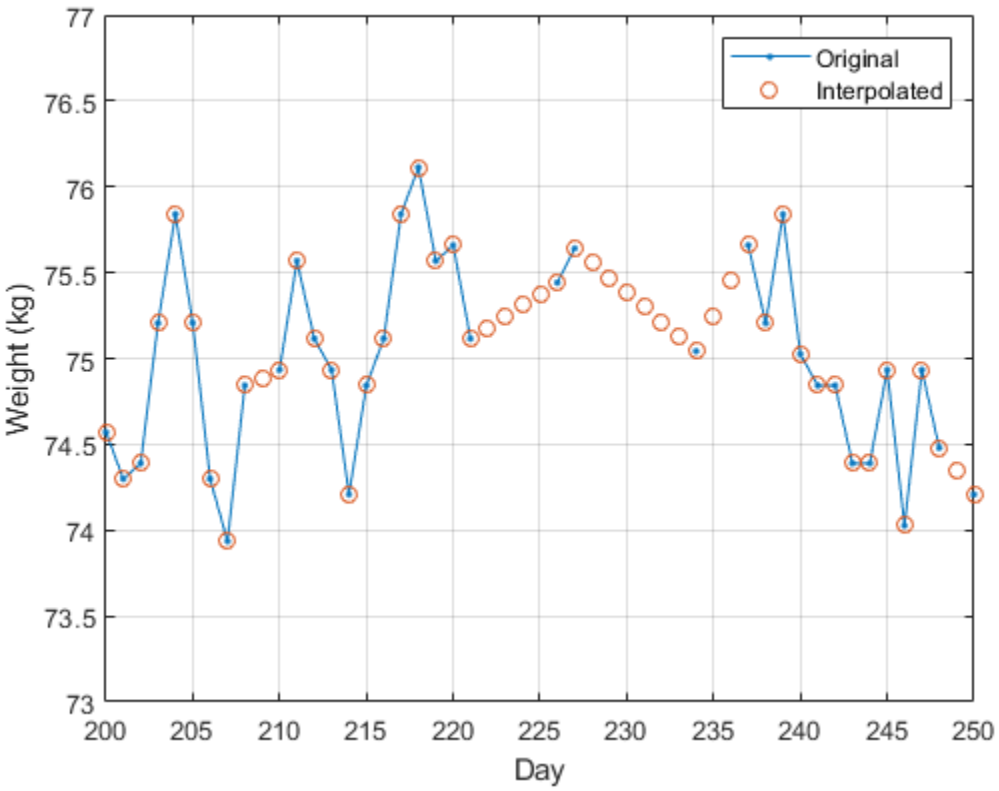
Load the data and convert the measurements to kilograms. Missed readings are set to `NaN`. Determine how many points are missing.

```
load('weight2012.dat')

wgt = weight2012(:,2)/2.20462;
daynum = 1:length(wgt);
missing = isnan(wgt);

fprintf('Missing %d samples of %d\n',sum(missing),max(daynum))
```

```
Missing 27 samples of 366
```

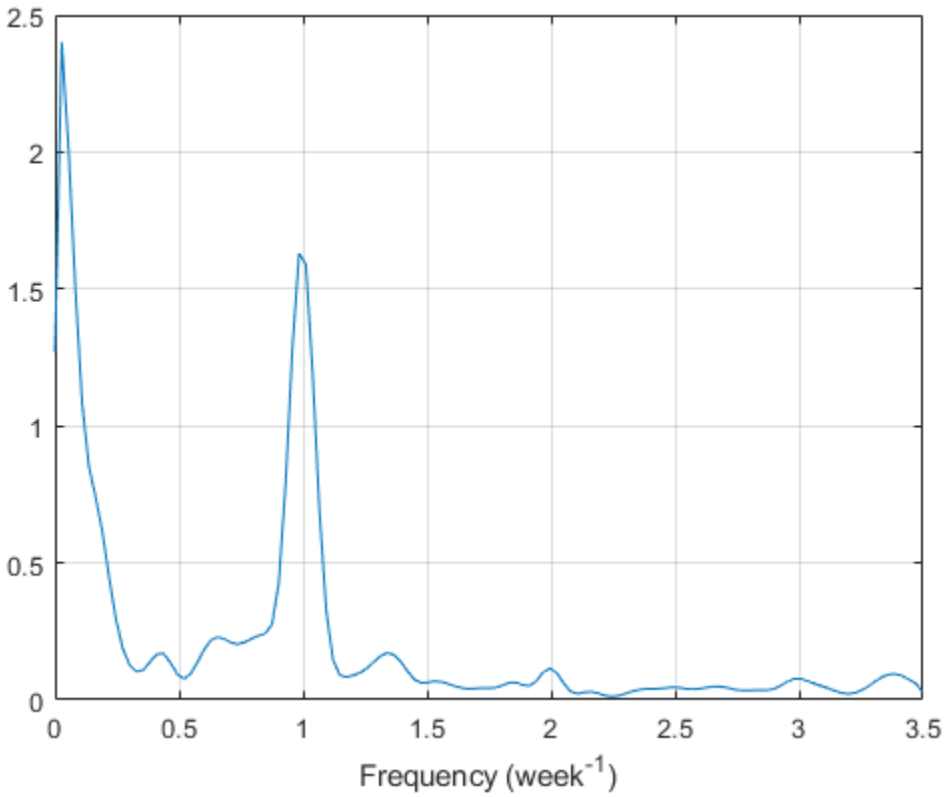
Assign values to the missing points using `resample`. By default, `resample` makes estimates using linear interpolation. Plot the original and interpolated readings. Zoom in on days 200 through 250, which contain about half of the missing points.

```
wgt_orig = wgt;
wgt = resample(wgt,daynum);

plot(daynum,wgt_orig,'.-',daynum,wgt,'o')
xlabel('Day')
ylabel('Weight (kg)')
axis([200 250 73 77])
legend('Original','Interpolated')
grid
```



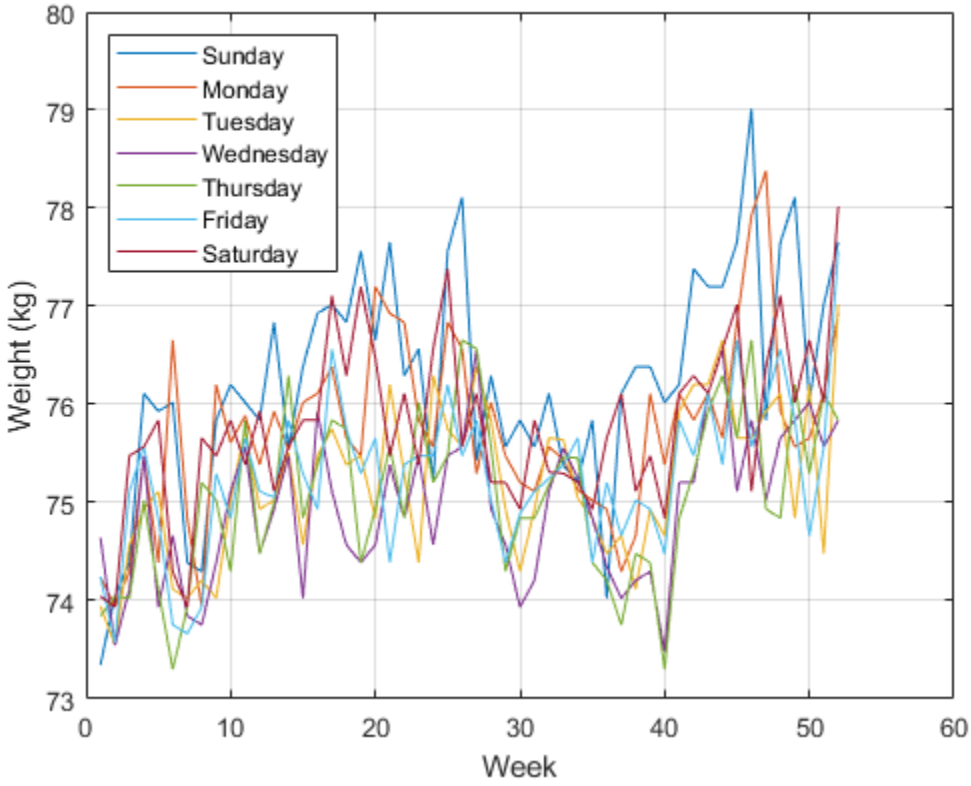

Determine if the signal is periodic by analyzing it in the frequency domain. Find the cycle duration, measuring time in weeks. Subtract the mean to concentrate on fluctuations.

```
Fs = 7;

[p,f] = pwelch(wgt-mean(wgt),[],[],[],Fs);

plot(f,p)
xlabel('Frequency (week^{-1})')
grid
```



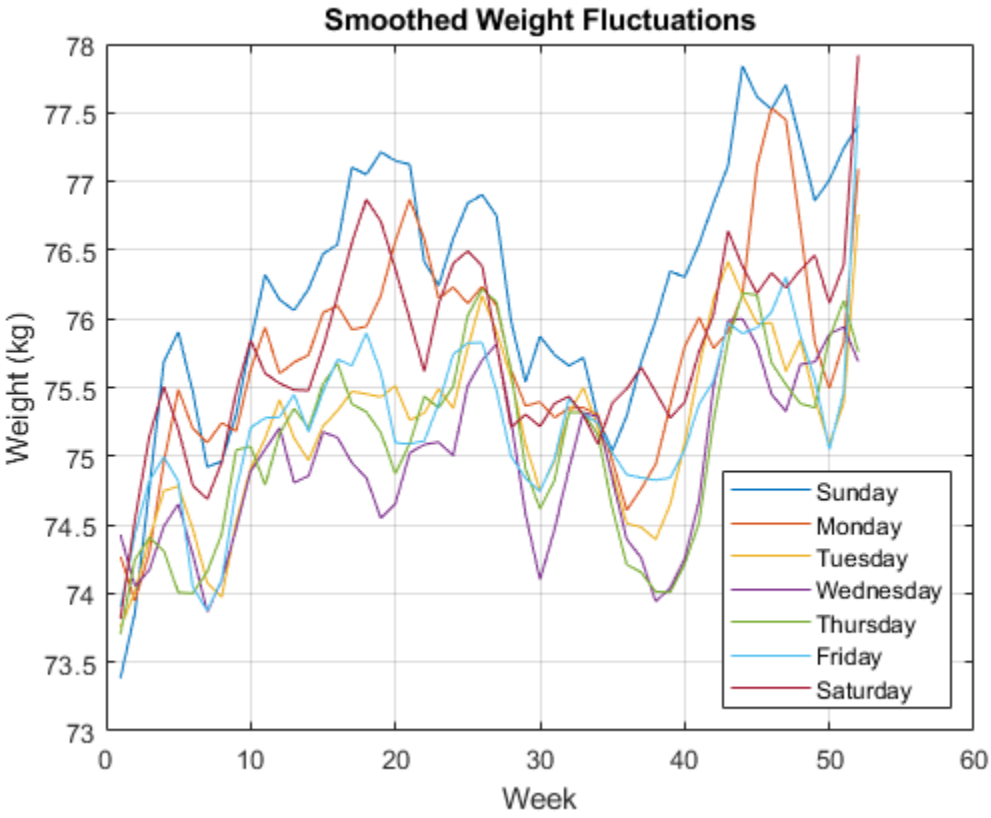

Notice how the person's weight oscillates weekly. Is there a noticeable pattern from week to week? Eliminate the last two days of the year to get 52 weeks. Reorder the measurements according to the day of the week.

```
wgd = reshape(wgt(1:7*52),[7 52]);

plot(wgd')
xlabel('Week')
ylabel('Weight (kg)')

q = legend(datestr(datenum(2012,1,1:7),'dddd'));
q.Location = 'NorthWest';
grid
```




Smooth out the fluctuations using a filter that fits low-order polynomials to subsets of the data. Specifically, set it to fit cubic polynomials to sets of seven days.

```
wgs = sgolayfilt(wgd',3,7);

plot(wgs)
xlabel('Week')
ylabel('Weight (kg)')
title('Smoothed Weight Fluctuations')

q = legend(datestr(datenum(2012,1,1:7),'dddd'));
q.Location = 'SouthEast';
grid
```




This person tends to eat more, and thus weigh more, during the weekend. Verify by computing the daily means.

```
for jk = 1:7
    fprintf('%3s mean: %5.1f kg\n', ...
        datestr(datenum(2012,1,jk),'ddd')',mean(wgd(jk,:)))
end
```

```
Sun mean:  76.2 kg
Mon mean:  75.7 kg
Tue mean:  75.2 kg
Wed mean:  74.9 kg
Thu mean:  75.1 kg
Fri mean:  75.3 kg
Sat mean:  75.8 kg
```

## See Also

`pwelch` | `sgolayfilt`

## Related Examples

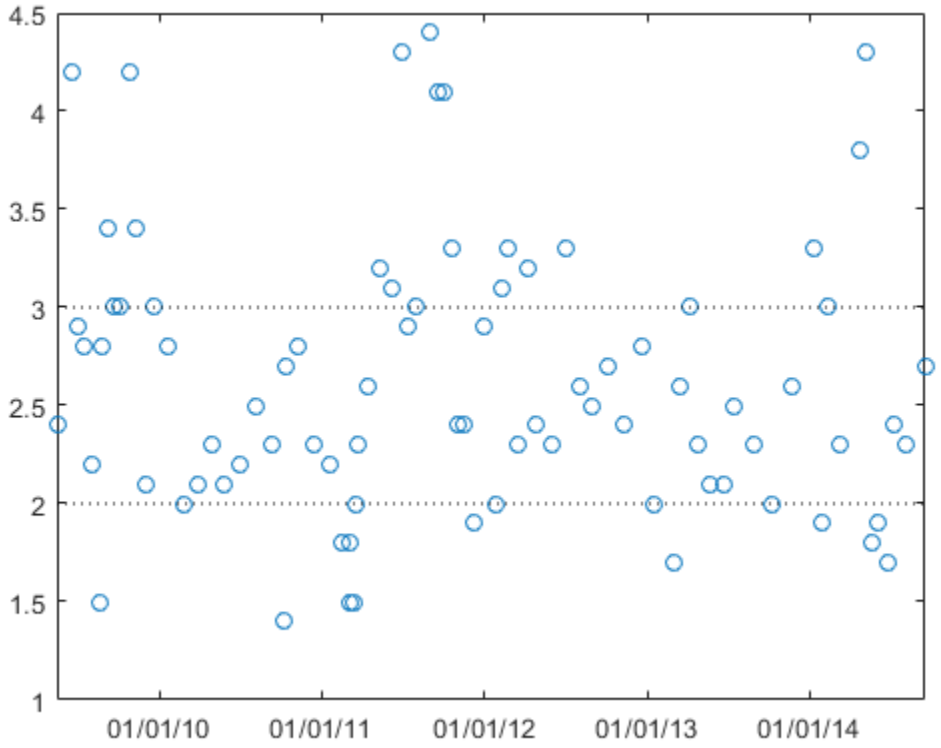
- “Signal Smoothing” on page 24-10

# Reconstruct a Signal from Irregularly Sampled Data

People predisposed to blood clotting are treated with warfarin, a blood thinner. The international normalized ratio (INR) measures the effect of the drug. Larger doses increase the INR and smaller doses decrease it. Patients are monitored regularly by a nurse, and when their INRs fall out of the target range, their doses and the frequencies of their tests change.

The file `INR.mat` contains the INR measurements performed on a patient over a five-year period. The file includes a `datetime` array with the date and time of each measurement, and a vector with the corresponding INR readings. Load the data. Plot the INR as a function of time and overlay the target INR range.

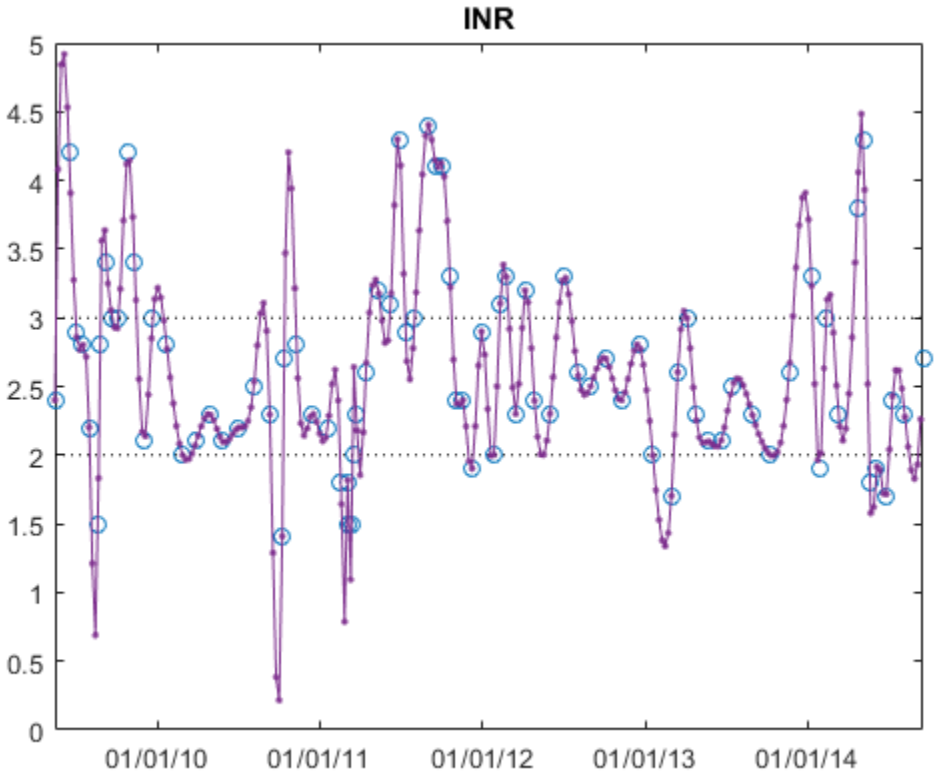
```
load('INR.mat')

plot(Date,INR,'o','DatetimeTickFormat','MM/dd/yy')

xlim([Date(1) Date(end)])
hold on
plot([xlim;xlim]',[2 3;2 3],'k:')
```




Resample the data to make the INR readings uniformly spaced. The first reading was taken at 11:28 a.m. on a Friday. Use `resample` to estimate the patient's INR at that time on every subsequent Friday. Specify a sample rate of one reading per week, or equivalently, 1/(7 × 86400) readings per second. Use spline interpolation for the resampling.

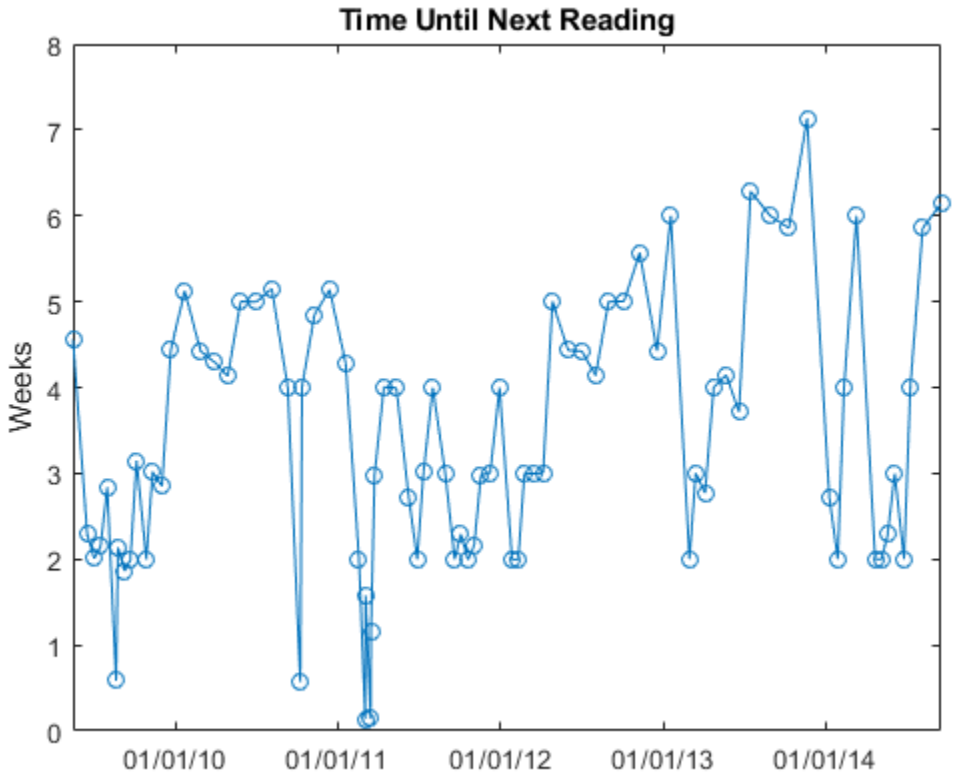
```
Date.Format = 'eeee, MM/dd/yy, HH:mm';
First = Date(1)
```

```
First = datetime
   Friday, 05/15/09, 11:28


perweek = 1/7/86400;

[rum,tee] = resample(INR,Date,perweek,1,1,'spline');

plot(tee,rum,'.-','DatetimeTickFormat','MM/dd/yy')

title('INR')
xlim([Date(1) Date(end)])
hold off
```




Each INR reading determines when the patient must be tested next. Use `diff` to construct a vector of time intervals between measurements. Express the intervals in weeks and plot them using the same *x*-axis as before. For the last point, use the next date prescribed by the anticoagulation nurse. The measurements are carried out in the United States.

```
nxt = datetime('10/30/2014 07:00 PM','Locale','en_US');

plot(Date,diff(datenum([Date;nxt]))/7,'o-', ...
    'DatetimeTickFormat','MM/dd/yy')

title('Time Until Next Reading')
xlim([Date(1) Date(end)])
ylabel('Weeks')
```




When the INR is out of range, the times between INR readings remain short. When the INR is too low, patients get their readings more often because the risk of thrombosis is elevated. When the patient's INR is in range, the times between readings increase steadily until the ratio becomes too small or too large.

The large fluctuations in the resampling could be a sign of overshooting. However, warfarin has an enormous effect on the body. Small changes in warfarin dose can change the INR drastically, as can changes in diet, time spent in airplanes, or other factors. Moreover, when the ratio goes very low (as in late 2010, where the fluctuations are largest), the warfarin is supplemented by emergency injections of enoxaparin, whose effects are even greater.

## See Also

`datenum` | `datestr` | `resample`

## External Websites

- National Institutes of Health. *Blood Thinners*. https://www.nlm.nih.gov/medlineplus/bloodthinners.html

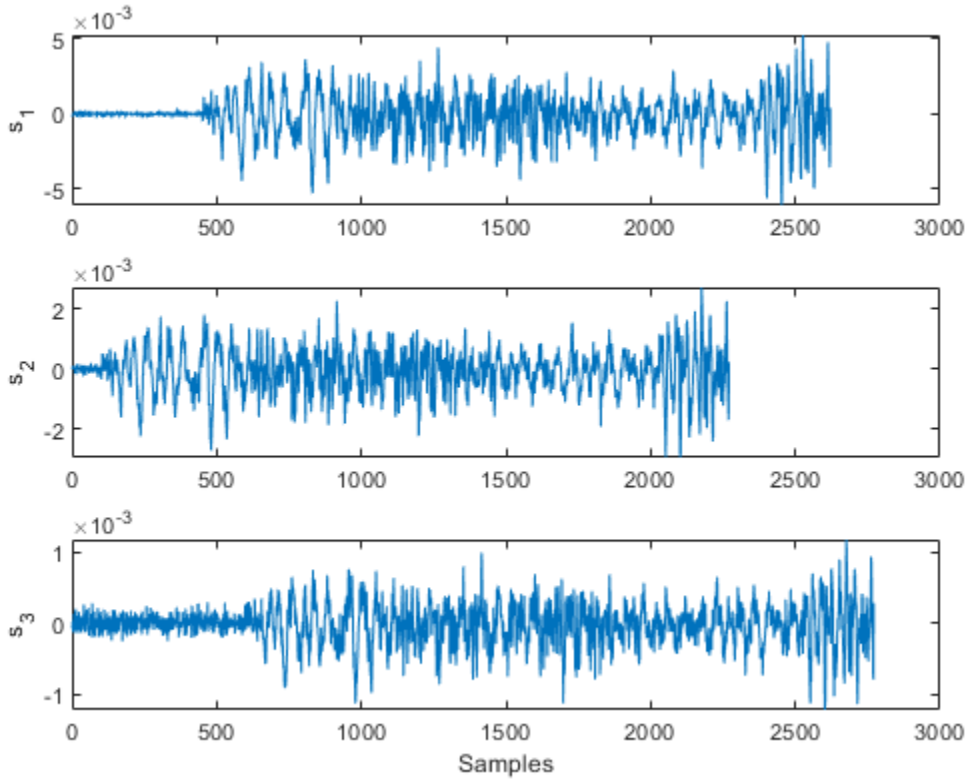
# Align Signals with Different Start Times

Many measurements involve data collected asynchronously by multiple sensors. If you want to integrate the signals, you have to synchronize them. The Signal Processing Toolbox™ has functions that let you do just that.

For example, consider a car crossing a bridge. The vibrations it produces are measured by three identical sensors located at different spots. The signals have different arrival times.

Load the signals into the MATLAB® workspace and plot them.

```
load relatedsig

ax(1) = subplot(3,1,1);
plot(s1)
ylabel('s_1')

ax(2) = subplot(3,1,2);
plot(s2)
ylabel('s_2')

ax(3) = subplot(3,1,3);
plot(s3)
ylabel('s_3')
xlabel('Samples')

linkaxes(ax,'x')
```



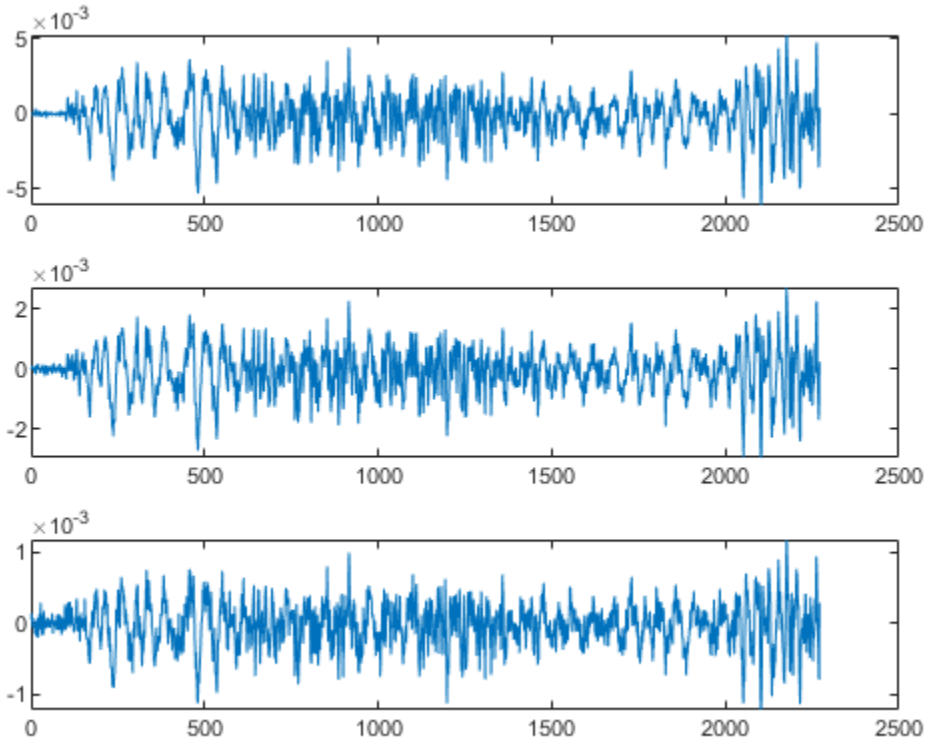
Signal `s1` lags `s2` and in turn leads `s3`. The delays can be computed exactly using `finddelay`. You see that `s2` leads `s1` by 350 samples, `s3` lags `s1` by 150 samples, and `s2` leads `s3` by 500 samples.

```
t21 = finddelay(s2,s1)
t31 = finddelay(s3,s1)
t32 = finddelay(s2,s3)
```

```
t21 =

   350


t31 =

  -150


t32 =

   500
```

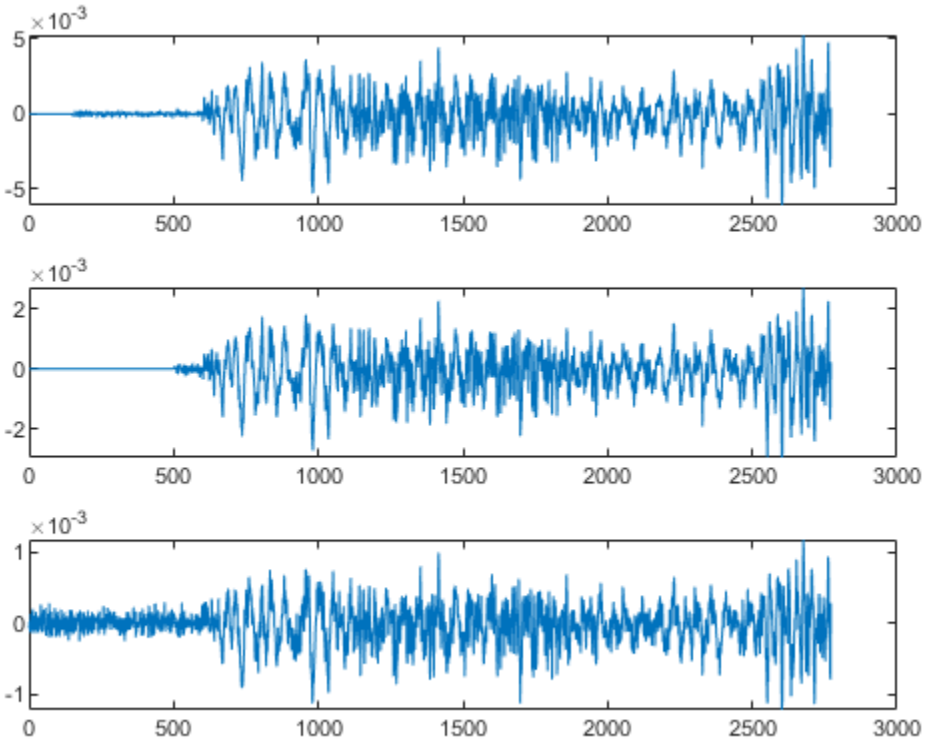
Line up the signals by leaving the earlier signal untouched and clipping the delays out of the other vectors. Add 1 to the lag differences to account for the one-based indexing used by MATLAB®. This method aligns the signals using as reference the earliest arrival time, that of `s2`.

```
axes(ax(1))
plot(s1(t21+1:end))

axes(ax(2))
plot(s2)

axes(ax(3))
plot(s3(t32+1:end))
```




Use `alignsignals` to align the signals. The function works by delaying earlier signals, so use as reference the latest arrival time, that of `s3`.

```
[x1,x3] = alignsignals(s1,s3);
x2 = alignsignals(s2,s3);

axes(ax(1))
plot(x1)

axes(ax(2))
plot(x2)

axes(ax(3))
plot(x3)
```




The signals are now synchronized and ready for further processing.

## See Also

`alignsignals` | `finddelay` | `xcorr`

## Related Examples

- “Measuring Signal Similarities” on page 24-71

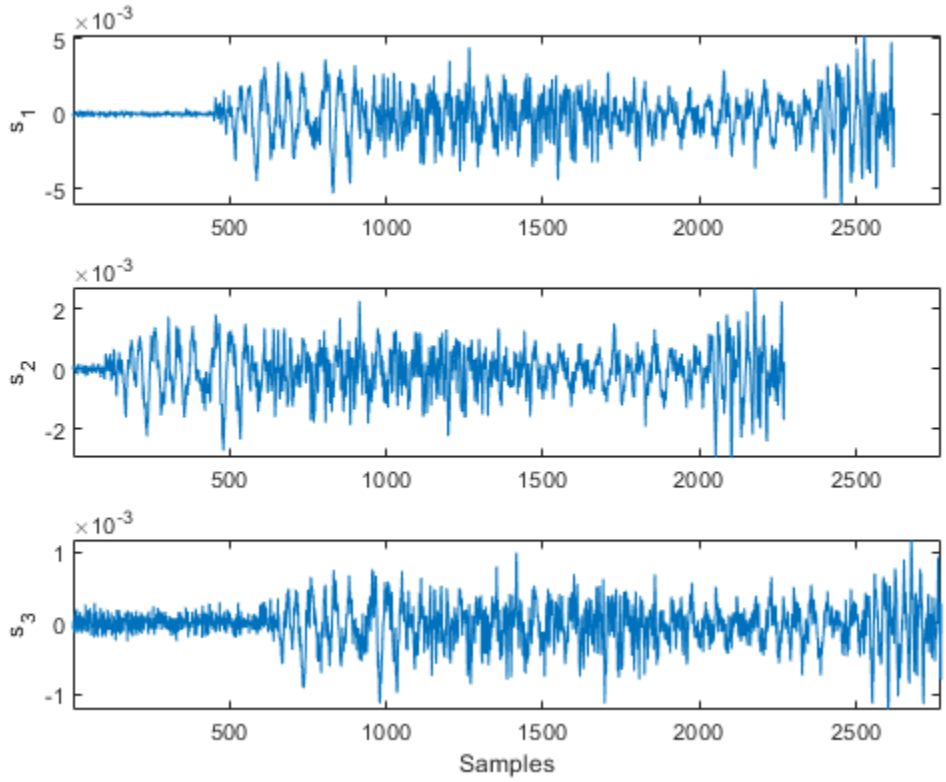
# Align Signals Using Cross-Correlation

Many measurements involve data collected asynchronously by multiple sensors. If you want to integrate the signals and study them in tandem, you have to synchronize them. Use `xcorr` for that purpose.

For example, consider a car crossing a bridge. The vibrations it produces are measured by three identical sensors located at different spots. The signals have different arrival times.

Load the signals into the MATLAB® workspace and plot them.

```
load relatedsig

ax(1) = subplot(3,1,1);
plot(s1)
ylabel('s_1')
axis tight

ax(2) = subplot(3,1,2);
plot(s2)
ylabel('s_2')
axis tight

ax(3) = subplot(3,1,3);
plot(s3)
ylabel('s_3')
axis tight
xlabel('Samples')

linkaxes(ax,'x')
```




Compute the cross-correlations between the three pairs of signals. Normalize them so their maximum value is one.

```
[C21,lag21] = xcorr(s2,s1);
C21 = C21/max(C21);

[C31,lag31] = xcorr(s3,s1);
C31 = C31/max(C31);

[C32,lag32] = xcorr(s3,s2);
C32 = C32/max(C32);
```

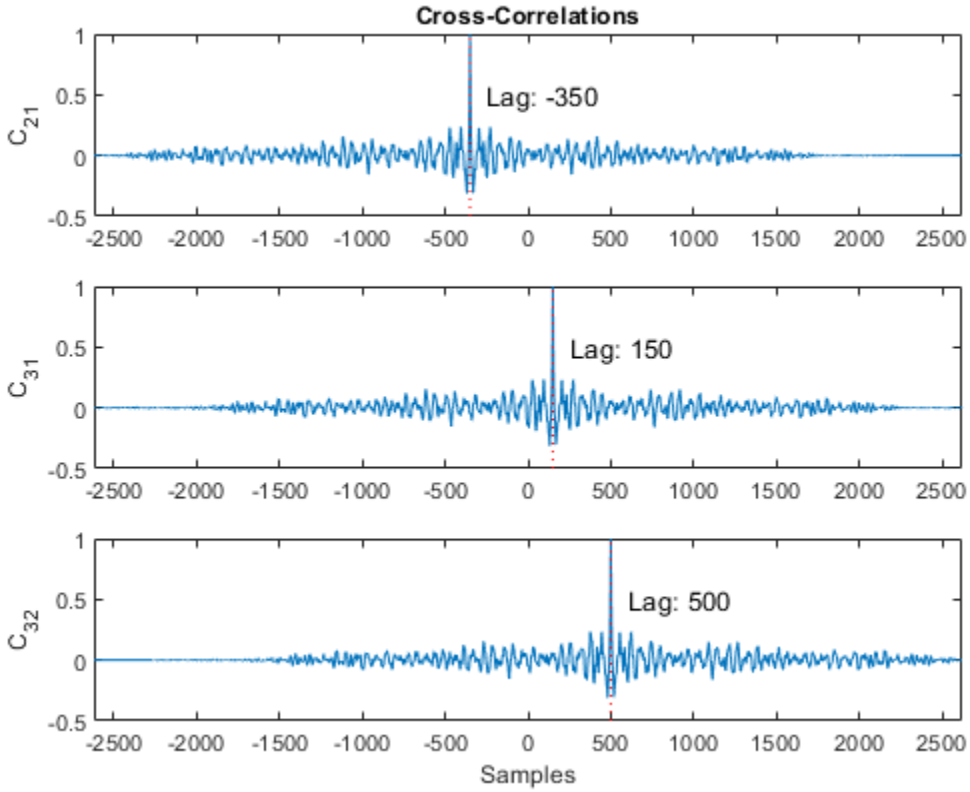
The locations of the maximum values of the cross-correlations indicate time leads or lags.

```
[M21,I21] = max(C21);
t21 = lag21(I21);

[M31,I31] = max(C31);
t31 = lag31(I31);

[M32,I32] = max(C32);
t32 = lag31(I32);
```

Plot the cross-correlations. In each plot display the location of the maximum.

```
subplot(3,1,1)
plot(lag21,C21,[t21 t21],[-0.5 1],'r:')
text(t21+100,0.5,['Lag: ' int2str(t21)])
```

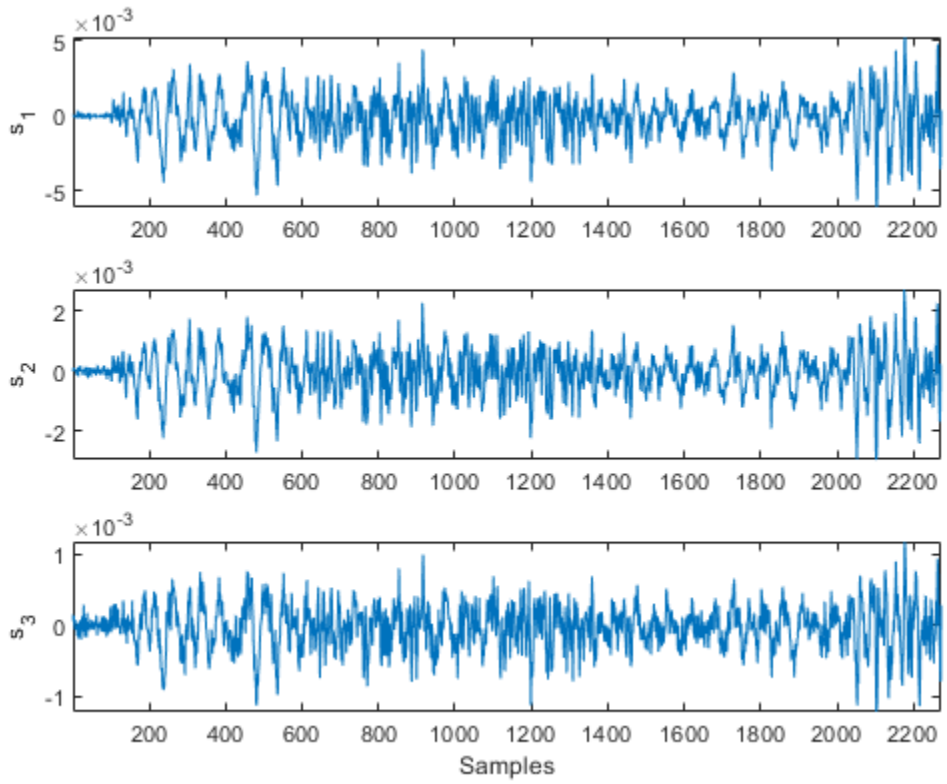
```
ylabel('C_{21}')
axis tight
title('Cross-Correlations')

subplot(3,1,2)
plot(lag31,C31,[t31 t31],[-0.5 1],'r:')
text(t31+100,0.5,['Lag: ' int2str(t31)])
ylabel('C_{31}')
axis tight

subplot(3,1,3)
plot(lag32,C32,[t32 t32],[-0.5 1],'r:')
text(t32+100,0.5,['Lag: ' int2str(t32)])
ylabel('C_{32}')
axis tight
xlabel('Samples')
```




`s2` leads `s1` by 350 samples; `s3` lags `s1` by 150 samples. Thus `s2` leads `s3` by 500 samples. Line up the signals by clipping the vectors with longer delays.

```
s1 = s1(-t21:end);
s3 = s3(t32:end);

ax(1) = subplot(3,1,1);
plot(s1)
ylabel('s_1')
axis tight
```

```
ax(2) = subplot(3,1,2);
plot(s2)
ylabel('s_2')
axis tight

ax(3) = subplot(3,1,3);
plot(s3)
ylabel('s_3')
axis tight
xlabel('Samples')

linkaxes(ax,'x')
```




The signals are now synchronized and ready for further processing.

## See Also

`alignsignals` | `finddelay` | `xcorr`

## Related Examples

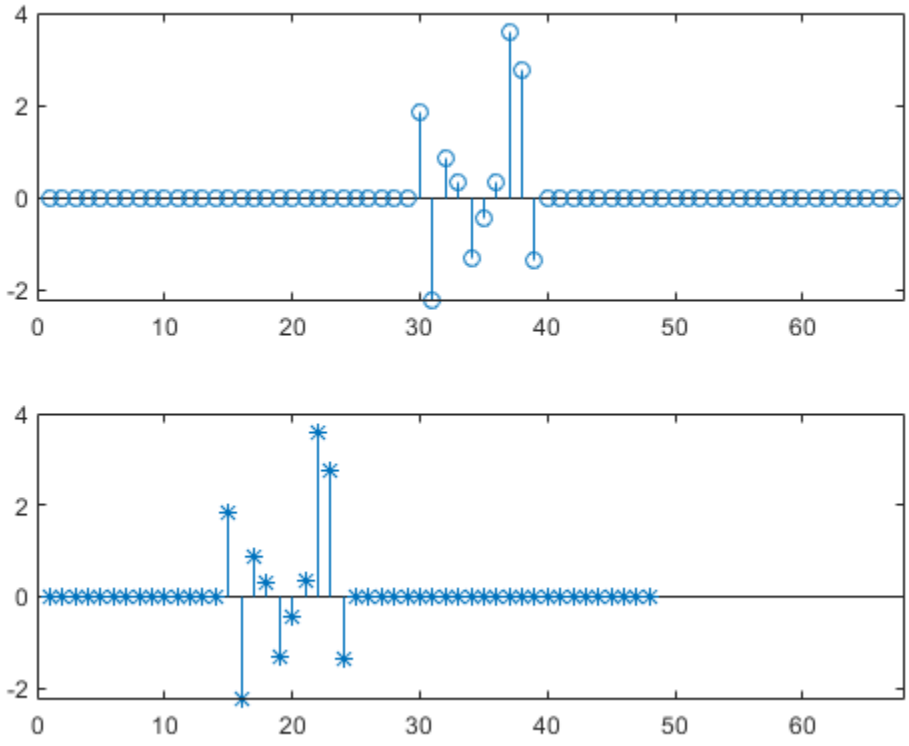
- “Measuring Signal Similarities” on page 24-71

# Align Two Simple Signals

This example shows how to use cross-correlation to align signals. In the most general case, the signals have different lengths, and to synchronize them properly, you must take into account the lengths and the order in which you input the arguments to `xcorr`.

Consider two signals, identical except for the number of surrounding zeros and for the fact that one of them lags the other.

```
sz = 30;
sg = randn(1,randi(8)+3);
s1 = [zeros(1,randi(sz)-1) sg zeros(1,randi(sz)-1)];
s2 = [zeros(1,randi(sz)-1) sg zeros(1,randi(sz)-1)];

mx = max(numel(s1),numel(s2));

subplot(2,1,1)
stem(s1)
xlim([0 mx+1])

subplot(2,1,2)
stem(s2,'*')
xlim([0 mx+1])
```




Determine which of the two signals is longer than the other in the sense of having more elements, be they zeros or not.

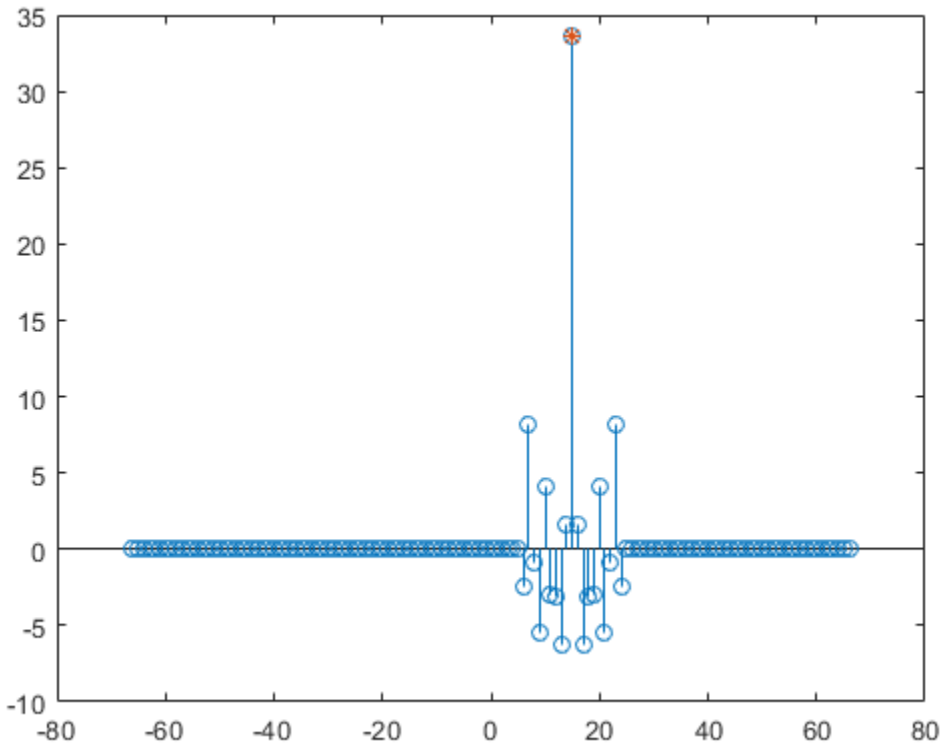
```
if numel(s1) > numel(s2)
    slong = s1;
    sshort = s2;
else
    slong = s2;
    sshort = s1;
end
```

Compute the cross-correlation of the two signals. Run `xcorr` with the longer signal as first argument and the shorter signal as second argument. Plot the result.

```
[acor,lag] = xcorr(slong,sshort);

[acormax,I] = max(abs(acor));
lagDiff = lag(I)
```

```
lagDiff = 15
```

```
figure
stem(lag,acor)
hold on
plot(lagDiff,acormax,'*')
hold off
```




Align the signals. Think of the lagging signal as being "longer" than the other, in the sense that you have to "wait longer" to detect it.

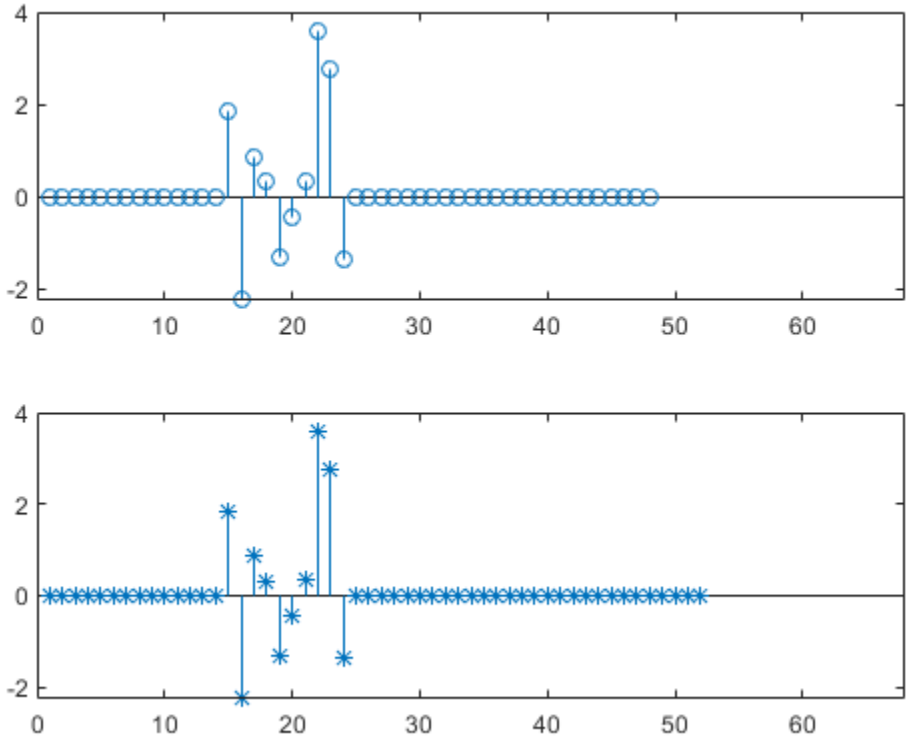
- If `lagDiff` is positive, "shorten" the long signal by considering its elements from `lagDiff`+1 to the end.
- If `lagDiff` is negative, "lengthen" the short signal by considering its elements from `-lagDiff`+1 to the end.

You must add 1 to the lag difference because MATLAB® uses one-based indexing.

```
if lagDiff > 0
    sorig = sshort;
    salign = slong(lagDiff+1:end);
else
    sorig = slong;
    salign = sshort(-lagDiff+1:end);
end
```

Plot the aligned signals.

```
subplot(2,1,1)
stem(sorig)
xlim([0 mx+1])

subplot(2,1,2)
stem(salign,'*')
xlim([0 mx+1])
```

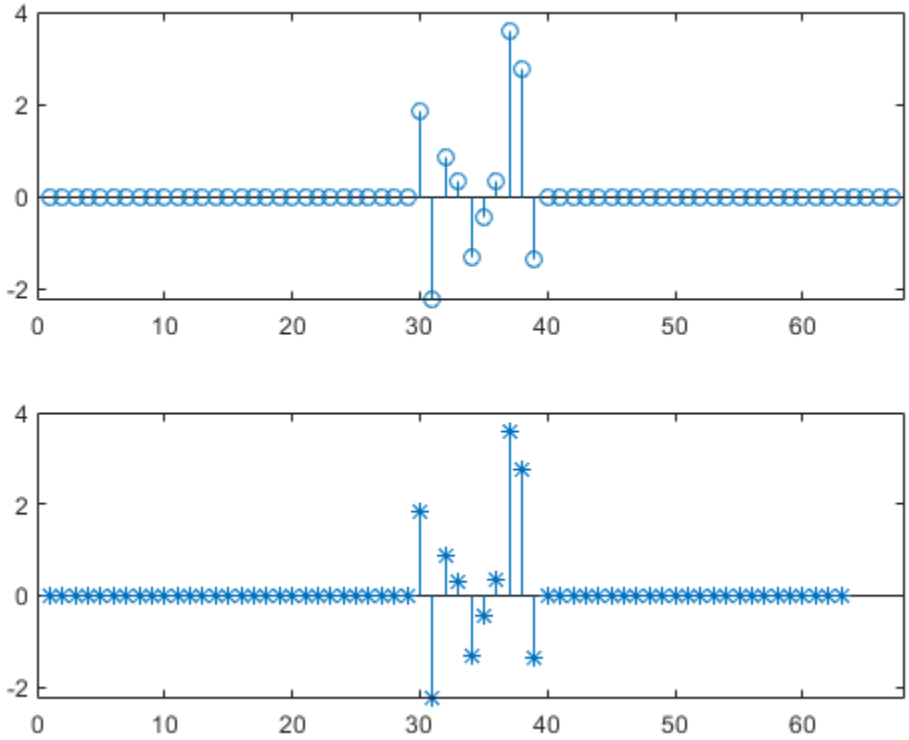



The method works because the cross-correlation operation is antisymmetric and because `xcorr` deals with signals of different lengths by adding zeros at the *end* of the shorter signal. This interpretation

lets you align the signals easily using the MATLAB® `end` operator without having to pad them by hand.

You can also align the signals at one stroke by invoking the `alignsignals` function.

```
[x1,x2] = alignsignals(s1,s2);

subplot(2,1,1)
stem(x1)
xlim([0 mx+1])

subplot(2,1,2)
stem(x2,'*')
xlim([0 mx+1])
```




## See Also

`alignsignals` | `xcorr`

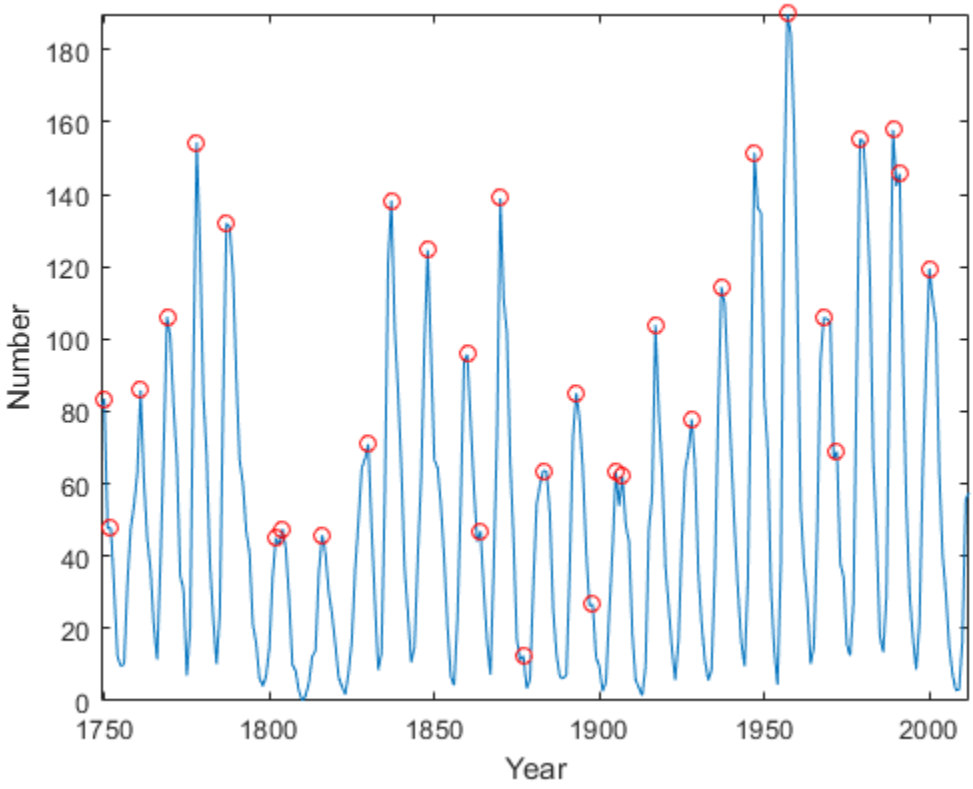
# Find Peaks in Data

Use `findpeaks` to find values and locations of local maxima in a set of data.

The file `spots_num.mat` contains the average number of sunspots observed every year from 1749 to 2012. The data are available from NASA.

Find the maxima and their years of occurrence. Plot them along with the data.

```
load('spots_num.mat')

[pks,locs] = findpeaks(avSpots);

plot(year,avSpots,year(locs),pks,'or')
xlabel('Year')
ylabel('Number')
axis tight
```

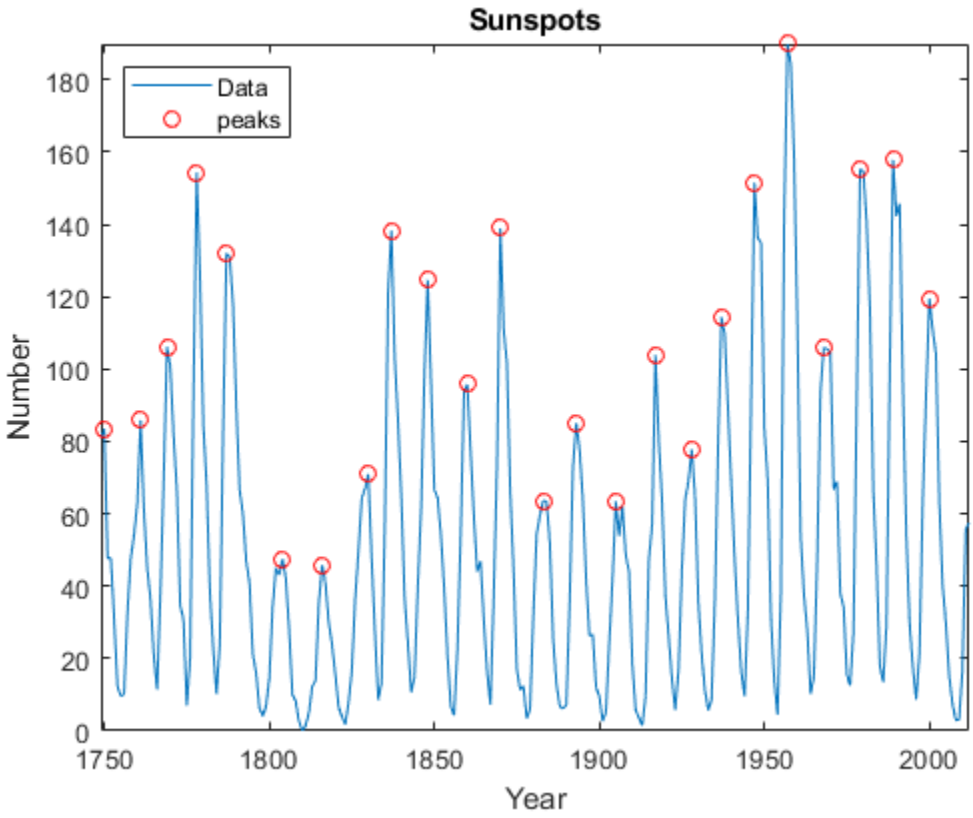



Some peaks are very close to each other. The ones that are not recur at regular intervals. There are roughly five such peaks per 50-year period.

To make a better estimate of the cycle duration, use `findpeaks` again, but this time restrict the peak-to-peak separation to at least six years. Compute the mean interval between maxima.

```
[pks,locs] = findpeaks(avSpots,'MinPeakDistance',6);

plot(year,avSpots,year(locs),pks,'or')
```

```
xlabel('Year')
ylabel('Number')
title('Sunspots')
axis tight

legend('Data','peaks','Location','NorthWest')
```



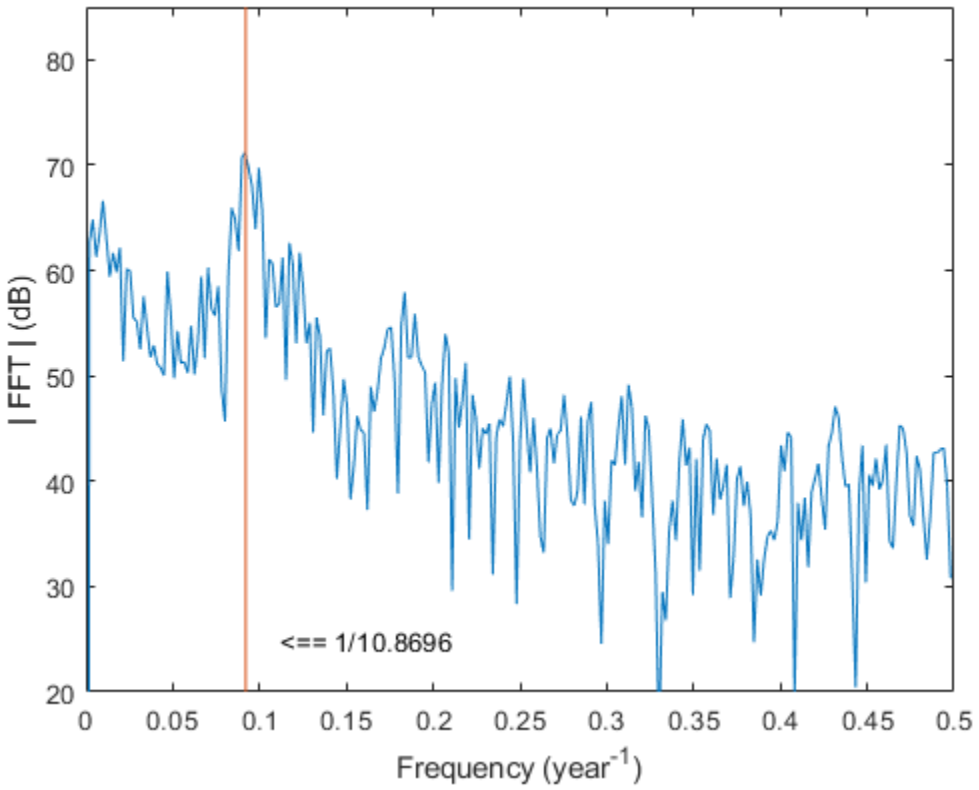

```
cycles = diff(locs);
meanCycle = mean(cycles)
```

```
meanCycle = 10.8696
```

It is well known that solar activity cycles roughly every 11 years. Check by using the Fourier transform. Remove the mean of the signal to concentrate on its fluctuations. Recall that the sample rate is measured in years. Use frequencies up to the Nyquist frequency.

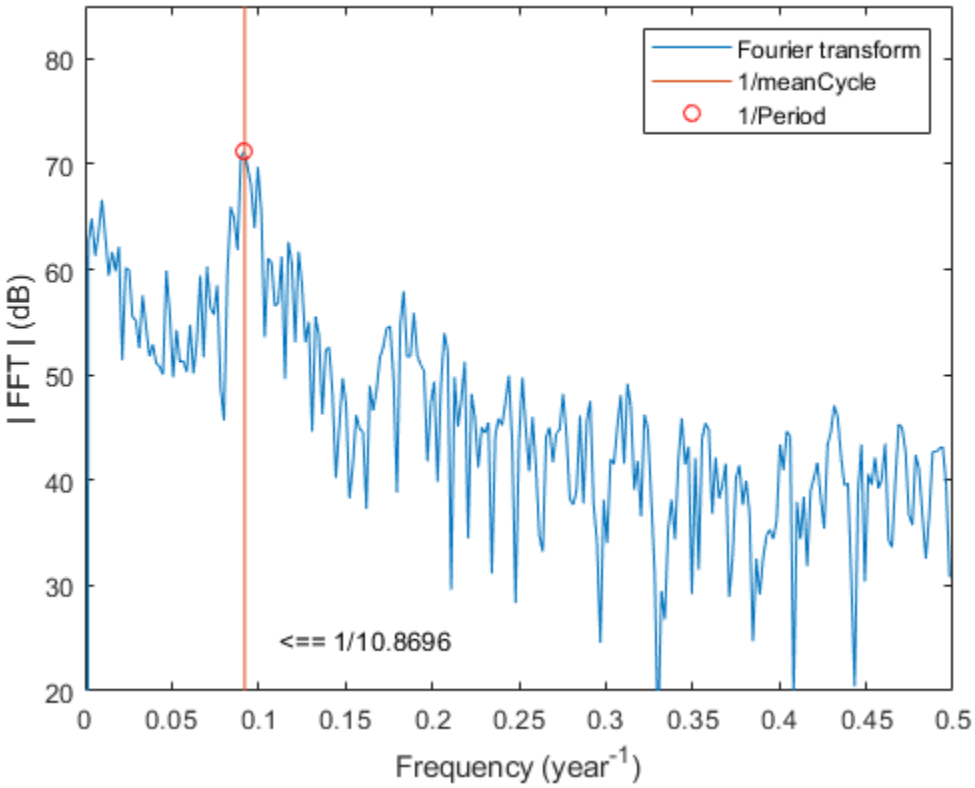
```
Fs = 1;
Nf = 512;

df = Fs/Nf;
f = 0:df:Fs/2-df;

trSpots = fftshift(fft(avSpots-mean(avSpots),Nf));

dBspots = 20*log10(abs(trSpots(Nf/2+1:Nf)));

yaxis = [20 85];
```

```
plot(f,dBspots,1./[meanCycle meanCycle],yaxis)
xlabel('Frequency (year^{-1})')
ylabel('| FFT | (dB)')
axis([0 1/2 yaxis])
text(1/meanCycle + .02,25,['<== 1/' num2str(meanCycle)])
```




The Fourier transform indeed peaks at the expected frequency, confirming the 11-year conjecture. You also can find the period by locating the highest peak of the Fourier transform.

```
[pk,MaxFreq] = findpeaks(dBspots,'NPeaks',1,'SortStr','descend');

Period = 1/f(MaxFreq)

Period = 10.8936


hold on
plot(f(MaxFreq),pk,'or')
hold off
legend('Fourier transform','1/meanCycle','1/Period')
```




The two estimates coincide quite well.

## See Also

`dlmread` | `findpeaks`

## Related Examples

- “Peak Analysis” on page 24-60

# Find a Signal in a Measurement

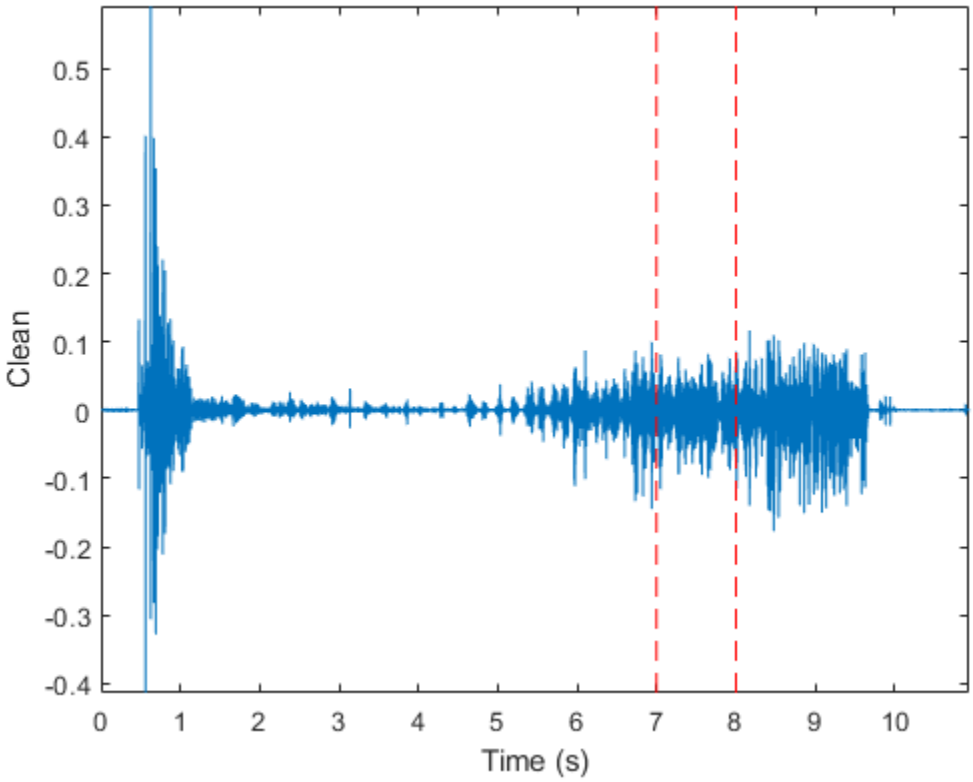
You receive some data and would like to know if it matches a longer stream you have measured. Cross-correlation allows you to make that determination, even when the data are corrupted by noise.

Load into the workspace a recording of a ring spinning on a tabletop. Crop a one-second fragment and listen to it.

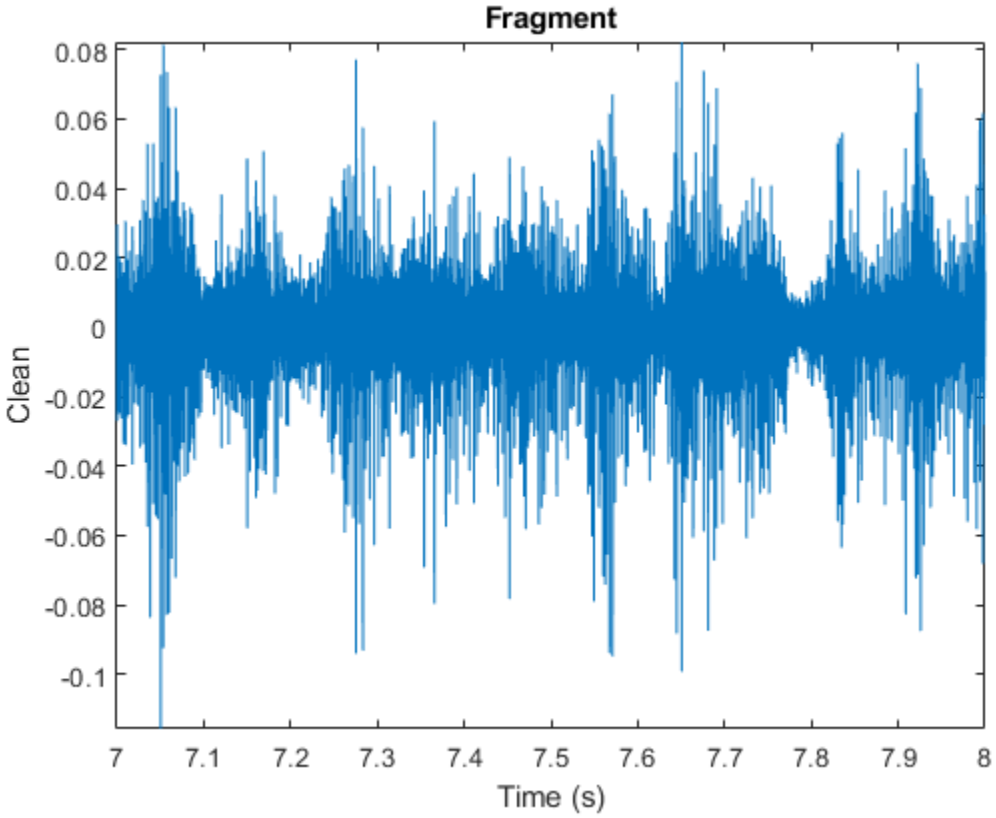
```
load('Ring.mat')

Time = 0:1/Fs:(length(y)-1)/Fs;

m = min(y);
M = max(y);

Full_sig = double(y);

timeA = 7;
timeB = 8;
snip = timeA*Fs:timeB*Fs;

Fragment = Full_sig(snip);

% To hear, type soundsc(Fragment,Fs)
```

Plot the signal and the fragment. Highlight the fragment endpoints for reference.

```
plot(Time,Full_sig,[timeA timeB;timeA timeB],[m m;M M],'r--')
xlabel('Time (s)')
ylabel('Clean')
axis tight
```

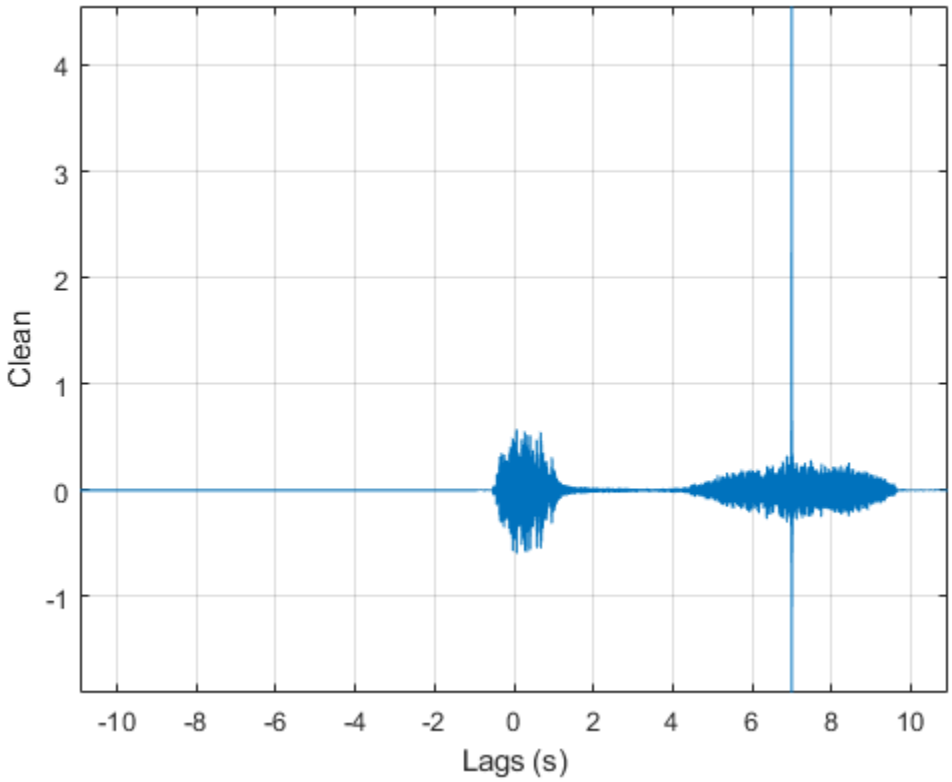



```
plot(snip/Fs,Fragment)
xlabel('Time (s)')
ylabel('Clean')
title('Fragment')
axis tight
```



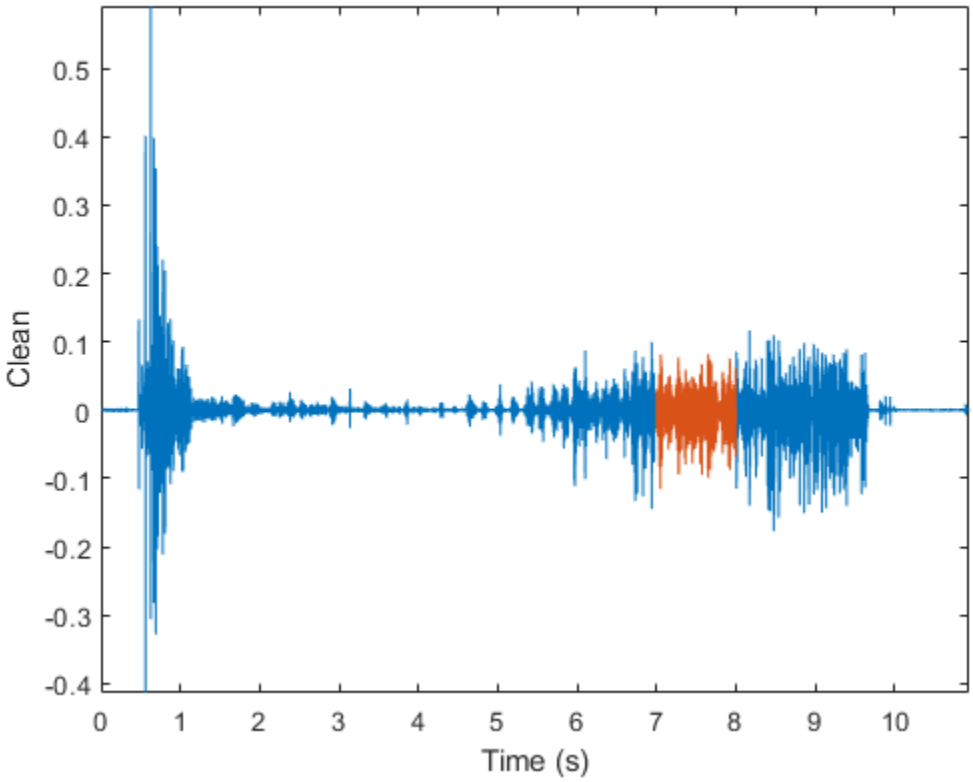

Compute and plot the cross-correlation of the full signal and the fragment.

```
[xCorr,lags] = xcorr(Full_sig,Fragment);

plot(lags/Fs,xCorr)
grid
xlabel('Lags (s)')
ylabel('Clean')
axis tight
```



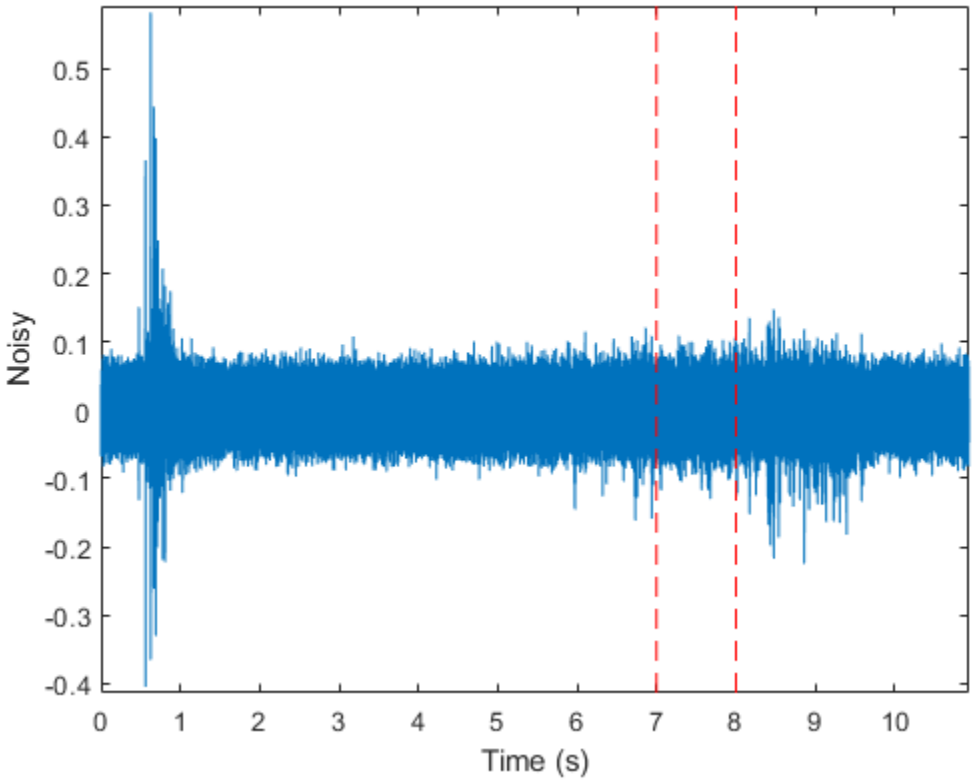

The lag at which the cross-correlation is greatest is the time delay between the signals' starting points. Replot the signal and overlay the fragment.

```
[~,I] = max(abs(xCorr));
maxt = lags(I);

Trial = NaN(size(Full_sig));
Trial(maxt+1:maxt+length(Fragment)) = Fragment;

plot(Time,Full_sig,Time,Trial)
xlabel('Time (s)')
ylabel('Clean')
axis tight
```



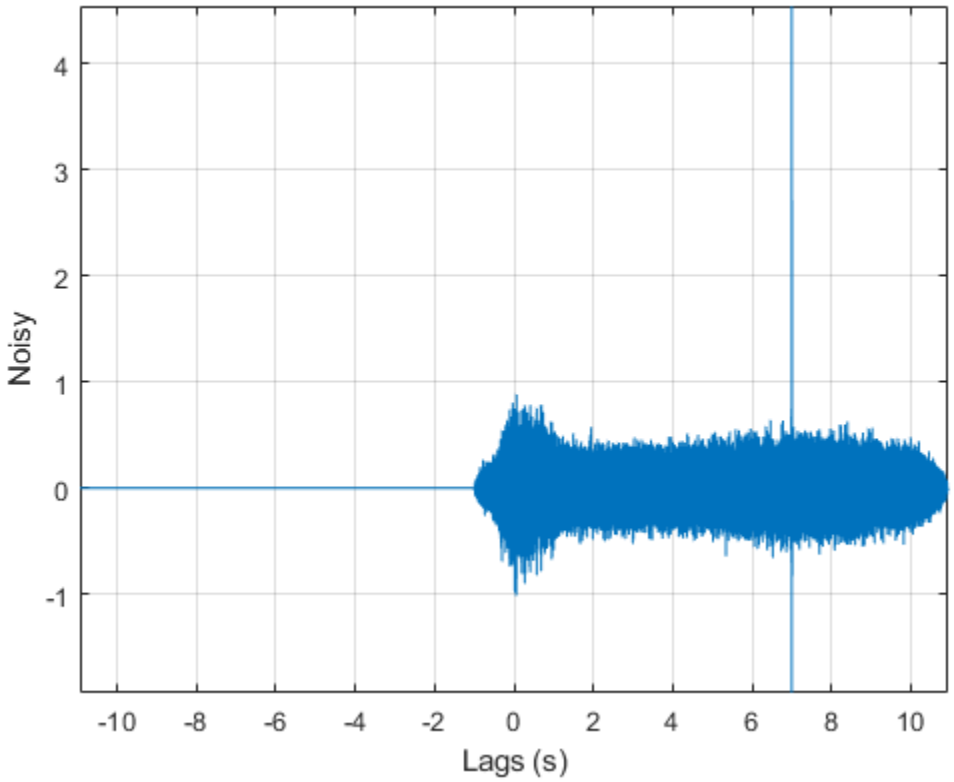

Repeat the procedure, but add noise separately to signal and fragment. The sound cannot be picked out from the noise.

```
NoiseAmp = 0.2*max(abs(Fragment));

Fragment = Fragment+NoiseAmp*randn(size(Fragment));

Full_sig = Full_sig+NoiseAmp*randn(size(Full_sig));

% To hear, type soundsc(Fragment,Fs)

plot(Time,Full_sig,[timeA timeB;timeA timeB],[m m;M M],'r--')
xlabel('Time (s)')
ylabel('Noisy')
axis tight
```

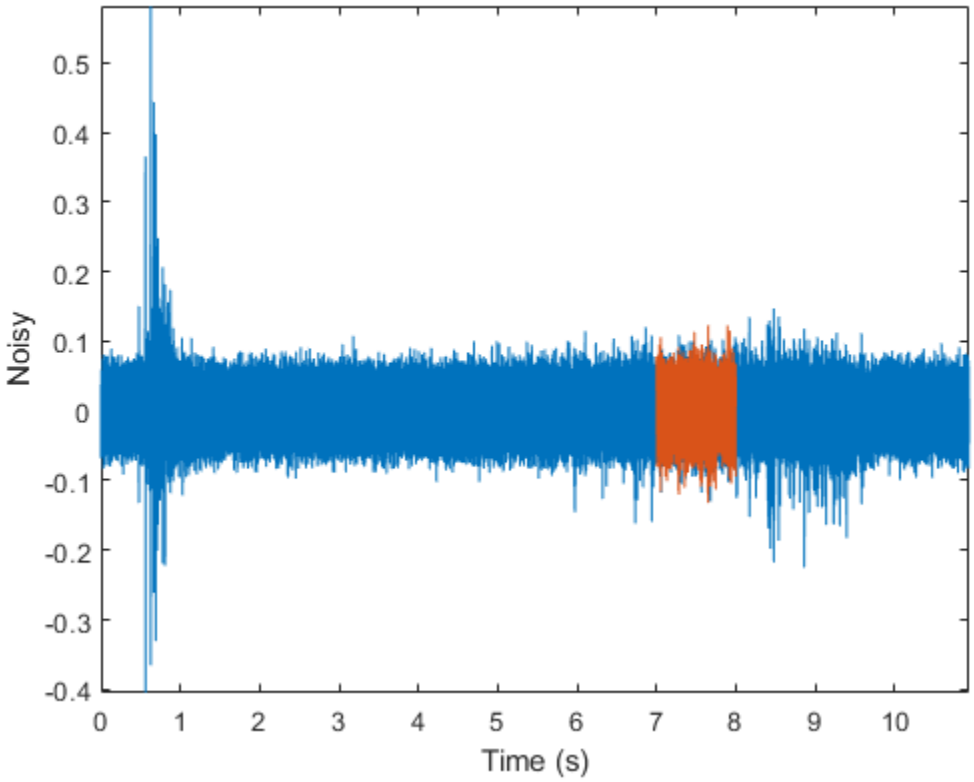



The procedure finds the missing fragment despite the high noise level.

```
[xCorr,lags] = xcorr(Full_sig,Fragment);

plot(lags/Fs,xCorr)
grid
xlabel('Lags (s)')
ylabel('Noisy')
axis tight
```




```
[~,I] = max(abs(xCorr));
maxt = lags(I);

Trial = NaN(size(Full_sig));
Trial(maxt+1:maxt+length(Fragment)) = Fragment;

figure
plot(Time,Full_sig,Time,Trial)
xlabel('Time (s)')
ylabel('Noisy')
axis tight
```




## See Also

`xcorr`

## Related Examples

- “Measuring Signal Similarities” on page 24-71

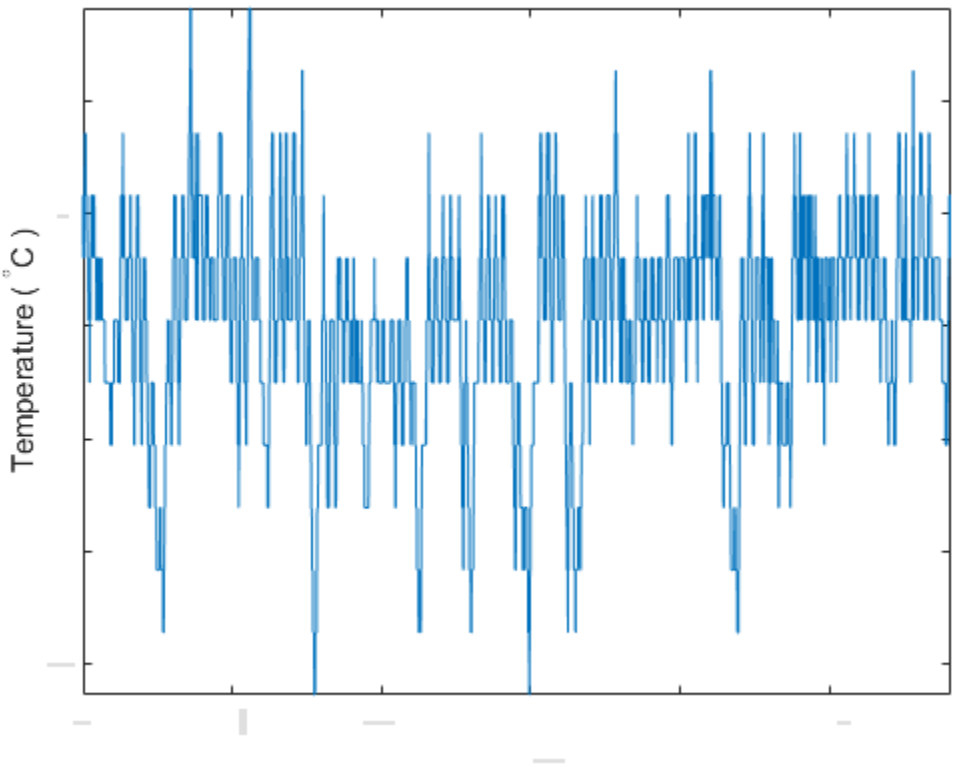
# Find Periodicity Using Autocorrelation

Measurement uncertainty and noise sometimes make it difficult to spot oscillatory behavior in a signal, even if such behavior is expected.

The autocorrelation sequence of a periodic signal has the same cyclic characteristics as the signal itself. Thus, autocorrelation can help verify the presence of cycles and determine their durations.

Consider a set of temperature data collected by a thermometer inside an office building. The device takes a reading every half hour for four months. Load the data and plot it. Subtract the mean to concentrate on temperature fluctuations. Convert the temperature to degrees Celsius. Measure time in days. The sample rate is thus 2 measurements/hour × 24 hours/day = 48 measurements/day.

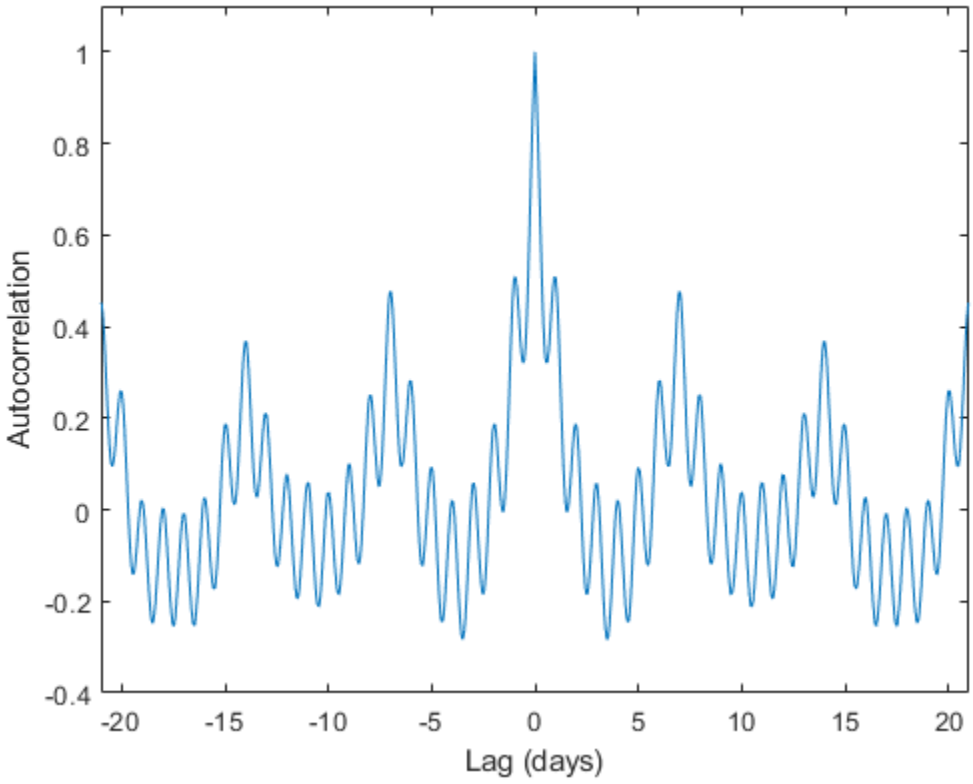
```
load officetemp

tempC = (temp-32)*5/9;

tempnorm = tempC-mean(tempC);

fs = 2*24;
t = (0:length(tempnorm) - 1)/fs;

plot(t,tempnorm)
xlabel('Time (days)')
ylabel('Temperature ( {}^\circC )')
axis tight
```



The temperature does seem to oscillate, but the lengths of the cycles cannot be read out easily.

Compute the autocorrelation of the temperature such that it is unity at zero lag. Restrict the positive and negative lags to three weeks. Note the double periodicity of the signal.

```
[autocor,lags] = xcorr(tempnorm,3*7*fs,'coeff');

plot(lags/fs,autocor)
xlabel('Lag (days)')
ylabel('Autocorrelation')
axis([-21 21 -0.4 1.1])
```




Determine the short and long periods by finding the peak locations and determining the average time differences between them.

To find the long period, restrict `findpeaks` to look for peaks separated by more than the short period and with a minimum height of 0.3.

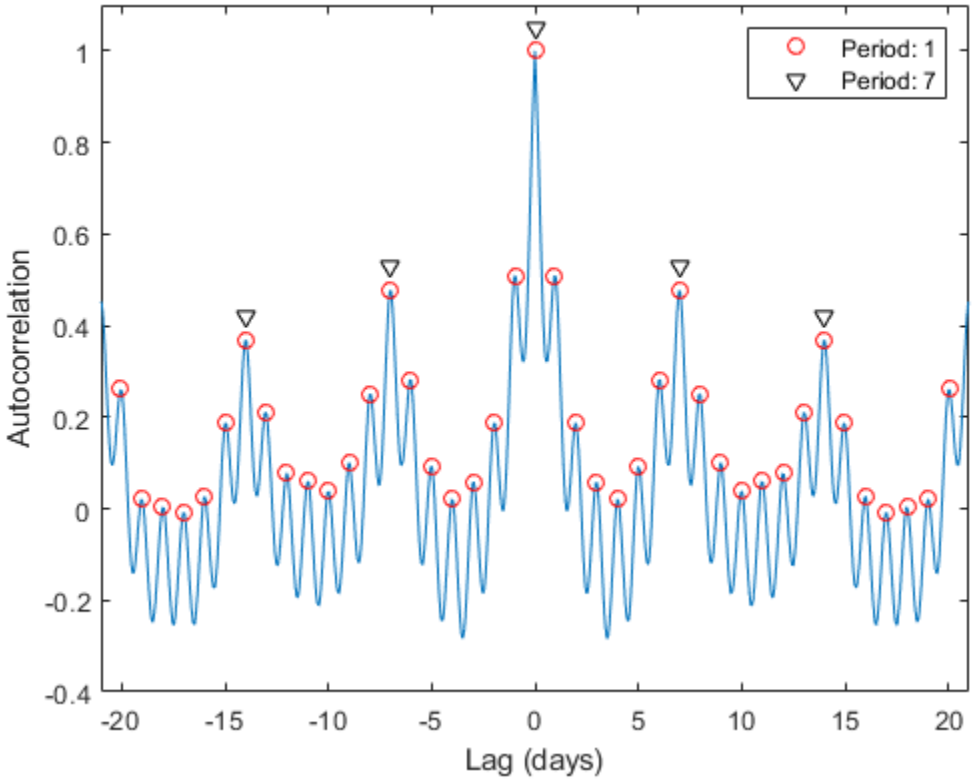
```
[pksh,lcsh] = findpeaks(autocor);
short = mean(diff(lcsh))/fs
```

```
short = 1.0021
```

```
[pklg,lclg] = findpeaks(autocor, ...
    'MinPeakDistance',ceil(short)*fs,'MinPeakheight',0.3);
long = mean(diff(lclg))/fs
```

```
long = 6.9896
```

```
hold on
pks = plot(lags(lcsh)/fs,pksh,'or', ...
    lags(lclg)/fs,pklg+0.05,'vk');
hold off
legend(pks,[repmat('Period: ',[2 1]) num2str([short;long],0)])
axis([-21 21 -0.4 1.1])
```




To a very good approximation, the autocorrelation oscillates both daily and weekly. This is to be expected, since the temperature in the building is higher when people are at work and lower at nights and on weekends.

## See Also

`findpeaks` | `xcorr`

## Related Examples

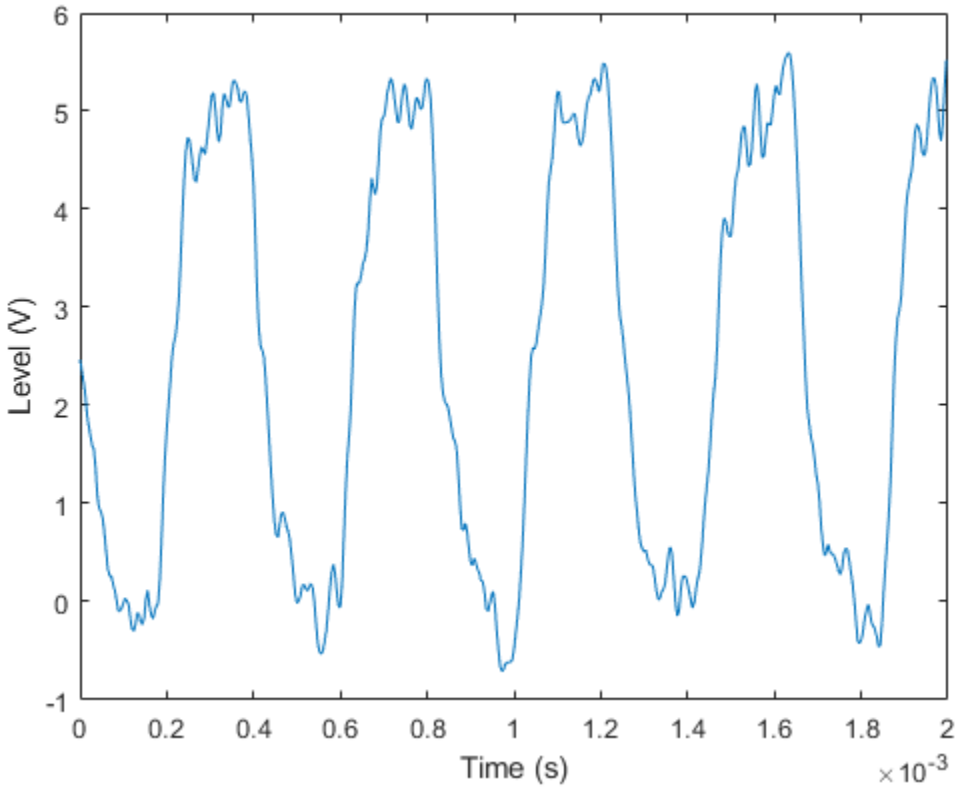
- “Find Periodicity Using Frequency Analysis” on page 23-73

# Extract Features of a Clock Signal

How sharply does an on/off signal turn on and off? How often and for how long is it activated? Determine all those characteristics for the output of a clock.

Load the signal and plot it. The time is measured in seconds and the level in volts.

```
load('clock.mat')

plot(tclock,clocksig)
xlabel('Time (s)')
ylabel('Level (V)')
```

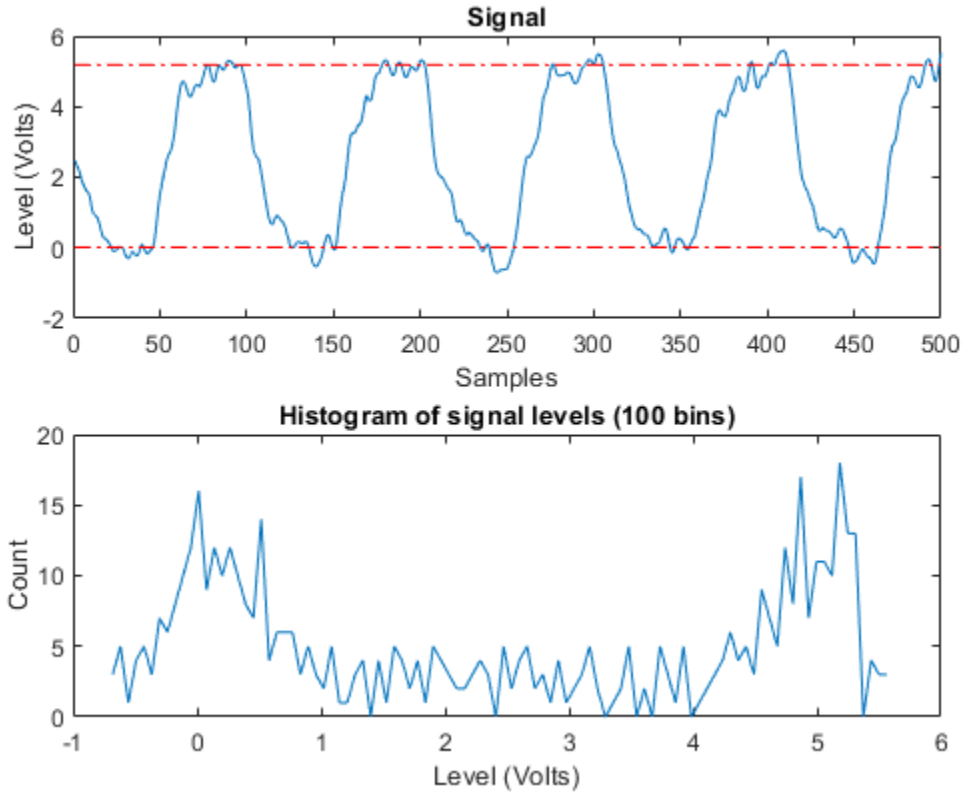



Use `statelevels` to find the lower and upper levels of the signal by means of a histogram. If you do not specify an output, the function plots the signal, marks the levels, and displays the histogram.

```
levels = statelevels(clocksig)
```

```
levels = 1×2

    0.0138    5.1848
```

```
statelevels(clocksig);
```



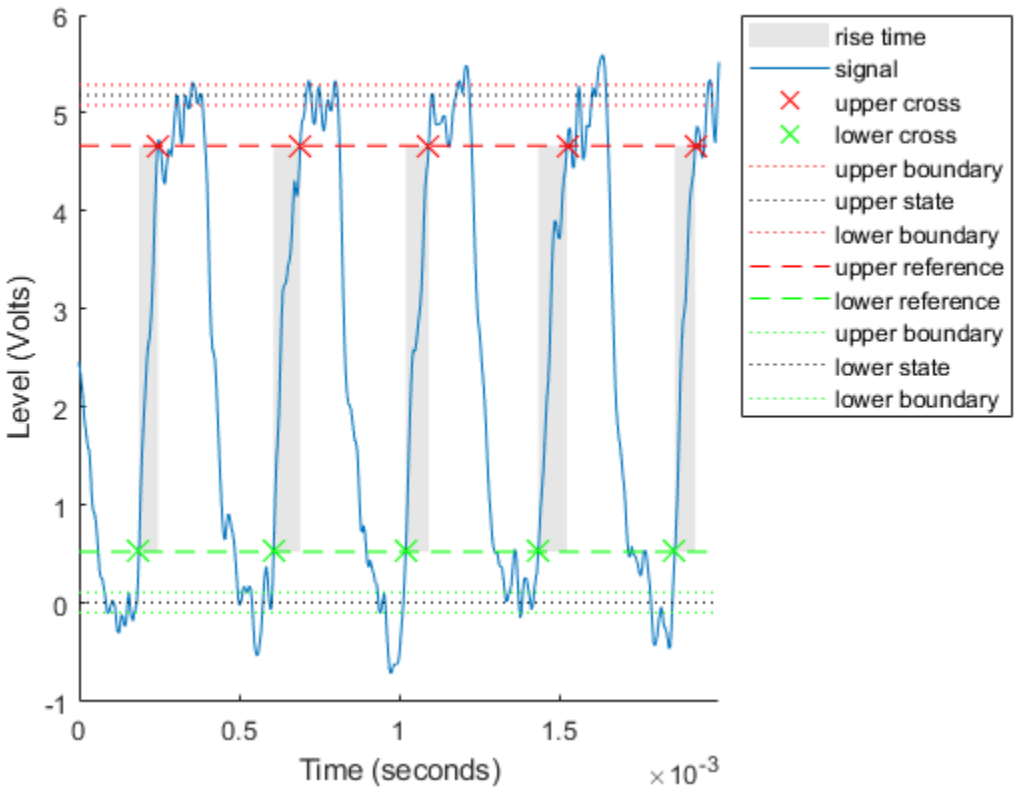

Determine how fast the signal rises at each transition. `risetime` uses the lower and upper levels found by `statelevels`. It defines the rise time as the time it takes the signal to rise from 10% to 90% of the difference between the levels.

```
[Rise,LoTime,HiTime,LoLev,HiLev] = risetime(clocksig,tclock);

Levels = [LoLev HiLev; (levels(2)-levels(1))*[0.1 0.9]+levels(1)]
```

```
Levels = 2×2

    0.5309    4.6677
    0.5309    4.6677
```

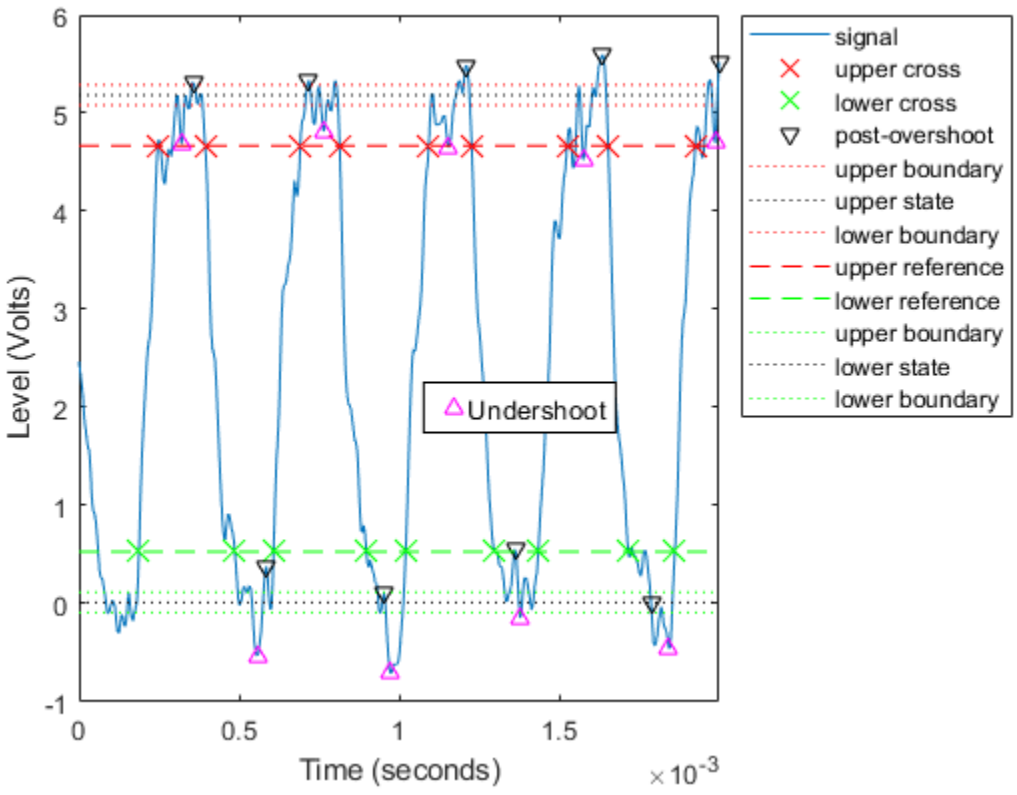
If you call `risetime` without outputs, the function draws an annotated plot of the signal. The rise times are shaded, the crossing points are marked, and the levels are displayed. You can use the time vector or the sample rate as input.

```
risetime(clocksig,Fs);
```



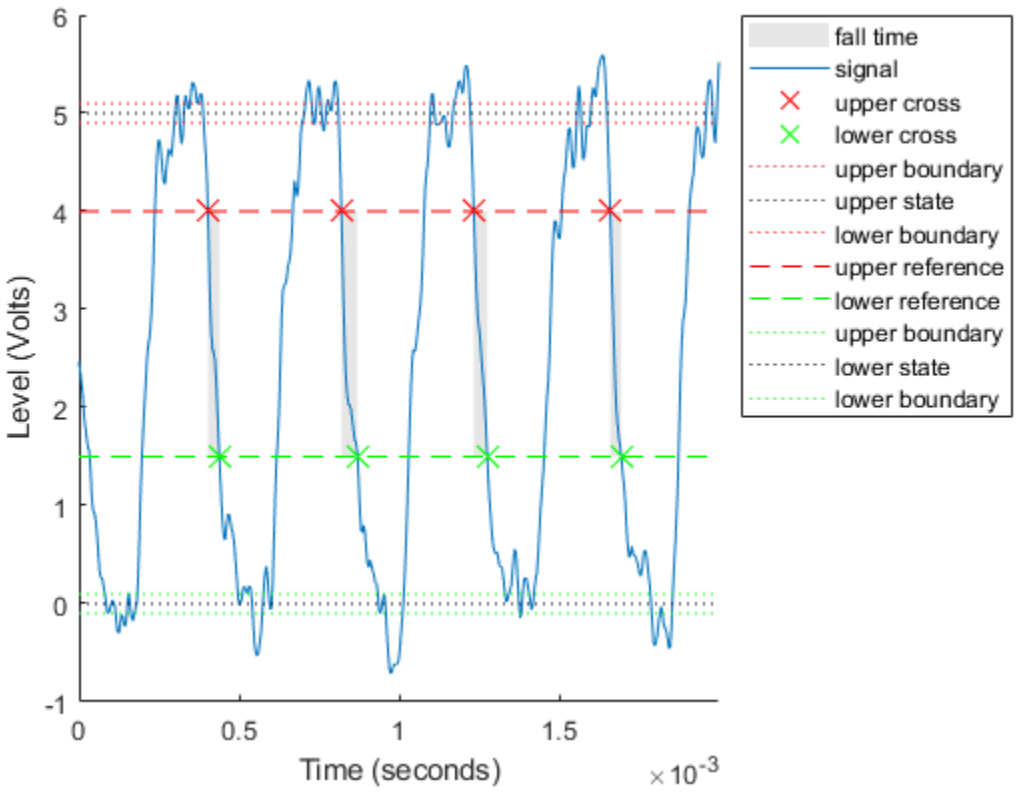

`overshoot` and `undershoot` show how far the signal deviates from the state levels at each transition. The results are expressed as percentages of the difference between the levels. Further outputs give the actual times and signal values.

```
overshoot(clocksig,Fs);

[pctgs,values,times] = undershoot(clocksig,Fs);

hold on
text(1.1e-3,2,'     Undershoot','Background','w','Edge','k')
plot([times;1.17e-3],[values;2],'^m','HandleVisibility','off')
hold off
```




Determine how fast the signal falls using `falltime`. You can set the state levels and the percentage reference levels manually. You can do the same with `risetime`.

```
falltime(clocksig,tclock, ...
    'PercentReferenceLevels',[30 80],'StateLevels',[0 5]);
```

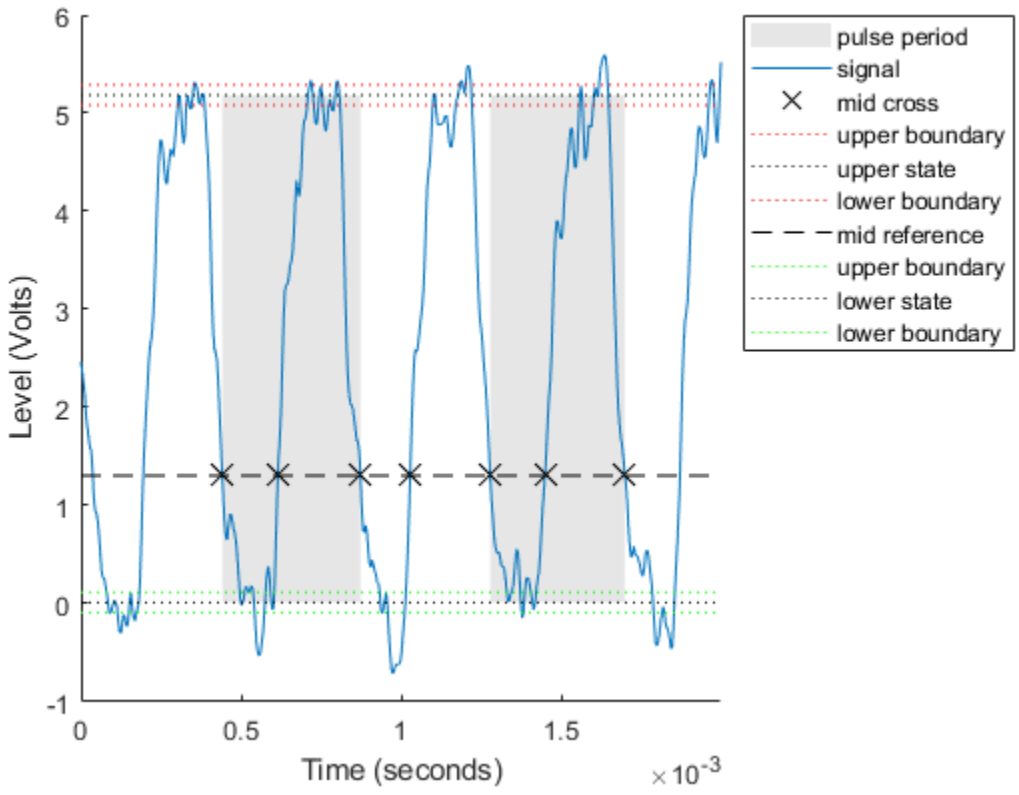



Find the period of the signal. By default, the period is defined as the time elapsed between consecutive rising crossings of the reference level halfway between the state levels. You can change the polarity of the crossings, specify the state levels, or adjust the reference level.

```
per = pulseperiod(clocksig,tclock)
```

```
per = 4×1
10^-3 ×

    0.4143
    0.4200
    0.4188
    0.4111
```

```
pulseperiod(clocksig,Fs,'Polarity','negative','MidPct',25);
```




The duty cycle is the ratio of pulse width to pulse period. Determine it directly or using a dedicated function.

```
dut = dutycycle(clocksig,Fs);

wdt = pulsewidth(clocksig,Fs);

compare = [wdt./per dut]
```

```
compare = 4×2

    0.4862    0.4862
    0.4756    0.4756
    0.4871    0.4871
    0.4886    0.4886
```

## See Also

`dutycycle` | `falltime` | `overshoot` | `pulseperiod` | `pulsewidth` | `risetime` | `slewrate` | `statelevels` | `undershoot`

## Related Examples

- “Measurement of Pulse and Transition Characteristics” on page 24-82

# Find Periodicity in a Categorical Time Series

This example shows how to perform spectral analysis of categorical-valued time-series data. The spectral analysis of categorical-valued time series is useful when you are interested in cyclic behavior of data whose values are not inherently numerical. This example reproduces in part the analysis reported in Stoffer et al. (1988). The data are taken from Stoffer, Tyler, and Wendt (2000).

The data are from a study of sleep states in newborn children. A pediatric neurologist scored an infant's electroencephalographic (EEG) recording every minute for approximately two hours. The neurologist categorized the infant's sleep state into one of the following:

- `qt` - Quiet sleep, trace alternant
- `qh` - Quiet sleep, high voltage
- `tr` - Transitional sleep
- `al` - Active sleep, low voltage
- `ah` - Active sleep, high voltage
- `aw` - Awake

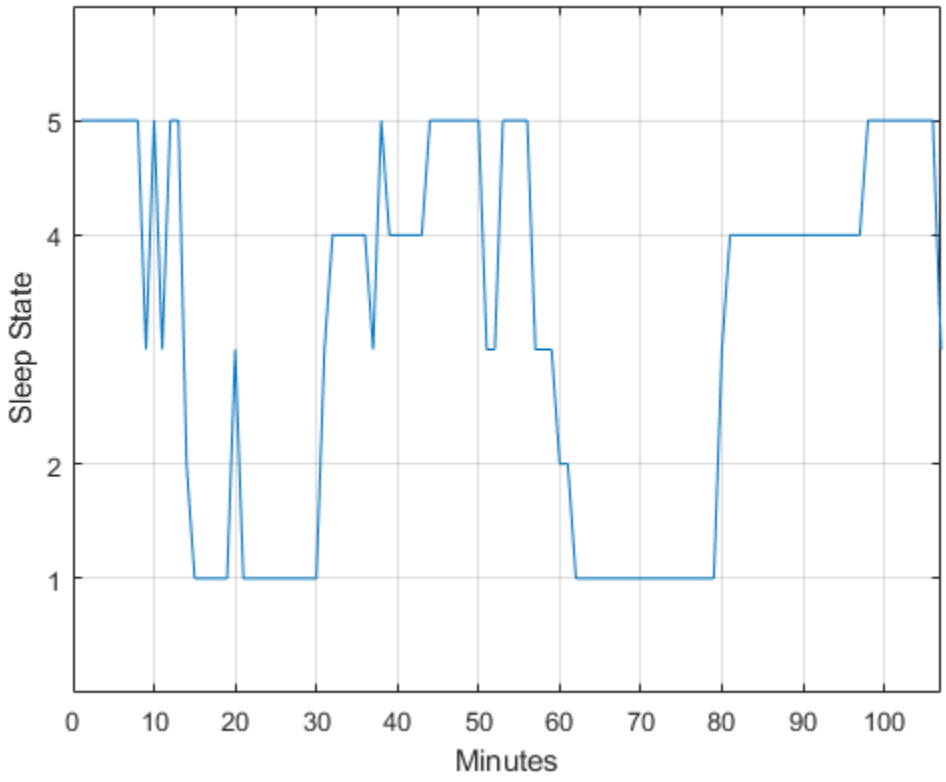
Enter the data. The infant was never awake during the EEG recording.

```
data = {'ah','ah','ah','ah','ah','ah','ah','ah','tr','ah','tr','ah', ...
    'ah','qh','qt','qt','qt','qt','qt','tr','qt','qt','qt','qt','qt', ...
    'qt','qt','qt','qt','qt','tr','al','al','al','al','al','tr','ah', ...
    'al','al','al','al','al','ah','ah','ah','ah','ah','ah','ah','tr', ...
    'tr','ah','ah','ah','ah','tr','tr','tr','qh','qh','qt','qt','qt', ...
    'qt','qt','qt','qt','qt','qt','qt','qt','qt','qt','qt','qt','qt', ...
    'qt','qt','tr','al','al','al','al','al','al','al','al','al','al', ...
    'al','al','al','al','al','al','al','ah','ah','ah','ah','ah','ah', ...
    'ah','ah','ah','tr'};

lend = length(data);
t = 1:lend;
```

The easiest way to analyze categorical-valued time series data for cyclic patterns involves assigning numerical values to the categories. There are at least two meaningful ways of assigning values to the infant's sleep states. First, note that you can order the six states from 1 to 6. This assignment makes sense along the scale of least active to most active.

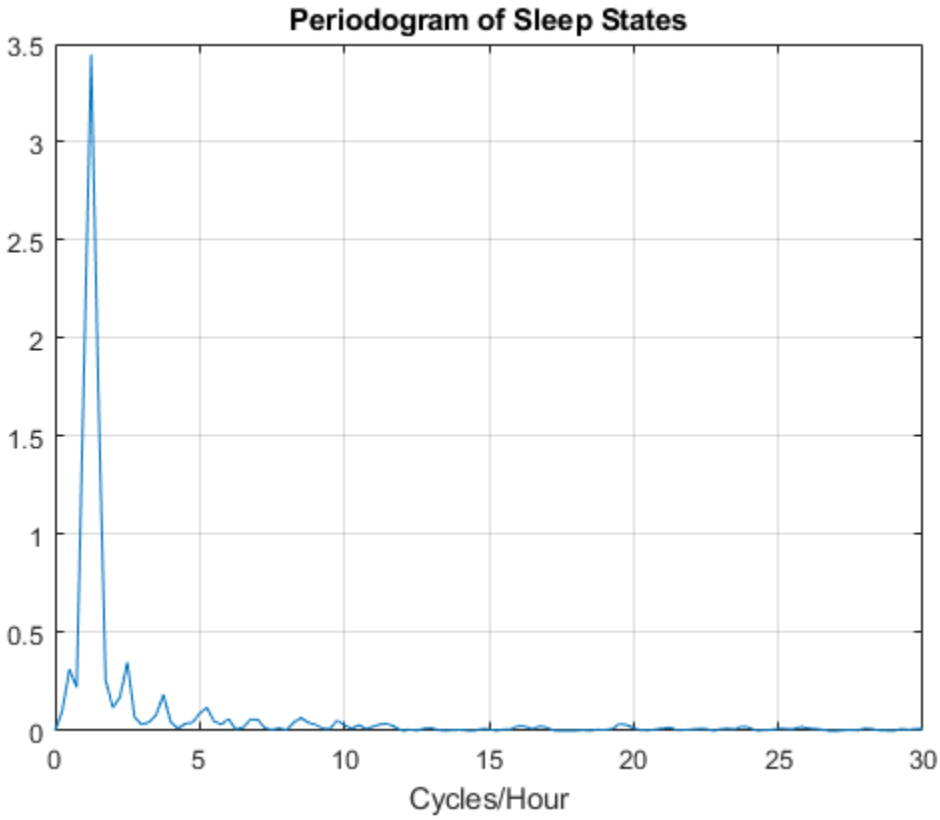
Replace the six sleep states with their numerical equivalents and plot the data.

```
states = ['qt';'qh';'tr';'al';'ah';'aw'];
levelssix = [1 2 3 4 5 6];

for nn = 1:6
    datasix(strcmp(data,states(nn,:))) = levelssix(nn);
end

plot(t,datasix)
axis([0 lend 0 6])
ax = gca;
ax.YTick = [1 2 4 5];
grid
xlabel('Minutes')
ylabel('Sleep State')
```




The data exhibit cyclic behavior when you focus on the transitions between the quietest states (1 and 2) and the most active ones (4 and 5). To determine the cycle of that behavior, use spectral analysis. Recall that the sleep states are assigned in one-minute intervals. Sampling the data in one-minute intervals is equivalent to sampling the data 60 times per hour.

```
Fs = 60;
[Pxx,F] = periodogram(detrend(datasix,0),[],240,Fs);

plot(F,Pxx)
grid
xlabel('Cycles/Hour')
title('Periodogram of Sleep States')
```




The spectral analysis shows a clear peak indicating a dominant oscillation, or cycle in the data. Determine the frequency of the peak.

```
[~,maxidx] = max(Pxx);
Fsix = F(maxidx)
```

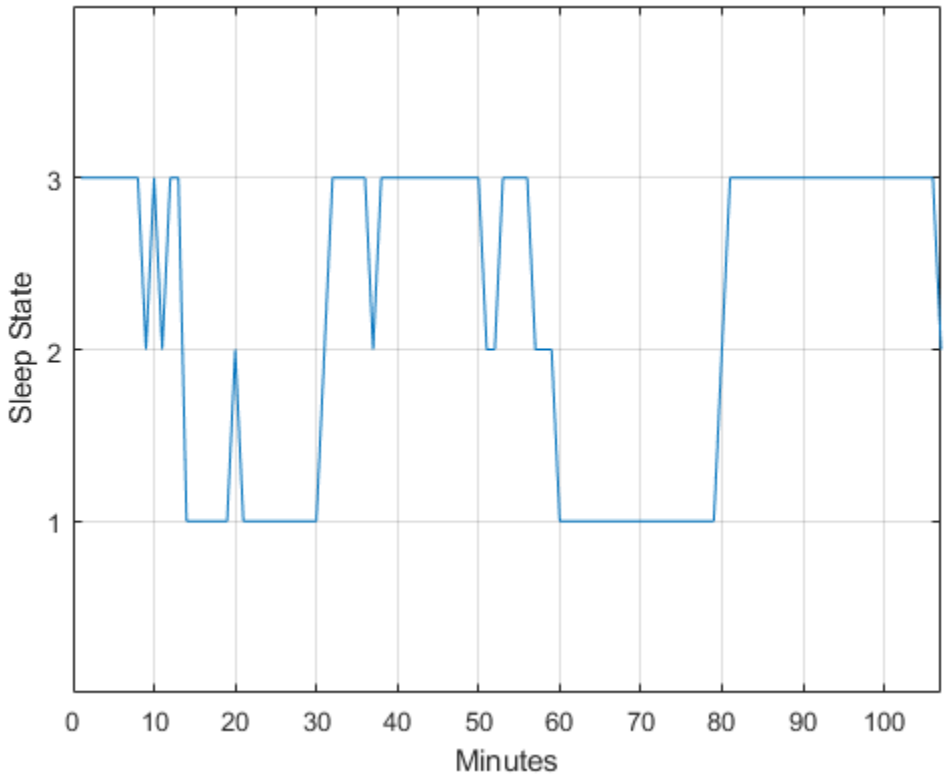
```
Fsix = 1.2500
```

The infant's sleep states exhibit cyclic behavior with a frequency of approximately 1.25 cycles/hour.

Instead of assigning the sleep states the values 1 to 6, repeat the analysis focusing only on the distinction between quiet and active sleep. Assign the quiet states, `qt` and `qh`, the value 1. Assign the transitional state, `tr`, the value 2. Finally, assign the two active sleep states, `al` and `ah`, the value 3. For completeness, assign the awake state, `aw`, the value 4, even though the state does not occur in the data.

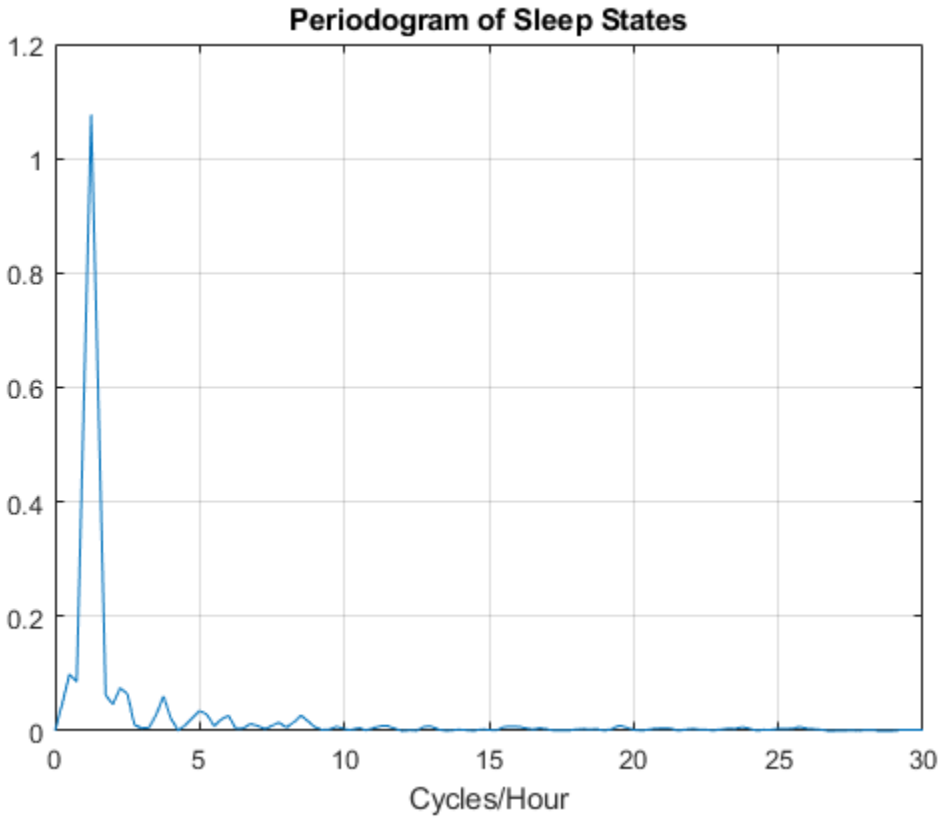
```
states = ['qt';'qh';'tr';'al';'ah';'aw'];
levelsfou = [1 1 2 3 3 4];

for nn = 1:6
    datafou(strcmp(data,states(nn,:))) = levelsfou(nn);
end

plot(t,datafou)
axis([0 lend 0 4])
ax = gca;
ax.YTick = [1 2 3];
```

```
grid
xlabel('Minutes')
ylabel('Sleep State')
```




With this rule of assignment between the sleep states and the values 1 to 3, the cyclic behavior of the data is clearer. Repeat the spectral analysis with the new assignment.

```
[Pxx,F] = periodogram(detrend(datafou,0),[],240,Fs);

plot(F,Pxx)
grid
xlabel('Cycles/Hour')
title('Periodogram of Sleep States')
```




```
[maxval,maxidx] = max(Pxx);
F(maxidx)
```

```
ans = 1.2500
```

The new assignment has not changed the conclusion. The data show a dominant oscillation at 1.25 cycles/hour. Because the mapping between the sleep states and the integers representing those states was consistent, the analysis and conclusions were not affected. Based on a spectral analysis of this categorical data, you conclude that the infant's sleep state cycles between quiet and active sleep approximately once every hour.

**References**

Stoffer, David S., Mark S. Scher, Gale A. Richardson, Nancy L. Day, and Patricia A. Coble. "A Walsh-Fourier Analysis of the Effects of Moderate Maternal Alcohol Consumption on Neonatal Sleep-State Cycling." *Journal of the American Statistical Association.* Vol. 83, 1988, pp. 954-963.

Stoffer, David S., D. E. Tyler, and D. A. Wendt. "The Spectral Envelope and Its Applications." *Statistical Science.* Vol. 15, 2000, pp. 224-253.

## See Also

`detrend` | `periodogram`

# Compensate for the Delay Introduced by an FIR Filter

Filtering a signal introduces a delay. This means that the output signal is shifted in time with respect to the input. This example shows you how to counteract this effect.

Finite impulse response filters often delay all frequency components by the same amount. This makes it easy to correct for the delay by shifting the signal in time.

Take an electrocardiogram reading sampled at 500 Hz for 1 s. Add random noise. Reset the random number generator for reproducibility.

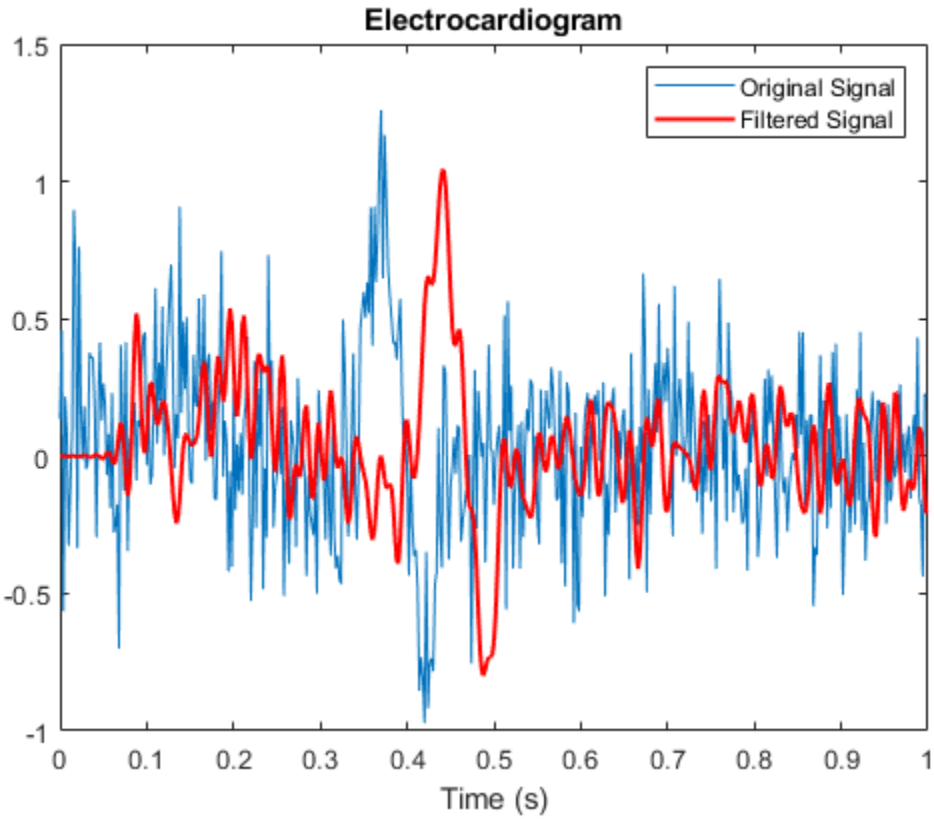
```
Fs = 500;
N = 500;
rng default

xn = ecg(N)+0.25*randn([1 N]);
tn = (0:N-1)/Fs;
```

Remove some of the noise with a filter that stops frequencies above 75 Hz. Use `designfilt` to design a filter of order 70.

```
nfilt = 70;
Fst = 75;

d = designfilt('lowpassfir','FilterOrder',nfilt, ...
               'CutoffFrequency',Fst,'SampleRate',Fs);
```

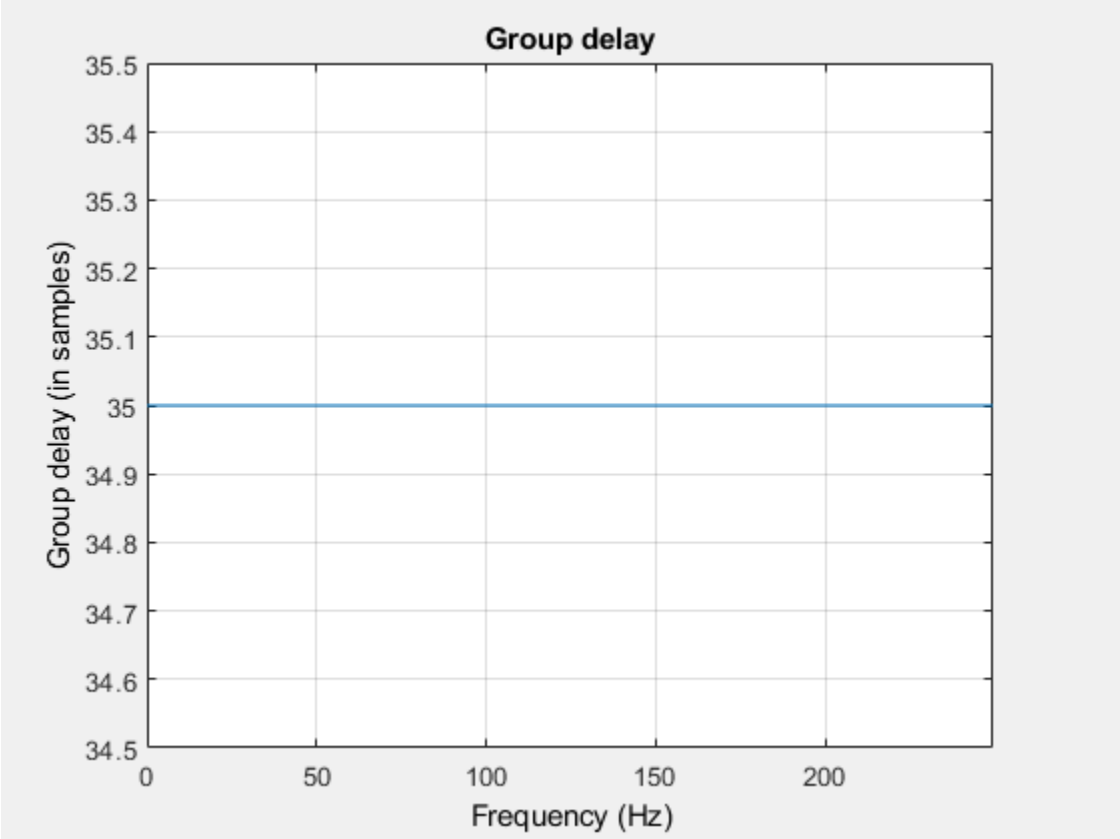
Filter the signal and plot it. The result is smoother than the original, but lags behind it.

```
xf = filter(d,xn);

plot(tn,xn)
hold on, plot(tn,xf,'-r','linewidth',1.5), hold off
title 'Electrocardiogram'
xlabel 'Time (s)', legend('Original Signal','Filtered Signal')
```




Use `grpdelay` to check that the delay caused by the filter equals half the filter order.

```
grpdelay(d,N,Fs)
```




```
delay = mean(grpdelay(d))
```

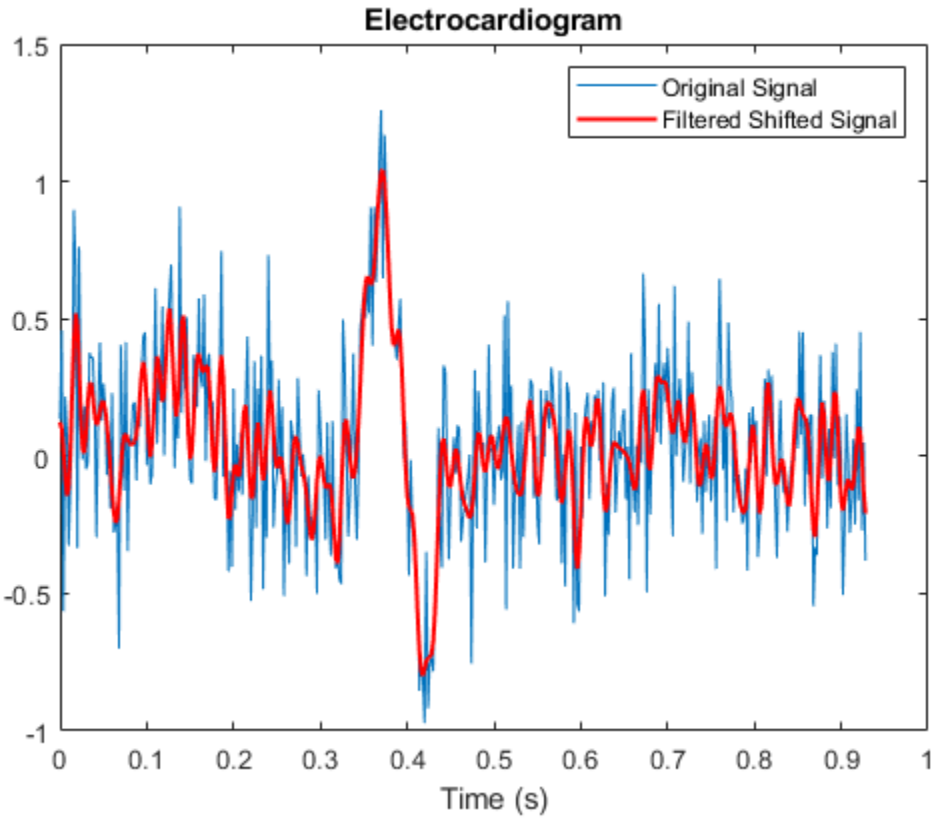
```
delay = 35
```

Shift the filtered signal to line up the data. Remove its first `delay` samples. Remove the last `delay` samples of the original and of the time vector.

```
tt = tn(1:end-delay);
sn = xn(1:end-delay);

sf = xf;
sf(1:delay) = [];
```

Plot the signals and verify that they are aligned.

```
plot(tt,sn)
hold on, plot(tt,sf,'-r','linewidth',1.5), hold off
title 'Electrocardiogram'
xlabel('Time (s)'), legend('Original Signal','Filtered Shifted Signal')
```




## See Also

`designfilt` | `filter` | `filtfilt` | `grpdelay`

## Related Examples

- “Compensate for the Delay Introduced by an IIR Filter” on page 23-63
- “Practical Introduction to Digital Filtering” on page 24-174

# Compensate for the Delay Introduced by an IIR Filter

Filtering a signal introduces a delay. This means that the output signal is shifted in time with respect to the input.

Infinite impulse response filters delay some frequency components more than others. They effectively distort the input signal. The function `filtfilt` compensates for the delays introduced by such filters, and thus corrects for filter distortion. This "zero-phase filtering" results from filtering the signal in the forward and backward directions.

Take an electrocardiogram reading sampled at 500 Hz for 1 s. Add random noise.

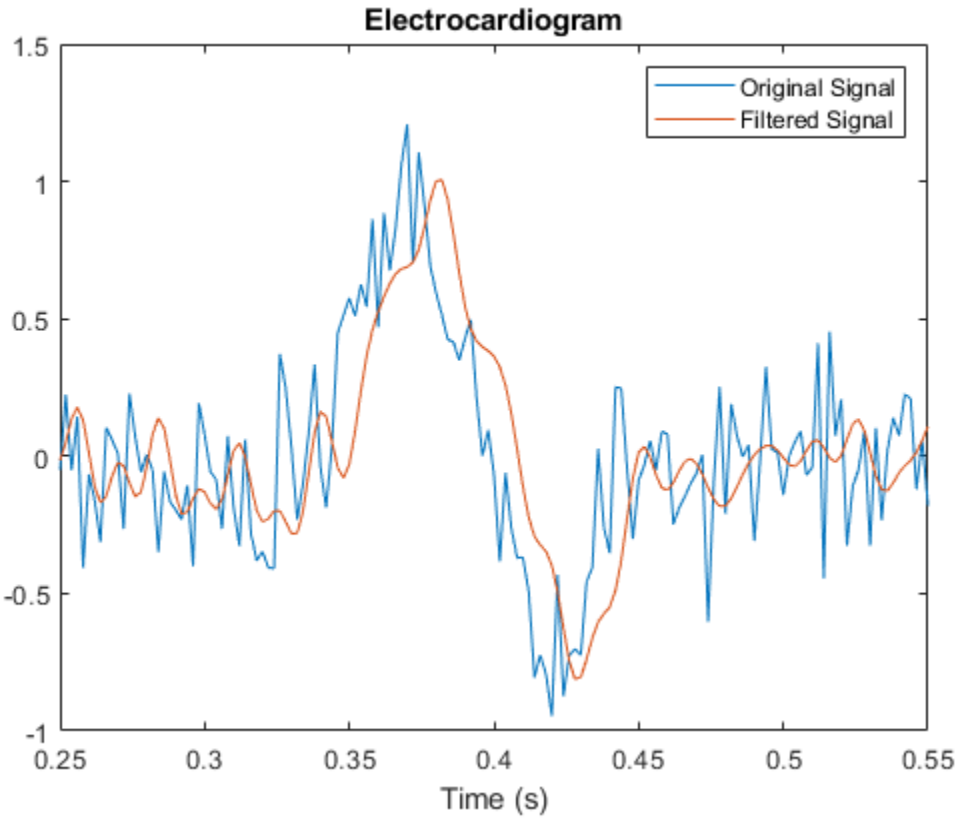
```
Fs = 500;
N = 500;

rng default
xn = ecg(N) + 0.2*randn([1 N]);
tn = (0:N-1)/Fs;
```

Remove some of the noise with a filter that stops frequencies above 75 Hz. Specify a 7th-order IIR filter with 1 dB of passband ripple and 60 dB of stopband attenuation.

```
Nf = 7;
Fp = 75;
Ap = 1;
As = 60;

d = designfilt('lowpassiir','FilterOrder',Nf,'PassbandFrequency',Fp, ...
    'PassbandRipple',Ap,'StopbandAttenuation',As,'SampleRate',Fs);
```

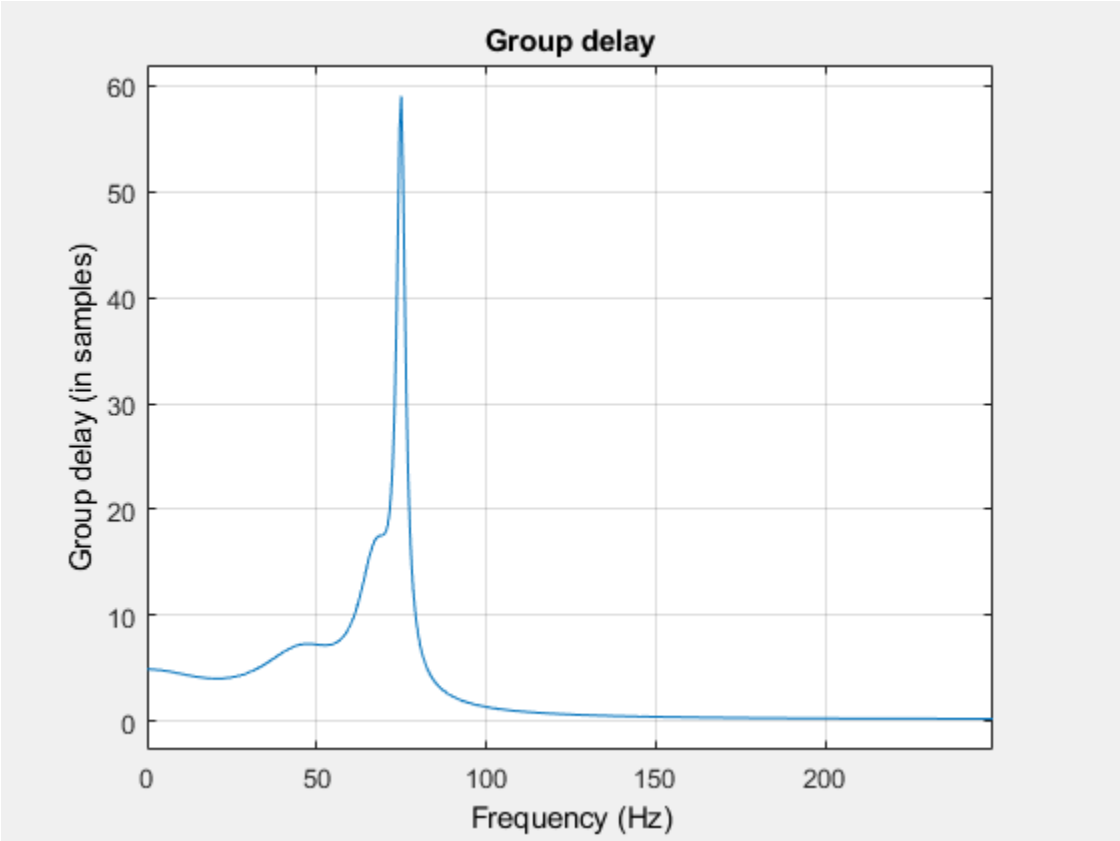
Filter the signal. The filtered signal is cleaner than the original, but lags in time with respect to it. It is also distorted due to the nonlinear phase of the filter. Zoom in close to the peak.

```
xfilter = filter(d,xn);

plot(tn,xn,tn,xfilter)

title 'Electrocardiogram'
xlabel 'Time (s)', legend('Original Signal','Filtered Signal')
axis([0.25 0.55 -1 1.5])
```



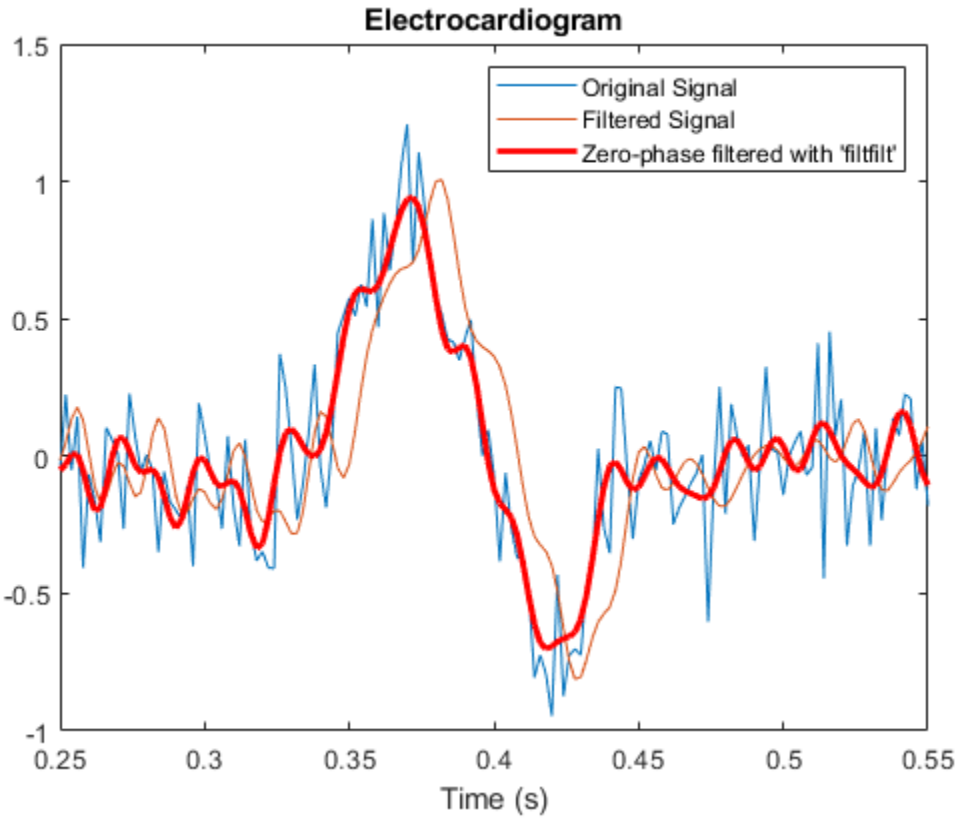

A look at the *group delay* introduced by the filter shows that the delay is frequency-dependent.

```
grpdelay(d,N,Fs)
```




Filter the signal using `filtfilt`. The delay and distortion have been effectively removed. Use `filtfilt` when it is critical to keep the phase information of a signal intact.

```
xfiltfilt = filtfilt(d,xn);

plot(tn,xn,tn,xfilter)
hold on
plot(tn,xfiltfilt,'r','linewidth',2)
hold off

title 'Electrocardiogram'
xlabel 'Time (s)'
legend('Original Signal','Filtered Signal', ...
       'Zero-phase filtered with ''filtfilt'''')
axis([0.25 0.55 -1 1.5])
```




## See Also

`designfilt` | `filter` | `filtfilt` | `grpdelay`

## Related Examples

- “Compensate for the Delay Introduced by an FIR Filter” on page 23-59
- “Practical Introduction to Digital Filtering” on page 24-174

# Take Derivatives of a Signal

You want to differentiate a signal without increasing the noise power. MATLAB®'s function `diff` amplifies the noise, and the resulting inaccuracy worsens for higher derivatives. To fix this problem, use a differentiator filter instead.

Analyze the displacement of a building floor during an earthquake. Find the speed and acceleration as functions of time.

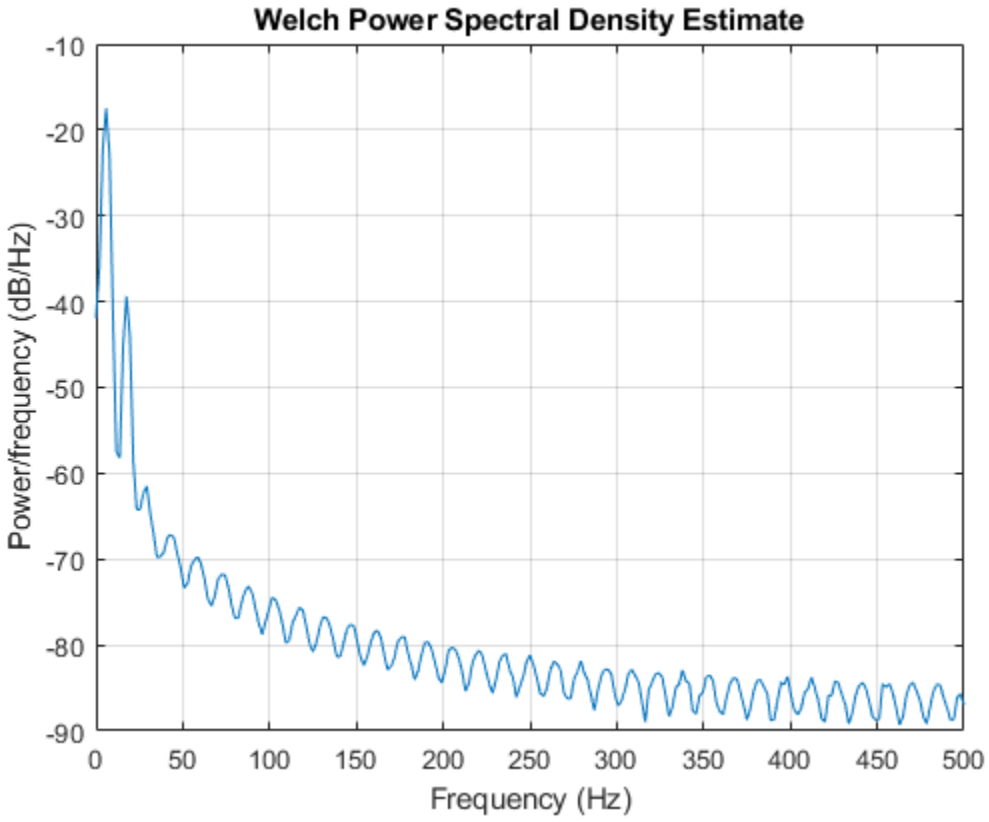
Load the file `earthquake`. The file contains the following variables:

- `drift`: Floor displacement, measured in centimeters
- `t`: Time, measured in seconds
- `Fs`: Sample rate, equal to 1 kHz

```
load('earthquake.mat')
```

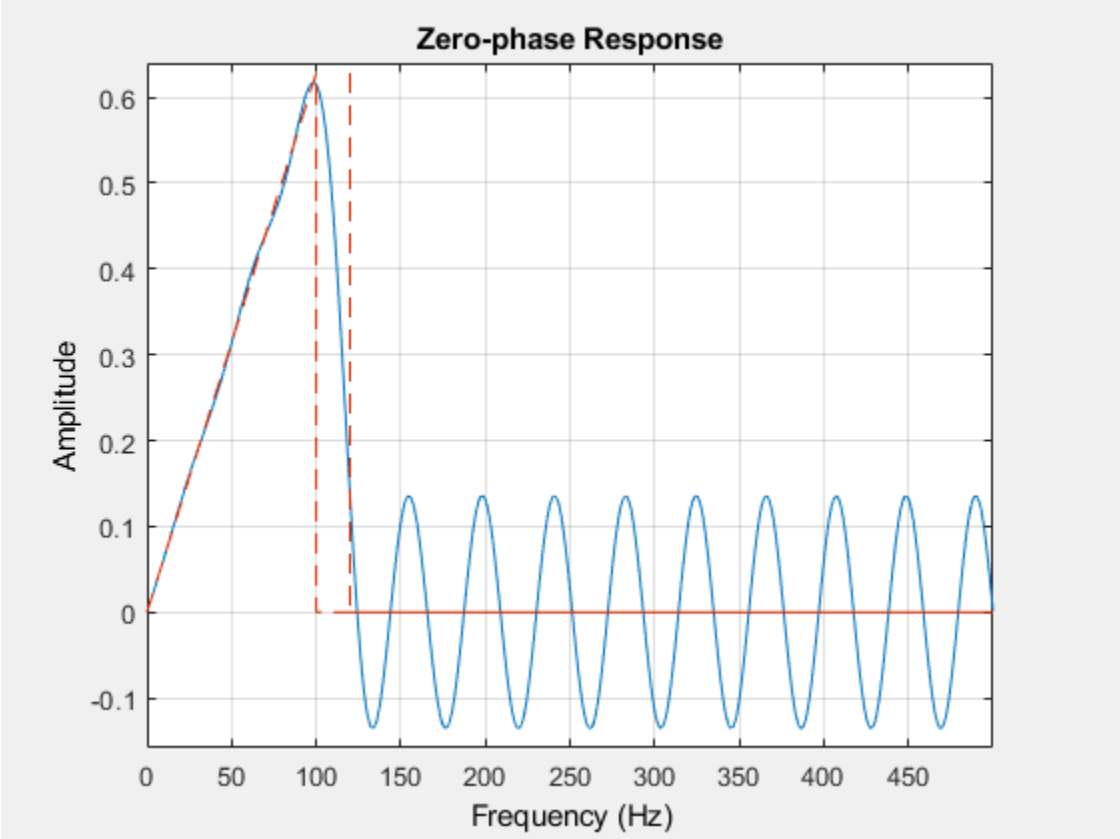
Use `pwelch` to display an estimate of the power spectrum of the signal. Note how most of the signal energy is contained in frequencies below 100 Hz.

```
pwelch(drift,[],[],[],Fs)
```




Use `designfilt` to design an FIR differentiator of order 50. To include most of the signal energy, specify a passband frequency of 100 Hz and a stopband frequency of 120 Hz. Inspect the filter with `fvtool`.

```
Nf = 50;
Fpass = 100;
Fstop = 120;

d = designfilt('differentiatorfir','FilterOrder',Nf, ...
    'PassbandFrequency',Fpass,'StopbandFrequency',Fstop, ...
    'SampleRate',Fs);

fvtool(d,'MagnitudeDisplay','zero-phase','Fs',Fs)
```




Differentiate the drift to find the speed. Divide the derivative by `dt`, the time interval between consecutive samples, to set the correct units.

```
dt = t(2)-t(1);

vdrift = filter(d,drift)/dt;
```

The filtered signal is delayed. Use `grpdelay` to determine that the delay is half the filter order. Compensate for it by discarding samples.

```
delay = mean(grpdelay(d))
```

```
delay = 25
```

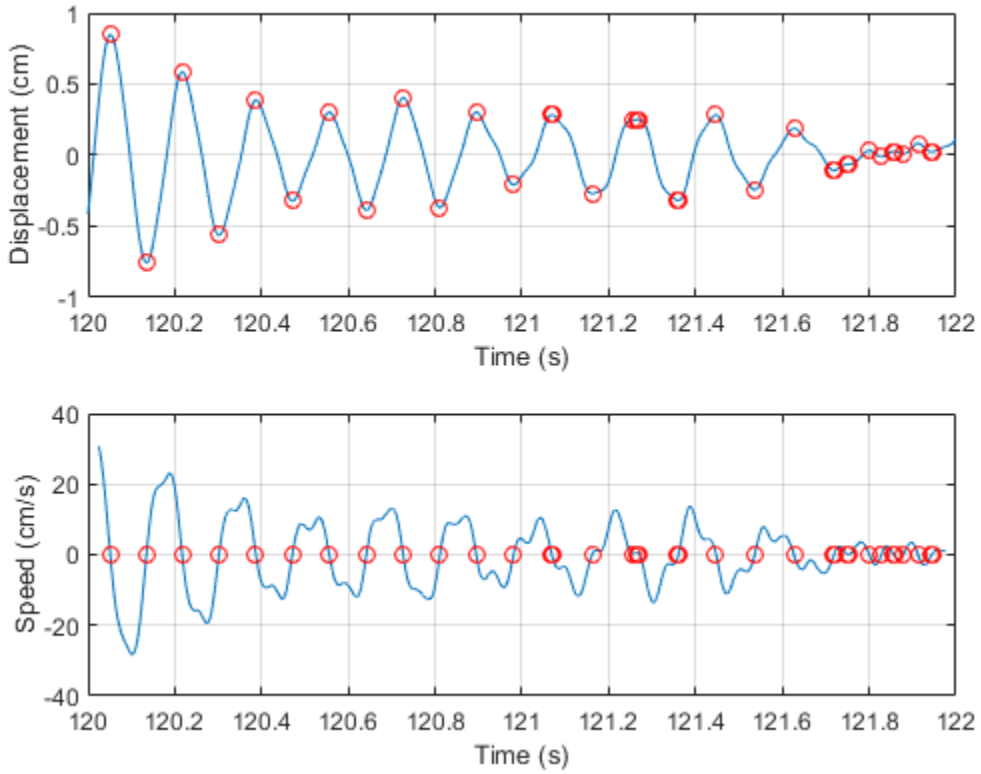
```
tt = t(1:end-delay);
vd = vdrift;
vd(1:delay) = [];
```

The output also includes a transient whose length equals the filter order, or twice the group delay. `delay` samples were discarded above. Discard `delay` more to eliminate the transient.

```
tt(1:delay) = [];
vd(1:delay) = [];
```

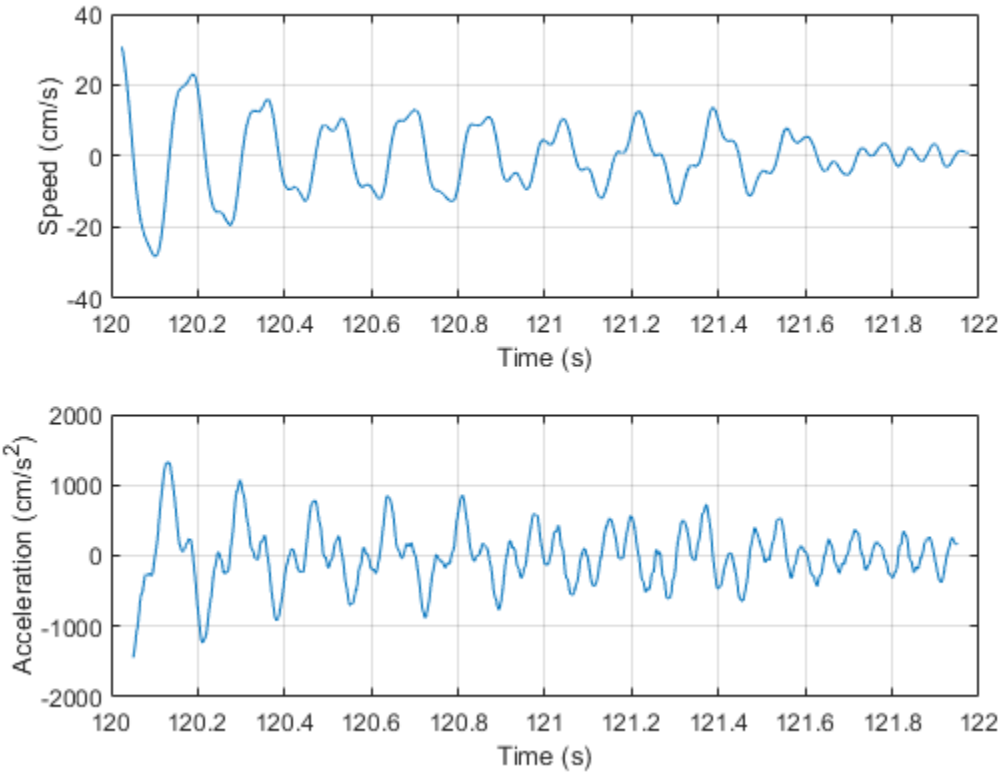
Plot the drift and the drift speed. Use `findpeaks` to verify that the maxima and minima of the drift correspond to the zero crossings of its derivative.

```
[pkp,lcp] = findpeaks(drift);
zcp = zeros(size(lcp));

[pkm,lcm] = findpeaks(-drift);
zcm = zeros(size(lcm));

subplot(2,1,1)
plot(t,drift,t([lcp lcm]),[pkp -pkm],'or')
xlabel('Time (s)')
ylabel('Displacement (cm)')
grid

subplot(2,1,2)
plot(tt,vd,t([lcp lcm]),[zcp zcm],'or')
xlabel('Time (s)')
ylabel('Speed (cm/s)')
grid
```



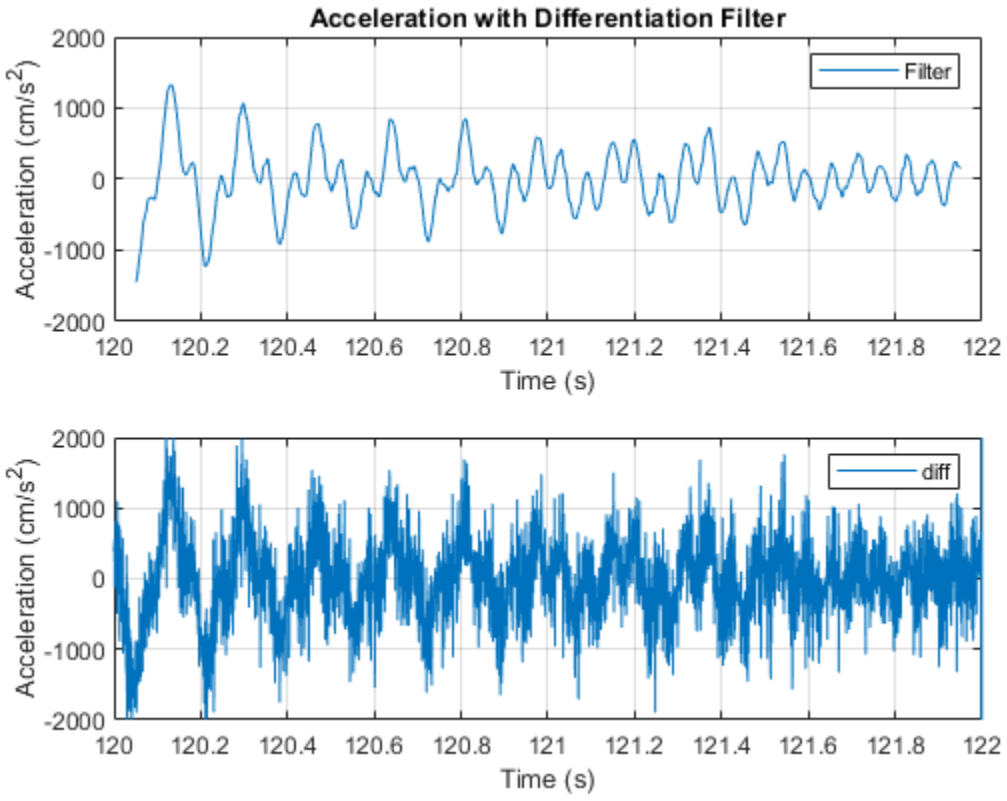
Differentiate the drift speed to find the acceleration. The lag is twice as long. Discard twice as many samples to compensate for the delay, and the same number to eliminate the transient. Plot the speed and acceleration.

```
adrift = filter(d,vdrift)/dt;

at = t(1:end-2*delay);
ad = adrift;
ad(1:2*delay) = [];

at(1:2*delay) = [];
ad(1:2*delay) = [];

subplot(2,1,1)
plot(tt,vd)
xlabel('Time (s)')
ylabel('Speed (cm/s)')
grid

subplot(2,1,2)
plot(at,ad)
ax = gca;
ax.YLim = 2000*[-1 1];
xlabel('Time (s)')
ylabel('Acceleration (cm/s^2)')
grid
```



Compute the acceleration using `diff`. Add zeros to compensate for the change in array size. Compare the result to that obtained with the filter. Notice the amount of high-frequency noise.

```
vdiff = diff([drift;0])/dt;
adiff = diff([vdiff;0])/dt;

subplot(2,1,1)
plot(at,ad)
ax = gca;
ax.YLim = 2000*[-1 1];
xlabel('Time (s)')
ylabel('Acceleration (cm/s^2)')
grid
legend('Filter')
title('Acceleration with Differentiation Filter')

subplot(2,1,2)
plot(t,adiff)
ax = gca;
ax.YLim = 2000*[-1 1];
xlabel('Time (s)')
ylabel('Acceleration (cm/s^2)')
grid
legend('diff')
```



## See Also

**FVTool** | `designfilt` | `findpeaks` | `grpdelay` | `periodogram`

## Related Examples

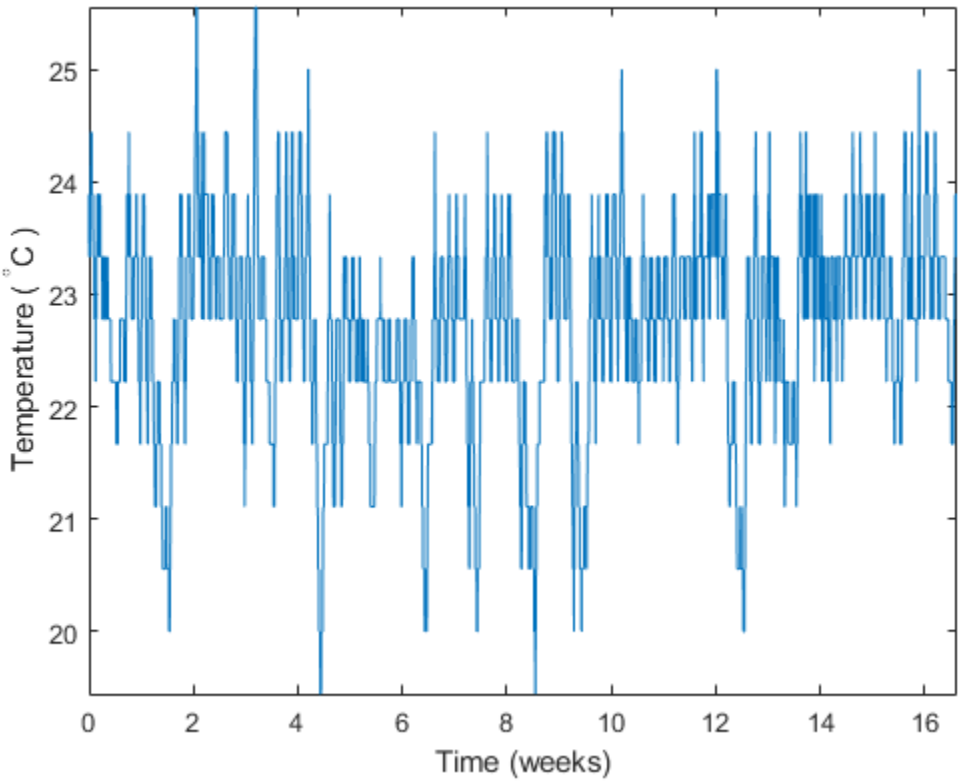
- “Practical Introduction to Digital Filtering” on page 24-174

# Find Periodicity Using Frequency Analysis

It is often difficult to characterize oscillatory behavior in data by looking at time measurements. Spectral analysis can help determine if a signal is periodic and measure the different cycles.

A thermometer in an office building measures the inside temperature every half hour for four months. Load the data and plot it. Convert the temperature to degrees Celsius. Measure time in weeks. The sample rate is thus 2 measurements/hour × 24 hours/day × 7 days/week = 336 measurements/week.

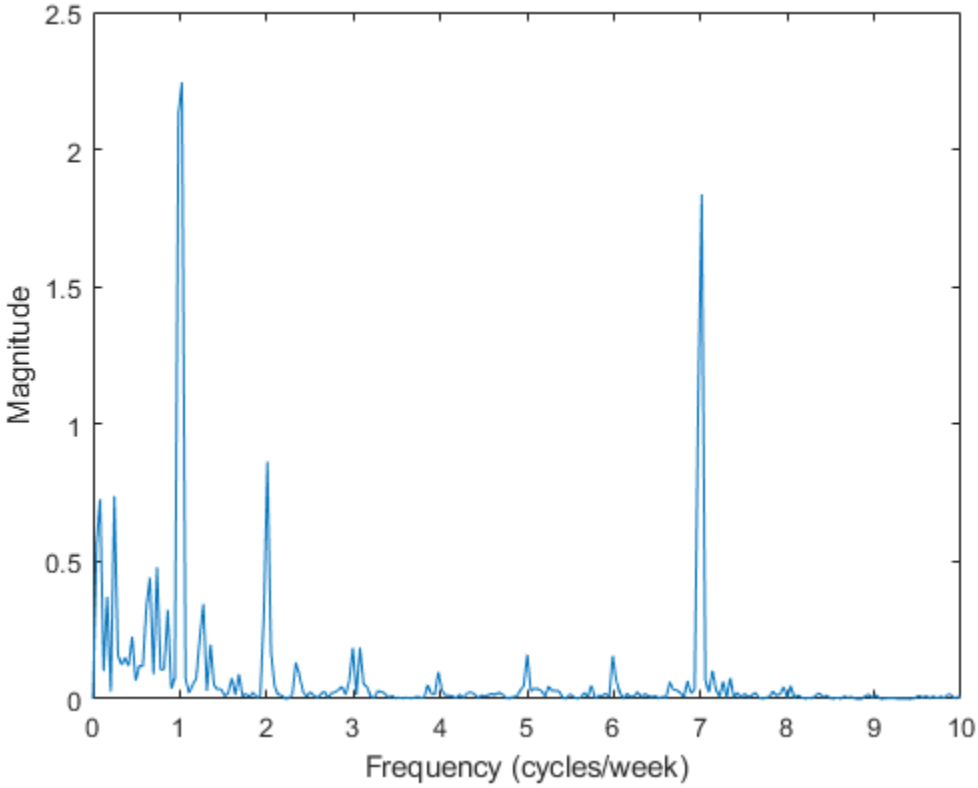
```
load officetemp

tempC = (temp - 32)*5/9;

fs = 2*24*7;
t = (0:length(tempC) - 1)/fs;

plot(t,tempC)
xlabel('Time (weeks)')
ylabel('Temperature ( {}^\circC )')
axis tight
```




The temperature does seem to oscillate, but the lengths of the cycles cannot be determined easily. Look at the signal's frequency content instead.

Subtract the mean to concentrate on temperature fluctuations. Compute and plot the periodogram.

```
tempnorm = tempC - mean(tempC);

[pxx,f] = periodogram(tempnorm,[],[],fs);

plot(f,pxx)
ax = gca;
ax.XLim = [0 10];
xlabel('Frequency (cycles/week)')
ylabel('Magnitude')
```




The temperature clearly has a daily cycle and a weekly cycle. The result is not surprising: the temperature is higher when people are at work and lower at nights and on weekends.

## See Also

`findpeaks` | `periodogram` | `xcorr`

## Related Examples

- “Find Periodicity Using Autocorrelation” on page 23-45
- “Practical Introduction to Frequency-Domain Analysis” on page 24-254

# Detect a Distorted Signal in Noise

The presence of noise often makes it difficult to determine the spectral content of a signal. Frequency analysis can help in such cases.

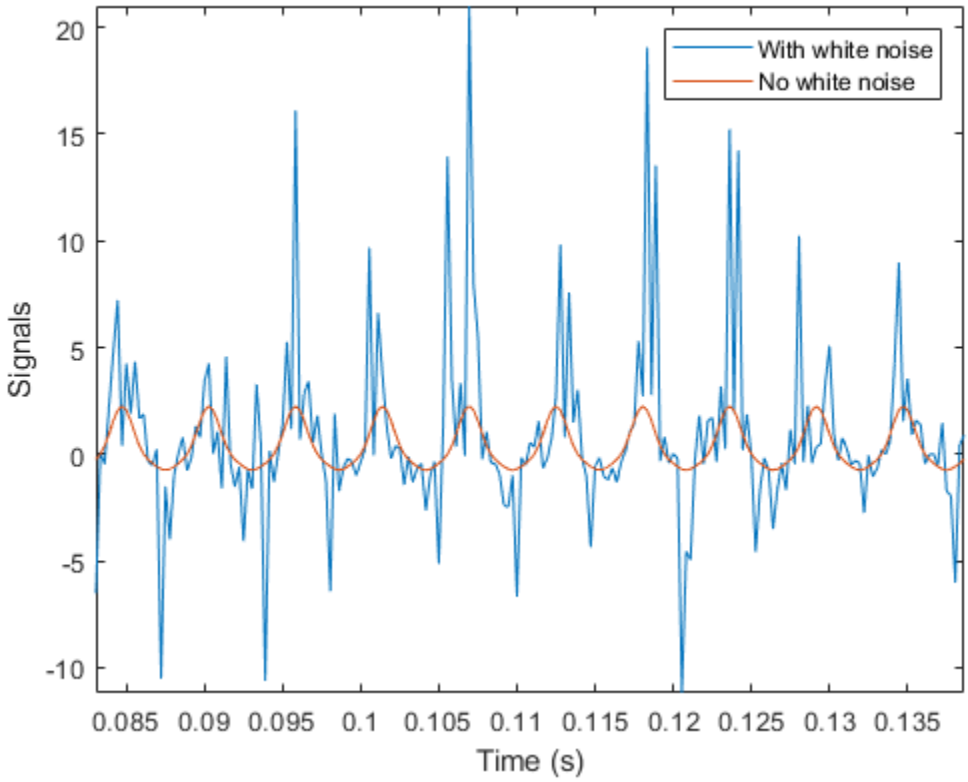
Consider for example the simulated output of a nonlinear amplifier that introduces third-order distortion.

The input signal is a 180 Hz unit-amplitude sinusoid sampled at 3.6 kHz. Generate 10000 samples.

```
N = 1e4;
n = 0:N-1;
fs = 3600;
f0 = 180;
t = n/fs;
y = sin(2*pi*f0*t);
```

Add unit-variance white noise to the input. Model the amplifier using a third-order polynomial. Pass the input signal through the amplifier using `polyval`. Plot a section of the output. For comparison plot the output of a pure sinusoid.

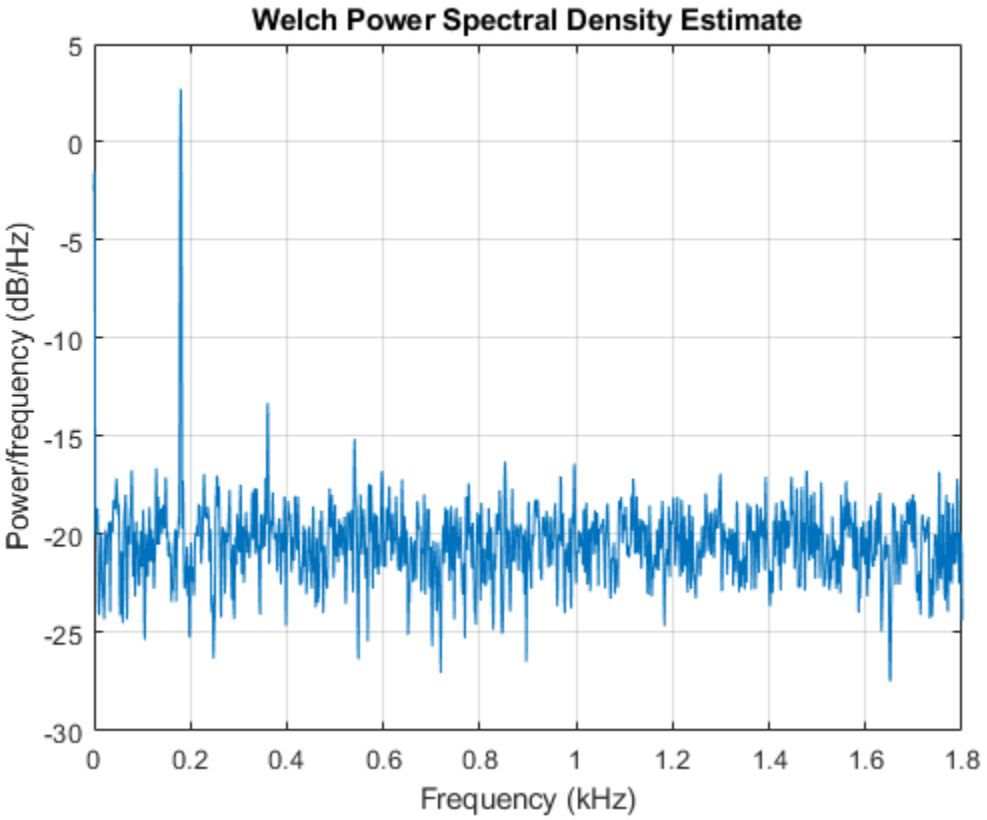
```
rng default
noise = randn(size(y));

dispol = [0.5 0.75 1 0];
out = polyval(dispol,y+noise);

ns = 300:500;

plot(t(ns),[out(ns);polyval(dispol,y(ns))])
xlabel('Time (s)')
ylabel('Signals')
axis tight
legend('With white noise','No white noise')
```




Use `pwelch` to compute and plot the power spectral density of the output.

```
[pxx,f] = pwelch(out,[],[],[],fs);
```

```
pwelch(out,[],[],[],fs)
```

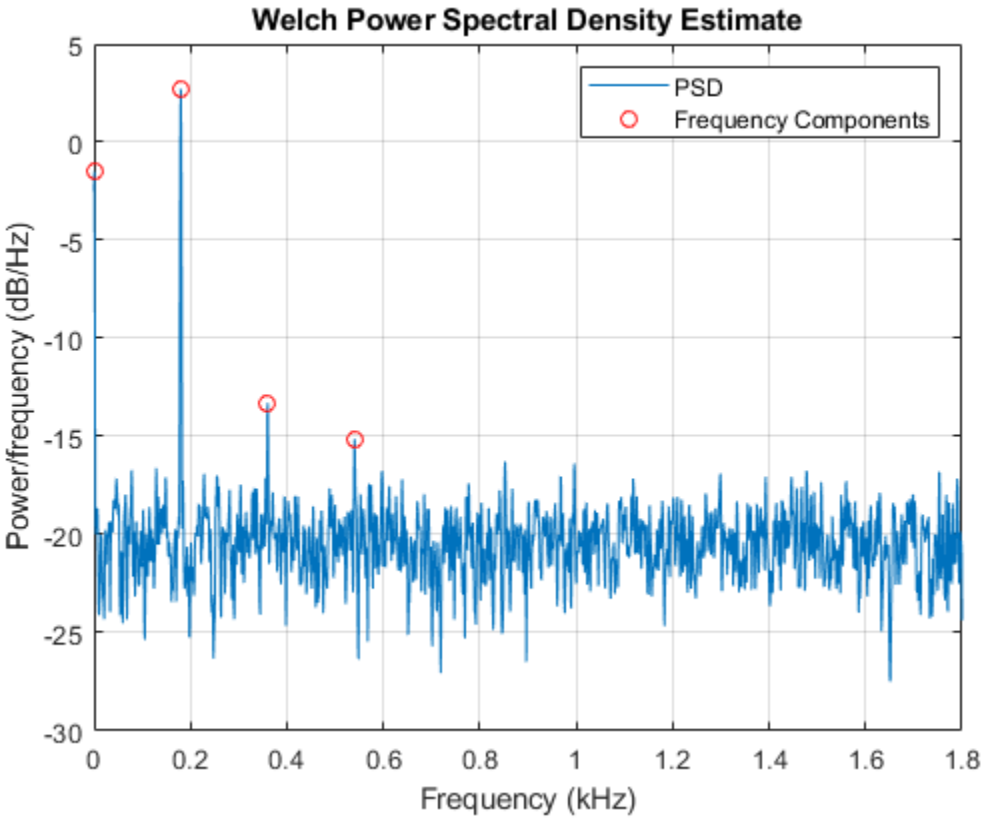



Because the amplifier introduces third-order distortion, the output signal is expected to have:

- A *DC* (zero-frequency) component;
- A *fundamental* component with the same frequency as the input, 180 Hz;
- Two *harmonics* -- frequency components at twice and three times the frequency of the input, 360 and 540 Hz.

Verify that the output is as expected for a cubic nonlinearity.

```
[pks,lox] = findpeaks(pxx,'NPeaks',4,'SortStr','descend');

hold on
plot(f(lox)/1000,10*log10(pks),'or')
hold off

legend('PSD','Frequency Components')
```



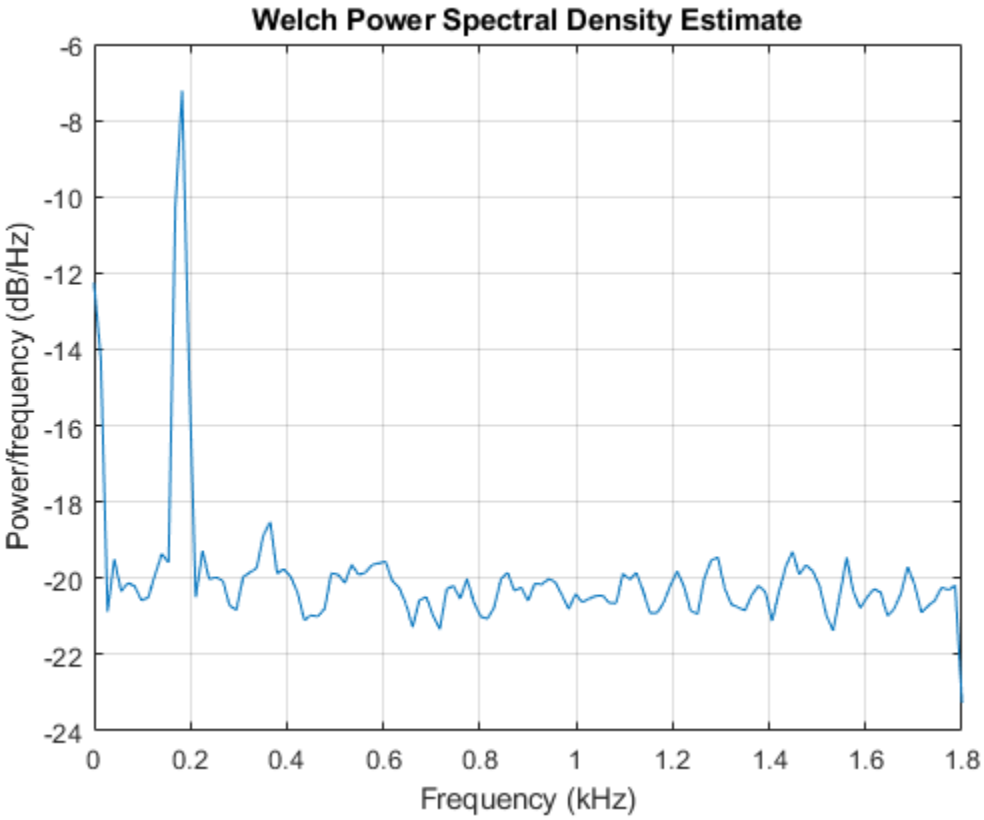

```
components = sort([f(lox) f0*(0:3)'])'
```

```
components = 2×4

    0.8789  180.1758  360.3516  540.5273
         0  180.0000  360.0000  540.0000
```

`pwelch` works by dividing the signal into overlapping segments, computing the periodogram of each segment, and averaging. By default, the function uses eight segments with 50% overlap. For 10000 samples, this corresponds to 2222 samples per segment.

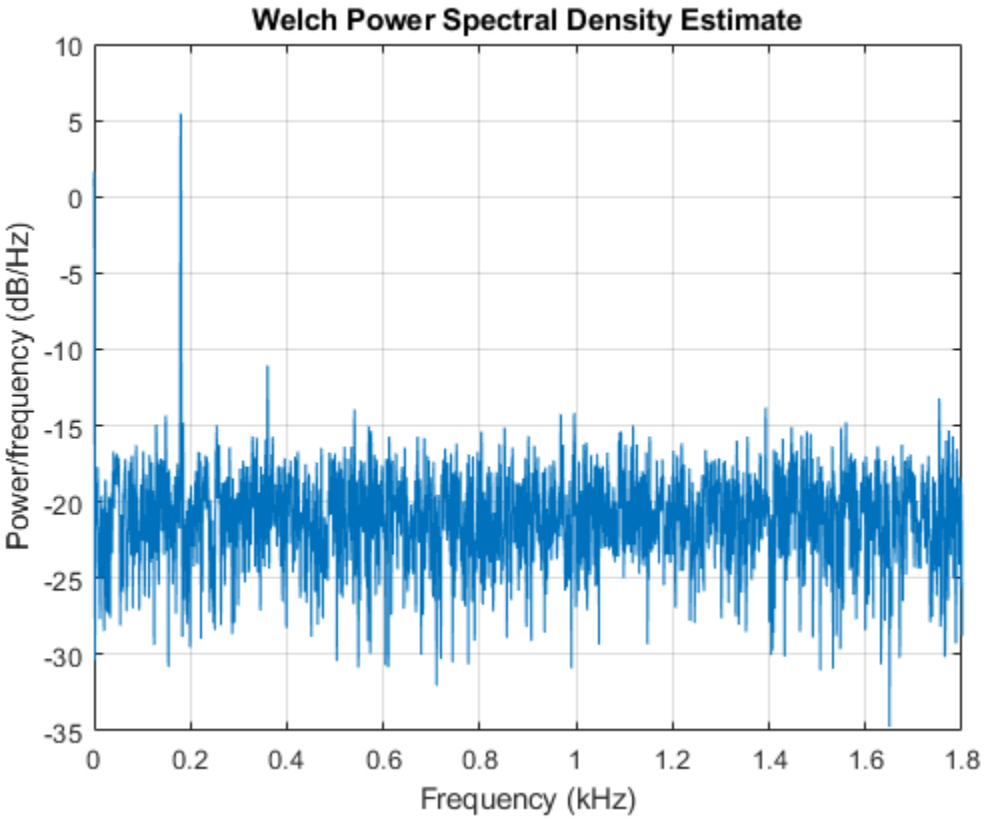
Dividing the signal into shorter segments results in more averaging. The periodogram is smoother, but has lower resolution. The higher harmonic cannot be distinguished.

```
pwelch(out,222,[],[],fs)
```




Dividing the signal into longer segments increases the resolution, but also the randomness. The signal and the harmonics are precisely at the expected locations. However, there is at least one spurious high-frequency peak with more power than the higher harmonic.

```
pwelch(out,4444,[],[],fs)
```




## See Also

`findpeaks` | `pwelch`

## Related Examples

- “Practical Introduction to Frequency-Domain Analysis” on page 24-254

# Measure the Power of a Signal

The power of a signal is the sum of the absolute squares of its time-domain samples divided by the signal length, or, equivalently, the square of its RMS level. The function `bandpower` allows you to estimate signal power in one step.

Consider a unit chirp embedded in white Gaussian noise and sampled at 1 kHz for 1.2 seconds. The chirp's frequency increases in one second from an initial value of 100 Hz to 300 Hz. The noise has variance $0.01^2$. Reset the random number generator for reproducible results.

```
N = 1200;
Fs = 1000;
t = (0:N-1)/Fs;

sigma = 0.01;
rng('default')

s = chirp(t,100,1,300)+sigma*randn(size(t));
```

Verify that the power estimate given by `bandpower` is equivalent to the definition.

```
pRMS = rms(s)^2
```

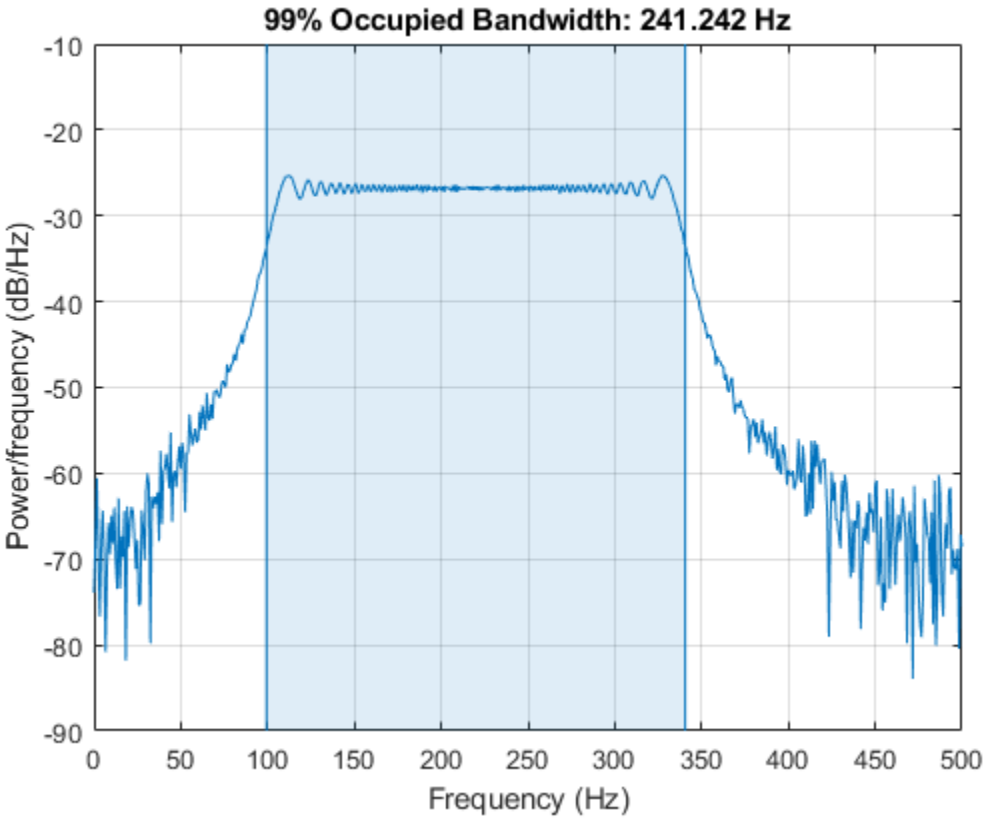
```
pRMS = 0.5003
```

```
powbp = bandpower(s,Fs,[0 Fs/2])
```

```
powbp = 0.5005
```

Use the `obw` function to estimate the width of the frequency band that contains 99% of the power of the signal, the lower and upper bounds of the band, and the power in the band. The function also plots the spectrum estimate and annotates the occupied bandwidth.

```
obw(s,Fs);
```




```
[wd,lo,hi,power] = obw(s,Fs);
powtot = power/0.99
```

```
powtot = 0.5003
```

A nonlinear power amplifier is given a 60 Hz sinusoid as input and outputs a noisy signal with third-order distortion. The signal is sampled at 3.6 kHz for 2 seconds.

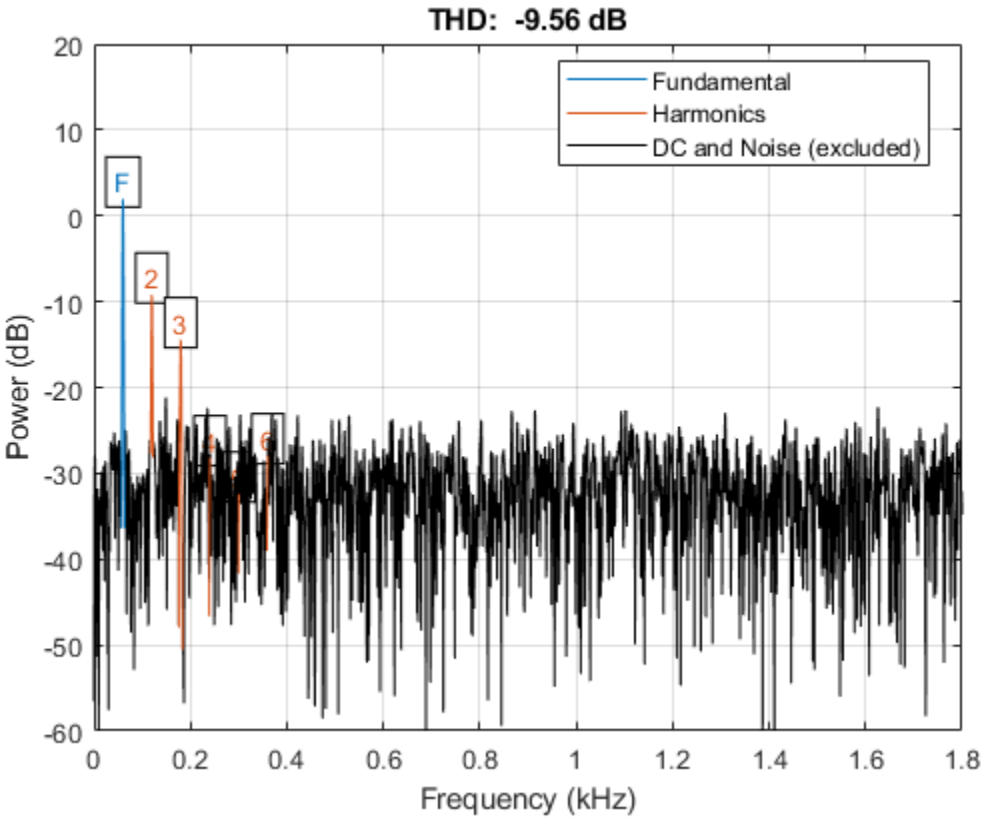
```
Fs = 3600;
t = 0:1/Fs:2-1/Fs;
x = sin(2*pi*60*t);
y = polyval(ones(1,4),x) + randn(size(x));
```

Because the amplifier introduces third-order distortion, the output signal is expected to have:

- A *fundamental* component with the same frequency as the input, 60 Hz.
- Two *harmonics* — frequency components at twice and three times the frequency of the input, 120 and 180 Hz.

Use the `thd` function to visualize the spectrum of the signal. Annotate the fundamental and the harmonics.

```
thd(y,Fs);
```




Use `bandpower` to determine the power stored in the fundamental and the harmonics. Express each value as a percentage of the total power and in decibels. Display the values as a table.

```
pwrTot = bandpower(y,Fs,[0 Fs/2]);

Harmonic = {'Fundamental';'First';'Second'};

Freqs = [60 120 180]';

Power = zeros([3 1]);
for k = 1:3
    Power(k) = bandpower(y,Fs,Freqs(k)+[-10 10]);
end

Percent = Power/pwrTot*100;

inDB = pow2db(Power);

T = table(Freqs,Power,Percent,inDB,'RowNames',Harmonic)
```

```
T=3×4 table
                 Freqs     Power     Percent     inDB
                 _____    _______    _______    _______

    Fundamental    60       1.5777    31.788      1.9804
    First         120      0.13141    2.6476     -8.8137
    Second        180      0.04672    0.9413     -13.305
```

## See Also

`bandpower` | `pow2db` | `pwelch` | `snr`

## Related Examples

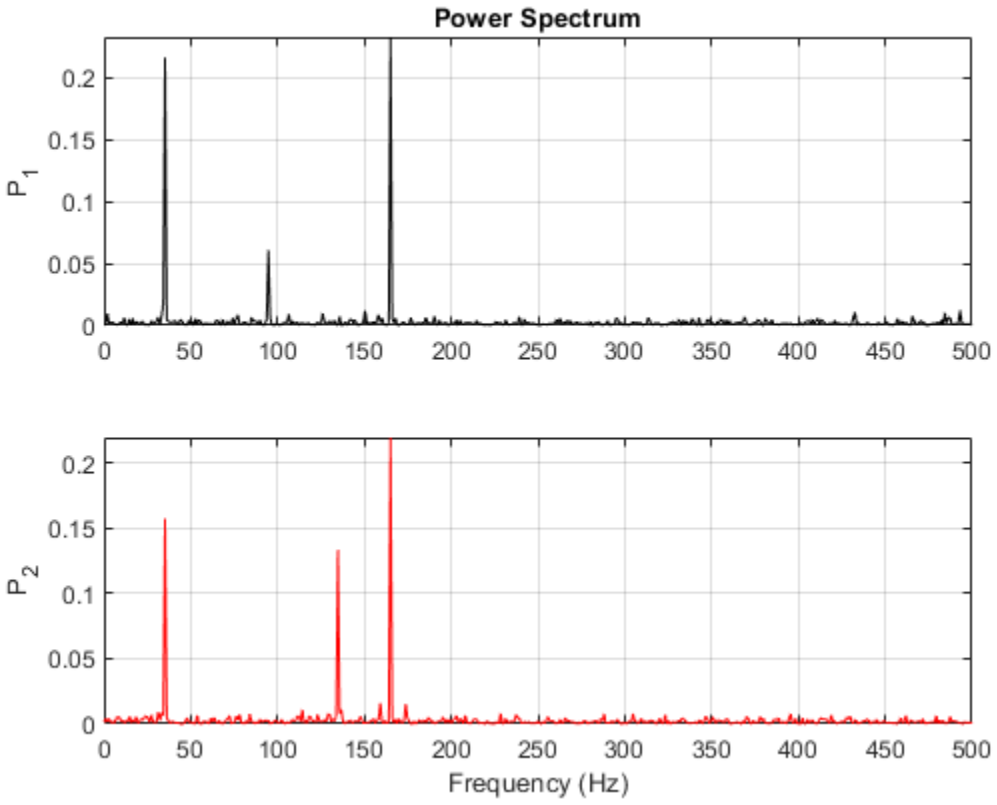
- “Practical Introduction to Frequency-Domain Analysis” on page 24-254

# Compare the Frequency Content of Two Signals

Spectral coherence helps identify similarity between signals in the frequency domain. Large values indicate frequency components common to the signals.

Load two sound signals into the workspace. They are sampled at 1 kHz. Compute their power spectra using `periodogram` and plot them next to each other.

```
load relatedsig

Fs = FsSig;

[P1,f1] = periodogram(sig1,[],[],Fs,'power');
[P2,f2] = periodogram(sig2,[],[],Fs,'power');

subplot(2,1,1)
plot(f1,P1,'k')
grid
ylabel('P_1')
title('Power Spectrum')

subplot(2,1,2)
plot(f2,P2,'r')
grid
ylabel('P_2')
xlabel('Frequency (Hz)')
```



Each signal has three frequency components with significant energy. Two of those components appear to be shared. Find the corresponding frequencies using `findpeaks`.

```
[pk1,lc1] = findpeaks(P1,'SortStr','descend','NPeaks',3);
P1peakFreqs = f1(lc1)
```

```
P1peakFreqs = 3×1

  165.0391
   35.1563
   94.7266
```

```
[pk2,lc2] = findpeaks(P2,'SortStr','descend','NPeaks',3);
P2peakFreqs = f2(lc2)
```

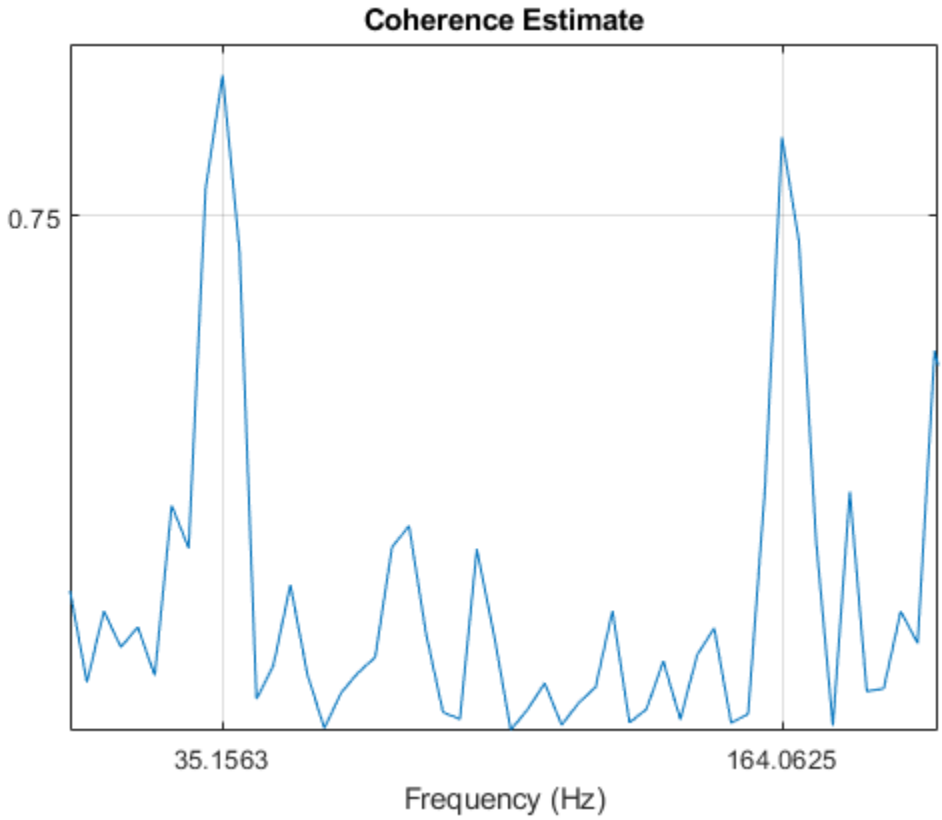
```
P2peakFreqs = 3×1

  165.0391
   35.1563
  134.7656
```

The common components are located around 165 and 35 Hz. You can use `mscohere` to find the matching frequencies directly. Plot the coherence estimate. Find the peaks above a threshold of 0.75.

```
[Cxy,f] = mscohere(sig1,sig2,[],[],[],Fs);

thresh = 0.75;
[pks,locs] = findpeaks(Cxy,'MinPeakHeight',thresh);
MatchingFreqs = f(locs)
```

```
MatchingFreqs = 2×1

   35.1563
  164.0625
```

```
figure
plot(f,Cxy)
ax = gca;
grid
xlabel('Frequency (Hz)')
title('Coherence Estimate')
ax.XTick = MatchingFreqs;
ax.YTick = thresh;
axis([0 200 0 1])
```




You get the same values as before. You can find the frequency content common to two signals without studying the two signals separately.

## See Also

`findpeaks` | `mscohere` | `periodogram`

## Related Examples

- “Practical Introduction to Frequency-Domain Analysis” on page 24-254

# Detect Periodicity in a Signal with Missing Samples

Consider the weight of a person as recorded (in pounds) during the leap year 2012. The person did not record their weight every day. You would like to study the periodicity of the signal, even though some data points are missing.

Load the data and convert the measurements to kilograms. Missed readings are set to `NaN`. Determine how many points are missing.

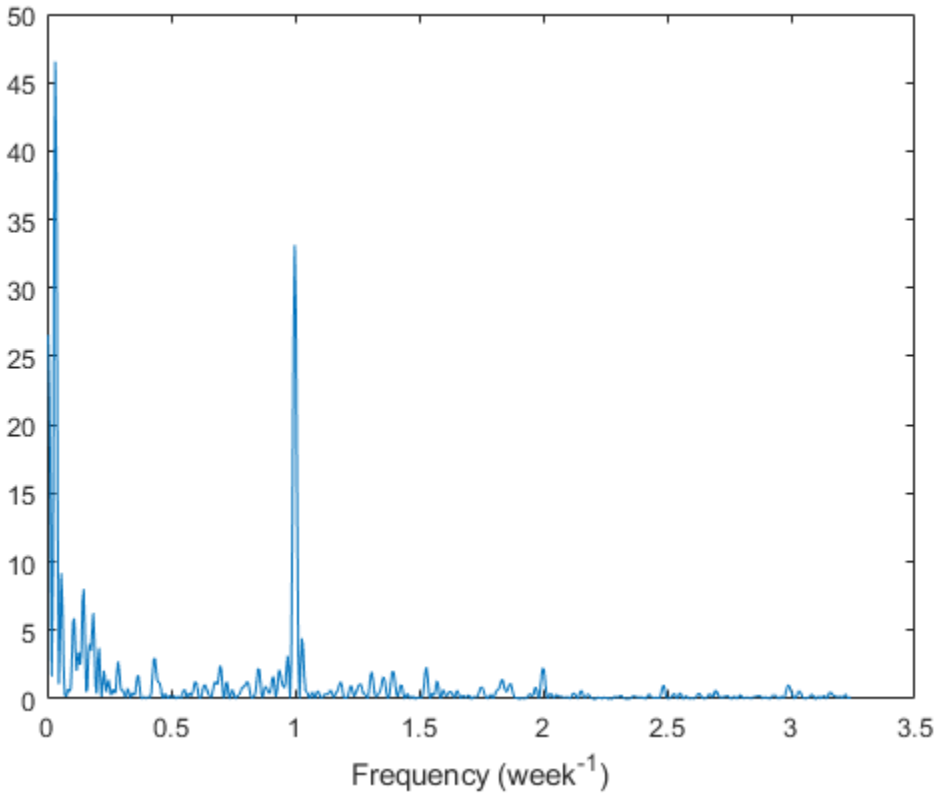
```
load('weight2012.dat')

wgt = weight2012(:,2)/2.20462;

fprintf('Missing %d samples of %d\n',sum(isnan(wgt)),length(wgt))
```

```
Missing 27 samples of 366
```

Determine if the signal is periodic by analyzing it in the frequency domain. The Lomb-Scargle algorithm is designed to handle data with missing samples or data that have been sampled irregularly.

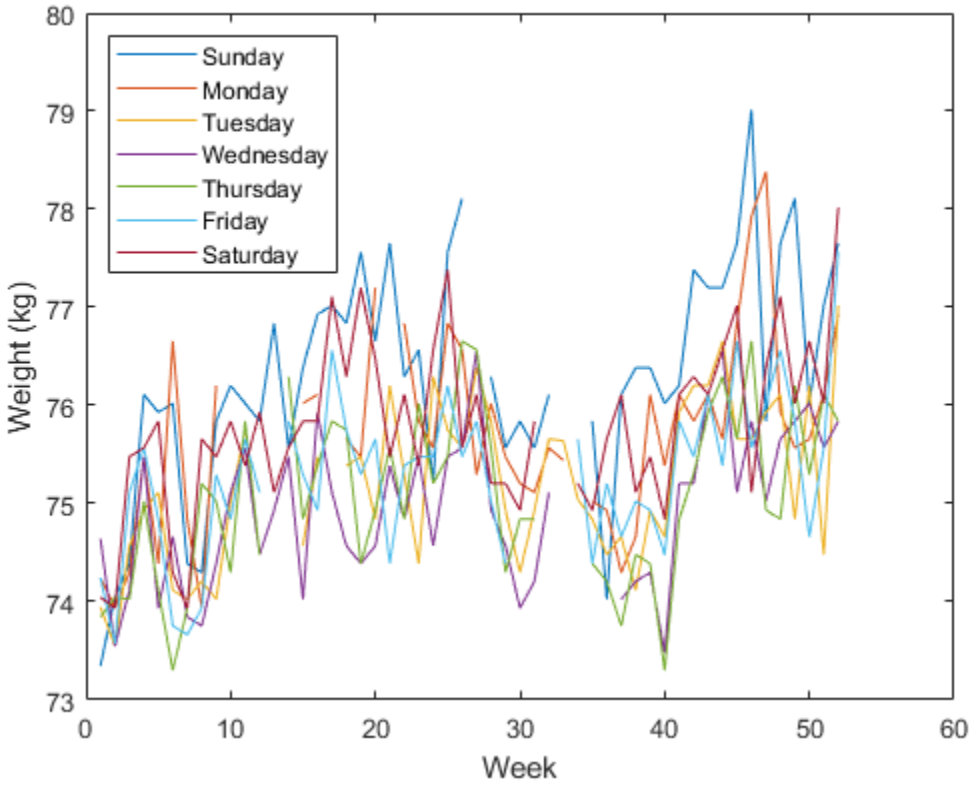
Find the cycle durations, measuring time in weeks.

```
[p,f] = plomb(wgt,7,'normalized');

plot(f,p)
xlabel('Frequency (week^{-1})')
```



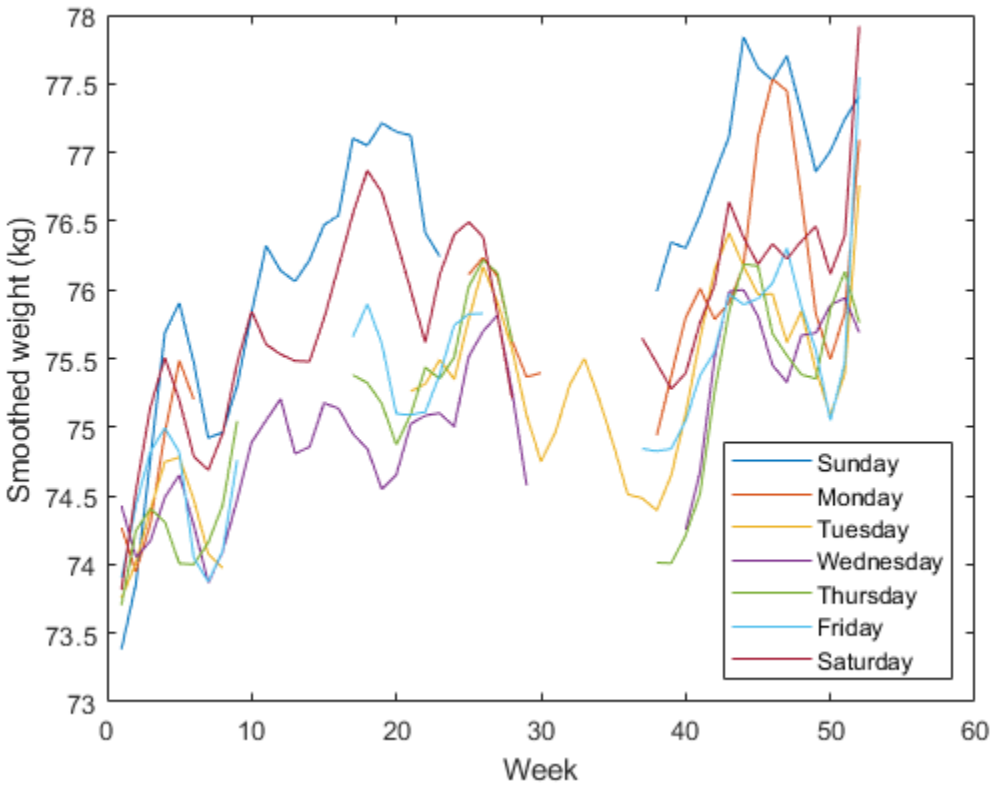
Notice how the person's weight oscillates weekly. Is there a noticeable pattern from week to week? Eliminate the last two days of the year to get 52 weeks. Reorder the measurements according to the day of the week.

```
wgd = reshape(wgt(1:7*52),[7 52])';

plot(wgd)
xlabel('Week')
ylabel('Weight (kg)')

q = legend(datestr(datenum(2012,1,1:7),'dddd'));
q.Location = 'NorthWest';
```




Smooth out the fluctuations using a filter that fits low-order polynomials to subsets of the data. Specifically, set it to fit cubic polynomials to sets of seven days.

```
wgs = sgolayfilt(wgd,3,7);

plot(wgs)
xlabel('Week')
ylabel('Smoothed weight (kg)')

q = legend(datestr(datenum(2012,1,1:7),'dddd'));
q.Location = 'SouthEast';
```




This person tends to eat more, and thus weigh more, during the weekend. Verify by computing the daily means. Exclude the missing values from the calculation.

```
for jk = 1:7
    wgm = find(~isnan(wgd(:,jk)));
    fprintf('%3s mean: %5.1f kg\n', ...
        datestr(datenum(2012,1,jk),'ddd')',mean(wgd(wgm,jk)))
end
```

```
Sun mean:  76.3 kg
Mon mean:  75.7 kg
Tue mean:  75.2 kg
Wed mean:  74.9 kg
Thu mean:  75.1 kg
Fri mean:  75.3 kg
Sat mean:  75.8 kg
```

## See Also

`datestr` | `plomb` | `sgolayfilt`

## Related Examples

- “Signal Smoothing” on page 24-10

# Echo Cancelation

A speech recording includes an echo caused by reflection off a wall. Use autocorrelation to filter it out.

In the recording, a person says the word MATLAB®. Load the data and the sample rate, $F_s$ = 7418 Hz.

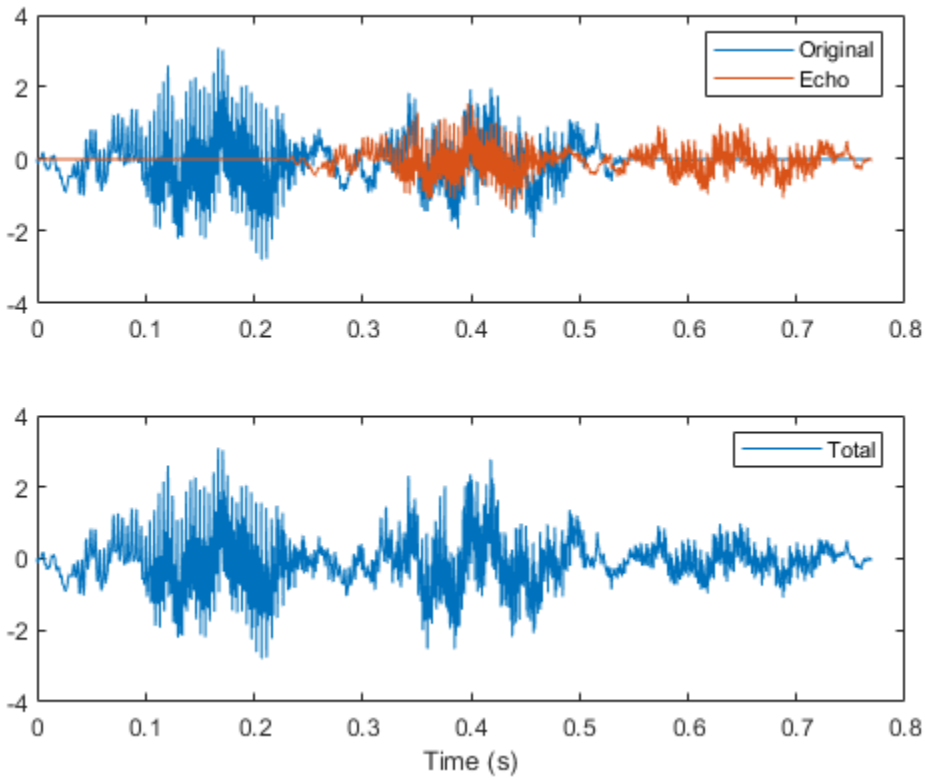
```
load mtlb

% To hear, type soundsc(mtlb,Fs)
```

Model the echo by adding to the recording a copy of the signal delayed by $\Delta$ samples and attenuated by a known factor $\alpha$: $y(n) = x(n) + \alpha x(n - \Delta)$. Specify a time lag of 0.23 s and an attenuation factor of 0.5.

```
timelag = 0.23;
delta = round(Fs*timelag);
alpha = 0.5;

orig = [mtlb;zeros(delta,1)];
echo = [zeros(delta,1);mtlb]*alpha;

mtEcho = orig + echo;
```

Plot the original, the echo, and the resulting signal.

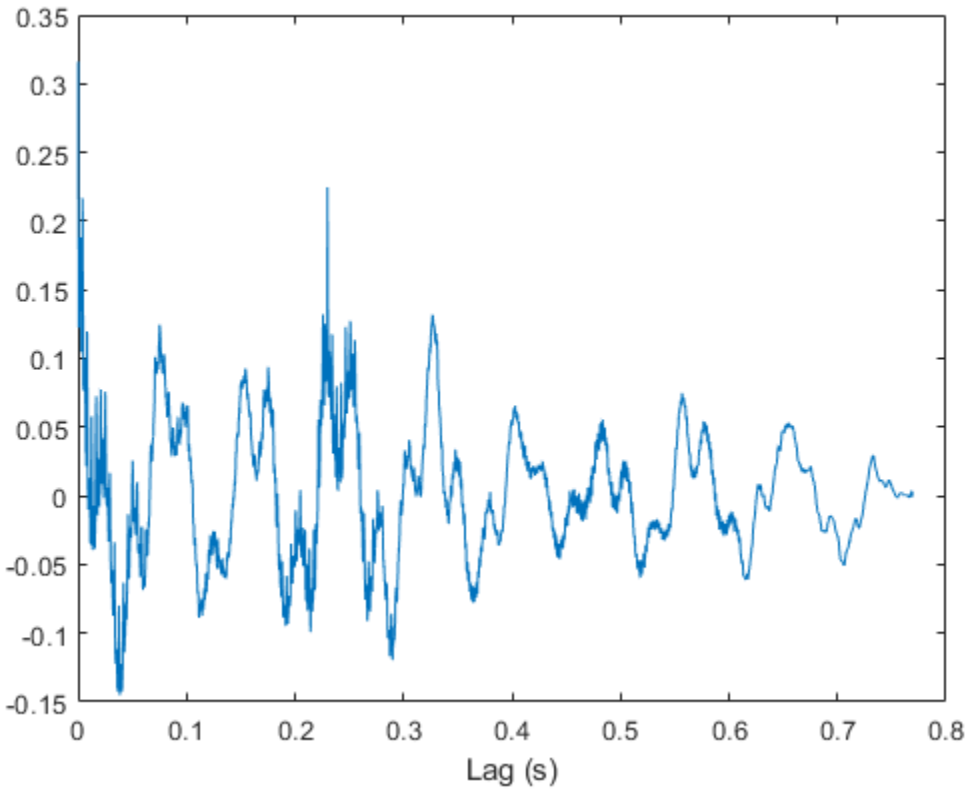
```
t = (0:length(mtEcho)-1)/Fs;

subplot(2,1,1)
plot(t,[orig echo])
legend('Original','Echo')

subplot(2,1,2)
plot(t,mtEcho)
legend('Total')
xlabel('Time (s)')
```




```
% To hear, type soundsc(mtEcho,Fs)
```

Compute an unbiased estimate of the signal autocorrelation. Select and plot the section that corresponds to lags greater than zero.

```
[Rmm,lags] = xcorr(mtEcho,'unbiased');

Rmm = Rmm(lags>0);
lags = lags(lags>0);

figure
plot(lags/Fs,Rmm)
xlabel('Lag (s)')
```

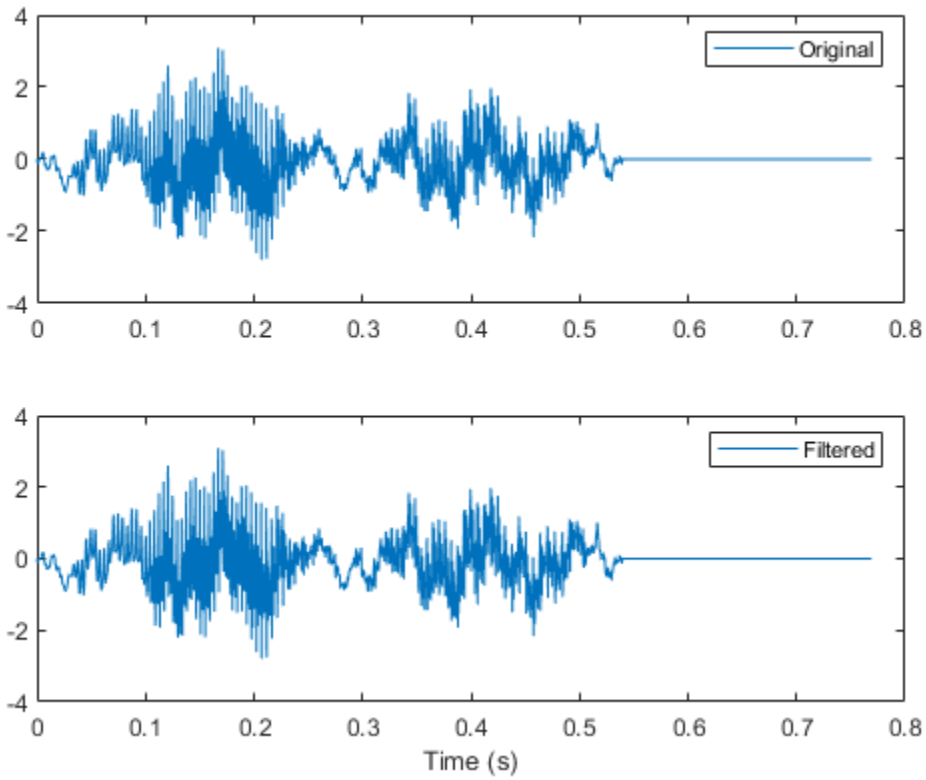



The autocorrelation has a sharp peak at the lag at which the echo arrives. Cancel the echo by filtering the signal through an IIR system whose output, $w$, obeys $w(n) + \alpha w(n - \Delta) = y(n)$.

```
[~,dl] = findpeaks(Rmm,lags,'MinPeakHeight',0.22);

mtNew = filter(1,[1 zeros(1,dl-1) alpha],mtEcho);
```

Plot the filtered signal and compare to the original.

```
subplot(2,1,1)
plot(t,orig)
legend('Original')

subplot(2,1,2)
plot(t,mtNew)
legend('Filtered')
xlabel('Time (s)')
```




```
% To hear, type soundsc(mtNew,Fs)
```

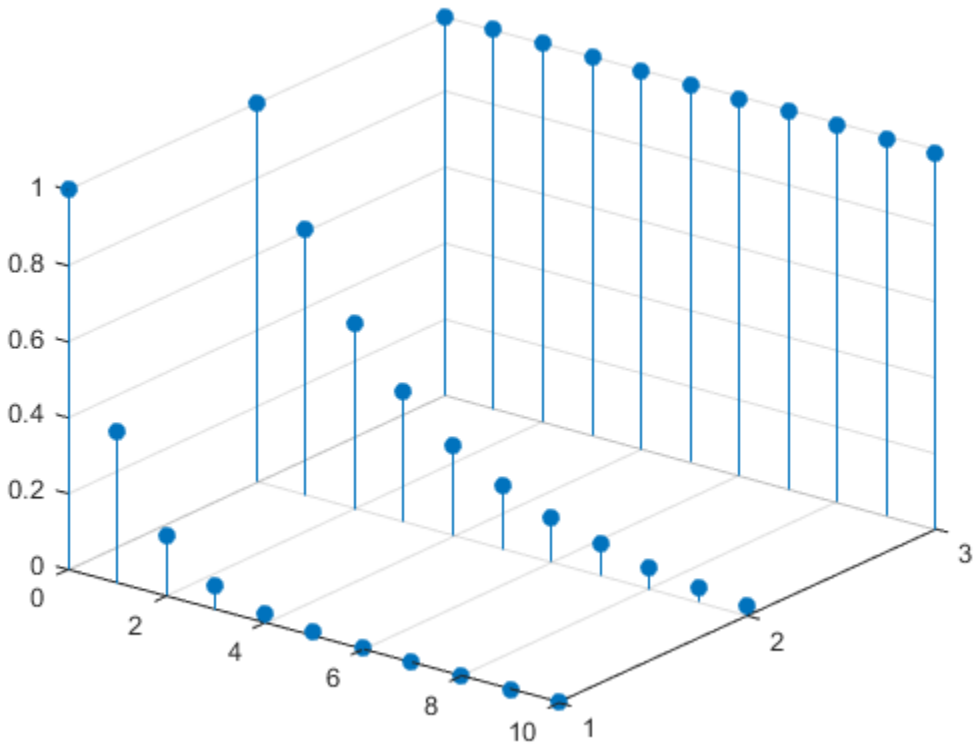
## See Also

**Functions**
`findpeaks` | `xcorr`

# Cross-Correlation with Multichannel Input

Generate three 11-sample exponential sequences given by $0.4^n$, $0.7^n$, and $0.999^n$, with $n \geq 0$. Use `stem3` to plot the sequences side by side.

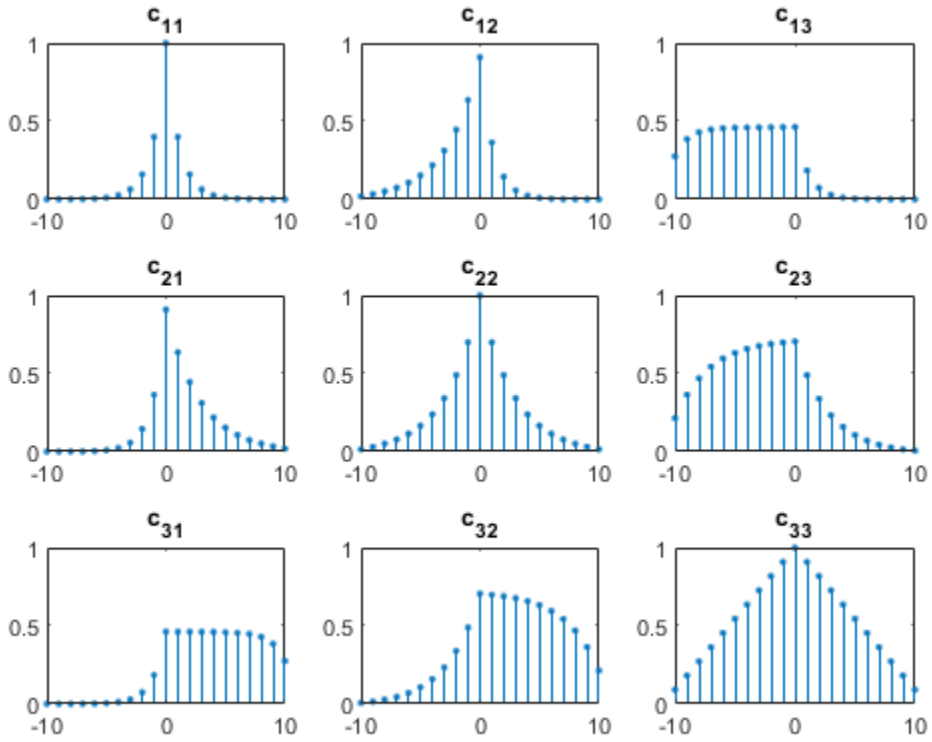
```
N = 11;
n = (0:N-1)';

a = 0.4;
b = 0.7;
c = 0.999;

xabc = [a.^n b.^n c.^n];

stem3(n,1:3,xabc','filled')
ax = gca;
ax.YTick = 1:3;
view(37.5,30)
```




Compute the autocorrelations and mutual cross-correlations of the sequences. Output the lags so you do not have to keep track of them. Normalize the result so the autocorrelations have unit value at zero lag.

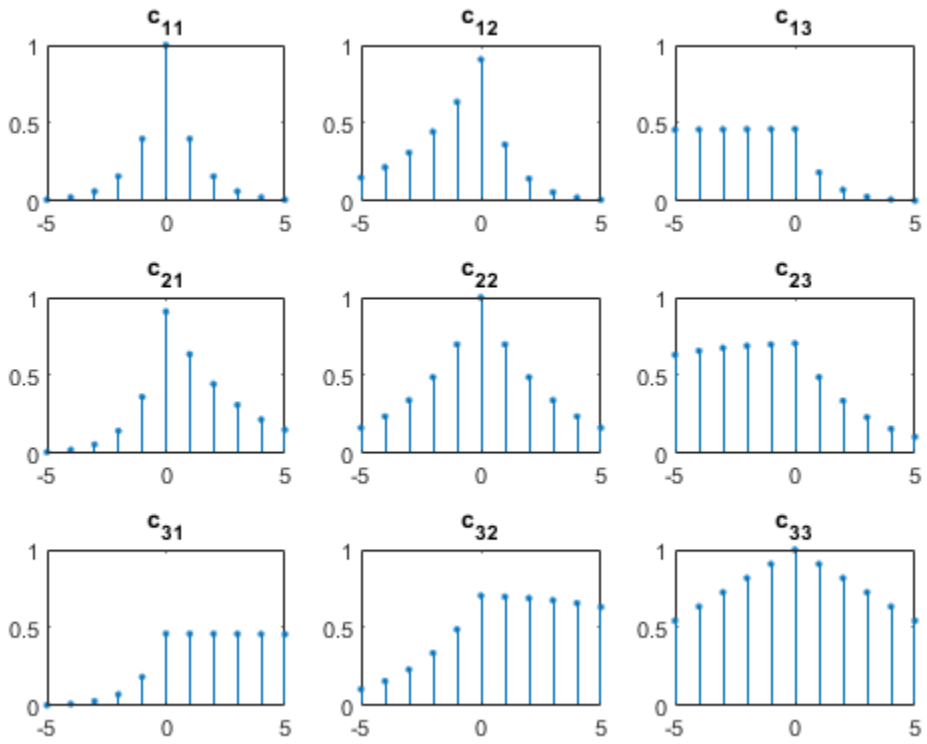
```
[cr,lgs] = xcorr(xabc,'coeff');

for row = 1:3
```

```
    for col = 1:3
        nm = 3*(row-1)+col;
        subplot(3,3,nm)
        stem(lgs,cr(:,nm),'.')
        title(sprintf('c_{%d%d}',row,col))
        ylim([0 1])
    end
end
```



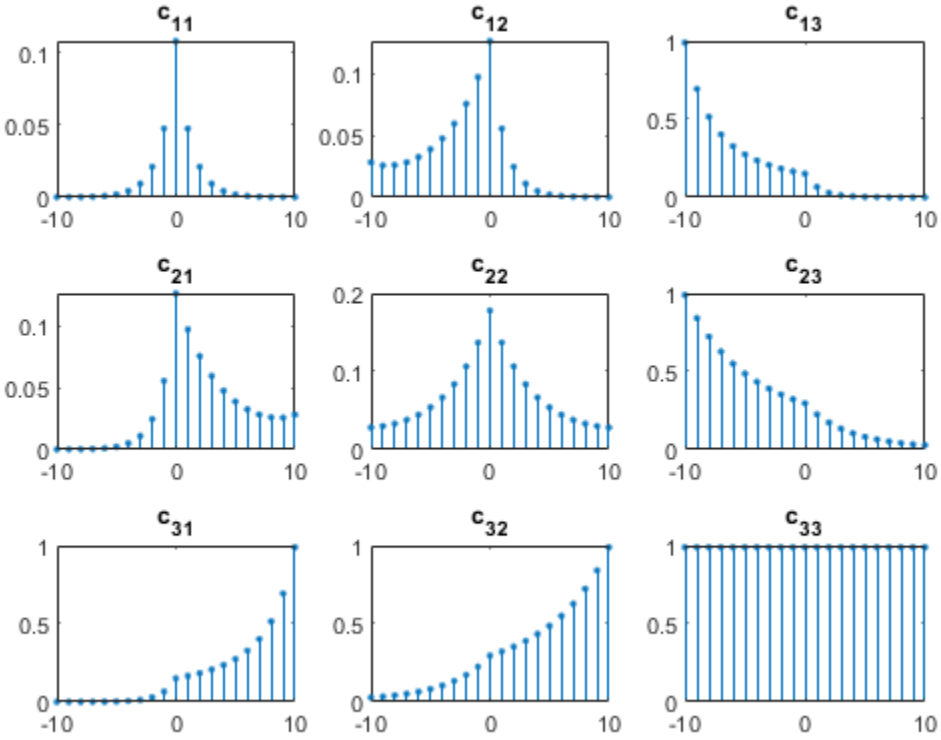

Restrict the calculation to lags between −5 and 5.

```
[cr,lgs] = xcorr(xabc,5,'coeff');

for row = 1:3
    for col = 1:3
        nm = 3*(row-1)+col;
        subplot(3,3,nm)
        stem(lgs,cr(:,nm),'.')
        title(sprintf('c_{%d%d}',row,col))
        ylim([0 1])
    end
end
```




Compute unbiased estimates of the autocorrelations and mutual cross-correlations. By default, the lags run between $-(N-1)$ and $N-1$.

```
cu = xcorr(xabc,'unbiased');

for row = 1:3
    for col = 1:3
        nm = 3*(row-1)+col;
        subplot(3,3,nm)
        stem(-(N-1):(N-1),cu(:,nm),'.')
        title(sprintf('c_{%d%d}',row,col))
    end
end
```




## See Also

**Functions**
`xcorr`

# Autocorrelation Function of Exponential Sequence

Compute the autocorrelation function of a 28-sample exponential sequence, $x = 0.95^n$ for $n \geq 0$.

```
a = 0.95;

N = 28;
n = 0:N-1;
lags = -(N-1):(N-1);

x = a.^n;
c = xcorr(x);
```

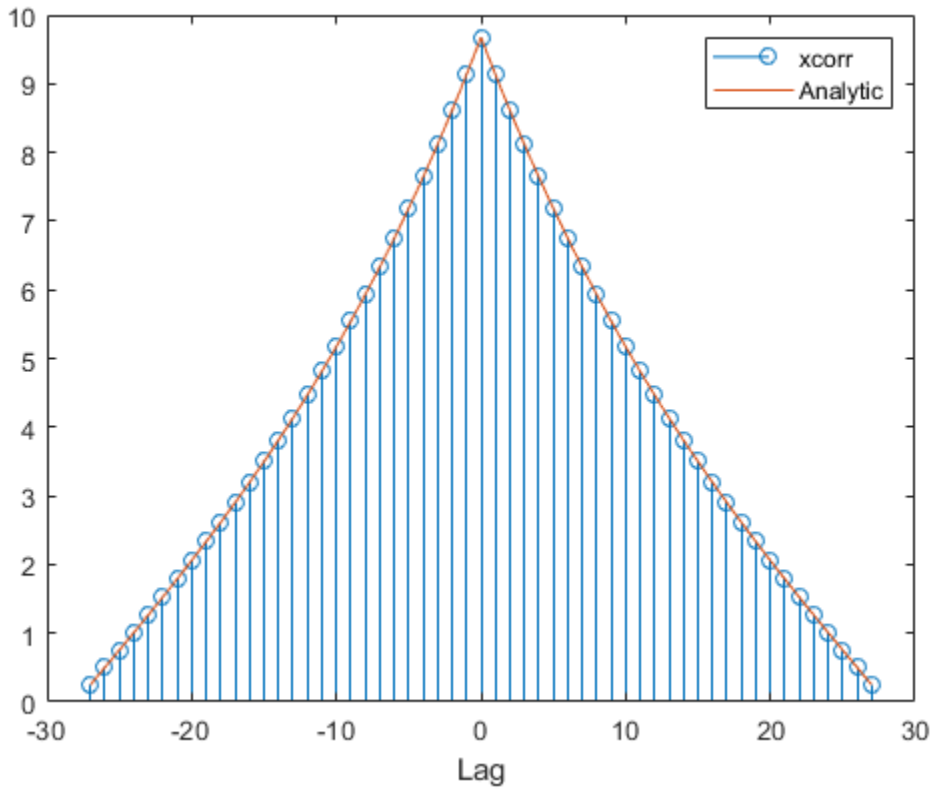
Determine $c$ analytically to check the correctness of the result. Use a larger sample rate to simulate a continuous situation. The autocorrelation function of the sequence $x(n) = a^n$ for $n \geq 0$, with $|a| < 1$, is

$$c(n) = \frac{1 - a^{2(N - |n|)}}{1 - a^2} \times a^{|n|}.$$

```
fs = 10;
nn = -(N-1):1/fs:(N-1);

dd = (1-a.^(2*(N-abs(nn))))/(1-a^2).*a.^abs(nn);
```

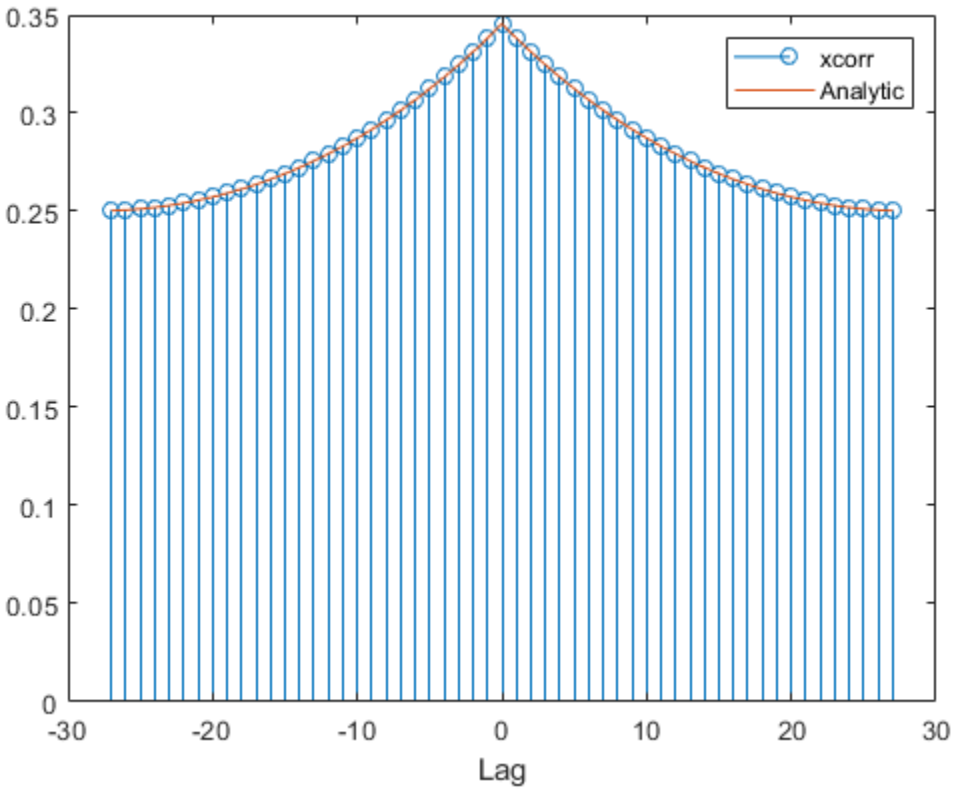
Plot the sequences on the same figure.

```
stem(lags,c);
hold on
plot(nn,dd)
xlabel('Lag')
legend('xcorr','Analytic')
hold off
```



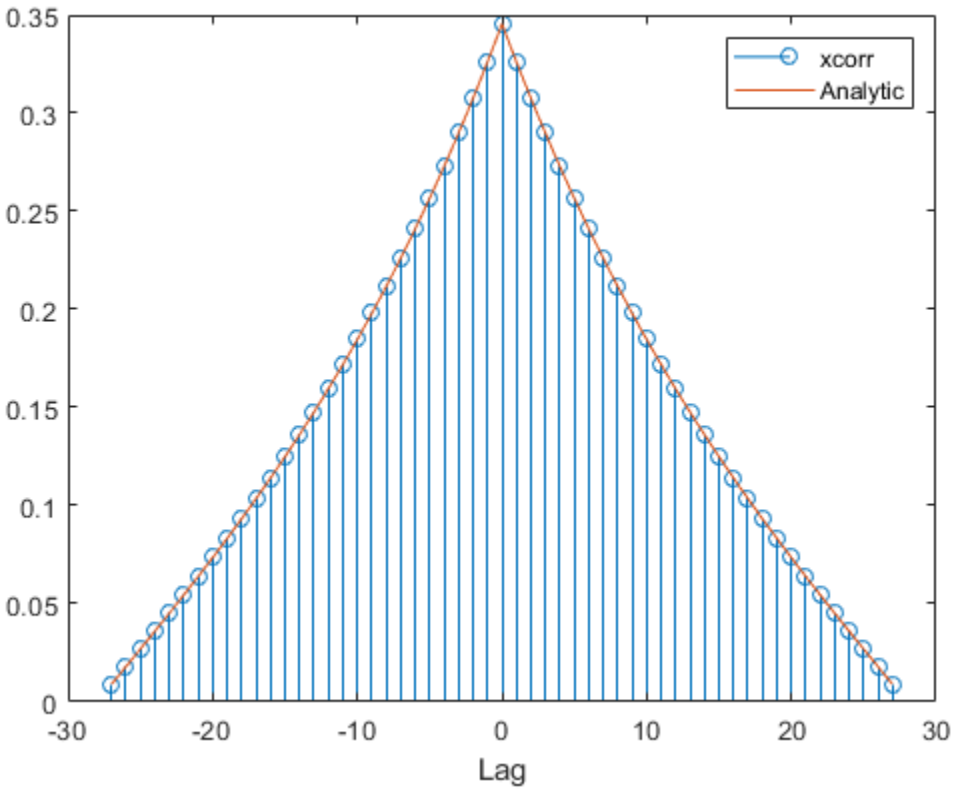

Repeat the calculation, but now find an *unbiased* estimate of the autocorrelation. Verify that the unbiased estimate is given by $c_u(n) = c(n)/(N - |n|)$.

```
cu = xcorr(x,'unbiased');

du = dd./(N-abs(nn));

stem(lags,cu);
hold on
plot(nn,du)
xlabel('Lag')
legend('xcorr','Analytic')
hold off
```



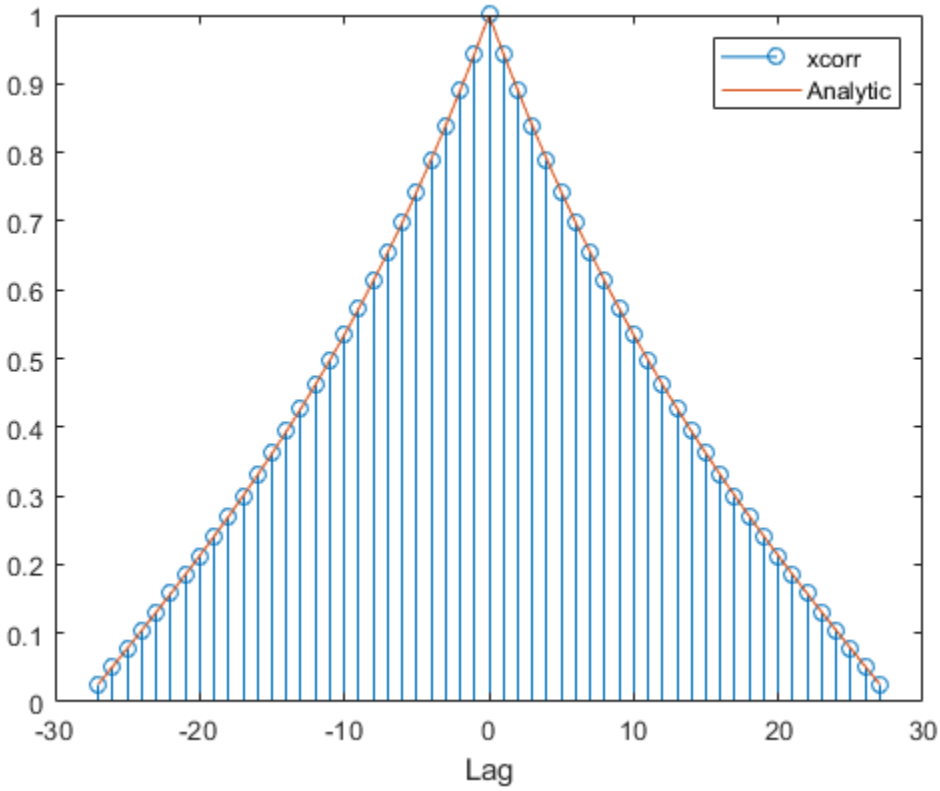

Repeat the calculation, but now find a *biased* estimate of the autocorrelation. Verify that the biased estimate is given by $c_b(n) = c(n)/N$.

```
cb = xcorr(x,'biased');

db = dd/N;

stem(lags,cb);
hold on
plot(nn,db)
xlabel('Lag')
legend('xcorr','Analytic')
hold off
```




Find an estimate of the autocorrelation whose value at zero lag is unity.

```
cz = xcorr(x,'coeff');

dz = dd/max(dd);

stem(lags,cz);
hold on
plot(nn,dz)
xlabel('Lag')
legend('xcorr','Analytic')
hold off
```



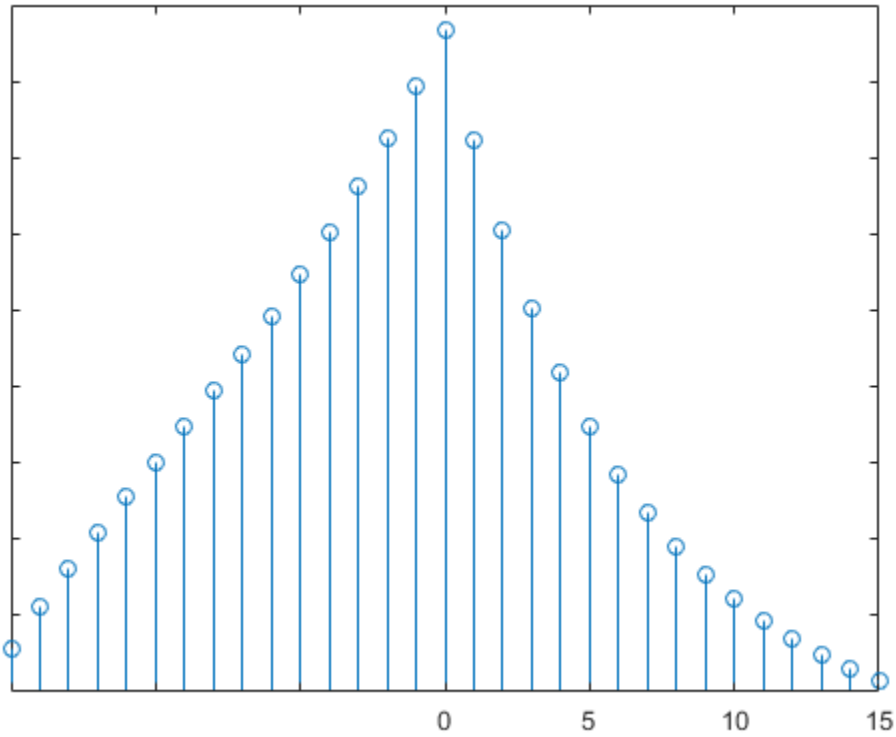

## See Also

**Functions**
`xcorr`

# Cross-Correlation of Two Exponential Sequences

Compute and plot the cross-correlation of two 16-sample exponential sequences, $x_a = 0.84^n$ and $x_b = 0.92^n$, with $n \geq 0$.

```
N = 16;
n = 0:N-1;

a = 0.84;
b = 0.92;

xa = a.^n;
xb = b.^n;

r = xcorr(xa,xb);

stem(-(N-1):(N-1),r)
```



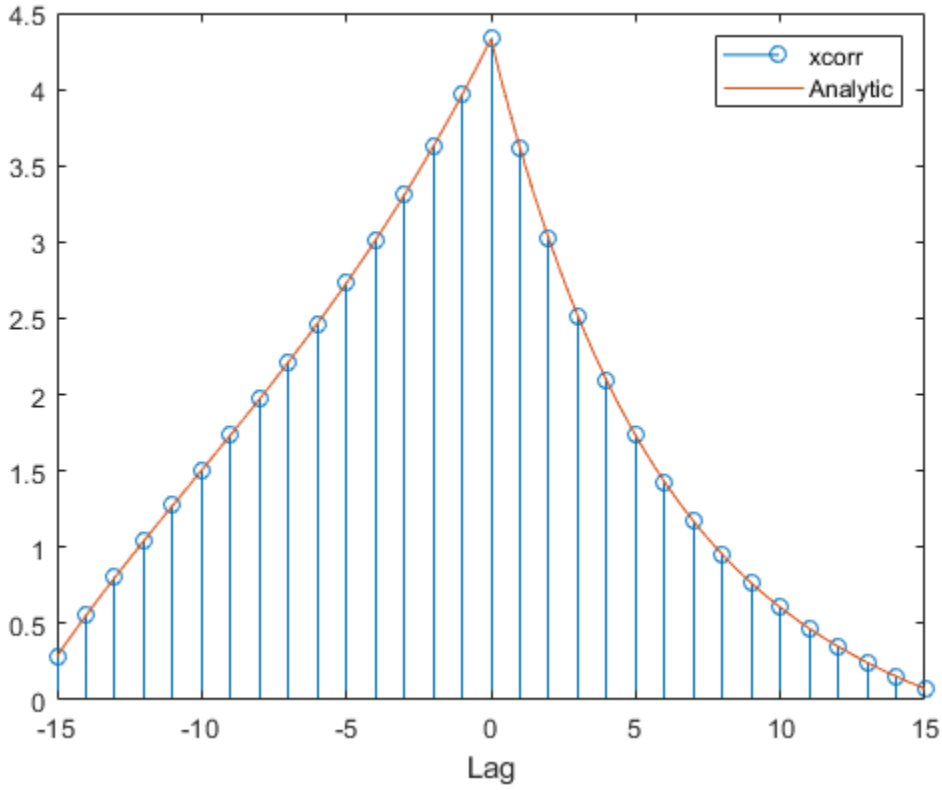

Determine $c$ analytically to check the correctness of the result. Use a larger sample rate to simulate a continuous situation. The cross-correlation function of the sequences $x_a(n) = a^n$ and $x_b(n) = b^n$ for $n \geq 0$, with $0 < a, b < 1$, is

$$c_{ab}(n) = \frac{1-(ab)^{N-|n|}}{1-ab} \times \begin{cases} a^n, & n > 0, \\ 1, & n = 0, \\ b^{-n}, & n < 0. \end{cases}$$

```
fs = 10;
nn = -(N-1):1/fs:(N-1);

cn = (1 - (a*b).^(N-abs(nn)))/(1 - a*b) .* ...
     (a.^nn.*(nn>0) + (nn==0) + b.^-(nn).*(nn<0));
```

Plot the sequences on the same figure.

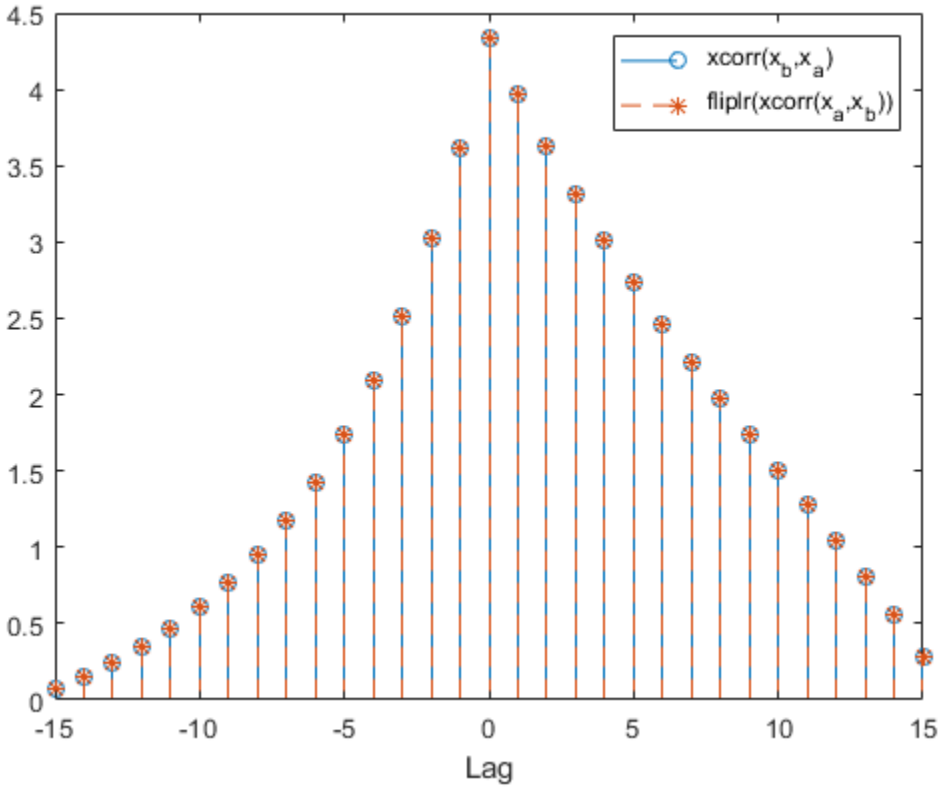
```
hold on
plot(nn,cn)

xlabel('Lag')
legend('xcorr','Analytic')
```




Verify that switching the order of the operands reverses the sequence.

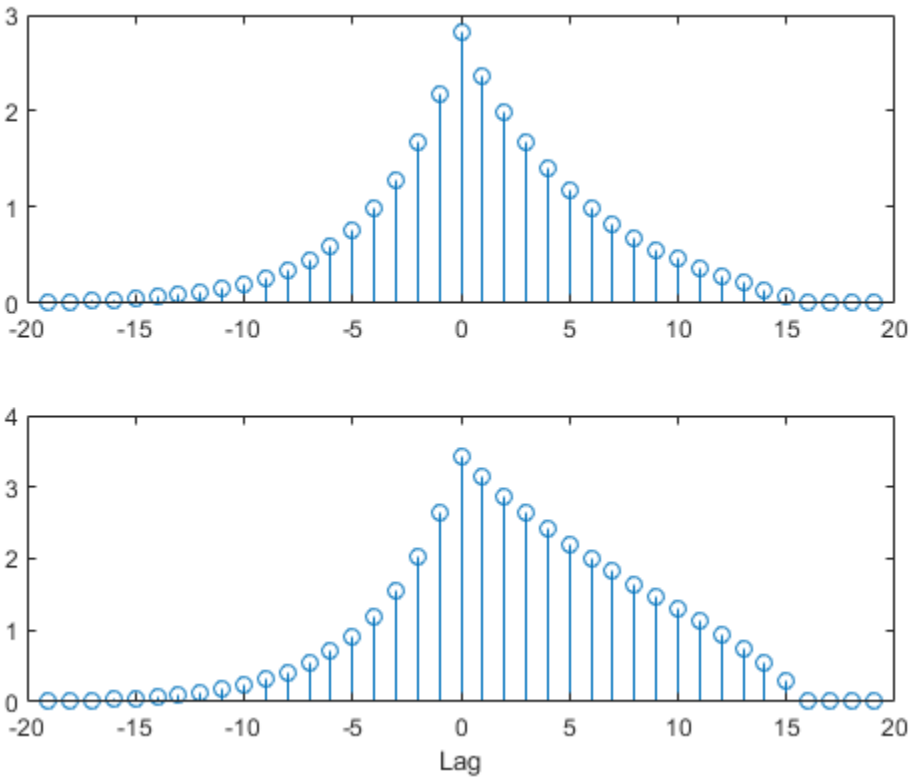
```
figure

stem(-(N-1):(N-1),xcorr(xb,xa))

hold on
stem(-(N-1):(N-1),fliplr(r),'--*')
```

```
xlabel('Lag')
legend('xcorr(x_b,x_a)','fliplr(xcorr(x_a,x_b))')
```




Generate the 20-sample exponential sequence $x_c = 0.77^n$. Compute and plot its cross-correlations with $x_a$ and $x_b$. Output the lags to make the plotting easier. `xcorr` appends zeros at the end of the shorter sequence to match the length of the longer one.

```
xc = 0.77.^(0:20-1);

[xca,la] = xcorr(xa,xc);
[xcb,lb] = xcorr(xb,xc);

figure

subplot(2,1,1)
stem(la,xca)
subplot(2,1,2)
stem(lb,xcb)
xlabel('Lag')
```




## See Also

**Functions**
`xcorr`

# Featured Examples

# Signal Generation and Visualization

This example shows how to generate widely used periodic and aperiodic waveforms, swept-frequency sinusoids, and pulse trains using functions available in Signal Processing Toolbox™.

### Periodic Waveforms

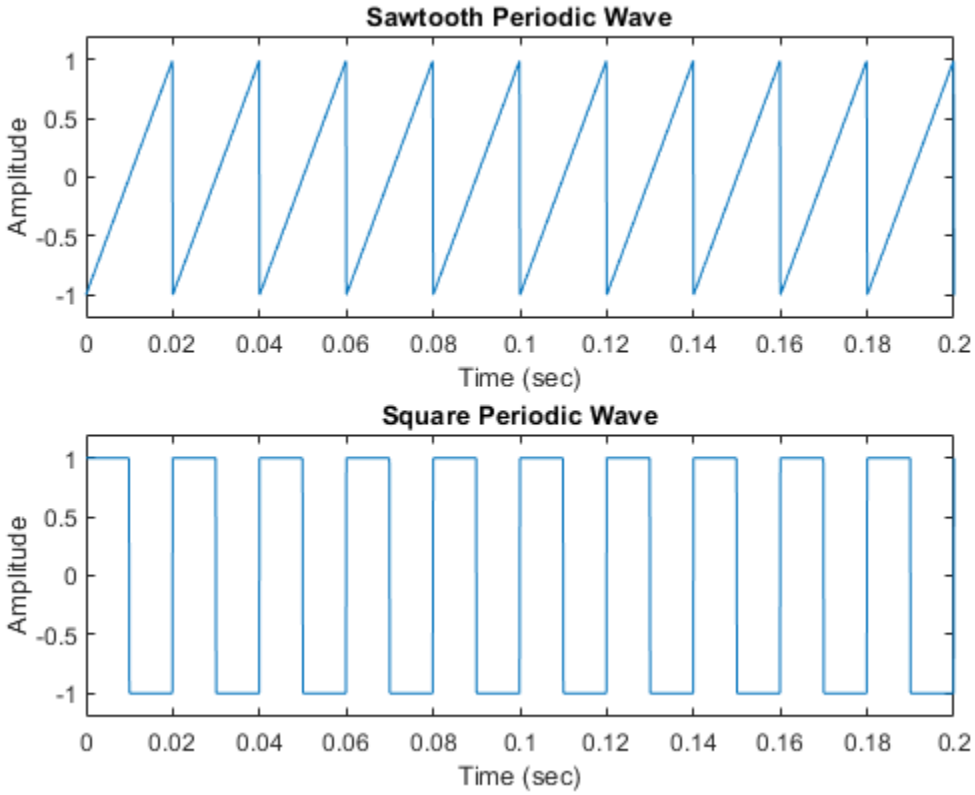
In addition to the `sin` and `cos` functions in MATLAB®, Signal Processing Toolbox™ offers other functions, such as `sawtooth` and `square`, that produce periodic signals.

The `sawtooth` function generates a sawtooth wave with peaks at ±1 and a period of $2\pi$. An optional width parameter specifies a fractional multiple of $2\pi$ at which the signal's maximum occurs.

The `square` function generates a square wave with a period of $2\pi$. An optional parameter specifies duty cycle, the percent of the period for which the signal is positive.

Generate 1.5 seconds of a 50 Hz sawtooth wave with a sample rate of 10 kHz. Repeat the computation for a square wave.

```
fs = 10000;
t = 0:1/fs:1.5;
x1 = sawtooth(2*pi*50*t);
x2 = square(2*pi*50*t);

subplot(2,1,1)
plot(t,x1)
axis([0 0.2 -1.2 1.2])
xlabel('Time (sec)')
ylabel('Amplitude')
title('Sawtooth Periodic Wave')

subplot(2,1,2)
plot(t,x2)
axis([0 0.2 -1.2 1.2])
xlabel('Time (sec)')
ylabel('Amplitude')
title('Square Periodic Wave')
```




### Aperiodic Waveforms

To generate triangular, rectangular and Gaussian pulses, the toolbox offers the `tripuls`, `rectpuls`, and `gauspuls` functions.

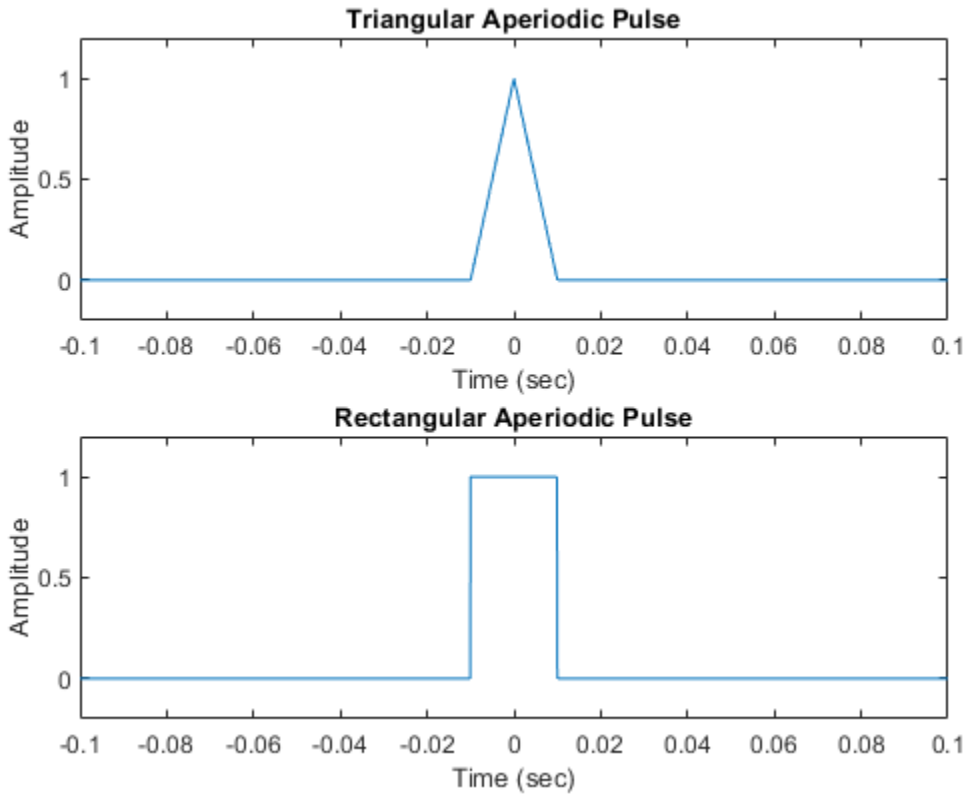
The `tripuls` function generates a sampled aperiodic, unit-height triangular pulse centered about $t = 0$ and with a default width of 1.

The `rectpuls` function generates a sampled aperiodic, unit-height rectangular pulse centered about $t = 0$ and with a default width of 1. The interval of nonzero amplitude is defined to be open on the right: `rectpuls(-0.5) = 1` while `rectpuls(0.5) = 0`.

Generate 2 seconds of a triangular pulse with a sample rate of 10 kHz and a width of 20 ms. Repeat the computation for a rectangular pulse.

```
fs = 10000;
t = -1:1/fs:1;
x1 = tripuls(t,20e-3);
x2 = rectpuls(t,20e-3);

figure
subplot(2,1,1)
plot(t,x1)
axis([-0.1 0.1 -0.2 1.2])
xlabel('Time (sec)')
ylabel('Amplitude')
title('Triangular Aperiodic Pulse')
```

```
subplot(2,1,2)
plot(t,x2)
axis([-0.1 0.1 -0.2 1.2])
xlabel('Time (sec)')
ylabel('Amplitude')
title('Rectangular Aperiodic Pulse')
```




The `gauspuls` function generates a Gaussian-modulated sinusoidal pulse with a specified time, center frequency, and fractional bandwidth.

The `sinc` function computes the mathematical sinc function for an input vector or matrix. The sinc function is the continuous inverse Fourier transform of a rectangular pulse of width $2\pi$ and unit height.

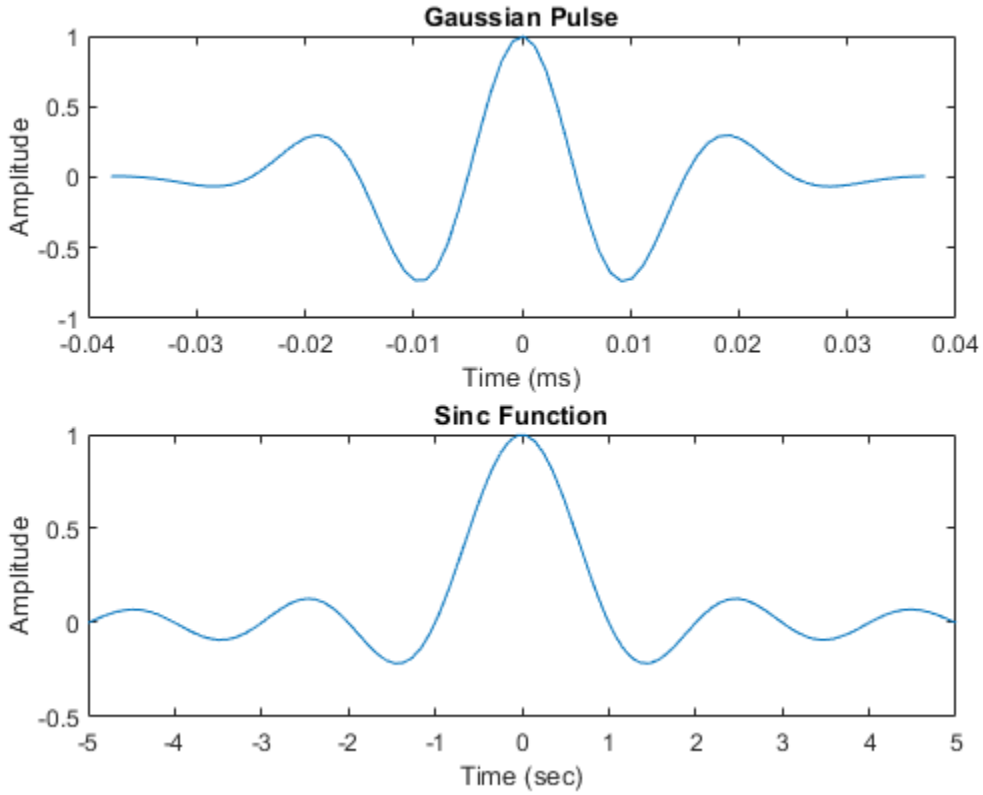
Generate a 50 kHz Gaussian RF pulse with 60% bandwidth, sampled at a rate of 1 MHz. Truncate the pulse where the envelope falls 40 dB below the peak.

```
tc = gauspuls('cutoff',50e3,0.6,[],-40);
t1 = -tc : 1e-6 : tc;
y1 = gauspuls(t1,50e3,0.6);
```

Generate the sinc function for a linearly spaced vector:

```
t2 = linspace(-5,5);
y2 = sinc(t2);
```

```
figure
subplot(2,1,1)
plot(t1*1e3,y1)
xlabel('Time (ms)')
ylabel('Amplitude')
title('Gaussian Pulse')

subplot(2,1,2)
plot(t2,y2)
xlabel('Time (sec)')
ylabel('Amplitude')
title('Sinc Function')
```




### Swept-Frequency Waveforms

The toolbox also provides functions to generate swept-frequency waveforms such as the `chirp` function. Two optional parameters specify alternative sweep methods and initial phase in degrees. Below are several examples of using the `chirp` function to generate linear or quadratic, convex, and concave quadratic chirps.

Generate a linear chirp.

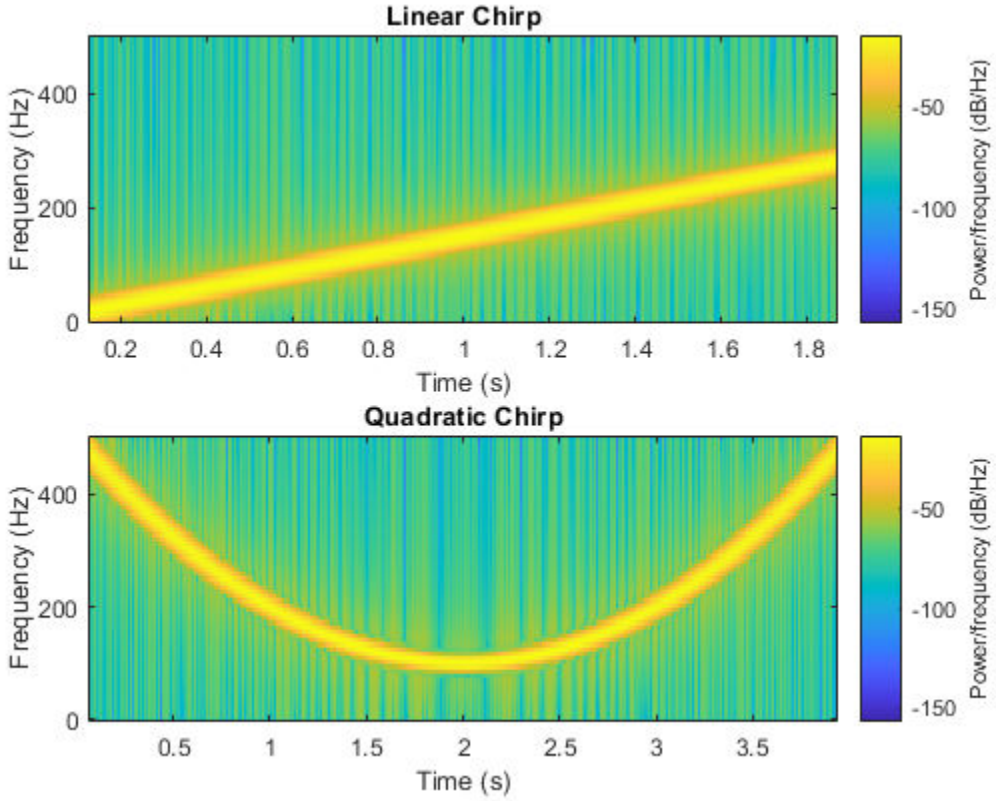
```
t = 0:0.001:2;                  % 2 secs @ 1kHz sample rate
ylin = chirp(t,0,1,150);        % Start @ DC, cross 150Hz at t=1sec
```

Generate a quadratic chirp.

```
t = -2:0.001:2;                % +/-2 secs @ 1kHz sample rate
yq = chirp(t,100,1,200,'q');   % Start @ 100Hz, cross 200Hz at t=1sec
```

Compute and display the spectrograms of the chirps.

```
figure
subplot(2,1,1)
spectrogram(ylin,256,250,256,1E3,'yaxis')
title('Linear Chirp')

subplot(2,1,2)
spectrogram(yq,128,120,128,1E3,'yaxis')
title('Quadratic Chirp')
```




Generate a convex quadratic chirp.

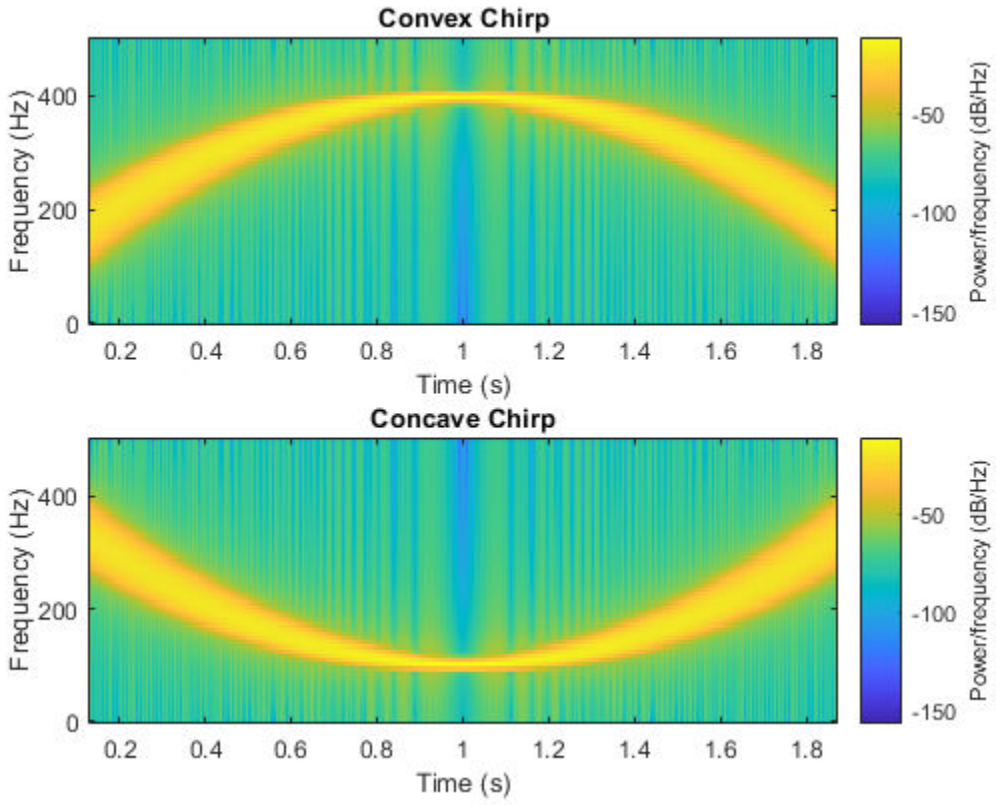
```
t = -1:0.001:1;          % +/-1 second @ 1kHz sample rate
fo = 100;
f1 = 400;                % Start at 100Hz, go up to 400Hz
ycx = chirp(t,fo,1,f1,'q',[],'convex');
```

Generate a concave quadratic chirp.

```
t = -1:0.001:1;          % +/-1 second @ 1kHz sample rate
fo = 400;
f1 = 100;                % Start at 400Hz, go down to 100Hz
ycv = chirp(t,fo,1,f1,'q',[],'concave');
```

Compute and display the spectrograms of the chirps.

```
figure
subplot(2,1,1)
spectrogram(ycx,256,255,128,1000,'yaxis')
title('Convex Chirp')

subplot(2,1,2)
spectrogram(ycv,256,255,128,1000,'yaxis')
title('Concave Chirp')
```




Another function generator is the `vco` (voltage-controlled oscillator), which generates a signal oscillating at a frequency determined by the input vector. The input vector can be a triangle, a rectangle, or a sinusoid, among other possibilities.

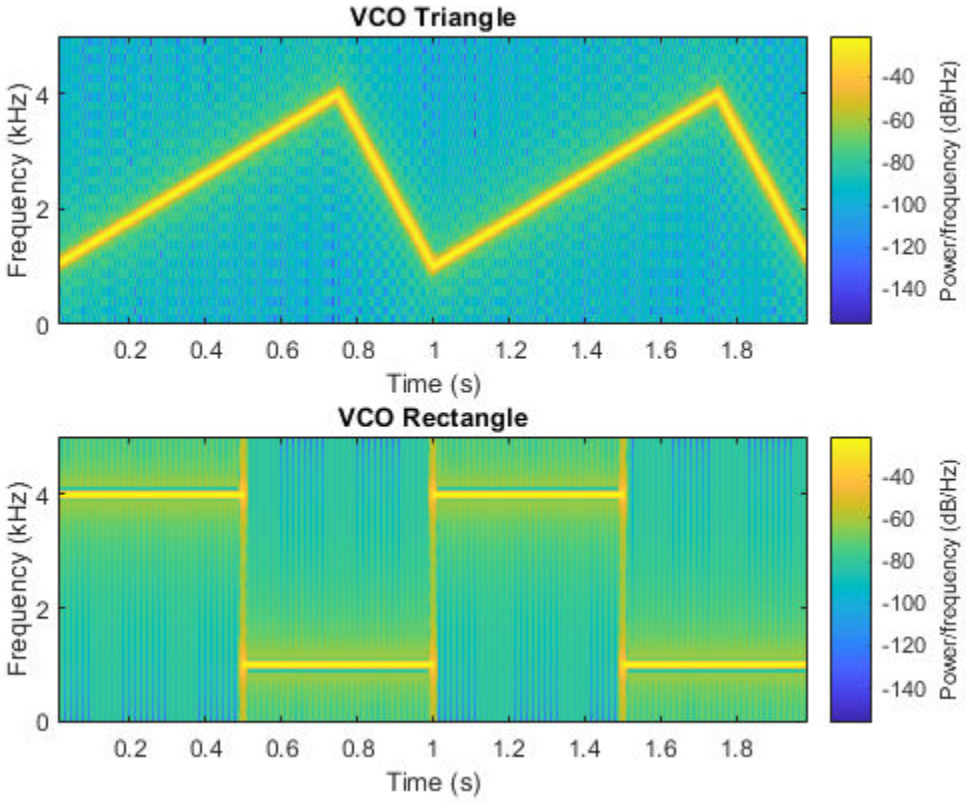
Generate 2 seconds of a signal sampled at 10 kHz whose instantaneous frequency is a triangle. Repeat the computation for a rectangle.

```
fs = 10000;
t = 0:1/fs:2;
x1 = vco(sawtooth(2*pi*t,0.75),[0.1 0.4]*fs,fs);
x2 = vco(square(2*pi*t),[0.1 0.4]*fs,fs);
```

Plot the spectrograms of the generated signals.

```
figure
subplot(2,1,1)
spectrogram(x1,kaiser(256,5),220,512,fs,'yaxis')
title('VCO Triangle')
```

```
subplot(2,1,2)
spectrogram(x2,256,255,256,fs,'yaxis')
title('VCO Rectangle')
```




### Pulse Trains

To generate pulse trains, you can use the `pulstran` function.

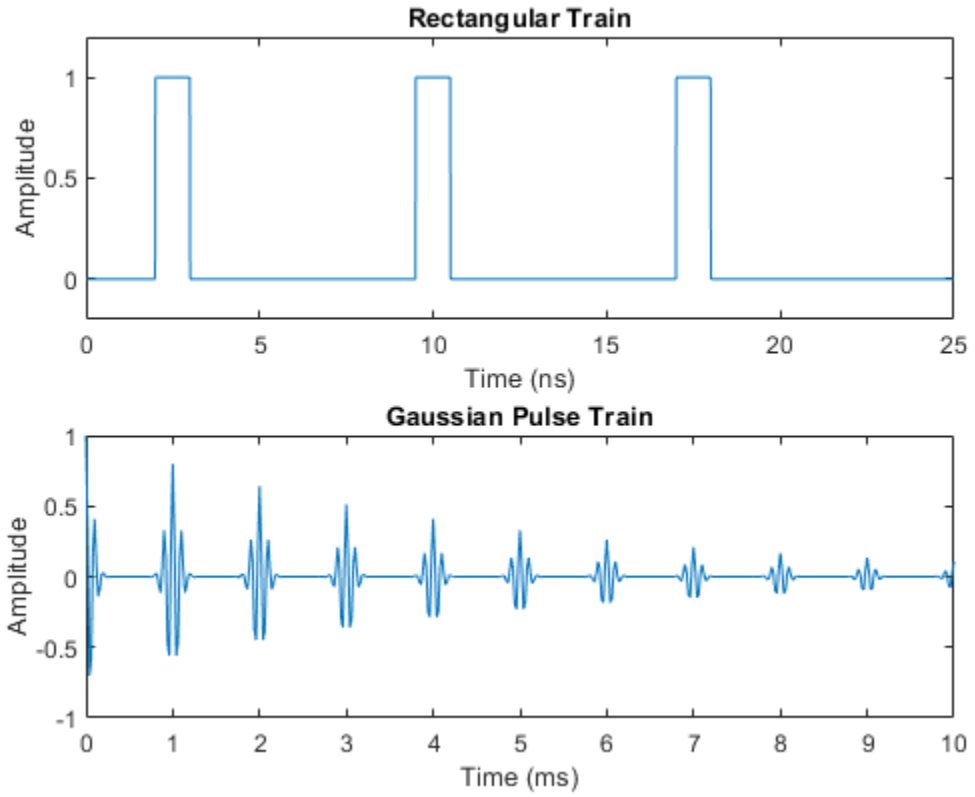
Construct a train of 2 GHz rectangular pulses sampled at a rate of 100 GHz at a spacing of 7.5 ns.

```
fs = 100E9;                     % sample freq
D = [2.5 10 17.5]' * 1e-9;      % pulse delay times
t = 0 : 1/fs : 2500/fs;         % signal evaluation time
w = 1e-9;                       % width of each pulse
yp = pulstran(t,D,@rectpuls,w);
```

Generate a periodic Gaussian pulse signal at 10 kHz with 50% bandwidth. The pulse repetition frequency is 1 kHz, the sample rate is 50 kHz, and the pulse train length is 10 milliseconds. The repetition amplitude should attenuate by 0.8 each time. Uses a function handle to specify the generator function.

```
T = 0 : 1/50e3 : 10e-3;
D = [0 : 1/1e3 : 10e-3 ; 0.8.^(0:10)]';
Y = pulstran(T,D,@gauspuls,10E3,.5);

figure
subplot(2,1,1)
```

```
plot(t*1e9,yp);
axis([0 25 -0.2 1.2])
xlabel('Time (ns)')
ylabel('Amplitude')
title('Rectangular Train')

subplot(2,1,2)
plot(T*1e3,Y)
xlabel('Time (ms)')
ylabel('Amplitude')
title('Gaussian Pulse Train')
```




## See Also

chirp | gauspuls | pulstran | rectpuls | sawtooth | sin | sinc | square | tripuls | vco

# Signal Smoothing

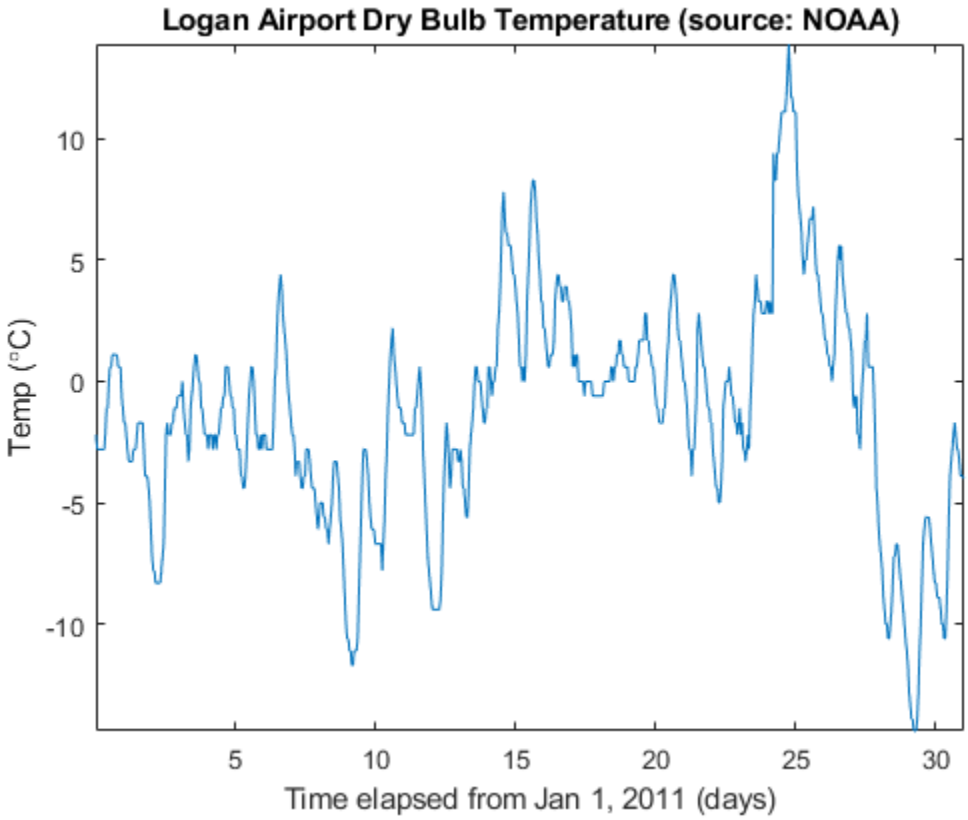
This example shows how to use moving average filters and resampling to isolate the effect of periodic components of the time of day on hourly temperature readings, as well as remove unwanted line noise from an open-loop voltage measurement. The example also shows how to smooth the levels of a clock signal while preserving the edges by using a median filter. The example also shows how to use a Hampel filter to remove large outliers.

### Motivation

Smoothing is how we discover important patterns in our data while leaving out things that are unimportant (i.e. noise). We use filtering to perform this smoothing. The goal of smoothing is to produce slow changes in value so that it's easier to see trends in our data.

Sometimes when you examine input data you may wish to smooth the data in order to see a trend in the signal. In our example we have a set of temperature readings in Celsius taken every hour at Logan Airport for the entire month of January, 2011.

```
load bostemp
days = (1:31*24)/24;
plot(days, tempC)
axis tight
ylabel('Temp (\circC)')
xlabel('Time elapsed from Jan 1, 2011 (days)')
title('Logan Airport Dry Bulb Temperature (source: NOAA)')
```



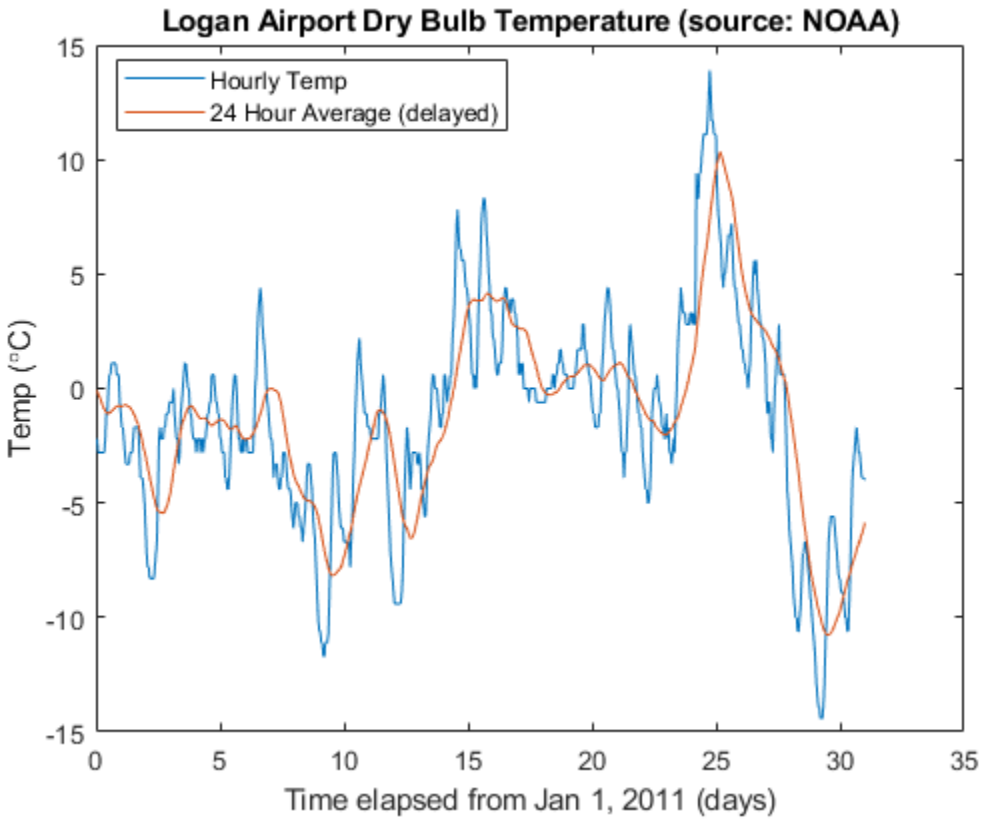
Note that we can visually see the effect that the time of day has upon the temperature readings. If you are only interested in the daily temperature variation over the month, the hourly fluctuations only contribute noise, which can make the daily variations difficult to discern. To remove the effect of the time of day, we would now like to smooth our data by using a moving average filter.

### A Moving Average Filter

In its simplest form, a moving average filter of length N takes the average of every N consecutive samples of the waveform.

To apply a moving average filter to each data point, we construct our coefficients of our filter so that each point is equally weighted and contributes 1/24 to the total average. This gives us the average temperature over each 24 hour period.

```
hoursPerDay = 24;
coeff24hMA = ones(1, hoursPerDay)/hoursPerDay;

avg24hTempC = filter(coeff24hMA, 1, tempC);
plot(days,[tempC avg24hTempC])
legend('Hourly Temp','24 Hour Average (delayed)','location','best')
ylabel('Temp (\circC)')
xlabel('Time elapsed from Jan 1, 2011 (days)')
title('Logan Airport Dry Bulb Temperature (source: NOAA)')
```

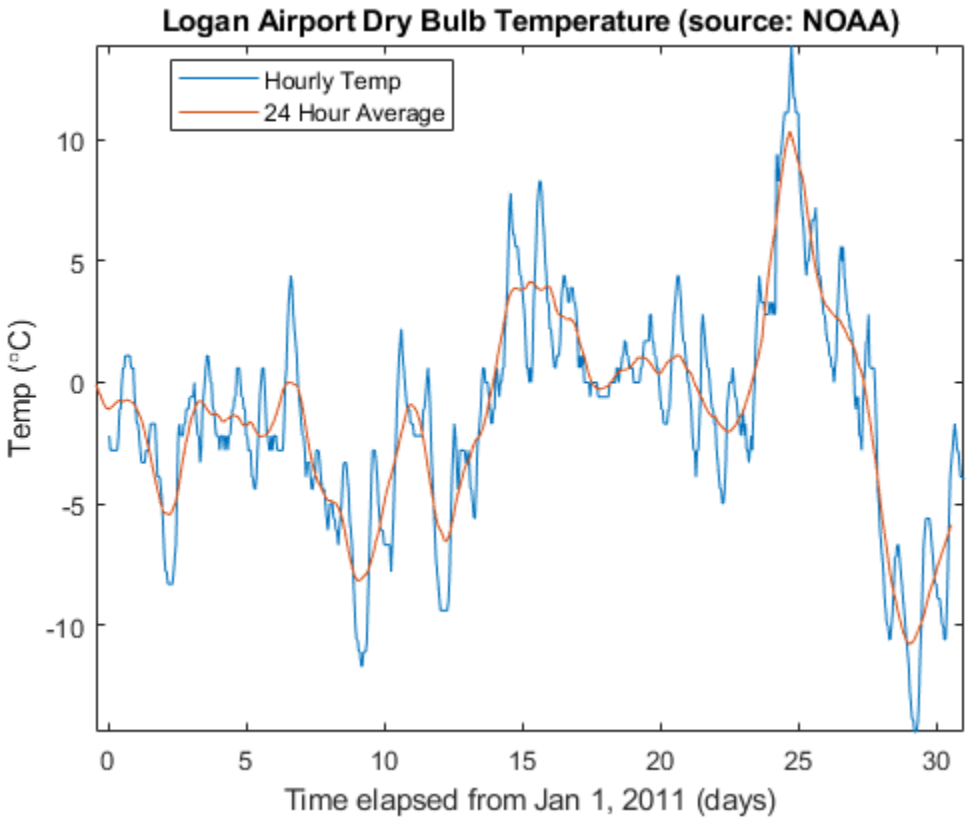


### Filter Delay

Note that the filtered output is delayed by about twelve hours. This is due to the fact that our moving average filter has a delay.

Any symmetric filter of length N will have a delay of (N-1)/2 samples. We can account for this delay manually.

```
fDelay = (length(coeff24hMA)-1)/2;
plot(days,tempC, ...
     days-fDelay/24,avg24hTempC)
axis tight
legend('Hourly Temp','24 Hour Average','location','best')
ylabel('Temp (\circC)')
xlabel('Time elapsed from Jan 1, 2011 (days)')
title('Logan Airport Dry Bulb Temperature (source: NOAA)')
```

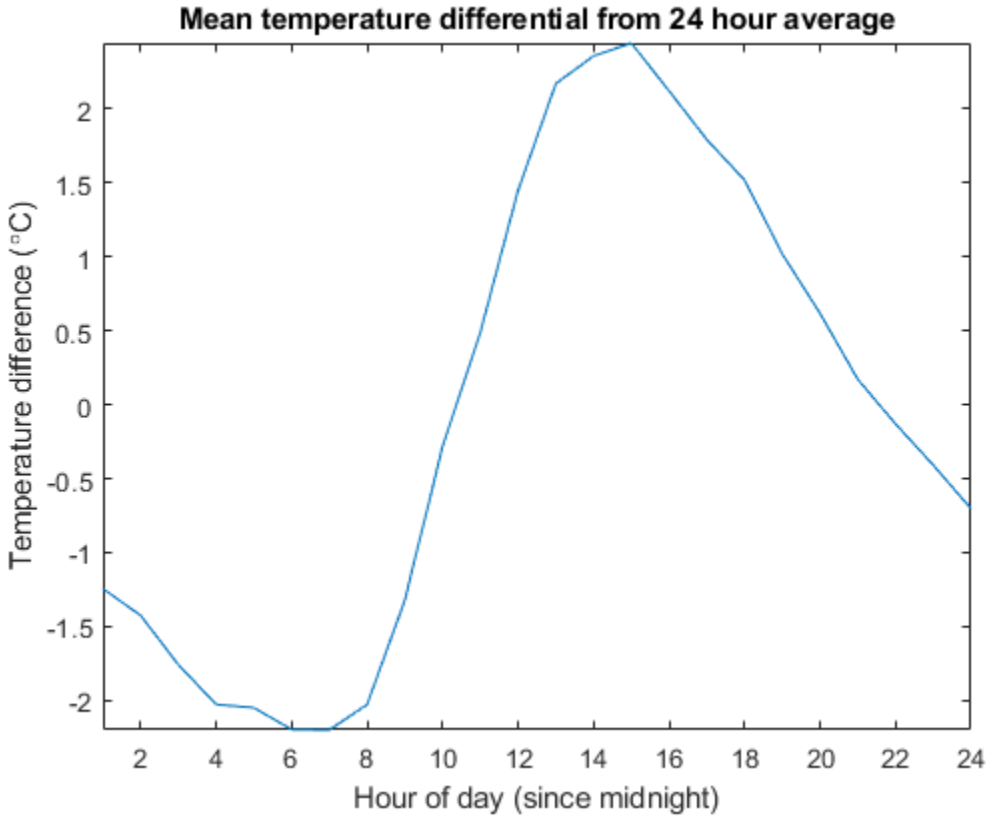



### Extracting Average Differences

Alternatively, we can also use the moving average filter to obtain a better estimate of how the time of day affects the overall temperature. To do this, first, subtract the smoothed data from the hourly temperature measurements. Then, segment the differenced data into days and take the average over all 31 days in the month.

```
figure
deltaTempC = tempC - avg24hTempC;
deltaTempC = reshape(deltaTempC, 24, 31).';
```

```
plot(1:24, mean(deltaTempC))
axis tight
title('Mean temperature differential from 24 hour average')
xlabel('Hour of day (since midnight)')
ylabel('Temperature difference (\circC)')
```

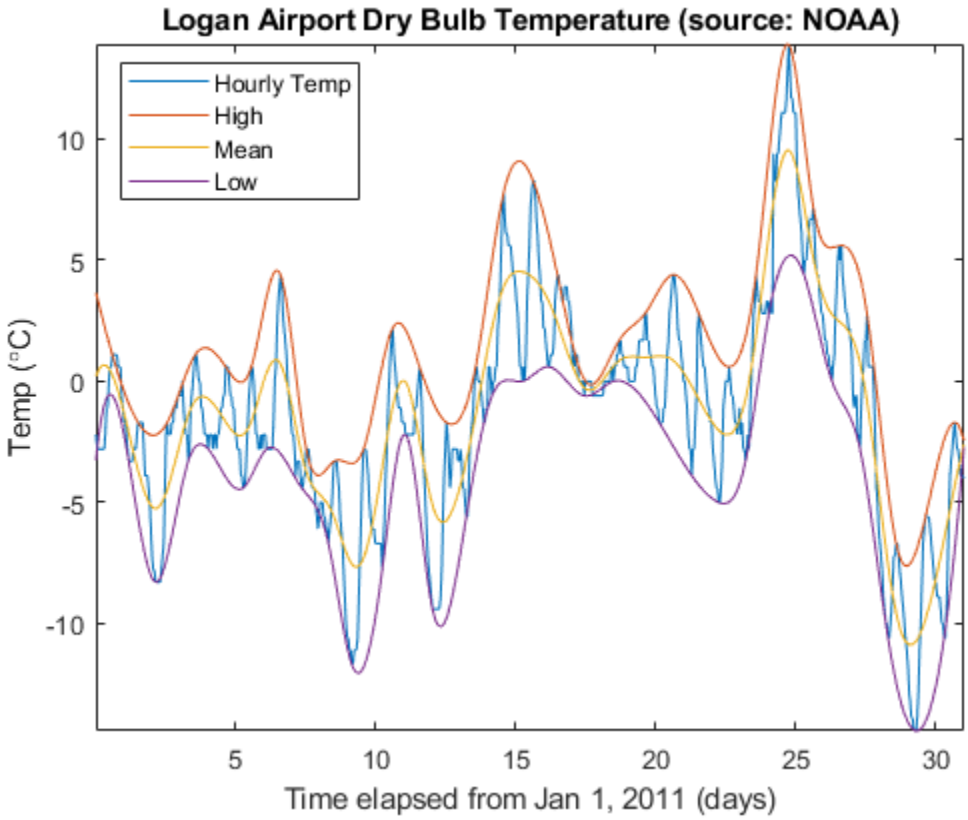



### Extracting Peak Envelope

Sometimes we would also like to have a smoothly varying estimate of how the highs and lows of our temperature signal change daily. To do this we can use the `envelope` function to connect extreme highs and lows detected over a subset of the 24 hour period. In this example, we ensure there are at least 16 hours between each extreme high and extreme low. We can also get a sense of how the highs and lows are trending by taking the average between the two extremes.

```
[envHigh, envLow] = envelope(tempC,16,'peak');
envMean = (envHigh+envLow)/2;

plot(days,tempC, ...
     days,envHigh, ...
     days,envMean, ...
     days,envLow)

axis tight
legend('Hourly Temp','High','Mean','Low','location','best')
ylabel('Temp (\circC)')
```

```
xlabel('Time elapsed from Jan 1, 2011 (days)')
title('Logan Airport Dry Bulb Temperature (source: NOAA)')
```



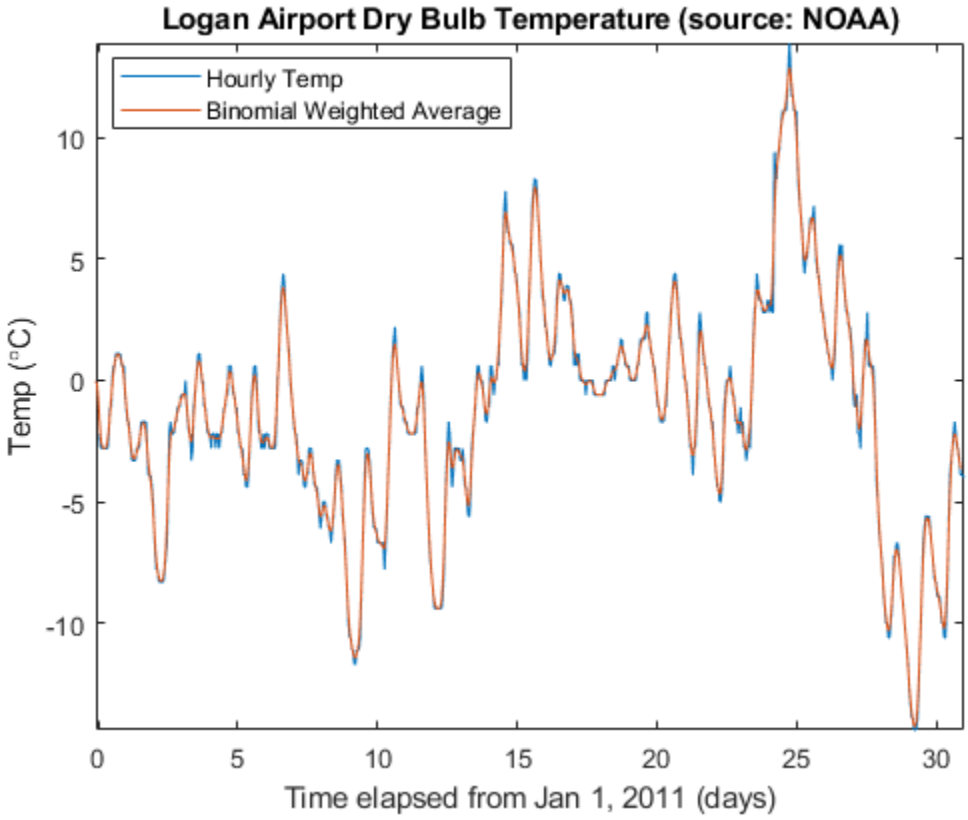

### Weighted Moving Average Filters

Other kinds of moving average filters do not weight each sample equally.

Another common filter follows the binomial expansion of $[1/2, 1/2]^n$. This type of filter approximates a normal curve for large values of n. It is useful for filtering out high frequency noise for small n. To find the coefficients for the binomial filter, convolve $[1/2, 1/2]$ with itself and then iteratively convolve the output with $[1/2, 1/2]$ a prescribed number of times. In this example, use five total iterations.

```
h = [1/2 1/2];
binomialCoeff = conv(h,h);
for n = 1:4
    binomialCoeff = conv(binomialCoeff,h);
end

figure
fDelay = (length(binomialCoeff)-1)/2;
binomialMA = filter(binomialCoeff, 1, tempC);
plot(days,tempC, ...
     days-fDelay/24,binomialMA)
axis tight
legend('Hourly Temp','Binomial Weighted Average','location','best')
ylabel('Temp (\circC)')
```

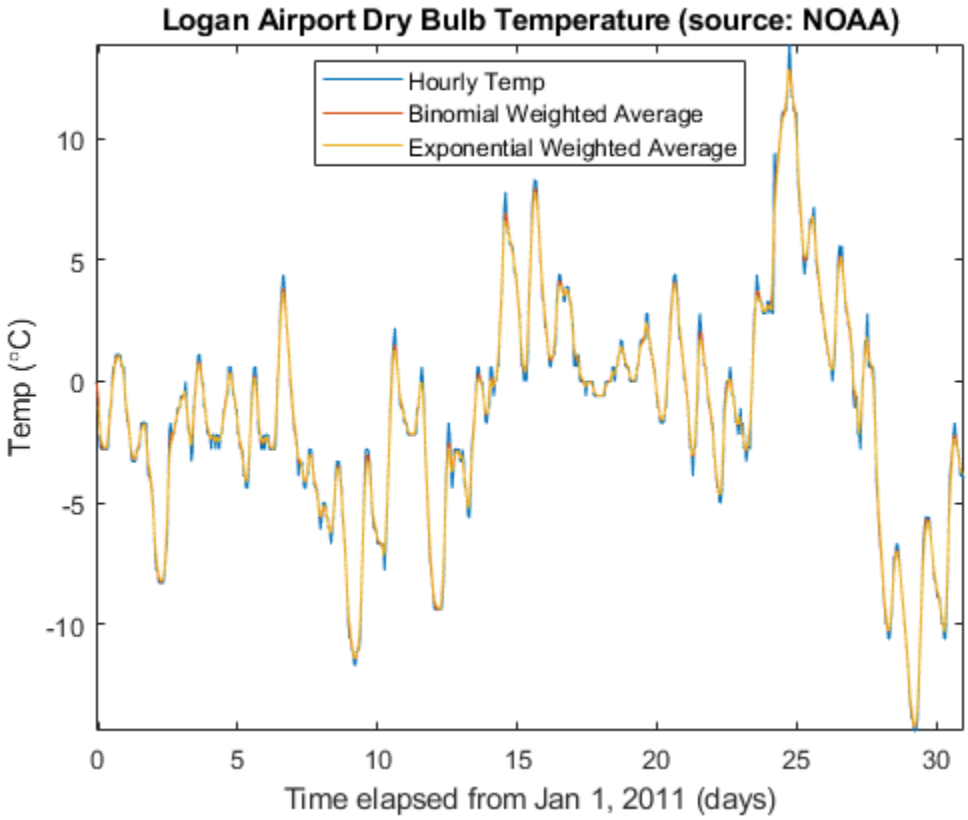
```
xlabel('Time elapsed from Jan 1, 2011 (days)')
title('Logan Airport Dry Bulb Temperature (source: NOAA)')
```




Another filter somewhat similar to the Gaussian expansion filter is the exponential moving average filter. This type of weighted moving average filter is easy to construct and does not require a large window size.

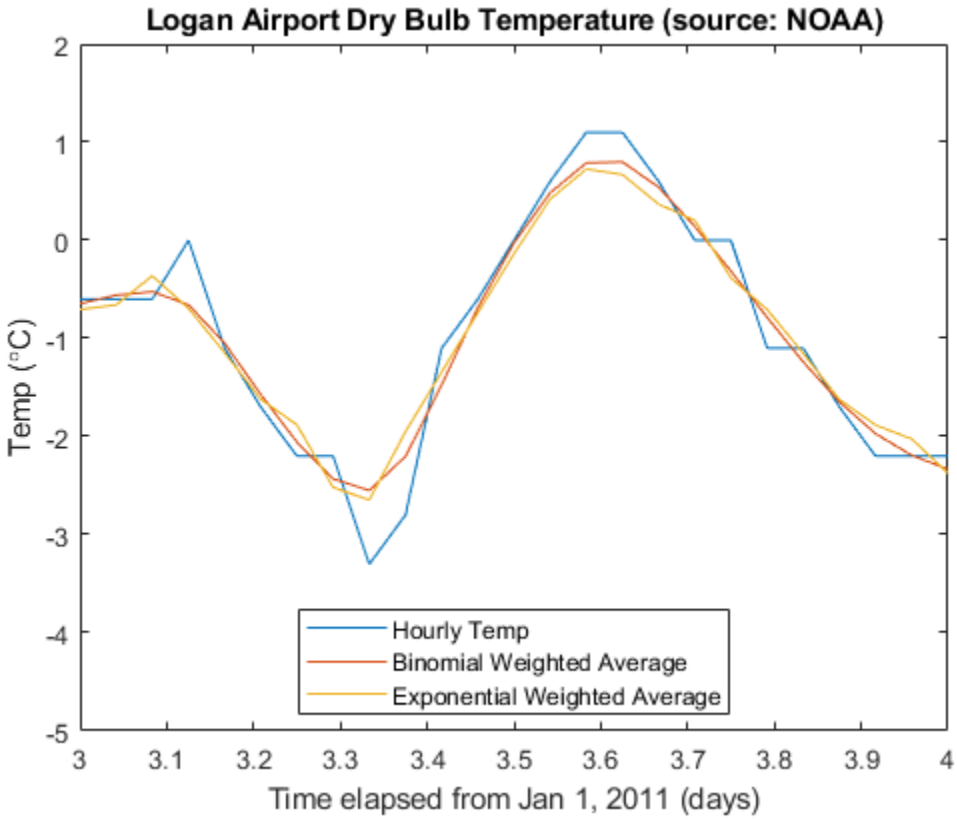
You adjust an exponentially weighted moving average filter by an alpha parameter between zero and one. A higher value of alpha will have less smoothing.

```
alpha = 0.45;
exponentialMA = filter(alpha, [1 alpha-1], tempC);
plot(days,tempC, ...
     days-fDelay/24,binomialMA, ...
     days-1/24,exponentialMA)

axis tight
legend('Hourly Temp', ...
       'Binomial Weighted Average', ...
       'Exponential Weighted Average','location','best')
ylabel('Temp (\circC)')
xlabel('Time elapsed from Jan 1, 2011 (days)')
title('Logan Airport Dry Bulb Temperature (source: NOAA)')
```




Zoom in on the readings for one day.

```
axis([3 4 -5 2])
```



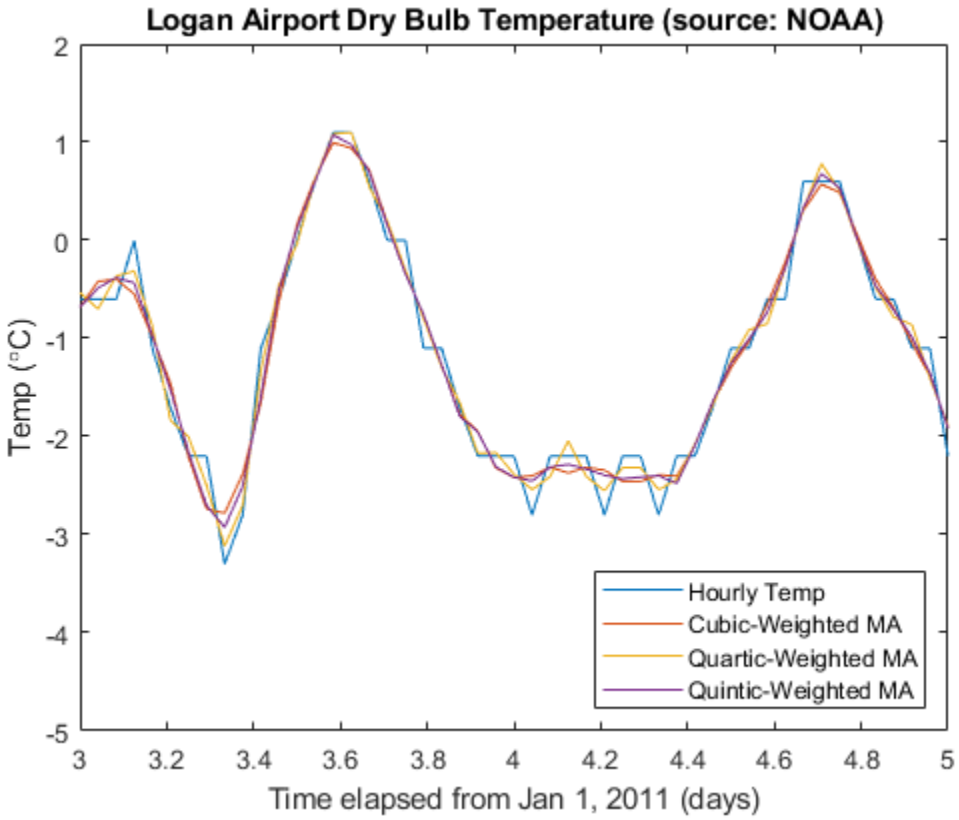

### Savitzky-Golay Filters

You'll note that by smoothing the data, the extreme values were somewhat clipped.

To track the signal a little more closely, you can use a weighted moving average filter that attempts to fit a polynomial of a specified order over a specified number of samples in a least-squares sense.

As a convenience, you can use the function `sgolayfilt` to implement a Savitzky-Golay smoothing filter. To use `sgolayfilt`, you specify an odd-length segment of the data and a polynomial order strictly less than the segment length. The `sgolayfilt` function internally computes the smoothing polynomial coefficients, performs delay alignment, and takes care of transient effects at the start and end of the data record.

```
cubicMA   = sgolayfilt(tempC, 3, 7);
quarticMA = sgolayfilt(tempC, 4, 7);
quinticMA = sgolayfilt(tempC, 5, 9);
plot(days,[tempC cubicMA quarticMA quinticMA])
legend('Hourly Temp','Cubic-Weighted MA', 'Quartic-Weighted MA', ...
       'Quintic-Weighted MA','location','southeast')
ylabel('Temp (\circC)')
xlabel('Time elapsed from Jan 1, 2011 (days)')
title('Logan Airport Dry Bulb Temperature (source: NOAA)')
axis([3 5 -5 2])
```

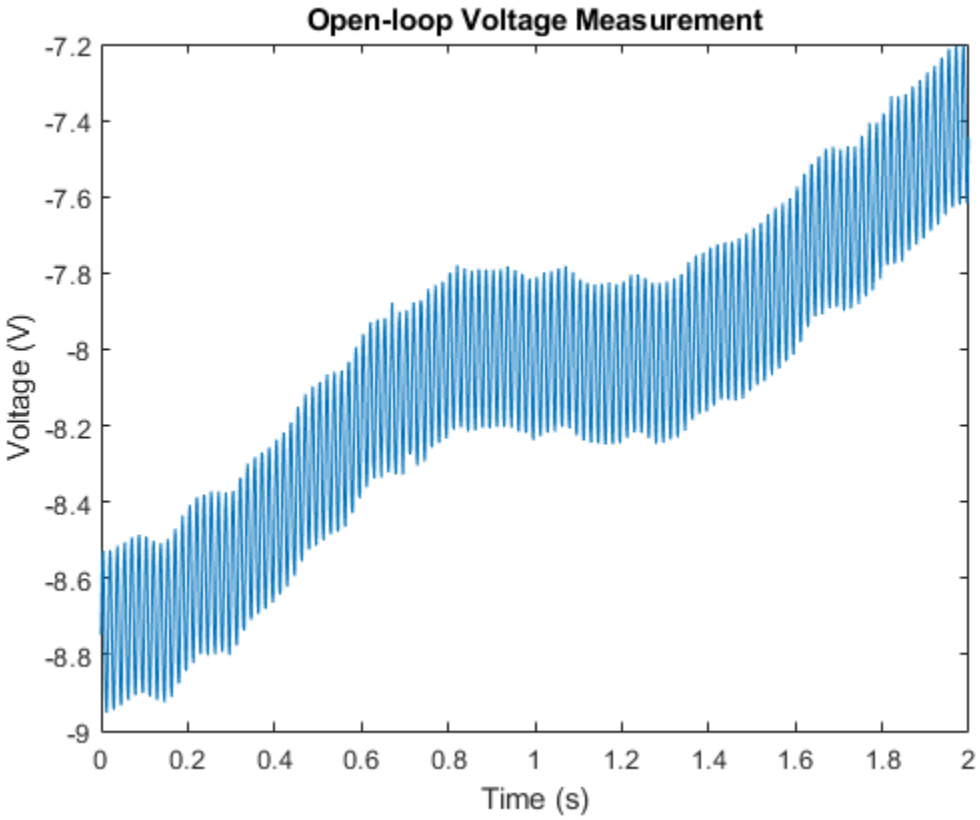



### Resampling

Sometimes it is beneficial to resample a signal in order to properly apply a moving average.

In our next example, we sampled the open-loop voltage across the input of an analog instrument in the presence of interference from 60 Hz AC power line noise. We sampled the voltage with a 1 kHz sampling rate.

```
load openloop60hertz
fs = 1000;
t = (0:numel(openLoopVoltage)-1) / fs;
plot(t,openLoopVoltage)
ylabel('Voltage (V)')
xlabel('Time (s)')
title('Open-loop Voltage Measurement')
```

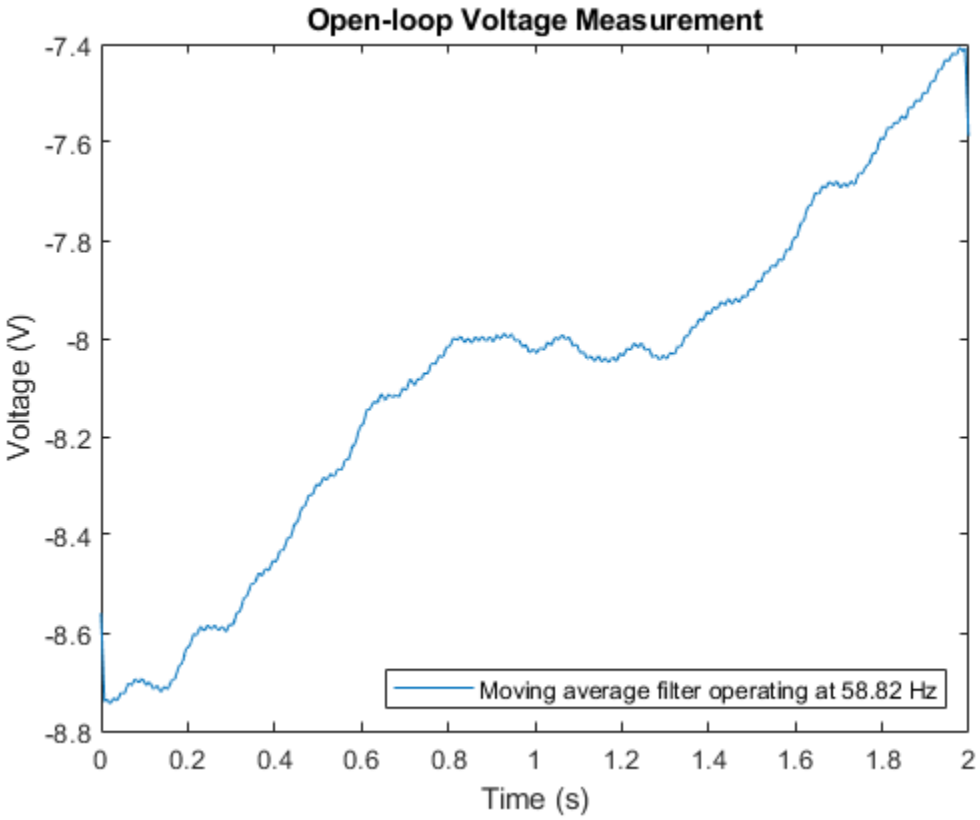



Let's attempt to remove the effect of the line noise by using a moving average filter.

If you construct a uniformly weighted moving average filter, it will remove any component that is periodic with respect to the duration of the filter.

There are roughly 1000 / 60 = 16.667 samples in a complete cycle of 60 Hz when sampled at 1000 Hz. Let's attempt to "round up" and use a 17-point filter. This will give us maximal filtering at a fundamental frequency of 1000 Hz / 17 = 58.82 Hz.

```
plot(t,sgolayfilt(openLoopVoltage,1,17))
ylabel('Voltage (V)')
xlabel('Time (s)')
title('Open-loop Voltage Measurement')
legend('Moving average filter operating at 58.82 Hz', ...
       'Location','southeast')
```

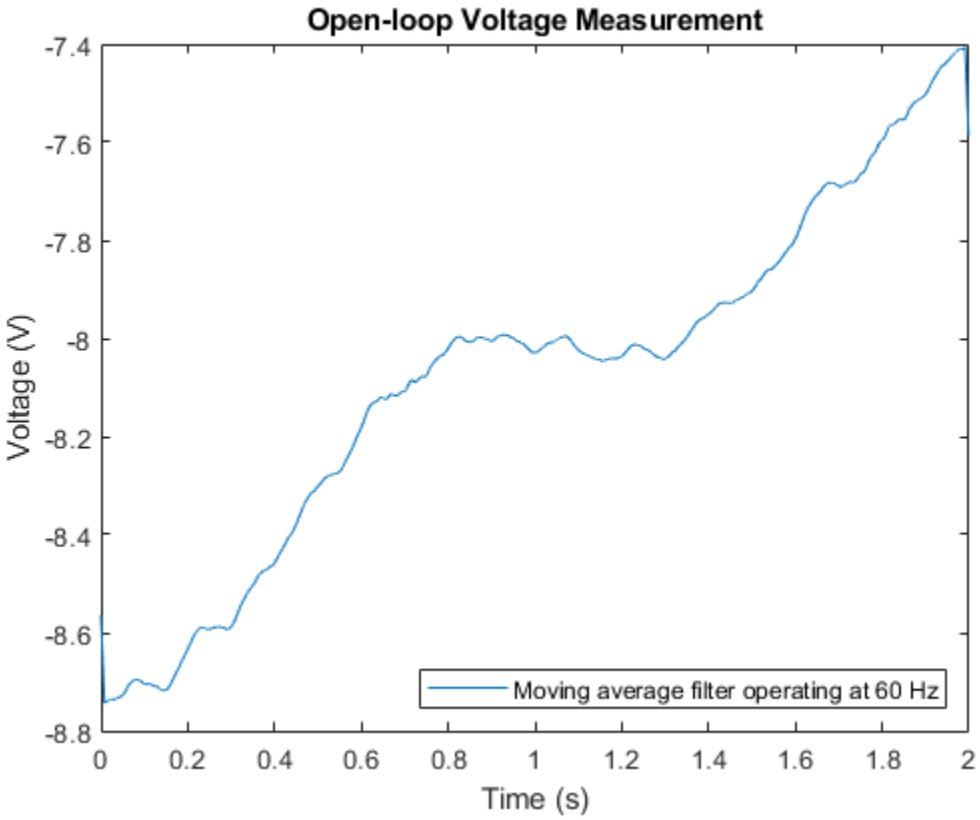



Note that while the voltage is significantly smoothed, it still contains a small 60 Hz ripple.

We can significantly reduce the ripple if we resample the signal so that we capture a complete full cycle of the 60 Hz signal by our moving average filter.

If we resample the signal at 17 * 60 Hz = 1020 Hz, we can use our 17 point moving average filter to remove the 60 Hz line noise.

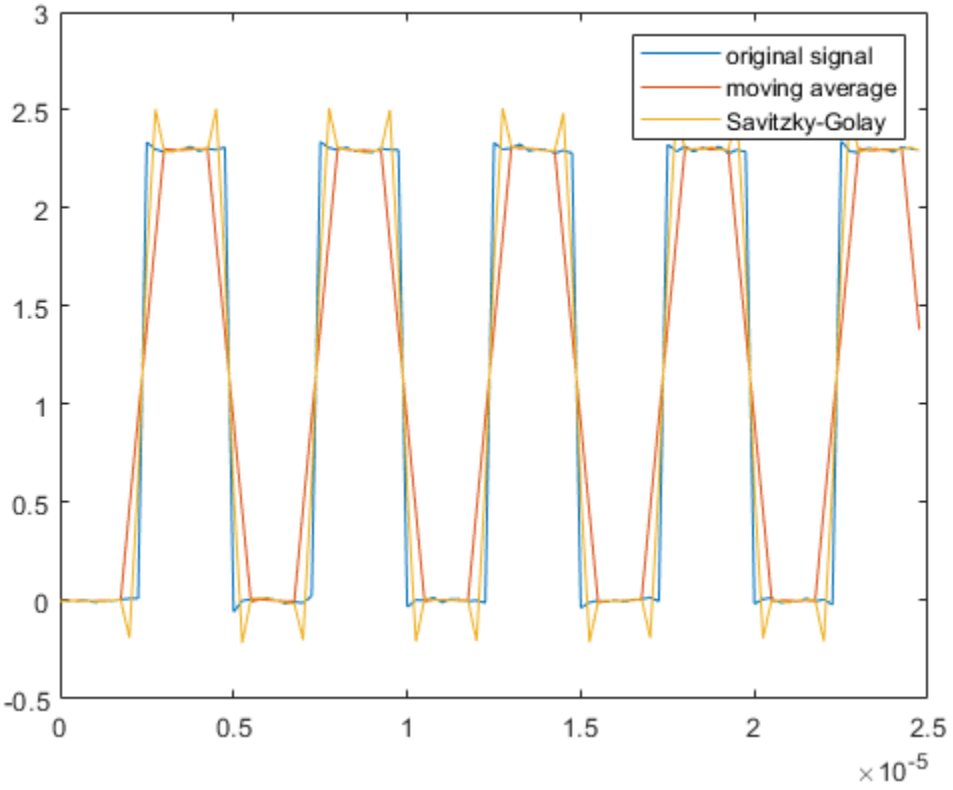
```
fsResamp = 1020;
vResamp = resample(openLoopVoltage, fsResamp, fs);
tResamp = (0:numel(vResamp)-1) / fsResamp;
vAvgResamp = sgolayfilt(vResamp,1,17);
plot(tResamp,vAvgResamp)
ylabel('Voltage (V)')
xlabel('Time (s)')
title('Open-loop Voltage Measurement')
legend('Moving average filter operating at 60 Hz', ...
    'Location','southeast')
```




### Median Filter

Moving average, weighted moving average, and Savitzky-Golay filters smooth all of the data they filter. This, however, may not always be what is wanted. For example, what if our data is taken from a clock signal and has sharp edges that we do not wish to smooth? The filters discussed so far do not work so well:

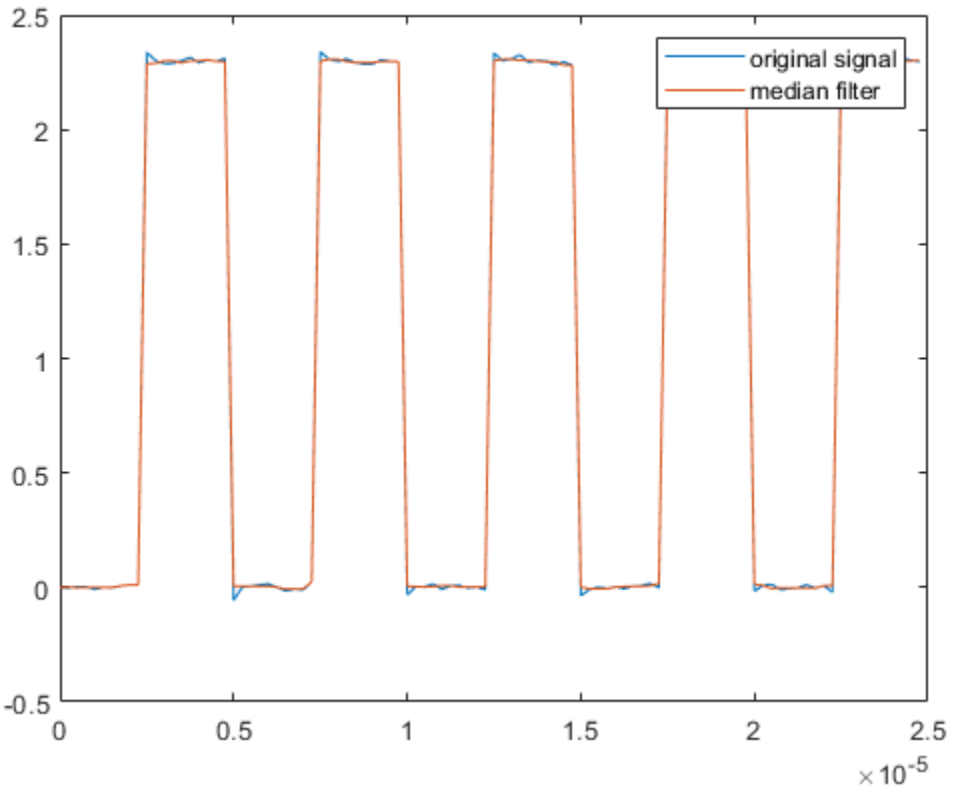
```
load clockex

yMovingAverage = conv(x,ones(5,1)/5,'same');
ySavitzkyGolay = sgolayfilt(x,3,5);
plot(t,x, ...
     t,yMovingAverage, ...
     t,ySavitzkyGolay)

legend('original signal','moving average','Savitzky-Golay')
```




The moving average and Savitzky-Golay filters respectively under-correct and over-correct near the edges of the clock signal.

A simple way to preserve the edges, but still smooth the levels is to use a median filter:

```
yMedFilt = medfilt1(x,5,'truncate');
plot(t,x, ...
     t,yMedFilt)
legend('original signal','median filter')
```

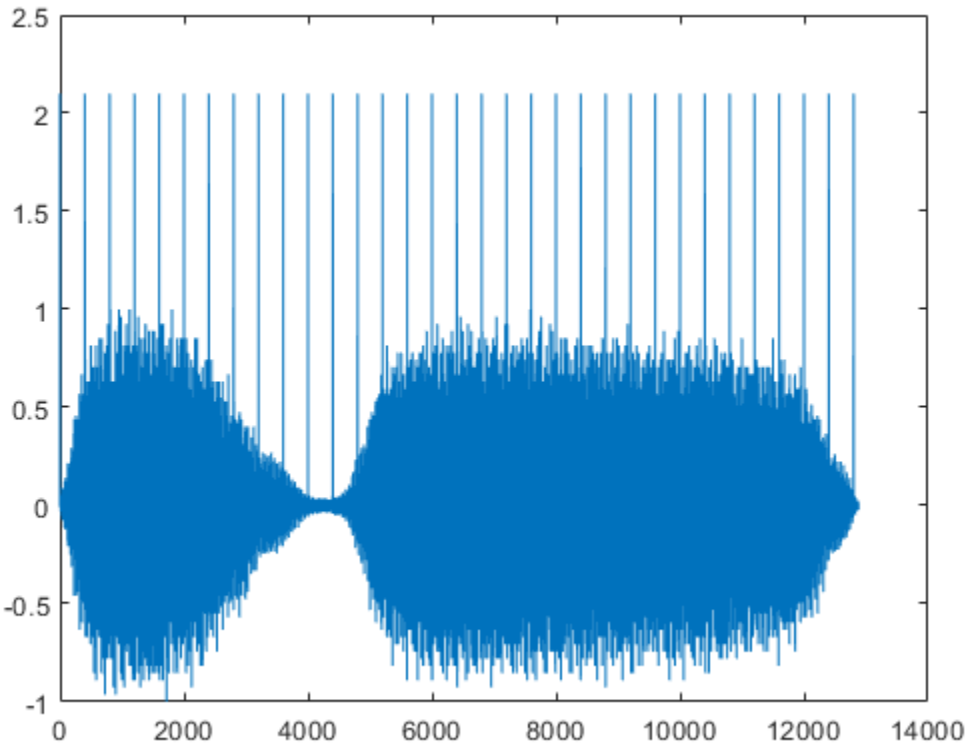



### Outlier Removal via Hampel Filter

Many filters are sensitive to outliers. A filter which is closely related to the median filter is the Hampel filter. This filter helps to remove outliers from a signal without overly smoothing the data.

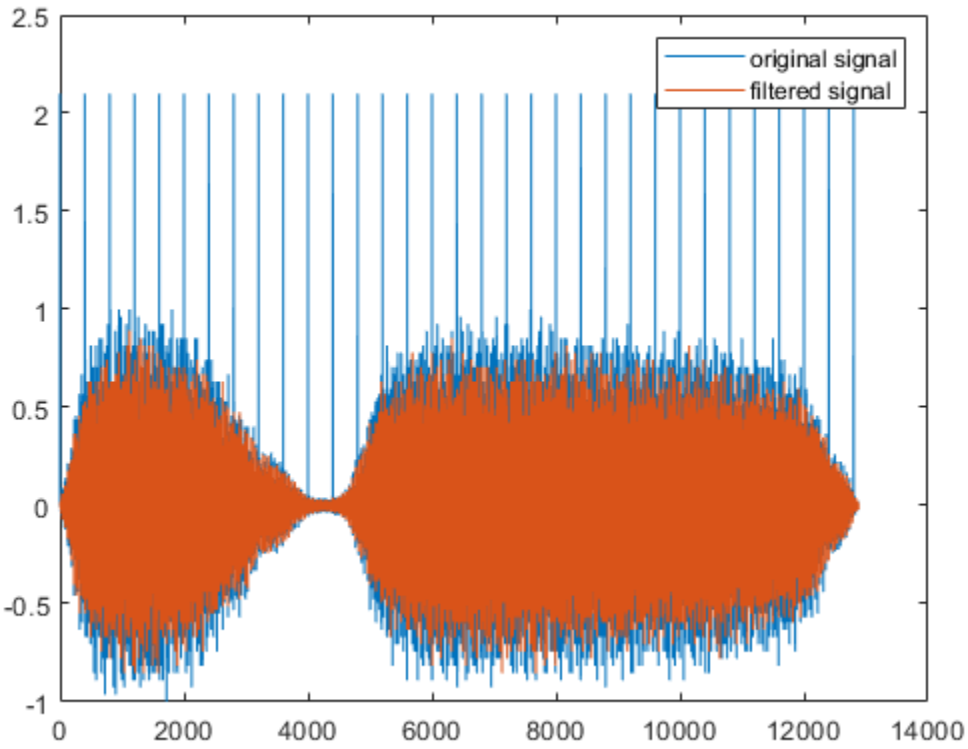
To see this, load an audio recording of a train whistle and add some artificial noise spikes:

```
load train
y(1:400:end) = 2.1;
plot(y)
```



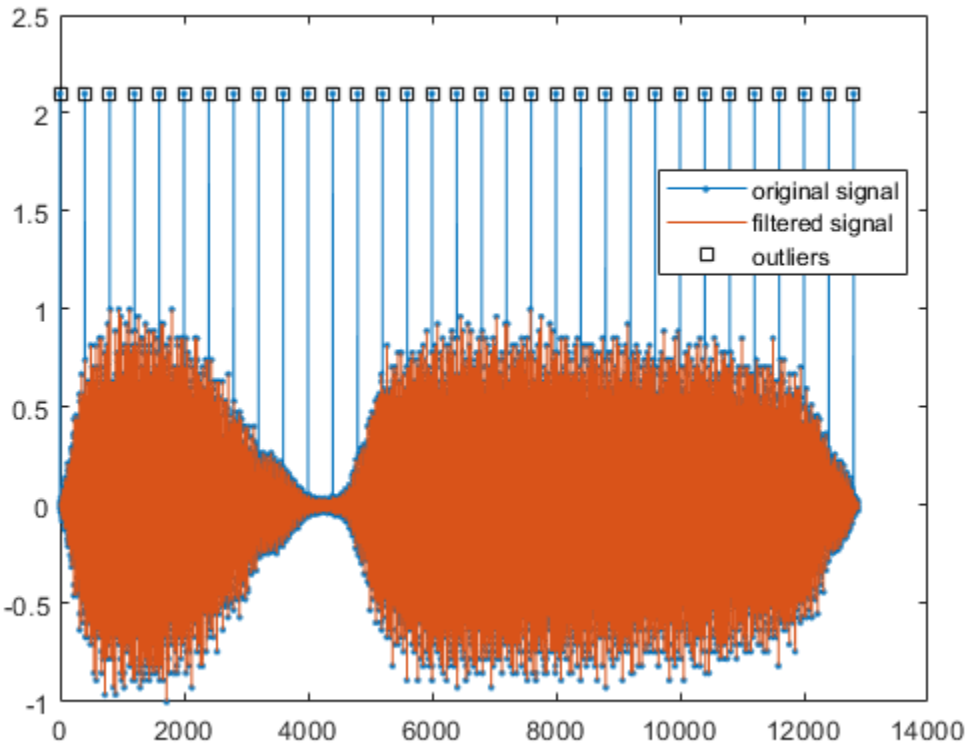

Since each spike we introduced has a duration of just one sample, we can use a median filter of just three elements to remove the spikes.

```
hold on
plot(medfilt1(y,3))
hold off
legend('original signal','filtered signal')
```




The filter removed the spikes, but it also removed a large number of data points of the original signal. A Hampel filter works similar to a median filter, however it replaces just the values which are equivalent to a few standard deviations away from the local median value.

```
hampel(y,13)
legend('location','best')
```




Only the outliers are removed from the original signal.

**Further Reading**

For more information on filtering and resampling see the Signal Processing Toolbox.

Reference: Kendall, Maurice G., Alan Stuart, and J. Keith Ord. *The Advanced Theory of Statistics, Vol. 3: Design and Analysis, and Time-Series*. 4th Ed. London: Macmillan, 1983.

## See Also

`envelope` | `hampel` | `medfilt1` | `resample` | `sgolayfilt`

# Reconstructing Missing Data

This example shows how to reconstruct missing data via interpolation, anti-aliasing filtering, and autoregressive modeling.

### Introduction

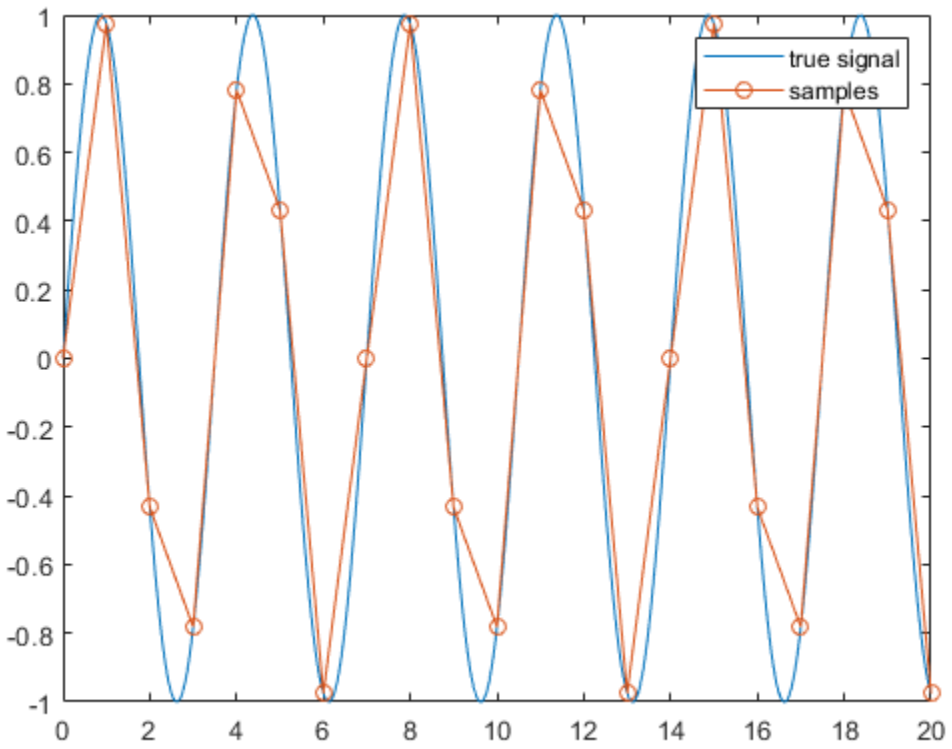
With the advent of cheap data acquisition hardware, you often have access to signals that are rapidly sampled at regular intervals. This allows you to gain a fine approximation to the underlying signal. But what happens when the data you are measuring are coarsely sampled or otherwise missing significant portions? How do you infer the values of the signals at points in between the samples that you know?

### Linear Interpolation

Linear interpolation is by far the most common method of inferring values between sampled points. By default, when you plot a vector in MATLAB, you see the points connected by straight lines. You need to sample a signal at very fine detail in order to approximate the true signal.

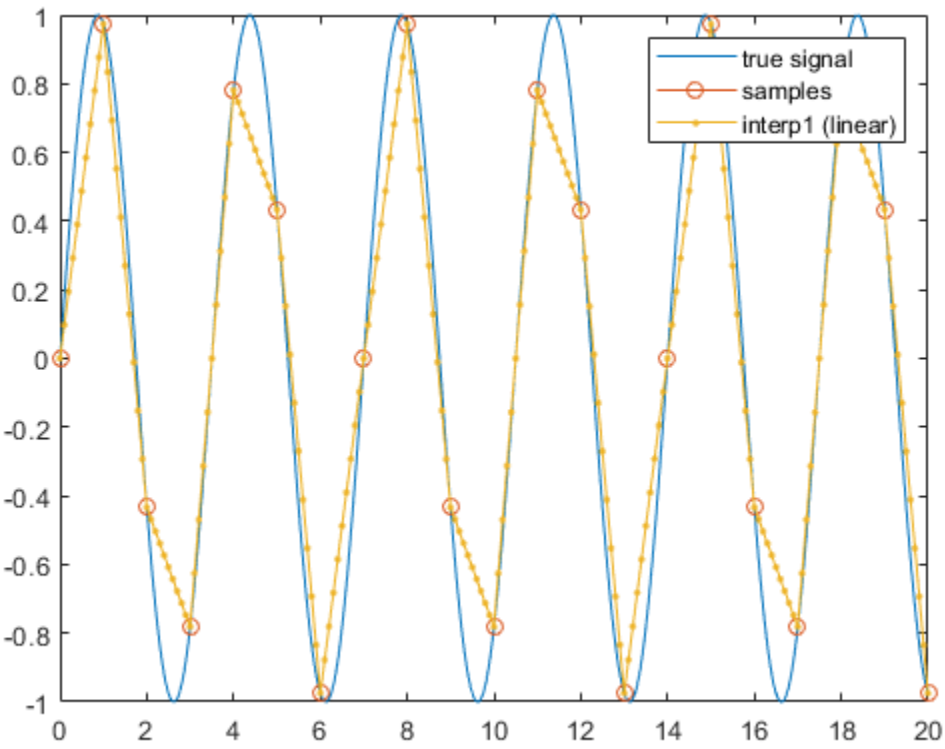
In this example, a sinusoid is sampled with both fine and coarse resolution. When plotted on a graph, the finely sampled sinusoid very closely resembles what the true continuous sinusoid would look like. Thus, you can use it as a model of the "true signal." In the plot below, the samples of a coarsely sampled signal are shown as circles connected by straight lines.

```
tTrueSignal = 0:0.01:20;
xTrueSignal = sin(2*pi*2*tTrueSignal/7);

tSampled = 0:20;
xSampled = sin(2*pi*2*tSampled/7);

plot(tTrueSignal,xTrueSignal,'-', ...
    tSampled,xSampled,'o-')
legend('true signal','samples')
```




It is straightforward to recover intermediate samples in the same way that `plot` performs interpolation. This can be accomplished with the linear method of the `interp1` function.

```
tResampled = 0:0.1:20;
xLinear = interp1(tSampled,xSampled,tResampled,'linear');
plot(tTrueSignal,xTrueSignal,'-', ...
     tSampled, xSampled, 'o-', ...
     tResampled,xLinear,'.-')
legend('true signal','samples','interp1 (linear)')
```

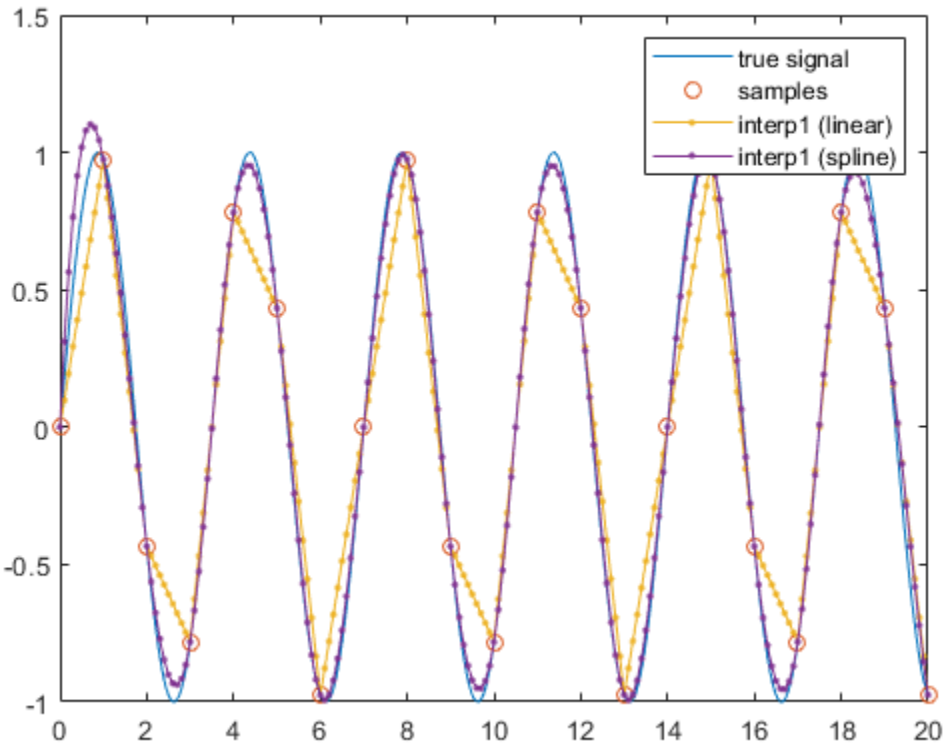



The problem with linear interpolation is that the result is not very smooth. Other interpolation methods can produce smoother approximations.

### Spline Interpolation

Many physical signals are like sinusoids in that they are continuous and have continuous derivatives. You can reconstruct such signals by using cubic spline interpolation, which ensures that the first and second derivatives of the interpolated signal are continuous at every data point:

```
xSpline = interp1(tSampled,xSampled,tResampled,'spline');
plot(tTrueSignal,xTrueSignal,'-', ...
     tSampled, xSampled,'o', ...
     tResampled,xLinear,'.-', ...
     tResampled,xSpline,'.-')
legend('true signal','samples','interp1 (linear)','interp1 (spline)')
```

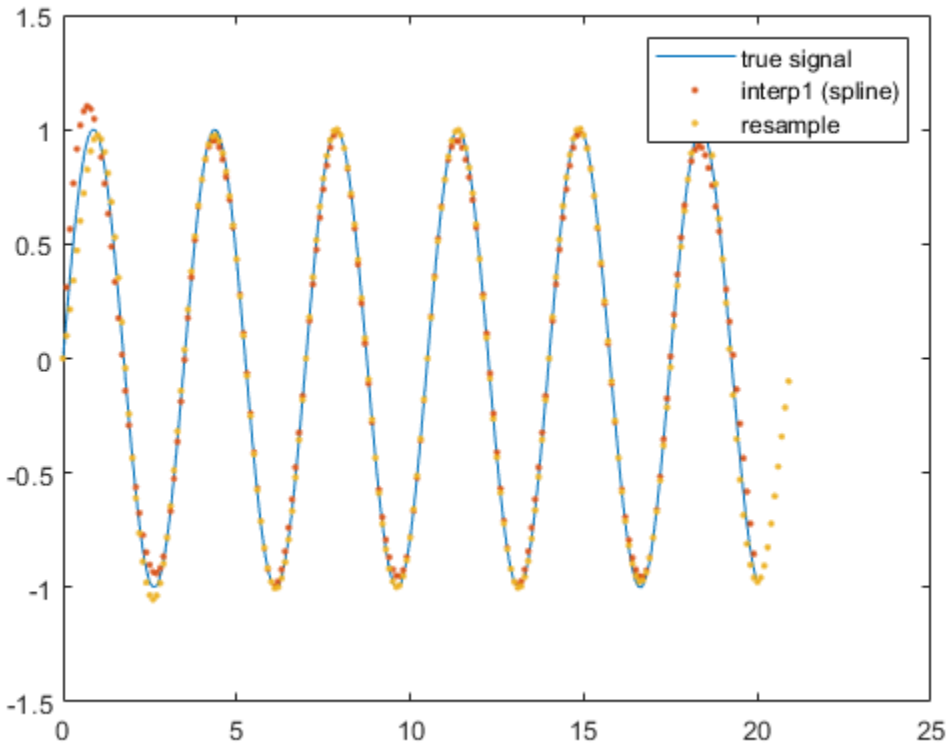



Cubic splines are particularly effective when interpolating signals that consist of sinusoids. However, there are other techniques that can be used to gain greater fidelity to physical signals which have continuous derivatives up to a very high order.

**Resampling with Antialiasing Filters**

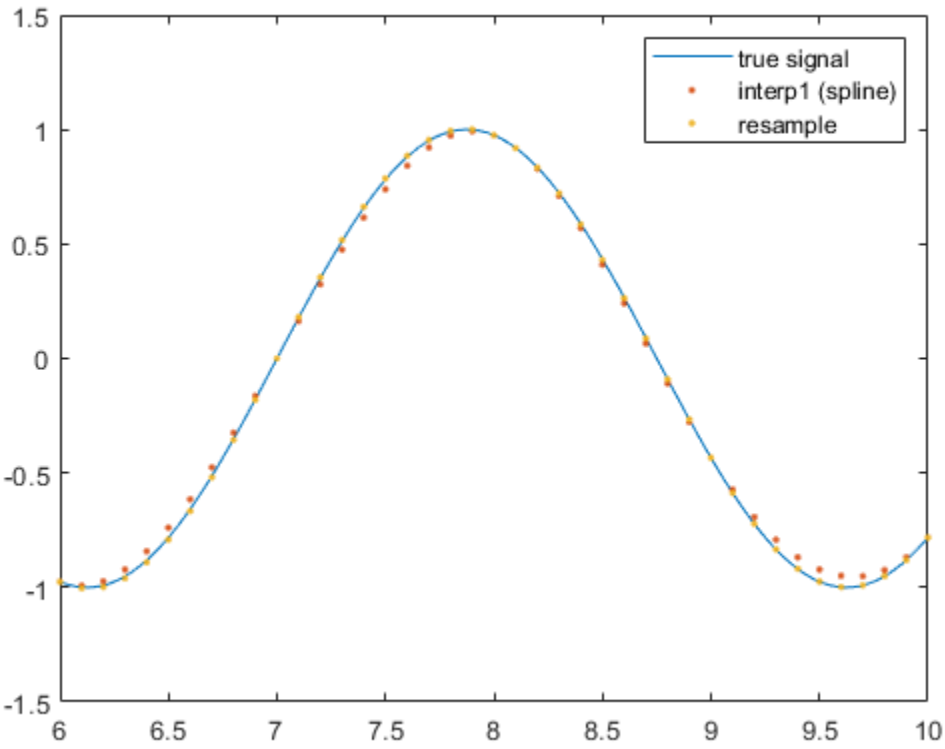
The `resample` function in the Signal Processing Toolbox provides another technique to fill in missing data. `resample` can reconstruct sinusoidal components of low frequency with very low distortion.

```
xResample = resample(xSampled, 10, 1);
tResample = 0.1*((1:numel(xResample))-1);

plot(tTrueSignal,xTrueSignal,'-', ...
     tResampled,xSpline, '.', ...
     tResample, xResample,'.')
legend('true signal','interp1 (spline)','resample')
```




Like the other methods, `resample` has some difficulty reconstructing the endpoints. On the other hand, the central portion of the reconstructed signal agrees very well with the true signal.

```
xlim([6 10])
```



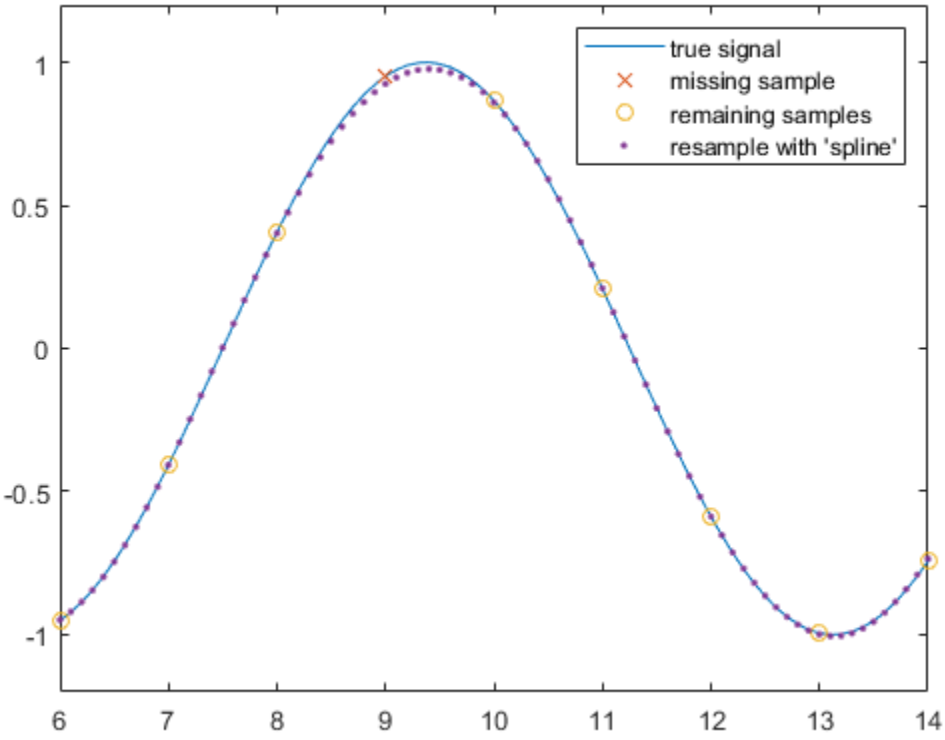

### Resampling with Missing Samples

`resample` can accommodate nonuniformly sampled signals. The technique works best when the signal is sampled at a high rate.

In the following example, we create a slowly moving sinusoid, remove a sample, and zoom into the vicinity of the missing sample.

```
tTrueSignal = 0:.1:20;
xTrueSignal = sin(2*pi*2*tTrueSignal/15);

Tx = 0:20;

Tmissing = Tx(10);
Tx(10) = [];

x = sin(2*pi*2*Tx/15);
Xmissing = sin(2*pi*2*Tmissing/15);

[y, Ty] = resample(x,Tx,10,'spline');

plot(tTrueSignal, xTrueSignal, '-', ...
     Tmissing,Xmissing,'x ', ...
     Tx,x,'o ', ...
     Ty,y,'. ')
legend('true signal','missing sample','remaining samples','resample with ''spline''')
```

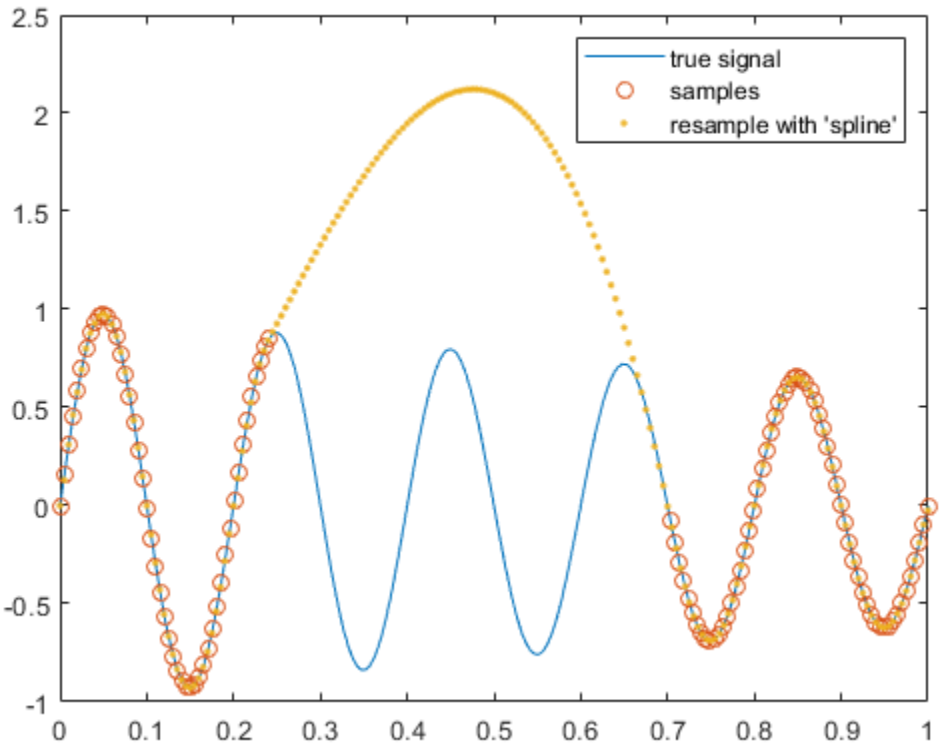
```
ylim([-1.2 1.2])
xlim([6 14])
```




The reconstructed sinusoid tracks the shape of the true signal reasonably well, with only a slight error in the vicinity of the missing sample.

However, `resample` does not work well when there is a large gap in the signal. For example, consider a dampened sinusoid whose middle portion is missing:

```
tTrueSignal = (0:199)/199;
xTrueSignal = exp(-0.5*tTrueSignal).*sin(2*pi*5*tTrueSignal);

tMissing = tTrueSignal;
xMissing = xTrueSignal;
tMissing(50:140) = [];
xMissing(50:140) = [];

[y,Ty] = resample(xMissing, tMissing, 200, 'spline');

plot(tTrueSignal,xTrueSignal,'-', ...
     tMissing,xMissing,'o',...
     Ty,y,'.')
legend('true signal','samples','resample with ''spline''')
```



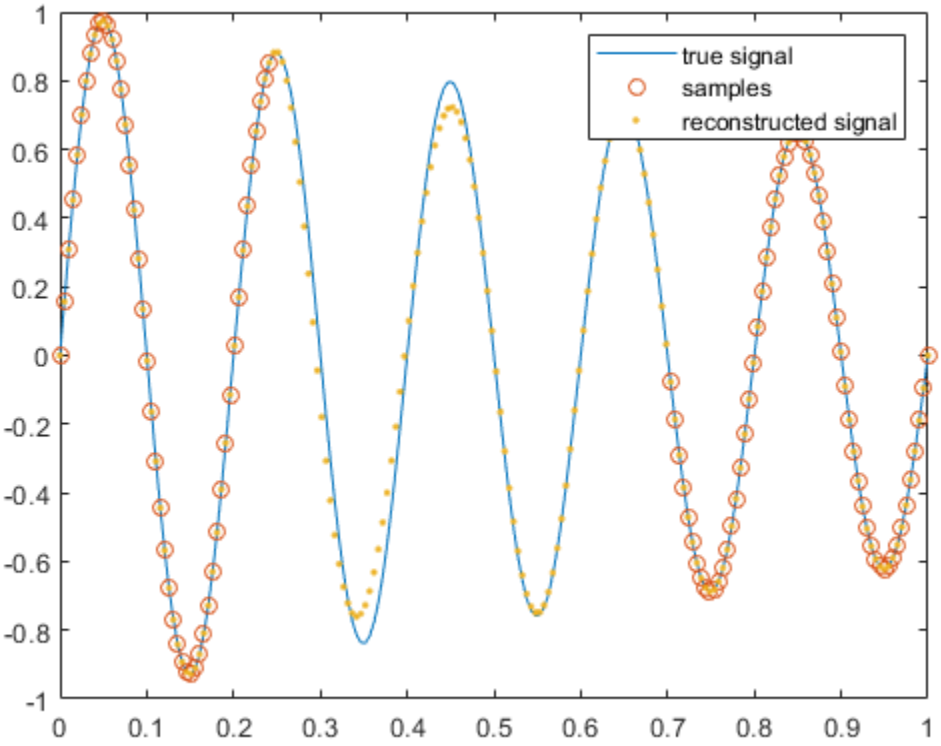

Here `resample` ensures that the reconstructed signal is continuous and has continuous derivatives in the vicinity of the missing points. However, it cannot adequately reconstruct the missing portion.

### Reconstructing Large Gaps

As can be seen above, filtering and cubic interpolation alone might not be sufficient to deal with large gaps. However, with certain kinds of sampled signals, such as those that arise when observing oscillating phenomena, you can often infer the values of missing samples based upon the data immediately preceding or following the gap.

The `fillgaps` function can replace missing samples (specified by `NaN`) in an otherwise uniformly sampled signal by fitting an autoregressive model to the samples surrounding a gap and extrapolating into the gap from both directions.

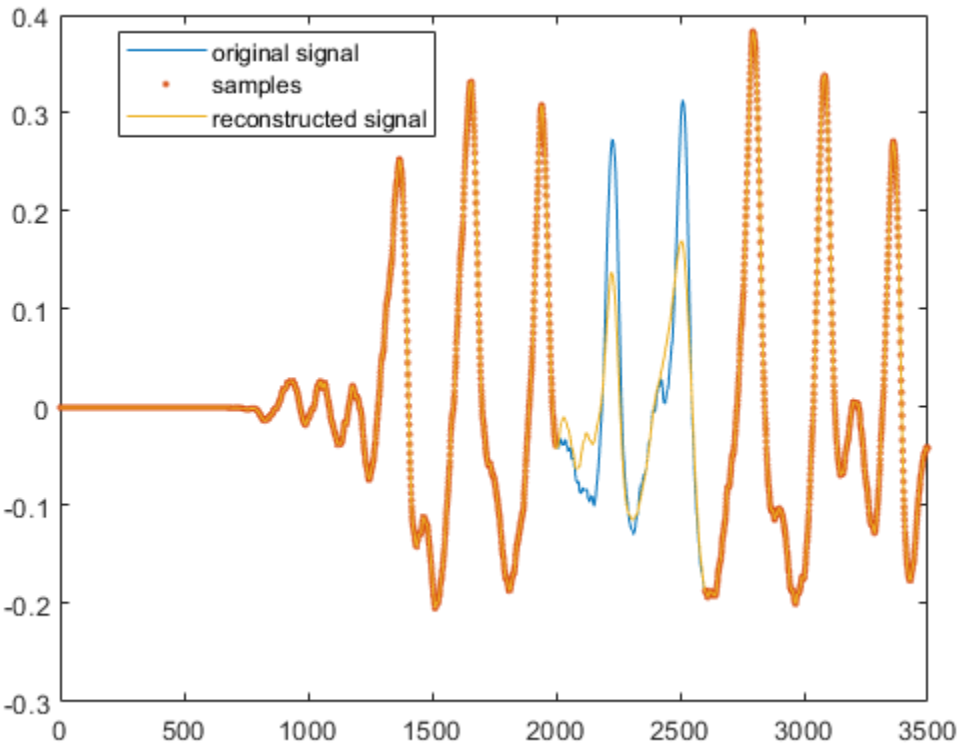
```
tTrueSignal = (0:199)/199;
xTrueSignal = exp(-.5*tTrueSignal).*sin(2*pi*5*tTrueSignal);
gapSignal = xTrueSignal;
gapSignal(50:140) = NaN;
y = fillgaps(gapSignal);

plot(tTrueSignal,xTrueSignal,'-', ...
     tTrueSignal,gapSignal,'o', ...
     tTrueSignal,y,'.')

legend('true signal','samples','reconstructed signal')
```




The technique works because autoregressive signals have information that is spread out over many samples. Only a few samples within any segment are needed to completely reconstruct the full signal.

This type of reconstruction can be adapted to estimate missing samples of more complicated signals. Consider a sampled audio signal of a plucked guitar string after removing six hundred samples immediately after the pluck:

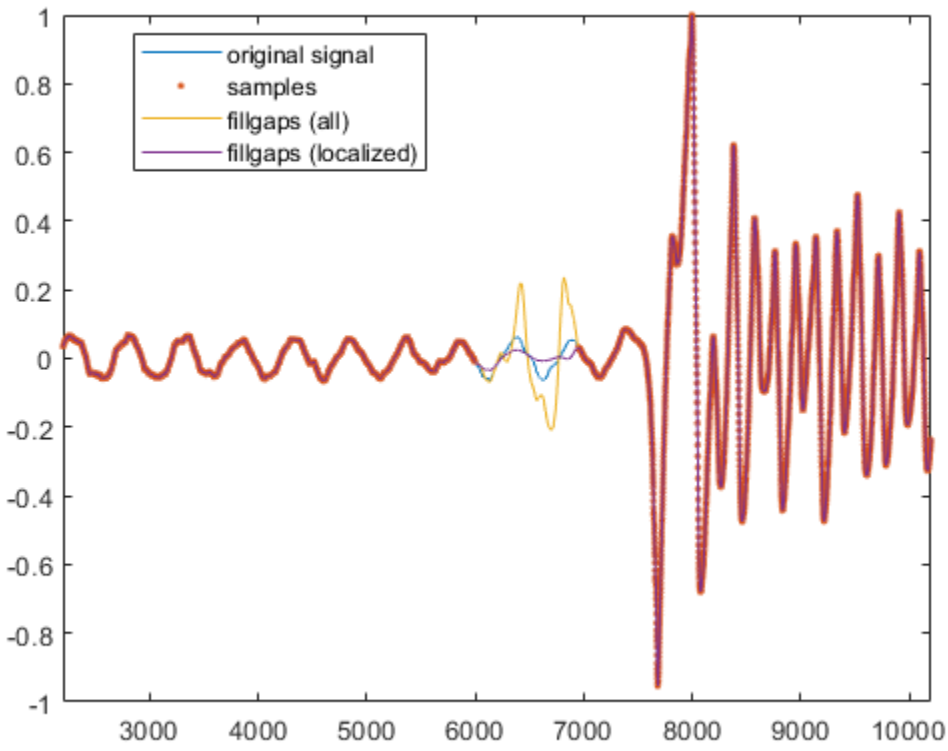
```
[y,fs] = audioread('guitartune.wav');
x = y(1:3500);
x(2000:2600) = NaN;
y2 = fillgaps(x);
plot(1:3500, y(1:3500), '-', ...
     1:3500, x, '.', ...
     1:3500, y2, '-')
legend('original signal','samples','reconstructed signal',...
       'Location','best')
```




### Reconstructing Gaps with Localized Estimation

It is fairly straightforward to fill data within a gap when it is known that the signal in the vicinity of the gap can be modeled with a single autoregressive process. You can mitigate problems when a signal consists of a non-constant autoregressive process by restricting the area over which the model parameters are computed. This is useful when you are trying to fill a gap within the "ringing" period of one resonance that comes immediately before or after another, stronger, resonance.

```
x = y(350001:370000);
x(6000:6950) = NaN;
y2 = fillgaps(x);
y3 = fillgaps(x,1500);
plot(1:20000, y(350001:370000), '-', ...
     1:20000, x, '.', ...
     1:20000, y2, '-', ...
     1:20000, y3, '-')
xlim([2200 10200])
legend('original signal','samples','fillgaps (all)','fillgaps (localized)',...
       'Location','best')
```




In the plot above, the waveform is missing a section just before a large resonance. As before, `fillgaps` is used to extrapolate into the gap region using all available data. A second call to `fillgaps` uses only 1500 samples on either side of the gap to perform the modeling. This mitigates the effect of the subsequent guitar pluck after sample 7500.

### Summary

You have seen several ways to reconstruct missing data from its neighboring sample values using interpolation, resampling and autoregressive modeling.

Interpolation and resampling work for slowly varying signals. Resampling with antialiasing filters often does a better job at reconstructing signals that consist of low-frequency components. For reconstructing large gaps in oscillating signals, autoregressive modeling in the vicinity of the gap can be particularly effective.

### Further reading

For more information on uniform and nonuniform resampling, see

- “Resampling Uniformly Sampled Signals” on page 24-38
- “Resampling Nonuniformly Sampled Signals” on page 24-46

## See Also

`fillgaps` | `interp1` | `resample`

# Resampling Uniformly Sampled Signals

This example shows how to resample a uniformly sampled signal to a new uniform rate. It shows how to reduce the impact of large transients as well as how to remove unwanted high frequency content.

### Rate Conversion by a Rational Factor

The `resample` function performs rate conversion from one sample rate to another. `resample` allows you to upsample by an integral factor, `p`, and subsequently decimate by another integral factor, `q`. In this way you can resample to a rational multiple (`p` / `q`) of the original sample rate.

To use the `resample` function on uniform samples, you must provide both the numerator and denominator of this rational factor. To determine the integers you need, you can use the `rat` function.

Here is an example of how to call `rat` when converting from 48 kHz to 44.1 kHz:

```
originalFs = 48000;
desiredFs = 44100;

[p,q] = rat(desiredFs / originalFs)
```

```
p = 147
```

```
q = 160
```

`rat` indicates that you can upsample by 147 and decimate by 160. To verify that this produces the desired rate, multiply `p` / `q` by the original sample rate:

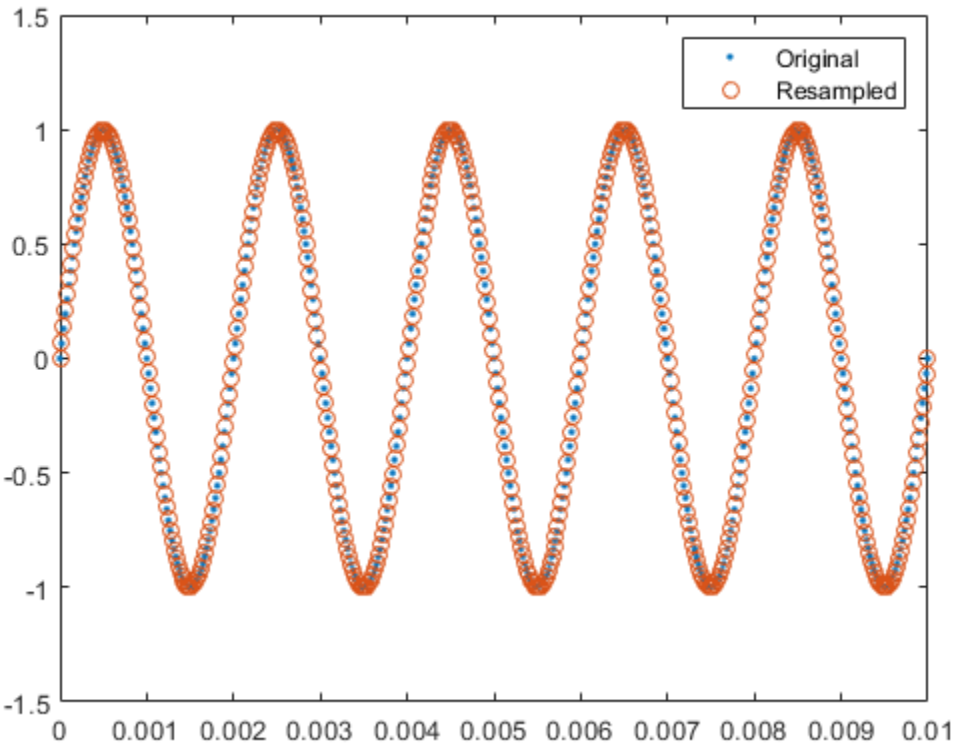
```
originalFs * p / q
```

```
ans = 44100
```

Once you have the ratio between the new and original sample rates, you can call `resample`.

For example, create a 10 millisecond long 500 Hz sinusoid using the original sample rate of 48 kHz and convert it to 44.1 kHz:

```
tEnd = 0.01;
Tx = 0:1/originalFs:tEnd;
f = 500;
x = sin(2*pi*f*Tx);

y = resample(x,p,q);
Ty = (0:numel(y)-1)/desiredFs;

plot(Tx,x,'. ')
hold on
plot(Ty,y,'o ')
hold off
legend('Original','Resampled')
```




For well-behaved signals such as the sinusoid above, simply using `resample` with a carefully chosen `p` and `q` should be enough to reconstruct it properly.

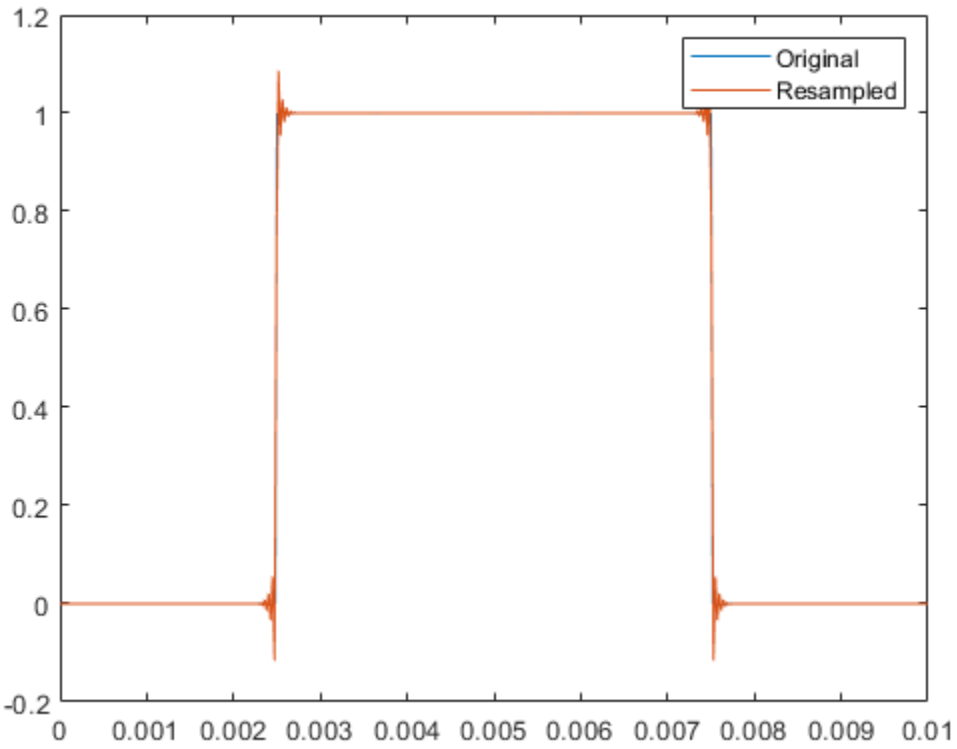
For signals with transients or significant noise, you may wish to have greater control over the poly-phase anti-aliasing filter in `resample`.

### Filtering Transients

The `resample` function uses a filter when it performs rate conversion. This filtering is sensitive to large transients in the signal.

To illustrate this, resample a rectangular pulse:

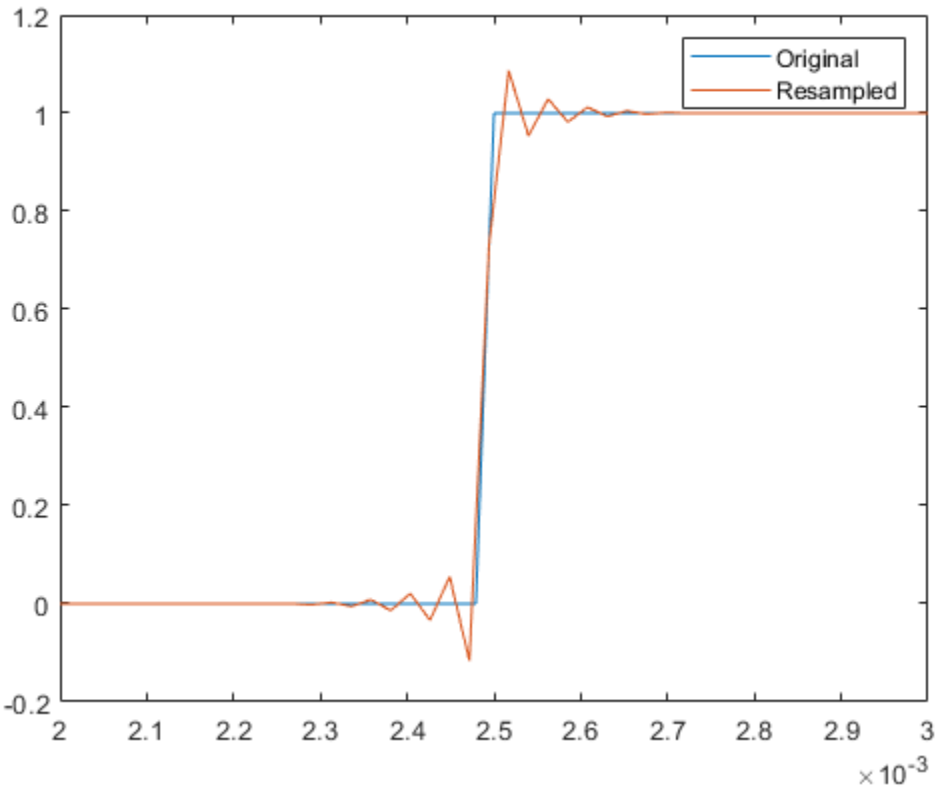
```
x = [zeros(1,120) ones(1,241) zeros(1,120)];
y = resample(x,p,q);

plot(Tx,x,'-', Ty,y,'-')
legend('Original','Resampled')
```




The function does a good job of reconstructing the flat regions of the pulse. However, there are spikes at the edges of the pulse.

Zoom in on the edge of the first pulse:

```
xlim([2e-3 3e-3])
```



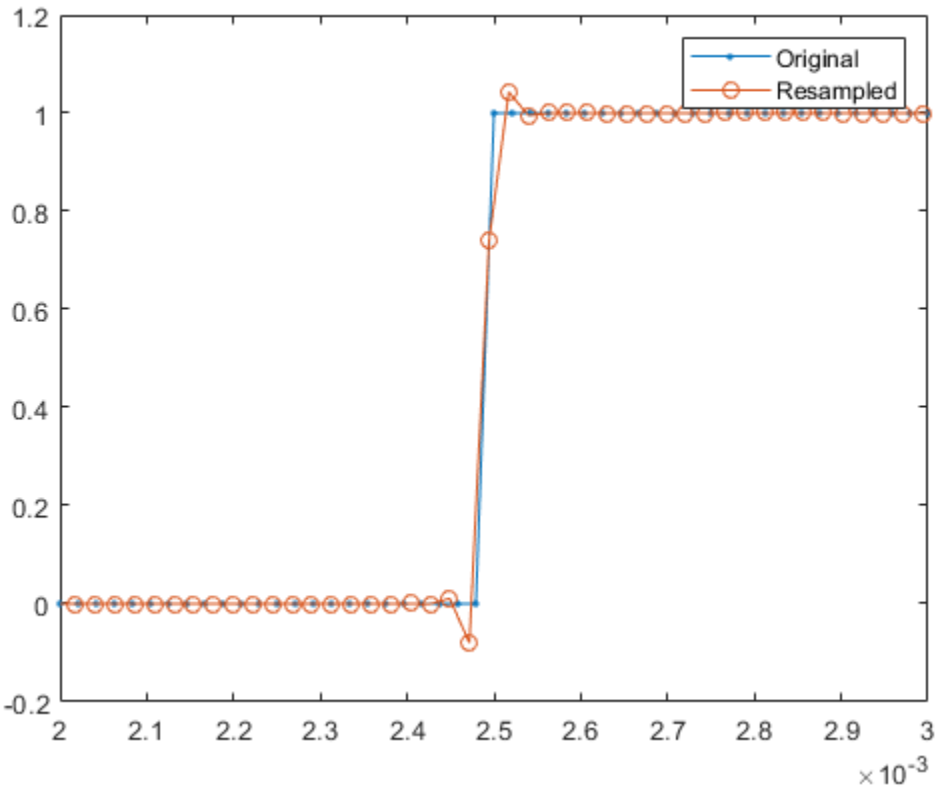

There is a damped oscillation in the transition region. You can diminish this oscillation by adjusting the settings of the internal filter.

`resample` allows you to have control over a Kaiser window applied to the anti-aliasing filter that can mitigate some of the edge effects.

Two parameters, `n` and `beta`, control the relative length of the filter and the amount of smoothing it attempts to perform. A larger value of `n` will have a larger filter length. A `beta` of 0 will have no additional smoothing. Larger `beta` values will have larger smoothing. By default, `n` is 10 and `beta` is 5.

A practical way to proceed is to start with the default values and adjust as needed. Here, set `n` to 5 and `beta` to 20.

```
n = 5;
beta = 20;

y = resample(x,p,q,n,beta);

plot(Tx,x,'.-')
hold on
plot(Ty,y,'o-')
hold off
legend('Original','Resampled')
xlim([2e-3 3e-3])
```




The oscillation is significantly reduced.

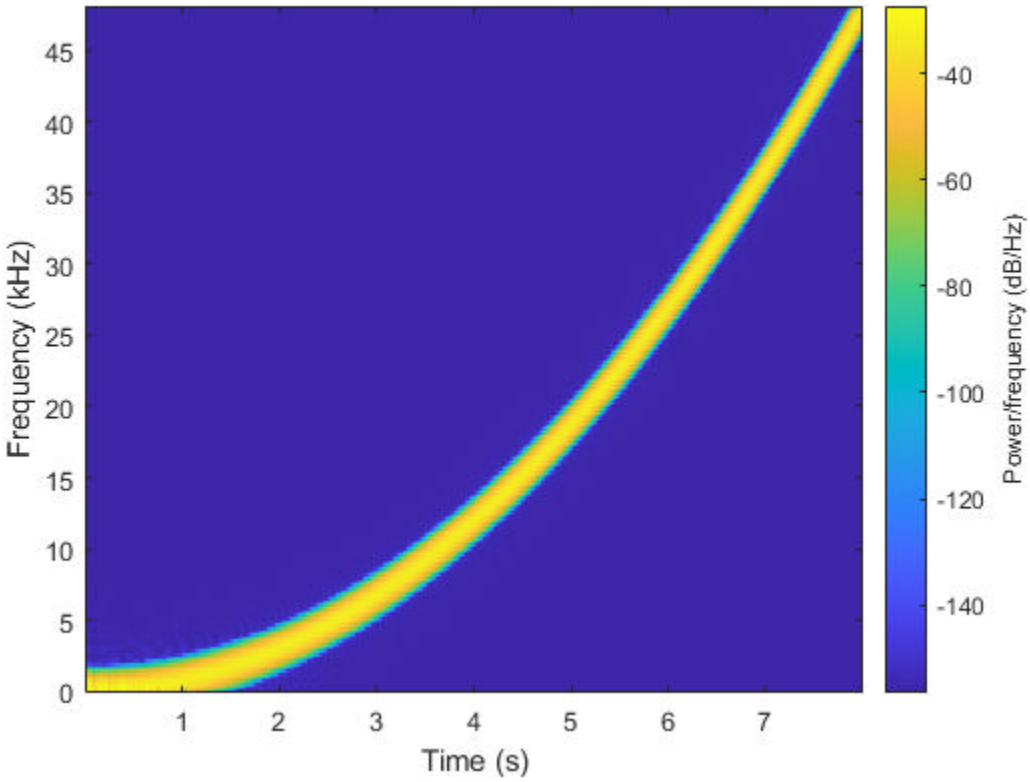
### Filtering Aliases

The `resample` function is designed to convert sample rates to either higher or lower rates. As a consequence, the frequency cutoff of the anti-aliasing filter is set to the Nyquist frequency of the input or output sample rate (whichever is lower). This default setting allows the `resample` function to cover a wide range of applications.

There are times when direct control of the filter may be beneficial.

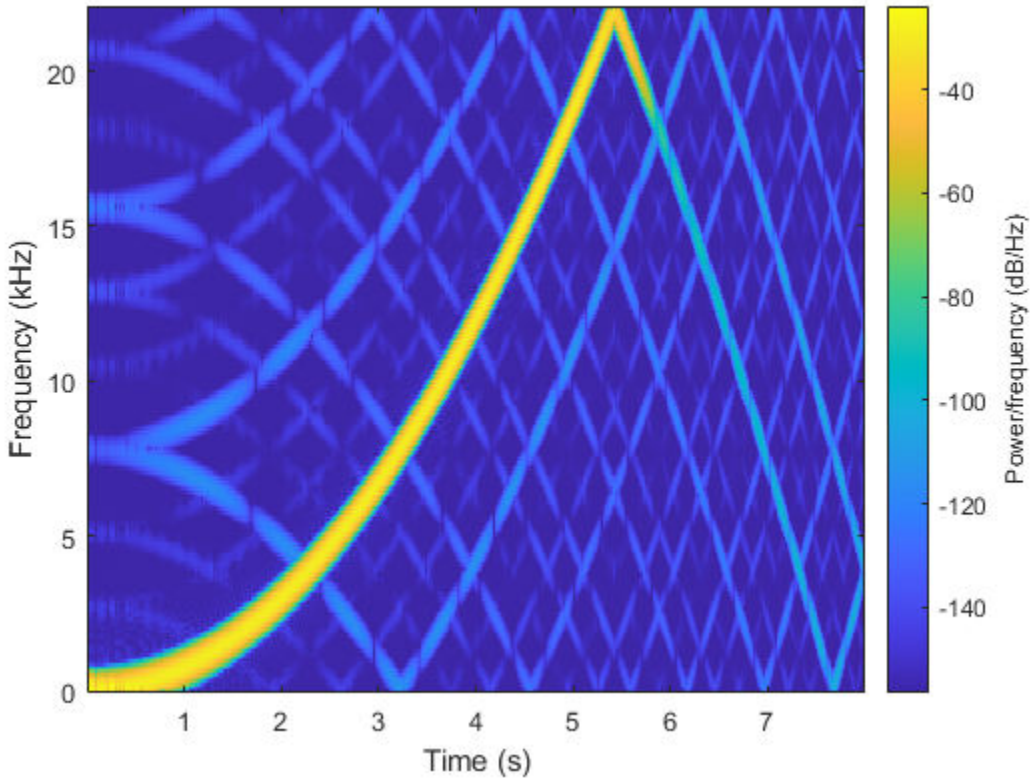
To illustrate this, construct and view the spectrogram of a chirp signal sampled at 96 kHz. The chirp signal consists of a sinusoid whose frequency varies quadratically over the entire Nyquist range from 0 Hz to 48 kHz over a duration of eight seconds:

```
fs1 = 96000;
t1 = 0:1/fs1:8;
x = chirp(t1, 0, 8, fs1/2, 'quadratic');
spectrogram(x,kaiser(256,15),220,412,fs1,'yaxis')
```




Next, convert the chirp to 44.1 kHz using the default settings of `resample` and view the spectrogram:

```
fs2 = 44100;

[p,q] = rat(fs2/fs1);
y = resample(x,p,q);

spectrogram(y,kaiser(256,15),220,412,fs2,'yaxis')
```

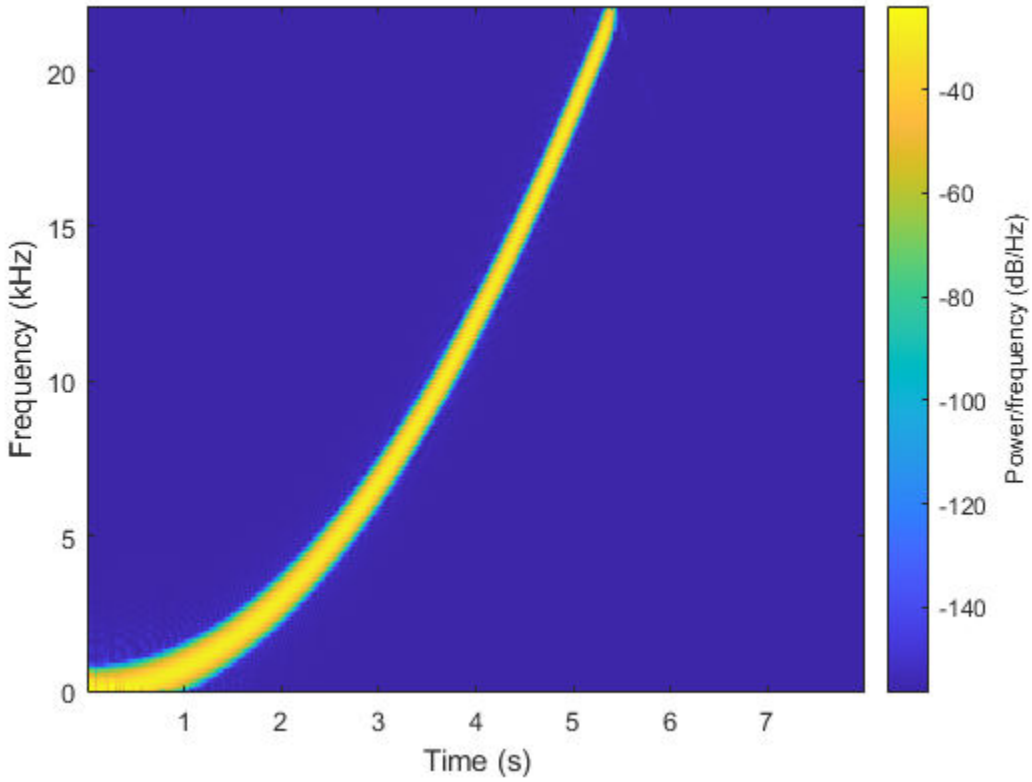



Here you can see the original signal as well as unwanted frequency content. Ideally the sinusoid should start at 0 Hz and continue until it reaches the Nyquist frequency of 22.05 kHz at 5.422 s. Instead, there are artifacts introduced due to small discontinuities introduced at the edges of the default filter used for resampling. To prevent these artifacts, you can provide a longer filter with a slightly lower cutoff frequency and greater stop-band rejection than the default filter.

To have proper temporal alignment, the filter should have odd length. The length should be a few times larger than `p` or `q` (whichever is larger). Similarly, divide the desired normalized cutoff frequency by the larger of `p` or `q`. In either case, multiply the resulting coefficients by `p`.

Here is an example of a filter with a cutoff at 98% (0.98) of the output Nyquist frequency with an order of 256 times the decimation factor, windowed with a Kaiser window with a `beta` of 12.

```
normFc = .98 / max(p,q);
order = 256 * max(p,q);
beta = 12;

lpFilt = firls(order, [0 normFc normFc 1],[1 1 0 0]);
lpFilt = lpFilt .* kaiser(order+1,beta)';
lpFilt = lpFilt / sum(lpFilt);

% multiply by p
lpFilt = p * lpFilt;

% resample and plot the response
y = resample(x,p,q,lpFilt);
spectrogram(y,kaiser(256,15),220,412,fs2,'yaxis')
```




Note that the aliases are removed.

**Further Reading**

For more information on resampling see the Signal Processing Toolbox.

Reference: fredric j harris, "Multirate Signal Processing for Communications Systems", Prentice Hall, 2004.

## See Also

`chirp` | `firls` | `resample`

# Resampling Nonuniformly Sampled Signals

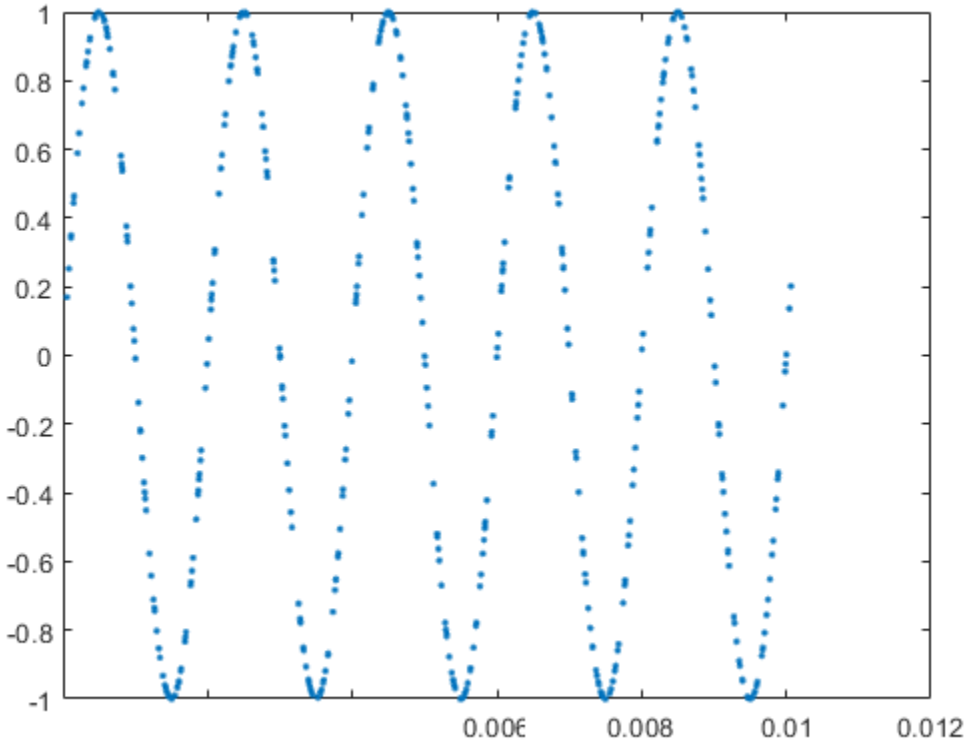
This example shows how to resample nonuniformly sampled signals to a new uniform rate. It shows how to apply a custom filter on irregularly sampled data to reduce aliasing. It also shows how to use detrending to remove transients at the start and at the end of the signal.

### Resampling Nonuniformly Sampled Signals to a Desired Rate

The `resample` function allows you to convert a nonuniformly sampled signal to a new uniform rate.

Create a 500 Hz sinusoid sampled irregularly at about 48 kHz. We simulate the irregularity by adding random values to the uniform vector.

```
rng default
nominalFs = 48000;
f = 500;
Tx = 0:1/nominalFs:0.01;
irregTx = sort(Tx + 1e-4*rand(size(Tx)));
x = sin(2*pi*f*irregTx);
plot(irregTx,x,'.')
```

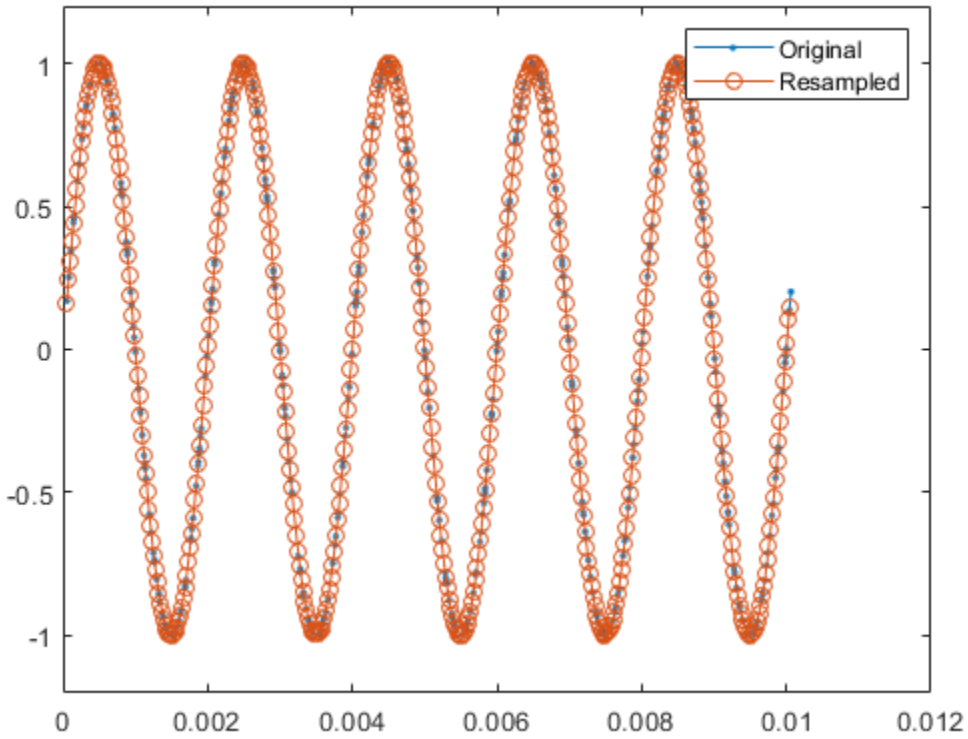



To resample a nonuniformly sampled signal, you can call `resample` with a time vector input.

The next example converts our original signal to a uniform 44.1 kHz rate.

```
desiredFs = 44100;
[y, Ty] = resample(x,irregTx,desiredFs);
```

```
plot(irregTx,x,'.-', Ty,y,'o-')
legend('Original','Resampled')
ylim([-1.2 1.2])
```



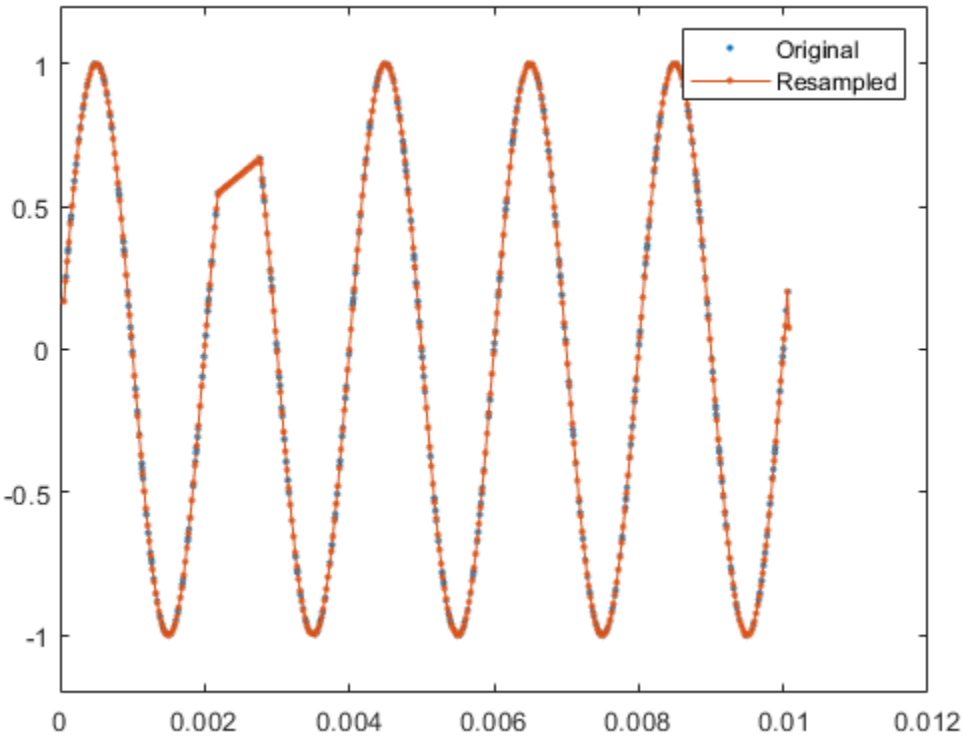

You can see that our resampled signal has the same shape and size as the original signal.

### Choosing an Interpolation Method

The conversion algorithm in `resample` works best when the input samples are as close to regularly spaced as possible, so it is instructive to observe what may happen when a section of the input samples is missing from the sampled data.

The next example notches out the second crest of the input sinusoid and apply resampling.

```
irregTx(105:130) = [];
x = sin(2*pi*f*irregTx);
[y, Ty] = resample(x,irregTx,desiredFs);

plot(irregTx,x,'. ')
hold on
plot(Ty,y,'.-')
hold off
legend('Original','Resampled')
ylim([-1.2 1.2])
```

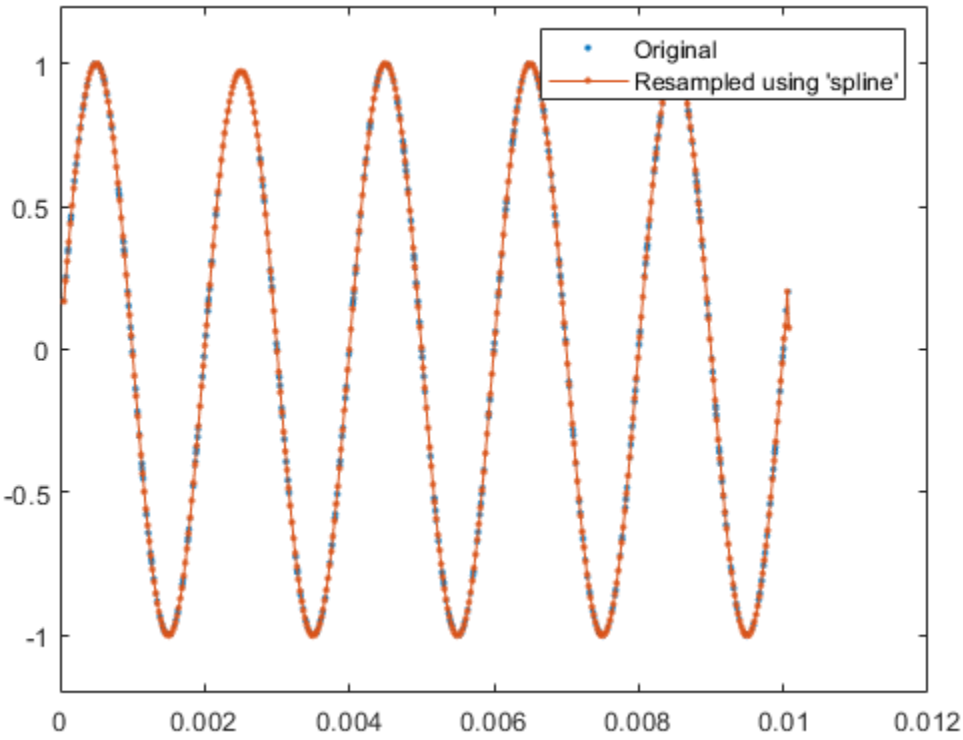



The missing segment is connected by linear interpolation. Linear interpolation is the default method used by the `resample` function to resample nonuniformly sampled data.

In some cases where you have missing data or large gaps in your input, you can recover some of the missing data by choosing a different interpolation method.

For low-noise, low-bandwidth signals, splines can be very effective when used to reconstruct the original signal. To use a cubic spline during resampling, supply the 'spline' interpolation method:

```
[y, Ty] = resample(x,irregTx,desiredFs,'spline');

plot(irregTx,x,'. ')
hold on
plot(Ty,y,'.-')
hold off
legend('Original','Resampled using ''spline'''')
ylim([-1.2 1.2])
```



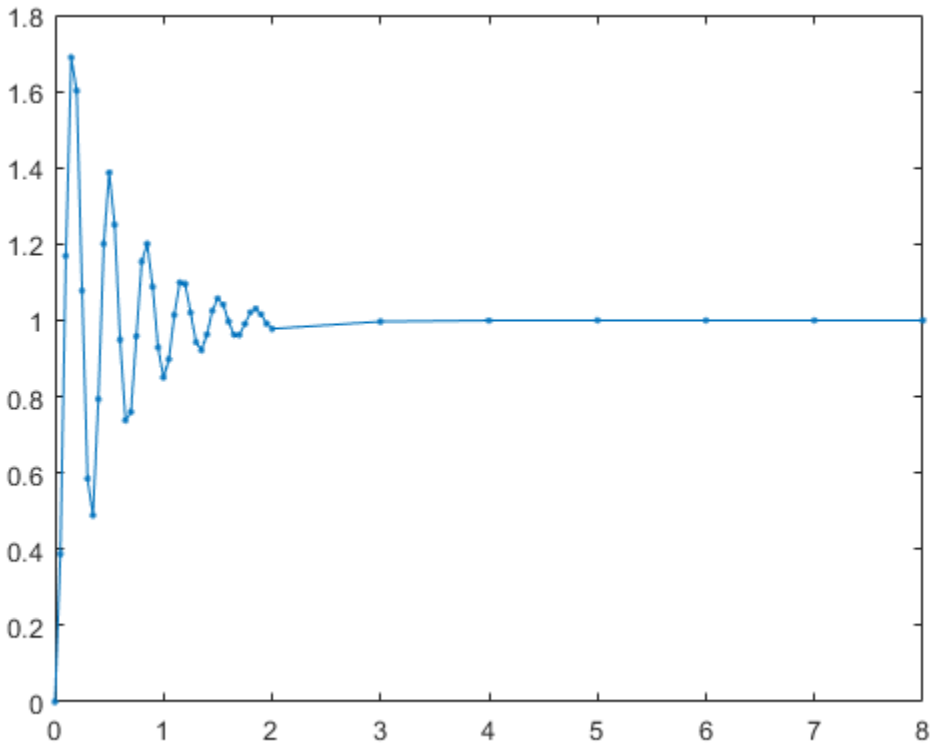

### Controlling the Interpolation Grid

By default, `resample` constructs an intermediate grid that is a close rational approximation of the ratio between the desired sample rate and the average sample rate of the signal.

If a section of your input samples contain high-frequency components, you can control the spacing of the intermediate grid by choosing integer coefficients, `p` and `q`, to select this rational ratio.

Examine the step response of an underdampened second order filter that oscillates at a rate of about 3 Hz:

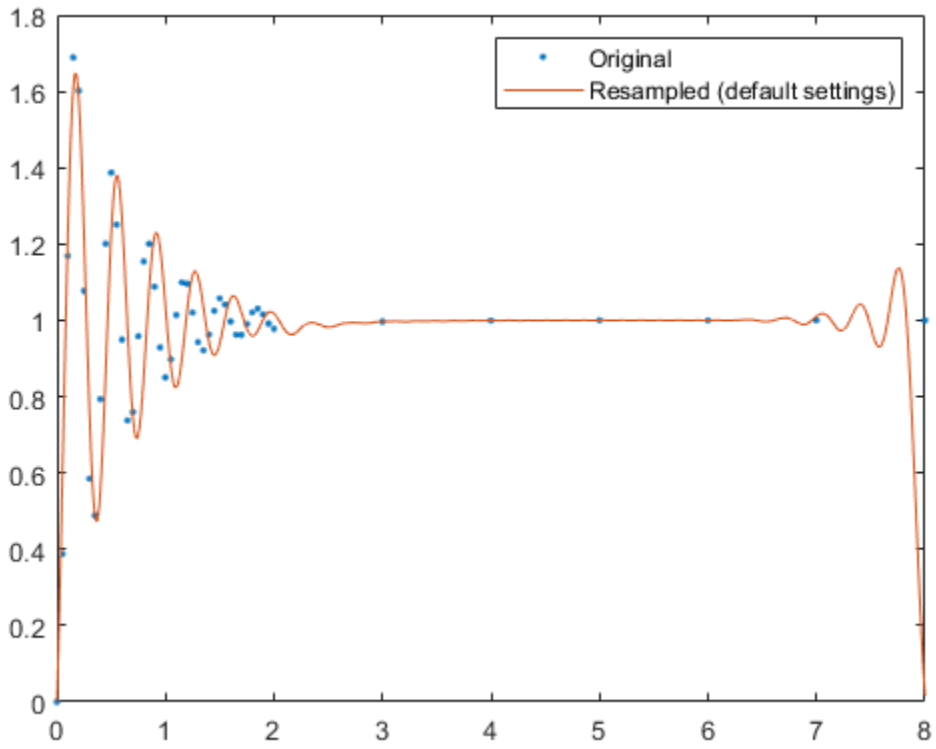
```
w = 2*pi*3;
d = .1002;
z = sin(d);
a = cos(d);

t = [0:0.05:2 3:8];

x = 1 - exp(-z*w*t).*cos(w*a*t-d)/a;
plot(t,x,'.-')
```




The step response is sampled at a high rate where it is oscillating and at a low rate where it is not.

Now resample the signal at 100 Hz just using the default settings:

```
Fs = 100;
[y, Ty] = resample(x,t,Fs);
plot(t,x,'. ')
hold on
plot(Ty,y)
hold off
legend('Original','Resampled (default settings)')
```




The envelope of the oscillation at the start of the waveform is attenuated and oscillates more slowly than the original signal.

`resample`, by default, interpolates to a grid of regularly spaced intervals that correspond to the average sample rate of the input signal.

```
avgFs = (numel(t)-1) /(t(end)-t(1))
```

```
avgFs = 5.7500
```

The sample rate of the grid should be higher than twice the largest frequency that you wish to measure. The sample rate of the grid, 5.75 samples per second, is below the Nyquist sample rate, 6 Hz, of the ringing frequency.

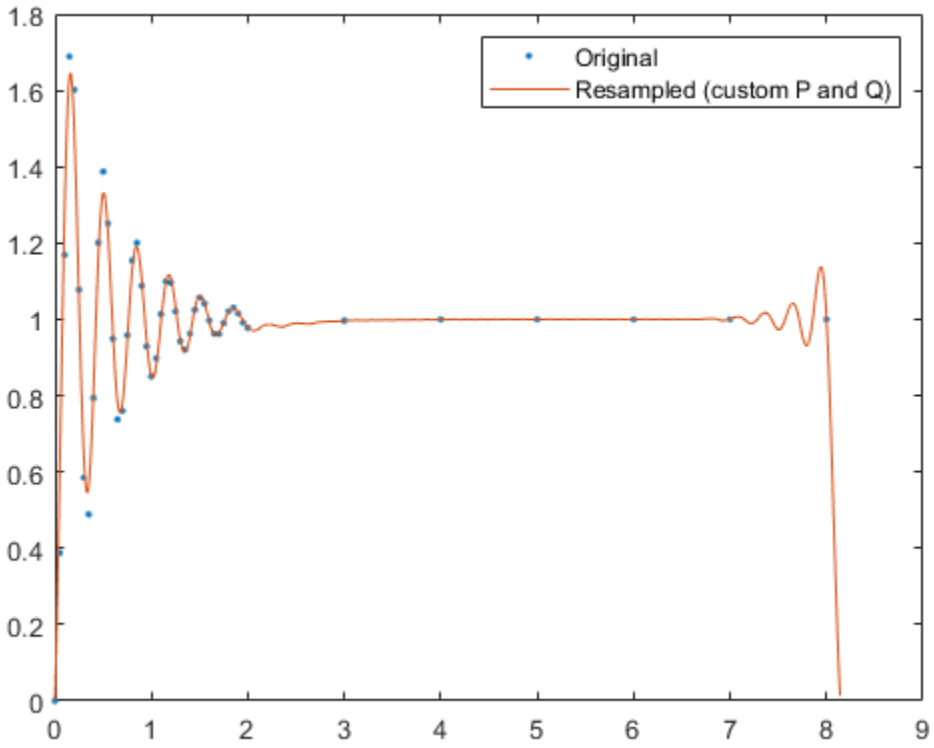
To make the grid have a higher sample rate, you can supply the integer parameters, `p` and `q`. `resample` adjusts the sample rate of the grid to Q*Fs/P, interpolates the signal, and then applies its internal sample rate converter (upsampling by P and downsampling by Q) to recover the desired sample rate, Fs. Use `rat` to select `p` and `q`.

Since the ringing of the oscillation is 3 Hz, specify the grid with a sample rate of 7 Hz, which is a little higher than the Nyquist rate. The headroom of 1 Hz accounts for additional frequency content due to the decaying exponential envelope.

```
Fgrid = 7;
[p,q] = rat(Fs/Fgrid)
```

```
p = 100
```

```
q = 7

[y, Ty] = resample(x,t,Fs,p,q);
plot(t,x,'.')
hold on
plot(Ty,y)
hold off
legend('Original','Resampled (custom P and Q)')
```

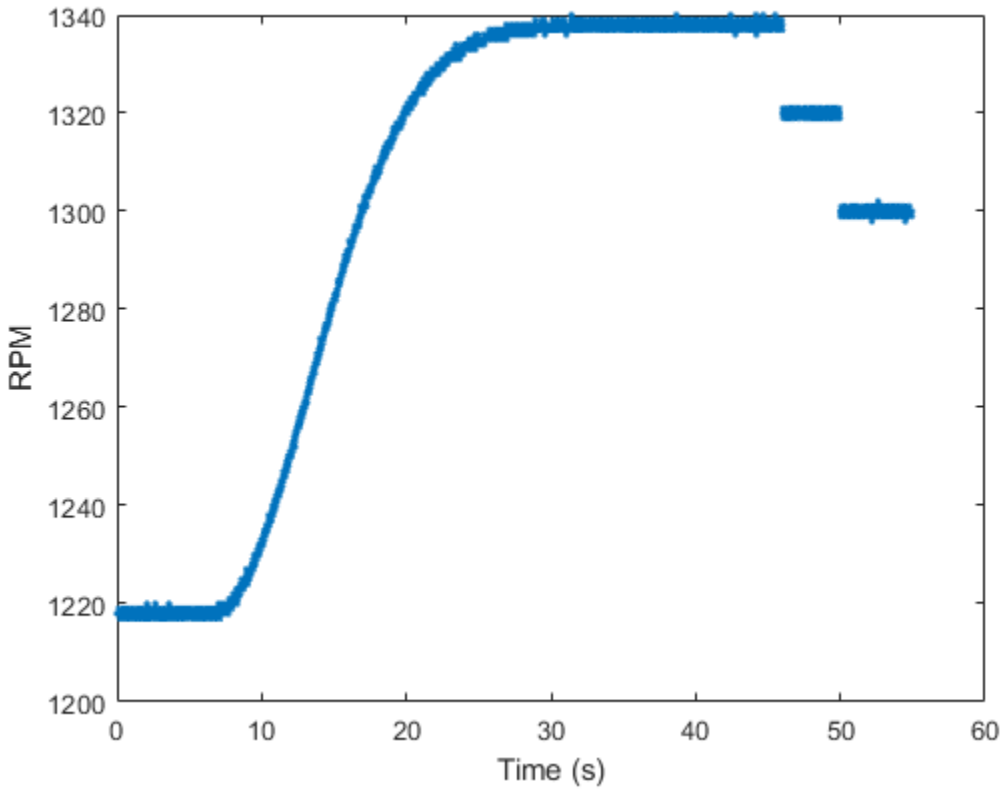



### Specifying the Anti-Aliasing Filter

In the next example, you can view the output of a digitizer that measures the throttle setting on an airplane engine. The throttle setting is nonuniformly sampled about a nominal rate of 100 Hz. We will try to resample this signal at a uniform 10 Hz rate.

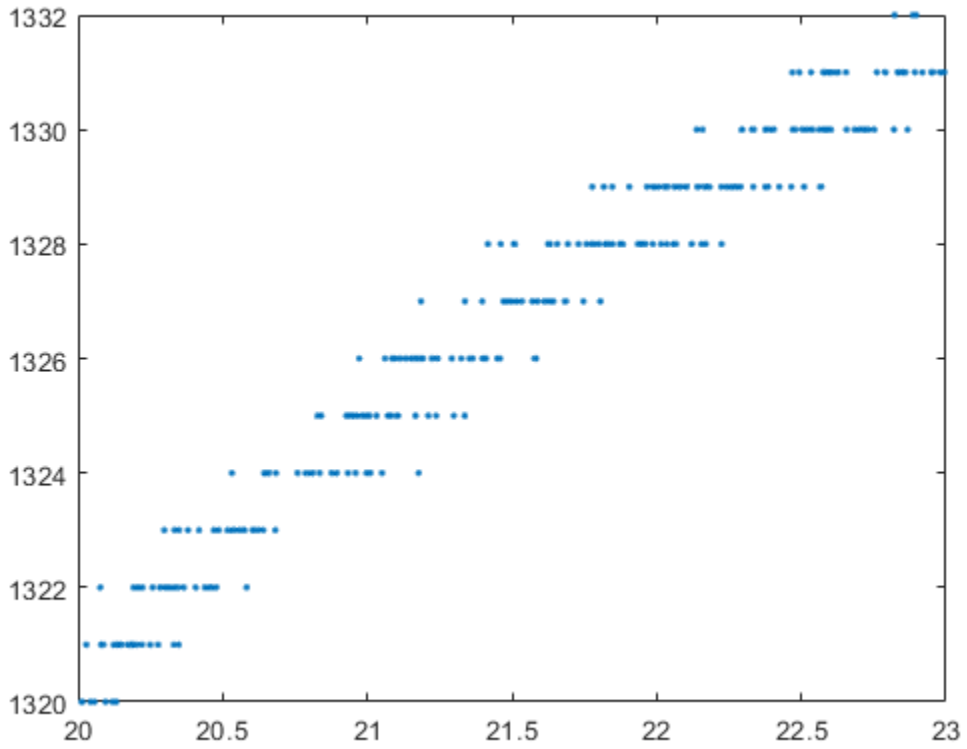
Here are the samples of our original signal.

```
load engineRPM
plot(t,x,'.')
xlabel('Time (s)')
ylabel('RPM')
```



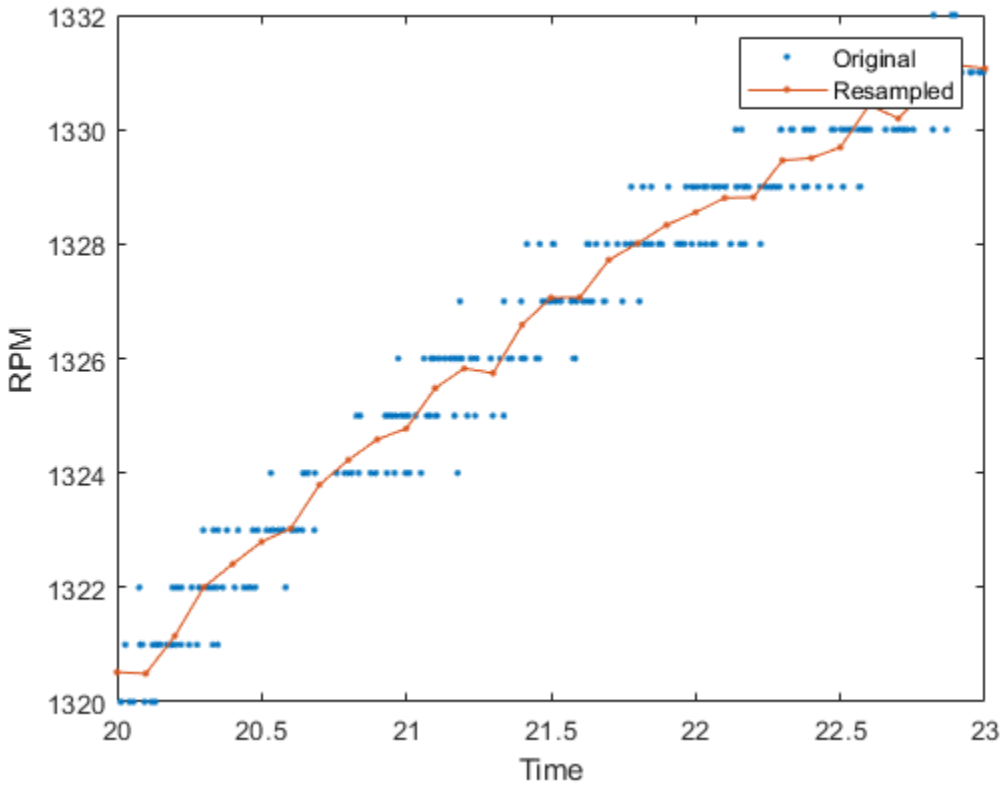

Our signal is quantized. Now zoom into the ascending region in the time interval from 20 seconds to 23 seconds:

```
plot(t,x,'.')
xlim([20 23])
```




The signal is slowly varying within this region. This allows you to remove some of the quantization noise by using the anti-aliasing filter in the resampler.

```
desiredFs = 10;
[y,ty] = resample(x,t,desiredFs);

plot(t,x,'.')
hold on
plot(ty,y,'.-')
hold off
xlabel('Time')
ylabel('RPM')
legend('Original','Resampled')
xlim([20 23])
```




This works reasonably well. However, the resampled signal can be smoothed further by providing `resample` a filter with a low cutoff frequency.

First, set the grid spacing to be about our nominal 100 Hz sample rate.

```
nominalFs = 100;
```

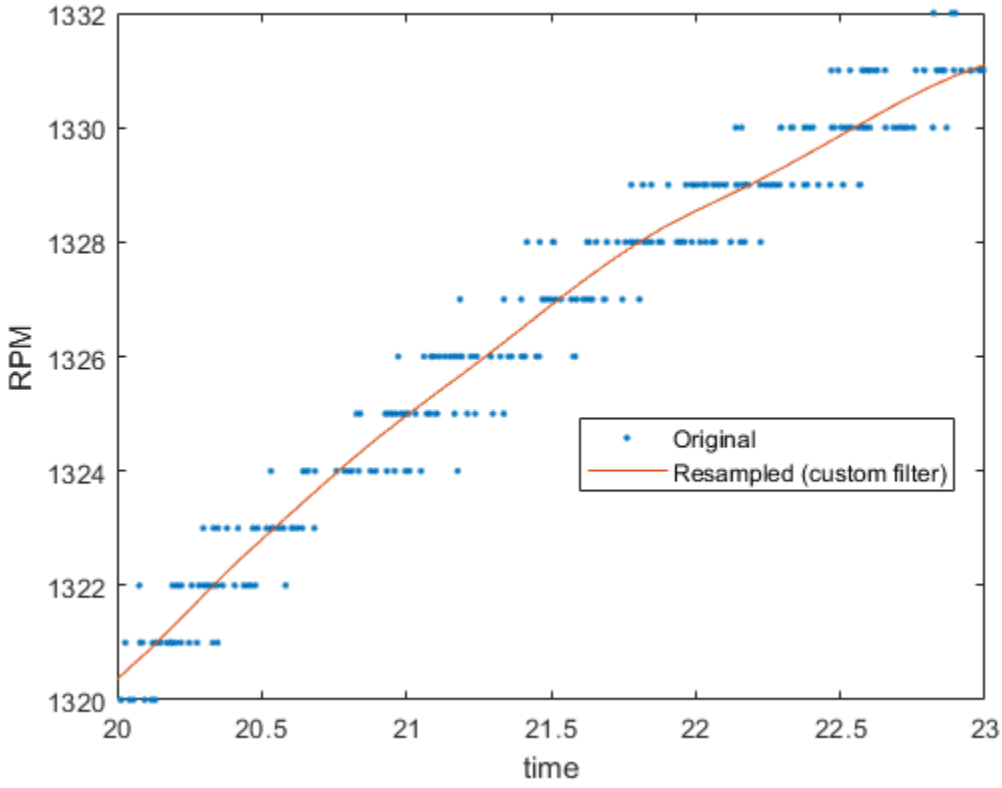
Next, determine a reasonable `p` and `q` to obtain the desired rate. Since the nominal rate is 100 Hz and our desired rate is 10 Hz, you need to decimate by 10. This is equivalent to setting `p` to 1 and setting `q` to 10.

```
p = 1;
q = 10;
```

You can supply `resample` with your own filter. To have proper temporal alignment, the filter should be of odd length. The filter length should be a few times larger than `p` or `q` (whichever is larger). Set the cutoff frequency to be 1 / `q` desired cutoff and then multiply the resulting coefficients by `p`.

```
% ensure an odd length filter
n = 10*q+1;

% use .25 of Nyquist range of desired sample rate
cutoffRatio = .25;

% construct lowpass filter
lpFilt = p * fir1(n, cutoffRatio * 1/q);
```

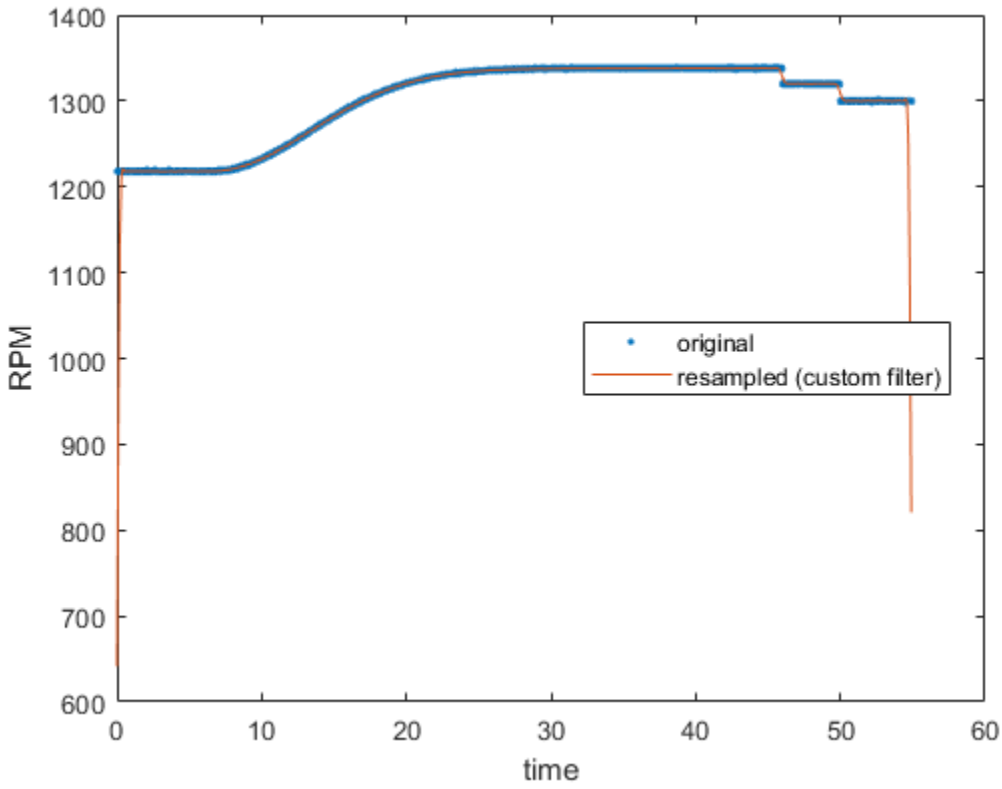
```
% resample and plot the response
[y,ty] = resample(x,t,desiredFs,p,q,lpFilt);

plot(t,x,'.')
hold on
plot(ty,y)
hold off
xlabel('time')
ylabel('RPM')
legend('Original','Resampled (custom filter)','Location','best')
xlim([20 23])
```




### Removing Endpoint Effects

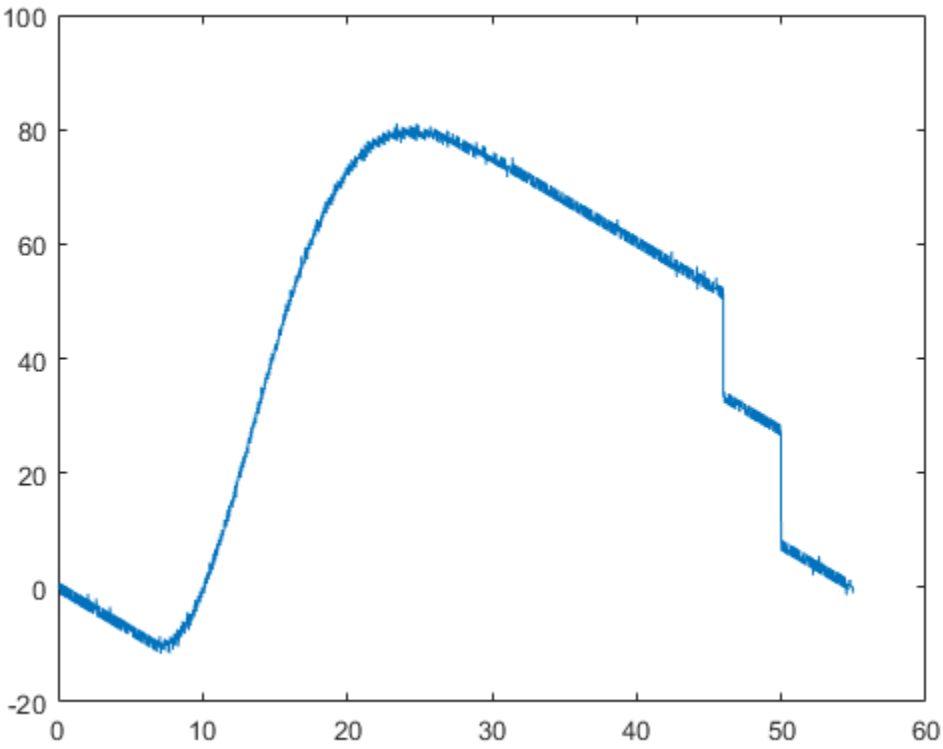
Now zoom out to view our original signal. Note that there is significant offset at the endpoints.

```
plot(t,x,'.',ty,y)
xlabel('time')
ylabel('RPM')
legend('original','resampled (custom filter)','Location','best')
```



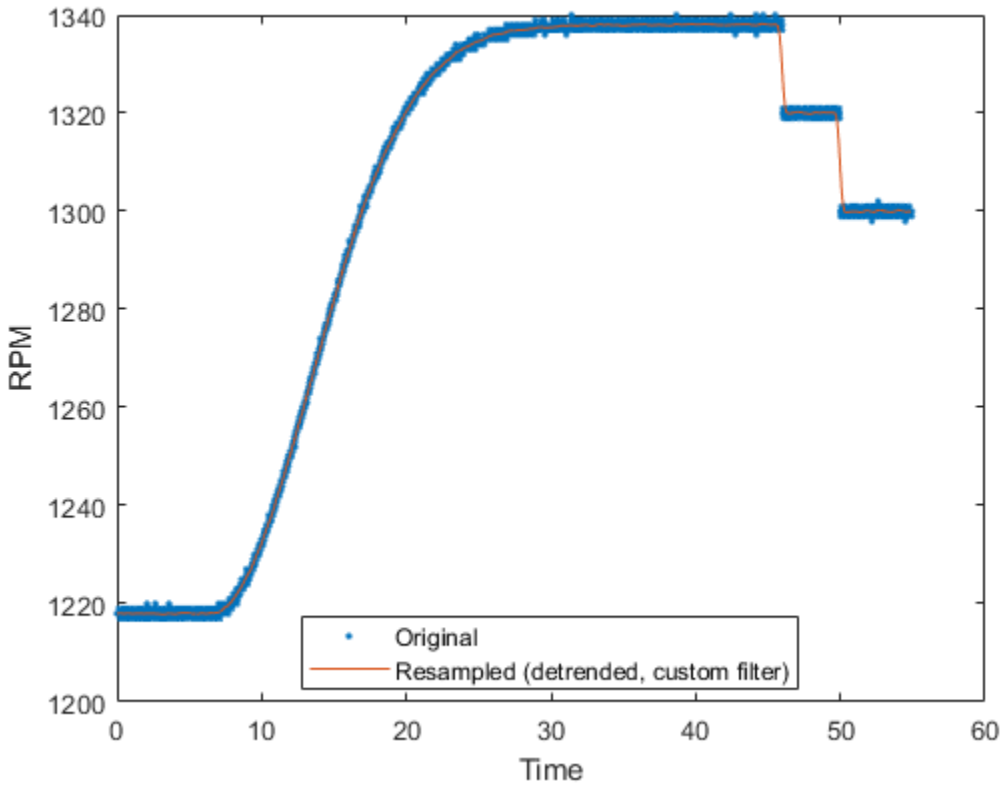

These artifacts arise because `resample` assumes that the signal is zero outside the borders of the signal. To reduce the effect of these discontinuities, subtract off a line between the endpoints of the signal, perform resampling, and then add back the line to the original function. You can do this by computing the slope and the offset of the line between the first and last sample, and using `polyval` to construct the line to subtract.

```
% compute slope and offset (y = a1 x + a2)
a(1) = (x(end)-x(1)) / (t(end)-t(1));
a(2) = x(1);

% detrend the signal
xdetrend = x - polyval(a,t);
plot(t,xdetrend)
```




The detrended signal now has both of its endpoints near zero, which reduces the transients introduced . Call `resample` and then add back the trend.

```
[ydetrend,ty] = resample(xdetrend,t,desiredFs,p,q,lpFilt);

y = ydetrend + polyval(a,ty);

plot(t,x,'.',ty,y)
xlabel('Time')
ylabel('RPM')
legend('Original','Resampled (detrended, custom filter)','Location','best')
```




### Summary

This example shows how to use `resample` to convert uniformly and nonuniformly sampled signals to a fixed rate.

### Further Reading

For more information on reconstructing nonuniformly spaced samples with custom splines, you can consult the Curve Fitting Toolbox™ documentation.

## See Also

`resample`

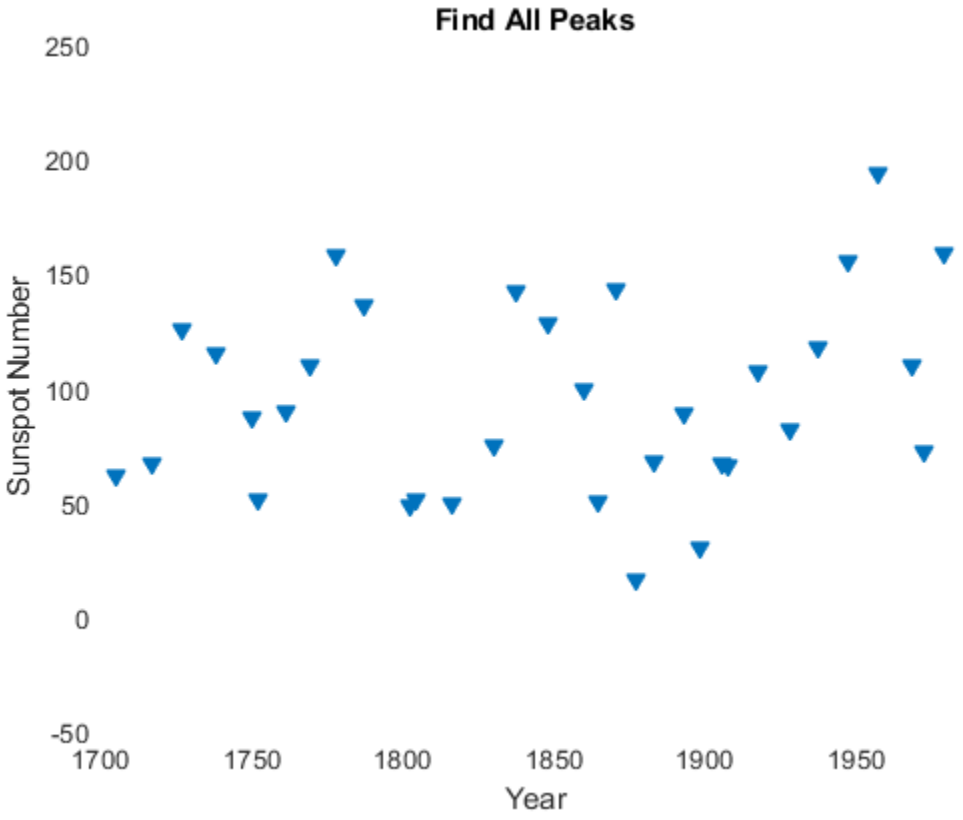
# Peak Analysis

This example shows how to perform basic peak analysis. It will help you answer questions such as: How do I find peaks in my signal? How do I measure distance between peaks? How do I measure the amplitude of peaks of a signal which is affected by a trend? How do I find peaks in a noisy signal? How do I find local minima?

### Finding Maxima or Peaks

The Zurich sunspot relative number measures both the number and size of sunspots. Use the `findpeaks` function to find the locations and the value of the peaks.

```
load sunspot.dat
year = sunspot(:,1);
relNums = sunspot(:,2);
findpeaks(relNums,year)
xlabel('Year')
ylabel('Sunspot Number')
title('Find All Peaks')
```



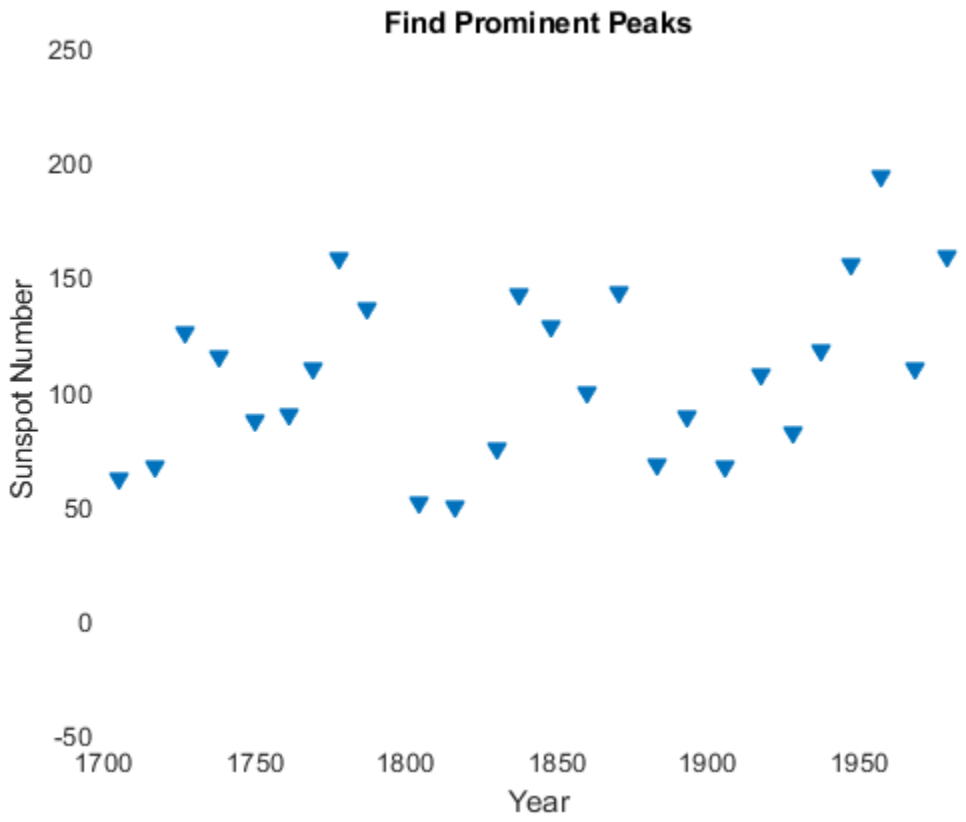

The above plot shows sunspot numbers tabulated over 300 years and labels the detected peaks. The next section shows how to measure distance between these peaks.

### Measuring Distance Between Peaks

Peaks in the signal seem to appear at regular intervals. However, some of the peaks are very close to each other. The `MinPeakProminence` property can be used filter out these peaks. Consider peaks

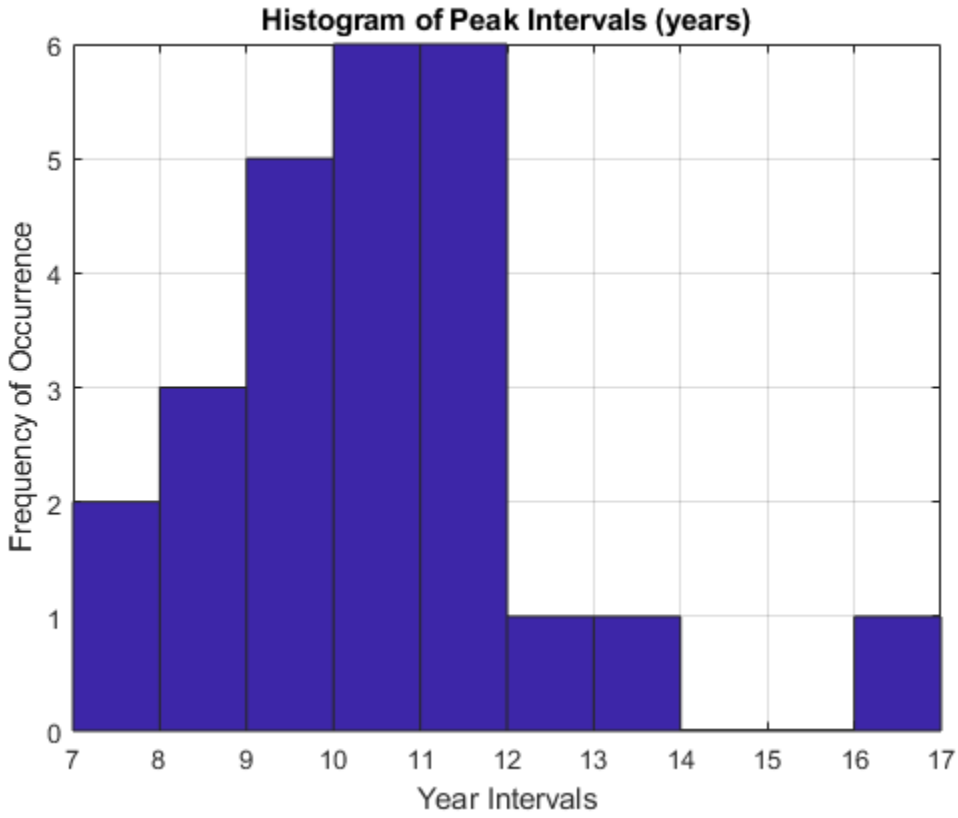
that drop off on both sides by at least 40 relative sunspot numbers before encountering a larger value.

```
findpeaks(relNums,year,'MinPeakProminence',40)
xlabel('Year')
ylabel('Sunspot Number')
title('Find Prominent Peaks')
```




The following histogram shows the distribution of peak intervals in years:

```
figure
[pks, locs] = findpeaks(relNums,year,'MinPeakProminence',40);
peakInterval = diff(locs);
hist(peakInterval)
grid on
xlabel('Year Intervals')
ylabel('Frequency of Occurrence')
title('Histogram of Peak Intervals (years)')
```




```
AverageDistance_Peaks = mean(diff(locs))
```

```
AverageDistance_Peaks = 10.9600
```

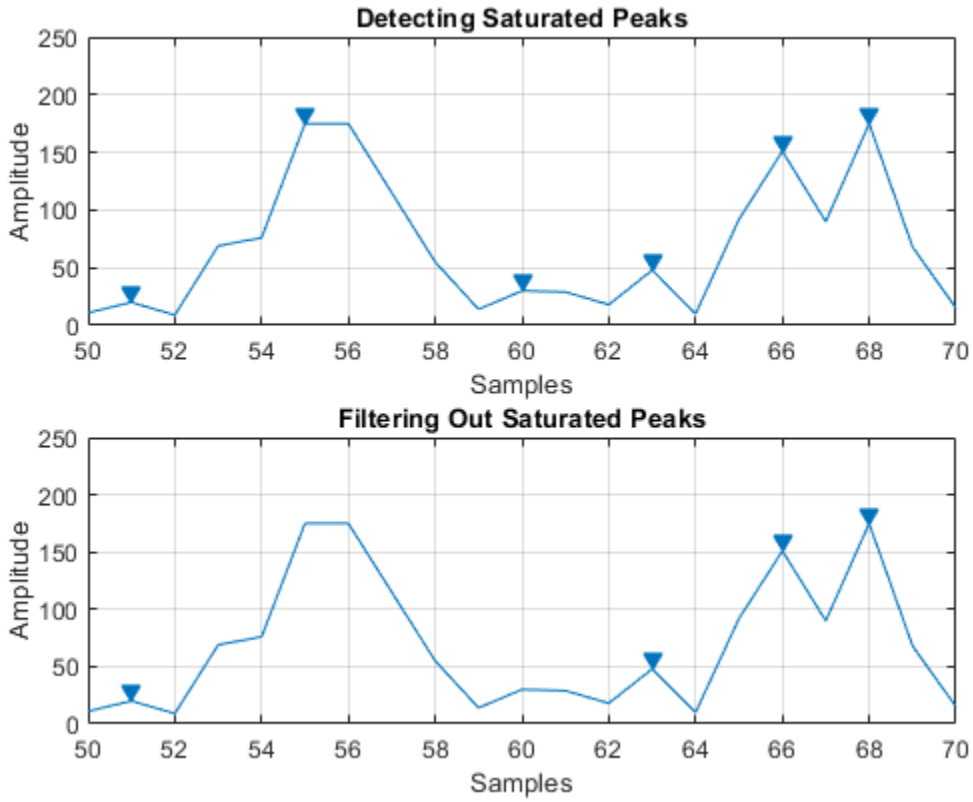
The distribution shows that majority of peak intervals lie between 10 and 12 years indicating the signal has a cyclic nature. Also, the average interval of 10.96 years between the peaks matches the known cyclic sunspot activity of 11 years.

**Finding Peaks in Clipped or Saturated Signals**

You may want to consider flat peaks as peaks or exclude them. In the latter case, a minimum excursion which is defined as the amplitude difference between a peak and its immediate neighbors is specified using the `threshold` property.

```
load clippedpeaks.mat

figure

% Show all peaks in the first plot
ax(1) = subplot(2,1,1);
findpeaks(saturatedData)
xlabel('Samples')
ylabel('Amplitude')
title('Detecting Saturated Peaks')

% Specify a minimum excursion in the second plot
```

```
ax(2) = subplot(2,1,2);
findpeaks(saturatedData,'threshold',5)
xlabel('Samples')
ylabel('Amplitude')
title('Filtering Out Saturated Peaks')

% link and zoom in to show the changes
linkaxes(ax(1:2),'xy')
axis(ax,[50 70 0 250])
```



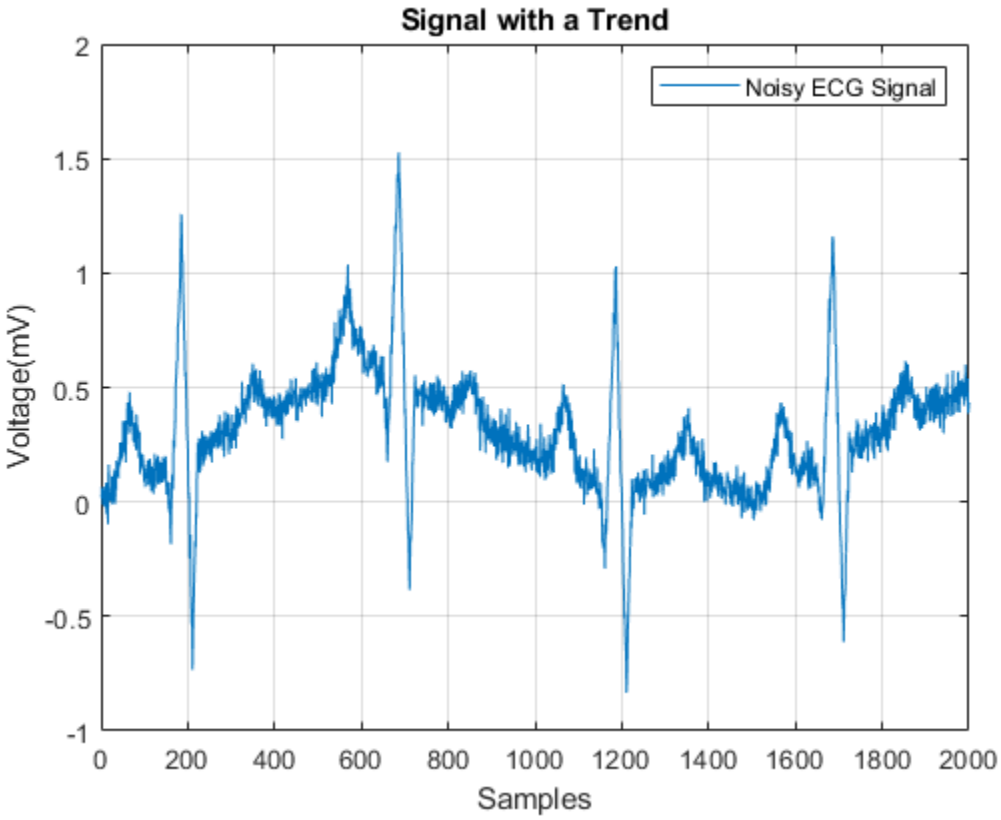

The first subplot shows, that in case of a flat peak, the rising edge is detected as the peak. The second subplot shows that specifying a threshold can help to reject flat peaks.

### Measuring Amplitudes of Peaks

This example shows peak analysis in an ECG (Electro-cardiogram) signal. ECG is a measure of electrical activity of the heart over time. The signal is measured by electrodes attached to the skin and is sensitive to disturbances such as power source interference and noises due to movement artifacts.

```
load noisyecg.mat
t = 1:length(noisyECG_withTrend);

figure
plot(t,noisyECG_withTrend)
title('Signal with a Trend')
xlabel('Samples');
```

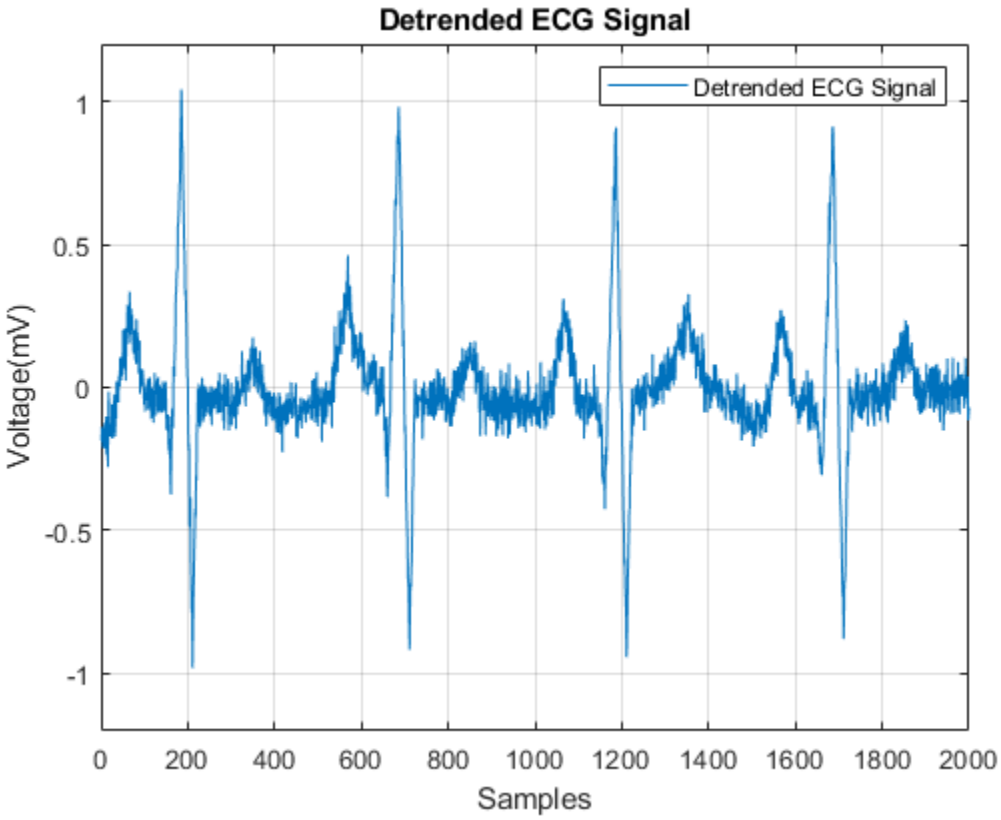
```
ylabel('Voltage(mV)')
legend('Noisy ECG Signal')
grid on
```



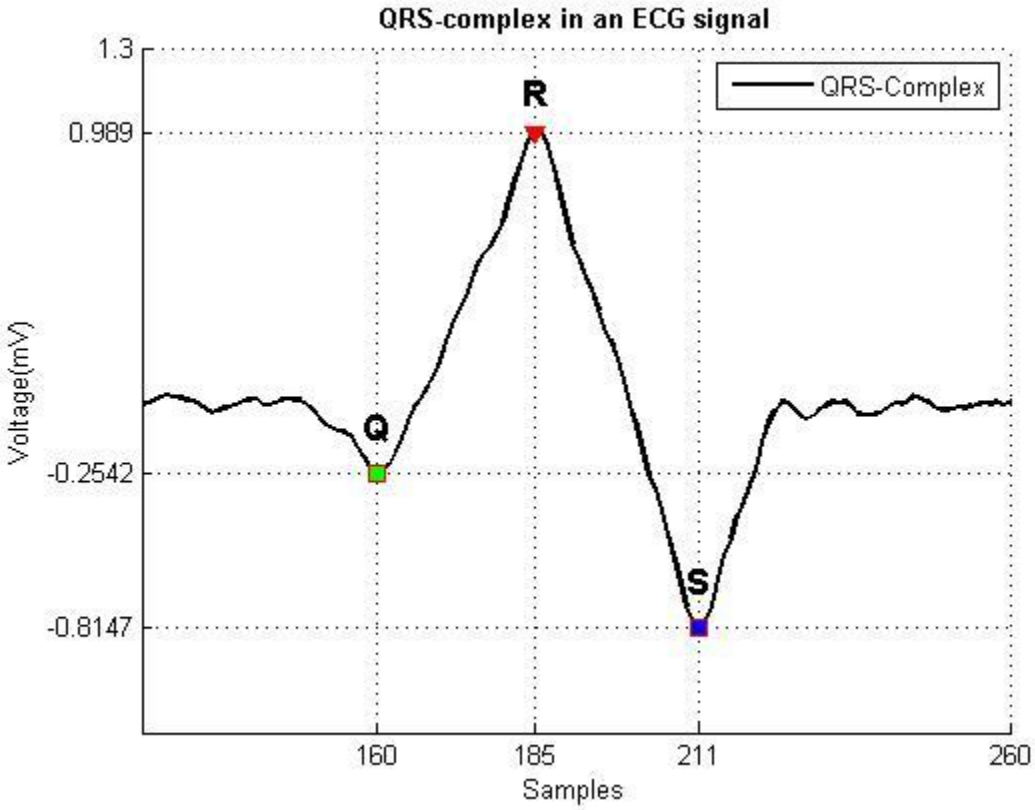

### Detrending Data

The above signal shows a baseline shift and therefore does not represent the true amplitude. In order to remove the trend, fit a low order polynomial to the signal and use the polynomial to detrend it.

```
[p,s,mu] = polyfit((1:numel(noisyECG_withTrend))',noisyECG_withTrend,6);
f_y = polyval(p,(1:numel(noisyECG_withTrend))',[],mu);

ECG_data = noisyECG_withTrend - f_y;        % Detrend data

figure
plot(t,ECG_data)
grid on
ax = axis;
axis([ax(1:2) -1.2 1.2])
title('Detrended ECG Signal')
xlabel('Samples')
ylabel('Voltage(mV)')
legend('Detrended ECG Signal')
```




After detrending, find the QRS complex, which is the most prominent repeating peak in the ECG signal. The QRS complex corresponds to the depolarization of the right and left ventricles of the human heart. It can be used to determine a patient's cardiac rate or predict abnormalities in heart function. The following figure shows the shape of the QRS complex in an ECG signal.




### Thresholding to Find Peaks of Interest

The QRS complex consists of three major components: **Q wave, R wave, S wave**. The R waves can be detected by thresholding peaks above 0.5 mV. Notice that the R waves are separated by more than 200 samples. Use this information to remove unwanted peaks by specifying a 'MinPeakDistance'.

```
[~,locs_Rwave] = findpeaks(ECG_data,'MinPeakHeight',0.5,...
                                    'MinPeakDistance',200);
```

For detection of the S-waves, find the local minima in the signal and apply thresholds appropriately.

### Finding Local Minima in Signal

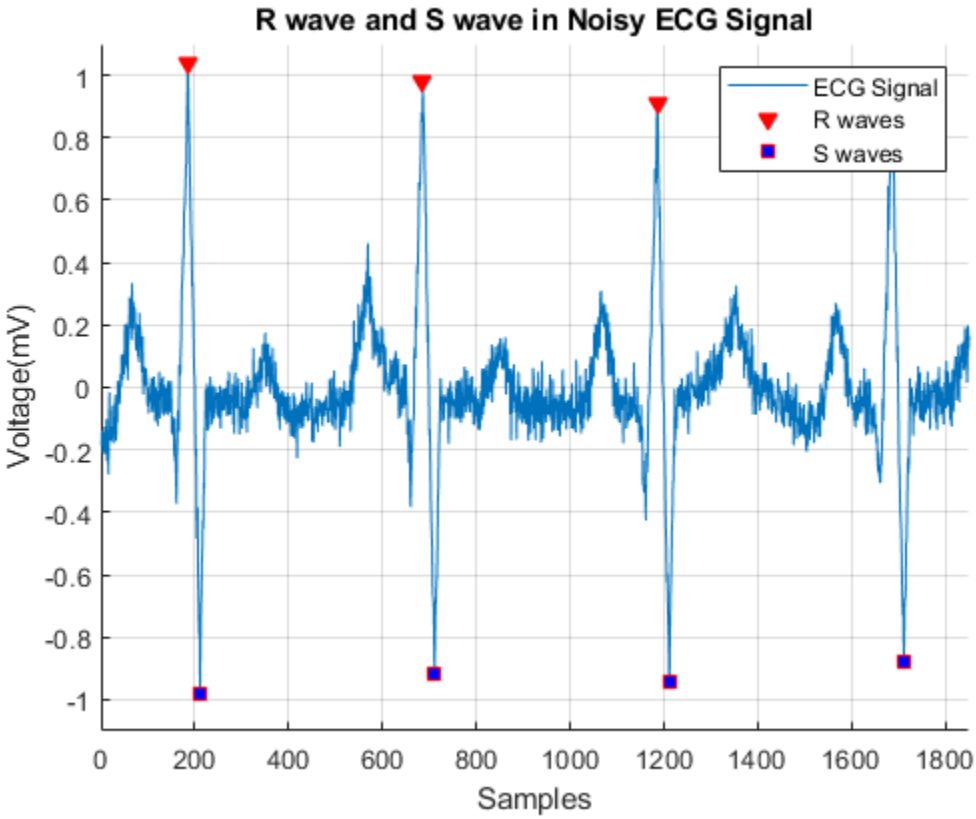
Local minima can be detected by finding peaks on an inverted version of the original signal.

```
ECG_inverted = -ECG_data;
[~,locs_Swave] = findpeaks(ECG_inverted,'MinPeakHeight',0.5,...
                                        'MinPeakDistance',200);
```

The following plot shows the R waves and S waves detected in the signal.

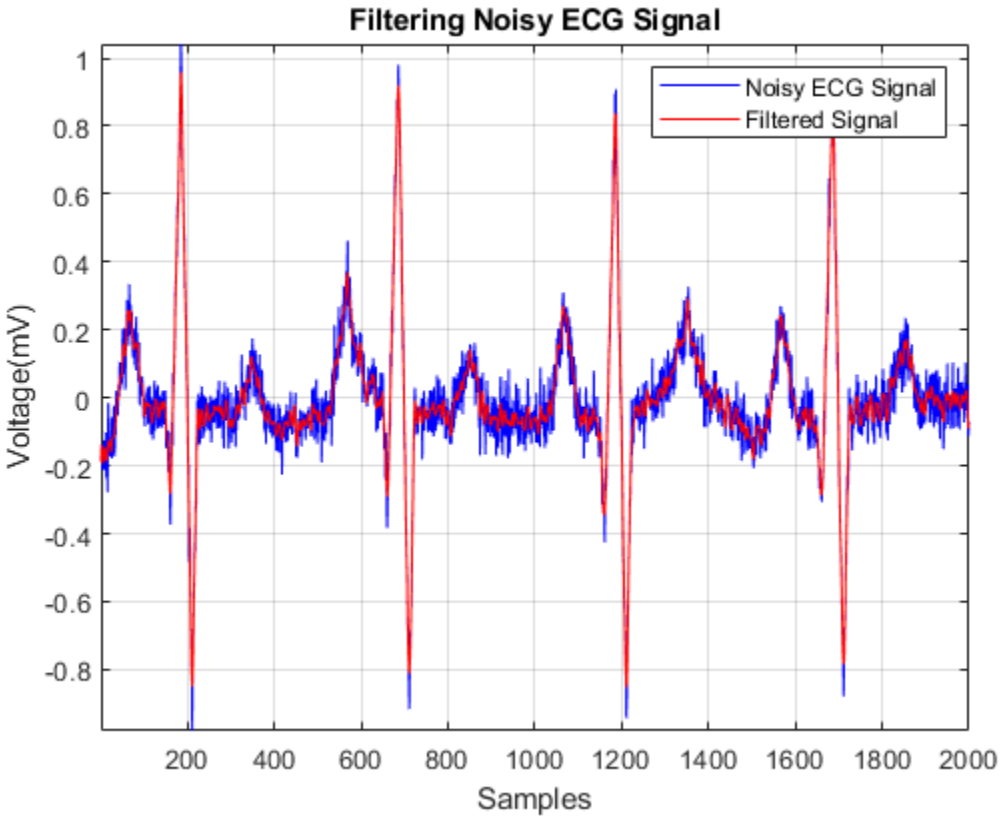
```
figure
hold on
plot(t,ECG_data)
plot(locs_Rwave,ECG_data(locs_Rwave),'rv','MarkerFaceColor','r')
plot(locs_Swave,ECG_data(locs_Swave),'rs','MarkerFaceColor','b')
axis([0 1850 -1.1 1.1])
grid on
```

```
legend('ECG Signal','R waves','S waves')
xlabel('Samples')
ylabel('Voltage(mV)')
title('R wave and S wave in Noisy ECG Signal')
```



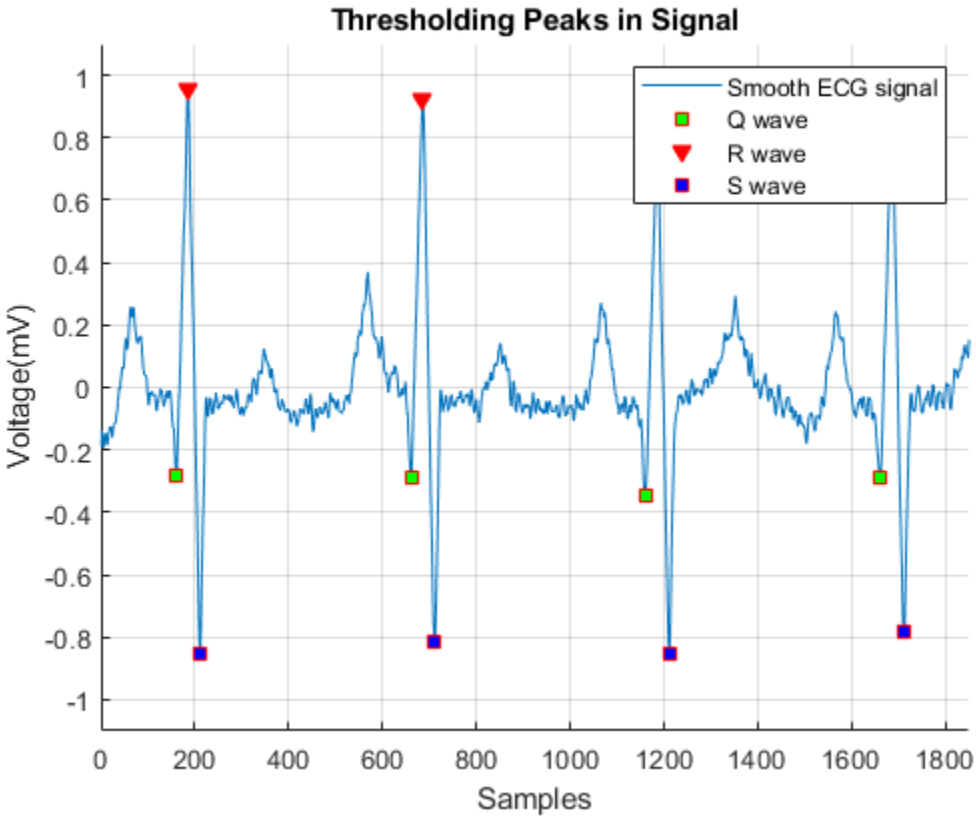

Next, we try and determine the locations of the Q waves. Thresholding the peaks to locate the Q waves results in detection of unwanted peaks as the Q waves are buried in noise. We filter the signal first and then find the peaks. Savitzky-Golay filtering is used to remove noise in the signal.

```
smoothECG = sgolayfilt(ECG_data,7,21);

figure
plot(t,ECG_data,'b',t,smoothECG,'r')
grid on
axis tight
xlabel('Samples')
ylabel('Voltage(mV)')
legend('Noisy ECG Signal','Filtered Signal')
title('Filtering Noisy ECG Signal')
```




We perform peak detection on the smooth signal and use logical indexing to find the locations of the Q waves.

```
[~,min_locs] = findpeaks(-smoothECG,'MinPeakDistance',40);

% Peaks between -0.2mV and -0.5mV
locs_Qwave = min_locs(smoothECG(min_locs)>-0.5 & smoothECG(min_locs)<-0.2);

figure
hold on
plot(t,smoothECG);
plot(locs_Qwave,smoothECG(locs_Qwave),'rs','MarkerFaceColor','g')
plot(locs_Rwave,smoothECG(locs_Rwave),'rv','MarkerFaceColor','r')
plot(locs_Swave,smoothECG(locs_Swave),'rs','MarkerFaceColor','b')
grid on
title('Thresholding Peaks in Signal')
xlabel('Samples')
ylabel('Voltage(mV)')
ax = axis;
axis([0 1850 -1.1 1.1])
legend('Smooth ECG signal','Q wave','R wave','S wave')
```




The above figure shows that the QRS complex successfully detected in the noisy ECG signal.

**Error Between Noisy and Smooth Signal**

Notice the average difference between the QRS complex in the raw and the detrended filtered signal.

```
% Values of the Extrema
[val_Qwave, val_Rwave, val_Swave] = deal(smoothECG(locs_Qwave), smoothECG(locs_Rwave), smoothECG
```

```
meanError_Qwave = mean((noisyECG_withTrend(locs_Qwave) - val_Qwave))
```

```
meanError_Qwave = 0.2771
```

```
meanError_Rwave = mean((noisyECG_withTrend(locs_Rwave) - val_Rwave))
```

```
meanError_Rwave = 0.3476
```

```
meanError_Swave = mean((noisyECG_withTrend(locs_Swave) - val_Swave))
```

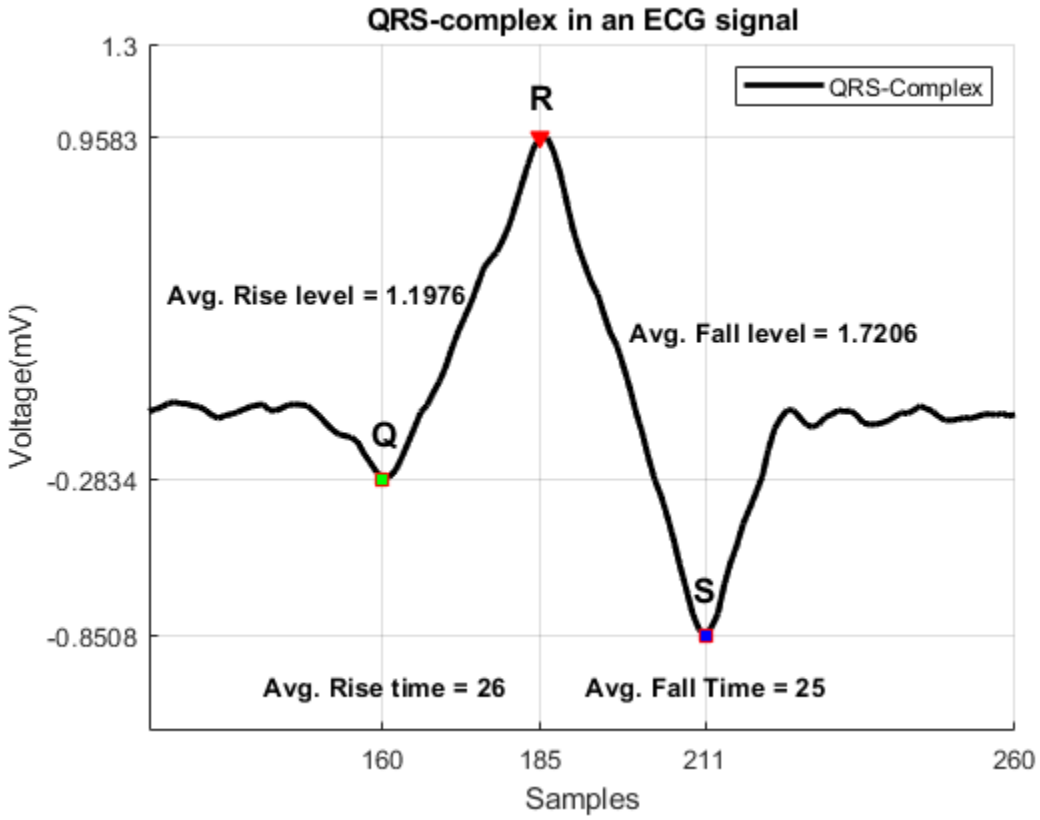
```
meanError_Swave = 0.1844
```

This demonstrates that it is essential to detrend a noisy signal for efficient peak analysis.

**Peak Properties**

Some important peak properties involve rise time, fall time, rise level, and fall level. These properties are computed for each of the QRS complexes in the ECG signal. The average values for these properties are displayed on the figure below.

```
avg_riseTime = mean(locs_Rwave-locs_Qwave); % Average Rise time
avg_fallTime = mean(locs_Swave-locs_Rwave); % Average Fall time
avg_riseLevel = mean(val_Rwave-val_Qwave);  % Average Rise Level
avg_fallLevel = mean(val_Rwave-val_Swave);  % Average Fall Level

helperPeakAnalysisPlot(t,smoothECG,...
                    locs_Qwave,locs_Rwave,locs_Swave,...
                    val_Qwave,val_Rwave,val_Swave,...
                    avg_riseTime,avg_fallTime,...
                    avg_riseLevel,avg_fallLevel)
```




## See Also

`findpeaks`

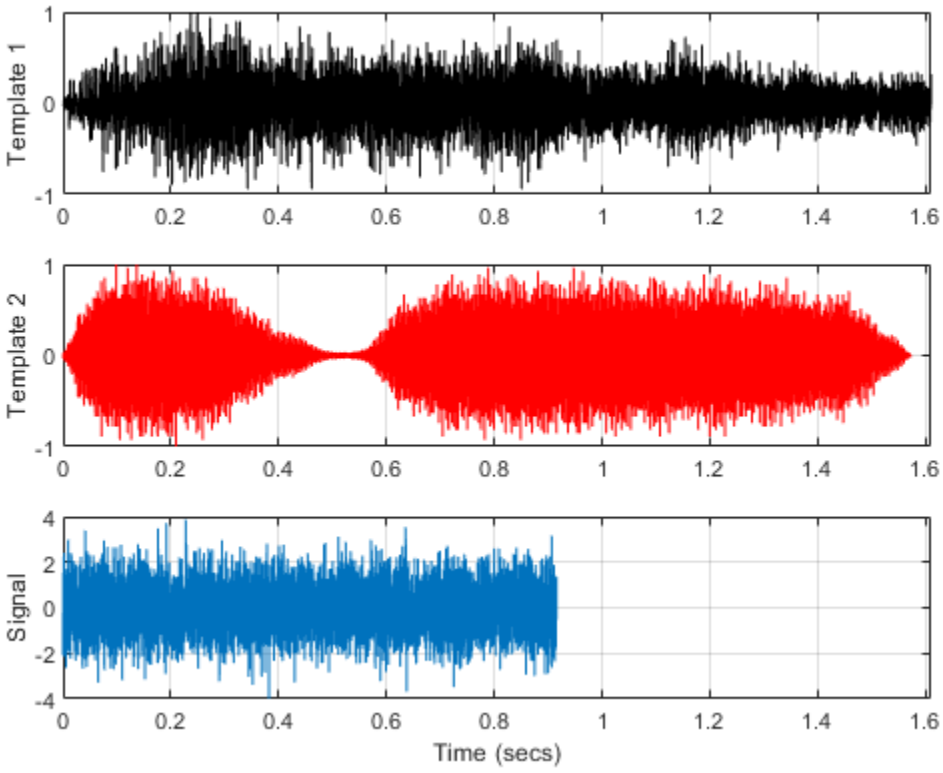
# Measuring Signal Similarities

This example shows how to measure signal similarities. It will help you answer questions such as: How do I compare signals with different lengths or different sampling rates? How do I find if there is a signal or just noise in a measurement? Are two signals related? How to measure a delay between two signals (and how do I align them)? How do I compare the frequency content of two signals? Similarities can also be found in different sections of a signal to determine if a signal is periodic.

### Comparing Signals with Different Sampling Rates

Consider a database of audio signals and a pattern matching application where you need to identify a song as it is playing. Data is commonly stored at a low sampling rate to occupy less memory.

```
% Load data
load relatedsig.mat

figure
ax(1) = subplot(3,1,1);
plot((0:numel(T1)-1)/Fs1,T1,'k')
ylabel('Template 1')
grid on
ax(2) = subplot(3,1,2);
plot((0:numel(T2)-1)/Fs2,T2,'r')
ylabel('Template 2')
grid on
ax(3) = subplot(3,1,3);
plot((0:numel(S)-1)/Fs,S)
ylabel('Signal')
grid on
xlabel('Time (secs)')
linkaxes(ax(1:3),'x')
axis([0 1.61 -4 4])
```




The first and the second subplot show the template signals from the database. The third subplot shows the signal which we want to search for in our database. Just by looking at the time series, the signal does not seem to match to any of the two templates. A closer inspection reveals that the signals actually have different lengths and sampling rates.

```
[Fs1 Fs2 Fs]
```

```
ans = 1×3

        4096        4096        8192
```

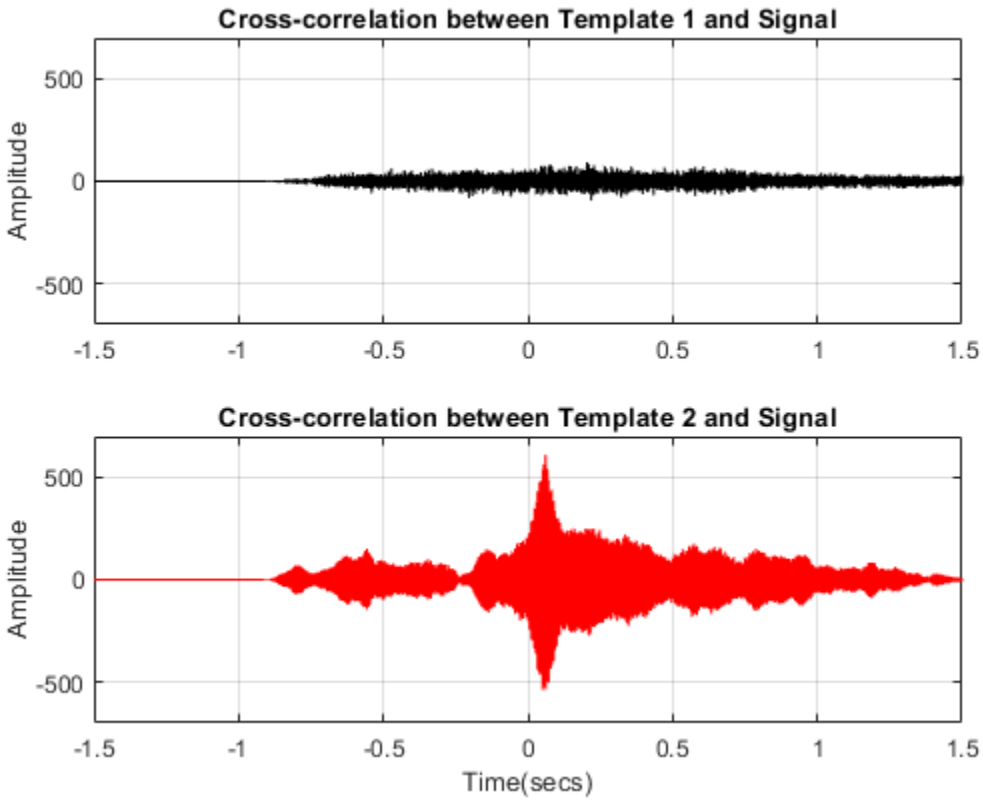
Different lengths prevent you from calculating the difference between two signals but this can easily be remedied by extracting the common part of signals. Furthermore, it is not always necessary to equalize lengths. Cross-correlation can be performed between signals with different lengths, but it is essential to ensure that they have identical sampling rates. The safest way to do this is to resample the signal with a lower sampling rate. The `resample` function applies an anti-aliasing(low-pass) FIR filter to the signal during the resampling process.

```
[P1,Q1] = rat(Fs/Fs1);          % Rational fraction approximation
[P2,Q2] = rat(Fs/Fs2);          % Rational fraction approximation
T1 = resample(T1,P1,Q1);        % Change sampling rate by rational factor
T2 = resample(T2,P2,Q2);        % Change sampling rate by rational factor
```

### Finding a Signal in a Measurement

We can now cross-correlate signal S to templates T1 and T2 with the `xcorr` function to determine if there is a match.

```
[C1,lag1] = xcorr(T1,S);
[C2,lag2] = xcorr(T2,S);

figure
ax(1) = subplot(2,1,1);
plot(lag1/Fs,C1,'k')
ylabel('Amplitude')
grid on
title('Cross-correlation between Template 1 and Signal')
ax(2) = subplot(2,1,2);
plot(lag2/Fs,C2,'r')
ylabel('Amplitude')
grid on
title('Cross-correlation between Template 2 and Signal')
xlabel('Time(secs)')
axis(ax(1:2),[-1.5 1.5 -700 700 ])
```




The first subplot indicates that the signal and template 1 are less correlated while the high peak in the second subplot indicates that signal is present in the second template.

```
[~,I] = max(abs(C2));
SampleDiff = lag2(I)
```

```
SampleDiff = 499
```

```
timeDiff = SampleDiff/Fs
```

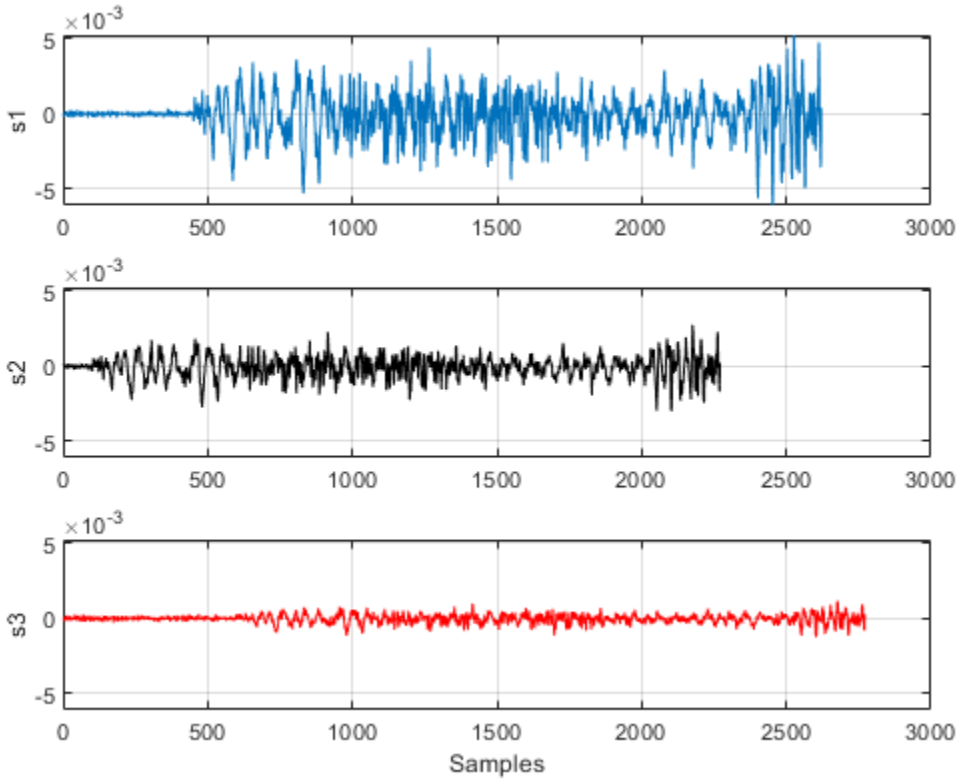
```
timeDiff = 0.0609
```

The peak of the cross correlation implies that the signal is present in template T2 starting after 61 ms. In other words, signal T2 leads signal S by 499 samples as indicated by SampleDiff. This information can be used to align the signals.

### Measuring Delay Between Signals and Aligning Them

Consider a situation where you are collecting data from different sensors, recording vibrations caused by cars on both sides of a bridge. When you analyze the signals, you may need to align them. Assume you have 3 sensors working at same sampling rates and they are measuring signals caused by the same event.

```
figure,
ax(1) = subplot(3,1,1);
plot(s1)
ylabel('s1')
grid on
ax(2) = subplot(3,1,2);
plot(s2,'k')
ylabel('s2')
grid on
ax(3) = subplot(3,1,3);
plot(s3,'r')
ylabel('s3')
grid on
xlabel('Samples')
linkaxes(ax,'xy')
```




We can also use the `finddelay` function to find the delay between two signals.

```
t21 = finddelay(s1,s2)
```

```
t21 = -350
```

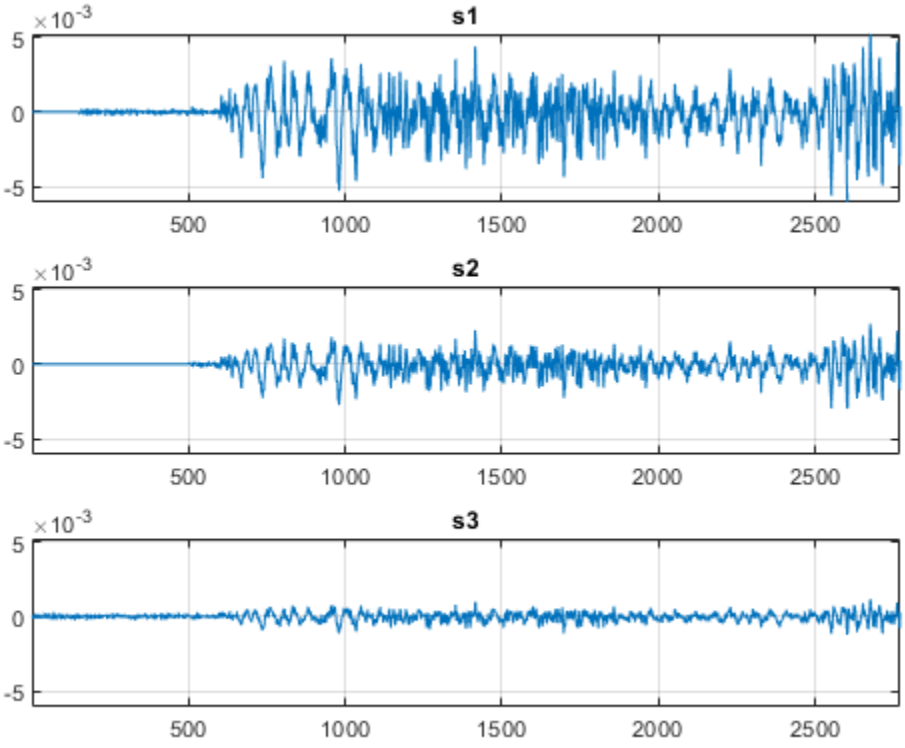
```
t31 = finddelay(s1,s3)
```

```
t31 = 150
```

t21 indicates that s2 lags s1 by 350 samples, and t31 indicates that s3 leads s1 by 150 samples. This information can now used to align the 3 signals by time shifting the signals. We can also use the `alignsignals` function directly to align the signals which aligns two signals by delaying the earliest signal.

```
s1 = alignsignals(s1,s3);
s2 = alignsignals(s2,s3);

figure
ax(1) = subplot(3,1,1);
plot(s1)
grid on
title('s1')
axis tight
ax(2) = subplot(3,1,2);
plot(s2)
grid on
title('s2')
```

```
axis tight
ax(3) = subplot(3,1,3);
plot(s3)
grid on
title('s3')
axis tight
linkaxes(ax,'xy')
```

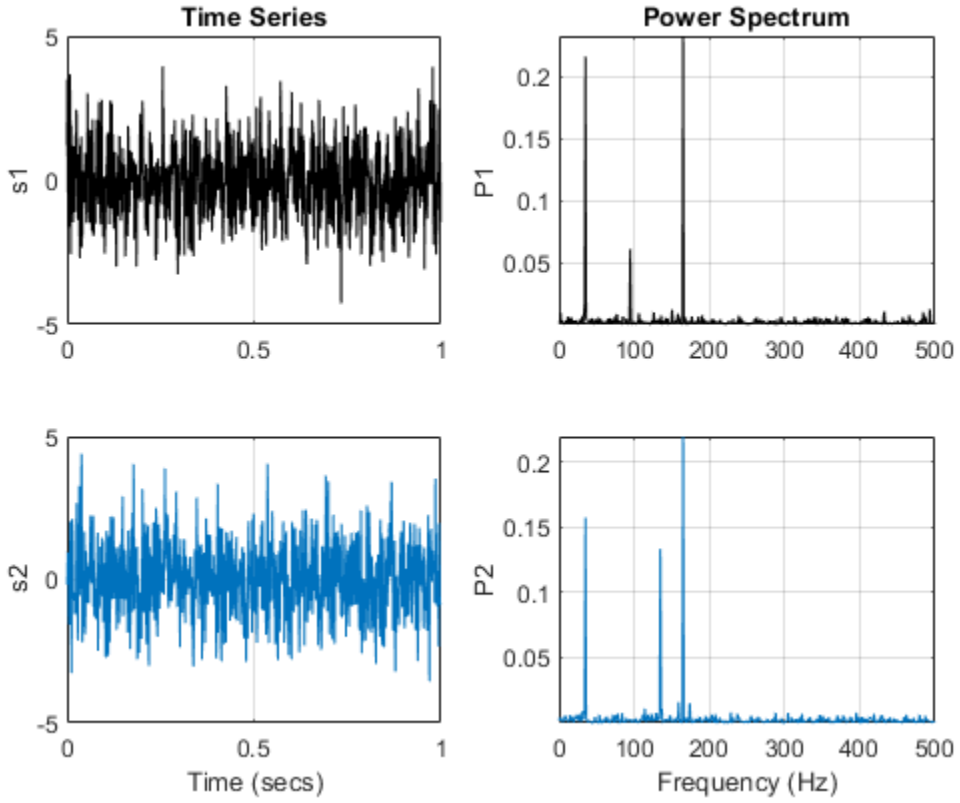



### Comparing the Frequency Content of Signals

A power spectrum displays the power present in each frequency. Spectral coherence identifies frequency-domain correlation between signals. Coherence values tending towards 0 indicate that the corresponding frequency components are uncorrelated while values tending towards 1 indicate that the corresponding frequency components are correlated. Consider two signals and their respective power spectra.

```
Fs = FsSig;          % Sampling Rate

[P1,f1] = periodogram(sig1,[],[],Fs,'power');
[P2,f2] = periodogram(sig2,[],[],Fs,'power');

figure
t = (0:numel(sig1)-1)/Fs;
subplot(2,2,1)
plot(t,sig1,'k')
ylabel('s1')
grid on
```

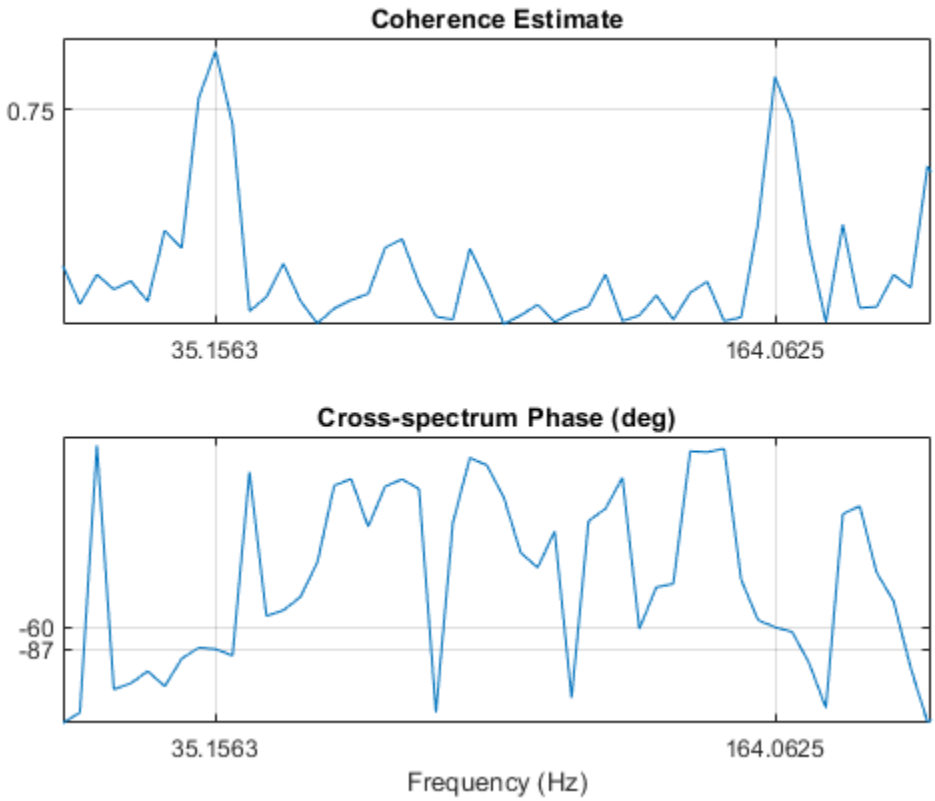
```
title('Time Series')
subplot(2,2,3)
plot(t,sig2)
ylabel('s2')
grid on
xlabel('Time (secs)')
subplot(2,2,2)
plot(f1,P1,'k')
ylabel('P1')
grid on
axis tight
title('Power Spectrum')
subplot(2,2,4)
plot(f2,P2)
ylabel('P2')
grid on
axis tight
xlabel('Frequency (Hz)')
```




The `mscohere` function calculates the spectral coherence between the two signals. It confirms that sig1 and sig2 have two correlated components around 35 Hz and 165 Hz. In frequencies where spectral coherence is high, the relative phase between the correlated components can be estimated with the cross-spectrum phase.

```
[Cxy,f] = mscohere(sig1,sig2,[],[],[],Fs);
Pxy     = cpsd(sig1,sig2,[],[],[],Fs);
phase   = -angle(Pxy)/pi*180;
```

```
[pks,locs] = findpeaks(Cxy,'MinPeakHeight',0.75);

figure
subplot(2,1,1)
plot(f,Cxy)
title('Coherence Estimate')
grid on
hgca = gca;
hgca.XTick = f(locs);
hgca.YTick = 0.75;
axis([0 200 0 1])
subplot(2,1,2)
plot(f,phase)
title('Cross-spectrum Phase (deg)')
grid on
hgca = gca;
hgca.XTick = f(locs);
hgca.YTick = round(phase(locs));
xlabel('Frequency (Hz)')
axis([0 200 -180 180])
```

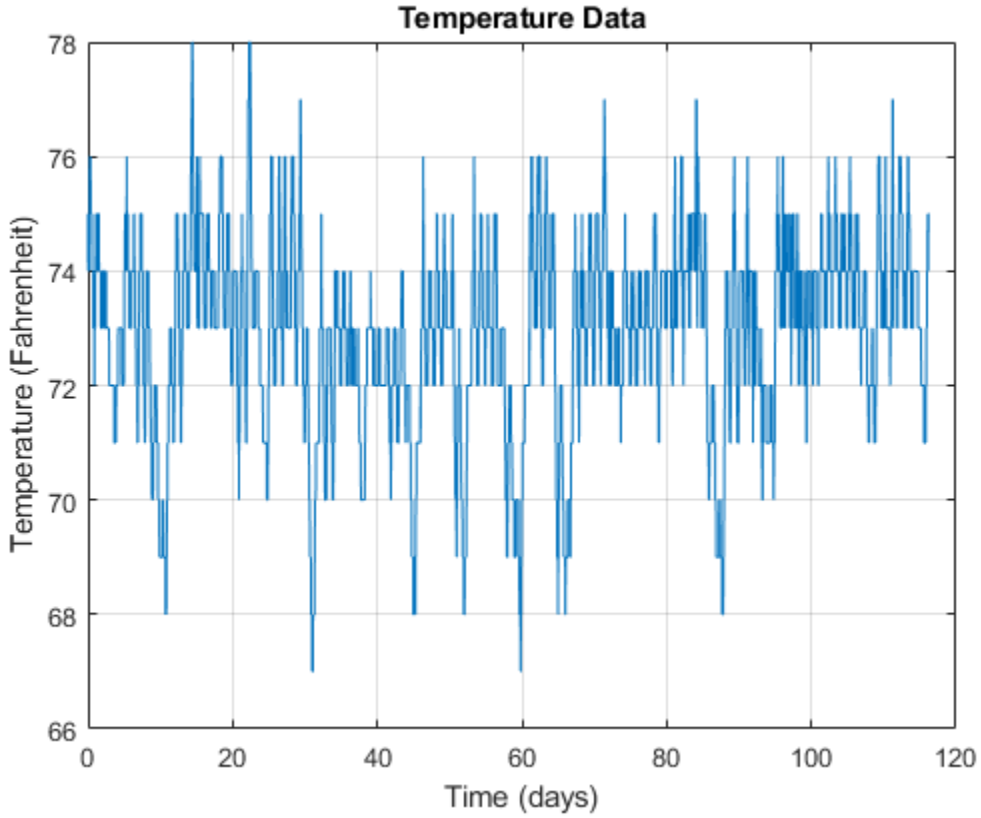



The phase lag between the 35 Hz components is close to -90 degrees, and the phase lag between the 165 Hz components is close to -60 degrees.

### Finding Periodicities in a Signal

Consider a set of temperature measurements in an office building during the winter season. Measurements were taken every 30 minutes for about 16.5 weeks.

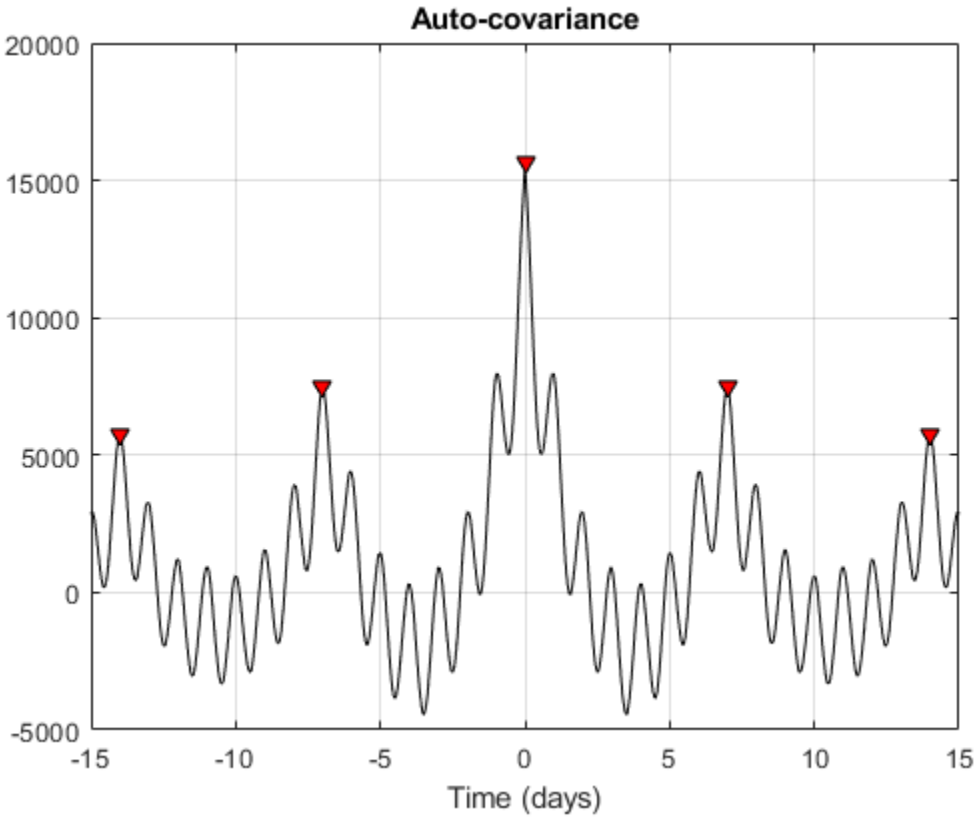
```
% Load Temperature Data
load officetemp.mat
Fs = 1/(60*30);                 % Sample rate is 1 sample every 30 minutes
days = (0:length(temp)-1)/(Fs*60*60*24);

figure
plot(days,temp)
title('Temperature Data')
xlabel('Time (days)')
ylabel('Temperature (Fahrenheit)')
grid on
```




With the temperatures in the low 70s, you need to remove the mean to analyze small fluctuations in the signal. The `xcov` function removes the mean of the signal before computing the cross-correlation. It returns the cross-covariance. Limit the maximum lag to 50% of the signal to get a good estimate of the cross-covariance.

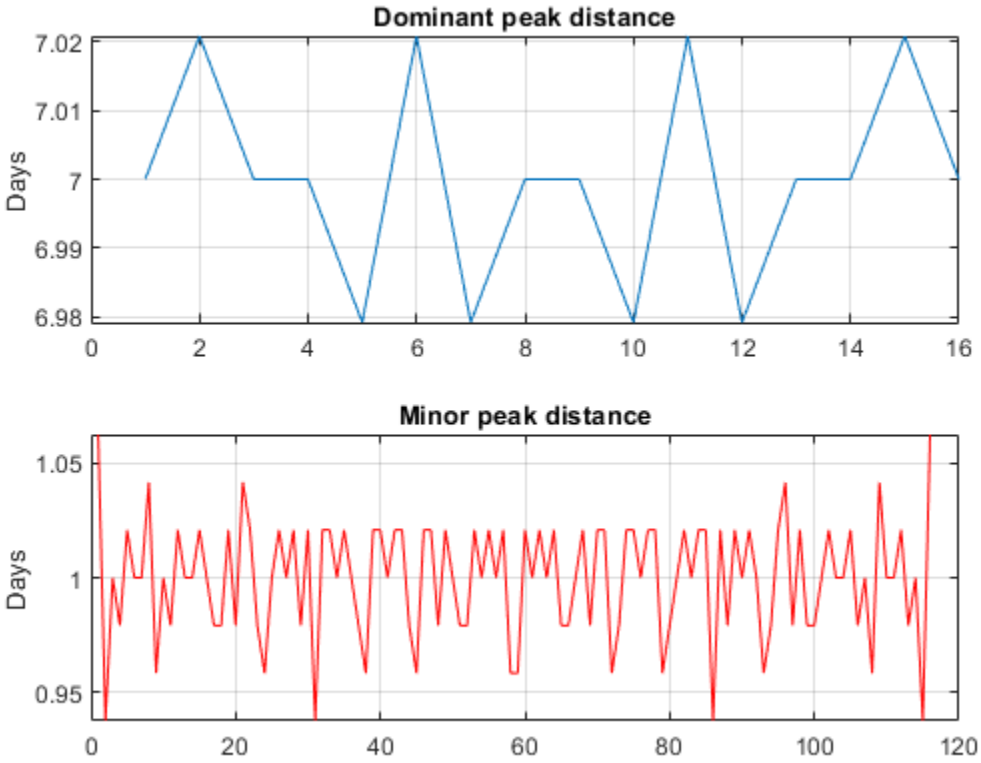
```
maxlags = numel(temp)*0.5;
[xc,lag] = xcov(temp,maxlags);

[~,df] = findpeaks(xc,'MinPeakDistance',5*2*24);
[~,mf] = findpeaks(xc);
```

```
figure
plot(lag/(2*24),xc,'k',...
     lag(df)/(2*24),xc(df),'kv','MarkerFaceColor','r')
grid on
xlim([-15 15])
xlabel('Time (days)')
title('Auto-covariance')
```




Observe dominant and minor fluctuations in the auto-covariance. Dominant and minor peaks appear equidistant. To verify if they are, compute and plot the difference between the locations of subsequent peaks.

```
cycle1 = diff(df)/(2*24);
cycle2 = diff(mf)/(2*24);

subplot(2,1,1)
plot(cycle1)
ylabel('Days')
grid on
title('Dominant peak distance')
subplot(2,1,2)
plot(cycle2,'r')
ylabel('Days')
grid on
title('Minor peak distance')
```




```
mean(cycle1)
```

```
ans = 7
```

```
mean(cycle2)
```

```
ans = 1
```

The minor peaks indicate 7 cycles/week and the dominant peaks indicate 1 cycle per week. This makes sense given that the data comes from a temperature-controlled building on a 7 day calendar. The first 7-day cycle indicates that there is a weekly cyclic behavior of the building temperature where temperatures lower during the weekends and go back to normal during the week days. The 1-day cycle behavior indicates that there is also a daily cyclic behavior - temperatures lower during the night and increase during the day.

## See Also

`alignsignals` | `cpsd` | `finddelay` | `findpeaks` | `mscohere` | `xcorr` | `xcov`

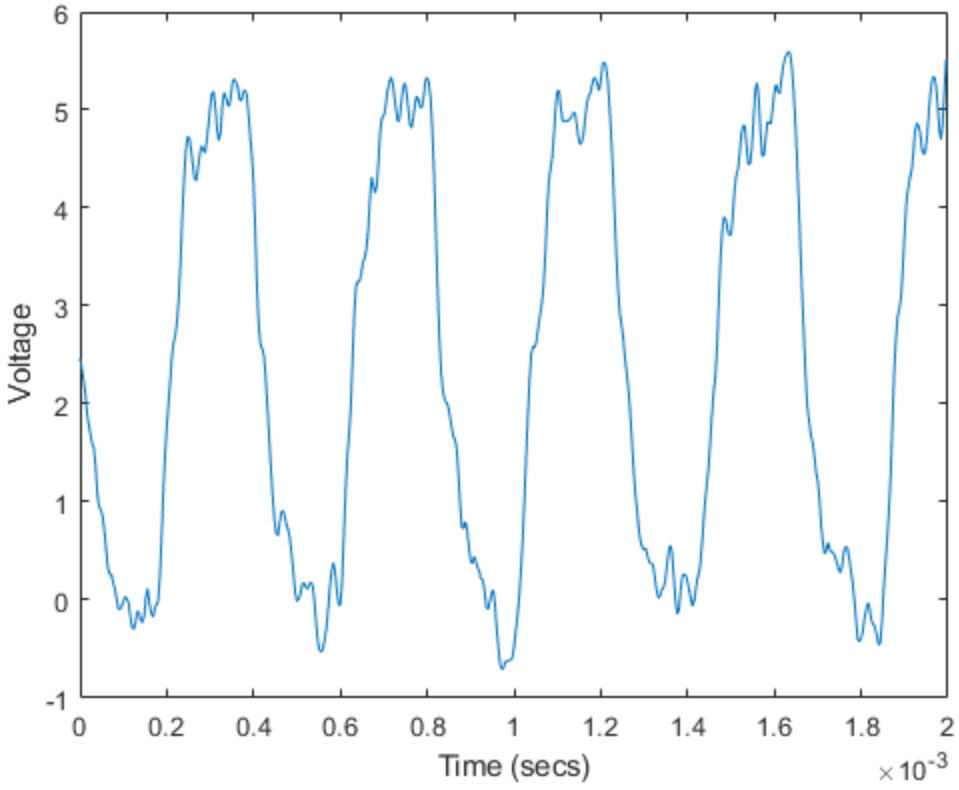
# Measurement of Pulse and Transition Characteristics

This example shows how to analyze pulses and transitions and compute metrics including rise time, fall time, slew rate, overshoot, undershoot, pulse width, and duty cycle.

### Clock Signal with Noise

First let's view the samples from a noisy clock signal.

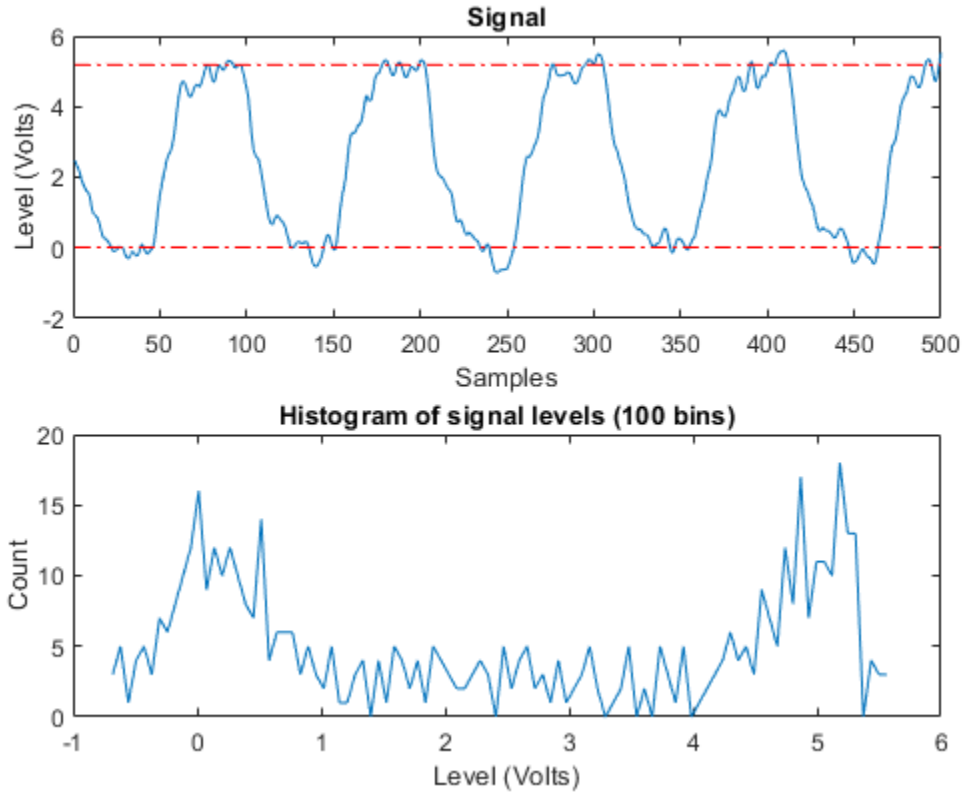
```
load clocksig clock1 time1 Fs
plot(time1, clock1)
xlabel('Time (secs)')
ylabel('Voltage')
```




### Estimate State Levels

Use `statelevels` with no output argument to visualize the state levels. State levels are estimated via histogram.

```
statelevels(clock1)
```




```
ans = 1×2

    0.0138    5.1848
```

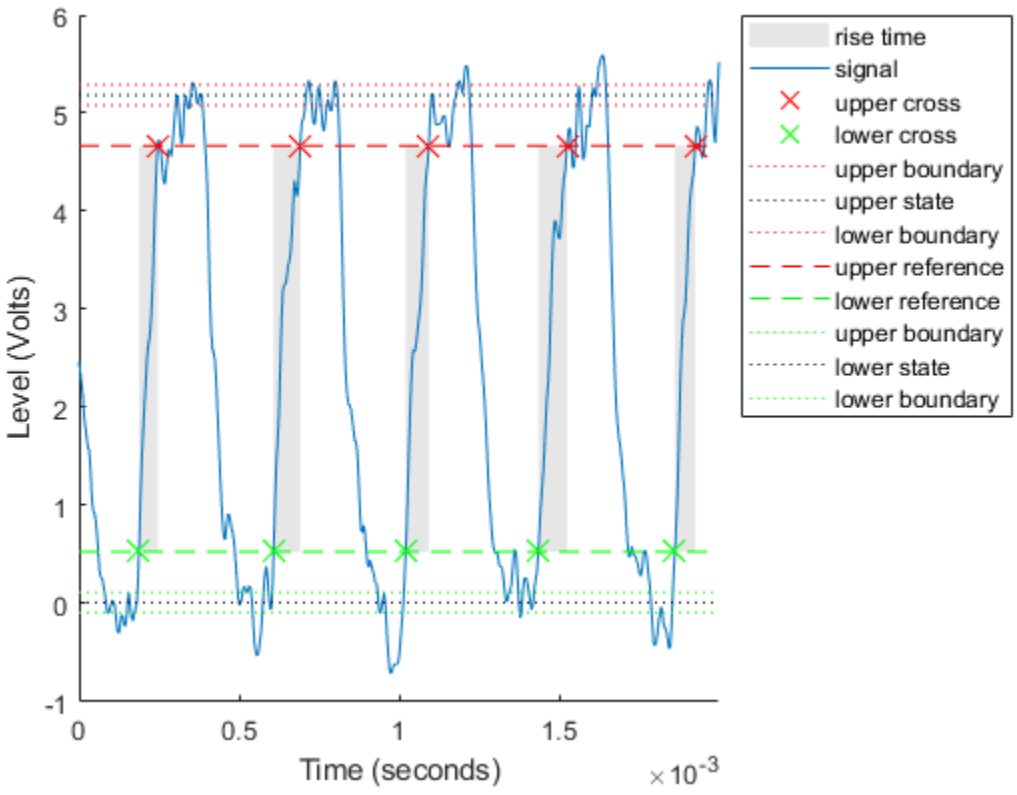
The computed histogram is divided into two equal sized regions between the first and last bin. The mode of each region of the histogram is returned as an estimated state level value in the command window.

Use optional input arguments to specify the number of histogram bins, histogram bounds, and the state level estimation method.

### Measure Rise Time, Fall Time and Slew Rate

Use `risetime` with no output argument to visualize the rise time of positive-going edges.

```
risetime(clock1,time1)
```

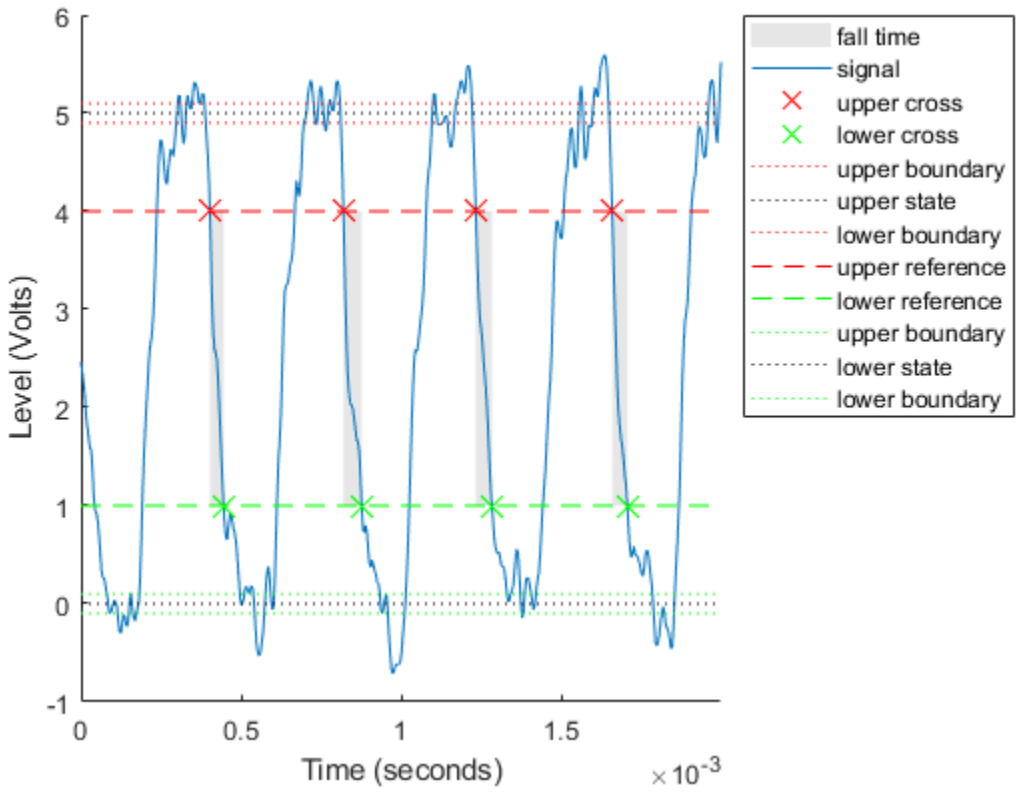



```
ans = 5×1
10^-4 ×

    0.5919
    0.8344
    0.7185
    0.8970
    0.6366
```

The default reference levels for computing rise time and fall time are set at 10% and 90% of the waveform amplitude.

Specify custom reference and state levels via optional input arguments as shown below for a fall time measurement.

```
falltime(clock1,time1,'PercentReferenceLevels',[20 80],'StateLevels',[0 5])
```




```
ans = 4×1
10^-4 ×

    0.4294
    0.5727
    0.5032
    0.4762
```

Obtain measurements programmatically by calling functions with one or more output arguments. For uniformly sampled data, you can provide a sample rate in place of the time vector. Use `slewrate` to measure the slope of each positive-going or negative-going edge.

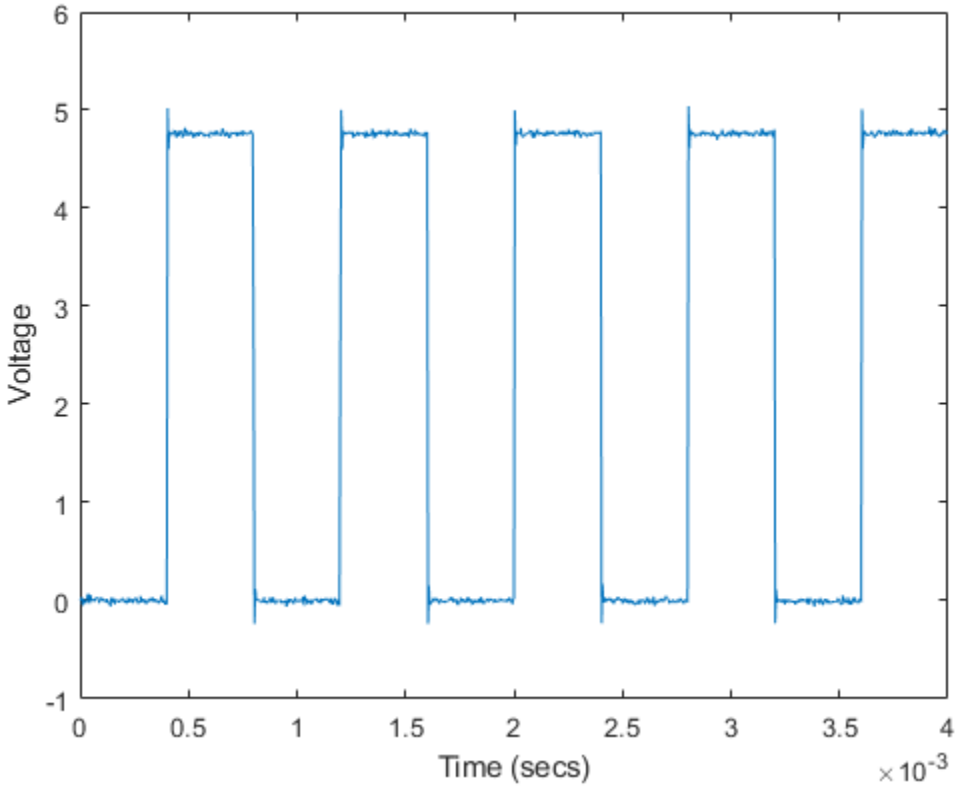
```
sr = slewrate(clock1(1:100),Fs)
```

```
sr = 7.0840e+04
```

### Analyze Overshoot and Undershoot

Now let's view data from a clock with significant overshoot and undershoot.

```
load clocksig clock2 time2 Fs
plot(time2, clock2)
xlabel('Time (secs)')
ylabel('Voltage')
```

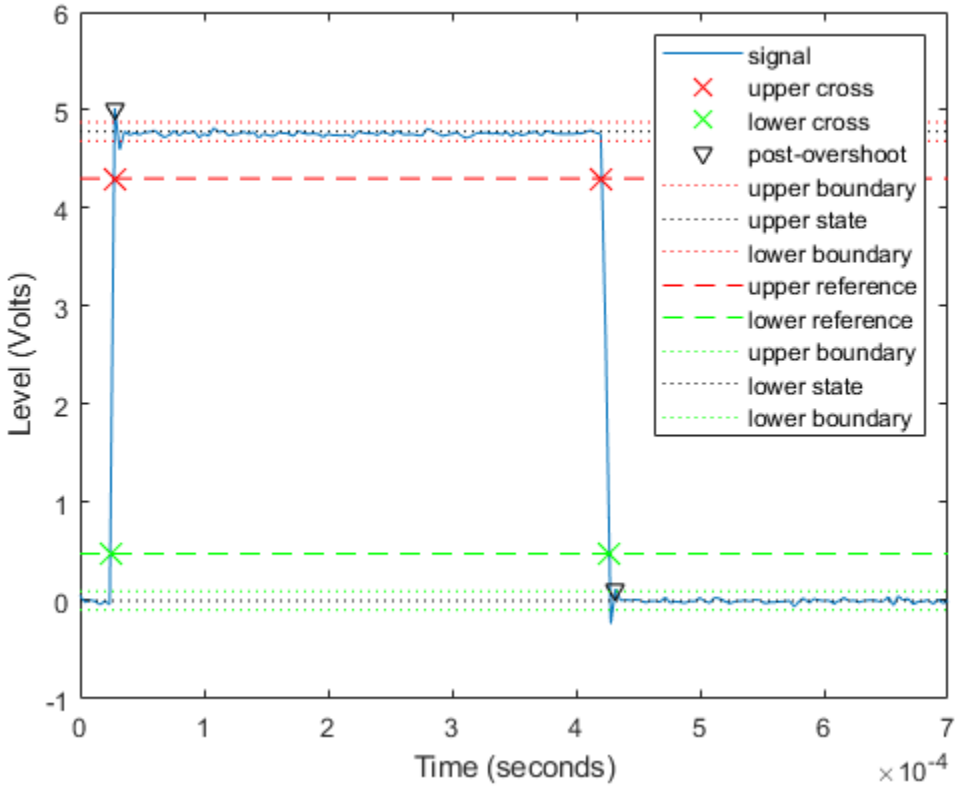



Underdamped clock signals have overshoots. Overshoots are expressed as a percentage of the difference between state levels. Overshoots can occur just after an edge, at the start of the post-transition aberration region. These are called postshoot overshoots. You can measure them by using the `overshoot` function.

```
overshoot(clock2(95:270),Fs)
```

```
ans = 2×1

    4.9451
    2.5399
```

```
legend('Location','NorthEast')
```

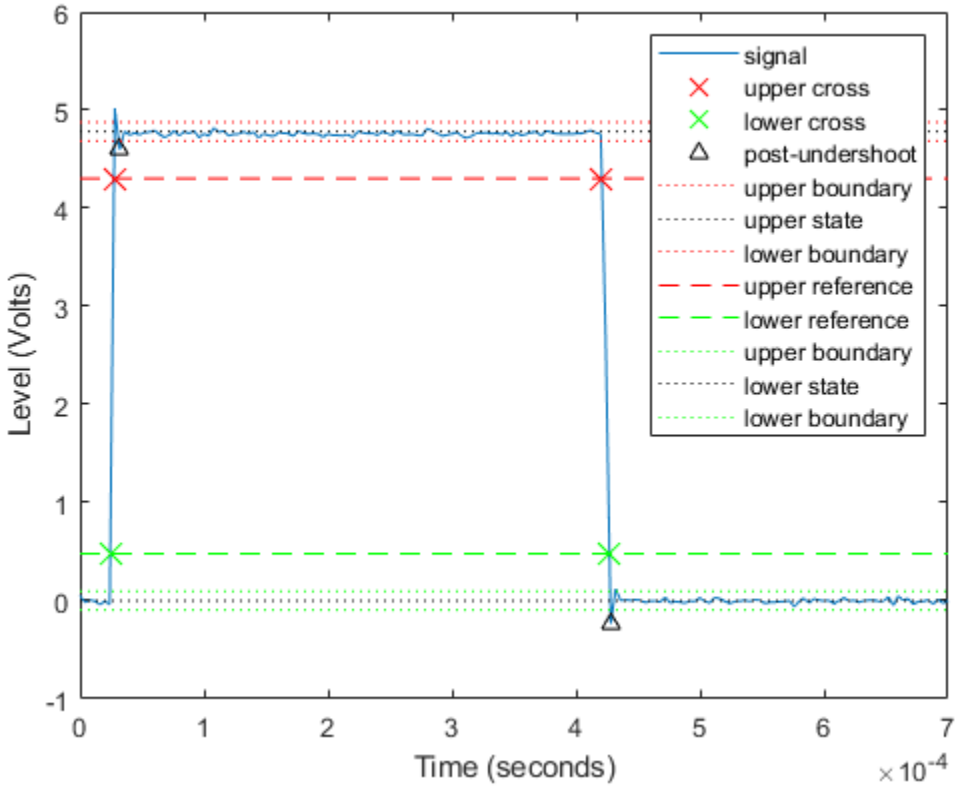



Overshoots may also occur just before an edge, at the end of the pre-transition aberration region. These are called preshoot overshoots. Similarly, you can measure undershoots in the pre and post-aberration regions. Undershoots are also expressed as a percentage of the difference between the state levels. Use optional input arguments to specify the regions in which to measure aberrations.

```
undershoot(clock2(95:270),Fs,'Region','Postshoot')
```

```
ans = 2×1

    3.8499
    4.9451
```

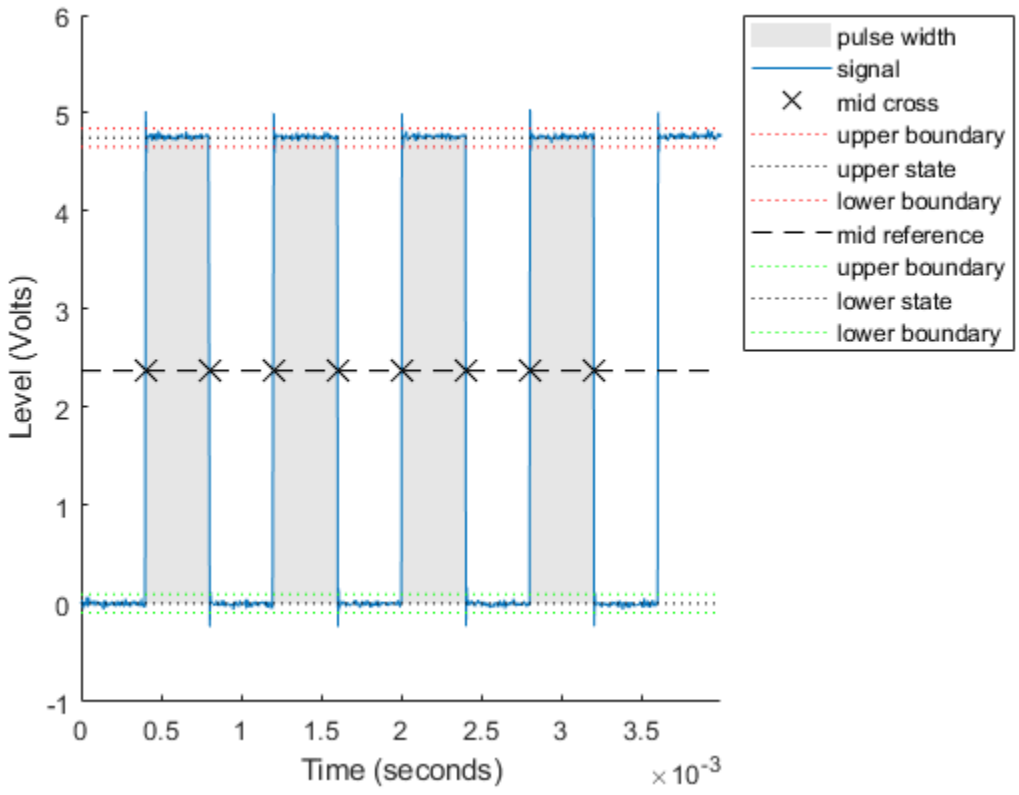
```
legend('Location','NorthEast')
```




**Measure Pulse Width and Duty Cycle**

Use `pulsewidth` with no output argument to plot highlighted pulse widths.

```
pulsewidth(clock2, time2, 'Polarity', 'Positive');
```




This displays pulses of positive polarity. Select negative polarity to see the widths of negative polarity pulses.

Use `dutycycle` to compute the ratio of the pulse width to the pulse period for each positive-polarity or negative-polarity pulse.

```
d = dutycycle(clock2,time2,'Polarity','negative')
```

```
d = 3×1

    0.4979
    0.5000
    0.5000
```

Use `pulseperiod` to obtain the periods of each cycle of the waveform. Use this information to compute other metrics like the average frequency of the waveform or the total observed jitter.

```
pp = pulseperiod(clock2, time2);
```

```
avgFreq = 1./mean(pp)
```

```
avgFreq = 1.2500e+03
```

```
totalJitter = std(pp)
```

```
totalJitter = 1.9866e-06
```

## See Also

dutycycle | falltime | overshoot | pulseperiod | pulsewidth | risetime | slewrate | statelevels | undershoot

# Analyzing Harmonic Distortion

This example shows how to analyze the harmonic distortion of a weakly non-linear system in the presence of noise.

### Introduction

In this example we will explore the output of a simplified model of an amplifier that has noise coupled to the input signal and exhibits non-linearity. We will explore how attenuation at the input can reduce harmonic distortion. We will also give an example of how to mathematically correct for the distortion at the output of the amplifier.

### Viewing the Effects of Non-Linearity

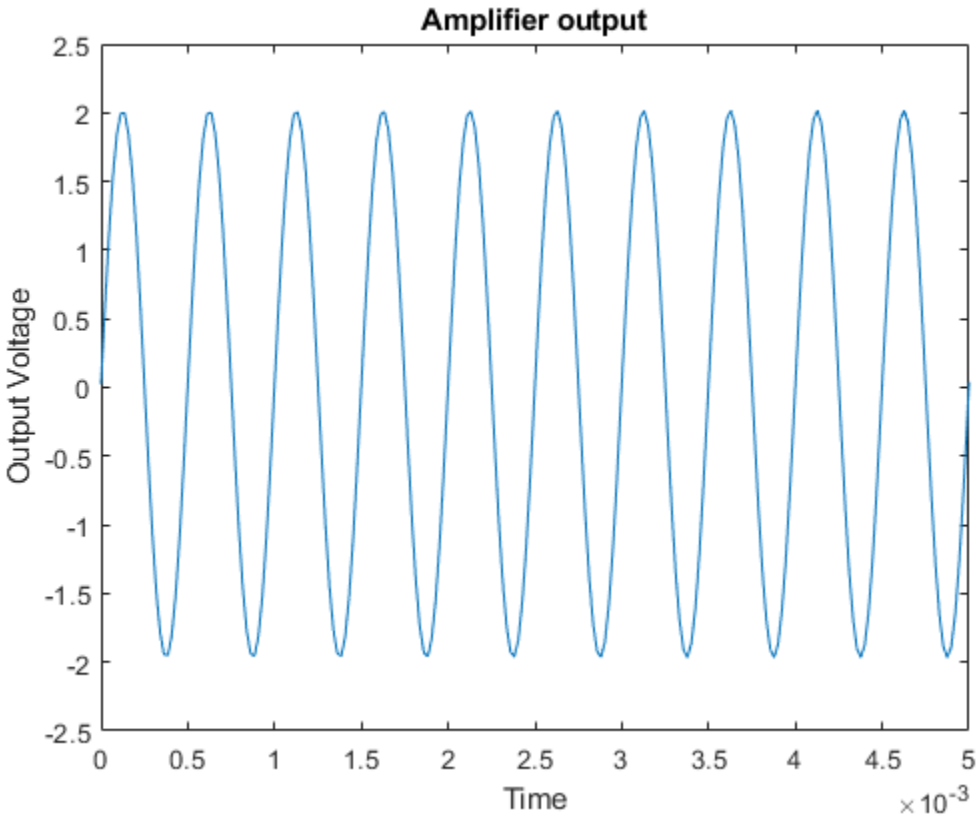
A convenient way to view the effect of the non-linearity of the amplifier is to view the periodogram of its output when stimulated with a sinusoid. The amplitude of the sinusoid is set to the maximum allowable voltage of the amplifier. (2 Vpk)

In this example we will source a 2 kHz sinusoid for a duration of 50ms.

```
VmaxPk = 2;       % Maximum operating voltage
Fi = 2000;        % Sinusoidal frequency of 2 kHz
Fs = 44.1e3;      % Sample rate of 44.1kHz
Tstop = 50e-3;    % Duration of sinusoid
t = 0:1/Fs:Tstop; % Input time vector

% Use the maximum allowable voltage of the amplifier
inputVmax = VmaxPk*sin(2*pi*Fi*t);
outputVmax = helperHarmonicDistortionAmplifier(inputVmax);
```

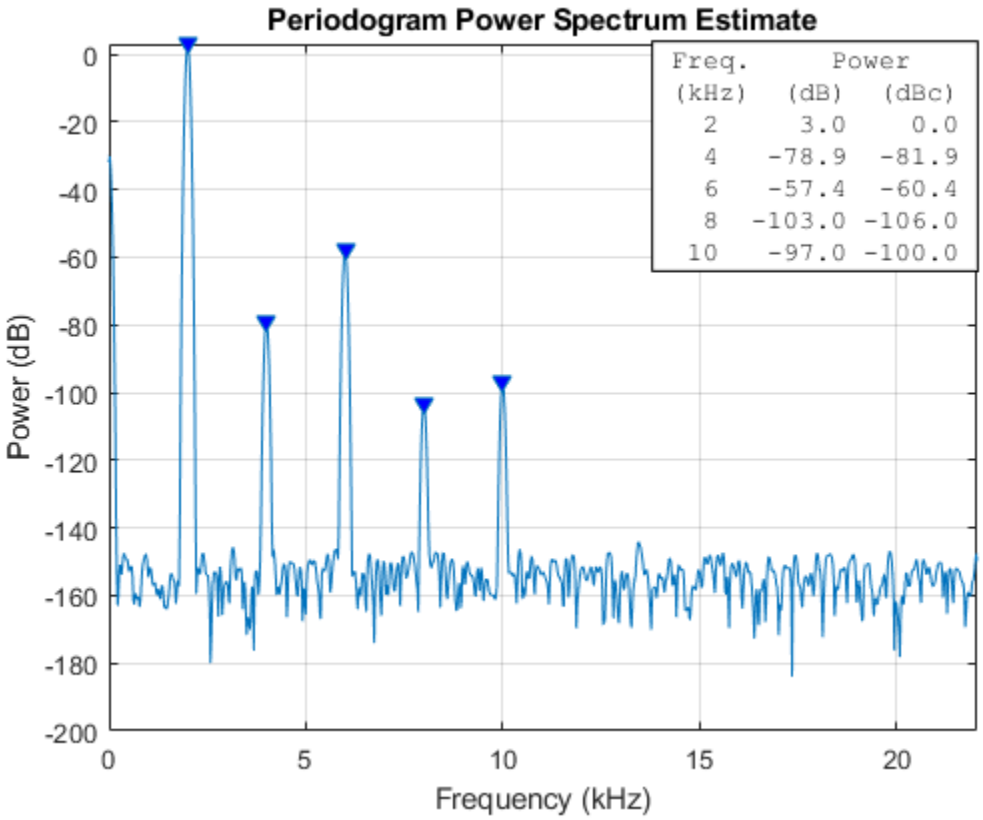
View a zoomed-in region of the output sinusoid. Note that the imperfections of our amplifier are difficult to see visually when plotted with respect to time.

```
plot(t, outputVmax)
xlabel('Time')
ylabel('Output Voltage')
axis([0 5e-3 -2.5 2.5])
title('Amplifier output')
```




Now let's view the periodogram of our amplifier output.

```
helperPlotPeriodogram(outputVmax, Fs, 'power','annotate');
```




Note that instead of seeing just the 2 kHz sinusoid that we placed at the input, we see other sinusoids at 4 kHz, 6 kHz, 8 kHz, and 10 kHz. These sinusoids are multiples of the fundamental 2 kHz frequency and are due to the non-linearity of the amplifier.

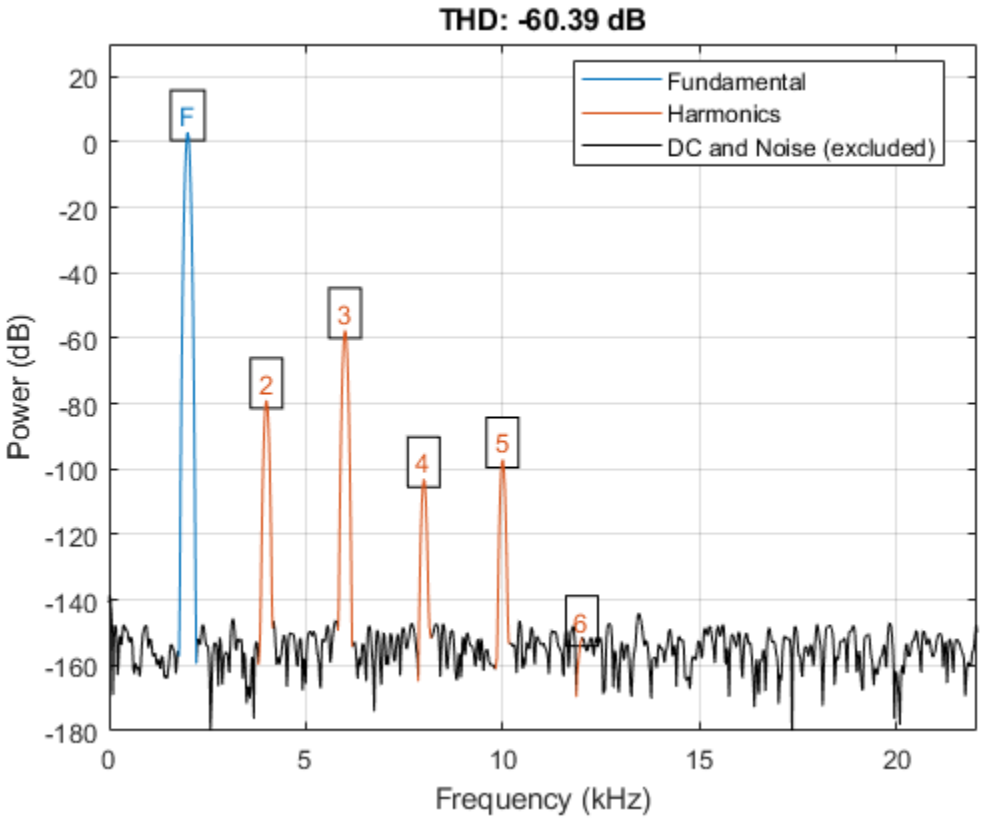
We also see a relatively flat band of noise power.

**Quantifying Non-Linear Distortion**

Let's examine some common distortion metrics for comparison purposes

Our periodogram shows some very well defined harmonics of the fundamental signal. This suggests we measure the total harmonic distortion of the input signal which returns the ratio of power of all harmonic content to the fundamental signal.

```
thd(outputVmax, Fs)
```

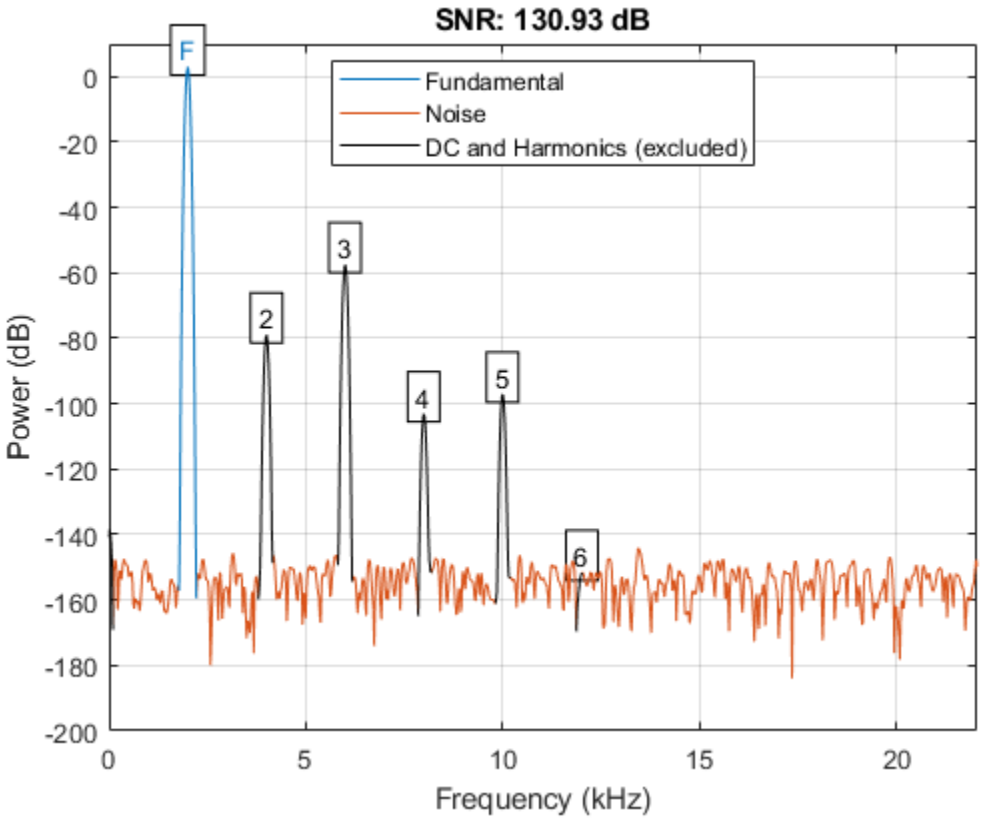



```
ans = -60.3888
```

Notice that the third and largest harmonic is about 60 dB down from the fundamental. This is where most of the distortion is occurring.

We can also obtain an estimate of the total noise present in our input. To do this, we call SNR which returns the ratio of the power of the fundamental to the power of all non-harmonic content.

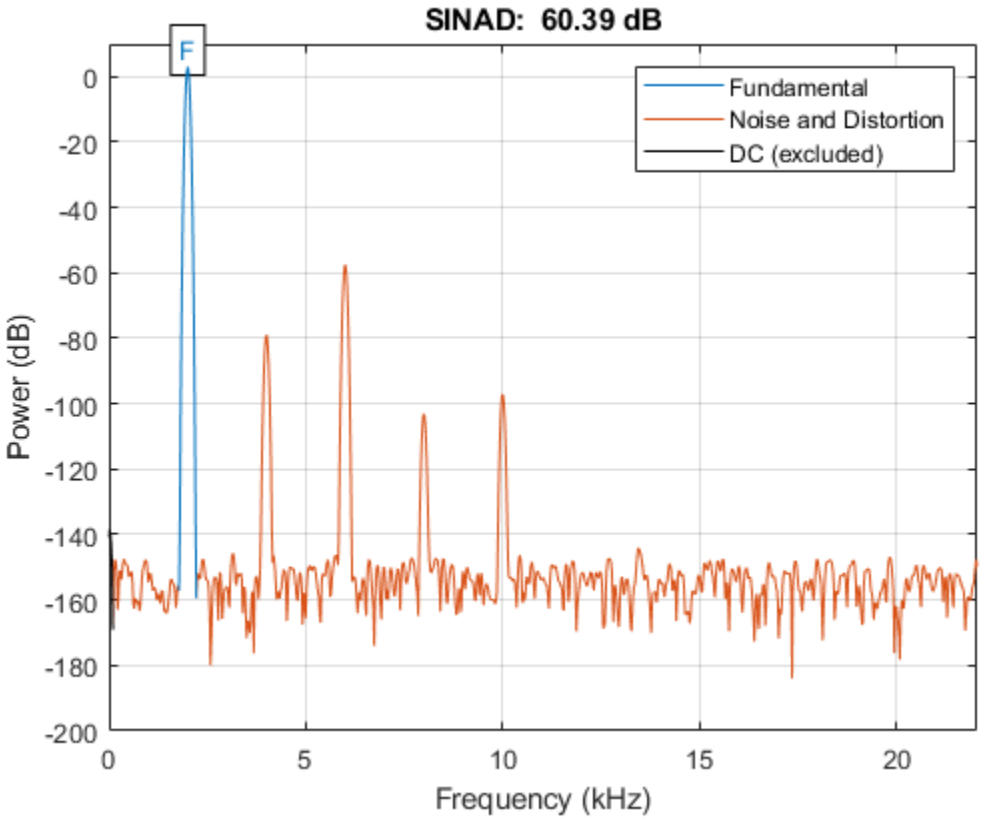
```
snr(outputVmax, Fs)
```




```
ans = 130.9300
```

Another useful metric to compute is SINAD. This computes the ratio of the power to all other harmonic and noise content in the signal.

```
sinad(outputVmax, Fs)
```




```
ans = 60.3888
```

The THD, SNR, and SINAD were -60 dB, 131 dB and 60 dB, respectively. Since the magnitude of THD is roughly equal to SINAD, we can attribute that most of the distortion is due to harmonic distortion.

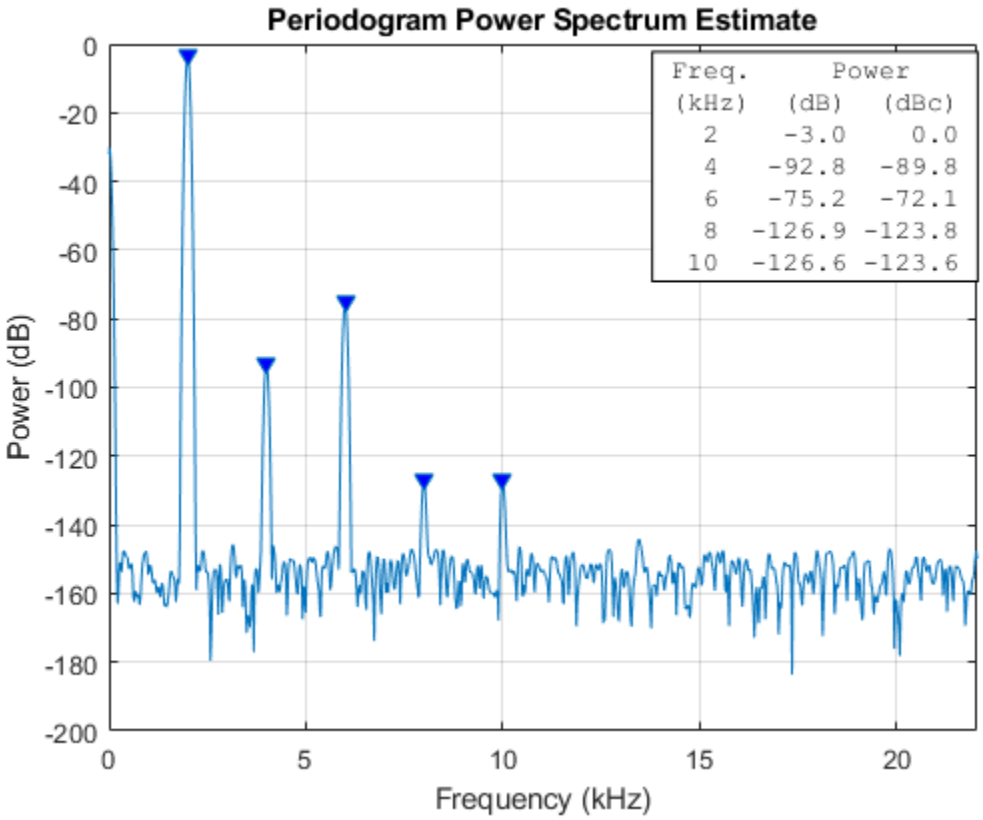
If we inspect the periodogram, we can notice that the third harmonic dominates the distortion of the output.

**Input Attenuation to Reduce Harmonic Distortion**

Most analog circuitry performing amplification has an inherent trade-off between harmonic distortion and noise power. In our example, our amplifier has relatively low noise power compared to the harmonic distortion. This makes it suitable for detecting low power signals. If our input can be attenuated to enter this low-power region, we can recover some of the harmonic distortion.

Let's repeat the measurements by lowering the input voltage by a factor of two.

```
inputVhalf = (VmaxPk/2) * sin(2*pi*Fi*t);
outputVhalf = helperHarmonicDistortionAmplifier(inputVhalf);
helperPlotPeriodogram(outputVhalf, Fs, 'power','annotate');
```




Let's redo our metrics again, this time measuring the effect of lowering the input voltage.

```
thdVhalf = thd(outputVhalf, Fs)
```

```
thdVhalf = -72.0676
```

```
snrVhalf = snr(outputVhalf, Fs)
```

```
snrVhalf = 124.8767
```

```
sinadVhalf = sinad(outputVhalf, Fs)
```

```
sinadVhalf = 72.0676
```

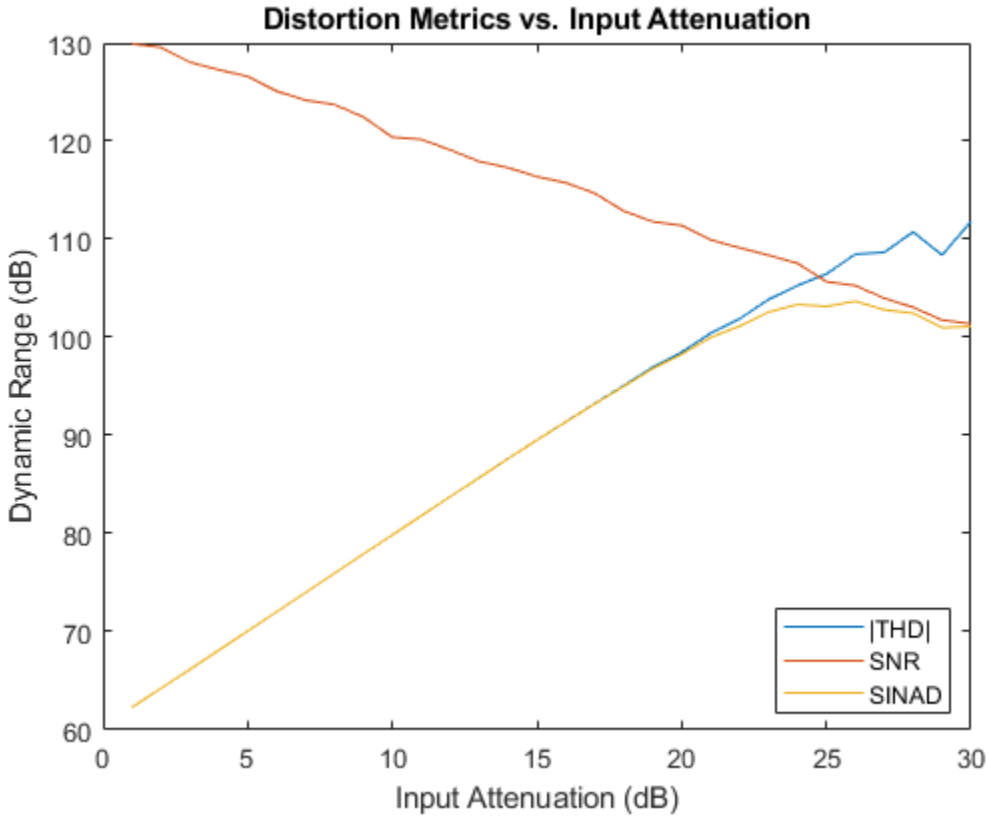
Notice that simply attenuating the input power level by 6 dB reduces the harmonic content. The SINAD and THD improved from ~60 dB to ~72 dB. This came at the expense of lowering the SNR from 131 dB to 125 dB.

### SNR THD and SINAD as a Function of Input Attenuation

Can further attenuation improve our overall distortion performance? Let's plot THD, SNR and SINAD as a function of input attenuation, sweeping the input attenuator from 1 to 30 dB.

```
% Allocate a table with 30 entries
nReadings = 30;
distortionTable = zeros(nReadings, 3);

% Compute the THD, SNR and SINAD for each of the attenuation settings
```

```
for i = 1:nReadings
  inputVbestAtten = db2mag(-i) * VmaxPk * sin(2*pi*Fi*t);
  outputVbestAtten = helperHarmonicDistortionAmplifier(inputVbestAtten);
  distortionTable(i,:) = [abs(thd(outputVbestAtten, Fs))
                          snr(outputVbestAtten, Fs)
                          sinad(outputVbestAtten, Fs)];
end

% Plot results
plot(distortionTable)
xlabel('Input Attenuation (dB)')
ylabel('Dynamic Range (dB)')
legend('|THD|','SNR','SINAD','Location','best')
title('Distortion Metrics vs. Input Attenuation')
```




The graph shows the usable dynamic range corresponding to each metric. The *magnitude* of THD corresponds to the range that is free of harmonics. Similarly, SNR corresponds to the dynamic range of that is unaffected by noise; SINAD corresponds to the total dynamic range that is free of distortion.

As can be seen from the graph, SNR degrades as input power attenuation increases. This is because when you attenuate the signal, only the signal is attenuated, but the noise floor of the amplifier stays the same.

Also note that the magnitude of the total harmonic distortion improves steadily until it intersects the SNR curve, after which the measurement becomes unstable. This happens when the harmonics have "disappeared" beneath the noise of the amplifier.

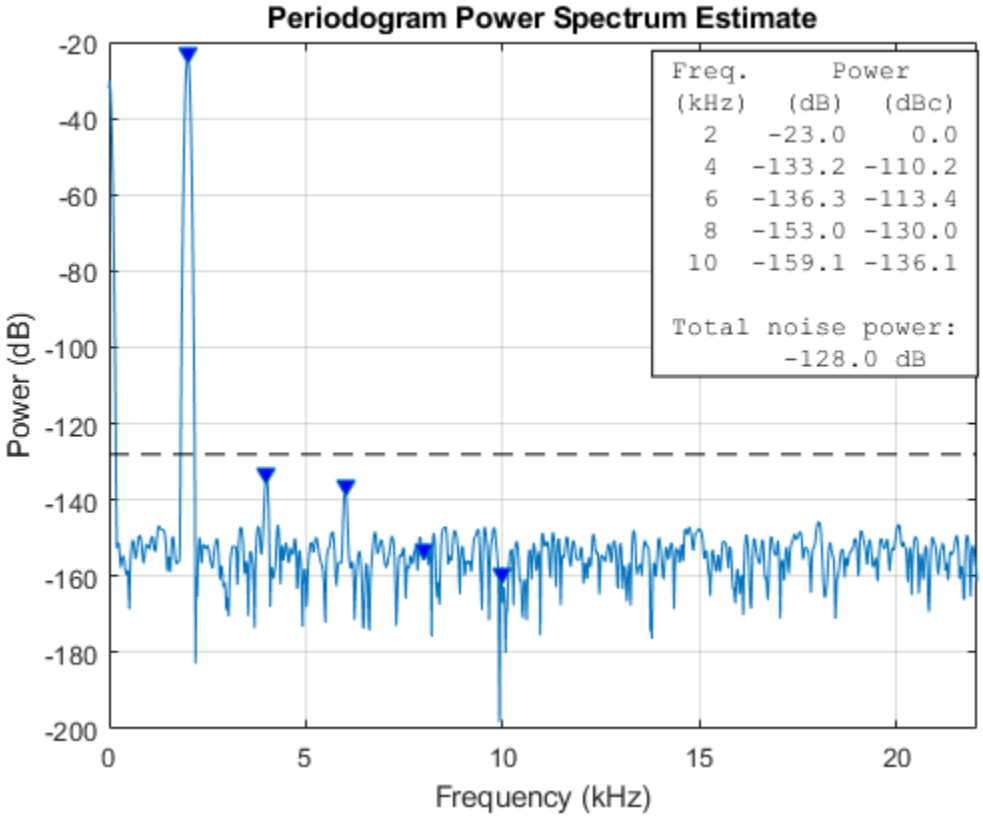
A practical choice of amplifier attenuation for the amplifier would be 26 dB (yielding a SINAD of 103 dB). This would be a reasonable tradeoff between harmonic and noise distortion.

```
% Search the table for the largest SINAD reading
[maxSINAD, iAtten] = max(distortionTable(:,3));
fprintf('Max SINAD (%.1f dB) occurs at %.f dB attenuation\n', ...
  maxSINAD, iAtten)
```

```
Max SINAD (103.7 dB) occurs at 26 dB attenuation
```

Let's plot the periodogram when the attenuator is set to 26 dB.

```
inputVbestAtten = db2mag(-iAtten) * VmaxPk * sin(2*pi*Fi*t);
outputVbestAtten = helperHarmonicDistortionAmplifier(inputVbestAtten);
helperPlotPeriodogram(outputVbestAtten, Fs, 'power','annotate','shownoise');
```




Here we have additionally plotted the level of *total* noise power that is spread across the spectrum. Note that at this attenuation setting, the second and third harmonic are still visible in the spectrum but also considerably less than the total noise power. If we were to have an application that uses a smaller bandwidth of the available spectrum we would benefit from further increasing the attenuation to reduce the harmonic content.

### Post-processing to Remove Distortion

Occasionally we can correct for some of the non-linearity of the amplifier. If the output of the amplifier is digitized, we can recover more useful dynamic range by digitally post-processing the captured output and correcting for the non-linearity mathematically.

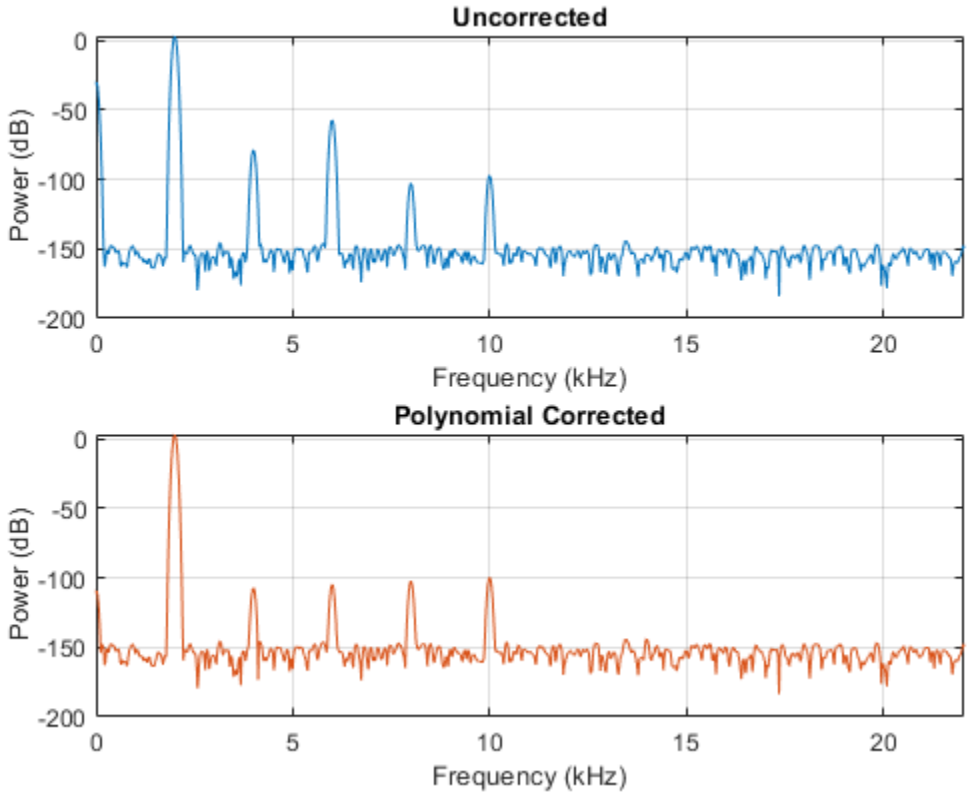
In our case, we stimulate the input with a linear ramp and fit a third-order polynomial that best fits the input.

```
inputRamp = -2:0.00001:2;
outputRamp = helperHarmonicDistortionAmplifier(inputRamp);
polyCoeff = polyfit(outputRamp,inputRamp,3)
```

```
polyCoeff = 1×4

    0.0010   -0.0002    1.0000   -0.0250
```

Now that we have the coefficients we can then perform post-correction at the output and compare side-by-side with our original uncorrected output

```
correctedOutputVmax = polyval(polyCoeff, outputVmax);

helperPlotPeriodogram([outputVmax; correctedOutputVmax],Fs,'power');
subplot(2,1,1)
title('Uncorrected')
subplot(2,1,2)
title('Polynomial Corrected')
```




Note that the second and third harmonics are significantly reduced when using polynomial correction.

Let's repeat the measurements again with the corrected output.

```
thdCorrectedVmax = thd(correctedOutputVmax, Fs)
```

```
thdCorrectedVmax = -99.6194
```

```
snrCorrectedVmax = snr(correctedOutputVmax, Fs)
```

```
snrCorrectedVmax = 130.7491
```

```
sinadCorrectedVmax = sinad(correctedOutputVmax, Fs)
```

```
sinadCorrectedVmax = 99.6162
```

Notice that our SINAD (and THD) dropped from 60 dB down to 99 dB, while preserving our original SNR of 131 dB.

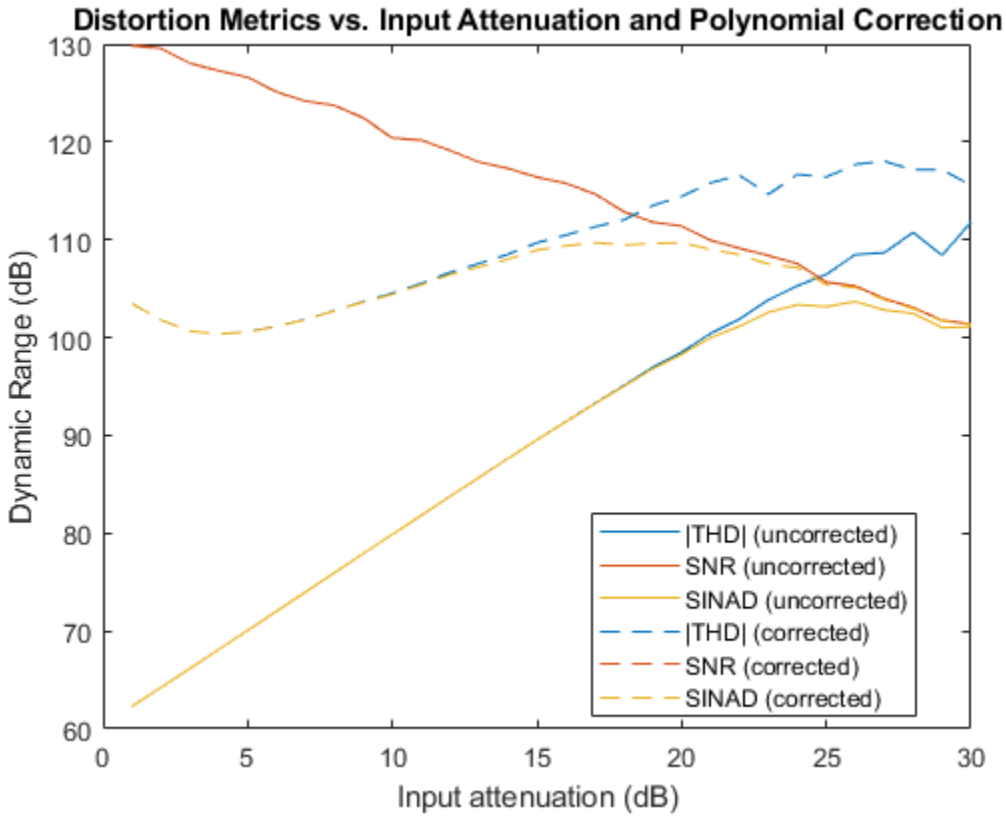
### Combining Techniques

We can combine attenuation with polynomial evaluation to find the ideal operating voltage that minimizes the overall SINAD of our system.

```
subplot(1,1,1)
% Add three more columns to our distortion table
distortionTable = [distortionTable zeros(nReadings,3)];
for i = 1:nReadings
  inputVreduced = db2mag(-i) * VmaxPk * sin(2*pi*Fi*t);
  outputVreduced = helperHarmonicDistortionAmplifier(inputVreduced);
  correctedOutput = polyval(polyCoeff, outputVreduced);
  distortionTable(i,4:6) = [abs(thd(correctedOutput, Fs))
                            snr(correctedOutput, Fs)
                            sinad(correctedOutput, Fs)];
end

h = plot(distortionTable)
```

```
h =
  6x1 Line array:

  Line
  Line
  Line
  Line
  Line
  Line
```

```
xlabel('Input attenuation (dB)')
ylabel('Dynamic Range (dB)')
for i = 1:3
  h(i+3).Color = h(i).Color;
  h(i+3).LineStyle = '--' ;
end
legend('|THD| (uncorrected)','SNR (uncorrected)','SINAD (uncorrected)', ...
 '|THD| (corrected)','SNR (corrected)','SINAD (corrected)','Location','best')
title('Distortion Metrics vs. Input Attenuation and Polynomial Correction');
```




Here, we've plotted all three metrics alongside for both the uncorrected and polynomial corrected amplifier.

As can be seen from the graph, THD has improved considerably, whereas SNR was not affected by polynomial correction. This is to be expected since the polynomial correction only affects the harmonic distortion and not the noise distortion.

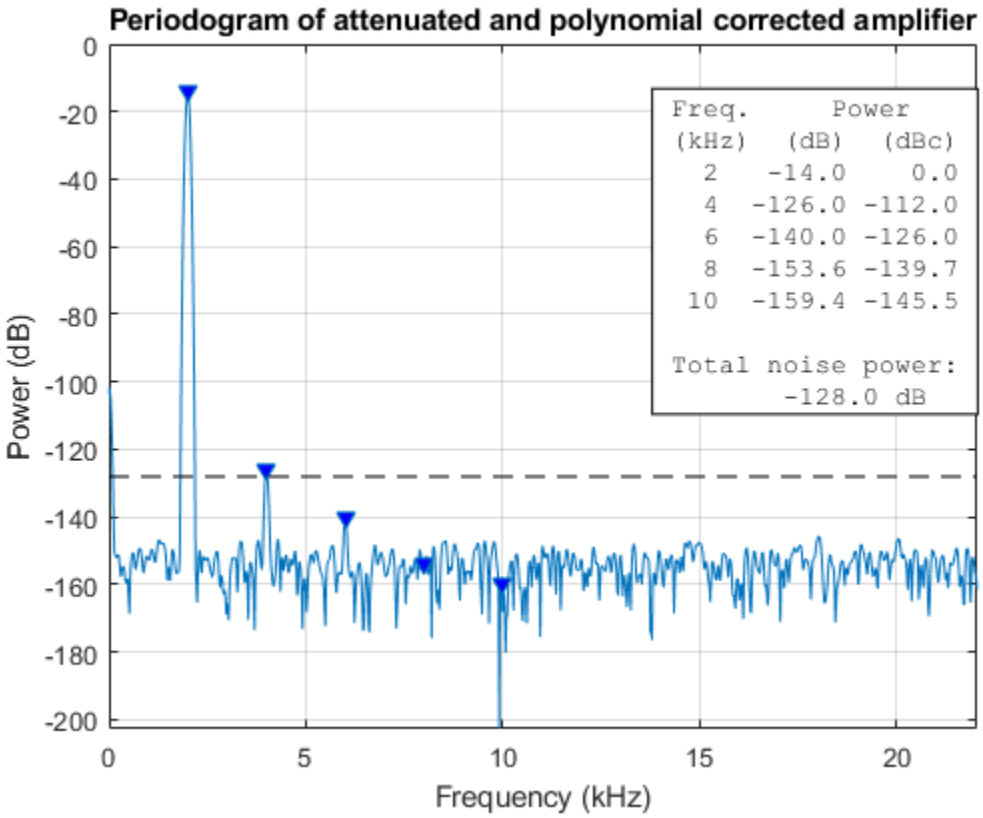
Let's show the maximum SINAD possible when corrected by the polynomial

```
[maxSINADcorrected, iAttenCorr] = max(distortionTable(:,6));
fprintf('Corrected:    Max SINAD (%.1f dB) at %.f dB attenuation\n', ...
  maxSINADcorrected, iAttenCorr)
```

```
Corrected:    Max SINAD (109.7 dB) at 17 dB attenuation
```

A good choice of amplifier attenuation for the polynomial corrected amplifier would be 20dB (yielding a SINAD of 109.8 dB).

```
% Recompute amplifier at maximum SINAD attenuation setting with polynomial
inputVreduced = db2mag(-iAttenCorr) * VmaxPk * sin(2*pi*Fi*t);
outputVreduced = helperHarmonicDistortionAmplifier(inputVreduced);
correctedOutputVbestAtten = polyval(polyCoeff, outputVreduced);

helperPlotPeriodogram(correctedOutputVbestAtten, Fs, 'power','annotate','shownoise');
title('Periodogram of attenuated and polynomial corrected amplifier')
```




Note that all but the second harmonic disappeared entirely with polynomial correction under the ideal attenuation setting. As noted before the second harmonic appears just underneath the power level of the total noisefloor. This provides a reasonable tradeoff in applications which use the full bandwidth of the amplifier.

**Summary**

We have shown how polynomial correction can be applied to the output of an amplifier experiencing distortion and how to pick a reasonable attenuation value to reduce the effects of harmonic distortion.

## See Also

`sinad` | `snr` | `thd`

# Spurious-Free Dynamic Range (SFDR) Measurement

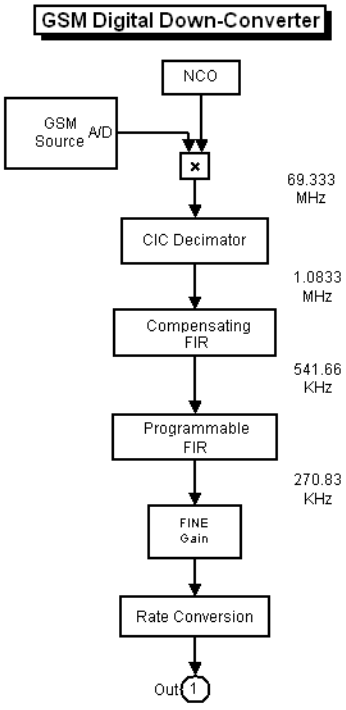
This example shows how to analyze a numerically controlled oscillator (NCO) of a digital down-converter (DDC) implemented in fixed-point arithmetic. The example measures the spurious free dynamic range (SFDR) of the NCO, and explore the effects of adding phase dither. The number of dither bits affects hardware implementation choices. The example shows trade-offs among noise floor level, spurious effects, and number of dither bits. The DDC in the example, designed to meet the GSM specification, models the Graychip 4016.

### Introduction

Numerically controlled oscillators (NCOs) are an efficient means of generating sinusoidal signals, and are useful when you require a continuous-phase sinusoidal signal with variable frequency.

A DDC is a key component of digital radios. It translates the high-input sample rates of a digital radio down to lower sample rates (baseband) for further and easier processing. Our DDC has an input rate of 69.333 MHz and is tasked with converting the rate down to 270.833 kHz in accordance with GSM specifications.

The DDC consists of an NCO and a mixer to quadrature down convert the input signal to baseband. A Cascaded Integrator-Comb (CIC) then low-pass filters the baseband signal, and along with two FIR decimating filters downsample the signal to achieve the desired low sample rate, which is then ready for further processing. The final stage, depending on the application, often includes a resampler that interpolates or decimates the signal to achieve the desired sample rate. Further filtering can be achieved with the resampler. See the block diagram of a typical DDC, below. Note that Simulink® handles complex signals, so we don't have to treat the I and Q channels separately.



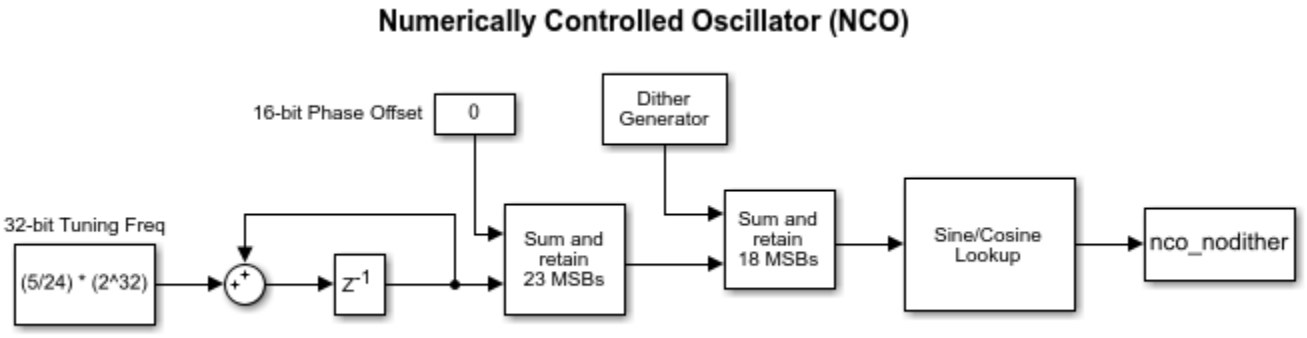

While this example focuses on the analysis of the NCO, an example titled "Implementing the Filter Chain of a Digital Down-Converter", focusing on designing the three-stage, multirate, fixed-point filter chain and HDL code generation is available in the DSP System Toolbox™.

### The Numerically Controlled Oscillator

The digital mixer section of the DDC includes a multiplier and an NCO, which provide channel selection or tuning for the radio. The mixer is basically a sine-cosine generator, creating complex values for each sine-cosine pair. The typical NCO has four components: the phase accumulator, the phase adder, the dither generator, and sine-cosine lookup table.

Here is a typical NCO circuit modeled in Simulink, similar to what you might see in the Graychip data sheet.

```
open_system('ddcncomodel')
```




Based on the input frequency, the NCO's phase accumulator produces values that address a sine-cosine lookup table. The phase adder specifies a phase offset that modulates the output of the phase accumulator. The Dither Generator provides phase dithering to reduce amplitude quantization noise, and improving the SFDR of the NCO. The Sine/Cosine Lookup block produces the actual complex sinusoidal signal, and the output is stored in the variable nco_nodither.

In the Graychip, the tuning frequency is specified as a normalized value relative to the chip's clock rate. So for a tuning frequency of F, the normalized frequency is F/Fclk, where Fclk is the chip's clock rate. The phase offset is specified in radians, ranging from 0 to 2pi. In this example the normalized tuning frequency is set to 5/24 while the phase offset is set to 0. The tuning frequency is specified as a 32-bit word and the phase offset is specified as a 16-bit word.

Since the NCO is implemented using fixed-point arithmetic, it experiences undesirable amplitude quantization effects. These numerical distortions are due to the effects of finite word length. Basically, sinusoids are quantized creating cumulative, deterministic, and periodic errors in the time domain. These errors, appear as line spectra or spurs in the frequency domain. The amount of attenuation from the peak of the signal of interest to the highest spur is the SFDR.

The SFDR of the Graychip is 106 dB, but the GSM specification requires that the SFDR be greater than 110 dB. There are several ways to improve the SFDR, and you will explore adding phase dither to the NCO.

The Graychip's NCO contains a phase dither generator which is basically a random integer generator used to improve the oscillator's output purity. Dithering causes the unintended periodicities of the quantization noise (which causes "spikes" in spectra and thus poor SFDR) to be spread across a broad spectrum, effectively reducing these undesired spectral peaks. Conservation of energy applies, however, so the spreading effectively raises the overall noise floor. That is, the dithering is good for SFDR, but only up to a point.

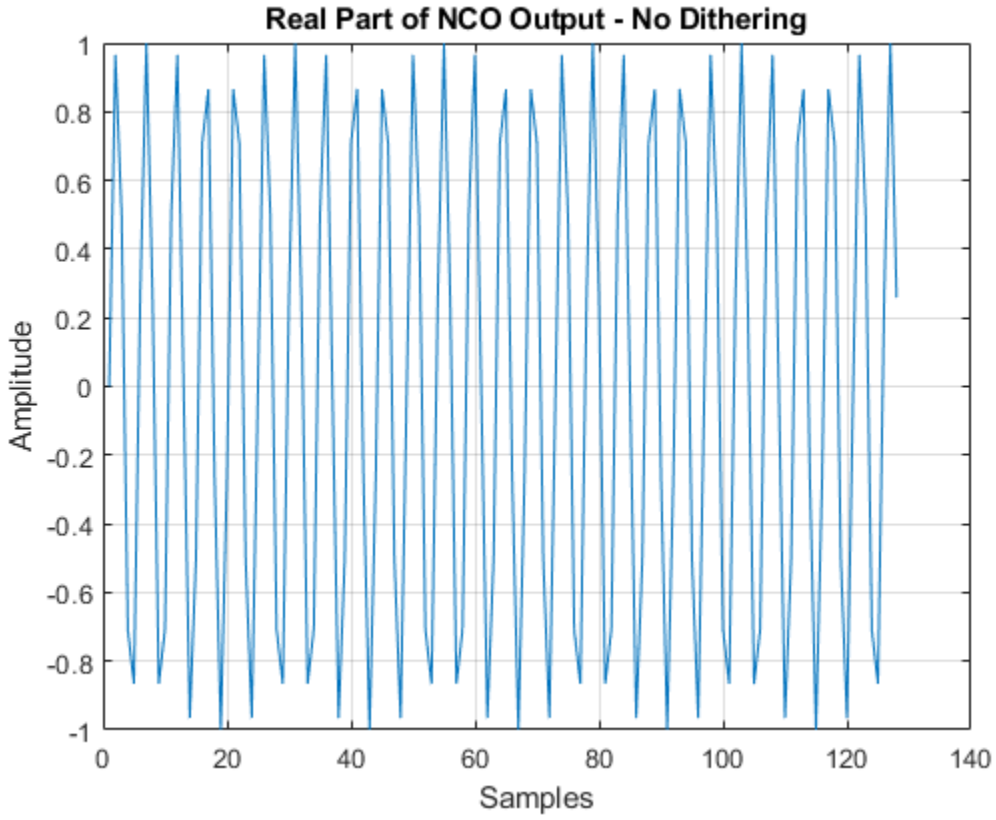
Let's run the NCO model and analyze its output in the MATLAB workspace. This model does not use dithering.

```
sim('ddcncomodel')
whos nco*
```

```
  Name              Size                  Bytes  Class     Attributes

  nco_nodither      1x1x20545            328720  double    complex
```

The plot below displays the real part of the first 128 samples of the NCO's output, stored in the variable, nco_nodither.

```
plot(real(squeeze(nco_nodither(1:128))))
grid
title('Real Part of NCO Output - No Dithering')
ylabel('Amplitude')
xlabel('Samples')
```

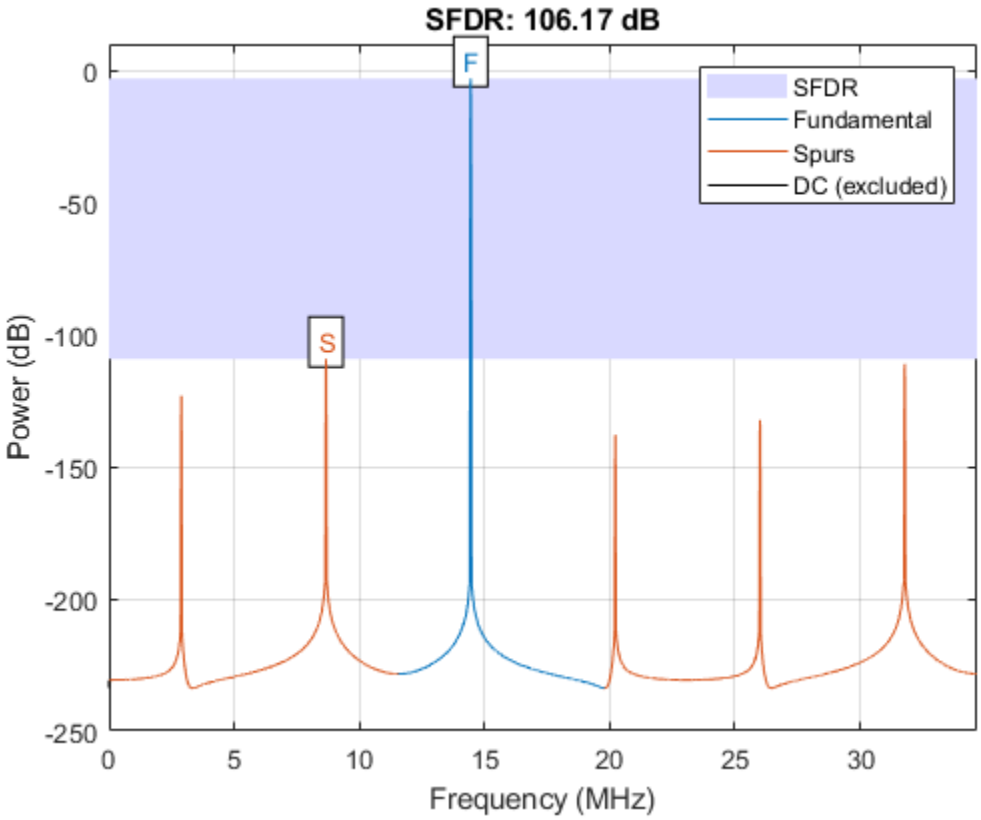



### SFDR of NCO Output

Let's take a look at the SFDR of the output of the NCO.

Calculate and plot the SFDR of the NCO output

```
Fs = 69.333e6;
sfdr(real(nco_nodither),Fs);
```



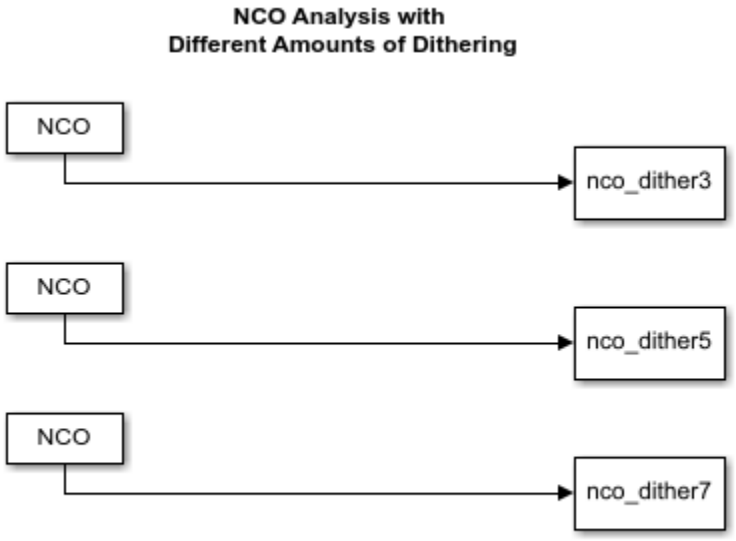

As expected, the power spectrum plot shows a peak at 14.44 MHz, which is the NCO's tuning frequency, 5/24*Fs = 14.444 MHz.

The SFDR, however, is about 106.17 dB, which is too high to meet the GSM specification, which requires 110 dB or more. We can improve this dynamic range by use of phase dithering.

### Exploring the Effects of Dithering

To explore adding dither to the NCO, the NCO circuit shown above has been encapsulated in a subsystem and duplicated three times. A different amount of dither was selected for each NCO. Although the NCO allows a range of 1 to 19 bits of dither to be specified, you will try just few values. Running this model will produce three different NCO outputs based on the amount of dither added.

```
open_system('ddcncowithdithermodel')
```




Running the simulation will produce three signals in the MATLAB workspace that you can then analyze using the spectral analysis algorithm defined above. You can run the simulation from the model or from the command line using the sim command.

```
sim('ddcncowithdithermodel')
```

After the simulation completes you are left with the signals that are the NCOs' output. Each signal shows a different amount of dithering.

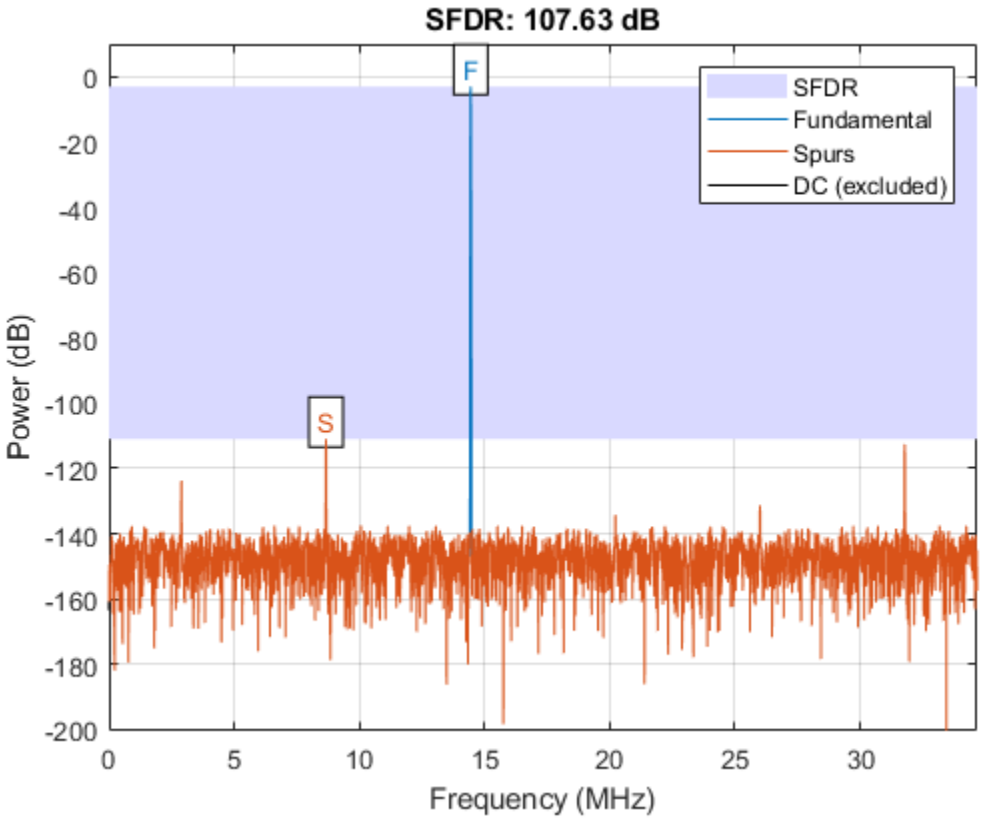
```
whos nco*
```

```
  Name              Size                 Bytes  Class     Attributes

  nco_dither3       1x1x20501           328016  double    complex
  nco_dither5       1x1x20501           328016  double    complex
  nco_dither7       1x1x20501           328016  double    complex
  nco_nodither      1x1x20545           328720  double    complex
```

Compute and plot the SFDR after adding 3 bits of dithering.

```
figure
sfdr(real(nco_dither3),Fs)
```

```
ans =

  107.6285
```

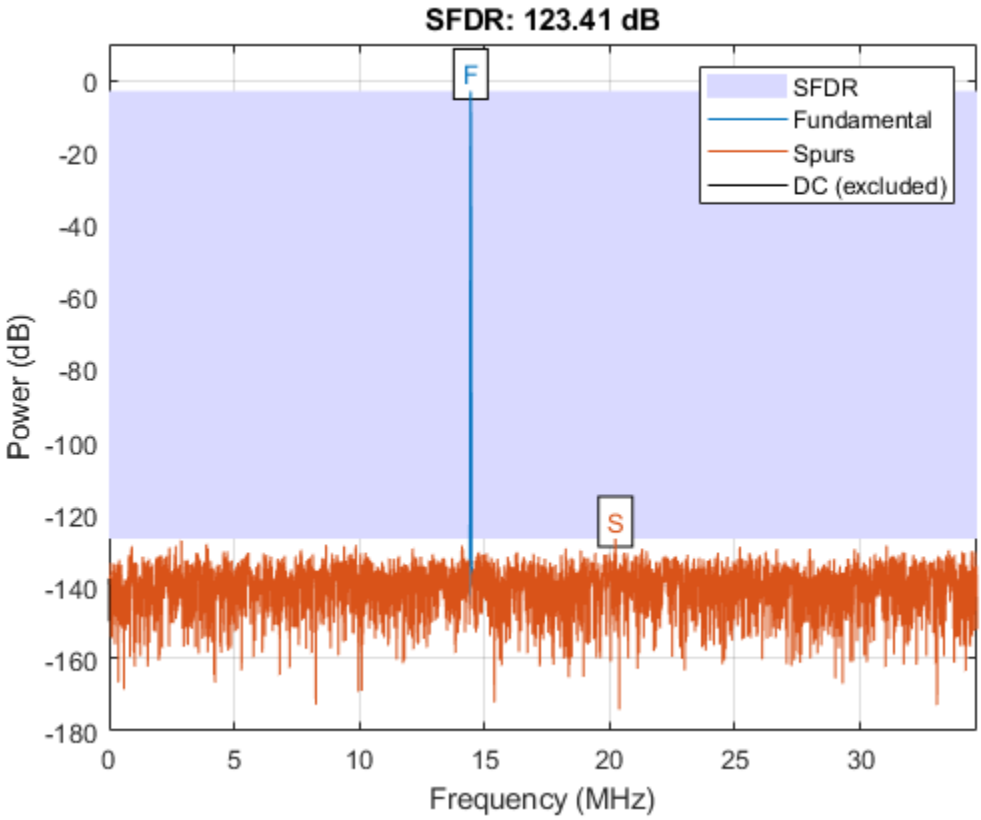



With three bits of dither added, the highest spur is now about -112 dB. The SFDR is 107.63 dB. It still fails to meet the GSM specification.

Now add 5 bits of dithering.

```
figure
sfdr(real(nco_dither5),Fs)
```

```
ans =

  123.4065
```



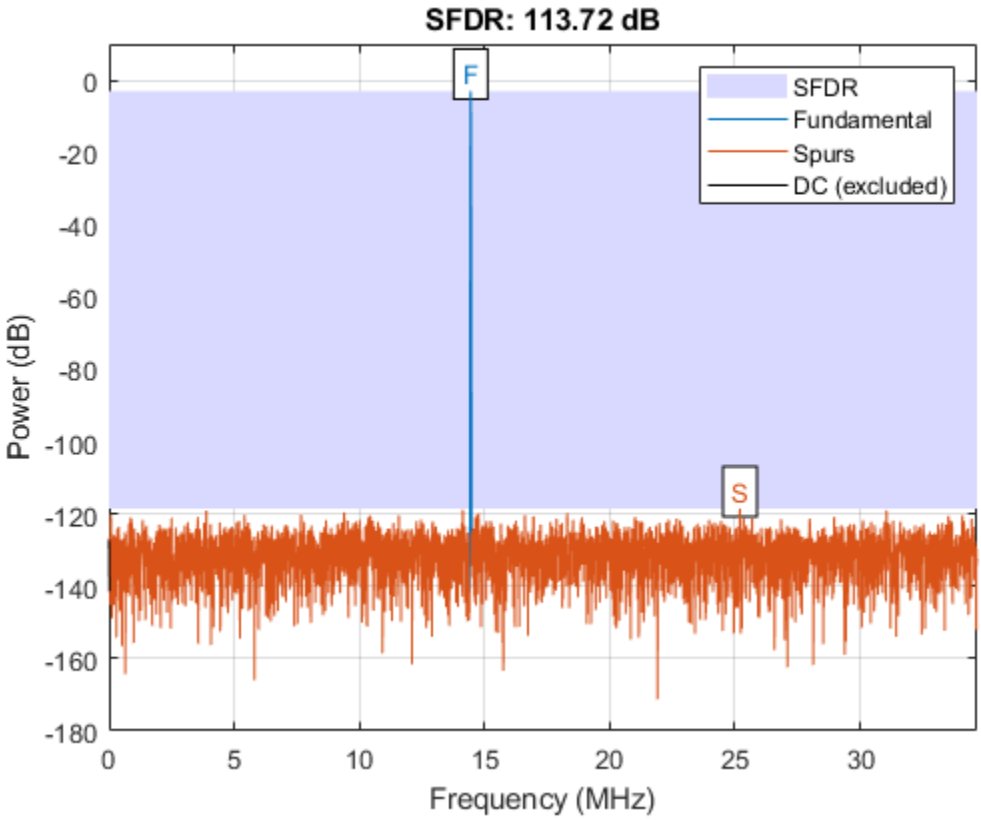

With five bits of dither added, the highest spur is now about -127 dB.

The SFDR is 123.41 dB, exceeding the GSM specification.

It appears that more dither gives better results, so add 7 bits of dithering.

```
figure
sfdr(real(nco_dither7),Fs)
```

```
ans =

  113.7189
```




Note that our computed SFDR degraded to 113.72 dB. This is due to the broadband noise generated by dithering starting to overtake the power of the spurious content.

Using 7 bits of dithering meets the GSM specification, but is not as effective as using 5 bits of dithering.

### Comparing Results

Tabulate the SFDR for each NCO output against the amounts of dithering for each NCO output.

```
Number of     Spur Free Dynamic
Dither bits   Range(dB)

   0            106.17

   3            107.63

   5            123.41

   7            113.72
```

### Summary

This example analyzed the output of an NCO used in a digital down converter for a GSM application. Spectral analysis was used to measure the SFDR, the difference between the highest spur and the peak of the signal of interest. Spurs are deterministic, periodic errors that result from quantization effects. The example also explored the effects of adding dither in the NCO, which adds random data

to the NCO to improve its purity. We found that using five bits of dithering achieved the highest SFDR.

## See Also

`sfdr`

# Extracting Classification Features from Physiological Signals

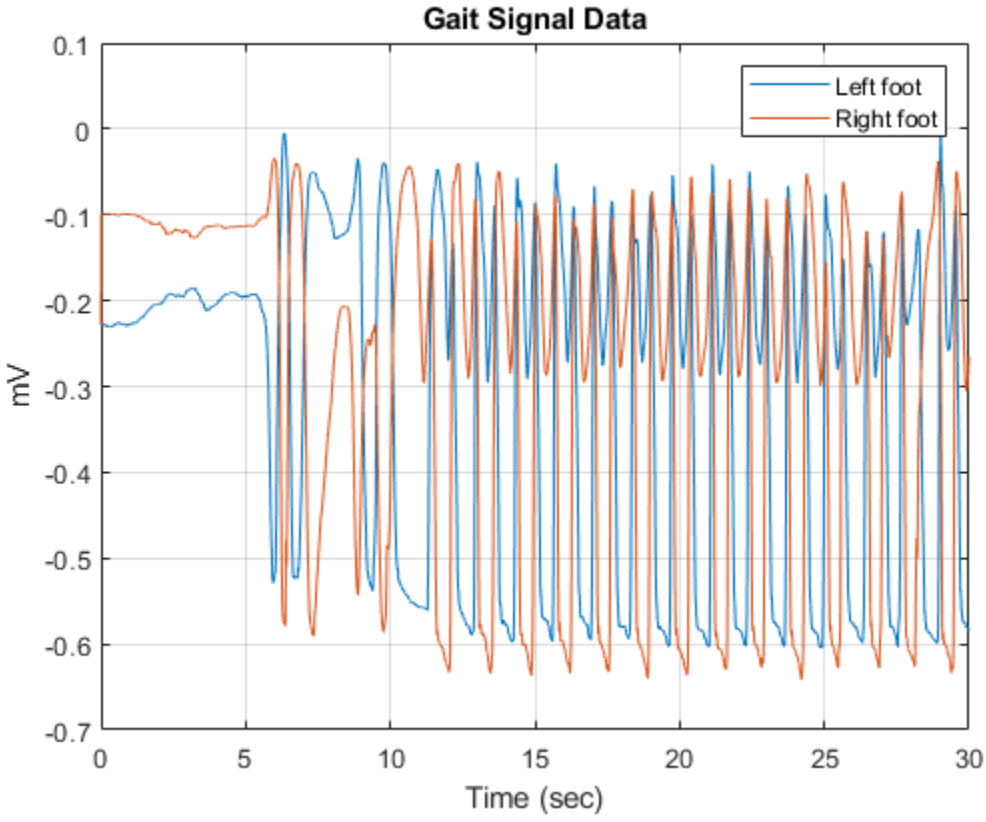
This example shows how to use the functions `midcross` and `dtw` to extract features from gait signal data. Gait signals are used to study the walking patterns of patients with neurodegenerative disease. The time between strides has been reported to differ between healthy and sick individuals. `midcross` offers a convenient way to calculate these times. People also change their walking speed over time. `dtw` provides a convenient way to quantitatively compare the shape of gait signals by warping to align them in time. This example uses `midcross` to locate each step in a gait signal and `dtw` to compute distances between gait signal segments. These results are examined as potential features for signal classification. While this example is specific to gait signals, other physiological signals, such as electrocardiogram (ECG) or photoplethysmogram (PPG), can also be analyzed using these functions.

### Measure Inter-Stride Time Intervals

The dataset being analyzed contains force data collected during walking for patients with Amyotrophic Lateral Sclerosis (ALS) and a control group. ALS is a disease made famous by Lou Gehrig, Stephen Hawking, and the 2014 'Ice Bucket Challenge'.

Load and plot the first 30 seconds of gait signal data for one patient.

```
helperGaitPlot('als1m');
xlim([0 30])
```

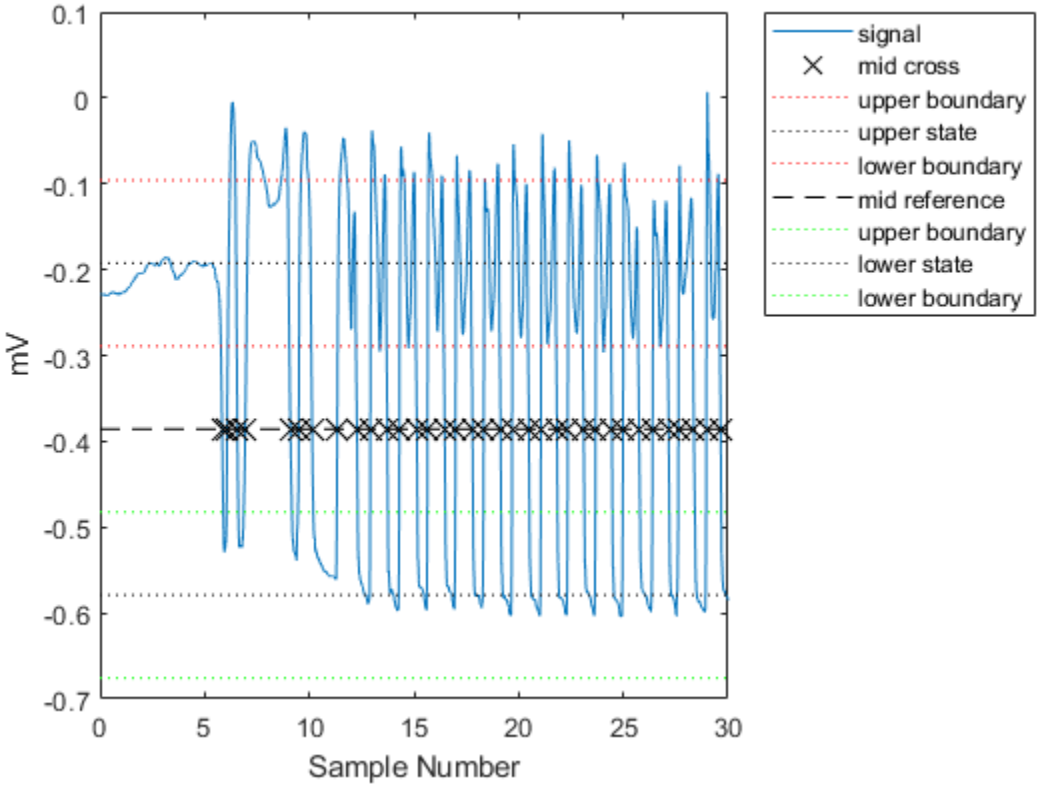



This dataset represents the force exerted by a foot on a force sensitive resistor. The force is measured in millivolts. Each record is one minute in length and contains separate channels for the left and right

foot of a subject. Each step in the dataset is characterized by a sharp change in force as the foot impacts and leaves the ground. Use `midcross` to find these sharp changes for an ALS patient.

Use `midcross` to find and plot the location of each crossing for the left foot of an ALS patient. Chose a tolerance of 25% to ensure that every crossing is detected.

```
Fs = 300;
gaitSignal = helperGaitImport('als1m');
midcross(gaitSignal(1,:),Fs,'tolerance',25);
xlim([0 30])
xlabel('Sample Number')
ylabel('mV')
```

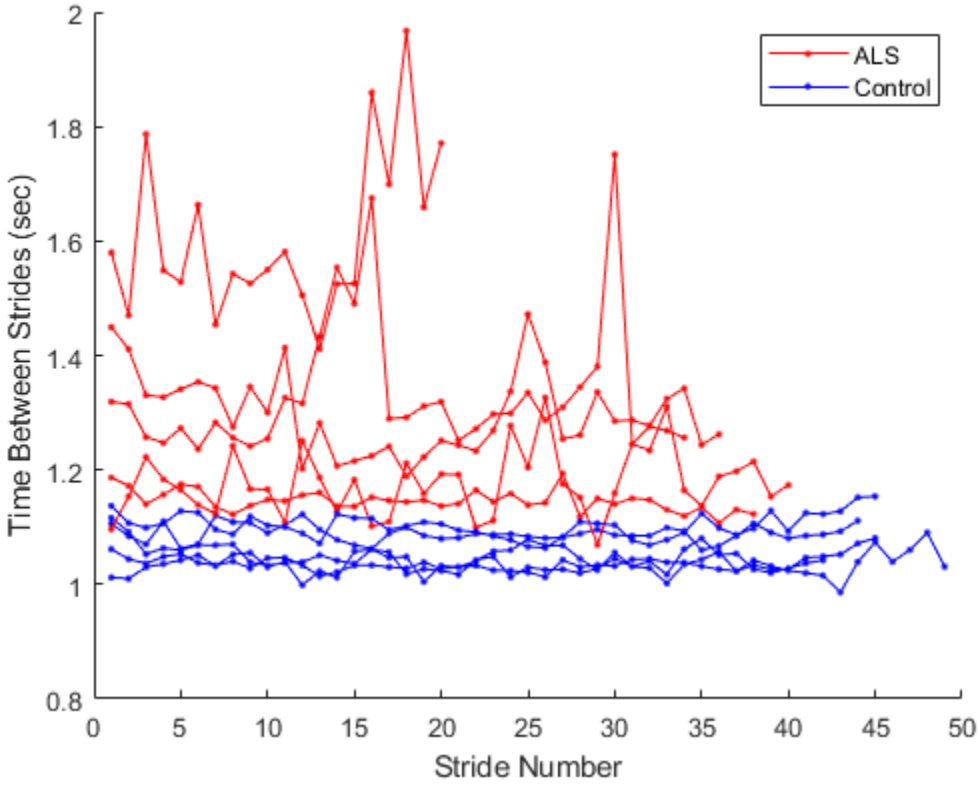



`midcross` correctly identifies the crossings. Now use it to calculate the inter-stride times for a group of ten patients. Five patients are control subjects, and five patients have ALS. Use the left foot record for each patient and exclude the first eight crossings to remove transients.

```
pnames = helperGaitImport();
for i = 1:10
  gaitSignal = helperGaitImport(pnames{i});
  IND2 = midcross(gaitSignal(1,:),Fs,'Tolerance',25);
  IST{i} = diff(IND2(9:2:end));
  varIST(i) = var(IST{i});
end
```

Plot the inter-stride times.

```
figure
hold on
for i = 1:5
  plot(1:length(IST{i}),IST{i},'.-r')
  plot(1:length(IST{i+5}),IST{i+5},'.-b')
end
xlabel('Stride Number')
ylabel('Time Between Strides (sec)')
legend('ALS','Control')
```




The variance of the inter-stride times is higher overall for the ALS patients.

### Measure Similarity of Walking Patterns

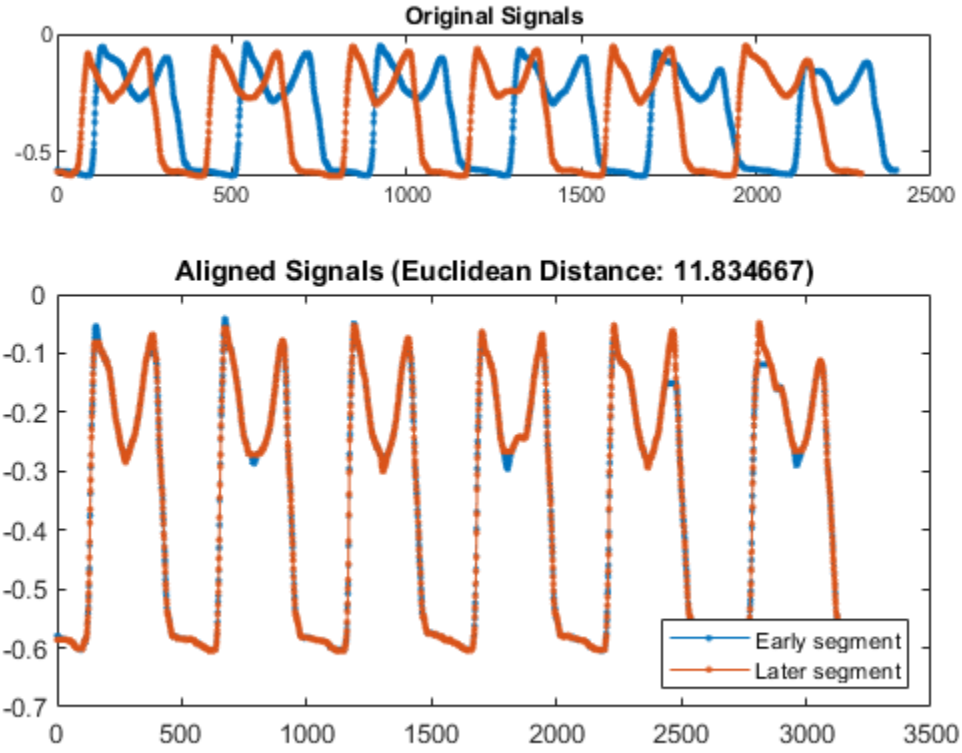
Having quantified the distance between steps, proceed to analyze the shape of the gait signal data independent of these inter-step variations. Compare two segments of the signal using `dtw`. Ideally, one would compare the shape of the gait signal over time as treatment or disease progresses. Here, we compare two segments of the same record, one segment taken early in the recording (`sigsInitialLeft`), and the second towards the end (`sigsFinalLeft`). Each segment contains six steps.

Load the gait signal data segments.

```
load PNGaitSegments.mat
```

The patient does not walk at the same rate throughout the record. `dtw` provides a measure of the distance between segments by warping them to align them in time. Compare the two segments using `dtw`.

```
figure
dtw(sigsInitialLeft{1},sigsFinalLeft{1});
legend('Early segment','Later segment','location','southeast')
```




The two segments are aligned in time. Although the step rate of the patient appears to change over time, as can be seen in the offset of the original signals, `dtw` matches the two segments by allowing samples of either segment to repeat. The distance via `dtw`, along with the variance of the inter-stride times, will be explored as features for a gait signal classifier.

### Construct a Feature Vector to Classify Signals

Suppose you are building a classifier to decide whether or not a patient is healthy based on a gait signals. Investigate the variance of inter-stride times, `feature1`, and the distance via `dtw` between initial and final signal segments, `feature2`, as classification features.

Feature 1 was previously computed using `midcross`.

```
feature1 = varIST;
```

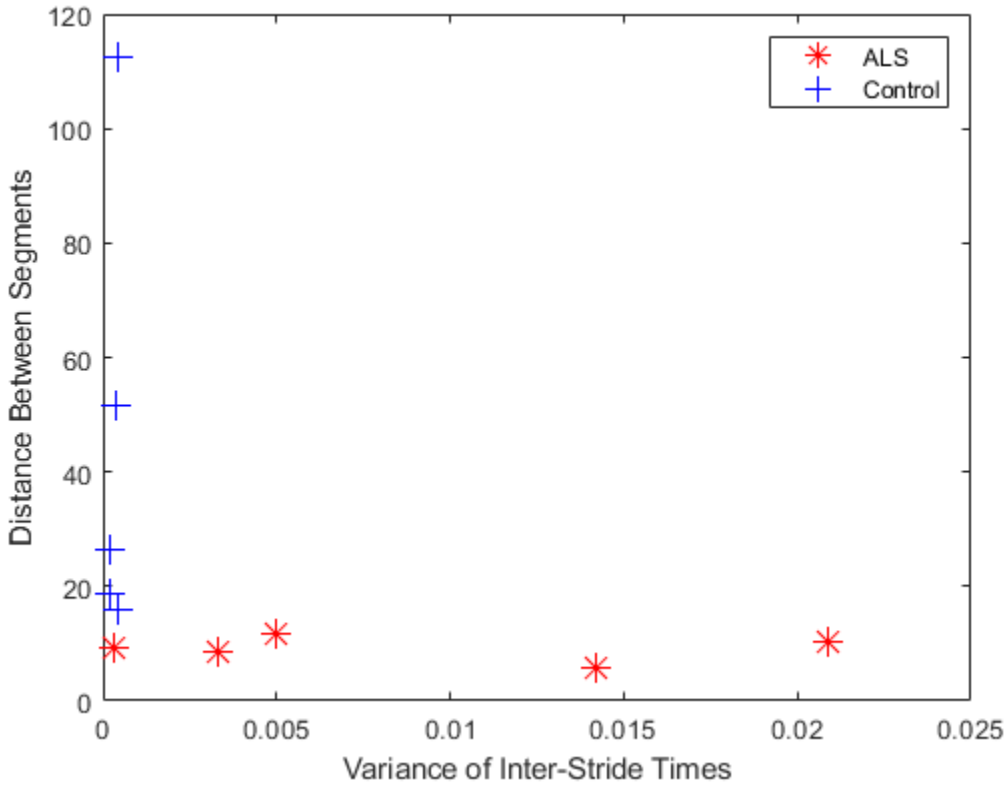
Extract Feature 2 for the ALS patients and the control group.

```
feature2 = zeros(10,1);
for i = 1:length(sigsInitialLeft)
```

```
  feature2(i) = dtw(sigsInitialLeft{i},sigsFinalLeft{i});
end
```

Plot the features for ALS subjects and control subjects.

```
figure
plot(feature1(1:5),feature2(1:5),'r*',...
    feature1(6:10),feature2(6:10),'b+',...
    'MarkerSize',10,'LineWidth',1)
xlabel('Variance of Inter-Stride Times')
ylabel('Distance Between Segments')
legend('ALS','Control')
```




ALS patients seem to have a larger variance in their inter-stride times, but a smaller distance via `dtw` between segments. These features compliment each other and can be explored for use in a classifier such as a Neural Network or Support Vector Machine.

### Conclusions

`midcross` and `dtw` provide a convenient way to compare gait signals and other physiological data which repeat irregularly over time due to different rates of motion or activity. In this example, step times were located using `midcross` and segment distances were computed using `dtw`. These were complimentary measures, as `dtw` removed any time variation that `midcross` distances would measure. As features, these two metrics showed separation between control and ALS patients for this dataset. `midcross` and `dtw` could likewise be used to examine other physiological signals whose shape varies as a function of activity.

### References

[1] Goldberger, A. L., L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, R. G. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." Circulation. Vol. 101, Number 23, 2000, pp. e215-e200.

## See Also

`dtw` | `midcross`

# Detecting Outbreaks and Significant Changes in Signals

This example shows how to determine changes or breakouts in signals via cumulative sums and changepoint detection.

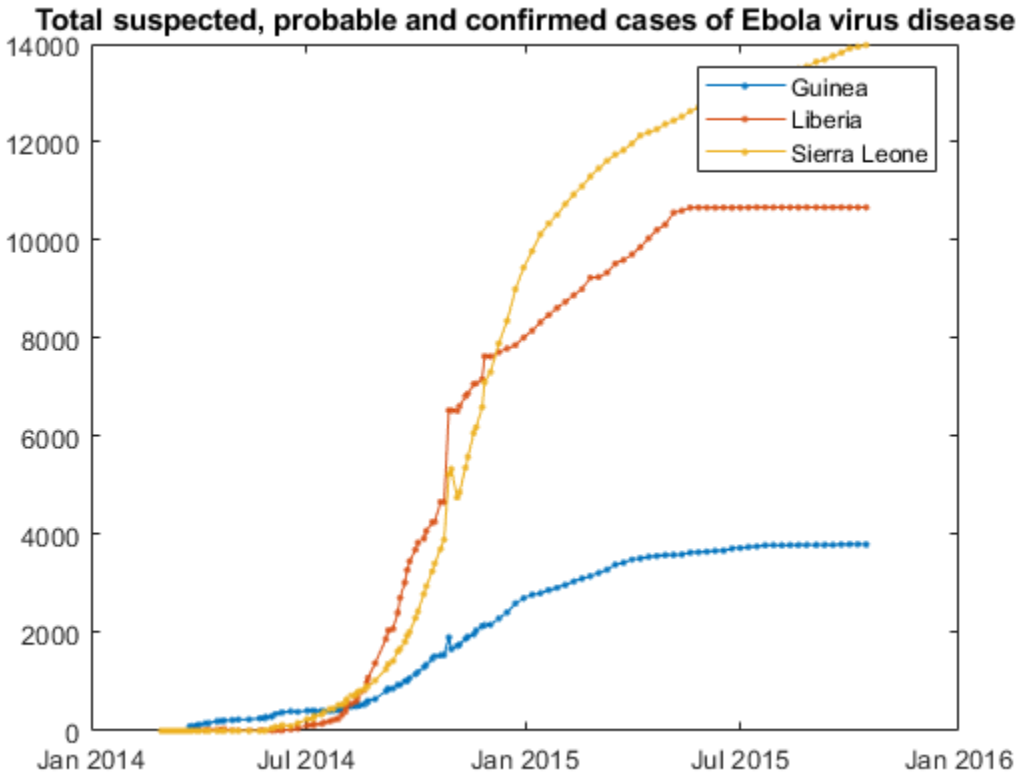
### Detecting Outbreaks via Cumulative Sums

There are many practical applications where you are monitoring data and you want to be alerted as quickly as possible when the underlying process has changed. A very popular technique to achieve this is by means of a cumulative sum (CUSUM) control chart.

To illustrate how CUSUM works, first examine the total reported cases of the 2014 West African Ebola outbreak.

Source: Centers for Disease Control and Prevention

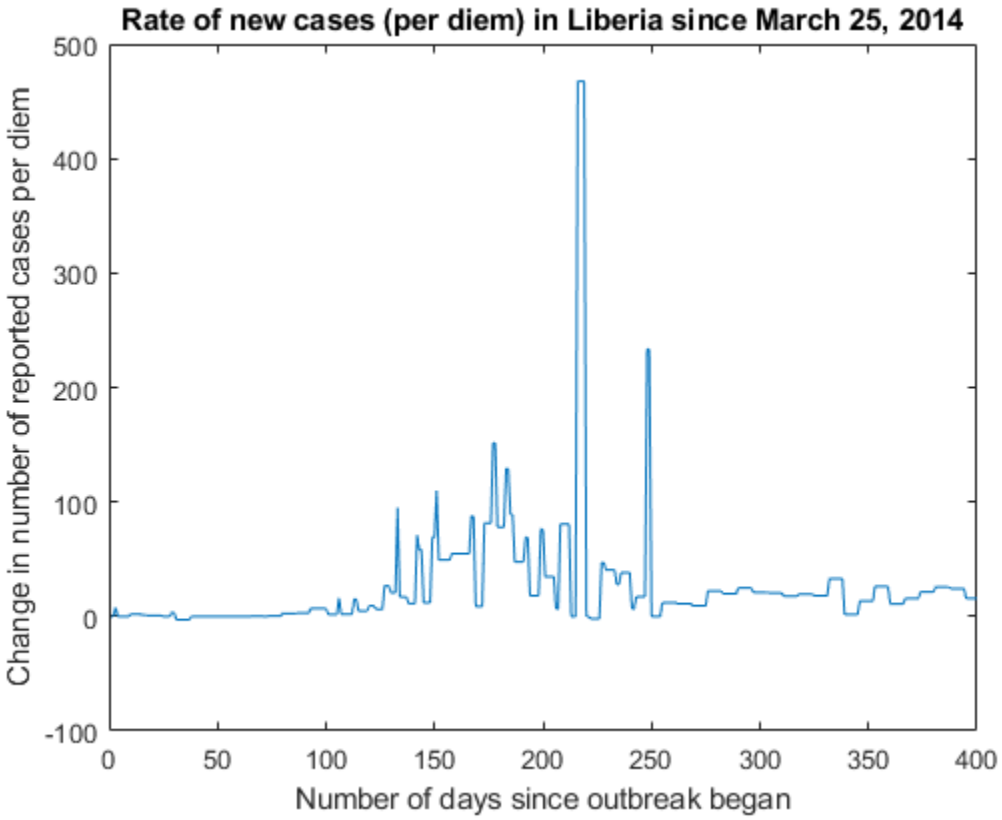
```
load WestAfricanEbolaOutbreak2014
plot(WHOreportdate, [TotalCasesGuinea TotalCasesLiberia TotalCasesSierraLeone],'.-')
legend('Guinea','Liberia','Sierra Leone');
title('Total suspected, probable and confirmed cases of Ebola virus disease');
```




If you look at the leading edge of the first outbreak in Guinea, you can see that the first hundred cases were reported around March 25, 2014, and increase significantly after that date. What is interesting to note is that while Liberia also had a few suspected cases in March, the number of cases stayed relatively in control until about thirty days later.

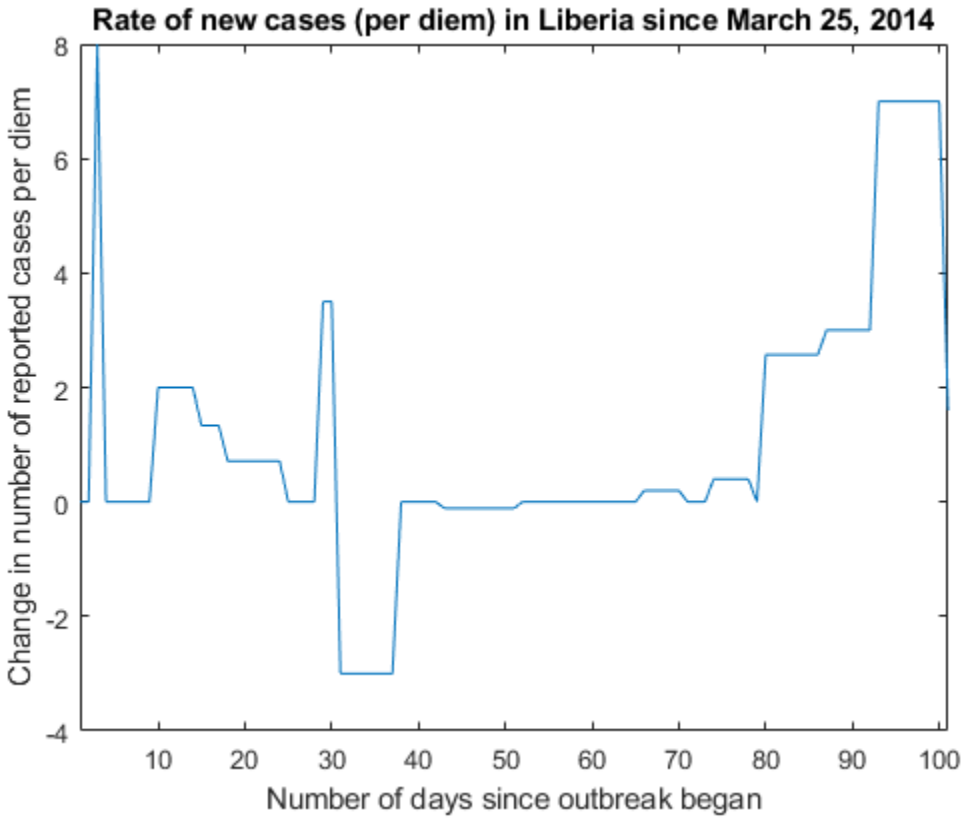
To get a sense of the incoming rate of new patients, plot the relative day-to-day changes in the total number of cases, beginning at the onset on March 25, 2015.

```
daysSinceOutbreak = datetime(2014, 3, 24+(0:400));
cases = interp1(WHOreportdate, TotalCasesLiberia, daysSinceOutbreak);
dayOverDayCases = diff(cases);

plot(dayOverDayCases)
title('Rate of new cases (per diem) in Liberia since March 25, 2014');
ylabel('Change in number of reported cases per diem');
xlabel('Number of days since outbreak began');
```




If you zoom in on the first hundred days of data, you can see that while there was an initial influx of cases, many of them were ruled out after day 30, where rate of changes dropped below zero temporarily. You also see a significant upward trend between days 95 and 100, where a rate of seven new cases per day was reached.

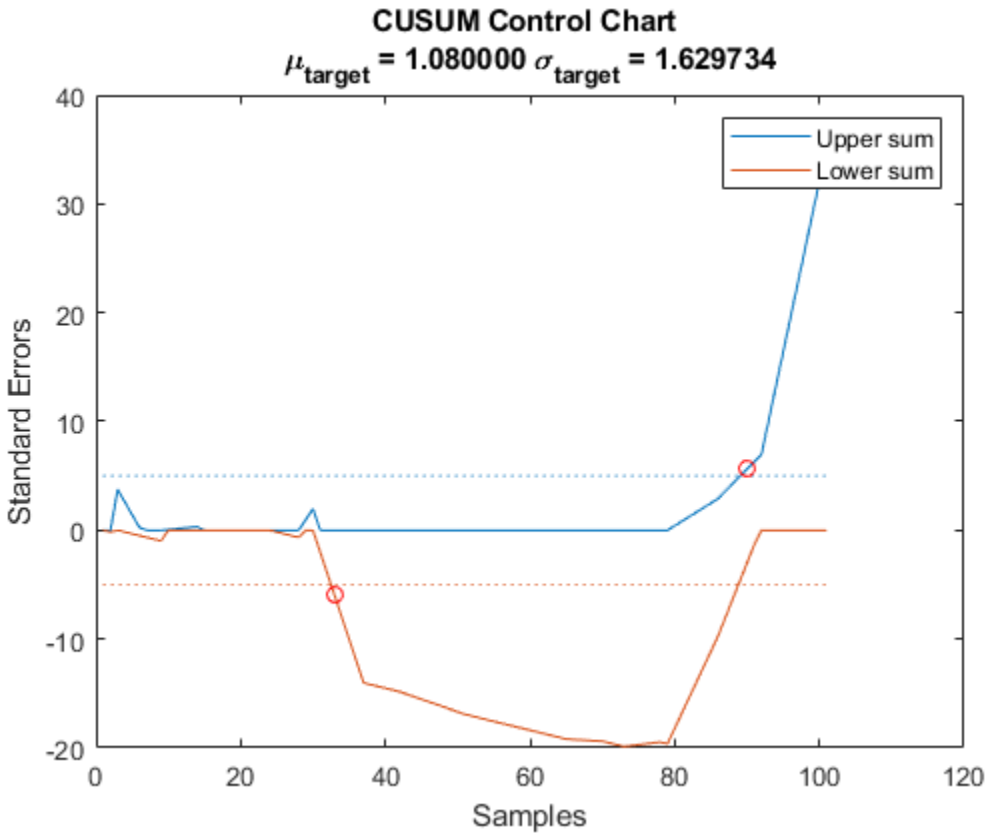
```
xlim([1 101])
```




Performing a CUSUM test on the input data can be a quick way to determine when an outbreak occurs. CUSUM keeps track of two cumulative sums: an upper sum that detects when the local mean shifts upward, and a lower sum that detects when the mean shifts downward. The integration technique provides CUSUM the ability to ignore a large (transient) spike in the incoming rate but still have sensitivity to steadier small changes in rate.

Calling CUSUM with default arguments will inspect the data of the first twenty-five samples and alarm when it encounters a shift in mean more than five standard deviations from within the initial data.

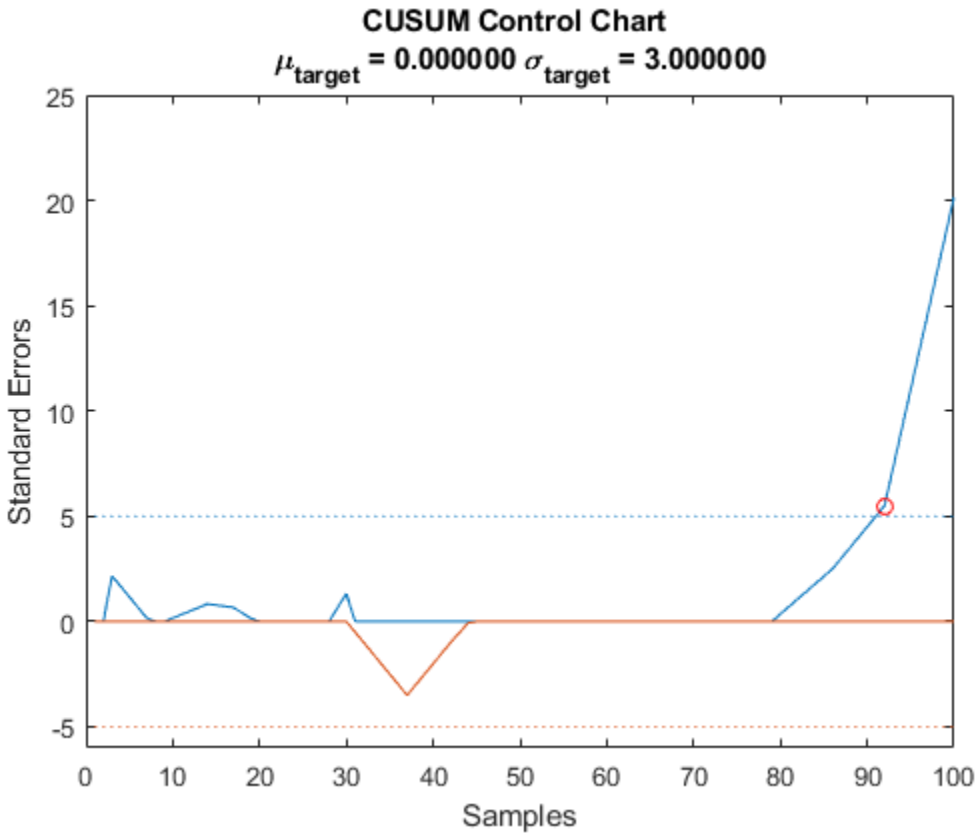
```
cusum(dayOverDayCases(1:101))
legend('Upper sum','Lower sum')
```




Note that CUSUM caught the false reported cases at day 30 (at day 33) and picked up the initial onset of the outbreak starting at day 80 (at day 90). If you compare these results carefully against the previous plot, you can see that CUSUM was able to ignore the spurious uptick at day 29 but still trigger an alarm five days before the large upward trend starting on day 95.

If you adjust CUSUM so that it has a target mean of zero cases/day with a target of plus or minus three cases/day, you can ignore the false alarm at day 30 and pick up the outbreak at day 92:

```
climit = 5;
mshift = 1;
tmean = 0;
tdev = 3;
cusum(dayOverDayCases(1:100),climit,mshift,tmean,tdev)
```

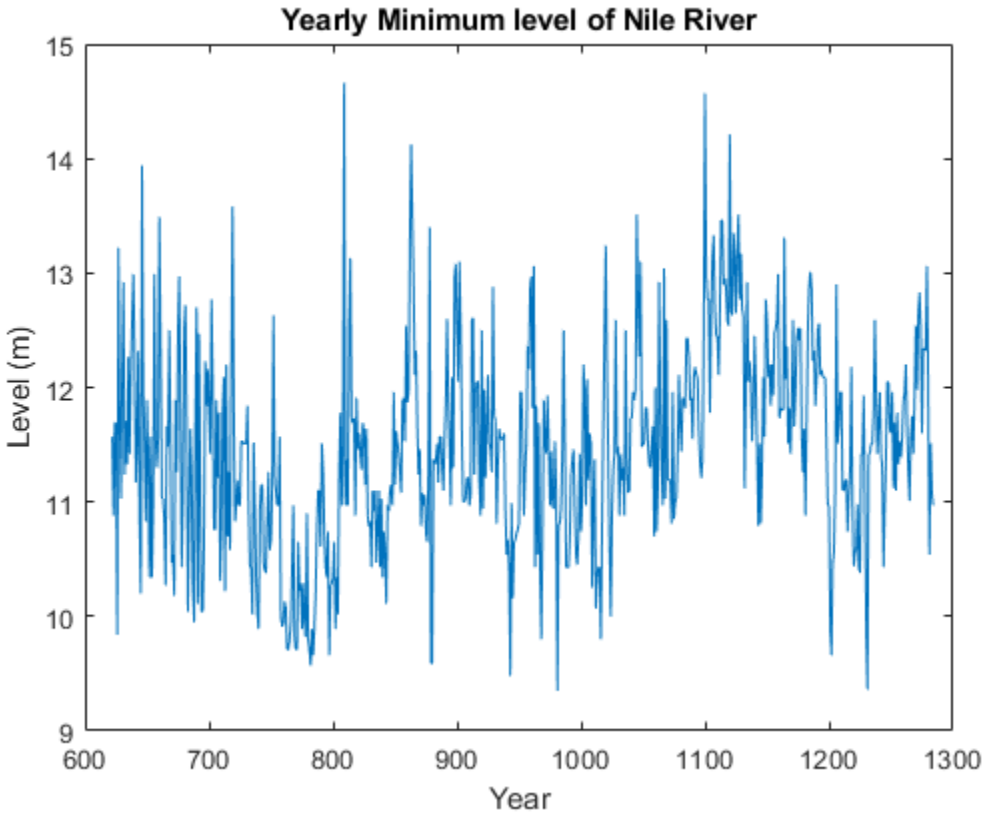



### Finding a Significant Change in Variance

Another method of detecting abrupt changes in statistics is through changepoint detection, which partitions a signal into adjacent segments where a statistic (e.g. mean, variance, slope, etc.) is constant within each segment.

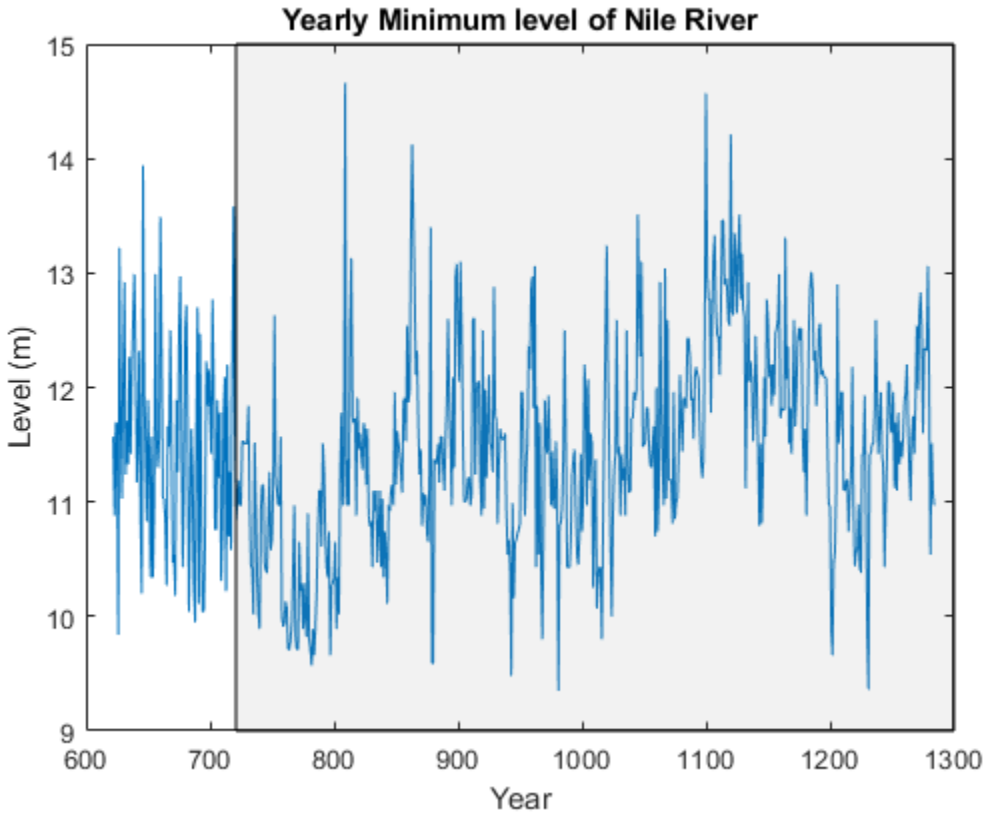
The next example analyzes the yearly minimal water level of the Nile river for the years 622 to 1281 AD measured at the Roda gauge near Cairo.

```
load nilometer
years = 622:1284;
plot(years,nileriverminima)
title('Yearly Minimum level of Nile River')
xlabel('Year')
ylabel('Level (m)')
```




Construction began on a newer more accurate measuring device around 715 AD. Not much is known before this time, but on further examination, you can see that there is considerably less variability after around 722. To find the period of time when the new device became operational, you can search for the best change in the root-mean-square water level after performing element-wise differentiation to remove any slowly varying trends.

```
i = findchangepts(diff(nileriverminima),'Statistic','rms');

ax = gca;
xp = [years(i) ax.XLim([2 2]) years(i)];
yp = ax.YLim([1 1 2 2]);
patch(datenum(xp),yp,[.5 .5 .5],'facealpha',0.1);
```

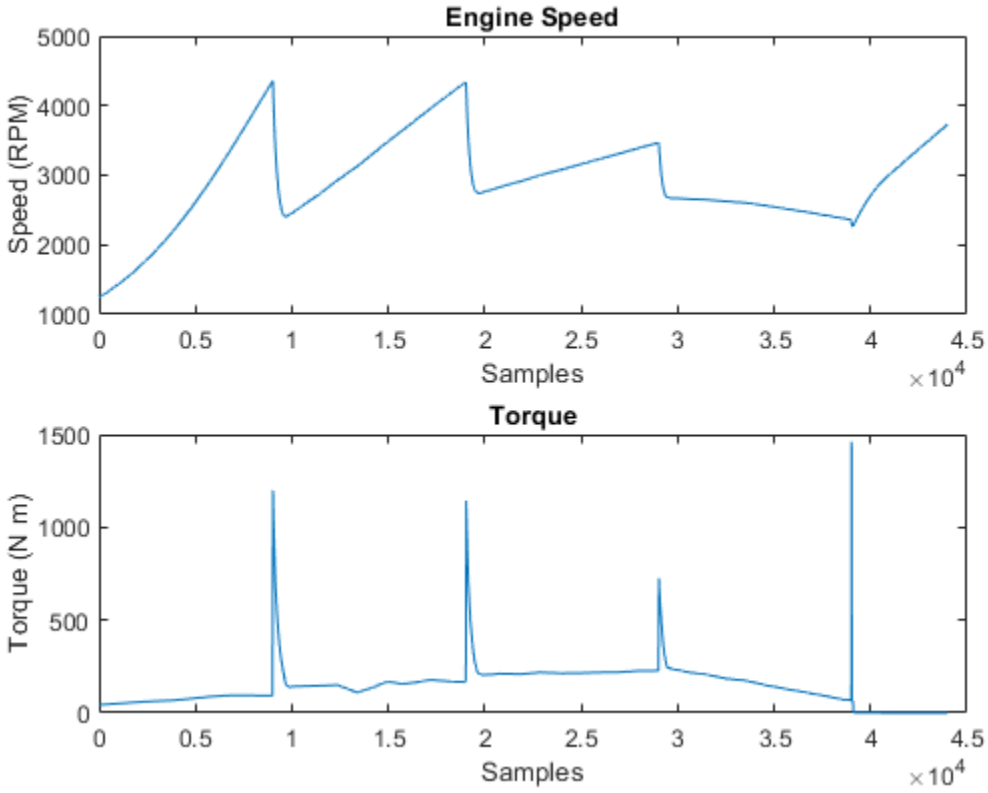



While sample-wise differentiation is a simple method to remove trends, there are other more sophisticated methods to examine variance over larger scales. For an example of how to perform changepoint detection via wavelets using this dataset, see “Wavelet Changepoint Detection” (Wavelet Toolbox).

### Detecting Multiple Changes in an Input Signal

The next example is concerned with a 45 second simulation of a CR-CR 4-speed transmission block, sampled at 1 ms intervals. The simulation data of the car engine RPM and torque are shown below.

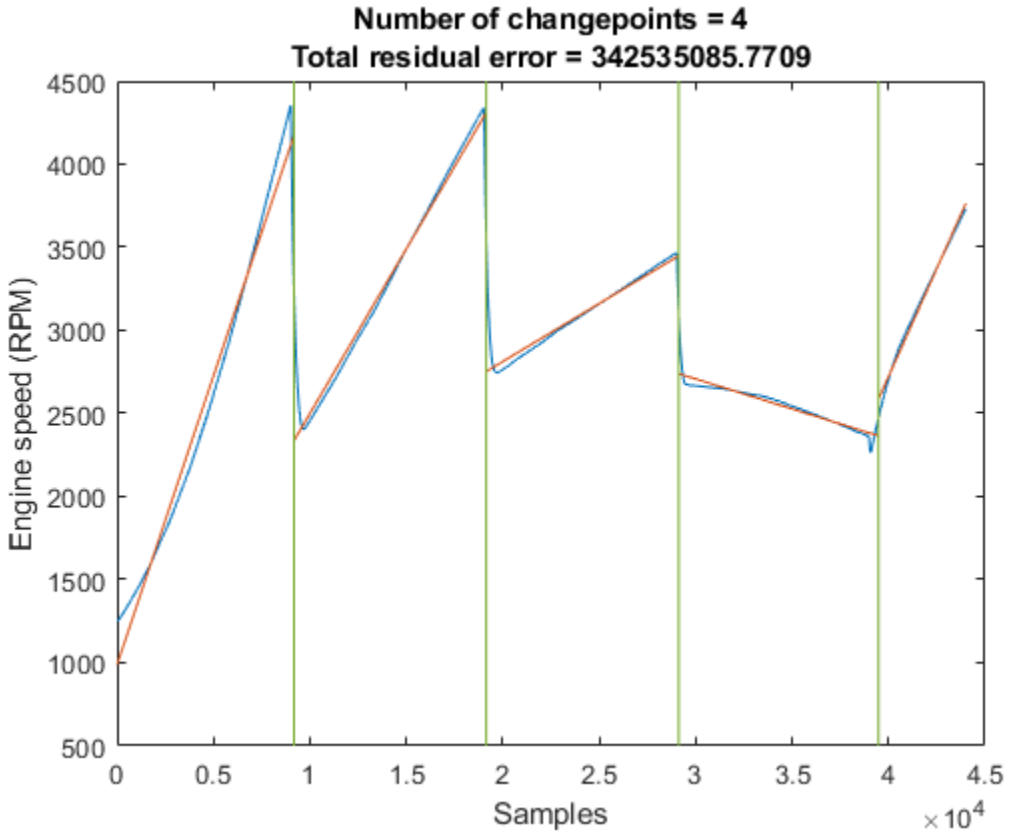
```
load simcarsig

subplot(2,1,2);
plot(carTorqueNM);
xlabel('Samples');
ylabel('Torque (N m)');
title('Torque');

subplot(2,1,1);
plot(carEngineRPM);
xlabel('Samples');
ylabel('Speed (RPM)');
title('Engine Speed');
```




Here the car accelerates, changes gears three times, switches to neutral, and then applies the brake.

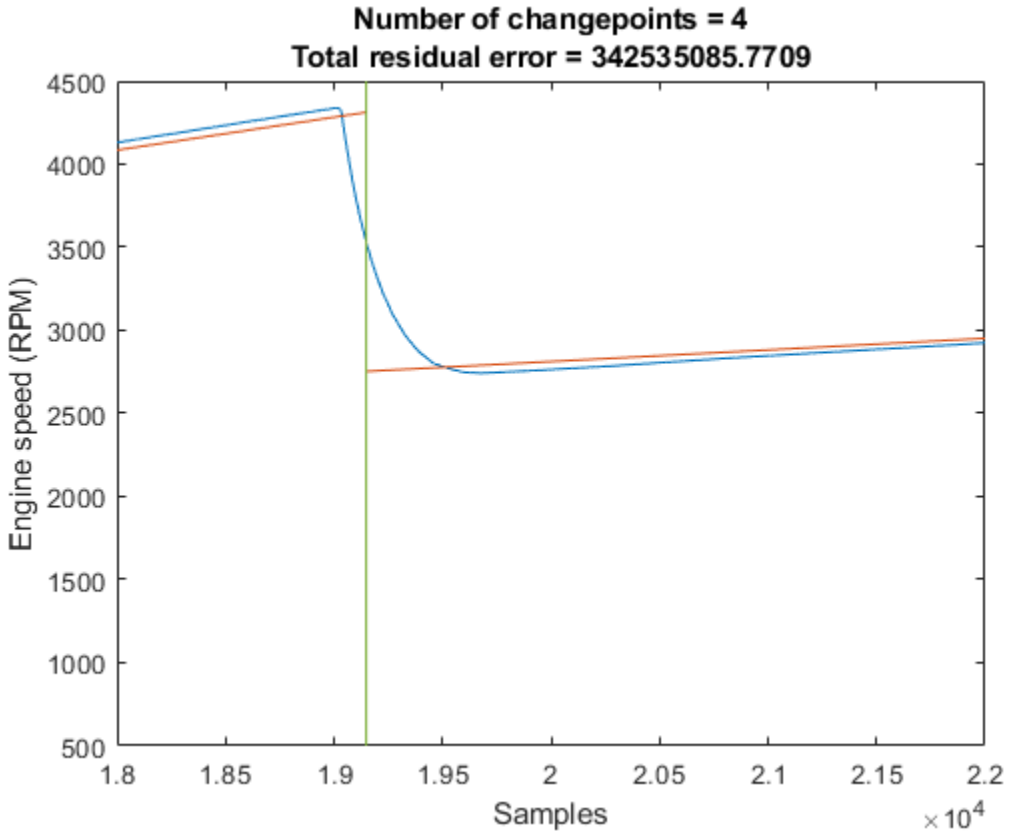
Since the engine speed can be naturally modeled as a series of linear segments, you can use `findchangepts` to find the samples where the car changes gears.

```
figure
findchangepts(carEngineRPM,'Statistic','linear','MaxNumChanges',4)
xlabel('Samples');
ylabel('Engine speed (RPM)');
```




Here you can see four changes (between five linear segments) and that they occurred around the 10,000, 20,000, 30,000, and 40,000 sample mark. Zoom into the idle portion of the waveform:

```
xlim([18000 22000])
```

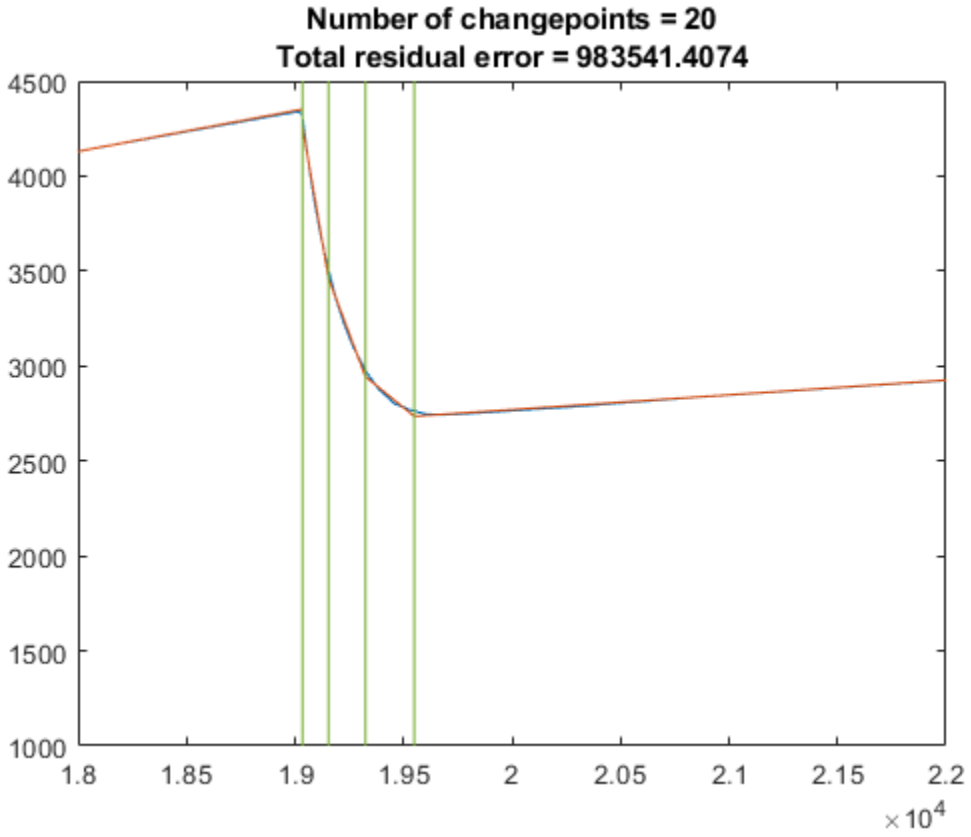



Note that the straight-line fit closely tracks the input waveform, however it can be improved.

**Observing Changes of a Multi-Stage Event Shared Between Signals**

To see the improvement, increase the number of changepoints to 20, and observe the changes within the vicinity of the gear change at sample number 19000

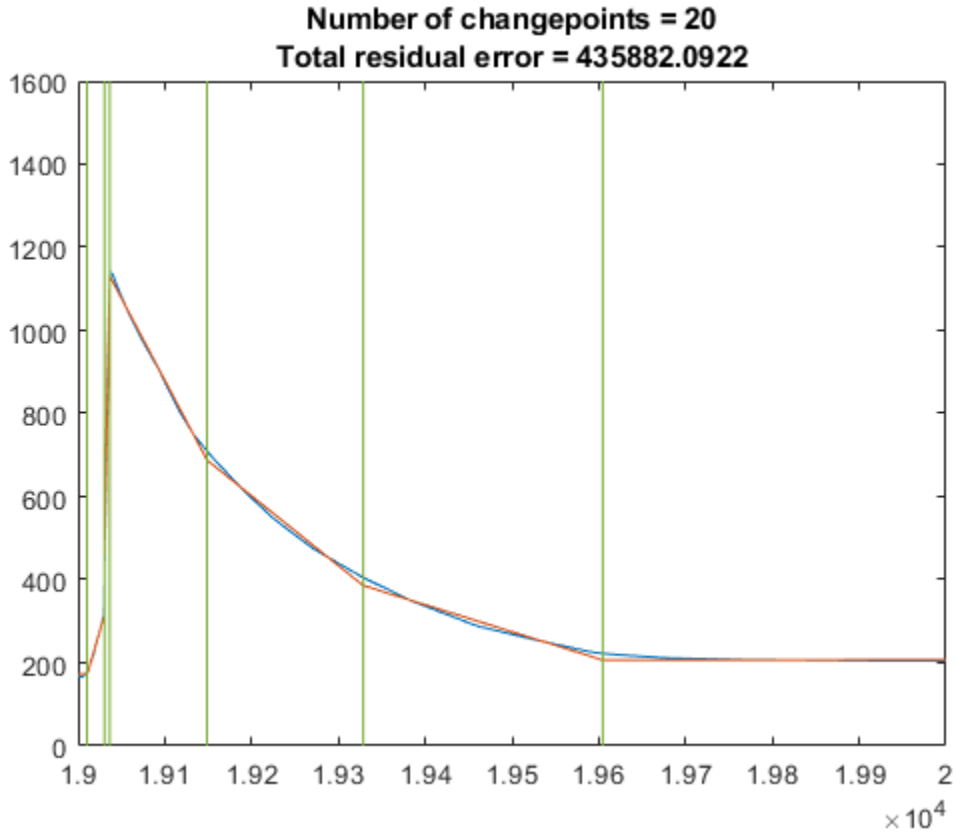
```
findchangepts(carEngineRPM,'Statistic','linear','MaxNumChanges',20)
xlim([18000 22000])
```




Observe that the engine speed started decreasing at sample 19035 and took 510 samples before it settled at sample 19550. Since the sampling interval is 1 ms, this is a ~0.51 s delay and is a typical amount of time after changing gears.

Now look at the changepoints of engine torque within the same region:

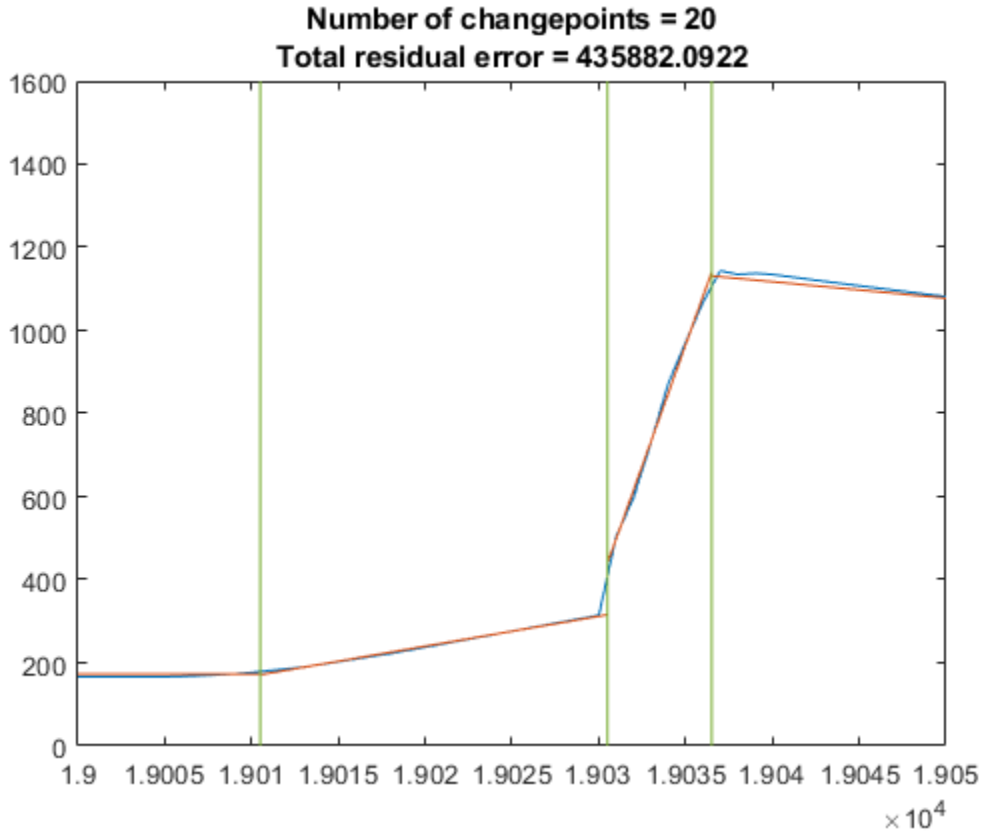
```
findchangepts(carTorqueNM,'Statistic','Linear','MaxNumChanges',20)
xlim([19000 20000])
```




Observe that the engine torque was fully delivered to the axle at sample 19605, 55 milliseconds after the engine speed finished settling. This time is related to the delay between the intake stroke of the engine and torque production.

To find when the clutch became engaged you can zoom further into the signal.

```
xlim([19000 19050])
```




The clutch was depressed at sample 19011 and took about 30 samples (milliseconds) to become completely disengaged.

## See Also

`cusum` | `findchangepts` | `interp1`

# Finding a Signal in Data

This example shows how to use `findsignal` to find a time-varying signal in your data. It includes examples of how to find exact and closely matching signals by using a distance metric, how to compensate for a slowly varying offset, and the use of dynamic time-warping to allow for variations in sampling.

### Finding Exact Matches

When you wish to find *numerically exact* matches of a signal, you can use `strfind` to perform the matching.

For example, if we have a vector of data:

```
data = [1 4 3 2 55 2 3 1 5 2 55 2 3 1 6 4 2 55 2 3 1 6 4 2];
```

and we want to find the location of the signal:

```
signal = [55 2 3 1];
```

we can use `strfind` to find the starting indices of where the signal exists in the data so long as the signal and data are numerically exact.

```
iStart = strfind(data,signal)
```

```
iStart = 1×3

     5    11    18
```

### Finding the Closest Matching Signal

`strfind` works well for *numerically exact* matches. However, this approach fails when there may be errors due to quantization noise or other artifacts in your signal.

For example, if you have a sinusoid:

```
data = sin(2*pi*(0:25)/16);
```

and you want to find the location of the signal:

```
signal = cos(2*pi*(0:10)/16);
```

`strfind` is unable to locate the sinusoid in the data which starts at the fifth sample:

```
iStart = strfind(data,signal)
```

```
iStart =

     []
```

`strfind` cannot find the signal in the data because, due to round-off error, not all values are numerically equal. To see this, subtract the data from the signal in the matching region.

```
data(5:15) - signal
```

```
ans = 1×11
10^-15 ×
```

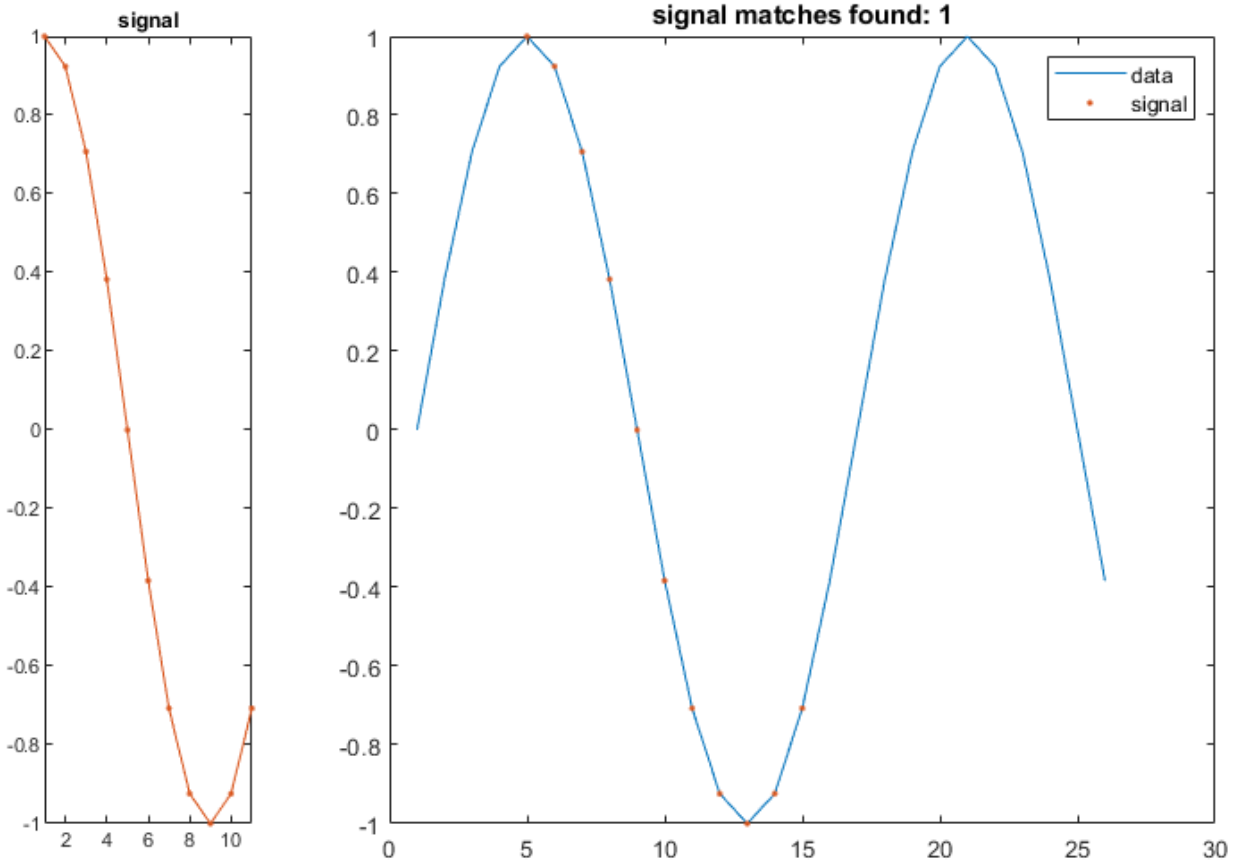
```
         0         0         0    0.0555    0.0612    0.0555         0    0.2220         0    0.2
```

There are numerical differences on the order of 1e-15.

To remedy this, you can use `findsignal`, which by default sweeps the signal across the data and computes the sum of the squared differences between the signal and data locally at each location, looking for the lowest sum.

To produce a plot of the signal and data where the best matching location is highlighted, you can call `findsignal` as follows:

```
findsignal(data,signal)
```

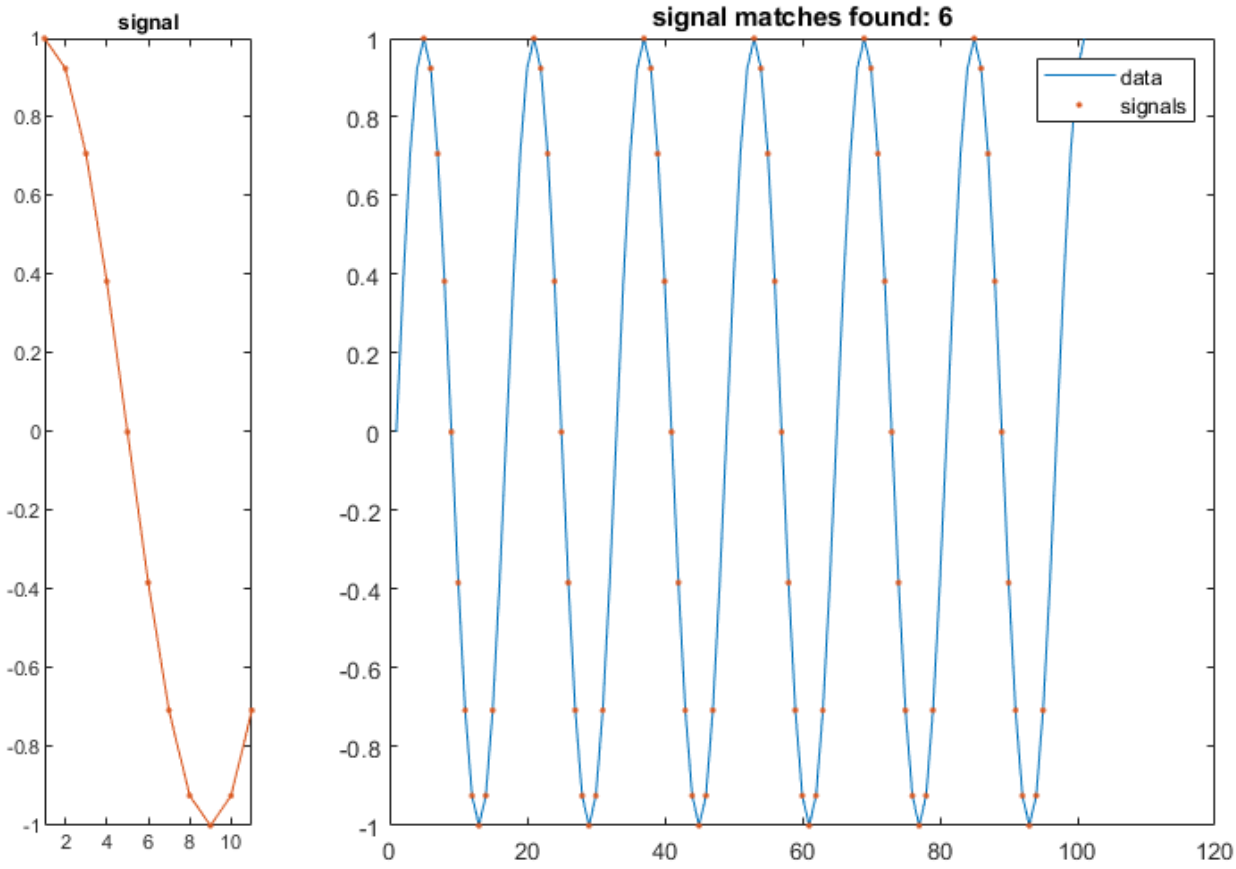



### Finding the Closest Matches under a Threshold

By default `findsignal` always returns the closest match of the signal with the data. To return multiple matches, you can specify a bound on the maximum sum squared difference.

```
data = sin(2*pi*(0:100)/16);
signal = cos(2*pi*(0:10)/16);
```

```
findsignal(data,signal,'MaxDistance',1e-14)
```




`findsignal` returns matches in sorted order of closeness

```
[iStart, iStop, distance] = findsignal(data,signal,'MaxDistance',1e-14);
fprintf('iStart iStop  total squared distance\n')
```

```
iStart iStop  total squared distance
```

```
fprintf('%4i %5i     %.7g\n',[iStart; iStop; distance])
```

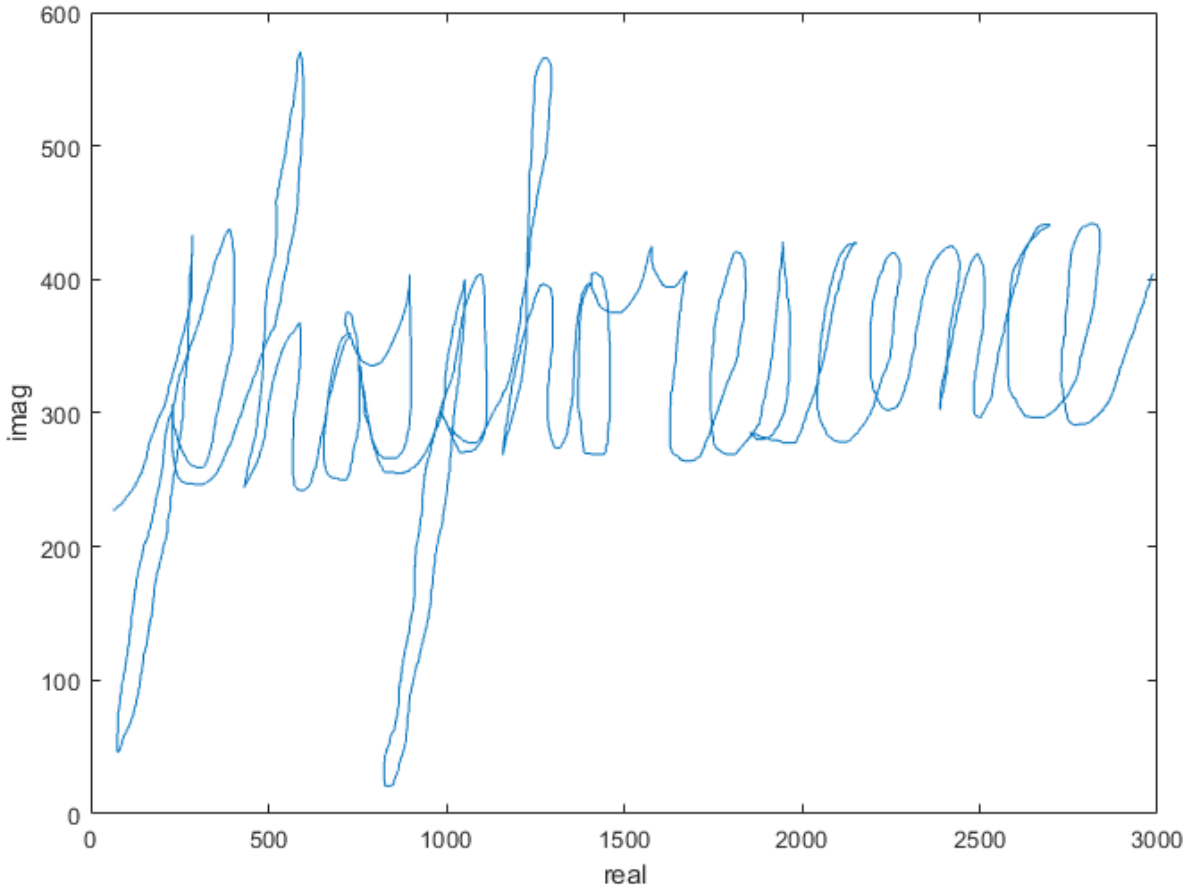
```
  37    47     0
  69    79     0
  85    95     0
   5    15     1.776357e-15
  21    31     1.776357e-15
  53    63     1.776357e-15
```

### Searching for a Complex Signal Trajectory with a Varying Offset

This next example shows how to use `findsignal` to find a signal that traces a known trajectory. The file "cursiveex.mat" contains a recording of the x- and y- position of the tip of a pen as it traced out

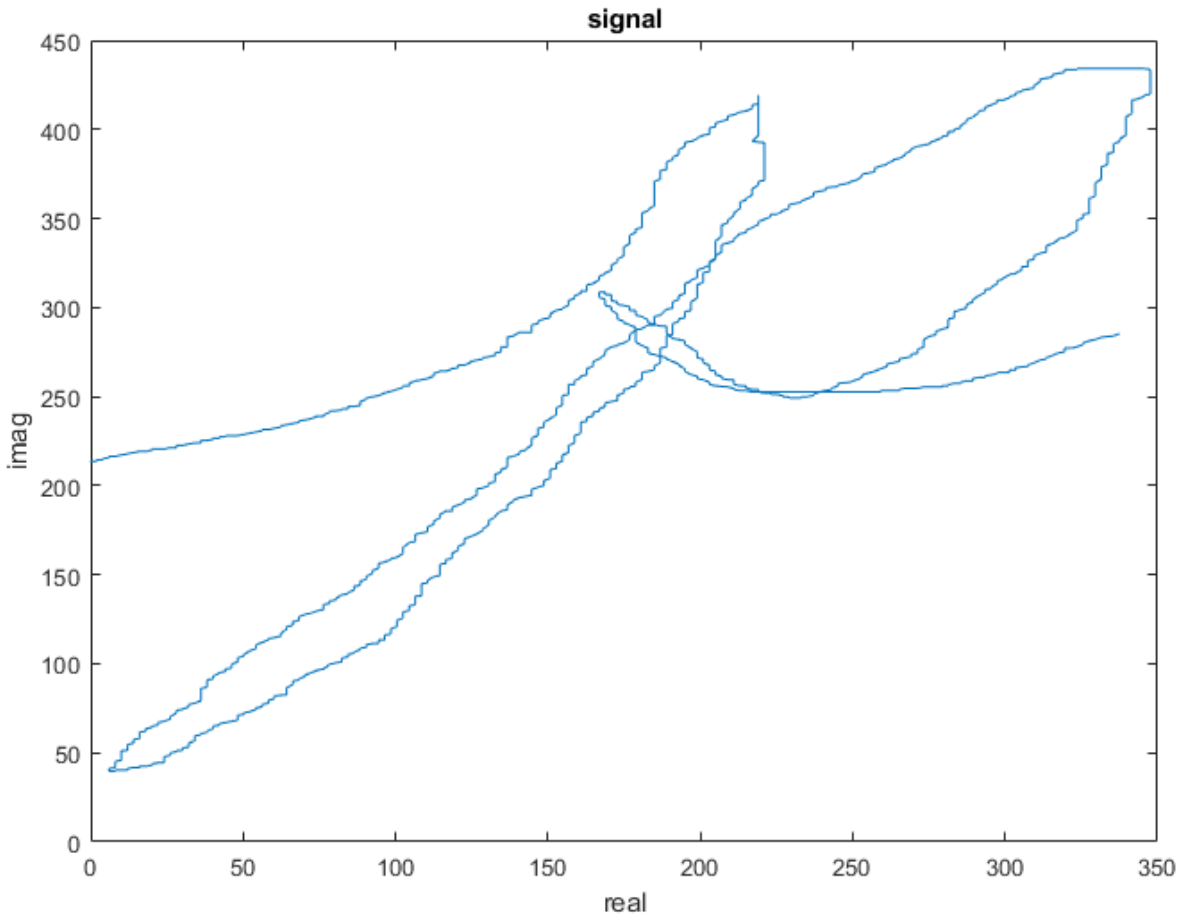
the word "phosphorescence" on a piece of paper. The x,y data is encoded as the real and imaginary components of a complex signal, respectively.

```
load cursiveex
plot(data)
xlabel('real')
ylabel('imag')
```



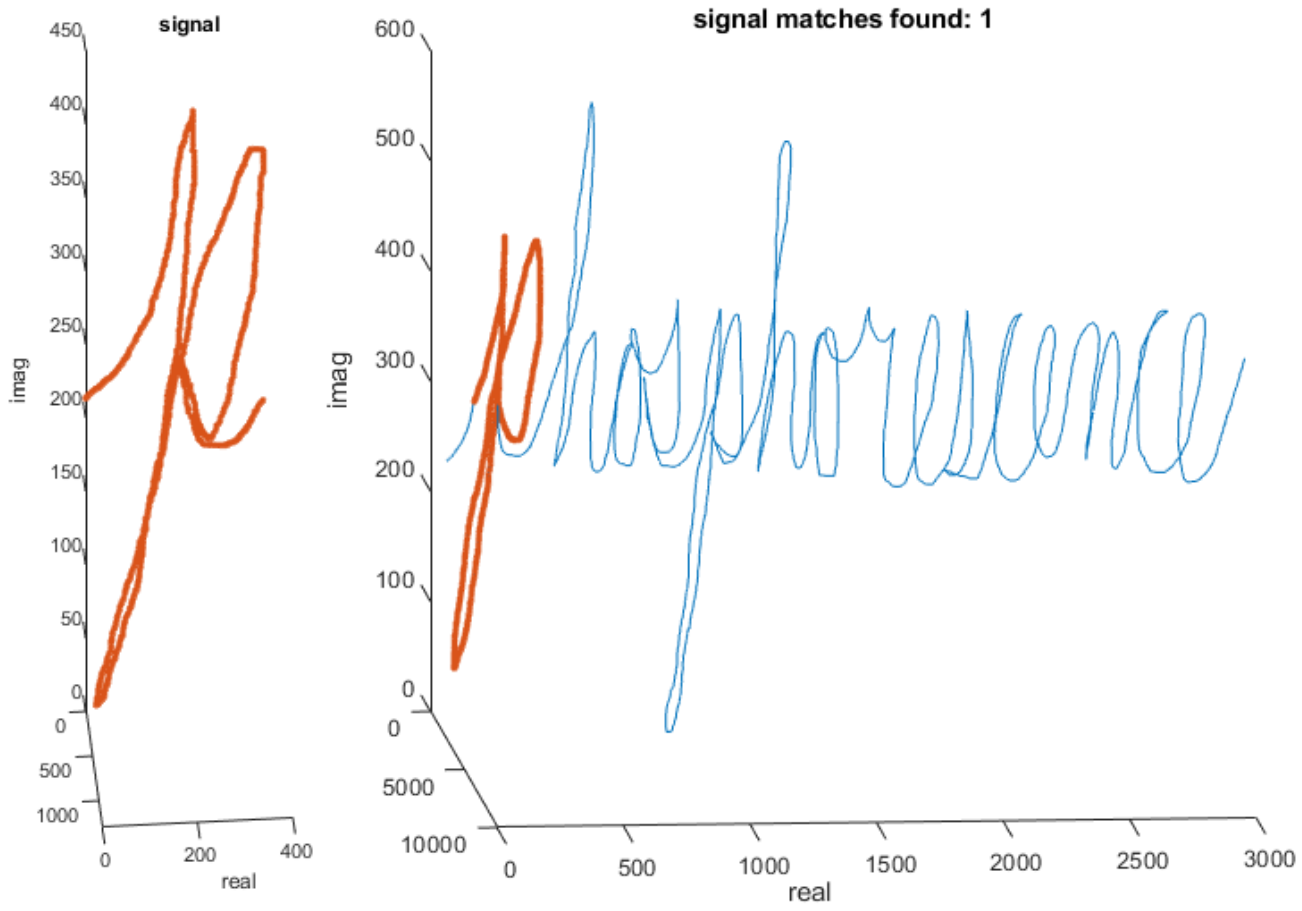

The same writer traced out a letter "p" as a template signal.

```
plot(signal)
title('signal')
xlabel('real')
ylabel('imag')
```




You can find the first "p" in the data fairly easily using `findsignal`. This is because values of the signal line up fairly well at the beginning of the data.

```
findsignal(data,signal)
```

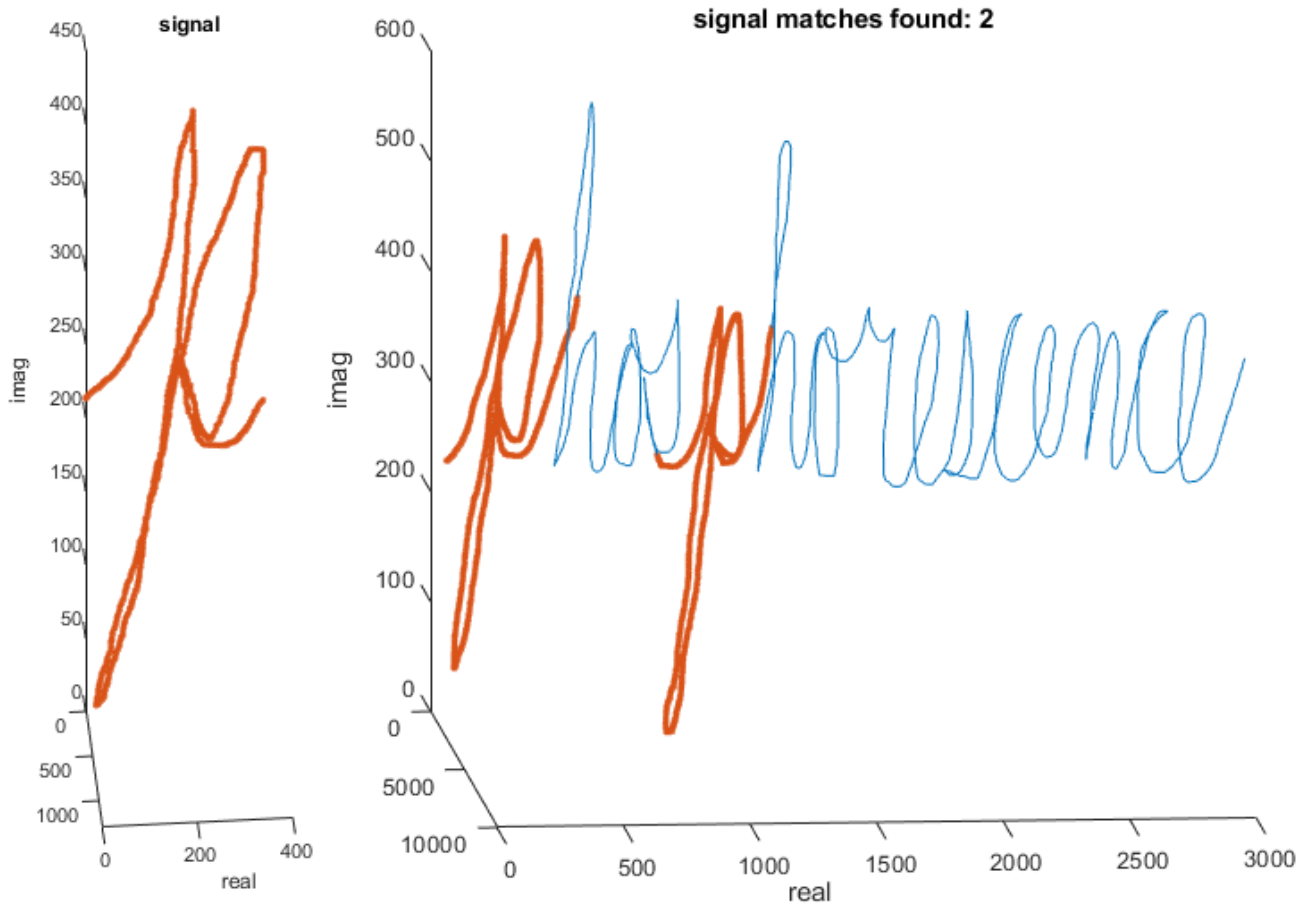



However, the second "p" has two characteristics that make it difficult for `findsignal` to identify: It has a significant but constant offset from the first letter, and parts of the letter were drawn at a different rate of speed than the template signal.

If you are interested in just matching the overall shape of the letter, you can subtract off a windowed local mean from both the signal and data element. This allows you to mitigate the effect of constant shifts.

To mitigate the effect of the varying speeds at which the letters are drawn, you can use dynamic time warping, which will stretch either the signal or data to a common time base as it performs the search:

```
findsignal(data,signal,'TimeAlignment','dtw', ...
                'Normalization','center', ...
                'NormalizationLength',600, ...
                'MaxNumSegments',2)
```




### Finding Time-Stretched Power Signals

This next example shows how to use `findsignal` to find the location of a spoken word in a phrase.

The following file contains an audio recording of the phrase: "Accelerating the pace of engineering and science" and a separate audio recording of "engineering" spoken by the same speaker.

```
load slogan
soundsc(phrase,fs)
soundsc(hotword,fs)
```

It is common for the same speaker to vary the pronunciation of individual spoken words in a sentence or phrase. The speaker in this example pronounced "engineering" in two different ways: The speaker took roughly 0.5 seconds to pronounce the word in the phrase, stressing the second syllable ("en-GIN-eer-ing"); the same speaker took 0.75 seconds to pronounce the word in isolation, stressing the third syllable ("en-gin-EER-ing").

To compensate for these local variations in both time and volume, you can use a spectrogram to report the spectral power distribution as it evolves across time.

To get started, use a spectrogram with a fairly coarse frequency resolution. This is done to deliberately blur the narrow-band glottal pulses of the vocal tract, leaving just the wider-band

resonances of the oral and nasal cavities undisturbed. This allows you to lock onto the spoken vowels of a word. Consonants (especially plosives and fricatives) are considerably more difficult to identify using spectrograms. The code below computes a spectrogram

```
Nwindow = 64;
Nstride = 8;
Beta = 64;

Noverlap = Nwindow - Nstride;
[~,~,~,PxxPhrase] = spectrogram(phrase, kaiser(Nwindow,Beta), Noverlap);
[~,~,~,PxxHotWord] = spectrogram(hotword, kaiser(Nwindow,Beta), Noverlap);
```

Now that you have the spectrogram of the phrase and search word, you can use dynamic time warping to account for local variations in word length. Similarly, you can account for variations in power by using power normalization in conjunction with the symmetric Kullback-Leibler distance.

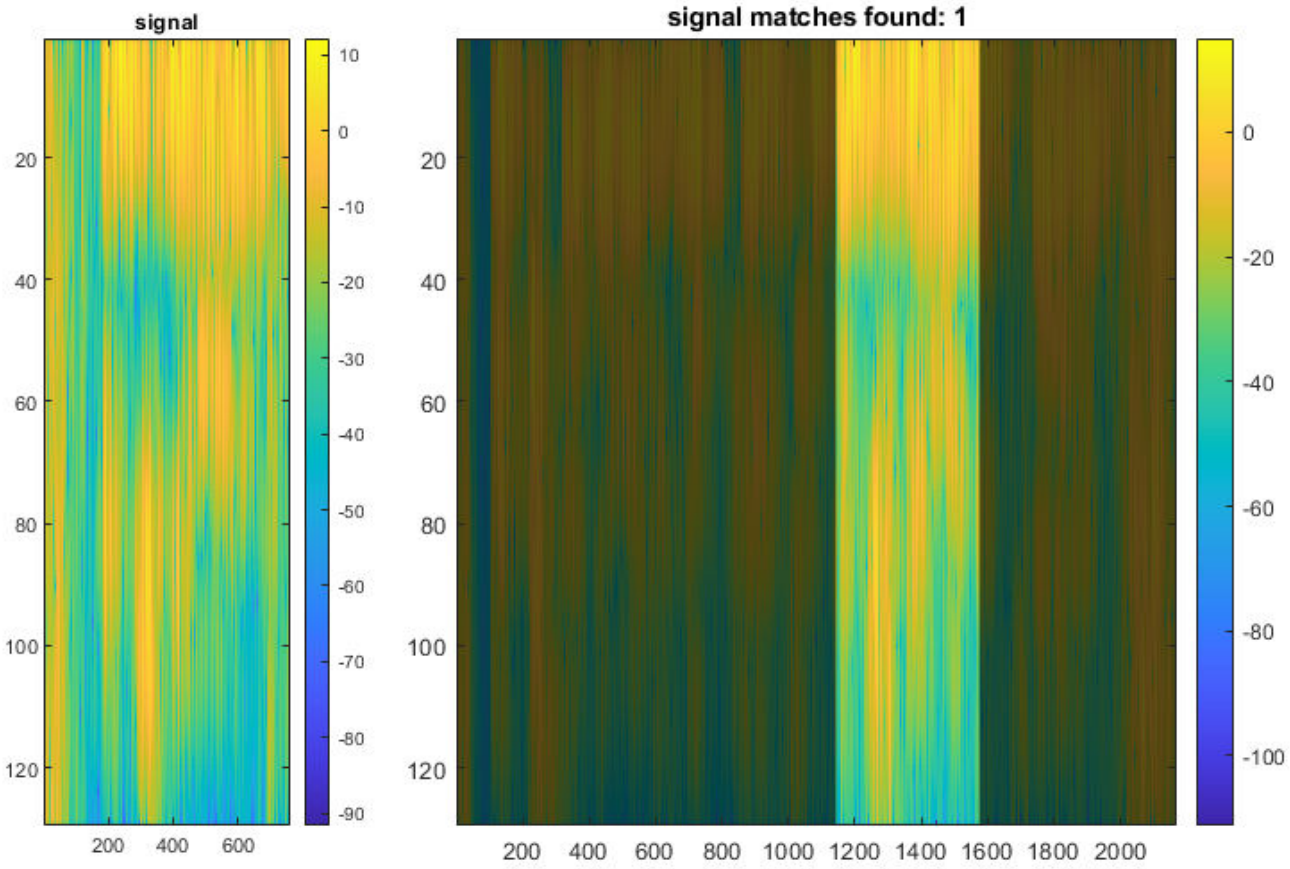
```
[istart,istop] = findsignal(PxxPhrase, PxxHotWord, ...
    'Normalization','power','TimeAlignment','dtw','Metric','symmkl')
```

```
istart = 1144
```

```
istop = 1575
```

Plot and play the identified word.

```
findsignal(PxxPhrase, PxxHotWord, 'Normalization','power', ...
    'TimeAlignment','dtw','Metric','symmkl')
```




```
soundsc(phrase(Nstride*istart-Nwindow/2 : Nstride*istop+Nwindow/2),fs)
```

## See Also

`findsignal`

# Filter Design Gallery

This example shows how to design a variety of FIR and IIR digital filters with the `designfilt` function in the Signal Processing Toolbox® product.

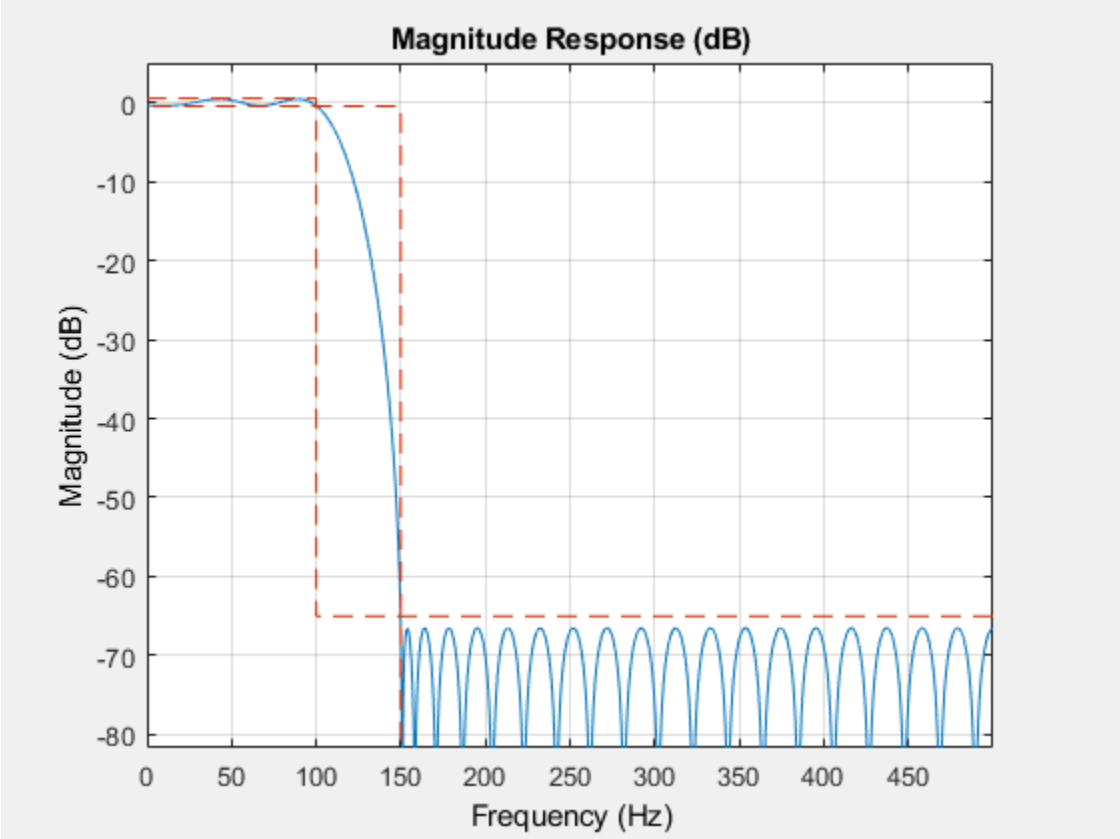
The gallery is designed for you to identify a filter response of interest, view the code, and use it in your own project. It contains examples for each of the available filter responses offered by `designfilt`. Note, however, that these are only a few of the possible ways in which you can design filters for each response type. For an exhaustive list of specification sets, see the Signal Processing Toolbox documentation.

Except when noted otherwise, in this example all frequency units are in hertz, and all ripple and attenuation values are in decibels.

### Lowpass FIR Filters

#### Equiripple Design

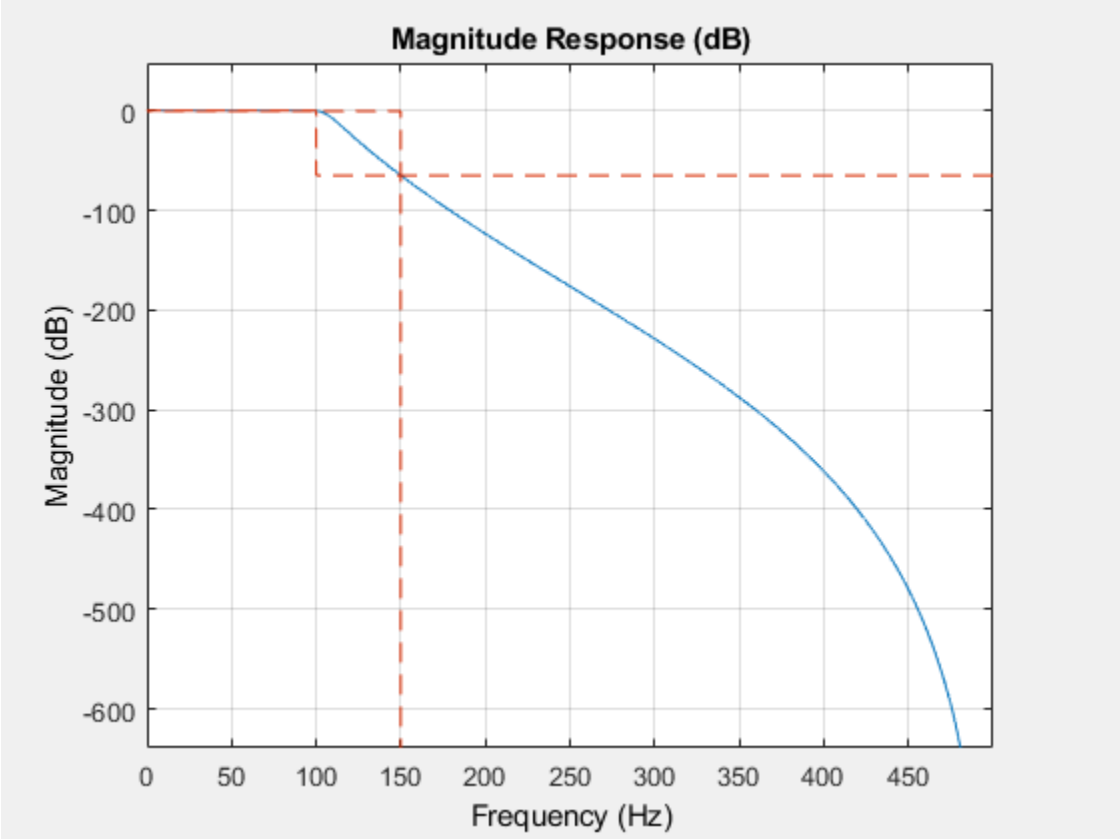
```
Fpass = 100;
Fstop = 150;
Apass = 1;
Astop = 65;
Fs = 1e3;

d = designfilt('lowpassfir', ...
  'PassbandFrequency',Fpass,'StopbandFrequency',Fstop, ...
  'PassbandRipple',Apass,'StopbandAttenuation',Astop, ...
  'DesignMethod','equiripple','SampleRate',Fs);

fvtool(d)
```




**Lowpass IIR Filters**

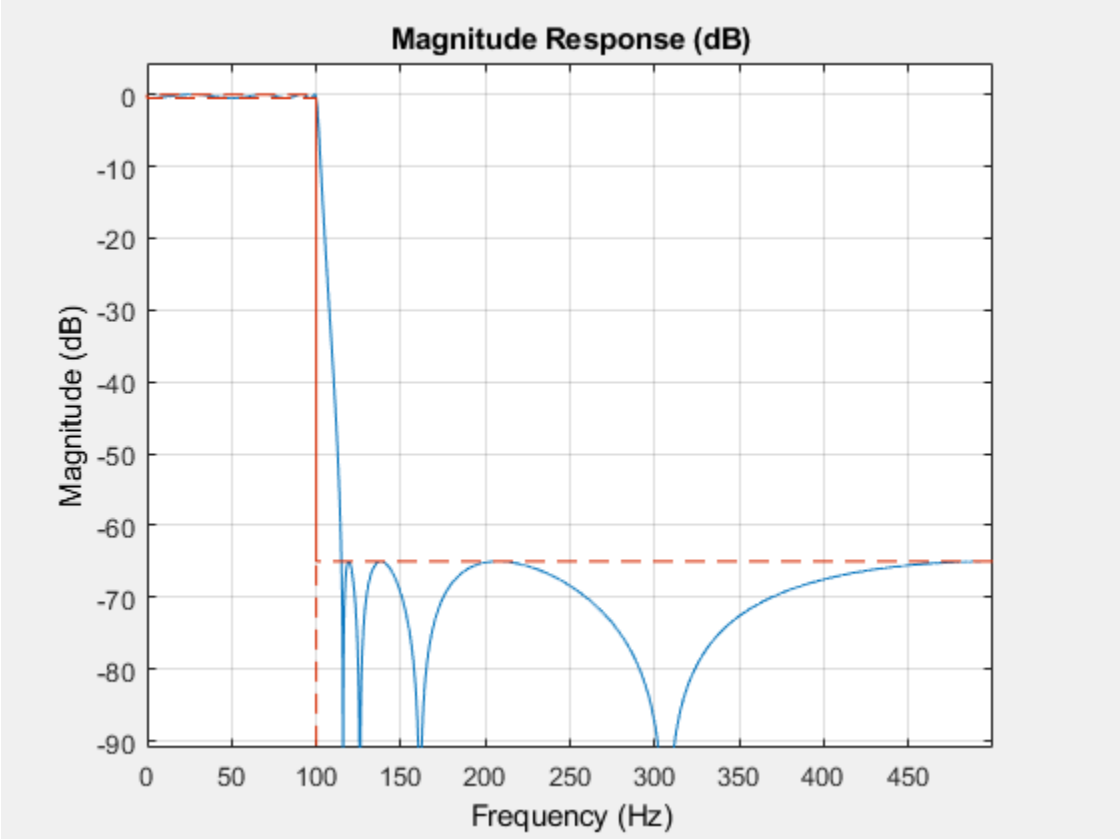
**Maximally Flat Design**

```
Fpass = 100;
Fstop = 150;
Apass = 0.5;
Astop = 65;
Fs = 1e3;

d = designfilt('lowpassiir', ...
  'PassbandFrequency',Fpass,'StopbandFrequency',Fstop, ...
  'PassbandRipple',Apass,'StopbandAttenuation',Astop, ...
  'DesignMethod','butter','SampleRate',Fs);

fvtool(d)
```




**Ripple in Passband and Stopband**

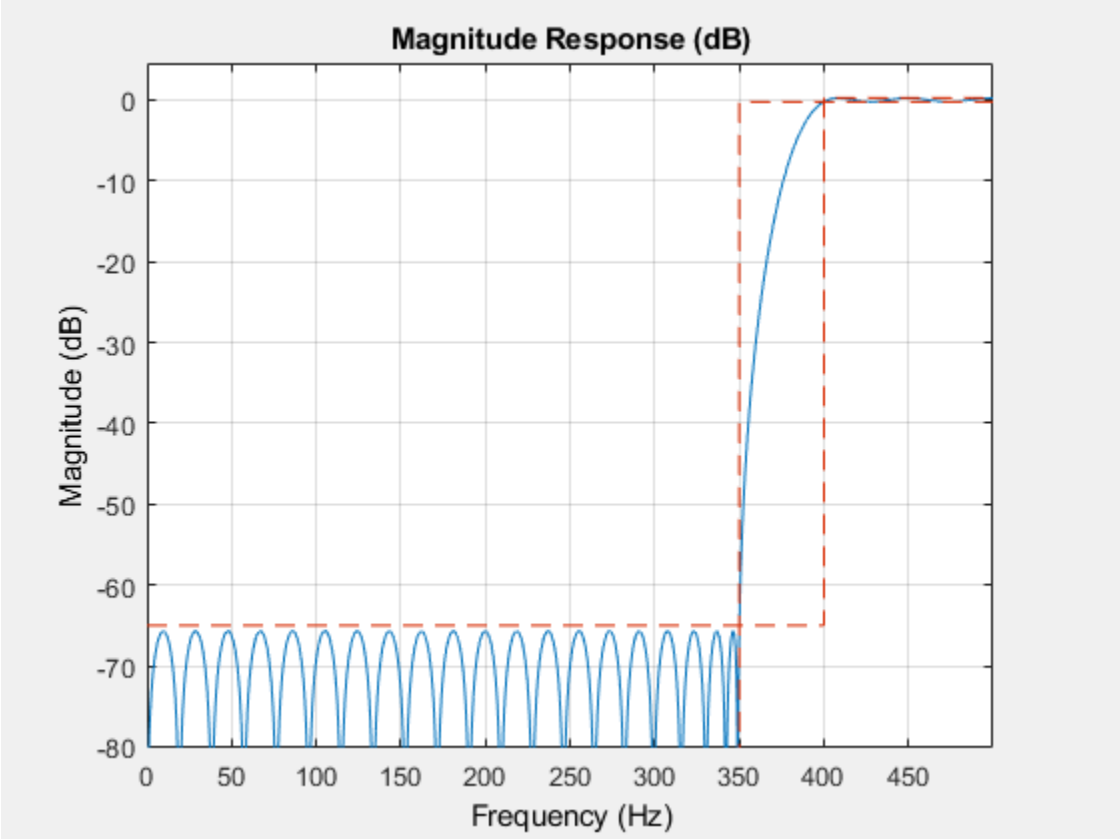
```
N = 8;
Fpass = 100;
Apass = 0.5;
Astop = 65;
Fs = 1e3;

d = designfilt('lowpassiir', ...
  'FilterOrder',N, ...
  'PassbandFrequency',Fpass, ...
  'PassbandRipple',Apass,'StopbandAttenuation',Astop, ...
  'SampleRate',Fs);

fvtool(d)
```




### Highpass FIR Filters

#### Equiripple Design

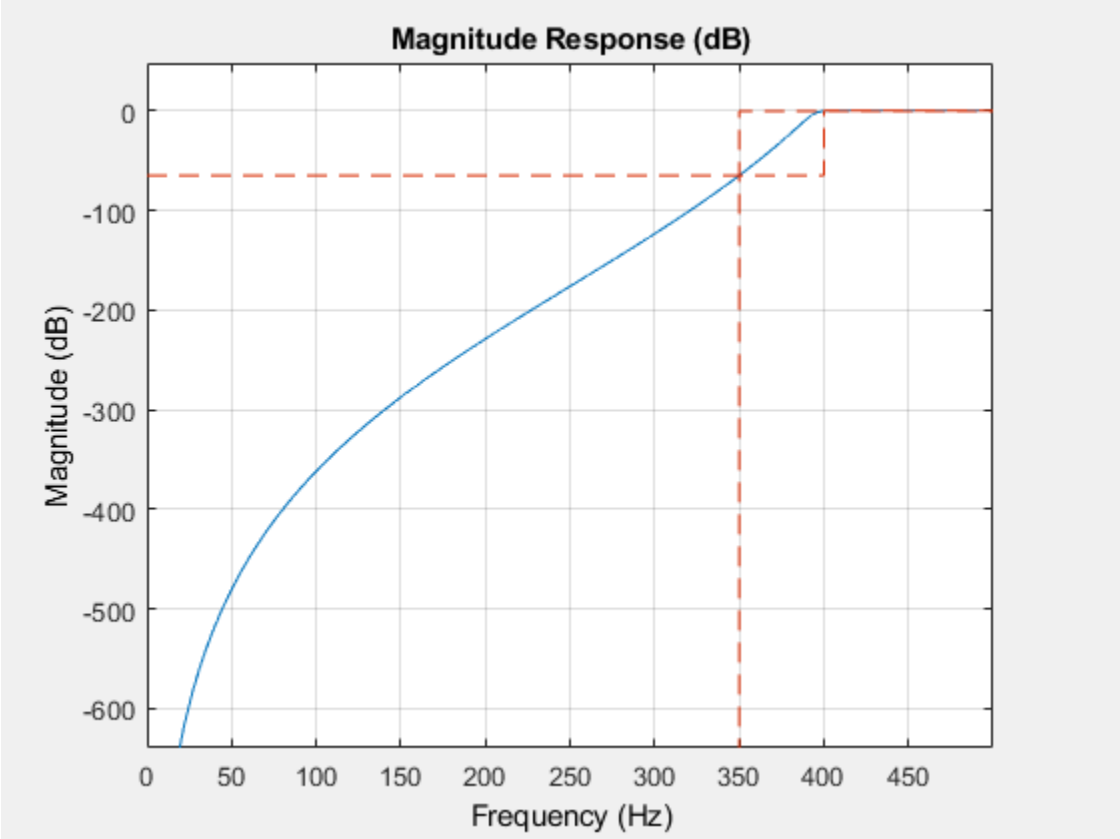
```
Fstop = 350;
Fpass = 400;
Astop = 65;
Apass = 0.5;
Fs = 1e3;

d = designfilt('highpassfir','StopbandFrequency',Fstop, ...
  'PassbandFrequency',Fpass,'StopbandAttenuation',Astop, ...
  'PassbandRipple',Apass,'SampleRate',Fs,'DesignMethod','equiripple');

fvtool(d)
```




### Highpass IIR Filters

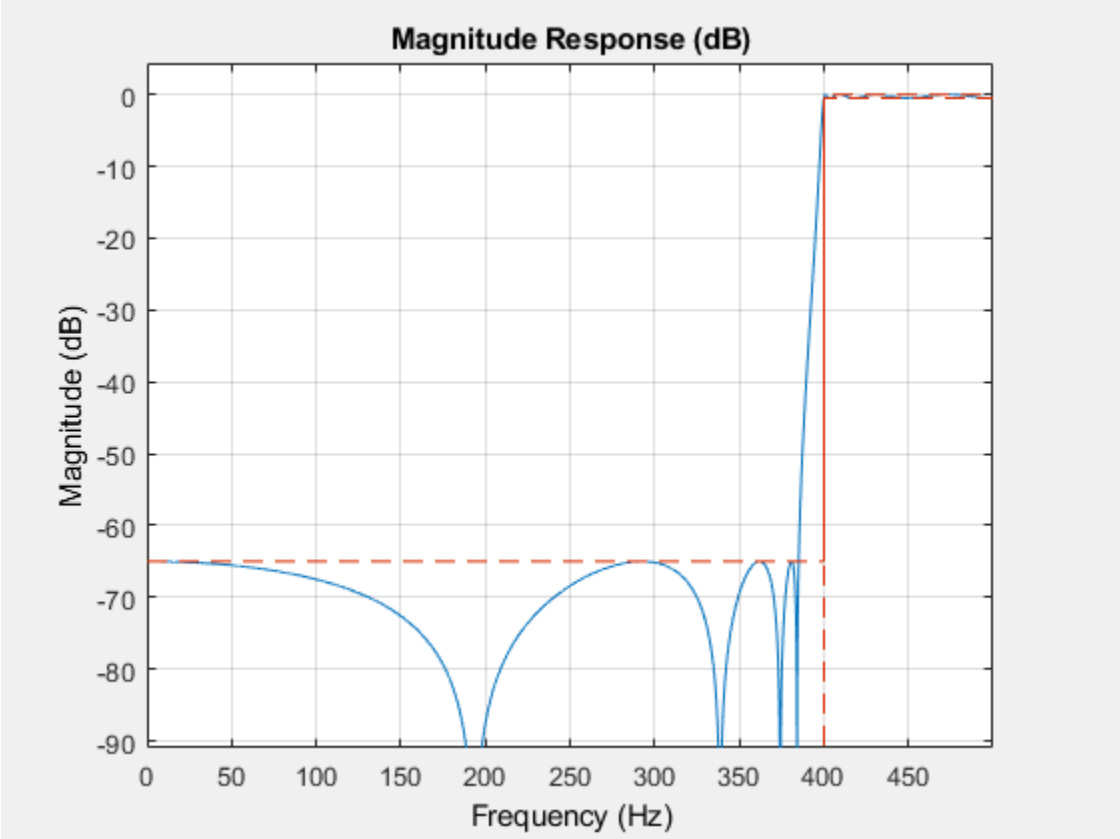
#### Maximally Flat Design

```
Fstop = 350;
Fpass = 400;
Astop = 65;
Apass = 0.5;
Fs = 1e3;

d = designfilt('highpassiir','StopbandFrequency',Fstop ,...
  'PassbandFrequency',Fpass,'StopbandAttenuation',Astop, ...
  'PassbandRipple',Apass,'SampleRate',Fs,'DesignMethod','butter');

fvtool(d)
```




**Ripple in Passband and Stopband**

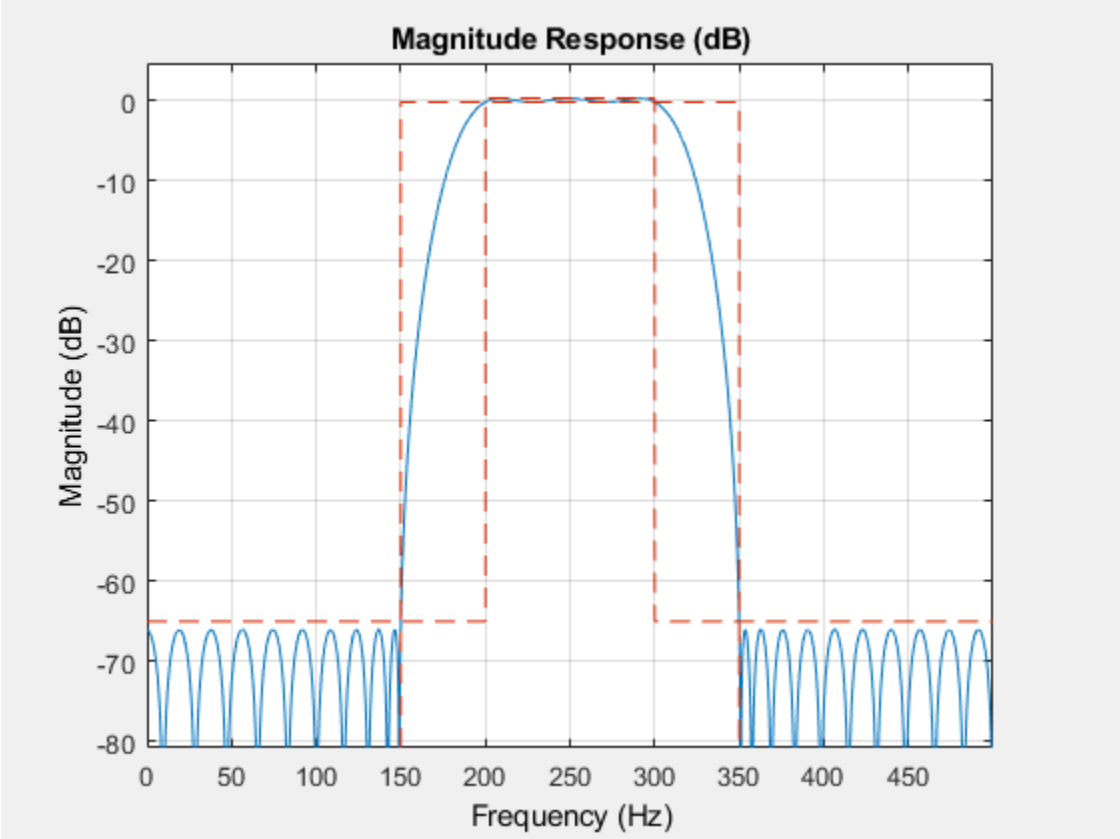
```
N = 8;
Fpass = 400;
Astop = 65;
Apass = 0.5;
Fs = 1e3;

d = designfilt('highpassiir', ...
  'FilterOrder',N, ...
  'PassbandFrequency',Fpass, ...
  'StopbandAttenuation',Astop,'PassbandRipple',Apass, ...
  'SampleRate',Fs);

fvtool(d)
```




**Bandpass FIR Filters**

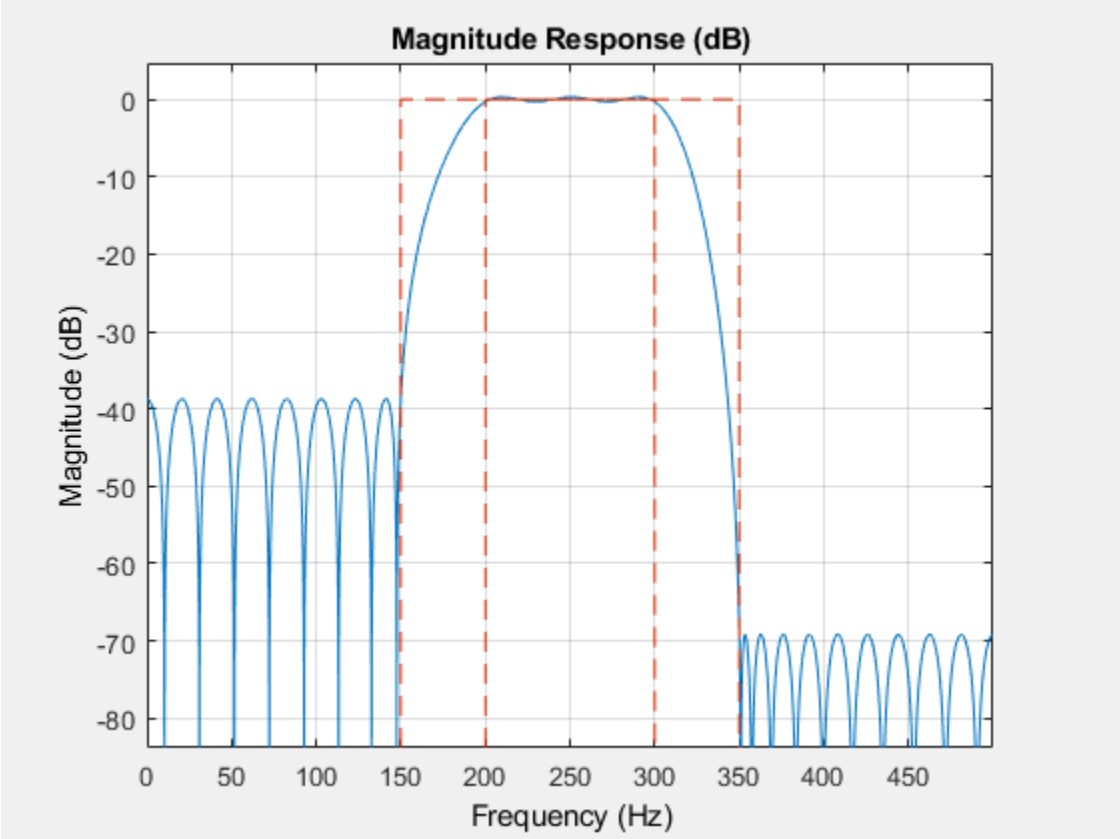
### Equiripple Design

```
Fstop1 = 150;
Fpass1 = 200;
Fpass2 = 300;
Fstop2 = 350;
Astop1 = 65;
Apass  = 0.5;
Astop2 = 65;
Fs = 1e3;

d = designfilt('bandpassfir', ...
  'StopbandFrequency1',Fstop1,'PassbandFrequency1', Fpass1, ...
  'PassbandFrequency2',Fpass2,'StopbandFrequency2', Fstop2, ...
  'StopbandAttenuation1',Astop1,'PassbandRipple', Apass, ...
  'StopbandAttenuation2',Astop2, ...
  'DesignMethod','equiripple','SampleRate',Fs);

fvtool(d)
```




**Asymmetric Band Attenuations**

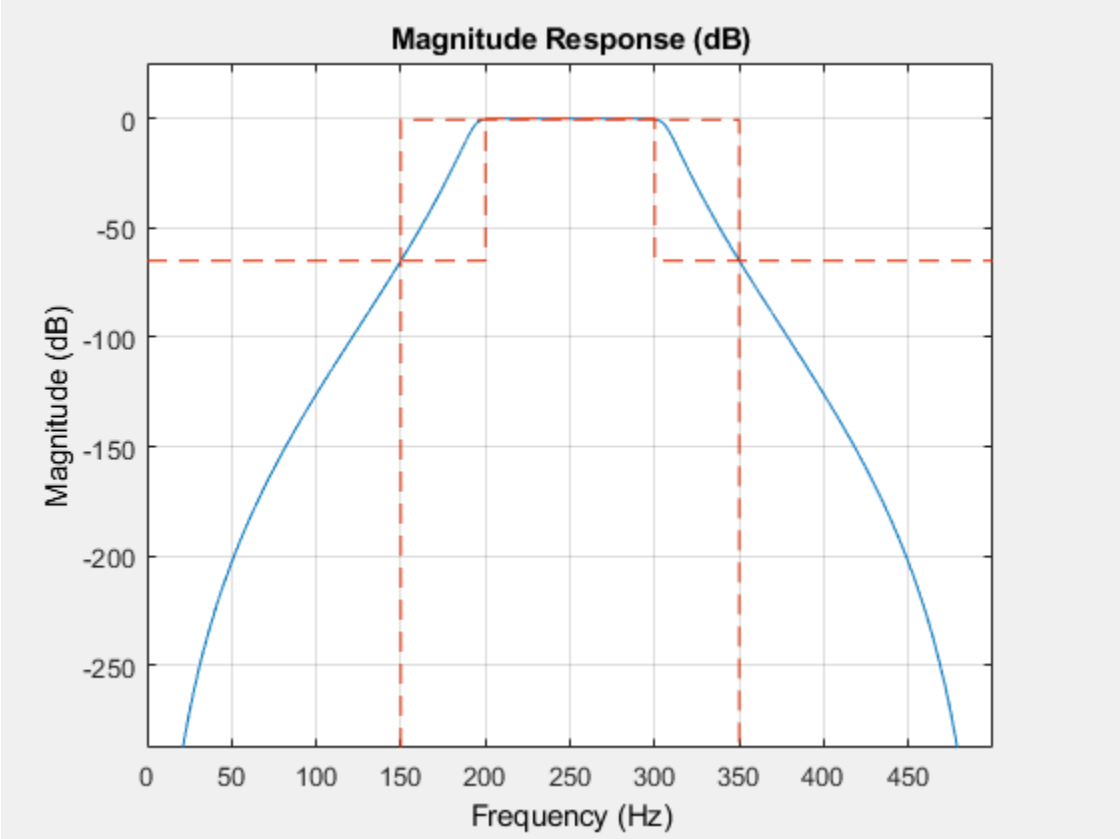
```
N = 50;
Fstop1 = 150;
Fpass1 = 200;
Fpass2 = 300;
Fstop2 = 350;
Wstop1 = 3;
Wstop2 = 100;
Fs = 1e3;

d = designfilt('bandpassfir', ...
  'FilterOrder',N, ...
  'StopbandFrequency1',Fstop1,'PassbandFrequency1', Fpass1, ...
  'PassbandFrequency2',Fpass2,'StopbandFrequency2', Fstop2, ...
  'StopbandWeight1',Wstop1,'StopbandWeight2',Wstop2, ...
   'DesignMethod','equiripple','SampleRate',Fs);

fvtool(d)
```




### Bandpass IIR Filters

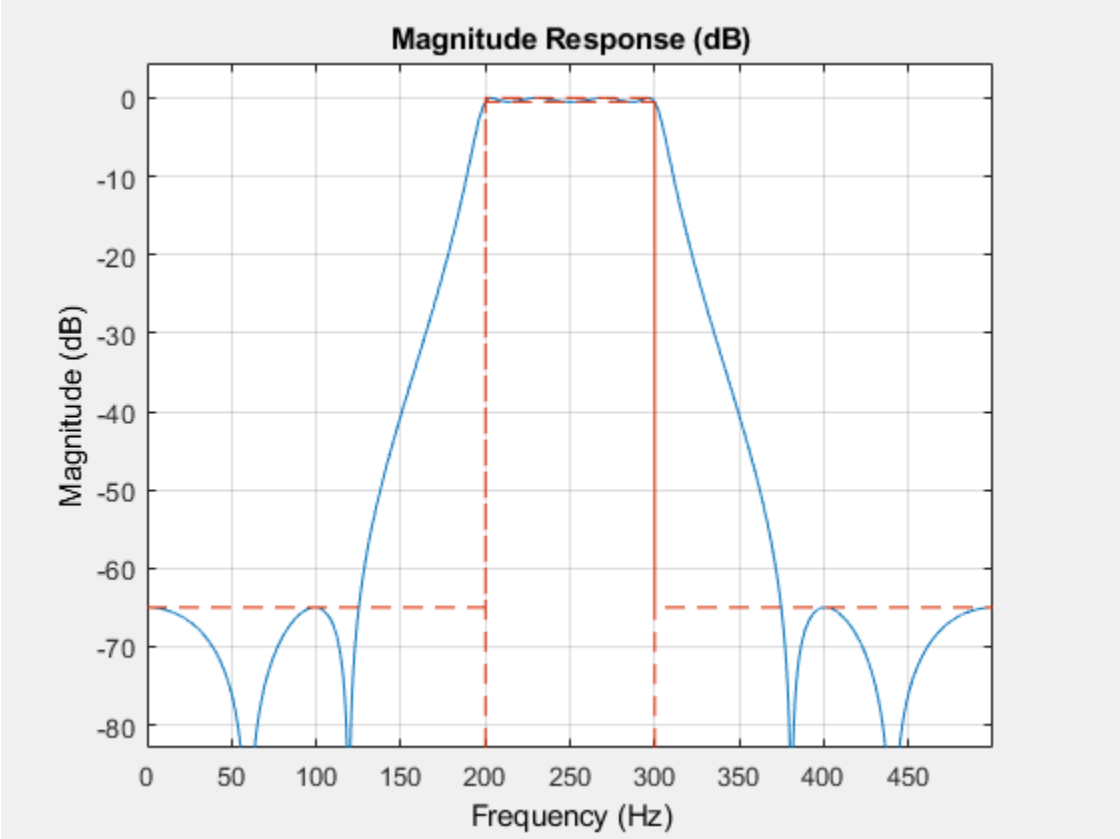
#### Maximally Flat Design

```
Fstop1 = 150;
Fpass1 = 200;
Fpass2 = 300;
Fstop2 = 350;
Astop1 = 65;
Apass  = 0.5;
Astop2 = 65;
Fs = 1e3;

d = designfilt('bandpassiir', ...
  'StopbandFrequency1',Fstop1,'PassbandFrequency1', Fpass1, ...
  'PassbandFrequency2',Fpass2,'StopbandFrequency2', Fstop2, ...
  'StopbandAttenuation1',Astop1,'PassbandRipple', Apass, ...
  'StopbandAttenuation2',Astop2, ...
  'DesignMethod','butter','SampleRate', Fs);

fvtool(d)
```




**Ripple in Passband and Stopband**

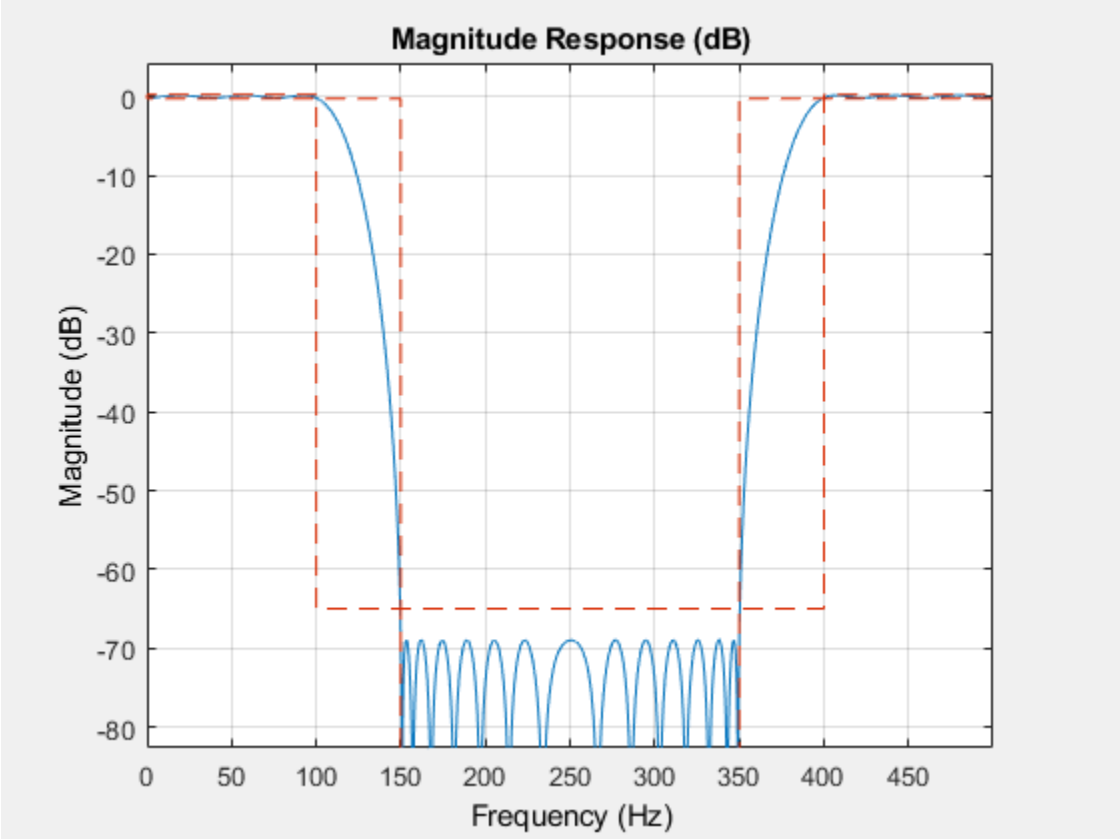
```
N = 8;
Fpass1 = 200;
Fpass2 = 300;
Astop1 = 65;
Apass  = 0.5;
Astop2 = 65;
Fs = 1e3;

d = designfilt('bandpassiir', ...
  'FilterOrder',N, ...
  'PassbandFrequency1', Fpass1,'PassbandFrequency2', Fpass2, ...
  'StopbandAttenuation1', Astop1, 'PassbandRipple', Apass, ...
  'StopbandAttenuation2', Astop2, ...
  'SampleRate', Fs);

fvtool(d)
```




**Bandstop FIR Filters**

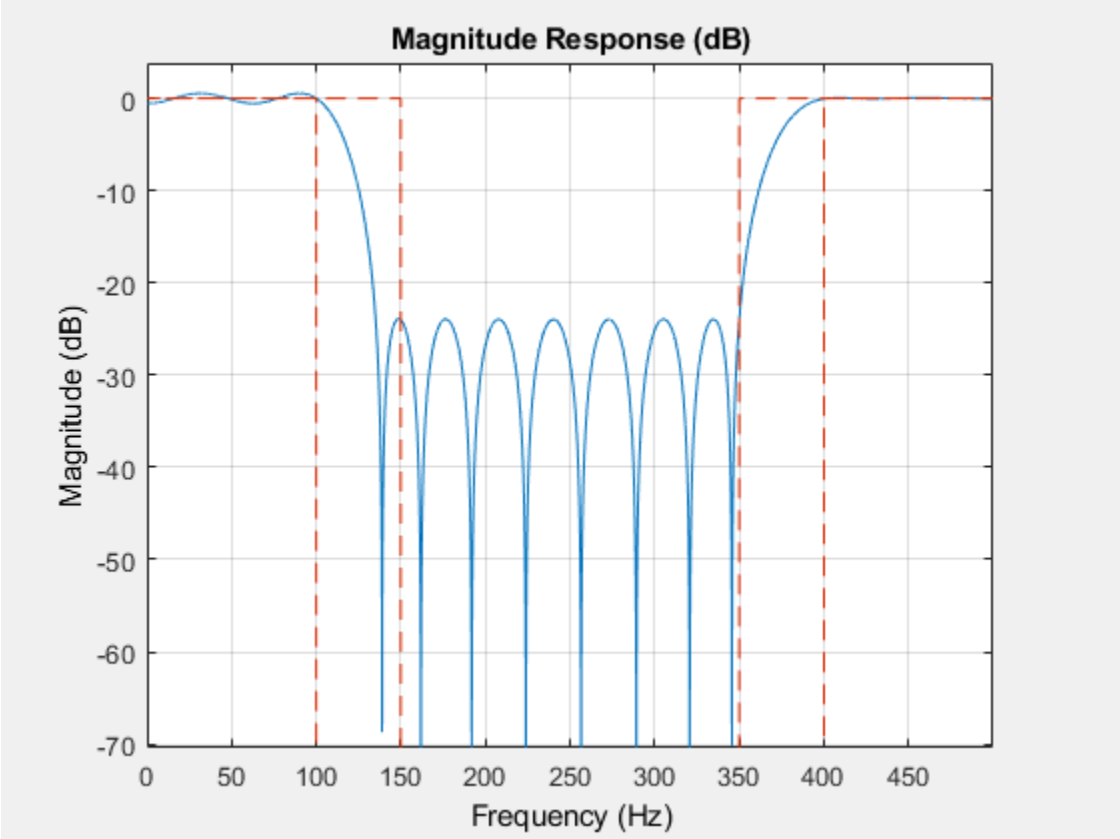
### Equiripple Design

```
Fpass1 = 100;
Fstop1 = 150;
Fstop2 = 350;
Fpass2 = 400;
Apass1 = 0.5;
Astop  = 65;
Apass2 = 0.5;
Fs = 1e3;

d = designfilt('bandstopfir', ...
  'PassbandFrequency1',Fpass1,'StopbandFrequency1',Fstop1, ...
  'StopbandFrequency2',Fstop2,'PassbandFrequency2',Fpass2, ...
  'PassbandRipple1',Apass1,'StopbandAttenuation',Astop, ...
  'PassbandRipple2', Apass2, ...
  'DesignMethod','equiripple','SampleRate', Fs);

fvtool(d)
```




**Asymmetric Passband Ripples**

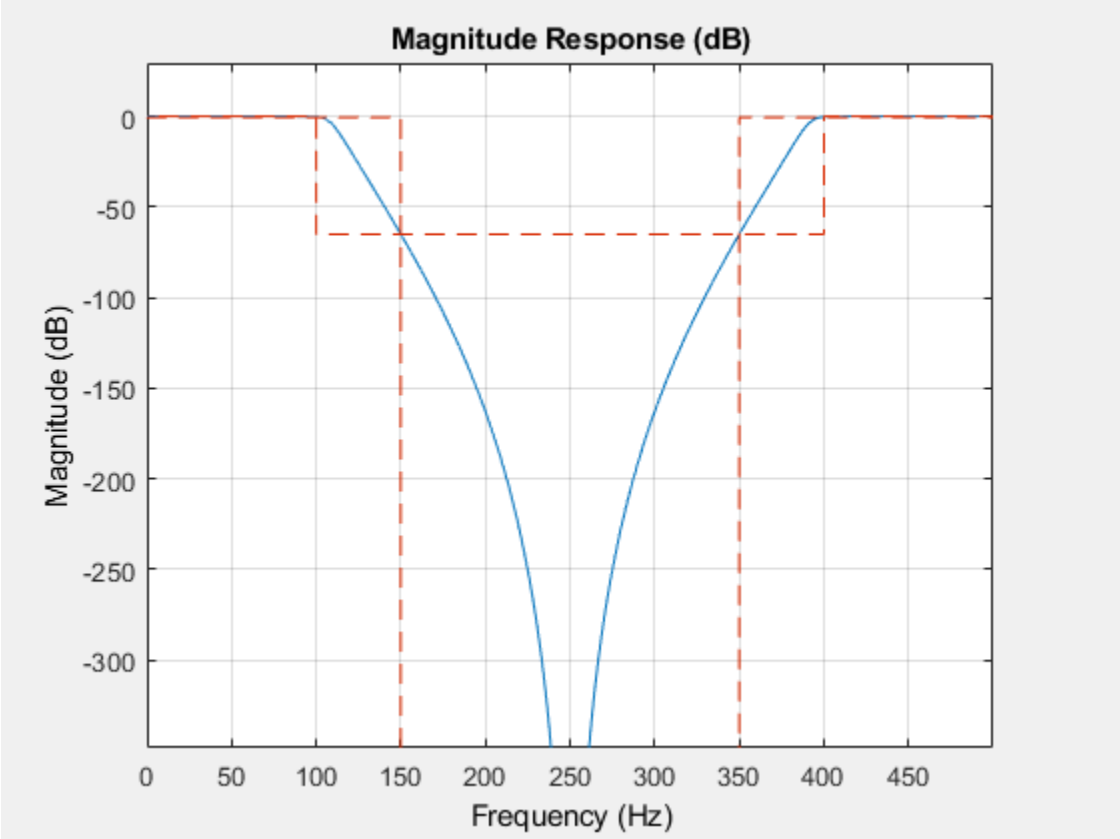
```
N = 30;
Fpass1 = 100;
Fstop1 = 150;
Fstop2 = 350;
Fpass2 = 400;
Wpass1 = 1;
Wpass2  = 10;
Fs = 1e3;

d = designfilt('bandstopfir', ...
  'FilterOrder',N, ...
  'PassbandFrequency1',Fpass1,'StopbandFrequency1',Fstop1, ...
  'StopbandFrequency2',Fstop2,'PassbandFrequency2',Fpass2, ...
  'PassbandWeight1',Wpass1,'PassbandWeight2',Wpass2, ...
  'DesignMethod','equiripple','SampleRate', Fs);

fvtool(d)
```




### Bandstop IIR Filters

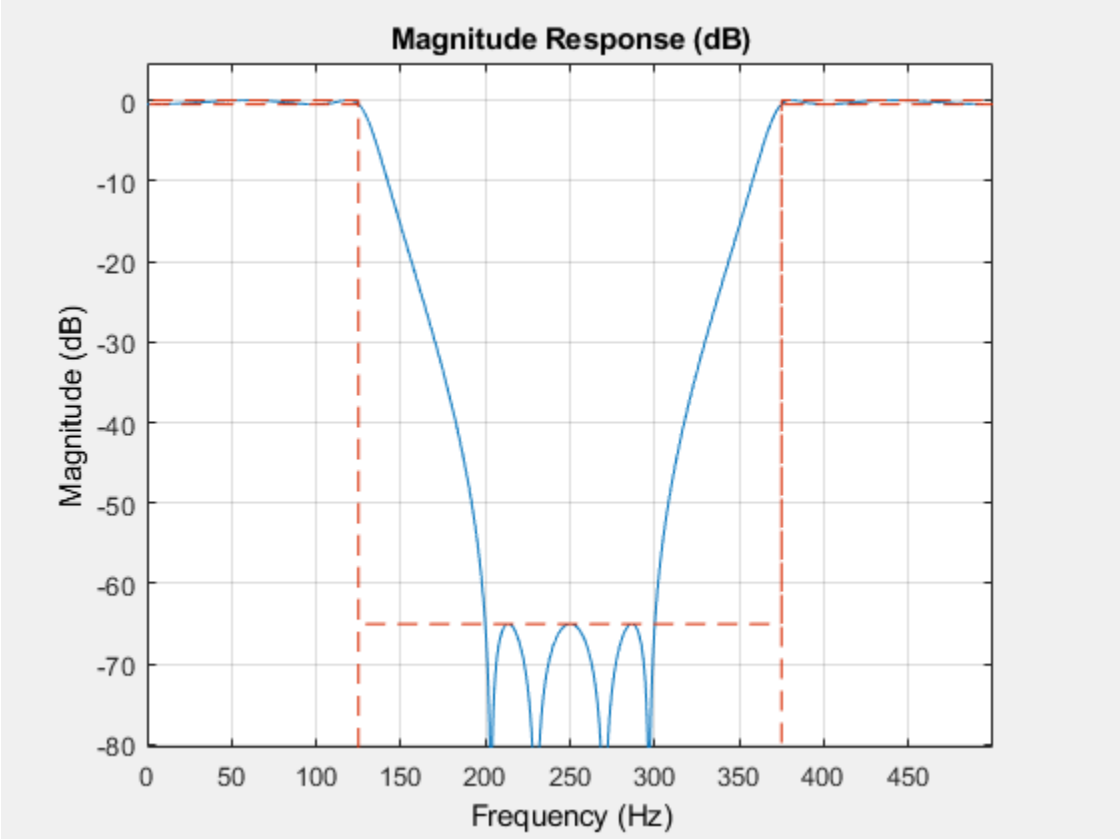
#### Maximally Flat Design

```
Fpass1 = 100;
Fstop1 = 150;
Fstop2 = 350;
Fpass2 = 400;
Apass1 = 0.5;
Astop  = 65;
Apass2 = 0.5;
Fs = 1e3;

d = designfilt('bandstopiir', ...
  'PassbandFrequency1',Fpass1,'StopbandFrequency1',Fstop1, ...
  'StopbandFrequency2',Fstop2,'PassbandFrequency2',Fpass2, ...
  'PassbandRipple1',Apass1,'StopbandAttenuation',Astop, ...
  'PassbandRipple2', Apass2, ...
  'DesignMethod','butter','SampleRate', Fs);

fvtool(d)
```




**Ripple in Passband and Stopband**

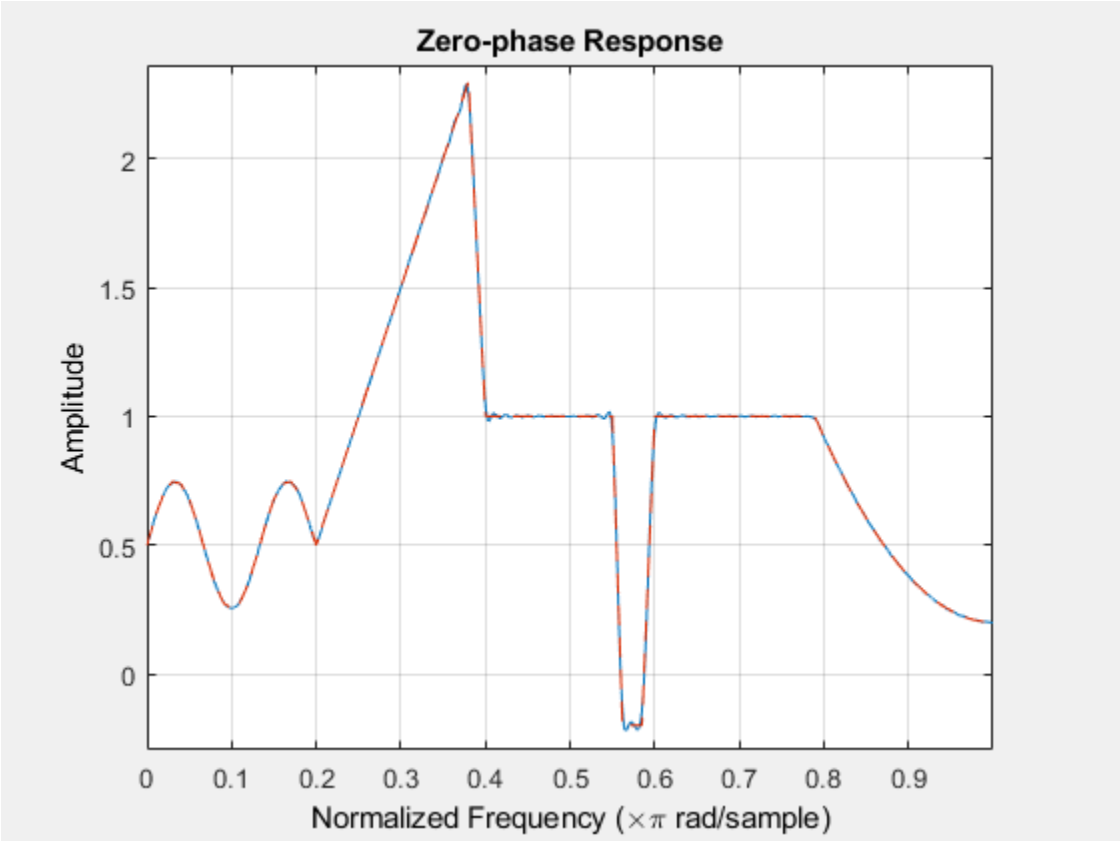
```
N = 8;
Fpass1 = 125;
Fpass2 = 375;
Apass = 0.5;
Astop  = 65;
Fs = 1e3;

d = designfilt('bandstopiir', ...
  'FilterOrder',N, ...
  'PassbandFrequency1',Fpass1,'PassbandFrequency2',Fpass2, ...
  'PassbandRipple',Apass,'StopbandAttenuation', Astop, ...
  'SampleRate',Fs);

fvtool(d)
```




### Arbitrary Magnitude FIR Filters

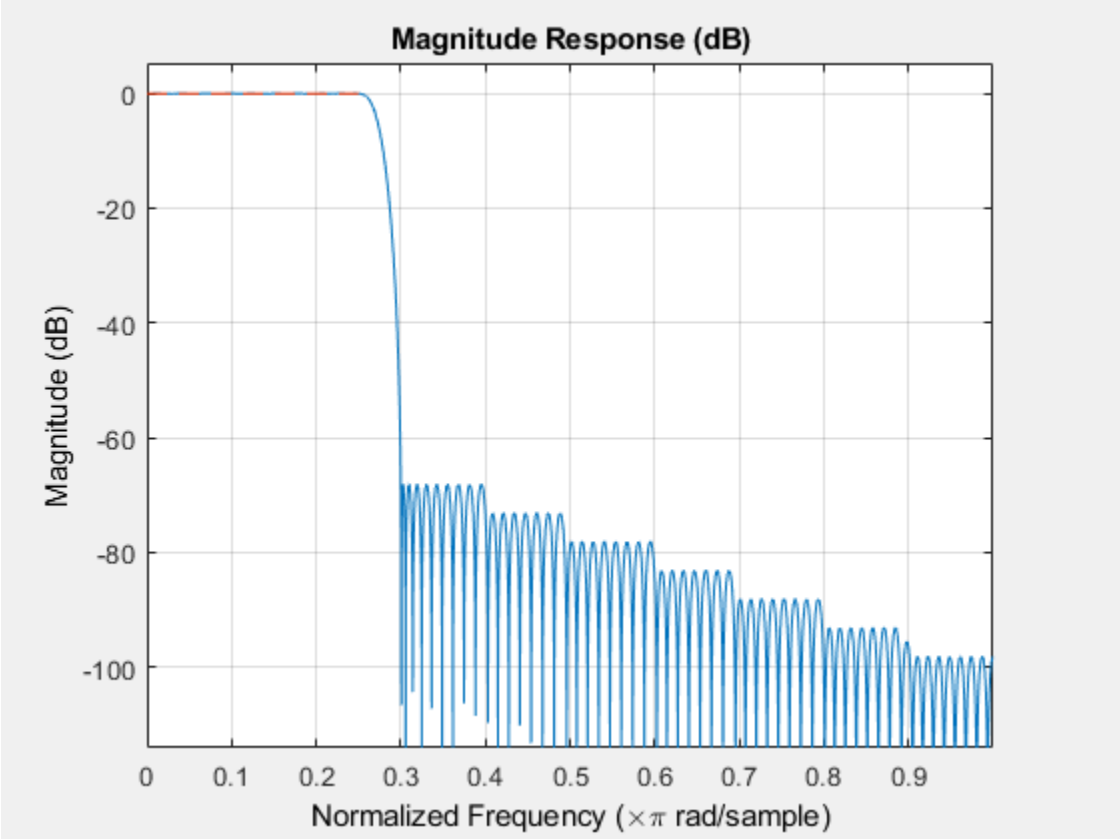
#### Single-Band Arbitrary Magnitude Design

```
N = 300;

% Frequencies are in normalized units
F1 = 0:0.01:0.18;
F2 = [.2 .38 .4 .55 .562 .585 .6 .78];
F3 = 0.79:0.01:1;
FreqVect = [F1 F2 F3]; % vector of frequencies

% Define desired response using linear units
A1 = .5+sin(2*pi*7.5*F1)/4;    % Sinusoidal section
A2 = [.5 2.3 1 1 -.2 -.2 1 1]; % Piecewise linear section
A3 = .2+18*(1-F3).^2;          % Quadratic section

AmpVect = [A1 A2 A3];

d = designfilt('arbmagfir',...
  'FilterOrder',N,'Amplitudes',AmpVect,'Frequencies',FreqVect,...
  'DesignMethod','freqsamp');

fvtool(d,'MagnitudeDisplay','Zero-phase')
```




**Multiband Lowpass Design with Stepped Attenuation Levels on Stopband**

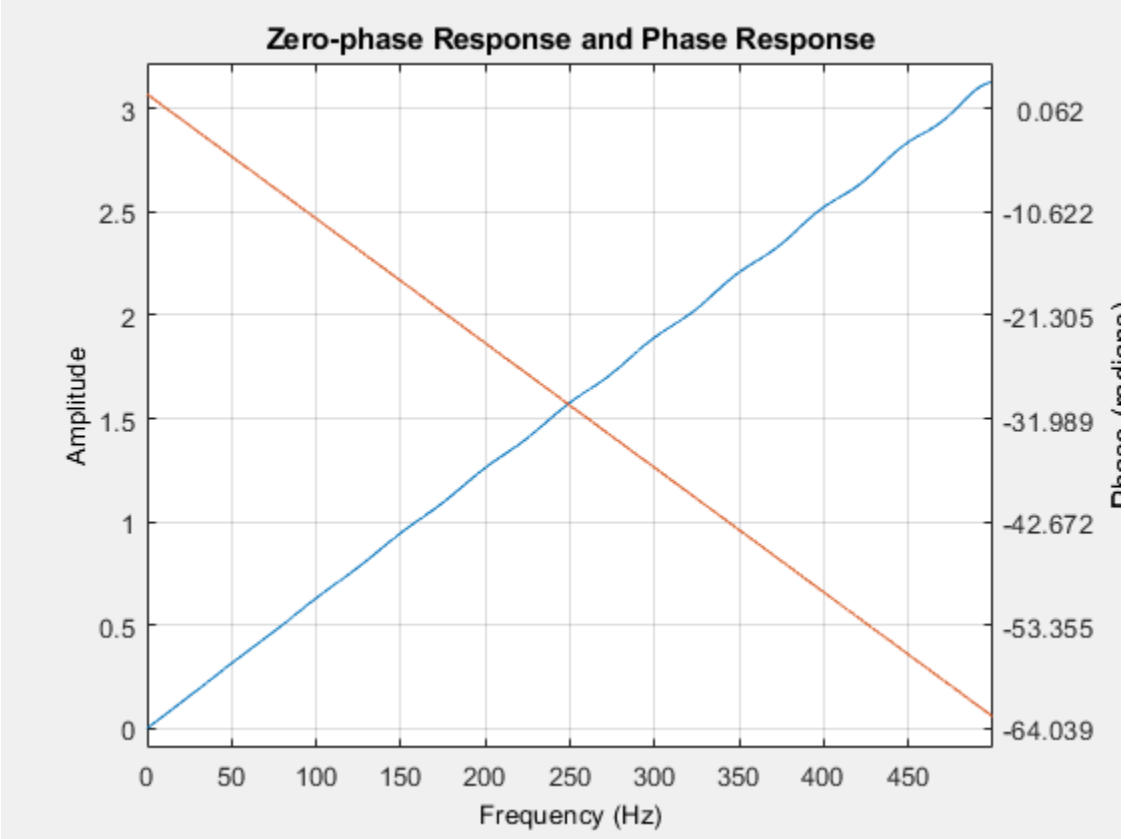
```
N = 150;
B = 2; % Number of bands
% Frequencies are in normalized units
F1 = [0 .25]; % Passband
F2 = [.3 .4 .401 .5 .501 .6 .601 .7 .701 .8 .801 .9 .901 1]; % Stopband
A1 = ones(size(F1));  % Desired amplitudes for band 1 in linear units
A2 = zeros(size(F2)); % Desired amplitudes for band 2 in linear units
% Vector of weights
W = 10.^([0 0 5 5 10 10 15 15 20 20 25 25 30 30 35 35]/20);
W1 = W(1:2);   % Weights for band 1
W2 = W(3:end); % Weights for band 2

d = designfilt('arbmagfir', ...
  'FilterOrder',N,'NumBands',B, ...
  'BandFrequencies1',F1,'BandAmplitudes1',A1, ...
  'BandFrequencies2',F2,'BandAmplitudes2',A2, ...
  'BandWeights1',W1,'BandWeights2', W2);

fvtool(d)
```




**Differentiator FIR Filters**

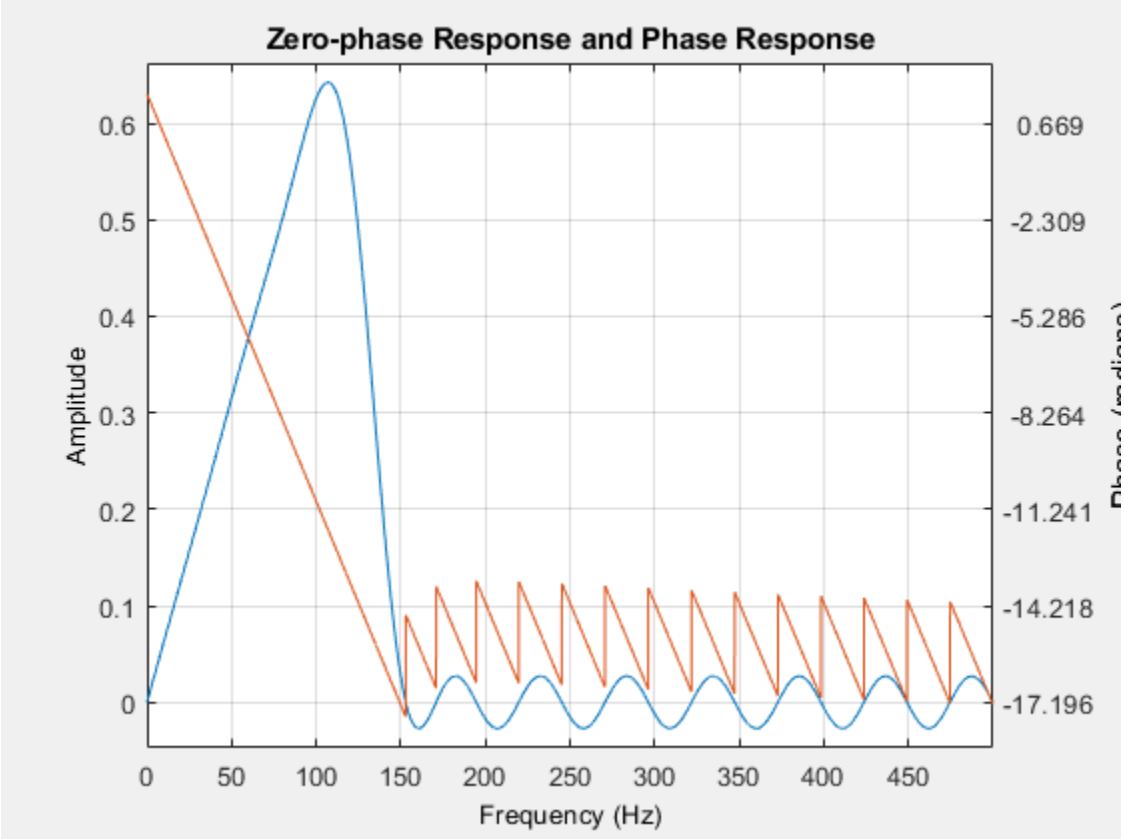
### Full Band Design

```
N = 41;
Fs = 1e3;

d = designfilt('differentiatorfir', ...
'FilterOrder',N, 'DesignMethod','equiripple','SampleRate',Fs);

fvtool(d,'MagnitudeDisplay','zero-phase','OverlayedAnalysis','phase')
```




**Partial Band Design**

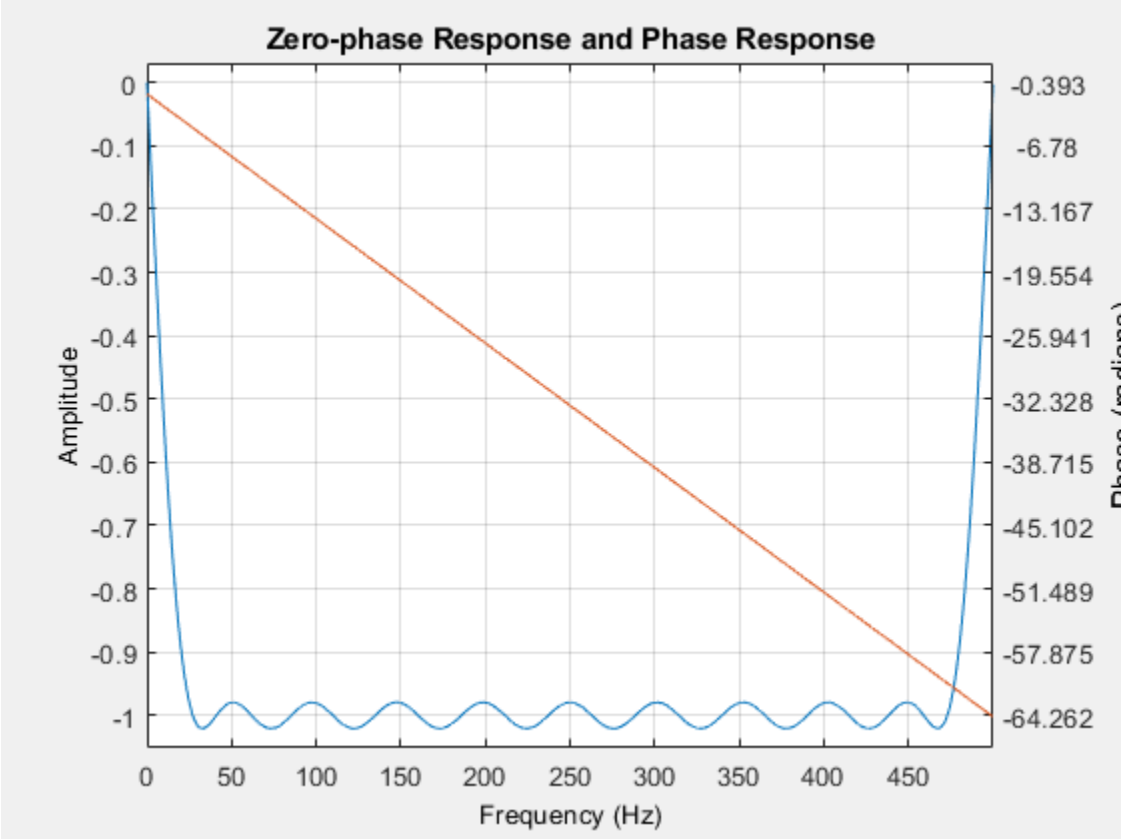
```
N = 40;
Fpass = 100;
Fstop = 150;
Fs = 1e3;

d = designfilt('differentiatorfir', ...
'FilterOrder',N, ...
'PassbandFrequency',Fpass,'StopbandFrequency',Fstop, ...
'DesignMethod','equiripple','SampleRate',Fs);

fvtool(d,'MagnitudeDisplay','zero-phase','OverlayedAnalysis','phase')
```




### Hilbert FIR Filters

#### Equiripple Design

```
N = 40;
Tw = 50;
Fs = 1e3;

d = designfilt('hilbertfir', ...
'FilterOrder',N,'TransitionWidth',Tw, ...
'DesignMethod','equiripple','SampleRate',Fs);

fvtool(d,'MagnitudeDisplay','Zero-phase','OverlayedAnalysis','phase')
```




## See Also

**FVTool** | `designfilt`

# Practical Introduction to Digital Filter Design

This example shows how to design FIR and IIR filters based on frequency response specifications using the `designfilt` function in the Signal Processing Toolbox® product. The example concentrates on lowpass filters but most of the results apply to other response types as well.

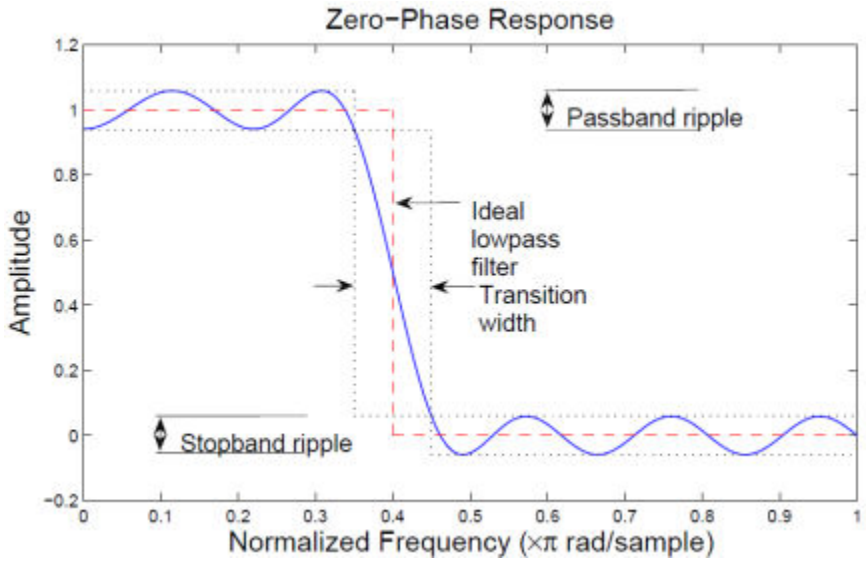
This example focuses on the design of digital filters rather than on their applications. If you want to learn more about digital filter applications, see “Practical Introduction to Digital Filtering” on page 24-174.

### FIR Filter Design

#### Lowpass Filter Specifications

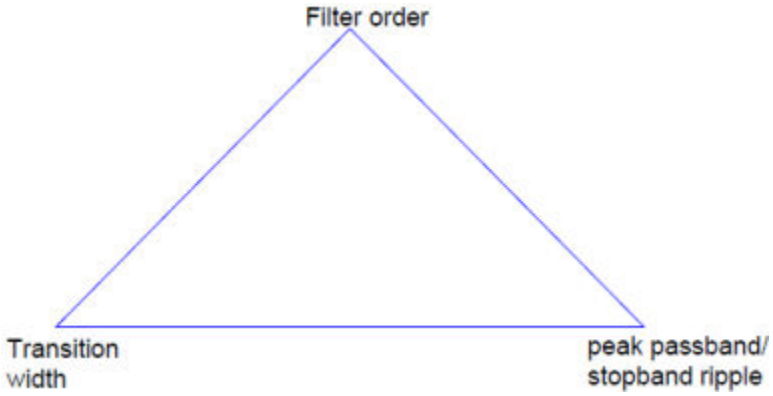
The ideal lowpass filter is one that leaves unchanged all frequency components of a signal below a designated cutoff frequency, $\omega_c$, and rejects all components above $\omega_c$. Because the impulse response required to implement the ideal lowpass filter is infinitely long, it is impossible to design an ideal FIR lowpass filter. Finite length approximations to the ideal impulse response lead to the presence of ripples in both the passband ($\omega < \omega_c$) and the stopband ($\omega > \omega_c$) of the filter, as well as to a nonzero transition width between passband and stopband.

Both the passband/stopband ripples and the transition width are undesirable but unavoidable deviations from the response of an ideal lowpass filter when approximated with a finite impulse response. These deviations are depicted in the following figure:




- **Practical FIR designs typically consist of filters that have a transition width and maximum passband and stopband ripples that do not exceed allowable values. In addition to those design specifications, one must select the filter order, or, equivalently, the length of the truncated impulse response.**

A useful metaphor for the design specifications in filter design is to think of each specification as one of the angles in the triangle shown in the figure below.




The triangle is used to understand the degrees of freedom available when choosing design specifications. Because the sum of the angles is fixed, one can at most select the values of two of the specifications. The third specification will be determined by the particular design algorithm. Moreover, as with the angles in a triangle, if we make one of the specifications larger/smaller, it will impact one or both of the other specifications.

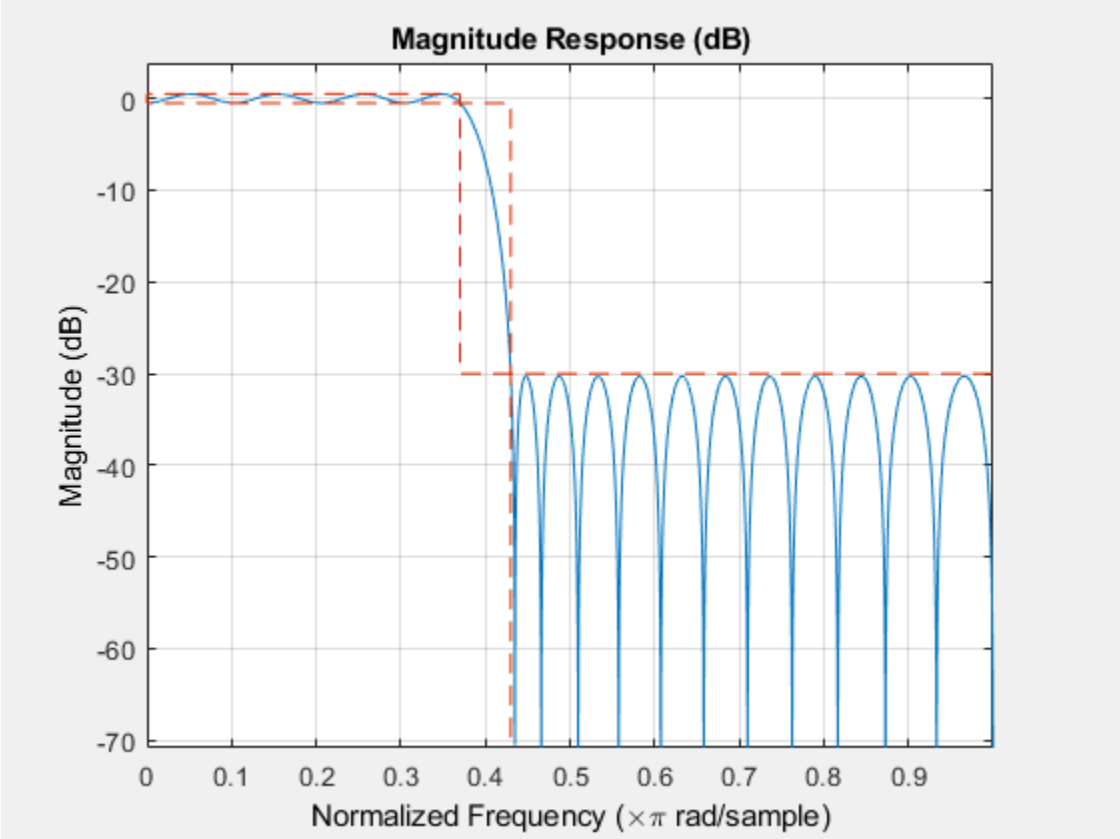
FIR filters are very attractive because they are inherently stable and can be designed to have linear phase. Nonetheless, these filters can have long transient responses and might prove computationally expensive in certain applications.

**Minimum-Order FIR Designs**

Minimum-order designs are obtained by specifying passband and stopband frequencies as well as a passband ripple and a stopband attenuation. The design algorithm then chooses the minimum filter length that complies with the specifications.

Design a minimum-order lowpass FIR filter with a passband frequency of 0.37*pi rad/sample, a stopband frequency of 0.43*pi rad/sample (hence the transition width equals 0.06*pi rad/sample), a passband ripple of 1 dB and a stopband attenuation of 30 dB.

```
Fpass = 0.37;
Fstop = 0.43;
Ap = 1;
Ast = 30;

d = designfilt('lowpassfir','PassbandFrequency',Fpass,...
  'StopbandFrequency',Fstop,'PassbandRipple',Ap,'StopbandAttenuation',Ast);

hfvt = fvtool(d);
```




The resulting filter order can be queried using the `filtord` function.

```
N = filtord(d)
```

```
N = 39
```

You can use the `info` function to get information about the parameters used to design the filter

```
info(d)
```

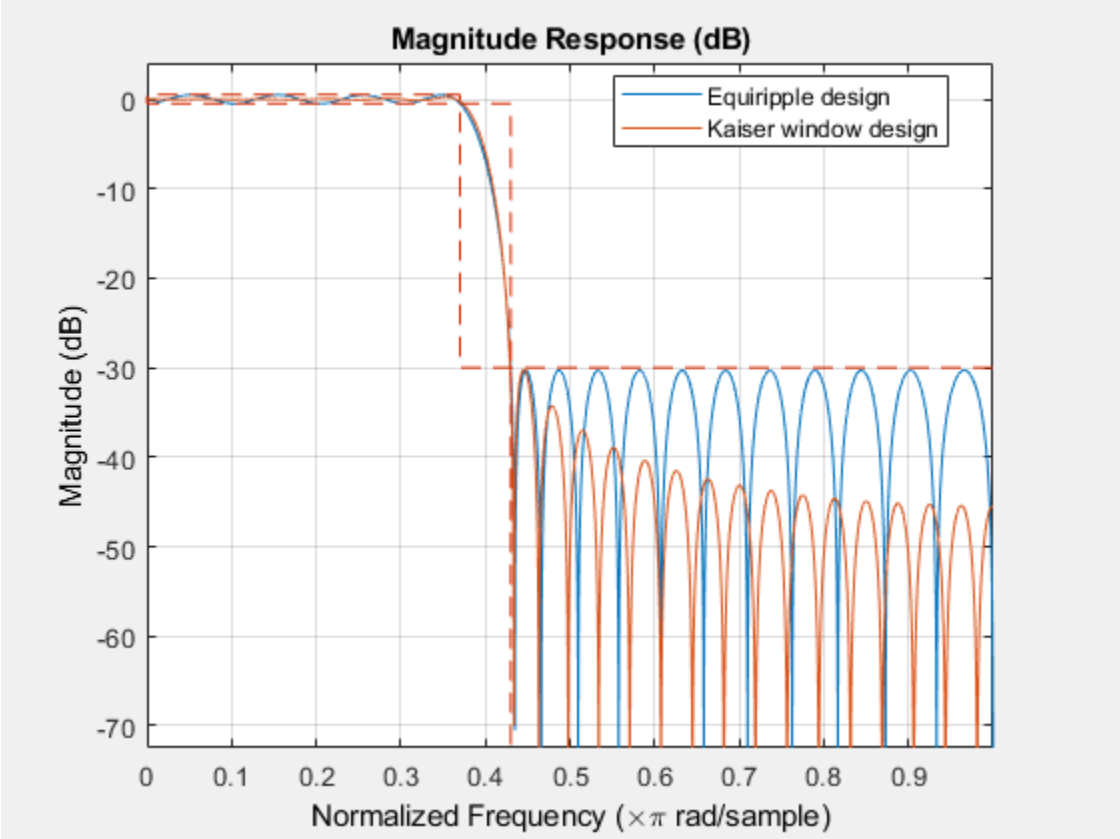
```
ans = 17x44 char array
    'FIR Digital Filter (real)                  '
    '--------------------------                 '
    'Filter Length  : 40                        '
    'Stable         : Yes                       '
    'Linear Phase   : Yes (Type 2)              '
    '                                           '
    'Design Method Information                  '
    'Design Algorithm : Equiripple              '
    '                                           '
    'Design Specifications                      '
    'Sample Rate     : N/A (normalized frequency)'
    'Response        : Lowpass                  '
    'Specification   : Fp,Fst,Ap,Ast            '
    'Passband Ripple : 1 dB                     '
    'Stopband Atten. : 30 dB                    '
    'Stopband Edge   : 0.43                     '
    'Passband Edge   : 0.37                     '
```

- **The `designfilt` function chooses an equiripple design algorithm by default. Linear-phase equiripple filters are desirable because for a given order they have the smallest possible maximum deviation from the ideal filter.**

Note, however, that minimum-order designs can also be obtained using a Kaiser window. Even though the Kaiser window method yields a larger filter order for the same specifications, the algorithm is less computationally expensive and less likely to have convergence issues when the design specifications are very stringent. This may occur if the application requires a very narrow transition width or a very large stopband attenuation.

Design a filter with the same specifications as above using the Kaiser window method and compare its response to the equiripple filter.

```
dk = designfilt('lowpassfir','PassbandFrequency',Fpass,...
  'StopbandFrequency',Fstop,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast, 'DesignMethod', 'kaiserwin');

addfilter(hfvt,dk);
legend(hfvt,'Equiripple design', 'Kaiser window design')
```



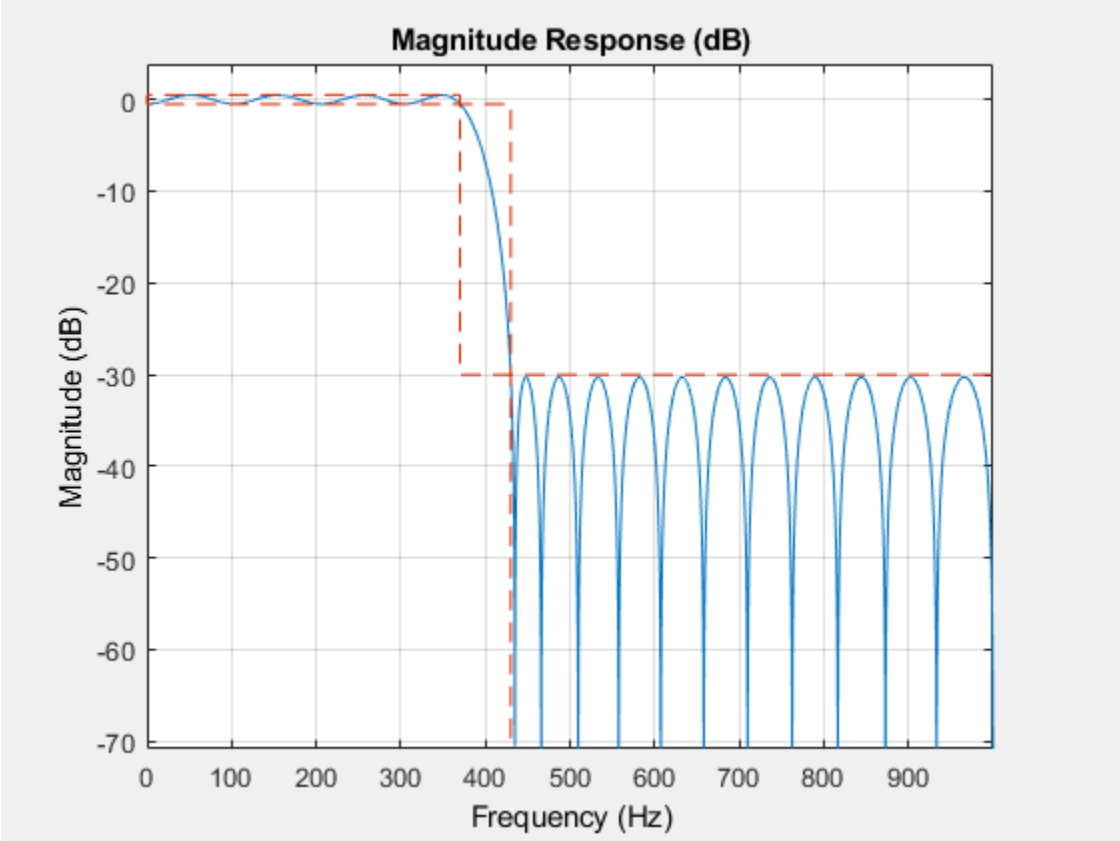

```
N = filtord(dk)
```

```
N = 52
```

**Specifying Frequency Parameters in Hertz**

If you know the sample rate at which the filter will operate, you can specify the sample rate and the frequencies in hertz. Redesign the minimum-order equiripple filter for a sample rate of 2 kHz.

```
Fpass = 370;
Fstop = 430;
Ap = 1;
Ast = 30;
Fs = 2000;

d = designfilt('lowpassfir','PassbandFrequency',Fpass,...
  'StopbandFrequency',Fstop,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs);

hfvt = fvtool(d);
```



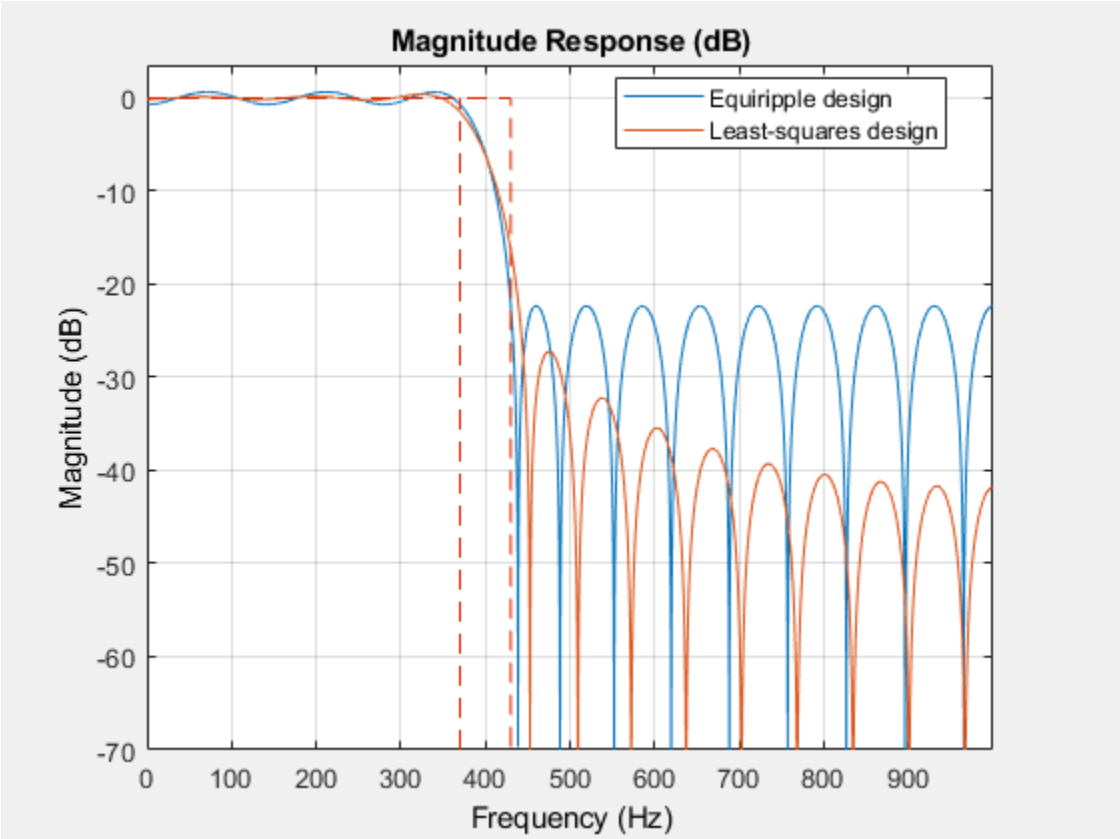

**Fixed Order, Fixed Transition Width**

Fixed-order designs are useful for applications that are sensitive to computational load or impose a limit on the number of filter coefficients. An option is to fix the transition width at the expense of control over the passband ripple/stopband attenuation.

Consider a 30-th order lowpass FIR filter with a passband frequency of 370 Hz, a stopband frequency of 430 Hz, and sample rate of 2 kHz. There are two design methods available for this particular set of specifications: equiripple and least squares. Let us design one filter for each method and compare the results.

```
N = 30;
Fpass = 370;
```

```
Fstop = 430;
Fs = 2000;

% Design method defaults to 'equiripple' when omitted
deq = designfilt('lowpassfir','FilterOrder',N,'PassbandFrequency',Fpass,...
  'StopbandFrequency',Fstop,'SampleRate',Fs);

dls = designfilt('lowpassfir','FilterOrder',N,'PassbandFrequency',Fpass,...
  'StopbandFrequency',Fstop,'SampleRate',Fs,'DesignMethod','ls');

hfvt = fvtool(deq,dls);
legend(hfvt,'Equiripple design', 'Least-squares design')
```

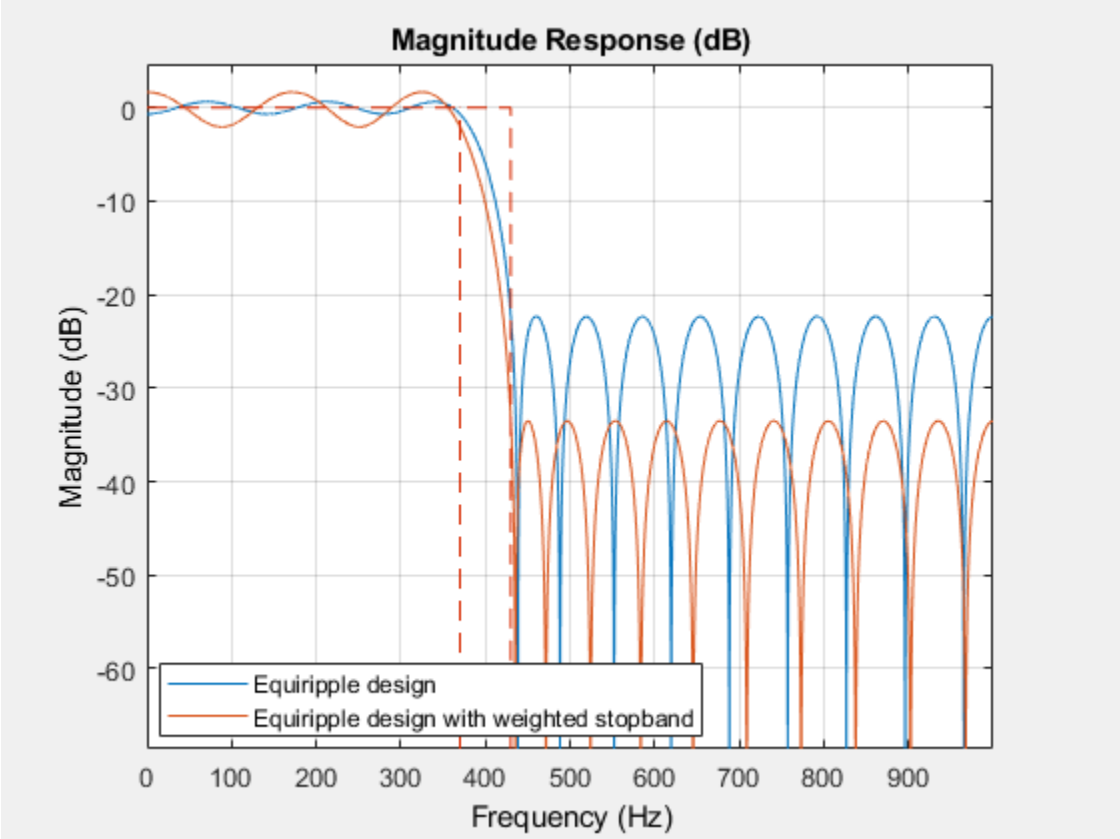



Equiripple filters are ideally suited for applications in which a specific tolerance must be met, such as designing a filter with a given minimum stopband attenuation or a given maximum passband ripple. On the other hand, these designs may not be desirable if we want to minimize the energy of the error (between ideal and actual filter) in the passband/stopband.

- **If you want to reduce the energy of a signal as much as possible in a certain frequency band, use a least-squares design.**

In the examples above, the designed filters had the same ripple in the passband and in the stopband. We can use weights to reduce the ripple in one of the bands while keeping the filter order fixed. For example, if you wish the stopband ripple to be a tenth of that in the passband, you must give the stopband ten times the passband weight. Redesign the equiripple filter using that fact.

```
deqw = designfilt('lowpassfir','FilterOrder',N,'PassbandFrequency',Fpass,...
  'StopbandFrequency',Fstop,'SampleRate',Fs,...
  'PassbandWeight',1,'StopbandWeight',10);

hfvt = fvtool(deq,deqw);
legend(hfvt,'Equiripple design', 'Equiripple design with weighted stopband')
```



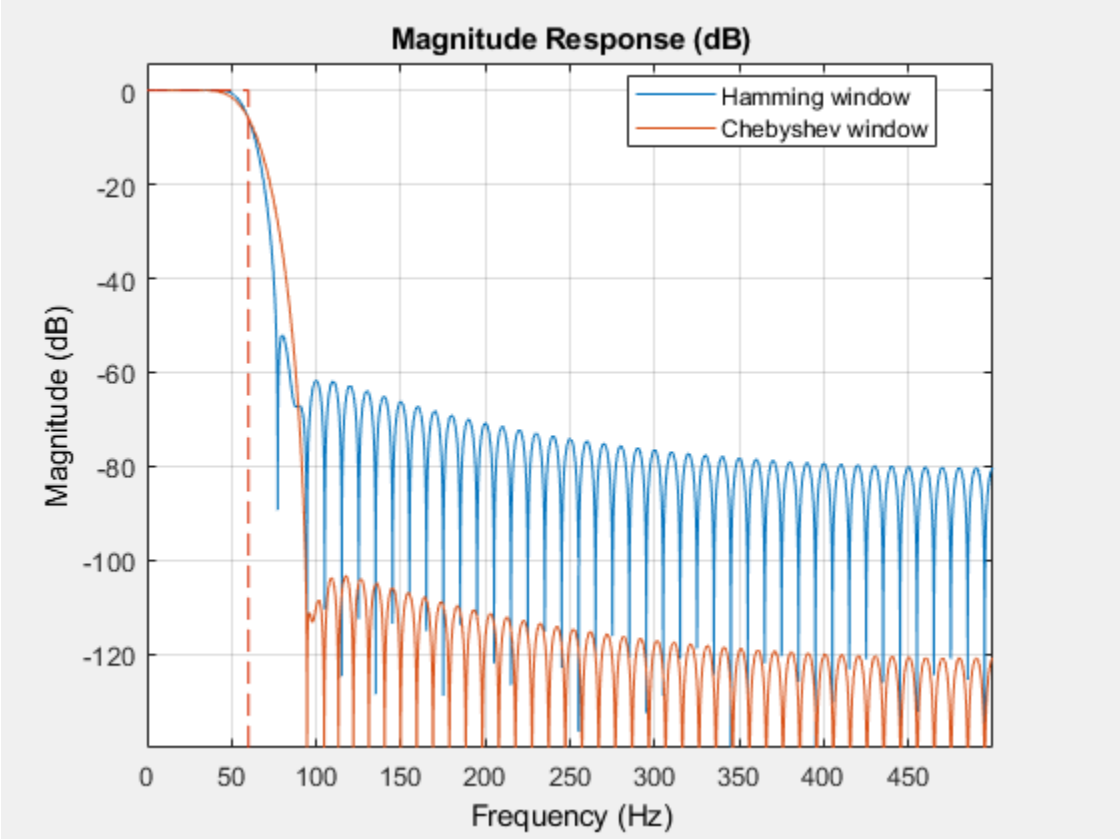

### Fixed Order, Fixed Cutoff Frequency

You can design filters with fixed filter order and cutoff frequency using a window design method.

- **You can use different windows to control the stopband attenuation while keeping the filter order unchanged.**

For example, consider a 100-th order lowpass FIR filter with a cutoff frequency of 60 Hz and a sample rate of 1 kHz. Compare designs that result from using a Hamming window, and a Chebyshev window with 90 dB of sidelobe attenuation.

```
dhamming = designfilt('lowpassfir','FilterOrder',100,'CutoffFrequency',60,...
  'SampleRate',1000,'Window','hamming');

dchebwin = designfilt('lowpassfir','FilterOrder',100,'CutoffFrequency',60,...
  'SampleRate',1000,'Window',{'chebwin',90});

hfvt = fvtool(dhamming,dchebwin);
legend(hfvt,'Hamming window', 'Chebyshev window')
```




There are other ways in which you can specify a filter with fixed order: fixed cutoff frequency, passband ripple, and stopband attenuation; fixed transition width; and fixed half-power (3dB) frequency.

### IIR Filter Design

One of the drawbacks of FIR filters is that they require a large filter order to meet some design specifications. If the ripples are kept constant, the filter order grows inversely proportional to the transition width. By using feedback, it is possible to meet a set of design specifications with a far smaller filter order. This is the idea behind IIR filter design. The term "infinite impulse response" (IIR) stems from the fact that, when an impulse is applied to the filter, the output never decays to zero.

- **IIR filters are useful when computational resources are at a premium. However, stable, causal IIR filters cannot have perfectly linear phase. Avoid IIR designs in cases where phase linearity is a requirement.**

Another important reason for using IIR filters is their small group delay relative to FIR filters, which results in a shorter transient response.

### Butterworth Filters

Butterworth filters are maximally flat IIR filters. The flatness in the passband and stopband causes the transition band to be very wide. Large orders are required to obtain filters with narrow transition widths.

Design a minimum-order Butterworth filter with passband frequency 100 Hz, stopband frequency 300 Hz, maximum passband ripple 1 dB, and 60 dB stopband attenuation. The sample rate is 2 kHz.

```
Fp = 100;
Fst = 300;
Ap = 1;
Ast = 60;
Fs = 2e3;

dbutter = designfilt('lowpassiir','PassbandFrequency',Fp,...
  'StopbandFrequency',Fst,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs,'DesignMethod','butter');
```

### Chebyshev Type I Filters

Chebyshev Type I filters attain smaller transition widths than Butterworth filters of the same order by allowing for passband ripple.

- **Butterworth and Chebyshev Type I filters both have maximally flat stopbands. For a given filter order, the tradeoff is between passband ripple and transition width.**

Design a Chebyshev Type I filter with the same specifications as the Butterworth filter above.

```
dcheby1 = designfilt('lowpassiir','PassbandFrequency',Fp,...
  'StopbandFrequency',Fst,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs,'DesignMethod','cheby1');
```

### Chebyshev Type II Filters

- **Chebyshev Type II filters have maximally flat passbands and equiripple stopbands.**

Since extremely large attenuations are typically not required, we may be able to attain the required transition width with a relatively small order by allowing for some stopband ripple.

Design a minimum-order Chebyshev Type II filter with the same specifications as in the previous examples.

```
dcheby2 = designfilt('lowpassiir','PassbandFrequency',Fp,...
  'StopbandFrequency',Fst,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs,'DesignMethod','cheby2');
```

### Elliptic Filters

Elliptic filters generalize Chebyshev and Butterworth filters by allowing for ripple in both the passband and the stopband. As ripples are made smaller, elliptic filters can approximate arbitrarily close the magnitude and phase response of either Chebyshev or Butterworth filters.

- **Elliptic filters attain a given transition width with the smallest order.**

```
dellip = designfilt('lowpassiir','PassbandFrequency',Fp,...
  'StopbandFrequency',Fst,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs,'DesignMethod','ellip');
```

Compare the response and the order of the four IIR filters.

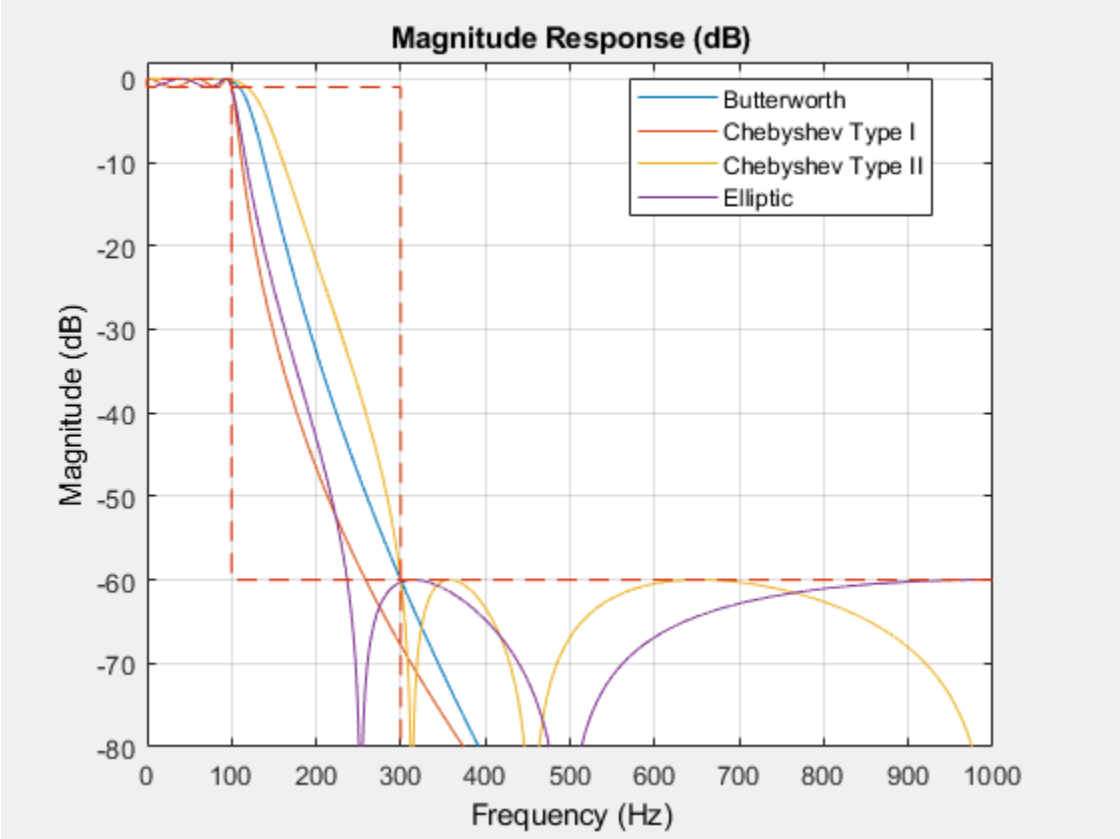
- **For the same specification constraints, the Butterworth method yields the highest order and the elliptic method yields the smallest.**

```
FilterOrders = [filtord(dbutter) filtord(dcheby1) filtord(dcheby2) filtord(dellip)]
```

```
FilterOrders = 1×4
```

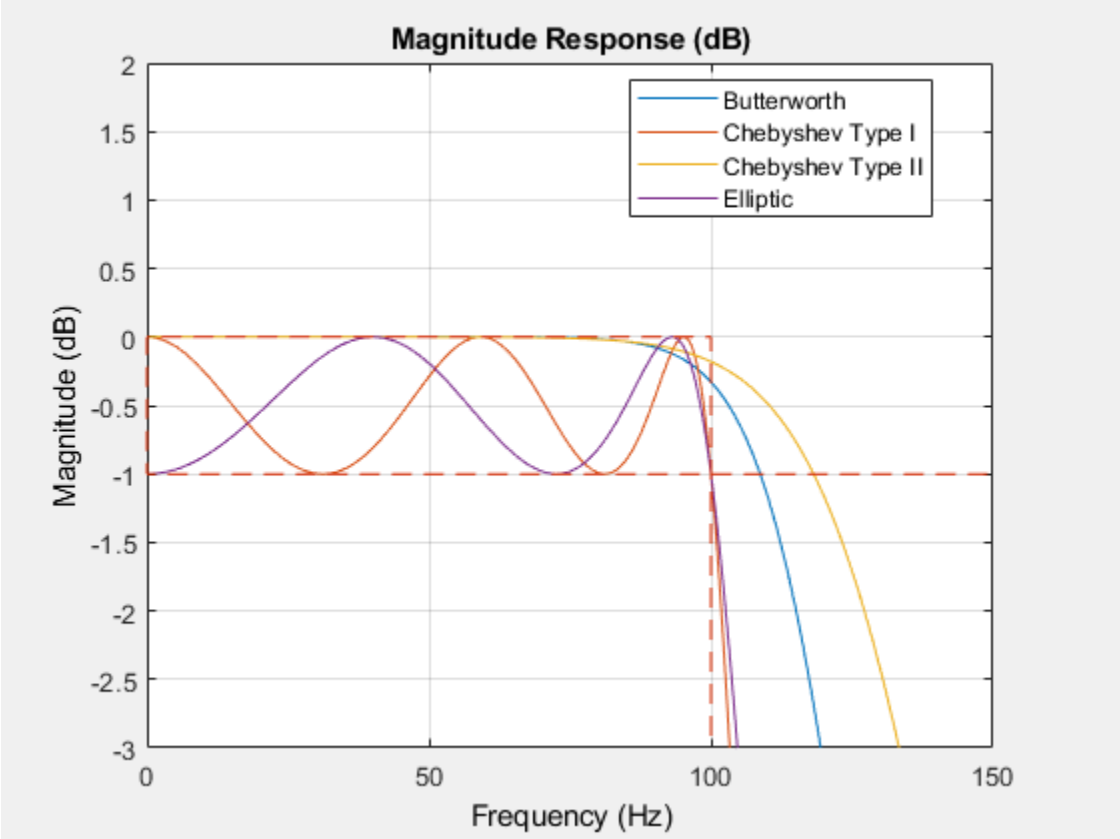
```
     7       5       5       4
```

```
hfvt = fvtool(dbutter,dcheby1,dcheby2,dellip);
axis([0 1e3 -80 2]);
legend(hfvt,'Butterworth', 'Chebyshev Type I',...
  'Chebyshev Type II','Elliptic')
```




Zoom into the passband to see the ripple differences.

```
axis([0 150 -3 2]);
```



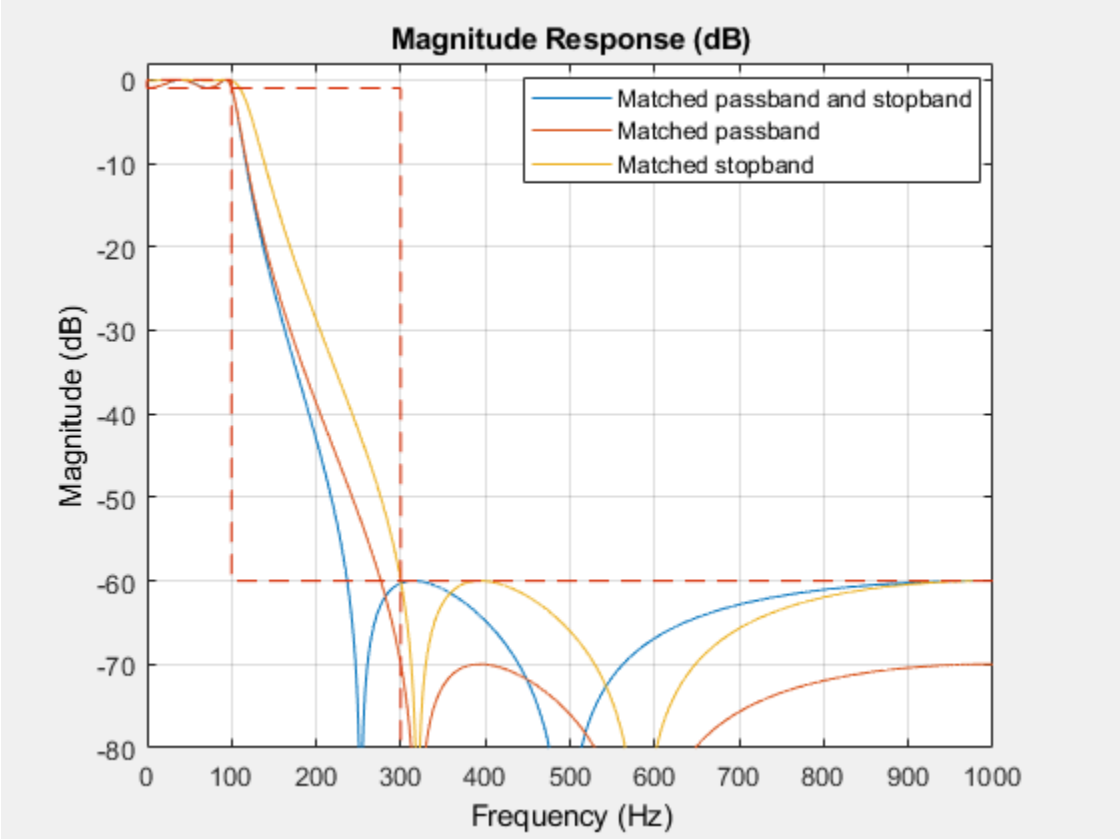

### Matching Exactly the Passband or Stopband Specifications

With minimum-order designs, the ideal order needs to be rounded to the next integer. This additional fractional order allows the algorithm to actually exceed the specifications.

- **Use the `'MatchExactly'` parameter to constrain the design algorithm to match one band exactly. The other band exceeds its specification.**

By default, Chebyshev Type I designs match the passband, Butterworth and Chebyshev Type II match the stopband, and elliptic designs match both the passband and the stopband (while the stopband edge frequency is exceeded):

```
dellip1 = designfilt('lowpassiir','PassbandFrequency',Fp,...
  'StopbandFrequency',Fst,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs,'DesignMethod','ellip',...
  'MatchExactly','passband');

dellip2 = designfilt('lowpassiir','PassbandFrequency',Fp,...
  'StopbandFrequency',Fst,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs,'DesignMethod','ellip',...
  'MatchExactly','stopband');

hfvt = fvtool(dellip, dellip1, dellip2);
legend(hfvt,'Matched passband and stopband','Matched passband',...
  'Matched stopband');
axis([0 1e3 -80 2]);
```




The matched-passband and matched-both designs have a ripple of exactly 1 dB at the passband frequency value of 100 Hz.

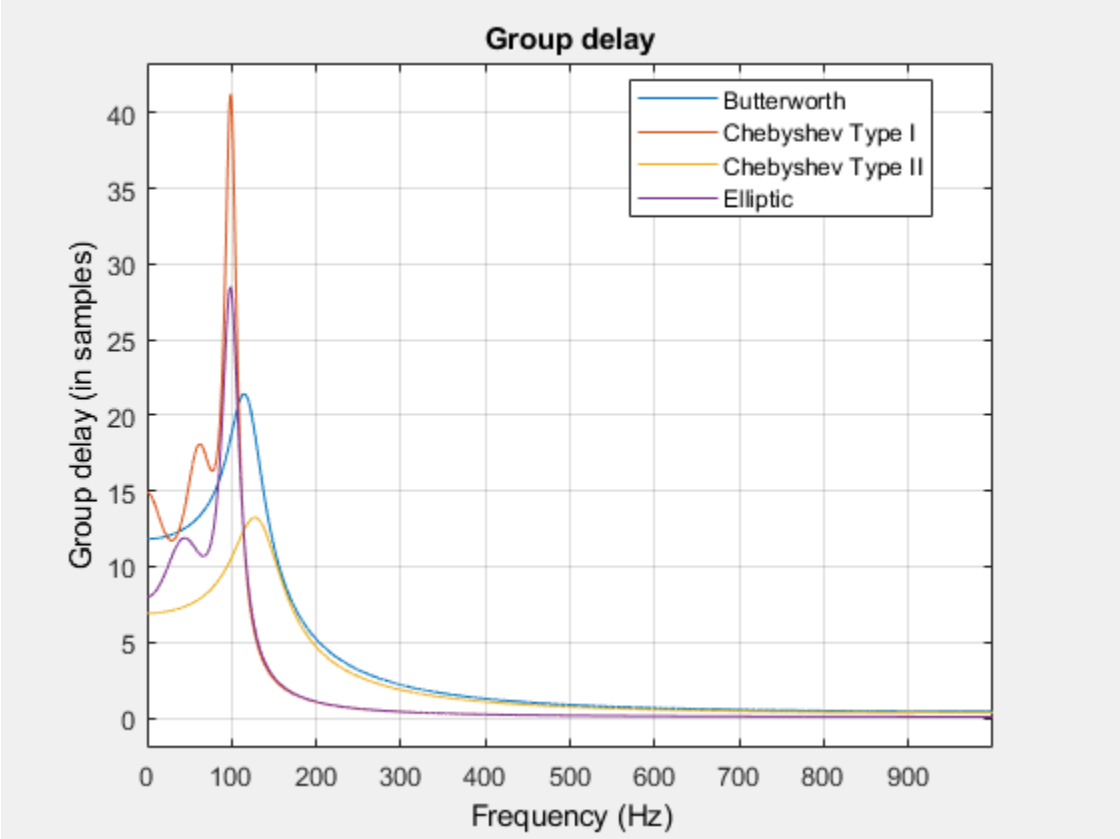
**Group Delay Comparison**

With IIR filters, we need to consider not only the ripple/transition width tradeoff, but also the degree of phase distortion. We know that it is impossible to have linear-phase throughout the entire Nyquist interval. Thus we may want to see how far from linear the phase response is. A good way to do this is to look at the (ideally constant) group delay and see how flat it is.

Compare the group delay of the four IIR filters designed above.

- **If phase is an issue, keep in mind that Butterworth and Chebyshev Type II designs have the flattest group delay and thus introduce the least distortion.**

```
hfvt = fvtool(dbutter,dcheby1,dcheby2,dellip,'Analysis','grpdelay');
legend(hfvt,'Butterworth', 'Chebyshev Type I',...
  'Chebyshev Type II','Elliptic')
```




### Conclusions

In this example, you learned how to use `designfilt` to obtain a variety of lowpass FIR and IIR filters with different constraints and design methods. `designfilt` can also be used to obtain highpass, bandpass, bandstop, arbitrary-magnitude, differentiator, and Hilbert designs. To learn more about all available options, “Filter Design Gallery” on page 24-141.

### Further Reading

For more information on filter design and analysis, see the Signal Processing Toolbox® software documentation. For more information on filter applications see “Practical Introduction to Digital Filtering” on page 24-174.

## See Also

**FVTool** | `designfilt` | `filtord`

# Practical Introduction to Digital Filtering

This example shows how to design, analyze, and apply a digital filter to your data. It will help you answer questions such as: how do I compensate for the delay introduced by a filter?, How do I avoid distorting my signal?, How do I remove unwanted content from my signal?, How do I differentiate my signal?, and How do I integrate my signal?

Filters can be used to shape the signal spectrum in a desired way or to perform mathematical operations such as differentiation and integration. In what follows you will learn some practical concepts that will ease the use of filters when you need them.

This example focuses on applications of digital filters rather than on their design. If you want to learn more about how to design digital filters see the “Practical Introduction to Digital Filter Design” on page 24-161 example.

### Compensating for Delay Introduced by Filtering

Digital filters introduce delay in your signal. Depending on the filter characteristics, the delay can be constant over all frequencies, or it can vary with frequency. The type of delay determines the actions you have to take to compensate for it. The `grpdelay` function allows you to look at the filter delay as a function of frequency. Looking at the output of this function allows you to identify if the delay of the filter is constant or if it varies with frequency (in other words, if it is frequency-dependent).

Filter delay that is constant over all frequencies can be easily compensated for by shifting the signal in time. FIR filters usually have constant delay. On the other hand, delay that varies with frequency causes phase distortion and can alter a signal waveform significantly. Compensating for frequency-dependent delay is not as trivial as for the constant delay case. IIR filters introduce frequency-dependent delay.

### Compensating for Constant Filter Delay

As mentioned before, you can measure the group of delay of the filter to verify that it is a constant function of frequency. You can use the `grpdelay` function to measure the filter delay, D, and compensate for this delay by appending D zeros to the input signal and shifting the output signal in time by D samples.

Consider a noisy electrocardiogram signal that you want to filter to remove high frequency noise above 75 Hz. You want to apply an FIR lowpass filter and compensate for the filter delay so that the noisy and filtered signals are aligned correctly and can be plotted on top of each other for comparison.

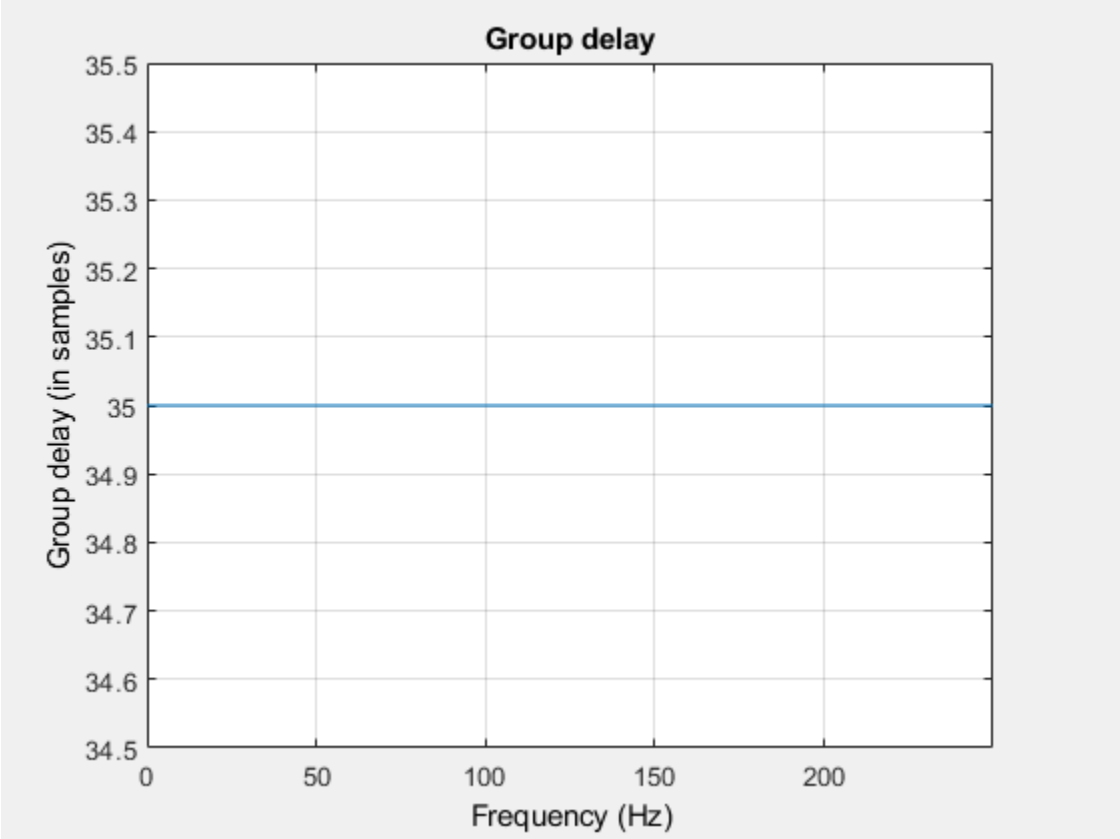
```
Fs = 500;                    % Sample rate in Hz
N = 500;                     % Number of signal samples
rng default;
x = ecg(N)'+0.25*randn(N,1); % Noisy waveform
t = (0:N-1)/Fs;              % Time vector
```

Design a 70th-order lowpass FIR filter with a cutoff frequency of 75 Hz.

```
Fnorm = 75/(Fs/2);           % Normalized frequency
df = designfilt('lowpassfir','FilterOrder',70,'CutoffFrequency',Fnorm);
```

Plot the group delay of the filter to verify that it is constant across all frequencies indicating that the filter is linear phase. Use the group delay to measure the delay of the filter.

```
grpdelay(df,2048,Fs)          % Plot group delay
```

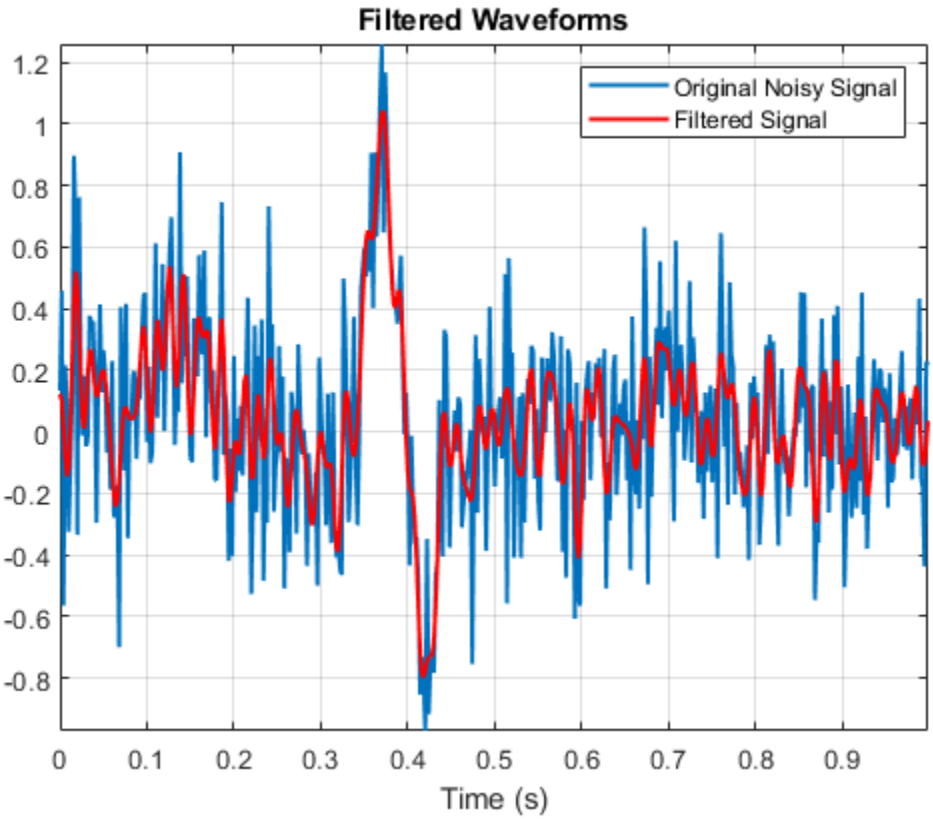



```
D = mean(grpdelay(df)) % Filter delay in samples
```

```
D = 35
```

Before filtering, append D zeros at the end of the input data vector, x. This ensures that all the useful samples are flushed out of the filter, and that the input signal and the delay-compensated output signal have the same length. Filter the data and compensate for the delay by shifting the output signal by D samples. This last step effectively removes the filter transient.

```
y = filter(df,[x; zeros(D,1)]);  % Append D zeros to the input data
y = y(D+1:end);                  % Shift data to compensate for delay

plot(t,x,t,y,'r','linewidth',1.5)
title('Filtered Waveforms')
xlabel('Time (s)')
legend('Original Noisy Signal','Filtered Signal')
grid on
axis tight
```




**Compensating for Frequency-Dependent Delay**

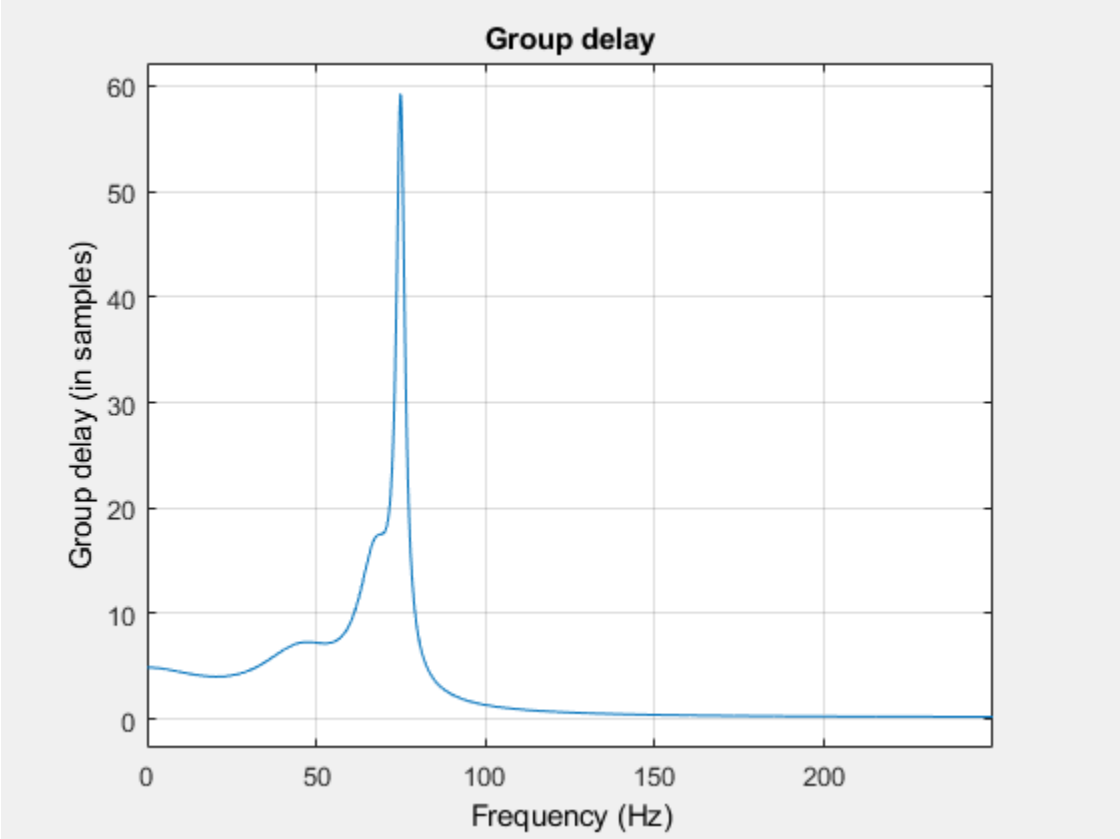
Frequency-dependent delay causes phase distortion in the signal. Compensating for this type of delay is not as trivial as for the constant delay case. If your application allows off-line processing, you can remove the frequency-dependent delay by implementing zero-phase filtering using the `filtfilt` function. `filtfilt` performs zero-phase filtering by processing the input data in both the forward and reverse directions. The main effect is that you obtain zero-phase distortion, i.e., you filter data with an equivalent filter that has a constant delay of 0 samples. Other effects are that you get a filter transfer function which equals the squared magnitude of the original filter transfer function, and a filter order that is double the order of the original filter.

Consider the ECG signal defined in the previous section. Filter this signal with and without delay compensation. Design a 7th-order lowpass IIR elliptic filter with a cutoff frequency of 75 Hz.

```
Fnorm = 75/(Fs/2); % Normalized frequency
df = designfilt('lowpassiir',...
                'PassbandFrequency',Fnorm,...
                'FilterOrder',7,...
                'PassbandRipple',1,...
                'StopbandAttenuation',60);
```

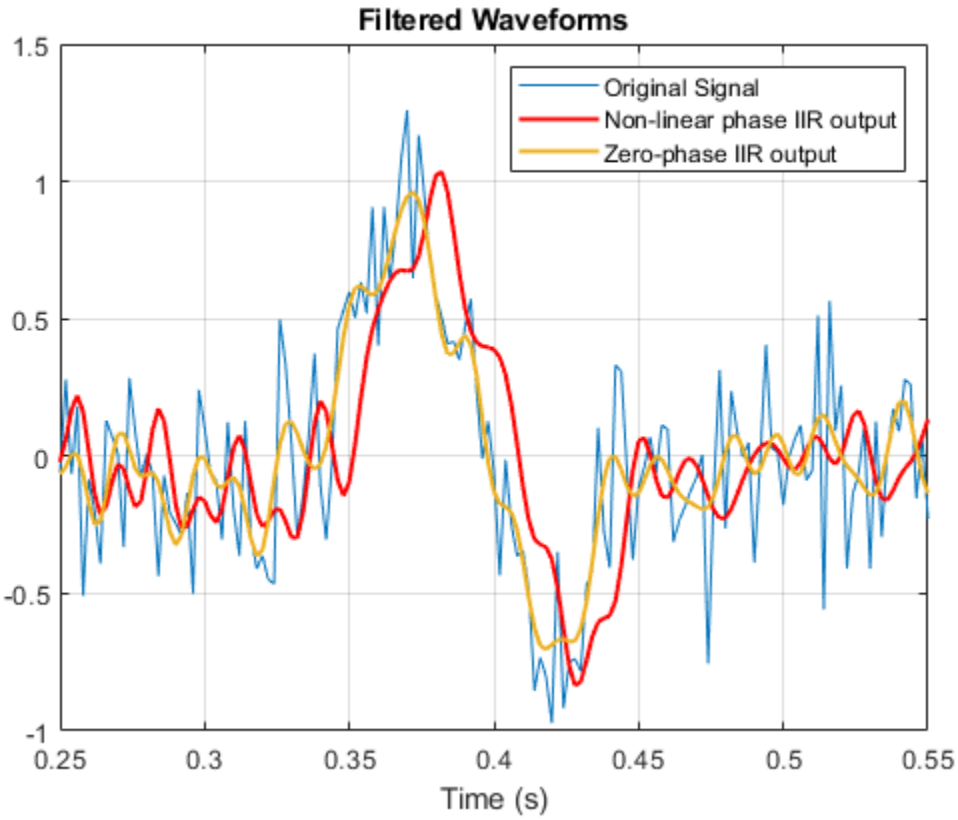
Plot the group delay of the filter. The group delay varies with frequency, indicating that the filter delay is frequency-dependent.

```
grpdelay(df,2048,'half',Fs)
```




Filter the data and look at the effects of each filter implementation on the time signal. Zero-phase filtering effectively removes the filter delay.

```
y1 = filter(df,x);    % Nonlinear phase filter - no delay compensation
y2 = filtfilt(df,x);  % Zero-phase implementation - delay compensation

plot(t,x)
hold on
plot(t,y1,'r','linewidth',1.5)
plot(t,y2,'linewidth',1.5)
title('Filtered Waveforms')
xlabel('Time (s)')
legend('Original Signal','Non-linear phase IIR output',...
  'Zero-phase IIR output')
xlim([0.25 0.55])
grid on
```




Zero-phase filtering is a great tool if your application allows for the non-causal forward/backward filtering operations, and for the change of the filter response to the square of the original response.

Filters that introduce constant delay are linear phase filters. Filters that introduce frequency-dependent delay are non-linear phase filters.

**Removing Unwanted Spectral Content from a Signal**

Filters are commonly used to remove unwanted spectral content from a signal. You can choose from a variety of filters to do this. You choose a lowpass filter when you want to remove high frequency content, or a highpass filter when you want to remove low frequency content. You can also choose a bandpass filter to remove low and high frequency content while leaving an intermediate band of frequencies intact. You choose a bandstop filter when you want to remove frequencies over a given band.

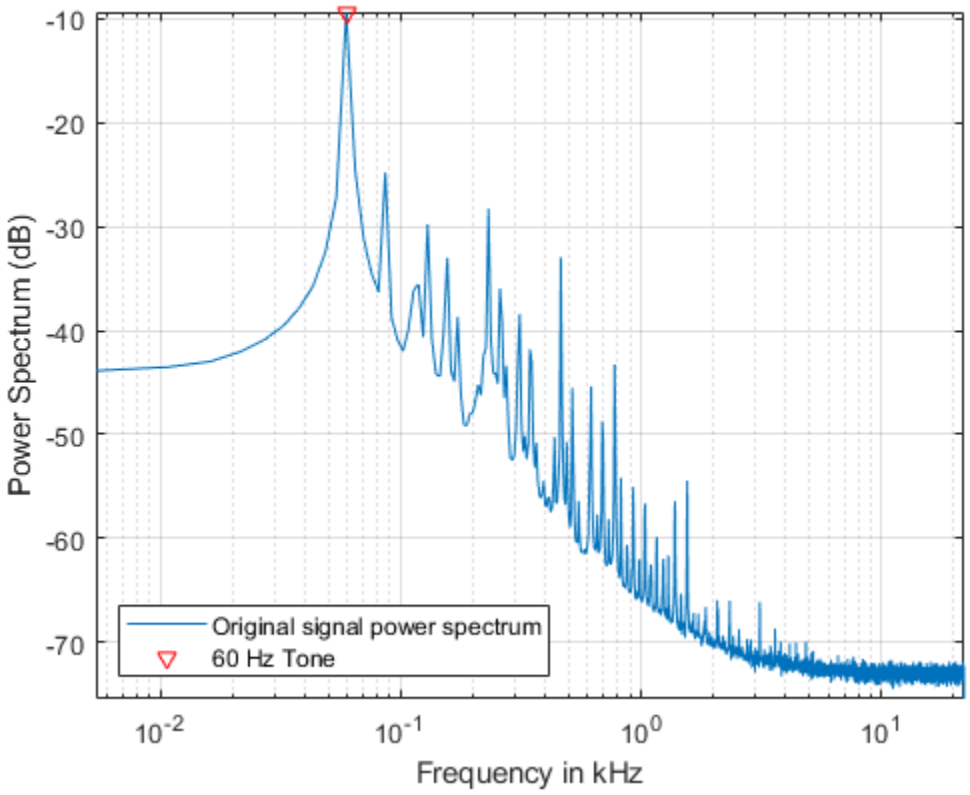
Consider an audio signal that has a power-line hum and white noise. The power-line hum is caused by a 60 Hz tone. White noise is a signal that exists across all the audio bandwidth.

Load the audio signal. Specify a sample rate of 44.1 kHz.

```
Fs = 44100;
y = audioread('noisymusic.wav');
```

Plot the power spectrum of the signal. The red triangular marker shows the strong 60 Hz tone interfering with the audio signal.

```
[P,F] = pwelch(y,ones(8192,1),8192/2,8192,Fs,'power');
helperFilterIntroductionPlot1(F,P,[60 60],[-9.365 -9.365],...
  {'Original signal power spectrum', '60 Hz Tone'})
```



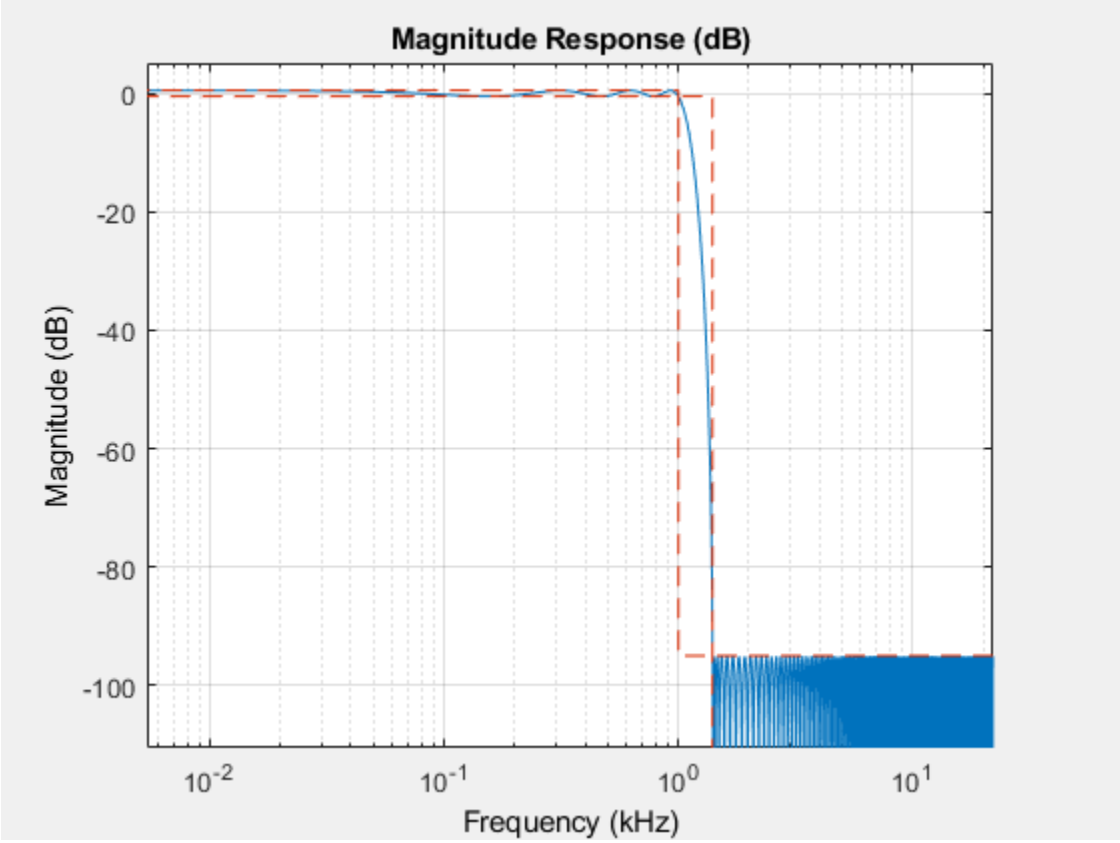

You can first remove as much white noise spectral content as possible using a lowpass filter. The passband of the filter should be set to a value that offers a good trade-off between noise reduction and audio degradation due to loss of high frequency content. Applying the lowpass filter before removing the 60 Hz hum is very convenient since you will be able to downsample the band-limited signal. The lower rate signal will allow you to design a sharper and narrower 60 Hz bandstop filter with a smaller filter order.

Design a lowpass filter with passband frequency of 1 kHz and stopband frequency of 1.4 kHz. Choose a minimum-order design.

```
Fp = 1e3;    % Passband frequency in Hz
Fst = 1.4e3; % Stopband frequency in Hz
Ap = 1;      % Passband ripple in dB
Ast = 95;    % Stopband attenuation in dB

df = designfilt('lowpassfir','PassbandFrequency',Fp,...
                'StopbandFrequency',Fst,'PassbandRipple',Ap,...
                'StopbandAttenuation',Ast,'SampleRate',Fs);
```

Analyze the filter response.

```
fvtool(df,'Fs',Fs,'FrequencyScale','log',...
  'FrequencyRange','Specify freq. vector','FrequencyVector',F)
```

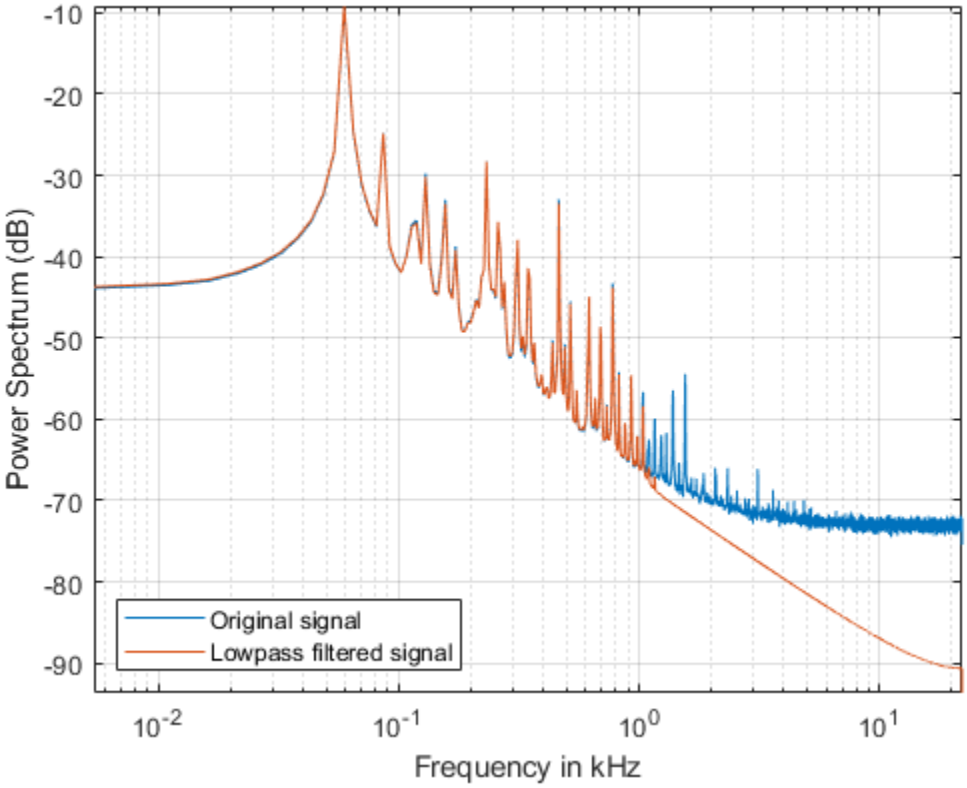



Filter the data and compensate for delay.

```
D = mean(grpdelay(df)); % Filter delay
ylp = filter(df,[y; zeros(D,1)]);
ylp = ylp(D+1:end);
```

Look at the spectrum of the lowpass filtered signal. The frequency content above 1400 Hz has been removed.

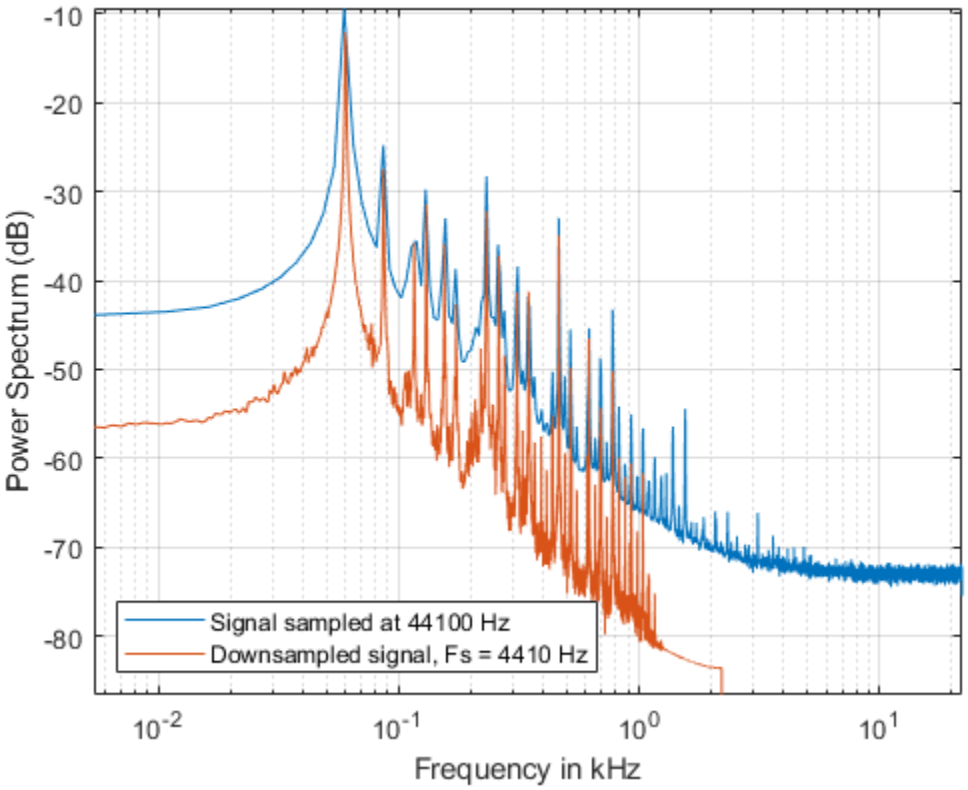
```
[Plp,Flp] = pwelch(ylp,ones(8192,1),8192/2,8192,Fs,'power');
helperFilterIntroductionPlot1(F,P,Flp,Plp,...
  {'Original signal','Lowpass filtered signal'})
```




From the power spectrum plot above, you can see that the maximum non-negligible frequency content of the lowpass filtered signal is at 1400 Hz. By the sampling theorem, a sample rate of $2 \times 1400 = 2800$ Hz would suffice to represent the signal correctly, you however, are using a sample rate of 44100 Hz which is a waste since you will need to process more samples than those necessary. You can downsample the signal to reduce the sample rate and reduce the computational load by reducing the number of samples that you need to process. A lower sample rate will also allow you to design a sharper and narrower bandstop filter, needed to remove the 60 Hz noise, with a smaller filter order.

Downsample the lowpass filtered signal by a factor of 10 to obtain a sample rate of Fs/10 = 4.41 kHz. Plot the spectrum of the signal before and after downsampling.

```
Fs = Fs/10;
yds = downsample(ylp,10);

[Pds,Fds] = pwelch(yds,ones(8192,1),8192/2,8192,Fs,'power');
helperFilterIntroductionPlot1(F,P,Fds,Pds,...
  {'Signal sampled at 44100 Hz', 'Downsampled signal, Fs = 4410 Hz'})
```

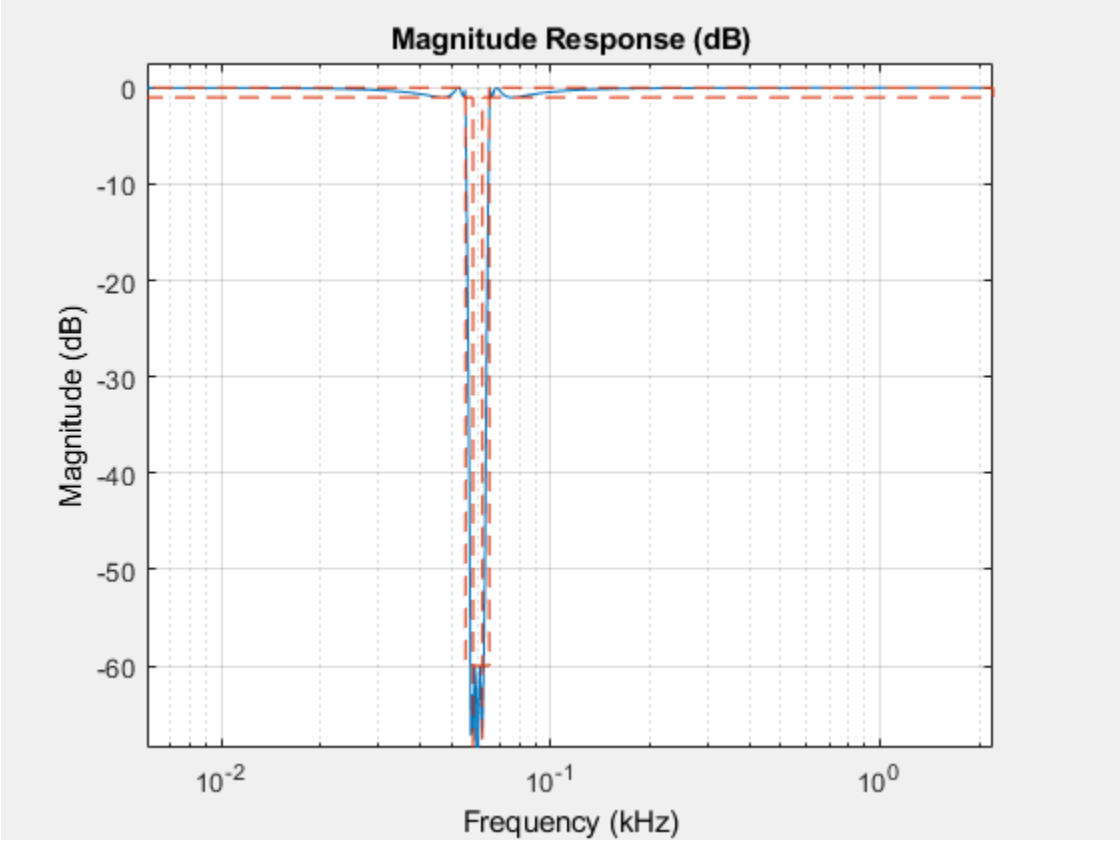



Now remove the 60 Hz tone using an IIR bandstop filter. Let the stopband have a width of 4 Hz centered at 60 Hz. Choose an IIR filter to achieve a sharp frequency notch, small passband ripple, and a relatively low order.

```
df = designfilt('bandstopiir','PassbandFrequency1',55,...
                'StopbandFrequency1',58,'StopbandFrequency2',62,...
                'PassbandFrequency2',65,'PassbandRipple1',1,...
                'StopbandAttenuation',60,'PassbandRipple2',1,...
                'SampleRate',Fs,'DesignMethod','ellip');
```

Analyze the magnitude response.

```
fvtool(df,'Fs',Fs,'FrequencyScale','log',...
  'FrequencyRange','Specify freq. vector','FrequencyVector',Fds(Fds>F(2)))
```




Perform zero-phase filtering to avoid phase distortion. Use the `filtfilt` function to process the data.

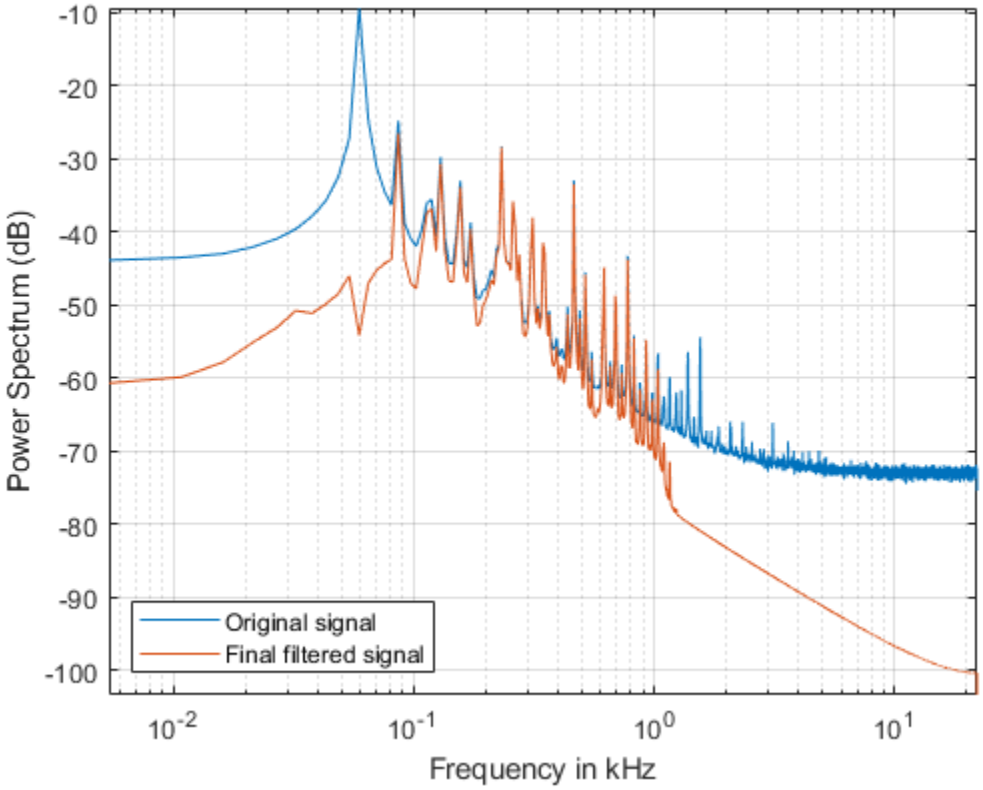
```
ybs = filtfilt(df,yds);
```

Finally, upsample the signal to bring it back to the original audio sample rate of 44.1 kHz which is compatible with audio soundcards.

```
yf = interp(ybs,10);
Fs = Fs*10;
```

Take a final look at the spectrum of the original and processed signals. The high frequency noise floor and the 60 Hz tone have been attenuated by the filters.

```
[Pfinal,Ffinal] = pwelch(yf,ones(8192,1),8192/2,8192,Fs,'power');
helperFilterIntroductionPlot1(F,P,Ffinal,Pfinal,...
  {'Original signal','Final filtered signal'})
```




If your computer has a sound card, you can listen to the signal before and after processing. As mentioned above, the end result is that you have effectively attenuated the 60 Hz hum and the high-frequency noise in the audio file.

```
% To play the original signal, uncomment the next two lines
% hplayer = audioplayer(y, Fs);
% play(hplayer)

% To play the noise-reduced signal, uncomment the next two lines
% hplayer = audioplayer(yf, Fs);
% play(hplayer);
```

**Differentiating a Signal**

The MATLAB `diff` function differentiates a signal with the drawback that you can potentially increase the noise levels at the output. A better option is to use a differentiator filter that acts as a differentiator in the band of interest, and as an attenuator at all other frequencies, effectively removing high frequency noise.

As an example, analyze the speed of displacement of a building floor during an earthquake. Displacement or drift measurements were recorded on the first floor of a three story test structure under earthquake conditions and saved in the quakedrift.mat file. The length of the data vector is 10e3, the sample rate is 1 kHz, and the units of the measurements are cm.

Differentiate the displacement data to obtain estimates of the speed and acceleration of the building floor during the earthquake. Compare the results using diff and an FIR differentiator filter.

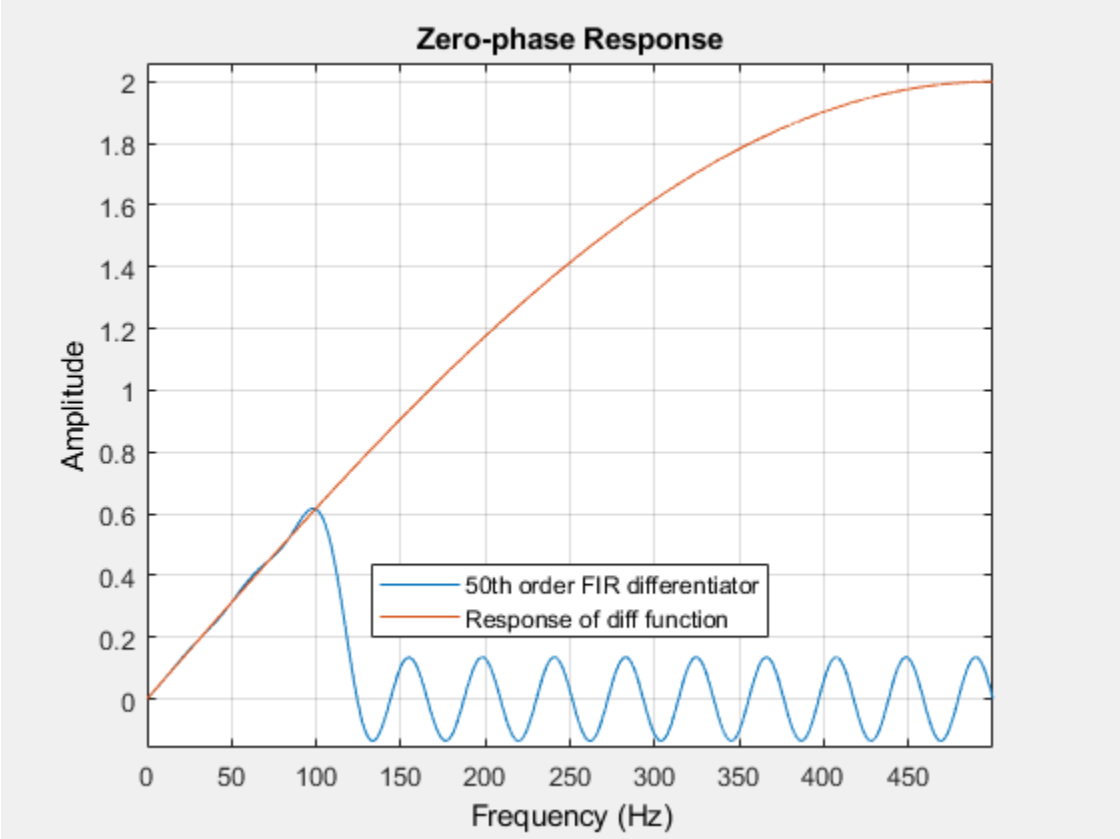
```
load quakedrift.mat

Fs  = 1000;                 % Sample rate
dt = 1/Fs;                  % Time differential
t = (0:length(drift)-1)*dt; % Time vector
```

Design a 50th-order differentiator filter with a passband frequency of 100 Hz, which is the bandwidth over which most of the signal energy is found. Set the stopband frequency of the filter to 120 Hz.

```
df = designfilt('differentiatorfir','FilterOrder',50,...
                'PassbandFrequency',100,'StopbandFrequency',120,...
                'SampleRate',Fs);
```

The `diff` function can be seen as a first order FIR filter with response $H(Z) = 1 - Z^{-1}$. Use FVTool to compare the magnitude response of the 50th-order differentiator FIR filter and the response of the `diff` function. Clearly, both responses are equivalent in the passband region (from 0 to 100 Hz). However, in the stopband region, the 50th-order filter attenuates components while the diff response amplifies components. This effectively increases the levels of high frequency noise.

```
hfvt = fvtool(df,[1 -1],1,'MagnitudeDisplay','zero-phase','Fs',Fs);
legend(hfvt,'50th order FIR differentiator','Response of diff function');
```




Differentiate using the `diff` function. Add zeros to compensate for the missing samples due to the diff operation.

```
v1 = diff(drift)/dt;
a1 = diff(v1)/dt;
```

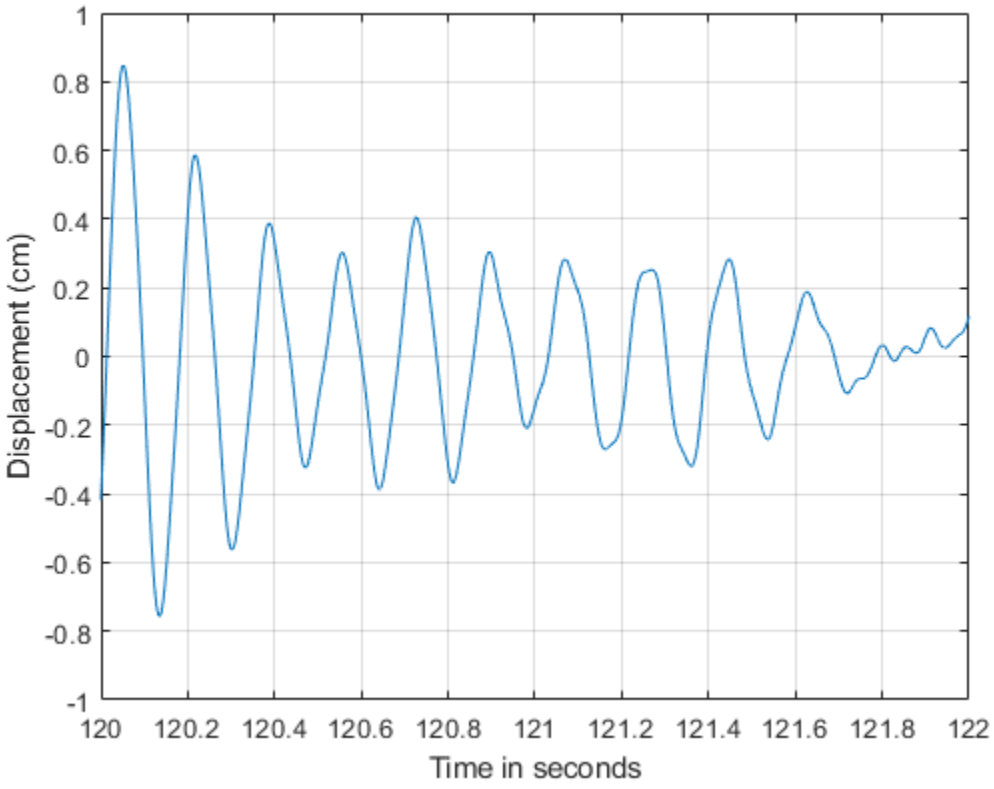
```
v1 = [0; v1];
a1 = [0; 0; a1];
```

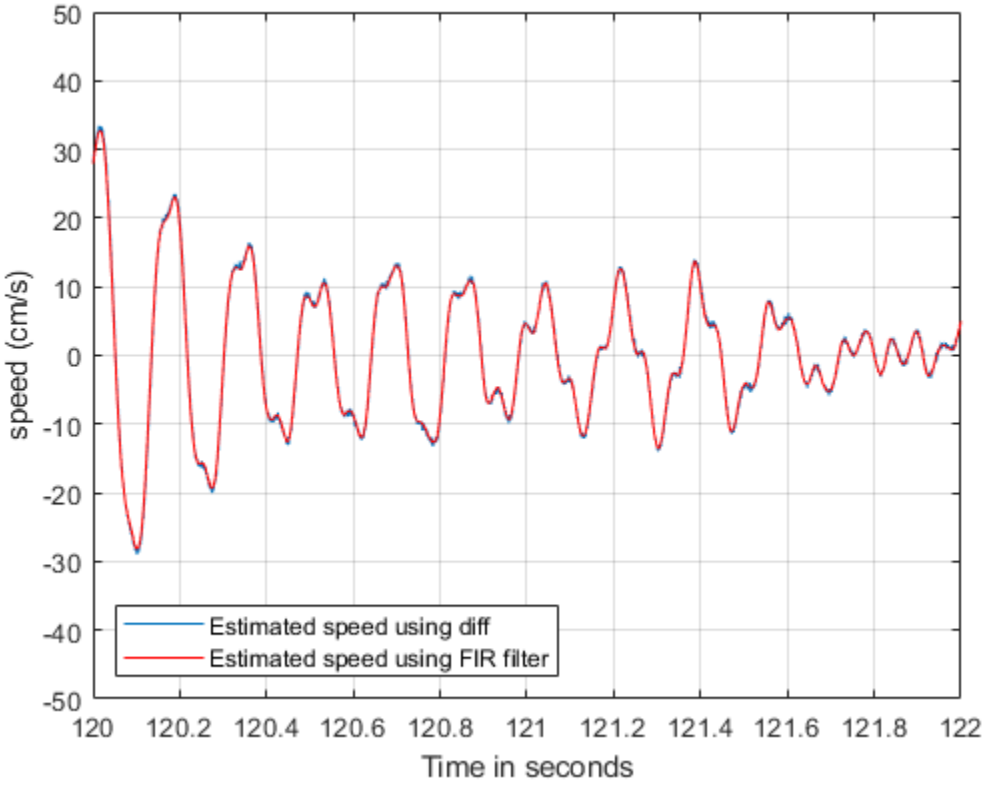
Differentiate using the 50th order FIR filter and compensate for delay.

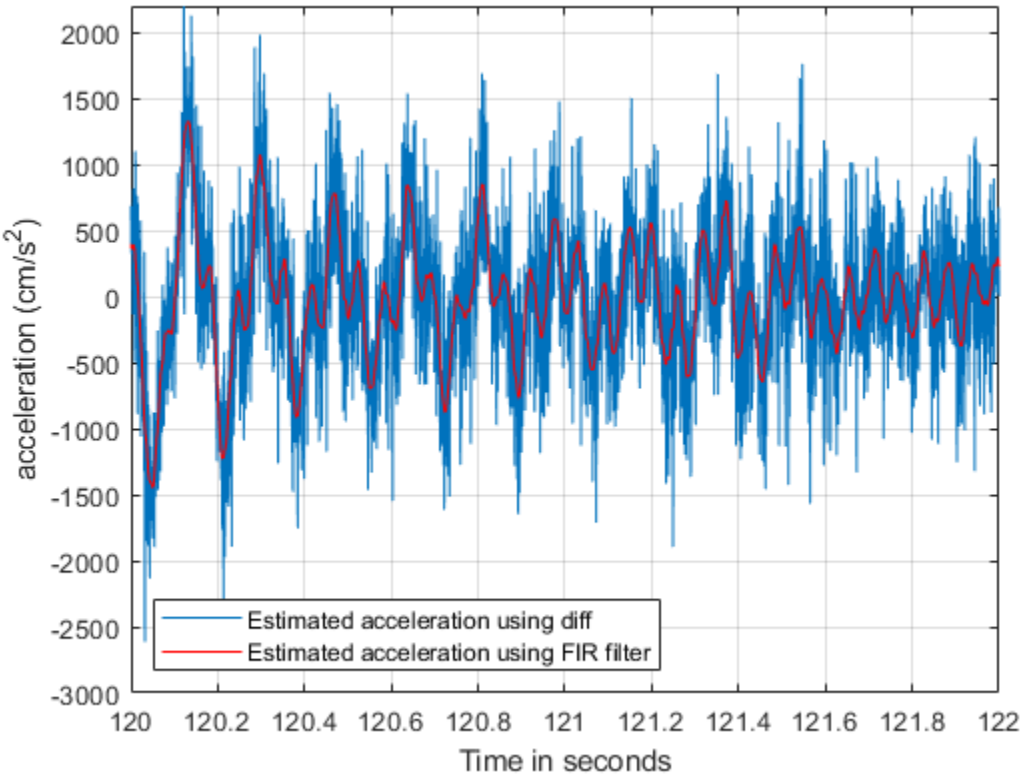
```
D = mean(grpdelay(df)); % Filter delay
v2 = filter(df,[drift; zeros(D,1)]);
v2 = v2(D+1:end);
a2 = filter(df,[v2; zeros(D,1)]);
a2 = a2(D+1:end);
v2 = v2/dt;
a2 = a2/dt^2;
```

Plot a few data points of the floor displacement. Plot also a few data points of the speed and acceleration as computed with diff and with the 50th order FIR filter. Notice how the noise has been slightly amplified in the speed estimates and largely amplified in the acceleration estimates obtained with `diff`.

```
helperFilterIntroductionPlot2(t,drift,v1,v2,a1,a2)
```




50
40
30
20
10
0
-10
-20
-30
-40
-50
speed (cm/s)
120
120.2
120.4
120.6
120.8
121
121.2
121.4
121.6
121.8
122
Time in seconds
Estimated speed using diff
Estimated speed using FIR filter

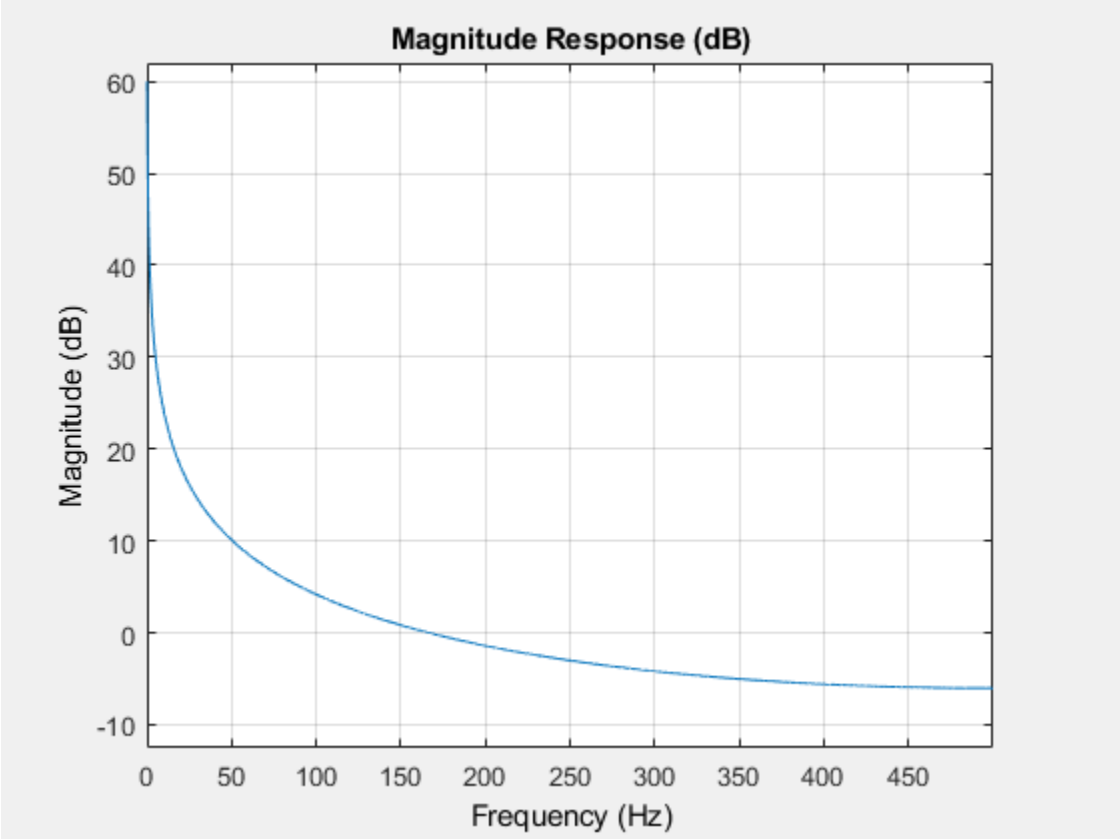



### Integrating a Signal

A leaky integrator filter is an all-pole filter with transfer function $H(Z) = 1/[1 - cZ^{-1}]$ where $c$ is a constant that must be smaller than 1 to ensure stability of the filter. It is no surprise that as $c$ approaches one, the leaky integrator approaches the inverse of the `diff` transfer function. Apply the leaky integrator to the acceleration and speed estimates obtained in the previous section to get back the speed and the drift respectively. Use the estimates obtained with **the** `diff` function since they are noisier.

Use a leaky integrator with $a = 0.999$. Plot the magnitude response of the leaky integrator filter. The filter acts as a lowpass filter effectively eliminating high-frequency noise.

```
fvtool(1,[1 -.999],'Fs',Fs)
```

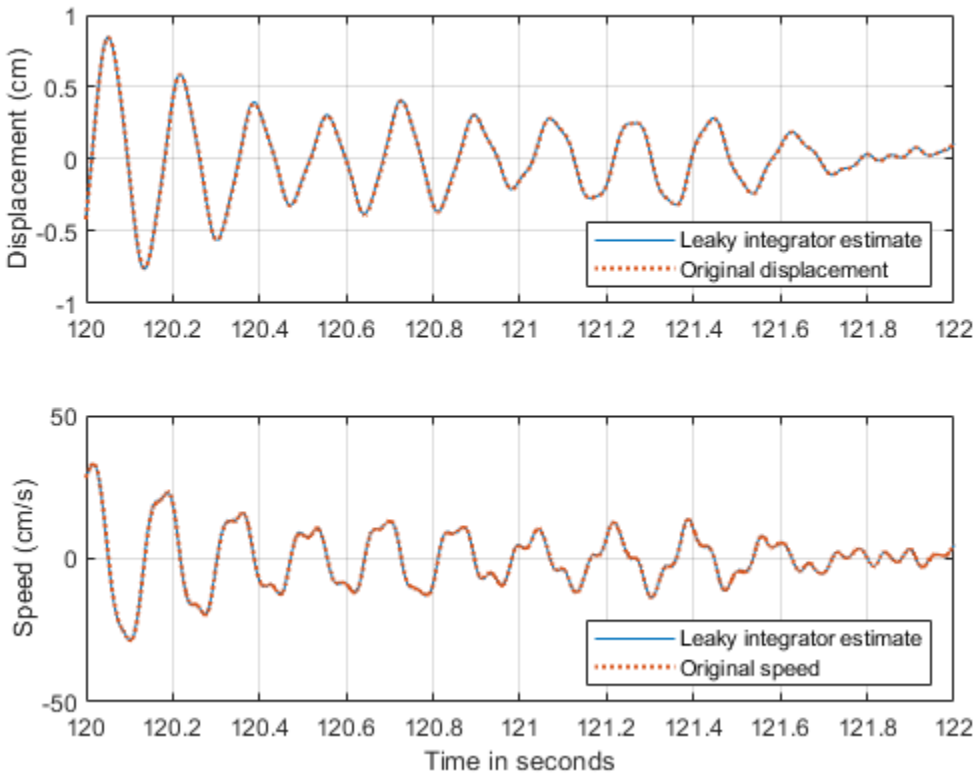



Filter the velocity and acceleration with the leaky integrator. Multiply by time differential.

```
v_original = v1;
a_original = a1;

d_leakyint = filter(1,[1 -0.999],v_original);
v_leakyint = filter(1,[1 -0.999],a_original);

d_leakyint = d_leakyint * dt;
v_leakyint = v_leakyint * dt;
```

Plot the displacement and speed estimates and compare to the original signals.

```
helperFilterIntroductionPlot3(t,drift,d_leakyint,v_original,v_leakyint)
```




You can also integrate a signal using the `cumsum` and `cumtrapz` functions. Results will be similar to those obtained with the leaky integrator.

### Conclusions

In this example you learned about linear and nonlinear phase filters and you learned how to compensate for the phase delay introduced by each filter type. You also learned how to apply filters to remove unwanted frequency components from a signal, and how to downsample a signal after limiting its bandwidth with a lowpass filter. Finally, you learned how to differentiate and integrate a signal using digital filter designs. Throughout the example you also learned how to use analysis tools to look at the response and group delay of your filters.

For more information on filter applications, see the Signal Processing Toolbox™ documentation. For more information on how to design digital filters see the “Practical Introduction to Digital Filter Design” on page 24-161 example.

### References

Proakis, J. G., and D. G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Orfanidis, S. J. *Introduction To Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

### Appendix

The following helper functions are used in this example:

- helperFilterIntroductionPlot1.m
- helperFilterIntroductionPlot2.m
- helperFilterIntroductionPlot3.m.

## See Also

`bandpass` | `bandstop` | `designfilt` | `fftfilt` | `filter` | `filtfilt` | `grpdelay` | `highpass` | `lowpass`

# Introduction to Filter Designer

This example shows how to use Filter Designer as a convenient alternative to the command-line filter design functions.

Filter Designer is a powerful graphical user interface (GUI) in Signal Processing Toolbox™ for designing and analyzing filters.

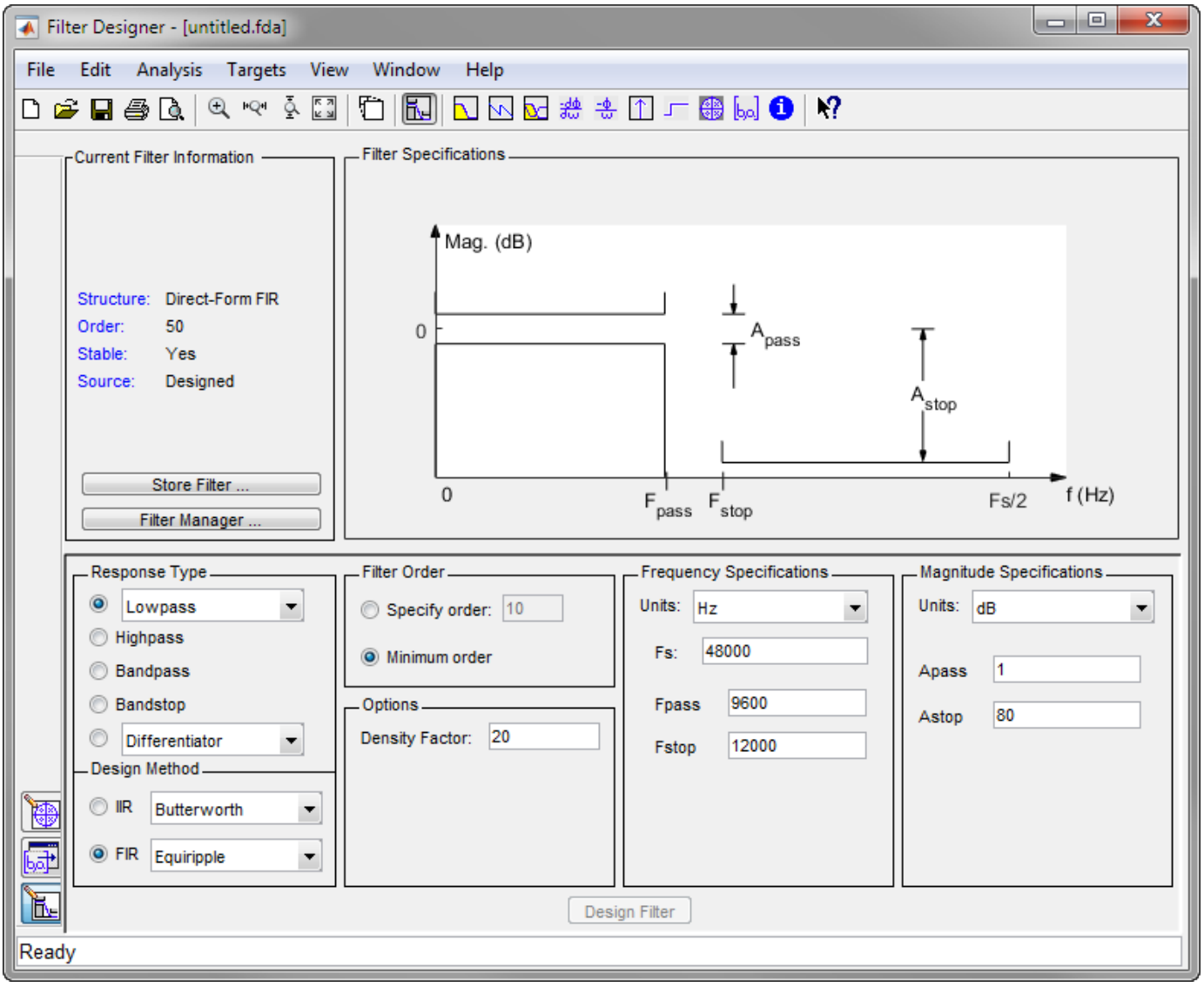
Filter Designer enables you to quickly design digital FIR or IIR filters by setting filter performance specifications, by importing filters from your MATLAB® workspace or by adding, moving, or deleting poles and zeros. Filter Designer also provides tools for analyzing filters, such as magnitude and phase response plots and pole-zero plots.

### Getting Started

Type `filterDesigner` at the MATLAB command prompt:

```
>> filterDesigner
```

A **Tip of the Day** dialog displays with suggestions for using Filter Designer. Then, the GUI displays with a default filter.




The GUI has three main regions:

- The Current Filter Information region
- The Filter Display region and
- The Design panel

The upper half of the GUI displays information on filter specifications and responses for the current filter. The Current Filter Information region, in the upper left, displays filter properties, namely the filter structure, order, number of sections used and whether the filter is stable or not. It also provides access to the Filter manager for working with multiple filters.

The Filter Display region, in the upper right, displays various filter responses, such as, magnitude response, group delay and filter coefficients.

The lower half of the GUI is the interactive portion of Filter Designer. The Design Panel, in the lower half is where you define your filter specifications. It controls what is displayed in the other two upper regions. Other panels can be displayed in the lower half by using the sidebar buttons.

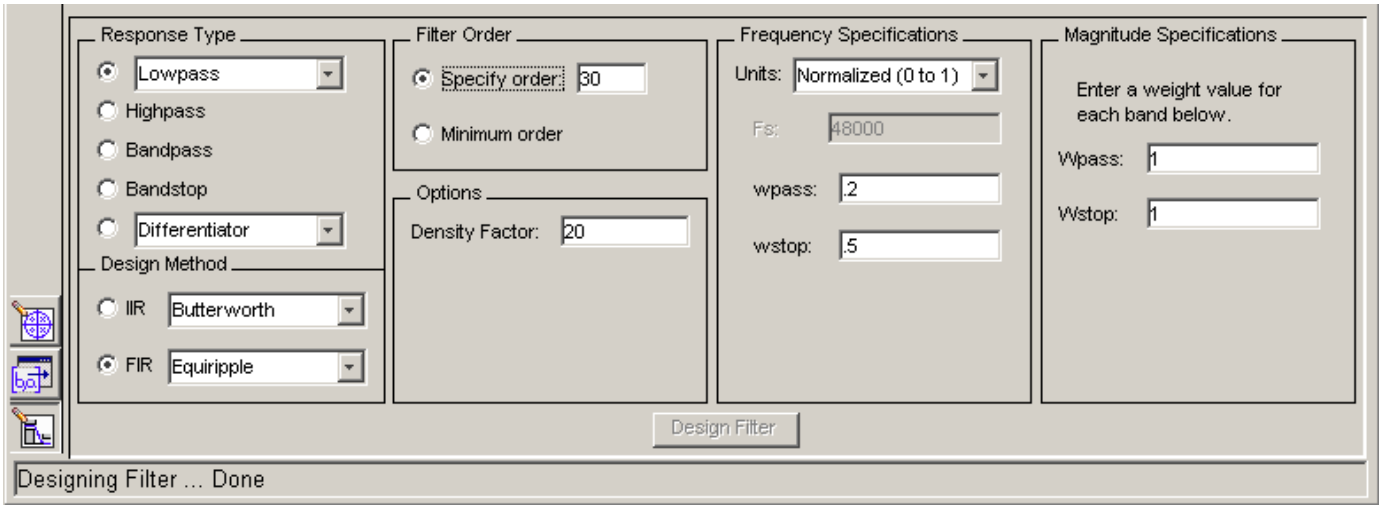
The tool includes Context-sensitive help. You can right-click or click the **What's This?** button to get information on the different parts of the tool.

### Designing a Filter

We will design a low pass filter that passes all frequencies less than or equal to 20% of the Nyquist frequency (half the sampling frequency) and attenuates frequencies greater than or equal to 50% of the Nyquist frequency. We will use an FIR Equiripple filter with these specifications:

- Passband attenuation 1 dB
- Stopband attenuation 80 dB
- A passband frequency 0.2 [Normalized (0 to 1)]
- A stopband frequency 0.5 [Normalized (0 to 1)]

To implement this design, we will use the following specifications:




1. Select **Lowpass** from the dropdown menu under **Response Type** and **Equiripple** under **FIR Design Method**. In general, when you change the Response Type or Design Method, the filter parameters and Filter Display region update automatically.

2. Select **Specify order** in the **Filter Order** area and enter **30**.

3. The FIR Equiripple filter has a **Density Factor** option which controls the density of the frequency grid. Increasing the value creates a filter which more closely approximates an ideal equiripple filter, but more time is required as the computation increases. Leave this value at 20.

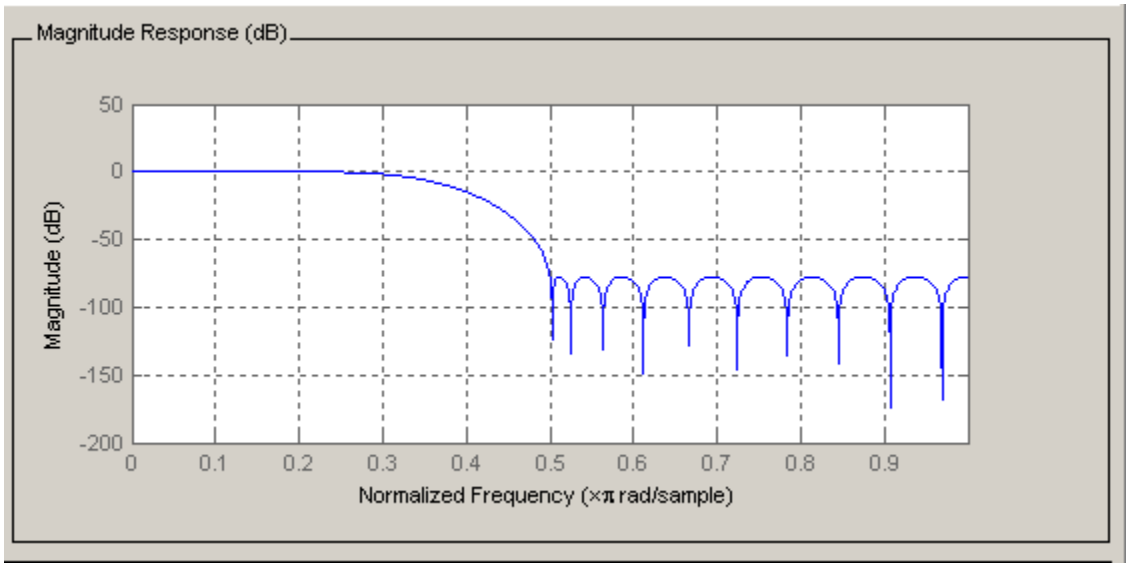
4. Select **Normalized (0 to 1)** in the Units pull down menu in the **Frequency Specifications** area.

5. Enter **0.2** for **wpass** and **0.5** for **wstop** in the **Frequency Specifications** area.

6. **Wpass** and **Wstop**, in the **Magnitude Specifications** area are positive weights, one per band, used during optimization in the FIR Equiripple filter. Leave these values at 1.

7. After setting the design specifications, click the **Design Filter** button at the bottom of the GUI to design the filter.

The magnitude response of the filter is displayed in the Filter Analysis area after the coefficients are computed.




### Viewing other Analyses

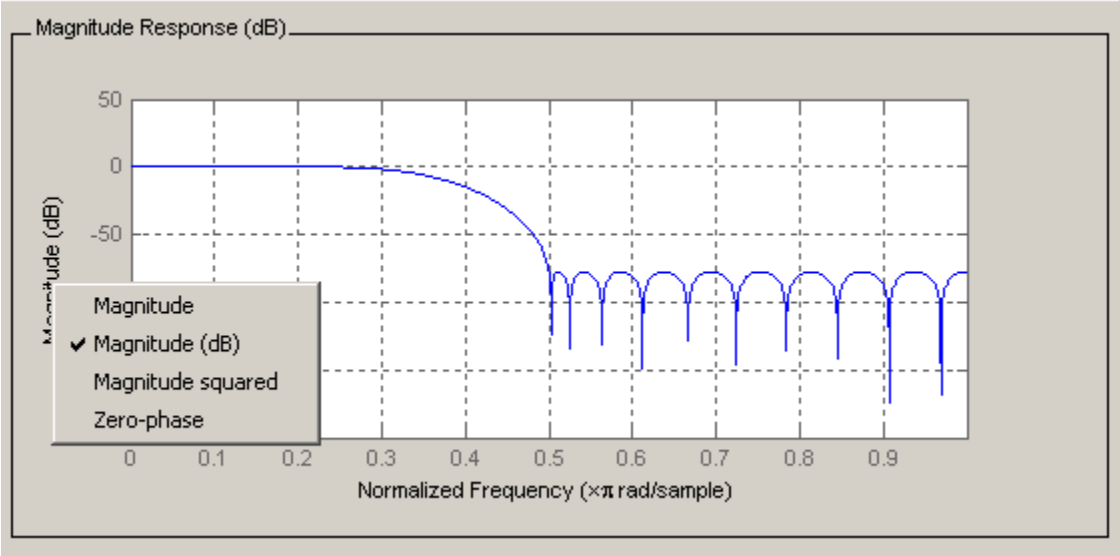
Once you have designed the filter, you can view the following filter analyses in the display window by clicking any of the buttons on the toolbar:

In order from left to right, the buttons are

- Magnitude response
- Phase response
- Magnitude and Phase responses
- Group delay response
- Phase delay response
- Impulse response
- Step response
- Pole-zero plot
- Filter Coefficients
- Filter Information

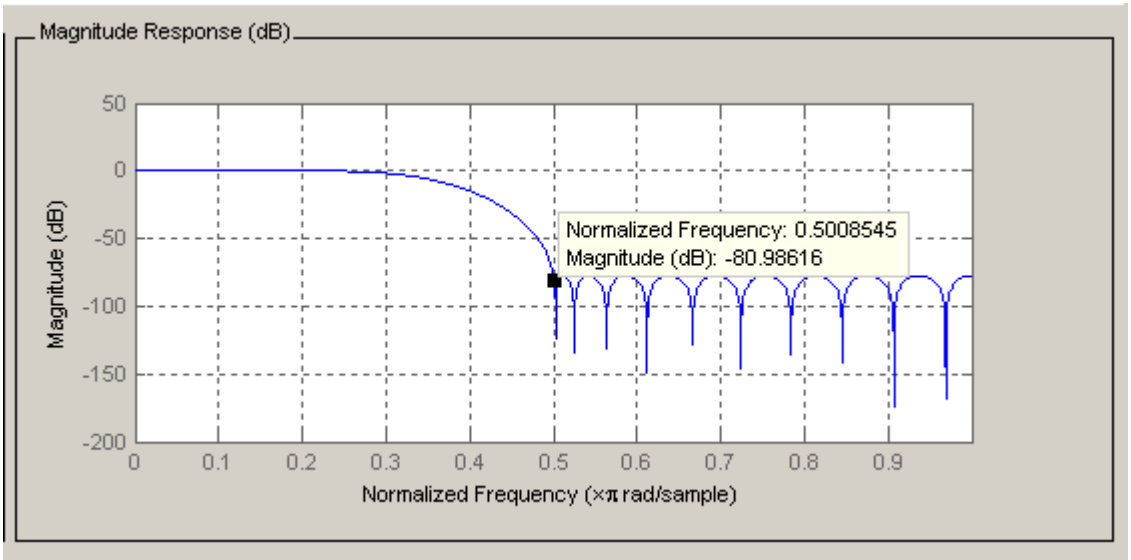
### Changing Axes Units

You can change the x- or y-axis units by right-clicking the mouse on an axis label and selecting the desired units. The current units have a check mark.




### Marking Data Points

In the Display region, you can click on any point in the plot to add a data marker, which displays the values at that point. Right-clicking on the data marker displays a menu where you can move, delete or adjust the appearance of the data markers.

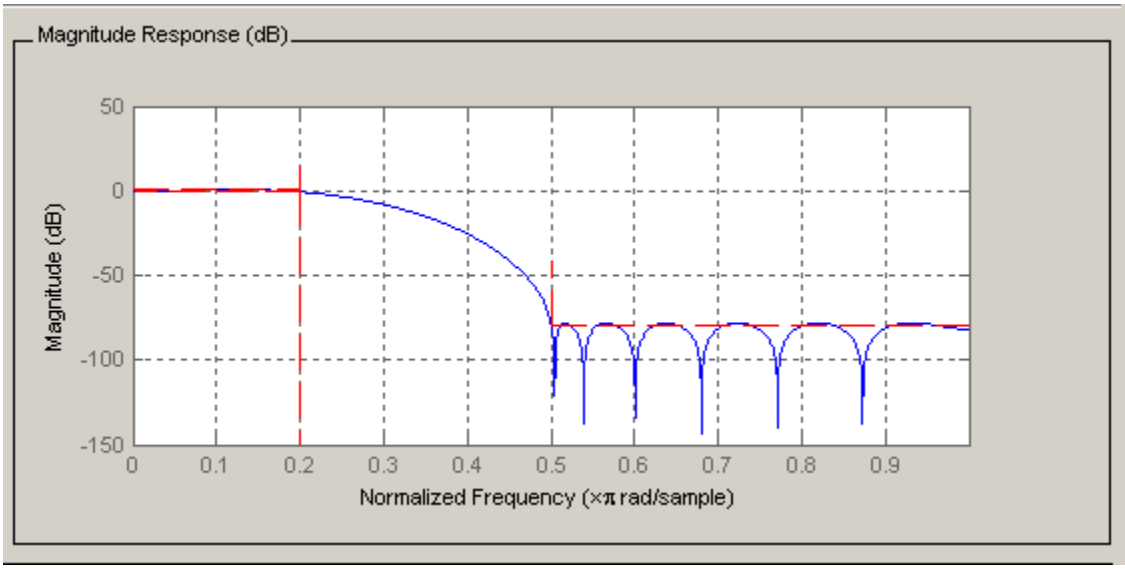



### Optimizing the Design

To minimize the cost of implementation of the filter, we will try to reduce the number of coefficients by using **Minimum Order** option in the design panel.

Change the selection in **Filter Order** to **Minimum Order** in the Design Region and leave the other parameters as they are.

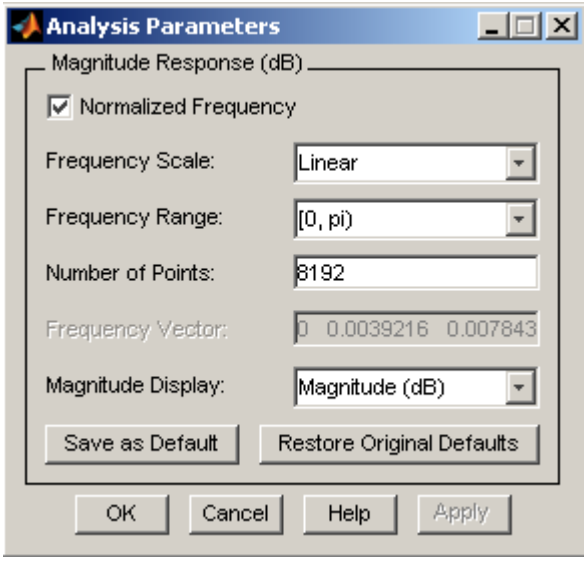
Click the **Design Filter** button to design the new filter.




As you can see in the Current Filter Information area, the filter order decreased from 30 to 16, the number of ripples decreased and the transition width became wider. The passband and the stopband specifications still meet the design criteria.

### Changing Analyses Parameters

By right-clicking on the plot and selecting Analysis Parameters, you can display a dialog box for changing analysis-specific parameters. (You can also select Analysis Parameters from the Analysis menu.)



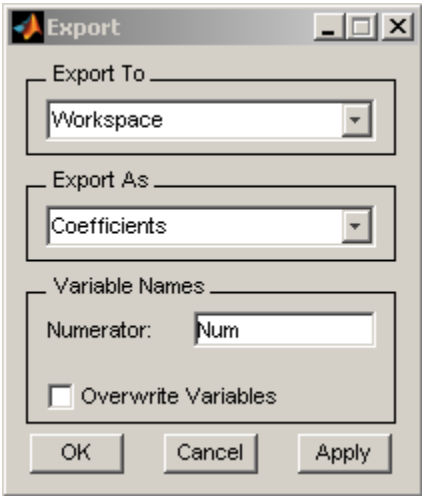

To save the displayed parameters as the default values, click **Save as Default**. To restore the MATLAB-defined default values, click **Restore Original Defaults**.

### Exporting the Filter

Once you are satisfied with your design, you can export your filter to the following destinations:

- MATLAB workspace
- MAT-file
- Text-file

Select **Export** from the **File** menu.




When you choose to export to the MATLAB workspace or to a MAT-file, you can export the filter as coefficients. If a DSP System Toolbox™ is available you can also export your filter as a System object.

### Generating a MATLAB File

Filter Designer allows you to generate MATLAB code to re-create your filter. This enables you to embed your design into existing code or automate the creation of your filters in a script.

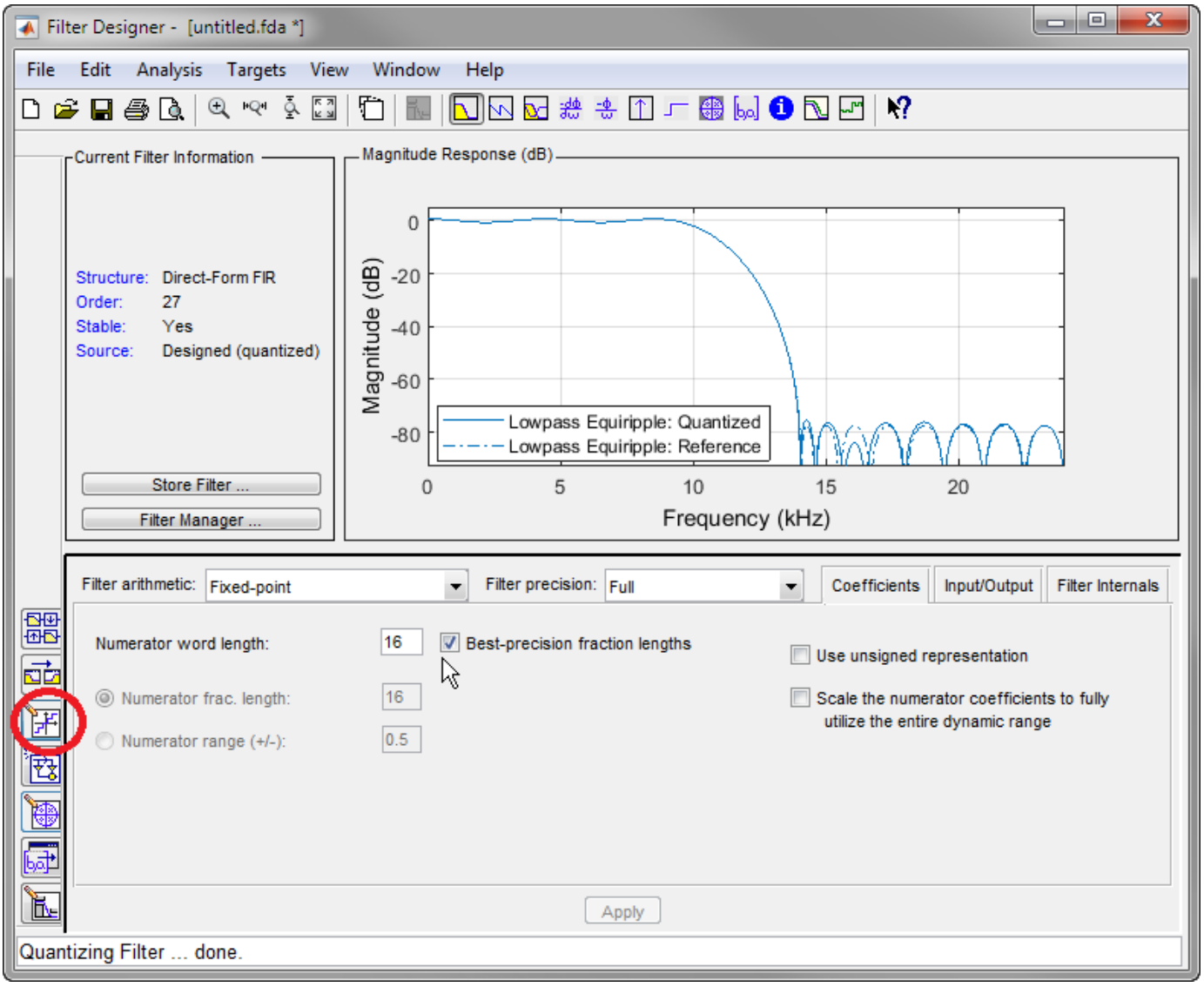
Select **Generate MATLAB code** from the **File** menu, choose **Filter Design Function** and specify the filename in the Generate MATLAB code dialog box.

The following code was generated from the minimum order filter we designed above:

```
function B = minorderlowfir
%MINORDERLOWFIR Returns discrete-time filter coefficients.

% MATLAB Code
% Generated by MATLAB(R) 8.1 and the Signal Processing Toolbox 6.19.
% Generated on: 11-Oct-2012 12:19:05

% Equiripple Lowpass filter designed using the FIRPM function.

% All frequency values are normalized to 1.

Fpass = 0.2;             % Passband Frequency
Fstop = 0.5;             % Stopband Frequency
Dpass = 0.057501127785;  % Passband Ripple
Dstop = 0.0001;          % Stopband Attenuation
dens  = 20;              % Density Factor

% Calculate the order from the parameters using FIRPMORD.
[N, Fo, Ao, W] = firpmord([Fpass, Fstop], [1 0], [Dpass, Dstop]);

% Calculate the coefficients using the FIRPM function.
B = firpm(N, Fo, Ao, W, {dens});
```

### Quantizing a Filter

If you have the DSP System Toolbox™ installed, the **Set quantization parameters** panel is available on the sidebar:




You can use this panel to quantize and analyze double-precision filters. With the DSP System Toolbox you can quantize from double-precision to single-precision. If you have the Fixed-Point Designer, you can quantize filters to fixed-point precision. Note that you cannot mix floating-point and fixed-point arithmetic in your filter.

### Targets

The **Targets** menu of Filter Designer allows you to generate various types of code representing your filter. For example, you can generate C header files, XILINX coefficients(COE) files (with the DSP System Toolbox) and VHDL, Verilog along with test benches (with Filter Design HDL Coder™).

### Additional Features

Filter Designer also integrates additional functionality from these other MathWorks™ products:

- **DSP System Toolbox**- Adds advanced FIR and IIR design techniques (i.e. Filter transformations, Multirate filters) and generates equivalent block for the filter

- **Embedded Coder™**- Generates, builds and deploys code for Texas Instruments C6000 processors.
- **Filter Design HDL Coder**- Generates synthesizable VHDL or Verilog code for fixed-point filters
- **Simulink®**- Generates filters from atomic Simulink blocks

## See Also

**Filter Designer**

# Filter Analysis using FVTool

This example shows how to use several filter analysis functions in a single figure window by using the Filter Visualization Tool (FVTool), a Graphical User Interface available in the Signal Processing Toolbox™.

FVTool also has an Application Program Interface (API) that allows you to interact with the GUI from the command line. This enables you to integrate FVTool into other applications.

### Launching FVTool

We want to create a lowpass filter with a passband frequency of 0.4π rad/sample, a stopband frequency of 0.6π rad/sample, a passband ripple of 1 dB and a stopband attenuation of 80 dB. We will design the filters using some of the Signal Processing Toolbox's filter design tools and then analyze the results in FVTool.

Design a lowpass equiripple FIR filter

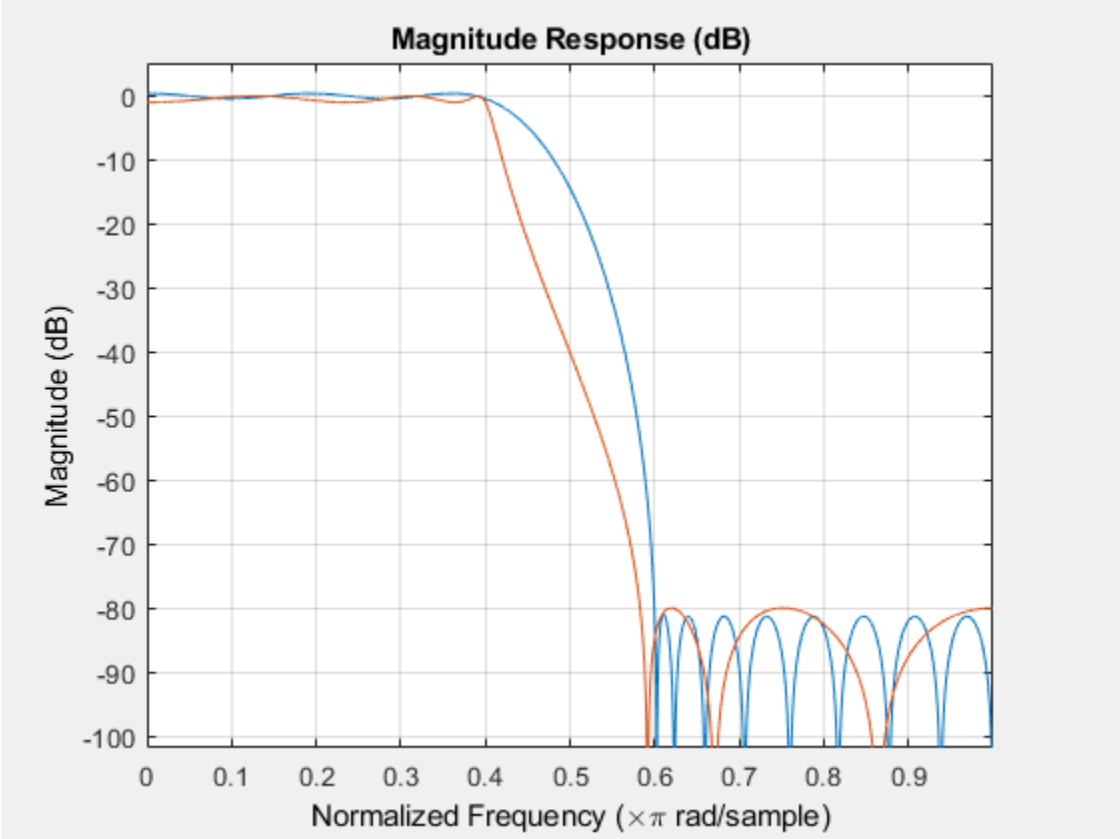
```
Df1 = designfilt('lowpassfir','PassbandFrequency',0.4,...
                              'StopbandFrequency',0.6,...
                              'PassbandRipple',1,...
                              'StopbandAttenuation',80,...
                              'DesignMethod','equiripple');
```

Design a lowpass elliptic IIR filter

```
Df2 = designfilt('lowpassiir','PassbandFrequency',0.4,...
                              'StopbandFrequency',0.6,...
                              'PassbandRipple',1,...
                              'StopbandAttenuation',80,...
                              'DesignMethod','ellip');
```

Launch FVTool with the filter objects and return a handle to FVTool which enables us to reuse the same FVTool figure.

```
hfvt = fvtool(Df1, Df2);
```

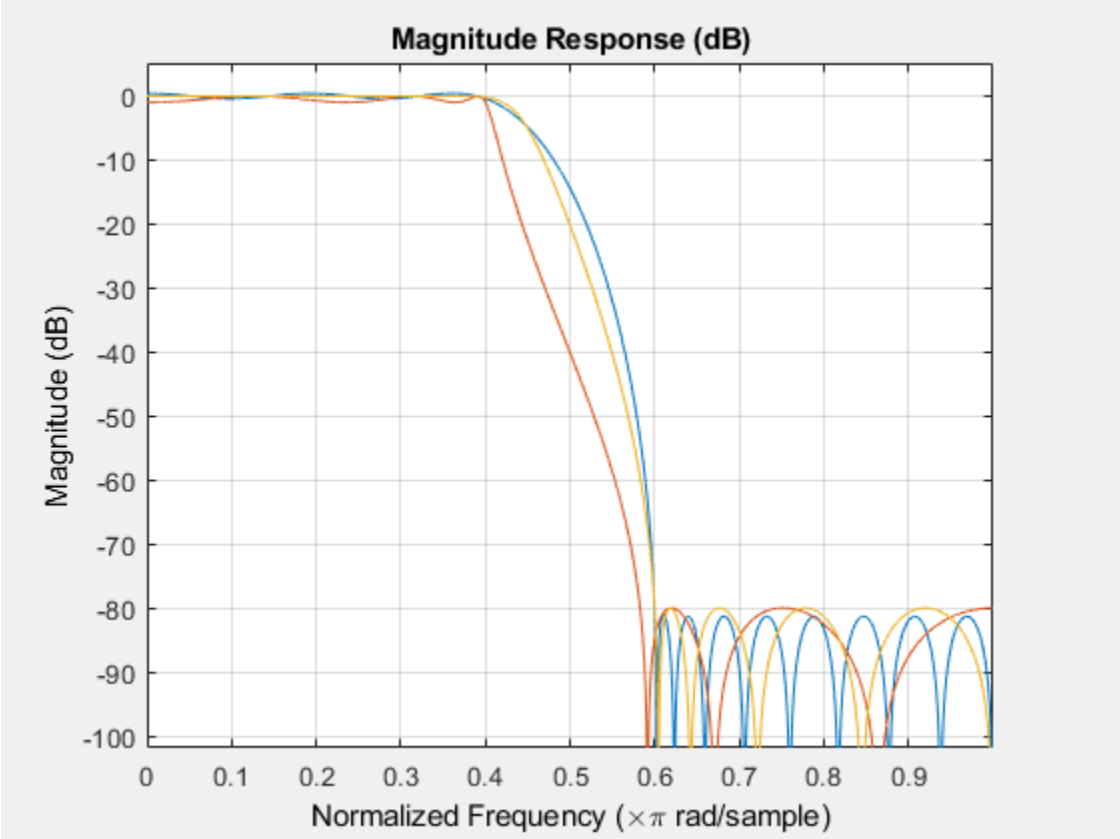



### Adding and Removing Filters

We can observe that both filters meet the design specifications, but we also want to see how well the Chebyshev Type II design performs.

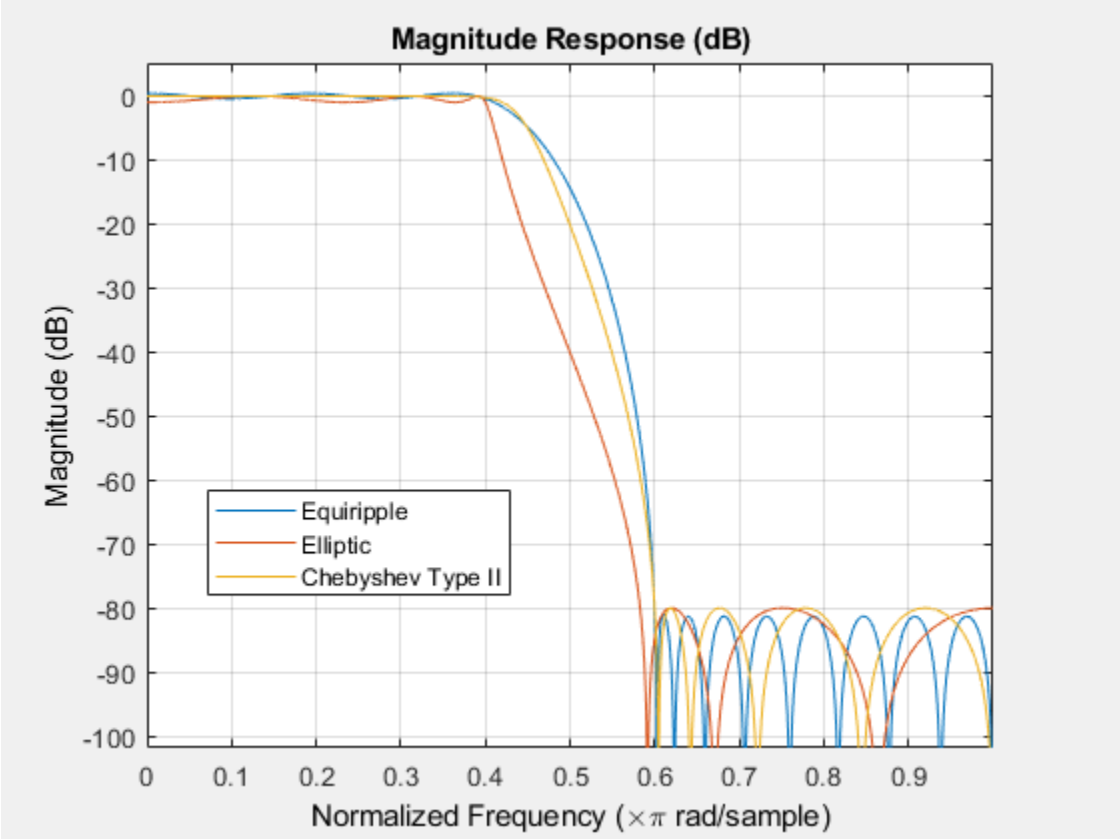
You can add a filter to FVTool using the ADDFILTER function.

```
Df3 = designfilt('lowpassiir','PassbandFrequency',0.4,...
                              'StopbandFrequency',0.6,...
                              'PassbandRipple',1,...
                              'StopbandAttenuation',80,...
                              'DesignMethod','cheby2');
addfilter(hfvt, Df3);
```



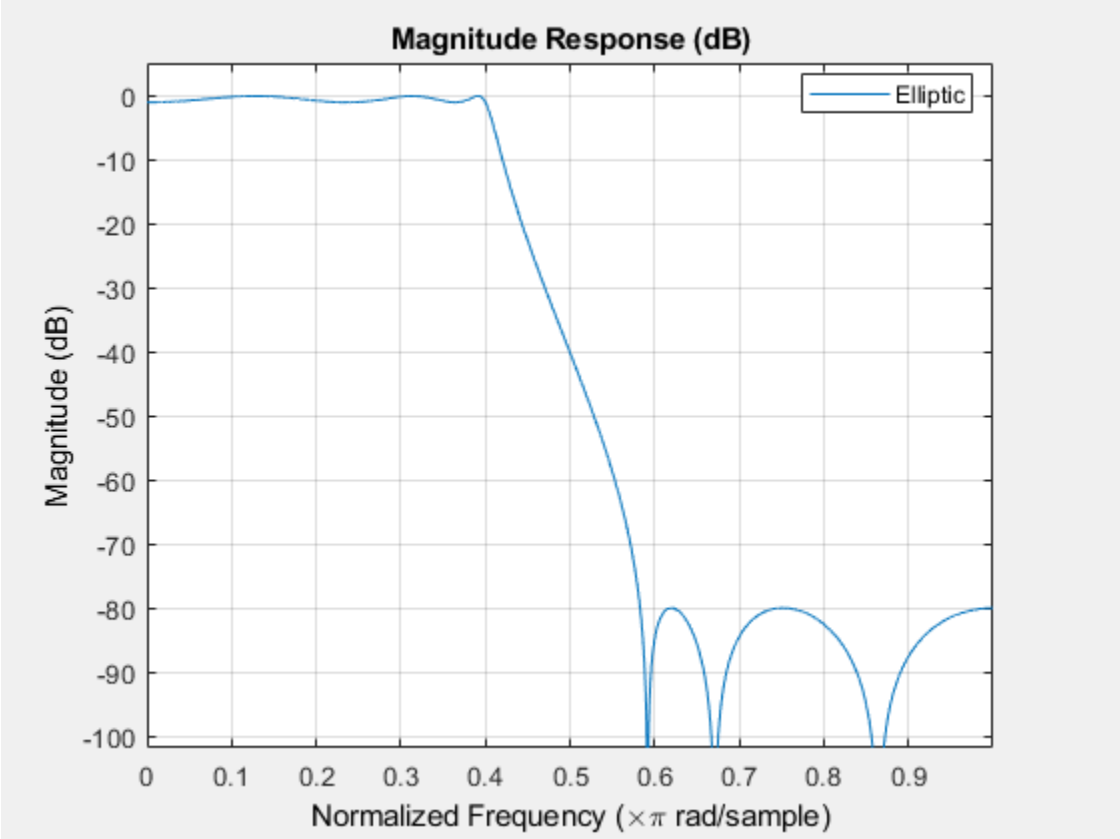

To identify which line on the plot belongs to which filter, you can add a legend using the LEGEND function of the FVTool handle.

```
legend(hfvt, 'Equiripple', 'Elliptic', 'Chebyshev Type II');
```




You can remove a filter from FVTool using the DELETEFILTER function and passing the index of the filter(s) that you want to remove.

```
deletefilter(hfvt, [1 3]);
```

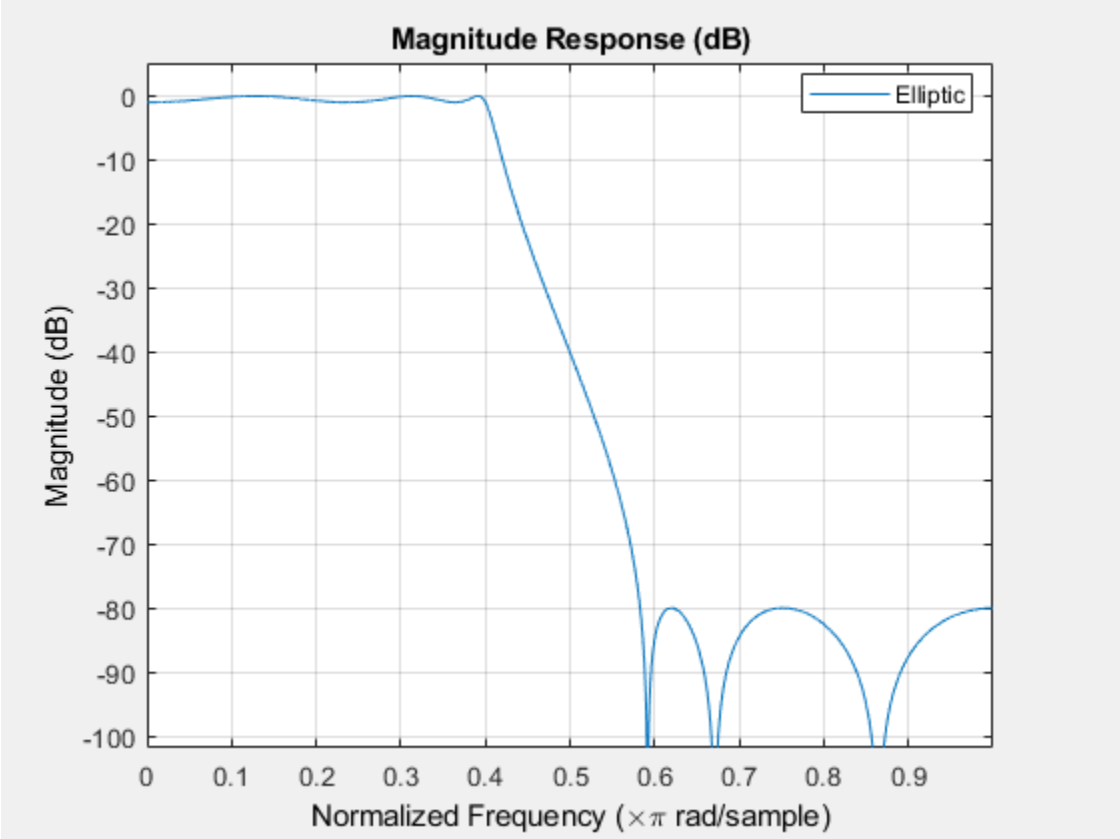



### Changing the Analysis Parameters

The handle that FVTool returns contains properties that allow you to interact with both the filter and the current analysis.

To see all of the available properties you can use the GET command. The first properties are those of a regular MATLAB® figure. The last fourteen properties are FVTool specific. The last six of these (from FrequencyScale to MagnitudeDisplay) are analysis specific.

```
s = get(hfvt);
```




```
% Keep the last 14 properties
c = struct2cell(s);
f = fieldnames(s);
s = cell2struct(c(end-14:end),f(end-14:end),1)
```

```
s = struct with fields:
       SelectionHighlight: on
                      Tag: 'filtervisualizationtool'
                 UserData: []
                  Visible: on
        OverlayedAnalysis: ''
            ShowReference: 'on'
           FrequencyScale: 'Linear'
                 Analysis: 'magnitude'
           FrequencyRange: '[0, pi)'
         MagnitudeDisplay: 'Magnitude (dB)'
      NormalizedFrequency: 'on'
            PolyphaseView: 'off'
    NormalizeMagnitudeto1: 'off'
          FrequencyVector: [1x256 double]
           NumberofPoints: 8192
```

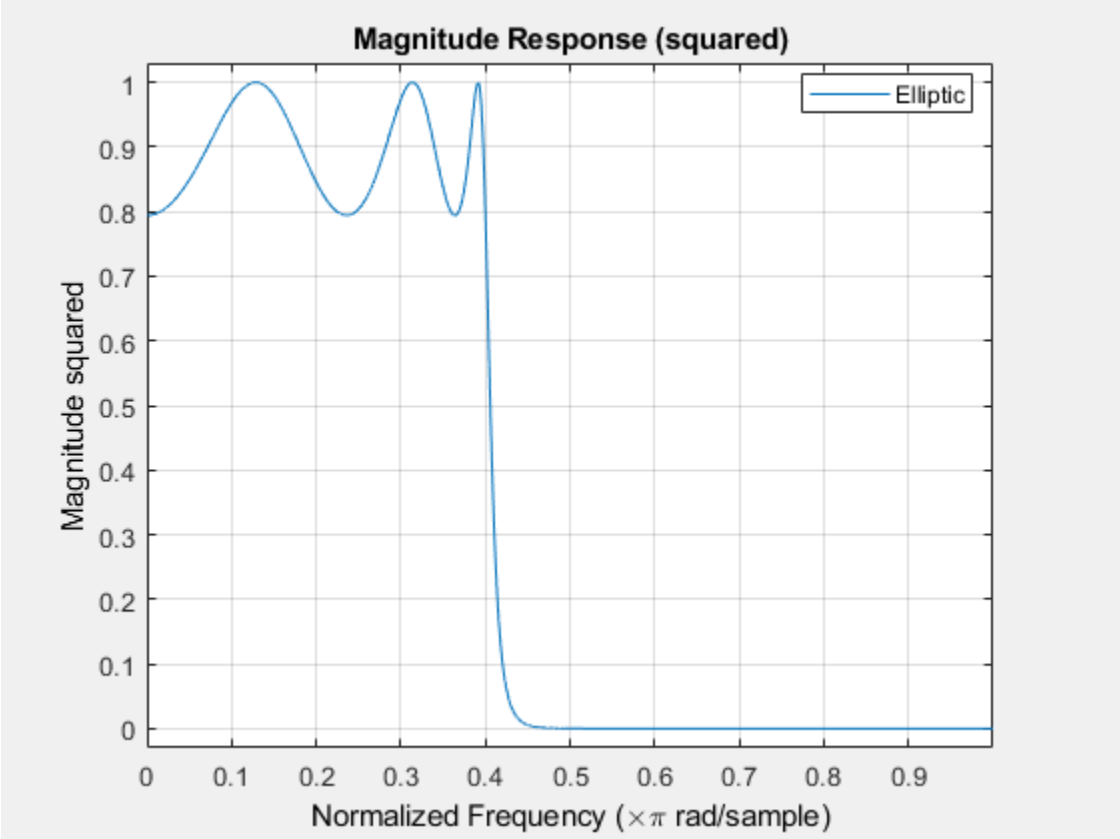
All the parameters that are available from the FVTool's Analysis Parameters dialog are also available as properties of the FVTool object. The SET command with only two input arguments returns all possible values.

```
set(hfvt, 'MagnitudeDisplay')
```

```
ans = 1x4 cell
  Columns 1 through 3

    {'Magnitude'}    {'Magnitude (dB)'}    {'Magnitude squared'}

  Column 4

    {'Zero-phase'}
```

Turn the display to 'Magnitude Squared'

```
hfvt.MagnitudeDisplay = 'Magnitude Squared';
```




Get all possible values for the 'Analysis' property

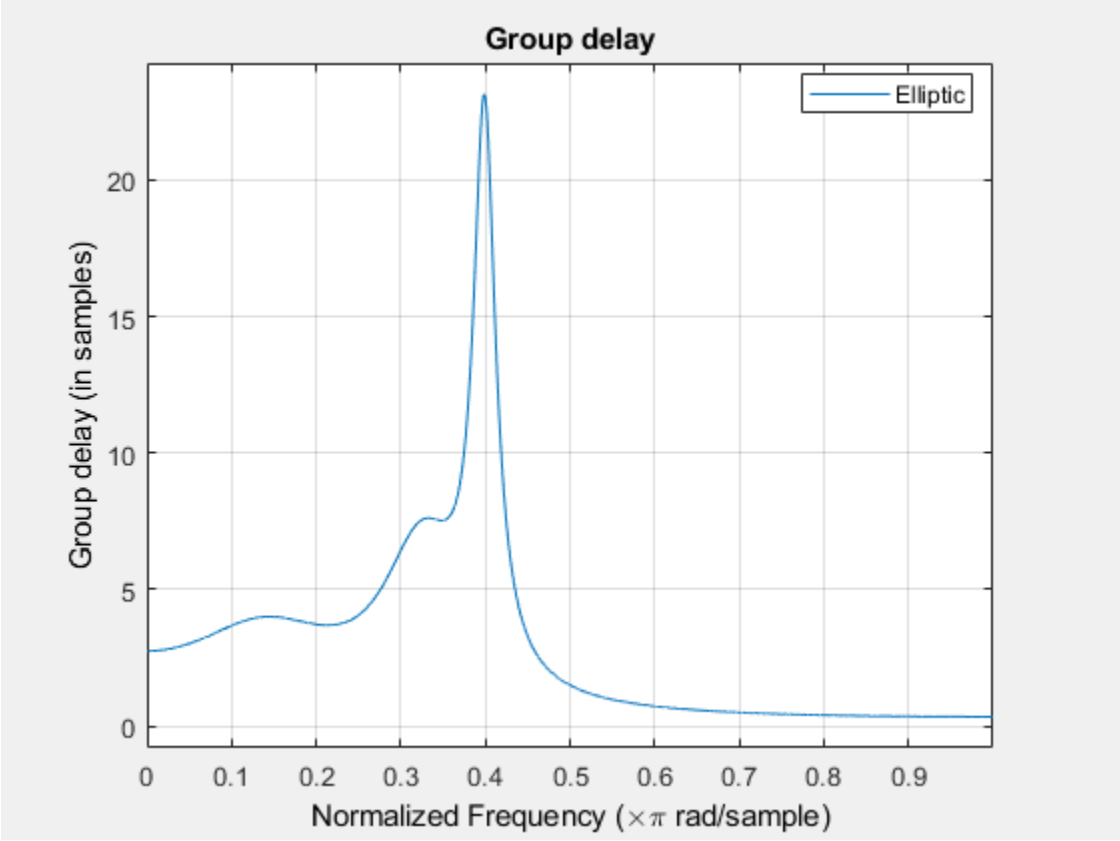
```
set(hfvt, 'Analysis')
```

```
ans = 1x12 cell
  Columns 1 through 5

    {'magnitude'}    {'phase'}    {'freq'}    {'grpdelay'}    {'phasedelay'}

  Columns 6 through 10

    {'impulse'}    {'step'}    {'polezero'}    {'coefficients'}    {'info'}

  Columns 11 through 12
```

```
    {'magestimate'}    {'noisepower'}
```

Let us now change the analysis to look at the Group Delay Response of the filter.

```
hfvt.Analysis = 'grpdelay';
```




The GET command will return new Analysis Parameters for the new analysis.

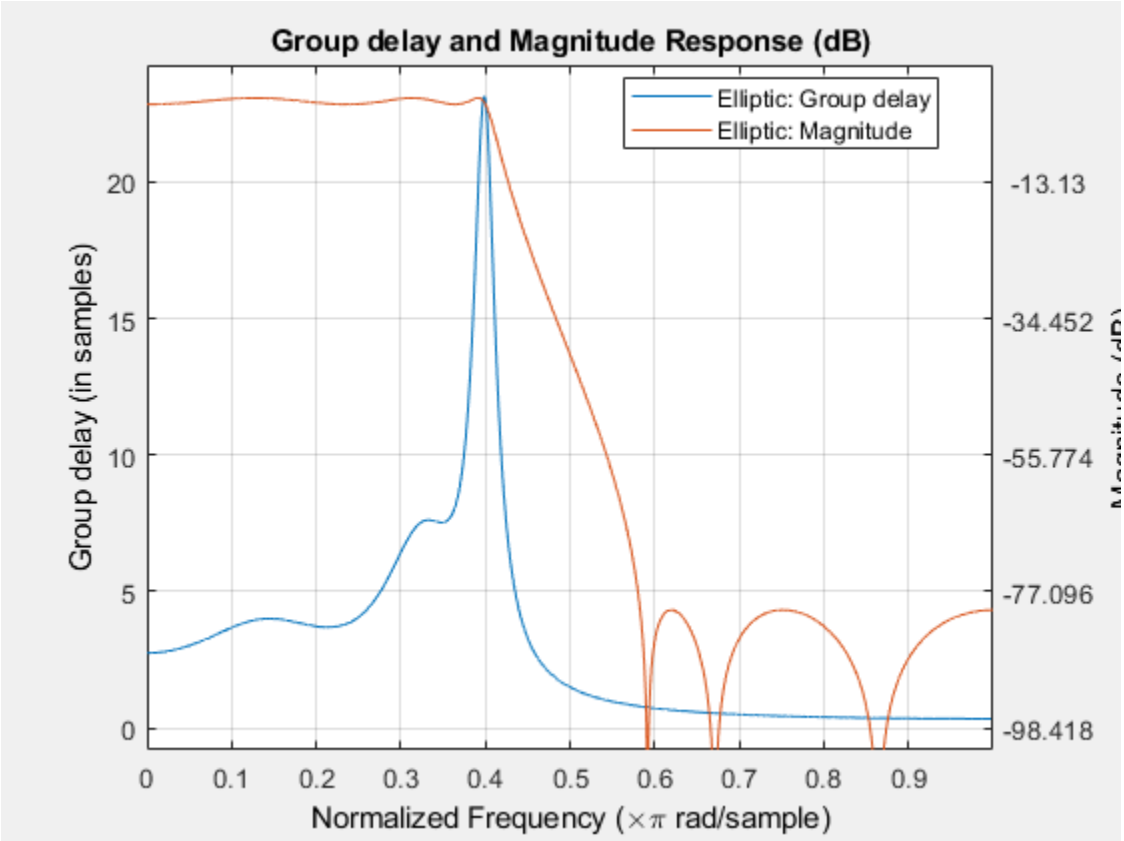
```
GroupDelayUnits = hfvt.GroupDelayUnits;
```

### Overlaying Two Analyses

We would also like to see how the Group Delay and the magnitude response overlap in the frequency domain.

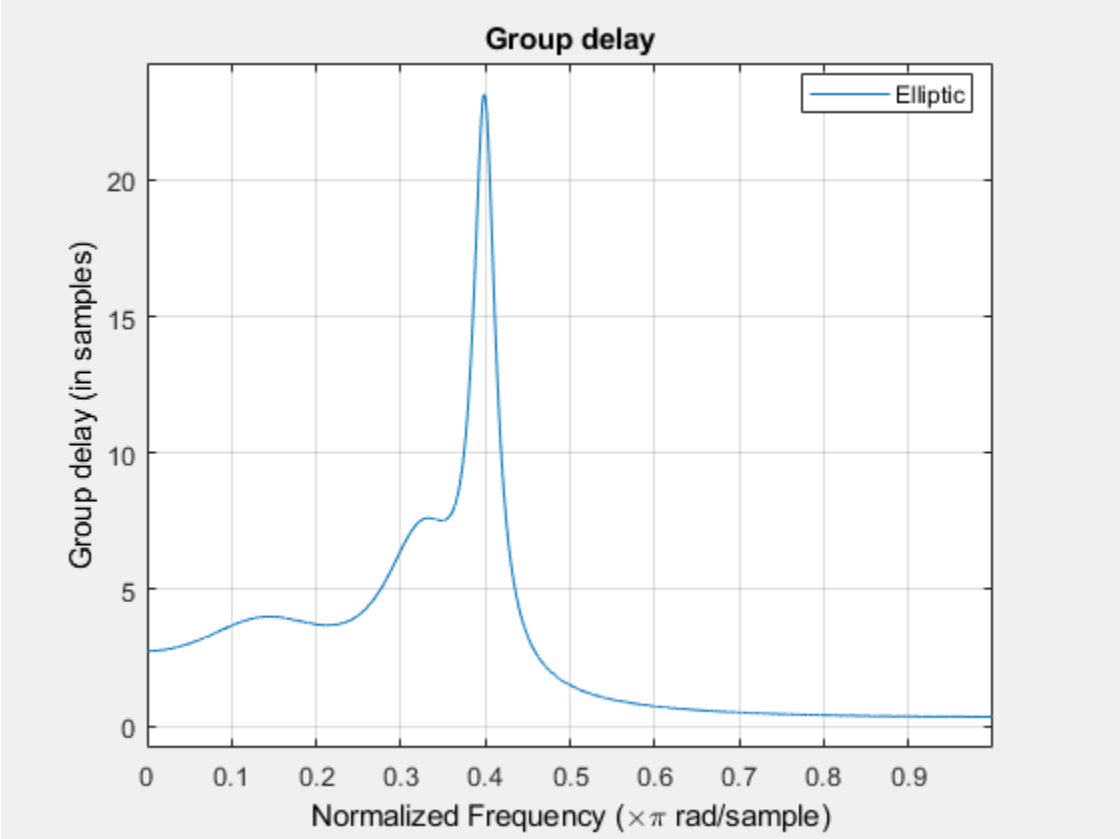
You can overlay any two analyses in FVTool that share a common x-axis (time or frequency) by setting the 'OverlayedAnalysis' property.

```
set(hfvt, 'OverlayedAnalysis', 'magnitude', 'Legend', 'On')
```




To turn off the overlayed analysis simply set the 'OverlayedAnalysis' property to ''.

```
hfvt.OverlayedAnalysis = '';
```

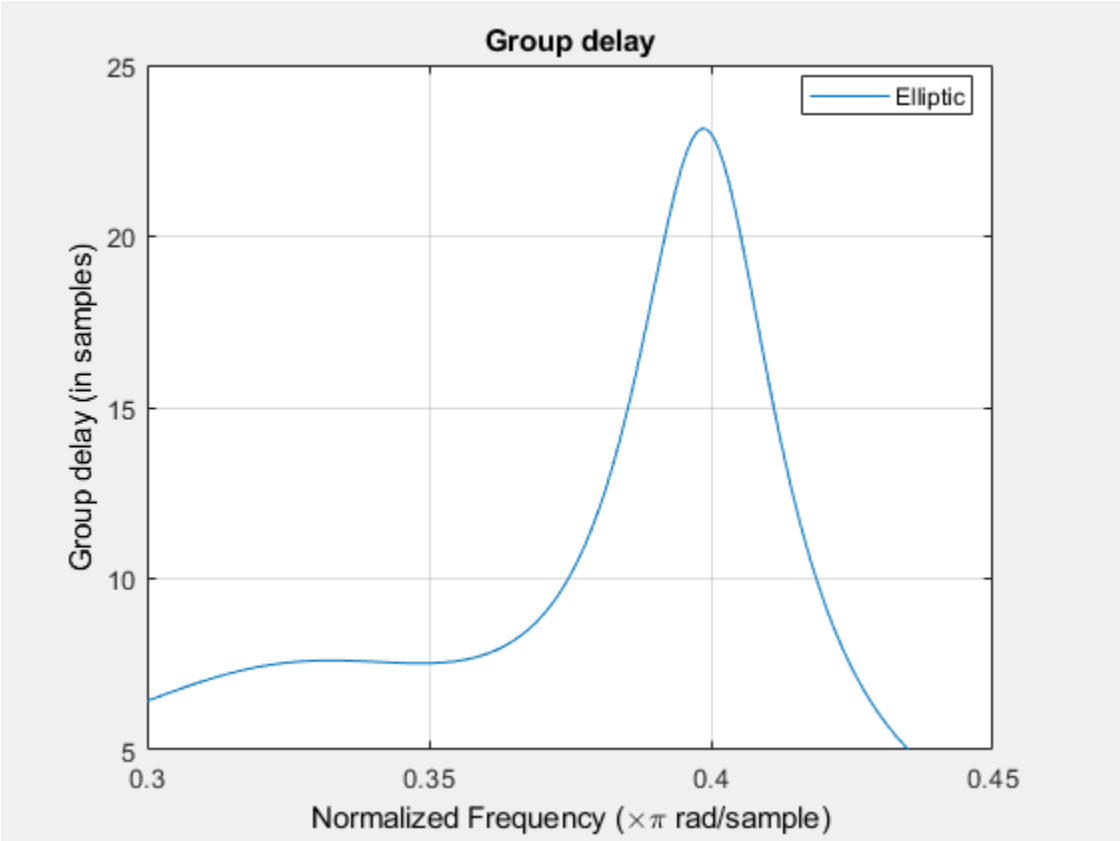



### Interacting with FVTool like a Figure Window

The FVTool window can also be annotated like a normal figure window.

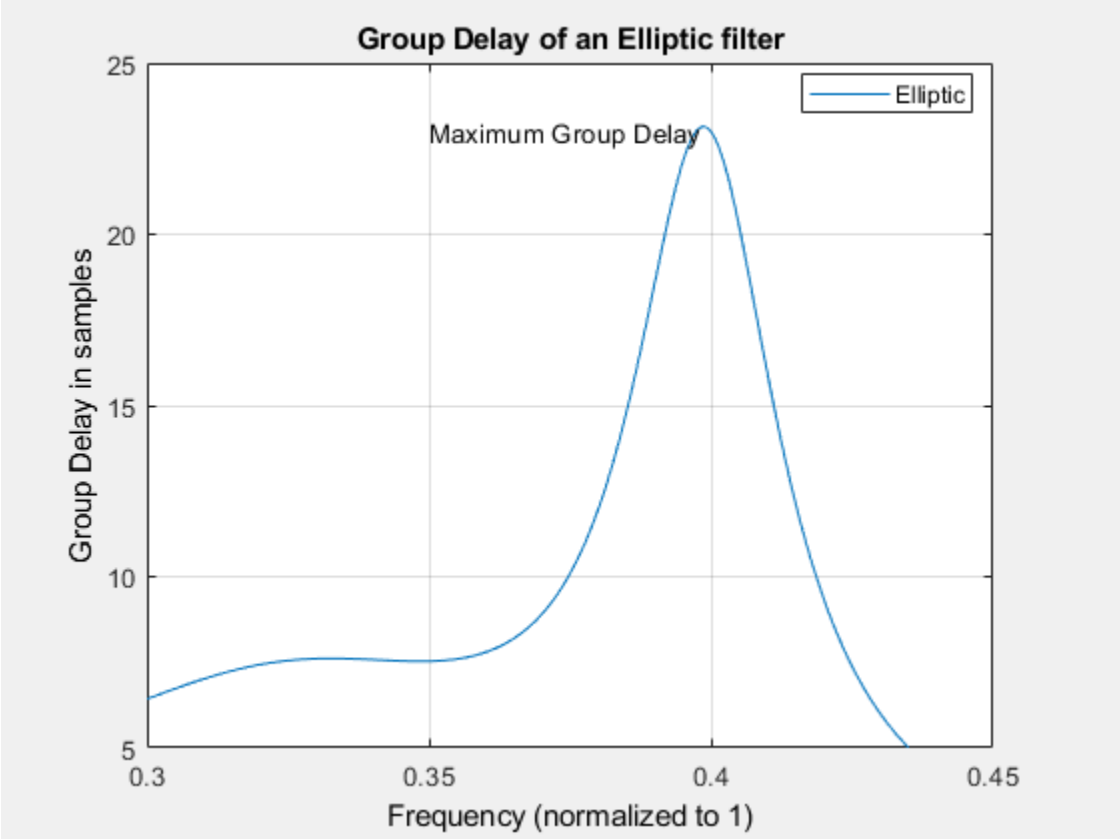
The FVTool figure behaves as a normal figure window. This allows you to use MATLAB's grid and axis functions.

```
grid on
axis([.3 .45 5 25]);
```




The axis is also accessible from the command line. This allows you to change the title and labels.

```
title('Group Delay of an Elliptic filter');
xlabel('Frequency (normalized to 1)');
ylabel('Group Delay in samples');

htext = text(.35, 23, 'Maximum Group Delay');
```




FVTool will not automatically delete additional annotations from your analysis, you can do this by deleting the handle itself. You can close the FVTool figure by calling the close function on the FVTool handle.

```
delete(htext);
close(hfvt)
```

## See Also

**FVTool** | `designfilt`

# FIR Gaussian Pulse-shaping Filter Design

This example shows how to design a Gaussian pulse-shaping FIR filter and the parameters influencing this design. The FIR Gaussian pulse-shaping filter design is done by truncating a sampled version of the continuous-time impulse response of the Gaussian filter which is given by:

$$h(t) = \frac{\sqrt{\pi}}{a} e^{-\frac{\pi^2 t^2}{a^2}}$$

The parameter 'a' is related to 3-dB bandwidth-symbol time product (B*Ts) of the Gaussian filter as given by:

$$a = \frac{1}{BT_s}\sqrt{\frac{\log 2}{2}}$$

There are two approximation errors in this design: a truncation error and a sampling error. The truncation error is due to a finite-time (FIR) approximation of the theoretically infinite impulse response of the ideal Gaussian filter. The sampling error (aliasing) is due to the fact that a Gaussian frequency response is not really band-limited in a strict sense (i.e. the energy of the Gaussian signal beyond a certain frequency is not exactly zero). This can be noted from the transfer function of the continuous-time Gaussian filter, which is given as below:

$$H(f) = e^{-a^2 f^2}$$

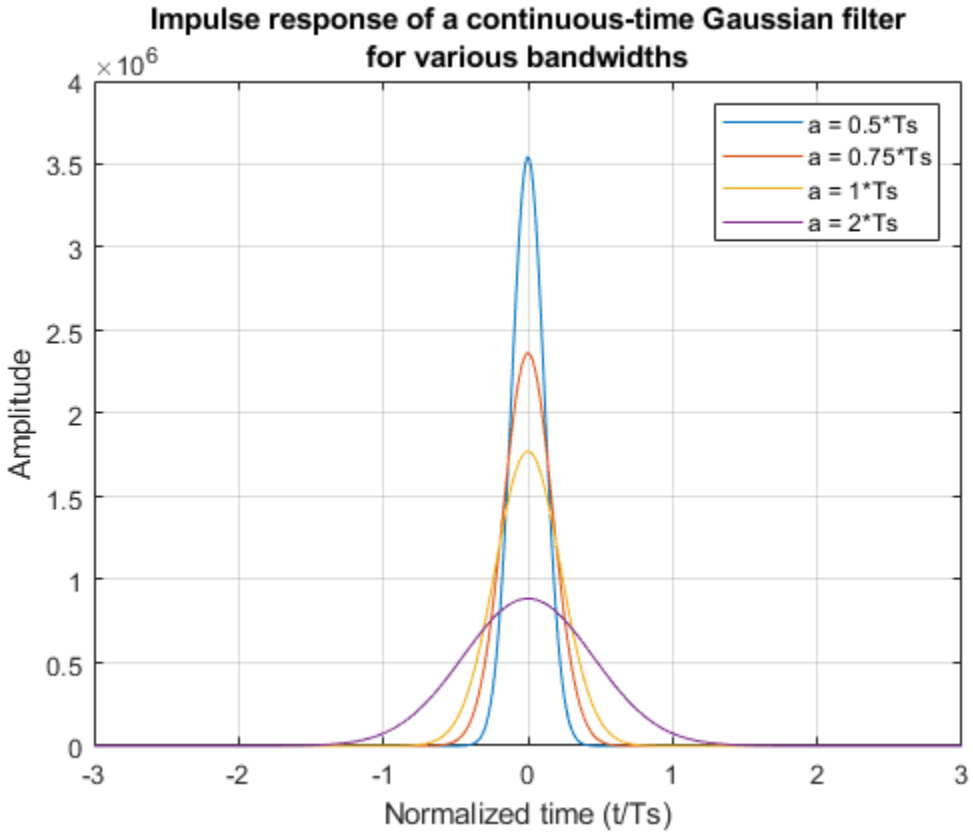
As f increases, the frequency response tends to zero, but never is exactly zero, which means that it cannot be sampled without some aliasing occurring.

### Continuous-time Gaussian Filter

To design a continuous-time Gaussian filter, let us define the symbol time (Ts) to be 1 micro-second and the number of symbols between the start of the impulse response and its end (filter span) to be 6. From the equations above, we can see that the impulse response and the frequency response of the Gaussian filter depend on the parameter 'a' which is related to the 3 dB bandwidth-symbol time product. To study the effect of this parameter on the Gaussian FIR filter design, we will define various values of 'a' in terms of Ts and compute the corresponding bandwidths. Then, we will plot the impulse response for each 'a' and the magnitude response for each bandwidth.

```
Ts   = 1e-6; % Symbol time (sec)
span = 6;    % Filter span in symbols

a = Ts*[.5, .75, 1, 2];
B = sqrt(log(2)/2)./(a);

t = linspace(-span*Ts/2,span*Ts/2,1000)';
hg = zeros(length(t),length(a));
for k = 1:length(a)
  hg(:,k) = sqrt(pi)/a(k)*exp(-(pi*t/a(k)).^2);
end

plot(t/Ts,hg)
title({'Impulse response of a continuous-time Gaussian filter';...
  'for various bandwidths'});
xlabel('Normalized time (t/Ts)')
ylabel('Amplitude')
```

```
legend(sprintf('a = %g*Ts',a(1)/Ts),sprintf('a = %g*Ts',a(2)/Ts),...
  sprintf('a = %g*Ts',a(3)/Ts),sprintf('a = %g*Ts',a(4)/Ts))
grid on;
```



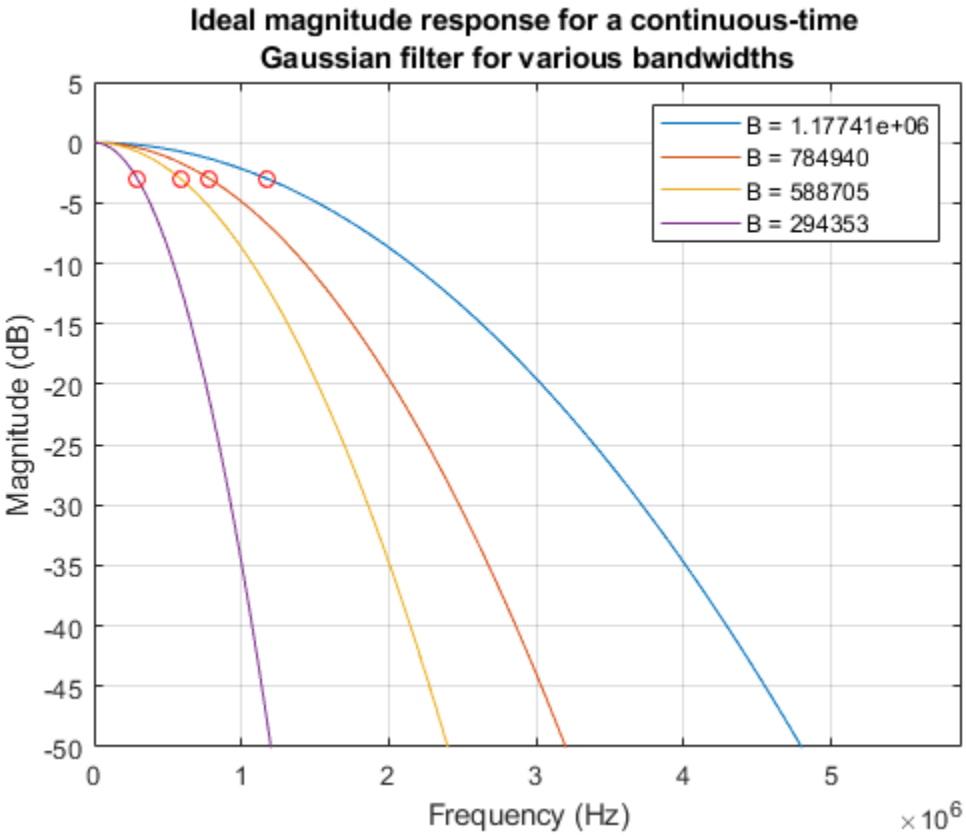

Note that the impulse responses are normalized to the symbol time.

### Frequency Response for Continuous-time Gaussian Filter

We will compute and plot the frequency response for continuous-time Gaussian filters with different bandwidths. In the graph below, the 3-dB cutoff is indicated by the red circles ('o') on the magnitude response curve. Note that 3-dB bandwidth is between DC and B.

```
f = linspace(0,32e6,10000)';
Hideal = zeros(length(f),length(a));
for k = 1:length(a)
  Hideal(:,k) = exp(-a(k)^2*f.^2);
end

plot(f,20*log10(Hideal))
titleStr = {'Ideal magnitude response for a continuous-time ';...
  'Gaussian filter for various bandwidths'};
title(titleStr);
legend(sprintf('B = %g',B(1)),sprintf('B = %g',B(2)),...
  sprintf('B = %g',B(3)),sprintf('B = %g',B(4)))
hold on
for k = 1:length(a)
  plot(B,20*log10(exp(-a.^2.*B.^2)),'ro','HandleVisibility','off')
end
```

```
axis([0 5*max(B) -50 5])
xlabel('Frequency (Hz)')
ylabel('Magnitude (dB)')
grid on;
```




**FIR approximation of the Gaussian Filter**

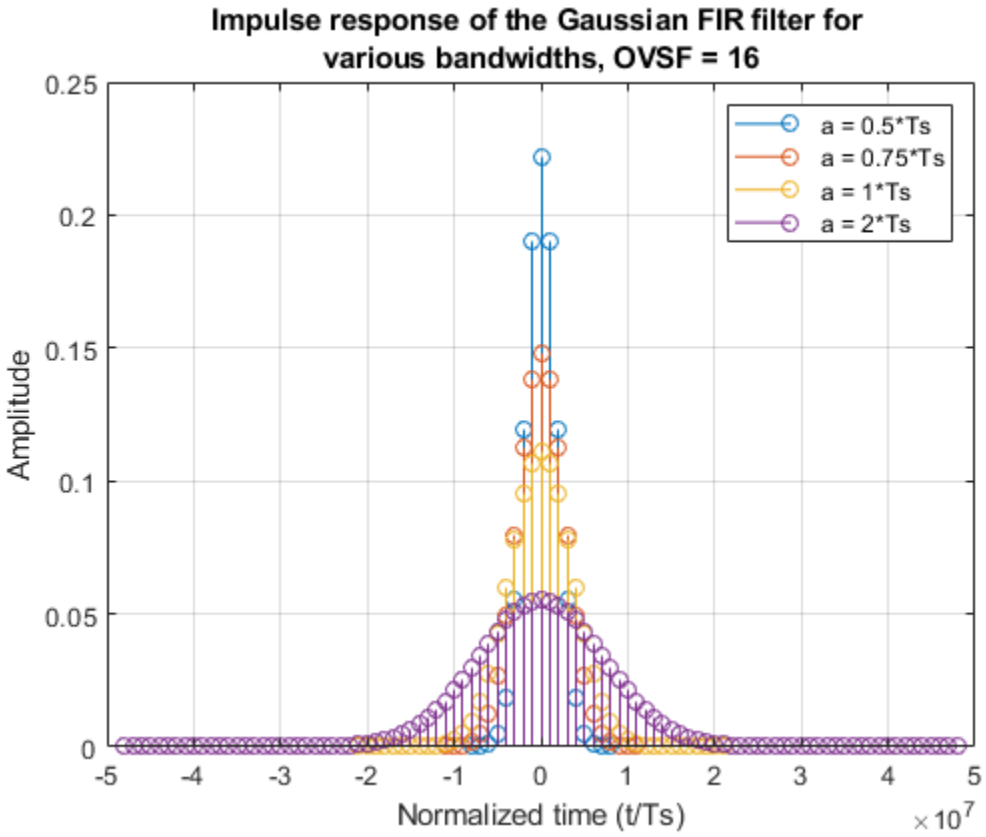
We will design the FIR Gaussian filter using the **gaussdesign** function. The inputs to this function are the 3-dB bandwidth-symbol time product, the number of symbol periods between the start and end of the filter impulse response, i.e. filter span in symbols, and the oversampling factor (i.e. the number of samples per symbol).

The oversampling factor (OVSF) determines the sampling frequency and the filter length and hence, plays a significant role in the Gaussian FIR filter design. The approximation errors in the design can be reduced with an appropriate choice of oversampling factor. We illustrate this by comparing the Gaussian FIR filters designed with two different oversampling factors.

First, we will consider an oversampling factor of 16 to design the discrete Gaussian filter.

```
ovsf = 16; % Oversampling factor (samples/symbol)
h = zeros(97,4);
iz = zeros(97,4);
for k = 1:length(a)
  BT = B(k)*Ts;
  h(:,k) = gaussdesign(BT,span,ovsf);
  [iz(:,k),t] = impz(h(:,k));
```

```
end
figure('Color','white')
t = (t-t(end)/2)/Ts;
stem(t,iz)
title({'Impulse response of the Gaussian FIR filter for ';...
  'various bandwidths, OVSF = 16'});
xlabel('Normalized time (t/Ts)')
ylabel('Amplitude')
legend(sprintf('a = %g*Ts',a(1)/Ts),sprintf('a = %g*Ts',a(2)/Ts),...
  sprintf('a = %g*Ts',a(3)/Ts),sprintf('a = %g*Ts',a(4)/Ts))
grid on;
```

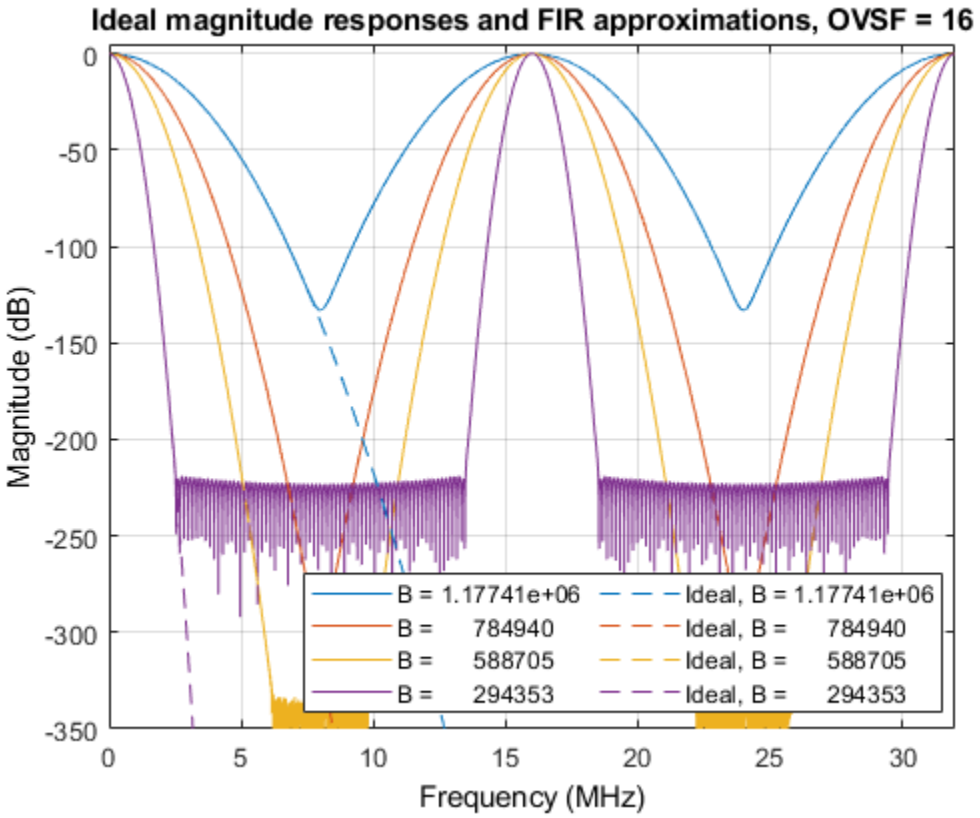



**Frequency Response for FIR Gaussian Filter (oversampling factor=16)**

We will calculate the frequency response for the Gaussian FIR filter with an oversampling factor of 16 and we will compare it with the ideal frequency response (i.e. frequency response of a continuous-time Gaussian filter).

```
Fs = ovsf/Ts;
fvtool(h(:,1),1,h(:,2),1,h(:,3),1,h(:,4),1,...
  'FrequencyRange', 'Specify freq. vector', ...
  'FrequencyVector',f,'Fs',Fs,'Color','white');
title('Ideal magnitude responses and FIR approximations, OVSF = 16')
hold on
plot(f*Ts,20*log10(Hideal),'--')
hold off
axis([0 32 -350 5])
```

```
legend(append(["B = " "Ideal, B = "],string(num2str(B','%g'))), ...
   'NumColumns',2,'Location','best')
```




Notice that the first two FIR filters exhibit aliasing errors and the last two FIR filters exhibit truncation errors. Aliasing occurs when the sampling frequency is not greater than the Nyquist frequency. In case of the first two filters, the bandwidth is large enough that the oversampling factor does not separate the spectral replicas enough to avoid aliasing. The amount of aliasing is not very significant however.

On the other hand, the last two FIR filters show the FIR approximation limitation before any aliasing can occur. The magnitude responses of these two filters reach a floor before they can overlap with the spectral replicas.

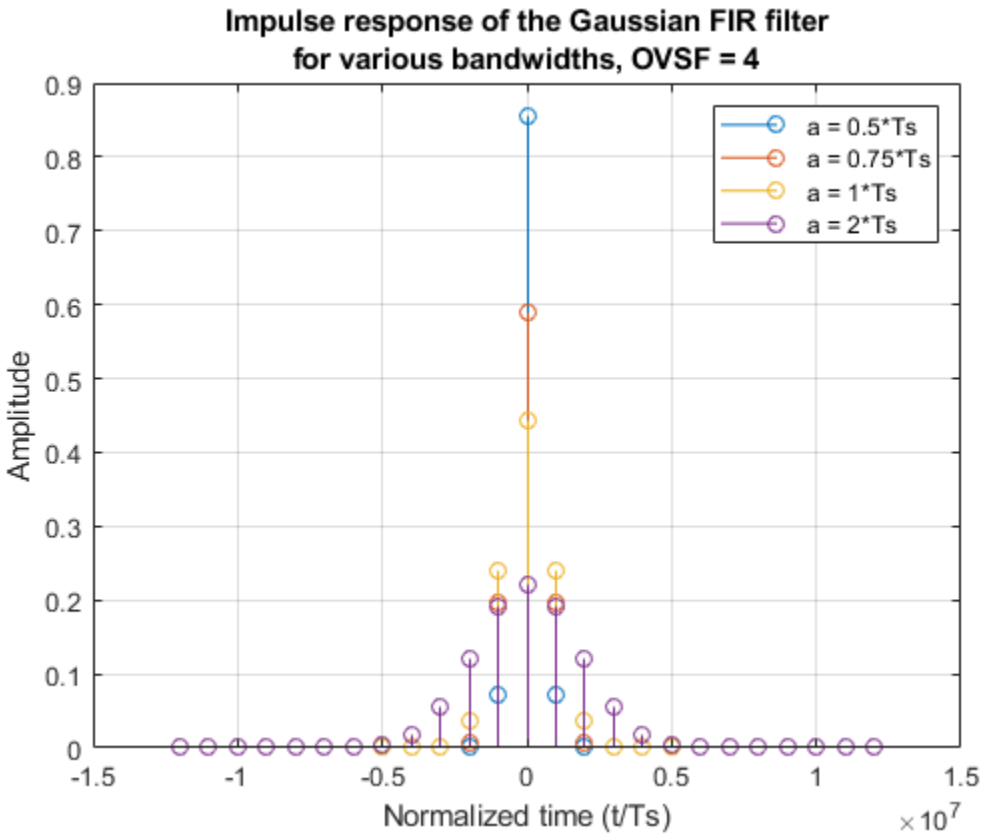
**Significance of the Oversampling Factor**

The aliasing and truncation errors vary according to the oversampling factor. If the oversampling factor is reduced, these errors will be more severe, since this reduces the sampling frequency (thereby moving the replicas closer) and also reduces the filter lengths (increasing the error in the FIR approximation).

For example, if we select an oversampling factor of 4, we will see that all the FIR filters exhibit aliasing errors as the sampling frequency is not large enough to avoid the overlapping of the spectral replicas.

```
ovsf = 4; % Oversampling factor (samples/symbol)
h = zeros(25,4);
```

```
iz = zeros(25,4);
for k = 1:length(a)
  BT = B(k)*Ts;
  h(:,k) = gaussdesign(BT,span,ovsf);
  [iz(:,k),t] = impz(h(:,k));
end
figure('Color','white')
t = (t-t(end)/2)/Ts;
stem(t,iz)
title({'Impulse response of the Gaussian FIR filter'; 'for various bandwidths, OVSF = 4'});
xlabel('Normalized time (t/Ts)')
ylabel('Amplitude')
legend(sprintf('a = %g*Ts',a(1)/Ts),sprintf('a = %g*Ts',a(2)/Ts),...
  sprintf('a = %g*Ts',a(3)/Ts),sprintf('a = %g*Ts',a(4)/Ts))
grid on;
```

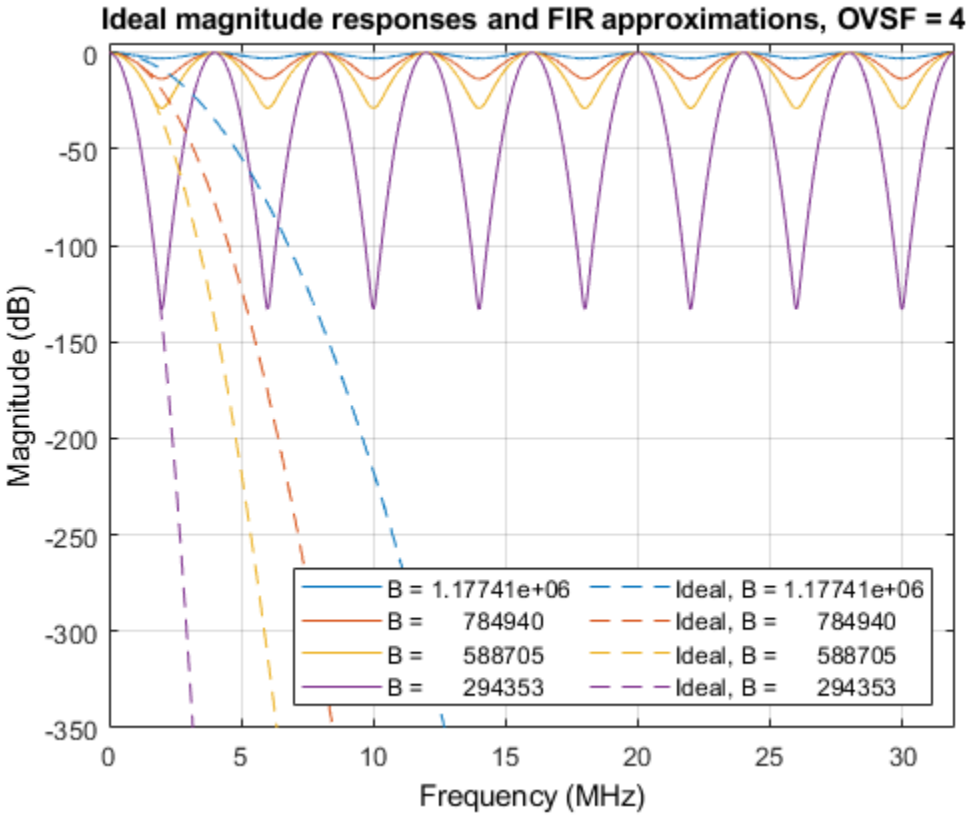



**Frequency Response for FIR Gaussian Filter (oversampling factor=4)**

We will plot and study the frequency response for the Gaussian FIR filter designed with oversampling factor of 4. A smaller oversampling factor means smaller sampling frequency. As a result, this sampling frequency is not enough to avoid the spectral overlap and all the FIR approximation filters exhibit aliasing.

```
Fs = ovsf/Ts;
fvtool(h(:,1),1,h(:,2),1,h(:,3),1,h(:,4),1,...
  'FrequencyRange', 'Specify freq. vector', ...
  'FrequencyVector',f,'Fs',Fs,'Color','white');
```

```
title('Ideal magnitude responses and FIR approximations, OVSF = 4')
hold on
plot(f*Ts,20*log10(Hideal),'--')
hold off
axis([0 32 -350 5])
legend(append(["B = " "Ideal, B = "],string(num2str(B','%g'))), ...
   'NumColumns',2,'Location','southeast')
```




## See Also

**FVTool** | `gaussdesign`

# Generating Guitar Chords Using the Karplus-Strong Algorithm

This example shows how to generate realistic guitar chords using the Karplus-Strong Algorithm and discrete-time filters.

### Setup

Begin by defining variables that we will be using later, e.g. the sampling frequency, the first harmonic frequency of the A string, the offset of each string relative to the A string.

```
Fs       = 44100;
A        = 110; % The A string of a guitar is normally tuned to 110 Hz
Eoffset  = -5;
Doffset  = 5;
Goffset  = 10;
Boffset  = 14;
E2offset = 19;
```

Generate the frequency vector that we will use for analysis.

```
F = linspace(1/Fs, 1000, 2^12);
```

Generate 4 seconds of zeros to be used to generate the guitar notes.

```
x = zeros(Fs*4, 1);
```

### Play a Note on an Open String

When a guitar string is plucked or strummed, it produces a sound wave with peaks in the frequency domain that are equally spaced. These are called the harmonics and they give each note a full sound. We can generate sound waves with these harmonics with discrete-time filter objects.

Determine the feedback delay based on the first harmonic frequency.

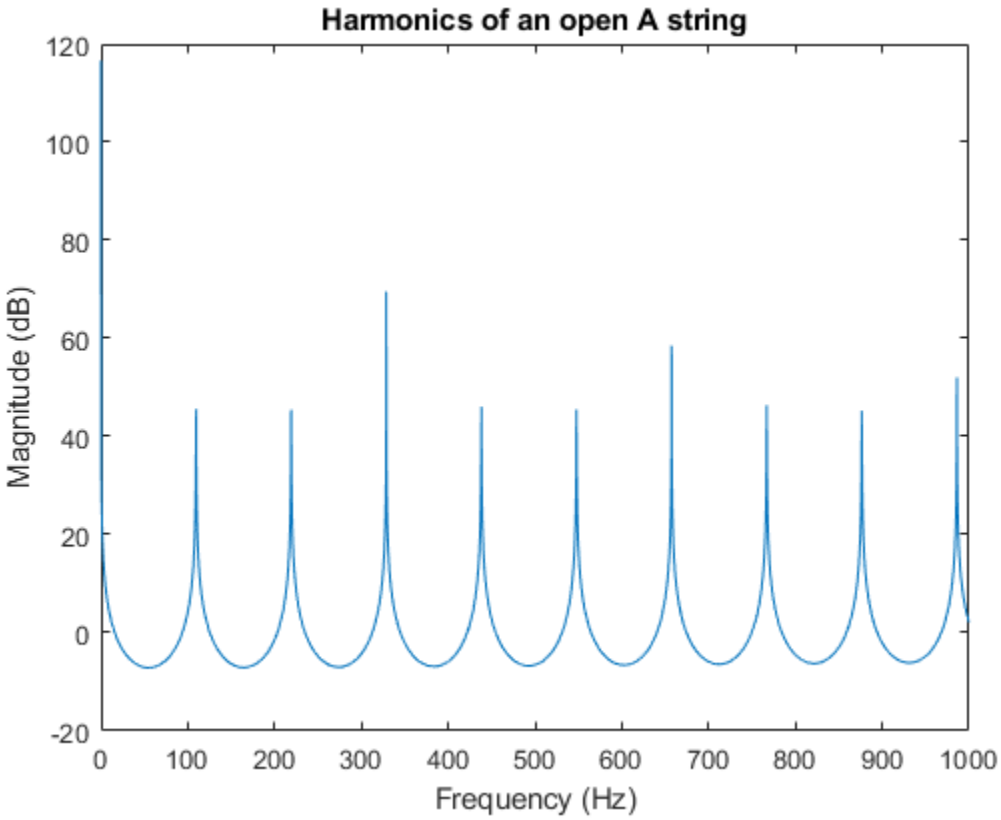
```
delay = round(Fs/A);
```

Generate an IIR filter whose poles approximate the harmonics of the A string. The zeros are added for subtle frequency domain shaping.

```
b  = firls(42, [0 1/delay 2/delay 1], [0 0 1 1]);
a  = [1 zeros(1, delay) -0.5 -0.5];
```

Show the magnitude response of the filter.

```
[H,W] = freqz(b, a, F, Fs);
plot(W, 20*log10(abs(H)));
title('Harmonics of an open A string');
xlabel('Frequency (Hz)');
ylabel('Magnitude (dB)');
```




To generate a 4 second synthetic note first we create a vector of states with random numbers. Then we filter zeros using these initial states. This forces the random states to exit the filter shaped into the harmonics.

```
zi = rand(max(length(b),length(a))-1,1);
note = filter(b, a, x, zi);
```

Normalize the sound for the audioplayer.

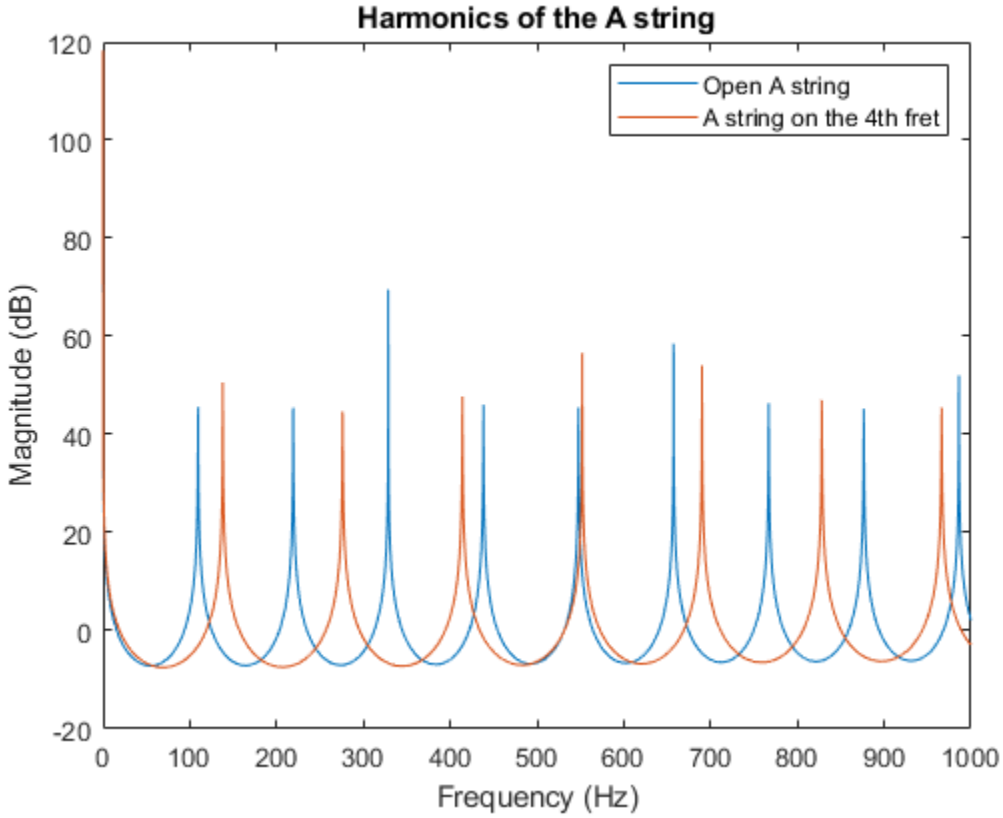
```
note = note-mean(note);
note = note/max(abs(note));

% To hear, type: hplayer = audioplayer(note, Fs); play(hplayer)
```

**Play a Note on a Fretted String**

Each fret along a guitar's neck allows the player to play a half tone higher, or a note whose first harmonic is $2^{1/12}$ higher.

```
fret  = 4;
delay = round(Fs/(A*2^(fret/12)));

b  = firls(42, [0 1/delay 2/delay 1], [0 0 1 1]);
a  = [1 zeros(1, delay) -0.5 -0.5];

[H,W] = freqz(b, a, F, Fs);
hold on
```

```
plot(W, 20*log10(abs(H)));
title('Harmonics of the A string');
legend('Open A string', 'A string on the 4th fret');
```




Populate the states with random numbers.

```
zi = rand(max(length(b),length(a))-1,1);
```

Create a 4 second note.

```
note = filter(b, a, x, zi);
```

Normalize the sound for the audioplayer.

```
note = note-mean(note);
note = note/max(note);

% To hear, type: hplayer = audioplayer(note, Fs); play(hplayer)
```

**Play a Chord**

A chord is a group of notes played together whose harmonics enforce each other. This happens when there is a small integer ratio between the two notes, e.g. a ratio of 2/3 would mean that the first notes third harmonic would align with the second notes second harmonic.

Define the frets for a G major chord.

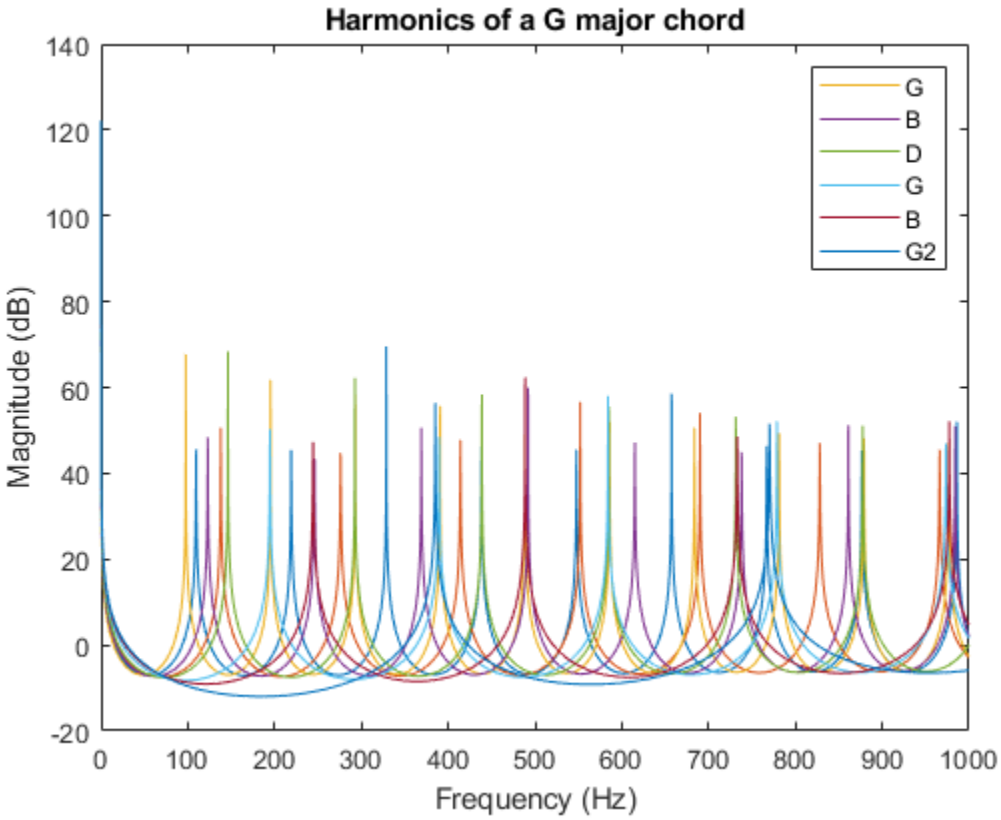
```
fret = [3 2 0 0 0 3];
```

Get the delays for each note based on the frets and the string offsets.

```
delay = [round(Fs/(A*2^((fret(1)+Eoffset)/12))), ...
    round(Fs/(A*2^(fret(2)/12))), ...
    round(Fs/(A*2^((fret(3)+Doffset)/12))), ...
    round(Fs/(A*2^((fret(4)+Goffset)/12))), ...
    round(Fs/(A*2^((fret(5)+Boffset)/12))), ...
    round(Fs/(A*2^((fret(6)+E2offset)/12)))];


b = cell(length(delay),1);
a = cell(length(delay),1);
H = zeros(length(delay),4096);
note = zeros(length(x),length(delay));
for indx = 1:length(delay)

    % Build a cell array of numerator and denominator coefficients.
    b{indx} = firls(42, [0 1/delay(indx) 2/delay(indx) 1], [0 0 1 1]).';
    a{indx} = [1 zeros(1, delay(indx)) -0.5 -0.5].';

    % Populate the states with random numbers and filter the input zeros.
    zi = rand(max(length(b{indx}),length(a{indx}))-1,1);

    note(:, indx) = filter(b{indx}, a{indx}, x, zi);

    % Make sure that each note is centered on zero.
    note(:, indx) = note(:, indx)-mean(note(:, indx));

    [H(indx,:),W] = freqz(b{indx}, a{indx}, F, Fs);
end
```

Display the magnitude for all the notes in the chord.

```
hline = plot(W,20*log10(abs(H.')));
title('Harmonics of a G major chord');
xlabel('Frequency (Hz)');
ylabel('Magnitude (dB)');
legend(hline,'G','B','D','G','B','G2');
```




Combine the notes and normalize them.

```
combinedNote = sum(note,2);
combinedNote = combinedNote/max(abs(combinedNote));

% To hear, type: hplayer = audioplayer(combinedNote, Fs); play(hplayer)
```

**Add a Strumming Effect**

To add a strumming effect we simply offset each previously created note.

Define the offset between strings as 50 milliseconds.

```
offset = 50;
offset = ceil(offset*Fs/1000);
```

Add 50 milliseconds between each note by prepending zeros.

```
for indx = 1:size(note, 2)
    note(:, indx) = [zeros(offset*(indx-1),1); ...
                note((1:end-offset*(indx-1)), indx)];
end

combinedNote = sum(note,2);
combinedNote = combinedNote/max(abs(combinedNote));
```

```
% To hear, type: hplayer = audioplayer(combinedNote, Fs); play(hplayer)
```

## See Also

`firls` | `freqz`

# DFT Estimation with the Goertzel Algorithm

This example shows how to use the Goertzel function to implement a DFT-based DTMF detection algorithm.

Dual-tone Multi-Frequency (DTMF) signaling is the basis for voice communications control and is widely used worldwide in modern telephony to dial numbers and configure switchboards. It is also used in systems such as in voice mail, electronic mail and telephone banking.

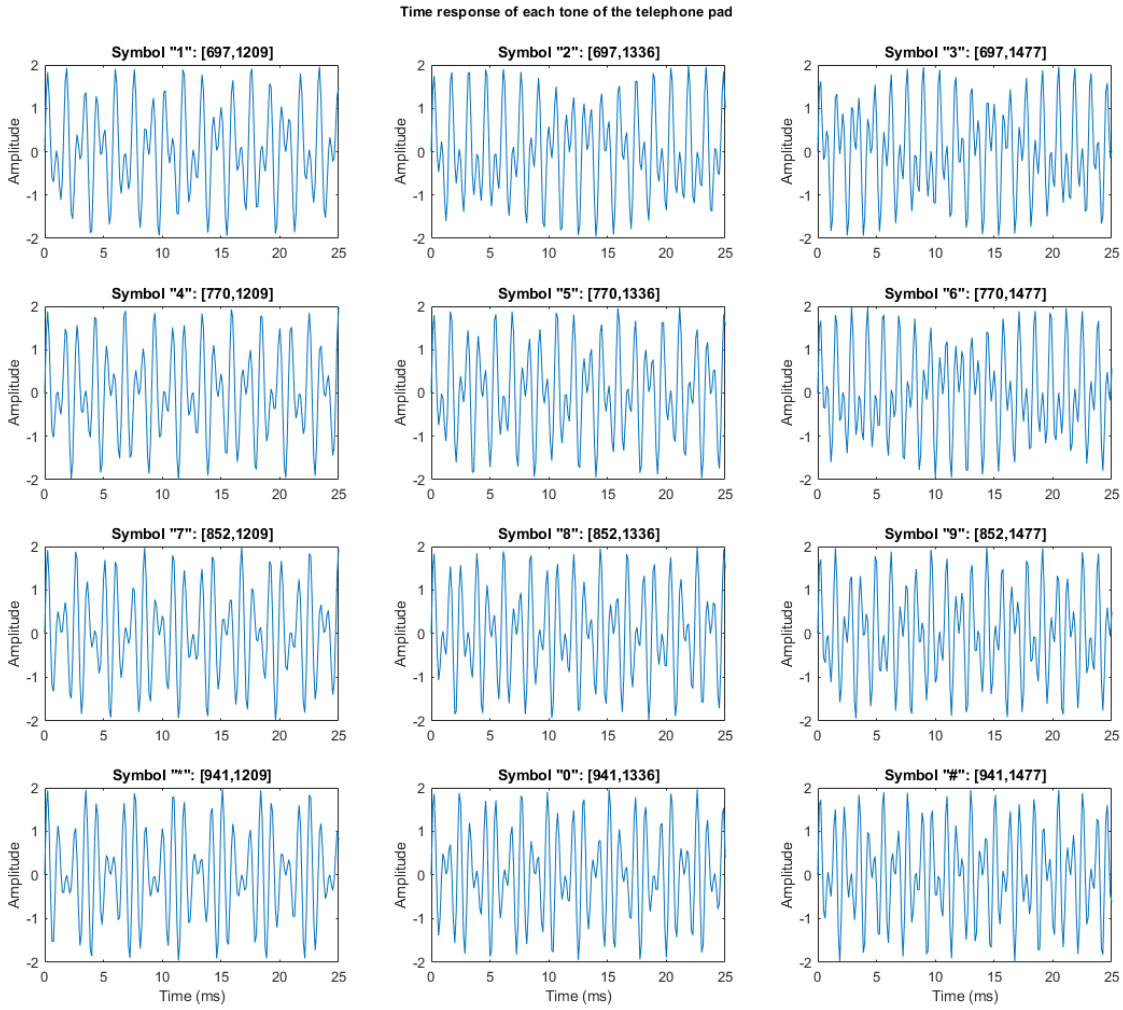
### Generating DTMF Tones

A DTMF signal consists of the sum of two sinusoids – or tones – with frequencies taken from two mutually exclusive groups. These frequencies were chosen to prevent any harmonics from being incorrectly detected by the receiver as some other DTMF frequency. Each pair of tones contains one frequency of the low group (697 Hz, 770 Hz, 852 Hz, 941 Hz) and one frequency of the high group (1209 Hz, 1336 Hz, 1477 Hz) and represents a unique symbol. The frequencies allocated to the push-buttons of the telephone pad are shown below:

| | 1209 Hz | 1336 Hz | 1477 Hz |
|---|---|---|---|
| 697 Hz | 1 | 2 | 3 |
| 770 Hz | 4 | 5 | 6 |
| 852 Hz | 7 | 8 | 9 |
| 941 Hz | * | 0 | # |

Generate and plot a DTMF signal for each button on the telephone pad. Each signal has a sample rate of 8 kHz and a duration of 100 ms.

```
symbol = {'1','2','3','4','5','6','7','8','9','*','0','#'};
[tones, Fs, f, lfg, hfg] = helperDTMFToneGenerator(symbol, false);
helperDFTEstimationPlot1(tones, symbol, Fs, f);
```




### Playing DTMF Tones

For example, play the tones corresponding to phone number 508-647-7000. The `"0"` symbol corresponds to the 11th tone.

```
% To hear, uncomment these lines:

% for i = [5 11 8 6 4 7 7 11 11 11]
%     p = audioplayer(tones(:,i),Fs,16);
%     play(p)
%     pause(0.5)
% end
```

### Estimating DTMF Tones with the Goertzel Algorithm

The minimum duration of a DTMF signal defined by the ITU standard is 40 ms. Therefore, there are at most 0.04 x 8000 = 320 samples available for estimation and detection. The DTMF decoder needs to estimate the frequencies contained in these short signals.

One common approach to this estimation problem is to compute the Discrete-Time Fourier Transform (DFT) samples close to the seven fundamental tones. For a DFT-based solution, it has been shown that using 205 samples in the frequency domain minimizes the error between the original frequencies and the points at which the DFT is estimated.

```
Nt = 205;
original_f = [lfg(:);hfg(:)]  % Original frequencies
```

```
original_f = 7×1

         697
         770
         852
         941
        1209
        1336
        1477
```

```
k = round(original_f/Fs*Nt);  % Indices of the DFT
estim_f = round(k*Fs/Nt)      % Frequencies at which the DFT is estimated
```

```
estim_f = 7×1

         702
         780
         859
         937
        1210
        1327
        1483
```

To minimize the error between the original frequencies and the points at which the DFT is estimated, we truncate the tones, keeping only 205 samples or 25.6 ms for further processing.

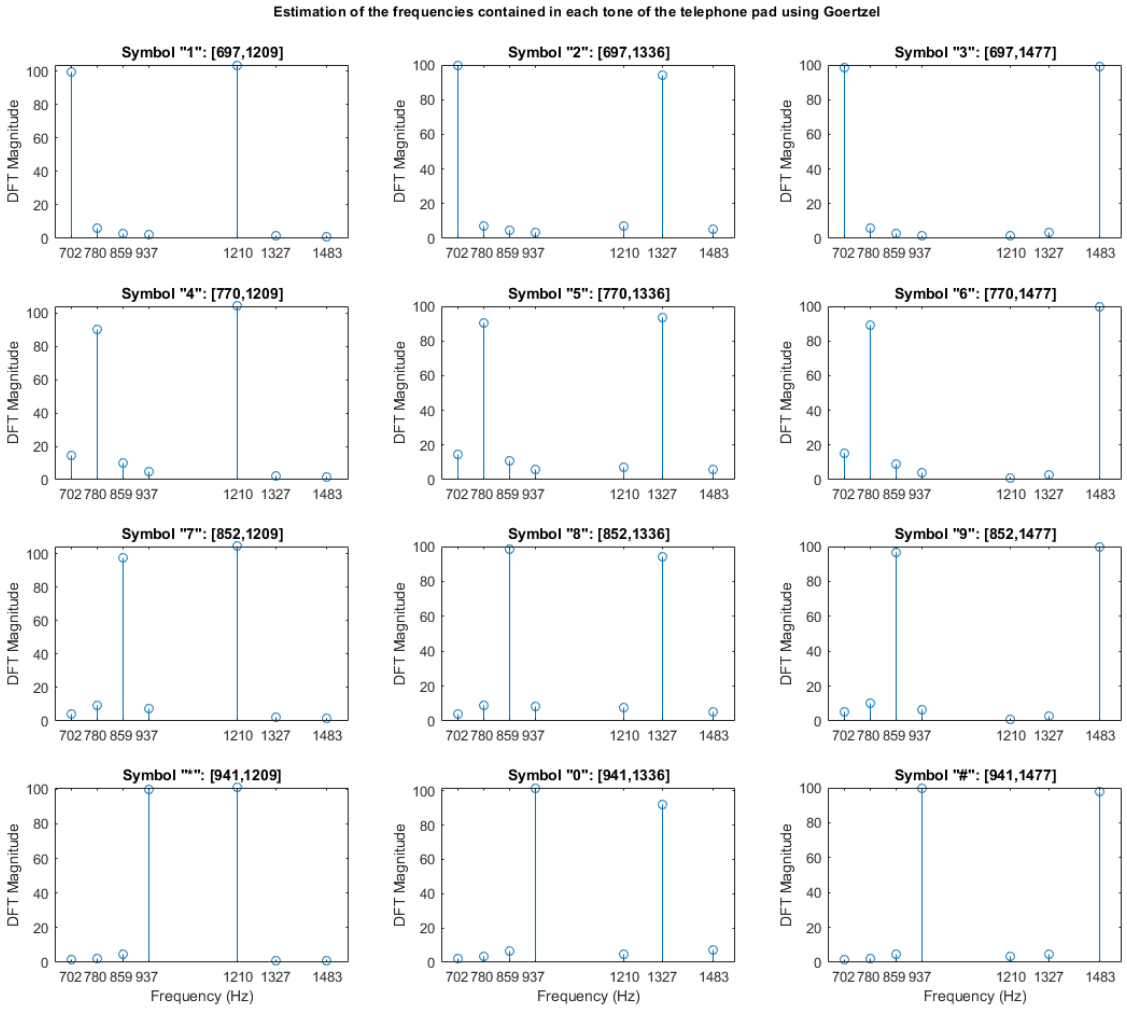
```
tones = tones(1:205,:);
```

At this point we could use the Fast Fourier Transform (FFT) algorithm to calculate the DFT. However, the popularity of the Goertzel algorithm in this context lies in the small number of points at which the DFT is estimated. In this case, the Goertzel algorithm is more efficient than the FFT algorithm.

```
for toneChoice = 1:12
    % Select tone
    tone = tones(:,toneChoice);
    % Estimate DFT using Goertzel
    ydft(:,toneChoice) = goertzel(tone,k+1); % Goertzel uses 1-based indexing
end
```

Plot Goertzel's DFT magnitude estimate of each tone on a grid corresponding to the telephone pad.

```
helperDFTEstimationPlot2(ydft,symbol,f, estim_f);
```




### Detecting DTMF Tones

The digital tone detection can be achieved by measuring the energy present at the seven frequencies estimated above. Each symbol can be separated by simply taking the component of maximum energy in the lower and upper frequency groups.

### Appendix

The following helper functions are used in this example.

- helperDTMFToneGenerator.m
- helperDFTEstimationPlot1.m
- helperDFTEstimationPlot2.m

## See Also

`goertzel`

# Discrete Walsh-Hadamard Transform

### Introduction

This example shows how to use the Walsh-Hadamard transform (WHT) and some of its properties by showcasing two applications: communications using spread spectrum and processing of ECG signals.

WHTs are used in many different applications, such as power spectrum analysis, filtering, processing speech and medical signals, multiplexing and coding in communications, characterizing non-linear signals, solving non-linear differential equations, and logical design and analysis.

The WHT is a suboptimal, non-sinusoidal, orthogonal transformation that decomposes a signal into a set of orthogonal, rectangular waveforms called Walsh functions. The transformation has no multipliers and is real because the amplitude of Walsh (or Hadamard) functions has only two values, +1 or -1.

### Walsh (or Hadamard) Functions

Walsh functions are rectangular or square waveforms with values of -1 or +1. An important characteristic of Walsh functions is sequency which is determined from the number of zero-crossings per unit time interval. Every Walsh function has a unique sequency value.

Walsh functions can be generated in many ways (see [1]). Here we use the `hadamard` function in MATLAB® to generate Walsh functions. Length eight Walsh functions are generated as follows.

```
N = 8;  % Length of Walsh (Hadamard) functions
hadamardMatrix = hadamard(N)
```

```
hadamardMatrix = 8×8

     1     1     1     1     1     1     1     1
     1    -1     1    -1     1    -1     1    -1
     1     1    -1    -1     1     1    -1    -1
     1    -1    -1     1     1    -1    -1     1
     1     1     1     1    -1    -1    -1    -1
     1    -1     1    -1    -1     1    -1     1
     1     1    -1    -1    -1    -1     1     1
     1    -1    -1     1    -1     1     1    -1
```

The rows (or columns) of the symmetric `hadamardMatrix` contain the Walsh functions. The Walsh functions in the matrix are not arranged in increasing order of their sequencies or number of zero-crossings (i.e. 'sequency order') but are arranged in 'Hadamard order'. The Walsh matrix, which contains the Walsh functions along the rows or columns in the increasing order of their sequencies is obtained by changing the index of the `hadamardMatrix` as follows.

```
HadIdx = 0:N-1;                          % Hadamard index
M = log2(N)+1;                           % Number of bits to represent the index
```

Each column of the sequency index (in binary format) is given by the modulo-2 addition of columns of the bit-reversed Hadamard index (in binary format).

```
binHadIdx = fliplr(dec2bin(HadIdx,M))-'0'; % Bit reversing of the binary index
binSeqIdx = zeros(N,M-1);                  % Pre-allocate memory
for k = M:-1:2
    % Binary sequency index
```

```
    binSeqIdx(:,k) = xor(binHadIdx(:,k),binHadIdx(:,k-1));
end
SeqIdx = binSeqIdx*pow2((M-1:-1:0)');    % Binary to integer sequency index
walshMatrix = hadamardMatrix(SeqIdx+1,:) % 1-based indexing
```

```
walshMatrix = 8×8

     1     1     1     1     1     1     1     1
     1     1     1     1    -1    -1    -1    -1
     1     1    -1    -1    -1    -1     1     1
     1     1    -1    -1     1     1    -1    -1
     1    -1    -1     1     1    -1    -1     1
     1    -1    -1     1    -1     1     1    -1
     1    -1     1    -1    -1     1    -1     1
     1    -1     1    -1     1    -1     1    -1
```

### Discrete Walsh-Hadamard Transform

The forward and inverse Walsh transform pair for a signal x(t) of length N are

$$y_n = \frac{1}{N}\sum_{i=0}^{N-1} x_i WAL(n,i), n = 1, 2, \ldots, N-1$$

$$x_i = \sum_{n=0}^{N-1} y_n WAL(n,i), i = 1, 2, \ldots, N-1$$

Fast algorithms, similar to the Cooley-Tukey algorithm, have been developed to implement the Walsh-Hadamard transform with complexity $O(N \log N)$ (see [1] and [2]). Since the Walsh matrix is symmetric, both the forward and inverse transformations are identical operations except for the scaling factor of 1/N. The functions `fwht` and `ifwht` implement the forward and the inverse WHT respectively.

**Example 1** Perform WHT on the Walsh matrix. The expected result is an identity matrix because the rows (or columns) of the symmetric Walsh matrix contain the Walsh functions.

```
y1 = fwht(walshMatrix)                % Fast Walsh-Hadamard transform
```

```
y1 = 8×8

     1     0     0     0     0     0     0     0
     0     1     0     0     0     0     0     0
     0     0     1     0     0     0     0     0
     0     0     0     1     0     0     0     0
     0     0     0     0     1     0     0     0
     0     0     0     0     0     1     0     0
     0     0     0     0     0     0     1     0
     0     0     0     0     0     0     0     1
```

**Example 2** Construct a discontinuous signal by scaling and adding arbitrary columns of the Hadamard matrix. This signal is formed using weighted Walsh functions, so the WHT should return non-zero values equal to the weights at the respective sequency indices. While evaluating the WHT, the `ordering` is specified as `'hadamard'`, because a Hadamard matrix (instead of the Walsh matrix) is used to obtain the Walsh functions.

```
N = 8;
H = hadamard(N);                        % Hadamard matrix
% Construct a signal by adding a few weighted Walsh functions
x = 8.*H(1,:) + 12.*H(3,:) + 18.*H(5,:) + 10.*H(8,:);
y = fwht(x,N,'hadamard')
```

```
y = 1×8

     8     0    12     0    18     0     0    10
```

WHT is a reversible transform and the original signal can be recovered perfectly using the inverse transform. The norm between the original signal and the signal obtained from inverse transformation equals zero, indicating perfect reconstruction.

```
xHat = ifwht(y,N,'hadamard');
norm(x-xHat)
```

```
ans = 0
```

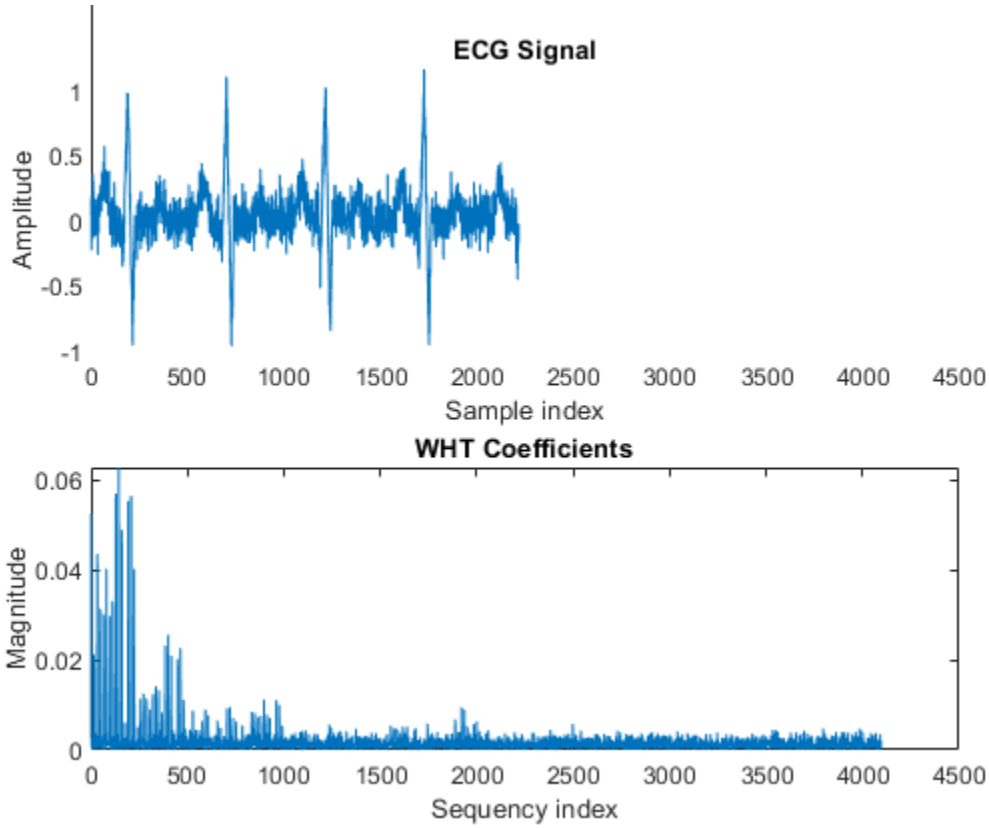
The Walsh-Hadamard transform involves expansion using a set of rectangular waveforms, so it is useful in applications involving discontinuous signals that can be readily expressed in terms of Walsh functions. Below are two applications of Walsh-Hadamard transforms.

### Walsh-Transform Applications

**ECG signal processing** Often, it is necessary to record electro-cardiogram (ECG) signals of patients at different instants of time. This results in a large amount of data, which needs to be stored for analysis, comparison, etc. at a later time. Walsh-Hadamard transform is suitable for compression of ECG signals because it offers advantages such as fast computation of Walsh-Hadamard coefficients, less required storage space since it suffices to store only those sequency coefficients with large magnitudes, and fast signal reconstruction.

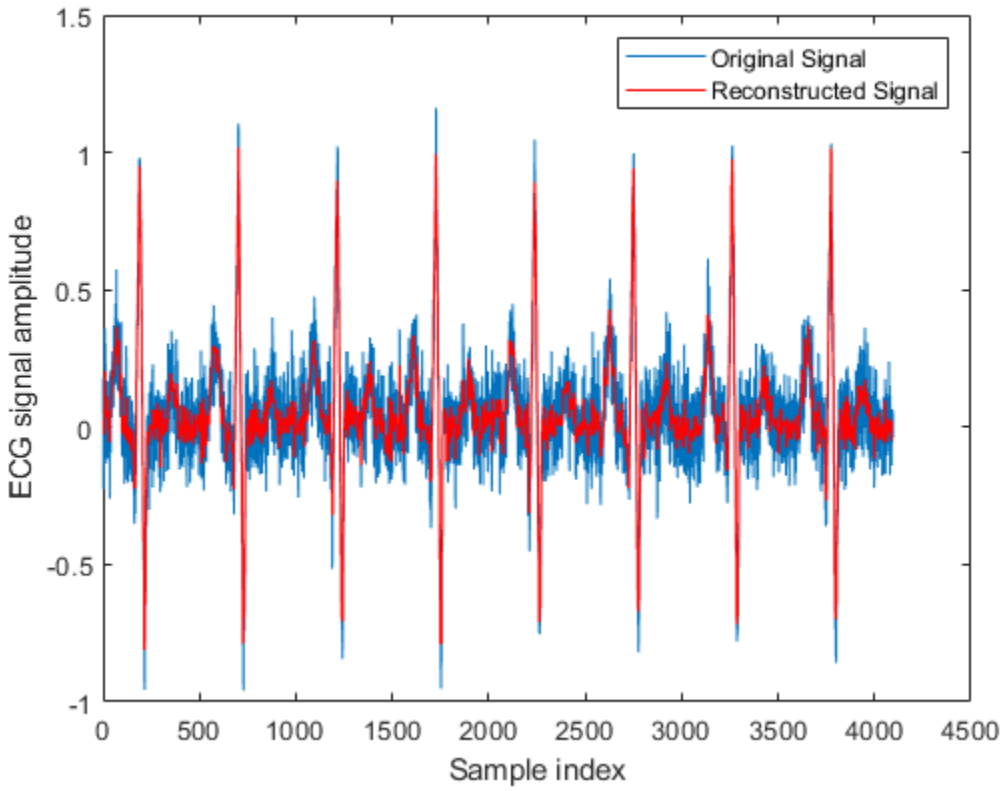
An ECG signal and its corresponding Walsh-Hadamard transform is evaluated and shown below.

```
x1 = ecg(512);                    % Single ecg wave
x = repmat(x1,1,8);
x = x + 0.1.*randn(1,length(x));  % Noisy ecg signal
y = fwht(x);                      % Fast Walsh-Hadamard transform
figure
subplot(2,1,1)
plot(x)
xlabel('Sample index')
ylabel('Amplitude')
title('ECG Signal')
subplot(2,1,2)
plot(abs(y))
xlabel('Sequency index')
ylabel('Magnitude')
title('WHT Coefficients')
```




As can be seen in the above plot, most of the signal energy is concentrated at lower sequency values. For investigation purposes, only the first 1024 coefficients are stored and used to reconstruct the original signal. Truncating the higher sequency coefficients also helps with noise suppression. The original and the reproduced signals are shown below.

```
y(1025:length(x)) = 0;              % Zeroing out the higher coefficients
xHat = ifwht(y);                    % Signal reconstruction using inverse WHT
figure
plot(x)
hold on
plot(xHat,'r')
xlabel('Sample index')
ylabel('ECG signal amplitude')
legend('Original Signal','Reconstructed Signal')
```




The reproduced signal is very close to the original signal.

To reconstruct the original signal, we stored only the first 1024 coefficients and the ECG signal length. This represents a compression ratio of approximately 4:1.

```
req = [length(x) y(1:1024)];
whos x req
```

```
  Name      Size              Bytes  Class     Attributes

  req       1x1025             8200  double
  x         1x4096            32768  double
```

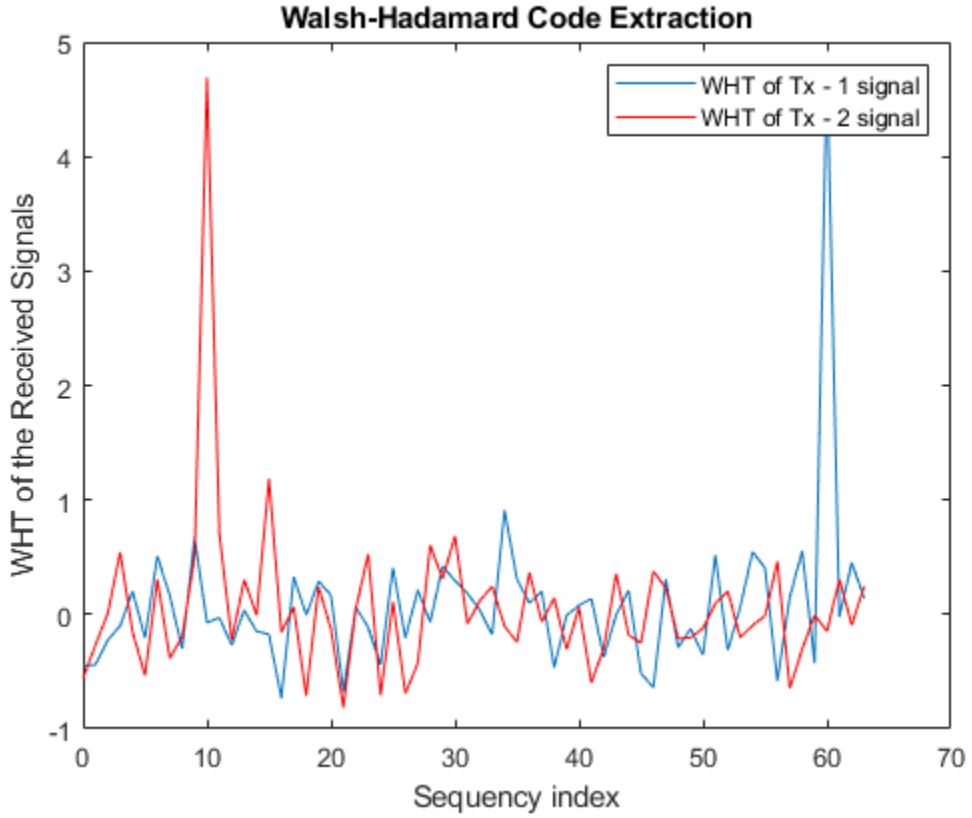
**Communication using Spread Spectrum** Spread spectrum-based communication technologies, like CDMA, use Walsh codes (derived from Walsh functions) to spread message signals and WHT transforms to despread them. Since Walsh codes are orthogonal, any Walsh-encoded signal appears as random noise to a terminal unless that terminal uses the same code for encoding. Below we show the process of spreading, determining Walsh codes used for spreading, and despreading to recover the message signal.

Two CDMA terminals spread their respective message signals using two different Walsh codes (also known as Hadamard codes) of length 64. The spread message signals are corrupted by a additive white Gaussian noise of variance 0.1.

At the receiver (base station), signal processing is non-coherent and the received sequence of length N needs to be correlated with 2^N Walsh codewords to extract the Walsh codes used by the respective transmitters. This can be effectively done by transforming the received signals to sequency

domain using the fast Walsh-Hadamard transform. Using the sequency location at a which a peak occurs, the corresponding Walsh-Hadamard code (or the Walsh function) used can be determined. The plot below shows that Walsh-Hadamard codes with sequency (with `ordering = 'hadamard'`) 60 and 10 were used in the first and the second transmitter, respectively.

```
load mess_rcvd_signals.mat
N = length(rcvdSig1);
y1 = fwht(rcvdSig1,N,'hadamard');
y2 = fwht(rcvdSig2,N,'hadamard');
figure
plot(0:63,y1,0:63,y2,'r')
xlabel('Sequency index')
ylabel('WHT of the Received Signals')
title('Walsh-Hadamard Code Extraction')
legend('WHT of Tx - 1 signal','WHT of Tx - 2 signal')
```




Despreading (or decoding) to extract the message signal can be carried out in a straightforward manner by multiplying the received signals by the respective Walsh-Hadamard codes generated using the `hadamard` function. (Note that the indexing in MATLAB® starts from 1, hence Walsh-Hadamard codes with sequency 60 and 10 are obtained from by selecting the columns (or rows) 61 and 11 in the Hadamard matrix.)

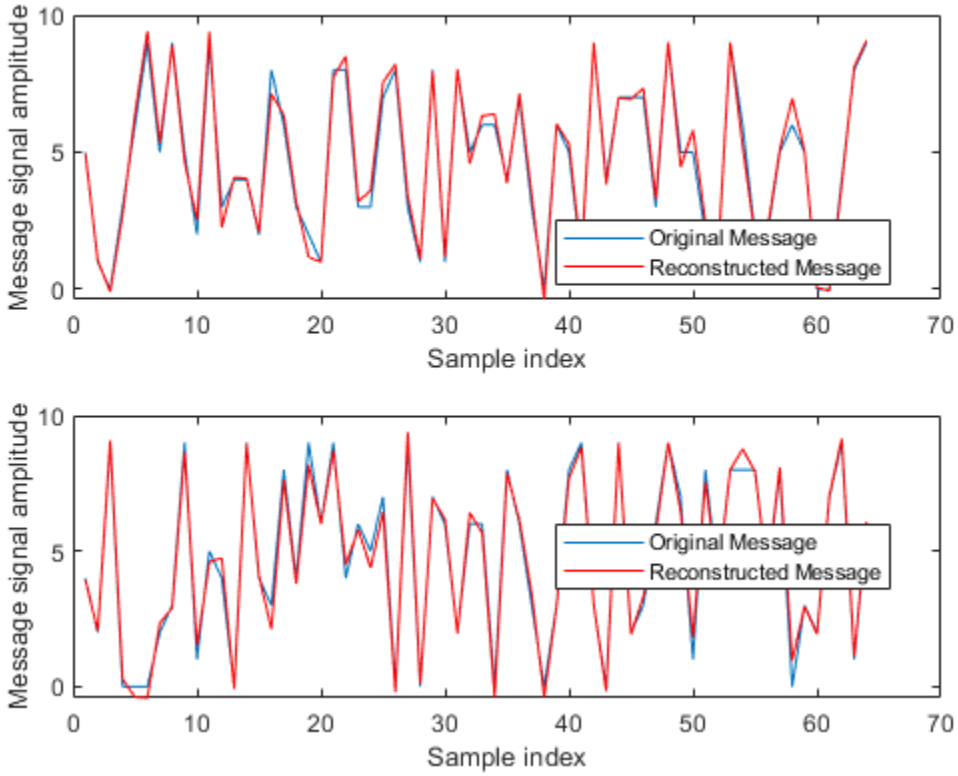
```
N = 64;
hadamardMatrix = hadamard(N);
codeTx1 = hadamardMatrix(:,61);          % Code used by transmitter 1
codeTx2 = hadamardMatrix(:,11);          % Code used by transmitter 2
```

The decoding operation to recover the original message signal is

```
xHat1 = codeTx1 .* rcvdSig1;            % Decoded signal at receiver 1
xHat2 = codeTx2 .* rcvdSig2;            % Decoded signal at receiver 2
```

The recovered message signals at the receiver side are shown below and superimposed with the original signals for comparison.

```
subplot(2,1,1)
plot(x1)
hold on
plot(xHat1,'r')
legend('Original Message','Reconstructed Message','Location','Best')
xlabel('Sample index')
ylabel('Message signal amplitude')
subplot(2,1,2)
plot(x2)
hold on
plot(xHat2,'r')
legend('Original Message','Reconstructed Message','Location','Best')
xlabel('Sample index')
ylabel('Message signal amplitude')
```




## References

**1** K.G. Beauchamp, *Applications of Walsh and Related Functions - With an Introduction to Sequency Theory*, Academic Press, 1984

2 T. Beer, *Walsh Transforms*, American Journal of Physics, Vol. 49, Issue 5, May 1981

## See Also

`fwht` | `ifwht`

# Single Sideband Modulation via the Hilbert Transform

This example shows how to use the discrete Hilbert Transform to implement Single Sideband Modulation.

The Hilbert Transform finds applications in modulators and demodulators, speech processing, medical imaging, direction of arrival (DOA) measurements, essentially anywhere complex-signal (quadrature) processing simplifies the design.

### Introduction

Single Sideband (SSB) Modulation is an efficient form of Amplitude Modulation (AM) that uses half the bandwidth used by AM. This technique is most popular in applications such as telephony, HAM radio, and HF communications, i.e., voice-based communications. This example shows how to implement SSB Modulation using a Hilbert Transformer.

To motivate the need to use a Hilbert Transformer in SSB modulation, it's helpful to first quickly review double sideband modulation.

### Double Sideband Modulation

A simple form of AM is the Double Sideband (DSB) Modulation, which typically consists of two frequency-shifted copies of a modulated signal on either side of a carrier frequency. More precisely this is referred to as a DSB Suppressed Carrier, and is defined as

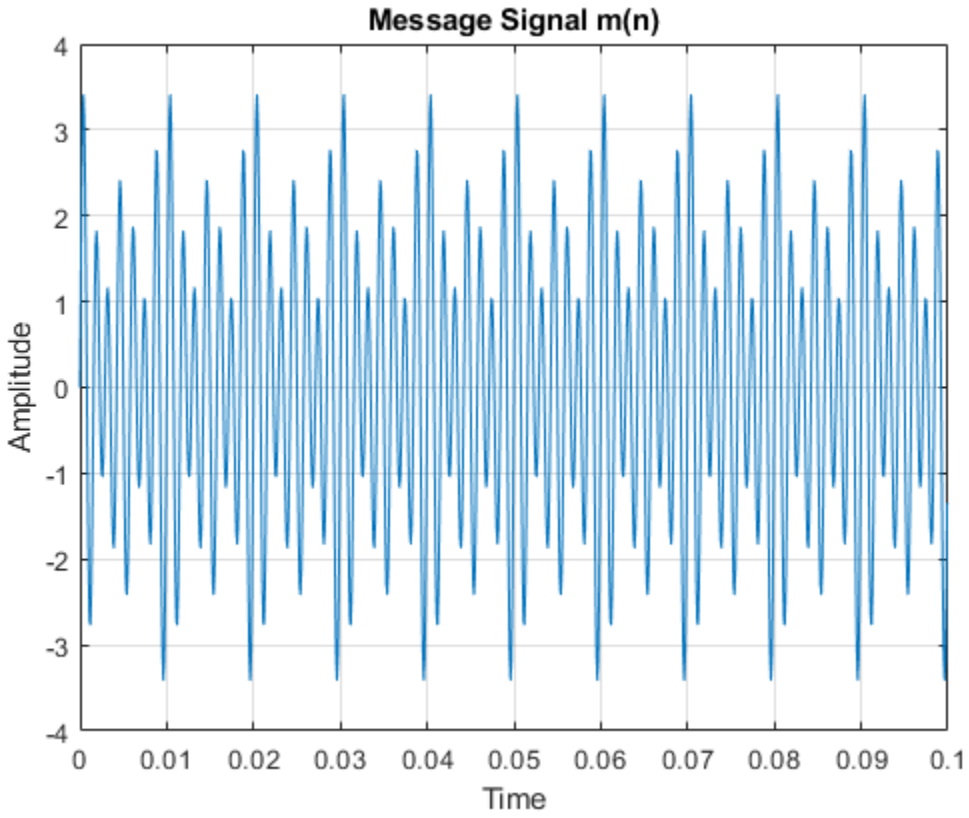
$$f(n) = m(n)\cos(2\pi f_0 n/f_s)$$

where $m(n)$ is usually referred to as the message signal and $f_0$ is the carrier frequency. As shown in the equation above, DSB modulation consists of multiplying the message signal $m(n)$ by the carrier $\cos(2\pi f_0 n/f_s)$, therefore, we can use the modulation theorem of Fourier transforms to calculate the transform of $f(n)$

$$F(f) = \frac{1}{2}[M(f - f_0) + M(f + f_0)]$$

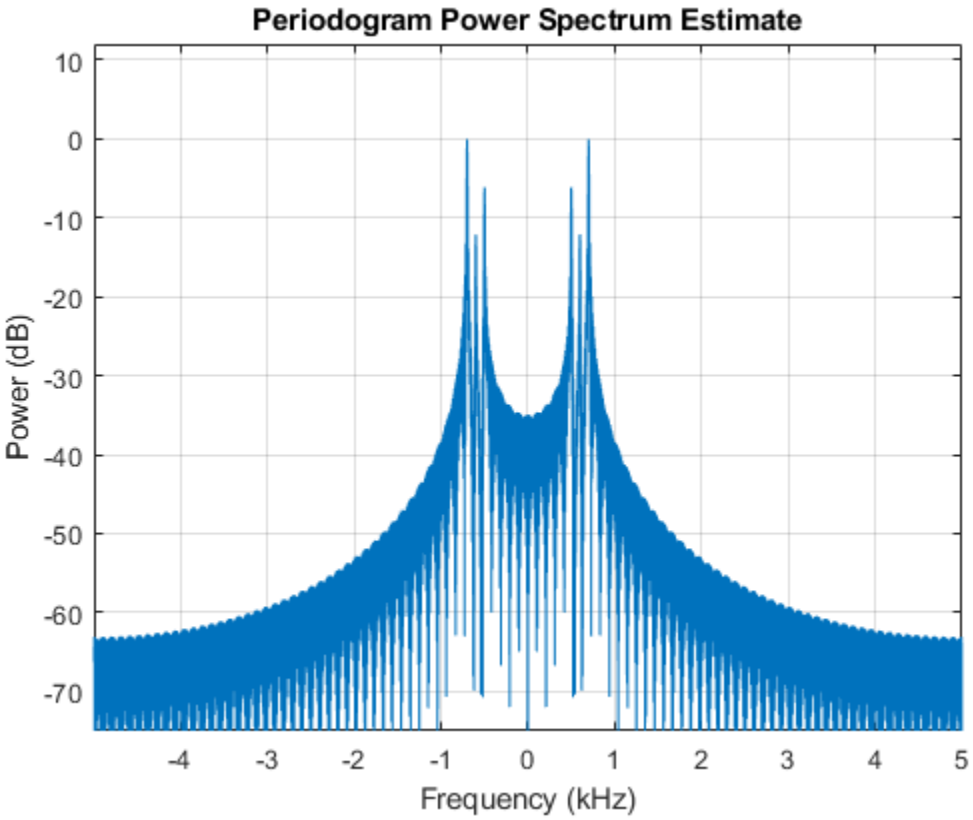
where $M(f)$ is the Discrete-time Fourier Transform (DTFT) of $m(n)$. If the message signal is lowpass with bandwidth $W$, then $F(f)$ is a bandpass signal with twice the bandwidth. Let's look at an example DSB signal and its spectrum.

```
% Define and plot a message signal which contains three tones at 500, 600,
% and 700 Hz with varying amplitudes.
Fs = 10e3;
t = 0:1/Fs:0.1-1/Fs;
m = sin(2*pi*500*t) + 0.5*sin(2*pi*600*t) + 2*sin(2*pi*700*t);
plot(t,m)

grid
xlabel('Time')
ylabel('Amplitude')
title('Message Signal m(n)')
```



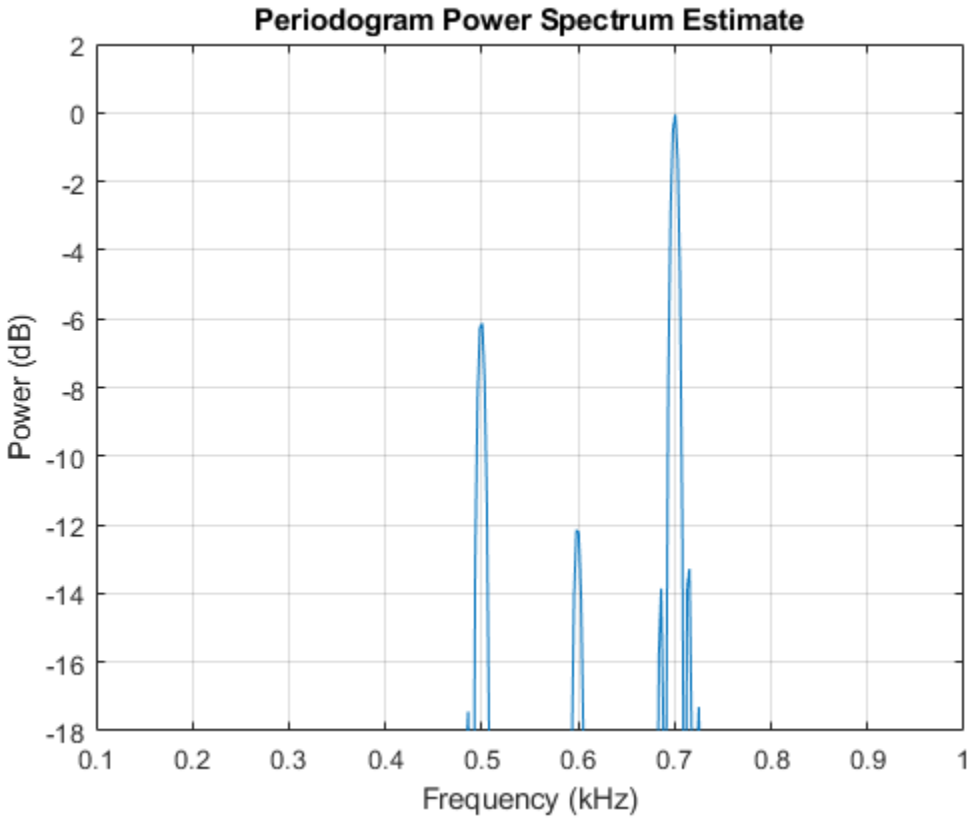

Below we calculate and plot the power spectrum of the message signal.

```
periodogram(m,[],4096,Fs,'power','centered')
ylim([-75 12])
```




The double-sided power spectrum clearly shows the three tones near DC. If we zoom in further we'll be able to read the power of each component.

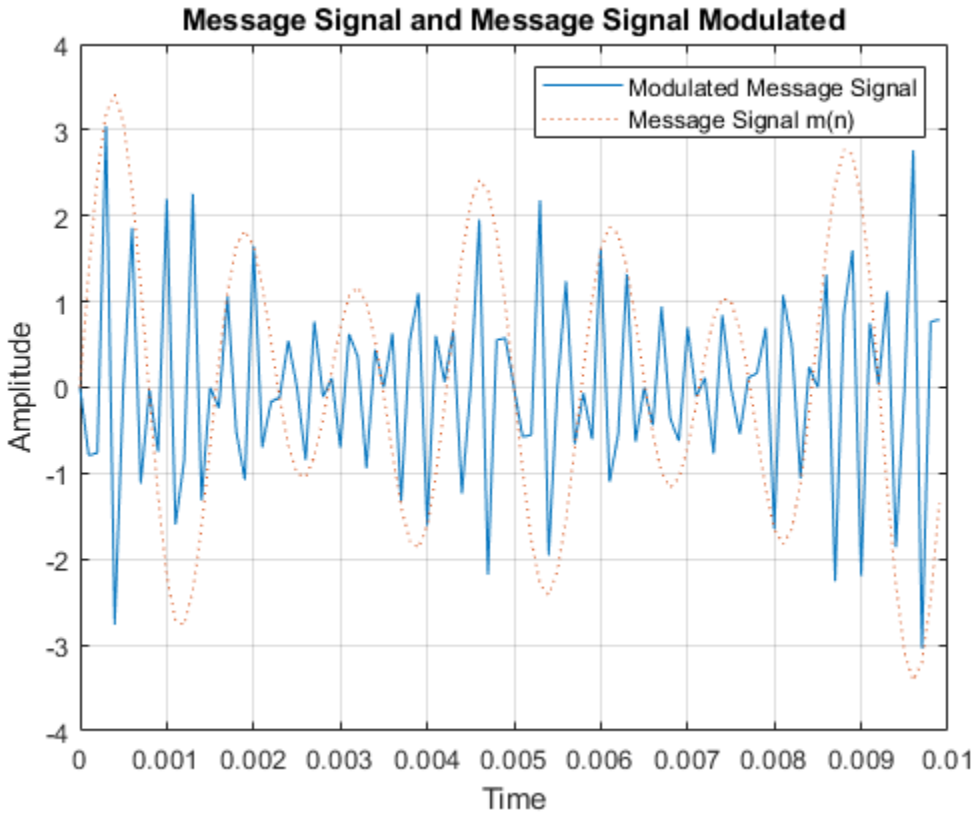
```
xlim([0.1 1])
ylim([-18 2])
```




The power for the 500 Hz tone is roughly –6 dB, for the 600 Hz tone is –12 dB, and for the 700 Hz tone is 0 dB, which corresponds to the message signal's tone amplitudes of 1, 0.5, and 2, respectively.

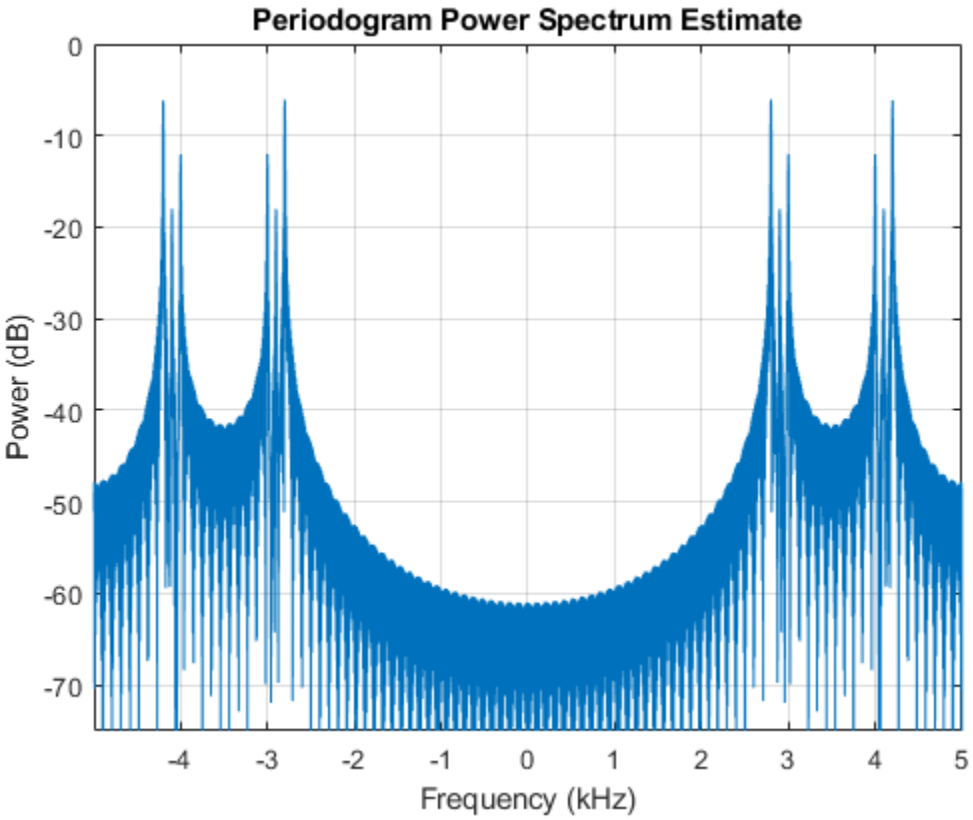
Using this message signal $m(n)$, let's multiply it by a carrier to create a DSB signal, and look at its spectrum.

```
fo = 3.5e3; % Carrier frequency in Hz
f = m.*cos(2*pi*fo*t);
idx = 100;
plot(t(1:idx),f(1:idx),t(1:idx),m(1:idx),':')
grid

xlabel('Time')
ylabel('Amplitude')
title('Message Signal and Message Signal Modulated')
legend('Modulated Message Signal','Message Signal m(n)')
```



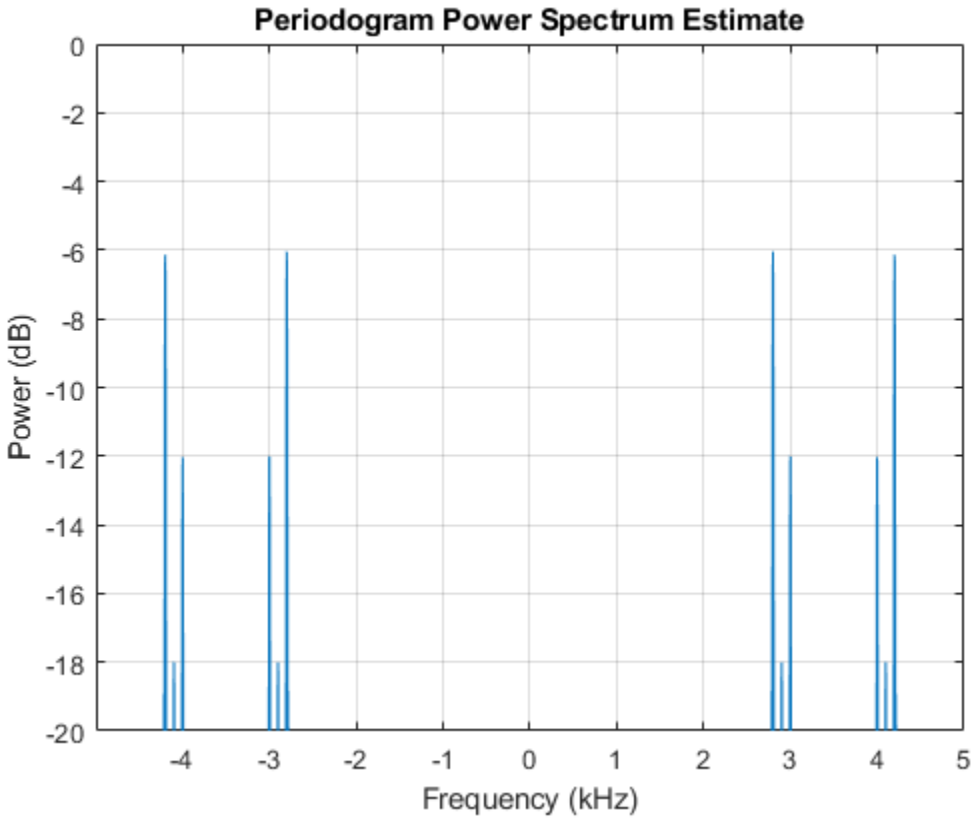

The blue solid line is the modulated message signal, and the red dotted line is the slow varying message signal. The power spectrum of our modulated signal is then

```
periodogram(f,[],4096,Fs,'power','centered')
ylim([-75 0])
```




We can see that the message signal (three tones), has been shifted to the center frequency $f_0$. Moreover, each component's power has been reduced to one quarter, due to the amplitudes being halved, as indicated by the DTFT of the modulated $m(n)$. Let's zoom to read the new power values

```
ylim([-20 0])
```




Our positive frequency components are now at –6 dB, –18 dB, and –12 dB.

Now that we've defined DSB modulation, let's take a look at single sideband modulation.

### Single Sideband Modulation

Single Sideband (SSB) Modulation is similar to DSB modulation, but instead of using the whole spectrum it uses a filter to select either the lower or upper sideband. The selection of the lower or upper sideband results in the lower sideband (LSB) or upper sideband (USB) modulation, respectively. There are two approaches to eliminating one of the sidebands, one is the filter method and the other is the phasing method. The process of selective filtering of the upper or lower sideband is difficult due to the stringent filters required, especially if there's signal content close to DC. This example shows how to use the phasing method, which uses a Hilbert Transformer to implement SSB Modulation.

SSB modulation requires the shifting of the message signal to another center frequency without creating pairs of frequency components $X(f - f_0)$ and $X(f + f_0)$ as in the case of the DSB modulation, i.e., avoiding the need to filter either the upper or lower sideband. This can be done by using a Hilbert Transformer.

Let's first review the definition and properties of the ideal Hilbert Transform before we discuss its use in SSB modulation. This will help motivate its use in SSB modulation.

### Ideal Hilbert Transform

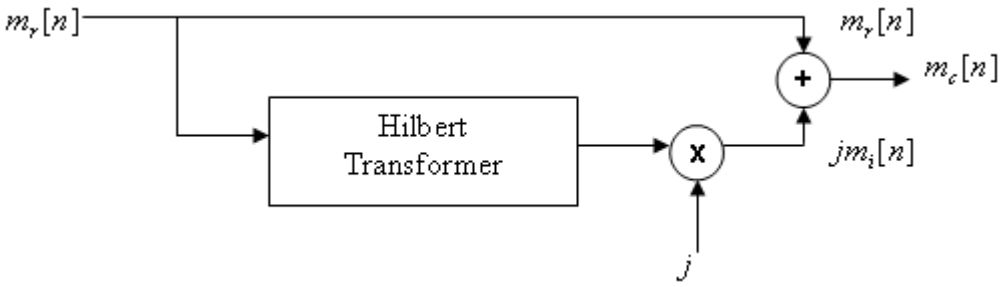
The discrete Hilbert Transform is a process by which a signal's negative frequencies are phase-advanced by 90 degrees and the positive frequencies are phase-delayed by 90 degrees. Shifting the

results of the Hilbert Transform (+ $j$) and adding it to the original signal creates a complex signal as we'll see below.

If $m_i(n)$ is the Hilbert Transform of $m_r(n)$, then:

$$m_c(n) = m_r(n) + jm_i(n)$$

is a complex signal known as the Analytic Signal. The diagram below shows the generation of an analytic signal by means of the ideal Hilbert Transform.

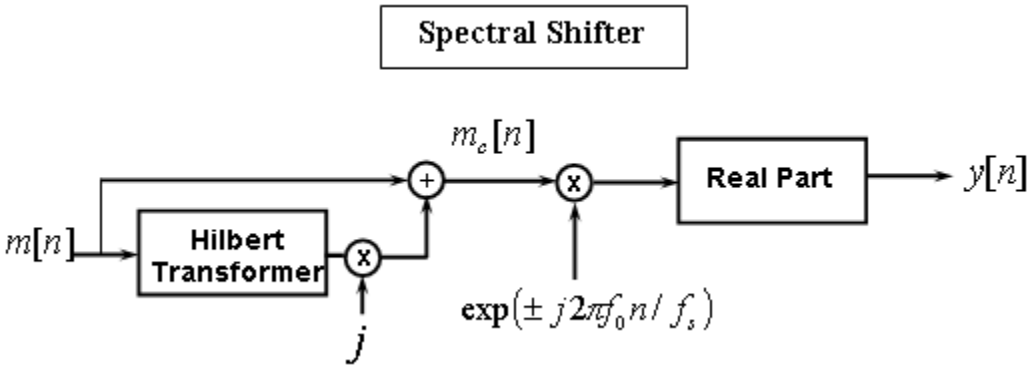



One important characteristic of the analytic signal is that its spectral content lies in the positive Nyquist interval. This is because if we shift the imaginary part of our analytic (complex) signal by 90 degrees (+ $j$) and add it to the real part, the negative frequencies will cancel while the positive frequencies will add. This results in a signal with no negative frequencies. Also, the magnitude of the frequency component in the complex signal is twice the magnitude of the frequency component in the real signal. This is similar to a one-sided spectrum, which contains the total signal power in the positive frequencies.

Next we introduce a Spectral Shifter. The Spectral Shifter shifts (translates) the spectral content of a signal by modulating the analytic signal formed from the signal whose spectrum we want to shift. This concept can be used for SSB modulation as shown later.

**Spectral Shifter**

Using the message signal $m(n)$ defined above we'll create an analytic signal by employing the Hilbert Transform, which will then be modulated to the desired center frequency. The scheme is shown in the diagram below.




Using this method of spectral shifting will ensure that the power of our signal is shifted to the frequency of interest while maintaining a real-valued signal in the end.

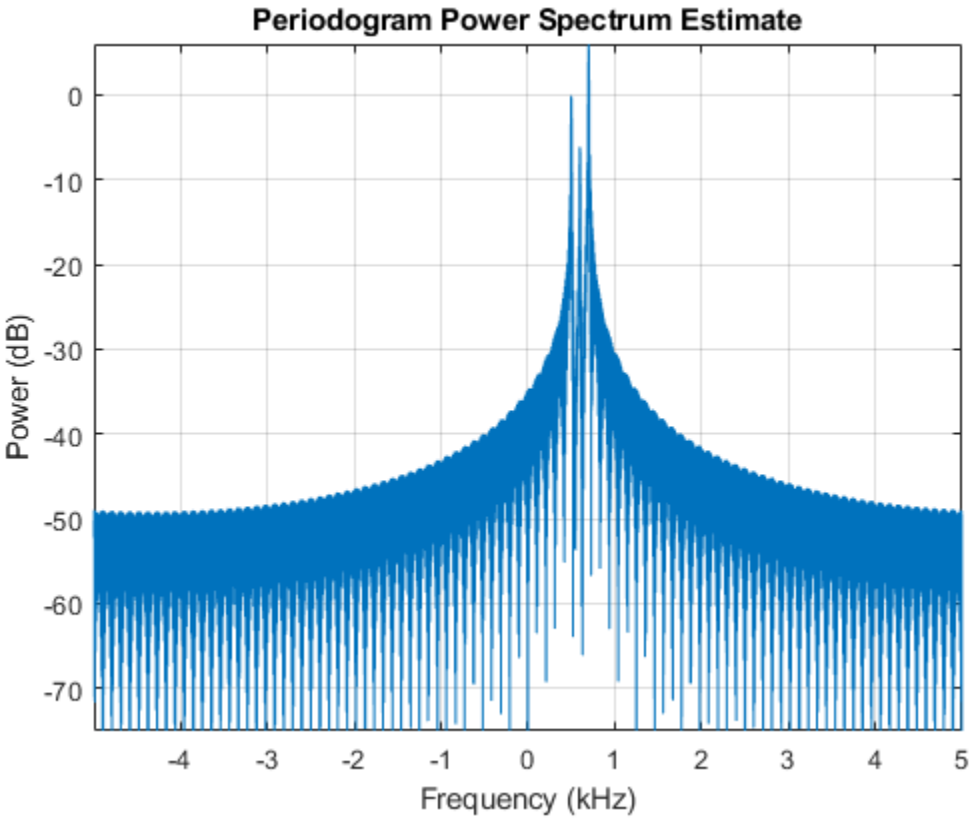
As we indicated earlier the analytic signal is made up of the original real-valued signal plus the Hilbert Transform of that real signal. Running the real signal by the hilbert function in the Signal Processing Toolbox™ will produce an analytic signal.

Note: The hilbert function produces the complete analytic (complex) signal, not just the imaginary part.

```
mc = hilbert(m);
```

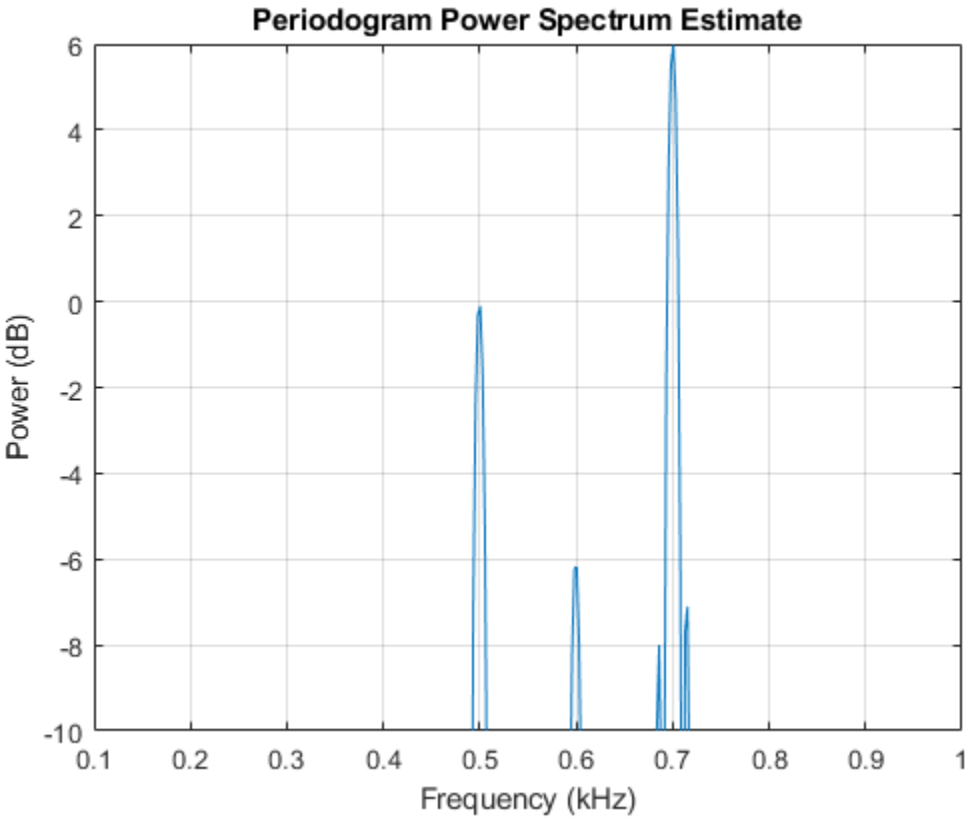
We can also calculate and plot the spectral content of the analytic signal constructed from our message signal $m(n)$.

```
periodogram(mc,[],4096,Fs,'power','centered')
ylim([-75 6])
```




As shown in the spectrum plot, our analytic signal is complex and only contains positive frequency components. Moreover, if we measure the power, or zoom in our plot further at the positive frequency component we'll see that the power of the frequency components of the analytic signal is twice the total power of the positive (or negative) frequency component of the real signal, i.e., it's similar to a one-sided spectrum which contains the signal's total power. See zoomed-in plot below.

```
xlim([0.1 1])
ylim([-10 6])
```




We see that the power of the analytic (complex) signal's frequency components 500 Hz, 600 Hz, and 700 Hz are roughly 0, –6 dB, and 6 dB, respectively, which is the original signal's total power. These values correspond to our original real-valued signal which has three tones with amplitudes of 1, 0.5, and 2, respectively.

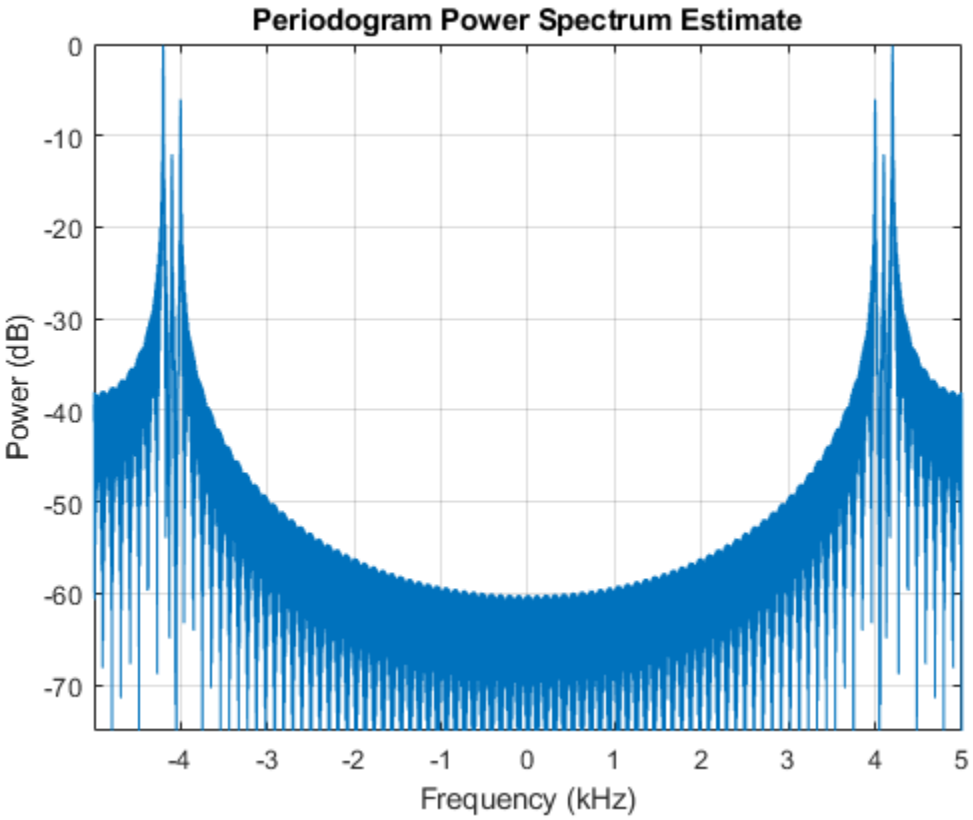
At this point we can modulate the analytic signal to shift the spectral content to another center frequency without producing frequency component pairs and maintain a real-valued signal.

To modulate the signal to the carrier frequency $f_0$, we'll multiply the analytic signal by an exponential.

```
mcm = mc.*exp(1i*2*pi*fo*t);
```

As shown in the Spectral Shifter diagram, after modulating our signal we'll compute the real part. The spectrum of which is

```
periodogram(real(mcm),[],4096,Fs,'power','centered')
ylim([-75 0])
```




As shown in the plot above our signal has been modulated to a new center frequency of $f_0$ without creating the frequency pairs, i.e., it resulted in upper sideband.

If we compare the spectral plot above with that of the DSB modulation we can see that the Spectral Shifter accomplished the SSB modulation.

**Efficient Implementation of SSB Modulation**

From our previous derivation we can see that the SSB modulated signal, $f(n)$ can be written as

$$f(n) = \Re\,[m_c(n)\exp(j2\pi f_0 n/f_s)]$$

where $m_c(n)$ is the analytic signal defined as

$$m_c(n) = m(n) + j\widetilde{m}(n)$$

Expanding that equation and taking the real part we get

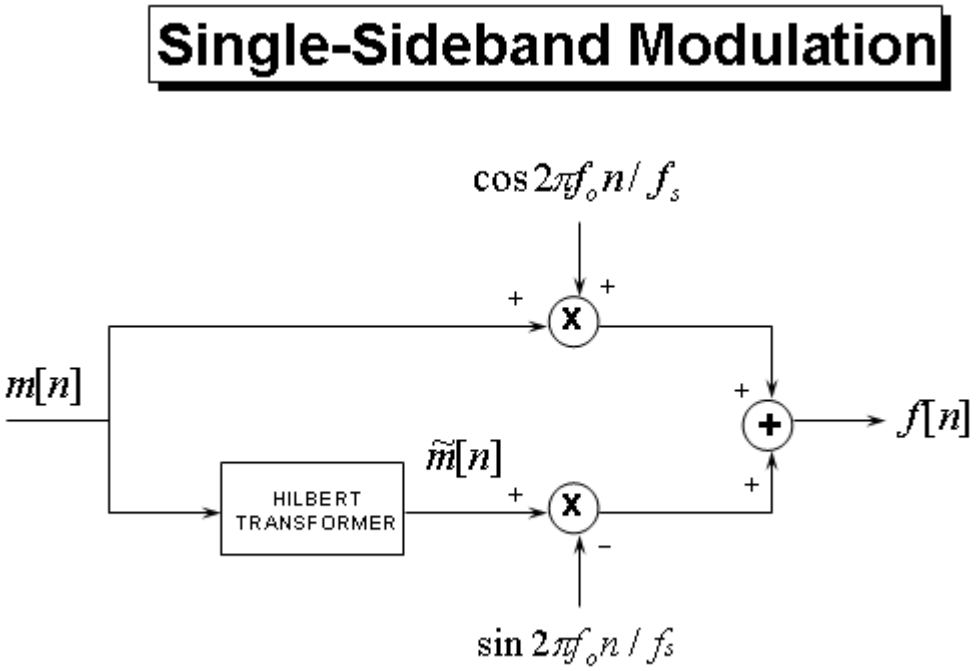
$$f(n) = [m(n)\cos(2\pi f_0 n/f_s) - \widetilde{m}(n)\sin(2\pi f_0 n/f_s)]$$

which results in a single sideband, upper sideband (SSBU). Similarly, we can define the SSB lower sideband (SSBL) by

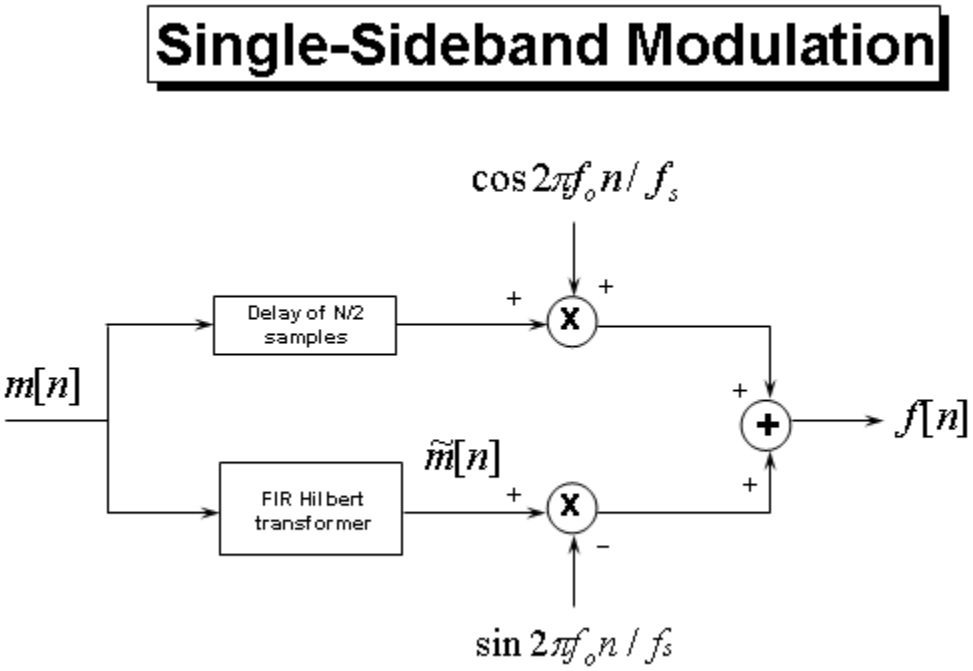
$$f(n) = \Re\,[m_c(n)\exp(-j2\pi f_0 n/f_s)]$$

$$f(n) = [m(n)\cos(2\pi f_0 n/f_s) + \widetilde{m}(n)\sin(2\pi f_0 n/f_s)]$$

The SSBU equation above suggests a more efficient way of implementing SSB. Rather than performing the complex multiplication of $m_c(n)$ with $\exp(j2\pi f_0 n/f_s)$ and then throwing away the imaginary part, we can compute only the quantities we need by implementing SSBU as shown below.




To implement the SSB modulation shown above we need to calculate the Hilbert Transform of our message signal $m(n)$ and modulate both signals. But before we do that we need to point out the fact that ideal Hilbert transformers are not realizable. However, algorithms that approximate the Hilbert Transformer, such as the Parks-McClellan FIR filter design technique, have been developed which can be used. Signal Processing Toolbox™ provides the `firpm` function, which designs such filters. Also, since the filter introduces a delay we need to compensate for that delay by adding delay ($N/2$, where $N$ is the filter order) to the signal that is being multiplied by the cosine term as shown below.

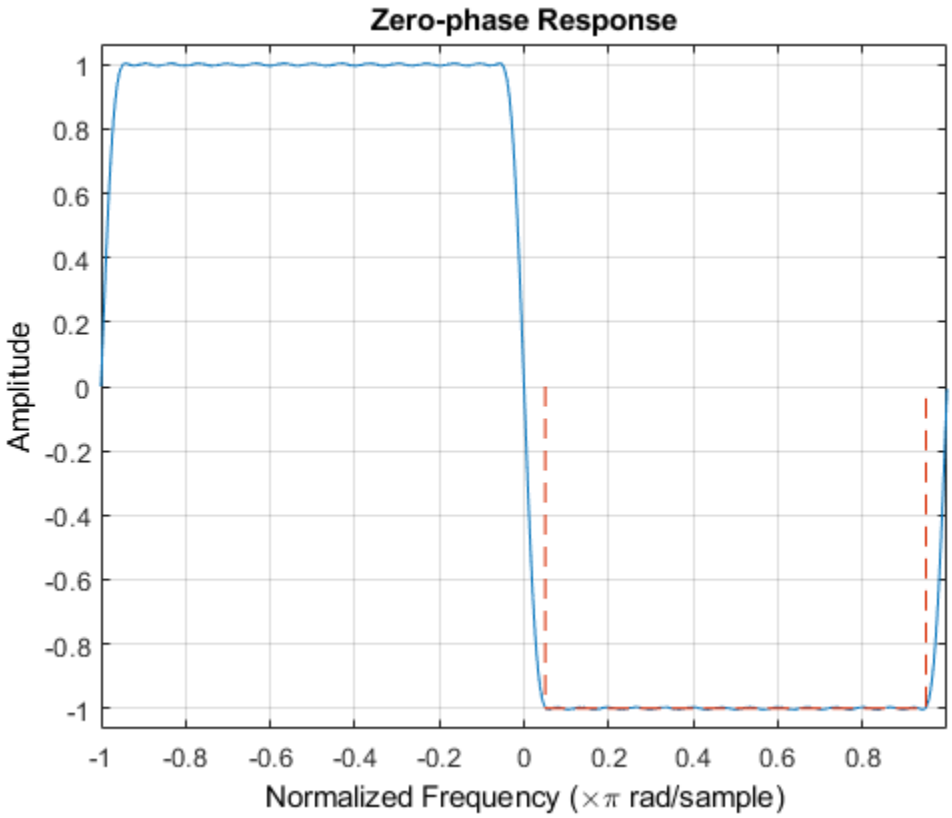



For the FIR Hilbert transformer we will use an odd length filter which is computationally more efficient than an even length filter. Albeit even length filters enjoy smaller passband errors. The savings in odd length filters is a result that these filters have several of the coefficients that are zero. Also, using an odd length filter will require a shift by an integer time delay, as opposed to a fractional time delay that is required by an even length filter. For an odd length filter, the magnitude response of a Hilbert Transformer is zero for $w = 0$ and $w = \pi$. For even length filers the magnitude response doesn't have to be 0 at $\pi$, therefore they have increased bandwidths. So for odd length filters the useful bandwidth is limited to

$$0 < w < \pi$$

Let's design the filter and plot its zero-phase response.

```
Hd = designfilt('hilbertfir','FilterOrder',60, ...
    'TransitionWidth',0.1,'DesignMethod','equiripple');

hfv = fvtool(Hd,'MagnitudeDisplay','Zero-phase', ...
    'FrequencyRange','[-pi, pi)');
hfv.Color = 'white';
```




To approximate the Hilbert Transform we'll filter the message signal with the filter.

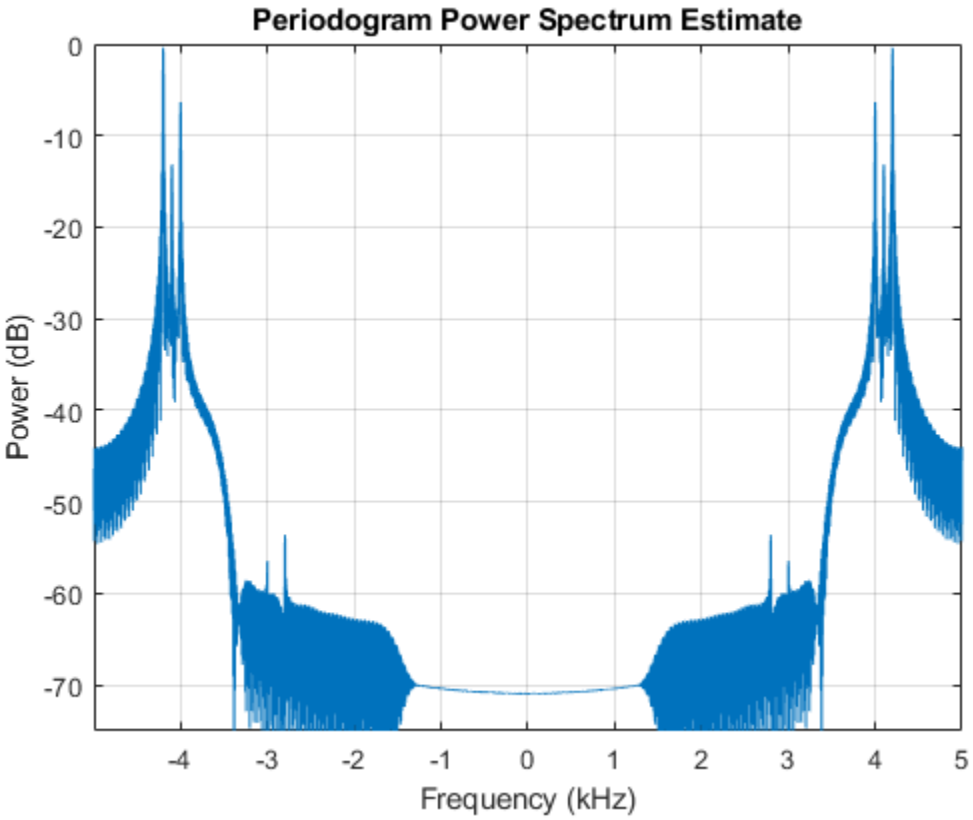
```
m_tilde = filter(Hd,m);
```

The upper sideband signal is then

```
G = filtord(Hd)/2;   % Filter delay
m_delayed = [zeros(1,G),m(1:end-G)];
f = m_delayed.*cos(2*pi*fo*t) - m_tilde.*sin(2*pi*fo*t);
```

and the spectrum is

```
periodogram(f,[],4096,Fs,'power','centered')
ylim([-75 0])
```




As seen in the plot above we successfully modulated the message signal (three tones) to the carrier frequency of 3.5 kHz and kept only the upper sideband.

**Summary**

As we have seen, by using an approximation to the Hilbert Transform we can produce analytic signals, which are useful in many signal applications that require spectral shifting. Specifically we have seen how an approximate Hilbert Transformer can be used to implement Single Sideband Modulation.

## See Also

`designfilt` | `hilbert` | `periodogram`

# Practical Introduction to Frequency-Domain Analysis

This example shows how to perform and interpret basic frequency-domain signal analysis. The example discusses the advantages of using frequency-domain versus time-domain representations of a signal and illustrates basic concepts using simulated and real data. The example answers basic questions such as: what is the meaning of the magnitude and phase of an FFT? Is my signal periodic? How do I measure power? Is there one, or more than one signal in this band?

Frequency-domain analysis is a tool of utmost importance in signal processing applications. Frequency-domain analysis is widely used in such areas as communications, geology, remote sensing, and image processing. While time-domain analysis shows how a signal changes over time, frequency-domain analysis shows how the signal's energy is distributed over a range of frequencies. A frequency-domain representation also includes information on the phase shift that must be applied to each frequency component in order to recover the original time signal with a combination of all the individual frequency components.

A signal can be converted between the time and frequency domains with a pair of mathematical operators called a transform. An example is the Fourier transform, which decomposes a function into the sum of a (potentially infinite) number of sine wave frequency components. The 'spectrum' of frequency components is the frequency domain representation of the signal. The inverse Fourier transform converts the frequency domain function back to a time function. The `fft` and `ifft` functions in MATLAB allow you to compute the Discrete Fourier transform (DFT) of a signal and the inverse of this transform respectively.

### Magnitude and Phase Information of the FFT

The frequency-domain representation of a signal carries information about the signal's magnitude and phase at each frequency. This is why the output of the FFT computation is complex. A complex number, $x$, has a real part, $x_r$, and an imaginary part, $x_i$, such that $x = x_r + ix_i$. The magnitude of $x$ is computed as $\sqrt{(x_r^2 + x_i^2)}$, and the phase of $x$ is computed as $\arctan(x_i/x_r)$. You can use MATLAB functions `abs` and `angle` to respectively get the magnitude and phase of any complex number.

Use an audio example to develop some insight on what information is carried by the magnitude and the phase of a signal. To do this, load an audio file containing 15 seconds of acoustic guitar music. The sample rate of the audio signal is 44.1 kHz.

```
Fs = 44100;
y = audioread('guitartune.wav');
```

Use `fft` to observe the frequency content of the signal.

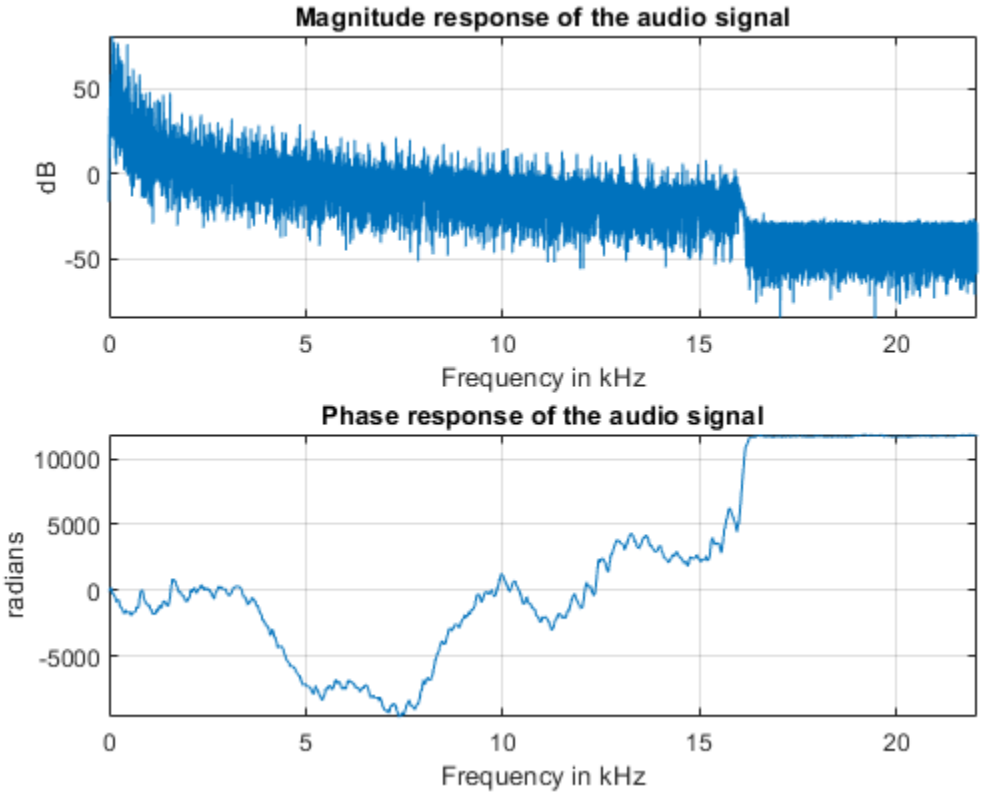
```
NFFT = length(y);
Y = fft(y,NFFT);
F = ((0:1/NFFT:1-1/NFFT)*Fs).';
```

The output of the FFT is a complex vector containing information about the frequency content of the signal. The magnitude tells you the strength of the frequency components relative to other components. The phase tells you how all the frequency components align in time.

Plot the magnitude and the phase components of the frequency spectrum of the signal. The magnitude is conveniently plotted in a logarithmic scale (dB). The phase is unwrapped using the `unwrap` function so that we can see a continuous function of frequency.

```
magnitudeY = abs(Y);        % Magnitude of the FFT
phaseY = unwrap(angle(Y));  % Phase of the FFT
```

```
helperFrequencyAnalysisPlot1(F,magnitudeY,phaseY,NFFT)
```




You can apply an inverse Fourier transform to the frequency domain vector, Y, to recover the time signal. The 'symmetric' flag tells `ifft` that you are dealing with a real-valued time signal so it will zero out the small imaginary components that appear on the inverse transform due to numerical inaccuracies in the computations. Notice that the original time signal, y, and the recovered signal, y1, are practically the same (the norm of their difference is on the order of 1e-14). The very small difference between the two is also due to the numerical inaccuracies mentioned above. Play and listen the un-transformed signal y1.

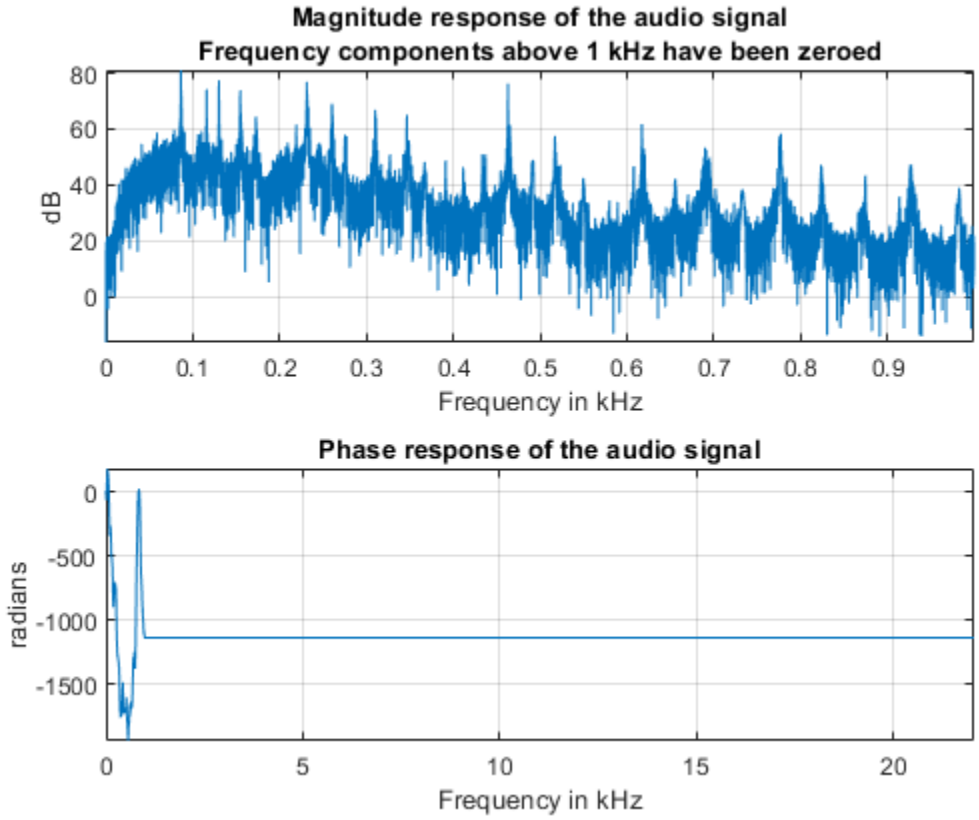
```
y1 = ifft(Y,NFFT,'symmetric');
norm(y-y1)
```

```
ans =

   3.9112e-14
```

```
hplayer = audioplayer(y1, Fs);
play(hplayer);
```

To see the effects of changing the magnitude response of the signal, remove frequency components above 1 kHz directly from the FFT output (by making the magnitudes equal to zero) and listen to the effect this has on the sound of the audio file. Removing high frequency components of a signal is referred to as lowpass filtering.

```
Ylp = Y;
Ylp(F>=1000 & F<=Fs-1000) = 0;

helperFrequencyAnalysisPlot1(F,abs(Ylp),unwrap(angle(Ylp)),NFFT,...
  'Frequency components above 1 kHz have been zeroed')
```




Get the filtered signal back into time domain using `ifft`.

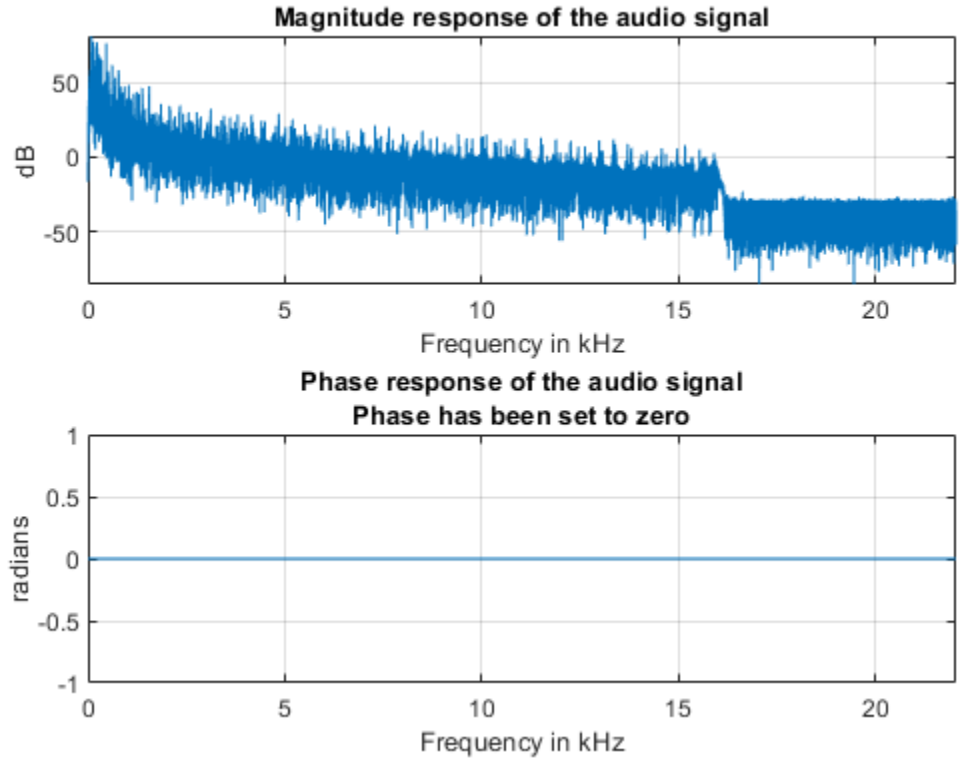
```
ylp = ifft(Ylp,'symmetric');
```

Play the signal. You can still hear the melody but it sounds like if you had covered your ears (you filter high frequency sounds when you do this). Even though guitars produce notes that are between 400 and 1 kHz, as you play a note on a string, the string also vibrates at multiples of the base frequency. These higher frequency components, referred to as harmonics, are what give the guitar its particular tone. When you remove them, you make the sound seem "opaque".

```
hplayer = audioplayer(ylp, Fs);
play(hplayer);
```

The phase of a signal has important information about when in time the notes of the song appear. To illustrate the importance of phase on the audio signal, remove the phase information completely by taking the magnitude of each frequency component. Note that by doing this you keep the magnitude response unchanged.

```
% Take the magnitude of each FFT component of the signal
Yzp = abs(Y);
helperFrequencyAnalysisPlot1(F,abs(Yzp),unwrap(angle(Yzp)),NFFT,[],...
  'Phase has been set to zero')
```




Get the signal back in the time domain and play the audio. You cannot recognize the original sound at all. The magnitude response is the same, no frequency components have been removed this time, but the order of the notes has disappeared completely. The signal now consists of a group of sinusoids all aligned at time equal to zero. In general, phase distortions caused by filtering can damage a signal to the point of rendering it unrecognizable.

```
yzp = ifft(Yzp,'symmetric');
hplayer = audioplayer(yzp, Fs);
play(hplayer);
```

### Finding Signal Periodicities

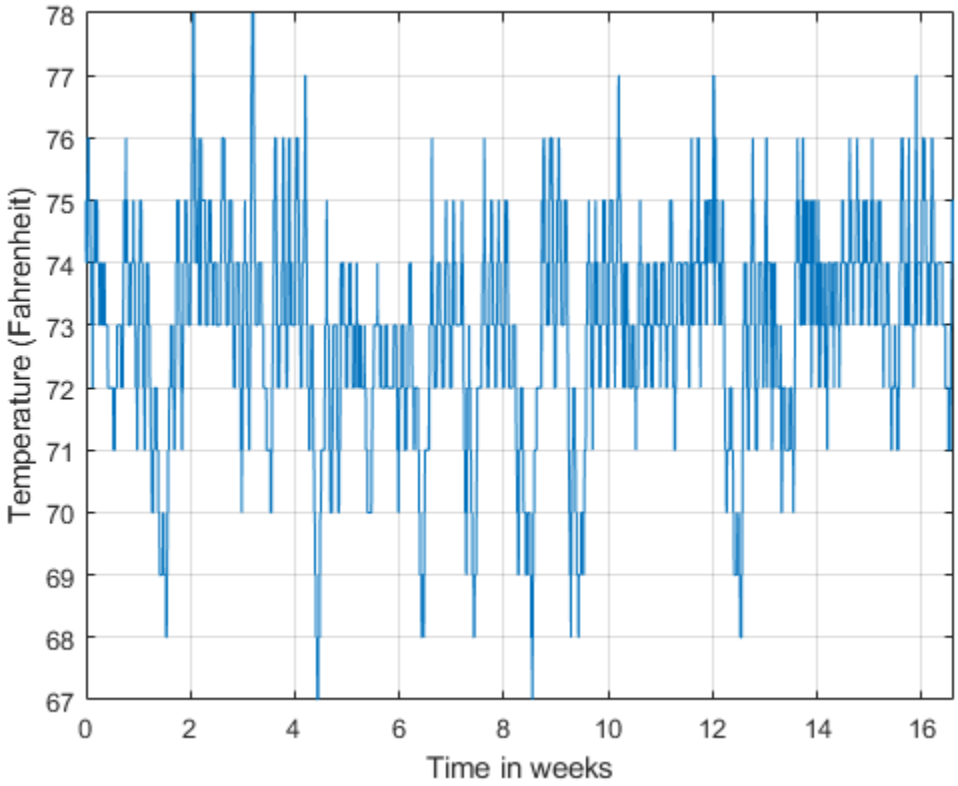
The frequency domain representation of a signal allows you to observe several characteristics of the signal that are either not easy to see, or not visible at all when you look at the signal in the time domain. For instance, frequency-domain analysis becomes useful when you are looking for cyclic behavior of a signal.

#### Analyzing Cyclic Behavior of the Temperature in an Office Building

Consider a set of temperature measurements in an office building during the winter season. Measurements were taken every 30 minutes for about 16.5 weeks. Look at the time domain data with the time axis scaled to weeks. Could there be any periodic behavior on this data?

```
load officetemp.mat
Fs = 1/(60*30);   % Sample rate is 1 sample every 30 minutes
t = (0:length(temp)-1)/Fs;
```

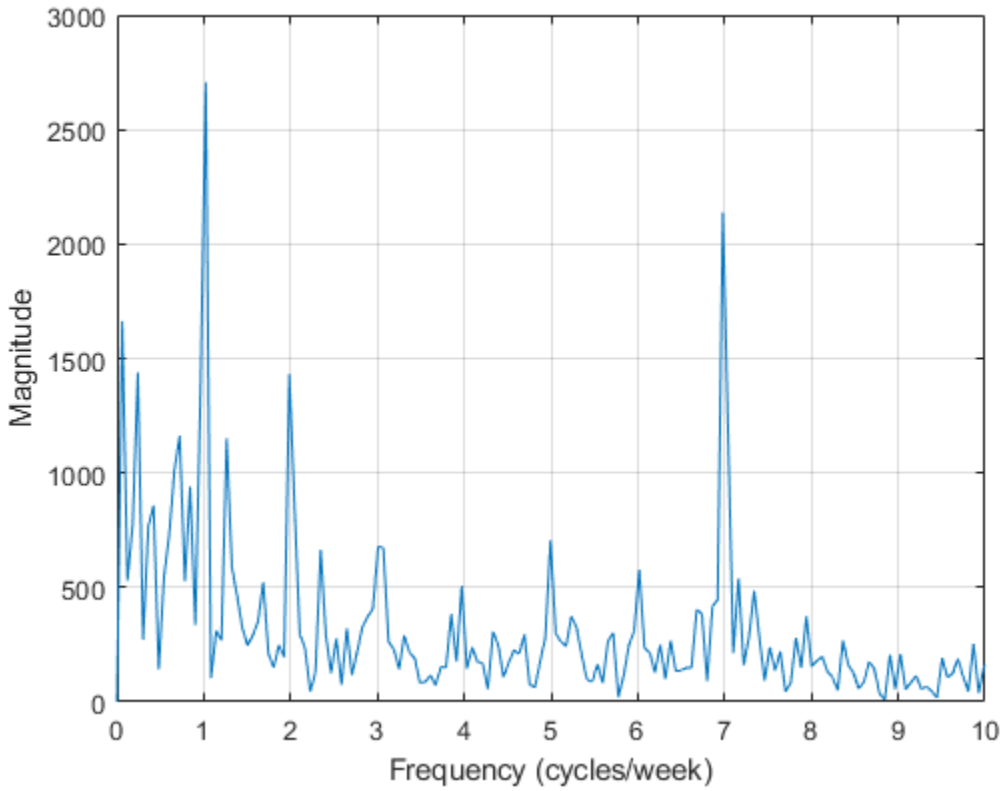
```
helperFrequencyAnalysisPlot2(t/(60*60*24*7),temp,...
  'Time in weeks','Temperature (Fahrenheit)')
```




It is almost impossible to know if there is any cyclic behavior on the office temperatures by looking at the time-domain signal. However, the cyclic behavior of the temperature becomes evident if we look at its frequency-domain representation.

Obtain the frequency-domain representation of the signal. If you plot the magnitude of the FFT output with a frequency axis scaled to cycles/week, you can see that there are two spectral lines that are clearly larger than any other frequency component. One spectral line lies at 1 cycle/week, the other one lies at 7 cycles/week. This makes sense given that the data comes from a temperature-controlled building on a 7 day calendar. The first spectral line indicates that building temperatures follow a weekly cycle with lower temperatures on the weekends and higher temperatures during the week. The second line indicates that there is also a daily cycle with lower temperatures during the night and higher temperatures during the day.

```
NFFT = length(temp);              % Number of FFT points
F = (0 : 1/NFFT : 1/2-1/NFFT)*Fs; % Frequency vector

TEMP = fft(temp,NFFT);
TEMP(1) = 0; % remove the DC component for better visualization

helperFrequencyAnalysisPlot2(F*60*60*24*7,abs(TEMP(1:NFFT/2)),...
  'Frequency (cycles/week)','Magnitude',[],[],[0 10])
```




### Measuring Power

The `periodogram` function computes the signal's FFT and normalizes the output to obtain a power spectral density, PSD, or a power spectrum from which you can measure power. The PSD describes how the power of a time signal is distributed with frequency, it has units of watts/Hz. You compute the power spectrum by integrating each point of the PSD over the frequency interval at which that point is defined (i.e. over the resolution bandwidth of the PSD). The units of the power spectrum are watts. You can read power values directly from the power spectrum without having to integrate over an interval. Note that the PSD and power spectrum are real, so they do not contain any phase information.

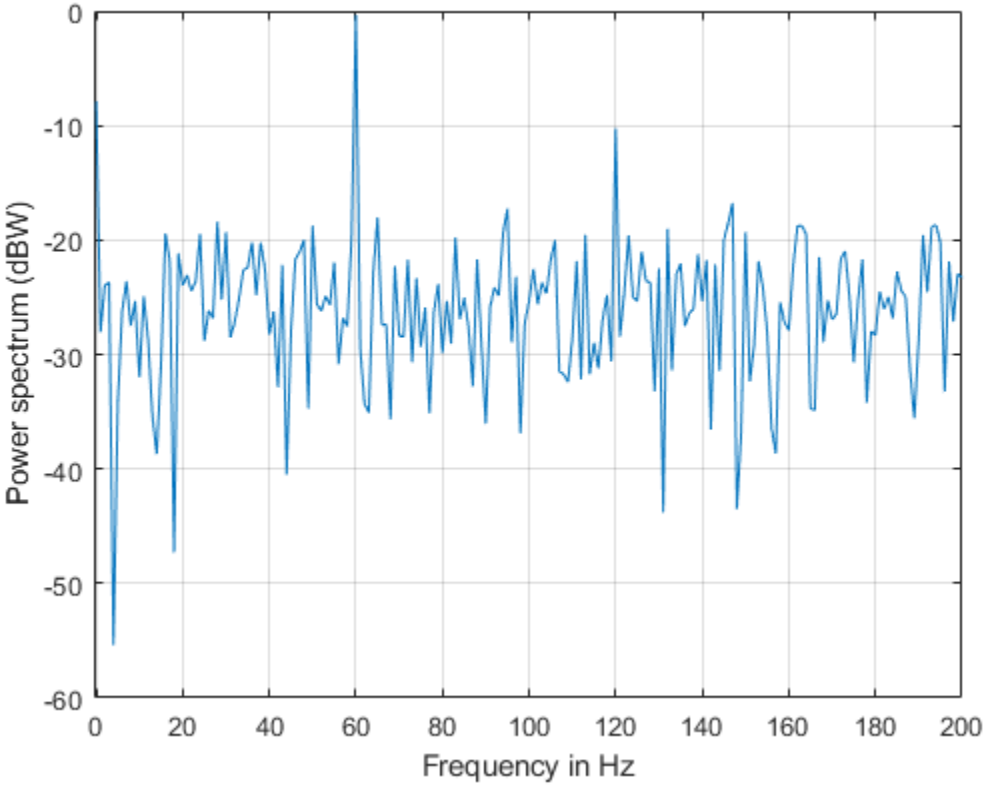
### Measuring Harmonics at the Output of a Non-Linear Power Amplifier

Load the data measured at the output of a power amplifier that has third order distortion of the form $v_o = v_i + 0.75v_i^2 + 0.5v_i^3$, where $v_o$ is the output voltage and $v_i$ is the input voltage. The data was captured with a sample rate of 3.6 kHz. The input $v_i$ consists of a 60 Hz sinusoid with unity amplitude. Due to the nature of the non-linear distortion, you should expect the amplifier output signal to contain a DC component, a 60 Hz component, and second and third harmonics at 120 and 180 Hz.

Load 3600 samples of the amplifier output, compute the power spectrum, and plot the result in a logarithmic scale (decibels-watts or dBW).

```
load ampoutput1.mat
Fs = 3600;
NFFT = length(y);
```

```
% Power spectrum is computed when you pass a 'power' flag input
[P,F] = periodogram(y,[],NFFT,Fs,'power');

helperFrequencyAnalysisPlot2(F,10*log10(P),'Frequency in Hz',...
  'Power spectrum (dBW)',[],[],[-0.5 200])
```




The plot of the power spectrum shows three of the four expected peaks at DC, 60, and 120 Hz. It also shows several more spurious peaks that must be caused by noise in the signal. Note that the 180 Hz harmonic is completely buried in the noise.

Measure the power of the visible expected peaks:

```
PdBW = 10*log10(P);
power_at_DC_dBW = PdBW(F==0)   % dBW

[peakPowers_dBW, peakFreqIdx] = findpeaks(PdBW,'minpeakheight',-11);
peakFreqs_Hz = F(peakFreqIdx)
peakPowers_dBW
```

```
power_at_DC_dBW =

   -7.8873


peakFreqs_Hz =
```

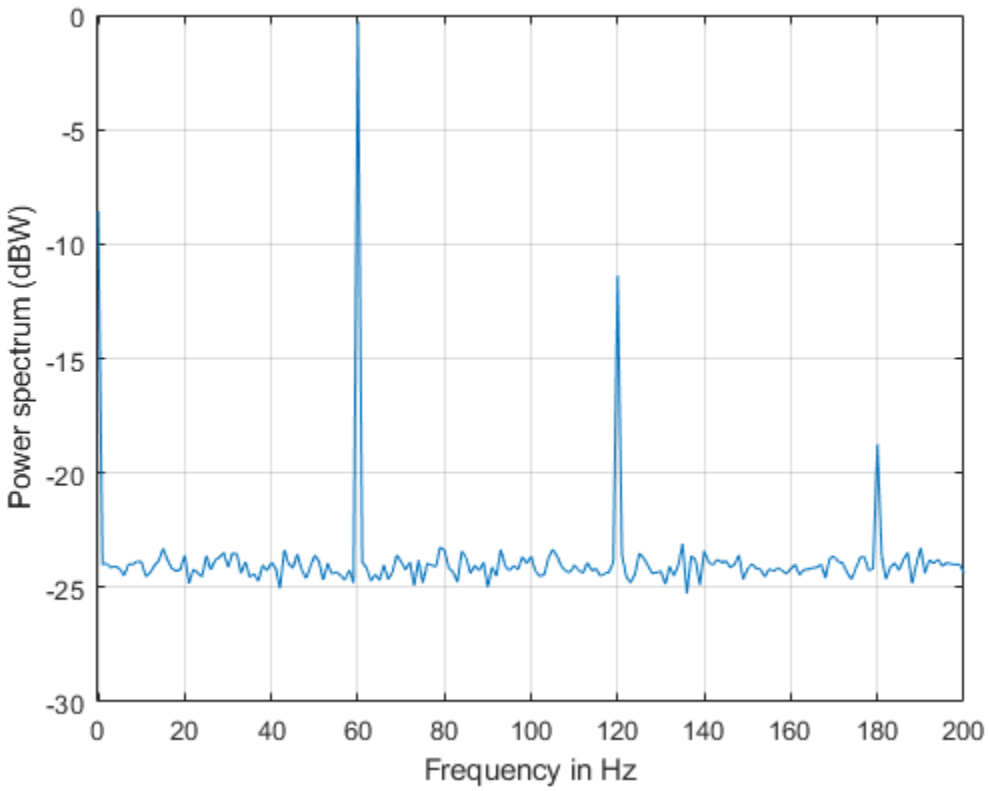
```
    60
   120


peakPowers_dBW =

   -0.3175
  -10.2547
```

### Improving Power Measurements for Noisy Signals

As seen on the plot above, the periodogram shows several frequency peaks that are not related to the signal of interest. The spectrum looks very noisy. The reason for this is that you only analyzed one short realization of the noisy signal. Repeating the experiment several times and averaging would remove the spurious spectral peaks and yield more accurate power measurements. You can achieve this averaging using the `pwelch` function. This function will take a large data vector, break it into smaller segments of a specified length, compute as many periodograms as there are segments, and average them. As the number of available segments increases, the `pwelch` function will yield a smoother power spectrum (less variance) with power values closer to the expected values.

Load a larger observation consisting of 500e3 points of the amplifier output. Keep the number of points used to perform the FFTs as 3600 so that floor(500e3/3600) = 138 FFTs are averaged to obtain the power spectrum.

```
load ampoutput2.mat
SegmentLength = NFFT;

% Power spectrum is computed when you pass a 'power' flag input
[P,F] = pwelch(y,ones(SegmentLength,1),0,NFFT,Fs,'power');

helperFrequencyAnalysisPlot2(F,10*log10(P),'Frequency in Hz',...
  'Power spectrum (dBW)',[],[],[-0.5 200])
```




As seen on the plot, `pwelch` effectively removes all the spurious frequency peaks caused by noise. The spectral component at 180 Hz that was buried in noise is now visible. Averaging removes variance from the spectrum and this effectively yields more accurate power measurements.

**Measuring Total Average Power and Power Over a Frequency Band**

Measuring the total average power of a time-domain signal is an easy and common task. For the amplifier output signal, y, the total average power is computed in the time domain as:

```
pwr = sum(y.^2)/length(y) % in watts
```

```
pwr =

    8.1697
```

In the frequency-domain, the total average power is computed as the sum of the power of all the frequency components of the signal. The value of pwr1 consists of the sum of all the frequency components available in the power spectrum of the signal. The value agrees with the value of pwr computed above using the time domain signal:

```
pwr1 = sum(P) % in watts
```

```
pwr1 =
```

```
    8.1698
```

But what if you wanted to measure the total power available over a band of frequencies? You can use the `bandpower` function to compute the power over any desired frequency band. You can pass the time-domain signal directly as an input to this function to obtain the power over a specified band. In this case, the function will estimate the power spectrum with the periodogram method.

Compute the power over the 50 Hz to 70 Hz band. The result will include the 60 Hz power plus the noise power over the band of interest:

```
pwr_band = bandpower(y,Fs,[50 70]);
pwr_band_dBW = 10*log10(pwr_band) % dBW
```

```
pwr_band_dBW =

    0.0341
```

If you want to control the computation of the power spectrum used to measure the power in a band, you can pass a PSD vector to the `bandpower` function. For instance, you can use the `pwelch` function as you did before to compute the PSD and ensure averaging of the noise effects:

```
% Power spectral density is computed when you specify the 'psd' option
[PSD,F]  = pwelch(y,ones(SegmentLength,1),0,NFFT,Fs,'psd');
pwr_band1 = bandpower(PSD,F,[50 70],'psd');
pwr_band_dBW1 = 10*log10(pwr_band1) % dBW
```

```
pwr_band_dBW1 =

    0.0798
```

### Finding Spectral Components

A signal might be composed of one or more frequency components. The ability to observe all the spectral components depends on the frequency resolution of your analysis. The frequency resolution or resolution bandwidth of the power spectrum is defined as R = Fs/N, where N is the length of the signal observation. Only spectral components separated by a frequency larger than the frequency resolution will be resolved.

#### Analyzing a Building's Earthquake Vibration Control System

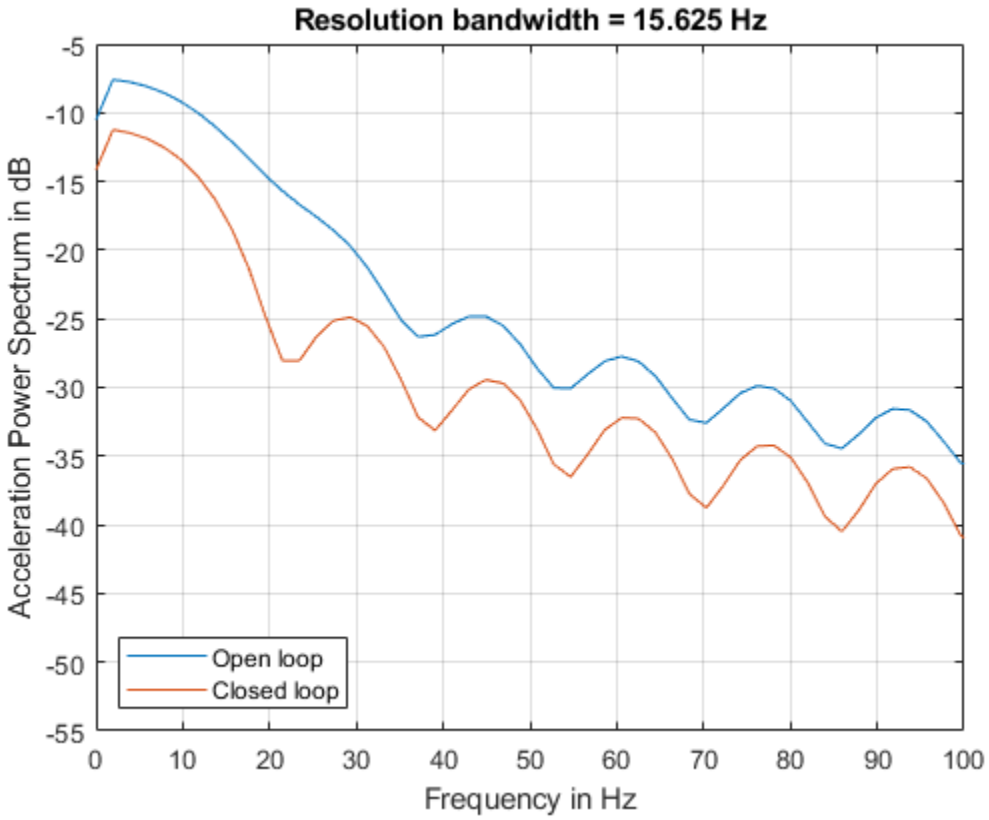
Active Mass Driver (AMD) control systems are used to reduce vibration in a building under an earthquake. An active mass driver is placed on the top floor of the building and, based on displacement and acceleration measurements of the building floors, a control system sends signals to the driver so that the mass moves to attenuate ground disturbances. Acceleration measurements were recorded on the first floor of a three story test structure under earthquake conditions. Measurements were taken without the active mass driver control system (open loop condition), and with the active control system (closed loop condition).

Load the acceleration data and compute the power spectrum for the acceleration of the first floor. The length of the data vectors is 10e3 and the sample rate is 1 kHz. Use `pwelch` with segments of length 64 data points to obtain floor(10e3/64) = 156 FFT averages and a resolution bandwidth of

Fs/64 = 15.625 Hz. As was shown before, averaging reduces noise effects and yields more accurate power measurements. Use 512 FFT points. Using NFFT > N effectively interpolates frequency points rendering a more detailed spectrum plot (this is achieved by appending NFFT-N zeros at the end of the time signal and taking the NFFT-point FFT of the zero padded vector).

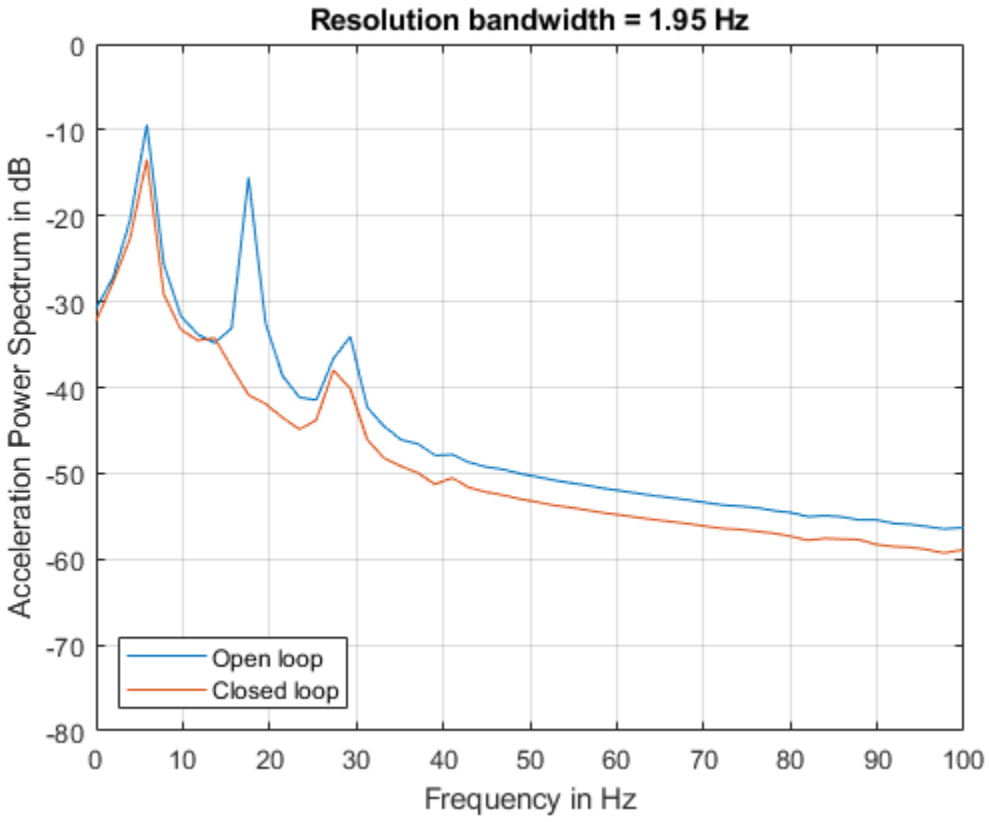
The open loop and close loop acceleration power spectra show that when the control system is active, the acceleration power spectrum decreases between 4 and 11 dB. The maximum attenuation occurs at about 23.44 kHz. An 11 dB reduction means that the vibration power is reduced by a factor of 12.6. The total power is reduced from 0.1670 to 0.059 watts, a factor of 2.83.

```
load quakevibration.mat

Fs = 1e3;                % sample rate
NFFT = 512;              % number of FFT points
segmentLength = 64;      % segment length

% open loop acceleration power spectrum
[P1_OL,F] = pwelch(gfloor1OL,ones(segmentLength,1),0,NFFT,Fs,'power');

% closed loop acceleration power spectrum
P1_CL     = pwelch(gfloor1CL,ones(segmentLength,1),0,NFFT,Fs,'power');

helperFrequencyAnalysisPlot2(F,10*log10([(P1_OL) (P1_CL)]),...
  'Frequency in Hz','Acceleration Power Spectrum in dB',...
  'Resolution bandwidth = 15.625 Hz',{'Open loop', 'Closed loop'},[0 100])
```



You are analyzing vibration data and you know that vibrations have a cyclic behavior. Then how is it that the spectrum plots shown above do not contain any sharp spectral lines typical of cyclic behavior? Maybe you are missing those lines because they are not resolvable with the resolution obtained with 64 point segment lengths? Increase the frequency resolution to see if there are spectral lines that were not resolvable before. Do this by increasing the data segment length used in the `pwelch` function to 512 points. This yields a new resolution of Fs/512 = 1.9531 Hz. In this case, the number of FFT averages is reduced to floor(10e3/512) = 19. Clearly, there is a trade-off between number of averages and frequency resolution when using `pwelch`. Keep the number of FFT points equal to 512.

```
NFFT = 512;             % number of FFT points
segmentLength = 512;    % segment length

[P1_OL,F] = pwelch(gfloor1OL,ones(segmentLength,1),0,NFFT,Fs,'power');
P1_CL     = pwelch(gfloor1CL,ones(segmentLength,1),0,NFFT,Fs,'power');

helperFrequencyAnalysisPlot2(F,10*log10([(P1_OL) (P1_CL)]),...
  'Frequency in Hz','Acceleration Power Spectrum in dB',...
  'Resolution bandwidth = 1.95 Hz',{'Open loop', 'Closed loop'},[0 100])
```




Notice how the increase in frequency resolution allows you to observe three peaks on the open loop spectrum and two on the close loop spectrum. These peaks were not resolvable before. The separation between the peaks on the open loop spectrum is about 11 Hz which is smaller than the frequency resolution obtained with segments of length 64 but larger than the resolution obtained with segments of length 512. The cyclic behavior of the vibrations is now visible. The main vibration frequency is at 5.86 Hz, and the equispaced frequency peaks suggest that they are harmonically

related. While it has already been observed that the control system reduces the overall power of the vibrations, the higher resolution spectra shows that another effect of the control system is to notch the harmonic component at 17.58 Hz. So the control system not only reduces the vibration but also brings it closer to a sinusoid.

It is important to note that frequency resolution is determined by the number of signal points, not by the number of FFT points. Increasing the number of FFT points interpolates the frequency data to give you more details on the spectrum but it does not improve resolution.

### Conclusions

In this example you learned how to perform frequency-domain analysis of a signal using the `fft`, `ifft`, `periodogram`, `pwelch`, and `bandpower` functions. You understood the complex nature of the FFT and what is the information contained in the magnitude and the phase of the frequency spectrum. You saw the advantages of using frequency domain data when analyzing the periodicity of a signal. You learned how compute the total power or power over a particular band of frequencies of a noisy signal. You understood how increasing the frequency resolution of the spectrum allows you to observe closely spaced frequency components and you learned about the tradeoff between frequency resolution and spectral averaging.

### Further Reading

For more information on frequency domain analysis see the Signal Processing Toolbox.

Reference: J.G. Proakis and D. G. Manolakis, "Digital Signal Processing. Principles, Algorithms, and Applications", Prentice Hall, 1996.

### Appendix

The following helper functions are used in this example.

- helperFrequencyAnalysisPlot1.m
- helperFrequencyAnalysisPlot2.m

## See Also

`fft` | `periodogram` | `pspectrum` | `pwelch`

# Practical Introduction to Time-Frequency Analysis

This example shows how to perform and interpret basic time-frequency signal analysis. In practical applications, many signals are nonstationary. This means that their frequency-domain representation (their spectrum) changes over time. The example discusses the advantages of using time-frequency techniques over frequency-domain or time-domain representations of a signal. It answers basic questions, such as: When is a particular frequency component present in my signal? How do I increase time or frequency resolution? How can I sharpen the spectrum of a component or extract a particular mode? How do I measure power in a time-frequency representation? How do I visualize the time-frequency information of my signal? How do I find intermittent interference within the frequency content of the signal of interest?

### Using Time-Frequency Analysis to Identify Numbers in a DTMF Signal

You can divide almost any time-varying signal into time intervals short enough that the signal is essentially stationary in each section. Time-frequency analysis is most commonly performed by segmenting a signal into those short periods and estimating the spectrum over sliding windows. The `pspectrum` function used with the `'spectrogram'` option computes an FFT-based spectral estimate over each sliding window and lets you visualize how the frequency content of the signal changes over time.

Consider the signaling system of a digital phone dial. The signals produced by such a system are known as dual-tone multi-frequency (DTMF) signals. The sound generated by each dialed number consists of the sum of two sinusoids − or tones − with frequencies taken from two mutually exclusive groups. Each pair of tones contains one frequency of the low group (697 Hz, 770 Hz, 852 Hz, or 941 Hz) and one frequency of the high group (1209 Hz, 1336 Hz, or 1477Hz) and represents a unique symbol. The following are the frequencies allocated to the buttons of a telephone pad:

| | 1209 Hz | 1336 Hz | 1477 Hz |
|---|---|---|---|
| 697 Hz | 1 | 2 | 3 |
| 770 Hz | 4 | 5 | 6 |
| 852 Hz | 7 | 8 | 9 |
| 941 Hz | * | 0 | # |

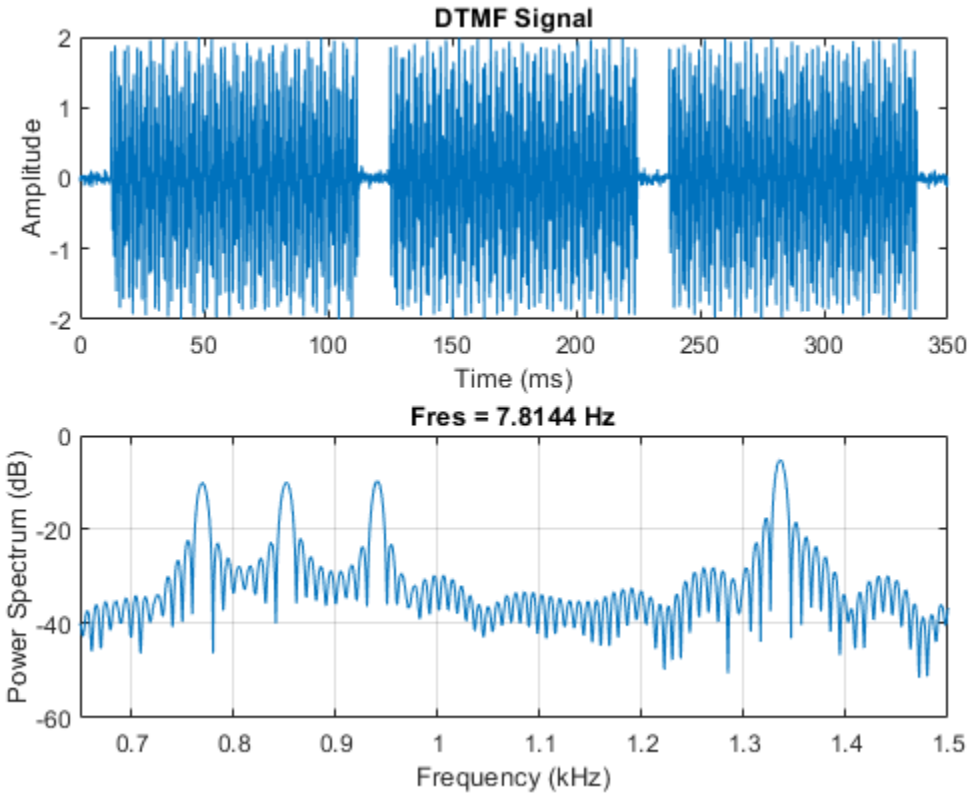
Generate a DTMF signal and listen to it.

```
[tones, Fs] = helperDTMFToneGenerator();
p = audioplayer(tones,Fs,16);
play(p)
```

Listening to the signal, you can tell that a three-digit number was dialed. However, you cannot tell which number it was. Next, visualize the signal in time and in frequency domain over the 650 to 1500 Hz band. Set the `'Leakage'` parameter of the `pspectrum` function to 1 to use a rectangular window and improve frequency resolution.

```
N = numel(tones);
t = (0:N-1)/Fs;
```

```
subplot(2,1,1)
plot(1e3*t,tones)
xlabel('Time (ms)')
ylabel('Amplitude')
title('DTMF Signal')
subplot(2,1,2)
pspectrum(tones,Fs,'Leakage',1,'FrequencyLimits',[650, 1500])
```

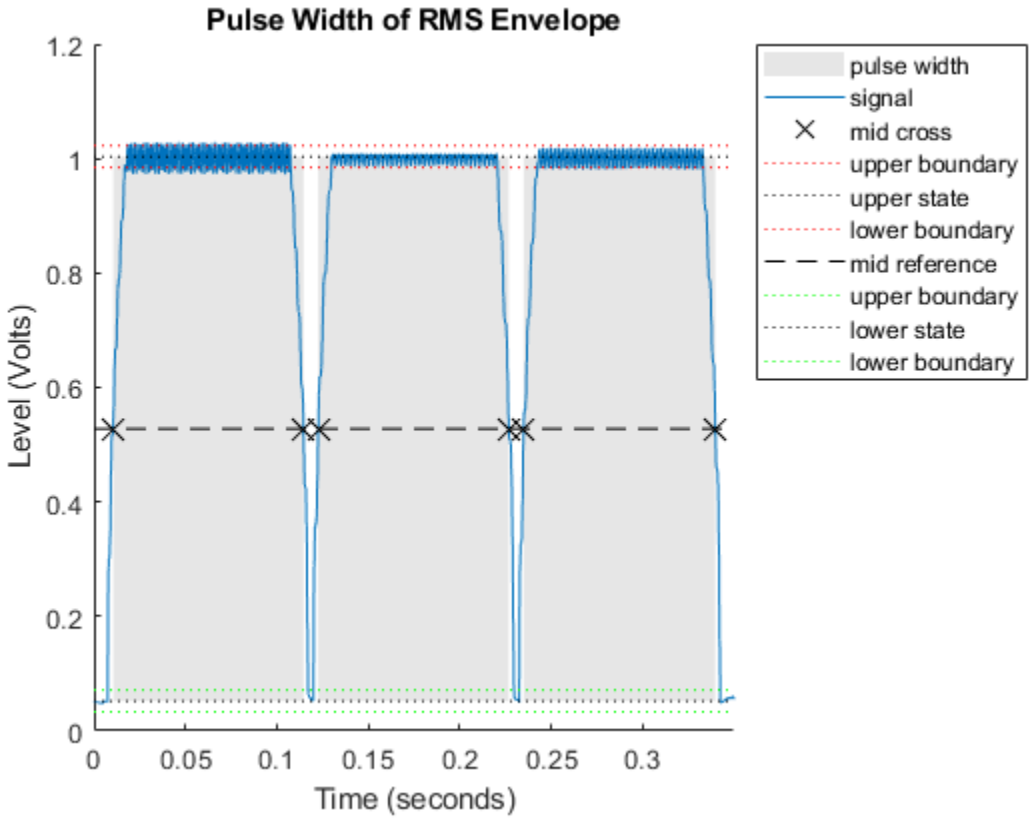



The time-domain plot of the signal confirms the presence of three bursts of energy, corresponding to three pushed buttons. To measure the length of the burst, you can take the pulse width of the RMS envelope.

```
env = envelope(tones,80,'rms');
pulsewidth(env,Fs)
```

```
ans = 3×1

    0.1041
    0.1042
    0.1047
```

```
title('Pulse Width of RMS Envelope')
```




Here you can see three pulses, each one approximately 100 milliseconds long. However, you cannot tell which numbers were dialed. A frequency-domain plot helps you figure this out because it shows the frequencies present in the signal.

Locate the frequency peaks by estimating the mean frequency in four different frequency bands.

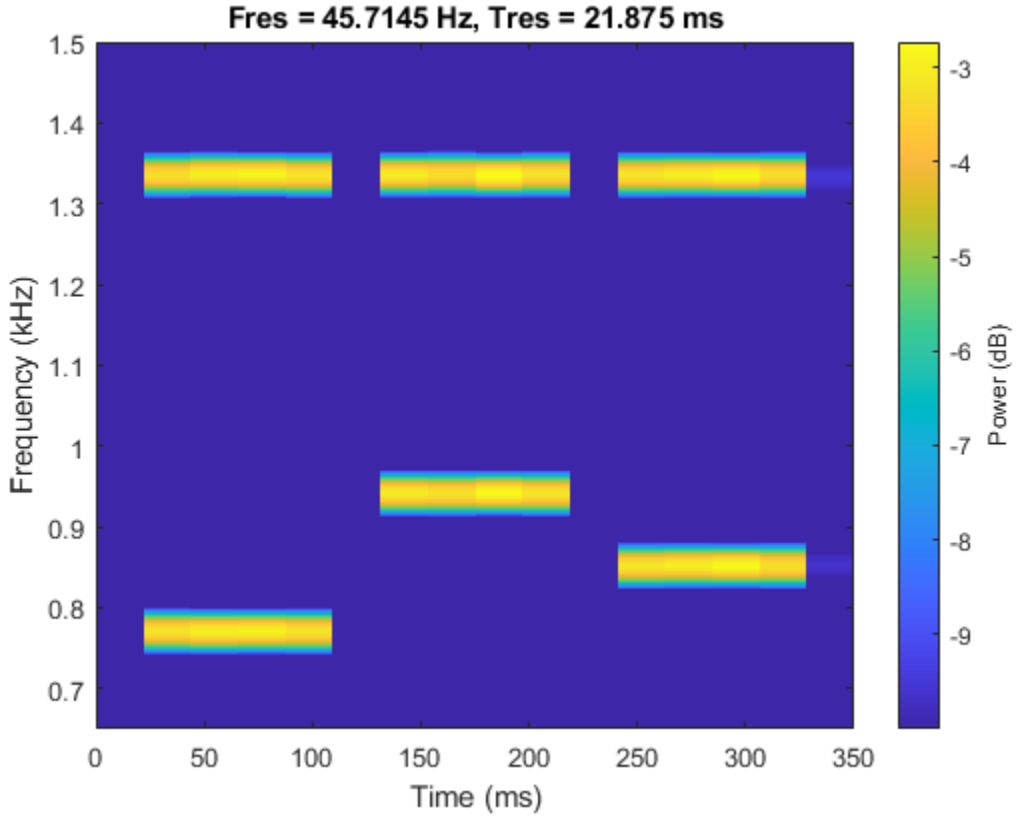
```
f = [meanfreq(tones,Fs,[700 800]), ...
     meanfreq(tones,Fs,[800 900]), ...
     meanfreq(tones,Fs,[900 1000]), ...
     meanfreq(tones,Fs,[1300 1400])];
round(f)
```

```
ans = 1×4

         770         852         941        1336
```

By matching the estimated frequencies to the diagram of the telephone pad, you can say that the dialed buttons were '5', '8', and '0'. However, the frequency-domain plot does not provide any type of time information that would allow you to figure out the order in which they were dialed. The combination could be '580','508','805','850', '085', or '058'. To solve this puzzle, use the pspectrum function to compute the spectrogram and observe how the frequency content of the signal varies with time.

Compute the spectrogram over the 650 to 1500 Hz band and remove content below the −10 dB power level to visualize only the main frequency components. To see the tone durations and their locations in time use 0% overlap.

```
pspectrum(tones,Fs,'spectrogram','Leakage',1,'OverlapPercent',0, ...
    'MinThreshold',-10,'FrequencyLimits',[650, 1500]);
```




The colors of the spectrogram encode frequency power levels. Yellow colors indicate frequency content with higher power; blue colors indicate frequency content with very low power. A strong yellow horizontal line indicates the existence of a tone at a particular frequency. The plot clearly shows the presence of a 1336 Hz tone in all three dialed digits, telling you that they are all on the second column of the keypad. From the plot you can see that the lowest frequency, 770 Hz, was dialed first. The highest frequency, 941 Hz, was next. The middle frequency, 852 Hz, came last. Hence, the dialed number was 508.

**Trading Off Time and Frequency Resolution to Get the Best Representation of Your Signal**

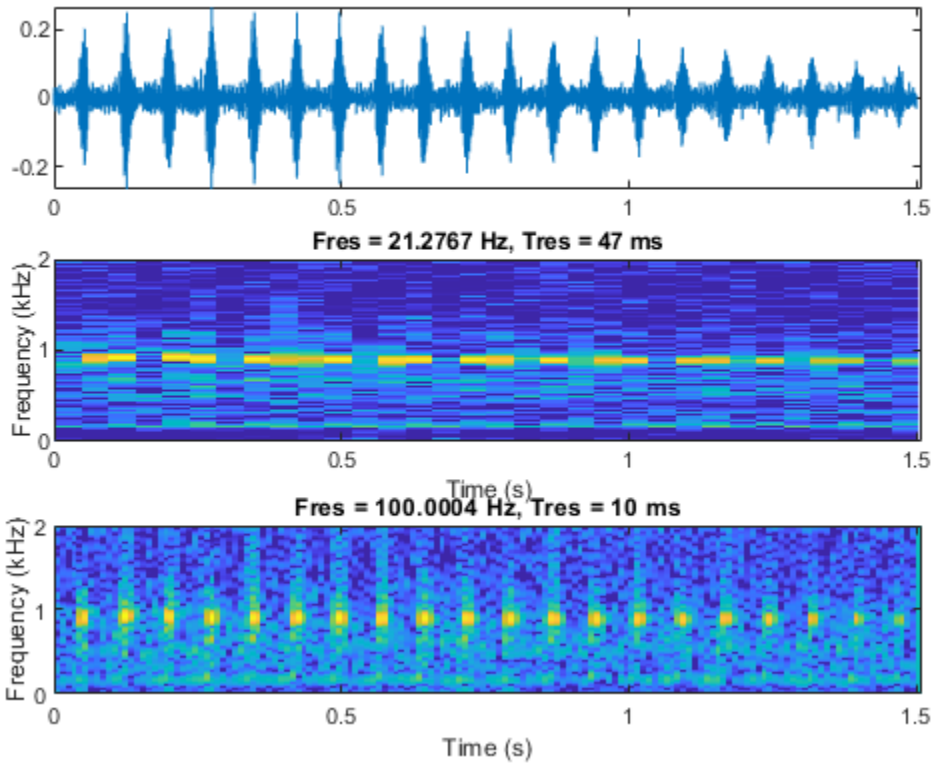
The `pspectrum` function divides a signal into segments. Longer segments provide better frequency resolution; shorter segments provide better time resolution. The segment lengths can be controlled using the `'FrequencyResolution'` and `'TimeResolution'` parameters. When no frequency resolution or time resolution values are specified, `pspectrum` attempts to find a good balance between time and frequency resolutions based on the input signal length.

Consider the following signal, sampled at 4 kHz, that consists of the trill portion of a Pacific blue whale song:

```
load whaleTrill
p = audioplayer(whaleTrill,Fs,16);
play(p)
```

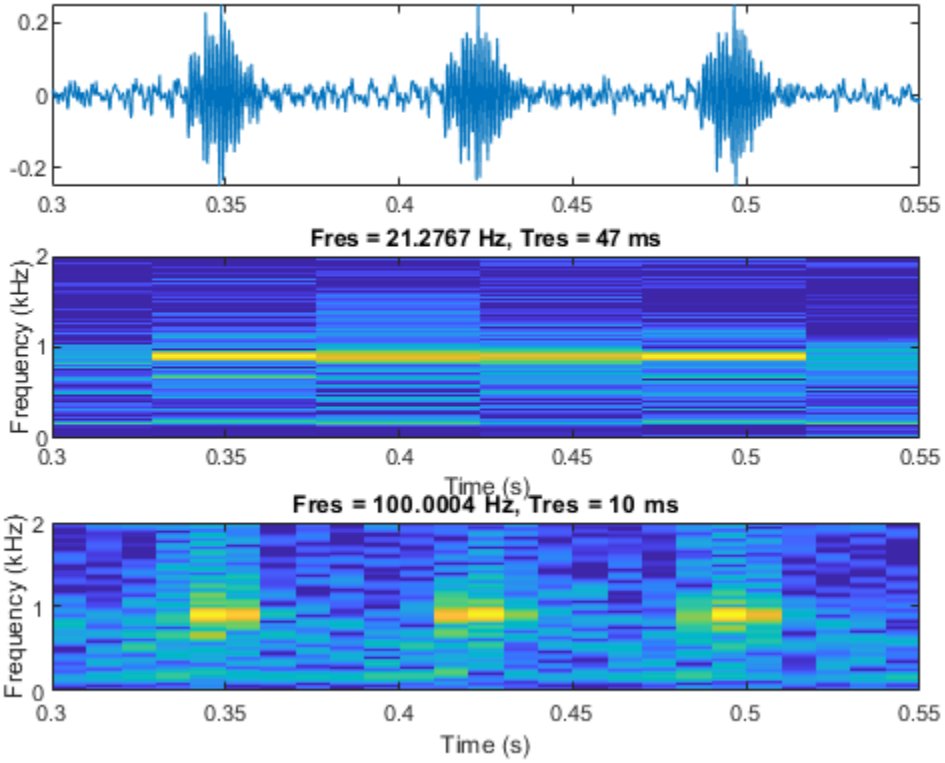
The trill signal consists of a train of tonal pulses. Look at the time signal and the spectrogram obtained by `pspectrum` when no resolution is specified and when time resolution is set to 10 milliseconds. Set the `'Leakage'` parameter to 1 to use rectangular windows. Since we want to localize the time position of the pulses, set overlap percent to 0. Finally, use a `'MinThreshold'` of −60 dB to remove background noise from the spectrogram view.

```
t = (0:length(whaleTrill)-1)/Fs;
figure
ax1 = subplot(3,1,1);
plot(t,whaleTrill)
ax2 = subplot(3,1,2);
pspectrum(whaleTrill,Fs,'spectrogram','OverlapPercent',0, ...
    'Leakage',1,'MinThreshold',-60)
colorbar(ax2,'off')
ax3 = subplot(3,1,3);
pspectrum(whaleTrill,Fs,'spectrogram','OverlapPercent',0, ...
    'Leakage',1,'MinThreshold',-60,'TimeResolution', 10e-3)
colorbar(ax3,'off')
linkaxes([ax1,ax2,ax3],'x')
```



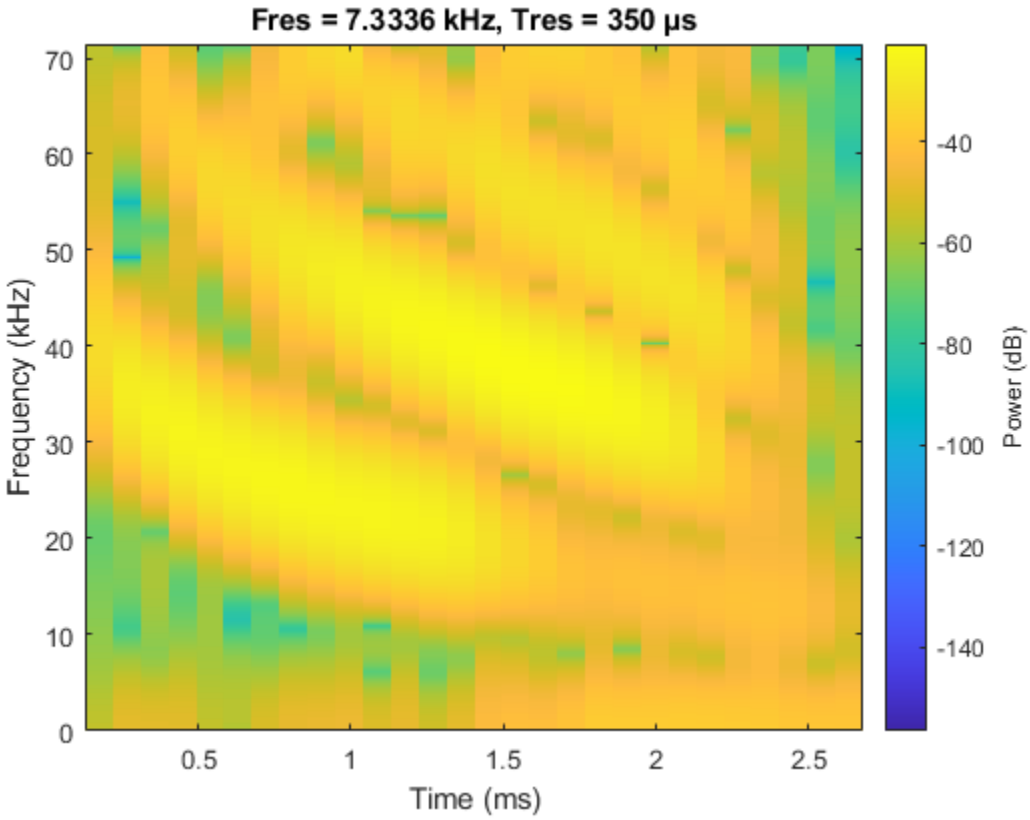

The 47 milliseconds time resolution chosen by `pspectrum` is not small enough to localize all the trill pulses in the spectrogram. On the other hand, a time resolution of 10 milliseconds is enough to localize each trill pulse in time. This becomes even clearer if we zoom into a few pulses:

```
xlim([0.3 0.55])
```



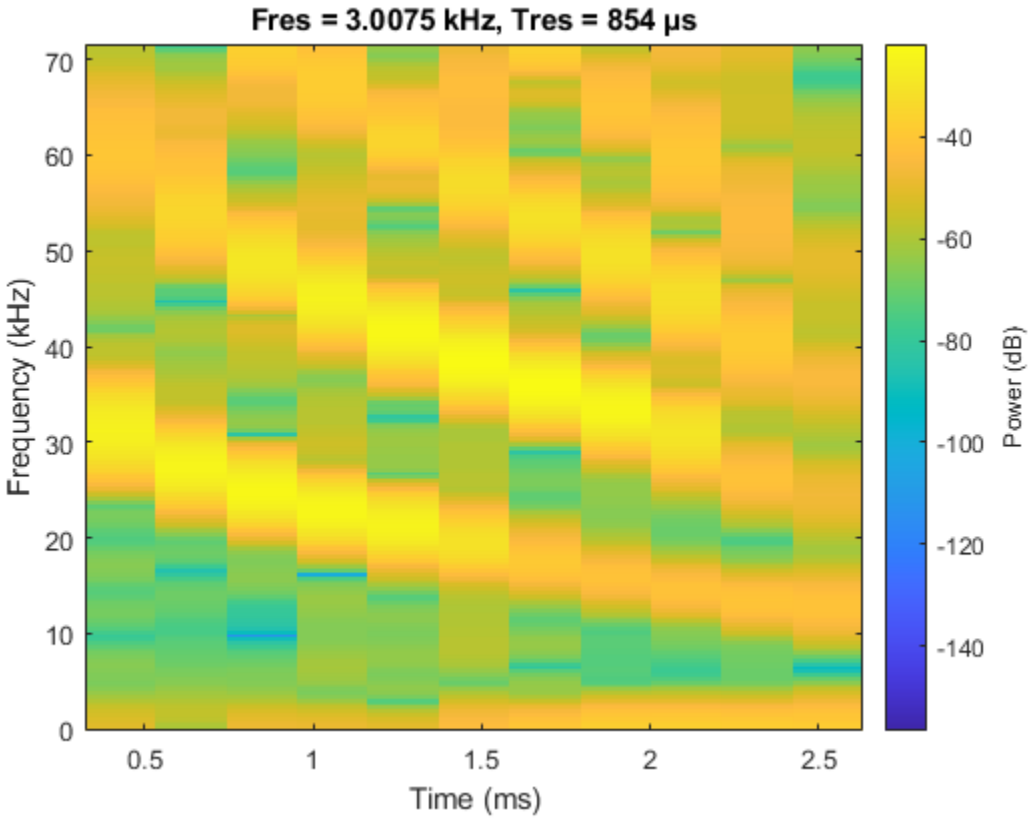

Now load a signal that consists of an echolocation pulse emitted by a big brown bat (Eptesicus fuscus). The signal is measured with a sampling interval of 7 microseconds. Analyze the spectrogram of the signal.

```
load batsignal
Fs = 1/DT;
figure
pspectrum(batsignal,Fs,'spectrogram')
```



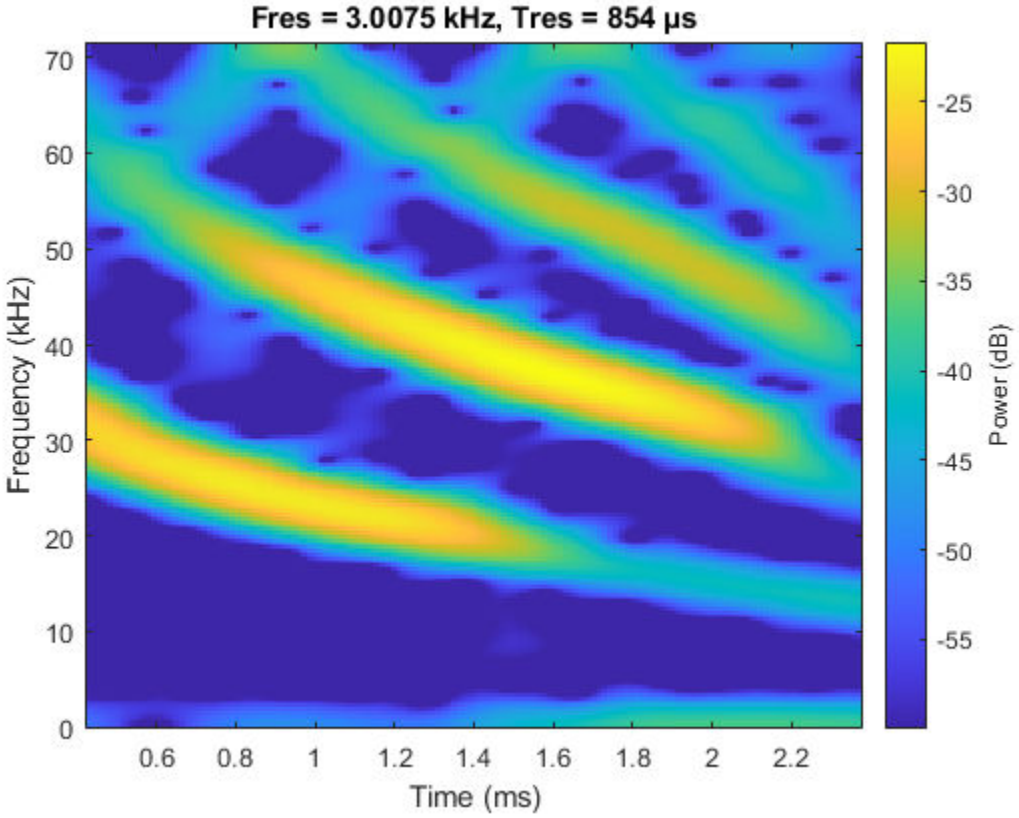

The spectrogram with default parameter values shows four coarse time-frequency ridges. Reduce the frequency resolution value to 3 kHz to get more details on the frequency variation of each ridge.

```
pspectrum(batsignal,Fs,'spectrogram','FrequencyResolution',3e3)
```




Observe that now the frequency ridges are better localized in frequency. However, since frequency and time resolution are inversely proportional, the time resolution of the spectrogram is considerably smaller. Set an overlap of 99% to smooth out the time windows. Use a `'MinThreshold'` of −60 dB to remove unwanted background content.

```
pspectrum(batsignal,Fs,'spectrogram','FrequencyResolution',3e3, ...
    'OverlapPercent',99,'MinTHreshold',-60)
```



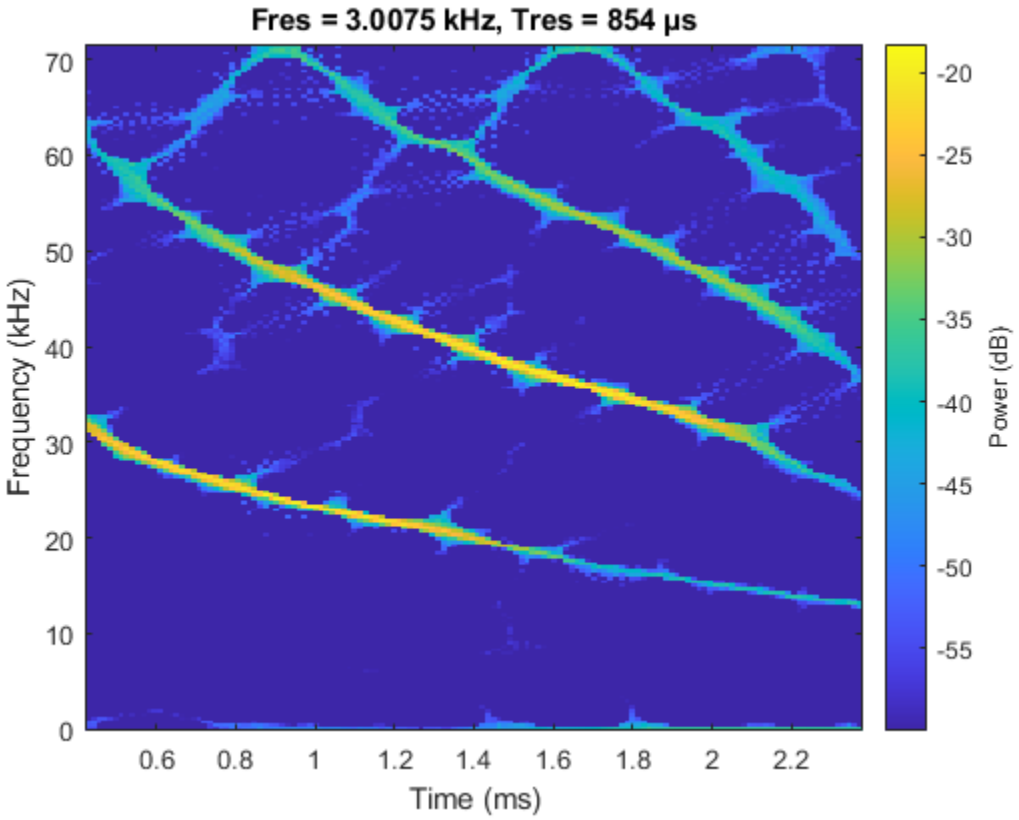

The new settings yield a spectrogram that clearly shows the four frequency ridges of the echolocation signal.

### Time-Frequency Reassignment

Even though we have been able to identify four frequency ridges, we can still see that each ridge is spread over several adjacent frequency bins. This is due to the leakage of the windowing method used in both time and frequency.

The `pspectrum` function is capable of estimating the center of energy for each spectral estimate in both time and frequency. If you reassign the energy of each estimate to the bin closest to the new time and frequency centers, you can correct for some of the leakage of the window. You can do this by using the `'Reassign'` parameter. Setting this parameter to `true` computes the reassigned spectrogram of the signal.

```
pspectrum(batsignal,Fs,'spectrogram','FrequencyResolution',3e3, ...
    'OverlapPercent',99,'MinTHreshold',-60,'Reassign',true)
```

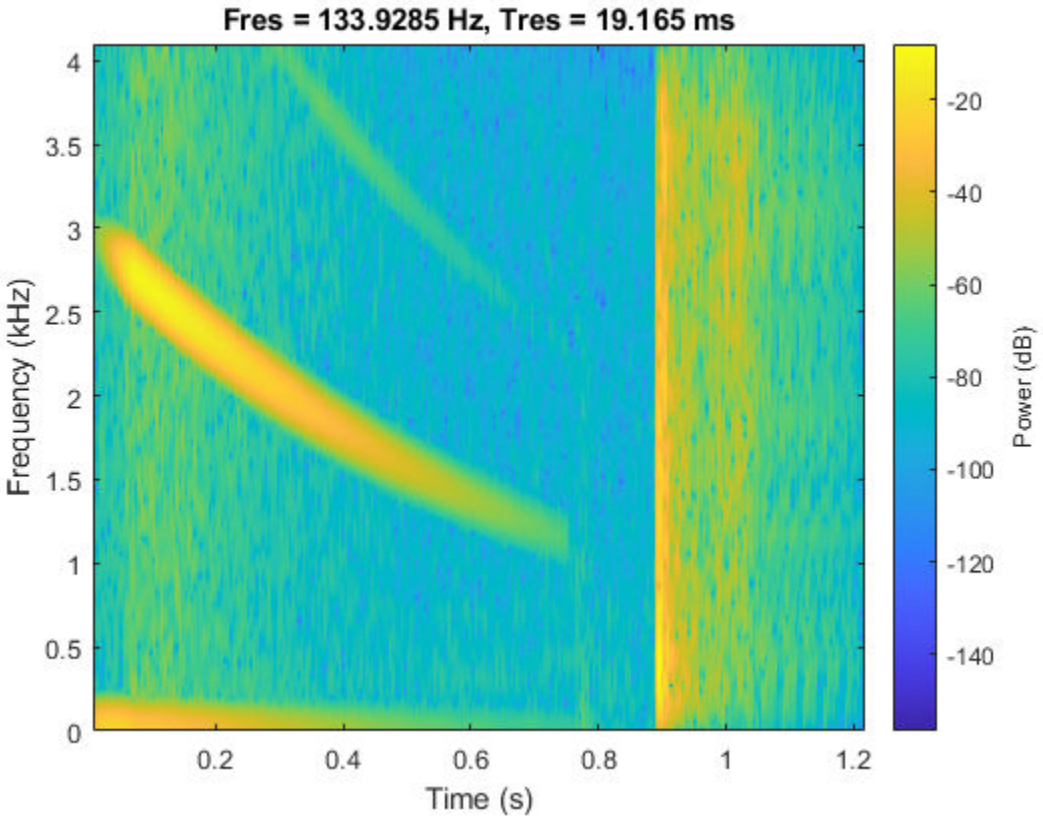



Now the frequency ridges are much sharper and better localized in time. You can also localize the signal energy using the function `fsst`, which is discussed in the next section.

### Reconstructing a Time-Frequency Ridge

Consider the following recording, consisting of a chirp signal whose frequency decreases over time and a final splat sound.

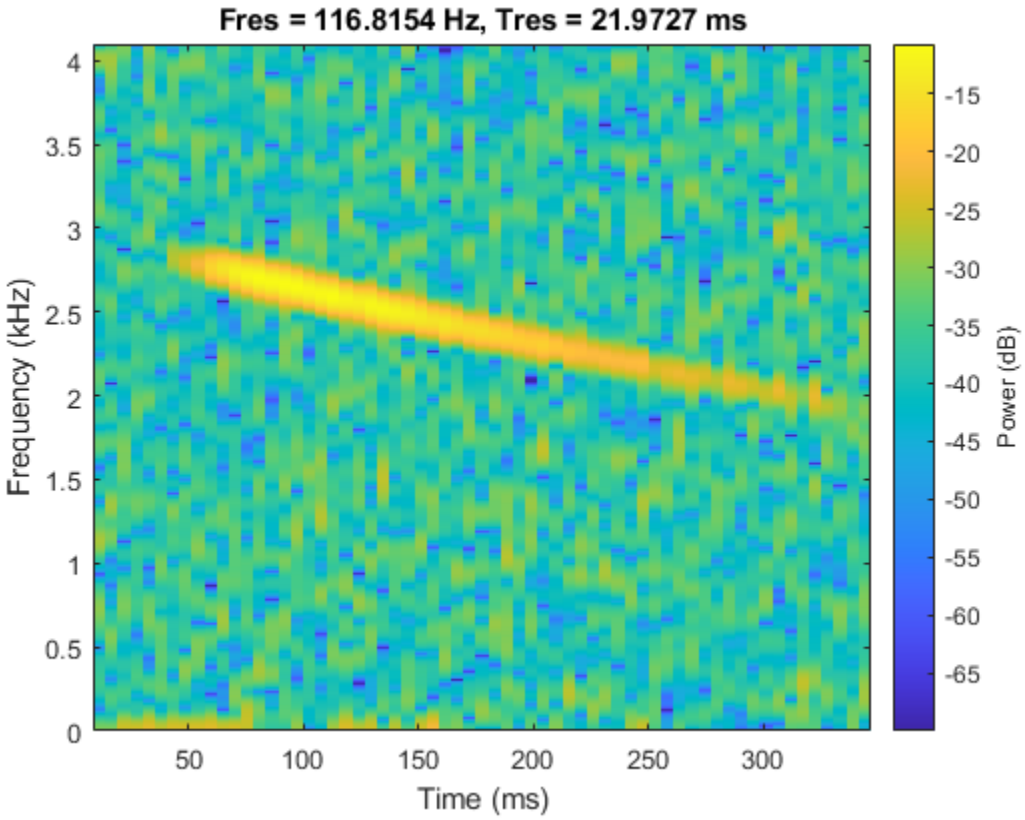
```
load splat
p = audioplayer(y,Fs,16);
play(p)
pspectrum(y,Fs,'spectrogram')
```




Let us reconstruct a portion of the "splat" sound by extracting a ridge in the time-frequency plane. We use `fsst` to sharpen the spectrum of a noisy version the splat signal, `tfridge` to identify the ridge of the chirp sound, and `ifsst` to reconstruct the chirp. The process denoises the reconstructed signal.

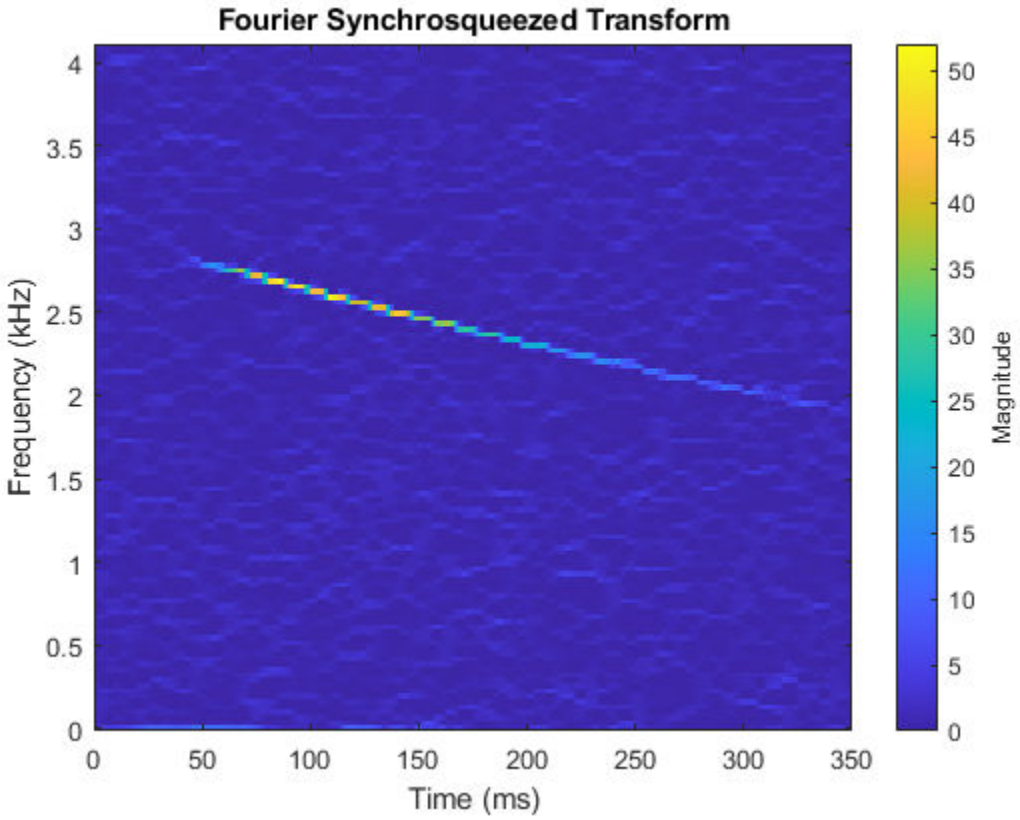
Add Gaussian noise to the chirp portion of the "splat" sound. The added noise simulates an audio recording taken with an inexpensive microphone. Examine the time-frequency spectral content.

```
rng('default')
t = (0:length(y)-1)/Fs;
yNoise = y + 0.1*randn(size(y));
yChirp = yNoise(t<0.35);
pspectrum(yChirp,Fs,'spectrogram','MinThreshold',-70)
```



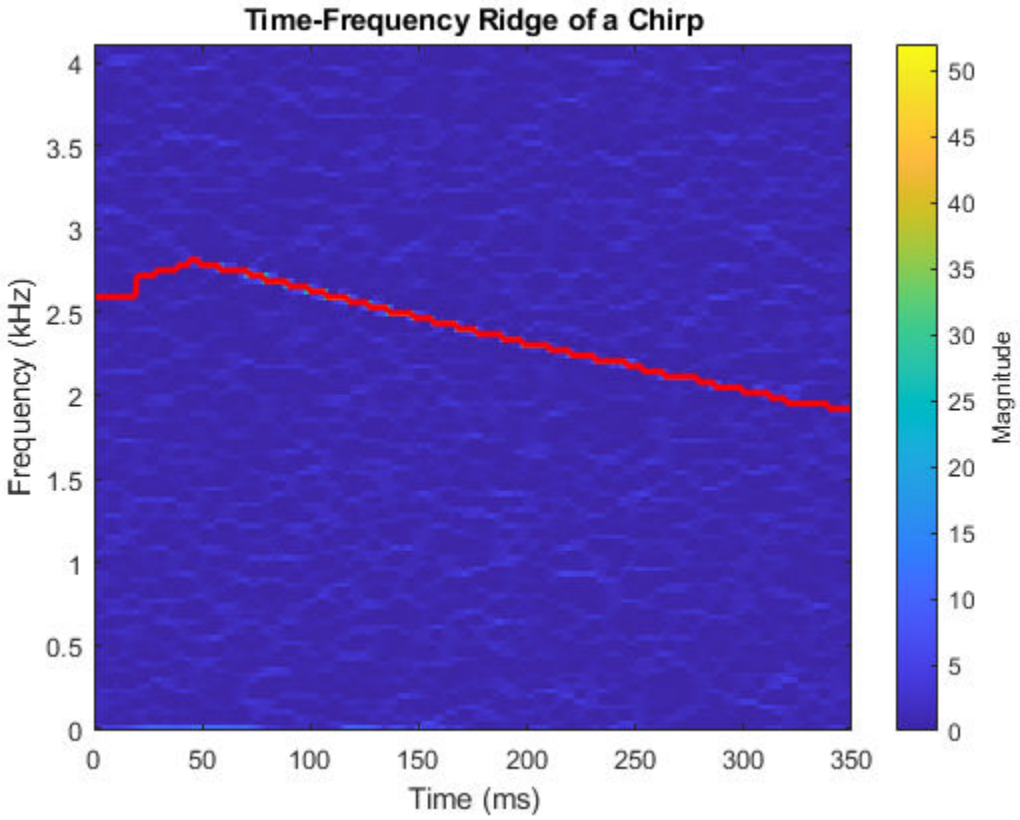

Sharpen the spectrum using the Fourier synchrosqueezed transform, `fsst`. `fsst` localizes energy in the time-frequency plane by reassigning energy in frequency for a fixed time. Compute and plot the synchrosqueezed transform of the noisy chirp.

```
fsst(yChirp,Fs,'yaxis')
```



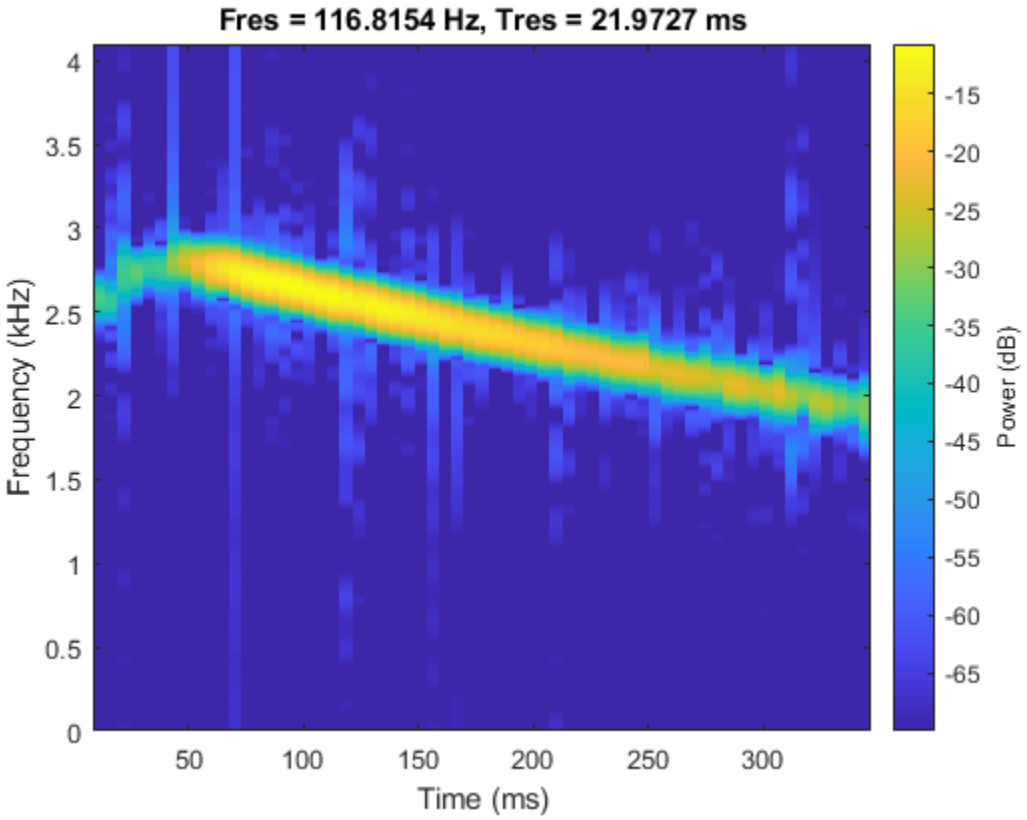

The chirp appears as a localized ridge in the time-frequency plane. Identify the ridge using `tfridge`. Plot the ridge along with the transform.

```
[sst,f] = fsst(yChirp,Fs);
[fridge, iridge] = tfridge(sst,f,10);
helperPlotRidge(yChirp,Fs,fridge);
```




Next, reconstruct the chirp signal using the ridge index vector `iridge`. Include one bin on each side of the ridge. Plot the spectrogram of the reconstructed signal.

```
yrec = ifsst(sst,kaiser(256,10),iridge,'NumFrequencyBins',1);
pspectrum(yrec,Fs,'spectrogram','MinThreshold',-70)
```



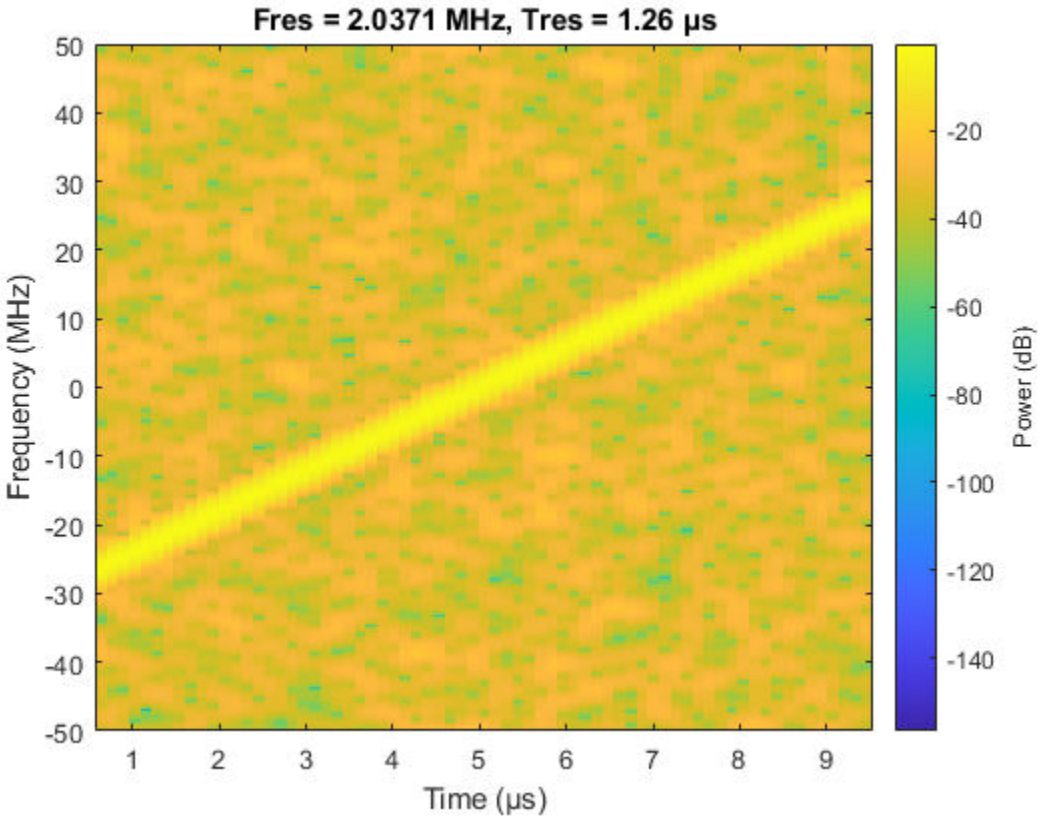

Reconstructing the ridge has removed noise from the signal. Play the noisy and denoised signals consecutively to hear the difference.

```
p = audioplayer([yChirp;zeros(size(yChirp));yrec],Fs,16);
play(p);
```

### Measuring Power

Consider a complex linear frequency modulated (LFM) pulse, which is a common radar waveform. Compute the spectrogram of the signal using a time resolution of 1.27 microseconds and 90% overlap.

```
Fs = 1e8;
bw = 60e6;
t = 0:1/Fs:10e-6;
IComp = chirp(t,-bw/2,t(end), bw/2,'linear',90)+0.15*randn(size(t));
QComp = chirp(t,-bw/2,t(end), bw/2,'linear',0) +0.15*randn(size(t));
IQData = IComp + 1i*QComp;

segmentLength = 128;
pspectrum(IQData,Fs,'spectrogram','TimeResolution',1.27e-6,'OverlapPercent',90)
```

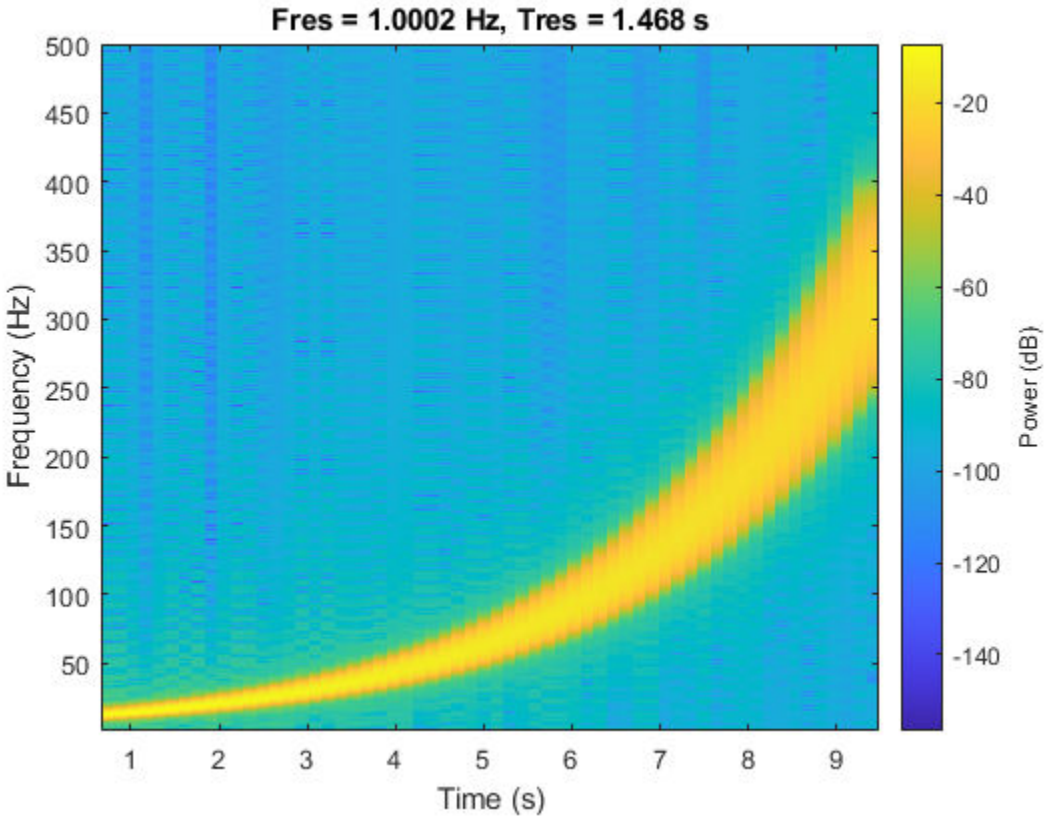



The parameters used to compute the spectrogram give a clear time-frequency representation of the LFM signal. `pspectrum` computes a power spectrogram, this means that the color values correspond to true power levels in dB. The color bar shows that the power level of the signal is around −4 dB.

### Logarithmic Frequency Scale Visualization

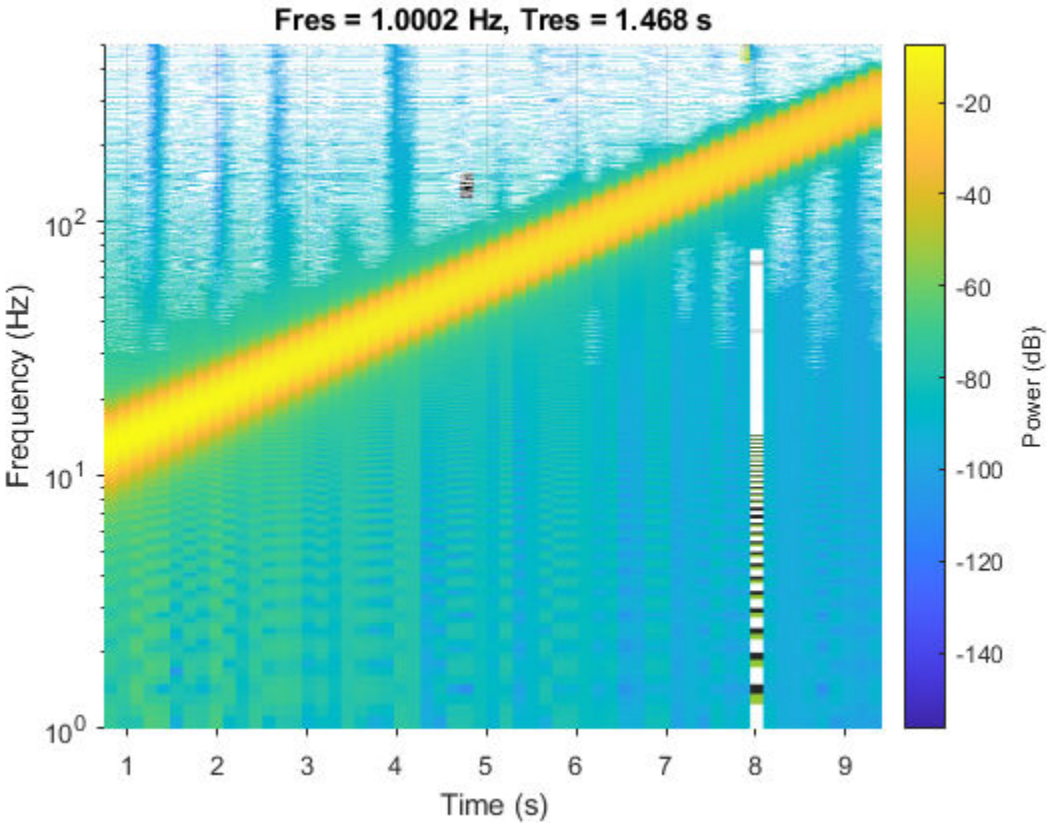
In certain applications, it may be preferable to visualize the spectrogram of a signal on a logarithmic frequency scale. You can achieve this by changing the `YScale` property of the *y*-axis. For example, consider a logarithmic chirp sampled at 1 kHz. The frequency of the chirp increases from 10 Hz to 400 Hz in 10 seconds.

```
Fs = 1e3;
t = 0:1/Fs:10;
fo = 10;
f1 = 400;
y = chirp(t,fo,10,f1,'logarithmic');
pspectrum(y,Fs,'spectrogram','FrequencyResolution',1, ...
    'OverlapPercent',90,'Leakage',0.85,'FrequencyLimits',[1 Fs/2])
```




The spectrogram of the chirp becomes a straight line when the frequency scale is logarithmic.

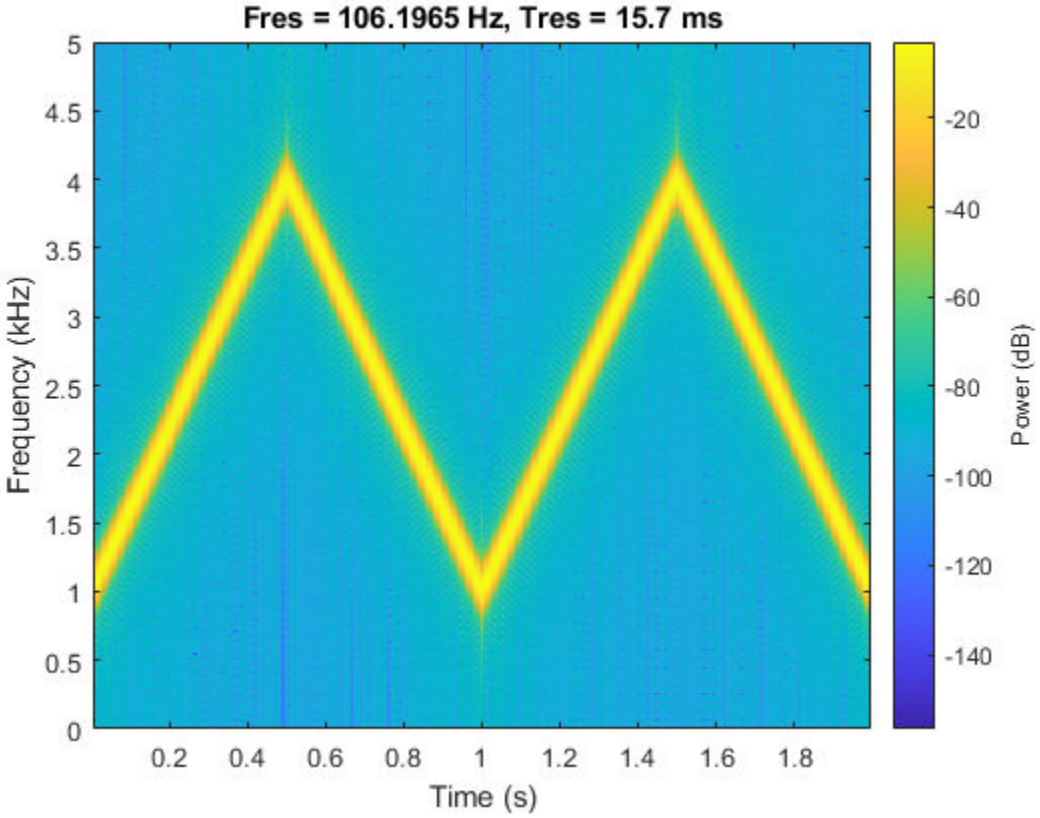
```
ax = gca;
ax.YScale = 'log';
```



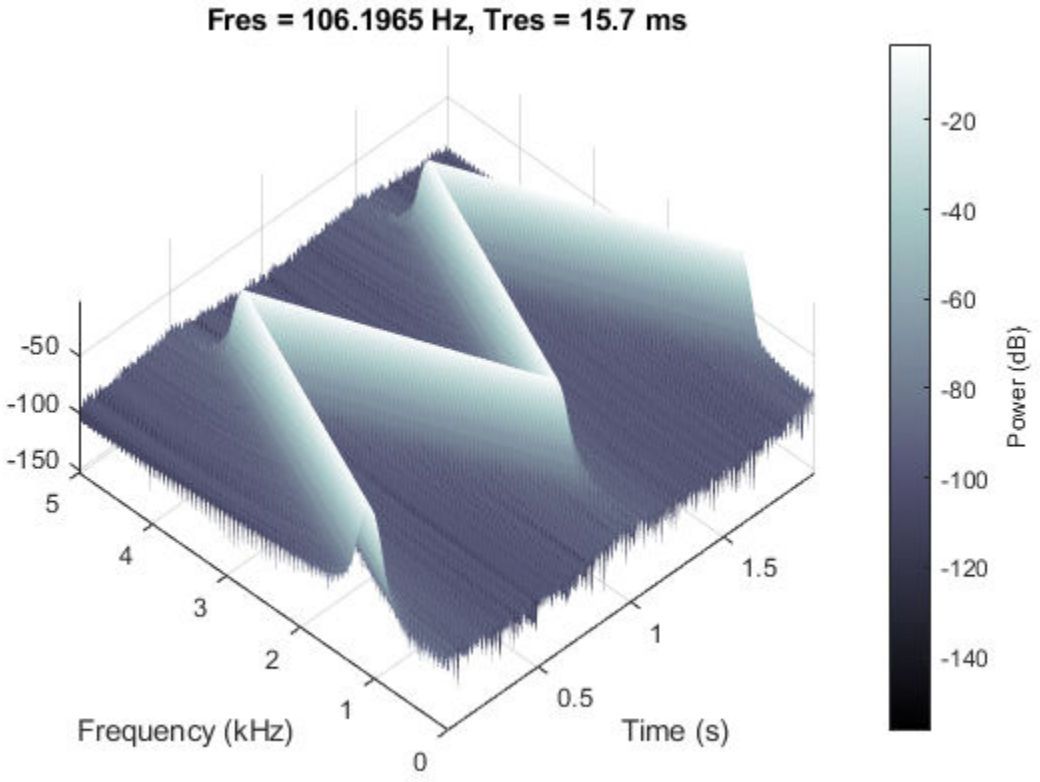

### Three-Dimensional Waterfall Visualization

With the `view` command, you can visualize the spectrogram of a signal as a three-dimensional waterfall plot. You can also change the display colors with the `colormap` function.

```
Fs = 10e3;
t = 0:1/Fs:2;
x1 = vco(sawtooth(2*pi*t,0.5),[0.1 0.4]*Fs,Fs);
pspectrum(x1,Fs,'spectrogram','Leakage',0.8)
```




```
view(-45,65)
colormap bone
```




### Finding Interferences Using Persistence Spectrum

The persistence spectrum of a signal is a time-frequency view that shows the percentage of the time that a given frequency is present in a signal. The persistence spectrum is a histogram in power-frequency space. The longer a particular frequency persists in a signal as the signal evolves, the higher its time percentage and thus the brighter or "hotter" its color in the display. Use the persistence spectrum to identify signals hidden in other signals.

Consider an interference narrowband signal embedded within a broadband signal. Generate a chirp sampled at 1 kHz for 500 seconds. The frequency of the chirp increases from 180 Hz to 220 Hz during the measurement.

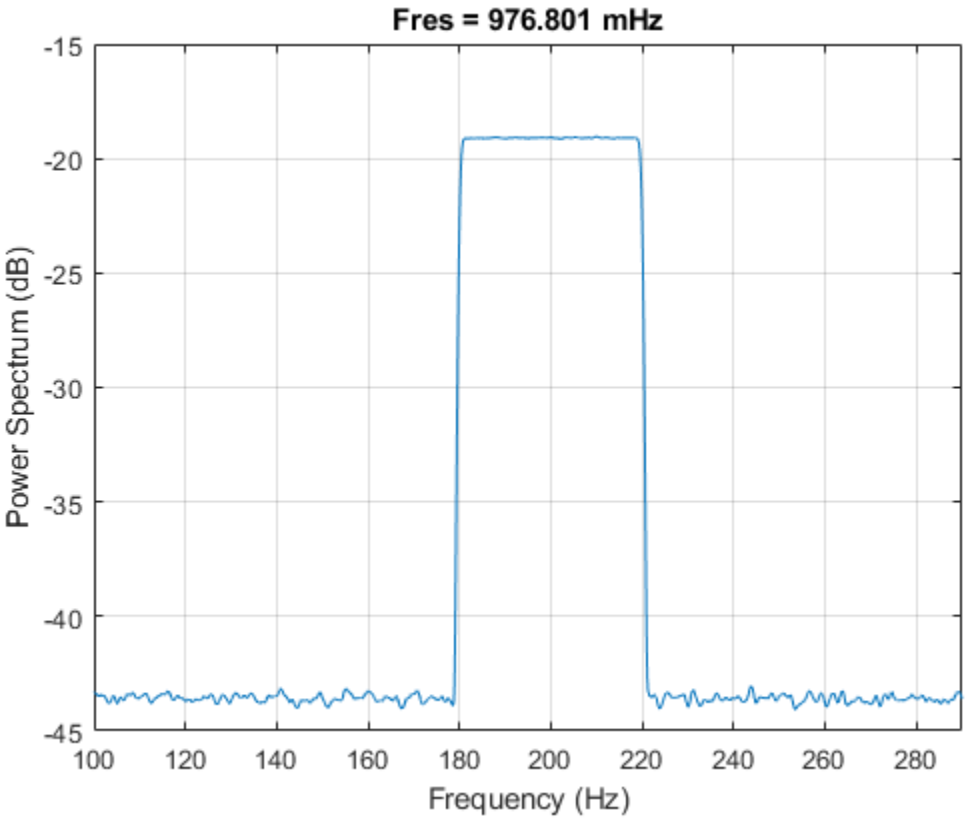
```
fs = 1000;
t = (0:1/fs:500)';

x = chirp(t,180,t(end),220) + 0.15*randn(size(t));
```

The signal also contains a 210 Hz interference, with an amplitude of 0.05, that is present only for 1/6 of the total signal duration.

```
idx = floor(length(x)/6);
x(1:idx) = x(1:idx) + 0.05*cos(2*pi*t(1:idx)*210);
```

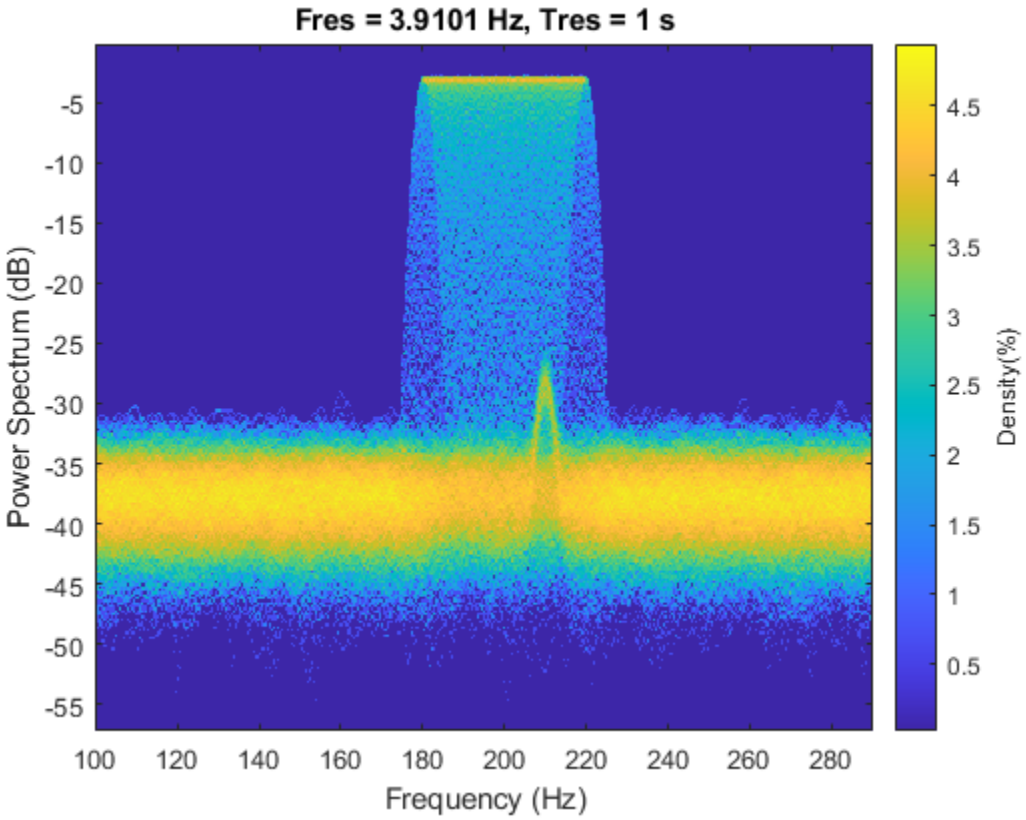
Compute the power spectrum of the signal over the 100 to 290 Hz interval. The weak sinusoid is obscured by the chirp.

```
pspectrum(x,fs,'FrequencyLimits',[100 290])
```




Compute the persistence spectrum of the signal. Now both signal components are clearly visible.

```
figure
colormap parula
pspectrum(x,fs,'persistence','FrequencyLimits',[100 290],'TimeResolution',1)
```




### Conclusions

In this example, you learned how to perform time-frequency analysis using the pspectrum function and how to interpret spectrogram data and power levels. You learned how to change time and frequency resolution to improve your understanding of signal and how to sharpen spectra and extract time-frequency ridges using `fsst`, `ifsst`, and `tfridge`. You learned how to configure the spectrogram plot to get a logarithmic frequency scale and three-dimensional visualization. Finally, you learned how to find interference signals by computing a persistence spectrum.

### Appendix

The following helper functions are used in this example.

- helperDTMFToneGenerator.m
- helperTimeFrequencyAnalysisPlotReassignment.m

## See Also

`fsst` | `ifsst` | `pspectrum` | `tfridge`

# Measure Power of Deterministic Periodic Signals

This example shows how to measure the power of deterministic periodic signals. Although continuous in time, periodic deterministic signals produce discrete power spectra. The example also shows how to improve power measurements using the reassignment technique.

### Signal Classification

In general, signals can be classified into three broad categories, power signals, energy signals, or neither. Deterministic signals which are made up of sinusoids, are an example of *power signals*, which have infinite energy but finite average power. Random signals also have finite average power and fall into the category of power signals. A transient signal is an example of *energy signals*, which start and end with zero amplitude. There are still other signals that cannot be characterized as either power signals or energy signals.

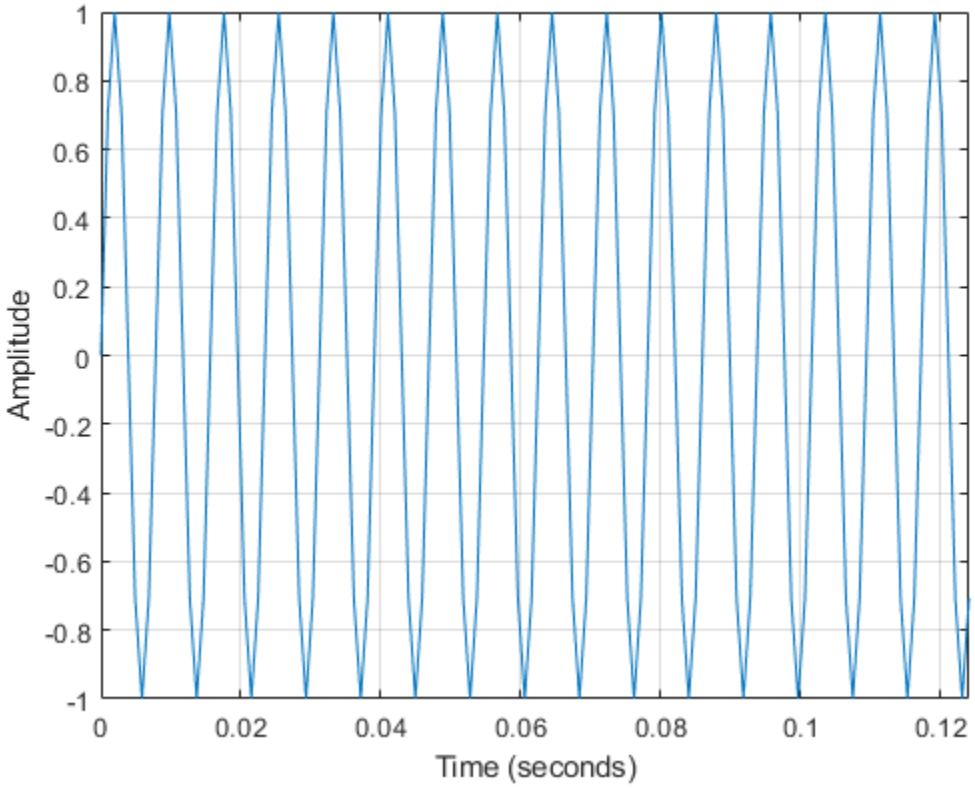
### Theoretical Power of a Single Sinusoid

As a first example, estimate the average power of a sinusoidal signal with a peak amplitude of 1 and a frequency component at 128 Hz.

```
Fs = 1024;
t = 0:1/Fs:1-(1/Fs);
A = 1;
F1 = 128;
x = A*sin(2*pi*t*F1);
```

Plot a portion of the signal in the time domain.

```
idx = 1:128;
plot(t(idx),x(idx))
ylabel('Amplitude')
xlabel('Time (seconds)')
axis tight
grid
```




The theoretical average power (mean-square) of each complex sinusoid is $A^2/4$, which in this example is 0.25 or –6.02 dB. So, accounting for the power in the positive and negative frequencies results in an average power of $2 \times A^2/4$.

```
power_theoretical = (A^2/4)*2
```

```
power_theoretical = 0.5000
```

Compute in dB the power contained in the positive frequencies:

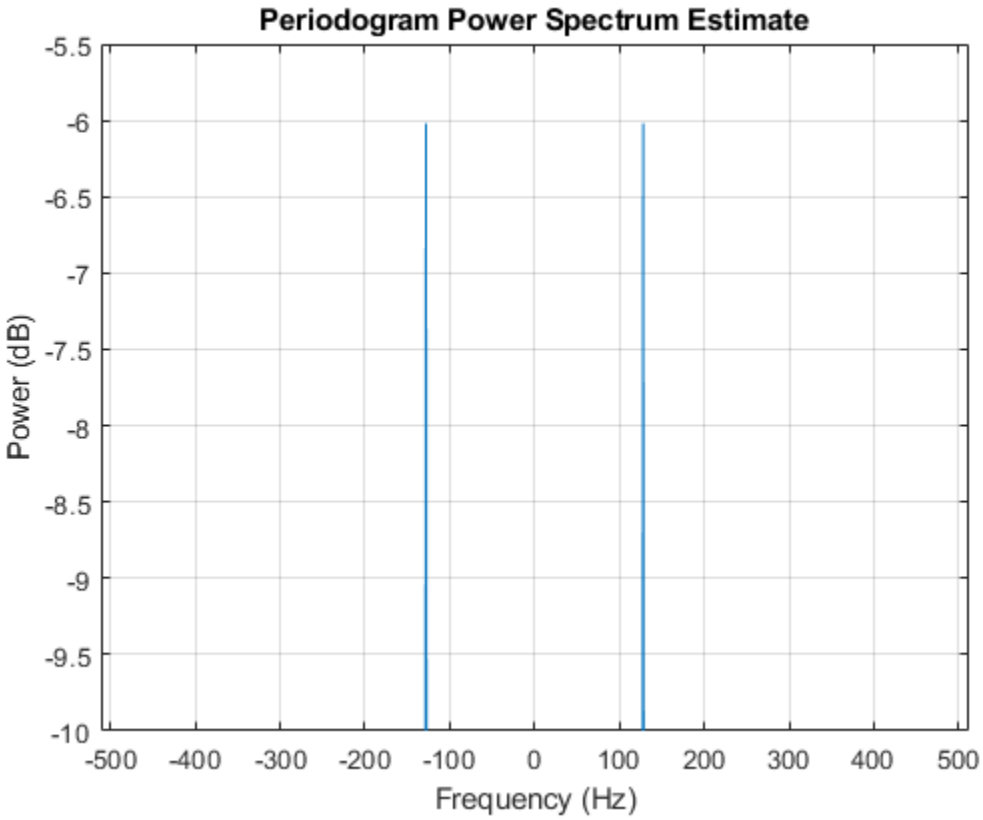
```
pow2db(power_theoretical/2)
```

```
ans = -6.0206
```

**Measuring the Power of a Single Sinusoid**

To measure the average power of the signal, call `periodogram` and specify the `'power'` option.

```
periodogram(x,hamming(length(x)),[],Fs,'centered','power')
ylim([-10 -5.5])
```

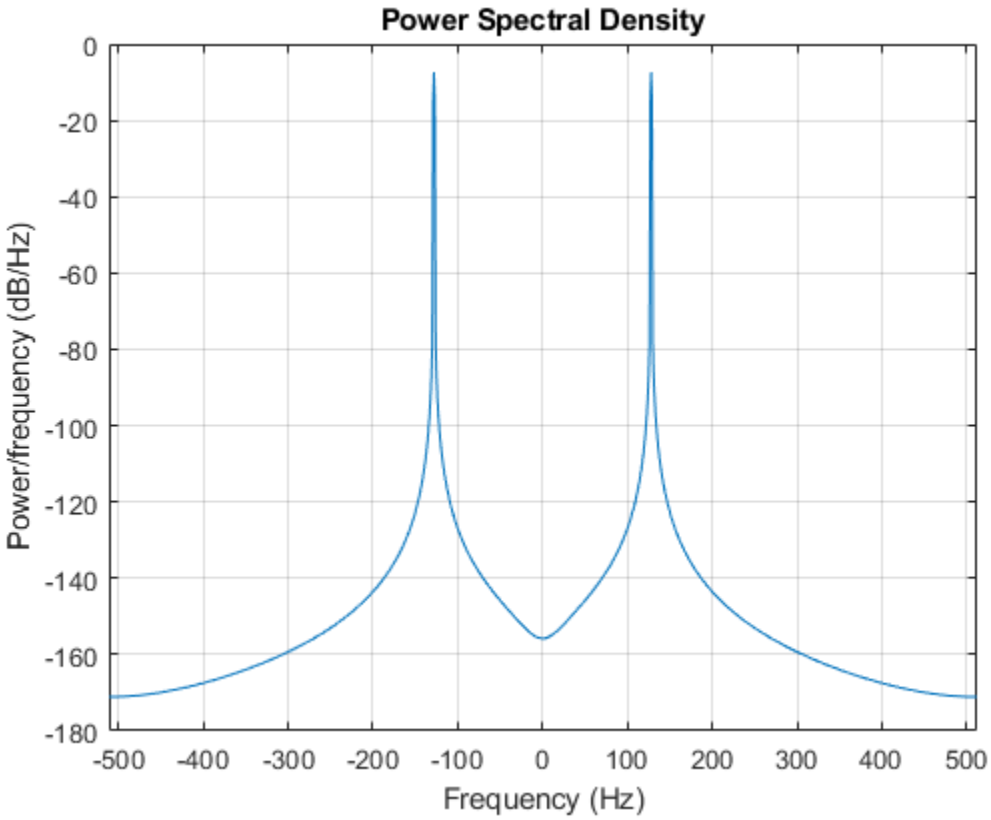



As you can see from the zoomed-in portion of the plot, each complex sinusoid has an average power of roughly –6 dB.

### Estimating the Power of a Single Sinusoid via PSD

Another way to calculate the average power of a signal is by "integrating" the area under the PSD curve.

```
periodogram(x,hamming(length(x)),[],Fs,'centered','psd')
```




In this plot, the peaks of the spectrum plot do not have the same height as in the power spectrum plot. The heights are different because it is the area under the curve — which is the measure of the average power — that matters when taking power spectral density (PSD) measurements. To verify that statement, use the `bandpower` function, which calculates the average power using the rectangle approximation to integrate under the curve.

```
[Pxx_hamming,F] = periodogram(x,hamming(length(x)),[],Fs,'psd');
power_freqdomain = bandpower(Pxx_hamming,F,'psd')
```

```
power_freqdomain = 0.5000
```

According to Parseval's theorem, the total average power of a sinusoid is the same in both the time domain and the frequency domain. Use that fact to check the value of the signal's estimated total average power by summing up the signal in the time domain.

```
power_timedomain = sum(abs(x).^2)/length(x)
```

```
power_timedomain = 0.5000
```

### Theoretical Power of Multiple Sinusoids

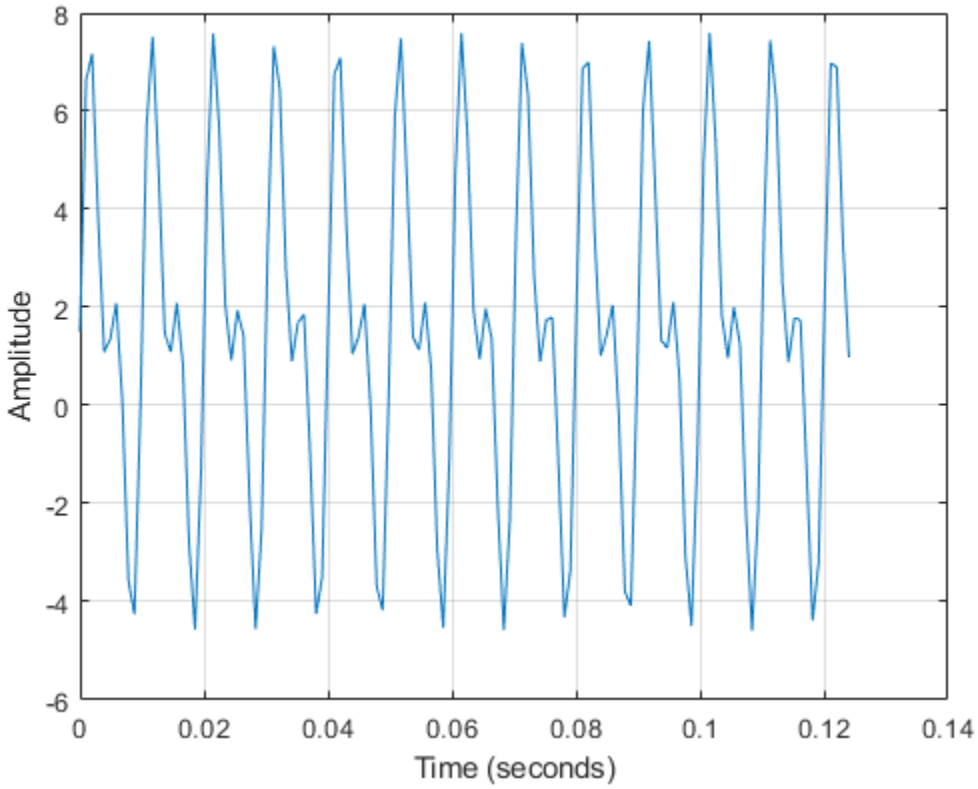
For the second example, estimate the total average power of a signal containing energy at multiple frequency components: one at DC with amplitude 1.5, one at 100 Hz with amplitude 4, and one at 200 Hz with amplitude 3.

```
Fs = 1024;
t  = 0:1/Fs:1-(1/Fs);
```

```
Ao = 1.5;
A1 = 4;
A2 = 3;
F1 = 100;
F2 = 200;
x  = Ao + A1*sin(2*pi*t*F1) + A2*sin(2*pi*t*F2);
```

Plot the first 128 samples of the signal.

```
idx = 1:128;
plot(t(idx),x(idx))
grid
ylabel('Amplitude')
xlabel('Time (seconds)')
```




As in the previous example, the theoretical average power of each complex sinusoid is $A^2/4$. The DC average power of the signal is equal to its peak power (since it is constant) and therefore is given by $A_0^2$. Accounting for the power in the positive and negative frequencies results in a total average power value (sum of the average power of each harmonic component) of $A_0^2 + 2 \times A_1^2/4 + 2 \times A_2^2/4$ for the signal.

```
power_theoretical = Ao^2 + (A1^2/4)*2 + (A2^2/4)*2
```

```
power_theoretical = 14.7500
```

Calculate the average power of each unique frequency component in dB to see that the theoretical results match the mean-square spectrum plot below.

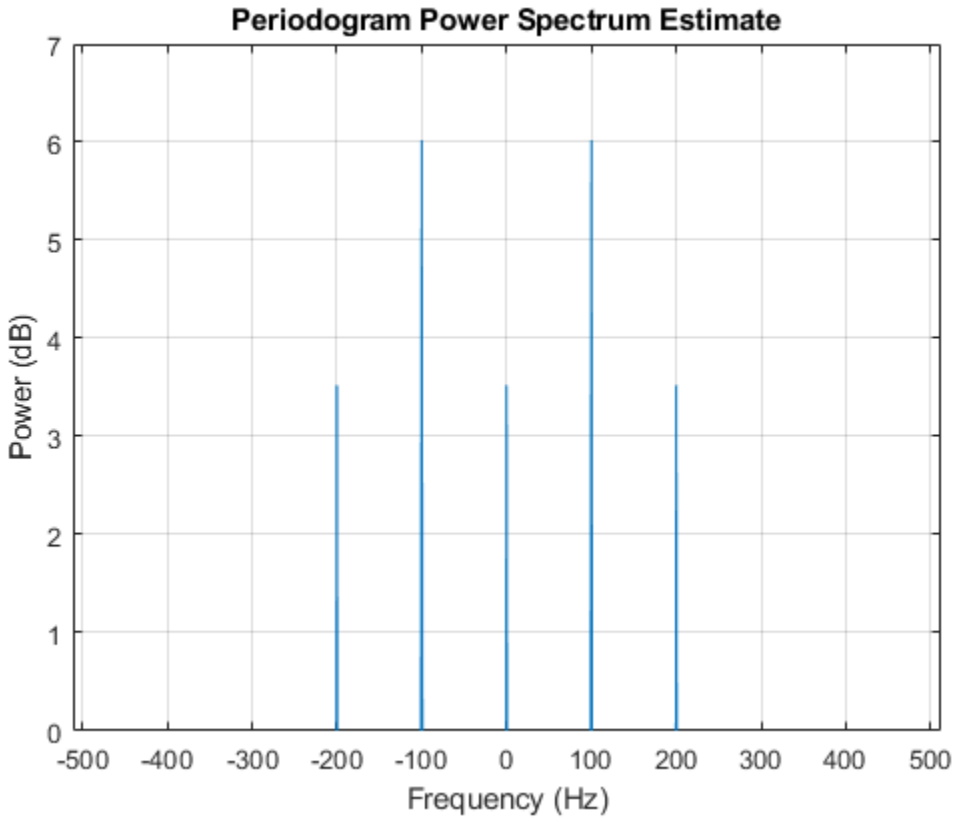
```
pow2db([Ao^2 A1^2/4 A2^2/4])
```

```
ans = 1×3

    3.5218    6.0206    3.5218
```

### Measuring the Power of Multiple Sinusoids

To measure once again the average power of the signal, use the `periodogram` function once more to calculate and plot the power spectrum of the signal.

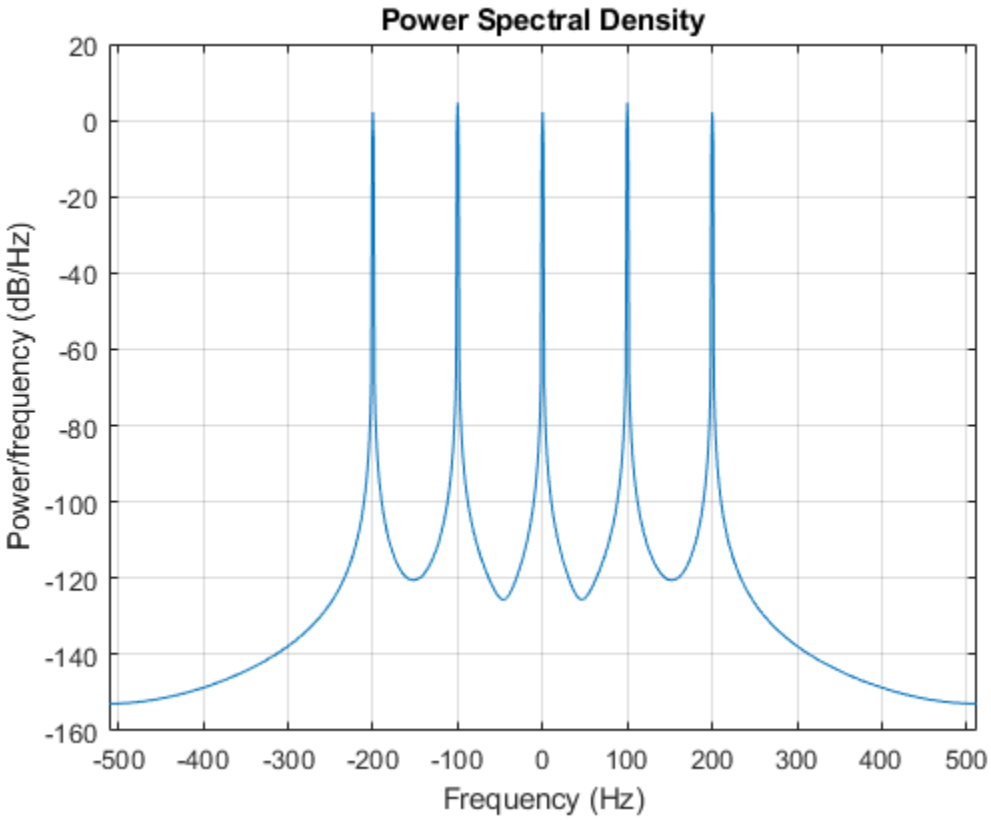
```
periodogram(x,hamming(length(x)),[],Fs,'centered','power')
ylim([0 7])
```




### Estimating the Power of Multiple Sinusoids Using PSD

As in the first example, estimate the total average power of the signal by "integrating" under the PSD curve.

```
periodogram(x,hamming(length(x)),[],Fs,'centered','psd')
```




Once again the height of the peaks of the spectral density plot at a specific frequency component may not match the ones of the plot of the power spectrum. The difference is due to the reasons noted in the first example.

```
[Pxx, F] = periodogram(x, hamming(length(x)),[],Fs,'centered','psd');
power_freqdomain = bandpower(Pxx,F,'psd')
```

```
power_freqdomain = 14.7500
```

Again verify the estimated average power of the signal by invoking Parseval's theorem and summing up the signal in the time domain.

```
power_timedomain = sum(abs(x).^2)/length(x)
```

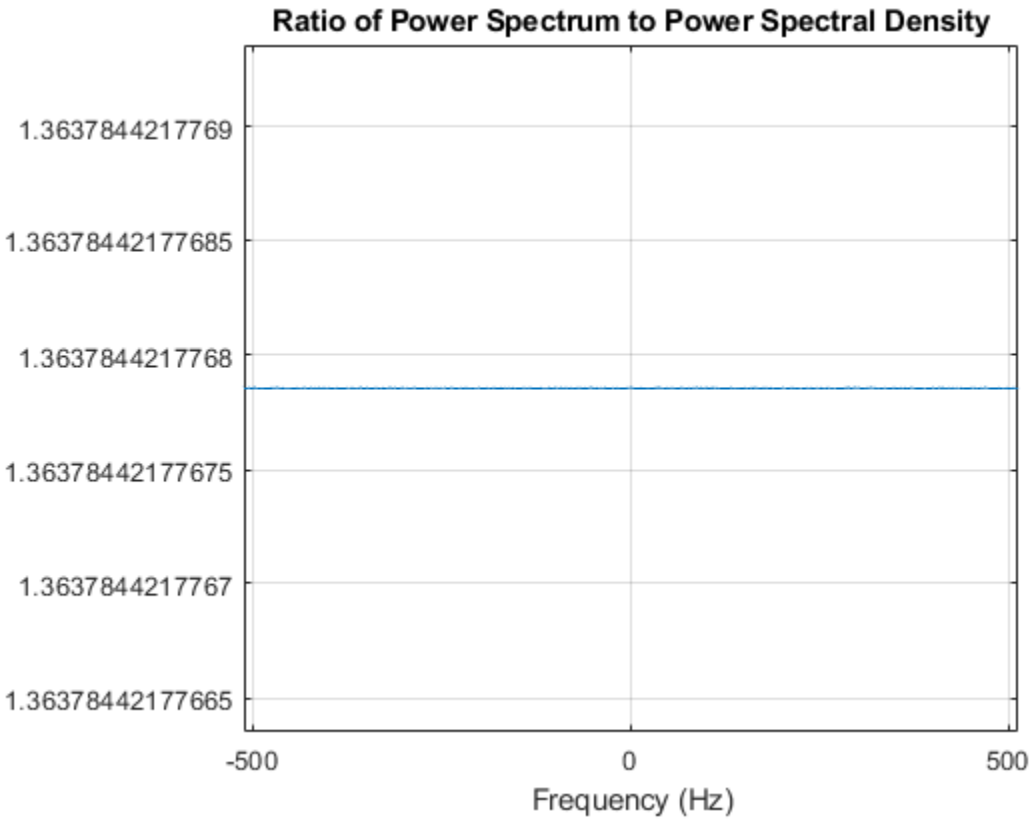
```
power_timedomain = 14.7500
```

**Relationship between Power Spectrum, Power Spectral Density and ENBW**

You may have noticed that, while the height of the peaks of the power and power spectral density plots are different, the ratio of one to the other is constant.

```
Pxx = periodogram(x,hamming(length(x)),[],Fs,'centered','psd');
Sxx = periodogram(x,hamming(length(x)),[],Fs,'centered','power');

plot(F,Sxx./Pxx)
grid
axis tight
```

```
xlabel('Frequency (Hz)')
title('Ratio of Power Spectrum to Power Spectral Density')
```




```
ratio = mean(Sxx./Pxx)
```

```
ratio = 1.3638
```

The ratio of power to power spectral density is related to the two-sided equivalent noise bandwidth (ENBW) of the window. You can compute this ratio directly by calling the `enbw` function with the window and its corresponding sample rate as input arguments.

```
bw = enbw(hamming(length(x)),Fs)
```

```
bw = 1.3638
```

### Enhanced Power Measurements Using Reassigned Periodogram

In the previous sections, power was measured from one or multiple sinusoids having a frequency that coincided with a bin. Peak power estimates are usually less accurate when the signal frequency is out of bin. To see this effect, create a sinusoid with a non-integer number of cycles over a one-second period.

```
Fs = 1024;
t = 0:1/Fs:1-(1/Fs);
A = 1;
F = 20.4;
x = A*sin(2*pi*F*t);
```

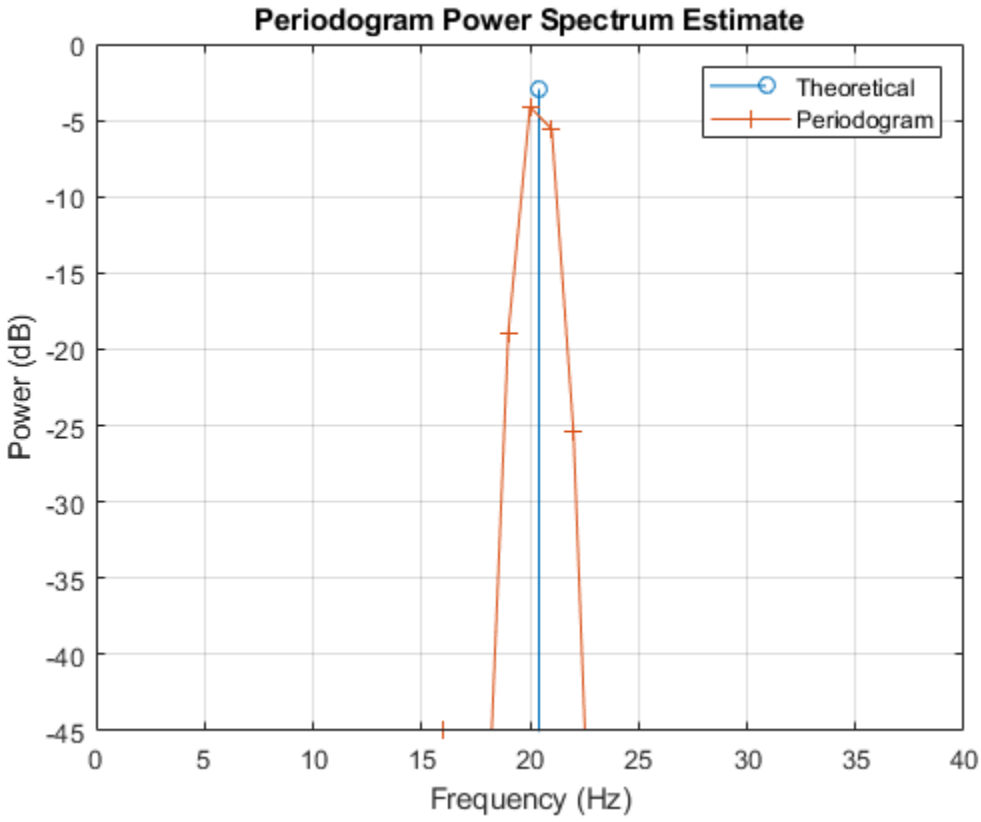
```
nfft = length(x);
power_theoretical = pow2db(A^2/4*2);
```

Create a Hamming window and a flat top window.

```
w1 = hamming(length(x));
w2 = flattopwin(length(x));
```

Compute the periodogram of x using the Hamming window. Zoom in on the peak.

```
h1 = figure;
stem(F,power_theoretical,'BaseValue',-50);

[Pxx1,f1] = periodogram(x,w1,nfft,Fs,'power');

hold on
plot(f1,pow2db(Pxx1),'+-')

axis([0 40 -45 0])
legend('Theoretical','Periodogram')
xlabel('Frequency (Hz)')
ylabel('Power (dB)')
title('Periodogram Power Spectrum Estimate')
grid
```




The peak power estimate is below the theoretical peak, and the frequency of the peak estimate differs from the true frequency.

```
[Pmax,imax] = max(Pxx1);
dPmax_w1 = pow2db(Pmax) - power_theoretical
```

```
dPmax_w1 = -1.1046
```

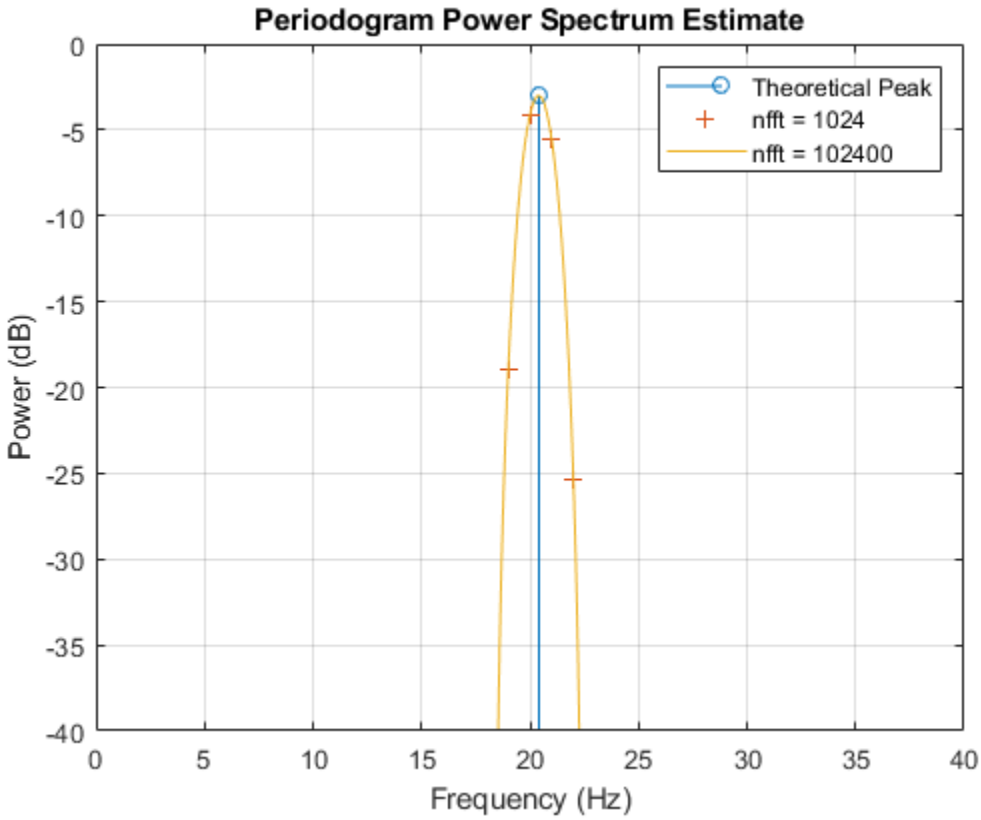
```
dFreq = f1(imax) - F
```

```
dFreq = -0.4000
```

**Reduce Amplitude Error with Zero-Padding**

To see why this is happening, compute the periodogram using a larger number of FFT bins.

```
[Pxx2,f2] = periodogram(x,w1,100*nfft,Fs,'power');

figure
stem(F,power_theoretical,'BaseValue',-50)

hold on
plot(f1,pow2db(Pxx1),'+')
plot(f2,pow2db(Pxx2))
hold off

axis([0 40 -40 0])
legend('Theoretical Peak','nfft = 1024','nfft = 102400')
xlabel('Frequency (Hz)')
ylabel('Power (dB)')
title('Periodogram Power Spectrum Estimate')
grid
```

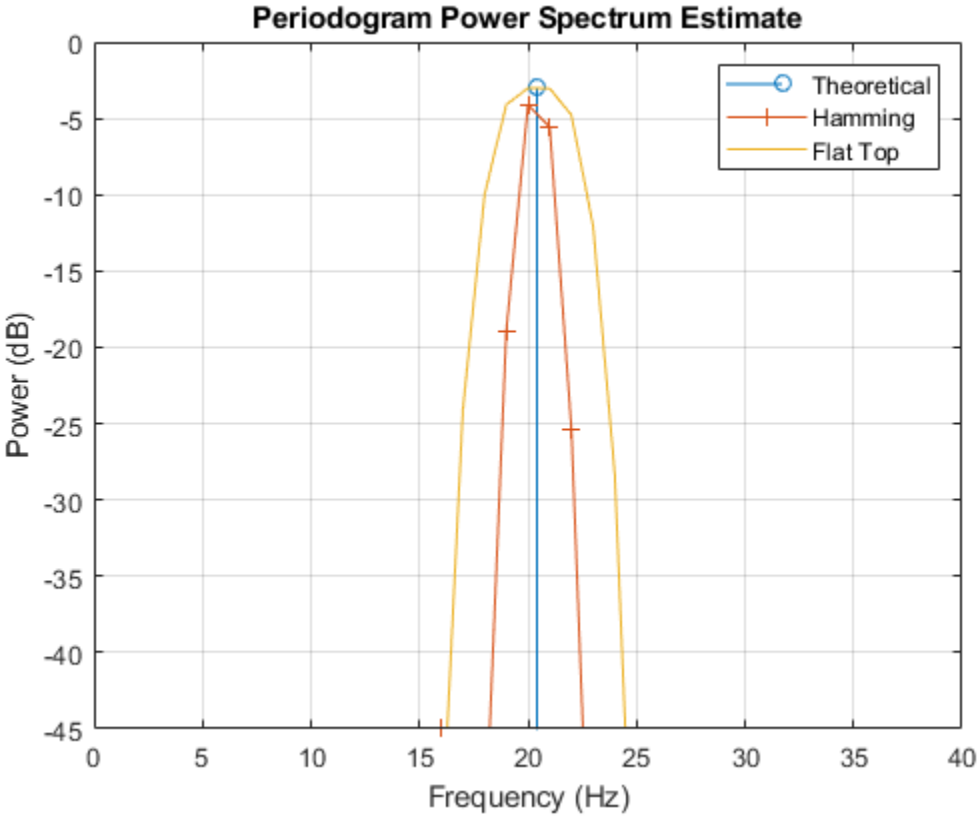


In the original periodogram, the spectral peak is located between two bins, and for that reason the estimated peak is below the theoretical peak. Increasing the number of FFT bins gives a better picture of the spectrum, although this may be a computationally expensive way to improve peak measurements.

**Reduce Amplitude Error with a Flat Top Window**

Another way to produce a better estimate for the peak amplitude is to use a different window. Compute the periodogram of x using the flat top window.

```
[Pxx,F1] = periodogram(x,w2,nfft,Fs,'power');

figure(h1)
plot(F1,pow2db(Pxx))
legend('Theoretical','Hamming','Flat Top')
hold off
```




The flat top window is broad and flat. It produces a peak estimate closer to the theoretical value when x does not contain an integer number of cycles, and hence the spectral peak does not fall exactly on a bin.

```
dPmax_w2 = pow2db(max(Pxx)) - power_theoretical
```

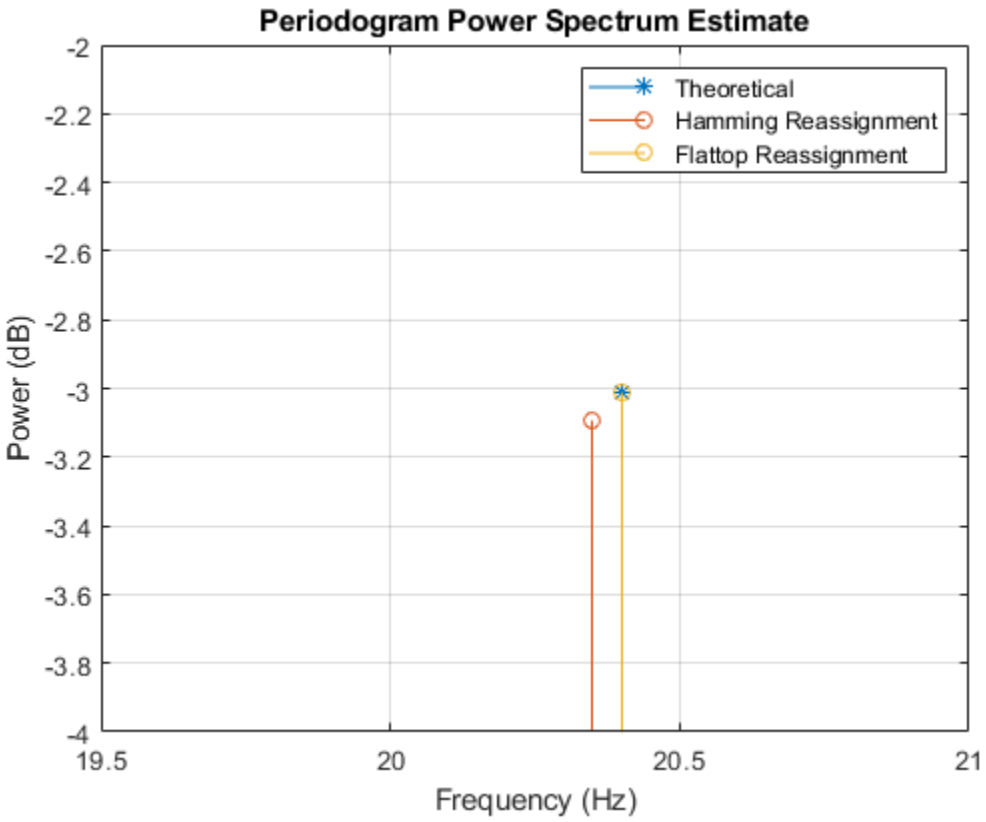
```
dPmax_w2 = -6.2007e-04
```

The broader peak that the flat top window produces could be a disadvantage when trying to resolve closely spaced peaks, and the frequency of the measured peak is again different from the frequency of the theoretical peak.

### Reduce Amplitude Error with Reassigned Periodogram

Now add the `'reassigned'` flag to `periodogram`. Periodogram reassignment uses phase information, which is normally discarded, to reassign the signal to its center of energy. The procedure can result in sharper spectral estimates. Plot the reassigned periodogram of `x` and zoom in on the peak. Use the Hamming window and the flat top window.

```
[RPxx1,~,~,Fc1] = periodogram(x,w1,nfft,Fs,'power','reassigned');
[RPxx2,~,~,Fc2] = periodogram(x,w2,nfft,Fs,'power','reassigned');

stem(F,power_theoretical,'*','BaseValue',-40)
hold on
stem(Fc1,pow2db(RPxx1),'BaseValue',-50)
stem(Fc2,pow2db(RPxx2),'BaseValue',-50)
hold off

legend('Theoretical','Hamming Reassignment','Flattop Reassignment')
xlabel('Frequency (Hz)')
ylabel('Power (dB)')
title('Periodogram Power Spectrum Estimate')
axis([19.5 21 -4 -2])
grid
```




The reassigned estimates of power are closer to the theoretical value for both windows, with the flat top window producing the best peak measurement.

```
[RPxx1max,imax1] = max(RPxx1);
[RPxx2max,imax2] = max(RPxx2);
dPmax_reassign_w1 = pow2db(RPxx1max) - power_theoretical
```

```
dPmax_reassign_w1 = -0.0845
```

```
dPmax_reassign_w2 = pow2db(RPxx2max) - power_theoretical
```

```
dPmax_reassign_w2 = -1.1131e-05
```

The frequency estimates are also improved using the reassigned periodogram, with the flat top window again giving the best results.

```
Fc1(imax1)-F
```

```
ans = -0.0512
```

```
Fc2(imax2)-F
```

```
ans = 5.6552e-04
```

## See Also

`bandpower` | `enbw` | `periodogram` | `pow2db`

# Spectral Analysis of Nonuniformly Sampled Signals

This example shows how to perform spectral analysis on nonuniformly sampled signals. It helps you determine if a signal is uniformly sampled or not, and if not, it shows how to compute its spectrum or its power spectral density.

The example introduces the Lomb-Scargle periodogram, which can compute spectra of nonuniformly sampled signals.

### Nonuniformly Sampled Signals

Nonuniformly sampled signals are often found in the automotive industry, in communications, and in fields as diverse as medicine and astronomy. Nonuniform sampling might be due to imperfect sensors, mismatched clocks, or event-triggered phenomena.

The computation and study of spectral content is an important part of signal analysis. Conventional spectral analysis techniques like the periodogram and the Welch method require the input signal to be uniformly sampled. When the sampling is nonuniform, one can resample or interpolate the signal onto a uniform sample grid. This, however, can add undesired artifacts to the spectrum and might lead to analysis errors.

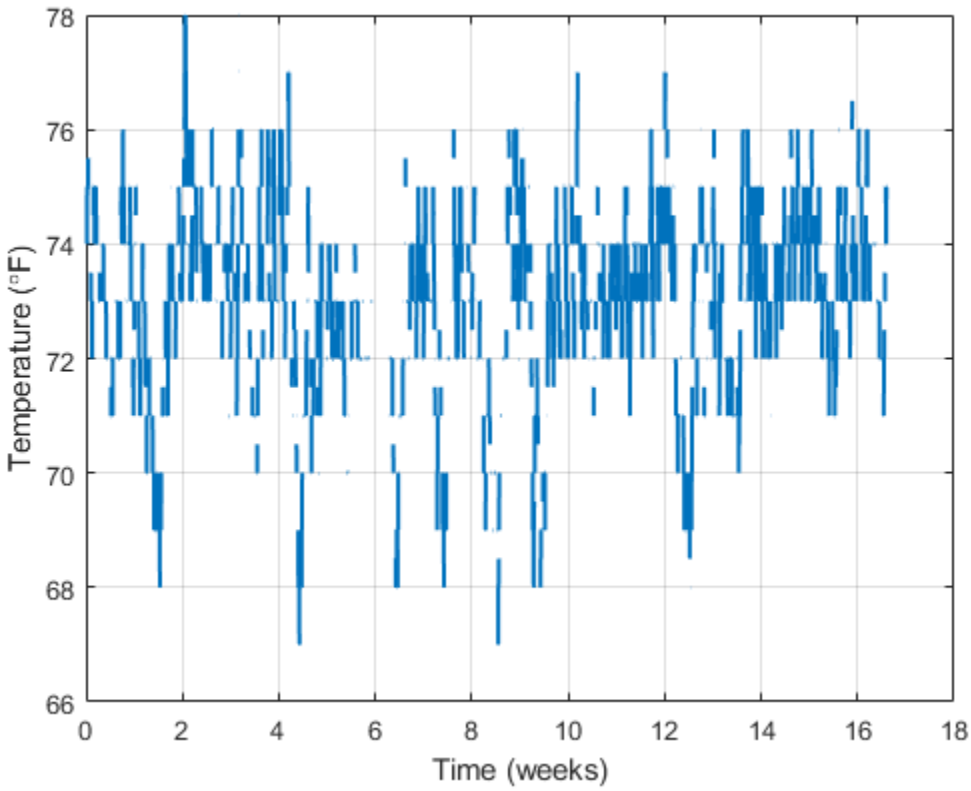
A better alternative is to use the Lomb-Scargle method, which works directly with the nonuniform samples and thus makes it unnecessary to resample or interpolate. The algorithm has been implemented in the `plomb` function.

### Spectral Analysis of Signals with Missing Data

Consider a temperature monitoring system in which a microcontroller records the temperature of a room and transmits this reading every 15 minutes to a cloud-based server that stores it. It is known that glitches in internet connectivity prevent the cloud-based system from receiving some of the readings sent by the microcontroller. Also, at least once during the measurement period the microcontroller's battery ran out, leading to a large gap in the sampling.

Load the temperature readings and the corresponding timestamps.

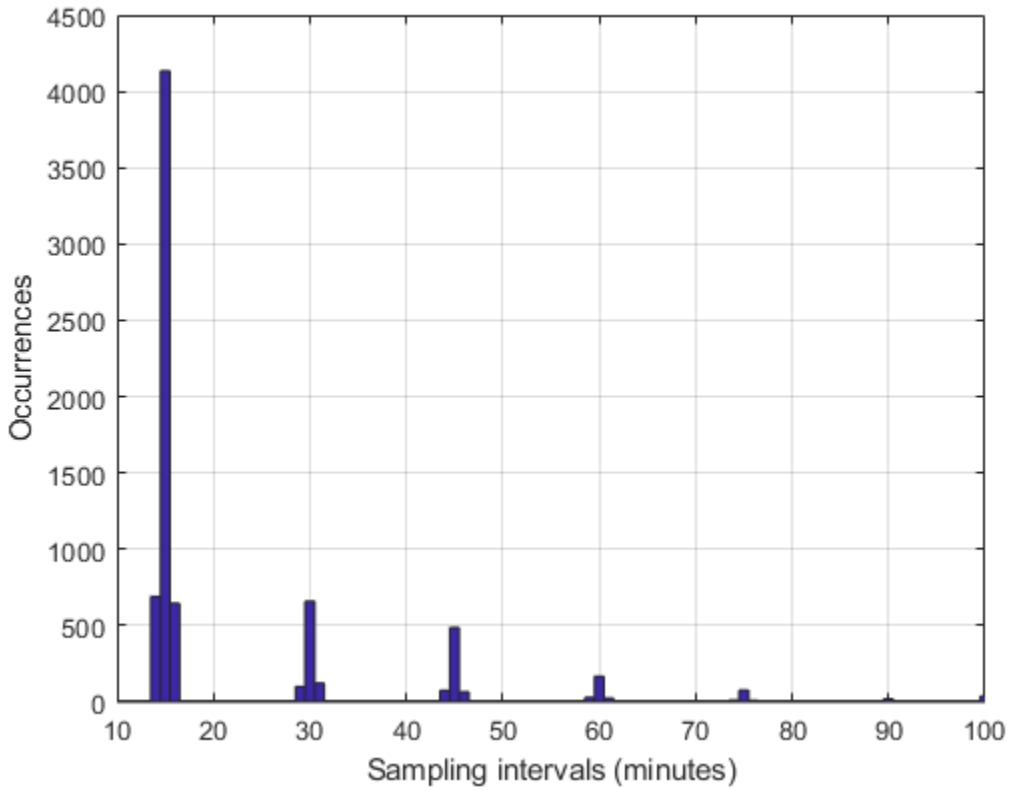
```
load('nonuniformdata.mat','roomtemp','t1')

figure
plot(t1/(60*60*24*7),roomtemp,'LineWidth',1.2)

grid
xlabel('Time (weeks)')
ylabel('Temperature (\circF)')
```




An easy way to determine if a signal is uniformly sampled is to take a histogram of the intervals between successive sample times.

Plot a histogram of sampling intervals (time differences) in minutes. Include only points at which samples are present.

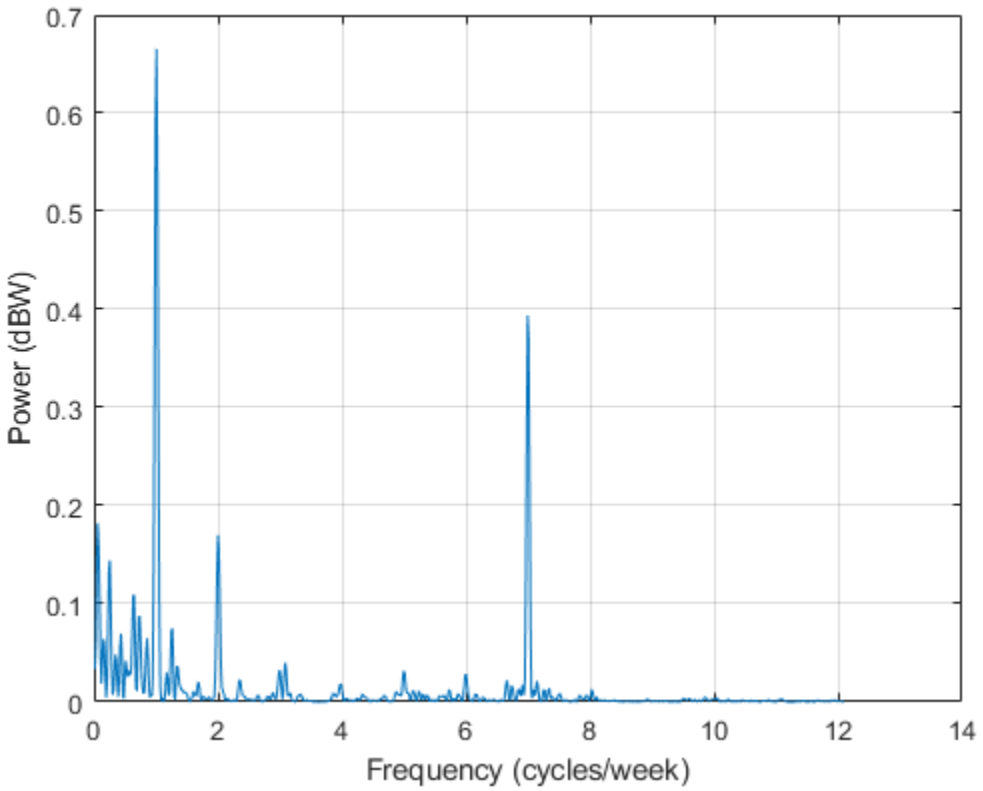
```
tAtPoints = t1(~isnan(roomtemp))/60;
TimeIntervalDiff = diff(tAtPoints);

figure
hist(TimeIntervalDiff,0:100)
grid
xlabel('Sampling intervals (minutes)')
ylabel('Occurrences')
xlim([10 100])
```




The majority of the measurements are spaced approximately 15 minutes apart, as expected. However, a fair number of occurrences have sampling intervals of around 30 and 45 minutes, which correspond to one or two consecutive dropped samples. This causes the signal to be nonuniformly sampled. Furthermore, the histogram shows some jitter surrounding the bars showing high occurrences. This could relate to TCP/IP latency.

Use the Lomb-Scargle method to compute and visualize the spectral content of the signal. To help visualize the spectrum better, consider frequencies up to 0.02 mHz, which correspond to about 13 cycles per week.

```
[Plomb,flomb] = plomb(roomtemp,t1,2e-5,'power');

figure
plot(flomb*60*60*24*7,Plomb)
grid
xlabel('Frequency (cycles/week)')
ylabel('Power (dBW)')
```




The spectrum shows dominant periodicities at 7 cycles per week and 1 cycle per week. This is understandable, given that the data comes from a temperature-controlled building on a seven-day calendar. The spectral line showing a peak at 1 cycle per week indicates that the temperature in the building follows a weekly cycle, with lower temperatures on weekends and higher temperatures during the week. The spectral line of 7 cycles per week indicates that there is also a daily cycle with lower temperatures at night and higher temperatures during the day.

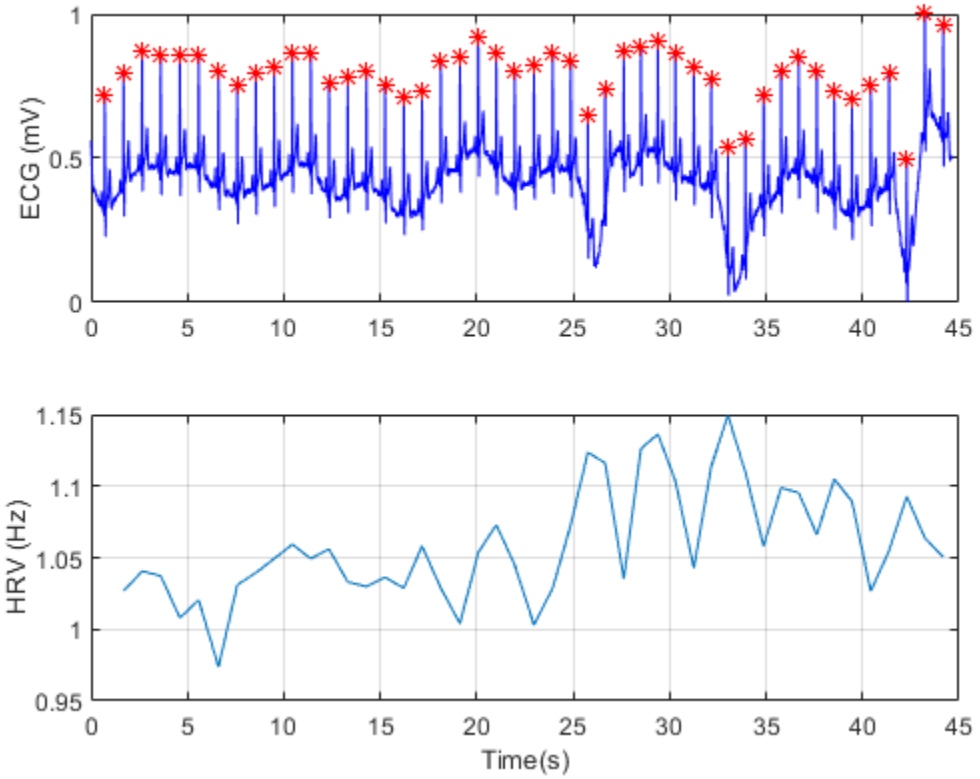
### Spectral Analysis of Signals with Unevenly Spaced Samples

Heart-rate variability (HRV) signals, which represent the physiological variation in time between heartbeats, are typically unevenly sampled because human heart rates are not constant. HRV signals are derived from electrocardiogram (ECG) readings.

The sample points of an HRV signal are located at the R-Peak times of the ECG. The amplitude of each point is computed as the inverse of the time difference between consecutive R-Peaks and is placed at the instant of the second R-Peak.

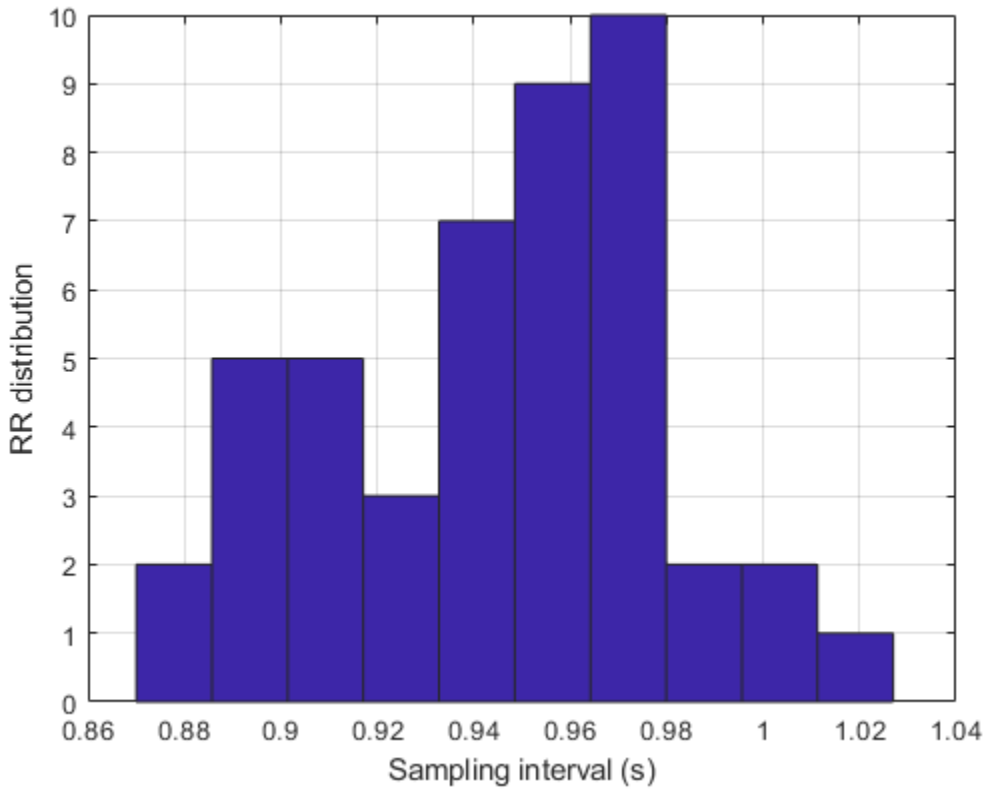
```
% Load the signal, the timestamps, and the sample rate
load('nonuniformdata.mat','ecgsig','t2','Fs')

% Find the ECG peaks
[pks,locs] = findpeaks(ecgsig,Fs, ...
    'MinPeakProminence',0.3,'MinPeakHeight',0.2);

% Determine the RR intervals
RLocsInterval = diff(locs);
```

```
% Derive the HRV signal
tHRV = locs(2:end);
HRV = 1./RLocsInterval;

% Plot the signals
figure
a1 = subplot(2,1,1);
plot(t2,ecgsig,'b',locs,pks,'*r')
grid
a2 = subplot(2,1,2);
plot(tHRV,HRV)
grid
xlabel(a2,'Time(s)')
ylabel(a1,'ECG (mV)')
ylabel(a2,'HRV (Hz)')
```




The varying intervals between the R-peaks cause the sample-time nonuniformity in the HRV data. Consider the peak locations of the signal and plot a histogram of their separations in seconds.

```
figure
hist(RLocsInterval)

grid
xlabel('Sampling interval (s)')
ylabel('RR distribution')
```



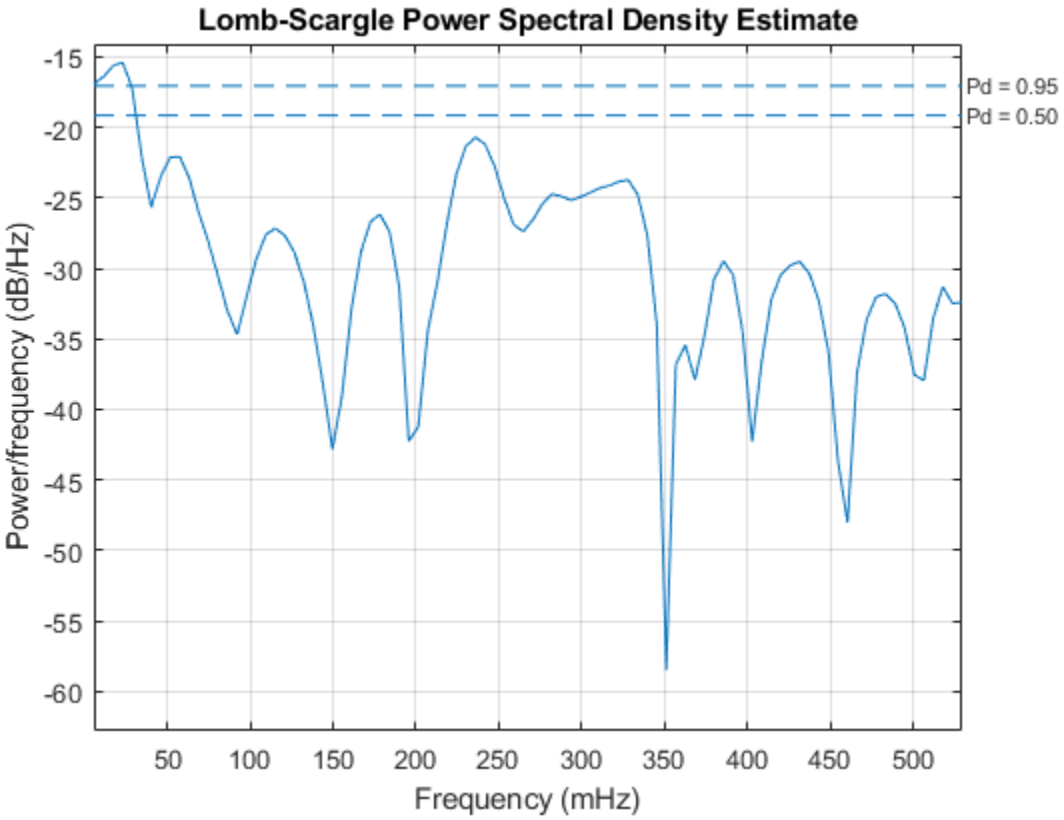

The typical frequency bands of interest in HRV spectra are:

- Very Low Frequency (VLF), from 3.3 to 40 mHz,
- Low Frequency (LF), from 40 to 150 mHz,
- High Frequency (HF), from 150 to 400 mHz.

These bands approximately confine the frequency ranges of the distinct biological regulatory mechanisms that contribute to HRV. Fluctuations in any of these bands have biological significance.

Use `plomb` to calculate the spectrum of the HRV signal.

```
figure
plomb(HRV,tHRV,'Pd',[0.95, 0.5])
```




The dashed lines denote 95% and 50% detection probabilities. These thresholds measure the statistical significance of peaks. The spectrum shows peaks in all three bands of interest listed above. However, only the peak located at 23.2 mHz in the VLF range shows a detection probability 95%, while the other peaks have detection probabilities of less than 50%. The peaks lying below 40 mHz are thought to be due to long-term regulatory mechanisms, such as the thermoregulatory system and hormonal factors.

## See Also

`plomb`

# Linear Prediction and Autoregressive Modeling

This example shows how to compare the relationship between autoregressive modeling and linear prediction. Linear prediction and autoregressive modeling are two different problems that can yield the same numerical results. In both cases, the ultimate goal is to determine the parameters of a linear filter. However, the filter used in each problem is different.

### Introduction

In the case of linear prediction, the intention is to determine an FIR filter that can optimally predict future samples of an autoregressive process based on a linear combination of past samples. The difference between the actual autoregressive signal and the predicted signal is called the prediction error. Ideally, this error is white noise.

For the case of autoregressive modeling, the intention is to determine an all-pole IIR filter, that when excited with white noise produces a signal with the same statistics as the autoregressive process that we are trying to model.

### Generate an AR Signal using an All-Pole Filter with White Noise as Input

Here we use the LPC function and an FIR filter simply to come up with parameters we will use to create the autoregressive signal we will work with. The use of FIR1 and LPC are not critical here. For example, we could replace d with something as simple as [1 1/2 1/3 1/4 1/5 1/6 1/7 1/8] and p0 with something like 1e-6. But the shape of this filter is nicer so we use it instead.

```
b = fir1(1024, .5);
[d,p0] = lpc(b,7);
```

To generate the autoregressive signal, we will excite an all-pole filter with white Gaussian noise of variance p0. Notice that to get variance p0, we must use SQRT(p0) as the 'gain' term in the noise generator.

```
rng(0,'twister'); % Allow reproduction of exact experiment
u = sqrt(p0)*randn(8192,1); % White Gaussian noise with variance p0
```

We now use the white Gaussian noise signal and the all-pole filter to generate an AR signal.

```
x = filter(1,d,u);
```

### Find AR Model from Signal using the Yule-Walker Method

Solving the Yule-Walker equations, we can determine the parameters for an all-pole filter that when excited with white noise will produce an AR signal whose statistics match those of the given signal, x. Once again, this is called autoregressive modeling. In order to solve the Yule-Walker equations, it is necessary to estimate the autocorrelation function of x. The Levinson algorithm is used then to solve the Yule-Walker equations in an efficient manner. The function ARYULE does all this for us.

```
[d1,p1] = aryule(x,7);
```

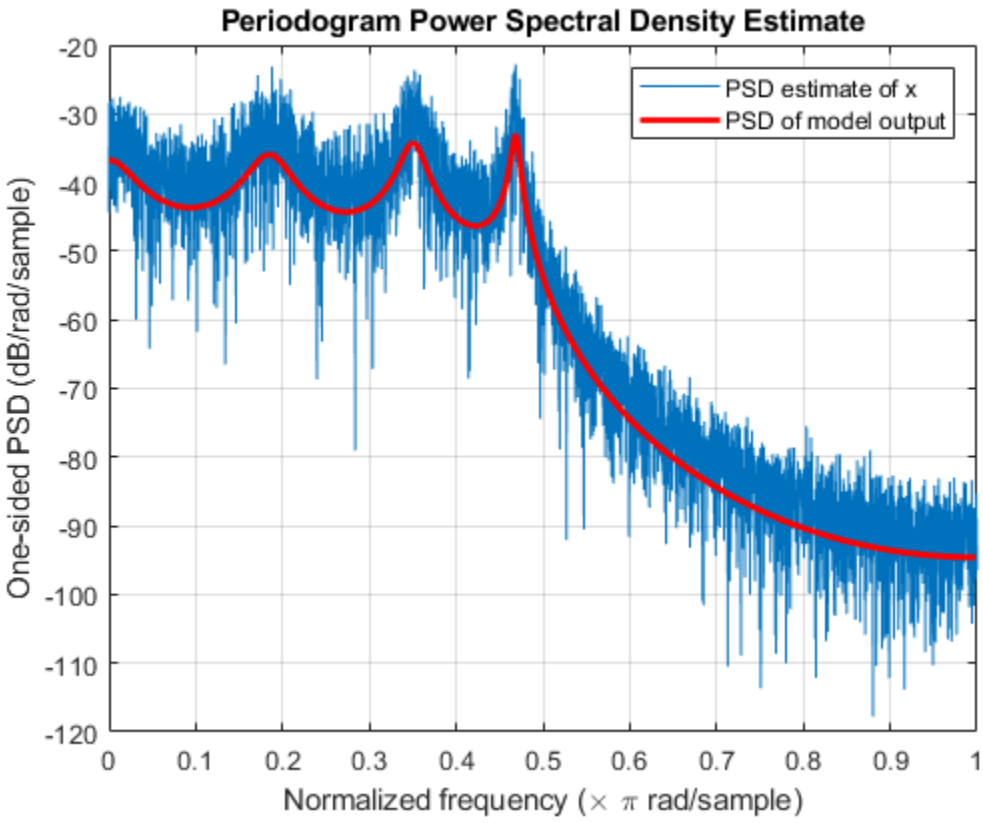
### Compare AR Model with AR Signal

We now would like to compute the frequency response of the all-pole filter we have just used to model the AR signal x. It is well-known that the power spectral density of the output of this filter, when the filter is excited with white Gaussian noise is given by the magnitude-squared of its frequency response multiplied by the variance of the white-noise input. One way to compute this output power spectral density is by using FREQZ as follows:

```
[H1,w1] = freqz(sqrt(p1),d1);
```

In order to get an idea of how well we have modeled the autoregressive signal x, we overlay the power spectral density of the output of the model, computed using FREQZ, with the power spectral density estimate of x, computed using PERIODOGRAM. Notice that the periodogram is scaled by 2*pi and is one-sided. We need to adjust for this in order to compare.

```
periodogram(x)
hold on
hp = plot(w1/pi,20*log10(2*abs(H1)/(2*pi)),'r'); % Scale to make one-sided PSD
hp.LineWidth = 2;
xlabel('Normalized frequency (\times \pi rad/sample)')
ylabel('One-sided PSD (dB/rad/sample)')
legend('PSD estimate of x','PSD of model output')
```




### Use LPC to Perform Linear Prediction

We now turn to the linear prediction problem. Here we try to determine an FIR prediction filter. We use LPC to do so, but the result from LPC requires a little interpretation. LPC returns the coefficients of the entire whitening filter A(z), this filter takes as input the autoregressive signal x and returns as output the prediction error. However, A(z) has the prediction filter embedded in it, in the form B(z) = 1- A(z), where B(z) is the prediction filter. Note that the coefficients and error variance computed with LPC are essentially the same as those computed with ARYULE, but their interpretation is different.

```
[d2,p2] = lpc(x,7);
[d1.',d2.']
```

```
ans = 8×2

    1.0000    1.0000
   -3.5245   -3.5245
    6.9470    6.9470
   -9.2899   -9.2899
    8.9224    8.9224
   -6.1349   -6.1349
    2.8299    2.8299
   -0.6997   -0.6997
```

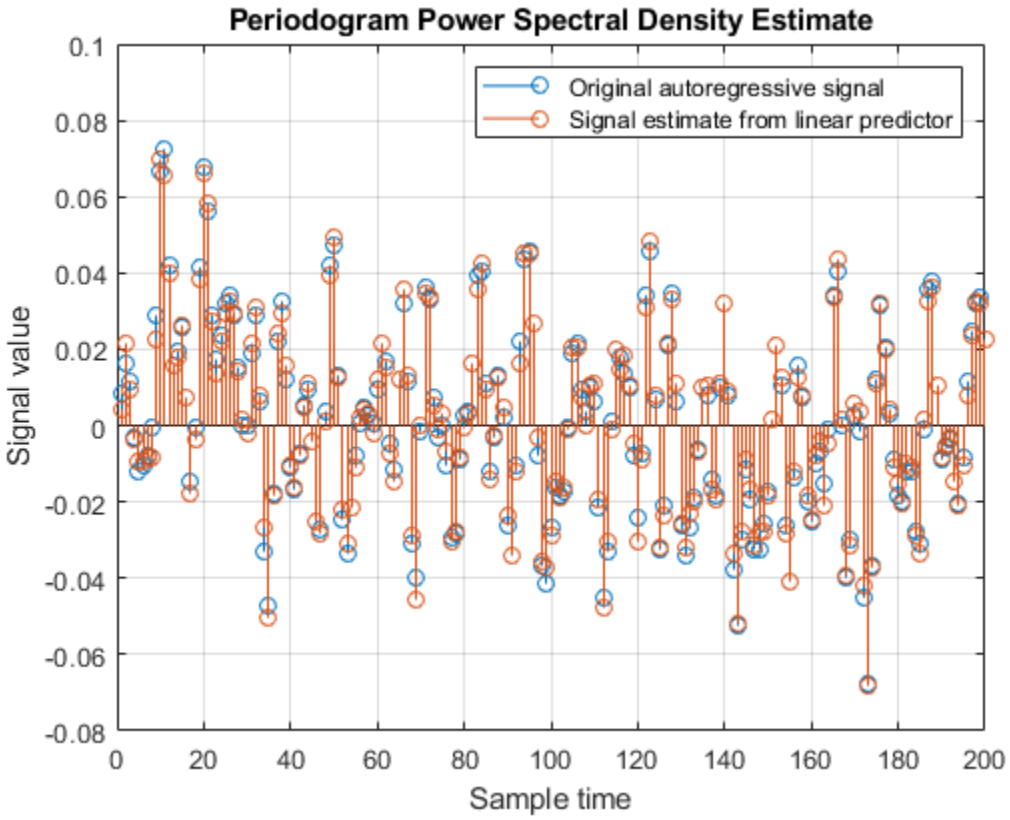
We now extract B(z) from A(z) as described above to use the FIR linear predictor filter to obtain an estimate of future values of the autoregressive signal based on linear combinations of past values.

```
xh = filter(-d2(2:end),1,x);
```

### Compare Actual and Predicted Signals

To get a feeling for what we have done with a 7-tap FIR prediction filter, we plot (200 samples) of the original autoregressive signal along with the signal estimate resulting from the linear predictor keeping in mind the one-sample delay in the prediction filter.

```
cla
stem([x(2:end),xh(1:end-1)])
xlabel('Sample time')
ylabel('Signal value')
legend('Original autoregressive signal','Signal estimate from linear predictor')
axis([0 200 -0.08 0.1])
```




### Compare Prediction Errors

The prediction error power (variance) is returned as the second output from LPC. Its value is (theoretically) the same as the variance of the white noise driving the all-pole filter in the AR modeling problem (p1). Another way of estimating this variance is from the prediction error itself:

```
p3 = norm(x(2:end)-xh(1:end-1),2)^2/(length(x)-1);
```

All of the following values are theoretically the same. The differences are due to the various computation and approximation errors herein.

```
[p0 p1 p2 p3]
```

```
ans = 1×4
10^-5 ×

    0.5127    0.5305    0.5305    0.5068
```

## See Also

`aryule` | `lpc`

# Classify ECG Signals Using Long Short-Term Memory Networks

This example shows how to classify heartbeat electrocardiogram (ECG) data from the PhysioNet 2017 Challenge using deep learning and signal processing. In particular, the example uses Long Short-Term Memory networks and time-frequency analysis.

### Introduction

ECGs record the electrical activity of a person's heart over a period of time. Physicians use ECGs to detect visually if a patient's heartbeat is normal or irregular.

Atrial fibrillation (AFib) is a type of irregular heartbeat that occurs when the heart's upper chambers, the atria, beat out of coordination with the lower chambers, the ventricles.

This example uses ECG data from the PhysioNet 2017 Challenge [1 on page 24-0 ], [2 on page 24-0 ], [3 on page 24-0 ], which is available at https://physionet.org/challenge/2017/. The data consists of a set of ECG signals sampled at 300 Hz and divided by a group of experts into four different classes: Normal (N), AFib (A), Other Rhythm (O), and Noisy Recording (~). This example shows how to automate the classification process using deep learning. The procedure explores a binary classifier that can differentiate Normal ECG signals from signals showing signs of AFib.

This example uses long short-term memory (LSTM) networks, a type of recurrent neural network (RNN) well-suited to study sequence and time-series data. An LSTM network can learn long-term dependencies between time steps of a sequence. The LSTM layer (`lstmLayer` (Deep Learning Toolbox)) can look at the time sequence in the forward direction, while the 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.

To accelerate the training process, run this example on a machine with a GPU. If your machine has a GPU and Parallel Computing Toolbox™, then MATLAB® automatically uses the GPU for training; otherwise, it uses the CPU.

### Load and Examine the Data

Run the `ReadPhysionetData` script to download the data from the PhysioNet website and generate a MAT-file (`PhysionetData.mat`) that contains the ECG signals in the appropriate format. Downloading the data might take a few minutes.

```
ReadPhysionetData
load PhysionetData
```

The loading operation adds two variables to the workspace: `Signals` and `Labels`. `Signals` is a cell array that holds the ECG signals. `Labels` is a categorical array that holds the corresponding ground-truth labels of the signals.

```
Signals(1:5)
```

```
ans=5×1 cell array
    {1×9000  double}
    {1×9000  double}
    {1×18000 double}
    {1×9000  double}
    {1×18000 double}
```

```
Labels(1:5)
```

```
ans = 5×1 categorical
     N
     N
     N
     A
     A
```

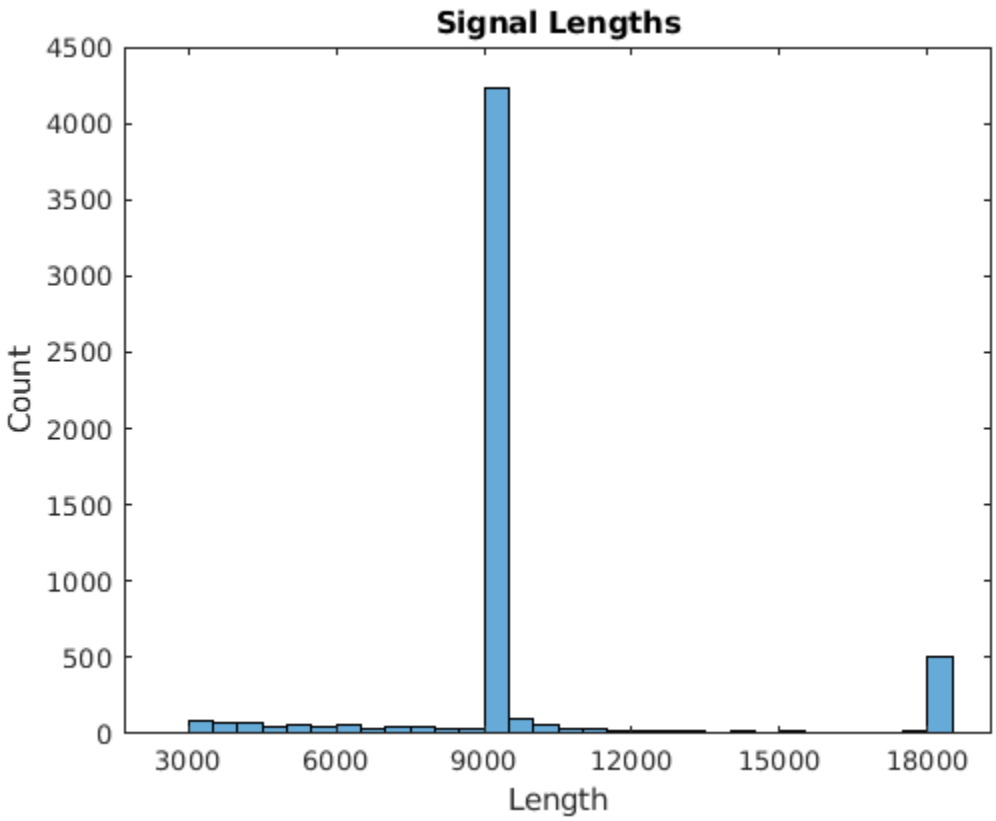
Use the `summary` function to see how many AFib signals and Normal signals are contained in the data.

```
summary(Labels)
```

```
     A       738
     N      5050
```

Generate a histogram of signal lengths. Most of the signals are 9000 samples long.

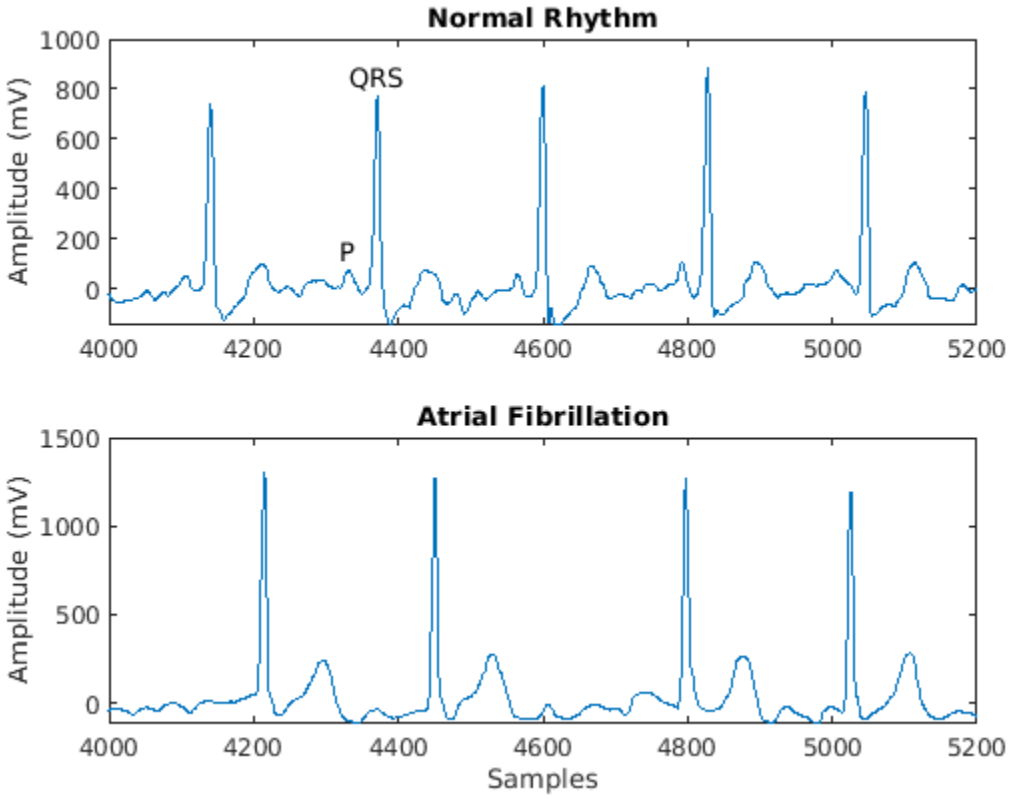
```
L = cellfun(@length,Signals);
h = histogram(L);
xticks(0:3000:18000);
xticklabels(0:3000:18000);
title('Signal Lengths')
xlabel('Length')
ylabel('Count')
```




Visualize a segment of one signal from each class. AFib heartbeats are spaced out at irregular intervals while Normal heartbeats occur regularly. AFib heartbeat signals also often lack a P wave,

which pulses before the QRS complex in a Normal heartbeat signal. The plot of the Normal signal shows a P wave and a QRS complex.

```
normal = Signals{1};
aFib = Signals{4};

subplot(2,1,1)
plot(normal)
title('Normal Rhythm')
xlim([4000,5200])
ylabel('Amplitude (mV)')
text(4330,150,'P','HorizontalAlignment','center')
text(4370,850,'QRS','HorizontalAlignment','center')

subplot(2,1,2)
plot(aFib)
title('Atrial Fibrillation')
xlim([4000,5200])
xlabel('Samples')
ylabel('Amplitude (mV)')
```




### Prepare the Data for Training

During training, the `trainNetwork` function splits the data into mini-batches. The function then pads or truncates signals in the same mini-batch so they all have the same length. Too much padding or truncating can have a negative effect on the performance of the network, because the network might interpret a signal incorrectly based on the added or removed information.

To avoid excessive padding or truncating, apply the `segmentSignals` function to the ECG signals so they are all 9000 samples long. The function ignores signals with fewer than 9000 samples. If a signal has more than 9000 samples, `segmentSignals` breaks it into as many 9000-sample segments as possible and ignores the remaining samples. For example, a signal with 18500 samples becomes two 9000-sample signals, and the remaining 500 samples are ignored.

```
[Signals,Labels] = segmentSignals(Signals,Labels);
```

View the first five elements of the `Signals` array to verify that each entry is now 9000 samples long.

```
Signals(1:5)
```

```
ans=5×1 cell array
    {1×9000 double}
    {1×9000 double}
    {1×9000 double}
    {1×9000 double}
    {1×9000 double}
```

### Train the Classifier Using Raw Signal Data

To design the classifier, use the raw signals generated in the previous section. Split the signals into a training set to train the classifier and a testing set to test the accuracy of the classifier on new data.

Use the `summary` function to show that the ratio of AFib signals to Normal signals is 718:4937, or approximately 1:7.

```
summary(Labels)
```

```
     A       718
     N      4937
```

Because about 7/8 of the signals are Normal, the classifier would learn that it can achieve a high accuracy simply by classifying all signals as Normal. To avoid this bias, augment the AFib data by duplicating AFib signals in the dataset so that there is the same number of Normal and AFib signals. This duplication, commonly called oversampling, is one form of data augmentation used in deep learning.

Split the signals according to their class.

```
afibX = Signals(Labels=='A');
afibY = Labels(Labels=='A');

normalX = Signals(Labels=='N');
normalY = Labels(Labels=='N');
```

Next, use `dividerand` to divide targets from each class randomly into training and testing sets.

```
[trainIndA,~,testIndA] = dividerand(718,0.9,0.0,0.1);
[trainIndN,~,testIndN] = dividerand(4937,0.9,0.0,0.1);

XTrainA = afibX(trainIndA);
YTrainA = afibY(trainIndA);

XTrainN = normalX(trainIndN);
YTrainN = normalY(trainIndN);
```

```
XTestA = afibX(testIndA);
YTestA = afibY(testIndA);

XTestN = normalX(testIndN);
YTestN = normalY(testIndN);
```

Now there are 646 AFib signals and 4443 Normal signals for training. To achieve the same number of signals in each class, use the first 4438 Normal signals, and then use `repmat` to repeat the first 634 AFib signals seven times.

For testing, there are 72 AFib signals and 494 Normal signals. Use the first 490 Normal signals, and then use `repmat` to repeat the first 70 AFib signals seven times. By default, the neural network randomly shuffles the data before training, ensuring that contiguous signals do not all have the same label.

```
XTrain = [repmat(XTrainA(1:634),7,1); XTrainN(1:4438)];
YTrain = [repmat(YTrainA(1:634),7,1); YTrainN(1:4438)];

XTest = [repmat(XTestA(1:70),7,1); XTestN(1:490)];
YTest = [repmat(YTestA(1:70),7,1); YTestN(1:490);];
```

The distribution between Normal and AFib signals is now evenly balanced in both the training set and the testing set.

```
summary(YTrain)
```

```
     A      4438
     N      4438
```

```
summary(YTest)
```

```
     A      490
     N      490
```

### 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`, as it looks at the sequence in both forward and backward directions.

Because the input signals have one dimension each, specify the input size to be sequences of size 1. Specify a bidirectional LSTM layer with an output size of 100 and output the last element of the sequence. This command instructs the bidirectional LSTM layer to map the input time series into 100 features and then prepares the output for the fully connected layer. Finally, specify two classes by including a fully connected layer of size 2, followed by a softmax layer and a classification layer.

```
layers = [ ...
    sequenceInputLayer(1)
    bilstmLayer(100,'OutputMode','last')
    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer
    ]
```

```
layers =
  5x1 Layer array with layers:

     1   ''   Sequence Input          Sequence input with 1 dimensions
```

```
     2   ''   BiLSTM                  BiLSTM with 100 hidden units
     3   ''   Fully Connected         2 fully connected layer
     4   ''   Softmax                 softmax
     5   ''   Classification Output   crossentropyex
```

Next specify the training options for the classifier. Set the `'MaxEpochs'` to 10 to allow the network to make 10 passes through the training data. A `'MiniBatchSize'` of 150 directs the network to look at 150 training signals at a time. An `'InitialLearnRate'` of 0.01 helps speed up the training process. Specify a `'SequenceLength'` of 1000 to break the signal into smaller pieces so that the machine does not run out of memory by looking at too much data at one time. Set '`GradientThreshold`' to 1 to stabilize the training process by preventing gradients from getting too large. Specify `'Plots'` as `'training-progress'` to generate plots that show a graphic of the training progress as the number of iterations increases. Set `'Verbose'` to `false` to suppress the table output that corresponds to the data shown in the plot. If you want to see this table, set `'Verbose'` to `true`.

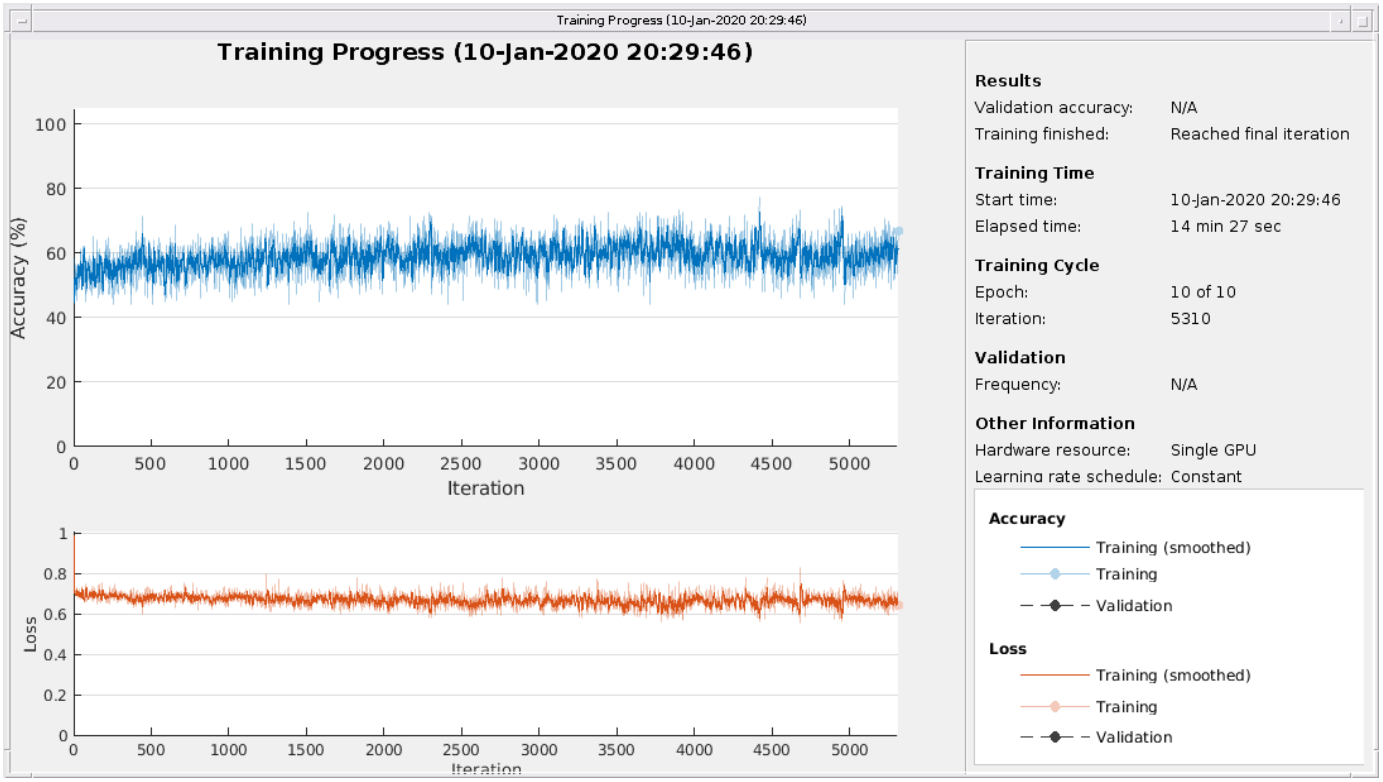
This example uses the adaptive moment estimation (ADAM) solver. ADAM performs better with RNNs like LSTMs than the default stochastic gradient descent with momentum (SGDM) solver.

```
options = trainingOptions('adam', ...
    'MaxEpochs',10, ...
    'MiniBatchSize', 150, ...
    'InitialLearnRate', 0.01, ...
    'SequenceLength', 1000, ...
    'GradientThreshold', 1, ...
    'ExecutionEnvironment',"auto",...
    'plots','training-progress', ...
    'Verbose',false);
```

**Train the LSTM Network**

Train the LSTM network with the specified training options and layer architecture by using `trainNetwork`. Because the training set is large, the training process can take several minutes.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```




The top subplot of the training-progress plot represents the training accuracy, which is the classification accuracy on each mini-batch. When training progresses successfully, this value typically increases towards 100%. The bottom subplot displays the training loss, which is the cross-entropy loss on each mini-batch. When training progresses successfully, this value typically decreases towards zero.

If the training is not converging, the plots might oscillate between values without trending in a certain upward or downward direction. This oscillation means that the training accuracy is not improving and the training loss is not decreasing. This situation can occur from the start of training, or the plots might plateau after some preliminary improvement in training accuracy. In many cases, changing the training options can help the network achieve convergence. Decreasing `MiniBatchSize` or decreasing `InitialLearnRate` might result in a longer training time, but it can help the network learn better.

The classifier's training accuracy oscillates between about 50% and about 60%, and at the end of 10 epochs, it already has taken several minutes to train.

### Visualize the Training and Testing Accuracy

Calculate the training accuracy, which represents the accuracy of the classifier on the signals on which it was trained. First, classify the training data.

```
trainPred = classify(net,XTrain,'SequenceLength',1000);
```

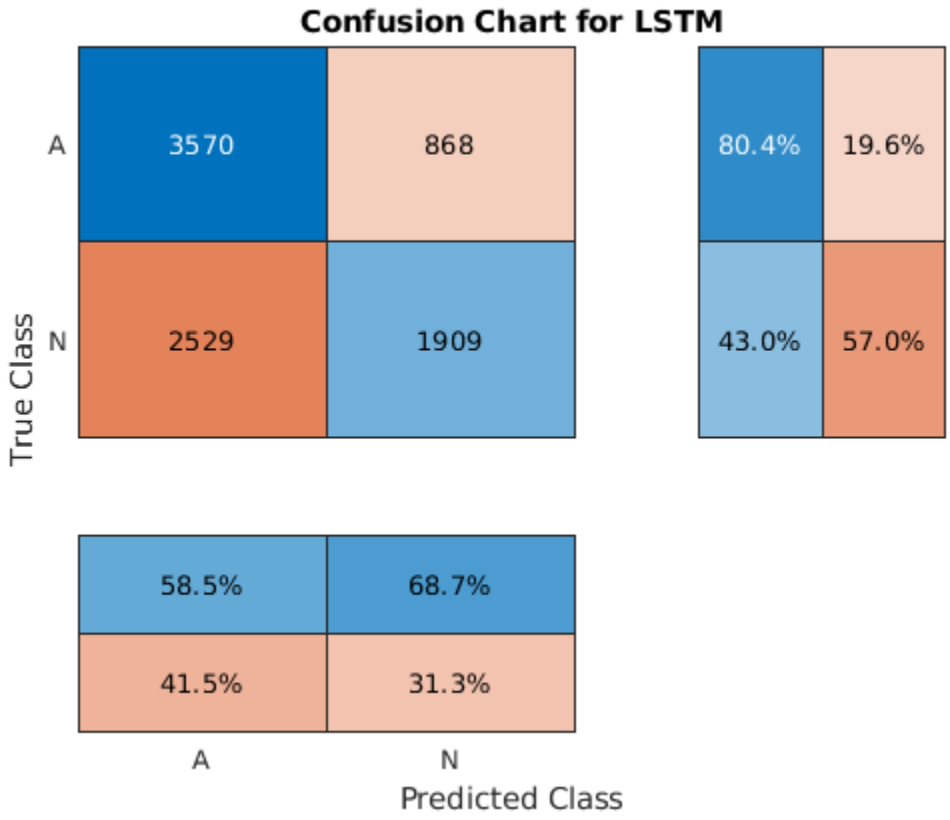
In classification problems, confusion matrices are used to visualize the performance of a classifier on a set of data for which the true values are known. The Target Class is the ground-truth label of the signal, and the Output Class is the label assigned to the signal by the network. The axes labels represent the class labels, AFib (A) and Normal (N).

Use the `confusionchart` command to calculate the overall classification accuracy for the testing data predictions. Specify `'RowSummary'` as `'row-normalized'` to display the true positive rates and false positive rates in the row summary. Also, specify `'ColumnSummary'` as `'column-normalized'` to display the positive predictive values and false discovery rates in the column summary.

```
LSTMAccuracy = sum(trainPred == YTrain)/numel(YTrain)*100
```

```
LSTMAccuracy = 61.7283
```

```
figure
confusionchart(YTrain,trainPred,'ColumnSummary','column-normalized',...
               'RowSummary','row-normalized','Title','Confusion Chart for LSTM');
```




Now classify the testing data with the same network.

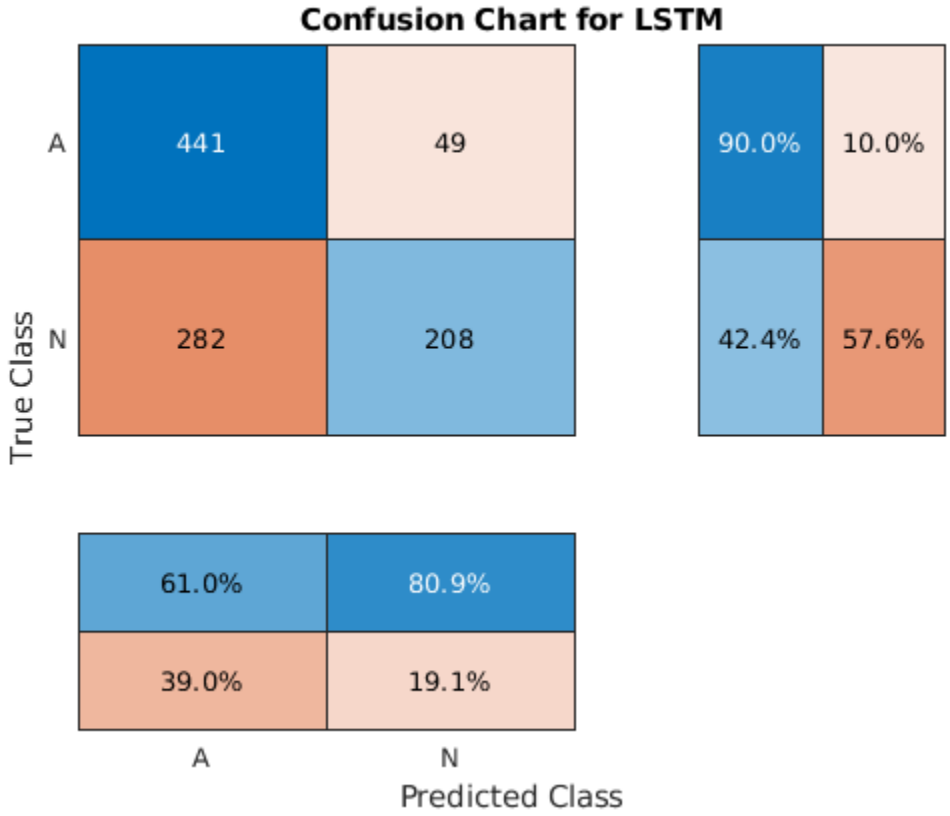
```
testPred = classify(net,XTest,'SequenceLength',1000);
```

Calculate the testing accuracy and visualize the classification performance as a confusion matrix.

```
LSTMAccuracy = sum(testPred == YTest)/numel(YTest)*100
```

```
LSTMAccuracy = 66.2245
```

```
figure
confusionchart(YTest,testPred,'ColumnSummary','column-normalized',...
               'RowSummary','row-normalized','Title','Confusion Chart for LSTM');
```

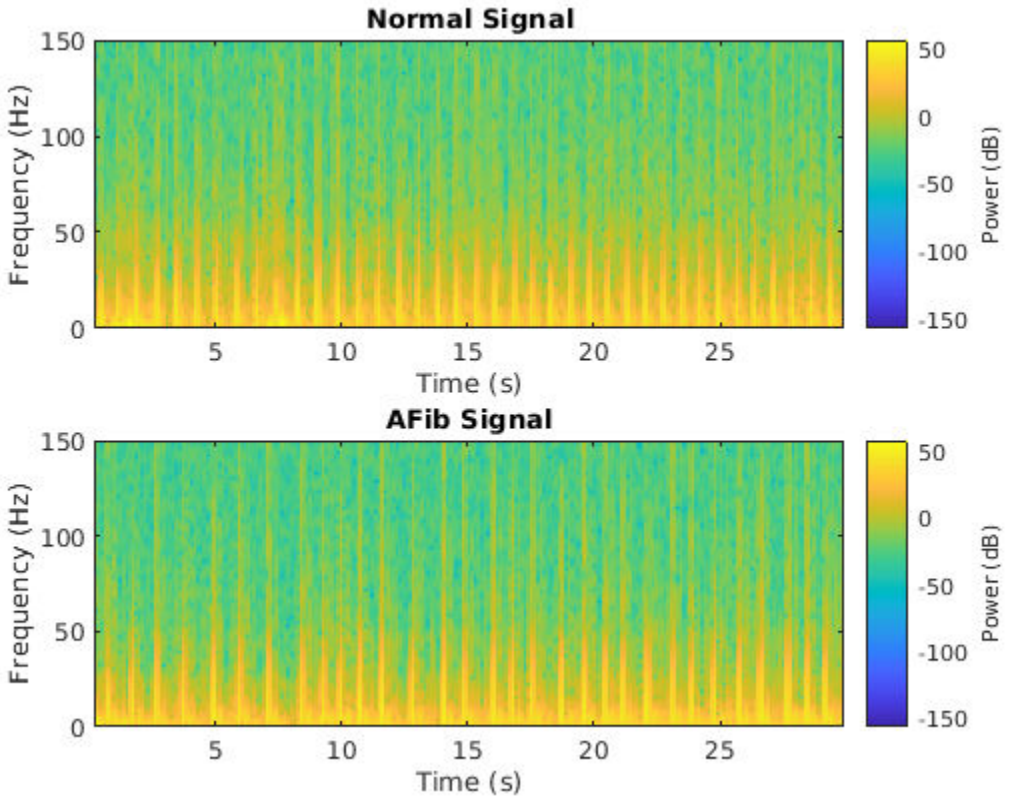



### Improve the Performance with Feature Extraction

Feature extraction from the data can help improve the training and testing accuracies of the classifier. To decide which features to extract, this example adapts an approach that computes time-frequency images, such as spectrograms, and uses them to train convolutional neural networks (CNNs) [4 on page 24-0 ], [5 on page 24-0 ].

Visualize the spectrogram of each type of signal.

```
fs = 300;

figure
subplot(2,1,1);
pspectrum(normal,fs,'spectrogram','TimeResolution',0.5)
title('Normal Signal')

subplot(2,1,2);
pspectrum(aFib,fs,'spectrogram','TimeResolution',0.5)
title('AFib Signal')
```




Because this example uses an LSTM instead of a CNN, it is important to translate the approach so it applies to one-dimensional signals. Time-frequency (TF) moments extract information from the spectrograms. Each moment can be used as a one-dimensional feature to input to the LSTM.

Explore two TF moments in the time domain:

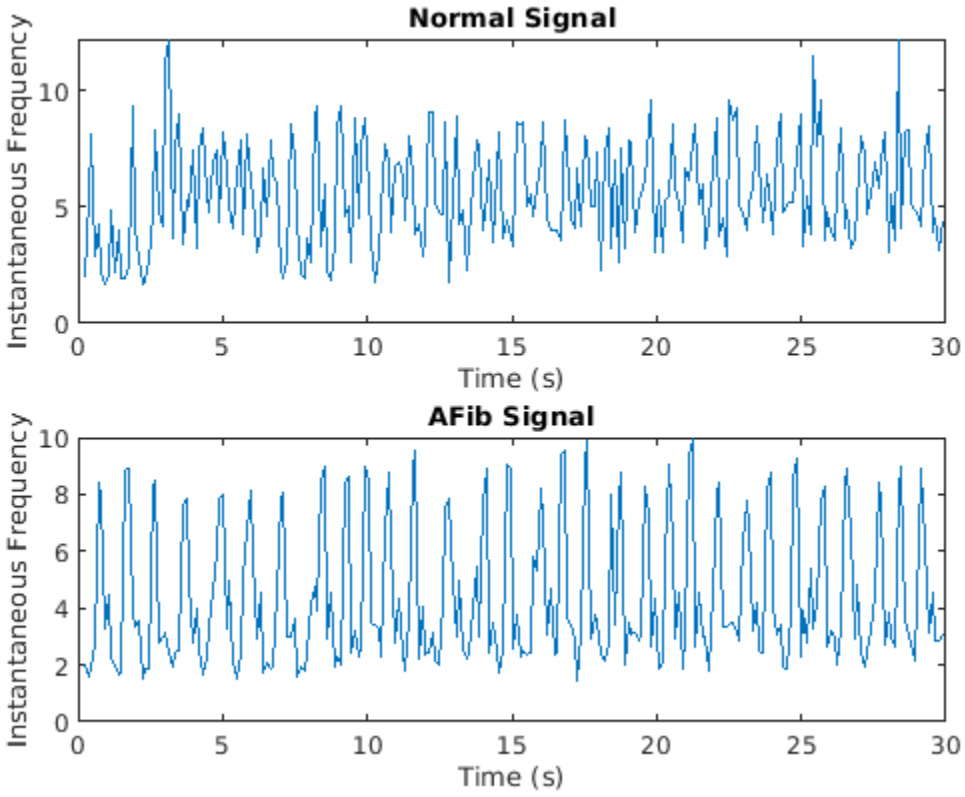
- Instantaneous frequency (`instfreq`)
- Spectral entropy (`pentropy`)

The `instfreq` function estimates the time-dependent frequency of a signal as the first moment of the power spectrogram. The function computes a spectrogram using short-time Fourier transforms over time windows. In this example, the function uses 255 time windows. The time outputs of the function correspond to the centers of the time windows.

Visualize the instantaneous frequency for each type of signal.

```
[instFreqA,tA] = instfreq(aFib,fs);
[instFreqN,tN] = instfreq(normal,fs);

figure
subplot(2,1,1);
plot(tN,instFreqN)
title('Normal Signal')
xlabel('Time (s)')
ylabel('Instantaneous Frequency')
```

```
subplot(2,1,2);
plot(tA,instFreqA)
title('AFib Signal')
xlabel('Time (s)')
ylabel('Instantaneous Frequency')
```




Use `cellfun` to apply the `instfreq` function to every cell in the training and testing sets.

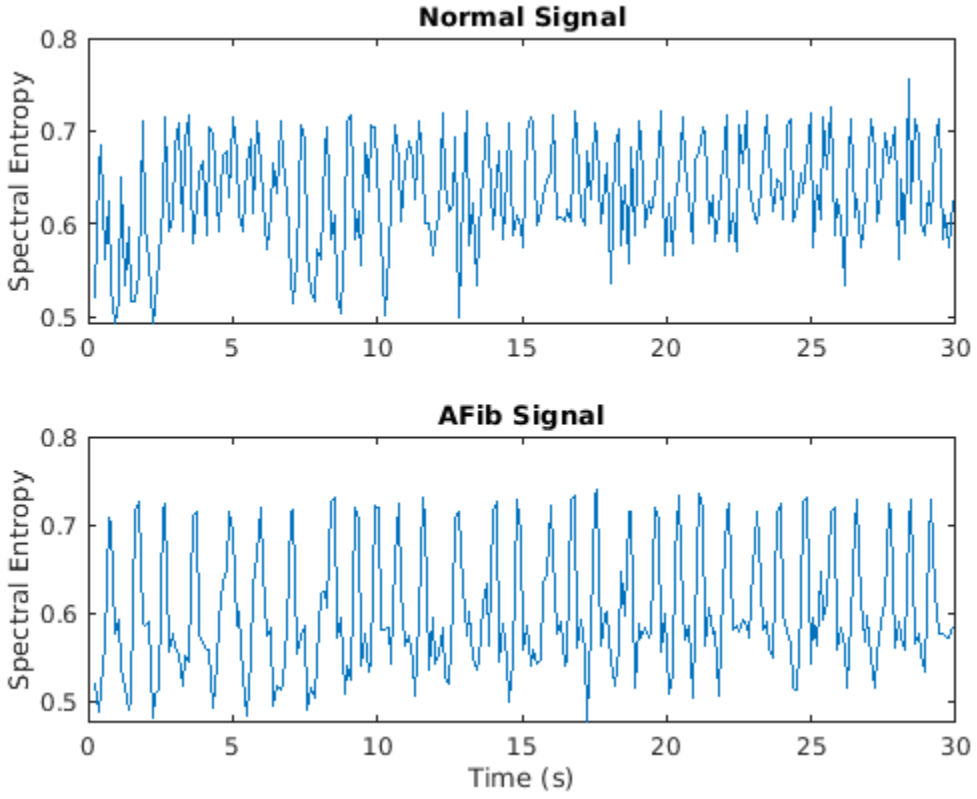
```
instfreqTrain = cellfun(@(x)instfreq(x,fs)',XTrain,'UniformOutput',false);
instfreqTest = cellfun(@(x)instfreq(x,fs)',XTest,'UniformOutput',false);
```

The spectral entropy measures how spiky flat the spectrum of a signal is. A signal with a spiky spectrum, like a sum of sinusoids, has low spectral entropy. A signal with a flat spectrum, like white noise, has high spectral entropy. The `pentropy` function estimates the spectral entropy based on a power spectrogram. As with the instantaneous frequency estimation case, `pentropy` uses 255 time windows to compute the spectrogram. The time outputs of the function correspond to the center of the time windows.

Visualize the spectral entropy for each type of signal.

```
[pentropyA,tA2] = pentropy(aFib,fs);
[pentropyN,tN2] = pentropy(normal,fs);

figure

subplot(2,1,1)
plot(tN2,pentropyN)
```

```
title('Normal Signal')
ylabel('Spectral Entropy')

subplot(2,1,2)
plot(tA2,pentropyA)
title('AFib Signal')
xlabel('Time (s)')
ylabel('Spectral Entropy')
```




Use `cellfun` to apply the `pentropy` function to every cell in the training and testing sets.

```
pentropyTrain = cellfun(@(x)pentropy(x,fs)',XTrain,'UniformOutput',false);
pentropyTest = cellfun(@(x)pentropy(x,fs)',XTest,'UniformOutput',false);
```

Concatenate the features such that each cell in the new training and testing sets has two dimensions, or two features.

```
XTrain2 = cellfun(@(x,y)[x;y],instfreqTrain,pentropyTrain,'UniformOutput',false);
XTest2 = cellfun(@(x,y)[x;y],instfreqTest,pentropyTest,'UniformOutput',false);
```

Visualize the format of the new inputs. Each cell no longer contains one 9000-sample-long signal; now it contains two 255-sample-long features.

```
XTrain2(1:5)
```

```
ans=5×1 cell array
    {2×255 double}
    {2×255 double}
```

```
    {2×255 double}
    {2×255 double}
    {2×255 double}
```

### Standardize the Data

The instantaneous frequency and the spectral entropy have means that differ by almost one order of magnitude. Furthermore, the instantaneous frequency mean might be too high for the LSTM to learn effectively. When a network is fit on data with a large mean and a large range of values, large inputs could slow down the learning and convergence of the network [6 on page 24-0 ].

```
mean(instFreqN)
```

```
ans = 5.5615
```

```
mean(pentropyN)
```

```
ans = 0.6326
```

Use the training set mean and standard deviation to standardize the training and testing sets. Standardization, or z-scoring, is a popular way to improve network performance during training.

```
XV = [XTrain2{:}];
mu = mean(XV,2);
sg = std(XV,[],2);

XTrainSD = XTrain2;
XTrainSD = cellfun(@(x)(x-mu)./sg,XTrainSD,'UniformOutput',false);

XTestSD = XTest2;
XTestSD = cellfun(@(x)(x-mu)./sg,XTestSD,'UniformOutput',false);
```

Show the means of the standardized instantaneous frequency and spectral entropy.

```
instFreqNSD = XTrainSD{1}(1,:);
pentropyNSD = XTrainSD{1}(2,:);

mean(instFreqNSD)
```

```
ans = -0.3211
```

```
mean(pentropyNSD)
```

```
ans = -0.2416
```

### Modify the LSTM Network Architecture

Now that the signals each have two dimensions, it is necessary to modify the network architecture by specifying the input sequence size as 2. Specify a bidirectional LSTM layer with an output size of 100, and output the last element of the sequence. Specify two classes by including a fully connected layer of size 2, followed by a softmax layer and a classification layer.

```
layers = [ ...
    sequenceInputLayer(2)
    bilstmLayer(100,'OutputMode','last')
    fullyConnectedLayer(2)
    softmaxLayer
```

```
    classificationLayer
    ]

layers =
  5x1 Layer array with layers:

     1   ''   Sequence Input          Sequence input with 2 dimensions
     2   ''   BiLSTM                 BiLSTM with 100 hidden units
     3   ''   Fully Connected         2 fully connected layer
     4   ''   Softmax                softmax
     5   ''   Classification Output   crossentropyex
```

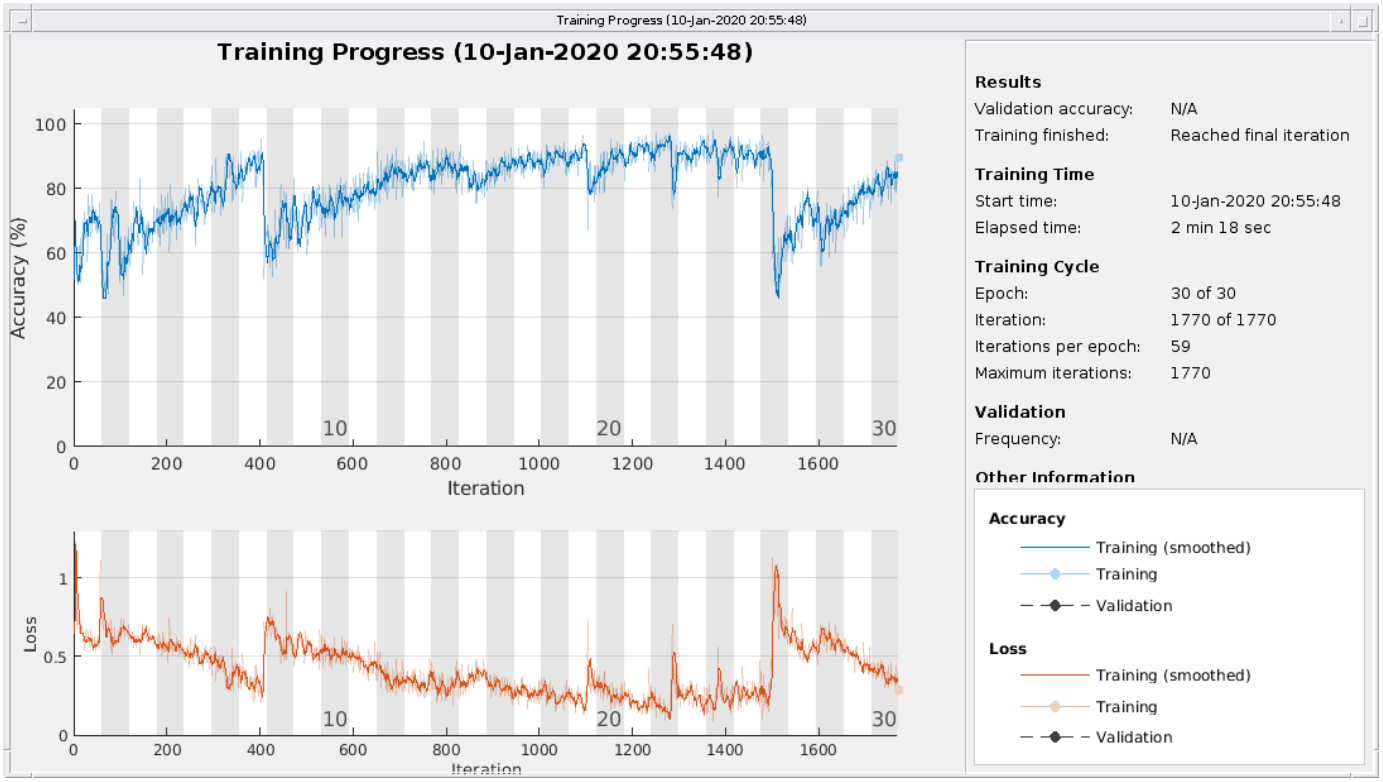
Specify the training options. Set the maximum number of epochs to 30 to allow the network to make 30 passes through the training data.

```
options = trainingOptions('adam', ...
    'MaxEpochs',30, ...
    'MiniBatchSize', 150, ...
    'InitialLearnRate', 0.01, ...
    'GradientThreshold', 1, ...
    'ExecutionEnvironment',"auto",...
    'plots','training-progress', ...
    'Verbose',false);
```

### Train the LSTM Network with Time-Frequency Features

Train the LSTM network with the specified training options and layer architecture by using `trainNetwork`.

```
net2 = trainNetwork(XTrainSD,YTrain,layers,options);
```



There is a great improvement in the training accuracy. The cross-entropy loss trends towards 0. Furthermore, the time required for training decreases because the TF moments are shorter than the raw sequences.

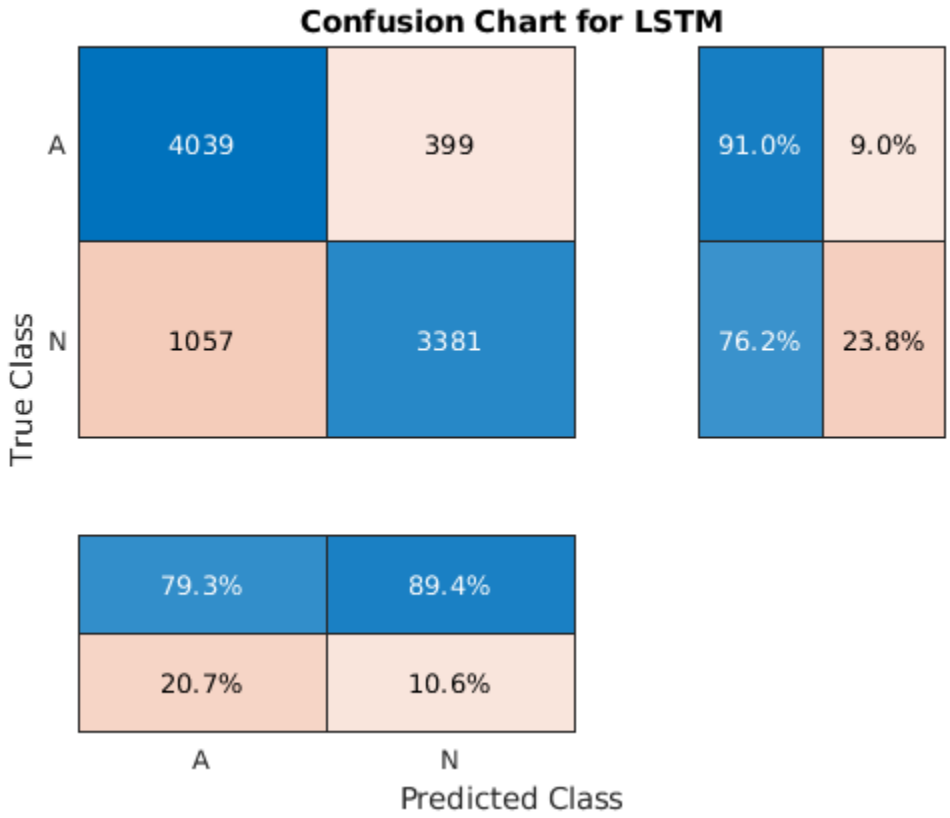
### Visualize the Training and Testing Accuracy

Classify the training data using the updated LSTM network. Visualize the classification performance as a confusion matrix.

```
trainPred2 = classify(net2,XTrainSD);
LSTMAccuracy = sum(trainPred2 == YTrain)/numel(YTrain)*100
```

```
LSTMAccuracy = 83.5962
```

```
figure
confusionchart(YTrain,trainPred2,'ColumnSummary','column-normalized',...
              'RowSummary','row-normalized','Title','Confusion Chart for LSTM');
```

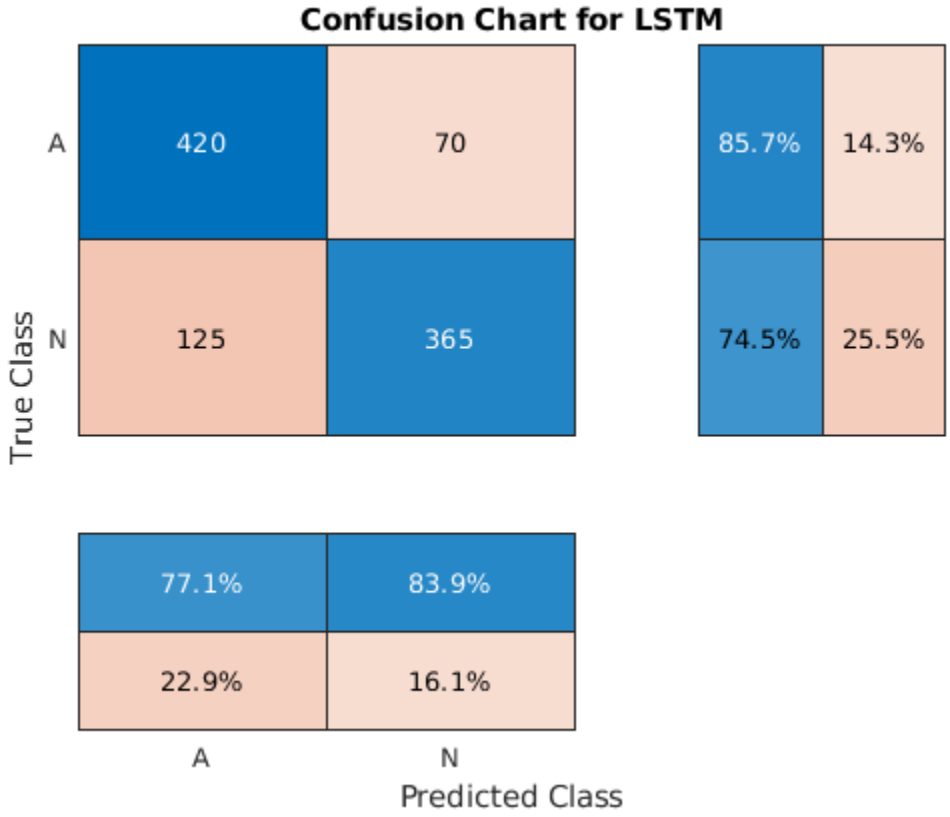



Classify the testing data with the updated network. Plot the confusion matrix to examine the testing accuracy.

```
testPred2 = classify(net2,XTestSD);

LSTMAccuracy = sum(testPred2 == YTest)/numel(YTest)*100
```

```
LSTMAccuracy = 80.1020
```

```
figure
confusionchart(YTest,testPred2,'ColumnSummary','column-normalized',...
               'RowSummary','row-normalized','Title','Confusion Chart for LSTM');
```




### Conclusion

This example shows how to build a classifier to detect atrial fibrillation in ECG signals using an LSTM network. The procedure uses oversampling to avoid the classification bias that occurs when one tries to detect abnormal conditions in populations composed mainly of healthy patients. Training the LSTM network using raw signal data results in a poor classification accuracy. Training the network using two time-frequency-moment features for each signal significantly improves the classification performance and also decreases the training time.

### References

[1] *AF Classification from a Short Single Lead ECG Recording: the PhysioNet/Computing in Cardiology Challenge, 2017*. https://physionet.org/challenge/2017/

[2] Clifford, Gari, Chengyu Liu, Benjamin Moody, Li-wei H. Lehman, Ikaro Silva, Qiao Li, Alistair Johnson, and Roger G. Mark. "AF Classification from a Short Single Lead ECG Recording: The PhysioNet Computing in Cardiology Challenge 2017." *Computing in Cardiology* (Rennes: IEEE). Vol. 44, 2017, pp. 1–4.

[3] Goldberger, A. L., L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals". *Circulation*. Vol. 101, No. 23, 13 June 2000, pp. e215–e220. http://circ.ahajournals.org/content/101/23/e215.full

[4] Pons, Jordi, Thomas Lidy, and Xavier Serra. "Experimenting with Musically Motivated Convolutional Neural Networks". *14th International Workshop on Content-Based Multimedia Indexing (CBMI)*. June 2016.

[5] Wang, D. "Deep learning reinvents the hearing aid," *IEEE Spectrum*, Vol. 54, No. 3, March 2017, pp. 32–37. doi: 10.1109/MSPEC.2017.7864754.

[6] Brownlee, Jason. *How to Scale Data for Long Short-Term Memory Networks in Python*. 7 July 2017. https://machinelearningmastery.com/how-to-scale-data-for-long-short-term-memory-networks-in-python/.

## See Also

**Functions**

`bilstmLayer` | `instfreq` | `lstmLayer` | `pentropy` | `trainNetwork` | `trainingOptions`

## More About

- “Long Short-Term Memory Networks” (Deep Learning Toolbox)

# Waveform Segmentation Using Deep Learning

This example shows how to segment human electrocardiogram (ECG) signals using recurrent deep learning networks and time-frequency analysis.

### Introduction

The electrical activity in the human heart can be measured as a sequence of amplitudes away from a baseline signal. For a single normal heartbeat cycle, the ECG signal can be divided into the following beat morphologies [1 on page 24-0 ]:

- P wave — A small deflection before the QRS complex representing atrial depolarization
- QRS complex — Largest-amplitude portion of the heartbeat
- T wave — A small deflection after the QRS complex representing ventricular repolarization

The segmentation of these regions of ECG waveforms can provide the basis for measurements useful for assessing the overall health of the human heart and the presence of abnormalities [2 on page 24-0 ]. Manually annotating each region of the ECG signal can be a tedious and time-consuming task. Signal processing and deep learning methods potentially can help streamline and automate region-of-interest annotation.

This example uses ECG signals from the publicly available QT Database [3 on page 24-0 ] [4 on page 24-0 ]. The data consists of roughly 15 minutes of ECG recordings, with a sample rate of 250 Hz, measured from a total of 105 patients. To obtain each recording, the examiners placed two electrodes on different locations on a patient's chest, resulting in a two-channel signal. The database provides signal region labels generated by an automated expert system [2 on page 24-0 ]. This example aims to use a deep learning solution to provide a label for every ECG signal sample according to the region where the sample is located. This process of labeling regions of interest across a signal is often referred to as *waveform segmentation*.

To train a deep neural network to classify signal regions, you can use a Long Short-Term Memory (LSTM) network. This example shows how signal preprocessing techniques and time-frequency analysis can be used to improve LSTM segmentation performance. In particular, the example uses the Fourier synchrosqueezed transform to represent the nonstationary behavior of the ECG signal.

### Download and Prepare the Data

Each channel of the 105 two-channel ECG signals was labeled independently by the automated expert system and is treated independently, for a total of 210 ECG signals that were stored together with the region labels in 210 MAT-files. The files are available at the following location: https://www.mathworks.com/supportfiles/SPT/data/QTDatabaseECGData.zip.

Download the data files into your temporary directory, whose location is specified by MATLAB®'s `tempdir` command. If you want to place the data files in a folder different from `tempdir`, change the directory name in the subsequent instructions.

```
% Download the data
dataURL = 'https://www.mathworks.com/supportfiles/SPT/data/QTDatabaseECGData1.zip';
datasetFolder = fullfile(tempdir,'QTDataset');
zipFile = fullfile(tempdir,'QTDatabaseECGData.zip');
if ~exist(datasetFolder,'dir')
     websave(zipFile,dataURL);
     unzip(zipFile,tempdir);
end
```

The `unzip` operation creates the `QTDatabaseECGData` folder in your temporary directory with 210 MAT-files in it. Each file contains an ECG signal in variable `ecgSignal` and a table of region labels in variable `signalRegionLabels`. Each file also contains the sample rate of the signal in variable `Fs`. In this example all signals have a sample rate of 250 Hz.

Create a signal datastore to access the data in the files. This example assumes the dataset has been stored in your temporary directory under the `QTDatabaseECGData` folder. If this is not the case, change the path to the data in the code below. Specify the signal variable names you want to read from each file using the `SignalVariableNames` parameter.

```
sds = signalDatastore(datasetFolder,'SignalVariableNames',["ecgSignal","signalRegionLabels"])
```

```
sds =
  signalDatastore with properties:

                      Files:{
                             '/tmp/QTDataset/ecg1.mat';
                             '/tmp/QTDataset/ecg10.mat';
                             '/tmp/QTDataset/ecg100.mat'
                              ... and 207 more
                             }
    AlternateFileSystemRoots: [0×0 string]
                    ReadSize: 1
         SignalVariableNames: ["ecgSignal"    "signalRegionLabels"]
```

The datastore returns a two-element cell array with an ECG signal and a table of region labels each time you call the `read` function. Use the `preview` function of the datastore to see that the content of the first file is a 225,000 samples long ECG signal and a table containing 3385 region labels.

```
data = preview(sds)
```

```
data=2×1 cell array
    {225000×1 double}
    {  3385×2 table }
```

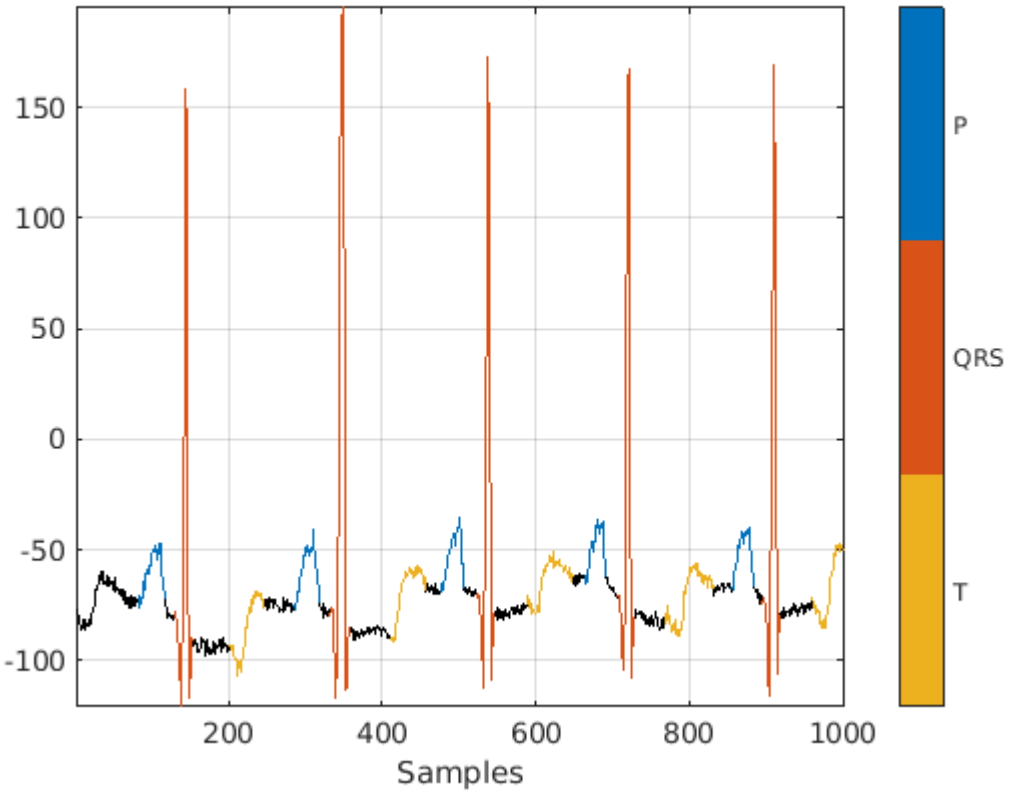
Look at the first few rows of the region labels table and observe that each row contains the region limit indices and the region class value (P, T, or QRS).

```
head(data{2})
```

```
ans=8×2 table
    ROILimits     Value
    _________    _____

     83    117     P
    130    153     QRS
    201    246     T
    285    319     P
    332    357     QRS
    412    457     T
    477    507     P
    524    547     QRS
```

Visualize the labels for the first 1000 samples using a `signalMask` object.

```
M = signalMask(data{2});
plotsigroi(M,data{1}(1:1000))
```




The usual machine learning classification procedure is the following:

1. Divide the database into training and testing datasets.
2. Train the network using the training dataset.
3. Use the trained network to make predictions on the testing dataset.

The network is trained with 70% of the data and tested with the remaining 30%.

For reproducible results, reset the random number generator. Use the `dividerand` function to get random indices to shuffle the files, and the `subset` function of `signalDatastore` to divide the data into training and testing datastores.

```
rng default
[trainIdx,~,testIdx] = dividerand(numel(sds.Files),0.7,0,0.3);
trainDs = subset(sds,trainIdx);
testDs = subset(sds,testIdx);
```

In this segmentation problem, the input to the LSTM network is an ECG signal and the output is a sequence or mask of labels with the same length as the input signal. The network task is to label each signal sample with the name of the region it belongs to. For this reason, it is necessary to transform the region labels on the dataset to sequences containing one label per signal sample. Use a transformed datastore and the `getmask` helper function to transform the region labels. The `getmask` function adds a label category, `"n/a"`, to label samples that do not belong to any region of interest.

```
type getmask.m
```

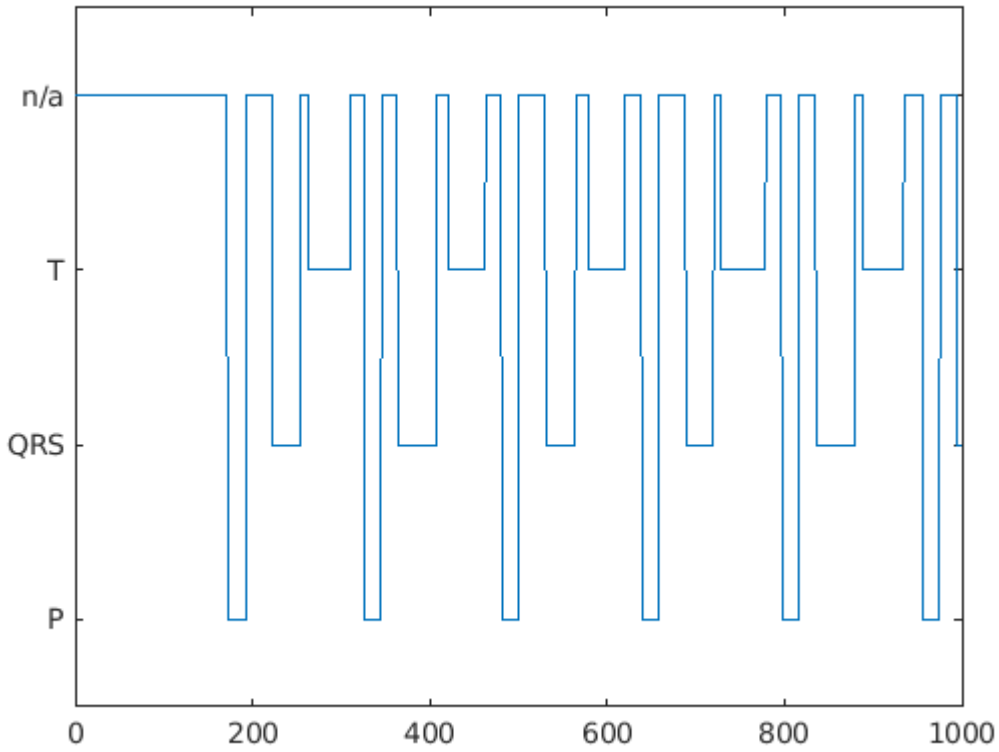
```
function outputCell = getmask(inputCell)
%GETMASK Convert region labels to a mask of labels of size equal to the
%size of the input ECG signal.
%
%   inputCell is a two-element cell array containing an ECG signal vector
%   and a table of region labels.
%
%   outputCell is a two-element cell array containing the ECG signal vector
%   and a categorical label vector mask of the same length as the signal.

% Copyright 2020 The MathWorks, Inc.

sig = inputCell{1};
roiTable = inputCell{2};
L = length(sig);
M = signalMask(roiTable);

% Get categorical mask and give priority to QRS regions when there is overlap
mask = catmask(M,L,'OverlapAction','prioritizeByList','PriorityList',[2 1 3]);

% Set missing values to "n/a"
mask(ismissing(mask)) = "n/a";

outputCell = {sig,mask};
end
```

Preview the transformed datastore to observe that it returns a signal vector and a label vector of equal lengths. Plot the first 1000 element of the categorical mask vector.

```
trainDs = transform(trainDs, @getmask);
testDs = transform(testDs, @getmask);

transformedData = preview(trainDs)
```

```
transformedData=1×2 cell array
    {224993×1 double}    {224993×1 categorical}
```

```
plot(transformedData{2}(1:1000))
```




Passing very long input signals into the LSTM network can result in estimation performance degradation and excessive memory usage. To avoid these effects, break the ECG signals and their corresponding label masks using a transformed datastore and the `resizeData` helper function. The helper function creates as many 5000-sample segments as possible and discards the remaining samples. A preview of the output of the transformed datastore shows that the first ECG signal and its label mask are broken into 5000-sample segments. Note that preview of the transformed datastore only shows the first 8 elements of the otherwise `floor(224993/5000)` = 44 element cell array that would result if we called the datastore `read` function.

```
trainDs = transform(trainDs,@resizeData);
testDs = transform(testDs,@resizeData);
preview(trainDs)
```

```
ans=8×2 cell array
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
```

### Choose to Train Networks or Download Pre-Trained Networks

The next sections of this example compare three different approaches to train LSTM networks. Due to the large size of the dataset, the training process of each network may take several minutes. If your machine has a GPU and Parallel Computing Toolbox™, then MATLAB automatically uses the GPU for faster training. Otherwise, it uses the CPU.

You can skip the training steps and download the pre-trained networks using the selector below. If you want to train the networks as the example runs, select 'Train Networks'. If you want to skip the training steps, select 'Download Networks' and a file containing all three pre-trained networks - `rawNet`, `filteredNet`, and `fsstNet-` will be downloaded into your temporary directory, whose location is specified by MATLAB®'s `tempdir` command. If you want to place the downloaded file in a folder different from `tempdir`, change the directory name in the subsequent instructions.

```
actionFlag = Train networks;
if actionFlag == "Download networks"
    % Download the pre-trained networks
    dataURL = 'https://ssd.mathworks.com/supportfiles/SPT/data/QTDatabaseECGSegmentationNetworks
    modelsFolder = fullfile(tempdir,'QTDatabaseECGSegmentationNetworks');
    modelsFile = fullfile(modelsFolder,'trainedNetworks.mat');
    zipFile = fullfile(tempdir,'QTDatabaseECGSegmentationNetworks.zip');
    if ~exist(modelsFolder,'dir')
        websave(zipFile,dataURL);
        unzip(zipFile,fullfile(tempdir,'QTDatabaseECGSegmentationNetworks'));
    end
    load(modelsFile)
end
```

Results between the downloaded networks and newly trained networks may vary slightly since the networks are trained using random initial weights.

### Input Raw ECG Signals Directly into the LSTM Network

First, train an LSTM network using the raw ECG signals from the training dataset.

Define the network architecture before training. Specify a `sequenceInputLayer` of size 1 to accept one-dimensional time series. Specify an LSTM layer with the `'sequence'` output mode to provide classification for each sample in the signal. Use 200 hidden nodes for optimal performance. Specify a `fullyConnectedLayer` with an output size of 4, one for each of the waveform classes. Add a `softmaxLayer` and a `classificationLayer` to output the estimated labels.

```
layers = [ ...
    sequenceInputLayer(1)
    lstmLayer(200,'OutputMode','sequence')
    fullyConnectedLayer(4)
    softmaxLayer
    classificationLayer];
```

Choose options for the training process that ensure good network performance. Refer to the `trainingOptions` (Deep Learning Toolbox) documentation for a description of each parameter.

```
options = trainingOptions('adam', ...
    'MaxEpochs',10, ...
    'MiniBatchSize',50, ...
    'InitialLearnRate',0.01, ...
    'LearnRateDropPeriod',3, ...
```

```
    'LearnRateSchedule','piecewise', ...
    'GradientThreshold',1, ...
    'Plots','training-progress',...
    'shuffle','every-epoch',...
    'Verbose',0,...
    'DispatchInBackground',true);
```

Because the entire training dataset fits in memory, it is possible to use the `tall` function of the datastore to transform the data in parallel, if Parallel Computing Toolbox™ is available, and then gather it into the workspace. Neural network training is iterative. At every iteration, the datastore reads data from files and transforms the data before updating the network coefficients. If the data fits into the memory of your computer, importing the data into the workspace enables faster training because the data is read and transformed only once. Note that if the data does not fit in memory, you must to pass the datastore into the training function, and the transformations are performed at every training epoch.

Create tall arrays for both the training and test sets. Depending on your system, the number of workers in the parallel pool that MATLAB creates may be different.

```
tallTrainSet = tall(trainDs);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 8).
```

```
tallTestSet = tall(testDs);
```

Now call the `gather` function of the tall arrays to compute the transformations over the entire dataset and obtain cell arrays with the training and test signals and labels.

```
 trainData = gather(tallTrainSet);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 11 sec
Evaluation completed in 12 sec
```

```
 trainData(1,:)
```

```
ans=1×2 cell array
    {1×5000 double}    {1×5000 categorical}
```

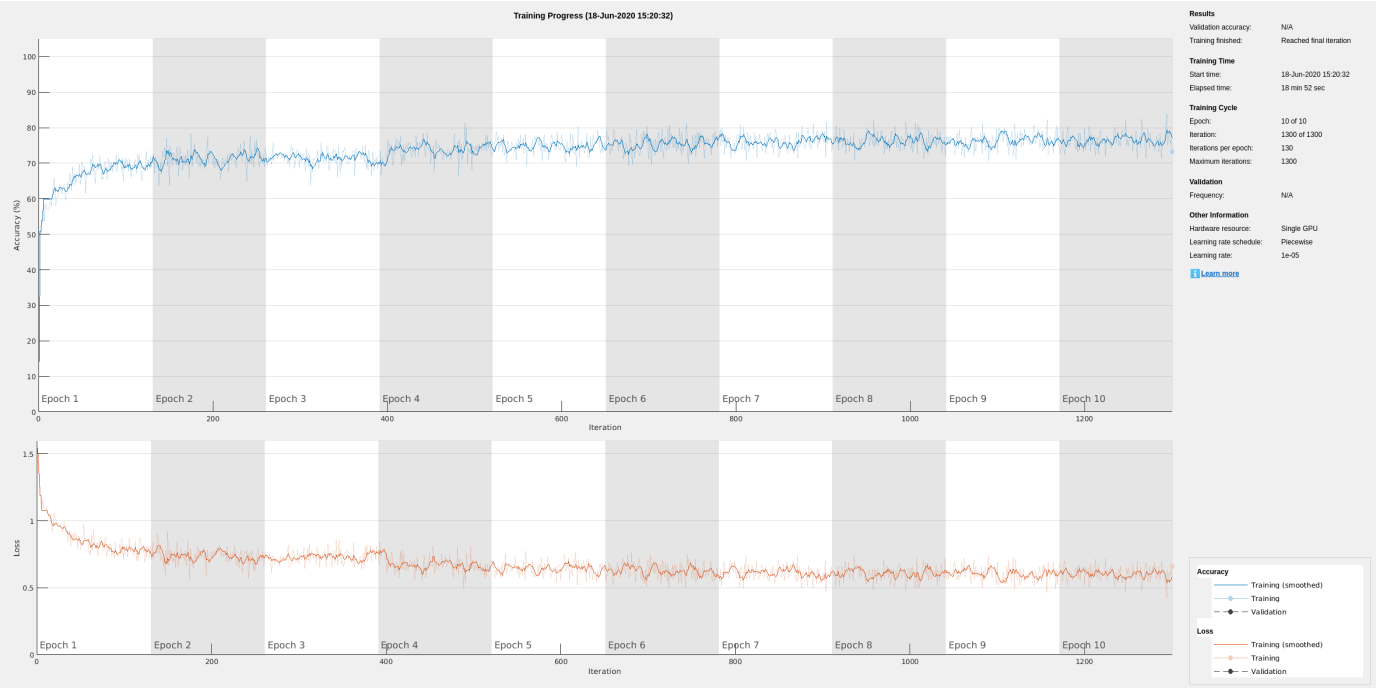
```
 testData = gather(tallTestSet);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 2.9 sec
Evaluation completed in 3.1 sec
```

**Train Network**

Use the `trainNetwork` command to train the LSTM network.

```
if actionFlag == "Train networks"
     rawNet = trainNetwork(trainData(:,1),trainData(:,2),layers,options);
end
```




The training accuracy and loss subplots in the figure track the training progress across all iterations. Using the raw signal data, the network correctly classifies about 77% of the samples as belonging to a P wave, a QRS complex, a T wave, or an unlabeled region `"n/a"`.

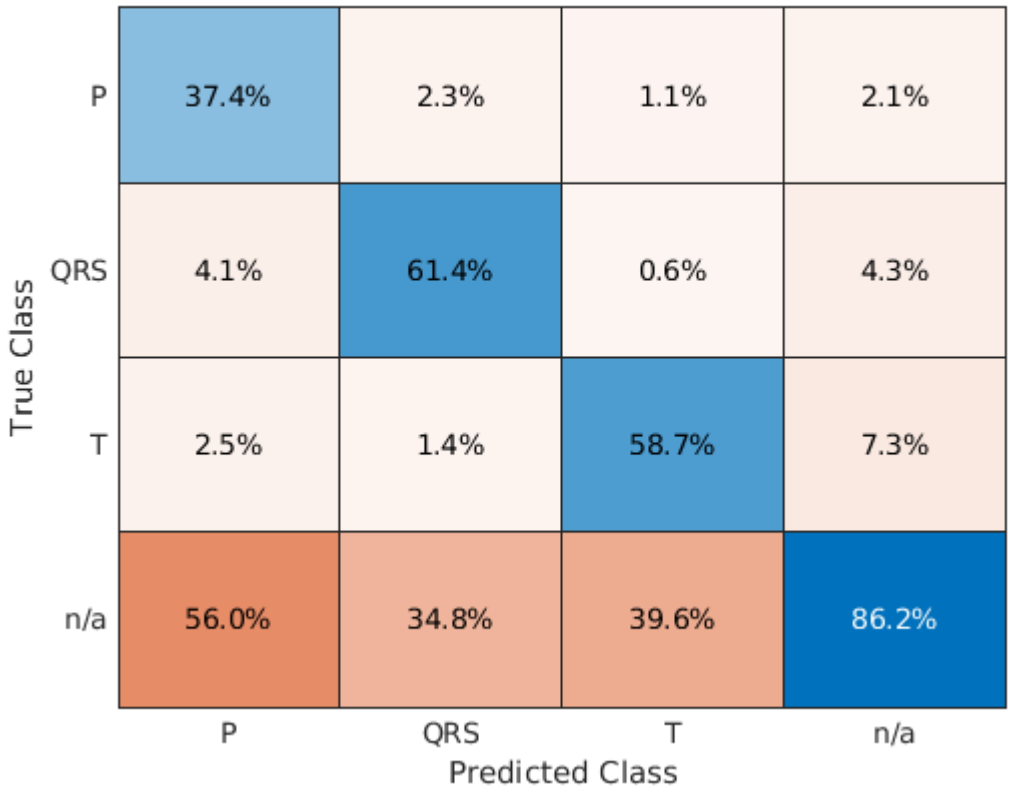
### Classify Testing Data

Classify the testing data using the trained LSTM network and the `classify` command. Specify a mini-batch size of 50 to match the training options.

```
predTest = classify(rawNet,testData(:,1),'MiniBatchSize',50);
```

A confusion matrix provides an intuitive and informative means to visualize classification performance. Use the `confusionchart` command to calculate the overall classification accuracy for the testing data predictions. For each input, convert the cell array of categorical labels to a row vector. Specify a column-normalized display to view results as percentages of samples for each class.

```
confusionchart([predTest{:}],[testData{:,2}],'Normalization','column-normalized');
```




Using the raw ECG signal as input to the network, only about 60% of T-wave samples, 40% of P-wave samples, and 60% of QRS-complex samples were correct. To improve performance, apply some knowledge of the ECG signal characteristics prior to input to the deep learning network, for instance the baseline wandering caused by a patient's respiratory motion.

**Apply Filtering Methods to Remove Baseline Wander and High-Frequency Noise**

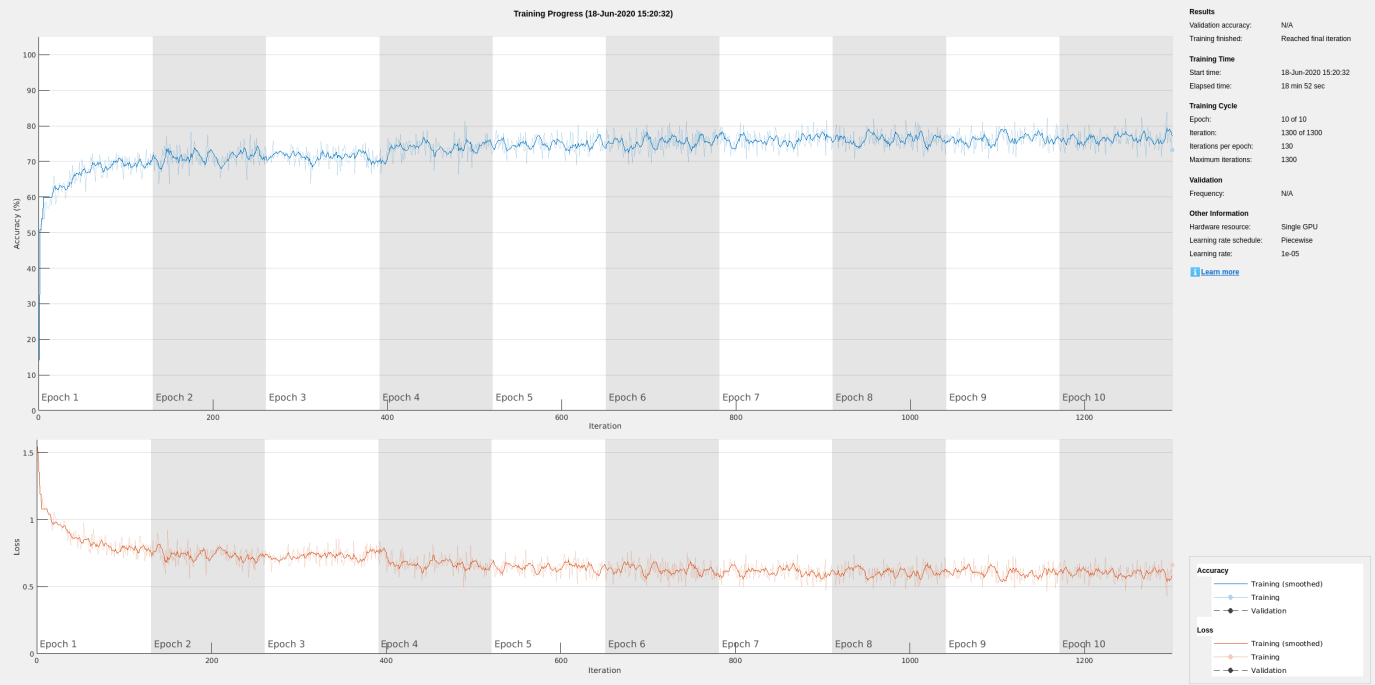
The three beat morphologies occupy different frequency bands. The spectrum of the QRS complex typically has a center frequency around 10–25 Hz, and its components lie below 40 Hz. The P and T waves occur at even lower frequencies: P-wave components are below 20 Hz, and T-wave components are below 10 Hz [5 on page 24-0 ].

Baseline wander is a low-frequency (< 0.5 Hz) oscillation caused by the patient's breathing motion. This oscillation is independent from the beat morphologies and does not provide meaningful information [6 on page 24-0 ].

Design a bandpass filter with passband frequency range of [0.5, 40] Hz to remove the wander and any high frequency noise. Removing these components improves the LSTM training because the network does not learn irrelevant features. Use `cellfun` on the tall data cell arrays to filter the dataset in parallel.

```
% Bandpass filter design
hFilt = designfilt('bandpassiir', 'StopbandFrequency1',0.4215,'PassbandFrequency1', 0.5, ...
    'PassbandFrequency2',40,'StopbandFrequency2',53.345,...
    'StopbandAttenuation1',60,'PassbandRipple',0.1,'StopbandAttenuation2',60,...
    'SampleRate',250,'DesignMethod','ellip');
```

```
% Create tall arrays from the transformed datastores and filter the signals
tallTrainSet = tall(trainDs);
tallTestSet = tall(testDs);

filteredTrainSignals = gather(cellfun(@(x)filter(hFilt,x),tallTrainSet(:,1),'UniformOutput',false

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: 0% complete
Evaluation 0% complete
```



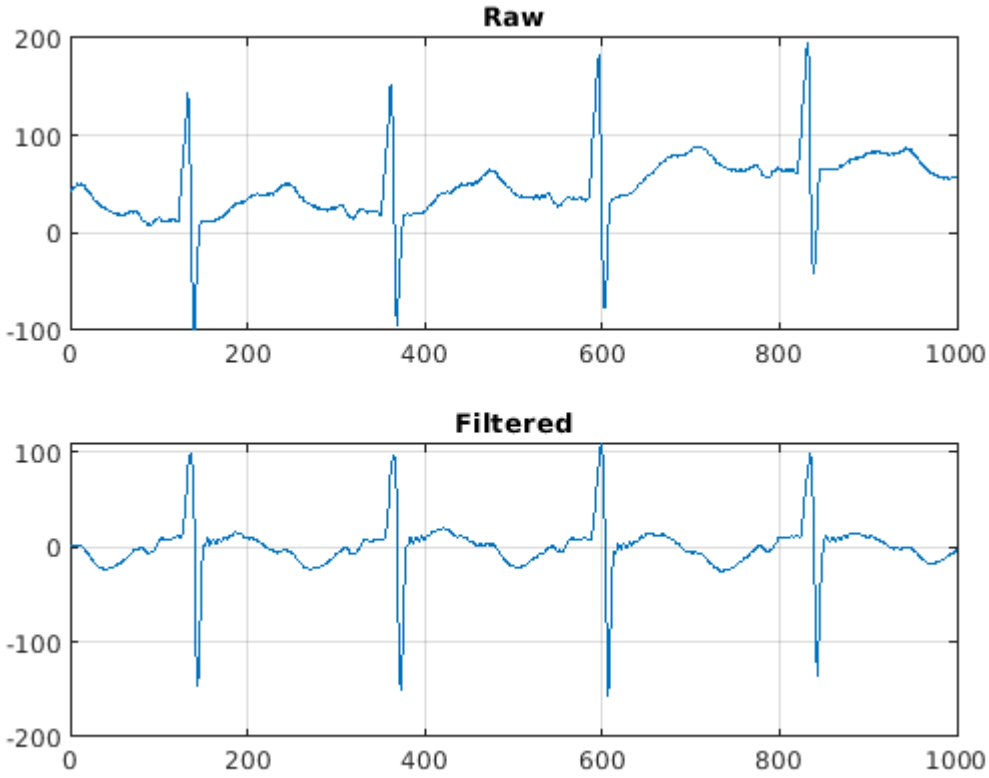

```
- Pass 1 of 1: Completed in 13 sec
Evaluation completed in 14 sec

trainLabels = gather(tallTrainSet(:,2));

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 3.6 sec
Evaluation completed in 4 sec

filteredTestSignals = gather(cellfun(@(x)filter(hFilt,x),tallTestSet(:,1),'UniformOutput',false)

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 2.4 sec
Evaluation completed in 2.5 sec

testLabels = gather(tallTestSet(:,2));

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1.9 sec
Evaluation completed in 2 sec
```

Plot the raw and filtered signals for a typical case.

```
trainData = gather(tallTrainSet);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 4 sec
Evaluation completed in 4.2 sec
```

```
figure
subplot(2,1,1)
plot(trainData{95,1}(2001:3000))
title('Raw')
grid
subplot(2,1,2)
plot(filteredTrainSignals{95}(2001:3000))
title('Filtered')
grid
```

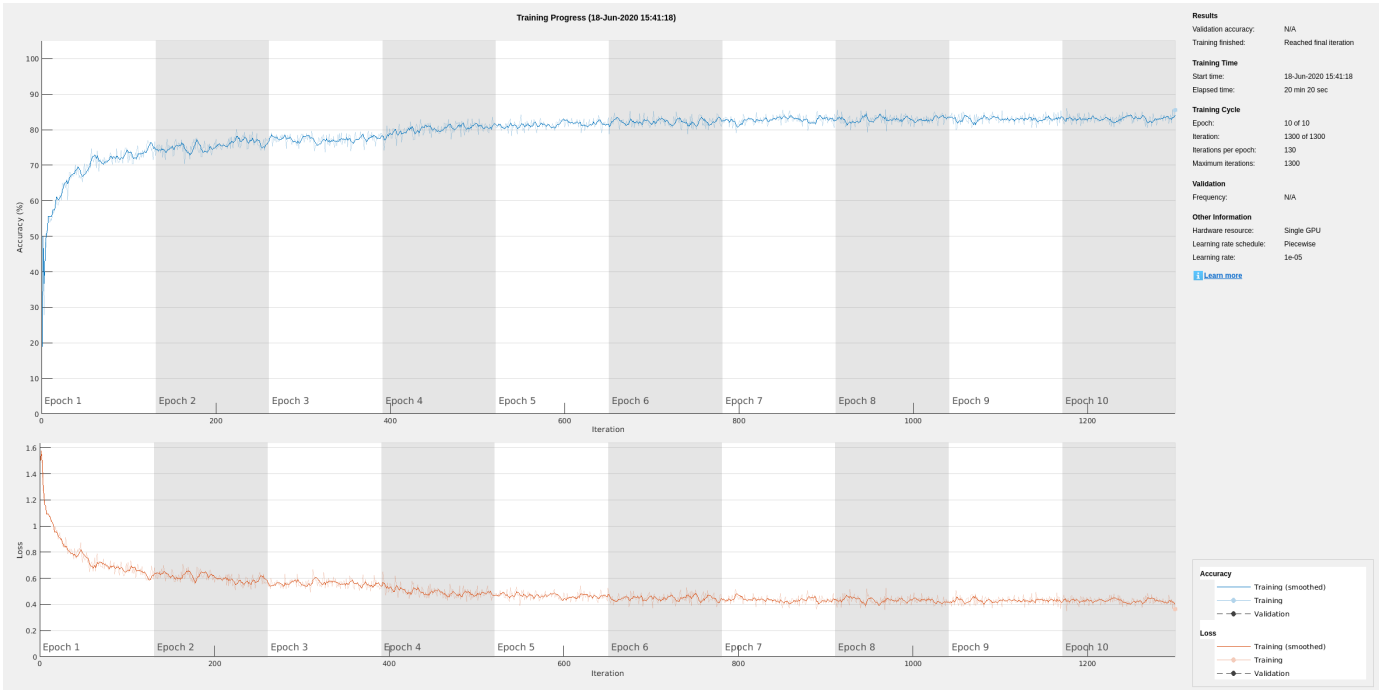


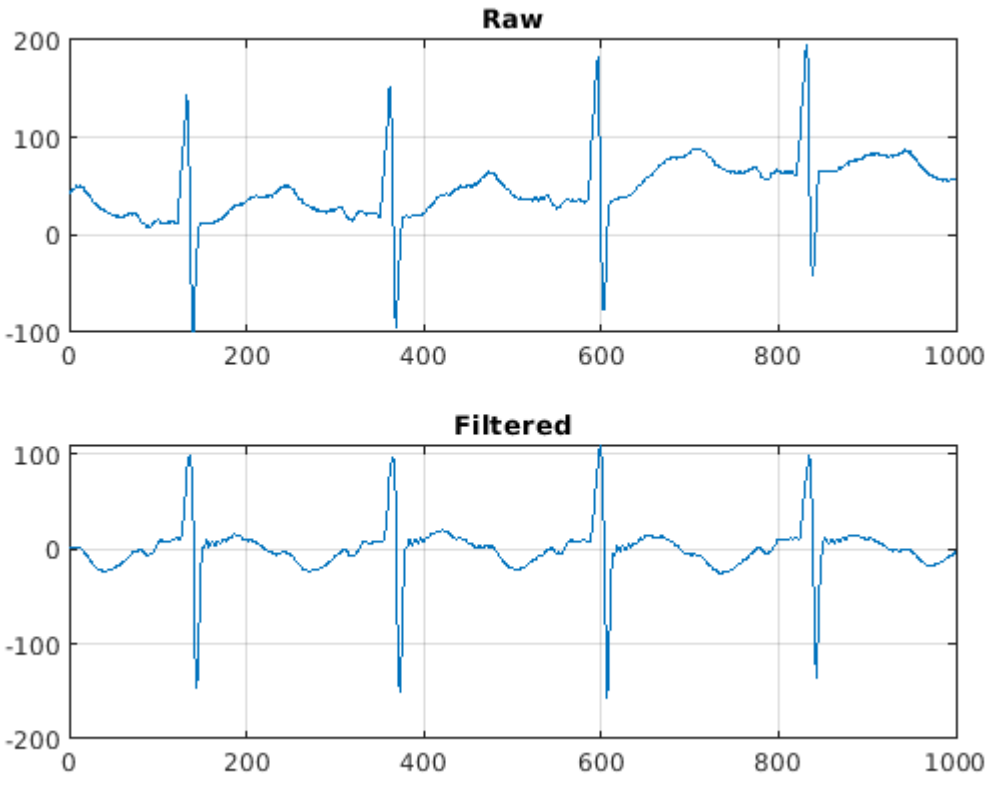

Even though the baseline of the filtered signals may confuse a physician that is used to traditional ECG measurements on medical devices, the network will actually benefit from the wandering removal.

### Train Network with Filtered ECG Signals

Train the LSTM network on the filtered ECG signals using the same network architecture as before.

```
if actionFlag == "Train networks"
    filteredNet = trainNetwork(filteredTrainSignals,trainLabels,layers,options);
end
```




Preprocessing the signals improves the training accuracy to better than 80%.

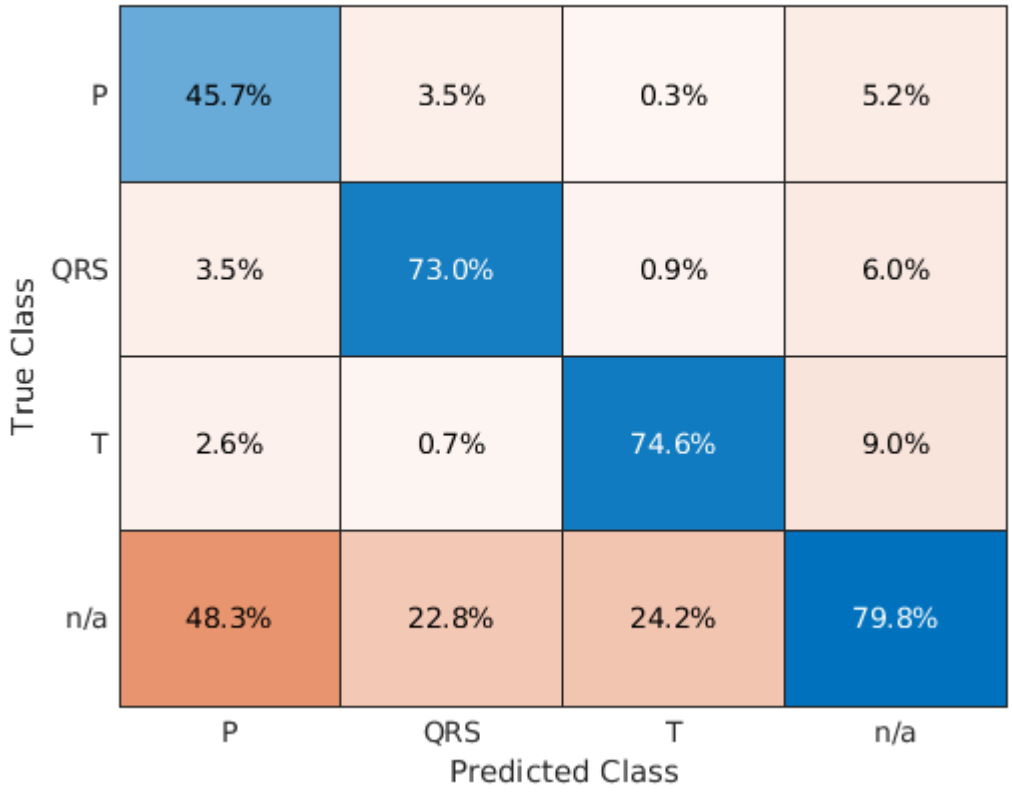
### Classify Filtered ECG Signals

Classify the preprocessed test data with the updated LSTM network.

```
predFilteredTest = classify(filteredNet,filteredTestSignals,'MiniBatchSize',50);
```

Visualize the classification performance as a confusion matrix.

```
figure
confusionchart([predFilteredTest{:}],[testLabels{:}],'Normalization','column-normalized');
```



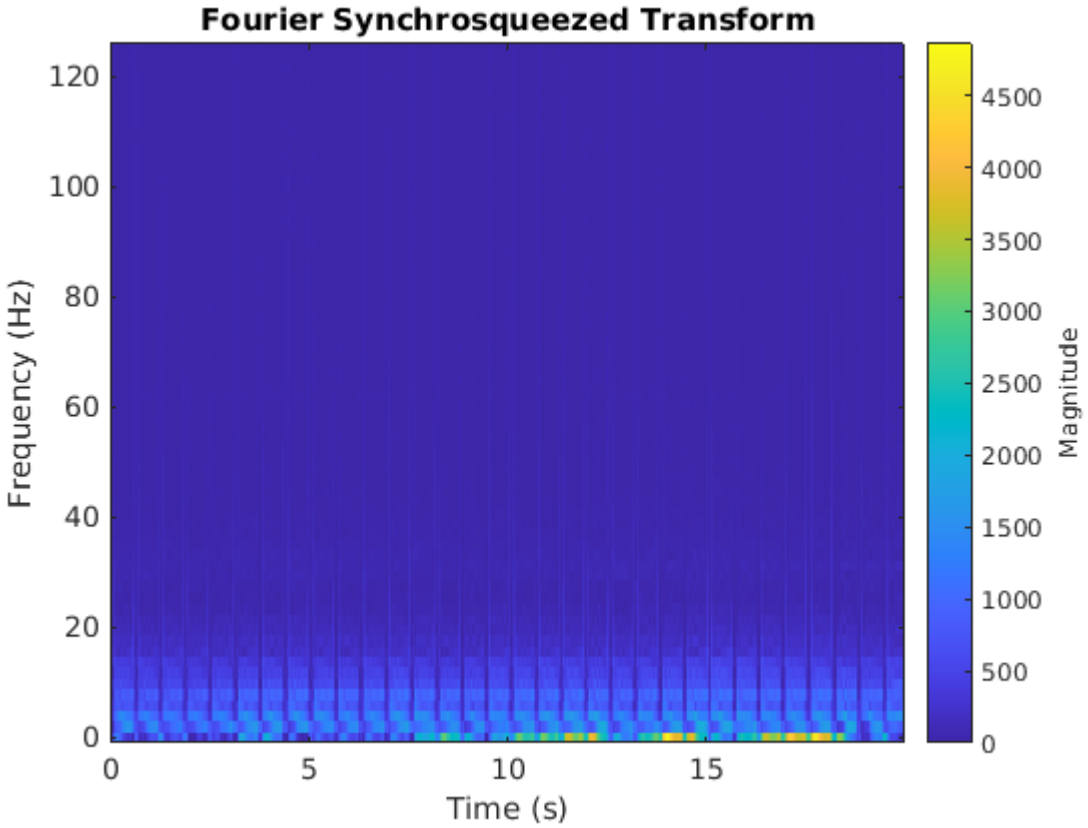

Simple preprocessing improves T-wave classification by about 15%, and QRS-complex and P-wave classification by about 10%.

### Time-Frequency Representation of ECG Signals

A common approach for successful classification of time-series data is to extract time-frequency features and feed them to the network instead of the original data. The network then learns patterns across time and frequency simultaneously [7 on page 24-0 ].

The Fourier synchrosqueezed transform (FSST) computes a frequency spectrum for each signal sample so it is ideal for the segmentation problem at hand where we need to maintain the same time resolution as the original signals. Use the `fsst` function to inspect the transform of one of the training signals. Specify a Kaiser window of length 128 to provide adequate frequency resolution.

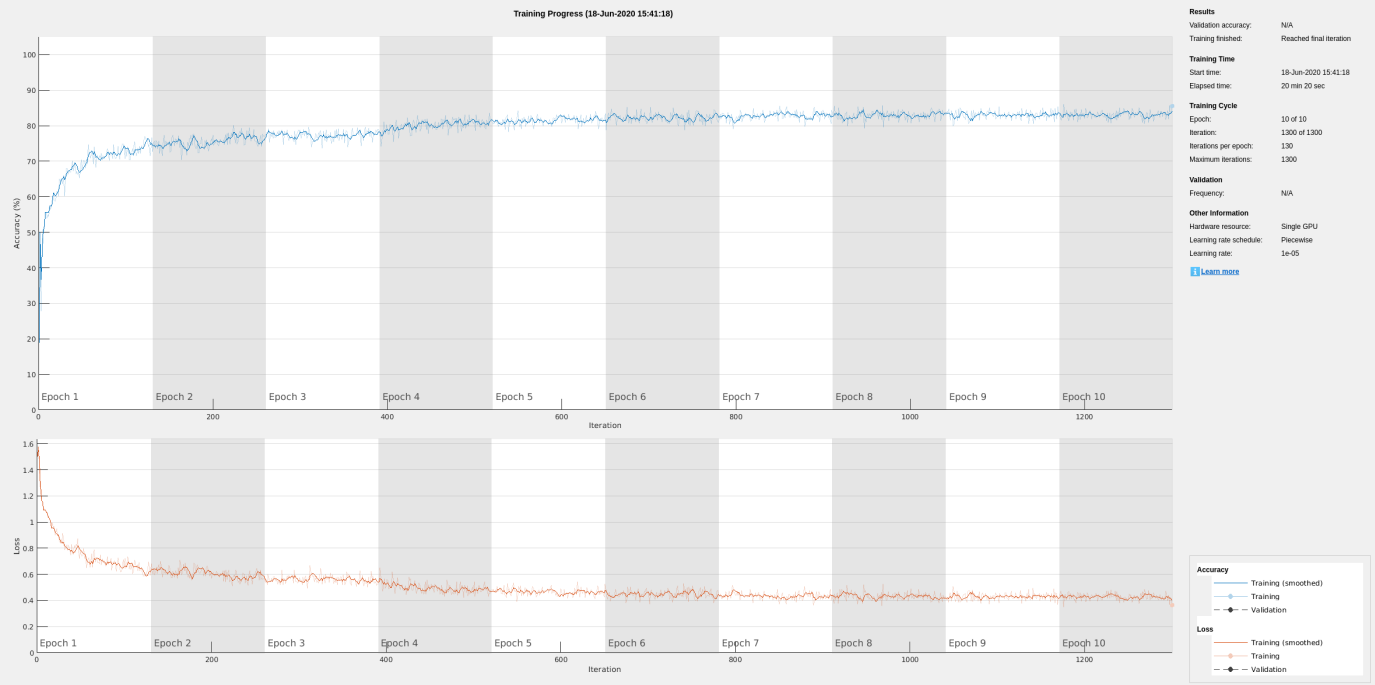
```
data =  preview(trainDs);
figure
fsst(data{1,1},250,kaiser(128),'yaxis')
```




Calculate the FSST of each signal in the training dataset over the frequency range of interest, [0.5, 40] Hz. Treat the real and imaginary parts of the FSST as separate features and feed both components into the network. Furthermore, standardize the training features by subtracting the mean and dividing by the standard deviation. Use a transformed datastore, the `extractFSSTFeatures` helper function, and the `tall` function to process the data in parallel.

```
fsstTrainDs = transform(trainDs,@(x)extractFSSTFeatures(x,250));
fsstTallTrainSet = tall(fsstTrainDs);
fsstTrainData = gather(fsstTallTrainSet);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: 0% complete
Evaluation 0% complete
```




```
- Pass 1 of 1: Completed in 2 min 35 sec
Evaluation completed in 2 min 35 sec
```

Repeat this procedure for the testing data.

```
fsstTTestDs = transform(testDs,@(x)extractFSSTFeatures(x,250));
fsstTallTestSet = tall(fsstTTestDs);
fsstTestData = gather(fsstTallTestSet);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1 min 4 sec
Evaluation completed in 1 min 4 sec
```

### Adjust Network Architecture

Modify the LSTM architecture so that the network accepts a frequency spectrum for each sample instead of a single value. Inspect the size of the FSST to see the number of frequencies.

```
size(fsstTrainData{1,1})
```

```
ans = 1×2

          40        5000
```

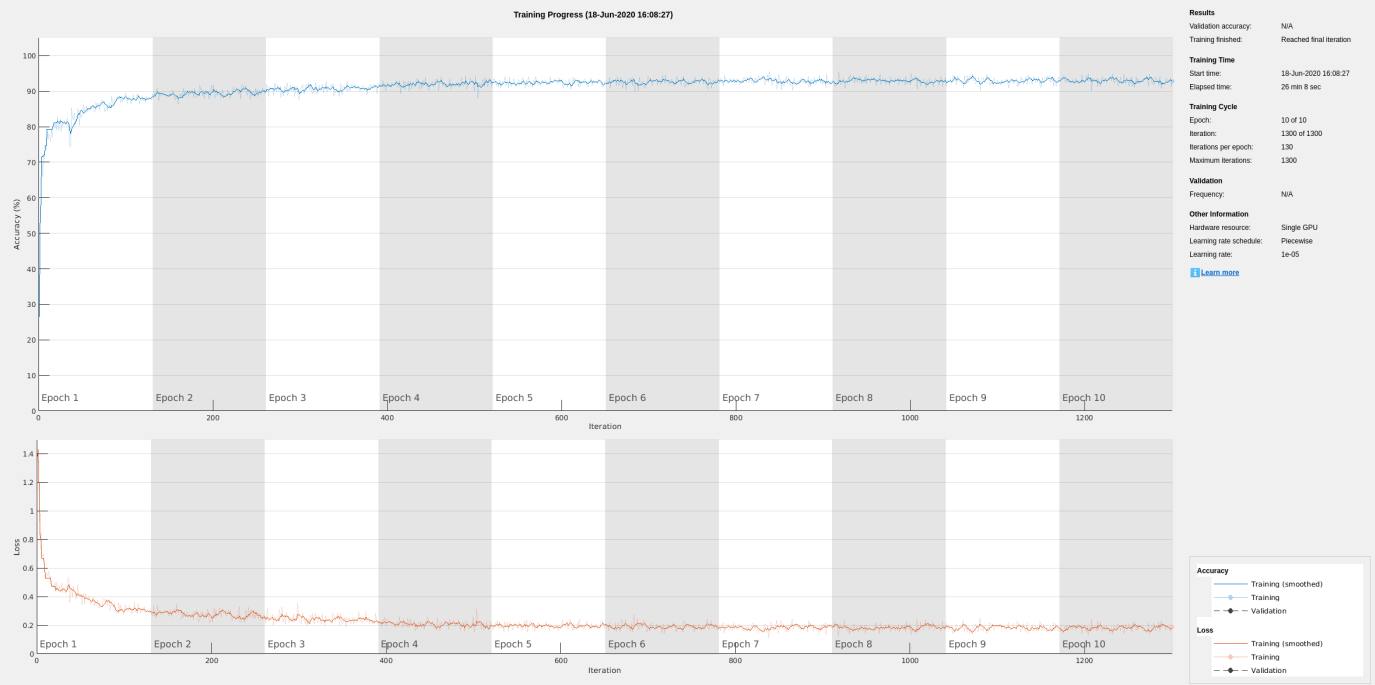
Specify a `sequenceInputLayer` of 40 input features. Keep the rest of the network parameters unchanged.

```
layers = [ ...
    sequenceInputLayer(40)
    lstmLayer(200,'OutputMode','sequence')
    fullyConnectedLayer(4)
    softmaxLayer
    classificationLayer];
```

### Train Network with FSST of ECG Signals

Train the updated LSTM network with the transformed dataset.

```
if actionFlag == "Train networks"
    fsstNet = trainNetwork(fsstTrainData(:,1),fsstTrainData(:,2),layers,options);
end
```




Using time-frequency features improves the training accuracy, which now exceeds 90%.

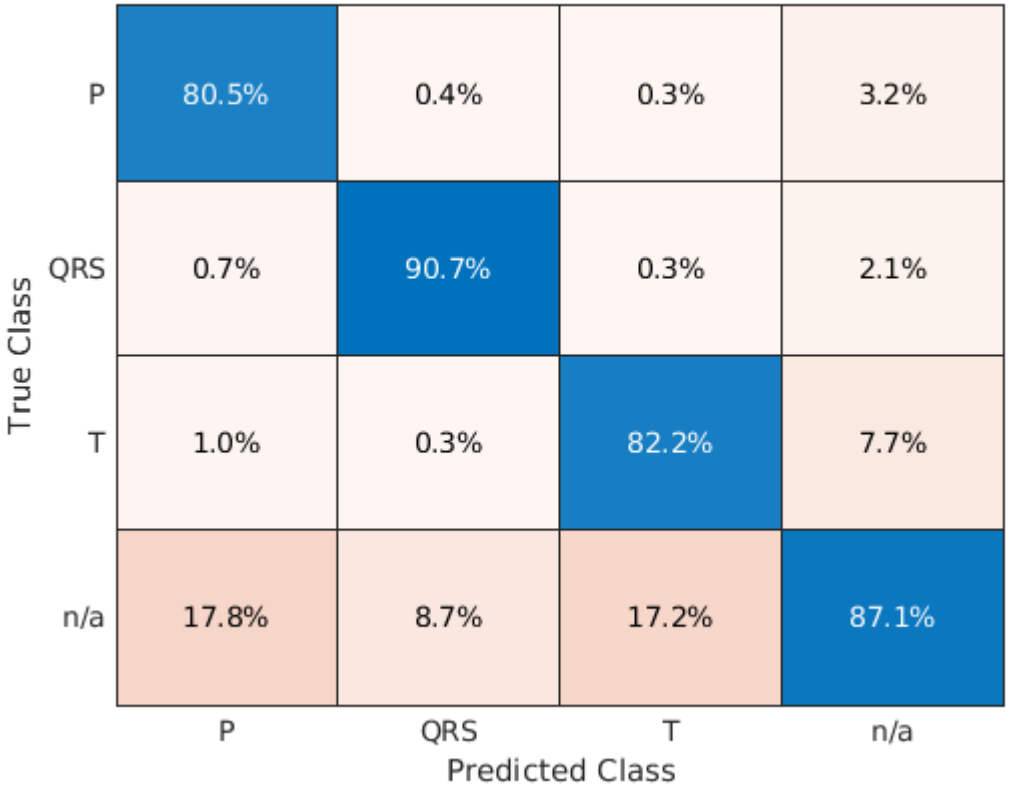
### Classify Test Data with FSST

Using the updated LSTM network and extracted FSST features, classify the testing data.

```
predFsstTest = classify(fsstNet,fsstTestData(:,1),'MiniBatchSize',50);
```

Visualize the classification performance as a confusion matrix.

```
confusionchart([predFsstTest{:}],[fsstTestData{:,2}],'Normalization','column-normalized');
```

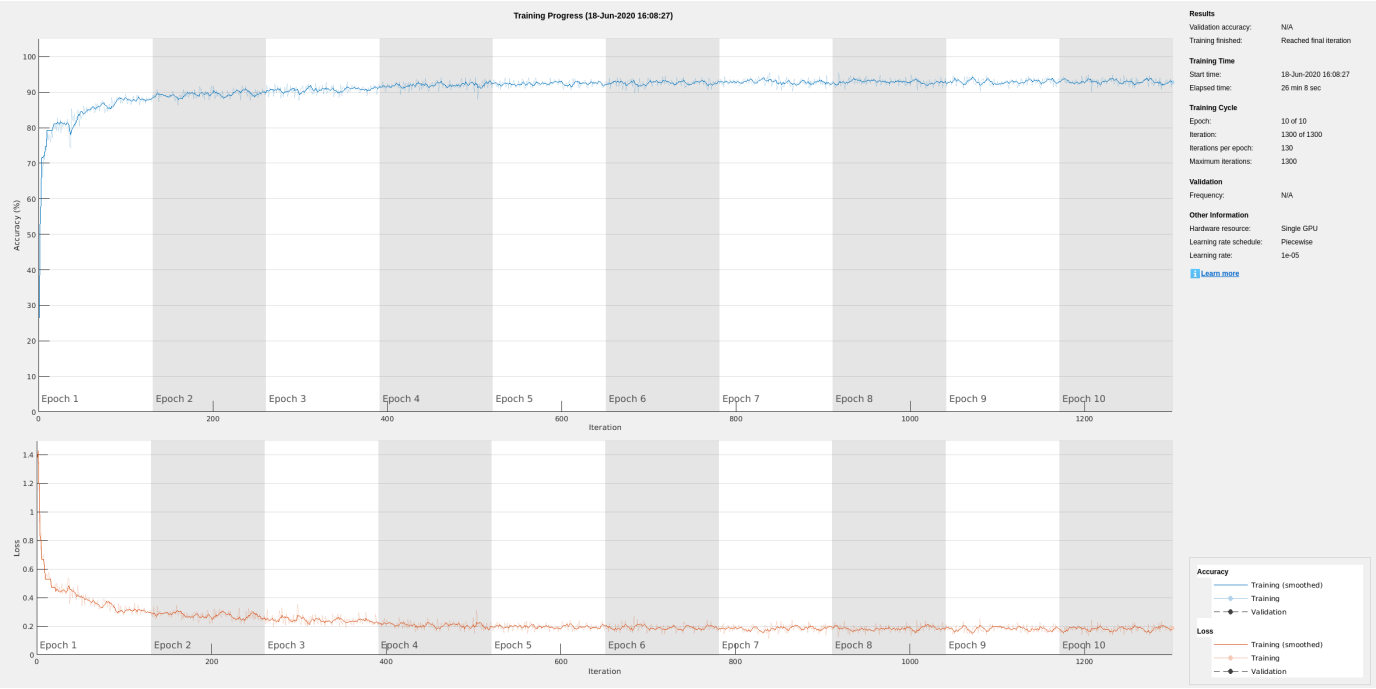



Using a time-frequency representation improves T-wave classification by about 25%, P-wave classification by about 40%, and QRS-complex classification by 30%, when compared to the raw data results.

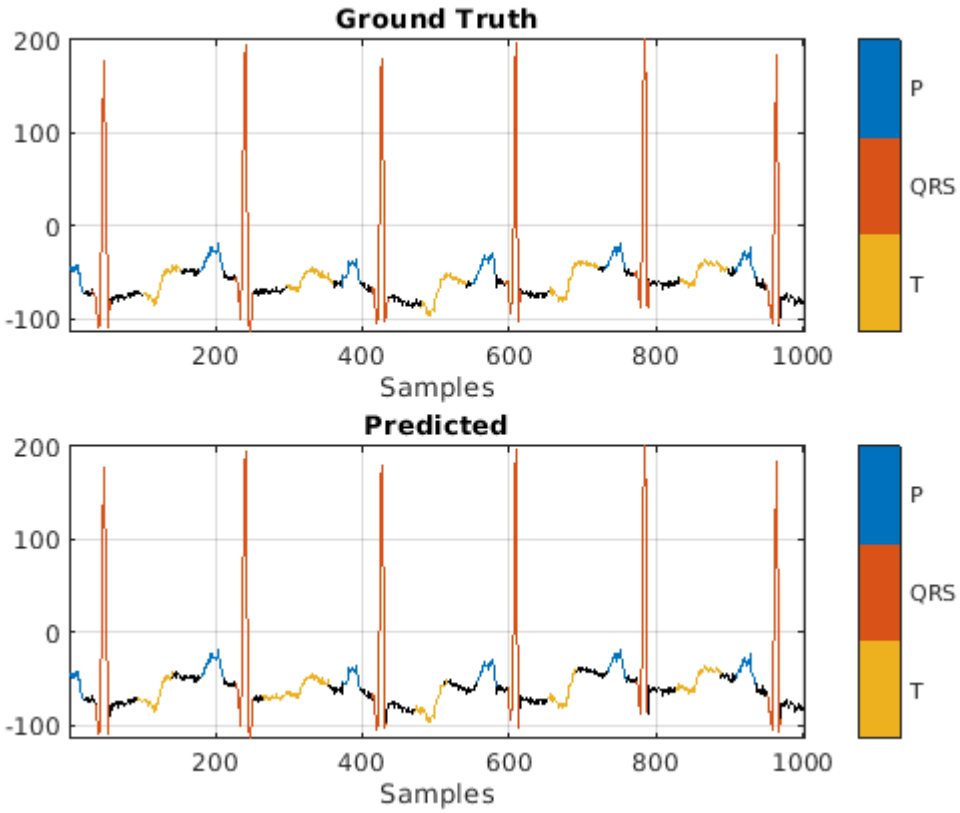
Use a `signalMask` object to compare the network prediction to the ground truth labels for a single ECG signal. Ignore the `"n/a"` labels when plotting the regions of interest.

```
testData = gather(tall(testDs));
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: 0% complete
Evaluation 0% complete
```




```
- Pass 1 of 1: Completed in 37 sec
Evaluation completed in 37 sec

Mtest = signalMask(testData{1,2}(3000:4000));
Mtest.SpecifySelectedCategories = true;
Mtest.SelectedCategories = find(Mtest.Categories ~= "n/a");

figure
subplot(2,1,1)
plotsigroi(Mtest,testData{1,1}(3000:4000))
title('Ground Truth')

Mpred = signalMask(predFsstTest{1}(3000:4000));
Mpred.SpecifySelectedCategories = true;
Mpred.SelectedCategories = find(Mpred.Categories ~= "n/a");

subplot(2,1,2)
plotsigroi(Mpred,testData{1,1}(3000:4000))
title('Predicted')
```




### Conclusion

This example showed how signal preprocessing and time-frequency analysis can improve LSTM waveform segmentation performance. Bandpass filtering and Fourier-based synchrosqueezing result in an average improvement across all output classes from 55% to around 85%.

### References

[1] McSharry, Patrick E., et al. "A dynamical model for generating synthetic electrocardiogram signals." *IEEE® Transactions on Biomedical Engineering.* Vol. 50, No. 3, 2003, pp. 289–294.

[2] Laguna, Pablo, Raimon Jané, and Pere Caminal. "Automatic detection of wave boundaries in multilead ECG signals: Validation with the CSE database." *Computers and Biomedical Research*. Vol. 27, No. 1, 1994, pp. 45–60.

[3] Goldberger, Ary L., Luis A. N. Amaral, Leon Glass, Jeffery M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H. Eugene Stanley. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." *Circulation*. Vol. 101, No. 23, 2000, pp. e215–e220. [Circulation Electronic Pages; http://circ.ahajournals.org/content/101/23/e215.full].

[4] Laguna, Pablo, Roger G. Mark, Ary L. Goldberger, and George B. Moody. "A Database for Evaluation of Algorithms for Measurement of QT and Other Waveform Intervals in the ECG." *Computers in Cardiology.* Vol.24, 1997, pp. 673–676.

[5] Sörnmo, Leif, and Pablo Laguna. "Electrocardiogram (ECG) signal processing." *Wiley Encyclopedia of Biomedical Engineering,* 2006.

[6] Kohler, B-U., Carsten Hennig, and Reinhold Orglmeister. "The principles of software QRS detection." *IEEE Engineering in Medicine and Biology Magazine*. Vol. 21, No. 1, 2002, pp. 42–57.

[7] Salamon, Justin, and Juan Pablo Bello. "Deep convolutional neural networks and data augmentation for environmental sound classification." *IEEE Signal Processing Letters.* Vol. 24, No. 3, 2017, pp. 279–283.

## See Also

**Functions**

`confusionchart` | `fsst` | `labeledSignalSet` | `lstmLayer` | `trainNetwork` | `trainingOptions`

## More About

- “Long Short-Term Memory Networks” (Deep Learning Toolbox)
- “Sequence-to-Sequence Regression Using Deep Learning” (Deep Learning Toolbox)
- “Sequence-to-Sequence Classification Using Deep Learning” (Deep Learning Toolbox)

# Deploy Signal Segmentation Deep Network on Raspberry Pi

This example details the workflow for waveform segmentation of an electrocardiogram (ECG) signal using short-time Fourier transform and a bidirectional long short-term memory (BiLSTM) network. The example also provides information on how to generate and deploy the code and the trained BiLSTM network for segmentation on a Raspberry Pi™ target (ARM®-based device).

The pretrained network in the example is similar to the "Waveform Segmentation Using Deep Learning" on page 24-330 example.

This example details:

- Processor-in-the-loop (PIL) based workflow to verify generated code deployed and running on a Raspberry Pi from MATLAB™
- Generation of a standalone executable

The PIL verification process is a crucial part of the design cycle to check that the behavior of the generated code matches the design before deploying a standalone executable.

### ECG Dataset

This example uses ECG signals from the publicly available QT Database [1 on page 24-0 ] [2 on page 24-0 ]. The data consists of roughly 15 minutes of labeled ECG recordings, with a sample rate of 250 Hz, measured from a total of 105 patients.

The ECG signal can be divided into the following beat morphologies [3 on page 24-0 ]:

- P wave — A small deflection before the QRS complex representing atrial depolarization
- QRS complex — Largest amplitude portion of the heartbeat
- T wave — A small deflection after the QRS complex representing ventricular repolarization

The segmentation of these regions of ECG waveforms can provide the basis for measurements that assess the overall health of the human heart and the presence of abnormalities.

### Prerequistes

- ARM processor that supports the NEON extension
- ARM Compute Library (on the target ARM hardware)
- MATLAB® Coder™
- Embedded Coder™
- Deep Learning Toolbox™
- Deep Learning Support for MATLAB Coder
- MATLAB Support Package for Raspberry Pi™

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

### Functionality of Generated Code

The core function in the generated executable:

- Uses 15,000 samples of single-precision ECG data as input.
- Computes the short-time Fourier transform of the signal.
- Standardizes and normalizes the output.
- Labels regions of the signal using the pretrained BiLSTM network.
- Generates an output file with the labels.

### `waveformSegmentation` Function

An *entry-point* function, also known as the *top-level* or *primary* function, is a function you define for code generation. You must define an entry-point function that calls code-generation-enabled functions and generates C/C++ code from the entry-point function. All functions within the entry-point function must support code generation.

In this example, w`aveformSegmentation` is the entry-point function. It takes an ECG signal as an input and passes it to the trained BiLSTM network for prediction. The `performPreprocessing` function preprocesses the raw signal and applies the short-time Fourier transform. The `genClassifiedResults` function passes the preprocessed signal to the network for prediction and dislays the classification results.

```
type waveformSegmentation
```

```
function out = waveformSegmentation(in)
%#codegen
persistent net;

if isempty(net)
    net = coder.loadDeepLearningNetwork('trained-network-STFTBILSTM.mat', 'net');
end


preprocessedSignal = performPreprocessing(in);
out = cell(3,1);


for indx = 1:3
  out{indx,1} =  genClassifedResults(net.predict(preprocessedSignal{1,indx}));
end



end
```

### Create a Connection to the Raspberry Pi

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

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

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

The example shows the PIL-based wokflow for verification of code and design and then creates and deploys a standalone executable. Optionally, if you want to directly deploy a standalone executable, you can skip PIL execution and go to creating a standalone execution.

### Generate PIL MEX Function

The first step shows a PIL-based workflow to generate a MEX function for the `waveformSegmentation` function.

### Set Up Code Generation Configuration Object for a Static Library

Create a code configuration object for a static library and set the verification mode to `'PIL'`. Set the target language to `'C++'`.

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

### Set Up Configuration Object for Deep Learning Code Generation

Create a `coder.ARMNEONConfig` object. Specify the version of the ARM Compute library as the one on the Raspberry Pi. Specify the architecture of the Raspberry Pi. (This example requires ARM Compute Library v19.05).

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

Set the `DeepLearningConfig` property of the code generation configuration object to the deep learning configuration object. Set the configuration object with MATLAB Source Comments visible in the codegen.

```
cfg.DeepLearningConfig = dlcfg;
cfg.MATLABSourceComments = 1;
```

### Configure Code Generation Hardware Parameters for Raspberry Pi

Create a `coder.Hardware` object for the 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.

```
cfg.Hardware.BuildDir = '~/waveformSegmentation';
```

### Generate Source C++ Code Using codegen Function

Use the `codegen` function to generate the C++ code. When `codegen` is used with MATLAB Support Package for Raspberry Pi Hardware, the generated code is downloaded to the board and compiled there. A PIL MEX function is generated to communicate between MATLAB and the generated code running on the Raspberry Pi.

Make sure to set the environment variables `ARM_COMPUTELIB` and `LD_LIBRARY_PATH` on the Raspberry Pi. See “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder) (MATLAB Coder).

```
codegen -config cfg waveformSegmentation -args {coder.typeof(single(ones(1,15000)),[1,15000],[0,0

### Target device has no native communication support. Checking connectivity configuration regist
 Deploying code. This may take a few minutes.
### Target device has no native communication support. Checking connectivity configuration regist
### Connectivity configuration for function 'waveformSegmentation': 'Raspberry Pi'
Location of the generated elf : /home/pi/waveformSegmentation/MATLAB_ws/R2020b/C/Users/eshashah/C
Code generation successful: View report
```

**Run the Executable Program on Raspberry Pi**

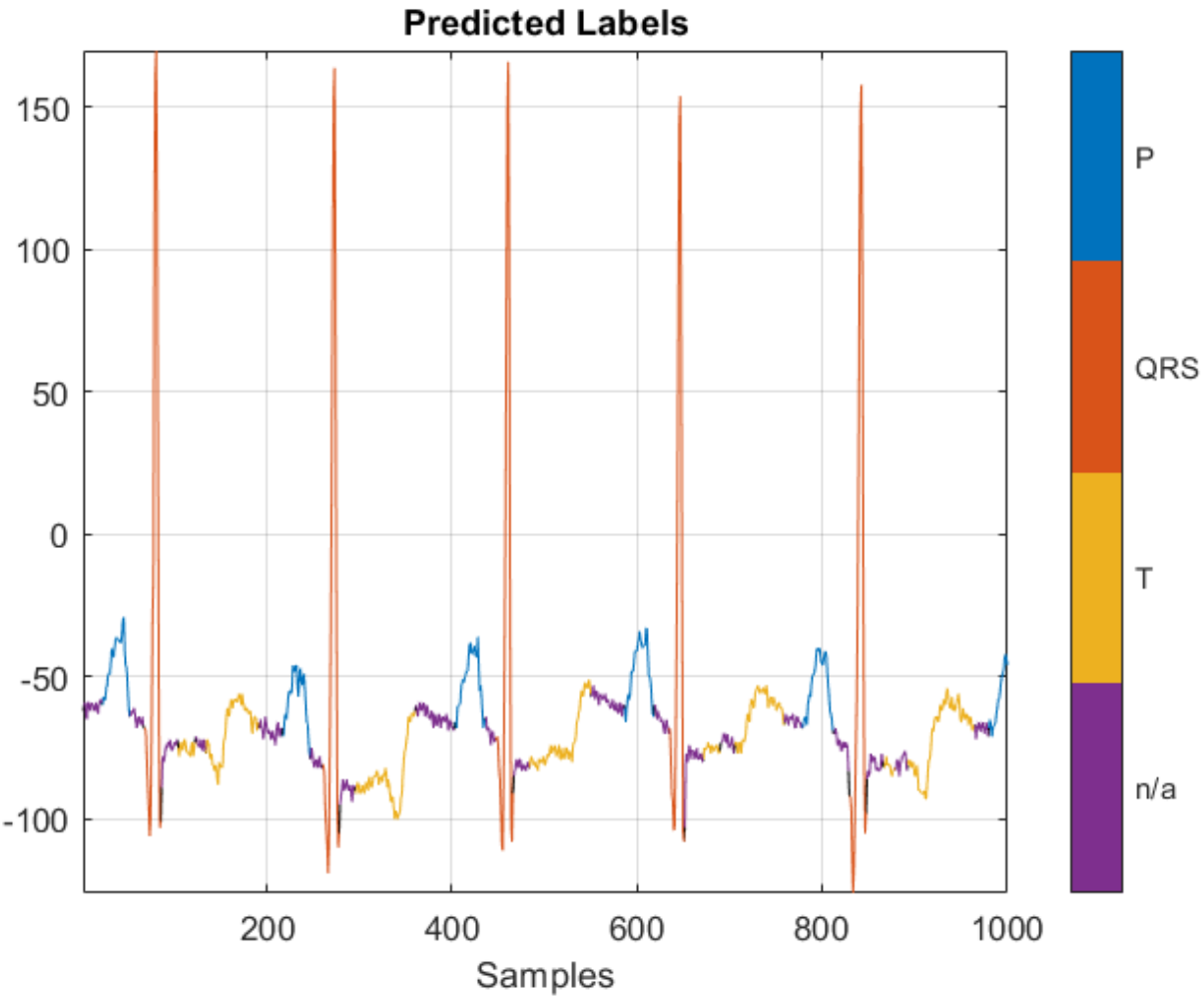
Load the MAT-file `ecgsignal_test`. The file stores a sample ECG signal on which you can test the generated code.

Run the generated `waveformSegmentation_pil` MEX function on the test signal.

```
load ecgsignal_test.mat;
out = waveformSegmentation_pil(test);

### Starting application: 'codegen\lib\waveformSegmentation\pil\waveformSegmentation.elf'
    To terminate execution: clear waveformSegmentation_pil
### Launching application waveformSegmentation.elf...
```

Display the signals with predicted labels.

```
labels = categorical(out{1}(1,2000:3000));
msk = signalMask(labels);
plotsigroi(msk,test(1,2000:3000))
title('Predicted Labels')
```




After the verifying the output of the PIL MEX function, you can create a standalone executable for the `waveformSegmentation` function.

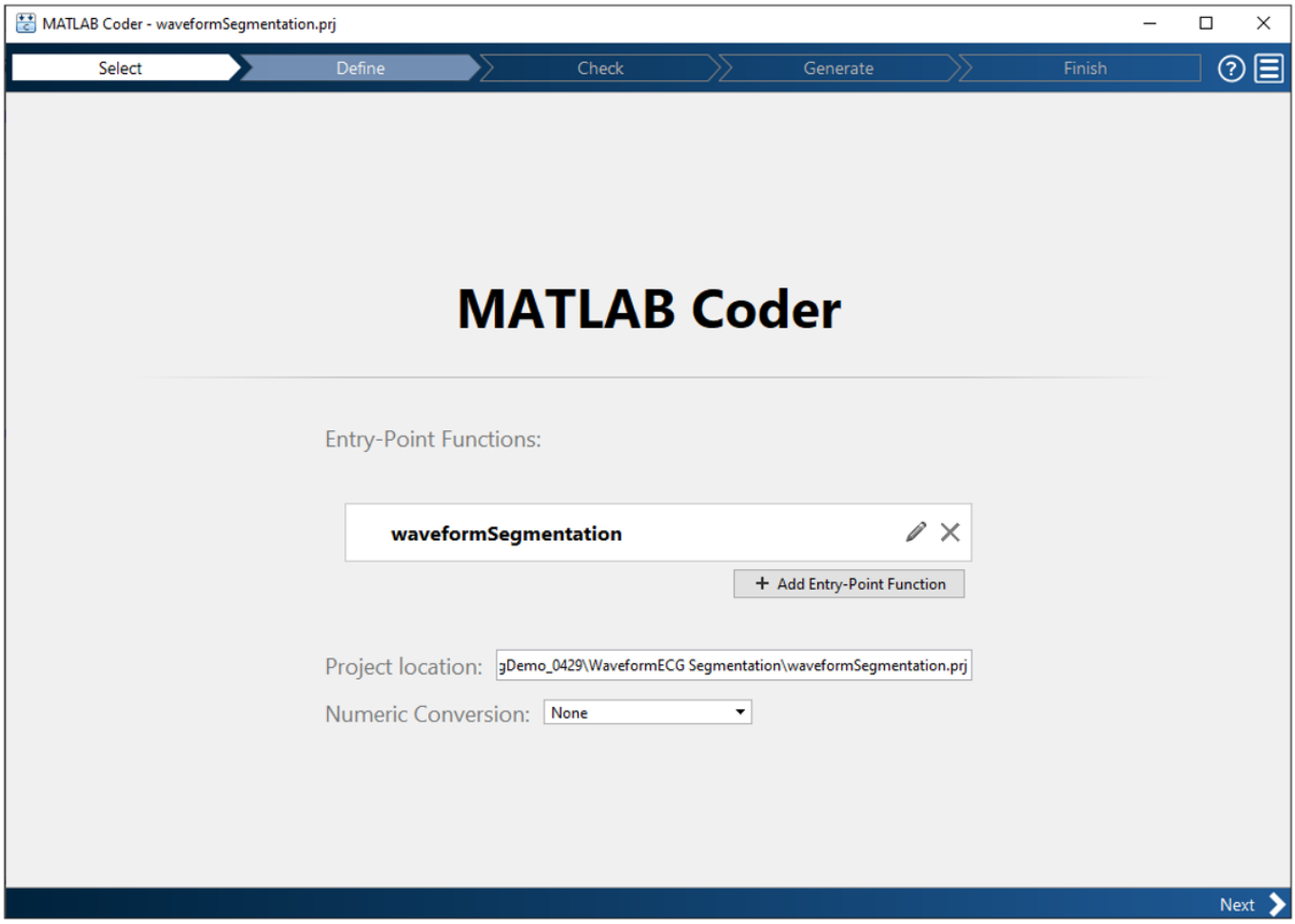
The next part shows the code generation workflow to create a generate and deploy a standalone executable in the code for prediction on a Raspberry Pi using the MATLAB Coder App.

### Create a Standalone Executable Using the MATLAB Coder App

The **MATLAB Coder** app generates C or C++ code from MATLAB® code. The workflow-based user interface steps you through the code generation process. The following steps describe a brief workflow using the MATLAB Coder app. For more details, see MATLAB Coder (MATLAB Coder) and “Generate C Code by Using the MATLAB Coder App” (MATLAB Coder).

### Select the Entry-Point Function File

On the **Apps** tab, click the down arrow on the far right of the toolstrip to expand the apps gallery. Under **Code Generation**, click **MATLAB Coder**. The app opens the **Select Source Files** page. Enter or select the name of the entry-point function, `waveformSegmentation`.




Click **Next** to go to the **Define Input Types** page.

**Define Input Types**

1. Select **Let me enter input or global types directly** and set the value of the input `in` as single `(1x15000)`.

2. Click **Next** to go to the **Generate Code** step. Skip the **Check for Run-Time Issues** step because MEX generation is not supported for code generation with the ARM Compute Library.

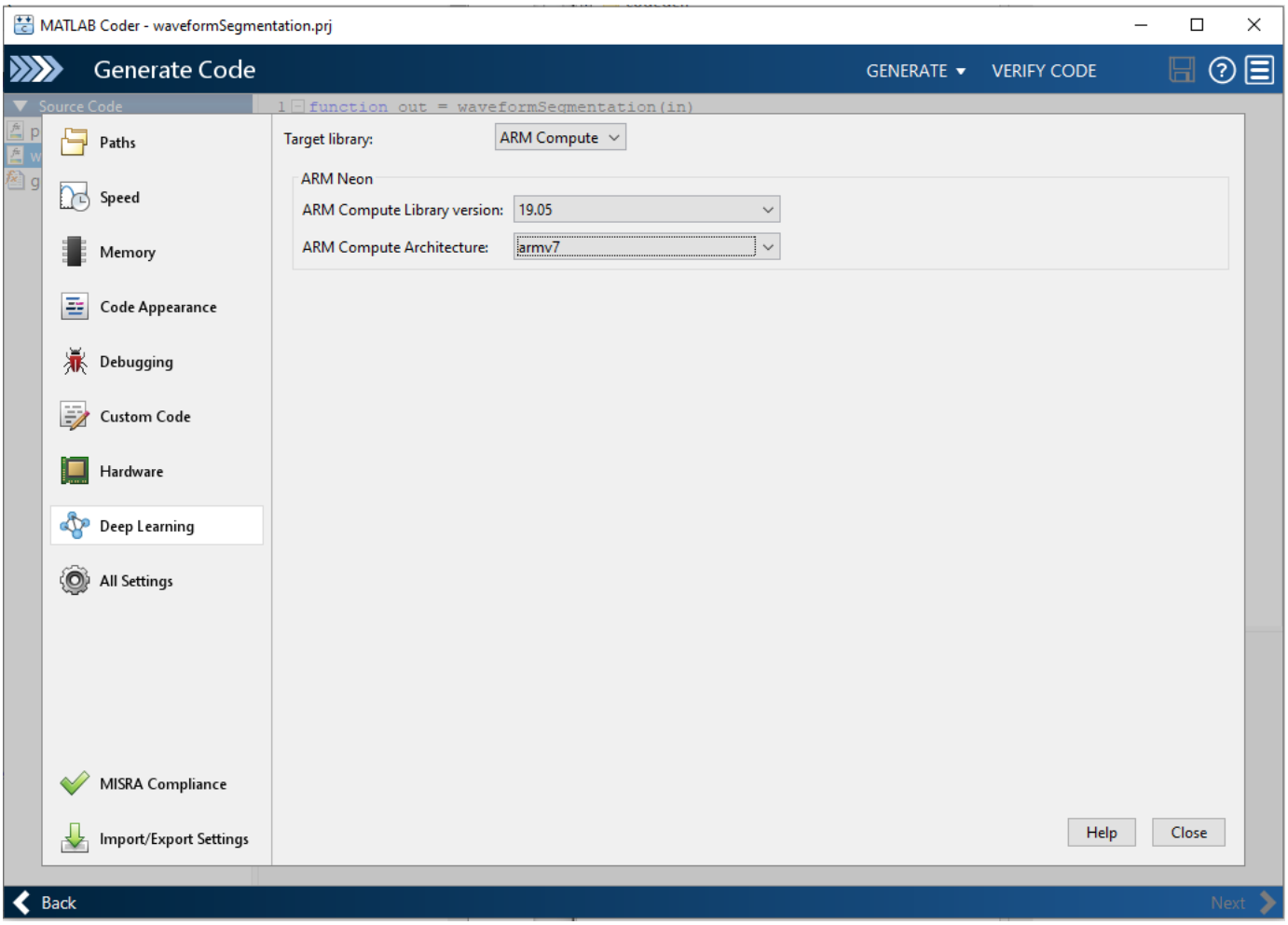
**Generate Code**

1. Set values in the generate code dialog box:

- Set **Build Type** to `Executable (.exe)`
- Set **Language** to `C++`
- Set **Hardware Board** as `Raspberry Pi`

2. Click the **More Settings** button:

- In the **Custom Code** pane, in additional source files, browse and select `ecgsegmentation_main.cpp`. For more information on writing a C/C++ main function, refer to “Structure of Generated Example C/C++ Main Function” (MATLAB Coder).
- In the **Hardware** pane, set the **username** and **password** for the Raspberry Pi board.
- In the **Deep Learning** pane, set **Target library** to `ARM Compute`. Specify **ARM Compute Library version** and **ARM Compute Architecture**.




3. Close the Settings window and generate code.

4. Click **Next** to go to the **Finish Workflow** page.

### Fetch Generated Executable Directory

Once the code generation is completed the following lines of code to test the generated code on the Raspberry Pi, copy the input ECG signal to the generated code directory. You can find this directory manually or by using the `raspi.utils.getRemoteBuildDirectory` API. This function lists the directories of the binary files that are generated by using the `codegen` function. Assuming that the binary is found in only one directory, enter:

```
applicationDirPaths =...
```

```
raspi.utils.getRemoteBuildDirectory('applicationName','waveformSegmentation')
;

targetDirPath = applicationDirPaths{1}.directory;
```

**Copy Input Files to the Raspberry Pi**

To copy files required to run the executable program, use `putFile`, which is available with the MATLAB Support Package for Raspberry Pi Hardware. The `input.csv` file contains a sample ECG signal that is used to test the deployed code.

```
r.putFile('input.csv', targetDirPath);

input = dlmread('input.csv');
```

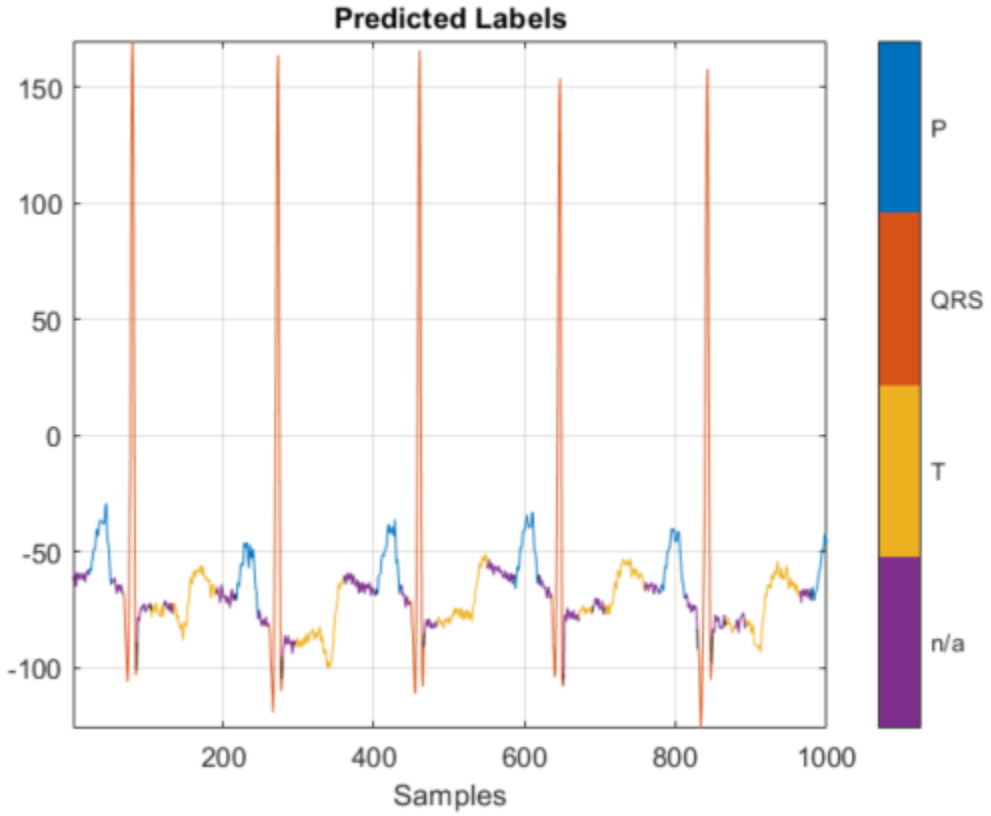
**Run Executable program on Raspberry Pi**

Run the executable program on the Raspberry Pi from MATLAB and get the output file to MATLAB. Input file name should be passed as the command line argument for the executable.

```
exeName = 'waveformSegmentation.elf'; % Executable name

command = ['cd ' targetDirPath ';./' exeName];

system(r,command)

outputPath = strcat(targetDirPath,'/*.txt');

getFile(r,outputPath)
```

Display the signals with predicted labels. The output is depicted in the figure.

```
load ecgsignal_test.mat;

labels = categorical(textread('out.txt','%s')');

msk = signalMask(labels(1,2000:3000));

plotsigroi(msk,test(1,2000:3000))

title('Predicted Labels')
```




### References

[1] McSharry, Patrick E., et al. "A dynamical model for generating synthetic electrocardiogram signals." *IEEE® Transactions on Biomedical Engineering.* Vol. 50, No. 3, 2003, pp. 289–294.

[2] Laguna, Pablo, Raimon Jané, and Pere Caminal. "Automatic detection of wave boundaries in multilead ECG signals: Validation with the CSE database." *Computers and Biomedical Research*. Vol. 27, No. 1, 1994, pp. 45–60.

[3] Goldberger, Ary L., Luis A. N. Amaral, Leon Glass, Jeffery M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H. Eugene Stanley. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." *Circulation.* Vol. 101, No. 23, 2000, pp. e215–e220. [Circulation Electronic Pages; http://circ.ahajournals.org/content/101/23/e215.full].

## See Also

**Apps**
**MATLAB Coder**

**Functions**
`codegen` | `fsst` | `signalMask`

## More About

- “Long Short-Term Memory Networks” (Deep Learning Toolbox)
- “Generate C Code by Using the MATLAB Coder App” (MATLAB Coder)
- “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder)
- “Structure of Generated Example C/C++ Main Function” (MATLAB Coder)

# Iterative Approach for Creating Labeled Signal Sets with Reduced Human Effort

This example presents an iterative deep learning-based workflow to label signals with reduced human labeling effort.

Labeling signal data is a tedious and expensive task that requires much human effort. Finding ways to reduce this effort can significantly speed up the development of deep learning solutions for signal processing problems.

Consider the task of labeling regions of interest in a signal data set. A first approach consists of labeling the data by hand. This approach requires much time and effort. An alternative approach, explored in this example, treats the labeling process iteratively. At each iteration, a subset of signals is selected from the unlabeled data set and is sent to a pretrained deep network for automated labeling. A human labeler examines the resulting labels and corrects wrong labels. The validated labeled signals are added to a training data set to retrain the deep network with the extended training data.

At each iteration, the human labeler still has to visit and examine all the signals labeled by the network. However, the task changes from labeling signals from scratch to correcting inaccurate labels generated by a reliable network. This latter task requires considerably less human labeling effort. At each new iteration, the network is trained with more and more data, causing the prediction and labeling performance of the network to improve. Hence, at each iteration, less and less human intervention is required to correct labels.

### Data

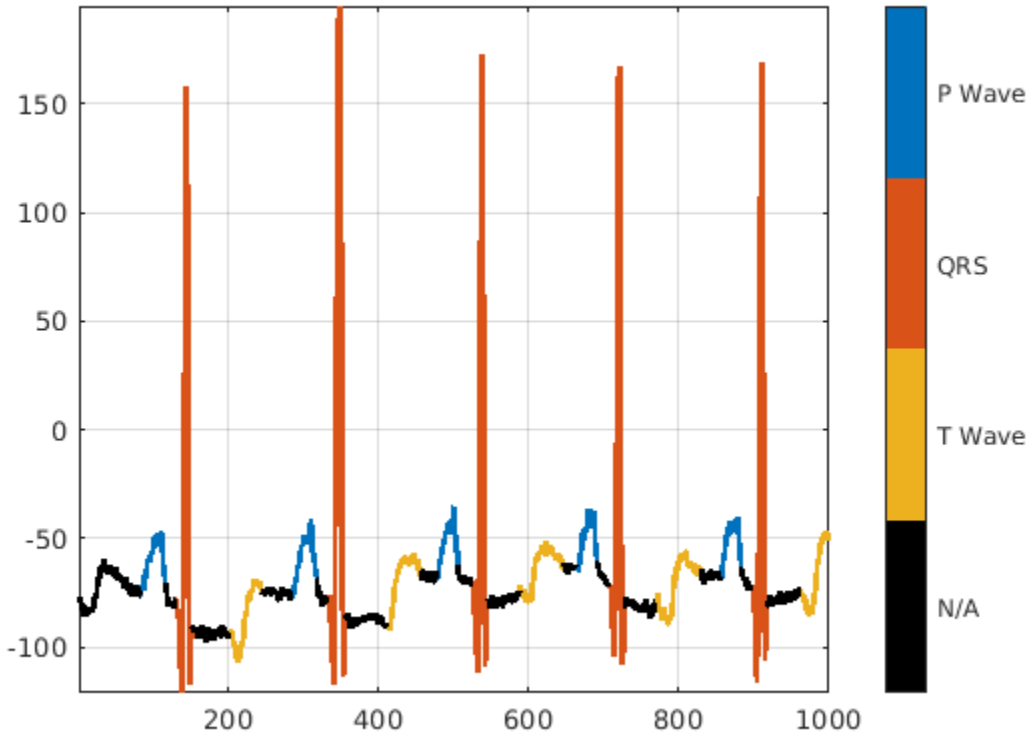
This example considers the labeling of ECG signal regions using data publicly available in the QT Database [1 on page 24-0 ] [2 on page 24-0 ]. The data consists of roughly 15 minutes of ECG recordings from a total of 105 patients. To obtain each recording, the examiners placed two electrodes on different locations on a patient's chest, resulting in a two-channel signal. The database provides signal region labels generated by an automated expert system [3 on page 24-0 ]. The labels correspond to the locations of P wave, T wave, and QRS complex regions in the ECG measurements. This example follows the procedure presented in “Waveform Segmentation Using Deep Learning” on page 24-330 to train a bidirectional long short-term memory (BiLSTM) network that can classify ECG signal samples as belonging to one of the three regions of interest.

Download the data from the GitHub Repository. This example assumes that you have put the file in your temporary directory, whose location is specified by MATLAB®'s `tempdir` command. Unzip the datafile and load the data.

```
unzip(fullfile(tempdir,'QT_Database-master.zip'),tempdir)
load(fullfile(tempdir,'QT_Database-master','QTData.mat'))
```

Use the `displayWaveformLabels` helper function to plot the first 1000 samples of the ECG signal from the first patient, with the regions labeled by the expert system overlaid in different colors. The dataset contains an extra label, 'N/A', that is used to identify signal samples that do not belong to any of the three regions of interest.

```
patientID = 1;
signalVals = getSignal(QTData,patientID);
labelVals = getLabelValues(QTData,patientID,'WaveformLabels_Chan1');

displayWaveformLabels(signalVals(1,1:1000),labelVals.Value(1:1000))
```




Use the `resizeSignals` helper function to break the ECG signals and label vectors into frames, each of them 5000 samples long. The function then shuffles the frames and divides them into a training data set containing 70% of the data and a test data set containing 30% of the data.

```
rng default
[testFrames,testLabels,ecgFrames,ecgLabels,testInfo,ecgInfo] = resizeSignals(QTData,0.3);
```

`ecgFrames` contains 6543 frames and `testFrames` contains 2768 frames. The variable `ecgInfo` stores the patient ID, the channel ID, and the sample index of each frame of the training data set. `testInfo` stores the same information for the test data set.

Transform the ECG frames to the time-frequency domain. Use the `featureExtraction` helper function to compute the Fourier synchrosqueezed transform of the ECG signal frames over the frequency range of interest, [0.5, 40] Hz, and standardize the resulting transforms by subtracting the mean and dividing by the standard deviation.

```
Fs = QTData.SampleRate;
testFrames = featureExtraction(testFrames,Fs);
ecgFrames = featureExtraction(ecgFrames,Fs);
```

This example shows that it is possible to decrease the human effort involved in signal labeling by iteratively training a deep network. At each iteration:

1. The network labels a subset of unlabeled data frames using previously labeled frames.
2. A human labeler corrects any labeling errors by hand.

3 The corrected labeling is added to the previously labeled frames.

4 The expanded set of labeled signals is used to train the network for the next iteration.

To make quantitative comparisons, simulate two scenarios:

- For the baseline scenario, in which a human being labels the entire dataset from scratch, train the network using the full labeled `ecgFrames` set.
- For the second scenario, pretend that the `ecgFrames` data is unlabeled and label it using the iterative method.

### Prediction Performance Using Fully Labeled ECG Data Set

Build a BiLSTM network and train it with the full labeled `ecgFrames` set to get a prediction-performance upper bound. As mentioned above, this approach requires brute-force labeling of the whole data set and hence the largest human labeling effort. Train the network with the labeled `ecgFrames` set and calculate the prediction accuracy on the test data set.

### Network Architecture

Create a BiLSTM network using deep learning layers.

- Specify a `sequenceInputLayer` with a size as the number of features in the FSST of the signals, which is the total number of frequency-domain samples (40 in this example).
- Specify a `bilstmLayer` with 200 hidden nodes and set the `OutputMode` to `sequence` since each signal sample has a label.
- Specify a `fullyConnectedLayer` with an output size of 4 corresponding to the four categories, P wave, QRS complex, T wave, and N/A.
- Add a `softmaxLayer` and a `classificationLayer` to output the estimated labels.

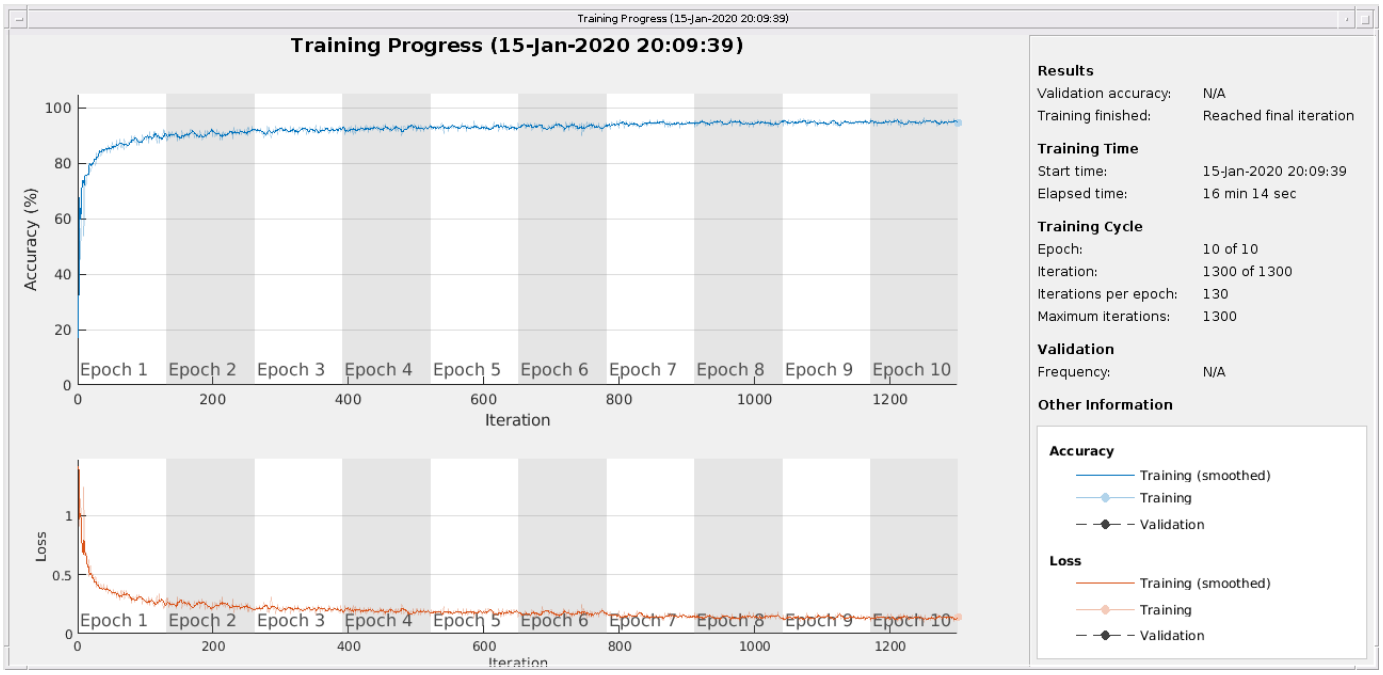
```
% Training with the full emulated unlabeled data set
layers = [ ...
    sequenceInputLayer(size(ecgFrames{1},1))
    bilstmLayer(200,'OutputMode','sequence')
    fullyConnectedLayer(4)
    softmaxLayer
    classificationLayer];
```

Use `traningOptions` to specify the optimization solver and the hyperparameters to train the network. This example uses the ADAM optimizer and a mini-batch size of 50. Train the network using either a CPU or GPU. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher. For information on other parameters, see `trainingOptions` (Deep Learning Toolbox). This example uses a GPU for training using the '`ExecutionEnvironment`' name-value pair.

```
options = trainingOptions('adam', ...
    'MaxEpochs',10, ...
    'MiniBatchSize',50, ...
    'ExecutionEnvironment','gpu', ...
    'InitialLearnRate',0.01, ...
    'LearnRateDropPeriod',6, ...
    'LearnRateSchedule','piecewise', ...
    'GradientThreshold',1, ...
    'Shuffle','every-epoch',...
    'Plots','training-progress',...
    'Verbose',0);
```

Train the network with the fully labeled `ecgFrames` data set.

```
baselineNet = trainNetwork(ecgFrames,ecgLabels,layers,options);
```




Classify the test frames using the trained network and compute the mean prediction accuracy. The baseline prediction accuracy is about 95%.

```
predictLabelsAll = classify(baselineNet,testFrames,'MiniBatchSize',50);
accuracyAll = mean(cellfun(@(x,y)mean(x==y),predictLabelsAll,testLabels));
fprintf('The baseline prediction accuracy is %2.1f%%.\n',accuracyAll*100);
```

```
The baseline prediction accuracy is 94.2%.
```

### Iterative Labeling with Human in the Loop

To reduce the labeling effort, try an iterative approach: Pretend that the `ecgFrames` data set is initially unlabeled and that the data are labeled manually. In reality, the example uses the ground truth labels provided by the data set.

### Train an Initial Network

Start by selecting 25 frames from the `ecgFrames` set and labeling them manually. Train a BiLSTM network with this initial labeled set to serve as the initial step of the iterative process.

```
numInitFrames = 25;

currentTrainingSet = ecgFrames(1:numInitFrames,1);
currentTrainingLabels = ecgLabels(1:numInitFrames);
```

Set the training options to have more training epochs and a smaller mini-batch size because there are only 25 frames within the initial training data set.

```
options = trainingOptions('adam', ...
    'MaxEpochs',20, ...
```

```
    'MiniBatchSize',5, ...
    'ExecutionEnvironment','gpu', ...
    'InitialLearnRate',0.01, ...
    'LearnRateDropPeriod',6, ...
    'LearnRateSchedule','piecewise', ...
    'GradientThreshold',1, ...
    'Shuffle','every-epoch', ...
    'Plots','none',...
    'Verbose',0);
```

Train the BiLSTM network with the initial training data set and predict labels using the same test data set used to establish the performance baseline. The prediction accuracy of the networks is less than about 80%.

```
initNet = trainNetwork(currentTrainingSet,currentTrainingLabels,layers,options);
initPrediction = classify(initNet,testFrames,'MiniBatchSize',50);
initAccuracy = mean(cellfun(@(x,y)mean(x==y),initPrediction,testLabels));
fprintf('The prediction accuracy is %2.1f%%.\n',initAccuracy*100);
```

```
The prediction accuracy is 78.3%.
```

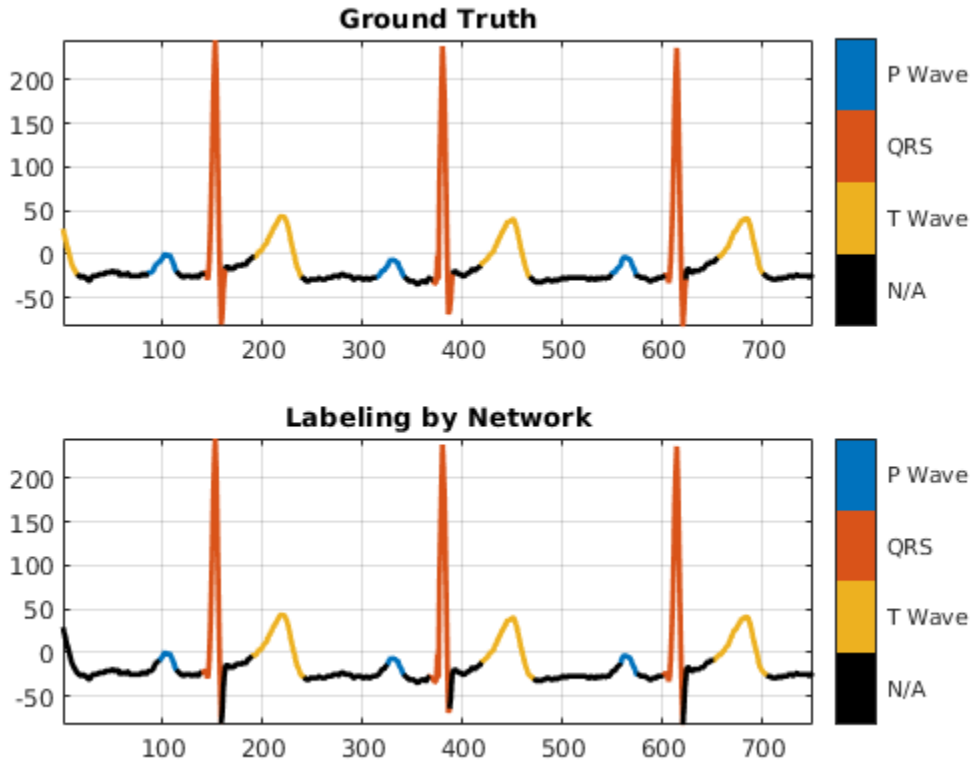
### Labeling

In the next step, select 200 new data frames from the `ecgFrames` set and feed them to the pretrained network, `initNet`, to label the signals automatically.

```
iteration = 1;
% Number of frames to label at each iteration
numFrames = 200;
% Select the next set of frames to label
indexNext = numInitFrames+1:numInitFrames+numFrames;
currentPrediction = classify(initNet,ecgFrames(indexNext),'MiniBatchSize',50);
```

Evaluate the labeling results generated by the network and compare them to the ground truth. Use the third of the selected 200-sample data frames as example: Plot its first 750 samples overlaid with the ground-truth labels and the labels predicted by the network.

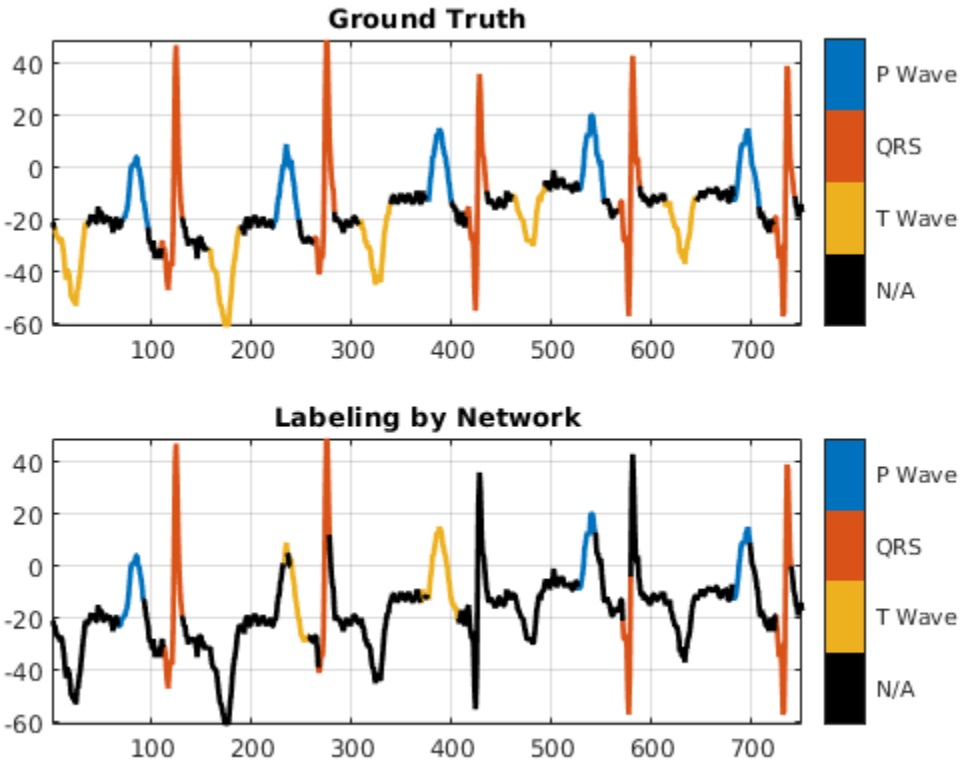
```
idx = 3;
info = ecgInfo{indexNext(idx)};
signal = QTData.Source{info.patientID}(info.channelID,info.indexID);
groundTruthLabels = ecgLabels{indexNext(idx)};
predictedLabels = currentPrediction{idx};

subplot(2,1,1)
displayWaveformLabels(signal,groundTruthLabels)
xlim([1 750])
title('Ground Truth')

subplot(2,1,2)
displayWaveformLabels(signal,predictedLabels)
xlim([1 750])
title('Labeling by Network')
```




The network did a good job labeling this frame. As a result, a human inspecting the results of the network can correct the predicted labels with little effort.

However, there are cases when the labeling performance of the network is not as strong. Consider the results obtained with the sixth of the selected 200-sample data frames.

```
idx = 6;
info = ecgInfo{indexNext(idx)};
signal = QTData.Source{info.patientID}(info.channelID,info.indexID);
groundTruthLabels = ecgLabels{indexNext(idx)};
predictedLabels = currentPrediction{idx};

figure
subplot(2,1,1)
displayWaveformLabels(signal,groundTruthLabels)
xlim([1 750])
title('Ground Truth')

subplot(2,1,2)
displayWaveformLabels(signal,predictedLabels)
xlim([1 750])
title('Labeling by Network')
```




The performance of the network on this signal is not as good. To inspect the results, a human labeler has to make several corrections to the predicted labels.

To quantify the correction effort for the 200 data frames, calculate the labeling error rate of the network and the average number of samples per frame that must be corrected by the human labeler.

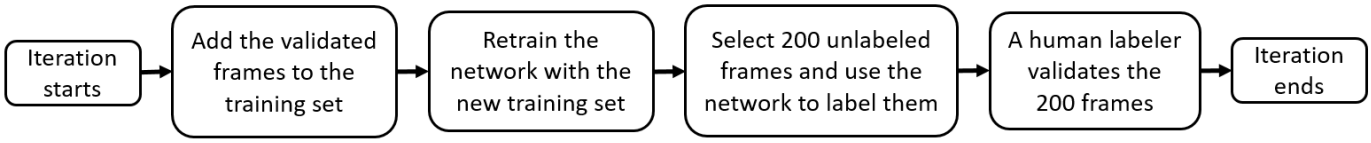
```
numSamplesPerFrame = 5000;
networkLabelingErrorRate(iteration) = 1-mean(cellfun(@(x,y)mean(x==y),currentPrediction,ecgLabels
averageNumOfCorrectionsPerFrame(iteration) = networkLabelingErrorRate(iteration) * numSamplesPerF
fprintf('The average number of corrections per frame is %2.1f.\n',averageNumOfCorrectionsPerFrame
```

```
The average number of corrections per frame is 1077.4.
```

For the first iteration, there are on average about 1000 samples per frame that must be corrected by a human.

At the end of the first iteration, the human will inspect the 200 frames and modify any label with incorrect values. With the help of the network and the human labeler, the data frames have correct labels at the end of the iteration.

On the next iteration, the 200 newly labeled frames can be added to the `currentTrainingSet` set to retrain the network and repeat the labeling iteration. This chart illustrates the workflow in each iteration after the first iteration:




### Repeat Labeling Iterations

Extend the training set by adding the newly corrected labeled frames, select another 200 data frames to be labeled, and repeat the labeling iteration until the performance is satisfactory.

```
% Include the initial training set and the 200 newly labeled data frames
maxIter = 15;
indexTraining = 1:numInitFrames+numFrames;

networkAccuracy = zeros(1,15);
networkAccuracy(iteration) = initAccuracy;

options = trainingOptions('adam', ...
    'MaxEpochs',20, ...
    'MiniBatchSize',50, ...
    'ExecutionEnvironment','gpu', ...
    'InitialLearnRate',0.01, ...
    'LearnRateDropPeriod',6, ...
    'LearnRateSchedule','piecewise', ...
    'GradientThreshold',1, ...
    'Shuffle','every-epoch', ...
    'Plots','none', ...
    'Verbose',0);

for iteration = 2:maxIter
    % Extended training data set
    currentTrainingSet = ecgFrames(indexTraining,1);
    % Emulate human correction by assigning ground-truth labels to the
    % extended training set
    currentTrainingLabels = ecgLabels(indexTraining);

    % Train network with extended training set
    currentNet = trainNetwork(currentTrainingSet,currentTrainingLabels,layers,options);

    % Predict labels for the test data set and calculate the accuracy to
    % compare to baseline performance
    currentTestSetPrediction = classify(currentNet,testFrames,'MiniBatchSize',50);
    networkAccuracy(iteration) = mean(cellfun(@(x,y)mean(x==y),currentTestSetPrediction,testLabe

    % Get another numFrames data frames for human labeler
    indexNext = indexTraining(end)+1:indexTraining(end)+numFrames;

    % Measure average number of human corrections per frame in this iteration
    currentPrediction = classify(currentNet,ecgFrames(indexNext),'MiniBatchSize',50);
    networkLabelingErrorRate(iteration) = 1-mean(cellfun(@(x,y)mean(x==y),currentPrediction,ecgLa
    averageNumOfCorrectionsPerFrame(iteration) = networkLabelingErrorRate(iteration) * numSamples

    indexTraining = 1:indexNext(end);
end
```

### Labeling Performance

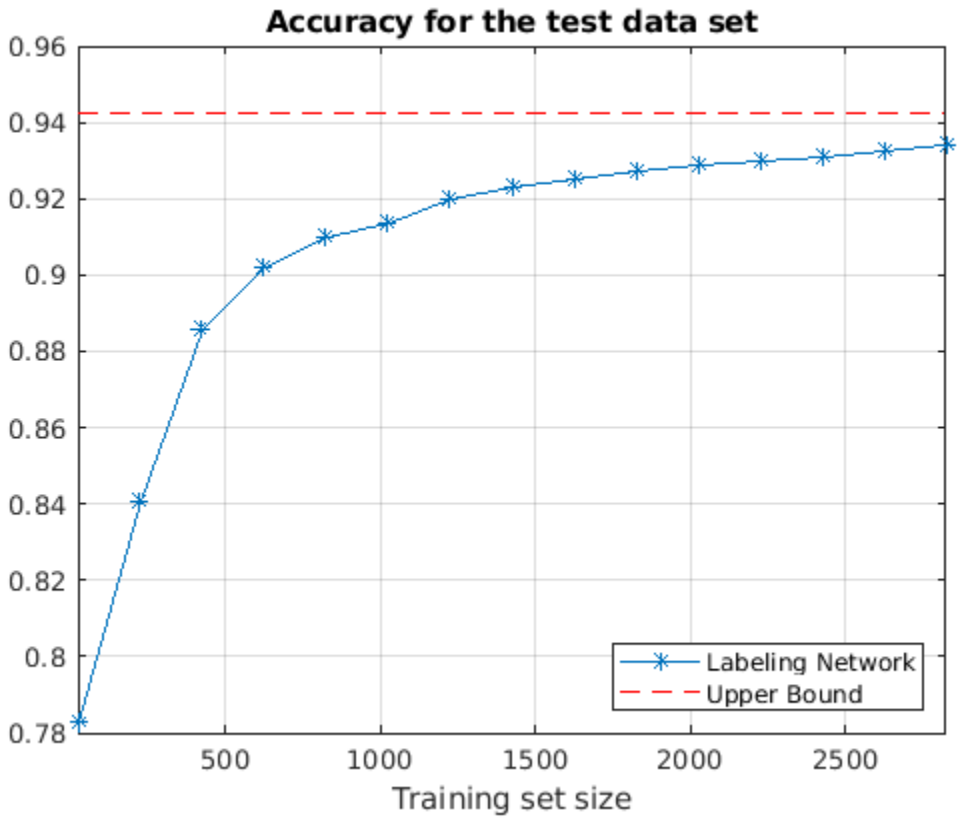
After 15 labeling iterations, there are 2825 data frames in `currentTrainingSet`, corresponding to about half of the 6543 data frames contained in the full `ecgDataset` set. The prediction accuracy of the network trained with 2825 frames is only about 1% lower than the baseline accuracy.

```
accuDiff = accuracyAll-networkAccuracy(end);
fprintf('The accuracy difference is %2.1f%%.\n',accuDiff*100);
```

```
The accuracy difference is 0.8%.
```

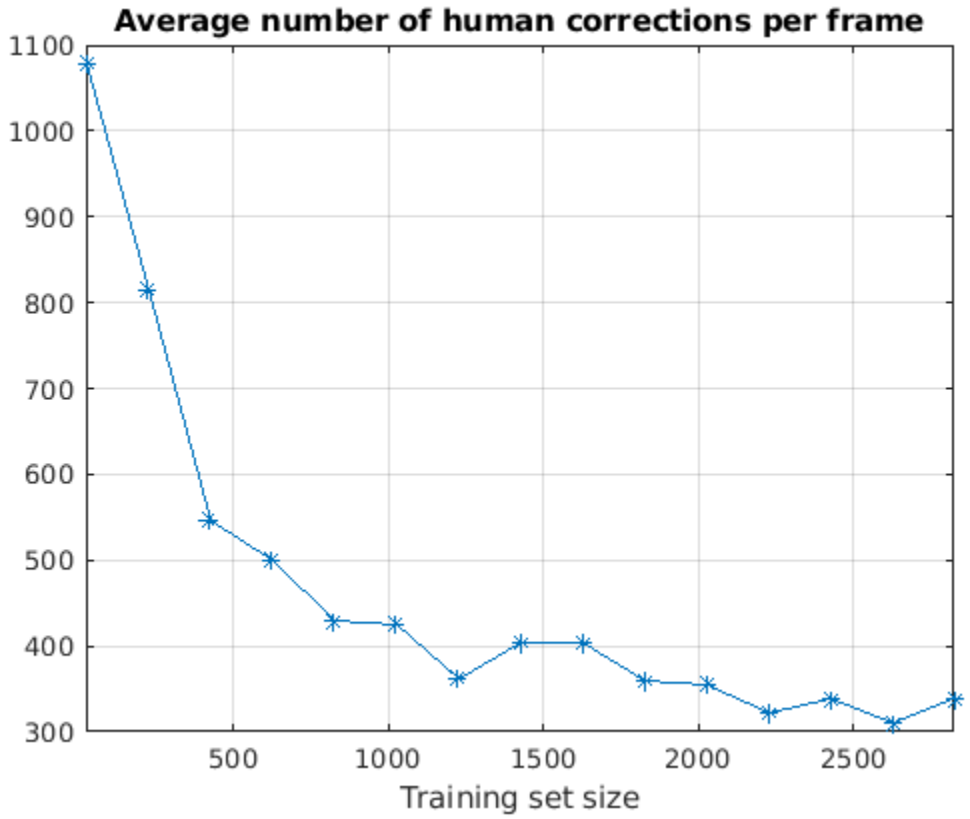
Plot the network prediction accuracy for the test data set with respect to the size of the training data set at each iteration. Show the upper bound of the accuracy obtained with the fully labeled data set. As more data frames are validated, the prediction accuracy of the network improves.

```
figure
examinedDataSize = 25:200:2825;
plot(examinedDataSize,networkAccuracy,'*-')
hold on
% Prediction accuracy upper bound
plot(examinedDataSize,ones(1,15)*accuracyAll,'r--')
grid on
xlabel('Training set size')
title('Accuracy for the test data set')
xlim([25 2825])
legend('Labeling Network','Upper Bound','Location','southeast')
```



At the end of each iteration, the average number of human corrections per frame decreases as the size of the training dataset increases. As more data frames are validated and used to train the network, less human effort is required to correct the labels of the selected frames. At the last iteration, there is an average of 300 samples per frame that require human correction.

```
figure
plot(examinedDataSize,averageNumOfCorrectionsPerFrame,'*-')
grid on
xlabel('Training set size')
title('Average number of human corrections per frame')
xlim([25 2825])
```




Throughout all 15 labeling iterations, an average of about 500 signal samples per frame required human corrections.

```
fprintf('The average number of corrections per frame is %2.1f.\n',mean(averageNumOfCorrectionsPe
```

```
The average number of corrections per frame is 465.7.
```

### Conclusion

This example showed that labeling only half of an ECG data set allows a deep network to achieve a prediction accuracy similar to that achieved by the same network when trained with the fully labeled data set. With the proposed iterative labeling workflow, a human labeler needs to look at only half of the data set and correct an average of 500 signal samples per frame. On the other hand, brute-force labeling requires looking at every frame in the data set and labeling all of its samples from scratch.

### References

[1] Goldberger, Ary L., Luis A. N. Amaral, Leon Glass, Jeffery M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H. Eugene Stanley. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." Circulation. Vol. 101, Number 23, 2000, pp. e215–e220. [Circulation Electronic Pages; http://circ.ahajournals.org/content/101/23/e215.full].

[2] Laguna, Pablo, Roger G. Mark, Ary L. Goldberger, and George B. Moody. "A Database for Evaluation of Algorithms for Measurement of QT and Other Waveform Intervals in the ECG." Computers in Cardiology. Vol.24, 1997, pp. 673–676.

[3] Laguna, Pablo, Raimon Jané, and Pere Caminal. "Automatic detection of wave boundaries in multilead ECG signals: Validation with the CSE database." Computers and Biomedical Research. Vol. 27, Number 1, 1994, pp. 45–60.

## See Also

`labeledSignalSet` | `signalLabelDefinition`

# Generate Synthetic Signals Using Conditional Generative Adversarial Network

This example shows how to generate synthetic pump signals using a conditional generative adversarial network.

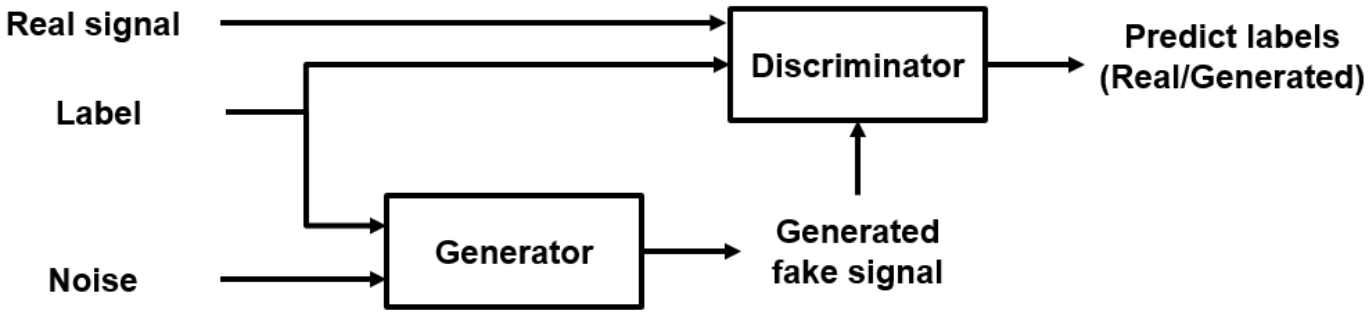
Generative adversarial networks (GANs) can be used to produce synthetic data that resembles real data input to the networks. GANs are useful when simulations are computationally expensive or experiments are costly. Conditional GANs (CGANs) can use data labels during the training process to generate data belonging to specific categories.

This example treats simulated signals obtained by a pump Simulink™ model as the "real" data that plays the role of training data set for a CGAN. The CGAN uses 1-D convolutional networks and is trained using a custom training loop and a deep learning array. In addition, this example uses principal component analysis (PCA) to visually compare the characteristics of generated and real signals.

### CGAN for Signal Synthesis

CGANs consist of two networks that train together as adversaries:

1. *Generator* network — Given a label and random array as input, this network generates data with the same structure as the training data observations corresponding to the same label. The objective of the generator is to generate labeled data that the discriminator classifies as "real."
2. *Discriminator* network — Given batches of labeled data containing observations from both training data and generated data from the generator, this network attempts to classify the observations as "real" or "generated." The objective of the discriminator is to not be "fooled" by the generator when given batches of both real and generated labeled data.




Ideally, these strategies result in a generator that generates convincingly realistic data corresponding to the input labels and a discriminator that has learned strong features characteristic of the training data for each label.

### Load Data

The simulated data is generated by the pump Simulink model presented in the “Multi-Class Fault Detection Using Simulated Data” (Predictive Maintenance Toolbox) example. The Simulink model is configured to model three types of faults: cylinder leaks, blocked inlets, and increased bearing friction. The data set contains 1575 pump output flow signals, of which 760 are healthy signals and 815 have a single fault, combinations of two faults, or combinations of three faults. Each signal has 1201 signal samples with a sample rate of 1000 Hz.

Download and unzip the data in your temporary directory, whose location is specified by MATLAB® `tempdir` command. If you have the data in a folder different from that specified by `tempdir`, change the directory name in the following code.

```
% Download the data
dataURL = 'https://ssd.mathworks.com/supportfiles/SPT/data/PumpSignalGAN.zip';
saveFolder = fullfile(tempdir,'PumpSignalGAN');
zipFile = fullfile(tempdir,'PumpSignalGAN.zip');
if ~exist(saveFolder,'dir')
    websave(zipFile,dataURL);
end

% Unzip the data
unzip(zipFile,saveFolder)
```

The zip file contains the training data set and a pretrained CGAN:

- `simulatedDataset` — Simulated signals and their corresponding categorical labels
- `GANModel` — Generator and discriminator trained on the simulated data

Load the training data set and standardize the signals to have zero mean and unit variance.

```
load(fullfile(saveFolder,'simulatedDataset.mat')) % load data set
meanFlow = mean(flow,2);
flowNormalized = flow-meanFlow;
stdFlow = std(flowNormalized(:));
flowNormalized = flowNormalized/stdFlow;
```

Healthy signals are labeled as 1 and faulty signals are labeled as 2.

**Define Generator Network**

Define the following two-input network, which generates flow signals given 1-by-1-by-100 arrays of random values and corresponding labels.

The network:

- Projects and reshapes the 1-by-1-by-100 arrays of noise to 4-by-1-by-1024 arrays by a custom layer.
- Converts the categorical labels to embedding vectors and reshapes them to a 4-by-1-by-1 arrays.
- Concatenates the results from the two inputs along the channel dimension. The output is a 4-by-1-by-1025 array.
- Upsamples the resulting arrays to 1201-by-1-by-1 arrays using a series of 1-D transposed convolution layers with batch normalization and ReLU layers.

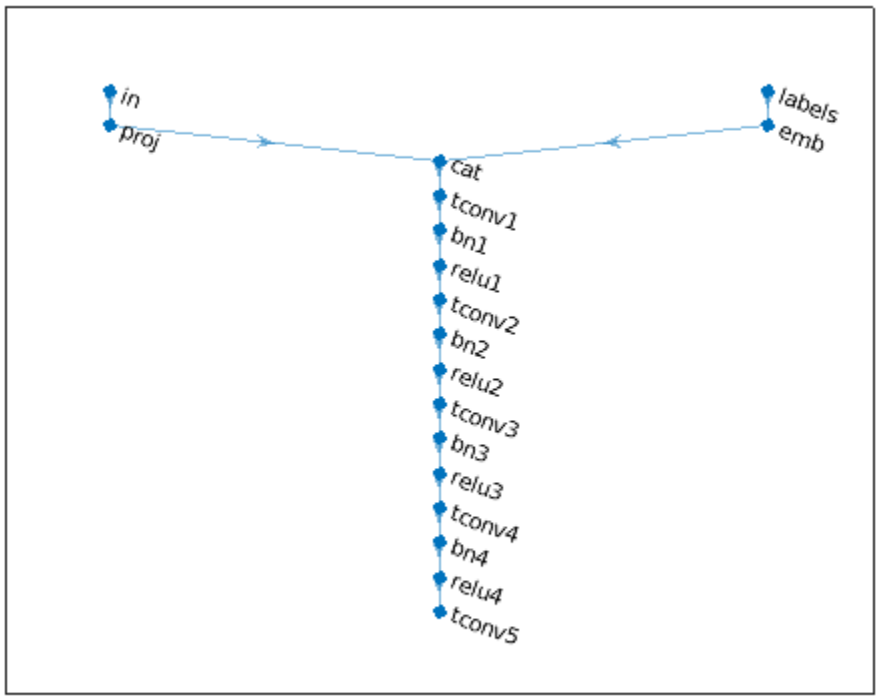
To project and reshape the noise input, use the custom layer `projectAndReshapeLayer`, attached to this example as a supporting file. The `projectAndReshapeLayer` object upscales the input using a fully connected layer and reshapes the output to the specified size.

To input the labels into the network, use an `imageInputLayer` object and specify a size of 1-by-1. To embed and reshape the label input, use the custom layer `embedAndReshapeLayer`, attached to this example as a supporting file. The `embedAndReshapeLayer` object converts a categorical label to a one-channel array of the specified size using an embedding and a fully connected operation. For categorical inputs, use an embedding dimension of 100.

```
% Generator Network

numFilters = 64;
numLatentInputs = 100;
projectionSize = [4 1 1024];
numClasses = 2;
embeddingDimension = 100;

layersGenerator = [
    imageInputLayer([1 1 numLatentInputs],'Normalization','none','Name','in')
    projectAndReshapeLayer(projectionSize,numLatentInputs,'proj');
    concatenationLayer(3,2,'Name','cat');
    transposedConv2dLayer([5 1],8*numFilters,'Name','tconv1')
    batchNormalizationLayer('Name','bn1','Epsilon',5e-5)
    reluLayer('Name','relu1')
    transposedConv2dLayer([10 1],4*numFilters,'Stride',4,'Cropping',[1 0],'Name','tconv2')
    batchNormalizationLayer('Name','bn2','Epsilon',5e-5)
    reluLayer('Name','relu2')
    transposedConv2dLayer([12 1],2*numFilters,'Stride',4,'Cropping',[1 0],'Name','tconv3')
    batchNormalizationLayer('Name','bn3','Epsilon',5e-5)
    reluLayer('Name','relu3')
    transposedConv2dLayer([5 1],numFilters,'Stride',4,'Cropping',[1 0],'Name','tconv4')
    batchNormalizationLayer('Name','bn4','Epsilon',5e-5)
    reluLayer('Name','relu4')
    transposedConv2dLayer([7 1],1,'Stride',2,'Cropping',[1 0],'Name','tconv5')
    ];

lgraphGenerator = layerGraph(layersGenerator);

layers = [
    imageInputLayer([1 1],'Name','labels','Normalization','none')
    embedAndReshapeLayer(projectionSize(1:2),embeddingDimension,numClasses,'emb')];

lgraphGenerator = addLayers(lgraphGenerator,layers);
lgraphGenerator = connectLayers(lgraphGenerator,'emb','cat/in2');
```

Plot the network structure for the generator.

```
plot(lgraphGenerator)
```




To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetGenerator = dlnetwork(lgraphGenerator);
```

**Define Discriminator Network**

Define the following two-input network, which classifies real and generated 1201-by-1 signals given a set of signals and their corresponding labels.

This network:

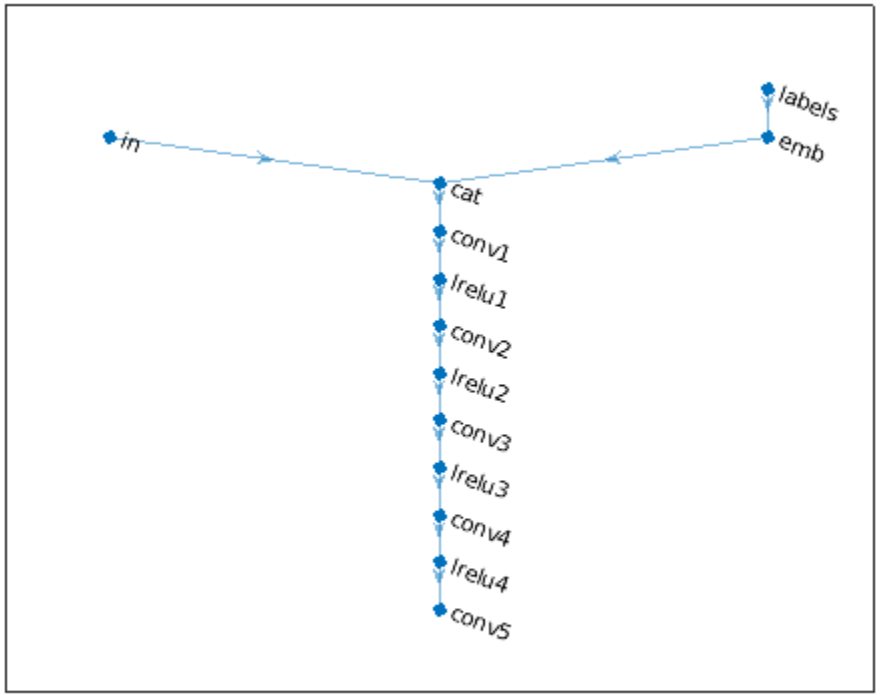
- Takes 1201-by-1-by-1 signals as input.
- Converts categorical labels to embedding vectors and reshapes them to a 1201-by-1-by-1 arrays.
- Concatenates the results from the two inputs along the channel dimension. The output is a 1201-by-1-by-1025 array.
- Downsamples the resulting arrays to scalar prediction scores, which are 1-by-1-by-1 arrays, using a series of 1-D convolution layers with leaky ReLU layers with a scale of 0.2.

```
% Discriminator Network

scale = 0.2;
inputSize = [1201 1 1];

layersDiscriminator = [
    imageInputLayer(inputSize,'Normalization','none','Name','in')
```

```
    concatenationLayer(3,2,'Name','cat')
    convolution2dLayer([17 1],8*numFilters,'Stride',2,'Padding',[1 0],'Name','conv1')
    leakyReluLayer(scale,'Name','lrelu1')
    convolution2dLayer([16 1],4*numFilters,'Stride',4,'Padding',[1 0],'Name','conv2')
    leakyReluLayer(scale,'Name','lrelu2')
    convolution2dLayer([16 1],2*numFilters,'Stride',4,'Padding',[1 0],'Name','conv3')
    leakyReluLayer(scale,'Name','lrelu3')
    convolution2dLayer([8 1],numFilters,'Stride',4,'Padding',[1 0],'Name','conv4')
    leakyReluLayer(scale,'Name','lrelu4')
    convolution2dLayer([8 1],1,'Name','conv5')];

lgraphDiscriminator = layerGraph(layersDiscriminator);

layers = [
    imageInputLayer([1 1],'Name','labels','Normalization','none')
    embedAndReshapeLayer(inputSize,embeddingDimension,numClasses,'emb')];

lgraphDiscriminator = addLayers(lgraphDiscriminator,layers);
lgraphDiscriminator = connectLayers(lgraphDiscriminator,'emb','cat/in2');
```

Plot the network structure for the discriminator.

```
plot(lgraphDiscriminator)
```




To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetDiscriminator = dlnetwork(lgraphDiscriminator);
```

### Train Model

Train the CGAN model using a custom training loop. Loop over the training data and update the network parameters at each iteration. To monitor the training progress, display generated healthy and faulty signals using two fixed arrays of random values to input into the generator as well as a plot of the scores of the two networks.

For each epoch, shuffle the training data and loop over mini-batches of data.

For each mini-batch:

- Generate a `dlarray` (Deep Learning Toolbox) 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 function `modelGradients`.
- Update the network parameters using the `adamupdate` (Deep Learning Toolbox) function.

The helper function `modelGradients` takes as input the generator and discriminator networks, a mini-batch of input data, and an array of random values, and returns the gradients of the loss with respect to the learnable parameters in the networks and the scores of the two networks. The loss function is defined in the helper function `ganLoss`.

### Specify Training Options

Set the training parameters.

```
params.numLatentInputs = numLatentInputs;
params.numClasses = numClasses;
params.sizeData = [inputSize length(labels)];
params.numEpochs = 1000;
params.miniBatchSize = 256;

% Specify the options for Adam optimizer
params.learnRate = 0.0002;
params.gradientDecayFactor = 0.5;
params.squaredGradientDecayFactor = 0.999;
```

Set the execution environment to run the CGANs on the CPU. To run the CGANs on the GPU, set `executionEnvironment` to "`gpu`" or select the "Run on GPU" option in Live Editor. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

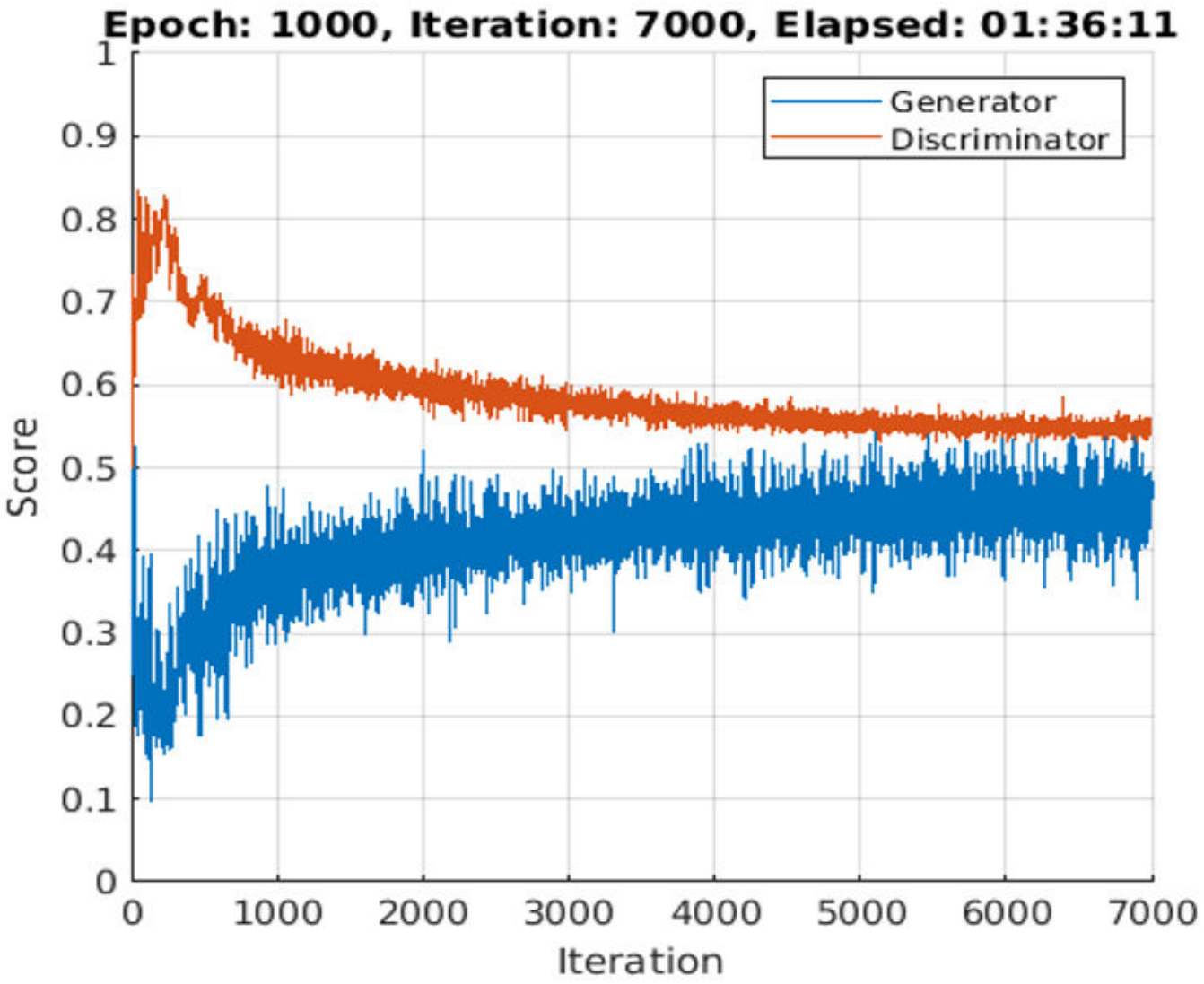
```
executionEnvironment = Run on CPU ;
params.executionEnvironment = executionEnvironment;
```

Skip the training process by loading the pretrained network. To train the network on your computer, set `trainNow` to `true` or select the "Train CGAN now" option in Live Editor.

```
trainNow = Use trained CGAN ;
if trainNow
    % Train the CGAN
    [dlnetGenerator,dlnetDiscriminator] = trainGAN(dlnetGenerator, ...
        dlnetDiscriminator,flowNormalized,labels,params); %#ok
```

```
else
    % Use pretrained CGAN (default)
    load(fullfile(tempdir,'PumpSignalGAN','GANModel.mat')) % load data set
end
```

The training plot below shows an example of scores of the generator and discriminator networks. To learn more about how to interpret the network scores, see “Monitor GAN Training Progress and Identify Common Failure Modes” (Deep Learning Toolbox). In this example, the scores of both the generator and discriminator converge close to 0.5, indicating that the training performance is good.




### Synthesize Flow Signals

Create a `dlarray` object containing a batch of 2000 1-by-1-by-100 arrays of random values to input into the generator network. Reset the random number generator for reproducible results.

```
rng default

numTests = 2000;
ZNew = randn(1,1,numLatentInputs,numTests,'single');
dlZNew = dlarray(ZNew,'SSCB');
```

Specify that the first 1000 random arrays are healthy and the rest are faulty.

```
TNew = ones(1,1,1,numTests,'single');
TNew(1,1,1,numTests/2+1:end) = single(2);
dlTNew = dlarray(TNew,'SSCB');
```

To generate signals using the GPU, convert the data to `gpuArray` objects.

```
if executionEnvironment == "gpu"
    dlZNew = gpuArray(dlZNew);
    dlTNew = gpuArray(dlTNew);
end
```

Use the `predict` function on the generator with the batch of 1-by-1-by-100 arrays of random values and labels to generate synthetic signals and revert the standardization step that you performed on the original flow signals.

```
dlXGeneratedNew = predict(dlnetGenerator,dlZNew,dlTNew)*stdFlow+meanFlow;
```

### Signal Feature Visualization

Unlike images and audio signals, general signals have characteristics that make them difficult for human perception to tell apart. To compare real and generated signals or healthy and faulty signals, you can apply principal component analysis (PCA) to the statistical features of the real signals and then project the features of the generated signals to the same PCA subspace.

### Feature Extraction

Combine the original real signal and the generated signals in one data matrix. Use the helper function `extractFeature` to extract the feature including common signal statistics such as the mean and variance as well as spectral characteristics.

```
idxGenerated = 1:numTests;
idxReal = numTests+1:numTests+size(flow,2);

XGeneratedNew = squeeze(extractdata(gather(dlXGeneratedNew)));
x = [XGeneratedNew single(flow)];

features = zeros(size(x,2),14,'like',x);

for ii = 1:size(x,2)
    features(ii,:) = extractFeatures(x(:,ii));
end
```

Each row of `features` corresponds to the features of one signal.

Modify the labels for the generated healthy and faulty signals as well as real healthy and faulty signals.

```
L = [squeeze(TNew)+2;labels.'];
```

The labels now have these definitions:

- 1 — Generated healthy signals
- 2 — Generated faulty signals
- 3 — Real healthy signals
- 4 — Real faulty signals

### Principal Component Analysis

Perform PCA on the features of the real signals and project the features of the generated signals to the same PCA subspace. `W` is the coefficient and `Y` is the score.

```
% PCA via svd
featuresReal = features(idxReal,:);
mu = mean(featuresReal,1);
[~,S,W] = svd(featuresReal-mu);
S = diag(S);
Y = (features-mu)*W;
```

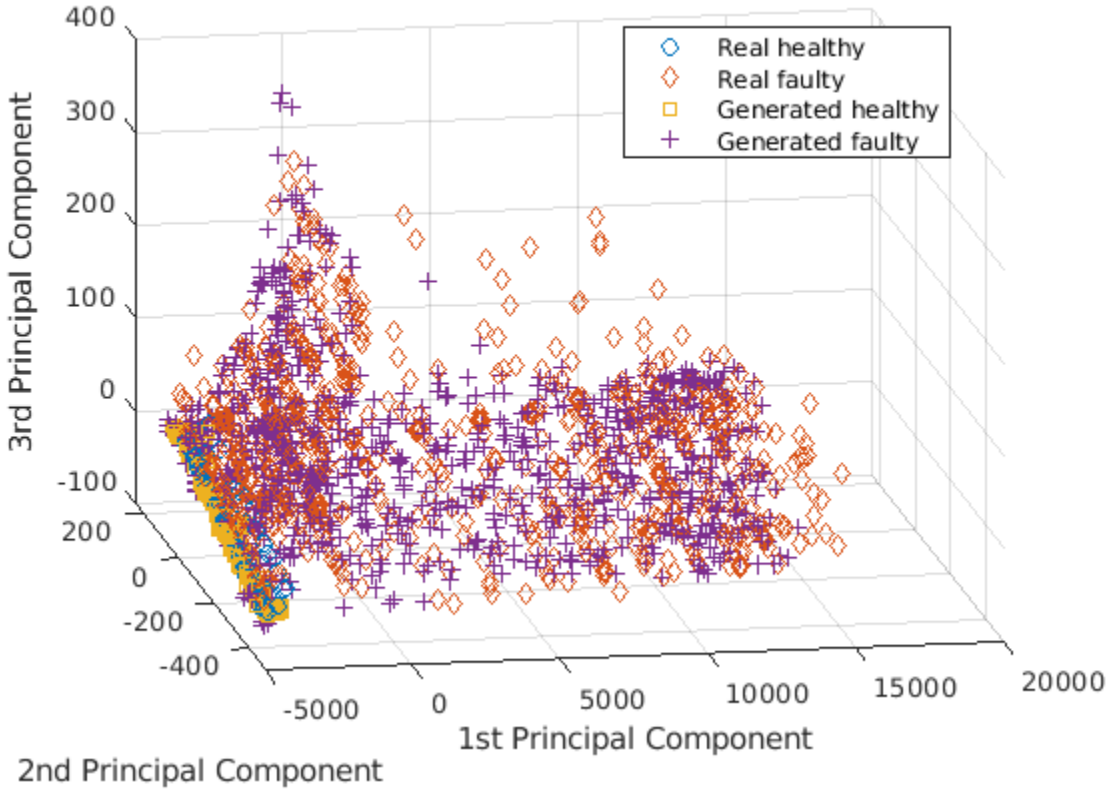
From the singular vector `S`, the first three singular values make up 99% of the energy in `S`. You can visualize the signal features by taking advantage of the first three principal components.

```
sum(S(1:3))/sum(S)
```

```
ans = single
    0.9923
```

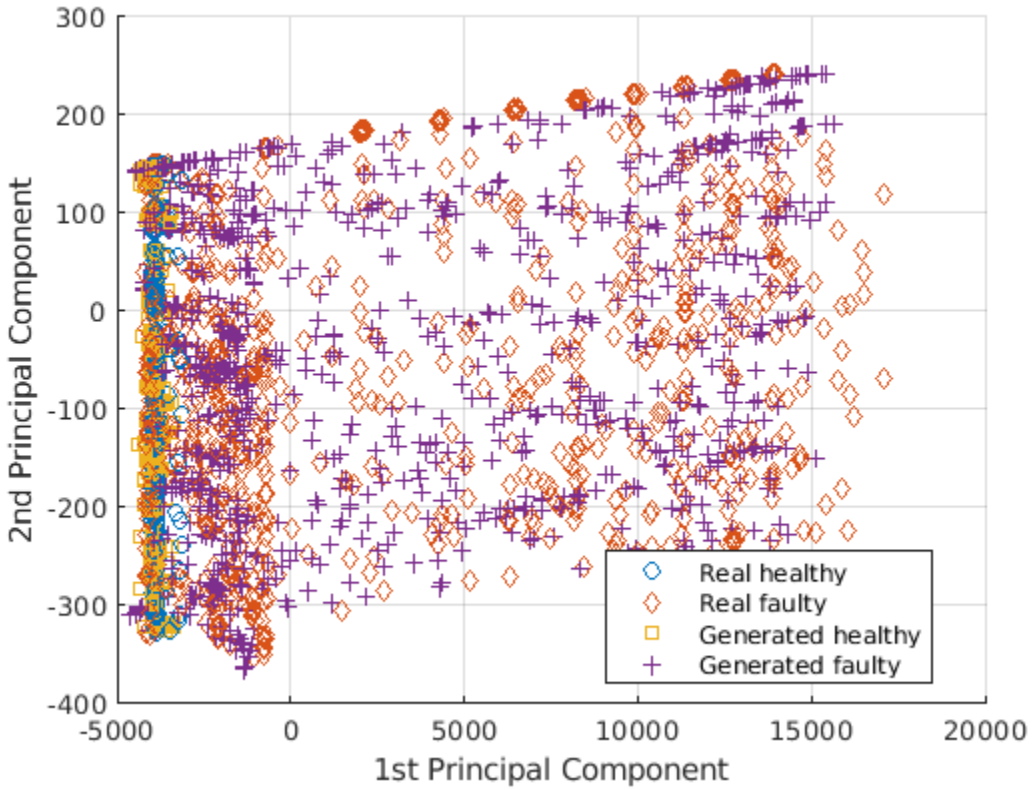
Plot the features of all the signals using the first three principal components. In the PCA subspace, the distribution of the generated signals is similar to the distribution of the real signals.

```
idxHealthyR = L==1;
idxFaultR = L==2;

idxHealthyG = L==3;
idxFaultG = L==4;

pp = Y(:,1:3);

figure
scatter3(pp(idxHealthyR,1),pp(idxHealthyR,2),pp(idxHealthyR,3),'o')
xlabel('1st Principal Component')
ylabel('2nd Principal Component')
zlabel('3rd Principal Component')
hold on
scatter3(pp(idxFaultR,1),pp(idxFaultR,2),pp(idxFaultR,3),'d')
scatter3(pp(idxHealthyG,1),pp(idxHealthyG,2),pp(idxHealthyG,3),'s')
scatter3(pp(idxFaultG,1),pp(idxFaultG,2),pp(idxFaultG,3),'+')
view(-10,20)
legend('Real healthy','Real faulty','Generated healthy','Generated faulty', ...
    'Location','Best')
hold off
```




To better capture the difference between the real signals and generated signals, plot the subspace using the first two principal components.

```
view(2)
```




Healthy and faulty signals lie in the same area of the PCA subspace regardless of their being real or generated, demonstrating that the generated signals have features similar to those of the real signals.

**Predict Labels of Real Signals**

To further illustrate the performance of the CGAN, train an SVM classifier based on the generated signals and then predict whether a real signal is healthy or faulty.

Set the generated signals as the training data set and the real signals as the test data set. Change the numeric labels to character vectors.

```
LABELS = {'Healthy','Faulty'};
strL = LABELS([squeeze(TNew);labels.']).';

dataTrain = features(idxGenerated,:);
dataTest = features(idxReal,:);

labelTrain = strL(idxGenerated);
labelTest = strL(idxReal);

predictors = dataTrain;
response = labelTrain;
cvp = cvpartition(size(predictors,1),'KFold',5);
```

Train an SVM classifier using the generated signals.

```
SVMClassifier = fitcsvm( ...
    predictors(cvp.training(1),:), ...
    response(cvp.training(1)),'KernelFunction','polynomial', ...
    'PolynomialOrder',2, ...
    'KernelScale','auto', ...
    'BoxConstraint',1, ...
    'ClassNames',LABELS, ...
    'Standardize',true);
```

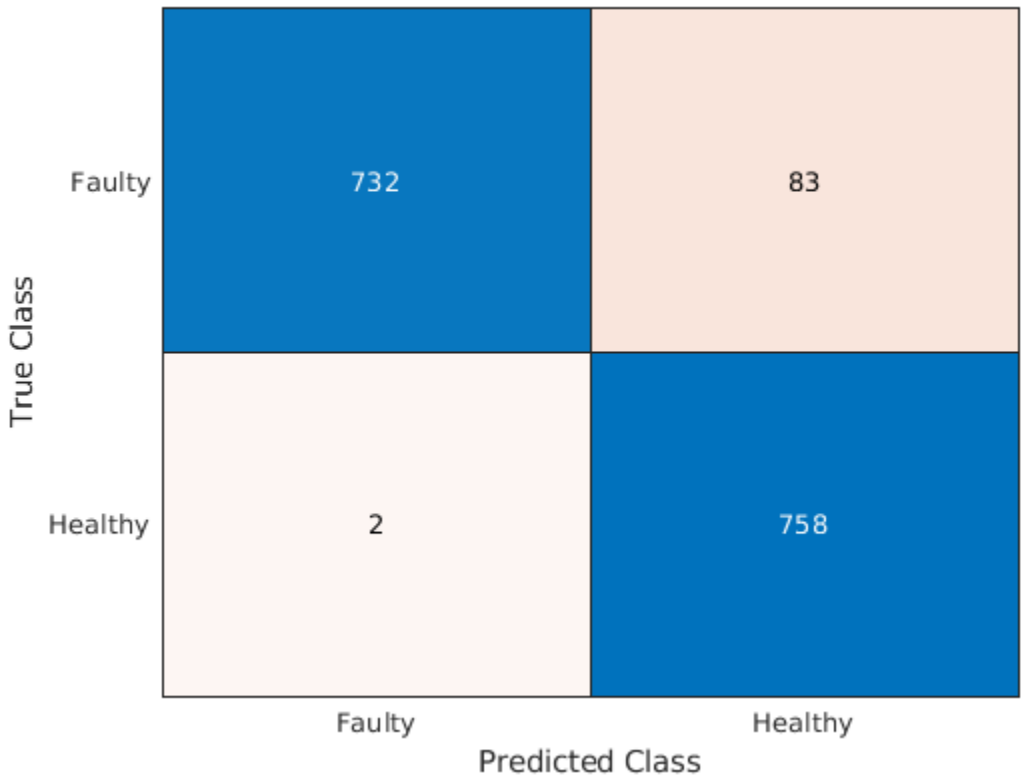
Use the trained classifier to obtain the predicted labels for the real signals. The classifier achieves a prediction accuracy above 90%.

```
actualValue = labelTest;
predictedValue = predict(SVMClassifier,dataTest);
predictAccuracy = mean(cellfun(@strcmp,actualValue,predictedValue))
```

```
predictAccuracy = 0.9460
```

Use a confusion matrix to view detailed information about prediction performance for each category. The confusion matrix shows that, in each category, the classifier trained based on the generated signals achieves a high degree of accuracy.

```
figure
confusionchart(actualValue,predictedValue)
```

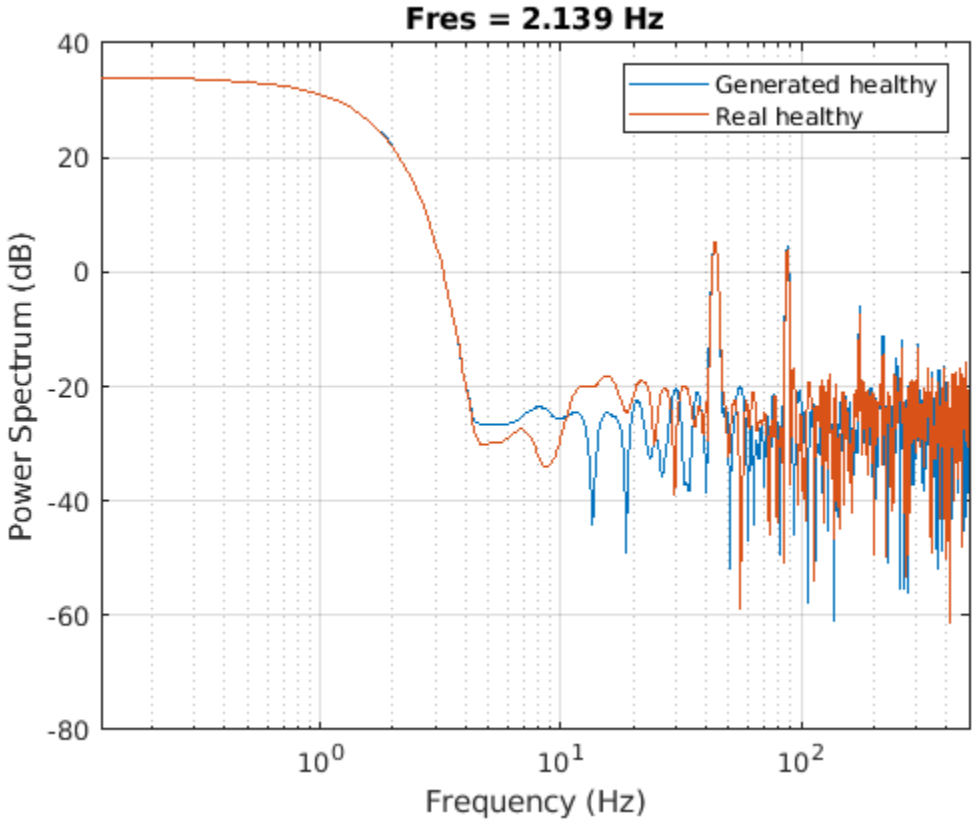


### Case Study

Compare the spectral characteristics of real and generated signals. Due to the nondeterministic behavior of GPU training, if you train the CGAN model yourself, your results might differ from the results in this example.

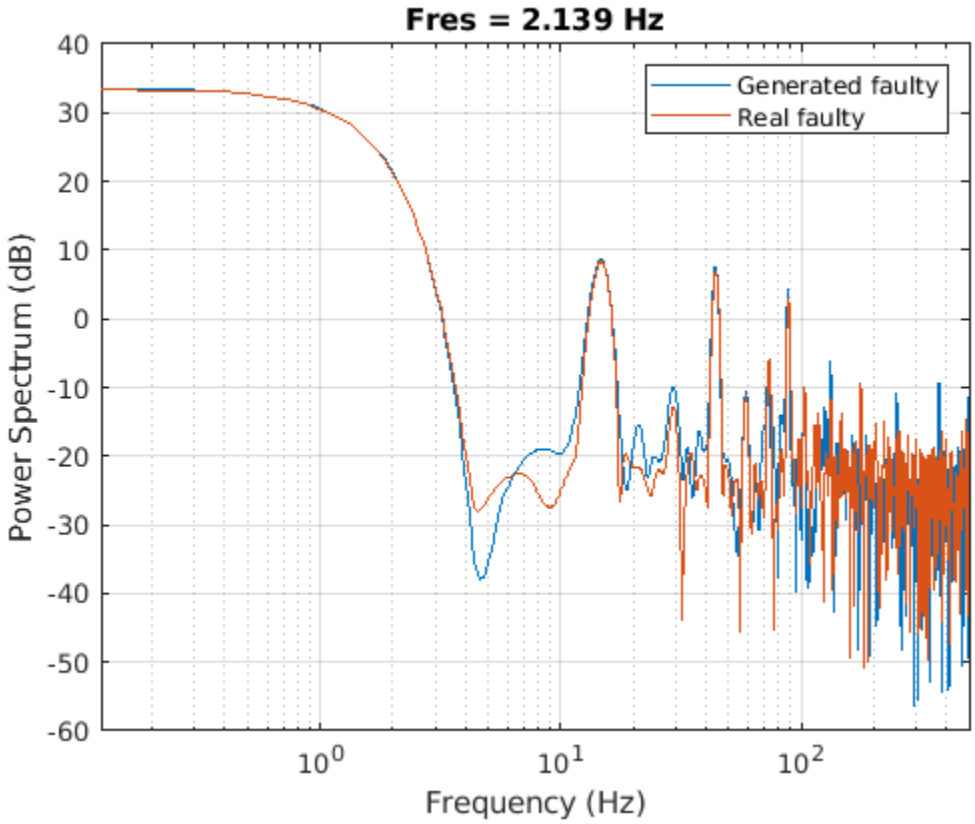
The pump motor speed is 950 rpm, or 15.833 Hz, and since the pump has three cylinders the flow is expected to have a fundamental at 3 times 15.833 Hz, or 47.5 Hz, and harmonics at multiples of 47.5 Hz. Plot the spectrum for one case of the real and generated healthy signals. From the plot, the generated healthy signal has relatively high power values at 47.5 Hz and 2 times 47.5 Hz, which is exactly the same as the real healthy signal.

```
Fs = 1000;
pspectrum([x(:,1) x(:,2006)],Fs)
set(gca,'XScale','log')
legend('Generated healthy','Real healthy')
```



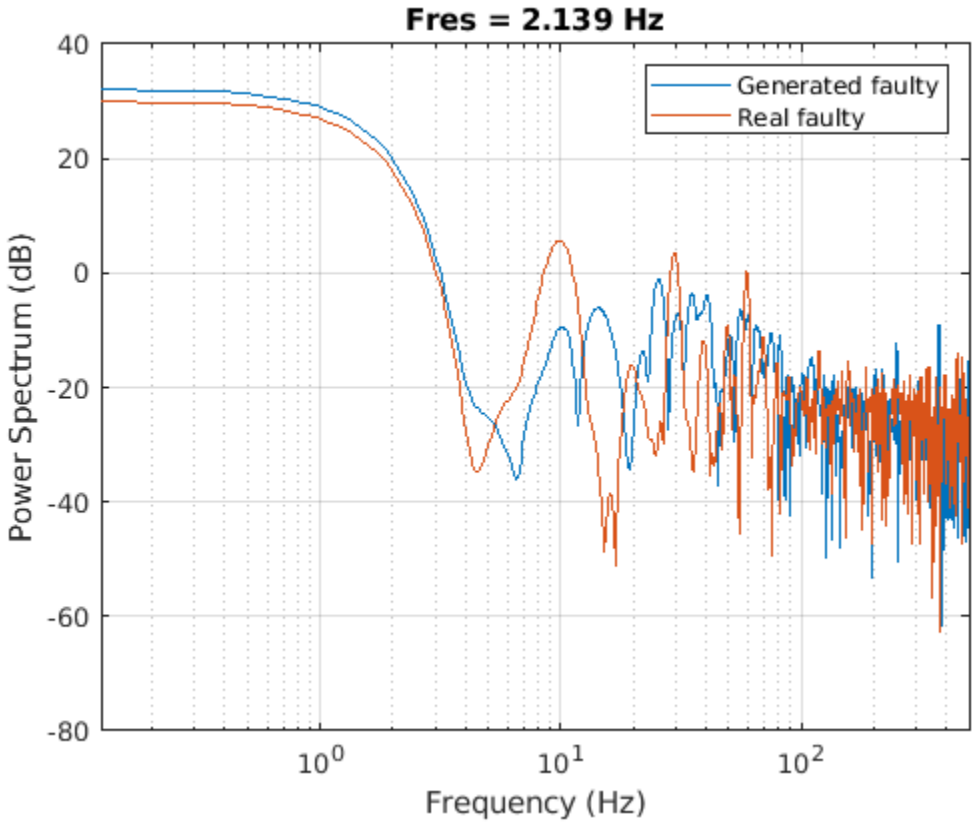

If faults exist, resonances will occur at the pump motor speed, 15.833 Hz, and its harmonics. Plot the spectra for one case of real and generated faulty signals. The generated signal has relatively high power values at around 15.833 Hz and its harmonics, which is similar to the real faulty signal.

```
pspectrum([x(:,1011) x(:,2100)],Fs)
set(gca,'XScale','log')
legend('Generated faulty','Real faulty')
```




Plot spectra for another case of real and generated faulty signals. The spectral characteristics of the generated faulty signals do not match the theoretical analysis very well and are different from the real faulty signal. The CGAN can still be possibly improved by tuning the network structure or hyperparameters.

```
pspectrum([x(:,1001) x(:,2600)],Fs)
set(gca,'XScale','log')
legend('Generated faulty','Real faulty')
```




### Computation Time

The Simulink simulation takes about 14 hours to generate 2000 pump flow signals. This duration can be reduced to about 1.7 hours with eight parallel workers if you have Parallel Computing Toolbox™.

The CGAN takes 1.5 hours to train and 70 seconds to generate the same amount of synthetic data with an NVIDIA Titan V GPU.

## See Also

`adamupdate` | `dlarray` | `dlfeval` | `gpuArray`

## More About

- “Train Conditional Generative Adversarial Network (CGAN)” (Deep Learning Toolbox)
- “Monitor GAN Training Progress and Identify Common Failure Modes” (Deep Learning Toolbox)
- “Multi-Class Fault Detection Using Simulated Data” (Predictive Maintenance Toolbox)

# Classify Time Series Using Wavelet Analysis and Deep Learning

This example shows how to classify human electrocardiogram (ECG) signals using the continuous wavelet transform (CWT) and a deep convolutional neural network (CNN).

Training a deep CNN from scratch is computationally expensive and requires a large amount of training data. In various applications, a sufficient amount of training data is not available, and synthesizing new realistic training examples is not feasible. In these cases, leveraging existing neural networks that have been trained on large data sets for conceptually similar tasks is desirable. This leveraging of existing neural networks is called transfer learning. In this example we adapt two deep CNNs, GoogLeNet and SqueezeNet, pretrained for image recognition to classify ECG waveforms based on a time-frequency representation.

GoogLeNet and SqueezeNet are deep CNNs originally designed to classify images in 1000 categories. We reuse the network architecture of the CNN to classify ECG signals based on images from the CWT of the time series data. The data used in this example are publicly available from PhysioNet.

### Data Description

In this example, you use ECG data obtained from three groups of people: persons with cardiac arrhythmia (ARR), persons with congestive heart failure (CHF), and persons with normal sinus rhythms (NSR). In total you use 162 ECG recordings from three PhysioNet databases: MIT-BIH Arrhythmia Database [3][7], MIT-BIH Normal Sinus Rhythm Database [3], and The BIDMC Congestive Heart Failure Database [1][3]. More specifically, 96 recordings from persons with arrhythmia, 30 recordings from persons with congestive heart failure, and 36 recordings from persons with normal sinus rhythms. The goal is to train a classifier to distinguish between ARR, CHF, and NSR.

### Download Data

The first step is to download the data from the GitHub repository. To download the data from the website, click `Clone or download` and select `Download ZIP`. Save the file `physionet_ECG_data-master.zip` in a folder where you have write permission. The instructions for this example assume you have downloaded the file to your temporary directory, `tempdir`, in MATLAB. Modify the subsequent instructions for unzipping and loading the data if you choose to download the data in folder different from `tempdir`. If you are familiar with Git, you can download the latest version of the tools (git) and obtain the data from a system command prompt using `git clone https://github.com/mathworks/physionet_ECG_data/`.

After downloading the data from GitHub, unzip the file in your temporary directory.

```
unzip(fullfile(tempdir,'physionet_ECG_data-master.zip'),tempdir)
```

Unzipping creates the folder `physionet-ECG_data-master` in your temporary directory. This folder contains the text file `README.md` and `ECGData.zip`. The `ECGData.zip` file contains

- `ECGData.mat`
- `Modified_physionet_data.txt`
- `License.txt`

`ECGData.mat` holds the data used in this example. The text file, `Modified_physionet_data.txt`, is required by PhysioNet's copying policy and provides the source attributions for the data as well as a description of the preprocessing steps applied to each ECG recording.

Unzip `ECGData.zip` in `physionet-ECG_data-master`. Load the data file into your MATLAB workspace.

```
unzip(fullfile(tempdir,'physionet_ECG_data-master','ECGData.zip'),...
    fullfile(tempdir,'physionet_ECG_data-master'))
load(fullfile(tempdir,'physionet_ECG_data-master','ECGData.mat'))
```

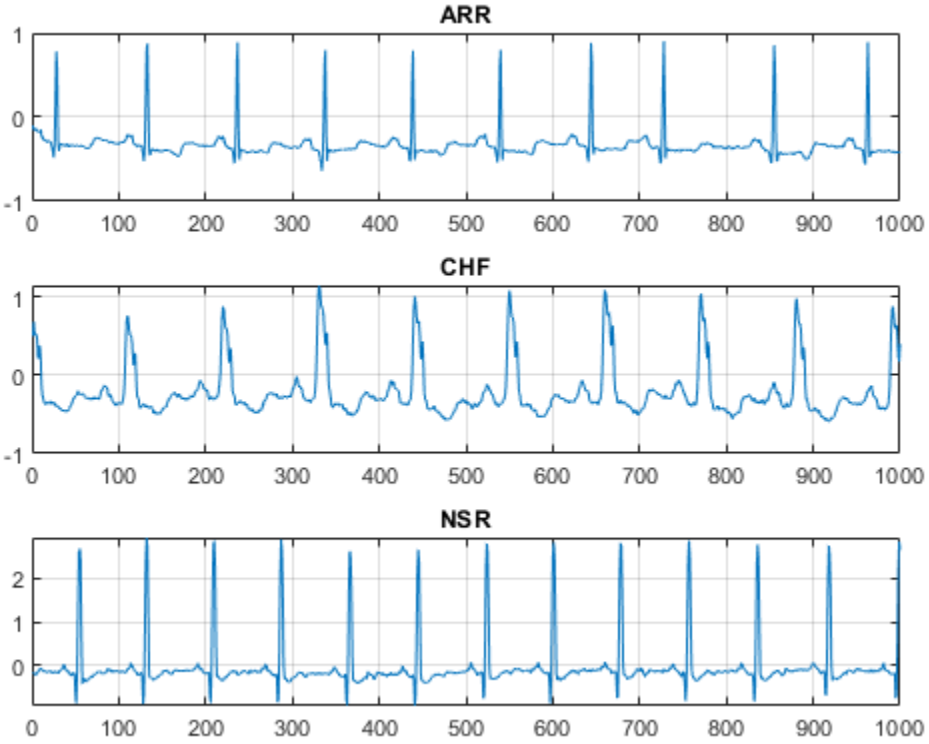
`ECGData` is a structure array with two fields: `Data` and `Labels`. The `Data` field is a 162-by-65536 matrix where each row is an ECG recording sampled at 128 hertz. `Labels` is a 162-by-1 cell array of diagnostic labels, one for each row of `Data`. The three diagnostic categories are: `'ARR'`, `'CHF'`, and `'NSR'`.

To store the preprocessed data of each category, first create an ECG data directory `dataDir` inside `tempdir`. Then create three subdirectories in `'data'` named after each ECG category. The helper function `helperCreateECGDirectories` does this. `helperCreateECGDirectories` accepts `ECGData`, the name of an ECG data directory, and the name of a parent directory as input arguments. You can replace `tempdir` with another directory where you have write permission. You can find the source code for this helper function in the Supporting Functions section at the end of this example.

```
parentDir = tempdir;
dataDir = 'data';
helperCreateECGDirectories(ECGData,parentDir,dataDir)
```

Plot a representative of each ECG category. The helper function `helperPlotReps` does this. `helperPlotReps` accepts `ECGData` as input. You can find the source code for this helper function in the Supporting Functions section at the end of this example.

```
helperPlotReps(ECGData)
```



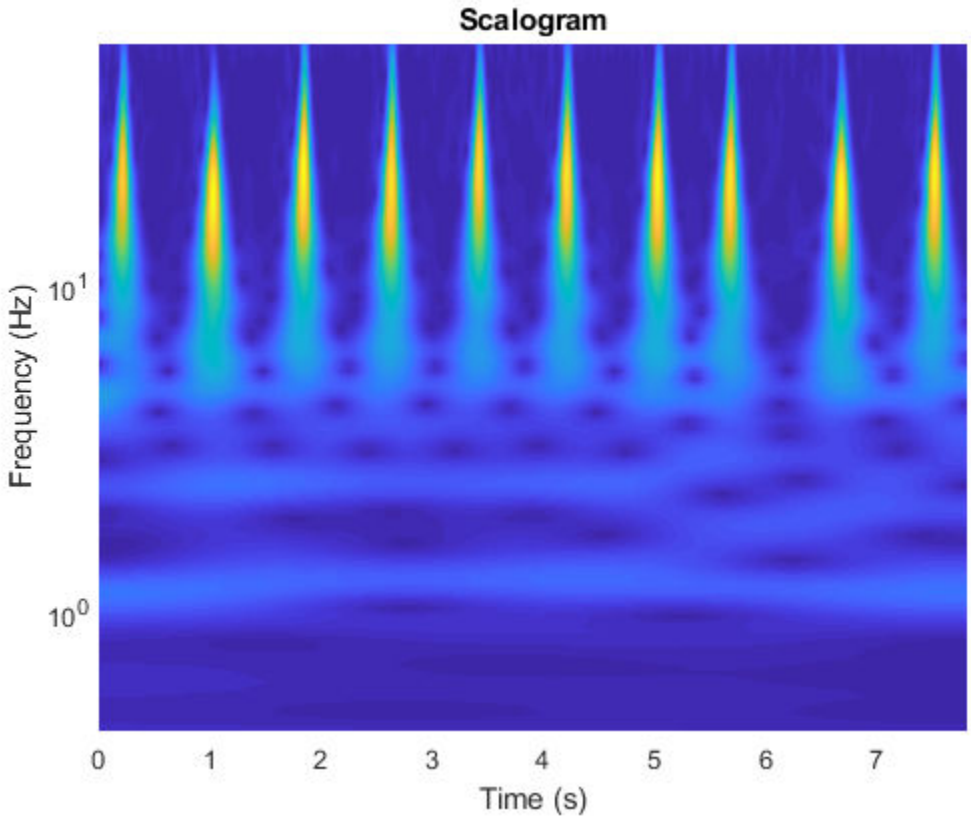

### Create Time-Frequency Representations

After making the folders, create time-frequency representations of the ECG signals. These representations are called scalograms. A scalogram is the absolute value of the CWT coefficients of a signal.

To create the scalograms, precompute a CWT filter bank. Precomputing the CWT filter bank is the preferred method when obtaining the CWT of many signals using the same parameters.

Before generating the scalograms, examine one of them. Create a CWT filter bank using `cwtfilterbank` (Wavelet Toolbox) for a signal with 1000 samples. Use the filter bank to take the CWT of the first 1000 samples of the signal and obtain the scalogram from the coefficients.

```
Fs = 128;
fb = cwtfilterbank('SignalLength',1000,...
    'SamplingFrequency',Fs,...
    'VoicesPerOctave',12);
sig = ECGData.Data(1,1:1000);
[cfs,frq] = wt(fb,sig);
t = (0:999)/Fs;figure;pcolor(t,frq,abs(cfs))
set(gca,'yscale','log');shading interp;axis tight;
title('Scalogram');xlabel('Time (s)');ylabel('Frequency (Hz)')
```




Use the helper function `helperCreateRGBfromTF` to create the scalograms as RGB images and write them to the appropriate subdirectory in `dataDir`. The source code for this helper function is in the Supporting Functions section at the end of this example. To be compatible with the GoogLeNet architecture, each RGB image is an array of size 224-by-224-by-3.

```
helperCreateRGBfromTF(ECGData,parentDir,dataDir)
```

### Divide into Training and Validation Data

Load the scalogram images as an image datastore. The `imageDatastore` function automatically labels the images based on folder names and stores the data as an ImageDatastore object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a CNN.

```
allImages = imageDatastore(fullfile(parentDir,dataDir),...
    'IncludeSubfolders',true,...
    'LabelSource','foldernames');
```

Randomly divide the images into two groups, one for training and the other for validation. Use 80% of the images for training, and the remainder for validation. For purposes of reproducibility, we set the random seed to the default value.

```
rng default
[imgsTrain,imgsValidation] = splitEachLabel(allImages,0.8,'randomized');
disp(['Number of training images: ',num2str(numel(imgsTrain.Files))]);
```

```
Number of training images: 130
```

```
disp(['Number of validation images: ',num2str(numel(imgsValidation.Files))]);
```

```
Number of validation images: 32
```

### GoogLeNet

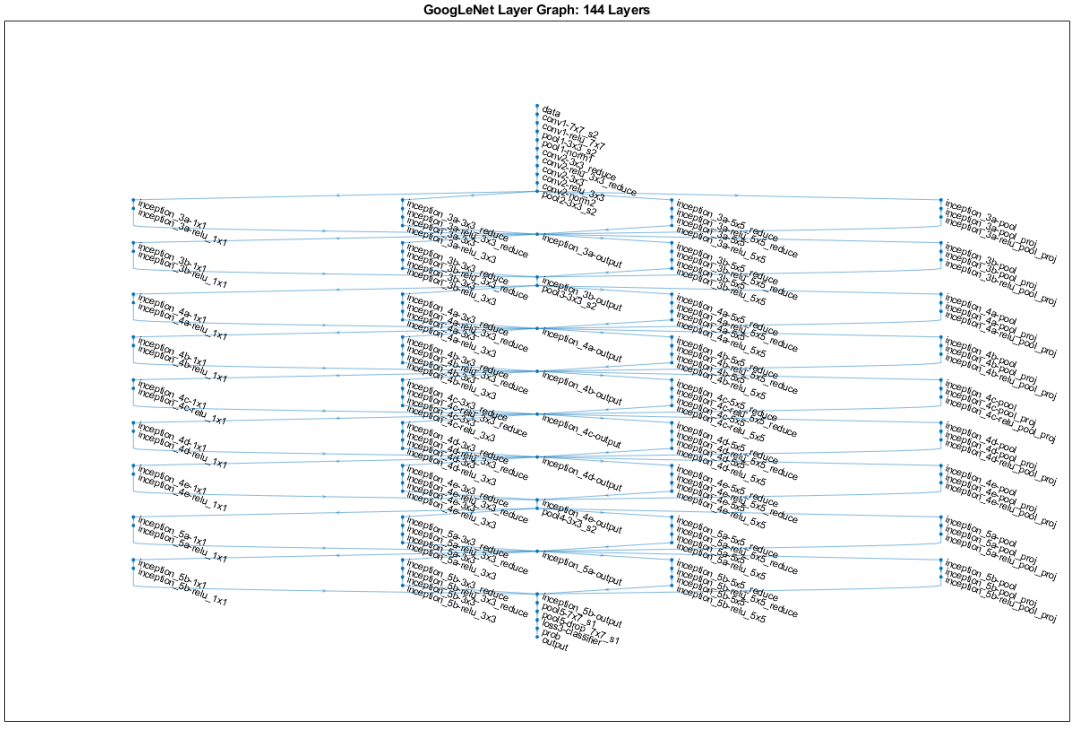
### Load

Load the pretrained GoogLeNet neural network. If Deep Learning Toolbox™ Model *for GoogLeNet Network* support package is not installed, the software provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**.

```
net = googlenet;
```

Extract and display the layer graph from the network.

```
lgraph = layerGraph(net);
numberOfLayers = numel(lgraph.Layers);
figure('Units','normalized','Position',[0.1 0.1 0.8 0.8]);
plot(lgraph)
title(['GoogLeNet Layer Graph: ',num2str(numberOfLayers),' Layers']);
```




Inspect the first element of the network Layers property. Confirm that GoogLeNet requires RGB images of size 224-by-224-by-3.

```
net.Layers(1)
```

```
ans =
  ImageInputLayer with properties:

                Name: 'data'
           InputSize: [224 224 3]

   Hyperparameters
    DataAugmentation: 'none'
       Normalization: 'zerocenter'
                Mean: [224×224×3 single]
```

### Modify GoogLeNet Network Parameters

Each layer in the network architecture can be considered a filter. The earlier layers identify more common features of images, such as blobs, edges, and colors. Subsequent layers focus on more specific features in order to differentiate categories. GoogLeNet is pretrained to classify images into 1000 object categories. You must retrain GoogLeNet for our ECG classification problem.

To prevent overfitting, a dropout layer is used. A dropout layer randomly sets input elements to zero with a given probability. See `dropoutLayer` (Deep Learning Toolbox) for more information. The default probability is 0.5. Replace the final dropout layer in the network, `'pool5-drop_7x7_s1'`, with a dropout layer of probability 0.6.

```
newDropoutLayer = dropoutLayer(0.6,'Name','new_Dropout');
lgraph = replaceLayer(lgraph,'pool5-drop_7x7_s1',newDropoutLayer);
```

The convolutional layers of the network extract image features that the last learnable layer and final classification layer use to classify the input image. These two layers, `'loss3-classifier'` and `'output'` in GoogLeNet, contain information on how to combine the features that the network extracts into class probabilities, a loss value, and predicted labels. To retrain GoogLeNet to classify the RGB images, replace these two layers with new layers adapted to the data.

Replace the fully connected layer `'loss3-classifier'` with a new fully connected layer with the number of filters equal to the number of classes. To learn faster in the new layers than in the transferred layers, increase the learning rate factors of the fully connected layer.

```
numClasses = numel(categories(imgsTrain.Labels));
newConnectedLayer = fullyConnectedLayer(numClasses,'Name','new_fc',...
    'WeightLearnRateFactor',5,'BiasLearnRateFactor',5);
lgraph = replaceLayer(lgraph,'loss3-classifier',newConnectedLayer);
```

The classification layer specifies the output classes of the network. Replace the classification layer with a new one without class labels. `trainNetwork` automatically sets the output classes of the layer at training time.

```
newClassLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'output',newClassLayer);
```

### Set Training Options and Train GoogLeNet

Training a neural network is an iterative process that involves minimizing a loss function. To minimize the loss function, a gradient descent algorithm is used. In each iteration, the gradient of the loss function is evaluated and the descent algorithm weights are updated.

Training can be tuned by setting various options. `InitialLearnRate` specifies the initial step size in the direction of the negative gradient of the loss function. `MiniBatchSize` specifies how large of a

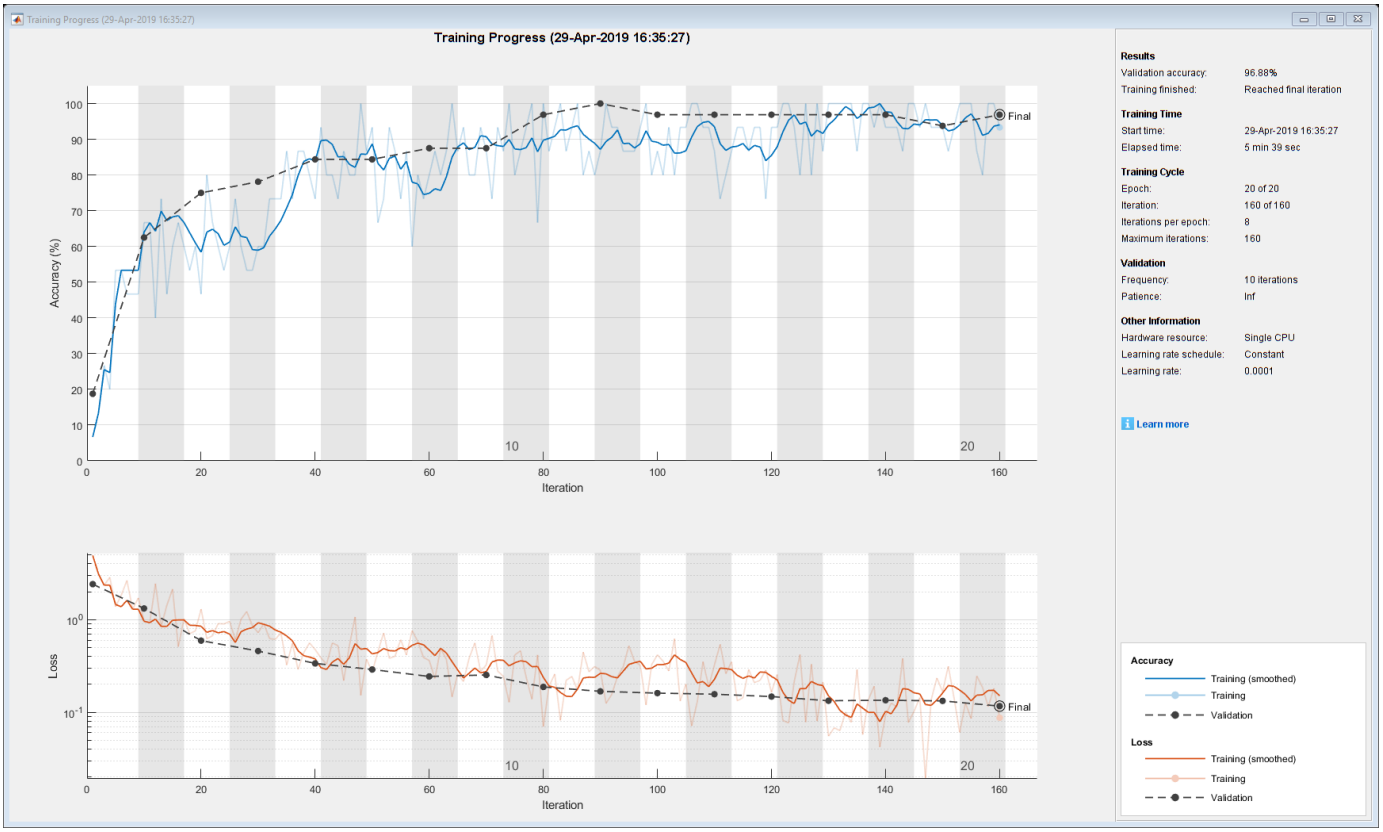
subset of the training set to use in each iteration. One epoch is a full pass of the training algorithm over the entire training set. `MaxEpochs` specifies the maximum number of epochs to use for training. Choosing the right number of epochs is not a trivial task. Decreasing the number of epochs has the effect of underfitting the model, and increasing the number of epochs results in overfitting.

Use the `trainingOptions` (Deep Learning Toolbox) function to specify the training options. Set `MiniBatchSize` to 10, `MaxEpochs` to 10, and `InitialLearnRate` to 0.0001. Visualize training progress by setting `Plots` to `training-progress`. Use the stochastic gradient descent with momentum optimizer. By default, training is done on a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). For purposes of reproducibility, set `ExecutionEnvironment` to `cpu` so that `trainNetwork` used the CPU. Set the random seed to the default value. Run times will be faster if you are able to use a GPU.

```
options = trainingOptions('sgdm',...
    'MiniBatchSize',15,...
    'MaxEpochs',20,...
    'InitialLearnRate',1e-4,...
    'ValidationData',imgsValidation,...
    'ValidationFrequency',10,...
    'Verbose',1,...
    'ExecutionEnvironment','cpu',...
    'Plots','training-progress');
rng default
```

Train the network. The training process usually takes 1-5 minutes on a desktop CPU. The command window displays training information during the run. Results include epoch number, iteration number, time elapsed, mini-batch accuracy, validation accuracy, and loss function value for the validation data.

```
trainedGN = trainNetwork(imgsTrain,lgraph,options);
```




```
Initializing input data normalization.
|=========================================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validati       |
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss        |
|=========================================================================================================|
|       1 |           1 |       00:00:03 |        6.67% |       18.75% |       4.9207 |          2.4    |
|       2 |          10 |       00:00:23 |       66.67% |       62.50% |       0.9589 |          1.3    |
|       3 |          20 |       00:00:43 |       46.67% |       75.00% |       1.2973 |          0.5    |
|       4 |          30 |       00:01:04 |       60.00% |       78.13% |       0.7219 |          0.4    |
|       5 |          40 |       00:01:25 |       73.33% |       84.38% |       0.4750 |          0.3    |
|       7 |          50 |       00:01:46 |       93.33% |       84.38% |       0.2714 |          0.2    |
|       8 |          60 |       00:02:07 |       80.00% |       87.50% |       0.3617 |          0.2    |
|       9 |          70 |       00:02:29 |       86.67% |       87.50% |       0.3246 |          0.2    |
|      10 |          80 |       00:02:50 |      100.00% |       96.88% |       0.0701 |          0.1    |
|      12 |          90 |       00:03:11 |       86.67% |      100.00% |       0.2836 |          0.1    |
|      13 |         100 |       00:03:32 |       86.67% |       96.88% |       0.4160 |          0.1    |
|      14 |         110 |       00:03:53 |       86.67% |       96.88% |       0.3237 |          0.1    |
|      15 |         120 |       00:04:14 |       93.33% |       96.88% |       0.1646 |          0.1    |
|      17 |         130 |       00:04:35 |      100.00% |       96.88% |       0.0551 |          0.1    |
|      18 |         140 |       00:04:57 |       93.33% |       96.88% |       0.0927 |          0.1    |
|      19 |         150 |       00:05:18 |       93.33% |       93.75% |       0.1666 |          0.1    |
|      20 |         160 |       00:05:39 |       93.33% |       96.88% |       0.0873 |          0.1    |
|=========================================================================================================|
```

Inspect the last layer of the trained network. Confirm the Classification Output layer includes the three classes.

```
trainedGN.Layers(end)
```

```
ans =
  ClassificationOutputLayer with properties:

           Name: 'new_classoutput'
        Classes: [ARR    CHF    NSR]
     OutputSize: 3

   Hyperparameters
    LossFunction: 'crossentropyex'
```

### Evaluate GoogLeNet Accuracy

Evaluate the network using the validation data.

```
[YPred,probs] = classify(trainedGN,imgsValidation);
accuracy = mean(YPred==imgsValidation.Labels);
disp(['GoogLeNet Accuracy: ',num2str(100*accuracy),'%'])
```

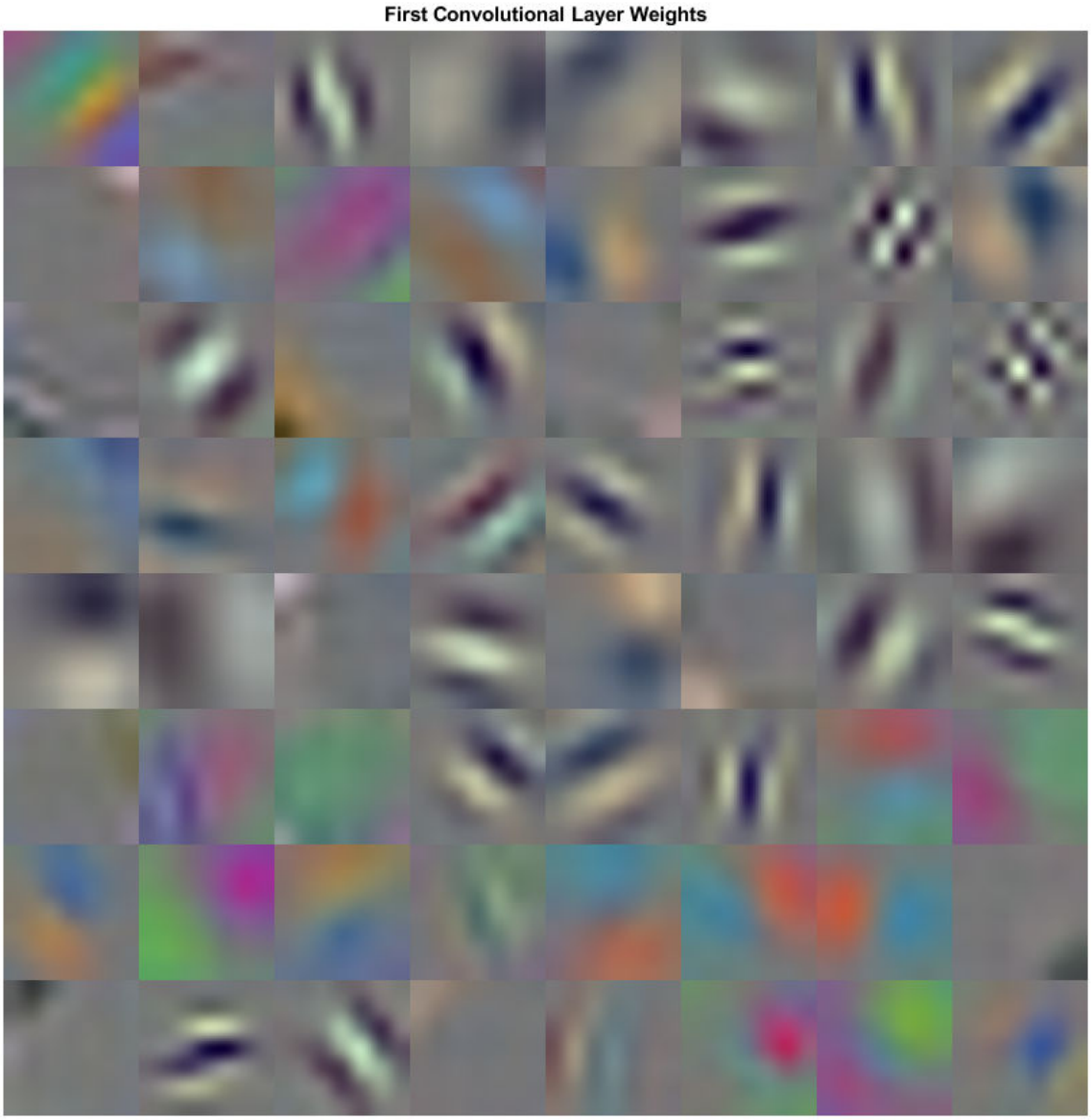
```
GoogLeNet Accuracy: 96.875%
```

The accuracy is identical to the validation accuracy reported on the training visualization figure. The scalograms were split into training and validation collections. Both collections were used to train GoogLeNet. The ideal way to evaluate the result of the training is to have the network classify data it has not seen. Since there is an insufficient amount of data to divide into training, validation, and testing, we treat the computed validation accuracy as the network accuracy.

### Explore GoogLeNet Activations

Each layer of a CNN produces a response, or activation, to an input image. However, there are only a few layers within a CNN that are suitable for image feature extraction. The layers at the beginning of the network capture basic image features, such as edges and blobs. To see this, visualize the network filter weights from the first convolutional layer. There are 64 individual sets of weights in the first layer.

```
wghts = trainedGN.Layers(2).Weights;
wghts = rescale(wghts);
wghts = imresize(wghts,5);
figure
montage(wghts)
title('First Convolutional Layer Weights')
```

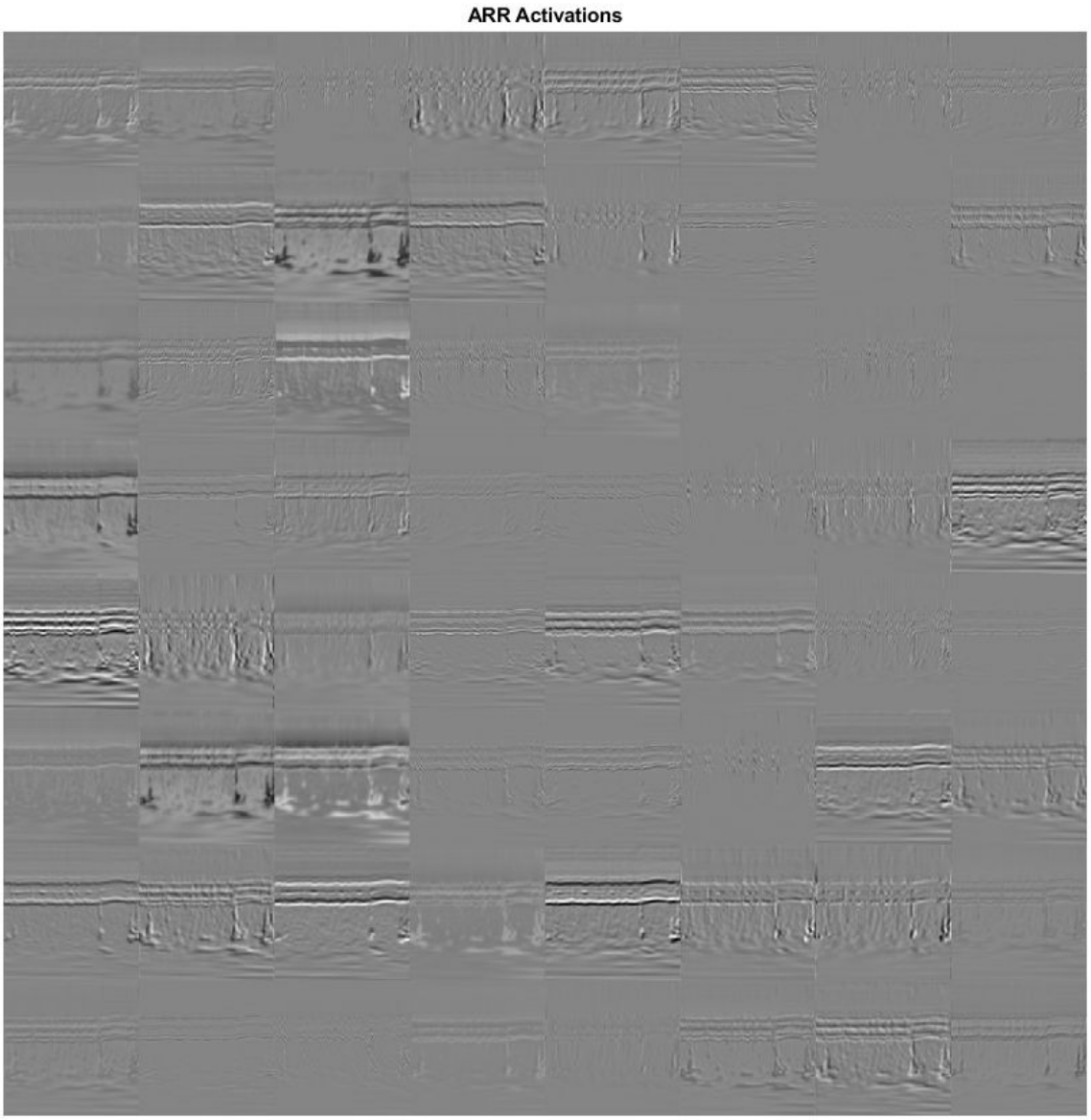



You can examine the activations and discover which features GoogLeNet learns by comparing areas of activation with the original image. For more information, see “Visualize Activations of a Convolutional Neural Network” (Deep Learning Toolbox) and “Visualize Features of a Convolutional Neural Network” (Deep Learning Toolbox).

Examine which areas in the convolutional layers activate on an image from the ARR class. Compare with the corresponding areas in the original image. Each layer of a convolutional neural network consists of many 2-D arrays called *channels*. Pass the image through the network and examine the output activations of the first convolutional layer, `'conv1-7x7_s2'`.

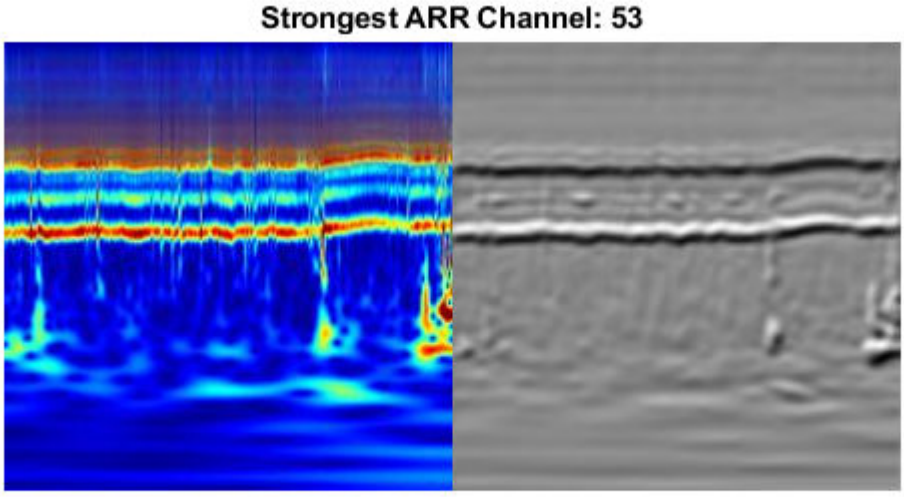
```
convLayer = 'conv1-7x7_s2';
```

```
imgClass = 'ARR';
imgName = 'ARR_10.jpg';
imarr = imread(fullfile(parentDir,dataDir,imgClass,imgName));

trainingFeaturesARR = activations(trainedGN,imarr,convLayer);
sz = size(trainingFeaturesARR);
trainingFeaturesARR = reshape(trainingFeaturesARR,[sz(1) sz(2) 1 sz(3)]);
figure
montage(rescale(trainingFeaturesARR),'Size',[8 8])
title([imgClass,' Activations'])
```




Find the strongest channel for this image. Compare the strongest channel with the original image.

```
imgSize = size(imarr);
imgSize = imgSize(1:2);
[~,maxValueIndex] = max(max(max(trainingFeaturesARR)));
arrMax = trainingFeaturesARR(:,:,:,maxValueIndex);
arrMax = rescale(arrMax);
arrMax = imresize(arrMax,imgSize);
figure;
imshowpair(imarr,arrMax,'montage')
title(['Strongest ',imgClass,' Channel: ',num2str(maxValueIndex)])
```




### SqueezeNet

SqueezeNet is a deep CNN whose architecture supports images of size 227-by-227-by-3. Even though the image dimensions are different for GoogLeNet, you do not have to generate new RGB images at the SqueezeNet dimensions. You can use the original RGB images.

#### Load

Load the pretrained SqueezeNet neural network. If Deep Learning Toolbox™ Model *for SqueezeNet Network* support package is not installed, the software provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**.

```
sqz = squeezenet;
```

Extract the layer graph from the network. Confirm SqueezeNet has fewer layers than GoogLeNet. Also confirm that SqueezeNet is configured for images of size 227-by-227-by-3

```
lgraphSqz = layerGraph(sqz);
disp(['Number of Layers: ',num2str(numel(lgraphSqz.Layers))])
```

```
Number of Layers: 68
```

```
disp(lgraphSqz.Layers(1).InputSize)
```

```
   227   227     3
```

### Modify SqueezeNet Network Parameters

To retrain SqueezeNet to classify new images, make changes similar to those made for GoogLeNet.

Inspect the last six network layers.

```
lgraphSqz.Layers(end-5:end)
```

```
ans =
  6x1 Layer array with layers:

     1   'drop9'                              Dropout                 50% dropout
     2   'conv10'                             Convolution             1000 1x1x512 convolutions wi
     3   'relu_conv10'                        ReLU                    ReLU
     4   'pool10'                             Average Pooling         14x14 average pooling with s
     5   'prob'                               Softmax                 softmax
     6   'ClassificationLayer_predictions'    Classification Output   crossentropyex with 'tench'
```

Replace the `'drop9'` layer, the last dropout layer in the network, with a dropout layer of probability 0.6.

```
tmpLayer = lgraphSqz.Layers(end-5);
newDropoutLayer = dropoutLayer(0.6,'Name','new_dropout');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newDropoutLayer);
```

Unlike GoogLeNet, the last learnable layer in SqueezeNet is a 1-by-1 convolutional layer, `'conv10'`, and not a fully connected layer. Replace the `'conv10'` layer with a new convolutional layer with the number of filters equal to the number of classes. As was done with GoogLeNet, increase the learning rate factors of the new layer.

```
numClasses = numel(categories(imgsTrain.Labels));
tmpLayer = lgraphSqz.Layers(end-4);
newLearnableLayer = convolution2dLayer(1,numClasses, ...
        'Name','new_conv', ...
        'WeightLearnRateFactor',10, ...
        'BiasLearnRateFactor',10);
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newLearnableLayer);
```

Replace the classification layer with a new one without class labels.

```
tmpLayer = lgraphSqz.Layers(end);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newClassLayer);
```

Inspect the last six layers of the network. Confirm the dropout, convolutional, and output layers have been changed.

```
lgraphSqz.Layers(63:68)
```

```
ans =
  6x1 Layer array with layers:

     1   'new_dropout'       Dropout                 60% dropout
     2   'new_conv'          Convolution             3 1x1 convolutions with stride [1  1] and pa
     3   'relu_conv10'       ReLU                    ReLU
     4   'pool10'            Average Pooling         14x14 average pooling with stride [1  1] and
     5   'prob'              Softmax                 softmax
     6   'new_classoutput'   Classification Output   crossentropyex
```

### Prepare RGB Data for SqueezeNet

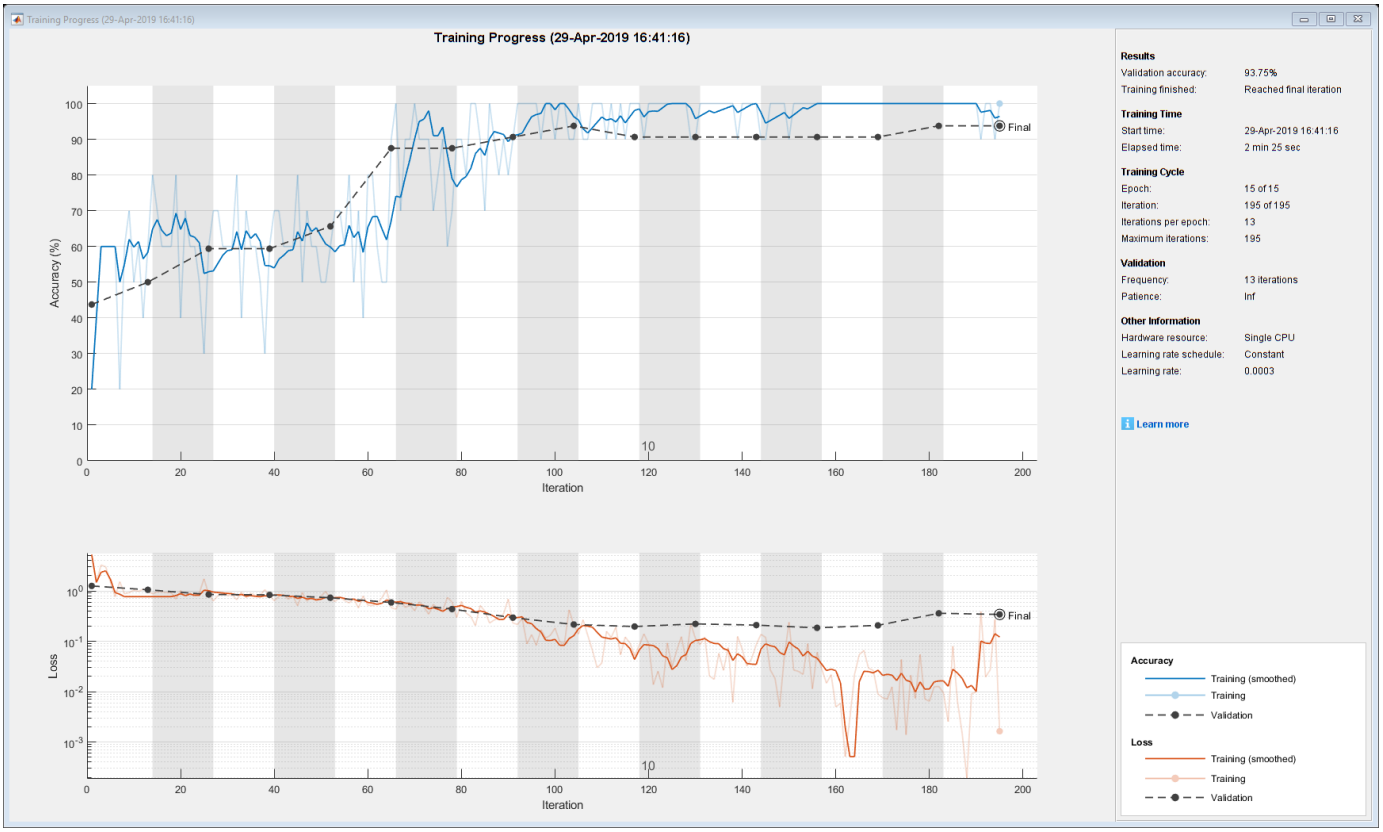
The RGB images have dimensions appropriate for the GoogLeNet architecture. Create augmented image datastores that automatically resize the existing RGB images for the SqueezeNet architecture. For more information, see `augmentedImageDatastore` (Deep Learning Toolbox).

```
augimgsTrain = augmentedImageDatastore([227 227],imgsTrain);
augimgsValidation = augmentedImageDatastore([227 227],imgsValidation);
```

### Set Training Options and Train SqueezeNet

Create a new set of training options to use with SqueezeNet. Set the random seed to the default value and train the network. The training process usually takes 1-5 minutes on a desktop CPU.

```
ilr = 3e-4;
miniBatchSize = 10;
maxEpochs = 15;
valFreq = floor(numel(augimgsTrain.Files)/miniBatchSize);
opts = trainingOptions('sgdm',...
    'MiniBatchSize',miniBatchSize,...
    'MaxEpochs',maxEpochs,...
    'InitialLearnRate',ilr,...
    'ValidationData',augimgsValidation,...
    'ValidationFrequency',valFreq,...
    'Verbose',1,...
    'ExecutionEnvironment','cpu',...
    'Plots','training-progress');

rng default
trainedSN = trainNetwork(augimgsTrain,lgraphSqz,opts);
```




```
Initializing input data normalization.
|=========================================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validati       |
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss        |
|=========================================================================================================|
|       1 |           1 |       00:00:01 |       20.00% |       43.75% |       5.2508 |       1.2       |
|       1 |          13 |       00:00:11 |       60.00% |       50.00% |       0.9912 |       1.0       |
|       2 |          26 |       00:00:20 |       60.00% |       59.38% |       0.8554 |       0.8       |
|       3 |          39 |       00:00:30 |       60.00% |       59.38% |       0.8120 |       0.8       |
|       4 |          50 |       00:00:38 |       50.00% |              |       0.7885 |                 |
|       4 |          52 |       00:00:40 |       60.00% |       65.63% |       0.7091 |       0.7       |
|       5 |          65 |       00:00:49 |       90.00% |       87.50% |       0.4639 |       0.5       |
|       6 |          78 |       00:00:59 |       70.00% |       87.50% |       0.6021 |       0.4       |
|       7 |          91 |       00:01:08 |       90.00% |       90.63% |       0.2307 |       0.2       |
|       8 |         100 |       00:01:15 |       90.00% |              |       0.1827 |                 |
|       8 |         104 |       00:01:18 |       90.00% |       93.75% |       0.2139 |       0.2       |
|       9 |         117 |       00:01:28 |      100.00% |       90.63% |       0.0521 |       0.1       |
|      10 |         130 |       00:01:38 |       90.00% |       90.63% |       0.1134 |       0.2       |
|      11 |         143 |       00:01:47 |      100.00% |       90.63% |       0.0855 |       0.2       |
|      12 |         150 |       00:01:52 |       90.00% |              |       0.2394 |                 |
|      12 |         156 |       00:01:57 |      100.00% |       90.63% |       0.0606 |       0.1       |
|      13 |         169 |       00:02:06 |      100.00% |       90.63% |       0.0090 |       0.2       |
|      14 |         182 |       00:02:16 |      100.00% |       93.75% |       0.0127 |       0.1       |
|      15 |         195 |       00:02:25 |      100.00% |       93.75% |       0.0016 |       0.1       |
|=========================================================================================================|
```

Inspect the last layer of the network. Confirm the Classification Output layer includes the three classes.

```
trainedSN.Layers(end)

ans =
  ClassificationOutputLayer with properties:

            Name: 'new_classoutput'
         Classes: [ARR    CHF    NSR]
      OutputSize: 3

   Hyperparameters
    LossFunction: 'crossentropyex'
```

### Evaluate SqueezeNet Accuracy

Evaluate the network using the validation data.

```
[YPred,probs] = classify(trainedSN,augimgsValidation);
accuracy = mean(YPred==imgsValidation.Labels);
disp(['SqueezeNet Accuracy: ',num2str(100*accuracy),'%'])
```

```
SqueezeNet Accuracy: 93.75%
```

### Conclusion

This example shows how to use transfer learning and continuous wavelet analysis to classify three classes of ECG signals by leveraging the pretrained CNNs GoogLeNet and SqueezeNet. Wavelet-based time-frequency representations of ECG signals are used to create scalograms. RGB images of the scalograms are generated. The images are used to fine-tune both deep CNNs. Activations of different network layers were also explored.

This example illustrates one possible workflow you can use for classifying signals using pretrained CNN models. Other workflows are possible. “Deploy Signal Classifier on NVIDIA Jetson Using Wavelet Analysis and Deep Learning” (Wavelet Toolbox) and “Deploy Signal Classifier Using Wavelets and Deep Learning on Raspberry Pi” (Wavelet Toolbox) show how to deploy code onto hardware for signal classification. GoogLeNet and SqueezeNet are models pretrained on a subset of the ImageNet database [10], which is used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [8]. The ImageNet collection contains images of real-world objects such as fish, birds, appliances, and fungi. Scalograms fall outside the class of real-world objects. In order to fit into the GoogLeNet and SqueezeNet architecture, the scalograms also underwent data reduction. Instead of fine-tuning pretrained CNNs to distinguish different classes of scalograms, training a CNN from scratch at the original scalogram dimensions is an option.

### References


1. Baim, D. S., W. S. Colucci, E. S. Monrad, H. S. Smith, R. F. Wright, A. Lanoue, D. F. Gauthier, B. J. Ransil, W. Grossman, and E. Braunwald. "Survival of patients with severe congestive heart failure treated with oral milrinone." *Journal of the American College of Cardiology*. Vol. 7, Number 3, 1986, pp. 661–670.
2. Engin, M. "ECG beat classification using neuro-fuzzy network." *Pattern Recognition Letters*. Vol. 25, Number 15, 2004, pp.1715–1722.
3. Goldberger A. L., L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. "PhysioBank, PhysioToolkit,and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." Circulation. Vol. 101, Number 23: e215–e220. [Circulation Electronic Pages; `http://circ.ahajournals.org/content/101/23/e215.full`]; 2000 (June 13). doi: 10.1161/01.CIR.101.23.e215.

4 Leonarduzzi, R. F., G. Schlotthauer, and M. E. Torres. "Wavelet leader based multifractal analysis of heart rate variability during myocardial ischaemia." In *Engineering in Medicine and Biology Society (EMBC), Annual International Conference of the IEEE*, 110–113. Buenos Aires, Argentina: IEEE, 2010.

5 Li, T., and M. Zhou. "ECG classification using wavelet packet entropy and random forests." *Entropy*. Vol. 18, Number 8, 2016, p.285.

6 Maharaj, E. A., and A. M. Alonso. "Discriminant analysis of multivariate time series: Application to diagnosis based on ECG signals." *Computational Statistics and Data Analysis*. Vol. 70, 2014, pp. 67–87.

7 Moody, G. B., and R. G. Mark. "The impact of the MIT-BIH Arrhythmia Database." *IEEE Engineering in Medicine and Biology Magazine*. Vol. 20. Number 3, May-June 2001, pp. 45–50. (PMID: 11446209)

8 Russakovsky, O., J. Deng, and H. Su et al. "ImageNet Large Scale Visual Recognition Challenge." *International Journal of Computer Vision*. Vol. 115, Number 3, 2015, pp. 211–252.

9 Zhao, Q., and L. Zhang. "ECG feature extraction and classification using wavelet transform and support vector machines." In *IEEE International Conference on Neural Networks and Brain*, 1089–1092. Beijing, China: IEEE, 2005.

10 *ImageNet*. `http://www.image-net.org`

### Supporting Functions

**helperCreateECGDataDirectories** creates a data directory inside a parent directory, then creates three subdirectories inside the data directory. The subdirectories are named after each class of ECG signal found in `ECGData`.

```
function helperCreateECGDirectories(ECGData,parentFolder,dataFolder)
% This function is only intended to support the ECGAndDeepLearningExample.
% It may change or be removed in a future release.

rootFolder = parentFolder;
localFolder = dataFolder;
mkdir(fullfile(rootFolder,localFolder))

folderLabels = unique(ECGData.Labels);
for i = 1:numel(folderLabels)
    mkdir(fullfile(rootFolder,localFolder,char(folderLabels(i))));
end
end
```

**helperPlotReps** plots the first thousand samples of a representative of each class of ECG signal found in `ECGData`.

```
function helperPlotReps(ECGData)
% This function is only intended to support the ECGAndDeepLearningExample.
% It may change or be removed in a future release.

folderLabels = unique(ECGData.Labels);

for k=1:3
    ecgType = folderLabels{k};
    ind = find(ismember(ECGData.Labels,ecgType));
    subplot(3,1,k)
    plot(ECGData.Data(ind(1),1:1000));
    grid on
```

```
    title(ecgType)
end
end
```

**helperCreateRGBfromTF** uses `cwtfilterbank` (Wavelet Toolbox) to obtain the continuous wavelet transform of the ECG signals and generates the scalograms from the wavelet coefficients. The helper function resizes the scalograms and writes them to disk as jpeg images.

```
function helperCreateRGBfromTF(ECGData,parentFolder,childFolder)
% This function is only intended to support the ECGAndDeepLearningExample.
% It may change or be removed in a future release.

imageRoot = fullfile(parentFolder,childFolder);

data = ECGData.Data;
labels = ECGData.Labels;

[~,signalLength] = size(data);

fb = cwtfilterbank('SignalLength',signalLength,'VoicesPerOctave',12);
r = size(data,1);

for ii = 1:r
    cfs = abs(fb.wt(data(ii,:)));
    im = ind2rgb(im2uint8(rescale(cfs)),jet(128));

    imgLoc = fullfile(imageRoot,char(labels(ii)));
    imFileName = strcat(char(labels(ii)),'_',num2str(ii),'.jpg');
    imwrite(imresize(im,[224 224]),fullfile(imgLoc,imFileName));
end
end
```

# Denoise Speech Using Deep Learning Networks

This example shows how to denoise speech signals using deep learning networks. The example compares two types of networks applied to the same task: fully connected, and convolutional.

### Introduction

The aim of speech denoising is to remove noise from speech signals while enhancing the quality and intelligibility of speech. This example showcases the removal of washing machine noise from speech signals using deep learning networks. The example compares two types of networks applied to the same task: fully connected, and convolutional.

### Problem Summary

Consider the following speech signal sampled at 8 kHz.

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

Add washing machine noise to the speech signal. Set the noise power such that the signal-to-noise ratio (SNR) is zero dB.

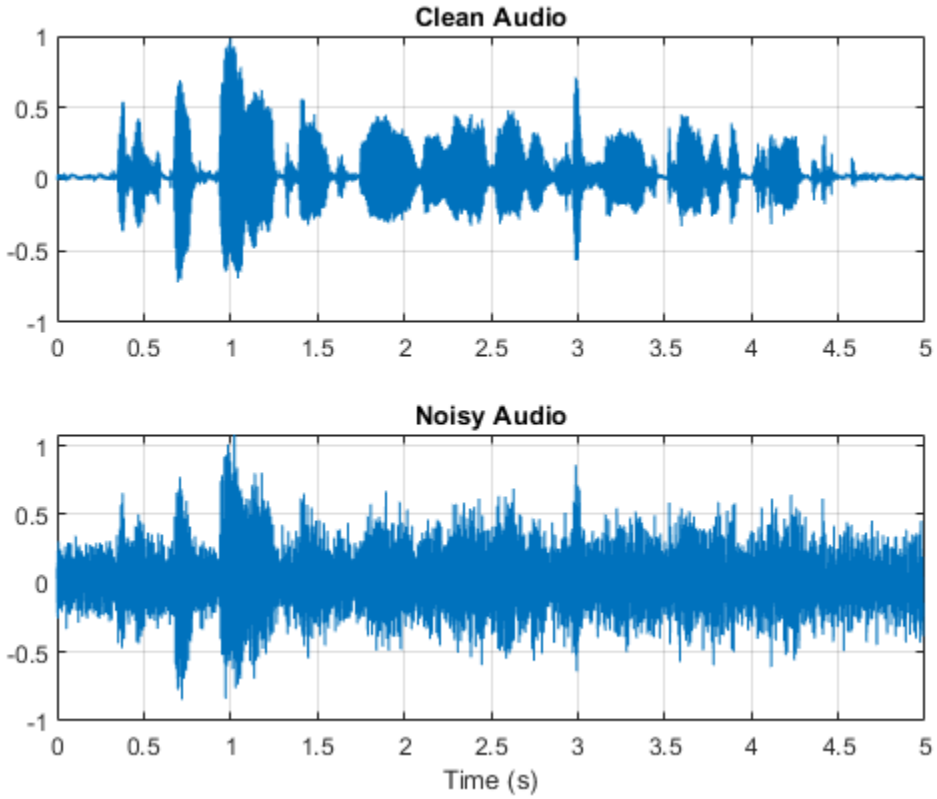
```
noise = audioread("WashingMachine-16-8-mono-1000secs.mp3");

% Extract a noise segment from a random location in the noise file
ind = randi(numel(noise) - numel(cleanAudio) + 1, 1, 1);
noiseSegment = noise(ind:ind + numel(cleanAudio) - 1);

speechPower = sum(cleanAudio.^2);
noisePower = sum(noiseSegment.^2);
noisyAudio = cleanAudio + sqrt(speechPower/noisePower) * noiseSegment;
```

Listen to the noisy speech signal.

```
sound(noisyAudio,fs)
```

Visualize the original and noisy signals.

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

subplot(2,1,1)
plot(t,cleanAudio)
title("Clean Audio")
grid on

subplot(2,1,2)
plot(t,noisyAudio)
title("Noisy Audio")
xlabel("Time (s)")
grid on
```




The objective of speech denoising is to remove the washing machine noise from the speech signal while minimizing undesired artifacts in the output speech.

### Examine the Dataset

This example uses a subset of the Mozilla Common Voice dataset [1 on page 24-0 ] to train and test the deep learning networks. The data set contains 48 kHz recordings of subjects speaking short sentences. Download the data set and unzip the downloaded file.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/commonvoice.zip';
downloadFolder = tempdir;
dataFolder = fullfile(downloadFolder,'commonvoice');

if ~exist(dataFolder,'dir')
    disp('Downloading data set (956 MB) ...')
    unzip(url,downloadFolder)
end
```

```
Downloading data set (956 MB) ...
```

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

```
adsTrain = audioDatastore(fullfile(dataFolder,'train'),'IncludeSubfolders',true);

reduceDataset = true;
if reduceDataset
```

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

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

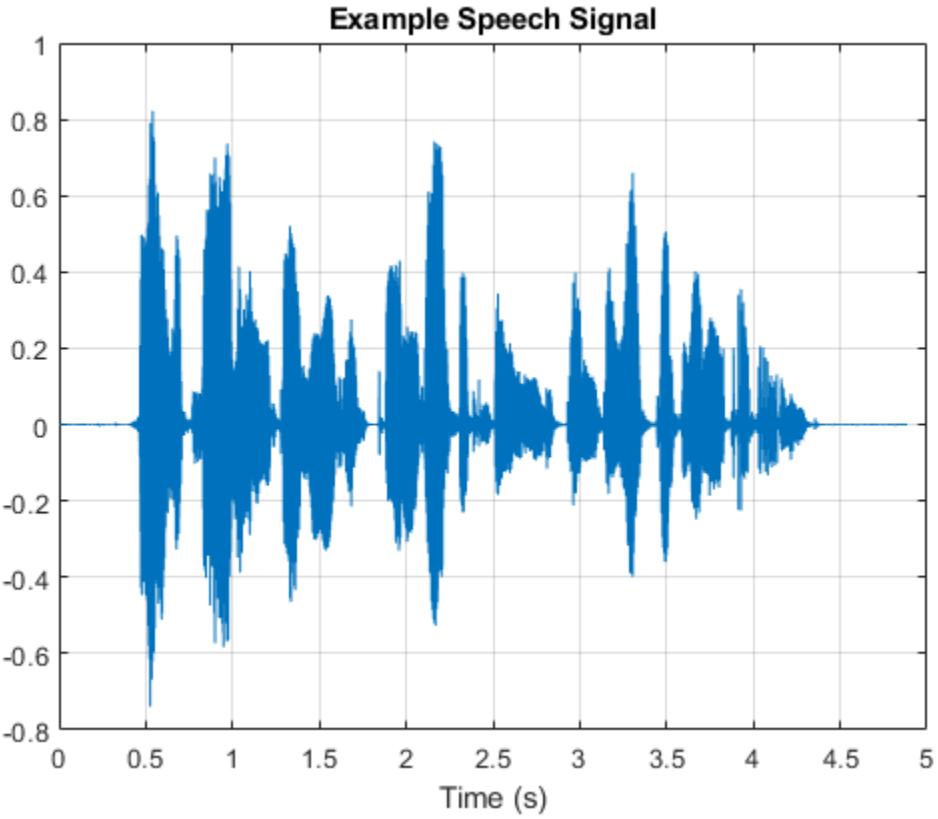
```
[audio,adsTrainInfo] = read(adsTrain);
```

Listen to the speech signal.

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

Plot the speech signal.

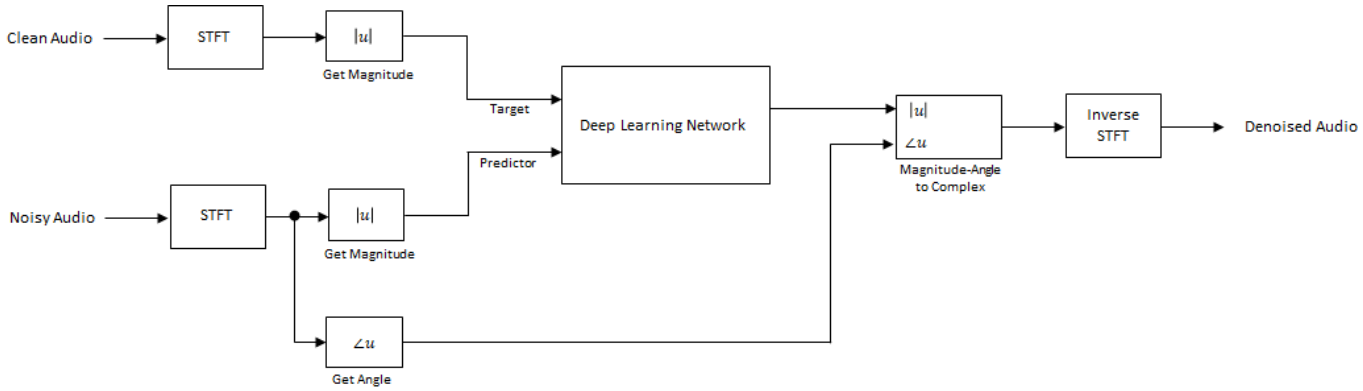
```
figure
t = (1/adsTrainInfo.SampleRate) * (0:numel(audio)-1);
plot(t,audio)
title("Example Speech Signal")
xlabel("Time (s)")
grid on
```



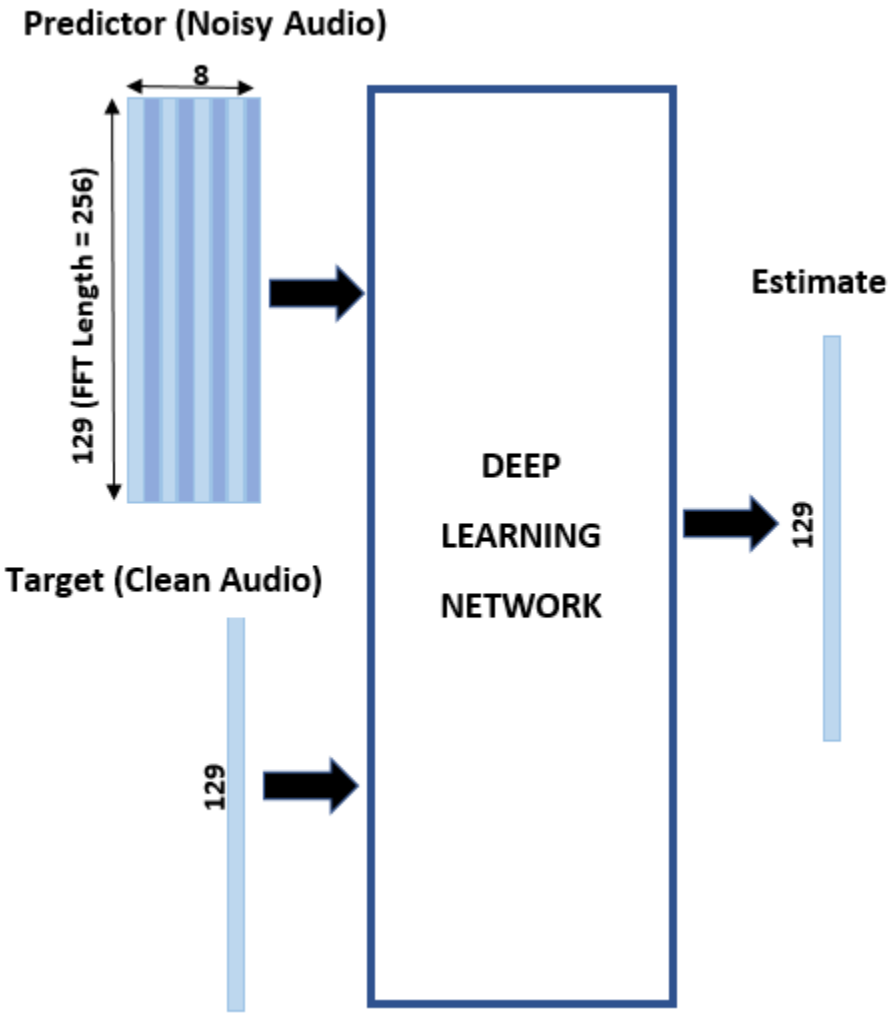

### Deep Learning System Overview

The basic deep learning training scheme is shown below. Note that, since speech generally falls below 4 kHz, you first downsample the clean and noisy audio signals to 8 kHz to reduce the computational load of the network. The predictor and target network signals are the magnitude spectra of the noisy and clean audio signals, respectively. The network's output is the magnitude spectrum of the denoised signal. The regression network uses the predictor input to minimize the

mean square error between its output and the input target. The denoised audio is converted back to the time domain using the output magnitude spectrum and the phase of the noisy signal [2 on page 24-0 ].




You transform the audio to the frequency domain using the Short-Time Fourier transform (STFT), with a window length of 256 samples, an overlap of 75%, and a Hamming window. You reduce the size of the spectral vector to 129 by dropping the frequency samples corresponding to negative frequencies (because the time-domain speech signal is real, this does not lead to any information loss). The predictor input consists of 8 consecutive noisy STFT vectors, so that each STFT output estimate is computed based on the current noisy STFT and the 7 previous noisy STFT vectors.




### STFT Targets and Predictors

This section illustrates how to generate the target and predictor signals from one training file.

First, define system parameters:

```
windowLength = 256;
win = hamming(windowLength,"periodic");
overlap = round(0.75 * windowLength);
ffTLength = windowLength;
inputFs = 48e3;
fs = 8e3;
numFeatures = ffTLength/2 + 1;
numSegments = 8;
```

Create a `dsp.SampleRateConverter` (DSP System Toolbox) 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 24-0 section. The `cellfun` command applies `HelperGenerateSpeechDenoisingFeatures` to the contents of each audio file in the datastore.

```
[targets,predictors] = cellfun(@(x)HelperGenerateSpeechDenoisingFeatures(x,noise,src),T,"UniformO
```

Use `gather` to evaluate the targets and predictors.

```
[targets,predictors] = gather(targets,predictors);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 49 sec
Evaluation completed in 2 min 11 sec
```

It is good practice to normalize all features to zero mean and unity standard deviation.

Compute the mean and standard deviation of the predictors and targets, respectively, and use them to normalize the data.

```
predictors = cat(3,predictors{:});
noisyMean = mean(predictors(:));
noisyStd = std(predictors(:));
predictors(:) = (predictors(:) - noisyMean)/noisyStd;

targets = cat(2,targets{:});
cleanMean = mean(targets(:));
cleanStd = std(targets(:));
targets(:) = (targets(:) - cleanMean)/cleanStd;
```

Reshape predictors and targets to the dimensions expected by the deep learning networks.

```
predictors = reshape(predictors,size(predictors,1),size(predictors,2),1,size(predictors,3));
targets = reshape(targets,1,1,size(targets,1),size(targets,2));
```

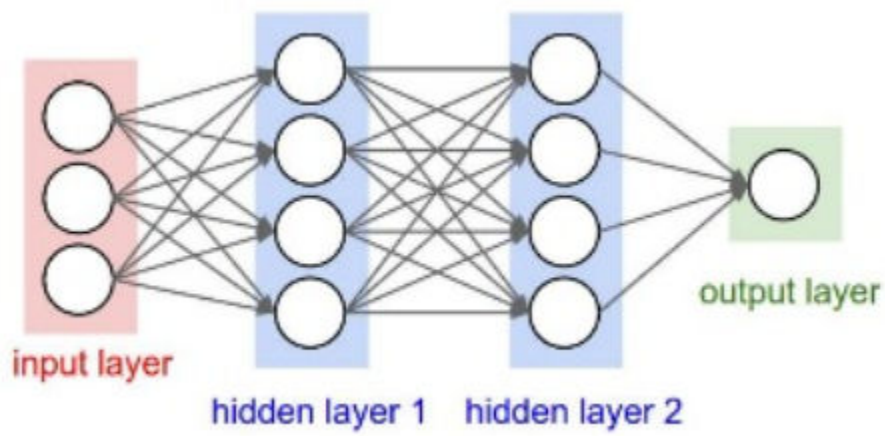
You will use 1% of the data for validation during training. Validation is useful to detect scenarios where the network is overfitting the training data.

Randomly split the data into training and validation sets.

```
inds = randperm(size(predictors,4));
L = round(0.99 * size(predictors,4));

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

validatePredictors = predictors(:,:,:,inds(L+1:end));
validateTargets = targets(:,:,:,inds(L+1:end));
```

### Speech Denoising with Fully Connected Layers

You first consider a denoising network comprised of fully connected layers. Each neuron in a fully connected layer is connected to all activations from the previous layer. A fully connected layer multiplies the input by a weight matrix and then adds a bias vector. The dimensions of the weight matrix and bias vector are determined by the number of neurons in the layer and the number of activations from the previous layer.



Define the layers of the network. Specify the input size to be images of size `NumFeatures`-by-`NumSegments` (129-by-8 in this example). Define two hidden fully connected layers, each with 1024 neurons. Since purely linear systems, follow each hidden fully connected layer with a Rectified Linear Unit (ReLU) layer. The batch normalization layers normalize the means and standard deviations of the outputs. Add a fully connected layer with 129 neurons, followed by a regression layer.

```
layers = [
    imageInputLayer([numFeatures,numSegments])
    fullyConnectedLayer(1024)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(1024)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(numFeatures)
    regressionLayer
    ];
```

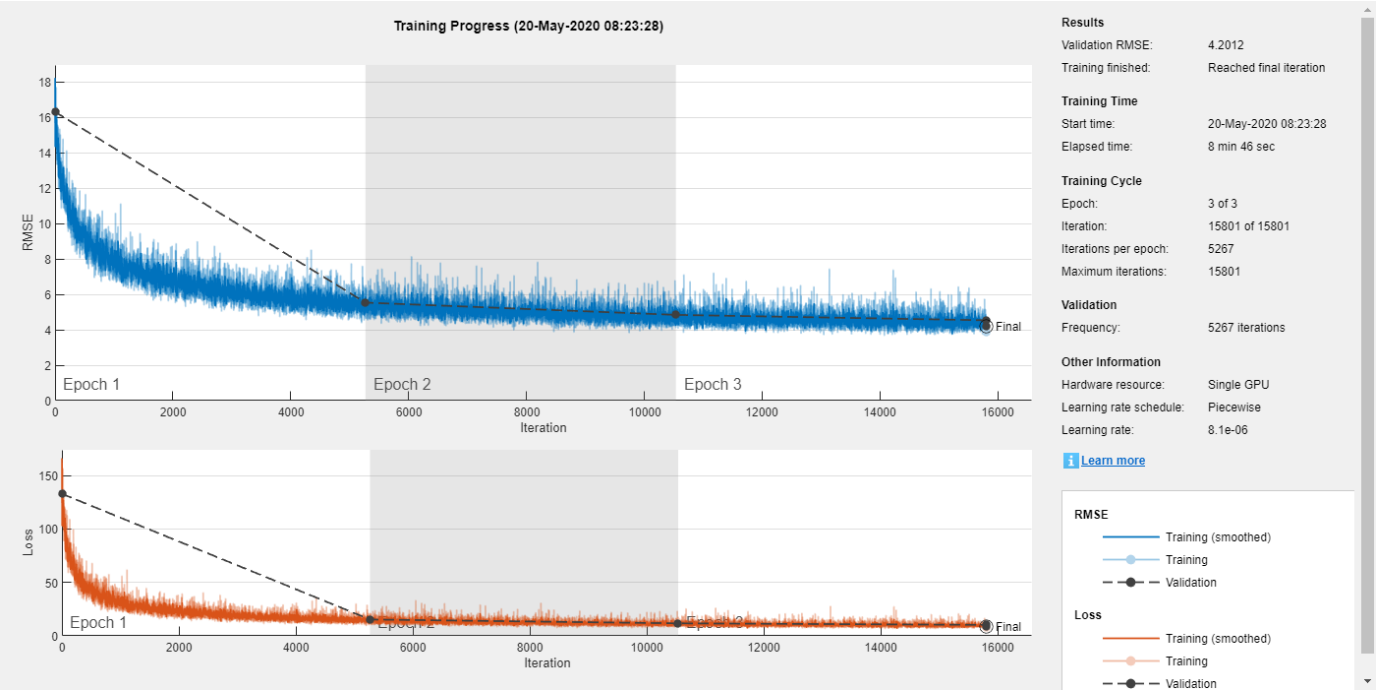
Next, specify the training options for the network. Set `MaxEpochs` to `3` so that the network makes 3 passes through the training data. Set `MiniBatchSize` of `128` so that the network looks at 128 training signals at a time. Specify `Plots` as `"training-progress"` to generate plots that show the training progress as the number of iterations increases. Set `Verbose` to `false` to disable printing the table output that corresponds to the data shown in the plot into the command line window. Specify `Shuffle` as `"every-epoch"` to shuffle the training sequences at the beginning of each epoch. Specify `LearnRateSchedule` to `"piecewise"` to decrease the learning rate by a specified factor (0.9) every time a certain number of epochs (1) has passed. Set `ValidationData` to the validation predictors and targets. Set `ValidationFrequency` such that the validation mean square error is computed once per epoch. This example uses the adaptive moment estimation (Adam) solver.

```
miniBatchSize = 128;
options = trainingOptions("adam", ...
    "MaxEpochs",3, ...
    "InitialLearnRate",1e-5,...
    "MiniBatchSize",miniBatchSize, ...
    "Shuffle","every-epoch", ...
    "Plots","training-progress", ...
    "Verbose",false, ...
    "ValidationFrequency",floor(size(trainPredictors,4)/miniBatchSize), ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.9, ...
    "LearnRateDropPeriod",1, ...
    "ValidationData",{validatePredictors,validateTargets});
```

Train the network with the specified training options and layer architecture using `trainNetwork`. Because the training set is large, the training process can take several minutes. To load a pre-trained network instead of training a network from scratch, set `doTraining` to `false`.

```
doTraining = true;
if doTraining
    denoiseNetFullyConnected = trainNetwork(trainPredictors,trainTargets,layers,options);
else
    s = load("denoisenet.mat");
    denoiseNetFullyConnected = s.denoiseNetFullyConnected;
    cleanMean = s.cleanMean;
    cleanStd = s.cleanStd;
    noisyMean = s.noisyMean;
```

```
    noisyStd = s.noisyStd;
end
```




Count the number of weights in the fully connected layers of the network.

```
numWeights = 0;
for index = 1:numel(denoiseNetFullyConnected.Layers)
    if isa(denoiseNetFullyConnected.Layers(index),"nnet.cnn.layer.FullyConnectedLayer")
        numWeights = numWeights + numel(denoiseNetFullyConnected.Layers(index).Weights);
    end
end
fprintf("The number of weights is %d.\n",numWeights);
```

```
The number of weights is 2237440.
```

### Speech Denoising with Convolutional Layers

Consider a network that uses convolutional layers instead of fully connected layers [3 on page 24-0 ]. A 2-D convolutional layer applies sliding filters to the input. The layer convolves the input by moving the filters along the input vertically and horizontally and computing the dot product of the weights and the input, and then adding a bias term. Convolutional layers typically consist of fewer parameters than fully connected layers.

Define the layers of the fully convolutional network described in [3 on page 24-0 ], comprising 16 convolutional layers. The first 15 convolutional layers are groups of 3 layers, repeated 5 times, with filter widths of 9, 5, and 9, and number of filters of 18, 30 and 8, respectively. The last convolutional layer has a filter width of 129 and 1 filter. In this network, convolutions are performed in only one direction (along the frequency dimension), and the filter width along the time dimension is set to 1 for all layers except the first one. Similar to the fully connected network, convolutional layers are followed by ReLu and batch normalization layers.

```
layers = [imageInputLayer([numFeatures,numSegments])
          convolution2dLayer([9 8],18,"Stride",[1 100],"Padding","same")
```

```
    batchNormalizationLayer
    reluLayer

    repmat( ...
    [convolution2dLayer([5 1],30,"Stride",[1 100],"Padding","same")
    batchNormalizationLayer
    reluLayer
    convolution2dLayer([9 1],8,"Stride",[1 100],"Padding","same")
    batchNormalizationLayer
    reluLayer
    convolution2dLayer([9 1],18,"Stride",[1 100],"Padding","same")
    batchNormalizationLayer
    reluLayer],4,1)

    convolution2dLayer([5 1],30,"Stride",[1 100],"Padding","same")
    batchNormalizationLayer
    reluLayer
    convolution2dLayer([9 1],8,"Stride",[1 100],"Padding","same")
    batchNormalizationLayer
    reluLayer

    convolution2dLayer([129 1],1,"Stride",[1 100],"Padding","same")

    regressionLayer
    ];
```

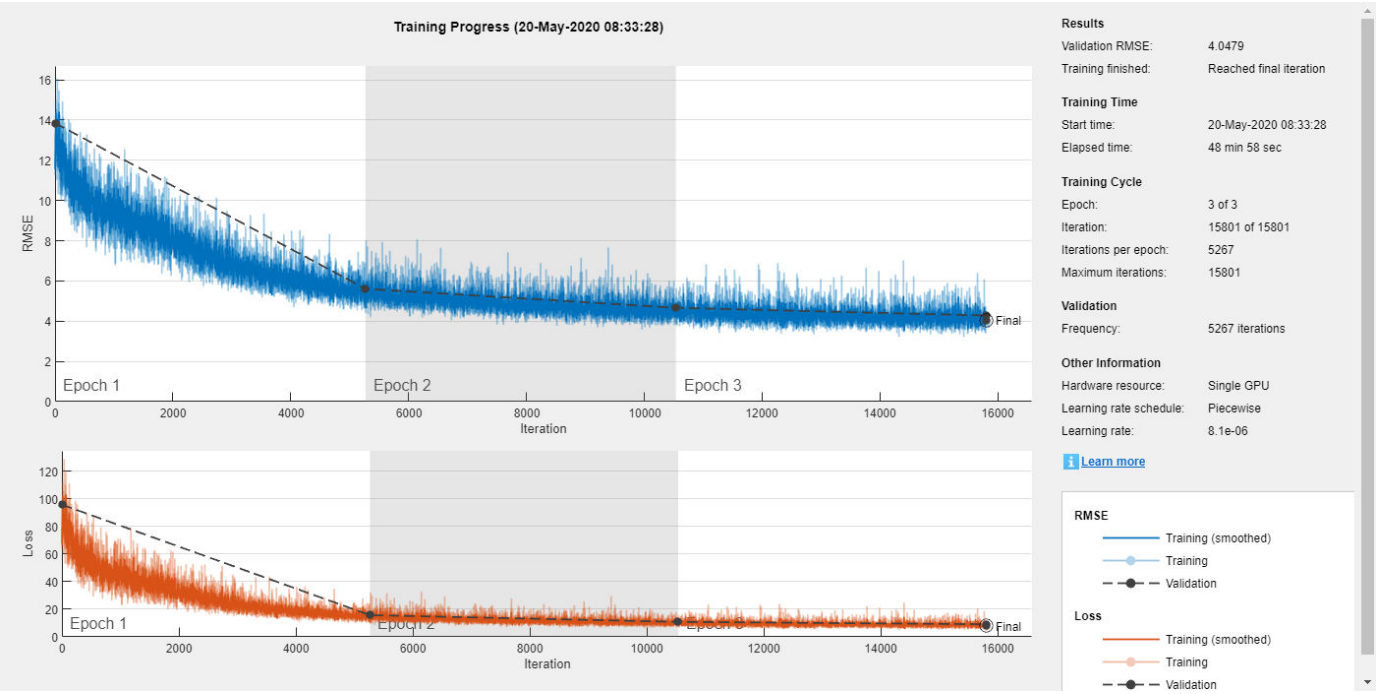
The training options are identical to the options for the fully connected network, except that the dimensions of the validation target signals are permuted to be consistent with the dimensions expected by the regression layer.

```
options = trainingOptions("adam", ...
    "MaxEpochs",3, ...
    "InitialLearnRate",1e-5, ...
    "MiniBatchSize",miniBatchSize, ...
    "Shuffle","every-epoch", ...
    "Plots","training-progress", ...
    "Verbose",false, ...
    "ValidationFrequency",floor(size(trainPredictors,4)/miniBatchSize), ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.9, ...
    "LearnRateDropPeriod",1, ...
    "ValidationData",{validatePredictors,permute(validateTargets,[3 1 2 4])});
```

Train the network with the specified training options and layer architecture using `trainNetwork`. Because the training set is large, the training process can take several minutes. To load a pre-trained network instead of training a network from scratch, set `doTraining` to `false`.

```
doTraining = true;
if doTraining
    denoiseNetFullyConvolutional = trainNetwork(trainPredictors,permute(trainTargets,[3 1 2 4]),
else
    s = load("denoisenet.mat");
    denoiseNetFullyConvolutional = s.denoiseNetFullyConvolutional;
    cleanMean = s.cleanMean;
    cleanStd = s.cleanStd;
    noisyMean = s.noisyMean;
```

```
    noisyStd = s.noisyStd;
end
```




Count the number of weights in the fully connected layers of the network.

```
numWeights = 0;
for index = 1:numel(denoiseNetFullyConvolutional.Layers)
    if isa(denoiseNetFullyConvolutional.Layers(index),"nnet.cnn.layer.Convolution2DLayer")
        numWeights = numWeights + numel(denoiseNetFullyConvolutional.Layers(index).Weights);
    end
end
fprintf("The number of weights in convolutional layers is %d\n",numWeights);
```

```
The number of weights in convolutional layers is 31812
```

### Test the Denoising Networks

Read in the test data set.

```
adsTest = audioDatastore(fullfile(dataFolder,'test'),'IncludeSubfolders',true);
```

Read the contents of a file from the datastore.

```
[cleanAudio,adsTestInfo] = read(adsTest);
```

Make sure the audio length is a multiple of the sample rate converter decimation factor.

```
L = floor(numel(cleanAudio)/decimationFactor);
cleanAudio = cleanAudio(1:decimationFactor*L);
```

Convert the audio signal to 8 kHz.

```
cleanAudio = src(cleanAudio);
reset(src)
```

In this testing stage, you corrupt speech with washing machine noise not used in the training stage.

```
noise = audioread("WashingMachine-16-8-mono-200secs.mp3");
```

Create a random noise segment from the washing machine noise vector.

```
randind = randi(numel(noise) - numel(cleanAudio), [1 1]);
noiseSegment = noise(randind : randind + numel(cleanAudio) - 1);
```

Add noise to the speech signal such that the SNR is 0 dB.

```
noisePower = sum(noiseSegment.^2);
cleanPower = sum(cleanAudio.^2);
noiseSegment = noiseSegment .* sqrt(cleanPower/noisePower);
noisyAudio = cleanAudio + noiseSegment;
```

Use `stft` to generate magnitude STFT vectors from the noisy audio signals.

```
noisySTFT = stft(noisyAudio,'Window',win,'OverlapLength',overlap,'FFTLength',ffTLength);
noisyPhase = angle(noisySTFT(numFeatures-1:end,:));
noisySTFT = abs(noisySTFT(numFeatures-1:end,:));
```

Generate the 8-segment training predictor signals from the noisy STFT. The overlap between consecutive predictors is 7 segments.

```
noisySTFT = [noisySTFT(:,1:numSegments-1) noisySTFT];
predictors = zeros( numFeatures, numSegments , size(noisySTFT,2) - numSegments + 1);
for index = 1:(size(noisySTFT,2) - numSegments + 1)
    predictors(:,:,index) = noisySTFT(:,index:index + numSegments - 1);
end
```

Normalize the predictors by the mean and standard deviation computed in the training stage.

```
predictors(:) = (predictors(:) - noisyMean) / noisyStd;
```

Compute the denoised magnitude STFT by using `predict` with the two trained networks.

```
predictors = reshape(predictors, [numFeatures,numSegments,1,size(predictors,3)]);
STFTFullyConnected = predict(denoiseNetFullyConnected, predictors);
STFTFullyConvolutional = predict(denoiseNetFullyConvolutional, predictors);
```

Scale the outputs by the mean and standard deviation used in the training stage.

```
STFTFullyConnected(:) = cleanStd * STFTFullyConnected(:) + cleanMean;
STFTFullyConvolutional(:) = cleanStd * STFTFullyConvolutional(:) + cleanMean;
```

Convert the one-sided STFT to a centered STFT.

```
STFTFullyConnected = STFTFullyConnected.' .* exp(1j*noisyPhase);
STFTFullyConnected = [conj(STFTFullyConnected(end-1:-1:2,:)); STFTFullyConnected];
STFTFullyConvolutional = squeeze(STFTFullyConvolutional) .* exp(1j*noisyPhase);
STFTFullyConvolutional = [conj(STFTFullyConvolutional(end-1:-1:2,:)) ; STFTFullyConvolutional];
```

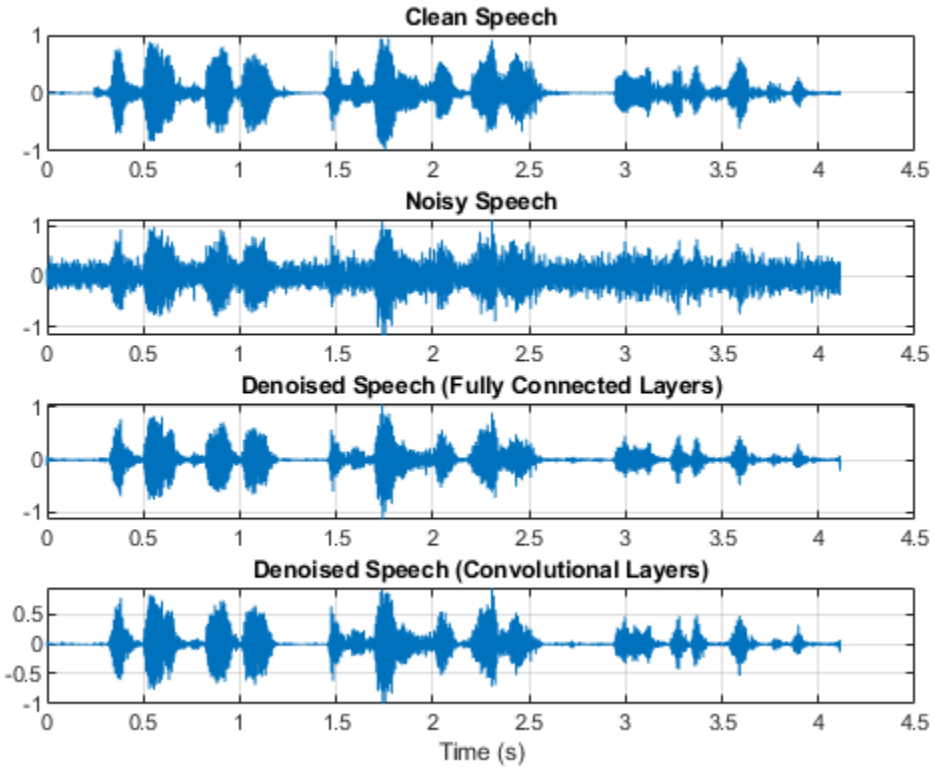
Compute the denoised speech signals. `istft` performs the inverse STFT. Use the phase of the noisy STFT vectors to reconstruct the time-domain signal.

```
denoisedAudioFullyConnected = istft(STFTFullyConnected,  ...
                                    'Window',win,'OverlapLength',overlap, ...
                                    'FFTLength',ffTLength,'ConjugateSymmetric',true);
```

```
denoisedAudioFullyConvolutional = istft(STFTFullyConvolutional,  ...
                                        'Window',win,'OverlapLength',overlap, ...
                                        'FFTLength',ffTLength,'ConjugateSymmetric',true);
```

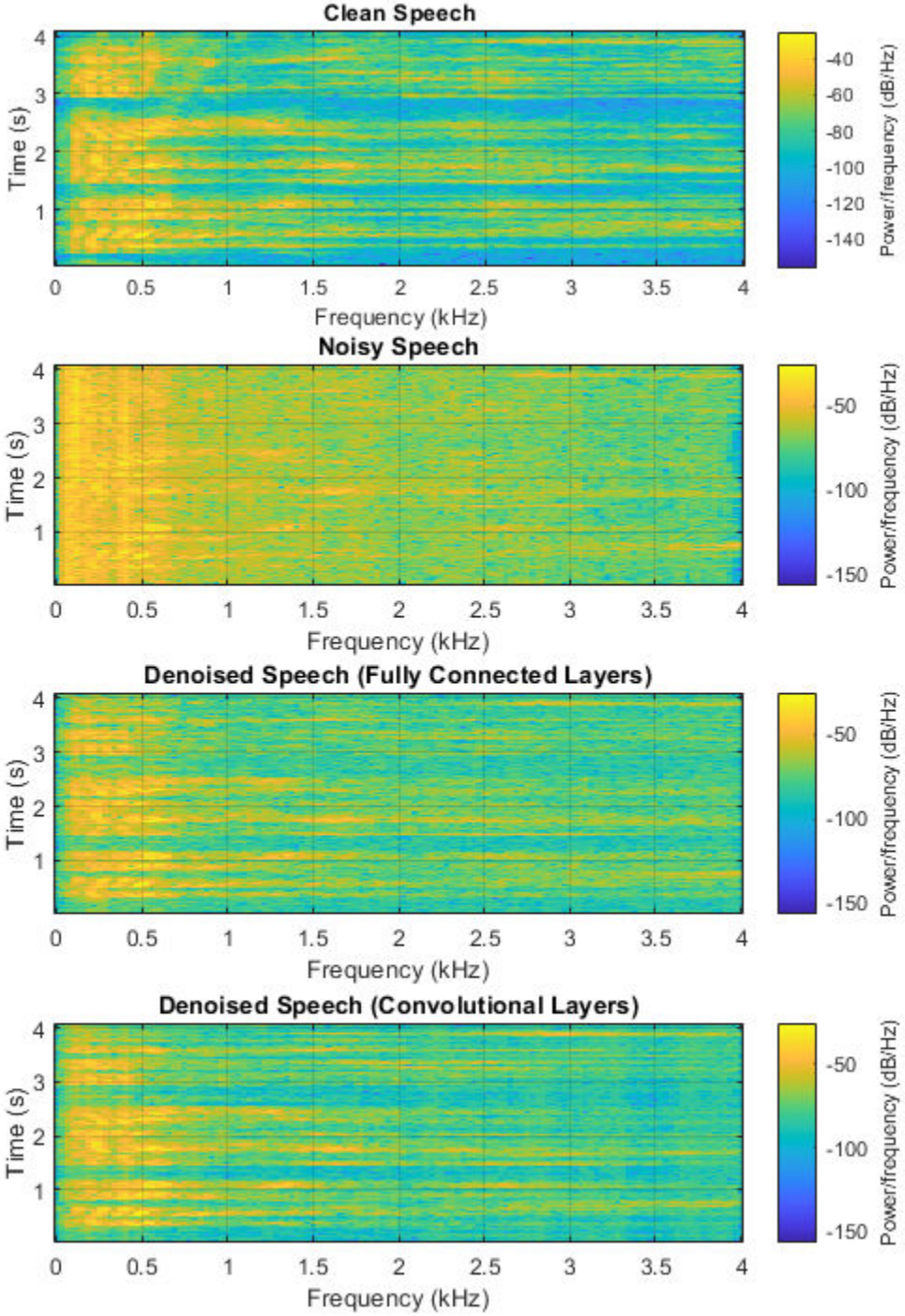
Plot the clean, noisy and denoised audio signals.

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

figure

subplot(4,1,1)
plot(t,cleanAudio(1:numel(denoisedAudioFullyConnected)))
title("Clean Speech")
grid on

subplot(4,1,2)
plot(t,noisyAudio(1:numel(denoisedAudioFullyConnected)))
title("Noisy Speech")
grid on

subplot(4,1,3)
plot(t,denoisedAudioFullyConnected)
title("Denoised Speech (Fully Connected Layers)")
grid on

subplot(4,1,4)
plot(t,denoisedAudioFullyConvolutional)
title("Denoised Speech (Convolutional Layers)")
grid on
xlabel("Time (s)")
```




Plot the clean, noisy, and denoised spectrograms.

```
h = figure;

subplot(4,1,1)
spectrogram(cleanAudio,win,overlap,ffTLength,fs);
title("Clean Speech")
grid on

subplot(4,1,2)
spectrogram(noisyAudio,win,overlap,ffTLength,fs);
title("Noisy Speech")
grid on

subplot(4,1,3)
spectrogram(denoisedAudioFullyConnected,win,overlap,ffTLength,fs);
title("Denoised Speech (Fully Connected Layers)")
grid on

subplot(4,1,4)
spectrogram(denoisedAudioFullyConvolutional,win,overlap,ffTLength,fs);
title("Denoised Speech (Convolutional Layers)")
grid on

p = get(h,'Position');
set(h,'Position',[p(1) 65 p(3) 800]);
```




Listen to the noisy speech.

```
sound(noisyAudio,fs)
```

Listen to the denoised speech from the network with fully connected layers.

```
sound(denoisedAudioFullyConnected,fs)
```

Listen to the denoised speech from the network with convolutional layers.

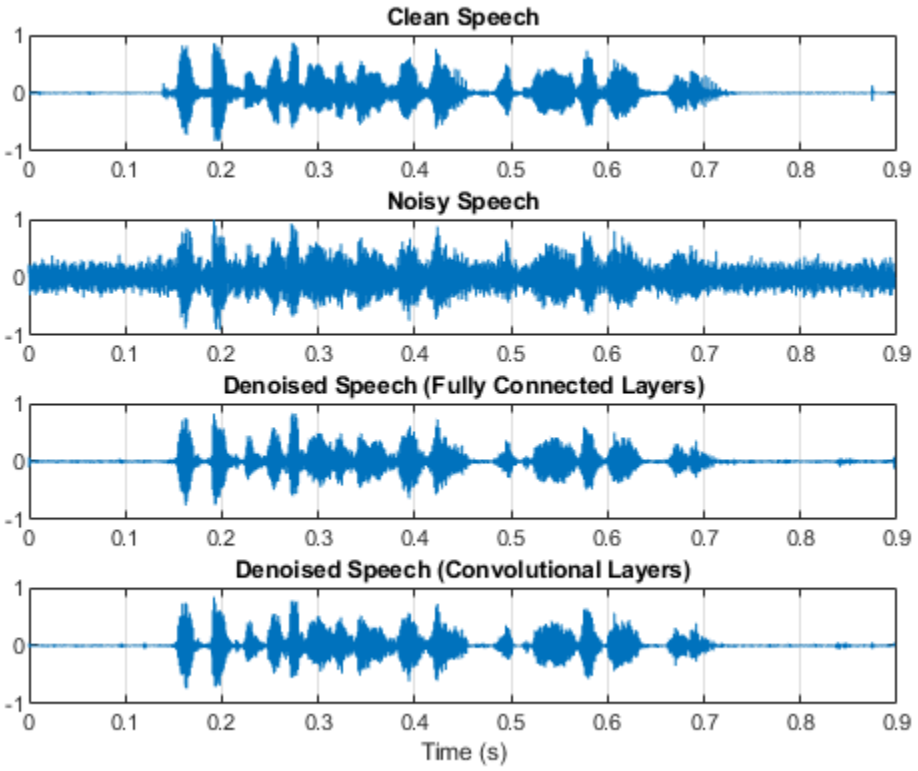
```
sound(denoisedAudioFullyConvolutional,fs)
```

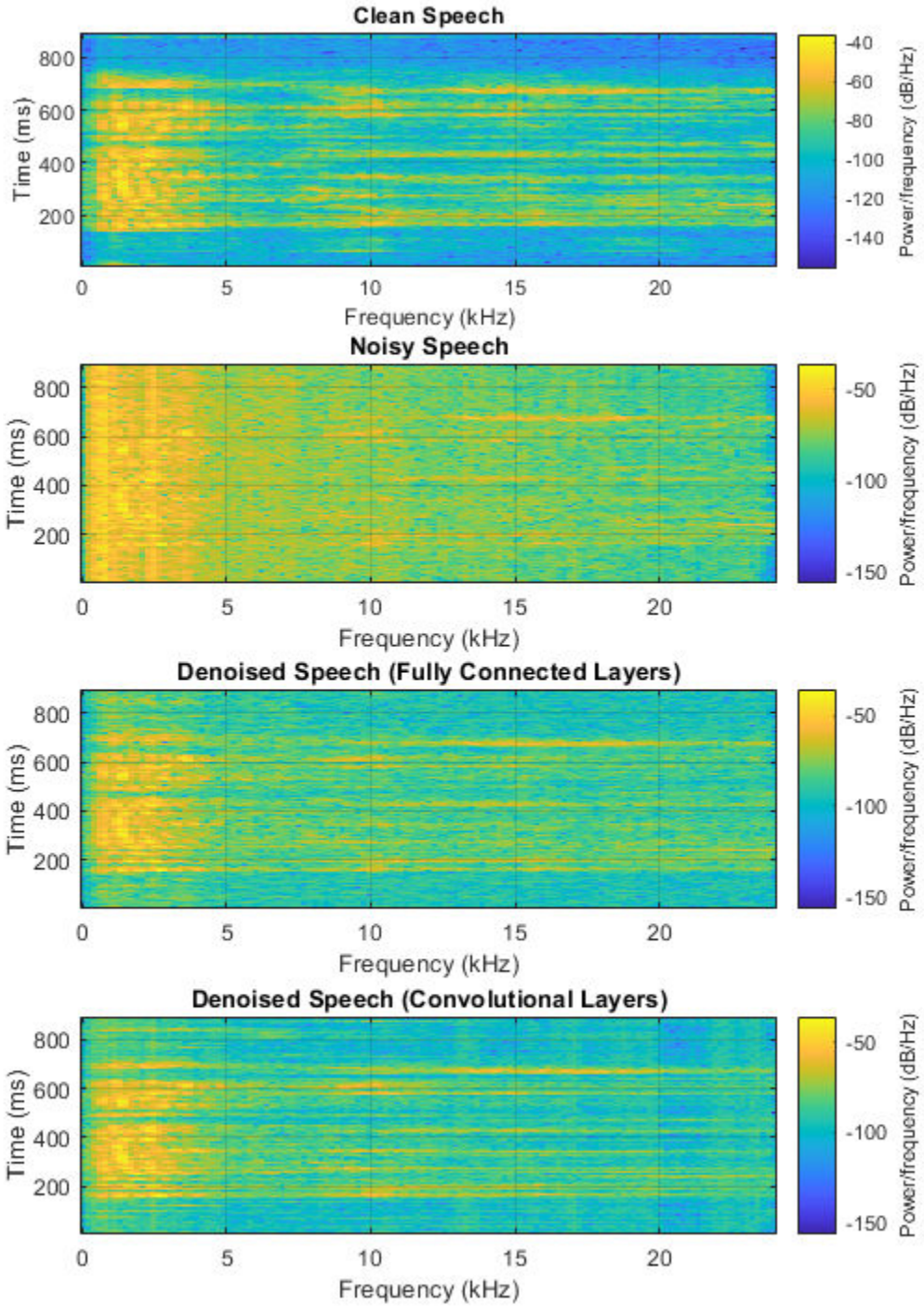
Listen to clean speech.

```
sound(cleanAudio,fs)
```

You can test more files from the datastore by calling `testDenoisingNets`. The function produces the time-domain and frequency-domain plots highlighted above, and also returns the clean, noisy, and denoised audio signals.

```
[cleanAudio,noisyAudio,denoisedAudioFullyConnected,denoisedAudioFullyConvolutional] = testDenois
```



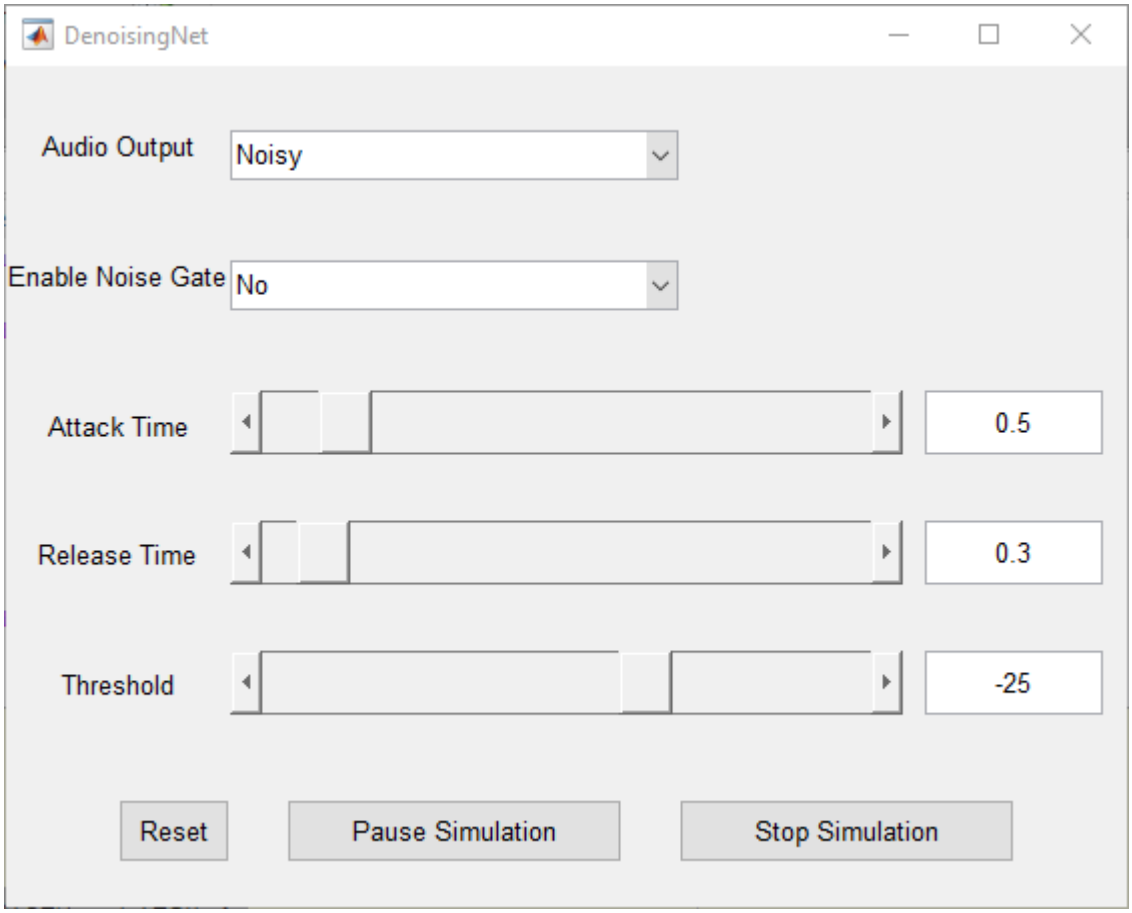

Clean Speech
Time (ms)
800
600
400
200
0
5
10
15
20
Frequency (kHz)
-40
-60
-80
-100
-120
-140
Power/frequency (dB/Hz)
Noisy Speech
Time (ms)
800
600
400
200
0
5
10
15
20
Frequency (kHz)
-50
-100
-150
Power/frequency (dB/Hz)
Denoised Speech (Fully Connected Layers)
Time (ms)
800
600
400
200
0
5
10
15
20
Frequency (kHz)
-50
-100
-150
Power/frequency (dB/Hz)
Denoised Speech (Convolutional Layers)
Time (ms)
800
600
400
200
0
5
10
15
20
Frequency (kHz)
-50
-100
-150
Power/frequency (dB/Hz)

### Real-Time Application

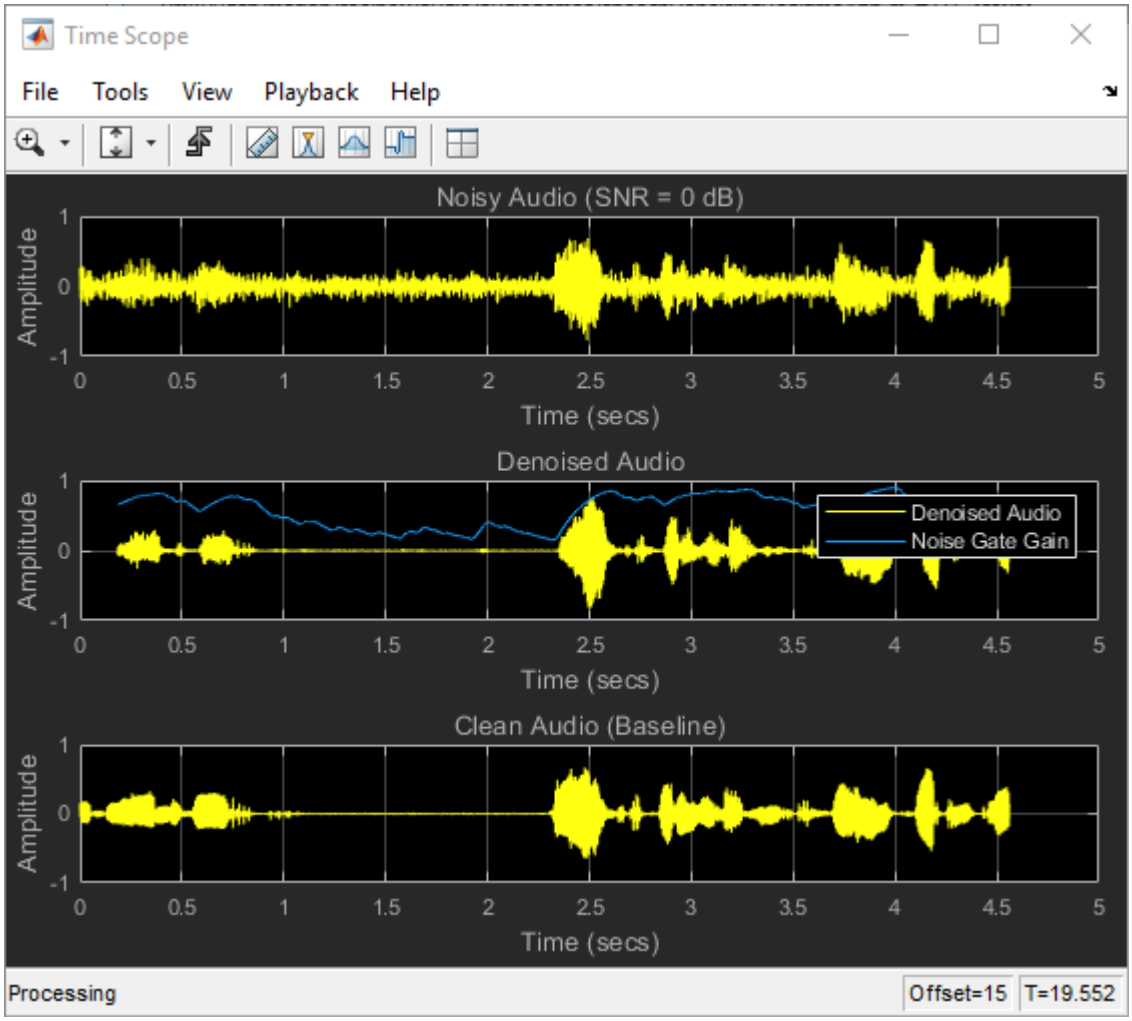
The procedure in the previous section passes the entire spectrum of the noisy signal to `predict`. This is not suitable for real-time applications where low latency is a requirement.

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.

# Order Analysis of a Vibration Signal

This example shows how to analyze a vibration signal using order analysis. Order analysis is used to quantify noise or vibration in rotating machinery whose rotational speed changes over time. An order refers to a frequency that is a certain multiple of a reference rotational speed. For example, a vibration signal with a frequency equal to twice the rotational frequency of a motor corresponds to an order of two and, likewise, a vibration signal that has a frequency equal to 0.5 times the rotational frequency of the motor corresponds to an order of 0.5. In this example, orders of large amplitude are determined to investigate the source of unwanted vibration in a helicopter cabin.

**Introduction**

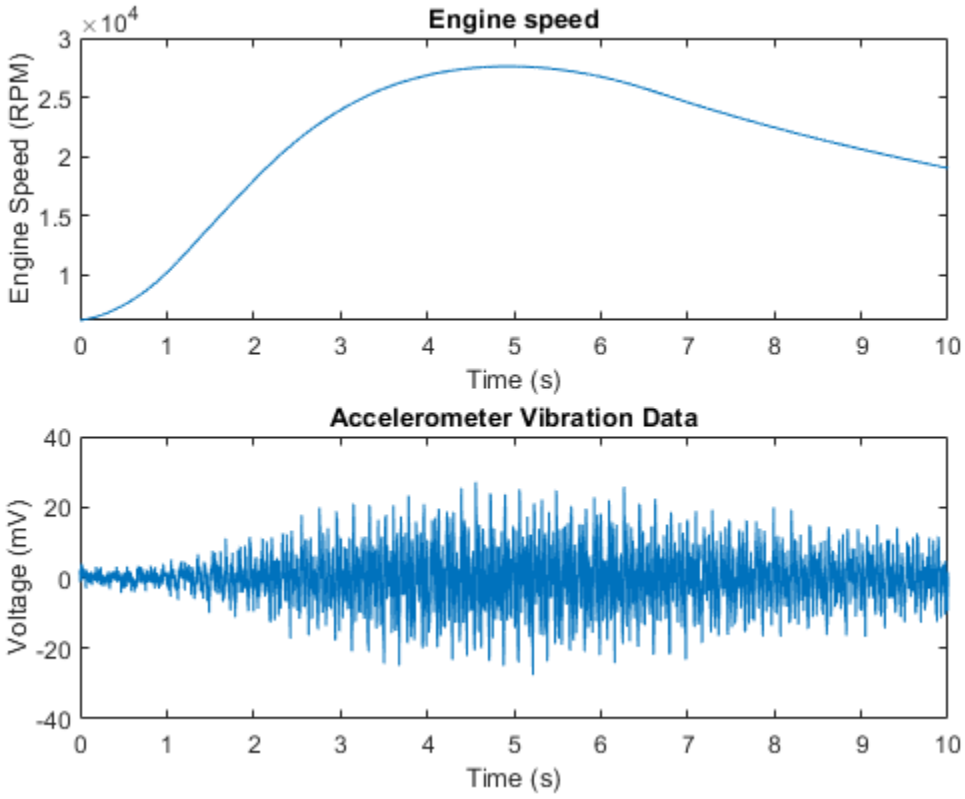
This example analyzes simulated vibration data from an accelerometer in the cabin of a helicopter during a run-up and coast-down of the main motor. A helicopter has several rotating components, including the engine, gearbox, and the main and tail rotors. Each component rotates at a known, fixed rate with respect to the main motor, and each may contribute to unwanted vibration. The frequency of dominant vibration components can be related to the rotational speed of the motor to investigate the source of high-amplitude vibration. The helicopter in this example has four blades in both the main and tail rotors. Important components of vibration from a helicopter rotor may be found at integer multiples of the rotational frequency of the rotor when the vibration is generated by the rotor blades.

The signal in this example is a time-dependent voltage, `vib`, sampled at a rate `fs` equal to 500 Hz. The data include `rpm`, the angular speed of the turbine engine, and a vector `t` of time instants. The ratio of rotor speed to engine speed for each rotor is stored in the variables `mainRotorEngineRatio` and `tailRotorEngineRatio`.

A motor speed signal commonly consists of a sequence of tachometer pulses. `tachorpm` can be used to extract an RPM signal from a tachometer pulse signal. `tachorpm` automatically identifies the pulse locations of a bilevel tachometer waveform and computes the interval between pulses to estimate rotational speed. In this example, the motor speed signal contains the rotational speed, `rpm`, and hence no conversion is needed.

Plot the motor speed and vibration data as functions of time:

```
load helidata
vib = vib - mean(vib);  % Remove the DC component

subplot(2,1,1)
plot(t,rpm)             % Plot the engine rotational speed
xlabel('Time (s)')
ylabel('Engine Speed (RPM)')
title('Engine speed')

subplot(2,1,2)
plot(t,vib)             % Plot the vibration signal
xlabel('Time (s)')
ylabel('Voltage (mV)')
title('Accelerometer Vibration Data')
```



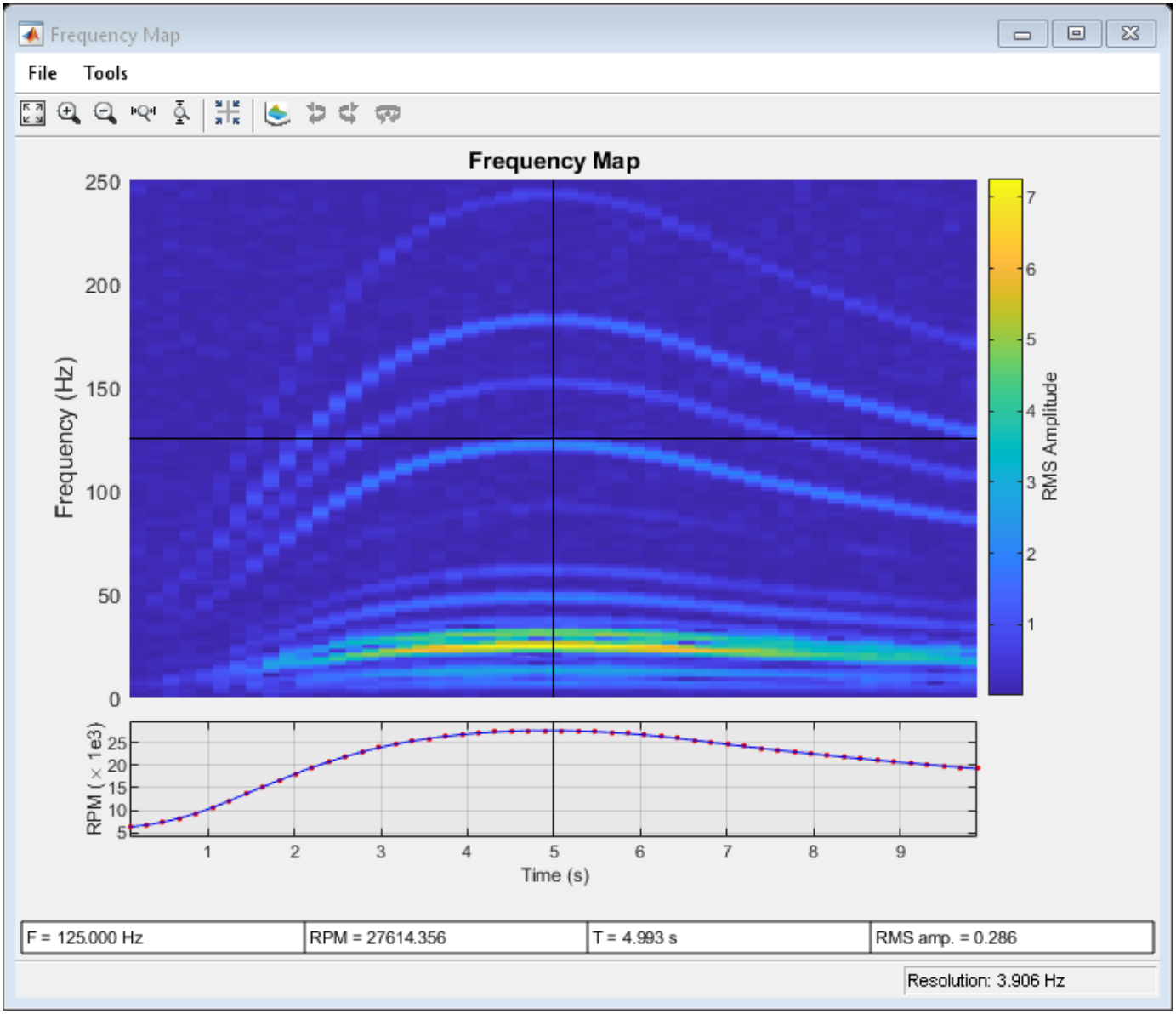

The engine speed increases during the run-up and decreases during the coast-down. The vibration amplitude changes as a function of rotational speed. This type of RPM profile is typical for analyzing vibration in rotating machinery.

**Visualizing Data Using An RPM-Frequency Map**

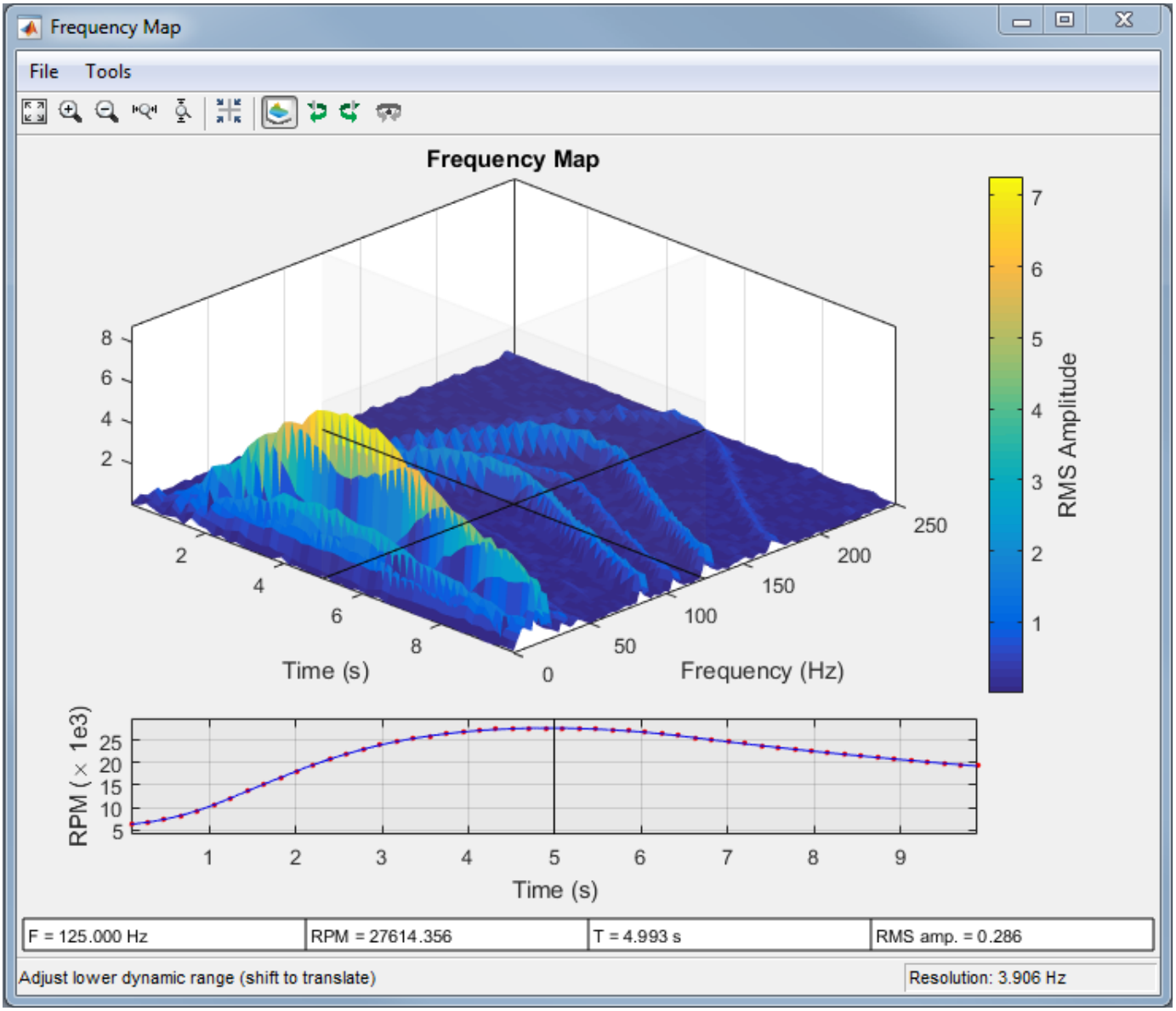
The vibration signal can be visualized in the frequency domain using the function `rpmfreqmap`. This function computes the short-time Fourier transform of the signal and generates an RPM-frequency map. `rpmfreqmap` displays the map in an interactive plot window when output arguments are omitted.

Generate and visualize an RPM-frequency map for the vibration data.

```
rpmfreqmap(vib,fs,rpm)
```




The interactive figure window produced by `rpmfreqmap` contains an RPM-frequency map, an RPM versus time curve corresponding to the map, and several numeric indicators that can be used to quantify vibration components. The amplitude of the map represents root-mean-square (RMS) amplitude by default. Other amplitude choices, including peak amplitude and power, can be specified with optional arguments. The waterfall plot menu button generates a three-dimensional view:



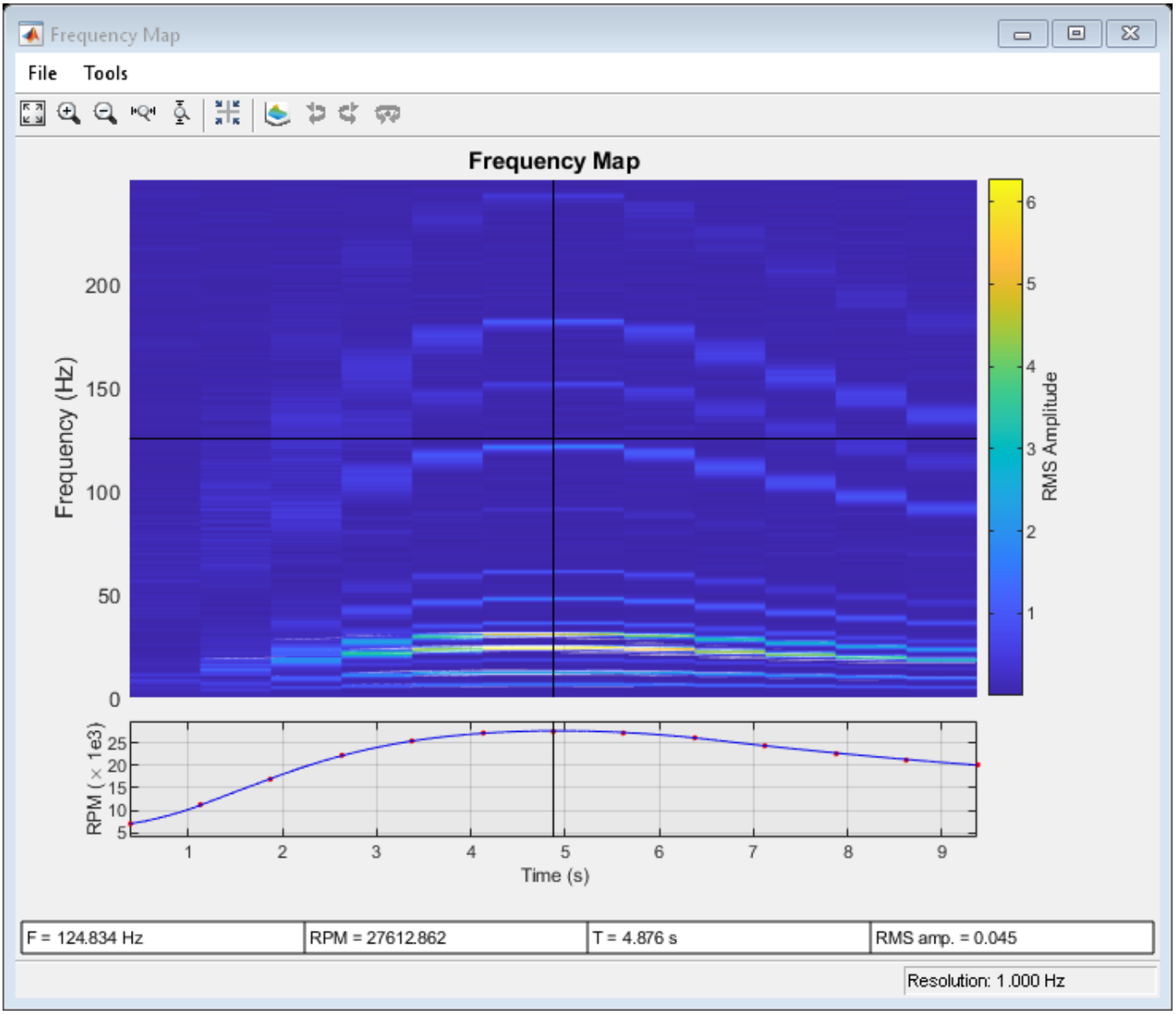

Many of the tracks in the RPM-frequency maps have frequencies that increase and decrease with the motor speed. This suggests that the tracks are orders of the motor rotational frequency. There are high-amplitude components near the RPM peak, with frequencies between 20 and 30 Hz. The crosshair cursor can be placed on the map at this location to view the frequency, RPM value, time, and map amplitude in the indicator boxes below the RPM curve.

By default, `rpmfreqmap` computes the resolution by dividing the sample rate by 128. The resolution is displayed in the lower-right corner of the figure, and is equal to 3.906 Hz in this case. A Hann window is used by default, but several other windows are available.

Pass a smaller the value of the resolution to `rpmfreqmap` to resolve certain frequency components better. For example, low-frequency components are not separated at the peak RPM. At low RPM values, the high-amplitude tracks appear to blend together.

Generate an RPM-frequency map with a resolution of 1 Hz to resolve these components.

```
rpmfreqmap(vib,fs,rpm,1)
```




Low-frequency components can now be resolved at the peak RPM, but there is significant smearing present when the rotational speed is changing more rapidly. Vibration orders change frequency within each time window as the motor speed increases or decreases, producing a broader spectral track. This smearing effect is more pronounced for a finer resolution because of the longer time windows that are required. In this case, improving the spectral resolution resulted in increased smearing artifacts during the run-up and coast-down phases. An order map can be generated to avoid this trade-off.

**Visualizing Data Using An RPM-Order Map**

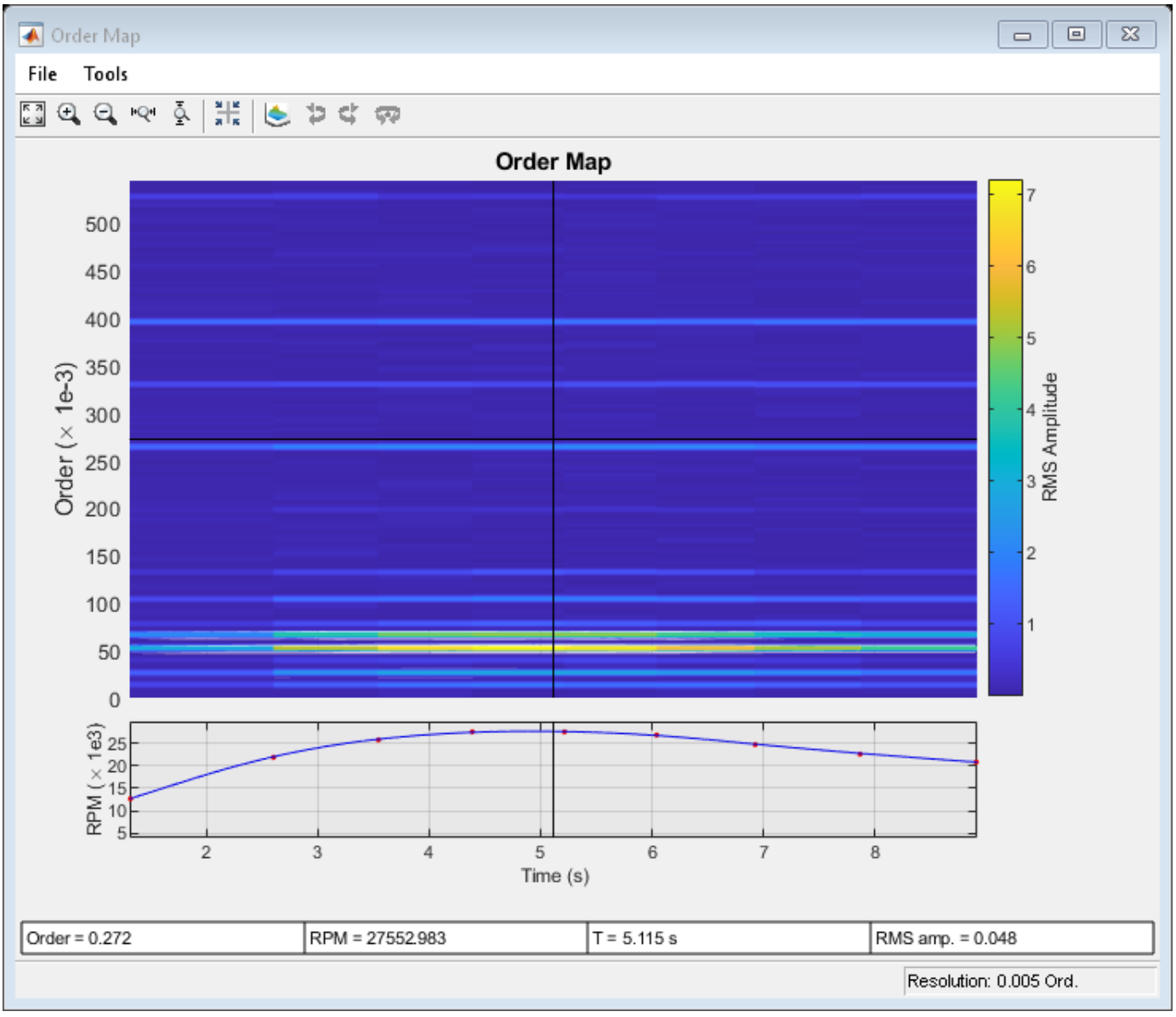
The function `rpmordermap` generates a spectral map of order versus RPM for order analysis. The approach removes smearing artifacts by resampling the signal at constant phase increments, producing a stationary sinusoid for each order. The resampled signal is analyzed using a short-time

Fourier transform. Since each order is a fixed multiple of the reference rotational speed, an order map contains a straight order track as a function of RPM for each order.

The function `rpmordermap` accepts the same arguments as `rpmfreqmap` and also produces an interactive plot window when called with no output arguments. The resolution parameter is now specified in orders, rather than in Hz, and the spectral axis of the map is now order, rather than frequency. The function uses a flat-top window by default.

Visualize the order map of the helicopter data using `rpmordermap`. Specify an order resolution of 0.005 orders.

```
rpmordermap(vib,fs,rpm,0.005)
```



The map contains a straight track for each order, indicating that the vibration occurs at a fixed multiple of the motor rotational speed. Order maps make it easy to relate each spectral component to the motor speed. Smearing artifacts are significantly reduced compared to the RPM-frequency map.

**Determining Peak Orders Using an Average Order Spectrum**

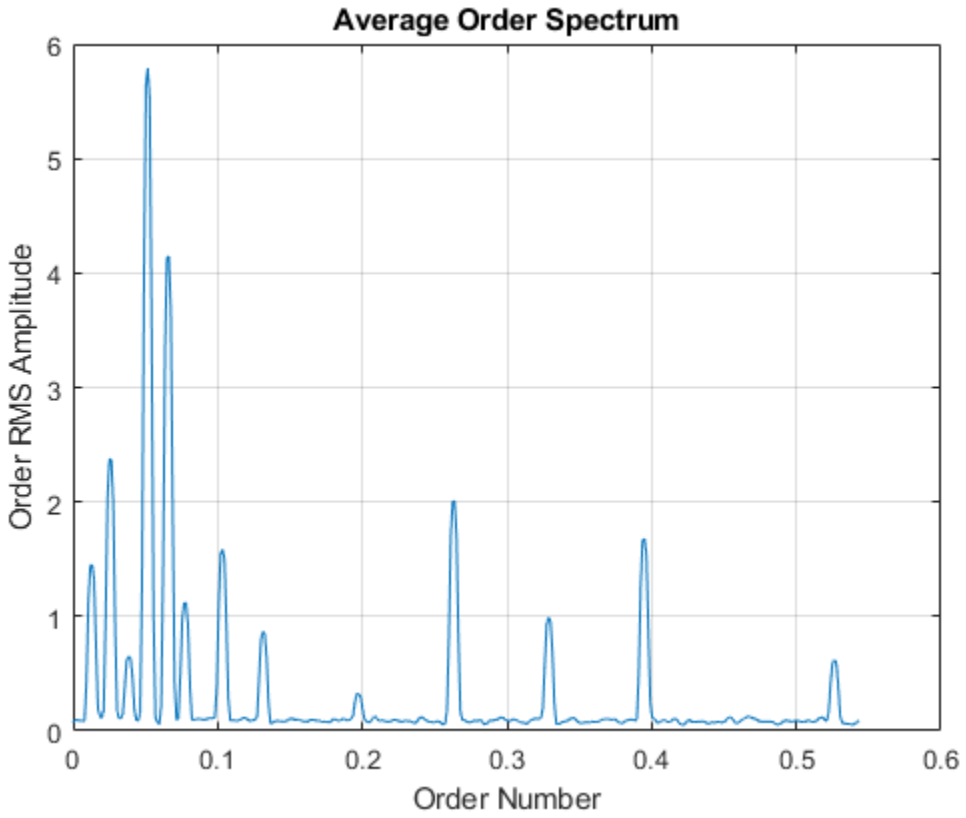
Next, determine the locations of the peaks of the order map. Look for orders that are integer multiples of the order of the main and tail rotors, where vibration generated by these rotors would occur. The function `rpmordermap` returns the map and corresponding order and RPM values as outputs. Analyze the data to determine the orders of high-amplitude vibration in the helicopter cabin.

Compute and return an order map of the data.

```
[map,mapOrder,mapRPM,mapTime] = rpmordermap(vib,fs,rpm,0.005);
```

Next, use `orderspectrum` to compute and plot the average spectrum of `map`. The function takes the order map generated by `rpmordermap` as input and averages it over time.

```
figure
orderspectrum(map,mapOrder)
```




Return the average spectrum and call `findpeaks` to return the locations of the two highest peaks.

```
[spec,specOrder] = orderspectrum(map,mapOrder);
[~,peakOrders] = findpeaks(spec,specOrder,'SortStr','descend','NPeaks',2);
peakOrders = round(peakOrders,3)
```

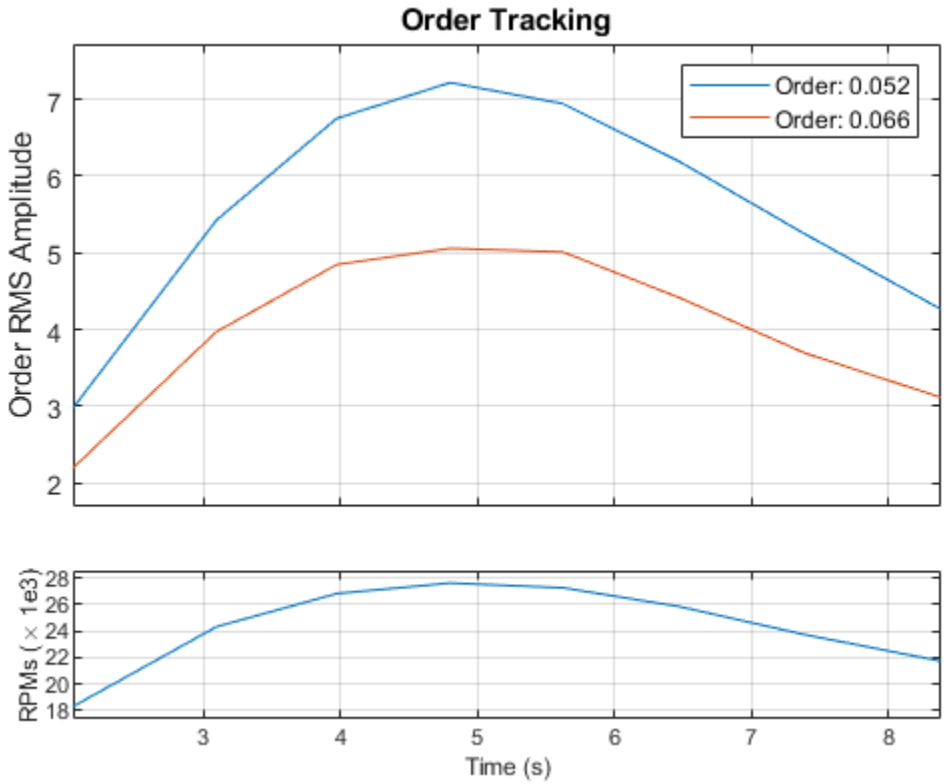
```
peakOrders = 2×1

    0.0520
    0.0660
```

Two closely spaced dominant peaks can be seen around order 0.05 in the plot. The orders are less than one because the frequency of vibration is lower that the rotational speed of the motor.

**Analyzing Peak Orders Over Time**

Next, find the amplitudes of the peak orders as a function of time using `ordertrack`. Use `map` as an input and plot the amplitude of the two peak orders by calling `ordertrack` with no output arguments.

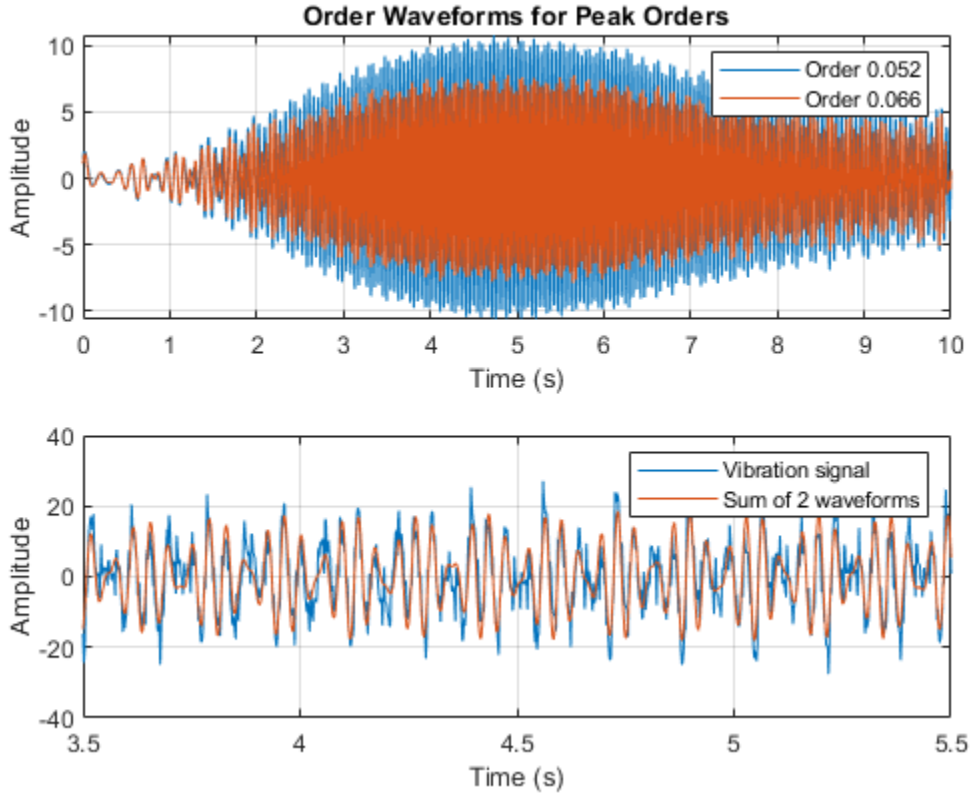
```
ordertrack(map,mapOrder,mapRPM,mapTime,peakOrders)
```




Both orders increase in amplitude as the rotational speed of the motor increases. Although orders can be easily separated in this case, `ordertrack` can also separate crossing orders when multiple RPM signals are present.

Next, extract a time-domain order waveform for each peak order using `orderwaveform`. Order waveforms can be compared directly to the original vibration signal and played back as audio. `orderwaveform` uses the Vold-Kalman filter to extract order waveforms for specified orders. Compare the sum of two peak-order waveforms to the original signal.

```
orderWaveforms = orderwaveform(vib,fs,rpm,peakOrders);
helperPlotOrderWaveforms(t,orderWaveforms,vib)
```




### Reducing Cabin Vibration

To identify the sources of the cabin vibration, compare the order of each peak to the order of each of the helicopter's rotors. The order of each rotor is equal to the fixed ratio of the rotor speed to the engine speed.

```
mainRotorOrder = mainRotorEngineRatio;
tailRotorOrder = tailRotorEngineRatio;

ratioMain = peakOrders/mainRotorOrder
```

```
ratioMain = 2×1

    4.0310
    5.1163
```

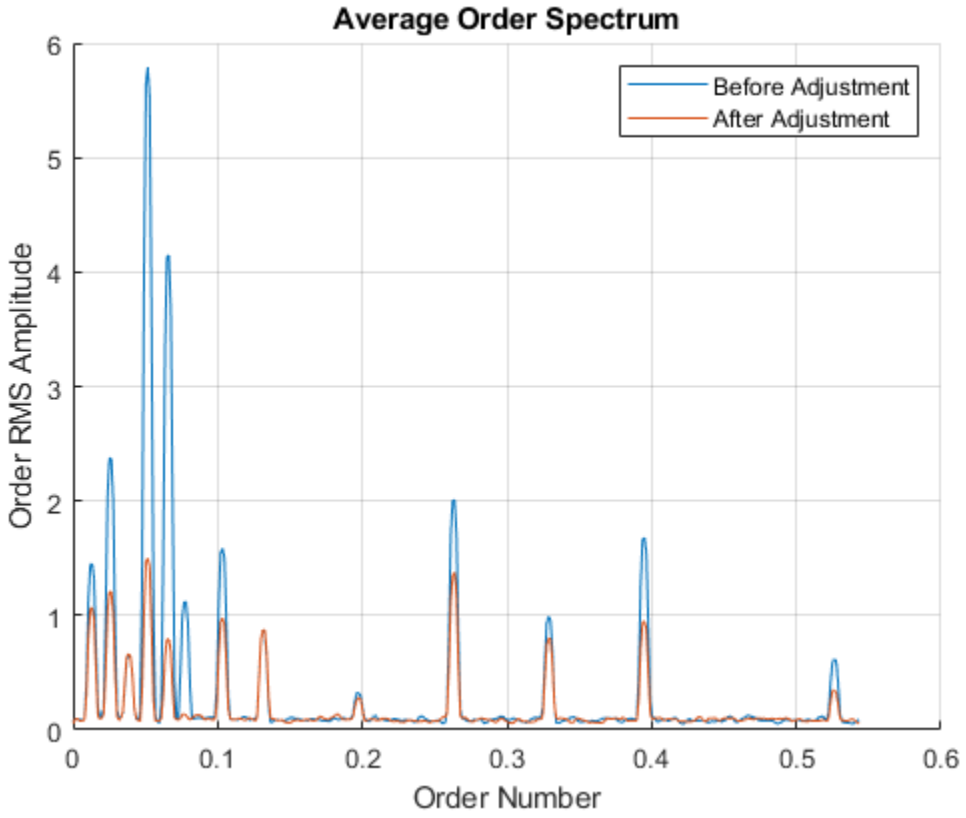
```
ratioTail = peakOrders/tailRotorOrder
```

```
ratioTail = 2×1

    0.7904
    1.0032
```

The highest peak is located at order four of the main rotor speed, so the frequency of the maximum-amplitude component is four times the frequency of the main rotor. The main rotor, which has four blades, is a good candidate for the source of this vibration because, for a helicopter with $N$ blades per rotor, vibration at $N$ times the rotor rotational speed is common. Likewise, the second largest component is located at order one of the tail rotor speed, suggesting vibration may originate from the tail rotor. Because the speeds of the rotors are not related by an integer factor, the order of the second largest peak with respect to the main rotor speed is not an integer.

After making track and balance adjustments to the main and tail rotors, a new data set is collected. Load it and compare the order spectra before and after the adjustment.

```
load helidataAfter
vib = vib - mean(vib);              % Remove the DC component
[mapAfter,mapOrderAfter] = rpmordermap(vib,fs,rpm,0.005);
figure
hold on
orderspectrum(map,mapOrder)
orderspectrum(mapAfter,mapOrderAfter)
legend('Before Adjustment','After Adjustment')
```




The amplitudes of the dominant peaks are now considerably lower.

**Conclusions**

This example used order analysis to identify the main and tail rotors of a helicopter as potential sources of high-amplitude vibration in the cabin. First, `rpmfreqmap` and `rpmordermap` were used to visualize orders. The RPM-order map provided order separation throughout the RPM range without

the smearing artifacts present in the RPM-frequency map. `rpmordermap` was the best choice to visualize vibration components at lower RPM during the engine run-up and coast-down.

Next, the example used `orderspectrum` to identify peak orders, `ordertrack` to visualize the amplitude of the peak orders over time, and `orderwaveform` to extract time-domain waveforms for the peak orders. The order of the largest amplitude vibration component was found at four times the rotational frequency of the main rotor, indicating an imbalance in the main rotor blades. The second largest component was found at the rotational frequency of the tail rotor. Adjustments to the rotors resulted in reduced vibration levels.

**References**

Brandt, Anders. *Noise and Vibration Analysis: Signal Analysis and Experimental Procedures*. Chichester, UK: John Wiley and Sons, 2011.

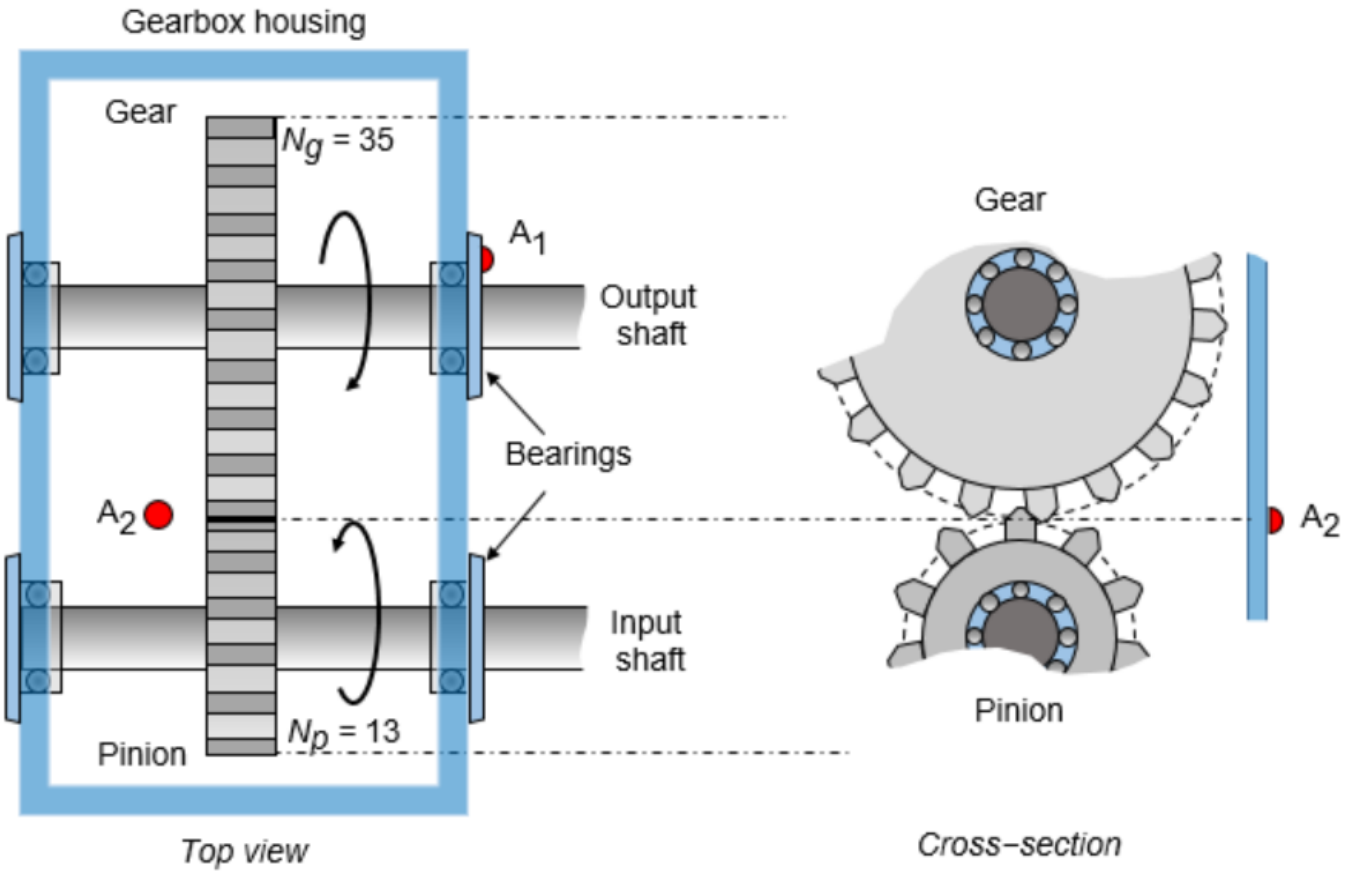
## See Also

`orderspectrum` | `ordertrack` | `orderwaveform` | `rpmfreqmap` | `rpmordermap`

# Vibration Analysis of Rotating Machinery

This example shows how to analyze vibration signals from a gearbox using time-synchronous averaging and envelope spectra. These functions are especially useful in the predictive maintenance of gearboxes, which contain multiple rotating components: gears, shafts and bearings.

This example generates and analyzes vibration data for a gearbox whose shafts rotate at a fixed speed. Time-synchronous averaging is used to isolate vibration components associated with a specific shaft or gear and average out all other components. Envelope spectra are especially useful in identifying localized bearing faults that cause high-frequency impacts.




Consider an idealized gearbox that consists of a 13-tooth pinion meshing with a 35-tooth gear. The pinion is coupled to an input shaft connected to a prime mover. The gear is connected to an output shaft. The shafts are supported by roller bearings on the gearbox housing. Two accelerometers, $A_1$ and $A_2$, are placed on the bearing and gearbox housings, respectively. The accelerometers operate at a sample rate of 20 kHz.

The pinion rotates at a rate $f_{\text{Pinion}}$ = 22.5 Hz or 1350 rpm. The rotating speed of the gear and output shaft is

$$f_{\text{Gear}} = f_{\text{Pinion}} \times \frac{\text{Number of pinion teeth } (N_p)}{\text{Number of gear teeth } (N_g)} .$$

The tooth-mesh frequency, also called gear-mesh frequency, is the rate at which gear and pinion teeth periodically engage:

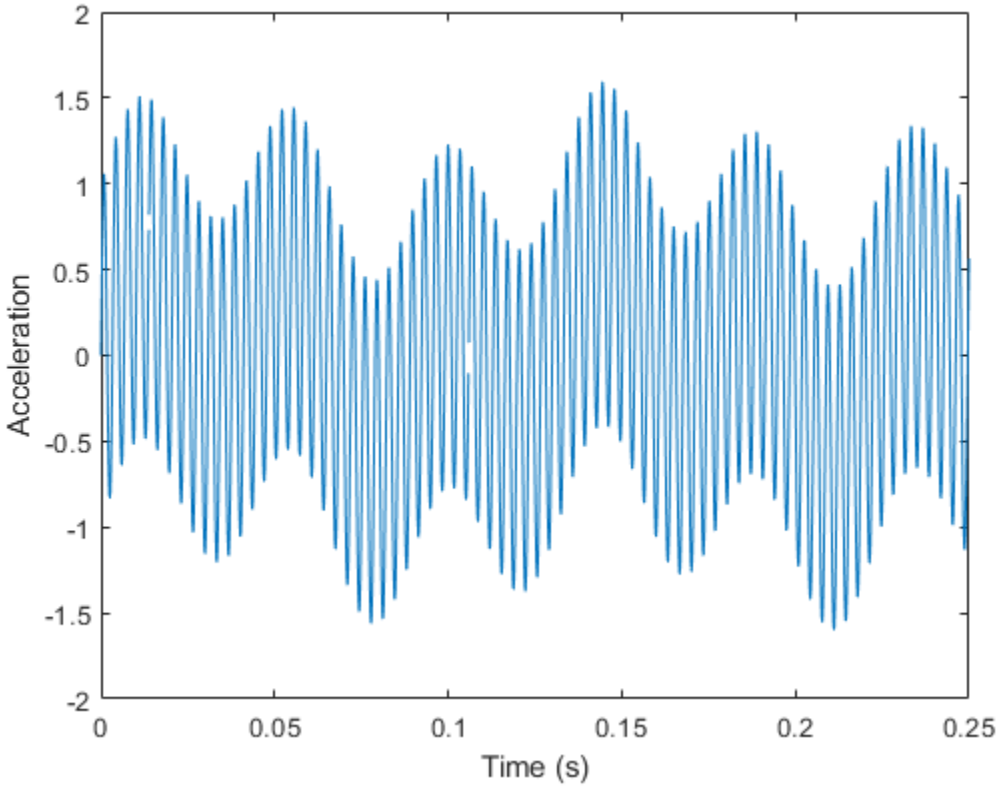
$f_{\text{Mesh}} = f_{\text{Pinion}} \times N_p = f_{\text{Gear}} \times N_g$ .

```
fs = 20E3;          % Sample Rate (Hz)

Np = 13;            % Number of teeth on pinion
Ng = 35;            % Number of teeth on gear

fPin = 22.5;        % Pinion (Input) shaft frequency (Hz)

fGear = fPin*Np/Ng; % Gear (Output) shaft frequency (Hz)

fMesh = fPin*Np;    % Gear Mesh frequency (Hz)
```

Generate vibration waveforms for the pinion and the gear. Model the vibrations as sinusoids occurring at primary shaft gear mesh frequencies. Analyze 20 seconds of vibration data.

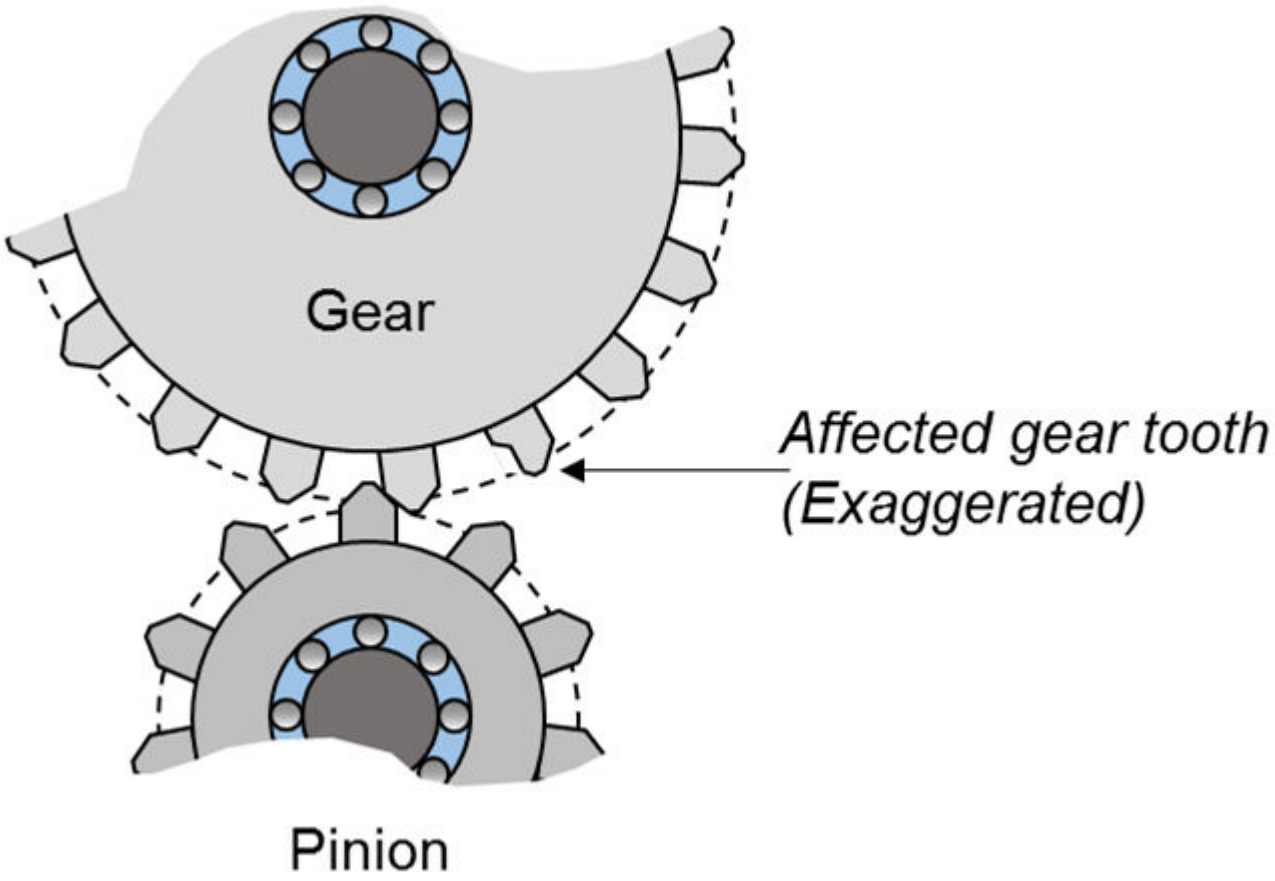
The gear-mesh waveform is responsible for transmitting load and thus possesses the highest vibration amplitude. $A_2$ records vibration contributions from the two shafts and the gear-mesh. For this experiment, the contributions of the bearing rolling elements to the vibration signals recorded by $A_2$ are considered negligible. Visualize a section of noise-free vibration signal.

```
t = 0:1/fs:20-1/fs;

vfIn = 0.4*sin(2*pi*fPin*t);    % Pinion waveform
vfOut = 0.2*sin(2*pi*fGear*t);  % Gear waveform

vMesh = sin(2*pi*fMesh*t);      % Gear-mesh waveform

plot(t, vfIn + vfOut + vMesh)
xlim([0 0.25])
xlabel('Time (s)')
ylabel('Acceleration')
```




**Generate High-Frequency Impacts Caused by a Local Fault on a Gear Tooth**

Assume that one of the teeth of the gear is suffering from a local fault such as a spall. This results in a high-frequency impact occurring once per rotation of the gear.




The local fault causes an impact that has a duration shorter than the duration of tooth mesh. A dent on the tooth surface of the gear generates high-frequency oscillations over the duration of the impact. The frequency of impact is dependent on gearbox component properties and its natural frequencies. In this example, it is arbitrarily assumed that the impact causes a 2 kHz vibration signal and occurs over a duration of about 8% of `1/fMesh`, or 0.25 milliseconds. The impact repeats once per rotation of the gear.

```
ipf = fGear;
fImpact = 2000;

tImpact = 0:1/fs:2.5e-4-1/fs;
xImpact = sin(2*pi*fImpact*tImpact)/3;
```

Make the impact periodic by convolving it with a comb function.

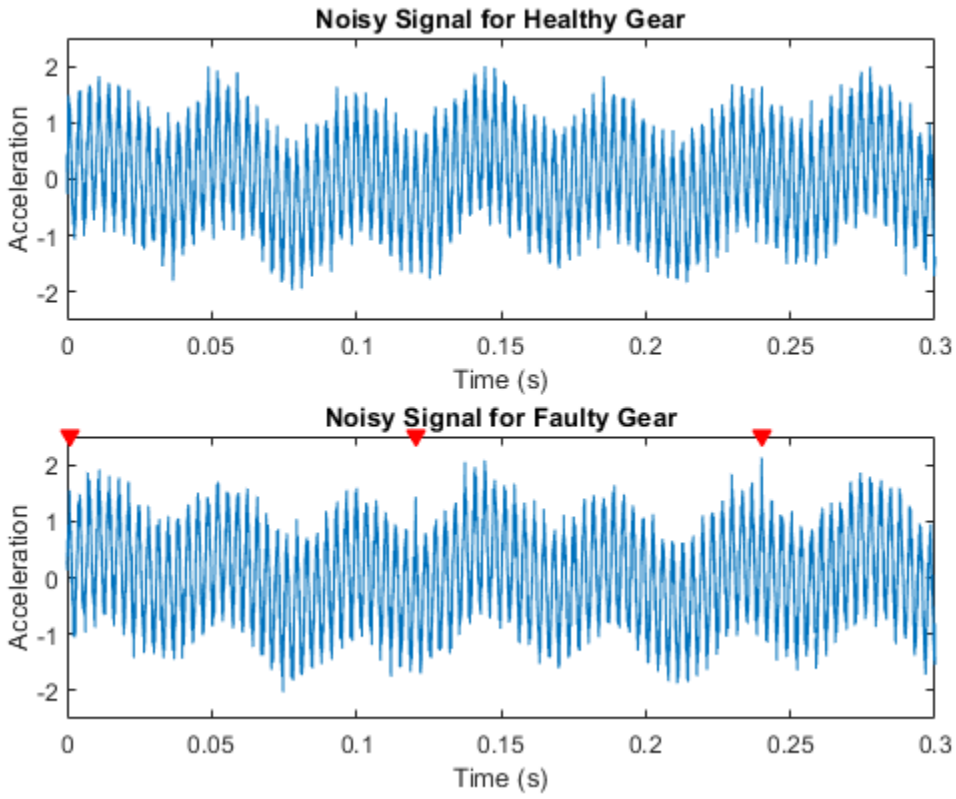
```
xComb = zeros(size(t));

Ind = (0.25*fs/fMesh):(fs/ipf):length(t);
Ind = round(Ind);
xComb(Ind) = 1;

xPer = 2*conv(xComb,xImpact,'same');
```

Add the fault signal `xPer` to the shaft signal. Add white Gaussian noise to the output signals for both the fault-free and the faulty gear to model the output from $A_2$.

```
vNoFault = vfIn + vfOut + vMesh;
vFault = vNoFault + xPer;

vNoFaultNoisy = vNoFault + randn(size(t))/5;
vFaultNoisy = vFault + randn(size(t))/5;
```

Visualize a segment of the time history. The impact locations are indicated on the plot for the faulty gear by the inverted red triangles. They are almost indistinguishable.

```
subplot(2,1,1)
plot(t,vNoFaultNoisy)
xlabel('Time (s)')
ylabel('Acceleration')
xlim([0.0 0.3])
ylim([-2.5 2.5])
title('Noisy Signal for Healthy Gear')


subplot(2,1,2)
plot(t,vFaultNoisy)
xlabel('Time (s)')
ylabel('Acceleration')
xlim([0.0 0.3])
ylim([-2.5 2.5])
title('Noisy Signal for Faulty Gear')
hold on
MarkX = t(Ind(1:3));
MarkY = 2.5;
plot(MarkX,MarkY,'rv','MarkerFaceColor','red')
hold off
```




### Compare Power Spectra for Both Signals

Localized tooth faults cause distributed sidebands to appear in the neighborhood of the gear mesh frequency:

$$f_{\text{sideband, Pinion}} = f_{\text{Mesh}} \pm m \times f_{\text{Pinion}} \qquad \forall\, m \in \{1,\, 2,\, 3,\, \ldots\}$$

$$f_{\text{sideband, Gear}} = f_{\text{Mesh}} \pm m \times f_{\text{Gear}} \qquad \forall\, m \in \{1,\, 2,\, 3,\, \ldots\}$$

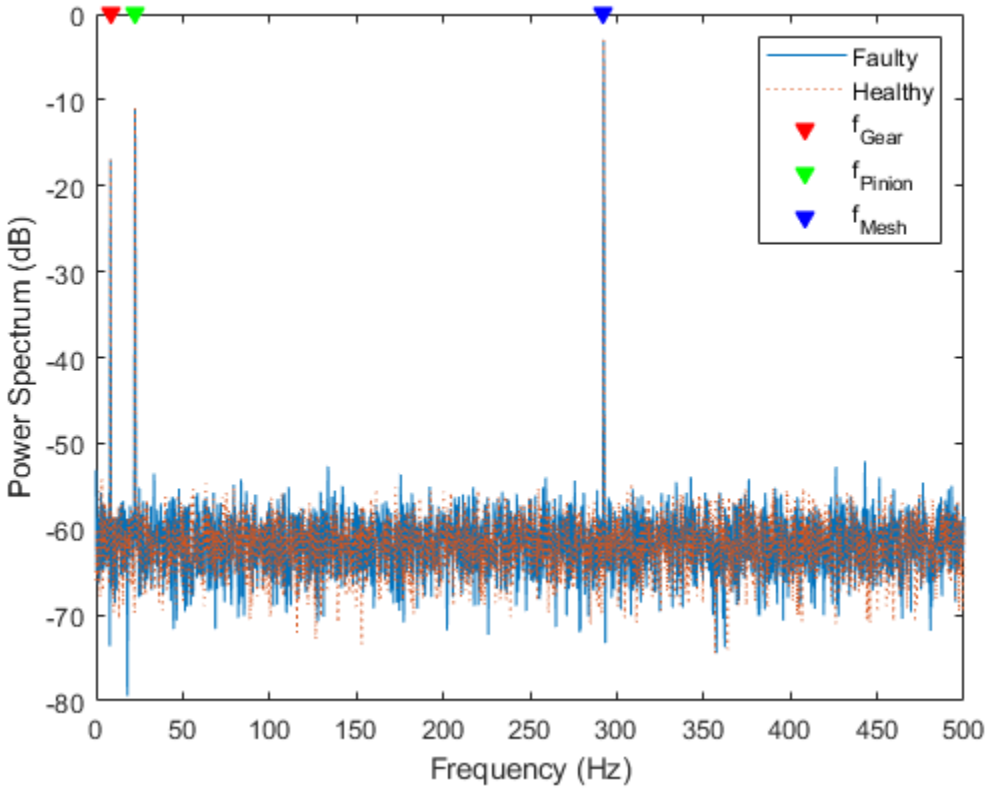
Calculate the spectrum of the healthy and faulty gears. Specify a frequency range that includes the shaft frequencies at 8.35 Hz and 22.5 Hz and the gear-mesh frequency at 292.5 Hz.

```
[Spect,f] = pspectrum([vFaultNoisy' vNoFaultNoisy'],fs,'FrequencyResolution',0.2,'FrequencyLimits
```

Plot the spectra. Because the fault is on the gear and not the pinion, sidebands are expected to appear at $f_{\text{sideband, Gear}}$ and spaced $f_{\text{Gear}}$ apart on the spectra. The spectra show the expected peaks at `fGear`, `fPin`, and `fMesh`. However, the presence of noise in the signal makes the sideband peaks at $f_{\text{sideband, Gear}}$ indistinguishable.

```
figure
plot(f,10*log10(Spect(:,1)),f,10*log10(Spect(:,2)),':')
xlabel('Frequency (Hz)')
ylabel('Power Spectrum (dB)')

hold on
plot(fGear,0,'rv','MarkerFaceColor','red')
```

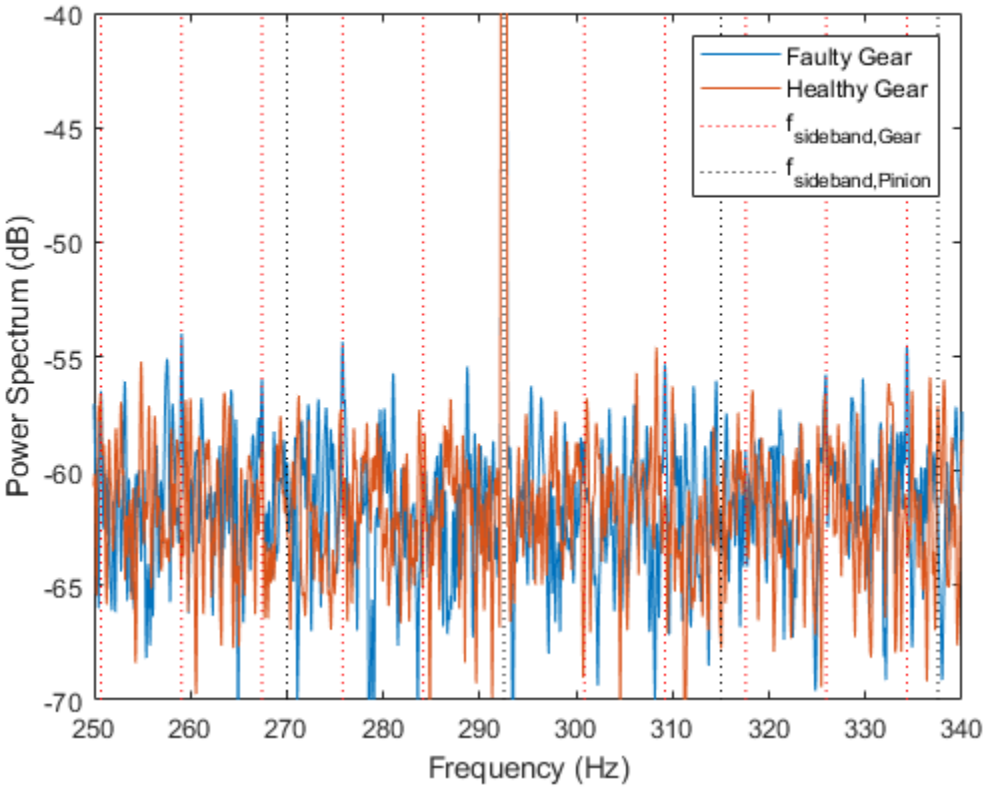
```
plot(fPin,0,'gv','MarkerFaceColor','green')
plot(fMesh,0,'bv','MarkerFaceColor','blue')
hold off

legend('Faulty','Healthy','f_{Gear}','f_{Pinion}','f_{Mesh}')
```




Zoom in on the neighborhood of the gear-mesh frequency. Create a grid of gear and pinion sidebands at $f_{\text{sideband, Gear}}$ and $f_{\text{sideband, Pinion}}$.

```
figure
p1 = plot(f,10*log10(Spect(:,1)));
xlabel('Frequency (Hz)')
ylabel('Power Spectrum (dB)')
xlim([250 340])
ylim([-70 -40])

hold on
p2 = plot(f,10*log10(Spect(:,2)));

harmonics = -5:5;
SBandsGear = (fMesh+fGear.*harmonics);
[X1,Y1] = meshgrid(SBandsGear,ylim);

SBandsPinion = (fMesh+fPin.*harmonics);
[X2,Y2] = meshgrid(SBandsPinion,ylim);

p3 = plot(X1,Y1,':r');
```

```
p4 = plot(X2,Y2,':k');
hold off
legend([p1 p2 p3(1) p4(1)],{'Faulty Gear';'Healthy Gear';'f_{sideband,Gear}';'f_{sideband,Pinion}
```




It is not clear if the peaks align with the gear sidebands $f_{\text{sideband, Gear}}$.

**Apply Time-Synchronous Averaging to the Output Vibration Signal**

Note that it is difficult to separate the peaks at the gear sidebands, $f_{\text{SideBand, Gear}}$, and the pinion sidebands, $f_{\text{SideBand, Pinion}}$. The previous section demonstrated difficulty in separating peaks and determining if the pinion or the gear is affected by faults. Time-synchronous averaging averages out zero-mean random noise and any waveforms not associated with frequencies of the particular shaft. This makes the process of fault detection easier.

Use the function `tsa` to generate time-synchronized waveforms for both the pinion and the gear.

Specify time-synchronized pulses for the pinion. Calculate the time-synchronous average for 10 rotations of the pinion.

```
tPulseIn = 0:1/fPin:max(t);
taPin = tsa(vFaultNoisy,fs,tPulseIn,'NumRotations',10);
```

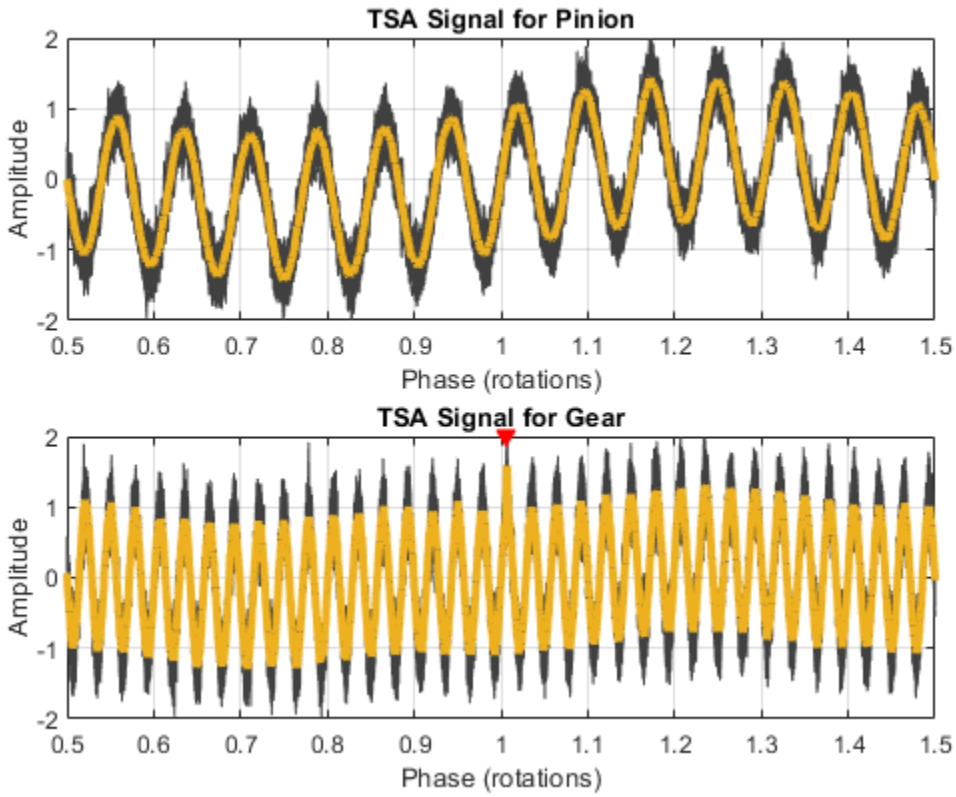
Specify time-synchronized pulses for the gear. Calculate the time-synchronous average for 10 rotations of the gear.

```
tPulseOut = 0:1/fGear:max(t);
taGear = tsa(vFaultNoisy,fs,tPulseOut,'NumRotations',10);
```

Visualize the time-synchronized signals for a single rotation. The impact is comparatively easier to see on the time-synchronous averaged signal for the gear, while it is averaged out for the pinion shaft. The location of the impact, indicated on the plot with a marker, has a higher amplitude than neighboring gear-mesh peaks.

The `tsa` function without output arguments plots the time-synchronous average signal and the time-domain signals corresponding to each signal segment in the current figure.

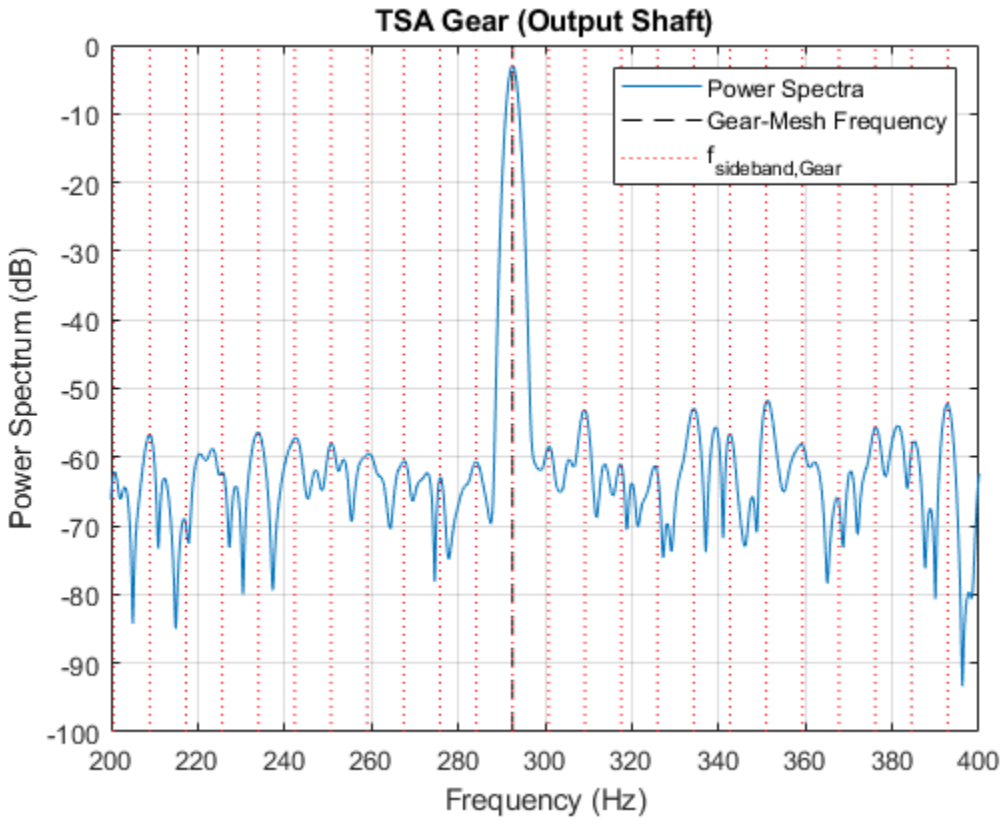
```
figure

subplot(2,1,1)
tsa(vFaultNoisy,fs,tPulseIn,'NumRotations',10)
xlim([0.5 1.5])
ylim([-2 2])
title('TSA Signal for Pinion')

subplot(2,1,2)
tsa(vFaultNoisy,fs,tPulseOut,'NumRotations',10)
xlim([0.5 1.5])
ylim([-2 2])
title('TSA Signal for Gear')
hold on
plot(1.006,2,'rv','MarkerFaceColor','red')
hold off
```



### Visualize the Power Spectra for Time-Synchronous Averaged Signals

Calculate the power spectrum of the time-synchronous averaged gear signal. Specify a frequency range that covers 15 gear sidebands on either side of the gear mesh frequency of 292.5 Hz. Notice the peaks at $f_{\text{sideband, Gear}}$.

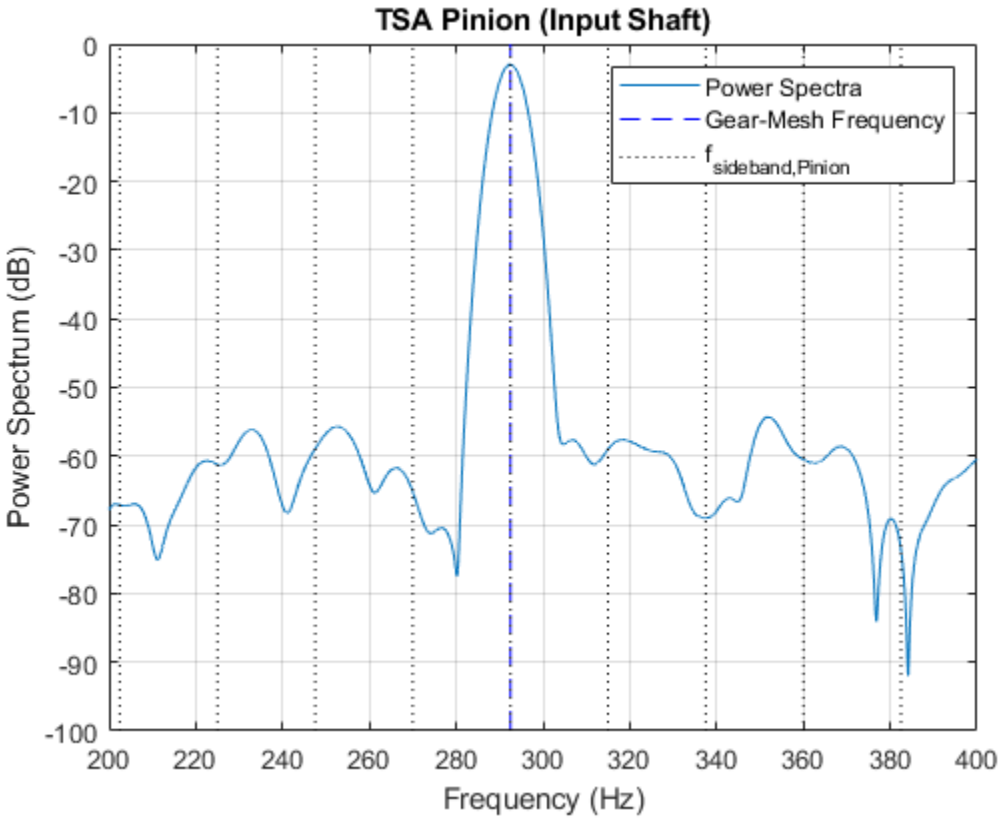
```
figure
pspectrum(taGear,fs,'FrequencyResolution',2.2,'FrequencyLimits',[200 400])

harmonics = -15:15;
SBandsGear=(fMesh+fGear.*harmonics);

[X1,Y1] = meshgrid(SBandsGear,ylim);
[XM,YM] = meshgrid(fMesh,ylim);

hold on
plot(XM,YM,'--k',X1,Y1,':r')
legend('Power Spectra','Gear-Mesh Frequency','f_{sideband,Gear}')
hold off

title('TSA Gear (Output Shaft)')
```




Visualize the power spectra of the time-synchronous averaged pinion signal in the same frequency range. This time, plot grid lines at $f_{\text{sideband, Pinion}}$ frequency locations.

```
figure
pspectrum(taPin,fs,'FrequencyResolution',5.8,'FrequencyLimits',[200 400])
```

```
SBandsPinion = (fMesh+fPin.*harmonics);

[X2,Y2] = meshgrid(SBandsPinion,ylim);
[XM,YM] = meshgrid(fMesh,ylim);

hold on
plot(XM,YM,'--b',X2,Y2,':k')
legend('Power Spectra','Gear-Mesh Frequency','f_{sideband,Pinion}')
hold off

title('TSA Pinion (Input Shaft)')
```



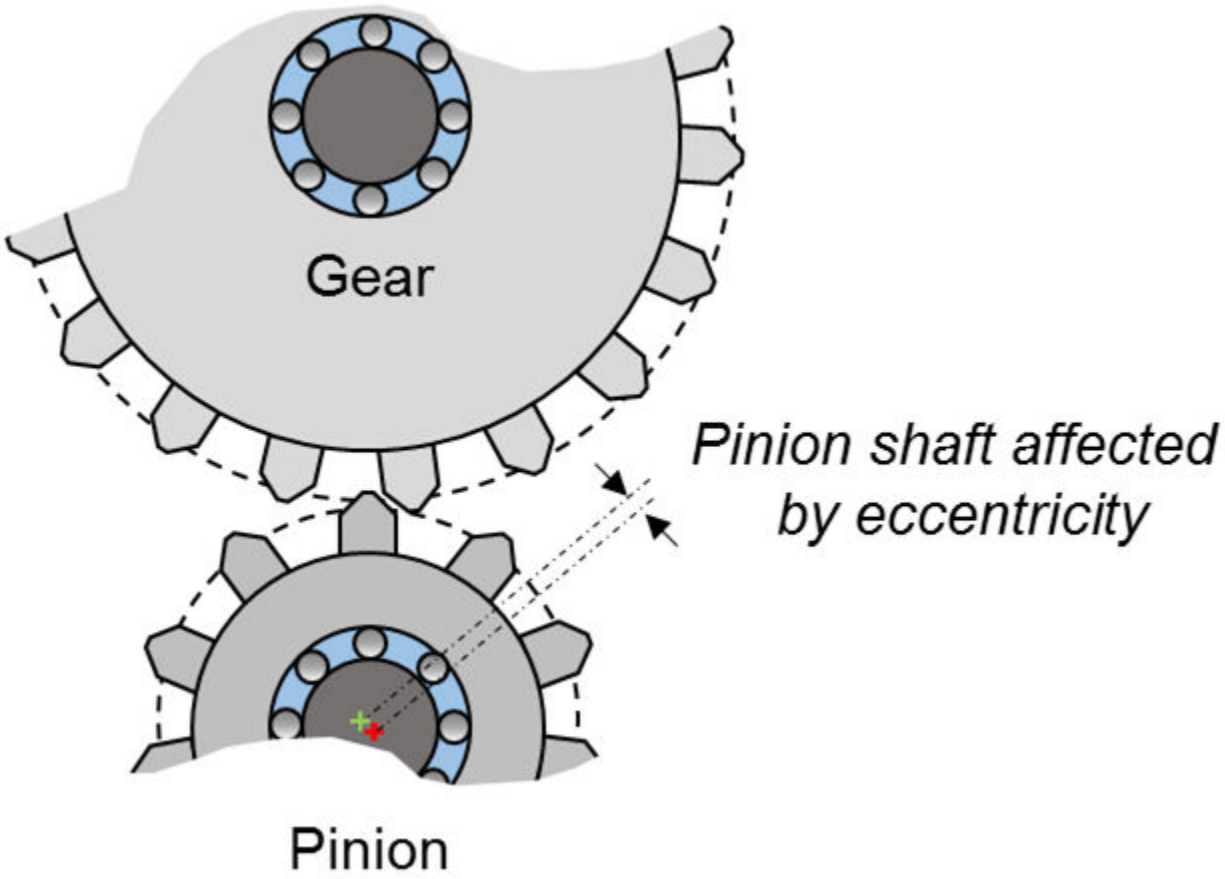

Notice the absence of prominent peaks at $f_{\text{sideband, Pinion}}$ in the plot.

The power spectra of the original signal contains waveforms from two different shafts, as well as noise. It is difficult to distinguish the sideband harmonics. However, observe the prominent peaks at the sideband locations on the spectrum of the time-synchronous averaged gear signal. Also observe the nonuniformity in sideband magnitudes, which are an indicator of localized faults on the gear. On the other hand, sideband peaks are absent from the spectrum of the time-synchronous averaged pinion signal. This helps us conclude that the pinion is potentially healthy.

By averaging out the waveforms that are not relevant, the `tsa` function helps identify the faulty gear by looking at sideband harmonics. This functionality is especially useful when it is desirable to extract a vibration signal corresponding to a single shaft, from a gearbox with multiple shafts and gears.

### Add a Distributed Fault in the Pinion and Incorporate its Effects into the Vibration Signal

A distributed gear fault, such as eccentricity or gear misalignment [1], causes higher-level sidebands that are narrowly grouped around integer multiples of the gear-mesh frequency.




To simulate a distributed fault, introduce three sideband components of decreasing amplitude on either side of the gear-mesh frequency.

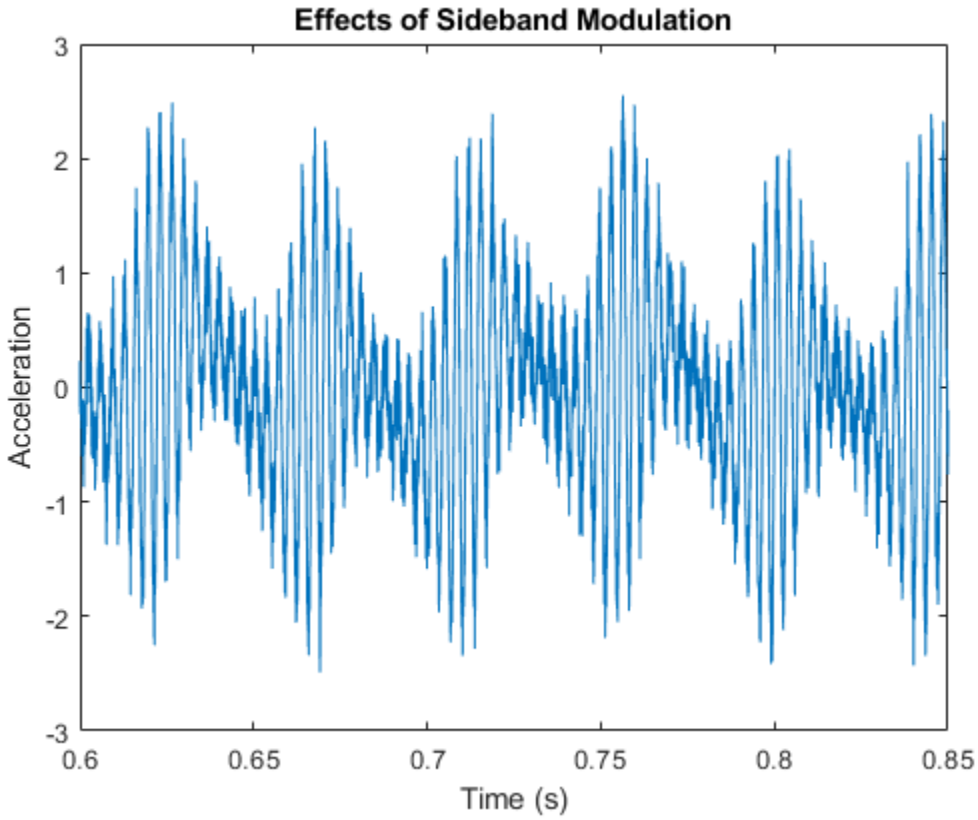
```
SideBands = -3:3;
SideBandAmp = [0.02 0.1 0.4 0 0.4 0.1 0.02];     % Sideband amplitudes
SideBandFreq = fMesh + SideBands*fPin;           % Sideband frequencies

vSideBands = SideBandAmp*sin(2*pi*SideBandFreq'.*t);
```

Add the sideband signals to the vibration signal. This results in amplitude modulation.

```
vPinFaultNoisy = vFaultNoisy + vSideBands;
```

Visualize a section of the time history for the gearbox affected by the distributed fault.

```
plot(t,vPinFaultNoisy)
xlim([0.6 0.85])
xlabel('Time (s)')
ylabel('Acceleration')
title('Effects of Sideband Modulation')
```

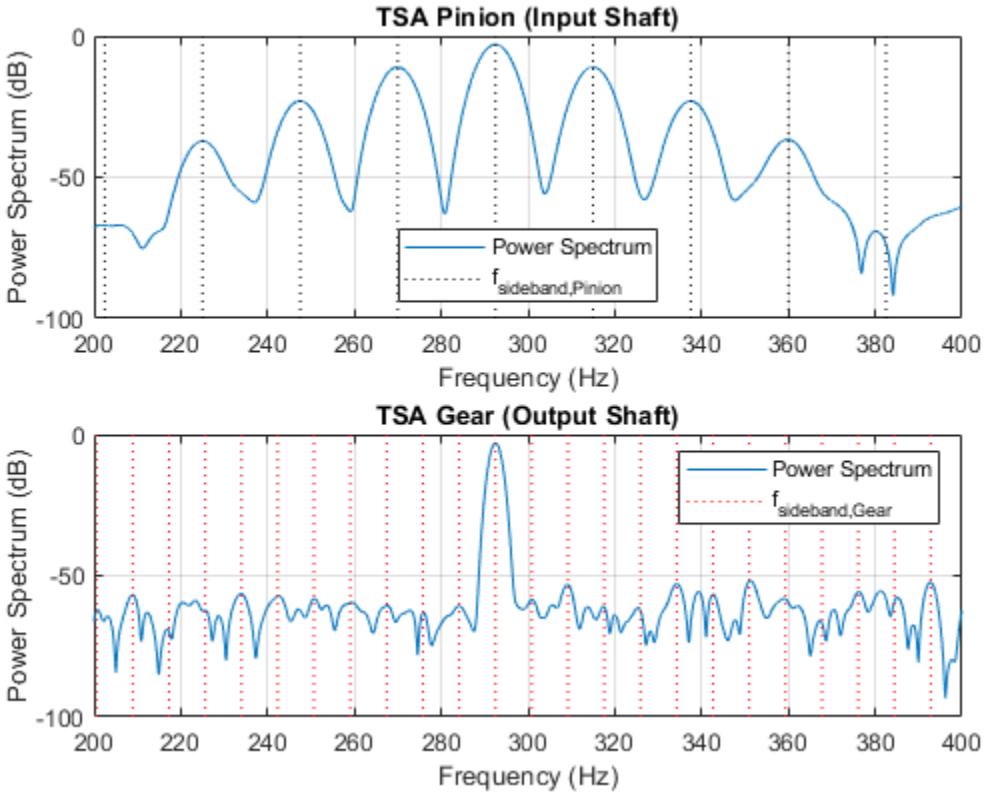



Recompute the time-synchronous averaged signal for the pinion and the gear.

```
taPin = tsa(vPinFaultNoisy,fs,tPulseIn,'NumRotations',10);
taGear = tsa(vFaultNoisy,fs,tPulseOut,'NumRotations',10);
```

Visualize the power spectrum of the time-synchronous averaged signal. The three sidebands in the time-synchronous averaged signal of the pinion are more pronounced which indicate the presence of distributed faults. However, the spectrum of the time-synchronous averaged gear signal remains unchanged.

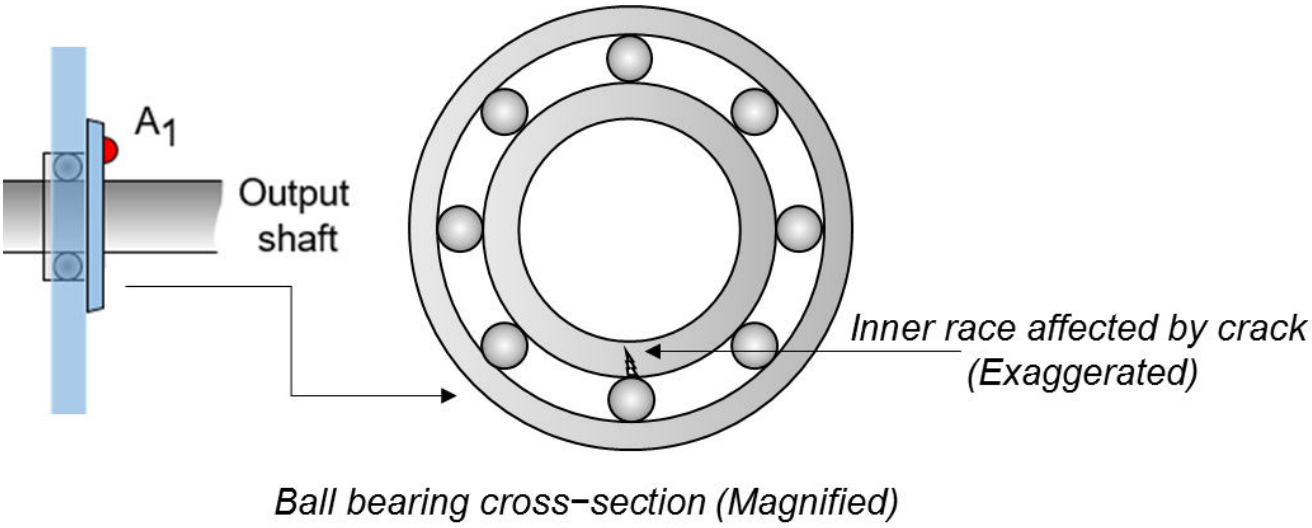
```
subplot(2,1,1)
pspectrum(taPin,fs,'FrequencyResolution',5.8,'FrequencyLimits',[200 400])
hold on
plot(X2,Y2,':k')
legend('Power Spectrum','f_{sideband,Pinion}','Location','south')
hold off
title ('TSA Pinion (Input Shaft)')


subplot(2,1,2)
pspectrum(taGear,fs,'FrequencyResolution',2.2,'FrequencyLimits',[200 400])
hold on
plot(X1,Y1,':r')
legend('Power Spectrum','f_{sideband,Gear}')
hold off
title ('TSA Gear (Output Shaft)')
```




In conclusion, the `tsa` function helps extract the gear and pinion contributions from the overall vibration signal. This in turn helps identify the specific components that are affected by localized and distributed faults.

### Vibration Analysis of Rolling Element Bearing Faults

Localized faults in a rolling element bearing may occur in the outer race, the inner race, the cage, or a rolling element. Each of these faults is characterized by its own frequency, which is usually listed by the manufacturer or calculated from the bearing specifications. An impact from a localized fault generates high-frequency vibrations in the gearbox structure between the bearing and response transducer [2]. Assume that the gears in the gearbox are healthy and that one of the bearings supporting the pinion shaft is affected by a localized fault in the inner race. Neglect the effects of radial load in the analysis.




The bearing, with a pitch diameter of 12 cm, has eight rolling elements. Each rolling element has a diameter of 2 cm. The angle of contact $\theta$ is 15˚. It is common practice to place the accelerometer on a bearing-housing while analyzing bearing vibration. Acceleration measurements are recorded by$A_1$, an accelerometer located on the faulty bearing housing.

Define the parameters for the bearing.

```
n = 8;          % Number of rolling element bearings
d = 0.02;       % Diameter of rolling elements
p = 0.12;       % Pitch diameter of bearing
thetaDeg = 15; % Contact angle in degrees
```

The impacts occur whenever a rolling element passes the localized fault on the inner race. The rate at which this happens is the ball pass frequency-inner race (BPFI). The BPFI can be calculated using

$$f_{\mathrm{BPFI}} = \frac{n \times f_{\mathrm{Pin}}}{2}\left(1 + \frac{d}{p}\cos\theta\right).$$

```
bpfi = n*fPin/2*(1 + d/p*cosd(thetaDeg))
```

```
bpfi = 104.4889
```

Model each impact as a 3 kHz sinusoid windowed by a Kaiser window. The defect causes a series of 5-millisecond impacts on the bearing. Impulses in the early stages of pits and spalls cover a wide frequency range up to about 100 kHz [2].

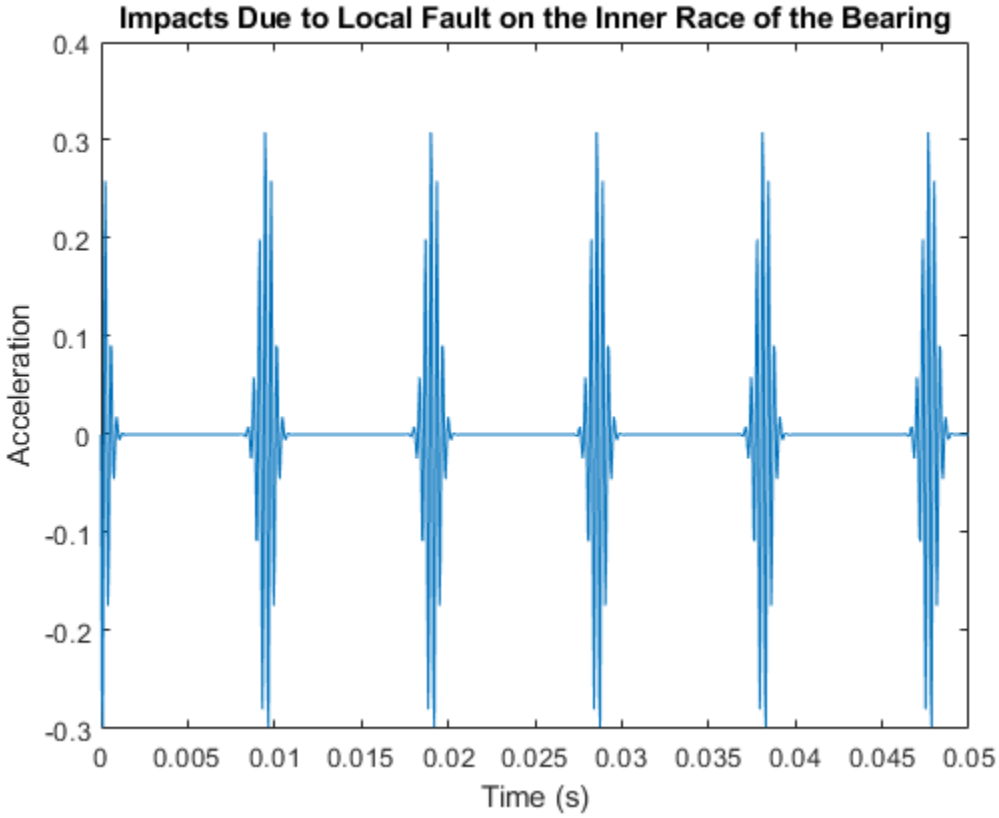
```
fImpact = 3000;
tImpact = 0:1/fs:5e-3-1/fs;
xImpact = sin(2*pi*fImpact*tImpact).*kaiser(length(tImpact),40)';
```

Make the impact periodic by convolving it with a comb function. Since $A_1$ is closer to the bearing, adjust the amplitude of the impact such that it is prominent with respect to the gearbox vibration signal recorded by $A_2$.

```
xComb = zeros(size(t));
xComb(1:round(fs/bpfi):end) = 1;
xBper = 0.33*conv(xComb,xImpact,'same');
```

Visualize the impact signal.

```
figure
plot(t,xBper)
xlim([0 0.05])
xlabel('Time (s)')
ylabel('Acceleration')
title('Impacts Due to Local Fault on the Inner Race of the Bearing')
```



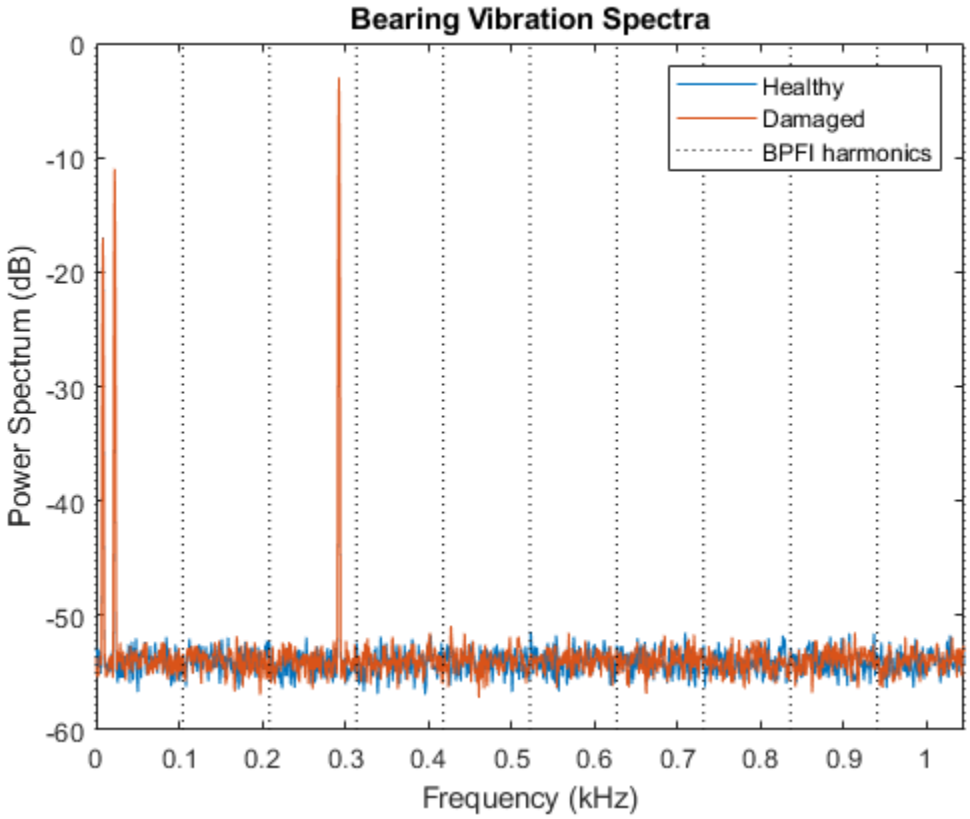

Add the periodic bearing fault to the vibration signal from the healthy gearbox.

```
vNoBFaultNoisy = vNoFault + randn(size(t))/5;
vBFaultNoisy = xBper + vNoFault + randn(size(t))/5;
```

Compute the spectra of the signals. Visualize the spectrum at lower frequencies. Create a grid of the first ten BPFI harmonics.

```
pspectrum([vBFaultNoisy' vNoBFaultNoisy' ],fs,'FrequencyResolution',1,'FrequencyLimits',[0 10*bp
legend('Damaged','Healthy')
title('Bearing Vibration Spectra')
grid off

harmImpact = (0:10)*bpfi;
[X,Y] = meshgrid(harmImpact,ylim);

hold on
plot(X/1000,Y,':k')
legend('Healthy','Damaged','BPFI harmonics')
```

```
xlim([0 10*bpfi]/1000)
hold off
```




At the lower end of the spectrum, the shaft and mesh frequencies and their orders obscure other features. The spectrum of the healthy bearing and the spectrum of the damaged bearing are indistinguishable. This flaw highlights the necessity for an approach that can isolate bearing faults.

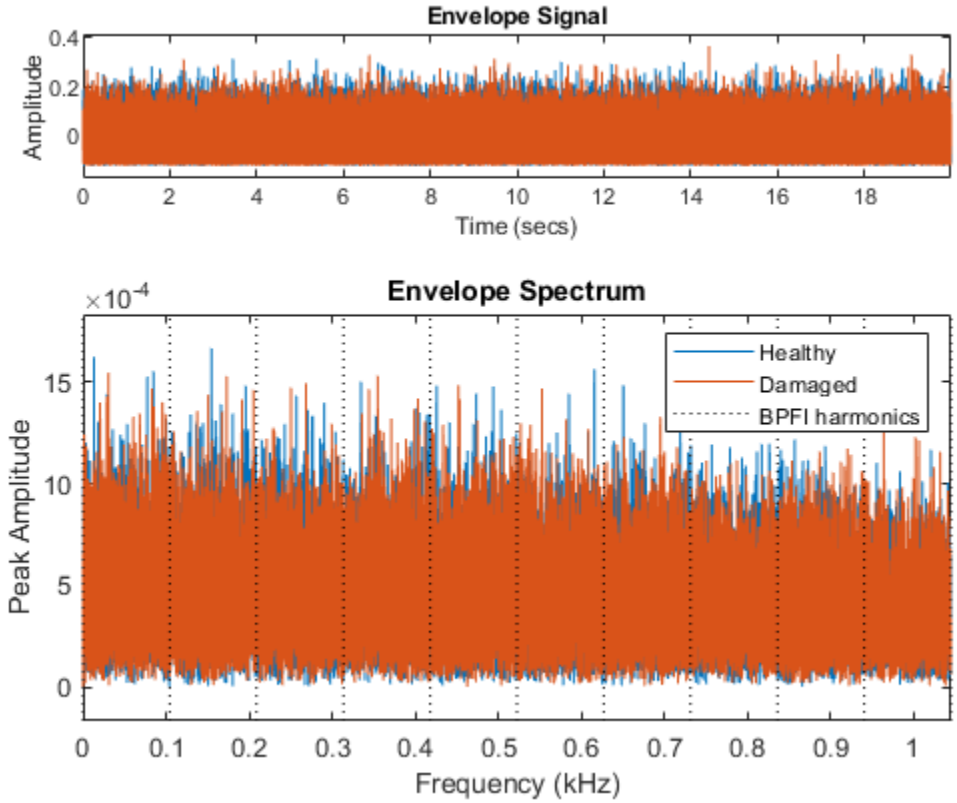
BPFI is dependent on the ratio $d/p$ and the cosine of the contact angle $\theta$. An irrational expression for BPFI implies that bearing impacts are not synchronous with an integer number of shaft rotations. The `tsa` function is not useful in this case because it averages out the impacts. The impacts do not lie on the same location in every averaged segment.

The function `envspectrum` (envelope spectrum) performs amplitude demodulation, and is useful in extracting information about high-frequency impacts.

Compute and plot the envelope signals and their spectra. Compare the envelope spectra for the signals with and without the bearing fault. Visualize the spectrum at lower frequencies. Create a grid of the first ten BPFI harmonics.

```
figure
envspectrum([vNoBFaultNoisy' vBFaultNoisy'],fs)
xlim([0 10*bpfi]/1000)
[X,Y] = meshgrid(harmImpact,ylim);

hold on
plot(X/1000,Y,':k')
legend('Healthy','Damaged','BPFI harmonics')
```

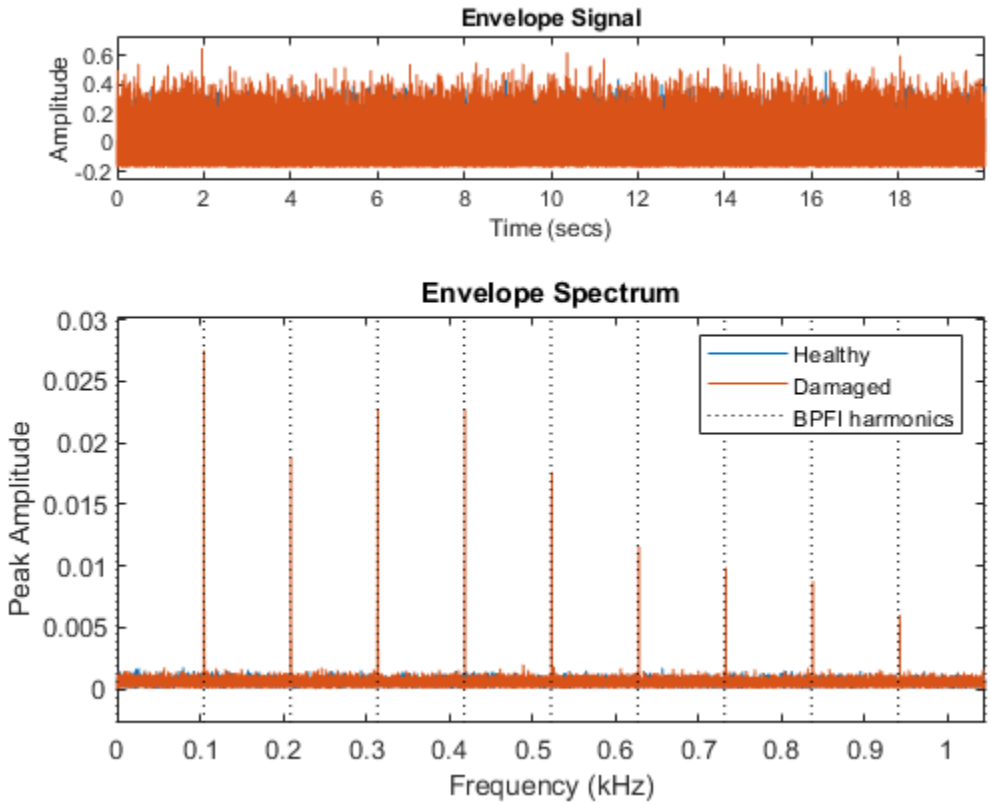
```
xlim([0 10*bpfi]/1000)
hold off
```




Observe that BPFI peaks are not prominent in the envelope spectrum because the signal is polluted by noise. Recall that performing `tsa` to average out noise is not useful for bearing-fault analysis because it also averages out the impact signals.

The `envspectrum` function offers a built-in filter that can be used to remove noise outside the band of interest. Apply a bandpass filter of order 200 centered at 3.125 kHz and 4.167 kHz wide.

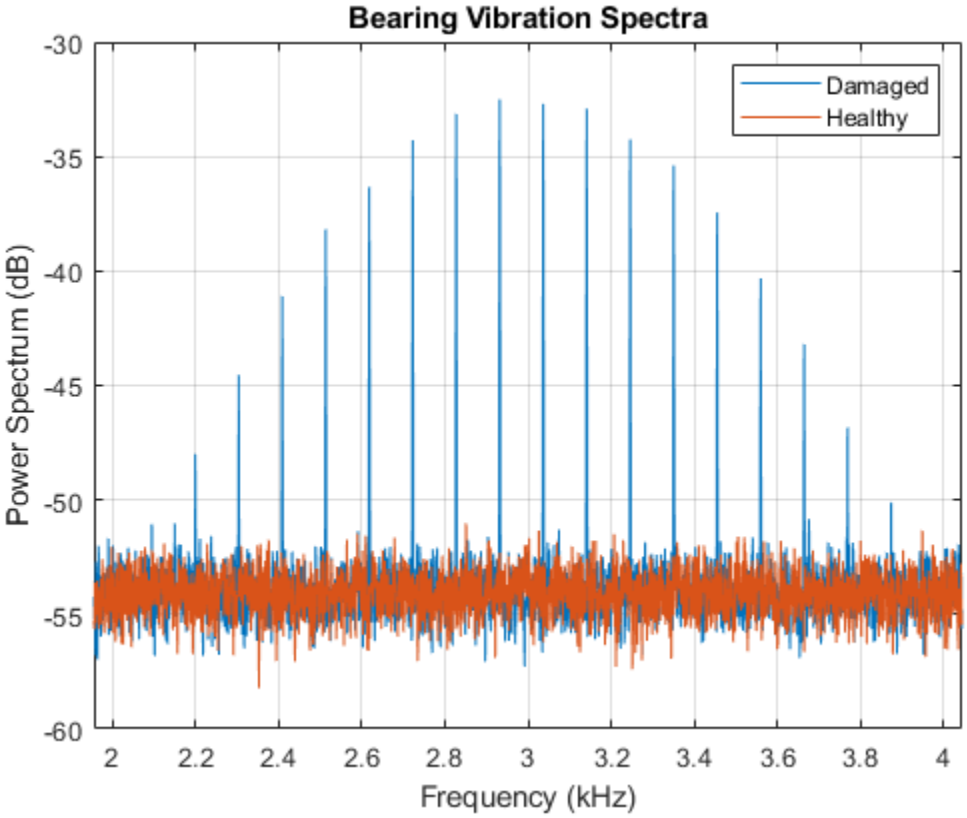
```
Fc = 3125;
BW = 4167;

envspectrum([vNoBFaultNoisy' vBFaultNoisy'],fs,'Method','hilbert','FilterOrder',200,'Band',[Fc-BW

harmImpact = (0:10)*bpfi;
[X,Y] = meshgrid(harmImpact,ylim);

hold on
plot(X/1000,Y,':k')
legend('Healthy','Damaged','BPFI harmonics')
xlim([0 10*bpfi]/1000)
hold off
```




The envelope spectrum effectively brings in the passband content to baseband, and therefore shows the presence of prominent peaks at the BPFI harmonics below 1 kHz. This helps conclude that the inner race of the bearing is potentially damaged.

In this case, the frequency spectrum of the faulty bearing clearly shows BPFI harmonics modulated by the impact frequency. Visualize this phenomenon in the spectra, close to the impact frequency of 3 kHz.

```
figure
pspectrum([vBFaultNoisy' vNoBFaultNoisy'],fs,'FrequencyResolution',1,'FrequencyLimits',(bpfi*[-1(
legend('Damaged','Healthy')
title('Bearing Vibration Spectra')
```




Observe that the separation in frequency between peaks is equal to BPFI.

### Conclusions

This example used time-synchronous averaging to separate vibration signals associated with both a pinion and a gear. In addition, `tsa` also attenuated random noise. In cases of fluctuating speed (and load [2]), order tracking can be used as a precursor to `tsa` to resample the signal in terms of shaft rotation angle. Time-synchronous averaging is also used in experimental conditions to attenuate the effects of small changes in shaft speed.

Broadband frequency analysis may be used as a good starting point in the fault analysis of bearings [3]. However, its usefulness is limited when the spectra in the neighborhood of bearing impact frequencies contain contributions from other components, such as higher harmonics of gear-mesh frequencies in a gearbox. Envelope analysis is useful under such circumstances. The function `envspectrum` can be used to extract envelope signals and spectra for faulty bearings, as an indicator of bearing wear and damage.

### References

1 Scheffer, Cornelius, and Paresh Girdhar. *Practical Machinery Vibration Analysis and Predictive Maintenance*. Amsterdam: Elsevier, 2004.

2 Randall, Robert Bond. *Vibration Based Condition Monitoring: Industrial, Aerospace and Automotive Applications*. Chichester, UK: John Wiley and Sons, 2011.

3 Lacey, S. J. *An Overview of Bearing Vibration Analysis*. (From: http://www.maintenanceonline.co.uk/maintenanceonline/content_images/p32-42%20Lacey%20paper%20M&AM.pdf)

**4** Brandt, Anders. *Noise and Vibration Analysis: Signal Analysis and Experimental Procedures*. Chichester, UK: John Wiley and Sons, 2011.

## See Also

`envspectrum` | `pspectrum` | `tsa`

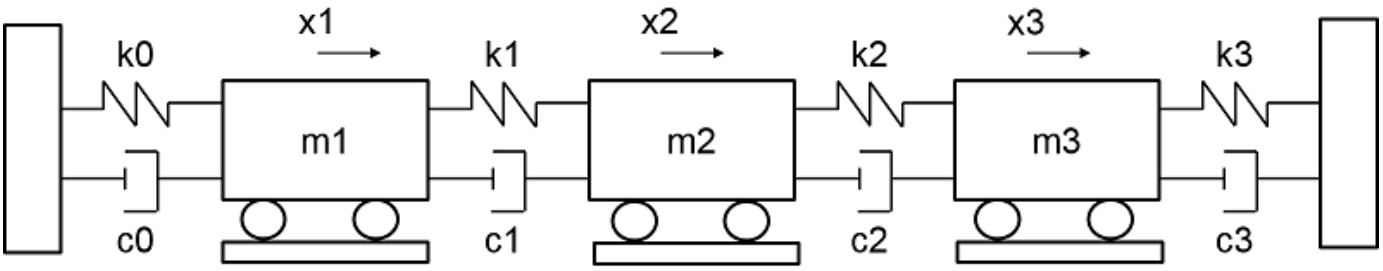
# Modal Analysis of a Simulated System and a Wind Turbine Blade

This example shows how to estimate frequency-response functions (FRFs) and modal parameters from experimental data. The first section describes a simulated experiment that excites a three-degree-of-freedom (3DOF) system with a sequence of hammer impacts and records the resulting displacement. Frequency-response functions, natural frequencies, damping ratios, and mode shape vectors are estimated for three modes of the structure. The second section estimates mode shape vectors from frequency-response function estimates from a wind turbine blade experiment. The turbine blade measurement configuration and resulting mode shapes are visualized. This example requires System Identification Toolbox (TM).

### Natural Frequency and Damping for a Simulated Beam

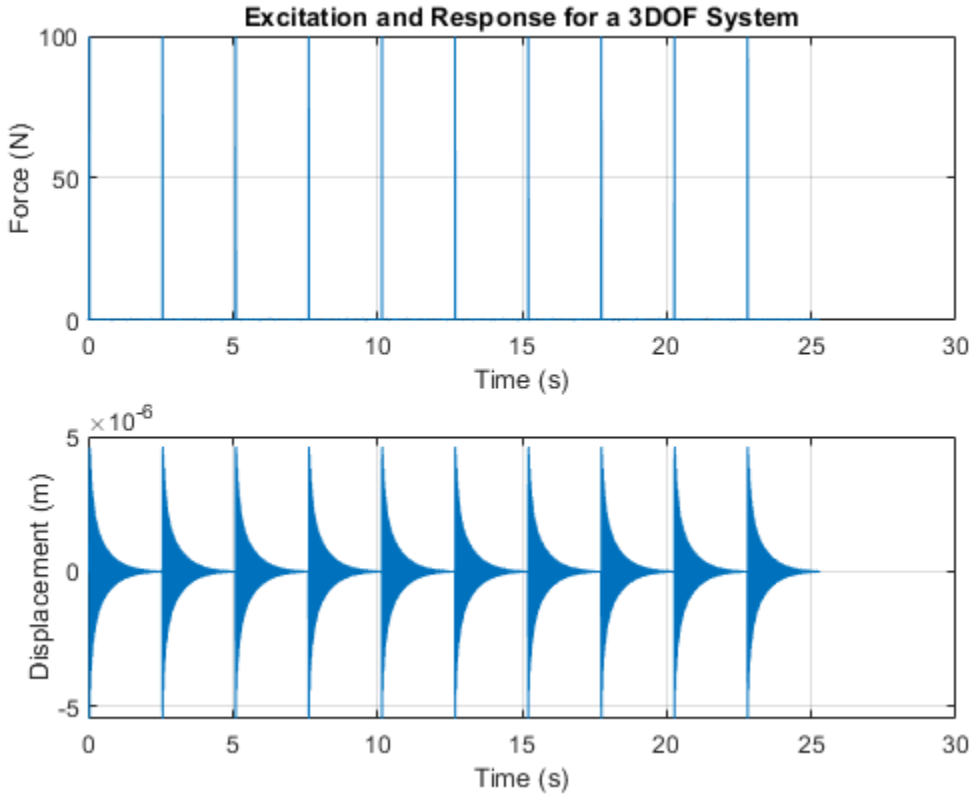
#### Single-Input/Single-Output Hammer Excitation

A series of hammer strikes excite a 3DOF system, and sensors record the resulting displacements. The system is proportionally damped, such that the damping matrix is a linear combination of the mass and stiffness matrices.



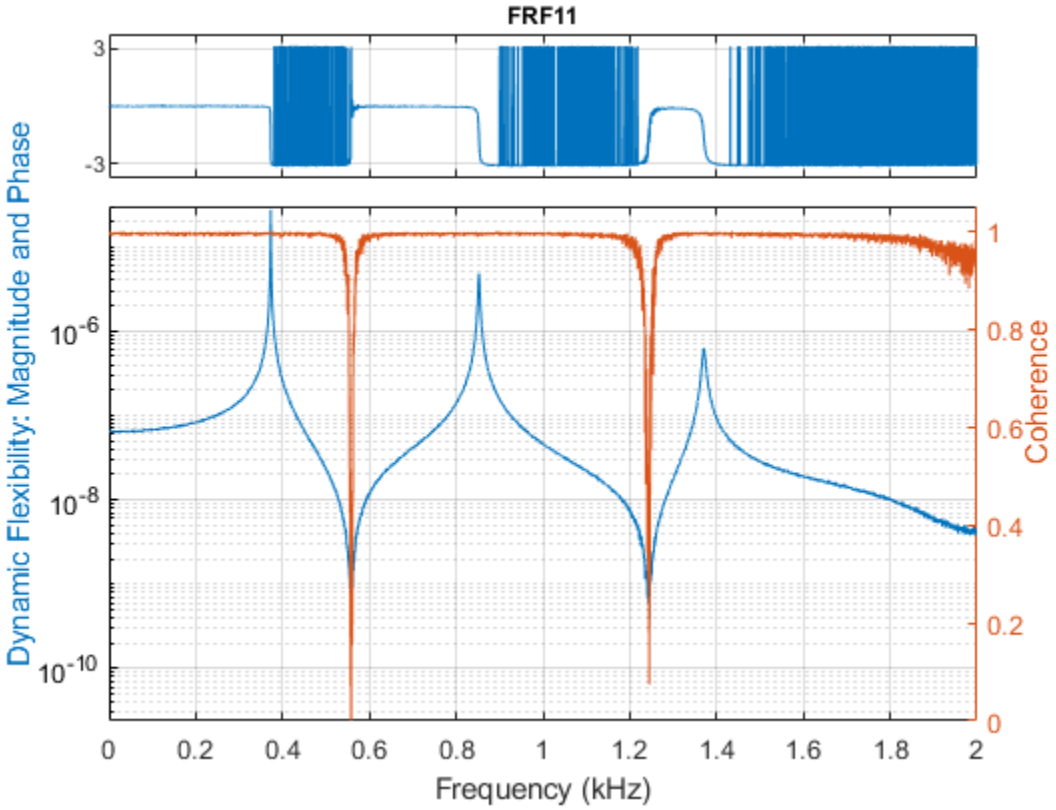

Import the data for two sets of measurements, including excitation signals, response signals, time signals, and ground truth frequency-response functions. The first set of response signals, `Y1`, measures the displacement of the first mass, and the second, `Y2`, measures the second mass. Each excitation signal consists of ten concatenated hammer impacts, and each response signal contains the corresponding displacement. The duration for each impact signal is 2.53 seconds. Additive noise is present in the excitation and response signals. Visualize the first excitation and response channel of the first measurement.

```
[t,fs,X1,X2,Y1,Y2,f0,H0] = helperImportModalData();
X0 = X1(:,1);
Y0 = Y1(:,1);
helperPlotModalAnalysisExample([t' X0 Y0]);
```




Compute and plot the FRF for the first excitation and response channels in terms of dynamic flexibility, which is a measure of displacement over force [1]. By default, the FRF is computed by averaging spectra of windowed segments. Since each hammer excitation decays substantially before the next excitation, a rectangular window can be used. Specify the sensor as displacement.

```
winLen = 2.5275*fs; % window length in samples
modalfrf(X0,Y0,fs,winLen,'Sensor','dis')
```




The FRF, estimated using the default `'H1'` estimator, contains three prominent peaks in the measured frequency band, corresponding to three flexible modes of vibration. The coherence is close to one near these peaks, and low in anti-resonance regions, where the signal-to-noise ratio of the response measurement is low. Coherence near to one indicates a high quality estimate. The `'H1'` estimate is optimal where noise exists only at the output measurement, whereas the `'H2'` estimator is optimal when there is additive noise only on the input [2]. Compute the `'H1'` and `'H2'` estimates for this FRF.

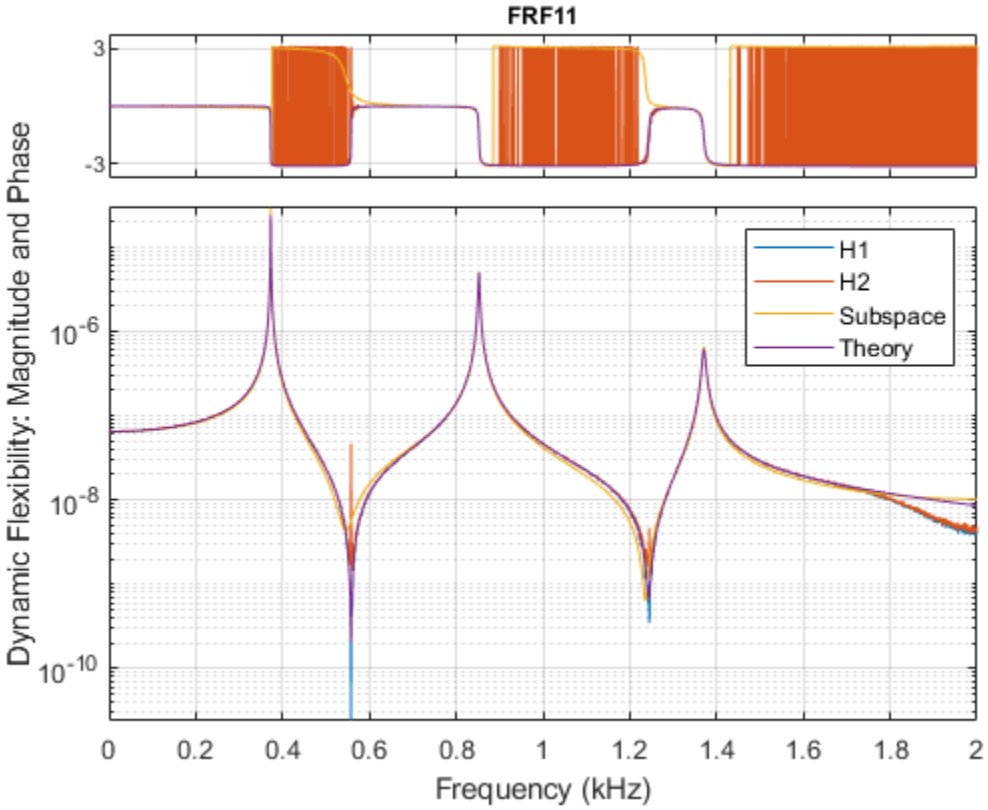
```
[FRF1,f1] = modalfrf(X0,Y0,fs,winLen,'Sensor','dis');  % Calculate FRF (H1)
[FRF2,f2] = modalfrf(X0,Y0,fs,winLen,'Sensor','dis','Estimator','H2');
```

When there is significant measurement noise or the excitation is poor, parametric methods can offer additional options for accurately extracting the FRF from the data. The 'subspace' method first fits a state-space model to the data [3] and then computes its frequency-response function. The order of the state-space model (equal to the number of poles) and presence or lack of feedthrough can be specified to configure the state-space estimation.

```
[FRF3,f3] = modalfrf(X0,Y0,fs,winLen,'Sensor','dis','Estimator','subspace','Feedthrough',true);
```

Here FRF3 is estimated by fitting a state-space model containing a feedthrough term and of the optimal order in the range 1:10. Compare the estimated FRFs using `'H1'`, `'H2'` and `'subspace'` methods to the theoretical FRF.

```
helperPlotModalAnalysisExample(f1,FRF1,f2,FRF2,f3,FRF3,f0,H0);
```




The estimators perform comparably near response peaks, while the `'H2'` estimator overestimates the response at the antiresonances. The coherence is not affected by the choice of the estimator.

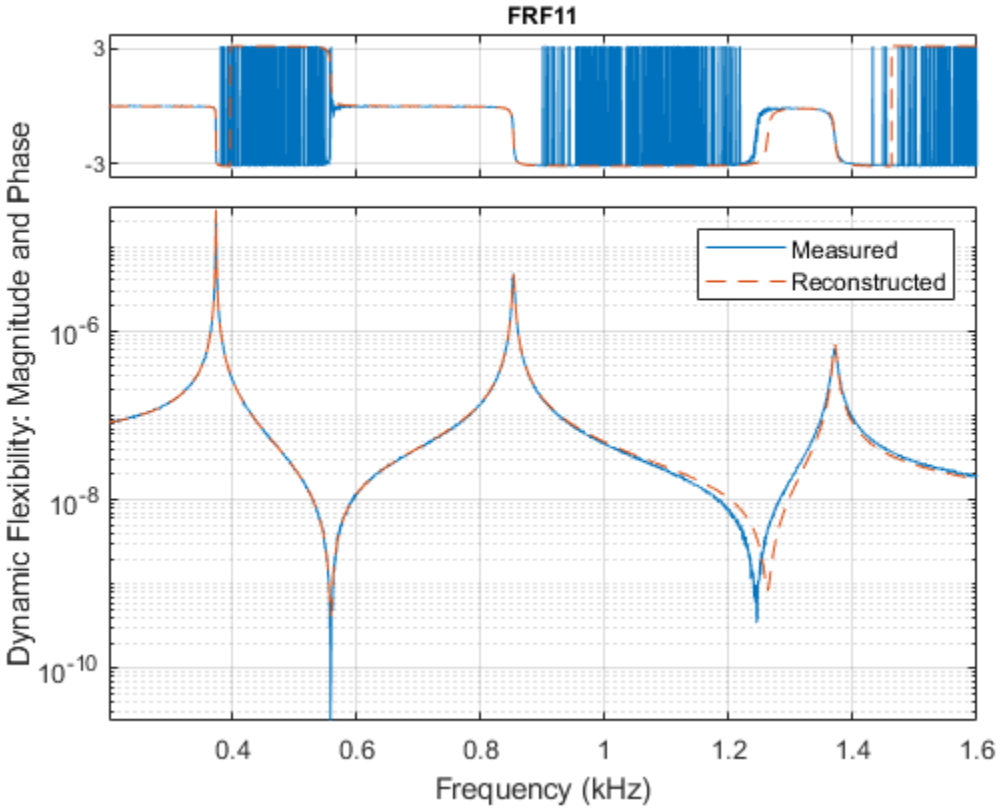
Next, estimate the natural frequency of each mode using the peak-picking algorithm. The peak-picking algorithm is a simple and fast procedure for identifying peaks in the FRF. It is a local method, since each estimate is generated from a single frequency-response function. It is also a single-degree-of-freedom (SDOF) method, since the peak for each mode is considered independently. As a result, a set of modal parameters is generated for each FRF. Based on the previous plot, specify a frequency range from 200 to 1600 Hz, which contains the three peaks.

```
fn = modalfit(FRF1,f1,fs,3,'FitMethod','PP','FreqRange',[200 1600])
```

```
fn =

   1.0e+03 *

    0.3727
    0.8525
    1.3707
```

The natural frequencies are approximately 373, 853, and 1371 Hz. Plot a reconstructed FRF and compare it to the measured data using `modalfit`. The FRF is reconstructed using the modal parameters estimated from the frequency-response function matrix, `FRF1`. Call `modalfit` again with no output arguments to produce a plot containing the reconstructed FRF.

```
modalfit(FRF1,f1,fs,3,'FitMethod','PP','FreqRange',[200 1600])
```

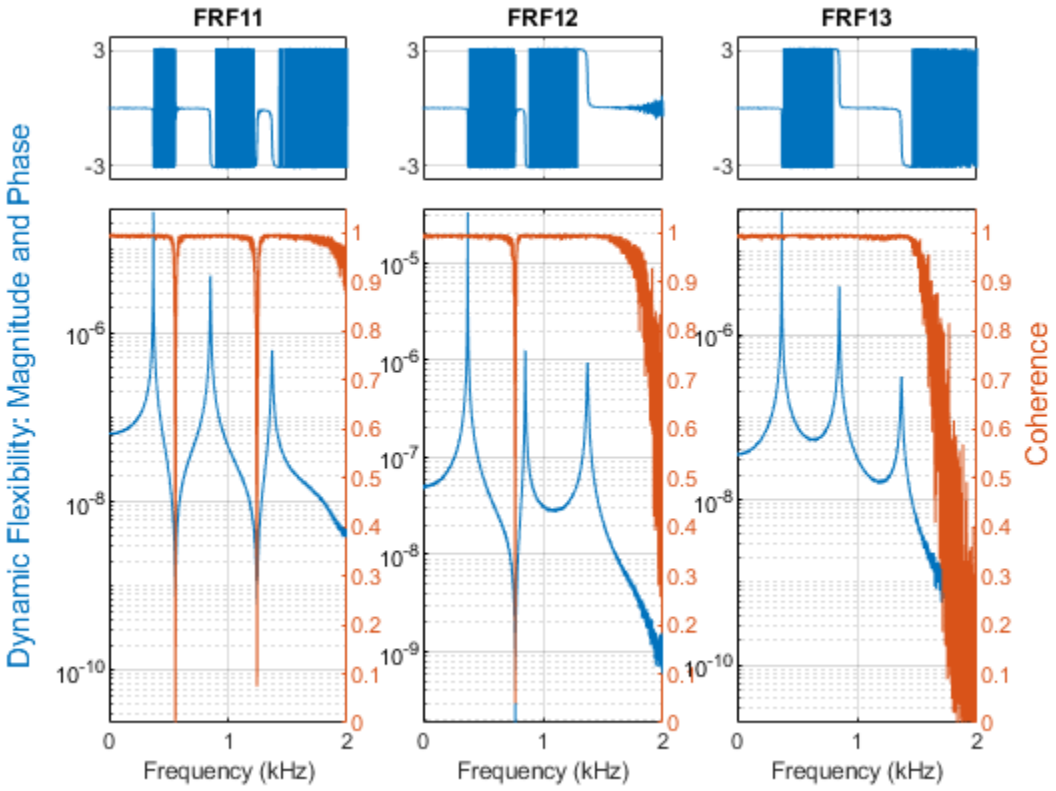



The reconstructed FRF agrees with the measured FRF of the first excitation and response channels. In the next section, two additional excitation locations are considered.

**Roving Hammer Excitation**

Compute and plot the FRFs for the responses of all three sensors using the default `'H1'` estimator. Specify the measurement type as `'rovinginput'` since we have a roving hammer excitation.

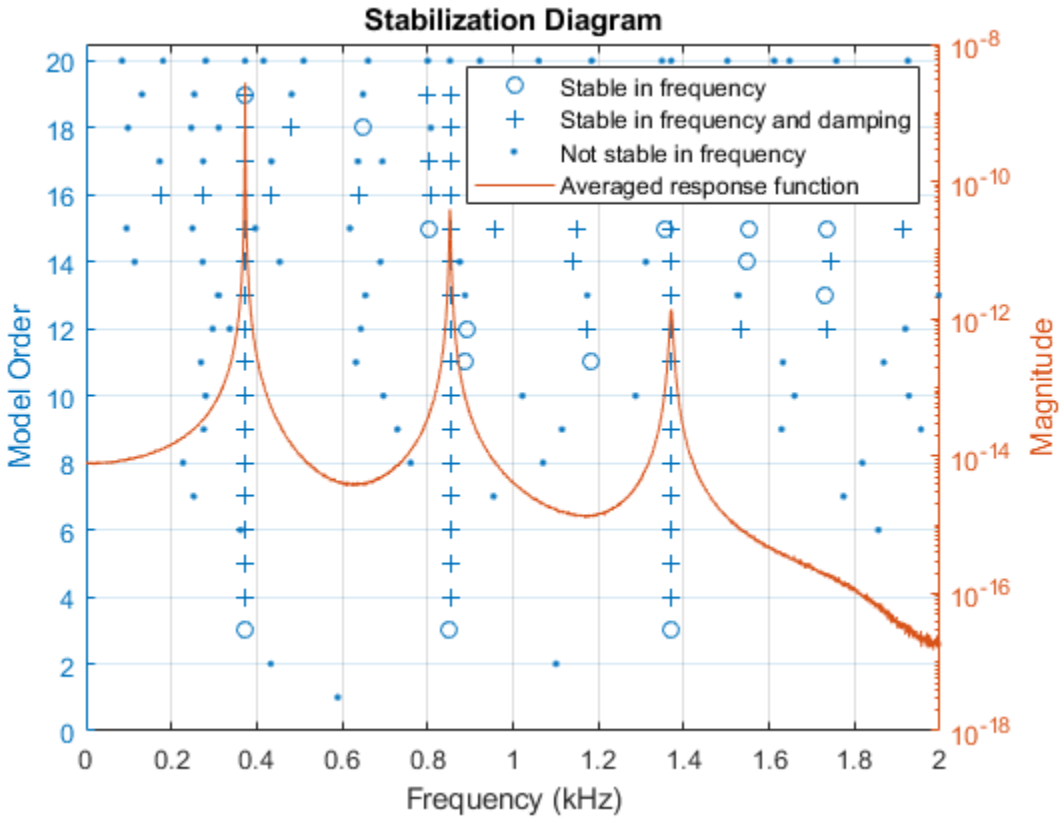
```
modalfrf(X1,Y1,fs,winLen,'Sensor','dis','Measurement','rovinginput')
```




In the previous section, a single set of modal parameters was computed from a single FRF. Now, estimate modal parameters using the least-squares complex exponential (LSCE) algorithm. The LSCE and LSRF algorithms generate a single set of modal parameters by analyzing multiple response signals simultaneously. These are global, multiple-degree-of-freedom (MDOF) methods, since the parameters for all modes are estimated simultaneously from multiple frequency-response functions.

The LSCE algorithm generates computational modes, which are not physically present in the structure. Use a stabilization diagram to identify physical modes by examining the stability of poles as the number of modes increases. The natural frequencies and damping ratios of physical modes tend to remain in the same place, or are 'stable'. Create a stabilization diagram and output the natural frequencies of those poles which are stable in frequency.

```
[FRF,f] = modalfrf(X1,Y1,fs,winLen,'Sensor','dis','Measurement','rovinginput');
fn = modalsd(FRF,f,fs,'MaxModes',20, 'FitMethod', 'lsce'); % Identify physical modes
```

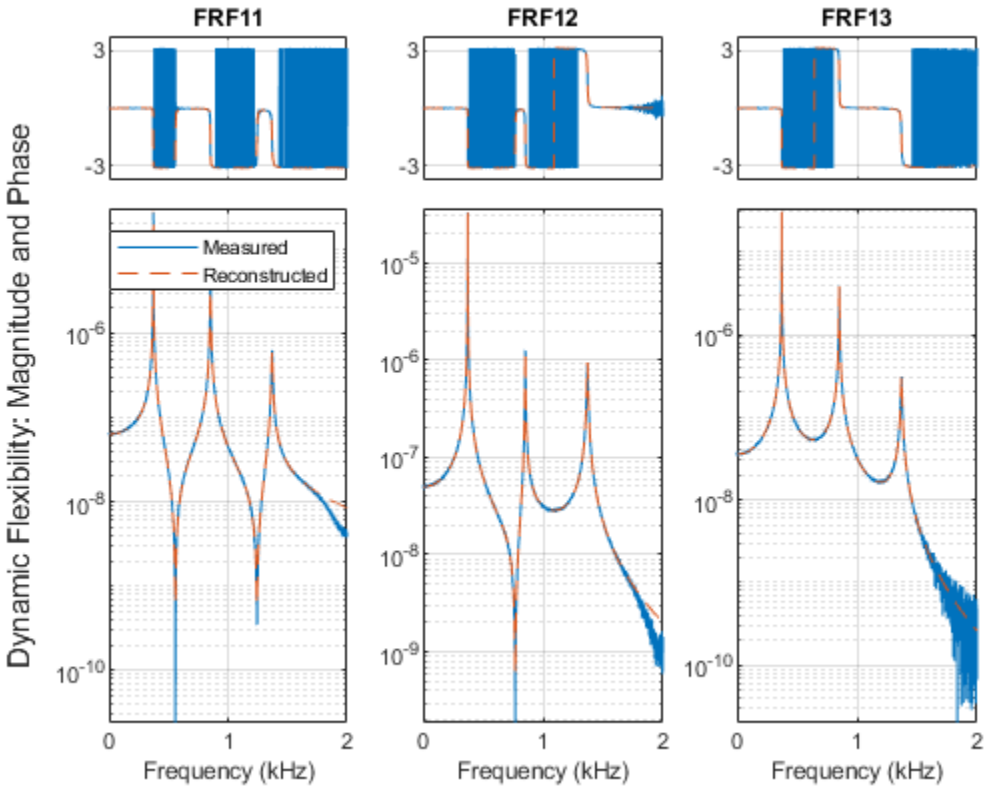



By default, poles are classified as stable in frequency if the natural frequency for the poles changes by less than one percent. Poles which are stable in frequency are further classified as stable in damping for a smaller than five percent change in damping ratio. Both criteria can be adjusted to different values. Based on the location of the stable poles, choose natural frequencies of 373, 852.5, and 1371 Hz. These frequencies are contained in the output `fn` of `modalsd`, along with natural frequencies of other frequency-stable poles. A higher model order than the number of modes physically present is generally needed to produce good modal parameters estimates using the LSCE algorithm. In this case, a model order of four modes indicates three stable poles. The frequencies of interest occur in the first three columns in the 4th row of `fn`.

```
physFreq = fn(4,[1 2 3]);
```

Estimate natural frequencies and damping and plot reconstructed and measured FRFs. Specify four modes and physical frequencies determined from the stability diagram, `'PhysFreq'`. `modalfit` returns modal parameters only for the specified modes.

```
modalfit(FRF,f,fs,4,'PhysFreq',physFreq)
```

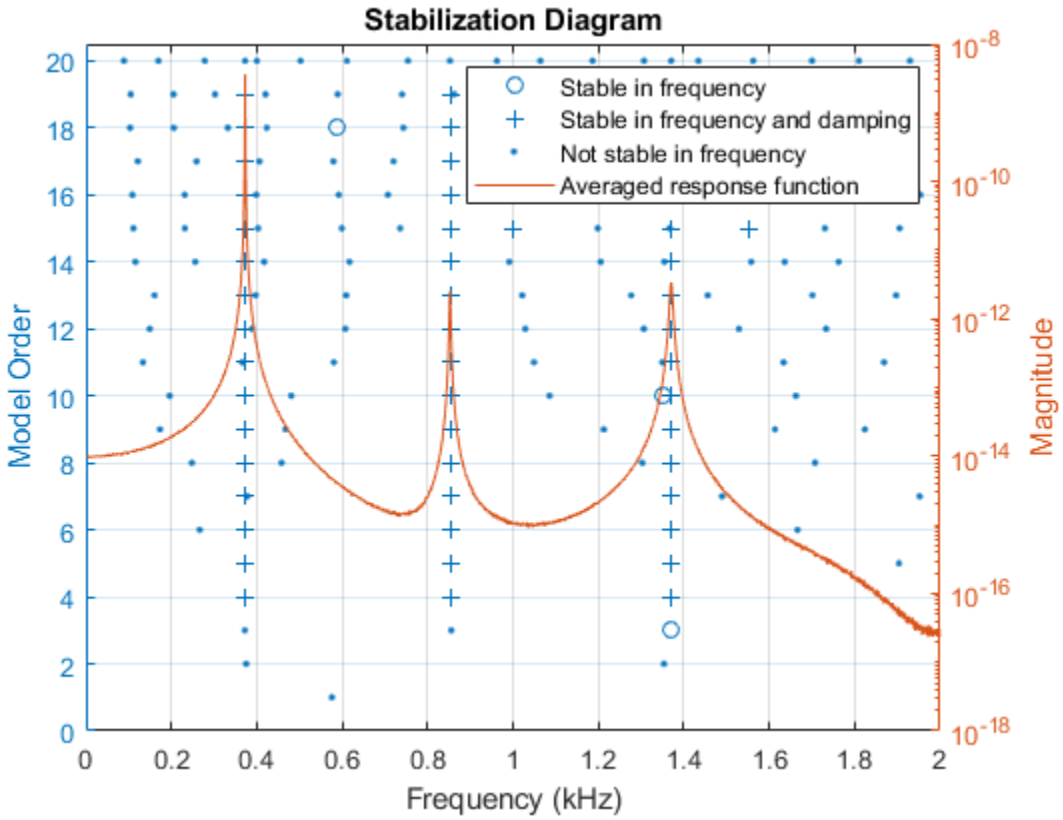



```
[fn1,dr1] = modalfit(FRF,f,fs,4,'PhysFreq',physFreq)
```

```
fn1 =

   1.0e+03 *

    0.3727
    0.8525
    1.3706


dr1 =

    0.0008
    0.0018
    0.0029
```

Next, compute FRFs and plot a stabilization diagram for a second set of hammer impacts with the sensor at a different location. Change the stability criterion to 0.1 percent for frequency and 2.5 percent for damping.

```
[FRF,f] = modalfrf(X2,Y2,fs,winLen,'Sensor','dis','Measurement','rovinginput');
fn = modalsd(FRF,f,fs,'MaxModes',20,'SCriteria',[0.001 0.025]);
```




With the more stringent criteria, the majority of poles are classified as not stable in frequency. The poles that are stable in frequency and damping align closely with the averaged FRF, suggesting they are present in the measured data.

```
physFreq = fn(4,[1 2 3]);
```

Extract modal parameters for this set of measurements and compare to the modal parameters for the first set of measurements. Specify the indices of the driving point FRF, corresponding to location where the excitation and response measurements coincide. The natural frequencies agree to within a fraction of a percent, and the damping ratios agree to within less than four percent, indicating that the modal parameters are consistent from measurement to measurement.

```
[fn2,dr2] = modalfit(FRF,f,fs,4,'PhysFreq',physFreq,'DriveIndex',[1 2])
```

```
fn2 =

   1.0e+03 *

    0.3727
    0.8525
    1.3705


dr2 =

    0.0008
```

```
    0.0018
    0.0029
```

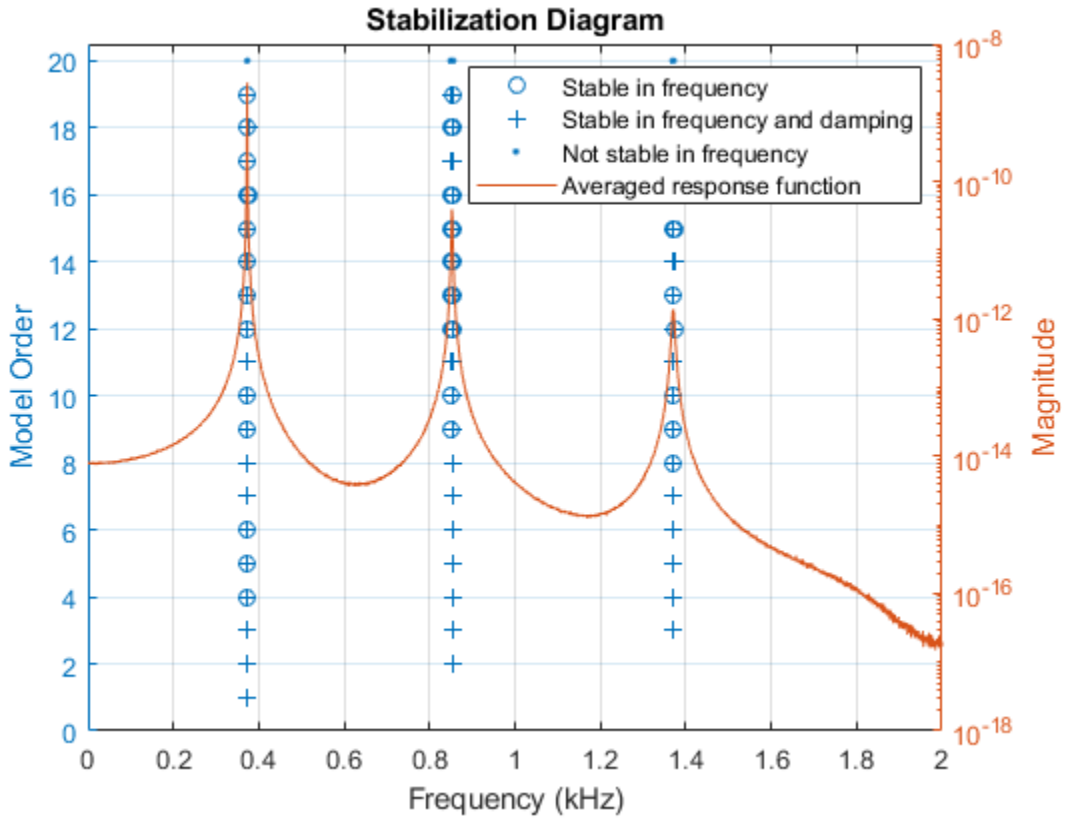
```
Tdiff2 = table((fn1-fn2)./fn1,(dr1-dr2)./dr1,'VariableNames',{'diffFrequency','diffDamping'})
```

```
Tdiff2 =

  3x2 table

    diffFrequency    diffDamping
    _____________    ___________

      2.9972e-06      -0.031648
     -5.9335e-06     -0.0099076
       1.965e-05      0.0001186
```

A parametric method for modal parameter estimation can provide a useful alternative to peak-picking and the LSCE method when there is measurement noise in the FRF or the FRF shows high modal density. The least-squares rational function (LSRF) approach fits a shared denominator transfer function to the multi-input, multi-output FRF and thus obtains a single, global estimate of modal parameters [4]. The procedure for using LSRF approach is similar to that for LSCE. You can use stabilization diagram to identify stationary modes and extract modal parameters corresponding to the identified physical frequencies.

```
[FRF,f] = modalfrf(X1,Y1,fs,winLen,'Sensor','dis','Measurement','rovinginput');
fn = modalsd(FRF,f,fs,'MaxModes',20, 'FitMethod', 'lsrf'); % Identify physical modes using lsfr
physFreq = fn(4,[1 2 3]);
[fn3,dr3] = modalfit(FRF,f,fs,4,'PhysFreq',physFreq,'DriveIndex',[1 2],'FitMethod','lsrf')
```

```
fn3 =

  372.6832
  372.9275
  852.4986


dr3 =

    0.0008
    0.0003
    0.0018
```




```
Tdiff3 = table((fn1-fn3)./fn1,(dr1-dr3)./dr1,'VariableNames',{'diffFrequency','diffDamping'})
```

```
Tdiff3 =

  3x2 table

    diffFrequency    diffDamping
    _____________    ___________

     -7.8599e-06      0.007982
         0.56255       0.83086
         0.37799       0.37626
```

A final note about parametric methods: the FRF estimation method ('subspace') and the modal parameter estimation method (`'lsrf'`) are similar to those used in System Identification Toolbox for fitting dynamic models to time-domain signals or to the frequency-response functions. If you have this toolbox available, you can identify models to fit your data using commands such as `tfest` and `ssest`. You can assess the quality of the identified models using `compare` and `resid` commands. Once you have validated the quality of the model, you can use them for extracting modal parameters. This is shown briefly using a state-space estimator.

```
Ts = 1/fs; % sample time
% Create a data object to be used for model estimation.
EstimationData = iddata(Y0(1:1000), X0(1:1000), 1/fs);
```

```
% Create a data object to be used for model validation
ValidationData = iddata(Y0(1001:2000), X0(1001:2000), 1/fs);
```

Identify a continuous-time state-space model of 6th order containing a feedthrough term.

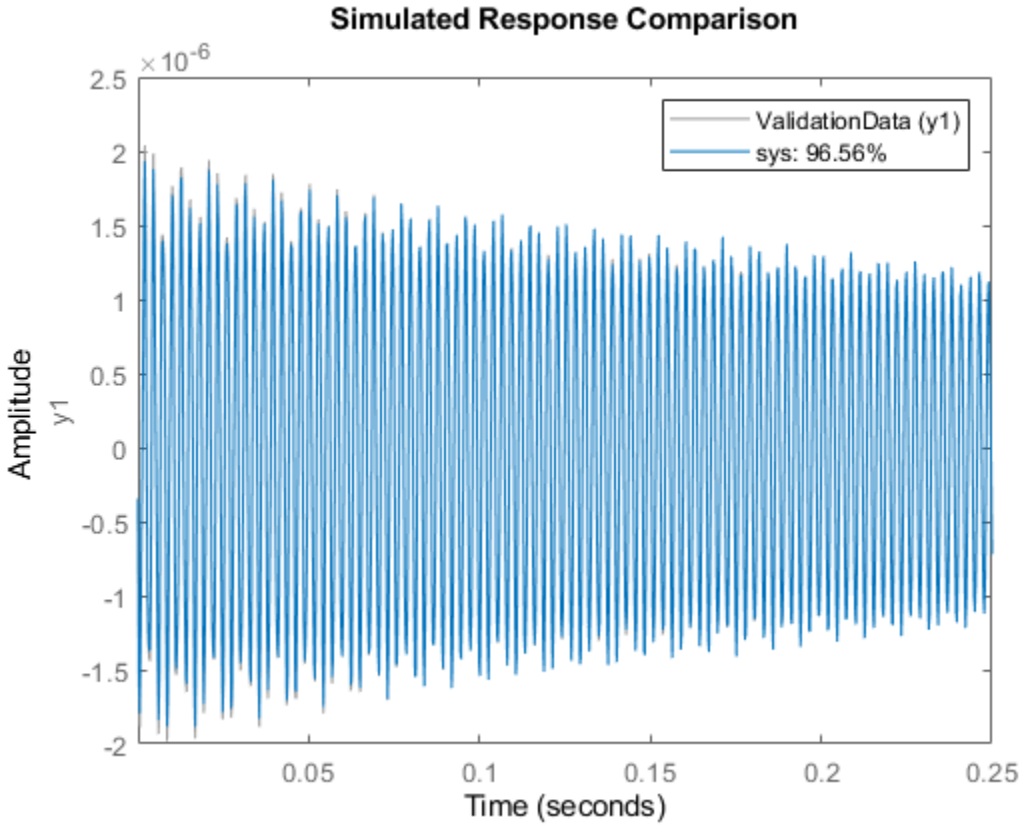
```
sys = ssest(EstimationData, 6, 'Feedthrough', true)
```

```
sys =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
           x1       x2       x3       x4       x5       x6
   x1     4.05    -1765    149.8    -1880   -49.64     -358
   x2     1764  -0.3332     2197   -232.5   -438.3   -128.4
   x3   -152.4    -2198     2.85     4715    255.9    547.5
   x4     1879    228.2    -4713    -15.9    -1216   -28.79
   x5    59.42    440.9   -275.5     1217    35.05    -8508
   x6    363.7    120.2   -545.4   -44.02     8508   -92.45

  B =
           u1
   x1  -0.1513
   x2   -1.911
   x3    4.439
   x4   -3.118
   x5  -0.9416
   x6   -8.039

  C =
              x1          x2          x3          x4          x5          x6
   y1   3.135e-05   2.511e-06   8.634e-06  -1.416e-05   2.218e-06  -6.271e-06

  D =
             u1
   y1  7.564e-09

  K =
              y1
   x1   3.513e+07
   x2  -3.244e+06
   x3  -3.598e+07
   x4  -1.059e+07
   x5   1.724e+08
   x6   7.521e+06

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: yes
   Disturbance component: estimate
   Number of free coefficients: 55
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "EstimationData".
```

```
Fit to estimation data: 99.3% (prediction focus)
FPE: 1.235e-16, MSE: 1.189e-16
```

Assess the quality of the model by checking how well it fits the validation data.

```
clf
compare(ValidationData, sys)  % plot shows good fit
```

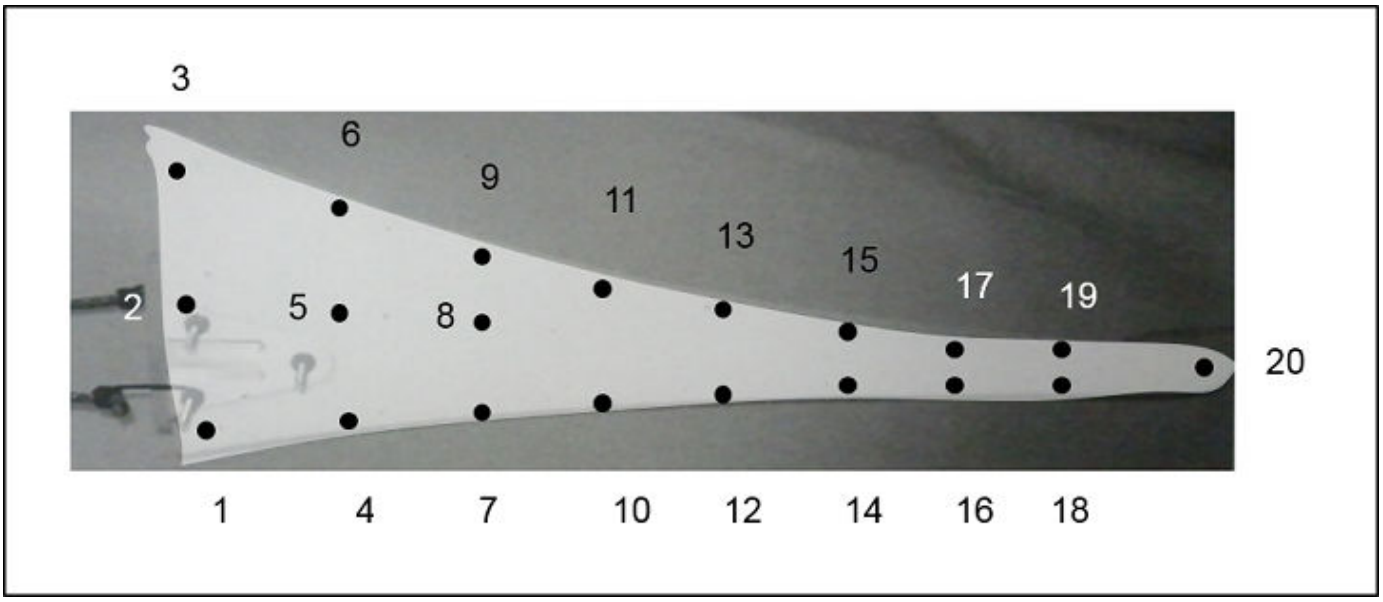



Use the model `sys` to compute modal parameters.

```
[fn4, dr4] = modalfit(sys, f, 3);
```

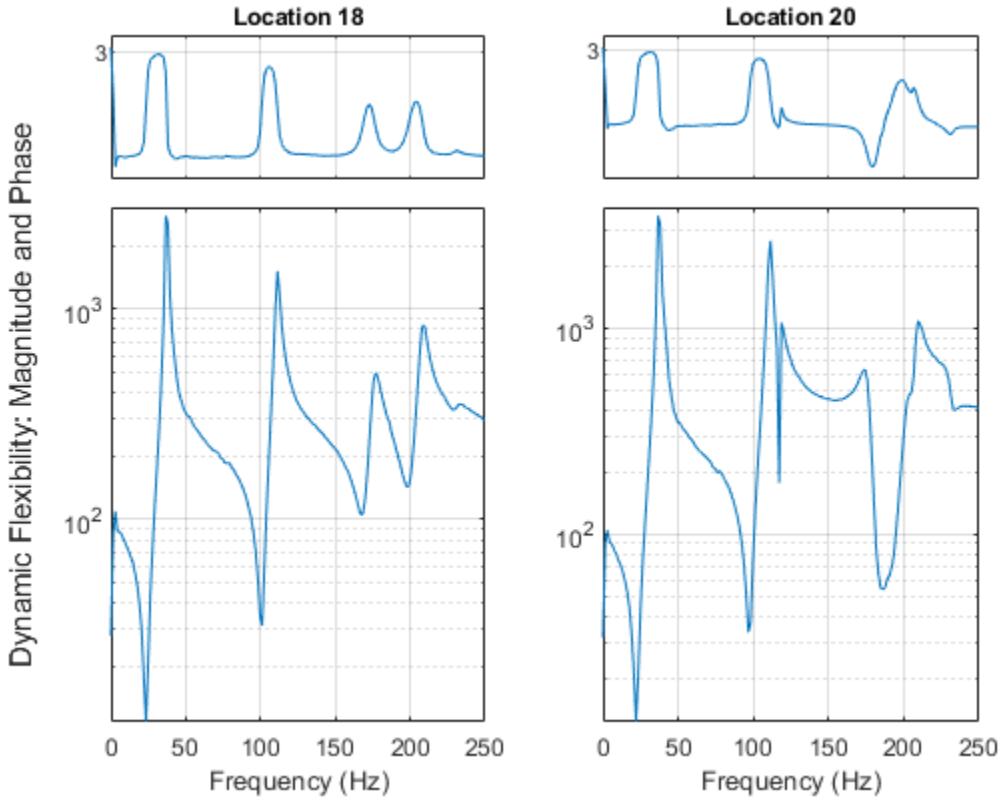
### Mode Shape Vectors of a Wind Turbine Blade

Understanding the dynamic behavior of wind turbine blades is important to optimize operational efficiency and predict blade failure. This section analyzes experimental modal analysis data for a wind turbine blade and visualizes mode shapes of the blade. A hammer excites the turbine blade at 20 locations, and a reference accelerometer measures the responses at location 18. An aluminum block is mounted at the base of the blade, and the blade is excited in the flap-wise orientation, perpendicular to the flat part of the blade. An FRF is collected for each location. The FRF data are kindly provided by the Structural Dynamics and Acoustics Systems Laboratory at the University of Massachusetts, Lowell. First, visualize the spatial arrangement of the measurement locations.



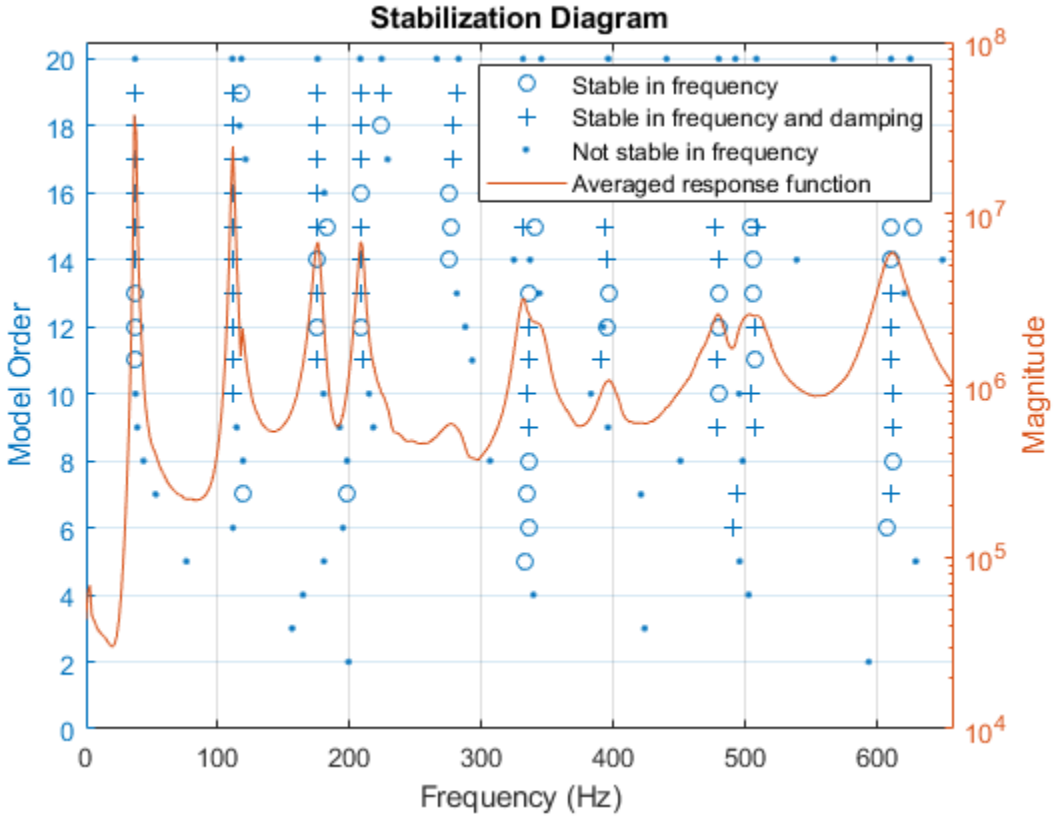

Load and plot the wind turbine blade FRF estimates for locations 18 and 20. Zoom in on the first few peaks.

```
[FRF,f,fs] = helperImportModalData();
helperPlotModalAnalysisExample(FRF,f,[18 20]);
```



The first two modes appear as peaks around 37 Hz and 111 Hz. Plot a stabilization diagram to estimate the natural frequencies. The first two values returned for a model order of 14 are stable in frequency and damping ratio.

```
fn = modalsd(FRF,f,fs,'MaxModes',20);
physFreq = fn(14,[1 2]);
```

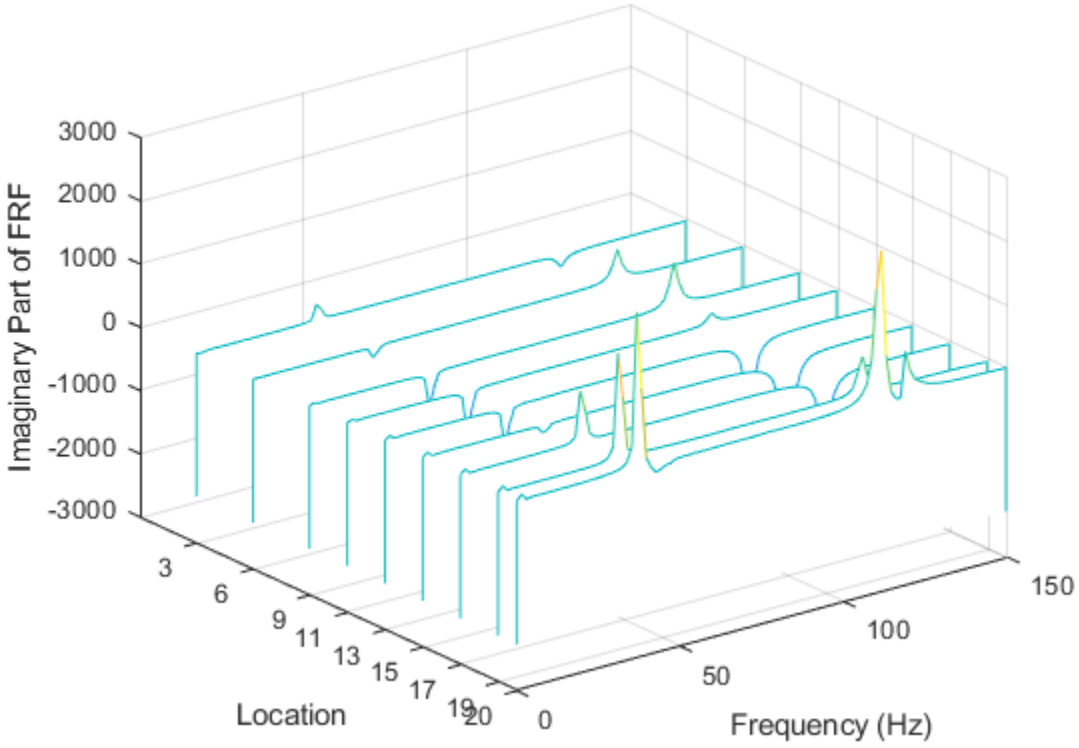



Next, extract the mode shapes for the first two modes using `modalfit`. Limit the fit to the frequency range from 0 to 250 Hz based on the previous plot.

```
[~,~,ms] = modalfit(FRF,f,fs,14,'PhysFreq',physFreq,'FreqRange',[0 250]);
```

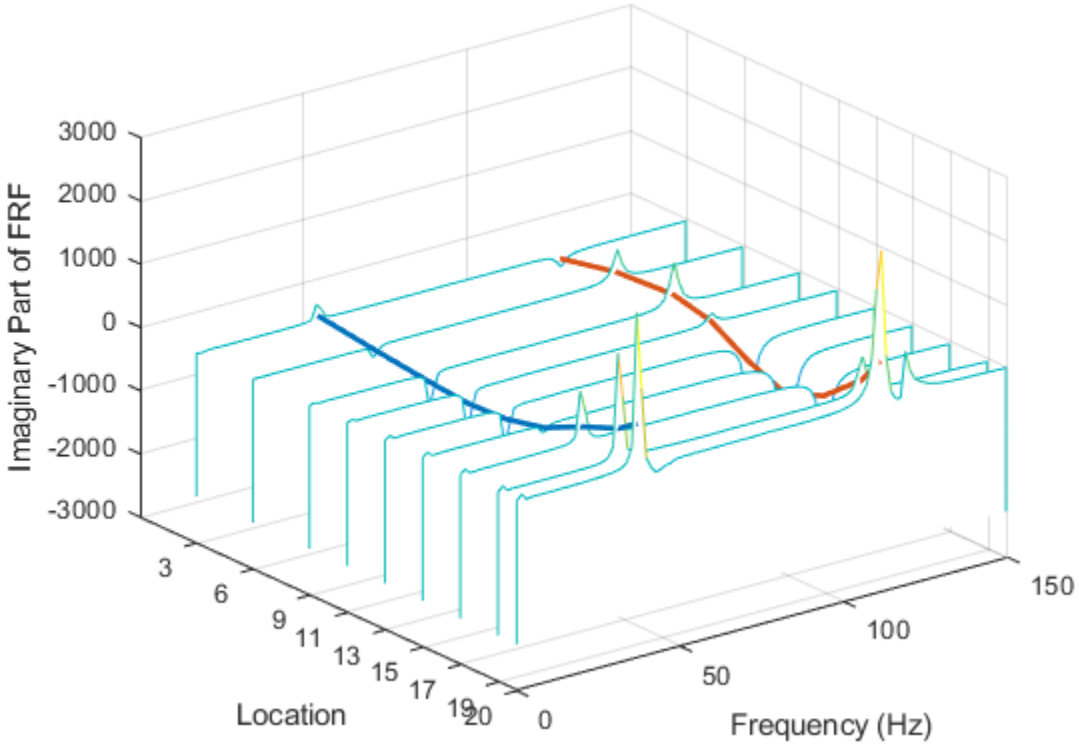
Mode shapes quantify the amplitude of motion for each mode of a structure at each location. To estimate a mode shape vector, one row or column of the frequency-response function matrix is needed. In practice, this means that either an excitation is needed at every measurement location of the structure (in this case, a roving hammer), or that a response measurement is needed at every location. Mode shapes can be visualized by examining the imaginary part of the FRF. Plot a waterfall diagram of the imaginary part of the FRF matrix for locations on one side of the blade. Limit the frequency range to a maximum of 150 Hz to examine the first two modes. The peaks of the plot represent mode shapes.

```
measlocs = [3 6 9 11 13 15 17 19 20]; % Measurement locations on blade edge
helperPlotModalAnalysisExample(FRF,f,measlocs,150);
```




The shapes indicated in the plot by the contour of the peaks represent the first and second bending moments of the blade. Next, plot the magnitude of mode shape vectors for the same measurement locations.

```
helperPlotModalAnalysisExample(ms,measlocs);
```




While the amplitudes are scaled differently (mode shape vectors are scaled to unity modal A), the mode shape contours agree in shape. The shape of the first mode has a large tip displacement and two nodes, where vibration amplitude is zero. The second mode also has a large tip displacement and has three nodes.

### Summary

This example analyzed and compared simulated modal analysis datasets for a 3DOF system excited by a roving hammer. It estimated natural frequency and damping using a stabilization diagram and the LSCE and LSRF algorithms. The modal parameters were consistent for two sets of measurements. In a separate use case, mode shapes of a wind turbine blade were visualized using the imaginary part of the FRF matrix and mode shape vectors.

### Acknowledgment

Thanks to Dr. Peter Avitabile from the Structural Dynamics and Acoustics Systems Laboratory at the University of Massachusetts Lowell for facilitating the collection of the wind turbine blade experimental data.

### References

[1] Brandt, Anders. *Noise and Vibration Analysis: Signal Analysis and Experimental Procedures*. Chichester, UK: John Wiley and Sons, 2011.

[2] Vold, Håvard, John Crowley, and G. Thomas Rocklin. "New Ways of Estimating Frequency Response Functions." *Sound and Vibration*. Vol. 18, November 1984, pp. 34-38.

[3] Peter Van Overschee and Bart De Moor. "N4SID: Subspace Algorithms for the Identification of Combined Deterministic-Stochastic Systems." *Automatica*. Vol. 30, January 1994, pp. 75-93.

[4] Ozdemir, A. A., and S. Gumussoy. "Transfer Function Estimation in System Identification Toolbox via Vector Fitting." *Proceedings of the 20th World Congress of the International Federation of Automatic Control*. Toulouse, France, July 2017.

## See Also

`modalfit` | `modalfrf` | `modalsd`

# Accelerating Correlation with GPUs

This example shows how to use a GPU to accelerate cross-correlation. Many correlation problems involve large data sets and can be solved much faster using a GPU. This example requires a Parallel Computing Toolbox™ user license. Refer to “GPU Support by Release” (Parallel Computing Toolbox) to see what GPUs are supported.

### Introduction

Start by learning some basic information about the GPU in your machine. To access the GPU, use the Parallel Computing Toolbox.

```
fprintf('Benchmarking GPU-accelerated Cross-Correlation.\n');

if ~(parallel.gpu.GPUDevice.isAvailable)
    fprintf(['\n\t**GPU not available. Stopping.**\n']);
    return;
else
    dev = gpuDevice;
    fprintf(...
    'GPU detected (%s, %d multiprocessors, Compute Capability %s)',...
    dev.Name, dev.MultiprocessorCount, dev.ComputeCapability);
end
```

```
Benchmarking GPU-accelerated Cross-Correlation.
GPU detected (TITAN Xp, 30 multiprocessors, Compute Capability 6.1)
```

### Benchmarking Functions

Because code written for the CPU can be ported to run on the GPU, a single function can be used to benchmark both the CPU and GPU. However, because code on the GPU executes asynchronously from the CPU, special precaution should be taken when measuring performance. Before measuring the time taken to execute a function, ensure that all GPU processing has finished by executing the 'wait' method on the device. This extra call will have no effect on the CPU performance.

This example benchmarks three different types of cross-correlation.

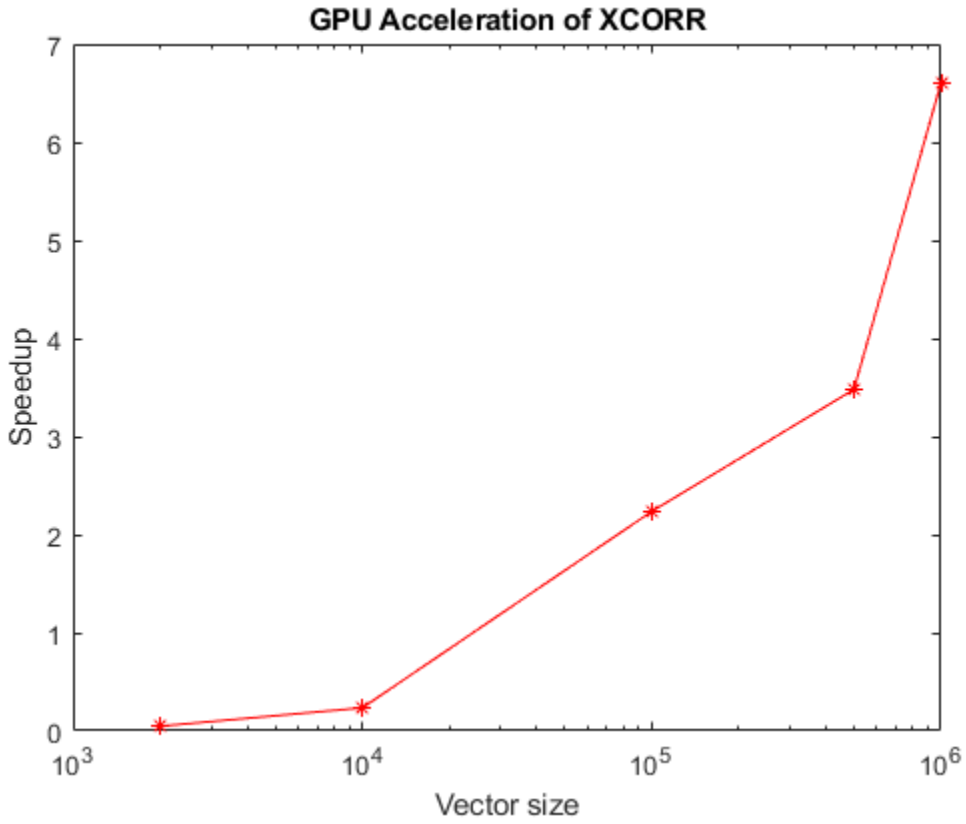
### Benchmark Simple Cross-Correlation

For the first case, two vectors of equal size are cross-correlated using the syntax `xcorr(u,v)`. The ratio of CPU execution time to GPU execution time is plotted against the size of the vectors.

```
fprintf('\n\n *** Benchmarking vector-vector cross-correlation*** \n\n');
fprintf('Benchmarking function :\n');
type('benchXcorrVec');
fprintf('\n\n');

sizes = [2000 1e4 1e5 5e5 1e6];
tc = zeros(1,numel(sizes));
tg = zeros(1,numel(sizes));
numruns = 10;

for s=1:numel(sizes);
    fprintf('Running xcorr of %d elements...\n', sizes(s));
    delchar = repmat('\b', 1,numruns);

    a = rand(sizes(s),1);
```

```
    b = rand(sizes(s),1);
    tc(s) = benchXcorrVec(a, b, numruns);
    fprintf([delchar '\t\tCPU  time : %.2f ms\n'], 1000*tc(s));
    tg(s) = benchXcorrVec(gpuArray(a), gpuArray(b), numruns);
    fprintf([delchar '\t\tGPU time :  %.2f ms\n'], 1000*tg(s));
end

%Plot the results
fig = figure;
ax = axes('parent', fig);
semilogx(ax, sizes, tc./tg, 'r*-');
ylabel(ax, 'Speedup');
xlabel(ax, 'Vector size');
title(ax, 'GPU Acceleration of XCORR');
drawnow;
```

```
 *** Benchmarking vector-vector cross-correlation***

Benchmarking function :

function t = benchXcorrVec(u,v, numruns)
%Used to benchmark xcorr with vector inputs on the CPU and GPU.

%   Copyright 2012 The MathWorks, Inc.

    timevec = zeros(1,numruns);
    gdev = gpuDevice;
    for ii=1:numruns
        ts = tic;
        o = xcorr(u,v); %#ok<NASGU>
        wait(gdev)
        timevec(ii) = toc(ts);
        fprintf('.');
    end
    t = min(timevec);
end


Running xcorr of 2000 elements...
        CPU  time : 0.21 ms
        GPU time :  4.26 ms
Running xcorr of 10000 elements...
        CPU  time : 1.03 ms
        GPU time :  4.37 ms
Running xcorr of 100000 elements...
        CPU  time : 14.04 ms
        GPU time :  6.28 ms
Running xcorr of 500000 elements...
        CPU  time : 55.98 ms
        GPU time :  16.09 ms
Running xcorr of 1000000 elements...
        CPU  time : 169.00 ms
        GPU time :  25.60 ms
```




**Benchmarking Matrix Column Cross-Correlation**

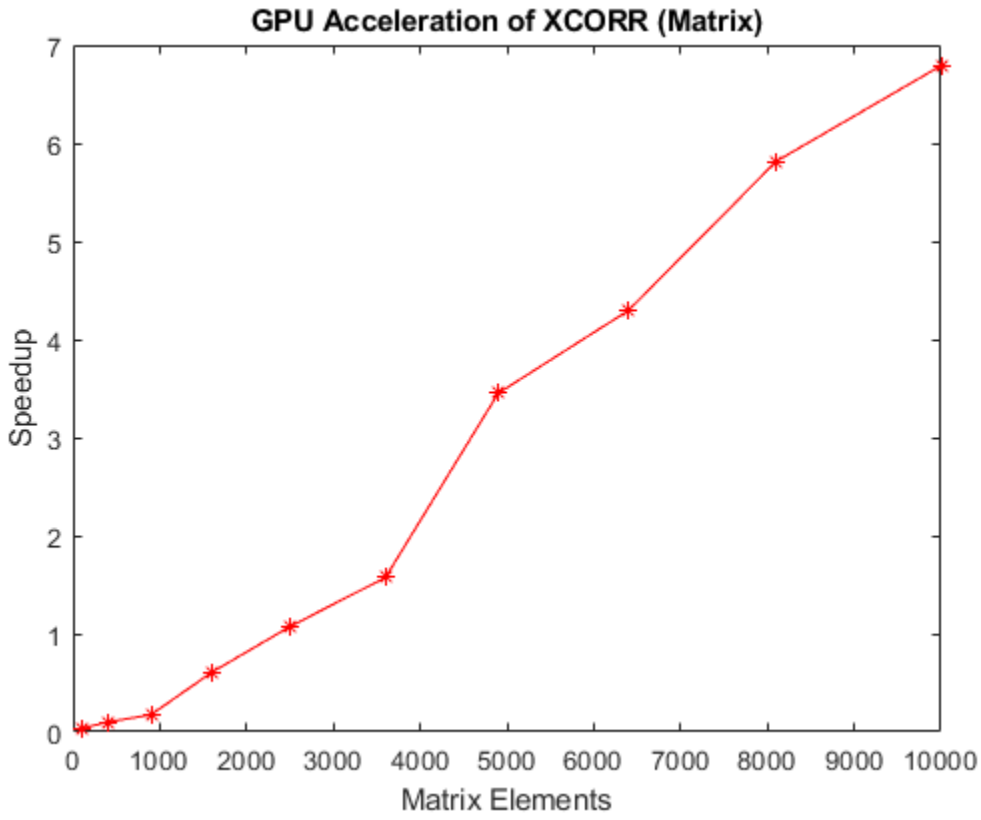
For the second case, the columns of a matrix A are pairwise cross-correlated to produce a large matrix output of all correlations using the syntax xcorr(A). The ratio of CPU execution time to GPU execution time is plotted against the size of the matrix A.

```
fprintf('\n\n *** Benchmarking matrix column cross-correlation*** \n\n');
fprintf('Benchmarking function :\n');
type('benchXcorrMatrix');
fprintf('\n\n');

sizes = floor(linspace(0,100, 11));
sizes(1) = [];
tc = zeros(1,numel(sizes));
tg = zeros(1,numel(sizes));
numruns = 10;

for s=1:numel(sizes);
    fprintf('Running xcorr (matrix) of a %d x %d matrix...\n', sizes(s), sizes(s));
    delchar = repmat('\b', 1,numruns);

    a = rand(sizes(s));
    tc(s) = benchXcorrMatrix(a, numruns);
    fprintf([delchar '\t\tCPU  time : %.2f ms\n'], 1000*tc(s));
    tg(s) = benchXcorrMatrix(gpuArray(a), numruns);
    fprintf([delchar '\t\tGPU time :  %.2f ms\n'], 1000*tg(s));
end
```

```
%Plot the results
fig = figure;
ax = axes('parent', fig);
plot(ax, sizes.^2, tc./tg, 'r*-');
ylabel(ax, 'Speedup');
xlabel(ax, 'Matrix Elements');
title(ax, 'GPU Acceleration of XCORR (Matrix)');
drawnow;
```

```
 *** Benchmarking matrix column cross-correlation***

Benchmarking function :

function t = benchXcorrMatrix(A, numruns)
%Used to benchmark xcorr with Matrix input on CPU and GPU.

%   Copyright 2012 The MathWorks, Inc.

    timevec = zeros(1,numruns);
    gdev = gpuDevice;
    for ii=1:numruns,
        ts = tic;
        o = xcorr(A); %#ok<NASGU>
        wait(gdev)
        timevec(ii) = toc(ts);
        fprintf('.');
    end
    t = min(timevec);
end


Running xcorr (matrix) of a 10 x 10 matrix...
        CPU  time : 0.18 ms
        GPU time :  5.00 ms
Running xcorr (matrix) of a 20 x 20 matrix...
        CPU  time : 0.48 ms
        GPU time :  4.83 ms
Running xcorr (matrix) of a 30 x 30 matrix...
        CPU  time : 0.85 ms
        GPU time :  4.84 ms
Running xcorr (matrix) of a 40 x 40 matrix...
        CPU  time : 3.38 ms
        GPU time :  5.57 ms
Running xcorr (matrix) of a 50 x 50 matrix...
        CPU  time : 5.60 ms
        GPU time :  5.22 ms
Running xcorr (matrix) of a 60 x 60 matrix...
        CPU  time : 8.49 ms
        GPU time :  5.39 ms
Running xcorr (matrix) of a 70 x 70 matrix...
        CPU  time : 20.43 ms
        GPU time :  5.92 ms
Running xcorr (matrix) of a 80 x 80 matrix...
        CPU  time : 26.79 ms
        GPU time :  6.24 ms
```

```
Running xcorr (matrix) of a 90 x 90 matrix...
        CPU  time : 40.04 ms
        GPU time :  6.89 ms
Running xcorr (matrix) of a 100 x 100 matrix...
        CPU  time : 49.69 ms
        GPU time :  7.32 ms
```




### Benchmarking Two-Dimensional Cross-Correlation

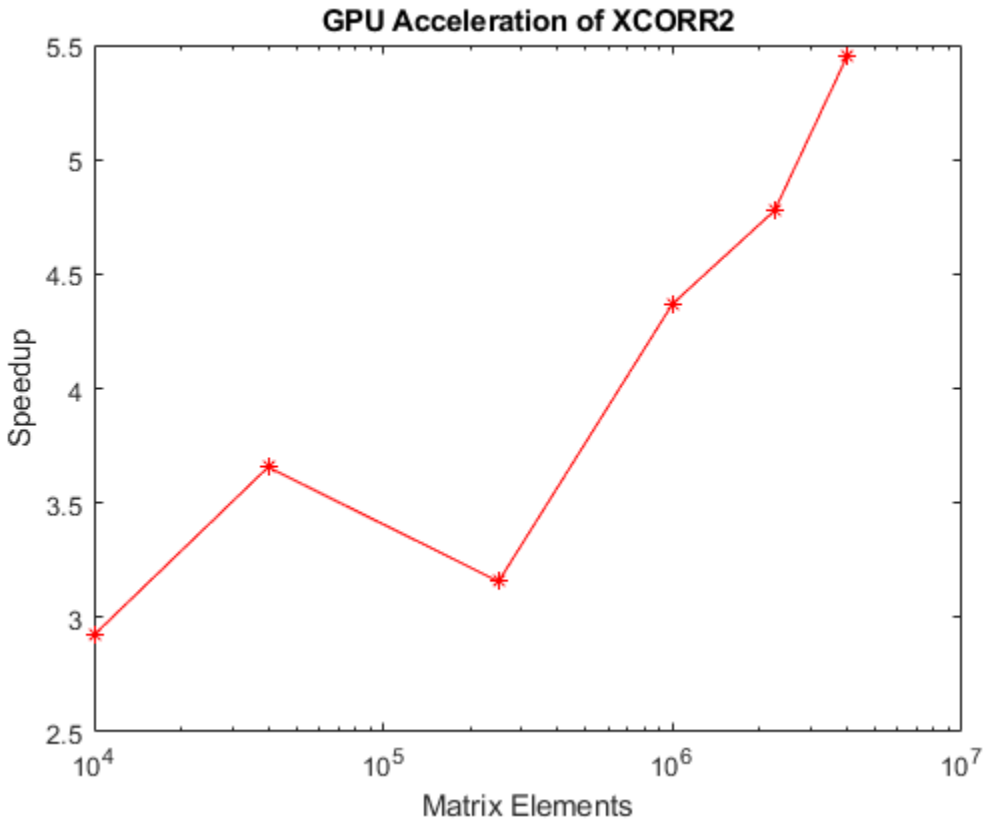
For the final case, two matrices, X and Y, are cross correlated using xcorr2(X,Y). X is fixed in size while Y is allowed to vary. The speedup is plotted against the size of the second matrix.

```
fprintf('\n\n *** Benchmarking 2-D cross-correlation*** \n\n');
fprintf('Benchmarking function :\n');
type('benchXcorr2');
fprintf('\n\n');

sizes = [100, 200, 500, 1000, 1500, 2000];
tc = zeros(1,numel(sizes));
tg = zeros(1,numel(sizes));
numruns = 4;
a = rand(100);

for s=1:numel(sizes);
    fprintf('Running xcorr2 of a 100x100 matrix and %d x %d matrix...\n', sizes(s), sizes(s));
    delchar = repmat('\b', 1,numruns);

    b = rand(sizes(s));
```

```
    tc(s) = benchXcorr2(a, b, numruns);
    fprintf([delchar '\t\tCPU  time : %.2f ms\n'], 1000*tc(s));
    tg(s) = benchXcorr2(gpuArray(a), gpuArray(b), numruns);
    fprintf([delchar '\t\tGPU time :  %.2f ms\n'], 1000*tg(s));
end

%Plot the results
fig = figure;
ax =axes('parent', fig);
semilogx(ax, sizes.^2, tc./tg, 'r*-');
ylabel(ax, 'Speedup');
xlabel(ax, 'Matrix Elements');
title(ax, 'GPU Acceleration of XCORR2');
drawnow;

fprintf('\n\nBenchmarking completed.\n\n');
```

```
 *** Benchmarking 2-D cross-correlation***

Benchmarking function :

function t = benchXcorr2(X, Y, numruns)
%Used to benchmark xcorr2 on the CPU and GPU.

%   Copyright 2012 The MathWorks, Inc.

    timevec = zeros(1,numruns);
    gdev = gpuDevice;
    for ii=1:numruns,
        ts = tic;
        o = xcorr2(X,Y); %#ok<NASGU>
        wait(gdev)
        timevec(ii) = toc(ts);
        fprintf('.');
    end
    t = min(timevec);
end


Running xcorr2 of a 100x100 matrix and 100 x 100 matrix...
        CPU  time : 20.35 ms
        GPU time :  6.96 ms
Running xcorr2 of a 100x100 matrix and 200 x 200 matrix...
        CPU  time : 42.87 ms
        GPU time :  11.72 ms
Running xcorr2 of a 100x100 matrix and 500 x 500 matrix...
        CPU  time : 125.23 ms
        GPU time :  39.67 ms
Running xcorr2 of a 100x100 matrix and 1000 x 1000 matrix...
        CPU  time : 386.59 ms
        GPU time :  88.46 ms
Running xcorr2 of a 100x100 matrix and 1500 x 1500 matrix...
        CPU  time : 788.38 ms
        GPU time :  165.04 ms
Running xcorr2 of a 100x100 matrix and 2000 x 2000 matrix...
        CPU  time : 1523.05 ms
```

```
        GPU time :  279.55 ms

Benchmarking completed.
```




### Other GPU Accelerated Signal Processing Functions

There are several other signal processing functions that can be run on the GPU. These functions include fft, ifft, conv, filter, fftfilt, and more. In some cases, you can achieve large acceleration relative to the CPU. For a full list of GPU accelerated signal processing functions, see the “GPU Algorithm Acceleration” section in the Signal Processing Toolbox™ documentation.

## See Also

`gather` | `gpuArray` | `xcorr`