

# Signal Processing Toolbox

For Use with MATLAB®

- Computation
- Visualization
- Programming

## How to Contact The MathWorks:



[www.mathworks.com](http://www.mathworks.com)  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab)  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html)

Web  
Newsgroup  
Technical support



[suggest@mathworks.com](mailto:suggest@mathworks.com)  
[bugs@mathworks.com](mailto:bugs@mathworks.com)  
[doc@mathworks.com](mailto:doc@mathworks.com)  
[service@mathworks.com](mailto:service@mathworks.com)  
[info@mathworks.com](mailto:info@mathworks.com)

Product enhancement suggestions  
Bug reports  
Documentation error reports  
Order status, license renewals, passcodes  
Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

For contact information about worldwide offices, see the MathWorks Web site.

### *Signal Processing Toolbox User's Guide*

© COPYRIGHT 1988–2006 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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks of The MathWorks, Inc. Other product or brand names are trademarks or registered trademarks of their respective holders.

### **Patents**

The MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## Revision History

1988	First printing	New
January 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 6.3 (Release 14SP2)
September 2005	Online only	Revised for Version 6.4 (Release 14SP3)
March 2006	Sixth printing	Revised for Version 6.5 (Release 2006a)



## Signal Processing Basics

1

<b>What Is the Signal Processing Toolbox? . . . . .</b>	<b>1-2</b>
Signal Processing Toolbox Central Features . . . . .	1-2
Filtering and FFTs . . . . .	1-3
Signals and Systems . . . . .	1-3
Key Areas: Filter Design and Spectral Analysis . . . . .	1-3
Interactive Tools . . . . .	1-4
Extensibility . . . . .	1-4
<b>Representing Signals . . . . .</b>	<b>1-5</b>
Vector Representation . . . . .	1-5
<b>Waveform Generation: Time Vectors and Sinusoids . . . . .</b>	<b>1-7</b>
Common Sequences: Unit Impulse, Unit Step, and Unit Ramp . . . . .	1-8
Multichannel Signals . . . . .	1-8
Common Periodic Waveforms . . . . .	1-9
Common Aperiodic Waveforms . . . . .	1-10
The pulstran Function . . . . .	1-11
The Sinc Function . . . . .	1-12
The Dirichlet Function . . . . .	1-13
<b>Working with Data . . . . .</b>	<b>1-14</b>
<b>Filter Implementation and Analysis . . . . .</b>	<b>1-15</b>
Convolution and Filtering . . . . .	1-15
Filters and Transfer Functions . . . . .	1-16
Filtering with the filter Function . . . . .	1-17
<b>The filter Function . . . . .</b>	<b>1-18</b>
<b>Other Functions for Filtering . . . . .</b>	<b>1-20</b>
Multirate Filter Bank Implementation . . . . .	1-20
Anti-Causal, Zero-Phase Filter Implementation . . . . .	1-21
Frequency Domain Filter Implementation . . . . .	1-23

<b>Impulse Response</b> .....	<b>1-24</b>
<b>Frequency Response</b> .....	<b>1-26</b>
Digital Domain .....	<b>1-26</b>
Analog Domain .....	<b>1-28</b>
Magnitude and Phase .....	<b>1-29</b>
Delay .....	<b>1-31</b>
<b>Zero-Pole Analysis</b> .....	<b>1-33</b>
<b>Linear System Models</b> .....	<b>1-35</b>
Discrete-Time System Models .....	<b>1-35</b>
Continuous-Time System Models .....	<b>1-44</b>
Linear System Transformations .....	<b>1-45</b>
<b>Discrete Fourier Transform</b> .....	<b>1-47</b>
<b>Selected Bibliography</b> .....	<b>1-50</b>

## Filter Design and Implementation

# 2

<b>Filter Requirements and Specification</b> .....	<b>2-2</b>
<b>IIR Filter Design</b> .....	<b>2-4</b>
Classical IIR Filter Design Using Analog Prototyping .....	<b>2-6</b>
Comparison of Classical IIR Filter Types .....	<b>2-9</b>
<b>FIR Filter Design</b> .....	<b>2-17</b>
Linear Phase Filters .....	<b>2-18</b>
Windowing Method .....	<b>2-19</b>
Multiband FIR Filter Design with Transition Bands .....	<b>2-24</b>
Constrained Least Squares FIR Filter Design .....	<b>2-31</b>
Arbitrary-Response Filter Design .....	<b>2-37</b>

<b>Special Topics in IIR Filter Design</b> .....	<b>2-42</b>
Analog Prototype Design .....	2-43
Frequency Transformation .....	2-43
Filter Discretization .....	2-46
<b>Filter Implementation</b> .....	<b>2-52</b>
Using dfilter .....	2-53
<b>Selected Bibliography</b> .....	<b>2-54</b>

## Statistical Signal Processing

### 3

<b>Correlation and Covariance</b> .....	<b>3-2</b>
Bias and Normalization .....	3-4
Multiple Channels .....	3-5
<b>Spectral Analysis</b> .....	<b>3-6</b>
Spectral Estimation Method .....	3-8
Nonparametric Methods .....	3-10
Parametric Methods .....	3-36
<b>Selected Bibliography</b> .....	<b>3-49</b>

## Special Topics

### 4

<b>Windows</b> .....	<b>4-2</b>
Graphical User Interface Tools .....	4-3
Basic Shapes .....	4-3
Generalized Cosine Windows .....	4-7
Kaiser Window .....	4-9
Chebyshev Window .....	4-14

<b>Parametric Modeling</b> .....	<b>4-15</b>
Time-Domain Based Modeling .....	<b>4-17</b>
Frequency-Domain Based Modeling .....	<b>4-22</b>
<b>Resampling</b> .....	<b>4-25</b>
<b>Cepstrum Analysis</b> .....	<b>4-28</b>
Inverse Complex Cepstrum .....	<b>4-31</b>
<b>FFT-Based Time-Frequency Analysis</b> .....	<b>4-32</b>
<b>Median Filtering</b> .....	<b>4-33</b>
<b>Communications Applications</b> .....	<b>4-34</b>
<b>Deconvolution</b> .....	<b>4-41</b>
<b>Specialized Transforms</b> .....	<b>4-42</b>
Chirp z-Transform .....	<b>4-42</b>
Discrete Cosine Transform .....	<b>4-44</b>
Hilbert Transform .....	<b>4-46</b>
<b>Selected Bibliography</b> .....	<b>4-48</b>

## **FDATool: A Filter Design and Analysis GUI**

# 5

<b>Overview</b> .....	<b>5-2</b>
Filter Design Methods .....	<b>5-3</b>
Using the Filter Design and Analysis Tool .....	<b>5-5</b>
Analyzing Filter Responses .....	<b>5-5</b>
Filter Design and Analysis Tool Panels .....	<b>5-6</b>
Getting Help .....	<b>5-7</b>
<b>Opening FDATool</b> .....	<b>5-8</b>



<b>Choosing a Response Type</b> .....	<b>5-9</b>
<b>Choosing a Filter Design Method</b> .....	<b>5-10</b>
<b>Setting the Filter Design Specifications</b> .....	<b>5-11</b>
Filter Order .....	<b>5-12</b>
Options .....	<b>5-13</b>
Bandpass Filter Frequency Specifications .....	<b>5-14</b>
Bandpass Filter Magnitude Specifications .....	<b>5-15</b>
<b>Computing the Filter Coefficients</b> .....	<b>5-16</b>
<b>Analyzing the Filter</b> .....	<b>5-17</b>
Using Data Markers .....	<b>5-19</b>
Drawing Spectral Masks .....	<b>5-20</b>
Changing the Sampling Frequency .....	<b>5-22</b>
Displaying the Response in FVTool .....	<b>5-23</b>
<b>Editing the Filter Using the Pole/Zero Editor</b> .....	<b>5-25</b>
<b>Converting the Filter Structure</b> .....	<b>5-29</b>
Converting to a New Structure .....	<b>5-29</b>
Converting to Second-Order Sections .....	<b>5-30</b>
<b>Importing a Filter Design</b> .....	<b>5-32</b>
Filter Structures .....	<b>5-33</b>
<b>Exporting a Filter Design</b> .....	<b>5-36</b>
Exporting Coefficients or Objects to the Workspace .....	<b>5-36</b>
Exporting Coefficients to an ASCII File .....	<b>5-38</b>
Exporting Coefficients or Objects to a MAT-File .....	<b>5-38</b>
Exporting to SPTool .....	<b>5-39</b>
Exporting to Simulink .....	<b>5-40</b>
<b>Generating a C Header File</b> .....	<b>5-44</b>
<b>Generating an M-File</b> .....	<b>5-47</b>

<b>Managing Filters in the Current Session</b> .....	<b>5-48</b>
<b>Saving and Opening Filter Design Sessions</b> .....	<b>5-50</b>

## **SPTool: A Signal Processing GUI Suite**

# 6

<b>SPTool: An Interactive Signal Processing Environment</b> ..	<b>6-2</b>
SPTool Data Structures .....	<b>6-3</b>
<b>Opening SPTool</b> .....	<b>6-4</b>
<b>Getting Context-Sensitive Help</b> .....	<b>6-6</b>
<b>Signal Browser</b> .....	<b>6-7</b>
Opening the Signal Browser .....	<b>6-7</b>
<b>FDATool</b> .....	<b>6-9</b>
Opening FDATool .....	<b>6-9</b>
<b>Filter Visualization Tool</b> .....	<b>6-10</b>
Opening the Filter Visualization Tool .....	<b>6-10</b>
Analysis Parameters .....	<b>6-11</b>
<b>Spectrum Viewer</b> .....	<b>6-12</b>
Opening the Spectrum Viewer .....	<b>6-12</b>
<b>Filtering and Analysis of Noise</b> .....	<b>6-15</b>
Step 1: Importing a Signal into SPTool .....	<b>6-16</b>
Step 2: Designing a Filter .....	<b>6-18</b>
Step 3: Applying a Filter to a Signal .....	<b>6-21</b>
Step 4: Analyzing a Signal .....	<b>6-22</b>
Step 5: Spectral Analysis in the Spectrum Viewer .....	<b>6-24</b>

<b>Exporting Signals, Filters, and Spectra</b> .....	<b>6-27</b>
Opening the Export Dialog Box .....	<b>6-27</b>
Exporting a Filter to the MATLAB Workspace .....	<b>6-28</b>
<b>Accessing Filter Parameters</b> .....	<b>6-29</b>
Accessing Filter Parameters in a Saved Filter .....	<b>6-29</b>
Accessing Parameters in a Saved Spectrum .....	<b>6-30</b>
<b>Importing Filters and Spectra</b> .....	<b>6-31</b>
Importing Filters .....	<b>6-31</b>
Importing Spectra .....	<b>6-34</b>
<b>Loading Variables from the Disk</b> .....	<b>6-35</b>
<b>Saving and Loading Sessions</b> .....	<b>6-36</b>
Filter Formats .....	<b>6-36</b>
<b>Selecting Signals, Filters, and Spectra</b> .....	<b>6-38</b>
<b>Editing Signals, Filters, or Spectra</b> .....	<b>6-39</b>
<b>Making Signal Measurements with Markers</b> .....	<b>6-40</b>
<b>Setting Preferences</b> .....	<b>6-42</b>
Setting the Filter Design Tool .....	<b>6-44</b>
<b>Using the Filter Designer</b> .....	<b>6-46</b>
Filter Types .....	<b>6-46</b>
FIR Filter Methods .....	<b>6-46</b>
IIR Filter Methods .....	<b>6-46</b>
Pole/Zero Editor .....	<b>6-46</b>
Spectral Overlay Feature .....	<b>6-47</b>
Opening the Filter Designer .....	<b>6-47</b>
Accessing Filter Parameters in a Saved Filter .....	<b>6-49</b>
Designing a Filter with the Pole/Zero Editor .....	<b>6-53</b>
Positioning Poles and Zeros .....	<b>6-54</b>
Redesigning a Filter Using the Magnitude Plot .....	<b>6-55</b>

## Function Reference

---

# 7

<b>Functions — By Category</b> .....	7-2
Digital Filters .....	7-3
Analog Filters .....	7-6
Linear Systems .....	7-8
Windows .....	7-10
Transforms .....	7-12
Cepstral Analysis .....	7-12
Statistical Signal Processing .....	7-12
Parametric Modeling .....	7-14
Linear Prediction .....	7-14
Multirate Signal Processing .....	7-16
Waveform Generation .....	7-16
Specialized Operations .....	7-16
Graphical User Interfaces .....	7-17
<b>Functions — Alphabetical List</b> .....	7-18

## Technical Conventions

---

# A

## Index

---

# Signal Processing Basics

---

The following chapter describes how to begin using MATLAB<sup>®</sup> and the Signal Processing Toolbox for your signal processing applications. It is assumed that you have basic knowledge and understanding of signals and systems, including such topics as filter and linear system theory and basic Fourier analysis.

What Is the Signal Processing Toolbox? (p. 1-2)	Major features and key areas of the toolbox
Representing Signals (p. 1-5)	Vector and matrix representation of signals
Waveform Generation: Time Vectors and Sinusoids (p. 1-7)	Periodic and aperiodic waveforms, sequences (impulse, step, ramp), multichannel signals, pulse trains, sinc and Dirichlet functions
Working with Data (p. 1-14)	Methods of inputting and importing data
Filter Implementation and Analysis (p. 1-15)	Filtering discrete signals
The filter Function (p. 1-18)	Mathematical information on the filter function
Other Functions for Filtering (p. 1-20)	Other types of filter functions available in the toolbox
Impulse Response (p. 1-24)	Impulse response details
Frequency Response (p. 1-26)	Frequency response details
Zero-Pole Analysis (p. 1-33)	Z-plane poles and zeros
Linear System Models (p. 1-35)	Discrete-time and continuous-time linear system models and transformations
Discrete Fourier Transform (p. 1-47)	DFT details
Selected Bibliography (p. 1-50)	Sources for additional information

## What Is the Signal Processing Toolbox?

The Signal Processing Toolbox is a collection of tools built on the MATLAB numeric computing environment. The toolbox supports a wide range of signal processing operations, from waveform generation to filter design and implementation, parametric modeling, and spectral analysis. The toolbox provides two categories of tools:

Command-line functions in the following categories:

- Analog and digital filter analysis
- Digital filter implementation
- FIR and IIR digital filter design
- Analog filter design
- Filter discretization
- Spectral Windows Transforms
- Cepstral analysis
- Statistical signal processing and spectral analysis
- Parametric modeling
- Linear Prediction
- Waveform generation

A suite of interactive graphical user interfaces for

- Filter design and analysis
- Window design and analysis
- Signal plotting and analysis
- Spectral analysis
- Filtering signals

### Signal Processing Toolbox Central Features

The Signal Processing Toolbox functions are algorithms, expressed mostly in M-files, that implement a variety of signal processing tasks. These toolbox functions are a specialized extension of the MATLAB computational and graphical environment.

## Filtering and FFTs

Two of the most important functions for signal processing are not in the Signal Processing Toolbox at all, but are built-in MATLAB functions:

- `filter` applies a digital filter to a data sequence.
- `fft` calculates the discrete Fourier transform of a sequence.

The operations these functions perform are the main computational workhorses of classical signal processing. Both are described in this chapter. The Signal Processing Toolbox uses many other standard MATLAB functions and language features, including polynomial root finding, complex arithmetic, matrix inversion and manipulation, and graphics tools.

## Signals and Systems

The basic entities that toolbox functions work with are signals and systems. The functions emphasize digital, or discrete, signals and filters, as opposed to analog, or continuous, signals. The principal filter type the toolbox supports is the linear, time-invariant digital filter with a single input and a single output. You can represent linear time-invariant systems using one of several models (such as transfer function, state-space, zero-pole-gain, and second-order section) and convert between representations.

## Key Areas: Filter Design and Spectral Analysis

In addition to its core functions, the toolbox provides rich, customizable support for the key areas of filter design and spectral analysis. It is easy to implement a design technique that suits your application, design digital filters directly, or create analog prototypes and discretize them. Toolbox functions also estimate power spectral density and cross spectral density, using either parametric or nonparametric techniques. Chapter 2, “Filter Design and Implementation” and Chapter 3, “Statistical Signal Processing,” respectively detail toolbox functions for filter design and spectral analysis.

Some filter design and spectral analysis functions included in the toolbox are

- Computation and graphical display of frequency response
- System identification
- Generating signals
- Discrete cosine, chirp-z, and Hilbert transforms
- Lattice filters
- Resampling
- Time-frequency analysis
- Basic communication systems simulation

## **Interactive Tools**

The power of the Signal Processing Toolbox is greatly enhanced by its easy-to-use interactive tools. SPTool provides a rich graphical environment for signal viewing, filter design, and spectral analysis. The Filter Design and Analysis Tool (FDATool) provides a more comprehensive collection of features for addressing the problem of filter design. The FDATool also offers seamless access to the additional filter design methods and quantization features of the Filter Design Toolbox when that product is installed. The Window Design and Analysis Tool (WinTool) provides an environment for designing and comparing spectral windows.

## **Extensibility**

Perhaps the most important feature of the MATLAB environment is that it is extensible. MATLAB lets you create your own M-files to meet numeric computation needs for research, design, or engineering of signal processing systems. Simply copy the M-files provided with the Signal Processing Toolbox and modify them as needed, or create new functions to expand the functionality of the toolbox.



## Representing Signals

The central data construct in MATLAB is the *numeric array*, an ordered collection of real or complex numeric data with two or more dimensions. The basic data objects of signal processing (one-dimensional signals or sequences, multichannel signals, and two-dimensional signals) are all naturally suited to array representation.

### Vector Representation

MATLAB represents ordinary one-dimensional sampled data signals, or sequences, as *vectors*. Vectors are 1-by- $n$  or  $n$ -by-1 arrays, where  $n$  is the number of samples in the sequence. One way to introduce a sequence into MATLAB is to enter it as a list of elements at the command prompt. The statement

```
x = [4 3 7 -9 1]
```

creates a simple five-element real sequence in a row vector. Transposition turns the sequence into a column vector

```
x = x'
```

resulting in

```
x =
     4
     3
     7
    -9
     1
```

Column orientation is preferable for single channel signals because it extends naturally to the multichannel case. For multichannel data, each column of a matrix represents one channel. Each row of such a matrix then corresponds to a sample point. A three-channel signal that consists of  $x$ ,  $2x$ , and  $x/\pi$  is

```
y = [x 2*x x/pi]
```

This results in

y =

4.0000	8.0000	1.2732
3.0000	6.0000	0.9549
7.0000	14.0000	2.2282
-9.0000	-18.0000	-2.8648
1.0000	2.0000	0.3183

## Waveform Generation: Time Vectors and Sinusoids

A variety of toolbox functions generate waveforms. Most require you to begin with a vector representing a time base. Consider generating data with a 1000 Hz sample frequency, for example. An appropriate time vector is

```
t = (0:0.001:1)';
```

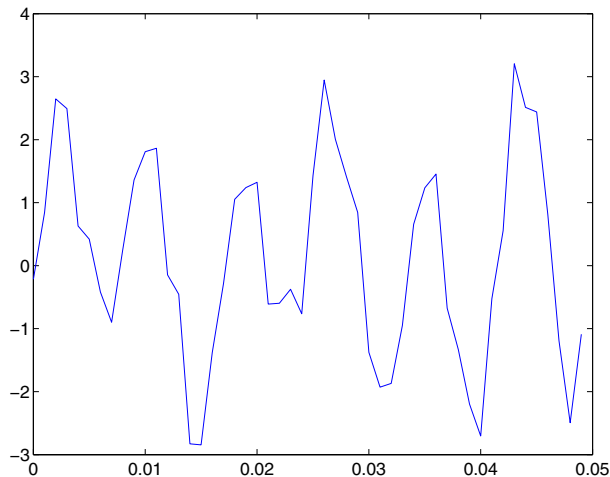
where the MATLAB colon operator creates a 1001-element row vector that represents time running from zero to one second in steps of one millisecond. The transpose operator ( ' ) changes the row vector into a column; the semicolon ( ; ) tells MATLAB to compute but not display the result.

Given  $t$  you can create a sample signal  $y$  consisting of two sinusoids, one at 50 Hz and one at 120 Hz with twice the amplitude.

```
y = sin(2*pi*50*t) + 2*sin(2*pi*120*t);
```

The new variable  $y$ , formed from vector  $t$ , is also 1001 elements long. You can add normally distributed white noise to the signal and graph the first fifty points using

```
randn('state',0);  
yn = y + 0.5*randn(size(t));  
plot(t(1:50),yn(1:50))
```



## Common Sequences: Unit Impulse, Unit Step, and Unit Ramp

Since MATLAB is a programming language, an endless variety of different signals is possible. Here are some statements that generate several commonly used sequences, including the unit impulse, unit step, and unit ramp functions:

```
t = (0:0.001:1)';  
y = [1; zeros(99,1)]; % impulse  
y = ones(100,1);      % step (filter assumes 0 initial cond.)  
y = t;                % ramp  
y = t.^2;  
y = square(4*t);
```

All of these sequences are column vectors. The last three inherit their shapes from `t`.

## Multichannel Signals

Use standard MATLAB array syntax to work with multichannel signals. For example, a multichannel signal consisting of the last three signals generated above is

```
z = [t t.^2 square(4*t)];
```

You can generate a multichannel unit sample function using the outer product operator. For example, a six-element column vector whose first element is one, and whose remaining five elements are zeros, is

```
a = [1 zeros(1,5)]';
```

To duplicate column vector `a` into a matrix without performing any multiplication, use the MATLAB colon operator and the ones function:

```
c = a(:,ones(1,3));
```

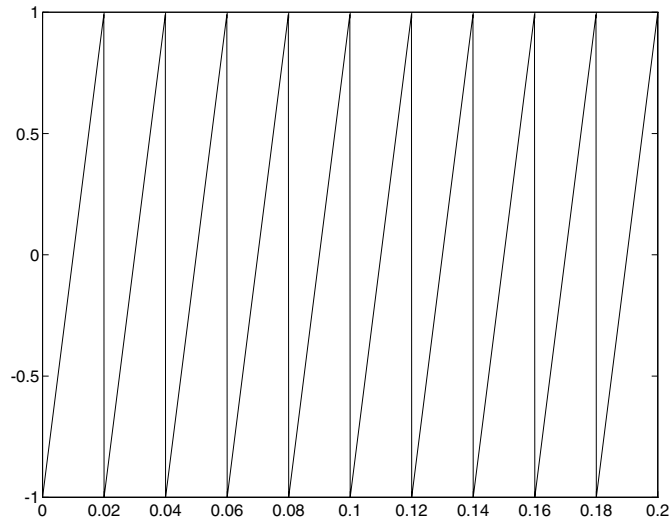
## Common Periodic Waveforms

The toolbox provides functions for generating widely used periodic waveforms:

- `sawtooth` generates a sawtooth wave with peaks at  $\pm 1$  and a period of  $2\pi$ . An optional width parameter specifies a fractional multiple of  $2\pi$  at which the signal's maximum occurs.
- `square` 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.

To generate 1.5 seconds of a 50 Hz sawtooth wave with a sample rate of 10 kHz and plot 0.2 seconds of the generated waveform, use

```
fs = 10000;  
t = 0:1/fs:1.5;  
x = sawtooth(2*pi*50*t);  
plot(t,x), axis([0 0.2 -1 1])
```



## Common Aperiodic Waveforms

The toolbox also provides functions for generating several widely used aperiodic waveforms:

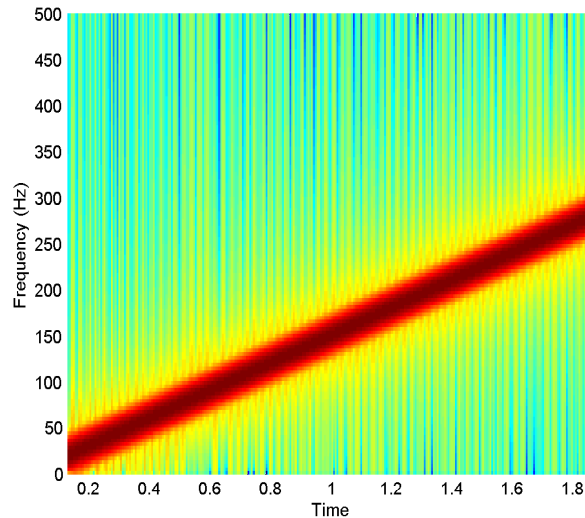
- `gauspuls` generates a Gaussian-modulated sinusoidal pulse with a specified time, center frequency, and fractional bandwidth. Optional parameters return in-phase and quadrature pulses, the RF signal envelope, and the cutoff time for the trailing pulse envelope.
- `chirp` generates a linear swept-frequency cosine signal. An optional parameter specifies alternative sweep methods. An optional parameter `phi` allows initial phase to be specified in degrees.

To compute 2 seconds of a linear chirp signal with a sample rate of 1 kHz, that starts at DC and crosses 150 Hz at 1 second, use

```
t = 0:1/1000:2;  
y = chirp(t,0,1,150);
```

To plot the spectrogram, use

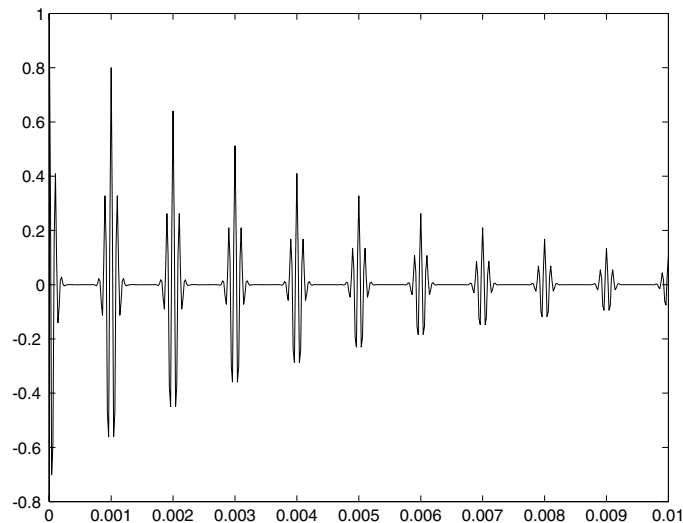
```
spectrogram(y,256,250,256,1000,'yaxis')
```



## The pulstran Function

The `pulstran` function generates pulse trains from either continuous or sampled prototype pulses. The following example generates a pulse train consisting of the sum of multiple delayed interpolations of a Gaussian pulse. The pulse train is defined to have a sample rate of 50 kHz, a pulse train length of 10 ms, and a pulse repetition rate of 1 kHz; `D` specifies the delay to each pulse repetition in column 1 and an optional attenuation for each repetition in column 2. The pulse train is constructed by passing the name of the `gauspuls` function to `pulstran`, along with additional parameters that specify a 10 kHz Gaussian pulse with 50% bandwidth:

```
T = 0:1/50E3:10E-3;  
D = [0:1/1E3:10E-3;0.8.^(0:10)]';  
Y = pulstran(T,D,'gauspuls',10E3,0.5);  
plot(T,Y)
```



## The Sinc Function

The `sinc` function computes the mathematical sinc function for an input vector or matrix  $x$ . The sinc function is the continuous inverse Fourier transform of the rectangular pulse of width  $2\pi$  and height 1.

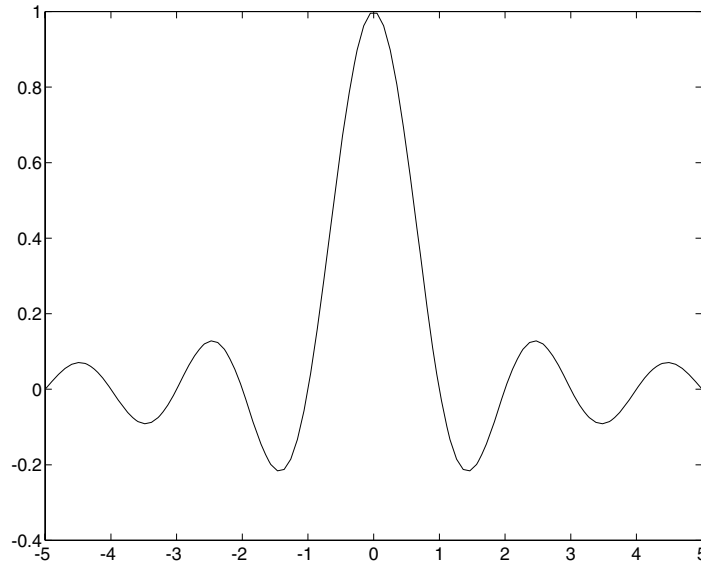
The sinc function has a value of 1 where  $x$  is zero, and a value of

$$\frac{\sin(\pi x)}{\pi x}$$

for all other elements of  $x$ .

To plot the sinc function for a linearly spaced vector with values ranging from -5 to 5, use the following commands:

```
x = linspace(-5,5);  
y = sinc(x);  
plot(x,y)
```





## The Dirichlet Function

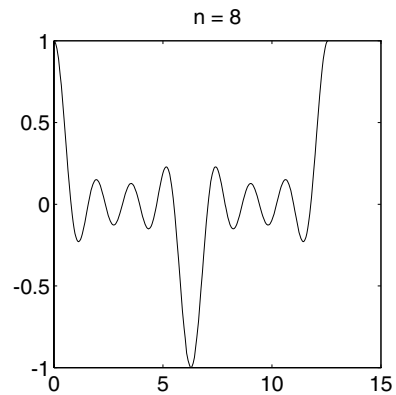
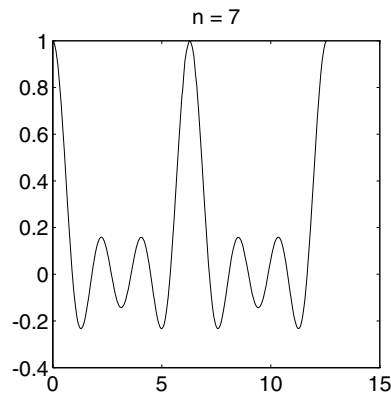
The toolbox function `diric` computes the Dirichlet function, sometimes called the *periodic sinc* or *aliased sinc* function, for an input vector or matrix  $x$ . The Dirichlet function is

$$\text{diric}(x) = \begin{cases} -1^{k(n-1)} & x = 2\pi k, k = 0, \pm 1, \pm 2, \dots \\ \frac{\sin(nx/2)}{n \sin(x/2)} & \text{otherwise} \end{cases}$$

where  $n$  is a user-specified positive integer. For  $n$  odd, the Dirichlet function has a period of  $2\pi$ ; for  $n$  even, its period is  $4\pi$ . The magnitude of this function is  $(1/n)$  times the magnitude of the discrete-time Fourier transform of the  $n$ -point rectangular window.

To plot the Dirichlet function over the range 0 to  $4\pi$  for  $n = 7$  and  $n = 8$ , use

```
x = linspace(0,4*pi,300);
plot(x,diric(x,7)); axis tight;
plot(x,diric(x,8)); axis tight;
```



## Working with Data

The examples in the preceding sections obtain data in one of two ways:

- By direct input, that is, entering the data manually at the keyboard
- By using a MATLAB or toolbox function, such as `sin`, `cos`, `sawtooth`, `square`, or `sinc`

Some applications, however, may need to import data from outside MATLAB. Depending on your data format, you can do this in the following ways:

- Load data from an ASCII file or MAT-file with the MATLAB `load` command.
- Read the data into MATLAB with a low-level file I/O function, such as `fopen`, `fread`, and `fscanf`.
- Develop a MEX-file to read the data.

Other resources are also useful, such as a high-level language program (in Fortran or C, for example) that converts your data into MAT-file format — see the MATLAB External Interfaces/API Reference documentation for details. MATLAB reads such files using the `load` command.

Similar techniques are available for exporting data generated within MATLAB. See the MATLAB documentation for more details on importing and exporting data.

---

**Note** All Signal Processing Toolbox functions accept double-precision inputs. The Filter Design Toolbox, along with the Fixed-Point Toolbox, enables single-precision floating-point and fixed-point support for most `dfilt` structures.

---

## Filter Implementation and Analysis

This section describes how to filter discrete signals using the MATLAB `filter` function and other functions in the Signal Processing Toolbox. It also discusses how to use the toolbox functions to analyze filter characteristics, including impulse response, magnitude and phase response, group delay, and zero-pole locations.

### Convolution and Filtering

The mathematical foundation of filtering is convolution. The MATLAB `conv` function performs standard one-dimensional convolution, convolving one vector with another:

```
conv([1 1 1],[1 1 1])
ans =
     1     2     3     2     1
```

---

**Note** Convolve rectangular matrices for two-dimensional signal processing using the `conv2` function.

---

A digital filter's output  $y(k)$  is related to its input  $x(k)$  by convolution with its impulse response  $h(k)$ .

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

If a digital filter's impulse response  $h(k)$  is finite length, and the input  $x(k)$  is also finite length, you can implement the filter using `conv`. Store  $x(k)$  in a vector `x`,  $h(k)$  in a vector `h`, and convolve the two:

```
x = randn(5,1); % A random vector of length 5
h = [1 1 1 1]/4; % Length 4 averaging filter
y = conv(h,x);
```

## Filters and Transfer Functions

In general, the  $z$ -transform  $Y(z)$  of a digital filter's output  $y(n)$  is related to the  $z$ -transform  $X(z)$  of the input by

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

where  $H(z)$  is the filter's *transfer function*. Here, the constants  $b(i)$  and  $a(i)$  are the filter coefficients and the order of the filter is the maximum of  $n$  and  $m$ .

---

**Note** The filter coefficients start with subscript 1, rather than 0. This reflects the standard indexing scheme used for vectors in MATLAB.

---

MATLAB stores the coefficients in two vectors, one for the numerator and one for the denominator. By convention, MATLAB uses row vectors for filter coefficients.

### Filter Coefficients and Filter Names

Many standard names for filters reflect the number of  $a$  and  $b$  coefficients present:

- When  $n = 0$  (that is,  $b$  is a scalar), the filter is an Infinite Impulse Response (IIR), all-pole, recursive, or autoregressive (AR) filter.
- When  $m = 0$  (that is,  $a$  is a scalar), the filter is a Finite Impulse Response (FIR), all-zero, nonrecursive, or moving-average (MA) filter.
- If both  $n$  and  $m$  are greater than zero, the filter is an IIR, pole-zero, recursive, or autoregressive moving-average (ARMA) filter.

The acronyms AR, MA, and ARMA are usually applied to filters associated with filtered stochastic processes.

## Filtering with the filter Function

It is simple to work back to a difference equation from the  $z$ -transform relation shown earlier. Assume that  $a(1) = 1$ . Move the denominator to the left-hand side and take the inverse  $z$ -transform.

$$y(k) + a_2y(k-1) + \cdots + a_{m+1}y(k-m) = b_1x(k) + b_2x(k-1) + \cdots + b_{n+1}x(k-m)$$

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

$$y(k) = b_1x(k) + b_2x(k-1) + \cdots + b_{n+1}x(k-n) - a_2y(k-1) - \cdots - a_{m+1}y(k-n)$$

This is the standard time-domain representation of a digital filter, computed starting with  $y(1)$  and assuming zero initial conditions. This representation's progression is

$$\begin{aligned} y(1) &= b_1x(1) \\ y(2) &= b_1x(2) + b_2x(1) - a_2y(1) \\ y(3) &= b_1x(3) + b_2x(2) + b_3x(1) - a_2y(2) - a_3y(1) \\ &\vdots = \vdots \end{aligned}$$

A filter in this form is easy to implement with the `filter` function. For example, a simple single-pole filter (lowpass) is

```
b = 1;           % Numerator
a = [1 -0.9];   % Denominator
```

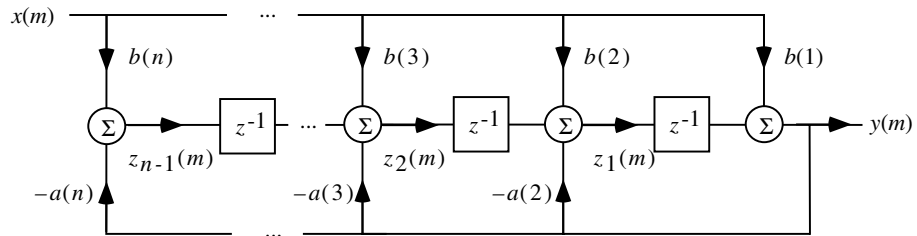
where the vectors `b` and `a` represent the coefficients of a filter in transfer function form. To apply this filter to your data, use

```
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, `filter` divides the coefficients by `a(1)` before implementing the difference equation.

## 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

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

In its most basic form, filter initializes the delay outputs  $z_i(1)$ ,  $i = 1, \dots, n-1$  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:

$$[y, zf] = \text{filter}(b, a, x, zi)$$

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,  $x_1$ , corresponds to the first 10 minutes of data and the second,  $x_2$ , to an additional 10 minutes. The whole sequence is  $x = [x_1; x_2]$ . If there is not sufficient memory to hold the combined sequence, filter the subsequences  $x_1$  and  $x_2$  one at a time. To ensure continuity of the filtered sequences, use the final conditions from  $x_1$  as initial conditions to filter  $x_2$ :

```
[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  $z_f$  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.

## Other Functions for Filtering

In addition to `filter`, several other functions in the Signal Processing Toolbox perform the basic filtering operation. These functions include `upfirdn`, which performs FIR filtering with resampling, `filtfilt`, which eliminates phase distortion in the filtering process, `fftfilt`, which performs the FIR filtering operation in the frequency domain, and `latcfilt`, which filters using a lattice implementation.

### Multirate Filter Bank Implementation

The function `upfirdn` 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

$$\begin{aligned} d &= \text{gcd}(48000, 44100); \\ p &= 48000/d; \\ q &= 44100/d; \end{aligned}$$

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

$$y = \text{upfirdn}(x, h, p, q)$$

This cascade of operations is implemented in an efficient manner using polyphase filtering techniques, and it is a central concept of multirate filtering (see reference [1] for details on multirate filter theory). Note that the quality of the resampling result relies on the quality of the FIR filter  $h$ .



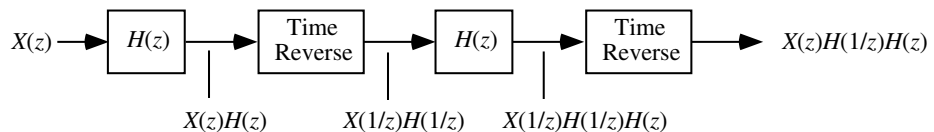
Filter banks may be implemented using `upfirdn` by allowing the filter `h` to be a matrix, with one FIR filter per column. A signal vector is passed independently through each FIR filter, resulting in a matrix of output signals.

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

## 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  $X(z)$ , the  $z$ -transform of the time reversed sequence  $x(n)$  is  $X(1/z)$ . Consider the processing scheme.



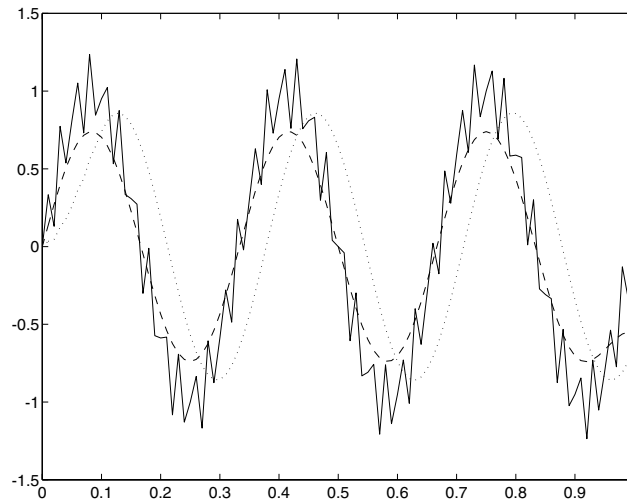
When  $|z| = 1$ , that is  $z = e^{j\omega}$ , the output reduces to  $X(e^{j\omega}) |H(e^{j\omega})|^2$ . Given all the samples of the sequence  $x(n)$ , a doubly filtered version of  $x$  that has zero-phase distortion is possible.

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 10-point averaging FIR filter, and filter  $x$  using both `filter` and `filtfilt` for comparison:

```
b = ones(1,10)/10;           % 10 point averaging filter
y = filtfilt(b,1,x);         % Noncausal filtering
yy = filter(b,1,x);          % Normal filtering
plot(t,x,t,y,'--',t,yy,':')
```



Both filtered versions eliminate the 40 Hz sinusoid evident in the original, solid line. The plot also shows how `filter` and `filtfilt` differ; the dashed (`filtfilt`) line is in phase with the original 3 Hz sinusoid, while the dotted (`filter`) line is delayed by about five samples. Also, the amplitude of the dashed line is smaller due to the magnitude squared effects of `filtfilt`.

`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.

## 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 (see reference [1] at the end of this chapter) to filter a long sequence with multiple medium-length FFTs. Its output is equivalent to `filter(b,1,x)`.

## Impulse Response

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

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

In MATLAB, you can generate an impulse sequence a number of ways; one straightforward way is


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

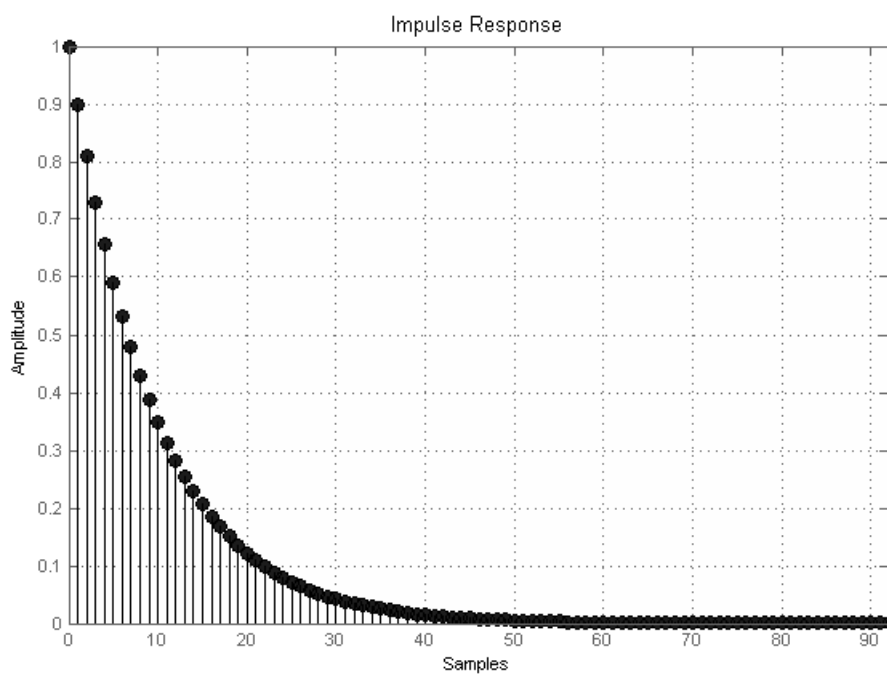
The impulse response of the simple filter  $b = 1$  and  $a = [1 \ -0.9]$  is

```
h = filter(b,a,imp);
```

A simple way to display the impulse response is with the Filter Visualization Tool (fvtool):

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

Then click the Impulse Response button  on the toolbar or select **Impulse Response** from the **Analysis** menu. This plot shows the exponential decay  $h(n) = 0.9^n$  of the single pole system:



## Frequency Response

The Signal Processing Toolbox enables you to perform frequency domain analysis of both analog and digital filters.

### 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.

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

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);
```

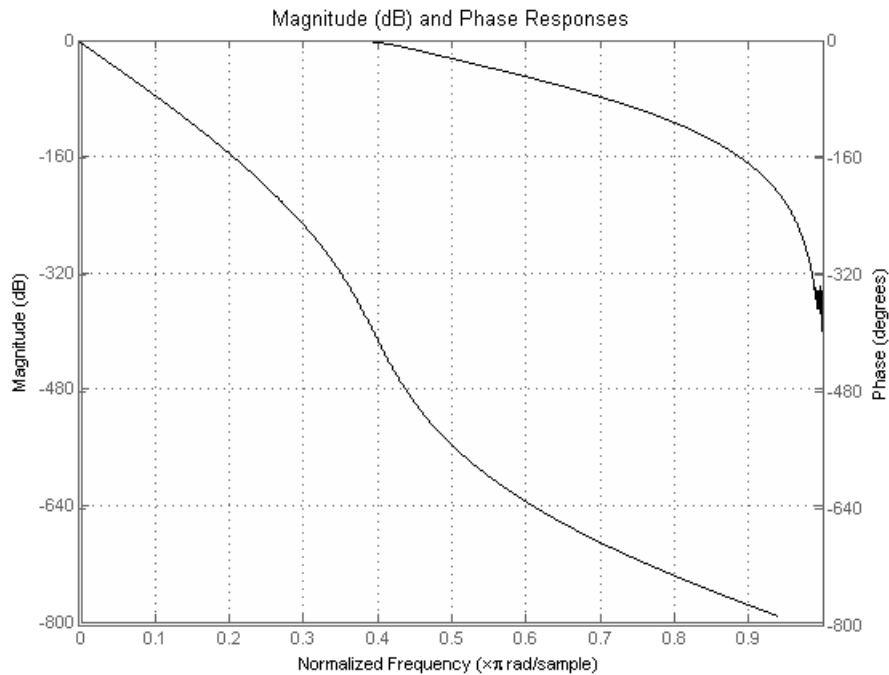
To calculate the 256-point complex frequency response for this filter, and plot the magnitude and phase with `freqz`, use

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

or to display the magnitude and phase responses in `fvtool`, which provides additional analysis tools, use

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

and click the Magnitude and Phase Response button  on the toolbar or select **Magnitude and Phase Response** from the **Analysis** menu.



`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.

## 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.

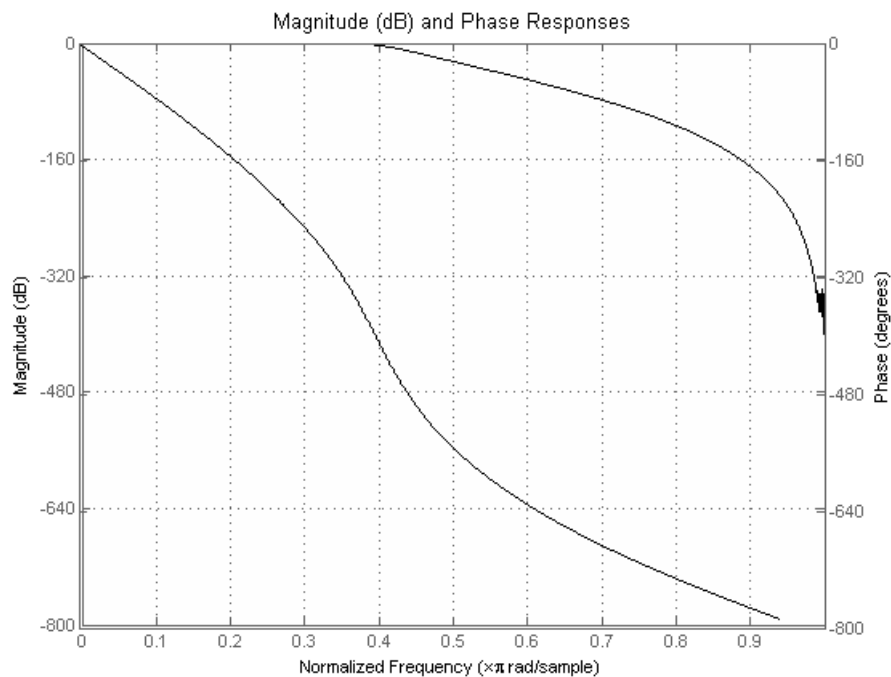


## Magnitude and Phase

MATLAB provides functions to extract magnitude and phase from a frequency response vector  $h$ . The function `abs` returns the magnitude of the response; `angle` returns the phase angle in radians. To extract the magnitude and phase of a Butterworth filter:

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

and click the Magnitude and Phase Response button  on the toolbar or select **Magnitude and Phase Response** from the **Analysis** menu to display the plot.

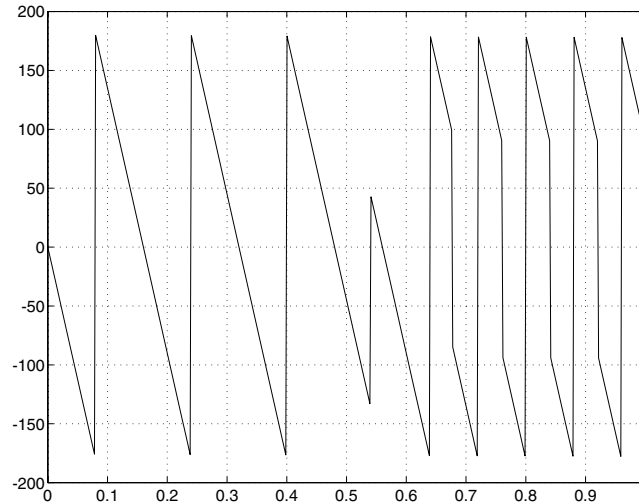


The `unwrap` function is also useful in frequency analysis. `unwrap` unwraps the phase to make it continuous across  $360^\circ$  phase discontinuities by adding multiples of  $\pm 360^\circ$ , as needed. To see how `unwrap` is useful, design a 25th-order lowpass FIR filter:

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

Obtain the filter's 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  $360^\circ$  jumps (an artifact of the arctangent function inside `angle`) from the  $180^\circ$  jumps that signify zeros in the frequency response.

`unwrap` eliminates the  $360^\circ$  jumps:

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

or you can use `phasez` to see the unwrapped phase.

## Delay

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

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

where  $\theta$  is the phase angle of  $H(e^{j\omega})$ . Compute group delay with

```
[gd,w] = grpdelay(b,a,n)
```

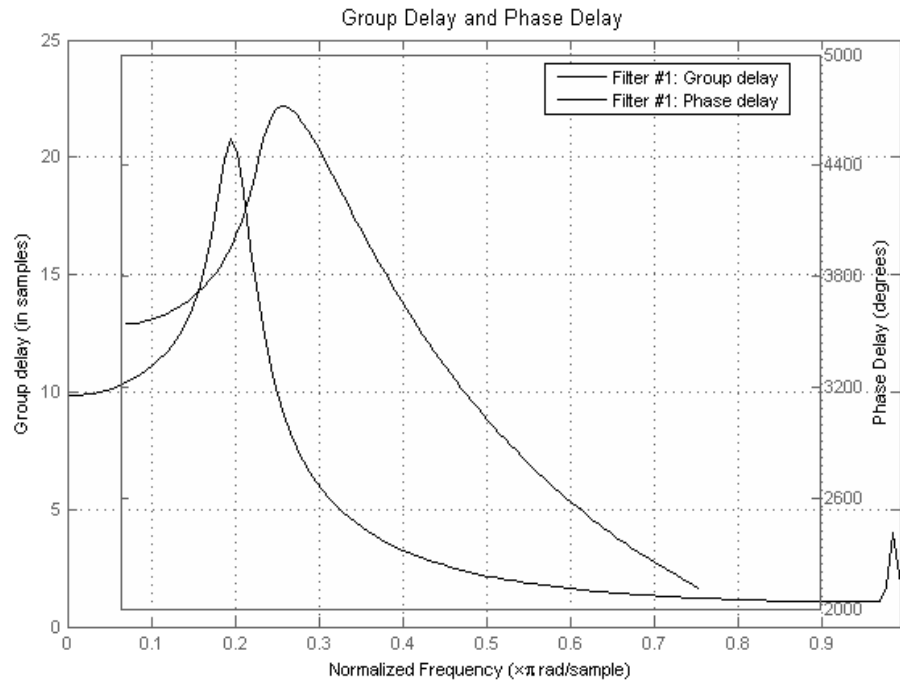
which returns the n-point group delay,  $\tau_g(\omega)$ , of the digital filter specified by b and a, evaluated at the frequencies in vector w.

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

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

To plot both the group and phase delays of a system on the same FVTool graph, type

```
[b,a] = butter(10,200/1000);
hFVT = fvtool(b,a,'Analysis','grpdelay');
set(hFVT,'NumberOfPoints',128,'OverlaidAnalysis','phasedelay');
legend(hFVT)
```



## 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]');
```

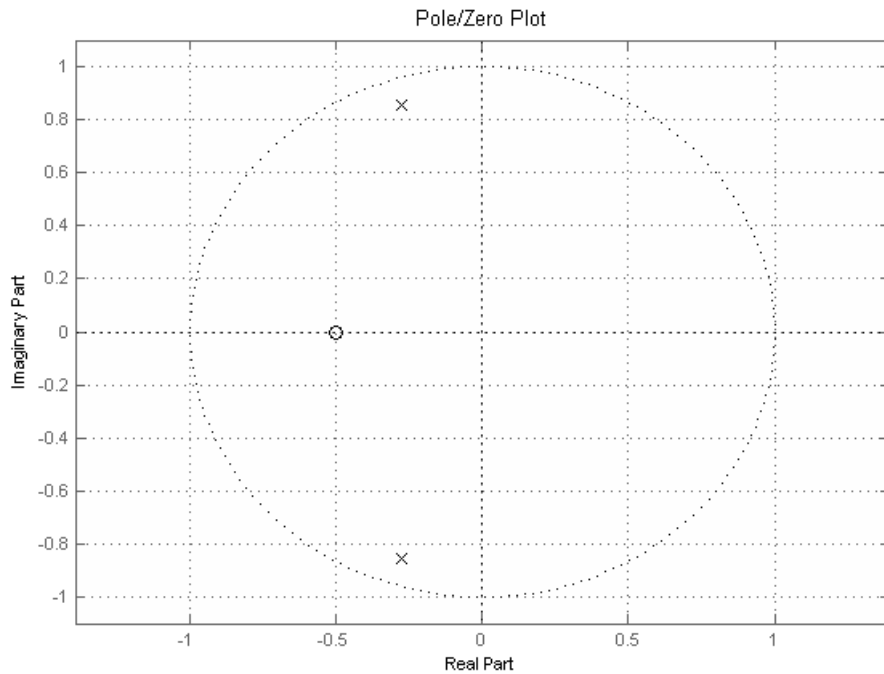
To view the pole-zero plot for this filter you can use

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

or, 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)
```

and click the Pole/Zero Plot toolbar button  on the toolbar or select **Pole/Zero Plot** from the **Analysis** menu to see the plot.



For a system in zero-pole form, supply column vector arguments  $z$  and  $p$  to `zplane`:

```
zplane(z, p)
```

For a system in transfer function form, supply row vectors  $b$  and  $a$  as arguments to `zplane`:

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

In this case `zplane` finds the roots of  $b$  and  $a$  using the `roots` function and plots the resulting zeros and poles.

See “Linear System Models” on page 1-35 for details on zero-pole and transfer function representation of systems.

## Linear System Models

The Signal Processing Toolbox provides several models for representing linear time-invariant systems. This flexibility lets you choose the representational scheme that best suits your application and, within the bounds of numeric stability, convert freely to and from most other models. This section provides a brief overview of supported linear system models and describes how to work with these models in MATLAB.

### Discrete-Time System Models

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

- “Transfer Function” on page 1-35
- “Zero-Pole-Gain” on page 1-36
- “State-Space” on page 1-37
- “Partial Fraction Expansion (Residue Form)” on page 1-38
- “Second-Order Sections (SOS)” on page 1-40
- “Lattice Structure” on page 1-40
- “Convolution Matrix” on page 1-43

### 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

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

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 MATLAB, 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 1-16 for more details on the transfer function form.

## Zero-Pole-Gain

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

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

By convention, MATLAB stores polynomial coefficients 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)  
  
q =  
-0.7500 + 1.1990i  
-0.7500 - 1.1990i  
  
p = roots(a)  
  
p =  
-1.0000  
-1.0000 + 0.0000i  
-1.0000 - 0.0000i  
  
k = b(1)/a(1)  
  
k =  
2
```



Returning to the original polynomials,

```
bb = k*poly(q)
bb =
    2.0000    3.0000    4.0000
aa = poly(p)
aa =
    1.0000    3.0000    3.0000    1.0000
```

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

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

For  $b = [2 \ 3 \ 4]$ , the roots function misses the zero for  $z$  equal to 0. In fact, it 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. In MATLAB,  $A$ ,  $B$ ,  $C$ , and  $D$  are rectangular arrays; MATLAB treats 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:

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

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, consult the Control System Toolbox 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

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

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:

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

The `residuez` function in the Signal Processing Toolbox 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)

r =
    -12
     8
p =
    -4
    -2
k =
    []
```

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:

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

To verify this in MATLAB, type

```
imp = [1 0 0 0 0];
resptf = filter(b,a,imp)

resptf =
    -4    32   -160    704   -2944
respres = filter(r(1),[1 -p(1)],imp) + filter(r(2),[1 -p(2)],imp)

respres =
    -4    32   -160    704   -2944
```

## 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. MATLAB 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.

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

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 1-45, perform pole-zero pairing, section scaling, and section ordering.

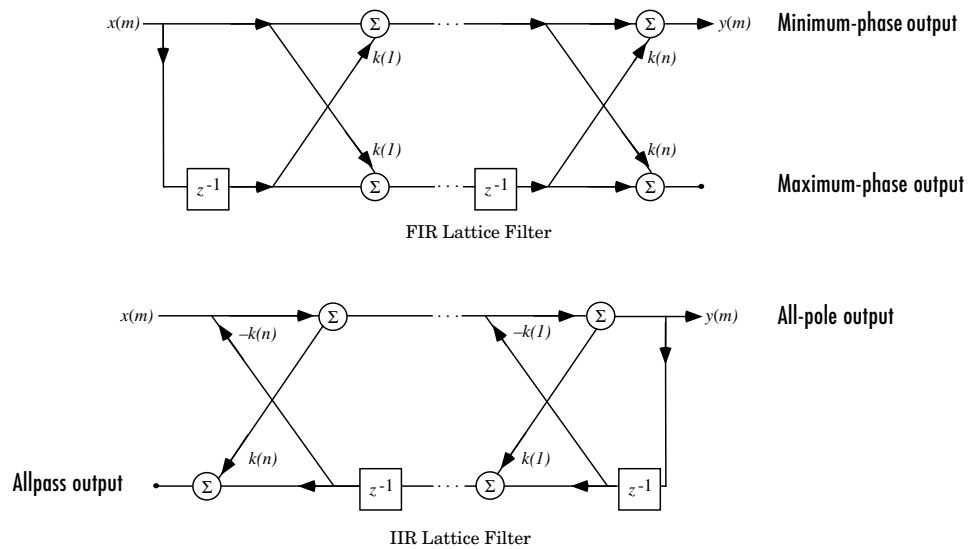
---

**Note** In the Signal Processing Toolbox, all second-order section transformations apply only to digital filters.

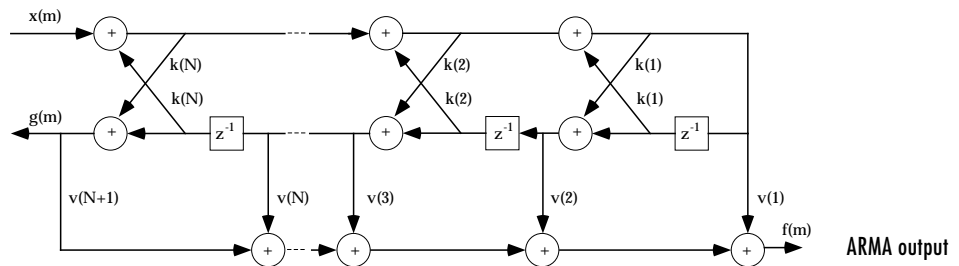
---

## Lattice Structure

For a discrete  $N$ th order all-pole or all-zero filter described by the polynomial coefficients  $a(n)$ ,  $n = 1, 2, \dots, N+1$ , there are  $N$  corresponding lattice structure coefficients  $k(n)$ ,  $n = 1, 2, \dots, N$ . 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.



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



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

$$b = [1.0000 \quad 0.6149 \quad 0.9899 \quad 0.0000 \quad 0.0031 \quad -0.0082];$$

This filter's lattice (reflection coefficient) representation is

$$k = \text{tf2latc}(b)$$

k =

0.3090  
0.9801  
0.0031  
0.0081  
-0.0082

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

$$[k, v] = \text{tf2latc}(b, a)$$

k =

0.3750  
0.3333  
0  
0  
0

v =

0.6252  
0.1212  
0.9879  
-0.0009  
0.0072  
-0.0082

Because  $\text{abs}(k) < 1$  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  $k$  (above), the corresponding polynomial form is

```
b = latc2tf(k)
b =
    1.0000    0.6149    0.9899 -0.0000    0.0031 -0.0082
```

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)
C =
     1     0     0
     2     1     0
     3     2     1
     0     3     2
     0     0     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

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

where  $u$  is a vector of  $nu$  inputs,  $x$  is an  $nx$ -element state vector, and  $y$  is a vector of  $ny$  outputs. In MATLAB, store A, B, C, and D 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

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

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

The factored zero-pole-gain form is

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



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

## Linear System Transformations

The Signal Processing Toolbox provides a number of functions that 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.

	Transfer Function	State-Space	Zero-Pole-Gain	Partial Fraction	Lattice Filter	Second-Order Sections	Convolution Matrix
Transfer Function		tf2ss	tf2zp 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** In the Signal Processing Toolbox, all 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 Signal Processing Toolbox 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.

MATLAB 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

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

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

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

$$W_N = e^{-j\left(\frac{2\pi}{N}\right)}$$

---

**Note** MATLAB uses 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:

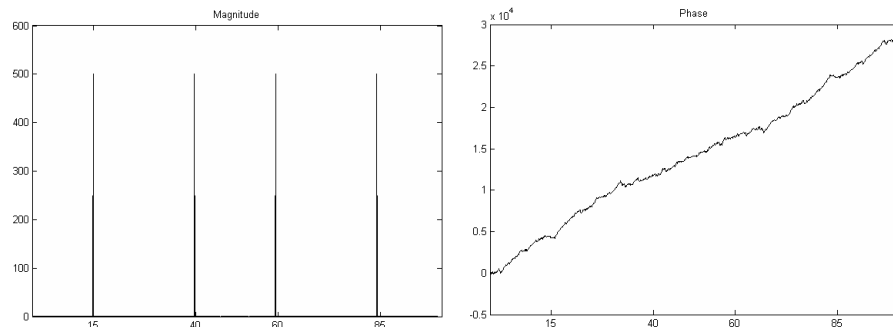
```
t = (0:1/100:10-1/100);           % Time vector
x = sin(2*pi*15*t) + sin(2*pi*40*t); % Signal
```

The DFT of the signal, and the magnitude and phase of the transformed sequence, are then

```
y = fft(x);                       % Compute DFT of x
m = abs(y); p = unwrap(angle(y)); % Magnitude and phase
```

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

```
f = (0:length(y)-1)*99/length(y); % Frequency vector
plot(f,m); title('Magnitude');
set(gca,'XTick',[15 40 60 85]);
figure; plot(f,p*180/pi); title('Phase');
set(gca,'XTick',[15 40 60 85]);
```



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

```
y = fft(x,n);
```

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 in the MATLAB documentation 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)));
```

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 MATLAB function `fftshift` moves the zero frequency component to the center of a vector or matrix.

## Selected Bibliography

Algorithm development for the Signal Processing Toolbox has drawn heavily upon the references listed below. All are recommended to the interested reader who needs to know more about signal processing than is covered in this manual.

- [1] Crochiere, R.E., and L.R. Rabiner. *Multi-Rate Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1983. Pgs. 88-91.
- [2] IEEE. *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979.
- [3] Jackson, L.B. *Digital Filters and Signal Processing*. Third Ed. Boston: Kluwer Academic Publishers, 1989.
- [4] Kay, S.M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice Hall, 1988.
- [5] Oppenheim, A.V., and R.W. Schaffer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.
- [6] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987.
- [7] Percival, D.B., and A.T. Walden. *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques*. Cambridge: Cambridge University Press, 1993.
- [8] Pratt, W.K. *Digital Image Processing*. New York: John Wiley & Sons, 1991.
- [9] Proakis, J.G., and D.G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. Upper Saddle River, NJ: Prentice Hall, 1996.
- [10] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975.
- [11] Welch, P.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 (June 1967). Pgs. 70-73.

# Filter Design and Implementation

---

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 the Signal Processing Toolbox.

Filter Requirements and Specification (p. 2-2)	Overview of filter design
IIR Filter Design (p. 2-4)	Infinite impulse response filters (Butterworth, Chebyshev, elliptic, Bessel, Yule-Walker, and parametric methods)
FIR Filter Design (p. 2-17)	Finite impulse response filters (windowing, multiband, least squares, nonlinear phase, complex filters, raised cosine)
Special Topics in IIR Filter Design (p. 2-42)	Analog design, frequency transformation, filter discretization
Filter Implementation (p. 2-52)	Filtering with your filter
Selected Bibliography (p. 2-54)	Sources for additional information

## Filter Requirements and Specification

The goal of filter design is to perform frequency dependent alteration of a data sequence. A possible requirement might be to remove noise above 30 Hz from a data sequence sampled at 100 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. For “loosely specified” requirements, as in the first case above, a Butterworth IIR filter is often sufficient. To design a fifth-order 30 Hz lowpass Butterworth filter and apply it to the data in vector  $x$ :

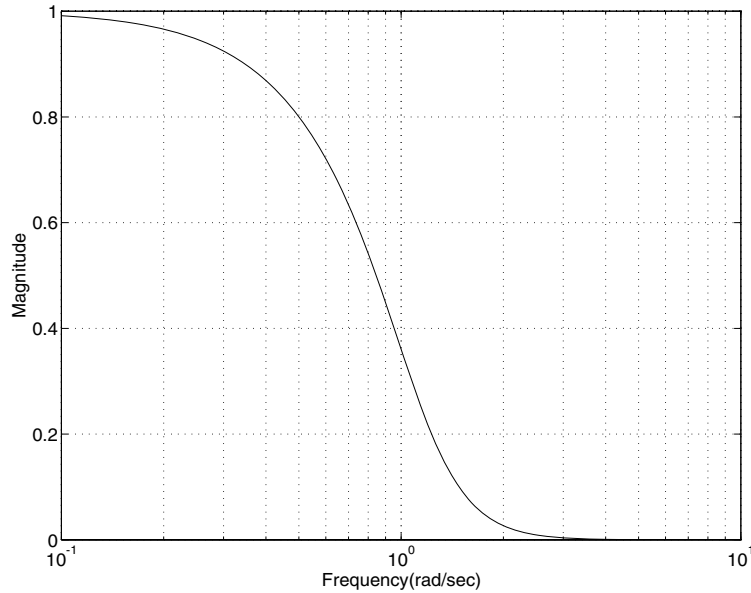
```
[b,a] = butter(5,30/50);  
Hd = dfilt.df2t(b,a);      %Direct-form II transposed structure  
y = filter(Hd,x);
```

The second input argument to `butter` specifies the cutoff frequency, normalized to half the sampling frequency (the Nyquist frequency).

All of the filter design functions operate with normalized frequencies, so they do not require the system sampling rate as an extra input argument. This toolbox uses the convention that unit frequency is the Nyquist frequency, defined as half the sampling frequency. The normalized frequency, therefore, is always in the interval  $0 \leq f \leq 1$ . For a system with a 1000 Hz sampling frequency, 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.

More rigorous filter requirements traditionally include passband ripple ( $R_p$ , in decibels), stopband attenuation ( $R_s$ , in decibels), and transition width ( $W_s - W_p$ , in hertz).





You can design Butterworth, Chebyshev Type I, Chebyshev Type II, and elliptic filters that meet this type of performance specification. The toolbox order selection functions estimate the minimum filter order that meets a given set of requirements.

To meet specifications with more rigid constraints like linear phase or arbitrary filter shape, use the FIR and direct IIR filter design routines.

### IIR Filter Design

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 is commonly performed “off-line,” 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.

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).

The direct filter design function `yulewalk` finds a filter with magnitude response approximating a desired 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 4-15.

The generalized Butterworth design function `maxflat` is discussed in the section “Generalized Butterworth Filter Design” on page 2-15.

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

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

### 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 2-42 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 Type	Design Function
Bessel (analog only)	$[b,a] = \text{besself}(n,Wn,options)$ $[z,p,k] = \text{besself}(n,Wn,options)$ $[A,B,C,D] = \text{besself}(n,Wn,options)$
Butterworth	$[b,a] = \text{butter}(n,Wn,options)$ $[z,p,k] = \text{butter}(n,Wn,options)$ $[A,B,C,D] = \text{butter}(n,Wn,options)$
Chebyshev Type I	$[b,a] = \text{cheby1}(n,Rp,Wn,options)$ $[z,p,k] = \text{cheby1}(n,Rp,Wn,options)$ $[A,B,C,D] = \text{cheby1}(n,Rp,Wn,options)$
Chebyshev Type II	$[b,a] = \text{cheby2}(n,Rs,Wn,options)$ $[z,p,k] = \text{cheby2}(n,Rs,Wn,options)$ $[A,B,C,D] = \text{cheby2}(n,Rs,Wn,options)$
Elliptic	$[b,a] = \text{ellip}(n,Rp,Rs,Wn,options)$ $[z,p,k] = \text{ellip}(n,Rp,Rs,Wn,options)$ $[A,B,C,D] = \text{ellip}(n,Rp,Rs,Wn,options)$

By default, each of these functions returns a lowpass filter; you need only specify the desired cutoff frequency  $Wn$  in normalized frequency (Nyquist frequency = 1 Hz). For a highpass filter, append the string '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, appending the string '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,.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.

---

**Note** All classical IIR lowpass filters are ill-conditioned for extremely low cut-off 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)

n =
    12
Wn =
    0.1951    0.4080

[b,a] = butter(n,Wn);
```

An elliptic filter that meets the same requirements is given by

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

n =
     5
Wn =
    0.2000    0.4000

[b,a] = ellip(n,1,60,Wn);
```

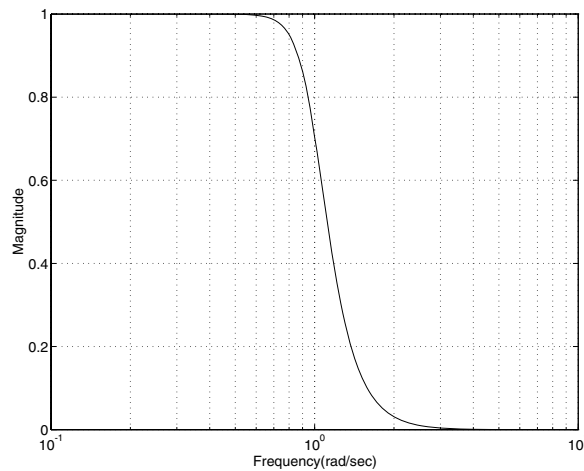
These functions also work with the other standard band configurations, as well as for analog filters.

## 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.

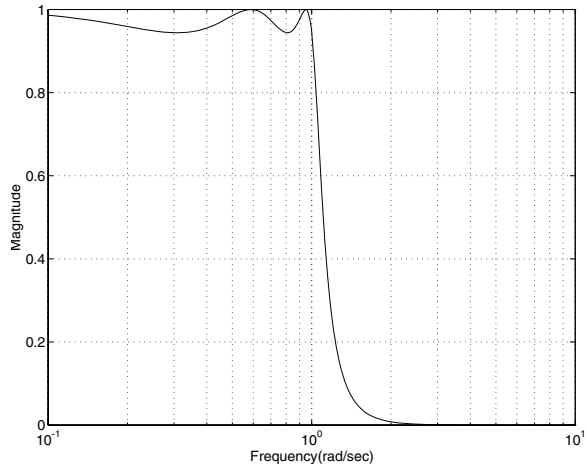
### Butterworth Filter

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  $\Omega = \infty$ .  $|H(j\Omega)| = \sqrt{1/2}$  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  $R_p$  dB in the passband. Stopband response is maximally flat. The transition from passband to stopband is more rapid than for the Butterworth filter.  $|H(j\Omega)| = 10^{-R_p/20}$  at  $\Omega = 1$ .

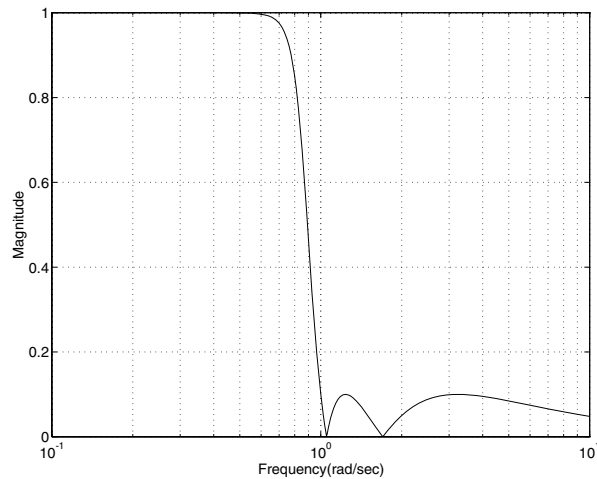




## Chebyshev Type II Filter

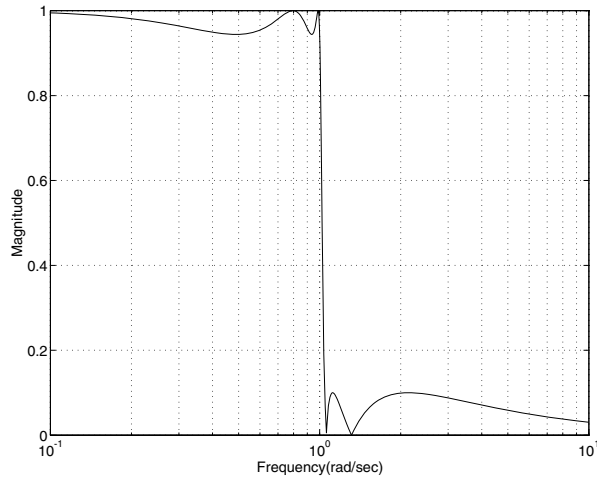
The Chebyshev Type II filter minimizes the absolute difference between the ideal and actual frequency response over the entire stopband by incorporating an equal ripple of  $R_s$  dB in the stopband. Passband response is maximally flat.

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.  $|H(j\Omega)| = 10^{-R_s/20}$  at  $\Omega = 1$ .



### Elliptic Filter

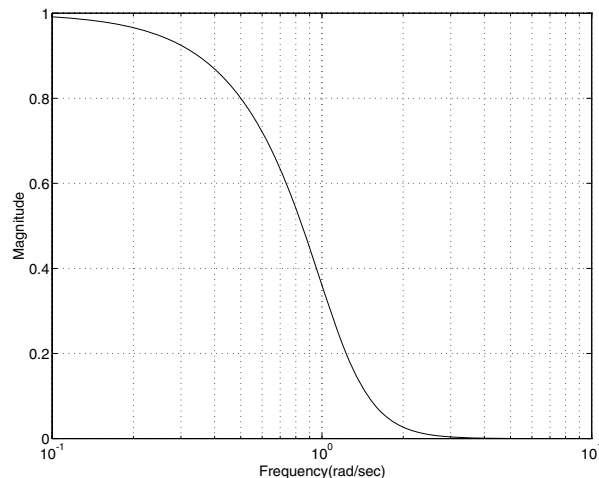
Elliptic filters are equiripple in both the passband and stopband. They generally meet filter requirements with the lowest order of any supported filter type. Given a filter order  $n$ , passband ripple  $R_p$  in decibels, and stopband ripple  $R_s$  in decibels, elliptic filters minimize transition width.  $|H(j\Omega)| = 10^{-R_p/20}$  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. Frequency mapped and digital Bessel filters, however, do not have this maximally flat property; this 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 order  $n$  analog filter of the appropriate type with 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 2-42 for details.

---

To create similar plots, use  $n = 5$  and, as needed,  $R_p = 0.5$  and  $R_s = 20$ . 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
```

### 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 4-15 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 desired magnitude-squared response and solves the “modified Yule-Walker equations” using the resulting autocorrelation function samples. The statement

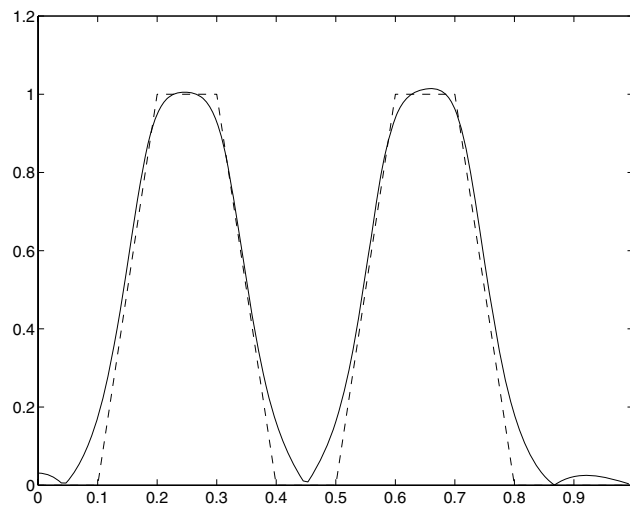
```
[b,a] = yulewalk(n,f,m)
```

returns row vectors `b` and `a` containing the  $n+1$  numerator and denominator coefficients of the order  $n$  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 desired 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 2-17 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 desired 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 desired 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

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.

<b>Filter Design Method</b>	<b>Description</b>	<b>Filter Functions</b>
Windowing	Apply window to truncated inverse Fourier transform of desired “brick wall” filter	<code>fir1</code> , <code>fir2</code> , <code>kaiserord</code>
Multiband with Transition Bands	Equiripple or least squares approach over sub-bands of the frequency range	<code>firls</code> , <code>firpm</code> , <code>firpmord</code>
Constrained Least Squares	Minimize squared integral error over entire frequency range subject to maximum error constraints	<code>fircls</code> , <code>fircls1</code>

Filter Design Method	Description	Filter Functions
Arbitrary Response	Arbitrary responses, including nonlinear phase and complex filters	<code>cfirpm</code>
Raised Cosine	Lowpass response with smooth, sinusoidal transition	<code>firrcos</code>

### 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: $b(k) = b(n+2-k), k = 1, \dots, n+1$	No restriction	No restriction
Type II	Odd		No restriction	$H(1) = 0$
Type III	Even	odd: $b(k) = -b(n+2-k), k = 1, \dots, n+1$	$H(0) = 0$	$H(1) = 0$
Type IV	Odd		$H(0) = 0$	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`, `fircls1`, and `firrcos` all design type I and II linear phase FIR filters by default. 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  $\pi$ . Its impulse response sequence  $h(n)$  is

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

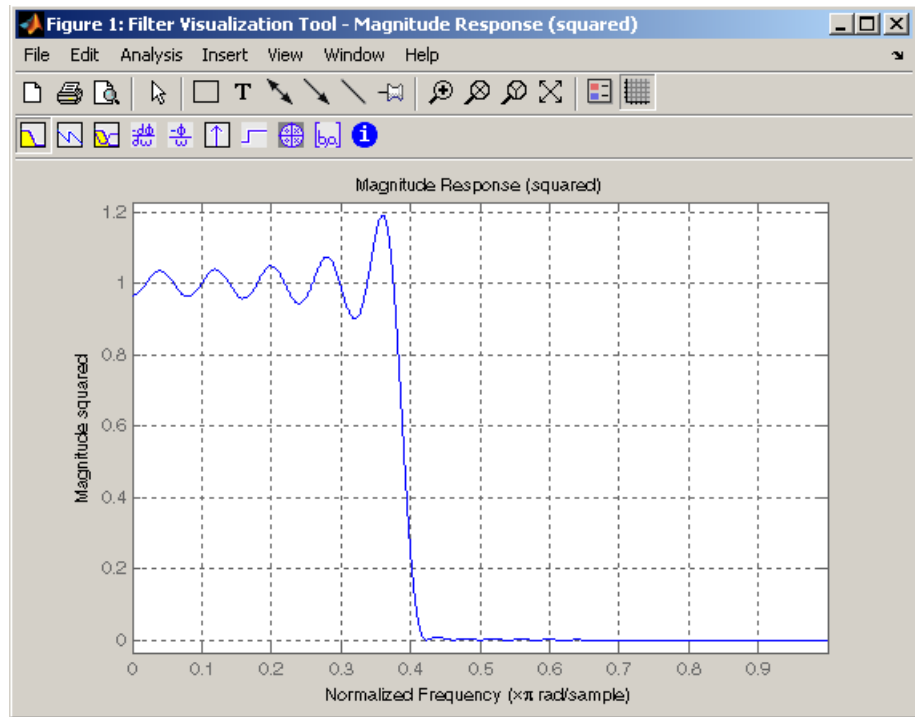
This filter is not implementable since its impulse response is infinite and noncausal. To create a finite-duration impulse response, truncate it by applying a window. By retaining the central section of impulse response in this truncation, you obtain a linear phase FIR filter. For example, a length 51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

$$b = 0.4 * \text{sinc}(0.4 * (-25:25));$$

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:

$$\text{fvtool}(b, 1)$$

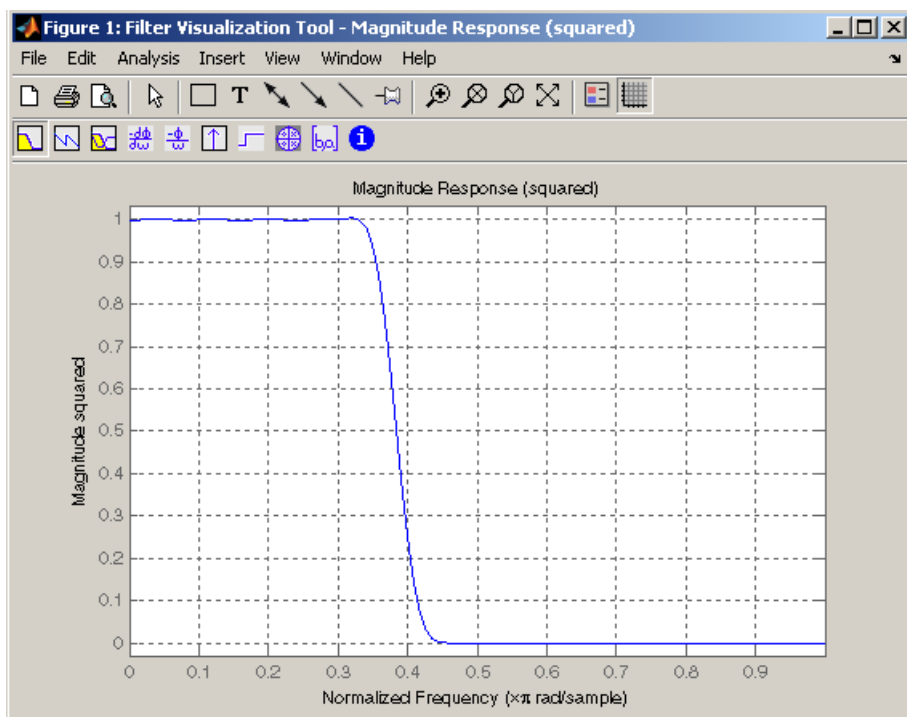
Note that the  $y$ -axis shown in the figure below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.



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).

The functions `fir1` and `fir2` are based on this windowing process. Given a filter order and description of an ideal desired filter, these functions return a windowed inverse Fourier transform of that ideal filter. Both use a Hamming window by default, but they accept any window function. See the “Windows” on page 4-2 for an overview of windows and their properties.

### Standard Band FIR Filter Design: `fir1`

`fir1` implements the classical method of windowed linear phase FIR digital filter design. It 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 the string '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 the string '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 desired 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 desired 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 desired 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 desired and actual frequency responses. The filters are optimal in the sense that they minimize the maximum error between the desired 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 desire 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];        % Desired 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.

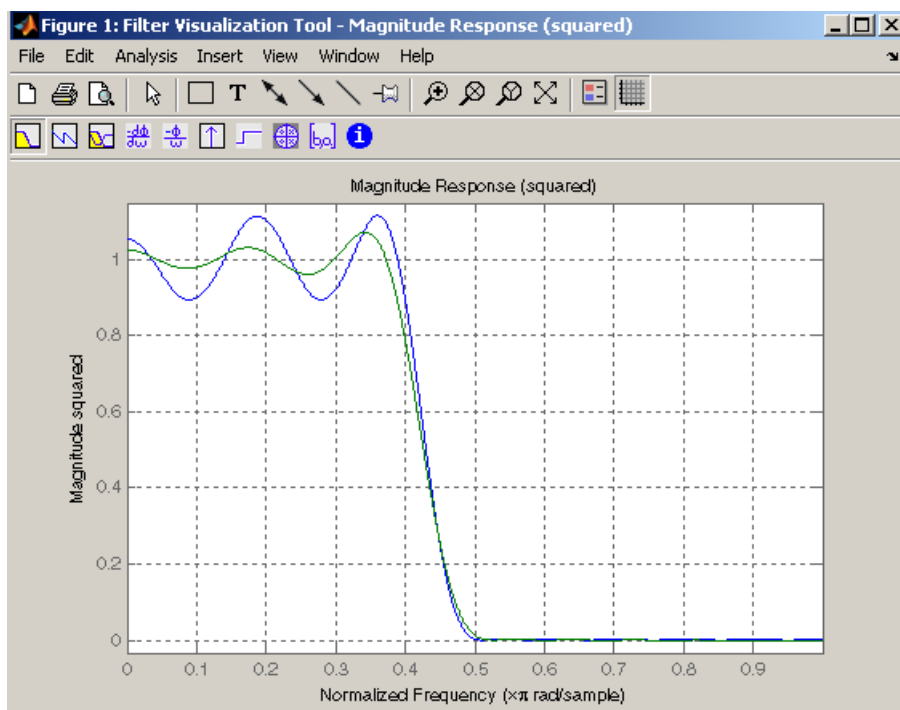
To compare least squares to equiripple filter design, use `firls` to create a similar filter. Type

```
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 desired 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

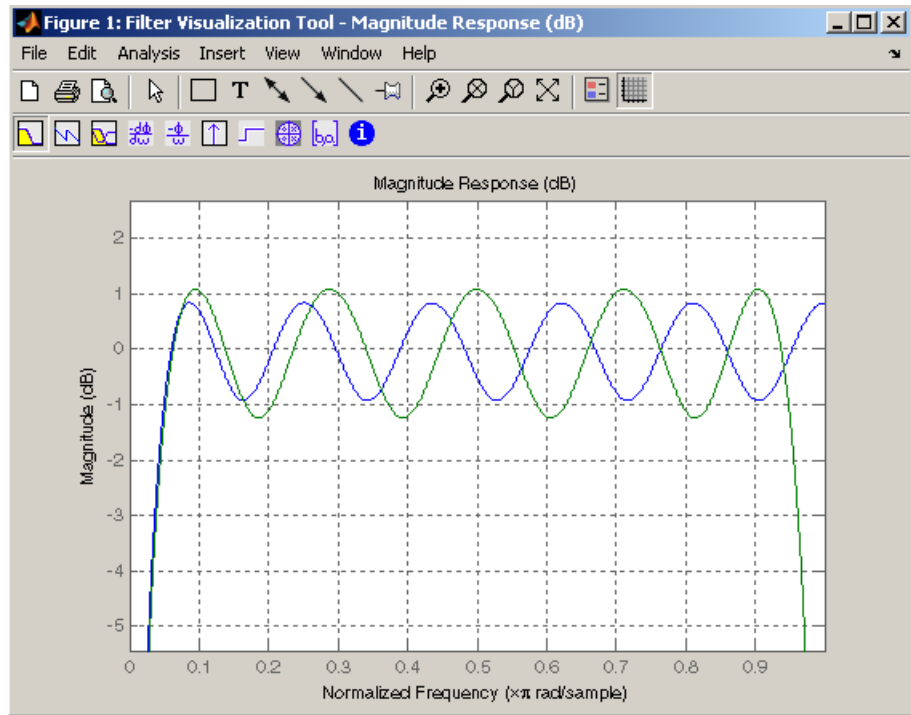
```
n = 20;           % Filter order
f = [0 0.4 0.5 1]; % Frequency band edges
a = [1 1 0 0];   % Desired amplitudes
w = [1 10];      % Weight vector
b = firpm(n,f,a,w);
```

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 “Specialized Transforms” on page 4-42, 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');
```

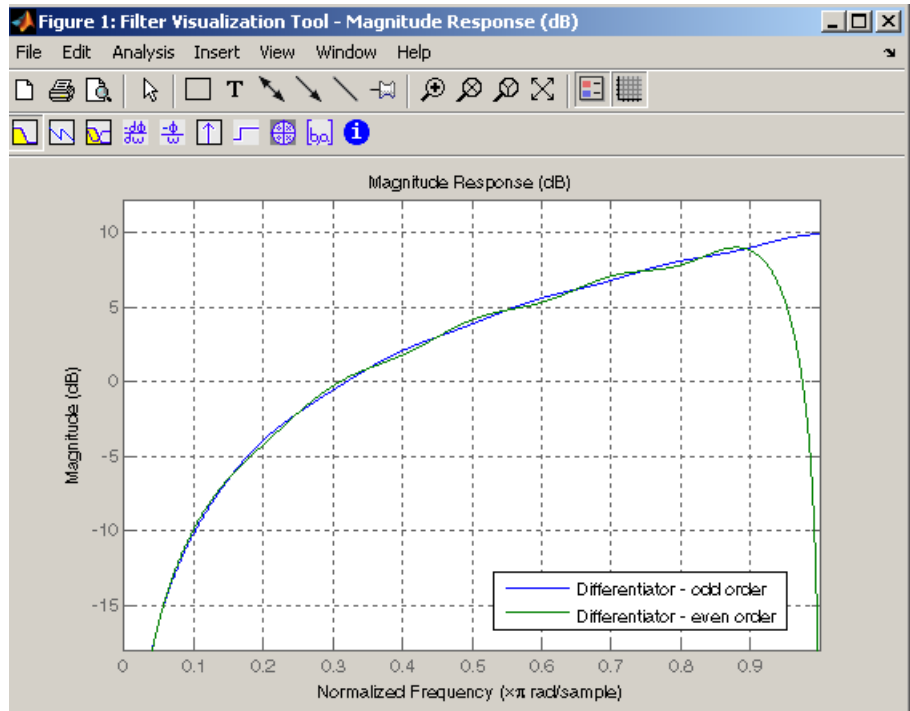
For a type III filter, the differentiation band should stop short of the Nyquist frequency, and the amplitude vector must reflect that change to ensure the correct slope:

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

In the 'd' mode, `firpm` weights the error by  $1/\omega$  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)
```



## 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 desired response. 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 Gibbs' phenomena.

There are two toolbox functions that implement this design technique.

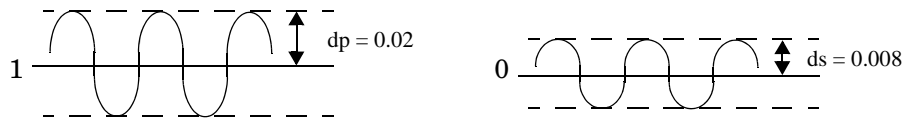
Description	Function
Constrained least square multiband FIR filter design	fircls
Constrained least square filter design for lowpass and highpass linear phase filters	fircls1

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:

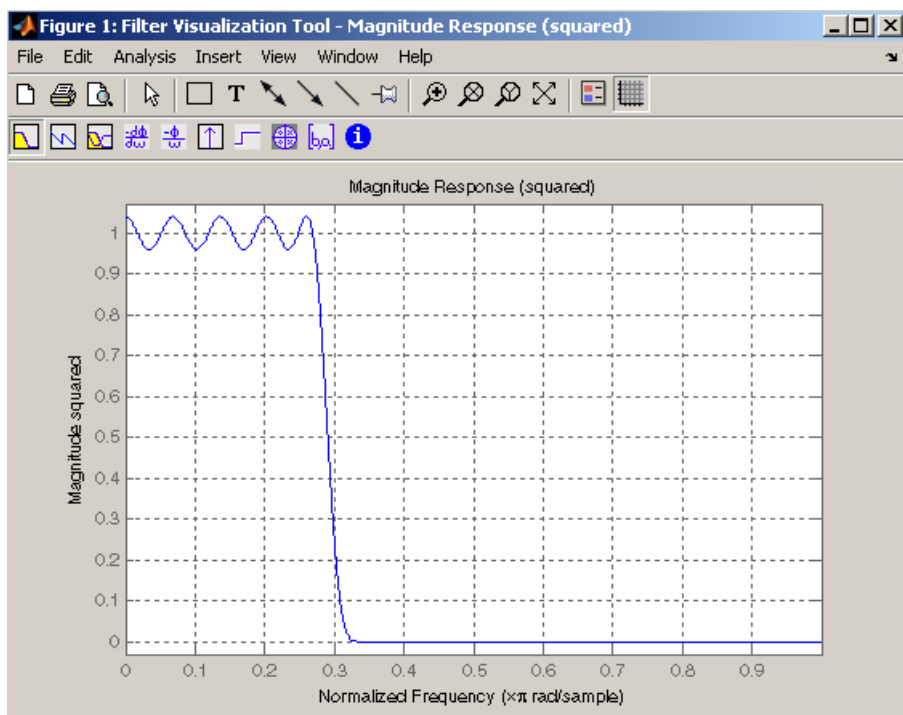
- Maximum passband deviation from 1 (passband ripple) of 0.02.
- Maximum stopband deviation from 0 (stopband ripple) of 0.008.



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 desired 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.

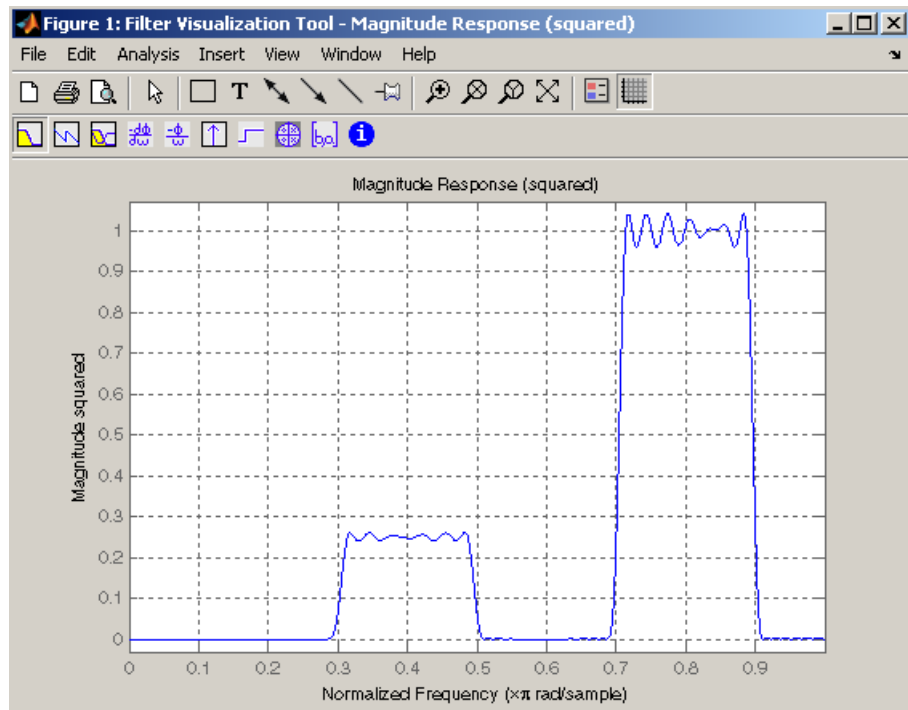
For example, assume the specifications for a filter call for:

- 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.

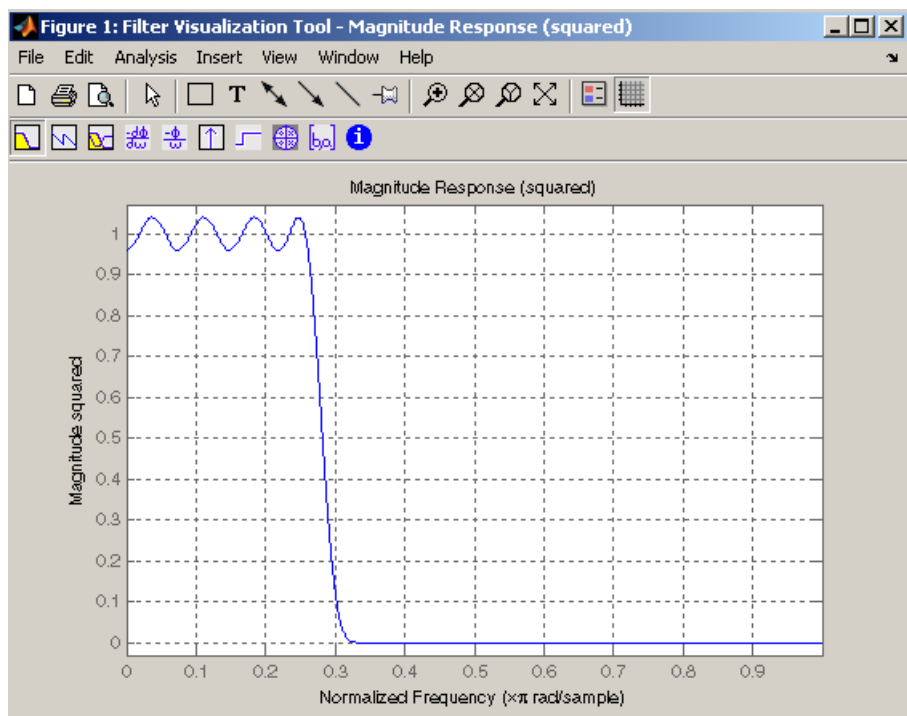
For example, consider specifications that call for an FIR filter with impulse response order of 55 and cutoff frequency of 0.3 (normalized). Also assume maximum allowable passband ripple of 0.02 and maximum allowable stopband ripple of 0.004. In addition, add weighting requirements:

- 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`:

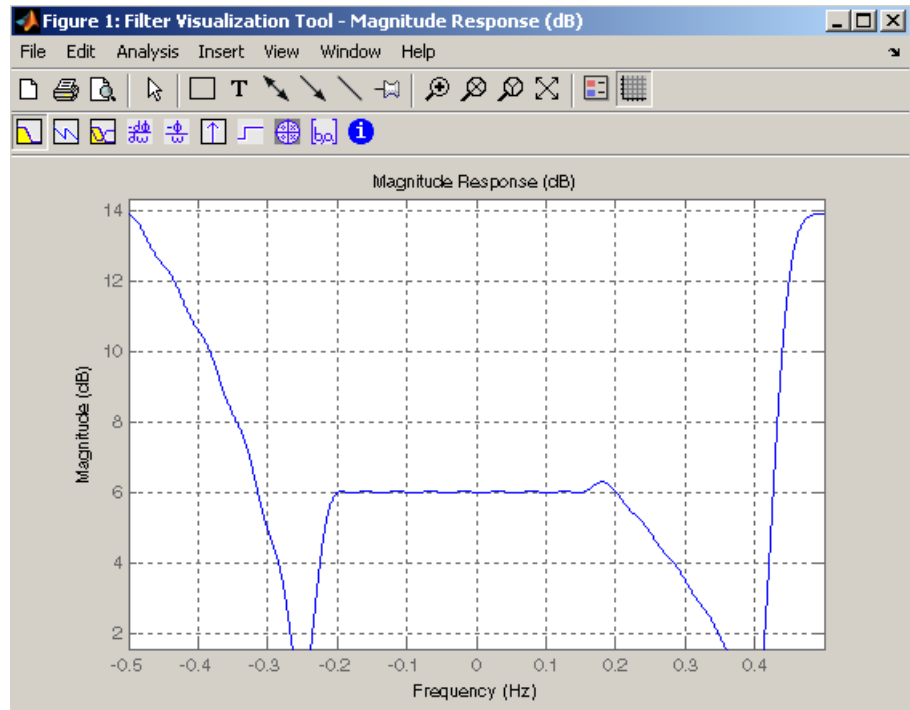
```
b = cfirpm(38,[-1 -0.5 -0.4 0.3 0.4 0.8], ...
           [5 1 2 2 2 1], [1 10 5]);
```

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.

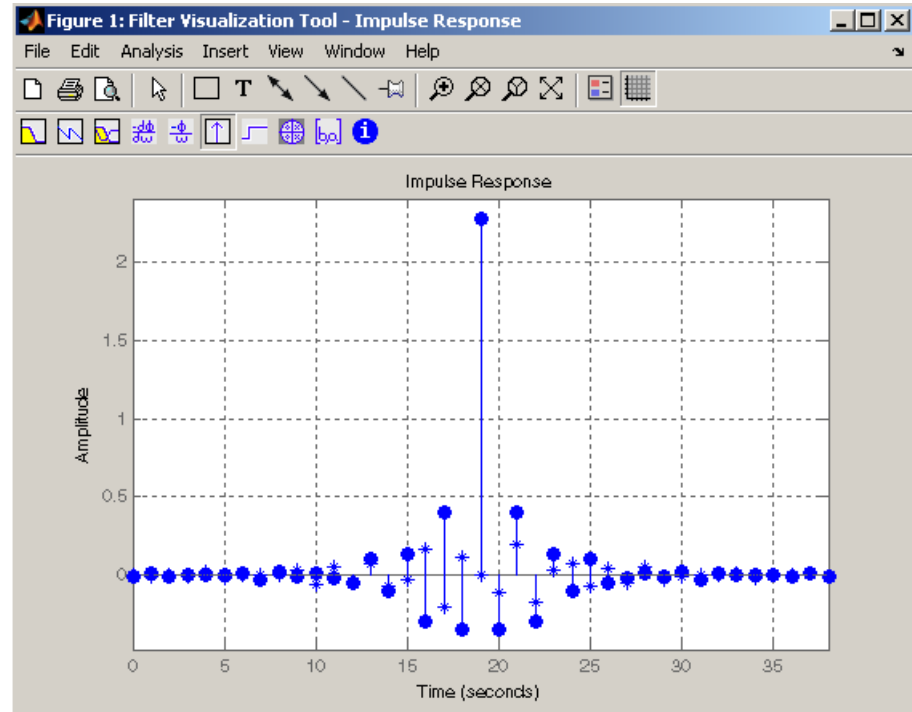
In either case, the frequency response is obtained and plotted using linear scale in FVTool:

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

Note that the range of data shown below is  $(-F_s/2, F_s/2)$ . You can set this range by changing the  $x$ -axis units to **Frequency (Fs = 1 Hz)**.



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

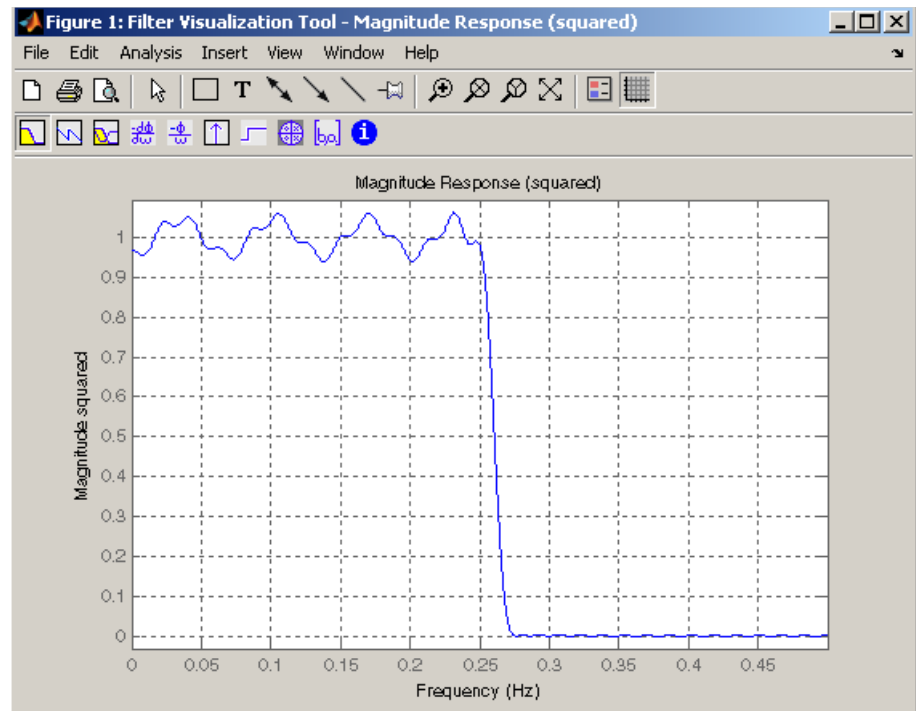
Consider the design of a 62-tap lowpass filter with a half-Nyquist cutoff. If we specify a negative offset value to the `lowpass` filter design function, the group delay offset for the design is significantly less than that obtained for a standard linear-phase design. This filter design may be computed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

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

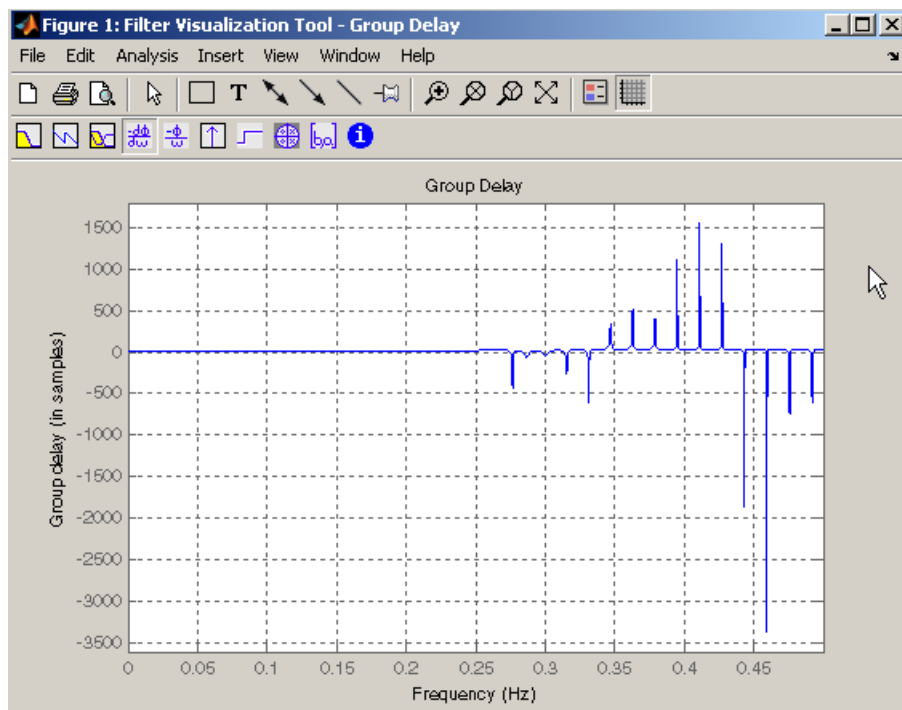
The resulting magnitude response is

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

Note that the range of data in this plot is  $(-F_s/2, F_s/2)$ , which you can set changing the *x*-axis units to **Frequency**. 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** 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  $2 \times 14.5$ , or 29. Using `b = cfilterpm(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

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 desired band configuration
- 2 Transform the filter to the digital domain.
- 3 Discretize the filter.

The toolbox provides functions for each of these steps.

<b>Design Task</b>	<b>Available functions</b>
Analog lowpass prototype	buttap, cheb1ap, besslap, 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	$[z, p, k] = \text{besselap}(n)$
Butterworth	$[z, p, k] = \text{buttap}(n)$
Chebyshev Type I	$[z, p, k] = \text{cheb1ap}(n, R_p)$
Chebyshev Type II	$[z, p, k] = \text{cheb2ap}(n, R_s)$
Elliptic	$[z, p, k] = \text{ellipap}(n, R_p, R_s)$

## 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 of the desired cutoff frequency.

<b>Freq. Transformation</b>	<b>Transformation Function</b>
Lowpass to lowpass $s' = s/\omega_0$	[numt, dent] = lp2lp(num, den, Wo) [At, Bt, Ct, Dt] = lp2lp(A, B, C, D, Wo)
Lowpass to highpass  $s' = \frac{\omega_0}{s}$	[numt, dent] = lp2hp(num, den, Wo) [At, Bt, Ct, Dt] = lp2hp(A, B, C, D, Wo)
Lowpass to bandpass $s' = \frac{\omega_0 (s/\omega_0)^2 + 1}{B_\omega s/\omega_0}$	[numt, dent] = lp2bp(num, den, Wo, Bw) [At, Bt, Ct, Dt] = lp2bp(A, B, C, D, Wo, Bw)
Lowpass to bandstop $s' = \frac{B_\omega s/\omega_0}{\omega_0 (s/\omega_0)^2 + 1}$	[numt, dent] = lp2bs(num, den, Wo, Bw) [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(5,3);
```

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 `lp2bp` 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:

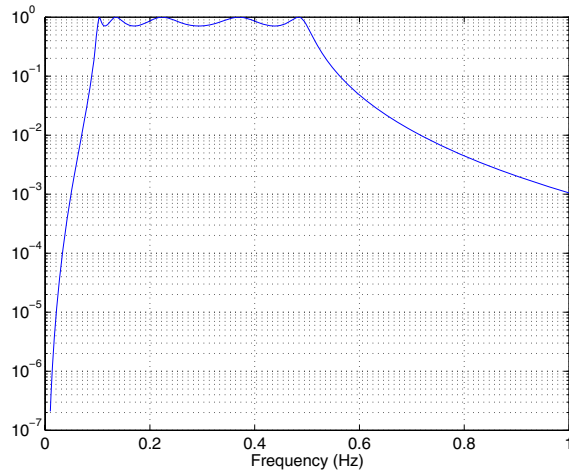
```
[A,B,C,D] = zp2ss(z,p,k); % Convert to state-space form.
```

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)), grid % Plot log magnitude vs. freq.
xlabel('Frequency (Hz)');
```



### 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	<code>[numd,dend] =impinvar(num,den,fs)</code>
Bilinear transform	<code>[zd,pd,kd] = bilinear(z,p,k,fs,Fp)</code> <code>[numd,dend] = bilinear(num,den,fs,Fp)</code> <code>[Ad,Bd,Cd,Dd] = bilinear(At,Bt,Ct,Dt,fs,Fp)</code>

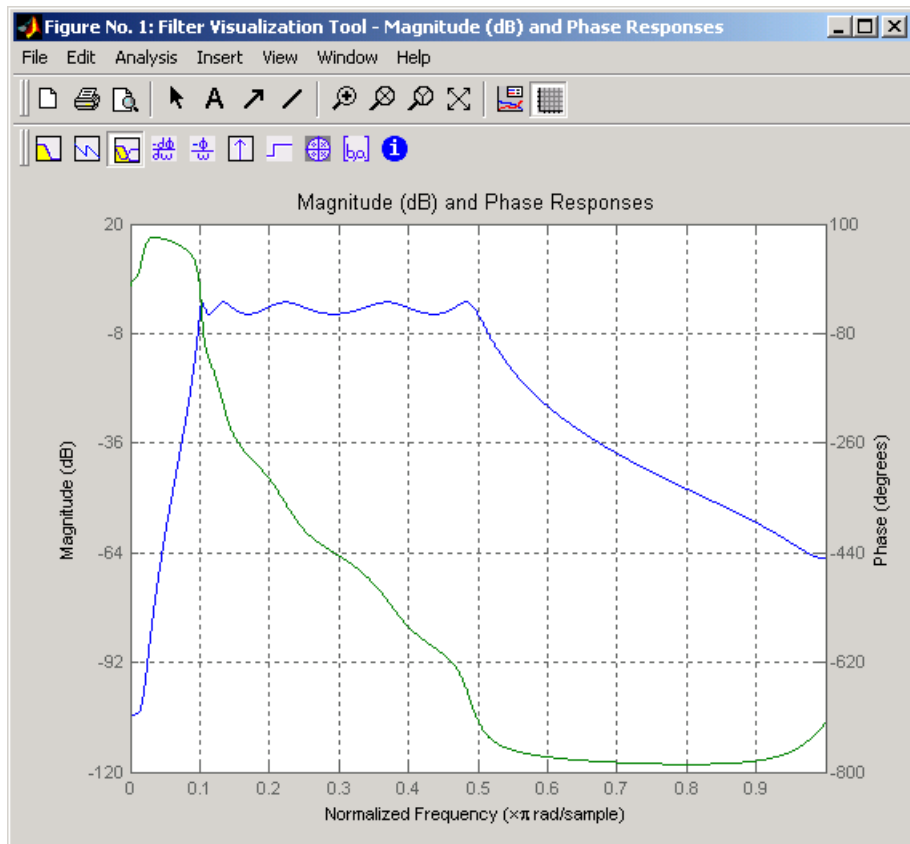
## Impulse Invariance

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 on 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 ( $2 * f_s$ ) 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  $F_p$ . If a match frequency  $F_p$  (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:

$$\omega = 2\pi f_p / f_s \quad (\text{in rad/sample}).$$

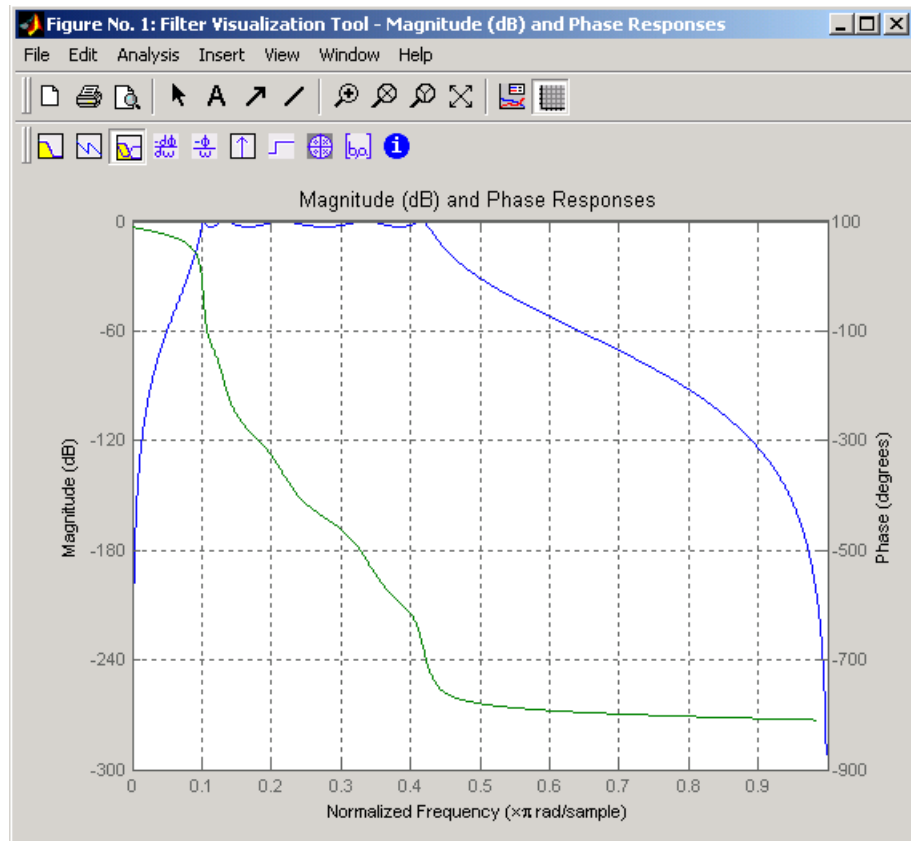
The `bilinear` function can perform this transformation on three different linear system representations: zero-pole-gain, transfer function, and state-space form. Try calling `bilinear` with the state-space matrices that describe the Chebyshev Type I filter from the previous section, using a sampling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

### Filter Implementation

After the filter design process has generated the filter coefficient vectors,  $b$  and  $a$ , two functions are available in the Signal Processing Toolbox for implementing your filter:

- `dfilt`—lets you specify the filter structure and creates a digital filter object.
- `filter`—for  $b$  and  $a$  coefficient input, implements a direct-form II transposed structure and filters the data. For `dfilt` input, `filter` uses the structure specified with `dfilt` and filters the data.

---

**Note** Using `filter` on  $b$  and  $a$  coefficients normalizes the filter by forcing the  $a_0$  coefficient to be equal to 1. Using `filter` on a `dfilt` object does not normalize the filter.

---

Choosing the best filter structure depends on the task the filter will perform. Some structures are more suited to or may be more computationally efficient for particular tasks. For example, often it is not possible to build recursive (IIR) filters to run at very high speeds and instead, you would use a nonrecursive (FIR) filter. FIR filters are always stable and have well-behaved roundoff noise characteristics. Direct-form IIR filters are usually realized in second-order-sections because they are sensitive to roundoff noise.

## Using `dfilt`

Implementing your digital filter using `dfilt` lets you specify the filter structure and creates a single filter object from the filter coefficient vectors. `dfilt` objects have many predefined methods which can provide information about the filter that is not easily obtained directly from the filter coefficients alone. For a complete list of these methods and for more information, see `dfilt`.

After you have created a `dfilt` object, you can use `filter` to apply your implemented filter to data. The complete process of designing, implementing, and applying a filter using a `dfilt` object is described below:

- 1 Generate the filter coefficients using any IIR or FIR filter design function.
- 2 Create the filter object from the filter coefficients and the specified filter structure using `dfilt`.
- 3 Apply the `dfilt` filter object to the data, `x` using `filter`.

For example, to design, implement as a direct-form II transposed structure, and apply a Butterworth filter to the data in `x`:

```
[b,a] = butter(5,0.4);  
Hd = dfilt.df2t(b,a);      %Implement direct-form II transposed  
filter(Hd,x)
```

Another way to implement a direct-form II structure is with `filter`:

```
[b,a] = butter(5,0.4);  
filter(b,a,x)
```

---

**Note** `filter` implements only a direct-form II structure and does not create a filter object.

---

## **Selected Bibliography**

[1] Karam, L.J., and J.H. McClellan. "Complex Chebyshev Approximation for FIR Filter Design." *IEEE Trans. on Circuits and Systems II*. March 1995.

[2] Selesnick, I.W., and C.S. Burrus. "Generalized Digital Butterworth Filter Design." *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing*. Vol. 3 (May 1996).

[3] Selesnick, I.W., M. Lang, and C.S. Burrus. "Constrained Least Square Design of FIR Filters without Specified Transition Bands." *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing*. Vol. 2 (May 1995). Pgs. 1260-1263.

# Statistical Signal Processing

---

The following chapter discusses statistical signal processing tools and applications, including correlations, covariance, and spectral estimation.

Correlation and Covariance (p. 3-2)

Correlation and covariance background information and toolbox functions

Spectral Analysis (p. 3-6)

Spectral estimation techniques and toolbox functions

Selected Bibliography (p. 3-49)

Sources of additional information

## Correlation and Covariance

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 true cross-correlation sequence is a statistical quantity defined as

$$R_{xy}(m) = E\{x_{n+m}y_n^*\} = E\{x_n y_{n-m}^*\}$$

where  $x_n$  and  $y_n$  are stationary random processes,  $-\infty < n < \infty$ , and  $E\{\cdot\}$  is the expected value operator. The covariance sequence is the mean-removed cross-correlation sequence

$$C_{xy}(m) = E\{(x_{n+m} - \mu_x)(y_n - \mu_y)^*\}$$

or, in terms of the cross-correlation,

$$C_{xy}(m) = R_{xy}(m) - \mu_x \mu_y^*$$

In practice, you must estimate these sequences, because it is possible to access only a finite segment of the infinite-length random process. 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)

$$\hat{R}_{xy}(m) = \begin{cases} \sum_{n=0}^{N-m-1} x_{n+m} y_n^* & m \geq 0 \\ \hat{R}_{yx}^*(-m) & m < 0 \end{cases}$$

where we assume for this discussion that  $x_n$  and  $y_n$  are indexed from 0 to  $N-1$ , and  $\hat{R}_{xy}(m)$  from  $-(N-1)$  to  $N-1$ . The `xcorr` function evaluates this sum 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)  
  
xyc =  
  
    1.0000  
    2.0000  
    3.0000  
    4.0000  
    5.0000  
    4.0000  
    3.0000  
    2.0000  
    1.0000
```

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

$$\hat{E}\{R_{xy}(m)\} = \sum_{n=0}^{N-|m|-1} E\{x_{n+m}y_n^*\} = (N-|m|)R_{xy}(m)$$

`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  $\text{norm}(x) \cdot \text{norm}(y)$  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  $(2M-1)$ -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

$$S = [s_1 \ s_2 \ s_3]$$

then the result of `xcorr(S)` is organized as

$$R = [R_{s_1s_1} \ R_{s_1s_2} \ R_{s_1s_3} \ R_{s_2s_1} \ R_{s_2s_2} \ R_{s_2s_3} \ R_{s_3s_1} \ R_{s_3s_2} \ R_{s_3s_3}]$$

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

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 wide-band noise.

The *power spectral density* (PSD) of a stationary random process  $x_n$  is mathematically related to the correlation 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  $f$  (e.g., in hertz) by using the relation  $\omega = 2\pi f/f_s$ , where  $f_s$  is the sampling frequency.

$$P_{xx}(f) = \frac{1}{f_s} \sum_{m=-\infty}^{\infty} R_{xx}(m)e^{-2\pi jfm/f_s}$$

The correlation sequence can be derived from the PSD by use of the inverse discrete-time Fourier transform:

$$R_{xx}(m) = \int_{-\pi}^{\pi} (P_{xx}(\omega)e^{j\omega m})d\omega = \int_{-f_s/2}^{f_s/2} (P_{xx}(f)e^{2\pi jfm/f_s})df$$

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  $[\omega_1, \omega_2]$ ,  $0 \leq \omega_1 < \omega_2 \leq \pi$ , can be found by integrating the PSD over that band:

$$\bar{P}_{[\omega_1, \omega_2]} = \int_{\omega_1}^{\omega_2} P_{xx}(\omega) d\omega + \int_{-\omega_1}^{-\omega_2} P_{xx}(\omega) d\omega$$

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  $\bar{P}_{[\omega_1, \omega_2]}$ .

For real signals, the PSD is symmetric about DC, and thus  $P_{xx}(\omega)$  for  $0 \leq \omega < \pi$  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

$$P_{onesided}(\omega) = \begin{cases} 0, & -\pi \leq \omega < 0 \\ 2P_{xx}(\omega), & 0 \leq \omega < \pi \end{cases}$$

The average power of a signal over the frequency band  $[\omega_1, \omega_2]$ ,  $0 \leq \omega_1 < \omega_2 \leq \pi$ , can be computed using the one-sided PSD as

$$\bar{P}_{[\omega_1, \omega_2]} = \int_{\omega_1}^{\omega_2} P_{onesided}(\omega) d\omega$$

## Spectral Estimation Method

The various methods of spectrum estimation available in the Signal Processing 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*. An improved version of the periodogram is *Welch's method* [8]. A more modern nonparametric technique is the *multitaper method (MTM)*.

*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.

*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 correlation 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.

All three categories of methods are listed in the table below with the corresponding toolbox function and spectrum object names. See “Parametric Modeling” on page 4-15 for details about `lpc` and other parametric estimation functions.

<b>Method</b>	<b>Description</b>	<b>Functions</b>
Periodogram	Power spectral density estimate	<code>spectrum.periodogram</code> , <code>periodogram</code>
Welch	Averaged periodograms of overlapped, windowed signal sections	<code>spectrum.welch</code> , <code>pwelch</code> , <code>cpsd</code> , <code>tfestimate</code> , <code>mscohere</code>
Multitaper)	Spectral estimate from combination of multiple orthogonal windows (or “tapers”)	<code>spectrum.mtm</code> , <code>pmtm</code>
Yule-Walker AR	Autoregressive (AR) spectral estimate of a time-series from its estimated autocorrelation function	<code>spectrum.yulear</code> , <code>pyulear</code>
Burg	Autoregressive (AR) spectral estimation of a time-series by minimization of linear prediction errors	<code>spectrum.burg</code> , <code>pburg</code>
Covariance	Autoregressive (AR) spectral estimation of a time-series by minimization of the forward prediction errors	<code>spectrum.cov</code> , <code>pcov</code>
Modified Covariance	Autoregressive (AR) spectral estimation of a time-series by minimization of the forward and backward prediction errors	<code>spectrum.mcov</code> , <code>pmcov</code>

Method	Description	Functions
MUSIC	Multiple signal classification	spectrum.music, pmusic
Eigenvector	Pseudospectrum estimate	spectrum.eigenvector, peig

## Nonparametric Methods

The following sections discuss the periodogram, modified periodogram, Welch, and multitaper methods of nonparametric estimation, along with the related CPSD function, transfer function estimate, and coherence function.

### 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 take the magnitude squared of the result. This estimate is called the *periodogram*.

The periodogram estimate of the PSD of a length- $L$  signal  $x_L[n]$  is

$$\hat{P}_{xx}(f) = \frac{|X_L(f)|^2}{f_s L}$$

where

$$X_L(f) = \sum_{n=0}^{L-1} x_L[n] e^{-2\pi jfn/f_s}$$

The actual computation of  $X_L(f)$  can be performed only at a finite number of frequency points,  $N$ , and usually employs the FFT. In practice, most implementations of the periodogram method compute the  $N$ -point PSD estimate

$$\hat{P}_{xx}[f_k] = \frac{|X_L[f_k]|^2}{f_s L}, \quad f_k = \frac{k f_s}{N}, \quad k = 0, 1, \dots, N-1$$

where

$$X_L[f_k] = \sum_{n=0}^{N-1} x_L[n] e^{-2\pi j k n / N}$$

It is wise to choose  $N > L$  so that  $N$  is the next power of two larger than  $L$ . To evaluate  $X_L[f_k]$ , we simply pad  $x_L[n]$  with zeros to length  $N$ . If  $L > N$ , we must wrap  $x_L[n]$  modulo- $N$  prior to computing  $X_L[f_k]$ .

As an example, consider the following 1001-element signal  $x_n$ , which consists of two sinusoids plus noise:

```
randn('state',0);
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));
```

---

**Note** The three last lines illustrate a convenient and general way to express the sum of sinusoids. Together they are equivalent to

```
xn = sin(2*pi*150*t) + 2*sin(2*pi*140*t) + 0.1*randn(size(t));
```

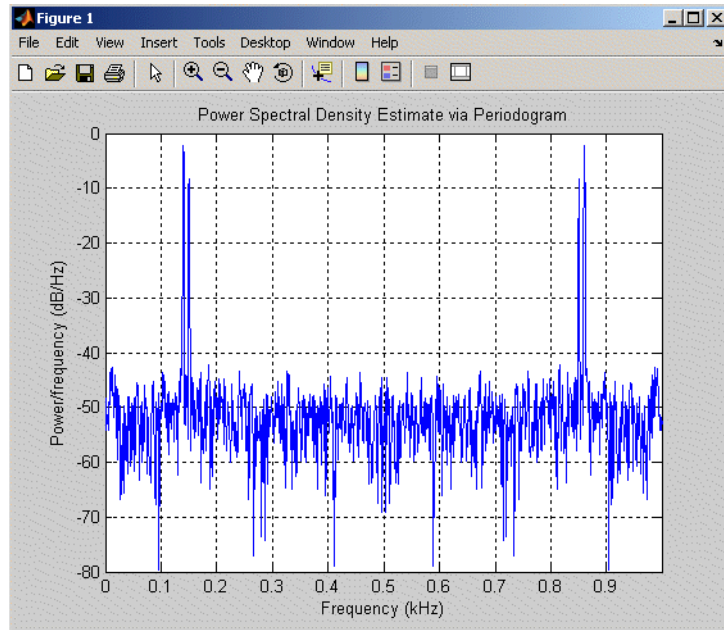
---

The periodogram estimate of the PSD can be computed by creating a periodogram object

```
Hs = spectrum.periodogram('Hamming');
```

and a plot of the estimate can be displayed with the psd method:

```
psd(Hs,xn,'Fs',fs,'NFFT',1024,'SpectrumType','twosided')
```



The average power can be computed by approximating the integral with the following sum:

```
Hdsp2= psd(Hs,xn,'Fs',fs,'NFFT',1024,'SpectrumType','twosided');  
Pow = avgpower(Hdsp2)
```

```
Pow =  
2.5059
```



You can also compute the average power from the one-sided PSD estimate:

```
Hdsp3= psd(Hs,xn,'Fs',fs,'NFFT',1024,'SpectrumType','onesided');
Pow = avgpower(Hdsp3)
```

```
Pow =
    2.5059
```

### 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 signal  $x_L[n]$ , as discussed in the “Periodogram” on page 3-10. 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,  $w_R[n]$ :

$$x_L[n] = x[n] \cdot w_R[n]$$

Because multiplication in the time domain corresponds to convolution in the frequency domain, the Fourier transform of the expression above is

$$X_L(f) = \frac{1}{f_s} \int_{-f_s/2}^{f_s/2} X(\rho) W_R(f - \rho) d\rho$$

The expression developed earlier for the periodogram,

$$\hat{P}_{xx}(f) = \frac{|X_L(f)|^2}{f_s L}$$

illustrates that the periodogram is also influenced by this convolution.

The effect of the convolution is best understood for sinusoidal data. Suppose that  $x[n]$  is composed of a sum of  $M$  complex sinusoids:

$$x[n] = \sum_{k=1}^M A_k e^{j\omega_k n}$$

Its spectrum is

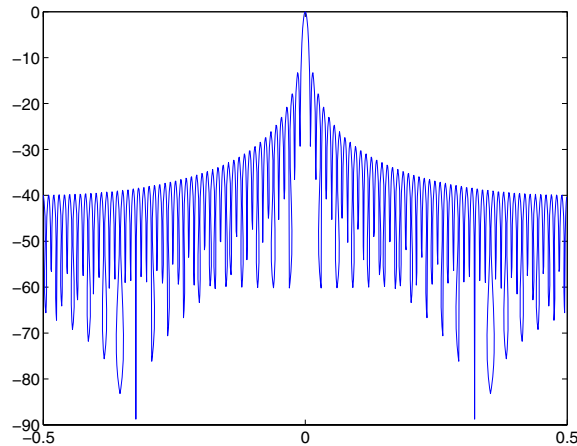
$$X(f) = f_s \sum_{k=1}^M A_k \delta(f - f_k)$$

which for a finite-length sequence becomes

$$X_L(f) = \int_{-f_s/2}^{f_s/2} \left( \sum_{k=1}^M A_k \delta(\rho - f_k) \right) W_R(f - \rho) d\rho = \sum_{k=1}^M A_k W_R(f - f_k)$$

So in the spectrum of the finite-length signal, the Dirac deltas have been replaced by terms of the form  $W_R(f - f_k)$ , which corresponds to the frequency response of a rectangular window centered on the frequency  $f_k$ .

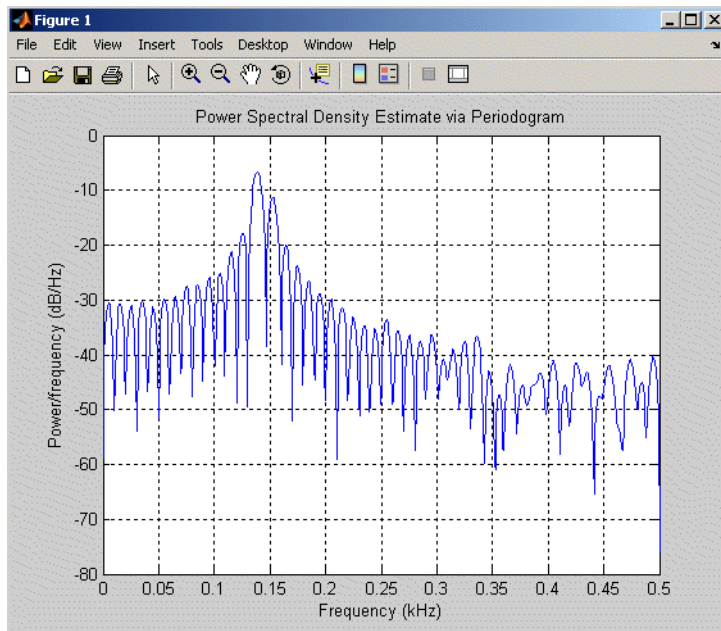
The frequency response of a rectangular window has the shape of a sinc signal, as shown below.



The plot displays a main lobe and several side lobes, 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:

```
randn('state',0)
fs = 1000;           % Sampling frequency
t = (0:fs/10)/fs;   % One-tenth of a 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));
Hs = spectrum.periodogram;
psd(Hs,xn,'Fs',fs,'NFFT',1024)
```



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  $f_s / L$ .

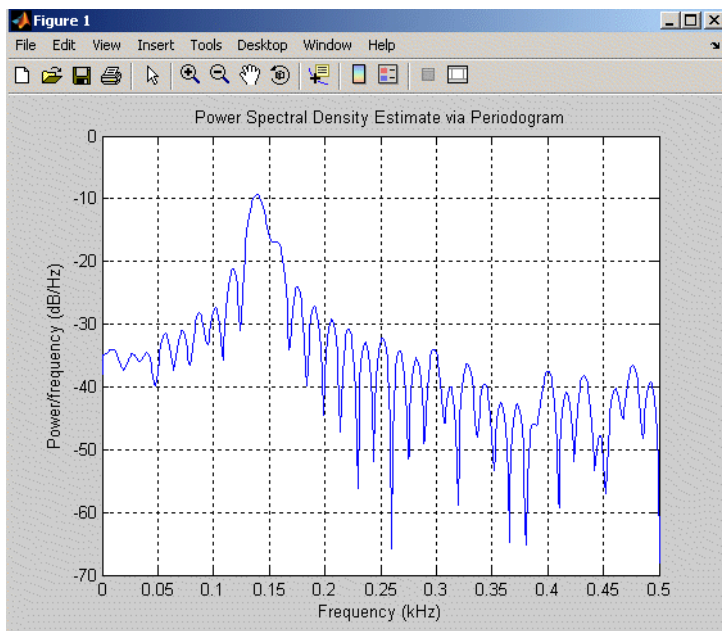
In other words, for two sinusoids of frequencies  $f_1$  and  $f_2$ , the resolvability condition requires that

$$\Delta f = (f_1 - f_2) > \frac{f_s}{L}$$

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:

```
randn('state',0)
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));
Hs=spectrum.periodogram;
psd(Hs,xn,'Fs',fs,'NFFT',1024)
```

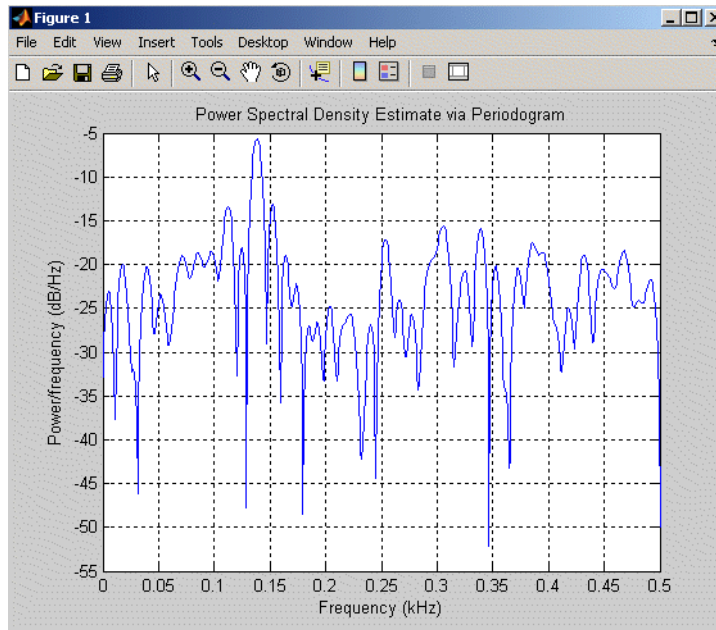


The above discussion about resolution did not consider the effects of noise since the signal-to-noise ratio (SNR) has been relatively high thus far. When the SNR is low, true spectral features are much harder to distinguish, and noise artifacts appear in spectral estimates based on the periodogram. The example below illustrates this:

```

randn('state',0)
fs = 1000;           % Sampling frequency
t = (0:fs/10)./fs; % One-tenth of a 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));
Hs=spectrum.periodogram;
psd(Hs,xn,'Fs',fs,'NFFT',1024)

```



**Bias of the Periodogram.** The periodogram is a biased estimator of the PSD. Its expected value can be shown to be

$$E\left\{\frac{|X_L(f)|^2}{f_s L}\right\} = \frac{1}{f_s L} \int_{-f_s/2}^{f_s/2} P_{xx}(\rho) |W_R(f - \rho)|^2 d\rho$$

which is similar to the first expression for  $X_L(f)$  in “Spectral Leakage” on page 3-13, except that the expression here is in terms of average power rather than magnitude. This suggests that the estimates produced by the periodogram correspond to a *leaky* PSD rather than the true PSD.

Note that  $|W_R(f - \rho)|^2$  essentially yields a triangular Bartlett window (which is apparent from the fact that the convolution of two rectangular pulses is a triangular pulse). This results in a height for the largest sidelobes of the leaky power spectra that is about 27 dB below the mainlobe peak; i.e., about twice the frequency separation relative to the non-squared rectangular window.

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 (also true for a Bartlett window). 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 approximately

$$\text{var} \left\{ \frac{|X_L(f)|^2}{f_s L} \right\} \approx P_{xx}^2(f) \left[ 1 + \left( \frac{\sin(2\pi Lf/f_s)}{L \sin(2\pi f/f_s)} \right)^2 \right]$$

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

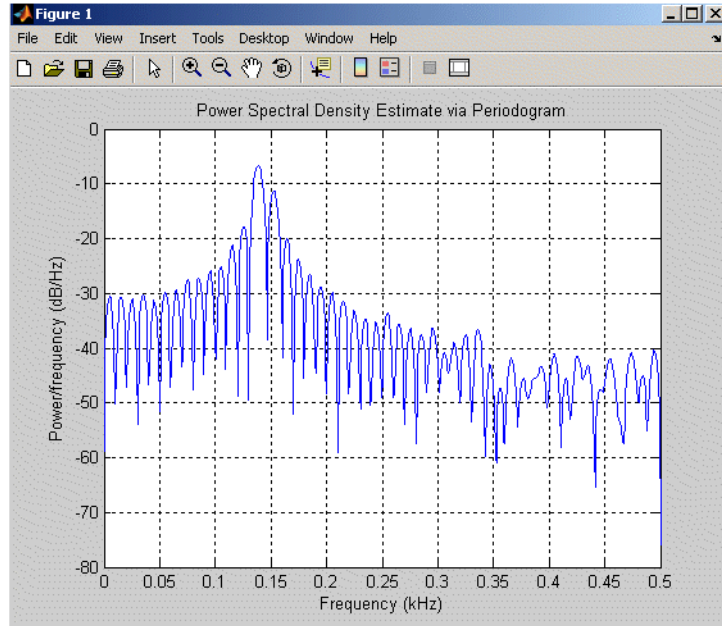
The *modified periodogram* windows the time-domain signal prior to computing the FFT in order to smooth the edges of the signal. This has the effect of reducing the height of the sidelobes or spectral leakage. This phenomenon gives rise to the interpretation of sidelobes as spurious frequencies introduced 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 net reduction of resolution.

The periodogram function 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:

```

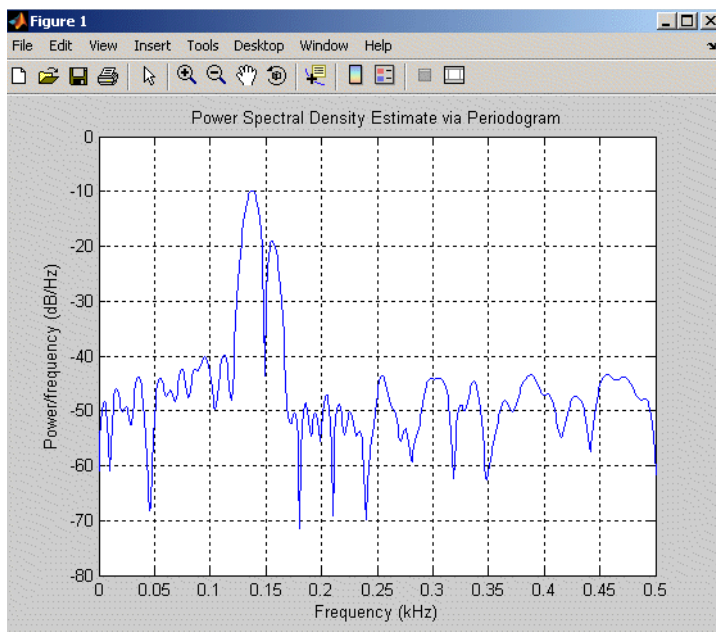
randn('state',0)
fs = 1000;           % Sampling frequency
t = (0:fs/10)./fs; % One-tenth of a second worth of samples
A = [1 2];          % Sinusoid amplitudes
f = [150;140];      % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
Hrect = spectrum.periodogram;
psd(Hrect,xn,'Fs',fs,'NFFT',1024);

```





```
Hamm = spectrum.periodogram('Hamming');  
psd(Hamm,xn,'Fs',fs,'NFFT',1024);
```



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, the periodogram function normalizes the window to have an average power of unity. This way the choice of window does not affect the average power of the signal.

The modified periodogram estimate of the PSD is

$$\hat{P}_{xx}(f) = \frac{|X_L(f)|^2}{f_s L U}$$

where  $U$  is the window normalization constant

$$U = \frac{1}{L} \sum_{n=0}^{L-1} |w(n)|^2$$

which is independent of the choice of window. 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 [8]. 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.

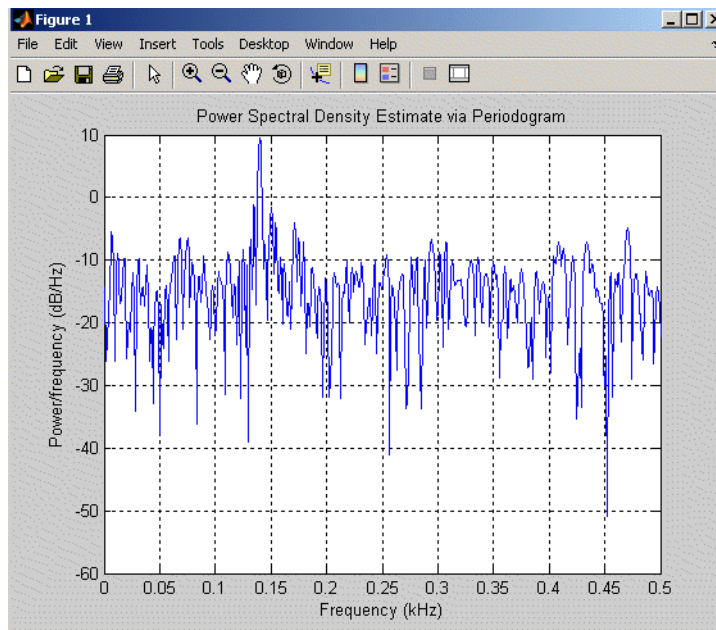
Welch's method is implemented in the Signal Processing Toolbox by the `spectrum.welch` object or `pwelch` function. By default, the data is divided into eight segments with 50% overlap between them. A Hamming window is used to compute the modified periodogram of each segment.

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 tends to introduce 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.

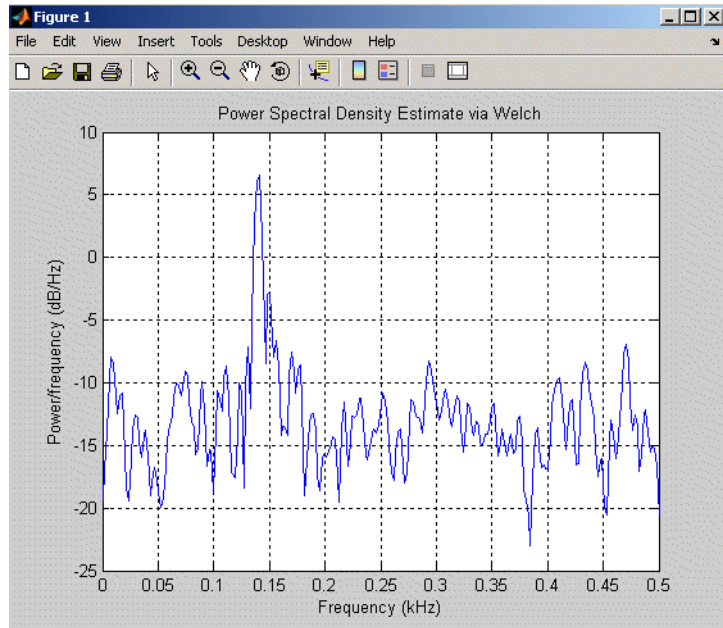
Consider an original signal consisting of 301 samples:

```
randn('state',1)
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));
Hs = spectrum.periodogram('rectangular')
psd(Hs,xn,'Fs',fs,'NFFT',1024);
```



We can obtain Welch's spectral estimate for 3 segments with 50% overlap with

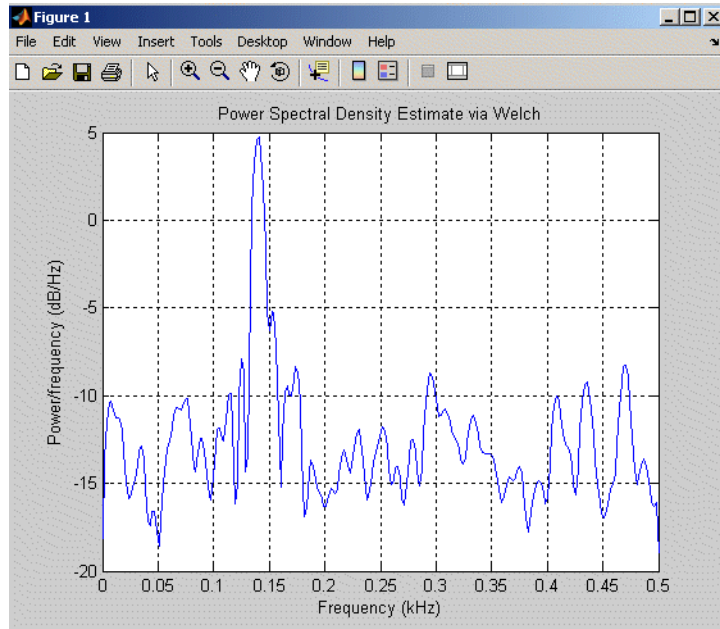
```
Hs = spectrum.welch('rectangular',150,50);  
psd(Hs,xn,'Fs',fs,'NFFT',512);
```



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:

```
Hs = spectrum.welch('rectangular',100,75);  
psd(Hs,xn,'Fs',fs,'NFFT',512);
```



For a more detailed discussion of Welch's method of PSD estimation, see Kay [2] and Welch [8].

### Bias and Normalization in Welch's Method

Welch's method yields a biased estimator of the PSD. The expected value can be found to be

$$E\{\hat{P}_{welch}\} = \frac{1}{f_s L_s U} \int_{-f_s/2}^{f_s/2} P_{xx}(\rho) |W(f-\rho)|^2 d\rho$$

where  $L_s$  is the length of the data segments and  $U$  is the same normalization constant present in the definition of the modified periodogram. 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  $L_s < L$ .

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 the rectangular window discussed in "Spectral Leakage" on page 3-13. 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.

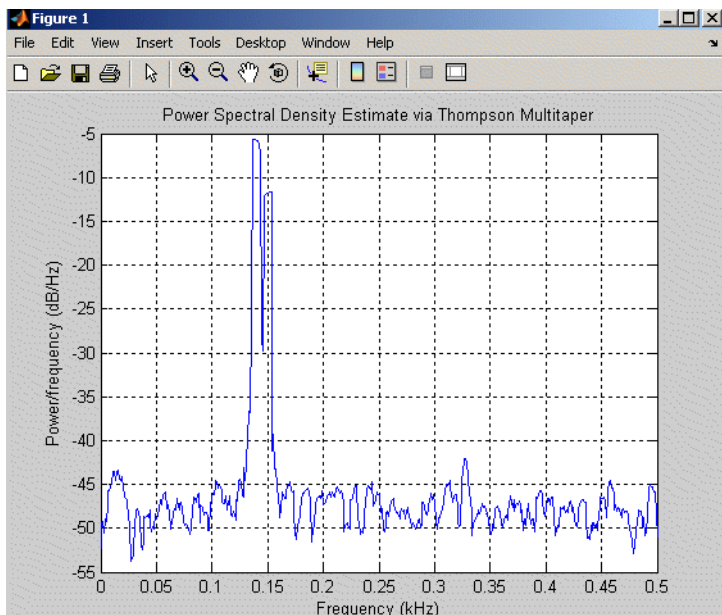
Welch's method can be given a similar interpretation in terms of a filter bank. In Welch's implementation, several samples are used to compute the output power, resulting in reduced variance of the estimate. On the other hand, the bandwidth of each bandpass filter is larger than that corresponding to 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  $2*NW-1$  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` and the object that implements it is `spectrum.mtm`. Use `spectrum.mtm` to compute the PSD of  $x_n$  from the previous examples:

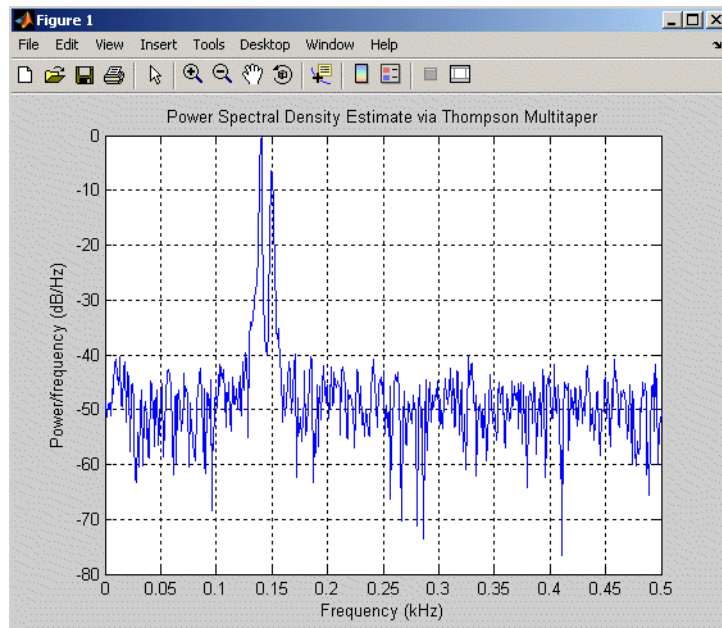
```
randn('state',0)
fs = 1000;           % Sampling frequency
t = (0:fs)/fs;     % One second worth of samples
A = [1 2];         % Sinusoid amplitudes
f = [150;140];     % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
Hs1 = spectrum.mtm(4,'adapt');
psd(Hs1,xn,'Fs',fs,'NFFT',1024)
```





By lowering the time-bandwidth product, you can increase the resolution at the expense of larger variance:

```
Hs2 = spectrum.mtm(3/2,'adapt');
psd(Hs2,xn,'Fs',fs,'NFFT',1024)
```



Note that the average power is conserved in both cases:

```
Hs1p = psd(Hs1,xn,'Fs',fs,'NFFT',1024);
Pow1 = avgpower(Hs1p)
```

```
Pow1 =
    2.4926
```

```
Hs2p = psd(Hs2,xn,'Fs',fs,'NFFT',1024);
Pow2 = avgpower(Hs2p)
```

```
Pow2 =
    2.4927
```

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. The M-files `dpsssav`, `dpssload`, `dpssdir`, and `dpssc` 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:

$$S_{xy} = \text{cpsd}(x,y,nwin,noverlap,nfft,fs)$$

### 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

$$P_{xy}(\omega) = H(\omega)P_{xx}(\omega)$$

An estimate of the transfer function between  $x(n)$  and  $y(n)$  is

$$\hat{H}(\omega) = \frac{\hat{P}_{xy}(\omega)}{\hat{P}_{xx}(\omega)}$$

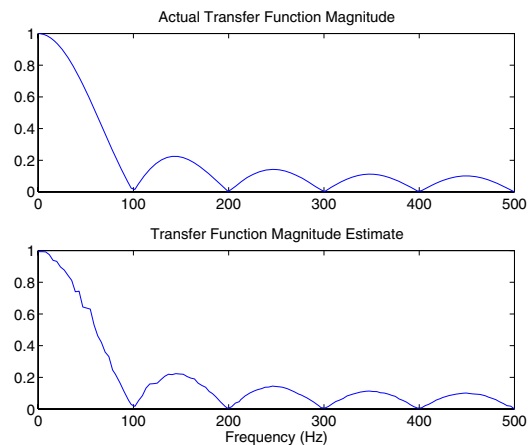
This method estimates both magnitude and phase information. The `tfestimate` function uses Welch's method to compute the CPSD and power spectrum, and then forms their quotient for the transfer function estimate. Use `tfestimate` the same way that you use the `cpsd` function.

Filter the signal `xn` with an FIR filter, then plot 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);

subplot(2,1,1); plot(f,abs(H));
title('Actual Transfer Function Magnitude');

subplot(2,1,2); plot(f,abs(HEST));
title('Transfer Function Magnitude Estimate');
xlabel('Frequency (Hz)');
```



**Coherence Function**

The magnitude-squared coherence between two signals  $x(n)$  and  $y(n)$  is

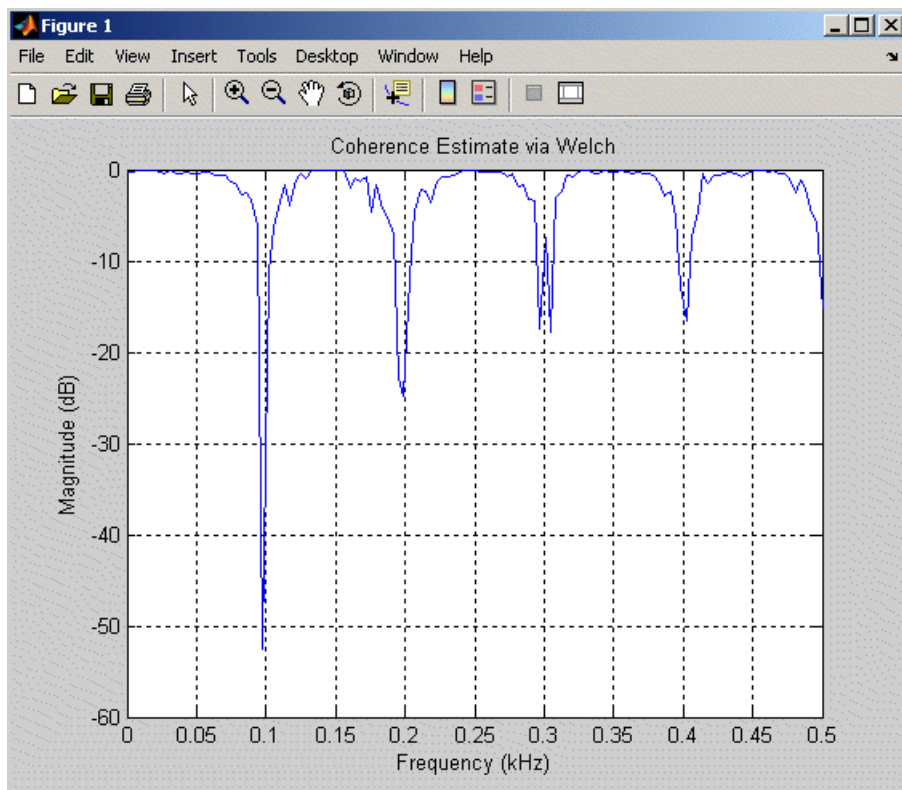
$$C_{xy}(\omega) = \frac{|P_{xy}(\omega)|^2}{P_{xx}(\omega)P_{yy}(\omega)}$$

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  $x$  and  $y$ , 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 `tfeestimate` functions.

The coherence function of  $x_n$  and the filter output  $y_n$  versus frequency is

```
mscohere(xn,yn,256,128,256,fs)
```



If the input sequence length  $nfft$ , window length  $window$ , and the number of overlapping data points in a window  $numoverlap$ , 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.

The following example shows how to use spectral analysis functions to compute the coherence:

```

randn('state',0)
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));
h = ones(1,10)/10; % Moving average filter
yn = filter(h,1,xn);

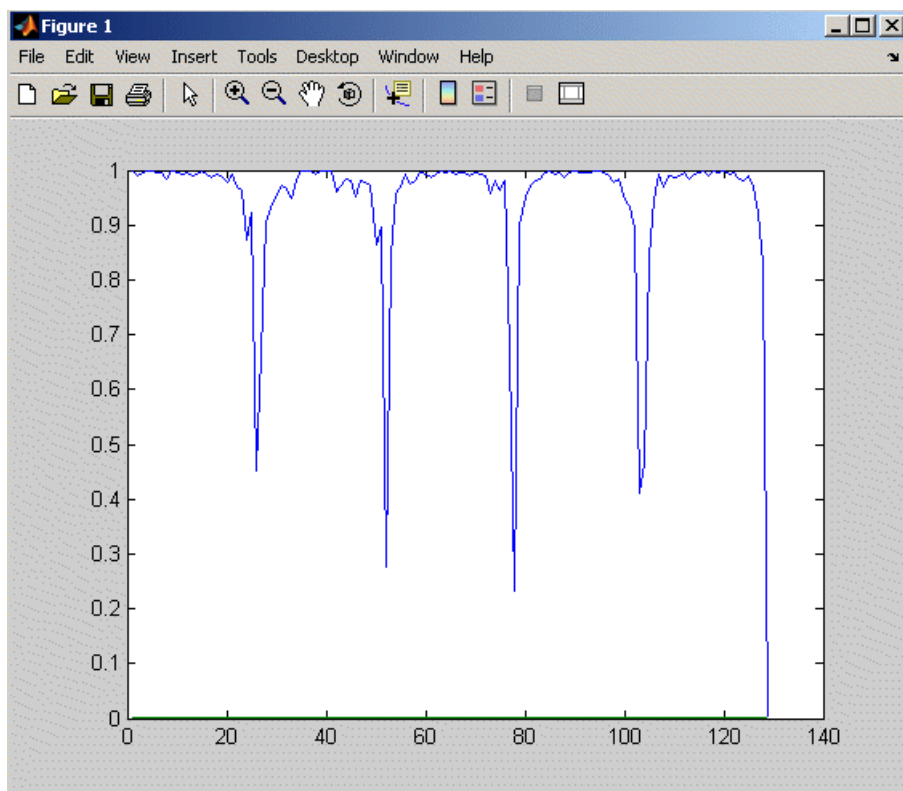
% Parameters common to both CPSD and MSCOHERE:
nfft = 256;
noverlap = 128;
nwin = hanning(256);
CxyCohere = mscohere(xn,yn,nwin,noverlap,nfft,fs);

% Compute the PSDs using Welch's method, which is similar to the
% MSCOHERE function
[Pxx,F] = pwelch(xn,nwin,noverlap,nfft,fs);
[Pyx,F] = pwelch(yn,nwin,noverlap,nfft,fs);

% Convert one-sided PSDs to Power Spectrums
Pxx = [Pxx(1); (Pxx(2:end-1)/2); Pxx(end)].*fs;
Pyx = [Pyx(1); (Pyx(2:end-1)/2); Pyx(end)].*fs;

% Call CPSD function with similar arguments as the MSCOHERE
function
% from above.
[Pxy,F] = cpsd(xn,yn,nwin,noverlap,nfft,fs);
Cxy = (abs(Pxy).^2)./(Pxx.*Pyx);
plot(1:length(Cxy), [CxyCohere Cxy]); % Compare the estimates

```



## 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.

The Signal Processing Toolbox offers the following AR methods for spectral estimation:

- Yule-Walker AR method (autocorrelation method)
- Burg method
- Covariance method
- Modified covariance method

All AR methods yield a PSD estimate given by

$$\hat{P}_{AR}(f) = \frac{1}{f_s} \left| \frac{\epsilon_p}{1 + \sum_{k=1}^p \hat{a}_p(k) e^{-2\pi j k f / f_s}} \right|^2$$

The different AR methods estimate the AR parameters  $a_p(k)$  slightly differently, yielding different PSD estimates. The following table provides a summary of the different AR methods.



	<b>Burg</b>	<b>Covariance</b>	<b>Modified Covariance</b>	<b>Yule-Walker</b>
<b>Characteristics</b>	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 (also called “Autocorrelation method”)
<b>Advantages</b>	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	
<b>Disadvantages</b>	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	
<b>Conditions for Nonsingularity</b>		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 forming a biased estimate of the signal's autocorrelation function, and solving the least squares minimization of the forward prediction error. This results in the Yule-Walker equations.

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \cdots & r(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(p+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(p+1) \end{bmatrix}$$

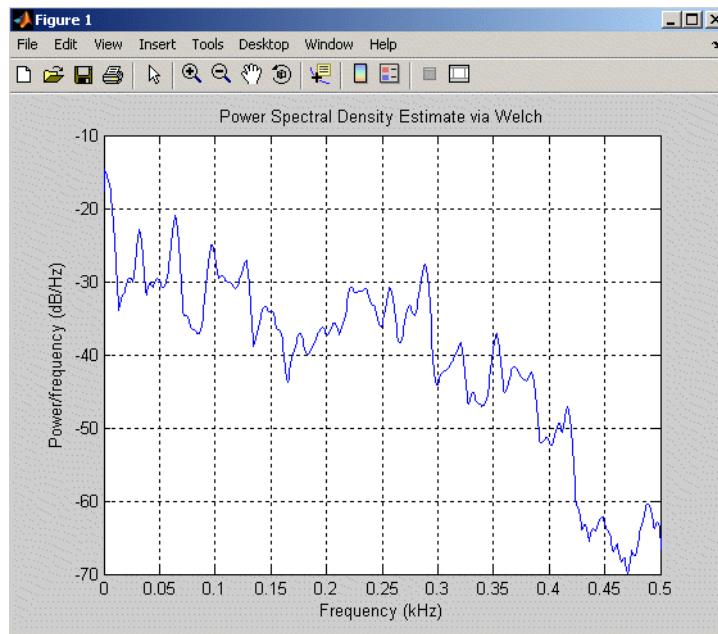
The Yule-Walker AR method produces the same results as a maximum entropy estimator. For more information, see page 155 of item [2] in the “Selected Bibliography” on page 3-49.

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 via Levinson's algorithm, which takes advantage of the Toeplitz structure of the autocorrelation matrix.

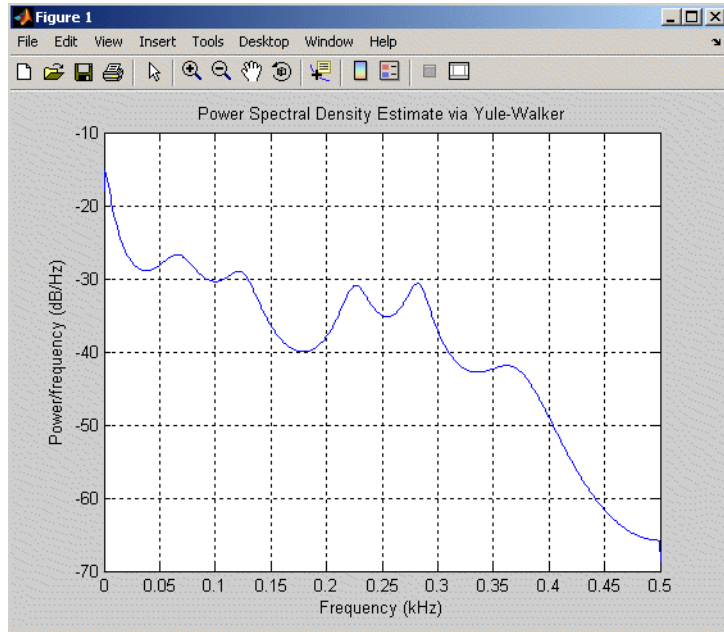
The toolbox object `spectrum.yulear` and function `pyulear` implement the Yule-Walker AR method.

For example, compare the spectrum of a speech signal using Welch's method and the Yule-Walker AR method:

```
load mtlb
Hwelch = spectrum.welch('hamming',256,50);
psd(Hwelch,mtlb,'Fs',Fs,'NFFT',1024)
```



```
Hyulear = spectrum.yulear(14);  
psd(Hyulear,mtlb,'Fs',Fs,'NFFT',1024)
```



The Yule-Walker AR spectrum is smoother than the periodogram because of the simple underlying all-pole model.

### Burg Method

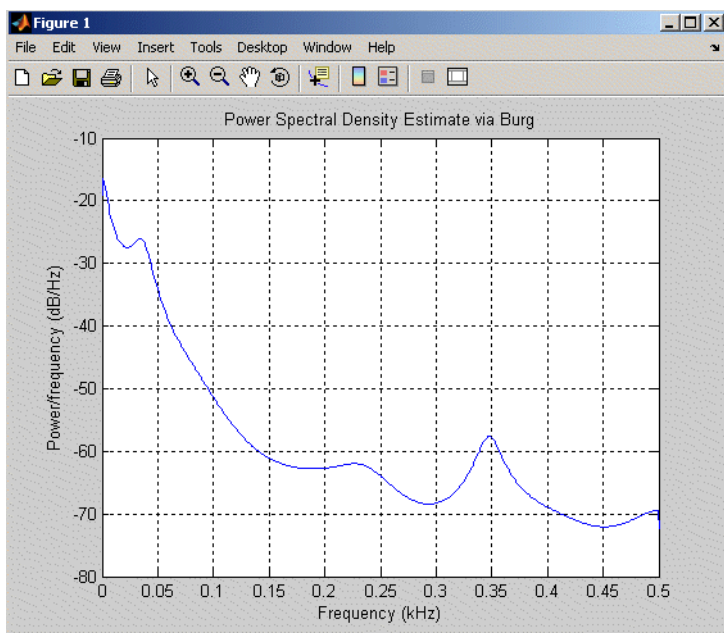
The Burg method for AR spectral estimation is based on minimizing the forward and backward prediction errors while satisfying the Levinson-Durbin recursion (see Marple [3], Chapter 7, and Proakis [6], Section 12.3.3). In contrast to other AR estimation methods, the Burg method avoids calculating the 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.

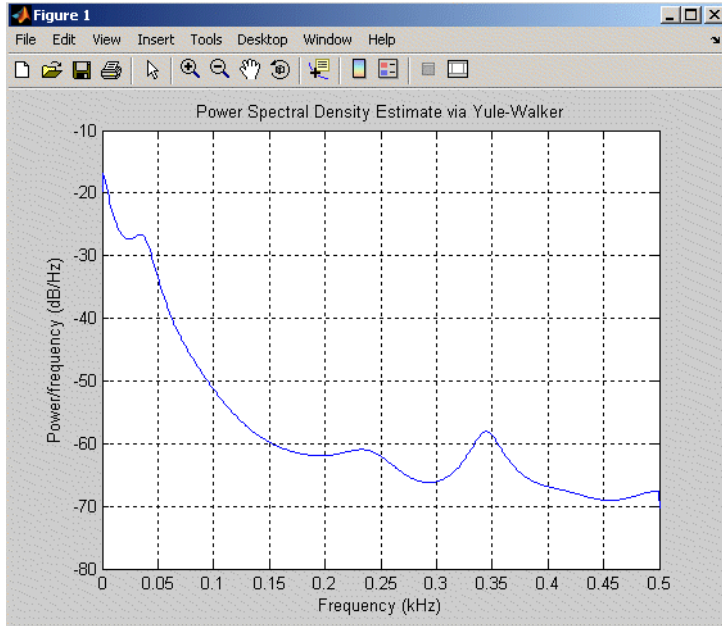
The accuracy of the Burg method is lower for high-order models, long data records, and high signal-to-noise ratios (which can cause *line splitting*, or the generation of extraneous peaks in the spectrum estimate). The spectral density estimate computed by the Burg method is also susceptible to frequency 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 object `spectrum.burg` and function `pburg` implement the Burg method. Compare the spectrum of the speech signal generated by both the Burg method and the Yule-Walker AR method. They are very similar for large signal lengths:

```
load mtlb
Hburg = spectrum.burg(14);           % 14th order model
psd(Hburg,mtlb(1:512),'Fs',Fs,'NFFT',1024)
```



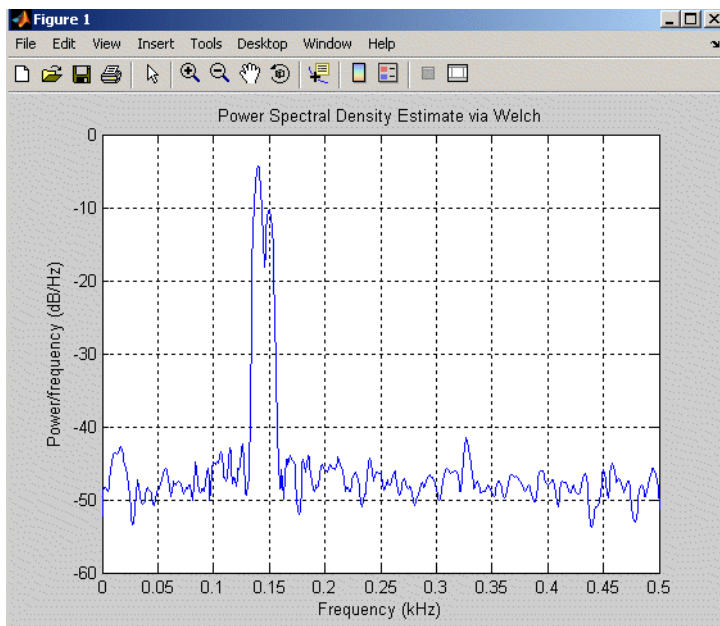
```
Hyulear = spectrum.yulear(14); % 14th order model  
psd(Hyulear,mtlb(1:512),'Fs',Fs,'NFFT',1024)
```



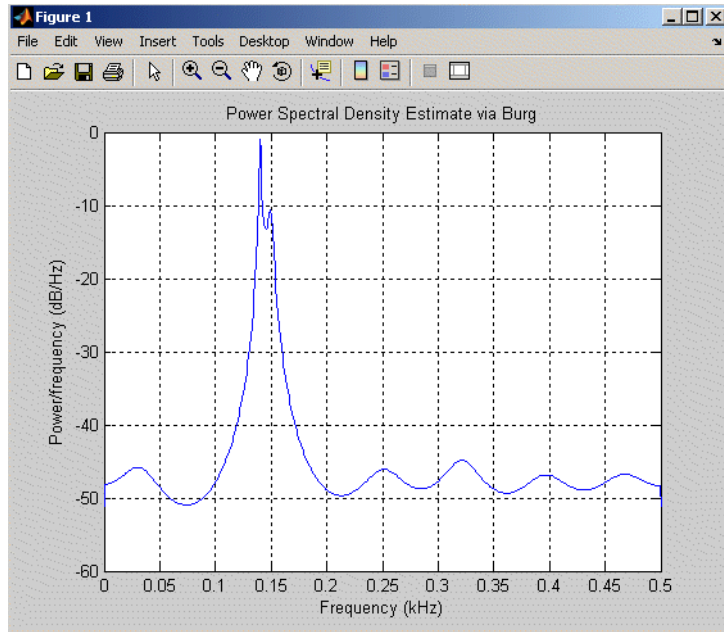
Compare the spectrum of a noisy signal computed using the Burg method and the Welch method:

```
randn('state',0)
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));

Hwelch = spectrum.welch('hamming',256,50);
psd(Hwelch,xn,'Fs',fs,'NFFT',1024)
```



```
Hburg = spectrum.burg(14);  
psd(Hburg,xn,'Fs',fs,'NFFT',1024)
```



Note that, as the model order for the Burg method is reduced, a frequency shift due to the initial phase of the sinusoids will become apparent.

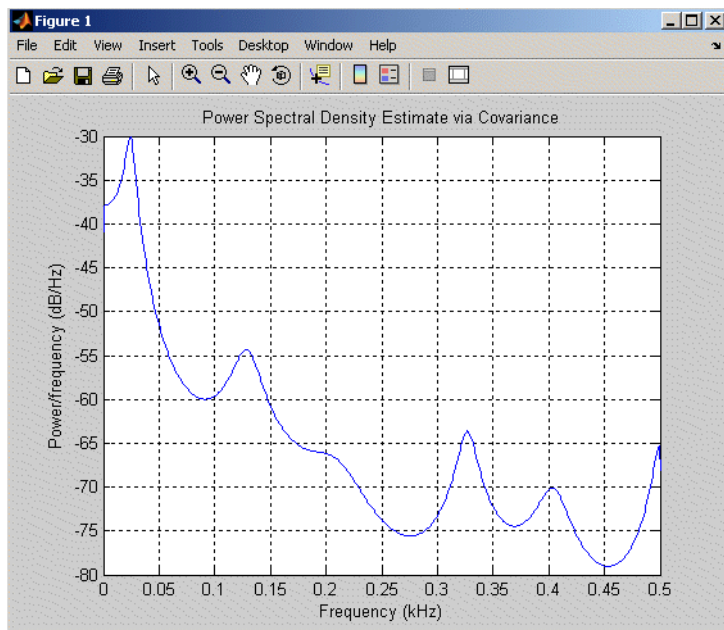


## Covariance and Modified Covariance Methods

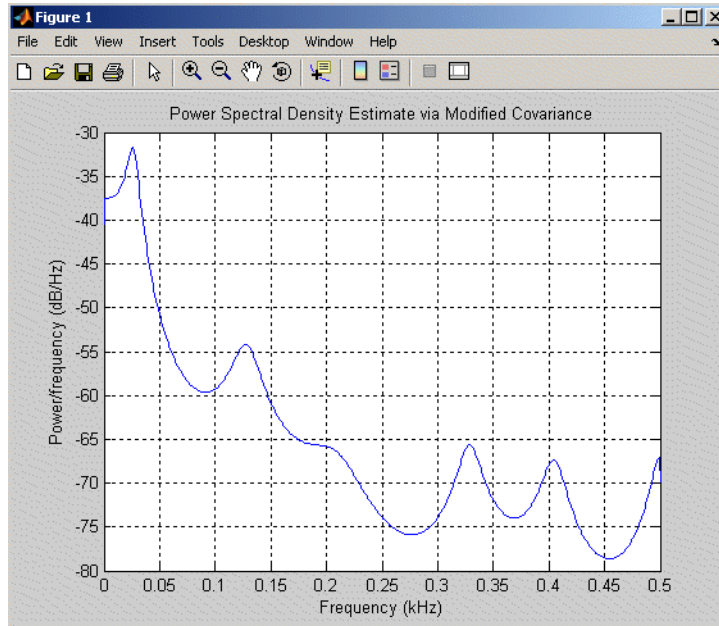
The covariance method for AR spectral estimation is based on minimizing the forward prediction error. The modified covariance method is based on minimizing the forward and backward prediction errors. The toolbox object `spectrum.cov` and function `pcov`, and object `spectrum.mcov` and function `pmcov` implement the respective methods.

Compare the spectrum of the speech signal generated by both the covariance method and the modified covariance method. They are nearly identical, even for a short signal length:

```
load mtlb
Hcov = spectrum.cov(14);           % 14th order model
psd(Hcov,mtlb(1:64),'Fs',Fs,'NFFT',1024)
```



```
Hmcov = spectrum.mcov(14); % 14th order model
psd(Hmcov,mtlb(1:64), 'Fs',Fs, 'NFFT',1024)
```



### MUSIC and Eigenvector Analysis Methods

The `spectrum.music` object and `pmusic` function, and `spectrum.eigenvector` object and `peig` function provide two related spectral analysis methods:

- `spectrum.music` and `pmusic` provide the multiple signal classification (MUSIC) method developed by Schmidt
- `spectrum.eigenvector` and `peig` provides the eigenvector (EV) method developed by Johnson

See Marple [3] (pgs. 373-378) for a summary of these methods.

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

$$P_{music}(f) = \frac{1}{\mathbf{e}^H(f) \left( \sum_{k=p+1}^N \mathbf{v}_k \mathbf{v}_k^H \right) \mathbf{e}(f)} = \frac{1}{N \sum_{k=p+1} |\mathbf{v}_k^H \mathbf{e}(f)|^2}$$

where  $N$  is the size of the eigenvectors and  $\mathbf{e}(f)$  is a vector of complex sinusoids.

$$\mathbf{e}(f) = [1 \exp(j2\pi f) \exp(j2\pi f \cdot 2) \exp(j2\pi f \cdot 4) \dots \exp(j2\pi f \cdot (n-1))]^H$$

$\mathbf{v}$  represents the eigenvectors of the input signal's correlation matrix;  $\mathbf{v}_k$  is the  $k^{\text{th}}$  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  $\mathbf{v}_k^H \mathbf{e}(f)$  is equivalent to a Fourier transform (the vector  $\mathbf{e}(f)$  consists of complex exponentials). This form is useful for numeric computation because the FFT can be computed for each  $\mathbf{v}_k$  and then the squared magnitudes can be summed.

The EV method weights the summation by the eigenvalues of the correlation matrix:

$$P_{ev}(f) = \frac{1}{\left( \sum_{k=p+1}^N |\mathbf{v}_k^H \mathbf{e}(f)|^2 \right) / \lambda_k}$$

The `pmusic` and `peig` functions in this toolbox use the `svd` (singular value decomposition) function in the signal case and the `eig` function for analyzing the correlation matrix and assigning eigenvectors to the signal or noise subspaces. When `svd` is used, `pmusic` and `peig` never compute the correlation matrix explicitly, but the singular values are the eigenvalues.

## Selected Bibliography

- [1] Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.
- [2] Kay, S.M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice Hall, 1988.
- [3] Marple, S.L. *Digital Spectral Analysis*. Englewood Cliffs, NJ: Prentice Hall, 1987.
- [4] Orfanidis, S.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, J.G., and D.G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1996.
- [7] Stoica, P., and R. Moses. *Introduction to Spectral Analysis*. Upper Saddle River, NJ: Prentice Hall, 1997.
- [8] Welch, P.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 (June 1967). Pgs. 70-73.



# Special Topics

---

The following chapter describes specialized techniques and applications provided in the Signal Processing Toolbox.

Windows (p. 4-2)	Window background information and toolbox functions and GUIs
Parametric Modeling (p. 4-15)	Mathematical techniques for modeling systems
Resampling (p. 4-25)	Functions for resampling a signal at a different sampling rate
Cepstrum Analysis (p. 4-28)	Functions for performing cepstrum analysis
FFT-Based Time-Frequency Analysis (p. 4-32)	Spectrograms
Median Filtering (p. 4-33)	Applying a sliding window to a sequence
Communications Applications (p. 4-34)	Functions for communications simulations
Deconvolution (p. 4-41)	Deconvolution information
Specialized Transforms (p. 4-42)	Chirp Z, discrete-cosine, and Hilbert transforms
Selected Bibliography (p. 4-48)	Sources of additional information

## 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.

The toolbox window functions are shown in the table below.

<b>Window</b>	<b>Function</b>
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 Valle-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:

- Window Design and Analysis Tool (`wintool`)
- Window Visualization Tool (`wvtool`)

## 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);
```

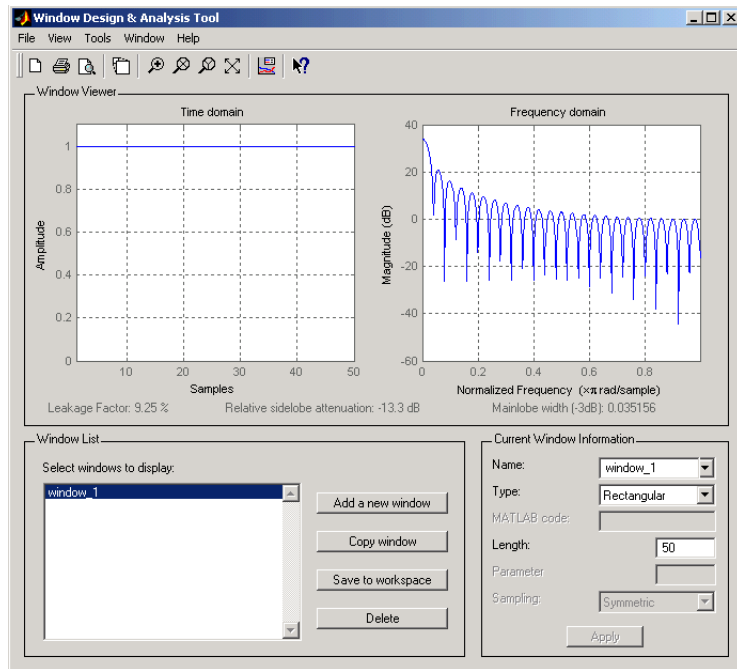
This toolbox stores windows in column vectors by convention, so an equivalent expression is

```
w = ones(50,1);
```

To use the Window Design and Analysis Tool to create this window, type

```
wintool
```

`wintool` opens with a default Hamming window. In the Current Window Information panel, set **Type = Rectangular** and **Length = 50** 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(7)
```

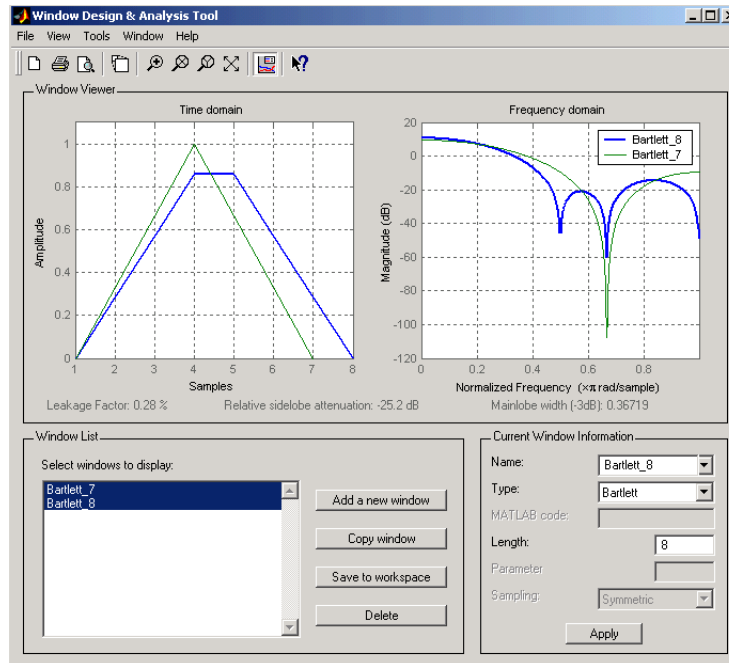
```
ans =
     0
    0.3333
    0.6667
    1.0000
    0.6667
    0.3333
     0
```

```
triang(5)
ans =
    0.3333
    0.6667
    1.0000
    0.6667
    0.3333
```

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:

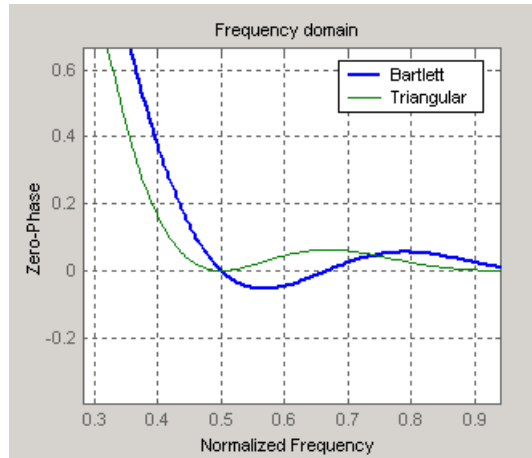
```
w = bartlett(8);
[w(2:7) triang(6)]
ans =
    0.2857    0.1667
    0.5714    0.5000
    0.8571    0.8333
    0.8571    0.8333
    0.5714    0.5000
    0.2857    0.1667
```

You can see the difference between odd and even Bartlett windows in WinTool.



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 is a zoomed version of the Frequency domain plot of 8-point Bartlett and Triangular windows with the y-axis set to **Zero-Phase**, illustrates this difference.



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 Kay [1], pg. 80).

## 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  $0$ ,  $2\pi/(N-1)$ , and  $4\pi/(N-1)$ , where  $N$  is the window length. One way to generate them is

```
ind = (0:n-1)'*2*pi/(n-1);
w = A - B*cos(ind) + C*cos(2*ind);
```

where  $A$ ,  $B$ , and  $C$  are constants you define. 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.

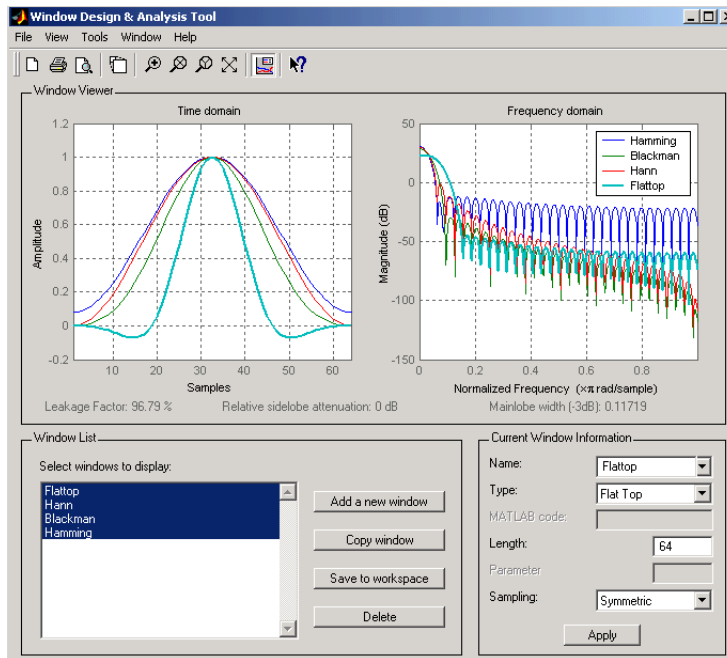
The Hamming and Hann windows are two-term generalized cosine windows, given by  $A = 0.54, B = 0.46$  for Hamming and  $A = 0.5, B = 0.5$  for Hann ( $C = 0$  in both cases). The hamming and hann functions, respectively, compute these windows.

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  $B = 0.5$ .

The Blackman window is a popular three-term window, given by  $A = 0.42, B = 0.5, C = 0.08$ . The blackman function computes this window.

The Flat Top window is a five-term window and is used for calibration. It is given by  $A = 1, B = 1.93, C = 1.29, D = 0.388, \text{ and } E = 0.322$ .

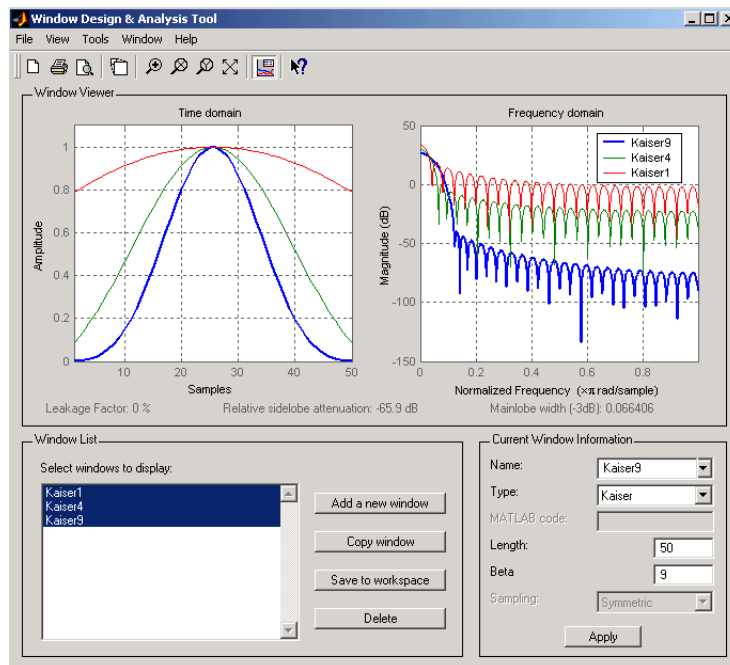
This WinTool compares Blackman, Hamming, Hann, and Flat Top windows.



## 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 sidelobe height. For a given  $\beta$ , the sidelobe height is fixed with respect to window length. The statement `kaiser(n,beta)` computes a length  $n$  Kaiser window with parameter  $\beta$ .

Examples of Kaiser windows with length 50 and beta parameters of 1, 4, and 9 are shown in this wintool example.

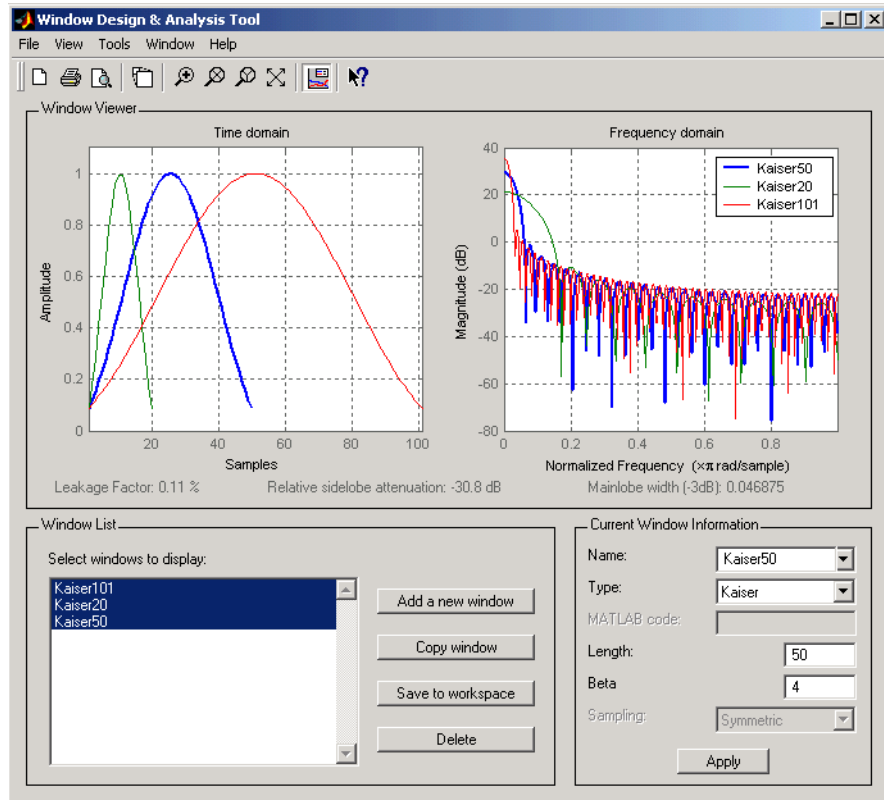


To create these Kaiser windows using the MATLAB command line,

```
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',3)
```

As  $\beta$  increases, the sidelobe height decreases and the mainlobe width increases. This wintool shows how the sidelobe height stays the same for a fixed  $\beta$  parameter as the length is varied.





To create these Kaiser windows using the MATLAB command line:

```
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 sidelobe height of  $-α$  dB, the beta parameter is

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$

For a transition width of  $\Delta\omega$  rad/s, use the length

$$n = \frac{\alpha - 8}{2.285\Delta\omega} + 1$$

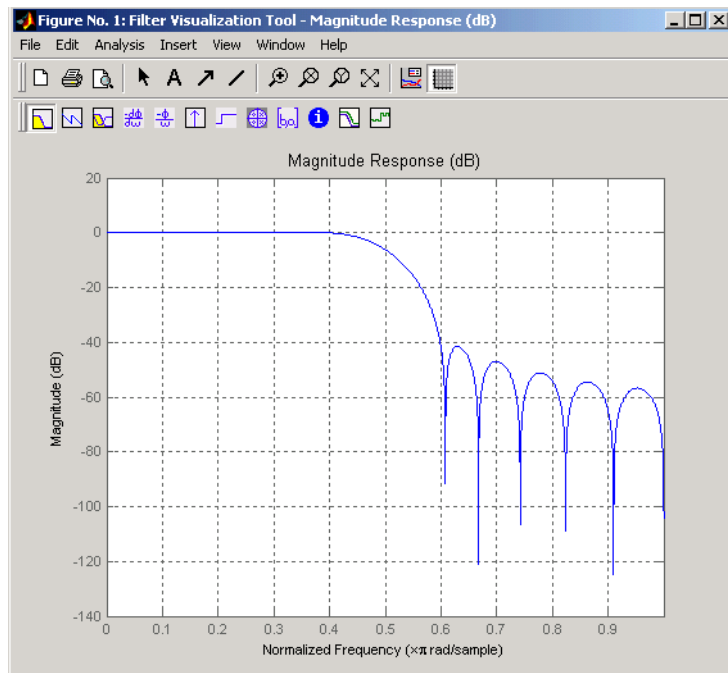
Filters designed using these heuristics will meet the specifications approximately, but you should verify this. To design a lowpass filter with cutoff frequency  $0.5\pi$  rad/s, transition width  $0.2\pi$  rad/s, and 40 dB of attenuation in the stopband, try

```
[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.

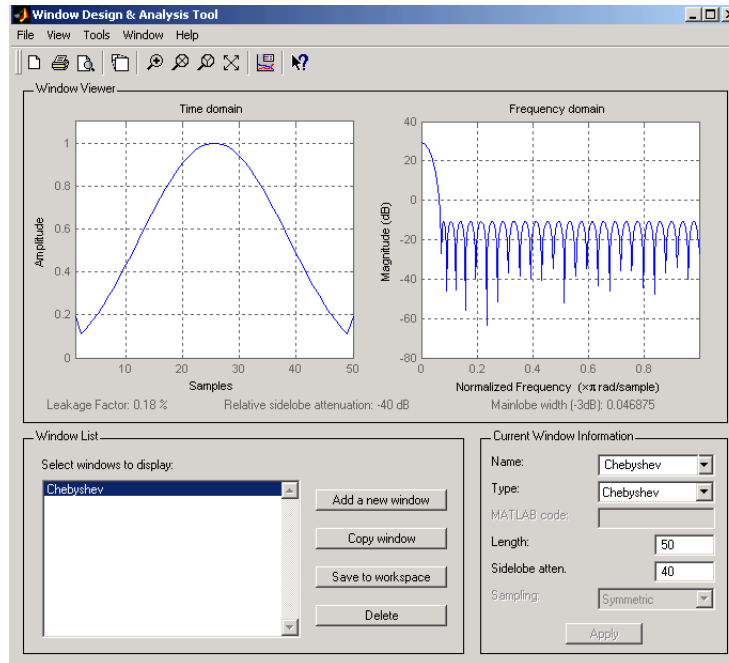
The ripple in the passband is roughly the same as the ripple in the stopband. As you can see from the frequency response, this filter nearly meets the specifications:

```
fvtool(h,1);
```



## Chebyshev Window

The Chebyshev window minimizes the mainlobe width, given a particular sidelobe height. It is characterized by an equiripple behavior, that is, its sidelobes all have the same height.



As shown in the Time Domain plot, the Chebyshev window has large spikes at its outer samples.

For a detailed discussion of the characteristics and applications of the various window types, see Oppenheim and Schaffer [3], pgs. 444-462, and Parks and Burrus [4], pgs. 71-73.

## 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.

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. These functions provide a data-oriented alternative to the IIR and FIR filter design functions discussed in Chapter 2, “Filter Design and Implementation.”

Here is a summary of the parametric modeling functions in this toolbox. Note that the System Identification Toolbox provides a more extensive collection of parametric modeling functions.

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.

Because yulewalk is geared explicitly toward ARMA filter design, it is discussed in Chapter 2, “Filter Design and Implementation”.

pburg and pyulear are discussed in Chapter 3, “Statistical Signal Processing” along with the other (nonparametric) spectral estimation methods.

## 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:

$$x(k) = -a(2)x(k-1) - a(3)x(k-2) - \dots - a(n+1)x(k-n)$$

An  $n$ th-order all-pole model of a signal  $x$  is

$$a = \text{lpc}(x, n)$$

To illustrate `lpc`, create a sample signal that is the impulse response of an all-pole filter with additive white noise:

```
randn('state',0);
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)
a =
    1.0000    0.2574    0.1666    0.1203    0.2598
```

`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)
a =
    1.0000    0.2574    0.1666    0.1203    0.2598
```

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)  
  
a =  
    1.0000    0.2574    0.1666    0.1203    0.2598
```

### **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 (pgs. 226-228). 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)

b =
    0.9567   -0.3351    0.1866   -0.3782

a =
    1.0000   -0.0716    0.2560   -0.2752
```

The `impz` command shows how well this filter's impulse response matches the original sequence:

```
format long
[x impz(b,a,10)]

ans =

    0.95674351884718    0.95674351884718
   -0.26655843782381   -0.26655843782381
   -0.07746676935252   -0.07746676935252
   -0.05223235796415   -0.05223235796415
   -0.18754713506815   -0.05726777015121
    0.15348154656430   -0.01204969926150
    0.13986742016521   -0.00057632797226
    0.00609257234067   -0.01271681570687
    0.03349954614087   -0.00407967053863
    0.01086719328209    0.00280486049427
```

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)

b =
    0.9567   -0.5181    0.5702   -0.5471

a =
    1.0000   -0.2384    0.5234   -0.3065
```

`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)

b =
    1.0000

a =
    1    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)]
```

```
ans =
```

```
-0.0433      0    -0.0000
-0.0240      0     0.0234
-0.0040      0    -0.0778
-0.0448  -0.0000   0.0498
-0.2130  -0.1303  -0.0742
 0.1545   0.1655   0.1270
 0.1426   0.1404   0.1055
 0.0068   0.0188   0.0465
 0.0329   0.0376   0.0530
 0.0108   0.0081  -0.0162
```

```
sum(ans.^2)
```

```
ans =
```

```
0.0953    0.0659    0.0471
```

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
b =
    0.0466    0.1863    0.2795    0.1863    0.0466
a =
    1.0000   -0.7821    0.6800   -0.1827    0.0301
[h,w] = freqz(b,a,64);    % Compute frequency response
[b4,a4] = invfreqz(h,w,4,4) % Model: n = 4, m = 4
b4 =
    0.0466    0.1863    0.2795    0.1863    0.0466
a4 =
    1.0000   -0.7821    0.6800   -0.1827    0.0301
```

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
b4 =
    0.0464    0.1785    0.2446    0.1276
a4 =
    1.0000   -0.9502    0.7382   -0.2006
```

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

$$\min_{b, a} \sum_{k=1}^n wt(k) |h(k)A(w(k)) - B(w(k))|^2$$

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 unity weighting vector
[b30,a30] = invfreqz(h,w,3,3,wt,30) % 30 iterations

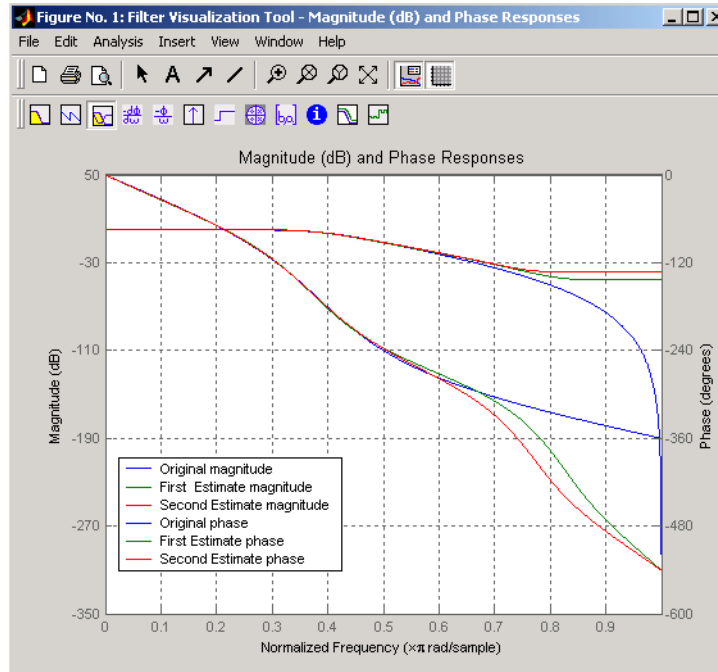
b30 =
    0.0464    0.1829    0.2572    0.1549

a30 =
    1.0000   -0.8664    0.6630   -0.1614
```

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
ans =
    0.0200
sum(abs(h-freqz(b30,a30,w)).^2) % Total error, algorithm 2
ans =
    0.0096
```

## Resampling

The 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

The `resample` function changes the sampling rate for a sequence to any rate that is 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 sampling 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 sampling rate to another, such as from 48 kHz (the digital audio tape standard) to 44.1 kHz (the compact disc standard).

The example file contains a length 4001 vector of speech sampled at 7418 Hz:

```
clear
load mtlb
whos

Name          Size          Bytes          Class
Fs            1x1              8              double array
mtlb          4001x1           32008          double array

Grand total is 4002 elements using 32016 bytes

Fs
Fs =
    7418
```

To play this speech signal on a workstation that can only play sound at 8192 Hz, use the `rat` function to find integers `p` and `q` that yield the correct resampling factor:

```
[p,q] = rat(8192/Fs,0.0001)

p =
    127

q =
    115
```

Since  $p/q \cdot F_s = 8192.05$  Hz, the tolerance of 0.0001 is acceptable; to resample the signal at very close to 8192 Hz:

```
y = resample(mtlb,p,q);
```

`resample` applies a lowpass filter to the input sequence to prevent aliasing during resampling. It designs this filter using the `firls` function with a Kaiser window. The syntax

```
resample(x,p,q,l,beta)
```

controls the filter's length and the beta parameter of the Kaiser window. Alternatively, use the function `intfilt` to design an interpolation filter `b` and use it with

```
resample(x,p,q,b)
```



The `decimate` and `interp` functions do the same thing as `resample` with  $p = 1$  and  $q = 1$ , respectively. These functions provide different anti-alias filtering options, and they incur a slight signal delay due to filtering. The `interp` function is significantly less efficient than the `resample` function with  $q = 1$ .

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 1-20.

The standard MATLAB environment contains a function, `spline`, that works with irregularly spaced data. The MATLAB function `interp1` performs interpolation, or table lookup, using various methods including linear and cubic interpolation.

## Cepstrum Analysis

Cepstrum analysis is a nonlinear signal processing technique with a variety of applications in areas such as speech and image processing. The Signal Processing Toolbox provides three functions for cepstrum analysis.

Operation	Function
Complex cepstrum	cceps
Inverse complex cepstrum	icceps
Real cepstrum	rceps

The complex cepstrum for 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:

```
xhat = cceps(x)
```

For sequences that have roots on the unit circle, cepstrum analysis has numerical problems. See Oppenheim and Schaffer [2] for information.

The complex cepstrum transformation is central to the theory and application of *homomorphic systems*, that is, systems that obey certain general rules of superposition. See Oppenheim and Schaffer [3] for a discussion of the complex cepstrum and homomorphic transformations, with details on speech processing applications.

Try using `cceps` in an echo detection application. First, create a 45 Hz sine wave sampled at 100 Hz:

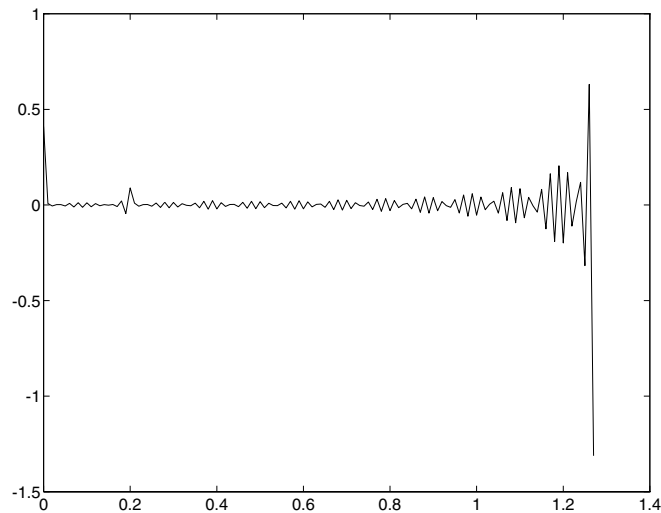
```
t = 0:0.01:1.27;
s1 = sin(2*pi*45*t);
```

Add an echo of the signal, with half the amplitude, 0.2 seconds after the beginning of the signal.

```
s2 = s1 + 0.5*[zeros(1,20) s1(1:108)];
```

The complex cepstrum of this new signal is

```
c = cceps(s2);
plot(t,c)
```



Note that 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.

$$c_x = \frac{1}{2\pi} \int_{-\pi}^{\pi} \log |X(e^{j\omega})| e^{j\omega n} d\omega$$

The toolbox function `rceps` performs this operation, returning the real cepstrum for a sequence  $x$ . The returned sequence is a real-valued vector the same size as the input vector:

```
y = rceps(x)
```

By definition, you cannot reconstruct the original sequence from its real cepstrum transformation, as the real cepstrum is based only on the magnitude of the Fourier transform for the sequence (see Oppenheim and Schaffer [3]). The `rceps` function also returns a unique minimum-phase sequence that has the same real cepstrum as  $x$ . 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  $y_m$  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 =
    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. For example:

```
x = 1:10;
[xhat,delay] = cceps(x)

xhat =
  Columns 1 through 7
  2.2428 -0.0420 -0.0210  0.0045  0.0366  0.0788  0.1386
  Columns 8 through 10
  0.2327  0.4114  0.9249

delay =
  1
```

To invert the complex cepstrum, use `icceps` with the original delay parameter:

```
icc = icceps(xhat,2)

ans =
  Columns 1 through 7
  2.0000  3.0000  4.0000  5.0000  6.0000  7.0000  8.0000
  Columns 8 through 10
  9.0000 10.0000  1.0000
```

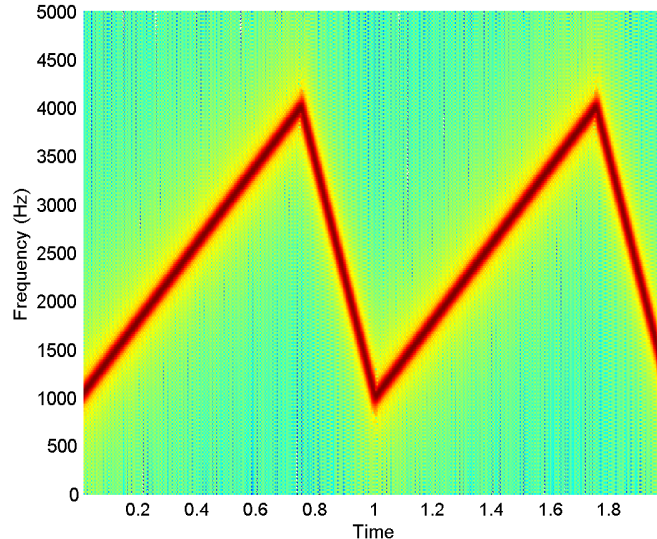
As shown in the above example, with any modification of the complex cepstrum, the original delay term may no longer be valid. You will not be able to invert the complex cepstrum exactly.

## FFT-Based Time-Frequency Analysis

The Signal Processing Toolbox provides a function, `spectrogram`, that returns the time-dependent Fourier transform for a sequence, or displays this information as a spectrogram. The Toolbox also includes a spectrogram demo. 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 linear FM signal:

```
fs = 10000;  
t = 0:1/fs:2;  
x = vco(sawtooth(2*pi*t,.75),[0.1 0.4]*fs,fs);  
spectrogram(x,kaiser(256,5),220,512,fs,'yaxis')
```



Note that the spectrogram display is an image, not a plot.

## 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  $(n-1)/2$  and  $(n-1)/2 + 1$  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 documentation for information on two-dimensional median filtering.

## Communications Applications

The toolbox provides three functions for communications simulation.

Operation	Function
Modulation	modulate
Demodulation	demod
Voltage controlled oscillation	vco

*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
<code>amdsb-sc</code> or <code>am</code>	Amplitude modulation, double side-band, suppressed carrier
<code>amdsb-tc</code>	Amplitude modulation, double side-band, transmitted carrier
<code>amssb</code>	Amplitude modulation, single side-band
<code>fm</code>	Frequency modulation
<code>pm</code>	Phase modulation
<code>ppm</code>	Pulse position modulation
<code>pwm</code>	Pulse width modulation
<code>qam</code>	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)
```

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

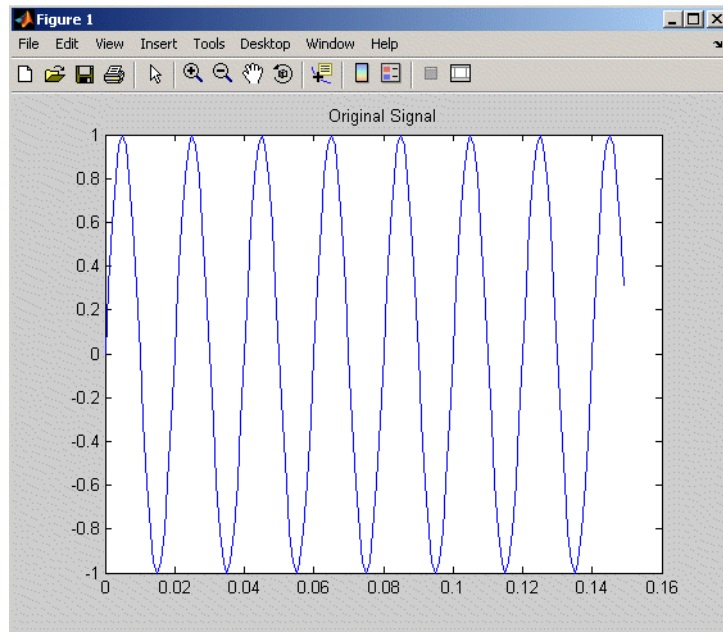
```
t = (0:1/1000:2);  
x = sin(2*pi*50*t);
```

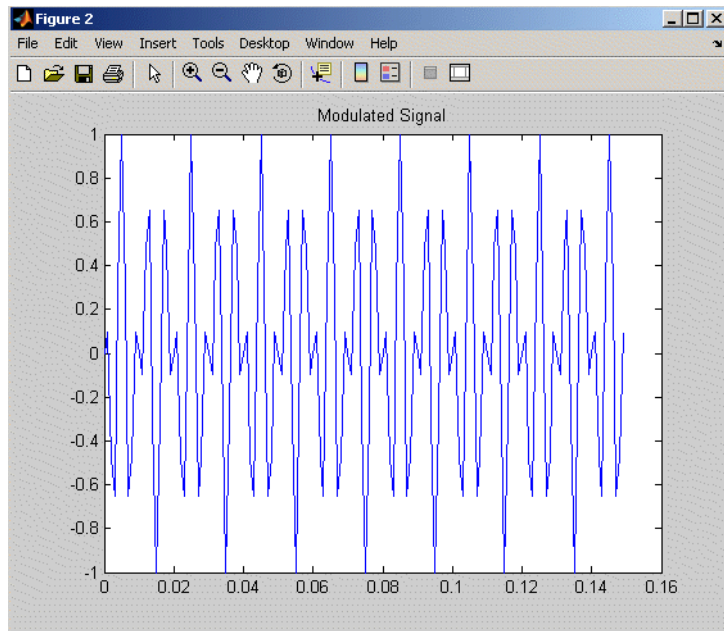
With a carrier frequency of 200 Hz, the modulated and demodulated versions of this signal are

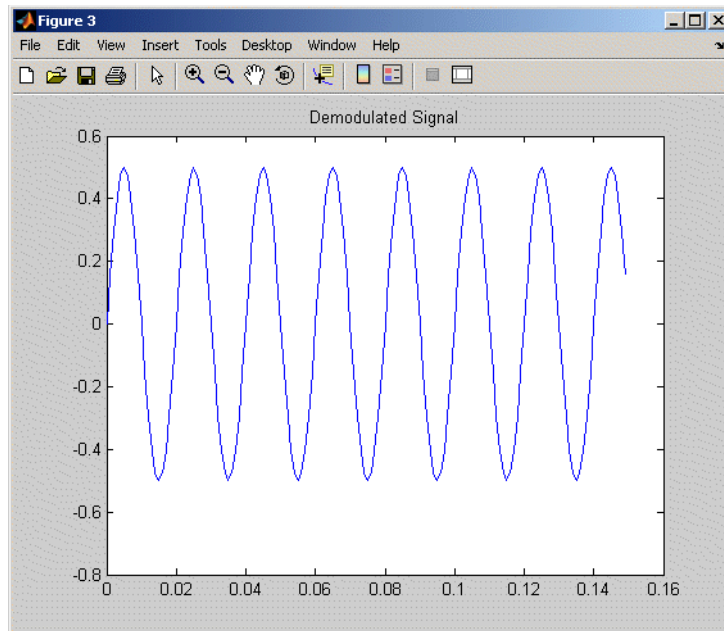
```
y = modulate(x,200,1000,'am');  
z = demod(y,200,1000,'am');
```

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');
```







---

**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.

---

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

$$y = \text{vco}(x, f_c, f_s)$$

where `fc` is the carrier frequency and `fs` is the sampling frequency.

To scale the frequency modulation range, use

$$y = \text{vco}(x, [F_{\min} F_{\max}], f_s)$$

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 4-32 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` (see Chapter 1, “Signal Processing Basics” for a description of the convolution function):

```
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
```

## Specialized Transforms

In addition to the discrete Fourier transform (see `fft`), the Signal Processing Toolbox and the MATLAB environment together provide the following transform functions:

- The chirp  $z$ -transform (CZT), 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 discrete cosine transform (DCT), closely related to the DFT. The DCT's energy compaction properties are useful for applications like signal coding.
- The Hilbert transform, which facilitates the formation of the analytic signal. The analytic signal is useful in the area of communications, particularly in bandpass signal processing.

### Chirp $z$ -Transform

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

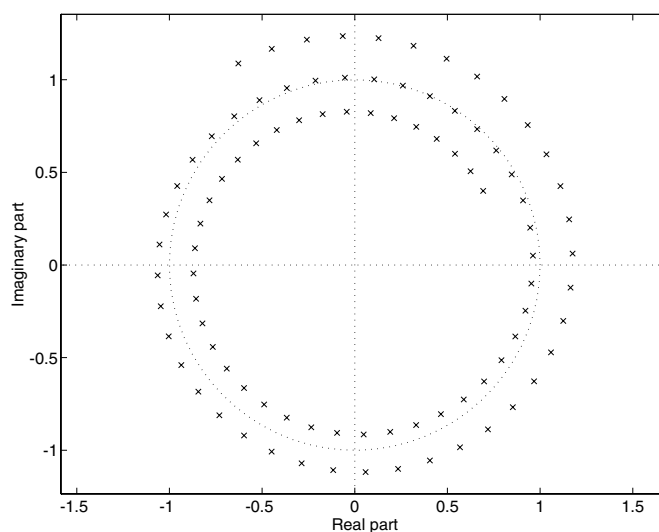
$$z_l = AW^{-l}, \quad l = 0, \dots, M-1$$

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(j*pi/6);  
W = 0.995*exp(-j*pi*.05);  
M = 91;  
z = A*(W.^(-(0:M-1)));  
zplane([],z.')
```





$\text{czt}(x, M, W, A)$  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

$$y = \text{czt}(x)$$

$\text{czt}$  may be faster than the  $\text{fft}$  function for computing the DFT of sequences with certain odd lengths, particularly long prime-length sequences.

## Discrete Cosine Transform

The toolbox function `dct` computes the unitary discrete cosine transform, or DCT, for an input vector or matrix. Mathematically, the unitary DCT of an input sequence  $x$  is

$$y(k) = w(k) \sum_{n=1}^N x(n) \cos \frac{\pi(2n-1)(k-1)}{2N}, \quad k = 1, \dots, N$$

where

$$w(k) = \begin{cases} \frac{1}{\sqrt{N}}, & k = 1 \\ \sqrt{\frac{2}{N}}, & 2 \leq k \leq N \end{cases}$$

The DCT is closely related to the discrete Fourier transform; the DFT is actually one step in the computation of the DCT for a sequence. The DCT, however, has better *energy compaction* properties, with just a few of the transform coefficients representing the majority of the energy in the sequence. The energy compaction properties of the DCT make it useful in applications such as data communications.

The function `idct` computes the inverse DCT for an input sequence, reconstructing a signal from a complete or partial set of DCT coefficients. The inverse discrete cosine transform is

$$x(n) = \sum_{k=1}^N w(k) y(k) \cos \frac{\pi(2n-1)(k-1)}{2N}, \quad n = 1, \dots, N$$

where

$$w(n) = \begin{cases} \frac{1}{\sqrt{N}}, & n = 1 \\ \sqrt{\frac{2}{N}}, & 2 \leq n \leq N \end{cases}$$

Because of the energy compaction mentioned above, it is possible to reconstruct a signal from only a fraction of its DCT coefficients. For example, generate a 25 Hz sinusoidal sequence, sampled at 1000 Hz:

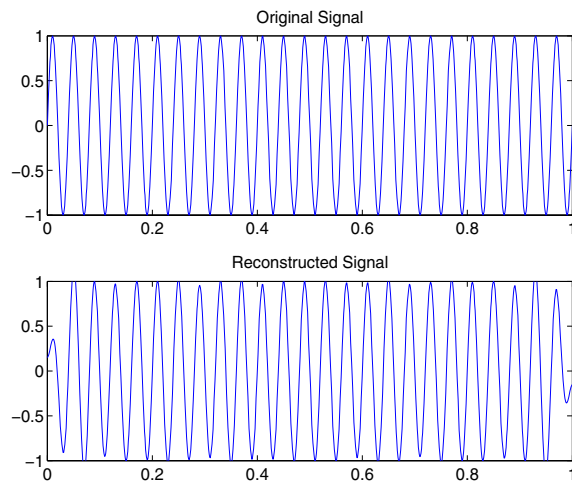
```
t = (0:1/999:1);  
x = sin(2*pi*25*t);
```

Compute the DCT of this sequence and reconstruct the signal using only those components with value greater than 0.1 (64 of the original 1000 DCT coefficients):

```
y = dct(x)           % Compute DCT  
y2 = find(abs(y) < 0.9); % Use 17 coefficients  
y(y2) = zeros(size(y2)); % Zero out points < 0.9  
z = idct(y);         % Reconstruct signal using inverse DCT
```

Plot the original and reconstructed sequences:

```
subplot(2,1,1); plot(t,x);  
title('Original Signal')  
subplot(2,1,2); plot(t,z), axis([0 1 -1 1])  
title('Reconstructed Signal')
```



One measure of the accuracy of the reconstruction is

$$\text{norm}(x - z) / \text{norm}(x)$$

that is, the norm of the difference between the original and reconstructed signals, divided by the norm of the original signal. In this case, the relative error of reconstruction is 0.1443. The reconstructed signal retains approximately 85% of the energy in the original signal.

## Hilbert Transform

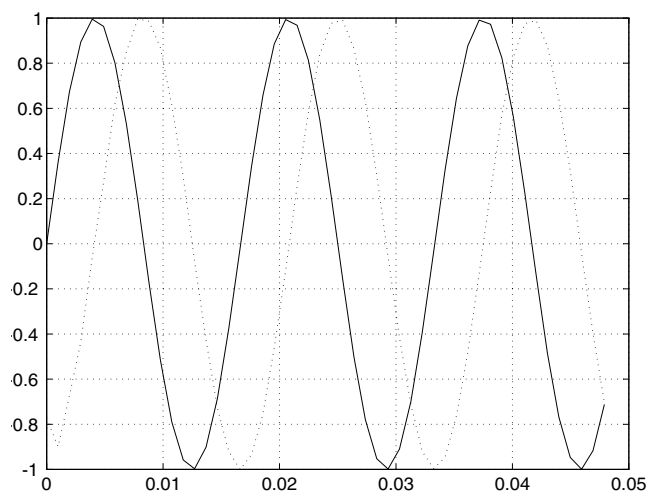
The toolbox function `hilbert` computes the Hilbert transform for a real input sequence `x` and returns a complex result of the same length

$$y = \text{hilbert}(x)$$

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° phase shift; sines become cosines and vice versa. To plot a portion of data (solid line) and its Hilbert transform (dotted line):

```
t = (0:1/1023:1);  
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
```



The analytic signal is useful in calculating *instantaneous attributes* of a time series, the attributes of the series at any point in time. The instantaneous amplitude of the input sequence is the amplitude of the analytic signal. The instantaneous phase angle of the input sequence is the (unwrapped) angle of the analytic signal; the instantaneous frequency is the time rate of change of the instantaneous phase angle. You can calculate the instantaneous frequency using the MATLAB function, `diff`.

## **Selected Bibliography**

- [1] Kay, S.M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice Hall, 1988.
- [2] Oppenheim, A.V., and R.W. Schaffer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.
- [3] Oppenheim, A.V., and R.W. Schaffer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975, Section 10.5.3.
- [4] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987.
- [5] Pratt, W.K. *Digital Image Processing*. New York: John Wiley & Sons, 1991.

# FDATool: A Filter Design and Analysis GUI

---

The following chapter describes the Filter Design and Analysis Tool (FDATool) and provides a detailed example showing how to use this Graphical User Interface.

Overview (p. 5-2)	Introduction to the tool
Opening FDATool (p. 5-8)	How to start the tool
Choosing a Response Type (p. 5-9)	Setting the filter response type
Choosing a Filter Design Method (p. 5-10)	Selecting a design method
Setting the Filter Design Specifications (p. 5-11)	Setting the filter parameters
Computing the Filter Coefficients (p. 5-16)	Calculating the filter
Analyzing the Filter (p. 5-17)	Tools for analyzing the filter
Editing the Filter Using the Pole/Zero Editor (p. 5-25)	Changing the filter by changing poles or zeros
Converting the Filter Structure (p. 5-29)	Changing the filter structure
Importing a Filter Design (p. 5-32)	Bringing a filter design into the tool
Exporting a Filter Design (p. 5-36)	Sending the filter design outside the tool
Generating a C Header File (p. 5-44)	Create C code of the filter design
Generating an M-File (p. 5-47)	Creating MATLAB code of the filter design
Managing Filters in the Current Session (p. 5-48)	Working with multiple filters
Saving and Opening Filter Design Sessions (p. 5-50)	Working with tool sessions

## Overview

The Filter Design and Analysis Tool (FDATool) is a powerful user interface for designing and analyzing filters quickly. FDATool 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. FDATool also provides tools for analyzing filters, such as magnitude and phase response and pole-zero plots. FDATool seamlessly integrates additional functionality from other MathWorks products as described in the following table.

<b>Product</b>	<b>Added Features</b>
Embedded Target for Texas Instruments C6000™ DSP	Download code to C6000 DSP target board
Filter Design HDL Coder	Generate synthesizable VHDL or Verilog for fixed-point filters
Filter Design Toolbox	<ul style="list-style-type: none"> <li>• Advanced FIR and IIR design techniques (see “Advanced Filter Design Methods” on page 5-4)</li> <li>• Filter transformations</li> <li>• Multirate filters</li> <li>• Fixed-point filters (available only with the Fixed-Point Toolbox)</li> </ul>
Link for Code Composer Studio™ Development Tools	Export code usable by Code Composer Studio
Signal Processing Blockset	Generate equivalent Signal Processing Blockset block for the filter
Simulink	Generate filters from atomic Simulink blocks



## Filter Design Methods

FDATool gives you access to the following filter design methods in the Signal Processing Toolbox.

<b>Design Method</b>	<b>Function</b>
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 in FDATool, all window functions in the Signal toolbox 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 the Filter Design Toolbox.

<b>Design Method</b>	<b>Function</b>
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
Second-order IIR peaking (resonator)	iirpeak

## Using the Filter Design and Analysis Tool

There are different ways that you can design filters using the Filter Design and Analysis Tool. 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, and create a C header file containing the coefficients, and analyze different filter responses in FDATool or in a separate Filter Visualization Tool (`fvttool`). See “Analyzing the Filter” on page 5-17 for more information. 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 Design and Analysis Tool Panels

The Filter Design and Analysis Tool has sidebar buttons that display particular panels in the lower half of the tool. The panels are

- **Design Filter.** See “Choosing a Filter Design Method” on page 5-10 for more information. You use this panel to
  - Design filters from scratch.
  - Modify existing filters designed in FDATool.
  - Analyze filters.
- **Import filter.** See “Importing a Filter Design” on page 5-32 for more information. 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 5-25. You use this panel to add, delete, and move poles and zeros in your filter design.


If you also have the Filter Design Toolbox installed, additional panels are available:

- **Set quantization parameters** — Use this panel to quantize double-precision filters that you design in FDATool, quantize double-precision filters that you import into FDATool, 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. See “Exporting to Simulink” on page 5-40 for information.

## Getting Help

At any time, you can right-click or click the **What's this?** button, , to get information on the different parts of the tool. You can also use the **Help** menu to see complete Help information.

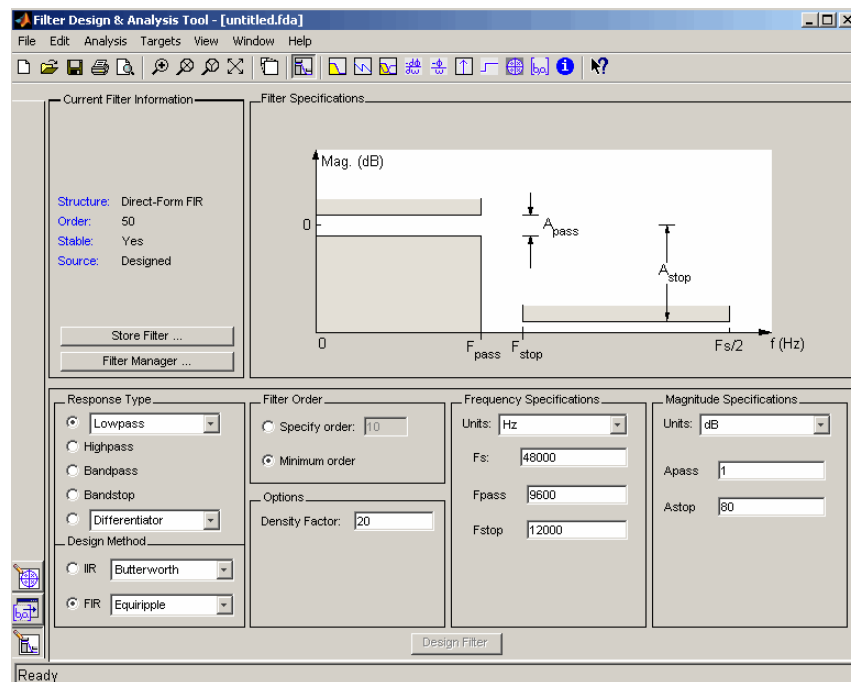
## Opening FDATool

To open the Filter Design and Analysis Tool (FDATool), type

```
fdatool
```

The Filter Design and Analysis Tool opens with the Design Filter panel displayed.

**Note** If you are viewing this online, click in the figure below to go to a description of the procedure for that area of the figure.



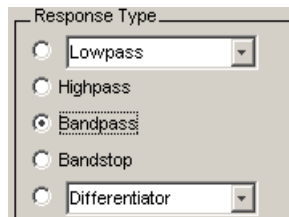
## Choosing a Response Type

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 the Filter Design Toolbox installed.

To design a bandpass filter, select the radio button next to **Bandpass** in the **Response Type** region of the GUI.



---

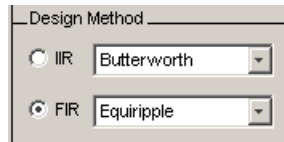
**Note** Not all filter design methods are available for all response types. Once you choose your response type, this may restrict the filter design methods available to you. Filter design methods that are not available for a selected response type are removed from the **Design Method** region of the GUI.

---

## 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 GUI.

To select the Remez algorithm to compute FIR filter coefficients, select the **FIR** radio button and choose Equiripple from the list of methods.





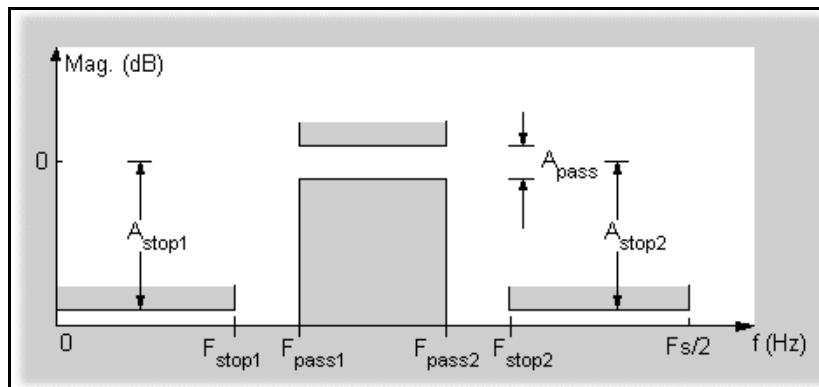
## Setting the Filter Design Specifications

The filter design specifications that you can set vary according to response type and design method. For example, to design a bandpass filter, you can enter

- “Filter Order” on page 5-12
- “Options” on page 5-13
- “Bandpass Filter Frequency Specifications” on page 5-14
- “Bandpass Filter Magnitude Specifications” on page 5-15

The display region illustrates filter specifications when you select **Filter Specifications** from the **Analysis** menu 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 **Specification Mask** from the **View** menu.

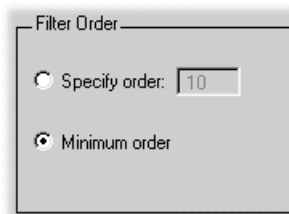


### Filter Order

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.

Select the **Minimum order** radio button for this example.



Note that filter order specification options depend on the filter design method you choose. Some filter methods may not have both options available.

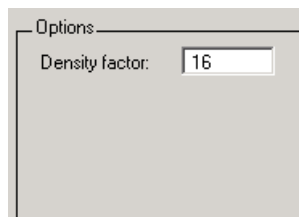
## Options

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
Tukey (tukeywin)	Alpha
User Defined	Function Name, Parameter

You can view the window in the Window Visualization Tool (`wvtool`) by clicking the View button.

For this example, set the **Density factor** to 16.



## Bandpass Filter Frequency Specifications

For a bandpass filter, you can set

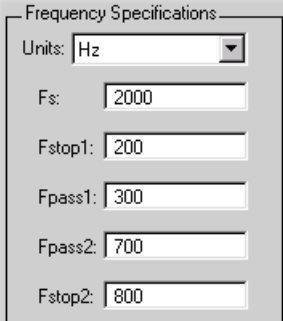
- Units of frequency (Hz, kHz, MHz, or Normalized (0 to 1))
- Sampling frequency
- Passband frequencies
- Stopband frequencies

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.

For this example:

- Keep the units in **Hz** (default).
- Set the sampling frequency (**F<sub>s</sub>**) to 2000 Hz.
- Set the end of the first stopband (**F<sub>stop1</sub>**) to 200 Hz.
- Set the beginning of the passband (**F<sub>pass1</sub>**) to 300 Hz.
- Set the end of the passband (**F<sub>pass2</sub>**) to 700 Hz.
- Set the beginning of the second stopband (**F<sub>stop2</sub>**) to 800 Hz.



Frequency Specifications

Units: Hz

Fs: 2000

Fstop1: 200

Fpass1: 300

Fpass2: 700

Fstop2: 800

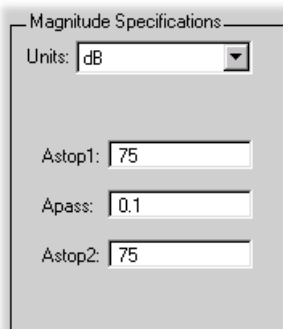
## Bandpass Filter Magnitude Specifications

For a bandpass filter, you can specify the following magnitude response characteristics:

- Units for the magnitude response (dB or linear)
- Passband ripple
- Stopband attenuation

For this example:

- Keep **Units** in dB (default).
- Set the passband ripple (**A<sub>pass</sub>**) to 0.1 dB.
- Set the stopband attenuation for both stopbands (**A<sub>stop1</sub>**, **A<sub>stop2</sub>**) to 75 dB.



The image shows a dialog box titled "Magnitude Specifications". It contains four input fields:

- Units: dB (selected in a dropdown menu)
- A<sub>stop1</sub>: 75
- A<sub>pass</sub>: 0.1
- A<sub>stop2</sub>: 75

## **Computing the Filter Coefficients**

Now that you've specified the filter design, click the Design Filter button to compute the filter coefficients.

Notice that 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

Once you've designed the filter, you can view the following filter response characteristics in the display region or in a separate window (see "Displaying the Response in FVTool" on page 5-23):


- 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.

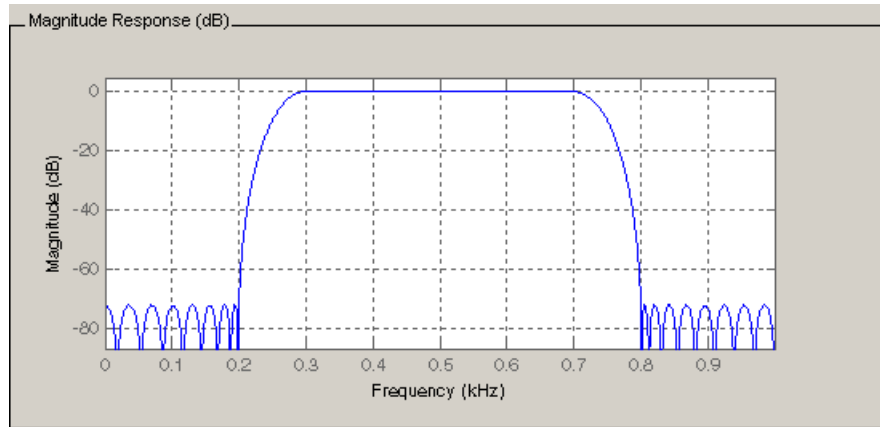
You can display two responses in the same plot by selecting **Overlay Analysis** from the **Analysis** menu and selecting an available response. A second *y*-axis is added to the right side of the response plot. (Note that not all responses can be overlaid on each other.)

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 on the *y*-axis of the plot and select **Zero-phase** from the context menu.

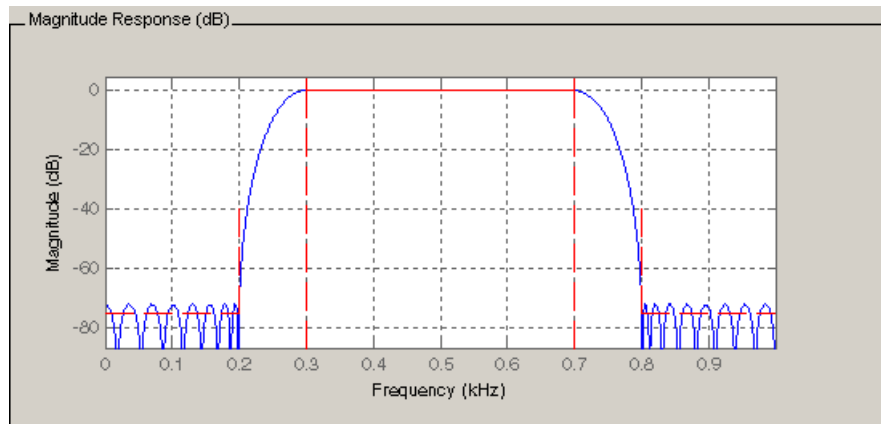


For example, to look at the filter's magnitude response, select the Magnitude Response button  on the toolbar.



You can also overlay the filter specifications on the Magnitude plot by selecting **Specification Mask** from the **View** menu.

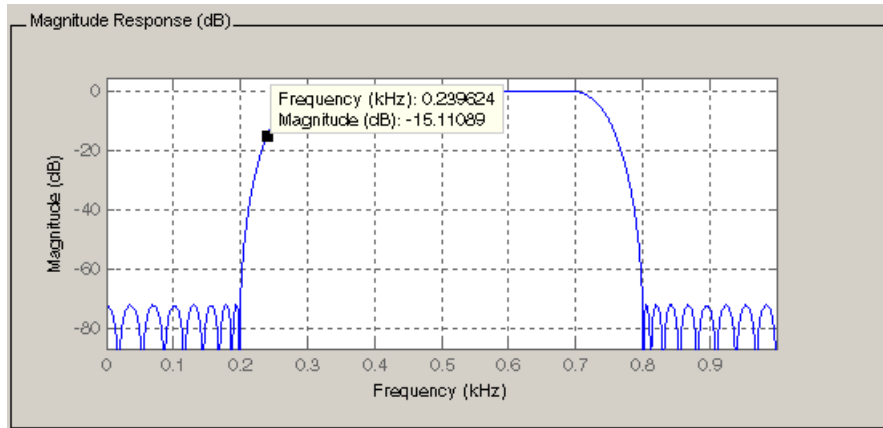
**Note** You can use specification masks in FVTool only if FVTool was launched from FDATool.





## Using Data Markers

You can click on the response to add plot data markers that display information about particular points on the response.



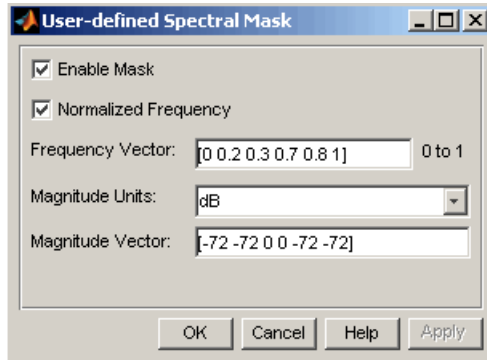
To move a data marker, grab its black square at the corner of the marker. Dragging the marker with your mouse changes the Frequency and Magnitude values.

To change the properties of a data marker, right-click on the marker to display the properties menu:

- **Alignment** — Change the position of the marker. Available options are top-right, top-left, bottom-right, and bottom-left.
- **Font Size** — Change the font size.
- **Movable** — Allow the marker to be moved on the response.
- **Interpolation** — Select Nearest to force the marker to snap to nearest point along the plotted curve. Select Linear to interpolate between points along the plotted curve.
- **Track Mode** — Restrict the marker to be movable in the x, y, or xy direction.
- **Delete** — Delete the selected marker.
- **Delete all** — Delete all markers.

## Drawing Spectral Masks

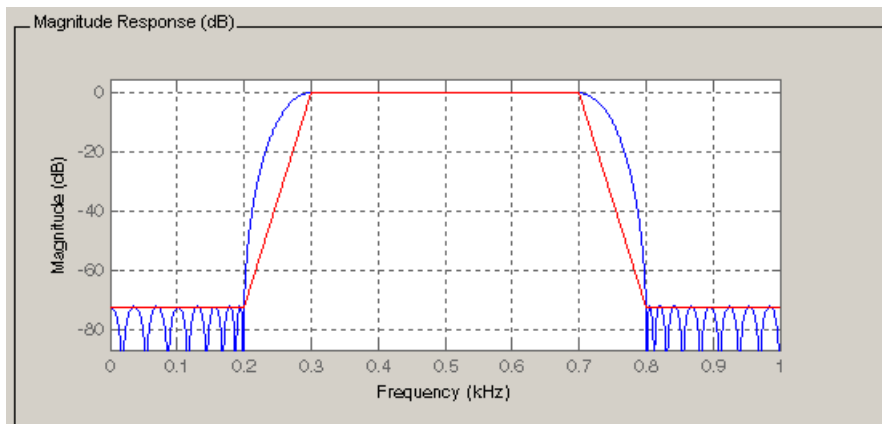
To add spectral masks or rejection area lines to your magnitude plot, click **User-defined Spectral Mask** from the **View** menu.



The mask is defined by a frequency vector and a magnitude vector. These vectors must be the same length.

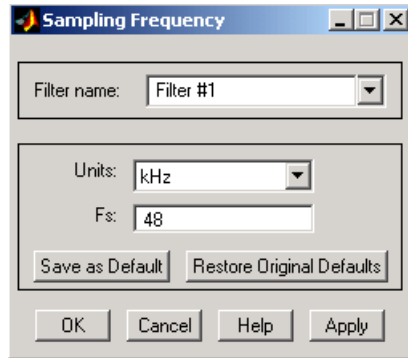
- **Enable Mask** — Select to turn on the mask display.
- **Normalized Frequency** — Select to normalize the frequency between 0 and 1 across the displayed frequency range.
- **Frequency Vector** — Enter a vector of  $x$ -axis frequency values.
- **Magnitude Units** — Select the desired magnitude units. These units should match the units used in the magnitude plot.
- **Magnitude Vector** — Enter a vector of  $y$ -axis magnitude values.

The magnitude response below shows a spectral mask.



## Changing the Sampling Frequency

To change the sampling frequency 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 sampling frequency, select the desired unit from **Units** and enter the sampling frequency in **Fs**. (For each filter in `fvtool`, you can specify a different sampling frequency or you can apply the sampling frequency to all filters.)

To save the displayed parameters as the default values to use when `FDATool` or `FVTool` is opened, click **Save as Default**.

To restore the MATLAB defined default values, click **Restore Original Defaults**.

---

## Displaying the Response in FVTool

To display the filter response characteristics in a separate window, select **Filter Visualization Tool** from the **View** menu (available if any analysis, except the filter specifications, is in the display region) or click the **Full View Analysis** button:



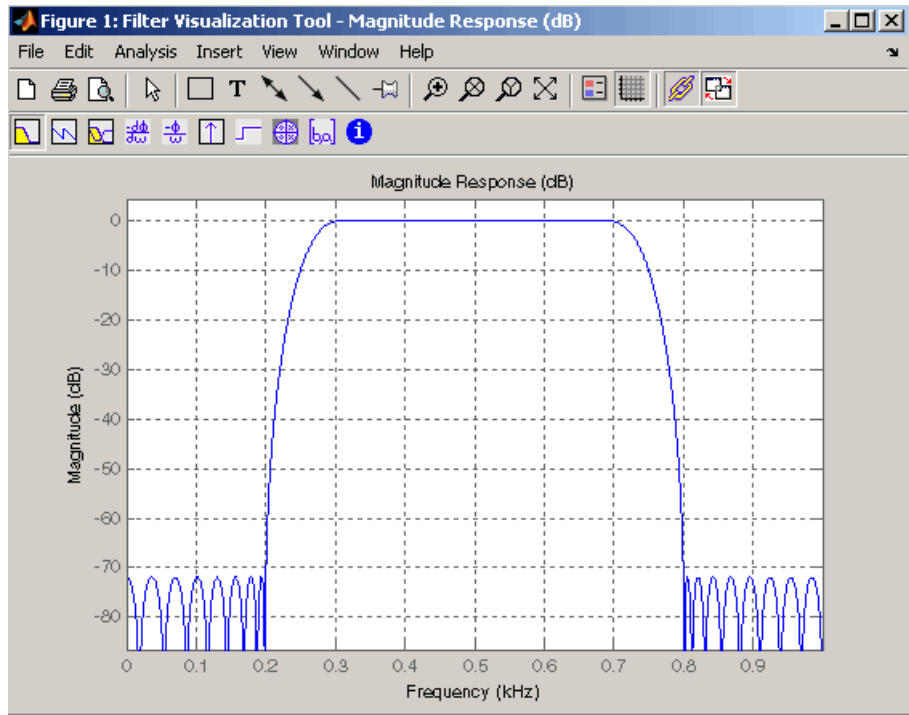
This launches the Filter Visualization Tool (fvtool).

---

**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 FDATool and FVTool so that changes made in FDATool are immediately reflected in FVTool. See `fvtool` for more information.



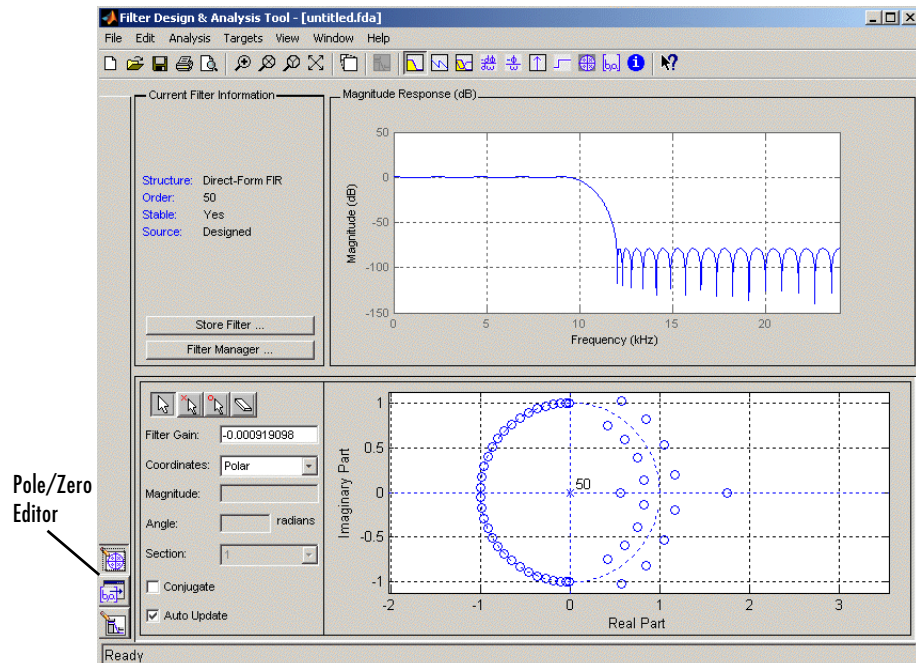
## Editing the Filter Using the Pole/Zero Editor

**Note** You cannot generate an M-file (**File > Generate M-file**) if your filter was designed or edited with the Pole/Zero Editor.

You can edit a designed or imported filter's coefficients by moving, deleting, or adding poles and/or zeros using the Pole/Zero Editor panel.

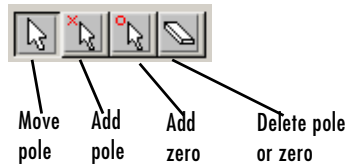
**Note** You cannot move quantized poles and zeros. You can only move the reference poles and zeros.

Click the Pole/Zero Editor button in the sidebar or select **Pole/Zero Editor** from the **Edit** menu to display this panel.



Poles are shown using **x** symbols and zeros are shown using **o** symbols.

**Plot mode buttons.** 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 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.

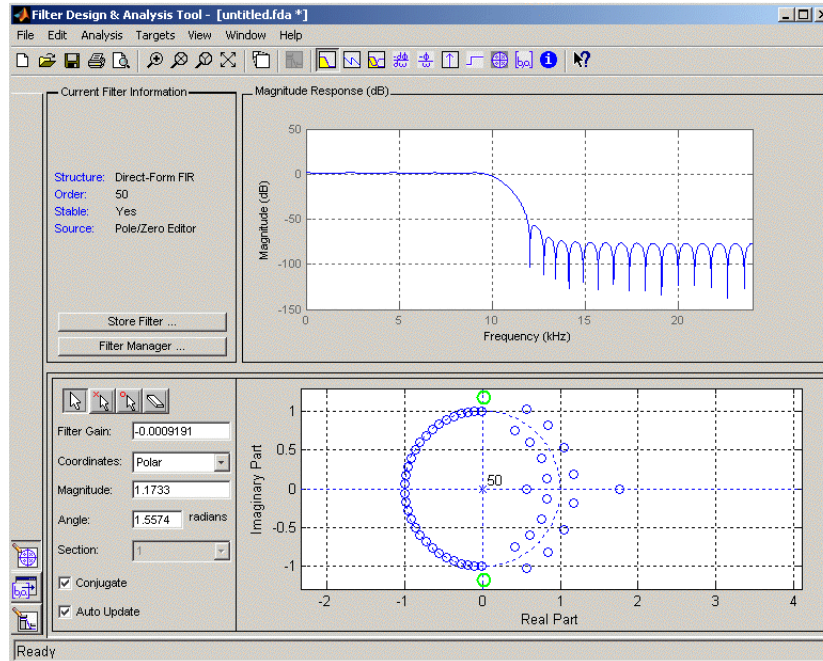
The **Pole-zero Editor** on the **Edit** menu has items for selecting multiple poles/zeros, for inverting and mirroring poles/zeros, and for deleting, scaling and rotating poles/zeros.



Select All
Select All Poles
Select All Zeros
Select Inside Unit Circle
Select Outside Unit Circle
Select Below Real Axis
Select Above Real Axis
Select Left Half
Select Right Half
Invert about the Real Axis
Invert about the Imaginary Axis
Invert about the Unit Circle
Mirror about the Real Axis
Mirror about the Imaginary Axis
Mirror about the Unit Circle
Delete current
Scale by a Factor
Rotate Counter-Clockwise

**These items are enabled only when one or more poles/zeros are selected.**

Moving one of the zeros on the vertical axis produces the following result:



- The selected zero pair is shown in green.
- When you select one of the zeros from a conjugate pair, the **Conjugate** check box and the conjugate are automatically selected.
- The **Magnitude Response** plot updates immediately because **Auto update** is active.

## 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 installed the Filter Design Toolbox 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

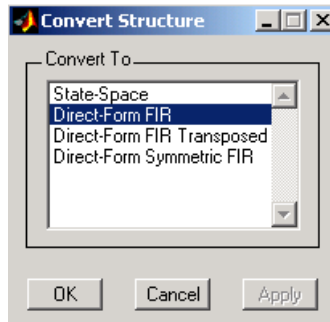
---

**Note** Converting from one filter structure to another 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.

---

For example:

- Select **Convert Structure** from the **Edit** menu 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 **Convert to Second-order Sections** on the **Edit** menu 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 the **Edit** menu **Order and Scale SOS**, 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 **Convert to Second-order Sections** from the **Edit** menu 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.

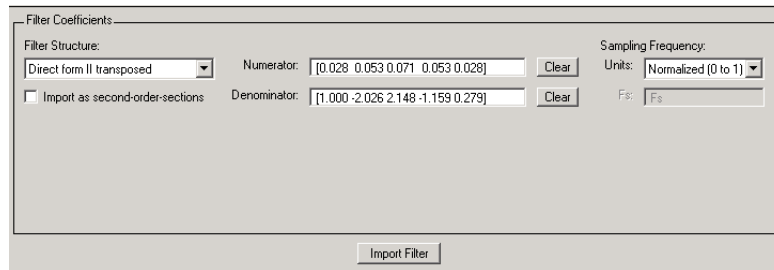
---

**Note** To convert from second-order sections back to a single section, use **Convert to Single Section** on the **Edit** menu.

---

## Importing a Filter Design

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 screenshot shows the "Filter Coefficients" panel. It contains the following elements:

- Filter Structure:** A pull-down menu currently set to "Direct form II transposed".
- Numerator:** A text input field containing the values "[0.028 0.053 0.071 0.053 0.028]" with a "Clear" button to its right.
- Denominator:** A text input field containing the values "[1.000 -2.026 2.148 -1.159 0.279]" with a "Clear" button to its right.
- Import as second-order-sections:** An unchecked checkbox.
- Sampling Frequency:** A section containing a "Units:" pull-down menu set to "Normalized (0 to 1)" and an "Fs:" input field containing "Fs".
- Import Filter:** A button located at the bottom center of the panel.

The imported filter can be in any of the representations listed in the **Filter Structure** pull-down menu and described in “Filter Structures” on page 5-33. 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 sampling frequency 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 (see “Editing the Filter Using the Pole/Zero Editor” on page 5-25).

## 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

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(m+1)z^{-m}}{a(1) + a(2)z^{-1} + \dots + a(n+1)z^{-n}}$$

- 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 1-35 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:

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

The **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

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

whose rows contain the numerator and denominator coefficients  $b_{ik}$  and  $a_{ik}$  of the second-order sections of  $H(z)$ .

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 1-40 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 1-40 for more information.



**Discrete-time Filter (dfilt object)**

For Discrete-time filter, specify the name of the `dfilt` object. See `dfilt` for more information.

**Multirate Filter (mfilt object)**

For Multirate filter, specify the name of the `mfilt` object. See `mfilt` in the Filter Design Toolbox for more information.

## Exporting a Filter Design

You can save your filter design by

- “Exporting Coefficients or Objects to the Workspace” on page 5-36
- “Exporting Coefficients to an ASCII File” on page 5-38
- “Exporting Coefficients or Objects to a MAT-File” on page 5-38
- “Exporting to SPTool” on page 5-39
- “Exporting to Simulink” on page 5-40

You can also send your filter to a C header file or generate an M-file. The M-file contains code that replicates the filter you designed. See the following sections:

- “Generating a C Header File” on page 5-44
- “Generating an M-File” on page 5-47

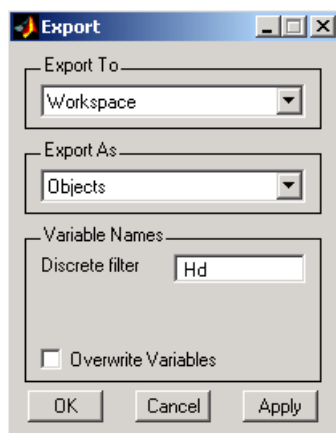
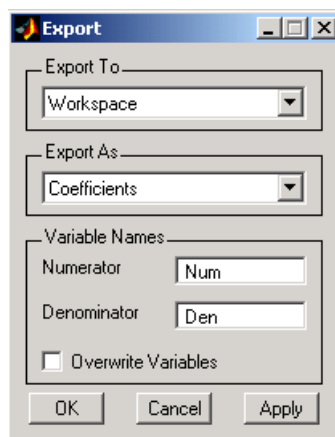
### Exporting Coefficients or Objects to the Workspace

You can save the filter either as filter coefficients variables or as a `dfilt` or `mfilt` filter object variable. (Note that you must have the Filter Design Toolbox installed to save as an `mfilt`.) To save the filter to the MATLAB workspace:

- 1 Select **Export** from the **File** menu. 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** (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.



## Exporting Coefficients to an ASCII File

To save filter coefficients to a text file,

- 1 Select **Export** from the **File** menu. 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 filename 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 **Export** from the **File** menu. 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 filename and click the **Save** button.

See also “Saving and Opening Filter Design Sessions” on page 5-50.

## Exporting to SPTool

You may want to use your designed filter in SPTool to do signal processing and analysis.

---

**Note** The magnitude response you see in SPTool will differ from the one in FDATool because the sampling frequency is preset at  $F_s = 2$  when a filter is exported from FDATool to SPTool.

---

- 1 Select **Export** from the **File** menu. The **Export** dialog box appears.
- 2 Select SPTool 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 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.

SPTool opens and the current FDATool filter appears in the **Filter** area list as the specified variable name followed by (Imported).

---

**Note** If you are using the Filter Design Toolbox and export a quantized filter, only the values of its quantized coefficients are exported. The reference coefficients are not exported. SPTool does not restrict the coefficient values, so if you edit them in SPTool by moving poles or zeros, the filter will no longer be in quantized form.

---

## Exporting to Simulink

If you have Simulink installed, you can export a Simulink block of your filter design and insert it into a new or existing Simulink model.

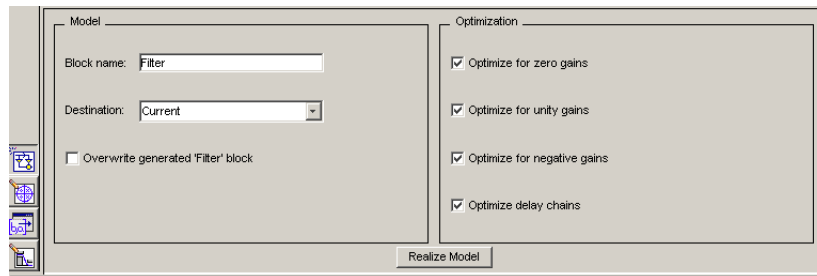
You can export a filter designed using any filter design method available in FDATool.

---

**Note** If you have the Filter Design Toolbox, you can export a CIC filter to Simulink, if you also have these toolboxes and blocksets installed in addition to Simulink: the Fixed-Point Toolbox and the Signal Processing Blockset.

---

- 1 After designing your filter, click the Realize Model sidebar button or select **Export to Simulink Model** from the **File** menu. The Realize Model panel is displayed.



- 2 Specify the name to use for your block in **Block name**.
- 3 Select the **Destination** — Current to insert the block into the current (most recently selected) Simulink model or New to open a new model.
- 4 If you want to overwrite a block previously created from this panel, check **Overwrite generated 'Filter' block**.

---

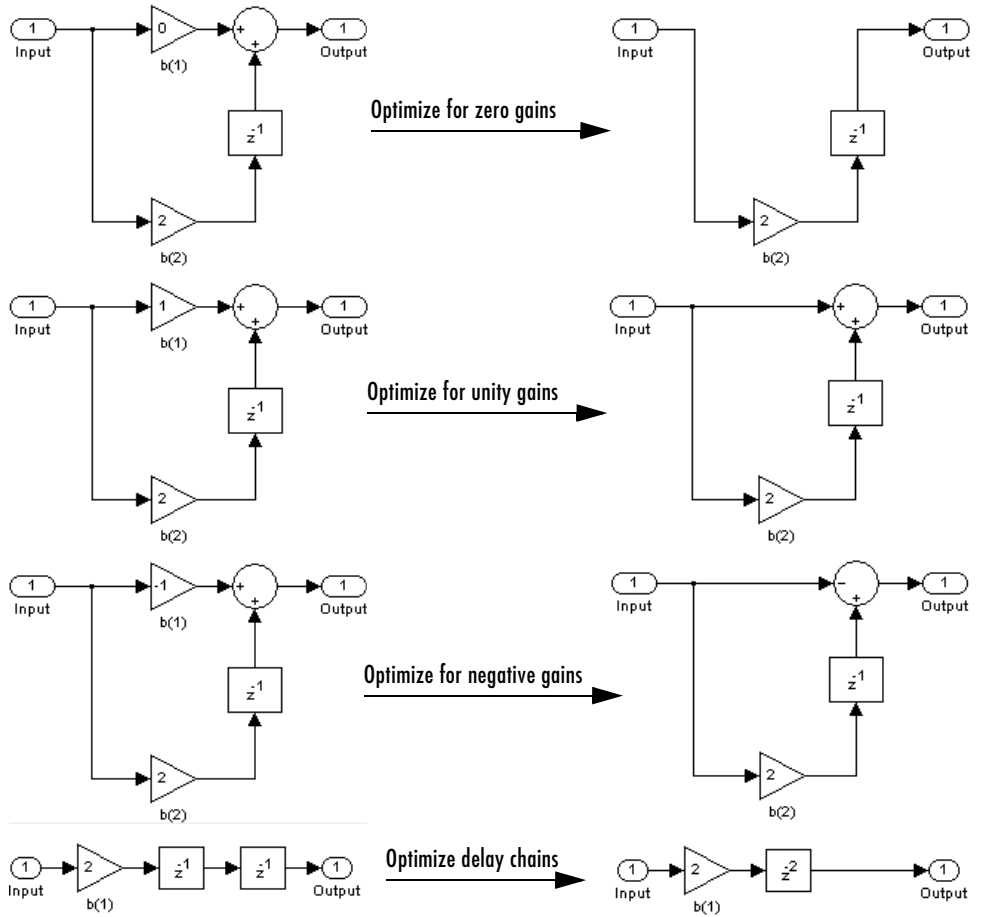
**Note** If you have the Signal Processing Blockset installed, a **Build model using basic elements** check box is included. If you deselect it, a Digital Filter block is created instead of a subsystem block, which uses separate subelements. See the Filter Realization Wizard and Choosing Between Filter Design Blocks in the Signal Processing Blockset documentation for information.

---

5 Select the desired optimization(s) for your block:

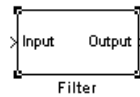
- **Optimize for zero gains** — Removes zero-valued gain paths from the filter structure.
- **Optimize for unity gains** — Substitutes a wire (short circuit) for gains equal to 1 in the filter structure.
- **Optimize for negative gains** — Substitutes a wire (short circuit) for gains equal to -1 and 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  $n$ .

The following illustration shows the effects of each optimization:

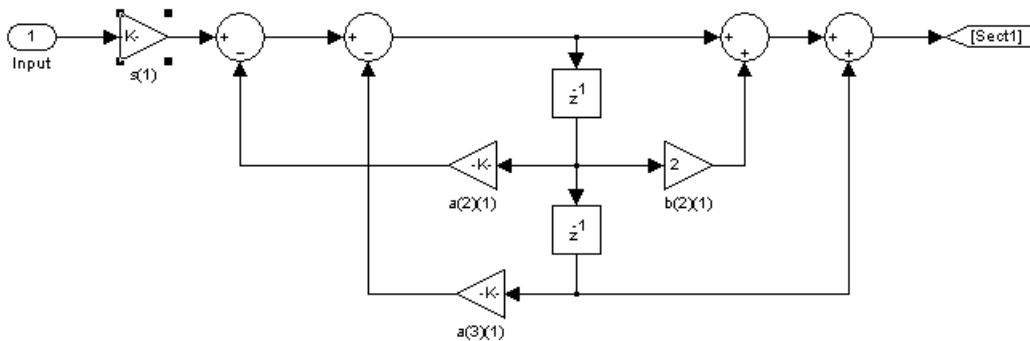




- 6 Click the Realize Model button to create the filter block. The filter is implemented as a subsystem block using Sum, Gain, and Integer Delay blocks.



If you double-click the Filter block in Simulink, the filter structure is displayed. The following figure shows the first section of the default four-section, direct form II filter.



## 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 **Generate C Header** from the **Targets** menu. The **Generate C Header** dialog box appears.

Generate C Header

Variable names in C header file

Numerator: NUM Numerator length: NL

Denominator: DEN Denominator length: DL

Number of sections: NS

Data type to use in export

Export suggested: Double-precision floating point

Export as: Signed 32-bit integer  
Fractional length: 31

OK Cancel Help Apply

- 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

<b>Filter Structure</b>	<b>Variable Parameter</b>
Direct-form I Direct-form II Direct-form I transposed Direct-form II transposed	Numerator, Numerator length*, Denominator, Denominator length*, and Number of sections (inactive if filter has only one section)
Lattice ARMA	Lattice coeffs, Lattice coeffs length*, Ladder coeffs, Ladder coeffs length*, Number of sections (inactive if filter has only one section)
Lattice MA	Lattice coeffs, Lattice coeffs 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 the Filter Design Toolbox 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 FDATool. This is due to rounding and truncating differences.

---

- 4 Click **OK** to save the file and close the dialog box or click **Apply** to save the file, but leave the dialog box open for additional C header file definitions.

## Generating an M-File

You can generate an M-file that contains all the code used to create the filter you designed in FDATool. Select **Generate M-file** from the **File** menu and specify the filename in the **Generate M-file** dialog box.

---

**Note** You cannot generate an M-file (**File > Generate M-file**) if your filter was designed or edited with the Pole/Zero Editor.

---

The following is a sample generated M-file of the default FDATool filter.

```
function Hd = untitled
%UNTITLED Returns a discrete-time filter object
%
% M-file generated by MATLAB(R) 6.5 and the Signal Processing
% Toolbox 6.0.
%
% Generated on: 24-Oct-2002 09:46:59
%
% Remez FIR Lowpass filter designed using the firpm function.
% All frequency values are in Hz.
Fs = 48000; % Sampling Frequency

Fpass = 9600;           % Passband Frequency
Fstop = 12000;         % Stopband Frequency
Dpass = 0.057501127785; % Passband Ripple
Dstop = 0.0001;        % Stopband Attenuation
dens = 16;             % Density Factor

% Calculate the order from the parameters using firpmord.
[N, Fo, Ao, W] = firpmord([Fpass, Fstop]/(Fs/2), [1 0], ...
[Dpass, Dstop]);

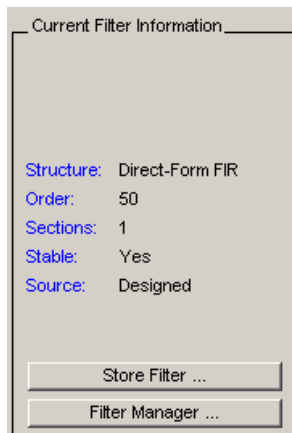
% Calculate the coefficients using the firpm function.
b = firpm(N, Fo, Ao, W, {dens});
Hd = dfilt.dffir(b);

% [EOF]
```

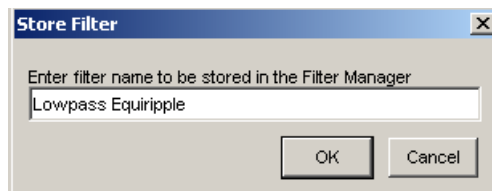
## Managing Filters in the Current Session

You can store filters designed in the current FDATool session for cascading together, exporting to FVTool or for recalling later in the same or future FDATool sessions.

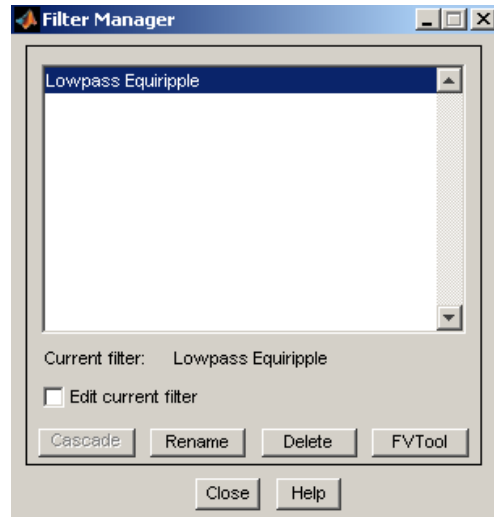
You store and access saved filters with the **Store filter** and **Filter Manager** buttons, respectively, in the Current Filter Information pane.



**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 listbox. To change the current filter, highlight the desired filter. If you select **Edit current filter**, FVTool 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.

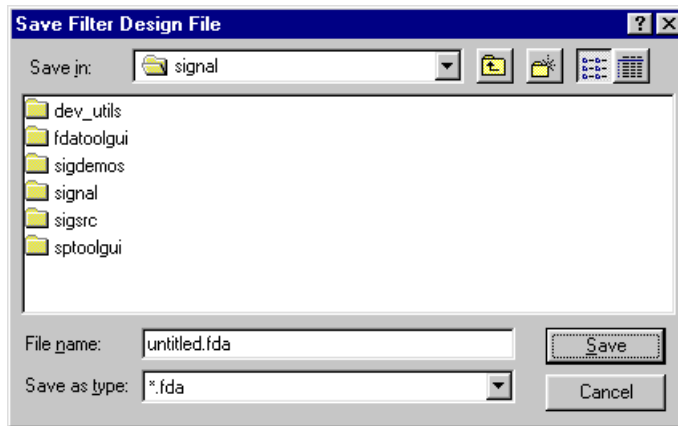
To remove a stored filter from the Filter Manager, press **Delete**.

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 File** browser opens, prompting you for a session name.




For example, save this design session as `TestFilter.fda` in your current working directory by typing `TestFilter` in the **File name** field.

The `.fda` extension is added automatically to all filter design sessions you save.

---

**Note** You can also use the **Save session** and **Save session as** menu items in the **File** menu to save a session. This dialog opens every time you select the **Save As** menu item.

---

You can load existing sessions into the Filter Design and Analysis Tool by selecting the **Open session** button, , or **Open Session** from the **File** menu. A **Load Filter Design File** browser opens that allows you to select from your previously saved filter design sessions.



# SPTool: A Signal Processing GUI Suite

---

The following chapter describes the Signal Processing Tool (SPTool) and provides a detailed example showing how to use this Graphical User Interface.

SPTool: An Interactive Signal Processing Environment (p. 6-2)	Overview of the tool
Opening SPTool (p. 6-4)	How to start the tool
Getting Context-Sensitive Help (p. 6-6)	How to get help
Signal Browser (p. 6-7)	Viewing signals
FDATool (p. 6-9)	Designing filters
Filter Visualization Tool (p. 6-10)	Viewing and analyzing filters
Spectrum Viewer (p. 6-12)	Viewing spectra
Filtering and Analysis of Noise (p. 6-15)	Full example using the tool
Exporting Signals, Filters, and Spectra (p. 6-27)	Sending data out of the tool
Accessing Filter Parameters (p. 6-29)	Using MATLAB to access saved filter parameters
Importing Filters and Spectra (p. 6-31)	Bringing data into the tool
Loading Variables from the Disk (p. 6-35)	Bringing data from a disk into the tool
Saving and Loading Sessions (p. 6-36)	Using sessions to save and recall work
Selecting Signals, Filters, and Spectra (p. 6-38)	Selecting data
Editing Signals, Filters, or Spectra (p. 6-39)	Editing data
Making Signal Measurements with Markers (p. 6-40)	Measuring signals
Setting Preferences (p. 6-42)	Customizing the tool
Using the Filter Designer (p. 6-46)	Information on previous Filter Designer

## **SPTool: An Interactive Signal Processing Environment**

SPTool is an interactive GUI for digital signal processing that can be 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 is for analyzing signals. You can also play portions of signals using your computer's audio hardware.
- The Filter Design and Analysis Tool (FDATool) is available for designing or editing FIR and IIR digital filters. Most of the Signal Processing Toolbox filter design methods available at the command line are also available in FDATool. Additionally, you can use FDATool to design a filter by using the Pole/Zero Editor to graphically place poles and zeros on the  $z$ -plane.
- The Filter Visualization Tool (FVTool) is for analyzing filter characteristics. See "Filter Visualization Tool" on page 6-10.
- The Spectrum Viewer is for spectral analysis. You can use the Signal Processing Toolbox spectral estimation methods to estimate the power spectral density of a signal. See "Spectrum Viewer" on page 6-12.

## SPTool Data Structures

You can use SPTool to analyze signals, filters, or spectra that you create at the MATLAB command line.

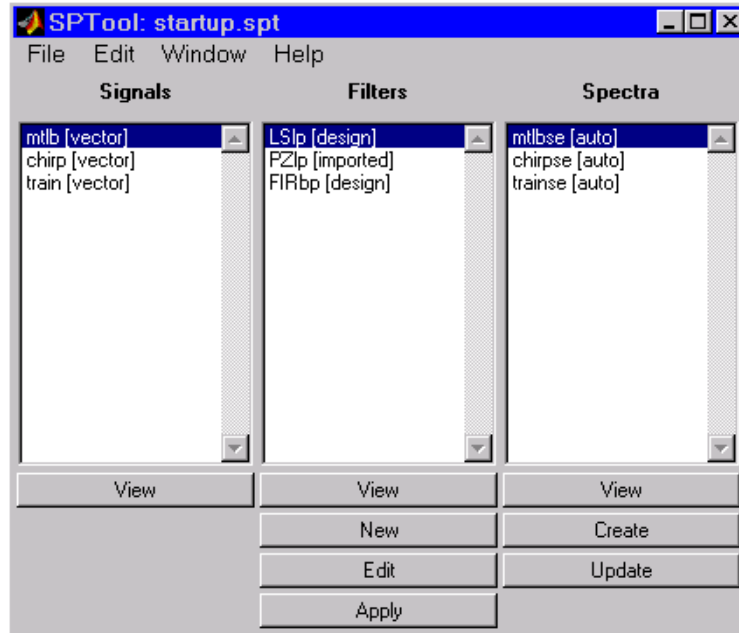
You can bring signals, filters, or spectra from the MATLAB workspace into the SPTool workspace using the **Import** item under the **File** menu. Signals, filters, or spectra that you create in (or import into) the SPTool workspace exist as MATLAB structures. See the MATLAB documentation for more information on MATLAB structures.

When you use the **Export** item under the **File** menu to save signals, filters, and spectra that you create or modify in SPTool, these are also saved as MATLAB structures.

## 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 6-42.

You can access these three GUIs from SPTool by selecting a signal, filter, or spectrum and clicking the appropriate **View** button:

- Signal Browser
- Filter Visualization Tool
- Spectrum Viewer

You can access FDATool 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 region of the Signal Browser, Filter Designer, or Spectrum Viewer:

- 1 Click **What's this?** .
- 2 Click on the region of the GUI you want information on.

You can also use the **What's this?** menu item in the **Help** menu to launch context-sensitive help.

## Signal Browser

You can use the Signal Browser to display and analyze signals listed in the **Signals** list box in SPTool.

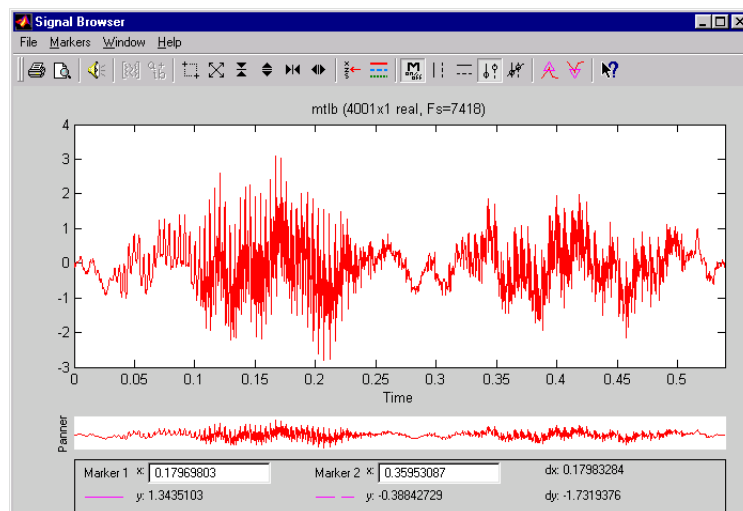
Using the Signal Browser you can

- 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, including markers for measuring, comparing, or playing signals
- A “panner” that displays the entire signal length, highlighting the portion currently active in the display region
- A marker measurements area
- A toolbar with buttons for convenient access to frequently used functions

Icon	Description
	Print and print preview
	Play an audio signal
	Display array and complex signals
	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 (See “Making Signal Measurements with Markers” on page 6-40)
	Turn on the What’s This help

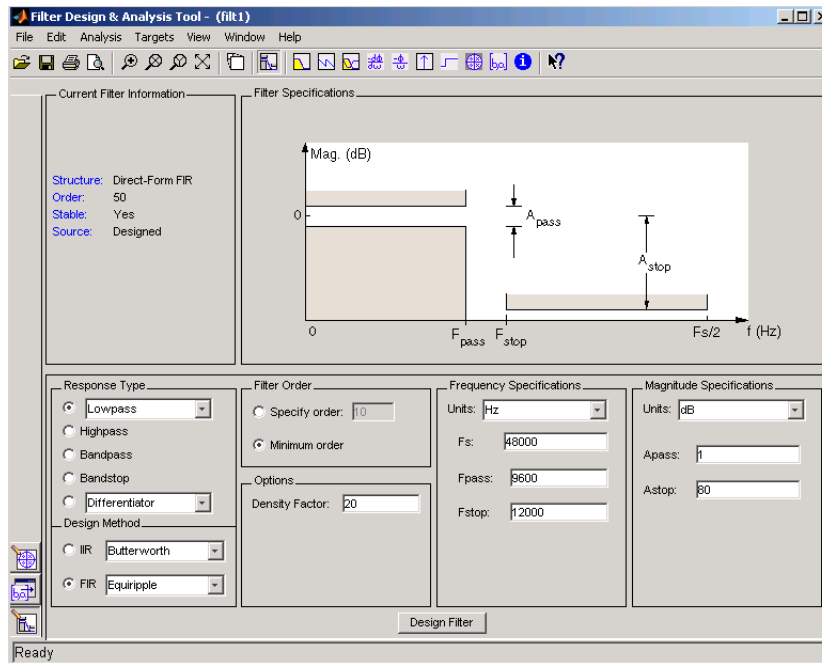


# FDATool

You can use a reduced version of the Filter Design and Analysis Tool (fdatool) to design and edit filters.

## Opening FDATool

To open FDATool from SPTool, click **New** under the **Filters** list to create a new filter or select one of the filters in the **Filters** list in SPTool and click **Edit** to edit that filter.



For details about FDATool, see “FDATool: A Filter Design and Analysis GUI” in Chapter 5.

**Note** When you open FDATool from SPTool, a reduced version of FDATool that is compatible with SPTool opens.

## Filter Visualization Tool

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 stand-alone 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.

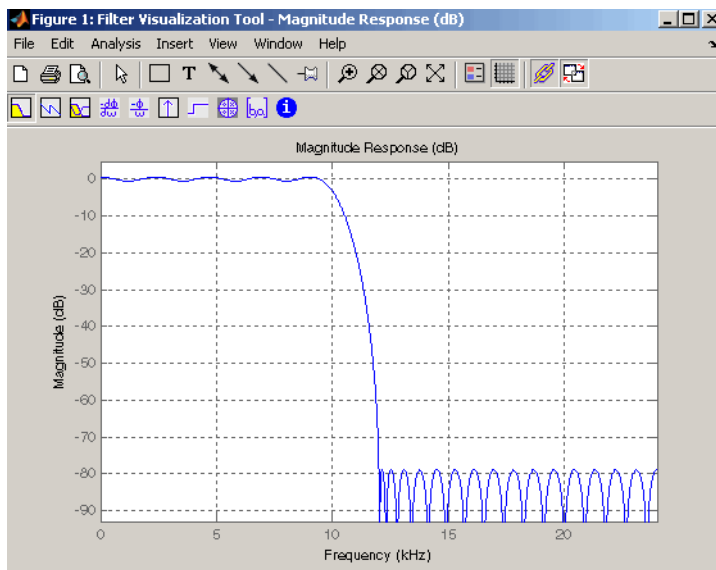
---

### 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 FDATool 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 **Sampling Frequency** in the SPTool **Edit** menu or the SPTool **Filters Edit** button.

## Spectrum Viewer

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:

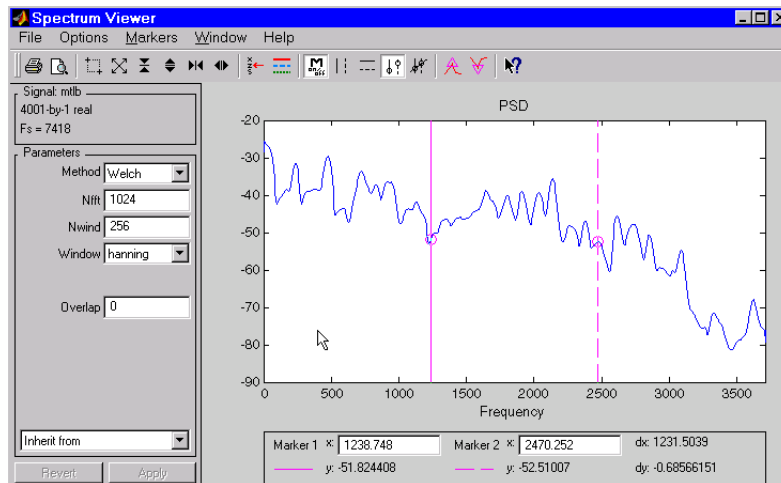
- 1** Select a signal from the **Signal** list box in SPTool.
- 2** Click **Create** in the **Spectra** list.
- 3** Click **Apply** in the Spectrum Viewer.

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 mt1b 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

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** Creating a noisy signal in the MATLAB workspace and importing it into SPTool
- 2** Designing a bandpass filter using FDATool
- 3** Applying the filter to the original noise signal to create a bandlimited noise signal
- 4** Comparing the time domain information of the original and filtered signals using the Signal Browser
- 5** Comparing the spectra of both signals using the Spectrum Viewer

### Step 1: 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

```
randn('state',0);  
x = randn(5000,1);
```

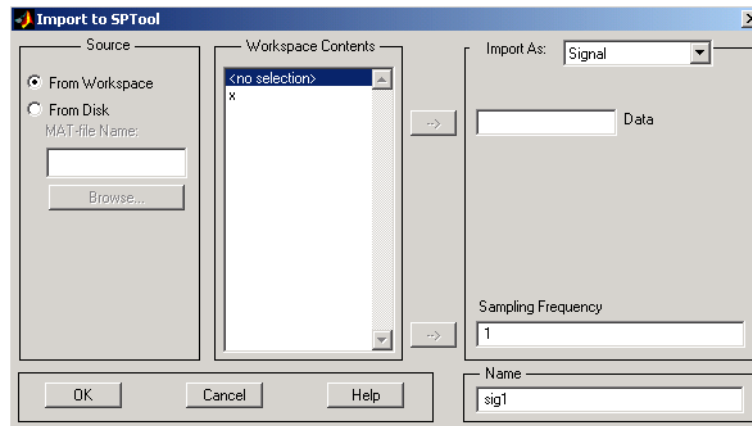
- 2 If SPTool is not already open, open SPTool by typing

```
sptool
```

The SPTool window is displayed.

- 3 Select **Import** from the **File** menu in SPTool. 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 and spectra into SPTool in much the same way as you import signals. See “Importing Filters and Spectra” on page 6-31 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 6-35 for more information.

---

Type `help sptool` for information about importing from the command line.

---

### Step 2: Designing a Filter

You can import an existing filter into SPTool, or you can design and edit a new filter using FDATool.

In this example, you

- 1 Open a default filter in FDATool.
- 2 Specify an equiripple bandpass FIR filter.

#### Opening FDATool

To open FDATool, click **New** in SPTool. FDATool 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 FDATool 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.

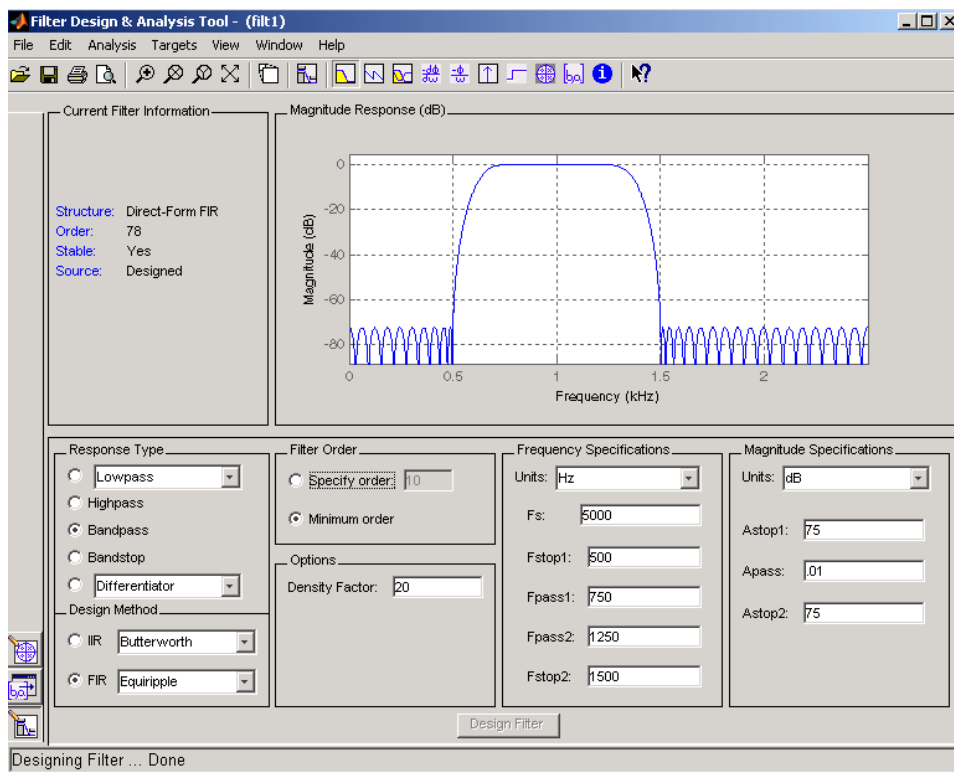
- 4** Under **Frequency Specifications**, set the sampling frequency (**Fs**) and the passband (**Fpass1**, **Fpass2**) and stopband (**Fstop1**, **Fstop2**) edges:

<b>Units</b>	Hz
<b>Fs</b>	5000
<b>Fstop1</b>	500
<b>Fpass1</b>	750
<b>Fpass2</b>	1250
<b>Fstop2</b>	1500

- 5** Under **Magnitude Specifications**, set the stopband attenuation (**Astop1**, **Astop2**) and the maximum passband ripple (**Apass**):

<b>Units</b>	dB
<b>Astop1</b>	75
<b>Apass</b>	0.01
<b>Astop2</b>	75

- 6** Click **Design Filter** to design the new filter. When the new filter is designed, the magnitude response of the filter is displayed.



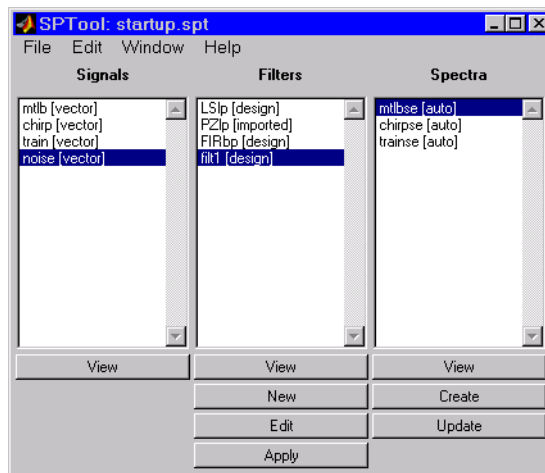
The resulting filter is an order-78 bandpass equiripple filter.

### Step 3: Applying a Filter to a Signal

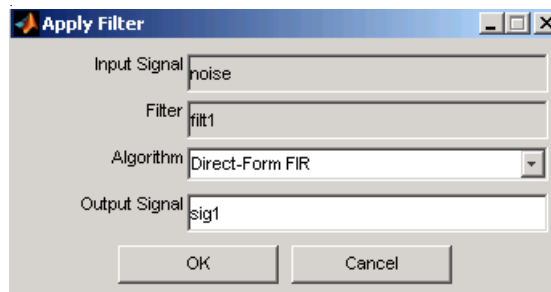
When you apply a filter to a signal, you create a new signal in SPTool representing the filtered signal.

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 FDATool **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.

## Step 4: 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 to the filtered signal `blnoise`:

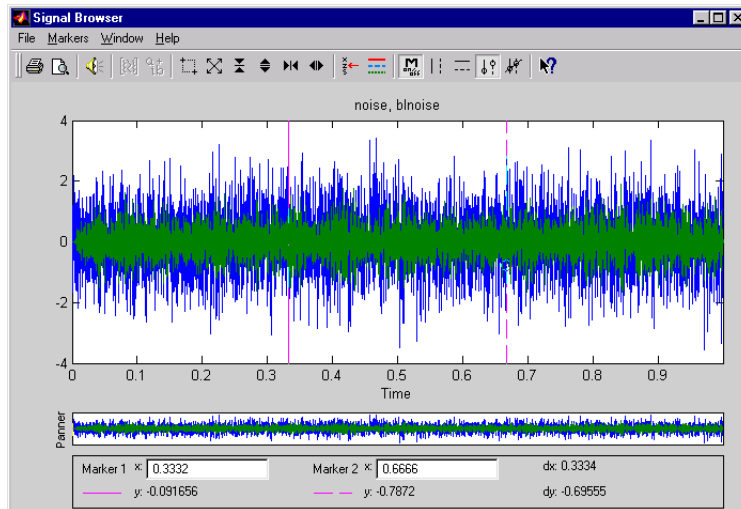
**1** **Shift**+click on the noise and `blnoise` signals in the **Signals** list of SPTool to select both signals.

**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 to 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 as in step 3 above. 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 to gray and the line color for blnoise 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.

### Step 5: 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, `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.
- 3 Print both spectra.

### Creating a PSD Object From a Signal

- 1 Click on SPTool, or select **SPTool** from the **Window** menu of any active open GUI. SPTool is now the active window.
- 2 Select the `noise[vector]` 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.

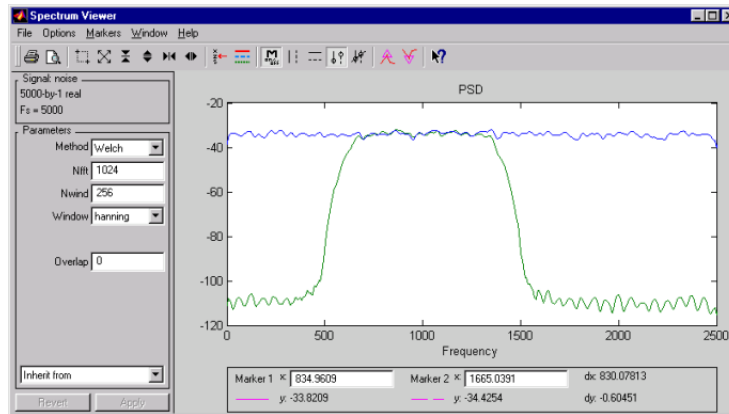
The PSD estimate spect1 is within 2 or 3 dB of 0, so the noise has a fairly "flat" power spectral density.

**5** Follow steps 1 through 4 for the bandlimited noise signal blnoise 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

You can export SPTool signals, filters, and spectra as structures to the MATLAB workspace or to your disk.

In each case, you

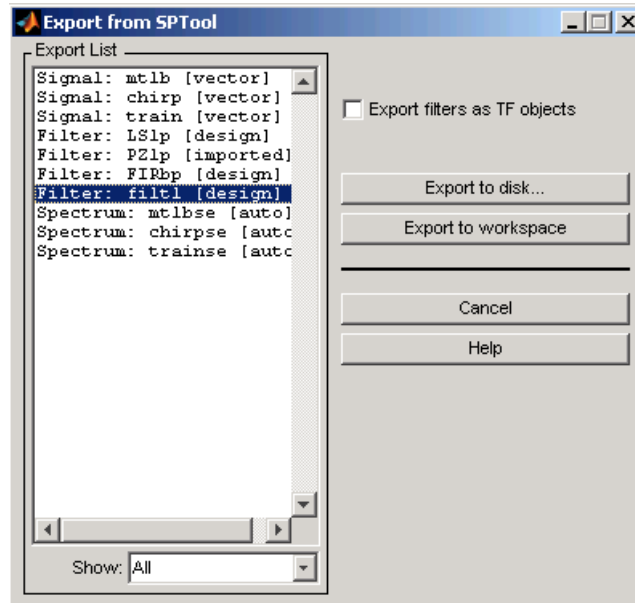
- 1 Select the items in SPTool you want to export.
- 2 Select **Export** from the **File** menu.

### 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 **Export** from the **File** menu.

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

You can access filter parameters in the following two ways.

- “Accessing Filter Parameters in a Saved Filter” on page 6-29
- “Accessing Parameters in a Saved Spectrum” on page 6-30

### 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 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

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(nb + 1)z^{-m}}{a(1) + a(2)z^{-1} + \dots + a(na + 1)z^{-n}}$$

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 `FDATool` and should not be changed.

---

## Accessing Parameters in a Saved Spectrum

The following structure fields describe the spectra saved by SPTool.

Field	Description
<code>P</code>	The spectral power vector.
<code>f</code>	The spectral frequency vector.
<code>confid</code>	A structure containing the confidence intervals data <ul style="list-style-type: none"> <li>• The <code>confid.level</code> field contains the chosen confidence level.</li> <li>• The <code>confid.Pc</code> field contains the spectral power data for the confidence intervals.</li> <li>• The <code>confid.enable</code> field contains a 1 if confidence levels are enabled for the power spectral density.</li> </ul>
<code>signalLabel</code>	The name of the signal from which the power spectral density was generated.
<code>Fs</code>	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

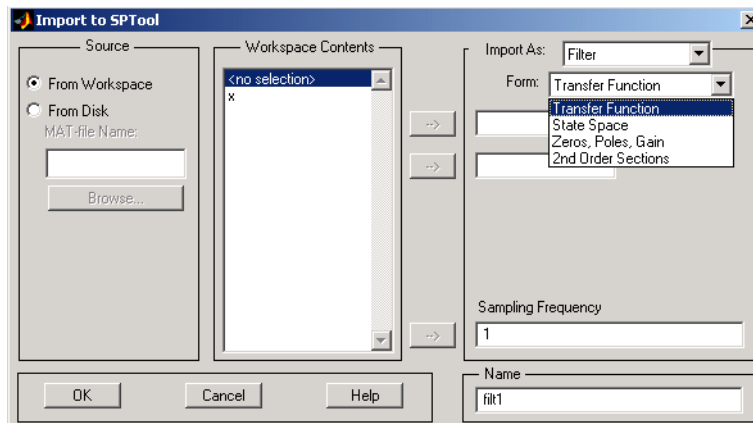
In addition to importing signals into SPTool, you can import filters or spectra into SPTool from either the workspace or from a file.

The procedures are very similar to those explained in

- “Step 1: Importing a Signal into SPTool” on page 6-16 for loading variables from the workspace
- “Loading Variables from the Disk” on page 6-35 for loading variables from your disk

### Importing Filters

When you import filters, first select the appropriate filter form from the **Form** list.



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:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(m+1)z^{-m}}{a(1) + a(2)z^{-1} + \dots + a(n+1)z^{-n}}$$

- 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:

$$H(z) = \frac{Z(z)}{P(z)} = k \frac{(z - z(1))(z - z(2)) \dots (z - z(m))}{(z - p(1))(z - p(2)) \dots (z - p(n))}$$

- 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

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

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.

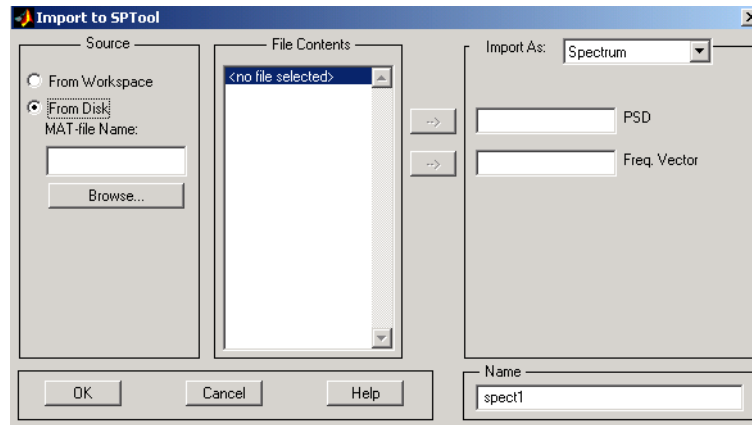
---

## 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 the **Select File to Open** dialog. 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

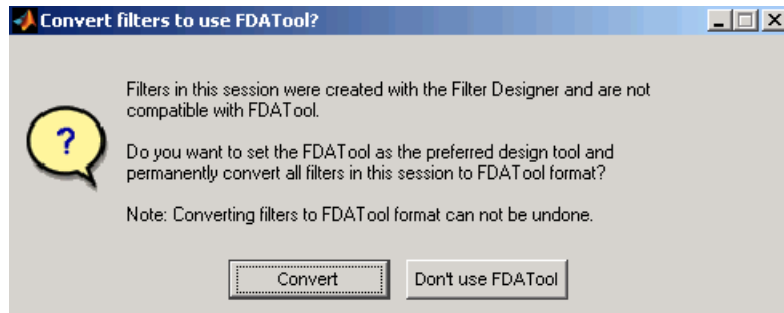
When you start SPTool, the default startup.spt session is loaded. To save your work in the startup SPTool session, use **File > Save Session** or to specify a session name, use **File > Save Session As**.

To recall a previously saved session, use **File > Open Session**.

### Filter Formats

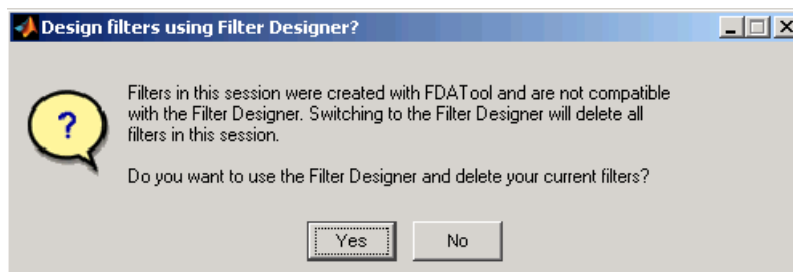
When you start SPTool or open a session, the current filter design format preference is compared to the filter formats in the session. See “Setting Preferences” on page 6-42.

- If the formats match, the session opens.
- If the filter preference is FDATool, but the session contains Filter Designer filters, this warning displays:



Click **Convert** to convert the filters to FDATool format. Click **Don't Use FDATool** 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 FDATool filters, this warning displays:



Click **Yes** to remove the current filters. Click **No** to leave the filters in FDATool and change the preference to **Use FDATool**.

## 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 on 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 sampling frequency associated with a signal's spectrum using the right-click menu on the Spectrum Viewer display region, the sampling frequency of the associated signal is automatically updated.

---








## Making Signal Measurements with Markers

You can use the markers on the Signal Browser or the Spectrum Viewer 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)
	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

Use **Preferences** from the SPTool **File** menu 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 MATLAB.

In the **Preferences** regions, you can:

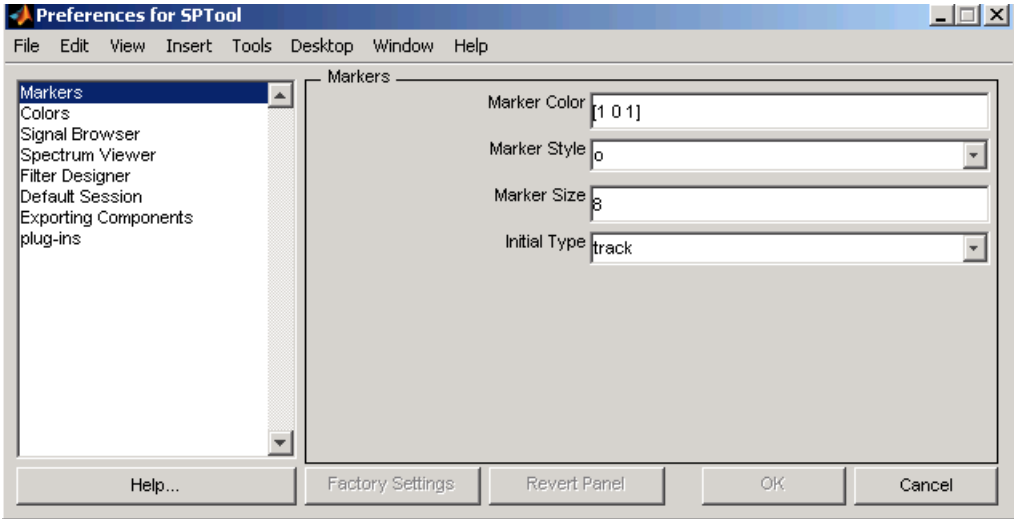
- Select colors and markers for all displays.
- Select colors and line styles for displayed signals.
- Configure labels, and enable/disable markers, panner, and zoom in the Signal Browser.
- Configure display parameters, and enable/disable markers and zoom in the Spectrum Viewer.
- Select whether to use the default FDATool or the Filter Designer to design filters. FDATool is the recommended designer.
- Enable/disable use of a default session file.
- Export filters for use with the Control System Toolbox.
- Enable/disable search for plug-ins at start-up.

---

**Note** You can set MATLAB preferences that affect the Filter Visualization Tool only from within FVTool by selecting **Preferences** from the **File** menu. You can set FVTool-specific preferences using the **Analysis Parameters** on the **Analysis** menu.

---

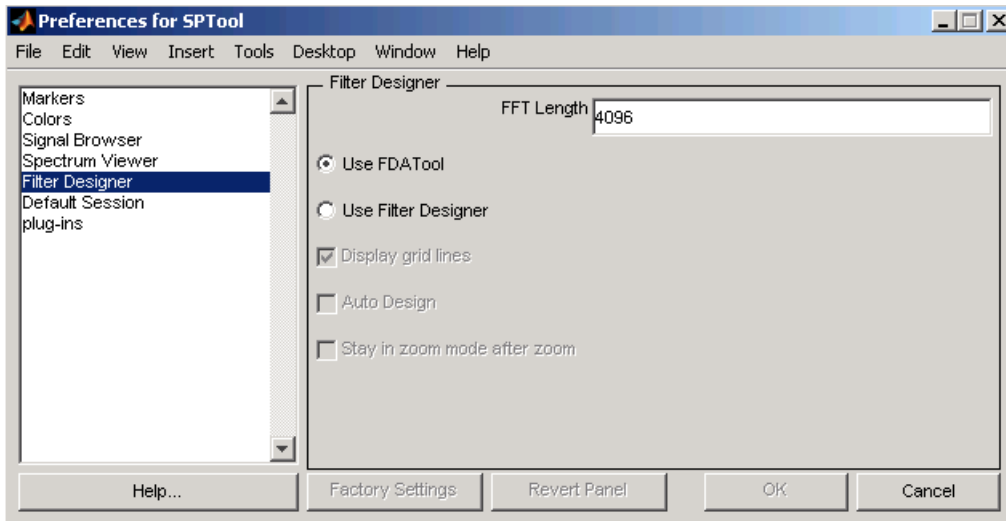
When you first select **Preferences**, the **Preferences** dialog box opens with **Markers** selected by default.



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.

## Setting the Filter Design Tool

The Filter Designer options include radio buttons to select the filter design tool.



FDATool is the default and recommended tool. You cannot change this preference if either FDATool or the Filter Designer is open.

---

**Note** Filters in any one SPTool session must be in the same format — either FDATool format or Filter Designer format. You can convert filters from the Filter Designer format to FDATool format, but you cannot convert FDATool filters to Filter Designer format.

---

When you change the preference from **Use FDATool** to **Use Filter Designer**, a warning message appears indicating that switching will delete current filters. See “Saving and Loading Sessions” on page 6-36 for more information.

When you change the preference from **Use Filter Designer** to **Use FDATool**, a confirmation message appears indicating that switching will convert your filters to FDATool format. See “Saving and Loading Sessions” on page 6-36 for information on this message.

Changes to Filter Designer format are saved only if you save the session.

## Using the Filter Designer

Although FDATool (described in “FDATool” on page 6-9) is the recommended filter design tool, the following information is provided for users of the Filter Designer, which provides an interactive graphical environment for the design of digital IIR and FIR filters based on specifications that you enter on a magnitude or pole-zero plot.

### Filter Types

You can design filters of the following types using the Filter Designer:

- Bandpass
- Lowpass
- Bandstop
- Highpass

### FIR Filter Methods

You can use the following filter methods to design FIR filters:

- Equiripple
- Least squares
- Window

### IIR Filter Methods

You can use the following filter methods to design IIR filters:

- Butterworth
- Chebyshev Type I
- Chebyshev Type II
- Elliptic

### Pole/Zero Editor

You can use the Pole/Zero Editor to design arbitrary FIR and IIR filters by placing and moving poles and zeros on the complex  $z$ -plane.

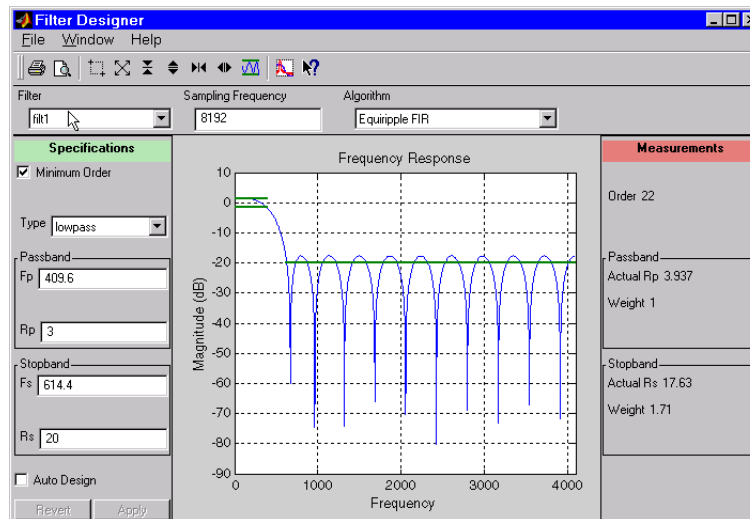
## Spectral Overlay Feature

You can also superimpose spectra on a filter's magnitude response to see if the filtering requirements are met.

## Opening the Filter Designer






Open the Filter Designer from SPTool by either:

- Clicking **New** in the **Filters** list in SPTool
- Selecting a filter you want to edit from the **Filters** list in SPTool, and then clicking **Edit**



The Filter Designer has the following components:

- A pull-down **Filter** menu for selecting a filter from the list in SPTool
- A **Sampling Frequency** text box
- A pull-down **Algorithm** menu for selecting a filter design method or a pole-zero plot display
- A **Specifications** area for viewing or modifying a filter's design parameters or pole-zero locations
- A plot display region for graphically adjusting filter magnitude responses or the pole-zero locations
- A **Measurements** area for viewing the response characteristics and stability of the current filter
- A toolbar with the following buttons

Icon	Description
	Print and print preview
	Zoom in and out
	Passband view
	Overlay spectrum
	Turn on the What's This help



## 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 a filter `filt1` to the MATLAB workspace, type

```
filt1
```

to display the fields of the MATLAB filter structure. The `tf`, `Fs`, and `specs` fields of the structure contain the 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

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(nb + 1)z^{-m}}{a(1) + a(2)z^{-1} + \dots + a(na + 1)z^{-n}}$$

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.

### The `Fs` Field: Accessing Filter Sample Frequency

The `Fs` field contains the sampling frequency of the filter in hertz.

### The specs Field: Accessing other Filter Parameters

The specs field is a structure containing parameters that you specified for the filter design. The first field, specs.currentModule, contains a string representing the most recent design method selected from the Filter Designer's **Algorithm** list before you exported the filter. The possible contents of the currentModule field and the corresponding design methods are shown below.

Contents of the currentModule field	Design Method
fdbutter	Butterworth IIR
fdcheby1	Chebyshev Type I IIR
fdcheby2	Chebyshev Type II IIR
fdellip	Elliptic IIR
fdfirls	Least Squares FIR
fdkaiser	Kaiser Window FIR
fdremez	Equiripple FIR

Following the specs.currentModule field, there may be up to seven additional fields, with labels such as specs.fdremez, specs.fdfirls, etc. The design specifications for the most recently exported filter are contained in the field whose label matches the currentModule string. For example, if the specs structure is

```
filt1.specs

ans
  currentModule: 'fdremez'
  fdremez: [1x1 struct]
```

the filter specifications are contained in the fdremez field, which is itself a data structure.

The specifications include the parameter values from the **Specifications** region of the Filter Designer, such as band edges and filter order. For example, the filter above has the following specifications stored in

`filt1.specs.fdremez`:

```
    filt1.specs.fdremez
```

```
ans =
```

```
    setOrderFlag: 0
```

```
        type: 3
```

```
            f: [0 0.2000 0.3000 0.5000 0.6000 1]
```

```
            m: [6x1 double]
```

```
            Rp: 0.0100
```

```
            Rs: 75
```

```
            wt: [3.2371 1 3.2371]
```

```
        order: 78
```

Because certain filter parameters are unique to a particular design, this structure has a different set of fields for each filter design.

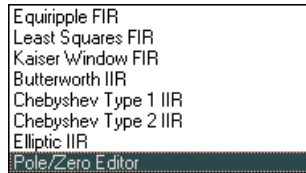
The table below describes the possible fields associated with the filter design specification field (the specs field) that can appear in the exported structure.

Parameter	Description
Beta	Kaiser window $\beta$ parameter.
f	Contains a vector of band-edge frequencies, normalized so that 1 Hz corresponds to half the sample frequency.
Fpass	Passband cutoff frequencies. Scalar for lowpass and highpass designs, two-element vector for bandpass and bandstop designs.
Fstop	Stopband cutoff frequencies. Scalar for lowpass and highpass designs, two-element vector for bandpass and bandstop designs.
m	The response magnitudes corresponding to the band-edge frequencies in f.
order	Filter order.
Rp	Passband ripple (dB)
Rs	Stopband attenuation (dB)
setOrderFlag	Contains 1 if the filter order was specified manually (i.e., the <b>Minimum Order</b> box in the <b>Specifications</b> region was not selected). Contains 0 if the filter order was computed automatically.
type	Contains 1 for lowpass, 2 for highpass, 3 for bandpass, or 4 for bandstop.
w3db	-3 dB frequency for Butterworth IIR designs.
wind	Vector of Kaiser window coefficients.
Wn	Cutoff frequency for the Kaiser window FIR filter when setOrderFlag = 1.
wt	Vector of weights, one weight per frequency band.

## Designing a Filter with the Pole/Zero Editor

To design a filter transfer function using the Filter Designer Pole/Zero Editor:

- 1 Select the Pole/Zero Editor option from the **Algorithm** list to open the Pole/Zero Editor in the Filter Designer display.



- 2 Enter the desired filter gain in the **Gain** edit box.
- 3 Select a pole or zero (or conjugate pair) by selecting one of the **x** (pole) or **o** (zero) symbols on the plot.
- 4 Choose the coordinates to work in by specifying Polar or Rectangular from the **Coordinates** list.
- 5 Specify the new location(s) of the selected pole, zero, or conjugate pair by typing values into the **Mag** and **Angle** fields (for angular coordinates) or **X** and **Y** (for rectangular coordinates) fields. Alternatively, position the poles and zeros by dragging the **x** and **o** symbols.
- 6 Use the **Conjugate pair** check box to create a conjugate pair from a lone pole or zero, or to break a conjugate pair into two individual poles or zeros.

Design a new filter or edit an existing filter in the same way.





---

**Note** Keep the Filter Visualization Tool (FVTool) open while designing a filter with the Pole/Zero Editor. Any changes that you make to the filter transfer function in the Pole/Zero Editor are then simultaneously reflected in the response plots of FVTool.

---

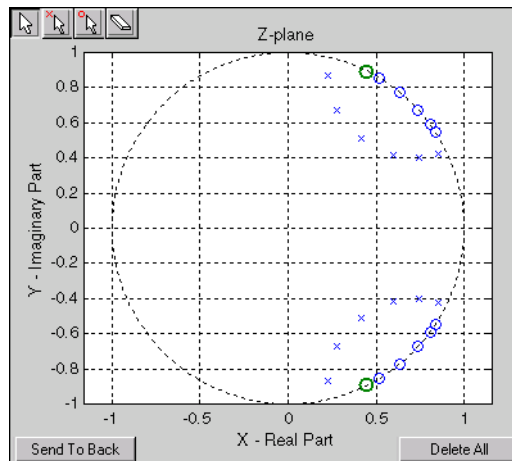
## Positioning Poles and Zeros


You can use your mouse to move poles and zeros around the pole/zero plot and modify your filter design.

Icon	Description
	Enable moving poles or zeros by dragging on the plot
	Add pole
	Add zero
	Erase poles or zeros

You can move both members of a conjugate pair simultaneously by manipulating just one of the poles or zeros.

To ungroup conjugates, select the desired pair and clear **Conjugate pair** in the **Specifications** region on the Filter Designer.



When you place two or more poles (or two or more zeros) directly on top of each other, a number is displayed next to the symbols (on the left for poles, and on the right for zeros) indicating the number of poles or zeros at that location (e.g., <sup>3</sup> for three zeros). This number makes it easy to keep track of all the poles and zeros in the plot area, even when several are superimposed on each other and are not visually differentiable. Note, however, that this number *does not* indicate the *multiplicity* of the poles or zeros to which it is attached.

To detect whether or not a set of poles or zeros are truly multiples, use the zoom tools to magnify the region around the poles or zeros in question. Because numerical limitations usually prevent any set of poles or zeros from sharing *exactly* the same value, at a high enough zoom level even truly multiple poles or zeros appear distinct from each other.

A common way to assess whether a particular group of poles or zeros contains multiples is by comparing the mutual proximity of the group members against a selected threshold value. As an example, the `residuez` function defines a pole or zero as being a multiple of another pole or zero if the absolute distance separating them is less than 0.1% of the larger pole or zero's magnitude.

## Redesigning a Filter Using the Magnitude Plot

After designing a filter in the Filter Designer, you can redesign it by dragging the specification lines on the magnitude plot. Use the specification lines to change passband ripple, stopband attenuation, and edge frequencies.

In the following example, create a Chebyshev filter and modify it by dragging the specification lines:

- 1 Select Chebyshev Type I IIR from the **Algorithm** menu.
- 2 Select highpass from the **Type** menu.
- 3 Type 2000 in the **Sampling Frequency** field.
- 4 Set the following parameters:
  - **F<sub>p</sub>** = 800
  - **F<sub>s</sub>** = 700
  - **R<sub>p</sub>** = 2.5
  - **R<sub>s</sub>** = 35

- 5 Select **Minimum Order** so the Filter Designer can calculate the lowest filter order that produces the desired characteristics.
- 6 Click **Apply** to compute the filter and update the response plot.
- 7 Position the cursor over the horizontal filter specification line for the stopband. This is the first (leftmost) horizontal specification line you see.  
  
The cursor changes to the up/down drag indicator.
- 8 Drag the line until the **Rs** (stopband attenuation) field reads 100.

---

**Note** The **Order** value in the **Measurements** region changes because a higher filter order is needed to meet the new specifications.

---



# Function Reference

---

Functions — By Category (p. 7-2)      Signal Toolbox functions arranged in categories  
Functions — Alphabetical List (p. 7-18)      Signal Toolbox functions arranged alphabetically

## Functions – By Category

“Digital Filters” on page 7-3	Digital filter design, simulation and analysis
“Analog Filters” on page 7-6	Analog filter design, frequency transformation, analysis, and discretization
“Linear Systems” on page 7-8	Conversion of linear system representations
“Windows” on page 7-10	Family of functions to window data
“Transforms” on page 7-12	CZT, FFT, DCT, Goertzel, Hilbert, etc.
“Cepstral Analysis” on page 7-12	Real, complex and inverse cepstrum
“Statistical Signal Processing” on page 7-12	Statistical signal processing and spectral analysis
“Parametric Modeling” on page 7-14	AR, ARMA, and frequency response fit modeling
“Linear Prediction” on page 7-14	Schur, Levinson, LPC, etc.
“Multirate Signal Processing” on page 7-16	Interpolation, decimation, and resampling
“Waveform Generation” on page 7-16	Pulses, periodic and aperiodic signals, vco, etc.
“Specialized Operations” on page 7-16	Plotting, vector manipulation, uniform encoding/decoding, etc.
“Graphical User Interfaces” on page 7-17	GUIs for data visualization, spectral analysis, filter design, and window design

## Digital Filters

“FIR Filter Design” on page 7-3	Design functions for FIR filters
“Communications Filters” on page 7-4	Design functions used in communications
“IIR Digital Filter Design” on page 7-4	Design functions for IIR filters
“IIR Filter Order Estimation” on page 7-4	Estimation functions for IIR filters
“Filter Analysis” on page 7-4	Functions for analyzing filters
“Filter Implementation” on page 7-6	Functions for implementing filters

## FIR Filter Design

<code>cfirpm</code>	Complex and nonlinear-phase equiripple FIR filter design
<code>fir1</code>	Window-based finite impulse response filter design
<code>fir2</code>	Frequency sampling-based finite impulse response filter design
<code>fircls</code>	Constrained least square FIR multiband filter design
<code>fircls1</code>	Constrained least square, lowpass and highpass, linear phase, FIR filter design
<code>firls</code>	Least square linear-phase FIR filter design
<code>firpm</code>	Parks-McClellan optimal FIR filter design
<code>firpmord</code>	Parks-McClellan optimal FIR filter order estimation
<code>intfilt</code>	Interpolation FIR filter design
<code>kaiserord</code>	Kaiser window FIR filter design estimation parameters
<code>sgolay</code>	Savitzky-Golay filter design

**Communications Filters**

<code>firrcos</code>	Raised cosine FIR filter design
<code>gaussfir</code>	Gaussian FIR pulse-shaping filter design

**IIR Digital Filter Design**

<code>butter</code>	Butterworth analog and digital filter design
<code>cheby1</code>	Chebyshev Type I filter design (passband ripple)
<code>cheby2</code>	Chebyshev Type II filter design (stopband ripple)
<code>ellip</code>	Elliptic (Cauer) filter design
<code>maxflat</code>	Generalized digital Butterworth filter design
<code>yulewalk</code>	Recursive digital filter design

**IIR Filter Order Estimation**

<code>buttord</code>	Butterworth filter order and cutoff frequency
<code>cheb1ord</code>	Chebyshev Type I filter order
<code>cheb2ord</code>	Chebyshev Type II filter order
<code>ellipord</code>	Minimum order for elliptic filters

**Filter Analysis**

<code>abs</code>	Absolute value (magnitude)
<code>angle</code>	Phase angle
<code>filternorm</code>	2-norm or infinity-norm of a digital filter
<code>freqz</code>	Frequency response of digital filter
<code>fvtool</code>	Filter Visualization Tool
<code>grpdelay</code>	Average filter delay (group delay)
<code>impz</code>	Impulse response of digital filter
<code>phasedelay</code>	Phase delay response of digital filter

phasez	Phase response of digital filter
stepz	Step response of digital filter
unwrap	Unwrap phase angles
zerophase	Zero-phase reponse of digital filter
zplane	Zero-pole plot

## Filter Implementation

<code>conv</code>	Convolution and polynomial multiplication
<code>conv2</code>	2-D convolution
<code>convmtx</code>	Convolution matrix
<code>deconv</code>	Deconvolution and polynomial division
<code>fftfilt</code>	FFT-based FIR filtering using overlap-add method
<code>filter</code>	Filter data with recursive (IIR) or nonrecursive (FIR) filter
<code>filter2</code>	2-D digital filtering
<code>filtfilt</code>	Zero-phase digital filtering
<code>filtic</code>	Initial conditions for transposed direct-form II filter implementation
<code>latcfilt</code>	Lattice and lattice-ladder filter implementation
<code>medfilt1</code>	1-D median filtering
<code>sgolayfilt</code>	Savitzky-Golay filtering
<code>sosfilt</code>	Second-order (biquadratic) IIR digital filtering
<code>upfirdn</code>	Upsample, apply FIR filter, and downsample

## Analog Filters

“Analog lowpass filter prototypes” on page 7-7	Prototyping functions for analog lowpass filters
“Analog filter design” on page 7-7	Design functions for analog filters
“Filter analysis” on page 7-7	Analysis functions for analog filters
“Analog filter transformation” on page 7-7	Transformation functions for analog filters
“Filter discretization” on page 7-8	Discretization functions for analog filters

### **Analog lowpass filter prototypes**

besselap	Bessel analog lowpass filter prototype
buttap	Butterworth analog lowpass filter prototype
cheb1ap	Chebyshev Type I analog lowpass filter prototype
cheb2ap	Chebyshev Type II analog lowpass filter prototype
ellipap	Elliptic analog lowpass filter prototype

### **Analog filter design**

besself	Bessel analog filter design
butter	Butterworth analog and digital filter design
cheby1	Chebyshev Type I filter design (passband ripple)
cheby2	Chebyshev Type II filter design (stopband ripple)
ellip	Elliptic (Cauer) filter design

### **Filter analysis**

abs	Absolute value (magnitude)
angle	Phase angle
freqs	Frequency response of analog filters
freqspace	Frequency spacing for frequency response

### **Analog filter transformation**

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

**Filter discretization**

<code>bilinear</code>	Bilinear transformation method for analog-to-digital filter conversion
<code>impinvar</code>	Impulse invariance method for analog-to-digital filter conversion

**Linear Systems**

<code>latc2tf</code>	Convert lattice filter parameters to transfer function form
<code>polyscale</code>	Scale roots of polynomial
<code>polystab</code>	Stabilize polynomial
<code>residuez</code>	$z$ -transform partial-fraction expansion
<code>sos2ss</code>	Convert digital filter second-order section parameters to state-space form
<code>sos2tf</code>	Convert digital filter second-order section data to transfer function form
<code>sos2zp</code>	Convert digital filter second-order sections parameters to zero-pole-gain form
<code>ss2sos</code>	Convert digital filter state-space parameters to second-order sections form
<code>ss2tf</code>	Convert state-space filter parameters to transfer function form
<code>ss2zp</code>	Convert state-space filter parameters to zero-pole-gain form
<code>tf2latc</code>	Convert transfer function filter parameters to lattice filter form
<code>tf2sos</code>	Convert digital filter transfer function data to second-order sections form
<code>tf2ss</code>	Convert transfer function filter parameters to state-space form
<code>tf2zpk</code>	Convert discrete-time transfer function filter parameters to zero-pole-gain form



zp2sos	Convert digital filter zero-pole-gain parameters to second-order sections form
zp2ss	Convert zero-pole-gain filter parameters to state-space form
zp2tf	Convert zero-pole-gain filter parameters to transfer function form

## Windows

<code>barthannwin</code>	Modified Bartlett-Hann window
<code>bartlett</code>	Bartlett window
<code>blackman</code>	Blackman window
<code>blackmanharris</code>	Minimum 4-term Blackman-Harris window
<code>bohmanwin</code>	Bohman window
<code>chebwin</code>	Chebyshev window
<code>dpss</code>	Discrete prolate spheroidal sequences (Slepian sequences)
<code>dpsscLEAR</code>	Remove discrete prolate spheroidal sequences from database
<code>dpssdir</code>	Discrete prolate spheroidal sequences database directory
<code>dpssload</code>	Load discrete prolate spheroidal sequences from database
<code>dpsssave</code>	Save discrete prolate spheroidal sequences in database
<code>flattopwin</code>	Flat top weighted window
<code>gausswin</code>	Gaussian window
<code>hamming</code>	Hamming window
<code>hann</code>	Hann (Hanning) window
<code>kaiser</code>	Kaiser window
<code>nuttallwin</code>	Nuttall-defined minimum 4-term Blackman-Harris window
<code>parzenwin</code>	Parzen (de la Valle-Poisson) window
<code>rectwin</code>	Rectangular window
<code>taylorwin</code>	Taylor window
<code>triang</code>	Triangular window
<code>tukeywin</code>	Tukey (tapered cosine) window

window	Window function gateway
wvtool	Window Visualization Tool

## Transforms

<code>bitrevorder</code>	Permute input into bit-reversed order
<code>czt</code>	Chirp z-transform
<code>dct</code>	Discrete cosine transform (DCT)
<code>dftmtx</code>	Discrete Fourier transform matrix
<code>digitrevorder</code>	Permute input into digit-reversed order
<code>fft</code>	1-D fast Fourier transform
<code>fft2</code>	2-D fast Fourier transform
<code>fftshift</code>	Rearrange FFT function outputs
<code>goertzel</code>	Discrete Fourier transform using second-order Goertzel algorithm
<code>hilbert</code>	Discrete-time analytic signal using Hilbert transform
<code>idct</code>	Inverse discrete cosine transform
<code>ifft</code>	1-D inverse fast Fourier transform
<code>ifft2</code>	2-D inverse fast Fourier transform

## Cepstral Analysis

<code>ceps</code>	Complex cepstral analysis
<code>icceps</code>	Inverse complex cepstrum
<code>rceps</code>	Real cepstrum and minimum phase reconstruction

## Statistical Signal Processing

<code>corrcoef</code>	Correlation coefficient matrix
<code>corrmtx</code>	Data matrix for autocorrelation matrix estimation
<code>cov</code>	Covariance matrix
<code>cpsd</code>	Cross power spectral density
<code>mscohere</code>	Magnitude squared coherence

---

<code>pburg</code>	Power spectral density using Burg method
<code>pcov</code>	Power spectral density using covariance method
<code>peig</code>	Pseudospectrum using eigenvector method
<code>periodogram</code>	PSD using a periodogram
<code>pmcov</code>	PSD using modified covariance method
<code>pmtm</code>	PSD using multitaper method (MTM)
<code>pmusic</code>	PSD using MUSIC algorithm
<code>pwelch</code>	PSD using Welch's method
<code>pyulear</code>	PSD using Yule-Walker AR method
<code>rooteig</code>	Frequency and power content estimate using eigenvector method
<code>rootmusic</code>	Frequency and power content estimate using root MUSIC algorithm
<code>spectrogram</code>	Time-dependent frequency analysis using short-time Fourier transform (STFT)
<code>spectrum</code>	Spectral estimation
<code>tfestimate</code>	Transfer function estimate
<code>xcorr</code>	Cross-correlation
<code>xcorr2</code>	2-D cross-correlation
<code>xcov</code>	Cross-covariance

## Parametric Modeling

arburg	Estimate AR model parameters using Burg method
arcov	Estimate AR model parameters using covariance method
armcov	Estimate AR model parameters using modified covariance method
aryule	Estimate AR model parameters using Yule-Walker method
ident	See the System Identification Toolbox documentation
invfreqs	Identify continuous-time filter parameters from frequency response data
invfreqz	Identify discrete-time filter parameters from frequency response data
prony	Prony's method for time domain IIR filter design
stmcb	Compute linear model using Steiglitz-McBride iteration

## Linear Prediction

ac2poly	Convert autocorrelation sequence to prediction polynomial
ac2rc	Convert autocorrelation sequence to reflection coefficients
is2rc	Convert inverse sine parameters to reflection coefficients
lar2rc	Convert log area ratio parameters to reflection coefficients
levinson	Levinson-Durbin recursion
lpc	Linear prediction filter coefficients
lsf2poly	Convert line spectral frequencies to prediction filter coefficients
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

rc2ac	Convert reflection coefficients to autocorrelation sequence
rc2is	Convert reflection coefficients to inverse sine parameters
rc2lar	Convert reflection coefficients to log area ratio parameters
rc2poly	Convert reflection coefficients to prediction filter polynomial
rlevinson	Reverse Levinson-Durbin recursion
schurrc	Compute reflection coefficients from autocorrelation sequence

## Multirate Signal Processing

<code>decimate</code>	Decimation — decrease sampling rate
<code>downsample</code>	Decrease sampling rate by integer factor
<code>interp</code>	Interpolation — increase sampling rate by integer factor
<code>interp1</code>	General 1-D interpolation
<code>resample</code>	Change sampling rate by rational factor
<code>spline</code>	Cubic spline interpolation
<code>upfirdn</code>	Upsample, apply FIR filter, and downsample
<code>upsample</code>	Increase sampling rate by integer factor

## Waveform Generation

<code>chirp</code>	Swept-frequency cosine
<code>diric</code>	Dirichlet or periodic sinc function
<code>gauspuls</code>	Gaussian-modulated sinusoidal pulse
<code>gmonopuls</code>	Gaussian monopulse
<code>pulstran</code>	Pulse train
<code>rectpuls</code>	Sampled aperiodic rectangle
<code>sawtooth</code>	Sawtooth or triangle wave
<code>sinc</code>	Sinc
<code>square</code>	Square wave
<code>tripuls</code>	Sampled aperiodic triangle
<code>vco</code>	Voltage controlled oscillator

## Specialized Operations

<code>buffer</code>	Buffer signal vector into matrix of data frames
<code>cell2sos</code>	Convert second-order sections cell array to a matrix



---

<code>cplxpair</code>	Group complex numbers into complex conjugate pairs
<code>demod</code>	Demodulation for communications simulation
<code>eqtflength</code>	Equalize lengths of transfer function's numerator and denominator
<code>modulate</code>	Modulation for communications simulation
<code>seqperiod</code>	Compute period of sequence
<code>sos2cell</code>	Convert second-order sections matrix to cell array
<code>stem</code>	Plot discrete data sequence
<code>strips</code>	Strip plot
<code>udocode</code>	Decode $2^n$ -level quantized integer inputs to floating-point outputs
<code>uencode</code>	Quantize and encode floating-point inputs to integer outputs

## Graphical User Interfaces

<code>fdatool</code>	Open Filter Design and Analysis Tool
<code>fvtool</code>	Open Filter Visualization Tool
<code>sptool</code>	Open Interactive digital signal processing tool
<code>wintool</code>	Open Window Design and Analysis Tool
<code>wvtool</code>	Open Window Visualization Tool

## **Functions – Alphabetical List**

This section contains function reference pages listed alphabetically.

**Purpose** Absolute value (magnitude)

**Description** abs is a MATLAB function.

**Signal-Specific Example** Calculate the magnitude of the FFT of a sequence.

```
t = (0:99)/100; % Time vector
x = sin(2*pi*15*t) + sin(2*pi*40*t); % Signal
y = fft(x); % Compute DFT of x
m = abs(y); % Magnitude
```

Plot the magnitude:

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

# ac2poly

---

**Purpose** Convert autocorrelation sequence to prediction polynomial

**Syntax**  
`a = ac2poly(r)`  
`[a,efinal] = ac2poly(r)`

**Description** `a = ac2poly(r)` finds the linear prediction, FIR filter polynomial `a` corresponding to the autocorrelation sequence `r`. `a` is the same length as `r`, and `a(1) = 1`. The prediction filter polynomial represents the coefficients of the prediction filter whose output produces a signal whose autocorrelation sequence is approximately the same as the given autocorrelation sequence `r`.

`[a,efinal] = ac2poly(r)` returns the final prediction error `efinal`, determined by running the filter for `length(r)` steps.

**Remarks** You can apply this function to real or complex data.

**Examples** Consider the autocorrelation sequence:

```
r = [5.0000 -1.5450 -3.9547 3.9331 1.4681 -4.7500];
```

The corresponding prediction filter polynomial is

```
[a,efinal] = ac2poly(r)
```

```
a =  
    1.0000    0.6147    0.9898    0.0004    0.0034   -0.0077
```

```
efinal =  
    0.1791
```

**References** [1] Kay, S.M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

**See Also** `ac2rc`, `poly2ac`, `rc2poly`

---

<b>Purpose</b>	Convert autocorrelation sequence to reflection coefficients
<b>Syntax</b>	$[k, r0] = \text{ac2rc}(r)$
<b>Description</b>	$[k, r0] = \text{ac2rc}(r)$ finds the reflection coefficients $k$ corresponding to the autocorrelation sequence $r$ . $r0$ contains the initial zero-lag autocorrelation. These reflection coefficients can be used to specify the lattice prediction filter that produces a sequence with approximately the same autocorrelation sequence as the given sequence $r$ .
<b>Remarks</b>	You can apply this function to real or complex data.
<b>References</b>	[1] Kay, S.M. <i>Modern Spectral Estimation</i> . Englewood Cliffs, NJ: Prentice-Hall, 1988.
<b>See Also</b>	ac2poly, poly2rc, rc2ac

# angle

---

**Purpose** Phase angle

**Description** angle is a MATLAB function.

**Signal-specific Example** Calculate the phase of the FFT of a sequence.

```
t = (0:99)/100; % Time vector
x = sin(2*pi*15*t) + sin(2*pi*40*t); % Signal
y = fft(x); % Compute DFT of x
p = unwrap(angle(y)); % Phase
```

Plot the phase:

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

**Purpose** Estimate AR model parameters using Burg method

**Syntax**  
`a = arburg(x,p)`  
`[a,e] = arburg(x,p)`  
`[a,e,k] = arburg(x,p)`

**Description** `a = arburg(x,p)` uses the Burg method to fit a  $p$ th order autoregressive (AR) model to the input signal,  $x$ , by minimizing (least squares) the forward and backward prediction errors while constraining the AR parameters to satisfy the Levinson-Durbin recursion.  $x$  is assumed to be the output of an AR system driven by white noise. Vector  $a$  contains the normalized estimate of the AR system parameters,  $A(z)$ , in descending powers of  $z$ .

$$H(z) = \frac{\sqrt{e}}{A(z)} = \frac{\sqrt{e}}{1 + a_2z^{-1} + \dots + a_{(p+1)}z^{-p}}$$

Since the method characterizes the input data using an all-pole model, the correct choice of the model order  $p$  is important.

`[a,e] = arburg(x,p)` returns the variance estimate,  $e$ , of the white noise input to the AR model.

`[a,e,k] = arburg(x,p)` returns a vector,  $k$ , of reflection coefficients.

**See Also** `arcov`, `armcov`, `aryule`, `lpc`, `pburg`, `prony`

## arccov

---

**Purpose** Estimate AR model parameters using covariance method

**Syntax** `a = arccov(x,p)`  
`[a,e] = arccov(x,p)`

**Description** `a = arccov(x,p)` uses the covariance method to fit a  $p$ th order autoregressive (AR) model to the input signal,  $x$ , which is assumed to be the output of an AR system driven by white noise. This method minimizes the forward prediction error in the least-squares sense. Vector  $a$  contains the normalized estimate of the AR system parameters,  $A(z)$ , in descending powers of  $z$ .

$$H(z) = \frac{\sqrt{e}}{A(z)} = \frac{\sqrt{e}}{1 + a_2 z^{-1} + \dots + a_{(p+1)} z^{-p}}$$

Because the method characterizes the input data using an all-pole model, the correct choice of the model order  $p$  is important.

`[a,e] = arccov(x,p)` returns the variance estimate,  $e$ , of the white noise input to the AR model.

**See Also** `arburg`, `armcov`, `aryule`, `lpc`, `pcov`, `prony`



**Purpose** Estimate AR model parameters using modified covariance method

**Syntax**  
`a = armcov(x,p)`  
`[a,e] = armcov(x,p)`

**Description** `a = armcov(x,p)` uses the modified covariance method to fit a  $p$ th order autoregressive (AR) model to the input signal,  $x$ , which is assumed to be the output of an AR system driven by white noise. This method minimizes the forward and backward prediction errors in the least-squares sense. Vector  $a$  contains the normalized estimate of the AR system parameters,  $A(z)$ , in descending powers of  $z$ .

$$H(z) = \frac{\sqrt{e}}{A(z)} = \frac{\sqrt{e}}{1 + a_1 z^{-1} + \dots + a_p z^{-p}}$$

Because the method characterizes the input data using an all-pole model, the correct choice of the model order  $p$  is important.

`[a,e] = armcov(x,p)` returns the variance estimate,  $e$ , of the white noise input to the AR model.

**See Also** `arburg`, `arcov`, `aryule`, `lpc`, `pmcov`, `prony`

# aryule

---

**Purpose** Estimate AR model parameters using Yule-Walker method

**Syntax**

```
a = aryule(x,p)
[a,e] = aryule(x,p)
[a,e,k] = aryule(x,p)
```

**Description** `a = aryule(x,p)` uses the Yule-Walker method, also called the autocorrelation method, to fit a  $p$ th order autoregressive (AR) model to the windowed input signal,  $x$ , by minimizing the forward prediction error in the least-squares sense. This formulation leads to the Yule-Walker equations, which are solved by the Levinson-Durbin recursion.  $x$  is assumed to be the output of an AR system driven by white noise. Vector  $a$  contains the normalized estimate of the AR system parameters,  $A(z)$ , in descending powers of  $z$ .

$$H(z) = \frac{\sqrt{e}}{A(z)} = \frac{\sqrt{e}}{1 + a_2 z^{-1} + \dots + a_{(p+1)} z^{-p}}$$

Because the method characterizes the input data using an all-pole model, the correct choice of the model order  $p$  is important.

`[a,e] = aryule(x,p)` returns the variance estimate,  $e$ , of the white noise input to the AR model.

`[a,e,k] = aryule(x,p)` returns a vector,  $k$ , of reflection coefficients.

**See Also** `arburg`, `arcov`, `armcov`, `lpc`, `prony`, `pyulear`

**Purpose** Modified Bartlett-Hann window

**Syntax** `w = barthannwin(L)`

**Description** `w = barthannwin(L)` returns an L-point modified Bartlett-Hann window in the column vector `w`. Like Bartlett, Hann, and Hamming windows, this window has a mainlobe at the origin and asymptotically decaying sidelobes on both sides. It is a linear combination of weighted Bartlett and Hann windows with near sidelobes lower than both Bartlett and Hann and with far sidelobes lower than both Bartlett and Hamming windows. The mainlobe width of the modified Bartlett-Hann window is not increased relative to either Bartlett or Hann window mainlobes.

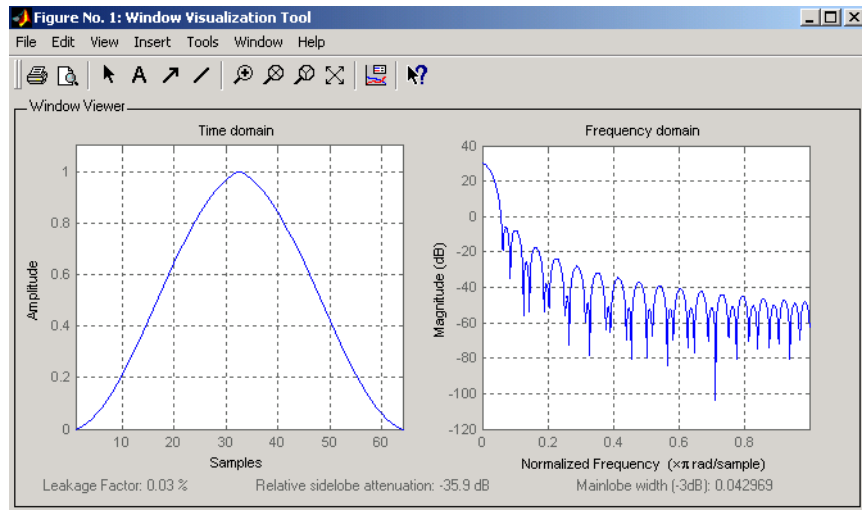
---

**Note** The Hann window is also called the Hanning window.

---

**Examples** Create a 64-point Bartlett-Hann window and display the result using WVTool:

```
L=64;
wvtool(barthannwin(L))
```



# barthannwin

---

## Algorithm

The equation for computing the coefficients of a Modified Bartlett-Hanning window is

$$w(n) = 0.62 - 0.48 \left| \left( \frac{n}{N} - 0.5 \right) \right| + 0.38 \cos \left( 2\pi \left( \frac{n}{N} - 0.5 \right) \right)$$

where  $0 \leq n \leq N$  and the window length  $L = N+1$ .

## References

Ha, Y.H., and J.A. Pearce. "A New Window and Comparison to Standard Windows." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 37, No. 2, (February 1999). pp. 298-301.

## See Also

bartlett, blackmanharris, bohmanwin, nuttallwin, parzenwin, rectwin, triang, window, wintool, wvtool

**Purpose** Bartlett window

**Syntax** `w = bartlett(L)`

**Description** `w = bartlett(L)` returns an L-point Bartlett window in the column vector `w`, where L must be a positive integer. The coefficients of a Bartlett window are computed as follows:

- For L odd

$$w(n) = \begin{cases} \frac{2n}{N}, & 0 \leq n \leq \frac{N}{2} \\ 2 - \frac{2n}{N}, & \frac{N}{2} \leq n \leq N \end{cases}$$

- For L even

$$w(n) = \begin{cases} \frac{2n}{N}, & 0 \leq n \leq \frac{L}{2} - 1 \\ 2 - \frac{2(N-n)}{N}, & \frac{L}{2} \leq n \leq N \end{cases}$$

The window length  $L = N+1$ .

The Bartlett window is very similar to a triangular window as returned by the `triang` function. The Bartlett window always ends with zeros at samples 1 and `n`, however, while the triangular window is nonzero at those points. For L odd, the center L-2 points of `bartlett(L)` are equivalent to `triang(L-2)`.

---

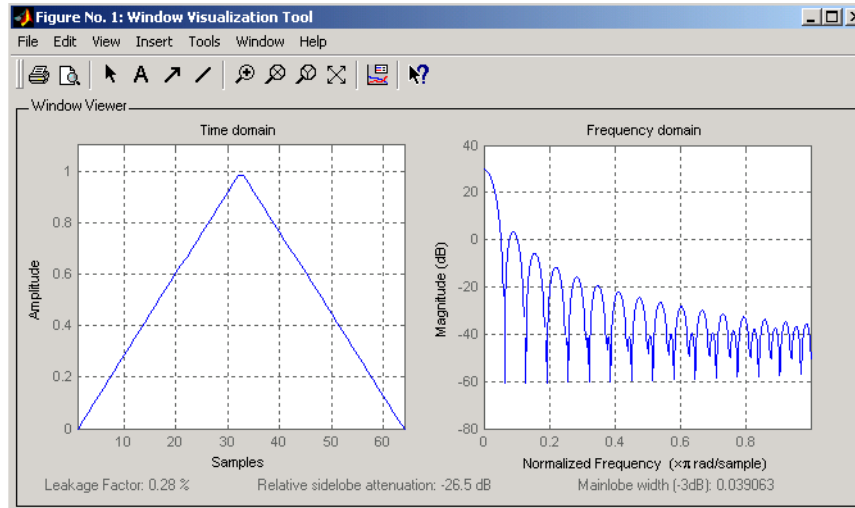
**Note** If you specify a one-point window (set `L=1`), the value 1 is returned.

---

## Examples

Create a 64-point Bartlett window and display the result using `WVTool`:

```
L=64;
wvtool(bartlett(L))
```



## References

[1] Oppenheim, A.V., and R.W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

## See Also

barthannwin, blackmanharris, bohmanwin, nuttallwin, parzenwin, rectwin, triang, window, wintool, wvtool

**Purpose** Bessel analog lowpass filter prototype

**Syntax** `[z,p,k] = besselap(n)`

**Description** `[z,p,k] = besselap(n)` returns the poles and gain of an order  $n$  Bessel analog lowpass filter prototype.  $n$  must be less than or equal to 25. The function returns the poles in the length  $n$  column vector  $p$  and the gain in scalar  $k$ .  $z$  is an empty matrix because there are no zeros. The transfer function is

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

`besselap` normalizes the poles and gain so that at low frequency and high frequency the Bessel prototype is asymptotically equivalent to the Butterworth prototype of the same order [1]. The magnitude of the filter is less than  $\sqrt{1/2}$  at the unity cutoff frequency  $\Omega_c = 1$ .

Analog Bessel filters are characterized by a group delay that is maximally flat at zero frequency and almost constant throughout the passband. The group delay at zero frequency is

$$\left( \frac{(2n)!}{2^n n!} \right)^{1/n}$$

**Algorithm** `besselap` finds the filter roots from a lookup table constructed using the Symbolic Math Toolbox.

**References** [1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pgs. 228-230.

**See Also** `besself`, `buttap`, `cheb1ap`, `cheb2ap`, `ellipap`

Also see the Symbolic Math Toolbox documentation.

# besself

---

**Purpose** Bessel analog filter design

**Syntax**  
`[b,a] = besself(n,Wo)`  
`[z,p,k] = besself(...)`  
`[A,B,C,D] = besself(...)`

**Description** `besself` designs lowpass, analog Bessel filters, which are characterized by almost constant group delay across the entire passband, thus preserving the wave shape of filtered signals in the passband. `besself` does not support the design of digital Bessel filters.

`[b,a] = besself(n,Wo)` designs an order  $n$  lowpass analog Bessel filter, where  $Wo$  is the frequency up to which the filter's group delay is approximately constant. Larger values of the filter order ( $n$ ) produce a group delay that better approximates a constant up to frequency  $Wo$ .

`besself` returns the filter coefficients in the length  $n+1$  row vectors  $b$  and  $a$ , with coefficients in descending powers of  $s$ , derived from this transfer function:

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \dots + b(n+1)}{s^n + a(2)s^{n-1} + \dots + a(n+1)}$$

`[z,p,k] = besself(...)` returns the zeros and poles in length  $n$  or  $2*n$  column vectors  $z$  and  $p$  and the gain in the scalar  $k$ .

`[A,B,C,D] = besself(...)` returns the filter design in state-space form, where  $A$ ,  $B$ ,  $C$ , and  $D$  are

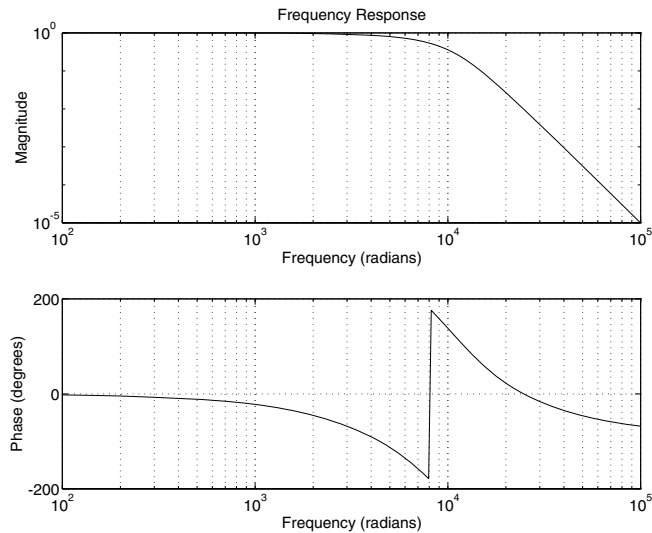
$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

and  $u$  is the input,  $x$  is the state vector, and  $y$  is the output.

**Examples** Design a fifth-order analog lowpass Bessel filter with an approximate constant group delay up to 10,000 rad/s and plot the frequency response of the filter using `freqs`:

```
[b,a] = besself(5,10000);  
freqs(b,a) % Plot frequency response
```





## Limitations

Lowpass Bessel filters have a monotonically decreasing magnitude response, as do lowpass Butterworth filters. Compared to the Butterworth, Chebyshev, and elliptic filters, the Bessel filter has the slowest rolloff and requires the highest order to meet an attenuation specification.

For high order filters, the state-space form is the most numerically accurate, followed by the zero-pole-gain form. The transfer function coefficient form is the least accurate; numerical problems can arise for filter orders as low as 15.

## Algorithm

`besself` performs a four-step algorithm:

- 1 It finds lowpass analog prototype poles, zeros, and gain using the `besselap` function.
- 2 It converts the poles, zeros, and gain into state-space form.
- 3 It transforms the lowpass prototype into a lowpass filter that meets the design specifications.
- 4 It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

## See Also

`besselap`, `butter`, `cheby1`, `cheby2`, `ellip`

# bilinear

---

**Purpose** Bilinear transformation method for analog-to-digital filter conversion

**Syntax**

```
[zd,pd,kd] = bilinear(z,p,k,fs)
[zd,pd,kd] = bilinear(z,p,k,fs,fp)
[numd,dend] = bilinear(num,den,fs)
[numd,dend] = bilinear(num,den,fs,fp)
[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs)
[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs,fp)
```

**Description** The *bilinear transformation* is a mathematical mapping of variables. In digital filtering, it is a standard method of mapping the  $s$  or analog plane into the  $z$  or digital plane. It transforms analog filters, designed using classical filter design techniques, into their discrete equivalents.

The bilinear transformation maps the  $s$ -plane into the  $z$ -plane by

$$H(z) = H(s) \Big|_{s = 2f_s \frac{z-1}{z+1}}$$

This transformation maps the  $j\Omega$  axis (from  $\Omega = -\infty$  to  $+\infty$ ) repeatedly around the unit circle ( $e^{j\omega}$ , from  $\omega = -\pi$  to  $\pi$ ) by

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

`bilinear` can accept an optional parameter `Fp` that specifies prewarping. `fp`, in hertz, indicates a “match” frequency, that is, a frequency for which the frequency responses before and after mapping match exactly. In prewarped mode, the bilinear transformation maps the  $s$ -plane into the  $z$ -plane with

$$H(z) = H(s) \Big|_{s = \frac{2\pi f_p (z-1)}{\tan\left(\pi \frac{f_p}{f_s}\right) (z+1)}}$$

With the prewarping option, `bilinear` maps the  $j\Omega$  axis (from  $\Omega = -\infty$  to  $+\infty$ ) repeatedly around the unit circle ( $e^{j\omega}$ , from  $\omega = -\pi$  to  $\pi$ ) by

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

In prewarped mode, `bilinear` matches the frequency  $2\pi f_p$  (in radians per second) in the  $s$ -plane to the normalized frequency  $2\pi f_p/f_s$  (in radians per second) in the  $z$ -plane.

The `bilinear` function works with three different linear system representations: zero-pole-gain, transfer function, and state-space form.

### Zero-Pole-Gain

`[zd,pd,kd] = bilinear(z,p,k,fs)` and

`[zd,pd,kd] = bilinear(z,p,k,fs,fp)` convert the  $s$ -domain transfer function specified by `z`, `p`, and `k` to a discrete equivalent. Inputs `z` and `p` are column vectors containing the zeros and poles, `k` is a scalar gain, and `fs` is the sampling frequency in hertz. `bilinear` returns the discrete equivalent in column vectors `zd` and `pd` and scalar `kd`. The optional match frequency, `fp` is in hertz and is used for prewarping.

### Transfer Function

`[numd,dend] = bilinear(num,den,fs)` and

`[numd,dend] = bilinear(num,den,fs,fp)` convert an  $s$ -domain transfer function given by `num` and `den` to a discrete equivalent. Row vectors `num` and `den` specify the coefficients of the numerator and denominator, respectively, in descending powers of  $s$ .

$$\frac{num(s)}{den(s)} = \frac{num(1)s^n + \dots + num(n)s + num(n+1)}{den(1)s^m + \dots + den(m)s + den(m+1)}$$

`fs` is the sampling frequency in hertz. `bilinear` returns the discrete equivalent in row vectors `numd` and `dend` in descending powers of  $z$  (ascending powers of  $z^{-1}$ ). `fp` is the optional match frequency, in hertz, for prewarping.

## State-Space

`[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs)` and

`[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs,fp)` convert the continuous-time state-space system in matrices A, B, C, D

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

to the discrete-time system:

$$x[n+1] = A_d x[n] + B_d u[n]$$

$$y[n] = C_d x[n] + D_d u[n]$$

`fs` is the sampling frequency in hertz. `bilinear` returns the discrete equivalent in matrices `Ad`, `Bd`, `Cd`, `Dd`. The optional match frequency, `fp` is in hertz and is used for prewarping.

## Algorithm

`bilinear` uses one of two algorithms depending on the format of the input linear system you supply. One algorithm works on the zero-pole-gain format and the other on the state-space format. For transfer function representations, `bilinear` converts to state-space form, performs the transformation, and converts the resulting state-space system back to transfer function form.

### Zero-Pole-Gain Algorithm

For a system in zero-pole-gain form, `bilinear` performs four steps:

- 1 If `fp` is present, it prewarps:  
`fp = 2*pi*fp;`  
`fs = fp/tan(fp/fs/2)`  
otherwise, `fs = 2*fs.`
- 2 It strips any zeros at  $\pm\infty$  using  
`z = z(finite(z));`

- 3 It transforms the zeros, poles, and gain using

```
pd = (1+p/fs)./(1-p/fs);      %Do bilinear transformation
zd = (1+z/fs)./(1-z/fs);
kd = real(k*prod(fs-z)./prod(fs-p));
```

- 4 It adds extra zeros at -1 so the resulting system has equivalent numerator and denominator order.

### State-Space Algorithm

For a system in state-space form, bilinear performs two steps:

- 1 If  $f_p$  is present,  $k = 2\pi f_p / \tan(\pi f_p / f_s)$ ; else  $k = 2f_s$ .
- 2 It computes  $A_d$ ,  $B_d$ ,  $C_d$ , and  $D_d$  in terms of  $A$ ,  $B$ ,  $C$ , and  $D$  using

$$A_d = \left( I + \left( \frac{1}{k} \right) A \right) \left( I - \left( \frac{1}{k} \right) A \right)^{-1}$$

$$B_d = \frac{2k}{r} \left( I - \left( \frac{1}{k} \right) A \right)^{-1} B$$

$$C_d = r C \left( I - \left( \frac{1}{k} \right) A \right)^{-1}$$

$$D_d = \left( \frac{1}{k} \right) C \left( I - \left( \frac{1}{k} \right) A \right)^{-1} B + D$$

bilinear implements these relations using conventional MATLAB statements. The scalar  $r$  is arbitrary; bilinear uses  $r = \sqrt{2/k}$  to ensure good quantization noise properties in the resulting system.

### Diagnostics

bilinear requires that the numerator order be no greater than the denominator order. If this is not the case, bilinear displays

```
Numerator cannot be higher order than denominator.
```

For bilinear to distinguish between the zero-pole-gain and transfer function linear system formats, the first two input parameters must be vectors with the same orientation in these cases. If this is not the case, bilinear displays

```
First two arguments must have the same orientation.
```

# bilinear

---

## References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Pgs. 209-213.

[2] Oppenheim, A.V., and R.W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 450-454.

## See Also

[impinvar](#), [lp2bp](#), [lp2bs](#), [lp2hp](#), [lp2lp](#)

**Purpose** Permute data into bit-reversed order

**Syntax**  
`y = bitrevorder(x)`  
`[y,i] = bitrevorder(x)`

**Description** `bitrevorder` is useful for pre-arranging filter coefficients so that bit-reversed ordering does not have to be performed as part of an `fft` or inverse FFT (`ifft`) computation. This can improve run-time efficiency for external applications or for Simulink Blockset models. Both MATLAB `fft` and `ifft` process linear input and output.

---

**Note** Using `bitrevorder` is equivalent to using `digitrevorder` with radix base 2.

---

`y = bitrevorder(x)` returns the input data in bit-reversed order in vector or matrix `y`. The length of `x` must be an integer power of 2. If `x` is a matrix, the bit-reversal occurs on the first dimension of `x` with size greater than 1. `y` is the same size as `x`.

`[y,i] = bitrevorder(x)` returns the bit-reversed vector or matrix `y` and the bit-reversed indices `i`, such that `y = x(i)`. Recall that MATLAB uses 1-based indexing, so the first index of `y` will be 1, not 0.

The following table shows the numbers 0 through 7, the corresponding bits and the bit-reversed numbers.

Linear Index	Bits	Bit-Reversed	Bit-Reversed Index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1

# bitrevorder

---

Linear Index	Bits	Bit-Reversed	Bit-Reversed Index
5	101	101	5
6	110	011	3
7	111	111	7

## Examples

Obtain the bit-reversed ordered output of a vector:

```
x=[0:7]';           % Create a column vector
[x,bitrevorder(x)]
ans =
     0     0
     1     4
     2     2
     3     6
     4     1
     5     5
     6     3
     7     7
```

## See Also

fft, digitrevorder, ifft



**Purpose** Blackman window

**Syntax**

```
w = blackman(L)
w = blackman(L, 'sflag')
```

**Description** `w = blackman(L)` returns the L-point symmetric Blackman window in the column vector `w`, where `L` is a positive integer.

`w = blackman(L, 'sflag')` returns an L-point Blackman window using the window sampling specified by `'sflag'`, which can be either `'periodic'` or `'symmetric'` (the default). When `'periodic'` is specified, `blackman` computes a length `L+1` window and returns the first `L` points.

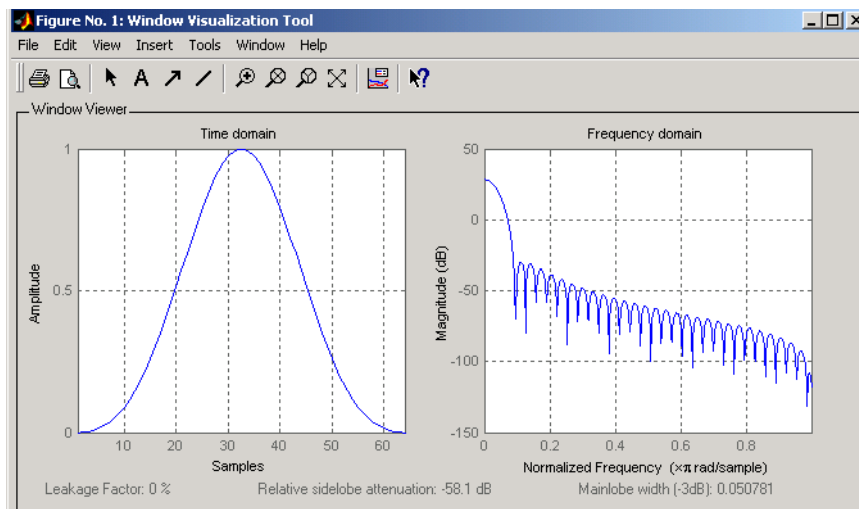
---

**Note** If you specify a one-point window (set `L=1`), the value 1 is returned.

---

**Examples** Create a 64-point Blackman window and display the result using WVTool:

```
L=64;
wvtool(blackman(L))
```



# blackman

---

## Algorithm

The equation for computing the coefficients of a Blackman window is

$$w(n) = 0.42 - 0.5 \cos\left(2\pi\frac{n}{N}\right) + 0.08 \cos\left(4\pi\frac{n}{N}\right) \quad 0 \leq n \leq N$$

The window length  $L = N+1$ .

Blackman windows have slightly wider central lobes and less sideband leakage than equivalent length Hamming and Hann windows.

## See Also

flattopwin, hamming, hann, window, wintool, wvtool

## References

[1] Oppenheim, A.V., and R.W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

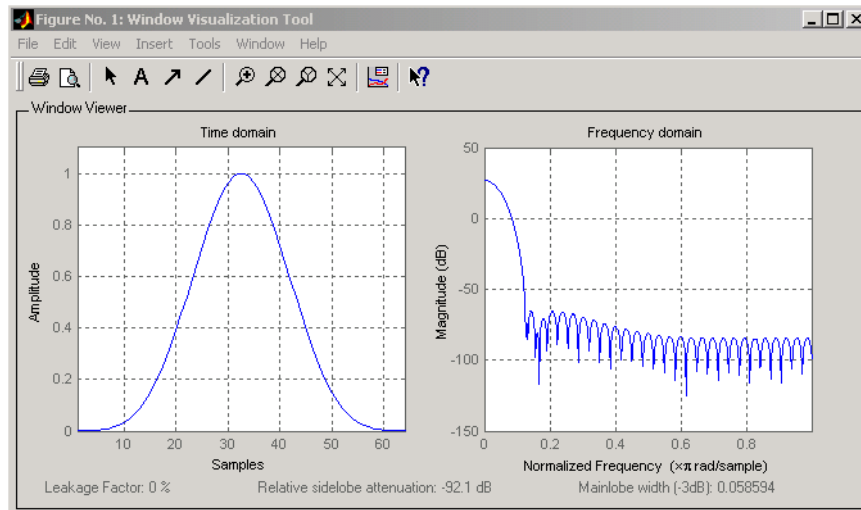
**Purpose** Minimum 4-term Blackman-Harris window

**Syntax** `w = blackmanharris(L)`

**Description** `w = blackmanharris(L)` returns an L-point, minimum, 4-term Blackman-harris window in the column vector `w`. The window is minimum in the sense that its maximum sidelobes are minimized.

**Examples** Create a 32-point Blackman-Harris window and display the result using WVTool:

```
L=32;
wvtool(blackmanharris(L))
```



**Algorithm** The equation for computing the coefficients of a minimum 4-term Blackman-Harris window is

$$w(n) = a_0 + a_1 \cos\left(\frac{2\pi}{N}n\right) + a_2 \cos\left(\frac{2\pi}{N}2n\right) + a_3 \cos\left(\frac{2\pi}{N}3n\right)$$

where  $-\frac{N}{2} \leq n \leq \frac{N}{2}$  and the window length  $L = N+1$ .

# blackmanharris

---

The coefficients for this window are

$$a_0 = 0.35875$$

$$a_1 = 0.48829$$

$$a_2 = 0.14128$$

$$a_3 = 0.01168$$

## References

[1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66 (January 1978). pp. 51-84.

## See Also

barthannwin, bartlett, bohmanwin, nuttallwin, parzenwin, rectwin, triang, window, wintool, wvtool

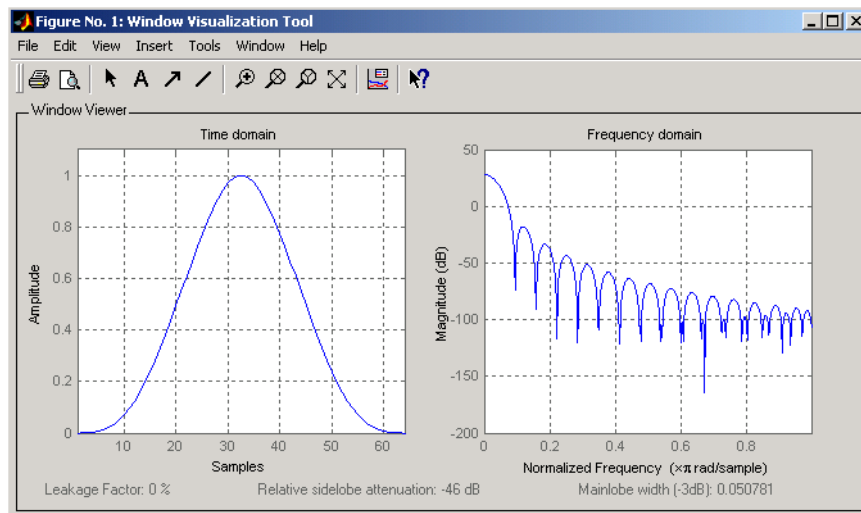
**Purpose** Bohman window

**Syntax** `w = bohmanwin(L)`

**Description** `w = bohmanwin(L)` returns an L-point Bohman window in column vector `w`. A Bohman window is the convolution of two half-duration cosine lobes. In the time domain, it is the product of a triangular window and a single cycle of a cosine with a term added to set the first derivative to zero at the boundary. Bohman windows fall off as  $1/w^4$ .

**Examples** Compute a 64-point Bohman window and display the result using WVTool:

```
L=64;
wvtool(bohmanwin(L))
```



**Algorithm** The equation for computing the coefficients of a Bohman window is

$$w(n) = \left(1.0 - \frac{|n|}{N/2}\right) \cos\left(\pi \frac{|n|}{N/2}\right) + \frac{1}{\pi} \sin\left(\pi \frac{|n|}{N/2}\right)$$

where  $0 \leq |n| \leq \frac{N}{2}$  and the window length  $L = N+1$ .

# bohmanwin

---

## References

[1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66 (January 1978). p. 67.

## See Also

barthannwin, bartlett, blackmanharris, nuttallwin, parzenwin, rectwin, triang, window, wintool, wvtool

**Purpose** Buffer signal vector into matrix of data frames

**Syntax**

```
y = buffer(x,n)
y = buffer(x,n,p)
y = buffer(x,n,p,opt)
[y,z] = buffer(...)
[y,z,opt] = buffer(...)
```

**Description**  $y = \text{buffer}(x,n)$  partitions a length- $L$  signal vector  $x$  into nonoverlapping data segments (frames) of length  $n$ . Each data frame occupies one column of matrix output  $y$ , which has  $n$  rows and  $\text{ceil}(L/n)$  columns. If  $L$  is not evenly divisible by  $n$ , the last column is zero-padded to length  $n$ .

$y = \text{buffer}(x,n,p)$  overlaps or underlaps successive frames in the output matrix by  $p$  samples:

- For  $0 < p < n$  (overlap), `buffer` repeats the final  $p$  samples of each frame at the beginning of the following frame. For example, if  $x = 1:30$  and  $n = 7$ , an overlap of  $p = 3$  looks like this.

$y =$

0	2	6	10	14	18	22	26
0	3	7	11	15	19	23	27
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	0
4	8	12	16	20	24	28	0

The first frame starts with  $p$  zeros (the default initial condition), and the number of columns in  $y$  is  $\text{ceil}(L/(n-p))$ .

- For  $p < 0$  (underlap), `buffer` skips  $p$  samples between consecutive frames. For example, if  $x = 1:30$  and  $n = 7$ , a buffer with underlap of  $p = -3$  looks like this.

# buffer

```

y =
  1  11  21
  2  12  22
  3  13  23
  4  14  24
  5  15  25
  6  16  26
  7  17  27

```

skipped  $\left\{ \begin{array}{ccc} 8 & 18 & 28 \\ 9 & 19 & 29 \\ 10 & 20 & 30 \end{array} \right\}$

The number of columns in  $y$  is  $\text{ceil}(L/(n-p))$ .

$y = \text{buffer}(x, n, p, \text{opt})$  specifies a vector of samples to precede  $x(1)$  in an overlapping buffer, or the number of initial samples to skip in an underlapping buffer:

- For  $0 < p < n$  (overlap),  $\text{opt}$  specifies a length- $p$  vector to insert before  $x(1)$  in the buffer. This vector can be considered an *initial condition*, which is needed when the current buffering operation is one in a sequence of consecutive buffering operations. To maintain the desired frame overlap from one buffer to the next,  $\text{opt}$  should contain the final  $p$  samples of the previous buffer in the sequence. See “Continuous Buffering” below.

By default,  $\text{opt}$  is  $\text{zeros}(p, 1)$  for an overlapping buffer. Set  $\text{opt}$  to 'nodelay' to skip the initial condition and begin filling the buffer immediately with  $x(1)$ . In this case,  $L$  must be  $\text{length}(p)$  or longer. For example, if  $x = 1:30$  and  $n = 7$ , a buffer with overlap of  $p = 3$  looks like this.

```

y =
  1  5  9  13  17  21  25
  2  6  10 14  18  22  26
  3  7  11 15  19  23  27
  4  8  12 16  20  24  28
  5  9  13 17  21  25  29
  6 10 14 18  22  26  30
  7 11 15 19  23  27  0

```

- For  $p < 0$  (underlap),  $\text{opt}$  is an integer value in the range  $[0, -p]$  specifying the number of initial input samples,  $x(1:\text{opt})$ , to skip before adding samples



to the buffer. The first value in the buffer is therefore  $x(\text{opt}+1)$ . By default,  $\text{opt}$  is zero for an underlapping buffer.

This option is especially useful when the current buffering operation is one in a sequence of consecutive buffering operations. To maintain the desired frame underlap from one buffer to the next,  $\text{opt}$  should equal the difference between the total number of points to skip between frames ( $p$ ) and the number of points that were *available* to be skipped in the previous input to buffer. If the previous input had fewer than  $p$  points that could be skipped after filling the final frame of that buffer, the remaining  $\text{opt}$  points need to be removed from the first frame of the current buffer. See “Continuous Buffering” below for an example of how this works in practice.

`[y,z] = buffer(...)` partitions the length- $L$  signal vector  $x$  into frames of length  $n$ , and outputs only the *full* frames in  $y$ . If  $y$  is an overlapping buffer, it has  $n$  rows and  $m$  columns, where

$$m = \text{floor}(L/(n-p)) \quad \% \text{ When } \text{length}(\text{opt}) = p$$

or

$$m = \text{floor}((L-n)/(n-p))+1 \quad \% \text{ When } \text{opt} = \text{'nodelay'}$$

If  $y$  is an underlapping buffer, it has  $n$  rows and  $m$  columns, where

$$m = \text{floor}((L-\text{opt})/(n-p)) + (\text{rem}((L-\text{opt}), (n-p)) \geq n)$$

If the number of samples in the input vector (after the appropriate overlapping or underlapping operations) exceeds the number of places available in the  $n$ -by- $m$  buffer, the remaining samples in  $x$  are output in vector  $z$ , which for an overlapping buffer has length

$$\text{length}(z) = L - m*(n-p) \quad \% \text{ When } \text{length}(\text{opt}) = p$$

or

$$\text{length}(z) = L - ((m-1)*(n-p)+n) \% \text{ When } \text{opt} = \text{'nodelay'}$$

and for an underlapping buffer has length

$$\text{length}(z) = (L-\text{opt}) - m*(n-p)$$

Output  $z$  shares the same orientation (row or column) as  $x$ . If there are no remaining samples in the input after the buffer with the specified overlap or underlap is filled,  $z$  is an empty vector.

`[y,z,opt] = buffer(...)` returns the last  $p$  samples of a overlapping buffer in output  $opt$ . In an underlapping buffer,  $opt$  is the difference between the total number of points to skip between frames ( $-p$ ) and the number of points in  $x$  that were *available* to be skipped after filling the last frame:

- For  $0 < p < n$  (overlap),  $opt$  (as an output) contains the final  $p$  samples in the last frame of the buffer. This vector can be used as the *initial condition* for a subsequent buffering operation in a sequence of consecutive buffering operations. This allows the desired frame overlap to be maintained from one buffer to the next. See “Continuous Buffering” below.
- For  $p < 0$  (underlap),  $opt$  (as an output) is the difference between the total number of points to skip between frames ( $-p$ ) and the number of points in  $x$  that were *available* to be skipped after filling the last frame.

`opt = m*(n-p) + opt - L`     %  $z$  is the empty vector.

where  $opt$  on the right-hand side is the input argument to `buffer`, and  $opt$  on the left-hand side is the output argument. Here  $m$  is the number of columns in the buffer, which is

`m = floor((L-opt)/(n-p)) + (rem((L-opt), (n-p))>=n)`

Note that for an underlapping buffer output  $opt$  is always zero when output  $z$  contains data.

The  $opt$  output for an underlapping buffer is especially useful when the current buffering operation is one in a sequence of consecutive buffering operations. The  $opt$  output from each buffering operation specifies the number of samples that need to be skipped at the start of the next buffering operation to maintain the desired frame underlap from one buffer to the next. If fewer than  $p$  points were available to be skipped after filling the final frame of the current buffer, the remaining  $opt$  points need to be removed from the first frame of the next buffer.

In a sequence of buffering operations, the  $opt$  output from each operation should be used as the  $opt$  input to the subsequent buffering operation. This ensures that the desired frame overlap or underlap is maintained from buffer

to buffer, as well as from frame to frame within the same buffer. See “Continuous Buffering” below for an example of how this works in practice.

### Continuous Buffering

In a continuous buffering operation, the vector input to the buffer function represents one frame in a sequence of frames that make up a discrete signal. These signal frames can originate in a frame-based data acquisition process, or within a frame-based algorithm like the FFT.

As an example, you might acquire data from an A/D card in frames of 64 samples. In the simplest case, you could rebuffer the data into frames of 16 samples; `buffer` with `n = 16` creates a buffer of four frames from each 64-element input frame. The result is that the signal of frame size 64 has been converted to a signal of frame size 16; no samples were added or removed.

In the general case where the original signal frame size,  $L$ , is not equally divisible by the new frame size,  $n$ , the overflow from the last frame needs to be captured and recycled into the following buffer. You can do this by iteratively calling `buffer` on input  $x$  with the two-output-argument syntax:

```
[y,z] = buffer([z;x],n)    % x is a column vector.  
[y,z] = buffer([z,x],n)  % x is a row vector.
```

This simply captures any buffer overflow in  $z$ , and prepends the data to the subsequent input in the next call to `buffer`. Again, the input signal,  $x$ , of frame size  $L$ , has been converted to a signal of frame size  $n$  without any insertion or deletion of samples.

Note that continuous buffering cannot be done with the single-output syntax `y = buffer(...)`, because the last frame of  $y$  in this case is zero padded, which adds new samples to the signal.

Continuous buffering in the presence of overlap and underlap is handled with the `opt` parameter, which is used as both an input and output to `buffer`. The following two examples demonstrate how the `opt` parameter should be used.

## Examples

### Example 1: Continuous Overlapping Buffers

First create a buffer containing 100 frames, each with 11 samples:

```
data = buffer(1:1100,11); % 11 samples per frame
```

Imagine that the frames (columns) in the matrix called `data` are the sequential outputs of a data acquisition board sampling a physical signal: `data(:,1)` is the first D/A output, containing the first 11 signal samples; `data(:,2)` is the second output, containing the next 11 signal samples, and so on.

You want to rebuffer this signal from the acquired frame size of 11 to a frame size of 4 with an overlap of 1. To do this, you will repeatedly call `buffer` to operate on each successive input frame, using the `opt` parameter to maintain consistency in the overlap from one buffer to the next.

Set the buffer parameters:

```
n = 4;          % New frame size
p = 1;          % Overlap
opt = -5;       % Value of y(1)
z = [];        % Initialize the carry-over vector.
```

Now repeatedly call `buffer`, each time passing in a new signal frame from `data`. Note that overflow samples (returned in `z`) are carried over and prepended to the input in the subsequent call to `buffer`:

```
for i=1:size(data,2), % Loop over each source frame (column).
    x = data(:,i);    % A single frame of the D/A output

    [y,z,opt] = buffer([z;x],n,p,opt);

    disp(y);          % Display the buffer of data.
    pause
end
```

Here's what happens during the first four iterations.

Iteration	Input frame [z;x]'	opt (input)	opt (output)	Output buffer (y)	Overflow (z)
i=1	[1:11]	5	9	$\begin{bmatrix} -5 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$	[10 11]
i=2	[10 11 12:22]	9	21	$\begin{bmatrix} 9 & 12 & 15 & 18 \\ 10 & 13 & 16 & 19 \\ 11 & 14 & 17 & 20 \\ 12 & 15 & 18 & 21 \end{bmatrix}$	[22]
i=3	[22 23:33]	21	33	$\begin{bmatrix} 21 & 24 & 27 & 30 \\ 22 & 25 & 28 & 31 \\ 23 & 26 & 29 & 32 \\ 24 & 27 & 30 & 33 \end{bmatrix}$	[]
i=4	[34:44]	33	42	$\begin{bmatrix} 33 & 36 & 39 \\ 34 & 37 & 40 \\ 35 & 38 & 41 \\ 36 & 39 & 42 \end{bmatrix}$	[43 44]

Note that the size of the output matrix, y, can vary by a single column from one iteration to the next. This is typical for buffering operations with overlap or underlap.

## Example 2: Continuous Underlapping Buffers

Again create a buffer containing 100 frames, each with 11 samples:

```
data = buffer(1:1100,11); % 11 samples per frame
```

Again, imagine that `data(:,1)` is the first D/A output, containing the first 11 signal samples; `data(:,2)` is the second output, containing the next 11 signal samples, and so on.

You want to rebuffer this signal from the acquired frame size of 11 to a frame size of 4 with an underlap of 2. To do this, you will repeatedly call `buffer` to operate on each successive input frame, using the `opt` parameter to maintain consistency in the underlap from one buffer to the next.

Set the buffer parameters:

```
n = 4;      % New frame size
p = -2;     % Underlap
opt = 1;    % Skip the first input element, x(1).
z = [];     % Initialize the carry-over vector.
```

Now repeatedly call `buffer`, each time passing in a new signal frame from `data`. Note that overflow samples (returned in `z`) are carried over and prepended to the input in the subsequent call to `buffer`:

```
for i=1:size(data,2),      % Loop over each source frame (column).
    x = data(:,i);        % A single frame of the D/A output

    [y,z,opt] = buffer([z;x],n,p,opt);

    disp(y);             % Display the buffer of data.
    pause
end
```

Here's what happens during the first three iterations.

Iteration	Input frame [z;x]'	opt (input)	opt (output)	Output buffer (y)	Overflow (z)
i=1	[1:11]	1	2		[]
i=2	[12:22]	2	0		[20 21 22]
i=3	[20 21 22 23:33]	0	0		[32 33]

## Diagnostics

Error messages are displayed when  $p \geq n$  or  $\text{length}(\text{opt}) \neq \text{length}(p)$  in an overlapping buffer case:

Frame overlap P must be less than the buffer size N.

Initial conditions must be specified as a length-P vector.

## See Also

reshape

# buttap

---

**Purpose** Butterworth analog lowpass filter prototype

**Syntax** [z,p,k] = buttap(n)

**Description** [z,p,k] = buttap(n) returns the poles and gain of an order n Butterworth analog lowpass filter prototype. The function returns the poles in the length n column vector p and the gain in scalar k. z is an empty matrix because there are no zeros. The transfer function is

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

Butterworth filters are characterized by a magnitude response that is maximally flat in the passband and monotonic overall. In the lowpass case, the first  $2n-1$  derivatives of the squared magnitude response are zero at  $\omega = 0$ . The squared magnitude response function is

$$|H(\omega)|^2 = \frac{1}{1 + (\omega/\omega_0)^{2n}}$$

corresponding to a transfer function with poles equally spaced around a circle in the left half plane. The magnitude response at the cutoff angular frequency  $\omega_0$  is always  $1/\sqrt{2}$  regardless of the filter order. buttap sets  $\omega_0$  to 1 for a normalized result.

**Algorithm**

```
z = [];  
p = exp(sqrt(-1)*(pi*(1:2:2*n-1)/(2*n)+pi/2)).';  
k = real(prod(-p));
```

**References** [1] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Chapter 7.

**See Also** besslap, butter, cheb1ap, cheb2ap, ellipap



**Purpose** Butterworth analog and digital filter design

**Syntax**

```
[b,a] = butter(n,Wn)
[b,a] = butter(n,Wn,'ftype')
[b,a] = butter(n,Wn,'s')
[b,a] = butter(n,Wn,'ftype','s')
[z,p,k] = butter(...)
[A,B,C,D] = butter(...)
```

**Description** butter designs lowpass, bandpass, highpass, and bandstop digital and analog Butterworth filters. Butterworth filters are characterized by a magnitude response that is maximally flat in the passband and monotonic overall.

Butterworth filters sacrifice rolloff steepness for monotonicity in the pass- and stopbands. Unless the smoothness of the Butterworth filter is needed, an elliptic or Chebyshev filter can generally provide steeper rolloff characteristics with a lower filter order.

### Digital Domain

`[b,a] = butter(n,Wn)` designs an order  $n$  lowpass digital Butterworth filter with normalized cutoff frequency  $Wn$ . It returns the filter coefficients in length  $n+1$  row vectors  $b$  and  $a$ , with coefficients in descending powers of  $z$ .

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}}{1 + a(2)z^{-1} + \dots + a(n+1)z^{-n}}$$

*Cutoff frequency* is that frequency where the magnitude response of the filter is  $\sqrt{1/2}$ . For `butter`, the normalized cutoff frequency  $Wn$  must be a number between 0 and 1, where 1 corresponds to the Nyquist frequency,  $\pi$  radians per sample.

If  $Wn$  is a two-element vector,  $Wn = [w1 \ w2]$ , `butter` returns an order  $2*n$  digital bandpass filter with passband  $w1 < \omega < w2$ .

`[b,a] = butter(n,Wn,'ftype')` designs a highpass, lowpass, or bandstop filter, where the string `'ftype'` is one of the following.

- 'high' for a highpass digital filter with normalized cutoff frequency  $Wn$
- 'low' for a lowpass digital filter with normalized cutoff frequency  $Wn$

- 'stop' for an order  $2*n$  bandstop digital filter if  $Wn$  is a two-element vector,  $Wn = [w1 \ w2]$ . The stopband is  $w1 < \omega < w2$ .

With different numbers of output arguments, `butter` directly obtains other realizations of the filter. To obtain zero-pole-gain form, use three output arguments as shown below:

```
[z,p,k] = butter(n,Wn) or
```

```
[z,p,k] = butter(n,Wn,'ftype')
```

 returns the zeros and poles in length  $n$  column vectors  $z$  and  $p$ , and the gain in the scalar  $k$ .

To obtain state-space form, use four output arguments as shown below:

```
[A,B,C,D] = butter(n,Wn) or
```

```
[A,B,C,D] = butter(n,Wn,'ftype')
```

 where  $A$ ,  $B$ ,  $C$ , and  $D$  are

$$x[n+1] = Ax[n] + Bu[n]$$

$$y[n] = Cx[n] + Du[n]$$

and  $u$  is the input,  $x$  is the state vector, and  $y$  is the output.

## Analog Domain

`[b,a] = butter(n,Wn,'s')` designs an order  $n$  lowpass analog Butterworth filter with angular cutoff frequency  $Wn$  rad/s. It returns the filter coefficients in the length  $n+1$  row vectors  $b$  and  $a$ , in descending powers of  $s$ , derived from this transfer function:

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \dots + b(n+1)}{s^n + a(2)s^{n-1} + \dots + a(n+1)}$$

`butter`'s angular cutoff frequency  $Wn$  must be greater than 0 rad/s.

If  $Wn$  is a two-element vector with  $w1 < w2$ , `butter(n,Wn,'s')` returns an order  $2*n$  bandpass analog filter with passband  $w1 < \omega < w2$ .

`[b,a] = butter(n,Wn,'ftype','s')` designs a highpass, lowpass, or bandstop filter.

With different numbers of output arguments, `butter` directly obtains other realizations of the analog filter. To obtain zero-pole-gain form, use three output arguments as shown below:

```
[z,p,k] = butter(n,Wn,'s') or
```

`[z,p,k] = butter(n,Wn,'ftype','s')` returns the zeros and poles in length `n` or `2*n` column vectors `z` and `p` and the gain in the scalar `k`.

To obtain state-space form, use four output arguments as shown below:

```
[A,B,C,D] = butter(n,Wn,'s') or
```

```
[A,B,C,D] = butter(n,Wn,'ftype','s')
```

 where `A`, `B`, `C`, and `D` are

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

and  $u$  is the input,  $x$  is the state vector, and  $y$  is the output.

## Examples

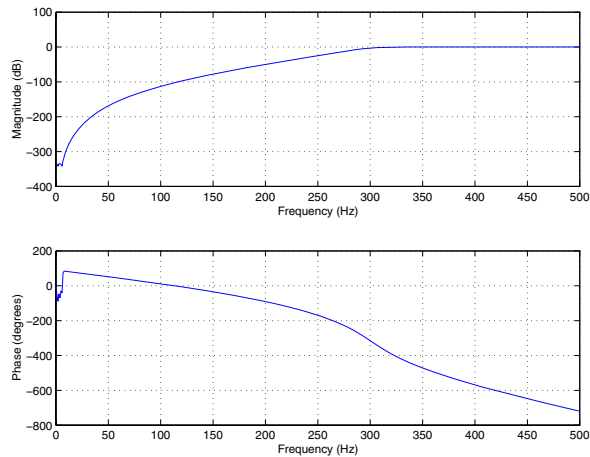
### Example 1

For data sampled at 1000 Hz, design a 9th-order highpass Butterworth filter with cutoff frequency of 300 Hz, which corresponds to a normalized value of 0.6:

```
[b,a] = butter(9,300/500,'high');
```

The filter's frequency response is

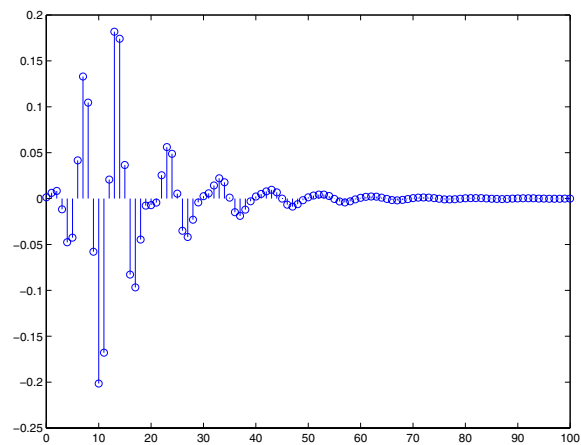
```
freqz(b,a,128,1000)
```



## Example 2

Design a 10th-order bandpass Butterworth filter with a passband from 100 to 200 Hz and plot its impulse response, or *unit sample response*:

```
n = 5; Wn = [100 200]/500;  
[b,a] = butter(n,Wn);  
[y,t] = impz(b,a,101);  
stem(t,y)
```



**Limitations**

For high order filters, the state-space form is the most numerically accurate, followed by the zero-pole-gain form. The transfer function coefficient form is the least accurate; numerical problems can arise for filter orders as low as 15.

**Algorithm**

`butter` uses a five-step algorithm:

- 1** It finds the lowpass analog prototype poles, zeros, and gain using the `buttap` function.
- 2** It converts the poles, zeros, and gain into state-space form.
- 3** It transforms the lowpass filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies, using a state-space transformation.
- 4** For digital filter design, `butter` uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at  $\omega_n$  or  $\omega_1$  and  $\omega_2$ .
- 5** It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

**See Also**

`besself`, `buttap`, `buttord`, `cheby1`, `cheby2`, `ellip`, `maxflat`

# buttord

---

**Purpose** Butterworth filter order and cutoff frequency

**Syntax**  $[n, Wn] = \text{buttord}(Wp, Ws, Rp, Rs)$   
 $[n, Wn] = \text{buttord}(Wp, Ws, Rp, Rs, 's')$

**Description** buttord calculates the minimum order of a digital or analog Butterworth filter required to meet a set of filter design specifications.

## Digital Domain

$[n, Wn] = \text{buttord}(Wp, Ws, Rp, Rs)$  returns the lowest order,  $n$ , of the digital Butterworth filter that loses no more than  $Rp$  dB in the passband and has at least  $Rs$  dB of attenuation in the stopband. The scalar (or vector) of corresponding cutoff frequencies,  $Wn$ , is also returned. Use the output arguments  $n$  and  $Wn$  in butter.

Choose the input arguments to specify the stopband and passband according to the following table.

## Description of Stopband and Passband Filter Parameters

Parameter	Description
$Wp$	Passband corner frequency $Wp$ , the cutoff frequency, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, $\pi$ radians per sample.
$Ws$	Stopband corner frequency $Ws$ , is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.
$Rp$	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.
$Rs$	Stopband attenuation, in decibels. This value is the number of decibels the stopband is down from the passband.

Use the following guide to specify filters of different types.

**Filter Type Stopband and Passband Specifications**

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	$W_p < W_s$ , both scalars	$(W_s, 1)$	$(0, W_p)$
Highpass	$W_p > W_s$ , both scalars	$(0, W_s)$	$(W_p, 1)$
Bandpass	The interval specified by $W_s$ contains the one specified by $W_p$ $(W_s(1) < W_p(1) < W_p(2) < W_s(2))$ .	$(0, W_s(1))$ and $(W_s(2), 1)$	$(W_p(1), W_p(2))$
Bandstop	The interval specified by $W_p$ contains the one specified by $W_s$ $(W_p(1) < W_s(1) < W_s(2) < W_p(2))$ .	$(0, W_p(1))$ and $(W_p(2), 1)$	$(W_s(1), W_s(2))$

If your filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design separate lowpass and highpass filters according to the specifications in this table, and cascade the two filters together.

**Analog Domain**

$[n, W_n] = \text{buttor}(W_p, W_s, R_p, R_s, 's')$  finds the minimum order  $n$  and cutoff frequencies  $W_n$  for an analog Butterworth filter. You specify the frequencies  $W_p$  and  $W_s$  similar those described in the Table , Description of Stopband and Passband Filter Parameters table above, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

Use `buttor` for lowpass, highpass, bandpass, and bandstop filters as described in the Table , Filter Type Stopband and Passband Specifications table above.

## Examples

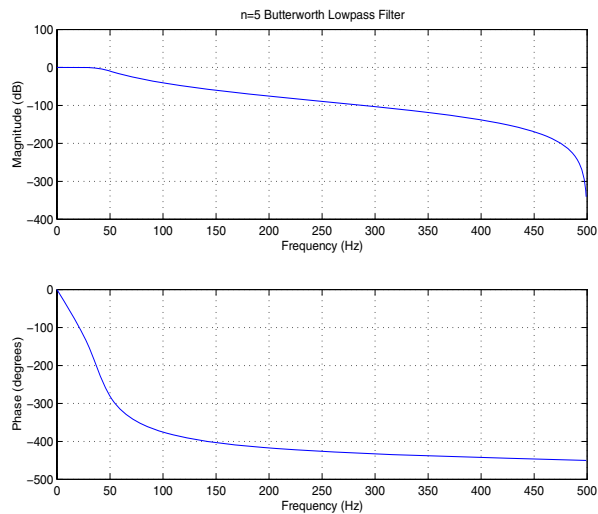
### Example 1

For data sampled at 1000 Hz, design a lowpass filter with less than 3 dB of ripple in the passband, defined from 0 to 40 Hz, and at least 60 dB of attenuation in the stopband, defined from 150 Hz to the Nyquist frequency (500 Hz). Plot the filter's frequency response:

```
Wp = 40/500; Ws = 150/500;  
[n,Wn] = buttord(Wp,Ws,3,60)
```

```
n =  
    5  
Wn =  
    0.0810
```

```
[b,a] = butter(n,Wn);  
freqz(b,a,512,1000); title('n=5 Butterworth Lowpass Filter')
```





### Example 2

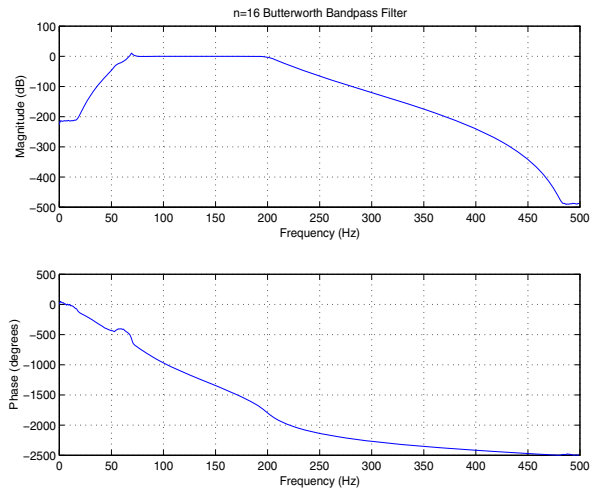
Next design a bandpass filter with passband of 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband, and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband:

```

Wp = [60 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 40;
[n,Wn] = buttord(Wp,Ws,Rp,Rs)

n =
    16
Wn =
    0.1198    0.4005

[b,a] = butter(n,Wn);
freqz(b,a,128,1000)
title('n=16 Butterworth Bandpass Filter')
    
```



# buttord

---

## Algorithm

buttord's order prediction formula is described in [1]. It operates in the analog domain for both analog and digital cases. For the digital case, it converts the frequency parameters to the  $s$ -domain before estimating the order and natural frequency, and then converts back to the  $z$ -domain.

buttord initially develops a lowpass filter prototype by transforming the passband frequencies of the desired filter to 1 rad/s (for lowpass and highpass filters) and to -1 and 1 rad/s (for bandpass and bandstop filters). It then computes the minimum order required for a lowpass filter to meet the stopband specification.

## References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 227.

## See Also

butter, cheb1ord, cheb2ord, ellipord, kaiserord

**Purpose** Complex cepstral analysis

**Syntax**

```
xhat = cceps(x)
[xhat,nd] = cceps(x)
[xhat,nd,xhat1] = cceps(x)
[... ] = cceps(x,n)
```

**Description** Cepstral analysis is a nonlinear signal processing technique that is applied most commonly in speech processing and homomorphic filtering [1].

---

**Note** cceps only works on real data.

---

`xhat = cceps(x)` returns the complex cepstrum of the real data sequence  $x$ . The input is altered, by the application of a linear phase term, to have no phase discontinuity at  $\pm\pi$  radians. That is, it is circularly shifted (after zero padding) by some samples, if necessary, to have zero phase at  $\pi$  radians.

`[xhat,nd] = cceps(x)` returns the number of samples  $nd$  of (circular) delay added to  $x$  prior to finding the complex cepstrum.

`[xhat,nd,xhat1] = cceps(x)` returns a second complex cepstrum, computed using an alternate rooting algorithm, in  $xhat1$ . The alternate method ([1] p.795) is useful for short sequences that can be rooted and do not have zeros on the unit circle. For these signals,  $xhat1$  can provide a verification of  $xhat$ .

`[... ] = cceps(x,n)` zero pads  $x$  to length  $n$  and returns the length  $n$  complex cepstrum of  $x$ .

**Algorithm** `cceps` is an M-file implementation of algorithm 7.1 in [2]. A lengthy Fortran program reduces to these three lines of MATLAB code, which compose the core of `cceps`:

```
h = fft(x);
logh = log(abs(h)) + sqrt(-1)*rcunwrap(angle(h));
y = real(ifft(logh));
```

---

**Note** `rcunwrap` in the above code segment is a special version of `unwrap` that subtracts a straight line from the phase. `rcunwrap` is a local function within `cceps` and is not available for use from the MATLAB command line.

---

### References

- [1] Oppenheim, A.V., and R.W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 788-789.
- [2] IEEE. *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979.

### See Also

`icceps`, `hilbert`, `rcceps`, `unwrap`

**Purpose** Convert second-order sections cell array to a matrix

**Syntax** `m = cell2sos(c)`

**Description** `m = cell2sos(c)` changes a 1-by- $L$  cell array `c` consisting of 1-by-2 cell arrays into an  $L$ -by-6 second-order section matrix `m`. Matrix `m` takes the same form as the matrix generated by `tf2sos`. You can use `m = cell2sos(c)` to invert the results of `c = sos2cell(m)`.

`c` must be a cell array of the form

$$c = \{ \{b_1 \ a_1\} \ \{b_2 \ a_2\} \ \dots \ \{b_L \ a_L\} \}$$

where both  $b_i$  and  $a_i$  are row vectors of at most length 3, and  $i = 1, 2, \dots, L$ . The resulting matrix `m` is given by

$$m = [b_1 \ a_1; b_2 \ a_2; \dots; b_L \ a_L]$$

**See Also** `sos2cell`, `tf2sos`

# cfirpm

---

**Purpose** Complex and nonlinear-phase equiripple FIR filter design

**Syntax**

```
b = cfirpm(n,f,@fresp)
b = cfirpm(n,f,@fresp,w)
b = cfirpm(n,f,a,w)
b = cfirpm(...,'sym')
b = cfirpm(...,'skip_stage2')
b = cfirpm(...,'debug')
b = cfirpm(...,{lgrid})
[b,delta,opt] = cfirpm(...)
```

**Description** `cfirpm` allows arbitrary frequency-domain constraints to be specified for the design of a possibly complex FIR filter. The Chebyshev (or minimax) filter error is optimized, producing equiripple FIR filter designs.

`b = cfirpm(n,f,@fresp)` returns a length  $n+1$  FIR filter with the best approximation to the desired frequency response as returned by function `fresp`, which is called by its function handle (`@fresp`). `f` is a vector of frequency band edge pairs, specified in the range  $-1$  and  $1$ , where  $1$  corresponds to the normalized Nyquist frequency. The frequencies must be in increasing order, and `f` must have even length. The frequency bands span  $f(k)$  to  $f(k+1)$  for  $k$  odd; the intervals  $f(k+1)$  to  $f(k+2)$  for  $k$  odd are “transition bands” or “don’t care” regions during optimization.

Predefined `fresp` frequency response functions are included for a number of common filter designs, as described below. For all of the predefined frequency response functions, the symmetry option `'sym'` defaults to `'even'` if no negative frequencies are contained in `f` and `d = 0`; otherwise `'sym'` defaults to `'none'`. (See the `'sym'` option below for details.) For all of the predefined frequency response functions, `d` specifies a group-delay offset such that the filter response has a group delay of  $n/2+d$  in units of the sample interval. Negative values create less delay; positive values create more delay. By default `d = 0`:

- @lowpass, @highpass, @allpass, @bandpass, @bandstop

These functions share a common syntax, exemplified below by the string 'lowpass'.

`b = cfirpm(n,f,@lowpass,...)` and

`b = cfirpm(n,f,{@lowpass,d},...)` design a linear-phase ( $n/2+d$  delay) filter.

- @multiband designs a linear-phase frequency response filter with arbitrary band amplitudes.

`b = cfirpm(n,f,{@multiband,a},...)` and

`b = cfirpm(n,f,{@multiband,a,d},...)` specify vector `a` containing the desired amplitudes at the band edges in `f`. The desired amplitude at frequencies between pairs of points  $f(k)$  and  $f(k+1)$  for  $k$  odd is the line segment connecting the points  $(f(k), a(k))$  and  $(f(k+1), a(k+1))$ .

- @differentiator designs a linear-phase differentiator. For these designs, zero-frequency must be in a transition band, and band weighting is set to be inversely proportional to frequency.

`b = cfirpm(n,f,{@differentiator,fs},...)` and

`b = cfirpm(n,f,{@differentiator,fs,d},...)` specify the sample rate `fs` used to determine the slope of the differentiator response. If omitted, `fs` defaults to 1.

- @hilbfilt designs a linear-phase Hilbert transform filter response. For Hilbert designs, zero-frequency must be in a transition band.

`b = cfirpm(n,f,@hilbfilt,...)` and

`b = cfirpm(N,F,{@hilbfilt,d},...)` design a linear-phase ( $n/2+d$  delay) Hilbert transform filter.

- @invsinc designs a linear-phase inverse-sinc filter response.

`b = cfirpm(n,f,{@invsinc,a},...)` and

`b = cfirpm(n,f,{@invsinc,a,d},...)` specify gain `a` for the sinc-function, computed as  $\text{sinc}(a*g)$ , where `g` contains the optimization grid frequencies normalized to the range  $[-1,1]$ . By default, `a=1`. The group-delay offset is `d`, such that the filter response will have a group delay of  $N/2 + d$  in units of the sample interval, where `N` is the filter order. Negative values create less delay and positive values create more delay. By default, `d=0`.

`b = cfirpm(n,f,@fresp,w)` uses the real, non-negative weights in vector `w` to weight the fit in each frequency band. The length of `w` is half the length of `f`, so there is exactly one weight per band.

`b = cfirpm(n,f,a,w)` is a synonym for `b = cfirpm(n,f,{@multiband,a},w)`.

`b = cfirpm(...,'sym')` imposes a symmetry constraint on the impulse response of the design, where `'sym'` may be one of the following:

- `'none'` indicates no symmetry constraint. This is the default if any negative band edge frequencies are passed, or if `fresp` does not supply a default.
- `'even'` indicates a real and even impulse response. This is the default for highpass, lowpass, allpass, bandpass, bandstop, `invsinc`, and multiband designs.
- `'odd'` indicates a real and odd impulse response. This is the default for Hilbert and differentiator designs.
- `'real'` indicates conjugate symmetry for the frequency response

If any `'sym'` option other than `'none'` is specified, the band edges should be specified only over positive frequencies; the negative frequency region is filled in from symmetry. If a `'sym'` option is not specified, the `fresp` function is queried for a default setting. Any user-supplied `fresp` function should return a valid `'sym'` string when it is passed the string `'defaults'` as the filter order `N`.

`b = cfirpm(...,'skip_stage2')` disables the second-stage optimization algorithm, which executes only when `cfirpm` determines that an optimal solution has not been reached by the standard `firpm` error-exchange. Disabling this algorithm may increase the speed of computation, but may incur a reduction in accuracy. By default, the second-stage optimization is enabled.

`b = cfirpm(...,'debug')` enables the display of intermediate results during the filter design, where `'debug'` may be one of `'trace'`, `'plots'`, `'both'`, or `'off'`. By default it is set to `'off'`.

`b = cfirpm(...,{lgrid})` uses the integer `lgrid` to control the density of the frequency grid, which has roughly  $2^{\text{nextpow2}(lgrid*n)}$  frequency points. The default value for `lgrid` is 25. Note that the `{lgrid}` argument must be a 1-by-1 cell array.



Any combination of the 'sym', 'skip\_stage2', 'debug', and {lgrid} options may be specified.

[b,delta] = cfirpm(...) returns the maximum ripple height delta.

[b,delta,opt] = cfirpm(...) returns a structure opt of optional results computed by cfirpm and contains the following fields.

Field	Description
opt.fgrid	Frequency grid vector used for the filter design optimization
opt.des	Desired frequency response for each point in opt.fgrid
opt.wt	Weighting for each point in opt.fgrid
opt.H	Actual frequency response for each point in opt.fgrid
opt.error	Error at each point in opt.fgrid
opt.iextr	Vector of indices into opt.fgrid for extremal frequencies
opt.fextr	Vector of extremal frequencies

User-definable functions may be used, instead of the predefined frequency response functions for @fresp. The function is called from within cfirpm using the following syntax

$$[dh,dw] = fresp(n,f,gf,w,p1,p2,\dots)$$

where:

- n is the filter order.
- f is the vector of frequency band edges that appear monotonically between -1 and 1, where 1 corresponds to the Nyquist frequency.
- gf is a vector of grid points that have been linearly interpolated over each specified frequency band by cfirpm. gf determines the frequency grid at which the response function must be evaluated. This is the same data returned by cfirpm in the fgrid field of the opt structure.

- $w$  is a vector of real, positive weights, one per band, used during optimization.  $w$  is optional in the call to `cfirpm`; if not specified, it is set to unity weighting before being passed to `fresp`.
- $dh$  and  $dw$  are the desired complex frequency response and band weight vectors, respectively, evaluated at each frequency in grid  $gf$ .
- $p1, p2, \dots$ , are optional parameters that may be passed to `fresp`.

Additionally, a preliminary call is made to `fresp` to determine the default symmetry property '*sym*'. This call is made using the syntax:

```
sym = fresp('defaults', {n, f, [], w, p1, p2, ...})
```

The arguments may be used in determining an appropriate symmetry default as necessary. The function `private/lowpass.m` may be useful as a template for generating new frequency response functions.

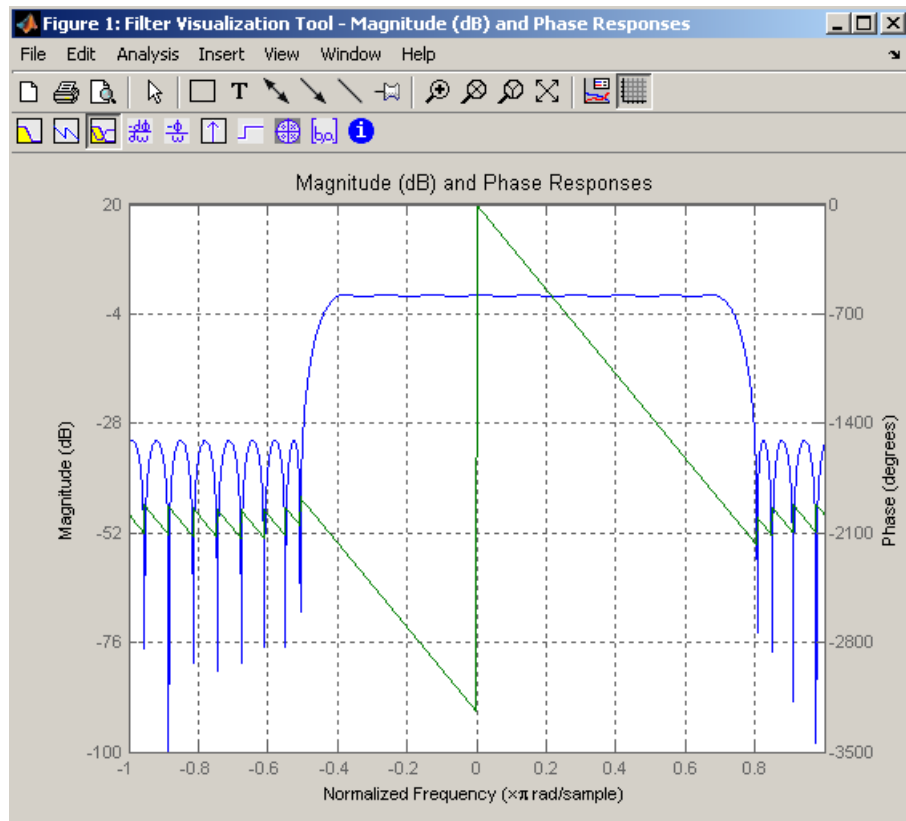
## Examples

### Example 1

Design a 31-tap, linear-phase, lowpass filter:

```
b = cfirpm(30, [-1 -0.5 -0.4 0.7 0.8 1], @lowpass);  
fvtool(b, 1)
```

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



## Example 2

Design a nonlinear-phase allpass FIR filter:

```
n = 22; % Filter order
f = [-1 1]; % Frequency band edges
w = [1 1]; % Weights for optimization
gf = linspace(-1,1,256); % Grid of frequency points
d = exp(-1i*pi*gf*n/2 + 1i*pi*pi*sign(gf).*gf.*gf*(4/pi));
% Desired frequency response
```

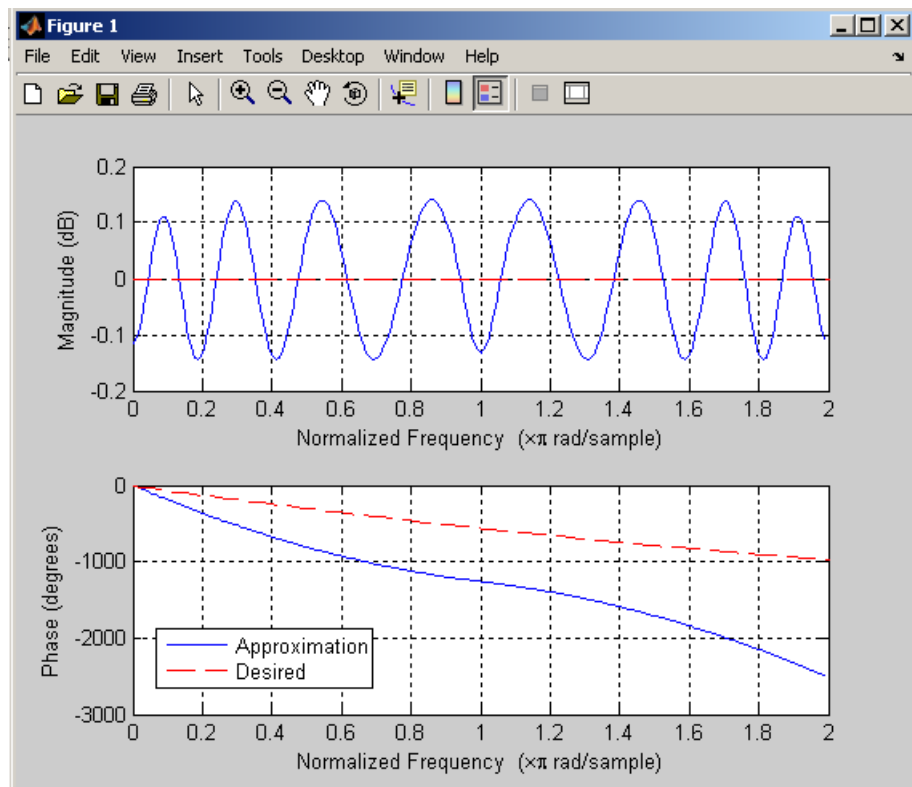
Vector `d` now contains the complex frequency response that we desire for the FIR filter computed by `cfirpm`.

Now compute the FIR filter that best approximates this response:

```
b = cfirpm(n,f,'allpass',w,'real'); % Approximation
freqz(b,1,256,'whole');

subplot(2,1,1); hold on %overlay desired response
plot(pi*(gf+1),20*log10(abs(fftshift(d))),'r--')

subplot(2,1,2); hold on
plot(pi*(gf+1),unwrap(angle(fftshift(d)))*180/pi,'r--')
legend('Approximation','Desired')
```



## Algorithm

An extended version of the Remez exchange method is implemented for the complex case. This exchange method obtains the optimal filter when the equiripple nature of the filter is restricted to have  $n+2$  extremals. When it does not converge, the algorithm switches to an ascent-descent algorithm that takes over to finish the convergence to the optimal solution. See the references for further details.

## References

- [1] Karam, L.J., and J.H. McClellan. "Complex Chebyshev Approximation for FIR Filter Design." *IEEE Trans. on Circuits and Systems II*. March 1995. Pgs. 207-216.
- [2] Karam, L.J. *Design of Complex Digital FIR Filters in the Chebyshev Sense*. Ph.D. Thesis, Georgia Institute of Technology, March 1995.

# cfirpm

---

[3] Demjanjov, V.F., and V.N. Malozemov. *Introduction to Minimax*. New York: John Wiley & Sons, 1974.

## See Also

`fir1`, `fir2`, `firls`, `firpm`, `function_handle`

**Purpose** Chebyshev Type I analog lowpass filter prototype

**Syntax** [z,p,k] = cheb1ap(n,Rp)

**Description** [z,p,k] = cheb1ap(n,Rp) returns the poles and gain of an order n Chebyshev Type I analog lowpass filter prototype with Rp dB of ripple in the passband. The function returns the poles in the length n column vector p and the gain in scalar k. z is an empty matrix, because there are no zeros. The transfer function is

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

Chebyshev Type I filters are equiripple in the passband and monotonic in the stopband. The poles are evenly spaced about an ellipse in the left half plane. The Chebyshev Type I passband edge angular frequency  $\omega_0$  is set to 1.0 for a normalized result. This is the frequency at which the passband ends and the filter has magnitude response of  $10^{-Rp/20}$ .

**References** [1] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Chapter 7.

**See Also** besslap, buttap, cheby1, cheb2ap, ellipap

# cheb1ord

---

**Purpose** Chebyshev Type I filter order

**Syntax**  
`[n,Wp] = cheb1ord(Wp,Ws,Rp,RS)`  
`[n,Wp] = cheb1ord(Wp,Ws,Rp,RS, 's')`

**Description** `cheb1ord` calculates the minimum order of a digital or analog Chebyshev Type I filter required to meet a set of filter design specifications.

## Digital Domain

`[n,Wp] = cheb1ord(Wp,Ws,Rp,RS)` returns the lowest order  $n$  of the Chebyshev Type I filter that loses no more than  $R_p$  dB in the passband and has at least  $R_s$  dB of attenuation in the stopband. The scalar (or vector) of corresponding cutoff frequencies  $W_p$ , is also returned. Use the output arguments  $n$  and  $W_p$  with the `cheby1` function.

Choose the input arguments to specify the stopband and passband according to the following table.

## Description of Stopband and Passband Filter Parameters

Parameter	Description
$W_p$	Passband corner frequency $W_p$ , the cutoff frequency, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, $\pi$ radians per sample.
$W_s$	Stopband corner frequency $W_s$ , is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.
$R_p$	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.
$R_s$	Stopband attenuation, in decibels. This value is the number of decibels the stopband is down from the passband.



Use the following guide to specify filters of different types

**Filter Type Stopband and Passband Specifications**

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	$W_p < W_s$ , both scalars	$(W_s, 1)$	$(0, W_p)$
Highpass	$W_p > W_s$ , both scalars	$(0, W_s)$	$(W_p, 1)$
Bandpass	The interval specified by $W_s$ contains the one specified by $W_p$ $(W_s(1) < W_p(1) < W_p(2) < W_s(2))$ .	$(0, W_s(1))$ and $(W_s(2), 1)$	$(W_p(1), W_p(2))$
Bandstop	The interval specified by $W_p$ contains the one specified by $W_s$ $(W_p(1) < W_s(1) < W_s(2) < W_p(2))$ .	$(0, W_p(1))$ and $(W_p(2), 1)$	$(W_s(1), W_s(2))$

If your filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design separate lowpass and highpass filters according to the specifications in this table, and cascade the two filters together.

**Analog Domain**

$[n, W_p] = \text{cheb1ord}(W_p, W_s, R_p, R_s, 's')$  finds the minimum order  $n$  and cutoff frequencies  $W_p$  for an analog Chebyshev Type I filter. You specify the frequencies  $W_p$  and  $W_s$  similar to those described in the Table , Description of Stopband and Passband Filter Parameters table above, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

Use `cheb1ord` for lowpass, highpass, bandpass, and bandstop filters as described in the Table , Filter Type Stopband and Passband Specifications table above.

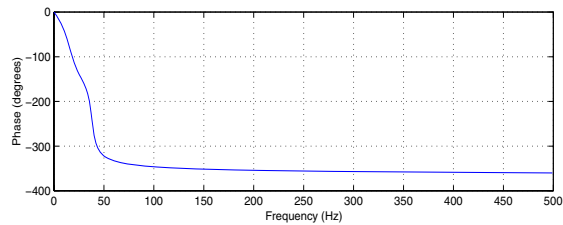
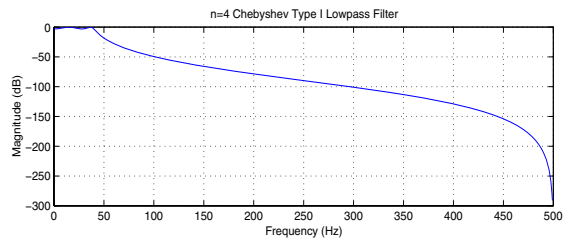
## Examples

For data sampled at 1000 Hz, design a lowpass filter with less than 3 dB of ripple in the passband defined from 0 to 40 Hz and at least 60 dB of ripple in the stopband defined from 150 Hz to the Nyquist frequency (500 Hz):

```
Wp = 40/500; Ws = 150/500;  
Rp = 3; Rs = 60;  
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)
```

```
n =  
    4  
Wp =  
    0.0800
```

```
[b,a] = cheby1(n,Rp,Wp);  
freqz(b,a,512,1000);  
title('n=4 Chebyshev Type I Lowpass Filter')
```



Next design a bandpass filter with a passband of 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband, and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband:

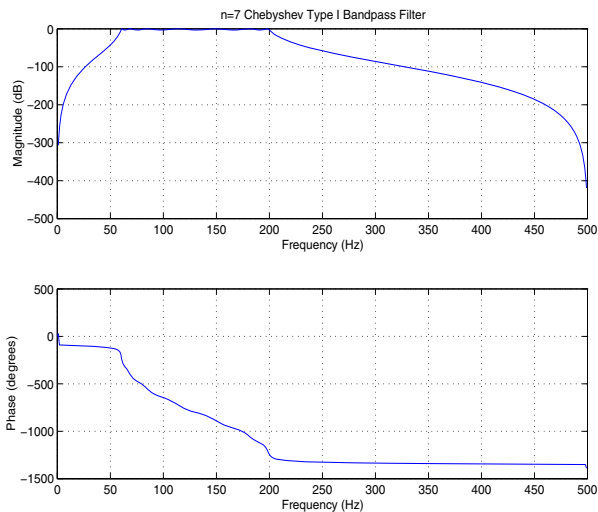
```

Wp = [60 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 40;
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)

n =
    7
Wp =
    0.1200    0.4000

[b,a] = cheby1(n,Rp,Wp);
freqz(b,a,512,1000);
title('n=7 Chebyshev Type I Bandpass Filter')

```



### Algorithm

cheb1ord uses the Chebyshev lowpass filter order prediction formula described in [1]. The function performs its calculations in the analog domain for both analog and digital cases. For the digital case, it converts the frequency parameters to the s-domain before the order and natural frequency estimation process, and then converts them back to the z-domain.

# cheb1ord

---

cheb1ord initially develops a lowpass filter prototype by transforming the passband frequencies of the desired filter to 1 rad/s (for low- or highpass filters) or to -1 and 1 rad/s (for bandpass or bandstop filters). It then computes the minimum order required for a lowpass filter to meet the stopband specification.

## References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 241.

## See Also

buttord, cheby1, cheb2ord, ellipord, kaiserord

**Purpose** Chebyshev Type II analog lowpass filter prototype

**Syntax** `[z,p,k] = cheb2ap(n,Rs)`

**Description** `[z,p,k] = cheb2ap(n,Rs)` finds the zeros, poles, and gain of an order  $n$  Chebyshev Type II analog lowpass filter prototype with stopband ripple  $R_s$  dB down from the passband peak value. `cheb2ap` returns the zeros and poles in length  $n$  column vectors  $z$  and  $p$  and the gain in scalar  $k$ . If  $n$  is odd,  $z$  is length  $n - 1$ . The transfer function is

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

Chebyshev Type II filters are monotonic in the passband and equiripple in the stopband. The pole locations are the inverse of the pole locations of `cheb1ap`, whose poles are evenly spaced about an ellipse in the left half plane. The Chebyshev Type II stopband edge angular frequency  $\omega_0$  is set to 1 for a normalized result. This is the frequency at which the stopband begins and the filter has magnitude response of  $10^{-R_s/20}$ .

**Algorithm** Chebyshev Type II filters are sometimes called *inverse Chebyshev* filters because of their relationship to Chebyshev Type I filters. The `cheb2ap` function is a modification of the Chebyshev Type I prototype algorithm:

- 1 `cheb2ap` replaces the frequency variable  $\omega$  with  $1/\omega$ , turning the lowpass filter into a highpass filter while preserving the performance at  $\omega = 1$ .
- 2 `cheb2ap` subtracts the filter transfer function from unity.

**References** [1] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Chapter 7.

**See Also** `besselap`, `buttap`, `cheb1ap`, `cheby2`, `ellipap`

# cheb2ord

---

**Purpose** Chebyshev Type II filter order

**Syntax** `[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)`  
`[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs, 's')`

**Description** `cheb2ord` calculates the minimum order of a digital or analog Chebyshev Type II filter required to meet a set of filter design specifications.

## Digital Domain

`[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)` returns the lowest order  $n$  of the Chebyshev Type II filter that loses no more than  $R_p$  dB in the passband and has at least  $R_s$  dB of attenuation in the stopband. The scalar (or vector) of corresponding cutoff frequencies  $W_s$ , is also returned. Use the output arguments  $n$  and  $W_s$  in `cheby2`.

Choose the input arguments to specify the stopband and passband according to the following table.

### Description of Stopband and Passband Filter Parameters

Parameter	Description
$W_p$	Passband corner frequency $W_p$ , the cutoff frequency, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, $\pi$ radians per sample.
$W_s$	Stopband corner frequency $W_s$ , is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.
$R_p$	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.
$R_s$	Stopband attenuation, in decibels. This value is the number of decibels the stopband is down from the passband.

Use the following guide to specify filters of different types.

**Filter Type Stopband and Passband Specifications**

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	$W_p < W_s$ , both scalars	$(W_s, 1)$	$(0, W_p)$
Highpass	$W_p > W_s$ , both scalars	$(0, W_s)$	$(W_p, 1)$
Bandpass	The interval specified by $W_s$ contains the one specified by $W_p$ $(W_s(1) < W_p(1) < W_p(2) < W_s(2))$ .	$(0, W_s(1))$ and $(W_s(2), 1)$	$(W_p(1), W_p(2))$
Bandstop	The interval specified by $W_p$ contains the one specified by $W_s$ $(W_p(1) < W_s(1) < W_s(2) < W_p(2))$ .	$(0, W_p(1))$ and $(W_p(2), 1)$	$(W_s(1), W_s(2))$

If your filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design separate lowpass and highpass filters according to the specifications in this table, and cascade the two filters together.

**Analog Domain**

$[n, W_s] = \text{cheb2ord}(W_p, W_s, R_p, R_s, 's')$  finds the minimum order  $n$  and cutoff frequencies  $W_s$  for an analog Chebyshev Type II filter. You specify the frequencies  $W_p$  and  $W_s$  similar to those described in the Table , Description of Stopband and Passband Filter Parameters table above, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

Use `cheb2ord` for lowpass, highpass, bandpass, and bandstop filters as described in the Table , Filter Type Stopband and Passband Specifications table above.

## Examples

### Example 1

For data sampled at 1000 Hz, design a lowpass filter with less than 3 dB of ripple in the passband defined from 0 to 40 Hz, and at least 60 dB of attenuation in the stopband defined from 150 Hz to the Nyquist frequency (500 Hz):

```
Wp = 40/500; Ws = 150/500;
```

```
Rp = 3; Rs = 60;
```

```
[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)
```

```
n =
```

```
4
```

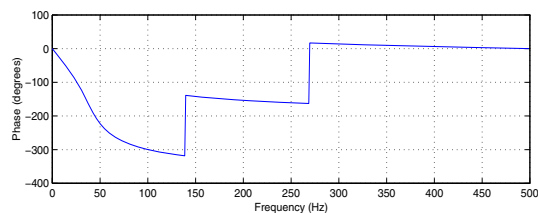
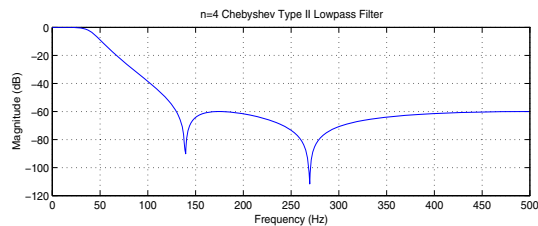
```
Ws =
```

```
0.2597
```

```
[b,a] = cheby2(n,Rs,Ws);
```

```
freqz(b,a,512,1000);
```

```
title('n=4 Chebyshev Type II Lowpass Filter')
```





### Example 2

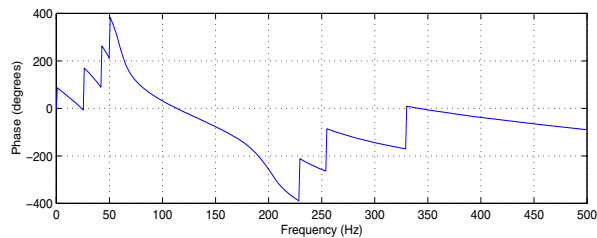
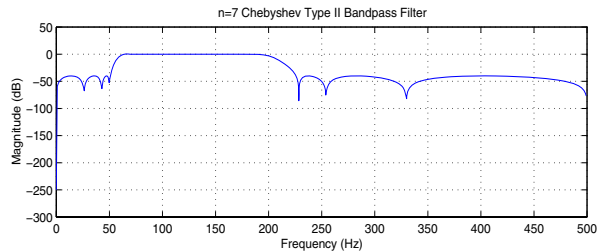
Next design a bandpass filter with a passband of 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband, and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband:

```
Wp = [60 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 40;
[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)
```

```
n =
     7
```

```
Ws =
     0.1019     0.4516
```

```
[b,a] = cheby2(n,Rs,Ws);
freqz(b,a,512,1000)
title('n=7 Chebyshev Type II Bandpass Filter')
```



# cheb2ord

---

## Algorithm

cheb2ord uses the Chebyshev lowpass filter order prediction formula described in [1]. The function performs its calculations in the analog domain for both analog and digital cases. For the digital case, it converts the frequency parameters to the  $s$ -domain before the order and natural frequency estimation process, and then converts them back to the  $z$ -domain.

cheb2ord initially develops a lowpass filter prototype by transforming the stopband frequencies of the desired filter to 1 rad/s (for low- and highpass filters) and to -1 and 1 rad/s (for bandpass and bandstop filters). It then computes the minimum order required for a lowpass filter to meet the passband specification.

## References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 241.

## See Also

buttord, cheb1ord, cheby2, ellipord, kaiserord

**Purpose** Chebyshev window

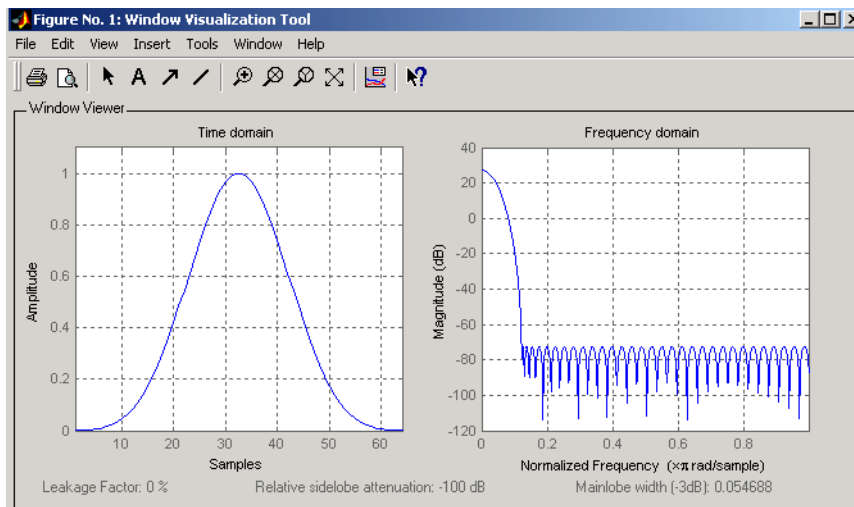
**Syntax** `w = chebwin(L,r)`

**Description** `w = chebwin(L,r)` returns the column vector `w` containing the length `L` Chebyshev window whose Fourier transform sidelobe magnitude is `r` dB below the mainlobe magnitude. The default value for `r` is 100.0 dB.

**Note** If you specify a one-point window (set `L=1`), the value 1 is returned.

**Examples** Create a 64-point Chebyshev window with 100 dB of sidelobe attenuation and display the result using WVTool:

```
L=64;
wvtool(chebwin(L))
```



**Algorithm** An artifact of the equiripple design method used in `chebwin` is the presence of impulses at the endpoints of the time-domain response. This is due to the constant-level sidelobes in the frequency domain. The magnitude of the impulses are on the order of the size of the spectral sidelobes. If the sidelobes

are large, the effect at the endpoints may be significant. For more information on this effect, see reference [2].

## References

[1] IEEE. *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979. Program 5.2.

[2] Harris, Fredric J. *Multirate Signal Processing for Communication Systems*, New Jersey: Prentice Hall PTR, 2004, pp. 60-64.

## See Also

gausswin, kaiser, tukeywin, window, wintool, wvtool

**Purpose** Chebyshev Type I filter design (passband ripple)

**Syntax**

```
[b,a] = cheby1(n,R,Wp)
[b,a] = cheby1(n,R,Wp,'ftype')
[b,a] = cheby1(n,R,Wp,'s')
[b,a] = cheby1(n,R,Wp,'ftype','s')
[z,p,k] = cheby1(...)
[A,B,C,D] = cheby1(...)
```

**Description** cheby1 designs lowpass, bandpass, highpass, and bandstop digital and analog Chebyshev Type I filters. Chebyshev Type I filters are equiripple in the passband and monotonic in the stopband. Type I filters roll off faster than type II filters, but at the expense of greater deviation from unity in the passband.

### Digital Domain

`[b,a] = cheby1(n,R,Wp)` designs an order  $n$  Chebyshev lowpass digital Chebyshev filter with normalized passband edge frequency  $W_p$  and  $R$  dB of peak-to-peak ripple in the passband. It returns the filter coefficients in the length  $n+1$  row vectors  $b$  and  $a$ , with coefficients in descending powers of  $z$ .

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}}{1 + a(2)z^{-1} + \dots + a(n+1)z^{-n}}$$

*Normalized passband edge frequency* is the frequency at which the magnitude response of the filter is equal to  $-R$  dB. For cheby1, the normalized passband edge frequency  $W_p$  is a number between 0 and 1, where 1 corresponds to half the sample rate,  $\pi$  radians per sample. Smaller values of passband ripple  $R$  lead to wider transition widths (shallower rolloff characteristics).

If  $W_p$  is a two-element vector,  $W_p = [w1 \ w2]$ , cheby1 returns an order  $2*n$  bandpass filter with passband  $w1 < \omega < w2$ .

`[b,a] = cheby1(n,R,Wp,'ftype')` designs a highpass, lowpass, or bandstop filter, where the string 'ftype' is one of the following

- 'high' for a highpass digital filter with normalized passband edge frequency  $W_p$

- 'low' for a lowpass digital filter with normalized passband edge frequency  $W_p$
- 'stop' for an order  $2*n$  bandstop digital filter if  $W_p$  is a two-element vector,  $W_p = [w_1 \ w_2]$ . The stopband is  $w_1 < \omega < w_2$ .

With different numbers of output arguments, `cheby1` directly obtains other realizations of the filter. To obtain zero-pole-gain form, use three output arguments as shown below:

`[z,p,k] = cheby1(n,R,Wp)` or

`[z,p,k] = cheby1(n,R,Wp,'ftype')` returns the zeros and poles in length  $n$  column vectors  $z$  and  $p$  and the gain in the scalar  $k$ .

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = cheby1(n,R,Wp)` or

`[A,B,C,D] = cheby1(n,R,Wp,'ftype')` where  $A$ ,  $B$ ,  $C$ , and  $D$  are

$$x[n+1] = Ax[n] + Bu[n]$$

$$y[n] = Cx[n] + Du[n]$$

and  $u$  is the input,  $x$  is the state vector, and  $y$  is the output.

## Analog Domain

`[b,a] = cheby1(n,R,Wp,'s')` designs an order  $n$  lowpass analog Chebyshev Type I filter with angular passband edge frequency  $W_p$  rad/s. It returns the filter coefficients in length  $n+1$  row vectors  $b$  and  $a$ , in descending powers of  $s$ , derived from the transfer function

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \dots + b(n+1)}{s^n + a(2)s^{n-1} + \dots + a(n+1)}$$

*Angular passband edge frequency* is the frequency at which the magnitude response of the filter is  $-R$  dB. For `cheby1`, the angular passband edge frequency  $W_p$  must be greater than 0 rad/s.

If  $Wp$  is a two-element vector  $Wp = [w1 \ w2]$  with  $w1 < w2$ , then `cheby1(n,R,Wp,'s')` returns an order  $2*n$  bandpass analog filter with passband  $w1 < \omega < w2$ .

`[b,a] = cheby1(n,R,Wp,'ftype','s')` designs a highpass, lowpass, or bandstop filter.

You can supply different numbers of output arguments for `cheby1` to directly obtain other realizations of the analog filter. To obtain zero-pole-gain form, use three output arguments as shown below.

`[z,p,k] = cheby1(n,R,Wp,'s')` or

`[z,p,k] = cheby1(n,R,Wp,'ftype','s')` returns the zeros and poles in length  $n$  or  $2*n$  column vectors  $z$  and  $p$  and the gain in the scalar  $k$ .

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = cheby1(n,R,Wp,'s')` or

`[A,B,C,D] = cheby1(n,R,Wp,'ftype','s')` where  $A$ ,  $B$ ,  $C$ , and  $D$  are defined as

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

and  $u$  is the input,  $x$  is the state vector, and  $y$  is the output.

## Examples

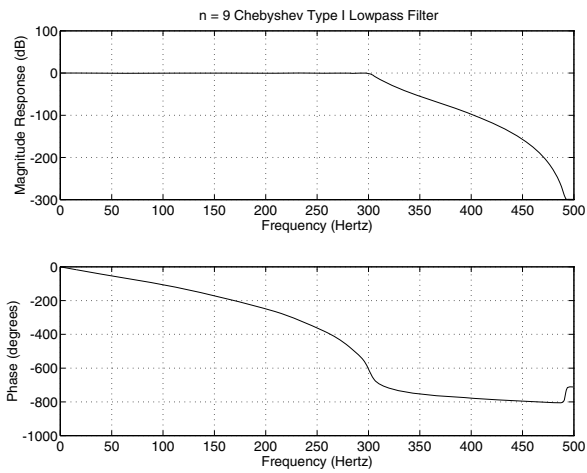
### Example 1: Lowpass Filter

For data sampled at 1000 Hz, design a 9th-order lowpass Chebyshev Type I filter with 0.5 dB of ripple in the passband and a passband edge frequency of 300 Hz, which corresponds to a normalized value of 0.6:

```
[b,a] = cheby1(9,0.5,300/500);
```

The frequency response of the filter is

```
freqz(b,a,512,1000)
```

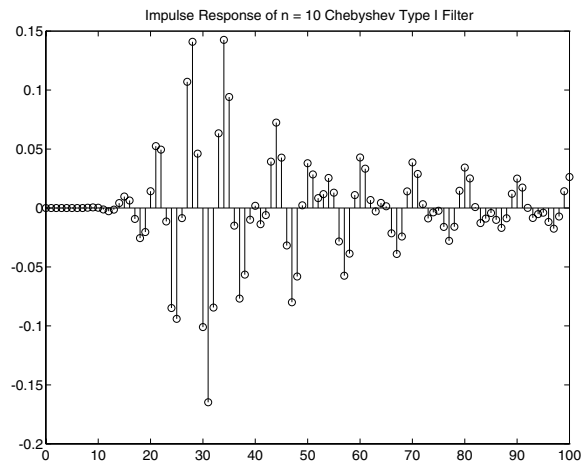


## Example 2: Bandpass Filter

Design a 10th-order bandpass Chebyshev Type I filter with a passband from 100 to 200 Hz and plot its impulse response:

```
n = 10; R = 0.5;  
Wp = [100 200]/500;  
[b,a] = cheby1(n,R,Wp);  
[y,t] = impz(b,a,101); stem(t,y)
```





## Limitations

For high order filters, the state-space form is the most numerically accurate, followed by the zero-pole-gain form. The transfer function form is the least accurate; numerical problems can arise for filter orders as low as 15.

## Algorithm

`cheby1` uses a five-step algorithm:

- 1** It finds the lowpass analog prototype poles, zeros, and gain using the `cheb1ap` function.
- 2** It converts the poles, zeros, and gain into state-space form.
- 3** It transforms the lowpass filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies, using a state-space transformation.
- 4** For digital filter design, `cheby1` uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at  $\omega_p$  or  $\omega_1$  and  $\omega_2$ .
- 5** It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

## See Also

`besself`, `butter`, `cheb1ap`, `cheb1ord`, `cheby2`, `ellip`

# cheby2

---

**Purpose** Chebyshev Type II filter design (stopband ripple)

**Syntax**

```
[b,a] = cheby2(n,R,Wst)
[b,a] = cheby2(n,R,Wst,'ftype')
[b,a] = cheby2(n,R,Wst,'s')
[b,a] = cheby2(n,R,Wst,'ftype','s')
[z,p,k] = cheby2(...)
[A,B,C,D] = cheby2(...)
```

**Description** cheby2 designs lowpass, highpass, bandpass, and bandstop digital and analog Chebyshev Type II filters. Chebyshev Type II filters are monotonic in the passband and equiripple in the stopband. Type II filters do not roll off as fast as type I filters, but are free of passband ripple.

## Digital Domain

`[b,a] = cheby2(n,R,Wst)` designs an order  $n$  lowpass digital Chebyshev Type II filter with normalized stopband edge frequency  $Wst$  and stopband ripple  $R$  dB down from the peak passband value. It returns the filter coefficients in the length  $n+1$  row vectors  $b$  and  $a$ , with coefficients in descending powers of  $z$ .

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}}{1 + a(2)z^{-1} + \dots + a(n+1)z^{-n}}$$

*Normalized stopband edge frequency* is the beginning of the stopband, where the magnitude response of the filter is equal to  $-R$  dB. For cheby2, the normalized stopband edge frequency  $Wst$  is a number between 0 and 1, where 1 corresponds to half the sample rate. Larger values of stopband attenuation  $R$  lead to wider transition widths (shallower rolloff characteristics).

If  $Wst$  is a two-element vector,  $Wst = [w1 \ w2]$ , cheby2 returns an order  $2*n$  bandpass filter with passband  $w1 < \omega < w2$ .

`[b,a] = cheby2(n,R,Wst,'ftype')` designs a highpass, lowpass, or bandstop filter, where the string `'ftype'` is one of the following.

- `'high'` for a highpass digital filter with normalized stopband edge frequency  $Wst$

- 'low' for a lowpass digital filter with normalized stopband edge frequency  $Wst$
- 'stop' for an order  $2*n$  bandstop digital filter if  $Wst$  is a two-element vector,  $Wst = [w1 \ w2]$ . The stopband is  $w1 < \omega < w2$ .

With different numbers of output arguments, cheby2 directly obtains other realizations of the filter. To obtain zero-pole-gain form, use three output arguments as shown below:

`[z,p,k] = cheby2(n,R,Wst)` or

`[z,p,k] = cheby2(n,R,Wst,'ftype')` returns the zeros and poles in length  $n$  column vectors  $z$  and  $p$  and the gain in the scalar  $k$ .

To obtain state-space form, use four output arguments as shown below.

`[A,B,C,D] = cheby2(n,R,Wst)` or

`[A,B,C,D] = cheby2(n,R,Wst,'ftype')` where  $A$ ,  $B$ ,  $C$ , and  $D$  are

$$\begin{aligned}x[n+1] &= Ax[n] + Bu[n] \\y[n] &= Cx[n] + Du[n]\end{aligned}$$

and  $u$  is the input,  $x$  is the state vector, and  $y$  is the output.

### Analog Domain

`[b,a] = cheby2(n,R,Wst,'s')` designs an order  $n$  lowpass analog Chebyshev Type II filter with angular stopband edge frequency  $Wst$  rad/s. It returns the filter coefficients in the length  $n+1$  row vectors  $b$  and  $a$ , with coefficients in descending powers of  $s$ , derived from the transfer function.

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \dots + b(n+1)}{s^n + a(2)s^{n-1} + \dots + a(n+1)}$$

*Angular stopband edge frequency* is the frequency at which the magnitude response of the filter is equal to  $-R$  dB. For cheby2, the angular stopband edge frequency  $Wst$  must be greater than 0 rad/s.

# cheby2

---

If  $Wst$  is a two-element vector  $Wst = [w1 \ w2]$  with  $w1 < w2$ , then `cheby2(n,R,Wst,'s')` returns an order  $2*n$  bandpass analog filter with passband  $w1 < \omega < w2$ .

`[b,a] = cheby2(n,R,Wst,'ftype','s')` designs a highpass, lowpass, or bandstop analog filter.

With different numbers of output arguments, `cheby2` directly obtains other realizations of the analog filter. To obtain zero-pole-gain form, use three output arguments as shown below:

`[z,p,k] = cheby2(n,R,Wst,'s')` or

`[z,p,k] = cheby2(n,R,Wst,'ftype','s')` returns the zeros and poles in length  $n$  or  $2*n$  column vectors  $z$  and  $p$  and the gain in the scalar  $k$ .

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = cheby2(n,R,Wst,'s')` or

`[A,B,C,D] = cheby2(n,R,Wst,'ftype','s')` where  $A$ ,  $B$ ,  $C$ , and  $D$  are

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

and  $u$  is the input,  $x$  is the state vector, and  $y$  is the output.

## Examples

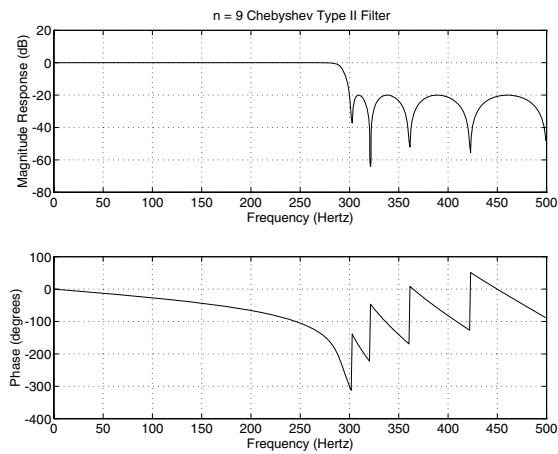
### Example 1: Lowpass Filter

For data sampled at 1000 Hz, design a ninth-order lowpass Chebyshev Type II filter with stopband attenuation 20 dB down from the passband and a stopband edge frequency of 300 Hz, which corresponds to a normalized value of 0.6:

```
[b,a] = cheby2(9,20,300/500);
```

The frequency response of the filter is

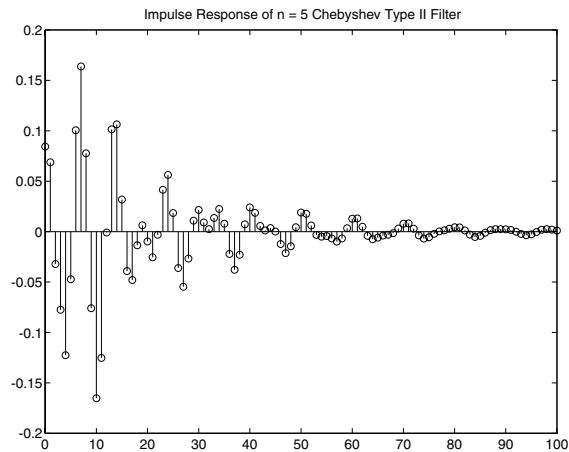
```
freqz(b,a,512,1000)
```



## Example 2: Bandpass Filter

Design a fifth-order bandpass Chebyshev Type II filter with passband from 100 to 200 Hz and plot the impulse response of the filter:

```
n = 5; r = 20;
Wst = [100 200]/500;
[b,a] = cheby2(n,r,Wst);
[y,t] = impz(b,a,101); stem(t,y)
```



# cheby2

---

## Limitations

For high order filters, the state-space form is the most numerically accurate, followed by the zero-pole-gain form. The transfer function coefficient form is the least accurate; numerical problems can arise for filter orders as low as 15.

## Algorithm

`cheby2` uses a five-step algorithm:

- 1** It finds the lowpass analog prototype poles, zeros, and gain using the `cheb2ap` function.
- 2** It converts poles, zeros, and gain into state-space form.
- 3** It transforms the lowpass filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies, using a state-space transformation.
- 4** For digital filter design, `cheby2` uses `biilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at  $\omega_{st}$  or  $\omega_1$  and  $\omega_2$ .
- 5** It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

## See Also

`besself`, `butter`, `cheb2ap`, `cheb1ord`, `cheby1`, `ellip`

**Purpose**

Swept-frequency cosine

**Syntax**

```
y = chirp(t,f0,t1,f1)
y = chirp(t,f0,t1,f1,'method')
y = chirp(t,f0,t1,f1,'method',phi)
y = chirp(t,f0,t1,f1,'quadratic',phi,'shape')
```

**Description**

`y = chirp(t,f0,t1,f1)` generates samples of a linear swept-frequency cosine signal at the time instances defined in array `t`, where `f0` is the instantaneous frequency at time 0, and `f1` is the instantaneous frequency at time `t1`. `f0` and `f1` are both in hertz. If unspecified, `f0` is  $e^{-6}$  for logarithmic chirp and 0 for all other methods, `t1` is 1, and `f1` is 100.

`y = chirp(t,f0,t1,f1,'method')` specifies alternative sweep method options, where `method` can be:

- `linear`, which specifies an instantaneous frequency sweep  $f_i(t)$  given by

$$f_i(t) = f_0 + \beta t$$

where

$$\beta = (f_1 - f_0)/t_1$$

and the default value for  $f_0$  is 0.  $\beta$  ensures that the desired frequency breakpoint  $f_1$  at time  $t_1$  is maintained.

- `quadratic`, which specifies an instantaneous frequency sweep  $f_i(t)$  given by

$$f_i(t) = f_0 + \beta t^2$$

where

$$\beta = (f_1 - f_0)/t_1^2$$

and the default value for  $f_0$  is 0. If  $f_0 > f_1$  (downsweep), the default shape is convex. If  $f_0 < f_1$  (upsweep), the default shape is concave.

- logarithmic specifies an instantaneous frequency sweep  $f_i(t)$  given by

$$f_i(t) = f_0 \times \beta^t$$

where

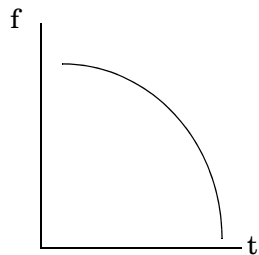
$$\beta = \left(\frac{f_1}{f_0}\right)^{\frac{1}{t_1}}$$

and the default value for  $f_0$  is  $1e^{-6}$ . Both an upsweep ( $f_1 > f_0$ ) and a downsweep ( $f_0 > f_1$ ) of frequency is possible.

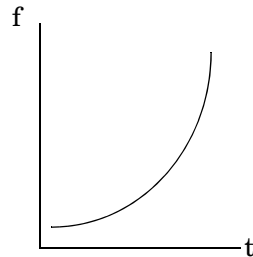
Each of the above methods can be entered as 'li', 'q', and 'lo', respectively.

`y = chirp(t, f0, t1, f1, 'method', phi)` allows an initial phase `phi` to be specified in degrees. If unspecified, `phi` is 0. Default values are substituted for empty or omitted trailing input arguments.

`y = chirp(t, f0, t1, f1, 'quadratic', phi, 'shape')` specifies the shape of the quadratic swept-frequency signal's spectrogram. `shape` is either concave or convex, which describes the shape of the parabola in the positive frequency axis. If `shape` is omitted, the default is convex for downsweep ( $f_0 > f_1$ ) and is concave for upsweep ( $f_0 < f_1$ ).



**Convex downsweep  
shape**



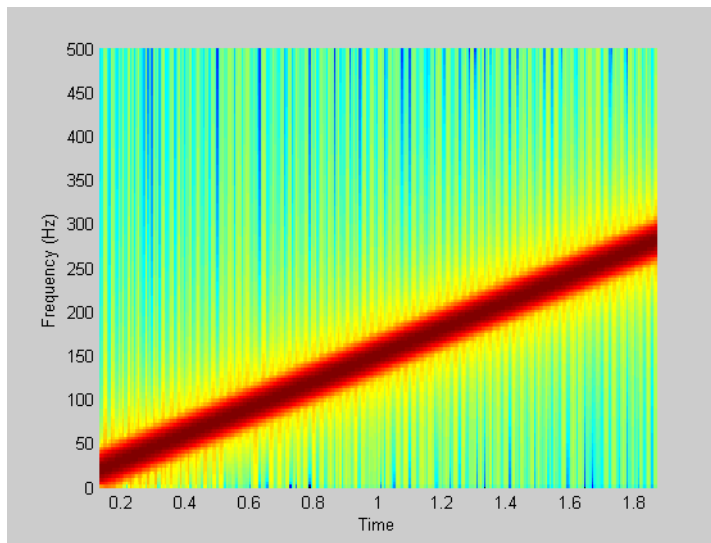
**Concave upsweep  
shape**



**Examples****Example 1**

Compute the spectrogram of a chirp with linear instantaneous frequency deviation:

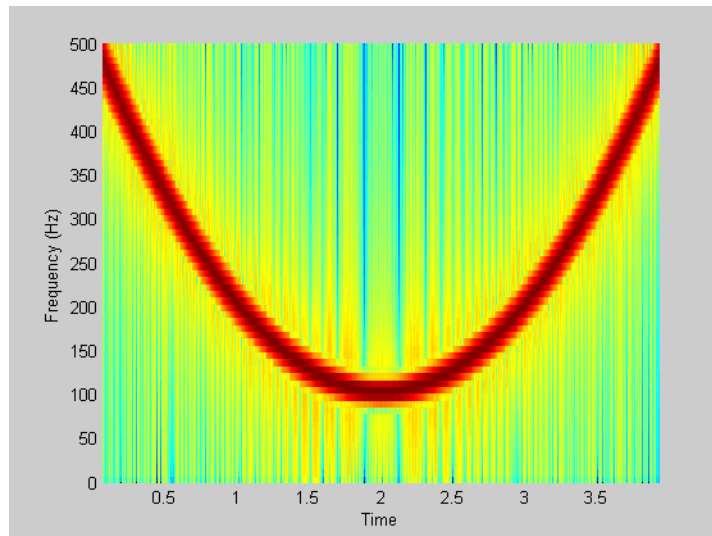
```
t = 0:0.001:2;           % 2 secs @ 1kHz sample rate
y = chirp(t,0,1,150);    % Start @ DC, cross 150Hz at t=1 sec
spectrogram(y,256,250,256,1E3,'yaxis')
```



## Example 2

Compute the spectrogram of a chirp with quadratic instantaneous frequency deviation:

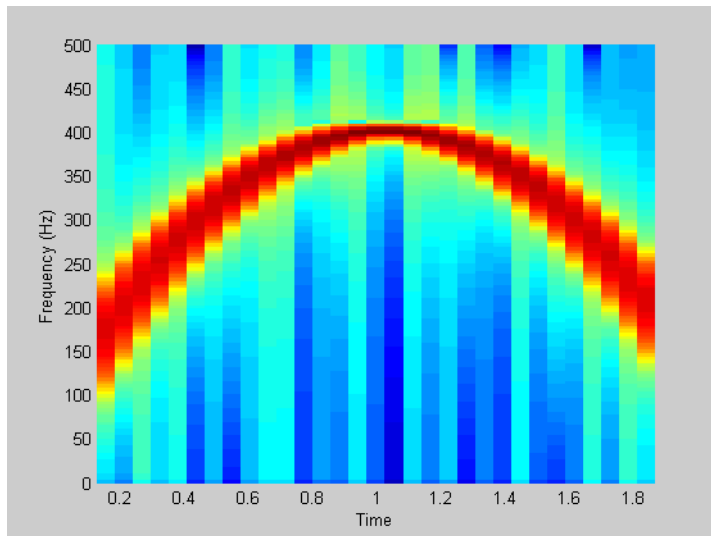
```
t = -2:0.001:2; % ±2 secs @ 1kHz sample rate
y = chirp(t,100,1,200,'quadratic'); % Start @ 100Hz, cross 200Hz
% at t=1 sec
spectrogram(y,128,120,128,1E3,'yaxis')
```



### Example 3

Compute the spectrogram of a convex quadratic chirp:

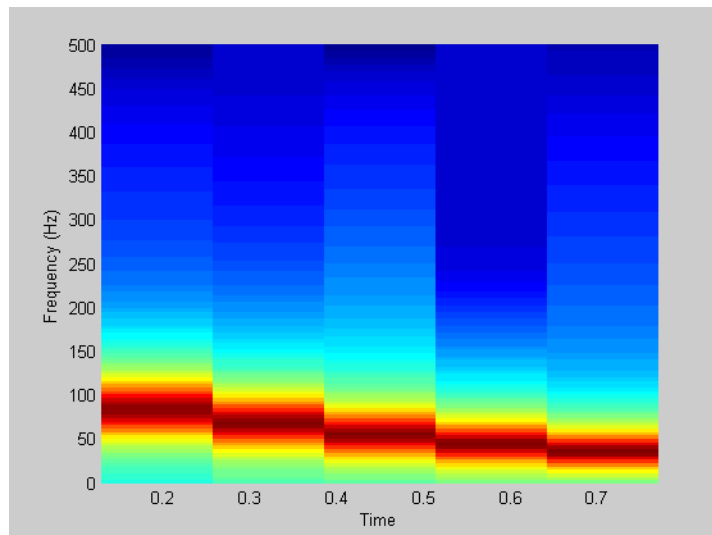
```
t = -1:0.001:1;           % +/-1 second @ 1kHz sample rate
fo = 100; f1 = 400;       % Start at 100Hz, go up to 400Hz
y = chirp(t,fo,1,f1,'q',[],'convex');
spectrogram(y,256,200,256,1000,'yaxis')
```



## Example 4

Compute the spectrogram of a concave quadratic chirp:

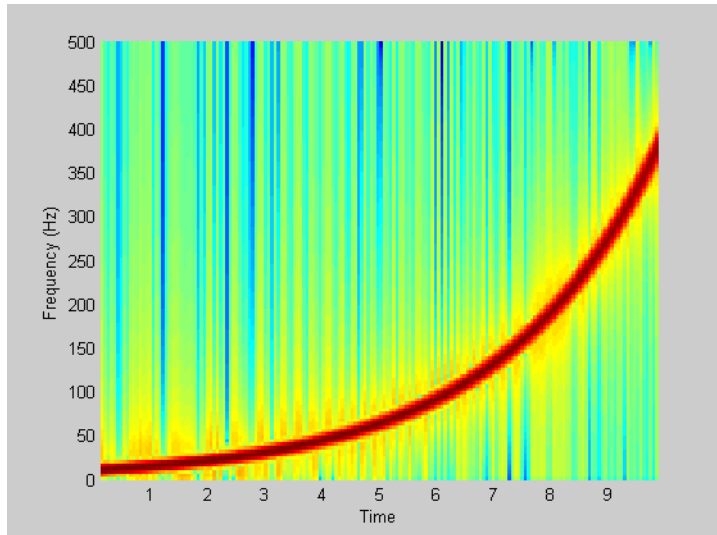
```
t = 0:0.001:1; % 1 second @ 1kHz sample rate
fo = 100; f1 = 25; % Start at 100Hz, go down to 25Hz
y = chirp(t,fo,1,f1,'q',[],'concave');
spectrogram(y,hanning(256),128,256,1000,'yaxis')
```



## Example 5

Compute the spectrogram of a logarithmic chirp:

```
t = 0:0.001:10; % 10 seconds @ 1kHz sample rate
fo = 10; f1 = 400; % Start at 10Hz, go up to 400Hz
y = chirp(t,fo,10,f1,'logarithmic');
spectrogram(y,256,200,256,1000,'yaxis')
```



**See Also**

cos, diric, gauspuls, pulstran, rectpuls, sawtooth, sin, sinc, square, tripuls

# conv

---

**Purpose** Convolution and polynomial multiplication

**Description** conv is a MATLAB function.

**Purpose** Two-dimensional convolution

**Description** conv2 is a MATLAB function.

# convmtx

---

**Purpose** Convolution matrix

**Syntax**  
`A = convmtx(c,n)`  
`A = convmtx(r,n)`

**Description** A *convolution matrix* is a matrix, formed from a vector, whose inner product with another vector is the convolution of the two vectors.

`A = convmtx(c,n)` where `c` is a length `m` column vector returns a matrix `A` of size  $(m+n-1)$ -by-`n`. The product of `A` and another column vector `x` of length `n` is the convolution of `c` with `x`.

`A = convmtx(r,n)` where `r` is a length `m` row vector returns a matrix `A` of size `n`-by- $(m+n-1)$ . The product of `A` and another row vector `x` of length `n` is the convolution of `r` with `x`.

**Examples** Generate a simple convolution matrix:

```
h = [1 2 3 2 1];
convmtx(h,7)
ans =

     1     2     3     2     1     0     0     0     0     0     0
     0     1     2     3     2     1     0     0     0     0     0
     0     0     1     2     3     2     1     0     0     0     0
     0     0     0     1     2     3     2     1     0     0     0
     0     0     0     0     1     2     3     2     1     0     0
     0     0     0     0     0     1     2     3     2     1     0
     0     0     0     0     0     0     1     2     3     2     1
```

Note that `convmtx` handles edge conditions by zero padding.

In practice, it is more efficient to compute convolution using

```
y = conv(c,x)
```

than by using a convolution matrix.

```
n = length(x);
y = convmtx(c,n)*x
```

**Algorithm** `convmtx` uses the function `toeplitz` to generate the convolution matrix.



**See Also**

conv, convn, conv2, dftmtx

# corrcoef

---

**Purpose** Correlation coefficient matrix

**Description** corrcoef is a MATLAB function.

**Purpose**

Data matrix for autocorrelation matrix estimation

**Syntax**

```
X = corrmtx(x,m)
X = corrmtx(x,m,'method')
[X,R] = corrmtx(...)
```

**Description**

`X = corrmtx(x,m)` returns an  $(n+m)$ -by- $(m+1)$  rectangular Toeplitz matrix  $X$ , such that  $X'X$  is a (biased) estimate of the autocorrelation matrix for the length  $n$  data vector  $x$ .

`X = corrmtx(x,m,'method')` computes the matrix  $X$  according to the method specified by the string *'method'*:

- *'autocorrelation'*: (default)  $X$  is the  $(n+m)$ -by- $(m+1)$  rectangular Toeplitz matrix that generates an autocorrelation estimate for the length  $n$  data vector  $x$ , derived using *prewindowed* and *postwindowed* data, based on an  $m$ th order prediction error model.
- *'prewindowed'*:  $X$  is the  $n$ -by- $(m+1)$  rectangular Toeplitz matrix that generates an autocorrelation estimate for the length  $n$  data vector  $x$ , derived using *prewindowed* data, based on an  $m$ th order prediction error model.
- *'postwindowed'*:  $X$  is the  $n$ -by- $(m+1)$  rectangular Toeplitz matrix that generates an autocorrelation estimate for the length  $n$  data vector  $x$ , derived using *postwindowed* data, based on an  $m$ th order prediction error model.
- *'covariance'*:  $X$  is the  $(n-m)$ -by- $(m+1)$  rectangular Toeplitz matrix that generates an autocorrelation estimate for the length  $n$  data vector  $x$ , derived using *nonwindowed* data, based on an  $m$ th order prediction error model.
- *'modified'*:  $X$  is the  $2(n-m)$ -by- $(m+1)$  modified rectangular Toeplitz matrix that generates an autocorrelation estimate for the length  $n$  data vector  $x$ , derived using forward and backward prediction error estimates, based on an  $m$ th order prediction error model.

`[X,R] = corrmtx(...)` also returns the  $(m+1)$ -by- $(m+1)$  autocorrelation matrix estimate  $R$ , calculated as  $X'X$ .

**Examples**

```
randn('state',1); n=0:99;
s=exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);
X=corrmtx(s,12,'mod');
```

## Algorithm

The Toeplitz data matrix computed by `corrmtx` depends on the method you select. The matrix determined by the autocorrelation (default) method is given by the following matrix.

$$X = \begin{array}{c} \left[ \begin{array}{ccc} x(1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ x(m+1) & \cdots & x(1) \\ \vdots & \ddots & \vdots \\ x(n-m) & \cdots & x(m+1) \\ \vdots & \ddots & \vdots \\ x(n) & \cdots & x(n-m) \\ \vdots & \ddots & \vdots \\ 0 & \cdots & x(n) \end{array} \right] \end{array}$$

In this matrix,  $m$  is the same as the input argument  $m$  to `corrmtx`, and  $n$  is `length(x)`. Variations of this matrix are used to return the output  $X$  of `corrmtx` for each method:

- 'autocorrelation': (default)  $X = X$ , above.
- 'prewindowed':  $X$  is the  $n$ -by- $(m+1)$  submatrix of  $X$  that is given by the portion of  $X$  above the lower gray line.
- 'postwindowed':  $X$  is the  $n$ -by- $(m+1)$  submatrix of  $X$  that is given by the portion of  $X$  below the upper gray line.
- 'covariance':  $X$  is the  $(n-m)$ -by- $(m+1)$  submatrix of  $X$  that is given by the portion of  $X$  between the two gray lines.
- 'modified':  $X$  is the  $2(n-m)$ -by- $(m+1)$  matrix  $X_{\text{mod}}$  shown below.

$$X_{\text{mod}} = \begin{bmatrix} x(m+1) & \cdots & x(1) \\ \vdots & \ddots & \vdots \\ x(n-m) & \cdots & x(m+1) \\ \vdots & \ddots & \vdots \\ x(n) & \cdots & x(n-m) \\ x^*(1) & \cdots & x^*(m+1) \\ \vdots & \ddots & \vdots \\ x^*(m+1) & \cdots & x^*(n-m) \\ \vdots & \ddots & \vdots \\ x^*(n-m) & \cdots & x^*(n) \end{bmatrix}$$

**References**

[1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, pp. 216-223.

**See Also**

peig, pmusic, rooteig, rootmusic, xcorr

## **COV**

---

**Purpose** Covariance matrix

**Description** cov is a MATLAB function.

**Purpose**            Group complex numbers into complex conjugate pairs

**Description**        `cplxpair` is a MATLAB function.

# cpsd

---

## Purpose

Cross power spectral density

## Syntax

```
Pxy = cpsd(x,y)
Pxy = cpsd(x,y,window)
Pxy = cpsd(x,y,window,noverlap)
[Pxy,W] = cpsd(x,y,window,noverlap,nfft)
[Pxy,F] = cpsd(x,y,window,noverlap,nfft,fs)
[...] = cpsd(...,'twosided')
cpsd(x,y,...)
```

## Description

`Pxy = cpsd(x,y)` estimates the cross power spectral density  $P_{xy}$  of the discrete-time signals  $x$  and  $y$  using the Welch's averaged, modified periodogram method of spectral estimation. The cross power spectral density is the distribution of power per unit frequency.

For real  $x$  and  $y$ , `cpsd` returns a one-sided CPSD and for complex  $x$  or  $y$ , it returns a two-sided CPSD.



`cpsd` uses the following default values:

Parameter	Description	Default Value
<code>nfft</code>	<p>FFT length which determines the frequencies at which the PSD is estimated</p> <p>For real <math>x</math> and <math>y</math>, the length of <math>P_{xy}</math> is <math>(nfft/2+1)</math> if <math>nfft</math> is even or <math>(nfft+1)/2</math> if <math>nfft</math> is odd. For complex <math>x</math> or <math>y</math>, the length of <math>P_{xy}</math> is <math>nfft</math>.</p> <p>If <math>nfft</math> is greater than the signal length, the data is zero-padded. If <math>nfft</math> is less than the signal length, the segment is wrapped using <code>datawrap</code> so that the length is equal to <math>nfft</math>.</p>	Maximum of 256 or the next power of 2 greater than the length of each section of $x$ or $y$
<code>fs</code>	Sampling frequency	1
<code>window</code>	Windowing function and number of samples to use for each section	Periodic Hamming window of length to obtain eight equal sections of $x$ and $y$
<code>noverlap</code>	Number of samples by which the sections overlap	Value to obtain 50% overlap

**Note** You can use the empty matrix `[]` to specify the default value for any input argument except  $x$  or  $y$ . For example, `Pxy = cpsd(x,y,[],[],128)` uses a Hamming window, default `noverlap` to obtain 50% overlap, and the specified 128 `nfft`.

`Pxy = cpsd(x,y>window)` specifies a windowing function, divides `x` and `y` into overlapping sections of the specified window length, and windows each section using the specified window function. If you supply a scalar for `window`, `Pxy` uses a Hamming window of that length. The `x` and `y` vectors are divided into eight equal sections of that length. If the signal cannot be sectioned evenly with 50% overlap, it is truncated.

`Pxy = cpsd(x,y>window,noverlap)` overlaps the sections of `x` by `noverlap` samples. `noverlap` must be an integer smaller than the length of `window`.

`[Pxy,W] = cpsd(x,y>window,noverlap,nfft)` uses the specified FFT length `nfft` in estimating the CPSD. It also returns `W`, which is the vector of normalized frequencies (in rad/sample) at which the CPSD is estimated. For real signals, the range of `W` is `[0, pi]` when `nfft` is even and `[0, pi)` when `nfft` is odd. For complex signals, the range of `W` is `[0, 2*pi)`.

`[Pxy,F] = cpsd(x,y>window,noverlap,nfft,fs)` returns `Pxy` as a function of frequency and a vector `F` of frequencies at which the CPSD is estimated. `fs` is the sampling frequency in Hz. For real signals, the range of `F` is `[0, fs/2]` when `nfft` is even and `[0, fs/2)` when `nfft` is odd. For complex signals, the range of `F` is `[0, fs)`.

`[...] = cpsd(...,'twosided')` returns the two-sided CPSD of real signals `x` and `y`. The length of the resulting `Pxy` is `nfft` and its range is `[0, 2*pi)` if you do not specify `fs`. If you specify `fs`, the range is `[0,fs)`. Entering `'onesided'` for a real signal produces the default. You can place the `'onesided'` or `'twosided'` string in any position after the `noverlap` parameter.

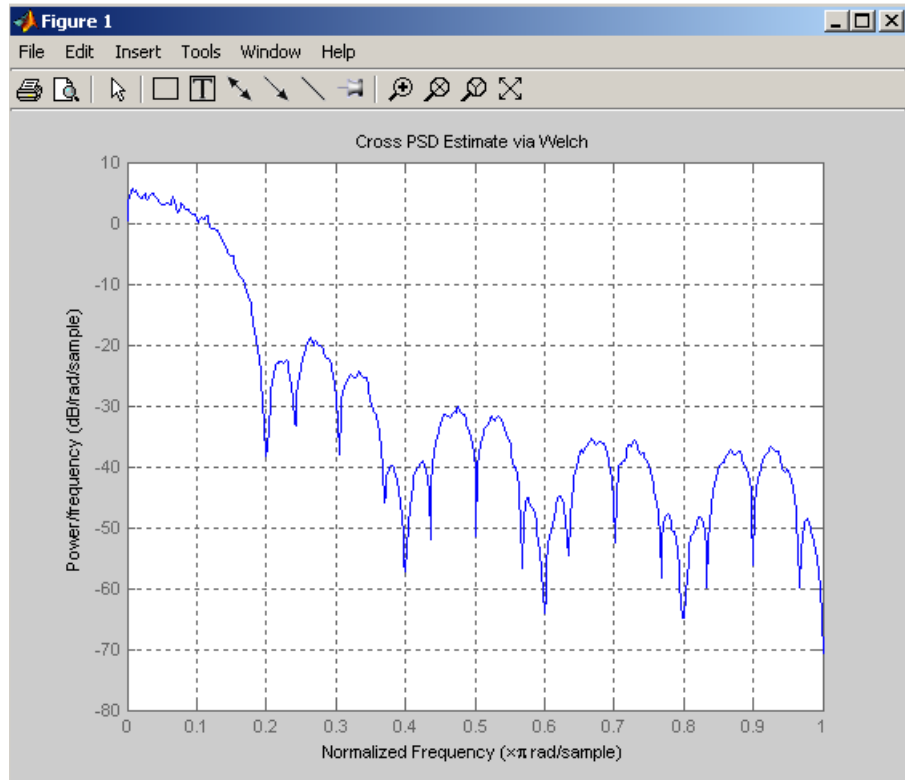
`cpsd(...)` plots the CPSD versus frequency in the current figure window.

## Examples

Generate two colored noise signals and plot their CPSD with a confidence interval of 95%. Specify a length 1024 FFT, a 500 point triangular window with no overlap, and a sampling frequency of 10 Hz:

```
randn('state',0);  
h = fir1(30,0.2,rectwin(31));  
h1 = ones(1,10)/sqrt(10);  
r = randn(16384,1);  
x = filter(h1,1,r);  
y = filter(h,1,x);
```

```
cpsd(x,y,triang(500),250,1024)
```



## Algorithm

cpsd uses Welch's averaged periodogram method. See the references listed below.

## References

- [1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pgs. 414-419.
- [2] Welch, P.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 (June 1967). Pgs. 70-73.

[3] Oppenheim, A.V., and R.W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 737.

## **See Also**

dspdata, mscohere, pburg, pcov, peig, periodogram, pmcov, pmtm, pmusic, pwelch, pyulear, spectrum, tfestimate

**Purpose**Chirp  $z$ -transform**Syntax**

```
y = czt(x,m,w,a)
y = czt(x)
```

**Description**

`y = czt(x,m,w,a)` returns the chirp  $z$ -transform of signal  $x$ . The chirp  $z$ -transform is the  $z$ -transform of  $x$  along a spiral contour defined by  $w$  and  $a$ .  $m$  is a scalar that specifies the length of the transform,  $w$  is the ratio between points along the  $z$ -plane spiral contour of interest, and scalar  $a$  is the complex starting point on that contour. The contour, a spiral or “chirp” in the  $z$ -plane, is given by

$$z = a \cdot (w.^{-(0:m-1)})$$

`y = czt(x)` uses the following default values:

- $m = \text{length}(x)$
- $w = \exp(-j \cdot 2 \cdot \pi / m)$
- $a = 1$

With these defaults, `czt` returns the  $z$ -transform of  $x$  at  $m$  equally spaced points around the unit circle. This is equivalent to the discrete Fourier transform of  $x$ , or `fft(x)`. The empty matrix `[]` specifies the default value for a parameter.

If  $x$  is a matrix, `czt(x,m,w,a)` transforms the columns of  $x$ .

**Examples**

Create a random vector  $x$  of length 1013 and compute its DFT using `czt`:

```
randn('state',0);
x = randn(1013,1);
y = czt(x);
```

Use `czt` to zoom in on a narrow-band section (100 to 150 Hz) of a filter’s frequency response. First design the filter:

```
h = fir1(30,125/500,rectwin(31)); % filter
```

Establish frequency and CZT parameters:

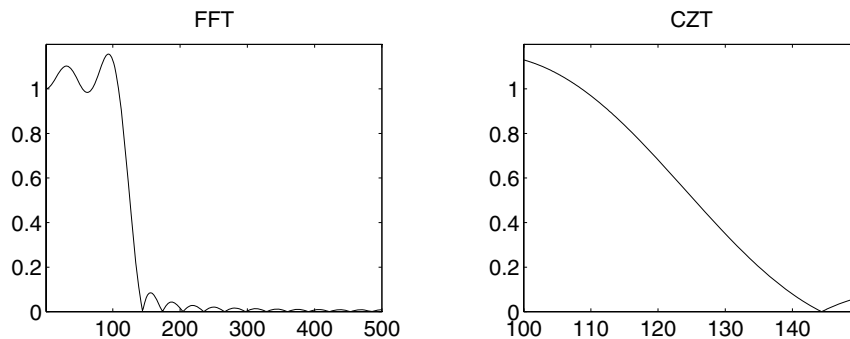
```
fs = 1000; f1 = 100; f2 = 150; % in hertz
m = 1024;
w = exp(-j*2*pi*(f2-f1)/(m*fs));
a = exp(j*2*pi*f1/fs);
```

Compute both the DFT and CZT of the filter:

```
y = fft(h,1000);
z = czf(h,m,w,a);
```

Create frequency vectors and compare the results:

```
fy = (0:length(y)-1)'*1000/length(y);
fz = ((0:length(z)-1)'*(f2-f1)/length(z)) + f1;
plot(fy(1:500),abs(y(1:500))); axis([1 500 0 1.2])
title('FFT')
figure
plot(fz,abs(z)); axis([f1 f2 0 1.2])
title('CZT')
```



## Algorithm

czf uses the next power-of-2 length FFT to perform a fast convolution when computing the  $z$ -transform on a specified chirp contour [1].

## Diagnostics

If  $m$ ,  $w$ , or  $a$  is not a scalar, czf gives the following error message:

```
Inputs M, W, and A must be scalars.
```

**References**

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pgs. 393-399.

**See Also**

fft, freqz

# dct

---

**Purpose** Discrete transform (DCT)

**Syntax**  
 $y = \text{dct}(x)$   
 $y = \text{dct}(x, n)$

**Description**  $y = \text{dct}(x)$  returns the unitary discrete cosine transform of  $x$

$$y(k) = w(k) \sum_{n=1}^N x(n) \cos \frac{\pi(2n-1)(k-1)}{2N}, \quad k = 1, \dots, N$$

where

$$w(k) = \begin{cases} \frac{1}{\sqrt{N}}, & k = 1 \\ \sqrt{\frac{2}{N}}, & 2 \leq k \leq N \end{cases}$$

$N$  is the length of  $x$ , and  $x$  and  $y$  are the same size. If  $x$  is a matrix, `dct` transforms its columns. The series is indexed from  $n = 1$  and  $k = 1$  instead of the usual  $n = 0$  and  $k = 0$  because MATLAB vectors run from 1 to  $N$  instead of from 0 to  $N-1$ .

$y = \text{dct}(x, n)$  pads or truncates  $x$  to length  $n$  before transforming.

The DCT is closely related to the discrete Fourier transform. You can often reconstruct a sequence very accurately from only a few DCT coefficients, a useful property for applications requiring data reduction.



**Examples**

Find how many DCT coefficients represent 99% of the energy in a sequence:

```
x = (1:100) + 50*cos((1:100)*2*pi/40);
X = dct(x);
[XX,ind] = sort(abs(X)); ind = fliplr(ind);
i = 1;
while (norm([X(ind(1:i)) zeros(1,100-i)])/norm(X)<.99)
    i = i + 1;
end

i =
    3
```

**References**

- [1] Jain, A.K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
- [2] Pennebaker, W.B., and J.L. Mitchell. *JPEG Still Image Data Compression Standard*. New York, NY: Van Nostrand Reinhold, 1993. Chapter 4.

**See Also**

fft, idct, dct2, idct2

# decimate

---

**Purpose** Decimation — decrease sampling rate

**Syntax**

```
y = decimate(x,r)
y = decimate(x,r,n)
y = decimate(x,r,'fir')
y = decimate(x,r,n,'fir')
```

**Description** Decimation reduces the original sampling rate for a sequence to a lower rate, the opposite of interpolation. The decimation process filters the input data with a lowpass filter and then resamples the resulting smoothed signal at a lower rate.

`y = decimate(x,r)` reduces the sample rate of `x` by a factor `r`. The decimated vector `y` is `r` times shorter in length than the input vector `x`. By default, `decimate` employs an eighth-order lowpass Chebyshev Type I filter with a cutoff frequency of  $0.8*(F_s/2)/r$ . It filters the input sequence in both the forward and reverse directions to remove all phase distortion, effectively doubling the filter order.

`y = decimate(x,r,n)` uses an order `n` Chebyshev filter. Orders above 13 are not recommended because of numerical instability. MATLAB displays a warning in this case.

---

**Note** For better results when `r` is greater than 13, you should break `r` into its factors and call `decimate` several times.

---

`y = decimate(x,r,'fir')` uses an order 30 FIR filter, instead of the Chebyshev IIR filter. Here `decimate` filters the input sequence in only one direction. This technique conserves memory and is useful for working with long sequences.

`y = decimate(x,r,n,'fir')` uses an order `n` FIR filter.

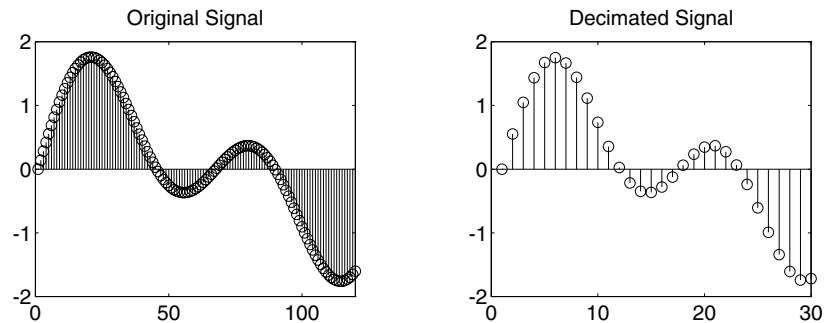
**Examples** Decimate a signal by a factor of four:

```
t = 0:.00025:1; % Time vector
x = sin(2*pi*30*t) + sin(2*pi*60*t);
```

```
y = decimate(x,4);
```

View the original and decimated signals:

```
stem(x(1:120)), axis([0 120 -2 2]) % Original signal
title('Original Signal')
figure
stem(y(1:30)) % Decimated signal
title('Decimated Signal')
```



## Algorithm

decimate uses decimation algorithms 8.2 and 8.3 from [1]:

- 1** It designs a lowpass filter. By default, decimate uses a Chebyshev Type I filter with normalized cutoff frequency  $0.8/r$  and 0.05 dB of passband ripple. For the `fir` option, decimate designs a lowpass FIR filter with cutoff frequency  $1/r$  using `fir1`.
- 2** For the FIR filter, decimate applies the filter to the input vector in one direction. In the IIR case, decimate applies the filter in forward and reverse directions with `filtfilt`.
- 3** decimate resamples the filtered data by selecting every  $r$ th point.

## Diagnostics

If  $r$  is not an integer, decimate gives the following error message:

```
Resampling rate R must be an integer.
```

If  $n$  specifies an IIR filter with order greater than 13, decimate gives the following warning:

```
Warning: IIR filters above order 13 may be unreliable.
```

# decimate

---

## References

[1] IEEE. *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979. Chapter 8.

## See Also

`cheby1`, `downsample`, `filtfilt`, `fir1`, `mfilt`, `interp`, `resample`

**Purpose** Deconvolution and polynomial division

**Description** deconv is a MATLAB function.

# demod

---

**Purpose** Demodulation for communications simulation

**Syntax**

```
x = demod(y,fc,fs,'method')
x = demod(y,fc,fs,'method',opt)
x = demod(y,fc,fs,'pwm','centered')
[x1,x2] = demod(y,fc,fs,'qam')
```

**Description** demod performs demodulation, that is, it obtains the original signal from a modulated version of the signal. demod undoes the operation performed by modulate.

`x = demod(y,fc,fs,'method')` and

`x = demod(y,fc,fs,'method',opt)` demodulate the real carrier signal `y` with a carrier frequency `fc` and sampling frequency `fs`, using one of the options listed below for `method`. (Note that some methods accept an option, `opt`.)

<b>Method</b>	<b>Description</b>
---------------	--------------------

<code>amdsb-sc</code> or <code>am</code>	<b>Amplitude demodulation, double sideband, suppressed carrier.</b> Multiplies <code>y</code> by a sinusoid of frequency <code>fc</code> and applies a fifth-order Butterworth lowpass filter using <code>filtfilt</code> .
--	---

```
x = y.*cos(2*pi*fc*t);
[b,a] = butter(5,fc*2/fs);
x = filtfilt(b,a,x);
```

<code>amdsb-tc</code>	<b>Amplitude demodulation, double sideband, transmitted carrier.</b> Multiplies <code>y</code> by a sinusoid of frequency <code>fc</code> , and applies a fifth-order Butterworth lowpass filter using <code>filtfilt</code> .
-----------------------	--

```
x = y.*cos(2*pi*fc*t);
[b,a] = butter(5,fc*2/fs);
x = filtfilt(b,a,x);
```

If you specify `opt`, demod subtracts scalar `opt` from `x`. The default value for `opt` is 0.

- amssb** **Amplitude demodulation, single sideband.** Multiplies  $y$  by a sinusoid of frequency  $fc$  and applies a fifth-order Butterworth lowpass filter using `filtfilt`.
- ```
x = y.*cos(2*pi*fc*t);
[b,a] = butter(5,fc*2/fs);
x = filtfilt(b,a,x);
```
- fm** **Frequency demodulation.** Demodulates the FM waveform by modulating the Hilbert transform of  $y$  by a complex exponential of frequency  $-fc$  Hz and obtains the instantaneous frequency of the result.
- pm** **Phase demodulation.** Demodulates the PM waveform by modulating the Hilbert transform of  $y$  by a complex exponential of frequency  $-fc$  Hz and obtains the instantaneous phase of the result.
- ppm** **Pulse-position demodulation.** Finds the pulse positions of a pulse-position modulated signal  $y$ . For correct demodulation, the pulses cannot overlap.  $x$  is `length(t)*fc/fs`.
- pwm** **Pulse-width demodulation.** Finds the pulse widths of a pulse-width modulated signal  $y$ . `demod` returns in  $x$  a vector whose elements specify the width of each pulse in fractions of a period. The pulses in  $y$  should start at the beginning of each carrier period, that is, they should be left justified.
- qam** **Quadrature amplitude demodulation.**  
`[x1,x2] = demod(y,fc,fs,'qam')` multiplies  $y$  by a cosine and a sine of frequency  $fc$  and applies a fifth-order Butterworth lowpass filter using `filtfilt`.
- ```
x1 = y.*cos(2*pi*fc*t);
x2 = y.*sin(2*pi*fc*t);
[b,a] = butter(5,fc*2/fs);
x1 = filtfilt(b,a,x1);
x2 = filtfilt(b,a,x2);
```

The default method is 'am'. In all cases except 'ppm' and 'pwm',  $x$  is the same size as  $y$ .

If  $y$  is a matrix, `demod` demodulates its columns.

# demod

---

`x = demod(y,fc,fs,'pwm','centered')` finds the pulse widths assuming they are centered at the beginning of each period. `x` is `length(y)*fc/fs`.

## See Also

`modulate`, `vco`, `fskdemod`, `genqamdemod`, `mskdemod`, `pamdemod`, `pmdemod`, `qamdemod`



**Purpose** Discrete-time filter

**Syntax**  
`Hd = dfilt.structure(input1,...)`  
`Hd = [dfilt.structure(input1,...),dfilt.structure(input1,...),...]`

**Description** `Hd = dfilt.structure(input1,...)` returns a discrete-time filter, `Hd`, of type *structure*. Each structure takes one or more inputs. If you specify a `dfilt.structure` with no inputs, a default filter is created.

---

**Note** You must use a *structure* with `dfilt`.

---

`Hd = [dfilt.structure(input1,...),dfilt.structure(input1,...),...]` returns a vector containing `dfilt` filters.

### Structures

Structures for `dfilt` specify the type of filter structure. Available types of structures for `dfilt` are shown below.

<b>dfilt.structure</b>	<b>Description</b>
<code>dfilt.delay</code>	Delay
<code>dfilt.df1</code>	Direct-form I
<code>dfilt.df1sos</code>	Direct-form I, second-order sections
<code>dfilt.df1t</code>	Direct-form I transposed
<code>dfilt.df1tsos</code>	Direct-form I transposed, second-order sections
<code>dfilt.df2</code>	Direct-form II
<code>dfilt.df2sos</code>	Direct-form II, second-order sections
<code>dfilt.df2t</code>	Direct-form II transposed
<code>dfilt.df2tsos</code>	Direct-form II transposed, second-order sections

# dfilt

<b>dfilt.structure</b>	<b>Description</b>
dfilt.dffir	Direct-form FIR
dfilt.dffirt	Direct-form FIR transposed
dfilt.dfsymfir	Direct-form symmetric FIR
dfilt.dfasymfir	Direct-form antisymmetric FIR
dfilt.fftfir	Overlap-add FIR
dfilt.latticeallpass	Lattice allpass
dfilt.latticear	Lattice autoregressive (AR)
dfilt.latticearma	Lattice autoregressive moving- average (ARMA)
dfilt.latticemamax	Lattice moving-average (MA) for maximum phase
dfilt.latticemamin	Lattice moving-average (MA) for minimum phase
dfilt.calattice	Coupled, allpass lattice (available only with the Filter Design Toolbox)
dfilt.calatticepc	Coupled, allpass lattice with power complementary output (available only with the Filter Design Toolbox)
dfilt.statespace	State-space
dfilt.scalar	Scalar gain object
dfilt.cascade	Filters arranged in series
dfilt.parallel	Filters arranged in parallel

For more information on each structure, use the syntax help `dfilt.structure` at the MATLAB prompt or refer to its reference page.

## Methods

Methods provide ways of performing functions directly on your `dfilt` object without having to specify the filter parameters again. You can apply these methods directly on the variable you assigned to your `dfilt` object.

For example, if you create a `dfilt` object, `Hd`, you can check whether it has linear phase with `islinphase(Hd)`, view its frequency response plot with `fvtool(Hd)`, or obtain its frequency response values with `h=freqz(Hd)`. You can use all of the methods below in this way.

---

**Note** If your variable is a 1-D array of `dfilt` filters, the method is applied to each object in the array. Only `freqz`, `grpdelay`, `impz`, `is*`, `order`, and `stepz` methods can be applied to arrays. The `zplane` method can be applied to an array only if it is used without outputs.

---

Some of the methods listed below have the same name as functions in the Signal Processing Toolbox and they behave similarly. This is called *overloading* of functions.

<b>Method</b>	<b>Description</b>
addstage	Adds a stage to a cascade or parallel object, where a stage is a separate, modular filter. See <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .
block	<p>(Available only with the Signal Processing Blockset)</p> <p><code>block(Hd)</code> creates a Signal Processing Blockset block of the <code>dfilt</code> object. The <code>block</code> method can specify these properties/values:</p> <ul style="list-style-type: none"><li>'Destination' indicates where to place the block.</li><li>'Current' places the block in the current Simulink model. 'New' creates a new model. Default value is 'Current'.</li><li>'Blockname' assigns the entered string to the block name. Default name is 'Filter'.</li><li>'OverwriteBlock' indicates whether to overwrite the block generated by the <code>block</code> method ('on') and defined by <code>Blockname</code>. Default is 'off'.</li><li>'MapStates' specifies initial conditions in the block ('on'). Default is 'off'. See “Using Filter States” on page 7-147.</li></ul>
cascade	Returns the series combination of two <code>dfilt</code> objects. See <code>dfilt.cascade</code> .
coeffs	Returns the filter coefficients in a structure containing fields that use the same property names as those in the original <code>dfilt</code> .
convert	Converts a <code>dfilt</code> object from one filter structure, to another filter structure

<b>Method</b>	<b>Description</b>
<code>fcfwrite</code>	<p>Writes a filter coefficient ASCII file. The file can contain a single filter or a vector of objects. If the Filter Design Toolbox is installed, the file can contain multirate filters (<code>mfilt</code>) or adaptive filters (<code>adaptfilt</code>). Default filename is <code>untitled.fcf</code>.</p> <p><code>fcfwrite(Hd,filename)</code> writes to a disk file named <code>filename</code> in the current working directory. The <code>.fcf</code> extension is added automatically.</p> <p><code>fcfwrite(...,fmt)</code> writes the coefficients in the format <code>fmt</code>, where valid <code>fmt</code> strings are:  '<code>hex</code>' for hexadecimal  '<code>dec</code>' for decimal  '<code>bin</code>' for binary representation.</p>
<code>fftcoeffs</code>	Returns the frequency-domain coefficients used when filtering with a <code>dfilt.fftfir</code>
<code>filter</code>	Performs filtering using the <code>dfilt</code> object
<code>firtype</code>	Returns the type (1-4) of a linear phase FIR filter
<code>freqz</code>	Plots the frequency response in <code>fvtool</code> . Note that unlike the <code>freqz</code> function, this <code>dfilt</code> <code>freqz</code> method has a default length of 8192.
<code>grpdelay</code>	Plots the group delay in <code>fvtool</code>
<code>impz</code>	Plots the impulse response in <code>fvtool</code>
<code>impzlength</code>	Returns the length of the impulse response
<code>info</code>	Displays <code>dfilt</code> information, such as filter structure, length, stability, linear phase, and, when appropriate, lattice and ladder length. For multistage filters ( <code>cascade</code> and <code>parallel</code> ), use <code>info(Hd.Stage(x))</code> , where <code>x</code> is the stage number, to see information about that stage.

<b>Method</b>	<b>Description</b>
<code>isallpass</code>	Returns a logical 1 (i.e., true) if the <code>dfilt</code> object in an allpass filter or a logical 0 (i.e., false) if it is not
<code>iscascade</code>	Returns a logical 1 if the <code>dfilt</code> object is cascaded or a logical 0 if it is not
<code>isfir</code>	Returns a logical 1 if the <code>dfilt</code> object has finite impulse response (FIR) or a logical 0 if it does not
<code>islinphase</code>	Returns a logical 1 if the <code>dfilt</code> object is linear phase or a logical 0 if it is not
<code>ismaxphase</code>	Returns a logical 1 if the <code>dfilt</code> object is maximum-phase or a logical 0 if it is not
<code>isminphase</code>	Returns a logical 1 if the <code>dfilt</code> object is minimum-phase or a logical 0 if it is not
<code>isparallel</code>	Returns a logical 1 if the <code>dfilt</code> object has parallel stages or a logical 0 if it does not
<code>isreal</code>	Returns a logical 1 if the <code>dfilt</code> object has real-valued coefficients or a logical 0 if it does not
<code>isscalar</code>	Returns a logical 1 if the <code>dfilt</code> object is a scalar or a logical 0 if it is not scalar
<code>issos</code>	Returns a logical 1 if the <code>dfilt</code> object has second-order sections or a logical 0 if it does not
<code>isstable</code>	Returns a logical 1 if the <code>dfilt</code> object is stable or a logical 0 if it are not
<code>nsections</code>	Returns the number of sections in a second-order sections filter. If a multistage filter contains stages with multiple sections, using <code>nsections</code> returns the total number of sections in all the stages (a stage with a single section returns 1).
<code>nstages</code>	Returns the number of stages of the filter, where a stage is a separate, modular filter

---

<b>Method</b>	<b>Description</b>
nstates	Returns the number of states for an object
order	Returns the filter order. If Hd is a single-stage filter, the order is given by the number of delays needed for a minimum realization of the filter. If Hd has multiple stages, the order is given by the number of delays needed for a minimum realization of the overall filter.
parallel	Returns the parallel combination of two dfilt filters. See dfilt.parallel.
phasez	Plots the phase response in fvtool

Method	Description
realizemd1	<p>(Available only with Simulink)</p> <p>realizemd1(Hd) creates a Simulink model containing a subsystem block realization of your dfilt.</p> <p>realizemd1(Hd,p1,v1,p2,v2,...) creates the block using the properties p1, p2,... and values v1, v2,... specified.</p> <p>The following properties are available:</p> <p>'Blockname' specifies the name of the block. The default value is 'Filter'.</p> <p>'Destination' specifies whether to add the block to a current Simulink model or create a new model. Valid values are 'Current' and 'New'.</p> <p>'OverwriteBlock' specifies whether to overwrite an existing block that was created by realizemd1 or create a new block. Valid values are 'on' and 'off'. Note that only blocks created by realizemd1 are overwritten.</p> <p>The following properties optimize the block structure. Specifying 'on' turns the optimization on and 'off' creates the block without optimization. The default for each block is 'off'.</p> <p>'OptimizeZeros' removes zero-gain blocks.</p> <p>'OptimizeOnes' replaces unity-gain blocks with a direct connection.</p> <p>'OptimizeNegOnes' replaces negative unity-gain blocks with a sign change at the nearest summation block.</p> <p>'OptimizeDelayChains' replaces cascaded chains of delay block with a single integer delay block set to the appropriate delay.</p>



Method	Description
<code>removestage</code>	Removes a stage from a cascade or parallel <code>dfilt</code> . See <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .
<code>setstage</code>	Overwrites a stage of a cascade or parallel <code>dfilt</code> . See <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .
<code>sos</code>	<p>Converts the <code>dfilt</code> to a second-order sections <code>dfilt</code>. If <code>Hd</code> has a single section, the returned filter has the same class.</p> <p><code>sos(Hd, flag)</code> specifies the ordering of the second-order sections. If <code>flag= 'up'</code>, the first row contains the poles closest to the origin, and the last row contains the poles closest to the unit circle. If <code>flag= 'down'</code>, the sections are ordered in the opposite direction. The zeros are always paired with the poles closest to them.</p> <p><code>sos(Hd, flag, scale)</code> specifies the scaling of the gain and the numerator coefficients of all second-order sections. <code>scale</code> can be <code>'none'</code>, <code>'inf'</code> (infinity-norm) or <code>'two'</code> (2-norm). Using infinity-norm scaling with up ordering minimizes the probability of overflow in the realization. Using 2-norm scaling with down ordering minimizes the peak roundoff noise.</p>
<code>ss</code>	Converts the <code>dfilt</code> to state-space. To see the separate <code>A, B, C, D</code> matrices for the state-space model, use <code>[A, B, C, D]=ss(Hd)</code> .

Method	Description
stepz	Plots the step response in fvtool  stepz(Hd,n) computes the first n samples of the step response.  stepz(Hd,n,Fs) separates the time samples by $T = 1/Fs$ , where Fs is assumed to be in Hz.
tf	Converts the dfilt to a transfer function
zerophase	Plots the zero-phase response in fvtool
zpk	Converts the dfilt to zeros-pole-gain form
zplane	Plots a pole-zero plot in fvtool

For more information on each method, use the syntax `help dfilt/method` at the MATLAB prompt.

## Viewing Properties

As with any object, you can use `get` to view a `dfilt` properties. To see a specific property, use

```
get(Hd, 'property')
```

To see all properties for an object, use

```
get(Hd)
```

---

**Note** If you have the Filter Design Toolbox, an arithmetic property is displayed. You can change the internal arithmetic of the filter from double-precision to single-precision using:  
`Hd.arithmetic = 'single'`

If you have both the Filter Design Toolbox and the Fixed-Point Toolbox, you

can change the arithmetic property to fixed-point using:

```
Hd.arithmetic = 'fixed'
```

---

### Changing Properties

To set specific properties, use

```
set(Hd, 'property1', value, 'property2', value, ...)
```

Note that you must use single quotation marks around the property name.

### Copying an Object

To create a copy of an object, use the copy method.

```
H2 = copy(Hd)
```

---

**Note** Using the syntax `H2 = Hd` copies only the object handle and does not create a new object.

---

### Converting Between Filter Structures

To change the filter structure of a `dfilt` object `Hd`, use

```
Hd2=convert(Hd, 'structure_string');
```

where `structure_string` is any valid structure name in single quotation marks. If `Hd` is a cascade or parallel structure, each of its stages is converted to the new structure.

### Using Filter States

Two properties control the filter states:

- `states`—stores the current states of the filter. Before the filter is applied, the states correspond to the initial conditions and after the filter is applied, the states correspond to the final conditions. For `df1`, `df1t`, `df1sos` and `df1tsos` structures, `states` returns a `filtstates` object.
- `PersistentMemory`—controls whether filter states are saved. The default value is `'false'`, which causes the initial conditions to be reset to zero before

filtering and turns off the display of states information. Setting `PersistentMemory` to `'true'` allows the filter to use your initial conditions or to reuse the final conditions of a previous filtering operation as the initial conditions of the next filtering operation. It also displays information about the filter states.

---

**Note** If you set the states and want to use them for filtering, you must set `PersistentMemory` to `'true'` before you use the filter.

---

## Examples

Create a direct-form I filter and use a method to see if it is stable.

```
[b,a] = butter(8,0.25);
Hd = dfilt.df1(b,a)

Hd =
    FilterStructure: 'Direct-Form I'
      Numerator: [1x9 double]
      Denominator: [1x9 double]
 PersistentMemory: false

isstable(Hd)
ans =
    1
```

If a `dfilt`'s numerator values do not fit on a single line, a description of the vector is displayed. To see the specific numerator values for this example, use

```
get(Hd, 'numerator')

ans =
Columns 1 through 6
    0.0001    0.0009    0.0030    0.0060    0.0076    0.0060
Columns 7 through 9
    0.0030    0.0009    0.0001
```

Create an array containing two `dfilt` objects, apply a method and verify that the method acts on both objects, and use a method to test whether the objects are FIR objects.

```

b = fir1(5,.5);
Hd = dfilt.dffir(b);           % create an FIR object
[b,a] = butter(5,.5);
Hd(2) = dfilt.df2t(b,a);     % create a DF2T object and place
                             % it in the second column of Hd

[h,w] = freqz(Hd);
size(h)                       % verify that resulting h is
ans =                          % 2 columns
      8192                    2
size(w)                       % verify that resulting w is
ans =                          % 1 column
      8192                    1

test_fir = isfir(Hd)
test_fir =
      1      0                % Hd(1) is FIR and Hd(2) is not

```

Refer to the reference pages for each structure for more examples.

### See Also

dfilt.cascade, dfilt.df1, dfilt.df1t, dfilt.df2, dfilt.df2t, dfilt.dfasymfir, dfilt.dffir, dfilt.dffirt, dfilt.dfsymfir, dfilt.latticeallpass, dfilt.latticear, dfilt.latticearma, dfilt.latticemamax, dfilt.latticemamin, dfilt.parallel, dfilt.statespace, filter, freqz, grpdelay, impz, sos, step, tf, zpk, zplane

# dfilt.cascade

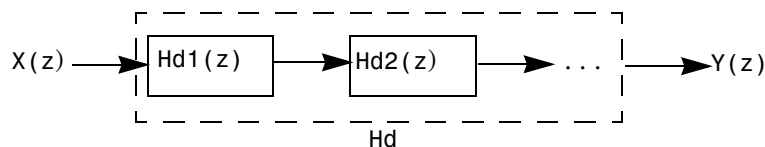
**Purpose** Cascade of discrete-time filters

**Syntax** `Hd = dfilt.cascade(Hd1,Hd2,...)`

**Description** `Hd = dfilt.cascade(Hd1,Hd2,...)` returns a discrete-time filter, `Hd`, of type `cascade`, which is a serial interconnection of two or more `dfilt` filters, `Hd1`, `Hd2`, and so on. Each filter in a cascade is a separate stage.

You can also use the `nondot` notation format for calling a cascade:

`cascade(Hd1,Hd2,...)`



## Examples

Cascade a lowpass filter and a highpass filter to produce a bandpass filter:

```
[b1,a1]=butter(8,0.6);           % Lowpass
[b2,a2]=butter(8,0.4,'high');    % Highpass
H1=dfilt.df2t(b1,a1);
H2=dfilt.df2t(b2,a2);
Hcas=dfilt.cascade(H1,H2)        % Bandpass with passband 0.4-0.6

Hcas =
    FilterStructure: Cascade
           Stage(1): Direct-Form II Transposed
           Stage(2): Direct-Form II Transposed
 PersistentMemory: false
```

To view details of the first stage, use

```
info(Hcas.Stage(1))
```

```
Discrete-Time IIR Filter (real)
```

```
-----
```

```
Filter Structure      : Direct-Form II Transposed
```

```
Numerator Length     : 9
```

```
Denominator Length   : 9
```

```
Stable                : Yes
```

```
Linear Phase         : No
```

To view the states of a stage, use

```
Hcas.stage(1).states
```

You can display states for individual stages only.

## See Also

`dfilt`, `dfilt.parallel`, `dfilt.scalar`

# dfilt.delay

---

**Purpose** Delay filter

**Syntax** Hd = dfilt.delay  
Hd = dfilt.delay(latency)

**Description** Hd = dfilt.delay returns a discrete-time filter, Hd, of type delay, which adds a single delay to any signal filtered with Hd. The filtered signal has its values shifted by one sample.

Hd = dfilt.delay(latency) returns a discrete-time filter, Hd, of type delay, which adds the number of delay units specified in latency to any signal filtered with Hd. The filtered signal has its values shifted by the latency number of samples. The values that appear before the shifted signal are the filter states.

**Examples** Create a delay filter with a latency of 4 and filter a simple signal to view the impact of applying a delay.

```
h = dfilt.delay(4)
h =
    FilterStructure: 'Delay'
           Latency: 4
 PersistentMemory: false

sig = 1:7                                % Create some simple signal data
sig =
     1     2     3     4     5     6     7

states = h.states                          % Filter states before filtering
states =
     0
     0
     0
     0

filter(h,sig)                              % Filter using the delay filter
ans =
     0     0     0     0     1     2     3
```



```
states=h.states           % Filter states after filtering
states =
    4
    5
    6
    7
```

## See Also

dfilt

# dfilt.df1

**Purpose** Discrete-time, direct-form I filter

**Syntax**  
`Hd = dfilt.df1(b,a)`  
`Hd = dfilt.df1`

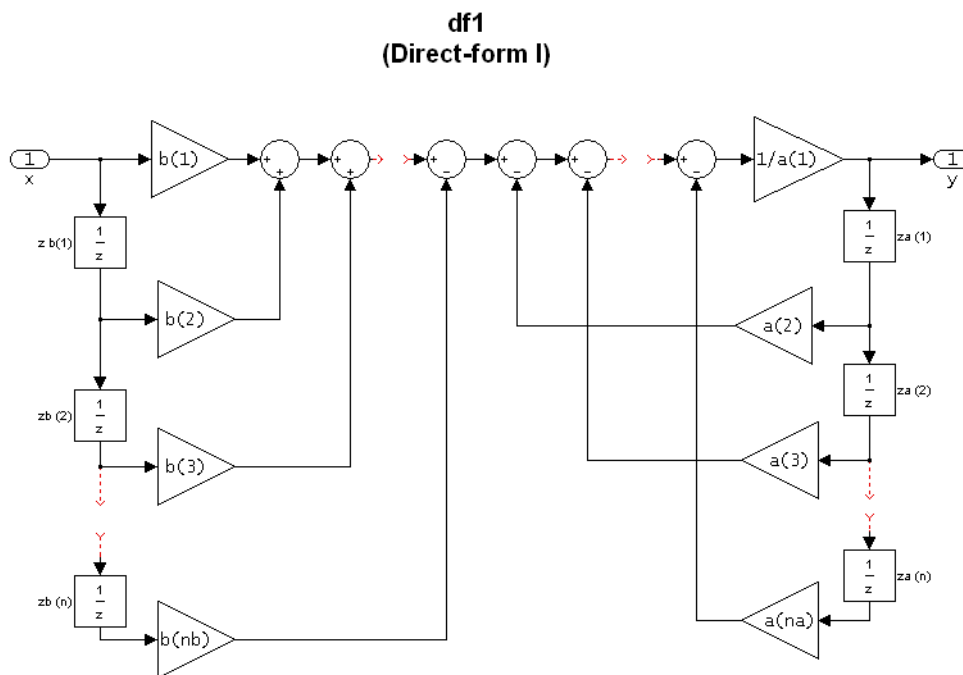
**Description** `Hd = dfilt.df1(b,a)` returns a discrete-time, direct-form I filter, `Hd`, with numerator coefficients `b` and denominator coefficients `a`. The filter states for this object are stored in a `filtstates` object.

`Hd = dfilt.df1` returns a default, discrete-time, direct-form I filter, `Hd`, with `b=1` and `a=1`. This filter passes the input through to the output unchanged.

---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0.

---



To display the filter states, use this code to access the `filtstates` object.

```
Hs = Hd.states           % Where Hd is the dfilt.df1 object and
double (Hs)             % Hs is the filtstates object
```

The vector is

$$\begin{bmatrix} zb(1) \\ zb(2) \\ \dots \\ zb(n) \\ za(1) \\ za(2) \\ \dots \\ za(n) \end{bmatrix}$$

## Examples

Create a direct-form I discrete-time filter with coefficients from a fourth-order lowpass Butterworth design

```
[b,a] = butter(4,.5);
Hd = dfilt.df1(b,a)

FilterStructure: 'Direct-Form I'
Numerator: [0.0940 0.3759 0.5639 0.3759 0.0940]
Denominator: [1 -3.6082e-016 0.4860 3.6545e-017 0.0177]
PersistentMemory: false
```

## See Also

`dfilt`, `dfilt.df1t`, `dfilt.df2`, `dfilt.df2t`

# dfilt.df1sos

---

**Purpose** Discrete-time, second-order section, direct-form I filter

**Syntax**

```
Hd = dfilt.df1sos(s)
Hd = dfilt.df1sos(b1,a1,b2,a2,...)
Hd = dfilt.df1sos(...,g)
```

**Description** Hd = dfilt.df1sos(s) returns a discrete-time, second-order section, direct-form I filter, Hd, with coefficients given in the s matrix. The filter states for this object are stored in a filtstates object.

Hd = dfilt.df1sos(b1,a1,b2,a2,...) returns a discrete-time, second-order section, direct-form I filter, Hd, with coefficients for the first section given in the b1 and a1 vectors, for the second section given in the b2 and a2 vectors, etc.

Hd = dfilt.df1sos(...,g) includes a gain vector g. The elements of g are the gains for each section. The maximum length of g is the number of sections plus one. If g is not specified, all gains default to one.

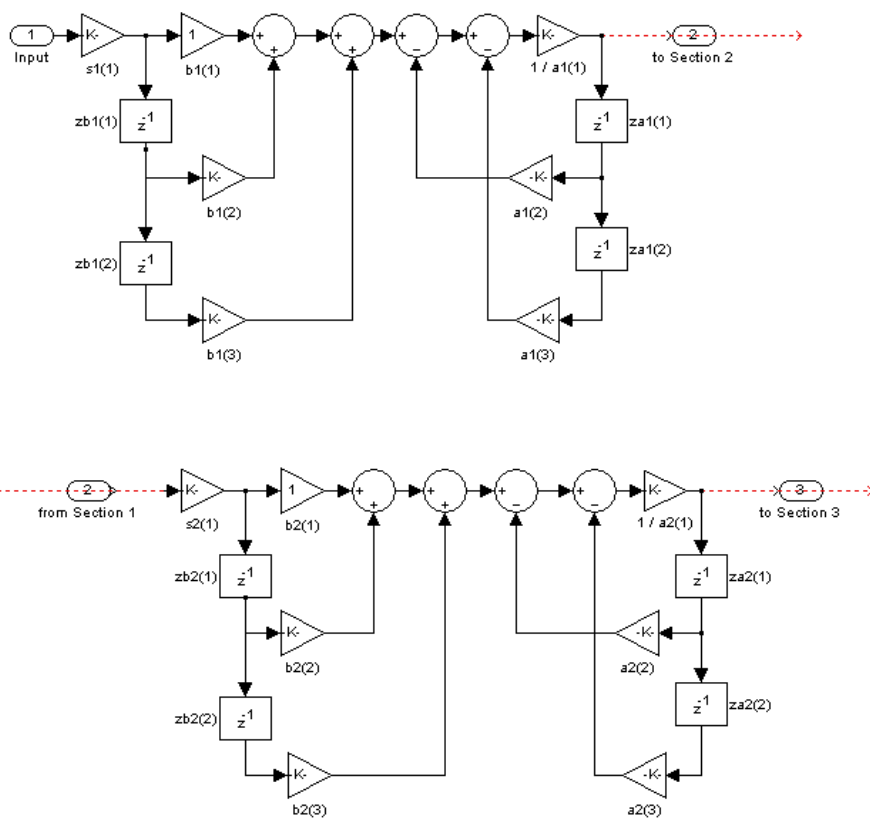
Hd = dfilt.df1sos returns a default, discrete-time, second-order section, direct-form I filter, Hd. This filter passes the input through to the output unchanged.

---

**Note** The leading coefficient of the denominator a(1) cannot be 0.

---

**df1sos**  
(Direct-form I, second-order sections)



To display the filter states, use this code to access the `filtstates` object.

```
Hs = Hd.states           % Where Hd is the dfilt.df1 object and
double (Hs)             % Hs is the filtstates object
```

The vector is

$$\begin{pmatrix} zb1(1) & zb2(1) \\ zb1(2) & zb2(2) \\ za1(1) & za2(1) \\ za1(2) & za2(2) \end{pmatrix}$$

For filters with more than one section, each section is a separate column in the matrix.

## Examples

Specify a second-order sections, direct-form I discrete-time filter with coefficients from a sixth order, low pass, elliptical filter using the following code. The resulting filter has three sections.

```
[z,p,k] = ellip(6,1,60,.4);    % Obtain filter coefficients
[s,g] = zp2sos(z,p,k);        % Convert to SOS
Hd = dfilt.df1sos(s,g)
```

```
Hd =
    FilterStructure: 'Direct-Form I, Second-Order Sections'
           sosMatrix: [3x6 double]
           ScaleValues: [0.0153280112138154;1;1;1]
    PersistentMemory: false
```

## See Also

dfilt, dfilt.df1tsos, dfilt.df2sos, dfilt.df2tsos

**Purpose** Discrete-time, direct-form I transposed filter

**Syntax** `Hd = dfilt.df1t(b,a)`  
`Hd = dfilt.df1t`

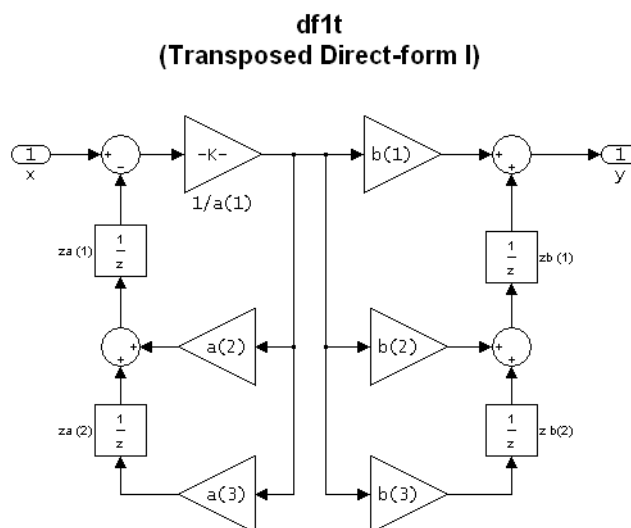
**Description** `Hd = dfilt.df1t(b,a)` returns a discrete-time, direct-form I transposed filter, `Hd`, with numerator coefficients `b` and denominator coefficients `a`. The filter states for this object are stored in a `filtstates` object.

`Hd = dfilt.df1t` returns a default, discrete-time, direct-form I transposed filter, `Hd`, with `b=1` and `a=1`. This filter passes the input through to the output unchanged.

---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0.

---



To display the filter states, use this code to access the `filtstates` object.

```
Hs = Hd.states % Where Hd is the dfilt.df1 object and
```

# dfilt.df1t

---

```
double (Hs)           % Hs is the filtstates object
```

The vector is

$$\begin{pmatrix} zb1(1) & zb2(1) \\ zb1(2) & zb2(2) \\ za1(1) & za2(1) \\ za1(2) & za2(2) \end{pmatrix}$$

## Examples

Create a direct-form I transposed discrete-time filter with coefficients from a fourth-order lowpass Butterworth design

```
[b,a] = butter(4,.5);  
Hd = dfilt.df1t(b,a)  
  
Hd =  
    FilterStructure: 'Direct-Form I Transposed'  
      Numerator: [0.0940 0.3759 0.5639 0.3759 0.0940]  
    Denominator: [1 -3.6082e-016 0.4860 3.6545e-017 0.0177]  
 PersistentMemory: false
```

## See Also

dfilt, dfilt.df1, dfilt.df2, dfilt.df2t



**Purpose** Discrete-time, second-order section, direct-form I transposed filter

**Syntax**  
`Hd = dfilt.df1tsos(s)`  
`Hd = dfilt.df1tsos(b1,a1,b2,a2,...)`  
`Hd = dfilt.df1tsos(...,g)`

**Description** `Hd = dfilt.df1tsos(s)` returns a discrete-time, second-order section, direct-form I, transposed filter, `Hd`, with coefficients given in the `s` matrix. The filter states for this object are stored in a `filtstates` object.

`Hd = dfilt.df1tsos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form I, transposed filter, `Hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

`Hd = dfilt.df1tsos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

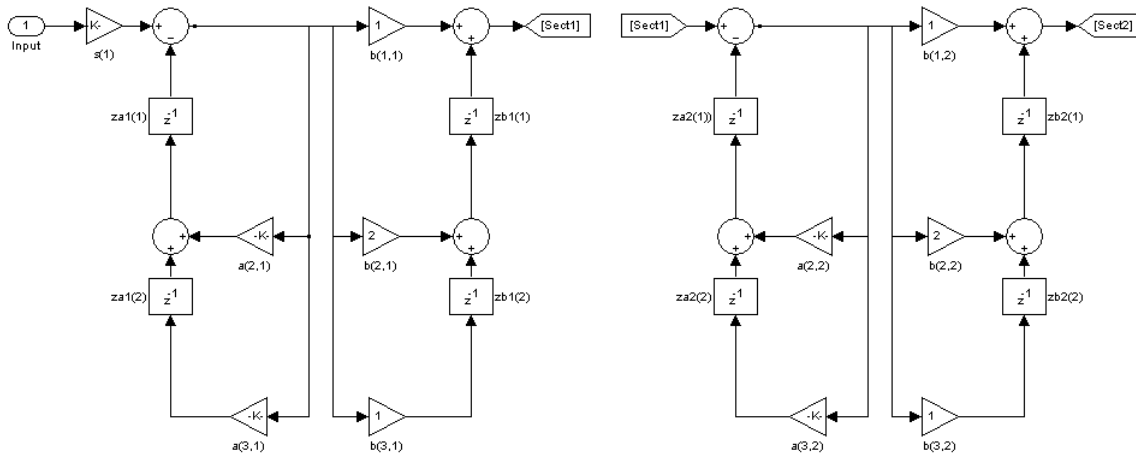
`Hd = dfilt.df1tsos` returns a default, discrete-time, second-order section, direct-form I, transposed filter, `Hd`. This filter passes the input through to the output unchanged.

---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0.

---

**df1tsos**  
(Transposed Direct-form I, second-order sections)



To display the filter states, use this code to access the filtstates object.

```
Hs = Hd.states
double (Hs)
```

```
% Where Hd is the dfilt.df1 object and
% Hs is the filtstates object
```

The matrix is

$$\begin{pmatrix} zb1(1) & zb2(1) \\ zb1(2) & zb2(2) \\ za1(1) & za2(1) \\ za1(2) & za2(2) \end{pmatrix}$$

**Examples**

Specify a second-order sections, direct-form I, transposed discrete-time filter with coefficients from a sixth order, low pass, elliptical filter using the following code:

```
[z,p,k] = ellip(6,1,60,.4);    % Obtain filter coefficients
[s,g] = zp2sos(z,p,k);        % Convert to SOS
Hd = dfilt.df1tsos(s,g)
```

```
Hd =
    FilterStructure: [1x47 char]
           sosMatrix: [3x6 double]
           ScaleValues: [0.0153280112138154;1;1;1]
           PersistentMemory: false
```

```
Hd.FilterStructure    % Display FilterStructure string
ans =
```

```
Direct-Form I Transposed, Second-Order Sections
```

**See Also**

dfilt, dfilt.df1sos, dfilt.df2sos, dfilt.df2tsos

# dfilt.df2

---

**Purpose** Discrete-time, direct-form II filter

**Syntax** Hd = dfilt.df2(b,a)  
Hd = dfilt.df2

**Description** Hd = dfilt.df2(b,a) returns a discrete-time, direct-form II filter, Hd, with numerator coefficients b and denominator coefficients a.

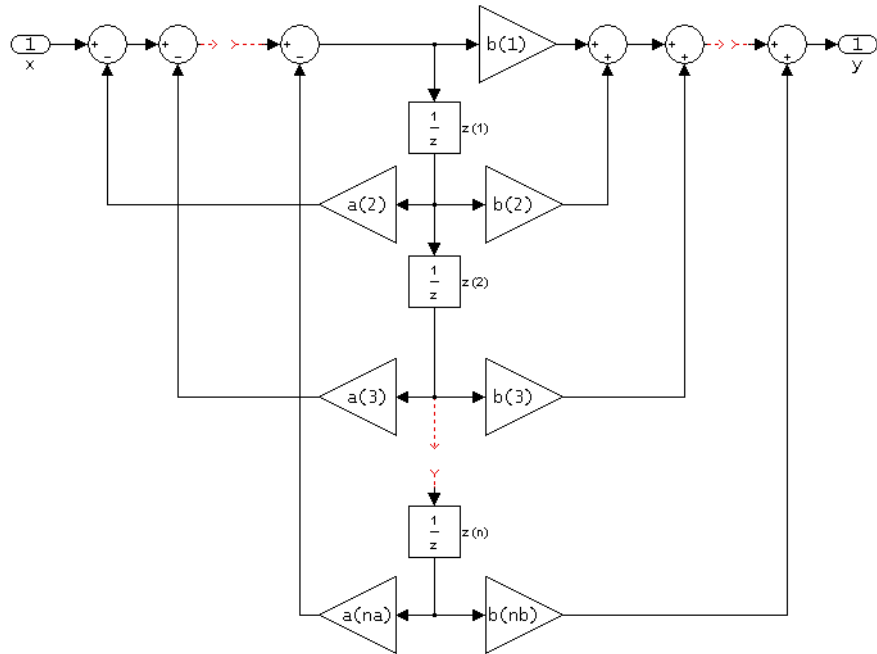
Hd = dfilt.df2 returns a default, discrete-time, direct-form II filter, Hd, with b=1 and a=1. This filter passes the input through to the output unchanged.

---

**Note** The leading coefficient of the denominator a(1) cannot be 0.

---

df2  
(Direct-form II)



The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ \dots \\ z(n) \end{bmatrix}$$

# dfilt.df2

---

## Examples

Create a direct-form II discrete-time filter with coefficients from a fourth-order lowpass Butterworth design

```
[b,a] = butter(4,.5);  
Hd = dfilt.df2(b,a)  
  
Hd =  
    FilterStructure: 'Direct-Form II'  
      Numerator: [0.0940 0.3759 0.5639 0.3759 0.0940]  
    Denominator: [1 -3.6082e-016 0.4860 3.6545e-017 0.0177]  
 PersistentMemory: false
```

## See Also

dfilt, dfilt.df1, dfilt.df1t, dfilt.df2t

**Purpose** Discrete-time, second-order section, direct-form II filter

**Syntax**

```
Hd = dfilt.df2sos(s)
Hd = dfilt.df2sos(b1,a1,b2,a2,...)
Hd = dfilt.df2sos(...,g)
```

**Description**

`Hd = dfilt.df2sos(s)` returns a discrete-time, second-order section, direct-form II filter, `Hd`, with coefficients given in the `s` matrix.

`Hd = dfilt.df2sos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form II object, `Hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

`Hd = dfilt.df2sos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

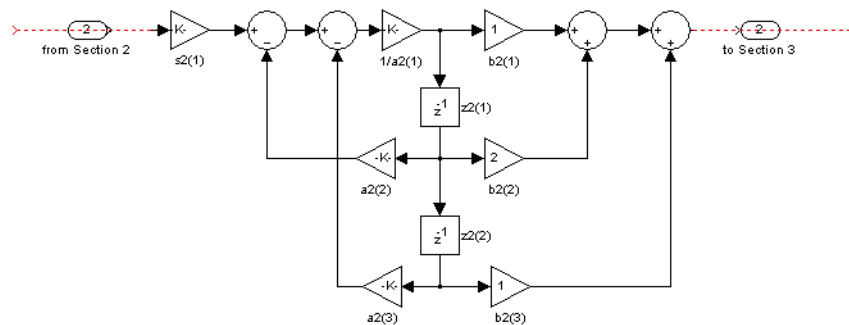
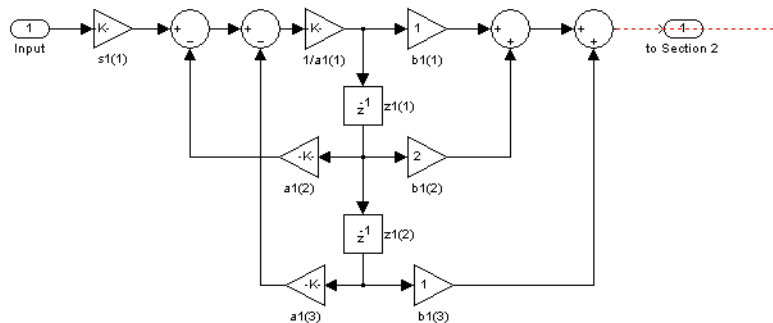
`Hd = dfilt.df2sos` returns a default, discrete-time, second-order section, direct-form II filter, `Hd`. This filter passes the input through to the output unchanged.

---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0.

---

**df2sos**  
(Direct-form II, second-order sections)



The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

For filters with more than one section, each section is a separate column in the vector.



**Examples**

Specify a second-order sections, direct-form II discrete-time filter with coefficients from a sixth order, low pass, elliptical filter using the following code:

```
[z,p,k] = ellip(6,1,60,.4);    % Obtain filter coefficients
[s,g] = zp2sos(z,p,k);        % Convert to SOS
Hd = dfilt.df2sos(s,g)
```

```
Hd =
```

```
    FilterStructure: [1x37 char]
           sosMatrix: [3x6 double]
           ScaleValues: [0.0153280112138154;1;1;1]
    PersistentMemory: false
```

```
Hd.FilterStructure    % Display FilterStructure string
ans =
Direct-Form II Transposed, Second-Order Sections
```

**See Also**

dfilt, dfilt.df1sos, dfilt.df1tsos, dfilt.df2tsos

# dfilt.df2t

## Purpose

Discrete-time, direct-form II transposed filter

## Syntax

`Hd = dfilt.df2t(b,a)`

`Hd = dfilt.df2t`

## Description

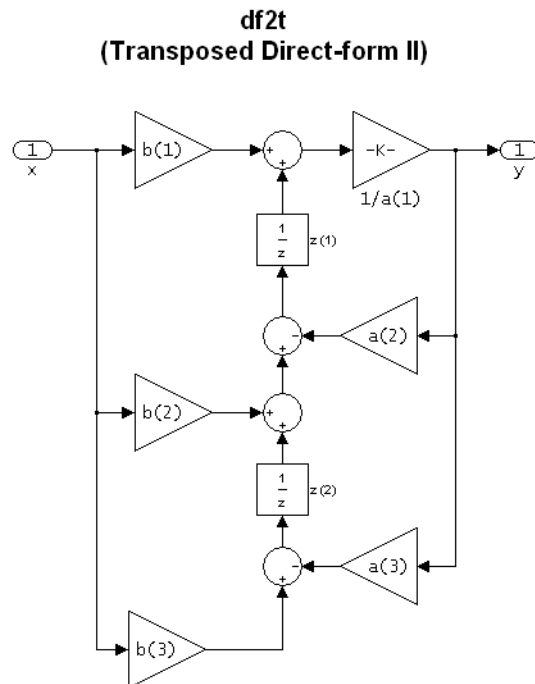
`Hd = dfilt.df2t(b,a)` returns a discrete-time, direct-form II transposed filter, `Hd`, with numerator coefficients `b` and denominator coefficients `a`.

`Hd = dfilt.df2t` returns a default, discrete-time, direct-form II transposed filter, `Hd`, with `b=1` and `a=1`. This filter passes the input through to the output unchanged.

---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0.

---



The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

## Examples

Create a direct-form II transposed discrete-time filter with coefficients from a fourth-order lowpass Butterworth design

```
[b,a] = butter(4,.5);  
Hd = dfilt.df2t(b,a)  
  
Hd =  
    FilterStructure: 'Direct-Form II Transposed'  
          Numerator: [0.0940 0.3759 0.5639 0.3759 0.0940]  
          Denominator: [1 -3.6082e-016 0.4860 3.6545e-017 0.0177]  
    PersistentMemory: false
```

## See Also

dfilt, dfilt.df1, dfilt.df1t, dfilt.df2

# dfilt.df2tsos

---

**Purpose** Discrete-time, second-order section, direct-form II transposed filter

**Syntax**

```
Hd = dfilt.df2tsos(s)
Hd = dfilt.df2tsos(b1,a1,b2,a2,...)
Hd = dfilt.df2tsos(...,g)
```

**Description** Hd = dfilt.df2tsos(s) returns a discrete-time, second-order section, direct-form II, transposed filter, Hd, with coefficients given in the s matrix.

Hd = dfilt.df2tsos(b1,a1,b2,a2,...) returns a discrete-time, second-order section, direct-form II, transposed filter, Hd, with coefficients for the first section given in the b1 and a1 vectors, for the second section given in the b2 and a2 vectors, etc.

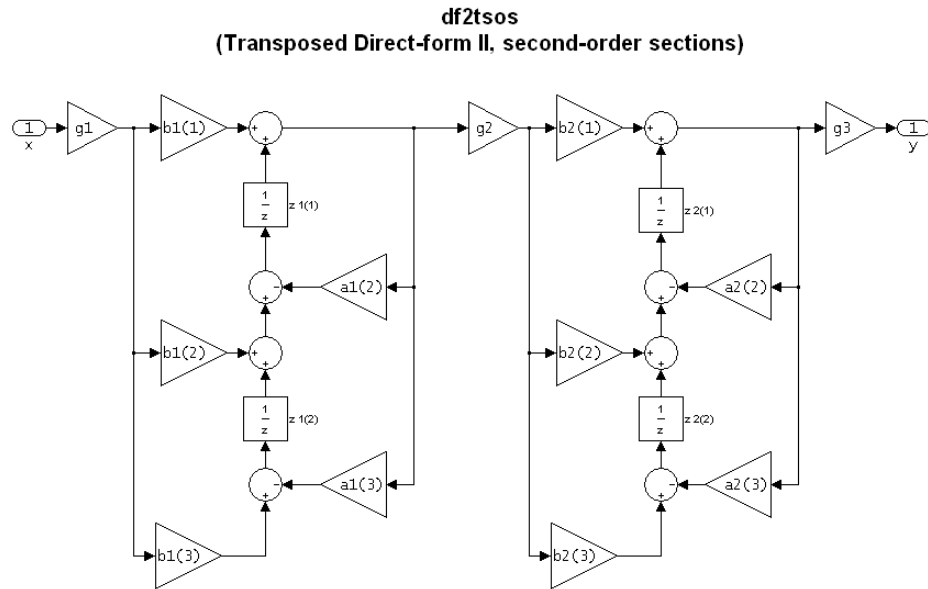
Hd = dfilt.df2tsos(...,g) includes a gain vector g. The elements of g are the gains for each section. The maximum length of g is the number of sections plus one. If g is not specified, all gains default to one.

Hd = dfilt.df2tsos returns a default, discrete-time, second-order section, direct-form II, transposed filter, Hd. This filter passes the input through to the output unchanged.

---

**Note** The leading coefficient of the denominator a(1) cannot be 0.

---



The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

## Examples

Specify a second-order sections, direct-form II, transposed discrete-time filter with coefficients from a sixth order, low pass, elliptical filter using the following code:

```
[z,p,k] = ellip(6,1,60,.4);    % Obtain filter coefficients
[s,g] = zp2sos(z,p,k);        % Convert to SOS
Hd = dfilt.df2tsos(s,g)
```

## dfilt.df2tsos

---

```
Hd =  
    FilterStructure: [1x48 char]  
        sosMatrix: [3x6 double]  
        ScaleValues: [0.0153280112138154;1;1;1]  
    PersistentMemory: false
```

### See Also

dfilt, dfilt.df1sos, dfilt.df1tsos, dfilt.df2sos

**Purpose** Discrete-time, direct-form antisymmetric FIR filter

**Syntax** `Hd = dfilt.dfasymfir(b)`  
`Hd = dfilt.dfasymfir`

**Description** `Hd = dfilt.dfasymfir(b)` returns a discrete-time, direct-form, antisymmetric FIR filter, `Hd`, with numerator coefficients `b`.

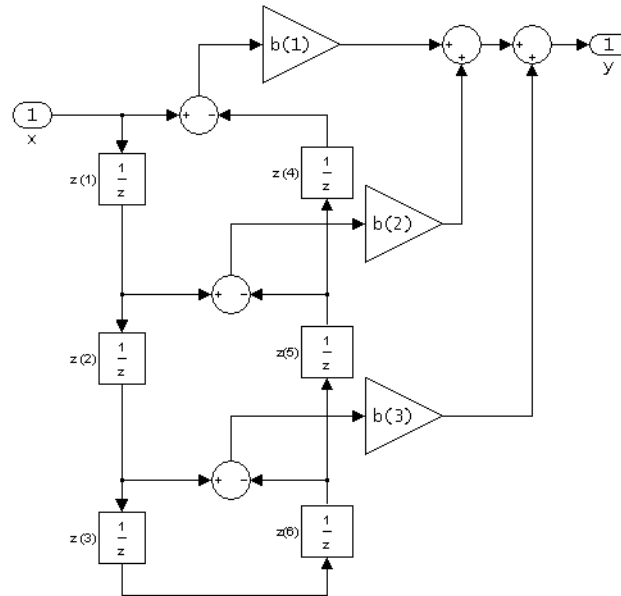
`Hd = dfilt.dfasymfir` returns a default, discrete-time, direct-form, antisymmetric FIR filter, `Hd`, with `b=1`. This filter passes the input through to the output unchanged.

---

**Note** Only the first half of vector `b` is used because the second half is assumed to be antisymmetric. In the figure below for an odd number of coefficients,  $b(3) = 0$ ,  $b(4) = -b(2)$  and  $b(5) = -b(1)$ , and in the next figure for an even number of coefficients,  $b(4) = -b(3)$ ,  $b(5) = -b(2)$ , and  $b(6) = -b(1)$ .

---

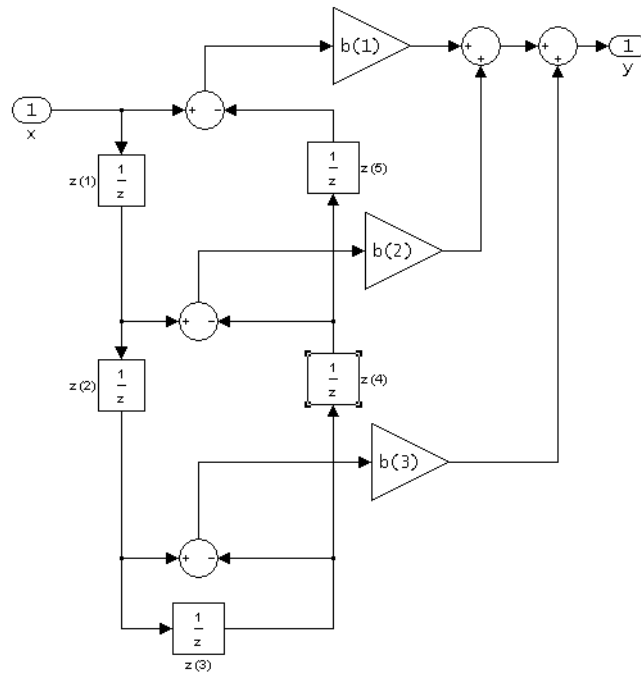
**dfasymfir**  
**(Antisymmetric FIR)**  
**Even order**  
**Odd number of coefficients, length(b) = 7**



**Note that antisymmetry is defined as**  
 **$b(i) == -b(\text{end} - i + 1)$**   
**so that the middle coefficient is zero for odd length**  
 **$b((\text{end}+1)/2) = 0$**



**dfasymfir**  
**(Antisymmetric FIR)**  
**Even number of coefficients, length(b) = 6**



$$b(i) == -b(\text{end} - i + 1)$$

The resulting filter states column vector for the odd number of coefficients example above is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \\ z(4) \\ z(5) \\ z(6) \end{bmatrix}$$

## Examples

### Odd Order

Create a Type 4 25<sup>th</sup> order highpass direct-form antisymmetric FIR filter structure for a `dfilt` object, `Hd`, with the following code:

```
Hd = firpm(25,[0 .4 .5 1],[0 0 1 1],'h');
```

### Even Order

Create a 44<sup>th</sup> order lowpass direct-form antisymmetric FIR differentiator filter structure for a `dfilt` object, `Hd`, with the following code:

```
h=firpm(44,[0 .3 .4 1],[0 .2 0 0],'differentiator');
```

## See Also

`dfilt`, `dfilt.dffir`, `dfilt.dffirt`, `dfilt.dfsymfir`

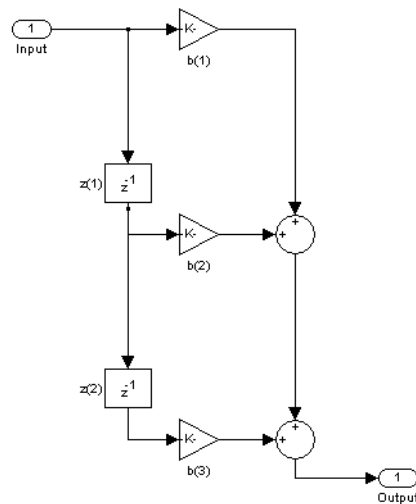
**Purpose** Discrete-time, direct-form, FIR filter

**Syntax**  
`Hd = dfilt.dffir(b)`  
`Hd = dfilt.dffir`

**Description** `Hd = dfilt.dffir(b)` returns a discrete-time, direct-form finite impulse response (FIR) filter, `Hd`, with numerator coefficients, `b`.

`Hd = dfilt.dffir` returns a default, discrete-time, direct-form FIR filter, `Hd`, with `b=1`. This filter passes the input through to the output unchanged.

**dffir**  
**(Direct-form FIR = Tapped delay line)**



The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

# dfilt.dffir

---

## Examples

Create a direct-form FIR discrete-time filter with coefficients from a 30<sup>th</sup> order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);  
Hd = dfilt.dffir(b)  
  
Hd =  
    FilterStructure: 'Direct-Form FIR'  
    Numerator: [1x31 double]  
    PersistentMemory: false
```

## See Also

dfilt, dfilt.dfasymfir, dfilt.dffirt, dfilt.dfsymfir

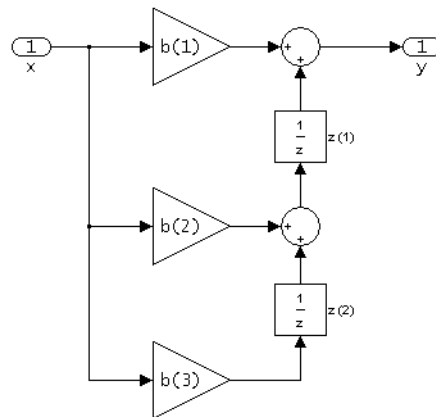
**Purpose** Discrete-time, direct-form FIR transposed filter

**Syntax** Hd = dfilt.dffirt(b)  
Hd = dfilt.dffirt

**Description** Hd = dfilt.dffirt(b) returns a discrete-time, direct-form FIR transposed filter, Hd, with numerator coefficients b.

Hd = dfilt.dffirt returns a default, discrete-time, direct-form FIR transposed filter, Hd, with b=1. This filter passes the input through to the output unchanged.

**dffirt**  
(Transposed Direct-form FIR)



The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

# dfilt.dffirt

---

## Examples

Create a direct-form FIR transposed discrete-time filter with coefficients from a 30<sup>th</sup> order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.dffirt(b)

Hd =
    FilterStructure: 'Direct-Form FIR Transposed'
      Numerator: [1x31 double]
 PersistentMemory: false
```

## See Also

[dfilt](#), [dfilt.dffir](#), [dfilt.dfasymfir](#), [dfilt.dfsymfir](#)

**Purpose** Discrete-time, direct-form symmetric FIR filter

**Syntax** `Hd = dfilt.dfsymfir(b)`  
`Hd = dfilt.dfsymfir`

**Description** `Hd = dfilt.dfsymfir(b)` returns a discrete-time, direct-form symmetric FIR filter, `Hd`, with numerator coefficients `b`.

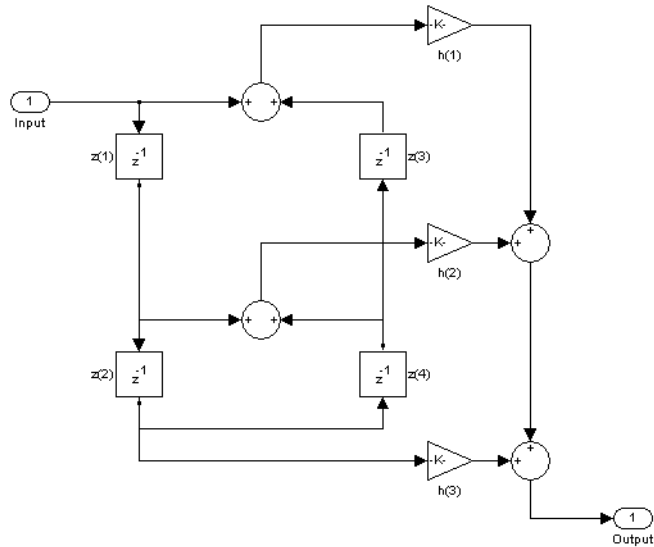
`Hd = dfilt.dfsymfir` returns a default, discrete-time, direct-form symmetric FIR filter, `Hd`, with `b=1`. This filter passes the input through to the output unchanged.

---

**Note** Only the first half of vector `b` is used because the second half is assumed to be symmetric. In the figure below for an odd number of coefficients,  $b(3) = 0$ ,  $b(4) = b(2)$  and  $b(5) = b(1)$ , and in the next figure for an even number of coefficients,  $b(4) = b(3)$ ,  $b(5) = b(2)$ , and  $b(6) = b(1)$ .

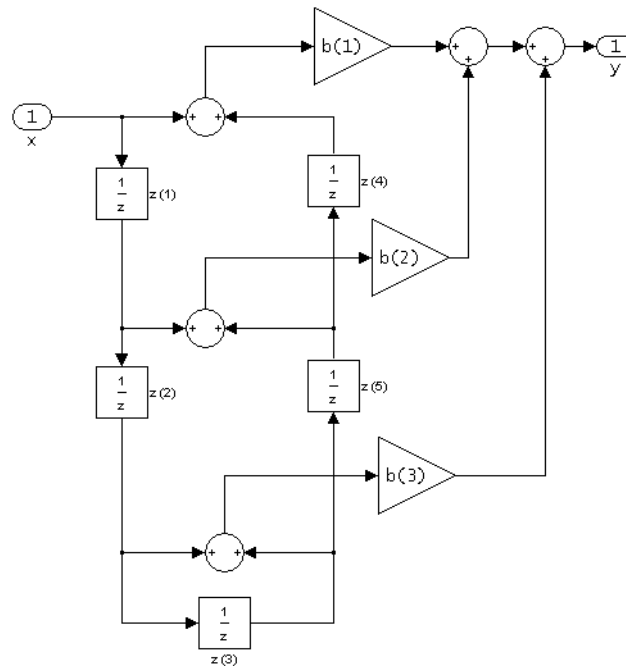
---

**dfsymfir**  
(Symmetric FIR)  
Even order  
Odd number of coefficients, length(b) = 5  
 $b(i) == b(\text{end} - i + 1)$





**dfsymfir**  
**(Symmetric FIR)**  
**Odd order**  
**Even number of coefficients, length(b) = 6**  
 **$b(i) == b(\text{end} - i + 1)$**



The resulting filter states column vector for the odd number of coefficients example above is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \\ z(4) \end{bmatrix}$$

## Examples

### Odd Order

Specify a fifth-order direct-form symmetric FIR filter structure for a `dfilt` object, `Hd`, with the following code:

```
b = [-0.008 0.06 0.44 0.44 0.06 -0.008];
Hd = dfilt.dfsymfir(b)

Hd =
    FilterStructure: 'Direct-Form Symmetric FIR'
      Numerator: [-0.0080 0.0600 0.4400 0.4400 0.0600 -0.0080]
 PersistentMemory: false
```

### Even Order

Specify a fourth-order direct-form symmetric FIR filter structure for a `dfilt` object, `Hd`, with the following code:

```
b = [-0.01 0.1 0.8 0.1 -0.01];
Hd = dfilt.dfsymfir(b)

Hd =
    FilterStructure: 'Direct-Form Symmetric FIR'
      Numerator: [-0.0100 0.1000 0.8000 0.1000 -0.0100]
 PersistentMemory: false
```

## See Also

`dfilt`, `dfilt.dfasymfir`, `dfilt.dffir`, `dfilt.dffirt`

---

**Purpose** Discrete-time, overlap-add, FIR filter

**Syntax**

```
Hd = dfilt.fftfir(b,len)
Hd = dfilt.fftfir(b)
Hd = dfilt.fftfir
```

**Description** This object uses the overlap-add method of block FIR filtering, which is very efficient for streaming data.

`Hd = dfilt.fftfir(b,len)` returns a discrete-time, FFT, FIR filter, `Hd`, with numerator coefficients, `b` and block length, `len`. The block length is the number of input points to use for each overlap-add computation.

`Hd = dfilt.fftfir(b)` returns a discrete-time, FFT, FIR filter, `Hd`, with numerator coefficients, `b` and block length, `len=100`.

`Hd = dfilt.fftfir` returns a default, discrete-time, FFT, FIR filter, `Hd`, with the numerator `b=1` and block length, `len=100`. This filter passes the input through to the output unchanged.

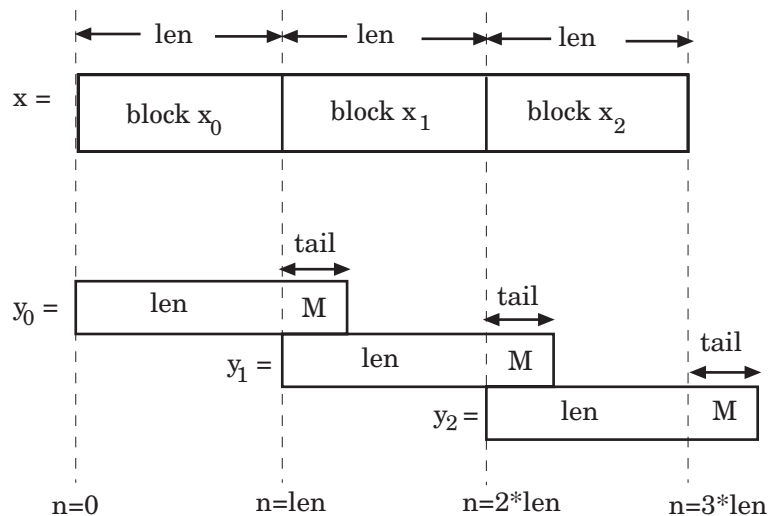
---

**Note** When you use a `dfilt.fftfir` object to filter, the input signal length must be an integer multiple of the object's block length, `len`.

The resulting number of FFT points = (filter length + the block length - 1). The filter is most efficient if the number of FFT points is a power of 2.

---

The `fftfir` uses an overlap-add block processing algorithm, which is represented as follows,



where  $len$  is the block length and  $M$  is the length of the numerator-1,  $(length(b) - 1)$ , which is also the number of states. The output of each convolution is a block that is longer than the input block by a tail of  $(length(b) - 1)$  samples. These tails overlap the next block and are added to it. The states reported by `dfilt.fftfir` are the tails of the final convolution.

## Examples

Create an FFT FIR discrete-time filter with coefficients from a 30<sup>th</sup> order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.fftfir(b)

Hd =
    FilterStructure: 'Overlap-Add FIR'
           Numerator: [1x31 double]
           BlockLength: 100
 NonProcessedSamples: []
 PersistentMemory: false
```

To view the frequency domain coefficients used in the filtering, use the following command.

```
fftcoeffs(Hd)
```

**See Also**

`dfilt`, `dfilt.dffir`, `dfilt.dfasymfir`, `dfilt.dffirt`, `dfilt.dfsymfir`

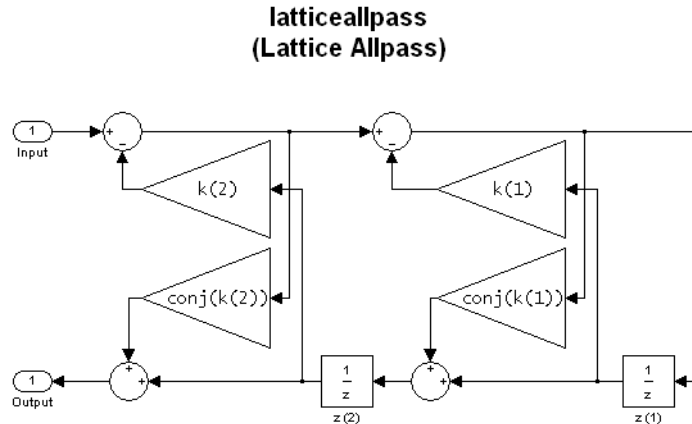
# dfilt.latticeallpass

**Purpose** Discrete-time, lattice allpass filter

**Syntax**  
`Hd = dfilt.latticeallpass(k)`  
`Hd = dfilt.latticeallpass`

**Description** `Hd = dfilt.latticeallpass(k)` returns a discrete-time, lattice allpass filter, `Hd`, with lattice coefficients, `k`.

`Hd = dfilt.latticeallpass` returns a default, discrete-time, lattice allpass filter, `Hd`, with `k=[]`. This filter passes the input through to the output unchanged.



The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

## Examples

Form a third-order lattice allpass filter structure for a `dfilt` object, `Hd`, using the following lattice coefficients:

```
k = [.66 .7 .44];  
Hd = dfilt.latticeallpass(k)  
  
Hd =
```

```
FilterStructure: 'Lattice Allpass'  
    Lattice: [0.6600 0.7000 0.4400]  
PersistentMemory: false
```

### See Also

dfilt, dfilt.latticear, dfilt.latticearma, dfilt.latticemamax,  
dfilt.latticemamin

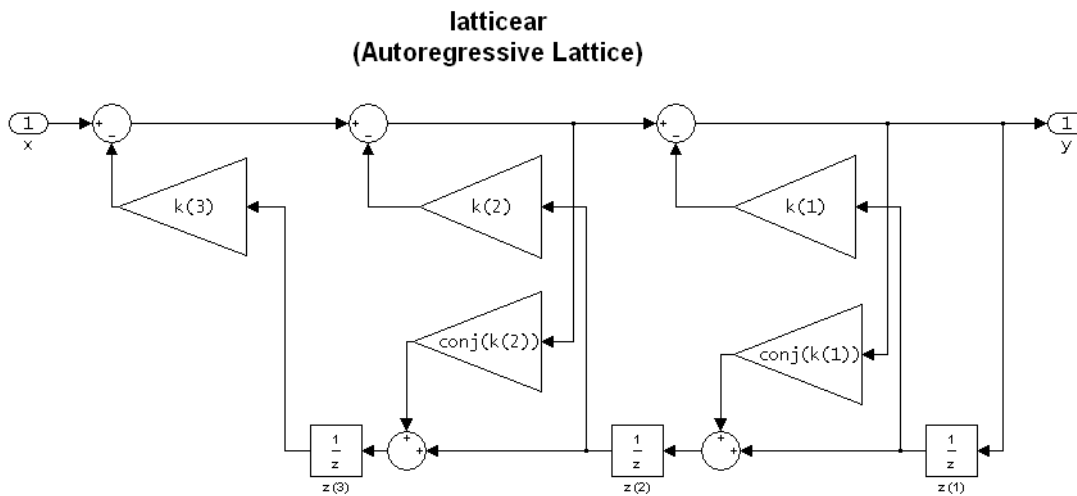
# dfilt.latticear

**Purpose** Discrete-time, lattice, autoregressive filter

**Syntax**  
`Hd = dfilt.latticear(k)`  
`Hd = dfilt.latticear`

**Description** `Hd = dfilt.latticear(k)` returns a discrete-time, lattice autoregressive filter, `Hd`, with lattice coefficients, `k`.

`Hd = dfilt.latticear` returns a default, discrete-time, lattice autoregressive filter, `Hd`, with `k=[]`. This filter passes the input through to the output unchanged.



The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \end{bmatrix}$$



## Examples

Form a third-order lattice autoregressive filter structure for a `dfilt` object, `Hd`, using the following lattice coefficients:

```
k = [.66 .7 .44];  
Hd = dfilt.latticear(k)  
  
Hd =  
           FilterStructure: 'Lattice Autoregressive (AR)'  
           Lattice: [0.6600 0.7000 0.4400]  
           PersistentMemory: false
```

## See Also

`dfilt`, `dfilt.latticeallpass`, `dfilt.latticearma`, `dfilt.latticemax`,  
`dfilt.latticemin`

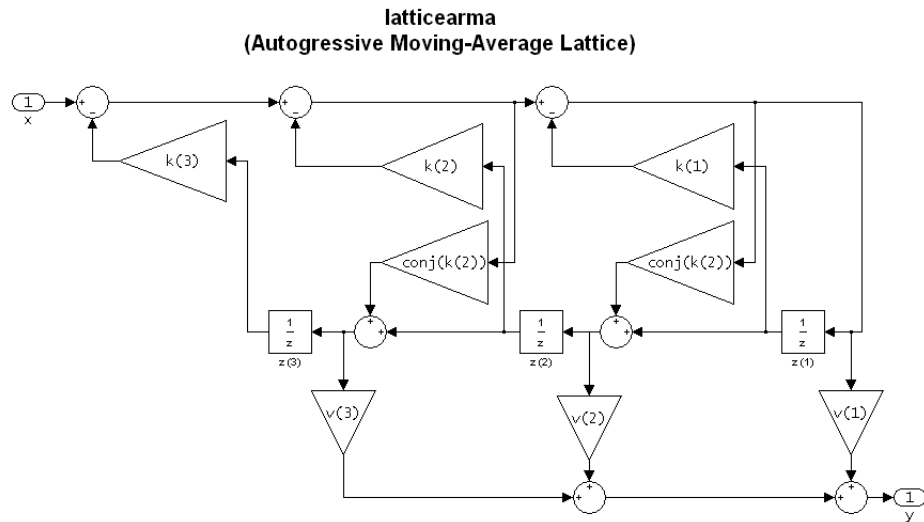
# dfilt.latticearma

**Purpose** Discrete-time, lattice, autoregressive, moving-average filter

**Syntax**  
`Hd = dfilt.latticearma(k,v)`  
`Hd = dfilt.latticearma`

**Description**  
`Hd = dfilt.latticearma(k,v)` returns a discrete-time, lattice autoregressive, moving-average filter, `Hd`, with lattice coefficients, `k` and ladder coefficients `v`.

`Hd = dfilt.latticearma` returns a default, discrete-time, lattice autoregressive, moving-average filter, `Hd`, with `k=[]` and `v=1`. This filter passes the input through to the output unchanged.



The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \end{bmatrix}$$

## Examples

Form a third-order lattice autoregressive, moving-average filter structure for a `dfilt` object, `Hd`, using the following lattice coefficients:

```
k = [.66 .7 .44];  
Hd = dfilt.latticearma(k)  
  
Hd =  
  FilterStructure: 'Lattice Autoregressive Moving-Average (ARMA)'  
    Lattice: [0.6600 0.7000 0.4400]  
      Ladder: 1  
 PersistentMemory: false
```

## See Also

`dfilt`, `dfilt.latticeallpass`, `dfilt.latticear`, `dfilt.latticemamax`,  
`dfilt.latticemamin`

# dfilt.latticemamax

**Purpose** Discrete-time, lattice, moving-average filter

**Syntax**  
`Hd = dfilt.latticemamax(k)`  
`Hd = dfilt.latticemamax`

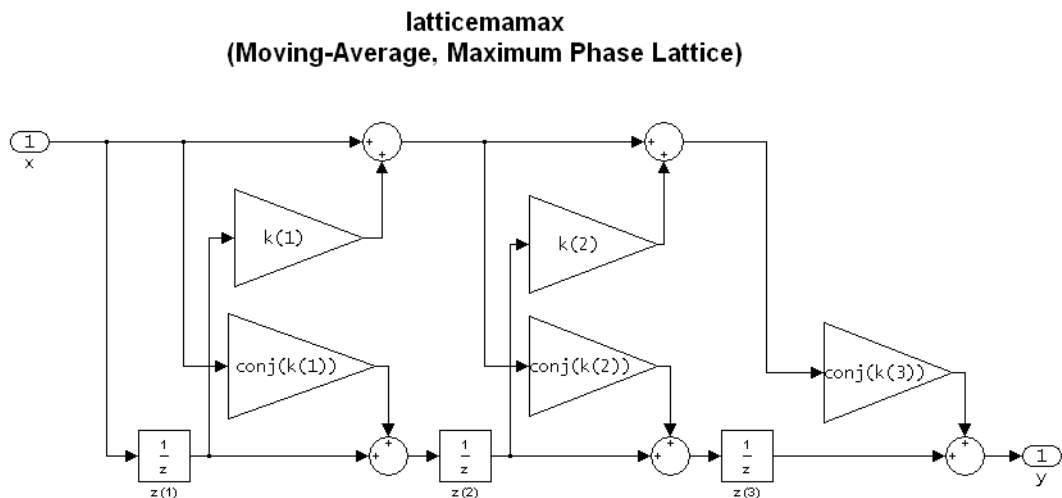
**Description** `Hd = dfilt.latticemamax(k)` returns a discrete-time, lattice, moving-average filter, `Hd`, with lattice coefficients `k`.

---

**Note** If the `k` coefficients define a maximum phase filter, the resulting filter in this structure is maximum phase. If your coefficients do not define a maximum phase filter, placing them in this structure does not produce a maximum phase filter.

---

`Hd = dfilt.latticemamax` returns a default discrete-time, lattice, moving-average filter, `Hd`, with `k=[]`. This filter passes the input through to the output unchanged.



The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \end{bmatrix}$$

**Examples**

Form a fourth-order lattice, moving-average, maximum phase filter structure for a `dfilt` object, `Hd`, using the following lattice coefficients:

```
k = [.66 .7 .44 .33];  
Hd = dfilt.latticemamax(k)  
  
Hd =  
  FilterStructure: 'Lattice Moving-Average (MA) For Maximum Phase'  
    Arithmetic: 'double'  
      Lattice: [0.6600 0.7000 0.4400 0.3300]  
 PersistentMemory: false
```

**See Also**

`dfilt`, `dfilt.latticeallpass`, `dfilt.latticear`, `dfilt.latticearma`,  
`dfilt.latticemamin`

# dfilt.latticemamin

**Purpose** Discrete-time, lattice, moving-average filter

**Syntax**  
`Hd = dfilt.latticemamin(k)`  
`Hd = dfilt.latticemamin`

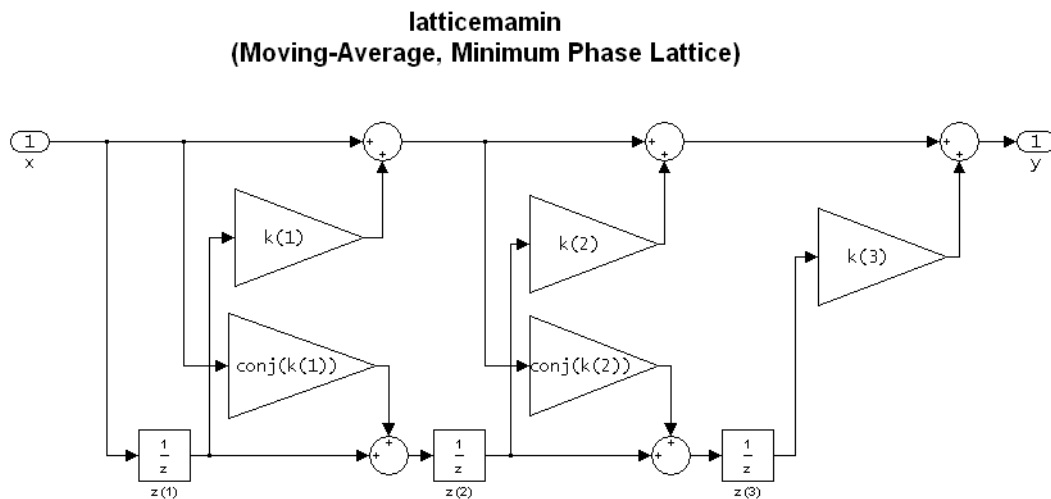
**Description** `Hd = dfilt.latticemamin(k)` returns a discrete-time, lattice, moving-average, minimum phase, filter, `Hd`, with lattice coefficients `k`.

---

**Note** If the `k` coefficients define a minimum phase filter, the resulting filter in this structure is minimum phase. If your coefficients do not define a minimum phase filter, placing them in this structure does not produce a minimum phase filter.

---

`Hd = dfilt.latticemamin` returns a default discrete-time, lattice, moving-average, minimum phase, filter, `Hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.



The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \end{bmatrix}$$

**Examples**

Form a third-order lattice, moving-average, minimum phase, filter structure for a `dfilt` object, `Hd`, using the following lattice coefficients.

```
k = [.66 .7 .44];  
Hd = dfilt.latticemamin(k)  
  
Hd =  
  FilterStructure: 'Lattice Moving-Average (MA) For Minimum Phase'  
    Lattice: [0.6600 0.7000 0.4400]  
 PersistentMemory: false
```

**See Also**

`dfilt`, `dfilt.latticeallpass`, `dfilt.latticear`, `dfilt.latticearma`,  
`dfilt.latticemamax`

# dfilt.parallel

**Purpose** Discrete-time, parallel structure filter

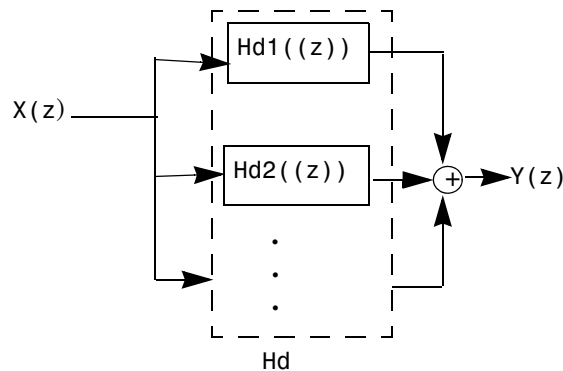
**Syntax** `Hd = dfilt.parallel(Hd1,Hd2,...)`

**Description** `Hd = dfilt.parallel(Hd1,Hd2,...)` returns a discrete-time filter, `Hd`, which is a structure of two or more `dfilt` filters, `Hd1`, `Hd2`, etc. arranged in parallel. Each filter in a parallel structure is a separate stage. You can display states for individual stages only. To view the states of a stage use

```
Hd.stage(1).states
```

You can also use the `nondot` notation format for calling a parallel structure.

```
parallel(Hd1,Hd2,...)
```



## Examples

Using a parallel structure, create a coupled-allpass decomposition of a 7th order lowpass digital, elliptic filter with a normalized cutoff frequency of 0.5, 1 decibel of peak-to-peak ripple and a minimum stopband attenuation of 40 decibels.

```
k1 = [-0.0154    0.9846   -0.3048    0.5601];  
Hd1 = dfilt.latticeallpass(k1);  
k2 = [-0.1294    0.8341   -0.4165];  
Hd2 = dfilt.latticeallpass(k2);  
Hpar = parallel(Hd1 ,Hd2);
```



```
gain = dfilt.scalar(0.5);      % Normalize the passband gain
Hcas = cascade(gain,Hpar);
```

For details on the stages of this filter use:

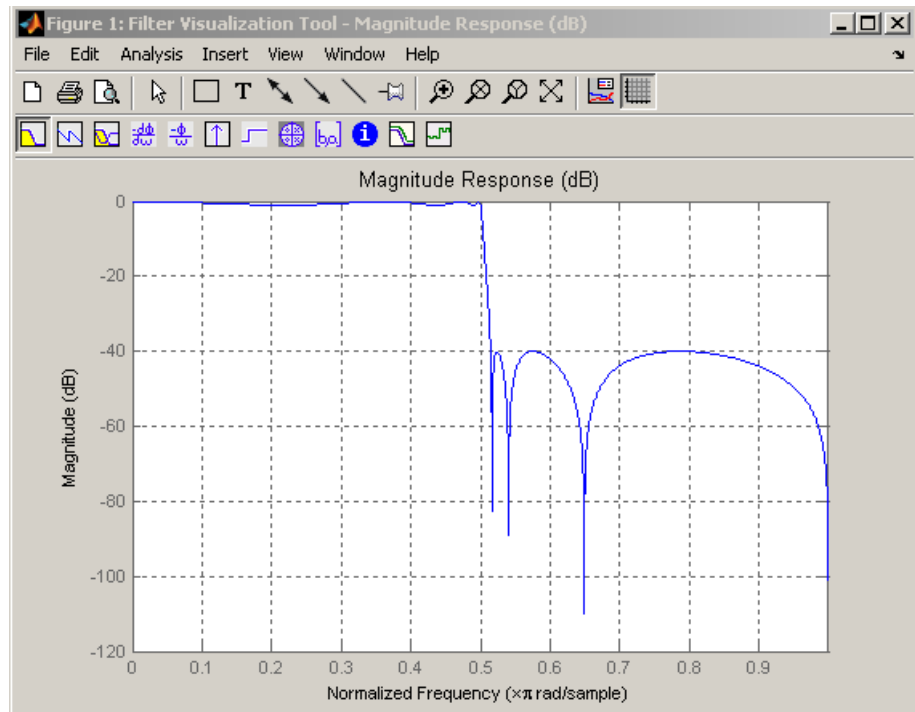
```
info(Hcas.Stage(1))
```

and

```
info(Hcas.Stage(2))
```

To view this filter, use:

```
fvtool(Hcas)
```



## See Also

dfilt, dfilt.cascade

# dfilt.scalar

---

**Purpose** Discrete-time, scalar filter

**Syntax** Hd = dfilt.scalar(g)  
Hd = dfilt.scalar

**Description** Hd = dfilt.scalar(g) returns a discrete-time, scalar filter, Hd, with gain g, where g is a scalar.

Hd = dfilt.scalar returns a default, discrete-time scalar gain filter, Hd, with gain 1.

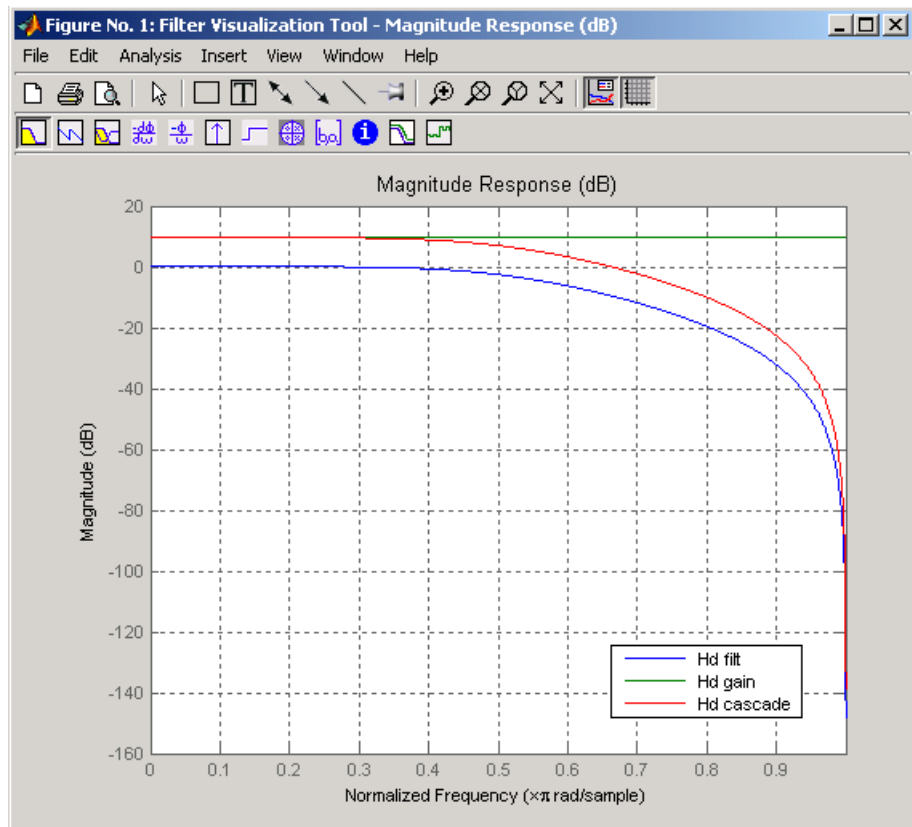
**Example** Create a direct-form I filter and a scalar object with a gain of 3 and cascade them together.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
Hd_filt = dfilt.df1(b,a)
Hd_gain = dfilt.scalar(3)
Hd=cascade(Hd_gain,Hd_filt)
fvtool(Hd_filt,Hd_gain,Hd)

Hd_filt =
    FilterStructure: 'Direct-Form I'
        Numerator: [0.3000 0.6000 0.3000]
        Denominator: [1 0 0.2000]
    PersistentMemory: false

Hd_gain =
    FilterStructure: 'Scalar'
        Gain: 3
    PersistentMemory: false

Hd =
    FilterStructure: Cascade
        Stage(1): Scalar
        Stage(2): Direct-Form I
    PersistentMemory: false
```



To view the stages of the cascaded filter, use

```
Hd.stage(1)
```

```
ans =
```

```
    FilterStructure: 'Scalar'  
           Gain: 3  
 PersistentMemory: false
```

and

## dfilt.scalar

---

```
Hd.stage(2)
```

```
ans =
```

```
    FilterStructure: 'Direct-Form I'  
      Numerator: [0.3 0.6 0.3]  
      Denominator: [1 0 0.2]  
 PersistentMemory: false
```

### See Also

dfilt, dfilt.cascade

**Purpose** Discrete-time, state-space filter

**Syntax** Hd = dfilt.statespace(A,B,C,D)  
Hd = dfilt.statespace

**Description** Hd = dfilt.statespace(A,B,C,D) returns a discrete-time state-space filter, Hd, with rectangular arrays A, B, C and D.

A, B, C, and D are from the matrix or state-space form of a filter's difference equations

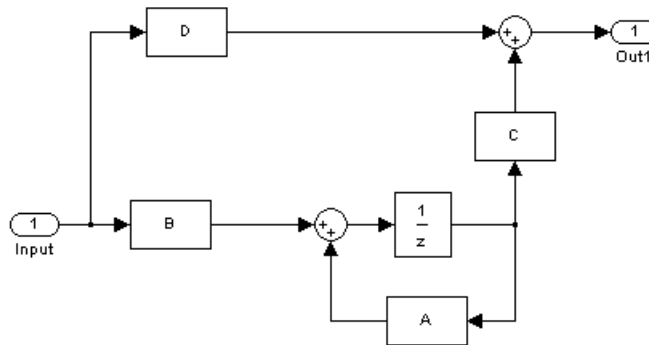
$$x(n+1) = Ax(n) + Bu(n)$$

$$y(n) = Cx(n) + Du(n)$$

where  $x(n)$  is the vector states at time  $n$ ,  $u(n)$  is the input at time  $n$ ,  $y$  is the output at time  $n$ ,  $A$  is the state-transition matrix,  $B$  is the input-to-state transmission matrix,  $C$  is the state-to-output transmission matrix, and  $D$  is the input-to-output transmission matrix. 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.

Hd = dfilt.statespace returns a default, discrete-time state-space filter, Hd, with A=[ ], B=[ ], C=[ ], and D=1. This filter passes the input through to the output unchanged.

## Statespace



The resulting filter states column vector has the same number of rows as the number of rows of A or B.

### Examples

Create a second-order, state-space filter structure from a second-order, lowpass Butterworth design.

```
[A,B,C,D] = butter(2,0.5);  
Hd = dfilt.statespace(A,B,C,D)
```

```
Hd =
```

```
FilterStructure: 'State-space'  
A: [2x2 double]  
B: [0.8284;0.8284]  
C: [0.2071 0.5]  
D: 0.2929  
PersistentMemory: false
```

### See Also

dfilt

<b>Purpose</b>	Discrete Fourier transform matrix
<b>Syntax</b>	<code>A = dftmtx(n)</code>
<b>Description</b>	<p>A <i>discrete Fourier transform matrix</i> is a complex matrix of values around the unit circle, whose matrix product with a vector computes the discrete Fourier transform of the vector.</p> <p><code>A = dftmtx(n)</code> returns the n-by-n complex matrix A that, when multiplied into a length n column vector x.</p> $y = A*x$ <p>computes the discrete Fourier transform of x.</p> <p>The inverse discrete Fourier transform matrix is</p> $A_i = \text{conj}(dftmtx(n)) / n$
<b>Examples</b>	<p>In practice, the discrete Fourier transform is computed more efficiently and uses less memory with an FFT algorithm</p> <pre>x = 1:256; y1 = fft(x);</pre> <p>than by using the Fourier transform matrix.</p> <pre>n = length(x); y2 = x*dftmtx(n); norm(y1-y2)</pre> <pre>ans =     1.8297e-009</pre>
<b>Algorithm</b>	<code>dftmtx</code> takes the FFT of the identity matrix to generate the transform matrix.
<b>See Also</b>	<code>convmtx</code> , <code>fft</code>

# digitrevorder

**Purpose** Permute input into digit-reversed order

**Syntax**  
`y = digitrevorder(x,r)`  
`[y,i] = digitrevorder(x,r)`

**Description** `digitrevorder` is useful for preordering a vector of filter coefficients for use in frequency-domain filtering algorithms, in which the `fft` and `ifft` transforms are computed without digit-reversed ordering for improved run-time efficiency.

`y = digitrevorder(x,r)` returns the input data in digit-reversed order in vector or matrix `y`. The digit-reversal is computed using the number system base (radix base) `r`, which can be any integer from 2 to 36. The length of `x` must be an integer power of `r`. If `x` is a matrix, the digit reversal occurs on the first dimension of `x` with size greater than 1. `y` is the same size as `x`.

`[y,i] = digitrevorder(x,r)` returns the digit-reversed vector or matrix `y` and the digit-reversed indices `i`, such that `y = x(i)`. Recall that MATLAB uses 1-based indexing, so the first index of `y` will be 1, not 0.

The following table shows the numbers 0 through 15, the corresponding digits and the digit-reversed numbers using radix base-4. The corresponding radix base-2 bits and bit-reversed indices are also shown.

Linear Index	Base-4 Digits	Digit-Reversed	Digit-Reversed Index	Base-2 Bits	Base-2 Reversed (bitrevorder)	Bit-Reversed Index
0	00	00	0	0000	0000	0
1	01	10	4	0001	1000	8
2	02	20	8	0010	0100	4
3	03	30	12	0011	1100	12
4	10	01	1	0100	0010	2
5	11	11	5	0101	1010	10
6	110	011	9	0110	0110	6



Linear Index	Base-4 Digits	Digit-Reversed	Digit-Reversed Index	Base-2 Bits	Base-2 Reversed (bitrevorder)	Bit-Reversed Index
7	13	31	13	0111	1110	14
8	20	02	2	1000	0001	1
9	21	12	6	1001	1001	9
10	22	22	10	1010	0101	5
11	23	32	14	1011	1101	13
12	30	03	3	1100	0011	3
13	31	13	7	1101	1011	11
14	32	23	11	1110	0111	7
15	33	33	15	1111	1111	15

**Examples**

Obtain the digit-reversed, radix base-3 ordered output of a vector containing 9 values:

```
x=[0:8]';           % Create a column vector
[x,digitrevorder(x,3)]
ans =

     0     0
     1     3
     2     6
     3     1
     4     4
     5     7
     6     2
     7     5
     8     8
```

**See Also**

bitrevorder, fft, ifft

# diric

---

**Purpose** Dirichlet or periodic sinc function

**Syntax** `y = diric(x,n)`

**Description** `y = diric(x,n)` returns a vector or array `y` the same size as `x`. The elements of `y` are the Dirichlet function of the elements of `x`. `n` must be a positive integer.

The Dirichlet function, or periodic sinc function, is

$$\text{diric}(x, n) = \begin{cases} -1 \frac{x}{2\pi} (n-1) & x = 0, \pm 2\pi, \pm 4\pi, \dots \\ \frac{\sin(nx/2)}{n \sin(x/2)} & \text{else} \end{cases}$$

for any nonzero integer `n`. This function has period  $2\pi$  for `n` odd and period  $4\pi$  for `n` even. Its peak value is 1, and its minimum value is -1 for `n` even. The magnitude of this function is  $(1/n)$  times the magnitude of the discrete-time Fourier transform of the `n`-point rectangular window.

**Diagnostics** If `n` is not a positive integer, `diric` gives the following error message:

Requires `n` to be a positive integer.

**See Also** `cos`, `gauspuls`, `pulstran`, `rectpuls`, `sawtooth`, `sin`, `sinc`, `square`, `tripuls`

**Purpose** Decrease sampling rate by integer factor

**Syntax**  
`y = downsample(x,n)`  
`y = downsample(x,n,phase)`

**Description** `y = downsample(x,n)` decreases the sampling rate of `x` by keeping every  $n^{\text{th}}$  sample starting with the first sample. `x` can be a vector or a matrix. If `x` is a matrix, each column is considered a separate sequence.

`y = downsample(x,n,phase)` specifies the number of samples by which to offset the downsampled sequence. `phase` must be an integer from 0 to  $n-1$ .

**Examples** Decrease the sampling rate of a sequence by 3:

```
x = [1 2 3 4 5 6 7 8 9 10];  
y = downsample(x,3)  
  
y =  
    1     4     7    10
```

Decrease the sampling rate of the sequence by 3 and add a phase offset of 2:

```
y = downsample(x,3,2)  
  
y =  
    3     6     9
```

Decrease the sampling rate of a matrix by 3:

```
x = [1 2 3; 4 5 6; 7 8 9; 10 11 12];  
y = downsample(x,3);  
x,y  
  
x =  
    1     2     3  
    4     5     6  
    7     8     9  
   10    11    12  
  
y =  
    1     2     3  
   10    11    12
```

# downsample

---

## See Also

`decimate`, `interp`, `interp1`, `resample`, `spline`, `upfirdn`, `upsample`

**Purpose** Discrete prolate spheroidal sequences (Slepian sequences)

**Syntax**

```
[e,v] = dpss(n,nw)
[e,v] = dpss(n,nw,k)
[e,v] = dpss(n,nw,[k1 k2])
[e,v] = dpss(n,nw,'int')
[e,v] = dpss(n,nw,'int',Ni)
[e,v] = dpss(...,'trace')
```

**Description** `[e,v] = dpss(n,nw)` generates the first  $2 \cdot nw$  *discrete prolate spheroidal sequences* (DPSS) of length  $n$  in the columns of  $e$ , and their corresponding concentrations in vector  $v$ . They are also generated in the DPSS MAT-file database `dpss.mat`.  $nw$  must be less than  $n/2$ .

`[e,v] = dpss(n,nw,k)` returns the  $k$  most band-limited discrete prolate spheroidal sequences.  $k$  must be an integer such that  $1 \leq k \leq n$ .

`[e,v] = dpss(n,nw,[k1 k2])` returns the  $k1$ st through the  $k2$ nd discrete prolate spheroidal sequences, where  $1 \leq k1 \leq k2 \leq n$ .

For all of the above forms:

- The Slepian sequences are calculated directly.
- The sequences are generated in the frequency band  $|\omega| \leq (2\pi W)$ , where  $W = nw/n$  is the half-bandwidth and  $\omega$  is in rad/sample.
- $e(:,1)$  is the length  $n$  signal most concentrated in the frequency band  $|\omega| \leq (2\pi W)$  radians,  $e(:,2)$  is the signal orthogonal to  $e(:,1)$  that is most concentrated in this band,  $e(:,3)$  is the signal orthogonal to both  $e(:,1)$  and  $e(:,2)$  that is most concentrated in this band, etc.
- For multitaper spectral analysis, typical choices for  $nw$  are 2, 5/2, 3, 7/2, or 4.

`[e,v] = dpss(n,nw,'int')` uses the interpolation method specified by the string `'int'` to compute  $e$  and  $v$  from the sequences in `dpss.mat` with length closest to  $n$ . The string `'int'` can be either:

- `'spline'`: Use spline interpolation.
- `'linear'`: Use linear interpolation. This is much faster but less accurate than spline interpolation.

`[e,v] = dpss(n,nw,'int',Ni)` interpolates from existing length `Ni` sequences. The interpolation method `'linear'` requires `Ni > n`.

`[e,v] = dpss(...,'trace')` uses the trailing string `'trace'` to display which interpolation method DPSS uses. If you don't specify the interpolation method, the display indicates that you are using the *direct method*.

## Examples

### Example 1: Using `dpss`, `dpssave`, and `dpssdir`

Create a catalogue of 16 DPSS functions with `nw = 4`, and use spline interpolation on 10 of these functions while displaying the interpolation method you use. You can do this using `dpss`, `dpsssave`, and `dpssdir`:

```
% Create the catalogue of functions.
[e,v] = dpss(16,4);

% Save e and v in a MAT-file.
dpsssave(4,e,v);

% Find nw = 4. First create a structure called index.
index = dpssdir;
index.wlists
ans =
    NW: 4
    key: 1

% Use spline interpolation on 10 of the DPSS functions.
[e1,v1] = dpss(10,4,'spline',size(e,1),'trace');
```

### Example 2: Using `dpss` and `dpssload`

Create a set of DPSS functions using `dpss`, and use the spline method on a subset of these functions. Use `dpssload` to load the MAT-file created by `dpss`:

```
% Create the catalogue of functions.
[e,v] = dpss(16,4);

% Load dpss.mat, where e and v are saved.
[e1,v1] = dpssload(16,4);

% Use spline interpolation on 10 of the DPSS functions.
[e1,v1] = dpss(10,4,'spline');
```

**References**

[1] Percival, D.B., and A.T. Walden. *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques*. Cambridge: Cambridge University Press, 1993.

**See Also**

dpsscLEAR, dpssdir, dpssload, dpsssave, pmtm

# dpsscLEAR

---

**Purpose** Remove discrete prolate spheroidal sequences from database

**Syntax** `dpsscLEAR(n,nw)`

**Description** `dpsscLEAR(n,nw)` removes sequences with length  $n$  and time-bandwidth product  $nw$  from the DPSS MAT-file database `dpss.mat`.

**See Also** `dpss`, `dpssdir`, `dpssload`, `dpsssave`



**Purpose** Discrete prolate spheroidal sequences database directory

**Syntax**  
dpssdir  
dpssdir(n)  
dpssdir(nw, 'nw')  
dpssdir(n,nw)  
index = dpssdir

**Description** dpssdir manages the database directory that contains the generated DPSS samples in the DPSS MAT-file database `dpss.mat`.

dpssdir lists the directory of saved sequences in `dpss.mat`.

dpssdir(n) lists the sequences saved with length `n`.

dpssdir(nw, 'nw') lists the sequences saved with time-bandwidth product `nw`.

dpssdir(n,nw) lists the sequences saved with length `n` and time-bandwidth product `nw`.

index = dpssdir is a structure array describing the DPSS database. Pass `n` and `nw` options as for the no output case to get a filtered index.

**Examples** See “Example 1: Using dpss, dpssave, and dpssdir” on page 7-214.

**See Also** dpss, dpssc clear, dpssload, dpsssave

# dpssload

---

**Purpose** Load discrete prolate spheroidal sequences from database

**Syntax** `[e,v] = dpssload(n,nw)`

**Description** `[e,v] = dpssload(n,nw)` loads all sequences with length `n` and time-bandwidth product `nw` in the columns of `e` and their corresponding concentrations in vector `v` from the DPSS MAT-file database `dpss.mat`.

**Examples** See “Example 2: Using `dpss` and `dpssload`” on page 7-214.

**See Also** `dpss`, `dpsscLEAR`, `dpssdir`, `dpsssAVE`

**Purpose** Save discrete prolate spheroidal sequences in database

**Syntax** `dpsssave(nw,e,v)`  
`status = dpsssave(nw,e,v)`

**Description** `dpsssave(nw,e,v)` saves the sequences in the columns of `e` and their corresponding concentrations in vector `v` in the DPSS MAT-file database `dpss.mat`. It is not necessary to specify sequence length, because the length of the sequence is determined by the number of rows of `e`.

`nw` is the *time-bandwidth product* that was specified when the sequence was created using `dpss`.

`status = dpsssave(nw,e,v)` returns 0 if the save was successful and 1 if there was an error.

**Examples** See “Example 1: Using `dpss`, `dpsssave`, and `dpssdir`” on page 7-214.

**See Also** `dpss`, `dpsscLEAR`, `dpssdir`, `dpssload`

# dspdata

---

**Purpose** DSP data parameter information

**Syntax** `Hs = dspdata.dataobj(input1,...)`

**Description** `Hs = dspdata.dataobj(input1,...)` returns a dspdata object `Hs` of type `dataobj`. This object contains all the parameter information needed for the specified type of `dataobj`. Each `dataobj` takes one or more inputs, which are described on the individual reference pages. If you do not specify any input values, the returned object has default property values appropriate for the particular `dataobj` type.

---

**Note** You must use a `dataobj` with dspdata.

---

## Data Objects

A data object (`dataobj`) for dspdata specifies the type of data stored in the object. Available `dataobj` types for dspdata are shown below.

<b>dspdata.dataobj</b>	<b>Description</b>
<code>dspdata.msspectrum</code>	Mean-square spectrum data (power)
<code>dspdata.psd</code>	Power spectral density data (power/frequency)
<code>dspdata.pseudospectrum</code>	Pseudospectrum data (power)

For more information on each `dataobj` type, use the syntax `help dspdata.dataobj` at the MATLAB prompt or refer to its reference page.

## Methods

Methods provide ways of performing functions directly on your dspdata object. You can apply these methods directly on the variable you assigned to your dspdata object.

Method	Description
avgpower	<p>Note that this method applies only to <code>dspdata.psd</code> objects.</p> <p><code>avgpower(Hs)</code> computes the average power in a given frequency band. The technique uses a rectangle approximation of the integral of the <code>Hs</code> signal's power spectral density (PSD). If the signal is a matrix, the computation is done on each column. The average power is the total signal power and the <code>SpectrumType</code> property determines whether the total average power is contained in the one-sided or two-sided spectrum. For a one-sided spectrum, the range is <math>[0,\pi]</math> for even number of frequency points and <math>[0,\pi)</math> for odd. For a two-sided spectrum the range is <math>[0,2\pi)</math>.</p> <p><code>avgpower(Hs, freqrange)</code> specifies the frequency range over which to calculate the average power. <code>freqrange</code> is a two-element vector of the frequencies between which to calculate. If a frequency value does not match exactly the frequency in <code>Hs</code>, the next closest value is used. Note that the first frequency value in <code>freqrange</code> is included in the calculation and the second value is excluded.</p>
centerdc	<p><code>centerdc(Hs)</code> or <code>centerdc(Hs, true)</code> shifts the data and frequency values so that the DC component is at the center of the spectrum. If the <code>SpectrumType</code> property is 'onesided', it is changed to 'twosided' and then the DC component is centered.</p> <p><code>centerdc(Hs, 'false')</code> shifts the data and frequency values so that the DC component is at the left edge of the spectrum.</p>

<b>Method</b>	<b>Description</b>
halfrange	<p>halfrange(Hs) converts the Hs spectrum to a spectrum calculated over half the Nyquist interval. All associated properties affected by the new frequency range are adjusted automatically. This method is used for dspdata.pseudospectrum objects.</p> <p>Note that the spectrum is assumed to be from a real signal (that is, halfrange uses half the data points regardless of whether the data is symmetric).</p>
normalizefreq	<p>normalizefreq(Hs) or normalizefreq(Hs,true) normalizes the frequency specifications in the Hs object to Fs so the frequencies are between 0 and 1. It also sets the NormalizedFrequency property to true.</p> <p>normalizefreq(Hs,false) converts the frequencies to linear frequencies.</p> <p>normalizefreq(Hs,false,Fs) sets a new sampling frequency Fs. This can be used only with false.</p>
onesided	<p>onesided(Hs) converts the Hs spectrum to a spectrum calculated over half the Nyquist interval and containing the total signal power. All associated properties affected by the new frequency range are adjusted automatically. This method is used for dspdata.psd and dspdata.msspectrum objects.</p> <p>Note that the spectrum is assumed to be from a real signal (that is, onesided uses half the data points regardless of whether the data is symmetric).</p>

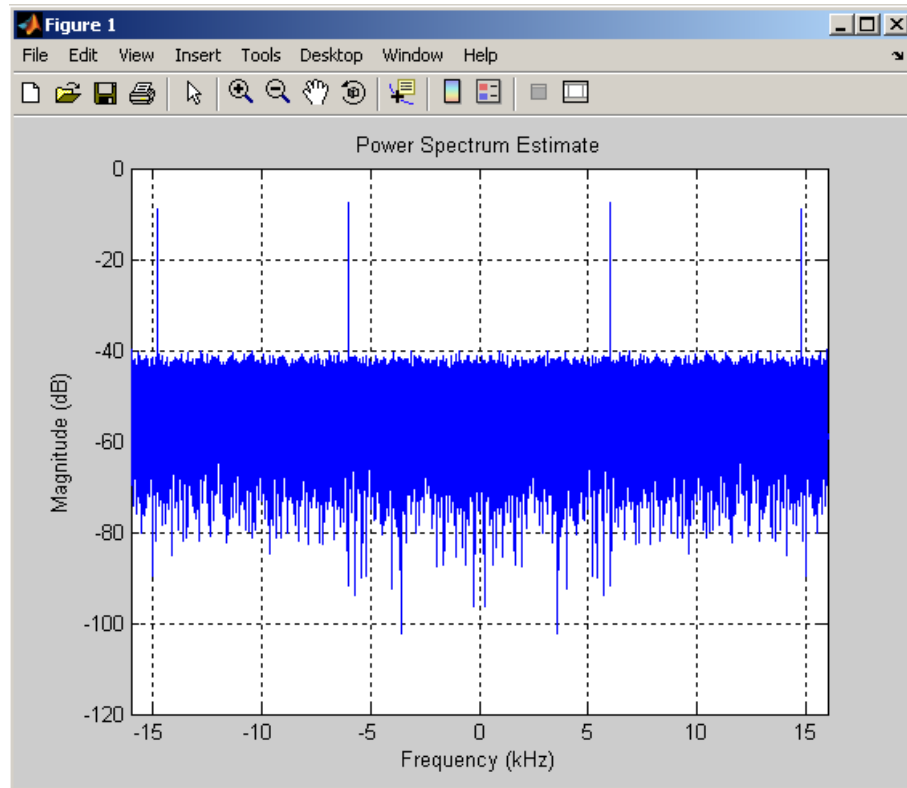
<b>Method</b>	<b>Description</b>
<code>plot</code>	<p>Displays the data graphically in the current figure window.</p> <p>For a <code>dspdata.psd</code> object, it displays the power spectral density in dB/Hz.</p> <p>For a <code>dspdata.msspectrum</code> object, it displays the mean-square in dB.</p> <p>For a <code>dspdata.pseudospectrum</code> object, it displays the pseudospectrum in dB.</p>
<code>twosided</code>	<p><code>twosided(Hs)</code> converts the <code>Hs</code> spectrum to a spectrum calculated over the whole Nyquist interval. All associated properties affected by the new frequency range are adjusted automatically. This method is used for <code>dspdata.psd</code> and <code>dspdata.msspectrum</code> objects.</p> <p>Note that if your data is nonuniformly sampled, converting from <code>onesided</code> to <code>twosided</code> may produce incorrect results.</p>
<code>wholerange</code>	<p><code>wholerange(Hs)</code> converts the <code>Hs</code> spectrum to a spectrum calculated over the whole Nyquist interval. All associated properties affected by the new frequency range are adjusted automatically. This method is used for <code>dspdata.pseudospectrum</code> objects.</p> <p>Note that if your data is nonuniformly sampled, converting from <code>half</code> to <code>wholerange</code> may produce incorrect results.</p>

For more information on each method, use the syntax `help dspdata/method` at the MATLAB prompt.

## Plotting a dspdata Object

The plot method displays the dspdata object spectrum in a separate figure window.

```
plot(Hs) % Plots an existing Hs object
```



## Modifying a dspdata Object

After you create a dspdata object, you can use any of the methods in the table above to modify the object properties.

For example, to change the object from two-sided to one-sided, use

```
onesided(Hs)
```

The Hs object is modified.



**Examples**

See the `msspectrum`, `psd`, or `pseudospectrum` reference pages for specific examples.

**See Also**

`dspdata.msspectrum`, `dspdata.psd`, `dspdata.pseudospectrum`

# dspdata.msspectrum

---

**Purpose** Mean-square (power) spectrum

**Syntax**

```
Hmss = dspdata.msspectrum(Data)
Hmss = dspdata.msspectrum(Data,Frequencies)
Hmss = dspdata.msspectrum(...,'Fs',Fs)
Hmss = dspdata.msspectrum(...,'SpectrumType',SpectrumType)
Hmss = dspdata.msspectrum(...,'CenterDC',flag)
```

**Description** The mean-squared spectrum (MSS) is intended for discrete spectra. Unlike the power spectral density (PSD), the peaks in the MSS reflect the power in the signal at a given frequency. The MSS of a signal is the Fourier transform of that signal's autocorrelation.

`Hmss = dspdata.msspectrum(Data)` uses the mean-square (power) spectrum data contained in `Data`, which can be in the form of a vector or a matrix, where each column is a separate set of data. Default values for other properties of the object are as follows:

Property	Default Value	Description
Name	'Mean-square Spectrum'	Read-only string
Frequencies	[ ] type double	<p>Vector of frequencies at which the spectrum is evaluated. The range of this vector depends on the SpectrumType value. For a one-sided spectrum, the default range is [0, pi) or [0, Fs/2) for odd length, and [0, pi] or [0, Fs/2] for even length, if Fs is specified. For a two-sided spectrum, it is [0, 2pi) or [0, Fs).</p> <p>The length of the Frequencies vector must match the length of the columns of Data.</p> <p>If you do not specify Frequencies, a default vector is created. If one-sided is selected, then the whole number of FFT points (nFFT) for this vector is assumed to be even.</p> <p>If onesided is selected and you specify Frequencies, the last frequency point is compared to the next-to-last point and to pi (or Fs/2, if Fs is specified). If the last point is closer to pi (or Fs/2) than it is to the previous point, nFFT is assumed to be even. If it is closer to the previous point, nFFT is assumed to be odd.</p>

# dspdata.msspectrum

Property	Default Value	Description
Fs	'Normalized'	Sampling frequency, which is 'Normalized' if NormalizedFrequency is true. If NormalizedFrequency is false Fs defaults to 1 Hz.
SpectrumType	'Onesided'	Nyquist interval over which the spectral density is calculated. Valid values are 'Onesided' and 'Twosided'. See the onesided and twosided methods in dspdata for information on changing this property.  The interval for Onesided is $[0 \pi)$ or $[0 \pi]$ depending on the number of FFT points, and for Twosided the interval is $[0 2\pi)$ .
NormalizedFrequency	true	Whether the frequency is normalized (true) or not (false). This property is set automatically at construction time based on Fs. If Fs is specified, NormalizedFrequency is set to false. See the normalizefreq method in dspdata for information on changing this property.

`Hmss = dspdata.msspectrum(Data, Frequencies)` uses the mean-square spectrum data contained in `Data` and `Frequencies` vectors.

`Hmss = dspdata.msspectrum(..., 'Fs', Fs)` uses the sampling frequency `Fs`. Specifying `Fs` uses a default set of linear frequencies (in Hz) based on `Fs` and sets `NormalizedFrequency` to false.

`Hmss = dspdata.msspectrum(..., 'SpectrumType', SpectrumType)` uses the `SpectrumType` string to specify the interval over which the mean-square spectrum was calculated. For data that ranges from  $[0 \pi)$  or  $[0 \pi]$ , set the

SpectrumType to onesided; for data that ranges from  $[0, 2\pi)$ , set the SpectrumType to twosided.

Hmss = dspdata.msspectrum(..., 'CenterDC', flag) uses the value of flag to indicate whether the zero-frequency (DC) component is centered. If flag is true, it indicates that the DC component is in the center of the two-sided spectrum. Set the flag to false if the DC component is on the left edge of the spectrum.

## Methods

Methods provide ways of performing functions directly on your dspdata object without having to specify the parameters again. You can apply a method directly on the variable you assigned to your dspdata.msspectrum object. You can use the following methods with a dspdata.msspectrum object.

- centerdc
- normalizefreq
- onesided
- plot
- twosided

For example, to normalize the frequency and set the NormalizedFrequency parameter to true, use

```
Hmss = normalizefreq(Hs)
```

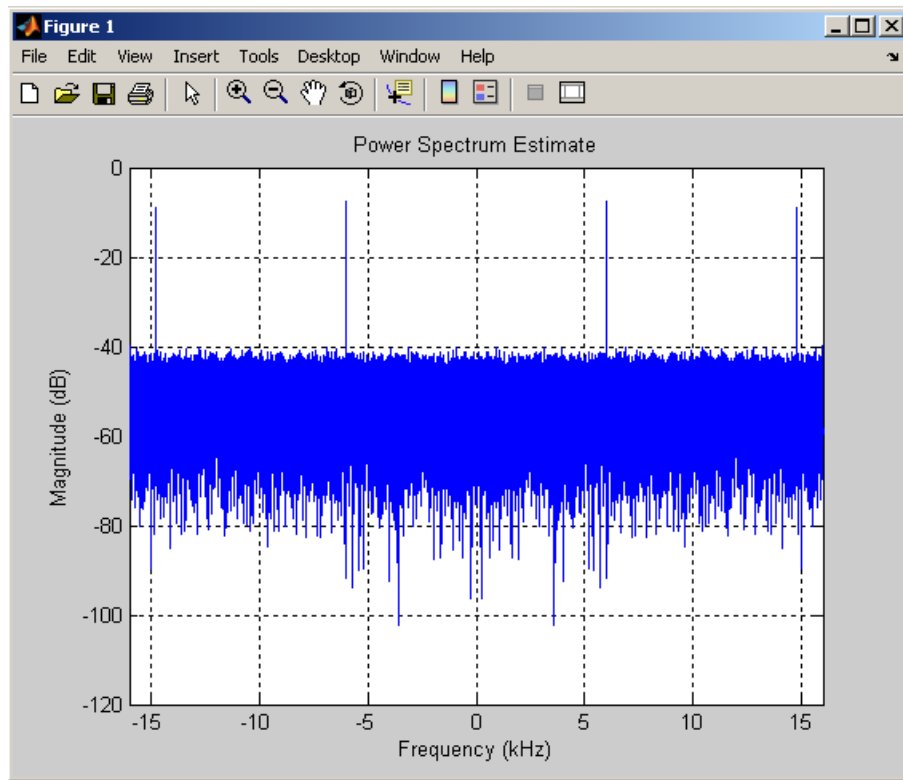
For detailed information on using the methods and plotting the spectrum, see the dspdata reference page.

## Examples

This example shows how to view the spectral content of two sinusoids with random noise.

```
Fs = 32e3;  
t = 0:1/Fs:2.96;  
x = cos(2*pi*t*1.24e3)+cos(2*pi*t*10e3)+randn(size(t));  
X = fft(x);  
P = (abs(X)/length(x)).^2;      % Compute the mean-square.  
  
% Create data object.  
Hmss = dspdata.msspectrum(P, 'Fs', Fs, 'centerdc', true);  
plot(Hmss);                    % Plot the mean-square spectrum.
```

# dspdata.msspectrum



**See Also** [dspdata.psd](#), [dspdata.pseudospectrum](#), [spectrum](#)

**Purpose** Power spectral density

**Syntax**

```
Hpsd = dspdata.psd(Data)
Hpsd = dspdata.psd(Data,Frequencies)
Hpsd = dspdata.psd(...,'Fs',Fs)
Hpsd = dspdata.psd(...,'SpectrumType',SpectrumType)
Hpsd = dspdata.psd(...,'CenterDC',flag)
```

**Description** The power spectral density (PSD) is intended for continuous spectra. The integral of the PSD over a given frequency band computes the average power in the signal over that frequency band. In contrast to the mean-squared spectrum, the peaks in this spectra do not reflect the power at a given frequency. See the `avgpower` method of `dspdata` for more information.

A one-sided PSD contains the total power of the signal in the frequency interval from DC to half of the Nyquist rate. A two-sided PSD contains the total power in the frequency interval from DC to the Nyquist rate.

`Hpsd = dspdata.psd(Data)` uses the power spectral density data contained in `Data`, which can be in the form of a vector or a matrix, where each column is a separate set of data. Default values for other properties of the object are shown below:

Property	Default Value	Description
Name	'Power Spectral Density'	Read-only string
Frequencies	[] type double	<p>Vector of frequencies at which the power spectral density is evaluated. The range of this vector depends on the SpectrumType value. For one-sided, the default range is [0, pi) or [0, Fs/2) for odd length, and [0, pi] or [0, Fs/2] for even length, if Fs is specified. For two-sided, it is [0, 2pi) or [0, Fs).</p> <p>If you do not specify Frequencies, a default vector is created. If one-sided is selected, then the whole number of FFT points (nFFT) for this vector is assumed to be even.</p> <p>If onesided is selected and you specify Frequencies, the last frequency point is compared to the next-to-last point and to pi (or Fs/2, if Fs is specified). If the last point is closer to pi (or Fs/2) than it is to the previous point, nFFT is assumed to be even. If it is closer to the previous point, nFFT is assumed to be odd.</p> <p>The length of the Frequencies vector must match the length of the columns of Data.</p>
Fs	'Normalized'	Sampling frequency, which is 'Normalized' if NormalizedFrequency is true. If NormalizedFrequency is false Fs defaults to 1.



Property	Default Value	Description
SpectrumType	'Onesided'	<p>Nyquist interval over which the power spectral density is calculated. Valid values are 'Onesided' and 'Twosided'. A one-sided PSD contains the total signal power in half the Nyquist interval. See the onesided and twosided methods in dspdata for information on changing this property.</p> <p>The range for half the Nyquist interval is <math>[0 \pi)</math> or <math>[0 \pi]</math> depending on the number of FFT points. For the whole Nyquist interval, the range is <math>[0 2\pi)</math>.</p>
NormalizedFrequency	true	<p>Whether the frequency is normalized (true) or not (false). This property is set automatically at construction time based on Fs. If Fs is specified, NormalizedFrequency is set to false. See the normalizefreq method in dspdata for information on changing this property.</p>

`Hpsd = dspdata.psd(Data,Frequencies)` uses the power spectral density estimation data contained in `Data` and `Frequencies` vectors.

`Hpsd = dspdata.psd(...,'Fs',Fs)` uses the sampling frequency `Fs`. Specifying `Fs` uses a default set of linear frequencies (in Hz) based on `Fs` and sets `NormalizedFrequency` to false.

`Hpsd = dspdata.psd(...,'SpectrumType',SpectrumType)` uses the `SpectrumType` string to specify the interval over which the power spectral density was calculated. For data that ranges from  $[0 \pi)$  or  $[0 \pi]$ , set the `SpectrumType` to `onesided`; for data that ranges from  $[0 2\pi)$ , set the `SpectrumType` to `twosided`.

`Hpsd = dspdata.psd(..., 'CenterDC', flag)` uses the value of `flag` to indicate whether the zero-frequency (DC) component is centered. If `flag` is `true`, it indicates that the DC component is in the center of the two-sided spectrum. Set the `flag` to `false` if the DC component is on the left edge of the spectrum.

## Methods

Methods provide ways of performing functions directly on your `dspdata` object. You can apply a method directly on the variable you assigned to your `dspdata.psd` object. You can use the following methods with a `dspdata.psd` object.

- `avgpower`
- `centerdc`
- `normalizefreq`
- `onesided`
- `plot`
- `twosided`

For example, to normalize the frequency and set the `NormalizedFrequency` parameter to `true`, use

```
Hpsd = normalizefreq(Hpsd)
```

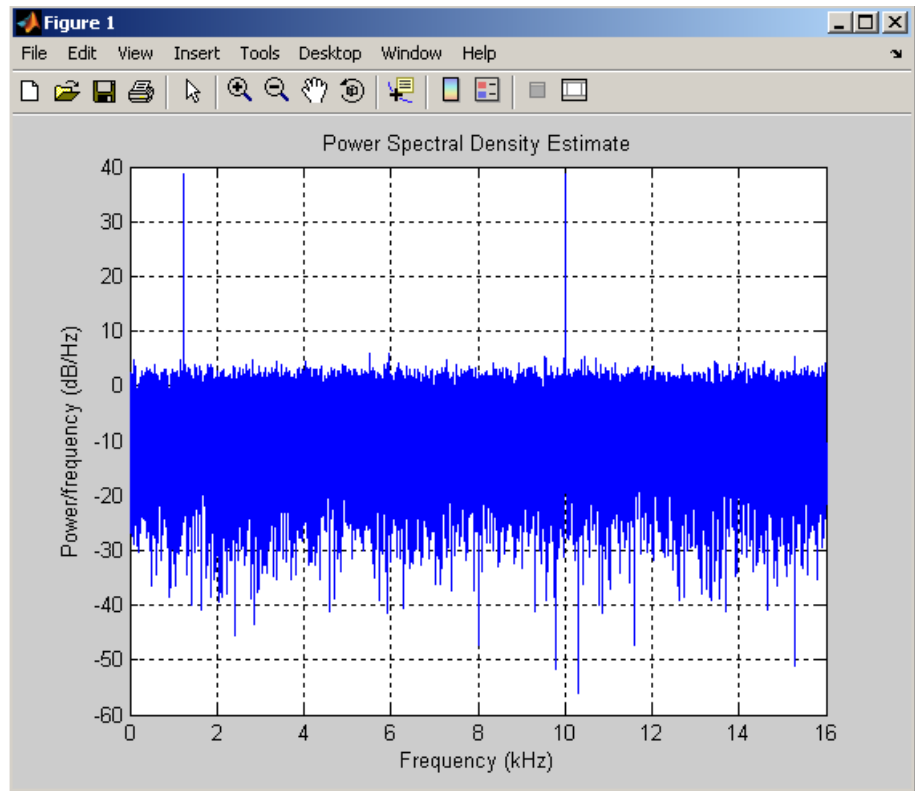
For detailed information on using the methods and plotting the spectrum, see the `dspdata` reference page.

## Examples

### Resolving Signal Components

Use the periodogram to estimate the power spectral density of a noisy sinusoidal signal with two frequency components and then store the results in a PSD data object and plot it.

```
Fs = 32e3;  
t = 0:1/Fs:2.96;  
x = cos(2*pi*t*1.24e3) + cos(2*pi*t*10e3) + randn(size(t));  
Pxx = periodogram(x);  
Hpsd = dspdata.psd(Pxx, 'Fs', Fs); % Create a PSD data object.  
plot(Hpsd); % Plot the PSD data object.
```

**See Also**

`dspdata.msspectrum`, `dspdata.pseudospectrum`, `spectrum`

# dspdata.pseudospectrum

---

**Purpose** Pseudospectrum dspdata object

**Syntax**

```
Hps = dspdata.pseudospectrum(Data)
Hps = dspdata.pseudospectrum(Data,Frequencies)
Hps = dspdata.pseudospectrum(...,'Fs',Fs)
Hps = dspdata.pseudospectrum(...,'SpectrumRange',SpectrumRange)
Hps = dspdata.pseudospectrum(...,'CenterDC',flag)
```

**Description** A pseudospectrum is an indicator of the presence of sinusoidal components in a signal.

`Hps = dspdata.pseudospectrum(Data)` uses the pseudospectrum data contained in `Data`, which can be in the form of a vector or a matrix, where each column is a separate set of data. Default values for other properties of the object are:

Property	Default Value	Description
Name	'Pseudospectrum'	Read-only string
Frequencies	[ ] type double	<p>Vector of frequencies at which the power spectral density is evaluated. The range of this vector depends on the SpectrumRange value. For half, the default range is [0, pi] or [0, Fs/2] for odd length, and [0, pi] or [0, Fs/2] for even length, if Fs is specified. For whole, it is [0, 2pi] or [0, Fs).</p> <p>If you do not specify Frequencies, a default vector is created. If half the Nyquist range is selected, then the whole number of FFT points (nFFT) for this vector is assumed to be even.</p> <p>If half the Nyquist range is selected and you specify Frequencies, the last frequency point is compared to the next-to-last point and to pi (or Fs/2, if Fs is specified). If the last point is closer to pi (or Fs/2) than it is to the previous point, nFFT is assumed to be even. If it is closer to the previous point, nFFT is assumed to be odd.</p> <p>The length of the Frequencies vector must match the length of the columns of Data.</p>
Fs	`Normalized'	<p>Sampling frequency, which is 'Normalized' if NormalizedFrequency is true. If NormalizedFrequency is false Fs defaults to 1.</p>

# dspdata.pseudospectrum

Property	Default Value	Description
SpectrumRange	'Half'	<p>Nyquist interval over which the pseudospectrum is calculated. Valid values are 'Half' and 'Whole'. See the half and whole methods in dspdata for information on changing this property.</p> <p>The interval for Half is <math>[0 \pi)</math> or <math>[0 \pi]</math> depending on the number of FFT points, and for Whole the interval is <math>[0 2\pi)</math>.</p>
NormalizedFrequency	true	<p>Whether the frequency is normalized (true) or not (false). This property is set automatically at construction time based on Fs. If Fs is specified, NormalizedFrequency is set to false. See the normalizefreq method in dspdata for information on changing this property.</p>

Hps = dspdata.pseudospectrum(Data, Frequencies) uses the pseudospectrum estimation data contained in the Data and Frequencies vectors.

Hps = dspdata.pseudospectrum(..., 'Fs', Fs) uses the sampling frequency Fs. Specifying Fs uses a default set of linear frequencies (in Hz) based on Fs and sets NormalizedFrequency to false.

Hps = dspdata.pseudospectrum(..., 'SpectrumRange', SpectrumRange) uses the SpectrumRange string to specify the interval over which the pseudospectrum was calculated. For data that ranges from  $[0 \pi)$  or  $[0 \pi]$ , set the SpectrumRange to half; for data that ranges from  $[0 2\pi)$ , set the SpectrumRange to whole.

Hps = dspdata.pseudospectrum(..., 'CenterDC', flag) uses the value of flag to indicate whether the zero-frequency (DC) component is centered. If

flag is true, it indicates that the DC component is in the center of the whole Nyquist range spectrum. Set the flag to false if the DC component is on the left edge of the spectrum.

## Methods

Methods provide ways of performing functions directly on your dspdata object. You can apply a method directly on the variable you assigned to your dspdata.pseudospectrum object. You can use the following methods with a dspdata.pseudospectrum object.

- centerdc
- halfrange
- normalizefreq
- plot
- wholerange

For example, to normalize the frequency and set the NormalizedFrequency parameter to true, use

```
Hps = normalizefreq(Hps)
```

For detailed information on using the methods and plotting the pseudospectrum, see the dspdata reference page.

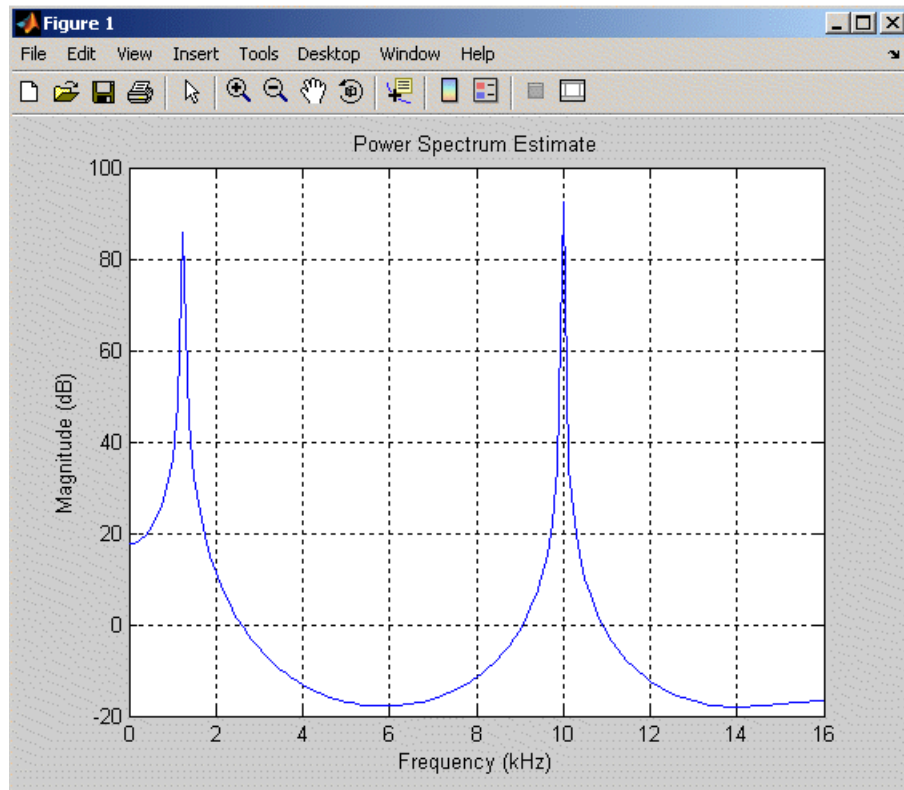
## Examples

### Storing and Plotting Pseudospectrum Data

Use eigenanalysis to estimate the pseudospectrum of a noisy sinusoidal signal with two frequency components. Then store the results in a pseudospectrum data object and plot it.

```
Fs = 32e3;  
t = 0:1/Fs:2.96;  
x = cos(2*pi*t*1.24e3) + cos(2*pi*t*10e3) + randn(size(t));  
P = pmusic(x,4);  
hps = dspdata.pseudospectrum(P,'Fs',Fs); % Create data object.  
plot(hps); % Plot the pseudospectrum.
```

# dspdata.pseudospectrum



## See Also

`dspdata.msspectrum`, `dspdata.psd`, `spectrum`



---

**Purpose** Open FDATool Realize Model panel to create Simulink filter block

**Syntax** dspfwiz

---

**Description** **Note** You must have Simulink installed to use this function.

---

dspfwiz opens FDATool with the Realize Model panel displayed. See “Exporting to Simulink” on page 5-40 for information on using this panel.

Use other panels in FDATool to design your filter and then use the Realize Model panel to create your filter as a subsystem block, which is a combination of Sum, Gain, and Integer Delay blocks, in a Simulink model.

If you also have the Signal Processing Blockset installed, you can create a Digital Filter block instead of a subsystem block, by deselecting the **Build model using basic elements** check box. For more information on the differences between these types of blocks, see Choosing Between Filter Design Blocks in the Signal Blockset documentation.

**See Also** fdatool, realizemdl

# dsptopts

---

**Purpose** Spectral estimation parameter information

**Syntax** `Hopts = dsptopts.optsobj`

**Description** `Hopts = dsptopts.optsobj` returns a DSP options object `Hopts` of type `optsobj`. This DSP options object contains optional parameter information for the specified type of `optsobj`.

You create options objects via methods on other objects. For example,

```
Hopts = psdopts(Hs,X)
```

returns a spectrum object `Hopts` options object. You can change the values of the `Hopts` object and then use this modified options object as an input to the `psd` method of a spectrum object:

```
Hpsd = psd(Hs,X,Hopts)
```

---

**Note** You must use an `optsobj` with `dsptopts`.

---

## Options Objects

An options object (`optsobj`) for `dsptopts` contain optional parameter values for the particular `optsobj`. Available `optsobj`s for `dsptopts` are as follows:

- `spectrum` — Spectrum options for any spectrum estimation method object. The spectrum object method to create this options object is `psdopts` and `msspectrumopts`. See `spectrum` for more information.  
Properties and valid values of a `dsptopts.spectrum` object are as follows:
  - `'NFFT'` — integer number of FFT points
  - `'NormalizedFrequency'` — whether frequency is normalized (`true`) or not (`false`)
  - `'Fs'` — sampling frequency, used only when `'NormalizedFrequency'` is `false`. If `'NormalizedFrequency'` is `true`, the value of `Fs` is `'Normalized'`.
  - `'SpectrumType'` — `'Onesided'` for half the Nyquist interval or `'Twosided'` for the whole Nyquist interval.

- pseudospectrum — Pseudospectrum options for `spectrum.music` and `spectrum.eigenvector` objects. The `spectrum` object method to create this options object is `pseudospectrumopts`. See `spectrum` for more information. Properties and valid values of a `dspopts.pseudospectrum` object are as follows:
  - 'NFFT' — integer number of FFT points
  - 'NormalizedFrequency' — whether frequency is normalized (`true`) or not (`false`)
  - 'Fs' — sampling frequency, used only when 'NormalizedFrequency' is `false`. If 'NormalizedFrequency' is `true`, the value of `Fs` is 'Normalized'.
  - 'SpectrumRange' — 'Half' for half the Nyquist interval or 'Whole' for the whole Nyquist interval.

### Modifying a DSPOPTS Object

To set specific properties, use

```
set(Hopts, 'property1', value, 'property2', value, ...)
```

or use the dot method

```
Hopts.property = value
```

where 'property1', 'property2', and `property` are the specific property names.

To view the options for a property use `set` without specifying a value

```
set(Hopts, 'property')
```

Note that you must use single quotation marks around the property name if you are not using the dot method.

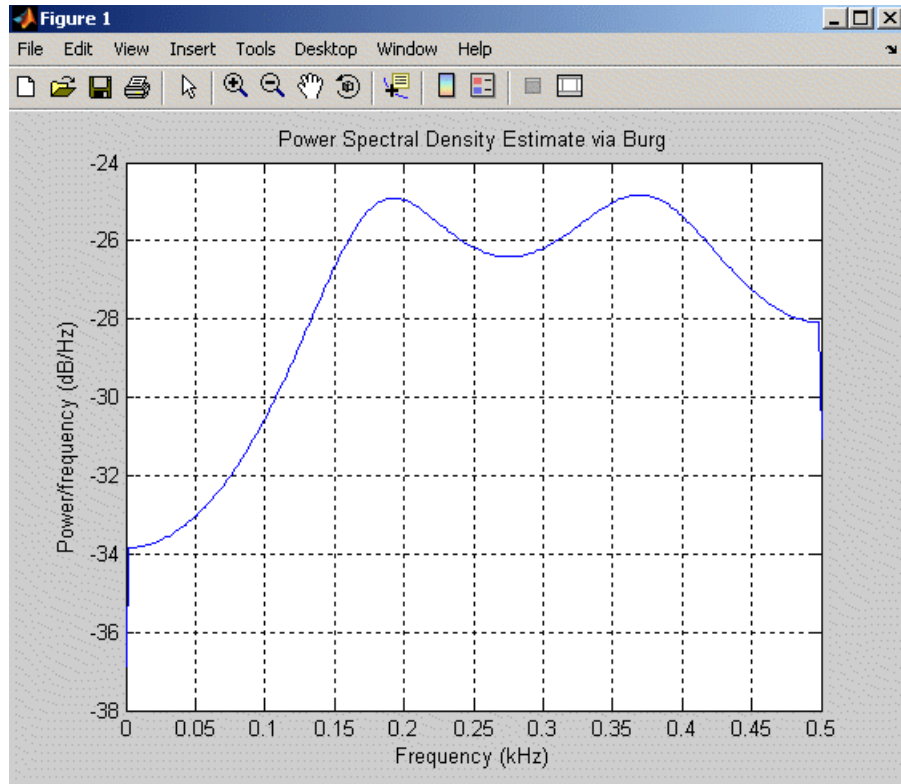
Another way to change an object's properties is by using the `inspect` command, which opens the **Property Inspector** window where you can edit any property, except dynamic properties.

```
inspect(Hopts)
```

## Examples

Define a fourth order auto-regressive model and view its power spectral density using the Burg algorithm. Use `dspopts` to specify the FFT length and sampling frequency.

```
randn('state',1);  
x=randn(100,1);  
x=filter(1,[1 1/2 1/3 1/4 1/5],x);%Fourth order AR filter  
Hs=spectrum.burg; %Fourth order AR model  
Hopts = dspopts.spectrum;  
set(Hs,'FFTLength','UserDefined');  
set(Hopts,'NFFT',512);  
set(Hopts,'Fs',1000);  
psd(Hs,x,Hopts)
```



**See Also**

`spectrum`, `spectrum.burg`, `spectrum.cov`, `spectrum.eigenvector`,  
`spectrum.mcov`, `spectrum.mtm`, `spectrum.music`, `spectrum.periodogram`,  
`spectrum.welch`, `spectrum.yulear`

# ellip

---

**Purpose** Elliptic (Cauer) filter design

**Syntax**

```
[b,a] = ellip(n,Rp,Rs,Wp)
[b,a] = ellip(n,Rp,Rs,Wp,'ftype')
[b,a] = ellip(n,Rp,Rs,Wp,'s')
[b,a] = ellip(n,Rp,Rs,Wp,'ftype','s')
[z,p,k] = ellip(...)
[A,B,C,D] = ellip(...)
```

**Description** `ellip` designs lowpass, bandpass, highpass, and bandstop digital and analog elliptic filters. Elliptic filters offer steeper rolloff characteristics than Butterworth or Chebyshev filters, but are equiripple in both the pass- and stopbands. In general, elliptic filters meet given performance specifications with the lowest order of any filter type.

## Digital Domain

`[b,a] = ellip(n,Rp,Rs,Wp)` designs an order  $n$  lowpass digital elliptic filter with normalized passband edge frequency  $W_p$ ,  $R_p$  dB of ripple in the passband, and a stopband  $R_s$  dB down from the peak value in the passband. It returns the filter coefficients in the length  $n+1$  row vectors  $b$  and  $a$ , with coefficients in descending powers of  $z$ .

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}}{1 + a(2)z^{-1} + \dots + a(n+1)z^{-n}}$$

The *normalized passband edge frequency* is the edge of the passband, at which the magnitude response of the filter is  $-R_p$  dB. For `ellip`, the normalized cutoff frequency  $W_p$  is a number between 0 and 1, where 1 corresponds to half the sampling frequency (Nyquist frequency). Smaller values of passband ripple  $R_p$  and larger values of stopband attenuation  $R_s$  both lead to wider transition widths (shallower rolloff characteristics).

If  $W_p$  is a two-element vector,  $W_p = [w1 \ w2]$ , `ellip` returns an order  $2*n$  bandpass filter with passband  $w1 < \omega < w2$ .

`[b,a] = ellip(n,Rp,Rs,Wp,'ftype')` designs a highpass, lowpass, or bandstop filter, where the string `'ftype'` is one of the following.

- `'high'` for a highpass digital filter with normalized passband edge frequency `Wp`
- `'low'` for a lowpass digital filter with normalized passband edge frequency `Wp`
- `'stop'` for an order  $2*n$  bandstop digital filter if `Wp` is a two-element vector, `Wp = [w1 w2]`. The stopband is  $w1 < \omega < w2$ .

With different numbers of output arguments, `ellip` directly obtains other realizations of the filter. To obtain zero-pole-gain form, use three output arguments as shown below.

`[z,p,k] = ellip(n,Rp,Rs,Wp)` or

`[z,p,k] = ellip(n,Rp,Rs,Wp,'ftype')` returns the zeros and poles in length `n` column vectors `z` and `p` and the gain in the scalar `k`.

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = ellip(n,Rp,Rs,Wp)` or

`[A,B,C,D] = ellip(n,Rp,Rs,Wp,'ftype')` where `A`, `B`, `C`, and `D` are

$$\begin{aligned}x[n + 1] &= Ax[n] + Bu[n] \\y[n] &= Cx[n] + Du[n]\end{aligned}$$

and `u` is the input, `x` is the state vector, and `y` is the output.

### Analog Domain

`[b,a] = ellip(n,Rp,Rs,Wp,'s')` designs an order `n` lowpass analog elliptic filter with angular passband edge frequency `Wp` rad/s and returns the filter coefficients in the length `n+1` row vectors `b` and `a`, in descending powers of `s`, derived from this transfer function:

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \dots + b(n+1)}{s^n + a(2)s^{n-1} + \dots + a(n+1)}$$

The *angular passband edge frequency* is the edge of the passband, at which the magnitude response of the filter is  $-R_p$  dB. For `ellip`, the angular passband edge frequency  $\omega_p$  must be greater than 0 rad/s.

If  $\omega_p$  is a two-element vector with  $\omega_1 < \omega_2$ , then `ellip(n,Rp,Rs,Wp,'s')` returns an order  $2*n$  bandpass analog filter with passband  $\omega_1 < \omega < \omega_2$ .

`[b,a] = ellip(n,Rp,Rs,Wp,'ftype','s')` designs a highpass, lowpass, or bandstop filter.

With different numbers of output arguments, `ellip` directly obtains other realizations of the analog filter. To obtain zero-pole-gain form, use three output arguments as shown below:

`[z,p,k] = ellip(n,Rp,Rs,Wp,'s')` or

`[z,p,k] = ellip(n,Rp,Rs,Wp,'ftype','s')` returns the zeros and poles in length  $n$  or  $2*n$  column vectors  $z$  and  $p$  and the gain in the scalar  $k$ .

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = ellip(n,Rp,Rs,Wp,'s')` or

`[A,B,C,D] = ellip(n,Rp,Rs,Wp,'ftype','s')` where  $A$ ,  $B$ ,  $C$ , and  $D$  are

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

and  $u$  is the input,  $x$  is the state vector, and  $y$  is the output.

## Examples

### Example 1

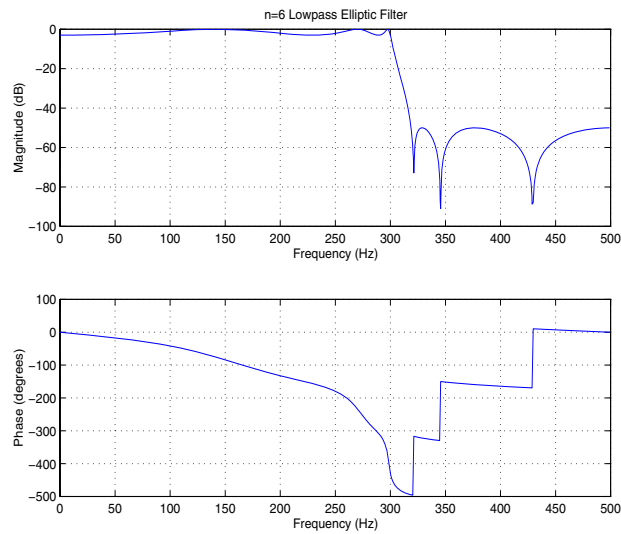
For data sampled at 1000 Hz, design a sixth-order lowpass elliptic filter with a passband edge frequency of 300 Hz, which corresponds to a normalized value of 0.6, 3 dB of ripple in the passband, and 50 dB of attenuation in the stopband:

```
[b,a] = ellip(6,3,50,300/500);
```

The filter's frequency response is

```
freqz(b,a,512,1000)
title('n=6 Lowpass Elliptic Filter')
```

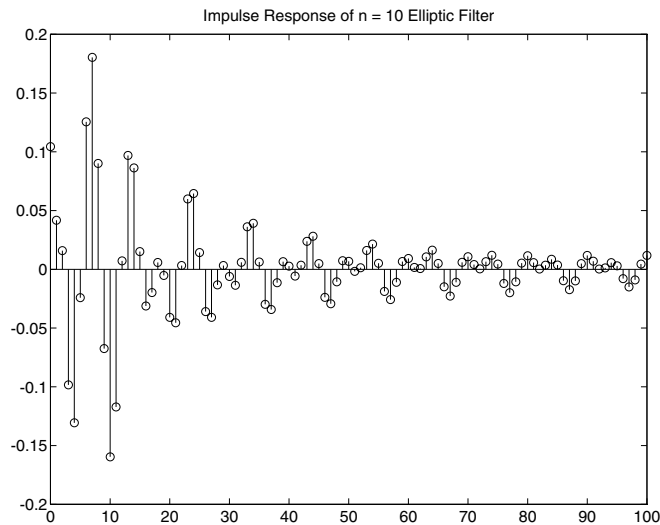




### Example 2

Design a 20th-order bandpass elliptic filter with a passband from 100 to 200 Hz and plot its impulse response:

```
n = 10; Rp = 0.5; Rs = 20;
Wp = [100 200]/500;
[b,a] = ellip(n,Rp,Rs,Wp);
[y,t] = impz(b,a,101); stem(t,y)
title('Impulse Response of n=10 Elliptic Filter')
```



## Limitations

For high order filters, the state-space form is the most numerically accurate, followed by the zero-pole-gain form. The transfer function form is the least accurate; numerical problems can arise for filter orders as low as 15.

## Algorithm

The design of elliptic filters is the most difficult and computationally intensive of the Butterworth, Chebyshev Type I and II, and elliptic designs. `ellip` uses a five-step algorithm:

- 1** It finds the lowpass analog prototype poles, zeros, and gain using the `ellipap` function.
- 2** It converts the poles, zeros, and gain into state-space form.
- 3** It transforms the lowpass filter to a bandpass, highpass, or bandstop filter with the desired cutoff frequencies using a state-space transformation.
- 4** For digital filter design, `ellip` uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at  $w_p$  or  $w_1$  and  $w_2$ .
- 5** It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

**See Also**

besself, butter, cheby1, cheby2, ellipap, ellipord

# ellipap

---

**Purpose** Elliptic analog lowpass filter prototype

**Syntax** `[z,p,k] = ellipap(n,Rp,Rs)`

**Description** `[z,p,k] = ellipap(n,Rp,Rs)` returns the zeros, poles, and gain of an order  $n$  elliptic analog lowpass filter prototype, with  $R_p$  dB of ripple in the passband, and a stopband  $R_s$  dB down from the peak value in the passband. The zeros and poles are returned in length  $n$  column vectors  $z$  and  $p$  and the gain in scalar  $k$ . If  $n$  is odd,  $z$  is length  $n - 1$ . The transfer function is

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

Elliptic filters offer steeper rolloff characteristics than Butterworth and Chebyshev filters, but they are equiripple in both the passband and the stopband. Of the four classical filter types, elliptic filters usually meet a given set of filter performance specifications with the lowest filter order.

`ellip` sets the passband edge angular frequency  $\omega_0$  of the elliptic filter to 1 for a normalized result. The *passband edge angular frequency* is the frequency at which the passband ends and the filter has a magnitude response of  $10^{-R_p/20}$ .

**Algorithm** `ellipap` uses the algorithm outlined in [1]. It employs the M-file `ellipk` to calculate the complete elliptic integral of the first kind and the M-file `ellipj` to calculate Jacobi elliptic functions.

**References** [1] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Chapter 7.

**See Also** `besselap`, `buttap`, `cheb1ap`, `cheb2ap`, `ellip`

**Purpose** Minimum order for elliptic filters

**Syntax**  
`[n,Wp] = ellipord(Wp,Ws,Rp,Rs)`  
`[n,Wp] = ellipord(Wp,Ws,Rp,Rs,'s')`

**Description** `ellipord` calculates the minimum order of a digital or analog elliptic filter required to meet a set of filter design specifications.

### Digital Domain

`[n,Wp] = ellipord(Wp,Ws,Rp,Rs)` returns the lowest order  $n$  of the elliptic filter that loses no more than  $R_p$  dB in the passband and has at least  $R_s$  dB of attenuation in the stopband. The scalar (or vector) of corresponding cutoff frequencies  $W_p$ , is also returned. Use the output arguments  $n$  and  $W_p$  in `ellip`.

Choose the input arguments to specify the stopband and passband according to the following table.

#### Description of Stopband and Passband Filter Parameters

Parameter	Description
$W_p$	Passband corner frequency $W_p$ , the cutoff frequency, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, $\pi$ radians per sample.
$W_s$	Stopband corner frequency $W_s$ , is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.
$R_p$	Passband ripple, in decibels. Twice this value specifies the maximum permissible passband width in decibels.
$R_s$	Stopband attenuation, in decibels. This value is the number of decibels the stopband is attenuated with respect to the passband response.

Use the following guide to specify filters of different types.

## Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	$W_p < W_s$ , both scalars	$(W_s, 1)$	$(0, W_p)$
Highpass	$W_p > W_s$ , both scalars	$(0, W_s)$	$(W_p, 1)$
Bandpass	The interval specified by $W_s$ contains the one specified by $W_p$ $(W_s(1) < W_p(1) < W_p(2) < W_s(2))$ .	$(0, W_s(1))$ and $(W_s(2), 1)$	$(W_p(1), W_p(2))$
Bandstop	The interval specified by $W_p$ contains the one specified by $W_s$ $(W_p(1) < W_s(1) < W_s(2) < W_p(2))$ .	$(0, W_p(1))$ and $(W_p(2), 1)$	$(W_s(1), W_s(2))$

If your filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design separate lowpass and highpass filters according to the specifications in this table, and cascade the two filters together.

## Analog Domain

$[n, W_p] = \text{ellipord}(W_p, W_s, R_p, R_s, 's')$  finds the minimum order  $n$  and cutoff frequencies  $W_p$  for an analog filter. You specify the frequencies  $W_p$  and  $W_s$  similar to those described in the “Description of Stopband and Passband Filter Parameters” table above, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

Use `ellipord` for lowpass, highpass, bandpass, and bandstop filters as described in the “Filter Type Stopband and Passband Specifications” table above.

Examples

Example 1

For 1000 Hz data, design a lowpass filter with less than 3 dB of ripple in the passband defined from 0 to 40 Hz and at least 60 dB of ripple in the stopband defined from 150 Hz to the Nyquist frequency (500 Hz):

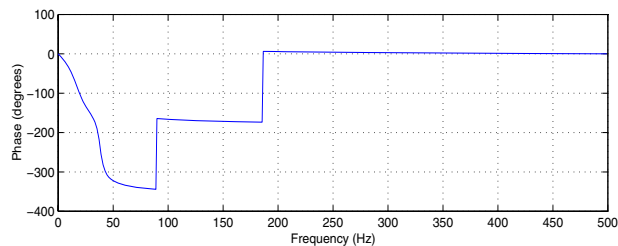
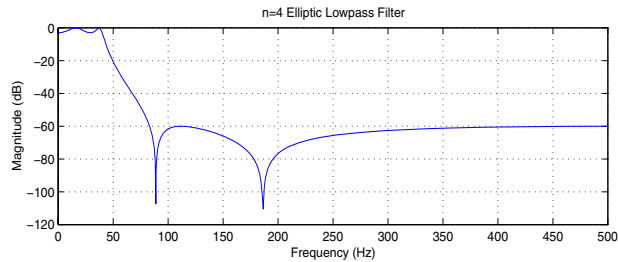
```

Wp = 40/500; Ws = 150/500;
Rp = 3; Rs = 60;
[n,Wp] = ellipord(Wp,Ws,Rp,Rs)

n =
    4

Wp =
    0.0800

[b,a] = ellip(n,Rp,Rs,Wp);
freqz(b,a,512,1000);
title('n=4 Elliptic Lowpass Filter')
    
```



## Example 2

Now design a bandpass filter with a passband from 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband, and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband:

```
Wp = [60 200]/500; Ws = [50 250]/500;
```

```
Rp = 3; Rs = 40;
```

```
[n,Wp] = ellipord(Wp,Ws,Rp,Rs)
```

```
n =
```

```
5
```

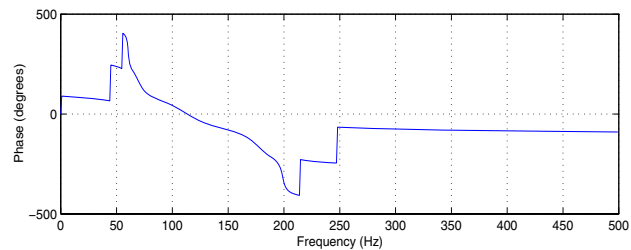
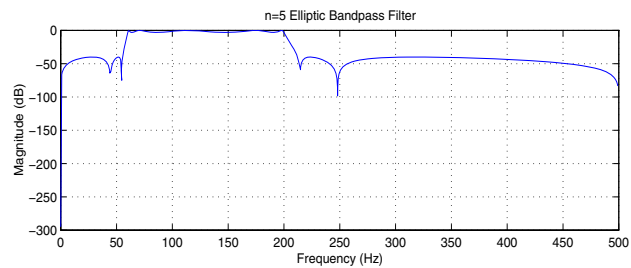
```
Wp =
```

```
0.1200 0.4000
```

```
[b,a] = ellip(n,Rp,Rs,Wp);
```

```
freqz(b,a,512,1000);
```

```
title('n=5 Elliptic Bandpass Filter')
```





**Algorithm**

ellipord uses the elliptic lowpass filter order prediction formula described in [1]. The function performs its calculations in the analog domain for both the analog and digital cases. For the digital case, it converts the frequency parameters to the  $s$ -domain before estimating the order and natural frequencies, and then converts them back to the  $z$ -domain.

ellipord initially develops a lowpass filter prototype by transforming the passband frequencies of the desired filter to 1 rad/s (for low- and highpass filters) and to -1 and 1 rad/s (for bandpass and bandstop filters). It then computes the minimum order required for a lowpass filter to meet the stopband specification.

**References**

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 241.

**See Also**

buttord, cheb1ord, cheb2ord, ellip

# eqtflength

---

**Purpose** Equalize lengths of transfer function's numerator and denominator

**Syntax**  
`[b,a] = eqtflength(num,den)`  
`[b,a,n,m] = eqtflength(num,den)`

**Description** `[b,a] = eqtflength(num,den)` modifies the vector `num` and/or the vector `den`, so that the resulting output vectors `b` and `a` have the same length. The input vectors `num` and `den` may have different lengths. The vector `num` represents the numerator polynomial of a given discrete-time transfer function, and the vector `den` represents its denominator. The resulting numerator `b` and denominator `a` represent the same discrete-time transfer function, but these vectors have the same length.

`[b,a,n,m] = eqtflength(num,den)` modifies the vectors as above and also returns the numerator order `n` and the denominator `m`, not including any trailing zeros.

Use `eqtflength` to obtain a numerator and denominator of equal length before applying transfer function conversion functions such as `tf2ss` and `tf2zp` to discrete-time models.

**Examples**

```
num = [1 0.5];  
den = [1 0.75 0.6 0];  
[b,a,n,m] = eqtflength(num,den)
```

```
b =  
    1.0000    0.5000         0
```

```
a =  
    1.0000    0.7500    0.6000
```

```
n =  
    1
```

```
m =  
    2
```

**Algorithm** `eqtflength(num,den)` appends zeros to either `num` or `den` as necessary. If both `num` and `den` have trailing zeros in common, these are removed.

**See Also** `tf2ss`, `tf2zp`

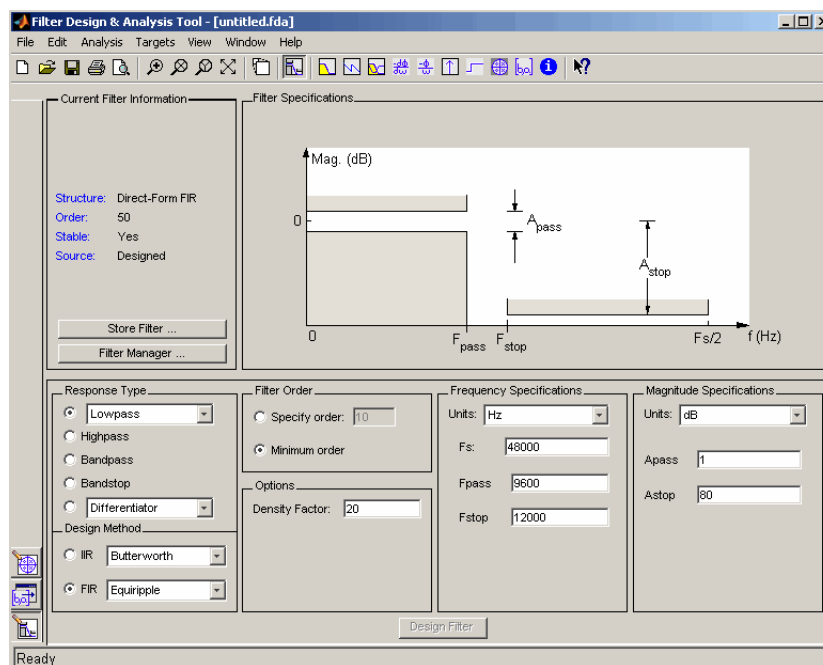
**Purpose** Open Filter Design and Analysis Tool

**Syntax** fdatool

**Description** fdatool opens the Filter Design and Analysis Tool (FDATool). Use this tool to

- Design filters
- Analyze filters
- Modify existing filter designs

See Chapter 5, “FDATool: A Filter Design and Analysis GUI” for more information.



**Remarks** The Filter Design and Analysis Tool provides more design methods than the SPTool Filter Designer. It also integrates advanced filter design methods from the Filter Design Toolbox.

# fdatool

---

---

**Note** The Filter Design and Analysis Tool requires a screen resolution greater than 640 x 480.

---

## See Also

fvtool, sptool, wvtool

---

**Purpose** 1-D fast Fourier transform

**Description** `fft` is a MATLAB function.

# fft2

---

**Purpose** 2-D fast Fourier transform

**Description** fft2 is a MATLAB function.

**Purpose** FFT-based FIR filtering using overlap-add method

**Syntax**  
 $y = \text{fftfilt}(b,x)$   
 $y = \text{fftfilt}(b,x,n)$

**Description** `fftfilt` filters data using the efficient FFT-based method of *overlap-add*, a frequency domain filtering technique that works only for FIR filters.

$y = \text{fftfilt}(b,x)$  filters the data in vector  $x$  with the filter described by coefficient vector  $b$ . It returns the data vector  $y$ . The operation performed by `fftfilt` is described in the *time domain* by the difference equation:

$$y(n) = b(1)x(n) + b(2)x(n-1) + \dots + b(nb+1)x(n-nb)$$

An equivalent representation is the  $z$ -transform or *frequency domain* description:

$$Y(z) = (b(1) + b(2)z^{-1} + \dots + b(nb+1)z^{-nb})X(z)$$

By default, `fftfilt` chooses an FFT length and data block length that guarantee efficient execution time.

If  $x$  is a matrix, `fftfilt` filters its columns. If  $b$  is a matrix, `fftfilt` applies the filter in each column of  $b$  to the signal vector  $x$ . If  $b$  and  $x$  are both matrices with the same number of columns, the  $i$ -th column of  $b$  is used to filter the  $i$ -th column of  $x$ .

$y = \text{fftfilt}(b,x,n)$  uses  $n$  to determine the length of the FFT. See the Algorithm section below for information.

`fftfilt` works for both real and complex inputs.

### Comparison to FILTER function

When the input signal is relatively large, it is advantageous to use `fftfilt` instead of `filter`. `filter` performs  $N$  multiplications for each sample in  $x$ , where  $N$  is the filter length. `fftfilt` performs 2 FFT operations — the FFT of the signal block of length  $L$  plus the inverse FT of the product of the FFTs — at the cost of

$$1/2 * L * \log_2(L)$$

where  $L$  is the block length. It then performs  $L$  pointwise multiplications for a total cost of

$$L + L \cdot \log_2(L) = L \cdot (1 + \log_2(L))$$

multiplications. The cost ratio is therefore

$$L \cdot (1 + \log_2(L)) / (N \cdot L) \Rightarrow (1 + \log_2(L)) / N$$

which is approximately  $\log_2(L) / N$ .

Therefore, `fftfilt` becomes advantageous when  $\log_2(L)$  is less than  $N$ .

## Examples

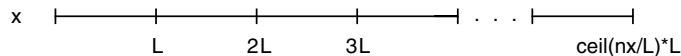
Show that the results from `fftfilt` and `filter` are identical:

```
b = [1 2 3 4];
x = [1 zeros(1,99)]';
norm(fftfilt(b,x) - filter(b,1,x))

ans =
    9.5914e-15
```

## Algorithm

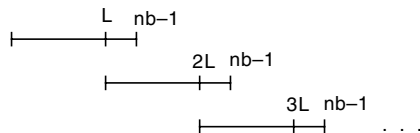
`fftfilt` uses `fft` to implement the *overlap-add method* [1], a technique that combines successive frequency domain filtered blocks of an input sequence. `fftfilt` breaks an input sequence  $x$  into length  $L$  data blocks, where  $L$  must be greater than the filter length  $N$ .



and convolves each block with the filter  $b$  by

```
y = ifft(fft(x(i:i+L-1),nfft).*fft(b,nfft));
```

where `nfft` is the FFT length. `fftfilt` overlaps successive output sections by  $n-1$  points, where  $n$  is the length of the filter, and sums them.





`fftfilt` chooses the key parameters `L` and `nfft` in different ways, depending on whether you supply an FFT length `n` and on the lengths of the filter and signal. If you do not specify a value for `n` (which determines FFT length), `fftfilt` chooses these key parameters automatically:

- If `length(x)` is greater than `length(b)`, `fftfilt` chooses values that minimize the number of blocks times the number of flops per FFT.
- If `length(b)` is greater than or equal to `length(x)`, `fftfilt` uses a single FFT of length

$$2^{\text{nextpow2}(\text{length}(b) + \text{length}(x) - 1)}$$

This essentially computes

$$y = \text{ifft}(\text{fft}(B, \text{nfft}) .* \text{fft}(X, \text{nfft}))$$

If you supply a value for `n`, `fftfilt` chooses an FFT length, `nfft`, of  $2^{\text{nextpow2}(n)}$  and a data block length of `nfft - length(b) + 1`. If `n` is less than `length(b)`, `fftfilt` sets `n` to `length(b)`.

## References

[1] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

## See Also

`conv`, `dfilt.fftfir`, `filter`, `filtfilt`

# fftshift

---

**Purpose** Rearrange FFT function outputs

**Description** `fftshift` is a MATLAB function.

**Purpose** Filter data with a recursive (IIR) or nonrecursive (FIR) filter

**Description** `filter` is a MATLAB function.

**Signal-specific Information** **Filter Method of DFILT**

Filter is also an overloaded method of the discrete-time filter object (`dfilt`). You can pass an object handle, data, and optionally, the dimension into the filter method.

The MATLAB `filter` function describes a `zi` input for initial conditions. Note that the recommended way of passing initial conditions into a `dfilt` is by using the `states` property. For more information, see the `dfilt` reference page.

**Filter Normalization**

Using `filter` on `b` and `a` coefficients normalizes the filter by forcing the  $a_0$  coefficient to be equal to 1.

Using the `filter` method on a `dfilt` object does not normalize the  $a_0$  coefficient.

# filter2

---

**Purpose** 2-D digital filtering

**Description** `filter2` is a MATLAB function.

**Purpose** 2-norm or infinity-norm of a digital filter

**Syntax**  
`filternorm(b,a)`  
`filternorm(b,a,pnorm)`  
`filternorm(b,a,2,tol)`

**Description** A typical use for filter norms is in digital filter scaling to reduce quantization effects. Scaling often improves the signal-to-noise ratio of the filter without resulting in data overflow. You, also, can use the 2-norm to compute the energy of the impulse response of a filter.

`filternorm(b,a)` computes the 2-norm of the digital filter defined by the numerator coefficients in `b` and denominator coefficients in `a`.

`filternorm(b,a,pnorm)` computes the 2- or infinity-norm (inf-norm) of the digital filter, where `pnorm` is either 2 or `inf`.

`filternorm(b,a,2,tol)` computes the 2-norm of an IIR filter with the specified tolerance, `tol`. The tolerance can be specified only for IIR 2-norm computations. `pnorm` in this case must be 2. If `tol` is not specified, it defaults to `1e-8`.

**Examples** Compute the 2-norm with a tolerance of `1e-10` of an IIR filter:

```
[b,a]=butter(5,.5);
L2=filternorm(b,a,2,1e-10)
```

L2 =

0.7071

Compute the inf-norm of an FIR filter:

```
b=firpm(30,[.1 .9],[1 1],'Hilbert');
Linf=filternorm(b,1,inf)
```

Linf =

1.0028

# filternorm

---

## Algorithm

Given a filter  $H(z)$  with frequency response  $H(e^{j\omega})$ , the  $L_p$ -norm is given by

$$\|H\|_p \equiv \left[ \frac{1}{2\pi} \int_{-\pi}^{\pi} |H(e^{j\omega})|^p d\omega \right]^{\frac{1}{p}}$$

For the case  $p = \infty$ , the  $L_\infty$  norm simplifies to

$$\|H\|_\infty = \max_{-\pi \leq \omega \leq \pi} |H(e^{j\omega})|$$

For the case  $p = 2$ , Parseval's theorem states that

$$\|H\|_2 = \left[ \frac{1}{2\pi} \int_{-\pi}^{\pi} |H(e^{j\omega})|^2 d\omega \right]^{\frac{1}{2}} = \left[ \sum_{n=-\infty}^{\infty} |h(n)|^2 \right]^{\frac{1}{2}}$$

where  $h(n)$  is the impulse response of the filter. The energy of the impulse response, then, is  $\|H\|_2^2$ .

## Reference

Jackson, L.B., *Digital Filters and Signal Processing*, Third Edition, Kluwer Academic Publishers, 1996, Chapter 11.

## See Also

zp2sos, norm

---

<b>Purpose</b>	Zero-phase digital filtering
<b>Syntax</b>	$y = \text{filtfilt}(b, a, x)$
<b>Description</b>	<p><math>y = \text{filtfilt}(b, a, x)</math> performs zero-phase digital filtering by processing the input data in both the forward and reverse directions (see problem 5.39 in [1]). After filtering in the forward direction, it reverses the filtered sequence and runs it back through the filter. The resulting sequence has precisely zero-phase distortion and double the filter order. <code>filtfilt</code> minimizes start-up and ending transients by matching initial conditions, and works for both real and complex inputs.</p> <p>Note that <code>filtfilt</code> should not be used with differentiator and Hilbert FIR filters, since the operation of these filters depends heavily on their phase response.</p>
<b>Algorithm</b>	<code>filtfilt</code> is an M-file that uses the <code>filter</code> function. In addition to the forward-reverse filtering, it attempts to minimize startup transients by adjusting initial conditions to match the DC component of the signal and by prepending several filter lengths of a flipped, reflected copy of the input signal.
<b>References</b>	<p>[1] Oppenheim, A.V., and R.W. Schaffer, <i>Discrete-Time Signal Processing</i>, Prentice-Hall, 1989, pp. 311-312.</p> <p>[2] Mitra, S.K., <i>Digital Signal Processing</i>, 2nd ed., McGraw-Hill, 2001, Sections 4.4.2 and 8.2.5.</p> <p>[3] Gustafsson, F., "Determining the initial states in forward-backward filtering," <i>IEEE Transactions on Signal Processing</i>, April 1996, Volume 44, Issue 4, pp. 988--992,</p>
<b>See Also</b>	<code>fftfilt</code> , <code>filter</code> , <code>filter2</code>

# filtic

## Purpose

Initial conditions for transposed direct-form II filter implementation

## Syntax

```
z = filtic(b,a,y,x)
z = filtic(b,a,y)
```

## Description

`z = filtic(b,a,y,x)` finds the initial conditions,  $z$ , for the delays in the *transposed direct-form II* filter implementation given past outputs  $y$  and inputs  $x$ . The vectors  $b$  and  $a$  represent the numerator and denominator coefficients, respectively, of the filter's transfer function.

The vectors  $x$  and  $y$  contain the most recent input or output first, and oldest input or output last.

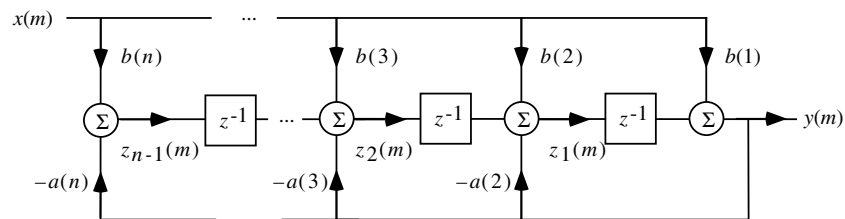
$$x = \{x(-1), x(-2), x(-3), \dots, x(-n), \dots\}$$
$$y = \{y(-1), y(-2), y(-3), \dots, y(-m), \dots\}$$

where  $n$  is  $\text{length}(b) - 1$  (the numerator order) and  $m$  is  $\text{length}(a) - 1$  (the denominator order). If  $\text{length}(x)$  is less than  $n$ , `filtic` pads it with zeros to length  $n$ ; if  $\text{length}(y)$  is less than  $m$ , `filtic` pads it with zeros to length  $m$ . Elements of  $x$  beyond  $x(n-1)$  and elements of  $y$  beyond  $y(m-1)$  are unnecessary so `filtic` ignores them.

Output  $z$  is a column vector of length equal to the larger of  $n$  and  $m$ .  $z$  describes the state of the delays given past inputs  $x$  and past outputs  $y$ .

`z = filtic(b,a,y)` assumes that the input  $x$  is 0 in the past.

The transposed direct-form II structure is shown in the following illustration.



$n-1$  is the filter order.

`filtic` works for both real and complex inputs.



- Algorithm**            `filtic` performs a reverse difference equation to obtain the delay states `z`.
- Diagnostics**        If any of the input arguments `y`, `x`, `b`, or `a` is not a vector (that is, if any argument is a scalar or array), `filtic` gives the following error message:
- Requires vector inputs.
- References**            Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 296, 301-302.
- See Also**              `filter`, `filtfilt`

# filtstates

---

**Purpose** Filter states

**Syntax** `Hs = filtstates.structure(input1,...)`

**Description** `Hs = filtstates.structure(input1,...)` returns a filter states object `Hs`, which contains the filter states.

You can extract a `filtstates` object from the `states` property of an object with

```
Hd = dfilt.df1
Hs = Hd.states
```

or, for an `mfilt` object in the Filter Design Toolbox, with

```
Hm = mfilt.cicdecim
Hs = Hm.states
```

## Structures

Structures for `filtstates` specify the type of filter structure. Available types of structures for `filtstates` are shown below.

<b>filtstates.structure</b>	<b>Description</b>
<code>filtstates.dfir</code>	filtstates for IIR direct-form I filters ( <code>dfilt.df1</code> , <code>dfilt.df1t</code> , <code>dfilt.df1sos</code> , and <code>dfilt.df1tsos</code> )
<code>filtstates.cic</code>	filtstates for cascaded integrator comb filters. (Available only with the Filter Design Toolbox and the Fixed Point Toolbox.)

Refer to the particular `filtstates.structure` reference page or use the syntax `help filtstates.structure` at the MATLAB command line for more information.

**See Also** `filtstates.dfir`, `dfilt`, `dfilt.df1`, `dfilt.df1t`, `dfilt.df1sos`, `dfilt.df1tsos`

**Purpose** IIR direct-form filter states

**Syntax** `Hs = filtstates.dfiir(numstates,denstates)`

**Description** `Hs = filtstates.dfiir(numstates,denstates)` returns an IIR direct-form filter states object `Hs` with two properties—`Numerator` and `Denominator`, which contain the filter states. These two properties are column vectors with each column representing a separate channel of filter states. The number of states is always one less than the number of filter numerator or denominator coefficients.

You can extract a `filtstates` object from the `states` property of an IIR direct-form I object with

```
Hd = dfilt.df1
Hs = Hd.states
```

## Methods

You can use the following methods on a `filtstates.dfiir` object.

Method	Description
<code>double</code>	Converts a <code>filtstates</code> object to a double-precision vector containing the values of the numerator and denominator states. The numerator states are listed first in this vector, followed by the denominator states.
<code>single</code>	Converts a <code>filtstates</code> object to a single-precision vector containing the values of the numerator and denominator states. (This method is used with the Filter Design Toolbox.)

## Examples

This example demonstrates the interaction of `filtstates` with a `dfilt.df1` object.

```
[b,a] = butter(4,0.5); % Design a butterworth filter
Hd = dfilt.df1(b,a); % Create a dfilt object
Hs = Hd.states % Extract a filter states object from
% the dfilt states property
Hs.Numerator = [1,1,1,1]' % Modify the numerator states
```

## filtstates.dfiir

---

```
Hd.states = Hs           % Set the modified states back to the  
                        % original object
```

```
Dbl = double(Hs)        % Create a double vector from states
```

### See Also

filtstates, dfilt, dfilt.df1, dfilt.df1t, dfilt.df1sos, dfilt.df1tsos

**Purpose** Window-based finite impulse response filter design

**Syntax**

```
b = fir1(n,Wn)
b = fir1(n,Wn,'ftype')
b = fir1(n,Wn>window)
b = fir1(n,Wn,'ftype',window)
b = fir1(...,'normalization')
```

**Description** `fir1` implements the classical method of windowed linear-phase FIR digital filter design [1]. It designs filters in standard lowpass, highpass, bandpass, and bandstop configurations. By default the filter is normalized so that the magnitude response of the filter at the center frequency of the passband is 0 dB.

---

**Note** Use `fir2` for windowed filters with arbitrary frequency response.

---

`b = fir1(n,Wn)` returns row vector `b` containing the  $n+1$  coefficients of an order  $n$  lowpass FIR filter. This is a Hamming-window based, linear-phase filter with normalized cutoff frequency  $Wn$ . The output filter coefficients, `b`, are ordered in descending powers of  $z$ .

$$B(z) = b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}$$

$Wn$  is a number between 0 and 1, where 1 corresponds to the Nyquist frequency.

If  $Wn$  is a two-element vector,  $Wn = [w1 \ w2]$ , `fir1` returns a bandpass filter with passband  $w1 < \omega < w2$ .

If  $Wn$  is a multi-element vector,  $Wn = [w1 \ w2 \ w3 \ w4 \ w5 \ \dots \ wn]$ , `fir1` returns an order  $n$  multiband filter with bands  $0 < \omega < w1$ ,  $w1 < \omega < w2$ , ...,  $wn < \omega < 1$ .

By default, the filter is scaled so that the center of the first passband has a magnitude of exactly 1 after windowing.

`b = fir1(n,Wn,'ftype')` specifies a filter type, where '*ftype*' is:

- 'high' for a highpass filter with cutoff frequency  $W_n$ .
- 'stop' for a bandstop filter, if  $W_n = [w1 \ w2]$ . The stopband frequency range is specified by this interval.
- 'DC-1' to make the first band of a multiband filter a passband.
- 'DC-0' to make the first band of a multiband filter a stopband.

`fir1` always uses an even filter order for the highpass and bandstop configurations. This is because for odd orders, the frequency response at the Nyquist frequency is 0, which is inappropriate for highpass and bandstop filters. If you specify an odd-valued  $n$ , `fir1` increments it by 1.

`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 no window is specified, `fir1` uses a Hamming window (see `hamming`) of length  $n+1$ .

`b = fir1(n,Wn,'ftype',window)` accepts both '*ftype*' and `window` parameters.

`b = fir1(...,'normalization')` specifies whether or not the filter magnitude is normalized. The string '*normalization*' can be the following:

- 'scale' (default): Normalize the filter so that the magnitude response of the filter at the center frequency of the passband is 0 dB.
- 'noscale': Do not normalize the filter.

The group delay of the FIR filter designed by `fir1` is  $n/2$ .

## **Algorithm**

`fir1` uses the window method of FIR filter design [1]. If  $w(n)$  denotes a window, where  $1 \leq n \leq N$ , and the impulse response of the ideal filter is  $h(n)$ , where  $h(n)$  is the inverse Fourier transform of the ideal frequency response, then the windowed digital filter coefficients are given by

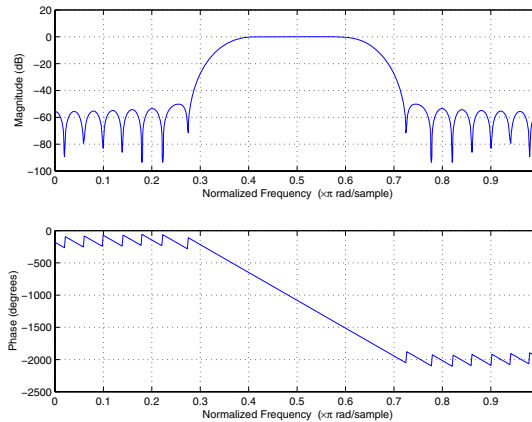
$$b(n) = w(n)h(n), \quad 1 \leq n \leq N$$

## Examples

### Example 1

Design a 48th-order FIR bandpass filter with passband  $0.35 \leq \omega \leq 0.65$ :

```
b = fir1(48,[0.35 0.65]);
freqz(b,1,512)
```



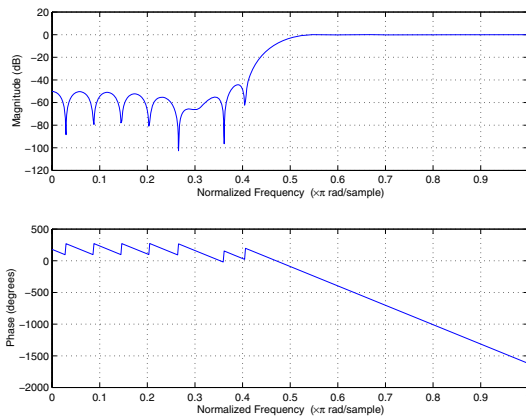
### Example 2

The `chirp.mat` file contains a signal, `y`, that has most of its power above  $f_s/4$ , or half the Nyquist frequency. Design a 34th-order FIR highpass filter to attenuate the components of the signal below  $f_s/4$ . Use a cutoff frequency of 0.48 and a Chebyshev window with 30 dB of ripple:

```
load chirp      % Load y and fs.
b = fir1(34,0.48,'high',chebwin(35,30));
freqz(b,1,512)
```

# **fir1**

---



## **References**

*Programs for Digital Signal Processing*, IEEE Press, New York, 1979.  
Algorithm 5.2.

## **See Also**

`cfirpm`, `filter`, `fir2`, `fircls`, `fircls1`, `firls`, `freqz`, `kaiserord`, `firpm`,  
`window`



**Purpose** Frequency sampling-based finite impulse response filter design

**Syntax**

```
b = fir2(n,f,m)
b = fir2(n,f,m>window)
b = fir2(n,f,m,npt)
b = fir2(n,f,m,npt>window)
b = fir2(n,f,m,npt,lap)
b = fir2(n,f,m,npt,lap>window)
```

**Description** `fir2` designs frequency sampling-based digital FIR filters with arbitrarily shaped frequency response.

---

**Note** Use `fir1` for windows-based standard lowpass, bandpass, highpass, and bandstop configurations.

---

`b = fir2(n,f,m)` returns row vector `b` containing the  $n+1$  coefficients of an order  $n$  FIR filter. The frequency-magnitude characteristics of this filter match those given by vectors `f` and `m`:

- `f` is a vector of frequency points in the range from 0 to 1, where 1 corresponds to the Nyquist frequency. The first point of `f` must be 0 and the last point 1. The frequency points must be in increasing order.
- `m` is a vector containing the desired magnitude response at the points specified in `f`.
- `f` and `m` must be the same length.
- Duplicate frequency points are allowed, corresponding to steps in the frequency response.

Use `plot(f,m)` to view the filter shape.

The output filter coefficients, `b`, are ordered in descending powers of  $z$ .

$$b(z) = b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}$$

`fir2` always uses an even filter order for configurations with a passband at the Nyquist frequency. This is because for odd orders, the frequency response at

the Nyquist frequency is necessarily 0. If you specify an odd-valued  $n$ , `fir2` increments it by 1.

`b = fir2(n,f,m>window)` uses the window specified in the column vector `window`. The vector `window` must be  $n+1$  elements long. If no window is specified, `fir2` uses a Hamming window (see `hamming`) of length  $n+1$ .

`b = fir2(n,f,m,npt)` or

`b = fir2(n,f,m,npt>window)` specifies the number of points, `npt`, for the grid onto which `fir2` interpolates the frequency response, without or with a window specification.

`b = fir2(n,f,m,npt,lap)` and

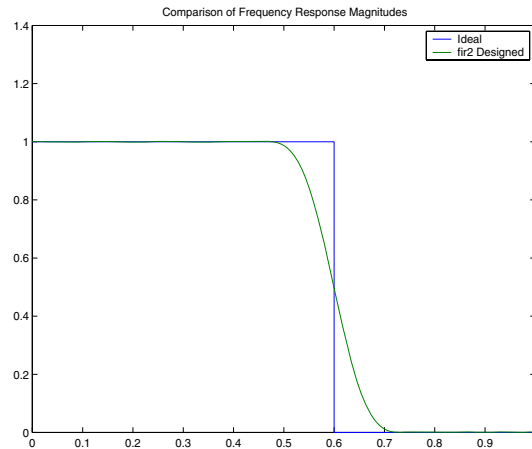
`b = fir2(n,f,m,npt,lap>window)` specify the size of the region, `lap`, that `fir2` inserts around duplicate frequency points, with or without a window specification.

See the “Algorithm” section for more on `npt` and `lap`.

## **Examples**

Design a 30th-order lowpass filter and overplot the desired frequency response with the actual frequency response:

```
f = [0 0.6 0.6 1]; m = [1 1 0 0];  
b = fir2(30,f,m);  
[h,w] = freqz(b,1,128);  
plot(f,m,w/pi,abs(h))  
legend('Ideal','fir2 Designed')  
title('Comparison of Frequency Response Magnitudes')
```



## Algorithm

The desired frequency response is interpolated onto a dense, evenly spaced grid of length `npt`. `npt` is 512 by default. If two successive values of `f` are the same, a region of `lap` points is set up around this frequency to provide a smooth but steep transition in the requested frequency response. By default, `lap` is 25. The filter coefficients are obtained by applying an inverse fast Fourier transform to the grid and multiplying by a window; by default, this is a Hamming window.

## References

Mitra, S.K., *Digital Signal Processing A Computer Based Approach*, First Edition, McGraw-Hill, New York, 1998, pp. 462-468.

Jackson, L.B., *Digital Filters and Signal Processing*, Third Edition, Kluwer Academic Publishers, Boston, 1996, pp. 301-307.

## See Also

`butter`, `cheby1`, `cheby2`, `ellip`, `fir1`, `maxflat`, `firpm`, `yulewalk`

# fircls

---

**Purpose** Constrained least square, FIR multiband filter design

**Syntax**  
`b = fircls(n,f,amp,up,lo)`  
`fircls(n,f,amp,up,lo,'design_flag')`

**Description** `b = fircls(n,f,amp,up,lo)` generates a length  $n+1$  linear phase FIR filter `b`. The frequency-magnitude characteristics of this filter match those given by vectors `f` and `amp`:

- `f` is a vector of transition frequencies in the range from 0 to 1, where 1 corresponds to the Nyquist frequency. The first point of `f` must be 0 and the last point 1. The frequency points must be in increasing order.
- `amp` is a vector describing the piecewise constant desired amplitude of the frequency response. The length of `amp` is equal to the number of bands in the response and should be equal to  $\text{length}(f) - 1$ .
- `up` and `lo` are vectors with the same length as `amp`. They define the upper and lower bounds for the frequency response in each band.

`fircls` always uses an even filter order for configurations with a passband at the Nyquist frequency (that is, highpass and bandstop filters). This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued `n`, `fircls` increments it by 1.

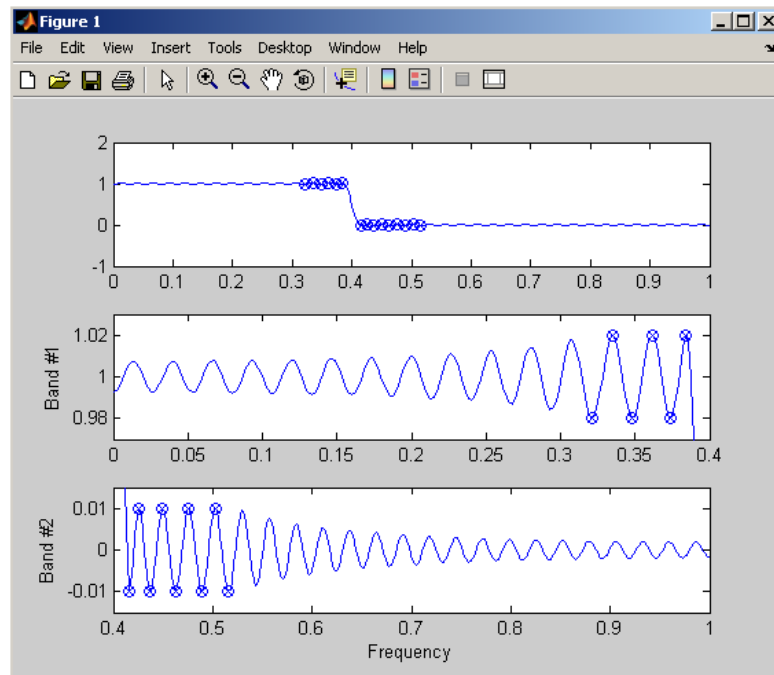
`fircls(n,f,amp,up,lo,'design_flag')` enables you to monitor the filter design, where `'design_flag'` can be

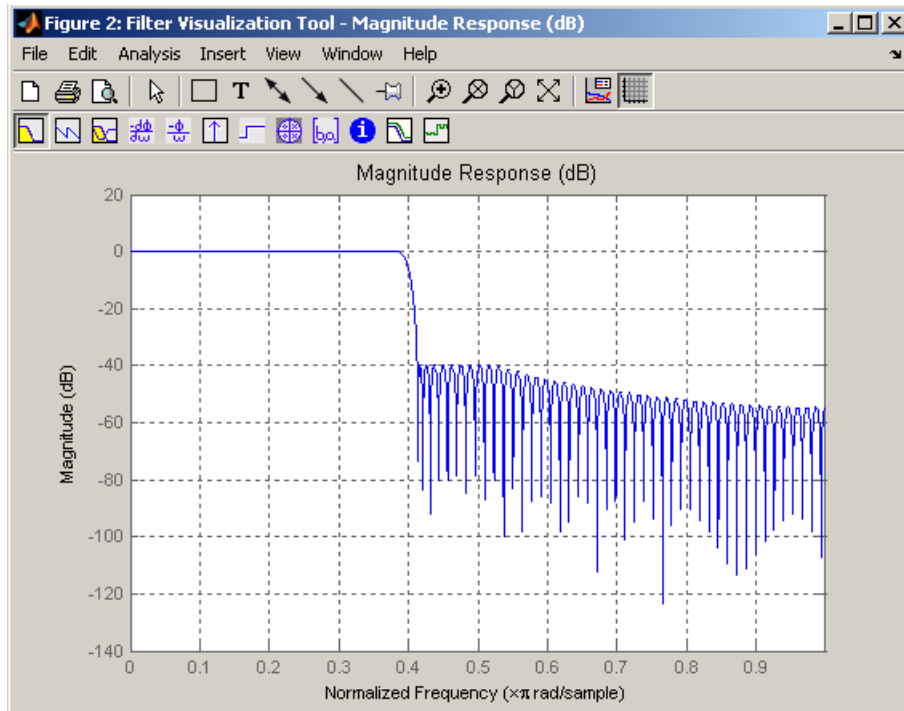
- `'trace'`, for a textual display of the design error at each iteration step.
- `'plots'`, for a collection of plots showing the filter's full-band magnitude response and a zoomed view of the magnitude response in each sub-band. All plots are updated at each iteration step. The O's on the plot are the estimated extremals of the new iteration and the X's are the estimated extremals of the previous iteration, where the extremals are the peaks (maximum and minimum) of the filter ripples. Only ripples that have a corresponding O and X are made equal.
- `'both'`, for both the textual display and plots.

## Examples

Design an order 150 bandpass filter:

```
n=150;
f=[0 0.4 1];
a=[1 0];
up=[1.02 0.01];
lo =[0.98 -0.01];
b = fircls(n,f,a,up,lo,'both'); % Display plots of the bands
Bound Violation = 0.0788344298966
Bound Violation = 0.0096137744998
Bound Violation = 0.0005681345753
Bound Violation = 0.0000051519942
Bound Violation = 0.0000000348656
Bound Violation = 0.0000000006231
% The above Bound Violations indicate iterations as
% the design converges.
fvtool(b) % Display magnitude plot
```





---

**Note** Normally, the lower value in the stopband will be specified as negative. By setting  $l_0$  equal to 0 in the stopbands, a nonnegative frequency response amplitude can be obtained. Such filters can be spectrally factored to obtain minimum phase filters.

---

## Algorithm

`fircls` uses an iterative least-squares algorithm to obtain an equiripple response. The algorithm is a multiple exchange algorithm that uses Lagrange multipliers and Kuhn-Tucker conditions on each iteration.

## References

[1] Selesnick, I.W., M. Lang, and C.S. Burrus, "Constrained Least Square Design of FIR Filters without Specified Transition Bands," *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing*, Vol. 2 (May 1995), pp. 1260-1263.

[2] Selesnick, I.W., M. Lang, and C.S. Burrus. "Constrained Least Square Design of FIR Filters without Specified Transition Bands." *IEEE Transactions on Signal Processing*, Vol. 44, No. 8 (August 1996).

**See Also**

`fircls1`, `firls`, `firpm`

# fircls1

---

**Purpose** Constrained least square, lowpass and highpass, linear phase, FIR filter design

**Syntax**

```
b = fircls1(n,wo,dp,ds)
b = fircls1(n,wo,dp,ds,'high')
b = fircls1(n,wo,dp,ds,wt)
b = fircls1(n,wo,dp,ds,wt,'high')
b = fircls1(n,wo,dp,ds,wp,ws,k)
b = fircls1(n,wo,dp,ds,wp,ws,k,'high')
b = fircls1(n,wo,dp,ds,...,'design_flag')
```

**Description** `b = fircls1(n,wo,dp,ds)` generates a lowpass FIR filter `b`, where `n+1` is the filter length, `wo` is the normalized cutoff frequency in the range between 0 and 1 (where 1 corresponds to the Nyquist frequency), `dp` is the maximum passband deviation from 1 (passband ripple), and `ds` is the maximum stopband deviation from 0 (stopband ripple).

`b = fircls1(n,wo,dp,ds,'high')` generates a highpass FIR filter `b`. `fircls1` always uses an even filter order for the highpass configuration. This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued `n`, `fircls1` increments it by 1.

`b = fircls1(n,wo,dp,ds,wt)` and

`b = fircls1(n,wo,dp,ds,wt,'high')` specifies a frequency `wt` above which (for `wt > wo`) or below which (for `wt < wo`) the filter is guaranteed to meet the given band criterion. This will help you design a filter that meets a passband or stopband edge requirement. There are four cases:

- Lowpass:
  - $0 < wt < wo < 1$ : the amplitude of the filter is within `dp` of 1 over the frequency range  $0 < \omega < wt$ .
  - $0 < wo < wt < 1$ : the amplitude of the filter is within `ds` of 0 over the frequency range  $wt < \omega < 1$ .
- Highpass:
  - $0 < wt < wo < 1$ : the amplitude of the filter is within `ds` of 0 over the frequency range  $0 < \omega < wt$ .



- $0 < \omega_0 < \omega_t < 1$ : the amplitude of the filter is within  $\delta_p$  of 1 over the frequency range  $\omega_t < \omega < 1$ .

$b = \text{fircls1}(n, \omega_0, \delta_p, \delta_s, \omega_p, \omega_s, k)$  generates a lowpass FIR filter  $b$  with a weighted function, where  $n+1$  is the filter length,  $\omega_0$  is the normalized cutoff frequency,  $\delta_p$  is the maximum passband deviation from 1 (passband ripple), and  $\delta_s$  is the maximum stopband deviation from 0 (stopband ripple).  $\omega_p$  is the passband edge of the L2 weight function and  $\omega_s$  is the stopband edge of the L2 weight function, where  $\omega_p < \omega_0 < \omega_s$ .  $k$  is the ratio (passband L2 error)/(stopband L2 error)

$$k = \frac{\int_0^{\omega_p} |A(\omega) - D(\omega)|^2 d\omega}{\int_{\omega_s}^{\pi} |A(\omega) - D(\omega)|^2 d\omega}$$

$b = \text{fircls1}(n, \omega_0, \delta_p, \delta_s, \omega_p, \omega_s, k, \text{'high'})$  generates a highpass FIR filter  $b$  with a weighted function, where  $\omega_s < \omega_0 < \omega_p$ .

$b = \text{fircls1}(n, \omega_0, \delta_p, \delta_s, \dots, \text{'design\_flag'})$  enables you to monitor the filter design, where *'design\_flag'* can be

- *'trace'*, for a textual display of the design table used in the design
- *'plots'*, for plots of the filter's magnitude, group delay, and zeros and poles. All plots are updated at each iteration step. The O's on the plot are the estimated extremals of the new iteration and the X's are the estimated extremals of the previous iteration, where the extremals are the peaks (maximum and minimum) of the filter ripples. Only ripples that have a corresponding O and X are made equal.
- *'both'*, for both the textual display and plots

---

**Note** In the design of very narrow band filters with small  $\delta_p$  and  $\delta_s$ , there may not exist a filter of the given length that meets the specifications.

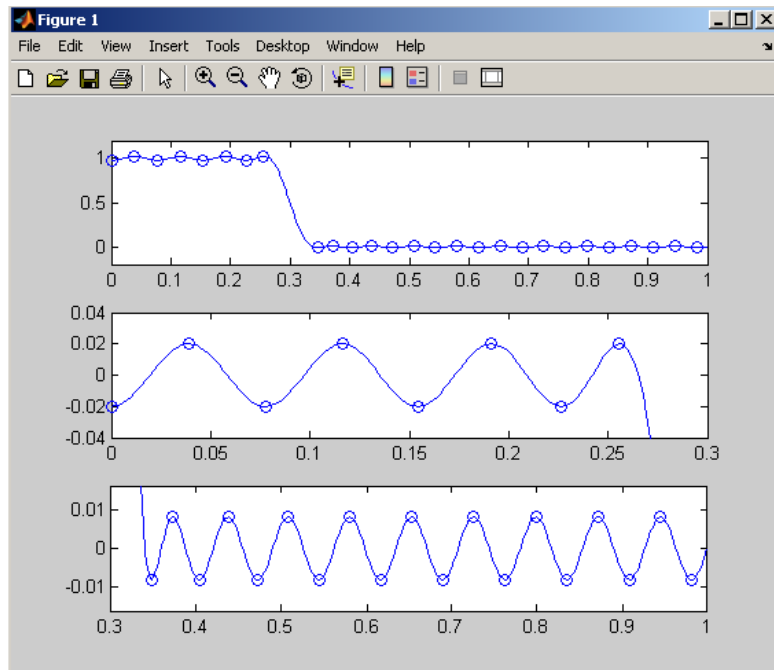
---

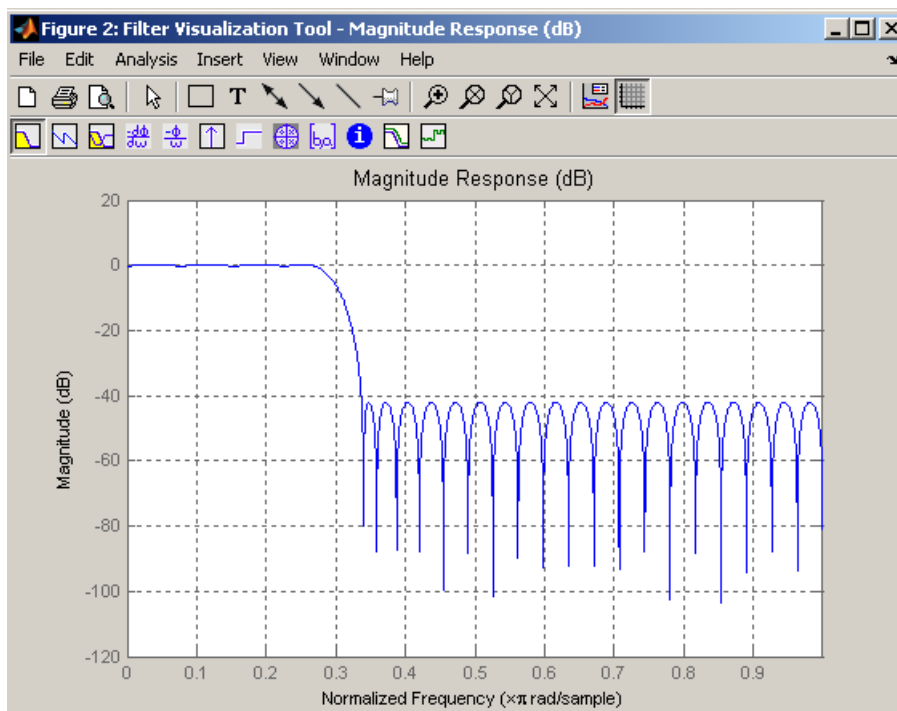
# fircls1

## Examples

Design an order 55 lowpass filter with a cutoff frequency at 0.3:

```
n = 55;      wo = 0.3;
dp = 0.02;   ds = 0.008;
b = fircls1(n,wo,dp,ds,'both'); % Display plots of the bands
Bound Violation = 0.0870385343920
Bound Violation = 0.0149343456540
Bound Violation = 0.0056513587932
Bound Violation = 0.0001056264205
Bound Violation = 0.0000967624352
Bound Violation = 0.0000000226538
Bound Violation = 0.0000000000038
% The above Bound Violations indicate iterations as
% the design converges.
fvtool(b) % Display magnitude plot
```





## Algorithm

`fircls1` uses an iterative least-squares algorithm to obtain an equiripple response. The algorithm is a multiple exchange algorithm that uses Lagrange multipliers and Kuhn-Tucker conditions on each iteration.

## References

- [1] Selesnick, I.W., M. Lang, and C.S. Burrus, "Constrained Least Square Design of FIR Filters without Specified Transition Bands," *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing*, Vol. 2 (May 1995), pp. 1260-1263.
- [2] Selesnick, I.W., M. Lang, and C.S. Burrus, "Constrained Least Square Design of FIR Filters without Specified Transition Bands," *IEEE Transactions on Signal Processing*, Vol. 44, No. 8 (August 1996).

## See Also

`fircls`, `firls`, `firpm`

# firls

---

**Purpose** Least square linear-phase FIR filter design

**Syntax**

```
b = firls(n,f,a)
b = firls(n,f,a,w)
b = firls(n,f,a,'ftype')
b = firls(n,f,a,w,'ftype')
```

**Description** `firls` designs a linear-phase FIR filter that minimizes the weighted, integrated squared error between an ideal piecewise linear function and the magnitude response of the filter over a set of desired frequency bands.

`b = firls(n,f,a)` returns row vector `b` containing the  $n+1$  coefficients of the order  $n$  FIR filter whose frequency-amplitude characteristics approximately match those given by vectors `f` and `a`. The output filter coefficients, or “taps,” in `b` obey the symmetry relation.

$$b(k) = b(n + 2 - k), \quad k = 1, \dots, n + 1$$

These are type I ( $n$  odd) and type II ( $n$  even) linear-phase filters. Vectors `f` and `a` specify the frequency-amplitude characteristics of the filter:

- `f` is a vector of pairs of frequency points, specified in the range between 0 and 1, where 1 corresponds to the Nyquist frequency. The frequencies must be in increasing order. Duplicate frequency points are allowed and, in fact, can be used to design a filter exactly the same as those returned by the `fir1` and `fir2` functions with a rectangular (`rectwin`) window.
- `a` is a vector containing the desired amplitude at the points specified in `f`.

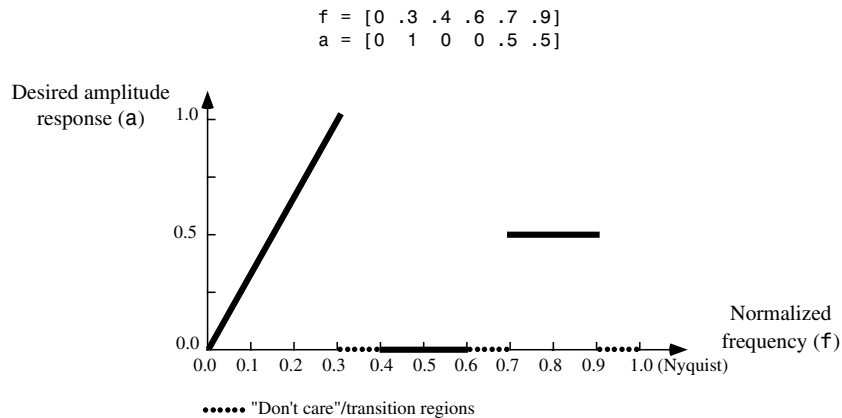
The desired amplitude function at frequencies between pairs of points  $(f(k), f(k+1))$  for  $k$  odd is the line segment connecting the points  $(f(k), a(k))$  and  $(f(k+1), a(k+1))$ .

The desired amplitude function at frequencies between pairs of points  $(f(k), f(k+1))$  for  $k$  even is unspecified. These are transition or “don’t care” regions.

- `f` and `a` are the same length. This length must be an even number.

`firls` always uses an even filter order for configurations with a passband at the Nyquist frequency. This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued `n`, `firls` increments it by 1.

The figure below illustrates the relationship between the `f` and `a` vectors in defining a desired amplitude response.



`b = firls(n,f,a,w)` uses the weights in vector `w` to weight the fit in each frequency band. The length of `w` is half the length of `f` and `a`, so there is exactly one weight per band.

`b = firls(n,f,a,'ftype')` and

`b = firls(n,f,a,w,'ftype')` specify a filter type, where `'ftype'` is:

- `'hilbert'` for linear-phase filters with odd symmetry (type III and type IV). The output coefficients in `b` obey the relation  $b(k) = -b(n+2-k)$ ,  $k = 1, \dots, n + 1$ . This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.
- `'differentiator'` for type III and type IV filters, using a special weighting technique. For nonzero amplitude bands, the integrated squared error has a weight of  $(1/f)^2$  so that the error at low frequencies is much smaller than at high frequencies. For FIR differentiators, which have an amplitude

characteristic proportional to frequency, the filters minimize the relative integrated squared error (the integral of the square of the ratio of the error to the desired amplitude).

## Examples

### Example 1

Design an order 255 lowpass filter with transition band:

```
b = firls(255,[0 0.25 0.3 1],[1 1 0 0]);
```

### Example 2

Design a 31 coefficient differentiator:

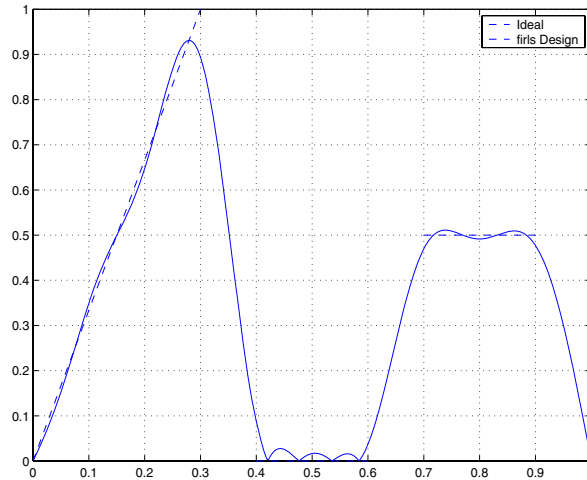
```
b = firls(30,[0 0.9],[0 0.9*pi],'differentiator');
```

An ideal differentiator has the response  $D(w) = jw$ . The amplitudes include a  $\pi$  multiplier because the frequencies are normalized by  $\pi$ .

### Example 3

Design a 24th-order anti-symmetric filter with piecewise linear passbands and plot the desired and actual frequency response:

```
F = [0 0.3 0.4 0.6 0.7 0.9];  
A = [0 1 0 0 0.5 0.5];  
b = firls(24,F,A,'hilbert');  
for i=1:2:6,  
    plot([F(i) F(i+1)],[A(i) A(i+1)],'--'), hold on  
end  
[H,f] = freqz(b,1,512,2);  
plot(f,abs(H)), grid on, hold off  
legend('Ideal','firls Design')
```



**Algorithm**

Reference [1] describes the theoretical approach behind firls. The function solves a system of linear equations involving an inner product matrix of size roughly  $n/2$  using the MATLAB \ operator.

This function designs type I, II, III, and IV linear-phase filters. Type I and II are the defaults for  $n$  even and odd respectively, while the 'hilbert' and 'differentiator' flags produce type III ( $n$  even) and IV ( $n$  odd) filters. The various filter types have different symmetries and constraints on their frequency responses (see [2] for details).

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

# firls

---

## Diagnostics

One of the following diagnostic messages is displayed when an incorrect argument is used:

```
F must be even length.  
F and A must be equal lengths.  
Requires symmetry to be 'hilbert' or 'differentiator'.  
Requires one weight per band.  
Frequencies in F must be nondecreasing.  
Frequencies in F must be in range [0,1].
```

A more serious warning message is

```
Warning: Matrix is close to singular or badly scaled.
```

This tends to happen when the product of the filter length and transition width grows large. In this case, the filter coefficients `b` might not represent the desired filter. You can check the filter by looking at its frequency response.

## References

- [1] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, 1987, pp. 54-83.
- [2] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 256-266.

## See Also

`fir1`, `fir2`, `firrcos`, `firpm`



**Purpose** Parks-McClellan optimal FIR filter design

**Syntax**

```
b = firpm(n,f,a)
b = firpm(n,f,a,w)
b = firpm(n,f,a,'ftype')
b = firpm(n,f,a,w,'ftype')
b = firpm(...,{lgrid})
[b,err] = firpm(...)
[b,err,res] = firpm(...)
b = firpm(n,f,@fresp,w)
b = firpm(n,f,@fresp,w,'ftype')
```

**Description** `firpm` designs a linear-phase FIR filter using the Parks-McClellan algorithm [1]. The Parks-McClellan algorithm uses the Remez exchange algorithm and Chebyshev approximation theory to design filters with an optimal fit between the desired and actual frequency responses. The filters are optimal in the sense that the maximum error between the desired frequency response and the actual frequency response is minimized. Filters designed this way exhibit an equiripple behavior in their frequency responses and are sometimes called *equiripple* filters. `firpm` exhibits discontinuities at the head and tail of its impulse response due to this equiripple nature.

`b = firpm(n,f,a)` returns row vector `b` containing the  $n+1$  coefficients of the order  $n$  FIR filter whose frequency-amplitude characteristics match those given by vectors `f` and `a`.

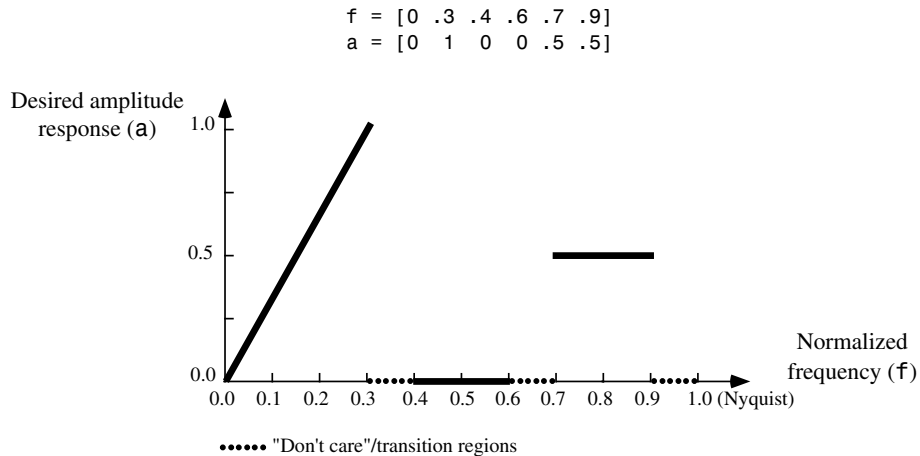
The output filter coefficients (taps) in `b` obey the symmetry relation:

$$b(k) = b(n + 2 - k), \quad k = 1, \dots, n + 1$$

Vectors  $f$  and  $a$  specify the frequency-magnitude characteristics of the filter:

- $f$  is a vector of pairs of normalized frequency points, specified in the range between 0 and 1, where 1 corresponds to the Nyquist frequency. The frequencies must be in increasing order.
- $a$  is a vector containing the desired amplitudes at the points specified in  $f$ . The desired amplitude at frequencies between pairs of points  $(f(k), f(k+1))$  for  $k$  odd is the line segment connecting the points  $(f(k), a(k))$  and  $(f(k+1), a(k+1))$ .  
The desired amplitude at frequencies between pairs of points  $(f(k), f(k+1))$  for  $k$  even is unspecified. The areas between such points are transition or “don’t care” regions.
- $f$  and  $a$  must be the same length. The length must be an even number.

The relationship between the  $f$  and  $a$  vectors in defining a desired frequency response is shown in the illustration below.



`firpm` always uses an even filter order for configurations with a passband at the Nyquist frequency. This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued  $n$ , `firpm` increments it by 1.

`b = firpm(n,f,a,w)` uses the weights in vector `w` to weight the fit in each frequency band. The length of `w` is half the length of `f` and `a`, so there is exactly one weight per band.

---

### Note

`b = firpm(n,f,a,w)` is a synonym for `b = firpm(n,f,@firpmfrf,a,w)`

where, `@firpmfrf` is the predefined frequency response function handle for `firpm`. If desired, you can write your own response function. Use `help private/firpmfrf` for information.

---

`b = firpm(n,f,a,'ftype')` and

`b = firpm(n,f,a,w,'ftype')` specify a filter type, where `'ftype'` is

- `'hilbert'`, for linear-phase filters with odd symmetry (type III and type IV)

The output coefficients in `b` obey the relation  $b(k) = -b(n+2-k)$ ,  $k = 1, \dots, n + 1$ . This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.

For example,

```
h = firpm(30,[0.1 0.9],[1 1],'hilbert');
```

designs an approximate FIR Hilbert transformer of length 31.

- `'differentiator'`, for type III and type IV filters, using a special weighting technique

For nonzero amplitude bands, it weights the error by a factor of  $1/f$  so that the error at low frequencies is much smaller than at high frequencies. For FIR differentiators, which have an amplitude characteristic proportional to frequency, these filters minimize the maximum relative error (the maximum of the ratio of the error to the desired amplitude).

`b = firpm(...,{lgrid})` uses the integer `lgrid` to control the density of the frequency grid, which has roughly  $(lgrid*n)/(2*bw)$  frequency points, where `bw` is the fraction of the total frequency band interval `[0,1]` covered by `f`. Increasing `lgrid` often results in filters that more exactly match an equiripple filter, but that take longer to compute. The default value of 16 is the minimum

value that should be specified for `lgrid`. Note that the `{lgrid}` argument must be a 1-by-1 cell array.

`[b,err] = firpm(...)` returns the maximum ripple height in `err`.

`[b,err,res] = firpm(...)` returns a structure `res` with the following fields.

<code>res.fgrid</code>	Frequency grid vector used for the filter design optimization
<code>res.des</code>	Desired frequency response for each point in <code>res.fgrid</code>
<code>res.wt</code>	Weighting for each point in <code>opt.fgrid</code>
<code>res.H</code>	Actual frequency response for each point in <code>res.fgrid</code>
<code>res.error</code>	Error at each point in <code>res.fgrid</code> ( <code>res.des-res.H</code> )
<code>res.iextr</code>	Vector of indices into <code>res.fgrid</code> for extremal frequencies
<code>res.fextr</code>	Vector of extremal frequencies

You can also use `firpm` to write a function that defines the desired frequency response. The predefined frequency response function handle for `firpm` is `@firpmfrf`, which designs a linear-phase FIR filter.

`b = firpm(n,f,@fresp,w)` returns row vector `b` containing the `n+1` coefficients of the order `n` FIR filter whose frequency-amplitude characteristics best approximate the response returned by function handle `@fresp`. The function is called from within `firpm` with the following syntax.

```
[dh,dw] = fresp(n,f,gf,w)
```

The arguments are similar to those for `firpm`:

- `n` is the filter order.
- `f` is the vector of normalized frequency band edges that appear monotonically between 0 and 1, where 1 is the Nyquist frequency.
- `gf` is a vector of grid points that have been linearly interpolated over each specified frequency band by `firpm`. `gf` determines the frequency grid at which the response function must be evaluated, and contains the same data returned by `cfirpm` in the `fgrid` field of the `opt` structure.

- $w$  is a vector of real, positive weights, one per band, used during optimization.  $w$  is optional in the call to `firpm`; if not specified, it is set to unity weighting before being passed to `fresp`.
- $dh$  and  $dw$  are the desired complex frequency response and band weight vectors, respectively, evaluated at each frequency in `grid`  $gf$ .

`b = firpm(n,f,@fresp,w,'ftype')` designs antisymmetric (odd) filters, where `'ftype'` is either `'d'` for a differentiator or `'h'` for a Hilbert transformer. If you do not specify an `ftype`, a call is made to `fresp` to determine the default symmetry property `sym`. This call is made using the syntax.

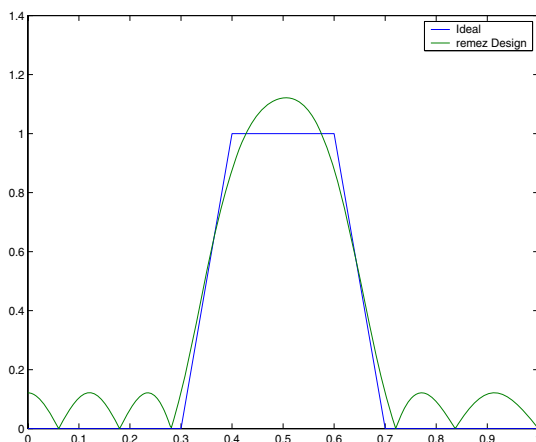
```
sym = fresp('defaults',{n,f,[],w,p1,p2,...})
```

The arguments  $n$ ,  $f$ ,  $w$ , etc., may be used as necessary in determining an appropriate value for `sym`, which `firpm` expects to be either `'even'` or `'odd'`. If `fresp` does not support this calling syntax, `firpm` defaults to even symmetry.

## Examples

Graph the desired and actual frequency responses of a 17th-order Parks-McClellan bandpass filter:

```
f = [0 0.3 0.4 0.6 0.7 1]; a = [0 0 1 1 0 0];
b = firpm(17,f,a);
[h,w] = freqz(b,1,512);
plot(f,a,w/pi,abs(h))
legend('Ideal','firpm Design')
```



# firpm

## Algorithm

`firpm` is a MEX-file version of the original Fortran code from [1], altered to design arbitrarily long filters with arbitrarily many linear bands.

`firpm` designs type I, II, III, and IV linear-phase filters. Type I and type II are the defaults for  $n$  even and  $n$  odd, respectively, while type III ( $n$  even) and type IV ( $n$  odd) are obtained with the 'hilbert' and 'differentiator' flags. The different types of filters have different symmetries and certain constraints on their frequency responses (see [5] for more details).

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

## Diagnostics

If you get the following warning message,

```
- Failure to Converge -  
Probable cause is machine rounding error.
```

it is possible that the filter design may still be correct. Verify the design by checking its frequency response.

## References

- [1] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979, Algorithm 5.1.
- [2] *Selected Papers in Digital Signal Processing, II*, IEEE Press, New York, 1979.
- [3] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, New York:, 1987, p. 83.
- [4] Rabiner, L.R., J.H. McClellan, and T.W. Parks, "FIR Digital Filter Design Techniques Using Weighted Chebyshev Approximations," *Proc. IEEE* 63 (1975).

[5] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 256-266.

**See Also**

butter, cheby1, cheby2, cfirpm, ellip, fir1, fir2, fircls, fircls1, fircls, firrcos, firgr, firpmord, function\_handle, yulewalk

# firpmord

---

## Purpose

Parks-McClellan optimal FIR filter order estimation

## Syntax

```
[n,fo,ao,w] = firpmord(f,a,dev)
[n,fo,ao,w] = firpmord(f,a,dev,fs)
c = firpmord(f,a,dev,fs,'cell')
```

## Description

`[n,fo,ao,w] = firpmord(f,a,dev)` finds the approximate order, normalized frequency band edges, frequency band amplitudes, and weights that meet input specifications `f`, `a`, and `dev`.

- `f` is a vector of frequency band edges (between 0 and  $f_s/2$ , where  $f_s$  is the sampling frequency), and `a` is a vector specifying the desired amplitude on the bands defined by `f`. The length of `f` is two less than twice the length of `a`. The desired function is piecewise constant.
- `dev` is a vector the same size as `a` that specifies the maximum allowable deviation or ripples between the frequency response and the desired amplitude of the output filter for each band.

Use `firpm` with the resulting order `n`, frequency vector `fo`, amplitude response vector `ao`, and weights `w` to design the filter `b` which approximately meets the specifications given by `firpmord` input parameters `f`, `a`, and `dev`.

```
b = firpm(n,fo,ao,w)
```

`[n,fo,ao,w] = firpmord(f,a,dev,fs)` specifies a sampling frequency `fs`. `fs` defaults to 2 Hz, implying a Nyquist frequency of 1 Hz. You can therefore specify band edges scaled to a particular application's sampling frequency.

In some cases `firpmord` underestimates the order `n`. If the filter does not meet the specifications, try a higher order such as `n+1` or `n+2`.

`c = firpmord(f,a,dev,fs,'cell')` generates a cell-array whose elements are the parameters to `firpm`.



## Examples

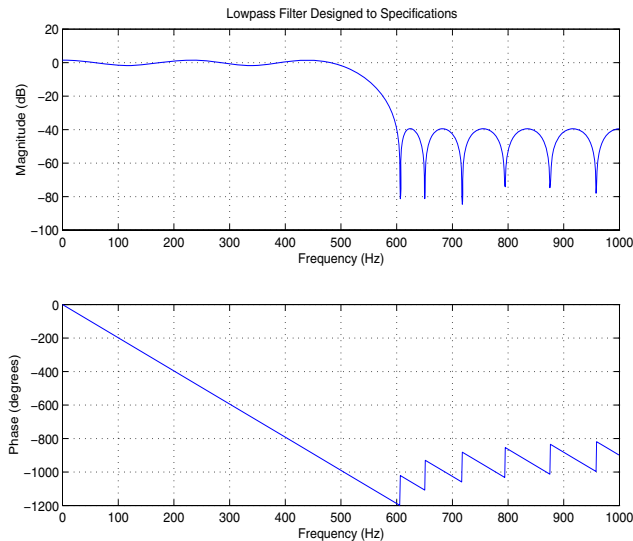
## Example 1

Design a minimum-order lowpass filter with a 500 Hz passband cutoff frequency and 600 Hz stopband cutoff frequency, with a sampling frequency of 2000 Hz, at least 40 dB attenuation in the stopband, and less than 3 dB of ripple in the passband:

```
rp = 3;           % Passband ripple
rs = 40;          % Stopband ripple
fs = 2000;        % Sampling frequency
f = [500 600];   % Cutoff frequencies
a = [1 0];        % Desired amplitudes

% Compute deviations
dev = [(10^(rp/20)-1)/(10^(rp/20)+1)  10^(-rs/20)];

[n,fo,ao,w] = firpmord(f,a,dev,fs);
b = firpm(n,fo,ao,w);
freqz(b,1,1024,fs);
title('Lowpass Filter Designed to Specifications');
```



Note that the filter falls slightly short of meeting the stopband attenuation and passband ripple specifications. Using  $n+1$  in the call to `firpm` instead of  $n$  achieves the desired amplitude characteristics.

## Example 2

Design a lowpass filter with a 1500 Hz passband cutoff frequency and 2000 Hz stopband cutoff frequency, with a sampling frequency of 8000 Hz, a maximum stopband amplitude of 0.1, and a maximum passband error (ripple) of 0.01:

```
[n,fo,ao,w] = firpmord([1500 2000],[1 0],[0.01 0.1],8000 );  
b = firpm(n,fo,ao,w);
```

This is equivalent to

```
c = firpmord( [1500 2000],[1 0],[0.01 0.1],8000,'cell');  
b = firpm(c{:});
```

---

**Note** In some cases, `firpmord` underestimates or overestimates the order  $n$ . If the filter does not meet the specifications, try a higher order such as  $n+1$  or  $n+2$ .

Results are inaccurate if the cutoff frequencies are near 0 or the Nyquist frequency.

---

## Algorithm

`firpmord` uses the algorithm suggested in [1]. This method is inaccurate for band edges close to either 0 or the Nyquist frequency ( $f_s/2$ ).

## References

- [1] Rabiner, L.R., and O. Herrmann, "The Predictability of Certain Optimum Finite Impulse Response Digital Filters," *IEEE Trans. on Circuit Theory*, Vol. CT-20, No. 4 (July 1973), pp. 401-408.
- [2] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975, pp. 156-157.

## See Also

`buttord`, `cheb1ord`, `cheb2ord`, `ellipord`, `kaiserord`, `firpm`

**Purpose**

Raised cosine FIR filter design

**Syntax**

```
b = firrcos(n,F0,df,fs)
b = firrcos(n,F0,df,fs,'bandwidth')
b = firrcos(n,F0,df)
b = firrcos(n,F0,r,fs,'rolloff')
b = firrcos(...,'type')
b = firrcos(...,'type',delay)
b = firrcos(...,'type',delay>window)
[b,a] = firrcos(...)
```

**Description**

`b = firrcos(n,F0,df,fs)` or, equivalently,

`b = firrcos(n,F0,df,fs,'bandwidth')` returns an order  $n$  lowpass linear-phase FIR filter with a raised cosine transition band. The order  $n$  must be even. The filter has cutoff frequency  $F0$ , transition bandwidth  $df$ , and sampling frequency  $fs$ , all in hertz.  $df$  must be small enough so that  $F0 \pm df/2$  is between 0 and  $fs/2$ . The coefficients in  $b$  are normalized so that the nominal passband gain is always equal to 1. Specify  $fs$  as the empty vector `[]` to use the default value  $fs = 2$ .

`b = firrcos(n,F0,df)` uses a default sampling frequency of  $fs = 2$ .

`b = firrcos(n,F0,r,fs,'rolloff')` interprets the third argument,  $r$ , as the rolloff factor instead of the transition bandwidth,  $df$ .  $r$  must be in the range  $[0,1]$ .

`b = firrcos(...,'type')` designs either a normal raised cosine filter or a square root raised cosine filter according to how you specify of the string `'type'`. Specify `'type'` as:

- `'normal'`, for a regular raised cosine filter. This is the default, and is also in effect when the `'type'` argument is left empty, `[]`.
- `'sqrt'`, for a square root raised cosine filter.

`b = firrcos(...,'type',delay)` specifies an integer delay in the range  $[0,n+1]$ . The default is  $n/2$  for even  $n$  and  $(n+1)/2$  for odd  $n$ .

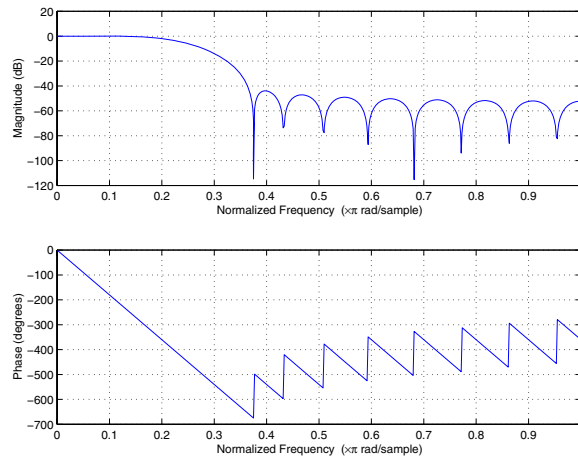
`b = firrcos(..., 'type', delay, window)` applies a length  $n+1$  window to the designed filter to reduce the ripple in the frequency response. `window` must be a length  $n+1$  column vector. If no window is specified, a rectangular (`rectwin`) window is used. Care must be exercised when using a window with a delay other than the default.

`[b,a] = firrcos(...)` always returns `a = 1`.

## Examples

Design an order 20 raised cosine FIR filter with cutoff frequency 0.25 of the Nyquist frequency and a transition bandwidth of 0.25:

```
h = firrcos(20,0.25,0.25);  
freqz(h,1)
```



## See Also

`fir1`, `fir2`, `firls`, `firpm`

**Purpose** Flat Top weighted window

**Syntax**  
`w = flattopwin(L)`  
`w = flattopwin(L, 'sflag')`

**Description** Flat Top windows have very low passband ripple ( $< 0.01$  dB) and are used primarily for calibration purposes. Their bandwidth is approximately 2.5 times wider than a Hann window.

`w = flattopwin(L)` returns the  $L$ -point symmetric Flat Top window in column vector  $w$ .

`w = flattopwin(L, sflag)` returns the  $L$ -point symmetric Flat Top window using `sflag` window sampling, where `sflag` is either 'symmetric' or 'periodic'.

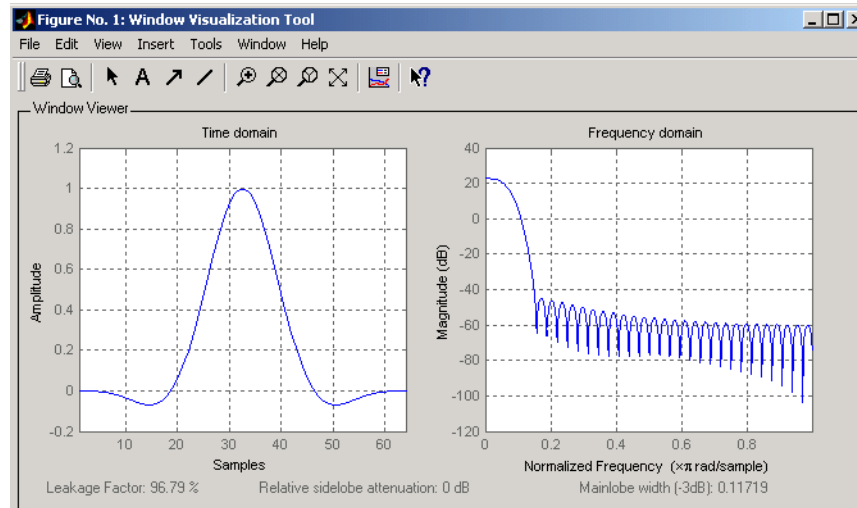
**Algorithm** Flat top windows are summations of cosines. The coefficients of a flat top window are computed from the following equation

$$w(n) = 1 - 1.93 \cos\left(\frac{2\pi t}{T}\right) + 1.29 \cos\left(\frac{4\pi t}{T}\right) - 0.388 \cos\left(\frac{6\pi t}{T}\right) + 0.0322 \cos\left(\frac{8\pi t}{T}\right)$$

where  $0 \leq t \leq T$  and  $w(t) = 0$  elsewhere. The window length  $L = N+1$ .

**Examples** Create a 64-point, symmetric Flat Top window and view the window using WVTTool:

```
w = flattopwin(64);  
wvtool(w);
```



## Reference

[1] Gade, Svend and Herlufsen, H., "Use of Weighting Functions in DFT/FFT Analysis (Part I)," Brüel & Kjær, *Windows to FFT Analysis (Part I) Technical Review*, No. 3, 1987, pp. 19-21.

## See Also

blackman, hamming, hann

<b>Purpose</b>	Frequency response of analog filters
<b>Syntax</b>	<pre>h = freqs(b,a,w) [h,w] = freqs(b,a) [h,w] = freqs(b,a,f) freqs(b,a)</pre>
<b>Description</b>	<p>freqs returns the complex frequency response <math>H(j\omega)</math> (Laplace transform) of an analog filter</p>

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

given the numerator and denominator coefficients in vectors **b** and **a**.

`h = freqs(b,a,w)` returns the complex frequency response of the analog filter specified by coefficient vectors **b** and **a**. `freqs` evaluates the frequency response along the imaginary axis in the complex plane at the angular frequencies in rad/sec specified in real vector **w**, which must contain more than one frequency.

`[h,w] = freqs(b,a)` automatically picks a set of 200 frequency points **w** on which to compute the frequency response **h**.

`[h,w] = freqs(b,a,f)` picks **f** number of frequencies on which to compute the frequency response **h**.

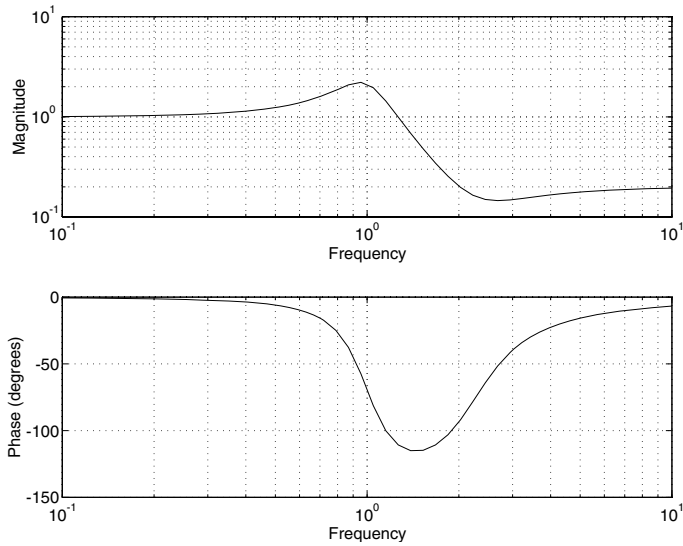
`freqs` with no output arguments plots the magnitude and phase response versus frequency in the current figure window.

`freqs` works only for real input systems and positive frequencies.

**Examples** Find and graph the frequency response of the transfer function given by:

$$H(s) = \frac{0.2s^2 + 0.3s + 1}{s^2 + 0.4s + 1}$$

```
a = [1 0.4 1];
b = [0.2 0.3 1];
w = logspace(-1,1);
freqs(b,a,w)
```



You can also create the plot with:

```
h = freqs(b,a,w);  
mag = abs(h);  
phase = angle(h);  
subplot(2,1,1), loglog(w,mag)  
subplot(2,1,2), semilogx(w,phase)
```

To convert to hertz, degrees, and decibels, use:

```
f = w/(2*pi);  
mag = 20*log10(mag);  
phase = phase*180/pi;
```

## Algorithm

freqs evaluates the polynomials at each frequency point, then divides the numerator response by the denominator response:

```
s = i*w;  
h = polyval(b,s)./polyval(a,s);
```

## See Also

abs, angle, freqz, invfreqs, logspace, polyval



**Purpose**            Frequency spacing for frequency response

**Description**        freqspace is a MATLAB function.

# freqz

---

## Purpose

Frequency response of digital filter

## Syntax

```
[h,w]= freqz(b,a,l)
h = freqz(b,a,w)
[h,w] = freqz(b,a,l, 'whole' )
[h,f] = freqz(b,a,l,fs)
h = freqz(b,a,f,fs)
[h,f] = freqz(b,a,l, 'whole', fs)
freqz(b,a,...)
freqz(Hd)
```

## Description

`[h,w] = freqz(b,a,l)` returns the frequency response vector `h` and the corresponding angular frequency vector `w` for the digital filter whose transfer function is determined by the (real or complex) numerator and denominator polynomials represented in the vectors `b` and `a`, respectively. The vectors `h` and `w` are both of length `l`. The angular frequency vector `w` has values ranging from 0 to  $\pi$  radians per sample. When you don't specify the integer `l`, or you specify it as the empty vector `[]`, the frequency response is calculated using the default value of 512 samples.

`h = freqz(b,a,w)` returns the frequency response vector `h` calculated at the frequencies (in radians per sample) supplied by the vector `w`. The vector `w` can have any length.

`[h,w] = freqz(b,a,l, 'whole' )` uses `n` sample points around the entire unit circle to calculate the frequency response. The frequency vector `w` has length `l` and has values ranging from 0 to  $2\pi$  radians per sample.

`[h,f] = freqz(b,a,l,fs)` returns the frequency response vector `h` and the corresponding frequency vector `f` for the digital filter whose transfer function is determined by the (real or complex) numerator and denominator polynomials represented in the vectors `b` and `a`, respectively. The vectors `h` and `f` are both of length `l`. For this syntax, the frequency response is calculated using the sampling frequency specified by the scalar `fs` (in hertz). The frequency vector `f` is calculated in units of hertz (Hz). The frequency vector `f` has values ranging from 0 to  $fs/2$  Hz.

`h = freqz(b,a,f,fs)` returns the frequency response vector `h` calculated at the frequencies (in Hz) supplied in the vector `f`. The vector `f` can be any length.

`[h,f] = freqz(b,a,1,'whole',fs)` uses `n` points around the entire unit circle to calculate the frequency response. The frequency vector `f` has length `1` and has values ranging from `0` to `fs` Hz.

`freqz(b,a,...)` plots the magnitude and unwrapped phase of the frequency response of the filter. The plot is displayed in the current figure window.

`freqz(Hd)` plots the magnitude and unwrapped phase of the frequency response of the filter. The plot is displayed in `fvtool`. The input `Hd` is a `dfilt` filter object or an array of `dfilt` filter objects.

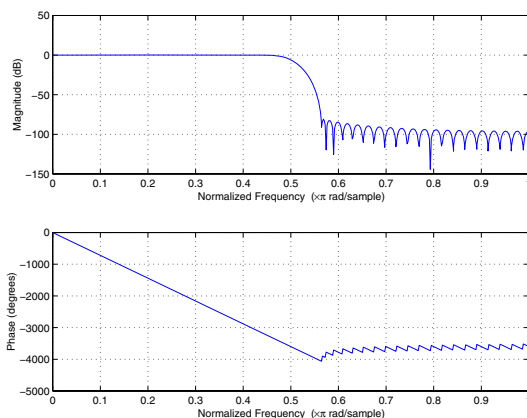
## Remarks

It is best to choose a power of 2 for the third input argument `n`, because `freqz` uses an FFT algorithm to calculate the frequency response. See the reference description of `fft` for more information.

## Examples

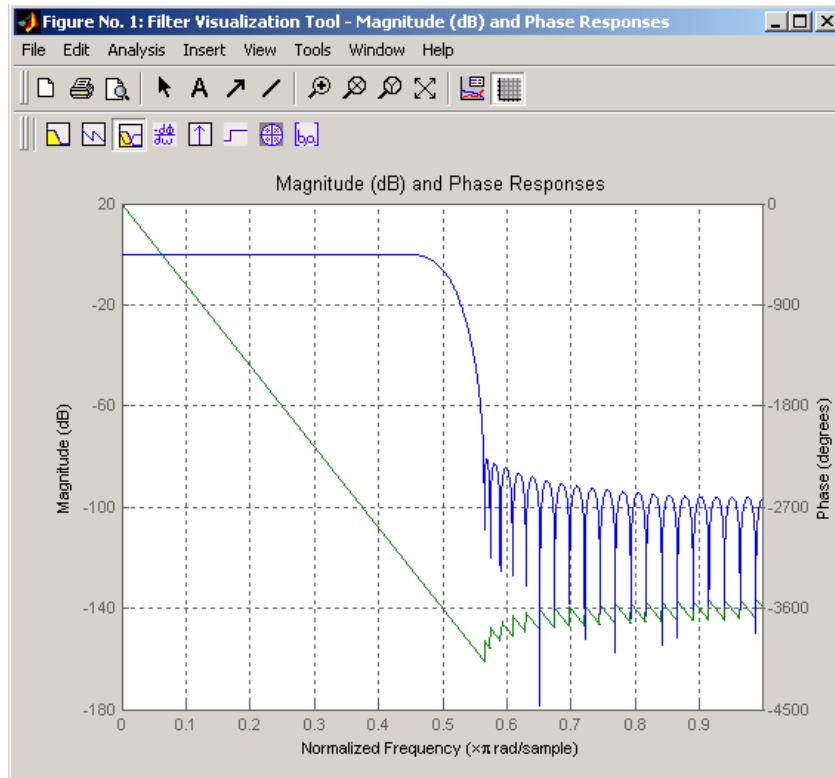
Plot the magnitude and phase response of an FIR filter:

```
b = fir1(80,0.5,kaiser(81,8));
freqz(b,1);
```



The same example using a `dfilt` object and displaying the result in the Filter Visualization Tool (`fvtool`) is

```
Hd = dfilt.dffir(b);
freqz(Hd);
```



## Algorithm

The frequency response [1] of a digital filter can be interpreted as the transfer function evaluated at  $z = e^{j\omega}$ . You can always write a rational transfer function in the following form.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \dots + a(m+1)z^{-m}}$$

`freqz` determines the transfer function from the (real or complex) numerator and denominator polynomials you specify, and returns the complex frequency

response  $H(e^{j\omega})$  of a digital filter. The frequency response is evaluated at sample points determined by the syntax that you use.

freqz generally uses an FFT algorithm to compute the frequency response whenever you don't supply a vector of frequencies as an input argument. It computes the frequency response as the ratio of the transformed numerator and denominator coefficients, padded with zeros to the desired length.

When you do supply a vector of frequencies as an input argument, then freqz evaluates the polynomials at each frequency point using Horner's method of nested polynomial evaluation [1], dividing the numerator response by the denominator response.

## References

[1] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 203-205.

## See Also

abs, angle, fft, filter, freqs, impz, invfreqs, logspace

# fvtool

---

**Purpose** Open Filter Visualization Tool

**Syntax**

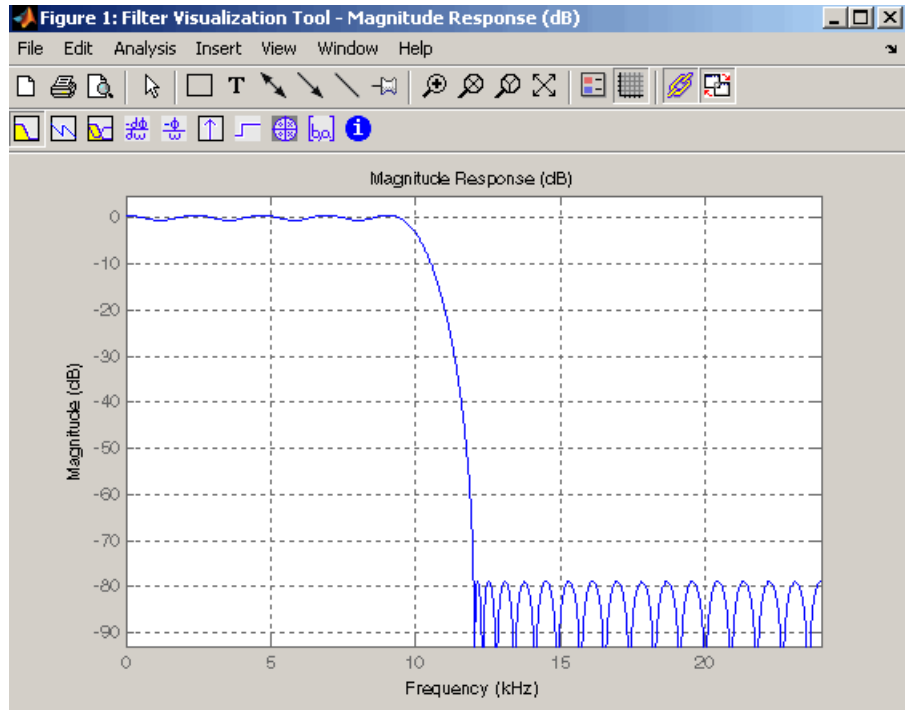
```
fvtool(b, a)
fvtool(b1, a1, b2, a2, ... bn, an)
fvtool(Hd1, Hd2, ...)
h = fvtool(...)
```

**Description** `fvtool(b, a)` opens FVTool and computes the magnitude response of the digital filter defined with numerator, `b` and denominator, `a`. Using FVTool you can display the phase response, group delay, impulse response, step response, pole-zero plot, and coefficients of the filter. You can export the displayed response to a file with **Export** on the **File** menu.

`fvtool(b1, a1, b2, a2, ... bn, an)` opens FVTool and computes the magnitude responses of multiple filters defined with numerators, `b1...bn` and denominators, `a1...an`.




`fvtool(Hd1, Hd2, ...)` opens FVTool and computes the magnitude responses of the filters in the `dfilt` objects `Hd1, Hd2`, etc. If you have the Filter Design Toolbox installed, you can also use `fvtool(H1, H2, ...)` to analyze quantized filter objects (`dfilt` with arithmetic set to 'single'), multirate filter (`mfilt`) objects, and adaptive filter (`adaptfilt`) objects.



`h = fvtool(...)` returns a figure handle `h`. You can use this handle to interact with FVTool from the command line. See “Controlling FVTool from the MATLAB Command Line” below.











FVTool has two toolbars.

- An extended version of the MATLAB plot editing toolbar. The following table shows the toolbar icons specific to FVTool.



Icon	Description
	Restore default view. This view displays buffer regions around the data and shows only significant data. To see the response using standard MATLAB plotting, which shows all data values, use <b>View &gt; Full View</b>
	Toggle legend
	Toggle grid

	Link to FDATool (appears only if FVTool was started from FDATool)
	Toggle Add mode/Replace mode (appears only if FVTool was launched from FDATool)



- Analysis toolbar with the following icons

	Magnitude response of the current filter. See <code>freqz</code> and <code>zerophase</code> for more information.  To see the zero-phase response, right-click on the y-axis label of the Magnitude plot and select <b>Zero-phase</b> from the context menu.
	Phase response of the current filter. See <code>phasez</code> for more information.
	Superimposes the magnitude response and the phase response of the current filter. See <code>freqz</code> for more information.
	Shows the group delay of the current filter. Group delay is the average delay of the filter as a function of frequency. See <code>grpdelay</code> for more information.
	Shows the phase delay of the current filter. Phase delay is the time delay the filter imposes on each component of the input signal. See <code>phasedelay</code> for more information.
	Impulse response of the current filter. The impulse response is the response of the filter to a impulse input. See <code>impz</code> for more information.
	Step response of the current filter. The step response is the response of the filter to a step input. See <code>stepz</code> for more information.
	Pole-zero plot, which shows the pole and zero locations of the current filter on the $z$ -plane. See <code>zplane</code> for more information.





	Filter coefficients of the current filter, which depend on the filter structure (e.g., direct-form, lattice, etc.) in a text box. For SOS filters, each section is displayed as a separate filter.
	Detailed filter information.

### Linking to FDATool

In `fdatool`, selecting **Filter Visualization Tool** from the **View** menu or the **Full View Analysis** toolbar button  when an analysis is displayed starts FVTool for the current filter. You can synchronize FDATool and FVTool with the **FDAToolLink** toolbar button . Any changes made to the filter in FDATool are immediately reflected in FVTool.

Two FDATool link modes are provided via the **Set Link Mode** toolbar button:

- Replace —removes the filter currently displayed in FVTool and inserts the new filter.
- Add —retains the filter currently displayed in FVTool and adds the new filter to the display.

### 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 Linear Frequency	Magnitude Magnitude(dB) Magnitude squared Zero-Phase
Phase	Normalized Frequency Linear Frequency	Phase Continuous Phase Degrees Radians

Plot	X-Axis Units	Y-Axis Units
Magnitude and Phase	Normalized Frequency Linear Frequency	(y-axis on left side) Magnitude Magnitude(dB) Magnitude squared Zero-Phase  (y-axis on right side) Phase Continuous Phase Degrees Radians
Group Delay	Normalized Frequency Linear Frequency	Samples Time
Phase Delay	Normalized Frequency Linear Frequency	Degrees Radians
Impulse Response	Samples Time	Amplitude
Step Response	Samples Time	Amplitude
Pole-Zero	Real Part	Imaginary Part

## 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 on the *x*-axis label and selecting Samples or Time.

To save the displayed parameters as the default values to use when FDATool or FVTool is opened, click **Save as default**.

To restore the MATLAB-defined default values, click **Restore original defaults**.

Data Markers display information about a particular point in the plot. See “Using Data Markers” on page 5-19 for more information.

When FVTool is started from FDATool, you can use Specification Masks to display filter specifications on a Magnitude plot. You can also draw your own specification masks. See “Analyzing the Filter” on page 5-17 for more information.

---

**Note** To use **Passband zoom** on the **View** menu, your filter must have been designed using design or FDATool. Passband zoom is not provided for cascaded integrator-comb (CIC) filters because CICs do not have conventional passbands.

---

## Overlaying a Response

You can overlay a second response on the plot by selecting **Overlay Analysis** from the **Analysis** menu and selecting an available response. A second  $y$ -axis is added to the right side of the response plot. The **Analysis Parameters** dialog box shows parameters for the  $x$ -axis and both  $y$ -axes.

## 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<sup>®</sup> properties (see Handle Graphics in the MATLAB documentation), FVTool has the following properties:

- 'Filters' — returns a cell array of the filters in FVTool.
- 'Analysis' — displays the specified type of analysis plot. The following table lists the analyses and corresponding analysis strings.

<b>Analysis Type</b>	<b>Analysis String</b>
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 (available only with Filter Design Toolbox)	'magestimate'
Round-off noise power (available only with Filter Design Toolbox)	'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 Filter Design Toolbox.) 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` in the MATLAB documentation for information.

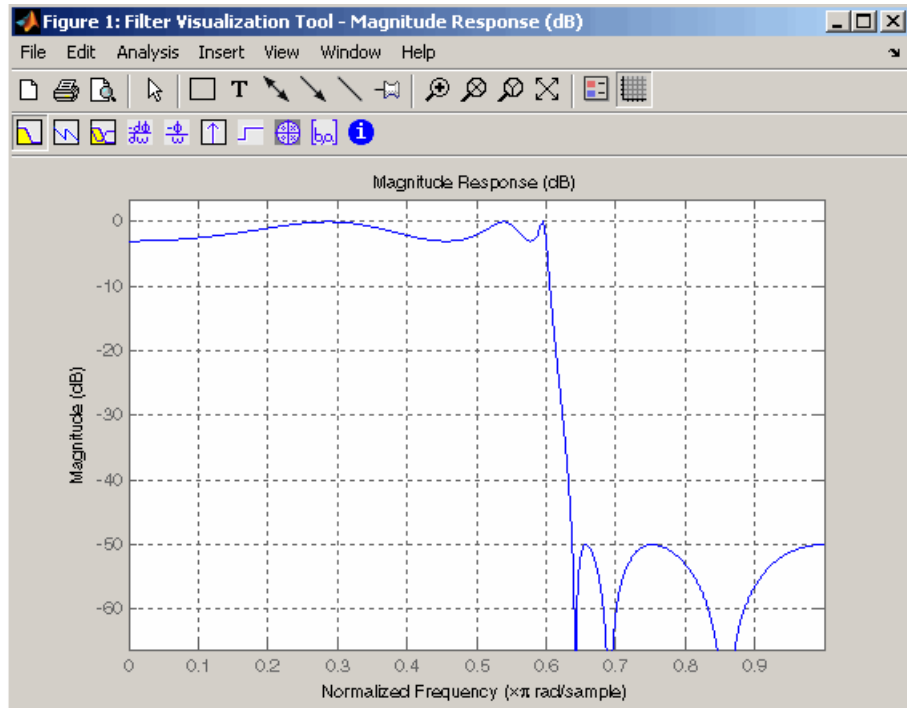
For more information on using FVTool from the command line, see the demo `fvtooldemo`.

## Examples

### Example 1

Display the magnitude response of an elliptic filter, starting FVTool from the command line:

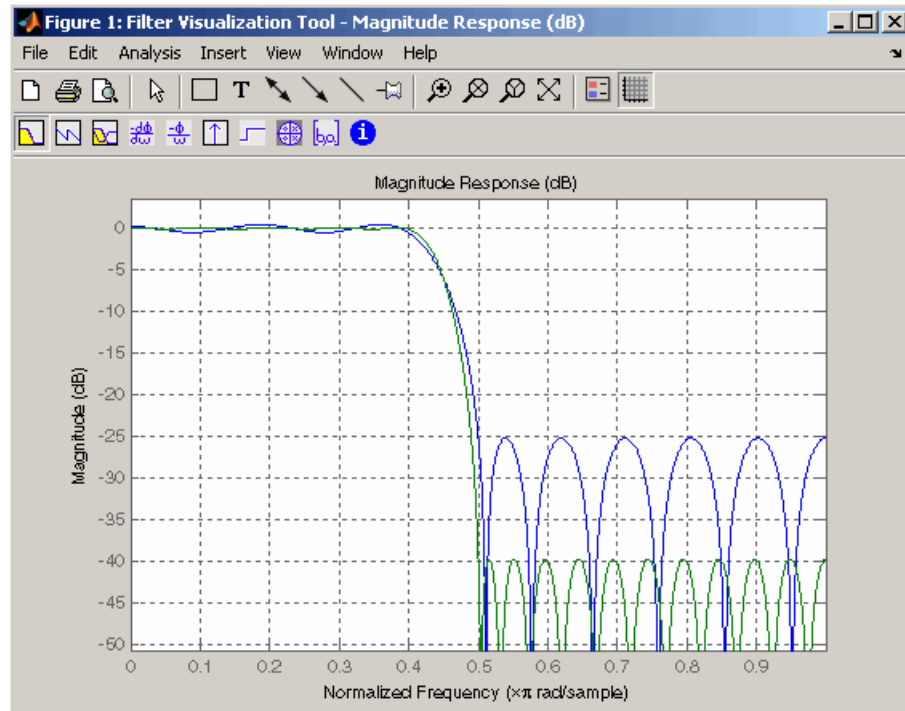
```
[b,a]=ellip(6,3,50,300/500);  
fvtool(b,a);
```



## Example 2

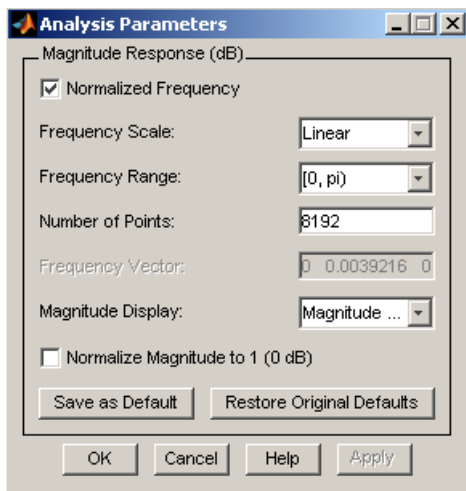
Display and analyze multiple FIR filters, starting FVTool from the command line. Then, display the associated analysis parameters for the magnitude:

```
b1 = firpm(20,[0 0.4 0.5 1],[1 1 0 0]);  
b2 = firpm(40,[0 0.4 0.5 1],[1 1 0 0]);  
fvtool(b1,1,b2,1);
```





Right-click on the plot and select **Analysis Parameters**.



### Example 3

Create a lowpass, equiripple filter of order 20 in FDATool and display it in FVTool.

```
fdatool      %start FDATool
```

Set these parameters in fdatool:

Parameter	Setting
Response Type	Lowpass
Design Method	FIR Equiripple
Filter Order	Specify order: 20
Density factor	16
Frequency specifications units	Normalized (0 to 1)
Wpass	0.4

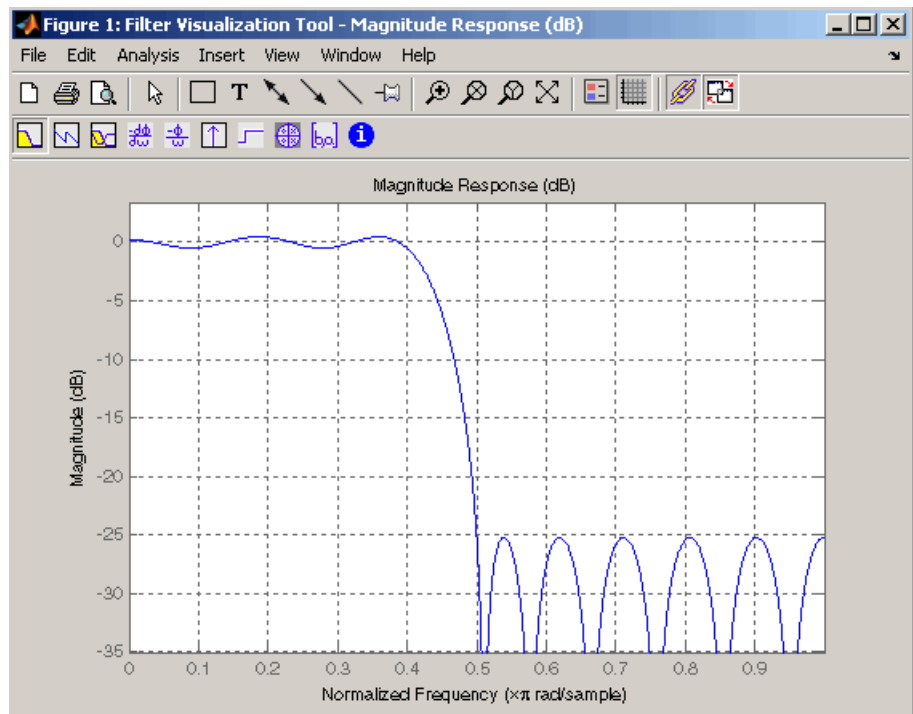
Parameter	Setting
Wstop	0.5
Magnitude specifications Wpass and Wstop	1

and then click the **Design Filter** button.

The screenshot shows the fvtool GUI with the following settings:

- Response Type:**
  - Lowpass
  - Highpass
  - Bandpass
  - Bandstop
  - Differentiator
- Design Method:**
  - IIR Butterworth
  - FIR Equiripple
- Filter Order:**
  - Specify order: 20
  - Minimum order
- Options:**
  - Density factor: 16
- Frequency Specifications:**
  - Units: Normalized (0 to 1)
  - Fs: 48000
  - wpass: 0.4
  - wstop: 0.5
- Magnitude Specifications:**
  - Enter a weight value for each band below.
  - Wpass: 1
  - Wstop: 1

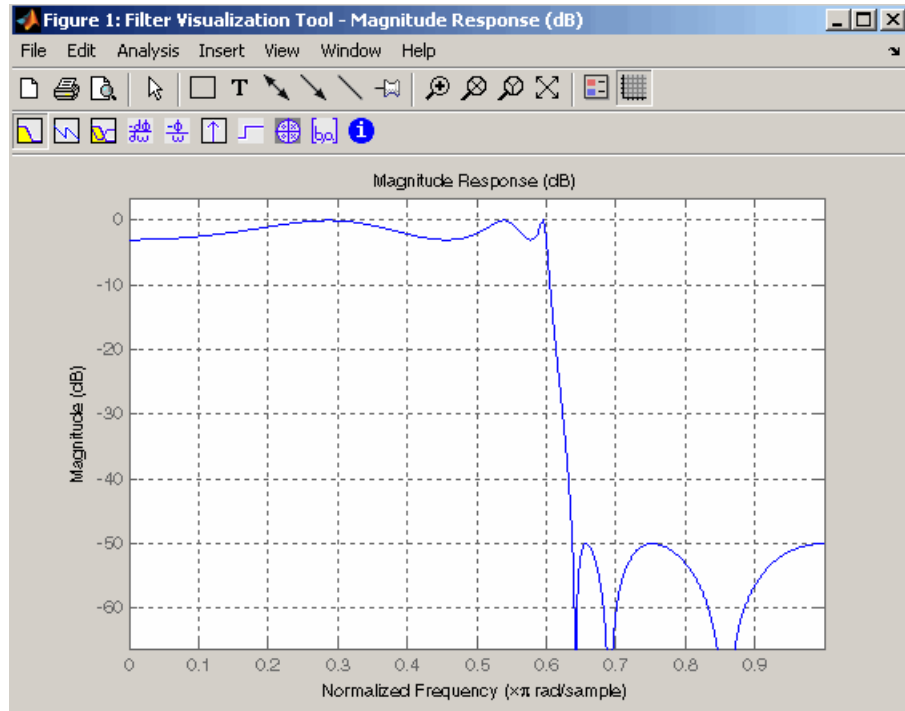
Click the **Full View Analysis** button to start FVTool.



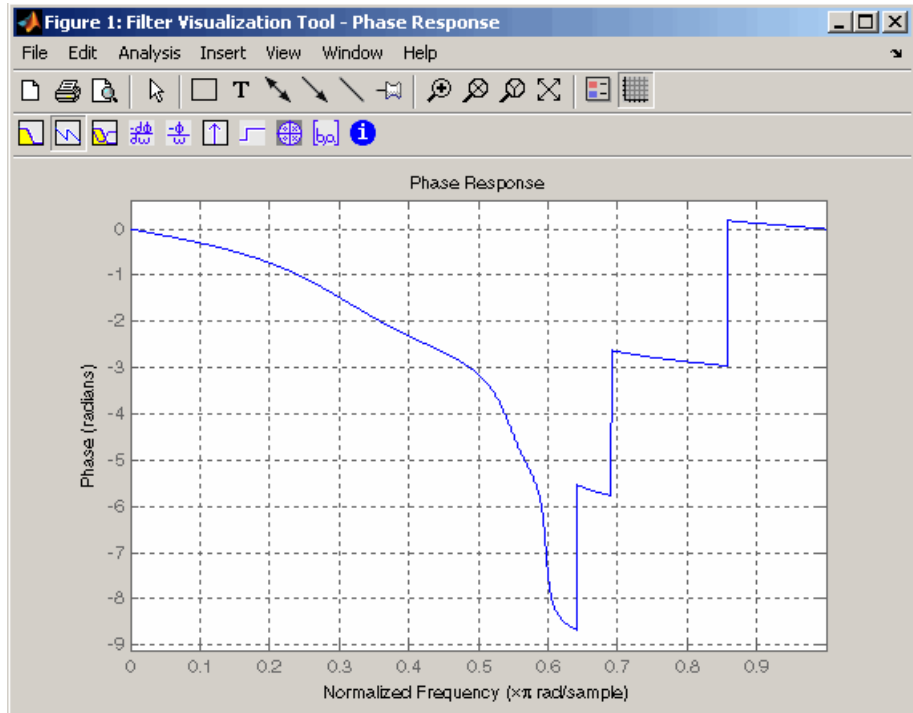
## Example 4

Create an elliptic filter and use some of FVTool's figure handle commands:

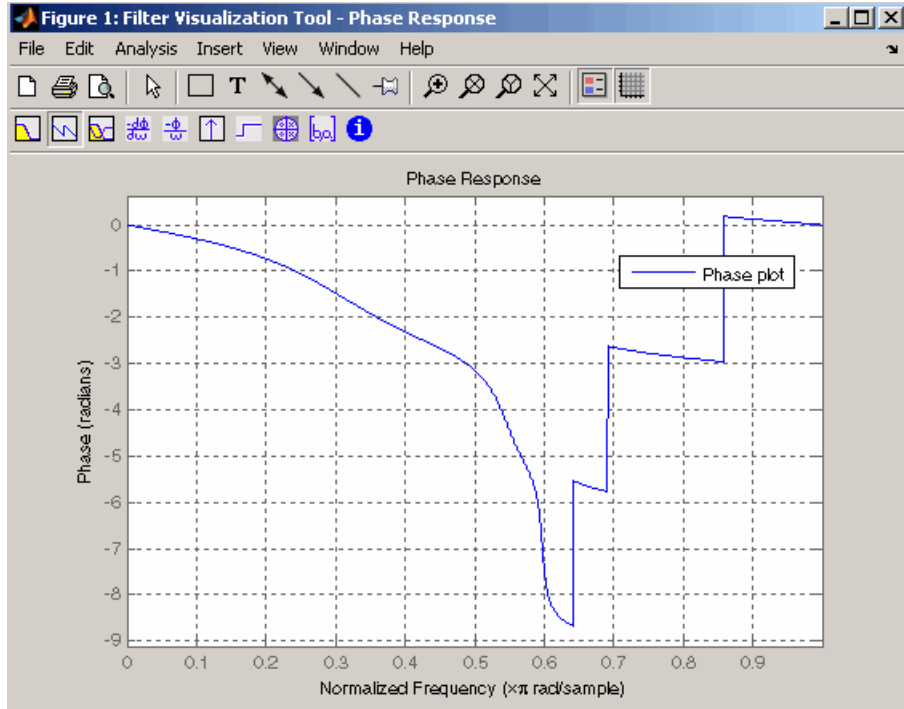
```
[b,a]=ellip(6,3,50,300/500);  
h = fvtool(b,a);           % Create handle, h and start FVTool  
                             % with magnitude plot
```



```
set(h,'Analysis','phase') % Change display to phase plot
```



```
set(h,'Legend','on')           % Turn legend on
legend(h,'Phase plot')        % Add legend text
```



```
get(h)                          % View all properties
                                % FVTool-specific properties are
                                %   at the end of this list.
```

```
AlphaMap: [1x64 double]
BackingStore: 'on'
CloseRequestFcn: 'closereq'
Color: [0.8314 0.8157 0.7843]
ColorMap: [64x3 double]
CurrentAxes: 208.0084
CurrentCharacter: ''
CurrentObject: []
CurrentPoint: [0 0]
DockControls: 'on'
```

```
    DoubleBuffer: 'on'
      FileName: ''
      FixedColors: [11x3 double]
    IntegerHandle: 'on'
  InvertHardcopy: 'on'
    KeyPressFcn: ''
      MenuBar: 'none'
    MinColormap: 64
      Name: 'Filter Visualization Tool - Phase Response'
        NextPlot: 'new'
      NumberTitle: 'on'
        PaperUnits: 'inches'
    PaperOrientation: 'portrait'
      PaperPosition: [0.2500 2.5000 8 6]
    PaperPositionMode: 'manual'
      PaperSize: [8.5000 11]
      PaperType: 'usletter'
      Pointer: 'arrow'
    PointerShapeCData: [16x16 double]
  PointerShapeHotSpot: [1 1]
    Position: [360 292 560 345]
    Renderer: 'painters'
      RendererMode: 'auto'
        Resize: 'on'
          ResizeFcn: ''
        SelectionType: 'normal'
          ShareColors: 'on'
            Toolbar: 'auto'
              Units: 'pixels'
    WindowButtonDownFcn: ''
  WindowButtonMotionFcn: ''
    WindowButtonUpFcn: ''
      WindowStyle: 'normal'
    BeingDeleted: 'off'
  ButtonDownFcn: ''
    Children: [15x1 double]
    Clipping: 'on'
    CreateFcn: ''
    DeleteFcn: ''
    BusyAction: 'queue'
```

```
HandleVisibility: 'on'
    HitTest: 'on'
    Interruptible: 'on'
    Parent: 0
    Selected: 'off'
SelectionHighlight: 'on'
    Tag: 'filtervisualizationtool'
    UIContextMenu: []
    UserData: []
    Visible: 'on'
AnalysisToolbar: 'on'
FigureToolbar: 'on'
    Filters: {[1x1 dfilt.df2t]}
    Grid: 'on'
    Legend: 'on'
    DesignMask: 'off'
    Fs: 1
    SOSViewSettings: [1x1 dspopts.sosview]
    Analysis: 'phase'
OverlaidAnalysis: ''
    ShowReference: 'on'
    PolyphaseView: 'off'
NormalizedFrequency: 'on'
    FrequencyScale: 'Linear'
    FrequencyRange: '[0, pi]'
    NumberOfPoints: 8192
    FrequencyVector: [1x256 double]
    PhaseUnits: 'Radians'
    PhaseDisplay: 'Phase'
```

## See Also

`fdatool`, `sptool`



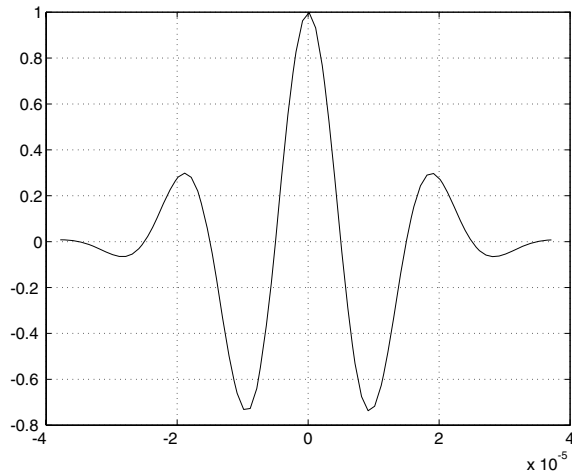
<b>Purpose</b>	Gaussian-modulated sinusoidal pulse
<b>Syntax</b>	<pre>yi = gauspuls(t,fc,bw) yi = gauspuls(t,fc,bw,bwr) [yi,yq] = gauspuls(...) [yi,yq,ye] = gauspuls(...) tc = gauspuls('cutoff',fc,bw,bwr,tpe)</pre>
<b>Description</b>	<p>gauspuls generates Gaussian-modulated sinusoidal pulses.</p> <p><code>yi = gauspuls(t,fc,bw)</code> returns a unity-amplitude Gaussian RF pulse at the times indicated in array <code>t</code>, with a center frequency <code>fc</code> in hertz and a fractional bandwidth <code>bw</code>, which must be greater than 0. The default value for <code>fc</code> is 1000 Hz and for <code>bw</code> is 0.5.</p> <p><code>yi = gauspuls(t,fc,bw,bwr)</code> returns a unity-amplitude Gaussian RF pulse with a fractional bandwidth of <code>bw</code> as measured at a level of <code>bwr</code> dB with respect to the normalized signal peak. The fractional bandwidth reference level <code>bwr</code> must be less than 0, because it indicates a reference level less than the peak (unity) envelope amplitude. The default value for <code>bwr</code> is -6 dB.</p> <p><code>[yi,yq] = gauspuls(...)</code> returns both the in-phase and quadrature pulses.</p> <p><code>[yi,yq,ye] = gauspuls(...)</code> returns the RF signal envelope.</p> <p><code>tc = gauspuls('cutoff',fc,bw,bwr,tpe)</code> returns the cutoff time <code>tc</code> (greater than or equal to 0) at which the trailing pulse envelope falls below <code>tpe</code> dB with respect to the peak envelope amplitude. The trailing pulse envelope level <code>tpe</code> must be less than 0, because it indicates a reference level less than the peak (unity) envelope amplitude. The default value for <code>tpe</code> is -60 dB.</p>
<b>Remarks</b>	Default values are substituted for empty or omitted trailing input arguments.

# gauspuls

## Examples

Plot 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:

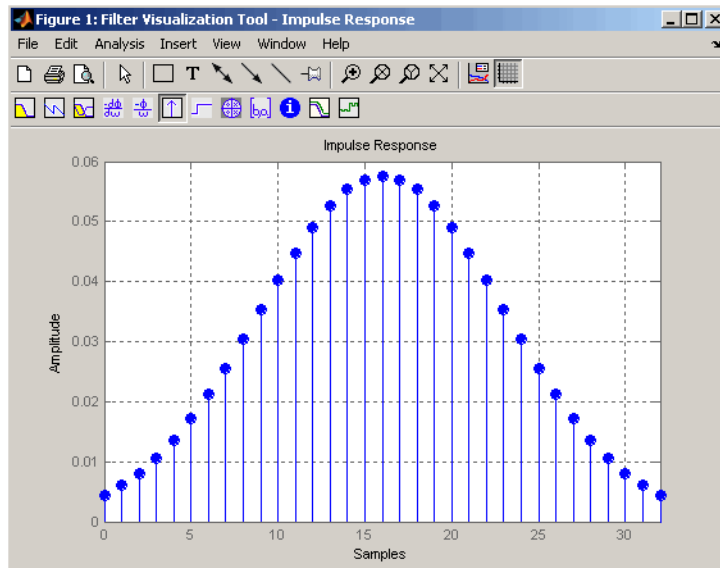
```
tc = gauspuls('cutoff',50e3,0.6,[],-40);  
t = -tc : 1e-6 : tc;  
yi = gauspuls(t,50e3,0.6);  
plot(t,yi)
```



## See Also

chirp, cos, diric, pulstran, rectpuls, sawtooth, sin, sinc, square, tripuls

<b>Purpose</b>	Gaussian FIR pulse-shaping filter
<b>Syntax</b>	<pre>h = gaussfir(bt) h = gaussfir(bt,n) h = gaussfir(bt,n,o)</pre>
<b>Description</b>	<p>This filter is used primarily in Gaussian minimum shift keying (GMSK) communications applications.</p> <p><code>h = gaussfir(bt)</code> designs a low pass FIR Gaussian pulse-shaping filter and returns the filter coefficients in the <code>h</code> vector. <code>bt</code> is the 3-dB bandwidth-symbol time product where <code>b</code> is the two-sided bandwidth in hertz and <code>t</code> is in seconds. Smaller <code>bt</code> products produce larger pulse widths. The number of symbol periods (<code>n</code>) defaults to 3 and the oversampling factor (<code>o</code>) defaults to 2.</p> <p>The length of the impulse response of the filter is given by <math>2*o*n+1</math>. The coefficients <code>h</code> are normalized so that the nominal passband gain is always equal to 1.</p> <p><code>h = gaussfir(bt,n)</code> uses <code>n</code> number of symbol periods between the start of the filter impulse response and its peak.</p> <p><code>h = gaussfir(bt,n,o)</code> uses an oversampling factor of <code>o</code>, which is the number of samples per symbol.</p>
<b>Examples</b>	<p>Design a Gaussian filter to be used in a Global System for Mobile communications (GSM) GMSK scheme.</p> <pre>bt = .3;           % 3-dB bandwidth-symbol time o = 8;            % Oversampling factor n = 2;           % 2 symbol periods to the filters peak. h = gaussfir(bt,n,o); hfvt = fvtool(h,'impulse');</pre>



## References

- [1] Rappaport T.S., *Wireless Communications Principles and Practice*, Prentice Hall, 1996.
- [2] Krishnapura N., Pavan S., Mathiazhagan C., Ramamurthi B., "A Baseband Pulse Shaping Filter for Gaussian Minimum Shift Keying," *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems*, 1998.

## See Also

firrcos, rcosfir

**Purpose** Gaussian window

**Syntax** `w = gausswin(L)`  
`w = gausswin(L,α)`

**Description** `w = gausswin(L)` returns an L-point Gaussian window in the column vector `w`. `L` is a positive integer. The coefficients of a Gaussian window are computed from the following equation.

$$w(n) = e^{-\frac{1}{2}\left(\alpha\frac{n}{N/2}\right)^2}$$

where  $-\frac{N}{2} \leq n \leq \frac{N}{2}$  and  $\alpha \geq 2$ . The window length  $L = N+1$ .

`w = gausswin(L,α)` returns an L-point Gaussian window where  $\alpha$  is the reciprocal of the standard deviation. The width of the window is inversely related to the value of  $\alpha$ ; a larger value of  $\alpha$  produces a more narrow window. If  $\alpha$  is omitted, it defaults to 2.5.

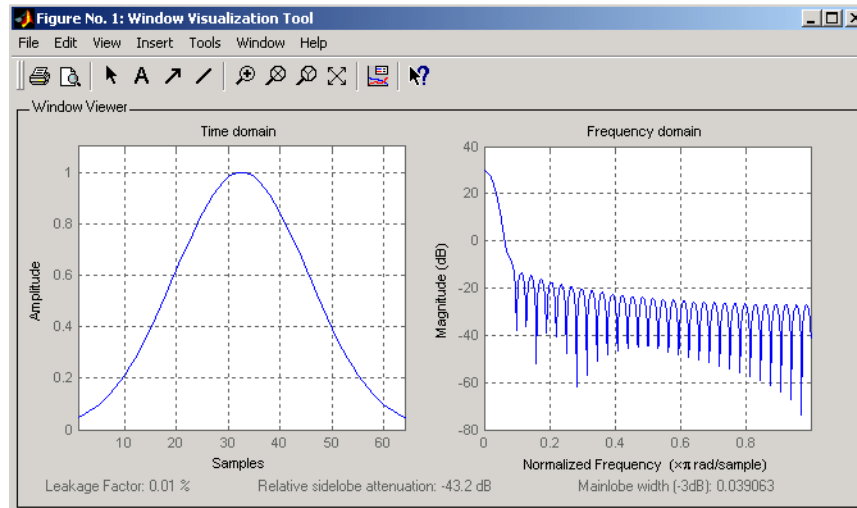
---

**Note** If the window appears to be clipped, increase the number of points (`L`) used for `gausswin(L)`.

---

**Examples** Create a 64-point Gaussian window and display the result in WVTTool:

```
L=64;  
wvtool(gausswin(L))
```



---

**Note** The shape of this window is similar in the frequency domain because the Fourier transform of a Gaussian is also a Gaussian.

---

## References

- [1] Harris, F.J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66, No. 1 (January 1978).
- [2] Roberts, Richard A., and C.T. Mullis. *Digital Signal Processing*. Reading, MA: Addison-Wesley, 1987, pp. 135-136.

## See Also

chebwin, kaiser, tukeywin, window, wintool, wvtool

**Purpose** Gaussian monopulse

**Syntax**  
`y = gmonopuls(t,fc)`  
`tc = gmonopuls('cutoff',fc)`

**Description** `y = gmonopuls(t,fc)` returns samples of the unity-amplitude Gaussian monopulse with center frequency `fc` (in hertz) at the times indicated in array `t`. By default, `fc = 1000` Hz.

`tc = gmonopuls('cutoff',fc)` returns the time duration between the maximum and minimum amplitudes of the pulse.

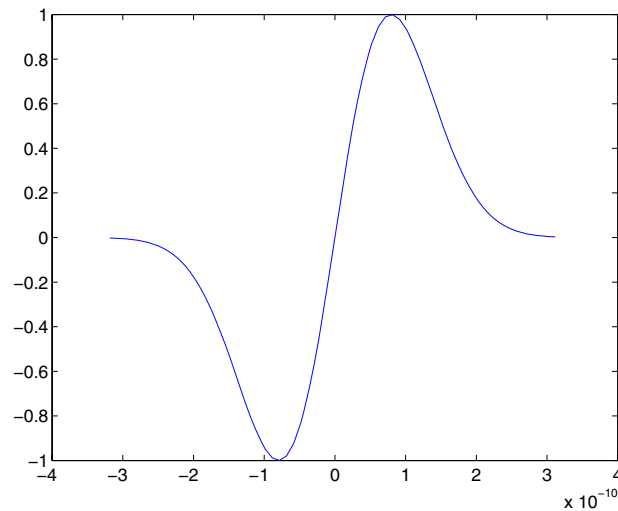
**Remarks** Default values are substituted for empty or omitted trailing input arguments.

## Examples

### Example 1

Plot a 2 GHz Gaussian monopulse sampled at a rate of 100 GHz:

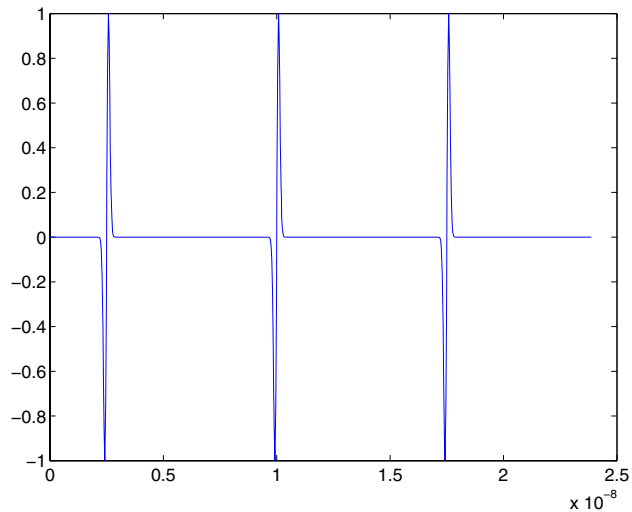
```
fc = 2E9; fs=100E9;  
tc = gmonopuls('cutoff',fc);  
t = -2*tc : 1/fs : 2*tc;  
y = gmonopuls(t,fc); plot(t,y)
```



## Example 2

Construct a pulse train from the monopulse of Example 1 using a spacing of 7.5 ns:

```
fc = 2E9; fs=100E9;           % center freq, sample freq
D = [2.5 10 17.5]' * 1e-9;   % pulse delay times
tc = gmonopuls('cutoff',fc); % width of each pulse
t = 0 : 1/fs : 150*tc;       % signal evaluation time
yp = pulstran(t,D,@gmonopuls,fc);
plot(t,yp)
```



## See Also

chirp, gauspuls, pulstran, rectpuls, tripuls



**Purpose** Discrete Fourier transform using second-order Goertzel algorithm

**Syntax**  
`y = goertzel(x,i)`  
`y = goertzel(x,i,dim)`

**Description** `goertzel` computes the discrete Fourier transform (DFT) of specific indices in a vector or matrix.

`y = goertzel(x,i)` returns the DFT of vector `x` at the indices in vector `i`, computed using the second-order Goertzel algorithm. If `x` is a matrix, `goertzel` computes each column separately. The indices in vector `i` must be integer values from 1 to `N`, where `N` is the length of the first matrix dimension of `x` that is greater than 1. The resulting `y` has the same dimensions as `x`. If `i` is omitted, it is assumed to be `[1:N]`, which results in a full DFT computation.

`y = goertzel(x,i,dim)` returns the discrete Fourier transform (DFT) of matrix `x` at the indices in vector `i`, computed along the dimension `dim` of `x`.

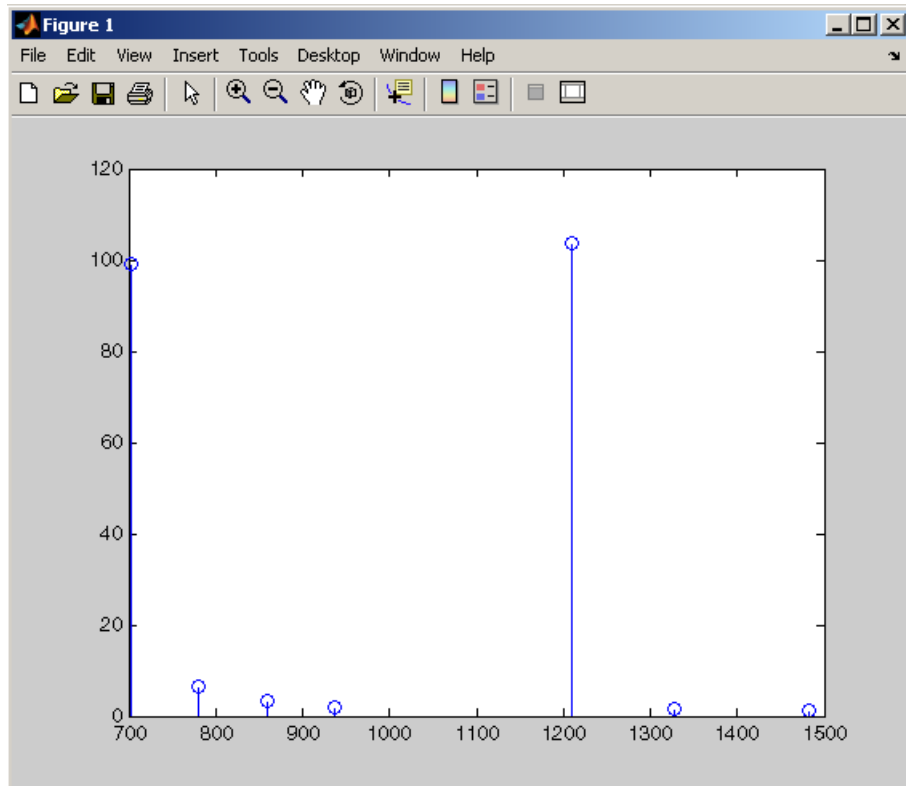
**Note** `fft` computes all DFT values at all indices, while `goertzel` computes DFT values at a specified subset of indices (i.e., a portion of the signal's frequency range). If less than  $\log_2(N)$  points are required, `goertzel` is more efficient than the Fast Fourier Transform (`fft`).

Two examples where `goertzel` can be useful are spectral analysis of very large signals and dual-tone multifrequency (DTMF) signal detection.

**Examples** Estimate the frequency of the two tones generated by the “1” button on a telephone keypad.

```
% Frequency tones of the telephone pad (Hz)
f = [697 770 852 941 1209 1336 1477];
Fs = 8000;
N = 205;
% Tones of 25.6 ms
tones = sum(sin(2*pi*[697;1209]*(0:N-1)/Fs));
k = round(f/Fs*N);           % Indices of the DFT
ydft = goertzel(tones,k+1);  % DC is represented by the value 1
```

```
estim_f = round(k*Fs/N); % Frequencies at which DFT estimated  
stem(estim_f,abs(ydft)) % Peaks detected around 697 & 1209 Hz
```



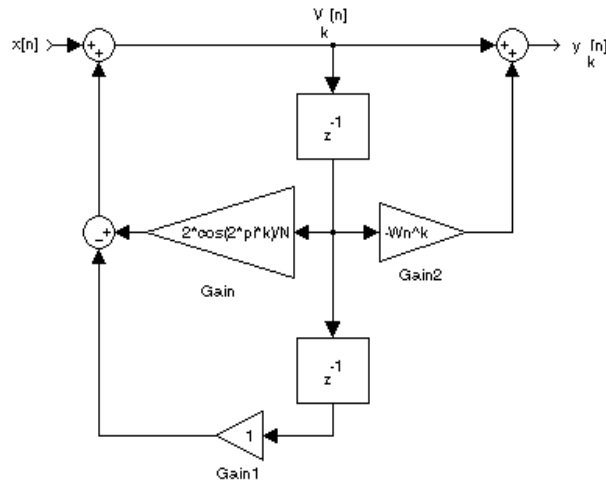
## Algorithm

goertzel implements this transfer function

$$H_k(z) = \frac{1 - W_N^k z^{-1}}{1 - 2 \cos\left(\frac{2\pi}{N}k\right) z^{-1} + z^{-2}}$$

where  $N$  is the length of the signal and  $k$  is the index of the computed DFT.  $k$  is related to the indices in vector  $i$  above as  $k = i - 1$ .

The signal flow graph for this transfer function is



and it is implemented as

$$v_k[n] = x_e[n] + 2 \cos\left(\frac{2\pi k}{N}\right)v_k[n-1] - v_k[n-2]$$

where

$$x_e(n) = \begin{cases} x(n), & 0 \leq n \leq N-1 \\ 0, & n < 0, n \geq N \end{cases}$$

and

$$X[k] = y_k[N] = v_k[N] - W_N^k v_k[N-1]$$

To compute  $X[k]$  for a particular  $k$ , the Goertzel algorithm requires  $4N$  real multiplications and  $4N$  real additions. Although this is less efficient than computing the DFT by the direct method, Goertzel uses recursion to compute

$$W_N^k \text{ and } \cos\left(\frac{2\pi k}{N}\right)$$

which are evaluated only at  $n = N$ . The direct DFT does not use recursion and must compute each complex term separately.

## References

[1] Burrus, C.S. and T.W. Parks. *DFT/FFT and Convolution Algorithms*. John Wiley & Sons, 1985, pp. 32-26.

[2] Mitra, Sanjit K. *Digital Signal Processing: A Computer-Based Approach*. New York, NY: McGraw-Hill, 1998, pp. 520-523.

## See Also

fft, fft2

**Purpose** Average filter delay (group delay)

**Syntax**

```

grpdelay(b,a)
[gd,w] = grpdelay(b,a,n)
[gd,f] = grpdelay(b,a,n,fs)
[gd,w] = grpdelay(b,a,n,'whole')
[gd,f] = grpdelay(b,a,n,'whole',fs)
gd = grpdelay(b,a,w)
gd = grpdelay(b,a,f,fs)
grpdelay(Hd)

```

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

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

where  $\omega$  is frequency and  $\theta$  is the phase angle of  $H(e^{j\omega})$ .

`grpdelay(b,a)` with no output arguments plots the group delay versus frequency in the current figure window.

`[gd,w] = grpdelay(b,a,1)` returns the *i*-point group delay,  $\tau_g(\omega)$ , of the digital filter

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \dots + a(m+1)z^{-m}}$$

given the numerator and denominator coefficients in vectors `b` and `a`. `grpdelay` returns both `gd`, the group delay, and `w`, a vector containing the *n* frequency points in radians. `grpdelay` evaluates the group delay at *n* points equally spaced around the upper half of the unit circle, so `w` contains *n* points between 0 and  $\pi$ .

`[gd,f] = grpdelay(b,a,n,fs)` specifies a positive sampling frequency `fs` in hertz. It returns a length *n* vector `f` containing the actual frequency points at which the group delay is calculated, also in hertz. `f` contains *n* points between 0 and `fs/2`.

# grpdelay

`[gd,w] = grpdelay(b,a,n,'whole')` and

`[gd,f] = grpdelay(b,a,n,'whole',fs)` use `n` points around the whole unit circle (from 0 to  $2\pi$ , or from 0 to `fs`).

`gd = grpdelay(b,a,w)` and

`gd = grpdelay(b,a,f,fs)` return the group delay evaluated at the points in `w` (in radians) or `f` (in hertz), respectively, where `fs` is the sampling frequency in hertz.

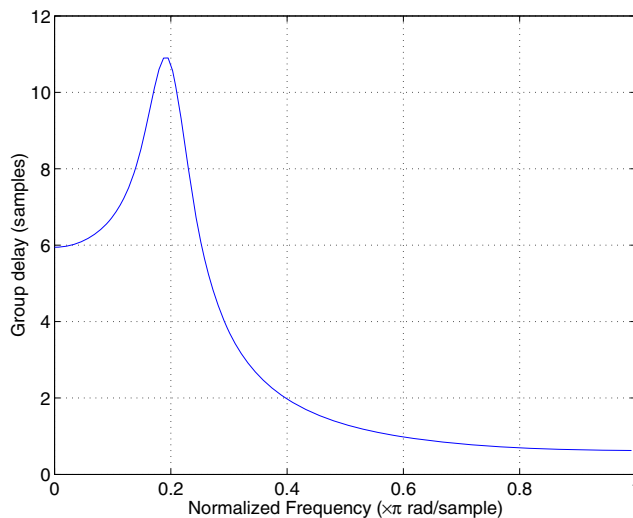
`grpdelay(Hd)` plots the group delay and displays the plot in `fvtool`. The input `Hd` is a `dfilt` filter object or an array of `dfilt` filter objects.

`grpdelay` works for both real and complex filters.

## Examples

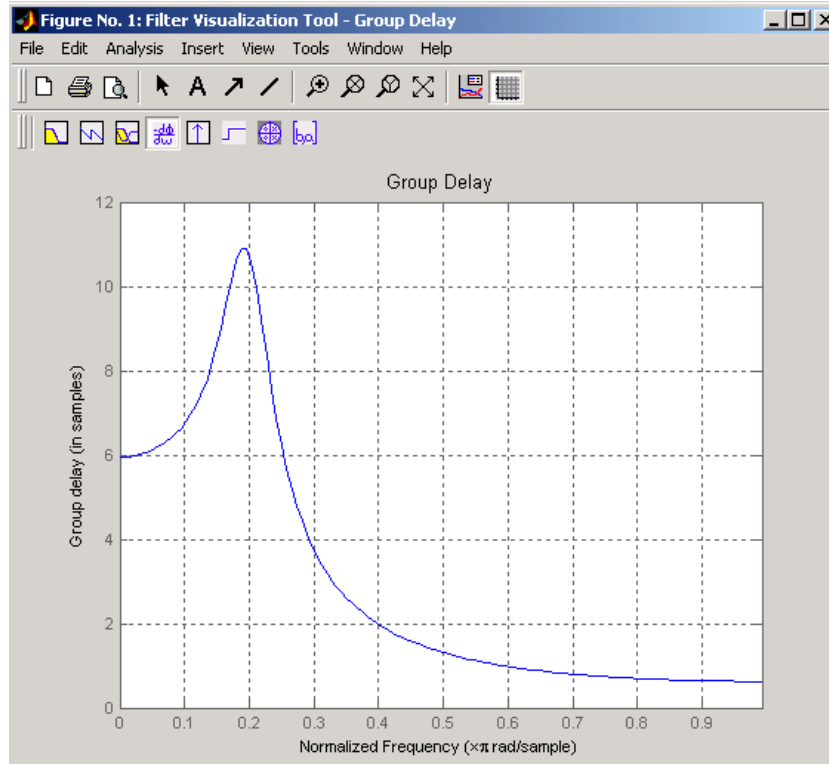
Plot the group delay of Butterworth filter  $b(z)/a(z)$ :

```
[b,a] = butter(6,0.2);  
grpdelay(b,a,128)
```



The same example using a `dfilt` object and displaying the result in the Filter Visualization Tool (`fvtool`) is

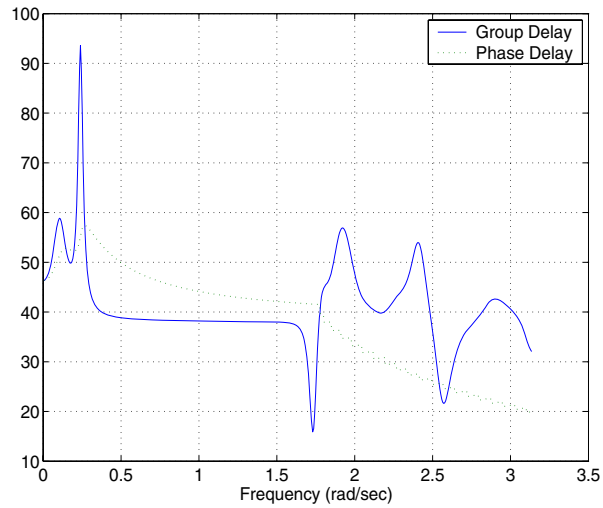
```
[b,a] = butter(6,0.2);
Hd=dfilt.df1(b,a);
grpdelay(Hd,128)
```



Plot both the group and phase delays of a system on the same graph:

```
gd = grpdelay(b,a,512);
gd(1) = []; % Avoid NaNs
[h,w] = freqz(b,a,512); h(1) = []; w(1) = [];
pd = -unwrap(angle(h))./w;
plot(w,gd,w,pd,':')
xlabel('Frequency (rad/sec)'); grid;
legend('Group Delay','Phase Delay');
```

# grpdelay



## Algorithm

grpdelay multiplies the filter coefficients by a unit ramp. After Fourier transformation, this process corresponds to differentiation.

## See Also

cceps, fft, freqz, hilbert, icceps, rceps



**Purpose** Hamming window

**Syntax**  
`w = hamming(L)`  
`w = hamming(L, 'sflag')`

**Description** `w = hamming(L)` returns an L-point symmetric Hamming window in the column vector `w`. `L` is a positive integer. The coefficients of a Hamming window are computed from the following equation.

$$w(n) = 0.54 - 0.46 \cos\left(2\pi\frac{n}{N}\right), \quad 0 \leq n \leq N$$

The window length  $L = N+1$ .

`w = hamming(L, 'sflag')` returns an L-point Hamming window using the window sampling specified by `'sflag'`, which can be either `'periodic'` or `'symmetric'` (the default). When `'periodic'` is specified, `hamming` computes a length `L+1` window and returns the first `n` points.

---

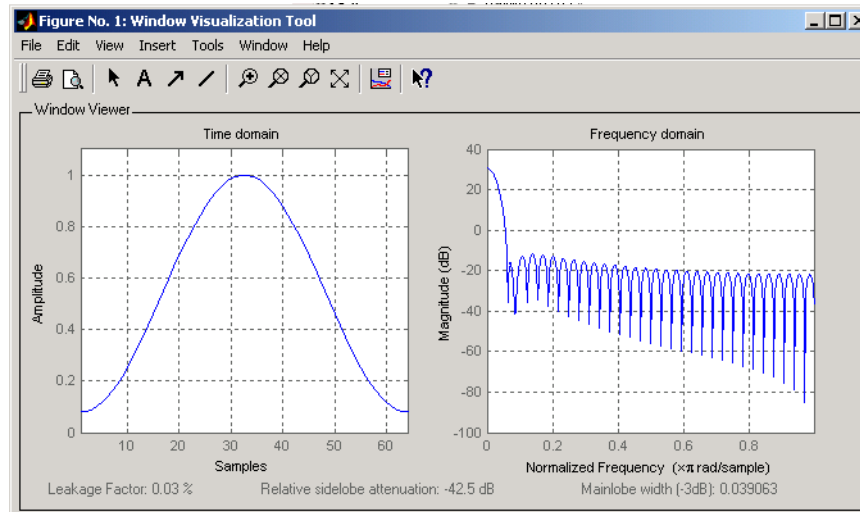
**Note** If you specify a one-point window (`L=1`), the value 1 is returned.

---

**Examples** Create a 64-point Hamming window and display the result in WVTool:

```
L=64;  
wvtool(hamming(L))
```

# hamming



## References

[1] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 447-448.

## See Also

blackman, flattopwin, hann, window, wintool, wvtool

**Purpose** Hann (Hanning) window

**Syntax**  
`w = hann(L)`  
`w = hann(L, 'sflag')`

**Description** `w = hann(L)` returns an L-point symmetric Hann window in the column vector `w`. `L` must be a positive integer. The coefficients of a Hann window are computed from the following equation.

$$w(n) = 0.5 \left( 1 - \cos \left( 2\pi \frac{n}{N} \right) \right), \quad 0 \leq n \leq N$$

The window length  $L = N+1$ .

`w = hann(L, 'sflag')` returns an L-point Hann window using the window sampling specified by `'sflag'`, which can be either `'periodic'` or `'symmetric'` (the default). When `'periodic'` is specified, `hann` computes a length `L+1` window and returns the first `L` points.

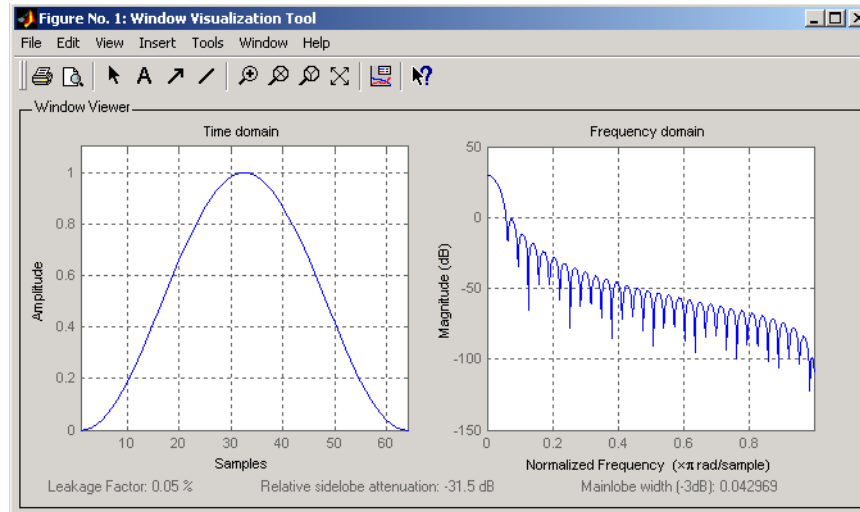
---

**Note** If you specify a one-point window (`L=1`), the value 1 is returned.

---

**Examples** Create a 64-point Hann window and display the result in WVTool:

```
L=64;  
wvtool(hann(L))
```



## References

[1] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 447-448.

## See Also

blackman, flattopwin, hamming, window, wintool, wvtool

**Purpose** Discrete-time analytic signal using Hilbert transform

**Syntax**

```
x = hilbert(xr)
x = hilbert(xr,n)
```

**Description** `x = hilbert(xr)` returns a complex helical sequence, sometimes called the *analytic signal*, from a real data sequence. The analytic signal  $x = x_r + i \cdot x_i$  has a real part,  $x_r$ , which is the original data, and an imaginary part,  $x_i$ , which contains the Hilbert transform. The imaginary part is a version of the original real sequence with a  $90^\circ$  phase shift. Sines are therefore transformed to cosines and vice versa. The Hilbert transformed series has the same amplitude and frequency content as the original real data and includes phase information that depends on the phase of the original data.

If  $x_r$  is a matrix, `x = hilbert(xr)` operates columnwise on the matrix, finding the Hilbert transform of each column.

`x = hilbert(xr,n)` uses an  $n$  point FFT to compute the Hilbert transform. The input data  $x_r$  is zero-padded or truncated to length  $n$ , as appropriate.

The Hilbert transform is useful in calculating instantaneous attributes of a time series, especially the amplitude and frequency. The instantaneous amplitude is the amplitude of the complex Hilbert transform; the instantaneous frequency is the time rate of change of the instantaneous phase angle. For a pure sinusoid, the instantaneous amplitude and frequency are constant. The instantaneous phase, however, is a sawtooth, reflecting the way in which the local phase angle varies linearly over a single cycle. For mixtures of sinusoids, the attributes are short term, or local, averages spanning no more than two or three points.

Reference [1] describes the Kolmogorov method for minimum phase reconstruction, which involves taking the Hilbert transform of the logarithm of the spectral density of a time series. The toolbox function `rceps` performs this reconstruction.

For a discrete-time analytic signal  $x$ , the last half of `fft(x)` is zero, and the first (DC) and center (Nyquist) elements of `fft(x)` are purely real.

# hilbert

---

## Examples

```
xr = [1 2 3 4];  
x = hilbert(xr)  
  
x =  
    1.0000+1.0000i  2.0000-1.0000i  3.0000-1.0000i  4.0000+1.0000i
```

You can see that the imaginary part,  $\text{imag}(x) = [1 \ -1 \ -1 \ 1]$ , is the Hilbert transform of  $xr$ , and the real part,  $\text{real}(x) = [1 \ 2 \ 3 \ 4]$ , is simply  $xr$  itself. Note that the last half of  $\text{fft}(x) = [10 \ -4+4i \ -2 \ 0]$  is zero (in this example, the last half is just the last element), and that the DC and Nyquist elements of  $\text{fft}(x)$ , 10 and -2 respectively, are purely real.

## Algorithm

The analytic signal for a sequence  $x$  has a *one-sided Fourier transform*, that is, negative frequencies are 0. To approximate the analytic signal, `hilbert` calculates the FFT of the input sequence, replaces those FFT coefficients that correspond to negative frequencies with zeros, and calculates the inverse FFT of the result.

In detail, `hilbert` uses a four-step algorithm:

- 1** It calculates the FFT of the input sequence, storing the result in a vector  $x$ .
- 2** It creates a vector  $h$  whose elements  $h(i)$  have the values:
  - 1 for  $i = 1, (n/2)+1$
  - 2 for  $i = 2, 3, \dots, (n/2)$
  - 0 for  $i = (n/2)+2, \dots, n$
- 3** It calculates the element-wise product of  $x$  and  $h$ .
- 4** It calculates the inverse FFT of the sequence obtained in step 3 and returns the first  $n$  elements of the result.

If the input data  $xr$  is a matrix, `hilbert` operates in a similar manner, extending each step above to handle the matrix case.

## References

- [1] Claerbout, J.F., *Fundamentals of Geophysical Data Processing*, McGraw-Hill, 1976, pp. 59-62.
- [2] Marple, S.L., "Computing the discrete-time analytic signal via FFT," *IEEE Transactions on Signal Processing*, Vol. 47, No. 9 (September 1999), pp. 2600-2603.

[3] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, 2nd ed., Prentice-Hall, 1998.

**See Also**

fft, ifft, rceps

# icceps

---

**Purpose** Inverse complex cepstrum

**Syntax** `x = icceps(xhat,nd)`

---

**Description** **Note** `icceps` only works on real data.

---

`x = icceps(xhat,nd)` returns the inverse complex cepstrum of the real data sequence `xhat`, removing `nd` samples of delay. If `xhat` was obtained with `cceps(x)`, then the amount of delay that was added to `x` was the element of `round(unwrap(angle(fft(x)))/pi)` corresponding to  $\pi$  radians.

**References** [1] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

**See Also** `cceps`, `hilbert`, `rcceps`, `unwrap`



**Purpose** Inverse discrete cosine transform

**Syntax**  
 $x = \text{idct}(y)$   
 $x = \text{idct}(y, n)$

**Description** The inverse discrete cosine transform reconstructs a sequence from its discrete cosine transform (DCT) coefficients. The `idct` function is the inverse of the `dct` function.

$x = \text{idct}(y)$  returns the inverse discrete cosine transform of  $y$

$$x(n) = \sum_{k=1}^N w(k)y(k) \cos \frac{\pi(2n-1)(k-1)}{2N}, \quad n = 1, \dots, N$$

where

$$w(k) = \begin{cases} \frac{1}{\sqrt{N}}, & k = 1 \\ \sqrt{\frac{2}{N}}, & 2 \leq k \leq N \end{cases}$$

and  $N = \text{length}(x)$ , which is the same as  $\text{length}(y)$ . The series is indexed from  $n = 1$  and  $k = 1$  instead of the usual  $n = 0$  and  $k = 0$  because MATLAB vectors run from 1 to  $N$  instead of from 0 to  $N-1$ .

$x = \text{idct}(y, n)$  appends zeros or truncates the vector  $y$  to length  $n$  before transforming.

If  $y$  is a matrix, `idct` transforms its columns.

**References**

[1] Jain, A.K., *Fundamentals of Digital Image Processing*, Prentice-Hall, 1989.  
 [2] Pennebaker, W.B., and J.L. Mitchell, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993, Chapter 4.

**See Also** `dct`, `dct2`, `idct2`, `ifft`

# ifft

---

**Purpose** 1-D inverse fast Fourier transform

**Description** `ifft` is a MATLAB function.

**Purpose** 2-D inverse fast Fourier transform

**Description** `ifft2` is a MATLAB function.

# impinvar

---

## Purpose

Impulse invariance method for analog-to-digital filter conversion

## Syntax

```
[bz,az] =impinvar(b,a,fs)
[bz,az] =impinvar(b,a)
[bz,az] =impinvar(b,a,fs,tol)
```

## Description

`[bz,az] =impinvar(b,a,fs)` creates a digital filter with numerator and denominator coefficients `bz` and `az`, respectively, whose impulse response is equal to the impulse response of the analog filter with coefficients `b` and `a`, scaled by  $1/fs$ . If you leave out the argument `fs`, or specify `fs` as the empty vector `[]`, it takes the default value of 1 Hz.

`[bz,az] =impinvar(b,a,fs,tol)` uses the tolerance specified by `tol` to determine whether poles are repeated. A larger tolerance increases the likelihood that `impinvar` interprets closely located poles as multiplicities (repeated ones). The default is 0.001, or 0.1% of a pole's magnitude. Note that the accuracy of the pole values is still limited to the accuracy obtainable by the `roots` function.

## Examples

### Example 1

Convert an analog lowpass filter to a digital filter using `impinvar` with a sampling frequency of 10 Hz:

```
[b,a] = butter(4,0.3,'s');
[bz,az] =impinvar(b,a,10)

bz =
    1.0e-006 *
    -0.0000    0.1324    0.5192    0.1273    0

az =
    1.0000   -3.9216    5.7679   -3.7709    0.9246
```

## Example 2

Illustrate the relationship between analog and digital impulse responses [2].

---

**Note** This example requires the `impulse` function from the Control System Toolbox.

---

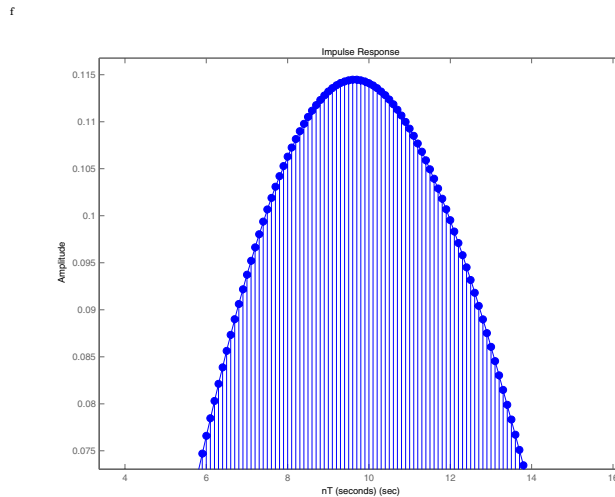
The steps used in this example are:

- 1 Create an analog Butterworth filter
- 2 Use `impinvar` with a sampling frequency  $F_s$  of 10 Hz to scale the coefficients by  $1/F_s$ . This compensates for the gain that will be introduced in Step 4 below.
- 3 Use Control System Toolbox `impulse` function to plot the continuous-time unit impulse response of an LTI system.
- 4 Plot the digital impulse response, multiplying the numerator by a constant ( $F_s$ ) to compensate for the  $1/F_s$  gain introduced in the impulse response of the derived digital filter.

```
[b,a] = butter(4,0.3,'s');  
[bz,az] =impinvar(b,a,10);  
sys = tf(b,a);  
impulse(sys);  
hold on;  
impz(10*bz,az,[],10);
```

Zooming the resulting plot shows that the analog and digital impulse responses are the same.

# impinvar



## Algorithm

`impinvar` performs the impulse-invariant method of analog-to-digital transfer function conversion discussed in reference [1]:

- 1** It finds the partial fraction expansion of the system represented by  $b$  and  $a$ .
- 2** It replaces the poles  $p$  by the poles  $\exp(p/fs)$ .
- 3** It finds the transfer function coefficients of the system from the residues from step 1 and the poles from step 2.

## References

- [1] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, 1987, pp. 206-209.
- [2] Antoniou, Andreas, *Digital Filters*, McGraw Hill, Inc, 1993, pp.221-224.

## See Also

`bilinear`, `lp2bp`, `lp2bs`, `lp2hp`, `lp2lp`

---

<b>Purpose</b>	Impulse response of digital filter
<b>Syntax</b>	<pre>[h,t] = impz(b,a) [h,t] = impz(b,a,n) [h,t] = impz(b,a,n,fs) impz(b,a) impz(Hd)</pre>
<b>Description</b>	<p><code>[h,t] = impz(b,a)</code> computes the impulse response of the filter with numerator coefficients <code>b</code> and denominator coefficients <code>a</code>. <code>impz</code> chooses the number of samples and returns the response in the column vector <code>h</code> and sample times in the column vector <code>t</code> (where <code>t = [0:n-1]'</code>, and <code>n = length(t)</code> is computed automatically).</p> <p><code>[h,t] = impz(b,a,n)</code> computes <code>n</code> samples of the impulse response when <code>n</code> is an integer (<code>t = [0:n-1]'</code>). If <code>n</code> is a vector of integers, <code>impz</code> computes the impulse response at those integer locations, starting the response computation from 0 (and <code>t = n</code> or <code>t = [0 n]</code>). If, instead of <code>n</code>, you include the empty vector <code>[]</code> for the second argument, the number of samples is computed automatically by default.</p> <p><code>[h,t] = impz(b,a,n,fs)</code> computes <code>n</code> samples and produces a vector <code>t</code> of length <code>n</code> so that the samples are spaced <code>1/fs</code> units apart.</p> <p><code>impz(b,a)</code> with no output arguments plots the impulse response and displays the response in the current figure window.</p> <p><code>impz(Hd)</code> plots the impulse response of the filter and displays the plot in <code>fvtool</code>. The input <code>Hd</code> is a <code>dfilt</code> filter object or an array of <code>dfilt</code> filter objects.</p>

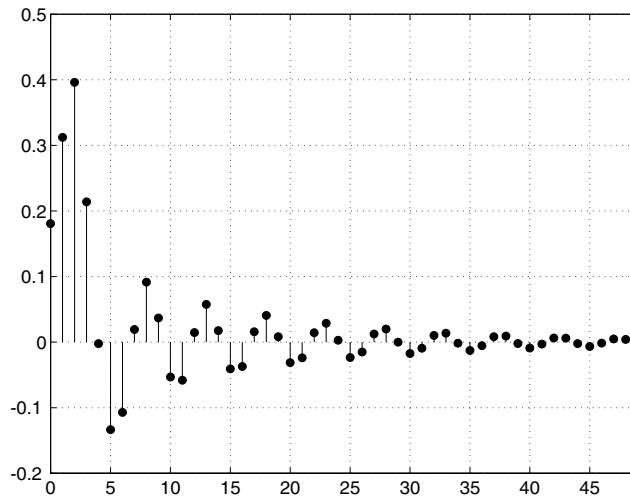
---

**Note** `impz` works for both real and complex input systems.

---

**Examples** Plot the first 50 samples of the impulse response of a fourth-order lowpass elliptic filter with cutoff frequency of 0.4 times the Nyquist frequency:

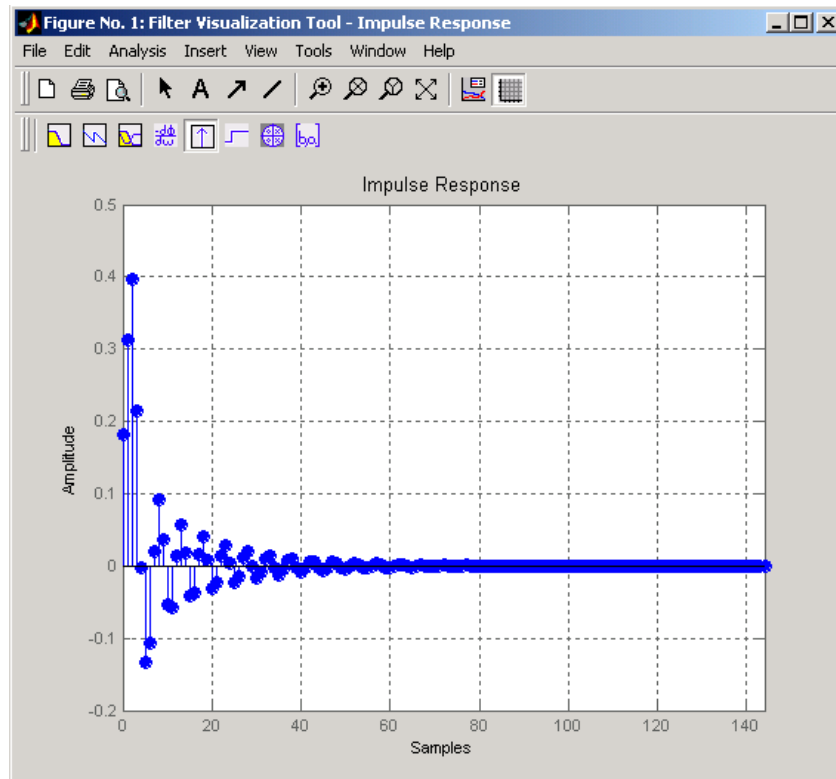
```
[b,a] = ellip(4,0.5,20,0.4);
impz(b,a,50)
```



The same example using a `dfilt` object and displaying the result in the Filter Tool (`fvtool`) is

```
[b,a] = ellip(4,0.5,20,0.4);  
Hd = dfilt.df1(b,a)  
impz(Hd,50)
```





## Algorithm

impz filters a length  $n$  impulse sequence using

```
filter(b,a,[1 zeros(1,n-1)])
```

and plots the results using stem.

To compute  $n$  in the auto-length case, impz either uses  $n = \text{length}(b)$  for the FIR case or first finds the poles using  $p = \text{roots}(a)$ , if  $\text{length}(a)$  is greater than 1.

If the filter is unstable,  $n$  is chosen to be the point at which the term from the largest pole reaches  $10^6$  times its original value.

If the filter is stable,  $n$  is chosen to be the point at which the term due to the largest amplitude pole is  $5 \cdot 10^{-5}$  of its original amplitude.

# impz

---

If the filter is oscillatory (poles on the unit circle only), `impz` computes five periods of the slowest oscillation.

If the filter has both oscillatory and damped terms, `n` is chosen to equal five periods of the slowest oscillation or the point at which the term due to the largest (nonunity) amplitude pole is  $5 \cdot 10^{-5}$  of its original amplitude, whichever is greater.

`impz` also allows for delays in the numerator polynomial. The number of delays is incorporated into the computation for the number of samples.

## See Also

`impulse`, `stem`

**Purpose** Interpolation — increase sampling rate by integer factor

**Syntax**

```
y = interp(x,r)
y = interp(x,r,l,alpha)
[y,b] = interp(x,r,l,alpha)
```

**Description** Interpolation increases the original sampling rate for a sequence to a higher rate. `interp` performs lowpass interpolation by inserting zeros into the original sequence and then applying a special lowpass filter.

`y = interp(x,r)` increases the sampling rate of `x` by a factor of `r`. The interpolated vector `y` is `r` times longer than the original input `x`.

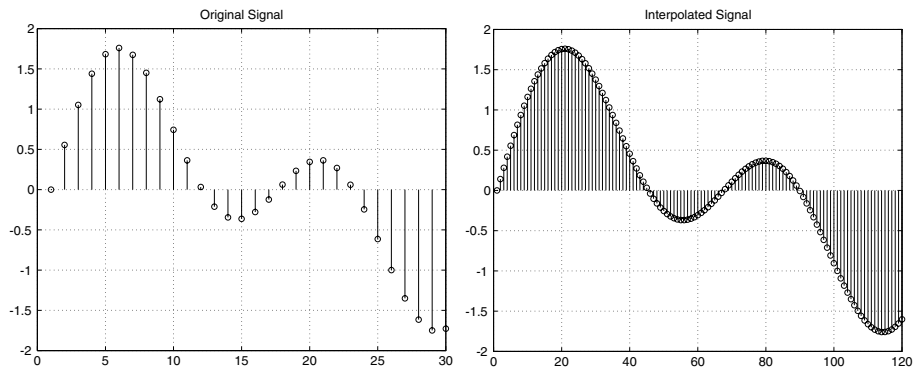
`y = interp(x,r,l,alpha)` specifies `l` (filter length) and `alpha` (cut-off frequency). The default value for `l` is 4 and the default value for `alpha` is 0.5.

`[y,b] = interp(x,r,l,alpha)` returns vector `b` containing the filter coefficients used for the interpolation.

**Examples** Interpolate a signal by a factor of four:

```
t = 0:0.001:1; % Time vector
x = sin(2*pi*30*t) + sin(2*pi*60*t);
y = interp(x,4);
stem(x(1:30));
title('Original Signal');
figure
stem(y(1:120));
title('Interpolated Signal');
```

# interp



## Algorithm

interp uses the lowpass interpolation Algorithm 8.1 described in [1]:

- 1 It expands the input vector to the correct length by inserting zeros between the original data values.
- 2 It designs a special symmetric FIR filter that allows the original data to pass through unchanged and interpolates between so that the mean-square errors between the interpolated points and their ideal values are minimized.
- 3 It applies the filter to the input vector to produce the interpolated output vector.

The length of the FIR lowpass interpolating filter is  $2 * l * r + 1$ . The number of original sample values used for interpolation is  $2 * l$ . Ordinarily,  $l$  should be less than or equal to 10. The original signal is assumed to be band limited with normalized cutoff frequency  $0 \leq \alpha \leq 1$ , where  $1$  is half the original sampling frequency (the Nyquist frequency). The default value for  $l$  is 4 and the default value for  $\alpha$  is 0.5.

## Diagnostics

If  $r$  is not an integer, interp gives the following error message:

```
Resampling rate R must be an integer.
```

## References

[1] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979, Algorithm 8.1.

## See Also

decimate, downsample, interp1, resample, spline, upfirdn, upsample

**Purpose**

Interpolation FIR filter design

**Syntax**

```
b = intfilt(l,p,alpha)
b = intfilt(l,n,'Lagrange')
```

**Description**

`b = intfilt(l,p,alpha)` designs a linear phase FIR filter that performs ideal bandlimited interpolation using the nearest  $2 \cdot p$  nonzero samples, when used on a sequence interleaved with  $l - 1$  consecutive zeros every  $l$  samples. It assumes an original bandlimitedness of  $\alpha$  times the Nyquist frequency. The returned filter is identical to that used by `interp`. `b` is length  $2 \cdot l \cdot p - 1$

$\alpha$  is inversely proportional to the transition bandwidth of the filter and it also affects the bandwidth of the don't-care regions in the stopband. Specifying  $\alpha$  allows you to specify how much of the Nyquist interval your input signal occupies. This is beneficial, particularly for signals to be interpolated, because it allows you to increase the transition bandwidth without affecting the interpolation and results in better stopband attenuation for a given  $l$  and  $p$ . If you set  $\alpha$  to 1, your signal is assumed to occupy the entire Nyquist interval. Setting  $\alpha$  to less than one allows for don't-care regions in the stopband. For example, if your input occupies half the Nyquist interval, you could set  $\alpha$  to 0.5.

`b = intfilt(l,n,'Lagrange')` designs an FIR filter that performs  $n$ th-order Lagrange polynomial interpolation on a sequence interleaved with  $l - 1$  consecutive zeros every  $l$  samples. `b` has length  $(n + 1) \cdot l$  for  $n$  even, and length  $(n + 1) \cdot l - 1$  for  $n$  odd. If both  $n$  and  $l$  are even, the filter designed is not linear phase.

Both types of filters are basically lowpass and have a gain of 1 in the passband.

**Examples**

Design a digital interpolation filter to upsample a signal by four, using the bandlimited method:

```
alpha = 0.5; % "Bandlimitedness" factor
h1 = intfilt(4,2,alpha); % Bandlimited interpolation
```

The filter `h1` works best when the original signal is bandlimited to  $\alpha$  times the Nyquist frequency. Create a bandlimited noise signal:

```
randn('state',0)
x = filter(fir1(40,0.5),1,randn(200,1)); % Bandlimit
```

Now zero pad the signal with three zeros between every sample. The resulting sequence is four times the length of  $x$ :

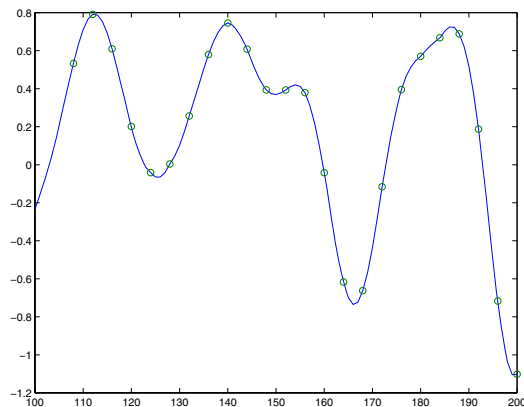
```
xr = reshape([x zeros(length(x),3)]',4*length(x),1);
```

Interpolate using the filter command:

```
y = filter(h1,1,xr);
```

$y$  is an interpolated version of  $x$ , delayed by seven samples (the group-delay of the filter). Zoom in on a section of one hundred samples to see this:

```
plot(100:200,y(100:200),7+(101:4:196),x(26:49),'o')
```



`intfilt` also performs Lagrange polynomial interpolation of the original signal. For example, first-order polynomial interpolation is just linear interpolation, which is accomplished with a triangular filter:

```
h2 = intfilt(4,1,'l') % Lagrange interpolation
```

```
h2 =  
0.2500 0.5000 0.7500 1.0000 0.7500 0.5000 0.2500
```

## Algorithm

The bandlimited method uses `firls` to design an interpolation FIR equivalent to that presented in [1]. The polynomial method uses Lagrange's polynomial interpolation formula on equally spaced samples to construct the appropriate filter.

**References**

[1] Oetken, Parks, and Schüßler, “New Results in the Design of Digital Interpolators,” *IEEE Trans. Acoust., Speech, Signal Processing*, Vol. ASSP-23 (June 1975), pp. 301-309.

**See Also**

decimate, downsample, interp, resample, upsample

# invfreqs

---

**Purpose** Identify continuous-time filter parameters from frequency response data

**Syntax**

```
[b,a] = invfreqs(h,w,n,m)
[b,a] = invfreqs(h,w,n,m,wt)
[b,a] = invfreqs(h,w,n,m,wt,iter)
[b,a] = invfreqs(h,w,n,m,wt,iter,tol)
[b,a] = invfreqs(h,w,n,m,wt,iter,tol,'trace')
[b,a] = invfreqs(h,w,'complex',n,m,...)
```

**Description** `invfreqs` is the inverse operation of `freqs`. It finds a continuous-time transfer function that corresponds to a given complex frequency response. From a laboratory analysis standpoint, `invfreqs` is useful in converting magnitude and phase data into transfer functions.

`[b,a] = invfreqs(h,w,n,m)` returns the real numerator and denominator coefficient vectors `b` and `a` of the transfer function

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

whose complex frequency response is given in vector `h` at the frequency points specified in vector `w`. Scalars `n` and `m` specify the desired orders of the numerator and denominator polynomials.

Frequency is specified in radians between 0 and  $\pi$ , and the length of `h` must be the same as the length of `w`. `invfreqs` uses `conj(h)` at `-w` to ensure the proper frequency domain symmetry for a real filter.

`[b,a] = invfreqs(h,w,n,m,wt)` weights the fit-errors versus frequency, where `wt` is a vector of weighting factors the same length as `w`.

`[b,a] = invfreqs(h,w,n,m,wt,iter)` and

`[b,a] = invfreqs(h,w,n,m,wt,iter,tol)` provide a superior algorithm that guarantees stability of the resulting linear system and searches for the best fit using a numerical, iterative scheme. The `iter` parameter tells `invfreqs` to end the iteration when the solution has converged, or after `iter` iterations, whichever comes first. `invfreqs` defines convergence as occurring when the norm of the (modified) gradient vector is less than `tol`, where `tol` is an optional parameter that defaults to 0.01. To obtain a weight vector of all ones, use



```
invfreqs(h,w,n,m,[],iter,tol)
```

`[b,a] = invfreqs(h,w,n,m,wt,iter,tol,'trace')` displays a textual progress report of the iteration.

`[b,a] = invfreqs(h,w,'complex',n,m,...)` creates a complex filter. In this case no symmetry is enforced, and the frequency is specified in radians between  $-\pi$  and  $\pi$ .

## Remarks

When building higher order models using high frequencies, it is important to scale the frequencies, dividing by a factor such as half the highest frequency present in `w`, so as to obtain well conditioned values of `a` and `b`. This corresponds to a rescaling of time.

## Examples

### Example 1

Convert a simple transfer function to frequency response data and then back to the original filter coefficients:

```
a = [1 2 3 2 1 4]; b = [1 2 3 2 3];
[h,w] = freqs(b,a,64);
[bb,aa] = invfreqs(h,w,4,5)
```

bb =

```
1.0000    2.0000    3.0000    2.0000    3.0000
```

aa =

```
1.0000    2.0000    3.0000    2.0000    1.0000    4.0000
```

Notice that `bb` and `aa` are equivalent to `b` and `a`, respectively. However, `aa` has poles in the left half-plane and thus the system is unstable. Use `invfreqs`'s iterative algorithm to find a stable approximation to the system:

```
[bbb,aaa] = invfreqs(h,w,4,5,[],30)

bbb =
    0.6816    2.1015    2.6694    0.9113   -0.1218

aaa =
    1.0000    3.4676    7.4060    6.2102    2.5413    0.0001
```

## Example 2

Suppose you have two vectors, `mag` and `phase`, that contain magnitude and phase data gathered in a laboratory, and a third vector `w` of frequencies. You can convert the data into a continuous-time transfer function using `invfreqs`:

```
[b,a] = invfreqs(mag.*exp(j*phase),w,2,3);
```

## Algorithm

By default, `invfreqs` uses an equation error method to identify the best model from the data. This finds `b` and `a` in

$$\min_{b,a} \sum_{k=1}^n wt(k)|h(k)A(w(k)) - B(w(k))|^2$$

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`). This algorithm is based on Levi [1]. Several variants have been suggested in the literature, where the weighting function `wt` gives less attention to high frequencies.

The superior (“output-error”) algorithm uses the damped Gauss-Newton method for iterative search [2], with the output of the first algorithm as the initial estimate. This solves the direct problem of minimizing the weighted sum of the squared error between the actual and the desired frequency response points.

$$\min_{b, a} \sum_{k=1}^n wt(k) \left| h(k) - \frac{B(w(k))}{A(w(k))} \right|^2$$

**References**

- [1] Levi, E.C., “Complex-Curve Fitting,” *IRE Trans. on Automatic Control*, Vol. AC-4 (1959), pp. 37-44.
- [2] Dennis, J.E., Jr., and R.B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, NJ: Prentice-Hall, 1983.

**See Also**

freqs, freqz, invfreqz, prony

# invfreqz

---

## Purpose

Identify discrete-time filter parameters from frequency response data

## Syntax

```
[b,a] = invfreqz(h,w,n,m)
[b,a] = invfreqz(h,w,n,m,wt)
[b,a] = invfreqz(h,w,n,m,wt,iter)
[b,a] = invfreqz(h,w,n,m,wt,iter,tol)
[b,a] = invfreqz(h,w,n,m,wt,iter,tol,'trace')
[b,a] = invfreqz(h,w,'complex',n,m,...)
```

## Description

`invfreqz` is the inverse operation of `freqz`; it finds a discrete-time transfer function that corresponds to a given complex frequency response. From a laboratory analysis standpoint, `invfreqz` can be used to convert magnitude and phase data into transfer functions.

`[b,a] = invfreqz(h,w,n,m)` returns the real numerator and denominator coefficients in vectors `b` and `a` of the transfer function

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \dots + a(m+1)z^{-m}}$$

whose complex frequency response is given in vector `h` at the frequency points specified in vector `w`. Scalars `n` and `m` specify the desired orders of the numerator and denominator polynomials.

Frequency is specified in radians between 0 and  $\pi$ , and the length of `h` must be the same as the length of `w`. `invfreqz` uses `conj(h)` at `-w` to ensure the proper frequency domain symmetry for a real filter.

`[b,a] = invfreqz(h,w,n,m,wt)` weights the fit-errors versus frequency, where `wt` is a vector of weighting factors the same length as `w`.

`[b,a] = invfreqz(h,w,n,m,wt,iter)` and

`[b,a] = invfreqz(h,w,n,m,wt,iter,tol)` provide a superior algorithm that guarantees stability of the resulting linear system and searches for the best fit using a numerical, iterative scheme. The `iter` parameter tells `invfreqz` to end the iteration when the solution has converged, or after `iter` iterations, whichever comes first. `invfreqz` defines convergence as occurring when the norm of the (modified) gradient vector is less than `tol`, where `tol` is an optional parameter that defaults to 0.01. To obtain a weight vector of all ones, use

```
invfreqz(h,w,n,m,[],iter,tol)
```

`[b,a] = invfreqz(h,w,n,m,wt,iter,tol,'trace')` displays a textual progress report of the iteration.

`[b,a] = invfreqz(h,w,'complex',n,m,...)` creates a complex filter. In this case no symmetry is enforced, and the frequency is specified in radians between  $-\pi$  and  $\pi$ .

## Examples

Convert a simple transfer function to frequency response data and then back to the original filter coefficients:

```
a = [1 2 3 2 1 4]; b = [1 2 3 2 3];
[h,w] = freqz(b,a,64);
[bb,aa] = invfreqz(h,w,4,5)
```

```
bb =
```

```
1.0000    2.0000    3.0000    2.0000    3.0000
```

```
aa =
```

```
1.0000    2.0000    3.0000    2.0000    1.0000    4.0000
```

Notice that `bb` and `aa` are equivalent to `b` and `a`, respectively. However, `aa` has poles outside the unit circle and thus the system is unstable. Use `invfreqz`'s iterative algorithm to find a stable approximation to the system:

```
[bbb,aaa] = invfreqz(h,w,4,5,[],30)
```

```
bbb =
```

```
0.2427    0.2788    0.0069    0.0971    0.1980
```

```
aaa =
```

```
1.0000   -0.8944    0.6954    0.9997   -0.8933    0.6949
```

## Algorithm

By default, `invfreqz` uses an equation error method to identify the best model from the data. This finds `b` and `a` in

$$\min_{b, a} \sum_{k=1}^n wt(k) |h(k)A(\omega(k)) - B(\omega(k))|^2$$

by creating a system of linear equations and solving them with the MATLAB \ operator. Here  $A(\omega(k))$  and  $B(\omega(k))$  are the Fourier transforms of the polynomials  $a$  and  $b$ , respectively, at the frequency  $\omega(k)$ , and  $n$  is the number of frequency points (the length of  $h$  and  $w$ ). This algorithm is based on Levi [1].

The superior (“output-error”) algorithm uses the damped Gauss-Newton method for iterative search [2], with the output of the first algorithm as the initial estimate. This solves the direct problem of minimizing the weighted sum of the squared error between the actual and the desired frequency response points.

$$\min_{b, a} \sum_{k=1}^n wt(k) \left| h(k) - \frac{B(\omega(k))}{A(\omega(k))} \right|^2$$

## References

- [1] Levi, E.C., “Complex-Curve Fitting,” *IRE Trans. on Automatic Control*, Vol. AC-4 (1959), pp. 37-44.
- [2] Dennis, J.E., Jr., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, 1983.

## See Also

freqs, freqz, invfreqz, prony

---

<b>Purpose</b>	Convert inverse sine parameters to reflection coefficients
<b>Syntax</b>	<code>k = is2rc(isin)</code>
<b>Description</b>	<code>k = is2rc(isin)</code> returns a vector of reflection coefficients <code>k</code> from a vector of inverse sine parameters <code>isin</code> .
<b>Examples</b>	<pre>isin = [0.2000 0.8727 0.0020 0.0052 -0.0052]; k = is2rc(isin)  k =     0.3090    0.9801    0.0031    0.0082   -0.0082</pre>
<b>References</b>	[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, <i>Discrete-Time Processing of Speech Signals</i> , Prentice-Hall, 1993.
<b>See Also</b>	<code>ac2rc</code> , <code>lar2rc</code> , <code>poly2rc</code> , <code>rc2is</code>

# kaiser

---

**Purpose** Kaiser window

**Syntax** `w = kaiser(L,beta)`

**Description** `w = kaiser(L,beta)` returns an L-point Kaiser ( $I_0 - \sinh$ ) window in the column vector `w`. `beta` is the Kaiser window  $\beta$  parameter that affects the sidelobe attenuation of the Fourier transform of the window. The default value for `beta` is 0.5.

To obtain a Kaiser window that designs an FIR filter with sidelobe height  $-\alpha$  dB, use the following  $\beta$ .

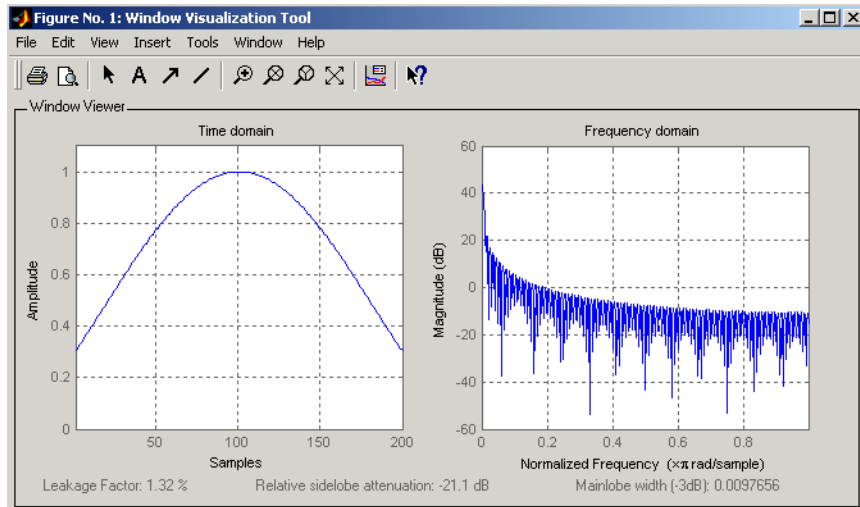
$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$

Increasing `beta` widens the main lobe and decreases the amplitude of the sidelobes (increases the attenuation).

**Examples** Create a 200-point Kaiser window with a `beta` of 2.5 and display the result using `WVTool`:

```
w = kaiser(200,2.5);  
wvtool(w)
```





## References

- [1] Kaiser, J.F., "Nonrecursive Digital Filter Design Using the  $I_0$  - sinh Window Function," *Proc. 1974 IEEE Symp. Circuits and Systems*, (April 1974), pp. 20-23.
- [2] *Selected Papers in Digital Signal Processing II*, IEEE Press, New York, 1975.
- [3] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 453.

## See Also

chebwin, gausswin, kaiserord, tukeywin, window, wintool, wvtool

# kaiserord

---

**Purpose** Kaiser window FIR filter design estimation parameters

**Syntax**

```
[n,Wn,beta,ftype] = kaiserord(f,a,dev)
[n,Wn,beta,ftype] = kaiserord(f,a,dev,fs)
c = kaiserord(f,a,dev,fs,'cell')
```

**Description** `kaiserord` returns a filter order `n` and beta parameter to specify a Kaiser window for use with the `fir1` function. Given a set of specifications in the frequency domain, `kaiserord` estimates the minimum FIR filter order that will approximately meet the specifications. `kaiserord` converts the given filter specifications into passband and stopband ripples and converts cutoff frequencies into the form needed for windowed FIR filter design.

`[n,Wn,beta,ftype] = kaiserord(f,a,dev)` finds the approximate order `n`, normalized frequency band edges `Wn`, and weights that meet input specifications `f`, `a`, and `dev`. `f` is a vector of band edges and `a` is a vector specifying the desired amplitude on the bands defined by `f`. The length of `f` is twice the length of `a`, minus 2. Together, `f` and `a` define a desired piecewise constant response function. `dev` is a vector the same size as `a` that specifies the maximum allowable error or deviation between the frequency response of the output filter and its desired amplitude, for each band. The entries in `dev` specify the passband ripple and the stopband attenuation. You specify each entry in `dev` as a positive number, representing absolute filter gain (not in decibels).

---

**Note** If, in the vector `dev`, you specify unequal deviations across bands, the minimum specified deviation is used, since the Kaiser window method is constrained to produce filters with minimum deviation in all of the bands.

---

`fir1` can use the resulting order `n`, frequency vector `Wn`, multiband magnitude type `ftype`, and the Kaiser window parameter `beta`. The `ftype` string is intended for use with `fir1`; it is equal to `'high'` for a highpass filter and `'stop'` for a bandstop filter. For multiband filters, it can be equal to `'dc-0'` when the first band is a stopband (starting at  $f = 0$ ) or `'dc-1'` when the first band is a passband.

To design an FIR filter `b` that approximately meets the specifications given by `kaiser` parameters `f`, `a`, and `dev`, use the following command.

```
b = fir1(n,Wn,kaiser(n+1,beta),ftype,'noscale')
```

`[n,Wn,beta,ftype] = kaiserord(f,a,dev,fs)` uses a sampling frequency `fs` in Hz. If you don't specify the argument `fs`, or if you specify it as the empty vector `[]`, it defaults to 2 Hz, and the Nyquist frequency is 1 Hz. You can use this syntax to specify band edges scaled to a particular application's sampling frequency. The frequency band edges in `f` must be from 0 to `fs/2`.

`c = kaiserord(f,a,dev,fs,'cell')` is a cell-array whose elements are the parameters to `fir1`.

---

**Note** In some cases, `kaiserord` underestimates or overestimates the order `n`. If the filter does not meet the specifications, try a higher order such as `n+1`, `n+2`, and so on, or a try lower order.

---

Results are inaccurate if the cutoff frequencies are near 0 or the Nyquist frequency, or if `dev` is large (greater than 10%).

---

## Remarks

Be careful to distinguish between the meanings of filter length and filter order. The filter *length* is the number of impulse response samples in the FIR filter. Generally, the impulse response is indexed from  $n = 0$  to  $n = L-1$ , where  $L$  is the filter length. The filter *order* is the highest power in a  $z$ -transform representation of the filter. For an FIR transfer function, this representation is a polynomial in  $z$ , where the highest power is  $z^{L-1}$  and the lowest power is  $z^0$ . The filter order is one less than the length ( $L-1$ ) and is also equal to the number of zeros of the  $z$  polynomial.

## Examples

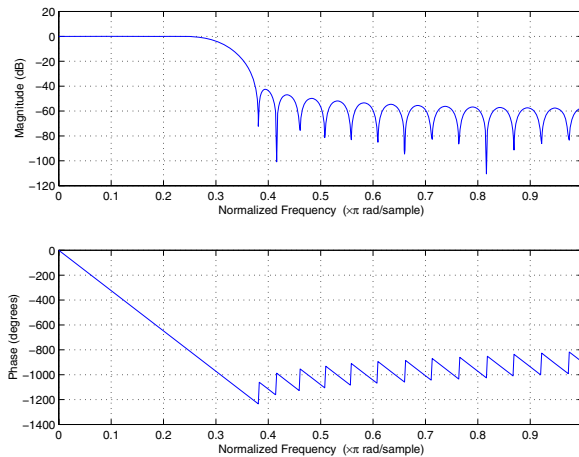
### Example 1

Design a lowpass filter with passband defined from 0 to 1 kHz and stopband defined from 1500 Hz to 4 kHz. Specify a passband ripple of 5% and a stopband attenuation of 40 dB:

```
fsamp = 8000;
```

# kaiserord

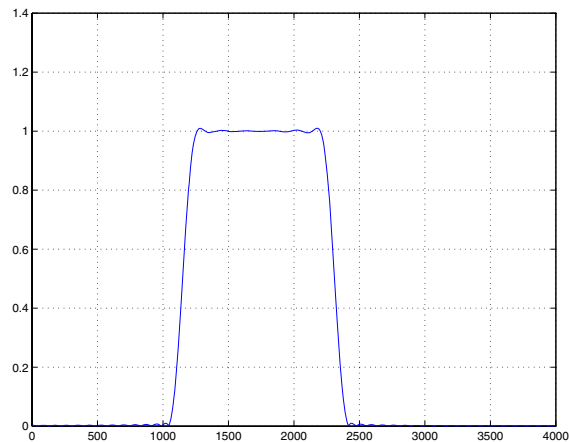
```
fcuts = [1000 1500];
mags = [1 0];
devs = [0.05 0.01];
[n,Wn,beta,ftype] = kaiserord(fcuts,mags,devs,fsamp);
hh = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
freqz(hh)
```



## Example 2

Design an odd-length bandpass filter (note that odd length means even order, so the input to `fir1` must be an even integer):

```
fsamp = 8000;
fcuts = [1000 1300 2210 2410];
mags = [0 1 0];
devs = [0.01 0.05 0.01];
[n,Wn,beta,ftype] = kaiserord(fcuts,mags,devs,fsamp);
n = n + rem(n,2);
hh = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
[H,f] = freqz(hh,1,1024,fsamp);
plot(f,abs(H)), grid on
```



### Example 3

Design a lowpass filter with a passband cutoff of 1500 Hz, a stopband cutoff of 2000 Hz, passband ripple of 0.01, stopband ripple of 0.1, and a sampling frequency of 8000 Hz:

```
[n,Wn,beta,ftype] = kaiserord([1500 2000],[1 0],...
                               [0.01 0.1],8000);
b = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
```

This is equivalent to

```
c = kaiserord([1500 2000],[1 0],[0.01 0.1],8000,'cell');
b = fir1(c{:});
```

### Algorithm

`kaiserord` uses empirically derived formulas for estimating the orders of lowpass filters, as well as differentiators and Hilbert transformers. Estimates for multiband filters (such as bandpass filters) are derived from the lowpass design formulas.

The design formulas that underlie the Kaiser window and its application to FIR filter design are

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$

where  $\alpha = -20\log_{10}\delta$  is the stopband attenuation expressed in decibels (recall that  $\delta_p = \delta_s$  is required).

The design formula is

$$n = \frac{\alpha - 7.95}{2.285(\Delta\omega)}$$

where  $n$  is the filter order and  $\Delta\omega$  is the width of the smallest transition region.

## References

- [1] Kaiser, J.F., "Nonrecursive Digital Filter Design Using the  $I_0$  - sinh Window Function," *Proc. 1974 IEEE Symp. Circuits and Systems*, (April 1974), pp. 20-23.
- [2] *Selected Papers in Digital Signal Processing II*, IEEE Press, New York, 1975, pp. 123-126.
- [3] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 458-562.

## See Also

`fir1`, `kaiser`, `firpmord`

---

<b>Purpose</b>	Convert log area ratio parameters to reflection coefficients
<b>Syntax</b>	$k = \text{lar2rc}(g)$
<b>Description</b>	$k = \text{lar2rc}(g)$ returns a vector of reflection coefficients $k$ from a vector of log area ratio parameters $g$ .
<b>Examples</b>	<pre>g = [0.6389    4.5989    0.0063    0.0163   -0.0163]; k = lar2rc(g)  k =     0.3090    0.9801    0.0031    0.0081   -0.0081</pre>
<b>References</b>	[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, “ <i>Discrete-Time Processing of Speech Signals</i> ,” Prentice-Hall, 1993.
<b>See Also</b>	ac2rc, is2rc, poly2rc, rc2lar

# latc2tf

---

**Purpose** Convert lattice filter parameters to transfer function form

**Syntax**

```
[num,den] = latc2tf(k,v)
[num,den] = latc2tf(k,'iioption')
num = latc2tf(k,'firoption')
num = latc2tf(k)
```

**Description** `[num,den] = latc2tf(k,v)` finds the transfer function numerator `num` and denominator `den` from the IIR lattice coefficients `k` and ladder coefficients `v`.

`[num,den] = latc2tf(k,'iioption')` produces an IIR filter transfer function according to the value of the string `'iioption'`:

- `'allpole'`: Produces an all-pole filter transfer function from the associated all-pole IIR lattice filter coefficients `k`.
- `'allpass'`: Produces an allpass filter transfer function from the associated allpass IIR lattice filter coefficients `k`.

`num = latc2tf(k,'firoption')` produces an FIR filter according to the value of the string `'firoption'`:

- `'min'`: Produces a minimum-phase FIR filter numerator from the associated minimum-phase FIR lattice filter coefficients `k`.
- `'max'`: Produces a maximum-phase FIR filter numerator from the associated maximum-phase FIR lattice filter coefficients `k`.
- `'FIR'`: Produces a general FIR filter numerator from the lattice filter coefficients `k` (default, if you leave off the string altogether).

**See Also** `latcfilt`, `tf2latc`



<b>Purpose</b>	Lattice and lattice-ladder filter implementation
<b>Syntax</b>	<pre>[f,g] = latcfilt(k,x) [f,g] = latcfilt(k,v,x) [f,g] = latcfilt(k,1,x) [f,g,zf] = latcfilt(...,'ic',zi)</pre>
<b>Description</b>	<p>When filtering data, lattice coefficients can be used to represent</p> <ul style="list-style-type: none"><li>• FIR filters</li><li>• All-pole IIR filters</li><li>• Allpass IIR filters</li><li>• General IIR filters</li></ul> <p><code>[f,g] = latcfilt(k,x)</code> filters <code>x</code> with the FIR lattice coefficients in the vector <code>k</code>. <code>f</code> is the forward lattice filter result and <code>g</code> is the backward filter result. If <math> k  \leq 1</math>, <code>f</code> corresponds to the minimum-phase output, and <code>g</code> corresponds to the maximum-phase output.</p> <p>If <code>k</code> and <code>x</code> are vectors, the result is a (signal) vector. Matrix arguments are permitted under the following rules:</p> <ul style="list-style-type: none"><li>• If <code>x</code> is a matrix and <code>k</code> is a vector, each column of <code>x</code> is processed through the lattice filter specified by <code>k</code>.</li><li>• If <code>x</code> is a vector and <code>k</code> is a matrix, each column of <code>k</code> is used to filter <code>x</code>, and a signal matrix is returned.</li><li>• If <code>x</code> and <code>k</code> are both matrices with the same number of columns, then the <math>i</math>th column of <code>k</code> is used to filter the <math>i</math>th column of <code>x</code>. A signal matrix is returned.</li></ul> <p><code>[f,g] = latcfilt(k,v,x)</code> filters <code>x</code> with the IIR lattice coefficients <code>k</code> and ladder coefficients <code>v</code>. <code>k</code> and <code>v</code> must be vectors, while <code>x</code> can be a signal matrix.</p> <p><code>[f,g] = latcfilt(k,1,x)</code> filters <code>x</code> with the IIR lattice specified by <code>k</code>, where <code>k</code> and <code>x</code> can be vectors or matrices. <code>f</code> is the all-pole lattice filter result and <code>g</code> is the allpass filter result.</p> <p><code>[f,g,zf] = latcfilt(...,'ic',zi)</code> accepts a length-<code>k</code> vector <code>zi</code> specifying the initial condition of the lattice states. Output <code>zf</code> is a length-<code>k</code> vector specifying the final condition of the lattice states.</p>

# latcfilt

---

## See Also

`filter`, `latc2tf`, `tf2latc`

**Purpose** Levinson-Durbin recursion

**Syntax**

```
a = levinson(r,n)
[a,e] = levinson(r,n)
[a,e,k] = levinson(r,n)
```

**Description** The Levinson-Durbin recursion is an algorithm for finding an all-pole IIR filter with a prescribed deterministic autocorrelation sequence. It has applications in filter design, coding, and spectral estimation. The filter that `levinson` produces is minimum phase.

`a = levinson(r,n)` finds the coefficients of an  $n$ th-order autoregressive linear process which has  $r$  as its autocorrelation sequence.  $r$  is a real or complex deterministic autocorrelation sequence. If  $r$  is a matrix, `levinson` finds the coefficients for each column of  $r$  and returns them in the rows of  $a$ .  $n$  is the order of denominator polynomial  $A(z)$ ; that is,  $a = [1 \ a(2) \ \dots \ a(n+1)]$ . The filter coefficients are ordered in descending powers of  $z$ .

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 + a(2)z^{-1} + \dots + a(n+1)z^{-n}}$$

`[a,e] = levinson(r,n)` returns the prediction error,  $e$ , of order  $n$ .

`[a,e,k] = levinson(r,n)` returns the reflection coefficients  $k$  as a column vector of length  $n$ .

---

**Note**  $k$  is computed internally while computing the  $a$  coefficients, so returning  $k$  simultaneously is more efficient than converting  $a$  to  $k$  with `tf2latc`.

---

**Algorithm** `levinson` solves the symmetric Toeplitz system of linear equations

# levinson

---

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(n)^* \\ r(2) & r(1) & \cdots & r(n-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(n) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(n+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(n+1) \end{bmatrix}$$

where  $r = [r(1) \dots r(n+1)]$  is the input autocorrelation vector, and  $r(i)^*$  denotes the complex conjugate of  $r(i)$ . The algorithm requires  $O(n^2)$  flops and is thus much more efficient than the MATLAB `\` command for large  $n$ . However, the `levinson` function uses `\` for low orders to provide the fastest possible execution.

## References

[1] Ljung, L., *System Identification: Theory for the User*, Prentice-Hall, 1987, pp. 278-280.

## See Also

`lpc`, `prony`, `rlevinson`, `schurrc`, `stmcb`

**Purpose** Transform lowpass analog filters to bandpass

**Syntax**  
`[bt,at] = lp2bp(b,a,Wo,Bw)`  
`[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw)`

**Description** lp2bp transforms analog lowpass filter prototypes with a cutoff angular frequency of 1 rad/s into bandpass filters with desired bandwidth and center frequency. The transformation is one step in the digital filter design process for the butter, cheby1, cheby2, and ellip functions.

lp2bp can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

**Transfer Function Form (Polynomial)**

`[bt,at] = lp2bp(b,a,Wo,Bw)` transforms an analog lowpass filter prototype given by polynomial coefficients into a bandpass filter with center frequency  $W_o$  and bandwidth  $B_w$ . Row vectors  $b$  and  $a$  specify the coefficients of the numerator and denominator of the prototype in descending powers of  $s$ .

$$\frac{b(s)}{a(s)} = \frac{b(1)s^n + \dots + b(n)s + b(n+1)}{a(1)s^m + \dots + a(m)s + a(m+1)}$$

Scalars  $W_o$  and  $B_w$  specify the center frequency and bandwidth in units of rad/s. For a filter with lower band edge  $w_1$  and upper band edge  $w_2$ , use  $W_o = \text{sqrt}(w_1*w_2)$  and  $B_w = w_2 - w_1$ .

lp2bp returns the frequency transformed filter in row vectors  $bt$  and  $at$ .

**State-Space Form**

`[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw)` converts the continuous-time state-space lowpass filter prototype in matrices  $A, B, C, D$  shown below

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

into a bandpass filter with center frequency  $\omega_0$  and bandwidth  $B_w$ . For a filter with lower band edge  $\omega_1$  and upper band edge  $\omega_2$ , use  $\omega_0 = \sqrt{\omega_1 \omega_2}$  and  $B_w = \omega_2 - \omega_1$ .

The bandpass filter is returned in matrices  $A_t$ ,  $B_t$ ,  $C_t$ ,  $D_t$ .

## Algorithm

lp2bp is a highly accurate state-space formulation of the classic analog filter frequency transformation. Consider the state-space system

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

where  $u$  is the input,  $x$  is the state vector, and  $y$  is the output. The Laplace transform of the first equation (assuming zero initial conditions) is

$$sX(s) = AX(s) + BU(s)$$

Now if a bandpass filter is to have center frequency  $\omega_0$  and bandwidth  $B_w$ , the standard  $s$ -domain transformation is

$$s = Q(p^2 + 1)/p$$

where  $Q = \omega_0/B_w$  and  $p = s/\omega_0$ . Substituting this for  $s$  in the Laplace transformed state-space equation, and considering the operator  $p$  as  $d/dt$  results in

$$Q\dot{x} + Qx = \dot{A}x + B\dot{u}$$

or

$$Q\ddot{x} - \dot{A}x - B\dot{u} = -Qx$$

Now define

$$Q\dot{\omega} = -Qx$$

which, when substituted, leads to

$$Q\dot{x} = Ax + Q\omega + Bu$$

The last two equations give equations of state. Write them in standard form and multiply the differential equations by  $\omega_0$  to recover the time/frequency scaling represented by  $p$  and find state matrices for the bandpass filter:

```
Q = Wo/Bw; [ma,m] = size(A);  
At = Wo*[A/Q eye(ma,m); -eye(ma,m) zeros(ma,m)];  
Bt = Wo*[B/Q; zeros(ma,n)];  
Ct = [C zeros(mc,ma)];  
Dt = d;
```

If the input to `lp2bp` is in transfer function form, the function transforms it into state-space form before applying this algorithm.

## See Also

`bilinear`, `impinvar`, `lp2bs`, `lp2hp`, `lp2lp`

# lp2bs

---

**Purpose** Transform lowpass analog filters to bandstop

**Syntax**  
[bt,at] = lp2bs(b,a,Wo,Bw)  
[At,Bt,Ct,Dt] = lp2bs(A,B,C,D,Wo,Bw)

**Description** lp2bs transforms analog lowpass filter prototypes with a cutoff angular frequency of 1 rad/s into bandstop filters with desired bandwidth and center frequency. The transformation is one step in the digital filter design process for the butter, cheby1, cheby2, and ellip functions.

lp2bs can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

## Transfer Function Form (Polynomial)

[bt,at] = lp2bs(b,a,Wo,Bw) transforms an analog lowpass filter prototype given by polynomial coefficients into a bandstop filter with center frequency Wo and bandwidth Bw. Row vectors b and a specify the coefficients of the numerator and denominator of the prototype in descending powers of s.

$$\frac{b(s)}{a(s)} = \frac{b(1)s^n + \dots + b(n)s + b(n+1)}{a(1)s^m + \dots + a(m)s + a(m+1)}$$

Scalars Wo and Bw specify the center frequency and bandwidth in units of radians/second. For a filter with lower band edge w1 and upper band edge w2, use Wo = sqrt(w1\*w2) and Bw = w2-w1.

lp2bs returns the frequency transformed filter in row vectors bt and at.

## State-Space Form

[At,Bt,Ct,Dt] = lp2bs(A,B,C,D,Wo,Bw) converts the continuous-time state-space lowpass filter prototype in matrices A, B, C, D shown below

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$



into a bandstop filter with center frequency  $\omega_0$  and bandwidth  $B_w$ . For a filter with lower band edge  $\omega_1$  and upper band edge  $\omega_2$ , use  $\omega_0 = \sqrt{\omega_1 \omega_2}$  and  $B_w = \omega_2 - \omega_1$ .

The bandstop filter is returned in matrices  $A_t$ ,  $B_t$ ,  $C_t$ ,  $D_t$ .

## Algorithm

lp2bs is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a bandstop filter is to have center frequency  $\omega_0$  and bandwidth  $B_w$ , the standard  $s$ -domain transformation is

$$s = \frac{p}{Q(p^2 + 1)}$$

where  $Q = \omega_0/B_w$  and  $p = s/\omega_0$ . The state-space version of this transformation is

$$\begin{aligned} Q &= \omega_0/B_w; \\ A_t &= [\omega_0/Q \cdot \text{inv}(A) \quad \omega_0 \cdot \text{eye}(m_a); -\omega_0 \cdot \text{eye}(m_a) \quad \text{zeros}(m_a)]; \\ B_t &= -[\omega_0/Q \cdot (A \ B); \text{zeros}(m_a, n)]; \\ C_t &= [C/A \quad \text{zeros}(m_c, m_a)]; \\ D_t &= D - C/A \cdot B; \end{aligned}$$

See lp2bp for a derivation of the bandpass version of this transformation.

## See Also

bilinear,impinvar,lp2bp,lp2hp,lp2lp

# lp2hp

---

**Purpose** Transform lowpass analog filters to highpass

**Syntax**  
[bt,at] = lp2hp(b,a,Wo)  
[At,Bt,Ct,Dt] = lp2hp(A,B,C,D,Wo)

**Description** lp2hp transforms analog lowpass filter prototypes with a cutoff angular frequency of 1 rad/s into highpass filters with desired cutoff angular frequency. The transformation is one step in the digital filter design process for the butter, cheby1, cheby2, and ellip functions.

The lp2hp function can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

## Transfer Function Form (Polynomial)

[bt,at] = lp2hp(b,a,Wo) transforms an analog lowpass filter prototype given by polynomial coefficients into a highpass filter with cutoff angular frequency Wo. Row vectors b and a specify the coefficients of the numerator and denominator of the prototype in descending powers of s.

$$\frac{b(s)}{a(s)} = \frac{b(1)s^n + \dots + b(n)s + b(n+1)}{a(1)s^m + \dots + a(m)s + a(m+1)}$$

Scalar Wo specifies the cutoff angular frequency in units of radians/second. The frequency transformed filter is returned in row vectors bt and at.

## State-Space Form

[At,Bt,Ct,Dt] = lp2hp(A,B,C,D,Wo) converts the continuous-time state-space lowpass filter prototype in matrices A, B, C, D below

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

into a highpass filter with cutoff angular frequency Wo. The highpass filter is returned in matrices At, Bt, Ct, Dt.

**Algorithm**

lp2hp is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a highpass filter is to have cutoff angular frequency  $\omega_0$ , the standard s-domain transformation is

$$s = \frac{\omega_0}{p}$$

The state-space version of this transformation is

$$\begin{aligned}A_t &= W_0 \text{inv}(A); \\B_t &= -W_0(A \setminus B); \\C_t &= C/A; \\D_t &= D - C/A * B;\end{aligned}$$

See lp2bp for a derivation of the bandpass version of this transformation.

**See Also**

bilinear,impinvar,lp2bp,lp2bs,lp2lp

**Purpose** Change cutoff frequency for lowpass analog filter

**Syntax**  
[bt,at] = lp2lp(b,a,Wo)  
[At,Bt,Ct,Dt] = lp2lp(A,B,C,D,Wo)

**Description** lp2lp transforms an analog lowpass filter prototype with a cutoff angular frequency of 1 rad/s into a lowpass filter with any specified cutoff angular frequency. The transformation is one step in the digital filter design process for the butter, cheby1, cheby2, and ellip functions.

The lp2lp function can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

### Transfer Function Form (Polynomial)

[bt,at] = lp2lp(b,a,Wo) transforms an analog lowpass filter prototype given by polynomial coefficients into a lowpass filter with cutoff angular frequency Wo. Row vectors b and a specify the coefficients of the numerator and denominator of the prototype in descending powers of s.

$$\frac{b(s)}{a(s)} = \frac{b(1)s^n + \dots + b(n)s + b(n+1)}{a(1)s^m + \dots + a(m)s + a(m+1)}$$

Scalar Wo specifies the cutoff angular frequency in units of radians/second. lp2lp returns the frequency transformed filter in row vectors bt and at.

### State-Space Form

[At,Bt,Ct,Dt] = lp2lp(A,B,C,D,Wo) converts the continuous-time state-space lowpass filter prototype in matrices A, B, C, D below

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

into a lowpass filter with cutoff angular frequency Wo. lp2lp returns the lowpass filter in matrices At, Bt, Ct, Dt.

**Algorithm**

lp2lp is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a lowpass filter is to have cutoff angular frequency  $\omega_0$ , the standard s-domain transformation is

$$s = p/\omega_0$$

The state-space version of this transformation is

$$\begin{aligned}A_t &= W_0 * A; \\B_t &= W_0 * B; \\C_t &= C; \\D_t &= D;\end{aligned}$$

See lp2bp for a derivation of the bandpass version of this transformation.

**See Also**

bilinear,impinvar,lp2bp,lp2bs,lp2hp

**Purpose** Linear prediction filter coefficients

**Syntax** `[a,g] = lpc(x,p)`

**Description** `lpc` determines the coefficients of a forward linear predictor by minimizing the prediction error in the least squares sense. It has applications in filter design and speech coding.

`[a,g] = lpc(x,p)` finds the coefficients of a  $p$ th-order linear predictor (FIR filter) that predicts the current value of the real-valued time series  $x$  based on past samples.

$$\hat{x}(n) = -a(2)x(n-1) - a(3)x(n-2) - \dots - a(p+1)x(n-p)$$

$p$  is the order of the prediction filter polynomial,  $a = [1 \ a(2) \ \dots \ a(p+1)]$ . If  $p$  is unspecified, `lpc` uses as a default  $p = \text{length}(x) - 1$ . If  $x$  is a matrix containing a separate signal in each column, `lpc` returns a model estimate for each column in the rows of matrix  $a$  and a row vector of prediction error variances  $g$ . The length of  $p$  must be less than or equal to the length of  $x$ .

**Examples** Estimate a data series using a third-order forward predictor, and compare to the original signal.

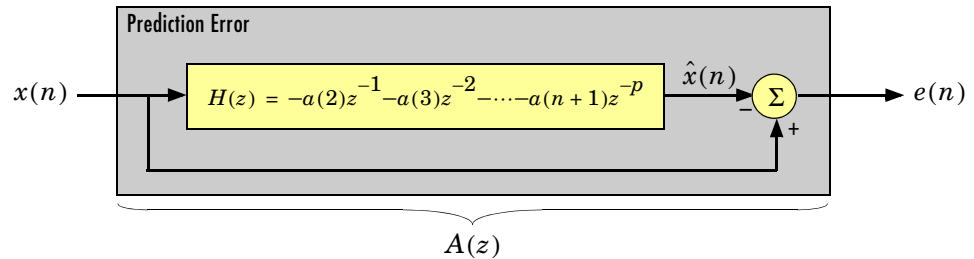
First, create the signal data as the output of an autoregressive process driven by white noise. Use the last 4096 samples of the AR process output to avoid start-up transients:

```
randn('state',0);  
noise = randn(50000,1); % Normalized white Gaussian noise  
x = filter(1,[1 1/2 1/3 1/4],noise);  
x = x(45904:50000);
```

Compute the predictor coefficients, estimated signal, prediction error, and autocorrelation sequence of the prediction error:

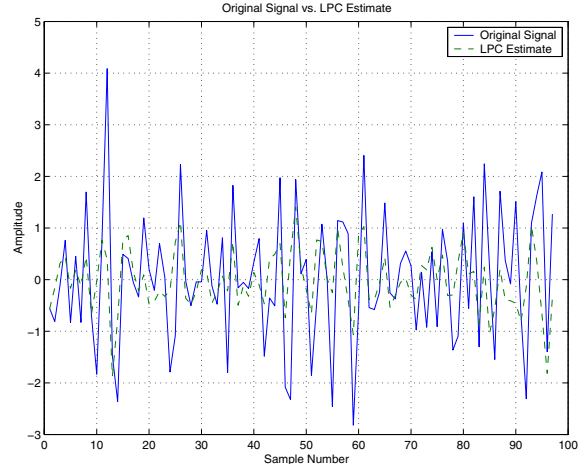
```
a = lpc(x,3);  
  
est_x = filter([0 -a(2:end)],1,x); % Estimated signal  
e = x - est_x; % Prediction error  
[acs,lags] = xcorr(e,'coeff'); % ACS of prediction error
```

The prediction error,  $e(n)$ , can be viewed as the output of the prediction error filter  $A(z)$  shown below, where  $H(z)$  is the optimal linear predictor,  $x(n)$  is the input signal, and  $\hat{x}(n)$  is the predicted signal.



Compare the predicted signal to the original signal:

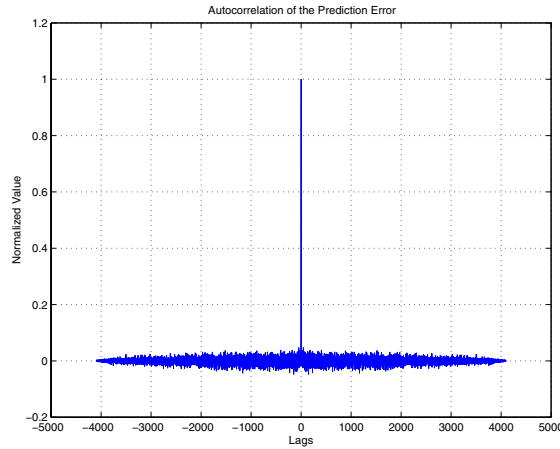
```
plot(1:97,x(4001:4097),1:97,est_x(4001:4097),'--');
title('Original Signal vs. LPC Estimate');
xlabel('Sample Number'); ylabel('Amplitude'); grid;
legend('Original Signal','LPC Estimate')
```



Look at the autocorrelation of the prediction error:

```
plot(lags,acs);
title('Autocorrelation of the Prediction Error');
xlabel('Lags'); ylabel('Normalized Value'); grid;
```

The prediction error is approximately white Gaussian noise, as expected for a third-order AR input process.



## Algorithm

`lpc` uses the autocorrelation method of autoregressive (AR) modeling to find the filter coefficients. The generated filter might not model the process exactly even if the data sequence is truly an AR process of the correct order. This is because the autocorrelation method implicitly windows the data, that is, it assumes that signal samples beyond the length of  $x$  are 0.

`lpc` computes the least squares solution to

$$Xa \approx b$$

where

$$X = \begin{bmatrix} x(1) & 0 & \dots & 0 \\ x(2) & x(1) & \ddots & \vdots \\ \vdots & x(2) & \ddots & 0 \\ x(m) & \vdots & \ddots & x(1) \\ 0 & x(m) & \ddots & x(2) \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & x(m) \end{bmatrix}, \quad a = \begin{bmatrix} 1 \\ a(2) \\ \vdots \\ a(p+1) \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$



and  $m$  is the length of  $x$ . Solving the least squares problem via the normal equations

$$X^H X a = X^H b$$

leads to the Yule-Walker equations

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \ddots & \vdots \\ \vdots & \ddots & \ddots & r(2)^* \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(p+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(p+1) \end{bmatrix}$$

where  $r = [r(1) \ r(2) \ \dots \ r(p+1)]$  is an autocorrelation estimate for  $x$  computed using `xcorr`. The Yule-Walker equations are solved in  $O(p^2)$  flops by the Levinson-Durbin algorithm (see `levinson`).

## References

[1] Jackson, L.B., *Digital Filters and Signal Processing*, Second Edition, Kluwer Academic Publishers, 1989. pp. 255-257.

## See Also

`aryule`, `levinson`, `prony`, `pyulear`, `stmcb`

# lsf2poly

---

**Purpose** Convert line spectral frequencies to prediction filter coefficients

**Syntax** `a = lsf2poly(lsf)`

**Description** `a = lsf2poly(lsf)` returns a vector `a` containing the prediction filter coefficients from a vector `lsf` of line spectral frequencies.

**Examples**

```
lsf = [0.7842    1.5605    1.8776    1.8984    2.3593];  
a = lsf2poly(lsf)  
  
a =  
    1.0000    0.6148    0.9899    0.0001    0.0031   -0.0081
```

**References**

[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, “*Discrete-Time Processing of Speech Signals*,” Prentice-Hall, 1993.

[2] Rabiner, L.R., and R.W. Schafer, “*Digital Processing of Speech Signals*,” Prentice-Hall, 1978.

**See Also** `ac2poly`, `poly2lsf`, `rc2poly`

**Purpose**

Generalized digital Butterworth filter design

**Syntax**

```
[b,a] = maxflat(n,m,Wn)
b = maxflat(n,'sym',Wn)
[b,a,b1,b2] = maxflat(n,m,Wn)
[b,a,b1,b2,sos,g] = maxflat(n,m,Wn)
[...] = maxflat(n,m,Wn,'design_flag')
```

**Description**

`[b,a] = maxflat(n,m,Wn)` is a lowpass Butterworth filter with numerator and denominator coefficients `b` and `a` of orders `n` and `m` respectively. `Wn` is the normalized cutoff frequency at which the magnitude response of the filter is equal to  $1/\sqrt{2}$  (approx. -3 dB). `Wn` must be between 0 and 1, where 1 corresponds to the Nyquist frequency.

`b = maxflat(n,'sym',Wn)` is a symmetric FIR Butterworth filter. `n` must be even, and `Wn` is restricted to a subinterval of  $[0,1]$ . The function raises an error if `Wn` is specified outside of this subinterval.

`[b,a,b1,b2] = maxflat(n,m,Wn)` returns two polynomials `b1` and `b2` whose product is equal to the numerator polynomial `b` (that is, `b = conv(b1,b2)`). `b1` contains all the zeros at  $z = -1$ , and `b2` contains all the other zeros.

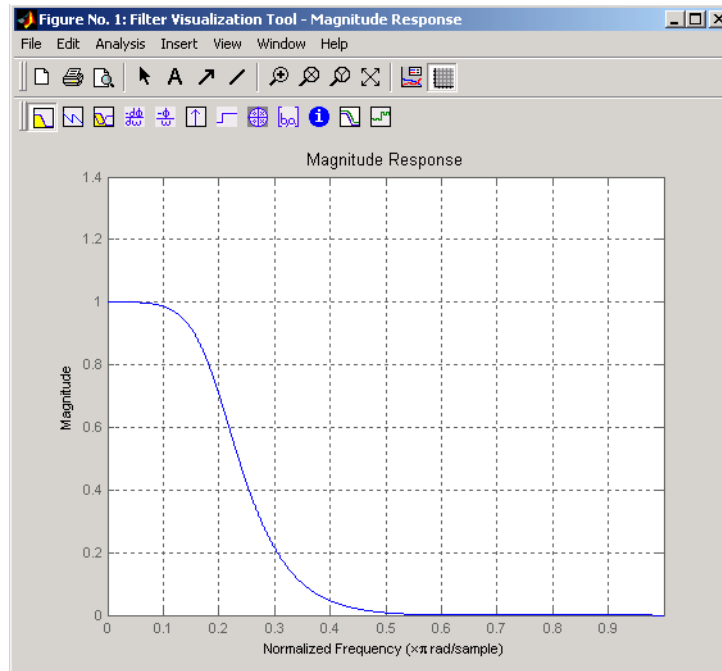
`[b,a,b1,b2,sos,g] = maxflat(n,m,Wn)` returns the second-order sections representation of the filter as the filter matrix `sos` and the gain `g`.

`[...] = maxflat(n,m,Wn,'design_flag')` enables you to monitor the filter design, where `'design_flag'` is

- `'trace'` for a textual display of the design table used in the design
- `'plots'` for plots of the filter's magnitude, group delay, and zeros and poles
- `'both'` for both the textual display and plots

**Examples**

```
n = 10; m = 2; Wn = 0.2;
[b,a] = maxflat(n,m,Wn)
fvtool(b,a) % Display the magnitude plot
```



## Algorithm

The method consists of the use of formulae, polynomial root finding, and a transformation of polynomial roots.

## References

[1] Selesnick, I.W., and C.S. Burrus, "Generalized Digital Butterworth Filter Design," *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing*, Vol. 3 (May 1996).

## See Also

butter, filter, freqz

<b>Purpose</b>	1-D median filtering
<b>Syntax</b>	<pre>y = medfilt1(x,n) y = medfilt1(x,n,blkosz) y = medfilt1(x,n,blkosz,dim)</pre>
<b>Description</b>	<p><code>y = medfilt1(x,n)</code> applies an order <math>n</math> one-dimensional median filter to vector <math>x</math>; the function considers the signal to be 0 beyond the end points. Output <math>y</math> has the same length as <math>x</math>.</p> <p>For <math>n</math> odd, <math>y(k)</math> is the median of <math>x(k - (n - 1) / 2 : k + (n - 1) / 2)</math>.</p> <p>For <math>n</math> even, <math>y(k)</math> is the median of <math>x(k - n / 2), x(k - (n / 2) + 1), \dots, x(k + (n / 2) - 1)</math>. In this case, <code>medfilt1</code> sorts the numbers, then takes the average of the <math>n / 2</math> and <math>(n / 2) + 1</math> elements.</p> <p>The default for <math>n</math> is 3.</p> <p><code>y = medfilt1(x,n,blkosz)</code> uses a for-loop to compute <code>blkosz</code> (block size) output samples at a time. Use <code>blkosz &lt;&lt; length(x)</code> if you are low on memory, since <code>medfilt1</code> uses a working matrix of size <math>n</math>-by-<code>blkosz</code>. By default, <code>blkosz = length(x)</code>; this provides the fastest execution if you have sufficient memory.</p> <p>If <math>x</math> is a matrix, <code>medfilt1</code> median filters its columns using</p> <pre>y(:,i) = medfilt1(x(:,i),n,blkosz)</pre> <p>in a loop over the columns of <math>x</math>.</p> <p><code>y = medfilt1(x,n,blkosz,dim)</code> specifies the dimension, <code>dim</code>, along which the filter operates.</p>
<b>References</b>	[1] Pratt, W.K., <i>Digital Image Processing</i> , John Wiley & Sons, 1978, pp. 330-333.
<b>See Also</b>	<code>filter</code> , <code>medfilt2</code> , <code>median</code>

# modulate

---

**Purpose** Modulation for communications simulation

**Syntax**

```
y = modulate(x,fc,fs,'method')
y = modulate(x,fc,fs,'method',opt)
[y,t] = modulate(x,fc,fs)
```

**Description** `y = modulate(x,fc,fs,'method')` and

`y = modulate(x,fc,fs,'method',opt)` modulate the real message signal `x` with a carrier frequency `fc` and sampling frequency `fs`, using one of the options listed below for `'method'`. Note that some methods accept an option, `opt`.

<b>Method</b>	<b>Description</b>
---------------	--------------------

<code>amdsb-sc</code>	<b>Amplitude modulation, double sideband, suppressed carrier.</b>
-----------------------	---

or	Multiplies <code>x</code> by a sinusoid of frequency <code>fc</code> .
----	--

<code>am</code>	$y = x \cdot \cos(2\pi \cdot fc \cdot t)$
-----------------	---

<code>amdsb-tc</code>	<b>Amplitude modulation, double sideband, transmitted carrier.</b> Subtracts scalar <code>opt</code> from <code>x</code> and multiplies the result by a sinusoid of frequency <code>fc</code> .
-----------------------	--

	$y = (x - opt) \cdot \cos(2\pi \cdot fc \cdot t)$
--	---

If the `opt` parameter is not present, `modulate` uses a default of `min(min(x))` so that the message signal `(x - opt)` is entirely nonnegative and has a minimum value of 0.

<code>amssb</code>	<b>Amplitude modulation, single sideband.</b> Multiplies <code>x</code> by a sinusoid of frequency <code>fc</code> and adds the result to the Hilbert transform of <code>x</code> multiplied by a phase shifted sinusoid of frequency <code>fc</code> .
--------------------	---

	$y = x \cdot \cos(2\pi \cdot fc \cdot t) + \text{imag}(\text{hilbert}(x)) \cdot \sin(2\pi \cdot fc \cdot t)$
--	--

This effectively removes the upper sideband.

- fm**      **Frequency modulation.** Creates a sinusoid with instantaneous frequency that varies with the message signal  $x$ .
- $$y = \cos(2\pi fc t + \text{opt} \cdot \text{cumsum}(x))$$
- $\text{cumsum}$  is a rectangular approximation to the integral of  $x$ . `modulate` uses `opt` as the constant of frequency modulation. If `opt` is not present, `modulate` uses a default of
- $$\text{opt} = (fc/fs) \cdot 2\pi / (\max(\max(x)))$$
- so the maximum frequency excursion from  $fc$  is  $fc$  Hz.
- pm**      **Phase modulation.** Creates a sinusoid of frequency  $fc$  whose phase varies with the message signal  $x$ .
- $$y = \cos(2\pi fc t + \text{opt} \cdot x)$$
- `modulate` uses `opt` as the constant of phase modulation. If `opt` is not present, `modulate` uses a default of
- $$\text{opt} = \pi / (\max(\max(x)))$$
- so the maximum phase excursion is  $\pi$  radians.
- pwm**      **Pulse-width modulation.** Creates a pulse-width modulated signal from the pulse widths in  $x$ . The elements of  $x$  must be between 0 and 1, specifying the width of each pulse in fractions of a period. The pulses start at the beginning of each period, that is, they are left justified.
- $$\text{modulate}(x, fc, fs, \text{'pwm'}, \text{'centered'})$$
- yields pulses centered at the beginning of each period.  $y$  is length  $\text{length}(x) \cdot fs / fc$ .
- ppm**      **Pulse-position modulation.** Creates a pulse-position modulated signal from the pulse positions in  $x$ . The elements of  $x$  must be between 0 and 1, specifying the left edge of each pulse in fractions of a period. `opt` is a scalar between 0 and 1 that specifies the length of each pulse in fractions of a period. The default for `opt` is 0.1.  $y$  is length  $\text{length}(x) \cdot fs / fc$ .
- qam**      **Quadrature amplitude modulation.** Creates a quadrature amplitude modulated signal from signals  $x$  and `opt`.
- $$y = x \cdot \cos(2\pi fc t) + \text{opt} \cdot \sin(2\pi fc t)$$
- `opt` must be the same size as  $x$ .

# modulate

---

If you do not specify *'method'*, then `modulate` assumes `am`. Except for the `pwm` and `ptm` cases, `y` is the same size as `x`.

If `x` is an array, `modulate` modulates its columns.

`[y,t] = modulate(x,fc,fs)` returns the internal time vector `t` that `modulate` uses in its computations.

## See Also

`demod`, `vco`, `fskdemod`, `genqamdemod`, `mskdemod`, `pamdemod`, `pmdemod`, `qamdemod`



**Purpose**

Magnitude squared coherence

**Syntax**

```
Cxy = mscohere(x,y)
Cxy = mscohere(x,y>window)
Cxy = mscohere(x,y>window,noverlap)
[Cxy,W] = mscohere(x,y>window,noverlap,nfft)
[Cxy,F] = mscohere(x,y>window,noverlap,nfft,fs)
[...] = mscohere(x,y,...,'whole')
mscohere(x,y,...)
```

**Description**

`Cxy = mscohere(x,y)` finds the magnitude squared coherence estimate `Cxy` of the input signals `x` and `y` using Welch's averaged, modified periodogram method. The magnitude squared coherence estimate is a function of frequency with values between 0 and 1 that indicates how well `x` corresponds to `y` at each frequency. The coherence is a function of the power spectral density ( $P_{xx}$  and  $P_{yy}$ ) of `x` and `y` and the cross power spectral density ( $P_{xy}$ ) of `x` and `y`.

$$C_{xy}(f) = \frac{|P_{xy}(f)|^2}{P_{xx}(f)P_{yy}(f)}$$

`x` and `y` must be the same length. For real `x` and `y`, `mscohere` returns a one-sided coherence estimate and for complex `x` or `y`, it returns a two-sided estimate.

mscohere uses the following default values:

Parameter	Description	Default Value
nfft	<p>FFT length which determines the frequencies at which the coherence is estimated</p> <p>For real <math>x</math> and <math>y</math>, the length of <math>C_{xy}</math> is <math>(nfft/2+1)</math> if <math>nfft</math> is even or <math>(nfft+1)/2</math> if <math>nfft</math> is odd. For complex <math>x</math> or <math>y</math>, the length of <math>C_{xy}</math> is <math>nfft</math>.</p> <p>If <math>nfft</math> is greater than the signal length, the data is zero-padded. If <math>nfft</math> is less than the signal length, the segment is wrapped using datawrap so that the length is equal to <math>nfft</math>.</p>	Maximum of 256 or the next power of 2 greater than the length of each section of $x$ or $y$
fs	Sampling frequency	1
window	Windowing function and number of samples to use for each section	Periodic Hamming window of length to obtain eight equal sections of $x$ and $y$
noverlap	Number of samples by which the sections overlap	Value to obtain 50% overlap

---

**Note** You can use the empty matrix `[]` to specify the default value for any input argument except  $x$  or  $y$ . For example, `Pxy = mscohere(x,y,[],[],128)` uses a Hamming window, default `noverlap` to obtain 50% overlap, and the specified 128 `nfft`.

---

`Cxy = mscohere(x,y>window)` specifies a windowing function, divides `x` and `y` into equal overlapping sections of the specified window length, and windows each section using the specified window function. If you supply a scalar for `window`, `Cxy` uses a Hamming window of that length. `mscohere` zero pads the sections if the window length exceeds `nfft`.

`Cxy = mscohere(x,y>window,noverlap)` overlaps the sections of `x` by `noverlap` samples. `noverlap` must be an integer smaller than the length of `window`.

`[Pxy,W] = mscohere(x,y>window,noverlap,nfft)` uses the specified FFT length `nfft` to calculate the coherence estimate. It also returns `W`, which is the vector of normalized frequencies (in rad/sample) at which the coherence is estimated. For real `x` and `y`, `Cxy` length is  $(nfft/2 + 1)$  if `nfft` is even and if `nfft` is odd, the length is  $(nfft+1)/2$ . For complex `x` or `y`, the length of `Cxy` is `nfft`. For real signals, the range of `W` is  $[0, \pi]$  when `nfft` is even and  $[0, \pi)$  when `nfft` is odd. For complex signals, the range of `W` is  $[0, 2*\pi)$ .

`[Cxy,F] = mscohere(x,y>window,noverlap,nfft,fs)` returns `Cxy` as a function of frequency and a vector `F` of frequencies at which the coherence is estimated. `fs` is the sampling frequency in Hz. For real signals, the range of `F` is  $[0, fs/2]$  when `nfft` is even and  $[0, fs/2)$  when `nfft` is odd. For complex signals, the range of `F` is  $[0, fs)$ .

`[...] = mscohere(x,y,...,'whole')` returns a coherence estimate with frequencies that range over the whole Nyquist interval. Specifying `'half'` uses half the Nyquist interval.

`mscohere(...)` plots the magnitude squared coherence versus frequency in the current figure window.

---

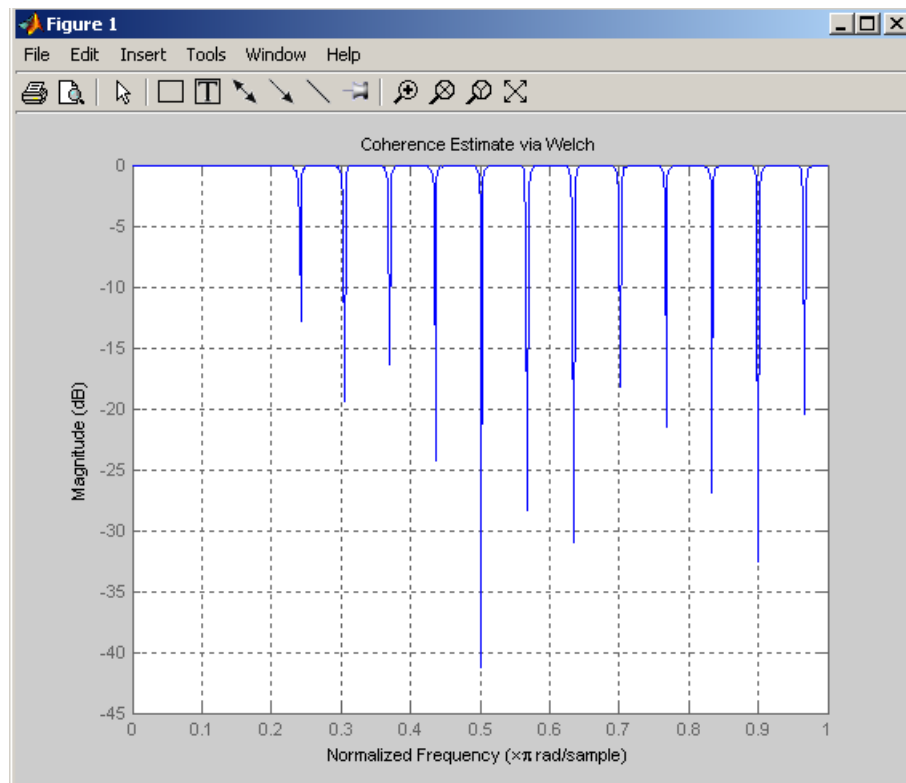
**Note** If you use `mscohere` on two linearly related signals [1] with a single, non-overlapping window, the output for all frequencies is `Cxy = 1`.

---

## Examples

Compute and plot the coherence estimate between two colored noise sequences  $x$  and  $y$ :

```
randn('state',0);  
h = fir1(30,0.2,rectwin(31));  
h1 = ones(1,10)/sqrt(10);  
r = randn(16384,1);  
x = filter(h1,1,r);  
y = filter(h,1,x);  
mscohere(x,y,hanning(1024),512,1024)
```



## Algorithm

`mscohere` estimates the magnitude squared coherence function [2] using Welch's averaged periodogram method (see references [3] and [4]).

**References**

- [1] Stoica, P., and R. Moses. *Introduction to Spectral Analysis*. Upper Saddle River, NJ: Prentice-Hall, 1997. Pgs. 61-64.
- [2] Kay, S.M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice-Hall, 1988. Pg. 454.
- [3] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.
- [4] Welch, P.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 (June 1967). Pgs. 70-73.

**See Also**

cpsd, periodogram, pwelch, spectrum, tfestimate

# nuttallwin

## Purpose

Nuttall-defined minimum 4-term Blackman-Harris window

## Syntax

```
w = nuttallwin(L)
```

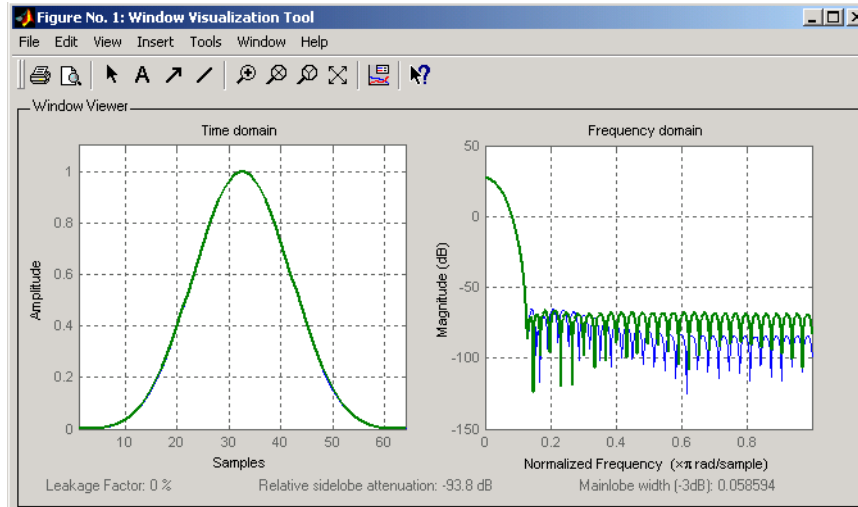
## Description

`w = nuttallwin(L)` returns a minimum, L-point, 4-term Blackman-harris window in the column vector `w`. The window is minimum in the sense that its maximum sidelobes are minimized. The coefficients for this window differ from the Blackman-Harris window coefficients computed with `blackmanharris` and produce slightly lower sidelobes.

## Examples

Compare 64-point Blackman-Harris and Nuttall's Blackman-Harris windows and plot them using WVTTool:

```
L = 64;  
w = blackmanharris(L);  
y = nuttallwin(L);  
wvtool(w,y)
```



The maximum difference between the two windows is

$$\max(\text{abs}(y-w))$$

ans =

0.0099

## Algorithm

The equation for computing the coefficients of a minimum 4-term Blackman-harris window, according to Nuttall, is

$$w(n) = a_0 - a_1 \cos\left(2\pi\frac{n}{N}\right) + a_2 \cos\left(4\pi\frac{n}{N}\right) - a_3 \cos\left(6\pi\frac{n}{N}\right)$$

where  $0 \leq n \leq N$  and the window length  $L = N+1$ .

The coefficients for this window are

$$a_0 = 0.3635819$$

$$a_1 = 0.4891775$$

$$a_2 = 0.1365995$$

$$a_3 = .0106411$$

## References

[1] Nuttall, Albert H. "Some Windows with Very Good Sidelobe Behavior." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. ASSP-29 (February 1981). pp. 84-91.

## See Also

barthannwin, bartlett, blackmanharris, bohmanwin, parzenwin, rectwin, triang, window, wintool, wvtool

# parzenwin

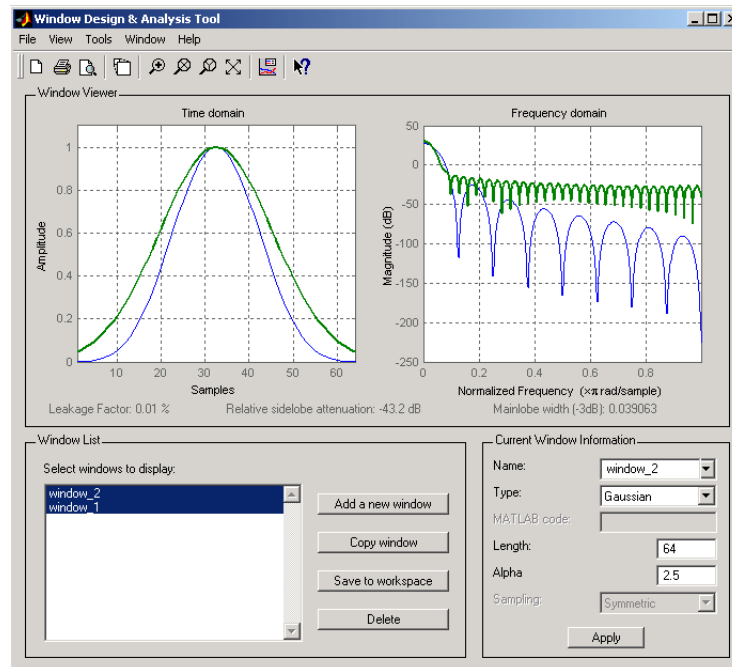
**Purpose** Parzen (de la Valle-Poussin) window

**Syntax** `w = parzenwin(L)`

**Description** `w = parzenwin(L)` returns the L-point Parzen (de la Valle-Poussin) window in column vector `w`. Parzen windows are piecewise cubic approximations of Gaussian windows. Parzen window sidelobes fall off as  $1/\omega^4$ .

**Examples** Compare 64-point Parzen and Gaussian windows and display the result using `sigwin` window objects and `wintool`:

```
wintool(sigwin.parzenwin(64),sigwin.gausswin(64))
```





**Algorithm**

The Parzen window is defined as

$$w(n) = 1.0 - 6\left(\frac{n}{N/2}\right)^2\left(1.0 - \frac{|n|}{N/2}\right) \quad 0 \leq |n| \leq \frac{N}{4}$$

$$w(n) = 2\left(1.0 - \frac{|n|}{N/2}\right)^3 \quad \frac{N}{4} \leq |n| \leq \frac{N}{2}$$

The window length  $L = N+1$ .

**References**

Harris, F.J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66, No. 1 (January 1978).

**See Also**

barthannwin, bartlett, blackmanharris, bohmanwin, nuttallwin, rectwin, triang, window, wintool, wvtool

# pburg

---

**Purpose** PSD using Burg method

**Syntax**

```
Pxx = pburg(x,p)
[Pxx,w] = pburg(x,p)
[Pxx,w] = pburg(x,p,nfft)
[Pxx,f] = pburg(x,p,nfft,fs)
[Pxx,f] = pburg(x,p,nfft,fs,'range')
[Pxx,w] = pburg(x,p,nfft,'range')
pburg(...)
```

**Description** `Pxx = pburg(x,p)` implements the Burg algorithm, a parametric spectral estimation method, and returns `Pxx`, an estimate of the power spectral density (PSD) of the vector `x`. The entries of `x` represent samples of a discrete-time signal, and `p` is the integer specifying the order of an autoregressive (AR) prediction model for the signal, used in estimating the PSD.

The power spectral density is calculated in units of power per radians per sample. Real-valued inputs produce full power one-sided (in frequency) PSDs (by default), while complex-valued inputs produce two-sided PSDs.

In general, the length of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) FFT length is 256. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

**PSD Vector Characteristics for an FFT Length of 256 (Default)**

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	129	$[0, \pi]$
Complex-valued	256	$[0, 2\pi)$

`[Pxx,w] = pburg(x,p)` also returns `w`, a vector of frequencies at which the PSD is estimated. `Pxx` and `w` have the same length. The units for frequency are rad/sample.

$[P_{xx}, w] = \text{pburg}(x, p, nfft)$  uses the Burg method to estimate the PSD while specifying the length of the FFT with the integer  $nfft$ . If you specify  $nfft$  as the empty vector  $[]$ , it takes the default value of 256.

The length of  $P_{xx}$  and the frequency range for  $w$  depend on  $nfft$  and the values of the input  $x$ . The following table indicates the length of  $P_{xx}$  and the frequency range for  $w$  in this syntax.

**PSD and Frequency Vector Characteristics**

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	$(nfft/2 + 1)$	$[0, \pi]$
Real-valued	Odd	$(nfft + 1)/2$	$[0, \pi)$
Complex-valued	Even or odd	$nfft$	$[0, 2\pi)$

$[P_{xx}, f] = \text{pburg}(x, p, nfft, fs)$  uses the sampling frequency  $fs$  specified as an integer in hertz (Hz) to compute the PSD vector ( $P_{xx}$ ) and the corresponding vector of frequencies ( $f$ ). In this case, the units for the frequency vector are in Hz. The spectral density produced is calculated in units of power per Hz. If you specify  $fs$  as the empty vector  $[]$ , the sampling frequency defaults to 1 Hz.

The frequency range for  $f$  depends on  $nfft$ ,  $fs$ , and the values of the input  $x$ . The length of  $P_{xx}$  is the same as in the table above. The following table indicates the frequency range for  $f$  for this syntax.

**PSD and Frequency Vector Characteristics with fs Specified**

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	$[0, fs/2]$
Real-valued	Odd	$[0, fs/2)$
Complex-valued	Even or odd	$[0, fs)$

$[P_{xx}, f] = \text{pburg}(x, p, nfft, fs, 'range')$  or

`[Pxx,w] = pburg(x,p,nfft,'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'range'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range  $[0, f_s)$ . This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `f_s` as the empty vector, `[]`, the frequency range is  $[0, 1)$ .
  - If you don't specify `f_s`, the frequency range is  $[0, 2\pi)$ .
- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`.

---

**Note** You can put the string argument `'range'` anywhere in the input argument list after `p`.

---

`pburg(...)` with no outputs plots the power spectral density in the current figure window. The frequency range on the plot is the same as the range of output `w` (or `f`) for a given set of parameters.

## Remarks

The power spectral density is computed as the distribution of power per unit frequency.

This algorithm depends on your selecting an appropriate model order for your signal.

## Examples

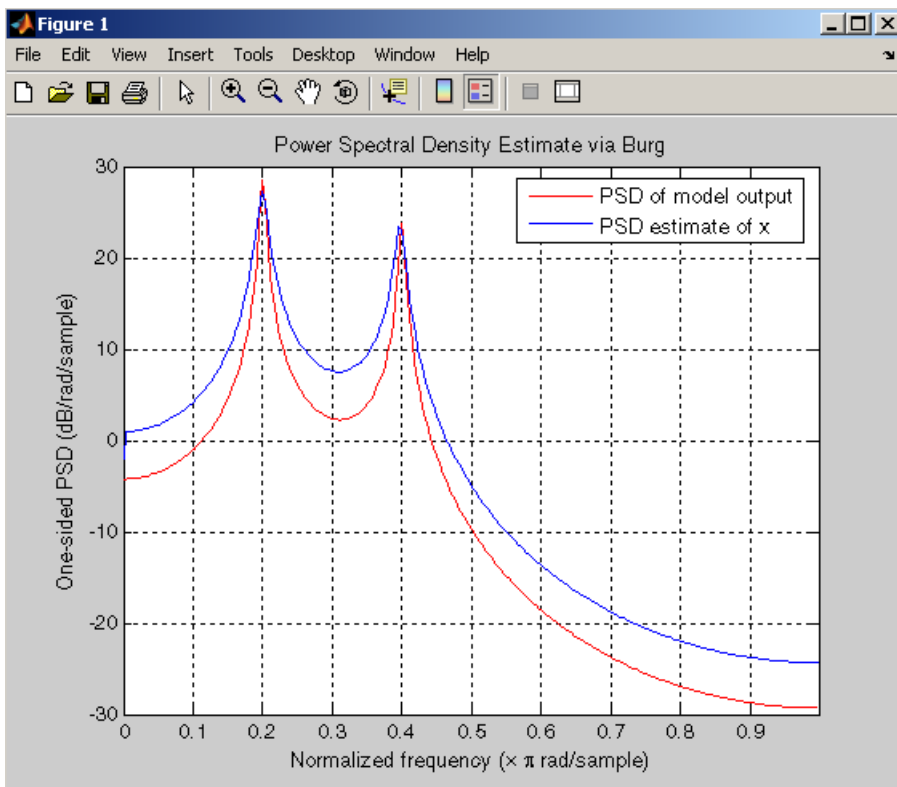
The Burg method estimates the spectral density by fitting an AR prediction model of a given order to the signal, so first generate a signal from an AR (all-pole) model of a given order. Use `freqz` to check the magnitude of the frequency response of your AR filter. Then, generate the input signal `x` by filtering white noise through the AR filter. Estimate the PSD of `x` based on a fourth-order AR prediction model because in this case we know that the original AR system model `a` has order 4:

```
a = [1 -2.2137 2.9403 -2.1697 0.9606]; % AR filter coefficients
[H,w] = freqz(1,a,256); % AR filter freq response
```

```

% Scale to make one-sided PSD
Hp = plot(w/pi,20*log10(2*abs(H)/(2*pi)), 'r');
hold on;
randn('state',1);
x = filter(1,a,randn(256,1)); % AR system output
pburg(x,4,511);
xlabel('Normalized frequency (\times \pi rad/sample)')
ylabel('One-sided PSD (dB/rad/sample)')
legend('PSD of model output','PSD estimate of x')

```



# pburg

---

## Algorithm

Linear prediction filters can be used to model the second-order statistical characteristics of a signal. The prediction filter output can be used to model the signal when the input is white noise.

The Burg method fits an AR linear prediction filter model of the specified order to the input signal by minimizing (using least squares) the arithmetic mean of the forward and backward prediction errors. The spectral density is then computed from the frequency response of the prediction filter. The AR filter parameters are constrained to satisfy the Levinson-Durbin recursion.

## References

- [1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, Chapter 7.
- [2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

## See Also

arburg, lpc, pcov, peig, periodogram, pmcov, pmtm, pmusic, pwelch, pyulear

**Purpose** PSD using covariance method

**Syntax**

```
Pxx = pcov(x,p)
[Pxx,w] = pcov(x,p)
[Pxx,w] = pcov(x,p,nfft)
[Pxx,f] = pcov(x,p,nfft,fs)
[Pxx,f] = pcov(x,p,nfft,fs,'range')
[Pxx,w] = pcov(x,p,nfft,'range')
pcov(...)
```

**Description** `Pxx = pcov(x,p)` implements the covariance algorithm, a parametric spectral estimation method, and returns `Pxx`, an estimate of the power spectral density (PSD) of the vector `x`. The entries of `x` represent samples of a discrete-time signal, and where `p` is the integer specifying the order of an autoregressive (AR) prediction model for the signal, used in estimating the PSD.

The power spectral density is calculated in units of power per radians per sample. Real-valued inputs produce full power one-sided (in frequency) PSDs (by default), while complex-valued inputs produce two-sided PSDs.

In general, the length of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) FFT length is 256. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

#### PSD Vector Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	129	$[0, \pi]$
Complex-valued	256	$[0, 2\pi)$

`[Pxx,w] = pcov(x,p)` also returns `w`, a vector of frequencies at which the PSD is estimated. `Pxx` and `w` have the same length. The units for frequency are rad/sample.

$[P_{xx}, w] = \text{pcov}(x, p, nfft)$  uses the covariance method to estimate the PSD while specifying the length of the FFT with the integer  $nfft$ . If you specify  $nfft$  as the empty vector  $[]$ , it takes the default value of 256.

The length of  $P_{xx}$  and the frequency range for  $w$  depend on  $nfft$  and the values of the input  $x$ . The following table indicates the length of  $P_{xx}$  and the frequency range for  $w$  in this syntax.

**PSD and Frequency Vector Characteristics**

<b>Real/Complex Input Data</b>	<b>nfft Even/Odd</b>	<b>Length of Pxx</b>	<b>Range of w</b>
Real-valued	Even	$(nfft/2 + 1)$	$[0, \pi]$
Real-valued	Odd	$(nfft + 1)/2$	$[0, \pi]$
Complex-valued	Even or odd	$nfft$	$[0, 2\pi)$

$[P_{xx}, f] = \text{pcov}(x, p, nfft, fs)$  uses the sampling frequency  $fs$  specified as an integer in hertz (Hz) to compute the PSD vector ( $P_{xx}$ ) and the corresponding vector of frequencies ( $f$ ). In this case, the units for the frequency vector are in Hz. The spectral density produced is calculated in units of power per Hz. If you specify  $fs$  as the empty vector  $[]$ , the sampling frequency defaults to 1 Hz.

The frequency range for  $f$  depends on  $nfft$ ,  $fs$ , and the values of the input  $x$ . The length of  $P_{xx}$  is the same as in the table above. The following table indicates the frequency range for  $f$  for this syntax.

**PSD and Frequency Vector Characteristics with fs Specified**

<b>Real/Complex Input Data</b>	<b>nfft Even/Odd</b>	<b>Range of f</b>
Real-valued	Even	$[0, fs/2]$
Real-valued	Odd	$[0, fs/2)$
Complex-valued	Even or odd	$[0, fs)$

$[P_{xx}, f] = \text{pcov}(x, p, nfft, fs, 'range')$  or



`[Pxx,w] = pcov(x,p,nfft,'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'range'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range  $[0, f_s)$ . This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `f_s` as the empty vector, `[]`, the frequency range is  $[0, 1)$ .
  - If you don't specify `f_s`, the frequency range is  $[0, 2\pi)$ .
- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`.

---

**Note** You can put the string argument `'range'` anywhere in the input argument list after `p`.

---

`pcov(...)` with no outputs plots the power spectral density in the current figure window. The frequency range on the plot is the same as the range of output `w` (or `f`) for a given set of parameters.

## Remarks

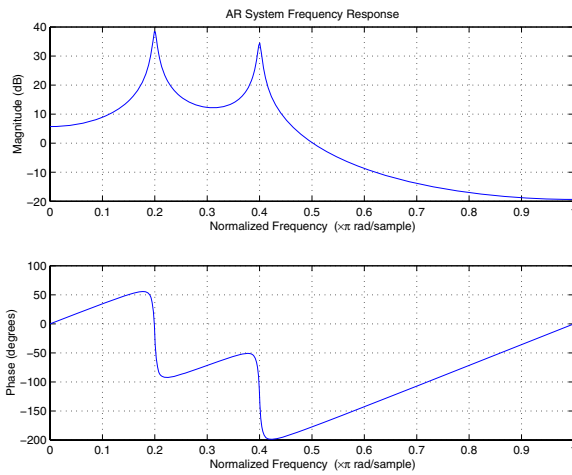
The power spectral density is computed as the distribution of power per unit frequency.

This algorithm depends on your selecting an appropriate model order for your signal.

## Examples

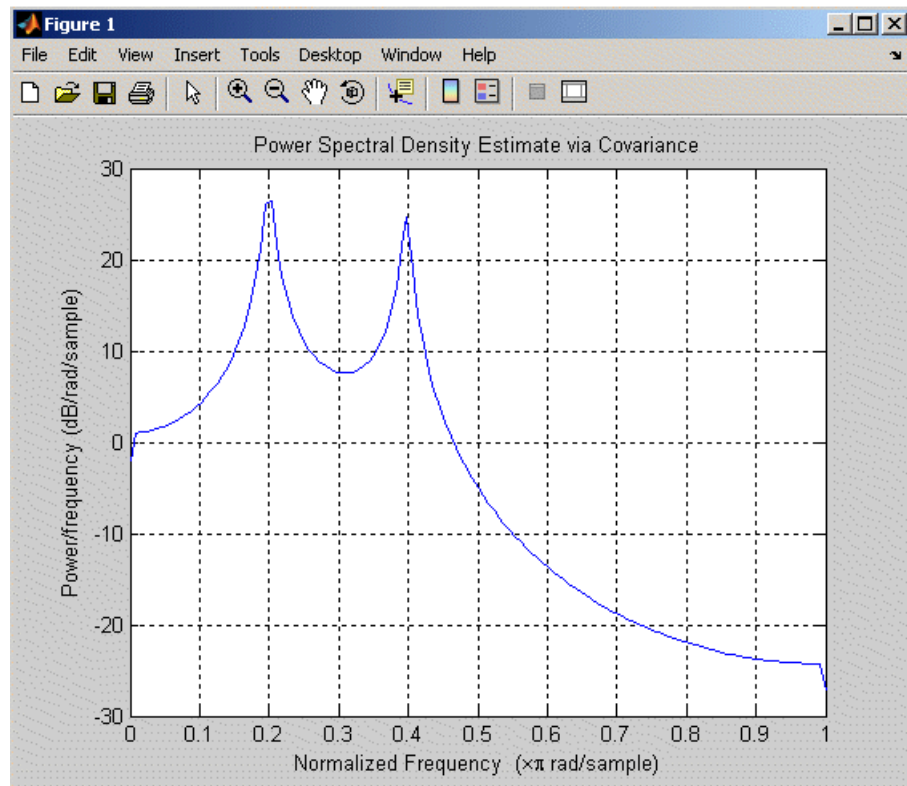
Because the covariance method estimates the spectral density by fitting an AR prediction model of a given order to the signal, first generate a signal from an AR (all-pole) model of a given order. You can use `freqz` to check the magnitude of the frequency response of your AR filter. This will give you an idea of what to expect when you estimate the PSD using `pcov`:

```
a = [1 -2.2137 2.9403 -2.1697 0.9606]; % AR filter coefficients
freqz(1,a) % AR filter frequency response
title('AR System Frequency Response')
```



Now generate the input signal  $x$  by filtering white noise through the AR filter. Estimate the PSD of  $x$  based on a fourth-order AR prediction model since in this case we know that the original AR system model  $a$  has order 4:

```
randn('state',1);  
x = filter(1,a,randn(256,1)); % Signal generated from AR filter  
pcov(x,4)                    % Fourth-order estimate
```



## Algorithm

Linear prediction filters can be used to model the second-order statistical characteristics of a signal. The prediction filter output can be used to model the signal when the input is white noise.

The covariance method estimates the PSD of a signal using the covariance method. The covariance (or nonwindowed) method fits an AR linear prediction filter model to the signal by minimizing the forward prediction error (based on causal observations of your input signal) in the least squares sense. The spectral estimate returned by `pcov` is the squared magnitude of the frequency response of this AR model.

## References

[1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, Chapter 7.

## pcov

---

[2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

### See Also

arcov, lpc, pburg, peig, periodogram, pmcov, pmtm, pmusic, pwelch, pyulear

**Purpose** Pseudospectrum using eigenvector method

**Syntax**

```
[S,w] = peig(x,p)
[S,w] = peig(...,nfft)
[S,f] = peig(x,p,nfft,fs)
[S,f] = peig(...,'corr')
[S,f] = peig(x,p,nfft,fs,nwin,noverlap)
[...] = peig(...,'range')
[... ,v,e] = peig(...)
peig(...)
```

**Description** `[S,w] = peig(x,p)` implements the eigenvector spectral estimation method and returns `S`, the pseudospectrum estimate of the input signal `x`, and `w`, a vector of normalized frequencies (in rad/sample) at which the pseudospectrum is evaluated. The pseudospectrum is calculated using estimates of the eigenvectors of a correlation matrix associated with the input data `x`, where `x` is specified as either:

- A row or column vector representing one observation of the signal
- A rectangular array for which each row of `x` represents a separate observation of the signal (for example, each row is one output of an array of sensors, as in array processing), such that  $x' * x$  is an estimate of the correlation matrix

---

**Note** You can use the output of `corrmtx` to generate such an array `x`.

---

You can specify the second input argument `p` as either:

- A scalar integer. In this case, the signal subspace dimension is `p`.
- A two-element vector. In this case, `p(2)`, the second element of `p`, represents a threshold that is multiplied by  $\lambda_{\min}$ , the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold  $\lambda_{\min} * p(2)$  are assigned to the noise subspace. In this case, `p(1)` specifies the maximum dimension of the signal subspace.

The extra threshold parameter in the second entry in `p` provides you more flexibility and control in assigning the noise and signal subspaces.

`S` and `w` have the same length. In general, the length of the FFT and the values of the input `x` determine the length of the computed `S` and the range of the corresponding normalized frequencies. The following table indicates the length of `S` (and `w`) and the range of the corresponding normalized frequencies for this syntax.

**S Characteristics for an FFT Length of 256 (Default)**

Real/Complex Input Data	Length of <code>S</code> and <code>w</code>	Range of the Corresponding Normalized Frequencies
Real-valued	129	$[0, \pi]$
Complex-valued	256	$[0, 2\pi)$

`[S,w] = peig(...,nfft)` specifies the length of the FFT used to estimate the pseudospectrum with the integer `nfft`. The default value for `nfft` (entered as an empty vector `[]`) is 256.

The following table indicates the length of `S` and `w`, and the frequency range for `w` for this syntax.

**S and Frequency Vector Characteristics**

Real/Complex Input Data	nfft Even/Odd	Length of <code>S</code> and <code>w</code>	Range of <code>w</code>
Real-valued	Even	$(nfft/2 + 1)$	$[0, \pi]$
Real-valued	Odd	$(nfft + 1)/2$	$[0, \pi)$
Complex-valued	Even or odd	<code>nfft</code>	$[0, 2\pi)$

`[S,f] = peig(x,p,nfft,fs)` returns the pseudospectrum in the vector `S` evaluated at the corresponding vector of frequencies `f` (in Hz). You supply the sampling frequency `fs` in Hz. If you specify `fs` with the empty vector `[]`, the sampling frequency defaults to 1 Hz.

The frequency range for  $f$  depends on  $nfft$ ,  $fs$ , and the values of the input  $x$ . The length of  $S$  (and  $f$ ) is the same as in the “S and Frequency Vector Characteristics” table above. The following table indicates the frequency range for  $f$  for this syntax.

### S and Frequency Vector Characteristics with $fs$ Specified

Real/Complex Input Data	nfft Even/Odd	Range of $f$
Real-valued	Even	$[0, fs/2]$
Real-valued	Odd	$[0, fs/2)$
Complex-valued	Even or odd	$[0, fs)$

$[S, f] = \text{peig}(\dots, 'corr')$  forces the input argument  $x$  to be interpreted as a correlation matrix rather than matrix of signal data. For this syntax  $x$  must be a square matrix, and all of its eigenvalues must be nonnegative.

$[S, f] = \text{peig}(x, p, nfft, fs, nwin, noverlap)$  allows you to specify  $nwin$ , a scalar integer indicating a rectangular window length, or a real-valued vector specifying window coefficients. Use the scalar integer  $noverlap$  in conjunction with  $nwin$  to specify the number of input sample points by which successive windows overlap.  $noverlap$  is not used if  $x$  is a matrix. The default value for  $nwin$  is  $2 * p(1)$  and  $noverlap$  is  $nwin - 1$ .

With this syntax, the input data  $x$  is segmented and windowed before the matrix used to estimate the correlation matrix eigenvalues is formulated. The segmentation of the data depends on  $nwin$ ,  $noverlap$ , and the form of  $x$ . Comments on the resulting windowed segments are described in the following table.

### Windowed Data Depending on $x$ and $nwin$

Input data $x$	Form of $nwin$	Windowed Data
Data vector	Scalar	Length is $nwin$
Data vector	Vector of coefficients	Length is $\text{length}(nwin)$

**Windowed Data Depending on x and nwin (Continued)**

Input data x	Form of nwin	Windowed Data
Data matrix	Scalar	Data is not windowed.
Data matrix	Vector of coefficients	length(nwin) must be the same as the column length of x, and noverlap is not used.

See the “Eigenvector Length Depending on Input Data and Syntax” table below for related information on this syntax.

---

**Note** The arguments `nwin` and `nooverlap` are ignored when you include the string `'corr'` in the syntax.

---

`[...] = peig(..., 'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'range'` can be either:

- `'whole'`: Compute the pseudospectrum over the frequency range  $[0, f_s)$ . This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `f_s` as the empty vector, `[]`, the frequency range is  $[0, 1)$ .
  - If you don't specify `f_s`, the frequency range is  $[0, 2\pi)$ .
- `'half'`: Compute the pseudospectrum over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`.

---

**Note** You can put the string arguments `'range'` or `'corr'` anywhere in the input argument list after `p`.

---

`[..., v, e] = peig(...)` returns the matrix `v` of noise eigenvectors, along with the associated eigenvalues in the vector `e`. The columns of `v` span the noise subspace of dimension `size(v, 2)`. The dimension of the signal subspace is `size(v, 1) - size(v, 2)`. For this syntax, `e` is a vector of estimated eigenvalues of the correlation matrix.



`peig(...)` with no output arguments plots the pseudospectrum in the current figure window.

## Remarks

In the process of estimating the pseudospectrum, `peig` computes the noise and signal subspaces from the estimated eigenvectors  $\mathbf{v}_j$  and eigenvalues  $\lambda_j$  of the signal's correlation matrix. The smallest of these eigenvalues is used in conjunction with the threshold parameter  $p(2)$  to affect the dimension of the noise subspace in some cases.

The length  $n$  of the eigenvectors computed by `peig` is the sum of the dimensions of the signal and noise subspaces. This eigenvector length depends on your input (signal data or correlation matrix) and the syntax you use.

The following table summarizes the dependency of the eigenvector length on the input argument.

### Eigenvector Length Depending on Input Data and Syntax

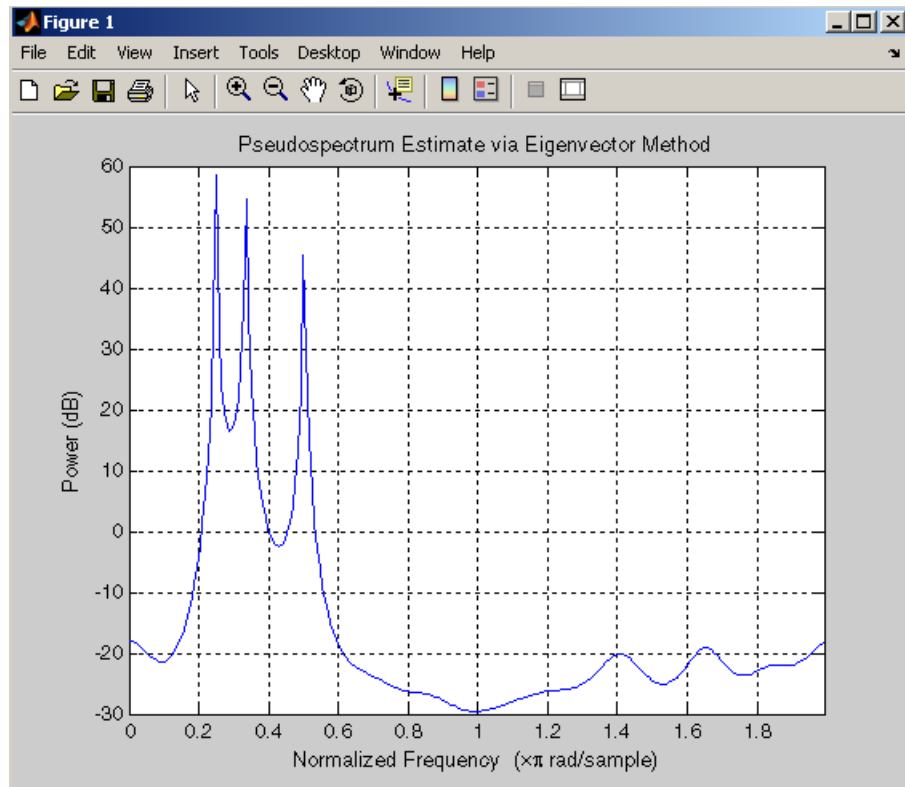
Form of Input Data $\mathbf{x}$	Comments on the Syntax	Length $n$ of Eigenvectors
Row or column vector	<code>nwin</code> is specified as a scalar integer.	<code>nwin</code>
Row or column vector	<code>nwin</code> is specified as a vector.	<code>length(nwin)</code>
Row or column vector	<code>nwin</code> is not specified.	$2 * p(1)$
$l$ -by- $m$ matrix	If <code>nwin</code> is specified as a scalar, it is not used. If <code>nwin</code> is specified as a vector, <code>length(nwin)</code> must equal $m$ .	$m$
$m$ -by- $m$ nonnegative definite matrix	The string ' <code>corr</code> ' is specified and <code>nwin</code> is not used.	$m$

You should specify `nwin > p(1)` or `length(nwin) > p(1)` if you want `p(2) > 1` to have any effect.

## Examples

Implement the eigenvector method to find the pseudospectrum of the sum of three sinusoids in noise, using the default FFT length of 256. Use the modified covariance method for the correlation matrix estimate:

```
randn('state',1); n=0:99;
s=exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);
X=corrmtx(s,12,'mod');
peig(X,3,'whole') % Uses the default NFFT of 256.
```



## Algorithm

The eigenvector method estimates the pseudospectrum from a signal or a correlation matrix using a weighted version of the MUSIC algorithm derived from Schmidt's eigenspace analysis method [1][2]. The algorithm performs eigenspace analysis of the signal's correlation matrix in order to estimate the signal's frequency content. The eigenvalues and eigenvectors of the signal's

correlation matrix are estimated using svd if you don't supply the correlation matrix. This algorithm is particularly suitable for signals that are the sum of sinusoids with additive white Gaussian noise.

The eigenvector method produces a pseudospectrum estimate given by

$$P_{ev}(f) = \frac{1}{\left( \sum_{k=1}^N |\mathbf{v}_k^H \mathbf{e}(f)|^2 \right) / \lambda_k}$$

where  $N$  is the dimension of the eigenvectors and  $\mathbf{v}_k$  is the  $k$ th eigenvector of the correlation matrix of the input signal. The integer  $p$  is the dimension of the signal subspace, so the eigenvectors  $\mathbf{v}_k$  used in the sum correspond to the smallest eigenvalues  $\lambda_k$  of the correlation matrix. The eigenvectors used in the PSD estimate span the noise subspace. The vector  $\mathbf{e}(f)$  consists of complex exponentials, so the inner product

$$\mathbf{v}_k^H \mathbf{e}(f)$$

amounts to a Fourier transform. This is used for computation of the PSD estimate. The FFT is computed for each  $\mathbf{v}_k$  and then the squared magnitudes are summed and scaled.

## References

- [1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, pp. 373-378.
- [2] Schmidt, R.O, "Multiple Emitter Location and Signal Parameter Estimation," *IEEE Trans. Antennas Propagation*, Vol. AP-34 (March 1986), pp. 276-280.
- [3] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

## See Also

corrmtx, dspdata, pburg, periodogram, pmtm, pmusic, prony, pwelch, rooteig, rootmusic, spectrum

# periodogram

---

**Purpose** PSD using a periodogram

**Syntax**

```
[Pxx,w] = periodogram(x)
[Pxx,w] = periodogram(x>window)
[Pxx,w] = periodogram(x>window,nfft)
[Pxx,f] = periodogram(x>window,nfft,fs)
[Pxx,...] = periodogram(x,...,'range')
periodogram(...)
```

**Description** `[Pxx,w] = periodogram(x)` returns the power spectral density (PSD) estimate `Pxx` of the sequence `x` using a periodogram. The power spectral density is calculated in units of power per radians per sample. The corresponding vector of frequencies `w` is computed in radians per sample, and has the same length as `Pxx`.

A real-valued input vector `x` produces a full power one-sided (in frequency) PSD (by default), while a complex-valued `x` produces a two-sided PSD.

In general, the length  $N$  of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) length  $N$  of the FFT is the larger of 256 and the next power of 2 greater than the length of `x`. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

## PSD Vector Characteristics for an FFT Length of $N$ (Default)

Real/Complex Input Data	Length of <code>Pxx</code>	Range of the Corresponding Normalized Frequencies
Real-valued	$(N/2) + 1$	$[0, \pi]$
Complex-valued	$N$	$[0, 2\pi)$

`[Pxx,w] = periodogram(x>window)` returns the PSD estimate `Pxx` computed using the modified periodogram method. The vector `window` specifies the coefficients of the window used in computing a modified periodogram of the input signal. Both input arguments must be vectors of the same length. When you don't supply the second argument `window`, or set it to the empty vector `[]`,

a rectangular window (`rectwin`) is used by default. In this case the standard periodogram is calculated.

`[Pxx,w] = periodogram(x,window,nfft)` uses the modified periodogram to estimate the PSD while specifying the length of the FFT with the integer `nfft`. If you set `nfft` to the empty vector `[]`, it takes the default value for  $N$  listed in the previous syntax.

The length of `Pxx` and the frequency range for `w` depend on `nfft` and the values of the input `x`. The following table indicates the length of `Pxx` and the frequency range for `w` for this syntax.

### PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	$(nfft/2 + 1)$	$[0, \pi]$
Real-valued	Odd	$(nfft + 1)/2$	$[0, \pi]$
Complex-valued	Even or odd	<code>nfft</code>	$[0, 2\pi]$

**Note** `periodogram` uses an `nfft`-point FFT of the windowed data (`x.*window`) to compute the periodogram. If the value you specify for `nfft` is less than the length of `x`, then `x.*window` is wrapped modulo `nfft`. If the value you specify for `nfft` is greater than the length of `x`, then `x.*window` is zero-padded to compute the FFT.

`[Pxx,f] = periodogram(x,window,nfft,fs)` uses the sampling frequency `fs` specified as an integer in hertz (Hz) to compute the PSD vector (`Pxx`) and the corresponding vector of frequencies (`f`). In this case, the units for the frequency vector are in Hz. The spectral density produced is calculated in units of power per Hz. If you specify `fs` as the empty vector `[]`, the sampling frequency defaults to 1 Hz.

# periodogram

---

The frequency range for  $f$  depends on  $nfft$ ,  $fs$ , and the values of the input  $x$ . The length of  $P_{xx}$  is the same as in the table above. The following table indicates the frequency range for  $f$  for this syntax.

## PSD and Frequency Vector Characteristics with $fs$ Specified

Real/Complex Input Data	nfft Even/Odd	Range of $f$
Real-valued	Even	$[0, fs/2]$
Real-valued	Odd	$[0, fs/2)$
Complex-valued	Even or odd	$[0, fs)$

$[P_{xx}, f] = \text{periodogram}(x, \text{window}, nfft, fs, 'range')$  or

$[P_{xx}, w] = \text{periodogram}(x, \text{window}, nfft, 'range')$  specifies the range of frequency values to include in  $f$  or  $w$ . This syntax is useful when  $x$  is real. *'range'* can be either:

- *'twosided'*: Compute the two-sided PSD over the frequency range  $[0, fs)$ . This is the default for determining the frequency range for complex-valued  $x$ .
  - If you specify  $fs$  as the empty vector,  $[]$ , the frequency range is  $[0, 1)$ .
  - If you don't specify  $fs$ , the frequency range is  $[0, 2\pi)$ .
- *'onesided'*: Compute the one-sided PSD over the frequency ranges specified for real  $x$ . This is the default for determining the frequency range for real-valued  $x$ .

---

**Note** You can put the string argument *'range'* anywhere in the input argument list after *window*.

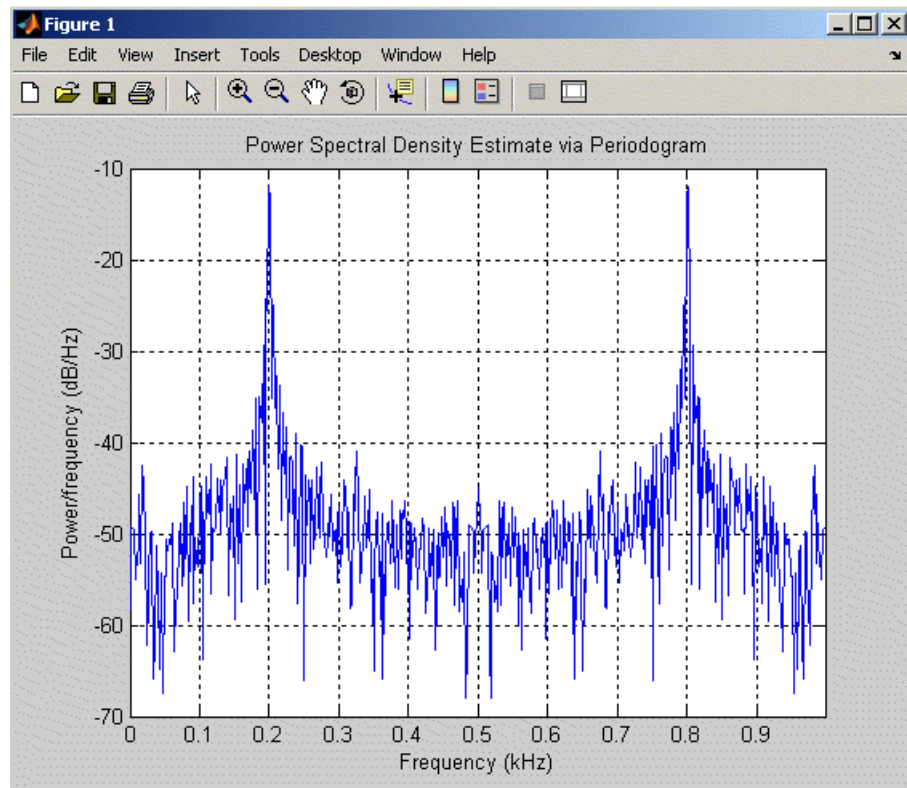
---

$\text{periodogram}(\dots)$  with no outputs plots the power spectral density in dB per unit frequency in the current figure window. The frequency range on the plot is the same as the range of output  $w$  (or  $f$ ) for the syntax you use.

## Examples

Compute the periodogram of a 200 Hz signal embedded in additive noise using the default window:

```
randn('state',0);
Fs = 1000;
t = 0:1/Fs:.3;
x = cos(2*pi*t*200)+0.1*randn(size(t));
periodogram(x,[],'twosided',512,Fs)
```



## Algorithm

The periodogram for a sequence  $[x_1, \dots, x_n]$  is given by the following formula:

$$S(e^{j\omega}) = \frac{1}{n} \left| \sum_{l=1}^n x_l e^{-j\omega l} \right|^2$$

# periodogram

---

This expression forms an estimate of the PSD of the signal defined by the sequence  $[x_1, \dots, x_n]$ .

If you weight your signal sequence by a window  $[w_1, \dots, w_n]$ , then the weighted or modified periodogram is defined as

$$S(e^{j\omega}) = \frac{\frac{1}{n} \left| \sum_{l=1}^n w_l x_l e^{-j\omega l} \right|^2}{\frac{1}{n} \sum_{l=1}^n |w_l|^2}$$

In either case, periodogram uses an  $n$ -point FFT to compute the power spectral density as  $S(e^{j\omega})/F$ , where  $F$  is

- $2\pi$  when you do not supply the sampling frequency
- $fs$  when you supply the sampling frequency

## References

- [1] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997, pp. 24-26.
- [2] Welch, P.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 Electroacoustics*, Vol. AU-15 (June 1967), pp. 70-73.
- [3] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 730-742.

## See Also

`dspdata.msspectrum`, `pburg`, `pcov`, `peig`, `pmcov`, `pmtm`, `pmusic`, `pwelch`, `pyulear`



## Purpose

Phase delay of digital filter

## Syntax

```
[phi,w] = phasedelay(b,a,n)
[phi,w] = phasedelay(b,a,n,'whole')
phi = phasedelay(b,a,w)
[phi,f] = phasedelay(b,a,n,fs)
[phi,f] = phasedelay(b,a,n,'whole',fs)
phi = phasedelay(b,a,f,fs)
[phi,w,s] = phasedelay(...)
[phi,f,s] = phasedelay(...)
phasedelay(b,a,...)
```

## Description

`[phi,w] = phasedelay(b,a,n)` returns the  $n$ -point phase delay response vector `phi` and the  $n$ -point frequency response vector `w` (in radians/sample) of the filter defined by numerator coefficients `b` and denominator coefficients `a`. The phase delay response is evaluated at  $n$  equally spaced points around the upper half of the unit circle. If  $n$  is omitted, it defaults to 512.

`[phi,w] = phasedelay(b,a,n,'whole')` uses  $n$  equally spaced points around the whole unit circle.

`phi = phasedelay(b,a,w)` returns the phase delay response at frequencies specified in vector `w` (in radians/sample). The frequencies are normally between 0 and  $\pi$ .

`[phi,f] = phasedelay(b,a,n,fs)` and `[phi,f] = phasedelay(b,a,n,'whole',fs)` return the phase delay vector `f` (in Hz), using the sampling frequency `fs` (in Hz).

`phi = phasedelay(b,a,f,fs)` returns the phase delay response at the frequencies specified in vector `f` (in Hz), using the sampling frequency `fs` (in Hz).

`[phi,w,s] = phasedelay(...)` and `[phi,f,s] = phasedelay(...)` return plotting information, where `s` is a structure with fields you can change to display different frequency response plots.

`phasedelay(b,a,...)` with no output arguments, plots the phase delay response of the filter.

# phasedelay

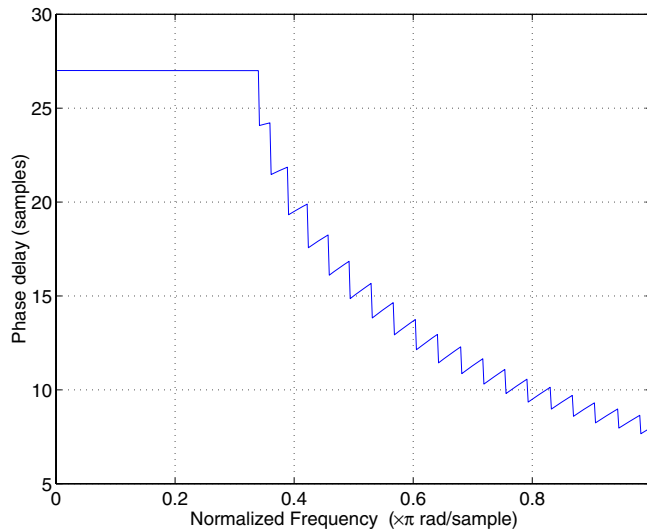
---

## Examples

### Example 1

Plot the phase delay response of a constrained least squares FIR filter:

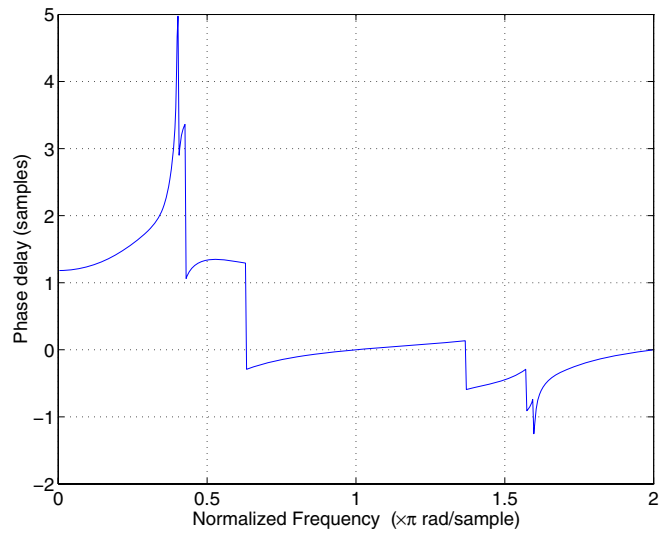
```
b=fircls1(54,.3,.02,.008);  
phasedelay(b)
```



### Example 2

Plot the phase delay response of an elliptic filter:

```
[b,a] = ellip(10,.5,20,.4);  
phasedelay(b,a,512,'whole')
```



## See Also

freqz, fvtool, phasez, grpdelay

# phasez

---

**Purpose** Phase response of digital filter

**Syntax**

```
[phi,w] = phasez(b,a,n)
[phi,w] = phasez(b,a,n,'whole')
phi = phasez(b,a,w)
[phi,f] = phasez(b,a,n,fs)
[phi,f] = phasez(b,a,n,'whole',fs)
phi = phasez(b,a,f,fs)
[phi,w,s] = phasez(...)
[phi,f,s] = phasez(...)
phasez(b,a,...)
phasez(Hd)
```

**Description** `[phi,w] = phasez(b,a,n)` returns the  $n$ -point phase response vector `phi` and the  $n$ -point frequency response vector `w` (in radians/sample) of the filter defined by numerator coefficients `b` and denominator coefficients `a`. The phase response is evaluated at  $n$  equally spaced points around the upper half of the unit circle. If  $n$  is omitted, it defaults to 512.

`[phi,w] = phasez(b,a,n,'whole')` uses  $n$  equally spaced points around the whole unit circle.

`phi = phasez(b,a,w)` returns the phase response at frequencies specified in vector `w` (in radians/sample). The frequencies are normally between 0 and  $\pi$ .

`[phi,f] = phasez(b,a,n,fs)` and `[phi,f] = phasez(b,a,n,'whole',fs)` return the phase vector `f` (in Hz), using the sampling frequency `fs` (in Hz).

`phi = phasez(b,a,f,fs)` returns the phase response at the frequencies specified in vector `f` (in Hz), using the sampling frequency `fs` (in Hz).

`[phi,w,s] = phasez(...)` and `[phi,f,s] = phasez(...)` return plotting information, where `s` is a structure with fields you can change to display different frequency response plots.

`phasez(b,a,...)` with no output arguments, plots the phase response of the filter in the current filter window.

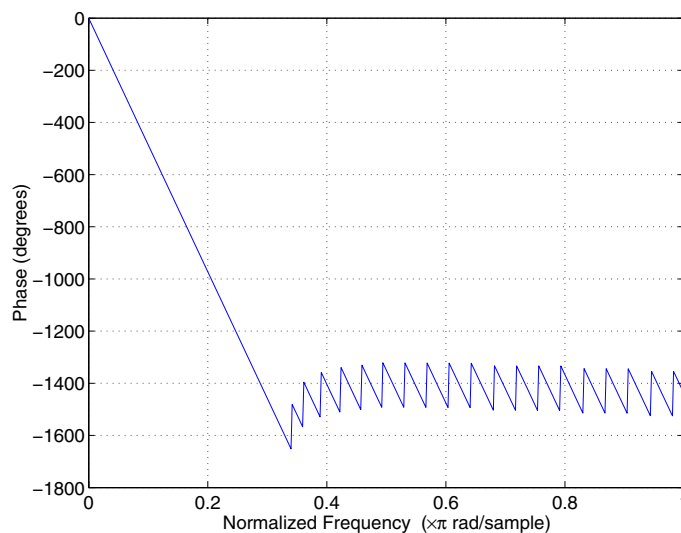
phasez(Hd) plots the phase response of the filter and displays the plot in fvtool. The input Hd is a dfilt filter object.

## Examples

### Example 1

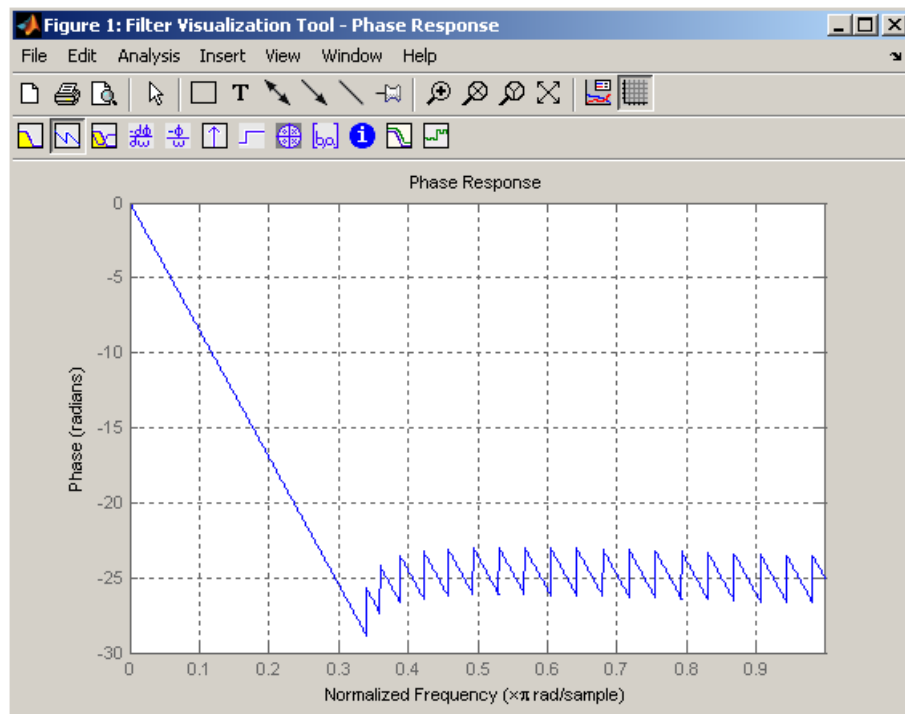
Plot the phase response of a constrained least squares FIR filter:

```
b=fircls1(54,.3,.02,.008);
phasez(b)
```



The same example using a dfilt filter object and displaying the result in fvtool, where you can perform more analyses, is

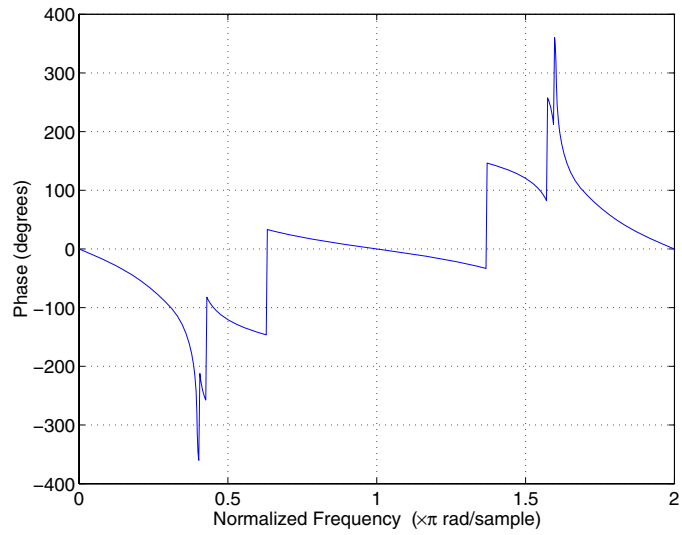
```
b=fircls1(54,.3,.02,.008);
Hd=dfilt.dffir(b);
phasez(Hd)
```



## Example 2

Plot the phase response of an elliptic filter:

```
[b,a] = ellip(10,.5,20,.4);  
phasez(b,a,512,'whole')
```



**See Also**

freqz, fvtool, phasedelay, grpdelay

# pmcov

---

**Purpose** PSD using modified covariance method

**Syntax**

```
Pxx = pmcov(x,p)
[Pxx,w] = pmcov(x,p)
[Pxx,w] = pmcov(x,p,nfft)
[Pxx,f] = pmcov(x,p,nfft,fs)
[Pxx,f] = pmcov(x,p,nfft,fs,'range')
[Pxx,w] = pmcov(x,p,nfft,'range')
pmcov(...)
```

**Description** `Pxx = pmcov(x,p)` implements the modified covariance algorithm, a parametric spectral estimation method, and returns `Pxx`, an estimate of the power spectral density (PSD) of the vector `x`. The entries of `x` represent samples of a discrete-time signal, and `p` is the integer specifying the order of an autoregressive (AR) prediction model for the signal, used in estimating the PSD.

The power spectral density is calculated in units of power per radians per sample. Real-valued inputs produce full power one-sided (in frequency) PSDs (by default), while complex-valued inputs produce two-sided PSDs.

In general, the length of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) FFT length is 256. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

### PSD Vector Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	129	$[0, \pi]$
Complex-valued	256	$[0, 2\pi)$

`[Pxx,w] = pmcov(x,p)` also returns `w`, a vector of frequencies at which the PSD is estimated. `Pxx` and `w` have the same length. The units for frequency are rad/sample.



$[P_{xx}, w] = \text{pmcov}(x, p, nfft)$  uses the covariance method to estimate the PSD while specifying the length of the FFT with the integer  $nfft$ . If you specify  $nfft$  as the empty vector  $[]$ , it takes the default value of 256.

The length of  $P_{xx}$  and the frequency range for  $w$  depend on  $nfft$  and the values of the input  $x$ . The following table indicates the length of  $P_{xx}$  and the frequency range for  $w$  for this syntax.

**PSD and Frequency Vector Characteristics**

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	$(nfft/2 + 1)$	$[0, \pi]$
Real-valued	Odd	$(nfft + 1)/2$	$[0, \pi)$
Complex-valued	Even or odd	$nfft$	$[0, 2\pi)$

$[P_{xx}, f] = \text{pmcov}(x, p, nfft, fs)$  uses the sampling frequency  $fs$  specified as an integer in hertz (Hz) to compute the PSD vector ( $P_{xx}$ ) and the corresponding vector of frequencies ( $f$ ). In this case, the units for the frequency vector are in Hz. The spectral density produced is calculated in units of power per Hz. If you specify  $fs$  as the empty vector  $[]$ , the sampling frequency defaults to 1 Hz.

The frequency range for  $f$  depends on  $nfft$ ,  $fs$ , and the values of the input  $x$ . The length of  $P_{xx}$  is the same as in the “PSD and Frequency Vector Characteristics” table above. The following table indicates the frequency range for  $f$  in this syntax.

**PSD and Frequency Vector Characteristics with fs Specified**

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	$[0, fs/2]$
Real-valued	Odd	$[0, fs/2)$
Complex-valued	Even or odd	$[0, fs)$

$[P_{xx}, f] = \text{pmcov}(x, p, nfft, fs, 'range')$  or

`[Pxx,w] = pmcov(x,p,nfft,'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'range'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range  $[0, f_s)$ . This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `fs` as the empty vector, `[]`, the frequency range is  $[0, 1)$ .
  - If you don't specify `fs`, the frequency range is  $[0, 2\pi)$ .
- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`.

---

**Note** You can put the string argument `'range'` anywhere in the input argument list after `p`.

---

`pmcov(...)` with no outputs plots the power spectral density in the current figure window. The frequency range on the plot is the same as the range of output `w` (or `f`) for a given set of parameters.

## Remarks

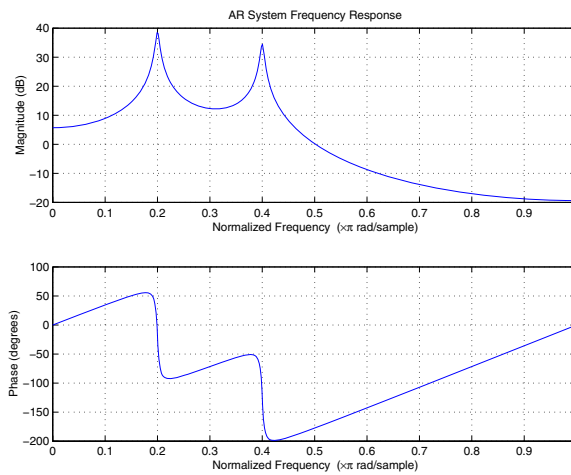
The power spectral density is computed as the distribution of power per unit frequency.

This algorithm depends on your selecting an appropriate model order for your signal.

## Examples

Because the modified covariance method estimates the spectral density by fitting an AR prediction model of a given order to the signal, first generate a signal from an AR (all-pole) model of a given order. You can use `freqz` to check the magnitude of the frequency response of your AR filter. This will give you an idea of what to expect when you estimate the PSD using `pmcov`:

```
a = [1 -2.2137 2.9403 -2.1697 0.9606]; % AR filter coefficients
freqz(1,a) % AR filter frequency response
title('AR System Frequency Response')
```

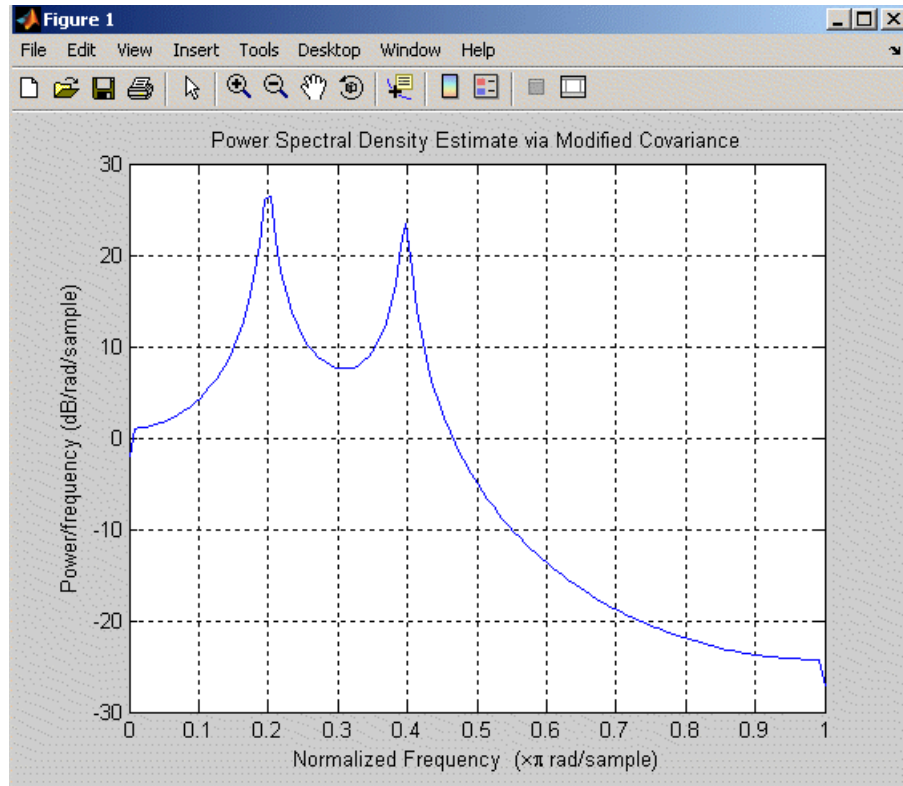


Now generate the input signal  $x$  by filtering white noise through the AR filter. Estimate the PSD of  $x$  based on a fourth-order AR prediction model since in this case we know that the original AR system model  $a$  has order 4:

```

randn('state',1);
x = filter(1,a,randn(256,1));    % AR filter output
pmcov(x,4)                      % Fourth-order estimate

```



## Algorithm

Linear prediction filters can be used to model the second-order statistical characteristics of a signal. The prediction filter output can be used to model the signal when the input is white noise.

pmcov estimates the PSD of the signal vector using the modified covariance method. This method fits an autoregressive (AR) linear prediction filter model to the signal by simultaneously minimizing the forward and backward prediction errors (based on causal observations of your input signal) in the least squares sense. The spectral estimate returned by pmcov is the magnitude squared frequency response of this AR model.

## References

[1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, Chapter 7.

[2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

**See Also**

armcov, lpc, pburg, pcov, peig, periodogram, pmtm, pmusic, pwelch, prony, pyulear

# pmtm

---

**Purpose** PSD using multitaper method (MTM)

**Syntax**

```
[Pxx,w] = pmtm(x,nw)
[Pxx,w] = pmtm(x,nw,nfft)
[Pxx,f] = pmtm(x,nw,nfft,fs)
[Pxx,Pxxc,f] = pmtm(x,nw,nfft,fs)
[Pxx,Pxxc,f] = pmtm(x,nw,nfft,fs,p)
[Pxx,Pxxc,f] = pmtm(x,e,v,nfft,fs,p)
[Pxx,Pxxc,f] = pmtm(x,dpss_params,nfft,fs,p)
[...] = pmtm(...,'method')
[...] = pmtm(...,'range')
pmtm(...)
```

**Description** `pmtm` estimates the power spectral density (PSD) of the time series  $x$  using the multitaper method (MTM) described in [1]. This method uses linear or nonlinear combinations of modified periodograms to estimate the PSD. These periodograms are computed using a sequence of orthogonal tapers (windows in the frequency domain) specified from the discrete prolate spheroidal sequences (see `dpss`).

`[Pxx,w] = pmtm(x,nw)` estimates the PSD  $P_{xx}$  for the input signal  $x$ , using  $2*nw-1$  discrete prolate spheroidal sequences as data tapers for the multitaper estimation method.  $nw$  is the time-bandwidth product for the discrete prolate spheroidal sequences. If you specify  $nw$  as the empty vector `[]`, a default value of 4 is used. Other typical choices are 2, 5/2, 3, or 7/2. `pmtm` also returns  $w$ , a vector of frequencies at which the PSD is estimated.  $P_{xx}$  and  $w$  have the same length. The units for frequency are rad/sample.

The power spectral density is calculated in units of power per radians per sample. Real-valued inputs produce (by default) full power one-sided (in frequency) PSDs, while complex-valued inputs produce two-sided PSDs.

In general, the length  $N$  of the FFT and the values of the input  $x$  determine the length of  $P_{xx}$  and the range of the corresponding normalized frequencies. For this syntax, the (default) length  $N$  of the FFT is the larger of 256 and the next power of 2 greater than the length of the segment. The following table indicates

the length of  $P_{xx}$  and the range of the corresponding normalized frequencies for this syntax.

**PSD Vector Characteristics for an FFT Length of  $N$  (Default)**

<b>Real/Complex Input Data</b>	<b>Length of <math>P_{xx}</math></b>	<b>Range of the Corresponding Normalized Frequencies</b>
Real-valued	$(N/2) + 1$	$[0, \pi]$
Complex-valued	$N$	$[0, 2\pi)$

$[P_{xx}, w] = \text{pmtm}(x, nw, nfft)$  uses the multitaper method to estimate the PSD while specifying the length of the FFT with the integer  $nfft$ . If you specify  $nfft$  as the empty vector  $[]$ , it adopts the default value for  $N$  described in the previous syntax.

The length of  $P_{xx}$  and the frequency range for  $w$  depend on  $nfft$  and the values of the input  $x$ . The following table indicates the length of  $P_{xx}$  and the frequency range for  $w$  for this syntax.

**PSD and Frequency Vector Characteristics**

<b>Real/Complex Input Data</b>	<b><math>nfft</math> Even/Odd</b>	<b>Length of <math>P_{xx}</math></b>	<b>Range of <math>w</math></b>
Real-valued	Even	$(nfft/2 + 1)$	$[0, \pi]$
Real-valued	Odd	$(nfft + 1)/2$	$[0, \pi)$
Complex-valued	Even or odd	$nfft$	$[0, 2\pi)$

$[P_{xx}, f] = \text{pmtm}(x, nw, nfft, fs)$  uses the sampling frequency  $fs$  specified as an integer in hertz (Hz) to compute the PSD vector ( $P_{xx}$ ) and the corresponding vector of frequencies ( $f$ ). In this case, the units for the frequency vector  $f$  are in Hz. The spectral density produced is calculated in units of power per Hz. If you specify  $fs$  as the empty vector  $[]$ , the sampling frequency defaults to 1 Hz.

The frequency range for  $f$  depends on  $nfft$ ,  $fs$ , and the values of the input  $x$ . The length of  $P_{xx}$  is the same as in the Table , PSD and Frequency Vector

Characteristics above. The following table indicates the frequency range for  $f$  for this syntax.

**PSD and Frequency Vector Characteristics with fs Specified**

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	$[0, f_s/2]$
Real-valued	Odd	$[0, f_s/2)$
Complex-valued	Even or odd	$[0, f_s)$

$[P_{xx}, P_{xxc}, f] = \text{pmtm}(x, nw, nfft, fs)$  returns  $P_{xxc}$ , the 95% confidence interval for  $P_{xx}$ . Confidence intervals are computed using a chi-squared approach.  $P_{xxc}$  is a two-column matrix with the same number of rows as  $P_{xx}$ .  $P_{xxc}(:, 1)$  is the lower bound of the confidence interval and  $P_{xxc}(:, 2)$  is the upper bound of the confidence interval.

$[P_{xx}, P_{xxc}, f] = \text{pmtm}(x, nw, nfft, fs, p)$  returns  $P_{xxc}$ , the  $p*100\%$  confidence interval for  $P_{xx}$ , where  $p$  is a scalar between 0 and 1. If you don't specify  $p$ , or if you specify  $p$  as the empty vector  $[]$ , the default 95% confidence interval is used.

$[P_{xx}, P_{xxc}, f] = \text{pmtm}(x, e, v, nfft, fs, p)$  returns the PSD estimate  $P_{xx}$ , the confidence interval  $P_{xxc}$ , and the frequency vector  $f$  from the data tapers contained in the columns of the matrix  $e$ , and their concentrations in the vector  $v$ . The length of  $v$  is the same as the number of columns in  $e$ . You can obtain the data to supply as these arguments from the outputs of `dpss`.

$[P_{xx}, P_{xxc}, f] = \text{pmtm}(x, dpss\_params, nfft, fs, p)$  uses the cell array `dpss_params` containing the input arguments to `dpss` (listed in order, but excluding the first argument) to compute the data tapers. For example, `pmtm(x, {3.5, 'trace'}, 512, 1000)` calculates the prolate spheroidal sequences for  $nw = 3.5$ , using  $nfft = 512$ , and  $fs = 1000$ , and displays the method that `dpss` uses for this calculation. See `dpss` for other options.

$[...] = \text{pmtm}(..., 'method')$  specifies the algorithm used for combining the individual spectral estimates. The string `'method'` can be one of the following:

- `'adapt'`: Thomson's adaptive nonlinear combination (default)



- 'unity': A linear combination of the weighted periodograms with unity weights
- 'eigen': A linear combination of the weighted periodograms with eigenvalue weights

[...] = pmtm(..., 'range') specifies the range of frequency values to include in *f* or *w*. This syntax is useful when *x* is real. 'range' can be either:

- 'twosided': Compute the two-sided PSD over the frequency range  $[0, f_s)$ . This is the default for determining the frequency range for complex-valued *x*.
  - If you specify *f<sub>s</sub>* as the empty vector, [], the frequency range is  $[0, 1)$ .
  - If you don't specify *f<sub>s</sub>*, the frequency range is  $[0, 2\pi)$ .
- 'onesided': Compute the one-sided PSD over the frequency ranges specified for real *x*. This is the default for determining the frequency range for real-valued *x*.

---

**Note** You can put the string arguments 'range' or 'method' anywhere after the input argument *nw* or *v*.

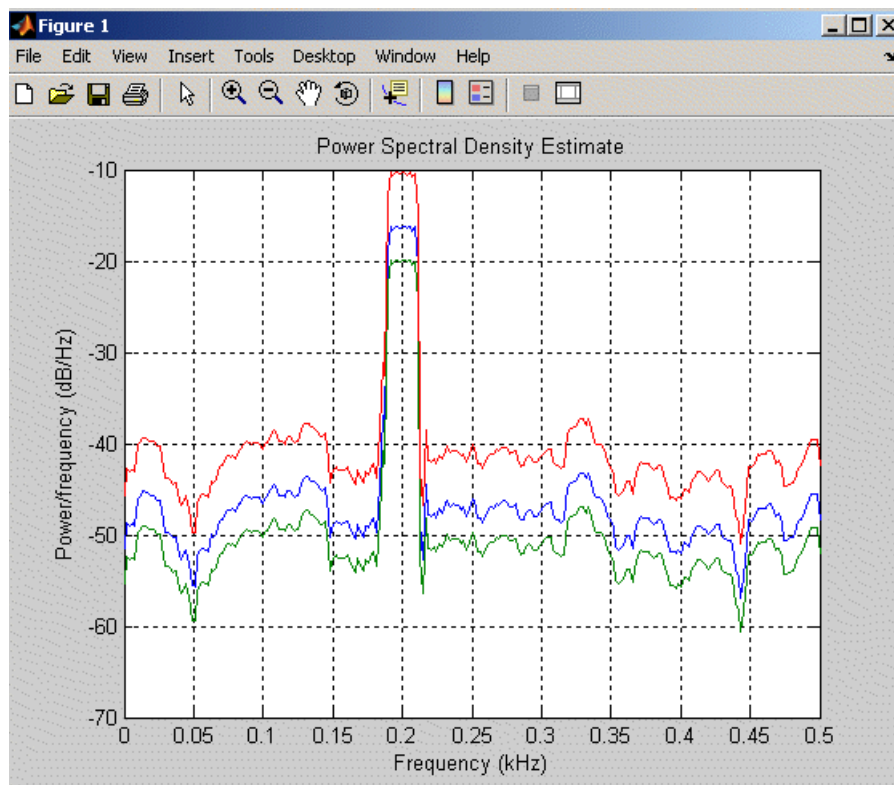
---

pmtm(...) with no output arguments plots the PSD estimate and the confidence intervals in the current figure window. If you don't specify *f<sub>s</sub>*, the 95% confidence interval is plotted. If you do specify *f<sub>s</sub>*, the confidence intervals plotted depend on the value of *p*.

## Examples

This example analyzes a sinusoid in white noise:

```
randn('state',0);  
fs = 1000;  
t = 0:1/fs:0.3;  
x = cos(2*pi*t*200) + 0.1*randn(size(t));  
[Pxx,Pxxc,f] = pmtm(x,3.5,512,fs,0.99);  
hpsd = dspdata.psd([Pxx Pxxc], 'Fs',fs);  
plot(hpsd)
```



## References

[1] Percival, D.B., and A.T. Walden, *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques*, Cambridge University Press, 1993.

[2] Thomson, D.J., "Spectrum estimation and harmonic analysis," *Proceedings of the IEEE*, Vol. 70 (1982), pp. 1055-1096.

**See Also**

dpss, pburg, pcov, peig, periodogram, pmcov, pmusic, pwelch, pyulear

# pmusic

---

**Purpose** Pseudospectrum using MUSIC algorithm

**Syntax**

```
[S,w] = pmusic(x,p)
[S,w] = pmusic(...,nfft)
[S,f] = pmusic(x,p,nfft,fs)
[S,f] = pmusic(...,'corr')
[S,f] = pmusic(x,p,nfft,fs,nwin,noverlap)
[...] = pmusic(...,'range')
[... ,v,e] = pmusic(...)
pmusic(...)
```

**Description** `[S,w] = pmusic(x,p)` implements the MUSIC (Multiple Signal Classification) algorithm and returns `S`, the pseudospectrum estimate of the input signal `x`, and a vector `w` of normalized frequencies (in rad/sample) at which the pseudospectrum is evaluated. The pseudospectrum is calculated using estimates of the eigenvectors of a correlation matrix associated with the input data `x`, where `x` is specified as either:

- A row or column vector representing one observation of the signal
- A rectangular array for which each row of `x` represents a separate observation of the signal (for example, each row is one output of an array of sensors, as in array processing), such that  $x' * x$  is an estimate of the correlation matrix

---

**Note** You can use the output of `corrmtx` to generate such an array `x`.

---

You can specify the second input argument `p` as either:

- A scalar integer. In this case, the signal subspace dimension is `p`.
- A two-element vector. In this case, `p(2)`, the second element of `p`, represents a threshold that is multiplied by  $\lambda_{\min}$ , the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold  $\lambda_{\min} * p(2)$  are assigned to the noise subspace. In this case, `p(1)` specifies the maximum dimension of the signal subspace.

The extra threshold parameter in the second entry in  $p$  provides you more flexibility and control in assigning the noise and signal subspaces.

$S$  and  $w$  have the same length. In general, the length of the FFT and the values of the input  $x$  determine the length of the computed  $S$  and the range of the corresponding normalized frequencies. The following table indicates the length of  $S$  (and  $w$ ) and the range of the corresponding normalized frequencies for this syntax.

### S Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of $S$ and $w$	Range of the Corresponding Normalized Frequencies
Real-valued	129	$[0, \pi]$
Complex-valued	256	$[0, 2\pi)$

$[S,w] = \text{pmusic}(\dots, \text{nfft})$  specifies the length of the FFT used to estimate the pseudospectrum with the integer  $\text{nfft}$ . The default value for  $\text{nfft}$  (entered as an empty vector  $[]$ ) is 256.

The following table indicates the length of  $S$  and  $w$ , and the frequency range for  $w$  in this syntax.

### S and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of $S$ and $w$	Range of $w$
Real-valued	Even	$(\text{nfft}/2 + 1)$	$[0, \pi]$
Real-valued	Odd	$(\text{nfft} + 1)/2$	$[0, \pi)$
Complex-valued	Even or odd	$\text{nfft}$	$[0, 2\pi)$

$[S,f] = \text{pmusic}(x,p,\text{nfft},f_s)$  returns the pseudospectrum in the vector  $S$  evaluated at the corresponding vector of frequencies  $f$  (in Hz). You supply the sampling frequency  $f_s$  in Hz. If you specify  $f_s$  with the empty vector  $[]$ , the sampling frequency defaults to 1 Hz.

The frequency range for  $f$  depends on  $nfft$ ,  $fs$ , and the values of the input  $x$ . The length of  $S$  (and  $f$ ) is the same as in the “S and Frequency Vector Characteristics” table above. The following table indicates the frequency range for  $f$  for this syntax.

**S and Frequency Vector Characteristics with fs Specified**

<b>Real/Complex Input Data</b>	<b>nfft Even/Odd</b>	<b>Range of f</b>
Real-valued	Even	$[0, fs/2]$
Real-valued	Odd	$[0, fs/2)$
Complex-valued	Even or odd	$[0, fs)$

`[S,f] = pmusic(..., 'corr')` forces the input argument  $x$  to be interpreted as a correlation matrix rather than matrix of signal data. For this syntax  $x$  must be a square matrix, and all of its eigenvalues must be nonnegative.

`[S,f] = pmusic(x,p,nfft,fs,nwin,noverlap)` allows you to specify  $nwin$ , a scalar integer indicating a rectangular window length, or a real-valued vector specifying window coefficients. Use the scalar integer  $noverlap$  in conjunction with  $nwin$  to specify the number of input sample points by which successive windows overlap.  $noverlap$  is not used if  $x$  is a matrix. The default value for  $nwin$  is  $2*p(1)$  and  $noverlap$  is  $nwin-1$ .

With this syntax, the input data  $x$  is segmented and windowed before the matrix used to estimate the correlation matrix eigenvalues is formulated. The segmentation of the data depends on  $nwin$ ,  $noverlap$ , and the form of  $x$ . Comments on the resulting windowed segments are described in the following table.

**Windowed Data Depending on x and nwin**

<b>Input data x</b>	<b>Form of nwin</b>	<b>Windowed Data</b>
Data vector	Scalar	Length is $nwin$
Data vector	Vector of coefficients	Length is $length(nwin)$

**Windowed Data Depending on x and nwin (Continued)**

Input data x	Form of nwin	Windowed Data
Data matrix	Scalar	Data is not windowed.
Data matrix	Vector of coefficients	length(nwin) must be the same as the column length of x, and noverlap is not used.

See the “Eigenvector Length Depending on Input Data and Syntax” table below for related information on this syntax.

**Note** The arguments `nwin` and `noverlap` are ignored when you include the string `'corr'` in the syntax.

`[...] = pmusic(..., 'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'range'` can be either:

- `'whole'`: Compute the pseudospectrum over the frequency range  $[0, f_s)$ . This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `fs` as the empty vector, `[]`, the frequency range is  $[0, 1)$ .
  - If you don't specify `fs`, the frequency range is  $[0, 2\pi)$ .
- `'half'`: Compute the pseudospectrum over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`.

**Note** You can put the string arguments `'range'` or `'corr'` anywhere in the input argument list after `p`.

`[...,v,e] = pmusic(...)` returns the matrix `v` of noise eigenvectors, along with the associated eigenvalues in the vector `e`. The columns of `v` span the noise subspace of dimension `size(v,2)`. The dimension of the signal subspace is `size(v,1) - size(v,2)`. For this syntax, `e` is a vector of estimated eigenvalues of the correlation matrix.

`pmusic(...)` with no output arguments plots the pseudospectrum in the current figure window.

## Remarks

In the process of estimating the pseudospectrum, `pmusic` computes the noise and signal subspaces from the estimated eigenvectors  $\mathbf{v}_j$  and eigenvalues  $\lambda_j$  of the signal's correlation matrix. The smallest of these eigenvalues is used in conjunction with the threshold parameter  $p(2)$  to affect the dimension of the noise subspace in some cases.

The length  $n$  of the eigenvectors computed by `pmusic` is the sum of the dimensions of the signal and noise subspaces. This eigenvector length depends on your input (signal data or correlation matrix) and the syntax you use.

The following table summarizes the dependency of the eigenvector length on the input argument.

**Eigenvector Length Depending on Input Data and Syntax**

Form of Input Data $\mathbf{x}$	Comments on the Syntax	Length $n$ of Eigenvectors
Row or column vector	$nwin$ is specified as a scalar integer.	$nwin$
Row or column vector	$nwin$ is specified as a vector.	$\text{length}(nwin)$
Row or column vector	$nwin$ is not specified.	$2 * p(1)$
$l$ -by- $m$ matrix	If $nwin$ is specified as a scalar, it is not used. If $nwin$ is specified as a vector, $\text{length}(nwin)$ must equal $m$ .	$m$
$m$ -by- $m$ nonnegative definite matrix	The string ' <b>corr</b> ' is specified and $nwin$ is not used.	$m$

You should specify  $nwin > p(1)$  or  $\text{length}(nwin) > p(1)$  if you want  $p(2) > 1$  to have any effect.

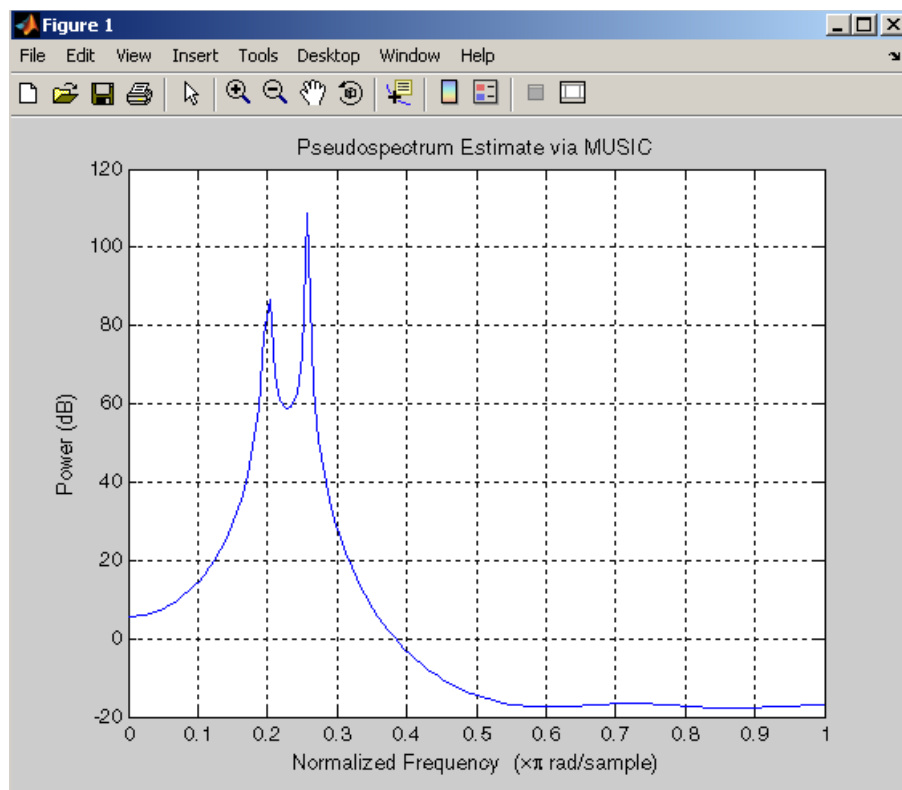


## Examples

### Example 1: pmusic with no Sampling Specified

This example analyzes a signal vector  $x$ , assuming that two real sinusoidal components are present in the signal subspace. In this case, the dimension of the signal subspace is 4 because each real sinusoid is the sum of two complex exponentials:

```
randn('state',0);
n = 0:199;
x = cos(0.257*pi*n) + sin(0.2*pi*n) + 0.01*randn(size(n));
pmusic(x,4)
```



## Example 2: Specifying Sampling Frequency and Subspace Dimensions

This example analyzes the same signal vector  $x$  with an eigenvalue cutoff of 10% above the minimum. Setting  $p(1) = \text{Inf}$  forces the signal/noise subspace decision to be based on the threshold parameter  $p(2)$ . Specify the eigenvectors of length 7 using the `nwin` argument, and set the sampling frequency `fs` to 8 kHz:

```
randn('state',0);
n = 0:199;
x = cos(0.257*pi*n) + sin(0.2*pi*n) + 0.01*randn(size(n));
[P,f] = pmusic(x,[Inf,1.1],[],8000,7); % Window length = 7
```

## Example 3: Entering a Correlation Matrix

Supply a positive definite correlation matrix  $R$  for estimating the spectral density. Use the default 256 samples:

```
R = toeplitz(cos(0.1*pi*[0:6])) + 0.1*eye(7);
[P,f] = pmusic(R,4,'corr');
```

## Example 4: Entering a Signal Data Matrix Generated from `corrmtx`

Enter a signal data matrix  $X_m$  generated from data using `corrmtx`:

```
randn('state',0);
n = 0:699;
x = cos(0.257*pi*(n)) + 0.1*randn(size(n));
Xm = corrmtx(x,7,'mod');
[P,w] = pmusic(Xm,2);
```

## Example 5: Using Windowing to Create the Effect of a Signal Data Matrix

Use the same signal, but let `pmusic` form the 100-by-7 data matrix using its windowing input arguments. In addition, specify an FFT of length 512:

```
randn('state',0);
n = 0:699;
x = cos(0.257*pi*(n)) + 0.1*randn(size(n));
[PP,ff] = pmusic(x,2,512,[],7,0);
```

## Algorithm

The name MUSIC is an acronym for MULTIPLE SIGNAL CLASSIFICATION. The MUSIC algorithm estimates the pseudospectrum from a signal or a correlation matrix using Schmidt's eigenspace analysis method [1]. The algorithm

performs eigenspace analysis of the signal's correlation matrix in order to estimate the signal's frequency content. This algorithm is particularly suitable for signals that are the sum of sinusoids with additive white Gaussian noise. The eigenvalues and eigenvectors of the signal's correlation matrix are estimated if you don't supply the correlation matrix.

The MUSIC pseudospectrum estimate is given by

$$P_{music}(f) = \frac{1}{\mathbf{e}^H(f) \left( \sum_{k=p+1}^N \mathbf{v}_k \mathbf{v}_k^H \right) \mathbf{e}(f)} = \frac{1}{\sum_{k=p+1}^N |\mathbf{v}_k^H \mathbf{e}(f)|^2}$$

where  $N$  is the dimension of the eigenvectors and  $\mathbf{v}_k$  is the  $k$ -th eigenvector of the correlation matrix. The integer  $p$  is the dimension of the signal subspace, so the eigenvectors  $\mathbf{v}_k$  used in the sum correspond to the smallest eigenvalues and also span the noise subspace. The vector  $\mathbf{e}(f)$  consists of complex exponentials, so the inner product

$$\mathbf{v}_k^H \mathbf{e}(f)$$

amounts to a Fourier transform. This is used for computation of the pseudospectrum estimate. The FFT is computed for each  $\mathbf{v}_k$  and then the squared magnitudes are summed.

## References

- [1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, pp. 373-378.
- [2] Schmidt, R.O, "Multiple Emitter Location and Signal Parameter Estimation," *IEEE Trans. Antennas Propagation*, Vol. AP-34 (March 1986), pp. 276-280.
- [3] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1997.

## See Also

corrmtx, dspdata, pburg, peig, periodogram, pmtm, prony, pwelch, rooteig, rootmusic, spectrum

# poly2ac

---

**Purpose** Convert prediction filter polynomial to autocorrelation sequence

**Syntax** `r = poly2ac(a,efinal)`

**Description** `r = poly2ac(a,efinal)` finds the autocorrelation vector `r` corresponding to the prediction filter polynomial `a`. The autocorrelation sequence produced is approximately the same as that of the output of the autoregressive prediction filter whose coefficients are determined by `a`. `poly2ac` also produces the final `length(r)` step prediction error `efinal`. If `a(1)` is not equal to 1, `poly2ac` normalizes the prediction filter polynomial by `a(1)`. `a(1)` cannot be 0.

**Remarks** You can apply this function to both real and complex polynomials.

**Examples**

```
a = [1.0000    0.6147    0.9898    0.0004    0.0034   -0.0077];
efinal = 0.2;
r = poly2ac(a,efinal)

r =
    5.5917
   -1.7277
   -4.4231
    4.3985
    1.6426
   -5.3126
```

**References** [1] Kay, S.M. *Modern Spectral Estimation*, Englewood Cliffs, NJ, Prentice-Hall, 1988.

**See Also** `ac2poly`, `poly2rc`, `rc2ac`

---

<b>Purpose</b>	Convert prediction filter coefficients to line spectral frequencies
<b>Syntax</b>	<code>lsf = poly2lsf(a)</code>
<b>Description</b>	<code>lsf = poly2lsf(a)</code> returns a vector <code>lsf</code> of line spectral frequencies from a vector <code>a</code> of prediction filter coefficients.
<b>Examples</b>	<pre>a = [1.0000    0.6149    0.9899    0.0000    0.0031   -0.0082]; lsf = poly2lsf(a)  lsf =     0.7842     1.5605     1.8776     1.8984     2.3593</pre>
<b>References</b>	<p>[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, “<i>Discrete-Time Processing of Speech Signals</i>,” Prentice-Hall, 1993.</p> <p>[2] Rabiner, L.R., and R.W. Schafer, “<i>Digital Processing of Speech Signals</i>,” Prentice-Hall, 1978.</p>
<b>See Also</b>	<code>lsf2poly</code>

# poly2rc

---

**Purpose** Convert prediction filter polynomial to reflection coefficients

**Syntax** `k = poly2rc(a)`  
`[k,r0] = poly2rc(a,efinal)`

**Description** `k = poly2rc(a)` converts the prediction filter polynomial `a` to the reflection coefficients of the corresponding lattice structure. `a` can be real or complex, and `a(1)` cannot be 0. If `a(1)` is not equal to 1, `poly2rc` normalizes the prediction filter polynomial by `a(1)`. `k` is a row vector of size `length(a) - 1`.

`[k,r0] = poly2rc(a,efinal)` returns the zero-lag autocorrelation, `r0`, based on the final prediction error, `efinal`.

A simple, fast way to check if `a` has all of its roots inside the unit circle is to check if each of the elements of `k` has magnitude less than 1.

```
stable = all(abs(poly2rc(a))<1)
```

**Examples**

```
a = [1.0000 0.6149 0.9899 0.0000 0.0031 -0.0082];  
efinal = 0.2;  
  
[k,r0] = poly2rc(a,efinal)
```

```
k =  
    0.3090  
    0.9801  
    0.0031  
    0.0081  
   -0.0082
```

```
r0 =  
    5.6032
```

**Limitations** If `abs(k(i)) == 1` for any `i`, finding the reflection coefficients is an ill-conditioned problem. `poly2rc` returns some NaNs and provide a warning message in this case.

**Algorithm** `poly2rc` implements this recursive relationship:

$$k(n) = a_n(n)$$

$$a_{n-1}(m) = \frac{a_n(m) - k(n)a_n(n-m)}{1 - k(n)^2}, \quad m = 1, 2, \dots, n-1$$

This relationship is based on Levinson's recursion [1]. To implement it, poly2rc loops through a in reverse order after discarding its first element. For each loop iteration i, the function:

- 1 Sets k(i) equal to a(i)
- 2 Applies the second relationship above to elements 1 through i of the vector a.

$$a = (a \ k(i) * \text{fliplr}(a)) / (1 - k(i)^2);$$

## References

[1] Kay, S.M. *Modern Spectral Estimation*, Englewood Cliffs, NJ, Prentice-Hall, 1988.

## See Also

ac2rc, latc2tf, latcfilt, poly2ac, rc2poly, tf2latc

# polyscale

---

**Purpose** Scale roots of polynomial

**Syntax** `b = polyscale(a,alpha)`

**Description** `b = polyscale(a,alpha)` scales the roots of a polynomial in the  $z$ -plane, where `a` is a vector containing the polynomial coefficients and `alpha` is the scaling factor.

If `alpha` is a real value in the range  $[0 \ 1]$ , then the roots of `a` are radially scaled toward the origin in the  $z$ -plane. Complex values for `alpha` allow arbitrary changes to the root locations.

**Remark** By reducing the radius of the roots in an autoregressive polynomial, the bandwidth of the spectral peaks in the frequency response is expanded (flattened). This operation is often referred to as *bandwidth expansion*.

**See Also** `polystab`, `roots`



**Purpose** Stabilize polynomial

**Syntax** `b = polystab(a)`

**Description** `polystab` stabilizes a polynomial with respect to the unit circle; it reflects roots with magnitudes greater than 1 inside the unit circle.

`b = polystab(a)` returns a row vector `b` containing the stabilized polynomial, where `a` is a vector of polynomial coefficients, normally in the  $z$ -domain.

$$a(z) = a(1) + a(2)z^{-1} + \dots + a(m+1)z^{-m}$$

**Examples** `polystab` can convert a linear-phase filter into a minimum-phase filter with the same magnitude response:

```
h = fir1(25,0.4);
hmin = polystab(h) * norm(h)/norm(polystab(h));
```

**Algorithm** `polystab` finds the roots of the polynomial and maps those roots found outside the unit circle to the inside of the unit circle:

```
v = roots(a);
vs = 0.5*(sign(abs(v)-1)+1);
v = (1-vs).*v + vs./conj(v);
b = a(1)*poly(v);
```

**See Also** `roots`

# prony

---

**Purpose** Prony's method for time domain IIR filter design

**Syntax** `[b,a] = prony(h,n,m)`

**Description** Prony's method is an algorithm for finding an IIR filter with a prescribed time domain impulse response. It has applications in filter design, exponential signal modeling, and system identification (parametric modeling).

`[b,a] = prony(h,n,m)` finds a filter with numerator order  $n$ , denominator order  $m$ , and the time domain impulse response in  $h$ . If the length of  $h$  is less than the largest order ( $n$  or  $m$ ),  $h$  is padded with zeros. `prony` returns the filter coefficients in row vectors  $b$  and  $a$ , of length  $n + 1$  and  $m + 1$ , respectively. The filter coefficients are in descending powers of  $z$ .

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \dots + a(m+1)z^{-m}}$$

## Examples

Recover the coefficients of a Butterworth filter from its impulse response:

```
[b,a] = butter(4,0.2)
b =
    0.0048    0.0193    0.0289    0.0193    0.0048
a =
    1.0000   -2.3695    2.3140   -1.0547    0.1874
h = filter(b,a,[1 zeros(1,25)]);
[bb,aa] = prony(h,4,4)
bb =
    0.0048    0.0193    0.0289    0.0193    0.0048
ab =
    1.0000   -2.3695    2.3140   -1.0547    0.1874
```

**Algorithm**

prony implements the method described in reference [1]. 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 impulse response of the output filter matches exactly the first  $n + 1$  samples of  $x$ . The filter is not necessarily stable, but potentially can recover the coefficients exactly if the data sequence is truly an autoregressive moving-average (ARMA) process of the correct order.

**References**

[1] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, 1987, pp. 226-228.

**See Also**

butter, cheby1, cheby2, ellip, invfreqz, levinson, lpc, stmcb

# pulstran

---

## Purpose

Pulse train

## Syntax

```
y = pulstran(t,d,'func')
y = pulstran(t,d,'func',p1,p2,...)
y = pulstran(t,d,p,fs)
y = pulstran(t,d,p)
```

## Description

`pulstran` generates pulse trains from continuous functions or sampled prototype pulses.

`y = pulstran(t,d,'func')` generates a pulse train based on samples of a continuous function, `'func'`, where `'func'` is

- `'gauspuls'`, for generating a Gaussian-modulated sinusoidal pulse
- `'rectpuls'`, for generating a sampled aperiodic rectangle
- `'tripuls'`, for generating a sampled aperiodic triangle

`pulstran` is evaluated `length(d)` times and returns the sum of the evaluations  $y = \text{func}(t-d(1)) + \text{func}(t-d(2)) + \dots$

The function is evaluated over the range of argument values specified in array `t`, after removing a scalar argument offset taken from the vector `d`. Note that `func` must be a vectorized function that can take an array `t` as an argument.

An optional gain factor may be applied to each delayed evaluation by specifying `d` as a two-column matrix, with the offset defined in column 1 and associated gain in column 2 of `d`. Note that a row vector will be interpreted as specifying delays only.

`pulstran(t,d,'func',p1,p2,...)` allows additional parameters to be passed to `'func'` as necessary. For example:

$$\text{func}(t-d(1),p1,p2,...) + \text{func}(t-d(2),p1,p2,...) + \dots$$

`pulstran(t,d,p,fs)` generates a pulse train that is the sum of multiple delayed interpolations of the prototype pulse in vector `p`, sampled at the rate `fs`, where `p` spans the time interval  $[0, (\text{length}(p) - 1) / \text{fs}]$ , and its samples are identically 0 outside this interval. By default, linear interpolation is used for generating delays.

`pulstran(t,d,p)` assumes that the sampling rate `fs` is equal to 1 Hz.

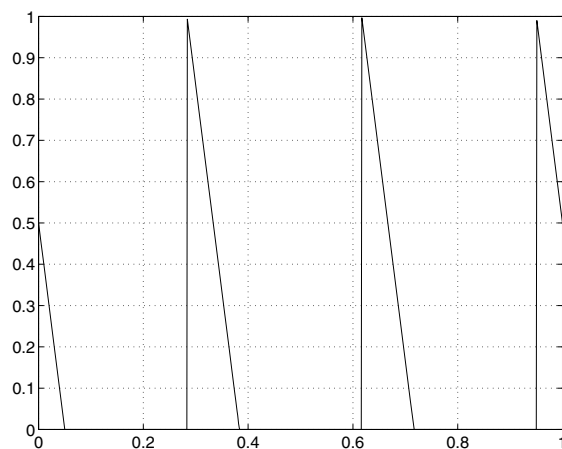
`pulstran(...,'func')` specifies alternative interpolation methods. See `interp1` for a list of available methods.

## Examples

### Example 1

This example generates an asymmetric sawtooth waveform with a repetition frequency of 3 Hz and a sawtooth width of 0.1s. It has a signal length of 1s and a 1 kHz sample rate:

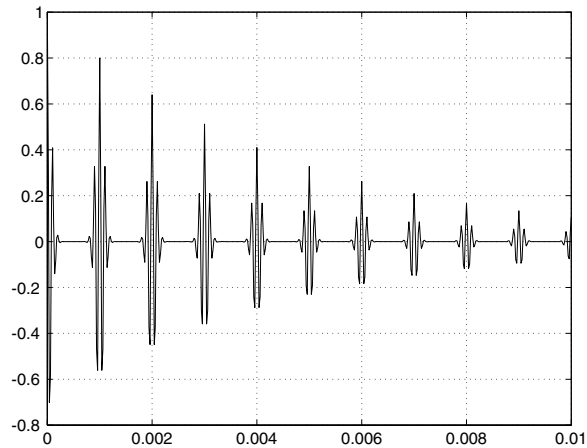
```
t = 0 : 1/1e3 : 1;          % 1 kHz sample freq for 1 sec
d = 0 : 1/3 : 1;           % 3 Hz repetition freq
y = pulstran(t,d,'tripuls',0.1,-1);
plot(t,y)
```



## Example 2

This example generates a periodic Gaussian pulse signal at 10 kHz, with 50% bandwidth. The pulse repetition frequency is 1 kHz, sample rate is 50 kHz, and pulse train length is 10 msec. The repetition amplitude should attenuate by 0.8 each time:

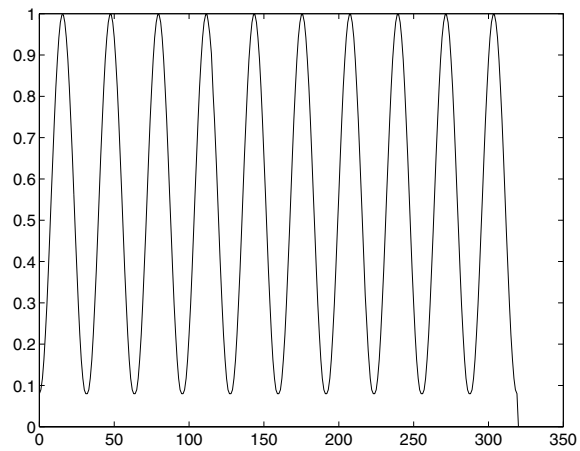
```
t = 0 : 1/50E3 : 10e-3;  
d = [0 : 1/1E3 : 10e-3 ; 0.8.^(0:10)]';  
y = pulstran(t,d,'gauspuls',10e3,0.5);  
plot(t,y)
```



### Example 3

This example generates a train of 10 Hamming windows:

```
p = hamming(32);  
t = 0:320; d = (0:9)'*32;  
y = pulstran(t,d,p);  
plot(t,y)
```



### See Also

chirp, cos, diric, gauspuls, rectpuls, sawtooth, sin, sinc, square, tripuls

# pwelch

---

**Purpose** PSD using Welch's method

**Syntax**

```
[Pxx,w] = pwelch(x)
[Pxx,w] = pwelch(x>window)
[Pxx,w] = pwelch(x>window,noverlap)
[Pxx,w] = pwelch(x>window,noverlap,nfft)
[Pxx,f] = pwelch(x>window,noverlap,nfft,fs)
[...] = pwelch(x>window,noverlap,...,'range')
pwelch(...)
```

**Description** `[Pxx,w] = pwelch(x)` estimates the power spectral density  $P_{xx}$  of the input signal vector  $x$  using Welch's averaged modified periodogram method of spectral estimation. With this syntax:

- The vector  $x$  is segmented into eight sections of equal length, each with 50% overlap.
- Any remaining (trailing) entries in  $x$  that cannot be included in the eight segments of equal length are discarded.
- Each segment is windowed with a Hamming window (see `hamming`) that is the same length as the segment.

The power spectral density is calculated in units of power per radians per sample. The corresponding vector of frequencies  $w$  is computed in radians per sample, and has the same length as  $P_{xx}$ .

A real-valued input vector  $x$  produces a full power one-sided (in frequency) PSD (by default), while a complex-valued  $x$  produces a two-sided PSD.

In general, the length  $N$  of the FFT and the values of the input  $x$  determine the length of  $P_{xx}$  and the range of the corresponding normalized frequencies. For this syntax, the (default) length  $N$  of the FFT is the larger of 256 and the next power of 2 greater than the length of the segment. The following table indicates



the length of  $P_{xx}$  and the range of the corresponding normalized frequencies for this syntax.

**PSD Vector Characteristics for an FFT Length of  $N$  (Default)**

<b>Real/Complex Input Data</b>	<b>Length of <math>P_{xx}</math></b>	<b>Range of the Corresponding Normalized Frequencies</b>
Real-valued	$(N/2) + 1$	$[0, \pi]$
Complex-valued	$N$	$[0, 2\pi)$

$[P_{xx}, w] = \text{pwelch}(x, \text{window})$  calculates the modified periodogram using either:

- The window length `window` for the Hamming window when `window` is a positive integer
- The window weights specified in `window` when `window` is a vector

With this syntax, the input vector  $x$  is divided into an integer number of segments with 50% overlap, and each segment is the same length as the window. Entries in  $x$  that are left over after it is divided into segments are discarded. If you specify `window` as the empty vector `[]`, then the signal data is divided into eight segments, and a Hamming window is used on each one.

$[P_{xx}, w] = \text{pwelch}(x, \text{window}, \text{noverlap})$  divides  $x$  into segments according to `window`, and uses the integer `noverlap` to specify the number of signal samples (elements of  $x$ ) that are common to two adjacent segments. `noverlap` must be less than the length of the window you specify. If you specify `noverlap` as the empty vector `[]`, then `pwelch` determines the segments of  $x$  so that there is 50% overlap (default).

$[P_{xx}, w] = \text{pwelch}(x, \text{window}, \text{noverlap}, \text{nfft})$  uses Welch's method to estimate the PSD while specifying the length of the FFT with the integer `nfft`. If you set `nfft` to the empty vector `[]`, it adopts the default value for  $N$  listed in the previous syntax.

The length of  $P_{xx}$  and the frequency range for  $w$  depend on  $nfft$  and the values of the input  $x$ . The following table indicates the length of  $P_{xx}$  and the frequency range for  $w$  for this syntax.

### PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of $P_{xx}$	Range of $w$
Real-valued	Even	$(nfft/2 + 1)$	$[0, \pi]$
Real-valued	Odd	$(nfft + 1)/2$	$[0, \pi)$
Complex-valued	Even or odd	$nfft$	$[0, 2\pi)$

$[P_{xx}, f] = \text{pwelch}(x, \text{window}, \text{noverlap}, nfft, fs)$  uses the sampling frequency  $fs$  specified in hertz (Hz) to compute the PSD vector ( $P_{xx}$ ) and the corresponding vector of frequencies ( $f$ ). In this case, the units for the frequency vector are in Hz. The spectral density produced is calculated in units of power per Hz. If you specify  $fs$  as the empty vector  $[]$ , the sampling frequency defaults to 1 Hz.

The frequency range for  $f$  depends on  $nfft$ ,  $fs$ , and the values of the input  $x$ . The length of  $P_{xx}$  is the same as in the “PSD and Frequency Vector Characteristics” table above. The following table indicates the frequency range for  $f$  for this syntax.

### PSD and Frequency Vector Characteristics with $fs$ Specified

Real/Complex Input Data	nfft Even/Odd	Range of $f$
Real-valued	Even	$[0, fs/2]$
Real-valued	Odd	$[0, fs/2)$
Complex-valued	Even or odd	$[0, fs)$

$[...] = \text{pwelch}(x, \text{window}, \text{noverlap}, \dots, 'range')$  specifies the range of frequency values. This syntax is useful when  $x$  is real. The string *'range'* can be either:

- 'twosided': Compute the two-sided PSD over the frequency range  $[0, f_s)$ . This is the default for determining the frequency range for complex-valued  $x$ .
  - If you specify  $f_s$  as the empty vector,  $[]$ , the frequency range is  $[0, 1)$ .
  - If you don't specify  $f_s$ , the frequency range is  $[0, 2\pi)$ .
- 'onesided': Compute the one-sided PSD over the frequency ranges specified for real  $x$ . This is the default for determining the frequency range for real-valued  $x$ .

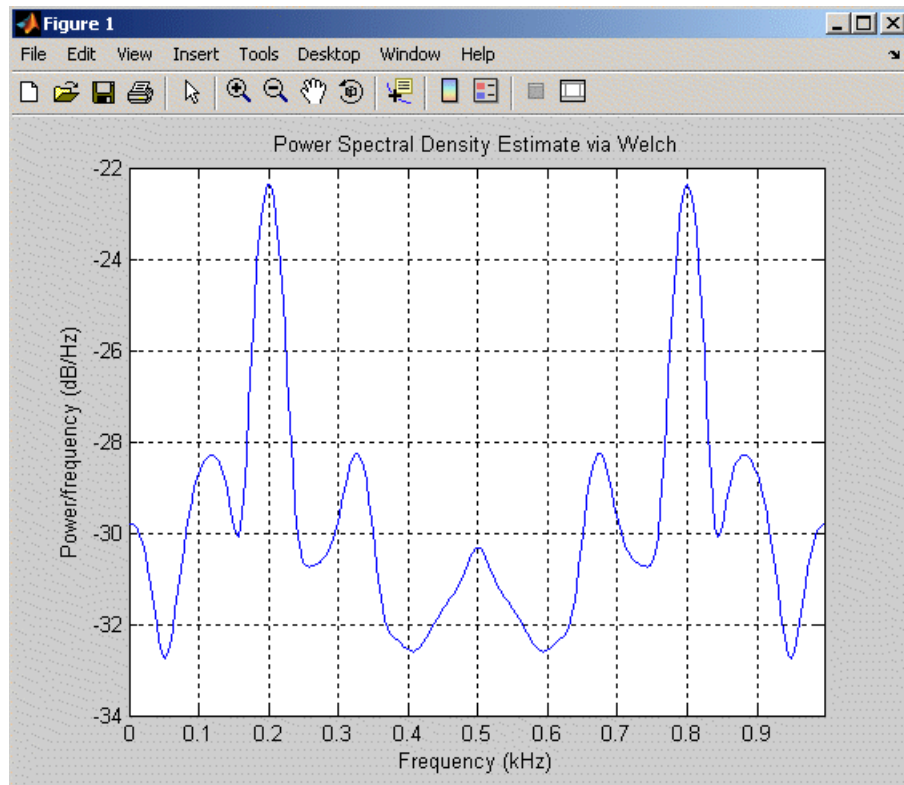
The string *'range'* can appear anywhere in the syntax after `noverlap`.

`pwelch(x, ...)` with no output arguments plots the PSD estimate in dB per unit frequency in the current figure window.

## Examples

Estimate the PSD of a signal composed of a sinusoid plus noise, sampled at 1000 Hz. Use 33-sample windows with 32-sample overlap, and the default FFT length, and display the two-sided PSD estimate:

```
randn('state',0);  
Fs = 1000;    t = 0:1/Fs:.3;  
x = cos(2*pi*t*200) + randn(size(t)); % 200Hz cosine plus noise  
pwelch(x,33,32,[],Fs,'twosided')
```



## Algorithm

pwelch calculates the power spectral density using Welch's method (see references):

- 1 The input signal vector  $x$  is divided into  $k$  overlapping segments according to window and `noverlap` (or their default values).
- 2 The specified (or default) window is applied to each segment of  $x$ .
- 3 An `nfft`-point FFT is applied to the windowed data.
- 4 The (modified) periodogram of each windowed segment is computed.
- 5 The set of modified periodograms is averaged to form the spectrum estimate  $S(e^{j\omega})$ .
- 6 The resulting spectrum estimate is scaled to compute the power spectral density as  $S(e^{j\omega})/F$ , where  $F$  is

- $2\pi$  when you do not supply the sampling frequency
- fs when you supply the sampling frequency

The number of segments  $k$  that  $x$  is divided into is calculated as:

- Eight if you don't specify window, or if you specify it as the empty vector []
  - $k = \frac{m-o}{l-o}$  if you specify window as a nonempty vector or a scalar
- In this equation,  $m$  is the length of the signal vector  $x$ ,  $o$  is the number of overlapping samples (`noverlap`), and  $l$  is the length of each segment (the window length).

## References

- [1] Hayes, M., *Statistical Digital Signal Processing and Modeling*, John Wiley & Sons, 1996.
- [2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1997, pp. 52-54.
- [3] Welch, P.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 Electroacoustics*, Vol. AU-15 (June 1967), pp. 70-73.

## See Also

`dspdata.msspectrum`, `pburg`, `pcov`, `peig`, `periodogram`, `pmcov`, `pmtm`, `pmusic`, `pyulear`

**Purpose** PSD using Yule-Walker AR method

**Syntax**

```
Pxx = pyulear(x,p)
[Pxx,w] = pyulear(x,p,nfft)
[Pxx,f] = pyulear(x,p,nfft,fs)
[Pxx,f] = pyulear(x,p,nfft,fs,'range')
[Pxx,w] = pyulear(x,p,nfft,'range')
pyulear(...)
```

**Description** `Pxx = pyulear(x,p)` implements the Yule-Walker algorithm, a parametric spectral estimation method, and returns `Pxx`, an estimate of the power spectral density (PSD) of the vector `x`. The entries of `x` represent samples of a discrete-time signal. `p` is the integer specifying the order of an autoregressive (AR) prediction model for the signal, used in estimating the PSD. This estimate is also an estimate of the maximum entropy.

The power spectral density is calculated in units of power per radians per sample. Real-valued inputs produce full power one-sided (in frequency) PSDs (by default), while complex-valued inputs produce two-sided PSDs.

In general, the length of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) FFT length is 256. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

**PSD Vector Characteristics for an FFT Length of 256 (Default)**

<b>Real/Complex Input Data</b>	<b>Length of Pxx</b>	<b>Range of the Corresponding Normalized Frequencies</b>
Real-valued	129	$[0, \pi]$
Complex-valued	256	$[0, 2\pi)$

`[Pxx,w] = pyulear(x,p)` also returns `w`, a vector of frequencies at which the PSD is estimated. `Pxx` and `w` have the same length. The units for frequency are rad/sample.

$[P_{xx}, w] = \text{pyulear}(x, p, nfft)$  uses the Yule-walker method to estimate the PSD while specifying the length of the FFT with the integer  $nfft$ . If you specify  $nfft$  as the empty vector  $[]$ , it adopts the default value of 256.

The length of  $P_{xx}$  and the frequency range for  $w$  depend on  $nfft$  and the values of the input  $x$ . The following table indicates the length of  $P_{xx}$  and the frequency range for  $w$  for this syntax.

**PSD and Frequency Vector Characteristics**

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	$(nfft/2 + 1)$	$[0, \pi]$
Real-valued	Odd	$(nfft + 1)/2$	$[0, \pi)$
Complex-valued	Even or odd	$nfft$	$[0, 2\pi)$

$[P_{xx}, f] = \text{pyulear}(x, p, nfft, fs)$  uses the sampling frequency  $fs$  specified as an integer in hertz (Hz) to compute the PSD vector ( $P_{xx}$ ) and the corresponding vector of frequencies ( $f$ ). In this case, the units for the frequency vector are in Hz. The spectral density produced is calculated in units of power per Hz. If you specify  $fs$  as the empty vector  $[]$ , the sampling frequency defaults to 1 Hz.

The frequency range for  $f$  depends on  $nfft$ ,  $fs$ , and the values of the input  $x$ . The length of  $P_{xx}$  is the same as in the “PSD and Frequency Vector Characteristics” table above. The following table indicates the frequency range for  $f$  for this syntax.

**PSD and Frequency Vector Characteristics with fs Specified**

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	$[0, fs/2]$
Real-valued	Odd	$[0, fs/2)$
Complex-valued	Even or odd	$[0, fs)$

$[P_{xx}, f] = \text{pyulear}(x, p, nfft, fs, 'range')$  or

`[Pxx,w] = pyulear(x,p,nfft,'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'range'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range  $[0, f_s)$ . This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `f_s` as the empty vector, `[]`, the frequency range is  $[0, 1)$ .
  - If you don't specify `f_s`, the frequency range is  $[0, 2\pi)$ .
- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`.

---

**Note** You can put the string argument `'range'` anywhere in the input argument list after `p`.

---

`pyulear(...)` plots the power spectral density in the current figure window. The frequency range on the plot is the same as the range of output `w` (or `f`) for a given set of parameters.

## Remarks

The power spectral density is computed as the distribution of power per unit frequency.

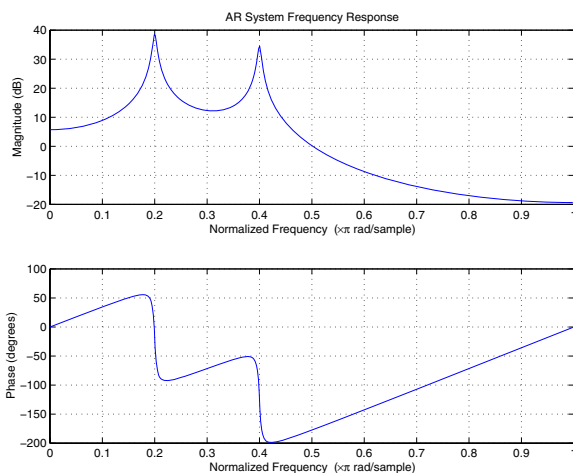
This algorithm depends on your selecting an appropriate model order for your signal.

## Examples

Because the Yule-walker method estimates the spectral density by fitting an AR prediction model of a given order to the signal, first generate a signal from an AR (all-pole) model of a given order. You can use `freqz` to check the magnitude of the frequency response of your AR filter. This will give you an idea of what to expect when you estimate the PSD using `pyulear`:

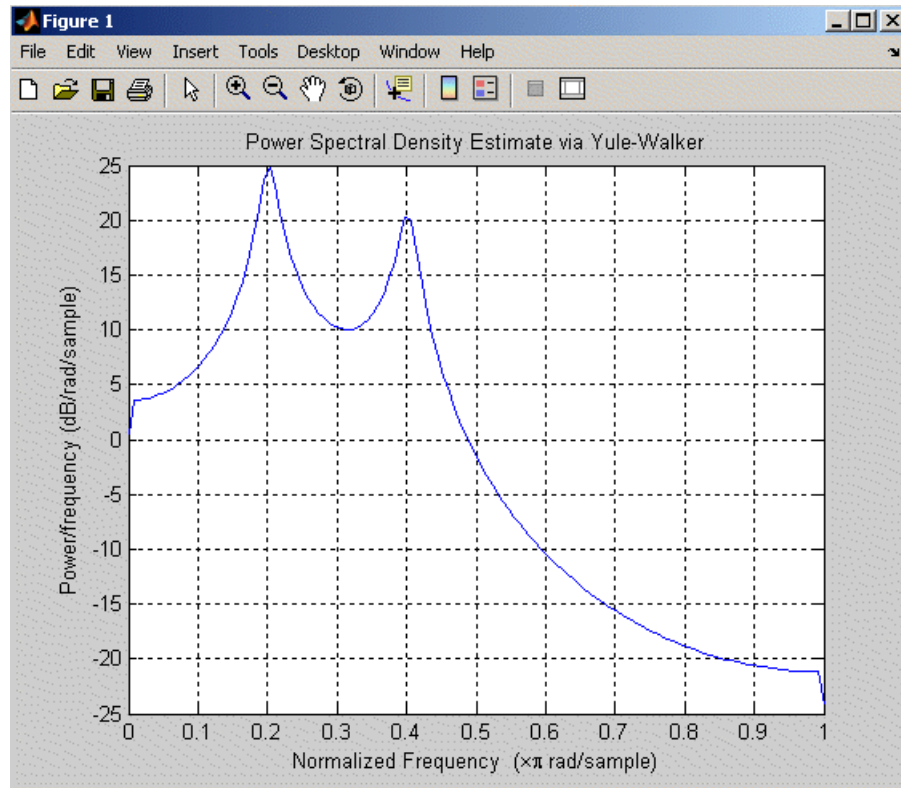
```
a = [1 -2.2137 2.9403 -2.1697 0.9606]; % AR filter coefficients
freqz(1,a) % AR filter frequency response
title('AR System Frequency Response')
```





Now generate the input signal  $x$  by filtering white noise through the AR filter. Estimate the PSD of  $x$  based on a fourth-order AR prediction model, since in this case, we know that the original AR system model  $a$  has order 4:

```
randn('state',1);  
x = filter(1,a,randn(256,1));    % AR system output  
pyulear(x,4)                    % Fourth-order estimate
```



## Algorithm

Linear prediction filters can be used to model the second-order statistical characteristics of a signal. The prediction filter output can be used to model the signal when the input is white noise.

pyulear estimates the PSD of an input signal vector using the Yule-Walker AR method. This method, also called the autocorrelation or windowed method, fits an autoregressive (AR) linear prediction filter model to the signal by minimizing the forward prediction error (based on all observations of the input sequence) in the least squares sense. This formulation leads to the Yule-Walker equations, which are solved by the Levinson-Durbin recursion. The spectral estimate returned by pyulear is the squared magnitude of the frequency response of this AR model.

**References**

- [1] Marple, S.L., *Digital Spectral Analysis*, Prentice-Hall, 1987, Chapter 7.
- [2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

**See Also**

aryule, lpc, pburg, pcov, peig, periodogram, pmcov, pmtm, pmusic, prony, pwelch

# rc2ac

---

**Purpose** Convert reflection coefficients to autocorrelation sequence

**Syntax** `r = rc2ac(k,r0)`

**Description** `r = rc2ac(k,r0)` finds the autocorrelation coefficients, `r`, of the output of the discrete-time prediction error filter from the lattice-form reflection coefficients `k` and initial zero-lag autocorrelation `r0`.

**Examples**

```
k = [0.3090    0.9800    0.0031    0.0082   -0.0082];
r0 = 0.1;
a = rc2ac(k,r0)

a =
    0.1000
   -0.0309
   -0.0791
    0.0787
    0.0294
   -0.0950
```

**References** [1] Kay, S.M., *Modern Spectral Estimation*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

**See Also** `ac2rc`, `poly2ac`, `rc2poly`

---

<b>Purpose</b>	Convert reflection coefficients to inverse sine parameters
<b>Syntax</b>	<code>isin = rc2is(k)</code>
<b>Description</b>	<code>isin = is2rc(k)</code> returns a vector of inverse sine parameters <code>isin</code> from a vector of reflection coefficients <code>k</code> .
<b>Examples</b>	<pre>k = [0.3090 0.9801 0.0031 0.0082 -0.0082]; isin = rc2is(k)  isin =      0.2000    0.8728    0.0020    0.0052   -0.0052</pre>
<b>References</b>	[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, <i>Discrete-Time Processing of Speech Signals</i> , Prentice-Hall, 1993.
<b>See Also</b>	<code>is2rc</code>

# rc2lar

---

**Purpose** Convert reflection coefficients to log area ratio parameters

**Syntax** `g = rc2lar(k)`

**Description** `g = rc2lar(k)` returns a vector of log area ratio parameters `g` from a vector of reflection coefficients `k`.

**Examples**

```
k = [0.3090 0.9801 0.0031 0.0082 -0.0082];  
g = rc2lar(k)  
  
g =  
    0.6389    4.6002    0.0062    0.0164   -0.0164
```

**References** [1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, "*Discrete-Time Processing of Speech Signals*," Prentice-Hall, 1993.

**See Also** `lar2rc`

- Purpose** Convert reflection coefficients to prediction filter polynomial
- Syntax** `a = rc2poly(k)`  
`[a,efinal] = rc2poly(k,r0)`
- Description** `a = rc2poly(k)` converts the reflection coefficients `k` corresponding to the lattice structure to the prediction filter polynomial `a`, with  $a(1) = 1$ . The output `a` is row vector of length  $\text{length}(k)+1$ .
- `[a,efinal] = rc2poly(k,r0)` returns the final prediction error `efinal` based on the zero-lag autocorrelation, `r0`.
- Examples** Consider a lattice IIR filter given by reflection coefficients `k`:
- ```
k = [0.3090    0.9800    0.0031    0.0082   -0.0082];
```
- Its equivalent prediction filter representation is given by
- ```
a = rc2poly(k)
```
- ```
a =
```
- ```
1.0000    0.6148    0.9899    0.0000    0.0032   -0.0082
```
- Algorithm** `rc2poly` computes output `a` using Levinson's recursion [1]. The function:
- 1** Sets the output vector `a` to the first element of `k`
  - 2** Loops through the remaining elements of `k`  
For each loop iteration `i`, `a = [a + a(i-1:-1:1)*k(i) k(i)]`.
  - 3** Implements `a = [1 a]`
- References** [1] Kay, S.M., *Modern Spectral Estimation*, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- See Also** `ac2poly`, `latc2tf`, `latcfilt`, `poly2rc`, `rc2ac`, `rc2is`, `rc2lar`, `tf2latc`

# rceps

---

**Purpose** Real cepstrum and minimum phase reconstruction

**Syntax**  
`y = rceps(x)`  
`[y,ym] = rceps(x)`

**Description** The *real cepstrum* is the inverse Fourier transform of the real logarithm of the magnitude of the Fourier transform of a sequence.

---

**Note** rceps only works on real data.

---

`rceps(x)` returns the real cepstrum of the real sequence `x`. The real cepstrum is a real-valued function.

`[y,ym] = rceps(x)` returns both the real cepstrum `y` and a minimum phase reconstructed version `ym` of the input sequence.

**Algorithm** rceps is an M-file implementation of algorithm 7.2 in [2], that is,

```
y = real(ifft(log(abs(fft(x)))));
```

Appropriate windowing in the cepstral domain forms the reconstructed minimum phase signal:

```
w = [1; 2*ones(n/2-1,1); ones(1 - rem(n,2),1); zeros(n/2-1,1)];  
ym = real(ifft(exp(fft(w.*y))));
```

**References** [1] Oppenheim, A.V., and R.W. Schaffer, *Digital Signal Processing*, Englewood Cliffs, NJ, Prentice-Hall, 1975.

[2] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979.

**See Also** cceps, fft, hilbert, icceps, unwrap



- Purpose** Sampled aperiodic rectangle
- Syntax**  
`y = rectpuls(t)`  
`y = rectpuls(t,w)`
- Description**  
`y = rectpuls(t)` returns a continuous, aperiodic, unity-height rectangular pulse at the sample times indicated in array `t`, centered about `t = 0` and with a default width of 1. Note that the interval of non-zero amplitude is defined to be open on the right, that is, `rectpuls(-0.5) = 1` while `rectpuls(0.5) = 0`.
- `y = rectpuls(t,w)` generates a rectangle of width `w`.
- `rectpuls` is typically used in conjunction with the pulse train generating function `pulstran`.
- See Also** `chirp`, `cos`, `diric`, `gauspuls`, `pulstran`, `sawtooth`, `sin`, `sinc`, `square`, `tripuls`

# rectwin

---

**Purpose** Rectangular window

**Syntax** `w = rectwin(L)`

**Description** `w = rectwin(L)` returns a rectangular window of length `L` in the column vector `w`. This function is provided for completeness; a rectangular window is equivalent to no window at all.

**Algorithm** `w = ones(L,1);`

**References** [1] Oppenheim, A.V., and R.W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

**See Also** `barthannwin`, `bartlett`, `blackmanharris`, `bohmanwin`, `nuttallwin`, `parzenwin`, `triang`, `window`, `wintool`, `wvtool`

<b>Purpose</b>	Change sampling rate by rational factor
<b>Syntax</b>	<pre> y = resample(x,p,q) y = resample(x,p,q,n) y = resample(x,p,q,n,beta) y = resample(x,p,q,b) [y,b] = resample(x,p,q) </pre>
<b>Description</b>	<p><code>y = resample(x,p,q)</code> resamples the sequence in vector <code>x</code> at <math>p/q</math> times the original sampling rate, using a polyphase filter implementation. <code>p</code> and <code>q</code> must be positive integers. The length of <code>y</code> is equal to <math>\text{ceil}(\text{length}(x) * p/q)</math>. If <code>x</code> is a matrix, <code>resample</code> works down the columns of <code>x</code>.</p> <p><code>resample</code> applies an anti-aliasing (lowpass) FIR filter to <code>x</code> during the resampling process. It designs the filter using <code>fir1s</code> with a Kaiser window.</p> <p><code>y = resample(x,p,q,n)</code> uses <code>n</code> terms on either side of the current sample, <code>x(k)</code>, to perform the resampling. The length of the FIR filter <code>resample</code> uses is proportional to <code>n</code>; larger values of <code>n</code> provide better accuracy at the expense of more computation time. The default for <code>n</code> is 10. If you let <code>n = 0</code>, <code>resample</code> performs a nearest-neighbor interpolation</p> $y(k) = x(\text{round}((k-1)*q/p)+1)$ <p>where <math>y(k) = 0</math> if the index to <code>x</code> is greater than <math>\text{length}(x)</math>.</p> <p><code>y = resample(x,p,q,n,beta)</code> uses <code>beta</code> as the design parameter for the Kaiser window that <code>resample</code> employs in designing the lowpass filter. The default for <code>beta</code> is 5.</p> <p><code>y = resample(x,p,q,b)</code> filters <code>x</code> using the vector of filter coefficients <code>b</code>.</p> <p><code>[y,b] = resample(x,p,q)</code> returns the vector <code>b</code>, which contains the coefficients of the filter applied to <code>x</code> during the resampling process.</p>

**Examples** Resample a simple linear sequence at  $3/2$  the original rate:

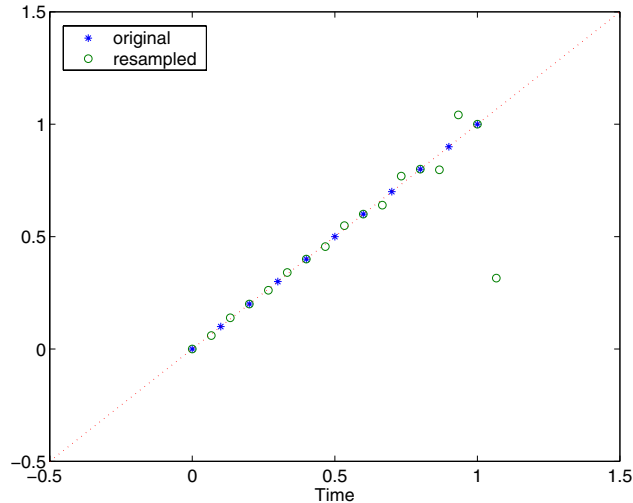
```

fs1 = 10;           % Original sampling frequency in Hz
t1 = 0:1/fs1:1;    % Time vector
x = t1;            % Define a linear sequence
y = resample(x,3,2); % Now resample it

```

# resample

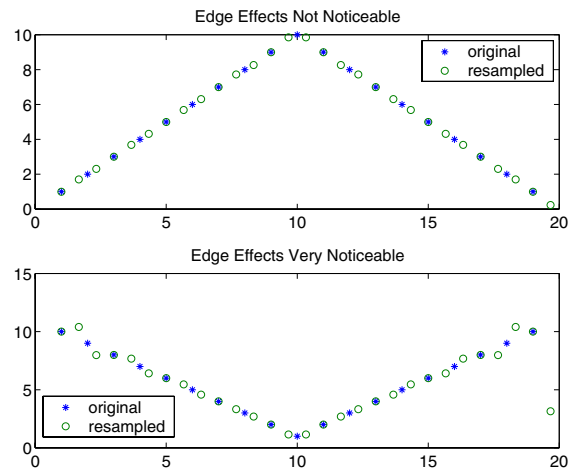
```
t2 = (0:(length(y)-1))*2/(3*fs1); % New time vector
plot(t1,x,'*',t2,y,'o',-0.5:0.01:1.5,-0.5:0.01:1.5,':')
legend('original','resampled'); xlabel('Time')
```



Notice that the last few points of the output  $y$  are inaccurate. In its filtering process, `resample` assumes the samples at times before and after the given samples in  $x$  are equal to zero. Thus large deviations from zero at the end points of the sequence  $x$  can cause inaccuracies in  $y$  at its end points. The following two plots illustrate this side effect of `resample`:

```
x = [1:10 9:-1:1]; y = resample(x,3,2);
subplot(2,1,1);
plot(1:19,x,'*', (0:28)*2/3 + 1,y,'o');
title('Edge Effects Not Noticeable');
legend('original','resampled');

x = [10:-1:1 2:10]; y = resample(x,3,2);
subplot(2,1,2);
plot(1:19,x,'*', (0:28)*2/3 + 1,y,'o');
title('Edge Effects Very Noticeable');
legend('original','resampled');
```

**Algorithm**

resample performs an FIR design using `firls`, followed by rate changing implemented with `upfirdn`.

**See Also**

`decimate`, `downsample`, `firls`, `interp`, `interp1`, `intfilt`, `kaiser`, `mfilt`, `spline`, `upfirdn`, `upsample`

# residuez

---

**Purpose**  $z$ -transform partial-fraction expansion

**Syntax**  
[r,p,k] = residuez(b,a)  
[b,a] = residuez(r,p,k)

**Description** residuez converts a discrete time system, expressed as the ratio of two polynomials, to partial fraction expansion, or residue, form. It also converts the partial fraction expansion back to the original polynomial coefficients.

[r,p,k] = residuez(b,a) finds the residues, poles, and direct terms of a partial fraction expansion of the ratio of two polynomials,  $b(z)$  and  $a(z)$ . Vectors b and a specify the coefficients of the polynomials of the discrete-time system  $b(z)/a(z)$  in descending powers of  $z$ .

$$b(z) = b_0 + b_1z^{-1} + b_2z^{-2} + \dots + b_mz^{-m}$$

$$a(z) = a_0 + a_1z^{-1} + a_2z^{-2} + \dots + a_nz^{-n}$$

If there are no multiple roots and  $a > n - 1$ ,

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

The returned column vector r contains the residues, column vector p contains the pole locations, and row vector k contains the direct terms. The number of poles is

$$n = \text{length}(a) - 1 = \text{length}(r) = \text{length}(p)$$

The direct term coefficient vector k is empty if length(b) is less than length(a); otherwise:

$$\text{length}(k) = \text{length}(b) - \text{length}(a) + 1$$

If  $p(j) = \dots = p(j+s-1)$  is a pole of multiplicity s, then the expansion includes terms of the form

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

$[b,a] = \text{residuez}(r,p,k)$  with three input arguments and two output arguments, converts the partial fraction expansion back to polynomials with coefficients in row vectors  $b$  and  $a$ .

The residue function in the standard MATLAB language is very similar to `residuez`. It computes the partial fraction expansion of continuous-time systems in the Laplace domain (see reference [1]), rather than discrete-time systems in the  $z$ -domain as does `residuez`.

## Algorithm

`residuez` applies standard MATLAB functions and partial fraction techniques to find  $r$ ,  $p$ , and  $k$  from  $b$  and  $a$ . It finds

- The direct terms  $a$  using `deconv` (polynomial long division) when  $\text{length}(b) > \text{length}(a) - 1$ .
- The poles using  $p = \text{roots}(a)$ .
- Any repeated poles, reordering the poles according to their multiplicities.
- The residue for each nonrepeating pole  $p_i$  by multiplying  $b(z)/a(z)$  by  $1/(1 - p_i z^{-1})$  and evaluating the resulting rational function at  $z = p_i$ .
- The residues for the repeated poles by solving

$$S2 * r2 = h - S1 * r1$$

for  $r2$  using `\`.  $h$  is the impulse response of the reduced  $b(z)/a(z)$ ,  $S1$  is a matrix whose columns are impulse responses of the first-order systems made up of the nonrepeating roots, and  $r1$  is a column containing the residues for the nonrepeating roots. Each column of matrix  $S2$  is an impulse response. For each root  $p_j$  of multiplicity  $s_j$ ,  $S2$  contains  $s_j$  columns representing the impulse responses of each of the following systems.

$$\frac{1}{1 - p_j z^{-1}}, \frac{1}{(1 - p_j z^{-1})^2}, \dots, \frac{1}{(1 - p_j z^{-1})^{s_j}}$$

The vector  $h$  and matrices  $S1$  and  $S2$  have  $n + \text{extra}$  rows, where  $n$  is the total number of roots and the internal parameter `extra`, set to 1 by default, determines the degree of overdetermination of the system of equations.

## References

- [1] Oppenheim, A.V., and R.W. Schaffer, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1975, pp. 166-170.

# residuez

---

## See Also

convmtx, deconv, poly, prony, residue, roots, ss2tf, tf2ss, tf2zp, tf2zpk, zp2ss



**Purpose** Reverse Levinson-Durbin recursion

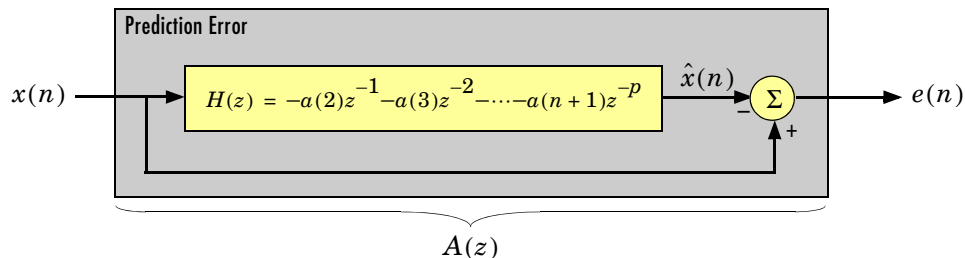
**Syntax**

```
r = rlevinson(a,efinal)
[r,u] = rlevinson(a,efinal)
[r,u,k] = rlevinson(a,efinal)
[r,u,k,e] = rlevinson(a,efinal)
```

**Description** The reverse Levinson-Durbin recursion implements the step-down algorithm for solving the following symmetric Toeplitz system of linear equations for  $r$ , where  $r = [r(1) \ \dots \ r(p+1)]$  and  $r(i)^*$  denotes the complex conjugate of  $r(i)$ .

$$\begin{bmatrix} r(1) & r(2)^* & \dots & r(p)^* \\ r(2) & r(1) & \dots & r(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(p) & \dots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(p+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(p+1) \end{bmatrix}$$

$r = \text{rlevinson}(a, \text{efinal})$  solves the above system of equations for  $r$  given vector  $a$ , where  $a = [1 \ a(2) \ \dots \ a(p+1)]$ . In linear prediction applications,  $r$  represents the autocorrelation sequence of the input to the prediction error filter, where  $r(1)$  is the zero-lag element. The figure below shows the typical filter of this type, where  $H(z)$  is the optimal linear predictor,  $x(n)$  is the input signal,  $\hat{x}(n)$  is the predicted signal, and  $e(n)$  is the prediction error.



Input vector  $a$  represents the polynomial coefficients of this prediction error filter in descending powers of  $z$ .

$$A(z) = 1 + a(2)z^{-1} + \dots + a(n+1)z^{-p}$$

The filter must be minimum phase to generate a valid autocorrelation sequence. `efinal` is the scalar prediction error power, which is equal to the variance of the prediction error signal,  $\sigma^2(e)$ .

`[r,u] = rlevinson(a,efinal)` returns upper triangular matrix  $U$  from the  $UDU^*$  decomposition

$$R^{-1} = UE^{-1}U^*$$

where

$$R = \begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \cdots & r(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix}$$

and  $E$  is a diagonal matrix with elements returned in output `e` (see below). This decomposition permits the efficient evaluation of the inverse of the autocorrelation matrix,  $R^{-1}$ .

Output matrix `u` contains the prediction filter polynomial, `a`, from each iteration of the reverse Levinson-Durbin recursion

$$U = \begin{bmatrix} a_1(1)^* & a_2(2)^* & \cdots & a_{p+1}(p+1)^* \\ 0 & a_2(1)^* & \ddots & a_{p+1}(p)^* \\ 0 & 0 & \ddots & a_{p+1}(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{p+1}(1)^* \end{bmatrix}$$

where  $a_i(j)$  is the  $j$ th coefficient of the  $i$ th order prediction filter polynomial (i.e., step  $i$  in the recursion). For example, the 5th order prediction filter polynomial is

$$a5 = u(5:-1:1,5)'$$

Note that `u(p+1:-1:1,p+1)'` is the input polynomial coefficient vector `a`.

`[r,u,k] = rlevinson(a,efinal)` returns a vector `k` of length  $(p+1)$  containing the reflection coefficients. The reflection coefficients are the conjugates of the values in the first row of `u`.

`k = conj(u(1,2:end))`

`[r,u,k,e] = rlevinson(a,efinal)` returns a vector of length  $p+1$  containing the prediction errors from each iteration of the reverse Levinson-Durbin recursion: `e(1)` is the prediction error from the first-order model, `e(2)` is the prediction error from the second-order model, and so on.

These prediction error values form the diagonal of the matrix  $E$  in the  $UDU^*$  decomposition of  $R^{-1}$ .

$$R^{-1} = UE^{-1}U^*$$

## References

[1] Kay, S.M., *Modern Spectral Estimation: Theory and Application*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

## See Also

`levinson`, `lpc`, `prony`, `stmcb`

# rooteig

---

**Purpose** Frequency and power content using eigenvector method

**Syntax**

```
[w,pow] = rooteig(x,p)
[f,pow] = rooteig(...,fs)
[w,pow] = rooteig(...,'corr')
```

**Description** `[w,pow] = rooteig(x,p)` estimates the frequency content in the time samples of a signal `x`, and returns `w`, a vector of frequencies in rad/sample, and the corresponding signal power in the vector `pow` in units of power, such as volts<sup>2</sup>. The input signal `x` is specified either as:

- A row or column vector representing one observation of the signal
- A rectangular array for which each row of `x` represents a separate observation of the signal (for example, each row is one output of an array of sensors, as in array processing), such that `x' * x` is an estimate of the correlation matrix

---

**Note** You can use the output of `corrmtx` to generate such an array `x`.

---

You can specify the second input argument `p` as either:

- A scalar integer. In this case, the signal subspace dimension is `p`.
- A two-element vector. In this case, `p(2)`, the second element of `p`, represents a threshold that is multiplied by  $\lambda_{\min}$ , the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold  $\lambda_{\min} * p(2)$  are assigned to the noise subspace. In this case, `p(1)` specifies the maximum dimension of the signal subspace.

The extra threshold parameter in the second entry in `p` provides you more flexibility and control in assigning the noise and signal subspaces.

The length of the vector `w` is the computed dimension of the signal subspace. For real-valued input data `x`, the length of the corresponding power vector `pow` is given by

$$\text{length}(\text{pow}) = 0.5 * \text{length}(w)$$

For complex-valued input data `x`, `pow` and `w` have the same length.

`[f,pow] = rooteig(...,fs)` returns the vector of frequencies `f` calculated in Hz. You supply the sampling frequency `fs` in Hz. If you specify `fs` with the empty vector `[]`, the sampling frequency defaults to 1 Hz.

`[w,pow] = rooteig(...,'corr')` forces the input argument `x` to be interpreted as a correlation matrix rather than a matrix of signal data. For this syntax, you must supply a square matrix for `x`, and all of its eigenvalues must be nonnegative.

---

**Note** You can place the string `'corr'` anywhere after `p`.

---

## Examples

Find the frequency content in a signal composed of three complex exponentials in noise. Use the modified covariance method to estimate the correlation matrix used by the eigenvector method:

```
randn('state',1); n=0:99;
s = exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);

% Estimate correlation matrix using modified covariance method.

X=corrmtx(s,12,'mod');
[W,P] = rooteig(X,3)

W =
    0.7811
    1.5767
    1.0554

P =
    3.9971
    1.1362
    1.4102
```

## Algorithm

The eigenvector method used by `rooteig` is the same as that used by `peig`. The algorithm performs eigenspace analysis of the signal's correlation matrix in order to estimate the signal's frequency content.

The difference between `peig` and `rooteig` is:

# rooteig

---

- `peig` returns the pseudospectrum at all frequency samples.
- `rooteig` returns the estimated discrete frequency spectrum, along with the corresponding signal power estimates.

`rooteig` is most useful for frequency estimation of signals made up of a sum of sinusoids embedded in additive white Gaussian noise.

## See Also

`corrmtx`, `peig`, `pmusic`, powerest method of spectrum, `rootmusic`, `spectrum.eigenvector`

**Purpose** Frequency and power content using root MUSIC algorithm

**Syntax**

```
[w,pow] = rootmusic(x,p)
[f,pow] = rootmusic(...,fs)
[w,pow] = rootmusic(...,'corr')
```

**Description** `[w,pow] = rootmusic(x,p)` estimates the frequency content in the time samples of a signal `x`, and returns `w`, a vector of frequencies in rad/sample, and the corresponding signal power in the vector `pow` in dB per rad/sample. The input signal `x` is specified either as:

- A row or column vector representing one observation of the signal
- A rectangular array for which each row of `x` represents a separate observation of the signal (for example, each row is one output of an array of sensors, as in array processing), such that  $x' * x$  is an estimate of the correlation matrix

---

**Note** You can use the output of `corrmtx` to generate such an array `x`.

---

The second input argument, `p` is the number of complex sinusoids in `x`. You can specify `p` as either:

- A scalar integer. In this case, the signal subspace dimension is `p`.
- A two-element vector. In this case, `p(2)`, the second element of `p`, represents a threshold that is multiplied by  $\lambda_{\min}$ , the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold  $\lambda_{\min} * p(2)$  are assigned to the noise subspace. In this case, `p(1)` specifies the maximum dimension of the signal subspace.

The extra threshold parameter in the second entry in `p` provides you more flexibility and control in assigning the noise and signal subspaces.

The length of the vector `w` is the computed dimension of the signal subspace. For real-valued input data `x`, the length of the corresponding power vector `pow` is given by

$$\text{length}(\text{pow}) = 0.5 * \text{length}(w)$$

# rootmusic

---

For complex-valued input data  $x$ ,  $pow$  and  $w$  have the same length.

`[f,pow] = rootmusic(...,fs)` returns the vector of frequencies  $f$  calculated in Hz. You supply the sampling frequency  $fs$  in Hz. If you specify  $fs$  with the empty vector `[]`, the sampling frequency defaults to 1 Hz.

`[w,pow] = rootmusic(...,'corr')` forces the input argument  $x$  to be interpreted as a correlation matrix rather than a matrix of signal data. For this syntax, you must supply a square matrix for  $x$ , and all of its eigenvalues must be nonnegative.

---

**Note** You can place the string `'corr'` anywhere after  $p$ .

---

## Examples

Find the frequency content in a signal composed of three complex exponentials in noise. Use the modified covariance method to estimate the correlation matrix used by the MUSIC algorithm:

```
randn('state',1); n=0:99;
s = exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);
% Estimate correlation matrix using modified covariance method.
X=corrmtx(s,12,'mod');
[W,P] = rootmusic(X,3)

W =
    1.5769
    0.7817
    1.0561

P =
    1.1358
    3.9975
    1.4102
```

## Algorithm

The MUSIC algorithm used by `rootmusic` is the same as that used by `pmusic`. The algorithm performs eigenspace analysis of the signal's correlation matrix in order to estimate the signal's frequency content.



The difference between `pmusic` and `rootmusic` is:

- `pmusic` returns the pseudospectrum at all frequency samples.
- `rootmusic` returns the estimated discrete frequency spectrum, along with the corresponding signal power estimates.

`rootmusic` is most useful for frequency estimation of signals made up of a sum of sinusoids embedded in additive white Gaussian noise.

## Diagnostics

If the input signal, `x` is real and an odd number of sinusoids, `p` is specified, this error message is displayed

Real signals require an even number `p` of complex sinusoids.

## See Also

`corrmtx`, `peig`, `pmusic`, powerest method of `spectrum`, `rooteig`, `spectrum.music`

# sawtooth

---

**Purpose** Sawtooth or triangle wave

**Syntax**  
`x = sawtooth(t)`  
`x = sawtooth(t,width)`

**Description** `sawtooth(t)` generates a sawtooth wave with period  $2\pi$  for the elements of time vector `t`. `sawtooth(t)` is similar to `sin(t)`, but creates a sawtooth wave with peaks of -1 and 1 instead of a sine wave. The sawtooth wave is defined to be -1 at multiples of  $2\pi$  and to increase linearly with time with a slope of  $1/\pi$  at all other times.

`sawtooth(t,width)` generates a modified triangle wave where `width`, a scalar parameter between 0 and 1, determines the point between 0 and  $2\pi$  at which the maximum occurs. The function increases from -1 to 1 on the interval 0 to  $2\pi*\text{width}$ , then decreases linearly from 1 to -1 on the interval  $2\pi*\text{width}$  to  $2\pi$ . Thus a parameter of 0.5 specifies a standard triangle wave, symmetric about time instant  $\pi$  with peak-to-peak amplitude of 1. `sawtooth(t,1)` is equivalent to `sawtooth(t)`.

**See Also** `chirp`, `cos`, `diric`, `gauspuls`, `pulstran`, `rectpuls`, `sin`, `sinc`, `square`, `tripuls`

<b>Purpose</b>	Compute reflection coefficients from autocorrelation sequence
<b>Syntax</b>	<pre>k = schurrc(r) [k,e] = schurrc(r)</pre>
<b>Description</b>	<p><code>k = schurrc(r)</code> uses the Schur algorithm to compute a vector <code>k</code> of reflection coefficients from a vector <code>r</code> representing an autocorrelation sequence. <code>k</code> and <code>r</code> are the same size. The reflection coefficients represent the lattice parameters of a prediction filter for a signal with the given autocorrelation sequence, <code>r</code>. When <code>r</code> is a matrix, <code>schurrc</code> treats each column of <code>r</code> as an independent autocorrelation sequence, and produces a matrix <code>k</code>, the same size as <code>r</code>. Each column of <code>k</code> represents the reflection coefficients for the lattice filter for predicting the process with the corresponding autocorrelation sequence <code>r</code>.</p> <p><code>[k,e] = schurrc(r)</code> also computes the scalar <code>e</code>, the prediction error variance. When <code>r</code> is a matrix, <code>e</code> is a row vector. The length of <code>e</code> is the same as the number of columns of <code>r</code>.</p>
<b>Examples</b>	<p>Create an autocorrelation sequence from the MATLAB speech signal contained in <code>mtlb.mat</code>, and use the Schur algorithm to compute the reflection coefficients of a lattice prediction filter for this autocorrelation sequence:</p> <pre>load mtlb r = xcorr(mtlb(1:5), 'unbiased'); k = schurrc(r(5:end))</pre> <p><code>k =</code></p> <pre>-0.7583  0.1384  0.7042 -0.3699</pre>
<b>References</b>	[1] Proakis, J. and D. Manolakis, <i>Digital Signal Processing: Principles, Algorithms, and Applications</i> , Third edition, Prentice-Hall, 1996, pp. 868-873.
<b>See Also</b>	<code>levinson</code>

# seqperiod

---

**Purpose** Compute period of sequence

**Syntax** `p = seqperiod(x)`  
`[p,num] = seqperiod(x)`

**Description** `p = seqperiod(x)` returns the integer `p` that corresponds to the period of the sequence in a vector `x`. The period `p` is computed as the minimum length of a subsequence `x(1:p)` of `x` that repeats itself continuously every `p` samples in `x`. The length of `x` does not have to be a multiple of `p`, so that an incomplete repetition is permitted at the end of `x`. If the sequence `x` is not periodic, then `p = length(x)`.

- If `x` is a matrix, then `seqperiod` checks for periodicity along each column of `x`. The resulting output `p` is a row vector with the same number of columns as `x`.
- If `x` is a multidimensional array, then `seqperiod` checks for periodicity along the first nonsingleton dimension of `x`. In this case:
  - `p` is a multidimensional array of integers with a leading singleton dimension.
  - The lengths of the remaining dimensions of `p` correspond to those of the dimensions of `x` after the first nonsingleton one.

`[p,num] = seqperiod(x)` also returns the number `num` of repetitions of `x(1:p)` in `x`. `num` might not be an integer.

## Examples

```
x = [4 0 1 6;  
     2 0 2 7;  
     4 0 1 5;  
     2 0 5 6];  
  
p = seqperiod(x)  
  
p =  
     2     1     4     3
```

The result implies:

- The first column of `x` has period 2.
- The second column of `x` has period 1.
- The third column of `x` is not periodic, so `p(3)` is just the number of rows of `x`.

- The fourth column of  $x$  has period 3, although the last (second) repetition of the periodic sequence is incomplete.

# sgolay

---

**Purpose** Savitzky-Golay filter design

**Syntax**

```
b = sgolay(k,f)
b = sgolay(k,f,w)
[b,g] = sgolay(...)
```

**Description** `b = sgolay(k,f)` designs a Savitzky-Golay FIR smoothing filter `b`. The polynomial order `k` must be less than the frame size, `f`, which must be odd. If `k = f - 1`, the designed filter produces no smoothing. The output, `b`, is an `f`-by-`f` matrix whose rows represent the time-varying FIR filter coefficients. In a smoothing filter implementation (for example, `sgolayfilt`), the last  $(f - 1) / 2$  rows (each an FIR filter) are applied to the signal during the startup transient, and the first  $(f - 1) / 2$  rows are applied to the signal during the terminal transient. The center row is applied to the signal in the steady state.

`b = sgolay(k,f,w)` specifies a weighting vector `w` with length `f`, which contains the real, positive-valued weights to be used during the least-squares minimization.

`[b,g] = sgolay(...)` returns the matrix `g` of differentiation filters. Each column of `g` is a differentiation filter for derivatives of order `p-1` where `p` is the column index. Given a signal `x` of length `f`, you can find an estimate of the  $p^{\text{th}}$  order derivative, `xp`, of its middle value from:

$$x_p((f+1)/2) = (\text{factorial}(p)) * g(:,p+1)' * x$$

**Remarks** Savitzky-Golay smoothing filters (also called digital smoothing polynomial filters or least squares smoothing filters) are typically used to “smooth out” a noisy signal whose frequency span (without noise) is large. In this type of application, Savitzky-Golay smoothing filters perform much better than standard averaging FIR filters, which tend to filter out a significant portion of the signal’s high frequency content along with the noise. Although Savitzky-Golay filters are more effective at preserving the pertinent high frequency components of the signal, they are less successful than standard averaging FIR filters at rejecting noise when noise levels are particularly high. The particular formulation of Savitzky-Golay filters preserves various moment orders better than other smoothing methods, which tend to preserve peak widths and heights better than Savitzky-Golay.

Savitzky-Golay filters are optimal in the sense that they minimize the least-squares error in fitting a polynomial to each frame of noisy data.

## Examples

Use `sgolay` to smooth a noisy sinusoid and compare the resulting first and second derivatives to the first and second derivatives computed using `diff`. Notice how using `diff` amplifies the noise and generates useless results.

```
N = 4; % Order of polynomial fit
F = 21; % Window length
[b,g] = sgolay(N,F); % Calculate S-G coefficients

dx = .2;
xLim = 200;
x = 0:dx:xLim-1;

y = 5*sin(0.4*pi*x) + randn(size(x)); % Sinusoid with noise

HalfWin = ((F+1)/2) - 1;
for n = (F+1)/2:996-(F+1)/2,
    % Zero-th derivative (smoothing only)
    SG0(n) = dot(g(:,1), y(n - HalfWin: n + HalfWin));

    % 1st differential
    SG1(n) = dot(g(:,2), y(n - HalfWin: n + HalfWin));

    % 2nd differential
    SG2(n) = 2*dot(g(:,3)', y(n - HalfWin: n + HalfWin))';
end

SG1 = SG1/dx; % Turn differential into derivative
SG2 = SG2/(dx*dx); % and into 2nd derivative

% Scale the "diff" results
DiffD1 = (diff(y(1:length(SG0)+1)))/ dx;
DiffD2 = (diff(diff(y(1:length(SG0)+2)))) / (dx*dx);

subplot(3,1,1);
plot([y(1:length(SG0))', SG0'])
legend('Noisy Sinusoid','S-G Smoothed sinusoid')
```

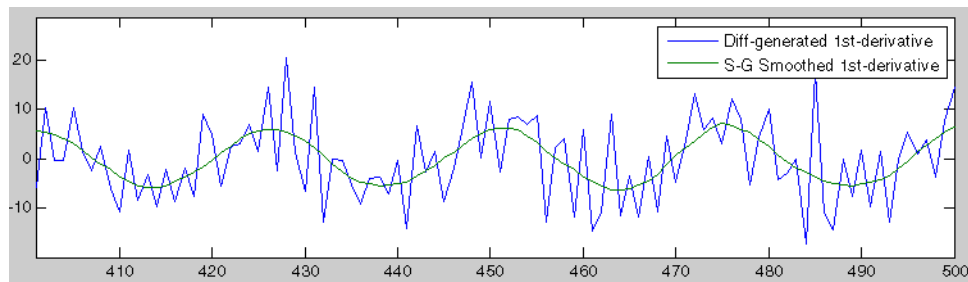
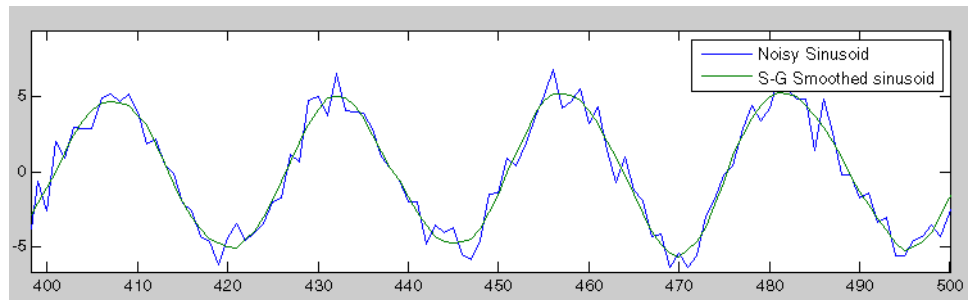
```
subplot(3, 1, 2);  
plot([DiffD1',SG1'])  
legend('Diff-generated 1st-derivative', 'S-G Smoothed  
1st-derivative')
```

```
subplot(3, 1, 3);  
plot([DiffD2',SG2'])  
legend('Diff-generated 2nd-derivative', 'S-G Smoothed  
2nd-derivative')
```

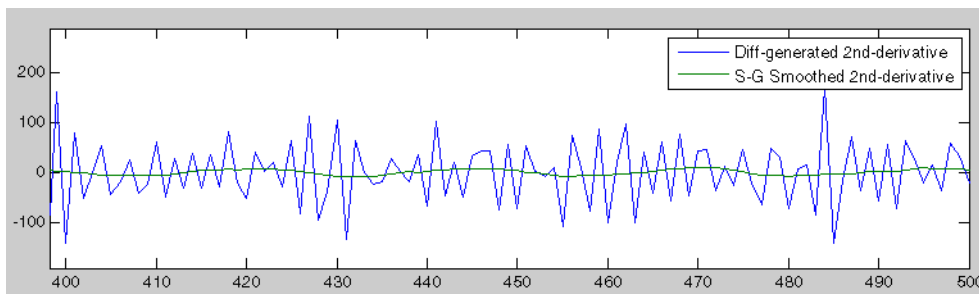
---

**Note** The figures below are zoomed in each figure window panel to show more detail.

---





**References**

[1] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

**See Also**

`fir1`, `firls`, `filter`, `sgolayfilt`

# sgolayfilt

---

**Purpose** Savitzky-Golay filtering

**Syntax**

```
y = sgolayfilt(x,k,f)
y = sgolayfilt(x,k,f,w)
y = sgolayfilt(x,k,f,w,dim)
```

**Description** `y = sgolayfilt(x,k,f)` applies a Savitzky-Golay FIR smoothing filter to the data in vector `x`. If `x` is a matrix, `sgolayfilt` operates on each column. The polynomial order `k` must be less than the frame size, `f`, which must be odd. If `k = f - 1`, the filter produces no smoothing.

`y = sgolayfilt(x,k,f,w)` specifies a weighting vector `w` with length `f`, which contains the real, positive-valued weights to be used during the least-squares minimization. If `w` is not specified or if it is specified as empty, `[]`, `w` defaults to an identity matrix.

`y = sgolayfilt(x,k,f,w,dim)` specifies the dimension, `dim`, along which the filter operates. If `dim` is not specified, `sgolayfilt` operates along the first non-singleton dimension; that is, dimension 1 for column vectors and nontrivial matrices, and dimension 2 for row vectors.

**Remarks** Savitzky-Golay smoothing filters (also called digital smoothing polynomial filters or least-squares smoothing filters) are typically used to “smooth out” a noisy signal whose frequency span (without noise) is large. In this type of application, Savitzky-Golay smoothing filters perform much better than standard averaging FIR filters, which tend to filter out a significant portion of the signal’s high frequency content along with the noise. Although Savitzky-Golay filters are more effective at preserving the pertinent high frequency components of the signal, they are less successful than standard averaging FIR filters at rejecting noise.

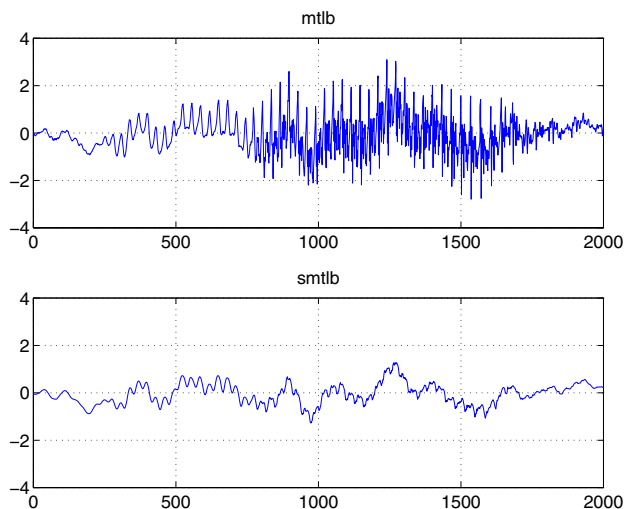
Savitzky-Golay filters are optimal in the sense that they minimize the least-squares error in fitting a polynomial to frames of noisy data.

**Examples** Smooth the `mtlb` signal by applying a cubic Savitzky-Golay filter to data frames of length 41:

```
load mtlb % Load the data.
smtlb = sgolayfilt(mtlb,3,41); % Apply the 3rd-order filter.
```

```
subplot(2,1,1)
plot([1:2000],mtlb(1:2000)); axis([0 2000 -4 4]);
title('mtlb'); grid;

subplot(2,1,2)
plot([1:2000],smtlb(1:2000)); axis([0 2000 -4 4]);
title('smtlb'); grid;
```



## References

[1] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

## See Also

medfilt1, filter, sgolay, sosfilt

# sigwin

---

**Purpose** Signal processing window

**Syntax** `w = sigwin.window(input1,...)`

**Description** `w = sigwin.window` returns a window object, *w*, of type *window*. Each window takes one or more inputs. If you specify a *sigwin.window* with no inputs, a default window of length 64 is created.

---

**Note** You must use a *window* with *sigwin*.

---

## Constructors

*window* for *sigwin* specifies the type of window. All windows in the Signal Processing Toolbox are available for use with *sigwin*. For a complete list, see the window reference page. To get help on a *sigwin*, use the syntax `help sigwin.window` at the MATLAB prompt.

## Methods

Methods provide ways of performing functions directly on your *sigwin* object without having to specify the window parameters again. You can apply this method directly on the variable you assigned to your *sigwin* object.

Method	Description
<code>generate</code>	Returns a column vector of values representing the window.
<code>info</code>	Returns information about the window object.
<code>winwrite</code>	Writes an ASCII file that contains window weights for a single window object or a vector of window objects. Default filename is <code>untitled.wf</code> .  <code>winwrite(Hd,filename)</code> writes to a disk file named <code>filename</code> in the current working directory. The <code>.wf</code> extension is added automatically.

## Viewing Object Parameters

As with any object, you can use `get` to view a sigwin object's parameters. To see a specific parameter,

```
get(w, 'parameter')
```

or to see all parameters for an object,

```
get(w)
```

## Changing Object Parameters

To set specific parameters,

```
set(w, 'parameter1', value, 'parameter2', value, ...)
```

Note that you must use single quotation marks around the parameter name.

## Examples

Create a default Bartlett window and view the results in the Window Visualization Tool (`wvtool`). See `bartlett` for information on Bartlett windows:

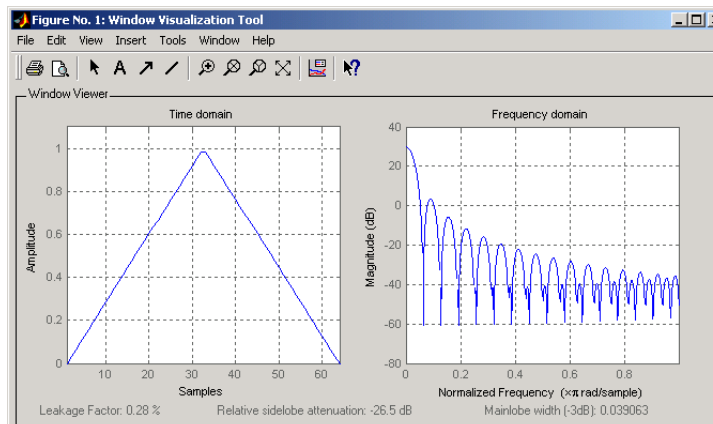
```
w=sigwin.bartlett
```

```
w =
```

```
Length: 64
```

```
Name: 'Bartlett'
```

```
wvtool(w)
```



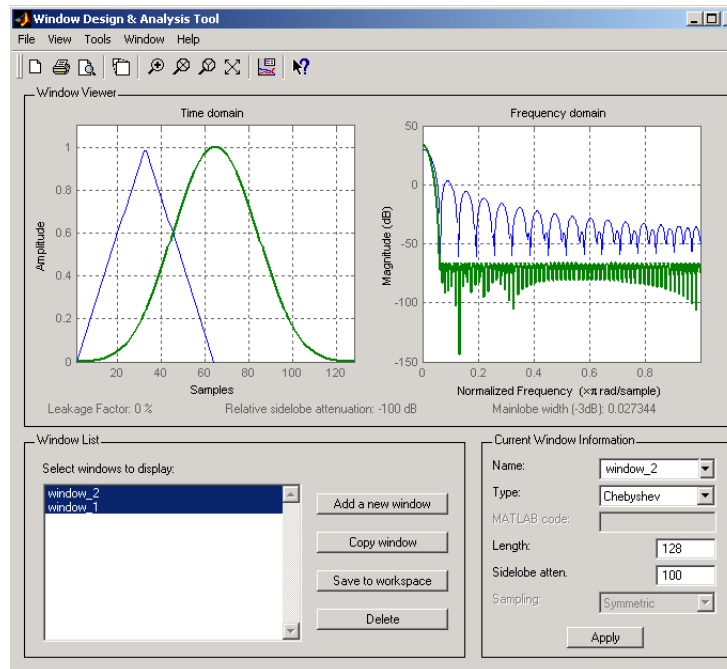
Create a 128-point Chebyshev window with 100 dB of sidelobe attenuation. (See `chebwin` for information on Chebyshev windows.) View the results of this and the above Bartlett window in the Window Design and Analysis Tool (`wintool`):

```
w1=sigwin.chebwin(128,100)
```

```
w1 =
```

```
    Length: 128  
    Name: 'Chebyshev'  
    SidelobeAtten: 100
```

```
wintool(w,w1)
```



To save the window values in a vector, use:

```
d = generate(w);
```

**See Also**

window, wintool, wvtool

# sinc

---

**Purpose** Sinc

**Syntax** `y = sinc(x)`

**Description** `sinc` computes the sinc function of an input vector or array, where the sinc function is

$$\text{sinc}(t) = \begin{cases} 1, & t = 0 \\ \frac{\sin(\pi t)}{\pi t}, & t \neq 0 \end{cases}$$

This function is the continuous inverse Fourier transform of the rectangular pulse of width  $2\pi$  and height 1.

$$\text{sinc}(t) = \frac{1}{2\pi} \int_{-\pi}^{\pi} e^{j\omega t} d\omega$$

`y = sinc(x)` returns an array `y` the same size as `x`, whose elements are the sinc function of the elements of `x`.

The space of functions bandlimited in the frequency range  $\omega \in [-\pi, \pi]$  is spanned by the infinite (yet countable) set of sinc functions shifted by integers. Thus any such bandlimited function  $g(t)$  can be reconstructed from its samples at integer spacings.

$$g(t) = \sum_{n=-\infty}^{\infty} g(n) \text{sinc}(t - n)$$

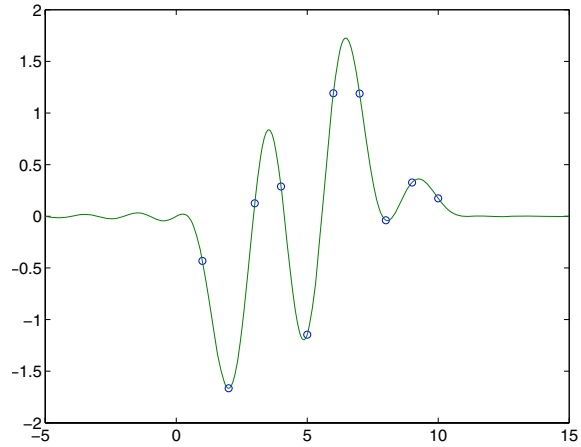
## Examples

Perform ideal bandlimited interpolation by assuming that the signal to be interpolated is 0 outside of the given time interval and that it has been sampled at exactly the Nyquist frequency:

```
t = (1:10)'; % A column vector of time samples
randn('state',0);
x = randn(size(t)); % A column vector of data
ts = linspace(-5,15,600)'; % times at which to interpolate data
```



```
y = sinc(ts(:,ones(size(t))) - t(:,ones(size(ts))))'*x;  
plot(t,x,'o',ts,y)
```

**See Also**

chirp, cos, diric, gauspuls, pulstran, rectpuls, sawtooth, sin, square, tripuls

# sos2cell

---

**Purpose** Convert second-order sections matrix to cell array

**Syntax**

```
c = sos2cell(m)
c = sos2cell(m,g)
```

**Description** `c = sos2cell(m)` changes an  $L$ -by-6 second-order section matrix  $m$  generated by `tf2sos` into a 1-by- $L$  cell array of 1-by-2 cell arrays  $c$ . You can use  $c$  to specify a quantized filter with  $L$  cascaded second-order sections.

The matrix  $m$  should have the form

$$m = [b_1 \ a_1; b_2 \ a_2; \dots; b_L \ a_L]$$

where both  $b_i$  and  $a_i$ , with  $i = 1, \dots, L$ , are 1-by-3 row vectors. The resulting  $c$  is a 1-by- $L$  cell array of cells of the form

$$c = \{ \{b_1 \ a_1\} \ \{b_2 \ a_2\} \ \dots \ \{b_L \ a_L\} \ }$$

`c = sos2cell(m,g)` with the optional gain term  $g$ , prepends the constant value  $g$  to  $c$ . When you use the added gain term in the command,  $c$  is a 1-by- $L$  cell array of cells of the form

$$c = \{ \{g, 1\} \ \{b_1, a_1\} \ \{b_2, a_2\} \ \dots \ \{b_L, a_L\} \ }$$

**Examples** Use `sos2cell` to convert the 2-by-6 second-order section matrix produced by `tf2sos` into a 1-by-2 cell array  $c$  of cells. Display the second entry in the first cell in  $c$ :

```
[b,a] = ellip(4,0.5,20,0.6);
m = tf2sos(b,a);
c = sos2cell(m);
c{1}{2}

ans =

    1.0000    0.1677    0.2575
```

**See Also** `tf2sos`, `cell2sos`

**Purpose** Convert digital filter second-order section parameters to state-space form

**Syntax**  
 $[A,B,C,D] = \text{sos2ss}(\text{sos})$   
 $[A,B,C,D] = \text{sos2ss}(\text{sos},g)$

**Description** sos2ss converts a second-order section representation of a given digital filter to an equivalent state-space representation.

$[A,B,C,D] = \text{sos2ss}(\text{sos})$  converts the system  $\text{sos}$ , in second-order section form, to a single-input, single-output state-space representation.

$$\begin{aligned} x[n+1] &= Ax[n] + Bu[n] \\ y[n] &= Cx[n] + Du[n] \end{aligned}$$

The discrete transfer function in second-order section form is given by

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

$\text{sos}$  is a  $L$ -by-6 matrix organized as

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

The entries of  $\text{sos}$  must be real for proper conversion to state space. The returned matrix  $A$  is size  $N$ -by- $N$ , where  $N = 2L-1$ ,  $B$  is a length  $N-1$  column vector,  $C$  is a length  $N-1$  row vector, and  $D$  is a scalar.

$[A,B,C,D] = \text{sos2ss}(\text{sos},g)$  converts the system  $\text{sos}$  in second-order section form with gain  $g$ .

$$H(z) = g \prod_{k=1}^L H_k(z)$$

## Examples

Compute the state-space representation of a simple second-order section system with a gain of 2:

```
sos = [1 1 1 1 0 -1; -2 3 1 1 10 1];  
[A,B,C,D] = sos2ss(sos)
```

```
A =  
   -10     0    10     1  
     1     0     0     0  
     0     1     0     0  
     0     0     1     0
```

```
B =  
     1  
     0  
     0  
     0
```

```
C =  
    21     2   -16    -1
```

```
D =  
    -2
```

## Algorithm

sos2ss first converts from second-order sections to transfer function using sos2tf, and then from transfer function to state-space using tf2ss.

## See Also

sos2tf, sos2zp, ss2sos, tf2ss, zp2ss

**Purpose** Convert digital filter second-order section data to transfer function form

**Syntax**  
`[b,a] = sos2tf(sos)`  
`[b,a] = sos2tf(sos,g)`

**Description** `sos2tf` converts a second-order section representation of a given digital filter to an equivalent transfer function representation.

`[b,a] = sos2tf(sos)` returns the numerator coefficients `b` and denominator coefficients `a` of the transfer function that describes a discrete-time system given by `sos` in second-order section form. The second-order section format of  $H(z)$  is given by

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

`sos` is an  $L$ -by-6 matrix that contains the coefficients of each second-order section stored in its rows.

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

Row vectors `b` and `a` contain the numerator and denominator coefficients of  $H(z)$  stored in descending powers of  $z$ .

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2z^{-1} + \dots + b_{n+1}z^{-n}}{a_1 + a_2z^{-1} + \dots + a_{m+1}z^{-m}}$$

`[b,a] = sos2tf(sos,g)` returns the transfer function that describes a discrete-time system given by `sos` in second-order section form with gain `g`.

$$H(z) = g \prod_{k=1}^L H_k(z)$$

# sos2tf

---

## Examples

Compute the transfer function representation of a simple second-order section system:

```
sos = [1 1 1 1 0 -1; -2 3 1 1 10 1];  
[b,a] = sos2tf(sos)
```

```
b =  
-2    1    2    4    1
```

```
a =  
1   10    0  -10  -1
```

## Algorithm

sos2tf uses the conv function to multiply all of the numerator and denominator second-order polynomials together.

## See Also

latc2tf, sos2ss, sos2zp, ss2tf, tf2sos, zp2tf

**Purpose** Convert digital filter second-order section parameters to zero-pole-gain form

**Syntax**  
 $[z,p,k] = \text{sos2zp}(\text{sos})$   
 $[z,p,k] = \text{sos2zp}(\text{sos},g)$

**Description** sos2zp converts a second-order section representation of a given digital filter to an equivalent zero-pole-gain representation.

$[z,p,k] = \text{sos2zp}(\text{sos})$  returns the zeros  $z$ , poles  $p$ , and gain  $k$  of the system given by  $\text{sos}$  in second-order section form. The second-order section format of  $H(z)$  is given by

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

$\text{sos}$  is an  $L$ -by-6 matrix that contains the coefficients of each second-order section in its rows.

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

Column vectors  $z$  and  $p$  contain the zeros and poles of the transfer function  $H(z)$ .

$$H(z) = k \frac{(z - z_1)(z - z_2) \cdots (z - z_n)}{(p - p_1)(p - p_2) \cdots (p - p_m)}$$

where the orders  $n$  and  $m$  are determined by the matrix  $\text{sos}$ .

$[z,p,k] = \text{sos2zp}(\text{sos},g)$  returns the zeros  $z$ , poles  $p$ , and gain  $k$  of the system given by  $\text{sos}$  in second-order section form with gain  $g$ .

$$H(z) = g \prod_{k=1}^L H_k(z)$$

# sos2zp

---

## Examples

Compute the poles, zeros, and gain of a simple system in second-order section form:

```
sos = [1 1 1 1 0 -1; -2 3 1 1 10 1];  
[z,p,k] = sos2zp(sos)
```

```
z =  
-0.5000 + 0.8660i  
-0.5000 - 0.8660i  
1.7808  
-0.2808
```

```
p =  
-1.0000  
1.0000  
-9.8990  
-0.1010
```

```
k =  
-2
```

## Algorithm

sos2zp finds the poles and zeros of each second-order section by repeatedly calling tf2zp.

## See Also

sos2ss, sos2tf, ss2zp, tf2zp, tf2zpk, zp2sos



**Purpose** Second-order (biquadratic) IIR digital filtering

**Syntax** `y = sosfilt(sos,x)`

**Description** `y = sosfilt(sos,x)` applies the second-order section digital filter `sos` to the vector `x`. The output, `y`, is the same length as `x`.

`sos` represents the second-order section digital filter  $H(z)$

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

by an  $L$ -by-6 matrix containing the coefficients of each second-order section in its rows.

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

If `x` is a matrix, `sosfilt` applies the filter to each column of `x` independently. Output `y` is a matrix of the same size, containing the filtered data corresponding to each column of `x`.

**References** [1] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

**See Also** `filter`, `medfilt1`, `sgolayfilt`

# spectrogram

---

**Purpose** Spectrogram using short-time Fourier transform

**Syntax**

```
S = spectrogram(x)
S = spectrogram(x,window)
S = spectrogram(x,window,noverlap)
S = spectrogram(x,window,noverlap,nfft)
S = spectrogram(x,window,noverlap,nfft,fs)
S = spectrogram(x,window,noverlap,F)
S = spectrogram(x,window,noverlap,F,fs)
[S,F,T] = spectrogram(...)
[S,F,T,P] = spectrogram(...)
spectrogram(...)
```

**Description** spectrogram computes the short-time Fourier transform of a signal. The spectrogram is the magnitude of this function.

`S = spectrogram(x)` returns the spectrogram of the input signal vector `x`. By default, `x` is divided into eight segments. If `x` cannot be divided exactly into eight segments, it is truncated. These default values are used.

- `window` is a Hamming window of length `nfft`.
- `noverlap` is the value that produces 50% overlap between segments.
- `nfft` is the FFT length and is the maximum of 256 or the next power of 2 greater than the length of each segment of `x`.
- `fs` is the sampling frequency, which defaults to normalized frequency.

Each column of `S` contains an estimate of the short-term, time-localized frequency content of `x`. Time increases across the columns of `S` and frequency increases down the rows.

If `x` is a length `Nx` complex signal, `S` is a complex matrix with `nfft` rows and `k` columns, where

$$k = \text{fix}((Nx - \text{noverlap}) / (\text{length}(\text{window} - \text{noverlap})))$$

For real `x`, the output `S` has  $(nfft/2+1)$  rows if `nfft` is even, and  $(nfft+1)/2$  rows if `nfft` is odd.

`S = spectrogram(x,window)` uses the window specified. If `window` is an integer, `x` is divided into segments equal to that integer value and a Hamming

window is used. If window is a vector, x is divided into segments equal to the length of window and then the segments are windowed using the window functions specified in the window vector.

`S = spectrogram(x,window,noverlap)` overlaps `noverlap` samples of each segment. `noverlap` must be an integer smaller than window or if window is a vector, smaller than the length of window.

`S = spectrogram(x,window,noverlap,nfft)` uses the `nfft` number of sampling points to calculate the discrete Fourier transform. `nfft` must be a scalar.

`S = spectrogram(x,window,noverlap,nfft,fs)` uses `fs` sampling frequency in Hz. If `fs` is specified as empty `[]`, it defaults to 1 Hz.

`S = spectrogram(x,window,noverlap,F)` uses a vector `F` of frequencies in Hz. `F` must be a vector with at least two elements. This case computes the spectrogram at the frequencies in `F` using the Goertzel algorithm. In all other syntax cases where `nfft` or a default for `nfft` is used, the short-time Fourier transform is used.

`S = spectrogram(x,window,noverlap,F,fs)` uses a vector `F` of frequencies in Hz as above and uses the `fs` sampling frequency in Hz. If `fs` is specified as empty `[]`, it defaults to 1 Hz.

`[S,F,T] = spectrogram(...)` returns a vector of frequencies `F` and a vector of times `T` at which the spectrogram is computed. The length of `F` is equal to the number of rows of `S`. The length of `T` is equal to `k`, as defined above and each value corresponds to the center of each segment.

`[S,F,T,P] = spectrogram(...)` returns a matrix `P` containing the power spectral density (PSD) of each segment. For real `x`, `P` contains the one-sided modified periodogram estimate of the PSD of each segment. For complex `x` and when you specify a vector of frequencies `F`, `P` contains the two-sided PSD.

`spectrogram(...)` plots the PSD estimate for each segment on a surface in a figure window. The plot is created using `surf(F,T,10*log10(abs(P)))`.

Using `spectrogram(...,'freqloc')` syntax and adding a `'freqloc'` string (either `'xaxis'` or `'yaxis'`) controls where MATLAB displays the frequency

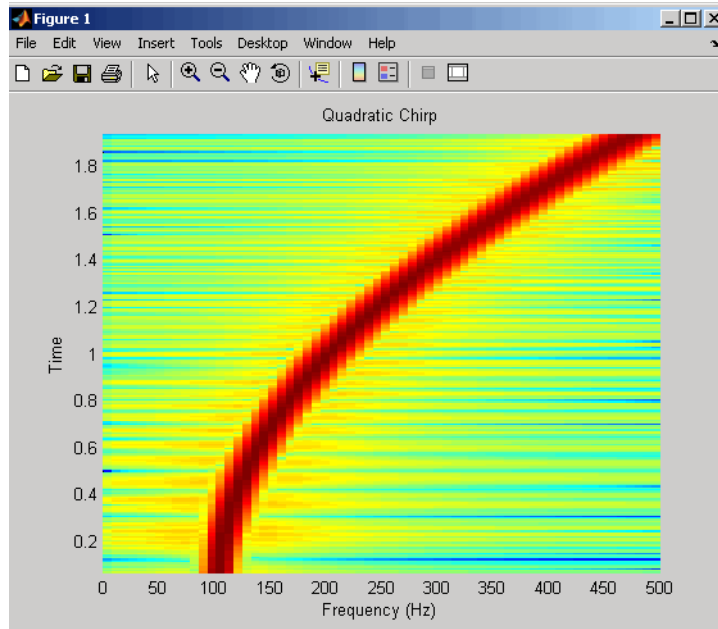
# spectrogram

axis. Using 'xaxis' displays the frequency on the  $x$ -axis. Using 'yaxis' displays frequency on the  $y$ -axis and time on the  $x$ -axis. The default is 'xaxis'. If you specify both a 'freqloc' string and output arguments, 'freqloc' is ignored.

## Examples

Compute and display the PSD of each segment of a quadratic chirp, which starts at 100 Hz and crosses 200 Hz at  $t = 1$  sec.

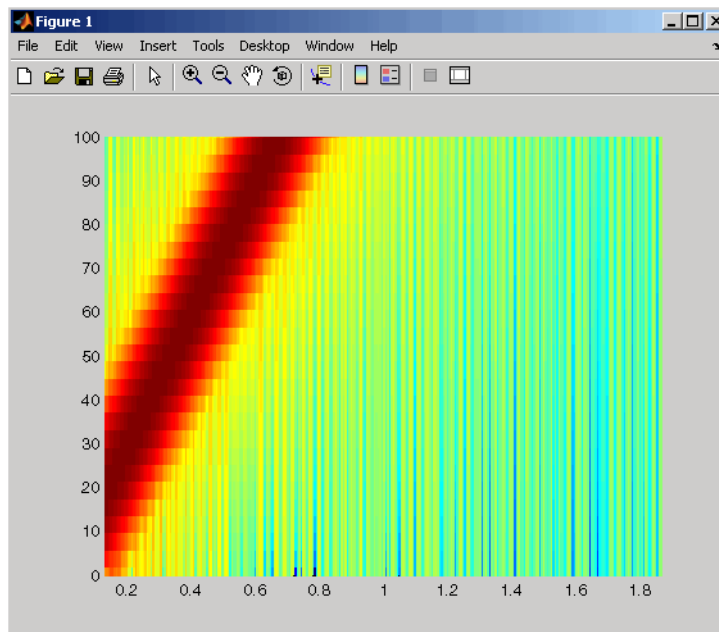
```
T = 0:0.001:2;  
X = chirp(T,100,1,200,'q');  
spectrogram(X,128,120,128,1E3);  
title('Quadratic Chirp');
```



Compute and display the PSD of each segment of a linear chirp, which starts at DC and crosses 150 Hz at  $t = 1$  sec. Display the frequency on the  $y$ -axis.

```
T = 0:0.001:2;  
X = chirp(T,0,1,150);  
F = 0:.1:100;  
[Y,F,T,P] = spectrogram(X,256,250,F,1E3,'yaxis');
```

```
% The following code produces the same result as calling  
% spectrogram with no outputs:  
surf(T,F,10*log10(abs(P)),'EdgeColor','none');  
axis xy; axis tight; colormap(jet); view(0,90);  
xlabel('Time');  
ylabel('Frequency (Hz)');
```



## References

- [1] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 713-718.
- [2] Rabiner, L.R., and R.W. Schaffer, *Digital Processing of Speech Signals*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

## See Also

goertzel, periodogram, pwelch, spectrum.periodogram, spectrum.welch

# spectrum

---

**Purpose** Spectral estimation

**Syntax** `Hs = spectrum.estimate(input1,...)`

**Description** `Hs = spectrum.estimate(input1,...)` returns a spectral estimation object `Hs` of type `estimate`. This object contains all the parameter information needed for the specified estimation method. Each estimation method takes one or more inputs, which are described on the individual reference pages.

---

**Note** If you need to obtain confidence intervals, use the `pmtm` function.

---

## Estimation Methods

---

**Note** You must use a spectral `estimate` with `spectrum`.

---

Estimation methods for `spectrum` specify the type of spectral estimation method to use. Available estimation methods for `spectrum` are listed below.

<code>spectrum.estimate</code>	Description	Corresponding Function
<code>spectrum.burg</code>	Burg	<code>pburg</code>
<code>spectrum.cov</code>	Covariance	<code>pcov</code>
<code>spectrum.eigenvector</code>	Eigenvector	<code>peig</code>
<code>spectrum.mcov</code>	Modified covariance	<code>pmcov</code>
<code>spectrum.mtm</code>	Thompson multitaper	<code>pmtm</code>
<code>spectrum.music</code>	Multiple Signal Classification	<code>pmusic</code>
<code>spectrum.periodogram</code>	Periodogram	<code>periodogram</code>
<code>spectrum.welch</code>	Welch	<code>pwelch</code>
<code>spectrum.yulear</code>	Yule-Walker	<code>pyulear</code>

For more information on each estimation method, use the syntax `help spectrum.estmethod` at the MATLAB prompt or refer to its reference page.

---

**Note** For estimation methods that use overlap and window length inputs, you specify the number of overlap samples as a percent overlap and you specify the segment length instead of the window length.

For estimation methods that use windows, if the window uses an additional parameter, a property is dynamically added to the spectrum object for that parameter. You can change that property using `set` (see [Changing Object Properties](#) on page 558).

---

## Methods

Methods provide ways of performing functions directly on your spectrum object without having to specify the spectral estimation parameters again. You can apply these methods directly on the variable you assigned to your spectrum object. For more information on any of these methods, use the syntax `help spectrum/method` at the MATLAB prompt.

# spectrum

Method	Options	Description
msspectrum	'onesided' 'twosided' Fs	<p>Note that the msspectrum method is only available for the periodogram and welch spectrum estimation objects.</p> <p>The mean-squared spectrum is intended for discrete spectra. Unlike the power spectral density (psd), the peaks in the mean-square spectrum reflect the power in the signal at a given frequency.</p> <p><code>Hmss = msspectrum(Hs, X)</code> returns a mean-square spectrum object containing the mean-square (power) estimate of the discrete-time signal <code>X</code> using the spectrum object <code>Hs</code>. Default for real <code>X</code> is the 'onesided' and for complex <code>X</code> is the 'twosided' Nyquist frequency range. <code>Hmss</code> contains a vector of normalized frequencies <code>W</code> at which the mean-square spectrum is estimated. For real signals, the range of <code>W</code> is <code>[0,pi]</code> if the number of FFT points (NFFT) is even and <code>[0,pi]</code> if NFFT is odd. For complex signals, the range of <code>W</code> is <code>[0,2pi]</code>.</p> <p><code>Hmss = msspectrum(Hs, X, 'Fs', Fs)</code> returns a mean-square spectrum object computed as a function of frequency, where <code>Fs</code> is the sampling frequency in Hz.</p>
		<p><code>Hmss = msspectrum(..., 'SpectrumType', 'twosided')</code> returns the two-sided mean-square spectrum. The spectrum length (NFFT) is computed over <code>[0,2pi]</code>, or if <code>Fs</code> is specified, <code>[0,Fs]</code>. Entering 'onesided' returns the one-sided mean-square spectrum, which contains the total signal power.</p> <p><code>Hmss = msspectrum(..., 'NFFT', nfft)</code> specifies the number of FFT points to use.</p>



Method	Options	Description
		<p><code>Hmss = msspectrum(..., 'Centerdc', true)</code> shifts the data and frequency values so that the DC component is at the center of the spectrum. The default value is <code>false</code>.</p> <p><code>msspectrum(...)</code> with no output arguments plots the mean-square spectrum in dB.</p>
<code>msspectrumopts</code>		<p><code>Hopts = msspectrumopts(Hs)</code> returns an object that contains options for the spectrum object <code>Hs</code>.</p> <p><code>Hopts = msspectrumopts(Hs, X)</code> returns an object with data-specific options and defaults.</p> <p>You can pass an <code>Hopts</code> options object as an argument to the <code>msspectrum</code> method. Any individual option you specify after the <code>Hopts</code> object overrides the value in <code>Hopts</code>. For example,</p> <p><code>Hmss = msspectrum(Hs, X, Hopts, 'SpectrumType', 'twosided')</code> overrides the <code>SpectrumType</code> value in <code>Hopts</code>.</p>
<code>psd</code>	<p>'onesided' 'twosided' Fs</p>	<p>Note that music and eigenvector spectrum objects do not support the <code>psd</code> method. See the <code>pseudospectrum</code> method below.</p> <p>The power spectral density (PSD) is intended for continuous spectra. The integral of the PSD over a given frequency band computes the average power in the signal over that frequency band. In contrast to the <code>msspectrum</code>, the peaks in this spectra do not reflect the power at a given frequency. See the <code>avgpower</code> method of <code>dspdata</code> for more information.</p> <p><code>Hpsd = psd(Hs, X)</code> returns a power spectral density object containing the power spectral density estimate of the discrete-time signal <code>X</code> using the spectrum object <code>Hs</code>. The PSD is the distribution of power per unit frequency. Default for real <code>X</code> is 'onesided' and for complex <code>X</code> is 'twosided'.</p>

# spectrum

Method	Options	Description
		<p>Hpsd contains a vector of normalized frequencies <math>W</math> at which the PSD is estimated. For real signals, the range of <math>W</math> is <math>[0,\pi]</math> if the number of FFT points (NFFT) is even and <math>[0,\pi)</math> if NFFT is odd. For complex signals, the range of <math>W</math> is <math>[0,2\pi)</math>.</p> <p><code>Hpsd = psd(Hs,X,'Fs',Fs)</code> returns a power spectral density object computed as a function of frequency, where <math>F_s</math> is the sampling frequency in Hz.</p> <p><code>Hpsd = psd(...,'SpectrumType','twosided')</code> returns the two-sided power spectral density of <math>X</math>. The spectrum length is NFFT and is computed over <math>[0,2\pi)</math> if <math>F_s</math> is not specified or <math>[0,F_s)</math> if <math>F_s</math> is specified. Entering 'onesided' returns the one-sided psd, which contains the total signal power.</p> <p><code>Hpsd = psd(...,'NFFT',nfft)</code> specifies the number of FFT points to use.</p> <p><code>Hpsd = psd(...,'Centerdc',true)</code> shifts the data and frequency values so that the DC component is at the center of the spectrum. The default value is false.</p> <p><code>psd(...)</code> with no output arguments plots the power spectral density estimate in dB per unit frequency.</p>

Method	Options	Description
psdopts		<p>Hopts = psdopts(Hs) returns an object that contains options for the spectrum object Hs. See dspopts, which is similar to psdopts, for an example.</p> <p>Hopts = psdopts(Hs,X) returns an object with data-specific options and defaults.</p> <p>You can pass an Hopts options object as an argument to the psd method. Any individual option you specify after the Hopts object overrides the value in Hopts. For example, Hpsd = psd(Hs,X,Hopts,'SpectrumType','twosided') overrides the SpectrumType value in Hopts.</p>
pseudospectrum	'half' 'whole' Fs	<p>Note that this method is used for only music or eigenvector spectrum objects.</p> <p>Hps = pseudospectrum(Hs,X) returns an object containing the pseudospectrum estimate of the discrete-time signal X using the spectrum object Hs. Hs must be a music or eigenvector object. Default for real X is 'half' and for complex X is the 'whole' Nyquist frequency range.</p> <p>Hps contains a vector of normalized frequencies W at which the pseudospectrum is estimated. For real signals, the range of W is [0,pi] if the number of FFT points (NFFT) is even and [0,pi] if NFFT is odd. For complex signals, the range of W is [0,2pi).</p>

# spectrum

Method	Options	Description
		<p><code>Hps = pseudospectrum(Hs,X,'Fs',Fs)</code> returns a pseudospectrum object computed as a function of frequency, where <code>Fs</code> is the sampling frequency in Hz.</p> <p><code>Hmps = pseudospectrum(...,'SpectrumRange','whole')</code> returns the pseudospectrum over the whole Nyquist range. The spectrum length is <code>NFFT</code> and is computed over <math>[0,2\pi)</math> if <code>Fs</code> is not specified or <math>[0,Fs)</math> if <code>Fs</code> is specified. Entering <code>'half'</code> returns the pseudospectrum calculated over half the Nyquist range.</p> <p><code>Hps = pseudospectrum(...,'NFFT',nfft)</code> specifies the number of FFT points to use</p> <p><code>Hps = pseudospectrum(...,'Centerdc',true)</code> shifts the data and frequency values so that the DC component is at the center of the spectrum. The default value is <code>false</code>.</p> <p><code>pseudospectrum(...)</code> with no output arguments plots the pseudospectrum in dB.</p>

Method	Options	Description
pseudospectrumopts		<p>Hopts = pseudospectrumopts(Hs) returns an object that contains options for the spectrum object Hs.</p> <p>Hopts = pseudospectrumopts(Hs,X) returns an object with data-specific options and defaults.</p> <p>You can pass an Hopts options object as an argument to the pseudospectrum method. Any individual option you specify after the Hopts object overrides the value in Hopts. For example, Hpseudospectrum= pseudospectrum(Hs,X, Hopts, 'SpectrumRange', 'whole') overrides the SpectrumRange value in Hopts.</p>
powerest	Fs	<p>Note that powerest is available only for music and eigenvector spectrum objects.</p> <p>POW = powerest(Hs,X) returns a vector POW containing estimates of the powers of the complex sinusoids in X. The input X can be a vector or a matrix. If it is a matrix it can be a data matrix, where <math>X^*X=R</math> or a correlation matrix <math>R</math>. The value the InputType property of Hs determines how X is interpreted. Hs must be a music or eigenvector spectrum object.</p> <p>[ POW,W]=powerest(Hs,X) returns POW and a vector W of the frequencies in rad/sample of the sinusoids in X.</p> <p>[ POW,F]=powerest(Hs,X,Fs) returns POW and a vector F of the frequencies in Hz of the sinusoids in X. Fs is the sampling frequency.</p>

### Viewing Object Properties

As with any object, you can use get to view a spectrum object's properties. To see a specific property, use

```
get(Hs, 'property')
```

where 'property' is the specific property name.

To see all properties for an object, use

```
get(Hs)
```

## Changing Object Properties

To set specific properties, use

```
set(Hs, 'property1', value, 'property2', value, ...)
```

where 'property1', 'property2', etc. are the specific property names.

To view the options for a property use set without specifying a value

```
set(Hs, 'property')
```

Note that you must use single quotation marks around the property name. For example, to change the order of a Burg spectrum object Hs to 6, use

```
set(Hs, 'order', 6)
```

Another example of using set to change an object's properties is this example of changing the dynamically created window property of a periodogram spectrum object.

```
Hs=spectrum.periodogram           %Create periodogram object
```

```
Hs =
```

```
    EstimationMethod: 'Periodogram'  
           FFTLength: 'NextPow2'  
           WindowName: 'Rectangular'
```

```
set(Hs, 'WindowName', 'Chebyshev') %Change window type
```

```
Hs                               %View changed object and  
                                %Note changed property
```

```
Hs =
```

```
    EstimationMethod: 'Periodogram'  
           FFTLength: 'NextPow2'  
           WindowName: 'Chebyshev'  
           SidelobeAtten: 100
```

```
set(Hs,'SidelobeAtten',150)    %Change dynamic property
Hs                             %View changed object
```

```
Hs =
```

```
    EstimationMethod: 'Periodogram'
           FFTLength: 'NextPow2'
           WindowName: 'Chebyshev'
           SidelobeAtten: 150
```

All spectrum object properties can be changed using the set command, except for the EstimationMethod property.

Another way to change an object's properties is by using the inspect command which opens the **Property Inspector** window where you can edit any property, except dynamic properties, such as those used with windows.

```
inspect(Hs)
```

### Copying an Object

To create a copy of an object, use the copy method.

```
H2 = copy(Hs)
```

---

**Note** Using the syntax H2 = Hs copies only the object handle and does not create a new object.

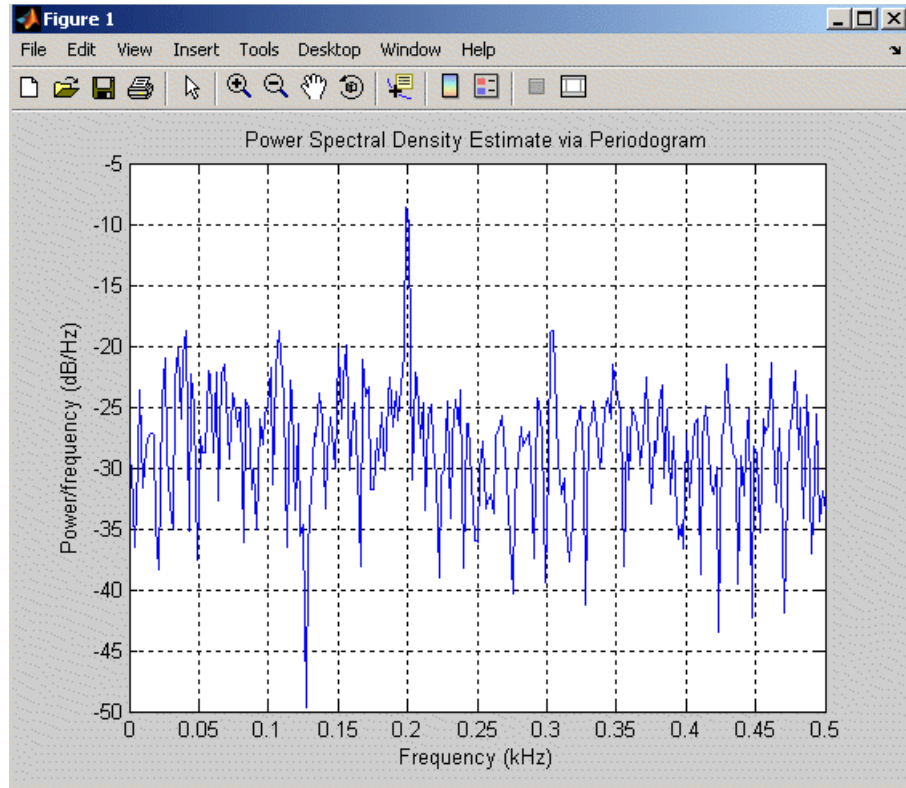
---

### Examples

Define a cosine of 200 Hz, add some noise and then view its power spectral density estimate generated with the periodogram algorithm.

```
Fs = 1000;
t = 0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
Hs=spectrum.periodogram;
psd(Hs,x,'Fs',Fs)
```

# spectrum



Refer to the reference pages for each estimation method for more examples.

## See Also

`dspdata`, `dsptops`, `spectrum.burg`, `spectrum.cov`, `spectrum.mcov`,  
`spectrum.yulear`, `spectrum.periodogram`, `spectrum.welch`, `spectrum.mtm`,  
`spectrum.eigenvector`, `spectrum.music`



**Purpose** Burg spectrum

**Syntax**  
`Hs = spectrum.burg`  
`Hs = spectrum.burg(order)`  
`Hs = spectrum.burg(order, FFTLength)`

**Description** `Hs = spectrum.burg` returns a default Burg spectrum object, `Hs`, that defines the parameters for the Burg parametric spectral estimation algorithm. The Burg algorithm estimates the spectral content by fitting an auto-regressive (AR) linear prediction filter model of a given order to the signal.

`Hs = spectrum.burg(order)` returns a spectrum object, `Hs` with the specified order and the `FFTLength` determined using `NextPow2`. The default value for order is 4.

`Hs = spectrum.burg(order, FFTLength)` returns a spectrum object, `Hs` with the specified order of the AR model and the specified way of determining the `FFTLength`. Valid values of the `FFTLength` string are:

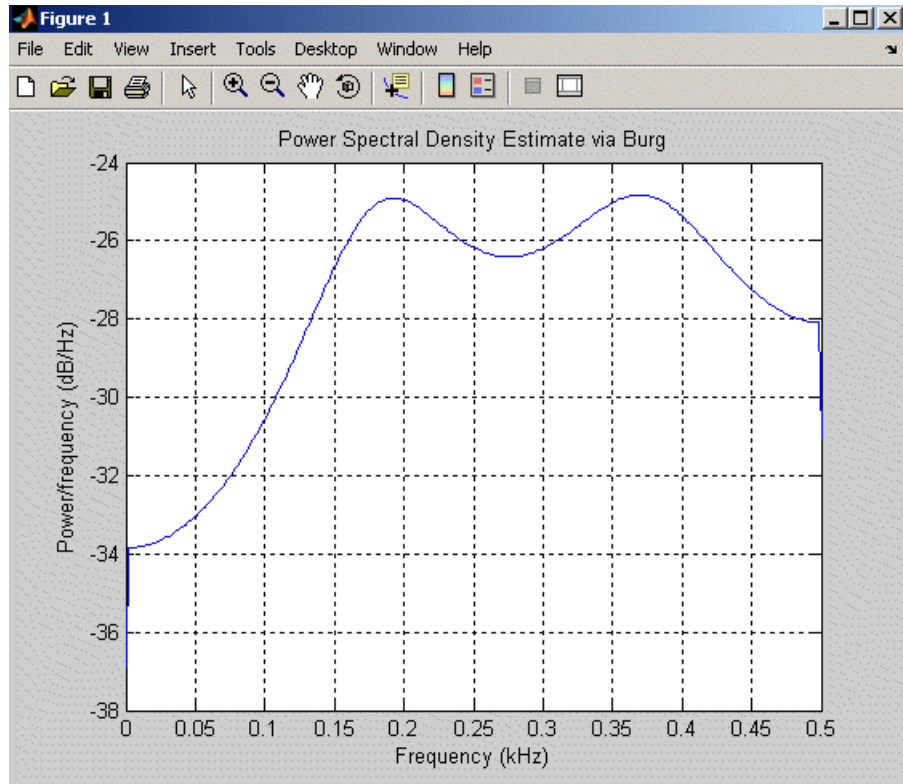
FFTLength string	Description
'InputLength'	Use the length of the input signal as the FFT length
'NextPow2'	Use the next power of 2 greater than the input signal length as the FFT length. This is the default value.
'UserDefined'	Use the FFT length provided as an input to the <code>psd</code> method or via a <code>dsports</code> object. See <code>dsports</code> for an example.

**Note** See `pburg` for more information on the Burg algorithm.

**Examples** Define a fourth order auto-regressive model and view its power spectral density using the Burg algorithm.

# spectrum.burg

```
randn('state',1);  
x=randn(100,1);  
x=filter(1,[1 1/2 1/3 1/4 1/5],x);%Fourth order AR filter  
Hs=spectrum.burg;                    %Fourth order AR model  
psd(Hs,x,'NFFT',512)
```



## See Also

`dspdata`, `dsplots`, `spectrum`, `spectrum.cov`, `spectrum.mcov`,  
`spectrum.yulear`, `spectrum.periodogram`, `spectrum.welch`, `spectrum.mtm`,  
`spectrum.eigenvector`, `spectrum.music`

**Purpose** Covariance spectrum

**Syntax**  
 Hs = spectrum.cov  
 Hs = spectrum.cov(order)  
 Hs = spectrum.cov(order,FFTLength)

**Description** Hs = spectrum.cov returns a default covariance spectrum object, Hs, that defines the parameters for the covariance spectral estimation algorithm. The covariance algorithm estimates the spectral content by fitting an auto-regressive (AR) linear prediction model of a given order to the signal.

Hs = spectrum.cov(order) returns a spectrum object, Hs with the specified order and the *FFTLength* determined using NextPow2. The default value for order is 4.

Hs = spectrum.cov(order,FFTLength) returns a covariance spectrum object, Hs with the order of the covariance model and the specified way of determining the FFTLength. Valid values of the FFTLength string are:

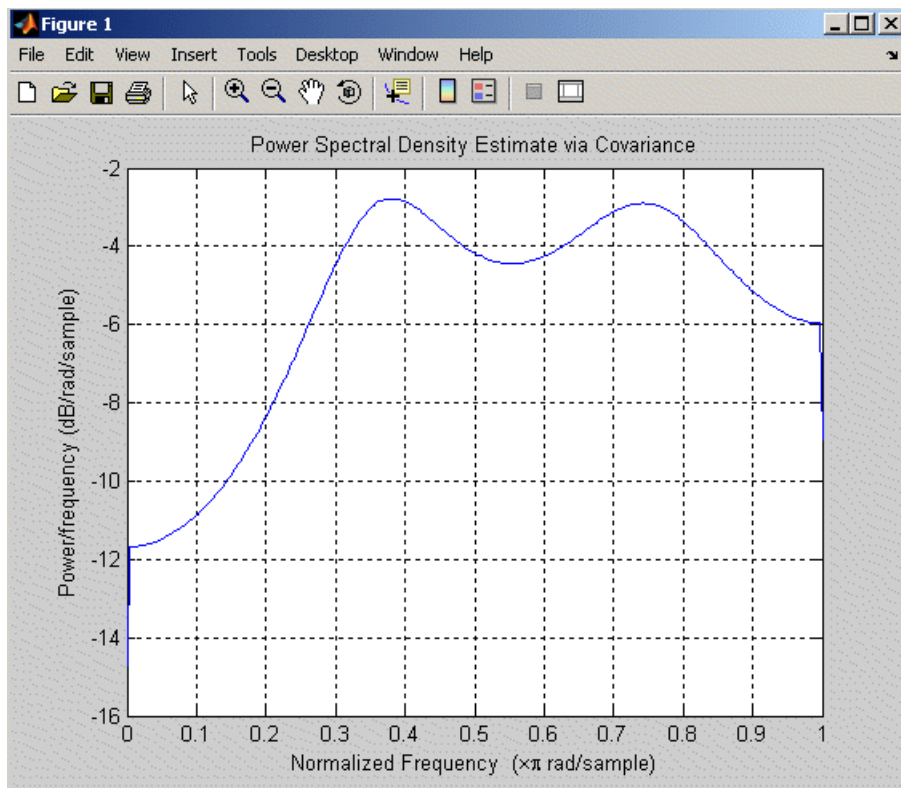
FFTLength string	Description
'InputLength'	Use the length of the input signal as the FFT length
'NextPow2'	Use the next power of 2 greater than the input signal length as the FFT length. This is the default value.
'UserDefined'	Use the FFT length provided as an input to the psd method or via a dspopts object. See dspopts for an example.

**Note** See pcov for more information on the covariance algorithm.

## Examples

Define a fourth order auto-regressive model and view its power spectral density using the covariance algorithm.

```
randn('state',1);  
x=randn(100,1);  
x=filter(1,[1 1/2 1/3 1/4 1/5],x);%Fourth order AR filter  
Hs=spectrum.cov; %Fourth order AR model  
psd(Hs,x,'NFFT',512)
```



## See Also

`dspdata`, `dsptopts`, `spectrum`, `spectrum.burg`, `spectrum.mcov`,  
`spectrum.yulear`, `spectrum.periodogram`, `spectrum.welch`, `spectrum.mtm`,  
`spectrum.eigenvector`, `spectrum.music`

**Purpose** Eigenvector spectrum

**Syntax**

```
Hs = spectrum.eigenvector
Hs = spectrum.eigenvector(NSinusoids)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength)
Hs = spectrum.eigenvector(...,OverlapPercent)
Hs = spectrum.eigenvector(...,WindowName)
Hs = spectrum.eigenvector(...,SubspaceThreshold)
Hs = spectrum.eigenvector(...,FFTLength)
Hs = spectrum.eigenvector(...,InputType)
```

**Description** Hs = spectrum.eigenvector returns a default eigenvector spectrum object, Hs, that defines the parameters for an eigenanalysis spectral estimation method. This object uses the following default values:

Property Name	Default Value	Description
NSinusoids	2	Number of complex sinusoids
SegmentLength	4	Segment length
OverlapPercent	50	Percent overlap between segments

# spectrum.eigenvector

Property Name	Default Value	Description
WindowName	'Rectangular'	<p>Window name string or 'User Defined' (see window for valid window names). For more information on each window, refer to its reference page.</p> <p>This argument can also be a cell array containing the window name string or 'User Defined' and, if used for the particular window, an optional parameter value. The syntax is {wname , wparam}.</p> <p>You can use set to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window (see spectrum for information on using set).</p>
SubspaceThreshold	0	<p>Threshold is the cutoff for signal and noise separation. The threshold is multiplied by <math>\lambda_{\min}</math>, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold (<math>\lambda_{\min} * \text{threshold}</math>) are assigned to the noise subspace.</p>

Property Name	Default Value	Description
FFTlength	'NextPow2'	String defining how the number of FFT points is determined. The default is the next power of 2 that is greater than the input length. Other valid values are 'InputLength' and 'UserDefined'. InputLength uses the length of the input signal as the FFT length. UserDefined uses the value provided via the pseudospectrum method or a dsports object. See dsports for an example.
InputType	'Vector'	Type of input that will be used with this spectrum object. Valid values are 'Vector', 'DataMatrix' and 'CorrelationMatrix'.

`Hs = spectrum.eigenvector(NSinusoids)` returns a spectrum object, `Hs`, with the specified number of sinusoids and default values for all other properties. Refer to the table above for default values.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength)` returns a spectrum object, `Hs`, with the specified segment length.

`Hs = spectrum.eigenvector(...,OverlapPercent)` returns a spectrum object, `Hs`, with the specified overlap between segments.

`Hs = spectrum.eigenvector(...,WindowName)` returns a spectrum object, `Hs`, with the specified window.

# spectrum.eigenvector

---

---

**Note** Window names must be enclosed in single quotes, such as `spectrum.eigenvector(3,32,50,'chebyshev')` or `spectrum.eigenvector(3,32,50,{'chebyshev',60})`

---

`Hs = spectrum.eigenvector(...,SubspaceThreshold)` returns a spectrum object, `Hs`, with the specified subspace threshold.

`Hs = spectrum.eigenvector(...,FFTLength)` returns a spectrum object, `Hs`, with the specified way of the determining the FFT length.

`Hs = spectrum.eigenvector(...,InputType)` returns a spectrum object, `Hs`, with the specified input type.

---

**Note** See `peig` for more information on the eigenanalysis algorithm.

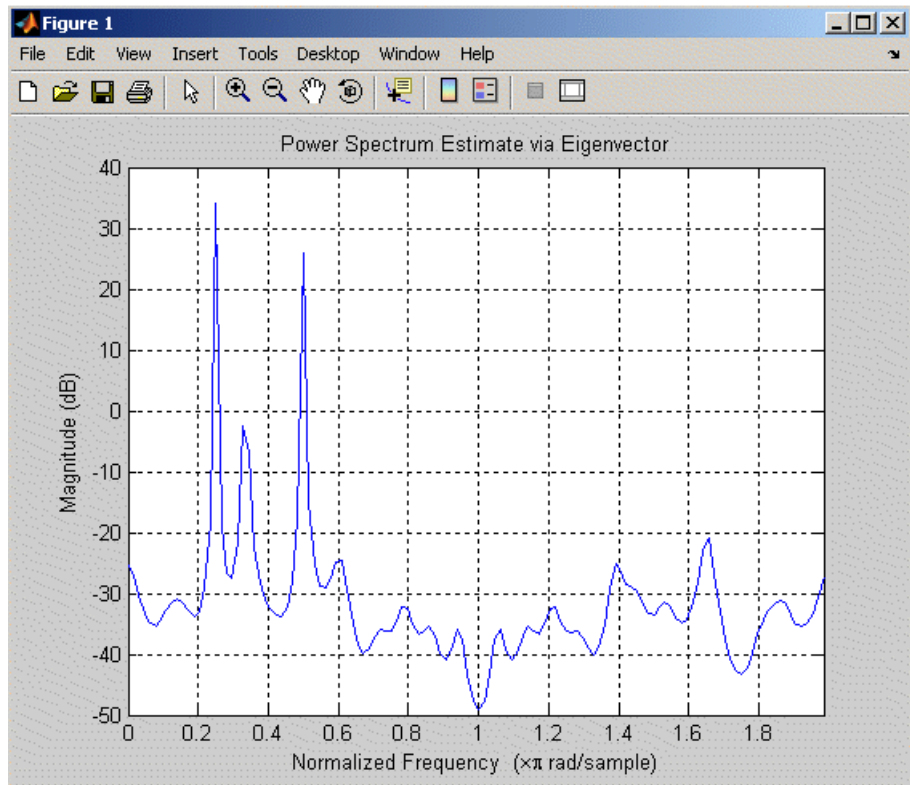
---

## Examples

Define a complex signal with three sinusoids, add noise, and view its pseudospectrum using eigenanalysis. Set the FFT length to 128.

```
randn('state',1);
n=0:99;
s=exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);
Hs=spectrum.eigenvector(3,32,95,'rectangular',5);
pseudospectrum(Hs,s,'NFFT',128)
```





## See Also

`dspdata`, `dsplots`, `spectrum`, `spectrum.music`, `spectrum.burg`, `spectrum.cov`, `spectrum.mcov`, `spectrum.yulear`, `spectrum.periodogram`, `spectrum.welch`, `spectrum.mtm`

# spectrum.mcov

---

**Purpose** Modified covariance spectrum

**Syntax**  
Hs = spectrum.mcov  
Hs = spectrum.mcov(order)  
Hs = spectrum.mcov(order,FFTLength)

**Description** Hs = spectrum.mcov returns a default modified covariance spectrum object, Hs, that defines the parameters for the modified covariance spectral estimation algorithm. The modified covariance algorithm estimates the spectral content by fitting an auto-regressive (AR) linear prediction filter model of a given order to the signal.

Hs = spectrum.mcov(order) returns a spectrum object, Hs with the specified order and the FFTLength determined using NextPow2. The default value for order is 4.

Hs = spectrum.mcov(order,FFTLength) returns a spectrum object, Hs with specified order and the specified way of determining the FFTLength. Valid values of the FFTLength string are as follows:

FFTLength string	Description
'InputLength'	Use the length of the input signal as the FFT length
'NextPow2'	Use the next power of 2 greater than the input signal length as the FFT length. This is the default value.
'UserDefined'	Use the FFT length provided as an input to the psd method or via a dspopts object. See dspopts for an example.

---

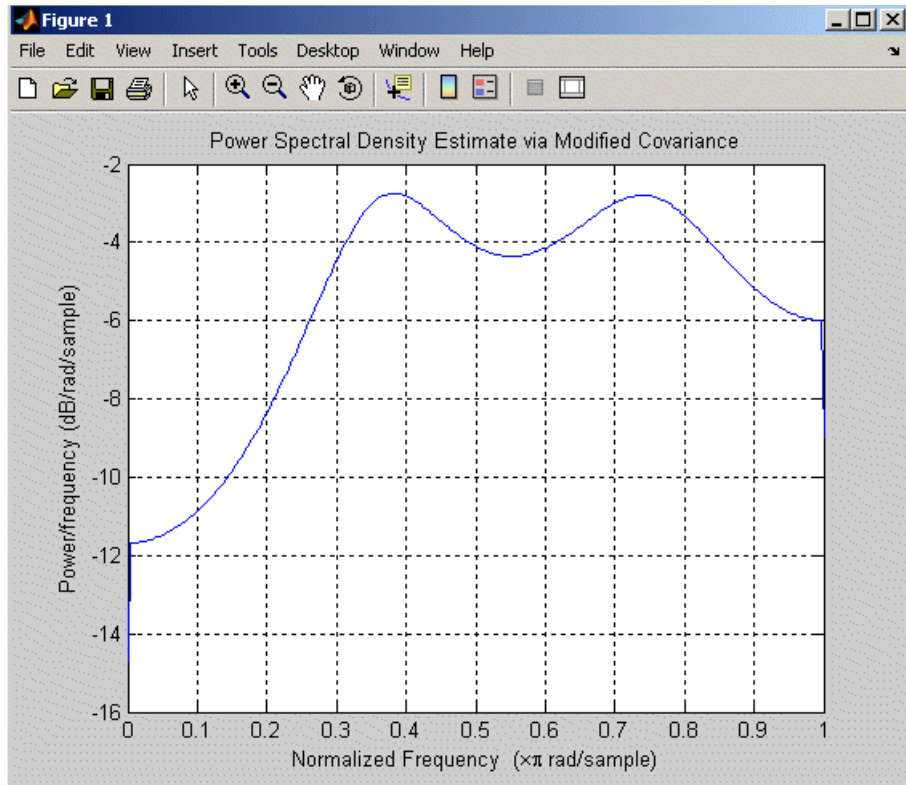
**Note** See pmcov for more information on the modified covariance algorithm.

---

**Examples**

Define a fourth order auto-regressive model and view its power spectral density using the modified covariance algorithm.

```
randn('state',1);
x=randn(100,1);
x=filter(1,[1 1/2 1/3 1/4 1/5],x);%Fourth order AR filter
Hs=spectrum.mcov;                %Fourth order AR model
psd(Hs,x,'NFFT',512)
```

**See Also**

dspdata, dspopts, spectrum, spectrum.burg, spectrum.cov, spectrum.yulear, spectrum.periodogram, spectrum.welch, spectrum.mtm, spectrum.eigenvector, spectrum.music

# spectrum.mtm

---

**Purpose** Thompson multitaper spectrum

**Syntax**

```
Hs = spectrum.mtm
Hs = spectrum.mtm(TimeBW)
Hs = spectrum.mtm(DPSS,Concentrations)
Hs = spectrum.mtm(...,CombineMethod)
Hs = spectrum.mtm(...,FFTLenght)
```

**Description** Hs = spectrum.mtm returns a default Thompson multitaper spectrum object, Hs that defines the parameters for the Thompson multitaper spectral estimation algorithm, which uses a linear or nonlinear combination of modified periodograms. The periodograms are computed using a sequence of orthogonal tapers (windows in the frequency domain) specified from discrete prolate spheroidal sequences (dpss). This object uses the following default values:

Property Name	Default Value	Description
TimeBW	4	Product of time and bandwidth for the discrete prolate spheroidal sequences (or Slepian sequences) used as data windows

Property Name	Default Value	Description
CombineMethod	'adaptive'	Algorithm for combining the individual spectral estimates. Valid values are 'adaptive'—adaptive (nonlinear) 'unity'—unity weights (linear) 'eigenvector'—Eigenvalue weights (linear)
FFTLength	'NextPow2'	String defining how the number of FFT points is determined. The default is the next power of 2 that is greater than the input length. Other valid values are: 'InputLength' and 'UserDefined'. InputLength uses the length of the input signal as the FFT length. UserDefined uses the value provided via a psd method or dspopts object. See dspopts for an example.

`Hs = spectrum.mtm(TimeBW)` returns a spectrum object, `Hs` with the specified time-bandwidth product.

`Hs = spectrum.mtm(DPSS,Concentrations)` returns a spectrum object, `Hs` with the specified dpss data tapers and their concentrations.

---

**Note** You can either specify the time-bandwidth product (`TimeBW`) or the DPSS data tapers and their `Concentrations`. See `dpss` and `pmtm` for more information.

---

`Hs = spectrum.mtm(...,CombineMethod)` returns a spectrum object, `Hs`, with the specified method for combining the spectral estimates. Refer to the table above for valid `CombineMethod` values.

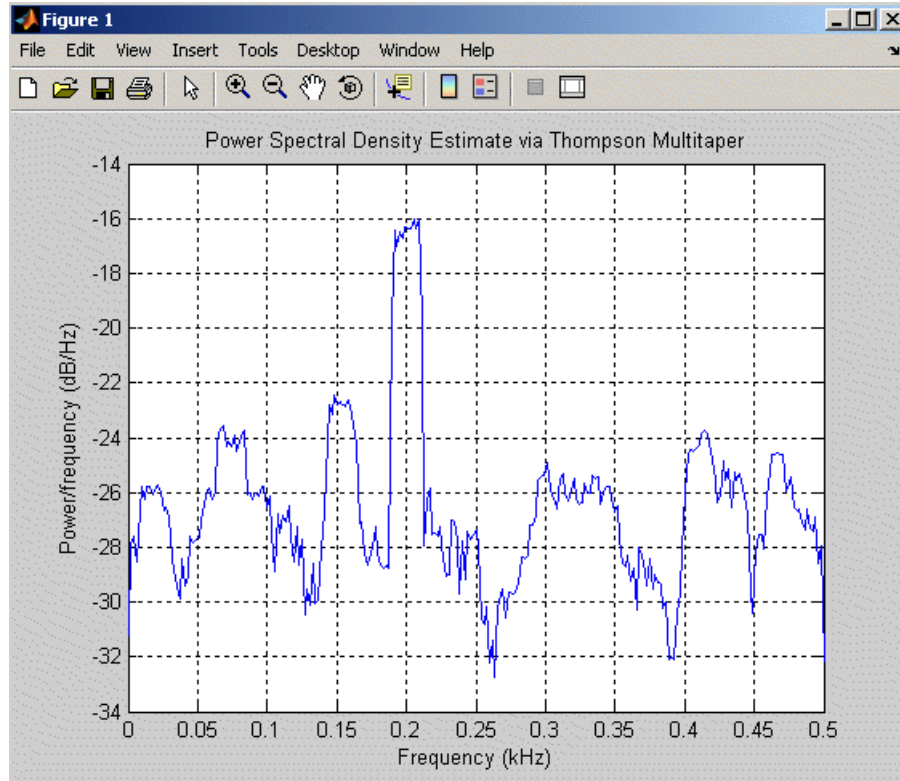
# spectrum.mtm

`Hs = spectrum.mtm(...,FFTLength)` returns a spectrum object, `Hs` with the specified way of determining the `FFTLength`. Refer to the table above for valid `FFTLength` values.

## Examples

Define a cosine of 200 Hz, add noise and view its power spectral density using the Thompson multitaper algorithm with a time-bandwidth product of 3.5.

```
Fs=1000;  
t=0:1/Fs:.3;  
x=cos(2*pi*t*200)+randn(size(t));  
Hs=spectrum.mtm(3.5);  
psd(Hs,x,'Fs',Fs)
```



The above example could be done by specifying the data tapers and concentrations instead of the time-bandwidth product.

```
Fs=1000;  
t=0:1/Fs:.3;  
x=cos(2*pi*t*200)+randn(size(t));  
[e,v]=dpss(length(x),3.5);  
Hs=spectrum.mtm(e,v);  
psd(Hs,x,'Fs',Fs)
```

**See Also**

`dspdata`, `dspspts`, `spectrum`, `spectrum.periodogram`, `spectrum.welch`,  
`spectrum.burg`, `spectrum.cov`, `spectrum.mcov`, `spectrum.yulear`,  
`spectrum.eigenvector`, `spectrum.music`

# spectrum.music

---

**Purpose** Multiple signal classification spectrum

**Syntax**

```
Hs = spectrum.music
Hs = spectrum.music(NSinusoids)
Hs = spectrum.music(NSinusoids,SegmentLength)
Hs = spectrum.music(...,OverlapPercent)
Hs = spectrum.music(...,WindowName)
Hs = spectrum.music(...,SubspaceThreshold)
Hs = spectrum.music(...,FFTLength)
Hs = spectrum.music(...,InputType)
```

**Description** Hs = spectrum.music returns a default multiple signal classification (MUSIC) spectrum object, Hs, that defines the parameters for the MUSIC spectral estimation algorithm, which uses Schmidt's eigenspace analysis algorithm. This object uses the following default values.

Property Name	Default Value	Description
NSinusoids	2	Number of complex sinusoids
SegmentLength	4	Segment length
OverlapPercent	50	Percent overlap between segments



Property Name	Default Value	Description
WindowName	'Rectangular'	<p>Window name string or 'User Defined' (see window for valid window names). For more information on each window, refer to its reference page).</p> <p>This argument can also be a cell array containing the window name string or 'User Defined' and, if used for the particular window, an optional parameter value. The syntax is {wname, wparam}.</p> <p>You can use set to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window (see spectrum for information on using set).</p>
SubspaceThreshold	0	<p>Threshold is the cutoff for signal and noise separation. The threshold is multiplied by <math>\lambda_{\min}</math>, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold (<math>\lambda_{\min} * \text{threshold}</math>) are assigned to the noise subspace.</p>

Property Name	Default Value	Description
FFTLength	'NextPow2'	String defining how the number of FFT points is determined. The default is the next power of 2 that is greater than the input length. Other valid values are 'InputLength' and 'UserDefined'. InputLength uses the length of the input signal as the FFT length. UserDefined uses the value provided via a pseudospectrum method or dspopts object. See dspopts for an example.
InputType	'Vector'	Type of input that will be used with this spectrum object. Valid values are 'Vector', 'DataMatrix' and 'CorrelationMatrix'.

`Hs = spectrum.music(NSinusoids)` returns a spectrum object, `Hs`, with the specified number of sinusoids and default values for all other properties. Refer to the table above for default values.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength)` returns a spectrum object, `Hs`, with the specified segment length.

`Hs = spectrum.music(...,OverlapPercent)` returns a spectrum object, `Hs`, with the specified overlap between segments.

`Hs = spectrum.music(...,WindowName)` returns a spectrum object, `Hs`, with the specified window.

---

**Note** Window names must be enclosed in single quotes, such as `spectrum.music(3,32,50,'chebyshev')` or `spectrum.music(3,32,50,{'chebyshev',60})`

---

`Hs = spectrum.music(...,SubspaceThreshold)` returns a spectrum object, `Hs`, with the specified subspace threshold.

`Hs = spectrum.music(...,FFTLenght)` returns a spectrum object, `Hs`, with the specified FFT length type.

`Hs = spectrum.music(...,InputType)` returns a spectrum object, `Hs`, with the specified input type.

---

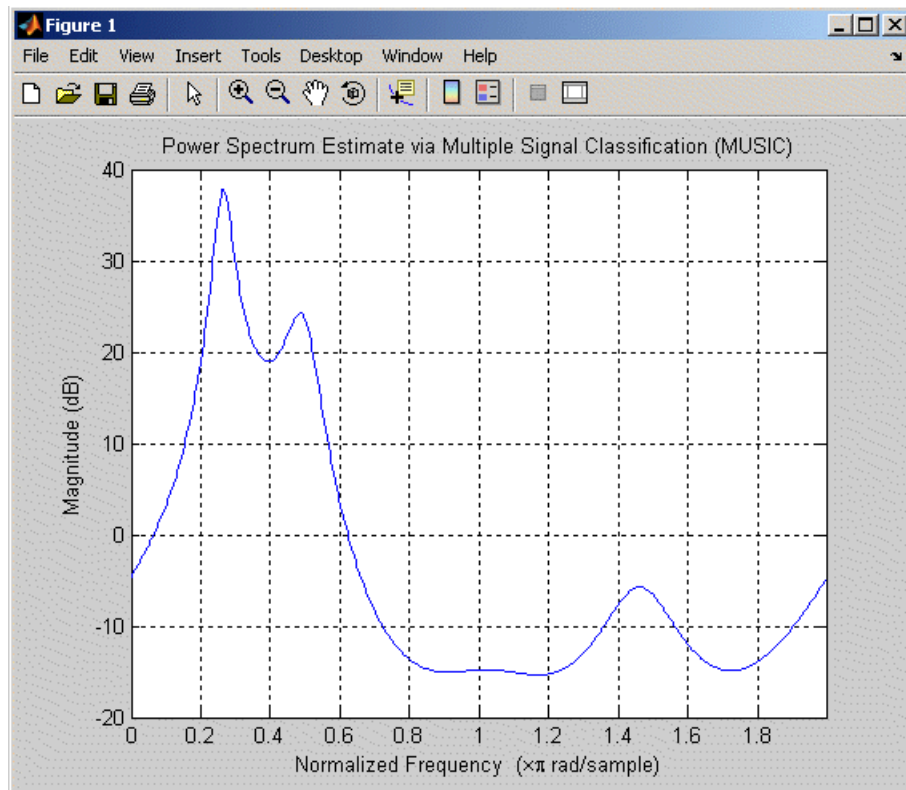
**Note** See `pmusic` for more information on the MUSIC algorithm.

---

## Examples

Define a complex signal with three sinusoids, add noise, and estimate its pseudospectrum using the MUSIC algorithm.

```
randn('state',1);
n=0:99;
s=exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);
Hs=spectrum.music(3);
pseudospectrum(Hs,s,'NFFT',512)
```



## See Also

`dspdata`, `dsplots`, `spectrum`, `spectrum.eigenvector`, `spectrum.burg`, `spectrum.cov`, `spectrum.mcov`, `spectrum.yulear`, `spectrum.periodogram`, `spectrum.welch`, `spectrum.mtm`

**Purpose** Periodogram spectrum

**Syntax**

```
Hs = spectrum.periodogram  
Hs = spectrum.periodogram(winname)  
Hs = spectrum.periodogram({winname,winparameter})  
Hs = spectrum.periodogram(...,FFTLength)
```

**Description** `Hs = spectrum.periodogram` returns a default periodogram spectrum object, `Hs`, that defines the parameters for the periodogram spectral estimation method. This default object uses a rectangular window and a default FFT length equal to the next power of 2 (`NextPow2`) that is greater than the input length.

`Hs = spectrum.periodogram(winname)` returns a spectrum object, `Hs`, that uses the specified window. If the window uses an optional associated window parameter, it is set to the default value. This object uses the default FFT length.

`Hs = spectrum.periodogram({winname,winparameter})` returns a spectrum object, `Hs`, that uses the specified window and optional associated window parameter, if any. You specify the window and window parameter in a cell array with a windowname string and the parameter value. This object uses the default FFT length.

Valid windowname strings are any valid window in the Signal Processing Toolbox or a user-defined window. Refer to the corresponding window function page for window parameter information.

You can use `set` to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window (see `spectrum` for information on using `set`).

---

**Note** Window names must be enclosed in single quotes, such as `spectrum.periodogram('tukey')` or `spectrum.periodogram({'tukey',0.7})`

---

## spectrum.periodogram

---

Hs = spectrum.periodogram(...,FFTLength) returns a spectrum object, Hs that uses the specified way of determining the FFTLength. Valid values of the FFTLength string are as follows.

FFTLength string	Description
'InputLength'	Use the length of the input signal as the FFT length
'NextPow2'	Use the next power of 2 greater than the input signal length as the FFT length. This is the default value.
'UserDefined'	Use the FFT length provided as an input to the psd method or via a dspopts object. See dspopts for an example.

---

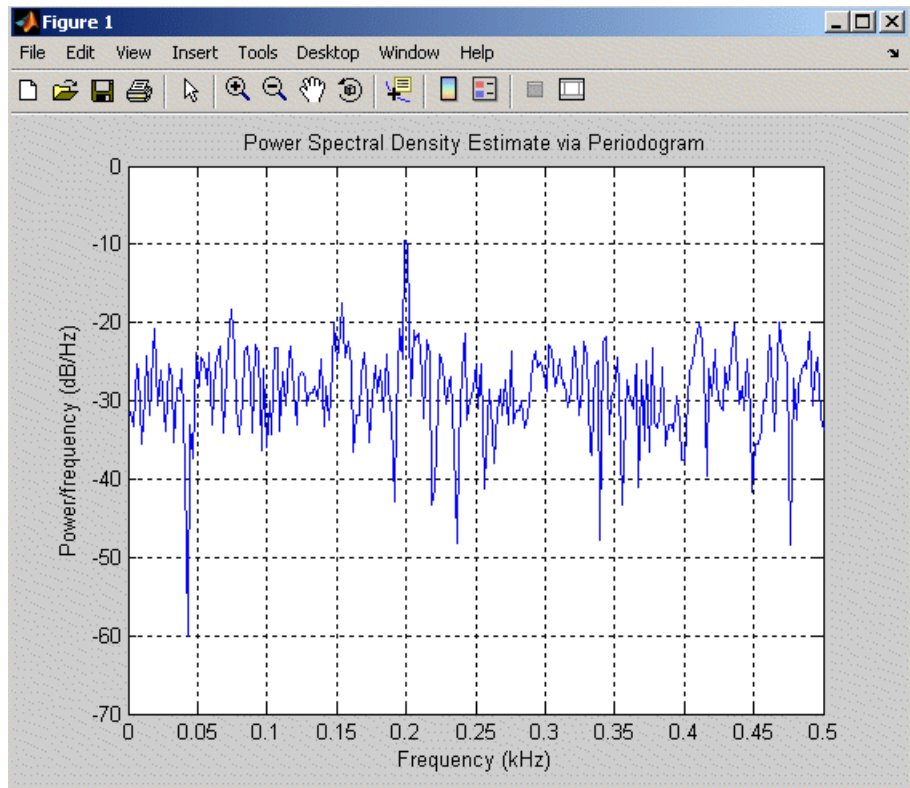
**Note** See periodogram for more information on the periodogram algorithm.

---

### Examples

Define a cosine of 200 Hz, add noise and view its spectral content using the periodogram spectral estimation technique.

```
Fs=1000;  
t=0:1/Fs:.3;  
x=cos(2*pi*t*200)+randn(size(t));  
Hs=spectrum.periodogram;           % Use default values  
psd(Hs,x,'Fs',Fs)
```



## See Also

`dspdata`, `dsplots`, `spectrum`, `spectrum.welch`, `spectrum.mtm`, `spectrum.burg`, `spectrum.cov`, `spectrum.mcov`, `spectrum.yulear`, `spectrum.eigenvector`, `spectrum.music`

# spectrum.welch

---

**Purpose** Welch spectrum

**Syntax**

```
Hs = spectrum.welch  
Hs = spectrum.welch(WindowName)  
Hs = spectrum.welch({WindowName,winparam})  
Hs = spectrum.welch(...,SegmentLength)  
Hs = spectrum.welch(...,OverlapPercent)  
Hs = spectrum.welch(...,FFTLength)
```

**Description** Hs = spectrum.welch returns a default Welch spectrum object, Hs, that defines the parameters for Welch's averaged, modified periodogram spectral estimation method. The object uses these default values.

Property Name	Default Value	Description
WindowName	'Hamming', SamplingFlag: symmetric	Window name string or 'User Defined' (see window for valid window names). If the window uses an optional property, it is set to the default value.
{WindowName, winparam}  Cell array containing WindowName and optional window parameter	'Hamming', SamplingFlag: symmetric	Cell array containing the window name string or 'User Defined' and, if used for the particular window, an optional parameter value. (See window for valid window names and for more information on each window, refer to its reference page.)  You can use set to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window. (See spectrum for information on using set.)



Property Name	Default Value	Description
SegmentLength	64	Segment length
OverlapPercent	50%	Percent overlap between segments
FFTLenght	'NextPow2'	String defining how the number of FFT points is determined. The default is the next power of 2 that is greater than the input length. Other valid values are 'InputLength' and 'UserDefined'. InputLength uses the length of the input signal as the FFT length. UserDefined uses the value provided via the psd method or dspopts object. See dspopts for an example.

`Hs = spectrum.welch(WindowName)` returns a spectrum object, `Hs`, using Welch's method with the specified window and the default values for all other parameters

**Note** Window names must be enclosed in single quotes, such as `spectrum.welch('chebyshev',32,50)` or `spectrum.music({'chebyshev',60},32,50)`

`Hs = spectrum.welch({WindowName,winparam})` returns a spectrum object, `Hs` with the specified window and associated parameter.

`Hs = spectrum.welch(...,SegmentLength)` returns a spectrum object, `Hs` with the specified segment length.

`Hs = spectrum.welch(...,OverlapPercent)` returns a spectrum object, `Hs` with the specified percentage overlap between segments.

# spectrum.welch

---

`Hs = spectrum.welch(...,FFTLength)` returns a spectrum object, `Hs` with the specified FFT length type.

---

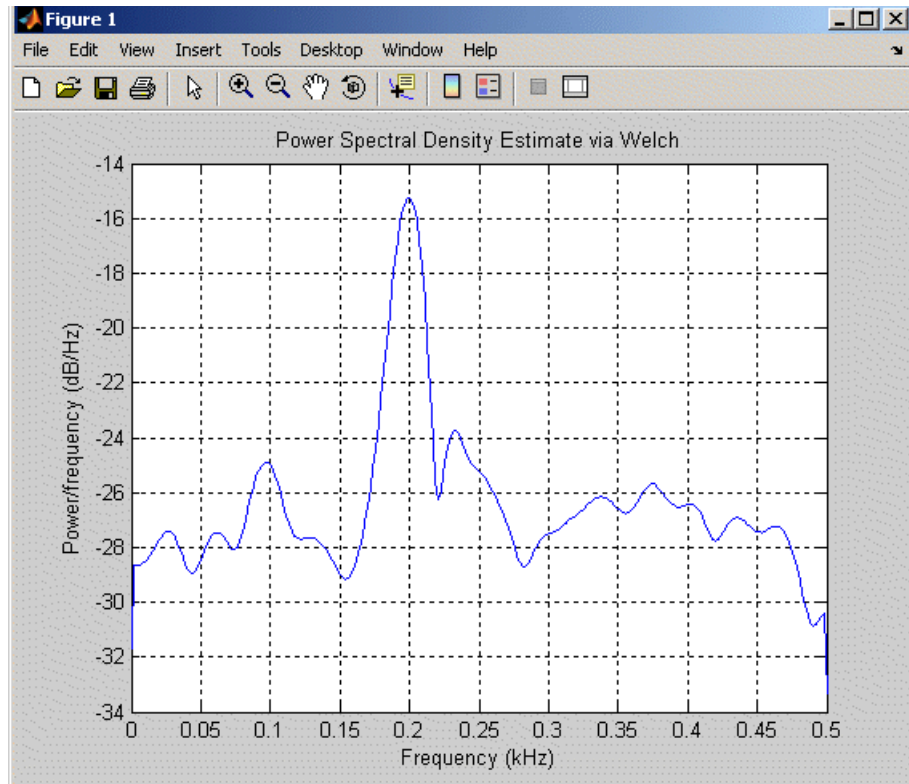
**Note** See `pwelch` for more information on the Welch algorithm.

---

## Examples

Define a cosine of 200 Hz, add noise and view its spectral content using the Welch algorithm.

```
Fs=1000;
t=0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
Hs=spectrum.welch;
psd(Hs,x,'Fs',Fs)
```

**See Also**

`dspdata`, `dspopts`, `spectrum`, `spectrum.periodogram`, `spectrum.mtm`,  
`spectrum.burg`, `spectrum.cov`, `spectrum.mcov`, `spectrum.yulear`,  
`spectrum.eigenvector`, `spectrum.music`

# spectrum.yulear

---

**Purpose** Yule-Walker spectrum object

**Syntax**  
Hs = spectrum.yulear  
Hs = spectrum.yulear(order)  
Hs = spectrum.yulear(order, FFTLength)

**Description** Hs = spectrum.yulear returns a default Yule-Walker spectrum object, Hs, that defines the parameters for the Yule-Walker spectral estimation algorithm. This method is also called the auto-correlation or windowed method. The Yule-Walker algorithm estimates the spectral content by fitting an auto-regressive (AR) linear prediction filter model of a given order to the signal. This leads to a set of Yule-Walker equations, which are solved using Levinson-Durbin recursion.

Hs = spectrum.yulear(order) returns a spectrum object, Hs, with the specified order and the FFTLength determined using NextPow2. The default value for order is 4.

Hs = spectrum.yulear(order, FFTLength) returns a spectrum object, Hs, with the specified order of the AR model and the specified way of determining the FFTLength. Valid values of the FFTLength string are as follows.

FFTLength string	Description
'InputLength'	Use the length of the input signal as the FFT length
'NextPow2'	Use the next power of 2 greater than the input signal length as the FFT length. This is the default value.
'UserDefined'	Use the FFT length provided as an input to the psd method or via a dspopts object. See dspopts for an example.

---

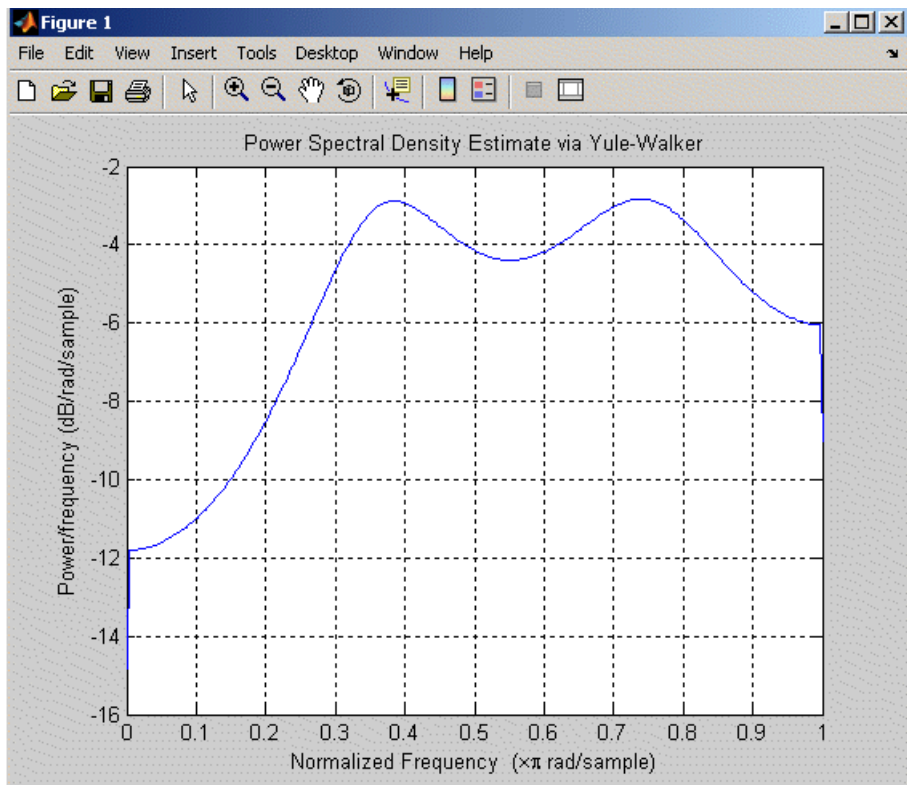
**Note** See pyulear for more information on the Yule-Walker algorithm.

---

**Examples**

Define a fourth order auto-regressive model and view its spectral content using the Yule-Walker algorithm.

```
randn('state',1);
x=randn(100,1);
x=filter(1,[1 1/2 1/3 1/4 1/5],x);%Fourth order AR filter
Hs=spectrum.yulear;                %Fourth order AR model
psd(Hs,x,'NFFT',512)
```

**See Also**

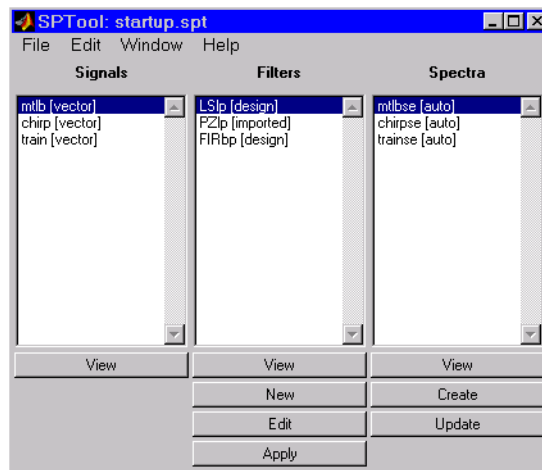
dspdata, dspopts, spectrum, spectrum.burg, spectrum.cov, spectrum.mcov, spectrum.periodogram, spectrum.welch, spectrum.mtm, spectrum.eigenvector, spectrum.music

# sptool

**Purpose** Open interactive digital signal processing tool

**Syntax** sptool

**Description** sptool opens SPTool, a graphical user interface (GUI) that manages a suite of four other GUIs: Signal Browser, FDATool, FVTool, and Spectrum Viewer. These GUIs provide access to many of the signal, filter, and spectral analysis functions in the toolbox. When you type sptool at the command line, the SPTool GUI opens.



Using SPTool you can

- Analyze signals listed in the **Signals** list box with the Signal Browser
- Design or edit filters with FDATool
- Analyze filter responses for filters listed in the **Filters** list box with FVTool
- Apply filters in the **Filters** list box to signals in the **Signals** list box
- Create and analyze signal spectra with the Spectrum Viewer
- Print the Signal Browser, FDATool responses, and Spectrum Viewer

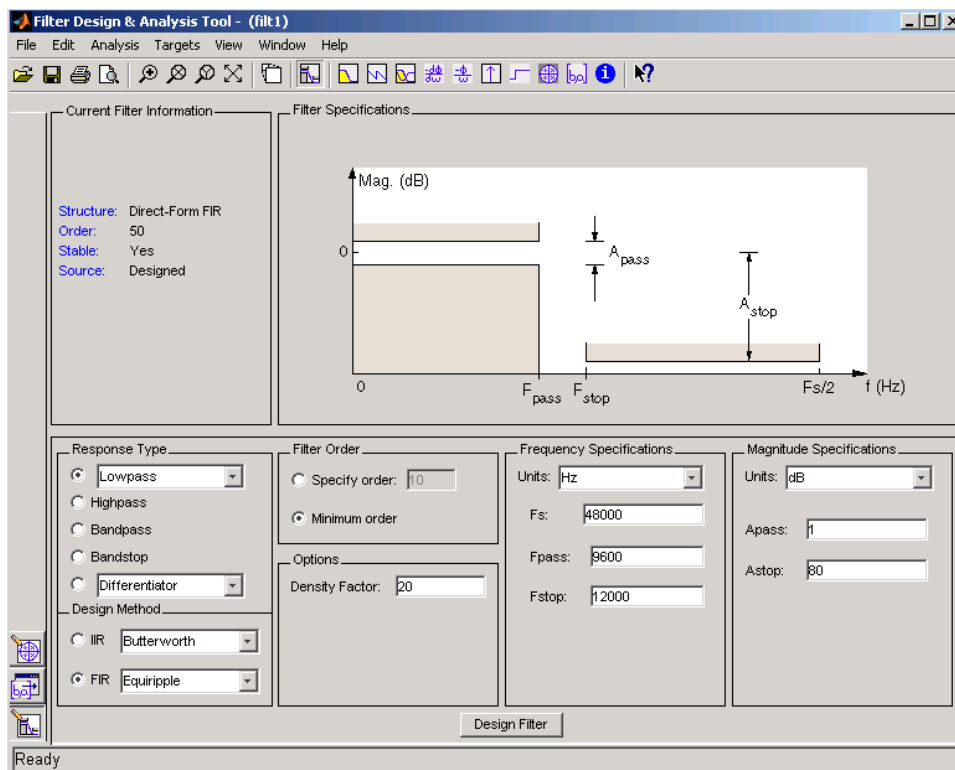
You can activate the four integrated signal processing GUIs from SPTool.

## Signal Browser

The Signal Browser allows you to view, measure, and analyze the time-domain information of one or more signals. To activate the Signal Browser, press the **View** button under the **Signals** list box in SPTool.

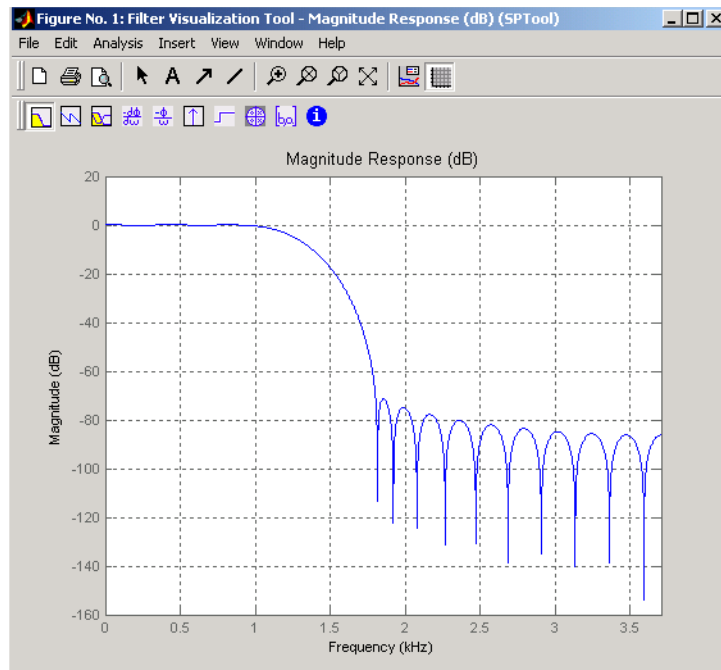
## FDATool

FDATool allows you to design and edit filters of various lengths and types. See `fdatool` for more information. To activate FDATool, press either the **New** button or the **Edit** button under the **Filters** list box in SPTool.



## Filter Visualization Tool

The Filter Visualization Tool (fvtool) allows you to view the characteristics of a designed or imported filter, including its magnitude response, phase response, group delay, phase delay, pole-zero plot, impulse response, and step response. To activate FVTool, click the **View** button under the **Filters** list box in SPTool.



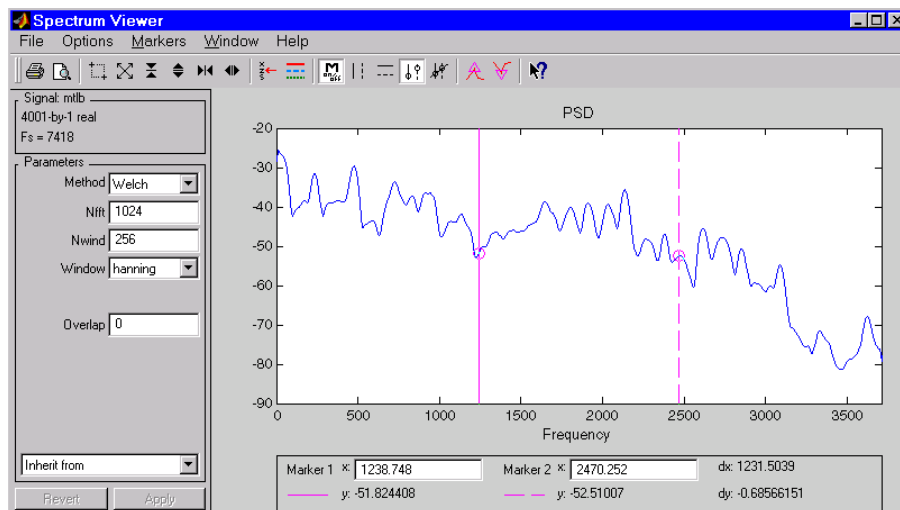
## Spectrum Viewer

The Spectrum Viewer allows you to analyze frequency-domain data graphically using a variety of methods of spectral density estimation, including the Burg method, the FFT method, the multitaper method, the MUSIC eigenvector method, Welch's method, and the Yule-Walker autoregressive method. To activate the Spectrum Viewer:

- Click the **Create** button under the **Spectra** list box to compute the power spectral density for a signal selected in the **Signals** list box in SPTool. You may need to click **Apply** to view the spectra.



- Click the **View** button to analyze spectra selected under the **Spectra** list box in SPTool.
- Click the **Update** button under the **Spectra** list box in SPTool to modify a selected power spectral density signal.



In addition, you can right-click in any plot display area of the GUIs to modify signal properties.

See Chapter 6, “SPTool: A Signal Processing GUI Suite” for a full discussion of how to use SPTool.

## See Also

fdatool, fvtool

# square

---

**Purpose** Square wave

**Syntax** `x = square(t)`  
`x = square(t,duty)`

**Description** `x = square(t)` generates a square wave with period  $2\pi$  for the elements of time vector `t`. `square(t)` is similar to `sin(t)`, but creates a square wave with peaks of  $\pm 1$  instead of a sine wave.

`x = square(t,duty)` generates a square wave with specified duty cycle, `duty`, which is a number between 0 and 100. The *duty cycle* is the percent of the period in which the signal is positive.

**See Also** `chirp`, `cos`, `diric`, `gauspuls`, `pulstran`, `rectpuls`, `sawtooth`, `sin`, `square`, `tripuls`

**Purpose** Convert digital filter state-space parameters to second-order sections form

**Syntax**

```
[ sos, g ] = ss2sos ( A, B, C, D )
[ sos, g ] = ss2sos ( A, B, C, D, iu )
[ sos, g ] = ss2sos ( A, B, C, D, 'order' )
[ sos, g ] = ss2sos ( A, B, C, D, iu, 'order' )
[ sos, g ] = ss2sos ( A, B, C, D, iu, 'order', 'scale' )
sos = ss2sos ( ... )
```

**Description** ss2sos converts a state-space representation of a given digital filter to an equivalent second-order section representation.

[ sos, g ] = ss2sos ( A, B, C, D ) finds a matrix sos in second-order section form with gain g that is equivalent to the state-space system represented by input arguments A, B, C, and D. The input system must be single output and real. sos is an  $L$ -by-6 matrix

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

whose rows contain the numerator and denominator coefficients  $b_{ik}$  and  $a_{ik}$  of the second-order sections of  $H(z)$ .

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

[ sos, g ] = ss2sos ( A, B, C, D, iu ) specifies a scalar iu that determines which input of the state-space system A, B, C, D is used in the conversion. The default for iu is 1.

[ sos, g ] = ss2sos ( A, B, C, D, 'order' ) and

[ sos, g ] = ss2sos ( A, B, C, D, iu, 'order' ) specify the order of the rows in sos, where 'order' is

- 'down', to order the sections so the first row of sos contains the poles closest to the unit circle
- 'up', to order the sections so the first row of sos contains the poles farthest from the unit circle (default)

The zeros are always paired with the poles closest to them.

`[sos,g] = ss2sos(A,B,C,D,iu,'order','scale')` specifies the desired scaling of the gain and the numerator coefficients of all second-order sections, where 'scale' is

- 'none', to apply no scaling (default)
- 'inf', to apply infinity-norm scaling
- 'two', to apply 2-norm scaling

Using infinity-norm scaling in conjunction with up-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling in conjunction with down-ordering minimizes the peak round-off noise.

---

**Note** Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

---

`sos = ss2sos(...)` embeds the overall system gain,  $g$ , in the first section,  $H_1(z)$ , so that

$$H(z) = \prod_{k=1}^L H_k(z)$$

---

**Note** Embedding the gain in the first section when scaling a direct-form II structure is not recommended and may result in erratic scaling. To avoid embedding the gain, use `ss2sos` with two outputs.

---

## Examples

Find a second-order section form of a Butterworth lowpass filter:

```
[A,B,C,D] = butter(5,0.2);
```

```
sos = ss2sos(A,B,C,D)
```

```
sos =
```

```
    0.0013    0.0013         0    1.0000   -0.5095         0
    1.0000    2.0008    1.0008    1.0000   -1.0966    0.3554
    1.0000    1.9979    0.9979    1.0000   -1.3693    0.6926
```

## Algorithm

ss2sos uses a four-step algorithm to determine the second-order section representation for an input state-space system:

- 1 It finds the poles and zeros of the system given by A, B, C, and D.
- 2 It uses the function zp2sos, which first groups the zeros and poles into complex conjugate pairs using the cplxpair function. zp2sos then forms the second-order sections by matching the pole and zero pairs according to the following rules:
  - a Match the poles closest to the unit circle with the zeros closest to those poles.
  - b Match the poles next closest to the unit circle with the zeros closest to those poles.
  - c Continue until all of the poles and zeros are matched.

ss2sos groups real poles into sections with the real poles closest to them in absolute value. The same rule holds for real zeros.
- 3 It orders the sections according to the proximity of the pole pairs to the unit circle. ss2sos normally orders the sections with poles closest to the unit circle last in the cascade. You can tell ss2sos to order the sections in the reverse order by specifying the 'down' flag.
- 4 ss2sos scales the sections by the norm specified in the 'scale' argument. For arbitrary  $H(\omega)$ , the scaling is defined by

$$\|H\|_p = \left[ \frac{1}{2\pi} \int_0^{2\pi} |H(\omega)|^p d\omega \right]^{\frac{1}{p}}$$

where  $p$  can be either  $\infty$  or 2. See the references for details. This scaling is an attempt to minimize overflow or peak round-off noise in fixed point filter implementations.

## Diagnostics

If there is more than one input to the system, `ss2sos` gives the following error message:

```
State-space system must have only one input.
```

## References

- [1] Jackson, L.B., *Digital Filters and Signal Processing*, 3rd ed., Kluwer Academic Publishers, Boston, 1996. Chapter 11.
- [2] Mitra, S.K., *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, 1998. Chapter 9.
- [3] Vaidyanathan, P.P., "Robust Digital Filter Structures," *Handbook for Digital Signal Processing*, S.K. Mitra and J.F. Kaiser, ed., John Wiley & Sons, New York, 1993, Chapter 7.

## See Also

`cplxpair`, `sos2ss`, `ss2tf`, `ss2zp`, `tf2sos`, `zp2sos`

**Purpose** Convert state-space filter parameters to transfer function form

**Syntax** `[b,a] = ss2tf(A,B,C,D,iu)`

**Description** `ss2tf` converts a state-space representation of a given system to an equivalent transfer function representation.

`[b,a] = ss2tf(A,B,C,D,iu)` returns the transfer function

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

of the system

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

from the `iu`-th input. Vector `a` contains the coefficients of the denominator in descending powers of  $s$ . The numerator coefficients are returned in array `b` with as many rows as there are outputs  $y$ . `ss2tf` also works with systems in discrete time, in which case it returns the  $z$ -transform representation.

The `ss2tf` function is part of the standard MATLAB language.

**Algorithm** The `ss2tf` function uses `poly` to find the characteristic polynomial  $\det(sI - A)$  and the equality:

$$H(s) = C(sI - A)^{-1}B = \frac{\det(sI - A + BC) - \det(sI - A)}{\det(sI - A)}$$

**See Also** `latc2tf`, `sos2tf`, `ss2sos`, `ss2zp`, `tf2ss`, `zp2tf`

**Purpose** Convert state-space filter parameters to zero-pole-gain form

**Syntax** `[z,p,k] = ss2zp(A,B,C,D,i)`

**Description** `ss2zp` converts a state-space representation of a given system to an equivalent zero-pole-gain representation. The zeros, poles, and gains of state-space systems represent the transfer function in factored form.

`[z,p,k] = ss2zp(A,B,C,D,i)` calculates the transfer function in factored form

$$H(s) = \frac{Z(s)}{P(s)} = k \frac{(s - z_1)(s - z_2) \cdots (s - z_n)}{(s - p_1)(s - p_2) \cdots (s - p_n)}$$

of the continuous-time system

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

from the *i*th input (using the *i*th columns of B and D). The column vector *p* contains the pole locations of the denominator coefficients of the transfer function. The matrix *z* contains the numerator zeros in its columns, with as many columns as there are outputs *y* (rows in C). The column vector *k* contains the gains for each numerator transfer function.

`ss2zp` also works for discrete time systems. The input state-space system must be real.

The `ss2zp` function is part of the standard MATLAB language.

**Examples** Here are two ways of finding the zeros, poles, and gains of a discrete-time transfer function:

$$H(z) = \frac{2 + 3z^{-1}}{1 + 0.4z^{-1} + z^{-2}}$$

```
b = [2 3];  
a = [1 0.4 1];  
[b,a] = eqtflength(b,a);  
[z,p,k] = tf2zp(b,a)
```



```

z =
    0.0000
   -1.5000

p =
   -0.2000 + 0.9798i
   -0.2000 - 0.9798i

k =
     2

[A,B,C,D] = tf2ss(b,a);
[z,p,k] = ss2zp(A,B,C,D,1)

z =
    0.0000
   -1.5000

p =
   -0.2000 + 0.9798i
   -0.2000 - 0.9798i

k =
     2

```

**Algorithm**

ss2zp finds the poles from the eigenvalues of the A array. The zeros are the finite solutions to a generalized eigenvalue problem:

```
z = eig([A B;C D], diag([ones(1,n) 0]));
```

In many situations this algorithm produces spurious large, but finite, zeros. ss2zp interprets these large zeros as infinite.

ss2zp finds the gains by solving for the first nonzero Markov parameters.

**References**

[1] Laub, A.J., and B.C. Moore, "Calculation of Transmission Zeros Using QZ Techniques," *Automatica* 14 (1978), p. 557.

**See Also**

pzmap, sos2zp, ss2sos, ss2tf, tf2zp, tf2zpk, zp2ss

# stepz

---

**Purpose** Step response of digital filter

**Syntax**

```
[h,t] = stepz(b,a)
[h,t] = stepz(b,a,n)
[h,t] = stepz(b,a,n,fs)
stepz(b,a)
stepz(Hd)
```

**Description** `[h,t] = stepz(b,a)` computes the step response of the filter with numerator coefficients `b` and denominator coefficients `a`. `stepz` chooses the number of samples and returns the response in the column vector `h` and sample times in the column vector `t` (where `t = [0:n-1]'`, and `n = length(t)` is computed automatically).

`[h,t] = stepz(b,a,n)` computes the first `n` samples of the step response when `n` is an integer (`t = [0:n-1]'`). I

`[h,t] = stepz(b,a,n,fs)` computes `n` samples and produces a vector `t` of length `n` so that the samples are spaced `1/fs` units apart. `fs` is assumed to be in Hz.

`stepz(b,a)` with no output arguments plots the step response in the current figure window.

`stepz(Hd)` plots the step response of the filter and displays the plot in `fvtool`. The input `Hd` is a `dfilt` filter object or an array of `dfilt` filter objects.

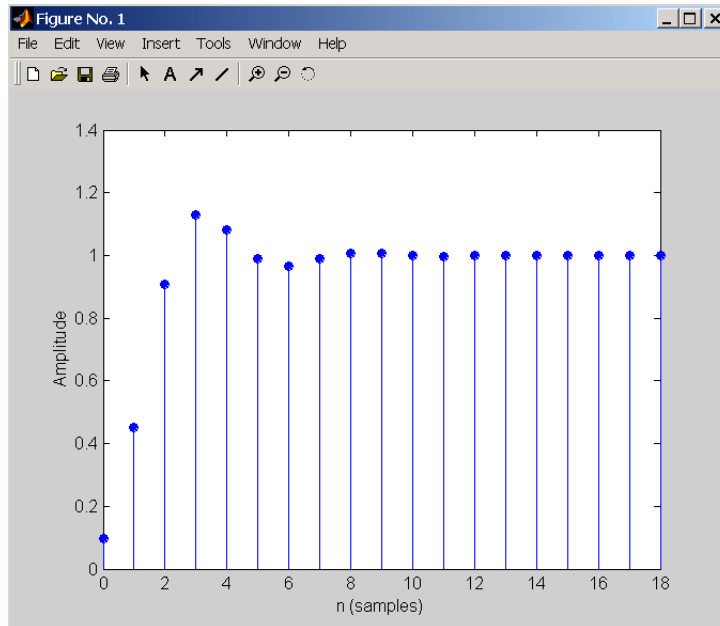
`stepz` works for both real and complex input systems.

## Examples

### Example 1

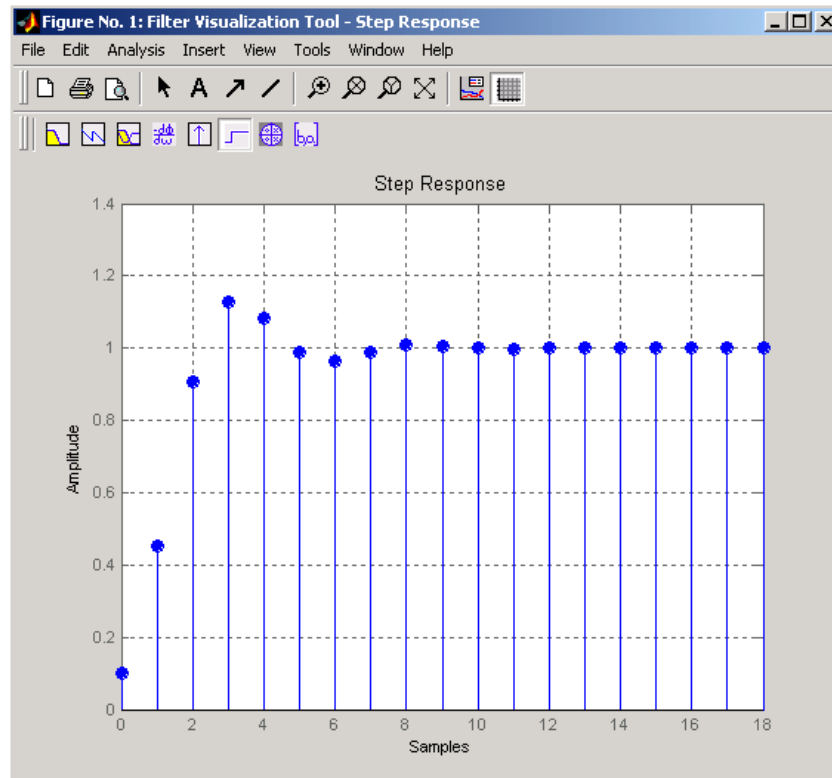
Plot the step response of a Butterworth filter:

```
[b,a] = butter(3,.4);
stepz(b,a)
```



The same example using a `dfilt` object and displaying the result in the Filter Visualization Tool (`fvtool`) is

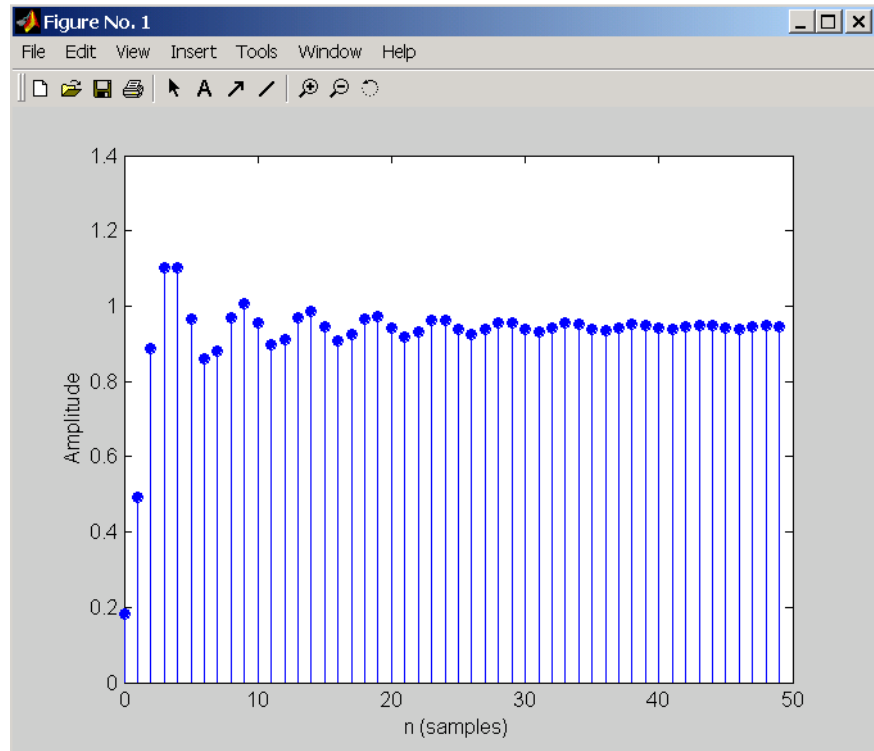
```
[b,a] = butter(3,.4);  
Hd=dfilt.df1(b,a);  
stepz(Hd)
```



## Example 2

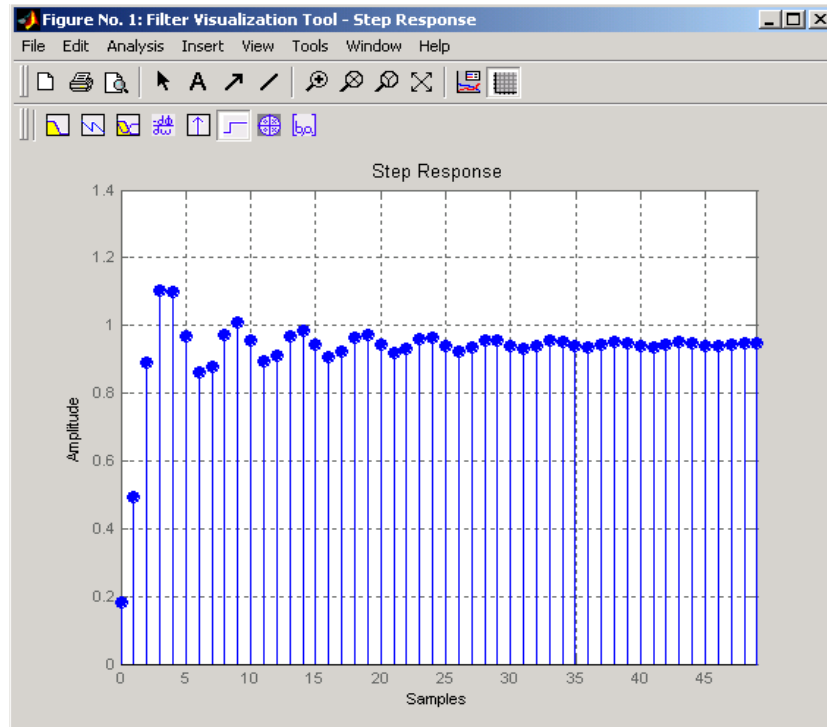
Plot the first 50 samples of the step response of a fourth-order lowpass elliptic filter with cutoff frequency of 0.4 times the Nyquist frequency:

```
[b,a] = ellip(4,0.5,20,0.4);
stepz(b,a,50)
```



The same example using a `dfilt` object and displaying the result in the Filter Visualization Tool (`fvtool`) is

```
[b,a] = ellip(4,0.5,20,0.4);  
Hd=dfilt.df1(b,a);  
stepz(Hd,50)
```



## Algorithm

stepz filters a length  $n$  step sequence using

```
filter(b,a,ones(1,n))
```

and plots the results using stem.

To compute  $n$  in the auto-length case, stepz either uses  $n = \text{length}(b)$  for the FIR case or first finds the poles using  $p = \text{roots}(a)$ , if  $\text{length}(a)$  is greater than 1.

If the filter is unstable,  $n$  is chosen to be the point at which the term from the largest pole reaches  $10^6$  times its original value.

If the filter is stable,  $n$  is chosen to be the point at which the term due to the largest amplitude pole is  $5 \cdot 10^{-5}$  of its original amplitude.

If the filter is oscillatory (poles on the unit circle only), stepz computes five periods of the slowest oscillation.

If the filter has both oscillatory and damped terms,  $n$  is chosen to equal five periods of the slowest oscillation or the point at which the term due to the largest (nonunity) amplitude pole is  $5 \cdot 10^{-5}$  of its original amplitude, whichever is greater.

stepz also allows for delays in the numerator polynomial. The number of delays is incorporated into the computation for the number of samples.

**See Also**

freqz, grpdelay, impz, phasez, zplane

# stmcb

---

**Purpose** Compute linear model using Steiglitz-McBride iteration

**Syntax**

```
[b,a] = stmcb(h,nb,na)
[b,a] = stmcb(y,x,nb,na)
[b,a] = stmcb(h,nb,na,niter)
[b,a] = stmcb(y,x,nb,na,niter)
[b,a] = stmcb(h,nb,na,niter,ai)
[b,a] = stmcb(y,x,nb,na,niter,ai)
```

**Description** Steiglitz-McBride iteration is an algorithm for finding an IIR filter with a prescribed time domain impulse response. It has applications in both filter design and system identification (parametric modeling).

`[b,a] = stmcb(h,nb,na)` finds the coefficients  $b$  and  $a$  of the system  $b(z)/a(z)$  with approximate impulse response  $h$ , exactly  $nb$  zeros, and exactly  $na$  poles.

`[b,a] = stmcb(y,x,nb,na)` finds the system coefficients  $b$  and  $a$  of the system that, given  $x$  as input, has  $y$  as output.  $x$  and  $y$  must be the same length.

`[b,a] = stmcb(h,nb,na,niter)` and

`[b,a] = stmcb(y,x,nb,na,niter)` use  $niter$  iterations. The default for  $niter$  is 5.

`[b,a] = stmcb(h,nb,na,niter,ai)` and

`[b,a] = stmcb(y,x,nb,na,niter,ai)` use the vector  $ai$  as the initial estimate of the denominator coefficients. If  $ai$  is not specified, `stmcb` uses the output argument from `[b,ai] = prony(h,0,na)` as the vector  $ai$ .

`stmcb` returns the IIR filter coefficients in length  $nb+1$  and  $na+1$  row vectors  $b$  and  $a$ . The filter coefficients are ordered in descending powers of  $z$ .

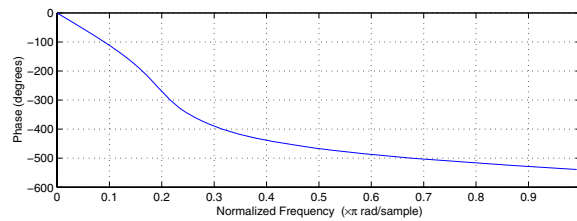
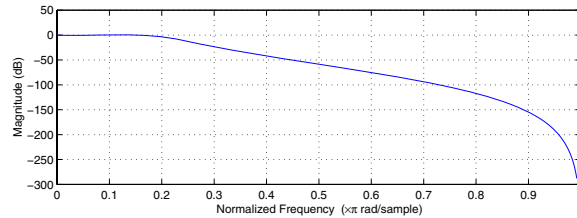
$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(nb+1)z^{-nb}}{a(1) + a(2)z^{-1} + \dots + a(na+1)z^{-na}}$$



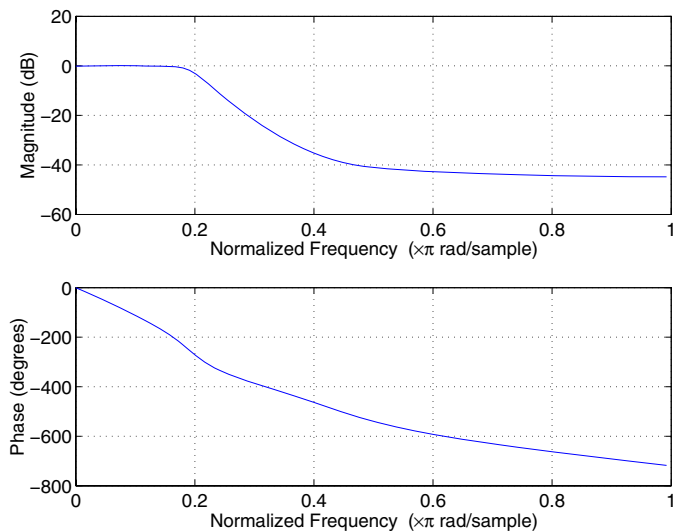
## Examples

Approximate the impulse response of a Butterworth filter with a system of lower order:

```
[b,a] = butter(6,0.2);  
h = filter(b,a,[1 zeros(1,100)]);  
freqz(b,a,128)
```



```
[bb,aa] = stmcb(h,4,4);  
freqz(bb,aa,128)
```



## Algorithm

stmcb attempts to minimize the squared error between the impulse response  $h$  of  $b(z)/a(z)$  and the input signal  $x$ .

$$\min_{a, b} \sum_{i=0}^{\infty} |x(i) - h(i)|^2$$

stmcb iterates using two steps:

- 1 It prefilters  $h$  and  $x$  using  $1/a(z)$ .
- 2 It solves a system of linear equations for  $b$  and  $a$  using  $\backslash$ .

stmcb repeats this process `niter` times. No checking is done to see if the  $b$  and  $a$  coefficients have converged in fewer than `niter` iterations.

## Diagnostics

If  $x$  and  $y$  have different lengths, stmcb produces this error message,

```
Input signal X and output signal Y must have the same length.
```

## References

[1] Steiglitz, K., and L.E. McBride, "A Technique for the Identification of Linear Systems," *IEEE Trans. Automatic Control*, Vol. AC-10 (1965), pp. 461-464.

[2] Ljung, L., *System Identification: Theory for the User*, Prentice-Hall, Englewood Cliffs, NJ, 1987, p. 297.

**See Also**

levinson, lpc, aryule, prony

# strips

---

## Purpose

Strip plot

## Syntax

```
strips(x)
strips(x,n)
strips(x,sd,fs)
strips(x,sd,fs,scale)
```

## Description

`strips(x)` plots vector `x` in horizontal strips of length 250. If `x` is a matrix, `strips(x)` plots each column of `x`. The left-most column (column 1) is the top horizontal strip.

`strips(x,n)` plots vector `x` in strips that are each `n` samples long.

`strips(x,sd,fs)` plots vector `x` in strips of duration `sd` seconds, given a sampling frequency of `fs` samples per second.

`strips(x,sd,fs,scale)` scales the vertical axes.

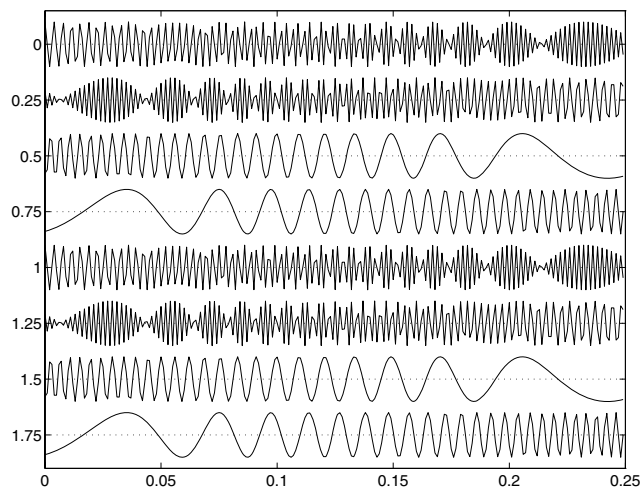
If `x` is a matrix, `strips(x,n)`, `strips(x,sd,fs)`, and `strips(x,sd,fs,scale)` plot the different columns of `x` on the same strip plot.

`strips` ignores the imaginary part of complex-valued `x`.

## Examples

Plot two seconds of a frequency modulated sinusoid in 0.25 second strips:

```
fs = 1000; % Sampling frequency
t = 0:1/fs:2; % Time vector
x = vco(sin(2*pi*t),[10 490],fs); % FM waveform
strips(x,0.25,fs)
```



**See Also**

plot, stem

# taylorwin

---

## Purpose

Taylor window

## Syntax

```
w = taylorwin(L)
w = taylorwin(L,nbar)
w = taylorwin(L,nbar,s11)
```

## Description

Taylor windows are similar to Chebyshev windows. While a Chebyshev window has the narrowest possible mainlobe for a specified sidelobe level, a Taylor window allows you to make tradeoffs between the mainlobe width and sidelobe level. The Taylor distribution avoids edge discontinuities, so Taylor window sidelobes decrease monotonically. Taylor window coefficients are not normalized. Taylor windows are typically used in radar applications, such as weighting synthetic aperture radar images and antenna design.

`w = taylorwin(L)` returns an L-point Taylor window in a column vector `w`. The values in this vector are the window weights or coefficients. `L` must be a positive integer. The default value for the number of approximately equal height sidelobes (`nbar`) is 4 and for the maximum sidelobe level (`s11`) is -30.

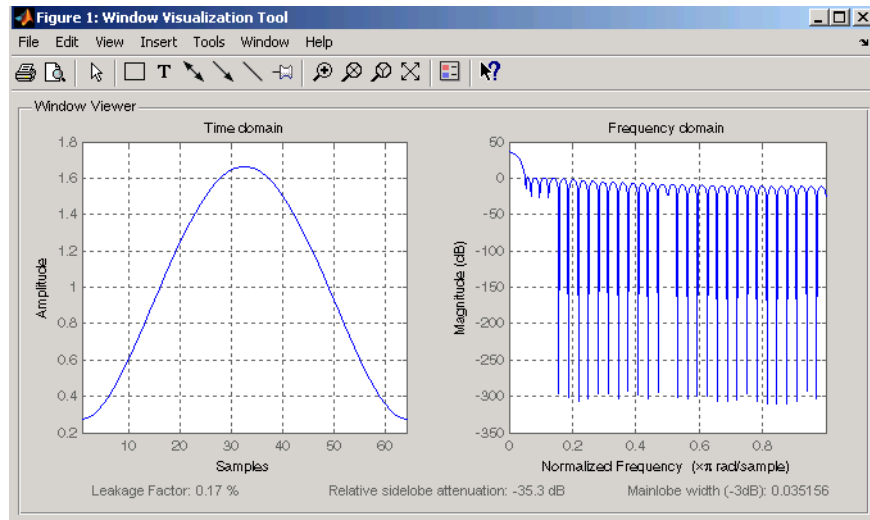
`w = taylorwin(L,nbar)` returns an L-point Taylor window with `nbar` nearly constant-level sidelobes adjacent to the mainlobe. These sidelobes are “nearly constant-level” because some decay occurs in the transition region. `nbar` must be a positive integer.

`w = taylorwin(L,nbar,s11)` returns an L-point Taylor window with a maximum sidelobe level of `s11` dB relative to the mainlobe peak. `s11` must be a negative value, such as -30, which produces sidelobes with peaks 30 dB down from the mainlobe peak.

## Examples

Generate a 64-point Taylor window with four nearly constant-level sidelobes and a peak sidelobe level of -35 dB relative to the mainlobe peak.

```
w = taylorwin(64,4,-35);
wvtool(w);
```



## References

- [1] Carrara, W.G., R.M. Majewski and R.S. Goodman, *Spotlight Synthetic Aperature Radar: Signal Processing Algorithms*, Artech House Publishers, Boston, 1995, Appendix D.2.
- [2] Brookner, Eli, *Practical Phased Array Antenna Systems*, Lex Book, Lexington, MA, 1991.

# tf2latc

---

## Purpose

Convert transfer function filter parameters to lattice filter form

## Syntax

```
[k,v] = tf2latc(b,a)
k = tf2latc(1,a)
[k,v] = tf2latc(1,a)
k = tf2latc(b)
k = tf2latc(b,'phase')
```

## Description

`[k,v] = tf2latc(b,a)` finds the lattice parameters `k` and the ladder parameters `v` for an IIR (ARMA) lattice-ladder filter, normalized by `a(1)`. Note that an error is generated if one or more of the lattice parameters are exactly equal to 1.

`k = tf2latc(1,a)` finds the lattice parameters `k` for an IIR all-pole (AR) lattice filter.

`[k,v] = tf2latc(1,a)` returns the scalar ladder coefficient at the correct position in vector `v`. All other elements of `v` are zero.

`k = tf2latc(b)` finds the lattice parameters `k` for an FIR (MA) lattice filter, normalized by `b(1)`.

`k = tf2latc(b,'phase')` specifies the type of FIR (MA) lattice filter, where `'phase'` is

- `'max'`, for a maximum phase filter.
- `'min'`, for a minimum phase filter.

## See Also

`latc2tf`, `latcfilt`, `tf2sos`, `tf2ss`, `tf2zp`, `tf2zpk`



**Purpose** Convert digital filter transfer function data to second-order sections form

**Syntax**

```
[sos,g] = tf2sos(b,a)
[sos,g] = tf2sos(b,a,'order')
[sos,g] = tf2sos(b,a,'order','scale')
sos = tf2sos(...)
```

**Description** tf2sos converts a transfer function representation of a given digital filter to an equivalent second-order section representation.

[sos,g] = tf2sos(b,a) finds a matrix sos in second-order section form with gain g that is equivalent to the digital filter represented by transfer function coefficient vectors a and b.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2z^{-1} + \dots + b_{n+1}z^{-n}}{a_1 + a_2z^{-1} + \dots + a_{m+1}z^{-m}}$$

sos is an  $L$ -by-6 matrix

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

whose rows contain the numerator and denominator coefficients  $b_{ik}$  and  $a_{ik}$  of the second-order sections of  $H(z)$ .

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

[sos,g] = tf2sos(b,a,'order') specifies the order of the rows in sos, where 'order' is

- 'down', to order the sections so the first row of sos contains the poles closest to the unit circle

- 'up', to order the sections so the first row of sos contains the poles farthest from the unit circle (default)

[sos,g] = tf2sos(b,a,'order','scale') specifies the desired scaling of the gain and numerator coefficients of all second-order sections, where 'scale' is:

- 'none', to apply no scaling (default)
- 'inf', to apply infinity-norm scaling
- 'two', to apply 2-norm scaling

Using infinity-norm scaling in conjunction with up-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling in conjunction with down-ordering minimizes the peak round-off noise.

---

**Note** Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

---

sos = tf2sos(...) embeds the overall system gain, g, in the first section,  $H_1(z)$ , so that

$$H(z) = \prod_{k=1}^L H_k(z)$$

---

**Note** Embedding the gain in the first section when scaling a direct-form II structure is not recommended and may result in erratic scaling. To avoid embedding the gain, use ss2sos with two outputs.

---

## Algorithm

tf2sos uses a four-step algorithm to determine the second-order section representation for an input transfer function system:

- 1 It finds the poles and zeros of the system given by b and a.
- 2 It uses the function zp2sos, which first groups the zeros and poles into complex conjugate pairs using the cplxpair function. zp2sos then forms the

second-order sections by matching the pole and zero pairs according to the following rules:

- a Match the poles closest to the unit circle with the zeros closest to those poles.
- b Match the poles next closest to the unit circle with the zeros closest to those poles.
- c Continue until all of the poles and zeros are matched.

tf2sos groups real poles into sections with the real poles closest to them in absolute value. The same rule holds for real zeros.

- 3 It orders the sections according to the proximity of the pole pairs to the unit circle. tf2sos normally orders the sections with poles closest to the unit circle last in the cascade. You can tell tf2sos to order the sections in the reverse order by specifying the 'down' flag.
- 4 tf2sos scales the sections by the norm specified in the 'scale' argument. For arbitrary  $H(\omega)$ , the scaling is defined by

$$\|H\|_p = \left[ \frac{1}{2\pi} \int_0^{2\pi} |H(\omega)|^p d\omega \right]^{\frac{1}{p}}$$

where  $p$  can be either  $\infty$  or 2. See the references for details on the scaling. This scaling is an attempt to minimize overflow or peak round-off noise in fixed point filter implementations.

## References

- [1] Jackson, L.B., *Digital Filters and Signal Processing*, 3rd ed., Kluwer Academic Publishers, Boston, 1996, Chapter 11.
- [2] Mitra, S.K., *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, 1998, Chapter 9.
- [3] Vaidyanathan, P.P., "Robust Digital Filter Structures," *Handbook for Digital Signal Processing*, S.K. Mitra and J.F. Kaiser, ed., John Wiley & Sons, New York, 1993, Chapter 7.

## See Also

cplxpair, sos2tf, ss2sos, tf2ss, tf2zp, tf2zpk, zp2sos

# tf2ss

---

**Purpose** Convert transfer function filter parameters to state-space form

**Syntax** `[A,B,C,D] = tf2ss(b,a)`

**Description** `tf2ss` converts the parameters of a transfer function representation of a given system to those of an equivalent state-space representation.

`[A,B,C,D] = tf2ss(b,a)` returns the A, B, C, and D matrices of a state space representation for the single-input transfer function

$$H(s) = \frac{B(s)}{A(s)} = \frac{b_1 s^{n-1} + \dots + b_{n-1} s + b_n}{a_1 s^{m-1} + \dots + a_{m-1} s + a_m} = C(sI - A)^{-1}B + D$$

in controller canonical form

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

The input vector `a` contains the denominator coefficients in descending powers of  $s$ . The rows of the matrix `b` contain the vectors of numerator coefficients (each row corresponds to an output). In the discrete-time case, you must supply `b` and `a` to correspond to the numerator and denominator polynomials with coefficients in descending powers of  $z$ .

For discrete-time systems you must make `b` have the same number of columns as the length of `a`. You can do this by padding each numerator represented in `b` (and possibly the denominator represented in the vector `a`) with trailing zeros. You can use the function `eqtflength` to accomplish this if `b` and `a` are vectors of unequal lengths.

The `tf2ss` function is part of the standard MATLAB language.

**Examples** Consider the system:

$$H(s) = \frac{\begin{bmatrix} 2s + 3 \\ s^2 + 2s + 1 \end{bmatrix}}{s^2 + 0.4s + 1}$$

To convert this system to state-space, type

```
b = [0 2 3; 1 2 1];  
a = [1 0.4 1];  
[A,B,C,D] = tf2ss(b,a)
```

```
A =  
  -0.4000   -1.0000  
   1.0000         0
```

```
B =  
     1  
     0
```

```
C =  
  2.0000   3.0000  
  1.6000         0
```

```
D =  
     0  
     1
```

---

**Note** There is disagreement in the literature on naming conventions for the canonical forms. It is easy, however, to generate similarity transformations that convert these results to other forms.

---

**See Also**

sos2ss, ss2tf, tf2sos, tf2zp, tf2zpk, zp2ss

## Purpose

Convert transfer function filter parameters to zero-pole-gain form

## Syntax

`[z,p,k] = tf2zp(b,a)`

## Description

tf2zp finds the zeros, poles, and gains of a continuous-time transfer function.

---

**Note** You should use tf2zp when working with positive powers ( $s^2 + s + 1$ ), such as in continuous-time transfer functions. A similar function, tf2zpk, is more useful when working with transfer functions expressed in inverse powers ( $1 + z^{-1} + z^{-2}$ ), which is how transfer functions are usually expressed in DSP.

---

`[z,p,k] = tf2zp(b,a)` finds the matrix of zeros  $z$ , the vector of poles  $p$ , and the associated vector of gains  $k$  from the transfer function parameters  $b$  and  $a$ :

- The numerator polynomials are represented as columns of the matrix  $b$ .
- The denominator polynomial is represented in the vector  $a$ .

Given a SIMO continuous-time system in polynomial transfer function form

$$H(s) = \frac{B(s)}{A(s)} = \frac{b_1 s^{n-1} + \dots + b_{n-1} s + b_n}{a_1 s^{m-1} + \dots + a_{m-1} s + a_m}$$

you can use the output of tf2zp to produce the single-input, multioutput (SIMO) factored transfer function form

$$H(s) = \frac{Z(s)}{P(s)} = k \frac{(s - z_1)(s - z_2) \cdots (s - z_m)}{(s - p_1)(s - p_2) \cdots (s - p_n)}$$

The following describes the input and output arguments for tf2zp:

- The vector  $a$  specifies the coefficients of the denominator polynomial  $A(s)$  (or  $A(z)$ ) in descending powers of  $s$  ( $z^{-1}$ ).
- The  $i$ th row of the matrix  $b$  represents the coefficients of the  $i$ th numerator polynomial (the  $i$ th row of  $B(s)$  or  $B(z)$ ). Specify as many rows of  $b$  as there are outputs.

- For continuous-time systems, choose the number  $nb$  of columns of  $b$  to be less than or equal to the length  $na$  of the vector  $a$ .
- For discrete-time systems, choose the number  $nb$  of columns of  $b$  to be equal to the length  $na$  of the vector  $a$ . You can use the function `eqtflength` to provide equal length vectors in the case that  $b$  and  $a$  are vectors of unequal lengths. Otherwise, pad the numerators in the matrix  $b$  (and, possibly, the denominator vector  $a$ ) with zeros.
- The zero locations are returned in the columns of the matrix  $z$ , with as many columns as there are rows in  $b$ .
- The pole locations are returned in the column vector  $p$  and the gains for each numerator transfer function in the vector  $k$ .

The `tf2zp` function is part of the standard MATLAB language.

### Examples

Find the zeros, poles, and gains of this continuous-time system:

$$H(s) = \frac{2s^2 + 3s}{s^2 + 0.4s + 1}$$

```
b = [2 3];
a = [1 0.4 1];
```

```
[b,a] = eqtflength(b,a);           % Make lengths equal.
[z,p,k] = tf2zp(b,a)              % Obtain the zero-pole-gain form.
```

```
z =
      0
 -1.5000

p =
-0.2000 + 0.9798i
-0.2000 - 0.9798i

k =
      2
```

### See Also

`sos2zp`, `ss2zp`, `tf2sos`, `tf2ss`, `tf2zpk`, `zp2tf`

# tf2zpk

## Purpose

Convert transfer function filter parameters to zero-pole-gain form

## Syntax

`[z,p,k] = tf2zpk(b,a)`

## Description

tf2zpk finds the zeros, poles, and gains of a discrete-time transfer function.

---

**Note** You should use tf2zpk when working with transfer functions expressed in inverse powers ( $1 + z^{-1} + z^{-2}$ ), which is how transfer functions are usually expressed in DSP. A similar function, tf2zp, is more useful for working with positive powers ( $s^2 + s + 1$ ), such as in continuous-time transfer functions.

---

`[z,p,k] = tf2zpk(b,a)` finds the matrix of zeros  $z$ , the vector of poles  $p$ , and the associated vector of gains  $k$  from the transfer function parameters  $b$  and  $a$ :

- The numerator polynomials are represented as columns of the matrix  $b$ .
- The denominator polynomial is represented in the vector  $a$ .

Given a single-input, multiple output (SIMO) discrete-time system in polynomial transfer function form

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} \cdots + b_{n-1} z^{-n} + b_n z^{-n-1}}{a_1 + a_2 z^{-1} \cdots + a_{m-1} z^{-m} + a_m z^{-m-1}}$$

you can use the output of tf2zpk to produce the single-input, multioutput (SIMO) factored transfer function form

$$H(z) = \frac{Z(z)}{P(z)} = k \frac{(z - z_1)(z - z_2) \cdots (z - z_m)}{(z - p_1)(z - p_2) \cdots (z - p_n)}$$

The following describes the input and output arguments for tf2zpk:

- The vector  $a$  specifies the coefficients of the denominator polynomial  $A(z)$  in descending powers of  $z$ .
- The  $i$ th row of the matrix  $b$  represents the coefficients of the  $i$ th numerator polynomial (the  $i$ th row of  $B(s)$  or  $B(z)$ ). Specify as many rows of  $b$  as there are outputs.



- The zero locations are returned in the columns of the matrix z, with as many columns as there are rows in b.
- The pole locations are returned in the column vector p and the gains for each numerator transfer function in the vector k.

## Examples

Find the poles, zeros, and gain of a Butterworth filter:

```
[b,a] = butter(3,.4);
```

```
[z,p,k] = tf2zpk(b,a)
```

```
z =
```

```
-1.0000
```

```
-1.0000 + 0.0000i
```

```
-1.0000 - 0.0000i
```

```
p =
```

```
0.2094 + 0.5582i
```

```
0.2094 - 0.5582i
```

```
0.1584
```

```
k =
```

```
0.0985
```

## See Also

sos2zp, ss2zp, tf2sos, tf2ss, tf2zp, zp2tf

# tfestimate

---

**Purpose** Transfer function estimate

**Syntax**

```
Txy = tfestimate(x,y)
Txy = tfestimate(x,y>window)
Txy = tfestimate(x,y>window,noverlap)
[Txy,W] = tfestimate(x,y>window,noverlap,nfft)
[Txy,F] = tfestimate(x,y>window,noverlap,nfft,fs)
[...] = tfestimate(x,y,...,'whole')
tfestimate(...)
```

**Description** `Txy = tfestimate(x,y)` finds a transfer function estimate `Txy` given input signal vector `x` and output signal vector `y`. Vectors `x` and `y` must be the same length. The relationship between the input `x` and output `y` is modeled by the linear, time-invariant transfer function `Txy`. The *transfer function* is the quotient of the cross power spectral density ( $P_{xy}$ ) of `x` and `y` and the power spectral density ( $P_{xx}$ ) of `x`.

$$T_{xy}(f) = \frac{P_{xy}(f)}{P_{xx}(f)}$$

If `x` is real, `tfestimate` estimates the transfer function at positive frequencies only; in this case, the output `Txy` is a column vector of length `nfft/2+1` for `nfft` even and `(nfft+1)/2` for `nfft` odd. If `x` or `y` is complex, `tfestimate` estimates the transfer function for both positive and negative frequencies and `Txy` has length `nfft`.

tfestimate uses the following default values:

Parameter	Description	Default Value
nfft	<p>FFT length which determines the frequencies at which the PSD is estimated</p> <p>For real <math>x</math> and <math>y</math>, the length of <math>T_{xy}</math> is <math>(nfft/2+1)</math> if <math>nfft</math> is even or <math>(nfft+1)/2</math> if <math>nfft</math> is odd. For complex <math>x</math> or <math>y</math>, the length of <math>T_{xy}</math> is <math>nfft</math>.</p>	Maximum of 256 or the next power of 2 greater than the length of each section of $x$ or $y$
fs	Sampling frequency	1
window	Windowing function and number of samples to use to section $x$ and $y$	Periodic Hamming window of length $nfft$
noverlap	Number of samples by which the sections overlap	Value to obtain 50% overlap

**Note** You can use the empty matrix `[]` to specify the default value for any input argument except  $x$  or  $y$ . For example, `Txy = tfestimate(x,y,[],[],128)` uses a Hamming window, default `noverlap` to obtain 50% overlap, and the specified 128 `nfft`.

`Txy = tfestimate(x,y>window)` specifies a windowing function, divides  $x$  and  $y$  into overlapping sections of the specified window length, and windows each section using the specified window function. If you supply a scalar for `window`, `Txy` uses a Hamming window of that length. The length of the window must be less than or equal to `nfft`. If the length of the window exceeds `nfft`, `tfestimate` zero pads the sections

# tfestimate

---

`Txy = tfestimate(x,y>window,noverlap)` overlaps the sections of `x` by `noverlap` samples. `noverlap` must be an integer smaller than the length of `window`.

`[Txy,W] = tfestimate(x,y>window,noverlap,nfft)` uses the specified FFT length `nfft` in estimating the PSD and CPSD estimates for the transfer function. It also returns `W`, which is the vector of normalized frequencies (in rad/sample) at which the `tfestimate` is estimated. For real signals, the range of `W` is `[0, pi]` when `nfft` is even and `[0, pi)` when `nfft` is odd. For complex signals, the range of `W` is `[0, 2pi)`.

`[Txy,F] = tfestimate(x,y>window,noverlap,nfft,fs)` returns `Txy` as a function of frequency and a vector `F` of frequencies at which `tfestimate` estimates the transfer function. `fs` is the sampling frequency in Hz. `F` is the same size as `Txy`, so `plot(f,Txy)` plots the transfer function estimate versus properly scaled frequency. For real signals, the range of `F` is `[0, fs/2]` when `nfft` is even and `[0, fs/2)` when `nfft` is odd. For complex signals, the range of `F` is `[0, fs)`.

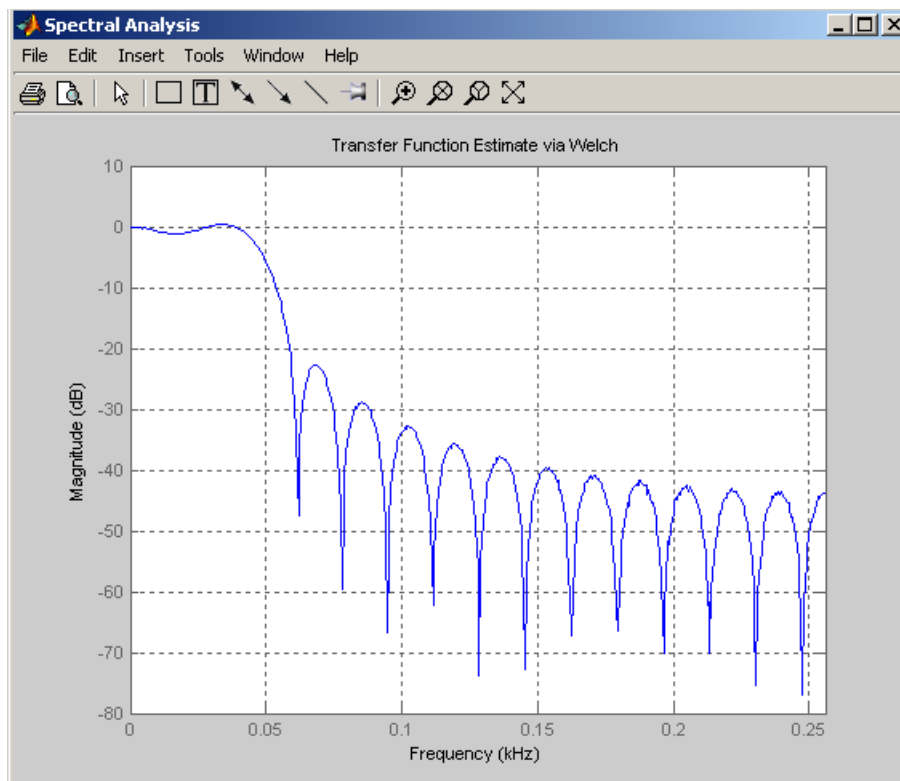
`[...] = tfestimate(x,y,...,'whole')` returns a transfer function estimate with frequencies that range over the whole Nyquist interval. Specifying `'half'` uses half the Nyquist interval.

`tfestimate(...)` with no output arguments plots the transfer function estimate in the current figure window.

## Examples

Compute and plot the transfer function estimate between two colored noise sequences `x` and `y`:

```
h = fir1(30,0.2,rectwin(31));
x = randn(16384,1);
y = filter(h,1,x);
tfestimate(x,y,1024,[],[],512)
```

**Algorithm**

tfestimate uses Welch's averaged periodogram method. See pwelch for details.

**See Also**

cpsd, mscohere, periodogram, pwelch, spectrum

# triang

---

**Purpose** Triangular window

**Syntax** `w = triang(L)`

**Description** `triang(L)` returns an L-point triangular window in the column vector `w`. The coefficients of a triangular window are:

For L odd:

$$w(n) = \begin{cases} \frac{2n}{L+1}, & 1 \leq n \leq \frac{L+1}{2} \\ -\frac{2(L-n+1)}{L+1}, & \frac{L+1}{2} < n \leq L \end{cases}$$

For L even:

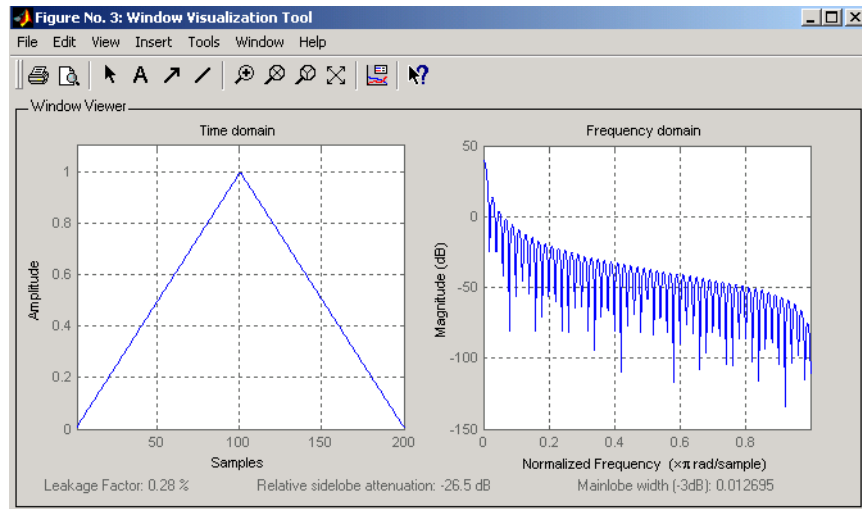
$$w(n) = \begin{cases} \frac{2n}{L}, & 1 \leq n \leq \frac{L}{2} \\ \frac{2(L-n)+1}{L}, & \frac{L}{2} + 1 \leq n \leq L \end{cases}$$

The triangular window is very similar to a Bartlett window. The Bartlett window always ends with zeros at samples 1 and L, while the triangular window is nonzero at those points. For L odd, the center L-2 points of `triang(L-2)` are equivalent to `bartlett(L)`.

## Examples

Create a 200-point triangular window and plot the result using `WVTool`.

```
L=200;  
wvtool(triang(L))
```



## References

[1] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 447-448.

## See Also

barthannwin, bartlett, blackmanharris, bohmanwin, nuttallwin, parzenwin, rectwin, window, wintool, wvtool

# tripuls

---

**Purpose**                    Sampled aperiodic triangle

**Syntax**                    `y = tripuls(T)`  
                              `y = tripuls(T,w)`  
                              `y = tripuls(T,w,s)`

**Description**              `y = tripuls(T)` returns a continuous, aperiodic, symmetric, unity-height triangular pulse at the times indicated in array `T`, centered about  $T=0$  and with a default width of 1.

`y = tripuls(T,w)` generates a triangular pulse of width `w`.

`y = tripuls(T,w,s)` generates a triangular pulse with skew `s`, where  $-1 < s < 1$ . When `s` is 0, a symmetric triangular pulse is generated.

**See Also**                    `chirp`, `cos`, `diric`, `gauspuls`, `pulstran`, `rectpuls`, `sawtooth`, `sin`, `square`,  
`tripuls`



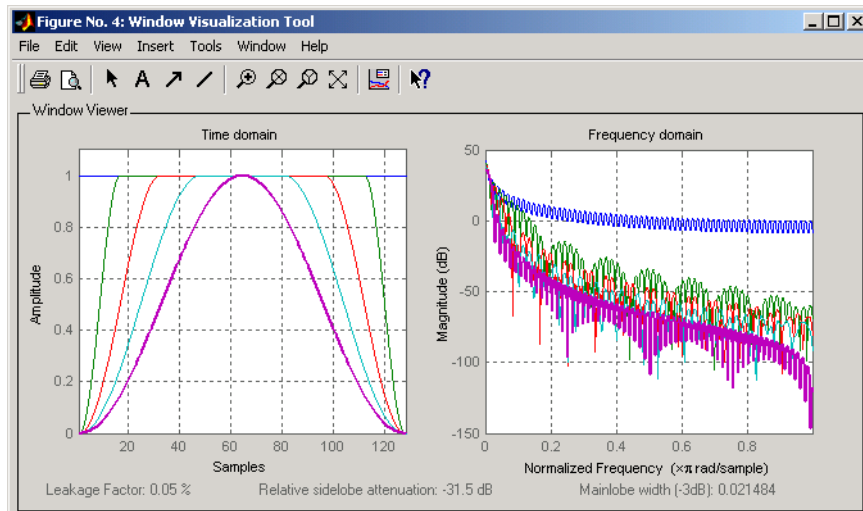
**Purpose** Tukey (tapered cosine) window

**Syntax** `w = tukeywin(L,r)`

**Description** `w = tukeywin(L,r)` returns an L-point, Tukey window in column vector `w`. Tukey windows are cosine-tapered windows. `r` is the ratio of taper to constant sections and is between 0 and 1.  $r \leq 0$  is a `rectwin` window and  $r \geq 1$  is a `hann` window. The default value for `r` is 0.5.

**Examples** Compute 128-point Tukey windows with five different tapers and display the results using `WVTool`:

```
L=128;
t0=tukeywin(L,0);           %Equivalent to rectangular window
t25=tukeywin(L,0.25);
t5=tukeywin(L);            %r=0.5
t75=tukeywin(L,0.75);
t1=tukeywin(L,1);         %Equivalent to Hann window
wvtool(t0,t25,t5,t75,t1)
```



# tukeywin

---

## Algorithm

The equation for computing the coefficients of a Tukey window is

$$w(n) = \begin{cases} 1.0 & 0 \leq |n| \leq \alpha \frac{N}{2} \\ \frac{1}{2} \left( 1 + \cos \left( \pi \frac{n - \alpha \frac{N}{2}}{2(1 - \alpha) \frac{N}{2}} \right) \right) & \alpha \frac{N}{2} \leq |n| \leq \frac{N}{2} \end{cases}$$

The window length  $L = N+1$ .

## References

[1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66 (January 1978). pp. 66-67.

## See Also

chebwin, gausswin, kaiser, window, wintool, wvtool

**Purpose** Decode  $2^n$ -level quantized integer inputs to floating-point outputs

**Syntax**

```
y = udecode(u,n)
y = udecode(u,n,v)
y = udecode(u,n,v, 'saturatemode')
```

**Description** `y = udecode(u,n)` inverts the operation of `uencode` and reconstructs quantized floating-point values from an encoded multidimensional array of integers `u`. The input argument `n` must be an integer between 2 and 32. The integer `n` specifies that there are  $2^n$  quantization levels for the inputs, so that entries in `u` must be either:

- Signed integers in the range  $[-2^n/2, (2^n/2) - 1]$
- Unsigned integers in the range  $[0, 2^n-1]$

Inputs can be real or complex values of any integer data type (`uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`). Overflows (entries in `u` outside of the ranges specified above) are saturated to the endpoints of the range interval. The output `y` has the same dimensions as `u`. Its entries have values in the range  $[-1, 1]$ .

`y = udecode(u,n,v)` decodes `u` such that the output `y` has values in the range  $[-v, v]$ , where the default value for `v` is 1.

`y = udecode(u,n,v, 'SaturateMode')` decodes `u` and treats input overflows (entries in `u` outside of  $[-v, v]$ ) according to the string `'saturatemode'`, which can be one of the following:

- `'saturate'`: Saturate overflows. This is the default method for treating overflows.
  - Entries in signed inputs `u` whose values are outside of the range  $[-2^n/2, (2^n/2) - 1]$  are assigned the value determined by the closest endpoint of this interval.
  - Entries in unsigned inputs `u` whose values are outside of the range  $[0, 2^n-1]$  are assigned the value determined by the closest endpoint of this interval.
- `'wrap'`: Wrap all overflows according to the following:

# udecode

---

- Entries in signed inputs  $u$  whose values are outside of the range  $[-2^n/2, (2^n/2) - 1]$  are wrapped back into that range using modulo  $2^n$  arithmetic (calculated using  $u = \text{mod}(u + 2^n/2, 2^n) - (2^n/2)$ ).
- Entries in unsigned inputs  $u$  whose values are outside of the range  $[0, 2^n - 1]$  are wrapped back into the required range before decoding using modulo  $2^n$  arithmetic (calculated using  $u = \text{mod}(u, 2^n)$ ).

## Examples

```
u = int8([-1 1 2 -5]); % Create a signed 8-bit integer string.  
ysat = udecode(u,3) % Decode with 3 bits.  
ysat =
```

```
-0.2500    0.2500    0.5000   -1.0000
```

Notice the last entry in  $u$  saturates to 1, the default peak input magnitude. Change the peak input magnitude:

```
ysatv = udecode(u,3,6) % Set the peak input magnitude to 6.  
ysatv =
```

```
-1.5000    1.5000    3.0000   -6.0000
```

The last input entry still saturates. Try wrapping the overflows:

```
ywrap = udecode(u,3,6,'wrap')  
ywrap =
```

```
-1.5000    1.5000    3.0000    4.5000
```

Try adding more quantization levels:

```
yprec = udecode(u,5)  
yprec =
```

```
-0.0625    0.0625    0.1250   -0.3125
```

## Algorithm

The algorithm adheres to the definition for uniform decoding specified in ITU-T Recommendation G.701. Integer input values are uniquely mapped (decoded) from one of  $2^n$  uniformly spaced integer values to quantized floating-point values in the range  $[-v, v]$ . The smallest integer input value allowed is mapped to  $-v$  and the largest integer input value allowed is mapped to  $v$ . Values outside of the allowable input range are either saturated or wrapped, according to specification.

The real and imaginary components of complex inputs are decoded independently.

## References

*General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms*, International Telecommunication Union, ITU-T Recommendation G.701, March, 1993.

## See Also

uencode

# uencode

---

## Purpose

Quantize and encode floating-point inputs to integer outputs

## Syntax

```
y = uencode(u,n)
y = uencode(u,n,v)
y = uencode(u,n,v, 'SignFlag')
```

## Description

`y = uencode(u,n)` quantizes the entries in a multidimensional array of floating-point numbers `u` and encodes them as integers using  $2^n$ -level quantization. `n` must be an integer between 2 and 32 (inclusive). Inputs can be real or complex, double- or single-precision. The output `y` and the input `u` are arrays of the same size. The elements of the output `y` are unsigned integers with magnitudes in the range  $[0, 2^n-1]$ . Elements of the input `u` outside of the range  $[-1, 1]$  are treated as overflows and are saturated.

- For entries in the input `u` that are less than -1, the value of the output of `uencode` is 0.
- For entries in the input `u` that are greater than 1, the value of the output of `uencode` is  $2^n-1$ .

`y = uencode(u,n,v)` allows the input `u` to have entries with floating-point values in the range  $[-v, v]$  before saturating them (the default value for `v` is 1). Elements of the input `u` outside of the range  $[-v, v]$  are treated as overflows and are saturated:

- For input entries less than `-v`, the value of the output of `uencode` is 0.
- For input entries greater than `v`, the value of the output of `uencode` is  $2^n-1$ .

`y = uencode(u,n,v, 'SignFlag')` maps entries in a multidimensional array of floating-point numbers `u` whose entries have values in the range  $[-v, v]$  to an integer output `y`. Input entries outside this range are saturated. The integer type of the output depends on the string `'SignFlag'` and the number of quantization levels  $2^n$ . The string `'SignFlag'` can be one of the following:

- `'signed'`: Outputs are signed integers with magnitudes in the range  $[-2^n/2, (2^n/2) - 1]$ .
- `'unsigned'` (default): Outputs are unsigned integers with magnitudes in the range  $[0, 2^n-1]$ .

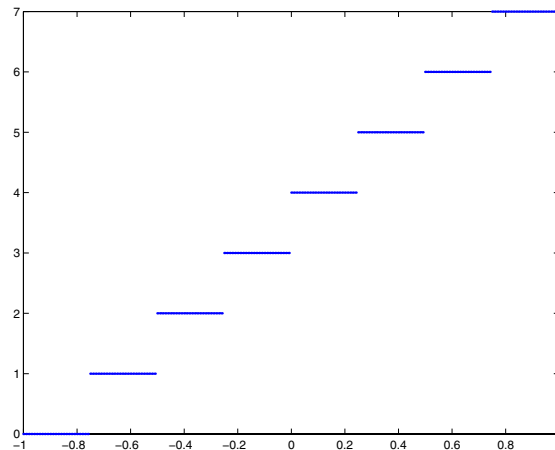
The output data types are optimized for the number of bits as shown in the table below.

<b>n</b>	<b>Unsigned Integer</b>	<b>Signed Integer</b>
2 to 8	uint8	int8
9 to 16	uint16	int16
17 to 32	uint32	int32

## Examples

Map floating-point scalars in  $[-1, 1]$  to uint8 (unsigned) integers, and produce a staircase plot. Note that the horizontal axis plots from -1 to 1 and the vertical axis plots from 0 to 7 ( $2^3-1$ ):

```
u = [-1:0.01:1];
y = uencode(u,3);
plot(u,y, '.')
```



Now look at saturation effects when you under specify the peak value for the input:

```
u = [-2:0.5:2];
y = uencode(u,5,1)
```

# uencode

---

```
y =  
    0    0    0    8   16   24   31   31   31
```

Now look at the output for

```
u = [-2:0.5:2];  
y = uencode(u,5,2,'signed')  
  
y =  
   -16   -12    -8    -4     0     4     8    12    15
```

## Algorithm

uencode maps the floating-point input value to an integer value determined by the requirement for  $2^n$  levels of quantization. This encoding adheres to the definition for uniform encoding specified in ITU-T Recommendation G.701. The input range  $[-v, v]$  is divided into  $2^n$  evenly spaced intervals. Input entries in the range  $[-v, v]$  are first quantized according to this subdivision of the input range, and then mapped to one of  $2^n$  integers. The range of the output depends on whether or not you specify that you want signed integers.

## References

*General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms*, International Telecommunication Union, ITU-T Recommendation G.701, March, 1993.

## See Also

udecode



**Purpose**            Unwrap phase angles

**Description**        unwrap is a MATLAB function.

# upfirdn

---

**Purpose** Upsample, apply FIR filter, and downsample

**Syntax**

```
yout = upfirdn(xin,h)
yout = upfirdn(xin,h,p)
yout = upfirdn(xin,h,p,q)
```

**Description** upfirdn performs a cascade of three operations:

- 1** Upsampling the input data in the matrix `xin` by a factor of the integer `p` (inserting zeros)
- 2** FIR filtering the upsampled signal data with the impulse response sequence given in the vector or matrix `h`
- 3** Downsampling the result by a factor of the integer `q` (throwing away samples)

upfirdn has been implemented as a MEX-file for maximum speed, so only the outputs actually needed are computed. The FIR filter is usually a lowpass filter, which you must design using another function such as `firpm` or `fir1`.

---

**Note** The function `resample` performs an FIR design using `firls`, followed by rate changing implemented with `upfirdn`.

---

`yout = upfirdn(xin,h)` filters the input signal `xin` with the FIR filter having impulse response `h`. If `xin` is a row or column vector, then it represents a single signal. If `xin` is a matrix, then each column is filtered independently. If `h` is a row or column vector, then it represents one FIR filter. If `h` is a matrix, then each column is a separate FIR impulse response sequence. If `yout` is a row or column vector, then it represents one signal. If `yout` is a matrix, then each column is a separate output. No upsampling or downsampling is implemented with this syntax.

`yout = upfirdn(xin,h,p)` specifies the integer upsampling factor `p`, where `p` has a default value of 1.

`yout = upfirdn(xin,h,p,q)` specifies the integer downsampling factor `q`, where `q` has a default value of 1.

---

**Note** Since `upfirdn` performs convolution and rate changing, the `yout` signals have a different length than `xin`. The number of rows of `yout` is approximately  $p/q$  times the number of rows of `xin`.

---

## Remarks

Usually the inputs `xin` and the filter `h` are vectors, in which case only one output signal is produced. However, when these arguments are arrays, each column is treated as a separate signal or filter. Valid combinations are:

- 1** `xin` is a vector and `h` is a vector.  
There is one filter and one signal, so the function convolves `xin` with `h`. The output signal `yout` is a row vector if `xin` is a row; otherwise, `yout` is a column vector.
- 2** `xin` is a matrix and `h` is a vector.  
There is one filter and many signals, so the function convolves `h` with each column of `xin`. The resulting `yout` will be a matrix with the same number of columns as `xin`.
- 3** `xin` is a vector and `h` is a matrix.  
There are many filters and one signal, so the function convolves each column of `h` with `xin`. The resulting `yout` will be a matrix with the same number of columns as `h`.
- 4** `xin` is a matrix and `h` is a matrix, both with the same number of columns.  
There are many filters and many signals, so the function convolves corresponding columns of `xin` and `h`. The resulting `yout` is a matrix with the same number of columns as `xin` and `h`.

## Examples

Change the sampling rate by a factor of 147/160. This factor is used to convert from 48kHz (DAT rate) to 44.1kHz (CD sampling rate).

```
L = 147; M = 160;           % Interpolation/decimation factors.
N = 24*M;
h = fir1(N,1/M,kaiser(N+1,7.8562));
h = L*h; % Passband gain = L
Fs = 48e3;                 % Original sampling frequency-48kHz
n = 0:10239;               % 10240 samples, 0.213 seconds long
x = sin(2*pi*1e3/Fs*n);    % Original signal, sinusoid at 1kHz
```

# upfirdn

---

```
y = upfirdn(x,h,L,M);           % 9408 samples, still 0.213 seconds

% Overlay original (48kHz) with resampled signal (44.1kHz) in red.

stem(n(1:49)/Fs,x(1:49)); hold on
stem(n(1:45)/(Fs*L/M),y(13:57),'r','filled');
xlabel('Time (sec)');ylabel('Signal value');
```

## Algorithm

upfirdn uses a polyphase interpolation structure. The number of multiply-add operations in the polyphase structure is approximately  $(L_h L_x - p L_x) / q$  where  $L_h$  and  $L_x$  are the lengths of  $h[n]$  and  $x[n]$ , respectively.

A more accurate flops count is computed in the program, but the actual count is still approximate. For long signals  $x[n]$ , the formula is often exact.

## Diagnostics

If  $p$  and  $q$  are large and do not have many common factors, you may see this message:

```
Filter length is too large - reduce problem complexity.
```

Instead, you should use an interpolation function, such as `interp1`, to perform the resampling and then filter the input.

## References

- [1] Crochiere, R.E., and L.R. Rabiner, *Multi-Rate Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1983, pp. 88-91.
- [2] Crochiere, R.E., "A General Program to Perform Sampling Rate Conversion of Data by Rational Ratios," *Programs for Digital Signal Processing*, IEEE Press, New York, 1979, pp. 8.2-1 to 8.2-7.

## See Also

conv, decimate, downsample, filter, interp, intfilt, resample, upsample

**Purpose** Increase sampling rate by integer factor

**Syntax**  
`y = upsample(x,n)`  
`y = upsample(x,n,phase)`

**Description** `y = upsample(x,n)` increases the sampling rate of `x` by inserting `n-1` zeros between samples. `x` can be a vector or a matrix. If `x` is a matrix, each column is considered a separate sequence. The upsampled `y` has `x*n` samples.

`y = upsample(x,n,phase)` specifies the number of samples by which to offset the upsampled sequence. `phase` must be an integer from 0 to `n-1`.

**Examples** Increase the sampling rate of a sequence by 3:

```
x = [1 2 3 4];
y = upsample(x,3);
x,y

x =
    1     2     3     4

y =
    1     0     0     2     0     0     3     0     0     4     0     0
```

Increase the sampling rate of the sequence by 3 and add a phase offset of 2:

```
x = [1 2 3 4];
y = upsample(x,3,2);
x,y

x =
    1     2     3     4

y =
    0     0     1     0     0     2     0     0     3     0     0     4
```

Increase the sampling rate of a matrix by 3:

```
x = [1 2; 3 4; 5 6;];
y = upsample(x,3);
x,y
```

# upsample

---

```
x =  
  1  2  
  3  4  
  5  6
```

```
y =  
  1  2  
  0  0  
  0  0  
  3  4  
  0  0  
  0  0  
  5  6  
  0  0  
  0  0
```

## See Also

decimate, downsample, interp, interp1, resample, spline, upfirdn

**Purpose** Voltage controlled oscillator

**Syntax**  
`y = vco(x,fc,fs)`  
`y = vco(x,[Fmin Fmax],fs)`

**Description** `y = vco(x,fc,fs)` creates a signal that oscillates at a frequency determined by the real input vector or array `x` with sampling frequency `fs`. `fc` is the carrier or reference frequency; when `x` is 0, `y` is an `fc` Hz cosine with amplitude 1 sampled at `fs` Hz. `x` ranges from -1 to 1, where `x = -1` corresponds to 0 frequency output, `x = 0` corresponds to `fc`, and `x = 1` corresponds to `2*fc`. Output `y` is the same size as `x`.

`y = vco(x,[Fmin Fmax],fs)` scales the frequency modulation range so that  $\pm 1$  values of `x` yield oscillations of `Fmin` Hz and `Fmax` Hz respectively. For best results, `Fmin` and `Fmax` should be in the range 0 to `fs/2`.

By default, `fs` is 1 and `fc` is `fs/4`.

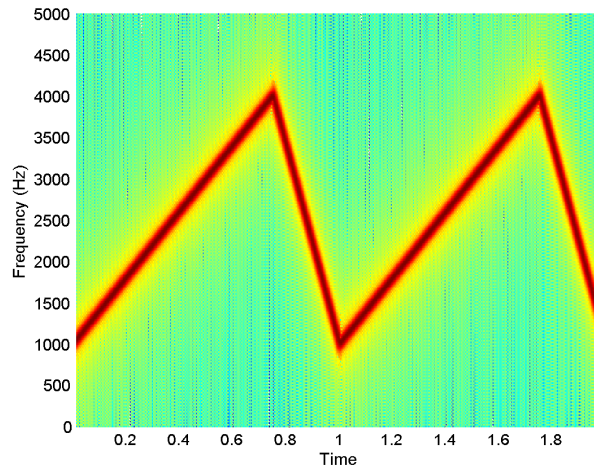
If `x` is a matrix, `vco` produces a matrix whose columns oscillate according to the columns of `x`.

**Examples** Generate two seconds of a signal sampled at 10,000 samples/second whose instantaneous frequency is a triangle function of time:

```
fs = 10000;  
t = 0:1/fs:2;  
x = vco(sawtooth(2*pi*t,0.75),[0.1 0.4]*fs,fs);
```

Plot the spectrogram of the generated signal:

```
spectrogram(x,kaiser(256,5),220,512,fs,'yaxis')
```



## Algorithm

vco performs FM modulation using the modulate function.

## Diagnostics

If any values of  $x$  lie outside  $[-1, 1]$ , vco gives the following error message.

X outside of range  $[-1,1]$ .

## See Also

demod, modulate



**Purpose** Window function gateway

**Syntax**

```
window  
w = window(fhandle,L)  
w = window(fhandle,L,winopt)
```

**Description** window opens the Window Design and Analysis Tool (`wintool`).

`w = window(fhandle,L)` returns the L-point window, specified by its function handle, `fhandle`, in column vector `w`. Function handles are window function names preceded by an @.

```
@barthannwin  
@bartlett  
@blackman  
@blackmanharris  
@bohmanwin  
@chebwin  
@flattopwin  
@gausswin  
@hamming  
@hann  
@kaiser  
@nuttallwin  
@parzenwin  
@rectwin  
@triang  
@tukeywin
```

---

**Note** For `chebwin`, `kaiser`, and `tukeywin`, you must use include a window parameter using the syntax below.

For more information on each window function and its option(s), refer to its reference page.

---

# window

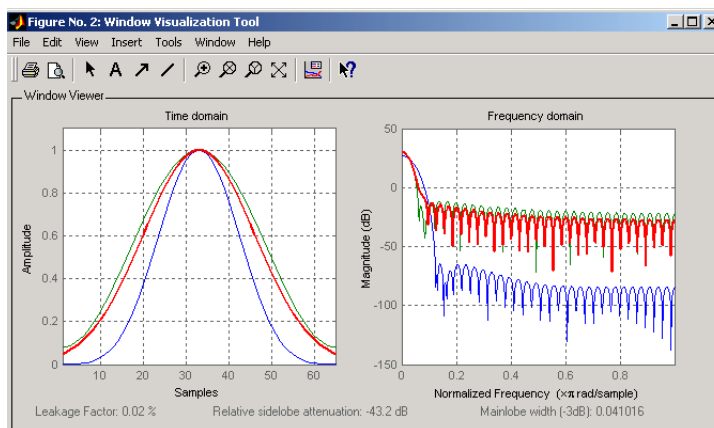
`w = window(fhandle,L,winopt)` returns the window specified by its function handle, `fhandle`, and its `winopt` value or sampling flag string. For `chebwin`, `kaiser`, and `tukeywin`, you must enter a `winopt` value. For the other windows listed below, `winopt` values are optional.

Window	winopt Description	winopt Value
blackman	sampling flag string	'periodic' or 'symmetric'
chebwin	sidelobe attenuation relative to mainlobe	numeric
flattopwin	sampling flag string	'periodic' or 'symmetric'
gausswin	alpha value (reciprocal of standard deviation)	numeric
hamming	sampling flag string	'periodic' or 'symmetric'
hann	sampling flag string	'periodic' or 'symmetric'
kaiser	beta value	numeric
tukeywin	ratio of taper to constant sections	numeric

## Examples

Create Blackman Harris, Hamming, and Gaussian windows and plot them in the same WVTool.

```
L = 65;  
w = window(@blackmanharris,L);  
w1 = window(@hamming,L);  
w2 = window(@gausswin,L,2.5);  
wvtool(w,w1,w2)
```

**See Also**

barthannwin, bartlett, blackman, blackmanharris, bohmanwin, chebwin, flattopwin, gausswin, hamming, hann, kaiser, nuttallwin, parzenwin, rectwin, triang, tukeywin

# wintool

---

**Purpose** Open Window Design and Analysis Tool

**Syntax**  
wintool  
wintool(obj1,obj2,...)

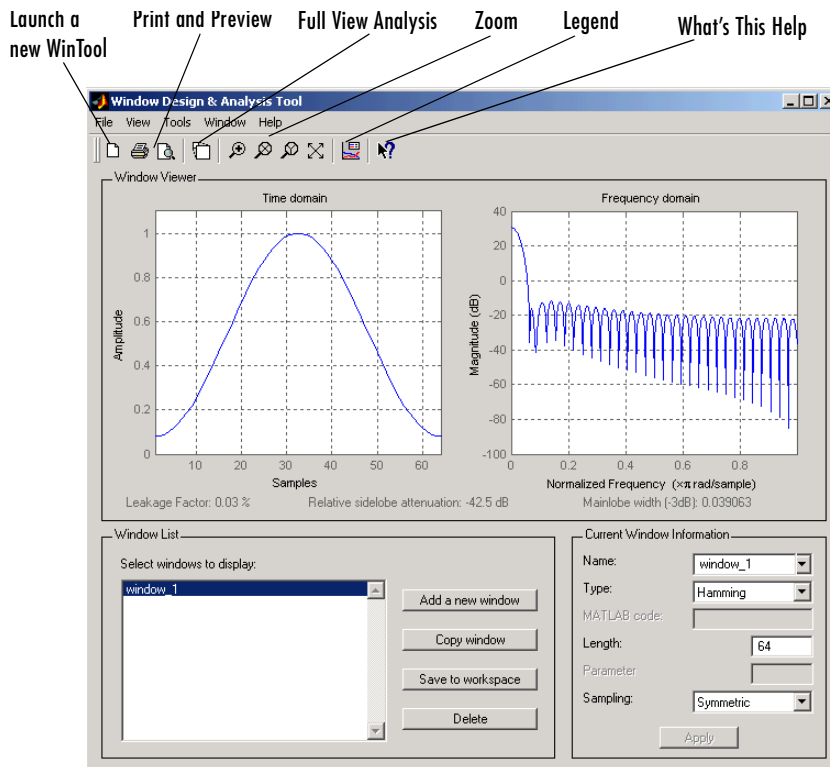
**Description** wintool opens the Window Design and Analysis Tool (WinTool), a graphical user interface (GUI) for designing and analyzing spectral windows. It opens with a default 64-point Hamming window.

wintool(obj1,obj2,...) opens WinTool with the sigwin window object(s) specified in obj1, obj2, etc.

---

**Note** A related tool, wvtool, is available for displaying, annotating, or printing windows.

---



wintool 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 in wintool 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 windows in the Signal Processing Toolbox 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`.

## WinTool Menus

In addition to the usual menu items, `wintool` contains these `wintool`-specific menu commands:

### File menu:

- **Export**—exports window coefficient vectors or `sigwin` window objects to the MATLAB workspace, a text file, or a MAT-file.

In the **Window List** in WinTool, highlight the window(s) you want to export and then select **Export** from the **File** menu. For exporting to the workspace or a MAT-file, specify the variable name for each window coefficient or object. 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 on 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	sec
kHz	ms
MHz	$\mu$ s
GHz	picosec

**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

window, wvtool



---

**Purpose** Open Window Visualization Tool

**Syntax**

```
wvtool(winname(n))  
wvtool(winname1(n),winname2(n),...winnamem(n))  
wvtool(w)  
h = wvtool(...)
```

**Description** `wvtool(winname(n))` opens the Window Visualization Tool (WVTool) with the time and frequency domain plots of the n-length window specified in `winname`, which can be any window in the Signal Processing Toolbox. For a list of valid window names, see the `window` function. In the `wvtool` command, do not precede the window name with `@`.

`wvtool(winname1(n),winname2(n),...winnamem(n))` opens WVTool with a time-domain plot and a frequency-domain plot that contain all the windows specified in `winname1,...winnamem`. The plots are shown on the same axes so that window characteristics can be compared and contrasted easily. WVTool is useful for displaying, annotating, and printing window responses.

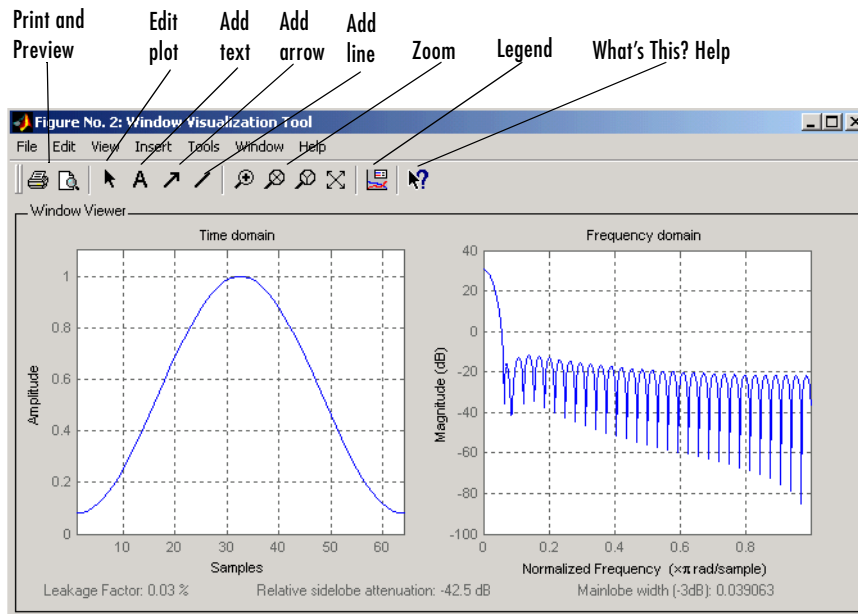
`wvtool(w)` launches the Window Visualization Tool with `sigwin` object `w`.

`h = wvtool(...)` returns the Handle Graphics figure handle `h`.

---

**Note** A related tool, `wintool`, is available for designing and analyzing windows.

---




---

**Note** If you launch WVTool from FDATool, an **Add/Replace** icon, which controls how new windows are added from FDATool, appears on the toolbar.

---

## WVTool Menus

In addition to the usual menu items, wvtool contains these wvtool-specific menu commands:

### File menu:

- **Export**—exports the displayed plot(s) to a graphic file.

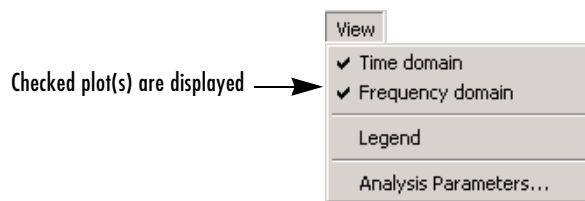
### Edit menu:

- **Copy figure**—copies the displayed plot(s) to the clipboard (available only on Windows platforms).

- **Copy options**—displays the **Preferences** dialog box (available only on Windows platforms).
- **Figure, Axes, and Current Object Properties**—displays the Property Editor.

**View** menu:

- **Time domain**—check to show the time domain plot.
- **Frequency domain**—check to show the frequency domain plot.



- **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 on the  $x$ -axis label of a plot in the Window Viewer panel.

• **Insert** menu:

You use the **Insert** menu to add labels, titles, arrows, lines, text, and axes to your plots.

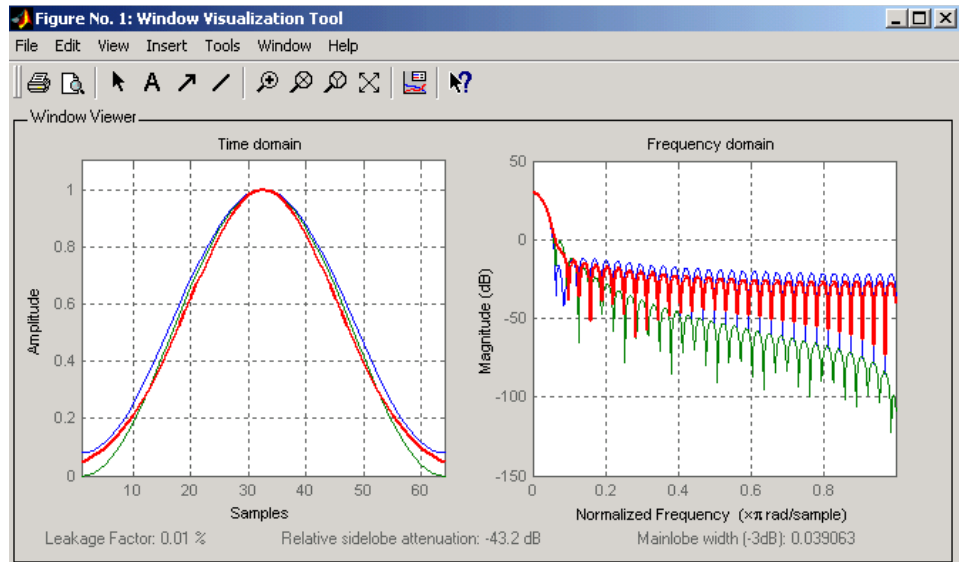
**Tools** menu:

- **Edit Plot**—turns on plot editing mode
- **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.

## Examples

Compare Hamming, Hann, and Gaussian windows:

```
wvtool(hamming(64),hann(64),gausswin(64))
```



## See Also

`fdatool`, `window`, `wintool`

**Purpose** Cross-correlation

**Syntax**

```

c = xcorr(x,y)
c = xcorr(x)
c = xcorr(x,y,'option')
c = xcorr(x,'option')
c = xcorr(x,y,maxlags)
c = xcorr(x,maxlags)
c = xcorr(x,y,maxlags,'option')
c = xcorr(x,maxlags,'option')
[c,lags] = xcorr(...)

```

**Description** xcorr estimates the cross-correlation sequence of a random process. Autocorrelation is handled as a special case.

The true cross-correlation sequence is

$$R_{xy}(m) = E\{x_{n+m}y_n^*\} = E\{x_n y_{n-m}^*\}$$

where  $x_n$  and  $y_n$  are jointly stationary random processes,  $-\infty < n < \infty$ , and  $E\{\cdot\}$  is the expected value operator. xcorr must estimate the sequence because, in practice, only a finite segment of one realization of the infinite-length random process is available.

`c = xcorr(x,y)` returns the cross-correlation sequence in a length  $2*N-1$  vector, where  $x$  and  $y$  are length  $N$  vectors ( $N>1$ ). If  $x$  and  $y$  are not the same length, the shorter vector is zero-padded to the length of the longer vector.

By default, xcorr computes raw correlations with no normalization.

$$\hat{R}_{xy}(m) = \begin{cases} \sum_{n=0}^{N-m-1} x_{n+m} y_n^* & m \geq 0 \\ \hat{R}_{yx}^*(-m) & m < 0 \end{cases}$$

The output vector `c` has elements given by  $c(m) = R_{xy}(m-N)$ ,  $m=1, \dots, 2N-1$ .

In general, the correlation function requires normalization to produce an accurate estimate (see below).

$c = \text{xcorr}(x)$  is the autocorrelation sequence for the vector  $x$ . If  $x$  is an  $N$ -by- $P$  matrix,  $c$  is a matrix with  $2N-1$  rows whose  $P^2$  columns contain the cross-correlation sequences for all combinations of the columns of  $x$ . For more information on matrix processing with `xcorr`, see “Multiple Channels” on page 3-5.

$c = \text{xcorr}(x,y,'option')$  specifies a normalization option for the cross-correlation, where *'option'* is

- *'biased'*: Biased estimate of the cross-correlation function

$$R_{xy,biased}(m) = \frac{1}{N}R_{xy}(m)$$

- *'unbiased'*: Unbiased estimate of the cross-correlation function

$$R_{xy,unbiased}(m) = \frac{1}{N-|m|}R_{xy}(m)$$

- *'coeff'*: Normalizes the sequence so the autocorrelations at zero lag are identically 1.0.
- *'none'*, to use the raw, unscaled cross-correlations (default)

See reference [1] for more information on the properties of biased and unbiased correlation estimates.

$c = \text{xcorr}(x,'option')$  specifies one of the above normalization options for the autocorrelation.

$c = \text{xcorr}(x,y,maxlags)$  returns the cross-correlation sequence over the lag range  $[-maxlags:maxlags]$ . Output  $c$  has length  $2*maxlags+1$ .

$c = \text{xcorr}(x,maxlags)$  returns the autocorrelation sequence over the lag range  $[-maxlags:maxlags]$ . Output  $c$  has length  $2*maxlags+1$ . If  $x$  is an  $N$ -by- $P$  matrix,  $c$  is a matrix with  $2*maxlags+1$  rows whose  $P^2$  columns contain the autocorrelation sequences for all combinations of the columns of  $x$ .

$c = \text{xcorr}(x,y,maxlags,'option')$  specifies both a maximum number of lags and a scaling option for the cross-correlation.

`c = xcorr(x,maxlags,'option')` specifies both a maximum number of lags and a scaling option for the autocorrelation.

`[c,lags] = xcorr(...)` returns a vector of the lag indices at which `c` was estimated, with the range `[-maxlags:maxlags]`. When `maxlags` is not specified, the range of lags is `[-N+1:N-1]`.

In all cases, the cross-correlation or autocorrelation computed by `xcorr` has the zeroth lag in the middle of the sequence, at element or row `maxlags+1` (element or row `N` if `maxlags` is not specified).

## Examples

The second output, `lags`, is useful for plotting the cross-correlation or autocorrelation. For example, the estimated autocorrelation of zero-mean Gaussian white noise  $c_{ww}(m)$  can be displayed for  $-10 \leq m \leq 10$  using:

```
ww = randn(1000,1);
[c_ww,lags] = xcorr(ww,10,'coeff');
stem(lags,c_ww)
```

Swapping the `x` and `y` input arguments reverses (and conjugates) the output correlation sequence. For row vectors, the resulting sequences are reversed left to right; for column vectors, up and down. The following example illustrates this property (`mat2str` is used for a compact display of complex numbers):

```
x = [1,2i,3]; y = [4,5,6];
[c1,lags] = xcorr(x,y);
c1 = mat2str(c1,2), lags
c1 =
    [6-i*8.9e-016 5+i*12 22+i*10 15+i*8 12+i*8.9e-016]
lags =
    -2    -1     0     1     2
c2 = conj(fliplr(xcorr(y,x)));
c2 = mat2str(c2,2)
c2 =
    [6-i*8.9e-016 5+i*12 22+i*10 15+i*8 12+i*8.9e-016]
```

For the case where input argument `x` is a matrix, the output columns are arranged so that extracting a row and rearranging it into a square array produces the cross-correlation matrix corresponding to the lag of the chosen row. For example, the cross-correlation at zero lag can be retrieved by:

```
randn('state',0)
```

## xcorr

---

```
X = randn(2,2);
[M,P] = size(X);
c = xcorr(X);
c0 = zeros(P); c0(:) = c(M,:) % Extract zero-lag row

c0 =
    2.9613   -0.5334
   -0.5334    0.0985
```

You can calculate the matrix of correlation coefficients that the MATLAB function `corrcoef` generates by substituting:

```
c = xcov(X, 'coef')
```

in the last example. The function `xcov` subtracts the mean and then calls `xcorr`.

Use `fftshift` to move the second half of the sequence starting at the zeroth lag to the front of the sequence. `fftshift` swaps the first and second halves of a sequence.

### Algorithm

For more information on estimating covariance and correlation functions, see [1].

### References

[1] Orfanidis, S.J., *Optimum Signal Processing. An Introduction. 2nd Edition*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

### See Also

`conv`, `corrcoef`, `cov`, `xcorr2`, `xcov`



**Purpose** 2-D cross-correlation

**Syntax**  
 $C = \text{xcorr2}(A)$   
 $C = \text{xcorr2}(A,B)$

**Description**  $C = \text{xcorr2}(A,B)$  returns the cross-correlation of matrices  $A$  and  $B$  with no scaling.  $\text{xcorr2}$  is the two-dimensional version of  $\text{xcorr}$ . It has its maximum value when the two matrices are aligned so that they are shaped as similarly as possible.

If matrix  $A$  has dimensions  $(Ma, Na)$  and matrix  $B$  has dimensions  $(Mb, Nb)$ , the equation for the two-dimensional discrete cross-correlation is

$$C(i,j) = \sum_{m=0}^{(Ma-1)(Na-1)} \sum_{n=0} A(m,n) \cdot \text{conj}(B(m+i,n+j))$$

where  $0 \leq i < Ma + Mb - 1$  and  $0 \leq j < Na + Nb - 1$ .

$\text{xcorr2}(A)$  is the autocorrelation matrix of input matrix  $A$ . It is identical to  $\text{xco}$

**Examples**

**Output Matrix Size**

If matrix  $I1$  has dimensions  $(4,3)$  and matrix  $I2$  has dimensions  $(2,2)$ , the following equations determine the number of rows and columns of the output matrix:

$$C_{\text{full\_rows}} = I1_{\text{rows}} + I2_{\text{rows}} - 1 = 4 + 2 - 1 = 5$$

$$C_{\text{full\_columns}} = I1_{\text{columns}} + I2_{\text{columns}} - 1 = 3 + 2 - 1 = 4$$

The resulting matrix is

$$C_{\text{full}} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \\ c_{40} & c_{41} & c_{42} & c_{43} \end{bmatrix}$$

## Computing a Specific Element

$$C_{\text{valid}_{\text{columns}}} = I1_{\text{columns}} - I2_{\text{columns}} + 1 = 2$$

In cross-correlation, the value of an output element is computed as a weighted sum of neighboring elements. For example, suppose the first input matrix represents an image and is defined as

$$I1 = \begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

The second input matrix also represents an image and is defined as

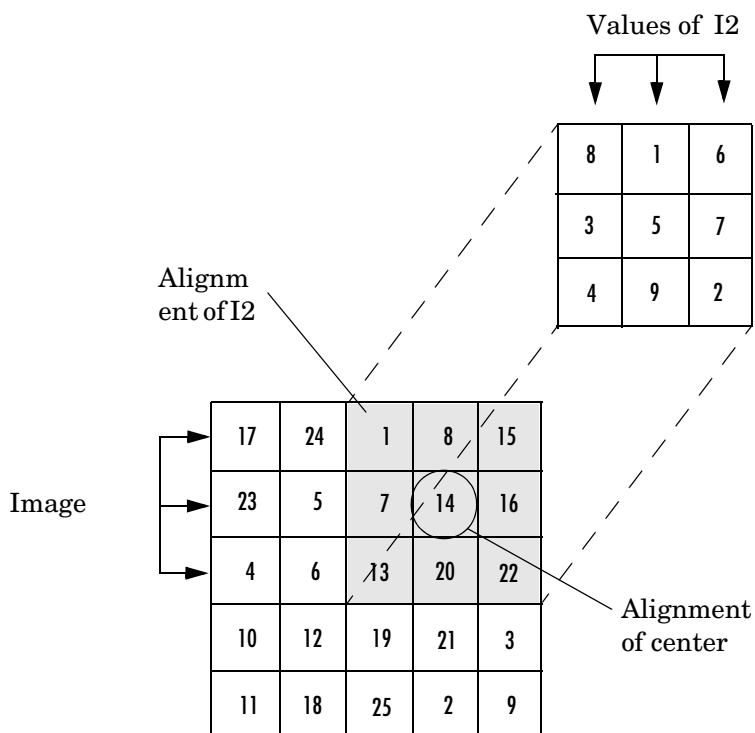
$$I2 = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

The following figure shows how to compute the (2,4) output element (zero-based indexing) using these steps:

- 1** Slide the center element of I2 so that lies on top of the (1,3) element of I1.
- 2** Multiply each weight in I2 by the element of I1 underneath.
- 3** Sum the individual products from step 2.

The (2,4) output element from the cross-correlation is

$$1 \cdot 8 + 8 \cdot 1 + 15 \cdot 6 + 7 \cdot 3 + 14 \cdot 5 + 16 \cdot 7 + 13 \cdot 4 + 20 \cdot 9 + 22 \cdot 2 = 585$$



The normalized cross-correlation of the (2,4) output element is

$$585/\text{sqrt}(\text{sum}(\text{dot}(I1p,I1p))*\text{sum}(\text{dot}(I2,I2))) = 0.8070$$

where  $I1p = [1\ 8\ 15; 7\ 14\ 16; 13\ 20\ 22]$ .

**See Also**

`conv2`, `filter2`, `xcorr`

**Purpose** Cross-covariance

**Syntax**

```
v = xcov(x,y)
v = xcov(x)
v = xcov(x,'option')
[c,lags] = xcov(x,y,maxlags)
[c,lags] = xcov(x,maxlags)
[c,lags] = xcov(x,y,maxlags,'option')
```

**Description** xcov estimates the cross-covariance sequence of random processes. Autocovariance is handled as a special case.

The true cross-covariance sequence is the cross-correlation of mean-removed sequences

$$\phi_{xy}(\mu) = E\{(x_{n+m} - \mu_x)(y_n - \mu_y)^*\}$$

where  $\mu_x$  and  $\mu_y$  are the mean values of the two stationary random processes, and  $E\{\cdot\}$  is the expected value operator. xcov estimates the sequence because, in practice, access is available to only a finite segment of the infinite-length random process.

$v = \text{xcov}(x,y)$  returns the cross-covariance sequence in a length  $2N-1$  vector, where  $x$  and  $y$  are length  $N$  vectors. For information on how arrays are processed with xcov, see “Multiple Channels” on page 3-5.

$v = \text{xcov}(x)$  is the autocovariance sequence for the vector  $x$ . Where  $x$  is an  $N$ -by- $P$  array,  $v = \text{xcov}(x)$  returns an array with  $2N-1$  rows whose  $P^2$  columns contain the cross-covariance sequences for all combinations of the columns of  $x$ .

By default, xcov computes raw covariances with no normalization. For a length  $N$  vector

$$c_{xy}(m) = \begin{cases} \sum_{n=0}^{N-|m|-1} \left( x(n+m) - \frac{1}{N} \sum_{i=0}^{N-1} x_i \right) \left( y_n^* - \frac{1}{N} \sum_{i=0}^{N-1} y_i^* \right) & m \geq 0 \\ c_{yx}^*(-m) & m < 0 \end{cases}$$

The output vector  $c$  has elements given by  $c(m) = c_{xy}(m-N)$ ,  $m = 1, \dots, 2N-1$ .

The covariance function requires normalization to estimate the function properly.

$v = \text{xcov}(x, 'option')$  specifies a scaling option, where *'option'* is

- *'biased'*, for biased estimates of the cross-covariance function
- *'unbiased'*, for unbiased estimates of the cross-covariance function
- *'coeff'*, to normalize the sequence so the auto-covariances at zero lag are identically 1.0
- *'none'*, to use the raw, unscaled cross-covariances (default)

See [1] for more information on the properties of biased and unbiased correlation and covariance estimates.

$[c, \text{lags}] = \text{xcov}(x, y, \text{maxlags})$  where  $x$  and  $y$  are length  $m$  vectors, returns the cross-covariance sequence in a length  $2*\text{maxlags}+1$  vector  $c$ .  $\text{lags}$  is a vector of the lag indices where  $c$  was estimated, that is,  $[-\text{maxlags}:\text{maxlags}]$ .

$[c, \text{lags}] = \text{xcov}(x, \text{maxlags})$  is the autocovariance sequence over the range of lags  $[-\text{maxlags}:\text{maxlags}]$ .

$[c, \text{lags}] = \text{xcov}(x, \text{maxlags})$  where  $x$  is an  $m$ -by- $p$  array, returns array  $c$  with  $2*\text{maxlags}+1$  rows whose  $P^2$  columns contain the cross-covariance sequences for all combinations of the columns of  $x$ .

$[c, \text{lags}] = \text{xcov}(x, y, \text{maxlags}, 'option')$  specifies a scaling option, where *'option'* is the last input argument.

In all cases,  $\text{xcov}$  gives an output such that the zeroth lag of the covariance vector is in the middle of the sequence, at element or row  $\text{maxlag}+1$  or at  $m$ .

## Examples

The second output  $\text{lags}$  is useful when plotting. For example, the estimated autocovariance of uniform white noise  $c_{ww}(m)$  can be displayed for  $-10 \leq m \leq 10$  using:

```
ww = randn(1000,1); % Generate uniform noise with mean = 1/2.
[cov_ww, lags] = xcov(ww,10, 'coeff');
stem(lags, cov_ww)
```

## **xCOV**

---

### **Algorithm**

xcov computes the mean of its inputs, subtracts the mean, and then calls xcorr. For more information on estimating covariance and correlation functions, see [1].

### **Diagnostics**

xcov does not check for any errors other than the correct number of input arguments. Instead, it relies on the error checking in xcorr.

### **References**

[1] Orfanidis, S.J., *Optimum Signal Processing. An Introduction. 2nd Edition*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

### **See Also**

conv, corrcoef, cov, xcorr, xcorr2

**Purpose** Recursive digital filter design

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

**Description** `yulewalk` designs recursive IIR digital filters using a least-squares fit to a specified frequency response.

`[b,a] = yulewalk(n,f,m)` returns row vectors `b` and `a` containing the  $n+1$  coefficients of the order  $n$  IIR filter whose frequency-magnitude characteristics approximately match those given in vectors `f` and `m`:

- `f` is a vector of frequency points, specified in the range between 0 and 1, where 1 corresponds to half the sample frequency (the Nyquist frequency). The first point of `f` must be 0 and the last point 1, with all intermediate points in increasing order. Duplicate frequency points are allowed, corresponding to steps in the frequency response.
- `m` is a vector containing the desired magnitude response at the points specified in `f`.
- `f` and `m` must be the same length.
- `plot(f,m)` displays the filter shape.

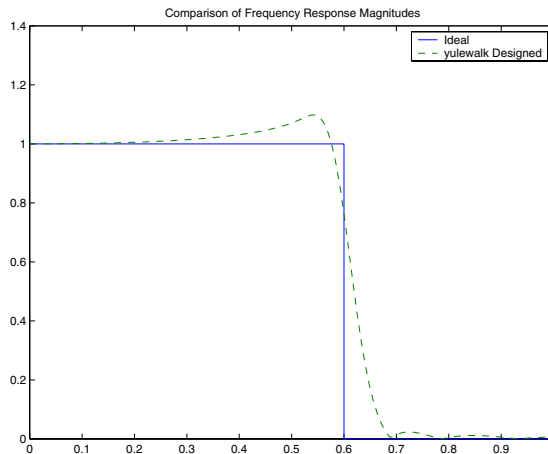
The output filter coefficients are ordered in descending powers of  $z$ .

$$\frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \dots + a(n+1)z^{-n}}$$

When specifying the frequency response, avoid excessively sharp transitions from passband to stopband. You may need to experiment with the slope of the transition region to get the best filter design.

**Examples** Design an 8th-order lowpass filter and overplot the desired frequency response with the actual frequency response:

```
f = [0 0.6 0.6 1];
m = [1 1 0 0];
[b,a] = yulewalk(8,f,m);
[h,w] = freqz(b,a,128);
plot(f,m,w/pi,abs(h),'--')
legend('Ideal','yulewalk Designed')
title('Comparison of Frequency Response Magnitudes')
```



## Algorithm

yulewalk performs a least-squares fit in the time domain. It computes the denominator coefficients using modified Yule-Walker equations, with correlation coefficients computed by inverse Fourier transformation of the specified frequency response. To compute the numerator, yulewalk takes the following steps:

- 1 Computes a numerator polynomial corresponding to an additive decomposition of the power frequency response.
- 2 Evaluates the complete frequency response corresponding to the numerator and denominator polynomials.
- 3 Uses a spectral factorization technique to obtain the impulse response of the filter.
- 4 Obtains the numerator polynomial by a least-squares fit to this impulse response.

## References

[1] Friedlander, B., and B. Porat, "The Modified Yule-Walker Method of ARMA Spectral Estimation," *IEEE Transactions on Aerospace Electronic Systems*, AES-20, No. 2 (March 1984), pp. 158-173.

## See Also

butter, cheby1, cheby2, ellip, fir2, firls, maxflat, firpm



**Purpose** Zero-phase response of digital filter

**Syntax**

```
[Hr,w] = zerophase(b,a)
[Hr,w] = zerophase(b,a,nfft)
[Hr,w] = zerophase(b,a,nfft,'whole')
[Hr,w] = zerophase(b,a,w)
[Hr,f] = zerophase(...,fs)
[Hr,w,phi] = zerophase(...)
zerophase(...)
```

**Description** [Hr,w] = zerophase(b,a) returns the zero-phase response Hr, and the frequency vector w (in radians/sample) at which Hr is computed, given a filter defined by numerator b and denominator a. For FIR filters where a=1, you can omit the value a from the command. The zero-phase response is evaluated at 512 equally spaced points on the upper half of the unit circle.

The zero-phase response,  $Hr(w)$ , is related to the frequency response,  $H(w)$  by

$$H(e^{j\omega}) = Hr(\omega) e^{j\varphi(\omega)},$$

where  $H(e^{j\omega})$  is the frequency response,  $Hr(\omega)$  is the zero-phase response and  $\varphi(\omega)$  is the continuous phase.

---

**Note** The zero-phase response is always real, but it is not the equivalent of the magnitude response. The former can be negative while the latter cannot be negative.

---

[Hr,w] = zerophase(b,a,nfft) returns the zero-phase response Hr and frequency vector w (radians/sample), using nfft frequency points on the upper half of the unit circle.

[Hr,w] = zerophase(b,a,nfft,'whole') returns the zero-phase response Hr and frequency vector w (radians/sample), using nfft frequency points around the whole unit circle.

[Hr,w] = zerophase(b,a,w) returns the zero-phase response Hr and frequency vector w (radians/sample) at frequencies in vector w.

# zerophase

---

`[Hr,f] = zerophase(...,fs)` returns the zero-phase response `Hr` and frequency vector `f` (Hz), using the sampling frequency `fs` (in Hz), to determine the frequency vector `f` (in Hz) at which `Hr` is computed.

`[Hr,w,phi] = zerophase(...)` returns the zero-phase response `Hr`, frequency vector `w` (rad/sample), and the continuous phase component, `phi`. (Note that this quantity is not equivalent to the phase response of the filter when the zero-phase response is negative.)

`zerophase(...)` with no output arguments, plots the zero-phase response versus frequency.

## Examples

### Example 1

Plot the zero-phase response of a constrained least squares FIR filter:

```
b=fircls1(54,.3,.02,.008);  
zerophase(b);
```

### Example 2

Plot the zero-phase response of an elliptic filter:

```
[b,a]=ellip(10,.5,20,.4);  
zerophase(b,a,512,'whole');
```

## See Also

`freqs`, `freqz`, `fvtool`, `grpdelay`, `invfreqz`, `phasedelay`, `phasez`

**Purpose** Convert zero-pole-gain filter parameters to second-order sections form

**Syntax**

```
[ sos, g ] = zp2sos( z, p, k )
[ sos, g ] = zp2sos( z, p, k, 'order' )
[ sos, g ] = zp2sos( z, p, k, 'order', 'scale' )
[ sos, g ] = zp2sos( z, p, k, 'order', 'scale', zeroflag )
```

```
sos = zp2sos( ... )
```

**Description** zp2sos converts a discrete-time zero-pole-gain representation of a given digital filter to an equivalent second-order section representation.

Use `[ sos, g ] = zp2sos( z, p, k )` to obtain a matrix `sos` in second-order section form with gain `g` equivalent to the discrete-time zero-pole-gain filter represented by input arguments `z`, `p`, and `k`. Vectors `z` and `p` contain the zeros and poles of the filter's transfer function  $H(z)$ , not necessarily in any particular order.

$$H(z) = k \frac{(z - z_1)(z - z_2) \cdots (z - z_n)}{(z - p_1)(z - p_2) \cdots (z - p_m)}$$

where  $n$  and  $m$  are the lengths of `z` and `p`, respectively, and `k` is a scalar gain. The zeros and poles must be real or complex conjugate pairs. `sos` is an  $L$ -by-6 matrix

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

whose rows contain the numerator and denominator coefficients  $b_{ik}$  and  $a_{ik}$  of the second-order sections of  $H(z)$ .

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

The number  $L$  of rows of the matrix `sos` is the closest integer greater than or equal to the maximum of  $n/2$  and  $m/2$ .

`[ sos, g ] = zp2sos( z, p, k, 'order' )` specifies the order of the rows in `sos`, where `'order'` is

- `'down'`, to order the sections so the first row of `sos` contains the poles closest to the unit circle
- `'up'`, to order the sections so the first row of `sos` contains the poles farthest from the unit circle (default)

`[ sos, g ] = zp2sos( z, p, k, 'order', 'scale' )` specifies the desired scaling of the gain and the numerator coefficients of all second-order sections, where `'scale'` is

- `'none'`, to apply no scaling (default)
- `'inf'`, to apply infinity-norm scaling
- `'two'`, to apply 2-norm scaling

Using infinity-norm scaling in conjunction with up-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling in conjunction with down-ordering minimizes the peak round-off noise.

---

**Note** Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

---

`[ sos, g ] = zp2sos( z, p, k, 'order', 'scale', zeroflag )` specifies whether to keep together real zeros that are the negatives of each other instead of ordering them according to proximity to poles. Setting `zeroflag` to true keeps the zeros together and results in a numerator with a middle coefficient equal to zero. The default for `zeroflag` is false.

`sos = zp2sos(...)` embeds the overall system gain,  $g$ , in the first section,  $H_1(z)$ , so that

$$H(z) = \prod_{k=1}^L H_k(z)$$

---

**Note** Embedding the gain in the first section when scaling a direct-form II structure is not recommended and may result in erratic scaling. To avoid embedding the gain, use `ss2sos` with two outputs.

---

## Examples

Find a second-order section form of a Butterworth lowpass filter:

```
[z,p,k] = butter(5,0.2);
sos = zp2sos(z,p,k);
```

## Algorithm

`zp2sos` uses a four-step algorithm to determine the second-order section representation for an input zero-pole-gain system:

- 1 It groups the zeros and poles into complex conjugate pairs using the `cplxpair` function.
- 2 It forms the second-order section by matching the pole and zero pairs according to the following rules:
  - a Match the poles closest to the unit circle with the zeros closest to those poles.
  - b Match the poles next closest to the unit circle with the zeros closest to those poles.
  - c Continue until all of the poles and zeros are matched.

`zp2sos` groups real poles into sections with the real poles closest to them in absolute value. The same rule holds for real zeros.
- 3 It orders the sections according to the proximity of the pole pairs to the unit circle. `zp2sos` normally orders the sections with poles closest to the unit circle last in the cascade. You can tell `zp2sos` to order the sections in the reverse order by specifying the down flag.
- 4 `zp2sos` scales the sections by the norm specified in the `'scale'` argument. For arbitrary  $H(\omega)$ , the scaling is defined by

$$\|H\|_p = \left[ \frac{1}{2\pi} \int_0^{2\pi} |H(\omega)|^p d\omega \right]^{\frac{1}{p}}$$

where  $p$  can be either  $\infty$  or 2. See the references for details on the scaling. This scaling is an attempt to minimize overflow or peak round-off noise in fixed point filter implementations.

## References

- [1] Jackson, L.B., *Digital Filters and Signal Processing*, 3rd ed., Kluwer Academic Publishers, Boston, 1996, Chapter 11.
- [2] Mitra, S.K., *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, 1998, Chapter 9.
- [3] Vaidyanathan, P.P., "Robust Digital Filter Structures," *Handbook for Digital Signal Processing*, S.K. Mitra and J.F. Kaiser, ed., John Wiley & Sons, New York, 1993, Chapter 7.

## See Also

cplxpair, filternorm, sos2zp, ss2sos, tf2sos, zp2ss, zp2tf

**Purpose** Convert zero-pole-gain filter parameters to state-space form

**Syntax** [A,B,C,D] = zp2ss(z,p,k)

**Description** zp2ss converts a zero-pole-gain representation of a given system to an equivalent state-space representation.

[A,B,C,D] = zp2ss(z,p,k) finds a single input, multiple output, state-space representation

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

given a system in factored transfer function form.

$$H(s) = \frac{Z(s)}{P(s)} = k \frac{(s - z_1)(s - z_2) \cdots (s - z_n)}{(s - p_1)(s - p_2) \cdots (s - p_n)}$$

Column vector p specifies the pole locations, and matrix z the zero locations with as many columns as there are outputs. The gains for each numerator transfer function are in vector k. The A, B, C, and D matrices are returned in controller canonical form.

Inf values may be used as place holders in z if some columns have fewer zeros than others.

**Algorithm** zp2ss, for single-input systems, groups complex pairs together into two-by-two blocks down the diagonal of the A matrix. This requires the zeros and poles to be real or complex conjugate pairs.

**See Also** sos2ss, ss2zp, tf2ss, zp2sos, zp2tf

# zp2tf

---

**Purpose** Convert zero-pole-gain filter parameters to transfer function form

**Syntax** [b,a] = zp2tf(z,p,k)

**Description** zp2tf forms transfer function polynomials from the zeros, poles, and gains of a system in factored form.

[b,a] = zp2tf(z,p,k) finds a rational transfer function

$$\frac{B(s)}{A(s)} = \frac{b_1 s^{(n-1)} + \dots + b_{(n-1)} s + b_n}{a_1 s^{(m-1)} + \dots + a_{(m-1)} s + a_m}$$

given a system in factored transfer function form

$$H(s) = \frac{Z(s)}{P(s)} = k \frac{(s-z_1)(s-z_2)\cdots(s-z_m)}{(s-p_1)(s-p_2)\cdots(s-p_n)}$$

Column vector **p** specifies the pole locations, and matrix **z** specifies the zero locations, with as many columns as there are outputs. The gains for each numerator transfer function are in vector **k**. The zeros and poles must be real or come in complex conjugate pairs. The polynomial denominator coefficients are returned in row vector **a** and the polynomial numerator coefficients are returned in matrix **b**, which has as many rows as there are columns of **z**.

Inf values can be used as place holders in **z** if some columns have fewer zeros than others.

**Algorithm** The system is converted to transfer function form using poly with **p** and the columns of **z**.

**See Also** sos2tf, ss2tf, tf2zp, tf2zpk, zp2sos, zp2ss



**Purpose** Zero-pole plot

**Syntax**

```
zplane(z,p)
zplane(b,a)
zplane(Hd)
[hz, hp, ht] = zplane(z,p)
```

**Description** This function displays the poles and zeros of discrete-time systems.

`zplane(z,p)` plots the zeros specified in column vector `z` and the poles specified in column vector `p` in the current figure window. The symbol 'o' represents a zero and the symbol 'x' represents a pole. The plot includes the unit circle for reference. If `z` and `p` are arrays, `zplane` plots the poles and zeros in the columns of `z` and `p` in different colors.

You can override the automatic scaling of `zplane` using

```
axis([xmin xmax ymin ymax])
```

or

```
set(gca, 'ylim', [ymin ymax])
```

or

```
set(gca, 'xlim', [xmin xmax])
```

after calling `zplane`. This is useful in the case where one or a few of the zeros or poles have such a large magnitude that the others are grouped tightly around the origin and are hard to distinguish.

`zplane(b,a)` where `b` and `a` are row vectors, first uses roots to find the zeros and poles of the transfer function represented by numerator coefficients `b` and denominator coefficients `a`.

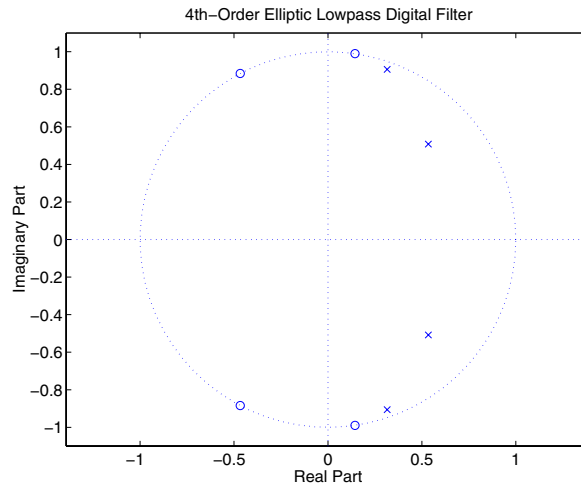
`zplane(Hd)` finds the zeros and poles of the transfer function represented by the `dfilt` filter object `Hd`. The pole-zero plot is displayed in `fvtool`.

`[hz, hp, ht] = zplane(z,p)` returns vectors of handles to the zero lines, `hz`, and the pole lines, `hp`. `ht` is a vector of handles to the axes/unit circle line and to text objects, which are present when there are multiple zeros or poles. If there are no zeros or no poles, `hz` or `hp` is the empty matrix `[]`.

## Examples

For data sampled at 1000 Hz, plot the poles and zeros of a 4th-order elliptic lowpass digital filter with cutoff frequency of 200 Hz, 3 dB of ripple in the passband, and 30 dB of attenuation in the stopband:

```
[z,p,k] = ellip(4,3,30,200/500);  
zplane(z,p);  
title('4th-Order Elliptic Lowpass Digital Filter');
```

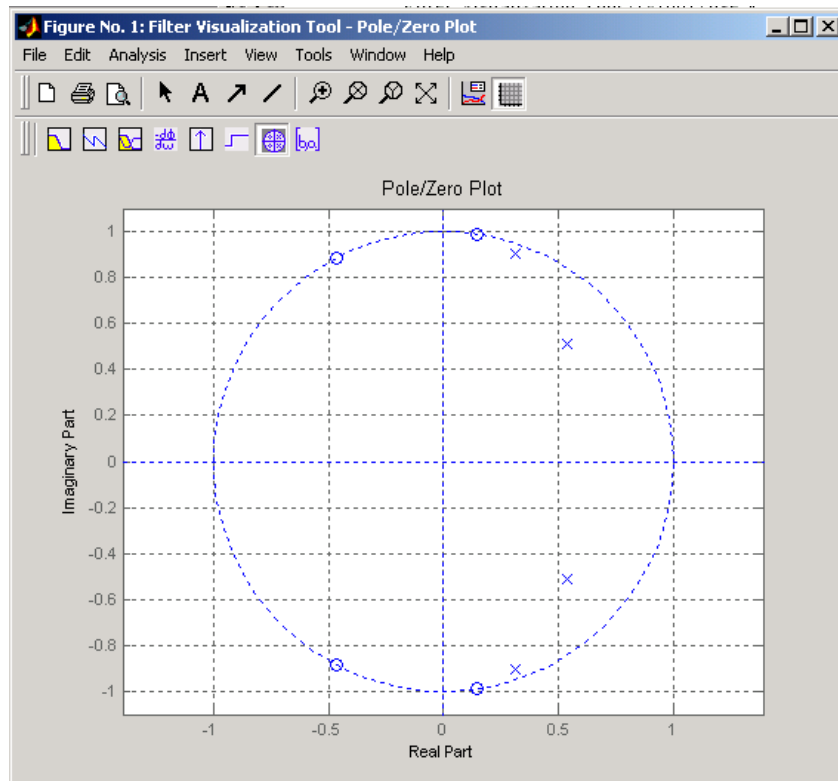


To generate the same plot with a transfer function representation of the filter, use:

```
[b,a] = ellip(4,3,30,200/500); % Transfer function  
zplane(b,a)
```

To generate the same plot using a `dfilt` object and displaying the result in the Filter Visualization Tool (`fvtool`) use:

```
[b,a] = ellip(4,3,30,200/500);  
Hd=dfilt.df1(b,a);  
zplane(Hd)
```



**See Also**

freqz



# Technical Conventions

---

This manual and the Signal Processing Toolbox functions use the following technical notations.

<b>Term or Symbol</b>	<b>Description</b>
Nyquist frequency	One-half the sampling frequency. Some toolbox functions normalize this value to 1.
$x(1)$	The first element of a data sequence or filter, corresponding to zero lag.
$\Omega$ or $w$	Analog frequency in radians per second.
$\omega$ or $w$	Digital frequency in radians per sample.
$f$	Digital frequency in hertz.
$[x, y)$	The interval from $x$ to $y$ , including $x$ but not including $y$ .
...	Ellipses in the argument list for a given syntax on a function reference page indicate all possible argument lists for that function appearing prior to the given syntax are valid.



**Numerics**

2-norm 7-269

**A**

A/D conversion 7-638

abs function 7-19

ac2poly function 7-20

ac2rc function 7-21

addstages method 7-140

aliasing

impulse invariance 2-47

preventing 4-26

reducing 4-46

sinc functions 1-13

all-pole filters

*See* IIR filters

all-zero filters

*See* FIR filters

AM

*See* amplitude modulation

amdsb function 7-414

amplitude demodulation 7-134

amplitude modulation 7-414

analog filters 2-43

bandpass 7-397

bandstop 7-400

Bessel 7-32

Bessel comparison 2-13

Bessel lowpass 7-31

bilinear transformation 2-49

Butterworth 7-57

Butterworth comparison 2-9

Butterworth lowpass 7-56

Butterworth order estimation 7-63

Chebyshev Type I 7-93

Chebyshev Type I comparison 2-10

Chebyshev Type I lowpass 7-79

Chebyshev Type I order estimation 7-81

Chebyshev Type II 7-99

Chebyshev Type II comparison 2-11

Chebyshev Type II order estimation 7-87

converting to digital 7-364

design 2-7

discretization 2-46

elliptic 7-246

elliptic order estimation 7-254

frequency response 7-311

frequency response example 2-14

highpass 7-402

impulse invariance 2-47

inverse 7-376

lowpass 7-404

models 1-44

plotting 2-14

*See also* IIR filters

analog frequency A-1

analog signals

*See* signals

analysis parameters 7-322

analytic signals 7-357

angle function 7-22

anti-symmetric filters 2-27

AR filter stability 7-478

AR models

*See* autoregressive (AR) models

arburg function 7-23

arcov function 7-24

ARMA filters

coefficients 1-16

Prony's method 4-18

Steiglitz-McBride method 4-20

*See also* IIR filters

armcov function 7-25  
ARX models 4-18  
aryule function 7-26  
ASCII files  
    importing 1-14  
autocorrelation 7-662  
    convert from LP coefficients 7-476  
    convert from reflection coefficients 7-500  
    convert to LP coefficients 7-20  
    convert to reflection coefficients 7-21  
    estimation 3-4  
    multiple channel filters 3-5  
    two-dimensional 7-665  
    variance 3-4  
autocovariance 7-668  
    multiple channels 3-5  
autoregressive (AR) models  
    Burg method 7-23  
    coefficients 1-16  
    covariance method 7-24  
    modified covariance method 7-25  
    power spectral density (Burg method) 7-426  
    power spectral density (covariance method)  
        7-431  
    power spectral density (modified covariance  
        method) 7-456  
    power spectral density (Yule-Walker method)  
        7-494  
    Yule-Walker method 7-26  
    *See also* IIR filters  
autoregressive moving-average (ARMA) filters  
    *See* ARMA filters  
avgpower method 7-221

**B**

band edges  
    prewarping 2-50  
bandpass filters  
    Butterworth 7-57  
    Chebyshev Type I 7-93  
    Chebyshev Type I example 2-45  
    Chebyshev Type II 7-98  
    design 2-6  
    elliptic 7-246  
    FIR 2-22  
    FIR example 7-279  
    impulse invariance 2-47  
    transform from lowpass 7-397  
bandstop filters  
    Butterworth 7-58  
    Chebyshev Type I 7-94  
    Chebyshev Type II 7-99  
    elliptic 7-247  
    FIR 7-278  
    transform from lowpass 7-400  
bandwidth 2-45  
barthannwin Bartlett Hann window function 7-27  
    comparison 4-2  
bartlett window function 7-29  
    comparison 4-2  
Bessel filters  
    characteristics 2-13  
    limitations 7-33  
    lowpass 7-31  
    prototype 7-31  
besselap function 7-31  
besself function 7-32



- bias 3-4
    - linear prediction 4-17
    - power spectral density 3-18
    - variance trade-off 3-4
    - Welch 3-26
  - bilinear function 7-34
  - bilinear transformations 7-34
    - characteristics 2-49
    - output 7-35
    - prewarping 7-34
    - prewarping example 2-50
  - bit reversal 7-39
  - bitrevorder function 7-39
  - blackman window function 7-41
    - comparison 4-2
  - blackmanharris window function 7-43
    - comparison 4-2
    - Nuttall 7-422
  - block method 7-140
  - bohmanwin window function 7-45
    - comparison 4-2
  - boxcar windows
    - See* rectangular windows
  - brackets A-1
  - buffer function 7-47
  - Burg method
    - characteristics 3-40
    - example 3-41
    - spectral estimation 3-8
    - Welch's method comparison 3-43
  - Burg spectrum object 7-561
  - buttap function 7-56
  - butter function 7-57
  - Butterworth filters 7-57
    - characteristics 2-9
    - comparison 2-9
    - generalized 2-15
    - limitations 7-61
    - lowpass 7-56
    - order estimation 7-62
  - buttord function 7-62
- ## C
- C header files 5-44
  - canonical forms 1-18
    - naming conventions 7-621
  - carrier frequencies 4-34
  - cascade method 7-140
  - cascades 1-40
  - Cauer filters
    - See* elliptic filters
  - cceps function 7-67
  - cell2sos function 7-69
  - center frequency 2-45
  - centerdc method 7-221
  - cepstrum 4-28
    - inverse function 7-504
  - cfirpm function 7-70
  - cheb1ap function 7-79
  - cheb1ord function 7-80
  - cheb2ap function 7-85
  - cheb2ord function 7-86
  - chebwin Chebyshev window function 7-91
    - comparison 4-2
  - cheby1 function 7-93
  - cheby2 function 7-98
  - Chebyshev error minimization 7-297
  - Chebyshev Type I filters 7-93
    - characteristics 2-10
    - example 2-45
    - lowpass 7-79
    - order estimation 7-80

- Chebyshev Type II filters 7-98
  - characteristics 2-11
  - limitations 7-97
  - order estimation 2-7
- chirp function 7-103
- chirp signals 1-10
- chirp z-transforms 7-125
  - characteristics 4-42
- CIC filters
  - exporting from FDATool to Simulink 5-40
- coding
  - PCM 7-638
- coefficients
  - accessing filter 6-29, 6-49
  - convert autocorrelation to filter 7-20
  - convert filter to autocorrelation 7-476
  - convert filter to reflection 7-478
  - convert reflection to autocorrelation 7-500
  - convert reflection to filter 7-503
  - filter 1-16
  - lattice 1-40
  - linear prediction 7-406
  - reflection 7-21
- coefficients method 7-140
- coherence 7-417
  - definition 3-32
  - linearly dependent data 3-33
- colors
  - sptool GUI 6-42
- communications
  - modeling 4-15
  - simulation 7-134
  - See also* demodulation, modulation, voltage controlled oscillation
- compaction
  - discrete cosine transform 4-45
- complex envelope
  - See* Hilbert transforms
- context-sensitive help 6-6
- continuous signals
  - See* signals
- continuous-time filters
  - See* analog filters
- conv function 7-110
- conv2 function 7-111
- conversions
  - autocorrelation to filter coefficients 7-20
  - autocorrelation to reflection coefficients 7-21
  - errors 5-29
  - filter coefficients to autocorrelation 7-476
  - filter coefficients to reflection coefficients 7-478
  - functions (table) 1-45
  - reflection coefficients to autocorrelation 7-500
  - reflection coefficients to filter coefficients 7-503
  - second-order section to zero-pole-gain 7-543
  - second-order sections to state-space 7-539
  - second-order sections to transfer functions 7-541
  - state-space to second-order sections 7-595
  - state-space to zero-pole-gain 7-600
  - transfer functions to lattice 7-616
  - transfer functions to second-order sections 7-617
  - transfer functions to state-space 7-620
  - zero-pole-gain to second-order sections 7-675
  - zero-pole-gain to state-space 7-679
- convert method 7-140
- convmtx function 7-112
- convolution
  - cross-correlation 3-2
  - filtering 1-15
  - matrix 1-43
  - matrix function (convmtx) 7-112

corrcoef function 7-114  
 correlation  
     bias 3-4  
     cross-correlation 7-661  
     linear prediction 4-17  
     matrices 7-115  
     *See also* autocorrelation, cross-correlation  
 correlation matrices 7-115  
 corrmatrix function 7-115  
 cosine windows 4-7  
 cov function 7-118  
 covariance  
     definition 3-9  
     methods 3-45  
     modified covariance spectrum object 7-570  
     spectrum object 7-563  
     *See also* autocovariance, cross-covariance,  
         modified covariance method  
 cplxpair function 7-119  
 cpsd function 7-120  
 cross power spectral density 7-120  
     definition 3-30  
 cross spectral density  
     definition 3-30  
     *See also* power spectral density, spectral  
         estimation  
 cross-correlation 7-661  
     discussion 3-2  
     two-dimensional 7-665  
 cross-covariance 7-668  
     comparison to cross-correlation 3-2  
     multiple channels 3-5  
 czt function  
     *See also* chirp z-transforms  
 7-125

## D

data  
     markers 5-19  
     multichannel matrix 1-5  
     multichannel signals 1-8  
     time vectors 1-7  
     vectors 1-5  
 DC component suppression 1-49  
 dct function 7-128  
 de la Valle-Poussin windows  
     *See* Parzen windows  
 decimate 7-130  
 decode 7-635  
 deconv function 7-133  
 deconvolution 4-41  
 default session  
     sptool GUI 6-42  
 delay 7-152  
 delays  
     group 1-31  
     noninteger 2-29  
     phase 1-31  
     signals 2-28  
 demod 7-134  
 demod function 7-134  
 demodulation 7-134  
     example 4-36  
 dfilt function 7-137  
     cascade 7-150  
     convert structures 7-147  
     copying 7-147  
     delay 7-152  
     direct-form antisymmetric FIR 7-175  
     direct-form FIR transposed 7-181  
     direct-form I 7-154  
     direct-form I sos 7-156  
     direct-form I transposed 7-159

- direct-form I transposed sos 7-161
- direct-form II 7-164
- direct-form II sos 7-167
- direct-form II transposed 7-170
- direct-form II transposed sos 7-172
- direct-form IIR 7-179
- direct-form symmetric FIR 7-183
- FFT FIR 7-187
- filter implementation 2-53
- lattice allpass 7-190
- lattice ARMA 7-194
- lattice autoregressive 7-192
- lattice moving-average maximum 7-196
- lattice moving-average minimum 7-198
- methods 7-138
- parallel 7-200
- scalar 7-202
- state space 7-205
- structures 7-137, 7-274
- dfilt.cascade function 7-150
- dfilt.delay function 7-152
- dfilt.df1 function 7-154
- dfilt.df1sos function 7-156
- dfilt.df1t function 7-159
- dfilt.df1tsos function 7-161
- dfilt.df2 function 7-164
- dfilt.df2sos function 7-167
- dfilt.df2t function 7-170
- dfilt.df2tsos function 7-172
- dfilt.dfasymfir function 7-175
- dfilt.dffir function 7-179
- dfilt.dffirt function 7-181
- dfilt.dfsymfir function 7-183
- dfilt.fffir function 7-187
- dfilt.latticeallpass function 7-190
- dfilt.latticear function 7-192
- dfilt.latticearma function 7-194
- dfilt.latticemamax function 7-196
- dfilt.latticemamin function 7-198
- dfilt.parallel function 7-200
- dfilt.scalar function 7-202
- dfilt.statespace function 7-205
- dft
  - See* discrete Fourier transforms
- dftmtx function 7-207
- difference equations 1-35
- differentiators
  - definition 2-29
  - least square linear-phase FIR 7-293
  - Parks-McClellan FIR 7-299
- digit reversal 7-208
- digital audio tape standards 4-25
- digital filters
  - anti-causal 1-21
  - Butterworth 7-57
  - Butterworth order estimation 7-62
  - cascade 1-40
  - Chebyshev Type I order estimation 7-80
  - Chebyshev Type II 7-98
  - Chebyshev Type II order estimation 7-86
  - coefficients 1-16
  - comparison to IIR 2-17
  - convolution 1-15
  - convolution matrices 1-43
  - design 2-2
  - elliptic 7-246
  - elliptic order estimation 7-253
  - equiripple FIR order estimation 7-304
  - FFT FIR overlap-add 7-263
  - FIR design 2-17
  - fixed-point implementation 1-40
  - frequency response 1-26
  - group delay description 1-31

- group delay function 7-349
- identification from frequency data 7-380
- IIR design 2-4
- implementation with filter 1-15
- impulse response 7-367
- impulse response definition 1-24
- initial conditions 1-18
- lattice/ladder structures 1-40
- models 1-35
- order 1-16
- phase delay definition 1-31
- poles 1-36
- second-order sections 1-40
- specifications 2-7
- state-space 1-37
- time-domain representation 1-17
- transfer functions representation 1-16
- transients 1-23
- transposed direct-form II structures 1-18
- types 1-16
- zero-phase 7-271
- zero-phase implementation 1-21
- zero-pole analysis 1-33
- zeros 1-36
- See also* FIR filters, IIR filters
- digital frequency A-1
- digitrevorder function 7-208
- diric function 7-210
- Dirichlet functions 7-210
  - definition 1-13
- discrete cosine transforms 7-128
  - definition 4-44
  - energy compaction property 4-45
  - example 4-45
  - inverse 7-361
  - reconstruct signals 4-45
- discrete Fourier transforms
  - algorithms 1-48
  - definition 1-47
  - eigenvector equivalent 3-47
  - goertzel 1-49
  - IIR filter implementation 1-23
  - inverse two-dimensional 1-49
  - magnitude 1-48
  - matrix 7-207
  - phase 1-48
  - power spectrum estimation 3-10
  - signal length dependencies 1-48
  - spectral analysis 3-6
  - time-dependent 4-32
  - two-dimensional 1-49
  - See also* fast Fourier transforms
- discrete prolate spheroidal sequences
  - See* dpss function
- discretization 7-364
  - bilinear transformations 2-49
  - filters 2-46
  - impulse invariance 2-47
- downsample function 7-211
- dpss function 7-213
  - example 3-30
- dpssclear function 7-216
- dpssdir function 7-217
- dpssload function 7-218
- dpsssave function 7-219
- dspdata object 7-220
  - mean-square spectrum 7-226
  - psd 7-231
  - pseudospectrum 7-236
- dspdata.msspectrumd function 7-226
- dspdata.psd function 7-231
- dspdata.pseudospectrum function 7-236

dsports object 7-242  
duty cycles 1-9

## E

echo detection 4-28  
edge effects 1-23  
eigenanalysis 3-47  
eigenvector method 7-437  
    definition 3-46  
    root MUSIC 7-516  
    spectral estimation 3-8  
    spectrum object 7-565  
    *See also* multiple signal classification method  
ellip function 7-246  
ellipap function 7-252  
ellipord function 7-253  
elliptic filters 7-246  
    definition 2-12  
    limitations 7-250  
    order estimation 7-253  
encoding 7-638  
eqtflength function 7-258  
equiripple 2-24  
    Chebyshev Type I filters 2-10  
    Chebyshev Type II filters 2-11  
    Chebyshev windows 4-14  
    elliptic filters 2-12  
    elliptic filters (analog) 7-252  
    elliptic filters (Cauer) 7-246  
    Parks-McClellan design 7-297  
error minimization 2-24  
    weighted frequency bands 2-27  
estimation 3-8  
    covariance method 7-24  
    cross spectral density 3-30  
    modified covariance method 7-25

Yule-Walker method 7-26  
*See also* parametric modeling

export  
    filter 5-36  
    window 7-655

## F

fast Fourier transforms 1-47  
    example 1-48  
    frequency response 1-26  
    Goertzel algorithm 7-345  
    implementation 1-23  
    output 1-49  
fcfwrite method 7-141  
FDATool  
    exporting to Simulink 5-40  
fdatool GUI 7-259  
    analysis buttons 5-17  
    computing coefficients 5-16  
    design methods 5-10  
    exporting filters 5-36  
    filter architecture 5-29  
    filter design specification 5-11  
    filter implementation 5-29  
    filter order specification 5-12  
    filters structure 5-29  
    frequency response specification 5-14  
    group delay 5-17  
    importing 5-32  
    impulse response 5-17  
    magnitude response 5-17  
    M-files 5-47  
    opening 5-8  
    phase delay 5-17  
    phase response 5-17  
    pole-zero plots 5-17

- response type 5-9
  - saving coefficients 5-36
  - second analysis 5-17
  - sessions 5-50
  - step response 5-17
- FFT
- See* fast Fourier transforms
- fftc coeffs method 7-141
- fftfilt function 7-263
- fftshift function 7-266
- Filter block 5-40
- filter design
- sptool Filter Designer GUI 6-46
  - See also* fdatool GUI
- Filter Designer GUI
- See* fdatool GUI
- filter function 7-267
- description 1-18
- filter method 7-141
- Filter Viewer
- introduction 7-592
  - open 6-10
  - printing 6-24
- Filter Visualization Tool
- See* fvtool GUI
- filternorm function 7-269
- filters
- analog 2-9
  - analog lowpass 7-31
  - analog lowpass prototype 7-56
  - anti-causal 1-21
  - anti-symmetric 2-27
  - bit reversal 7-39
  - Butterworth 7-57
  - Butterworth (generalized) 2-15
  - Butterworth order 7-62
  - C header file 5-44
  - Chebyshev Type I 7-93
  - Chebyshev Type I lowpass prototype 7-79
  - Chebyshev Type I order 7-80
  - Chebyshev Type II 7-98
  - Chebyshev Type II order 7-86
  - coefficients 1-16, 1-17
  - coefficients in sptool GUI 6-29, 6-49
  - convert coefficients to autocorrelation 7-476
  - convert from reflection coefficients 7-503
  - convert to reflection coefficients 7-478
  - convolution 1-15
  - design 2-5
  - digit reversal 7-208
  - discretization 2-46
  - elliptic 7-246
  - elliptic order 7-253
  - equiripple 2-24
  - export 5-36
  - filter and filtfilt functions comparison 1-22
  - filter function 1-17
  - filtstates object 7-274, 7-275
  - FIR 7-297
  - FIR design 2-24
  - FIR single band 2-22
  - frequency data 7-376
  - frequency domain 1-23
  - frequency transformations 2-43
  - fvtool GUI 7-318
  - implementation 2-52
  - importing to sptool GUI 6-31
  - initial conditions using dfilt 7-147
  - initial conditions using filter function 1-18
  - initial conditions using filtic function 7-272
  - inverse analog 7-376
  - inverse discrete-time 7-380
  - lattice/ladder 1-40

- linear phase 2-18
  - linear prediction 4-17
  - linear system models 1-35
  - linear time-invariant digital 1-3
  - median filtering 4-33
  - median function 7-413
  - minimax 2-24
  - minimum phase 7-481
  - norm 7-269
  - numerator and denominator length 7-258
  - objects 7-137
  - order 1-16
  - overlap-add using `dfilt.fftfir` 7-187
  - overlap-add using `fftfilt` 7-263
  - phase delay 7-449
  - phase distortion removal 1-20
  - phase modulation 4-31
  - phase response 7-452
  - pole-zero editor 5-25
  - sampling frequency 5-22
  - saving 5-48
  - Savitzky-Golay 7-530
  - Savitzky-Golay design 7-526
  - Schur realizations 7-523
  - second-order sections 1-40
  - second-order sections filtering 7-545
  - second-order sections IIR 7-545
  - specifications 2-7
  - `sptool` GUI Filter Designer 6-46
  - states 7-147
  - step response 7-602
  - structures 2-52
  - types 1-16
  - viewing 7-318
  - zero-phase 7-271
  - zero-phase implementation 1-21
  - zero-phase response 7-673
- See also* analog filters, digital filters, `fdatool` GUI, FIR filters, IIR filters
- `filtfilt` function 7-271
    - filter function comparison 1-22
  - `filtic` function 7-272
  - `filtstates` object 7-274, 7-275
  - FIR filters 2-17
    - arbitrary response 2-37
    - complex response 7-70
    - constrained least square 2-31
    - differentiators 2-29
    - equiripple 2-24
    - example 6-15
    - frequency domain 1-23
    - frequency response 7-281
    - Hilbert transformers 2-27
    - IIR filter comparison 2-17
    - implementation 1-18
    - interpolation 7-373
    - Kaiser windows 4-12
    - lattice/ladder 1-40
    - least square and equiripple comparison 2-25
    - least square linear phase 7-292
    - least square multiband 2-33
    - least square weighted 2-35
    - linear phase 2-18
    - linear phase Parks-McClellan 7-297
    - multiband 2-24
    - multiband example 2-23
    - nonlinear phase response 7-70
    - order estimation 7-304
    - overlap-add 7-263
    - Parks-McClellan method 7-297
    - reduced delay response 2-40
    - resample 1-20
    - `sptool` GUI Filter Designer 6-46
    - standard band 2-22



- types 7-302
- window-based 7-277
- windowing method 2-19
- fir1 function 7-277
  - example 2-21
- fir2 function 7-281
- fircls function 7-284
- fircls1 function 7-288
- firls function 7-292
  - differentiators 2-29
  - firpm comparison 2-25
  - weight vectors 2-27
- firpm function 7-297
  - differentiators 2-29
  - example 2-24
  - filter characteristics 7-302
  - firls comparison 2-25
  - Hilbert transformers 2-27
  - order estimation 7-304
  - weight vectors 2-27
- firpmord function 7-304
  - example 2-17
- firrcos function 7-307
- firtype method 7-141
- flattopwin flat top window function 7-309
- FM
  - See* frequency modulation
- fopen function 1-14
- Fourier transforms
  - See* discrete Fourier transforms, fast Fourier transforms
- fread function 1-14
- freqs function 7-311
- frequency 7-298
  - analog A-1
  - angular 2-2
  - center 2-45
  - cutoff 2-43
  - demodulation 7-135
  - digital A-1
  - estimation 3-46
  - modulation 7-414
  - normalization 2-2
  - Nyquist 2-2, A-1
  - prewarping 7-34
  - spectrogram 7-546
  - vectors 2-26
- frequency domain
  - duality with time-domain 1-23
  - filters 1-23
  - FIR filtering 1-20
  - lowpass to bandpass transformation 7-397
  - lowpass to bandstop transformation 7-400
  - lowpass to highpass transformation 7-402
  - transformation functions 2-43
- frequency domain based modeling
  - See* parametric modeling
- frequency modulation 7-415
- frequency response 1-26
  - Bessel filters 2-13
  - Butterworth filters 2-9
  - Chebyshev Type I filters 2-10
  - Chebyshev Type II filters 2-11
  - elliptic filters 2-12
  - error minimization 2-24
  - evaluating 1-26
  - example 1-27
  - inverse 7-376
  - Kaiser window 4-10
  - linear phase 2-18
  - magnitude 1-29
  - monotonic 2-9
  - multiband 2-14
  - phase 1-29

- plotting 1-27
- sampling frequency 1-26
- freqz function 7-314
  - sampling frequencies 1-26
- freqz method 7-141
- From Disk radio button 6-35
- FVTool
  - SOS view settings 7-325
- fvtool GUI 7-318

**G**

- gauspuls function 7-337
  - pulse trains 1-11
- gaussfir 7-339
- Gaussian monopulse 7-343
- Gauss-Newton method
  - analog domain 7-379
  - discrete domain 7-382
- gausswin Gaussian window function 7-341
- generalized Butterworth filters 2-15
- generalized cosine windows 4-7
- generalized filters 2-5
- generate method 7-532
- Gibbs effect 2-20
  - reduced by window 4-2
- gmonopuls function 7-343
- GMSK 7-339
- goertzel function 7-345
- graphical user interface (GUI) 1-4
  - See also* fdatool GUI, fvtool GUI, sptool GUI, wintool GUI, wvtool GUI
- group delay 1-31
  - comparison to phase delay 2-18
  - example 1-31
  - grpdelay function 7-349
  - passband 2-13

- grpdelay function 7-349
  - example 1-31
- grpdelay method 7-141

## H

- halfrange method 7-222
- hamming window function 7-353
  - comparison to boxcar 3-20
  - comparison to Hann 4-7
  - example 2-20
- hann window function 7-355
  - comparison to Hamming 4-7
- hanning
  - See* hann window function
- highpass filters
  - Butterworth 7-57
  - Butterworth order 7-63
  - Chebyshev Type I 7-93
  - Chebyshev Type I order 7-81
  - Chebyshev Type II 7-98
  - Chebyshev Type II order 7-87
  - elliptic 7-247
  - elliptic order 7-254
  - FIR 7-279
  - FIR example 2-22
  - lowpass transformation 7-402
- hilbert transform function 7-357
  - analytic signals 2-28
  - description 4-46
  - example 2-28
  - using firpls 7-293
  - using firpm 7-299
- homomorphic systems 4-28

- I**
- icceps function 7-360
    - example 4-31
  - idct function 7-361
    - example 4-44
  - ideal lowpass filters 2-19
    - See also* lowpass filters
  - ifft function
    - example 1-49
  - ifft2 function
    - example 1-49
  - IIR filters
    - analog prototype 2-6
    - Bessel 2-13
    - Butterworth 2-9
    - Chebyshev Type I 2-10
    - Chebyshev Type II 2-11
    - comparison 2-9
    - comparison to FIR 2-4
    - design 2-4
    - elliptic 2-12
    - Filter Designer GUI 6-46
    - frequency domain implementation 1-23
    - frequency response 2-14
    - generalized Butterworth 2-15
    - lattice/ladder 1-40
    - Levinson-Durbin recursion 7-395
    - maximally flat 2-15
    - multiband 2-14
    - order estimation 2-7
    - plotting responses 2-14
    - Prony's method 7-482
    - specifications 2-7
    - Steiglitz-McBride iteration 7-608
    - yulewalk function 7-671
    - Yule-Walker example 2-14
    - zero-phase implementation 1-21
      - See also* direct design
  - image processing 1-49
  - impinvar function 7-364
  - Import dialog box
    - sptool from disk 6-35
    - sptool from workspace 6-16
  - impulse invariance 7-364
    - example 2-47
  - impulse response 1-24
    - impz function 7-367
    - ideal 2-19
    - impulse invariance 2-47
  - impz function 7-367
  - impz method 7-141
  - impzlength method 7-141
  - indexing 1-16
  - inf-norm 7-269
  - info method
    - dfilt function 7-141
    - sigwin function 7-532
  - initial conditions
    - using dfilt states 7-147
    - using filtfilt function 1-22
    - using filtic function 7-272
  - instantaneous attributes 4-47
  - interpolation
    - bandlimited 7-536
    - FIR filters 7-373
    - interp function 7-371
  - interval notation A-1
  - intfilt function 7-373
  - inverse cepstrum, complex 4-31
  - inverse discrete cosine transforms 7-361
    - accuracy of signal reconstruction 4-46

- inverse discrete Fourier transforms 1-47
  - example 1-47
  - matrices 7-207
  - two-dimensional 1-49
- inverse filters
  - analog 7-376
  - discrete 7-380
- inverse Fourier transforms
  - See* sinc function
- inverse-sine parameters
  - transformations from reflection coefficients 7-383
  - transformations to reflection coefficients 7-501
- invfreqs function 7-376
  - example 4-22
- invfreqz function 7-380
  - example 4-22
- is2rc function 7-383
- isallpass method 7-142
- iscascade method 7-142
- isfir method 7-142
- islinphase method 7-142
- ismaxphase method 7-142
- isminphase method 7-142
- isparallel method 7-142
- isreal method 7-142
- isscalar method 7-142
- issos method 7-142
- isstable method 7-142

**K**

- kaiser window function 7-384
  - discussion 4-9
  - example 3-21
  - FIR filters 4-12
- kaiserord function 7-386

**L**

- ladder filters
  - See* lattice/ladder filters
- Lagrange interpolation filter 7-373
- Laplace transforms 1-44
- lar2rc function 7-391
- latc2tf function 7-392
  - example 1-43
- latcfilt function 7-393
  - example 1-20
- lattice/ladder filters 1-40
  - implementation 1-41
  - latcfilt function 1-43
  - Schur algorithm 7-523
  - transfer functions conversions 7-616
- least squares method FIR 7-292
- levinson function 7-395
  - example 4-17
  - parametric modeling 4-17
- line
  - drawing in FDATool 5-20
- line spectral frequencies
  - transformation from prediction polynomial 7-477
  - transformation to prediction polynomial 7-410
- line style 6-42
- linear models
  - See* models

linear phase filters 2-18  
     least squares FIR 7-292  
     optimal FIR 7-297  
 linear prediction  
     coefficients 7-406  
     modeling 4-17  
 linear predictive coding (lpc function) 4-17  
 linear system transformations  
     *See* conversions  
 log area ratio parameters  
     transformation from reflection coefficients  
         7-391  
     transformation to reflection coefficients 7-502  
 lowpass filters  
     Bessel 7-32  
     Butterworth 7-57  
     Butterworth order 7-63  
     Chebyshev Type I 7-93  
     Chebyshev Type I order 7-81  
     Chebyshev Type II 7-98  
     Chebyshev Type II order 7-87  
     cutoff frequency translation 7-404  
     decimation 7-130  
     elliptic 7-246  
     elliptic order 7-254  
     FIR 2-22  
     ideal 2-19  
     impulse invariance 2-47  
     impulse response 2-19  
     interpolation 7-371  
 lp2bp function 7-397  
     example 2-45  
 lp2bs function 7-400  
 lp2hp function 7-402  
 lp2lp function 7-404  
 lpc  
     *See* prony function, linear prediction

lpc function 7-406  
 lse 7-343  
 lsfpoly function 7-410

## M

magnitude  
     Fourier transforms 1-48  
     frequency response extraction 1-29  
     plots 6-55  
     transfer functions 3-31  
 match frequency prewarping 7-34  
 MAT-files  
     dpss.mat 3-30  
     format, converting to 1-14  
     importing 1-14  
     sptool GUI 6-35  
 matrices  
     convolution 1-43  
     convolution function 7-112  
     data 1-5  
     discrete Fourier transforms 7-207  
     inverse discrete Fourier transforms 7-207  
 matrix forms  
     *See* state-space forms  
 maxflat function 7-411  
     discussion 2-15  
 maximally flat filters  
     *See* maxflat function  
 maximum entropy estimate 3-38  
 mean-square spectrum 7-226  
 medfilt1 function 7-413  
     example 4-33  
 median filters  
     *See* medfilt1 function  
 MEX-files 1-14  
 M-files 1-4, 5-47

- minimax method
    - FIR filters 2-24
    - See also* Parks-McClellan algorithm
  - minimum phase 7-481
  - models 1-35
    - autoregressive Burg 7-23
    - autoregressive Burg PSD 7-426
    - autoregressive covariance 7-24
    - autoregressive covariance PSD 7-431
    - autoregressive modified covariance 7-25
    - autoregressive modified covariance PSD 7-456
    - autoregressive Yule-Walker 7-26
    - autoregressive Yule-Walker PSD 7-494
    - bilinear transformations 2-49
    - transformations 2-49
  - modified covariance method 3-45
  - modulate function 7-414
    - definition 4-34
    - example 4-36
    - See also* amplitude modulation
  - modulate function
    - time vector 4-35
  - modulation 4-34
  - moving-average (MA) filters 1-16
    - See also* FIR filters
  - mscohere function 7-417
  - msspectrum method 7-552
  - msspectrumopts method 7-553
  - MTM
    - See* multitaper method
  - multiband filters
    - FIR 2-23
    - IIR 2-14
  - multichannel data 1-8
  - multiple signal classification method (MUSIC)
    - correlation matrices 7-115
    - discussion 3-8
    - eigenvector method 7-437
    - example 3-46
    - pseudospectrum 7-468
  - multiplicity of zeros and poles 6-55
  - multirate filters 1-20
  - multitaper method (MTM) 3-27
  - multi-taper spectrum object 7-572
  - MUSIC algorithm
    - See* multiple signal classification method
  - MUSIC spectrum object 7-576
- ## N
- nonrecursive filters
    - See* FIR filters
  - normalization 3-4
    - cross-correlation 7-662
    - modified periodogram 3-19
    - periodogram bias 3-18
    - Welch's power spectral density 3-26
  - normalizefreq method 7-222
  - nsections method 7-142
  - nstages method 7-142
  - nstate method 7-143
  - nuttallwin Nuttall window function 7-422
  - Nyquist frequency A-1

**O**

## object

- changing properties 7-147
- copying 7-559
- dspdata 7-220
- dspspts 7-242
- filter 7-137
- filtstates 7-274, 7-275
- spectrum 7-550
- window 7-532

## onesided method 7-222

## options object 7-242

## order

- bit reversed 7-39
- Butterworth estimation 7-62
- Chebyshev Type I estimation 7-80
- digit reversed 7-208
- elliptic estimation 7-253
- estimation 2-7
- FIR optimal estimation 7-304

## order method 7-143

## oscillators 7-647

## overlap-add filter 7-187

## overlap-add method

- FIR filter implementation 1-23
- FIR filters 7-263

**P**

## Panner check box 6-42

## parallel method 7-143

## parametric modeling 4-15

- applications 4-15
- covariance method 7-24
- frequency domain based 4-22
- linear predictive coding 4-17
- modified covariance method 7-25

## Steiglitz-McBride method 4-20

## summary 2-5

## techniques 4-15

## time-domain based 4-17

## Yule-Walker method 7-26

## parentheses A-1

## Parks-McClellan algorithm

## FIR filters 2-24

*See also* minimax method

## Parks-McClellan method 7-297

## partial fraction expansion 1-44

## residue 1-38

## z-transform 7-510

## parzenwin Parzen window function 7-424

## passband

## Chebyshev Type I 2-10

## equiripple 2-12

## group delay 2-13

## pburg function 7-426

## example 3-41

## PCM 7-638

## pcov function 7-431

## example 3-45

## peig function 7-437

## period in sequence 7-524

## periodic sinc functions 7-210

*See also* Dirichlet functions

## periodogram function 7-444

## discussion 3-10

## spectrum object 7-581

## phase

## delay 1-31

## demodulation 7-135

## distortion 1-21

## Fourier transforms 1-48

## frequency response 1-29

## group delay 7-349

- linear delay 2-18
- modulation 7-415
- transfer functions 3-31
- unwrapping 1-29
- phase response 7-452
- phasedelay function 7-449
- phasez function 7-452
- phasez method 7-143
- plot method 7-223
- plots
  - analog filters 2-14
  - coherence function 3-32
  - complex cepstrum 4-29
  - DFT 1-48
  - frequency response 1-27
  - group delay 1-31
  - magnitude 6-55
  - magnitude and phase 1-29
  - phase 1-29
  - phase delays 1-31
  - strip plots 7-612
  - transfer functions 3-31
  - zero-pole 1-33
  - zplane function 7-681
- plug-ins 6-42
- pmcov function 7-456
  - example 3-45
- p-model
  - See* parametric modeling
- pmtm function 7-462
- pmusic function 7-468
- pole-zero editor 5-25
- pole-zero filters
  - See* IIR filters
- poly function
  - example 1-36
- poly2ac function 7-476
- poly2lsf function 7-477
- poly2rc function 7-478
- polynomials
  - division 4-41
  - roots 1-36
  - scaling 7-481
  - stability check 7-478
  - stabilization 7-480
- polyphase filtering techniques 1-20
- polyscale function 7-480
- polystab function 7-481
- power spectral density 3-6
  - Burg estimation 7-426
  - Burg estimation example 3-40
  - covariance estimation 7-431
  - covariance estimation example 3-45
  - dspdata object 7-231
  - eigenvector estimation 7-516
  - modified covariance estimation 7-456
  - multitaper estimation 7-462
  - multitaper estimation example 3-27
  - MUSIC estimation 7-468
  - MUSIC estimation example 3-46
  - periodogram bias 3-18
  - periodogram normalization 3-18
  - plots 6-12
  - sptool GUI 6-34
  - units 3-7
  - Welch's bias 3-26
  - Welch's estimation 7-488
  - Welch's estimation bias 3-25
  - Welch's estimation example 3-22
  - Welch's normalization 3-26
  - Yule-Walker estimation 7-494
  - Yule-Walker estimation example 3-38
- powerest method 7-557
- prediction filters 4-17



prediction polynomials  
  transformations from line spectral frequencies 7-410  
  transformations to line spectral frequencies 7-477

Preferences menu item 6-42

prewarping 7-34

Print dialog box 6-26

print to figure 5-23

prolate-spheroidal windows 4-9

prony function 7-482  
  example 4-18

Prony's method  
  *See* prony function

psd method 7-553

psdopts method 7-555

pseudospectrum object 7-236  
  eigenvector method 7-437  
  MUSIC algorithm 7-474

pseudospectrumopts object 7-557

pulse position demodulation 7-135

pulse position modulation 4-35

pulse time modulation 7-415

pulse train generator 7-484

pulse trains  
  example 1-11  
  Prony's method 7-484  
  pulstran function 1-11

pulse width demodulation 7-135

pulse width modulation 7-415

pulse-shaping filter 7-339

pwelch function 7-488

pyulear function 7-494  
  Burg comparison 3-41  
  example 3-38

**Q**

quadrature amplitude demodulation 7-135

quadrature amplitude modulation 7-415

quantization  
  decoding 7-635  
  encoding 7-638  
  reduction with filter norms 7-269

quantized filters  
  cell array coefficients 7-538  
  matrix coefficients 7-69

**R**

radar applications 4-32

raised cosine filters 7-307

range notation A-1

rc2ac function 7-500

rc2is function 7-501

rc2lar function 7-502

rc2poly function 7-503

rceps function 7-504

rceps function  
  example 4-30

realizemdl method 7-144

rebuffering 7-47

rectangular windows 4-3  
  rectwin function 7-506

rectpuls function 7-505

rectwin function 7-506  
  example 4-3

recursive filters  
  *See* IIR filters

references  
  general DSP 1-50  
  special topics 4-48  
  statistical signal processing 3-49

- reflection coefficients 1-41
  - autocorrelation sequence conversion 7-500
  - conversion from filter coefficients 7-478
  - conversion to prediction polynomial 7-503
  - definition 1-40
  - Schur algorithm 7-523
  - transformation from inverse sine parameters 7-501
  - transformation from log area ratio parameters 7-502
  - transformation to inverse sine parameters
    - transformation to 7-383
  - transformation to log area ratio parameters 7-391
- rejection area 5-20
- Remez exchange algorithm 7-297
- removestage method 7-145
- resample function 7-507
  - example 4-25
- resampling
  - See* decimation, interpolation
- residue forms
  - See* partial fraction expansion
- residuez function 7-510
- rlevinson function 7-513
- rooteig function 7-516
- rootmusic function 7-519
  - eigenvector method 7-516
- roots
  - polynomials 1-36
- rulers
  - sptool GUI 6-42
- S**
- sampling frequency 5-22
  - decrease 7-211
  - FIR filters 1-20
  - freqz function 1-28
  - increase 7-645
  - integer factor decrease 7-130
  - integer factor increase 7-371
  - irregularly spaced data 4-27
  - Nyquist interval 7-246
  - range 1-28
  - resample function 7-507
  - resampling discussion 4-25
  - spacing 1-28
  - using upfirdn function 1-20
- saved filters 5-48
- saving data
  - Spectrum Viewer 6-30
- Savitzky-Golay filters
  - design 7-526
  - filtering 7-530
- sawtooth function 7-522
  - example 1-9
- sawtooth wave 1-9
- scaling 7-480
- Schur algorithm 7-523
- schurrc function 7-523
- second-order section forms
  - zero-pole-gain conversion to 7-543
- second-order sections 1-40
  - cell array coefficients 7-538
  - conversion from transfer function 7-617
  - conversion to in fdatool 5-30
  - conversion to transfer functions 7-541
  - filter 7-545
  - filters 7-545
  - matrices 1-40

- matrix coefficients 7-69
  - sptool GUI 6-33
  - state-space conversion from 7-595
  - state-space conversion to 7-539
  - view 7-325
  - zero-pole-gain conversion from 7-675
- seqperiod function 7-524
- setstage method 7-145
- sgolay function 7-526
- sgolayfilt function 7-530
- Signal Browser 6-7
- axis labels 6-42
  - markers preferences 6-42
  - overview 6-7
  - Panner preferences 6-42
  - printing 6-24
  - signals, measuring 6-40
  - zooming, preferences 6-42
- signals 2-28
- adding noise 1-7
  - analytic 4-46
  - aperiodic 1-10
  - applications 4-46
  - array 6-7
  - buffering 7-47
  - carrier 4-34
  - chirp 1-10
  - continuous (analog) 1-3
  - DCT coefficients reconstruction 4-45
  - differentiators 2-29
  - diric function 1-13
  - discrete (digital) 1-3
  - generating 1-8
  - measurements 6-40
  - minimum phase reconstruction example 7-504
  - modulation 7-414
  - multichannel 3-5
  - periodic 1-9
  - plotting 1-7
  - properties 4-46
  - pulstran function 1-11
  - rebuffering 7-47
  - representing 1-5
  - sawtooth 1-9
  - sawtooth function 7-522
  - sinc 1-12
  - sinusoidal 1-7
  - square function 7-594
  - square wave 1-9
  - triangle 7-522
- sigwin function 7-532
- Simulink
- exporting from FDATool 5-40
- sinc function 7-536
- Dirichlet 7-210
  - example 1-12
- Slepian sequences
- See* discrete prolate spheroidal sequences
- sonar applications 4-32
- sos method 7-145
- SOS view settings 7-325
- sos2cell function 7-538
- sos2ss function 7-539
- sos2tf function 7-541
- sos2zp function 7-543
- sosfilt function 7-545
- spectral analysis 3-6
- cross spectral density 3-30
  - power spectral density 3-6
  - Spectrum Viewer 6-12
  - See also* spectral estimation

- spectral density 3-6
  - measurements 6-40
  - plots 6-12
  - Spectrum Viewer 6-12
  - units 3-7
  - See also* cross spectral density, power spectral density
- spectral estimation 3-10
  - AR covariance method 7-24
  - AR modified covariance method 7-25
  - AR Yule-Walker method 7-26
  - Burg method 7-426
  - Burg method example 3-40
  - covariance method 7-431
  - eigenvector method 7-437
  - modified covariance method 7-456
  - multitaper method 7-462
  - MUSIC method 7-469
  - periodograms 7-448
  - root eigenvector 7-516
  - root MUSIC 7-519
  - Welch's method bias 3-25
  - Welch's method discussion 3-22
  - Welch's method example 3-8
  - Yule-Walker AR method 7-494
  - Yule-Walker AR method example 3-38
- spectrogram 7-546
  - definition 4-32
  - VCO example 7-647
- spectrogram function 7-546
  - example 4-32
- spectrum
  - mask 5-20
- spectrum estimation methods 7-220
  - mean-square 7-226
  - psd 7-231
  - pseudospectrum 7-236
- spectrum function 7-550
  - burg 7-561
  - cov 7-563
  - eigenvector 7-565
  - estimation methods 7-550
  - mcov 7-570
  - methods 7-551
  - mtm 7-572
  - music 7-576
  - periodogram 7-581
  - welch 7-584
  - yulear 7-588
- Spectrum Viewer 6-12
  - activating 6-12
  - axis parameters 6-42
  - markers, preferences 6-42
  - measurements 6-40
  - opening 6-12
  - overview 6-12
  - printing 6-26
  - rulers 6-40
  - spectra structures 6-30
  - spectral density plots 6-12
  - windows 6-13
  - zooming 6-42
- spectrum.mtm function
  - example 3-27
- speech processing
  - parametric modeling 4-15
  - resampling 4-26
- spline function 4-27
- sptool GUI 7-590
  - colors, customizing 6-42
  - context-sensitive help 6-6
  - customizing 6-42
  - data entering 1-14
  - data objects 6-38

- data structures 6-3
- editing 6-39
- example 6-15
- exporting data 6-27
- filter coefficients 6-29, 6-49
- filter design 6-18
- filter importing 6-31
- filter parameters 6-29
- filter saving 6-28
- filtering 6-21
- filters 6-31
- help 6-6
- Import dialog 6-16
- importing filters and spectra 6-31
- importing signals 6-16
- items, selecting 6-38
- line style 6-42
- MAT-files 6-35
- MATLAB workspace 6-3
- multiselection of items 6-38
- operation 6-3
- preferences 6-42
- printing 6-26
- rulers 6-40
- sample frequency 6-49
- saving 6-27
- second-order section forms 6-33
- signal analysis 6-22
- signal measurement 6-40
- signal playing 6-23
- sound 6-23
- spectra analysis 6-24
- spectra import 6-34
- spectral densities import 6-31
- spectral densities plot 6-34
- Spectrum Viewer 6-24
- state-space forms 6-32
  - transfer functions 6-32
  - transfer functions export 6-29, 6-49
  - tutorial 6-2
  - workspace 6-3
  - zero-pole-gain forms 6-32
- square function 7-594
  - example 1-9
- square wave
  - See* square function
- ss method 7-145
- ss2sos function 7-595
- ss2tf function 7-599
- ss2zp function 7-600
- stability check
  - polynomials 7-478
- stabilization 7-481
- standards, digital audio tape 4-25
- startup transients 1-22
- state-space forms 1-37
  - continuous time 1-44
  - scalar 1-37
  - second-order section conversion from 7-539
  - second-order section conversion to 7-595
  - sptool GUI 6-32
  - transfer functions conversions to 7-620
  - zero-pole-gain conversion from 7-679
  - zero-pole-gain conversion to 7-600
- statistical operations 3-2
  - See also* autocorrelation sequences,  
cross-correlation sequences,  
cross-covariance
- Steiglitz-McBride method 7-608
  - example 4-20
- step response 7-602
- stepz method 7-146
- stepz function 7-602

- stmcb function 7-608
    - example 4-20
  - stopband
    - Chebyshev Type II 2-11
    - elliptic 2-12
  - strips function plots 7-612
  - structures
    - conversion 5-29
    - conversion round off 1-45
    - lattice/ladder 1-40
    - transposed direct-form II 1-18
  - swept-frequency cosine generator
    - See* chirp
  - system identification 4-18
- T**
- tapers (PSD estimates) 3-27
  - taps 2-18
  - Taylor window 7-614
  - taylorwin function 7-614
  - tf method 7-146
  - tf2latc function 7-616
    - example 1-41
  - tf2sos function 7-617
  - tf2ss function 7-620
  - tf2zp function 7-622
  - tfestimate function 7-626
    - example 3-30
  - time series attributes 4-47
  - time-domain based modeling
    - See* parametric modeling
  - transfer functions 1-16
    - coefficients 6-29, 6-49
    - discrete time models 1-35
    - factoring 1-36
    - lattice conversion to 7-616
    - partial fractions 1-38
    - second-order sections conversion from 7-541
    - second-order sections conversion to 7-617
    - sptool GUI 6-32
    - state-space conversion to 7-620
    - Welch's estimation 3-30
    - zero-pole-gain forms 1-36
  - transformations
    - bilinear 2-49
    - bilinear function 7-34
    - frequency 2-43
    - lowpass analog to bandpass 7-397
    - lowpass analog to bandstop 7-400
    - lowpass analog to highpass 7-402
    - lowpass cutoff change 7-404
    - models 1-45
  - transforms 4-42
    - chirp  $z$ -transforms (CZT) 7-125
    - chirp  $z$ -transforms (CZT) discussion 4-42
    - discrete cosine 7-128
    - discrete Fourier 1-47
    - hilbert 7-357
    - Hilbert discussion 4-46
    - inverse discrete cosine 7-361
    - inverse discrete cosine discussion 4-44
  - transients 1-23
  - transition band 2-25
  - transposed direct-form II
    - initial conditions 7-272
  - transposed direct-form II structure 1-18
  - triang triangle window function 7-630
    - Bartlett comparison 4-4
  - tripuls function 7-632
  - Tukey window function
    - See* tukeywin
  - tukeywin 7-633

two-dimensional operations 1-49  
twosided method 7-223

## U

udecode function 7-635  
uencode function 7-638  
uniform encoding 7-638  
unit circle 7-481  
unit impulse function 1-8  
unit ramp function 1-8  
unit sample multichannel 1-8  
unit step function 1-8  
units of power spectral density (PSD) 3-7  
unwrap function  
    example 1-29  
upfirdn function 7-642  
    example 1-20  
    resampling 4-27  
upsample function 7-645

## V

variables  
    load from disk 6-35  
variance 3-4  
vco  
    example 4-39  
vco function 7-647  
vectors  
    data representation 1-5  
    frequency 2-26  
    indexing 1-16  
    waveform generation 1-7  
    weighting 7-293  
voltage controlled oscillators 7-647  
    example 4-39

## W

waveforms  
    *See* signals  
Welch spectrum object 7-584  
Welch's method 3-22  
    AR Yule-Walker comparison 3-38  
    bias and normalization 3-26  
    Burg comparison 3-43  
    MTM comparison 3-30  
    nonparametric system identification 3-30  
    power spectral density estimation 3-30  
white noise 1-7  
wholerange method 7-223  
window function 7-649  
windows  
    Bartlett 7-29  
    Bartlett comparison 4-4  
    Bartlett-Hanning 7-27  
    Blackman 7-41  
    Blackman comparison 4-7  
    Blackman-Harris 7-43  
    Blackman-Harris vs. Nuttall 7-422  
    Bohman 7-45  
    boxcar 2-19  
    Chebyshev 4-14, 7-91  
    cosine 4-7  
    de la Valle-Poussin 7-424  
    designing 7-652  
    filters 2-19  
    FIR filters 2-19, 7-277  
        multiband filters 2-23  
        single band design 2-22  
    fir1 function 2-22  
    flat top weighted 7-309  
    Gaussian 7-341  
    Hamming 7-353  
    Hamming discussion 4-7

Hamming rectangular example 3-20  
Hamming ringing example 2-20  
Hann 7-355  
Hann example 4-7  
Kaiser 7-384  
Kaiser discussion 4-9  
Kaiser example 3-21  
Nuttall 7-422  
object 7-532  
Parzen 7-424  
prolate-spheroidal 4-9  
rectangular 7-506  
rectangular example 2-19  
shapes 4-3  
spectral leakage 3-13  
Taylor 7-614  
triangular 7-630  
Tukey 7-633  
viewing 7-657  
wintool GUI 7-652  
wvtool GUI 7-657  
wintool GUI 7-652  
winwrite method 7-532  
Workspace Contents list 6-16  
wvtool GUI 7-657

## X

xcorr function 7-661  
xcorr2 function 7-665  
xcov function 7-668

## Y

yulewalk function 7-671  
  example 2-15  
Yule-Walker AR method  
  description 3-38  
  example 3-41  
  Welch's method comparison 3-38  
Yule-Walker spectrum object 7-588

## Z

zero frequency component, centering 1-49  
zero-order hold  
  *See* averaging filters  
zero-phase  
  filtering 7-271  
  response 7-673  
zerophase function 7-673  
zerophase method 7-146  
zero-pole  
  analysis 7-681  
  multiplicity 6-55  
  plots 1-33  
  transfer functions 1-36  
zero-pole-gain 1-36  
zero-pole-gain forms 1-44  
  convert from second-order sections 7-543  
  convert from state-space 7-600  
  convert to second-order sections 7-675  
  convert to state-space 7-679  
  sptool GUI 6-32  
zoom  
  sptool GUI 6-42  
zp2sos function 7-675  
zp2ss function 7-679  
zp2tf function 7-680  
zpk method 7-146



zplane function 7-681

zplane method 7-146

z-transforms

  chirp z 4-42

  czt function 7-125

  definition 1-35

  discrete Fourier transforms 1-47

  equation 1-16

