Signal Processing Toolbox

For Use with MATLAB®

Computation

Visualization

Programming

The MATH WORKS

User's Guide

Version 5

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information
508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail

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

Signal Processing Toolbox User's Guide

© COPYRIGHT 1988 - 2001 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 or for the federal government of the United States. By accepting delivery of the Program, the government hereby agrees that this software qualifies as "commercial" computer software within the meaning of FAR Part 12.212, DFARS Part 227.7202-1, DFARS Part 227.7202-3, DFARS Part 252.227-7013, and DFARS Part 252.227-7014. The terms and conditions of The MathWorks, Inc. Software License Agreement shall pertain to the government's use and disclosure of the Program and Documentation, and shall supersede any conflicting contractual terms or conditions. If this license fails to meet the government's minimum needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to MathWorks.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and Target Language Compiler is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History:			
	1988	Original printing	
	January 1997	First printing	New for MATLAB 5.1
	January 1998	Second printing	Revised for MATLAB 5.2
	January 1999	(Online only)	Revised for Version 4.2 (Release 11)
	August 1999	(Online only)	Revised for Version 4.3 (Release 11.1)
	September 2000	Third printing	Revised for Version 5.0 (Release 12)
	June 2001	(Online only)	Revised for Version 5.1 (Release 12.1)

Contents

3 Statistical Signal Processing

Preface

Overview

This chapter provides an introduction to the Signal Processing Toolbox and the documentation. It contains the following sections:

- "What Is the Signal Processing Toolbox?"
- "R12 Related Products List"
- "How to Use This Manual"
- "Installing the Signal Processing Toolbox"
- "Technical Conventions"
- "Typographical Conventions"

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:

- Signal processing command line functions
- A suite of graphical user interfaces for:
  - Interactive filter design
  - Signal plotting and analysis
  - Spectral analysis
  - Filtering signals
  - Analyzing filter designs

R12 Related Products List

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with the Signal Processing Toolbox.

For more information about any of these products, see either:

- The online documentation for that product if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`; see the "products" section

Note The toolboxes listed below all include functions that extend MATLAB's capabilities. The blocksets all include blocks that extend Simulink's capabilities.

Product	Description
Communications Blockset	Simulink block libraries for modeling the physical layer of communications systems
Communications Toolbox	MATLAB functions for modeling the physical layer of communications systems
Data Acquisition Toolbox	MATLAB functions for direct access to live, measured data from MATLAB
Database Toolbox	Tool for connecting to, and interacting with, most ODBC/JDBC databases from within MATLAB

Product	Description
Developer’s Kit for Texas Instruments™ DSP	Developer’s kit that unites MATLAB, Simulink, and Real-Time Workshop code generation with the Texas Instruments Code Composer Studio™ to provide DSP software/ systems architects with suite of tools for DSP development, from simulating signal processing algorithms to optimizing and running code on Texas Instruments DSPs
DSP Blockset	Simulink block libraries for the design, simulation, and prototyping of digital signal processing systems
Fuzzy Logic Toolbox	Tool to help master fuzzy logic techniques and their application to practical control problems
Image Processing Toolbox	Complete suite of digital image processing and analysis tools for MATLAB
Neural Network Toolbox	Comprehensive environment for neural network research, design, and simulation within MATLAB
Motorola DSP Developer’s Kit	Developer's kit for co-simulating and verifying Motorola 56300 and 56600 fixed-point DSP code. Combines the algorithm development, simulation, and verification capabilities of the MathWorks system-level design tools with the Motorola Suite 56® assembly language development and debugging tools
Optimization Toolbox	Tool for general and large-scale optimization of nonlinear problems, as well as for linear programming, quadratic programming, nonlinear least squares, and solving nonlinear equations

Product	Description
Simulink	Interactive, graphical environment for modeling, simulating, and prototyping dynamic systems
Statistics Toolbox	Tool for analyzing historical data, modeling systems, developing statistical algorithms, and learning and teaching statistics
System Identification Toolbox	Tool for building accurate, simplified models of complex systems from noisy time-series data
Wavelet Toolbox	Tool for signal and image analysis, compression, and de-noising

How to Use This Manual

This section explains how to use the documentation to get help about the Signal Processing Toolbox. Read the topic that best fits your skill level:

- "If You Are a New User"
- "If You Are an Experienced Toolbox User"
- "All Toolbox Users"

If You Are a New User

Begin with Chapter 1, "Signal Processing Basics." This chapter introduces the MATLAB signal processing environment through the toolbox functions. It describes the basic functions of the Signal Processing Toolbox, reviewing its use in basic waveform generation, filter implementation and analysis, impulse and frequency response, zero-pole analysis, linear system models, and the discrete Fourier transform.

When you feel comfortable with the basic functions, move on to Chapter 2 and Chapter 3 for a more in-depth introduction to using the Signal Processing Toolbox:

- Chapter 2, "Filter Design," for a detailed explanation of using the Signal Processing Toolbox in infinite impulse response (IIR) and finite impulse response (FIR) filter design and implementation, including special topics in IIR filter design.
- Chapter 3, "Statistical Signal Processing," for how to use the correlation, covariance, and spectral analysis tools to estimate important functions of discrete random signals.

Once you understand the general principles and applications of the toolbox, learn how to use the interactive tools:

- Chapter 5, "Filter Design and Analysis Tool," and Chapter 6, "SPTool: A Signal Processing GUI Suite," for an overview of the interactive GUI environments and examples of how to use them for signal exploration, filter design and implementation, and spectral analysis.

Finally, see the following chapter:

- Chapter 4, “Special Topics,” for specialized functions, including filter windows, parametric modeling, resampling, cepstrum analysis, time-dependent Fourier transforms and spectrograms, median filtering, communications applications, deconvolution, and specialized transforms.

If You Are an Experienced Toolbox User

See Chapter 5, “Filter Design and Analysis Tool,” and Chapter 6, “SPTool: A Signal Processing GUI Suite,” for an overview of the interactive GUI environments and examples of how to use them for signal viewing, filter design and implementation, and spectral analysis.

All Toolbox Users

Use Chapter 7, “Function Reference,” for locating information on specific functions. Reference descriptions include a synopsis of the function’s syntax, as well as a complete explanation of options and operations. Many reference descriptions also include helpful examples, a description of the function’s algorithm, and references to additional reading material.

Use this manual in conjunction with the software to learn about the powerful features that MATLAB provides. Each chapter provides numerous examples that apply the toolbox to representative signal processing tasks.

Some examples use MATLAB’s random number generation function `randn`. In these cases, to duplicate the results in the example, type

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

before running the example.

Installing the Signal Processing Toolbox

To determine if the Signal Processing Toolbox is installed on your system, type this command at the MATLAB prompt.

```
ver
```

When you enter this command, MATLAB displays information about the version of MATLAB you are running, including a list of all toolboxes installed on your system and their version numbers.

For information about installing the toolbox, see the *MATLAB Installation Guide* for your platform.

Technical Conventions

This manual and the Signal Processing Toolbox functions use the following technical notations.

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.
Ω or `w`	Analog frequency in radians per second.
ω 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.

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention Used	Example
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names/syntax	`Monospace` font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Keys	**Boldface** with an initial capital letter	Press the **Return** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'`**`whole`**`')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$

MATLAB output	`Monospace font`	MATLAB responds with
		`A =`
		`5`

Menu titles, menu items, dialog boxes, and controls	**Boldface** with an initial capital letter	Choose the **File** menu.
New terms	*Italics*	An *array* is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'`*`method`*`')`

Signal Processing Basics

Overview

This chapter describes how to begin using MATLAB and the Signal Processing Toolbox for your signal processing applications. It assumes a basic knowledge and understanding of signals and systems, including such topics as filter and linear system theory and basic Fourier analysis. The chapter covers the following topics:

- "Signal Processing Toolbox Central Features"
- "Representing Signals"
- "Waveform Generation: Time Vectors and Sinusoids"
- "Working with Data"
- "Filter Implementation and Analysis"
- "The filter Function"
- "Other Functions for Filtering"
- "Impulse Response"
- "Frequency Response"
- "Zero-Pole Analysis"
- "Linear System Models"
- "Discrete Fourier Transform"
- "Selected Bibliography"

Many examples throughout the chapter demonstrate how to apply toolbox functions. If you are not already familiar with MATLAB's signal processing capabilities, use this chapter in conjunction with the software to try examples and learn about the powerful features available to you.

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. “Filter Design” on page 2-1 and “Statistical Signal Processing” on page 3-1, 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: SPTool and FDATool

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.

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/π 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)';
```

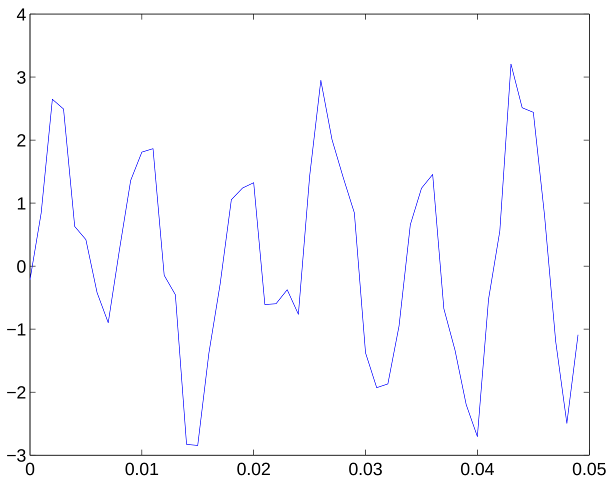
where MATLAB's colon operator creates a 1001-element row vector that represents time running from zero to one second in steps of one millisecond. The transpose operator (') changes the row vector into a column; the semicolon (;) tells MATLAB to compute but not display the result.

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 MATLAB's colon operator and the `ones` function.

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

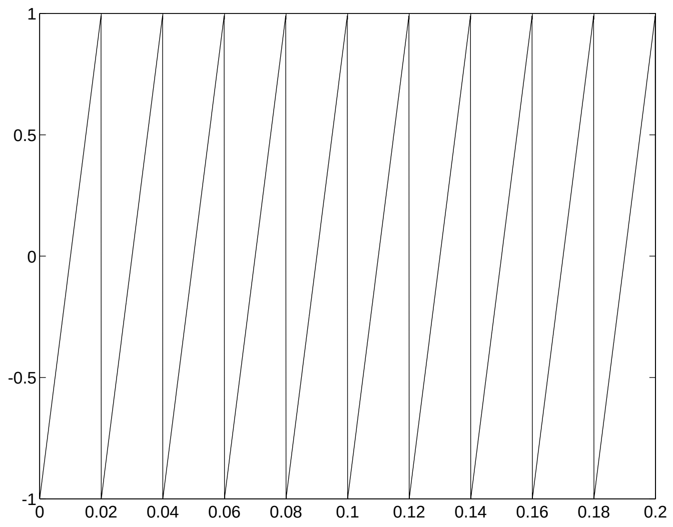
Common Periodic Waveforms

The toolbox provides functions for generating widely used periodic waveforms:

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

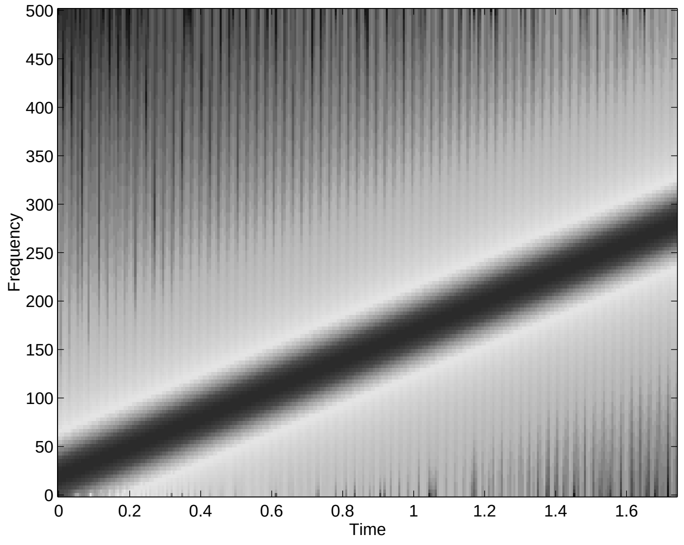
- `gauspuls` generates a Gaussian-modulated sinusoidal pulse with a specified time, center frequency, and fractional bandwidth. Optional parameters return in-phase and quadrature pulses, the RF signal envelope, and the cutoff time for the trailing pulse envelope.
- `chirp` generates a linear swept-frequency 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

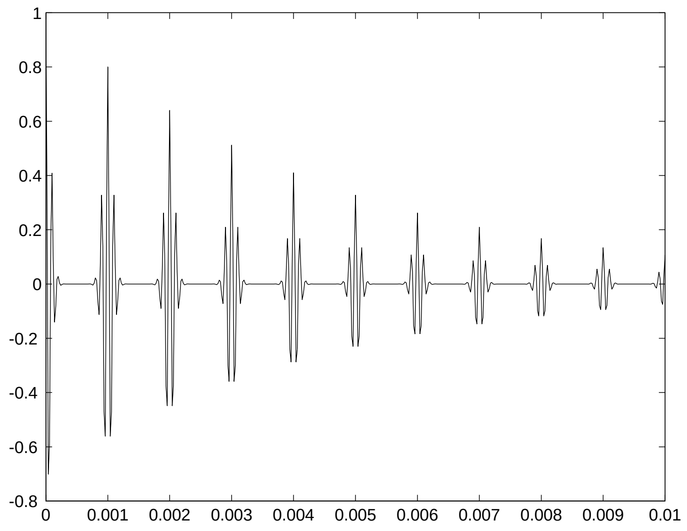
```
specgram(y,256,1000,256,250)
```



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 2π and height 1.

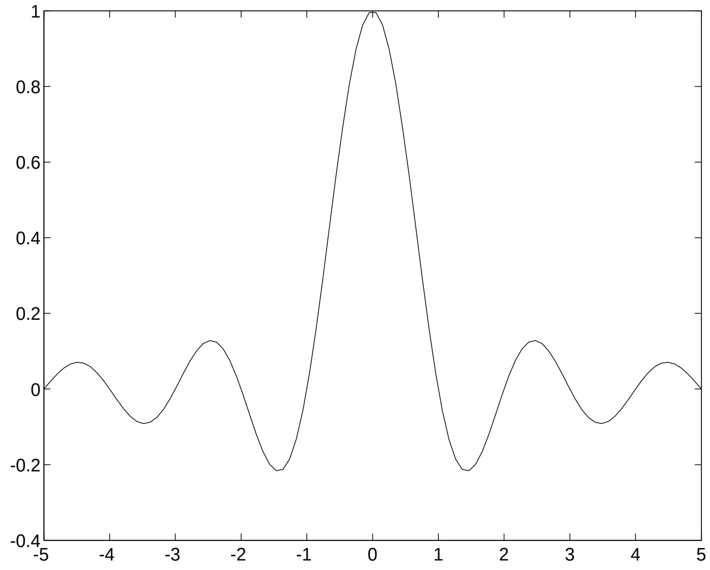
The sinc function has a value of 1 where `x` is zero, and a value of

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

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

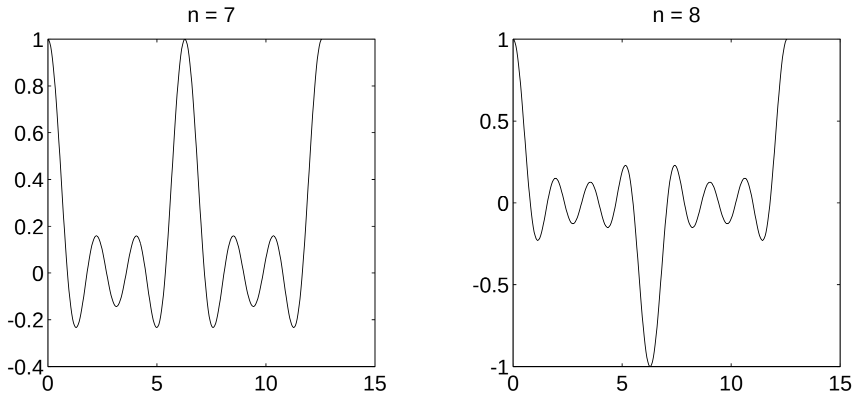
The toolbox function `diric` computes the Dirichlet function, sometimes called the *periodic sinc* or *aliased sinc* function, for an input vector or matrix `x`. The Dirichlet function is

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

where n is a user-specified positive integer. For n odd, the Dirichlet function has a period of 2π; for n even, its period is 4π. 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 4π for `n = 7` and `n = 8`, use

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



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 MATLAB's `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 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.

Filter Implementation and Analysis

This section describes how to filter discrete signals using MATLAB's `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. MATLAB's `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 MATLAB's standard indexing scheme for vectors.

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.

$$y(k) + a_2 y(k-1) + \cdots + a_{m+1} y(k-m) = b_1 x(k) + b_2 x(k-1) + \cdots + b_{n+1} x(k-m)$$

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

$$y(k) = b_1 x(k) + b_2 x(k-1) + \cdots + b_{n+1} x(k-n) - a_2 y(k-1) - \cdots - a_{m+1} y(k-n)$$

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

$$\begin{aligned} y(1) &= b_1 x(1) \\ y(2) &= b_1 x(2) + b_2 x(1) - a_2 y(1) \\ y(3) &= b_1 x(3) + b_2 x(2) + b_3 x(1) - a_2 y(2) - a_3 y(1) \\ \vdots &= \vdots \end{aligned}$$

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
```

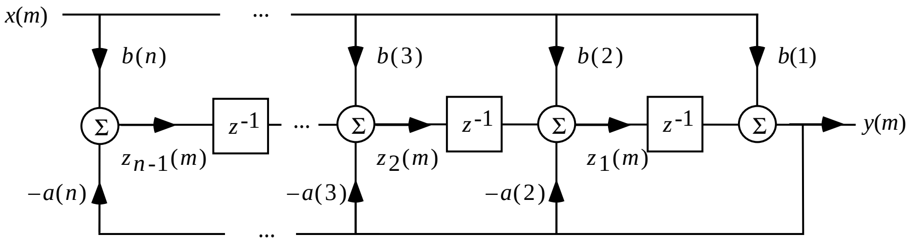
where the vectors `b` and `a` represent the coefficients of a filter in transfer function form. To apply this filter to your data, use

```
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 shown below, 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

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

In its most basic form, `filter` initializes the delay outputs $z_i(1)$, $i = 1, ..., n$-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] = filter(b,a,x,zi)
```

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

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

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

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

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

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

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

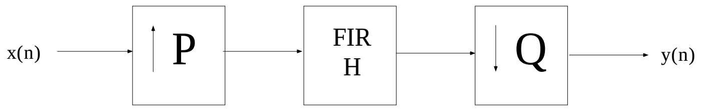
Other Functions for Filtering

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

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

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

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

This cascade of operations is implemented in an efficient manner using polyphase filtering techniques, and it is a central concept of multirate filtering (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`.

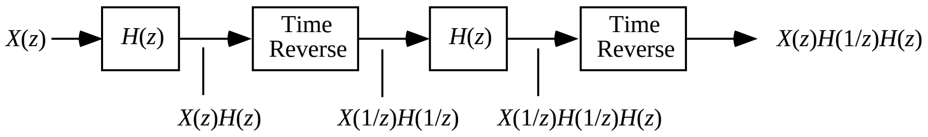
Filter banks may be implemented using `upfirdn` by allowing the filter `h` to be a matrix, with one FIR filter per column. A signal vector is passed independently through each FIR filter, resulting in a matrix of output signals.

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



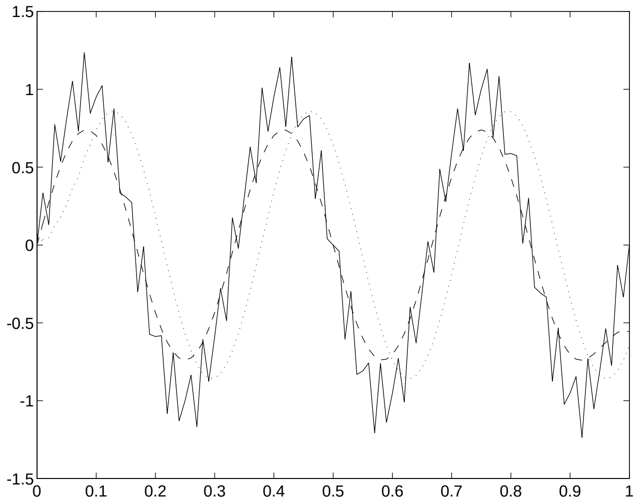
When $|z| = 1$, that is $z = e^{j\omega}$, the output reduces to $X(e^{j\omega})|H(e^{j\omega})|^2$. Given all the samples of the sequence $x(n)$, a doubly filtered version of x that has zero-phase distortion is possible.

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)];
```

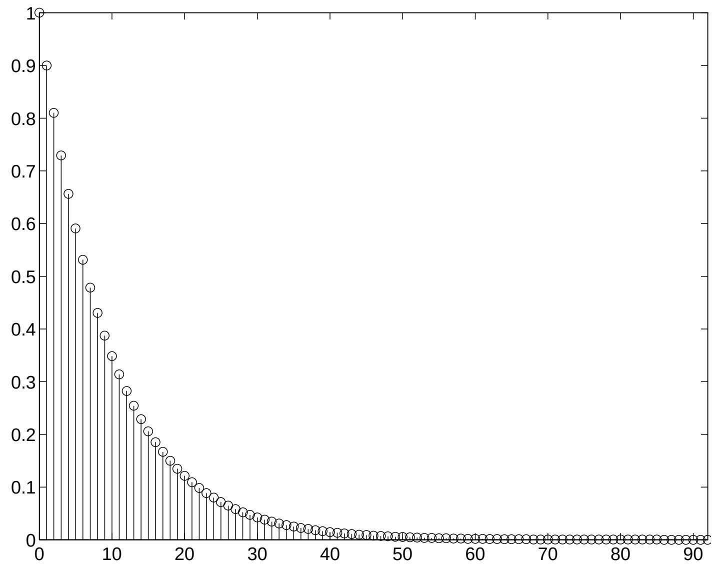
The impulse response of the simple filter `b = 1` and `a = [1 -0.9]` is

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

The `impz` function in the toolbox simplifies this operation, choosing the number of points to generate and then making a stem plot (using the `stem` function).

```
impz(b,a)
```

The plot shows the exponential decay `h(n) = 0.9n` 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.

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

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.

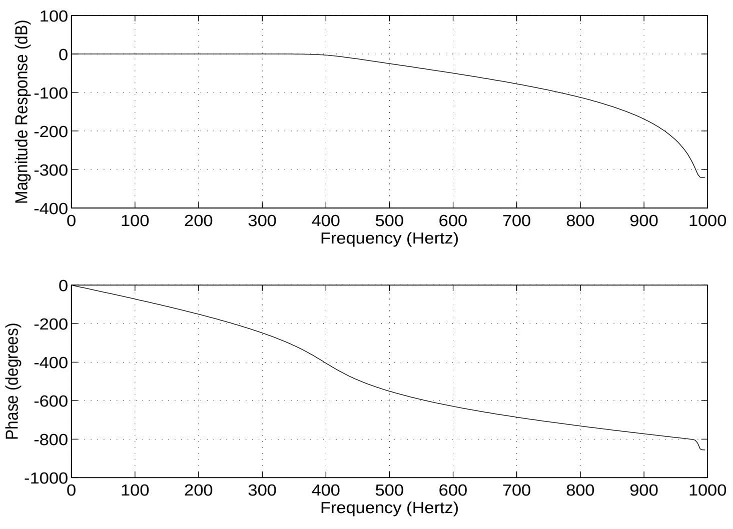
Frequency Normalization This toolbox uses the convention that unit frequency is the Nyquist frequency, defined as half the sampling frequency. The cutoff frequency parameter for all basic filter design functions is normalized by the Nyquist frequency. For a system with a 1000 Hz sampling frequency, for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

If you call `freqz` with no output arguments, it automatically 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);
```

Now calculate the 256-point complex frequency response for this filter, and plot the magnitude and phase with a call to `freqz`.

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



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

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

calculates the complex frequency response at the frequency points in `w` for the filter defined by vectors `b` and `a`. The frequency points can range from 0 to 2π. 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.

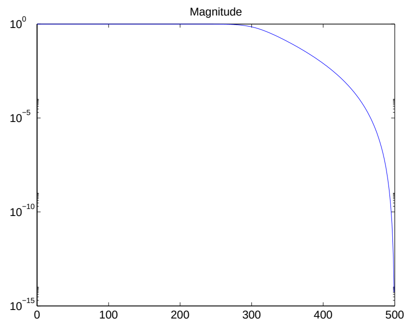
Analog Domain

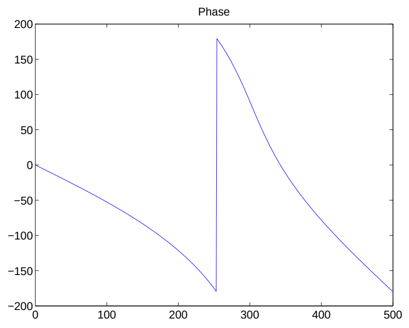
`freqs` evaluates frequency response for an analog filter defined by two input coefficient vectors, `b` and `a`. Its operation is similar to that of `freqz`; you can specify a number of frequency points to use, supply a vector of arbitrary frequency points, and plot the magnitude and phase response of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

```
[b,a] = butter(6,300/500);
[h,w] = freqz(b,a,512,1000);
m = abs(h); p = angle(h);
semilogy(w,m); title('Magnitude');
figure; plot(w,p*180/pi); title('Phase');
```



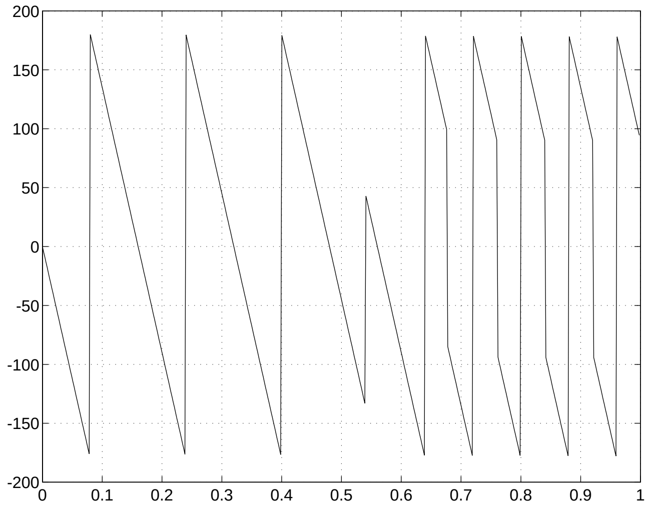


The unwrap function is also useful in frequency analysis. unwrap unwraps the phase to make it continuous across 360° phase discontinuities by adding multiples of ±360°, as needed. To see how unwrap is useful, design a 25th-order lowpass FIR filter.

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

Obtain the filter's frequency response with freqz, and plot the phase in degrees.

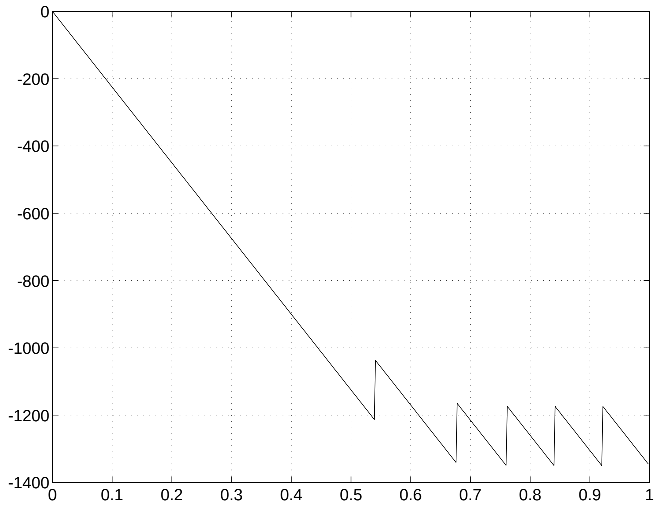
```
[H,f] = freqz(h,1,512,2);
plot(f,angle(H)*180/pi); grid
```



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

Use unwrap to eliminate the 360° jumps.

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



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

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

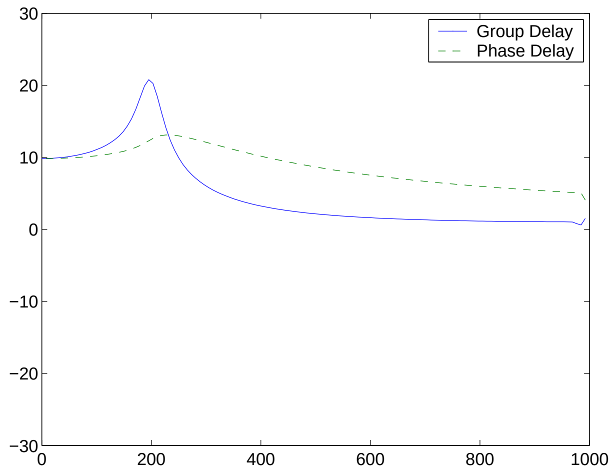
which returns the `n`-point group delay, $\tau_g(\omega)$, of the digital filter specified by `b` and `a`, evaluated at the frequencies in vector `w`.

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 graph, type

```
[b,a] = butter(10,200/1000);
gd = grpdelay(b,a,128);
[h,f] = freqz(b,a,128,2000);
pd = -unwrap(angle(h))*(2000/(2*pi))./f;
plot(f,gd,'-',f,pd,'--')
axis([0 1000 -30 30])
legend('Group Delay','Phase Delay')
```



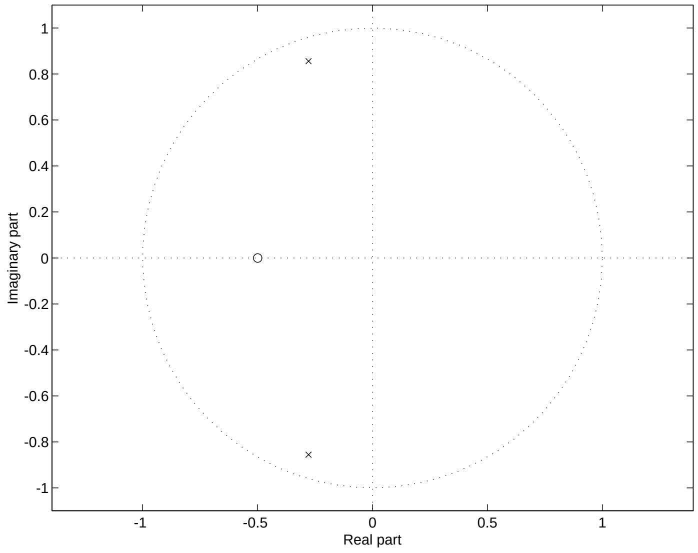
Zero-Pole Analysis

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]');
```

The zero-pole plot for the filter is

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



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 1-33 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"
- "Zero-Pole-Gain"
- "State-Space"
- "Partial Fraction Expansion (Residue Form)"
- "Second-Order Sections (SOS)"
- "Lattice Structure"
- "Convolution Matrix"

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

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

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.

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

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 Contr ol 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

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

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.

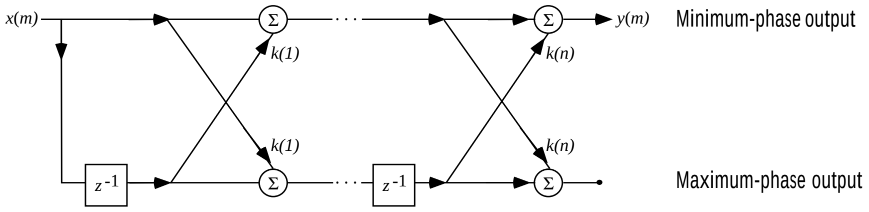
$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & a_{01} & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & a_{02} & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & a_{0L} & a_{1L} & a_{2L} \end{bmatrix}$$

There are an uncountable number of 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 1-43, perform pole-zero pairing, section scaling, and section ordering.

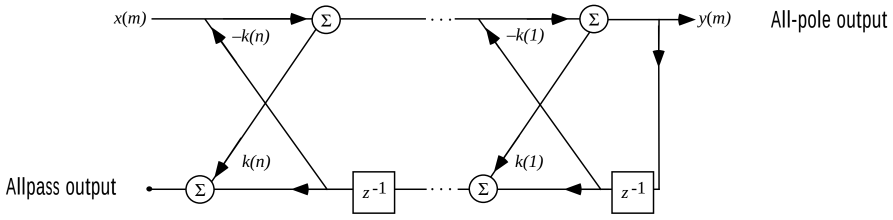
Note In the Signal Processing Toolbox, all second-order section transformations apply only to digital filters.

Lattice Structure

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

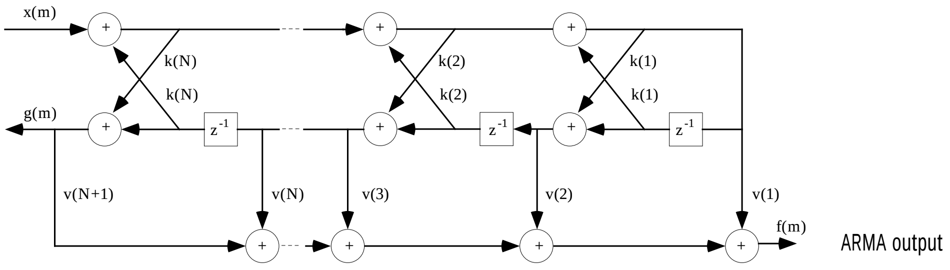


FIR Lattice Filter



IIR Lattice Filter

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   0.6149   0.9899   0.0000   0.0031  -0.0082];
```

This filter's lattice (reflection coefficient) representation is

```
k = 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] = 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 `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

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

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; see Chapter 7, Function Reference for a complete description of each. 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` `residue`	`tf2latc`		`convmtx`
State-space	`ss2tf`		`ss2zp`			`ss2sos`	
Zero-pole-gain	`zp2tf` `poly`	`zp2ss`				`zp2sos`	
Partial fraction	`residuez` `residue`						
Lattice filter	`latc2tf`						
SOS	`sos2tf`	`sos2ss`	`sos2zp`				
Convolution matrix							

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. See the descriptions of the individual filter design functions in Chapter 7, “Function Reference,” for more details.

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 MATLAB's 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.

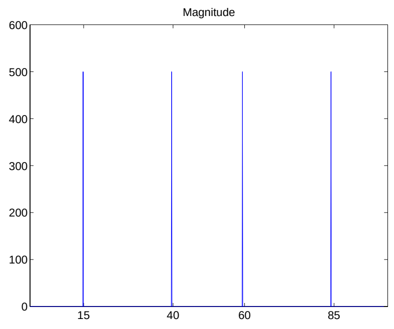
```
t = (0:1/100:10-1/100);                   % Time vector
x = sin(2*pi*15*t) + sin(2*pi*40*t);     % Signal
```

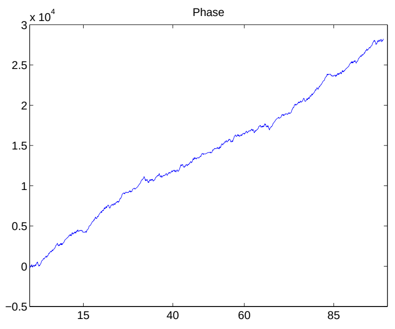
The DFT of the signal, and the magnitude and phase of the transformed sequence, are then

```
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.

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. See the descriptions in Chapter 7, "Function Reference," for details.

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. Schafer. *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] Pratt,W.K. *Digital Image Processing*. New York: John Wiley & Sons, 1991.

[8] Percival, D.B., and A.T. Walden. *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques*. Cambridge: Cambridge University Press, 1993.

[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

Overview

The Signal Processing Toolbox provides functions that support a range of filter design methodologies. The following sections explain how to apply the filter design tools to *Infinite Impulse Response* (IIR) and *Finite Impulse Response* (FIR) filter design problems:

- “Filter Requirements and Specification”
- “IIR Filter Design”
- “FIR Filter Design”
- “Special Topics in IIR Filter Design”
- “Selected Bibliography”

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.

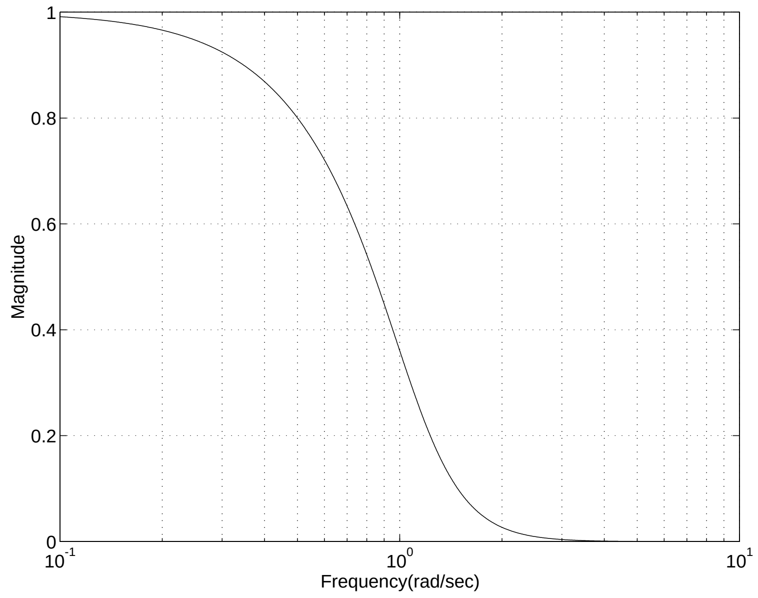
Filter design methods differ primarily in how performance is specified. For “loosely specified” requirements, as in the first case above, a Butterworth IIR filter is often sufficient. To design a fifth-order 30 Hz lowpass Butterworth filter and apply it to the data in vector `x`,

```
[b,a] = butter(5,30/50);
y = filter(b,a,x);
```

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

Frequency Normalization in the Signal Processing Toolbox 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 π. To convert normalized frequency back to hertz, multiply by half the sample frequency.

More rigorous filter requirements traditionally include passband ripple (`Rp`, in decibels), stopband attenuation (`Rs`, in decibels), and transition width (`Ws-Wp`, 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 4-12.

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.

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

Classical IIR Filter Design Using Analog Prototyping

The principal IIR digital filter design technique this toolbox provides is based on the conversion of classical lowpass analog filters to their digital equivalents. The following sections describe how to design filters and summarize the characteristics of the supported filter types. See “Special Topics in IIR Filter Design” on page 2-38 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] = besself(n,Wn,`*`options`*`)`
	`[z,p,k] = besself(n,Wn,`*`options`*`)`
	`[A,B,C,D] = besself(n,Wn,`*`options`*`)`

Butterworth	`[b,a] = butter(n,Wn,`*`options`*`)`
	`[z,p,k] = butter(n,Wn,`*`options`*`)`
	`[A,B,C,D] = butter(n,Wn,`*`options`*`)`

Chebyshev Type I	`[b,a] = cheby1(n,Rp,Wn,`*`options`*`)`
	`[z,p,k] = cheby1(n,Rp,Wn,`*`options`*`)`
	`[A,B,C,D] = cheby1(n,Rp,Wn,`*`options`*`)`

Chebyshev Type II	`[b,a] = cheby2(n,Rs,Wn,`*`options`*`)`
	`[z,p,k] = cheby2(n,Rs,Wn,`*`options`*`)`
	`[A,B,C,D] = cheby2(n,Rs,Wn,`*`options`*`)`

Elliptic	`[b,a] = ellip(n,Rp,Rs,Wn,`*`options`*`)`
	`[z,p,k] = ellip(n,Rp,Rs,Wn,`*`options`*`)`
	`[A,B,C,D] = 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);
```

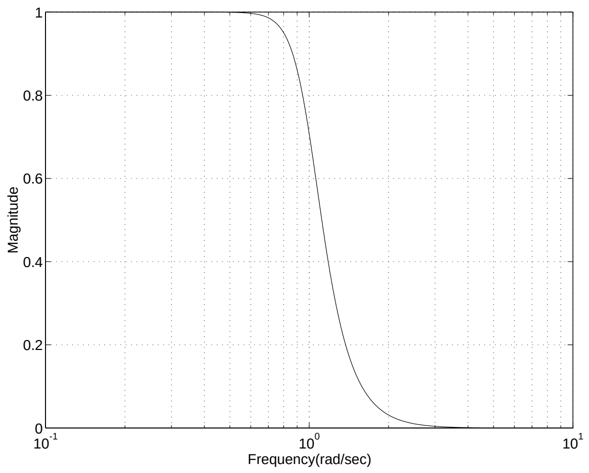
These functions also work with the other standard band configurations, as well as for analog filters; see Chapter 7, “Function Reference,” for details.

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.

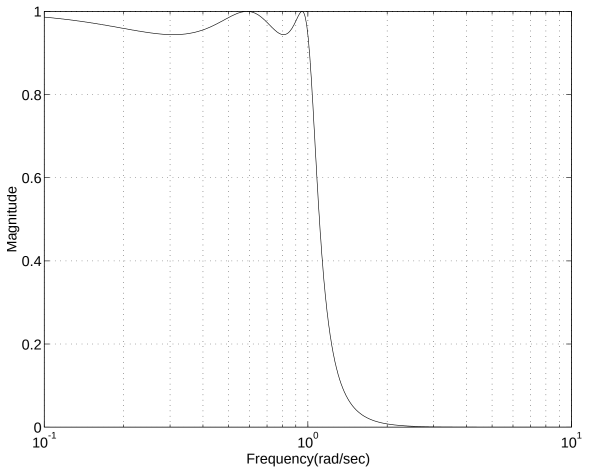
Butterworth Filter

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

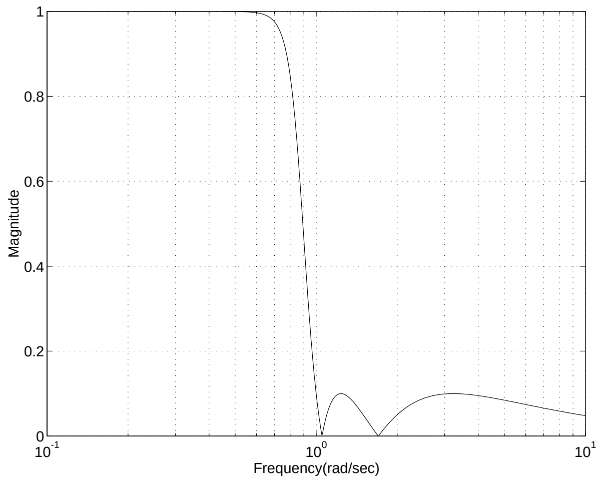
The Chebyshev Type I filter minimizes the absolute difference between the ideal and actual frequency response over the entire passband by incorporating an equal ripple of Rp dB in the passband. Stopband response is maximally flat. The transition from passband to stopband is more rapid than for the Butterworth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



Chebyshev Type II Filter

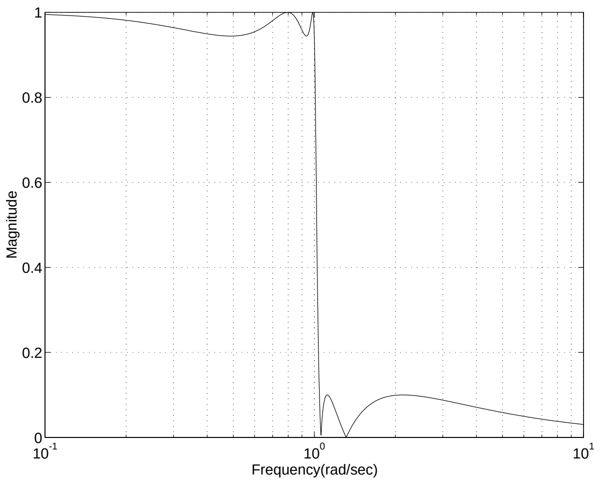
The Chebyshev Type II filter minimizes the absolute difference between the ideal and actual frequency response over the entire stopband by incorporating an equal ripple of Rs dB in the stopband. Passband response is maximally flat.

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



Elliptic Filter

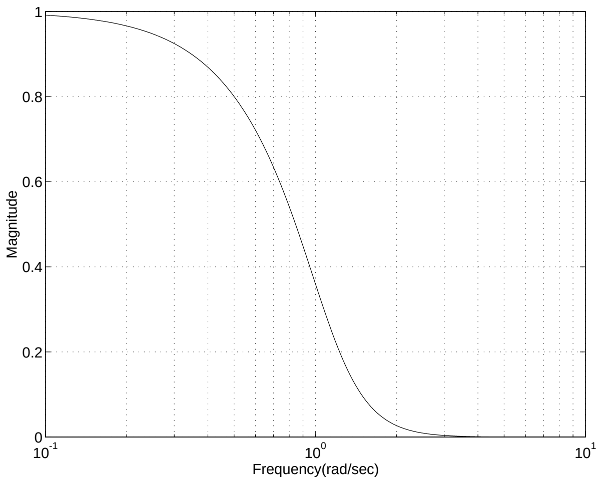
Elliptic filters are equiripple in both the passband and stopband. They generally meet filter requirements with the lowest order of any supported filter type. Given a filter order *n*, passband ripple Rp in decibels, and stopband ripple Rs in decibels, elliptic filters minimize transition width. $|H(j\Omega)|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



Bessel Filter

Analog Bessel lowpass filters have maximally flat group delay at zero frequency and retain nearly constant group delay across the entire passband. Filtered signals therefore maintain their waveshapes in the passband frequency range. 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 2-37 for details.

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

```
[z,p,k] = ellipap(5,0.5,20);
w = logspace(-1,1,1000);
h = freqs(k*poly(z),poly(p),w);
semilogx(w,abs(h)), grid
```

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 4-12 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 power spectrum and solves the "modified Yule-Walker equations" using the resulting autocorrelation function samples. The statement

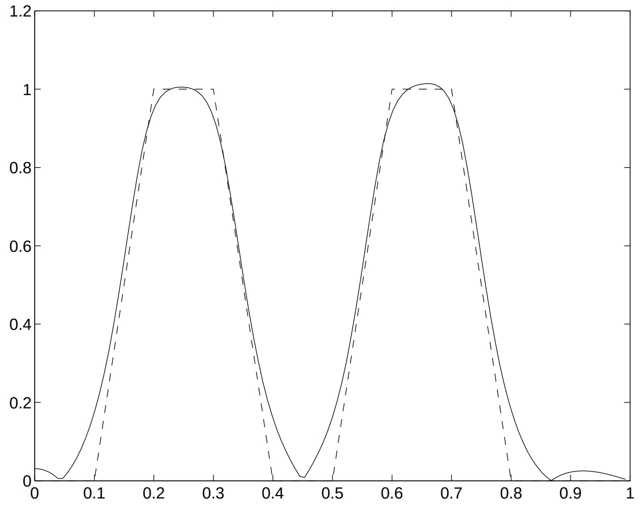
```
[b,a] = yulewalk(n,f,m)
```

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 2-16 for details.

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

Design a multiband filter with `yulewalk`, and plot the 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.

Method	**Description**	**Functions**
Windowing	Apply window to truncated inverse Fourier transform of desired “brick wall” filter	`fir1`, `fir2`, `kaiserord`
Multiband with Transition Bands	Equiripple or least squares approach over sub-bands of the frequency range	`firls`, `remez`, `remezord`
Constrained Least Squares	Minimize squared integral error over entire frequency range subject to maximum error constraints	`fircls`, `fircls1`
Arbitrary Response	Arbitrary responses, including nonlinear phase and complex filters	`cremez`
Raised Cosine	Lowpass response with smooth, sinusoidal transition	`firrcos`

Linear Phase Filters

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

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response H(f), f = 0	Response H(f), f = 1 (Nyquist)
Type I	Even	even: $b(k) = b(n+2-k), \quad k = 1, \ldots, n+1$	No restriction	No restriction
Type II	Odd		No restriction	$H(1) = 0$
Type III	Even	odd: $b(k) = -b(n+2-k), \quad k = 1, \ldots, n+1$	$H(0) = 0$	$H(1) = 0$
Type IV	Odd		$H(0) = 0$	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`, `remez`, `fircls`, `fircls1`, and `firrcos` all design type I and II linear phase FIR filters by default. Both `firls` and `remez` design type III and IV linear phase FIR filters given a `'hilbert'` or `'differentiator'` flag. `cremez` 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 ω_0 rad/s. This filter has magnitude 1 at all frequencies with magnitude less than ω_0, and magnitude 0 at frequencies with magnitude between ω_0 and π. Its impulse response sequence $h(n)$ is

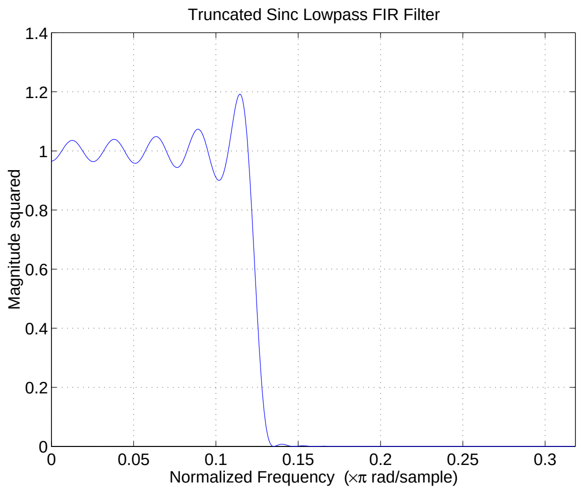
$$h(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H(\omega)e^{j\omega n}\,d\omega = \frac{1}{2\pi}\int_{-\omega_0}^{\omega_0} e^{j\omega n}\,d\omega = \frac{\omega_0}{\pi}\mathrm{sinc}\left(\frac{\omega_0}{\pi}n\right)$$

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 ω_0 of 0.4π rad/s is

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

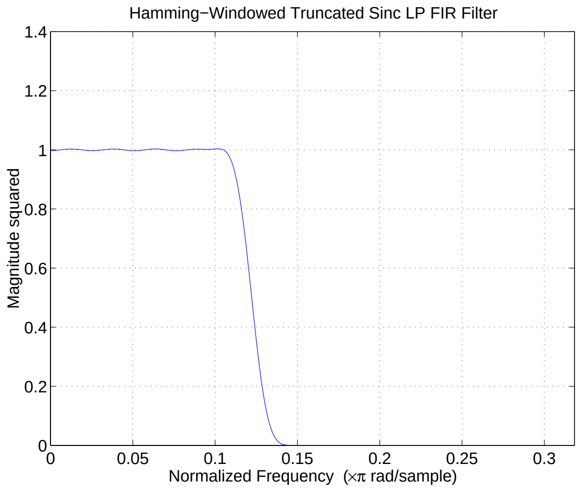
The window applied here is a simple rectangular window. By Parseval’s theorem, this is the length 51 filter that best approximates the ideal lowpass filter, in the integrated least squares sense. The following commands display the filter’s frequency response.

```
[H,w] = freqz(b,1,512,2);
s.xunits = 'rad/sample';
s.yunits = 'squared';
s.plot = 'mag';             % Plot magnitude only.
freqzplot(H,w,s)
title('Truncated Sinc Lowpass FIR Filter');
```



Note the ringing and ripples 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.

```
b = 0.4*sinc(0.4*(-25:25));
b = b.*hamming(51)';
[H,w] = freqz(b,1,512,2);
s.xunits = 'rad/sample';
s.yunits = 'squared';
s.plot = 'mag';
freqzplot(H,w,s)
title('Hamming-Windowed Truncated Sinc LP FIR Filter');
```



As you can see, this 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 “Overview” 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 2-5 for details.)

Multiband FIR Filter Design with Transition Bands

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

Basic Configurations

The default mode of operation of firls and remez 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 = remez(n,f,a);
```

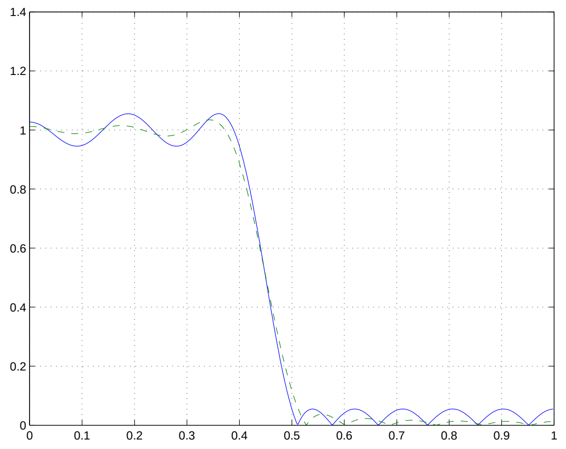
From 0.4 to 0.5 Hz, remez performs no error minimization; this is a transition band or "don't care" region. A transition band minimizes the error more in the bands that you do care about, at the expense of a slower transition rate. In this way, these types of filters have an inherent trade-off similar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

```
[H,w] = freqz(b);
[HH,w] = freqz(bb);
plot(w/pi,abs(H),w/pi,abs(HH),'--'), grid
```



You can see that the filter designed with `remez` 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 `remez` filter. This shows that the `remez` 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. `remez` and `firls` use this scheme to represent any piecewise linear desired function with any transition bands. `firls` and `remez` 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 remez 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

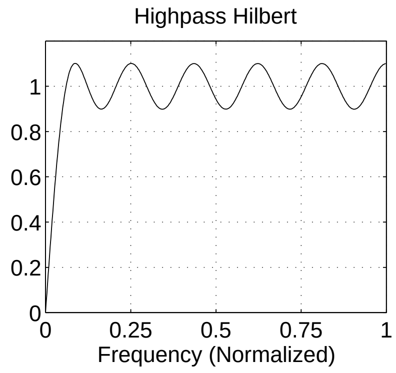
```
n = 20;              % Filter order
f = [0 0.4 0.5 1];   % Frequency band edges
a = [1  1   0  0];   % Desired amplitudes
w = [1 10];          % Weight vector
b = remez(n,f,a,w);
```

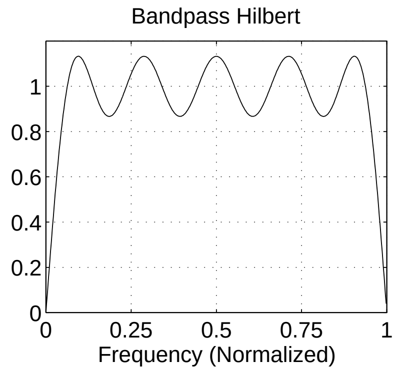
A legal weight vector is always half the length of the f and a vectors; there must be exactly one weight per band.

Anti-Symmetric Filters / Hilbert Transformers

When called with a trailing 'h' or 'Hilbert' option, remez 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.

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





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 4-36, 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 `remez` or `firls` with a `'d'` or `'differentiator'` option.

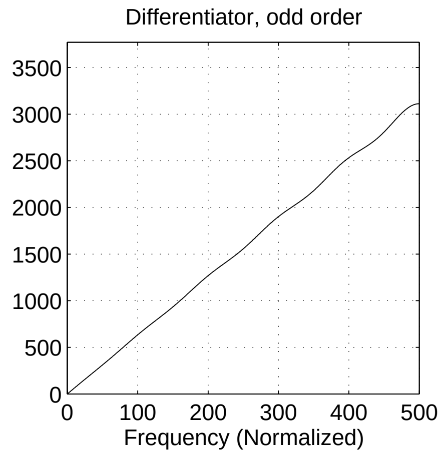
```
b = remez(21,[0 1],[0 pi*fs],'d');
```

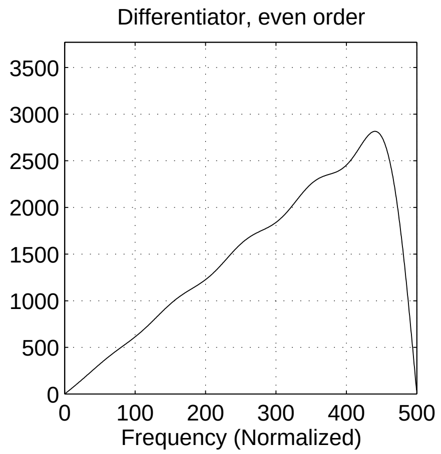
To obtain the correct derivative, scale by `pi*fs` rad/s, where `fs` is the sampling frequency in hertz. For a type III filter, the differentiation band should stop short of the Nyquist frequency, and the amplitude vector must reflect that change to ensure the correct slope.

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

In the `'d'` mode, `remez` 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 response for the differentiators above.





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.

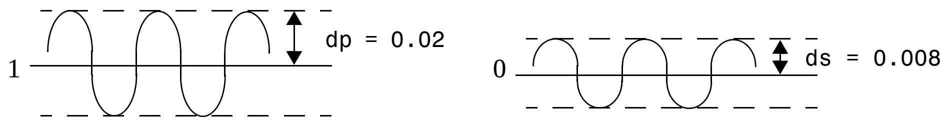
Description	Function
Constrained least square multiband FIR filter design	`fircls`
Constrained least square filter design for lowpass and highpass linear phase filters	`fircls1`

For details on the calling syntax for these functions, see their reference descriptions in Chapter 7, "Function Reference."

Basic Lowpass and Highpass CLS Filter Design

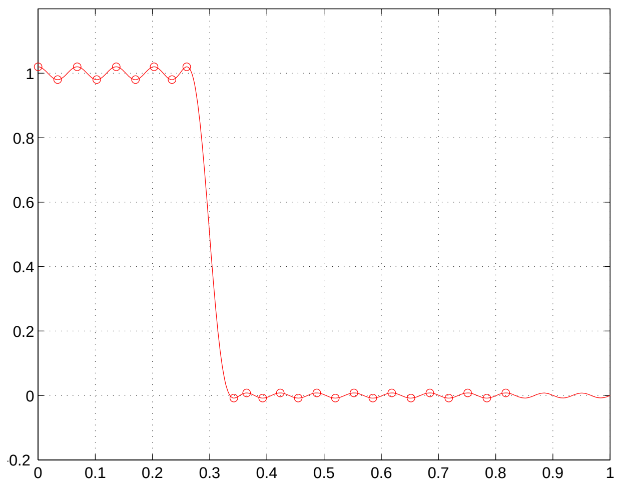
The most basic of the CLS design functions, `fircls1`, uses this technique to design lowpass and highpass FIR filters. As an example, consider designing a filter with order 61 impulse response and cutoff frequency of 0.3 (normalized). Further, define the upper and lower bounds that constrain the design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



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.

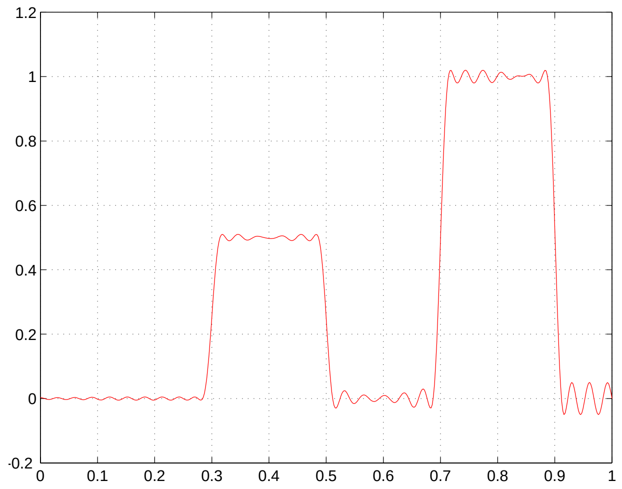
For example, assume the specifications for a filter call for:

- 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,'plot');
```



Weighted CLS Filter Design

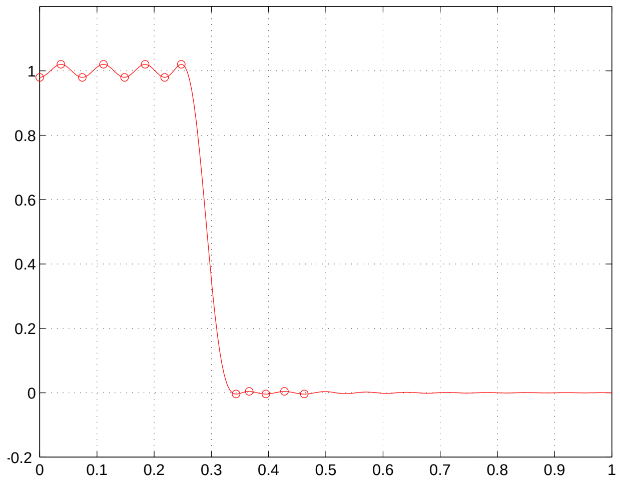
Weighted CLS filter design lets you design lowpass or highpass FIR filters with relative weighting of the error minimization in each band. The `fircls1` function enables you to specify the passband and stopband edges for the least squares weighting function, as well as a constant `k` that specifies the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



Arbitrary-Response Filter Design

The `cremez` 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 `cremez` 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 = cremez(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 `remez`.

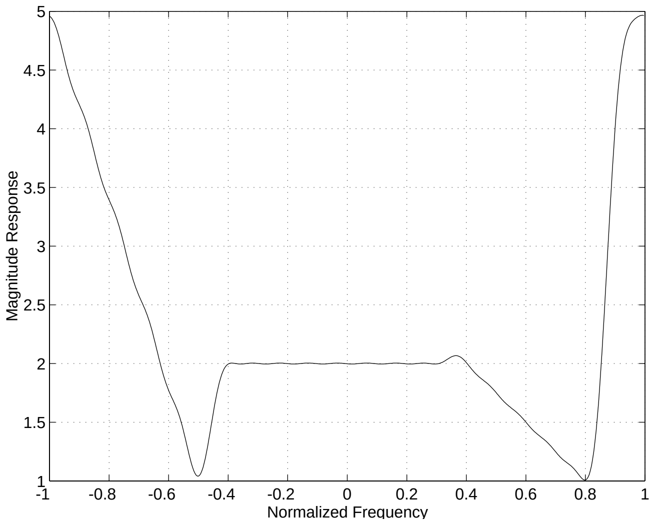
```
b = cremez(38,[-1 -0.5 -0.4 0.3 0.4 0.8], ...
              [5 1 2 2 2 1], [1 10 5]);
```

As with `remez`, a vector of band edges is passed to `cremez`. 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.

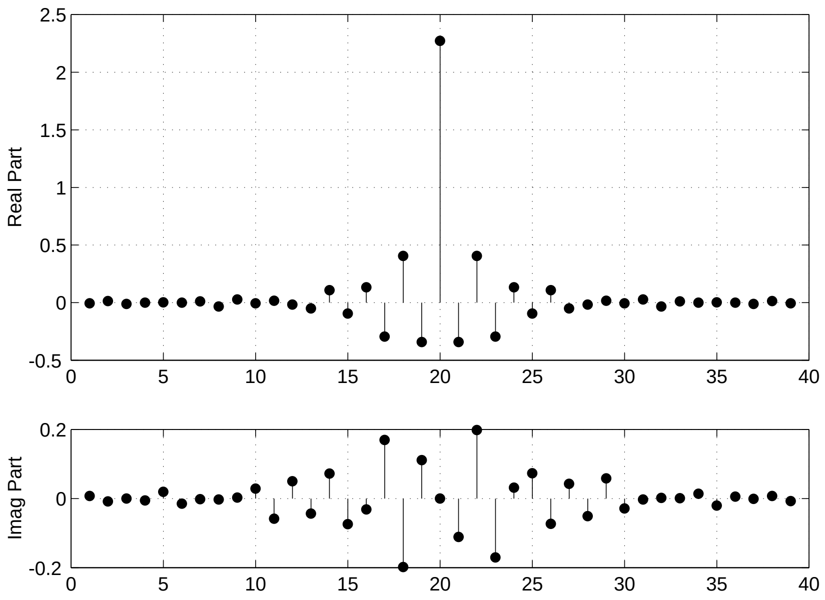
In either case, the frequency response is obtained and plotted using linear scale.

```
[h,w] = freqz(b,1,512,'whole');
plot(w/pi-1,fftshift(abs(h))); grid;
xlabel('Normalized Frequency');
ylabel('Magnitude Response');
```

Note that the frequency response has been calculated over the entire normalized frequency range [-1 +1] by passing the option `'whole'` to `freqz`. In order to plot the negative frequency information in a natural way, the response has been "wrapped," just as FFT data is, using `fftshift`.



The filter response for this multiband filter is complex, which is expected because of the asymmetry in the frequency domain.



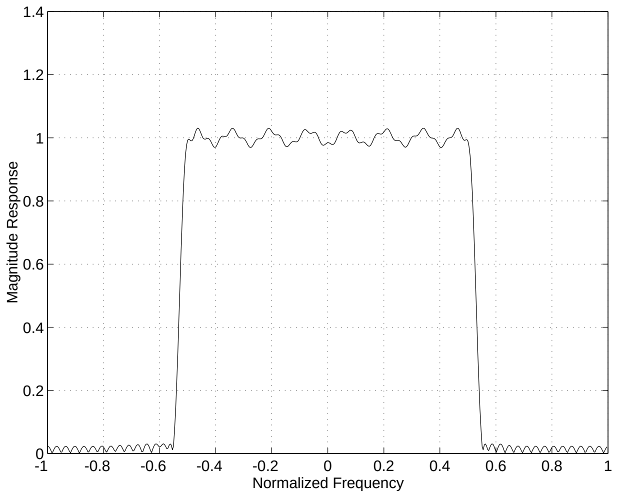
Filter Design with Reduced Delay

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.

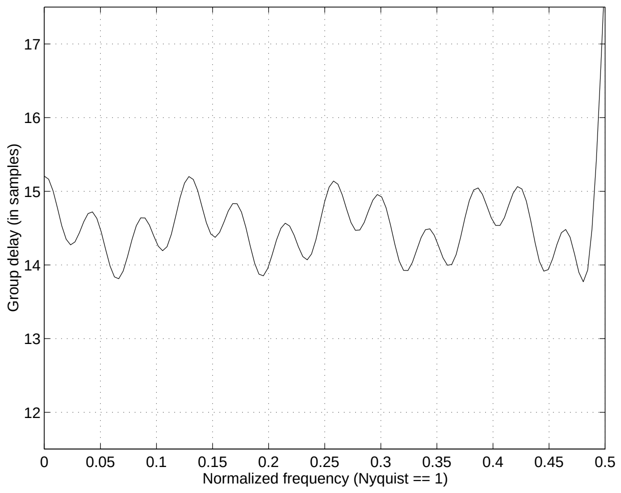
```
b = cremez(61,[0 0.5 0.55 1],{'lowpass',-16});
```

The resulting magnitude response is

```
[h,w] = freqz(b,1,512,'whole');
plot(w/pi-1,fftshift(abs(h))); grid;
xlabel('Normalized Frequency');
ylabel('Magnitude Response');
```



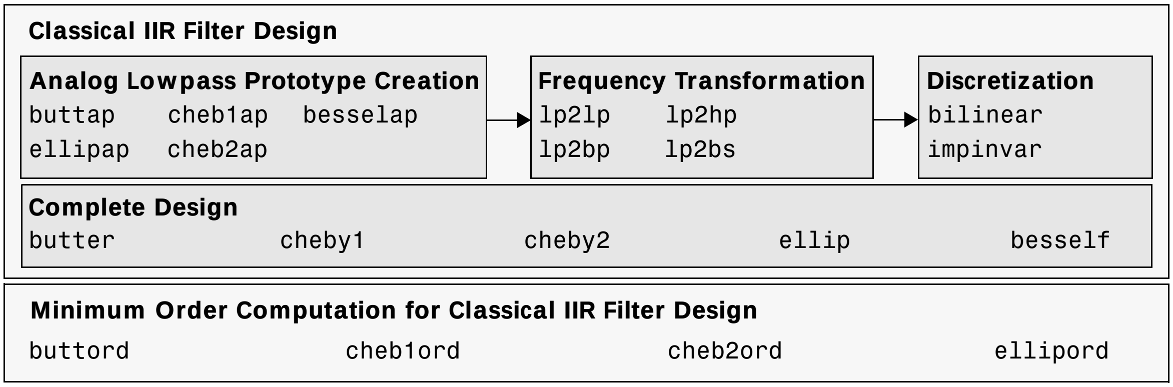
The group delay of the filter reveals that the offset has been reduced from `N/2` to `N/2-16` (i.e., from `30.5` to `14.5`). Now, however, the group delay is no longer flat in the passband region (plotted over the normalized frequency range 0 to 0.5 for clarity).



If we compare this nonlinear-phase filter to a linear-phase filter that has exactly 14.5 samples of group delay, the resulting filter is of order 2*14.5, or 29. Using `b = cremez(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 finds an analog lowpass filter with cutoff frequency of 1, translates this “prototype” filter to the desired band configuration, then transforms the filter to the digital domain. The toolbox provides functions for each step of this process.


Classical IIR Filter Design
Analog Lowpass Prototype Creation
buttap cheb1ap besselap
ellipap cheb2ap
Frequency Transformation
lp2lp lp2hp
lp2bp lp2bs
Discretization
bilinear
impinvar
Complete Design
butter cheby1 cheby2 ellip besself
Minimum Order Computation for Classical IIR Filter Design
buttord cheb1ord cheb2ord ellipord

The `butter`, `cheby1`, `cheby2`, and `ellip` 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 2-5.

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

Frequency Transformation

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

Freq. Transformation	**Transformation Function**
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 $s' = \frac{\omega_0}{B_\omega}\frac{(s/\omega_0)^2+1}{s/\omega_0}$	`[numt,dent]    = lp2bp(num,den,Wo,Bw)` `[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw)`
Lowpass to bandstop $s' = \frac{B_\omega}{\omega_0}\frac{s/\omega_0}{(s/\omega_0)^2+1}$	`[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 ω_1 is the lower band edge and ω_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 Ω_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.

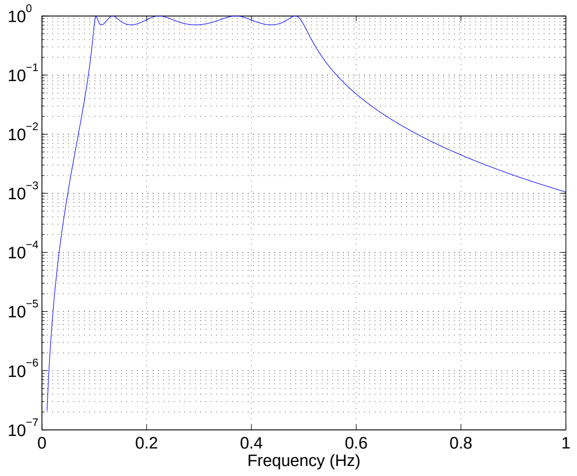
```
[A,B,C,D] = zp2ss(z,p,k); % Convert to state-space form.
```

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	`[numd,dend] = impinvar(num,den,fs)`

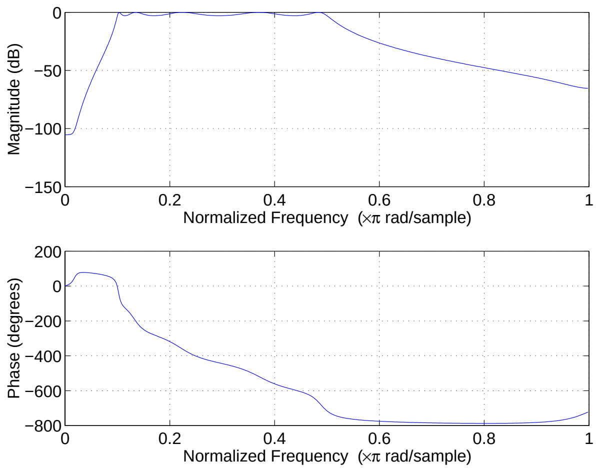
Bilinear transform	`[zd,pd,kd] = bilinear(z,p,k,fs,Fp)`
	`[numd,dend] = bilinear(num,den,fs,Fp)`
	`[Ad,Bd,Cd,Dd] = bilinear(At,Bt,Ct,Dt,fs,Fp)`

Impulse Invariance

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

Design a Chebyshev Type I filter and plot its frequency response.

```
[bz,az] = impinvar(b,a,2);
[H,w] = freqz(bz,az);
freqzplot(H,w)
```



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

Bilinear Transformation

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

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

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

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

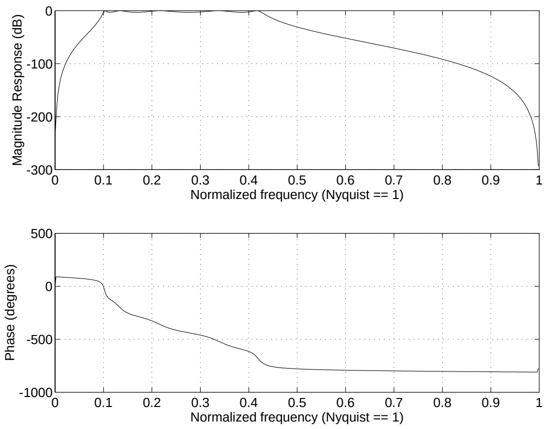
The toolbox function `bilinear` implements this operation, where the frequency warping constant k is equal to twice the sampling frequency (`2*fs`) by default, and equal to $2\pi f_p / \tan(\pi f_p / f_s)$ if you give `bilinear` a trailing argument that represents a "match" frequency `Fp`. If a match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

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

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

The frequency response of the resulting digital filter is

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



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. See Chapter 7, "Function Reference," for more on these functions.

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.

3

Statistical Signal Processing

Overview

The Signal Processing Toolbox provides tools for estimating important functions of random signals. In particular, there are tools to estimate correlation and covariance sequences and spectral density functions of discrete signals. The following sections explain the correlation and covariance functions and discuss the mathematically related functions for estimating the power spectrum:

- “Correlation and Covariance”
- “Spectral Analysis”
- “Selected Bibliography”

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)

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

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 Nth element is the correlation at lag 0. Also notice the triangular pulse of the output that results when convolving two square pulses.

The `xcov` function estimates autocovariance and cross-covariance sequences. This function has the same options and evaluates the same sum as `xcorr`, but first removes the means of `x` and `y`.

Bias and Normalization

An estimate of a quantity is *biased* if its expected value is not equal to the quantity it estimates. The expected value of the output of `xcorr` is

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

`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 `norm(x)*norm(y)` so that, for autocorrelations, the sample at zero lag is 1.

Multiple Channels

For a multichannel signal, `xcorr` and `xcov` estimate the autocorrelation and cross-correlation and covariance sequences for all of the channels at once. If `S` is an M-by-N signal matrix representing N channels in its columns, `xcorr(S)` returns a $(2M\text{-}1)$-by-N^2 matrix with the autocorrelations and cross-correlations of the channels of `S` in its N^2 columns. If `S` is a three-channel signal

```
S = [s1 s2 s3]
```

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

```
R = [Rs1s1 Rs1s2 Rs1s3 Rs2s1 Rs2s2 Rs2s3 Rs3s1 Rs3s2 Rs3s3]
```

Two related functions, `cov` and `corrcoef`, are available in the standard MATLAB environment. They estimate covariance and normalized covariance respectively between the different channels at lag 0 and arrange them in a square matrix.

Spectral Analysis

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 spectrum* 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

$$S_{xx}(\omega) = \sum_{m=-\infty}^{\infty} R_{xx}(m)e^{-j\omega m}$$

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.

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

The correlation sequence can be derived from the power spectrum by use of the inverse discrete-time Fourier transform.

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

The average power of the sequence x_n over the entire Nyquist interval is represented by

$$R_{xx}(0) = \int_{-\pi}^{\pi} \frac{S_{xx}(\omega)}{2\pi}d\omega = \int_{-f_s/2}^{f_s/2} \frac{S_{xx}(f)}{f_s}df$$

The quantities

$$P_{xx}(\omega) = \frac{S_{xx}(\omega)}{2\pi} \qquad \text{and} \qquad P_{xx}(f) = \frac{S_{xx}(f)}{f_s}$$

from the above expression are defined as the *power spectral density* (PSD) of the stationary random signal x_n.

The average power of a signal over a particular frequency band $[\omega_1,\omega_2]$, $0 \le \omega_1 < \omega_2 \le \pi$, can be found by integrating the PSD over that band.

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

You can see from the above expression that $P_{xx}(\omega)$ represents the power content of a signal in an *infinitesimal* frequency band, which is why we call it 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/rad/sample or simply watts/rad. 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 $0 \le \omega < \pi$ is sufficient to completely characterize the PSD. However, in order 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

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

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

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

Spectral Estimation Method Overview

The various methods of spectrum estimation available in the Signal Processing Toolbox can be categorized as follows:

- Nonparametric methods
- Parametric methods
- Subspace methods

Nonparametric methods are those in which the estimate of the PSD is made 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 signal whose PSD we want to estimate 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 names. More information about each function is on the corresponding function reference page. See "Parametric Modeling" on page 4-12 for details about `lpc` and other parametric estimation functions.

Method	**Description**	**Functions**
Periodogram	Power spectral density estimate	`periodogram`
Welch	Averaged periodograms of overlapped, windowed signal sections	`pwelch`, `csd`, `tfe`, `cohere`
Multitaper)	Spectral estimate from combination of multiple orthogonal windows (or “tapers”)	`pmtm`
Yule-Walker AR	Autoregressive (AR) spectral estimate of a time-series from its estimated autocorrelation function	`pyulear`
Burg	Autoregressive (AR) spectral estimation of a time-series by minimization of linear prediction errors	`pburg`
Covariance	Autoregressive (AR) spectral estimation of a time-series by minimization of the forward prediction errors	`pcov`
Modified Covariance	Autoregressive (AR) spectral estimation of a time-series by minimization of the forward and backward prediction errors	`pmcov`
MUSIC	Multiple signal classification	`pmusic`
Eigenvector	Pseudospectrum estimate	`peig`

Nonparametric Methods

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

The Periodogram

One way of estimating the power spectrum 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

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

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

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

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 xn, 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

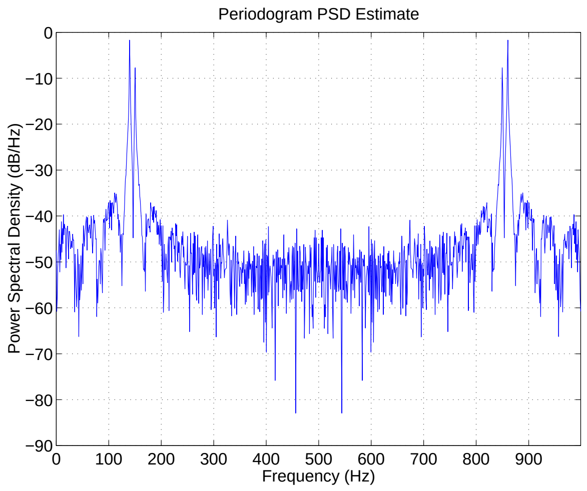
```
xn = sin(2*pi*150*t) + 2*sin(2*pi*140*t) + 0.1*randn(size(t));
```

The periodogram estimate of the PSD can be computed by

```
Pxx = periodogram(xn,[],'twosided',1024,fs);
```

and a plot of the estimate can be displayed by simply omitting the output argument, as below.

```
periodogram(xn,[],'twosided',1024,fs);
```



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

```
Pow = (fs/length(Pxx)) * sum(Pxx)

Pow =

    2.5028
```

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

```
Pxxo = periodogram(xn,[],1024,fs);

Pow = (fs/(2*length(Pxxo))) * sum(Pxxo)

Pow =

    2.4979
```

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 power spectrum or 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

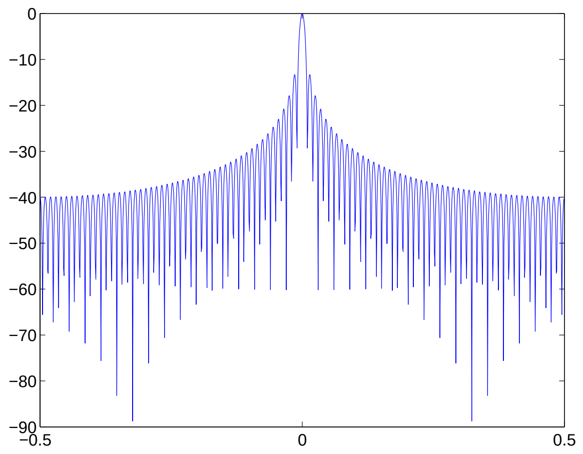
$$X(f) = f_s \sum_{k=1}^{M} A_k \delta(f - f_k)$$

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.

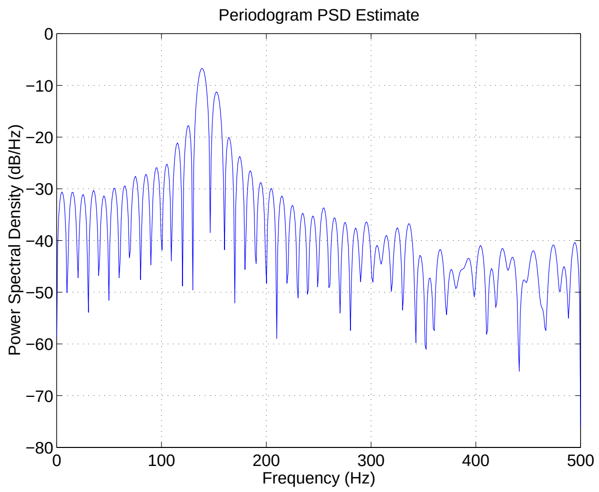
The frequency response of a rectangular window has the shape of a sinc signal, as shown below.



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));
periodogram(xn,[],1024,fs);
```



It is important to note that the effect of spectral leakage is contingent solely on the length of the data record. It is *not* a consequence of the fact that the periodogram is computed at a finite number of frequency samples.

Resolution. *Resolution* refers to the ability to discriminate spectral features, and is a key concept on the analysis of spectral estimator performance.

In order to resolve two sinusoids that are relatively close together in frequency, it is necessary for the difference between the two frequencies to be greater than the width of the mainlobe of the leaked spectra for either one of these sinusoids. The mainlobe width is defined to be the width of the mainlobe at the point where the power is half the peak mainlobe power (i.e., 3 dB width). This width is approximately equal to f_s / L.

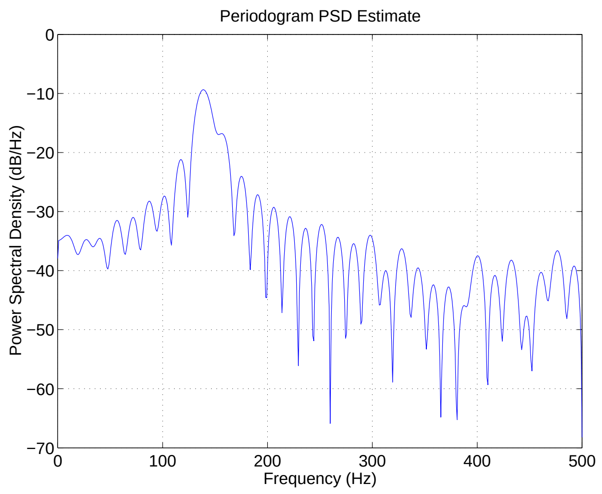
In other words, for two sinusoids of frequencies f_1 and f_2, the resolvability condition requires that

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

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.

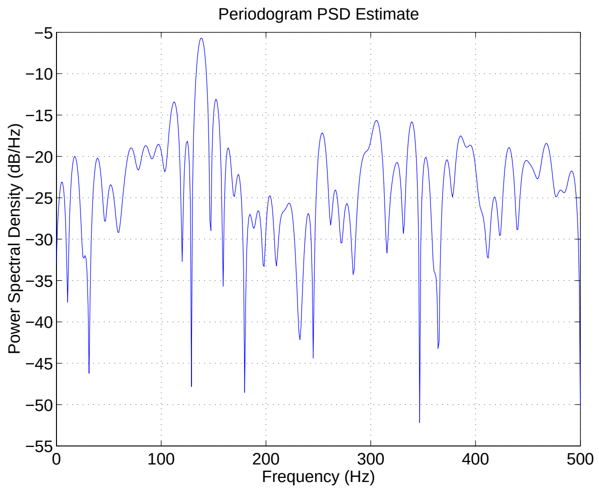
Consider a case where this criterion is not met, as for the sequence of 67 samples below.

```
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));
periodogram(xn,[],1024,fs);
```



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

```
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));
periodogram(xn,[],1024,fs);
```



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 3-12, 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

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

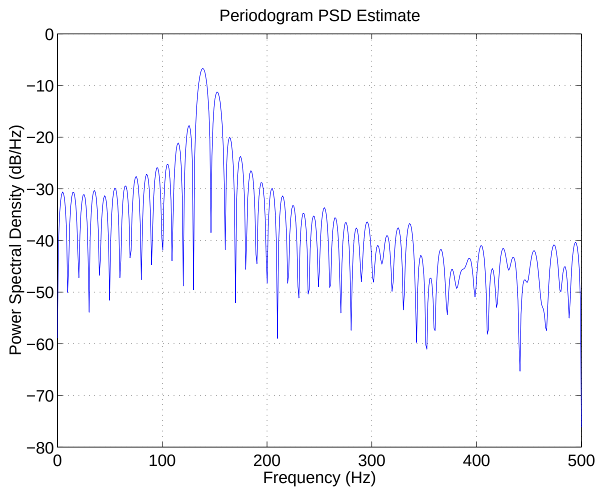
which indicates that the variance does not tend to zero as the data length L tends to infinity. In statistical terms, the periodogram is not a consistent estimator of the PSD. Nevertheless, the periodogram can be a useful tool for spectral estimation in situations where the SNR is high, and especially if the data record is long.

The Modified Periodogram

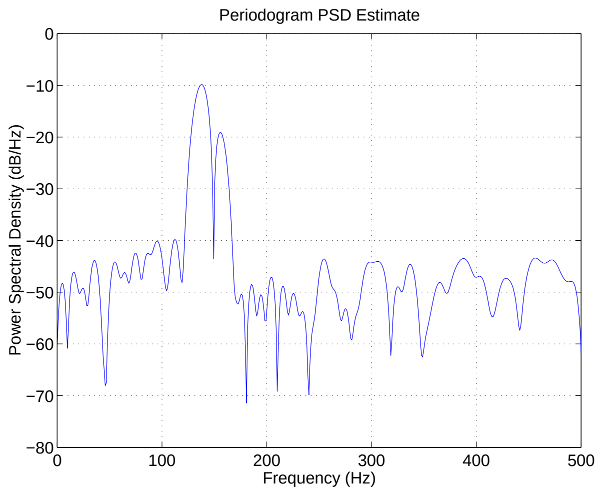
The *modified periodogram* windows the time-domain signal prior to computing the FFT in order to smooth the edges of the signal. This has the effect of reducing the height of the sidelobes or spectral leakage. This phenomenon gives rise to the interpretation of sidelobes as spurious frequencies introduced 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 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));
periodogram(xn,rectwin(length(xn)),1024,fs);
```



```
periodogram(xn,hamming(length(xn)),1024,fs);
```



You can verify that although the sidelobes are much less evident in the Hamming-windowed periodogram, the two main peaks are wider. In fact, the 3 dB width of the mainlobe corresponding to a Hamming window is approximately twice that of a rectangular window. Hence, for a fixed data length, the PSD resolution attainable with a Hamming window is approximately half that attainable with a rectangular window. The competing interests of mainlobe width and sidelobe height can be resolved to some extent by using variable windows such as the Kaiser window.

Nonrectangular windowing affects the average power of a signal because some of the time samples are attenuated when multiplied by the window. To compensate for this, 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.

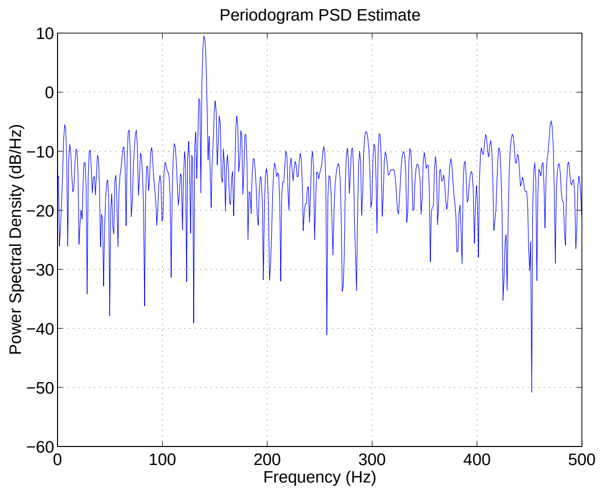
Welch's method is implemented in the Signal Processing Toolbox by the `pwelch` function. By default, the data is divided into eight segments with 50% overlap between them. A Hamming window is used to compute the modified periodogram of each segment.

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.

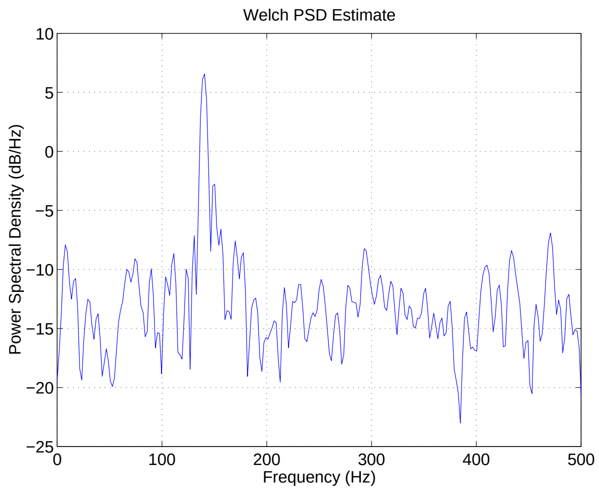
Consider an original signal consisting of 301 samples.

```
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));
periodogram(xn,rectwin(length(xn)),1024,fs);
```



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

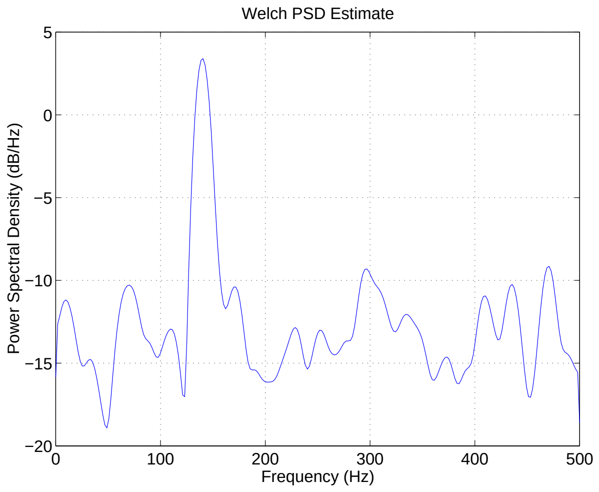
```
pwelch(xn,rectwin(150),75,512,fs);
```



In the periodogram above, noise and the leakage make one of the sinusoids essentially indistinguishable from the artificial peaks. In contrast, although the PSD produced by Welch's method has wider peaks, you can still distinguish the two sinusoids, which stand out from the "noise floor."

However, if we try to reduce the variance further, the loss of resolution causes one of the sinusoids to be lost altogether.

```
pwelch(xn,hamming(100),75,512,fs);
```



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 3-12. 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.

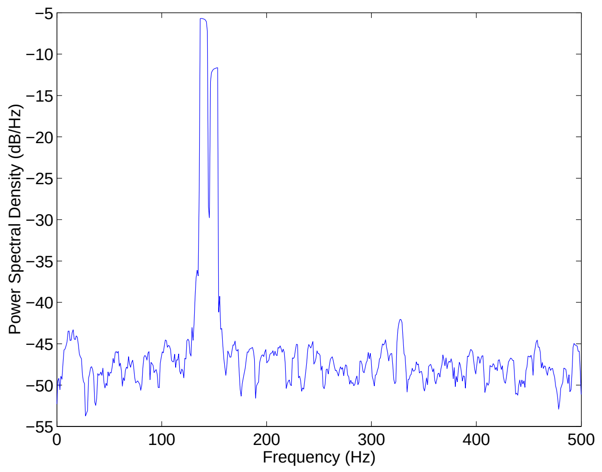
Welch's method can be given a similar interpretation in terms of a filter bank. In Welch's implementation, several samples are used to compute the output power, resulting in reduced variance of the estimate. On the other hand, the bandwidth of each bandpass filter is larger than that corresponding to 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.

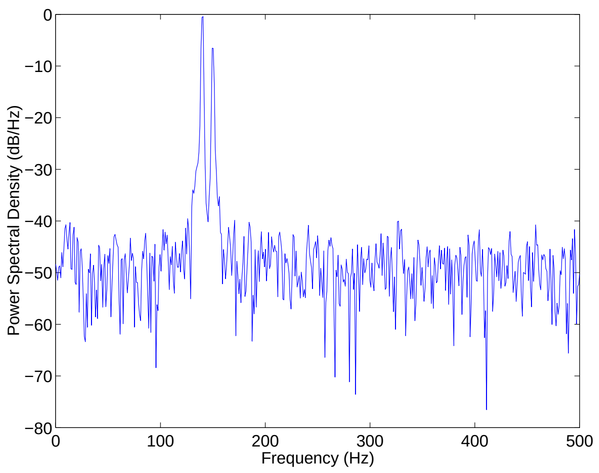
The Signal Processing Toolbox function that implements the MTM method is called `pmtm`. Use `pmtm` to compute the PSD of `xn` from the previous examples.

```
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));

[P,F] = pmtm(xn,4,1024,fs);
plot(F,10*log10(P))           % Plot in dB/Hz
xlabel('Frequency (Hz)');
ylabel('Power Spectral Density (dB/Hz)');
```



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

```
[P1,f] = pmtm(xn,3/2,1024,fs);
plot(f,10*log10(P1))        % Plot in dB/Hz
xlabel('Frequency (Hz)');
ylabel('Power Spectral Density (dB/Hz)');
```



Note that the average power is conserved in both cases.

```
Pow = (fs/1024) * sum(P)

Pow =

    2.4926

Pow1 = (fs/1024) * sum(P1)

Pow1 =

    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 `dpsssave`, `dpssload`, `dpssdir`, and `dpssclear` are provided to keep a database of saved DPSSs in the MAT-file `dpss.mat`.

Cross-Spectral Density Function

The PSD is a special case of the *cross spectral density* (CSD) function, defined between two signals x_n and y_n as

$$S_{xy}(\omega) = \sum_{m=-\infty}^{\infty} R_{xy}(m)e^{-j\omega m}$$

As is the case for the correlation and covariance sequences, the toolbox *estimates* the PSD and CSD because signal lengths are finite.

To estimate the cross-spectral density of two equal length signals `x` and `y` using Welch's method, the `csd` 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 CSD is a complex function. `csd` handles the sectioning and windowing of `x` and `y` in the same way as the `pwelch` function.

```
Sxy = csd(x,y,nfft,fs,window,numoverlap)
```

Confidence Intervals

You can compute confidence intervals using the `csd` function by including an additional input argument `p` that specifies the percentage of the confidence interval, and setting the `numoverlap` argument to 0.

```
[Sxy,Sxyc,f] = csd(x,y,nfft,fs,window,0,p)
```

`p` must be a scalar between 0 and 1. This function assumes chi-squared distributed periodograms of the nonoverlapping sections of windowed data in computing the confidence intervals. This assumption is valid when the signal is a Gaussian distributed random process. Provided these assumptions are correct, the confidence interval

```
[Sxy-Sxyc(:,1) Sxy+Sxyc(:,2)]
```

covers the true CSD with probability `p`. If you set `numoverlap` to any value other than `0`, you generate a warning indicating that the sections overlap and the confidence interval is not reliable.

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 CSD of $x(n)$ and $y(n)$ by

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

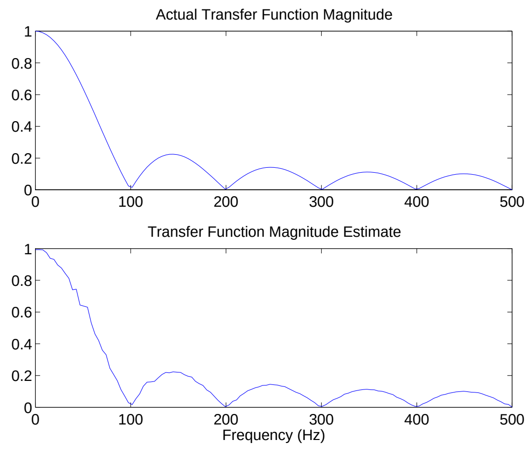
An estimate of the transfer function between $x(n)$ and $y(n)$ is

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

This method estimates both magnitude and phase information. The `tfe` function uses Welch's method to compute the CSD and power spectrum, and then forms their quotient for the transfer function estimate. Use `tfe` the same way that you use the `csd` 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] = tfe(xn,yn,256,fs,256,128,'none');
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

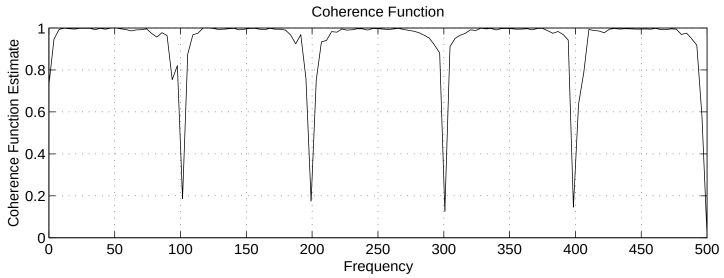
$$C_{xy}(\omega) = \frac{|S_{xy}(\omega)|^2}{S_{xx}(\omega)S_{yy}(\omega)}$$

This quotient is a real number between 0 and 1 that measures the correlation between $x(n)$ and $y(n)$ at the frequency ω.

The `cohere` function takes sequences `x` and `y`, computes their power spectra and CSD, and returns the quotient of the magnitude squared of the CSD and the product of the power spectra. Its options and operation are similar to the `csd` and `tfe` functions.

The coherence function of `xn` and the filter output `yn` versus frequency is

```
cohere(xn,yn,256,fs,256,128,'none')
```



If the input sequence length `nfft`, window length `window`, and the number of overlapping data points in a window `numoverlap`, are such that `cohere` operates on only a single record, the function returns all ones. This is because the coherence function for linearly dependent data is one.

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

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

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.

	Burg	**Covariance**	**Modified Covariance**	**Yule-Walker**
Characteristics	Does not apply window to data	Does not apply window to data	Does not apply window to data	Applies window to data
	Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion	Minimizes the forward prediction error in the least squares sense	Minimizes the forward and backward prediction errors in the least squares sense	Minimizes the forward prediction error in the least squares sense (also called "Autocorrelation method")
Advantages	High resolution for short data records	Better resolution than Y-W for short data records (more accurate estimates)	High resolution for short data records	Performs as well as other methods for large data records
	Always produces a stable model	Able to extract frequencies from data consisting of p or more pure sinusoids	Able to extract frequencies from data consisting of p or more pure sinusoids	Always produces a stable model
			Does not suffer spectral line-splitting	

	Burg	Covariance	Modified Covariance	Yule-Walker
Disadvantages	Peak locations highly dependent on initial phase	May produce unstable models	May produce unstable models	Performs relatively poorly for short data records
	May suffer spectral line-splitting for sinusoids in noise, or when order is very large	Frequency bias for estimates of sinusoids in noise	Peak locations slightly dependent on initial phase	Frequency bias for estimates of sinusoids in noise
	Frequency bias for estimates of sinusoids in noise		Minor frequency bias for estimates of sinusoids in noise	
Conditions for Nonsingularity		Order must be less than or equal to half the input frame size	Order must be less than or equal to 2/3 the input frame size	Because of the biased estimate, the autocorrelation matrix is guaranteed to positive-definite, hence nonsingular

Yule-Walker AR Method

The *Yule-Walker AR method* of spectral estimation computes the AR parameters by 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

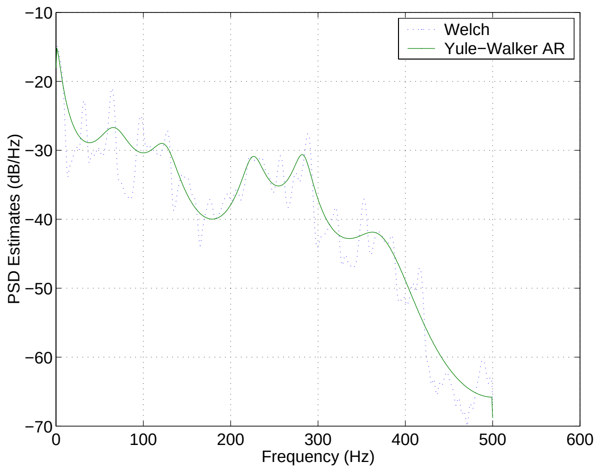
$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \cdots & r(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(p+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(p+1) \end{bmatrix}$$

The 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 function `pyulear` implements the Yule-Walker AR method.

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

```
load mtlb
[P1,f] = pwelch(mtlb,hamming(256),128,1024,fs);
[P2,f] = pyulear(mtlb,14,1024,fs);
plot(f,10*log10(P1),':',f,10*log10(P2)); grid
ylabel('PSD Estimates (dB/Hz)');
xlabel('Frequency (Hz)');
legend('Welch','Yule-Walker AR')
```



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

Burg Method

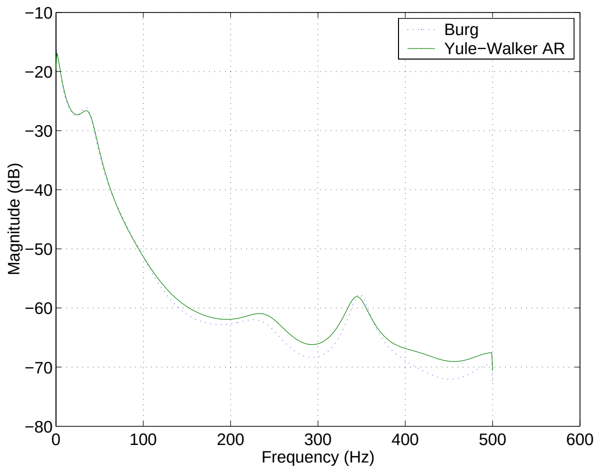
The Burg method for AR spectral estimation is based on minimizing the forward and backward prediction errors while satisfying the Levinson-Durbin recursion (see Marple [3], Chapter 7, and Proakis [6], Section 12.3.3). In contrast to other AR estimation methods, the Burg method avoids calculating the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

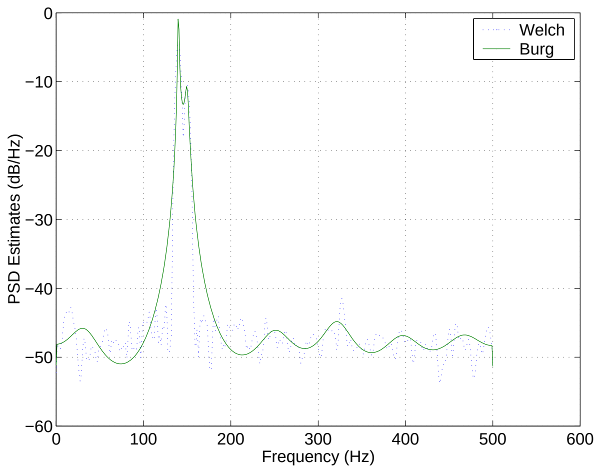
The toolbox function pburg implements the Burg method. Compare the spectrum of the speech signal generated by both the Burg method and the Yule-Walker AR method. They are very similar for large signal lengths.

```
load mtlb
[P1,f] = pburg(mtlb(1:512),14,1024,fs);    % 14th order model
[P2,f] = pyulear(mtlb(1:512),14,1024,fs);  % 14th order model
plot(f,10*log10(P1),':',f,10*log10(P2)); grid
ylabel('Magnitude (dB)'); xlabel('Frequency (Hz)');
legend('Burg','Yule-Walker AR')
```



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));

[P1,f] = pwelch(xn,hamming(256),128,1024,fs);
[P2,f] = pburg(xn,14,1024,fs);

plot(f,10*log10(P1),':',f,10*log10(P2)); grid
ylabel('PSD Estimates (dB/Hz)');
xlabel('Frequency (Hz)');
legend('Welch','Burg')
```



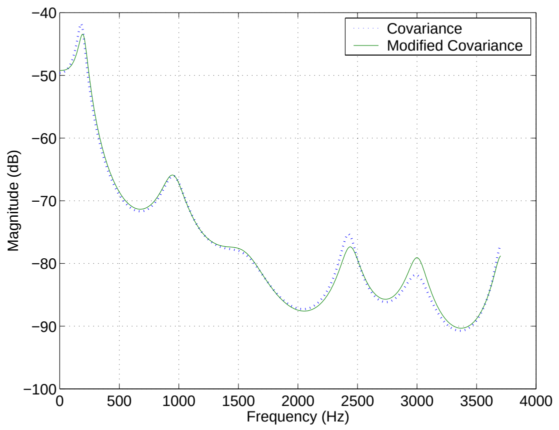
Note that, as the model order for the Burg method is reduced, a frequency shift due to the initial phase of the sinusoids will become apparent.

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 functions pcov and 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
[P1,f] = pcov(mtlb(1:64),14,1024,fs);  % 14th order model
[P2,f] = pmcov(mtlb(1:64),14,1024,fs); % 14th order model
plot(f,10*log10(P1),':',f,10*log10(P2)); grid
ylabel('Magnitude (dB)'); xlabel('Frequency (Hz)');
legend('Covariance','Modified Covariance')
```



MUSIC and Eigenvector Analysis Methods

The pmusic and peig functions provide two related spectral analysis methods:

- pmusic provides the multiple signal classification (MUSIC) method developed by Schmidt
- peig provides the eigenvector (EV) method developed by Johnson

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

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

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

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

$\mathbf{v}$ represents the eigenvectors of the input signal's correlation matrix; $\mathbf{v}_k$ is the k^{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.

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

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.

4

Special Topics

Overview

The Signal Processing Toolbox provides functions that allow you to apply a variety of signal processing techniques. The following sections describe how to use some of these functions:

- “Windows”
- “Parametric Modeling”
- “Resampling”
- “Cepstrum Analysis”
- “FFT-Based Time-Frequency Analysis”
- “Median Filtering”
- “Communications Applications”
- “Deconvolution”
- “Specialized Transforms”
- “Selected Bibliography”

Windows

In both digital filter design and power spectrum 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.

Window	**Function**
Bartlett-Hann window	`barthannwin`
Bartlett window	`bartlett`
Blackman window	`blackman`
Blackman-harris window	`blackmanharris`
Bohman window	`bohmanwin`
Chebyshev window	`chebwin`
Gaussian window	`gausswin`
Hamming window	`hamming`
Hann window	`hann`
Kaiser window	`kaiser`
Nuttall's Blackman-harris window	`nuttallwin`
Rectangular window	`rectwin`
Tapered cosine window	`tukeywin`
Triangular window	`triang`

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);
```

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
```

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.

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, 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.

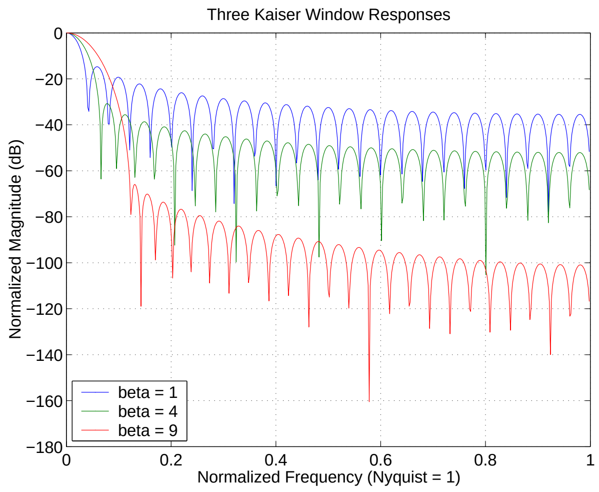
The Blackman window is a popular three-term window, given by A = 0.42, B = 0.5, C = 0.08. The blackman function computes this window.

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

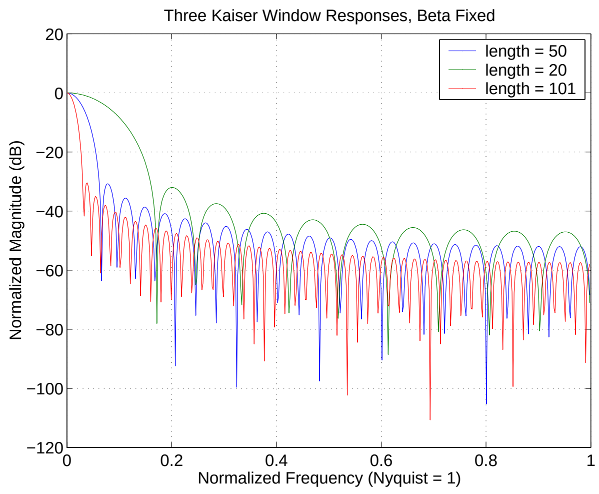
Examples of Kaiser windows with length 50 and various values for the `beta` parameter are

```
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)
title('Three Kaiser Window Responses')
xlabel('Normalized Frequency (Nyquist = 1)')
ylabel('Normalized Magnitude (dB)')
```



As β increases, the sidelobe height decreases and the mainlobe width increases. To see how the sidelobe height stays the same for a fixed β parameter as the length is varied, try

```
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')
title('Three Kaiser Window Responses, Beta Fixed')
xlabel('Normalized Frequency (Nyquist = 1)')
ylabel('Normalized Magnitude (dB)')
```



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 $-\alpha$ dB, the beta parameter is

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

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

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

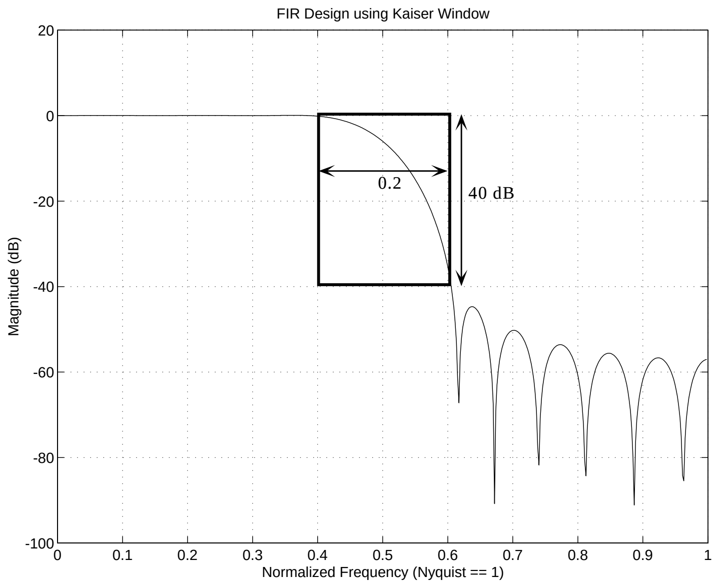
Filters designed using these heuristics will meet the specifications approximately, but you should verify this. To design a lowpass filter with cutoff frequency 0.5π rad/s, transition width 0.2π rad/s, and 40 dB of attenuation in the stopband, try

```
[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.

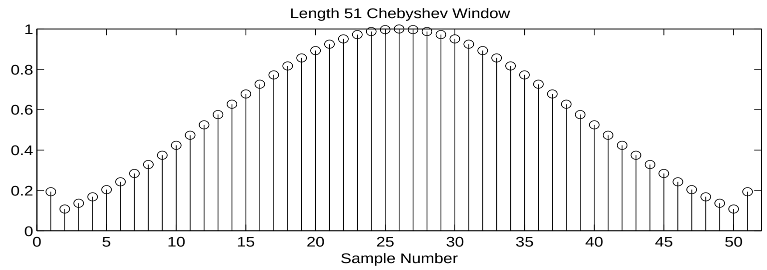
```
[H,f] = freqz(h,1,512,2);
plot(f,20*log10(abs(H))), grid
```


FIR Design using Kaiser Window
Magnitude (dB)
Normalized Frequency (Nyquist == 1)
0.2
40 dB
20
0
-20
-40
-60
-80
-100
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

Chebyshev Window

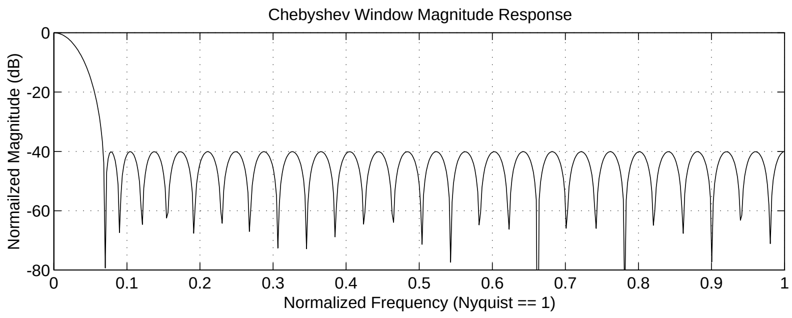
The Chebyshev window minimizes the mainlobe width, given a particular sidelobe height. It is characterized by an equiripple behavior, that is, its sidelobes all have the same height. The `chebwin` function, with length and sidelobe height parameters, computes a Chebyshev window.

```
n = 51;
Rs = 40;               % Sidelobe height in decibels
w = chebwin(n,Rs);
stem(w); title('Length 51 Chebyshev Window');
xlabel('Sample Number');
```



As shown in the plot, the Chebyshev window has large spikes at its outer samples. Plot the frequency response to see the equiripples at -40 dB.

```
[W,f] = freqz(w,1,512,2);
plot(f,20*log10(abs(W)/sum(w))), grid
title('Chebyshev Window Magnitude Response');
xlabel('Normalized Frequency (Nyquist = 1)');
ylabel('Normalized Magnitude (dB)');
```



For a detailed discussion of the characteristics and applications of the various window types, see Oppenheim and Schafer [2], pgs. 444-462, and Parks and Burrus [3], 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.”

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.” `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 nth-order all-pole model of a signal x is

```
a = lpc(x,n)
```

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

```
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 [3] 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
[bb,aa] = invfreqz(h,w,4,4)    % Model: n = 4, m = 4

bb =

    0.0466    0.1863    0.2795    0.1863    0.0466

aa =

    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

```
[bb,aa] = invfreqz(h,w,3,3)   % Find third-order IIR

bb =

    0.0464    0.1785    0.2446    0.1276

aa =

    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

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

by creating a system of linear equations and solving them with MATLAB's \ 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

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

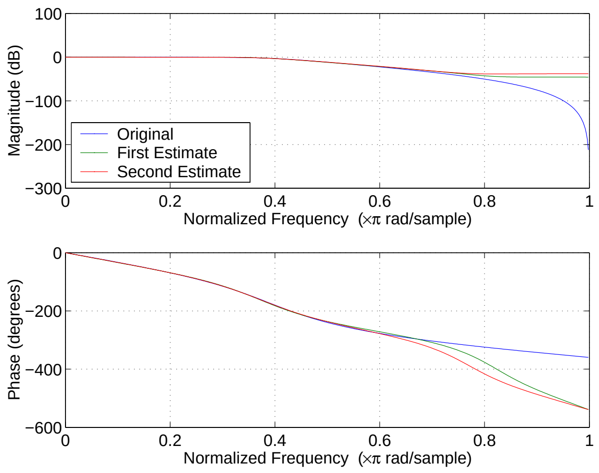
To use this algorithm, specify a parameter for the iteration count after the weight vector parameter

```
wt = ones(size(w));    % Create unity weighting vector
[bbb,aaa] = invfreqz(h,w,3,3,wt,30)  % 30 iterations

bbb =

    0.0464    0.1829    0.2572    0.1549

aaa =

    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.

```
[H1,w] = freqz(b,a);
[H2] = freqz(bb,aa);
[H3] = freqz(bbb,aaa);
s.plot = 'mag';
s.xunits = 'rad/sample';
s.yunits = 'linear';
freqzplot([H1 H2 H3],w);
legend('Original','First Estimate','Second Estimate',3);
grid on
```



To verify the superiority of the fit numerically, type

```
sum(abs(h-freqz(bb,aa,w)).^2)   % Total error, algorithm 1

ans =

    0.0200

sum(abs(h-freqz(bbb,aaa,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). In the next example, the sampling rates are different but the idea is the same.

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*Fs = 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 different sample rate than its original rate. 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)
```

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 Schafer [2] 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.

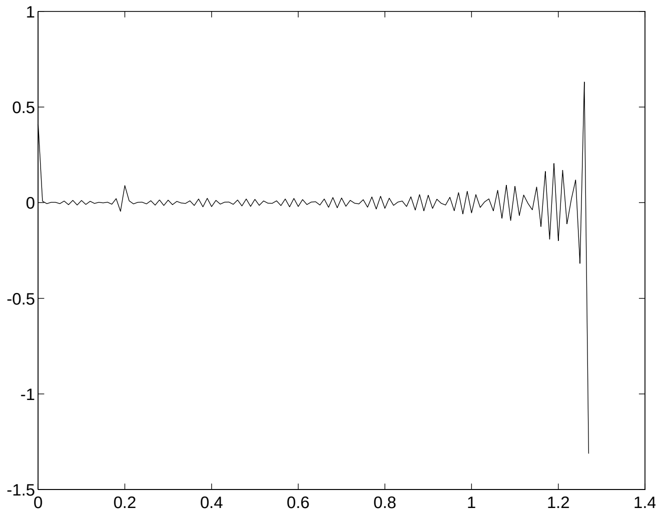
```
t = 0:0.01:1.27;
s1 = sin(2*pi*45*t);
```

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.

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

The toolbox function rceps performs this operation, returning the real cepstrum for a sequence 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 Schafer [2]). The rceps function, however, can reconstruct a minimum-phase version of the original sequence by applying a windowing function in the cepstral domain. To obtain both the real cepstrum and the minimum phase reconstruction for a sequence, use

```
[y,ym] = rceps(x)
```

where y is the real cepstrum and ym is the minimum phase reconstruction of x.

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;
[xh,nd] = cceps(x)

xh =
  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

nd =
     1
```

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

```
icceps(xh,nd)

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

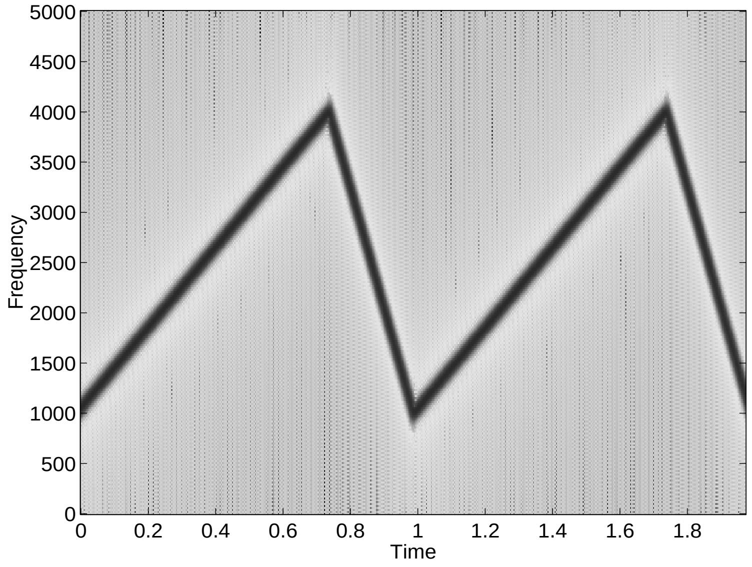
Note With any modification of the complex cepstrum, the original delay term may no longer be valid. Use the `icceps` function with care.

FFT-Based Time-Frequency Analysis

The Signal Processing Toolbox provides a function, `specgram`, that returns the time-dependent Fourier transform for a sequence, or displays this information as a spectrogram. The Toolbox also inclues 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);
specgram(x,512,fs,kaiser(256,5),220)
```



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 [4]. 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 User's Guide* 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**
`amdsb-sc` or `am`	Amplitude modulation, double side-band, suppressed carrier
`amdsb-tc`	Amplitude modulation, double side-band, transmitted carrier

Method	Description
amssb	Amplitude modulation, single side-band
fm	Frequency modulation
pm	Phase modulation
ppm	Pulse position modulation
pwm	Pulse width modulation
qam	Quadrature amplitude modulation

If the input `x` is an array rather than a vector, `modulate` modulates each column of the array.

To obtain the time vector that `modulate` uses to compute the modulated signal, specify a second output parameter.

```
[y,t] = modulate(x,fc,fs,'method',opt)
```

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

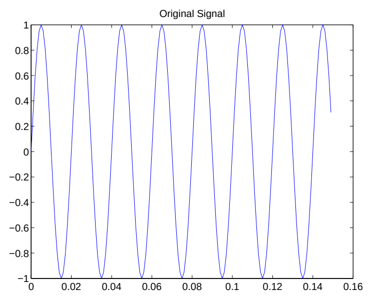
```
t = (0:1/1000:2);
x = sin(2*pi*50*t);
```

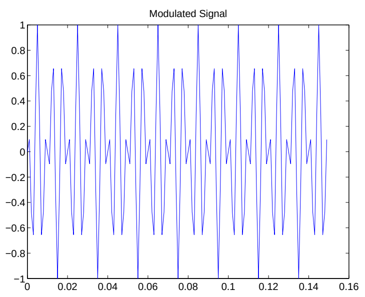
With a carrier frequency of 200 Hz, the modulated and demodulated versions of this signal are

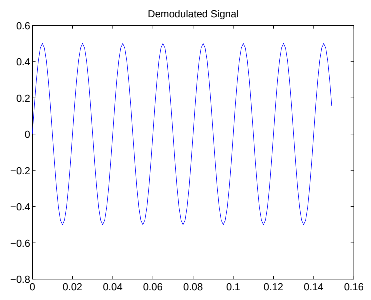
```
y = modulate(x,200,1000,'am');
z = demod(y,200,1000,'am');
```

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 = vco(x,fc,fs)
```

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

To scale the frequency modulation range, use

```
y = vco(x,[Fmin Fmax],fs)
```

In this case, `vco` scales the frequency modulation range so values of `x` on the interval `[-1 1]` map to oscillations of frequency on `[Fmin Fmax]`.

If the input `x` is an array, `vco` produces an array whose columns oscillate according to the columns of `x`.

See “FFT-Based Time-Frequency Analysis” on page 4-29 for an example using the `vco` function.

Deconvolution

Deconvolution, or polynomial division, is the inverse operation of convolution. Deconvolution is useful in recovering the input to a known filter, given the filtered output. This method is very sensitive to noise in the coefficients, however, so use caution in applying it.

The syntax for `deconv` is

```
[q,r] = deconv(b,a)
```

where `b` is the polynomial dividend, `a` is the divisor, `q` is the quotient, and `r` is the remainder.

To try `deconv`, first convolve two simple vectors `a` and `b` (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 (DFT) described in Chapter 1, “Signal Processing Basics,” 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

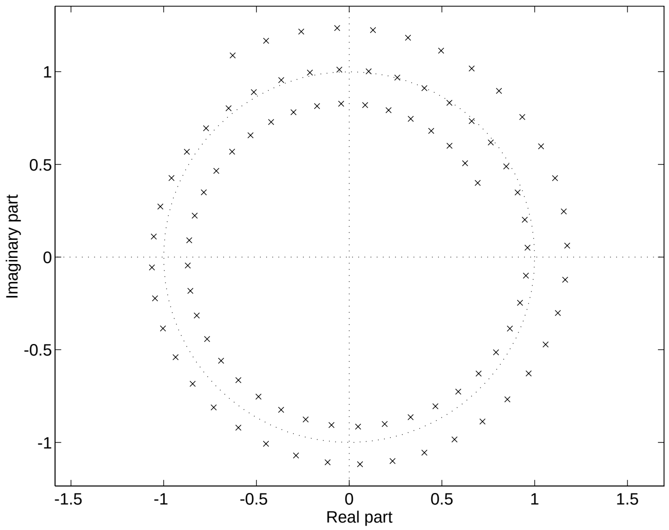
The chirp *z*-transform, or CZT, computes the *z*-transform along spiral contours in the *z*-plane for an input sequence. Unlike the DFT, the CZT is not constrained to operate along the unit circle, but can evaluate the *z*-transform along contours described by

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

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.')
```



`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 = czt(x)
```

`czt` may be faster than the `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

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

where

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

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

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

where

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

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.

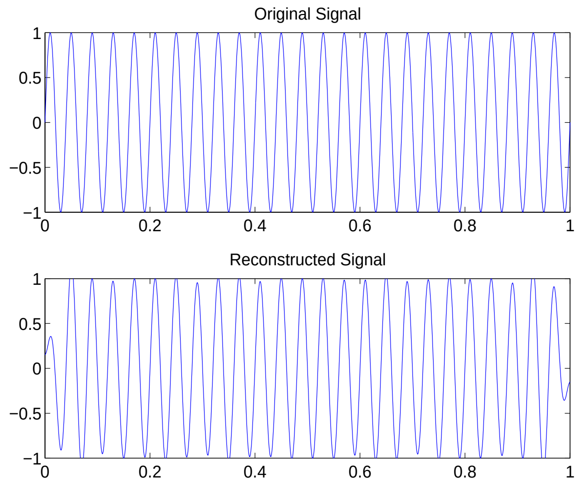
```
t = (0:1/999:1);
x = sin(2*pi*25*t);
```

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

```
norm(x-z)/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

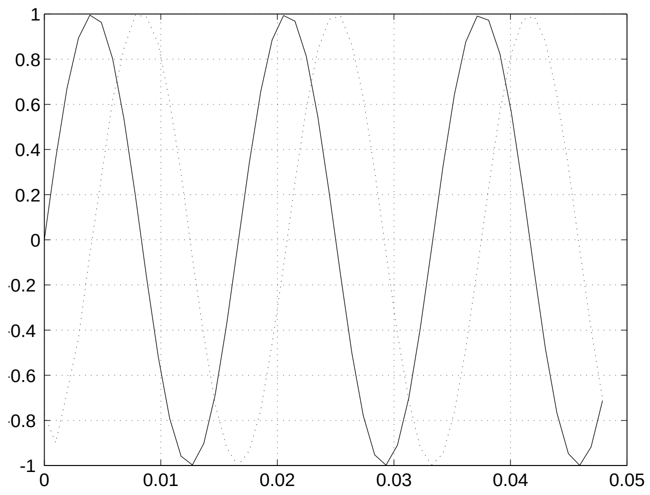
The toolbox function `hilbert` computes the Hilbert transform for a real input sequence `x` and returns a complex result of the same length

```
y = 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. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[3] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987.

[4] Pratt,W.K. *Digital Image Processing*. New York: John Wiley & Sons, 1991.

Filter Design and Analysis Tool

Overview

Note The Filter Design and Analysis Tool (FDATool) requires resolution greater than 640 x 480.

The Filter Design and Analysis Tool (FDATool) is a powerful user interface for designing and analyzing filters. FDATool enables you to quickly design digital FIR or IIR filters by setting filter performance specifications, by importing filters from your MATLAB workspace, or by directly specifying filter coefficients. FDATool also provides tools for analyzing filters, such as magnitude and phase response plots and pole-zero plots. You can use FDATool as a convenient alternative to the command line filter design functions. This chapter contains the following sections, which walk you through a typical filter design session using the FDATool:

- "Opening the Filter Design and Analysis Tool"
- "Choosing a Filter Type"
- "Choosing a Filter Type"
- "Choosing a Filter Design Method"
- "Setting the Filter Design Specifications"
- "Computing the Filter Coefficients"
- "Analyzing the Filter"
- "Converting the Filter Structure"
- "Importing a Filter Design"
- "Exporting a Filter Design"
- "Saving and Opening Filter Design Sessions"

Below is a brief introduction to the FDATool that will give you a better understanding of how it can be used.

Note If you have installed external hardware or targets that interact with FDATool, you may see additional options in FDATool. Refer to the specific hardware or target manual for information on these options.

These toolboxes/targets are integrated with FDATool.
Filter Design Toolbo*x*
Developer's Kit for Texas Instruments™ DSP

Filter Design Methods

The tool gives you access to all of the filter design methods in the Signal Processing Toolbox:

- Butterworth (`butter`)
- Chebyshev Type I (`cheby1`)
- Chebyshev Type II (`cheby2`)
- Elliptic (`ellip`)
- Equiripple (`remez`)
- Least squares (`firls`)
- Windows
  - Bartlett-Hanning (`barthannwin`)
  - Bartlett (`bartlett`)
  - Blackman (`blackman`)
  - Blackman-Harris (`blackmanharris`)
  - Bohman (`bohmanwin`)
  - Chebyshev (`chebwin`)
  - Gaussian (`gausswin`)
  - Hamming (`hamming`)
  - Hann (`hann`)
  - Kaiser (`kaiser`)
  - Nuttall's Blackman-Harris (`nuttallwin`)
  - Rectangular (`rectwin`)

- Tapered cosine (`tukeywin`)
- Triangular (`triang`)

You can implement any of these windows manually using `fir1` or `fir2`.

Additional filter design methods are available to users of the Filter Design Toolbox.

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 filter 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 analyze different filter responses in FDATool or in a separate Filter Visualization Tool (`fvtool`):

- Magnitude response (`freqz`)
- Phase response (`freqz`)
- Pole-zero plots (`zplane`)
- Impulse response (`impz`)
- Group delay (`grpdelay`)
- Step response

You can also display the filter coefficients, export the coefficients to the MATLAB workspace, and create a C header file containing the coefficients.

Filter Design and Analysis Tool Modes

The Filter Design and Analysis Tool operates in two modes:

- *Design mode*. In this mode, you can:

  - Design filters from scratch.
  - Modify existing filters designed by FDATool.
  - Analyze filters.

- *Import mode*. In this mode, you can:
  - Import previously saved filters or filter coefficients that you have stored in the MATLAB workspace.
  - Analyze imported filters.

If you also have the Filter Design Toolbox installed the additional *quantize mode* is available. Use this mode to:

- Quantize double-precision filters that you design in FDATool.
- Quantize double-precision filters that you import into FDATool.
- Analyze quantized filters.

Getting Help

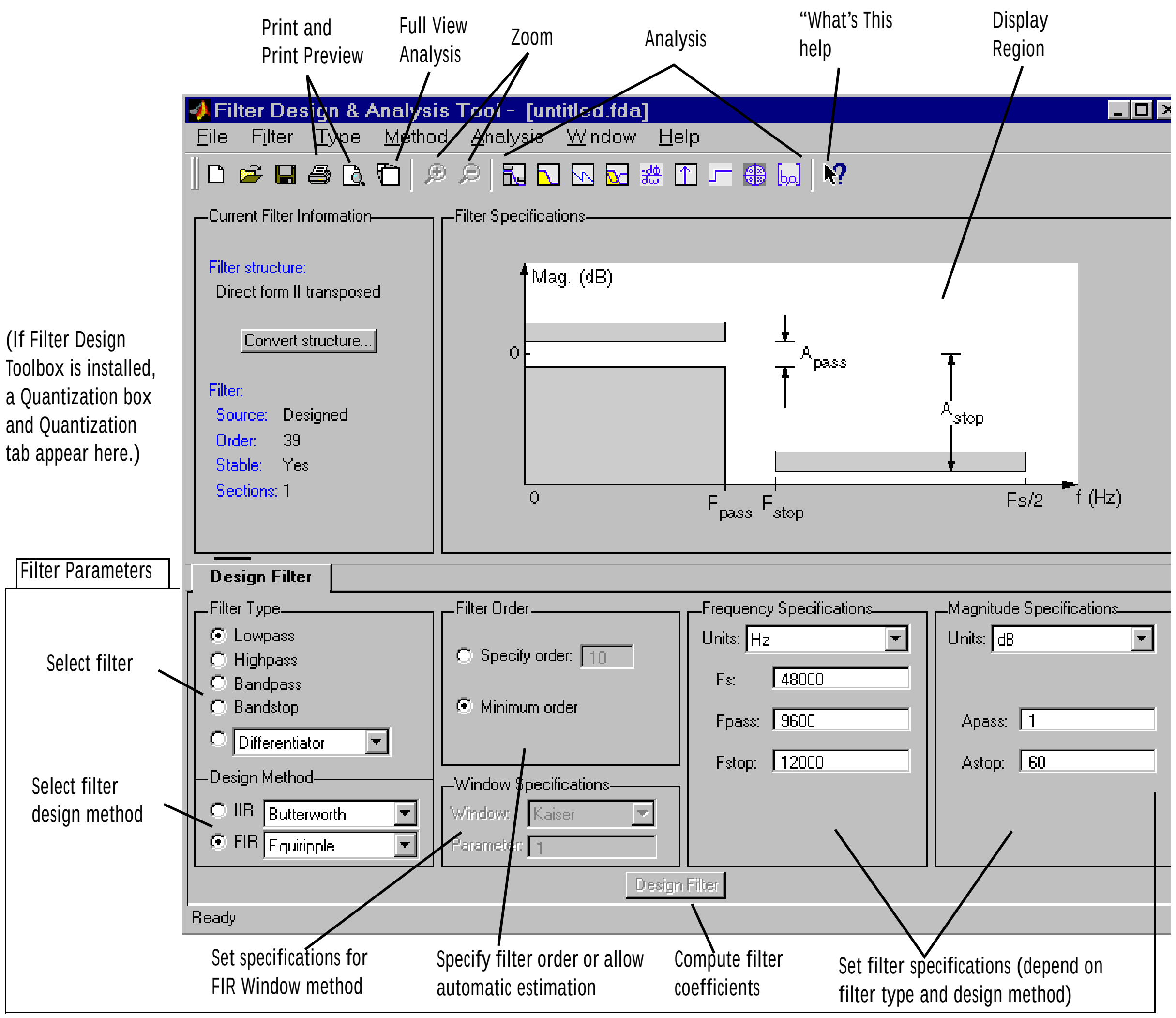
At any time, you can right-click or press the **What's this?** button, ⍰, to get information on the different parts of the GUI.

Opening the Filter Design and Analysis Tool

To open the Filter Design and Analysis Tool, type

```
fdatool
```

The Filter Design and Analysis Tool opens in the default design mode.

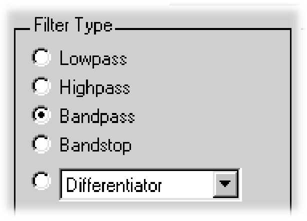


Choosing a Filter Type

You can choose from several filter types:

- Lowpass
- Highpass
- Bandpass
- Bandstop
- Differentiator
- Hilbert transformer
- Multiband
- Arbitrary magnitude
- Arbitrary group delay
- Raised cosine

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



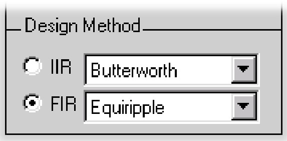
Note Not all filter design methods are available for all filter types. Once you choose your filter type, this may restrict the filter design methods available to you. Filter design methods that are not available for a selected filter type are dimmed in the **Method** menu, and removed from the **Design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

You can also use the **Type** menu to select a filter type.

Choosing a Filter Design Method

You can use the default filter design method for the filter 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.



Butterworth
Equiripple

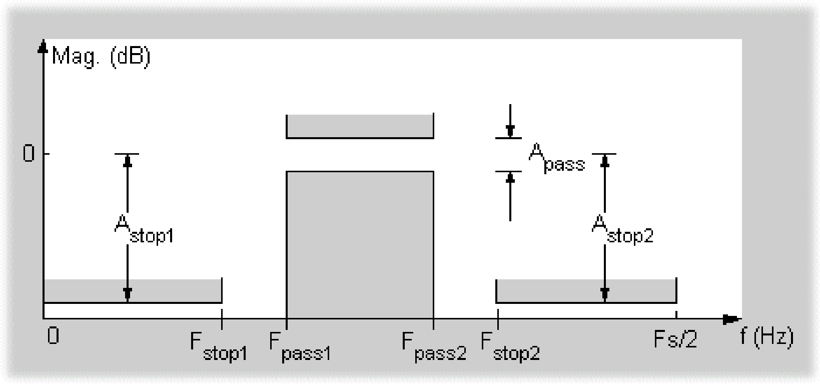
You can also use the **Method** menu to select a filter design method.

Setting the Filter Design Specifications

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

- Bandpass Filter Frequency Specifications
- Bandpass Filter Magnitude Specifications
- Filter Order

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



Bandpass Filter Frequency Specifications

For a bandpass filter, you can set:

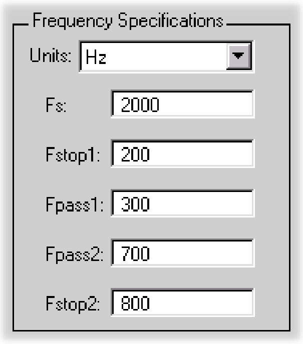
- Units of frequency:
  - Hz
  - kHz
  - MHz
  - Normalized (0 to 1)
- Sampling frequency
- Passband frequencies
- Stopband frequencies

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 (**Fs**) to 2000 Hz.
- Set the end of the first stopband (**Fstop1**) to 200 Hz.
- Set the beginning of the passband (**Fpass1**) to 300 Hz.
- Set the end of the passband (**Fpass2**) to 700 Hz.
- Set the beginning of the second stopband (**Fstop2**) to 800 Hz.



Frequency Specifications

Units:	Hz
Fs:	2000
Fstop1:	200
Fpass1:	300
Fpass2:	700
Fstop2:	800

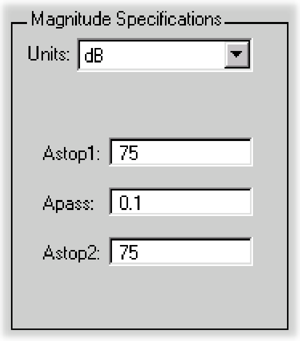
Bandpass Filter Magnitude Specifications

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 the units in **dB** (default).
- Set the passband ripple (**Apass**) to `0.1` dB.
- Set the stopband attenuation for both stopbands (**Astop1**, **Astop2**) to `75` dB.



Magnitude Specifications

Units: dB

Astop1: 75

Apass: 0.1

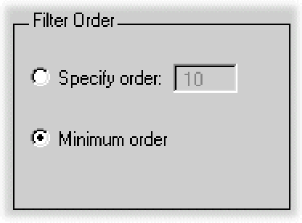
Astop2: 75

Filter Order

You have two mutually exclusive options for determining the filter order when you design an equiripple filter

- **Minimum order**: The filter design method determines the minimum order filter.
- **Specify order**: You enter the filter order in a text box.

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



Filter Order

Specify order: 10

[✓ Minimum order]

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

Computing the Filter Coefficients

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

Notice that the **Design Filter** button is dimmed 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

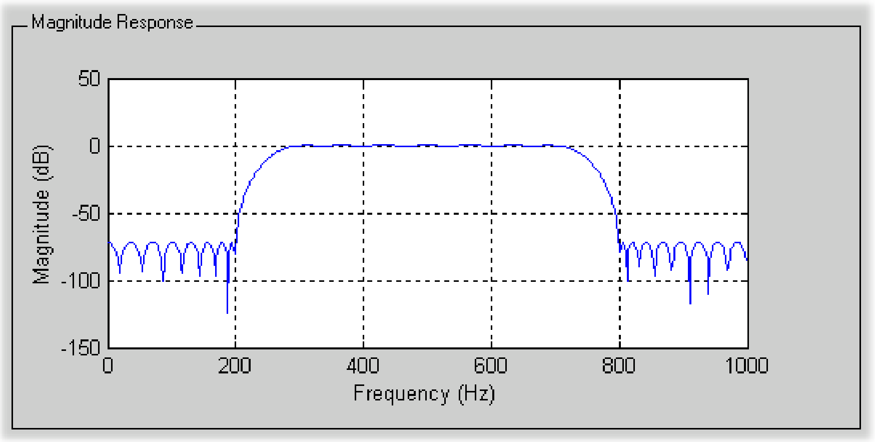
Once you've designed the filter, you can view the following filter response characteristics in the display region or in a separate window:

- Magnitude response
- Phase response
- Overlaid magnitude and phase responses
- Group delay
- Impulse response
- Step response
- Pole-zero plot

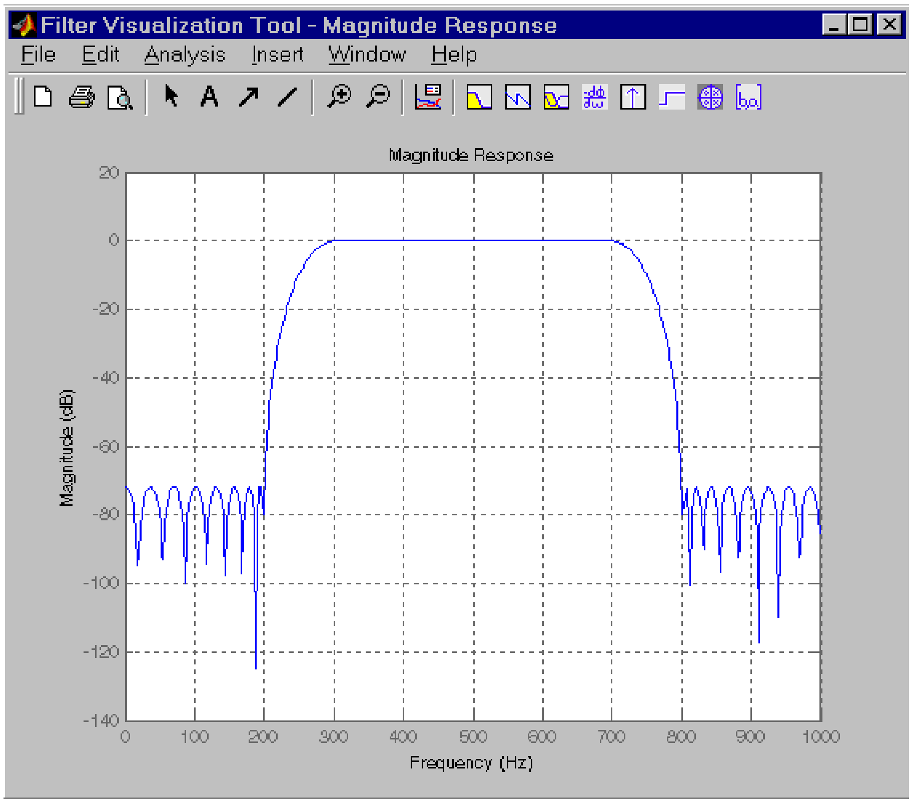
You can also display the filter coefficients in this region.

You can access the analysis methods as items in the **Analysis** menu, or by using the toolbar buttons.

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



To display the filter response characteristics in a separate window, select **Full View Analysis** from the **Analysis** menu. The Filter Visualization Tool (`fvtool`) opens. You can use this tool to annotate your design, view other filter analyses, and print your filter response.



You can click on the response to add a data marker that displays information about the particular point on the response. Right-clicking displays a menu where you can adjust the appearance of the data markers or delete them.

Converting the Filter Structure

You can use the **Convert structure** button 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`
- `State-space`
- `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 MA (min. phase)`
- Maximum phase FIR filters can be converted to `Lattice MA (max. 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.

When selected, the **Use second-order sections** check box instructs the tool to store the converted filter structure as a collection of second-order sections rather than as a monolithic higher-order structure. The subordinate **Scale** menu lets you select from the following options.

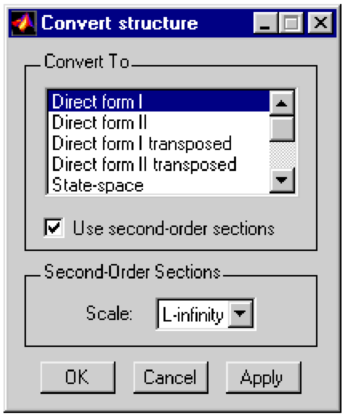
- `None` (default)

- `L-2` (L^2 norm)
- `L-infinity` (L^∞ norm)

The ordering of the second-order sections is optimized for the selected scaling option. (The `tf2sos` function that performs the conversion is called with option `'down'` for L^2 norm scaling, and with option `'up'` for L^∞ norm scaling.)

For example:

- Press the **Convert structure** button to open the **Convert structure** dialog box.
- Select `Direct form I` in the list of filter structures.
- Select the **Use second-order sections** check box.
- Select `L-infinity` from the **Scale** menu for L^∞ norm scaling.



Convert structure

Convert To

Direct form I
Direct form II
Direct form I transposed
Direct form II transposed
State-space

☑ Use second-order sections

Second-Order Sections

Scale: L-infinity

OK Cancel Apply

Importing a Filter Design

The **Import Filter** region allows you to import a filter by specifying the filter coefficients, either by entering them explicitly or by referring to variables in the MATLAB workspace. You can access this region by selecting **Import Filter** from the **Filter** menu, or by pressing **Ctrl+I**.

The imported filter can be in any of the representations listed in the **Filter Structure** menu, and described in “Filter Structures”.

Select the frequency units from among the following options in the **Units** menu, and specify the value of the sampling frequency in the **Fs** field.

To import the filter, press the **Import Filter** button. The **Display Region** is automatically updated when the new filter has been imported.

Filter Structures

- Direct Form filters
  - `Direct form I`
  - `Direct form II`
  - `Direct form I transposed`
  - `Direct form II transposed`
  - `Direct form II (Second-order sections)`
- `State-space`
- Lattice filters
  - `Lattica allpass`
  - `Lattice MA min. phase`
  - `Lattice MA max. phase`
  - `Lattice ARMA`
- `Quantized filter (Qfilt object)` — available only when Filter Design Toolbox is installed

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:

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

- The **Numerator** field specifies a variable name or value for the numerator coefficient vector `b`, which contains `m+1` coefficients in descending powers of z.
- The **Denominator** field specifies a variable name or value for the denominator coefficient vector `a`, which contains `n+1` coefficients in descending powers of z. 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`, `remez`, `butter`, `yulewalk`). See “Transfer Function” on page 1-33 for more information.

Direct Form II (Second-Order Sections)

For `Direct form II (Second-order sections)`, specify the filter by 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 1-38 for more information.

State-Space

For `State-Space`, specify the filter by its state-space representation:

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

The **A**, **B**, **C**, and **D** fields each specify a variable name or a value for the matrices in this system.

Filters in state-space form can be produced by functions such as `tf2ss` and `zp2ss`. See “State-Space” on page 1-35 for more information.

Lattice

For `Lattice allpass`, `Lattice MA`, 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)`, were `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)`, were `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)`, were `N` is the filter order.

Filters in lattice form can be produced by `tf2latc`. See "Lattice Structure" on page 1-38 for more information.

Quantized Filter (Qfilt Object)

For `Quantized filter`, specify the filter as a Qfilt object. See `qfilt` in the Filter Design Toolbox for more information.

Exporting a Filter Design

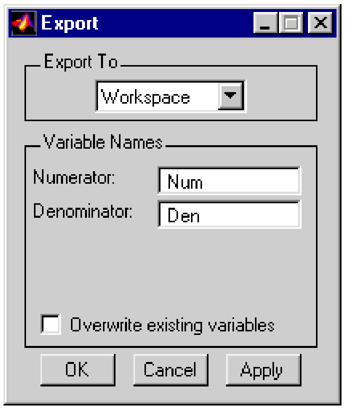
You can save your filter design by:

- Exporting Filter Coefficients to the Workspace
- Exporting Filter Coefficients to a Text File
- Exporting the filter coefficients to a MAT-file (see Saving and Opening Filter Design Sessions)
- Creating a C header file containing the coefficients

Exporting Filter Coefficients to the Workspace

To save filter coefficients as variables in the MATLAB workspace

1. Select **Export** from the **File** menu. The **Export** dialog box appears.
2. Select **Workspace** from the **Export To** menu.
3. Assign variable names using the text boxes in the **Variable Names** region. If you have variables with the same names in your workspace and you want to overwrite them, check the **Overwrite existing variables** box.
4. Press the **OK** button.


Export

Export To: Workspace

Variable Names

Numerator: Num

Denominator: Den

Overwrite existing variables

Exporting Filter Coefficients to a Text File

To save filter coefficients to a text file

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

2 Select **Text-file** from the **Export To** menu.

3 Press the **OK** button. The **Export Filter Coefficients to a Text-file** dialog box appears.

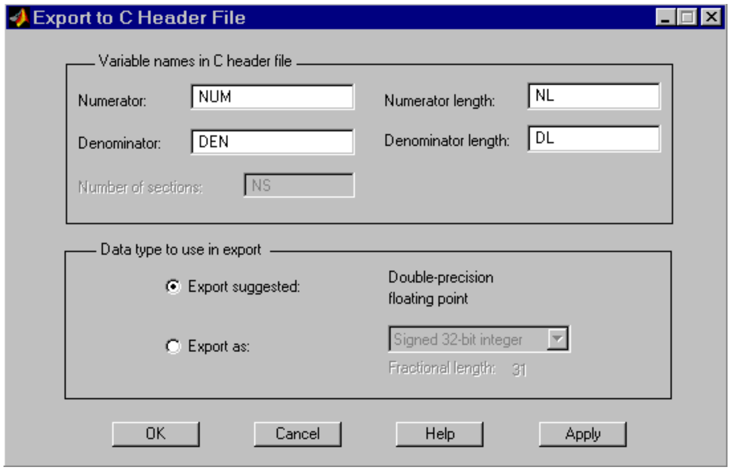
4 Choose a filename and press **Save**.

The coefficients are saved in the text file that you specified, and the MATLAB Editor opens to display the file.

Exporting to 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 **Export to C Header File** from the **File** menu. The **Export to C Header File** dialog box appears.



Export to C Header File

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

(● selected) Export suggested: Double-precision floating point

Export as: Signed 32-bit integer

Fractional length: 31

2 Enter the variable names to be used in the C header file. The particular filter structure determines the variables that are created in the file.

Filter Structure	Variable Parameter
Direct Form I 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 the **Export Suggested** button to use the suggested data type or select **Export As** and select the desired data type from the pulldown and select the desired **Fractional length**.

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.

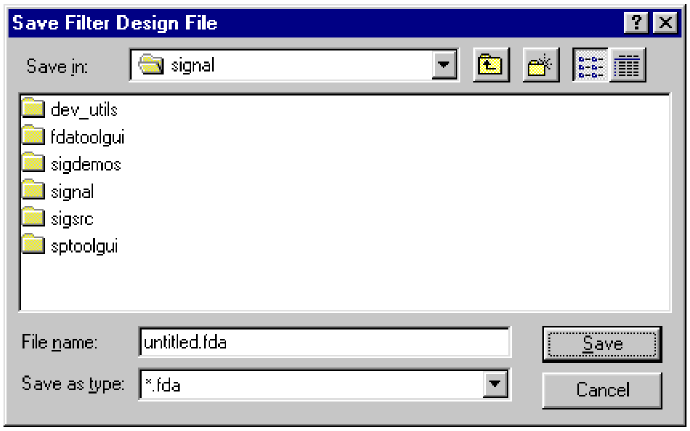
---

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.

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, . 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

Overview

SPTool is a graphical user interface (GUI) for analyzing and manipulating digital signals, filters, and spectra. The first few sections of this chapter give an overview of SPTool and its associated GUIs

- "SPTool: An Interactive Signal Processing Environment"
- "Opening SPTool"
- "Overview of the Signal Browser: Signal Analysis"
- "Overview of the Filter Designer: Filter Design"
- "Overview of the Filter Viewer: Filter Analysis"
- "Overview of the Spectrum Viewer: Spectral Analysis"
- "Using SPTool: Filtering and Analysis of Noise"

The rest of this chapter illustrates in detail how to use the GUI-based interactive tools by walking through some examples

- "Using SPTool: Filtering and Analysis of Noise"
- "Importing a Signal into SPTool"
- "Designing a Filter"
- "Applying a Filter to a Signal"
- "Analyzing Signals: Opening the Signal Browser"
- "Spectral Analysis in the Spectrum Viewer"
- "Exporting Signals, Filters, and Spectra"
- "Accessing Filter Parameters"
- "Redesigning a Filter Using the Magnitude Plot"
- "Accessing Parameters in a Saved Spectrum"
- "Importing Filters and Spectra into SPTool"
- "Loading Variables from the Disk"
- "Selecting Signals, Filters, and Spectra in SPTool"
- "Editing Signals, Filters, or Spectra in SPTool"
- "Designing a Filter with the Pole/Zero Editor"
- "Making Signal Measurements: Using Markers"

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 Designer* is 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 the Filter Designer. Additionally, you can design a filter by using the Pole/Zero Editor to graphically place poles and zeros on the *z*-plane.
- The *Filter Viewer* is for analyzing filter characteristics. See Overview of the Filter Viewer: Filter Analysis.
- 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 Overview of the Spectrum Viewer: Spectral Analysis.

SPTool Data Structures

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

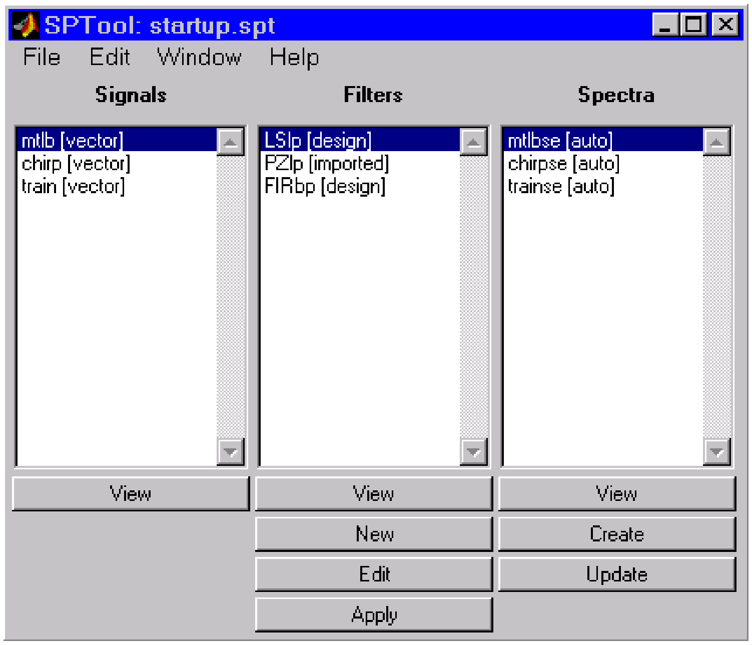
You can bring signals, filters, or spectra from the MATLAB workspace into the SPTool workspace using the **Import** item under the **File** menu. Signals, filters, or spectra that you create in (or import into) the SPTool workspace exist as MATLAB structures. See the MATLAB documentation for more information 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. You can specify your own preferences for what signals, filters, and spectra you want to see when SPTool opens. See “Designing a Filter with the Pole/Zero Editor” on page 6-46 for more details.

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

- Signal Browser
- Filter Viewer
- Spectrum Viewer

You can access the Filter Designer GUI by pressing the **New** button to create a new filter or the **Edit** button to edit a selected filter. The **Apply** button applies a selected filter to a selected signal.

The **Create** button opens the Spectrum Viewer and creates the power spectral density of the selected signal. The **Update** button opens the Spectrum Viewer for the selected spectrum.

Getting Help

Use the **What's this?** button to find out more information about a particular region of the GUI.

Context-Sensitive Help: The What's This? Button

To find information on a particular region of the Signal Browser, Filter Designer, Filter Viewer, or Spectrum Viewer

1 Press the **What's this?** button, ?.

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.

Overview of the Signal Browser: Signal Analysis

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

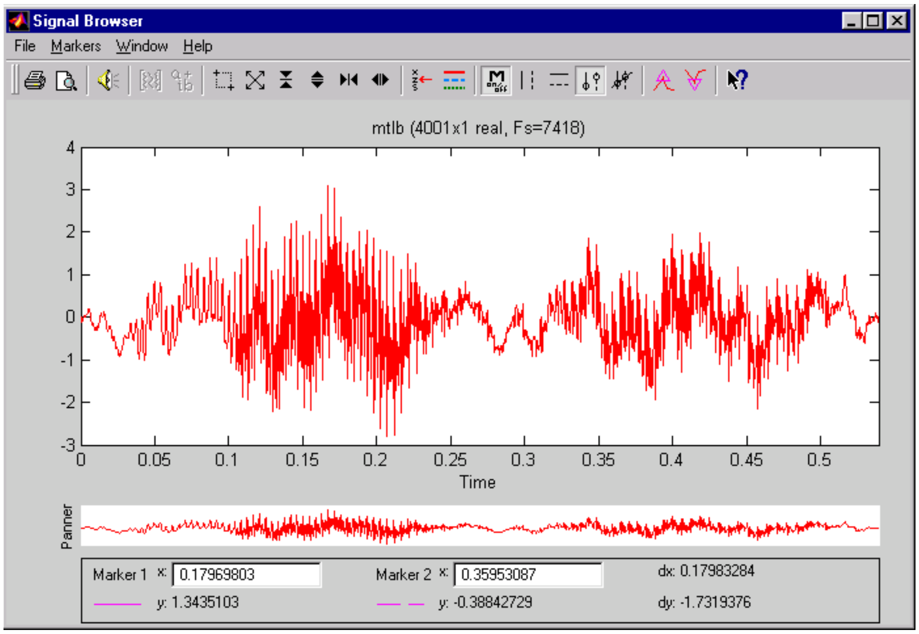
Using the Signal Browser you can

- 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. Selecting one or more signals in the **Signals** list in SPTool
2. Pressing the **View** button under the **Signals** list



Marker 1 x: 0.17969803 y: 1.3435103
Marker 2 x: 0.35953087 y: -0.38842729
dx: 0.17983284
dy: -1.7319376

The Signal Browser has the following components

• A toolbar with buttons for convenient access to frequently used functions

🖨 🔍	Print and print preview
🔈	Play an audio signal
⋮ ab	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
▣	Turn on the What's This help

- 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

Overview of the Filter Designer: Filter Design

The Filter Designer 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.

Note You can also use the Filter Design and Analysis Tool (FDATool) described in Chapter 5, “Filter Design and Analysis Tool,” for filter design and analysis.

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.

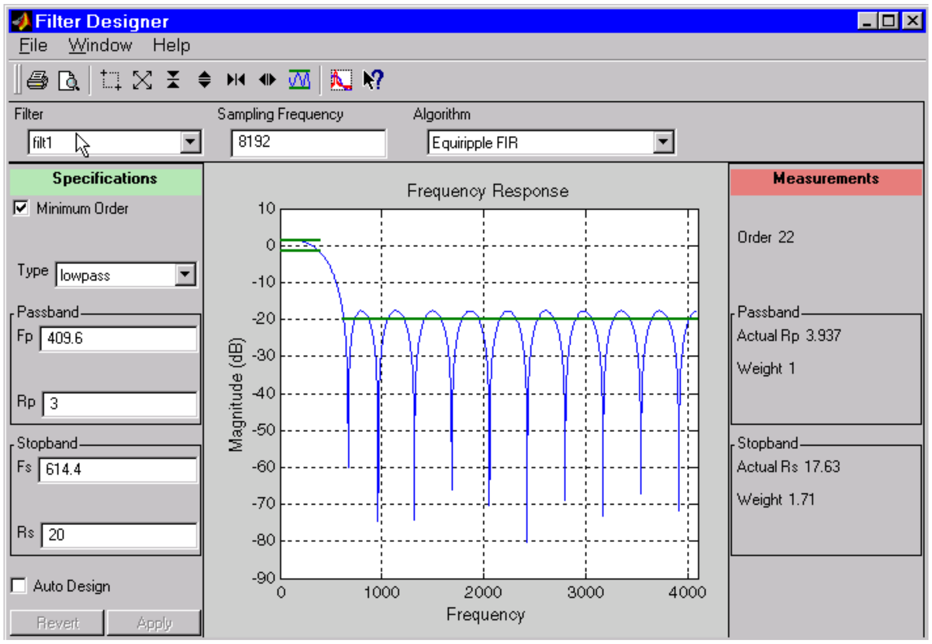
Spectral Overlay Feature

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

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



Filter: filt1
Sampling Frequency: 8192
Algorithm: Equiripple FIR
Type: lowpass
Passband
Fp: 409.6
Rp: 3
Stopband
Fs: 614.4
Rs: 20

Order 22
Passband
Actual Rp 3.937
Weight 1
Stopband
Actual Rs 17.63
Weight 1.71

The Filter Designer has the following components

- A toolbar with the following buttons

🖨 🔍	Print and print preview
⬚ ⤡ ⇕ ⇳ ⏮ ⇔	Zoom in and out
⩍	Passband view
📉	Overlay spectrum
❓	Turn on the What's This help

- A pulldown **Filter** menu for selecting a filter from the list in SPTool
- A **Sampling Frequency** text box
- A pulldown **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

Overview of the Filter Viewer: Filter Analysis

You can use the Filter Viewer to analyze the following response characteristics of selected filters

- Magnitude response
- Phase response
- Impulse response
- Step response
- Group delay
- Pole and zero locations

The Filter Viewer can display up to six different response characteristics plots for a selected filter at any time. The Filter Viewer provides features for

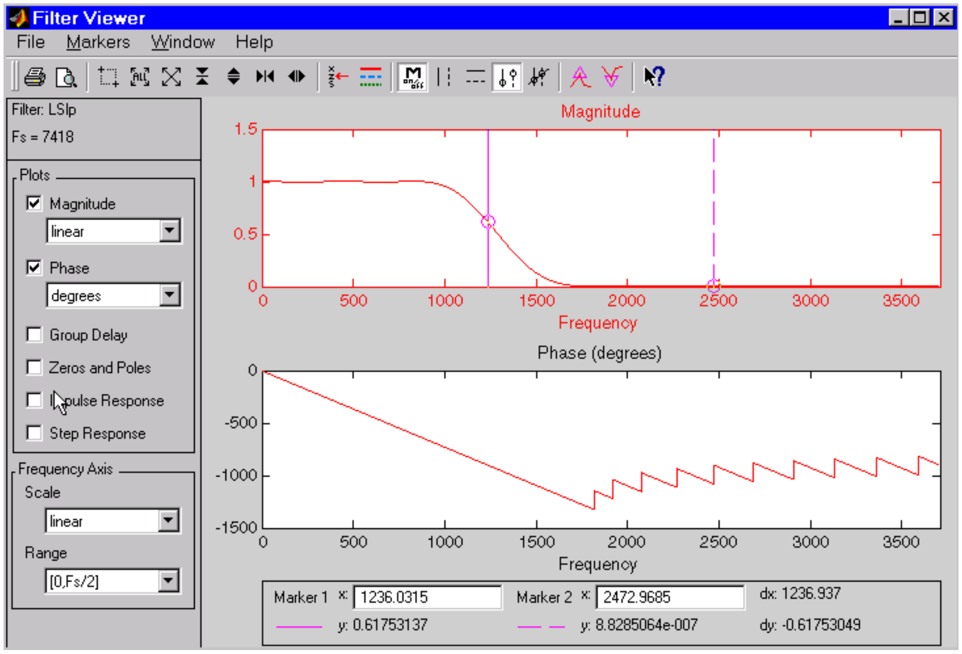
- Zooming
- Measuring filter responses
- Overlaying filter responses
- Modifying display parameters such as frequency ranges or magnitude units

Opening the Filter Viewer

You can open the Filter Viewer from SPTool by

1. Selecting one or more filters in the **Filters** list in SPTool
2. Pressing the **View** button under the **Filters** list

When you first open the Filter Viewer, it displays the default plot configuration for the selected filter(s).



Filter: LSlp
Fs = 7418

Plots
- [✓] Magnitude — linear
- [✓] Phase — degrees
- [ ] Group Delay
- [ ] Zeros and Poles
- [ ] Impulse Response
- [ ] Step Response

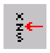
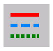
Frequency Axis
Scale: linear
Range: [0,Fs/2]

Marker 1 x: 1236.0315 y: 0.61753137
Marker 2 x: 2472.9685 y: 8.8285064e-007
dx: 1236.937 dy: -0.61753049

The filter's magnitude and phase plots are displayed by default. In addition, the frequency is displayed with a linear scale on the interval `[0,Fs/2]`.

The Filter Viewer has the following components

- A toolbar with buttons for convenient access to frequently used functions

	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

- A filter identification region that displays the filter name and sampling frequency for the currently selected filter(s)
- A **Plots** region for selecting the response plots you want to display in the display area, and for specifying some frequency response display characteristics
- A **Frequency Axis** region for specifying frequency scaling in the display area
- A display area for analyzing one or more frequency response plots for the selected filter(s)
- A Marker measurements area

Overview of the Spectrum Viewer: Spectral Analysis

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.

Using the Spectrum Viewer you can

- 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. Press the **Create** button in the **Spectra** list.
3. Press the **Apply** button in the Spectrum Viewer.

To open the Spectrum Viewer with a PSD estimate already listed in SPTool

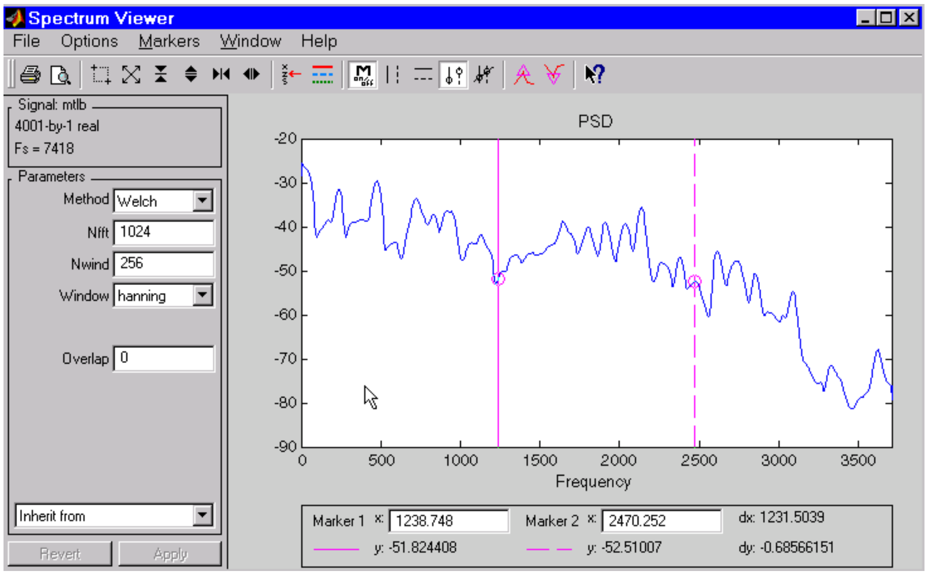
1. Select a PSD estimate from the **Spectra** list box in SPTool.
2. Press the **View** button in the Spectra list.

For example

1 Select `mtlb` in the default **Signals** list in SPTool.

2 Press the **Create** button in SPTool to open the Spectrum Viewer.

3 Press the **Apply** button in the Spectrum Viewer to plot the spectrum.



Signal: mtlb
4001-by-1 real
Fs = 7418

Parameters
Method: Welch
Nfft: 1024
Nwind: 256
Window: hanning
Overlap: 0
Inherit from

Marker 1 x: 1238.748 y: -51.824408
Marker 2 x: 2470.252 y: -52.51007
dx: 1231.5039
dy: -0.68566151

The Spectrum Viewer has the following components

- A toolbar with buttons for convenient access to frequently used functions

	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

- 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

Using SPTool: 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 the Filter Designer

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

6 Exporting the filter design to the MATLAB workspace as a structure

7 Accessing the filter coefficients from the exported MATLAB structure

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

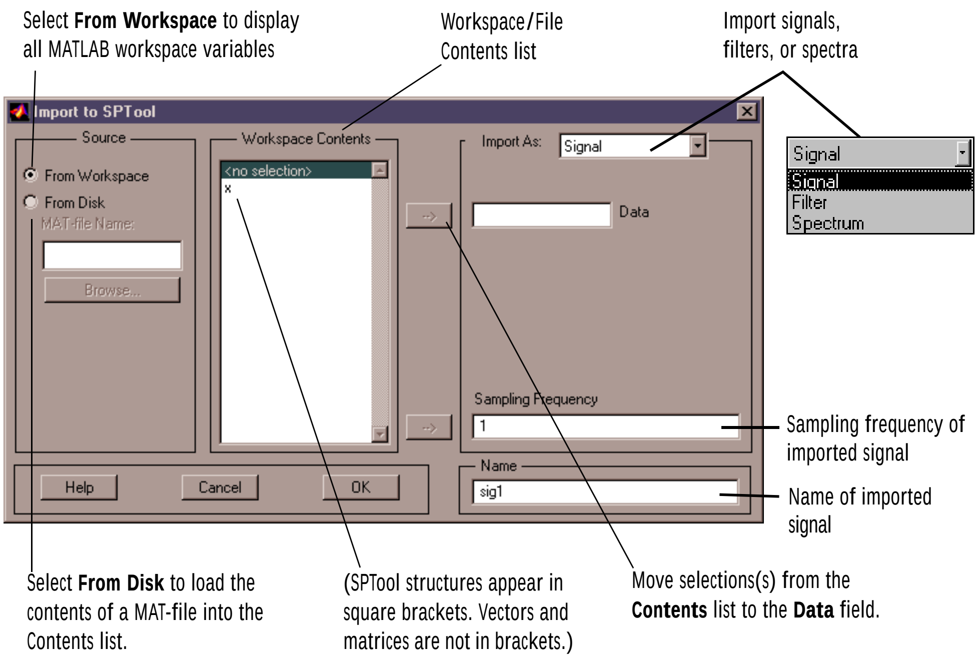
```
randn('state',0);
x = randn(5000,1);
```

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.


Select **From Workspace** to display all MATLAB workspace variables
Workspace/File Contents list
Import signals, filters, or spectra
Import to SPTool
Source
From Workspace
From Disk
MAT-file Name:
Browse...
Workspace Contents
<no selection>
x
Import As:
Signal
Data
Sampling Frequency
1
Name
sig1
Help
Cancel
OK
Signal
Signal
Filter
Spectrum
Sampling frequency of imported signal
Name of imported signal
Select **From Disk** to load the contents of a MAT-file into the Contents list.
(SPTool structures appear in square brackets. Vectors and matrices are not in brackets.)
Move selections(s) from the **Contents** list to the **Data** field.

Notice that 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 Make sure that **Signal** is selected in the **Import As** pull-down menu.

b Select the signal variable `x` in the **Workspace Contents** list.

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 Press **OK**.

At this point, the signal `noise[vector]` 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 into SPTool" on page 6-39 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 6-43 for more information.

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

Designing a Filter

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

In this example

1. Open a default filter in the Filter Designer.
2. Specify an equiripple bandpass FIR filter.

Opening the Filter Designer

To open the Filter Designer, press the **New** button in SPTool. This opens the Filter Designer 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 your filter in the Filter Designer to meet these specifications

1. Change the filter sampling frequency to 5000 by entering this value in the **Sampling Frequency** text box.

2. Select `Equiripple FIR` from the **Algorithm** list.

3. Select `bandpass` from the **Type** list.

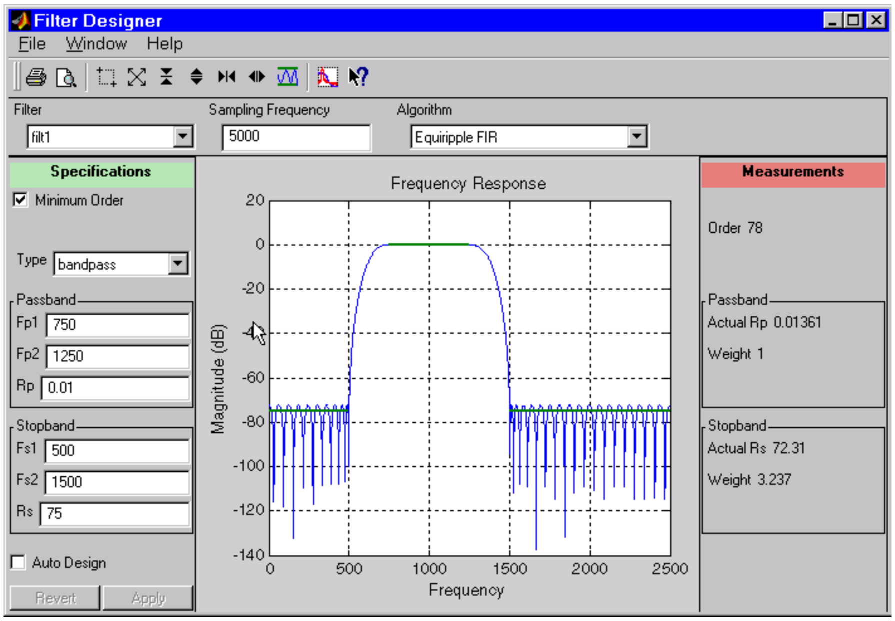
4. Set the passband edge frequencies by entering 750 for **Fp1** and 1250 for **Fp2**.

5. Set the stopband edge frequencies by entering 500 for **Fs1** and 1500 for **Fs2**.

6. Type `0.01` into the **Rp** field and `75` into the **Rs** field.

   Rp sets the maximum passband ripple and **Rs** sets the stopband attenuation for the filter.

7. Press the **Apply** button to design the new filter.When the new filter is designed, the magnitude response of the filter is displayed with a solid line in the display region.



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

Note You can use the solid line in the plot to modify your filter design. See “Redesigning a Filter Using the Magnitude Plot” on page 6-49 for more information.

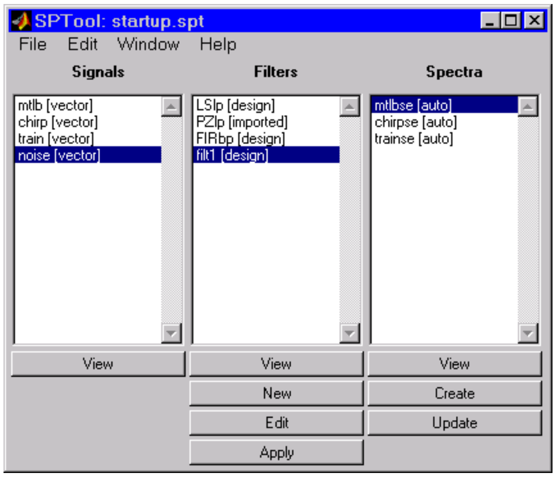
Applying a Filter to a Signal

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`

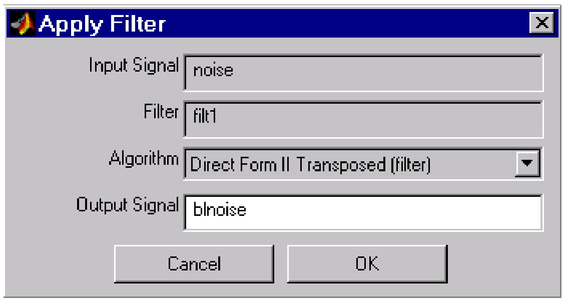
1 Select **SPTool** from the **Window** menu in the Filter Designer.

2 Select the signal `noise[vector]` from the **Signals** list and select the filter (named `filt1[design]`) from the **Filters** list, as shown below.



3 Press **Apply** to apply the filter `filt1` to the signal `noise`.

The **Apply Filter** dialog box is displayed.



Input Signal: noise
Filter: filt1
Algorithm: Direct Form II Transposed (filter)
Output Signal: blnoise

4 Keep the default filter structure selected in the **Algorithm** list.

5 Name the new signal by typing `blnoise` in the **Output Signal** field in this dialog box.

6 Press **OK** to close the **Apply Filter** dialog box.

   The filter is applied to the selected signal and the filtered signal `blnoise[vector]` is listed in the **Signals** list in SPTool.

Analyzing Signals: Opening the Signal Browser

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`

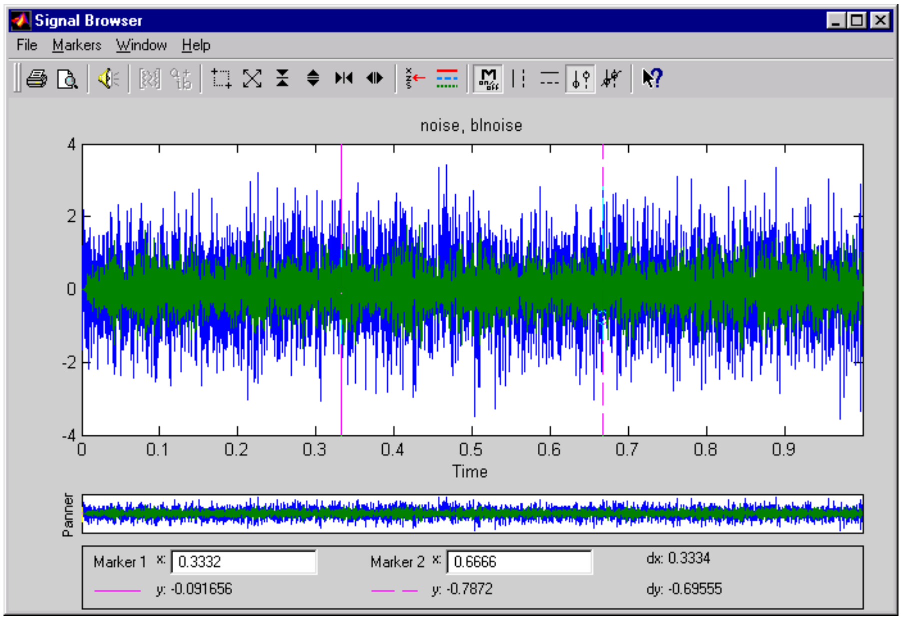
1 **Shift**+click on the `noise` and `blnoise` signals in the **Signals** list of SPTool to select both signals.

2 Press the **View** button 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, ☰.



Marker 1 x: 0.3332 y: -0.091656
Marker 2 x: 0.6666 y: -0.7872
dx: 0.3334
dy: -0.69555

Playing a Signal

When you press the **Play** button 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** Press the **Play** button.

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** Press the **Play** button again.

Printing a Signal

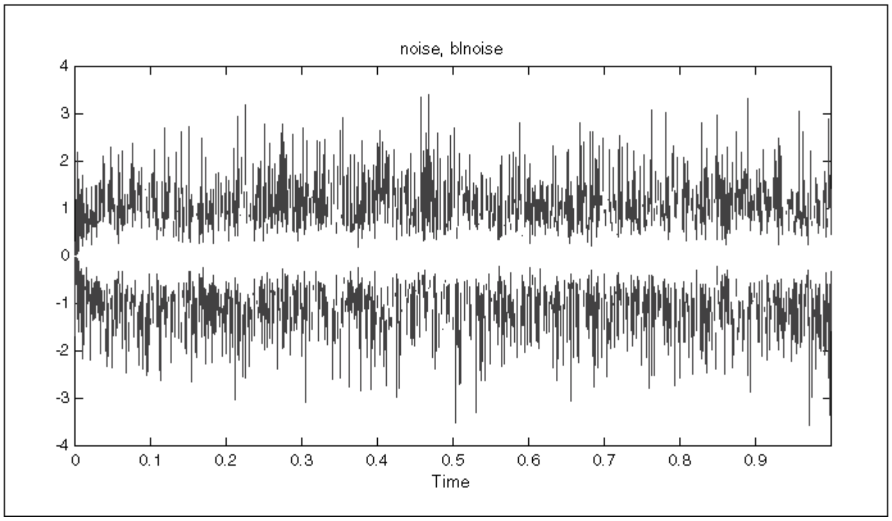
You can print from the Signal Browser using the **Print** button, .

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, press the **Print** button in the Signal Browser toolbar.



Spectral Analysis in the Spectrum Viewer

You can analyze the frequency content of a signal using the Spectrum Viewer, which estimates and displays a signal's power spectral density.

For example, to analyze and compare the spectra of `noise` and `blnoise`

1 Create a power spectral density (PSD) object, `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 Press **Create** in the **Spectra** list.

The Spectrum Viewer is activated, and a PSD object (`spect1`) corresponding to the `noise` signal is created in the **Spectra** list. The PSD is not computed or displayed yet.

4 Press **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`.

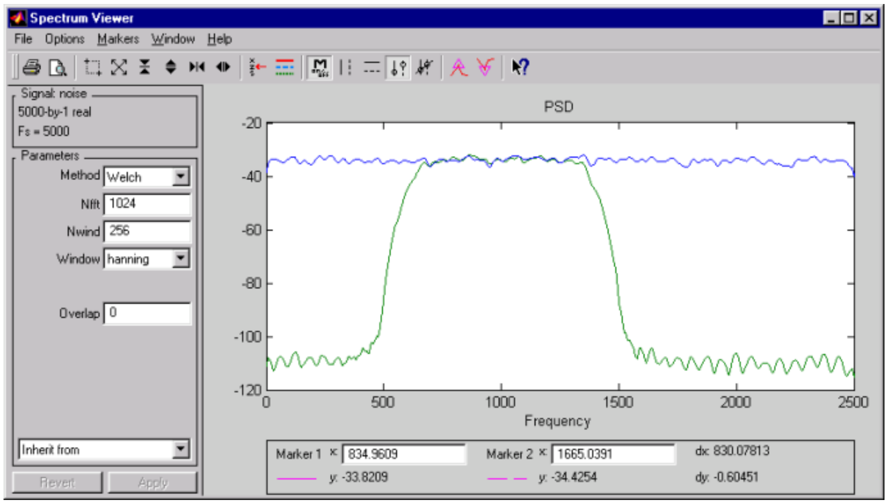
The PSD estimate `spect2` is flat between 750 and 1250 Hz and has 75 dB less power in the stopband regions of `filt1`.

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 Press **View** in the **Spectra** list to reactivate the Spectrum Viewer and display both spectra together.



Signal: noise
5000-by-1 real
Fs = 5000

Parameters
Method Welch
Nfft 1024
Nwind 256
Window hanning
Overlap 0
Inherit from

Marker 1 x 834.9609 y: -33.8209
Marker 2 x 1665.0391 y: -34.4254
dx: 830.07813
dy: -0.60451

Printing the Spectra

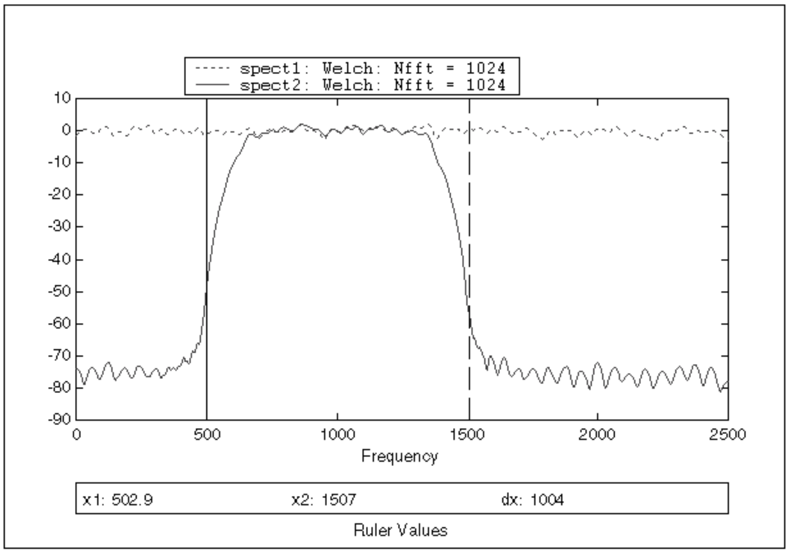
Before printing the two spectra together, use the color and line style selection button, , to differentiate the two plots by line style, rather than by color.

To print both spectra

1 Press the **Print Preview** button, , 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 Press the **Print** button in the **Spectrum Viewer Print Preview** window.


----- spect1: Welch: Nfft = 1024
—— spect2: Welch: Nfft = 1024
10
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
0
500
1000
1500
2000
2500
Frequency
x1: 502.9
x2: 1507
dx: 1004
Ruler Values

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.

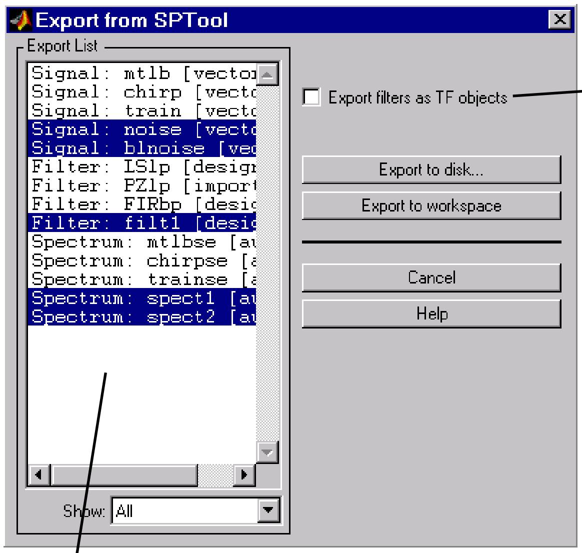
Opening the Export Dialog Box

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.



```
Signal: mtlb [vecto
Signal: chirp [vect
Signal: train [vect
Signal: noise [vect
Signal: blnoise [ve
Filter: LSlp [desig
Filter: PZlp [impor
Filter: FIRbp [desi
Filter: filt1 [desi
Spectrum: mtlbse [a
Spectrum: chirpse [
Spectrum: trainse [
Spectrum: spect1 [a
Spectrum: spect2 [a
```

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 Press the **Export to Workspace** button.

Accessing Filter Parameters

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

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

where

- b is a vector containing the coefficients from the `tf.num` field.
- a is a vector containing the coefficients from the `tf.den` field.
- m is the numerator order.
- n is the denominator order.

You can change the filter representation from the default transfer function to another form by using the `tf2ss` or `tf2zp` functions.

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.

Table 6-1: Filter Specifications currentModule Field Values

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.

Table 6-2: SPTool Structure Specifications for Filters

Parameter	Description
`Beta`	Kaiser window β 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.

Table 6-2: SPTool Structure Specifications for Filters (Continued)

Parameter	Description
`Rp`	Passband ripple (dB)
`Rs`	Stopband attenuation (dB)
`setOrderFlag`	Contains `1` if the filter order was specified manually (i.e., the **Minimum Order** box in the **Specifications** region was not checked). 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.

Accessing Parameters in a Saved Spectrum

The following structure fields describe the spectra saved by SPTool.

Field	Description
`P`	The spectral power vector.
`f`	The spectral frequency vector.

Field	Description
`confid`	A structure containing the confidence intervals data • The `confid.level` field contains the chosen confidence level. • The `confid.Pc` field contains the spectral power data for the confidence intervals. • The `confid.enable` field contains a `1` if confidence levels are enabled for the power spectral density.
`signalLabel`	The name of the signal from which the power spectral density was generated.
`Fs`	The associated signal's sample rate.

You can access the information in these fields as you do with every MATLAB structure.

For example, if you export an SPTool PSD estimate `spect1` to the workspace, type

```
spect1.P
```

to obtain the vector of associated power values.

Importing Filters and Spectra into SPTool

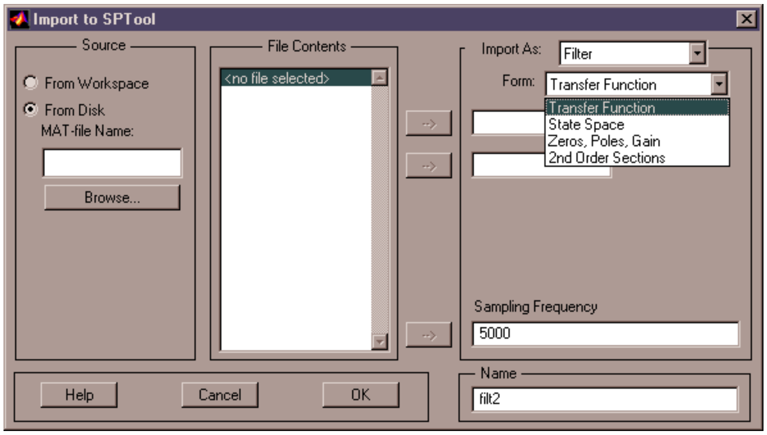
In addition to importing signals into SPTool, you can import filters or spectra into SPTool from either the workspace or from a file.

The procedures are very similar to those explained in

- "Importing a Signal into SPTool" on page 6-20 for loading variables from the workspace
- "Loading Variables from the Disk" on page 6-43 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

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

- The **Numerator** field specifies a variable name or value for the numerator coefficient vector *b*, which contains m+1 coefficients in descending powers of *z*.
- The **Denominator** field specifies a variable name or value for the denominator coefficient vector *a*, which contains n+1 coefficients in descending powers of *z*.

State Space. For `State Space`, you specify the filter by its state-space representation

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

The **A-Matrix**, **B-Matrix**, **C-Matrix**, and **D-Matrix** fields specify a variable name or a value for each matrix in this system.

Zeros, Poles, Gain. For `Zeros, Poles, Gain`, you specify the filter by its zero-pole-gain representation

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

- The **Zeros** field specifies a variable name or value for the zeros vector *z*, which contains the locations of *m* zeros.
- The **Poles** field specifies a variable name or value for the zeros vector *p*, which contains the locations of *n* poles.
- The **Gain** field specifies a variable name or value for the gain *k*.

Second Order Sections. For `2nd Order Sections` you specify the filter by its second-order section representation

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

The **SOS Matrix** field specifies a variable name or a value for the L-by-6 SOS matrix

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

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

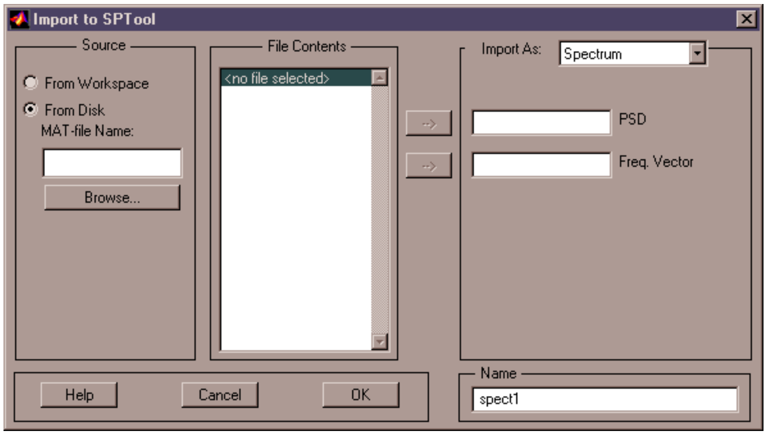
Note If you import a filter that was not created in SPTool, you can only edit that filter using the Pole/Zero Editor.

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.


Import to SPTool
Source
From Workspace
From Disk
MAT-file Name:
Browse...
File Contents
<no file selected>
Import As:
Spectrum
PSD
Freq. Vector
Help
Cancel
OK
Name
spect1

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. Press **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.

Selecting Signals, Filters, and Spectra in SPTool

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 in SPTool

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 Viewer, or the Spectrum Viewer

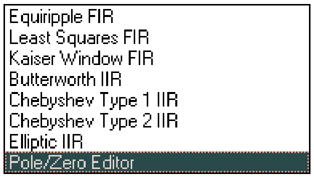
• The signal name
• The sampling frequency
• The line style properties

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.

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 × (pole) or ○ (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 × and ○ 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.

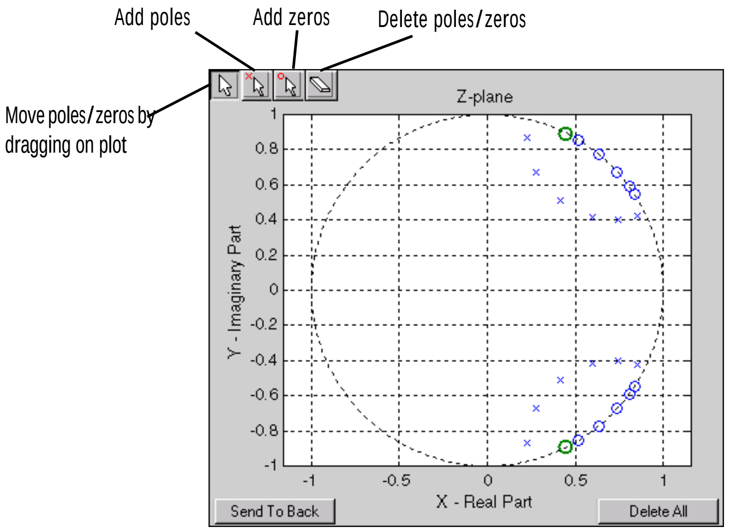
Tip Keep the Filter Viewer open while designing a filter with the Pole/Zero Editor. Any changes that you make to the filter transfer function in the Pole/Zero Editor are then simultaneously reflected in the response plots of the Filter Viewer.

Positioning Poles and Zeros

You can use your mouse to move poles and zeros around the pole/zero plot and modify your filter design

- Add poles or zeros using the toolbar buttons for pole placement, , and zero placement, .
- Erase poles and zeros using the eraser button, .
- 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 uncheck **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., ○3 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
   a. **Fp** = 800
   b. **Fs** = 700
   c. **Rp** = 2.5
   d. **Rs** = 35
5. Check **Minimum Order** so the Filter Designer can calculate the lowest filter order that produces the desired characteristics.
6. Press **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.

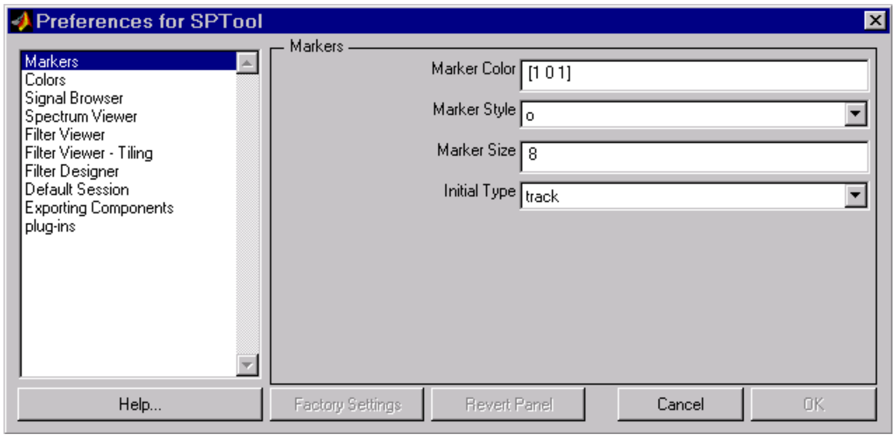
Setting Preferences

Use **Preferences** from the SPTool **File** menu to customize displays and certain parameters for SPTool and its four component GUIs. 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.
- Configure filter and display parameters, and enable/disable zoom in the Filter Viewer.
- Configure tiling preferences in the Filter Viewer.
- Specify FFT length, and enable/disable mouse zoom and grid in the Filter 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.

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



Markers

Marker Color [1 0 1]

Marker Style o

Marker Size 8

Initial Type track

You can

- Change the settings for markers from this panel of the **Preferences** dialog.
- Choose any of the other categories listed to customize its settings.

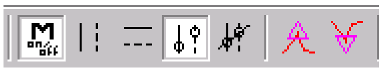
Click once on any listed category in the left pane of the **Preferences** dialog to select it.

Making Signal Measurements: Using Markers

You can use the markers on the Signal Browser, the Filter Viewer, or the Spectrum Viewer to make measurements on any of the following

- A signal in the Signal Browser
- A filter response in the Filter Viewer
- A power spectral density plotted in the Spectrum Viewer

The marker buttons from left to right are



- 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.

Function Reference

Detailed descriptions of all Signal Processing Toolbox functions are in these two sections:

- "Function Category List" – a list of functions, grouped by subject area
- "Alphabetical List of Functions" – reference pages in alphabetical order

Function Category List

The Signal Processing Toolbox functions are divided into the following categories

- “Filter Analysis”
- “Filter Implementation”
- “FIR Digital Filter Design”
- “IIR Digital Filter Design—Classical and Direct”
- “IIR Filter Order Estimation”
- “Analog Lowpass Filter Prototypes”
- “Analog Filter Design”
- “Analog Filter Transformation”
- “Filter Discretization”
- “Linear System Transformations”
- “Windows”
- “Transforms”
- “Cepstral Analysis”
- “Statistical Signal Processing and Spectral Analysis”
- “Parametric Modeling”
- “Linear Prediction”
- “Multirate Signal Processing”
- “Waveform Generation”
- “Specialized Operations”
- “Graphical User Interfaces”.

Filter Analysis	
`abs`	Absolute value (magnitude).
`angle`	Phase angle.
`freqs`	Frequency response of analog filters.
`freqspace`	Frequency spacing for frequency response.

Filter Analysis (Continued)	
freqz	Compute the frequency response of digital filters.
freqzplot	Plot frequency response data.
grpdelay	Compute the average filter delay (group delay).
impz	Compute the impulse response of digital filters.
unwrap	Unwrap phase angles.
zplane	Zero-pole plot.

Filter Implementation	
conv	Convolution and polynomial multiplication.
conv2	Two-dimensional convolution.
deconv	Deconvolution and polynomial division.
fftfilt	FFT-based FIR filtering using the overlap-add method.
filter	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.
filter2	Two-dimensional digital filtering.
filtfilt	Zero-phase digital filtering.
filtic	Find initial conditions for a transposed direct form II filter implementation.
latcfilt	Lattice and lattice-ladder filter implementation.
medfilt1	One-dimensional median filtering.
sgolayfilt	Savitzky-Golay filtering.
sosfilt	Second-order (biquadratic) IIR digital filtering.

Filter Implementation (Continued)	
`upfirdn`	Upsample, apply an FIR filter, and downsample.

FIR Digital Filter Design	
`convmtx`	Convolution matrix.
`cremez`	Complex and nonlinear-phase equiripple FIR filter design.
`fir1`	Design a window-based finite impulse response filter.
`fir2`	Design a frequency sampling-based finite impulse response filter.
`fircls`	Constrained least square FIR filter design for multiband filters.
`fircls1`	Constrained least square filter design for lowpass and highpass linear phase FIR filters.
`firls`	Least square linear-phase FIR filter design.
`firrcos`	Raised cosine FIR filter design.
`intfilt`	Interpolation FIR filter design.
`kaiserord`	Estimate parameters for an FIR filter design with Kaiser window.
`remez`	Compute the Parks-McClellan optimal FIR filter design.
`remezord`	Parks-McClellan optimal FIR filter order estimation.
`sgolay`	Savitzky-Golay filter design.

IIR Digital Filter Design—Classical and Direct	
`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.
`maxflat`	Generalized digital Butterworth filter design.
`prony`	Prony's method for time-domain IIR filter design.
`stmcb`	Compute a linear model using Steiglitz-McBride iteration.
`yulewalk`	Recursive digital filter design.

IIR Filter Order Estimation	
`buttord`	Calculate the order and cutoff frequency for a Butterworth filter.
`cheb1ord`	Calculate the order for a Chebyshev Type I filter.
`cheb2ord`	Calculate the order for a Chebyshev Type II filter.
`ellipord`	Calculate the minimum order for elliptic 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.

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 the cut-off frequency for a lowpass analog filter.

Filter Discretization	
`bilinear`	Bilinear transformation method for analog-to-digital filter conversion.
`impinvar`	Impulse invariance method for analog-to-digital filter conversion.

Linear System Transformations	
`latc2tf`	Convert lattice filter parameters to transfer function form.
`polystab`	Stabilize a polynomial.
`polyscale`	Scale the roots of a polynomial.
`residuez`	*z*-transform partial-fraction expansion.
`sos2ss`	Convert digital filter second-order section parameters to state-space form.
`sos2tf`	Convert digital filter second-order section data to transfer function form.
`sos2zp`	Convert digital filter second-order sections parameters to zero-pole-gain form.
`ss2sos`	Convert digital filter state-space parameters to second-order sections form.
`ss2tf`	Convert state-space filter parameters to transfer function form.
`ss2zp`	Convert state-space filter parameters to zero-pole-gain form.
`tf2latc`	Convert transfer function filter parameters to lattice filter form.

Linear System Transformations (Continued)	
`tf2sos`	Convert digital filter transfer function data to second-order sections form.
`tf2ss`	Convert transfer function filter parameters to state-space form.
`tf2zp`	Convert 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	
`barthannwin`	Compute a modified Bartlett-Hann window.
`bartlett`	Compute a Bartlett window.
`blackman`	Compute a Blackman window.
`blackmanharris`	Compute a minimum 4-term Blackman-Harris window.
`bohmanwin`	Compute a Bohman window.
`chebwin`	Compute a Chebyshev window.
`gausswin`	Compute a Gaussian window.
`hamming`	Compute a Hamming window.
`hann`	Compute the Hann (Hanning) window.
`kaiser`	Compute a Kaiser window.

Windows (Continued)	
`nuttallwin`	Compute a Nuttall-defined minimum 4-term Blackman-Harris window.
`rectwin`	Compute a rectangular window.
`triang`	Compute a triangular window.
`tukeywin`	Compute a Tukey (tapered cosine) window.
`window`	Window function gateway.

Transforms	
bitrevorder	Permute input into bit-reversed order.
czt	Chirp z-transform.
dct	Discrete cosine transform (DCT).
dftmtx	Discrete Fourier transform matrix.
fft	Compute the one-dimensional fast Fourier transform.
fft2	Compute the two-dimensional fast Fourier transform.
fftshift	Rearrange the outputs of the FFT functions.
goertzel	Compute the discrete Fourier transform using the second order Goertzel algorithm.
hilbert	Compute the discrete-time analytic signal using the Hilbert transform.
idct	Inverse discrete cosine transform.
ifft	One-dimensional inverse fast Fourier transform.
ifft2	Two-dimensional inverse fast Fourier transform.

Cepstral Analysis	
cceps	Complex cepstral analysis.
icceps	Inverse complex cepstrum.
rceps	Real cepstrum and minimum phase reconstruction.

Statistical Signal Processing and Spectral Analysis	
`cohere`	Estimate magnitude squared coherence function between two signals.
`corrcoef`	Compute the correlation coefficient matrix.
`corrmtx`	Compute a data matrix for autocorrelation matrix estimation.
`cov`	Compute the covariance matrix.
`csd`	Estimate the cross spectral density (CSD) of two signals.
`pburg`	Estimate the power spectral density using the Burg method.
`pcov`	Estimate the power spectral density using the covariance method.
`peig`	Estimate the pseudospectrum using the eigenvector method.
`periodogram`	Estimate the power spectral density (PSD) of a signal using a periodogram.
`pmcov`	Estimate the power spectral density using the modified covariance method.
`pmtm`	Estimate the power spectral density using the multitaper method (MTM).
`pmusic`	Estimate the power spectral density using MUSIC algorithm.
`psdplot`	Plot power spectral density (PSD) data.
`pwelch`	Estimate the power spectral density (PSD) of a signal using Welch’s method.

Statistical Signal Processing and Spectral Analysis (Continued)	
`pyulear`	Estimate the power spectral density using the Yule-Walker AR method.
`rooteig`	Estimate frequency and power content using the eigenvector method.
`rootmusic`	Estimate frequency and power content using the root MUSIC algorithm.
`tfe`	Estimate the transfer function from input and output.
`xcorr`	Estimate the cross-correlation function.
`xcorr2`	Estimate the two-dimensional cross-correlation.
`xcov`	Estimate the cross-covariance function (equal to mean-removed cross-correlation).

Parametric Modeling	
`arburg`	Compute an estimate of AR model parameters using the Burg method.
`arcov`	Compute an estimate of AR model parameters using the covariance method.
`armcov`	Compute an estimate of AR model parameters using the modified covariance method.
`aryule`	Compute an estimate of AR model parameters using the Yule-Walker method.
`ident`	See the System Identification Toolbox documentation.
`invfreqs`	Identify continuous-time filter parameters from frequency response data.

Parametric Modeling (Continued)	
`invfreqz`	Identify discrete-time filter parameters from frequency response data.
`prony`	Prony's method for time domain IIR filter design.
`stmcb`	Compute a linear model using Steiglitz-McBride iteration.

Linear Prediction	
`ac2poly`	Convert an autocorrelation sequence to prediction polynomial.
`ac2rc`	Convert an autocorrelation sequence to reflection coefficients.
`is2rc`	Convert inverse sine parameters to reflection coefficients.
`lar2rc`	Convert log area ratio parameters to reflection coefficients.
`levinson`	Compute the Levinson-Durbin recursion.
`lpc`	Compute linear prediction filter coefficients.
`lsf2poly`	Convert line spectral frequencies to a prediction filter coefficients.
`poly2ac`	Convert a prediction filter polynomial to an autocorrelation sequence.
`poly2lsf`	Convert prediction filter coefficients to line spectral frequencies.
`poly2rc`	Convert a prediction filter polynomial to reflection coefficients.
`rc2ac`	Convert reflection coefficients to an autocorrelation sequence.
`rc2is`	Convert reflection coefficients to inverse sine parameters.
`rc2lar`	Convert reflection coefficients to log area ratio parameters.
`rc2poly`	Convert reflection coefficients to a prediction filter polynomial.
`rlevinson`	Compute the reverse Levinson-Durbin recursion.
`schurrc`	Compute reflection coefficients from an autocorrelation sequence.

Multirate Signal Processing	
`decimate`	Decrease the sampling rate for a sequence (decimation).
`downsample`	Reduce the sampling rate by an integer factor.
`interp`	Increase sampling rate by an integer factor (interpolation).
`interp1`	One-dimensional data interpolation (table lookup).
`resample`	Change sampling rate by any rational factor.
`spline`	Cubic spline interpolation.
`upfirdn`	Upsample, apply an FIR filter, and downsample.
`upsample`	Increase the sampling rate by an integer factor

Waveform Generation	
`chirp`	Generate a swept-frequency cosine.
`diric`	Compute the Dirichlet or periodic sinc function.
`gauspuls`	Generate a Gaussian-modulated sinusoidal pulse.
`gmonopuls`	Generate a Gaussian monopulse.
`pulstran`	Generate a pulse train.
`rectpuls`	Generate a sampled aperiodic rectangle.
`sawtooth`	Generate a sawtooth or triangle wave.
`sinc`	Sinc function.
`square`	Generate a square wave.
`tripuls`	Generate a sampled aperiodic triangle.
`vco`	Voltage controlled oscillator.

Specialized Operations	
`buffer`	Buffer a signal vector into a matrix of data frames.
`cell2sos`	Convert a cell array for second-order sections to a second-order section matrix.
`cplxpair`	Group complex numbers into complex conjugate pairs.
`demod`	Demodulation for communications simulation.
`dpss`	Discrete prolate spheroidal sequences (Slepian sequences).
`dpssclear`	Remove discrete prolate spheroidal sequences from database.
`dpssdir`	Discrete prolate spheroidal sequences database directory.
`dpssload`	Load discrete prolate spheroidal sequences from database.
`dpsssave`	Save discrete prolate spheroidal sequences in database.
`eqtflength`	Make the lengths of a transfer function's numerator and denominator equal.
`modulate`	Modulation for communications simulation.
`seqperiod`	Compute the period of a sequence.
`sos2cell`	Convert a second-order section matrix to cell arrays.
`specgram`	Time-dependent frequency analysis (spectrogram).
`stem`	Plot discrete sequence data.
`strips`	Strip plot.
`udecode`	Decode 2^n-level quantized integer inputs to floating-point outputs.
`uencode`	Quantize and encode floating-point inputs to integer outputs.

Graphical User Interfaces	
`fdatool`	Open the Filter Design and Analysis Tool.
`fvtool`	Open the Filter Visualization Tool.
`sptool`	Interactive digital signal processing tool (SPTool).

Alphabetical List of Functions

abs

Purpose Absolute value (magnitude)

Syntax

```
y = abs(x)
```

Description

y = abs(x) returns the absolute value of the elements of x. If x is complex, abs returns the complex modulus (magnitude).

```
abs(x) = sqrt(real(x).^2 + imag(x).^2)
```

If x is a MATLAB string, abs returns the numeric values of the ASCII characters in the string. The display format of the string changes; the internal representation does not.

The abs function is part of the standard MATLAB language.

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)
```

See Also

angle

ac2poly

Purpose Convert an autocorrelation sequence to a linear prediction filter 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.

Example 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
```

See Also `ac2rc`, `poly2ac`, `rc2poly`

References [1] Kay, S.M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Purpose Convert an autocorrelation sequence to reflection coefficients

Syntax `[k,r0] = ac2rc(r)`

Description `[k,r0] = 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`.

Remarks You can apply this function to real or complex data.

See Also `ac2poly`, `poly2rc`, `rc2ac`

References [1] Kay, S.M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

angle

Purpose Phase angle

Syntax

```
p = angle(h)
```

Description `p = angle(h)` returns the phase angles, in radians, of the elements of complex vector or array `h`. The phase angles lie between $-\pi$ and π.

For complex sequence $h = x + iy = me^{ip}$, the magnitude and phase are given by

```
m = abs(h)
p = angle(h)
```

To convert to the original `h` from its magnitude and phase, type

```
i = sqrt(-1)
h = m.*exp(i*p)
```

The `angle` function is part of the standard MATLAB language.

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)
```

Algorithm `angle` can be expressed as

```
angle(x) = imag(log(x)) = atan2(imag(x),real(x))
```

See Also `abs`

arburg

Purpose Compute an estimate of AR model parameters using the 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 pth 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.

$$H(z) = \frac{\sqrt{e}}{A(z)} = \frac{\sqrt{e}}{1 + a_2 z^{-1} + \ldots + a_{(p+1)} z^{-p}}$$

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`

Purpose Compute an estimate of AR model parameters using the covariance method

Syntax

```
a = arcov(x,p)
[a,e] = arcov(x,p)
```

Description `a = arcov(x,p)` uses the covariance method to fit a pth 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.

$$H(z) = \frac{\sqrt{e}}{A(z)} = \frac{\sqrt{e}}{1 + a_2 z^{-1} + \ldots + a_{(p+1)} z^{-p}}$$

Because the method characterizes the input data using an all-pole model, the correct choice of the model order `p` is important.

`[a,e] = arcov(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

Compute an estimate of AR model parameters using the 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 pth 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.

$$H(z) = \frac{\sqrt{e}}{A(z)} = \frac{\sqrt{e}}{1 + a_2 z^{-1} + \ldots + a_{(p+1)} z^{-p}}$$

Because the method characterizes the input data using an all-pole model, the correct choice of the model order `p` is important.

`[a,e] = 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`

Purpose

Compute an estimate of AR model parameters using the 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 pth 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.

$$H(z) = \frac{\sqrt{e}}{A(z)} = \frac{\sqrt{e}}{1 + a_2 z^{-1} + \ldots + a_{(p+1)} z^{-p}}$$

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`

barthannwin

Purpose Compute a modified Bartlett-Hann window

Syntax

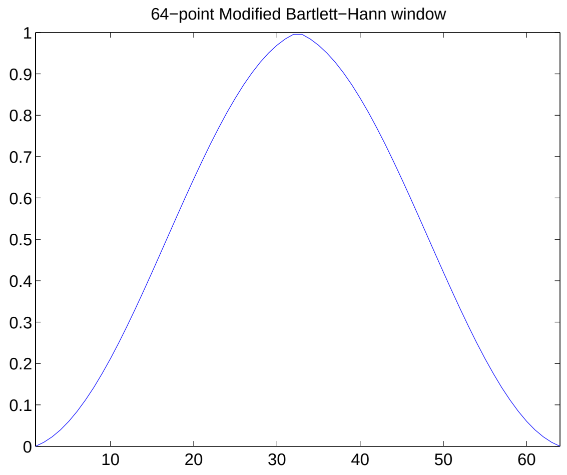
```
w = barthannwin(n)
```

Description `w = barthannwin(n)` returns an `n`-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.

Example

```
N=64;
w = barthannwin(N);
plot(w); axis([1 N 0 1]);
title('64-point Modified Bartlett-Hann window');
```



Algorithm

The equation for computing the coefficients of a Modified Bartlett-Hanning window is

$$w[k+1] = 0.62 - 0.48\left|\left(\frac{k}{n-1} - 0.5\right)\right| + 0.38\cos\left(2\pi\left(\frac{k}{n-1} - 0.5\right)\right)$$

where $0 \le k \le (n-1)$.

See Also

`bartlett`, `hann`, `blackman`, `blackmanharris`, `bohmanwin`, `chebwin`, `gausswin`, `hamming`, `kaiser`, `nuttallwin`, `rectwin`, `triang`, `tukeywin`, `window`

References

[1] 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.

[2] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, p. 468.

Purpose Compute a Bartlett window

Syntax

```
w = bartlett(n)
```

Description `w = bartlett(n)` returns an `n`-point Bartlett window in the column vector `w`, where `n` must be a positive integer. The coefficients of a Bartlett window are computed as follows:

- For `n` odd

$$w[k+1] = \begin{cases} \dfrac{2k}{n-1}, & 0 \le k \le \dfrac{n-1}{2} \\ 2 - \dfrac{2(k)}{n-1}, & \dfrac{n-1}{2} \le k \le n-1 \end{cases}$$

- For `n` even

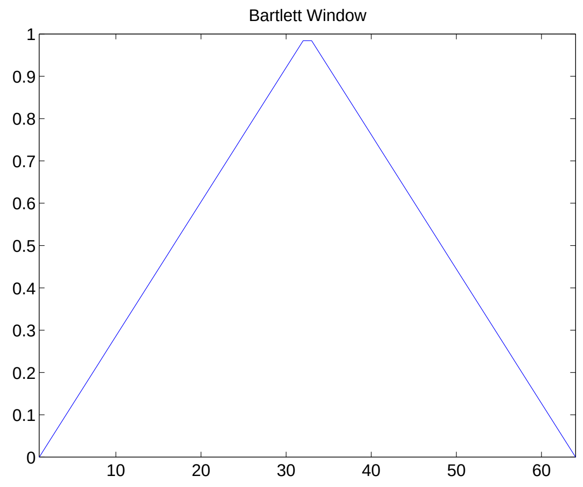
$$w[k+1] = \begin{cases} \dfrac{2(k)}{n-1}, & 0 \le k \le \dfrac{n}{2} - 1 \\ \dfrac{2(n-k-1)}{n-1}, & \dfrac{n}{2} \le k \le n-1 \end{cases}$$

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 `n` odd, the center `n-2` points of `bartlett(n)` are equivalent to `triang(n-2)`.

Note If you specify a one-point window (set `n=1`), the value `1` is returned.

Example

```
N=64;
w = bartlett(N);
plot(w); axis([1 N 0 1]);
title('Bartlett Window')
```



See Also `barthannwin`, `blackman`, `blackmanharris`, `bohmanwin`, `chebwin`, `gausswin`, `hamming`, `hann`, `kaiser`, `nuttallwin`, `rectwin`, `triang`, `tukeywin`, `window`

References [1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

besselap

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 look-up table constructed using the Symbolic Math Toolbox.

See Also `besself`, `buttap`, `cheb1ap`, `cheb2ap`, `ellipap`

Also see the Symbolic Math Toolbox documentation.

References [1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pgs. 228-230.

Purpose

Bessel analog filter design

Syntax

```
[b,a] = besself(n,Wn)
[b,a] = besself(n,Wn,'ftype')
[z,p,k] = besself(...)
[A,B,C,D] = besself(...)
```

Description

`besself` designs lowpass, bandpass, highpass, and bandstop analog Bessel filters. Analog Bessel filters are characterized by almost constant group delay across the entire passband, thus preserving the wave shape of filtered signals in the passband. Digital Bessel filters do not retain this quality, and `besself` therefore does not support the design of digital Bessel filters.

`[b,a] = besself(n,Wn)` designs an order `n` lowpass analog filter with cutoff frequency `Wn`. 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

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{s^n + a(2)s^{n-1} + \cdots + a(n+1)}$$

Cutoff frequency is the frequency at which the magnitude response of the filter begins to decrease significantly. For `besself`, the cutoff frequency `Wn` must be greater than 0. The magnitude response of a Bessel filter designed by `besself` is always less than $\sqrt{1/2}$ at the cutoff frequency, and it decreases as the order `n` increases.

If `Wn` is a two-element vector, `Wn = [w1 w2]` with `w1` < `w2`, then `besself(n,Wn)` returns an order `2*n` bandpass analog filter with passband `w1` < ω < `w2`.

`[b,a] = besself(n,Wn,'ftype')` designs a highpass or bandstop filter, where the string `'ftype'` is:

- `'high'` for a highpass analog filter with cutoff frequency `Wn`
- `'stop'` for an order `2*n` bandstop analog filter if `Wn` is a two-element vector, `Wn = [w1 w2]`

  The stopband is `w1` < ω < `w2`.

With different numbers of output arguments, `besself` directly obtains other realizations of the analog filter. To obtain zero-pole-gain form, use three output arguments as shown below.

`[z,p,k] = besself(n,Wn)` or

`[z,p,k] = besself(n,Wn,'`*`ftype`*`')` 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] = besself(n,Wn)` or

`[A,B,C,D] = besself(n,Wn,'`*`ftype`*`')` where `A`, `B`, `C`, and `D` are

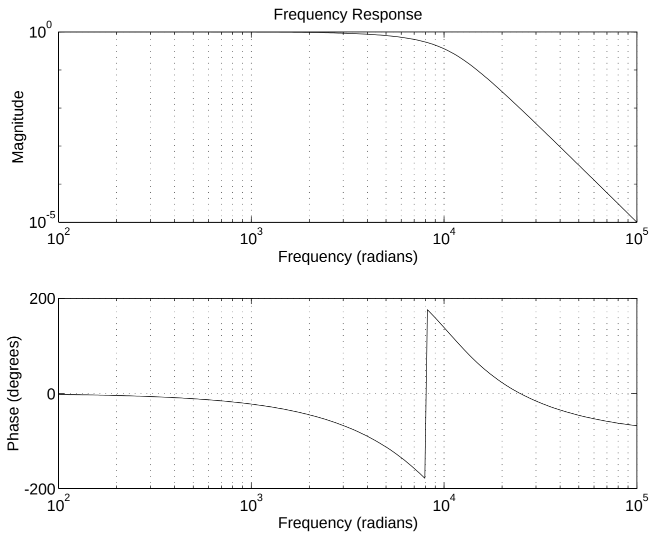
$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

and u is the input, x is the state vector, and y is the output.

Example

Design a fifth-order analog lowpass Bessel filter that suppresses frequencies greater than 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 filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies using a state-space transformation.
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 Ω = -∞ to +∞) repeatedly around the unit circle ($e^{j\omega}$, from $\omega = -\pi$ to π) 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

$$H(z) = H(s)\Big|_{s = \frac{2\pi f_p}{\tan\left(\pi\frac{f_p}{f_s}\right)}\frac{(z-1)}{(z+1)}}$$

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 π) 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`. `Fp` is the optional match frequency, in hertz, 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.

$$\frac{num(s)}{den(s)} = \frac{num(1)s^n + \cdots + num(n)s + num(n+1)}{den(1)s^m + \cdots + den(m)s + den(m+1)}$$

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,

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

to the discrete-time system

$$\begin{aligned} x[n+1] &= A_d x[n] + B_d u[n] \\ y[n] &= C_d x[n] + D_d u[n] \end{aligned}$$

fs is the sampling frequency in hertz. bilinear returns the discrete equivalent in matrices Ad, Bd, Cd, Dd. Fp is the optional match frequency, in hertz, 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, k = 2*pi*Fp/tan(pi*Fp/fs); otherwise k = 2*fs.
2. It strips any zeros at $\pm\infty$ using

```
z = z(find(finite(z)));
```

3 It transforms the zeros, poles, and gain using

```
pd = (1+p/k)./(1-p/k);
zd = (1+z/k)./(1-z/k);
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 `Fp` is present, `k = 2*pi*Fp/tan(pi*Fp/fs)`; else `k = 2*fs`.

2 It computes `Ad`, `Bd`, `Cd`, and `Dd` in terms of `A`, `B`, `C`, and `D` using

$$A_d = \left(I + \left(\frac{1}{k}\right)A\right)\left(I - \left(\frac{1}{k}\right)A\right)^{-1}$$

$$B_d = \frac{2k}{r}\left(I - \left(\frac{1}{k}\right)A\right)^{-1}B$$

$$C_d = rC\left(I - \left(\frac{1}{k}\right)A\right)^{-1}$$

$$D_d = \left(\frac{1}{k}\right)C\left(I - \left(\frac{1}{k}\right)A\right)^{-1}B + D$$

`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.
```

See Also

`impinvar`, `lp2bp`, `lp2bs`, `lp2hp`, `lp2lp`

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. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 450-454.

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 computation. This can improve run-time efficiency for external applications or for Simulink Blockset models. Note that MATLAB `fft` and `ifft` process linear input and output.

`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
5	101	101	5
6	110	011	3
7	111	111	7

Example 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`, `ifft`

Purpose

Compute a Blackman window

Syntax

```
w = blackman(n)
w = blackman(n,'sflag')
```

Description

`w = blackman(n)` returns the `n`-point symmetric Blackman window in the column vector `w`, where `n` is a positive integer.

`w = blackman(n,'sflag')` returns an `n`-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 `n+1` window and returns the first `n` points.

Note If you specify a one-point window (set `n=1`), the value `1` is returned.

Algorithm

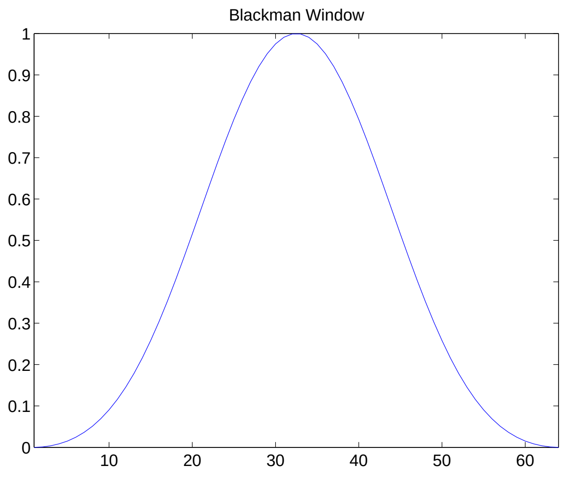
The equation for computing the coefficients of a Blackman window is

$$w[k+1] = 0.42 - 0.5\cos\left(2\pi\frac{k}{n-1}\right) + 0.08\cos\left(4\pi\frac{k}{n-1}\right), \quad k = 0, \ldots, n-1$$

Blackman windows have slightly wider central lobes and less sideband leakage than equivalent length Hamming and Hann windows.

Examples

```
N=64;
w = blackman(N);
plot(w); axis([1 N 0 1]);
title('Blackman Window')
```



Algorithm

The equation for computing the coefficients of a Blackman window is

$$w[k+1] = 0.42 - 0.5\cos\left(2\pi\frac{k}{n-1}\right) + 0.08\cos\left(4\pi\frac{k}{n-1}\right), \quad k = 0, \ldots, n-1$$

Blackman windows have slightly wider central lobes and less sideband leakage than equivalent length Hamming and Hann windows.

See Also

`barthannwin`, `bartlett`, `blackmanharris`, `bohmanwin`, `chebwin`, `gausswin`, `hamming`, `hann`, `kaiser`, `nuttallwin`, `rectwin`, `triang`, `tukeywin`, `window`

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

Purpose

Compute a minimum 4-term Blackman-harris window

Syntax

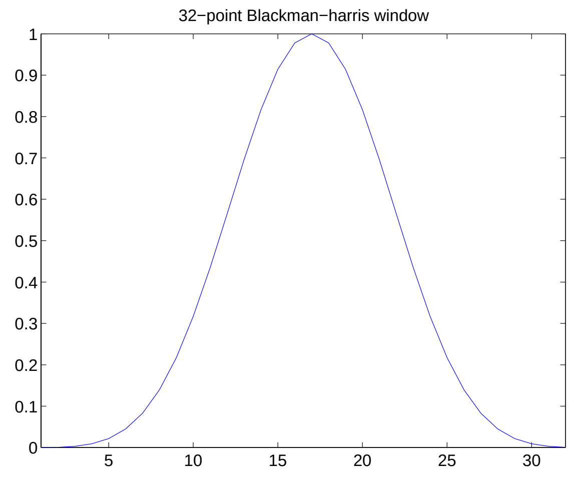
```
w = blackmanharris(n)
```

Description

`w = blackmanharris(n)` returns an n-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.

Example

```
N=32;
w = blackmanharris(N);
plot(w); axis([1 N 0 1]);
title('32-point Blackman-harris window');
```



Algorithm

The equation for computing the coefficients of a minimum 4-term Blackman-harris window is

$$w[k+1] = a_0 - a_1\cos\left(2\pi\frac{k}{n-1}\right) + a_2\cos\left(4\pi\frac{k}{n-1}\right) - a_3\cos\left(6\pi\frac{k}{n-1}\right)$$

where $0 \le k \le (n-1)$.

The coefficients for this window are

$$a_0 = 0.35875$$

$$a_1 = 0.48829$$

$$a_2 = 0.14128$$

$$a_3 = 0.01168$$

See Also

`barthannwin`, `bartlett`, `blackman`, `bohmanwin`, `chebwin`, `gausswin`, `hann`, `hamming`, `kaiser`, `nuttallwin`, `rectwin`, `triang`, `tukeywin`, `window`

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.

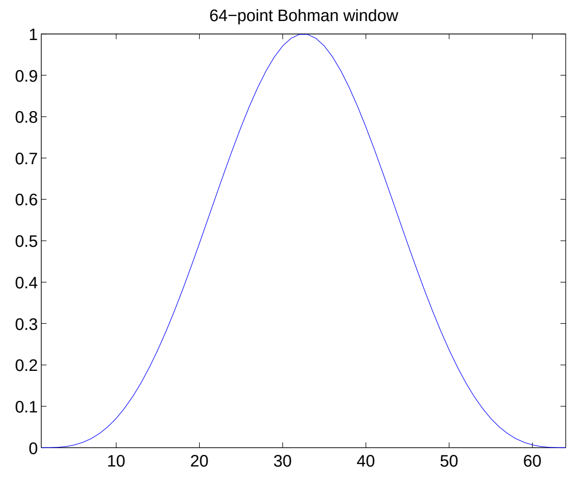
Purpose Compute a Bohman window

Syntax `w = bohmanwin(n)`

Description `w = bohmanwin(n)` returns an `n`-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 $1/w^4$.

Example Compute a 64-point Bohman window.

```
N=64;
w = bohmanwin(N);
plot(w); axis([1 N 0 1]);
title('64-point Bohman window');
```



Algorithm The equation for computing the coefficients of a Bohman window is

$$w[k+1] = \left[1.0 - \frac{\left(k - \frac{N}{2}\right)}{\frac{N}{2}}\right]\cos\left[\pi\frac{\left(k - \frac{N}{2}\right)}{\frac{N}{2}}\right] + \frac{1}{\pi}\sin\left[\pi\frac{\left(k - \frac{N}{2}\right)}{\frac{N}{2}}\right]$$

where $0 \le |k| \le N$.

See Also

`barthannwin`, `bartlett`, `blackman`, `blackmanharris`, `chebwin`, `gausswin`, `hann`, `hamming`, `kaiser`, `nuttallwin`, `rectwin`, `triang`, `tukeywin`, `window`

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.

Purpose

Buffer a signal vector into a matrix of data frames

Syntax

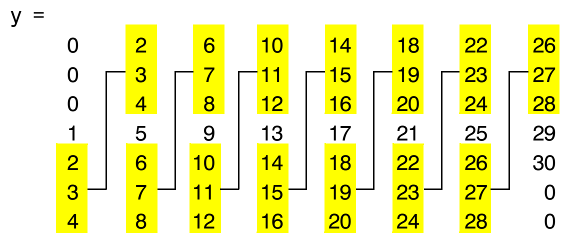
```
y = buffer(x,n)
y = buffer(x,n,p)
y = buffer(x,n,p,opt)
[y,z] = buffer(...)
[y,z,opt] = buffer(...)
```

Description

`y = 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 `ceil(L/n)` columns. If `L` is not evenly divisible by `n`, the last column is zero-padded to length `n`.

`y = 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.

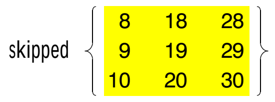


```
y =
     0     2     6    10    14    18    22    26
     0     3     7    11    15    19    23    27
     0     4     8    12    16    20    24    28
     1     5     9    13    17    21    25    29
     2     6    10    14    18    22    26    30
     3     7    11    15    19    23    27     0
     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 `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.

```
y =
      1     11     21
      2     12     22
      3     13     23
      4     14     24
      5     15     25
      6     16     26
      7     17     27
```

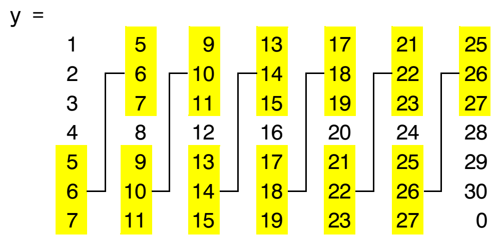


  The number of columns in y is `ceil(L/(n-p))`.

y = `buffer(x,n,p,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), `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, `opt` should contain the final p samples of the previous buffer in the sequence. See “Continuous Buffering” below.

  By default, `opt` is `zeros(p,1)` for an overlapping buffer. Set `opt` to `'nodelay'` to skip the initial condition and begin filling the buffer immediately with `x(1)`. In this case, L must be `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), `opt` is an integer value in the range `[0,-p]` specifying the number of initial input samples, `x(1:opt)`, to skip before adding samples to the buffer. The first value in the buffer is therefore `x(opt+1)`. By default, `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, `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 `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 = floor(L/(n-p))              % When length(opt) = p
```

or

```
m = floor((L-n)/(n-p))+1        % When opt = 'nodelay'
```

If `y` is an underlapping buffer, it has `n` rows and `m` columns, where

```
m = floor((L-opt)/(n-p)) + (rem((L-opt),(n-p)) >= 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

```
length(z) = L - m*(n-p)         % When length(opt) = p
```

or

```
length(z) = L - ((m-1)*(n-p)+n)% When opt = 'nodelay'
```

and for an underlapping buffer has length

```
length(z) = (L-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.

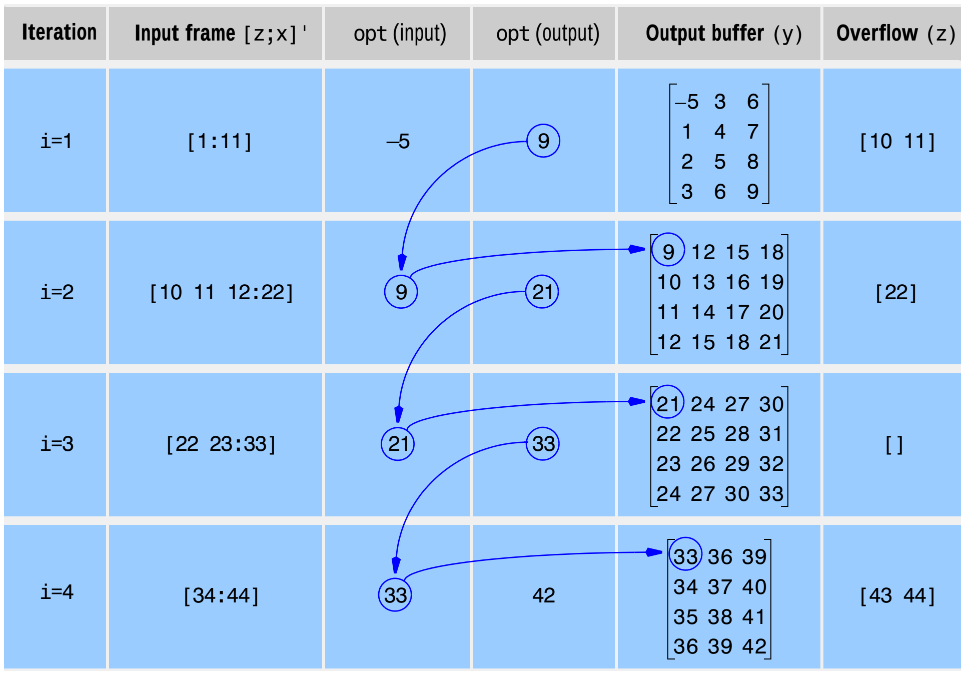
Set the buffer parameters.

```
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.

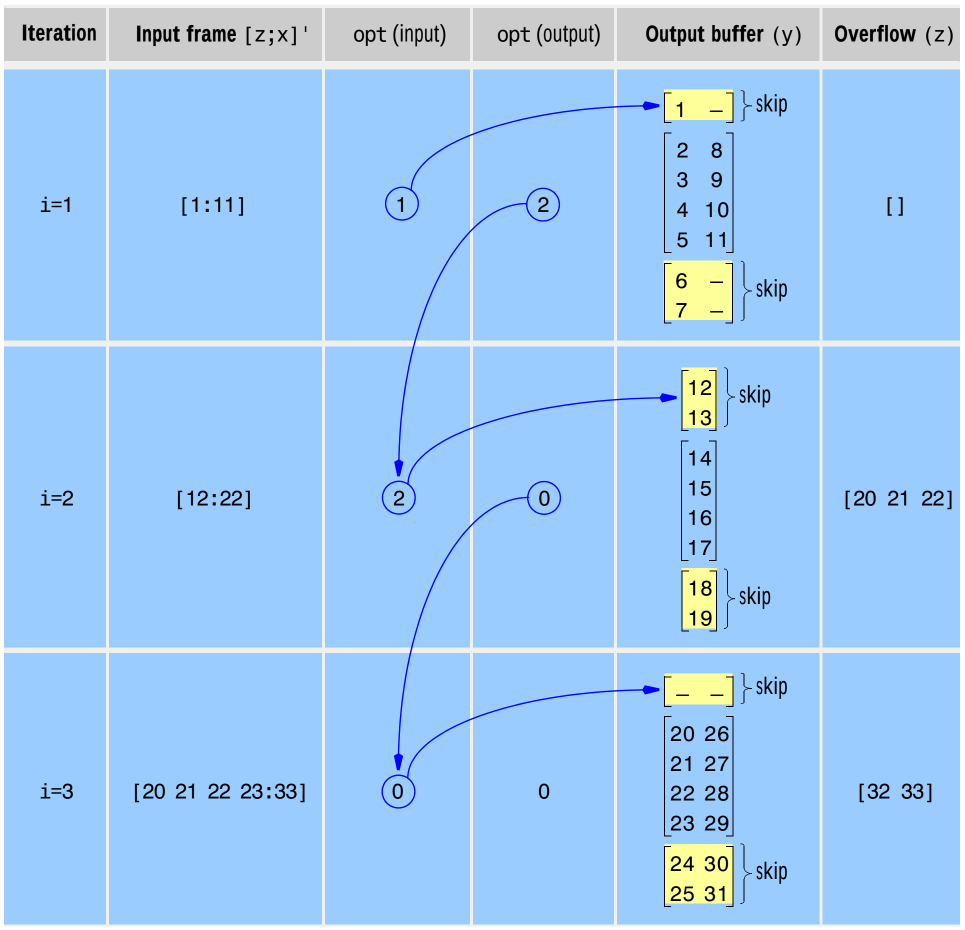
Set the buffer parameters.

```
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.



Diagnostics

Error messages are displayed when p ≥ n or `length(opt)` ≠ `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`

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 frequency ω_0 is always $1/\sqrt{2}$ regardless of the filter order. `buttap` sets ω_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));
```

See Also

`besselap`, `butter`, `cheb1ap`, `cheb2ap`, `ellipap`

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Chapter 7.

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 cutoff frequency `Wn`. It returns the filter coefficients in length `n+1` row vectors `b` and `a`, with coefficients in descending powers of z.

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

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, π 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` < ω < `w2`.

`[b,a] = butter(n,Wn,'ftype')` designs a highpass or bandstop filter, where the string `'ftype'` is either:

- `'high'` for a highpass digital filter with 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` < ω < `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

$$\begin{aligned} x[n+1] &= Ax[n] + Bu[n] \\ y[n] &= Cx[n] + Du[n] \end{aligned}$$

and u is the input, x is the state vector, and y is the output.

Analog Domain

`[b,a] = butter(n,Wn,'s')` designs an order `n` lowpass analog Butterworth filter with 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 the transfer function

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{s^n + a(2)s^{n-1} + \cdots + a(n+1)}$$

`butter`'s 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` < ω < `w2`.

`[b,a] = butter(n,Wn,'`*`ftype`*`','`**`s`**`')` designs a highpass 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

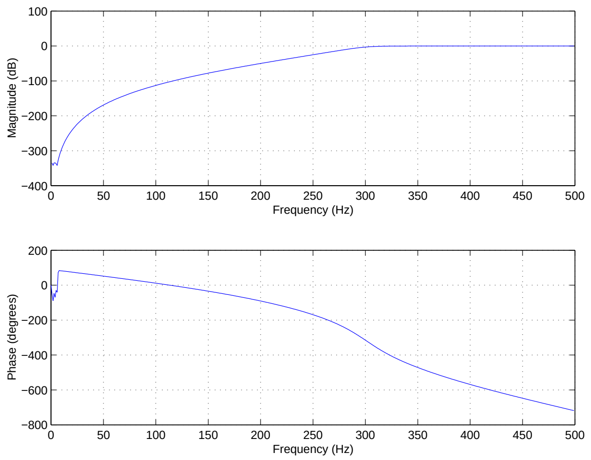
Example 1

For data sampled at 1000 Hz, design a 9th-order highpass Butterworth filter with cutoff frequency of 300 Hz.

```
[b,a] = butter(9,300/500,'high');
```

The filter's frequency response is

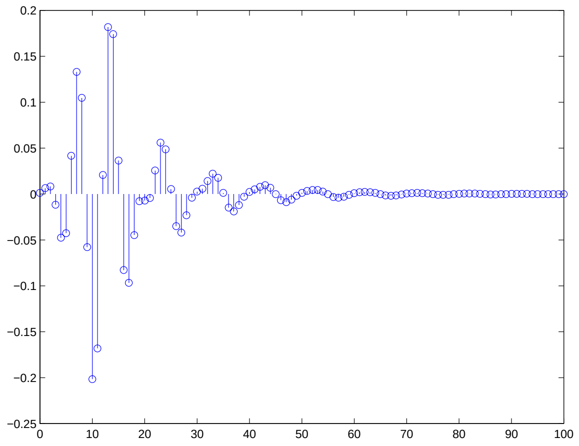
```
freqz(b,a,128,1000)
```



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 `Wn` or `w1` and `w2`.
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`

Purpose

Calculate the order and cutoff frequency for a Butterworth filter

Syntax

```
[n,Wn] = buttord(Wp,Ws,Rp,Rs)
[n,Wn] = 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] = 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.

Table 7-1: Description of Stopband and Passband Filter Parameters

`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, π 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.

Table 7-2: Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	`Wp` < `Ws`, both scalars	`(Ws,1)`	`(0,Wp)`
Highpass	`Wp` > `Ws`, both scalars	`(0,Ws)`	`(Wp,1)`
Bandpass	The interval specified by `Ws` contains the one specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`).	`(0,Ws(1))` and `(Ws(2),1)`	`(Wp(1),Wp(2))`
Bandstop	The interval specified by `Wp` contains the one specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`).	`(0,Wp(1))` and `(Wp(2),1)`	`(Ws(1),Ws(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,Wn] = buttord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Wn` for an analog Butterworth filter. You specify the frequencies `Wp` and `Ws` similar to Table 7-1, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

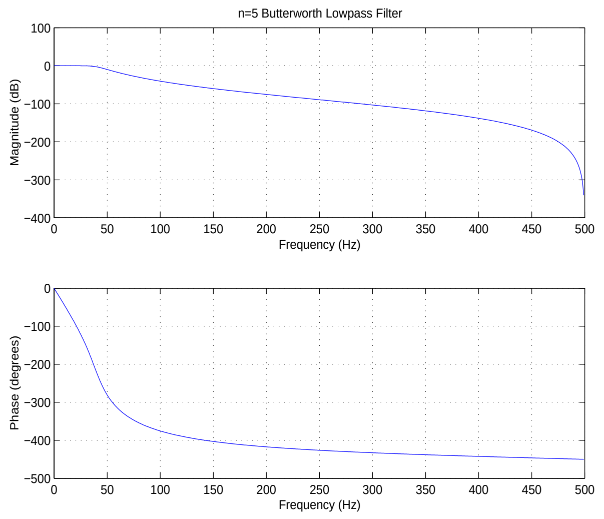
Use `buttord` for lowpass, highpass, bandpass, and bandstop filters as described in Table 7-2.

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.

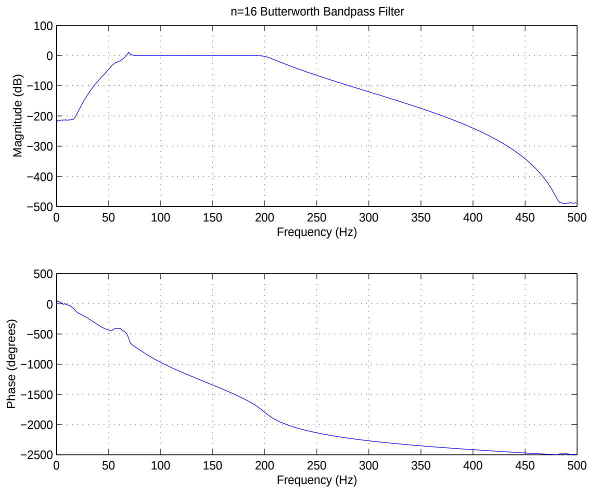
```
Wp = 40/500; Ws = 150/500;
[n,Wn] = buttord(Wp,Ws,3,60)

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')
```



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.

See Also

`butter`, `cheb1ord`, `cheb2ord`, `ellipord`, `kaiserord`

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 227.

cceps

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].

`xhat = cceps(x)` returns the complex cepstrum of the (assumed real) 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 π 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`, in its basic form, is an M-file implementation of algorithm 7.1 in [2]. A lengthy Fortran program reduces to three lines of MATLAB code:

```
h = fft(x);
logh = log(abs(h)) + sqrt(-1)*rcunwrap(angle(h));
y = real(ifft(logh));
```

`rcunwrap` is a special version of `unwrap` that subtracts a straight line from the phase.

See Also `icceps`, `hilbert`, `rceps`, `unwrap`

References

[1] Oppenheim, A.V., and R.W. Schafer. *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.

Purpose

Convert a cell array for second-order sections to a second-order section 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 = { {b1 a1} {b2 a2} ... {bL aL} }
```

where both `bi` and `ai` are row vectors of at most length 3, and i = 1, 2, ..., L. The resulting matrix `m` is given by

```
m = [b1 a1;b2 a2; ... ;bL aL]
```

See Also

`sos2cell`, `tf2sos`

cheb1ap

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 cutoff frequency ω_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 $10^{-Rp/20}$.

See Also

`besselap`, `buttap`, `cheby1`, `cheb2ap`, `ellipap`

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Chapter 7.

Purpose

Calculate the order for a Chebyshev Type I filter

Syntax

```
[n,Wn] = cheb1ord(Wp,Ws,Rp,Rs)
[n,Wn] = 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,Wn] = cheb1ord(Wp,Ws,Rp,Rs)` returns the lowest order `n` of the Chebyshev Type I 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` with the `cheby1` function.

Choose the input arguments to specify the stopband and passband according to the following table.

Table 7-3: Description of Stopband and Passband Filter Parameters

`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, π 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

Table 7-4: Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	`Wp` < `Ws`, both scalars	`(Ws,1)`	`(0,Wp)`
Highpass	`Wp` > `Ws`, both scalars	`(0,Ws)`	`(Wp,1)`
Bandpass	The interval specified by `Ws` contains the one specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`).	`(0,Ws(1))` and `(Ws(2),1)`	`(Wp(1),Wp(2))`
Bandstop	The interval specified by `Wp` contains the one specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`).	`(0,Wp(1))` and `(Wp(2),1)`	`(Ws(1),Ws(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,Wn] = cheb1ord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Wn` for an analog Chebyshev Type I filter. You specify the frequencies `Wp` and `Ws` similar to Table 7-3, 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 Table 7-4.

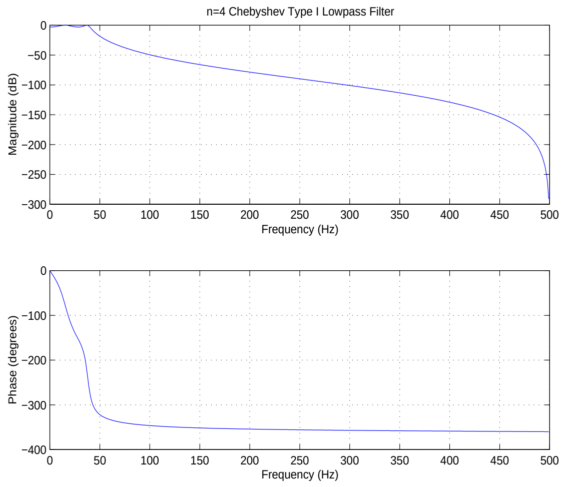
Examples

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,Wn] = cheb1ord(Wp,Ws,Rp,Rs)
```

```
n =
     4
Wn =
    0.0800
```

```
[b,a] = cheby1(n,Rp,Wn);
freqz(b,a,512,1000);
title('n=4 Chebyshev Type I Lowpass Filter')
```

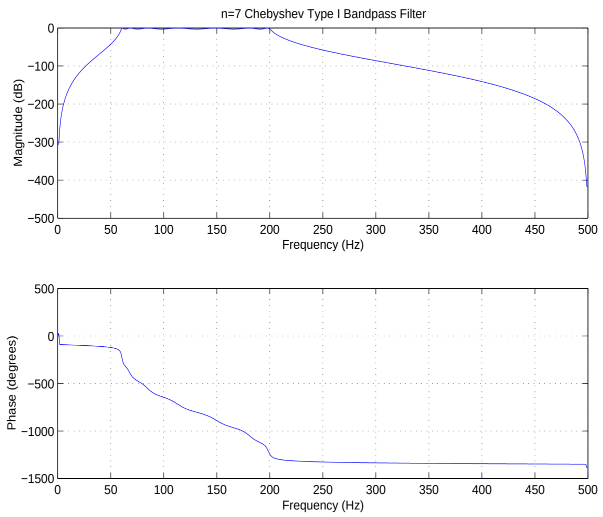


Next design a bandpass filter with a passband of 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband, and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband.

```
Wp = [60 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 40;
[n,Wn] = cheb1ord(Wp,Ws,Rp,Rs)
```

```
n =
     7
Wn =
    0.1200    0.4000
```

```
[b,a] = cheby1(n,Rp,Wn);
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` 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.

See Also

`buttord`, `cheby1`, `cheb2ord`, `ellipord`, `kaiserord`

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 241.

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 `Rs` 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 cutoff frequency ω_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 $10^{-Rs/20}$.

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 ω 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.

See Also `besselap`, `buttap`, `cheb1ap`, `cheby2`, `ellipap`

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Chapter 7.

Purpose

Calculate the order for a Chebyshev Type II filter

Syntax

```
[n,Wn] = cheb2ord(Wp,Ws,Rp,Rs)
[n,Wn] = 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,Wn] = cheb2ord(Wp,Ws,Rp,Rs) returns the lowest order n of the Chebyshev Type II 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 cheby2.

Choose the input arguments to specify the stopband and passband according to the following table.

Table 7-5: Description of Stopband and Passband Filter Parameters

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, π 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.

Table 7-6: Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	`Wp` < `Ws`, both scalars	`(Ws,1)`	`(0,Wp)`
Highpass	`Wp` > `Ws`, both scalars	`(0,Ws)`	`(Wp,1)`
Bandpass	The interval specified by `Ws` contains the one specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`).	`(0,Ws(1))` and `(Ws(2),1)`	`(Wp(1),Wp(2))`
Bandstop	The interval specified by `Wp` contains the one specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`).	`(0,Wp(1))` and `(Wp(2),1)`	`(Ws(1),Ws(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,Wn] = cheb2ord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Wn` for an analog Chebyshev Type II filter. You specify the frequencies `Wp` and `Ws` similar to Table 7-5, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

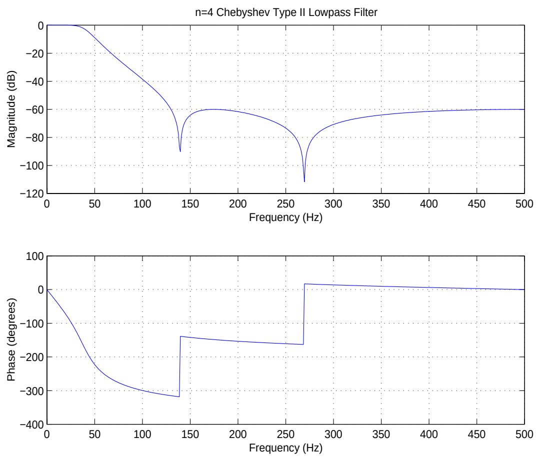
Use `cheb2ord` for lowpass, highpass, bandpass, and bandstop filters as described in Table 7-6.

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).

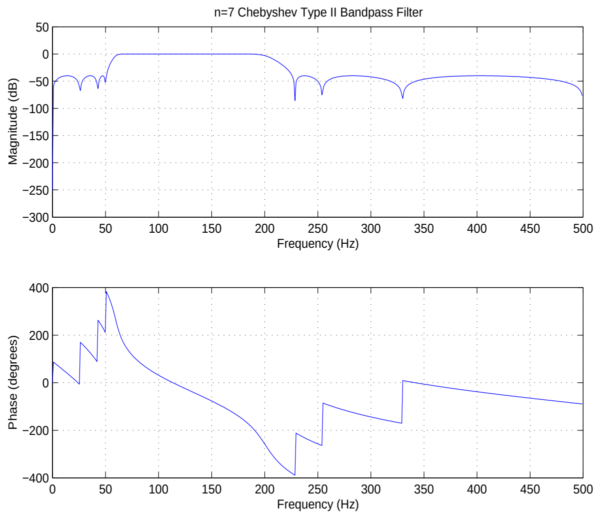
```
Wp = 40/500; Ws = 150/500;
Rp = 3; Rs = 60;
[n,Wn] = cheb2ord(Wp,Ws,Rp,Rs)

n =
     4

Wn =
    0.2597

[b,a] = cheby2(n,Rs,Wn);
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,Wn] = cheb2ord(Wp,Ws,Rp,Rs)

n =
     7

Wn =
    0.1019    0.4516

[b,a] = cheby2(n,Rs,Wn);
freqz(b,a,512,1000)
title('n=7 Chebyshev Type II Bandpass Filter')
```



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.

See Also

`buttord`, `cheb1ord`, `cheby2`, `ellipord`, `kaiserord`

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 241.

Purpose

Compute a Chebyshev window

Syntax

```
w = chebwin(n,r)
```

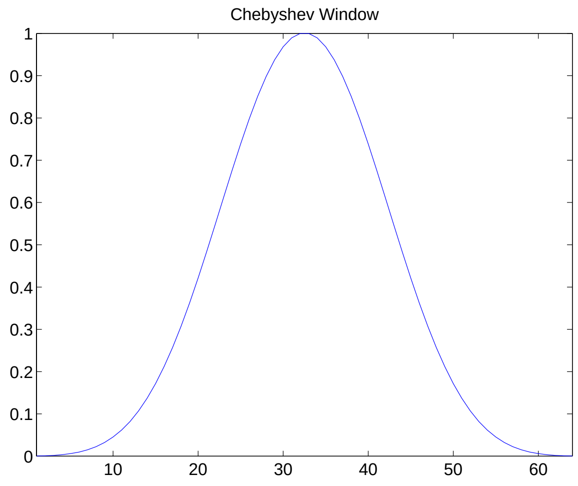
Description

`w = chebwin(n,r)` returns the column vector `w` containing the length `n` Chebyshev window whose Fourier transform sidelobe magnitude is `r` dB below the mainlobe magnitude.

Note If you specify a one-point window (set `n=1`), the value `1` is returned.

Examples

```
N=64;
w = chebwin(N,100);
plot(w); axis([1 N 0 1]);
title('Chebyshev Window')
```



See Also

`barthannwin`, `bartlett`, `blackman`, `blackmanharris`, `bohmanwin`, `gausswin`, `hamming`, `hann`, `kaiser`, `nuttallwin`, `rectwin`, `triang`, `tukeywin`, `window`

References

[1] IEEE. *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979. Program 5.2.

Purpose

Chebyshev Type I filter design (passband ripple)

Syntax

```
[b,a] = cheby1(n,Rp,Wn)
[b,a] = cheby1(n,Rp,Wn,'ftype')
[b,a] = cheby1(n,Rp,Wn,'s')
[b,a] = cheby1(n,Rp,Wn,'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,Rp,Wn) designs an order n Chebyshev lowpass digital Chebyshev filter with cutoff frequency Wn and Rp 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.

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

Cutoff frequency is the frequency at which the magnitude response of the filter is equal to -Rp dB. For cheby1, the cutoff frequency Wn is a number between 0 and 1, where 1 corresponds to the Nyquist frequency, π radians per sample. Smaller values of passband ripple Rp lead to wider transition widths (shallower rolloff characteristics).

If Wn is a two-element vector, Wn = [w1 w2], cheby1 returns an order 2*n bandpass filter with passband w1 < ω < w2.

`[b,a] = cheby1(n,Rp,Wn,'ftype')` designs a highpass or bandstop filter, where the string `'ftype'` is either:

- `'high'` for a highpass digital filter with 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` < ω < `w2`.

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,Rp,Wn)` or

`[z,p,k] = cheby1(n,Rp,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] = cheby1(n,Rp,Wn)` or

`[A,B,C,D] = cheby1(n,Rp,Wn,'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] = cheby1(n,Rp,Wn,'s')` designs an order `n` lowpass analog Chebyshev Type I filter with cutoff frequency `Wn 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

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{s^n + a(2)s^{n-1} + \cdots + a(n+1)}$$

Cutoff frequency is the frequency at which the magnitude response of the filter is `-Rp` dB. For `cheby1`, the cutoff frequency `Wn` must be greater than 0 rad/s.

If Wn is a two-element vector Wn = [w1 w2] with w1 < w2, then cheby1(n,Rp,Wn,'**s**') returns an order 2*n bandpass analog filter with passband w1 < ω < w2.

[b,a] = cheby1(n,Rp,Wn,'*ftype*','**s**') designs a highpass 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,Rp,Wn,'**s**') or

[z,p,k] = cheby1(n,Rp,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] = cheby1(n,Rp,Wn,'**s**') or

[A,B,C,D] = cheby1(n,Rp,Wn,'*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

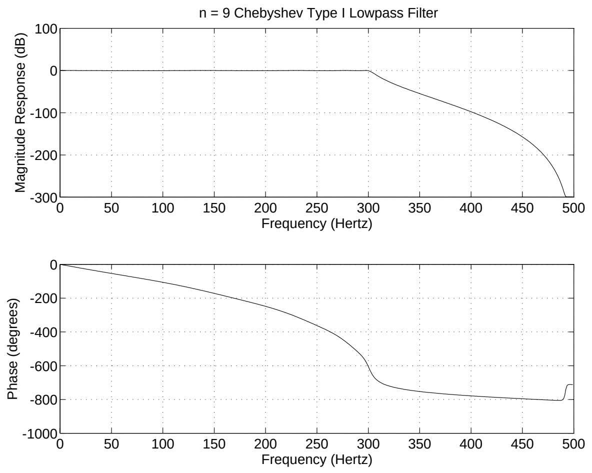
Example 1: Lowpass Filter

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 cutoff frequency of 300 Hz.

```
[b,a] = cheby1(9,0.5,300/500);
```

The frequency response of the filter is

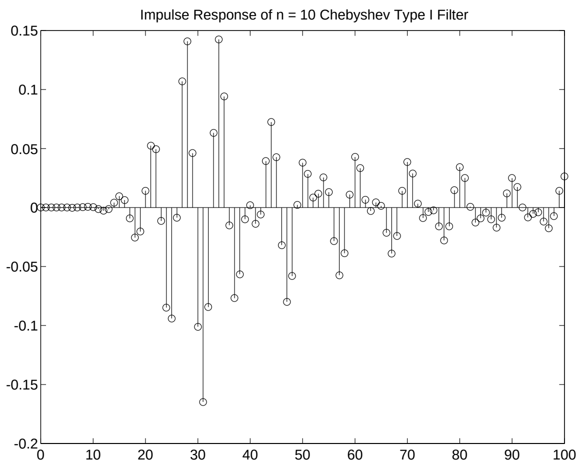
```
freqz(b,a,512,1000)
```



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; Rp = 0.5;
Wn = [100 200]/500;
[b,a] = cheby1(n,Rp,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 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 `Wn` or `w1` and `w2`.
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`

Purpose

Chebyshev Type II filter design (stopband ripple)

Syntax

```
[b,a] = cheby2(n,Rs,Wn)
[b,a] = cheby2(n,Rs,Wn,'ftype')
[b,a] = cheby2(n,Rs,Wn,'s')
[b,a] = cheby2(n,Rs,Wn,'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,Rs,Wn)` designs an order `n` lowpass digital Chebyshev Type II filter with cutoff frequency `Wn` and stopband ripple `Rs` 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.

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

Cutoff frequency is the beginning of the stopband, where the magnitude response of the filter is equal to `-Rs` dB. For `cheby2`, the normalized cutoff frequency `Wn` is a number between 0 and 1, where 1 corresponds to the Nyquist frequency. Larger values of stopband attenuation `Rs` lead to wider transition widths (shallower rolloff characteristics).

If `Wn` is a two-element vector, `Wn = [w1 w2]`, `cheby2` returns an order `2*n` bandpass filter with passband `w1` < ω < `w2`.

`[b,a] = cheby2(n,Rs,Wn,'ftype')` designs a highpass or bandstop filter, where the string `'ftype'` is either:

- `'high'` for a highpass digital filter with 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` < ω < `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,Rs,Wn)` or

`[z,p,k] = cheby2(n,Rs,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] = cheby2(n,Rs,Wn)` or

`[A,B,C,D] = cheby2(n,Rs,Wn,'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,Rs,Wn,'s')` designs an order n lowpass analog Chebyshev Type II filter with cutoff frequency Wn. 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.

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{s^n + a(2)s^{n-1} + \cdots + a(n+1)}$$

Cutoff frequency is the frequency at which the magnitude response of the filter is equal to -Rs dB. For cheby2, the cutoff frequency Wn must be greater than 0 rad/s.

If Wn is a two-element vector Wn = [w1 w2] with w1 < w2, then `cheby2(n,Rs,Wn,'s')` returns an order 2*n bandpass analog filter with passband w1 < ω < w2.

`[b,a] = cheby2(n,Rs,Wn,'ftype','s')` designs a highpass or bandstop 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,Rs,Wn,'s')` or

`[z,p,k] = cheby2(n,Rs,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] = cheby2(n,Rs,Wn,'s')` or

`[A,B,C,D] = cheby2(n,Rs,Wn,'ftype','s')` 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

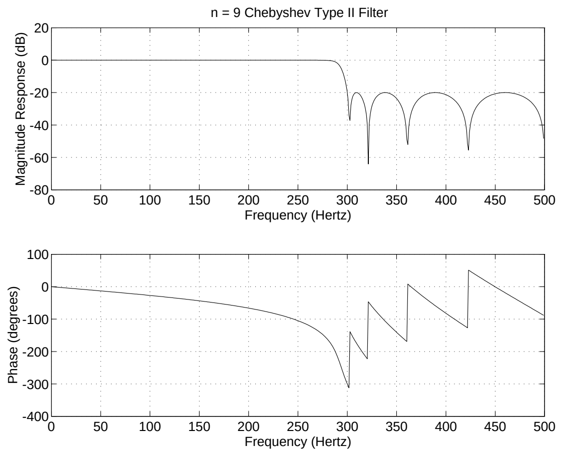
Example 1: Lowpass Filter

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 cutoff frequency of 300 Hz.

```
[b,a] = cheby2(9,20,300/500);
```

The frequency response of the filter is

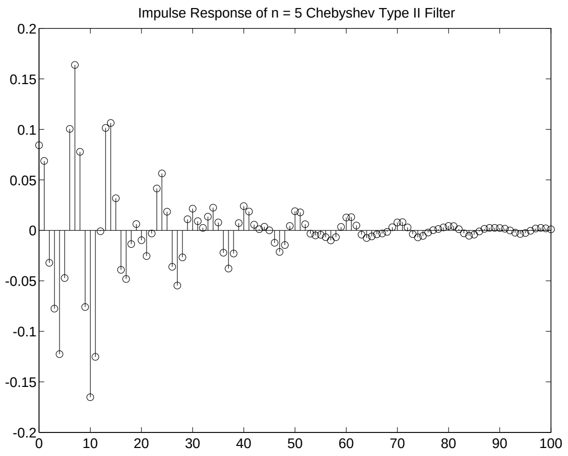
```
freqz(b,a,512,1000)
```



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;
Wn = [100 200]/500;
[b,a] = cheby2(n,r,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

`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 `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 `Wn` or `w1` and `w2`.
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

Generate a 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 `0`, `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

  $$\beta = (f_1 - f_0)/t_1$$

  β 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

  $$\beta = (f_1 - f_0)/t_1^2$$

  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

  $$f_i(t) = f_0 + 10^{\beta t}$$

  where

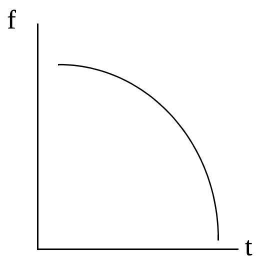
$$\beta = [\log_{10}(f_1 - f_0)] / t_1$$

For a log-sweep, `f1` must be greater than `f0`.

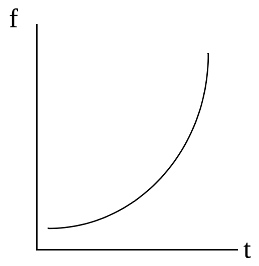
Each of the above methods can be entered as `'li'`, `'q'`, and `'lo'`, respectively.

`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



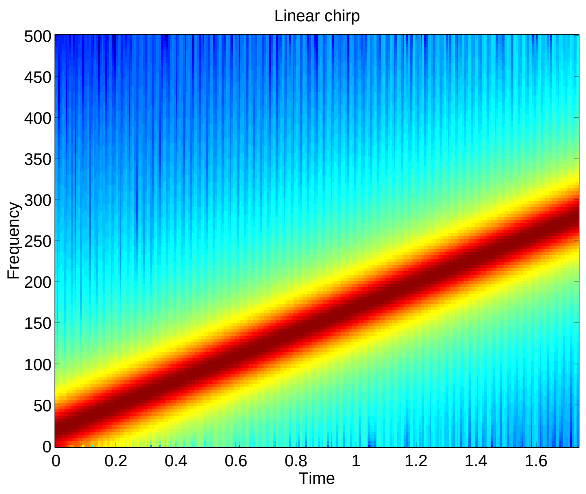
Concave upsweep shape

Examples

Example 1

Compute the spectrogram of a chirp with linear instantaneous frequency deviation.

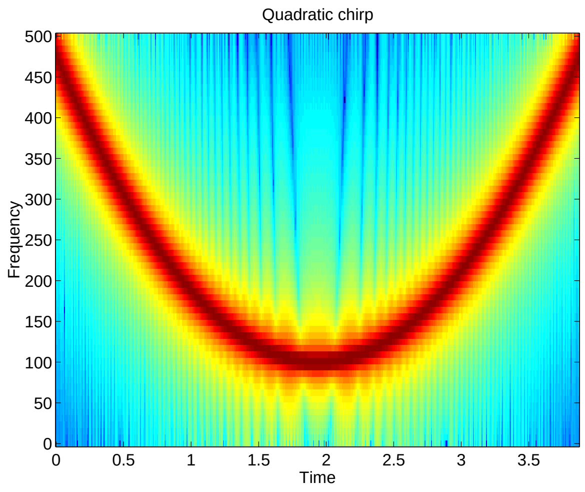
```
t = 0:0.001:2;                 % 2 secs @ 1kHz sample rate
y = chirp(t,0,1,150);          % Start @ DC, cross 150Hz at t=1 sec
specgram(y,256,1e3,256,250) % Display the spectrogram
```



Example 2

Compute the spectrogram of a chirp with quadratic instantaneous frequency deviation.

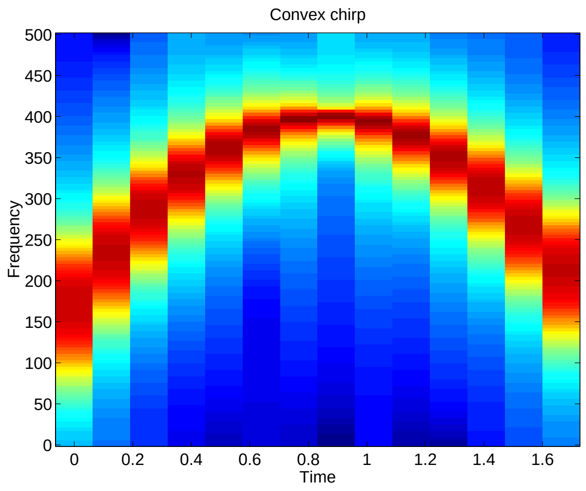
```
t = -2:0.001:2;                      % ±2 secs @ 1kHz sample rate
y = chirp(t,100,1,200,'quadratic'); % Start @ 100Hz, cross 200Hz
                                     %   at t=1 sec
specgram(y,128,1e3,128,120)          % Display the spectrogram
```



Example 3

Compute the spectrogram of a convex quadratic chirp.

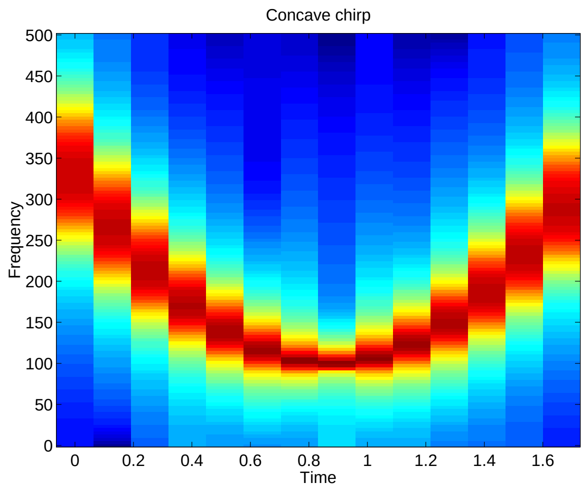
```
t= -1:0.001:1;                      % +/-1 second @ 1kHz sample rate
fo=100; f1=400;                     % Start at 100Hz, go up to 400Hz
y=chirp(t,fo,1,f1,'q',[],'convex');
specgram(y,256,1000)                % Display the spectrogram.
```



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');
specgram(y,256,1000)                % Display the spectrogram.
```



See Also

cos, diric, gauspuls, pulstran, rectpuls, sawtooth, sin, sinc, square, tripuls

cohere

Purpose Estimate magnitude squared coherence function between two signals

Syntax

```
Cxy = cohere(x,y)
Cxy = cohere(x,y,nfft)
[Cxy,f] = cohere(x,y,nfft,fs)
Cxy = cohere(x,y,nfft,fs,window)
Cxy = cohere(x,y,nfft,fs,window,numoverlap)
Cxy = cohere(x,y,...,'dflag')
cohere(x,y)
```

Description

`Cxy = cohere(x,y)` finds the magnitude squared coherence between length `n` signal vectors `x` and `y`. The coherence is a function of the power spectra of `x` and `y` and the cross spectrum 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.

`nfft` specifies the FFT length that `cohere` uses. This value determines the frequencies at which the coherence is estimated. `fs` is a scalar that specifies the sampling frequency. `window` specifies a windowing function and the number of samples `cohere` uses in its sectioning of the `x` and `y` vectors. `numoverlap` is the number of samples by which the window sections overlap for both `x` and `y`. Any arguments that you omit from the end of the parameter list use the default values shown below.

`Cxy = cohere(x,y)` uses the following default values:

- `nfft = min(256,length(x))`
- `fs = 2`
- `window` is a periodic Hann window of length `nfft`.
- `numoverlap = 0`

If `x` is real, `cohere` estimates the coherence function at positive frequencies only; in this case, the output `Cxy` is a column vector of length `nfft/2 + 1` for `nfft` even and `(nfft + 1)/2` for `n` odd. If `x` or `y` is complex, `cohere` estimates the coherence function at both positive and negative frequencies, and `Cxy` has length `nfft`.

`Cxy = cohere(x,y,nfft)` uses the FFT length `nfft` in estimating the power spectrum for `x`. Specify `nfft` as a power of 2 for fastest execution.

`Cxy = cohere(x,y,nfft,fs)` returns a vector `f` of frequencies at which the function evaluates the coherence. `fs` is the sampling frequency. `f` is the same size as `Cxy`, so `plot(f,Cxy)` plots the coherence function versus properly scaled frequency. `fs` has no effect on the output `Cxy`; it is a frequency scaling multiplier.

`cohere(x,y,nfft,fs,window)` specifies a windowing function and the number of samples per section of the vectors `x` and `y`. If you supply a scalar for `window`, `cohere` uses a Hann window of that length. The length of the window must be less than or equal to `nfft`; `cohere` zero pads the sections if the window length exceeds `nfft`.

`cohere(x,y,nfft,fs,window,numoverlap)` overlaps the sections of `x` by `numoverlap` samples.

Note If you use `cohere` on two linearly related signals [1] with a single, non-overlapping window, the output for all frequencies is `Cxy` = 1.

You can use the empty matrix `[]` to specify the default value for any input argument except `x` or `y`. For example,

```
cohere(x,y,[],[],kaiser(128,5));
```

uses `256` as the value for `nfft` and `2` as the value for `fs`.

`cohere(x,y,...,'`*`dflag`*`')` specifies a detrend option, where `'`*`dflag`*`'` is:

- `'linear'`, to remove the best straight-line fit from the prewindowed sections of `x` and `y`
- `'mean'`, to remove the mean from the prewindowed sections of `x` and `y`
- `'none'`, for no detrending (default)

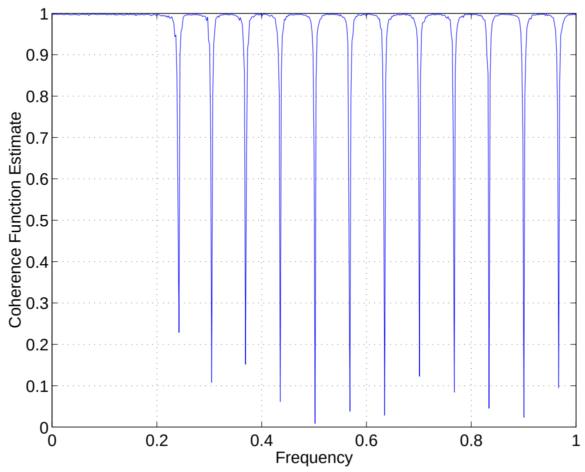
The `'`*`dflag`*`'` parameter must appear last in the list of input arguments. `cohere` recognizes a `'`*`dflag`*`'` string no matter how many intermediate arguments are omitted.

cohere with no output arguments plots the coherence estimate versus frequency in the current figure window.

Example

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);
cohere(x,y,1024,[],[],512)
```



Diagnostics

An appropriate diagnostic message is displayed when incorrect arguments are used.

```
Requires window's length to be no greater than the FFT length.
Requires NOVERLAP to be strictly less than the window length.
Requires positive integer values for NFFT and NOVERLAP.
Requires vector (either row or column) input.
Requires inputs X and Y to have the same length.
```

Algorithm

`cohere` estimates the magnitude squared coherence function [2] using Welch's method of power spectrum estimation (see references [3] and [4]), as follows:

1. It divides the signals `x` and `y` into separate overlapping sections, detrends each section, and multiplies each section by `window`.
2. It calculates the length `nfft` fast Fourier transform of each section.
3. It averages the squares of the spectra of the `x` sections to form `Pxx`, averages the squares of the spectra of the `y` sections to form `Pyy`, and averages the products of the spectra of the `x` and `y` sections to form `Pxy`. It calculates `Cxy` by the following formula.

   ```
   Cxy = abs(Pxy).^2/(Pxx.*Pyy)
   ```

See Also

`csd`, `pwelch`, `tfe`

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.

Purpose Convolution and polynomial multiplication

Syntax

```
c = conv(a,b)
```

Description `c = conv(a,b)` convolves vectors `a` and `b`. The convolution sum is

$$c(n+1) = \sum_{k=0}^{N-1} a(k+1)b(n-k)$$

where N is the maximum sequence length. The series is indexed from $n + 1$ and $k + 1$ instead of the usual n and k because MATLAB vectors run from 1 to n instead of from 0 to n-1.

The `conv` function is part of the standard MATLAB language.

Example The convolution of `a = [1 2 3]` and `b = [4 5 6]` is

```
c = conv(a,b)

c =
     4    13    28    27    18
```

Algorithm The `conv` function is an M-file that uses the `filter` primitive. `conv` computes the convolution operation as FIR filtering with an appropriate number of zeros appended to the input.

See Also `conv2`, `convmtx`, `convn`, `deconv`, `filter`, `residuez`, `xcorr`

conv2

Purpose

Two-dimensional convolution

Syntax

```
C = conv2(A,B)
C = conv2(A,B,'shape')
```

Description

`C = conv2(A,B)` computes the two-dimensional convolution of matrices `A` and `B`. If one of these matrices describes a two-dimensional FIR filter, the other matrix is filtered in two dimensions.

Each dimension of the output matrix `C` is equal in size to one less than the sum of the corresponding dimensions of the input matrices. When `[ma,na] = size(A)` and `[mb,nb] = size(B)`,

```
size(C) = [ma+mb-1,na+nb-1]
```

`C = conv2(A,B,'`*`shape`*`')` returns a subsection of the two-dimensional convolution with size specified by `'`*`shape`*`'`, where:

- `'full'` returns the full two-dimensional convolution (default)
- `'same'` returns the central part of the convolution that is the same size as `A`
- `'valid'` returns only those parts of the convolution that are computed without the zero-padded edges. Using this option, `size(C) = [ma-mb+1,na-nb+1]` when `size(A) > size(B)`

`conv2` executes most quickly when `size(A) > size(B)`.

The `conv2` function is part of the standard MATLAB language.

Examples

In image processing, the Sobel edge-finding operation is a two-dimensional convolution of an input array with the special matrix

```
s = [1 2 1; 0 0 0; -1 -2 -1];
```

Given any image, the following code extracts the horizontal edges.

```
h = conv2(I,s);
```

The following code extracts the vertical edges first, and then extracts both horizontal and vertical edges combined.

```
v = conv2(I,s');
v2 = (sqrt(h.^2 + v.^2))
```

See Also `conv`, `convn`, `deconv`, `filter2`, `xcorr`, `xcorr2`

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`.

Example

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`

Purpose

Compute the correlation coefficient matrix

Syntax

```
C = corrcoef(X)
C = corrcoef(X,Y)
```

Description

`corrcoef` returns a matrix of correlation coefficients calculated from an input matrix whose rows are observations and whose columns are variables. If `C = cov(X)`, then `corrcoef(X)` is the matrix whose element (i, j) is

$$\text{corrcoef}(i, j) = \frac{C(i, j)}{\sqrt{C(i, i)C(j, j)}}$$

`C = corrcoef(X)` is the zeroth lag of the covariance function, that is, the zeroth lag of `xcov(x,'coeff')` packed into a square array.

`C = corrcoef(X,Y)` is the same as `corrcoef([X Y])`; that is, it concatenates `X` and `Y` in the row direction before its computation.

`corrcoef` removes the mean from each column before calculating the results. See the `xcorr` function for cross-correlation options.

The `corrcoef` function is part of the standard MATLAB language.

See Also

`cov`, `mean`, `median`, `std`, `xcorr`, `xcov`

Purpose Compute a 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.

$$X = \left[\begin{array}{ccc} x(1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ \hline x(m+1) & \cdots & x1) \\ \vdots & \ddots & \vdots \\ x(n-m) & \cdots & x(m+1) \\ \vdots & \ddots & \vdots \\ x(n) & \cdots & x(n-m) \\ \hline \vdots & \ddots & \vdots \\ 0 & \cdots & x(n) \end{array}\right]$$

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_{mod} shown below.

$$X_{\mathrm{mod}} = \begin{bmatrix} x(m+1) & \cdots & x1) \\ \vdots & \ddots & \vdots \\ x(n-m) & \cdots & x(m+1) \\ \vdots & \ddots & \vdots \\ x(n) & \cdots & x(n-m) \\ x^*(1) & \cdots & x^*(m+1) \\ \vdots & \iddots & \vdots \\ x^*(m+1) & \cdots & x^*(n-m) \\ \vdots & \iddots & \vdots \\ x^*(n-m) & \cdots & x^*(n) \end{bmatrix}$$

See Also `peig`, `pmusic`, `rooteig`, `rootmusic`, `xcorr`

References

[1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, pp. 216-223.

Purpose Compute the covariance matrix

Syntax

```
c = cov(x)
c = cov(x,y)
```

Description `cov` computes the covariance matrix. If `x` is a vector, `c` is a scalar containing the variance. For an array where each row is an observation and each column a variable, `cov(X)` is the covariance matrix. `diag(cov(X))` is a vector of variances for each column, and `sqrt(diag(cov(X)))` is a vector of standard deviations.

`cov(x)` is the zeroth lag of the covariance function, that is, the zeroth lag of `xcov(x)/(n-1)` packed into a square array.

`cov(x,y)` where `x` and `y` are column vectors of equal length is equivalent to `cov([x y])`, that is, it concatenates `x` and `y` in the row direction before its computation.

`cov` removes the mean from each column before calculating the results.

The `cov` function is part of the standard MATLAB language.

Algorithm

```
[n,p] = size(x);
x = x-ones(n,1)*(sum(x)/n);
y = x'*x/(n-1);
```

See Also `corrcoef`, `mean`, `median`, `std`, `xcorr`, `xcov`

cplxpair

Purpose

Group complex numbers into complex conjugate pairs

Syntax

```
y = cplxpair(x)
y = cplxpair(x,tol)
```

Description

`y = cplxpair(x)` returns `x` with complex conjugate pairs grouped together. `cplxpair` orders the conjugate pairs by increasing real part. Within a pair, the element with negative imaginary part comes first. The function returns all purely real values following all the complex pairs.

`y = cplxpair(x,tol)` includes a tolerance, `tol`, for determining which numbers are real and which are paired complex conjugates. By default, `cplxpair` uses a tolerance of `100*eps` relative to `abs(x(i))`. `cplxpair` forces the complex conjugate pairs to be exact complex conjugates.

The `cplxpair` function is part of the standard MATLAB language.

Example

Order five poles evenly spaced around the unit circle into complex pairs.

```
cplxpair(exp(2*pi*sqrt(-1)*(0:4)/5)')

ans =
    -0.8090 - 0.5878i
    -0.8090 + 0.5878i
     0.3090 - 0.9511i
     0.3090 + 0.9511i
     1.0000
```

Diagnostics

If there is an odd number of complex numbers, or if the complex numbers cannot be grouped into complex conjugate pairs within the tolerance, `cplxpair` generates the error message.

```
Complex numbers can't be paired.
```

Purpose

Complex and nonlinear-phase equiripple FIR filter design

Syntax

```
b = cremez(n,f,'fresp')
b = cremez(n,f,'fresp',w)
b = cremez(n,f,{'fresp',p1,p2,...},w)
b = cremez(n,f,a,w)
b = cremez(...,'sym')
b = cremez(...,'skip_stage2')
b = cremez(...,'debug')
b = cremez(...,{lgrid})
[b,delta,opt] = cremez(...)
```

Description

`cremez` 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 = cremez(n,f,'`*`fresp`*`')` returns a length `n+1` FIR filter with the best approximation to the desired frequency response as returned by function `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.

`b = cremez(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 = cremez(n,f,{'`*`fresp`*`',p1,p2,...},...)` supplies optional parameters `p1, p2, ...`, to the frequency response function *`fresp`*. 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`, `bandpass`, `bandstop`

  These functions share a common syntax, exemplified below by the string `'lowpass'`.

  `b = cremez(n,f,'lowpass',...)` and

  `b = cremez(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 = cremez(n,f,{'multiband',a},...)` and

  `b = cremez(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 = cremez(n,f,{'differentiator',fs},...)` and

  `b = cremez(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 = cremez(n,f,'hilbfilt',...)` and

  `b = cremez(N,F,{'hilbfilt',d},...)` design a linear-phase (`n/2+d` delay) Hilbert transform filter.

`b = cremez(n,f,a,w)` is a synonym for
`b = cremez(n,f,{'multiband',a},w)`.

`b = cremez(...,'`*`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, bandpass, bandstop, 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 only be specified 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.

`b = cremez(...,'`**`skip_stage2`**`')` disables the second-stage optimization algorithm, which executes only when `cremez` determines that an optimal solution has not been reached by the standard Remez 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 = cremez(...,'`*`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 = cremez(...,{lgrid})` uses the integer `lgrid` to control the density of the frequency grid, which has roughly `2^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] = cremez(...)` returns the maximum ripple height `delta`.

`[b,delta,opt] = cremez(...)` returns a structure `opt` of optional results computed by `cremez` and contains the following fields.

`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 `cremez` using the following syntax

```
[dh,dw] = fresp(n,f,gf,w,p1,p2,...)
```

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 `cremez`. `gf` determines the frequency grid at which the response function must be evaluated. This is the same data returned by `cremez` 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 `cremez`; 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`, `...`, 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,...})
```

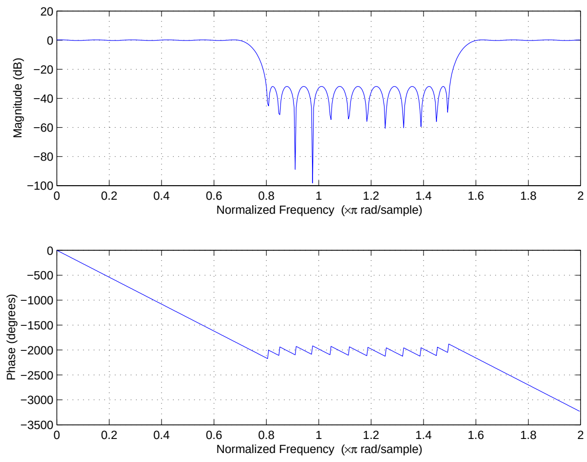
The arguments may be used in determining an appropriate symmetry default as necessary. The function `private/lowpass.m` may be useful as a template for generating new frequency response functions.

Examples

Example 1

Design a 31-tap, linear-phase, lowpass filter.

```
b = cremez(30,[-1 -0.5 -0.4 0.7 0.8 1],'lowpass');
freqz(b,1,512,'whole');
```



Example 2

Design a nonlinear-phase allpass FIR filter.

First select (or create) the function *fresp* that returns the desired frequency response. For this example, *fresp* is the `allpass.m` function in the `signal/signal/private` directory which returns the frequency response of a nonlinear-phase allpass filter. Copy `allpass.m` to another location on the MATLAB path before trying the example.

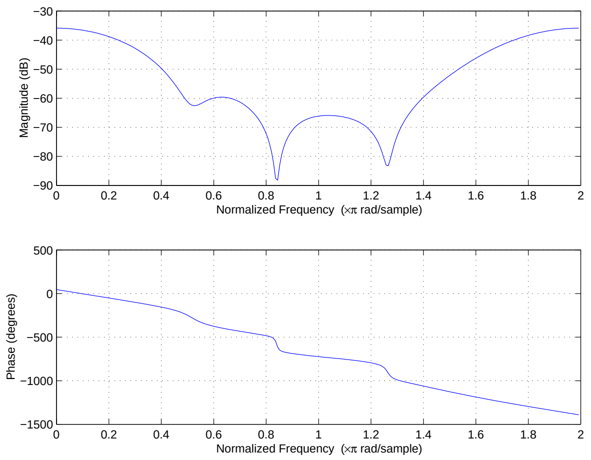
Before using `cremez` with `allpass.m` to generate the filter coefficients, call `allpass` alone to create the *desired* response.

```
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 = allpass(n,f,gf,w);        % Desired frequency response
```

Vector `d` now contains the complex frequency response that we desire for the FIR filter computed by `cremez`.

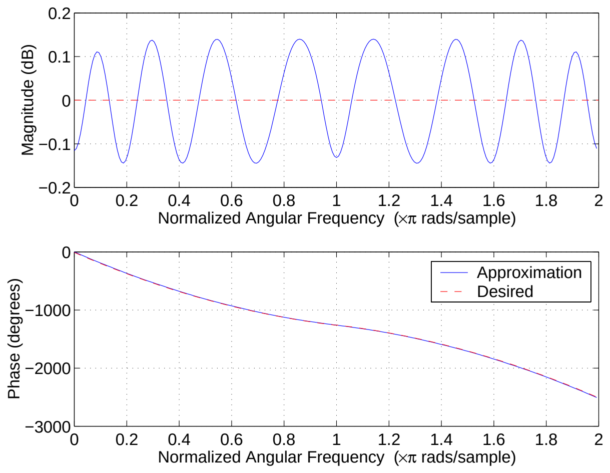
Now compute the FIR filter that best approximates this response.

```
b = cremez(n,f,'allpass',w,'real'); % Approximation
freqz(b,1,256,'whole');
```



The `freqz` plot shows the frequency response of the filter computed by `cremez` to approximate the desired response. Check the accuracy of the approximation by overlaying the *desired* frequency response on the plot.

```
subplot(2,1,1); hold on
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.

See Also

`fir1`, `fir2`, `firls`, `remez`

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.

[3] Demjanjov, V.F., and V.N. Malozemov. *Introduction to Minimax*. New York: John Wiley & Sons, 1974.

Purpose

Estimate the cross spectral density (CSD) of two signals

Syntax

```
Pxy = csd(x,y)
Pxy = csd(x,y,nfft)
[Pxy,f] = csd(x,y,nfft,fs)
Pxy = csd(x,y,nfft,fs,window)
Pxy = csd(x,y,nfft,fs,window,numoverlap)
Pxy = csd(x,y,...,'dflag')
[Pxy,Pxyc,f] = csd(x,y,nfft,fs,window,numoverlap,p)
csd(x,y,...)
```

Description

`Pxy = csd(x,y)` estimates the cross spectral density of the length `n` sequences `x` and `y` using the Welch method of spectral estimation. `Pxy = csd(x,y)` uses the following default values:

- `nfft = min(256,length(x))`
- `fs = 2`
- `window` is a periodic Hann window of length `nfft`.
- `numoverlap = 0`

`nfft` specifies the FFT length that `csd` uses. This value determines the frequencies at which the cross spectrum is estimated. `fs` is a scalar that specifies the sampling frequency. `window` specifies a windowing function and the number of samples `csd` uses in its sectioning of the `x` and `y` vectors. `numoverlap` is the number of samples by which the sections overlap. Any arguments omitted from the end of the parameter list use the default values shown above.

If `x` and `y` are real, `csd` estimates the cross spectral density at positive frequencies only; in this case, the output `Pxy` 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, `csd` estimates the cross spectral density at both positive and negative frequencies and `Pxy` has length `nfft`.

`Pxy = csd(x,y,nfft)` uses the FFT length `nfft` in estimating the cross spectral density of `x` and `y`. Specify `nfft` as a power of 2 for fastest execution.

`[Pxy,f] = csd(x,y,nfft,fs)` returns a vector `f` of frequencies at which the function evaluates the CSD. `f` is the same size as `Pxy`, so `plot(f,Pxy)` plots the spectrum versus properly scaled frequency. `fs` has no effect on the output `Pxy`; it is a frequency scaling multiplier.

`Pxy = csd(x,y,nfft,fs,window)` specifies a windowing function and the number of samples per section of the `x` vector. If you supply a scalar for `window`, `csd` uses a Hann window of that length. The length of the window must be less than or equal to `nfft`; `csd` zero pads the sections if the length of the window is less than `nfft`. `csd` returns an error if the length of the window is greater than `nfft`.

`Pxy = csd(x,y,nfft,fs,window,numoverlap)` overlaps the sections of `x` and `y` by `numoverlap` samples.

You can use the empty matrix `[]` to specify the default value for any input argument except `x` or `y`. For example,

```
csd(x,y,[],10000)
```

is equivalent to

```
csd(x)
```

but with a sampling frequency of 10,000 Hz instead of the default of 2 Hz.

`Pxy = csd(x,y,...,'`*`dflag`*`')` specifies a detrend option, where the string `'`*`dflag`*`'` is one of the following:

- `'linear'`, to remove the best straight-line fit from the prewindowed sections of `x` and `y`
- `'mean'`, to remove the mean from the prewindowed sections of `x` and `y`
- `'none'`, for no detrending (default)

The *`dflag`* parameter must appear last in the list of input arguments. `csd` recognizes a *`dflag`* string no matter how many intermediate arguments are omitted.

`[Pxy,Pxyc,f] = csd(x,y,nfft,fs,window,numoverlap,p)` where `p` is a positive scalar between 0 and 1 returns a vector `Pxyc` that contains an estimate of the `p*100` percent confidence interval for `Pxy`. `Pxyc` is a two-column matrix the same length as `Pxy`. The interval `[Pxyc(:,1),Pxyc(:,2)]` covers the true

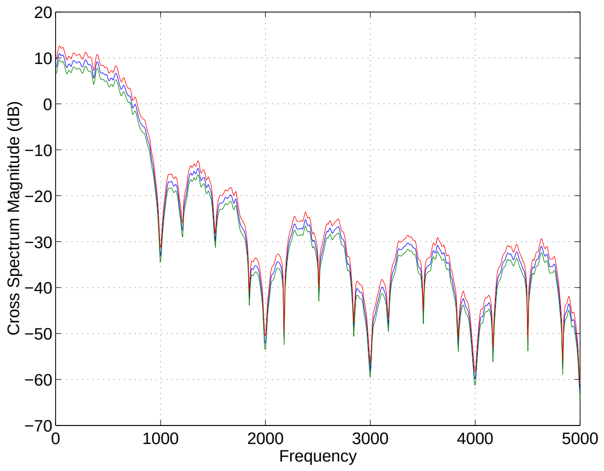
CSD with probability p. plot(f,[Pxy Pxyc]) plots the cross spectrum inside the p*100 percent confidence interval. If unspecified, p defaults to 0.95.

csd(x,y,...) plots the CSD versus frequency in the current figure window. If the p parameter is specified, the plot includes the confidence interval.

Example

Generate two colored noise signals and plot their CSD 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);
csd(x,y,1024,10000,triang(500),0,[])
```



Algorithm

csd implements the Welch method of spectral density estimation (see references [1] and [2]):

1 It applies the window specified by the window vector to each successive detrended section.

2 It transforms each section with an nfft-point FFT.

3 It forms the periodogram of each section by scaling the product of the transform of the `y` section and the conjugate of the transformed `x` section.

4 It averages the periodograms of the successive overlapping sections to form `Pxy`, the cross spectral density of `x` and `y`.

The number of sections that `csd` averages is `k`, where `k` is

```
fix((length(x)-numoverlap)/(length(window)-numoverlap))
```

Diagnostics

An appropriate diagnostic message is displayed when incorrect arguments to `csd` are used.

```
Requires window's length to be no greater than the FFT length.
Requires NOVERLAP to be strictly less than the window length.
Requires positive integer values for NFFT and NOVERLAP.
Requires vector (either row or column) input.
Requires inputs X and Y to have the same length.
Requires confidence parameter to be a scalar between 0 and 1.
```

See Also

`cohere`, `pburg`, `pmtm`, `pmusic`, `pwelch`, `pyulear`, `tfe`

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. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 737.

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*(w.^-(0:m-1))
```

`y = czt(x)` uses the following default values:

- `m = length(x)`
- `w = exp(j*2*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.

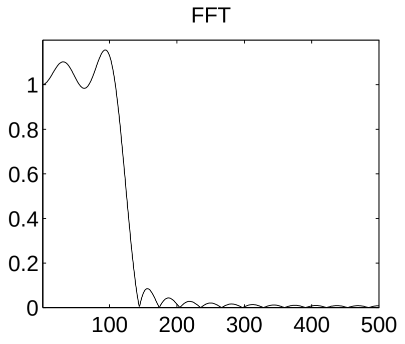
```
fs = 1000; f1 = 100; f2 = 150;  % in hertz
m = 1024;
w = exp(-j*2*pi*(f2-f1)/(m*fs));
a = exp(j*2*pi*f1/fs);
```

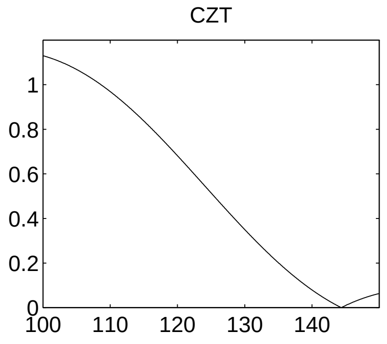
Compute both the DFT and CZT of the filter.

```
y = fft(h,1000);
z = czt(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

`czt` 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, `czt` gives the following error message.

```
Inputs M, W, and A must be scalars.
```

See Also

`fft`, `freqz`

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pgs. 393-399.

Purpose

Discrete cosine transform (DCT)

Syntax

```
y = dct(x)
y = dct(x,n)
```

Description

`y = dct(x)` returns the unitary discrete cosine transform of `x`

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

where

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

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 = 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.

Example

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
```

See Also

`fft`, `idct`, `dct2`, `idct2`

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.

decimate

Purpose

Decrease the sampling rate for a sequence (decimation)

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. 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.

`y = decimate(x,r,'fir')` uses a 30-point 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 a length `n` FIR filter.

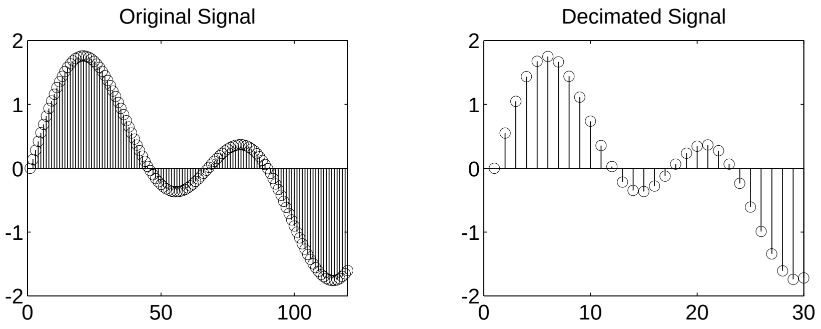
Example

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.
```

See Also

`downsample`, `interp`, `resample`, `spline`, `upfirdn`, `upsample`

References

[1] IEEE. *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979. Chapter 8.

Purpose

Deconvolution and polynomial division

Syntax

```
[q,r] = deconv(b,a)
```

Description

`[q,r] = deconv(b,a)` deconvolves vector `a` out of vector `b`, using long division. The result (quotient) is returned in vector `q` and the remainder in vector `r` such that `b = conv(q,a) + r`.

If `a` and `b` are vectors of polynomial coefficients, convolving them is equivalent to polynomial multiplication, and deconvolution is equivalent to polynomial division. The result of dividing `b` by `a` is quotient `q` and remainder `r`.

The `deconv` function is part of the standard MATLAB language.

Example

The convolution of `a = [1 2 3]` and `b = [4 5 6]` is

```
c = conv(a,b)
c =
     4    13    28    27    18
```

Use `deconv` to divide `b` back out.

```
[q,r] = deconv(c,a)
q =
     4     5     6
r =
     0     0     0     0     0
```

Algorithm

This function calls `filter` to compute the deconvolution as the impulse response of an IIR filter.

See Also

`conv`, `filter`, `residuez`

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`.)

`amdsb-sc` or `am` — **Amplitude demodulation, double sideband, suppressed carrier.** 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);
```

`amdsb-tc` — **Amplitude demodulation, double sideband, transmitted carrier.** 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);
```

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 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.

x = demod(y,fc,fs,'**pwm**','**centered**') finds the pulse widths assuming they are centered at the beginning of each period. x is length length(y)*fc/fs.

See Also

modulate, vco

dftmtx

Purpose

Discrete Fourier transform matrix

Syntax

```
A = dftmtx(n)
```

Description

A *discrete Fourier transform matrix* is a complex matrix of values around the unit circle, whose matrix product with a vector computes the discrete Fourier transform of the vector.

A = `dftmtx(n)` returns the `n`-by-`n` complex matrix `A` that, when multiplied into a length `n` column vector `x`.

```
y = A*x
```

computes the discrete Fourier transform of `x`.

The inverse discrete Fourier transform matrix is

```
Ai = conj(dftmtx(n))/n
```

Example

In practice, the discrete Fourier transform is computed more efficiently and uses less memory with an FFT algorithm

```
x = 1:256;
y1 = fft(x);
```

than by using the Fourier transform matrix.

```
n = length(x);
y2 = x*dftmtx(n);
norm(y1-y2)

ans =
    1.8297e-009
```

Algorithm

`dftmtx` takes the FFT of the identity matrix to generate the transform matrix.

See Also

`convmtx`, `fft`

Purpose

Compute the 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

$$\operatorname{diric}(x, n) = \begin{cases} -1^{\frac{x}{2\pi}(n-1)} & x = 0, \pm 2\pi, \pm 4\pi, \ldots \\ \dfrac{\sin(nx/2)}{n\sin(x/2)} & \text{else} \end{cases}$$

for any nonzero integer `n`. This function has period 2π for `n` odd and period 4π 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`

downsample

Purpose Decrease the sampling rate of the input signal

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`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
```

See Also decimate, interp, interp1, resample, spline, upfirdn, upsample

dpss

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*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 $1 \le k \le n$.

`[e,v] = dpss(n,nw,[k1 k2])` returns the `k1`st through the `k2`nd discrete prolate spheroidal sequences, where $1 \le k1 \le k2 \le n$.

For all of the above forms:

- The Slepian sequences are calculated directly.
- The sequences are generated in the frequency band $|\omega| \le (2\pi W)$, where `W = nw/n` is the half-bandwidth and ω is in rad/sample.
- `e(:,1)` is the length `n` signal most concentrated in the frequency band $|\omega| \le (2\pi W)$ 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');
```

See Also

dpssclear, dpssdir, dpssload, dpsssave, pmtm

References

[1] Percival, D.B., and A.T. Walden. *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques*. Cambridge: Cambridge University Press, 1993.

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`

dpssdir

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 7-149.

See Also `dpss`, `dpssclear`, `dpssload`, `dpsssave`

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 7-149.

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, dpssave, and dpssdir" on page 7-149.

See Also

`dpss`, `dpssclear`, `dpssdir`, `dpssload`

Purpose

Elliptic (Cauer) filter design

Syntax

```
[b,a] = ellip(n,Rp,Rs,Wn)
[b,a] = ellip(n,Rp,Rs,Wn,'ftype')
[b,a] = ellip(n,Rp,Rs,Wn,'s')
[b,a] = ellip(n,Rp,Rs,Wn,'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,Wn)` designs an order `n` lowpass digital elliptic filter with cutoff frequency `Wn`, `Rp` dB of ripple in the passband, and a stopband `Rs` 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.

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

The *cutoff frequency* is the edge of the passband, at which the magnitude response of the filter is `-Rp` dB. For `ellip`, the cutoff frequency `Wn` is a number between 0 and 1, where 1 corresponds to half the sampling frequency (Nyquist frequency). Smaller values of passband ripple `Rp` and larger values of stopband attenuation `Rs` both lead to wider transition widths (shallower rolloff characteristics).

If `Wn` is a two-element vector, `Wn = [w1 w2]`, `ellip` returns an order `2*n` bandpass filter with passband `w1` < ω < `w2`.

`[b,a] = ellip(n,Rp,Rs,Wn,'ftype')` designs a highpass or bandstop filter, where the string `'ftype'` is either:

- `'high'` for a highpass digital filter with 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, `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,Wn)` or

`[z,p,k] = ellip(n,Rp,Rs,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] = ellip(n,Rp,Rs,Wn)` or

`[A,B,C,D] = ellip(n,Rp,Rs,Wn,'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,Wn,'s')` designs an order `n` lowpass analog elliptic filter with cutoff frequency `Wn` and returns the filter coefficients in the length `n+1` row vectors `b` and `a`, in descending powers of s, derived from the transfer function

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{s^n + a(2)s^{n-1} + \cdots + a(n+1)}$$

The *cutoff frequency* is the edge of the passband, at which the magnitude response of the filter is `-Rp` dB. For `ellip`, the cutoff frequency `Wn` must be greater than 0 rad/s.

If `Wn` is a two-element vector with `w1` < `w2`, then `ellip(n,Rp,Rs,Wn,'s')` returns an order `2*n` bandpass analog filter with passband `w1` < ω < `w2`.

`[b,a] = ellip(n,Rp,Rs,Wn,'ftype','s')` designs a highpass 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,Wn,'s')` or

`[z,p,k] = ellip(n,Rp,Rs,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] = ellip(n,Rp,Rs,Wn,'s')` or

`[A,B,C,D] = ellip(n,Rp,Rs,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

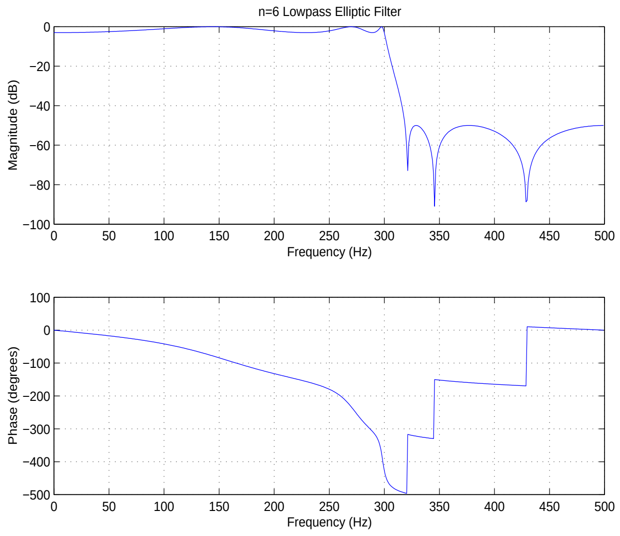
Example 1

For data sampled at 1000 Hz, design a sixth-order lowpass elliptic filter with a cutoff frequency of 300 Hz, 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

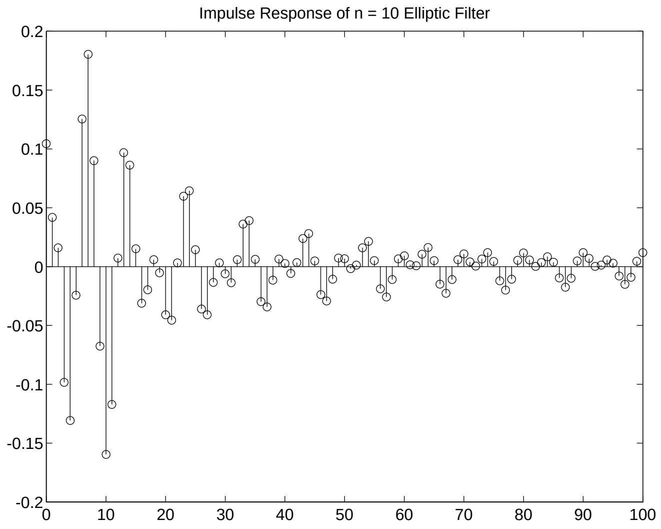
```
freqz(b,a,512,1000)
title('n=6 Lowpass Elliptic Filter')
```



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;
Wn = [100 200]/500;
[b,a] = ellip(n,Rp,Rs,Wn);
[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 `Wn` or `w1` and `w2`.
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

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 `Rp` dB of ripple in the passband, and a stopband `Rs` 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 cutoff frequency ω_0 of the elliptic filter to 1 for a normalized result. The *cutoff frequency* is the frequency at which the passband ends and the filter has a magnitude response of $10^{-Rp/20}$.

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.

See Also

`besselap`, `buttap`, `cheb1ap`, `cheb2ap`, `ellip`

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Chapter 7.

ellipord

Purpose

Calculate the minimum order for elliptic filters

Syntax

```
[n,Wn] = ellipord(Wp,Ws,Rp,Rs)
[n,Wn] = 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,Wn] = ellipord(Wp,Ws,Rp,Rs)` returns the lowest order `n` of the elliptic 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 `ellip`.

Choose the input arguments to specify the stopband and passband according to the following table.

Table 7-7: Description of Stopband and Passband Filter Parameters

`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, π 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. Twice this value specifies the maximum permissible passband width in decibels.
`Rs`	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.

Table 7-8: Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	`Wp` < `Ws`, both scalars	`(Ws,1)`	`(0,Wp)`
Highpass	`Wp` > `Ws`, both scalars	`(0,Ws)`	`(Wp,1)`
Bandpass	The interval specified by `Ws` contains the one specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`).	`(0,Ws(1))` and `(Ws(2),1)`	`(Wp(1),Wp(2))`
Bandstop	The interval specified by `Wp` contains the one specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`).	`(0,Wp(1))` and `(Wp(2),1)`	`(Ws(1),Ws(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,Wn] = ellipord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Wn` for an analog filter. You specify the frequencies `Wp` and `Ws` similar to Table 7-7, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

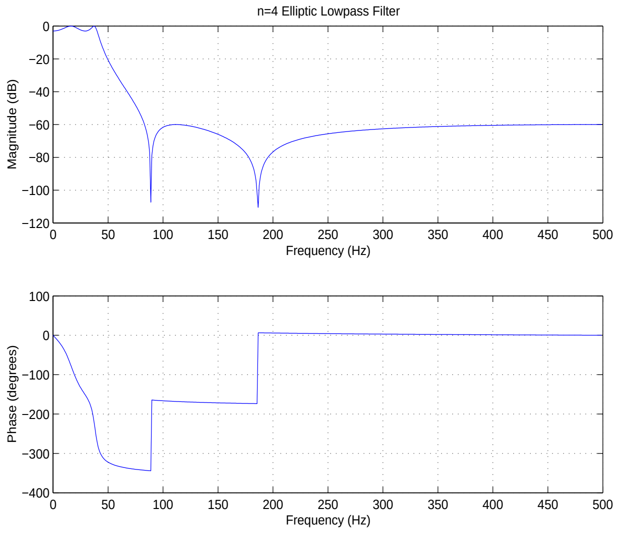
Use `ellipord` for lowpass, highpass, bandpass, and bandstop filters as described in Table 7-8.

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).

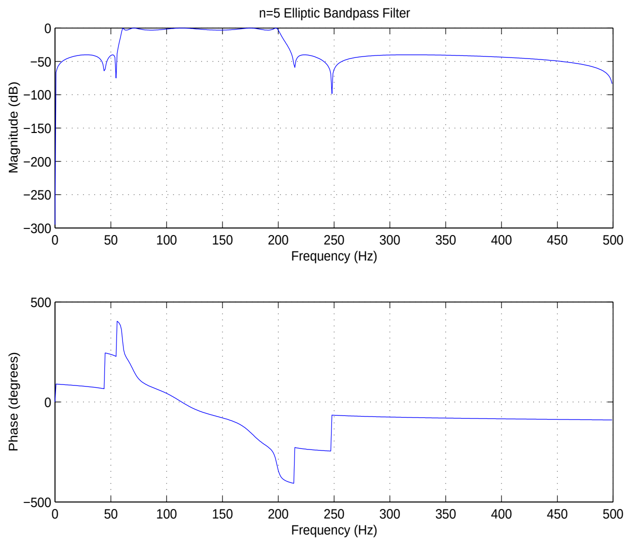
```
Wp = 40/500; Ws = 150/500;
Rp = 3; Rs = 60;
[n,Wn] = ellipord(Wp,Ws,Rp,Rs)

n =
     4

Wn =
    0.0800

[b,a] = ellip(n,Rp,Rs,Wn);
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,Wn] = ellipord(Wp,Ws,Rp,Rs)

n =
     5
Wn =
    0.1200    0.4000

[b,a] = ellip(n,Rp,Rs,Wn);
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.

See Also

`buttord`, `cheb1ord`, `cheb2ord`, `ellip`

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 241.

eqtflength

Purpose

Make the lengths of a transfer function's numerator and denominator equal

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`

fdatool

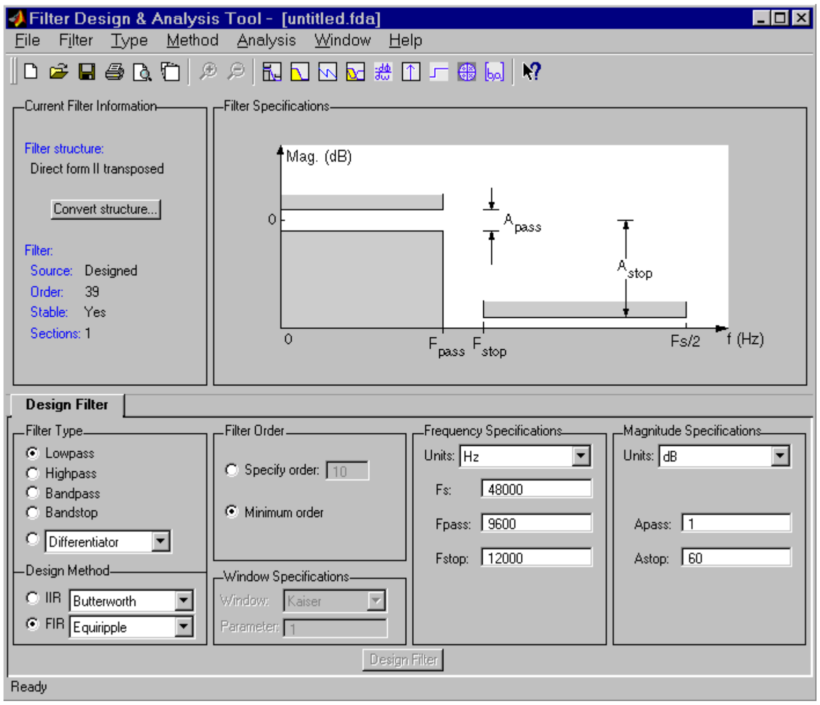
Purpose Open the Filter Design and Analysis Tool

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, "Filter Design and Analysis Tool," for more information.



Filter structure: Direct form II transposed

Filter:
Source: Designed
Order: 39
Stable: Yes
Sections: 1

Filter Type: [✓ Lowpass] Highpass Bandpass Bandstop Differentiator

Design Method: IIR Butterworth [✓ FIR Equiripple]

Filter Order: Specify order: 10 [✓ Minimum order]

Window Specifications: Window: Kaiser Parameter: 1

Frequency Specifications: Units: Hz Fs: 48000 Fpass: 9600 Fstop: 12000

Magnitude Specifications: Units: dB Apass: 1 Astop: 60

Remarks The Filter Design and Analysis Tool provides more design methods than the SPTool Filter Designer.

Note The Filter Design and Analysis Tool requires a screen resolution greater than 640 x 480.

See Also `sptool`, `fvtool`

Purpose Compute the one-dimensional fast Fourier transform

Syntax

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

Description `fft` computes the discrete Fourier transform of a vector or matrix. This function implements the transform given by

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

where $W_N = e^{-j(2\pi/N)}$ and N = `length(x)`. Note that the series is indexed as $n + 1$ and $k + 1$ instead of the usual n and k because MATLAB vectors run from 1 to N instead of from 0 to N-1.

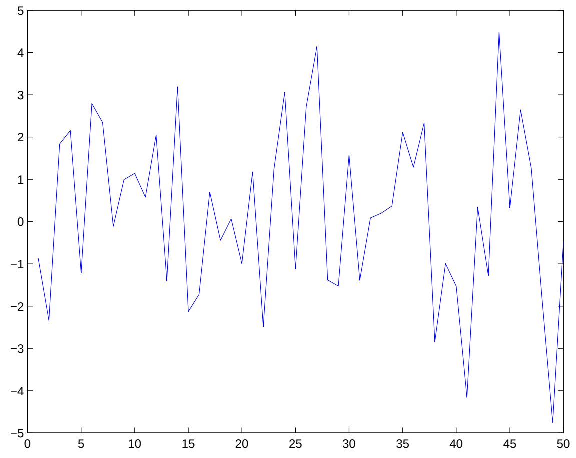
`y = fft(x)` is the discrete Fourier transform of vector `x`, computed with a fast Fourier transform (FFT) algorithm. If `x` is a matrix, `y` is the FFT of each column of the matrix.

`y = fft(x,n)` is the `n`-point FFT. If the length of `x` is less than `n`, `fft` pads `x` with trailing zeros to length `n`. If the length of `x` is greater than `n`, `fft` truncates the sequence `x`. If `x` is an array, `fft` adjusts the length of the columns in the same manner.

The `fft` function is part of the standard MATLAB language.

Examples A common use of the Fourier transform is to find the frequency components of a time-domain signal buried in noise. Consider data sampled at 1000 Hz. Form a signal consisting of 50 Hz and 120 Hz sinusoids and corrupt the signal with zero-mean random noise.

```
randn('state',0)
t = 0:0.001:0.6;
x = sin(2*pi*50*t) + sin(2*pi*120*t);
y = x + 2*randn(1,length(t));
plot(y(1:50))
```



It is difficult to identify the frequency components by studying the original signal. Convert to the frequency domain by taking the discrete Fourier transform of the noisy signal `y` using a 512-point fast Fourier transform (FFT).

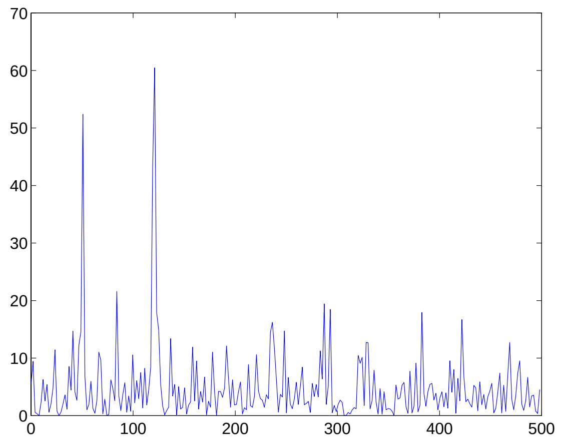
```
Y = fft(y,512);
```

The power spectral density, a measurement of the energy at various frequencies, is

```
Pyy = Y.*conj(Y) / 512;
```

Graph the first 256 points (the other 256 points are symmetric) on a meaningful frequency axis.

```
f = 1000*(0:255)/512;
plot(f,Pyy(1:256))
```



See the `pwelch` function for details on estimating the spectral density.

Sometimes it is useful to normalize the output of `fft` so that a unit sinusoid in the time domain corresponds to unit amplitude in the frequency domain. To produce a normalized discrete-time Fourier transform in this manner, use

```
Pn = abs(fft(x))*2/length(x)
```

Algorithm

`fft` is a built-in MATLAB function. See the `fft` reference page in the MATLAB documentation for details about the algorithm.

See Also

`dct`, `dftmtx`, `fft2`, `fftshift`, `filter`, `freqz`, `ifft`, `pwelch`

Purpose

Compute the two-dimensional fast Fourier transform

Syntax

```
Y = fft2(X)
Y = fft2(X,m,n)
```

Description

`Y = fft2(X)` performs a two-dimensional FFT, producing a result `Y` with the same size as `X`. If `X` is a vector, `Y` has the same orientation as `X`.

`Y = fft2(X,m,n)` truncates or zero pads `X`, if necessary, to create an `m`-by-`n` array before performing the FFT. The result `Y` is also `m`-by-`n`.

The `fft2` function is part of the standard MATLAB language.

Algorithm

`fft2(x)` is simply

```
fft(fft(x).').'
```

This computes the one-dimensional `fft` of each column of `x`, then of each row of the result. The time required to compute `fft2(x)` depends on the number of prime factors in `[m,n] = size(x)`. `fft2` is fastest when `m` and `n` are powers of 2.

See Also

`fft`, `fftshift`, `ifft`, `ifft2`

Purpose

FFT-based FIR filtering using the overlap-add method

Syntax

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

$$y(n) = b(1)x(n) + b(2)x(n-1) + \cdots + b(nb+1)x(n-nb)$$

An equivalent representation is the z-transform or *frequency domain* description

$$Y(z) = (b(1) + b(2)z^{-1} + \cdots + b(nb+1)z^{-nb})X(z)$$

By default, `fftfilt` chooses an FFT length and data block length that guarantee efficient execution time.

`y = fftfilt(b,x,n)` uses an FFT length of `nfft` = `2^nextpow2(n)` and a data block length of `nfft` - `length(b)` + `1`.

`fftfilt` works for both real and complex inputs.

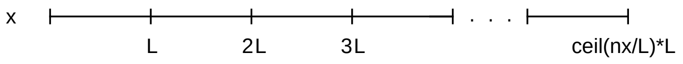
Examples

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

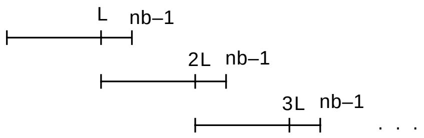
`fftfilt` uses `fft` to implement the *overlap-add method* [1], a technique that combines successive frequency domain filtered blocks of an input sequence. `fftfilt` breaks an input sequence `x` into length `L` data blocks



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^nextpow2(length(b) + length(x) - 1)
  ```

  This essentially computes

  ```
  y = ifft(fft(B,nfft).*fft(X,nfft))
  ```

If you supply a value for `n`, `fftfilt` chooses an FFT length, `nfft`, of `2^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)`.

See Also

`conv`, `filter`, `filtfilt`

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

Purpose

Rearrange the outputs of the FFT functions

Syntax

```
y = fftshift(x)
```

Description

`y = fftshift(x)` rearranges the outputs of `fft` and `fft2` by moving the zero frequency component to the center of the spectral density, which is sometimes a more convenient form.

For vectors, `fftshift(x)` returns a vector with the left and right halves swapped.

For arrays, `fftshift(x)` swaps quadrants one and three with quadrants two and four.

The `fftshift` function is part of the standard MATLAB language.

Examples

For any array `X`

```
Y = fft2(x)
```

has `Y(1,1) = sum(sum(X))`; the DC component of the signal is in the upper-left corner of the two-dimensional FFT. For

```
Z = fftshift(Y)
```

the DC component is near the center of the matrix.

See Also

`fft`, `fft2`

Purpose

Filter data with a recursive (IIR) or nonrecursive (FIR) filter

Syntax

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

Description

`y = filter(b,a,x)` filters the data in vector `x` with the filter described by coefficient vectors `a` and `b` to create the filtered data vector `y`. When `x` is a matrix, filter operates on the columns of `x`. When `x` is an *N*-dimensional array, filter operates on the first nonsingleton dimension. `a(1)` cannot be `0`, and if `a(1)` $\neq$ `1`, `filter` normalizes the filter coefficients by `a(1)`.

`[y,zf] = filter(b,a,x)` returns the final values `zf` of the state vector. When `x` is a vector, the size of the final condition vector `zf` is `max(length(b),length(a))-1`, the number of delays in the filter. When `x` is a multidimensional array, `zf` is an *s*-by-*c* matrix, where:

- *s* is the number of delays in the filter.
- *c* is `prod(size(x))/size(x,dim)`, where `dim` is the dimension into which you are filtering (the first nonsingleton dimension by default).

`[...] = filter(b,a,x,zi)` specifies initial state conditions in the vector `zi`. The size of the initial condition vector `zi` must be the same as `max(length(b),length(a))-1`, the number of delays in the filter.

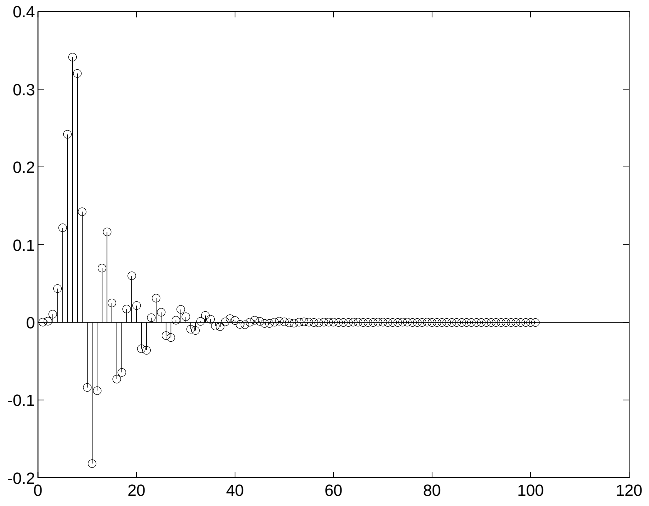
`[...] = filter(b,a,x,zi,dim)` filters the input data located along the dimension `dim` of `x`. Set `zi` to the empty vector `[]` to use zero initial conditions.

The `filter` function is part of the standard MATLAB language.

Examples

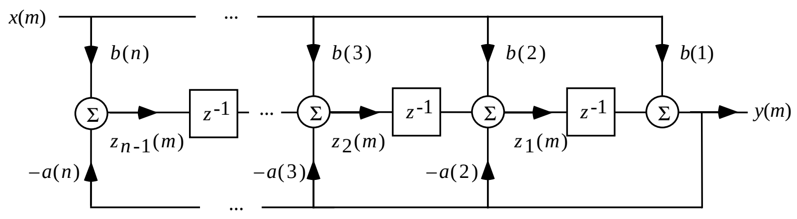
Find and graph the 101-point unit impulse response of a digital filter.

```
x = [1 zeros(1,100)];
[b,a] = butter(12,400/1000);
y = filter(b,a,x);
stem(y)
```



Algorithm

`filter` is implemented as a transposed direct form II structure [1], as shown below



where n-1 is the filter order. The *state vector* z is a vector whose components are derived from the values of the inputs to each delay in the filter.

The operation of `filter` at sample m is implemented using the transposed direct form II structure. This is calculated by the time domain difference equations for y and the states z_i.

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

Note the division by $a(1)$. For efficient computation, select this coefficient to be a power of 2.

You can use `filtic` to generate the state vector $z_i(0)$ from past inputs and outputs.

The input-output description of this filtering operation in the z-transform domain is a rational transfer function.

$$Y(z) = \frac{b(1) + b(2)z^{-1} + \cdots + b(nb+1)z^{-nb}}{a(1) + a(2)z^{-1} + \cdots + a(na+1)z^{-na}} X(z)$$

See Also

`fftfilt`, `filter2`, `filtfilt`, `filtic`

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 311-312.

filter2

Purpose Two-dimensional digital filtering

Syntax

```
Y = filter2(B,X)
Y = filter2(B,X,'shape')
```

Description `Y = filter2(B,X)` filters the two-dimensional data in `X` with the two-dimensional FIR filter in the matrix `B`. The result, `Y`, is computed using two-dimensional convolution and is the same size as `X`.

`Y = filter2(B,X,'`*`shape`*`')` returns `Y` computed with size specified by `'`*`shape`*`'`:

- `'same'` returns the central part of the convolution that is the same size as `X` (default).
- `'full'` returns the full two-dimensional convolution, when `size(Y)` exceeds `size(X)`.
- `'valid'` returns only those parts of the convolution that are computed without the zero-padded edges, when `size(Y)` is less than `size(X)`.

The `filter2` function is part of the standard MATLAB language.

Algorithm The `filter2` function uses `conv2` to compute the full two-dimensional convolution of the FIR filter with the input matrix. By default, `filter2` extracts and returns the central part of the convolution that is the same size as the input matrix. Use the `'`*`shape`*`'` parameter to specify an alternate part of the convolution for return.

See Also `conv2`, `filter`

Purpose

Compute the 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=remez(30,[.1 .9],[1 1],'Hilbert');
Linf=filternorm(b,1,inf)

Linf =

    1.0028
```

Algorithm

Given a filter H(z) with frequency reponse $H(e^{j\omega})$, the L_p-norm is given by

$$\|H\|_p \equiv \left[\frac{1}{2\pi}\int_{-\pi}^{\pi}\left|H(e^{j\omega})\right|^p d\omega\right]^{\frac{1}{p}}$$

For the case $p = \infty$, the L_∞ norm simplifies to

$$\|H\|_\infty = \max_{-\pi \le \omega \le \pi} \left|H(e^{j\omega})\right|$$

For the case $p = 2$, Parseval's theorem states that

$$\|H\|_2 = \left[\frac{1}{2\pi}\int_{-\pi}^{\pi}\left|H(e^{j\omega})\right|^2 d\omega\right]^{\frac{1}{2}} = \left[\sum_{n=-\infty}^{\infty}|h(n)|^2\right]^{\frac{1}{2}}$$

where h(n) is the impulse response of the filter. The energy of the impulse response, then, is $\|H\|_2^2$.

See Also

`zp2sos`, `norm`

Reference

Jackson, L.B., *Digital Filters and Signal Processing*, Third Edition, Kluwer Academic Publishers, 1996, Chapter 11.

filtfilt

Purpose

Zero-phase digital filtering

Syntax

```
y = filtfilt(b,a,x)
```

Description

`y = filtfilt(b,a,x)` 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. `filtfilt` minimizes start-up and ending transients by matching initial conditions, and works for both real and complex inputs.

Note that `filtfilt` should not be used with differentiator and Hilbert FIR filters, since the operation of these filters depends heavily on their phase response.

Algorithm

`filtfilt` is an M-file that uses the `filter` 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.

See Also

`fftfilt`, `filter`, `filter2`

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 311-312.

[2] Mitra, S.K., *Digital Signal Processing*, 2nd ed., McGraw-Hill, 2001, Sections 4.4.2 and 8.2.5.

[3] Gustafsson, F., “Determining the initial states in forward-backward filtering,” *IEEE Transactions on Signal Processing*, April 1996, Volume 44, Issue 4, pp. 988--992,

filtic

Purpose

Find initial conditions for a 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.

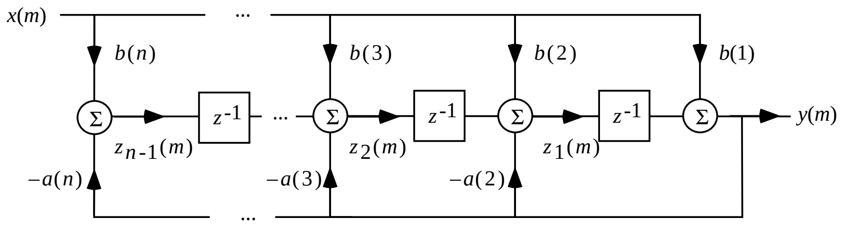
$$x = \{x(-1), x(-2), x(-3), \ldots, x(-n), \ldots\}$$
$$y = \{y(-1), y(-2), y(-3), \ldots, y(-m), \ldots\}$$

where `n` is `length(b)-1` (the numerator order) and `m` is `length(a)-1` (the denominator order). If `length(x)` is less than `n`, `filtic` pads it with zeros to length `n`; if `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



where 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.
```

See Also

`filter`, `filtfilt`

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 296, 301-302.

fir1

Purpose Design a window-based finite impulse response filter

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.

$$B(z) = b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}$$

`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 ... 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 `Wn`.
- `'stop'` for a bandstop filter, if `Wn = [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:

- `'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 $1 \le n \le N$, 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

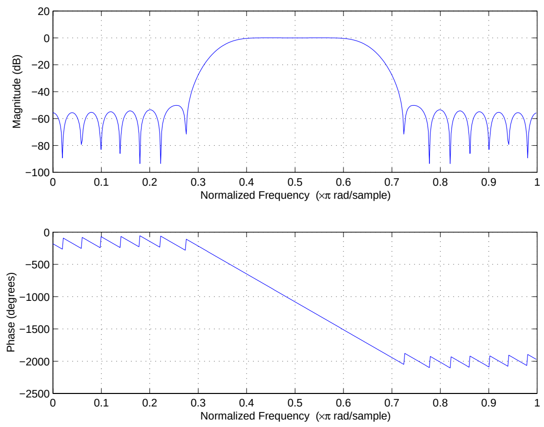
$$b(n) = w(n)h(n), \qquad 1 \le n \le N$$

Examples

Example 1

Design a 48th-order FIR bandpass filter with passband $0.35 \le \omega \le 0.65$.

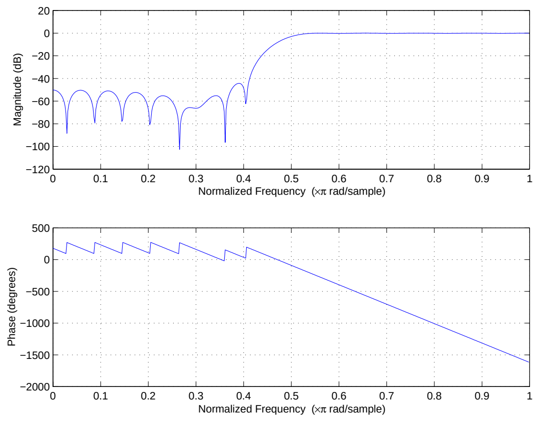
```
b = fir1(48,[0.35 0.65]);
freqz(b,1,512)
```



Example 2

The `chirp.mat` file contains a signal, `y`, that has most of its power above `fs/4`, or half the Nyquist frequency. Design a 34th-order FIR highpass filter to attenuate the components of the signal below `fs/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)
```



See Also

`cremez`, `filter`, `fir2`, `fircls`, `fircls1`, `firls`, `freqz`, `kaiserord`, `remez`, `window`

References

[1] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979. Algorithm 5.2.

Purpose

Design a frequency sampling-based finite impulse response filter

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.

$$b(z) = b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}$$

`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

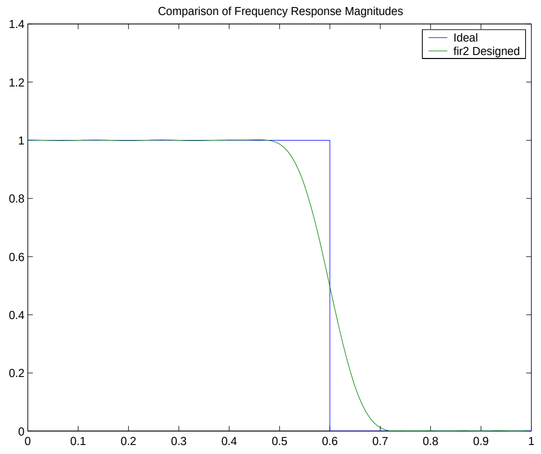
`b = fir2(n,f,m,npt,lap,window)` specify the size of the region, `lap`, that `fir2` inserts around duplicate frequency points, with or without a `window` specification.

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.

See Also

butter, cheby1, cheby2, ellip, fir1, maxflat, remez, yulewalk

Purpose

Constrained least square FIR filter design for multiband filters

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 `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. 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.

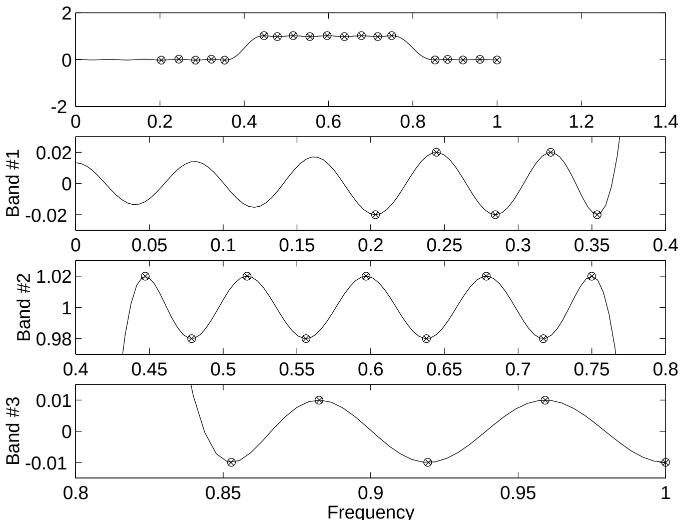
`fircls(n,f,amp,up,lo,'`*`design_flag`*`')` enables you to monitor the filter design, where `'`*`design_flag`*`'` can be:

- `'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.
- `'both'`, for both the textual display and plots.

Examples

Design an order 50 bandpass filter.

```
n = 50;
f = [0 0.4 0.8 1];
amp = [0 1 0];
up = [0.02 1.02 0.01];
lo = [-0.02 0.98 -0.01];
b = fircls(n,f,amp,up,lo,'plots');  % Plot magnitude response
```



Note Normally, the lower value in the stopband will be specified as negative. By setting `lo` 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

The algorithm is a multiple exchange algorithm that uses Lagrange multipliers and Kuhn-Tucker conditions on each iteration.

See Also

`fircls1`, `firls`, `remez`

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).

fircls1

Purpose

Constrained least square filter design for lowpass and highpass linear phase FIR filters

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` < ω < `wt`.
  - `0` < `wo` < `wt` < `1`: the amplitude of the filter is within `dp` of 1 over the frequency range `wt` < ω < `1`.

`b = fircls1(n,wo,dp,ds,wp,ws,k)` generates a lowpass FIR filter `b` with a weighted function, where `n+1` is the filter length, `wo` is the normalized cutoff frequency, `dp` is the maximum passband deviation from 1 (passband ripple), and `ds` is the maximum stopband deviation from 0 (stopband ripple). `wp` is the passband edge of the L2 weight function and `ws` is the stopband edge of the L2 weight function, where `wp` < `wo` < `ws`. `k` is the ratio (passband L2 error)/(stopband L2 error)

$$k = \frac{\int_0^{w_p} |A(\omega) - D(\omega)|^2 d\omega}{\int_{w_s}^{\pi} |A(\omega) - D(\omega)|^2 d\omega}$$

`b = fircls1(n,wo,dp,ds,wp,ws,k,'`**`high`**`')` generates a highpass FIR filter `b` with a weighted function, where `ws` < `wo` < `wp`.

`b = fircls1(n,wo,dp,ds,...,'`*`design_flag`*`')` enables you to monitor the filter design, where `'`*`design_flag`*`'` can be:

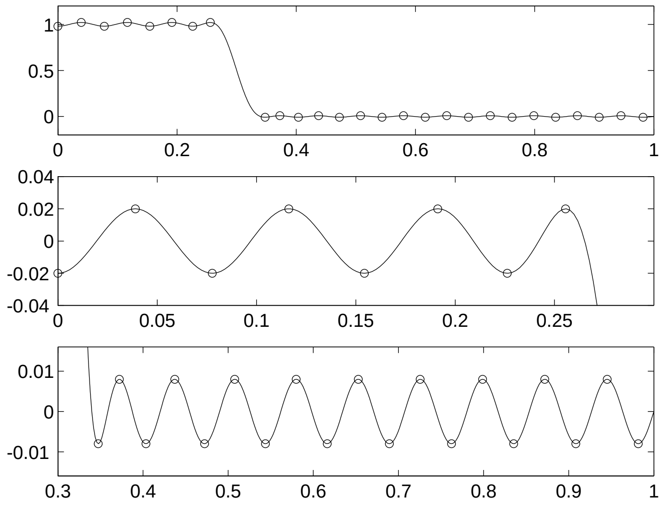
- `'trace'`, for a textual display of the design table used in the design
- `'plots'`, for plots of the filter's magnitude, group delay, and zeros and poles
- `'both'`, for both the textual display and plots

Note In the design of very narrow band filters with small `dp` and `ds`, there may not exist a filter of the given length that meets the specifications.

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,'plots'); % Plot magnitude response
```



Algorithm

The algorithm is a multiple exchange algorithm that uses Lagrange multipliers and Kuhn-Tucker conditions on each iteration.

See Also

`fircls`, `firls`, `remez`

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).

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.

$$b(k) = b(n+2-k), \qquad k = 1, \ldots, n+1$$

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:

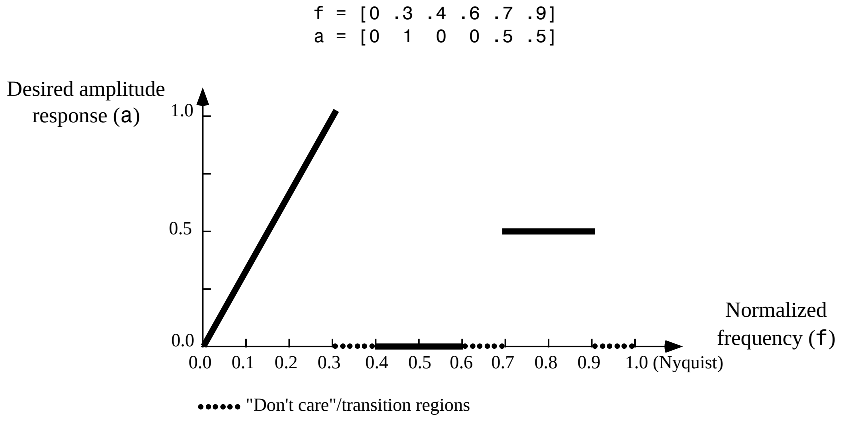
- `f` is a vector of pairs of frequency points, specified in the range between 0 and 1, where 1 corresponds to the Nyquist frequency. The frequencies must be in increasing order. Duplicate frequency points are allowed and, in fact, can be used to design a filter exactly the same as those returned by 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 $b(k) = -b(n+2-k)$, $k = 1, \ldots, n + 1$. 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

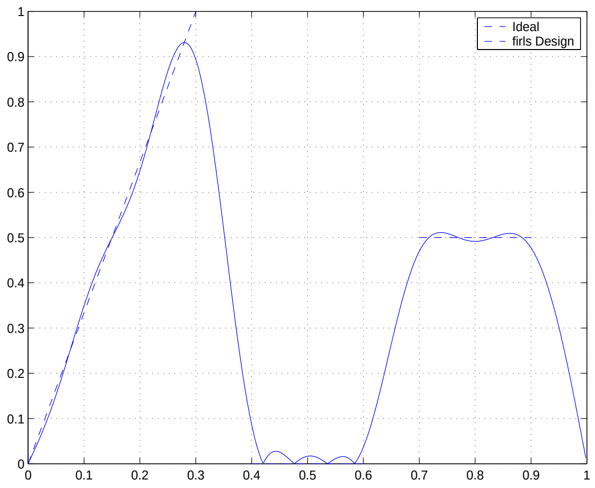
Design a 31 coefficient differentiator.

```
b = firls(30,[0 0.9],[0 0.9],'differentiator');
```

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 MATLAB's \ 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: $b(k) = b(n+2-k), \quad k = 1, \ldots, n+1$	No restriction	No restriction
Type II	Odd		No restriction	$H(1) = 0$
Type III	Even	odd: $b(k) = -b(n+2-k), \quad k = 1, \ldots, n+1$	$H(0) = 0$	$H(1) = 0$
Type IV	Odd		$H(0) = 0$	No restriction

Diagnostics

An appropriate diagnostic message is displayed when incorrect arguments are 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.

See Also

`fir1`, `fir2`, `firrcos`, `remez`

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. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 256-266.

firrcos

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 filter has cutoff frequency `F0`, transition bandwidth `df`, and sampling frequency `fs`, all in hertz. `df` must be small enough so that `F0` ± `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`.

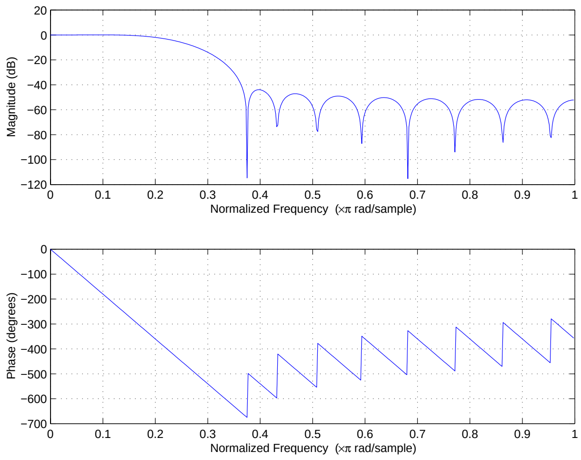
`b = firrcos(...,'`*`type`*`',delay,window)` applies a length `n+1` window to the designed filter to reduce the ripple in the frequency response. `window` must be a length `n+1` column vector. If no window is specified, a rectangular (`rectwin`) window is used. Care must be exercised when using a window 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`, `remez`

freqs

Purpose Frequency response of analog filters

Syntax

```
h = freqs(b,a,w)
[h,w] = freqs(b,a)
[h,w] = freqs(b,a,l)
freqs(b,a)
```

Description `freqs` returns the complex frequency response $H(j\omega)$ (Laplace transform) of an analog filter

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

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 frequencies specified in real vector `w`.

`[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,l)` picks `l` frequencies on which to compute the frequency response `h`.

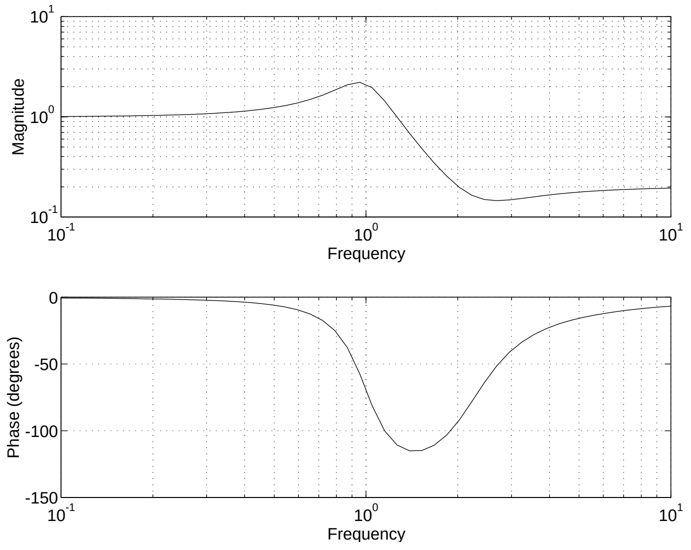
`freqs` with no output arguments plots the magnitude and phase response versus frequency in the current figure window.

`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`

freqspace

Purpose Frequency spacing for frequency response

Syntax

```
f = freqspace(n)
f = freqspace(n,'whole')
[f1,f2] = freqspace(n)
[f1,f2] = freqspace([m n])
[x1,y1] = freqspace(n,'meshgrid')
[x1,y1] = freqspace([m n],'meshgrid')
```

Description `freqspace` returns the implied frequency range for equally spaced frequency responses. This is useful when creating frequency vectors for use with `freqz`.

`f = freqspace(n)` returns the frequency vector `f` assuming `n` evenly spaced points around the unit circle. For `n` even or odd, `f` is `(0:2/n:1)`. For `n` even, `freqspace` returns `(n + 2)/2` points. For `N` odd, it returns `(n + 1)/2` points.

`f = freqspace(n,'whole')` returns `n` evenly spaced points around the whole unit circle. In this case, `f` is `0:2/n:2*(n-1)/n`.

`[f1,f2] = freqspace(n)` returns the two-dimensional frequency vectors `f1` and `f2` for an `n`-by-`n` matrix. For `n` odd, both `f1` and `f2` are `[-1 + 1/n:2/n:1-1/n]`. For `n` even, both `f1` and `f2` are `[-1:2/n:1-2/n]`.

`[f1,f2] = freqspace([m n])` returns the two-dimensional frequency vectors `f1` and `f2` for an `m`-by-`n` matrix.

`[x1,y1] = freqspace(n,'meshgrid')` and

`[x1,y1] = freqspace([m n],'meshgrid')` are equivalent to

```
[f1,f2] = freqspace(...);
[x1,y1] = meshgrid(f1,f2);
```

See the MATLAB documentation for details on the `meshgrid` function.

See Also `freqz`, `invfreqz`

Purpose

Compute the frequency response of digital filters

Syntax

```
[h,w]= freqz(b,a,l)
h = freqz(b,a,w)
[h,w] = freqz(b,a,l,'whole')
[h,w,s] = freqz(...)
[h,f] = freqz(b,a,l,fs)
h = freqz(b,a,f,fs)
[h,f] = freqz(b,a,l,'whole',fs)
[h,f,s] = freqz(...)
freqz(b,a,...)
```

Description

`[h,w] = freqz(b,a,l)` returns the frequency response vector `h` and the corresponding 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 frequency vector `w` has values ranging from 0 to π 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 2π radians per sample.

`[h,w,s] = freqz(...)` returns a structure of plotting information to be used with `freqzplot`.

`[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,l,'`**`whole`**`',fs)` uses `n` points around the entire unit circle to calculate the frequency response. The frequency vector `f` has length `l` and has values ranging from `0` to `fs` Hz.

`[h,f,units] = freqz(b,a,l,'`**`whole`**`',fs)` returns the optional string argument `units`, specifying the units for the frequency vector `f`. The string returned in `units` is `'Hz'`, denoting hertz.

`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.

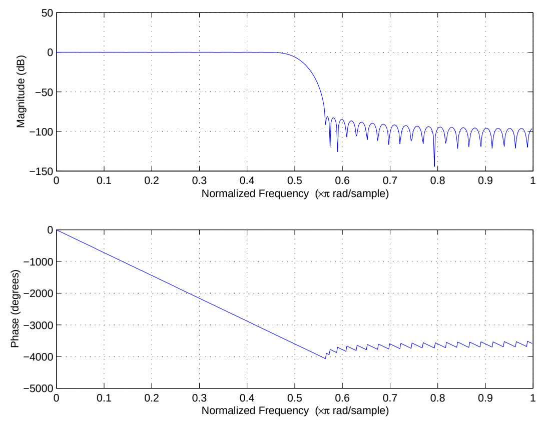
Remarks

It is best to choose a power of two 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);
```



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.

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

`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.

See Also

`abs`, `angle`, `fft`, `filter`, `freqs`, `freqzplot`, `impz`, `invfreqs`, `logspace`

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 203-205.

Purpose

Plot frequency response data

Syntax

```
freqzplot(h,w)
freqzplot(h,w,s)
freqzplot(h,w,str)
```

Description

`freqzplot(h,w)` plots the frequency response data contained in `h` at the frequencies specified in the vector `w`, where `h` can be either a vector or a matrix. `w` must be a vector whose length is the number of rows in `h`. The data in `h` is plotted versus frequency `w` on two plots:

- The magnitude of `h` is plotted in dB.
- The phase of `h` is plotted in degrees.

The units for frequency on the plots are in radians per sample. If `h` is a matrix, the frequency responses of each column of `h` is plotted.

`freqzplot(h,w,s)` specifies a structure of plotting options, `s`, with the following fields:

- `s.xunits`—a string specifying the frequency axis units. The contents of `s.xunits` can be one of the following:
  - `'rad/sample'` (default)
  - `'Hz'`
  - `'kHz'`
  - `'MHz'`
  - `'GHz'`
  - A user-specified string
- `s.yunits`—a string specifying the vertical axis units. The contents of `s.yunits` can be one of the following:
  - `'dB'` (default)
  - `'linear'`
  - `'squared'`

- `s.plot`—a string specifying the type of plot to produce. The contents of `s.plot` can be one of the following:
  - `'both'` (default)
  - `'mag'`
  - `'phase'`

Note that the `s` structure can be obtained as an output of `freqz`.

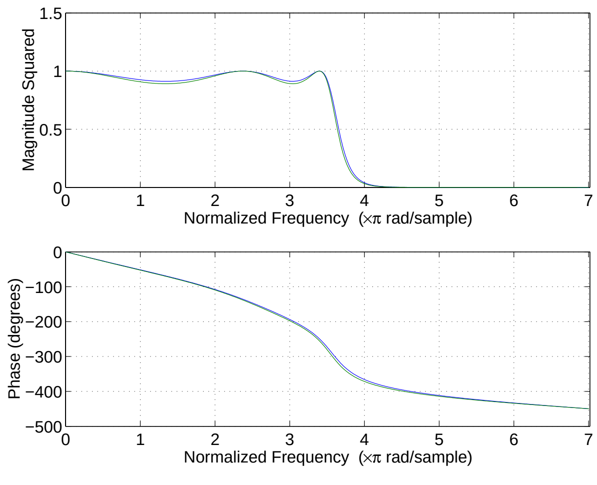
`freqzplot(h,w,str)` specifies a particular string option from the `s` structure. This is a quick way to select a single plotting option.

Examples

```
nfft = 512; Fs = 44.1;          % Fs is in kHz.
[b1,a1]  = cheby1(5,0.4,0.5);
[b2,a2]  = cheby1(5,0.5,0.5);
[h1,f,s] = freqz(b1,a1,nfft,Fs);
h2       = freqz(b2,a2,nfft,Fs);% Use the same nfft and Fs.
h = [h1 h2];
```

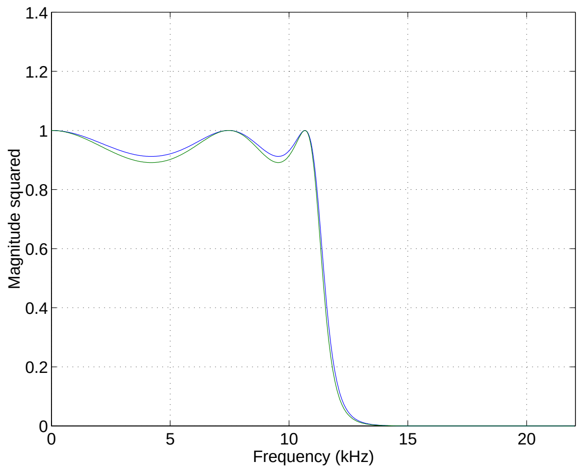
Use the quick method to specify only the `s.yunits`. The `s.plot` and `s.xunits` use the default values of ‘`both`’ and ‘`rad/sample`’, respectively.

```
freqzplot(h,f,'squared')
```



Define each of the s structure units to customize the plot.

```
s.plot   = 'mag';       % Plot the magnitude only.
s.xunits = 'khz';       % Label the frequency units correctly.
s.yunits = 'squared';   % Plot the magnitude squared.
freqzplot(h,f,s);       % Compare the two Chebyshev filters.
```



See Also

freqz, grpdelay, psdplot

fvtool

Purpose Open the Filter Visualization Tool

Syntax

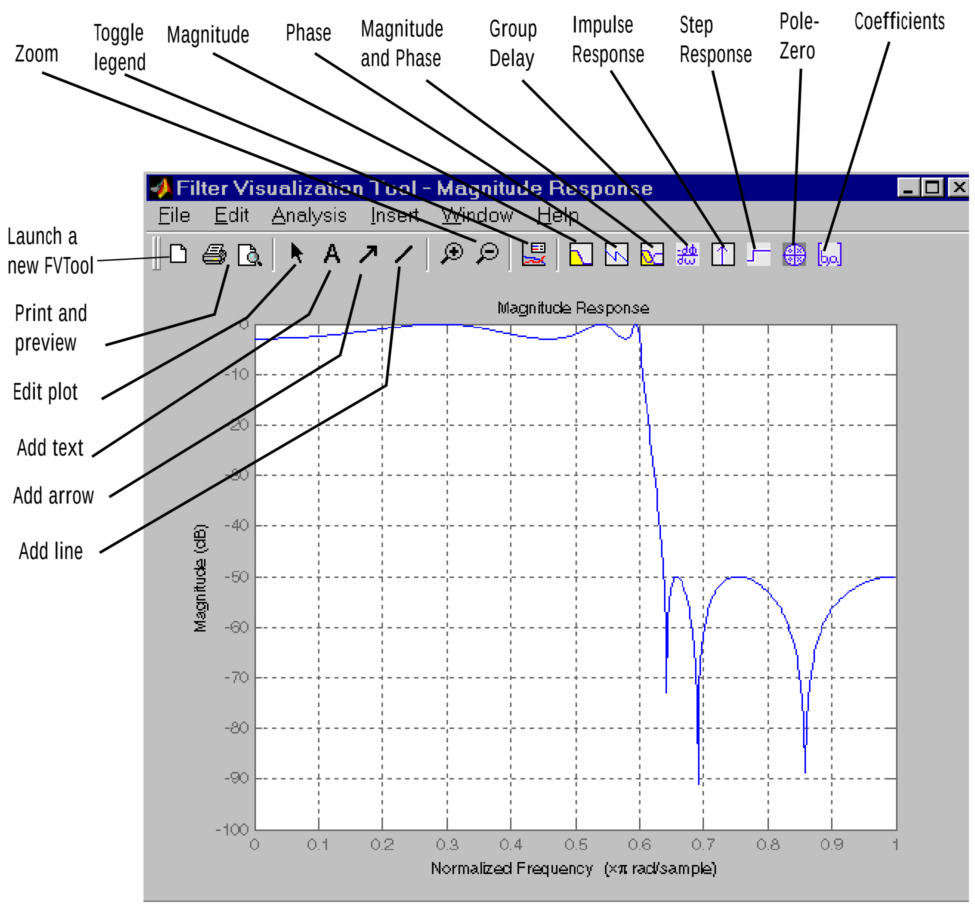
`fvtool(b,a)`

fvtool(b_1,a_1,b_2,a_2,...b_n,a_n)

Description `fvtool(b,a)` opens the Filter Visualization Tool (FVTool) and computes the magnitude response of the 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.

fvtool(b_1,a_1,b_2,a_2,...b_n,a_n) opens FVTool and computes the magnitude responses of multiple filters defined with numerators, b_1...b_n and denominators, a_1...a_n.

Note In `fdatool`, selecting **Full View Analysis** from the **Analysis** menu launches FVTool.



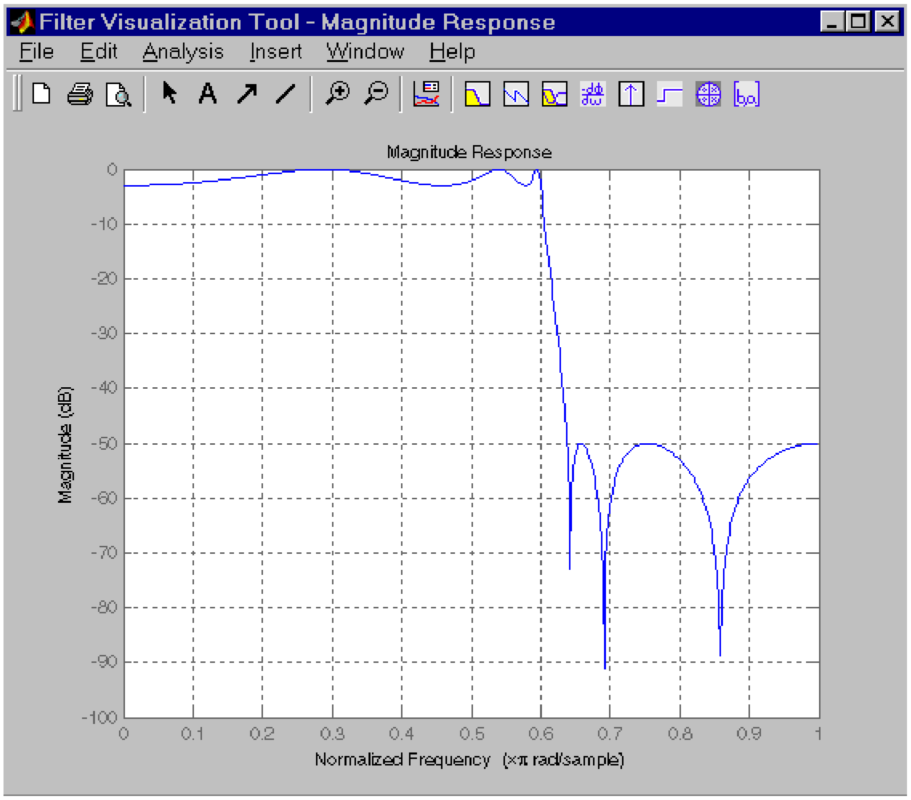
Note If you have the Filter Design Toolbox installed, you can use `fvtool` on quantized filter objects by using `fvtool(Hq,Hq1)`.

Examples

Example 1

Display the magnitude response of an elliptic filter.

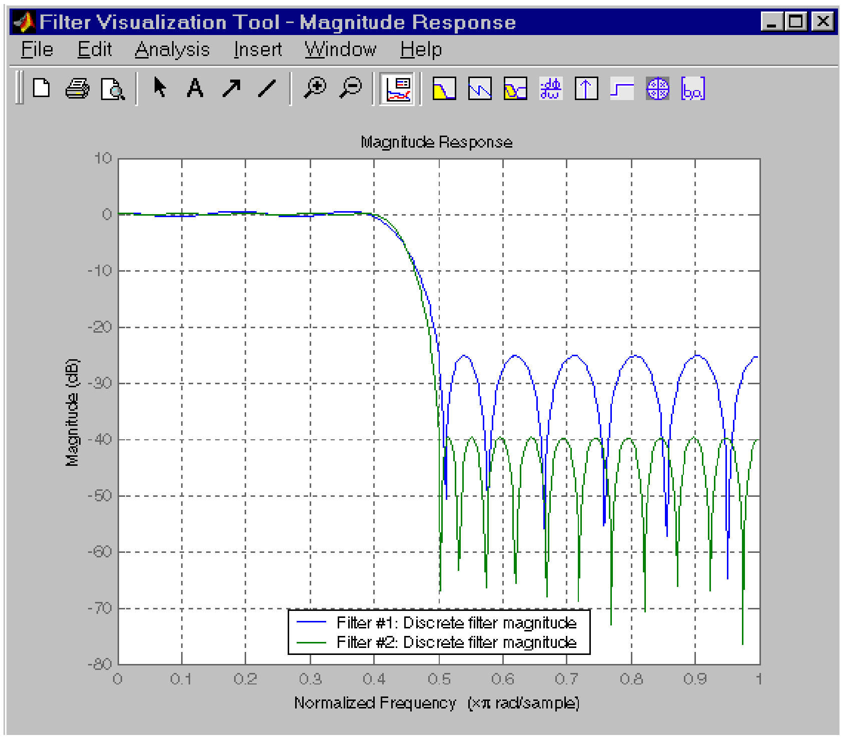
```
[b,a]=ellip(6,3,50,300/500);
fvtool(b,a);
```


Filter Visualization Tool - Magnitude Response
File Edit Analysis Insert Window Help
Magnitude Response
Magnitude (dB)
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
-100
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Normalized Frequency (×π rad/sample)

Example 2

Display and analyze multiple FIR filters.

```
n = 20;                   % Filter order
f = [0 0.4 0.5 1];        % Frequency band edges
a = [1 1 0 0];            % Desired amplitudes
b  = remez(n,f,a);
b2 = remez(n*2,f,a);      % Double the filter order
fvtool(b,1,b2,1);
```


Filter Visualization Tool - Magnitude Response
File Edit Analysis Insert Window Help
Magnitude Response
Magnitude (dB)
Normalized Frequency (×π rad/sample)
Filter #1: Discrete filter magnitude
Filter #2: Discrete filter magnitude
10
0
-10
-20
-30
-40
-50
-60
-70
-80
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

See Also `fdatool`, `sptool`

gauspuls

Purpose Generate a Gaussian-modulated sinusoidal pulse

Syntax

```
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)
```

Description `gauspuls` generates Gaussian-modulated sinusoidal pulses.

`yi = gauspuls(t,fc,bw)` returns a unity-amplitude Gaussian RF pulse at the times indicated in array `t`, with a center frequency `fc` in hertz and a fractional bandwidth `bw`, which must be greater than 0. The default value for `fc` is 1000 Hz and for `bw` is 0.5.

`yi = gauspuls(t,fc,bw,bwr)` returns a unity-amplitude Gaussian RF pulse with a fractional bandwidth of `bw` as measured at a level of `bwr` dB with respect to the normalized signal peak. The fractional bandwidth reference level `bwr` must be less than 0, because it indicates a reference level less than the peak (unity) envelope amplitude. The default value for `bwr` is -6 dB.

`[yi,yq] = gauspuls(...)` returns both the in-phase and quadrature pulses.

`[yi,yq,ye] = gauspuls(...)` returns the RF signal envelope.

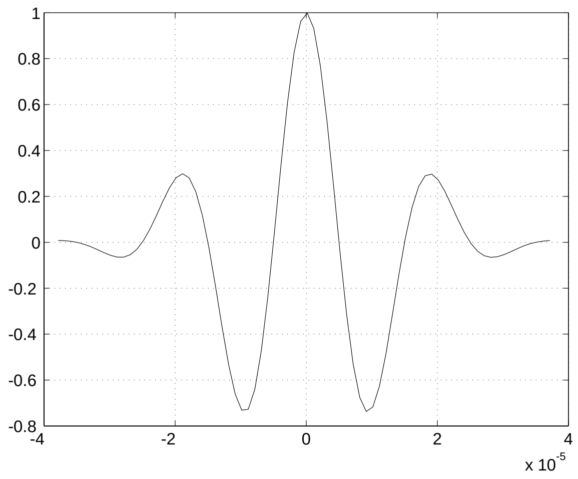
`tc = gauspuls('`**`cutoff`**`',fc,bw,bwr,tpe)` returns the cutoff time `tc` (greater than or equal to 0) at which the trailing pulse envelope falls below `tpe` dB with respect to the peak envelope amplitude. The trailing pulse envelope level `tpe` must be less than 0, because it indicates a reference level less than the peak (unity) envelope amplitude. The default value for `tpe` is -60 dB.

Remarks Default values are substituted for empty or omitted trailing input arguments.

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`

gausswin

Purpose Compute a Gaussian window

Syntax

```
w = gausswin(n)
w = gausswin(n,α)
```

Description `w = gausswin(n)` returns an `n`-point Gaussian window in the column vector `w`. `n` is a positive integer. The coefficients of a Gaussian window are computed from the following equation.

$$w\lceil k+1 \rceil = e^{-\frac{1}{2}\left(\alpha\frac{k-\frac{N}{2}}{N/2}\right)^2}$$

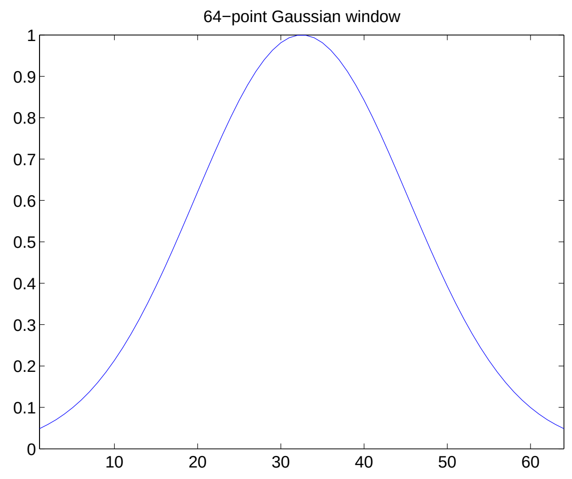
where $0 \le k \le N$ and $\alpha \ge 2$.

`w = gausswin(n,α)` returns an `n`-point Gaussian window where α is the reciprocal of the standard deviation. The width of the window is inversely related to the value of α; a larger value of α produces a narrower window. If α is omitted, it defaults to 2.5.

Note If the window appears to be clipped, increase the number of points (`n`) used for `gausswin(n)` .

Example

```
N=64;
w = gausswin(N);
plot(w); axis([1 N 0 1]);
title('64-point Gaussian window');
```



Note The shape of this window is similar in the frequency domain because the Fourier transform of a Gaussian is also a Gaussian.

See Also

`barthannwin`, `bartlett`, `blackman`, `blackmanharris`, `bohmanwin`, `chebwin`, `hann`, `hamming`, `kaiser`, `nuttallwin`, `rectwin`, `triang`, `tukeywin`, `window`

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.

gmonopuls

Purpose

Generate a 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

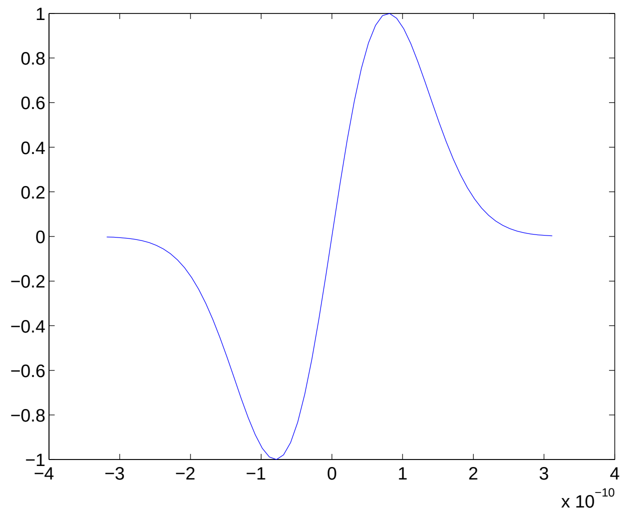
Default values are substituted for empty or omitted trailing input arguments.

Examples

Example 1

Plot a 2 GHz Gaussian monopulse sampled at a rate of 100 GHz.

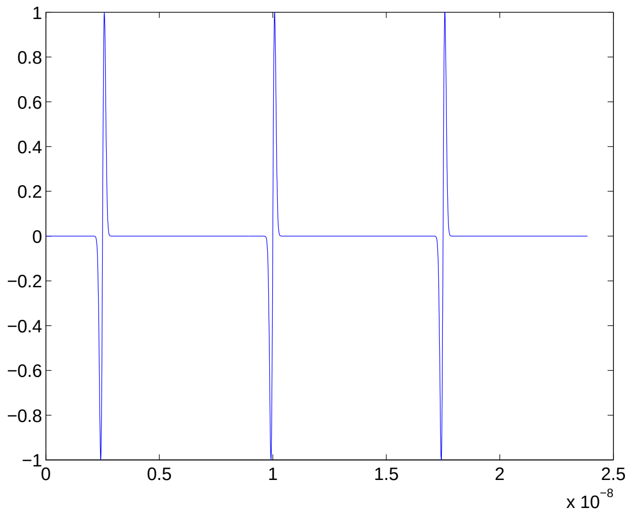
```
fc = 2E9; fs=100E9;
tc = gmonopuls('cutoff',fc);
t  = -2*tc : 1/fs : 2*tc;
y = gmonopuls(t,fc); plot(t,y)
```



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

goertzel

Purpose

Compute the DFT using the 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`.

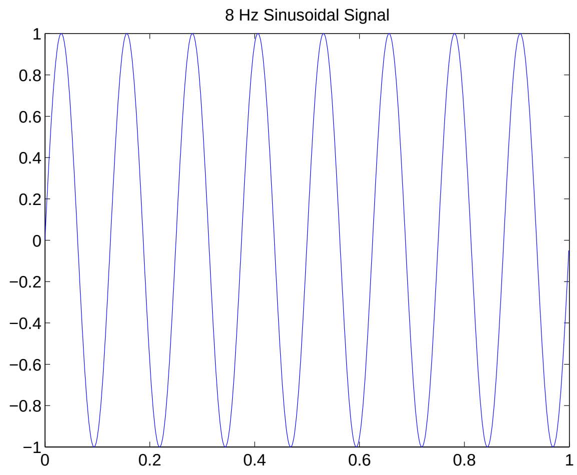
Note `fft` computes all DFT values at all indices, while `goertzel` computes DFT values at a specified subset of indices (i.e., a portion of the signal's frequency range). If less than $\log_2(N)$ points are required, `goertzel` is more efficient than the Fast Fourier Transform (`fft`).

Two examples where `goertzel` can be useful are spectral analysis of very large signals and dual-tone multifrequency (DTMF) signal detection.

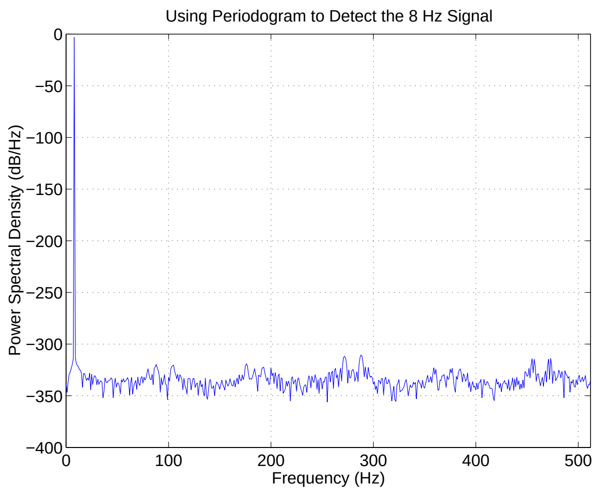
Example

Generate an 8-Hz sinusoid and use the Goertzel algorithm to detect it using the first 20 indices.

```
Fs = 1024;
Ts = 1/Fs;
f = 8;
N = 1024;
t = Ts*(0:N-1)';
x = sin(2*pi*f*t);          % Generate 8 Hz sinusoid
figure(1);
plot(t,x);
```

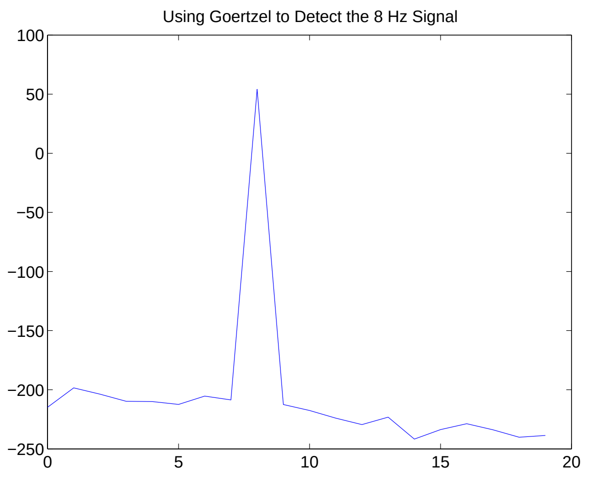


```
figure(2);
periodogram(x,[],[],Fs); % Use periodogram to obtain the PSD
                         % (computed with all N points of signal)
```



```
vec = 1:20;
X = goertzel(x,vec);     % Now use Goertzel to obtain the PSD
```

```
figure(3);                % only in the region of interest
plot(vec-1,20*log10(abs(X)));
```



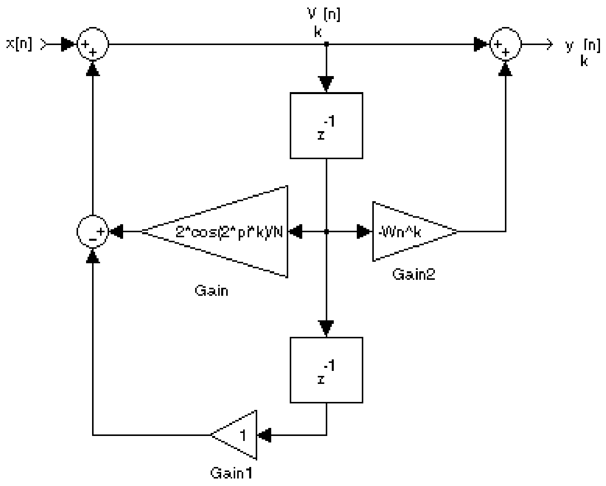
Algorithm

`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

$$v_k[n] = x[n] + 2\cos\left(\frac{2\pi k}{N}\right)v_k[n-1] - v_k[n-2]$$

where $0 \le n \le N$ 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.

See Also `fft`, `fft2`

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.

Purpose

Compute the average filter delay (group delay)

Syntax

```
[gd,w] = grpdelay(b,a,l)
[gd,f] = grpdelay(b,a,l,fs)
[gd,w] = grpdelay(b,a,l,'whole')
[gd,f] = grpdelay(b,a,l,'whole',fs)
gd = grpdelay(b,a,w)
gd = grpdelay(b,a,f,fs)
grpdelay(b,a)
```

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 ω is frequency and θ is the phase angle of $H(e^{j\omega})$.

`[gd,w] = grpdelay(b,a,l)` returns the `i`-point group delay, $\tau_g(\omega)$, of the digital filter

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

given the numerator and denominator coefficients in vectors `b` and `a`. `grpdelay` returns both `gd`, the group delay, and `w`, a vector containing the `l` frequency points in radians. `grpdelay` evaluates the group delay at `l` points equally spaced around the upper half of the unit circle, so `w` contains `l` points between 0 and π.

`[gd,f] = grpdelay(b,a,l,fs)` specifies a positive sampling frequency `fs` in hertz. It returns a length `l` vector `f` containing the actual frequency points at which the group delay is calculated, also in hertz. `f` contains `l` points between 0 and `fs/2`.

`[gd,w] = grpdelay(b,a,l,'whole')` and

`[gd,f] = grpdelay(b,a,l,'whole',fs)` use `n` points around the whole unit circle (from 0 to 2π, 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.

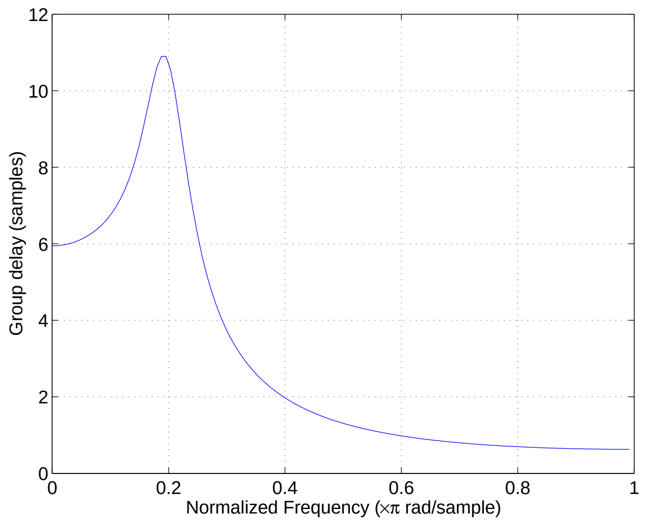
`grpdelay` with no output arguments plots the group delay versus frequency in the current figure window.

`grpdelay` works for both real and complex input systems.

Examples

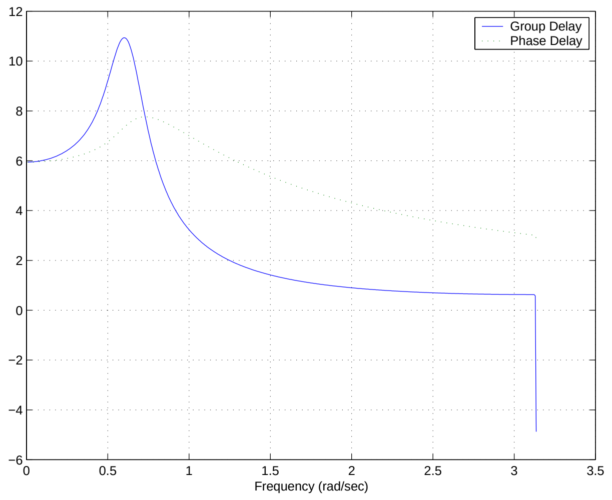
Plot the group delay of Butterworth filter $b(z)/a(z)$.

```
[b,a] = butter(6,0.2);
grpdelay(b,a,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');
```



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

hamming

Purpose Compute a Hamming window

Syntax

```
w = hamming(n)
w = hamming(n,'sflag')
```

Description `w = hamming(n)` returns an `n`-point symmetric Hamming window in the column vector `w`. `n` should be a positive integer. The coefficients of a Hamming window are computed from the following equation.

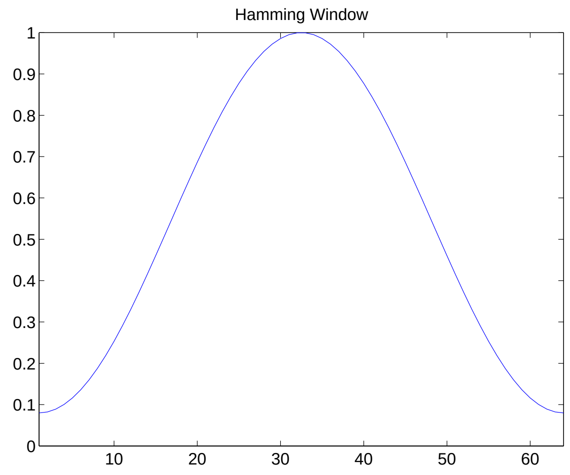
$$w[k+1] = 0.54 - 0.46\cos\left(2\pi\frac{k}{n-1}\right), \qquad k = 0, \ldots, n-1$$

`w = hamming(n,'sflag')` returns an `n`-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 `n+1` window and returns the first `n` points.

Note If you specify a one-point window (`n=1`), the value `1` is returned.

Examples

```
N=64;
w = hamming(N);
plot(w); axis([1 N 0 1]);
title('Hamming Window')
```



See Also

`barthannwin`, `bartlett`, `blackman`, `blackmanharris`, `bohmanwin`, `chebwin`, `gausswin`, `hann`, `kaiser`, `nuttallwin`, `rectwin`, `triang`, `tukeywin`, `window`

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 447-448.

Purpose

Compute a Hann (Hanning) window

Syntax

```
w = hann(n)
w = hann(n,'sflag')
```

Description

`w = hann(n)` returns an `n`-point symmetric Hann window in the column vector `w`. `n` must be a positive integer. The coefficients of a Hann window are computed from the following equation.

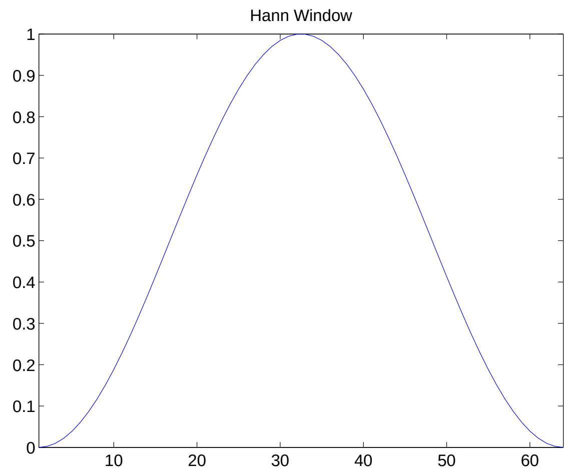
$$w[k+1] = 0.5\left(1 - \cos\left(2\pi\frac{k}{n-1}\right)\right), \qquad k = 0, \ldots, n-1$$

`w = hann(n,'sflag')` returns an `n`-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 `n+1` window and returns the first `n` points.

Note If you specify a one-point window (`n=1`), the value `1` is returned.

Examples

```
N=64;
w = hann(N);
plot(w); axis([1 N 0 1]);
title('Hann Window')
```



See Also

`barthannwin`, `bartlett`, `blackman`, `blackmanharris`, `bohmanwin`, `chebwin`, `gausswin`, `hamming`, `kaiser`, `nuttallwin`, `rectwin`, `triang`, `tukeywin`, `window`

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 447-448.

hilbert

Purpose Compute the discrete-time analytic signal using the 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 = xr + i*xi` has a real part, `xr`, which is the original data, and an imaginary part, `xi`, which contains the Hilbert transform. The imaginary part is a version of the original real sequence with a 90° 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 `xr` 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 `xr` 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.

Example

```
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, `imag(x) = [1 -1 -1 1]`, is the Hilbert transform of `xr`, and the real part, `real(x) = [1 2 3 4]`, is simply `xr` itself. Note that the last half of `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 `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, ... , (n/2)`
   - 0 for `i = (n/2)+2, ... , 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.

See Also

`fft`, `ifft`, `rceps`

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. Schafer, *Discrete-Time Signal Processing*, 2nd ed., Prentice-Hall, 1998.

icceps

Purpose Inverse complex cepstrum

Syntax `x = icceps(xhat,nd)`

Description `x = icceps(xhat,nd)` returns the inverse complex cepstrum of the (assumed real) 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 π radians.

See Also `cceps`, `hilbert`, `rceps`, `unwrap`

References [1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

Purpose

Inverse discrete cosine transform

Syntax

```
x = idct(y)
x = 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 = idct(y)` returns the inverse discrete cosine transform of `y`

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

where

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

and N = `length(x)`, which is the same as `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 = idct(y,n)` appends zeros or truncates the vector `y` to length `n` before transforming.

If `y` is a matrix, `idct` transforms its columns.

See Also

`dct`, `dct2`, `idct2`, `ifft`

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.

Purpose

One-dimensional inverse fast Fourier transform

Syntax

```
y = ifft(x)
y = ifft(x,n)
```

Description

`ifft` computes the inverse Fourier transform of a vector or array. This function implements the inverse transform given by

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

where $W_N = e^{-j(2\pi/N)}$ and N `= length(x)`. Note that the series is indexed as $n + 1$ and $k + 1$ instead of the usual n and k because MATLAB vectors run from 1 to N instead of from 0 to N-1.

`y = ifft(x)` is the inverse Fourier transform of vector `x`. If `x` is an array, `y` is the inverse FFT of each column of the matrix.

`y = ifft(x,n)` is the `n`-point inverse FFT. If the length of `x` is less than `n`, `ifft` pads `x` with trailing zeros to length `n`. If the length of `x` is greater than `n`, `ifft` truncates the sequence `x`. When `x` is an array, `ifft` adjusts the length of the columns in the same manner.

The `ifft` function is part of the standard MATLAB language.

Algorithm

The `ifft` function is an M-file. The algorithm for `ifft` is the same as that for `fft`, except for a sign change and a scale factor of `n = length(x)`. The execution time is fastest when `n` is a power of two and slowest when `n` is a large prime.

See Also

`fft`, `fft2`, `fftshift`, `ifft2`

Purpose

Two-dimensional inverse fast Fourier transform

Syntax

```
Y = ifft2(X)
Y = ifft2(X,m,n)
```

Description

`Y = ifft2(X)` returns the two-dimensional inverse fast Fourier transform (FFT) of the array `X`. If `X` is a vector, `Y` has the same orientation as `X`.

`Y = ifft2(X,m,n)` truncates or zero pads `X`, if necessary, to create an `m`-by-`n` array before performing the inverse FFT. The result `Y` is also `m`-by-`n`.

For any `X`, `ifft2(fft2(X))` equals `X` to within roundoff error. If `X` is real, `ifft2(fft2(X))` may have small imaginary parts.

The `ifft2` function is part of the standard MATLAB language.

Algorithm

The algorithm for `ifft2` is the same as that for `fft2`, except for a sign change and scale factors of `[m n] = size(X)`. The execution time is fastest when `m` and `n` are powers of two and slowest when they are large primes.

See Also

`fft`, `fft2`, `fftn`, `fftshift`, `ifft`, `ifftn`

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

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
```

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.

See Also

`bilinear`, `lp2bp`, `lp2bs`, `lp2hp`, `lp2lp`

References

[1] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, 1987, pp. 206-209.

Purpose Compute the impulse response of digital filters

Syntax

```
[h,t] = impz(b,a)
[h,t] = impz(b,a,n)
[h,t] = impz(b,a,n,fs)
impz(b,a)
impz(...)
```

Description `[h,t] = impz(b,a)` computes the impulse response of the filter with numerator coefficients `b` and denominator coefficients `a`. `impz` 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] = impz(b,a,n)` computes `n` samples of the impulse response when `n` is an integer (`t = [0:n-1]'`). If `n` is a vector of integers, `impz` computes the impulse response at those integer locations, starting the response computation from 0 (and `t = n` or `t = [0 n]`). If, instead of `n`, you include the empty vector `[]` for the second argument, the number of samples is computed automatically by default.

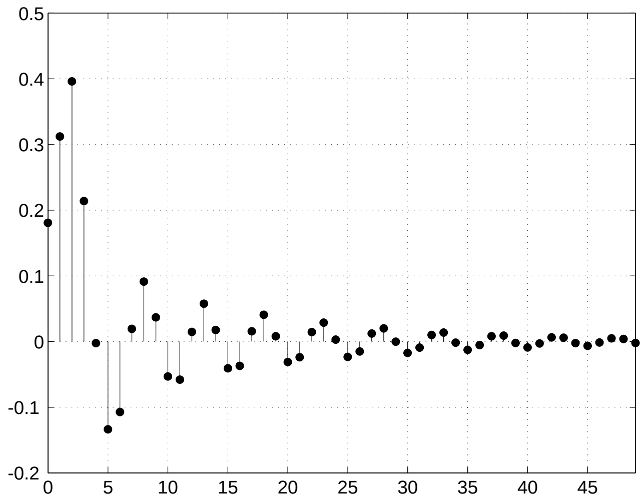
`[h,t] = impz(b,a,n,fs)` computes `n` samples and produces a vector `t` of length `n` so that the samples are spaced `1/fs` units apart.

`impz` with no output arguments plots the impulse response in the current figure window using `stem(t,h)`.

`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)
```



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 = length(b)` for the FIR case or first finds the poles using `p = roots(a)`, if `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*10^-5` of its original amplitude.

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*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 Increase sampling rate by an integer factor (interpolation)

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.

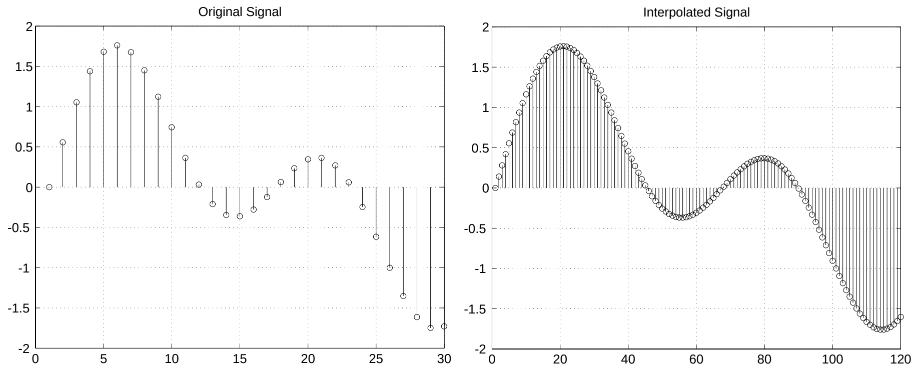
`y = interp(x,r)` increases the sampling rate of `x` by a factor of `r`. The interpolated vector `y` is `r` times longer than the original input `x`.

`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');
```



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 ≤ `alpha` ≤ 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.
```

See Also

`decimate`, `downsample`, `interp1`, `resample`, `spline`, `upfirdn`, `upsample`

References

[1] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979, Algorithm 8.1.

Purpose Interpolation FIR filter design

Syntax

```
b = intfilt(r,l,alpha)
b = intfilt(r,n,'Lagrange')
```

Description `b = intfilt(r,l,alpha)` designs a linear phase FIR filter that performs ideal bandlimited interpolation using the nearest `2*l` nonzero samples, when used on a sequence interleaved with `r-1` consecutive zeros every `r` samples. It assumes an original bandlimitedness of `alpha` times the Nyquist frequency. The returned filter is identical to that used by `interp`.

`b = intfilt(r,n,'`**`Lagrange`**`')` or

`b = intfilt(r,n,'`**`l`**`')` designs an FIR filter that performs `n`th-order Lagrange polynomial interpolation on a sequence interleaved with `r-1` consecutive zeros every `r` samples. `b` has length `(n + 1)*r` for `n` even, and length `(n + 1)*r-1` for `n` odd.

Both types of filters are basically lowpass and are intended for interpolation and decimation.

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`.

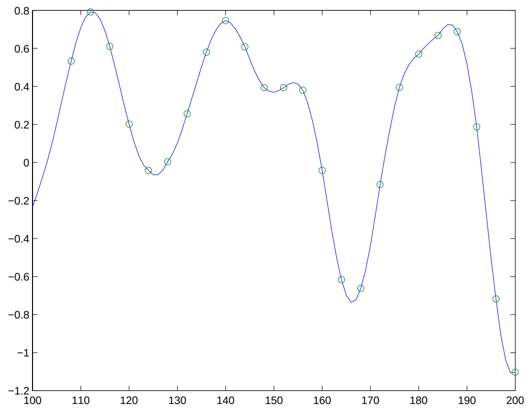
```
xr = reshape([x zeros(length(x),3)]',4*length(x),1);
```

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 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.

See Also

`decimate`, `downsample`, `interp`, `resample`, `upsample`

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.

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

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

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 π, 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 π.

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

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

by creating a system of linear equations and solving them with MATLAB's \ 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.

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

See Also

`freqs`, `freqz`, `invfreqz`, `prony`

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.

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

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

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 π, 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 -π and π.

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

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

by creating a system of linear equations and solving them with MATLAB's \ 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 a 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.

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

See Also

freqs, freqz, invfreqz, prony

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.

Purpose Convert inverse sine parameters to reflection coefficients

Syntax `k = is2rc(isin)`

Description `k = is2rc(isin)` returns a vector of reflection coefficients `k` from a vector of inverse sine parameters `isin`.

Examples

```
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
```

See Also `ac2rc`, `lar2rc`, `poly2rc`, `rc2is`

References [1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, "*Discrete-Time Processing of Speech Signals*," Prentice-Hall, 1993.

Purpose

Compute a Kaiser window

Syntax

```
w = kaiser(n,beta)
```

Description

`w = kaiser(n,beta)` returns an `n`-point Kaiser (I_0 – sinh) window in the column vector `w`. `beta` is the Kaiser window β parameter that affects the sidelobe attenuation of the Fourier transform of the window.

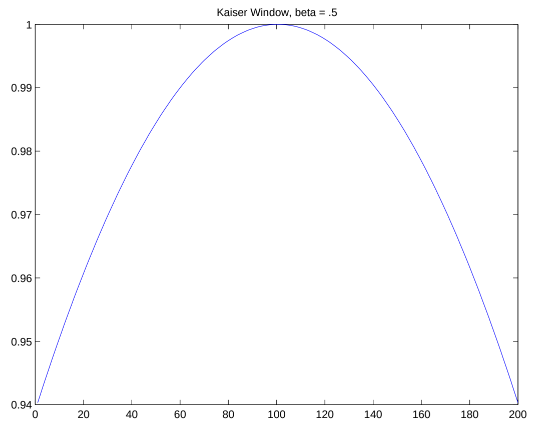
To obtain a Kaiser window that designs an FIR filter with sidelobe height $-\alpha$ dB, use the following β.

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

Increasing `beta` widens the main lobe and decreases the amplitude of the sidelobes (increases the attenuation).

Example

```
N=200;
w = kaiser(N,0.5);
plot(w);
title('Kaiser Window, beta = .5')
```



See Also

`barthannwin`, `bartlett`, `blackman`, `blackmanharris`, `bohmanwin`, `chebwin`, `gausswin`, `hamming`, `hann`, `kaiserord`, `nuttallwin`, `rectwin`, `triang`, `tukeywin`, `window`

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. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 453.

Purpose Estimate parameters for an FIR filter design with a Kaiser window

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.

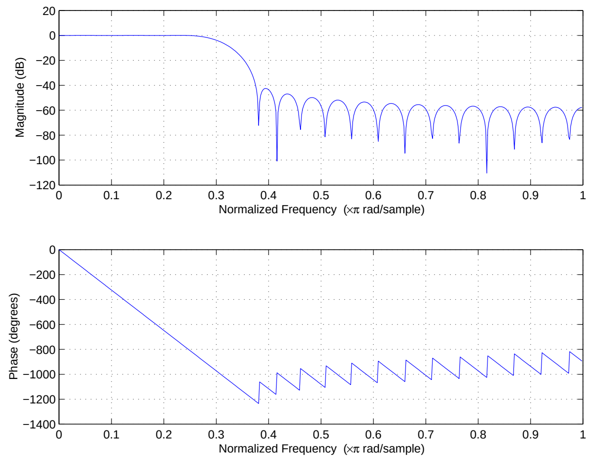
Examples

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;
fcuts = [1000 1500];
mags = [1 0];
```

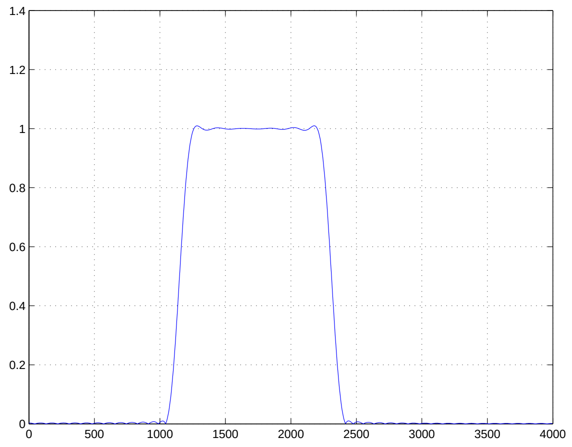
```
devs = [0.05 0.01];
[n,Wn,beta,ftype] = kaiserord(fcuts,mags,devs,fsamp);
hh = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
freqz(hh)
```



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

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

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.

See Also `fir1`, `kaiser`, `remezord`

References

[1] Kaiser, J.F., “Nonrecursive Digital Filter Design Using the I_0 - sinh Window Function,” P*roc. 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. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 458-562.

Purpose

Convert log area ratio parameters to reflection coefficients

Syntax

```
k = lar2rc(g)
```

Description

`k = lar2rc(g)` returns a vector of reflection coefficients `k` from a vector of log area ratio parameters `g`.

Examples

```
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
```

See Also

`ac2rc`, `is2rc`, `poly2rc`, `rc2lar`

References

[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, "*Discrete-Time Processing of Speech Signals*," Prentice-Hall, 1993.

Purpose

Convert lattice filter parameters to transfer function form

Syntax

```
[num,den] = latc2tf(k,v)
[num,den] = latc2tf(k,'iiroption')
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,'`*`iiroption`*`')` produces an IIR filter transfer function according to the value of the string `'`*`iiroption`*`'`:

- `'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`

latcfilt

Purpose Lattice and lattice-ladder filter implementation

Syntax

```
[f,g] = latcfilt(k,x)
[f,g] = latcfilt(k,v,x)
[f,g] = latcfilt(k,1,x)
[f,g,zf] = latcfilt(...,'ic',zi)
```

Description When filtering data, lattice coefficients can be used to represent:

- FIR filters
- All-pole IIR filters
- Allpass IIR filters
- General IIR filters

`[f,g] = latcfilt(k,x)` filters `x` with the FIR lattice coefficients in the vector `k`. `f` is the forward lattice filter result and `g` is the backward filter result. If $|k| \leq 1$, `f` corresponds to the minimum-phase output, and `g` corresponds to the maximum-phase output.

If `k` and `x` are vectors, the result is a (signal) vector. Matrix arguments are permitted under the following rules:

- If `x` is a matrix and `k` is a vector, each column of `x` is processed through the lattice filter specified by `k`.
- If `x` is a vector and `k` is a matrix, each column of `k` is used to filter `x`, and a signal matrix is returned.
- If `x` and `k` are both matrices with the same number of columns, then the *i*th column of `k` is used to filter the *i*th column of `x`. A signal matrix is returned.

`[f,g] = latcfilt(k,v,x)` filters `x` with the IIR lattice coefficients `k` and ladder coefficients `v`. `k` and `v` must be vectors, while `x` can be a signal matrix.

`[f,g] = latcfilt(k,1,x)` filters `x` with the IIR lattice specified by `k`, where `k` and `x` can be vectors or matrices. `f` is the all-pole lattice filter result and `g` is the allpass filter result.

`[f,g,zf] = latcfilt(...,'ic',zi)` accepts a length-`k` vector `zi` specifying the initial condition of the lattice states. Output `zf` is a length-`k` vector specifying the final condition of the lattice states.

See Also `filter`, `latc2tf`, `tf2latc`

Purpose

Compute the Levinson-Durbin recursion

Syntax

```
a = levinson(r,p)
```

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,p)` finds the coefficients of an pth-order autoregressive linear process which has `r` as its autocorrelation sequence. `r` is a real or complex deterministic autocorrelation sequence (a vector), and `p` is the order of denominator polynomial $A(z)$; that is, `a = [1 a(2) ... a(p+1)]`. The filter coefficients are ordered in descending powers of z.

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 + a(2)z^{-1} + \cdots + a(p+1)z^{-p}}$$

Algorithm

`levinson` solves the symmetric Toeplitz system of linear equations

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

where `r = [`$r(1)$ ... $r(p+1)$`]` is the input autocorrelation vector, and $r(i)^*$ denotes the complex conjugate of $r(i)$. The algorithm requires $O(p^2)$ flops and is thus much more efficient than the MATLAB \ command for large `p`. However, the `levinson` function uses \ for low orders to provide the fastest possible execution.

See Also

`lpc`, `prony`, `rlevinson`, `schurrc`, `stmcb`

References

[1] Ljung, L., *System Identification: Theory for the User*, Prentice-Hall, 1987, pp. 278-280.

lp2bp

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 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 `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*.

$$\frac{b(s)}{a(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n+1)}{a(1)s^m + \cdots + a(m)s + a(m+1)}$$

Scalars `Wo` and `Bw` specify the center frequency and bandwidth in units of rad/s. For a filter with lower band edge `w1` and upper band edge `w2`, use `Wo = sqrt(w1*w2)` and `Bw = w2-w1`.

`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 `Wo` and bandwidth `Bw`. For a filter with lower band edge `w1` and upper band edge `w2`, use `Wo = sqrt(w1*w2)` and `Bw = w2-w1`.

The bandpass filter is returned in matrices `At`, `Bt`, `Ct`, `Dt`.

Algorithm

`lp2bp` is a highly accurate state-space formulation of the classic analog filter frequency transformation. Consider the state-space system

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

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 ω_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

$$Q\ddot{x} + Qx = \dot{A}x + B\dot{u}$$

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 ω_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`

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 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*.

$$\frac{b(s)}{a(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n+1)}{a(1)s^m + \cdots + a(m)s + a(m+1)}$$

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 Wo and bandwidth Bw. For a filter with lower band edge w1 and upper band edge w2, use Wo = sqrt(w1*w2) and Bw = w2-w1.

The bandstop filter is returned in matrices At, Bt, Ct, Dt.

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 ω_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

```
Q = Wo/Bw;
At = [Wo/Q*inv(A) Wo*eye(ma);-Wo*eye(ma) zeros(ma)];
Bt = -[Wo/Q*(A B); zeros(ma,n)];
Ct = [C/A zeros(mc,ma)];
Dt = D - C/A*B;
```

See lp2bp for a derivation of the bandpass version of this transformation.

See Also

bilinear, impinvar, lp2bp, lp2hp, lp2lp

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 frequency of 1 rad/s into highpass filters with desired cutoff 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 frequency `Wo`. Row vectors `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of s.

$$\frac{b(s)}{a(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n+1)}{a(1)s^m + \cdots + a(m)s + a(m+1)}$$

Scalar `Wo` specifies the cutoff 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 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 frequency ω_0, the standard s-domain transformation is

$$s = \frac{\omega_0}{p}$$

The state-space version of this transformation is

```
At = Wo*inv(A);
Bt = -Wo*(A\B);
Ct = C/A;
Dt = D - C/A*B;
```

See lp2bp for a derivation of the bandpass version of this transformation.

See Also

bilinear, impinvar, lp2bp, lp2bs, lp2lp

Purpose

Change the cut-off frequency for a 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 frequency of 1 rad/s into a lowpass filter with any specified cutoff 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 frequency `Wo`. Row vectors `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of *s*.

$$\frac{b(s)}{a(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n+1)}{a(1)s^m + \cdots + a(m)s + a(m+1)}$$

Scalar `Wo` specifies the cutoff 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 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 frequency ω_0, the standard s-domain transformation is

$$s = p/\omega_0$$

The state-space version of this transformation is

```
At = Wo*A;
Bt = Wo*B;
Ct = C;
Dt = D;
```

See `lp2bp` for a derivation of the bandpass version of this transformation.

See Also

`bilinear`, `impinvar`, `lp2bp`, `lp2bs`, `lp2hp`

Purpose

Compute 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 pth-order linear predictor (FIR filter) that predicts the current value of the real-valued time series `x` based on past samples.

$$\hat{x}(n) = -a(2)x(n-1) - a(3)x(n-2) - \cdots - a(p+1)x(n-p)$$

`p` is the order of the prediction filter polynomial, `a = [1 a(2) ... a(p+1)]`. If `p` is unspecified, `lpc` uses as a default `p = 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`.

Examples

Estimate a data series using a third-order forward predictor, and compare to the original signal.

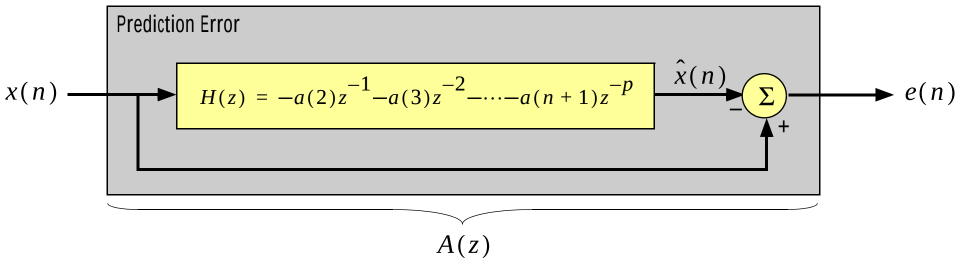
First, create the signal data as the output of an autoregressive process driven by white noise. Use the last 4096 samples of the AR process output to avoid start-up transients.

```
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.

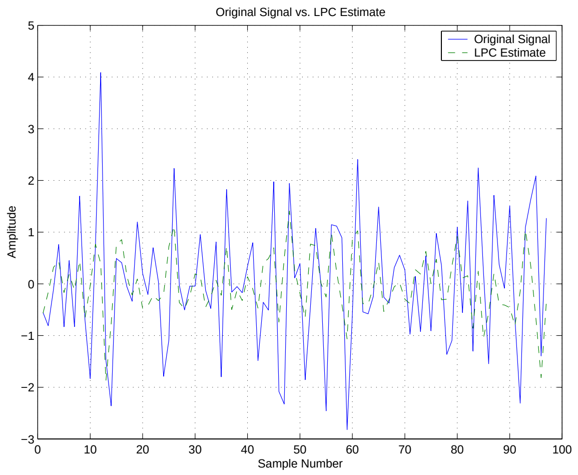
```
a = lpc(x,3);

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.

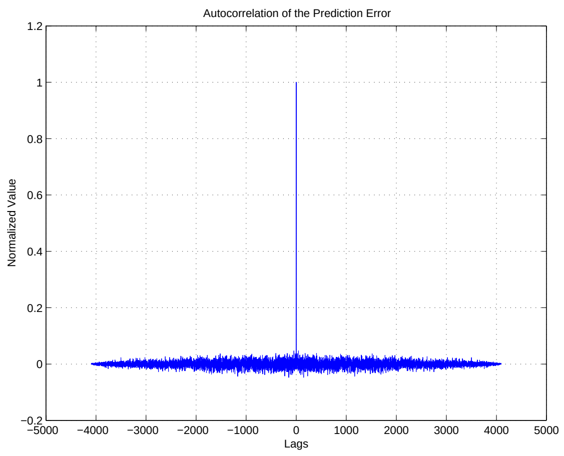
```
plot(1:97,x(4001:4097),1:97,est_x(4001:4097),'--');
title('Original Signal vs. LPC Estimate');
xlabel('Sample Number'); ylabel('Amplitude'); grid;
legend('Original Signal','LPC Estimate')
```



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

$$X = \begin{bmatrix} x(1) & 0 & \cdots & 0 \\ x(2) & x(1) & \ddots & \vdots \\ \vdots & x(2) & \ddots & 0 \\ x(m) & \vdots & \ddots & x(1) \\ 0 & x(m) & \ddots & x(2) \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & x(m) \end{bmatrix}, \quad a = \begin{bmatrix} 1 \\ a(2) \\ \vdots \\ a(p+1) \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

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 $r = [r(1)\ r(2)\ \ldots\ r(p+1)]$ 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).

See Also

aryule, levinson, prony, pyulear, stmcb

References

[1] Jackson, L.B., *Digital Filters and Signal Processing*, Second Edition, Kluwer Academic Publishers, 1989. pp. 255-257.

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 (also known as line spectrum pairs).

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
```

See Also

`ac2poly`, `poly2lsf`, `rc2poly`

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.

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)
[...] = 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.

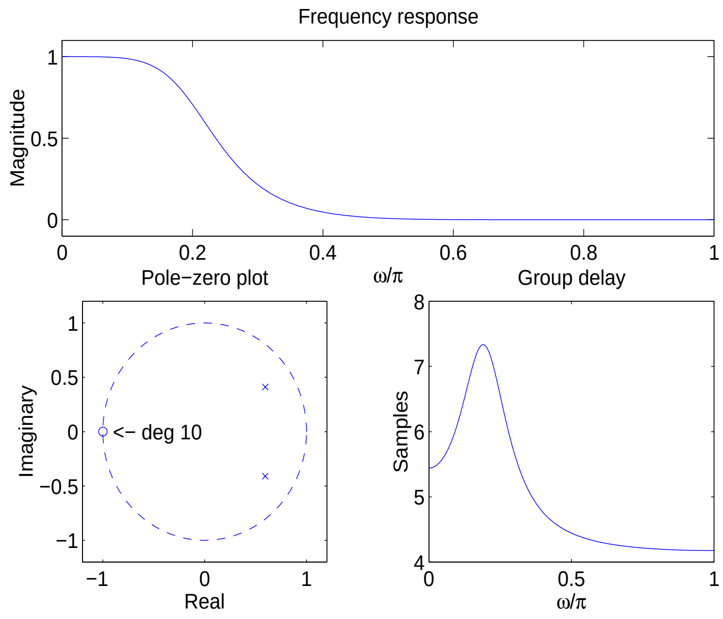
`[b,a,b1,b2] = maxflat(n,m,Wn)` returns two polynomials `b1` and `b2` whose product is equal to the numerator polynomial `b` (that is, `b = conv(b1,b2)`). `b1` contains all the zeros at `z = -1`, and `b2` contains all the other zeros.

`[...] = 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; na = 2; Wn = 0.2;
[b,a,b1,b2] = maxflat(n,m,Wn,'plots')
```



Algorithm

The method consists of the use of formulae, polynomial root finding, and a transformation of polynomial roots.

See Also

`butter`, `filter`, `freqz`

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).

Purpose

One-dimensional median filtering

Syntax

```
y = medfilt1(x,n)
y = medfilt1(x,n,blksz)
y = medfilt1(x,n,blksz,dim)
```

Description

`y = medfilt1(x,n)` applies an order `n` one-dimensional median filter to vector `x`; the function considers the signal to be 0 beyond the end points. Output `y` has the same length as `x`.

For `n` odd, `y(k)` is the median of `x(k-(n-1)/2:k+(n-1)/2)`.

For `n` even, `y(k)` is the median of `x(k-n/2)`, `x(k-(n/2)+1)`, ..., `x(k+(n/2)-1)`. In this case, `medfilt1` sorts the numbers, then takes the average of the `(n-1)/2` and `((n-1)/2)+1` elements.

The default for `n` is 3.

`y = medfilt1(x,n,blksz)` uses a `for`-loop to compute `blksz` (block size) output samples at a time. Use `blksz` << `length(x)` if you are low on memory, since `medfilt1` uses a working matrix of size `n`-by-`blksz`. By default, `blksz = length(x)`; this provides the fastest execution if you have sufficient memory.

If `x` is a matrix, `medfilt1` median filters its columns using

```
    y(:,i) = medfilt1(x(:,i),n,blksz)
```

in a loop over the columns of `x`.

`y = medfilt1(x,n,blksz,dim)` specifies the dimension, `dim`, along which the filter operates.

See Also

`filter`, `medfilt2`, `median`

References

[1] Pratt, W.K., *Digital Image Processing*, John Wiley & Sons, 1978, pp. 330-333.

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`.

`amdsb-sc` or `am` **Amplitude modulation, double sideband, suppressed carrier.** Multiplies `x` by a sinusoid of frequency `fc`.

```
y = x.*cos(2*pi*fc*t)
```

`amdsb-tc` **Amplitude modulation, double sideband, transmitted carrier.** Subtracts scalar `opt` from `x` and multiplies the result by a sinusoid of frequency `fc`.

```
y = (x-opt).*cos(2*pi*fc*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.

`amssb` **Amplitude modulation, single sideband.** Multiplies `x` by a sinusoid of frequency `fc` and adds the result to the Hilbert transform of `x` multiplied by a phase shifted sinusoid of frequency `fc`.

```
y =
x.*cos(2*pi*fc*t)+imag(hilbert(x)).*sin(2*pi*fc*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 + opt*cumsum(x))`
	`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
	`opt = (fc/fs)*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 + opt*x)`
	`modulate` uses `opt` as the constant of phase modulation. If `opt` is not present, `modulate` uses a default of
	`opt = pi/(max(max(x)))`
	so the maximum phase excursion is π 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. `modulate(x,fc,fs,'pwm','centered')` yields pulses centered at the beginning of each period. `y` is length `length(x)*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 `length(x)*fs/fc`.
`qam`	**Quadrature amplitude modulation.** Creates a quadrature amplitude modulated signal from signals `x` and `opt`. `y = x.*cos(2*pi*fc*t) + opt.*sin(2*pi*fc*t)` `opt` must be the same size as `x`.

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`

nuttallwin

Purpose

Compute a minimum 4-term Blackman-Harris window, as defined by Nuttall

Syntax

`w = nuttallwin(n)`

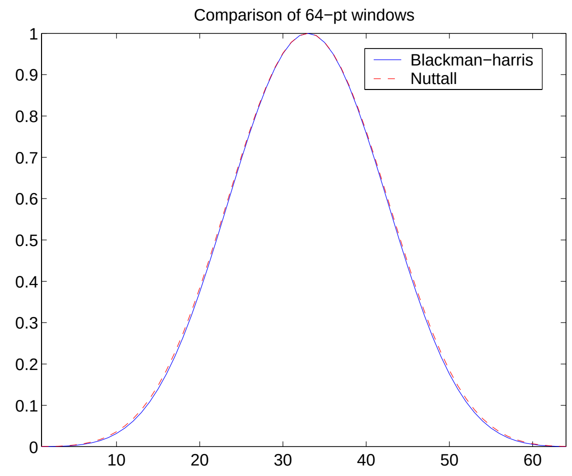
Description

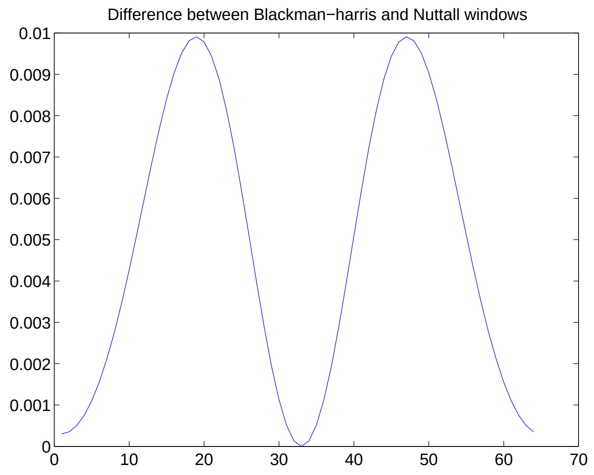
`w = nuttallwin(n)` returns a minimum, n-point, 4-term Blackman-harris window in the column vector `w`. The window is minimum in the sense that its maximum sidelobes are minimized. The coefficients for this window differ from the Blackman-harris window coefficients computed with `blackmanharris` and produce slightly lower sidelobes.

Example

Compare 64-point Blackman-Harris and Nuttall's Blackman-Harris windows and plot the difference between the windows.

```
N = 64;
w = blackmanharris(N);
y = nuttallwin(N);
plot(1:N,w,1:N,y,'r--');axis([1 N 0 1]);
title('Comparison of 64-pt windows');
legend('Blackman-harris', 'Nuttall');
plot(y-w);
title('Difference between Blackman-harris and Nuttall windows')
```





The maximum difference, using `max(abs(y-w))`, is `0.0099`.

Algorithm

The equation for computing the coefficients of a minimum 4-term Blackman-harris window, according to Nuttall, is

$$w[k+1] = a_0 - a_1\cos\left(2\pi\frac{k}{n-1}\right) + a_2\cos\left(4\pi\frac{k}{n-1}\right) - a_3\cos\left(6\pi\frac{k}{n-1}\right)$$

where $0 \le k \le (n-1)$.

The coefficients for this window are

$$a_0 = 0.3635819$$

$$a_1 = 0.4891775$$

$$a_2 = 0.1365995$$

$$a_3 = 0.0106411$$

See Also

`blackmanharris`, `barthannwin`, `bartlett`, `blackman`, `bohmanwin`, `chebwin`, `gausswin`, `hann`, `hamming`, `kaiser`, `rectwin`, `triang`, `tukeywin`, `window`

References

[1] Nuttall, Albert H. “Some Windows with Very Good Sidelobe Behavoir.” *IEEE Transactions on Acoustics, Speech, and Signal Processing.* Vol. ASSP-29 (February 1981). pp. 84-91.

Purpose

Estimate the power spectral density using the 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.

Table 7-9: 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.

`[Pxx,w] = 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 `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` in this syntax.

Table 7-10: 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`	$[0, 2\pi)$

`[Pxx,f] = pburg(x,p,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.

The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table above. The following table indicates the frequency range for `f` for this syntax.

Table 7-11: 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)`

`[Pxx,f] = 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,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, 2π).
- `'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

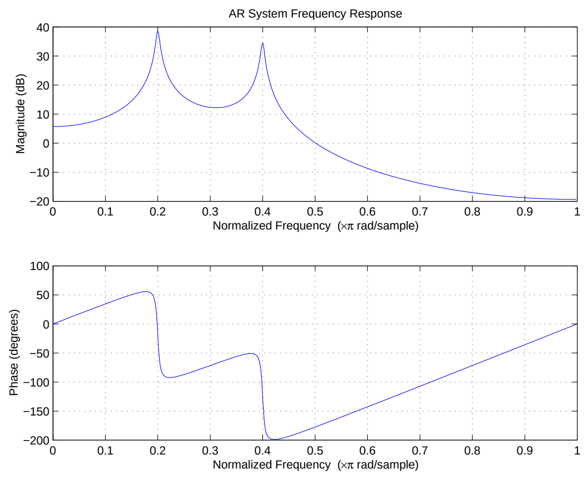
The power spectral density is computed as the distribution of power per unit frequency.

This algorithm depends on your selecting an appropriate model order for your signal.

Examples

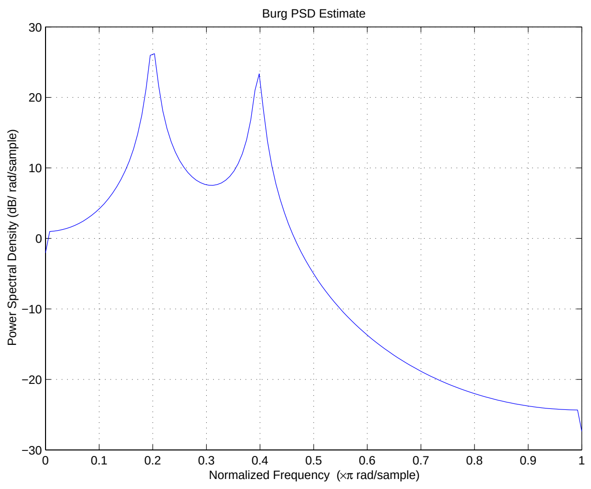
Because the Burg method estimates the spectral density by fitting an AR prediction model of a given order to the signal, first generate a signal from an AR (all-pole) model of a given order. You can use `freqz` to check the magnitude of the frequency response of your AR filter. This will give you an idea of what to expect when you estimate the PSD using `pburg`.

```
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
pburg(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 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.

See Also

`arburg`, `lpc`, `pcov`, `peig`, `periodogram`, `pmcov`, `pmtm`, `pmusic`, `pwelch`, `psdplot`, `pyulear`

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.

Purpose

Estimate the power spectral density using the 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.

Table 7-12: 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.

`[Pxx,w] = 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 `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` in this syntax.

Table 7-13: 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`	$[0, 2\pi)$

`[Pxx,f] = pcov(x,p,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.

The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table above. The following table indicates the frequency range for `f` for this syntax.

Table 7-14: 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)`

`[Pxx,f] = 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,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, 2π).
- `'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

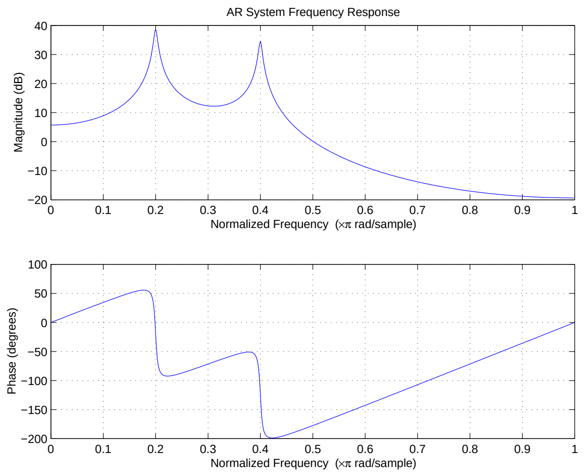
The power spectral density is computed as the distribution of power per unit frequency.

This algorithm depends on your selecting an appropriate model order for your signal.

Examples

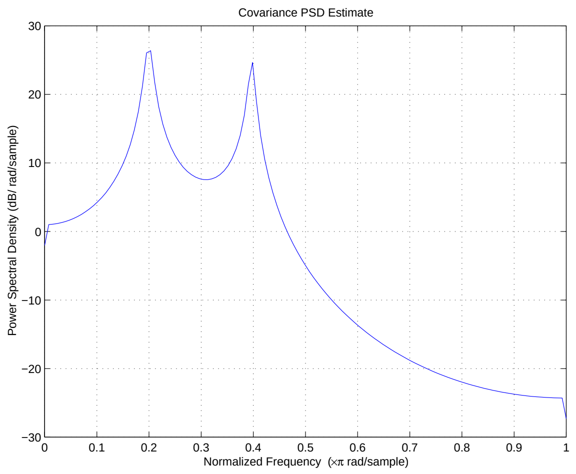
Because the covariance method estimates the spectral density by fitting an AR prediction model of a given order to the signal, first generate a signal from an AR (all-pole) model of a given order. You can use `freqz` to check the magnitude of the frequency response of your AR filter. This will give you 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.

See Also

`arcov`, `lpc`, `pburg`, `peig`, `periodogram`, `pmcov`, `pmtm`, `pmusic`, `pwelch`, `psdplot`, `pyulear`

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.

Purpose

Estimate the pseudospectrum using the 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 λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold λ_{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.

Table 7-15: 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] = 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.

Table 7-16: S and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of S and w	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`	$[0, 2\pi)$

`[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 Table 7-16. The following table indicates the frequency range for f for this syntax.

Table 7-17: 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] = peig(...,'**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] = 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.

Table 7-18: 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 length(nwin)

Table 7-18: 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 Table 7-19 for related information on this syntax.

Note The arguments `nwin` and `noverlap` 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,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, 2π).
- `'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 λ_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.

Table 7-19: Eigenvector Length Depending on Input Data and Syntax

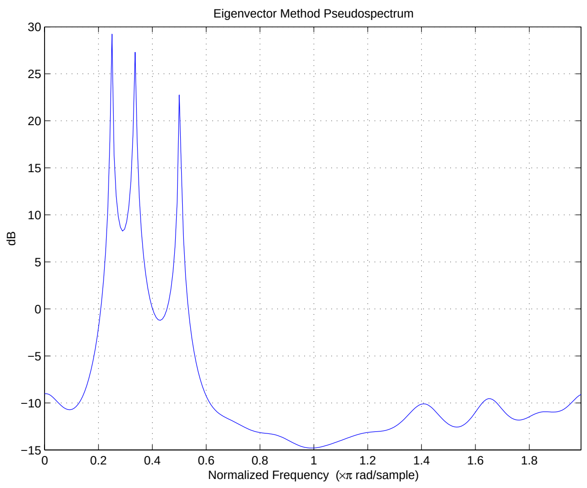
Form of Input Data x	Comments on the Syntax	Length n of Eigenvectors
Row or column vector	`nwin` is specified as a scalar integer.	`nwin`
Row or column vector	`nwin` is specified as a vector.	`length(nwin)`
Row or column vector	`nwin` is not specified.	`2*p(1)`
l-by-m matrix	If `nwin` is specified as a scalar, it is not used. If `nwin` is specified as a vector, `length(nwin)` must equal m.	m
m-by-m nonnegative definite matrix	The string `'corr'` is specified and `nwin` 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

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

where N is the dimension of the eigenvectors and $\mathbf{v}_k$ is the kth 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 λ_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.

See Also

`corrmtx`, `pburg`, `periodogram`, `pmtm`, `pmusic`, `prony`, `pwelch`, `psdplot`, `rooteig`, `rootmusic`

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.

Purpose Estimate the power spectral density (PSD) of a signal 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 two 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.

Table 7-20: PSD Vector Characteristics for an FFT Length of N (Default)

Real/Complex Input Data	Length of Pxx	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.

Table 7-21: 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	$[0, 2\pi)$

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.

The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table above. The following table indicates the frequency range for `f` for this syntax.

Table 7-22: 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)`

`[Pxx,f] = periodogram(x,window,nfft,fs,'`*`range`*`')` or

`[Pxx,w] = periodogram(x,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, 2π).
- `'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`.

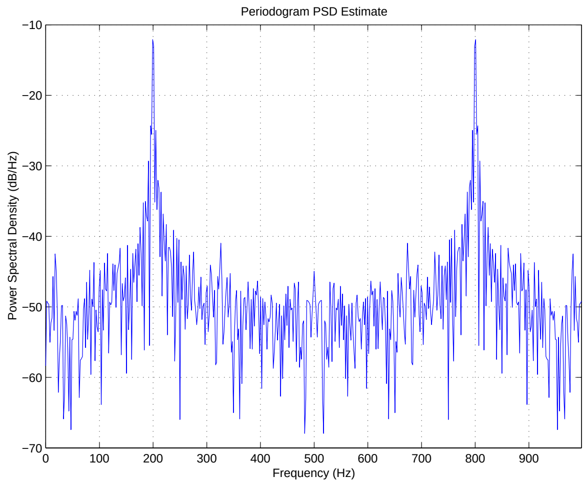
Note You can put the string argument `'`*`range`*`'` anywhere in the input argument list after `window`.

`periodogram(...)` 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.

$$S(e^{j\omega}) = \frac{1}{n}\left|\sum_{l=1}^{n} x_l e^{-j\omega l}\right|^2$$

This expression forms an estimate of the power spectrum of the signal defined by the sequence $[x_1, \dots, x_n]$.

If you weight your signal sequence by a window $[w_1, \ldots, w_n]$, 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 `nfft`-point FFT to compute the power spectral density as $S(e^{j\omega})/F$, where F is:

- 2π when you do not supply the sampling frequency
- `fs` when you supply the sampling frequency

See Also

`pburg`, `pcov`, `peig`, `pmcov`, `pmtm`, `pmusic`, `pwelch`, `psdplot`, `pyulear`

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. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 730-742.

Purpose

Estimate the power spectral density using the 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.

Table 7-23: 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.

`[Pxx,w] = 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 `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.

Table 7-24: 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`	$[0, 2\pi)$

`[Pxx,f] = pmcov(x,p,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.

The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in Table 7-24. The following table indicates the frequency range for `f` in this syntax.

Table 7-25: 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)`

`[Pxx,f] = 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,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, 2π).
- `'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

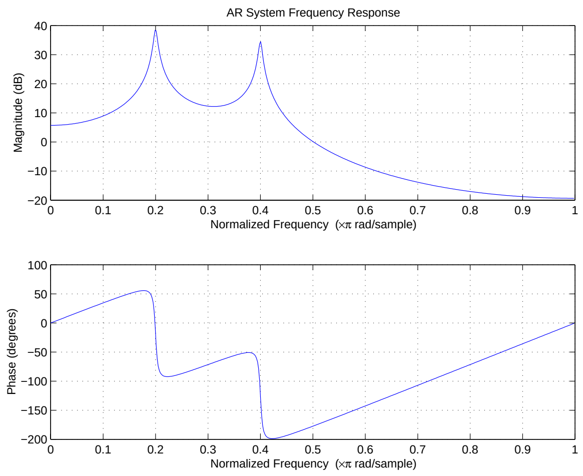
The power spectral density is computed as the distribution of power per unit frequency.

This algorithm depends on your selecting an appropriate model order for your signal.

Examples

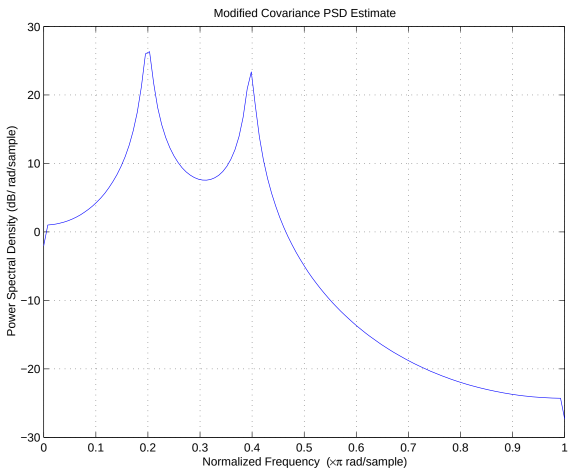
Because the modified covariance method estimates the spectral density by fitting an AR prediction model of a given order to the signal, first generate a signal from an AR (all-pole) model of a given order. You can use `freqz` to check the magnitude of the frequency response of your AR filter. This will 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.

See Also

`armcov`, `lpc`, `pburg`, `pcov`, `peig`, `periodogram`, `pmtm`, `pmusic`, `pwelch`, `prony`, `psdplot`, `pyulear`

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.

Purpose

Estimate the power spectral density using the 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 Pxx 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. Pxx 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 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 two greater than the length of the segment. The following table indicates the length of Pxx and the range of the corresponding normalized frequencies for this syntax.

Table 7-26: PSD Vector Characteristics for an FFT Length of N (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	$(N/2) +1$	$[0, \pi]$
Complex-valued	N	$[0, 2\pi)$

`[Pxx,w] = 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 `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.

Table 7-27: 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`	$[0, 2\pi)$

`[Pxx,f] = pmtm(x,nw,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 `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 Pxx is the same as in Table 7-27. The following table indicates the frequency range for f for this syntax.

Table 7-28: 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)

[Pxx,Pxxc,f] = pmtm(x,nw,nfft,fs) returns Pxxc, the 95% confidence interval for Pxx. Confidence intervals are computed using a chi-squared approach. Pxxc is a two-column matrix with the same number of rows as Pxx. Pxxc(:,1) is the lower bound of the confidence interval and Pxxc(:,2) is the upper bound of the confidence interval.

[Pxx,Pxxc,f] = pmtm(x,nw,nfft,fs,p) returns Pxxc, the p*100% confidence interval for Pxx, 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.

[Pxx,Pxxc,f] = pmtm(x,e,v,nfft,fs,p) returns the PSD estimate Pxx, the confidence interval Pxxc, 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.

[Pxx,Pxxc,f] = 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.

`[...] = 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,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, 2π).
- `'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`.

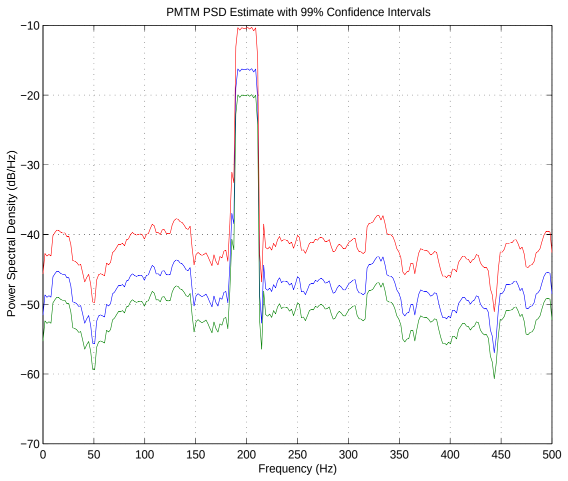
Note You can put the string arguments `'range'` or `'method'` anywhere after the input argument `nw` or `v`.

`pmtm(...)` with no output arguments plots the PSD estimate and the confidence intervals in the current figure window. If you don't specify `fs`, the 95% confidence interval is plotted. If you do specify `fs`, 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);
psdplot([Pxx Pxxc],f,'hz','db')
title('PMTM PSD Estimate with 99% Confidence Intervals')
```



See Also

`dpss`, `pburg`, `pcov`, `peig`, `periodogram`, `pmcov`, `pmusic`, `pwelch`, `psdplot`, `pyulear`

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.

Purpose

Estimate the pseudospectrum using the 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 λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold λ_{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.

Table 7-29: 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] = pmusic(...,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 in this syntax.

Table 7-30: S and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of S and w	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`	$[0, 2\pi)$

`[S,f] = pmusic(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 Table 7-30. The following table indicates the frequency range for f for this syntax.

Table 7-31: 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] = 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.

Table 7-32: 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 length(nwin)

Table 7-32: 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 Table 7-33 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,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 $[0, 2\pi)$.
- `'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 λ_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.

Table 7-33: Eigenvector Length Depending on Input Data and Syntax

Form of Input Data 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.	`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, `length(nwin)` must equal *m*.	*m*
m-by-*m* nonnegative definite matrix	The string `'corr'` is specified and `nwin` is not used.	*m*

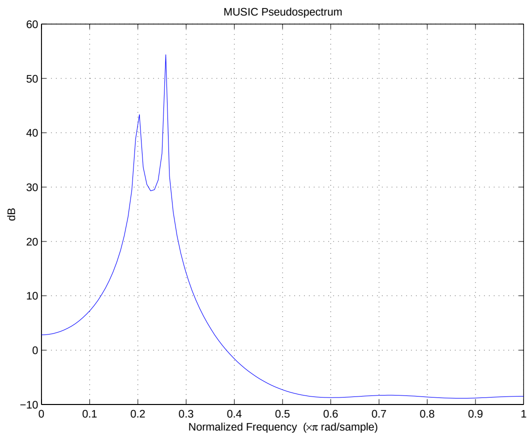
You should specify `nwin` > `p(1)` or `length(nwin)` > `p(1)` if you want `p(2)` > `1` to have any effect.

Examples

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) = 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 Xm 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

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

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.

See Also

`corrmtx`, `pburg`, `peig`, `periodogram`, `pmtm`, `prony`, `psdplot`, `pwelch`, `rooteig`, `rootmusic`

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.

poly2ac

Purpose Convert a prediction filter polynomial to an 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
```

See Also `ac2poly`, `poly2rc`, `rc2ac`

References

[1] Kay, S.M. *Modern Spectral Estimation*, Englewood Cliffs, NJ, Prentice-Hall, 1988.

poly2lsf

Purpose

Convert prediction filter coefficients to line spectral frequencies

Syntax

```
lsf = poly2lsf(a)
```

Description

`lsf = poly2lsf(a)` returns a vector `lsf` of line spectral frequencies (also known as line spectrum pairs) from a vector `a` of prediction filter coefficients.

Examples

```
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
```

See Also

`lsf2poly`

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.

Purpose Convert a 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 `NaN`s and provide a warning message in this case.

Algorithm

`poly2rc` implements the recursive relationship

$$k(n) = a_n(n)$$

$$a_{n-1}(m) = \frac{a_n(m) - k(n)a_n(n-m)}{1-k(n)^2}, \qquad m = 1, 2, \ldots, n-1$$

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)*fliplr(a))/(1–k(i)^2);
   ```

See Also

`ac2rc`, `latc2tf`, `latcfilt`, `poly2ac`, `rc2poly`, `tf2latc`

References

[1] Kay, S.M. *Modern Spectral Estimation*, Englewood Cliffs, NJ, Prentice-Hall, 1988.

Purpose

Scale the roots of a 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`

polystab

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.

$$a(z) = a(1) + a(2)z^{-1} + \cdots + a(m+1)z^{-m}$$

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`

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.

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

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.

See Also `butter`, `cheby1`, `cheby2`, `ellip`, `invfreqz`, `levinson`, `lpc`, `stmcb`

References [1] Parks, T.W., and C.S. Burrus, *Digital Filter Design,* John Wiley & Sons, 1987, pp. 226-228.

Purpose

Plot power spectral density (PSD) data

Syntax

```
psdplot(Pxx,w)
psdplot(Pxx,w,'units')
psdplot(Pxx,w,'units','yscale')
psdplot(Pxx,w,'units','yscale','title')
```

Description

`psdplot(Pxx,w)` plots the power spectral density data contained in the vector `Pxx` at the frequencies specified in the vector `w`. The vectors `Pxx` and `w` must have the same length. The magnitude of the data vector `Pxx` is plotted in dB per radians per sample versus frequency `w` on one plot. The units for frequency on the plots are rad/sample.

`psdplot(Pxx,w,'`*`units`*`')` displays the frequency units according to the string `'`*`units`*`'`. This string can be either:

- `'rad/sample'` (default)
- `'Hz'`

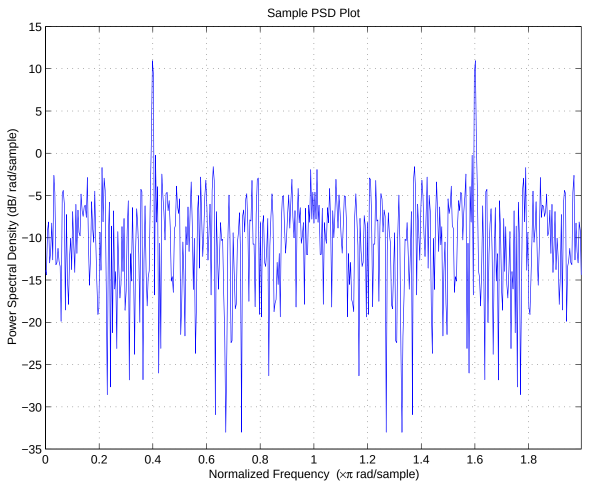
`psdplot(Pxx,w,'`*`units`*`','`*`yscale`*`')` displays the spectral density magnitude units according to the string `'`*`yscale`*`'`. This string can be either:

- `'db'` (default), for plotting the PSD in dB per units of frequency
- `'linear'`, for plotting the PSD in units of power per units of frequency

`psdplot(Pxx,w,'`*`units`*`','`*`yscale`*`','title')` uses the contents of the string `'title'` as the title of the plot.

Examples

```
t = 0:0.001:0.3;
x = cos(2*pi*t*200) + randn(size(t));
[Pxx,w] = periodogram(x,[],'twosided',512);
psdplot(Pxx,w,'','','Sample PSD Plot')
```



See Also

`freqzplot`, `pburg`, `pcov`, `peig`, `periodogram`, `pmcov`, `pmusic`, `pwelch`, `pyulear`

Purpose

Generate a 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 = func(t-d(1)) + func(t-d(2)) + ...`

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,

```
func(t-d(1),p1,p2,...) + func(t-d(2),p1,p2,...) + ...
```

`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,(length(p)-1)/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.

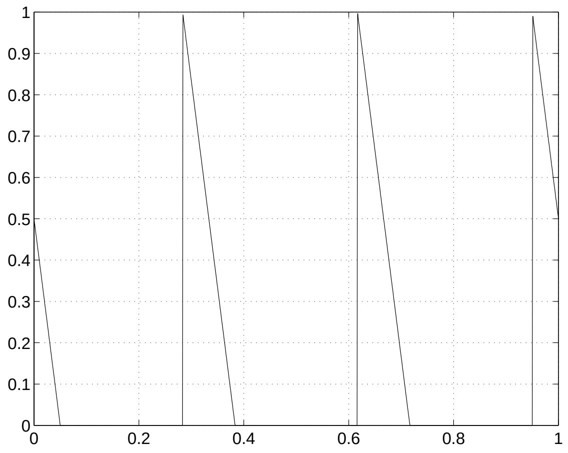
pulstran(...,'*func*') specifies alternative interpolation methods. See interp1 for a list of available methods.

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.

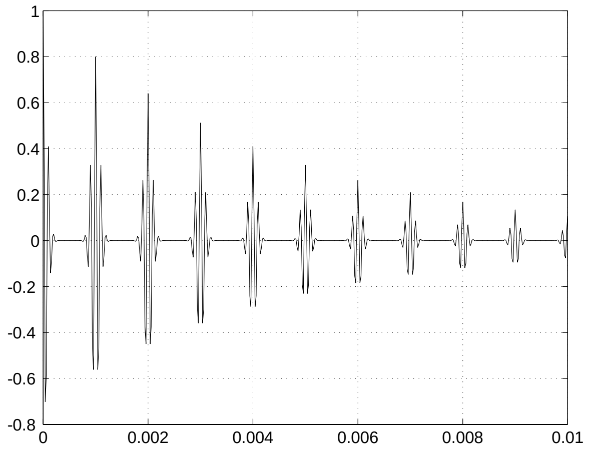
```
t = 0 : 1/1e3 : 1;          % 1 kHz sample freq for 1 sec
d = 0 : 1/3 : 1;            % 3 Hz repetition freq
y = pulstran(t,d,'tripuls',0.1,-1);
plot(t,y)
```



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.

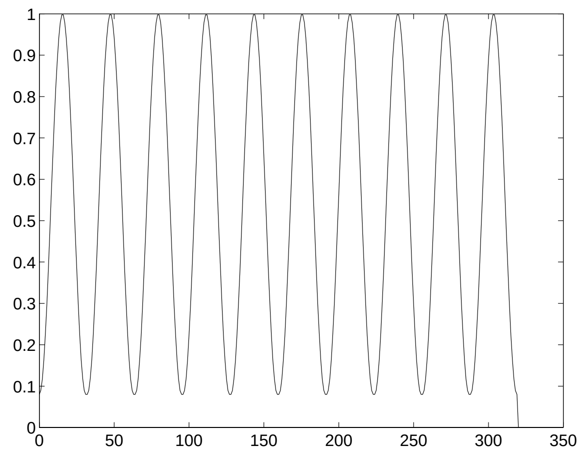
```
t = 0 : 1/50E3 : 10e-3;
d = [0 : 1/1E3 : 10e-3 ; 0.8.^(0:10)]';
y = pulstran(t,d,'gauspuls',10e3,0.5);
plot(t,y)
```



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

Purpose

Estimate the power spectral density (PSD) of a signal 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

Note `pwelch` computes the power spectral density, not the power spectrum. The difference between them is discussed in "Spectral Analysis" on page 3-6.

`[Pxx,w] = pwelch(x)` estimates the power spectral density `Pxx` 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 `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 two greater than the length of the segment. The following table indicates the length of Pxx and the range of the corresponding normalized frequencies for this syntax.

Table 7-34: PSD Vector Characteristics for an FFT Length of N (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	$(N/2)+1$	$[0, \pi]$
Complex-valued	N	$[0, 2\pi)$

`[Pxx,w] = pwelch(x,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.

`[Pxx,w] = pwelch(x,window,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).

`[Pxx,w] = pwelch(x,window,noverlap,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 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.

Table 7-35: 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	$[0, 2\pi)$

[Pxx,f] = pwelch(x,window,noverlap,nfft,fs) uses the sampling frequency fs specified 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.

The frequency range for f depends on nfft, fs, and the values of the input x. The length of Pxx is the same as in Table 7-35. The following table indicates the frequency range for f for this syntax.

Table 7-36: 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)

`[...] = pwelch(x,window,noverlap,...,'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,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`.

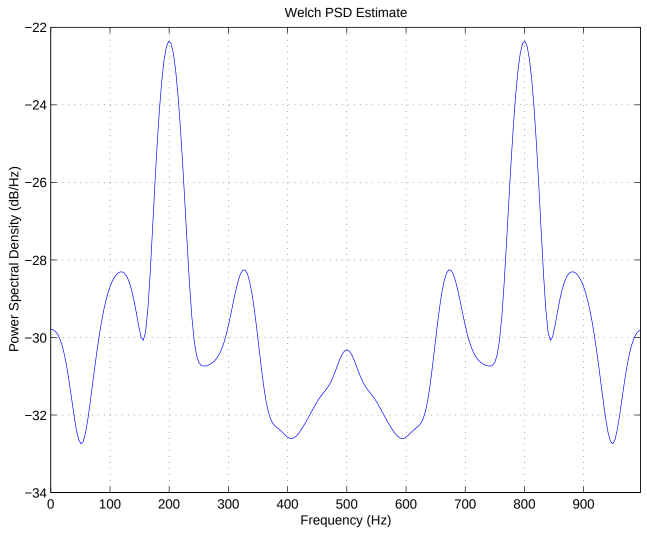
The string `'range'` can appear anywhere in the syntax after `noverlap`.

`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:
   - 2π 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).

See Also

`pburg`, `pcov`, `peig`, `periodogram`, `pmcov`, `pmtm`, `pmusic`, `psdplot`, `pyulear`

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.

Purpose

Estimate the power spectral density using the 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.

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.

Table 7-37: 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] = 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.

`[Pxx,w] = 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 `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.

Table 7-38: 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`	$[0, 2\pi)$

`[Pxx,f] = pyulear(x,p,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.

The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in Table 7-38. The following table indicates the frequency range for `f` for this syntax.

Table 7-39: 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)`

`[Pxx,f] = 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,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, 2π).
- `'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

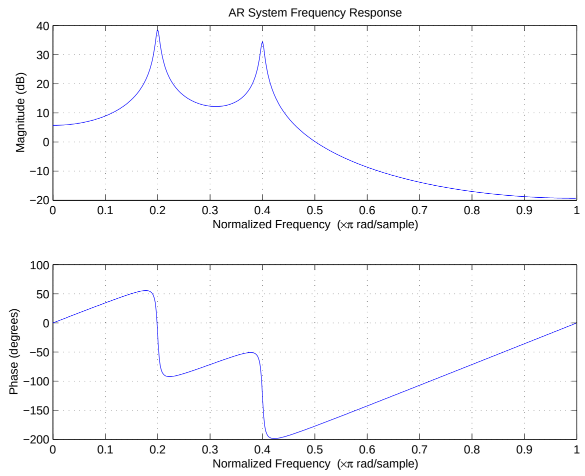
The power spectral density is computed as the distribution of power per unit frequency.

This algorithm depends on your selecting an appropriate model order for your signal.

Examples

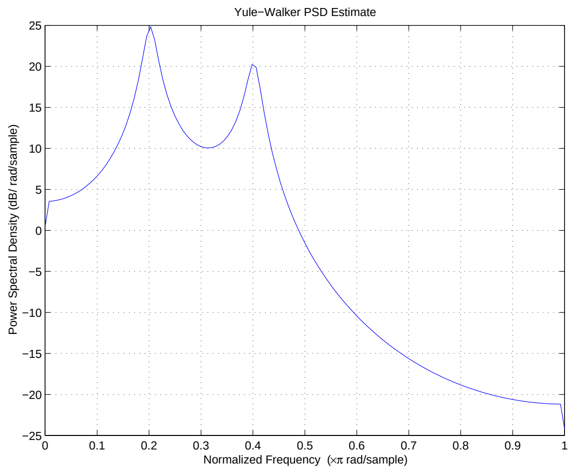
Because the Yule-walker method estimates the spectral density by fitting an AR prediction model of a given order to the signal, first generate a signal from an AR (all-pole) model of a given order. You can use `freqz` to check the magnitude of the frequency response of your AR filter. This will give 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 in put 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.

See Also

`aryule`, `lpc`, `pburg`, `pcov`, `peig`, `periodogram`, `pmcov`, `pmtm`, `pmusic`, `prony`, `psdplot`, `pwelch`

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.

Purpose

Convert reflection coefficients to an 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
```

See Also

`ac2rc`, `poly2ac`, `rc2poly`

References

[1] Kay, S.M., *Modern Spectral Estimation*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

Purpose

Convert reflection coefficients to inverse sine parameters

Syntax

```
isin = rc2is(k)
```

Description

`isin = is2rc(k)` returns a vector of inverse sine parameters `isin` from a vector of reflection coefficients `k`.

Examples

```
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
```

See Also

`is2rc`

References

[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, "*Discrete-Time Processing of Speech Signals*," Prentice-Hall, 1993.

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
```

See Also

`lar2rc`

References

[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, "*Discrete-Time Processing of Speech Signals*," Prentice-Hall, 1993.

Purpose

Convert reflection coefficients to a 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 `length(k)+1`.

`[a,efinal] = rc2poly(k,r0)` returns the final prediction error `efinal` based on the zero-lag autocorrelation, `r0`.

Example

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]`

See Also

`ac2poly`, `latc2tf`, `latcfilt`, `poly2rc`, `rc2ac`, `rc2is`, `rc2lar`, `tf2latc`

References

[1] Kay, S.M., *Modern Spectral Estimation*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

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.

`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))));
```

See Also

`cceps`, `fft`, `hilbert`, `icceps`, `unwrap`

References

[1] Oppenheim, A.V., and R.W. Schafer, *Digital Signal Processing*, Englewood Cliffs, NJ, Prentice-Hall, 1975.

[2] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979.

Purpose Generate a 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`

Purpose Compute a rectangular window

Syntax

```
w = rectwin(n)
```

Description `w = rectwin(n)` returns a rectangular window of length `n` in the column vector `w`. This function is provided for completeness; a rectangular window is equivalent to no window at all.

Algorithm

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

See Also `barthannwin`, `bartlett`, `blackman`, `blackmanharris`, `bohmanwin`, `chebwin`, `gausswin`, `hamming`, `hann`, `kaiser`, `nuttallwin`, `triang`, `tukeywin`, `window`

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

Purpose

Compute the Parks-McClellan optimal FIR filter design

Syntax

```
b = remez(n,f,a)
b = remez(n,f,a,w)
b = remez(n,f,a,'ftype')
b = remez(n,f,a,w,'ftype')
b = remez(...,{lgrid})
b = remez(n,f,'fresp',w)
b = remez(n,f,'fresp',w,'ftype')
b = remez(n,f,{'fresp',p1,p2,...},w)
b = remez(n,f,{'fresp',p1,p2,...},w,'ftype')
[b,delta] = remez(...)
[b,delta,opt] = remez(...)
```

Description

`remez` 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. `remez` exhibits discontinuities at the head and tail of its impulse response due to this equiripple nature.

`b = remez(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

$$b(k) = b(n + 2 - k), \qquad k = 1, \ldots, n + 1$$

Vectors `f` and `a` specify the frequency-magnitude characteristics of the filter:

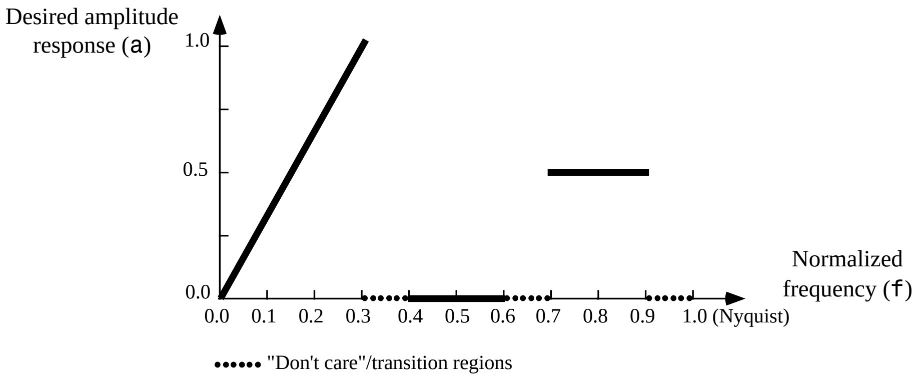
- `f` is a vector of pairs of normalized frequency points, specified in the range between 0 and 1, where 1 corresponds to the Nyquist frequency. The frequencies must be in increasing order.
- `a` is a vector containing the desired amplitudes at the points specified in `f`.

  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 example below.

```
f = [0 .3 .4 .6 .7 .9]
a = [0  1  0  0 .5 .5]
```



`remez` 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`, `remez` increments it by 1.

`remez(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 = remez(n,f,a,'`*`ftype`*`')` and

`b = remez(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 $b(k) = -b(n+2-k)$, $k = 1, ..., n + 1$. This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.

  For example,

  ```
  h = remez(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 = remez(...,{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 = remez(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 specified by function *`fresp`*. The function is called from within `remez` with the following syntax.

```
[dh,dw] = fresp(n,f,gf,w)
```

The arguments are similar to those for `remez`:

- `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 `remez`. `gf` determines the frequency grid at which the response function must be evaluated, and contains the same data returned by `cremez` 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 `remez`; 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`.

The predefined frequency response function *`fresp`* that `remez` calls is `remezfrf` in the `signal/private` directory.

`b = remez(n,f,{'`*`fresp`*`',p1,p2,...},w)` allows you to specify additional parameters (`p1`, `p2`, etc.) to pass to *`fresp`*. Note that `b = remez(n,f,a,w)` is a synonym for `b = remez(n,f,{'remezfrf',a},w)`, where `a` is a vector containing the desired amplitudes at the points specified in `f`.

`b = remez(n,f,'`*`fresp`*`',w,'`*`ftype`*`')` and

`b = remez(n,f,{'`*`fresp`*`',p1,p2,...},w,'`*`ftype`*`')` design antisymmetric (odd) rather than symmetric (even) filters, where `'`*`ftype`*`'` is either `'d'` for a differentiator or `'h'` for a Hilbert transformer.

In the absence of a specification for *`ftype`*, 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 `n`, `f`, `w`, etc., may be used as necessary in determining an appropriate value for `sym`, which `remez` expects to be either `'even'` or `'odd'`. If the *`fresp`* function does not support this calling syntax, `remez` defaults to even symmetry.

`[b,delta] = remez(...)` returns the maximum ripple height in `delta`.

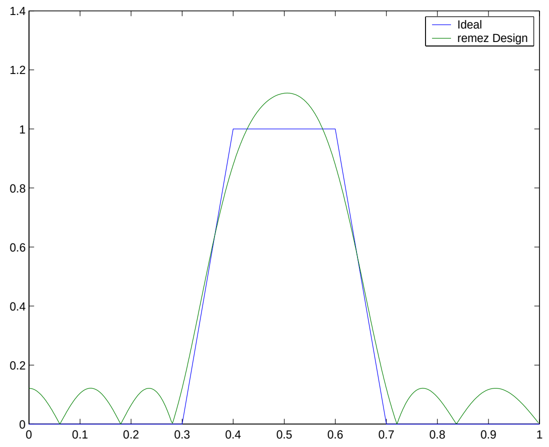
`[b,delta,opt] = remez(...)` returns a structure with the following fields.

`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.des-opt.H`)
`opt.iextr`	Vector of indices into `opt.fgrid` for extremal frequencies
`opt.fextr`	Vector of extremal frequencies

Example

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 = remez(17,f,a);
[h,w] = freqz(b,1,512);
plot(f,a,w/pi,abs(h))
legend('Ideal','remez Design')
```



Algorithm

`remez` is a MEX-file version of the original Fortran code from [1], altered to design arbitrarily long filters with arbitrarily many linear bands.

`remez` 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: $b(k) = b(n+2-k), \quad k = 1, \ldots, n+1$	No restriction	No restriction
Type II	Odd		No restriction	$H(1) = 0$
Type III	Even	odd: $b(k) = -b(n+2-k), \quad k = 1, \ldots, n+1$	$H(0) = 0$	$H(1) = 0$
Type IV	Odd		$H(0) = 0$	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.

See Also

`butter`, `cheby1`, `cheby2`, `cremez`, `ellip`, `fir1`, `fir2`, `fircls`, `fircls1`, `firls`, `firrcos`, `gremez`, `remezord`, `yulewalk`

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. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 256-266.

remezord

Purpose Parks-McClellan optimal FIR filter order estimation

Syntax

```
[n,fo,ao,w] = remezord(f,a,dev)
[n,fo,ao,w] = remezord(f,a,dev,fs)
c = remezord(f,a,dev,fs,'cell')
```

Description `[n,fo,ao,w] = remezord(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 `remez` 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 `remezord` input parameters `f`, `a`, and `dev`.

```
b = remez(n,fo,ao,w)
```

`[n,fo,ao,w] = remezord(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 `remezord` underestimates the order `n`. If the filter does not meet the specifications, try a higher order such as `n+1` or `n+2`.

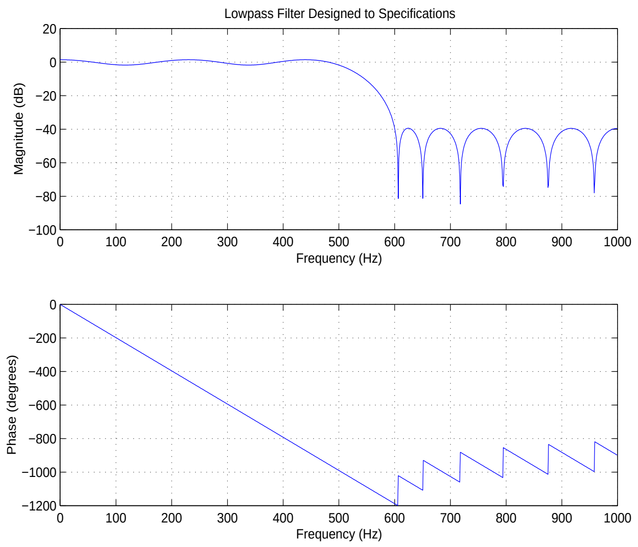
`c = remezord(f,a,dev,fs,'cell')` generates a cell-array whose elements are the parameters to `remez`.

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] = remezord(f,a,dev,fs);
b = remez(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 `remez` 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] = remezord([1500 2000],[1 0],[0.01 0.1],8000 );
b = remez(n,fo,ao,w);
```

This is equivalent to

```
c = remezord( [1500 2000],[1 0],[0.01 0.1],8000,'cell');
b = remez(c{:});
```

Note In some cases, `remezord` 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

`remezord` uses the algorithm suggested in [1]. This method is inaccurate for band edges close to either 0 or the Nyquist frequency (`fs/2`).

See Also

`buttord`, `cheb1ord`, `cheb2ord`, `ellipord`, `kaiserord`, `remez`

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.

Purpose Change sampling rate by any rational factor

Syntax

```
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)
```

Description

`y = resample(x,p,q)` resamples the sequence in vector `x` at `p/q` times the original sampling rate, using a polyphase filter implementation. `p` and `q` must be positive integers. The length of `y` is equal to `ceil(length(x)*p/q)`. If `x` is a matrix, `resample` works down the columns of `x`.

`resample` applies an anti-aliasing (lowpass) FIR filter to `x` during the resampling process. It designs the filter using `firls` with a Kaiser window.

`y = resample(x,p,q,n)` uses `n` terms on either side of the current sample, `x(k)`, to perform the resampling. The length of the FIR filter `resample` uses is proportional to `n`; larger values of `n` provide better accuracy at the expense of more computation time. The default for `n` is 10. If you let `n` = `0`, `resample` performs a nearest-neighbor interpolation

```
y(k) = x(round((k-1)*q/p)+1)
```

where `y(k)` = 0 if the index to `x` is greater than `length(x)`.

`y = resample(x,p,q,n,beta)` uses `beta` as the design parameter for the Kaiser window that `resample` employs in designing the lowpass filter. The default for `beta` is 5.

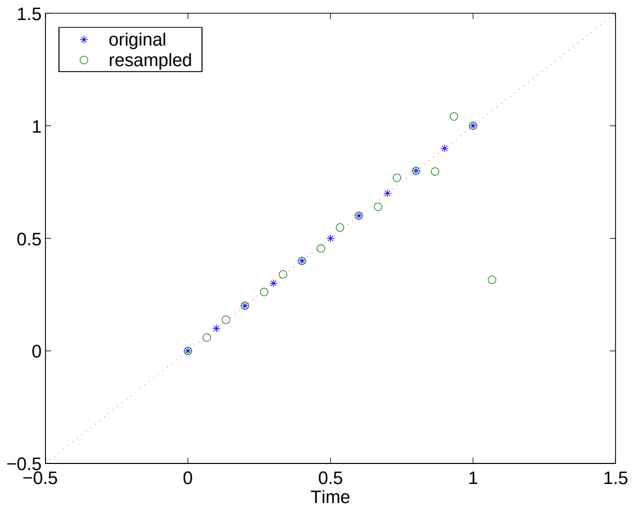
`y = resample(x,p,q,b)` filters `x` using the vector of filter coefficients `b`.

`[y,b] = resample(x,p,q)` returns the vector `b`, which contains the coefficients of the filter applied to `x` during the resampling process.

Examples

Resample a simple linear sequence at 3/2 the original rate.

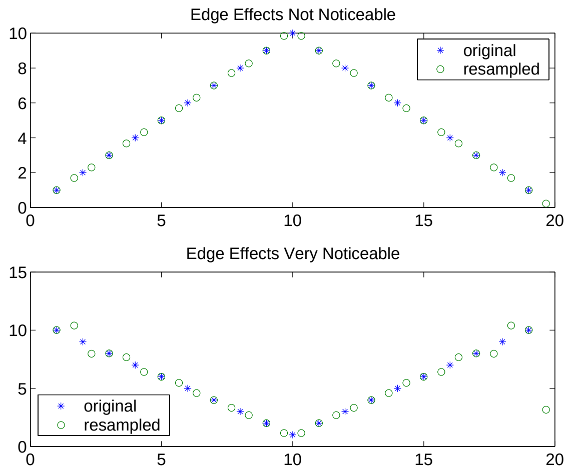
```
fs1 = 10;            % Original sampling frequency in Hz
t1 = 0:1/fs1:1;      % Time vector
x = t1;              % Define a linear sequence
y = resample(x,3,2); % Now resample it

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`, `spline`, `upfirdn`, `upsample`

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.

$$b(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_m z^{-m}$$

$$a(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_n z^{-n}$$

If there are no multiple roots and `a` > `n-1`,

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

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 = length(a)-1 = length(r) = length(p)
```

The direct term coefficient vector `k` is empty if `length(b)` is less than `length(a)`; otherwise

```
length(k) = length(b) - length(a) + 1
```

If `p(j) = ... = p(j+s-1)` is a pole of multiplicity `s`, then the expansion includes terms of the form

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

`[b,a] = 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 `length(b) > length(a)-1`.
- The poles using `p = 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.

  $$\frac{1}{1-pjz^{-1}}, \frac{1}{(1-pjz^{-1})^2}, \dots, \frac{1}{(1-pjz^{-1})^{s_j}}$$

  The vector `h` and matrices `S1` and `S2` have `n + xtra` rows, where `n` is the total number of roots and the internal parameter `xtra`, set to 1 by default, determines the degree of overdetermination of the system of equations.

See Also

`convmtx`, `deconv`, `poly`, `prony`, `residue`, `roots`, `ss2tf`, `tf2ss`, `tf2zp`, `zp2ss`

References

[1] Oppenheim, A.V., and R.W. Schafer, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1975, pp. 166-170.

Purpose

Compute the reverse Levinson-Durbin recursion

Syntax

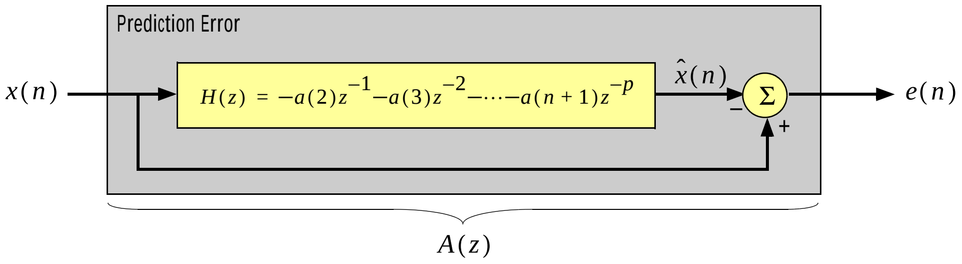
```
r = rlevinson(a,efinal)
[r,u] = rlevinson(a,efinal)
[r,u,k] = rlevinson(a,efinal)
[r,u,k,e] = rlevinson(a,efinal)
```

Description

The reverse Levinson-Durbin recursion implements the step-down algorithm for solving the following symmetric Toeplitz system of linear equations for r, where $r = [r(1)\ \cdots\ r(p+1)]$ and $r(i)^*$ denotes the complex conjugate of $r(i)$.

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

`r = rlevinson(a,efinal)` solves the above system of equations for r given vector `a`, where `a` = $[1\ a(2)\ \cdots\ a(p+1)]$. 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.

$$A(z) = 1 + a(2)z^{-1} + \cdots + a(n+1)z^{-p}$$

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 jth coefficient of the ith 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}.

$$R^{-1} = UE^{-1}U^*$$

See Also

`levinson`, `lpc`, `prony`, `stmcb`

References

[1] Kay, S.M., *Modern Spectral Estimation: Theory and Application*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

Purpose

Estimate frequency and power content using the 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 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`.

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 λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold λ_{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

```
length(pow) = 0.5*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:

- `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`, `rootmusic`

Purpose

Estimate frequency and power content using the 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 λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold λ_{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

```
length(pow) = 0.5*length(w)
```

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`, `rooteig`

sawtooth

Purpose Generate a sawtooth or triangle wave

Syntax

```
x = sawtooth(t)
x = sawtooth(t,width)
```

Description `sawtooth(t)` generates a sawtooth wave with period 2π 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 2π and to increase linearly with time with a slope of 1/π 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 2π at which the maximum occurs. The function increases from -1 to 1 on the interval 0 to 2π*`width`, then decreases linearly from 1 to -1 on the interval 2π*`width` to 2π. Thus a parameter of 0.5 specifies a standard triangle wave, symmetric about time instant π 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`

Purpose

Compute reflection coefficients from an autocorrelation sequence

Syntax

```
k = schurrc(r)
[k,e] = schurrc(r)
```

Description

`k = schurrc(r)` uses the Schur algorithm to compute a vector `k` of reflection coefficients from a vector `r` representing an autocorrelation sequence. `k` and `r` are the same size. The reflection coefficients represent the lattice parameters of a prediction filter for a signal with the given autocorrelation sequence, `r`. When `r` is a matrix, `schurrc` treats each column of `r` as an independent autocorrelation sequence, and produces a matrix `k`, the same size as `r`. Each column of `k` represents the reflection coefficients for the lattice filter for predicting the process with the corresponding autocorrelation sequence `r`.

`[k,e] = schurrc(r)` also computes the scalar `e`, the prediction error variance. When `r` is a matrix, `e` is a row vector. The length of `e` is the same as the number of columns of `r`.

Examples

Create an autocorrelation sequence from the MATLAB speech signal contained in `mtlb.mat`, and use the Schur algorithm to compute the reflection coefficients of a lattice prediction filter for this autocorrelation sequence.

```
load mtlb
r = xcorr(mtlb(1:5),'unbiased');
k = schurrc(r(5:end))

k =
   -0.7583
    0.1384
    0.7042
   -0.3699
```

See Also

`levinson`

References

[1] Proakis, J. and D. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, Third edition, Prentice-Hall, 1996, pp. 868-873.

seqperiod

Purpose

Compute the period of a 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.

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`th order derivative, `xp`, of its middle value from:

```
xp((f+1)/2) = (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.

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.

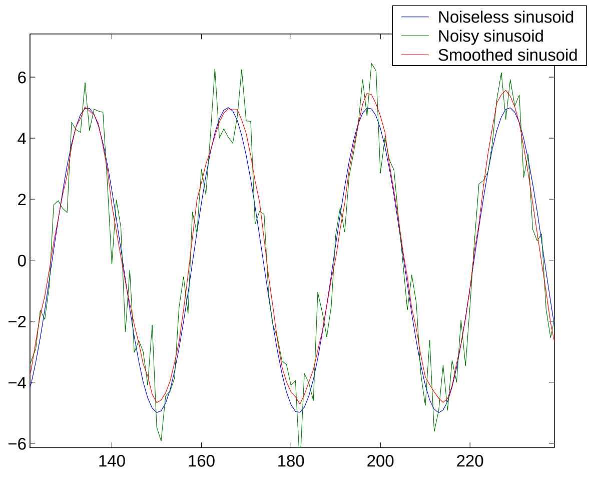
Example

Use sgolay to smooth a noisy sinusoid and display the result and the first and second derivatives.

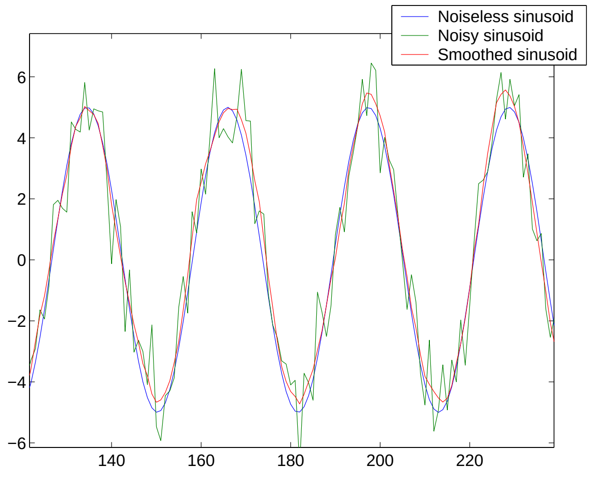
```
N = 4;
F = 21;
[b,g]=sgolay(N,F);
x=5*sin(.4*pi*0:.2:199);
y=x+randn(1,996); % Noisy sinusoid

for n = (F+1)/2:996-(F+1)/2,
% Zero-th order derivative (equivalent to sgolayfilt except
% that it doesn't compute transients)
   z0(n)=g(:,1)'*y(n - (F+1)/2 + 1: n + (F+1)/2 - 1)';
% 1st order derivative
   z1(n)=g(:,2)'*y(n - (F+1)/2 + 1: n + (F+1)/2 - 1)';
% 2nd order derivative
   z2(n)=2*g(:,3)'*y(n - (F+1)/2 + 1: n + (F+1)/2 - 1)';
end

plot([x(1:length(z0))',y(1:length(z0))',z0'])
legend('Noiseless sinusoid','Noisy sinusoid',...
'Smoothed sinusoid')
figure
plot([diff(x(1:length(z0)+1))',z1'])
legend('Noiseless first-order derivative',...
'Smoothed first-order derivative')
figure
plot([diff(diff(x(1:length(z0)+2)))',z2'])
legend('Noiseless second-order derivative',...
'Smoothed second-order derivative')
```

Note The figures below are zoomed to show more detail.

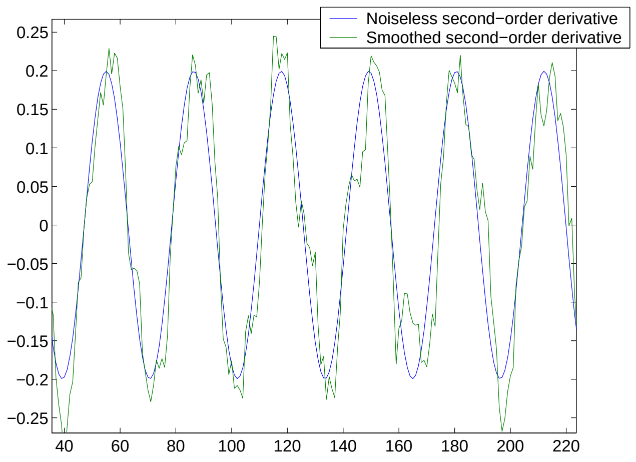
Zero-th order



First derivative



Second derivative



See Also

`fir1`, `firls`, `filter`, `sgolayfilt`

References

[1] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

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.

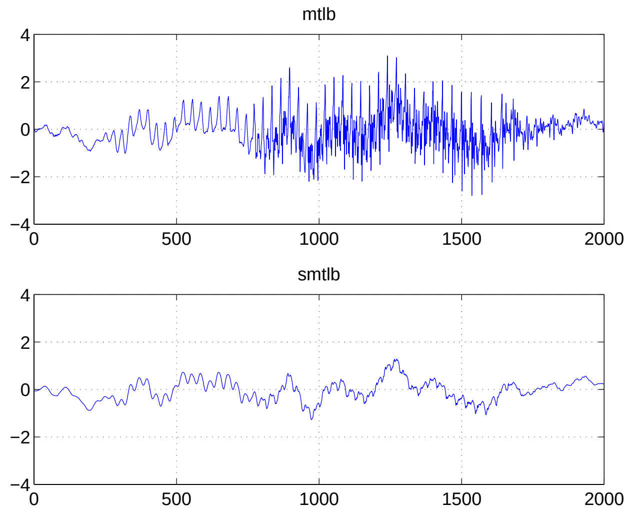
Savitzky-Golay filters are optimal in the sense that they minimize the least-squares error in fitting a polynomial to frames of noisy data.

Example

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;
```



See Also

`medfilt1`, `filter`, `sgolay`, `sosfilt`

References

[1] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

Purpose

Sinc function

Syntax

```
y = sinc(x)
```

Description

`sinc` computes the sinc function of an input vector or array, where the sinc function is

$$\operatorname{sinc}(t) = \begin{cases} 1, & t = 0 \\ \dfrac{\sin(\pi t)}{\pi t}, & t \neq 0 \end{cases}$$

This function is the continuous inverse Fourier transform of the rectangular pulse of width 2π and height 1.

$$\operatorname{sinc}(t) = \frac{1}{2\pi}\int_{-\pi}^{\pi} e^{j\omega t}\,d\omega$$

`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.

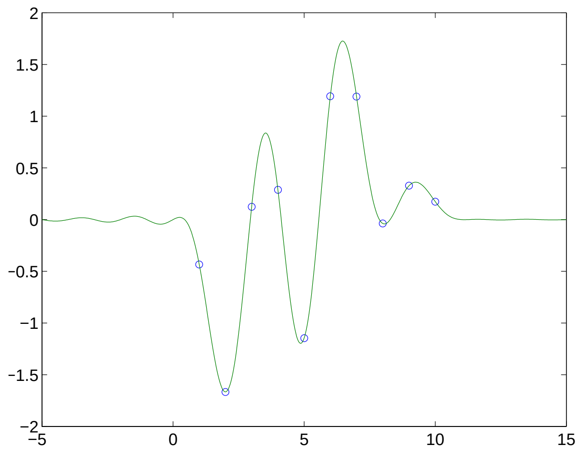
$$g(t) = \sum_{n=-\infty}^{\infty} g(n)\operatorname{sinc}(t-n)$$

Example

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 a second-order section matrix to cell arrays

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 = [b1 a1;b2 a2; ... ;bL aL]
```

where both `bi` and `ai`, with $i = 1, ..., L$, are 1-by-3 row vectors. The resulting `c` is a 1-by-L cell array of cells of the form

```
c = { {b1 a1} {b2 a2} ... {bL aL} }
```

`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} {b1,a1} {b2,a2}...{bL,aL} }
```

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] = sos2ss(sos)
[A,B,C,D] = sos2ss(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] = sos2ss(sos)` converts the system `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

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

`sos` is a L-by-6 matrix organized as

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

The entries of `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] = sos2ss(sos,g)` converts the system `sos` in second-order section form with gain `g`.

$$H(z) = g\prod_{k=1}^{L} H_k(z)$$

Example

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.

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

Row vectors `b` and `a` contain the numerator and denominator coefficients of $H(z)$ stored in descending powers of z.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2z^{-1} + \cdots + b_{n+1}z^{-n}}{a_1 + a_2z^{-1} + \cdots + a_{m+1}z^{-m}}$$

`[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`.

$$H(z) = g\prod_{k=1}^{L} H_k(z)$$

Example

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] = sos2zp(sos)
[z,p,k] = sos2zp(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] = sos2zp(sos)` returns the zeros `z`, poles `p`, and gain `k` of the 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 in its rows.

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

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 `sos`.

`[z,p,k] = sos2zp(sos,g)` returns the zeros `z`, poles `p`, and gain `k` of the system given by `sos` in second-order section form with gain `g`.

$$H(z) = g\prod_{k=1}^{L} H_k(z)$$

Example

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, 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`.

See Also `filter`, `medfilt1`, `sgolayfilt`

References

[1] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

Purpose

Time-dependent frequency analysis (spectrogram)

Syntax

```
B = specgram(a)
B = specgram(a,nfft)
[B,f] = specgram(a,nfft,fs)
[B,f,t] = specgram(a,nfft,fs)
B = specgram(a,nfft,fs,window)
B = specgram(a,nfft,fs,window,numoverlap)
specgram(a)
B = specgram(a,f,fs,window,numoverlap)
```

Description

`specgram` computes the windowed discrete-time Fourier transform of a signal using a sliding window. The spectrogram is the magnitude of this function.

`B = specgram(a)` calculates the windowed discrete-time Fourier transform for the signal in vector `a`. This syntax uses the default values:

- `nfft = min(256,length(a))`
- `fs = 2`
- `window` is a periodic Hann (Hanning) window of length `nfft`.
- `numoverlap = length(window)/2`

`nfft` specifies the FFT length that `specgram` uses. This value determines the frequencies at which the discrete-time Fourier transform is computed. `fs` is a scalar that specifies the sampling frequency. `window` specifies a windowing function and the number of samples `specgram` uses in its sectioning of vector `a`. `numoverlap` is the number of samples by which the sections overlap. Any arguments that you omit from the end of the input parameter list use the default values shown above.

If `a` is real, `specgram` computes the discrete-time Fourier transform at positive frequencies only. If `n` is even, `specgram` returns `nfft/2+1` rows (including the zero and Nyquist frequency terms). If `n` is odd, `specgram` returns `nfft/2` rows. The number of columns in `B` is

```
k = fix((n-numoverlap)/(length(window)-numoverlap))
```

If `a` is complex, `specgram` computes the discrete-time Fourier transform at both positive and negative frequencies. In this case, `B` is a complex matrix with `nfft` rows. Time increases linearly across the columns of B, starting with sample 1 in column 1. Frequency increases linearly down the rows, starting at 0.

B = specgram(a,nfft) uses the specified FFT length nfft in its calculations.

[B,f] = specgram(a,nfft,fs) returns a vector f of frequencies at which the function computes the discrete-time Fourier transform. fs has no effect on the output B; it is a frequency scaling multiplier.

[B,f,t] = specgram(a,nfft,fs) returns frequency and time vectors f and t respectively. t is a column vector of scaled times, with length equal to the number of columns of B. t(j) is the earliest time at which the *j*th window intersects a. t(1) is always equal to 0.

B = specgram(a,nfft,fs,window) specifies a windowing function and the number of samples per section of the x vector. If you supply a scalar for window, specgram uses a Hann window of that length. The length of the window must be less than or equal to nfft.

B = specgram(a,nfft,fs,window,numoverlap) overlaps the sections of x by numoverlap samples.

You can use the empty matrix [] to specify the default value for any input argument. For example,

```
B = specgram(x,[],10000)
```

is equivalent to

```
B = specgram(x)
```

but with a sampling frequency of 10,000 Hz instead of the default 2 Hz.

specgram(...) with no output arguments displays the scaled logarithm of the spectrogram in the current figure window using

```
imagesc(t,f,20*log10(abs(b))), axis xy, colormap(jet)
```

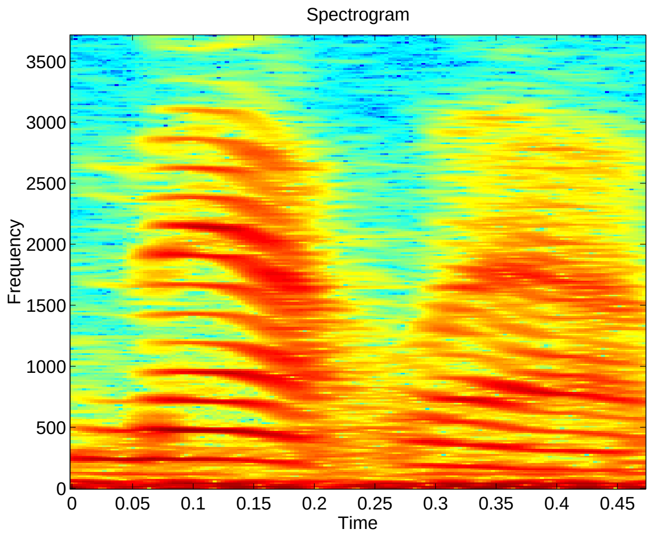
The axis xy mode displays the low-frequency content of the first portion of the signal in the lower-left corner of the axes. specgram uses fs to label the axes according to true time and frequency.

`B = specgram(a,f,fs,window,numoverlap)` computes the spectrogram at the frequencies specified in `f`, using either the chirp *z*-transform (for more than 20 evenly spaced frequencies) or a polyphase decimation filter bank. `f` is a vector of frequencies in hertz; it must have at least two elements.

Example

Display the spectrogram of a digitized speech signal.

```
load mtlb
specgram(mtlb,512,Fs,kaiser(500,5),475)
title('Spectrogram')
```



Note You can view and manipulate a similar spectrogram in the Signal Processing Toolbox Demos, which you access in from the MATLAB Launch Pad.

Algorithm

`specgram` calculates the spectrogram for a given signal as follows:

1 It splits the signal into overlapping sections and applies the window specified by the `window` parameter to each section.

2. It computes the discrete-time Fourier transform of each section with a length `nfft` FFT to produce an estimate of the short-term frequency content of the signal; these transforms make up the columns of `B`. The quantity `(length(window) - numoverlap)` specifies by how many samples `specgram` shifts the window.
3. For real input, `specgram` truncates the spectrogram to the first `nfft/2 + 1` points for `nfft` even and `(nfft + 1)/2` for `nfft` odd.

Diagnostics

An appropriate diagnostic message is displayed when incorrect arguments are used.

```
Requires window's length to be no greater than the FFT length.
Requires NOVERLAP to be strictly less than the window length.
Requires positive integer values for NFFT and NOVERLAP.
Requires vector input.
```

See Also

`cohere`, `csd`, `pwelch`, `tfe`

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 713-718.

[2] Rabiner, L.R., and R.W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

Purpose

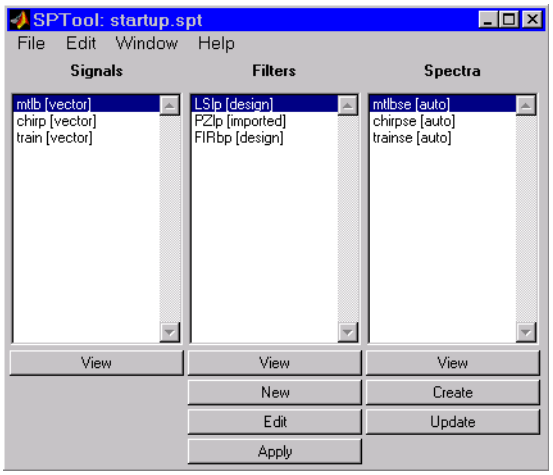
Open the digital signal processing GUI (SPTool)

Syntax

```
sptool
```

Description

`sptool` opens SPTool, a graphical user interface (GUI) that manages a suite of four other GUIs. 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 following GUI opens.



Using SPTool you can:

- Analyze signals listed in the **Signals** list box with the Signal Browser
- Design or edit filters with the Filter Designer (includes a Pole/Zero Editor)
- Analyze filter responses for filters listed in the **Filters** list box with the Filter Viewer
- 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, Filter Viewer, Filter Designer, and Spectrum Viewer

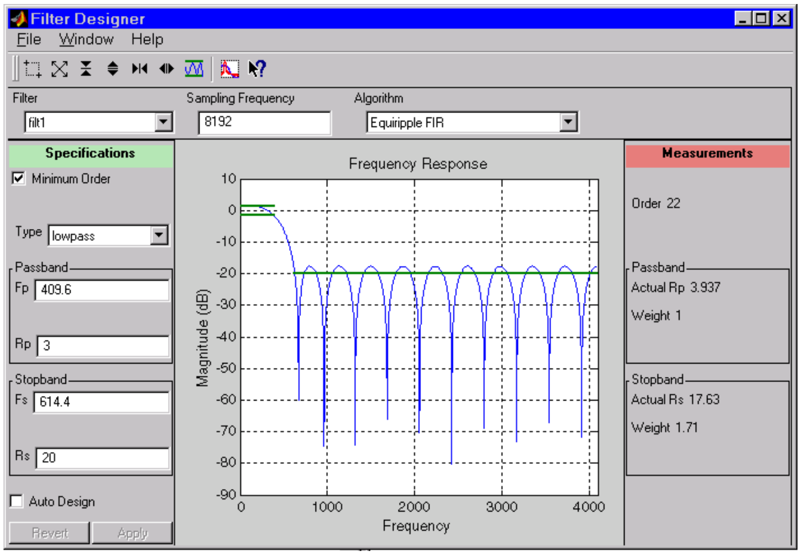
You can activate the four integrated signal processing GUIs from SPTool.

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.

Filter Designer

The Filter Designer allows you to design and edit FIR and IIR filters of various lengths and types, with standard (lowpass, highpass, bandpass, bandstop, and multiband) configurations. To activate the Filter Designer, press either the **New** button or the **Edit** button under the **Filters** list box in SPTool.

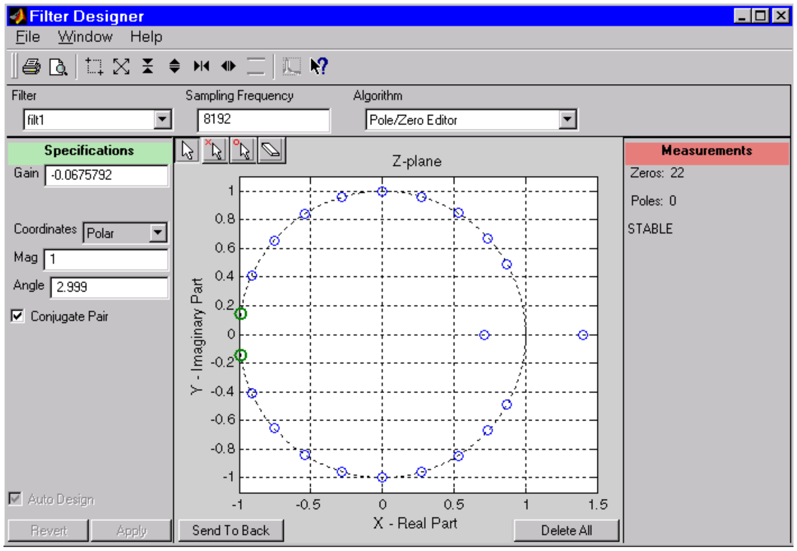


Filter: filt1
Sampling Frequency: 8192
Algorithm: Equiripple FIR

Specifications
Minimum Order ✓
Type: lowpass
Passband Fp: 409.6
Rp: 3
Stopband Fs: 614.4
Rs: 20
Auto Design

Measurements
Order 22
Passband
Actual Rp 3.937
Weight 1
Stopband
Actual Rs 17.63
Weight 1.71

The Filter Designer has a Pole/Zero Editor you can access from the **Algorithms** pulldown.



filt1

8192

Pole/Zero Editor

Gain -0.0675792

Coordinates Polar

Mag 1

Angle 2.999

Conjugate Pair

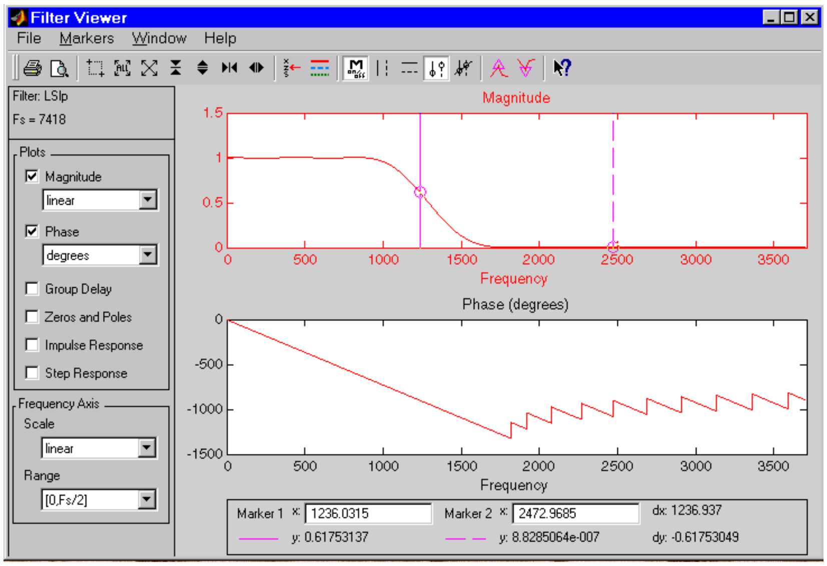
Zeros: 22

Poles: 0

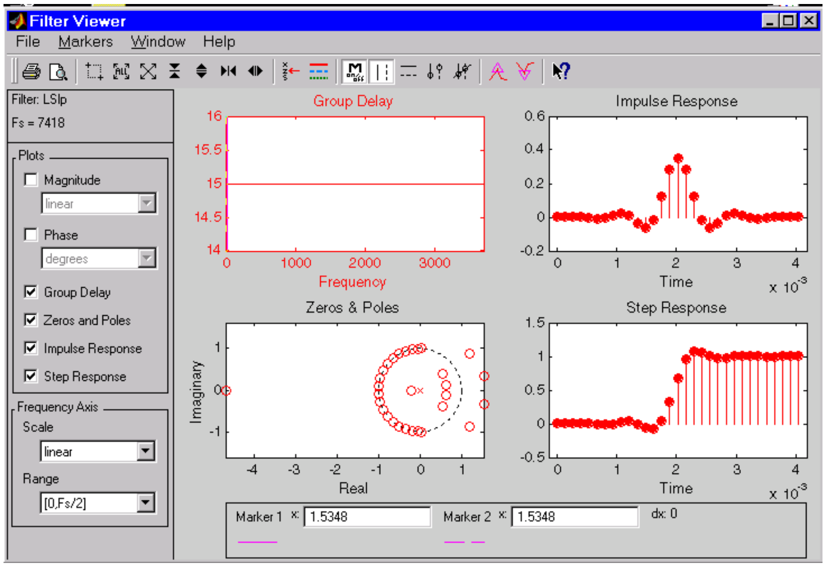
STABLE

Filter Viewer

The Filter Viewer allows you to view the characteristics of a designed or imported filter, including its magnitude response, phase response, group delay, pole-zero plot, impulse response, and step response. To activate the Filter Viewer, press the **View** button under the **Filters** list box in SPTool.



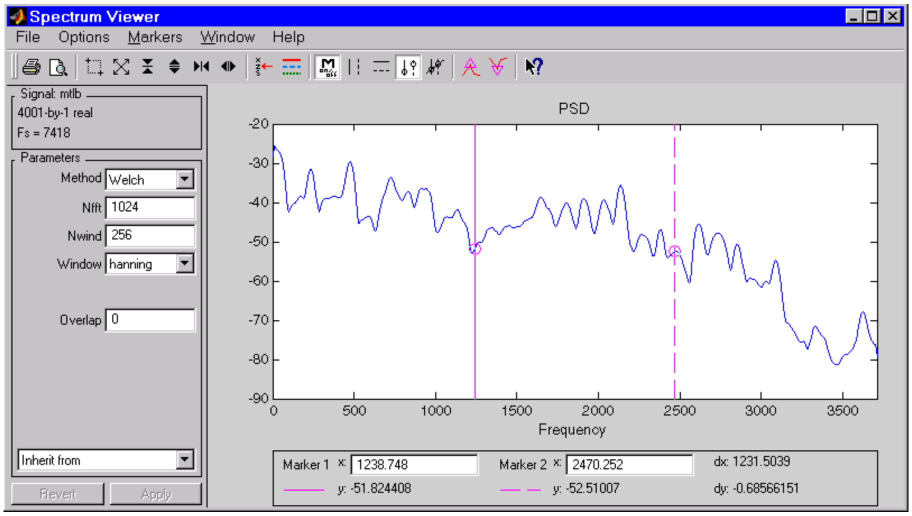
You can analyze multiple filter responses by checking several response plot options, as shown in the next figure.



Spectrum Viewer

The Spectrum Viewer allows you to graphically analyze frequency-domain data using a variety of methods of spectral density estimation, including the Burg method, the FFT method, the multitaper method (MTM), the MUSIC eigenvector method, Welch's method, and the Yule-Walker AR method. To activate the Spectrum Viewer:

- Press 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 press **Apply** to view the spectra.
- Press the **View** button to analyze spectra selected under the **Spectra** list box in SPTool.
- Press the **Update** button under the **Spectra** list box in SPTool to modify a selected power spectral density signal.



Signal: mtlb
4001-by-1 real
Fs = 7418

Parameters
Method: Welch
Nfft: 1024
Nwind: 256
Window: hanning
Overlap: 0

Marker 1 x: 1238.748 y: -51.824408
Marker 2 x: 2470.252 y: -52.51007
dx: 1231.5039
dy: -0.68566151

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`

Purpose Generate a square wave

Syntax

```
x = square(t)
x = square(t,duty)
```

Description `x = square(t)` generates a square wave with period 2π for the elements of time vector `t`. `square(t)` is similar to `sin(t)`, but creates a square wave with peaks of ±1 instead of a sine wave.

`x = square(t,duty)` generates a square wave with specified duty cycle, `duty`. 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.

`sos = ss2sos(...)` embeds the overall system gain, `g`, in the first section, $H_1(z)$, so that

$$H(z) = \prod_{k=1}^{L} H_k(z)$$

Example

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 ∞ 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.
```

See Also

cplxpair, sos2ss, ss2tf, ss2zp, tf2sos, zp2sos

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.

ss2tf

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

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

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

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

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.

Example

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.

See Also

pzmap, sos2zp, ss2sos, ss2tf, tf2zp, zp2ss

References

[1] Laub, A.J., and B.C. Moore, “Calculation of Transmission Zeros Using QZ Techniques,” *Automatica* 14 (1978), p. 557.

Purpose

Compute a 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`.

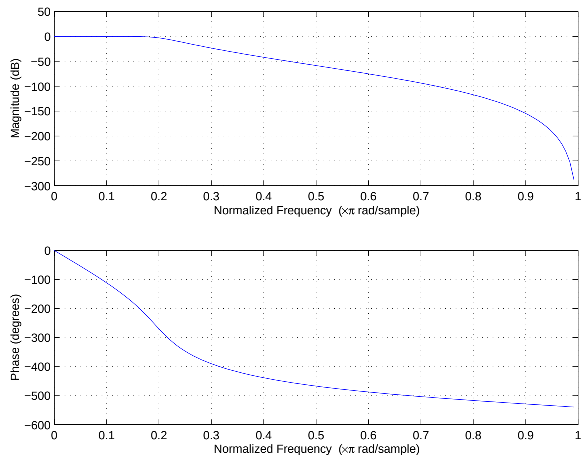
`stmcb` returns the IIR filter coefficients in length `nb+1` and `na+1` row vectors `b` and `a`. The filter coefficients are ordered in descending powers of z.

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

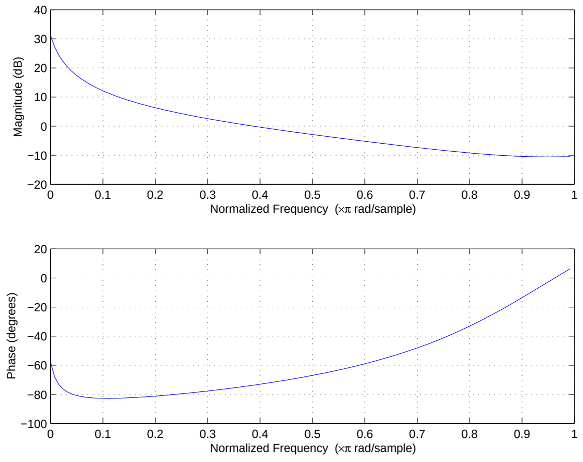
Example

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 `\`.

`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.
```

See Also

`levinson`, `lpc`, `aryule`, `prony`

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.

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.

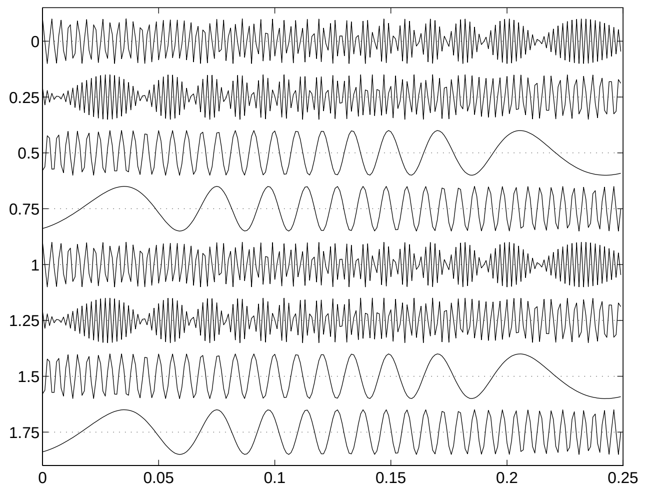
If `x` is a matrix, `strips(x,n)`, `strips(x,sd,fs)`, and `strips(x,sd,fs,scale)` plot the different columns of `x` on the same strip plot.

`strips` ignores the imaginary part of complex-valued `x`.

Example

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`

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`

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`.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \cdots + b_{n+1} z^{-n}}{a_1 + a_2 z^{-1} + \cdots + a_{m+1} z^{-m}}$$

`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.

`sos = tf2sos(...)` embeds the overall system gain, `g`, in the first section, $H_1(z)$, so that

$$H(z) = \prod_{k=1}^{L} H_k(z)$$

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 ∞ 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.

See Also

`cplxpair`, `sos2tf`, `ss2sos`, `tf2ss`, `tf2zp`, `zp2sos`

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.

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

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

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.

Example

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`, `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 transfer function.

`[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

$$H(s) = \frac{B(s)}{A(s)} = \frac{b_1 s^{n-1} + \cdots + b_{n-1}s + b_n}{a_1 s^{m-1} + \cdots + a_{m-1}s + a_m}$$

or

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} \cdots + b_{n-1}z^{-n} + b_n z^{-n-1}}{a_1 + a_2 z^{-1} \cdots + a_{m-1}z^{-m} + a_m z^{-m-1}}$$

you can use the output of `tf2zp` to produce the single-input, multioutput (SIMO) factored transfer function form

$$H(s) = \frac{Z(s)}{P(s)} = k\frac{(s-z_1)(s-z_2)\cdots(s-z_m)}{(s-p_1)(s-p_2)\cdots(s-p_n)}$$

or

$$H(z) = \frac{Z(z)}{P(z)} = k\frac{(z-z_1)(z-z_2)\cdots(z-z_m)}{(z-p_1)(z-p_2)\cdots(z-p_n)}$$

The following describes the input and output arguments for `tf2zp`:

- The vector `a` specifies the coefficients of the denominator polynomial $A(s)$ (or $A(z)$) in descending powers of s (z^{-1}).
- The ith row of the matrix `b` represents the coefficients of the ith numerator polynomial (the ith 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.

Example

Find the zeros, poles, and gains of the discrete-time system

$$H(z) = \frac{2 + 3z^{-1}}{1 + 0.4z^{-1} + z^{-2}}$$

```
b = [2 3];
a = [1 0.4 1];

[b,a] = eqtflength(b,a);       % 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`, `zp2tf`

Purpose

Estimate the transfer function from input and output

Syntax

```
Txy = tfe(x,y)
Txy = tfe(x,y,nfft)
[Txy,f] = tfe(x,y,nfft,fs)
Txy = tfe(x,y,nfft,fs,window)
Txy = tfe(x,y,nfft,fs,window,numoverlap)
Txy = tfe(x,y,...,'dflag')
tfe(x,y)
```

Description

`Txy = tfe(x,y)` finds a transfer function estimate `Txy` given input signal vector `x` and output signal vector `y`. The *transfer function* is the quotient of the cross spectrum of `x` and `y` and the power spectrum of `x`.

$$T_{xy}(f) = \frac{P_{xy}(f)}{P_{xx}(f)}$$

The relationship between the input `x` and output `y` is modeled by the linear, time-invariant transfer function `Txy`.

Vectors `x` and `y` must be the same length. `Txy = tfe(x,y)` uses the following default values:

- `nfft = min(256,(length(x))`
- `fs = 2`
- `window` is a periodic Hann (Hanning) window of length `nfft`
- `numoverlap = 0`

`nfft` specifies the FFT length that `tfe` uses. This value determines the frequencies at which the power spectrum is estimated. `fs` is a scalar that specifies the sampling frequency. `window` specifies a windowing function and the number of samples `tfe` uses in its sectioning of the `x` and `y` vectors. `numoverlap` is the number of samples by which the sections overlap. Any arguments that are omitted from the end of the parameter list use the default values shown above.

If `x` is real, `tfe` 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, `tfe` estimates the transfer function for both positive and negative frequencies and `Txy` has length `nfft`.

`Txy = tfe(x,y,nfft)` uses the specified FFT length `nfft` in estimating the transfer function.

`[Txy,f] = tfe(x,y,nfft,fs)` returns a vector `f` of frequencies at which `tfe` estimates the transfer function. `fs` is the sampling frequency. `f` is the same size as `Txy`, so `plot(f,Txy)` plots the transfer function estimate versus properly scaled frequency. `fs` has no effect on the output `Txy`; it is a frequency scaling multiplier.

`Txy = tfe(x,y,nfft,fs,window)` specifies a windowing function and the number of samples per section of the `x` vector. If you supply a scalar for `window`, `Txy` uses a Hann window of that length. The length of the window must be less than or equal to `nfft`; `tfe` zero pads the sections if the length of the window exceeds `nfft`.

`Txy = tfe(x,y,nfft,fs,window,numoverlap)` overlaps the sections of `x` by `numoverlap` samples.

You can use the empty matrix `[]` to specify the default value for any input argument except `x` or `y`. For example,

```
Txy = tfe(x,y,[],[],kaiser(128,5))
```

uses 256 as the value for `nfft` and 2 as the value for `fs`.

`Txy = tfe(x,y,...,'`*`dflag`*`')` specifies a detrend option, where `'`*`dflag`*`'` is:

- `'linear'`, to remove the best straight-line fit from the prewindowed sections of `x` and `y`
- `'mean'`, to remove the mean from the prewindowed sections of `x` and `y`
- `'none'`, for no detrending (default)

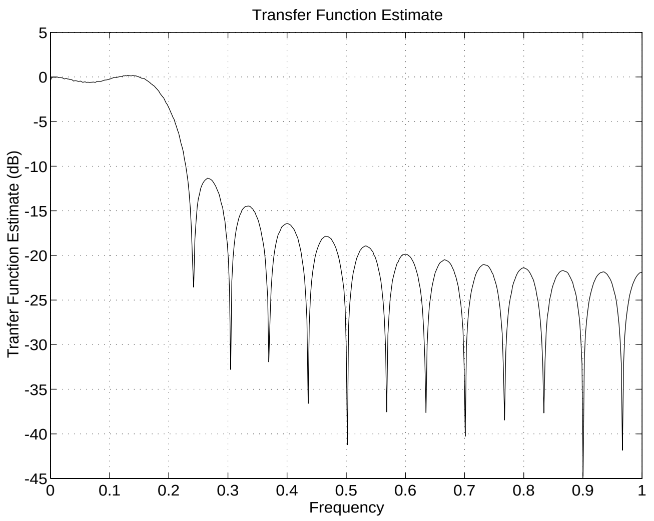
The `'`*`dflag`*`'` parameter must appear last in the list of input arguments. `tfe` recognizes a `'`*`dflag`*`'` string no matter how many intermediate arguments are omitted.

`tfe(...)` with no output arguments plots the magnitude of the transfer function estimate as decibels versus frequency in the current figure window.

Example

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);
tfe(x,y,1024,[],[],512)
title('Transfer Function Estimate')
```



Algorithm

`tfe` uses a four-step algorithm:

1. It multiplies the detrended sections by `window`.
2. It takes the length `nfft` FFT of each section.
3. It averages the squares of the spectra of the `x` sections to form `Pxx` and averages the products of the spectra of the `x` and `y` sections to form `Pxy`.
4. It calculates `Txy`:

   ```
   Txy = Pxy./Pxx
   ```

Diagnostics

An appropriate diagnostic message is displayed when incorrect arguments are used.

```
Requires window's length to be no greater than the FFT length.
Requires NOVERLAP to be strictly less than the window length.
Requires positive integer values for NFFT and NOVERLAP.
Requires vector (either row or column) input.
Requires inputs X and Y to have the same length.
```

See Also

`etfe`, `cohere`, `csd`, `pwelch`, `spa`

Purpose

Compute a triangular window

Syntax

```
w = triang(n)
```

Description

`triang(n)` returns an `n`-point triangular window in the column vector `w`. The coefficients of a triangular window are

For `n` odd:

$$w[k] = \begin{cases} \dfrac{2k}{n+1}, & 1 \le k \le \dfrac{n+1}{2} \\ \dfrac{2(n-k+1)}{n+1}, & \dfrac{n+1}{2} \le k \le n \end{cases}$$

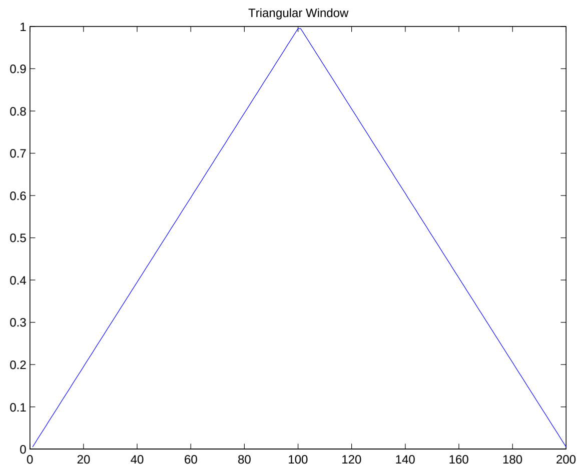
For `n` even:

$$w[k] = \begin{cases} \dfrac{2k-1}{n}, & 1 \le k \le \dfrac{n}{2} \\ \dfrac{2(n-k+1)}{n}, & \dfrac{n}{2}+1 \le k \le n \end{cases}$$

The triangular window is very similar to a Bartlett window. The Bartlett window always ends with zeros at samples 1 and `n`, while the triangular window is nonzero at those points. For `n` odd, the center `n-2` points of `triang(n-2)` are equivalent to `bartlett(n)`.

Example

```
N=200;
w = triang(200);
plot(w);
title('Triangular Window')
```



See Also

`bartlett`, `barthannwin`, `blackman`, `blackmanharris`, `bohmanwin`, `chebwin`, `gausswin`, `hamming`, `hann`, `kaiser`, `nuttallwin`, `rectwin`, `tukeywin`, `window`

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 447-448.

tripuls

Purpose

Generate a 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`

tukeywin

Purpose

Compute a Tukey window

Syntax

`w = tukeywin(n,α)`

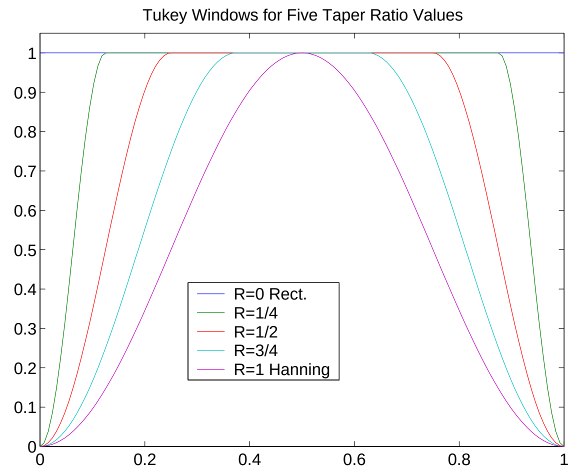
Description

`w = tukeywin(n,r)` returns an n-point, Tukey window in column vector `w`. Tukey windows are cosine-tapered windows. α is the ratio of taper to constant sections and is between 0 and 1 define. $\alpha \leq 0$ is a `rectwin` window and $\alpha \geq 1$ is a `hann` window.

Example

Compute 128-point Tukey windows with five different tapers.

```
wtuk = [];
for i = linspace(0,1,5)
    wtuk = [wtuk tukeywin(128,i)];
end
plot(linspace(0,1,128),wtuk)
axis([0 1 0 1.05 ] );
legend('R=0 Rect.','R=1/4','R=1/2','R=3/4','R=1 Hanning',0);
title('Tukey Windows for Five Taper Ratio Values');
```



Algorithm

The equation for computing the coefficients of a Tukey window is

$$w[k+1] = \begin{cases} 1.0, & 0 \le |k| \le \dfrac{N}{2}(1+\alpha) \\ 0.5\left[1.0 + \cos\left[\pi \dfrac{k - \frac{N}{2}(1+\alpha)}{N(1-\alpha)}\right]\right], & \dfrac{N}{2}(1+\alpha) \le |k| \le N \end{cases}$$

See Also

`barthannwin`, `bartlett`, `blackman`, `blackmanharris`, `bohmanwin`, `chebwin`, `gausswin`, `hann`, `hamming`, `kaiser`, `nuttallwin`, `rectwin`, `triang`, `window`

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.

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:
  - 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 = 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 = 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.

See Also uencode

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.

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].

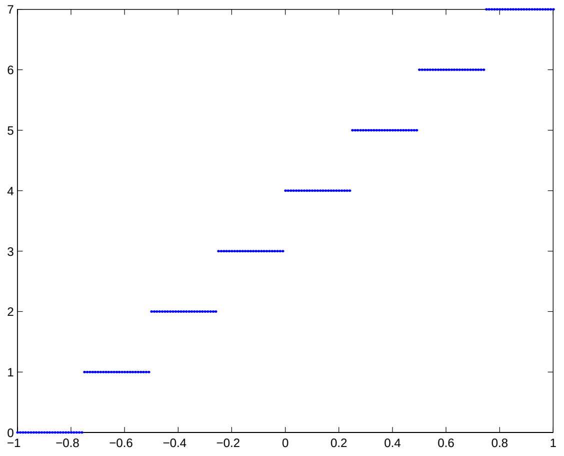
The output data types are optimized for the number of bits as shown in the table below.

n	Unsigned Integer	Signed Integer
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)

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 =
    -8    -4     0     4     8    12    16    20    23
```

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.

See Also

`udecode`

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.

unwrap

Purpose Unwrap phase angles

Syntax

```
p = unwrap(p)
```

Description `p = unwrap(p)` corrects the phase angles in vector `p` by adding multiples of $\pm 2\pi$, where needed, to smooth the transitions across branch cuts. When `p` is a matrix, `unwrap` corrects the phase angles down each column. The phase must be in radians.

The `unwrap` function is part of the standard MATLAB language.

Limitations `unwrap` tries to detect branch cut crossings, but it can be fooled by sparse, rapidly changing phase values.

See Also `abs`, `angle`

upfirdn

Purpose Upsample, apply an FIR filter, and downsample

Syntax

```
yout = upfirdn(xin,h)
yout = upfirdn(xin,h,p)
yout = upfirdn(xin,h,p,q)
[yout,zf] = upfirdn(xin,h,...,zi)
```

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 `remez` 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.

`[yout,zf] = upfirdn(xin,h,...,zi)` specifies initial state conditions in the vector `zi`.

The size of the initial condition vector `zi` must be the same as `length(h)-1`, the number of delays in the FIR filter.

When `xin` is a vector, the size of the final condition vector `zf` is `length(h)-1`, the number of delays in the filter. When `xin` is a matrix, `zf` is an matrix with `(length(h)-1)` rows and `(size(xin,2))` columns.

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 an 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 an 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 an matrix with the same number of columns as `xin` and `h`.

Examples

If both `p` and `q` are equal to 1 (that is, there is no rate changing), the result is ordinary convolution of two signals (equivalent to `conv`).

```
yy = upfirdn(xx,hh);
```

This example implements a seven-channel filter bank by convolving seven different filters with one input signal, then downsamples by five.

```
% Assume that hh is an L-by-7 array of filters.
yy = upfirdn(xx,hh,1,5);
```

Implement a rate change from 44.1 kHz (CD sampling rate) to 48 kHz (DAT rate), a ratio of 160/147. This requires a lowpass filter with cutoff frequency at $\omega_c = 2\pi/160$.

```
% Design lowpass filter with cutoff at 1/160th of fs.

hh = fir1(300,2/160);          % Need a very long lowpass filter
yy = upfirdn(xx,hh,160,147);
```

In this example, the filter design and resampling are separate steps. Note that `resample` would do both steps as one.

Algorithm

`upfirdn` uses a polyphase interpolation structure. The number of multiply-add operations in the polyphase structure is approximately $(L_h L_x - pL_x)/q$ 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.

See Also `conv`, `decimate`, `downsample`, `filter`, `interp`, `intfilt`, `resample`, `upsample`

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.

Purpose

Increase the sampling rate of the input signal

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

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 ±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`.

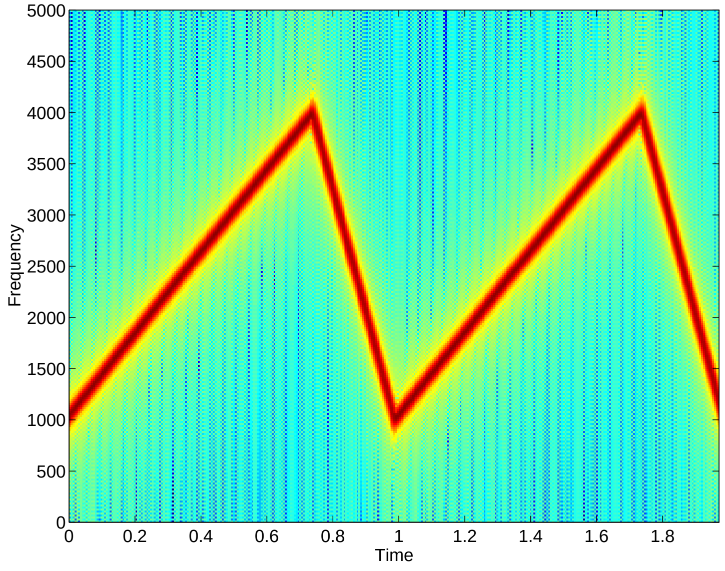
Example

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.

```
specgram(x,512,fs,kaiser(256,5),220)
```



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`

window

Purpose Compute a specific window

Syntax

```
w = window(fhandle,n)
w = window(fhandle,n,winopt)
w = window(fhandle,n,sidelobe)
```

Description

`w = window(fhandle,n)` returns the `n`-point window, specified by its function handle, `fhandle`, in column vector `w`. Function handles are window function names preceded by an @.

`@barthannwin`	`@hamming`
`@bartlett`	`@hann`
`@blackman`	`@kaiser`
`@blackmanharris`	`@nuttallwin`
`@bohmanwin`	`@rectwin`
`@chebwin`	`@triang`
`@gausswin`	`@tukeywin`

Note For more information on each window function and its option(s), refer to its reference page.

`w = window(fhandle,N,`*`winopt`*`)` returns the window specified by its function handle, `fhandle`, and its option value or string in *`winopt`*. Only `blackman`, `chebwin`, `gausswin`, `hamming`, `hann`, and `tukeywin` windows take an option.

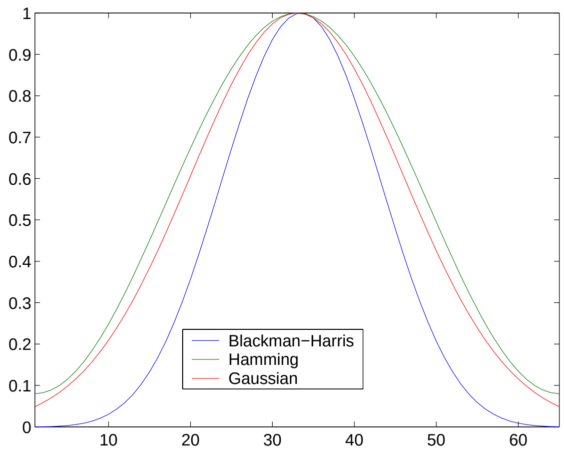
`w = window(fhandle,N,sidelobe)` returns the window specified by its function handle, `fhandle`. If `fhandle` is either `@chebwin` or `@kaiser`, a second numerical parameter must be entered. For Chebyshev `sidelobe` is the number of decibels that the sidelobe magnitude is below the mainlobe; for Kaiser `sidelobe` is the beta value.

Example

Create and plot three windows.

```
N = 65;
```

```
w = window(@blackmanharris,N);
w1 = window(@hamming,N);
w2 = window(@gausswin,N,2.5);
plot(1:N,[w,w1,w2]); axis([1 N 0 1]);
legend('Blackman-Harris','Hamming','Gaussian');
```



See Also

`barthannwin`, `bartlett`, `blackman`, `blackmanharris`, `bohmanwin`, `chebwin`, `gausswin`, `hamming`, `hann`, `kaiser`, `nuttallwin`, `rectwin`, `triang`, `tukeywin`

Purpose

Estimate the cross-correlation function

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 $c(m) = c_{xy}(m\text{-}N)$, m=1, ..., 2N-1.

In general, the correlation function requires normalization to produce an accurate estimate (see below).

`c = 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 = xcorr(x,y,'`*`option`*`')` specifies a normalization option for the cross-correlation, where `'`*`option`*`'` is:

- `'biased'`: Biased estimate of the cross-correlation function

$$c_{xy,\,biased}(m) = \frac{1}{N}c_{xy}(m)$$

- `'unbiased'`: Unbiased estimate of the cross-correlation function

$$c_{xy,\,unbiased}(m) = \frac{1}{N-|m|}c_{xy}(m)$$

- `'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 = xcorr(x,'`*`option`*`')` specifies one of the above normalization options for the autocorrelation.

`c = xcorr(x,y,maxlags)` returns the cross-correlation sequence over the lag range `[-maxlags:maxlags]`. Output `c` has length `2*maxlags+1`.

`c = 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 = 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 $-10 \le m \le 10$ 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)
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].

See Also

`conv`, `corrcoef`, `cov`, `xcorr2`, `xcov`

References

[1] Orfanidis, S.J., *Optimum Signal Processing. An Introduction. 2nd Edition*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

xcorr2

Purpose Estimate the two-dimensional cross-correlation

Syntax

```
C = xcorr2(A)
C = xcorr2(A,B)
```

Description `C = xcorr2(A,B)` returns the cross-correlation of matrices `A` and `B` with no scaling. `xcorr2` is the two-dimensional version of `xcorr`. It has its maximum value when the two matrices are aligned so that they are shaped as similarly as possible.

`xcorr2(A)` is the autocorrelation matrix of input matrix `A`. It is identical to `xcorr2(A,A)`.

See Also `conv2`, `filter2`, `xcorr`

Purpose Estimate the cross-covariance function (mean-removed cross-correlation)

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 μ_x and μ_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 = 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 = xcov(x)` is the autocovariance sequence for the vector `x`. Where `x` is an N-by-P array, `v = 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

$$c_{xy}(m) = \begin{cases} \displaystyle\sum_{n=0}^{N-|m|-1}\left(x(n+m)-\frac{1}{N}\sum_{i=0}^{N-1}x_i\right)\left(y_n^*-\frac{1}{N}\sum_{i=0}^{N-1}y_i^*\right) & m \geq 0 \\ c_{yx}^*(-m) & m < 0 \end{cases}$$

The output vector `c` has elements given by `c(m)` = $c_{xy}(m\text{-}N)$, m = 1, ..., $2N$-1.

The covariance function requires normalization to estimate the function properly.

`v = 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,lags] = xcov(x,y,maxlags)` where `x` and `y` are length `m` vectors, returns the cross-covariance sequence in a length `2*maxlags+1` vector `c`. `lags` is a vector of the lag indices where `c` was estimated, that is, `[-maxlags:maxlags]`.

`[c,lags] = xcov(x,maxlags)` is the autocovariance sequence over the range of lags `[-maxlags:maxlags]`.

`[c,lags] = xcov(x,maxlags)` where `x` is an `m`-by-`p` array, returns array `c` with `2*maxlags+1` rows whose P^2 columns contain the cross-covariance sequences for all combinations of the columns of `x`.

`[c,lags] = xcov(x,y,maxlags,'option')` specifies a scaling option, where `'option'` is the last input argument.

In all cases, `xcov` gives an output such that the zeroth lag of the covariance vector is in the middle of the sequence, at element or row `maxlag+1` or at `m`.

Examples

The second output `lags` is useful when plotting. For example, the estimated autocovariance of uniform white noise $c_{ww}(m)$ can be displayed for $-10 \le m \le 10$ using

```
ww = randn(1000,1);  % Generate uniform noise with mean = 1/2.
[cov_ww,lags] = xcov(ww,10,'coeff');
stem(lags,cov_ww)
```

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.

See Also

conv, corrcoef, cov, xcorr, xcorr2

References

[1] Orfanidis, S.J., *Optimum Signal Processing. An Introduction. 2nd Edition*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

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.

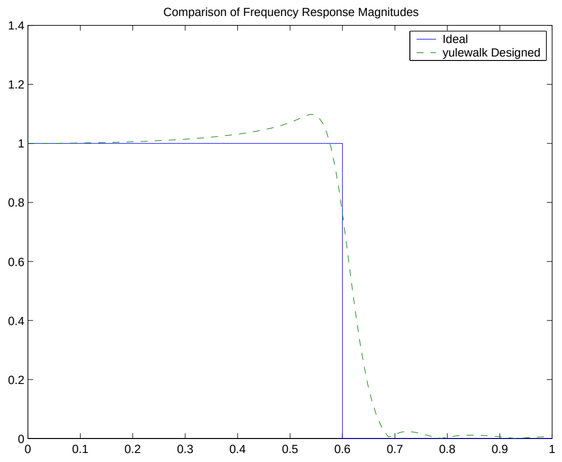
$$\frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}$$

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.

Example

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.

See Also

`butter`, `cheby1`, `cheby2`, `ellip`, `fir2`, `firls`, `maxflat`, `remez`

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.

Purpose

Convert digital filter zero-pole-gain 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 = 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.

`sos = zp2sos(...)` embeds the overall system gain, `g`, in the first section, $H_1(z)$, so that

$$H(z) = \prod_{k=1}^{L} H_k(z)$$

Example

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 ∞ 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.

See Also

`cplxpair`, `filternorm`, `sos2zp`, `ss2sos`, `tf2sos`, `zp2ss`, `zp2tf`

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.

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

$$\frac{B(s)}{A(s)} = \frac{b_1 s^{(n-1)} + \cdots + b_{(n-1)} s + b_n}{a_1 s^{(m-1)} + \cdots + a_{(m-1)} s + a_m}$$

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`, `zp2sos`, `zp2ss`

Purpose Zero-pole plota

Syntax

```
zplane(z,p)
zplane(b,a)
[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.

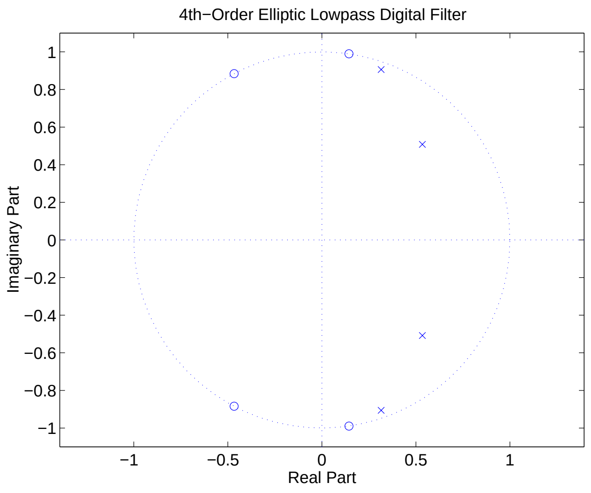
`zplane(b,a)` where `b` and `a` are row vectors, first uses `roots` to find the zeros and poles of the transfer function represented by numerator coefficients `b` and denominator coefficients `a`.

`[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 the following commands.

```
[b,a] = ellip(4,3,30,200/500);   % Transfer function
zplane(b,a)
```

See Also

`freqz`

Index