Signal Processing Toolbox™
User's Guide

R2011b

MATLAB®

How to Contact MathWorks

www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	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

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

Signal Processing Toolbox™ User's Guide

Trademarks

Patents

Revision History

Contents

Filtering, Linear Systems and Transforms Overview

1

Filter Design and Implementation

2

Designing a Filter in Fdesign — Process Overview

3

Designing a Filter in the Filterbuilder GUI

4

FDATool: A Filter Design and Analysis GUI

5

Statistical Signal Processing

6

Special Topics

7

SPTool: A Signal Processing GUI Suite

8

Code Generation from MATLAB Support in Signal Processing Toolbox

9

Technical Conventions

A

Index

1

Filtering, Linear Systems and Transforms Overview

Filter Implementation and Analysis

In this section...

Filtering Overview

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

Convolution and Filtering

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

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

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

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

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

If a digital filter's impulse response $h(k)$ is finite in length, and the input $x(k)$ is also of 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);
```

The length of the output is the sum of the finite-length input vectors minus 1.

Filters and Transfer Functions

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

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

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

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

MATLAB filter functions store the coefficients in two vectors, one for the numerator and one for the denominator. By convention, it 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.

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

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

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

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

$$y(1) = b(1)x(1)$$
$$y(2) = b(1)x(2) + b(2)x(1) - a(2)y(1)$$
$$y(3) = b(1)x(3) + b(2)x(2) + b(3)x(1) - a(2)y(2) - a(3)y(1)$$
$$\vdots \quad = \vdots$$

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. Note that the A coefficient vectors are written as if the output and input terms are separated in the difference equation. For the example, the previous coefficient vectors represent a linear constant-coefficient difference equation of

$$y(n) - 0.9y(n-1) = x(n)$$

Changing the sign of the `A(2)` coefficient, results in the difference equation

$$y(n) + 0.9y(n-1) = x(n)$$

The previous coefficients are represented as:

```
B = 1; %Numerator
A = [1 0.9]; %Denominator
```

and results in a highpass filter.

To apply this filter to your data, use

```
y = filter(B,A,x);
```

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

The filter Function

filter is implemented as the transposed direct-form II structure, where n-1 is the filter order. This is a canonical form that has the minimum number of delay elements.
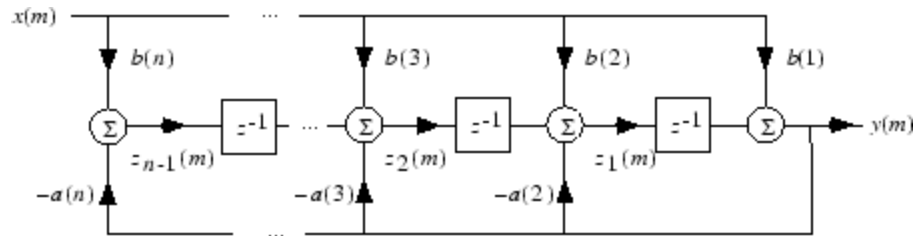
At sample m, filter computes the difference equations

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

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

Multirate Filter Bank Implementation

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

- Upsampling (zero insertion) by integer factor p

- Filtering by FIR filter h

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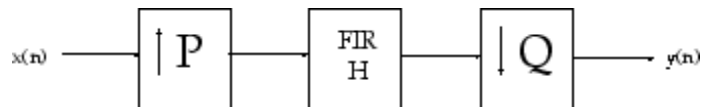
For example, to change the sample rate of a signal from 44.1 kHz to 48 kHz, we first find the smallest integer conversion ratio p/q. Set

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

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

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

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

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

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

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.
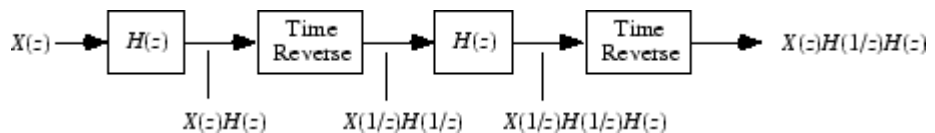
Image of Anti Causal Zero Phase Filter

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,':')
```
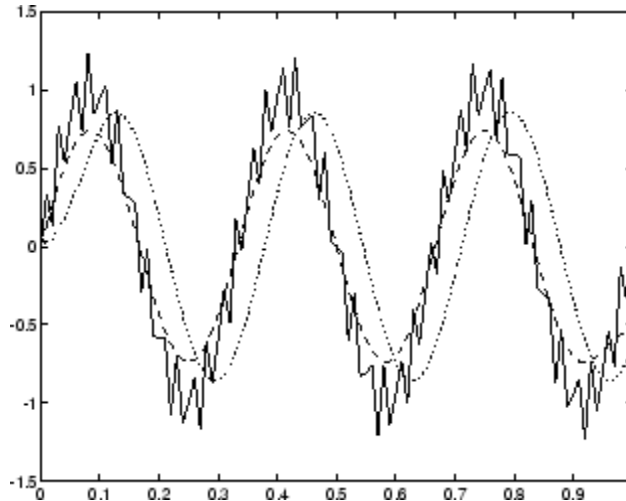

Both filtered versions eliminate the 40 Hz sinusoid evident in the original, solid line. The plot also shows how `filter` and `filtfilt` differ; the dashed (`filtfilt`) line is in phase with the original 3 Hz sinusoid, while the dotted (`filter`) line is delayed by about five samples. Also, the amplitude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

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

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

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

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

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

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

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

Then click the **Impulse Response** button 🔼 on the toolbar or select **Analysis > Impulse Response**. This plot shows the exponential decay `h(n) = 0.9n` of the single pole system:
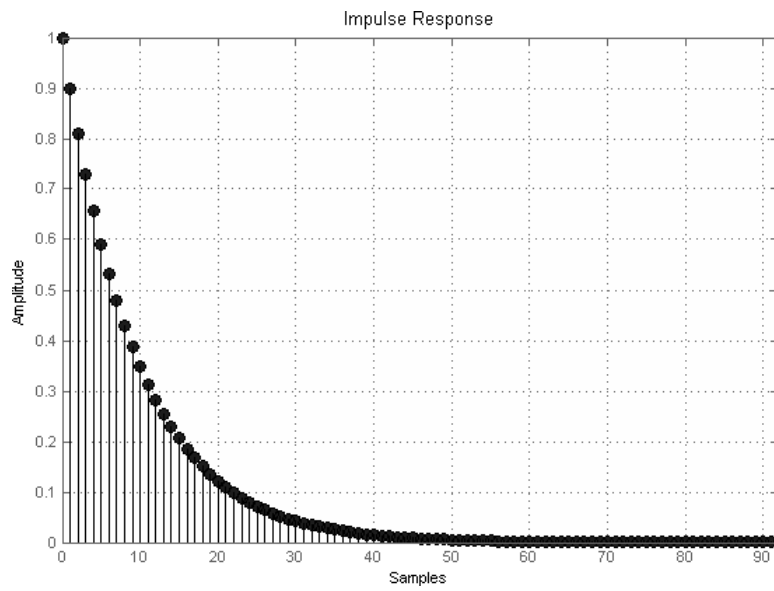
Frequency Response

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^jω), of the digital filter.

$$H(e^{jw}) = \frac{b(1) + b(2)e^{-jw} + ... + b(n+1)e^{-jwn}}{a(1) + a(2)e^{-jw} + ... + a(m+1)e^{-jwm}}$$

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

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

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

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

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

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

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

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

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

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

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

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

Magnitude (dB) and Phase Responses
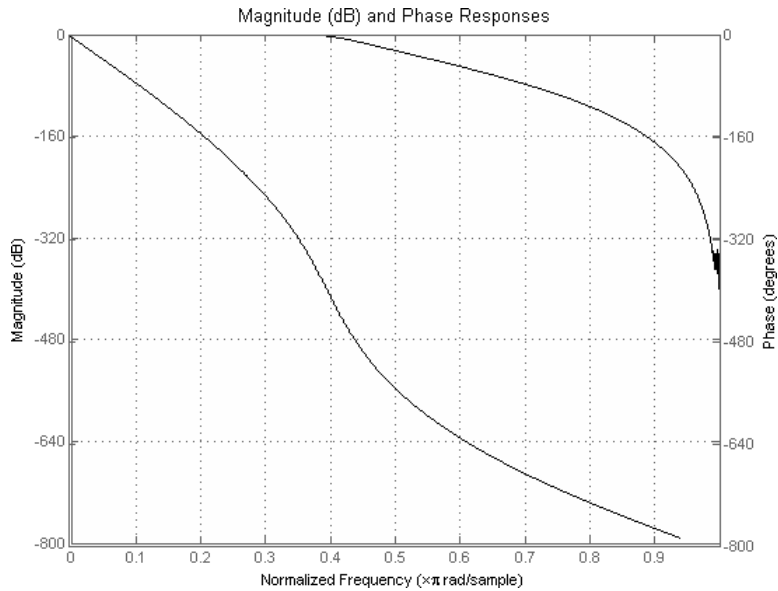
`freqz` can also accept a vector of arbitrary frequency points for use in the frequency response calculation. For example,

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

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

and click the **Magnitude and Phase Response** button ![icon] on the toolbar or select **Analysis > Magnitude and Phase Response** to display the plot.
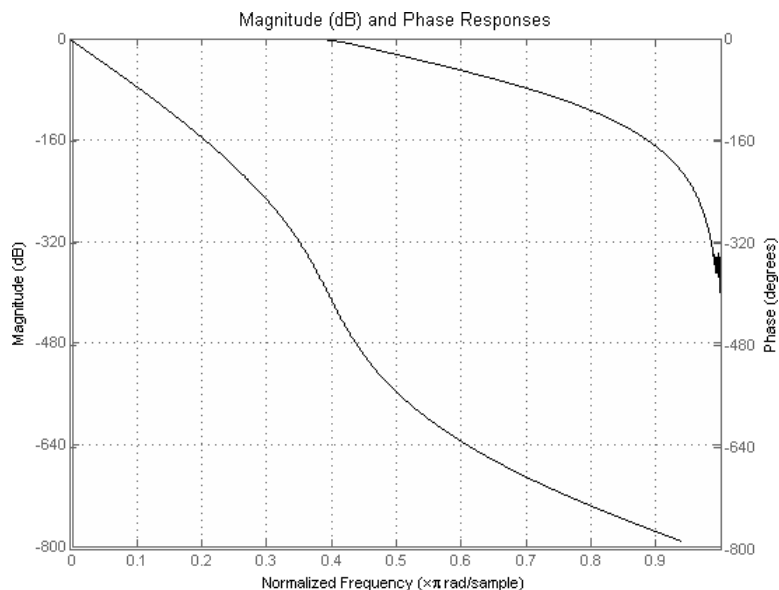
The unwrap function is also useful in frequency analysis. unwrap unwraps the phase to make it continuous across 360° phase discontinuities by adding multiples of ±360°, as needed. To see how unwrap is useful, design a 25th-order lowpass FIR filter:

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

Obtain the 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
```
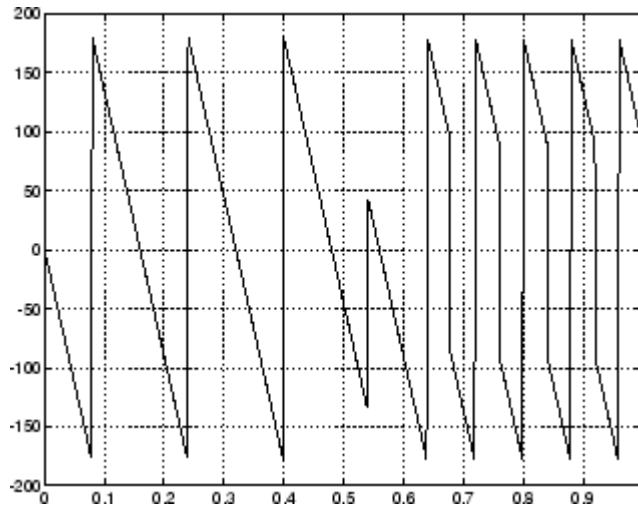

It is difficult to distinguish the 360° jumps (an artifact of the arctangent function inside angle) from the 180° jumps that signify zeros in the frequency response.

unwrap eliminates the 360° jumps:

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

or you can use phasez to see the unwrapped phase.

Delay

The *group delay* of a filter is a measure of the average time 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(ω) is the phase, or argument of H(e^{jω}). Compute group delay with

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

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

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

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

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

```
[b,a] = butter(10,200/1000);
hFVT = fvtool(b,a,'Analysis','grpdelay');
set(hFVT,'NumberofPoints',128,'OverlayedAnalysis','phasedelay');
legend(hFVT)
```
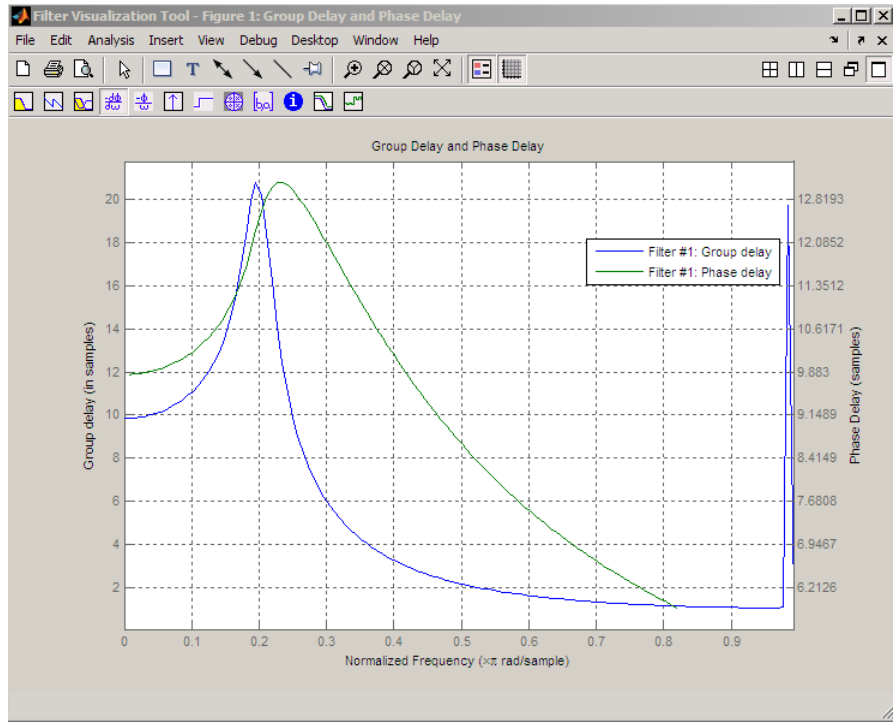
Zero-Pole Analysis

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

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

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

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

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

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

and click the **Pole/Zero Plot** toolbar button on the toolbar or select **Analysis > Pole/Zero Plot** to see the plot.
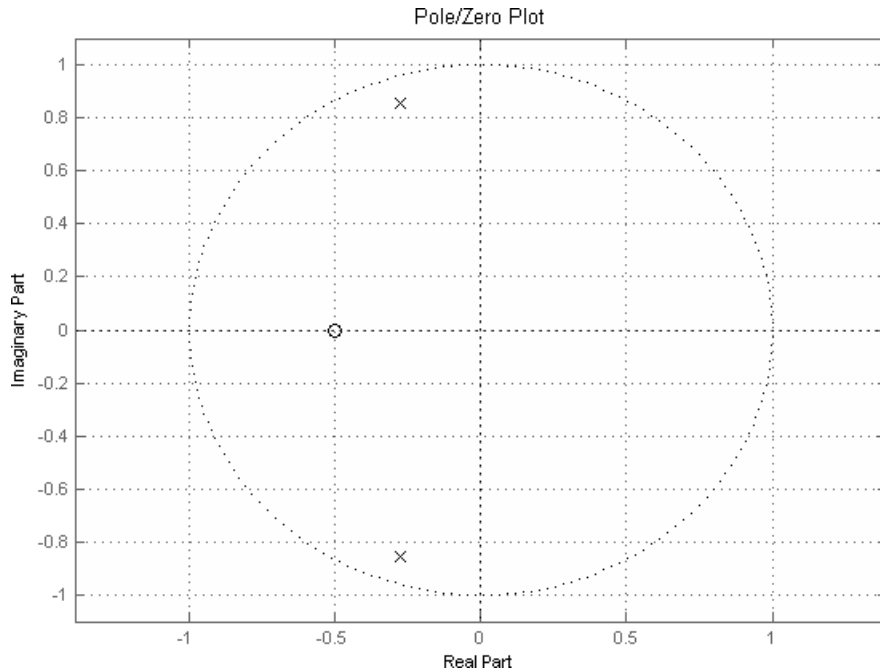
For a system in zero-pole form, supply column vector arguments z and p to zplane:

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

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

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

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

See "Linear System Models" on page 1-23 for details on zero-pole and transfer function representation of systems.

Linear System Models

In this section...
"Available Models" on page 1-23
"Discrete-Time System Models" on page 1-23
"Continuous-Time System Models" on page 1-31
"Linear System Transformations" on page 1-32

Available Models

Several Signal Processing Toolbox models are provided 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 the MATLAB technical computing environment.

Discrete-Time System Models

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

- "Transfer Function" on page 1-23
- "Zero-Pole-Gain" on page 1-24
- "State-Space" on page 1-25
- "Partial Fraction Expansion (Residue Form)" on page 1-26
- "Second-Order Sections (SOS)" on page 1-27
- "Lattice Structure" on page 1-28
- "Convolution Matrix" on page 1-30

Transfer Function

The *transfer function* is a basic *z*-domain representation of a digital filter, expressing the filter as a ratio of two polynomials. It is the principal

discrete-time model for this toolbox. The transfer function model description for the *z*-transform of a digital filter's difference equation is

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

Here, the constants $b(i)$ and $a(i)$ are the filter coefficients, and the order of the filter is the maximum of *n* and *m*. In the MATLAB environment, you store these coefficients in two vectors (row vectors by convention), one row vector for the numerator and one for the denominator. See "Filters and Transfer Functions" on page 1-3 for more details on the transfer function form.

Zero-Pole-Gain

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

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

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

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

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

The zeros and poles of this filter are

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

Returning to the original polynomials,

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

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

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

For b = [2 3 4], the roots function misses the zero for z equal to 0. In fact, 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

$$x(n + 1) = Ax(n) + Bu(n)$$
$$y(n) \quad = Cx(n) + Du(n)$$

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

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

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

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

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

Partial Fraction Expansion (Residue Form)

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

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

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

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

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

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

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

is

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

which corresponds to

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

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

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

To verify this in the MATLAB environment, type

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

Second-Order Sections (SOS)

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

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

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

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

There are many ways to represent a filter in second-order section form. Through careful pairing of the pole and zero pairs, ordering of the sections in the cascade, and multiplicative scaling of the sections, it is possible to reduce quantization noise gain and avoid overflow in some fixed-point filter implementations. The functions `zp2sos` and `ss2sos`, described in "Linear System Transformations" on page 1-32, perform pole-zero pairing, section scaling, and section ordering.

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

Lattice Structure

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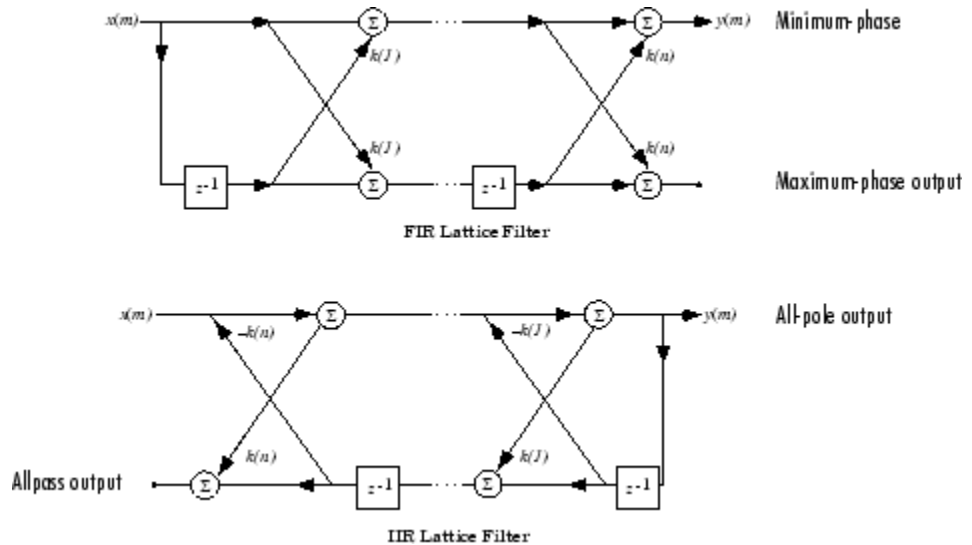
IIR Lattice Filter

FIR and IIR Lattice Filter structure diagrams

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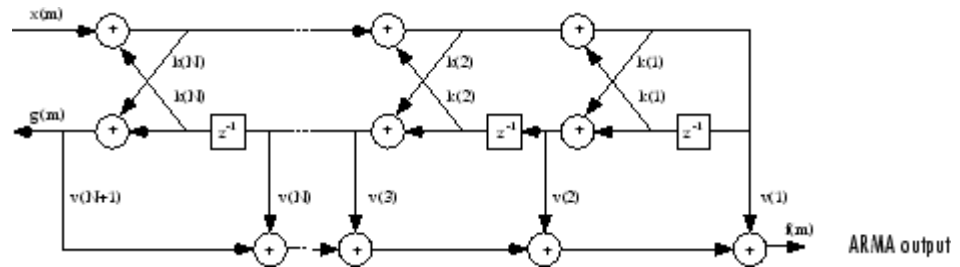
Diagram of lattice/ladder filter

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

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

This filter's lattice (reflection coefficient) representation is

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

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

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

The filter's lattice representation is

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

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

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

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

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

Convolution Matrix

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

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

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

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

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

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

Continuous-Time System Models

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

- State-space form

- Partial fraction expansion

- Transfer function

- Zero-pole-gain form

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

$$x = Ax + Bu$$
$$y = Cx + Du$$

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

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

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

where

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

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

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

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

The factored zero-pole-gain form is

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

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

Linear System Transformations

A number of Signal Processing Toolbox functions are provided to convert between the various linear system models; see "Functions — Alphabetical List" 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. Note that some cells of this table are empty.

	Transfer Function	State-Space	Zero-Pole-Gain	Partial Fraction	Lattice Filter	Second-Order Sections	Convolution Matrix
Transfer Function		tf2ss	tf2zp roots	residuez	tf2latc	none	convmtx
State-Space	ss2tf		ss2zp	none	none	ss2sos	none

	Transfer Function	**State-Space**	**Zero-Pole-Gain**	**Partial Fraction**	**Lattice Filter**	**Second-Order Sections**	**Convolution Matrix**
Zero-Pole-Gain	zp2tf poly	zp2ss		none	none	zp2sos	none
Partial Fraction	residuez	none	none		none	none	none
Lattice Filter	latc2tf	none	none	none		none	none
SOS	sos2tf	sos2ss	sos2zp	none	none		none

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

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


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

# Discrete Fourier Transform

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

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

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

and

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

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

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

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

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

For example, create a time vector and signal:

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

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]);
```
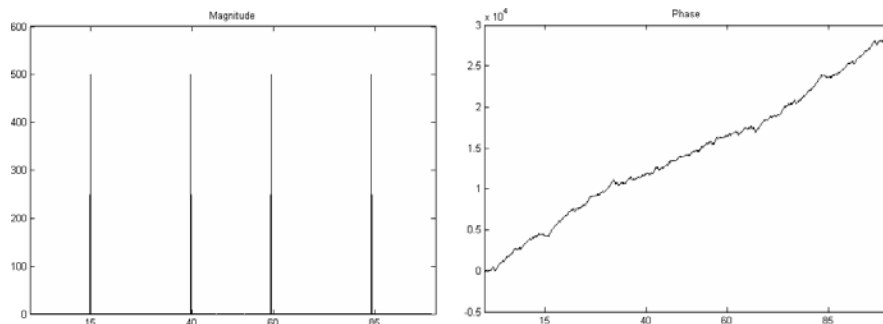

A second argument to fft specifies a number of points n for the transform, representing DFT length:

```
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 for details about the algorithm.

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

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

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

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

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

2

Filter Design and Implementation

Filter Requirements and Specification

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

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

To design a filter, the Signal Processing Toolbox software offers two approaches: object-oriented and non-object oriented. The object-oriented approach first constructs a filter specification object, `fdesign`, and then invokes an appropriate `design` method. To illustrate the object-oriented approach, design and implement a 5–th order lowpass Butterworth filter with a 3–dB frequency of 200 Hz. Assume a sampling frequency of 1 kHz. Apply the filter to input data.

```
Fs=1000; %Sampling Frequency
time = 0:(1/Fs):1; %time vector
% Data vector
x = cos(2*pi*60*time)+sin(2*pi*120*time)+randn(size(time));
d=fdesign.lowpass('N,F3dB',5,200,Fs); %lowpass filter specification object
% Invoke Butterworth design method
Hd=design(d,'butter');
y=filter(Hd,x);
```

The non-object oriented approach implements the filter using a function such as `butter` and `firpm`. All of the non-object oriented filter design functions operate with normalized frequencies. Convert frequency specifications in Hz to normalized frequency to use these functions. The Signal Processing Toolbox software defines normalized frequency to be in the closed interval

[0,1] with 1 denoting π radians/sample. For example, to specify a normalized frequency of π/2 radians/sample, enter 0.5.

To convert from Hz to normalized frequency, multiply the frequency in Hz by two and divide by the sampling frequency. To design a 5–th order lowpass Butterworth filter with a 3–dB frequency of 200 Hz using the non-object oriented approach, use `butter`:

```
Wn = (2*200)/1000; %Convert 3-dB frequency
% to normalized frequency: 0.4*pi rad/sample
[B,A] = butter(5,Wn,'low');
y = filter(B,A,x);
```

IIR Filter Design

IIR vs. FIR Filters

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

Classical IIR Filters

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

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

Other IIR Filters

The direct filter design function `yulewalk` finds a filter with magnitude response approximating a 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 7-13.

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

IIR Filter Method Summary

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

Toolbox Filters Methods and Available Functions

Filter Method	Description	Filter Functions

Analog Prototyping	Using the poles and zeros of a classical lowpass prototype filter in the continuous (Laplace) domain, obtain a digital filter through frequency transformation and filter discretization.	Complete design functions:
		`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`

Toolbox Filters Methods and Available Functions (Continued)

Filter Method	Description	Filter Functions
Generalized Butterworth Design	Design lowpass Butterworth filters with more zeros than poles.	`maxflat`

Parametric Modeling	Find a digital filter that approximates a prescribed time or frequency domain response. (See System Identification Toolbox™ documentation for an extensive collection of parametric modeling tools.)	Time-domain modeling functions:
		
		`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 2-43 for detailed steps on the filter design process.

Complete Classical IIR Filter Design

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

Filter Design Functions

Filter Type	Design Function
Bessel (analog only)	`[b,a] = besself(n,Wn,`*`options`*`)`
	`[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`*`)`

Filter Design Functions (Continued)

Filter Type	Design Function
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. In general, you should avoid

using the transfer function form because numerical problems caused by roundoff errors can occur. Instead, use the zero-pole-gain form which you can convert to a second-order section (SOS) form using zp2sos and then use the SOS form with dfilt to analyze or implement your filter.

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

Designing IIR Filters to Frequency Domain Specifications

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

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

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

```
[n,Wn] = buttord([1000 2000]/5000,[500 2500]/5000,1,60)
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 "Functions — Alphabetical List" 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 $\Omega = \infty$. $\left| H(j\Omega) \right| = \sqrt{1/2}$ at $\Omega = 1$.
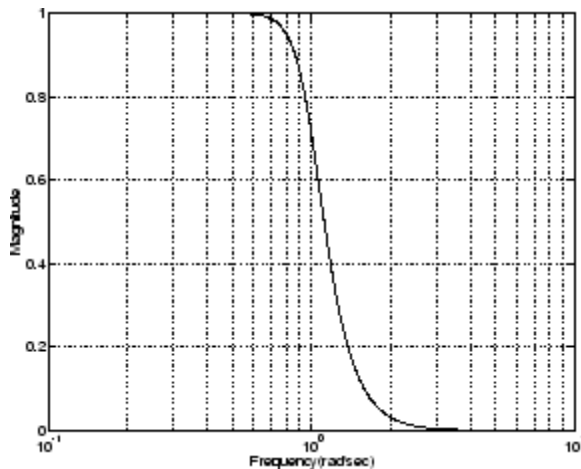
Chebyshev Type I Filter

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 $\Omega = 1$.
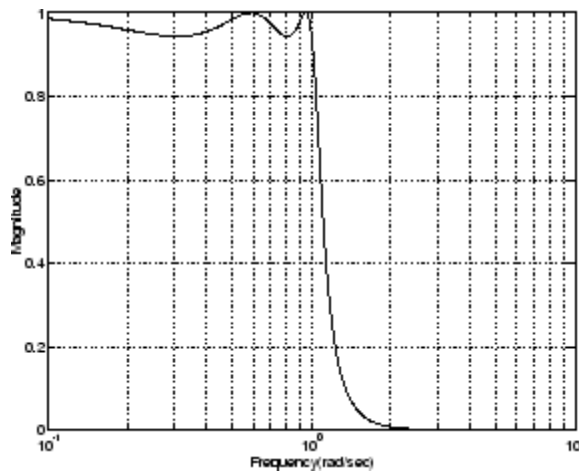
Chebyshev Type II Filter

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 $\Omega = 1$.
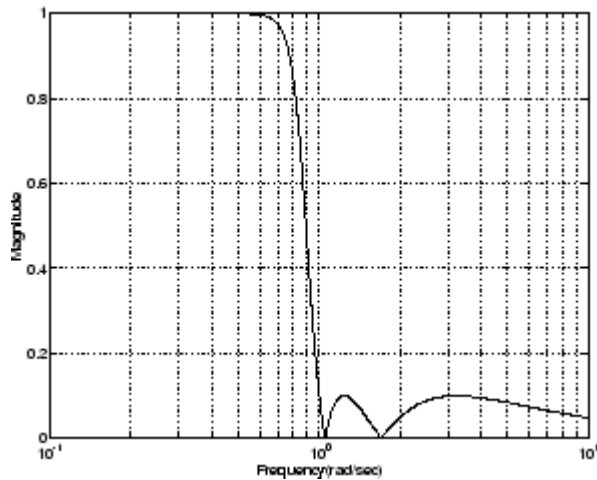
Elliptic Filter

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 $\Omega = 1$.
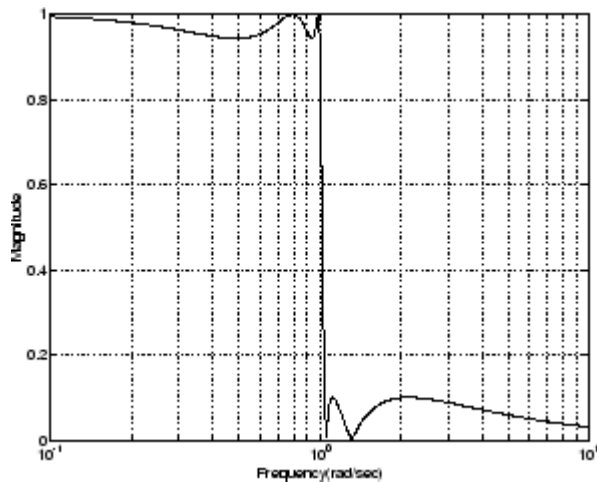
Bessel Filter

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.
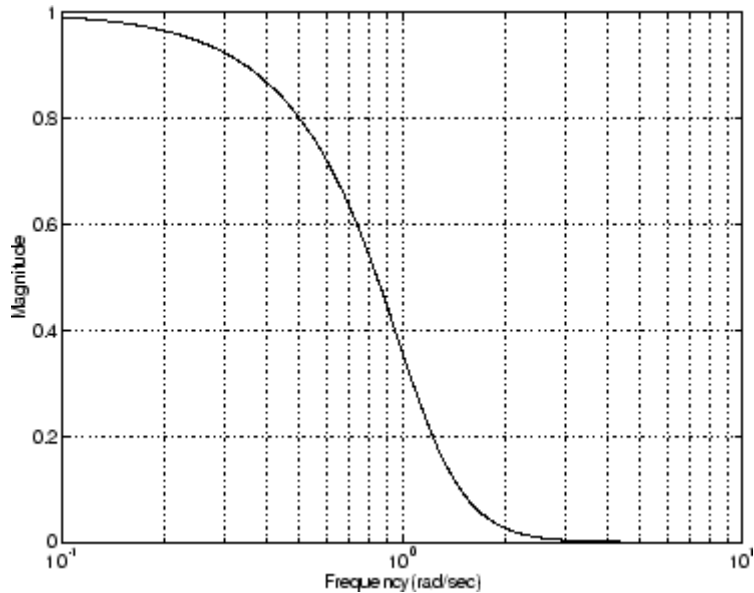
Note The lowpass filters shown above were created with the analog prototype functions `besselap`, `buttap`, `cheb1ap`, `cheb2ap`, and `ellipap`. These functions find the zeros, poles, and gain of an order n analog filter of the appropriate type with cutoff frequency of 1 rad/s. The complete filter design 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 2-43 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 7-13 discusses other methods that may also be considered direct, such as Prony's method, Linear Prediction, the Steiglitz-McBride method, and inverse frequency design.

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

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

returns row vectors `b` and `a` containing the `n+1` numerator and denominator coefficients of the order `n` IIR filter whose frequency-magnitude characteristics approximate those given in vectors `f` and `m`. `f` is a vector of frequency points ranging from 0 to 1, where 1 represents the Nyquist frequency. `m` is a vector containing the desired magnitude response at the points in `f`. `f` and `m` can describe any piecewise linear shape magnitude response, including a multiband response. The FIR counterpart of this function is `fir2`, which also designs a filter based on an arbitrary piecewise linear magnitude response. See "FIR Filter Design" on page 2-17 for details.

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

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

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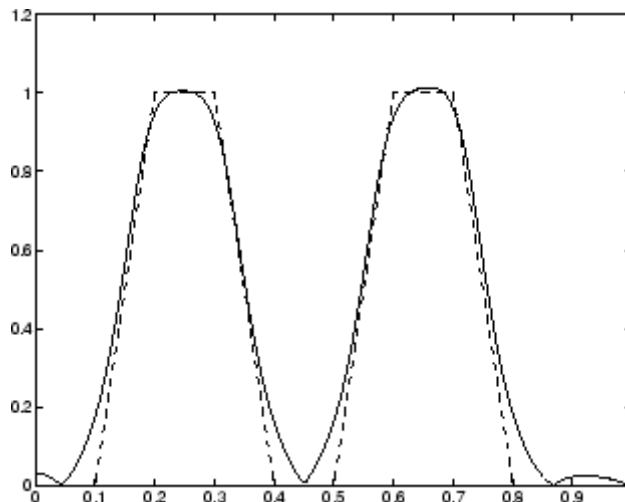
Generalized Butterworth Filter Design

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

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

```
[b,a] = maxflat(3,3,0.25)
b =
    0.0317    0.0951    0.0951    0.0317
a =
    1.0000   -1.4590    0.9104   -0.1978
[b,a] = butter(3,0.25)
b =
    0.0317    0.0951    0.0951    0.0317
a =
    1.0000   -1.4590    0.9104   -0.1978
```

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

```
[b,a] = maxflat(3,1,0.25)
b =
    0.0950    0.2849    0.2849    0.0950
a =
    1.0000   -0.2402
```

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

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

```
maxflat(4,'sym',0.3)
ans =
    0.0331    0.2500    0.4337    0.2500    0.0331
```

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

FIR Filter Design

FIR vs. IIR Filters

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

FIR filters have the following primary advantages:

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

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

FIR Filter Summary

FIR Filters

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

Linear Phase Filters

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

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

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

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

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

Windowing Method

Consider the ideal, or "brick wall," digital lowpass filter with a cutoff frequency of 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

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

This filter is not implementable since its impulse response is infinite and noncausal. To create a finite-duration impulse response, truncate it by applying a window. By retaining the central section of impulse response in this truncation, you obtain a linear phase FIR filter. For example, a length 51 filter with a lowpass cutoff frequency ω_0 of 0.4 π rad/s is

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

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

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

Note that the *y*-axis shown in the figure below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.
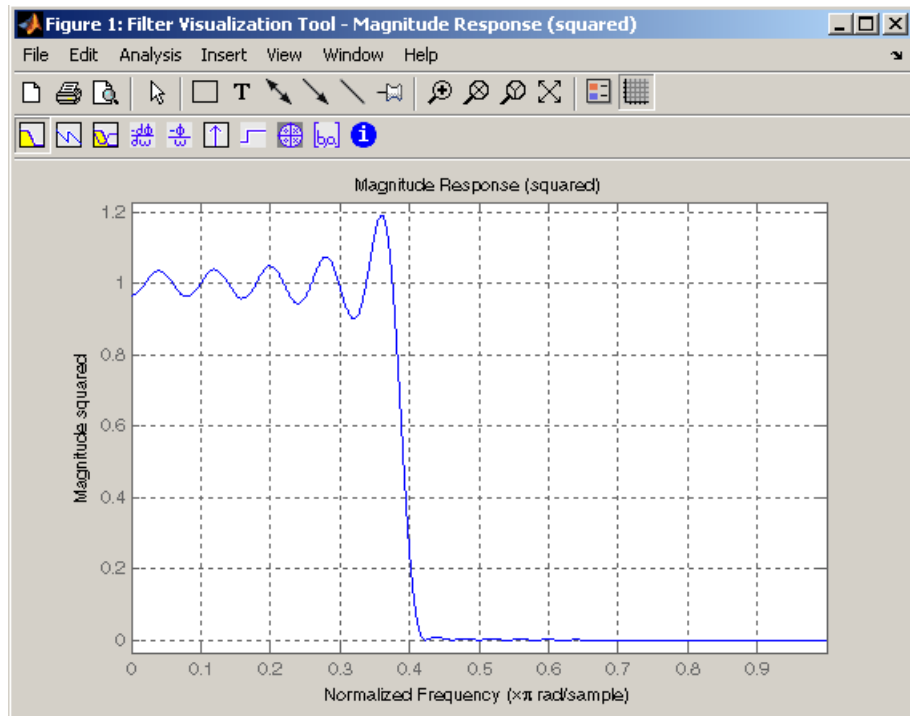
Ringing and ripples occur in the response, especially near the band edge. This "Gibbs effect" does not vanish as the filter length increases, but a nonrectangular window reduces its magnitude. Multiplication by a window in the time domain causes a convolution or smoothing in the frequency domain. Apply a length 51 Hamming window to the filter and display the result using FVTool:

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

Note that the *y*-axis shown in the figure below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.
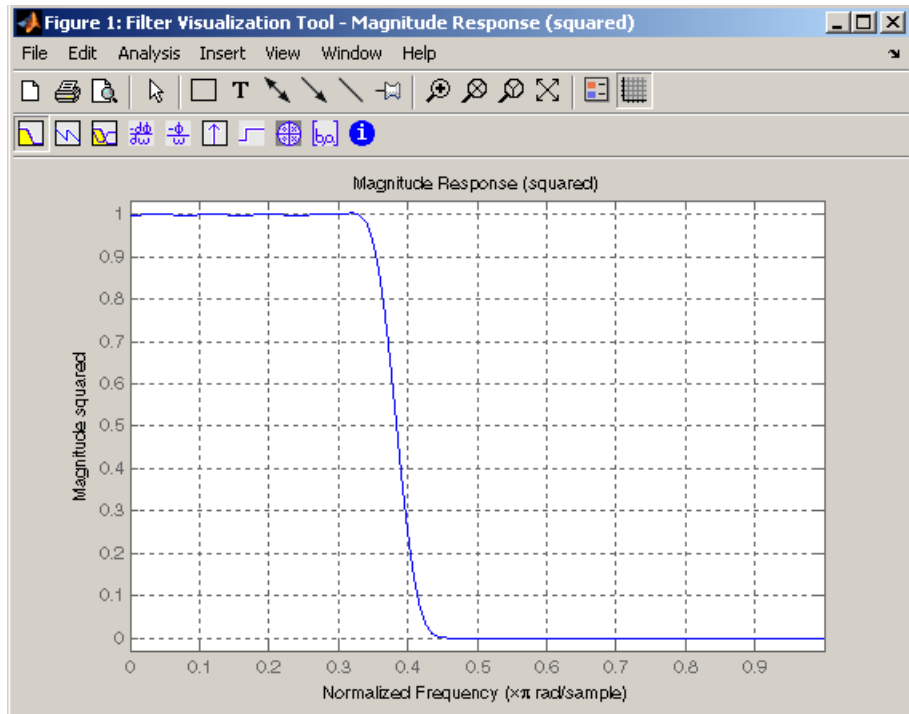
Using a Hamming window greatly reduces the ringing. This improvement is at the expense of transition width (the windowed version takes longer to ramp from passband to stopband) and optimality (the windowed version does not minimize the integrated squared error).

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 "Windows" on page 7-2 for an overview of windows and their properties.

Standard Band FIR Filter Design: fir1

fir1 implements the classical method of windowed linear phase FIR digital filter design. It resembles the IIR filter design functions in that it is formulated to design filters in standard band configurations: lowpass, bandpass, highpass, and bandstop.

The statements

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

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

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

Kaiser Window Order Estimation. The kaiserord function estimates the filter order, cutoff frequency, and Kaiser window beta parameter needed to meet a given set of specifications. Given a vector of frequency band edges and a corresponding vector of magnitudes, as well as maximum allowable ripple, kaiserord returns appropriate input parameters for the fir1 function.

Multiband FIR Filter Design: fir2

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

The commands

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

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

Multiband FIR Filter Design with Transition Bands

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

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

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

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

Basic Configurations

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

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

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

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

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

and compare their frequency responses using FVTool:

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

Note that the *y*-axis shown in the figure below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.
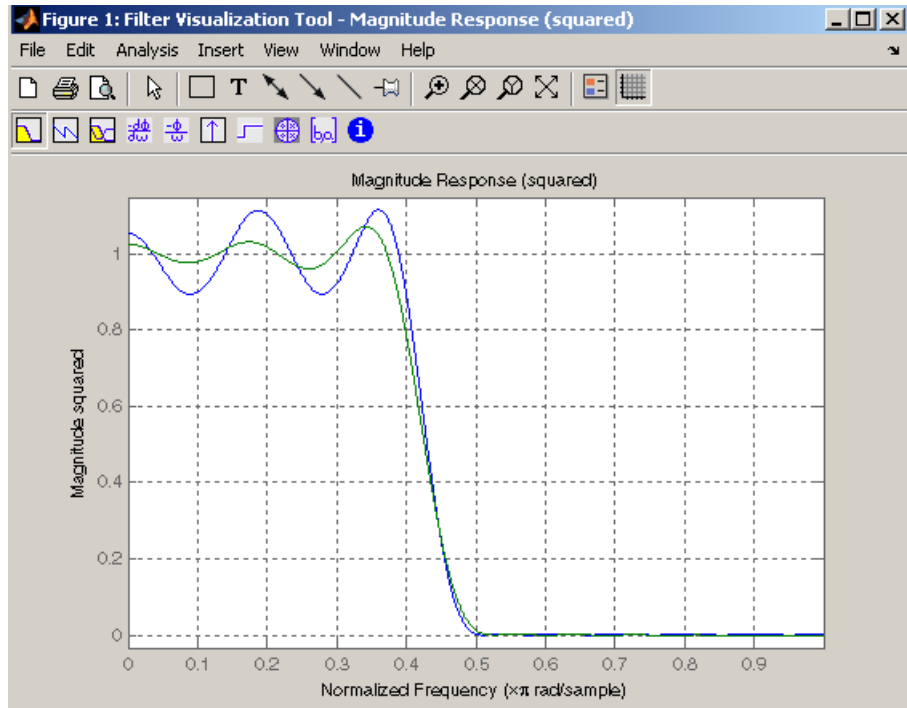
The filter designed with `firpm` exhibits equiripple behavior. Also note that the `firls` filter has a better response over most of the passband and stopband, but at the band edges (`f = 0.4` and `f = 0.5`), the response is further away from the ideal than the `firpm` filter. This shows that the `firpm` filter's *maximum* error over the passband and stopband is smaller and, in fact, it is the smallest possible for this band edge configuration and filter length.

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

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

```
a = [0  0   1   1   0   0];   % Bandpass filter amplitude
```

Technically, these f and a vectors define five bands:

- Two stopbands, from 0.0 to 0.3 and from 0.8 to 1.0

- A passband from 0.4 to 0.7

- Two transition bands, from 0.3 to 0.4 and from 0.7 to 0.8

Example highpass and bandstop filters are

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

An example multiband bandpass filter is

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

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

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

The Weight Vector

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

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

```
w = [1 10];              % Weight vector
b = firpm(n,f,a,w);
```

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

Anti-Symmetric Filters / Hilbert Transformers

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

```
b = firpm(21,[0.05 1],[1 1],'h');        % Highpass Hilbert
bb = firpm(20,[0.05 0.95],[1 1],'h');    % Bandpass Hilbert
fvtool(b,1,bb,1)
```
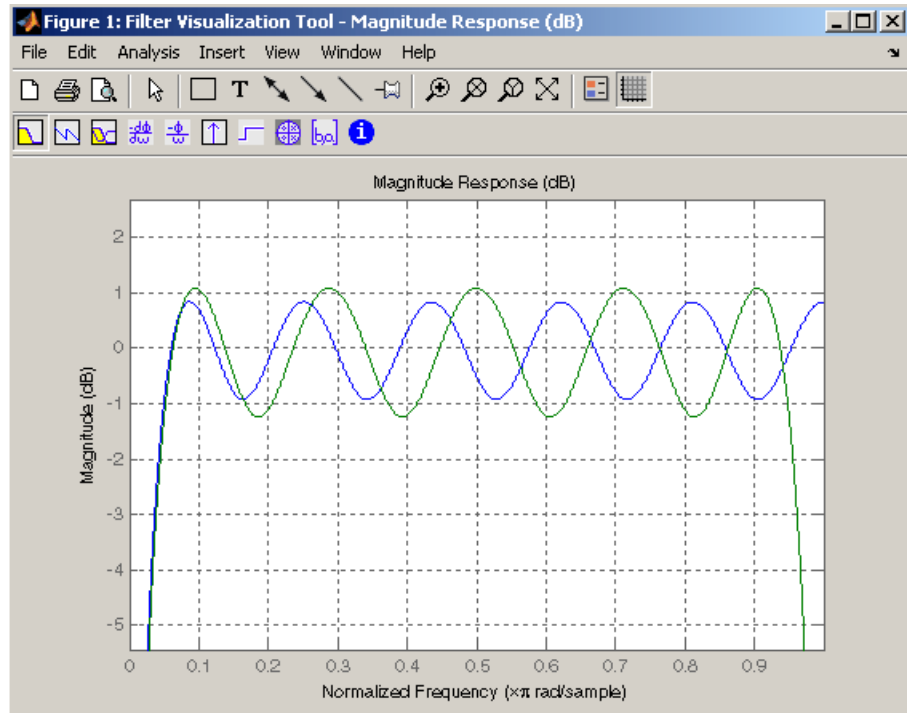
You can find the delayed Hilbert transform of a signal x by passing it through these filters.

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

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

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

This method does not work directly for filters of odd order, which require a noninteger delay. In this case, the `hilbert` function, described in "Specialized Transforms" on page 7-38, estimates the analytic signal. Alternatively, use the `resample` function to delay the signal by a noninteger number of samples.

Differentiators

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

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

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

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

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

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

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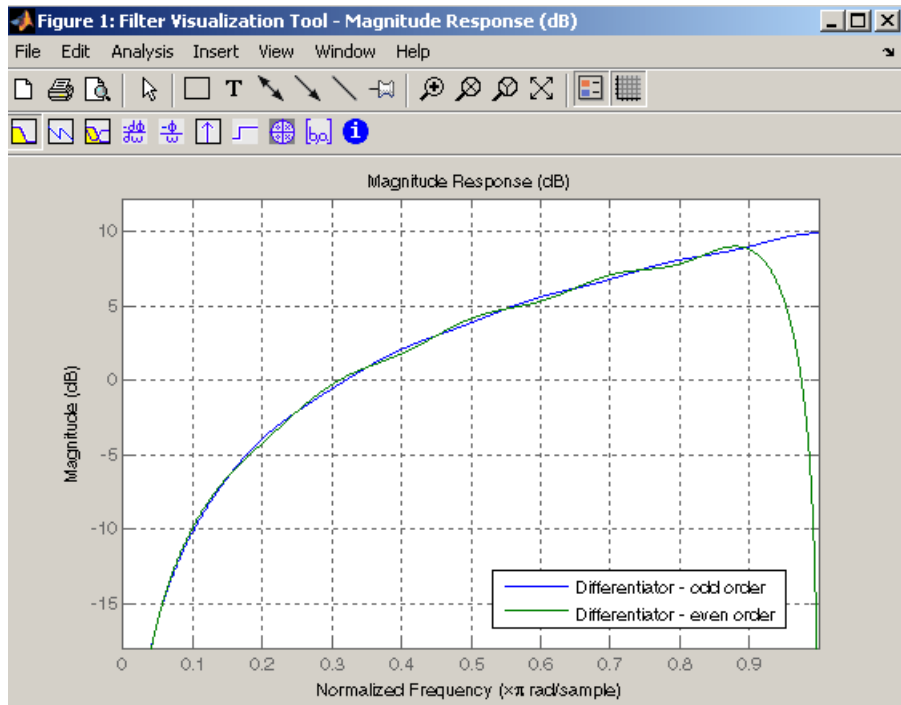

Constrained Least Squares FIR Filter Design

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

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

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

There are two toolbox functions that implement this design technique.

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

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

Basic Lowpass and Highpass CLS Filter Design

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

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

- Maximum stopband deviation from 0 (stopband ripple) of 0.008.
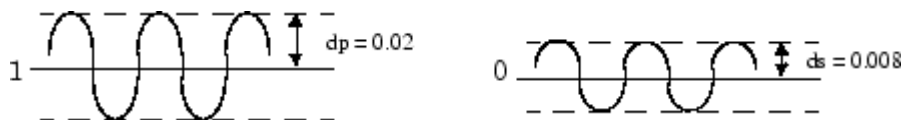
To approach this design problem using `fircls1`, use the following commands:

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

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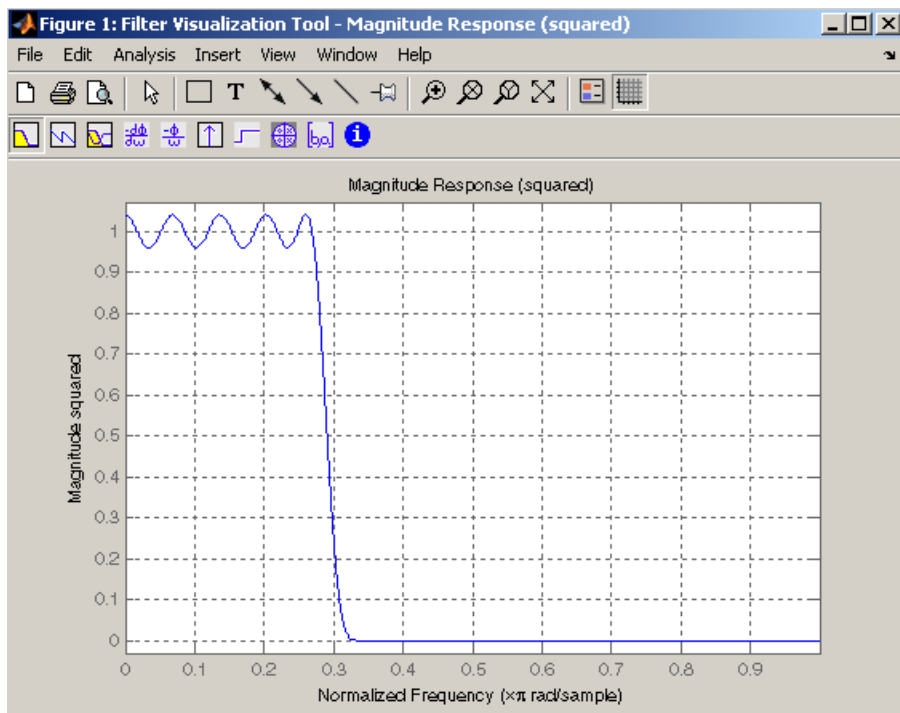
Multiband CLS Filter Design

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

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

• From 0 to 0.3 (normalized): amplitude 0, upper bound 0.005, lower bound –0.005

• From 0.3 to 0.5: amplitude 0.5, upper bound 0.51, lower bound 0.49

• From 0.5 to 0.7: amplitude 0, upper bound 0.03, lower bound –0.03

• From 0.7 to 0.9: amplitude 1, upper bound 1.02, lower bound 0.98

• From 0.9 to 1: amplitude 0, upper bound 0.05, lower bound –0.05

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

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

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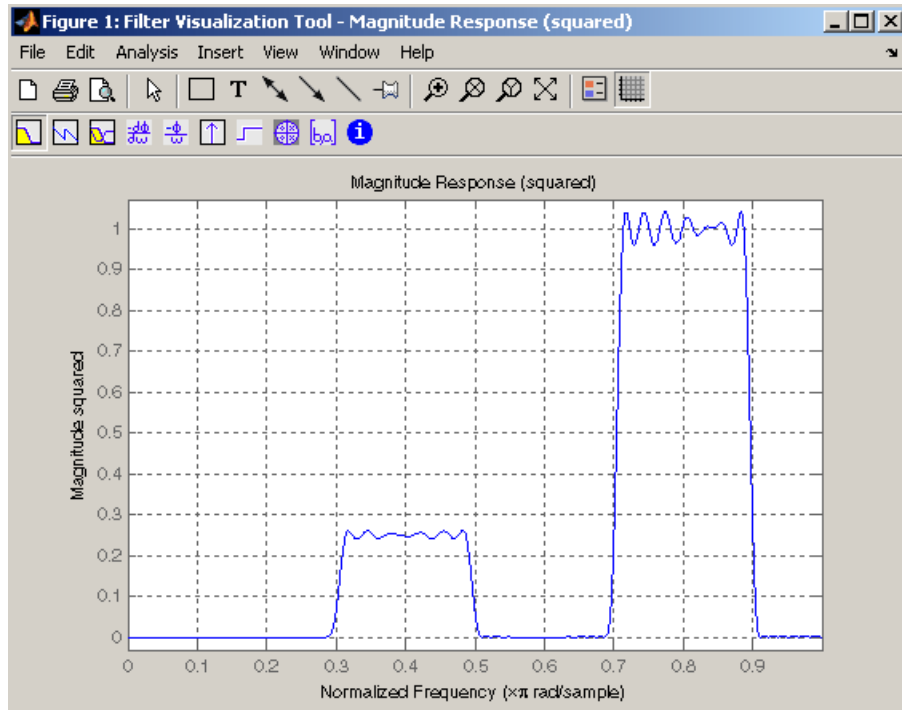
Weighted CLS Filter Design

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

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

- Passband edge for the weight function of 0.28 (normalized)

- Stopband edge for the weight function of 0.32

- Weight error minimization 10 times as much in the stopband as in the passband

To approach this using `fircls1`, type

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

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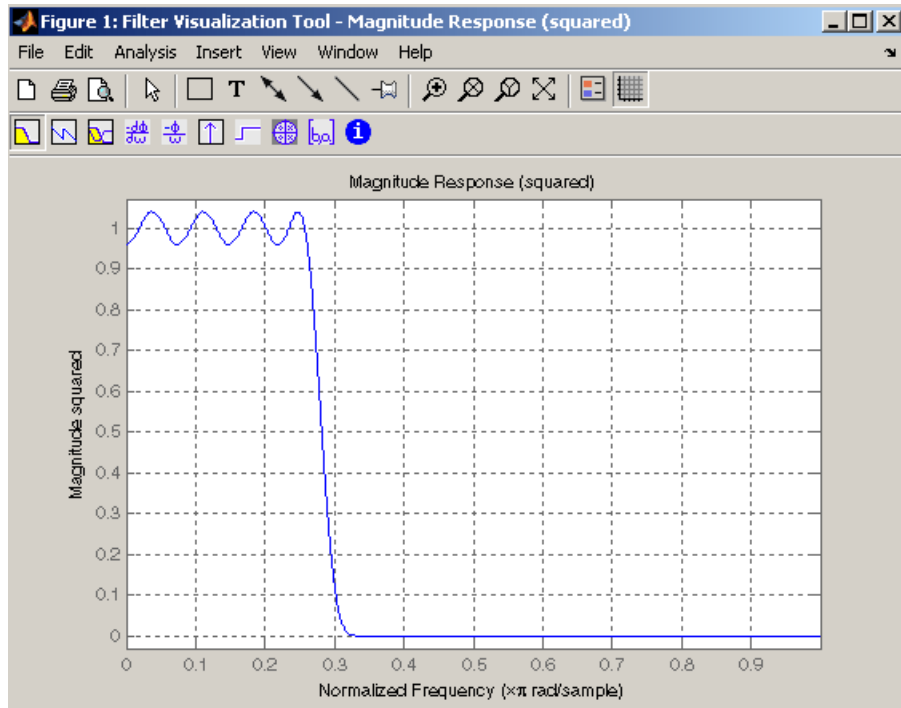
Arbitrary-Response Filter Design

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

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

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

Multiband Filter Design

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

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

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

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

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

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

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

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

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

Note that the range of data shown below is (-Fs/2,Fs/2). You can set this range by changing the *x*-axis units to **Frequency (Fs = 1 Hz)**.
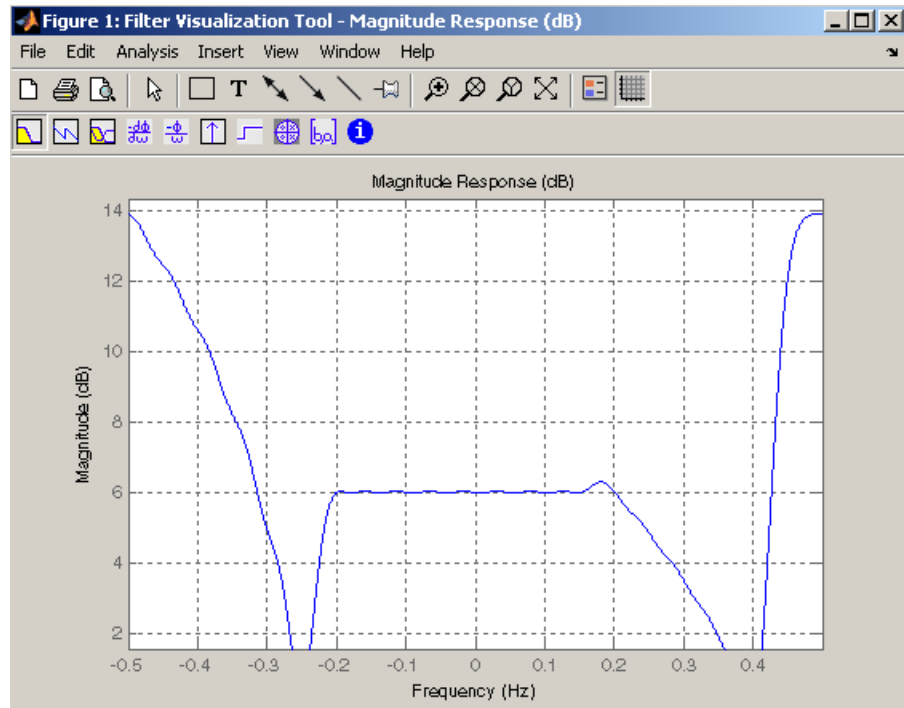
The filter response for this multiband filter is complex, which is expected because of the asymmetry in the frequency domain. The impulse response, which you can select from the FVTool toolbar, is shown below.
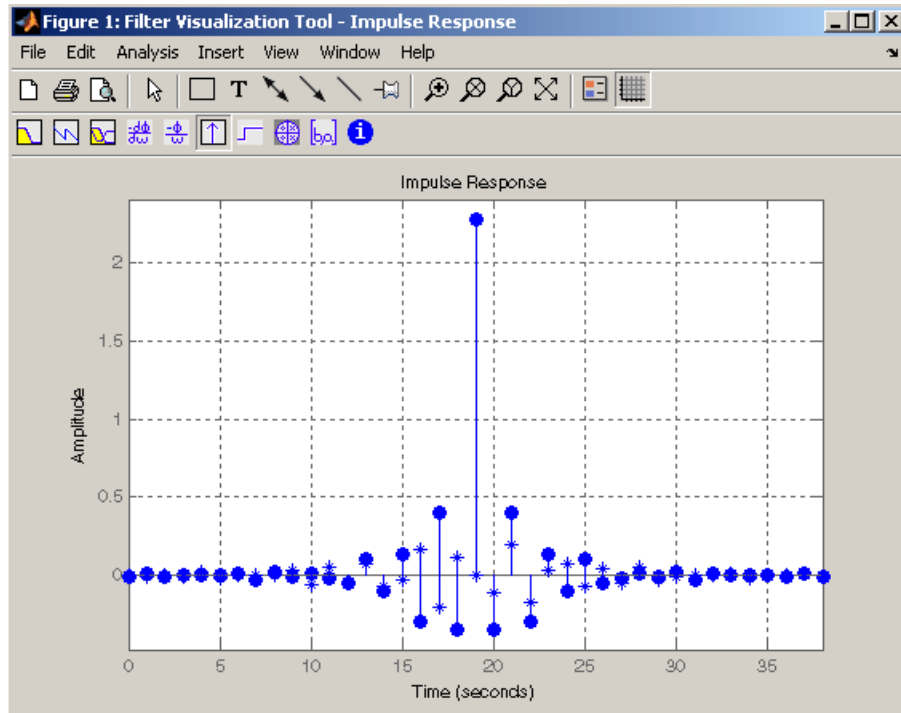
Filter Design with Reduced Delay

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

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

The resulting magnitude response is

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

Note that the range of data in this plot is (-Fs/2,Fs/2), which you can set changing the *x*-axis units to **Frequency**. The *y*-axis is in Magnitude Squared, which you can set by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.
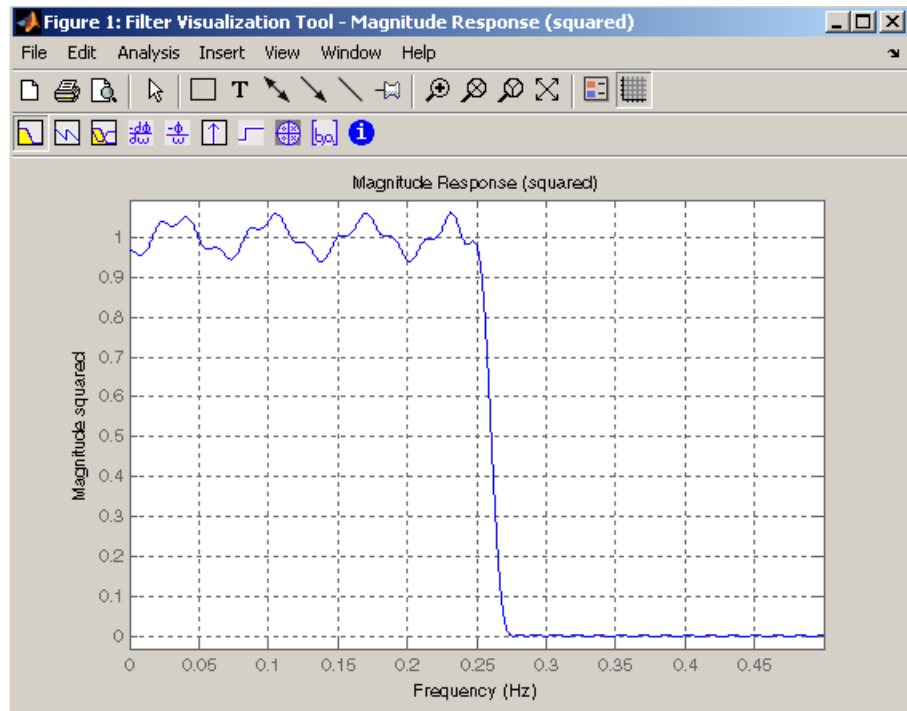
The group delay of the filter reveals that the offset has been reduced from N/2 to N/2-16 (i.e., from 30.5 to 14.5). Now, however, the group delay is no longer flat in the passband region. To create this plot, click the **Group Delay** button on the toolbar.
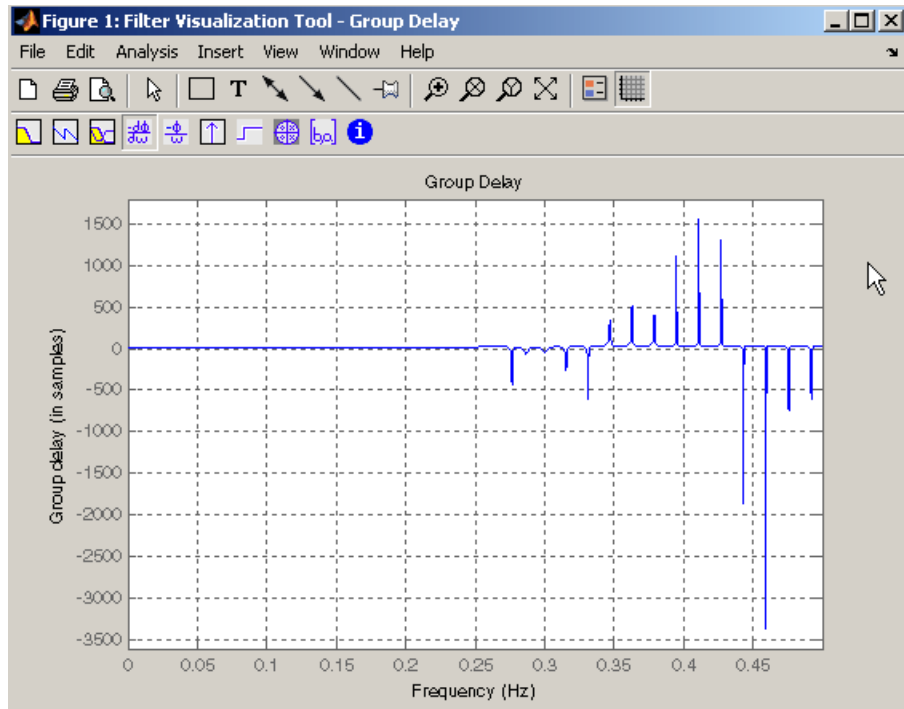
If we compare this nonlinear-phase filter to a linear-phase filter that has exactly 14.5 samples of group delay, the resulting filter is of order 2*14.5, or 29. Using b = cfirpm(29,[0 0.5 0.55 1],'lowpass'), the passband and stopband ripple is much greater for the order 29 filter. These comparisons can assist you in deciding which filter is more appropriate for a specific application.

Special Topics in IIR Filter Design

In this section...

Classic IIR Filter Design

The classic IIR filter design technique includes the following steps.

1 Find an analog lowpass filter with cutoff frequency of 1 and translate this prototype filter to the desired band configuration

2 Transform the filter to the digital domain.

3 Discretize the filter.

The toolbox provides functions for each of these steps.

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

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

Analog Prototype Design

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

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

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

Frequency Transformation

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

Frequency 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
 	
$s' = \dfrac{\omega_0}{s}$
	`[numt,dent]   = lp2hp (num,den,Wo)`
	
	`[At,Bt,Ct,Dt] = lp2hp (A,B,C,D,Wo)`

Frequency Transformation	Transformation Function
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)');
```
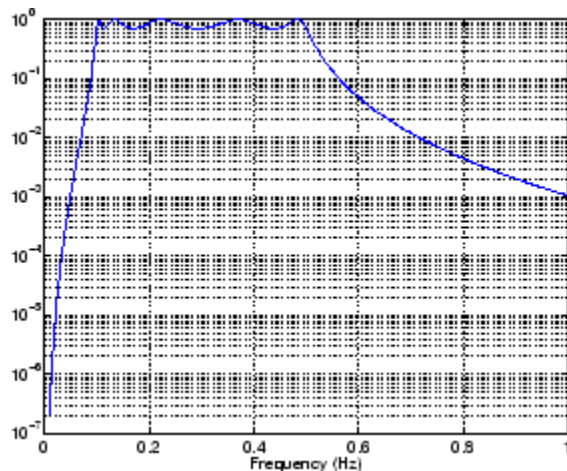

Filter Discretization

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 and phase response using FVTool:

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

Click the **Magnitude and Phase Response** toolbar button.
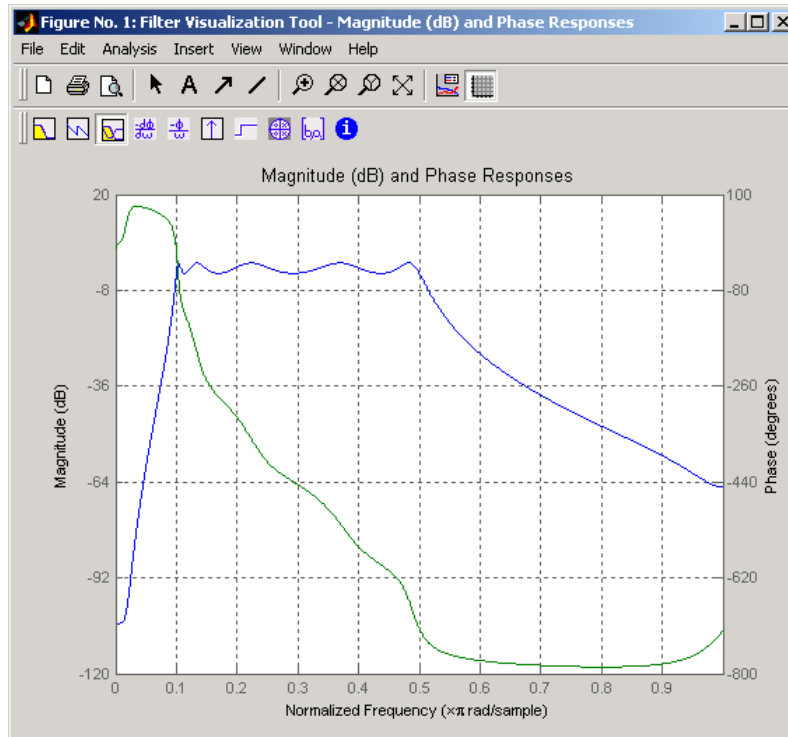
Impulse invariance retains the cutoff frequencies of 0.1 Hz and 0.5 Hz.

Bilinear Transformation

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

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

Bilinear transformation maps the *j*Ω-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 $2\pi f_p \big/ \tan\left(\pi f_p/f_s\right)$ if you give `bilinear` a trailing argument that represents a "match" frequency `Fp`. If a match frequency `Fp` (in hertz) is present, `bilinear` maps the frequency $\Omega = 2\pi f_p$ (in rad/s) to the same frequency in the discrete domain, normalized to the sampling rate: $\omega = 2\pi f_p/f_s$ (in rad/sample).

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

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

The frequency response of the resulting digital filter is

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

Click the **Magnitude and Phase Response** toolbar button.
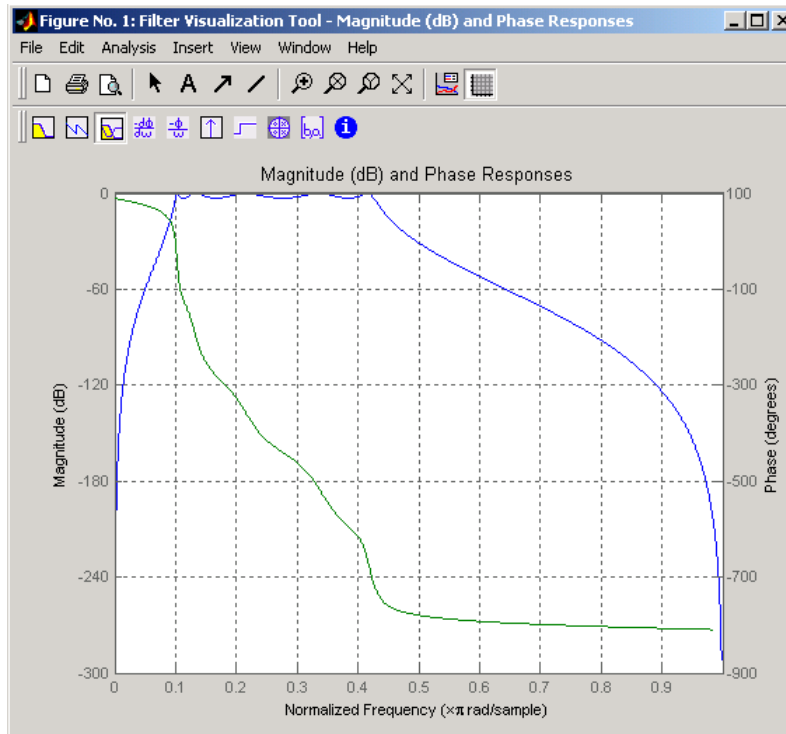
The lower band edge is at 0.1 Hz as expected. Notice, however, that the upper band edge is slightly less than 0.5 Hz, although in the analog domain it was exactly 0.5 Hz. This illustrates the nonlinear nature of the bilinear transformation. To counteract this nonlinearity, it is necessary to create analog domain filters with "prewarped" band edges, which map to the correct locations upon bilinear transformation. Here the prewarped frequencies u1 and u2 generate Bw and Wo for the lp2bp function:

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

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

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

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

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

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

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

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

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.

Designing a Filter in Fdesign — Process Overview

Process Flow Diagram and Filter Design Methodology

In this section...

Note You must minimally have the Signal Processing Toolbox installed to use `fdesign` and `design`. Some of the features described below may be unavailable if your installation does not additionally include the DSP System Toolbox™ license. The DSP System Toolbox significantly expands the functionality available for the specification, design, and analysis of filters. You can verify the presence of both toolboxes by typing `ver` at the command prompt.

Exploring the Process Flow Diagram

The process flow diagram shown in the following figure lists the steps and shows the order of the filter design process.
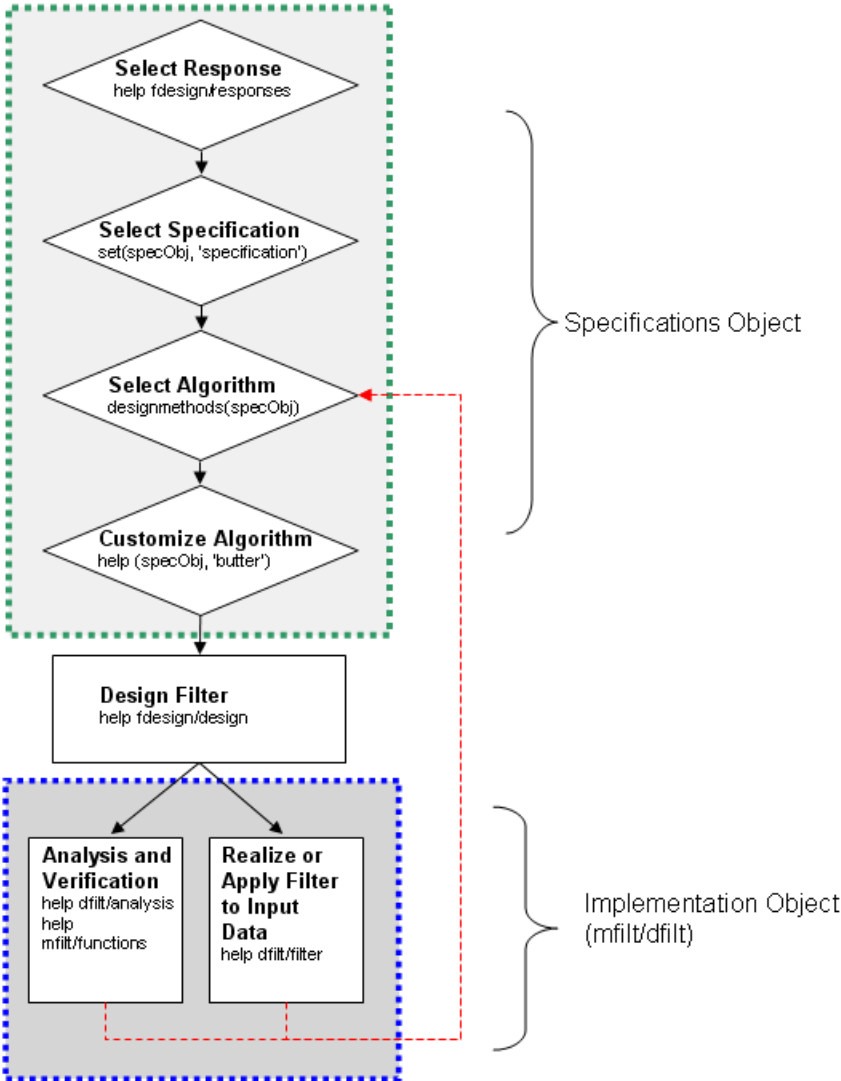
The first four steps of the filter design process relate to the filter Specifications Object, while the last two steps involve the filter Implementation Object. Both of these objects are discussed in more detail in the following sections. Step 5 - the design of the filter, is the transition step from the filter Specifications Object to the Implementation object. The analysis and verification step is

completely optional. It provides methods for the filter designer to ensure that the filter complies with all design criteria. Depending on the results of this verification, you can loop back to steps 3 and 4, to either choose a different algorithm, or to customize the current one. You may also wish to go back to steps 3 or 4 after you filter the input data with the designed filter (step 7), and find that you wish to tweak the filter or change it further.

The diagram shows the help command for each step. Enter the help line at the MATLAB command prompt to receive instructions and further documentation links for the particular step. Not all of the steps have to be executed explicitly. For example, you could go from step 1 directly to step 5, and the interim three steps are done for you by the software.

The following are the details for each of the steps shown above.

Selecting a Response

If you type:

```
help fdesign/responses
```

at the MATLAB command prompt, you see a list of all available filter responses. The responses marked with an asterisk require the DSP System Toolbox.

You must select a response to initiate the filter. In this example, a bandpass filter Specifications Object is created by typing the following:

```
d = fdesign.bandpass
```

Selecting a Specification

A *specification* is an array of design parameters for a given filter. The specification is a property of the Specifications Object.

Note A specification is not the same as the Specifications Object. A Specifications Object contains a specification as one of its properties.

When you select a filter response, there are a number of different specifications available. Each one contains a different combination of design parameters. After you create a filter Specifications Object, you can query the available specifications for that response. Specifications marked with an asterisk require the DSP System Toolbox.

```
>> d = fdesign.bandpass; % step 1 - choose the response
>> set (d, 'specification')

ans =

    'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
    'N,F3dB1,F3dB2'
    'N,F3dB1,F3dB2,Ap'
    'N,F3dB1,F3dB2,Ast'
    'N,F3dB1,F3dB2,Ast1,Ap,Ast2'
    'N,F3dB1,F3dB2,BWp'
    'N,F3dB1,F3dB2,BWst'
    'N,Fc1,Fc2'
    'N,Fp1,Fp2,Ap'
    'N,Fp1,Fp2,Ast1,Ap,Ast2'
    'N,Fst1,Fp1,Fp2,Fst2'
    'N,Fst1,Fp1,Fp2,Fst2,Ap'
    'N,Fst1,Fst2,Ast'
    'Nb,Na,Fst1,Fp1,Fp2,Fst2'

>> d=fdesign.arbmag;
>> set(d,'specification')

ans =

    'N,F,A'
    'N,B,F,A'
```

The set command can be used to select one of the available specifications as follows:

```
>> d = fdesign.lowpass; % step 1
>> % step 2: get a list of available specifications
>> set (d, 'specification')
```

```
ans =

    'Fp,Fst,Ap,Ast'
    'N,F3dB'
    'N,F3dB,Ap'
    'N,F3dB,Ap,Ast'
    'N,F3dB,Ast'
    'N,F3dB,Fst'
    'N,Fc'
    'N,Fc,Ap,Ast'
    'N,Fp,Ap'
    'N,Fp,Ap,Ast'
    'N,Fp,F3dB'
    'N,Fp,Fst'
    'N,Fp,Fst,Ap'
    'N,Fp,Fst,Ast'
    'N,Fst,Ap,Ast'
    'N,Fst,Ast'
    'Nb,Na,Fp,Fst'

>> %step 2: set the required specification
>> set (d, 'specification', 'N,Fc')
```

If you do not perform this step explicitly, fdesign returns the default
specification for the response you chose in "Select a Response" on page 4-3, and
provides default values for all design parameters included in the specification.

Selecting an Algorithm

The availability of algorithms depends the chosen filter response, the design
parameters, and the availability of the DSP System Toolbox. In other words,
for the same lowpass filter, changing the specification string also changes the
available algorithms. In the following example, for a lowpass filter and a
specification of 'N, Fc', only one algorithm is available—window.

```
>> %step 2: set the required specification
>> set (d, 'specification', 'N,Fc')
>> designmethods (d) %step3: get available algorithms
```

```
Design Methods for class fdesign.lowpass (N,Fc):


window
```

However, for a specification of `'Fp,Fst,Ap,Ast'`, a number of algorithms are available. If the user has only the Signal Processing Toolbox installed, the following algorithms are available:

```
>>set (d, 'specification', 'Fp,Fst,Ap,Ast')
>>designmethods(d)

Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
kaiserwin
```

If the user additionally has the DSP System Toolbox installed, the number of available algorithms for this response and specification string increases:

```
>>set(d,'specification','Fp,Fst,Ap,Ast')
>>designmethods(d)


Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
```

```
multistage
```

The user chooses a particular algorithm and implements the filter with the `design` function.

```
>>Hd=design(d,'butter');
```

The preceding code actually creates the filter, where `Hd` is the filter Implementation Object. This concept is discussed further in the next step.

If you do not perform this step explicitly, `design` automatically selects the optimum algorithm for the chosen response and specification.

Customizing the Algorithm

The customization options available for any given algorithm depend not only on the algorithm itself, selected in "Selecting an Algorithm" on page 3-6, but also on the specification selected in "Selecting a Specification" on page 3-4. To explore all the available options, type the following at the MATLAB command prompt:

```
help (d, 'algorithm-name')
```

where `d` is the Filter Specification Object, and `algorithm-name` is the name of the algorithm in single quotes, such as `'butter'` or `'cheby1'`.

The application of these customization options takes place while "Designing the Filter" on page 3-8, because these options are the properties of the filter Implementation Object, not the Specification Object.

If you do not perform this step explicitly, the optimum algorithm structure is selected.

Designing the Filter

This next task introduces a new object, the Filter Object, or `dfilt`. To create a filter, use the `design` command:

```
>> % design filter w/o specifying the algorithm
>> Hd = design(d);
```

where `Hd` is the Filter Object and `d` is the Specifications Object. This code creates a filter without specifying the algorithm. When the algorithm is not specified, the software selects the best available one.

To apply the algorithm chosen in "Selecting an Algorithm" on page 3-6, use the same `design` command, but specify the Butterworth algorithm as follows:

```
>> Hd = design(d, 'butter');
```

where `Hd` is the new Filter Object, and `d` is the Specifications Object.

To obtain help and see all the available options, type:

```
>> help fdesign/design
```

This help command describes not only the options for the `design` command itself, but also options that pertain to the method or the algorithm. If you are customizing the algorithm, you apply these options in this step. In the following example, you design a bandpass filter, and then modify the filter structure:

```
>> Hd = design(d, 'butter', 'filterstructure', 'df2sos')

f =

  FilterStructure: 'Direct-Form II, Second-Order Sections'
       Arithmetic: 'double'
        sosMatrix: [7x6 double]
      ScaleValues: [8x1 double]
 PersistentMemory: false
```

The filter design step, just like the first task of choosing a response, must be performed explicitly. A Filter Object is created only when `design` is called.

Design Analysis

After the filter is designed you may wish to analyze it to determine if the filter satisfies the design criteria. Filter analysis is broken into three main sections:

• Frequency domain analysis — Includes the magnitude response, group delay, and pole-zero plots.

- Time domain analysis — Includes impulse and step response

- Implementation analysis — Includes quantization noise and cost

To display help for analysis of a discrete-time filter, type:

```
>> help dfilt/analysis
```

To display help for analysis of a multirate filter, type:

```
>> help mfilt/functions
```

To display help for analysis of a farrow filter, type:

```
>> help farrow/functions
```

To analyze your filter, you must explicitly perform this step.

Realize or Apply the Filter to Input Data

After the filter is designed and optimized, it can be used to filter actual input data. The basic filter command takes input data x, filters it through the Filter Object, and produces output y:

```
>> y = filter (FilterObj, x)
```

This step is never automatically performed for you. To filter your data, you must explicitly execute this step. To understand how the filtering commands work, type:

```
>> help dfilt/filter
```

Note If you have Simulink®, you have the option of exporting this filter to a Simulink block using the realizemdl command. To get help on this command, type:

```
>> help realizemdl
```

4

Designing a Filter in the Filterbuilder GUI

The Graphical Interface to Fdesign

Introduction to Filterbuilder

The `filterbuilder` function provides a graphical interface to the `fdesign` object-object oriented filter design paradigm and is intended to reduce development time during the filter design process. `filterbuilder` uses a specification-centered approach to find the best algorithm for the desired response.

Note `filterbuilder` requires the Signal Processing Toolbox. The functionality of `filterbuilder` is greatly expanded by the DSP System Toolbox. Many of the features described or displayed below are only available if the DSP System Toolbox is installed. You may verify your installation by typing `ver` at the command prompt.

Filterbuilder Design Process

The design process when using `filterbuilder` is similar to the process outlined in the section titled Chapter 3, "Designing a Filter in Fdesign — Process Overview" in the Getting Started guide. The idea is to choose the constraints and specifications of the filter, and to use those as a starting point in the design. Postponing the choice of algorithm for the filter allows the best design method to be determined automatically, based upon the desired

performance criteria. The following are the details of each of the steps for designing a filter with `filterbuilder`.

Select a Response

When you open the `filterbuilder` tool by typing:

```
filterbuilder
```

at the MATLAB command prompt, the **Response Selection** dialog box appears, listing all possible filter responses available in DSP System Toolbox software.
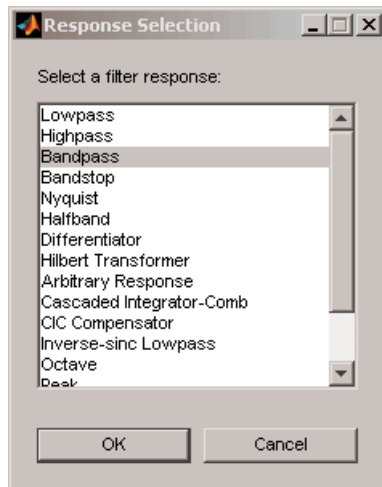

Note This step cannot be skipped because it is not automatically completed for you by the software. You must select a response to initiate the filter design process.

After you choose a response, say bandpass, you start the design of the Specifications Object, and the Bandpass Design dialog box appears. This dialog box contains a **Main** pane, a **Data Types** pane and a **Code Generation** pane. The specifications of your filter are generally set in the **Main** pane of the dialog box.

The **Data Types** pane provides settings for precision and data types, and the **Code Generation** pane contains options for various implementations of the completed filter design.

For the initial design of your filter, you will mostly use the **Main** pane.
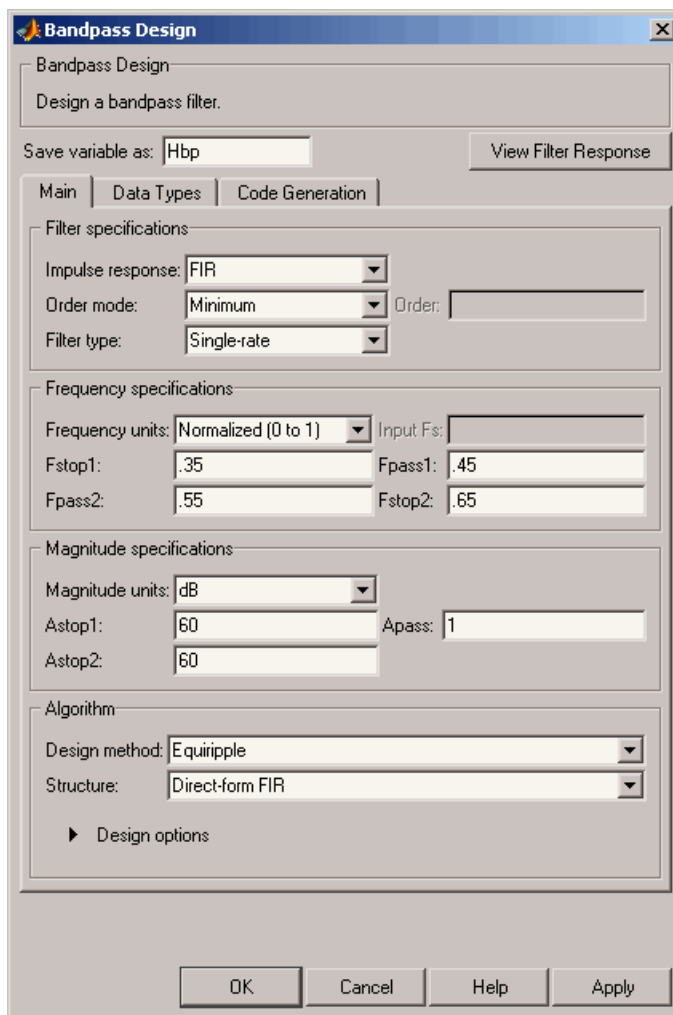
The **Bandpass Design** dialog box contains all the parameters you need to determine the specifications of a bandpass filter. The parameters listed in the **Main** pane depend upon the type of filter you are designing. However, no matter what type of filter you have chosen in the **Response Selection** dialog box, the filter design dialog box contains the **Main**, **Data Types**, and **Code Generation** panes.

Select a Specification

To choose the specification for the bandpass filter, you can begin by selecting an **Impulse Response**, **Order Mode**, and **Filter Type** in the **Filter Specifications** frame of the **Main Pane**. You can further specify the response of your filter by setting frequency and magnitude specifications in the appropriate frames on the **Main Pane**.

Note **Frequency**, **Magnitude**, and **Algorithm** specifications are interdependent and may change based upon your **Filter Specifications** selections. When choosing specifications for your filter, select your Filter Specifications first and work your way down the dialog box- this approach ensures that the best settings for dependent specifications display as available in the dialog box.

Select an Algorithm

The algorithms available for your filter depend upon the filter response and design parameters you have selected in the previous steps. For example, in the case of a bandpass filter, if the impulse response selected is IIR and the **Order Mode** field is set to `Minimum`, the design methods available is `Butterworth`, `Chebyshev` type I or II, or `Elliptic`, whereas if the **Order Mode** field is set to `Specify`, the design method available is `IIR least p-norm`.
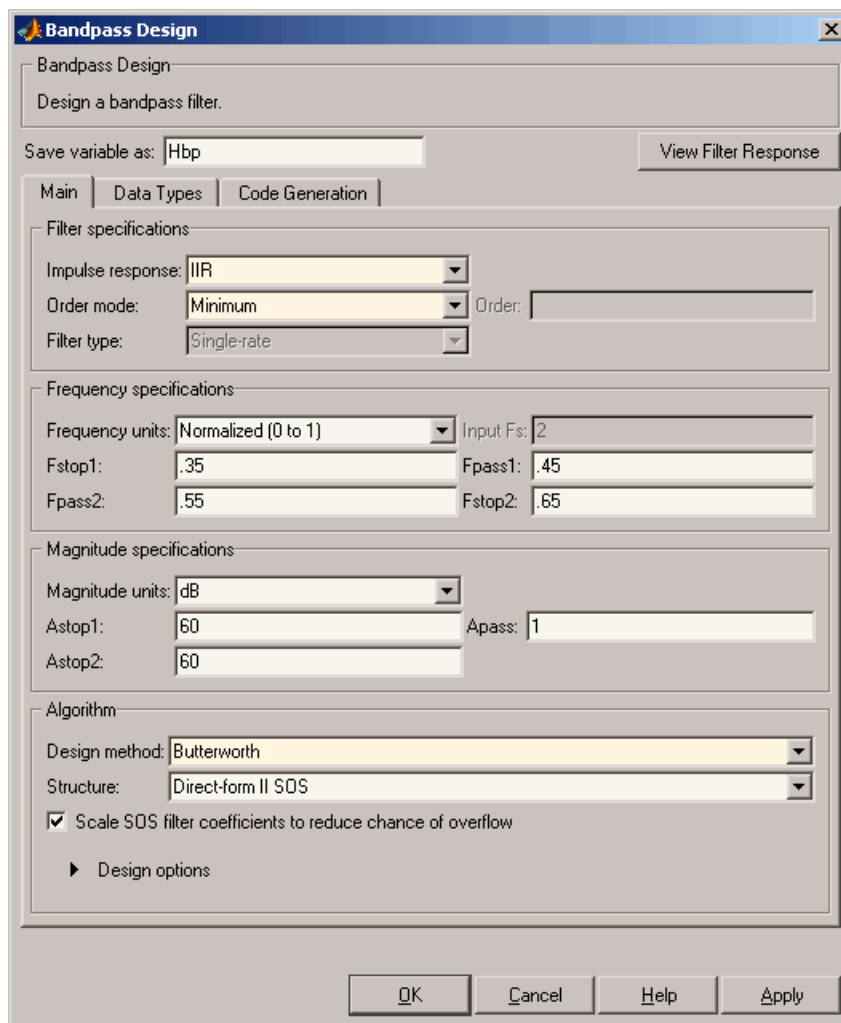
Customize the Algorithm

By expanding the **Design options** section of the **Algorithm** frame, you can further customize the algorithm specified. The options available will depend upon the algorithm and settings that have already been selected in the dialog box. In the case of a bandpass IIR filter using the Butterworth

method, design options such as **Match Exactly** are available, as shown in the following figure.
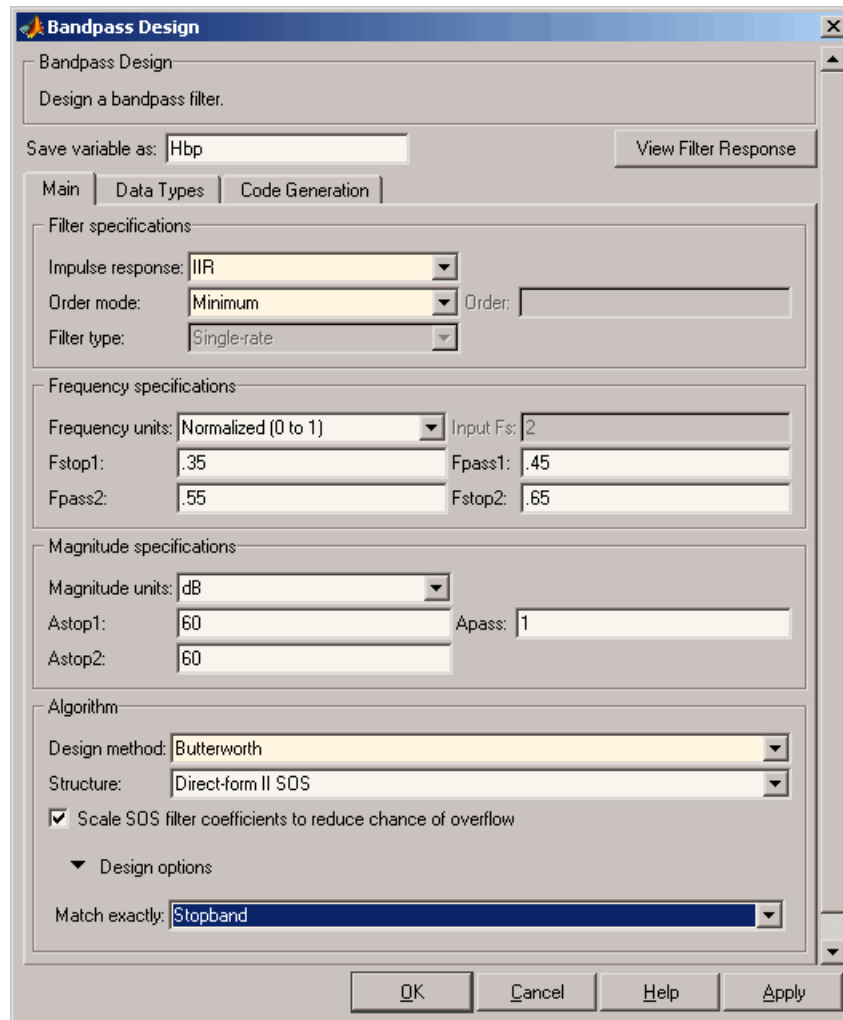
Analyze the Design

To analyze the filter response, click on the View Filter Response button. The Filter Visualization Tool opens displaying the magnitude plot of the filter response.
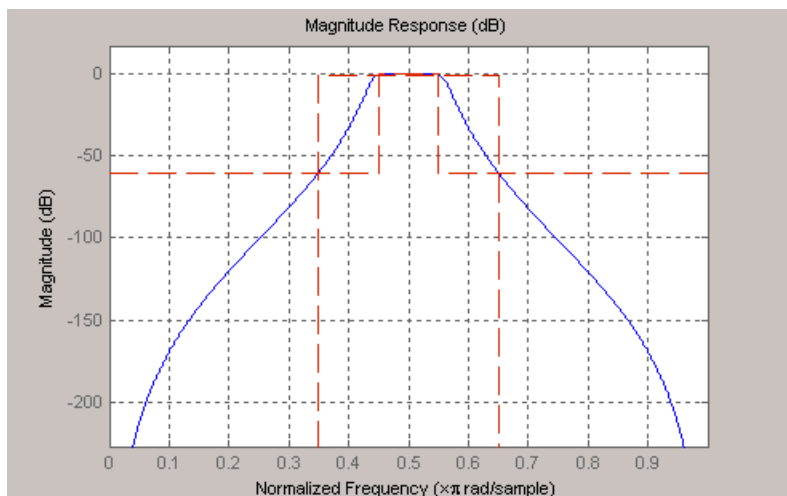
Realize or Apply the Filter to Input Data

When you have achieved the desired filter response through design iterations and analysis using the **Filter Visualization Tool**, apply the filter to the input data. Again, this step is never automatically performed for you by the software. To filter your data, you must explicitly execute this step. In the **Filter Visualization Tool**, click OK and DSP System Toolbox software creates the filter object with the name specified in the **Save variable as** field and exports it to the MATLAB workspace.

The filter is then ready to be used to filter actual input data. The basic filter command takes input data x, filters it through the Filter Object, and produces output y:

```
>> y = filter (FilterObj, x)
```

To understand how the filtering commands work, type:

```
>> help dfilt/filter
```

Tip If you have Simulink, you have the option of exporting this filter to a Simulink block using the `realizemdl` command. To get help on this command, type:

```
>> help realizemdl
```

Designing a FIR Filter Using filterbuilder

FIR Filter Design

Example – Using Filterbuilder to Design a Finite Impulse Response (FIR) Filter

To design a lowpass FIR filter using `filterbuilder`:

1 Open the Filterbuilder GUI by typing the following at the MATLAB prompt:

```
filterbuilder
```

The **Response Selection** dialog box appears. In this dialog box, you can select from a list of filter response types. Select `Lowpass` in the list box.
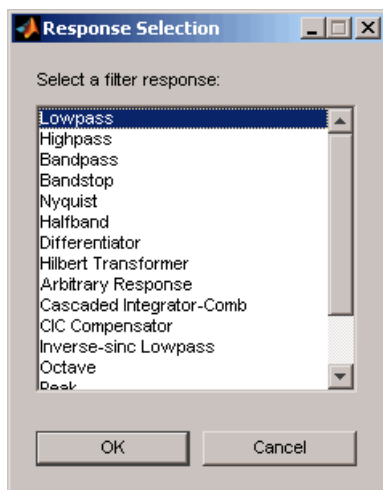
2 Hit the **OK** button. The **Lowpass Design** dialog box opens. Here you can specify the writable parameters of the Lowpass filter object. The components of the **Main** frame of this dialog box are described in the section titled *Lowpass Filter Design Dialog Box — Main Pane*. In the dialog box, make the following changes:

- Enter a Fpass value of 0.55.

- Enter a Fstop value of 0.65.
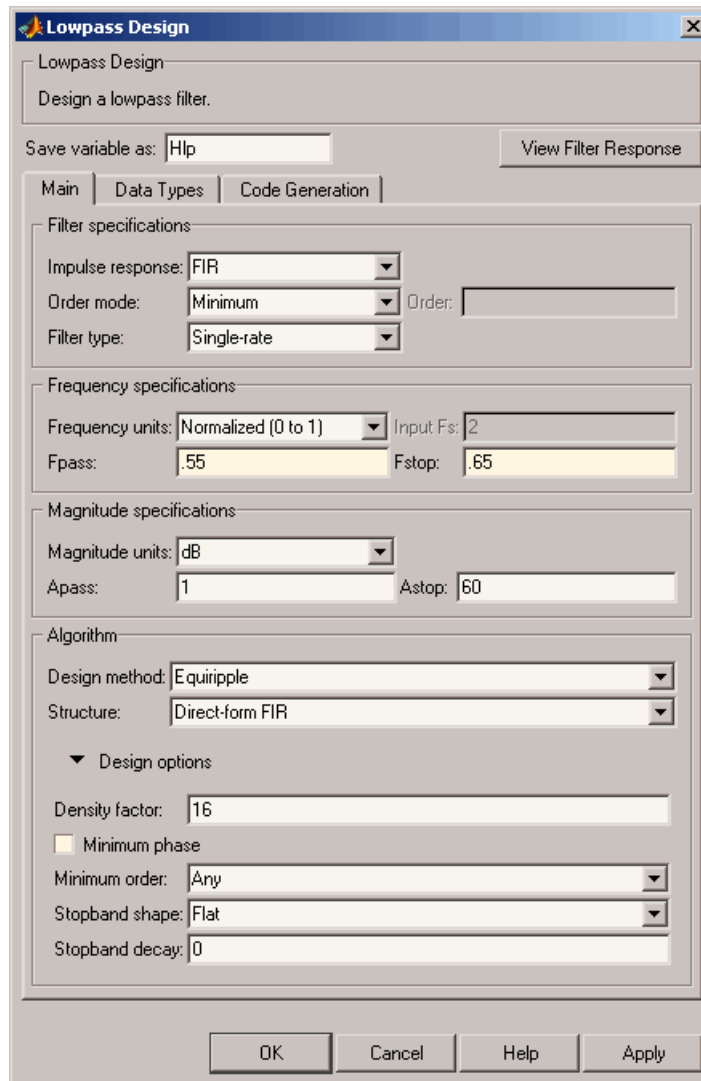
3 Click **Apply**, and the following message appears at the MATLAB prompt:

```
The variable 'Hlp' has been exported to the command window.
```

4 To check your design, click **View Filter Response**. The Filter Visualization tool appears, showing a plot of the magnitude response of the filter.
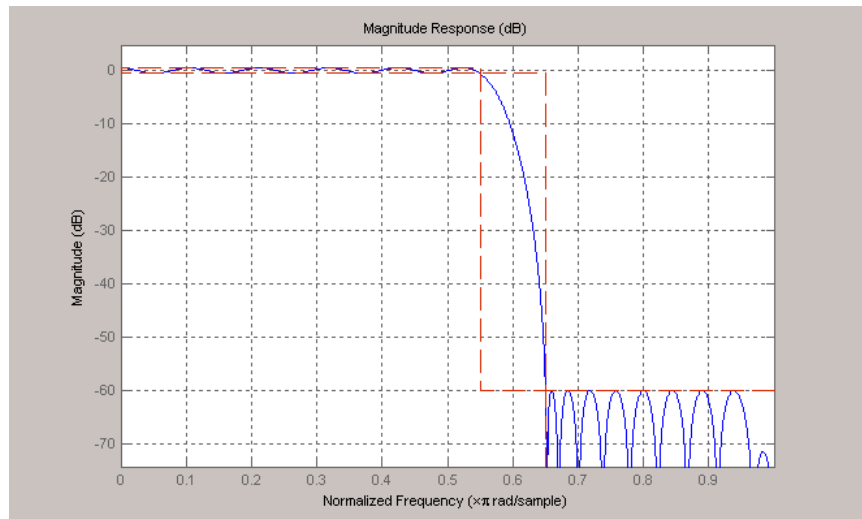
You can change the design and click **Apply**, followed by **View Filter Response**, as many times as needed until your design specifications are met.

5

FDATool: A Filter Design and Analysis GUI

Overview

Introduction to FDA Tool

The Filter Design and Analysis Tool (FDATool) is a user interface for designing and analyzing filters quickly. FDATool enables you to design digital FIR or IIR filters by setting filter specifications, by importing filters from your MATLAB workspace, or by adding, moving or deleting poles and zeros. FDATool also provides tools for analyzing filters, such as magnitude and phase response and pole-zero plots.

Integrated Products

FDATool seamlessly integrates additional functionality from other MathWorks® products as described in the following table.

Product	Added Features
Filter Design HDL Coder™	Generate synthesizable VHDL or Verilog for fixed-point filters
"DSP System Toolbox"	• Advanced FIR and IIR design techniques (see "Advanced Filter Design Methods" on page 5-4) • Filter transformations • Multirate filters

Product	Added Features
	• Fixed-point filters (available only with Simulink® Fixed Point™ product)
"Embedded Coder"	Export code usable by Code Composer Studio™ software
Simulink	Generate filters from atomic Simulink blocks

Filter Design Methods

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

Design Method	Function
Butterworth	`butter`
Chebyshev Type I	`cheby1`
Chebyshev Type II	`cheby2`
Elliptic	`ellip`
Maximally Flat	`maxflat`
Equiripple	`firpm`
Least-squares	`firls`
Constrained least-squares	`fircls`
Complex equiripple	`cfirpm`
Window	`fir1`

When using the `window` method in FDATool, all Signal Processing Toolbox window functions are available, and you can specify a user-defined window by entering its function name and input parameter.

Advanced Filter Design Methods

The following advanced filter design methods are available if you have DSP System Toolbox software.

Design Method	Function
Constrained equiripple FIR	`firceqrip`
Constrained-band equiripple FIR	`fircband`
Generalized remez FIR	`firgr`
Equripple halfband FIR	`firhalfband`
Least P-norm optimal FIR	`firlpnorm`
Equiripple Nyquist FIR	`firnyquist`
Interpolated FIR	`ifir`
IIR comb notching or peaking	`iircomb`
Allpass filter (given group delay)	`iirgrpdelay`
Least P-norm optimal IIR	`iirlpnorm`
Constrained least P-norm IIR	`iirlpnormc`
Second-order IIR notch	`iirnotch`
Second-order IIR peaking (resonator)	`iirpeak`

Using the Filter Design and Analysis Tool

There are different ways that you can design filters using the Filter Design and Analysis Tool. For example:

- You can first choose a response type, such as bandpass, and then choose from the available FIR or IIR filter design methods.

- You can specify the filter by its type alone, along with certain frequency- or time-domain specifications such as passband frequencies and stopband frequencies. The filter you design is then computed using the default filter design method and filter order.

Analyzing Filter Responses

Once you have designed your filter, you can display the filter coefficients and detailed filter information, export the coefficients to the MATLAB workspace, and create a C header file containing the coefficients, and analyze different filter responses in FDATool or in a separate Filter Visualization Tool (`fvtool`). The following filter responses are available:

- Magnitude response (`freqz`)

- Phase response (`phasez`)

- Group delay (`grpdelay`)

- Phase delay (`phasedelay`)

- Impulse response (`impz`)

- Step response (`stepz`)

- Pole-zero plots (`zplane`)

- Zero-phase response (`zerophase`)

Filter Design and Analysis Tool Panels

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

- Design Filter. See "Choosing a Filter Design Method" on page 5-9 for more information. You use this panel to

  - Design filters from scratch.

  - Modify existing filters designed in FDATool.

  - Analyze filters.

- Import filter. See "Importing a Filter Design" on page 5-30 for more information. You use this panel to

  - Import previously saved filters or filter coefficients that you have stored in the MATLAB workspace.

  - Analyze imported filters.

- Pole/Zero Editor. See "Editing the Filter Using the Pole/Zero Editor" on page 5-23. You use this panel to add, delete, and move poles and zeros in your filter design.

If you also have DSP System Toolbox product installed, additional panels are available:

- Set quantization parameters — Use this panel to quantize double-precision filters that you design in FDATool, quantize double-precision filters that you import into FDATool, and analyze quantized filters.

- Transform filter — Use this panel to change a filter from one response type to another.

- Multirate filter design — Use this panel to create a multirate filter from your existing FIR design, create CIC filters, and linear and hold interpolators.

If you have Simulink installed, this panel is available:

- Realize Model — Use this panel to create a Simulink block containing the filter structure. See "Exporting to a Simulink Model" on page 5-38 for more information.

Getting Help

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

Opening FDATool

To open FDATool, type

```
fdatool
```

at the MATLAB command prompt.

The Filter Design and Analysis Tool opens with the Design Filter panel displayed.
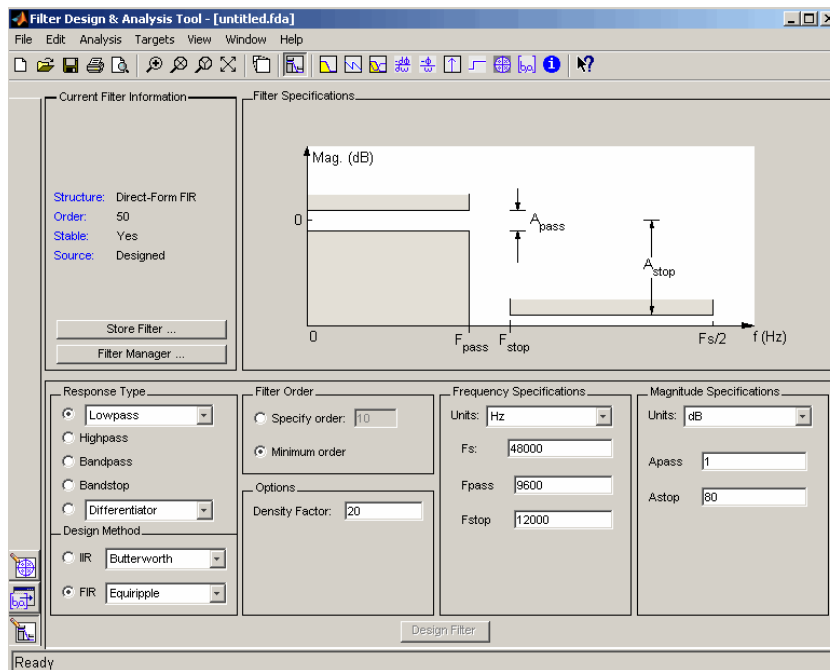
Note that when you open FDATool, **Design Filter** is not enabled. You must make a change to the default filter design in order to enable **Design Filter**. This is true each time you want to change the filter design. Changes to radio button items or drop down menu items such as those under **Response Type** or **Filter Order** enable **Design Filter** immediately. Changes to specifications in text boxes such as **Fs**, **Fpass**, and **Fstop** require you to click outside the text box to enable **Design Filter**.

Choosing a Response Type

You can choose from several response types:

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

Additional response types are available if you have the "DSP System Toolbox" software installed.

To design a bandpass filter, select the radio button next to **Bandpass** in the Response Type region of the GUI.
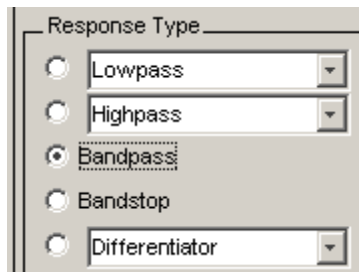
Note Not all filter design methods are available for all response types. Once you choose your response type, this may restrict the filter design methods available to you. Filter design methods that are not available for a selected response type are removed from the Design Method region of the GUI.

Choosing a Filter Design Method

You can use the default filter design method for the response type that you've selected, or you can select a filter design method from the available FIR and IIR methods listed in the GUI.

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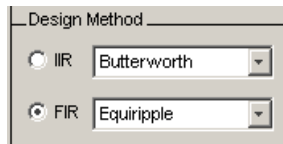
Setting the Filter Design Specifications

In this section...

Viewing Filter Specifications

The filter design specifications that you can set vary according to response type and design method. The display region illustrates filter specifications when you select **Analysis > Filter Specifications** or when you click the **Filter Specifications** toolbar button.

You can also view the filter specifications on the Magnitude plot of a designed filter by selecting **View > Specification Mask**.
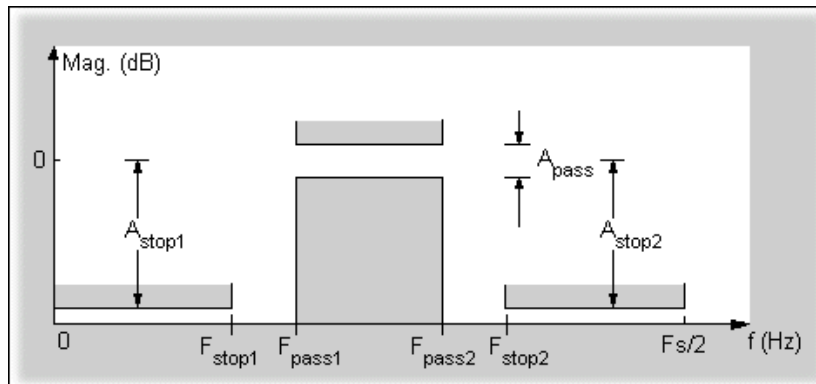
Filter Order

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

- **Specify order**: You enter the filter order in a text box.

- **Minimum order**: The filter design method determines the minimum order filter.

Select the **Minimum order** radio button for this example.
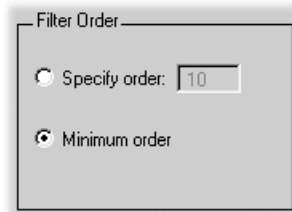
Note that filter order specification options depend on the filter design method you choose. Some filter methods may not have both options available.

Options

The available options depend on the selected filter design method. Only the FIR Equiripple and FIR Window design methods have settable options. For FIR Equiripple, the option is a **Density Factor**. See firpm for more information. For FIR Window the options are **Scale Passband**, **Window** selection, and for the following windows, a settable parameter:

Window	Parameter
Chebyshev (chebwin)	Sidelobe attenuation
Gaussian (gausswin)	Alpha
Kaiser (kaiser)	Beta
Taylor (taylorwin)	Nbar and Sidelobe level
Tukey (tukeywin)	Alpha
User Defined	Function Name, Parameter

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

For this example, set the **Density factor** to 16.
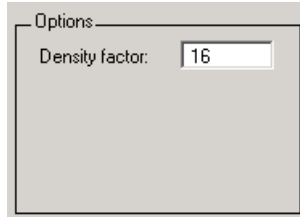
Bandpass Filter Frequency Specifications

For a bandpass filter, you can set

- Units of frequency:
  - Hz
  - kHz
  - MHz
  - 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.
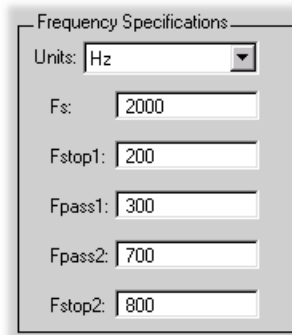
Similarly, you specify the stopband with two frequencies. The first frequency determines the upper edge of the first stopband, and the second frequency determines the lower edge of the second stopband.

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.

Bandpass Filter Magnitude Specifications

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

- Units for the magnitude response (dB or linear)

- Passband ripple

- Stopband attenuation

For this example:

- Keep **Units** in dB (default).

- Set the passband ripple (**Apass**) to 0.1 dB.

- Set the stopband attenuation for both stopbands (**Astop1**, **Astop2**) to 75 dB.

Computing the Filter Coefficients

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

Notice that the Design Filter button is disabled once you've computed the coefficients for your filter design. This button is enabled again once you make any changes to the filter specifications.

Analyzing the Filter

In this section...
"Displaying Filter Responses" on page 5-16
"Using Data Tips" on page 5-18
"Drawing Spectral Masks" on page 5-19
"Changing the Sampling Frequency" on page 5-20
"Displaying the Response in FVTool" on page 5-21

Displaying Filter Responses

You can view the following filter response characteristics in the display region or in a separate window (see "Displaying Filter Responses" on page 5-16):

- Magnitude response

- Phase response

- Magnitude and Phase responses

- Group delay response

- Phase delay response

- Impulse response

- Step response

- Pole-zero plot

- Zero-phase response — available from the *y*-axis context menu in a Magnitude or Magnitude and Phase response plot.

If you have DSP System Toolbox product installed, two other analyses are available: magnitude response estimate and round-off noise power. These two analyses are the only ones that use filter internals.

For descriptions of the above responses and their associated toolbar buttons and other FDATool toolbar buttons, see `fvtool`.

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

You can also display the filter coefficients and detailed filter information in this region.

For all the analysis methods, except zero-phase response, you can access them from the **Analysis** menu, the Analysis Parameters dialog box from the context menu, or by using the toolbar buttons. For zero-phase, right-click the *y*-axis of the plot and select **Zero-phase** from the context menu.

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

Magnitude Response (dB)

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

Note You can use specification masks in FVTool only if FVTool was launched from FDATool.
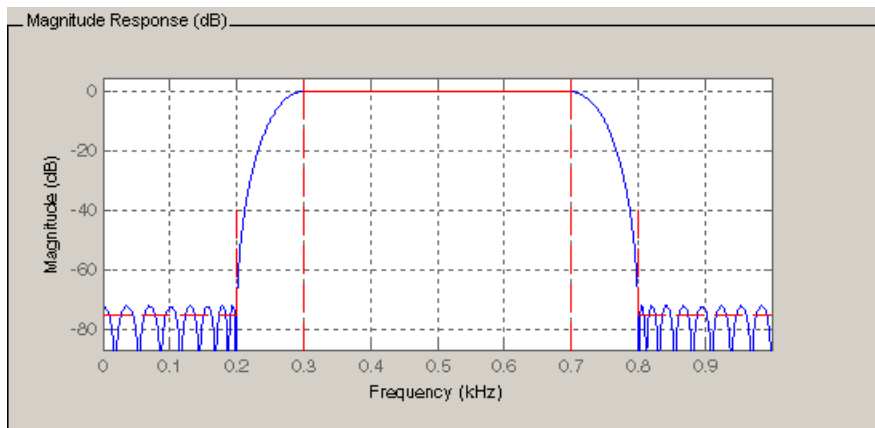
Using Data Tips

You can click the response to add plot data tips that display information about particular points on the response.
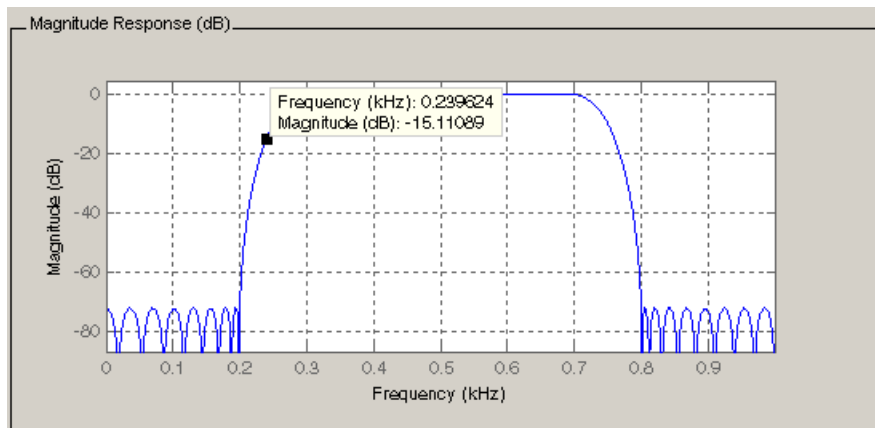
For information on using data tips, see "Data Cursor — Displaying Data Values Interactively" in the MATLAB documentation.

Drawing Spectral Masks

To add spectral masks or rejection area lines to your magnitude plot, click **View > User-defined Spectral Mask**.
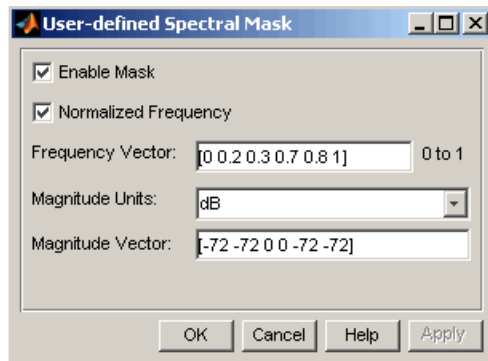
The mask is defined by a frequency vector and a magnitude vector. These vectors must be the same length.

- `Enable Mask` — Select to turn on the mask display.

- `Normalized Frequency` — Select to normalize the frequency between 0 and 1 across the displayed frequency range.

- `Frequency Vector` — Enter a vector of *x*-axis frequency values.

- `Magnitude Units` — Select the desired magnitude units. These units should match the units used in the magnitude plot.

- `Magnitude Vector` — Enter a vector of *y*-axis magnitude values.

The magnitude response below shows a spectral mask.
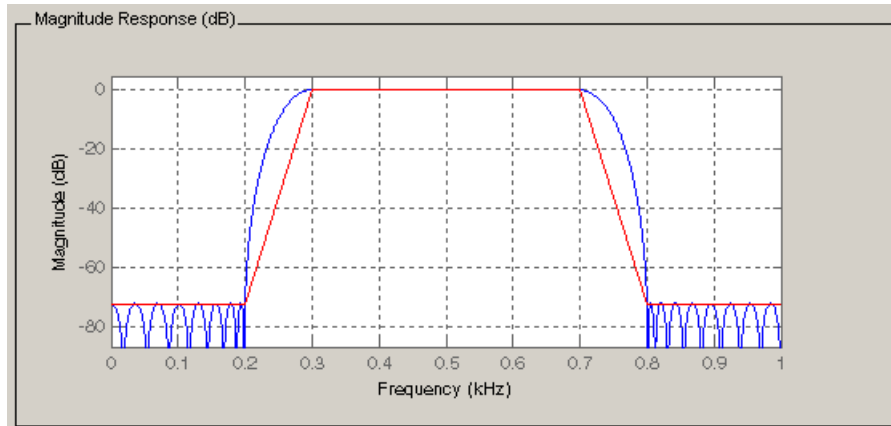
Changing the Sampling Frequency

To change the sampling frequency of your filter, right-click any filter response plot and select **Sampling Frequency** from the context menu.
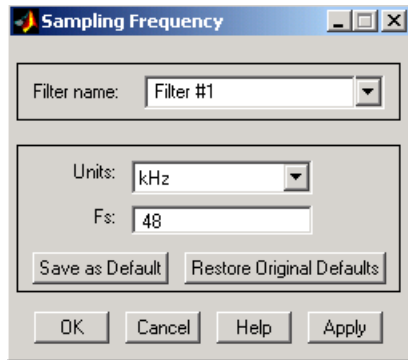
To change the filter name, type the new name in **Filter name**. (In fvtool, if you have multiple filters, select the desired filter and then enter the new name.)

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

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

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

Displaying the Response in FVTool

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

button:

This launches the Filter Visualization Tool (`fvtool`).

Note If Filter Specifications are shown in the display region, clicking the **Full View Analysis** toolbar button launches a MATLAB figure window instead of FVTool. The associated menu item is **Print to figure**, which is enabled only if the filter specifications are displayed.

You can use this tool to annotate your design, view other filter characteristics, and print your filter response. You can link FDATool and FVTool so that changes made in FDATool are immediately reflected in FVTool. See `fvtool` for more information.
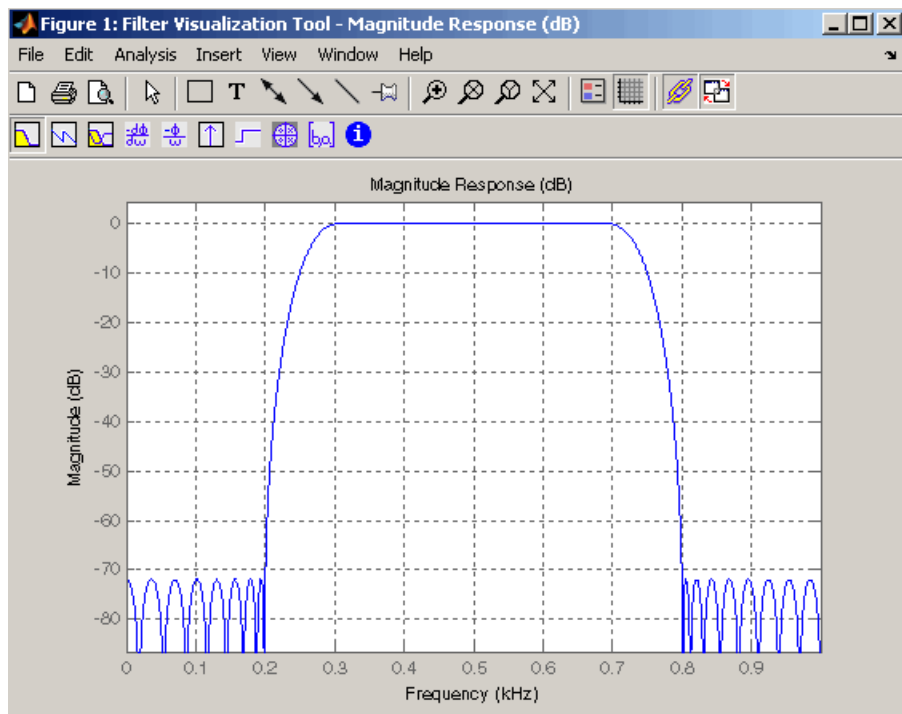
Editing the Filter Using the Pole/Zero Editor

In this section...

Displaying the Pole-Zero Plot

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

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

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

Click the **Pole/Zero Editor** button in the sidebar or select **Edit > Pole/Zero Editor** to display this panel.
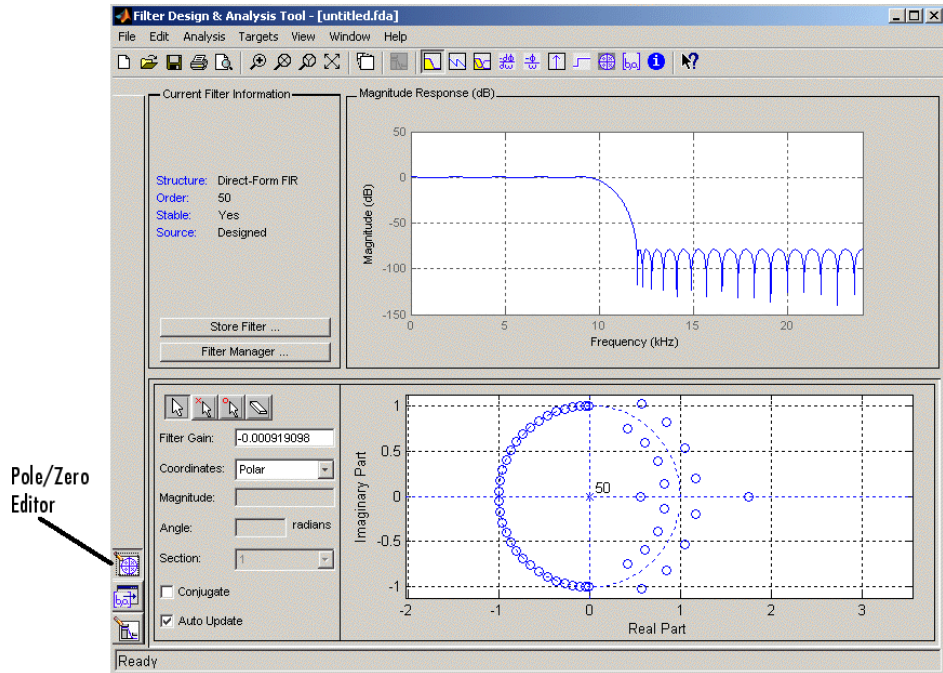
Poles are shown using x symbols and zeros are shown using o symbols.

Changing the Pole-Zero Plot

Plot mode buttons are located to the left of the pole/zero plot. Select one of the buttons to change the mode of the pole/zero plot. The Pole/Zero Editor has these buttons from left to right: move pole, add pole, add zero, and delete pole or zero.
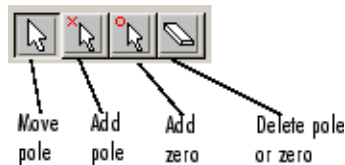
The following plot parameters and controls are located to the left of the pole/zero plot and below the plot mode buttons.

- **Filter gain** — factor to compensate for the filter's pole(s) and zero(s) gains

- **Coordinates** — units (`Polar` or `Rectangular`) of the selected pole or zero

- **Magnitude** — if polar coordinates is selected, magnitude of the selected pole or zero

- **Angle** — if polar coordinates is selected, angle of selected pole(s) or zero(s)

- **Real** — if rectangular coordinates is selected, real component of selected pole(s) or zero(s)

- **Imaginary** — if rectangular coordinates is selected, imaginary component of selected pole or zero

- **Section** — for multisection filters, number of the current section

- **Conjugate** — creates a corresponding conjugate pole or zero or automatically selects the conjugate pole or zero if it already exists.

- **Auto update** — immediately updates the displayed magnitude response when poles or zeros are added, moved, or deleted.

The **Edit > Pole/Zero Editor** has items for selecting multiple poles/zeros, for inverting and mirroring poles/zeros, and for deleting, scaling and rotating poles/zeros.

Select All
Select All Poles
Select All Zeros
Select Inside Unit Circle
Select Outside Unit Circle
Select Below Real Axis
Select Above Real Axis
Select Left Half
Select Right Half

Invert about the Real Axis
Invert about the Imaginary Axis
Invert about the Unit Circle

Mirror about the Real Axis
Mirror about the Imaginary Axis
Mirror about the Unit Circle

Delete current
Scale by a Factor
Rotate Counter-Clockwise

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

Moving one of the zeros on the vertical axis produces the following result:
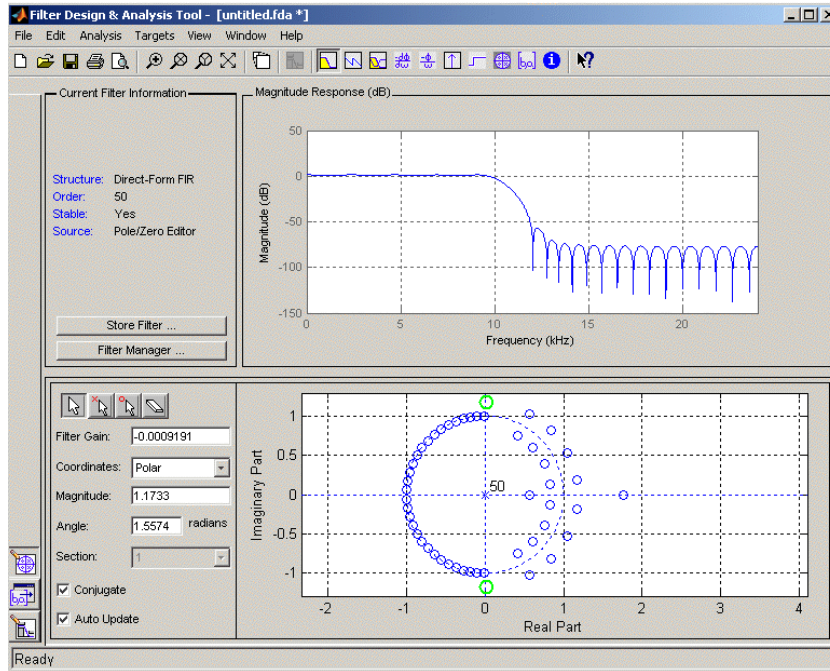
- The selected zero pair is shown in green.

- When you select one of the zeros from a conjugate pair, the Conjugate check box and the conjugate are automatically selected.

- The Magnitude Response plot updates immediately because **Auto update** is active.

Converting the Filter Structure

Converting to a New Structure

You can use **Edit > Convert Structure** to convert the current filter to a new structure. All filters can be converted to the following representations:

- Direct-form I
- Direct-form II
- Direct-form I transposed
- Direct-form II transposed
- Lattice ARMA

Note If you have DSP System Toolbox product installed, you will see additional structures in the Convert structure dialog box.

In addition, the following conversions are available for particular classes of filters:

- Minimum phase FIR filters can be converted to Lattice minimum phase
- Maximum phase FIR filters can be converted to Lattice maximum phase
- Allpass filters can be converted to Lattice allpass
- IIR filters can be converted to Lattice ARMA

Note Converting from one filter structure to another 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 roundoff computations.

For example:

- Select **Edit > Convert Structure** to open the Convert structure dialog box.

- Select Direct-form I in the list of filter structures.
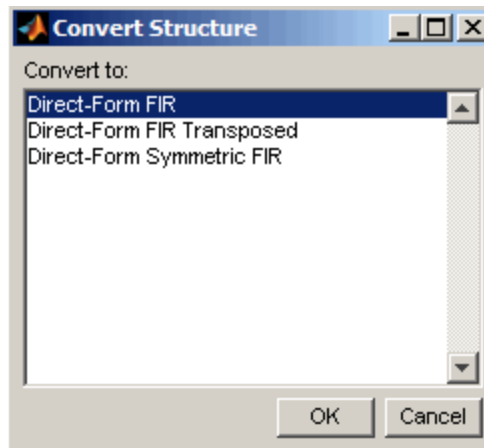
Converting to Second-Order Sections

You can use **Edit > Convert to Second-Order Sections** to store the converted filter structure as a collection of second-order sections rather than as a monolithic higher-order structure.

Note The following options are also used for **Edit > Reorder and Scale Scale Second-Order Sections**, which you use to modify an SOS filter structure.

The following **Scale** options are available when converting a direct-form II structure only:

- `None` (default)

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

- `L-infinity` (L^∞ norm)

The **Direction** (`Up` or `Down`) determines the ordering of the second-order sections. The optimal ordering changes depending on the **Scale** option selected.

For example:

- Select **Edit > Convert to Second-Order Sections** to open the Convert to SOS dialog box.

- Select `L-infinity` from the **Scale** menu for L^∞ norm scaling.

- Leave `Up` as the **Direction** option.

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

Importing a Filter Design
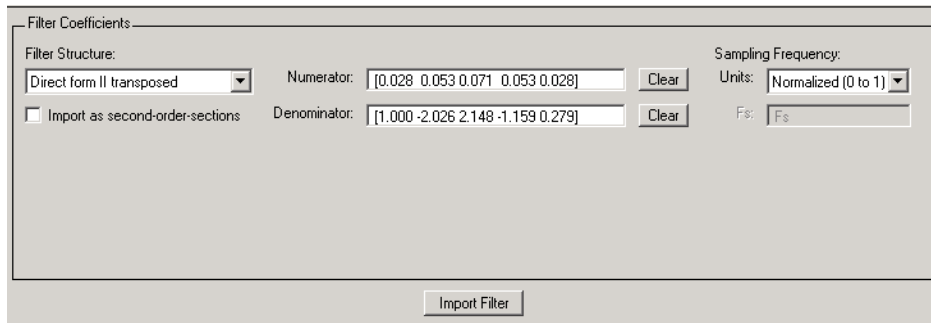
In this section...

Import Filter Panel

The Import Filter panel allows you to import a filter. You can access this region by clicking the **Import Filter** button in the sidebar.

The imported filter can be in any of the representations listed in the **Filter Structure** pull-down menu and described in "Filter Structures" on page 5-32. You can import a filter as second-order sections by selecting the check box.

Specify the filter coefficients in **Numerator** and **Denominator**, either by entering them explicitly or by referring to variables in the MATLAB workspace.

Select the frequency units from the following options in the **Units** menu, and for any frequency unit other than Normalized, specify the value or MATLAB workspace variable of the sampling frequency in the **Fs** field.

To import the filter, click the **Import Filter** button. The display region is automatically updated when the new filter has been imported.

You can edit the imported filter using the Pole/Zero Editor panel (see "Editing the Filter Using the Pole/Zero Editor" on page 5-23).

Filter Structures

The available filter structures are:

- "Direct-form" on page 5-32 , which includes direct-form I, direct-form II, direct-form I transposed, direct-form II transposed, and direct-form FIR

- "Lattice" on page 5-33, which includes lattice allpass, lattice MA min phase, lattice MA max phase, and lattice ARMA

- "Discrete-time Filter (dfilt object)" on page 5-34 (`dfilt` object)

The structure that you choose determines the type of coefficients that you need to specify in the text fields to the right.

Direct-form

For direct-form I, direct-form II, direct-form I transposed, and direct-form II transposed, specify the filter by its transfer function representation

$$H(z) = \frac{b(1) + b(2)z^{-1} + b(3)z^{-2} + \ldots b(m+1)z^{-m}}{a(1) + a(2)z^{-1} + a(3)Z^{-3} + \ldots a(n+1)z^{-n}}$$

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

- The **Denominator** field specifies a variable name or value for the denominator coefficient vector a, which contains `n+1` coefficients in descending powers of z. For FIR filters, the **Denominator** is `1`.

Filters in transfer function form can be produced by all of the Signal Processing Toolbox filter design functions (such as `fir1`, `fir2`, `firpm`, `butter`, `yulewalk`). See "Transfer Function" on page 1-23 for more information.

Importing as second-order sections. For all direct-form structures, except direct-form FIR, you an import the filter in its second-order section representation:

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

The **Gain** field specifies a variable name or a value for the gain G, and the **SOS Matrix** field specifies a variable name or a value for the L-by-6 SOS matrix

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

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

Filters in second-order section form can be produced by functions such as `tf2sos`, `zp2sos`, `ss2sos`, and `sosfilt`. See "Second-Order Sections (SOS)" on page 1-27 for more information.

Lattice

For lattice allpass, lattice minimum and maximum phase, and lattice ARMA filters, specify the filter by its lattice representation:

- For lattice allpass, the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, where N is the filter order.

- For lattice MA (minimum or maximum phase), the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, where N is the filter order.

- For lattice ARMA, the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, and the **Ladder coeff** field specifies the ladder coefficients, `v(1)` to `v(N+1)`, where N is the filter order.

Filters in lattice form can be produced by `tf2latc`. See "Lattice Structure" on page 1-28 for more information.

Discrete-time Filter (dfilt object)

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

Multirate Filter (mfilt object)

For Multirate filter, specify the name of the `mfilt` object. See `mfilt` in the DSP System Toolbox product for more information.

Exporting a Filter Design

Exporting Coefficients or Objects to the Workspace

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

1 Select **File > Export**. The Export dialog box appears.

2 Select `Workspace` from the **Export To** menu.

3 Select `Coefficients` from the **Export As** menu to save the filter coefficients or select `Objects` to save the filter in a filter object.

4 For coefficients, assign variable names using the **Numerator** (for FIR filters) or **Numerator** and **Denominator** (for IIR filters), or **SOS Matrix** and **Scale Values** (for IIR filters in second-order section form) text boxes in the Variable Names region.

For objects, assign the variable name in the **Discrete Filter (**or **Quantized Filter)** text box. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** check box.

5 Click the **Export** button.

Exporting Coefficients to an ASCII File

To save filter coefficients to a text file,

1 Select **File > Export**. The Export dialog box appears.

2 Select `Coefficients File (ASCII)` from the **Export To** menu.

3 Click the **Export** button. The Export Filter Coefficients to .FCF File dialog box appears.

4 Choose or enter a filename and click the **Save** button.

The coefficients are saved in the text file that you specified, and the MATLAB Editor opens to display the file. The text file also contains comments with the MATLAB version number, the Signal Processing Toolbox version number, and filter information.

Exporting Coefficients or Objects to a MAT-File

To save filter coefficients or a filter object as variables in a MAT-file:

1 Select **File > Export**. The Export dialog box appears.

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

3 Select Coefficients from the **Export As** menu to save the filter coefficients or select Objects to save the filter in a filter object.

4 For coefficients, assign variable names using the **Numerator** (for FIR filters) or **Numerator** and **Denominator** (for IIR filters), or **SOS Matrix** and **Scale Values** (for IIR filters in second-order section form) text boxes in the Variable Names region.

For objects, assign the variable name in the **Discrete Filter (**or **Quantized Filter)** text box. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** check box.

5 Click the **Export** button. The Export to a MAT-File dialog box appears.

6 Choose or enter a filename and click the **Save** button.

See also "Saving and Opening Filter Design Sessions" on page 5-49.

Exporting to SPTool

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

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

---

1 Select **File > Export**. The Export dialog box appears.

2 Select `SPTool` from the **Export To** menu.

3 Assign the variable name in the **Discrete Filter (**or **Quantized Filter)** text box. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** check box.

4 Click the **Export** button.

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

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

Exporting to a Simulink Model

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

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

Note If you have the DSP System Toolbox and Fixed-Point Toolbox™ installed, you can export a CIC filter to a Simulink model.

---

1 After designing your filter, click the **Realize Model** sidebar button or select **File > Export to Simulink Model**. The Realize Model panel is displayed.
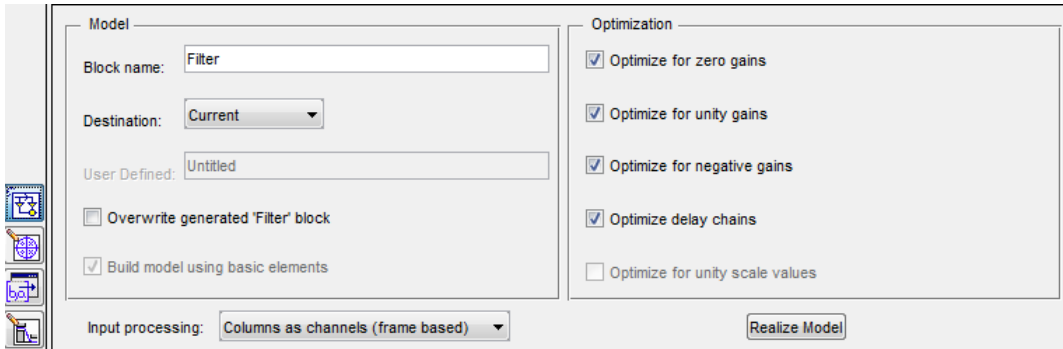
2 Specify the name to use for your block in **Block name**.

3 To insert the block into the current (most recently selected) Simulink model, set the **Destination** to Current. To inset the block into a new model, select New.

4 If you want to overwrite a block previously created from this panel, check **Overwrite generated 'Filter' block**.

5 If you select the **Build model using basic elements** check box, your filter is created as a subsystem block, which uses separate sub-elements. In this mode, the following optimization(s) are available:

- Optimize for zero gains — Removes zero-valued gain paths from the filter structure.

- Optimize for unity gains — Substitutes a wire (short circuit) for gains equal to 1 in the filter structure.

- Optimize for negative gains — Substitutes a wire (short circuit) for gains equal to -1 and changes corresponding additions to subtractions in the filter structure.

- Optimize delay chains — Substitutes delay chains composed of n unit delays with a single delay of n.

- Optimize for unity scale values — Removes multiplications for scale values equal to 1 from the filter structure.

The following illustration shows the effects of some of the optimizations:
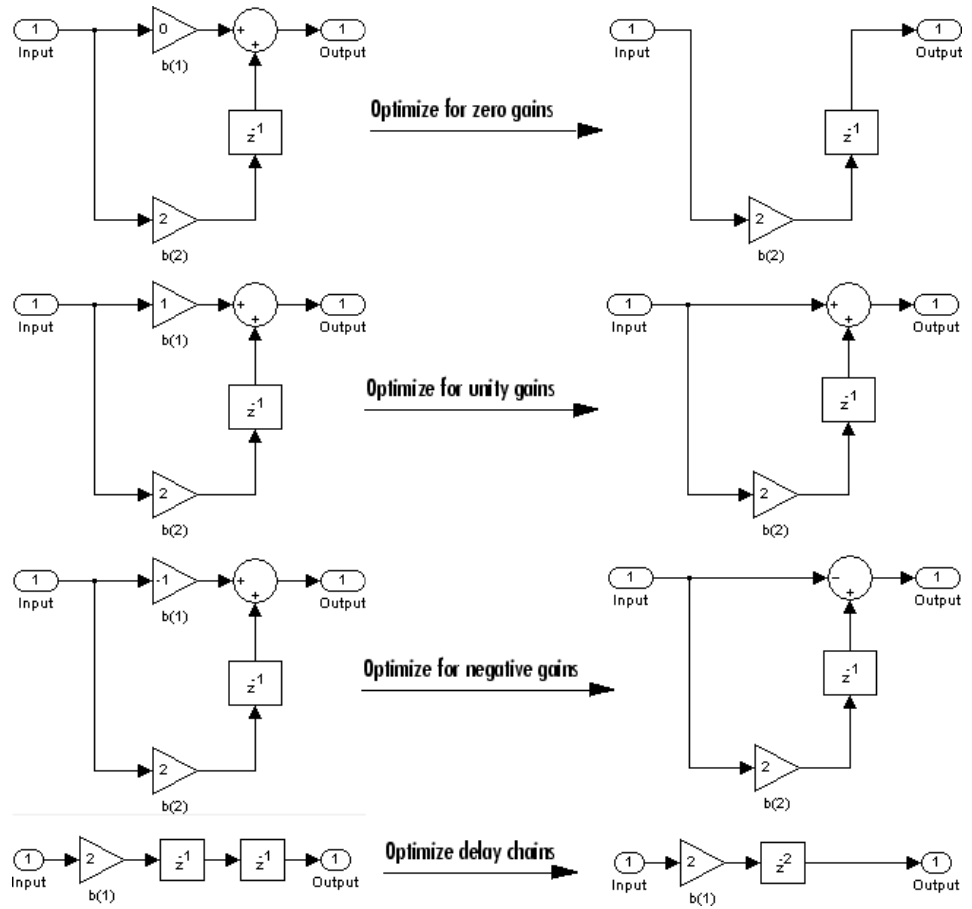
Optimization Effects

Note The **Build model using basic elements** check box is enabled only when you have a DSP System Toolbox license and your filter can be designed using a Digital Filter block. For more information, see the Filter Realization Wizard topic in the DSP System Toolbox documentation.

---

6 Set the **Input processing** parameter to specify whether the generated filter performs sample- or frame-based processing on the input. Depending on the type of filter you design, one or both of the following options may be available:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

7 Click the **Realize Model** button to create the filter block. When the **Build model using basic elements** check box is selected, FDATool implements the filter as a subsystem block using Sum, Gain, and Delay blocks.
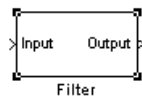
If you double-click the Simulink Filter block, the filter structure is displayed. The following figure shows the first section of the default four-section, direct form II filter.
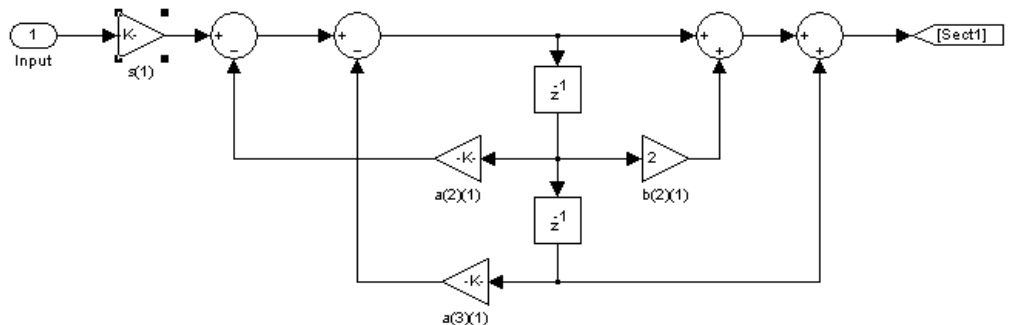
Other Ways to Export a Filter

You can also send your filter to a C header file or generate MATLAB code to construct your filter from the command line. For detailed instructions, see the following sections:

- "Generating a C Header File" on page 5-43
- "Generating MATLAB Code" on page 5-45

Generating a C Header File

You may want to include filter information in an external C program. To create a C header file with variables that contain filter parameter data, follow this procedure:

1 Select **Targets > Generate C Header**. The Generate C Header dialog box appears.
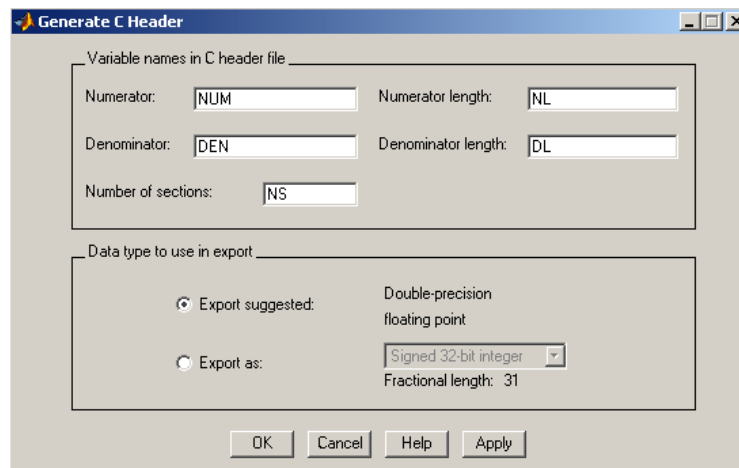
2 Enter the variable names to be used in the C header file. The particular filter structure determines the variables that are created in the file

Filter Structure	Variable Parameter
Direct-form I 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)

Filter Structure	Variable Parameter
Lattice MA	Lattice coeffs, Lattice coeffs length*, and Number of sections (inactive if filter has only one section)
Direct-form FIR Direct-form FIR transposed	Numerator, Numerator length*, and Number of sections (inactive if filter has only one section)

*length variables contain the total number of coefficients of that type.

Note Variable names cannot be C language reserved words, such as "for."

---

3 Select **Export Suggested** to use the suggested data type or select **Export As** and select the desired data type from the pull-down.

Note If you do not have DSP System Toolbox software installed, selecting any data type other than double-precision floating point results in a filter that does not exactly match the one you designed in the FDATool. This is due to rounding and truncating differences.

---

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

Generating MATLAB Code

You can generate MATLAB code that constructs the filter you designed in FDATool from the command line. Select **File > Generate MATLAB code** and specify the filename in the Generate MATLAB code dialog box.

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

The following is generated MATLAB code for the default lowpass filter in FDATool.

```
function Hd = ExFilter
%EXFILTER Returns a discrete-time filter object.

%
% MATLAB Code
% Generated by MATLAB(R) 7.11 and the Signal Processing Toolbox 6.14.
%
% Generated on: 17-Feb-2010 14:15:37
%

% Equiripple Lowpass filter designed using the FIRPM function.

% All frequency values are in Hz.
Fs = 48000;  % Sampling Frequency

Fpass = 9600;            % Passband Frequency
Fstop = 12000;           % Stopband Frequency
Dpass = 0.057501127785;  % Passband Ripple
Dstop = 0.0001;          % Stopband Attenuation
dens  = 20;              % Density Factor

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

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

```
Hd = dfilt.dffir(b);

% [EOF]
```

Managing Filters in the Current Session

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

You store and access saved filters with the **Store filter** and **Filter Manager** buttons, respectively, in the Current Filter Information pane.
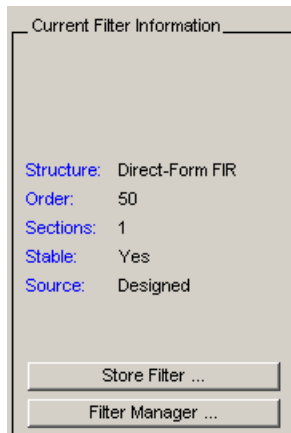
Store Filter — Displays the Store Filter dialog box in which you specify the filter name to use when storing the filter in the Filter Manager. The default name is the type of the filter.
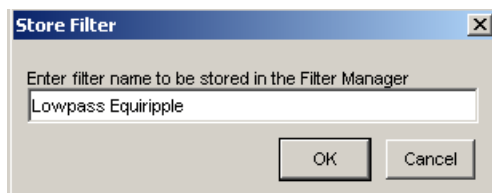
Filter Manager — Opens the Filter Manager.
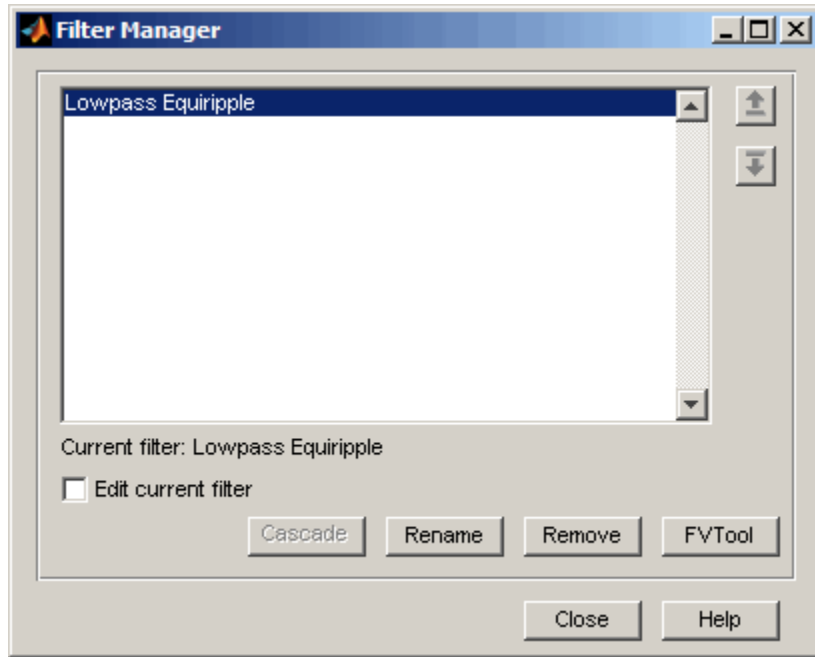
The current filter is listed below the listbox. To change the current filter, highlight the desired filter. If you select **Edit current filter**, FDATool displays the currently selected filter specifications. If you make any changes to the specifications, the stored filter is updated immediately.

To cascade two or more filters, highlight the desired filters and press **Cascade**. A new cascaded filter is added to the Filter Manager.

To change the name of a stored filter, press **Rename**. The Rename filter dialog box is displayed.

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

To export one or more filters to FVTool, highlight the filter(s) and press **FVTool**.

Saving and Opening Filter Design Sessions

You can save your filter design session as a MAT-file and return to the same session another time.

Select the **Save session** button ⊞ to save your session as a MAT-file. The first time you save a session, a Save Filter Design File browser opens, prompting you for a session name.
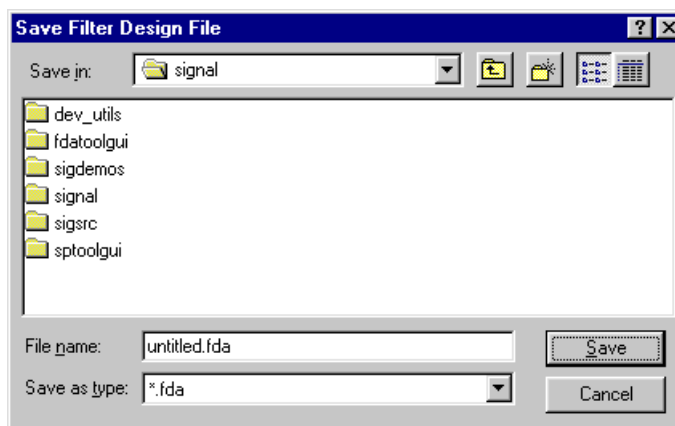
For example, save this design session as TestFilter.fda in your current working directory by typing TestFilter in the **File name** field.

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

Note You can also use the **File > Save session** and **File > Save session as** to save a session.

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

6

Statistical Signal Processing

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

Correlation and Covariance

Background Information

The cross-correlation sequence for two wide-sense stationary random process, x(n) and y(n) is:

$$R_{xy}(m) = E\{x(n+m)y^*(n)\}$$

where * denotes the complex conjugate and the expectation is over the ensemble of realizations that constitute the random processes.

Note that cross-correlation is not commutative, but a Hermitian (conjugate) symmetry property holds such that:

$$R_{xy}(m) = R_{yx}^*(-m)$$

The cross-covariance between x(n) and y(n) is:

$$C_{xy}(m) = E\{(x(n+m) - \mu_x)\ (y(n) - \mu_y)^*\} = R_{xy}(m) - \mu_x \mu_y^*$$

For zero-mean wide-sense stationary random processes, the cross-correlation and cross-covariance are equivalent.

In practice, you must estimate these sequences, because it is possible to access only a finite segment of the infinite-length random processes. Further, it is often necessary to estimate ensemble moments based on time averages because only a single realization of the random processes are available. A common estimate based on N samples of x_n and y_n is the deterministic cross-correlation sequence (also called the time-ambiguity function)

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

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.

Using xcorr and xcov Functions

The functions `xcorr` and `xcov` estimate the cross-correlation and cross-covariance sequences of random processes. They also handle autocorrelation and autocovariance as special cases. The `xcorr` function evaluates the sum shown above with an efficient FFT-based algorithm, given inputs x_n and y_n stored in length N vectors `x` and `y`. Its operation is equivalent to convolution with one of the two subsequences reversed in time.

For example:

```
x = [1 1 1 1 1]';
y = x;
xyc = xcorr(x,y)
```

Notice that the resulting sequence length is one less than twice the length of the input sequence. Thus, the 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

$$E\{R_{xy}(m)\} = (N - |m|)R_{xy}(m)$$

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 Nth 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-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

Background Information

The goal of *spectral estimation* is to describe the distribution (over frequency) of the power contained in a signal, based on a finite set of data. Estimation of power spectra is useful in a variety of applications, including the detection of signals buried in wideband noise.

The *power spectral density* (PSD) of a stationary random process x_n is mathematically related to the autocorrelation sequence by the discrete-time Fourier transform. In terms of normalized frequency, this is given by

$$P_{xx}(\omega) = \frac{1}{2\pi} \sum_{m=-\infty}^{\infty} R_{xx}(m) e^{-j\omega m}$$

This can be written as a function of physical frequency f (e.g., in hertz) by using the relation $\omega = 2\pi f / f_s$, where f_s is the sampling frequency.

$$P_{xx}(f) = \frac{1}{f_s} \sum_{m=-\infty}^{\infty} R_{xx}(m) e^{-j2\pi mf/f_s}$$

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

$$R_{xx}(m) = \int\limits_{-\pi}^{\pi} P_{xx}(\omega)e^{j\omega m}d\omega = \int\limits_{-f_s/2}^{f_s/2} P_{xx}(f)e^{j2\pi mf/f_s}df$$

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

$$R_{xx}(0) = \int\limits_{-\pi}^{\pi} P_{xx}(\omega)d\omega = \int\limits_{-f_s/2}^{f_s/2} P_{xx}(f)df$$

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

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

You can see from the above expression that $P_{xx}(\omega)$ represents the power content of a signal in an *infinitesimal* frequency band, which is why it is called the power spectral *density*.

The units of the PSD are power (e.g., watts) per unit of frequency. In the case of $P_{xx}(\omega)$, this is watts/radian/sample or simply watts/radian. In the case of $P_{xx}(f)$, the units are watts/hertz. Integration of the PSD with respect to frequency yields units of watts, as expected for the average power .

For real–valued signals, the PSD is symmetric about DC, and thus $P_{xx}(\omega)$ for $0 \le \omega \le \pi$ is sufficient to completely characterize the PSD. However, to obtain the average power over the entire Nyquist interval, it is necessary to introduce the concept of the *one-sided* PSD.

The one-sided PSD is given by

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

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

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

Spectral Estimation Method

The various methods of spectrum estimation available in the toolbox are categorized as follows:

- Nonparametric methods

- Parametric methods

- Subspace methods

Nonparametric methods are those in which the PSD is estimated directly from the signal itself. The simplest such method is the *periodogram*. Other nonparametric techniques such as *Welch's method* [8], the *multitaper method* (*MTM*) reduce the variance of the periodogram.

Parametric methods are those in which the PSD is estimated from a signal that is assumed to be output of a linear system driven by white noise. Examples are the *Yule-Walker autoregressive* (*AR*) *method* and the *Burg method*. These methods estimate the PSD by first estimating the parameters (coefficients) of the linear system that hypothetically generates the signal. They tend to produce better results than classical nonparametric methods when the data length of the available signal is relatively short. Parametric methods also produce smoother estimates of the PSD than nonparametric methods, but are subject to error from model misspecification.

Subspace methods, also known as *high-resolution methods* or *super-resolution methods*, generate frequency component estimates for a signal based on an eigenanalysis or eigendecomposition of the autocorrelation matrix. Examples are the *multiple signal classification* (*MUSIC*) *method* or the *eigenvector* (*EV*) *method*. These methods are best suited for line spectra — that is, spectra of sinusoidal signals — and are effective in the detection of sinusoids buried in noise, especially when the signal to noise ratios are low. The subspace methods do not yield true PSD estimates: they do not preserve process power between the time and frequency domains, and the autocorrelation sequence cannot be recovered by taking the inverse Fourier transform of the frequency estimate.

All three categories of methods are listed in the table below with the corresponding toolbox function and spectrum object names. More information about each function is on the corresponding function reference page. See "Parametric Modeling" on page 7-13 for details about `lpc` and other parametric estimation functions.

Spectral Estimation Methods/Functions

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

Spectral Estimation Methods/Functions (Continued)

Method	Description	Functions
Modified Covariance	Autoregressive (AR) spectral estimation of a time-series by minimization of the forward and backward prediction errors	`spectrum.mcov, pmcov`
MUSIC	Multiple signal classification	`spectrum.music, pmusic`
Eigenvector	Pseudospectrum estimate	`spectrum.eigenvector, peig`

Nonparametric Methods

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

Periodogram

In general terms, one way of estimating the PSD of a process is to simply find the discrete-time Fourier transform of the samples of the process (usually done on a grid with an FFT) and appropriately scale 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

$$P_{xx}(f) = \frac{1}{LF_s} \left| \sum_{n=0}^{L-1} x_L(n) e^{-j2\pi fn/F_s} \right|^2$$

where F_s is the sampling frequency.

In practice, the actual computation of $P_{xx}(f)$ can be performed only at a finite number of frequency points, and usually employs an FFT. Most

implementations of the periodogram method compute the *N*-point PSD estimate at the frequencies

$$f_k = \frac{kF_s}{N} \quad k = 0, 1, \ldots, N-1$$

In some cases, the computation of the periodogram via an FFT algorithm is more efficient if the number of frequencies is a power of two. Therefore it is not uncommon to pad the input signal with zeros to extend its length to a power of two.

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

```
fs = 1000;                % Sampling frequency
t = (0:fs)/fs;            % One second worth of samples
A = [1 2];                % Sinusoid amplitudes (row vector)
f = [150;140];            % Sinusoid frequencies (column vector)
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
```

Note The three last lines illustrate a convenient and general way to express the sum of sinusoids.

Together they are equivalent to xn = sin(2*pi*150*t) + 2*sin(2*pi*140*t) + 0.1*randn(size(t));

The periodogram estimate of the PSD can be computed by creating a periodogram object. In this case, the data vector is multiplied by a Hamming window.

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

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

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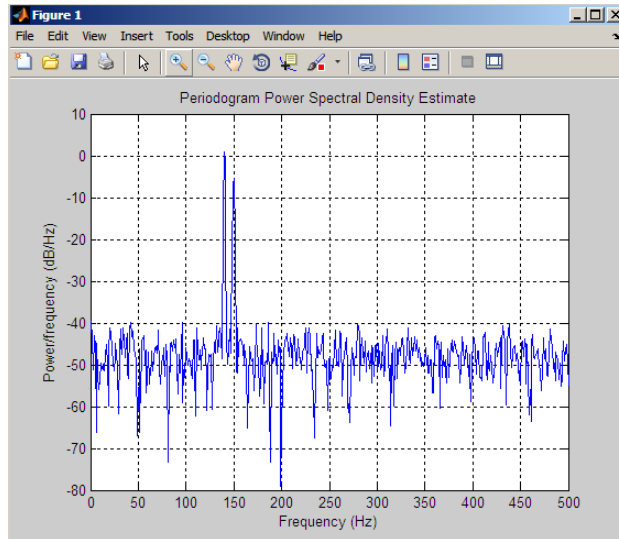

The average power can be computed by approximating the integral with the following sum:

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

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

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

Algorithm. Spectrum.periodogram computes and scales the output of the FFT to produce the power vs. frequency plot as follows. For a detailed example, see Tech Note 1702 on the MathWorks Web site.

1 For rapid and efficient processing, make the input signal length (`nfft`) equal to a power of 2 by padding it with zeros.

2 If the input signal is real-valued, the magnitude of the resulting FFT is symmetric with respect to zero frequency (DC). For an even-length FFT, only the first (1+nfft/2) points are unique. Determine the number of unique values and keep only those unique points.

3 Take the squared magnitudes of the unique FFT values. Scale the squared magnitudes (except for DC) by $2F_s/N$, where N is the length of signal *prior* to any zero padding. Scale the DC value by F_s/N.

4 Create a frequency vector from the number of unique points, the nfft and the sampling frequency.

5 Plot the resulting magnitude squared FFT vs. the frequency.

Performance of the Periodogram

The following sections discuss the performance of the periodogram with regard to the issues of leakage, resolution, bias, and variance.

Spectral Leakage. Consider the PSD of a finite-length (length L) signal $x_L[n]$, as discussed in the "Periodogram" on page 6-9 section. It is frequently useful to interpret $x_L[n]$ as the result of multiplying an infinite signal, $x[n]$, by a finite-length rectangular window, $w_R[n]$:

$$x_L[n] = x[n]w_R[n].$$

Because multiplication in the time domain corresponds to convolution in the frequency domain, the expected value of the periodogram in the frequency domain is:

$$E\{P_{xx}(f)\} = \frac{1}{Fs}\int_{-Fs/2}^{Fs/2} \frac{\sin^2(L\pi(f-f')/Fs)}{L\sin^2(\pi(f-f')/Fs)}P_{xx}(f')df',$$

showing that the expected value of the periodogram is the convolution of the true PSD with the square of the Dirichlet kernel.

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

$$x(n) = \sum_{k=1}^{N} A_k e^{j\omega_k n}$$

Its spectrum is

$$X(\omega) = \sum_{k=1}^{N} A_k \delta(\omega - \omega_k)$$

which for a finite-length sequence becomes

$$X(\omega) = \int_{-\pi}^{\pi} (\sum_{k=1}^{N} A_k \delta(\ - \omega_k)) W_R(\omega - \ ) d\ = \ = \ \backslash \ = \ \backslash$$

The preceding equation is equal to:

$$X(\omega) = \sum_{k=1}^{N} A_k W_R(\omega - \omega_k)$$

So in the spectrum of the finite-length signal, the Dirac deltas have been replaced by terms of the form $W_R(\omega\text{-}\omega_k)$, which corresponds to the frequency response of a rectangular window centered on the frequency ω_k.

The frequency response of a rectangular window has the shape of a sinc signal, as shown below.
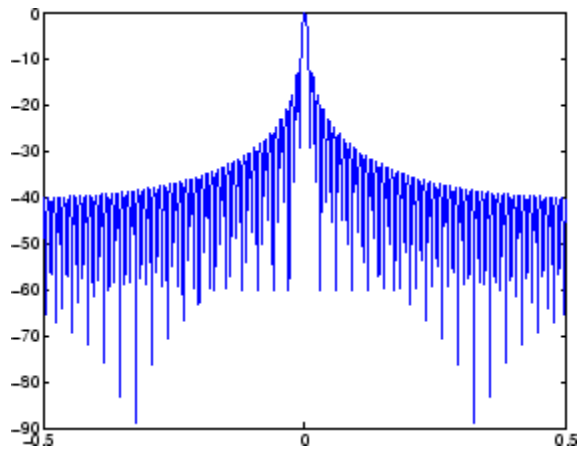
The plot displays a main lobe and several side lobes, the largest of which is approximately 13.5 dB below the mainlobe peak. These lobes account for the effect known as *spectral leakage*. While the infinite-length signal has its power concentrated exactly at the discrete frequency points f_k, the windowed (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:

```
fs = 1000;                    % Sampling frequency
t = (0:fs/10)/fs;             % One-tenth second worth of samples
A = [1 2];                    % Sinusoid amplitudes
f = [150;140];                % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
Hs = spectrum.periodogram;
psd(Hs,xn,'Fs',fs,'NFFT',1024)
```
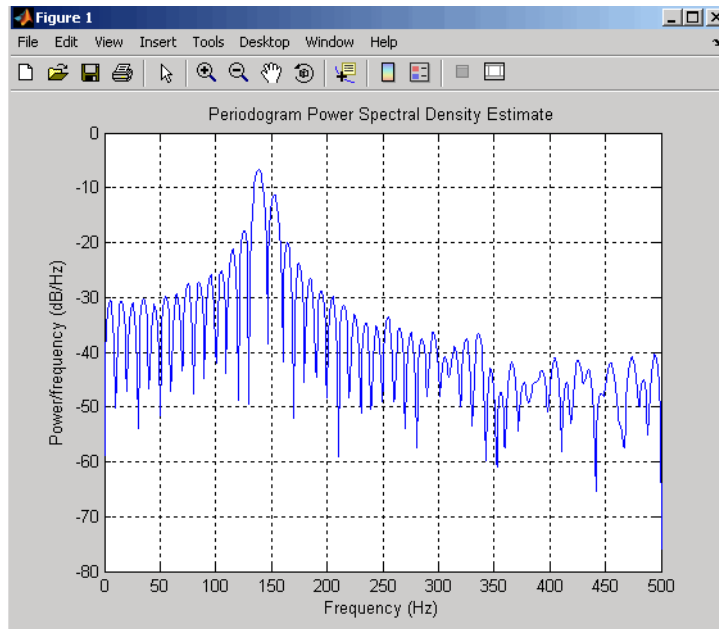
It is important to note that the effect of spectral leakage is contingent solely on the length of the data record. It is *not* a consequence of the fact that the periodogram is computed at a finite number of frequency samples.

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

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

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

$$(f_2 - f_1) > \frac{Fs}{L}$$

In the example above, where two sinusoids are separated by only 10 Hz, the data record must be greater than 100 samples to allow resolution of two distinct sinusoids by a periodogram.

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

```
fs = 1000;                  % Sampling frequency
t = (0:fs/15)./fs;          % 67 samples
A = [1 2];                  % Sinusoid amplitudes
f = [150;140];              % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
Hs=spectrum.periodogram;
psd(Hs,xn,'Fs',fs,'NFFT',1024)
```
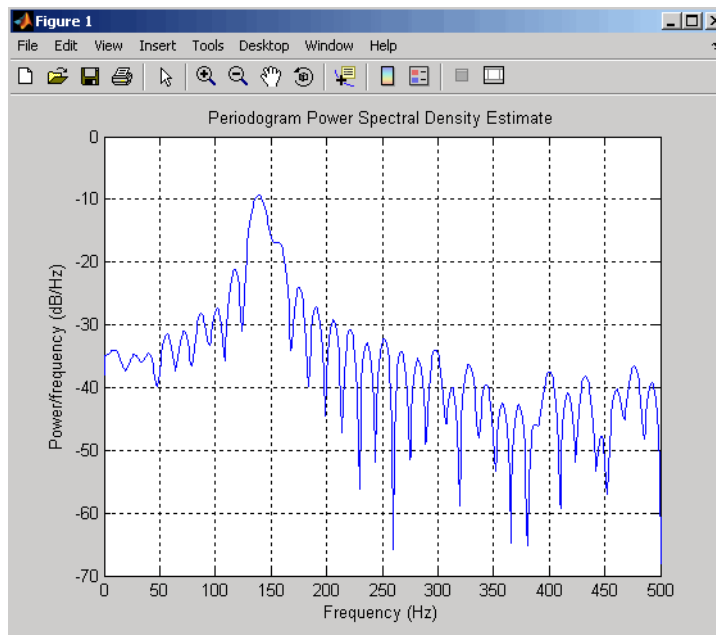

The above discussion about resolution did not consider the effects of noise since the signal-to-noise ratio (SNR) has been relatively high thus far. When the SNR is low, true spectral features are much harder to distinguish, and

noise artifacts appear in spectral estimates based on the periodogram. The example below illustrates this:

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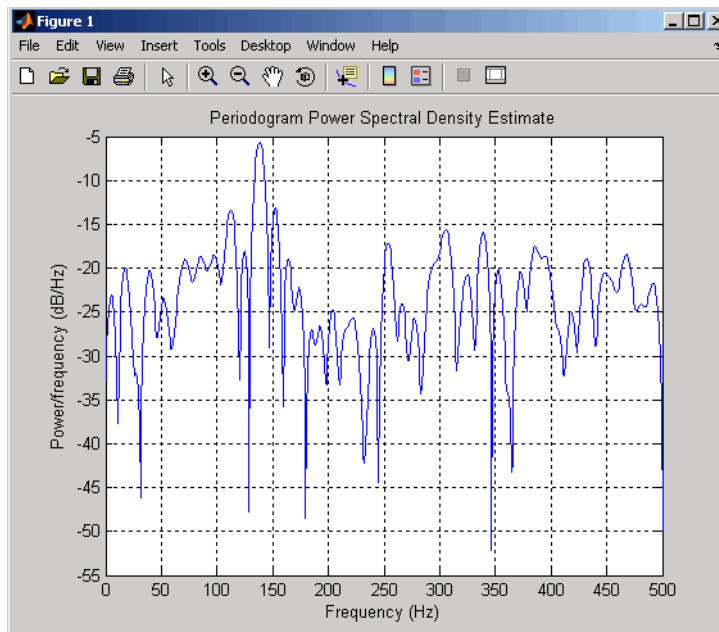

Bias of the Periodogram. The periodogram is a biased estimator of the PSD. Its expected value was previously shown to be:

$$E\{P_{xx}(f)\} = \frac{1}{Fs} \int_{-Fs/2}^{Fs/2} \frac{\sin^2(L\pi(f-f')/Fs)}{L\sin^2(\pi(f-f')/Fs)} P_{xx}(f')df',$$

The periodogram is asymptotically unbiased, which is evident from the earlier observation that as the data record length tends to infinity, the frequency response of the rectangular window more closely approximates the Dirac delta function. However, in some cases the periodogram is a poor estimator of the PSD even when the data record is long. This is due to the variance of the periodogram, as explained below.

Variance of the Periodogram. The variance of the periodogram can be shown to be:

$$\text{Var}(P_{xx}(f)) = \begin{cases} P_{xx}^2(f) & 0 < f < \text{Fs}\,/\,2 \\ 2P_{xx}^2(f) & f = 0 \quad \text{or}\, f = Fs\,/\,2 \end{cases}$$

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

The Modified Periodogram

The *modified periodogram* windows the time-domain signal prior to computing the DFT in order to smooth the edges of the signal. This has the effect of reducing the height of the sidelobes or spectral leakage. This phenomenon gives rise to the interpretation of sidelobes as spurious frequencies introduced into the signal by the abrupt truncation that occurs when a rectangular window is used. For nonrectangular windows, the end points of the truncated signal are attenuated smoothly, and hence the spurious frequencies introduced are much less severe. On the other hand, nonrectangular windows also broaden the mainlobe, which results in a reduction of resolution.

`spectrum.periodogram` allows you to compute a modified periodogram by specifying the window to be used on the data. For example, compare a default rectangular window and a Hamming window:

```
fs = 1000;               % Sampling frequency
t = (0:fs/10)./fs;       % One-tenth second worth of samples
A = [1 2];               % Sinusoid amplitudes
```

```
f = [150;140];                % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
Hrect = spectrum.periodogram;
psd(Hrect,xn,'Fs',fs,'NFFT',1024);
```
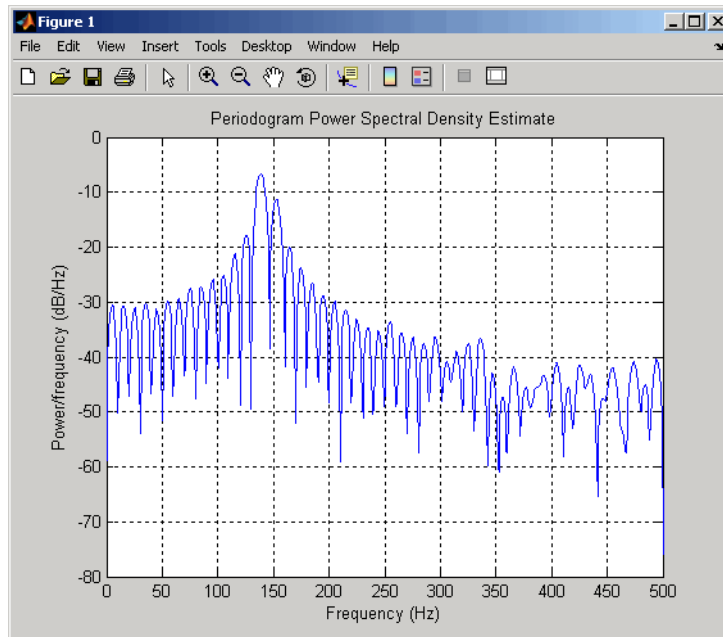
```
Hhamm = spectrum.periodogram('Hamming');
psd(Hhamm,xn,'Fs',fs,'NFFT',1024);
```
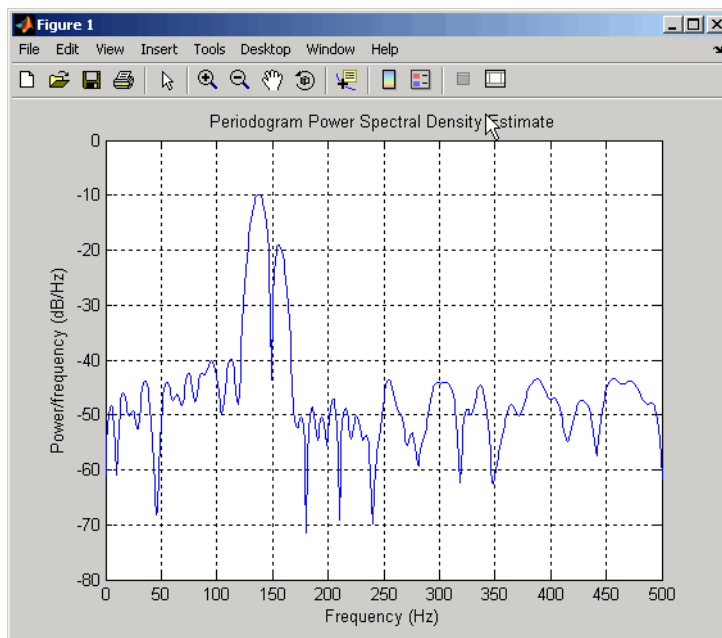

You can verify that although the sidelobes are much less evident in the Hamming-windowed periodogram, the two main peaks are wider. In fact, the 3 dB width of the mainlobe corresponding to a Hamming window is approximately twice that of a rectangular window. Hence, for a fixed data length, the PSD resolution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

Nonrectangular windowing affects the average power of a signal because some of the time samples are attenuated when multiplied by the window. To compensate for this, spectrum.periodogram and spectrum.welch normalize the window to have an average power of unity. This ensures that the measured average power is generally independent of window choice. If the frequency components are not well resolved by the PSD estimators, the window choice does affect the average power.

The modified periodogram estimate of the PSD is

$$\hat{P}_{xx}(f) = \frac{|X(f)|^2}{FsLU}$$

where U is the window normalization constant.

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

For large values of L, U tends to become independent of window length. The addition of U as a normalization constant ensures that the modified periodogram is asymptotically unbiased.

Welch's Method

An improved estimator of the PSD is the one proposed by Welch [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 toolbox by the `spectrum.welch` object or `pwelch` function

The averaging of modified periodograms tends to decrease the variance of the estimate relative to a single periodogram estimate of the entire data record. Although overlap between segments introduces redundant information, this effect is diminished by the use of a nonrectangular window, which reduces the importance or *weight* given to the end samples of segments (the samples that overlap).

However, as mentioned above, the combined use of short data records and nonrectangular windows results in reduced resolution of the estimator. In summary, there is a trade-off 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:

```
fs = 1000;                % Sampling frequency
t = (0:0.3*fs)./fs;       % 301 samples
A = [2 8];                % Sinusoid amplitudes (row vector)
f = [150;140];            % Sinusoid frequencies (column vector)
xn = A*sin(2*pi*f*t) + 5*randn(size(t));
Hs = spectrum.periodogram('rectangular');
psd(Hs,xn,'Fs',fs,'NFFT',1024);
```
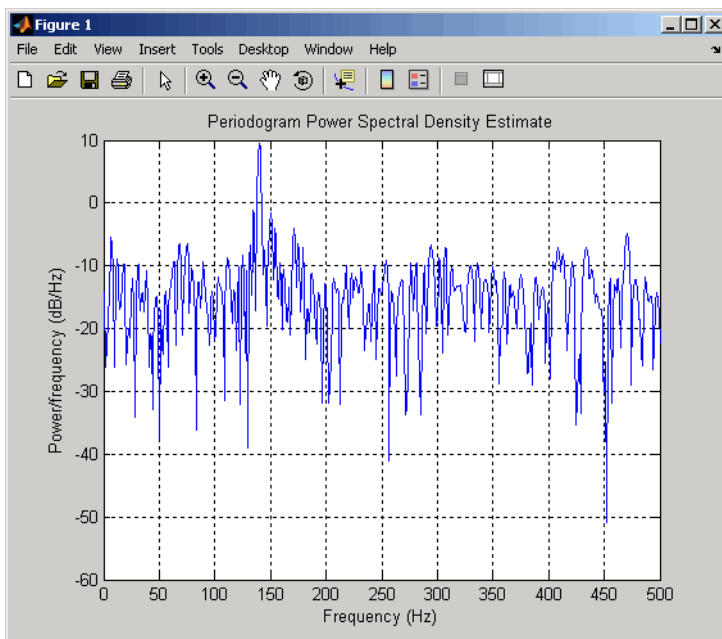

We can obtain Welch's spectral estimate for 3 segments with 50% overlap with

```
Hs = spectrum.welch('rectangular',150,50);
psd(Hs,xn,'Fs',fs,'NFFT',512);
```
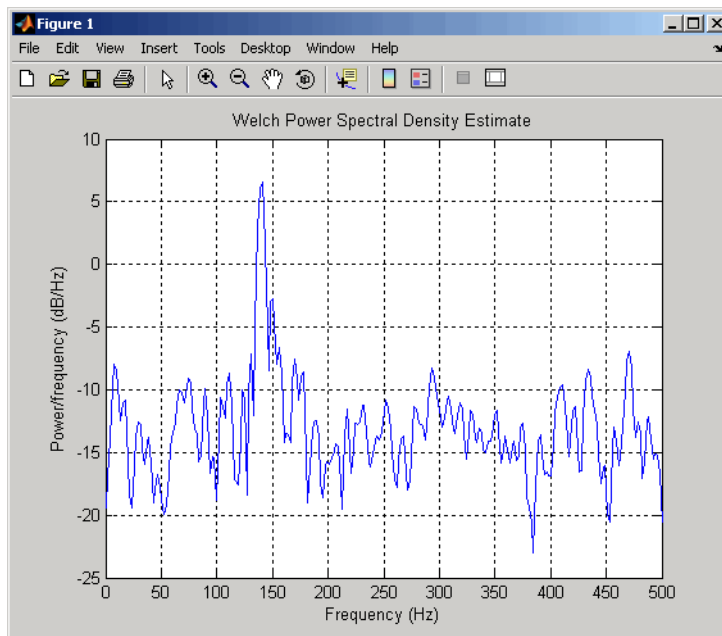
In the periodogram above, noise and the leakage make one of the sinusoids essentially indistinguishable from the artificial peaks. In contrast, although the PSD produced by Welch's method has wider peaks, you can still distinguish the two sinusoids, which stand out from the "noise floor."

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

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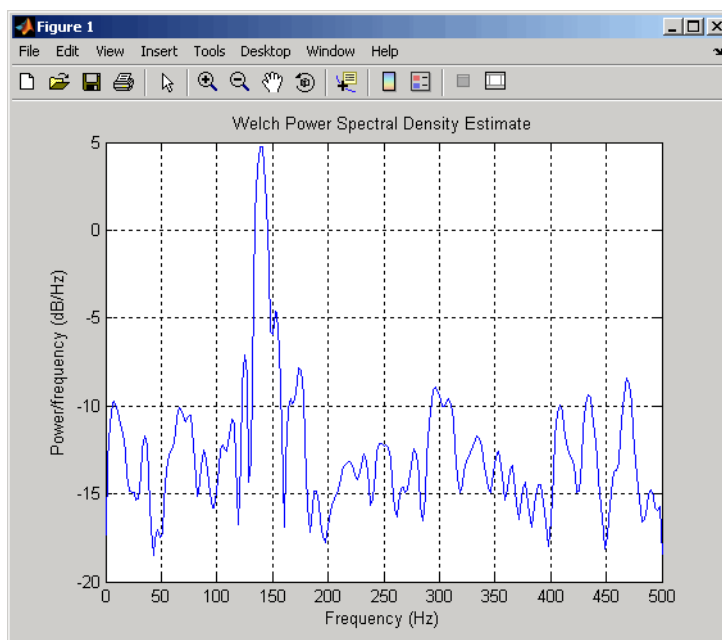

For a more detailed discussion of Welch's method of PSD estimation, see Kay [2] and Welch [8].

Bias and Normalization in Welch's Method

Welch's method yields a biased estimator of the PSD. The expected value of the PSD estimate is:

$$E\{P_{\text{Welch}}(f)\} = \frac{1}{FsLU} \int_{-Fs/2}^{F/2} |W(f - f')|^2 \ P_{xx}(f') df'$$

where L is the length of the data segments, U is the same normalization constant present in the definition of the modified periodogram, and $W(f)$ is the Fourier transform of the window function. As is the case for all periodograms, Welch's estimator is asymptotically unbiased. For a fixed length data record,

the bias of Welch's estimate is larger than that of the periodogram because the length of the segments is less than the length of the entire data sample.

The variance of Welch's estimator is difficult to compute because it depends on both the window used and the amount of overlap between segments. Basically, the variance is inversely proportional to the number of segments whose modified periodograms are being averaged.

Multitaper Method

The periodogram can be interpreted as filtering a length L signal, $x_L[n]$, through a filter bank (a set of filters in parallel) of L FIR bandpass filters. The 3 dB bandwidth of each of these bandpass filters can be shown to be approximately equal to f_s / L. The magnitude response of each one of these bandpass filters resembles that of the rectangular window discussed in "Spectral Leakage" on page 6-12. 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 `pmtm` and the object that implements it is `spectrum.mtm`. Use `spectrum.mtm` to compute the PSD of `xn` from the previous examples:

```
fs = 1000;                 % Sampling frequency
t = (0:fs)/fs;             % One second worth of samples
A = [1 2];                 % Sinusoid amplitudes
f = [150;140];             % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
Hs1 = spectrum.mtm(4,'adapt');
psd(Hs1,xn,'Fs',fs,'NFFT',1024)
```
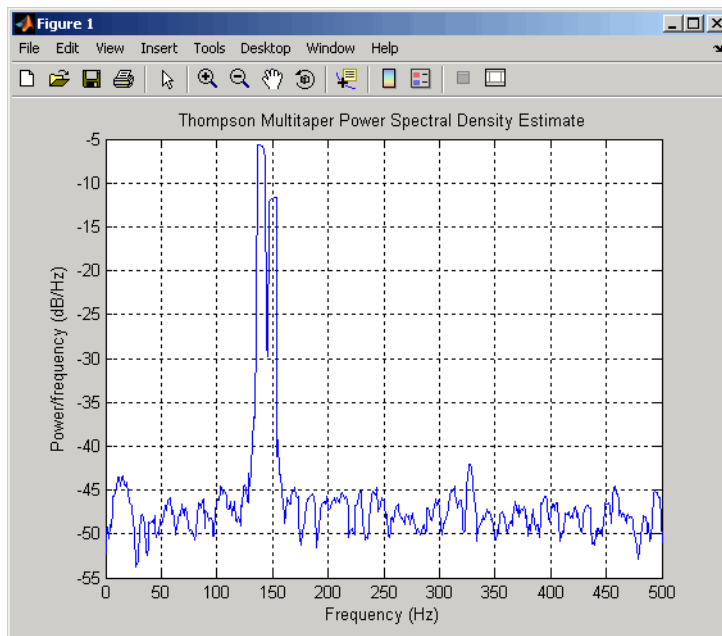
By lowering the time-bandwidth product, you can increase the resolution at the expense of larger variance:

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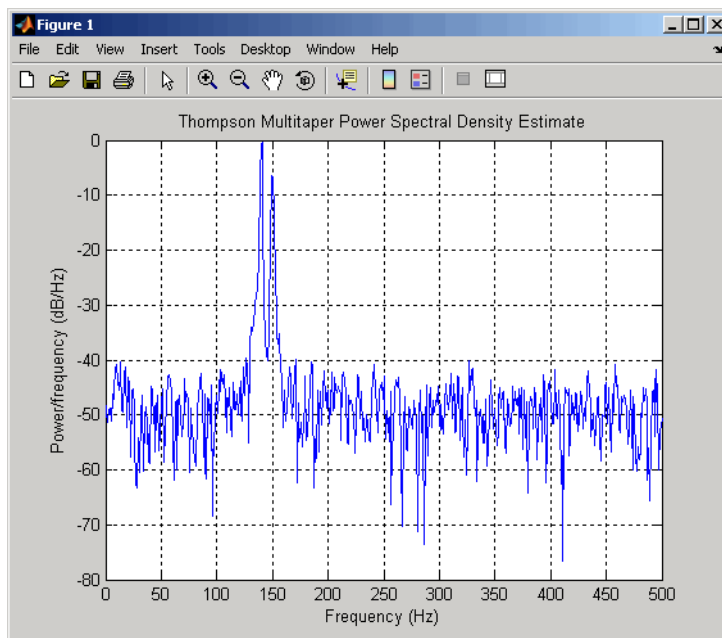

Note that the average power is conserved in both cases:

```
Hs1p = psd(Hs1,xn,'Fs',fs,'NFFT',1024);
Pow1 = avgpower(Hs1p)
Hs2p = psd(Hs2,xn,'Fs',fs,'NFFT',1024);
Pow2 = avgpower(Hs2p)
```

This method is more computationally expensive than Welch's method due to the cost of computing the discrete prolate spheroidal sequences. For long data series (10,000 points or more), it is useful to compute the DPSSs once and save them in a MAT-file. dpsssave, dpssload, dpssdir, and dpssclear are provided to keep a database of saved DPSSs in the MAT-file dpss.mat.

Cross-Spectral Density Function

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

$$P_{xy}(\omega) = \frac{1}{2\pi} \sum_{m=-\infty}^{\infty} R_{xy}(m)e^{-j\omega m}$$

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

To estimate the cross-spectral density of two equal length signals x and y using Welch's method, the cpsd function forms the periodogram as the product of the FFT of x and the conjugate of the FFT of y. Unlike the real-valued PSD, the CPSD is a complex function. cpsd handles the sectioning and windowing of x and y in the same way as the pwelch function:

```
Sxy = cpsd(x,y,nwin,noverlap,nfft,fs)
```

Transfer Function Estimate

One application of Welch's method is nonparametric system identification. Assume that *H* is a linear, time invariant system, and *x*(*n*) and *y*(*n*) are the input to and output of *H*, respectively. Then the power spectrum of *x*(*n*) is related to the CPSD of *x*(*n*) and *y*(*n*) by

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

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

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

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

Filter the signal xn with an FIR filter, then plot the actual magnitude response and the estimated response:

```
h = ones(1,10)/10;                % Moving-average filter
yn = filter(h,1,xn);
[HEST,f] = tfestimate(xn,yn,256,128,256,fs);
H = freqz(h,1,f,fs);
subplot(2,1,1); plot(f,abs(H));
title('Actual Transfer Function Magnitude');
subplot(2,1,2); plot(f,abs(HEST));
title('Transfer Function Magnitude Estimate');
xlabel('Frequency (Hz)');
```
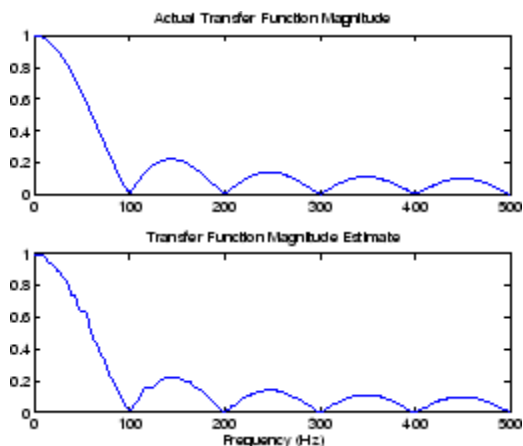

Coherence Function

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

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

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

The `mscohere` function takes sequences `x` and `y`, computes their power spectra and CPSD, and returns the quotient of the magnitude squared of the CPSD and the product of the power spectra. Its options and operation are similar to the `cpsd` and `tfestimate` functions.

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

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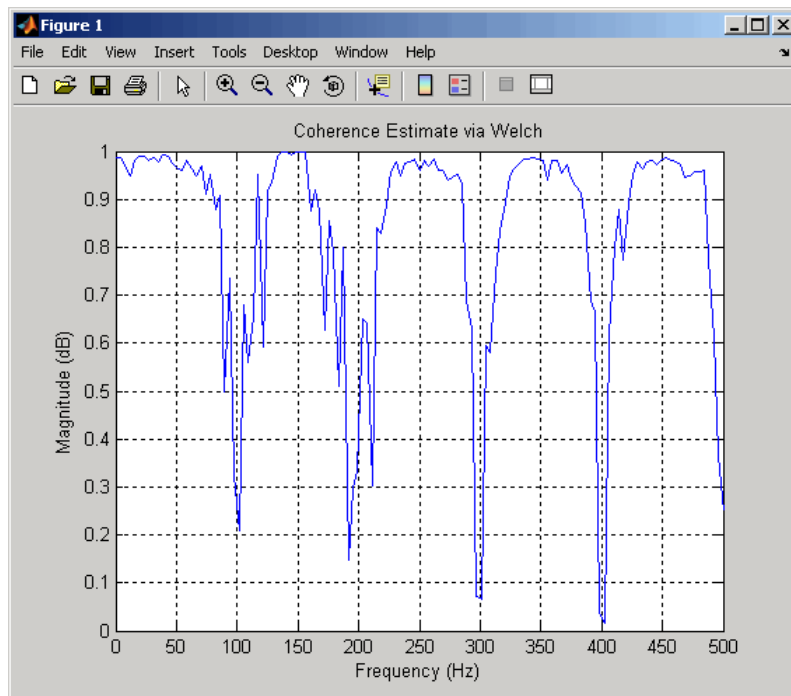

If the input sequence length `nfft`, window length `window`, and the number of overlapping data points in a window `numoverlap,` are such that `mscohere` operates on only a single record, the function returns all ones. This is because the coherence function for linearly dependent data is one.

Parametric Methods

Parametric methods can yield higher resolutions than nonparametric methods in cases when the signal length is short. These methods use a different approach to spectral estimation; instead of trying to estimate the PSD directly from the data, they *model* the data as the output of a linear system driven by white noise, and then attempt to estimate the parameters of that linear system.

The most commonly used linear system model is the *all-pole model*, a filter with all of its zeroes at the origin in the *z*-plane. The output of such a filter for white noise input is an autoregressive (AR) process. For this reason, these methods are sometimes referred to as *AR methods* of spectral estimation.

The AR methods tend to adequately describe spectra of data that is "peaky," that is, data whose PSD is large at certain frequencies. The data in many practical applications (such as speech) tends to have "peaky spectra" so that AR models are often useful. In addition, the AR models lead to a system of linear equations which is relatively simple to solve.

Signal Processing Toolbox AR methods for spectral estimation include:

- Yule-Walker AR method (autocorrelation method)

- Burg method

- Covariance method

- Modified covariance method

All AR methods yield a PSD estimate given by

$$\hat{P}(f) = \frac{1}{Fs} \frac{p}{|1 - \sum_{k=1}^{p} \hat{a}_p(k)e^{-j2\pi kf/Fs}|^2}$$

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

AR Methods

	Burg	**Covariance**	**Modified Covariance**	**Yule-Walker**
Characteristics	Does not apply window to data	Does not apply window to data	Does not apply window to data	Applies window to data
	Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion	Minimizes the forward prediction error in the least squares sense	Minimizes the forward and backward prediction errors in the least squares sense	Minimizes the forward prediction error in the least squares sense (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	

AR Methods (Continued)

	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 solving the following linear system, which give the Yule-Walker equations in matrix form:

$$\begin{pmatrix} r(0) & r^*(1) & \ldots & r^*(p-1) \\ r(1) & r(0) & \ldots & r^*(p-2) \\ r(p-1) & r(p-2) & \ldots & r(0) \end{pmatrix} \begin{pmatrix} a(1) \\ a(2) \\ \ldots \\ a(p) \end{pmatrix} = \begin{pmatrix} r(1) \\ r(2) \\ \ldots \\ r(p) \end{pmatrix}$$

In practice, the biased estimate of the autocorrelation is used for the unknown true autocorrelation. The Yule-Walker AR method produces the same results as a maximum entropy estimator. For more information, see page 155 of item [2] in the "Selected Bibliography" on page 6-48.

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 Hermitian Toeplitz structure of the autocorrelation matrix.

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

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

```
load mtlb
Hwelch = spectrum.welch('hamming',256,50);
psd(Hwelch,mtlb,'Fs',Fs,'NFFT',1024)
```
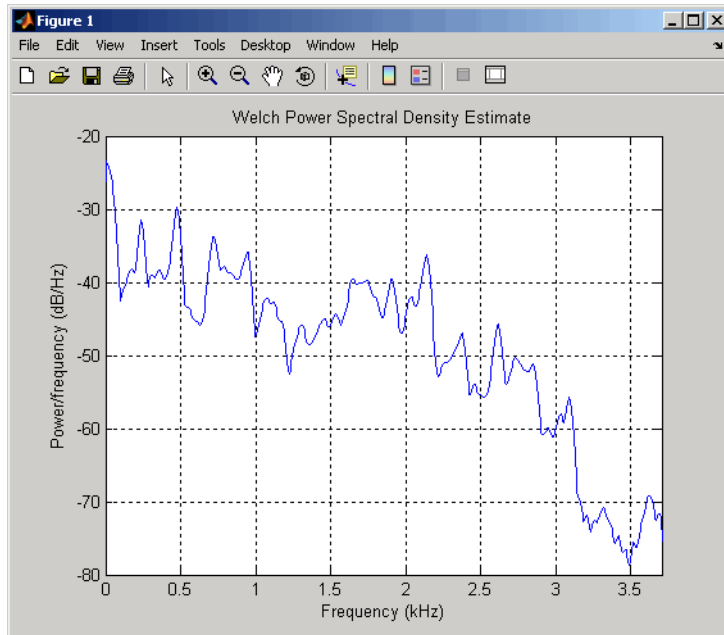
```
Hyulear = spectrum.yulear(14);
psd(Hyulear,mtlb,'Fs',Fs,'NFFT',1024)
```
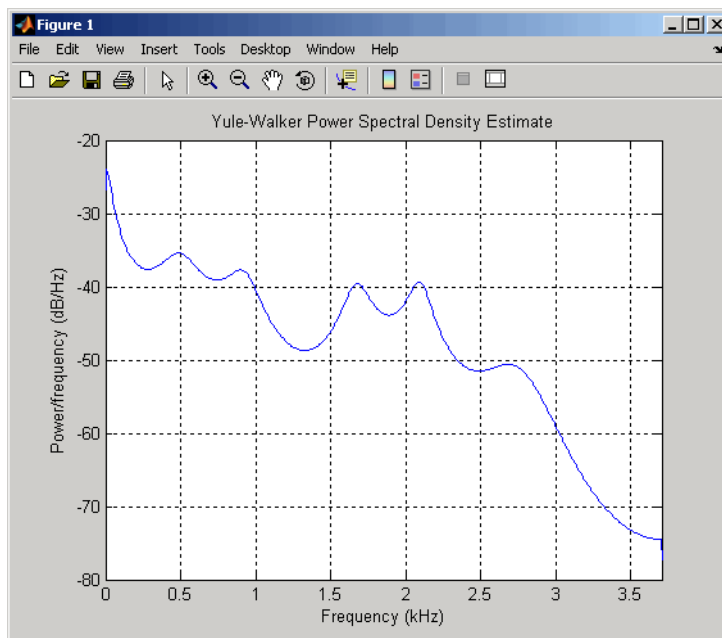

The Yule-Walker AR spectrum is smoother than the periodogram because of the simple underlying all-pole model.

Burg Method

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

The primary advantages of the Burg method are resolving closely spaced sinusoids in signals with low noise levels, and estimating short data records, in which case the AR power spectral density estimates are very close to the

true values. In addition, the Burg method ensures a stable AR model and is computationally efficient.

The accuracy of the Burg method is lower for high-order models, long data records, and high signal-to-noise ratios (which can cause *line splitting*, or the generation of extraneous peaks in the spectrum estimate). The spectral density estimate computed by the Burg method is also susceptible to frequency shifts (relative to the true frequency) resulting from the initial phase of noisy sinusoidal signals. This effect is magnified when analyzing short data sequences.

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

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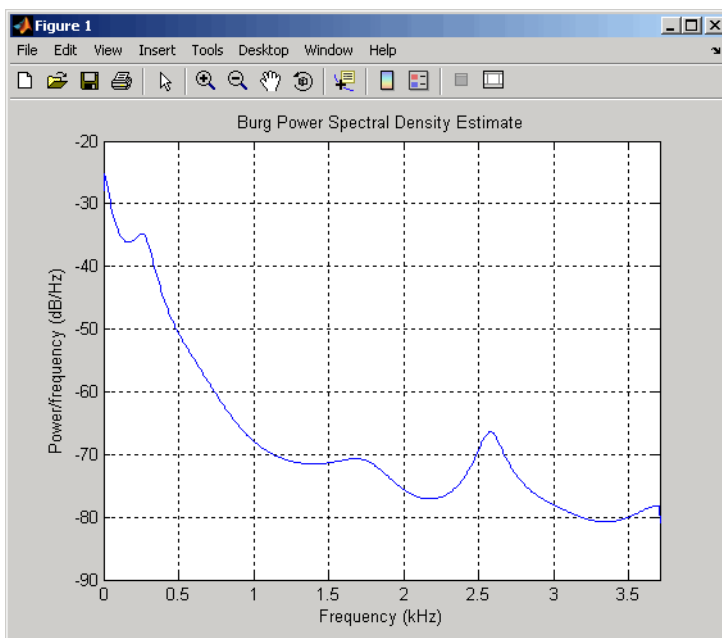
```
Hyulear = spectrum.yulear(14);        % 14th order model
psd(Hyulear,mtlb(1:512),'Fs',Fs,'NFFT',1024)
```
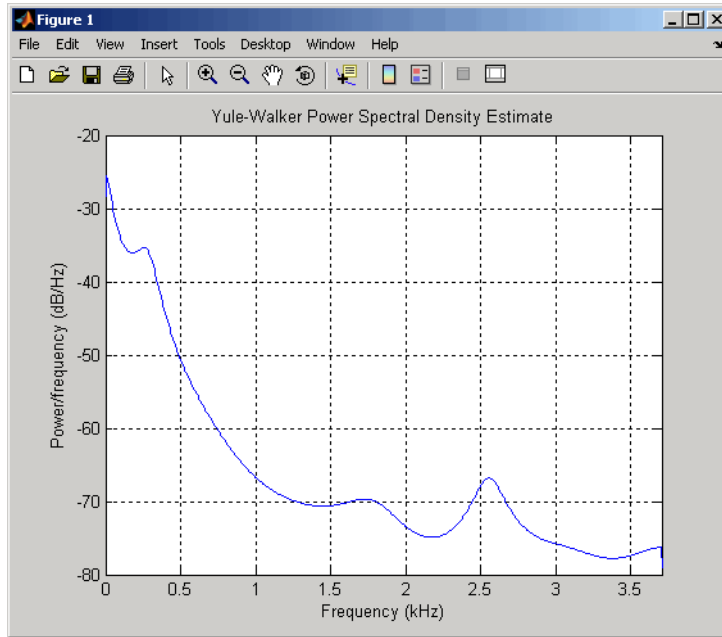

Compare the spectrum of a noisy signal computed using the Burg method and the Welch method:

```
fs = 1000;                 % Sampling frequency
t = (0:fs)/fs;             % One second worth of samples
A = [1 2];                 % Sinusoid amplitudes
f = [150;140];             % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
Hwelch = spectrum.welch('hamming',256,50);
psd(Hwelch,xn,'Fs',fs,'NFFT',1024)
```
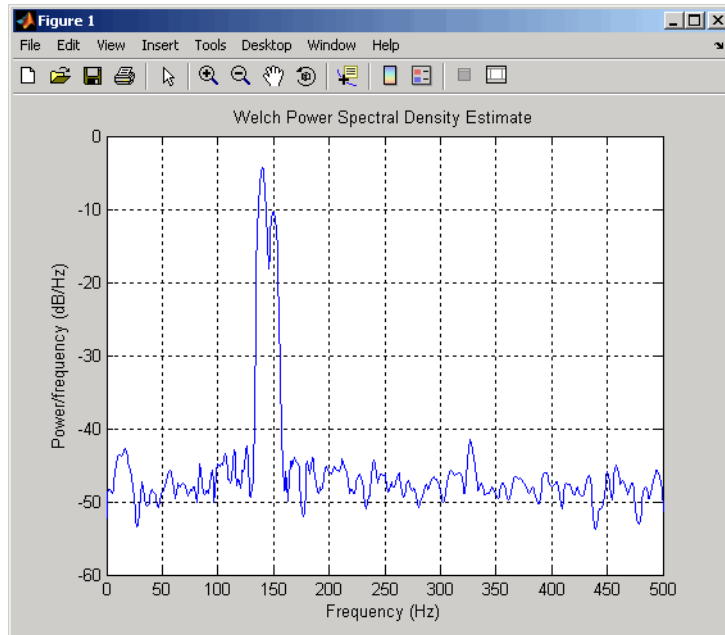
```
Hburg = spectrum.burg(14);
psd(Hburg,xn,'Fs',fs,'NFFT',1024)
```
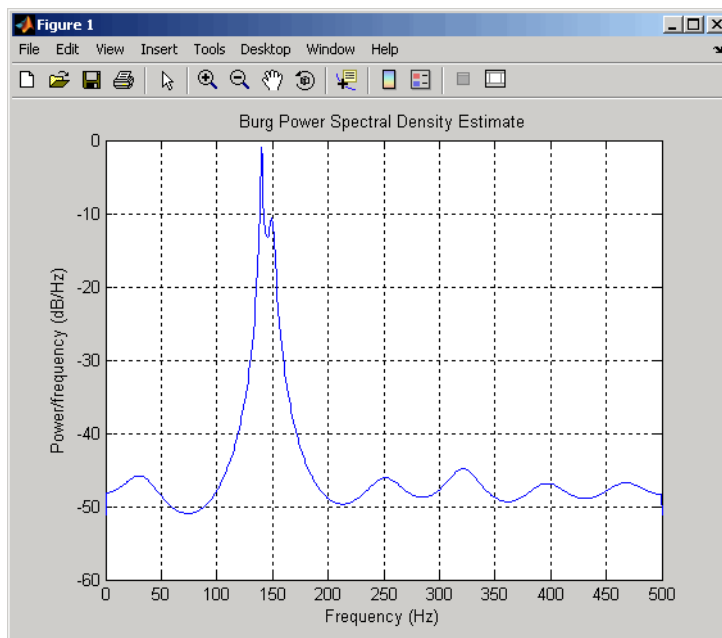

Note that, as the model order for the Burg method is reduced, a frequency shift due to the initial phase of the sinusoids will become apparent.

Covariance and Modified Covariance Methods

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

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

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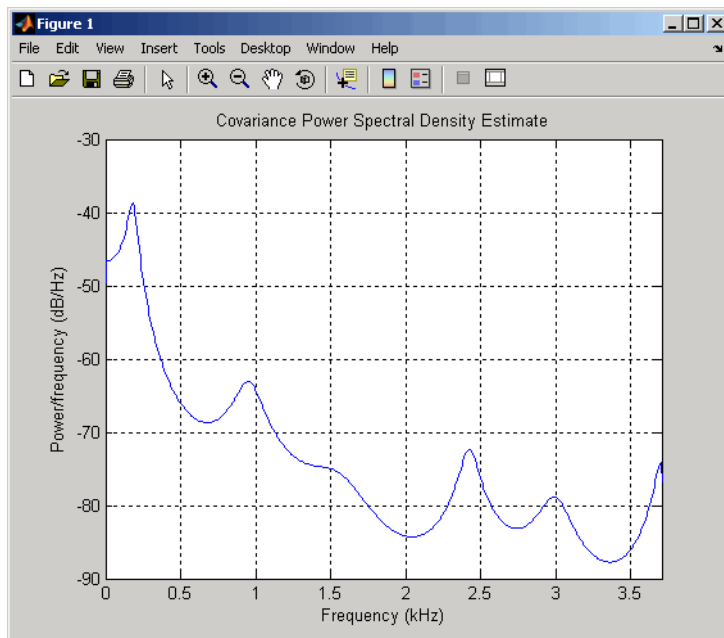


```
Hmcov = spectrum.mcov(14);                  % 14th order model
psd(Hmcov,mtlb(1:64),'Fs',Fs,'NFFT',1024)
```
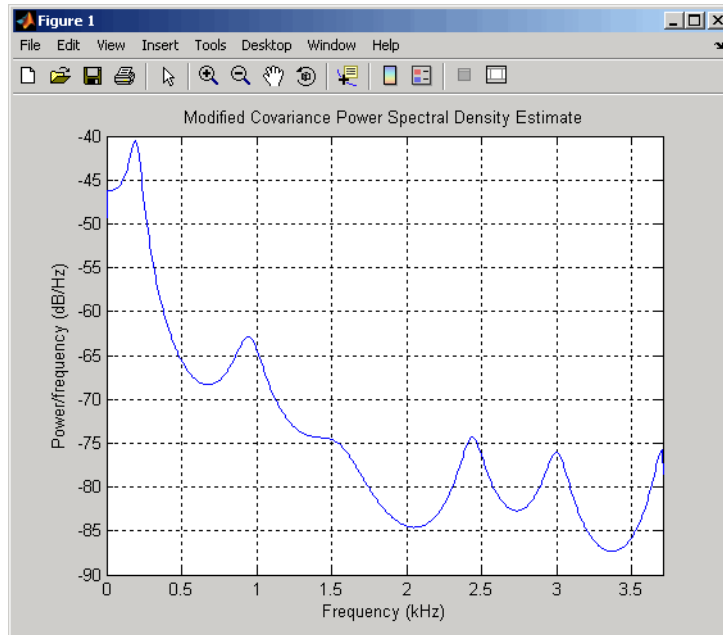
MUSIC and Eigenvector Analysis Methods

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

- `spectrum.music` and `pmusic` provide the multiple signal classification (MUSIC) method developed by Schmidt

- `spectrum.eigenvector` and `peig` provides the eigenvector (EV) method developed by Johnson

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

Both of these methods are frequency estimator techniques based on eigenanalysis of the autocorrelation matrix. This type of spectral analysis categorizes the information in a correlation or data matrix, assigning information to either a signal subspace or a noise subspace.

Eigenanalysis Overview

Consider a number of complex sinusoids embedded in white noise. You can write the autocorrelation matrix R for this system as the sum of the signal autocorrelation matrix (S) and the noise autocorrelation matrix (W):

$R = S + W$. There is a close relationship between the eigenvectors of the signal autocorrelation matrix and the signal and noise subspaces. The eigenvectors v of S span the same signal subspace as the signal vectors. If the system contains M complex sinusoids and the order of the autocorrelation matrix is p, eigenvectors v_{M+1} through v_{p+1} span the noise subspace of the autocorrelation matrix.

Frequency Estimator Functions. To generate their frequency estimates, eigenanalysis methods calculate functions of the vectors in the signal and noise subspaces. Both the MUSIC and EV techniques choose a function that goes to infinity (denominator goes to zero) at one of the sinusoidal frequencies in the input signal. Using digital technology, the resulting estimate has sharp peaks at the frequencies of interest; this means that there might not be infinity values in the vectors.

The MUSIC estimate is given by the formula

$$\hat{P}_{\mathrm{MUSIC}}(f) = \frac{1}{\displaystyle\sum_{k=p+1}^{M} |e(f)^H v_k|^2}$$

where the v_k are the eigenvectors of the noise subspace and $e(f)$ is a vector of complex sinusoids.

$$e(f) = [1 \quad e^{j2\pi f} \quad e^{j4\pi f} \ldots e^{j2(M-1)\pi}]^T$$

v represents the eigenvectors of the input signal's correlation matrix; v_k is the kth 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 $e(f)^H v_k$ is equivalent to a Fourier transform (the vector $e(f)$ consists of complex exponentials). This form is useful for numeric

computation because the FFT can be computed for each v_k and then the squared magnitudes can be summed.

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

$$\hat{P}_{\mathrm{EV}}(f) = \frac{\lambda_k}{\displaystyle\sum_{k=p+1}^{M} |e^H(f)v_k|^2}$$

The `pmusic` and `peig` functions in this toolbox interpret their first input either as a signal matrix or as a correlation matrix (if the `'corr'` input flag is set). In the former case, the singular value decomposition of the signal matrix is used to determine the signal and noise subspaces. In the latter case, the eigenvalue decomposition of the correlation matrix is used to determine the signal and noise subspaces.

Using FFT to Obtain Simple Spectral Analysis Plots

In general, you use `spectrum` objects to perform spectral analysis, but if you want to create spectral analysis plots manually starting with an FFT, follow this example. Assume you have a 200 Hz sinusoid signal and a sampling frequency of 1024.

```
Fs = 1024;             % Sampling frequency
t = 0:1/Fs:1;          % Time vector
x = sin(2*pi*t*200);   % 200 Hz sinusoid signal
```

Now, you take the FFT of the signal. To optimize the FFT, you pad the signal with enough zeros so that its length is a power of 2. The `fft` automatically does this if you provide an input argument specifying the length of the FFT.

```
nfft = 2^(nextpow2(length(x)));   % Find next power of 2
fftx = fft(x,nfft);
```

`nfft` is even, so the magnitude of the FFT will be symmetric, in that the first (1+nfft/2) points are unique, and the rest are symmetrically redundant. The DC component of `x` is `fftx(1)` and the Nyquist frequency is at `fftx(1+nfft/2)`. Note that if `nfft` is odd, the Nyquist frequency component is not evaluated, and the number of unique points is (nfft+1)/2 . You can use

the `ceil` function to determine the number of unique points. In this example, you know the FFT is symmetric, so you keep only the unique points. Then, calculate the magnitude of the FFT.

```
NumUniquePts = ceil((nfft+1)/2);
fftx = fftx(1:NumUniquePts);
mx = abs(fftx);
```

MATLAB does not scale the output of the FFT by the length of the input, so you scale the it so that it is not a function of the length of x. Then, square the magnitude and since you dropped half of the FFT, multiply by 2 to retain the same amount of energy, but do not multiply the DC or Nyquist components by 2. You then obtain an evenly spaced frequency vector with `NumUniquePts` points.

```
mx = mx/length(x);
mx = mx.^2;
if rem(nfft, 2)   % Odd nfft excludes Nyquist
  mx(2:end) = mx(2:end)*2;
else
  mx(2:end -1) = mx(2:end -1)*2;
end

f = (0:NumUniquePts-1)*Fs/nfft;
```

Finally, plot the resulting spectrum.

```
plot(f,mx);
title('Power Spectrum of a 200Hz Sine Wave');
xlabel('Frequency (Hz)');
ylabel('Power');
```
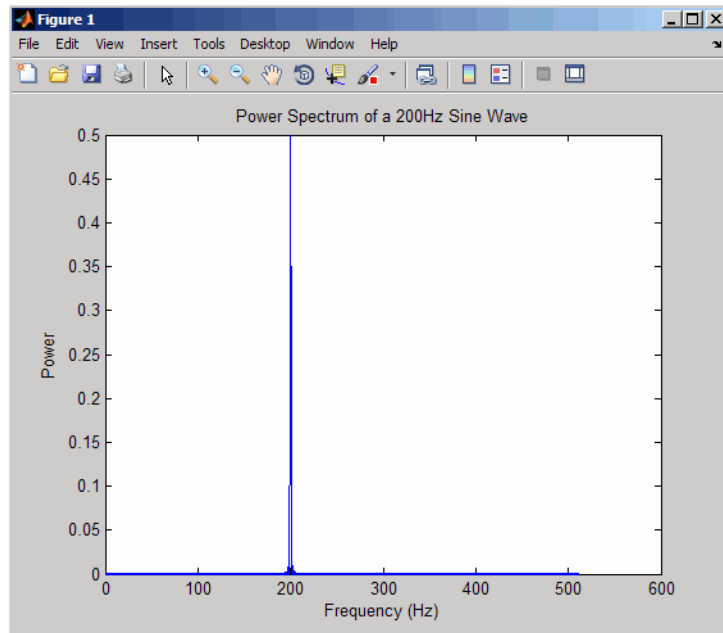
Selected Bibliography

[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.

7

Special Topics

Windows

In this section...

Why Use Windows?

In both digital filter design and spectral estimation, the choice of a windowing function can play an important role in determining the quality of overall results. The main role of the window is to damp out the effects of the Gibbs phenomenon that results from truncation of an infinite series.

Available Window Functions

Window	**Function**
Bartlett-Hann window	`barthannwin`
Bartlett window	`bartlett`
Blackman window	`blackman`
Blackman-Harris window	`blackmanharris`
Bohman window	`bohmanwin`
Chebyshev window	`chebwin`
Flat Top window	`flattopwin`
Gaussian window	`gausswin`
Hamming window	`hamming`

Window	Function
Hann window	`hann`
Kaiser window	`kaiser`
Nuttall's Blackman-Harris window	`nuttallwin`
Parzen (de la Valle-Poussin) window	`parzenwin`
Rectangular window	`rectwin`
Tapered cosine window	`tukeywin`
Triangular window	`triang`

Graphical User Interface Tools

Two graphical user interface tools are provided for working with windows in the Signal Processing Toolbox product:

- Window Design and Analysis Tool (`wintool`)

- Window Visualization Tool (`wvtool`)

Refer to the reference pages for these tools for detailed information.

Basic Shapes

The basic window is the *rectangular window*, a vector of ones of the appropriate length. A rectangular window of length 50 is

```
n = 50;
w = rectwin(n);
```

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

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

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

```
wintool
```

`wintool` opens with a default Hamming window. In the Current Window Information panel, set **Type = Rectangular** and **Length = 50** and then press **Apply**.
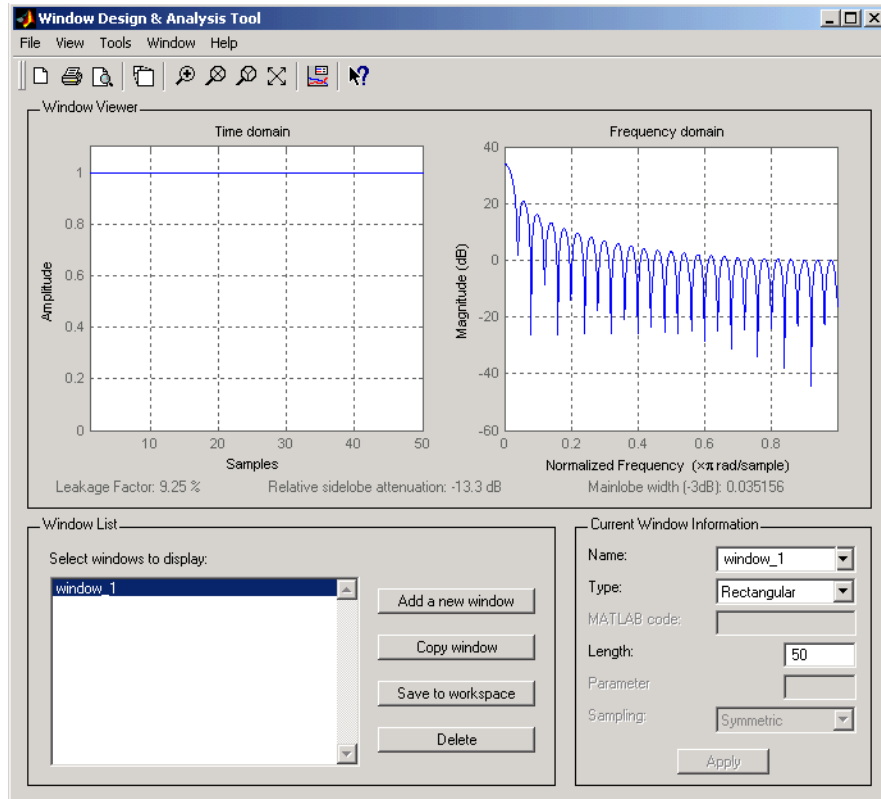
The *Bartlett* (or triangular) *window* is the convolution of two rectangular windows. The functions `bartlett` and `triang` compute similar triangular windows, with three important differences. The `bartlett` function always returns a window with two zeros on the ends of the sequence, so that for `n` odd, the center section of `bartlett(n+2)` is equivalent to `triang(n)`:

```
Bartlett = bartlett(7);
isequal(Bartlett(2:end-1),triang(5))
% Returns a 1
```

For n even, `bartlett` is still the convolution of two rectangular sequences. There is no standard definition for the triangular window for n even; the slopes of the line segments of the `triang` result are slightly steeper than those of `bartlett` in this case:

```
w = bartlett(8);
[w(2:7)  triang(6)]
```

You can see the difference between odd and even Bartlett windows in WinTool.
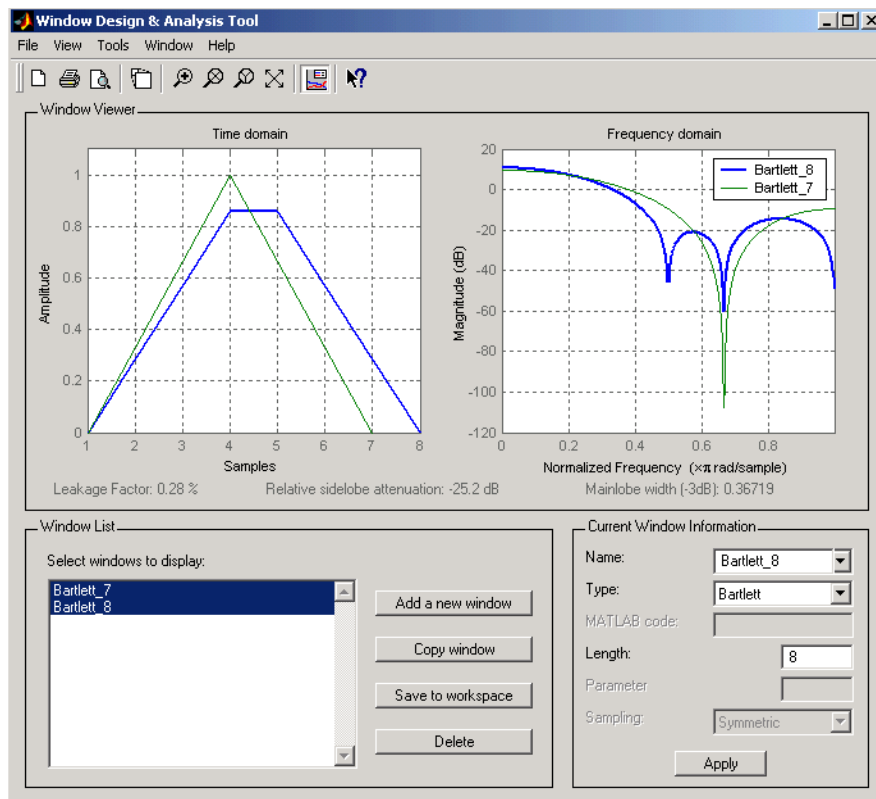
The final difference between the Bartlett and triangular windows is evident in the Fourier transforms of these functions. The Fourier transform of a Bartlett window is negative for n even. The Fourier transform of a triangular window, however, is always nonnegative.

The following figure, which is a zoomed version of the Frequency domain plot of 8-point Bartlett and Triangular windows with the *y*-axis set to Zerophase, illustrates this difference.
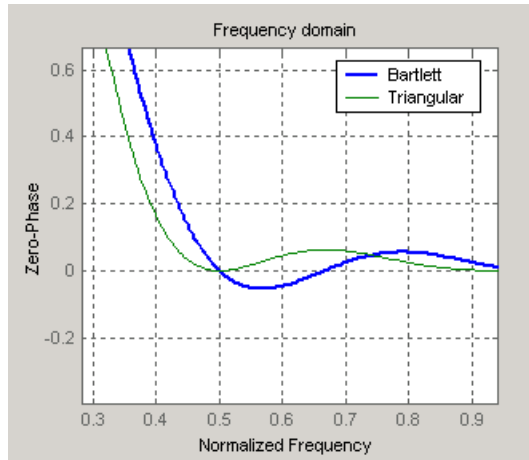
This difference can be important when choosing a window for some spectral estimation techniques, such as the Blackman-Tukey method. Blackman-Tukey forms the spectral estimate by calculating the Fourier transform of the autocorrelation sequence. The resulting estimate might be negative at some frequencies if the window's Fourier transform is negative (see Kay [1], pg. 80).

Generalized Cosine Windows

Blackman, flat top, Hamming, Hann, and rectangular windows are all special cases of the *generalized cosine window*. These windows are combinations of sinusoidal sequences with frequencies that are multiples of $2\pi/(N-1)$, where *N* is the window length. One special case is the Blackman window:

```
N = 128;
A = 0.42; B = 0.5; C = 0.08;
ind = (0:N-1)'*2*pi/(N-1);
w = A - B*cos(ind) + C*cos(2*ind);
```

Changing the values of the constants A, B, and C in the previous expression generates different generalized cosine windows like the Hamming and Hann

windows. Adding additional cosine terms of higher frequency generates the flat top window. The concept behind these windows is that by summing the individual terms to form the window, the low frequency peaks in the frequency domain combine in such a way as to decrease sidelobe height. This has the side effect of increasing the mainlobe width.

The Hamming and Hann windows are two-term generalized cosine windows, given by A = 0.54, B = 0.46 for the Hamming and A = 0.5, B = 0.5 for the Hann.

Note that the definition of the generalized cosine window shown in the earlier MATLAB code yields zeros at samples 1 and n for A = 0.5 and B = 0.5.

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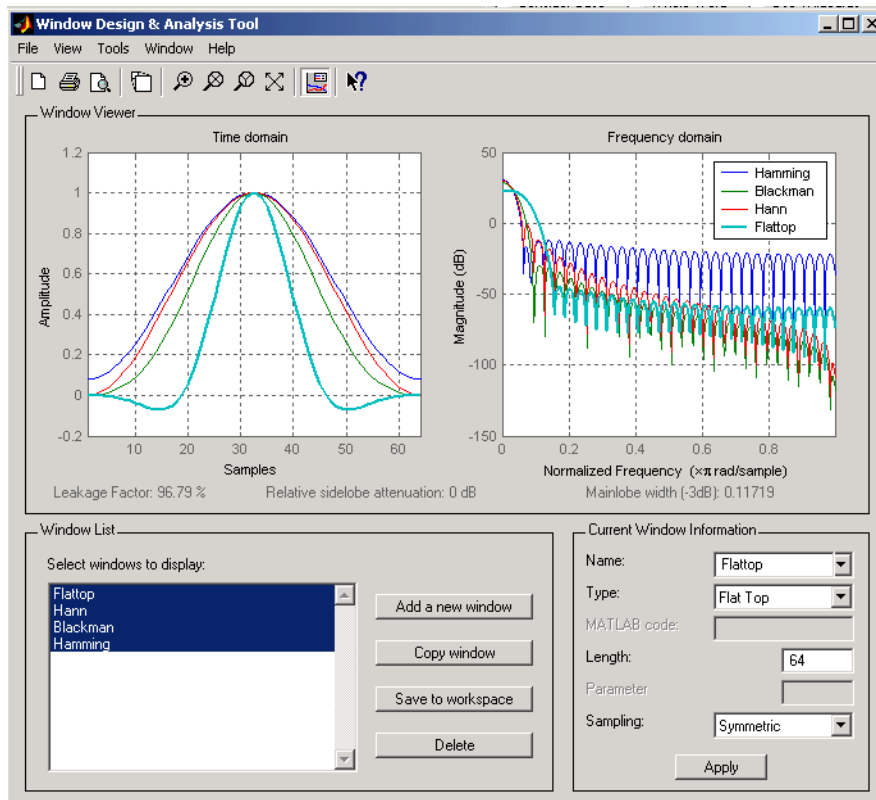
Kaiser Window

The *Kaiser window* is an approximation to the prolate-spheroidal window, for which the ratio of the mainlobe energy to the sidelobe energy is maximized. For a Kaiser window of a particular length, the parameter 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 `beta` parameters of 1, 4, and 9 are shown in this `wintool` example.
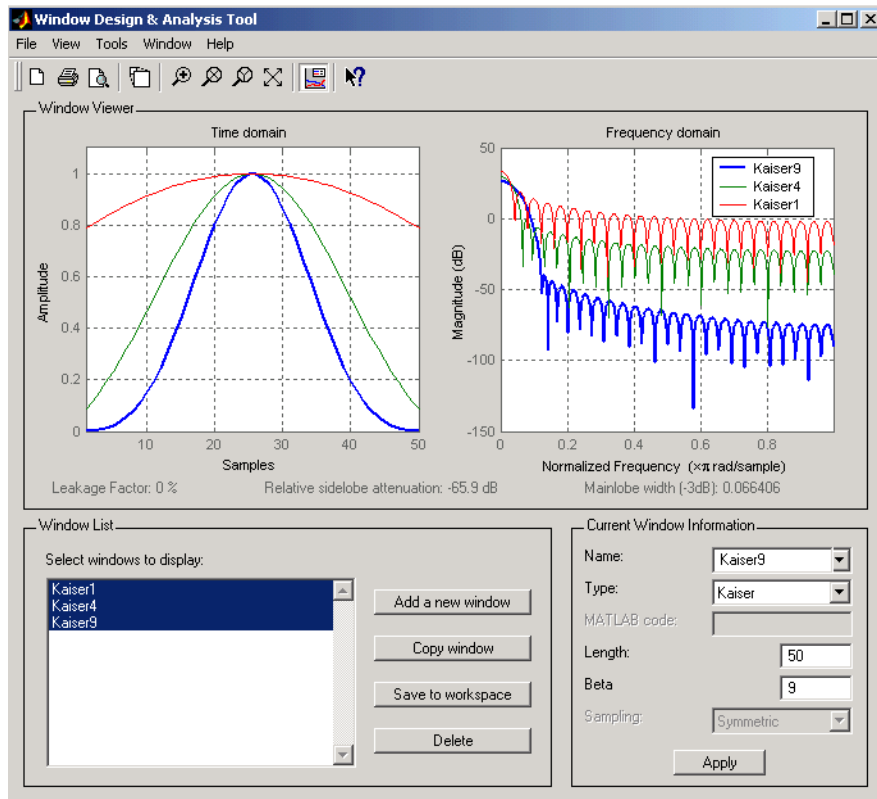
To create these Kaiser windows using the MATLAB command line,

```
n = 50;
```

```
w1 = kaiser(n,1);
w2 = kaiser(n,4);
w3 = kaiser(n,9);
[W1,f] = freqz(w1/sum(w1),1,512,2);
[W2,f] = freqz(w2/sum(w2),1,512,2);
[W3,f] = freqz(w3/sum(w3),1,512,2);
plot(f,20*log10(abs([W1 W2 W3]))); grid;
legend('beta = 1','beta = 4','beta = 9',3)
```

As β increases, the sidelobe height decreases and the mainlobe width increases. This wintool shows how the sidelobe height stays the same for a fixed β parameter as the length is varied.
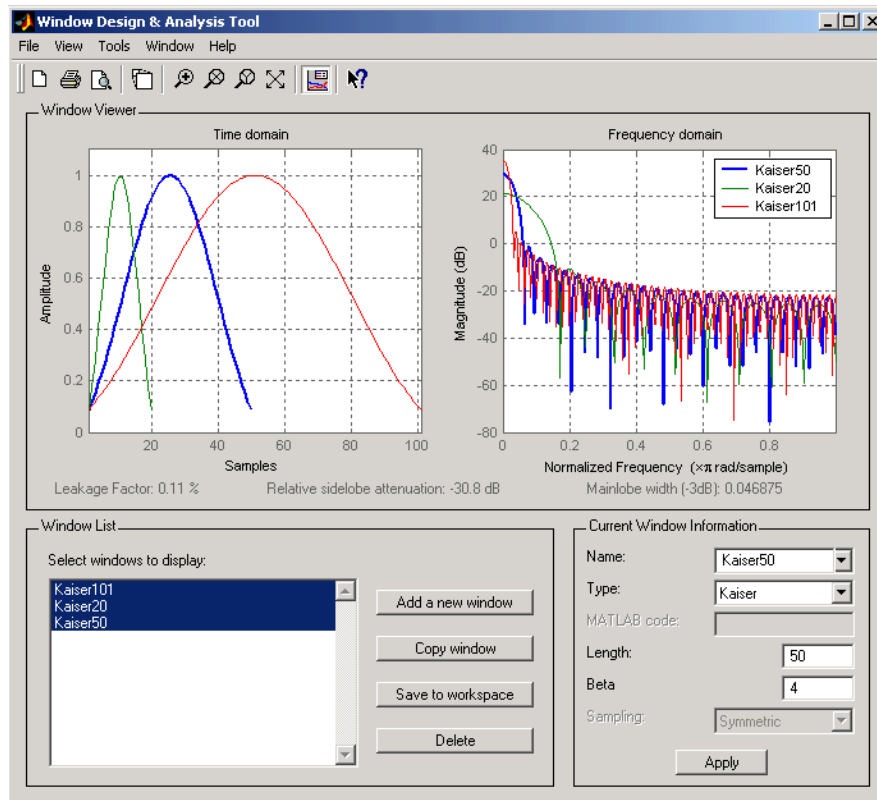
To create these Kaiser windows using the MATLAB command line:

```
w1 = kaiser(50,4);
w2 = kaiser(20,4);
w3 = kaiser(101,4);
[W1,f] = freqz(w1/sum(w1),1,512,2);
[W2,f] = freqz(w2/sum(w2),1,512,2);
[W3,f] = freqz(w3/sum(w3),1,512,2);
plot(f,20*log10(abs([W1 W2 W3]))); grid;
legend('length = 50','length = 20','length = 101')
```

Kaiser Windows in FIR Design

There are two design formulas that can help you design FIR filters to meet a set of filter specifications using a Kaiser window. To achieve a sidelobe height of -α dB, the `beta` parameter is

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

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

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

Filters designed using these heuristics will meet the specifications approximately, but you should verify this. To design a lowpass filter with cutoff frequency 0.5π 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:

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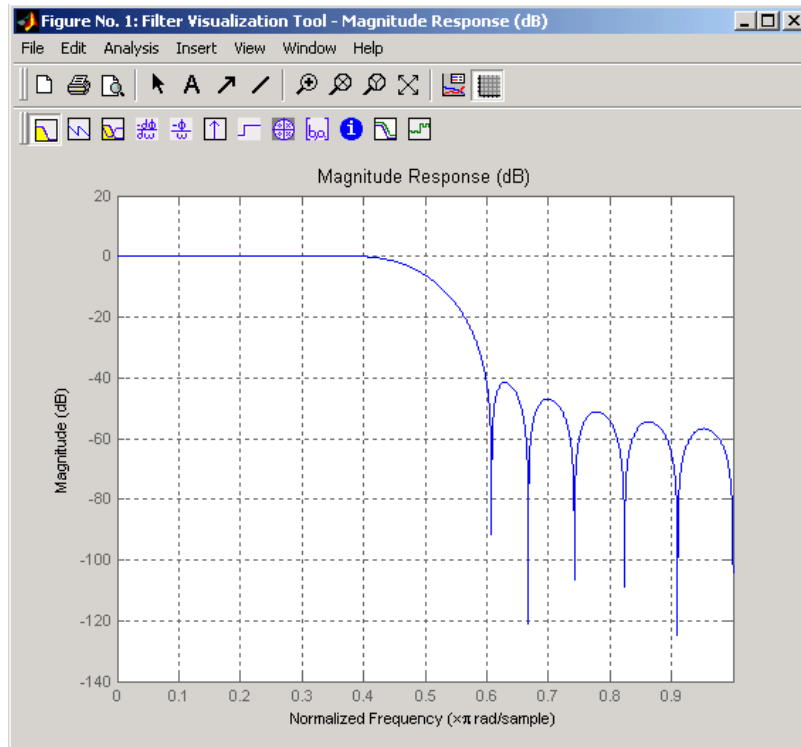
Chebyshev Window

The Chebyshev window minimizes the mainlobe width, given a particular sidelobe height. It is characterized by an equiripple behavior, that is, its sidelobes all have the same height.
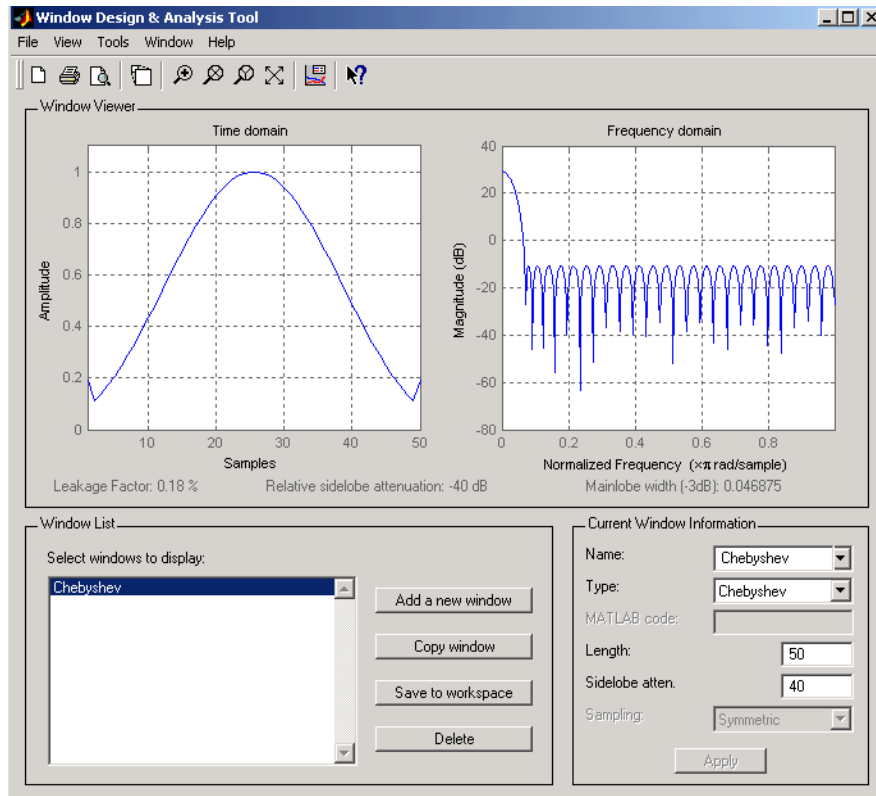
As shown in the Time Domain plot, the Chebyshev window has large spikes at its outer samples.

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

Parametric Modeling

In this section...
"What is Parametric Modeling" on page 7-13
"Available Parametric Modeling Functions" on page 7-13
"Time-Domain Based Modeling" on page 7-14
"Frequency-Domain Based Modeling" on page 7-19

What is Parametric Modeling

Parametric modeling techniques find the parameters for a mathematical model describing a signal, system, or process. These techniques use known information about the system to determine the model. Applications for parametric modeling include speech and music synthesis, data compression, high-resolution spectral estimation, communications, manufacturing, and simulation.

Note Since `yulewalk` is geared explicitly toward ARMA filter design, it is discussed in Chapter 2, "Filter Design and Implementation".

`pburg` and `pyulear` are discussed in Chapter 6, "Statistical Signal Processing", with the other (nonparametric) spectral estimation methods.

Available Parametric Modeling Functions

The toolbox parametric modeling functions operate with the rational transfer function model. Given appropriate information about an unknown system (impulse or frequency response data, or input and output sequences), these functions find the coefficients of a linear system that models the system.

One important application of the parametric modeling functions is in the design of filters that have a prescribed time or frequency response. These functions provide a data-oriented alternative to the IIR and FIR filter design functions discussed in Chapter 2, "Filter Design and Implementation".

Here is a summary of the parametric modeling functions in this toolbox. Note that "System Identification Toolbox" software 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.

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:

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

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 [4] Parks and Burrus (pgs. 226-228). This method uses a variation of the covariance method of AR modeling to find the denominator coefficients a, and then finds the numerator coefficients b for which the resulting filter's impulse response matches exactly the first n + 1 samples of x. The filter is not necessarily stable, but it can potentially recover the coefficients exactly if the data sequence is truly an autoregressive moving-average (ARMA) process of the correct order.

Note The functions prony and stmcb (described next) are more accurately described as ARX models in system identification terminology. ARMA modeling assumes noise only at the inputs, while ARX assumes an external input. prony and stmcb know the input signal: it is an impulse for prony and is arbitrary for stmcb.

A model for the test sequence x (from the earlier lpc example) using a third-order IIR filter is

```
[b,a] = prony(x,3,3)
b =
    0.9567   -0.3351    0.1866   -0.3782
a =
    1.0000   -0.0716    0.2560   -0.2752
```

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

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

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

Notice that the first four samples match exactly. For an example of exact recovery, recover the coefficients of a Butterworth filter from its impulse response:

```
[b,a] = butter(4,.2);
h = impz(b,a,26);
[bb,aa] = prony(h,4,4);
```

Try this example; you'll see that `bb` and `aa` match the original filter coefficients to within a tolerance of 10^{-13}.

Steiglitz-McBride Method (ARMA Modeling)

The `stmcb` function determines the coefficients for the system $b(z)/a(z)$ given an approximate impulse response x, as well as the desired number of zeros and poles. This function identifies an unknown system based on both input and output sequences that describe the system's behavior, or just the impulse response of the system. In its default mode, `stmcb` works like `prony`.

```
[b,a] = stmcb(x,3,3)
b =
    0.9567   -0.5181    0.5702   -0.5471
```

```
a =
     1.0000   -0.2384    0.5234    -0.3065
```

stmcb also finds systems that match given input and output sequences:

```
y = filter(1,[1 1],x);       % Create an output signal.
[b,a] = stmcb(y,x,0,1)
b =
     1.0000
a =
     1      1
```

In this example, stmcb correctly identifies the system used to create y from x.

The Steiglitz-McBride method is a fast iterative algorithm that solves for the numerator and denominator coefficients simultaneously in an attempt to minimize the signal error between the filter output and the given output signal. This algorithm usually converges rapidly, but might not converge if the model order is too large. As for prony, stmcb's resulting filter is not necessarily stable due to its exact modeling approach.

stmcb provides control over several important algorithmic parameters; modify these parameters if you are having trouble modeling the data. To change the number of iterations from the default of five and provide an initial estimate for the denominator coefficients:

```
n = 10;              % Number of iterations
a = lpc(x,3);        % Initial estimates for denominator
[b,a] = stmcb(x,3,3,n,a);
```

The function uses an all-pole model created with prony as an initial estimate when you do not provide one of your own.

To compare the functions lpc, prony, and stmcb, compute the signal error in each case:

```
a1 = lpc(x,3);
[b2,a2] = prony(x,3,3);
[b3,a3] = stmcb(x,3,3);
[x-impz(1,a1,10)  x-impz(b2,a2,10)  x-impz(b3,a3,10)]
```

```
ans =
   -0.0433        0    -0.0000
   -0.0240        0     0.0234
   -0.0040        0    -0.0778
   -0.0448   -0.0000    0.0498
   -0.2130   -0.1303   -0.0742
    0.1545    0.1655    0.1270
    0.1426    0.1404    0.1055
    0.0068    0.0188    0.0465
    0.0329    0.0376    0.0530
    0.0108    0.0081   -0.0162
sum(ans.^2)
ans =
    0.0953    0.0659    0.0471
```

In comparing modeling capabilities for a given order IIR model, the last result shows that for this example, `stmcb` performs best, followed by `prony`, then `lpc`. This relative performance is typical of the modeling functions.

Frequency-Domain Based Modeling

The `invfreqs` and `invfreqz` functions implement the inverse operations of `freqs` and `freqz`; they find an analog or digital transfer function of a specified order that matches a given complex frequency response. Though the following examples demonstrate `invfreqz`, the discussion also applies to `invfreqs`.

To recover the original filter coefficients from the frequency response of a simple digital filter:

```
[b,a] = butter(4,0.4)           % Design Butterworth lowpass
b =
    0.0466    0.1863    0.2795    0.1863    0.0466
a =
    1.0000   -0.7821    0.6800   -0.1827    0.0301

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

```
a4 =
    1.0000   -0.7821    0.6800   -0.1827    0.0301
```

The vector of frequencies w has the units in rad/sample, and the frequencies need not be equally spaced. `invfreqz` finds a filter of any order to fit the frequency data; a third-order example is

```
[b4,a4] = invfreqz(h,w,3,3)   % Find third-order IIR
b4 =
    0.0464    0.1785    0.2446    0.1276
a4 =
    1.0000   -0.9502    0.7382   -0.2006
```

Both `invfreqs` and `invfreqz` design filters with real coefficients; for a data point at positive frequency `f`, the functions fit the frequency response at both `f` and `-f`.

By default `invfreqz` uses an equation error method to identify the best model from the data. This finds b and a in

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

by creating a system of linear equations and solving them with the MATLAB \ operator. Here $A(w(k))$ and $B(w(k))$ are the Fourier transforms of the polynomials a and b respectively at the frequency $w(k)$, and n is the number of frequency points (the length of h and w). $wt(k)$ weights the error relative to the error at different frequencies. The syntax

```
invfreqz(h,w,n,m,wt)
```

includes a weighting vector. In this mode, the filter resulting from `invfreqz` is not guaranteed to be stable.

`invfreqz` provides a superior ("output-error") algorithm that solves the direct problem of minimizing the weighted sum of the squared error between the actual frequency response points and the desired response

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

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

```
wt = ones(size(w));   % Create unity weighting vector
[b30,a30] = invfreqz(h,w,3,3,wt,30)  % 30 iterations
b30 =
    0.0464    0.1829    0.2572    0.1549
a30 =
    1.0000   -0.8664    0.6630   -0.1614
```

The resulting filter is always stable.

Graphically compare the results of the first and second algorithms to the original Butterworth filter with FVTool (and select the Magnitude and Phase Responses):

```
fvtool(b,a,b4,a4,b30,a30)
```
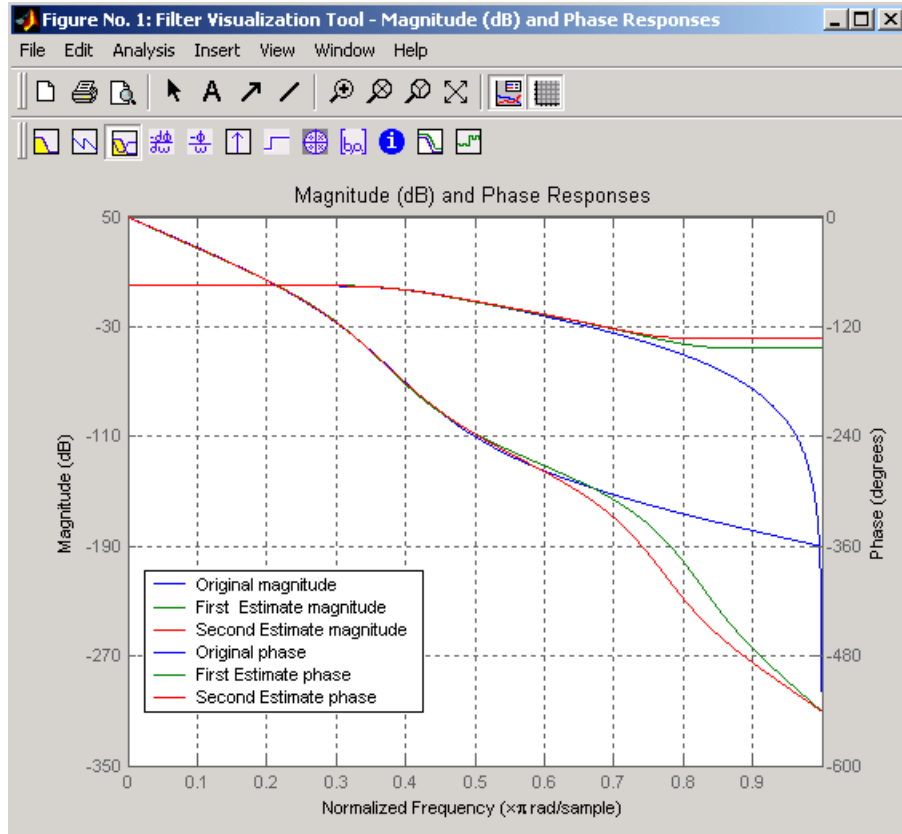
To verify the superiority of the fit numerically, type

```
sum(abs(h-freqz(b4,a4,w)).^2)    % Total error, algorithm 1
ans =
    0.0200
sum(abs(h-freqz(b30,a30,w)).^2)  % Total error, algorithm 2
ans =
    0.0096
```

Resampling

Available Resampling Functions

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

resample Function

The resample function changes the sampling rate for a sequence to any rate that is a ratio of two integers. The basic syntax for resample is

```
y = resample(x,p,q)
```

where the function resamples the sequence x at p/q times the original sampling rate. The length of the result y is p/q times the length of x.

One resampling application is the conversion of digitized audio signals from one sampling rate to another, such as from 48 kHz (the digital audio tape standard) to 44.1 kHz (the compact disc standard).

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

```
clear
load mtlb
whos

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

Grand total is 4002 elements using 32016 bytes
Fs
Fs =
        7418
```

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

```
[p,q] = rat(8192/Fs,0.0001)
p =
    127
q =
    115
```

Since p/q*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)
```

decimate and interp Functions

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.

upfirdn Function

The toolbox also contains a function, `upfirdn`, that applies an FIR filter to an input sequence and outputs the filtered sequence at a sample rate different than its original. See "Multirate Filter Bank Implementation" on page 1-8.

spline Function

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

In this section...
"What Is a Cepstrum?" on page 7-26
"Inverse Complex Cepstrum" on page 7-29

What Is a Cepstrum?

Cepstrum analysis is a nonlinear signal processing technique with a variety of applications in areas such as speech and image processing.

The complex cepstrum for a sequence x is calculated by finding the complex natural logarithm of the Fourier transform of x, then the inverse Fourier transform of the resulting sequence.

$$\hat{x} = \frac{1}{2\pi}\int_{-\pi}^{\pi} \log[X(e^{j\omega})]e^{j\omega n}d\omega$$

The toolbox function `cceps` performs this operation, estimating the complex cepstrum for an input sequence. It returns a real sequence the same size as the input sequence:

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

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

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

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

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

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

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

The complex cepstrum of this new signal is

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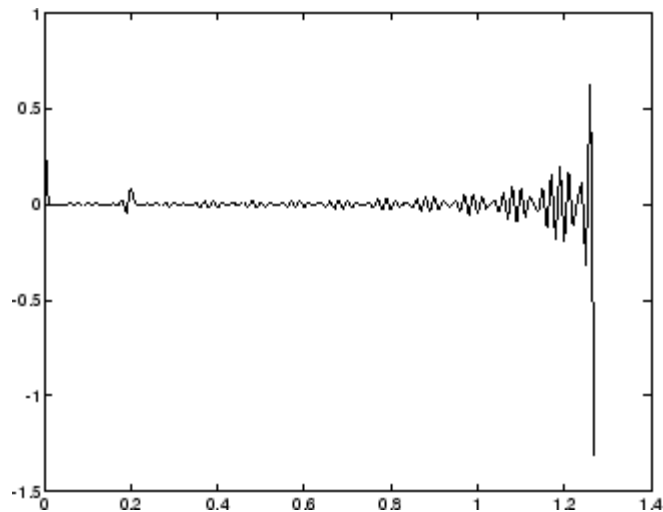

Note that the complex cepstrum shows a peak at 0.2 seconds, indicating the echo.

The *real cepstrum* of a signal *x*, sometimes called simply the cepstrum, is calculated by determining the natural logarithm of magnitude of the Fourier transform of *x*, then obtaining the inverse Fourier transform of the resulting sequence.

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

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

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

By definition, you cannot reconstruct the original sequence from its real cepstrum transformation, as the real cepstrum is based only on the magnitude of the Fourier transform for the sequence (see Oppenheim and Schafer [3]). The `rceps` function also returns a unique minimum-phase sequence that has the same real cepstrum as x. To obtain both the real cepstrum and the minimum phase reconstruction for a sequence, use

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

where y is the real cepstrum and ym is the minimum phase reconstruction of x. The following example shows that one output of rceps is a unique minimum-phase sequence with the same real cepstrum as x.

```
y = [4 1 5];                        % Non-minimum phase sequence
[xhat,yhat] = rceps(y);
xhat2= rceps(yhat);
[xhat' xhat2']

ans =
    1.6225     1.6225
    0.3400     0.3400
    0.3400     0.3400
```

Summary of Cepstrum Functions
The Signal Processing Toolbox product provides three functions for cepstrum analysis:

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

Inverse Complex Cepstrum

To invert the complex cepstrum, use the `icceps` function. Inversion is complicated by the fact that the `cceps` function performs a data dependent phase modification so that the unwrapped phase of its input is continuous at zero frequency. The phase modification is equivalent to an integer delay. This delay term is returned by `cceps` if you ask for a second output. For example:

```
x = 1:10;
[xhat,delay] = cceps(x)
xhat =
  Columns 1 through 4
    2.2428   -0.0420   -0.0210    0.0045
  Columns 5 through 8
    0.0366    0.0788    0.1386    0.2327
  Columns 9 through 10
    0.4114    0.9249
delay =
     1
```

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

```
icc = icceps(xhat,2)
ans =
  Columns 1 through 4
    2.0000    3.0000    4.0000    5.0000
  Columns 5 through 8
    6.0000    7.0000    8.0000    9.0000
  Columns 9 through 10
    10.0000   1.0000
```

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

FFT-Based Time-Frequency Analysis

The Signal Processing Toolbox product provides a function, `spectrogram`, that returns the time-dependent Fourier transform for a sequence, or displays this information as a spectrogram. The toolbox also includes a spectrogram demo. The *time-dependent Fourier transform* is the discrete-time Fourier transform for a sequence, computed using a sliding window. This form of the Fourier transform, also known as the short-time Fourier transform (STFT), has numerous applications in speech, sonar, and radar processing. The *spectrogram* of a sequence is the magnitude of the time-dependent Fourier transform versus time.

To display the spectrogram of a linear FM signal:

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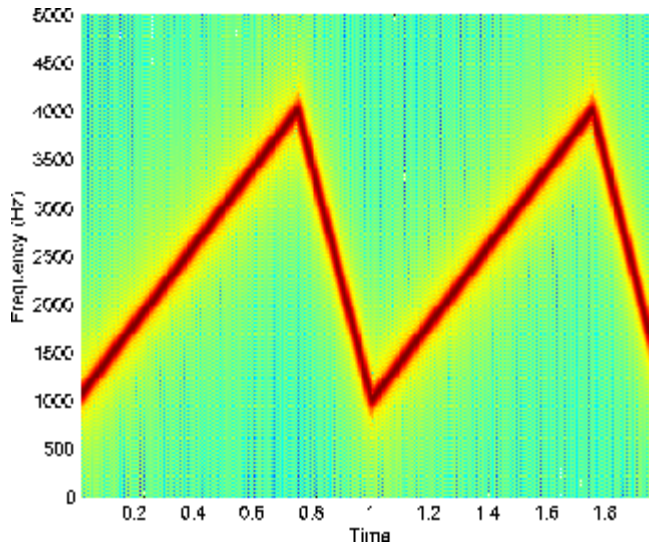
Median Filtering

The function `medfilt1` implements one-dimensional median filtering, a nonlinear technique that applies a sliding window to a sequence. The median filter replaces the center value in the window with the median value of all the points within the window [5]. In computing this median, `medfilt1` assumes zeros beyond the input points.

When the number of elements `n` in the window is even, `medfilt1` sorts the numbers, then takes the average of the `n/2` and `n/2 + 1` elements.

Two simple examples with fourth- and third-order median filters are

```
medfilt1([4 3 5 2 8 9 1],4)
ans =
    1.500 3.500 3.500 4.000 6.500 5.000 4.500

medfilt1([4 3 5 2 8 9 1],3)
ans =
     3    4    3    5    8    8    1
```

See the `medfilt2` function in the Image Processing Toolbox™ documentation for information on two-dimensional median filtering.

Communications Applications

In this section...
"Modulation" on page 7-32
"Demodulation" on page 7-33
"Voltage Controlled Oscillator" on page 7-36

Modulation

Modulation varies the amplitude, phase, or frequency of a *carrier signal* with reference to a *message signal*. The modulate function modulates a message signal with a specified modulation method.

The basic syntax for the modulate function is

```
y = modulate(x,fc,fs,'method',opt)
```

where:

- x is the message signal.
- fc is the carrier frequency.
- fs is the sampling frequency.
- *method* is a flag for the desired modulation method.
- opt is any additional argument that the method requires. (Not all modulation methods require an option argument.)

The table below summarizes the modulation methods provided; see the documentation for modulate, demod, and vco for complete details on each.

Method	Description
amdsb-sc or am	Amplitude modulation, double sideband, suppressed carrier
amdsb-tc	Amplitude modulation, double sideband, transmitted carrier

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

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

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

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

Demodulation

The demod function performs *demodulation*, that is, it obtains the original message signal from the modulated signal:

The syntax for demod is

```
x = demod(y,fc,fs,'method',opt)
```

demod uses any of the methods shown for modulate, but the syntax for quadrature amplitude demodulation requires two output parameters:

```
[X1,X2] = demod(y,fc,fs,'qam')
```

If the input y is an array, demod demodulates all columns.

Try modulating and demodulating a signal. A 50 Hz sine wave sampled at 1000 Hz is

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

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

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

To plot portions of the original, modulated, and demodulated signal:

```
figure; plot(t(1:150),x(1:150)); title('Original Signal');
figure; plot(t(1:150),y(1:150)); title('Modulated Signal');
figure; plot(t(1:150),z(1:150)); title('Demodulated Signal');
```
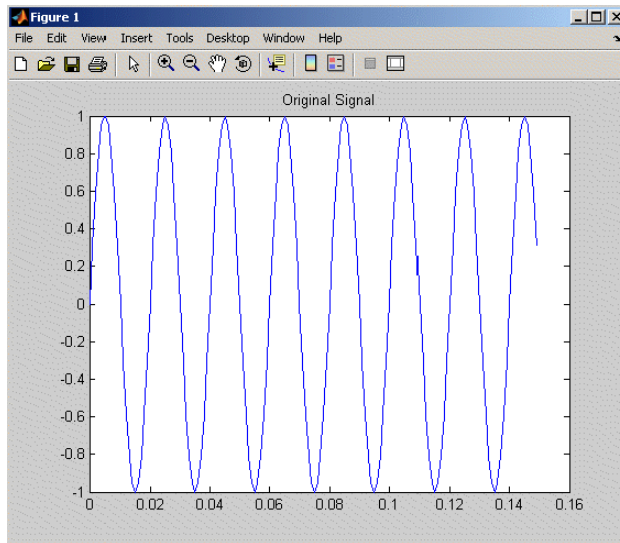

Original Signal
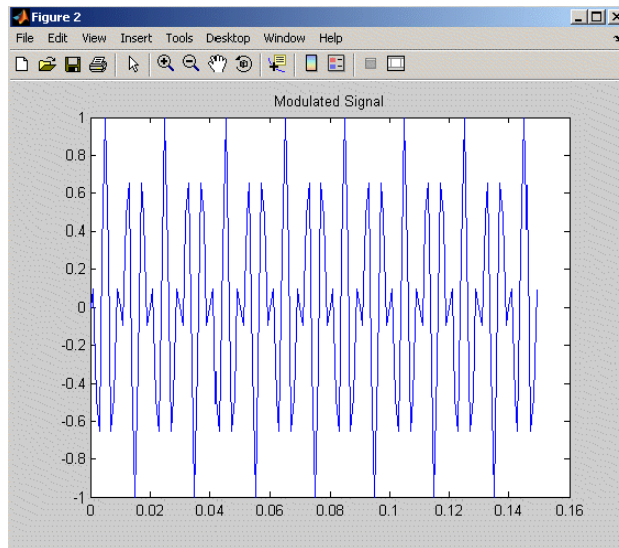
Modulated Signal
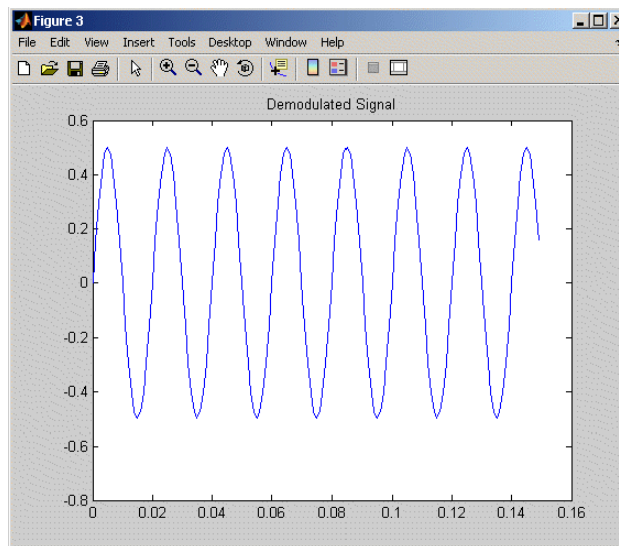
Demodulated Signal

Note The demodulated signal is attenuated because demodulation includes two steps: multiplication and lowpass filtering. The multiplication produces a component with frequency centered at 0 Hz and a component with frequency at twice the carrier frequency. The filtering removes the higher frequency component of the signal, producing the attenuated result.

Voltage Controlled Oscillator

The voltage controlled oscillator function vco creates a signal that oscillates at a frequency determined by the input vector. The basic syntax for vco is

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

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

To scale the frequency modulation range, use

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

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

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

See "FFT-Based Time-Frequency Analysis" on page 7-30 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, "Filtering, Linear Systems and Transforms Overview" 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

Chirp z-Transform

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 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

$$z_l = A\,W^{-l}, \ l = 0, \ldots, M-1$$

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.')
```
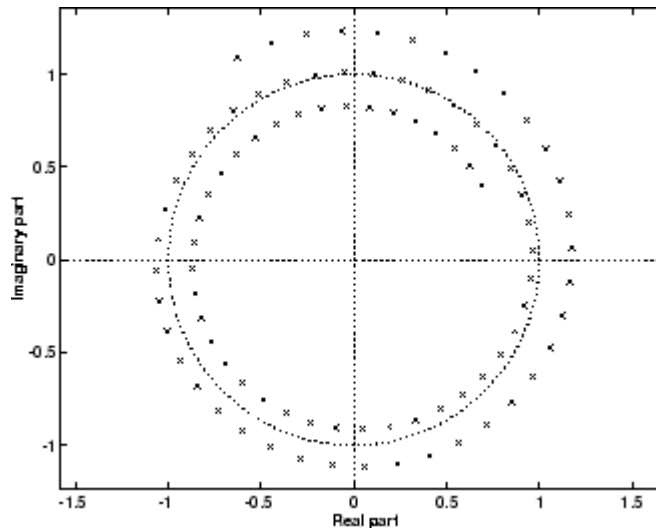
`czt(x,M,W,A)` computes the *z*-transform of x on these points.

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 discrete cosine transform (DCT), closely related to the DFT. The DCT's energy compaction properties are useful for applications like signal coding. 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

$$
w(k) = \begin{cases} \dfrac{1}{\sqrt{N}}, & k = 1 \\[3mm] \sqrt{\dfrac{2}{N}}, & 2 \le k \le N \end{cases}
$$

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

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

where

$$
w(n) = \begin{cases} \dfrac{1}{\sqrt{N}}, & n = 1 \\[3mm] \sqrt{\dfrac{2}{N}}, & 2 \le n \le N \end{cases}
$$

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 w/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')
```
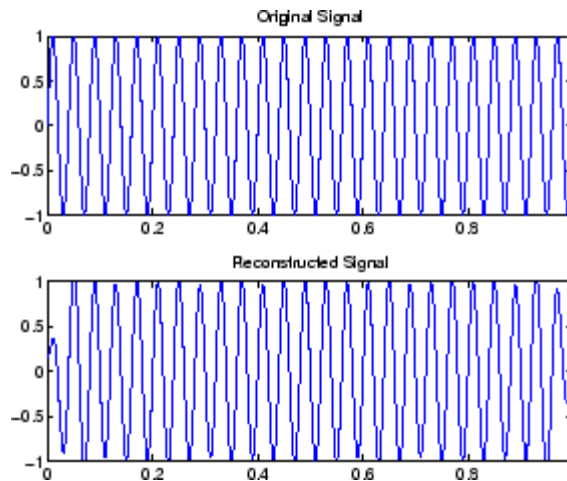

One measure of the accuracy of the reconstruction is

```
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 Hilbert transform facilitates the formation of the analytic signal. The analytic signal is useful in the area of communications, particularly in bandpass signal processing. The toolbox function `hilbert` computes the Hilbert transform for a real input sequence x and returns a complex result of the same length

```
y = hilbert(x)
```

where the real part of y is the original real data and the imaginary part is the actual Hilbert transform. y is sometimes called the *analytic signal*, in reference to the continuous-time analytic signal. A key property of the discrete-time analytic signal is that its *z*-transform is 0 on the lower half of the unit circle. Many applications of the analytic signal are related to this property; for example, the analytic signal is useful in avoiding aliasing effects for bandpass sampling operations. The magnitude of the analytic signal is the complex envelope of the original signal.

The Hilbert transform is related to the actual data by a 90° 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)),':');
axis([0 0.05 -1.1 2]);
legend('Real Part','Imaginary Part','location','northeast');
```
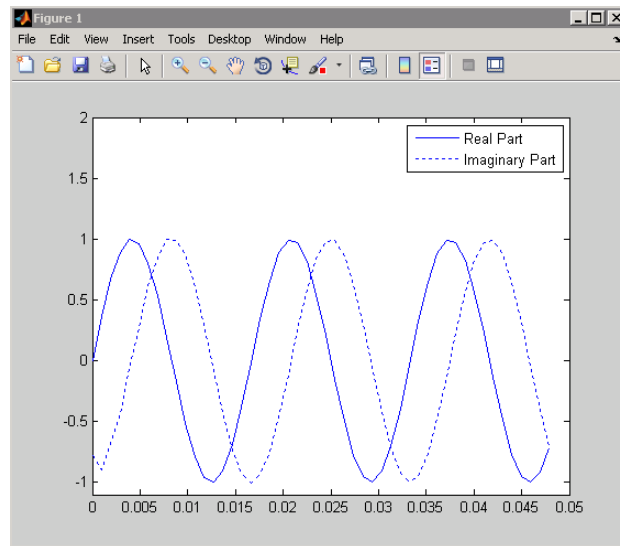
The analytic signal is useful in calculating *instantaneous attributes* of a time series, the attributes of the series at any point in time. The instantaneous amplitude of the input sequence is the amplitude of the analytic signal. The instantaneous phase angle of the input sequence is the (unwrapped) `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`.

Walsh–Hadamard Transform

The Walsh–Hadamard transform is a non-sinusoidal, orthogonal transformation technique that decomposes a signal into a set of basis functions. These basis functions are Walsh functions, which are rectangular or square waves with values of +1 or –1. Walsh–Hadamard transforms are also known as Hadamard (see the `hadamard` function in the MATLAB software), Walsh, or Walsh-Fourier transforms.

The first eight Walsh functions have these values:

Index	Walsh Function Values
0	1 1 1 1 1 1 1 1
1	1 1 1 1 -1 -1 -1 -1
2	1 1 -1 -1 -1 -1 1 1
3	1 1 -1 -1 1 1 -1 -1
4	1 -1 -1 1 1 -1 -1 1
5	1 -1 -1 1 -1 1 1 -1
6	1 -1 1 -1 -1 1 -1 1
7	1 -1 1 -1 1 -1 1 -1

The Walsh–Hadamard transform returns sequency values. Sequency is a more generalized notion of frequency and is defined as one half of the average number of zero-crossings per unit time interval. Each Walsh function has a unique sequency value. You can use the returned sequency values to estimate the signal frequencies in the original signal.

Three different ordering schemes are used to store Walsh functions: sequency, Hadamard, and dyadic. Sequency ordering, which is used in signal processing applications, has the Walsh functions in the order shown in the table above. Hadamard ordering, which is used in controls applications, arranges them as 0, 4, 6, 2, 3, 7, 5, 1. Dyadic or gray code ordering, which is used in mathematics, arranges them as 0, 1, 3, 2, 6, 7, 5, 4.

The Walsh–Hadamard transform is used in a number of applications, such as image processing, speech processing, filtering, and power spectrum analysis. It is very useful for reducing bandwidth storage requirements and spread-spectrum analysis. Like the FFT, the Walsh–Hadamard transform has a fast version, the fast Walsh–Hadamard transform (fwht). Compared to the FFT, the FWHT requires less storage space and is faster to calculate because it uses only real additions and subtractions, while the FFT requires complex values. The FWHT is able to represent signals with sharp discontinuities more accurately using fewer coefficients than the FFT. Both the FWHT and the inverse FWHT (ifwht) are symmetric and thus, use identical calculation processes. The FWHT and IFWHT for a signal x(t) of length N are defined as:

$$y_n = \frac{1}{N} \sum_{i=0}^{N-1} x_i WAL(n,i)$$

$$x_i = \sum_{i=0}^{N-1} y_n WAL(n,i)$$

where $i = 0,1,..., N\text{-}1$ and *WAL(n,i)* are Walsh functions. Similar to the Cooley-Tukey algorithm for the FFT, the *N* elements are decomposed into two sets of *N*/2 elements, which are then combined using a butterfly structure to form the FWHT. For images, where the input is typically a 2–D signal, the FWHT coefficients are calculated by first evaluating across the rows and then evaluating down the columns.

For the following simple signal, the resulting FWHT shows that x was created using Walsh functions with sequence values of 0, 1, 3, and 6, which are the non-zero indices of the transformed x. The inverse FWHT recreates the original signal.

```
x = [4 2 2 0 0 2 -2 0]
y = fwht(x)
y
    1    1    0    1    0    0    1    0

x1 = ifwht(y)
x1
    4    2    2    0    0    2    -2    0
```

Using Walsh-Hadamard Transform for Spectral Analysis and Compression of ECG Signals

The following example uses an electrocardiogram (ECG) signal to illustrate working with the Walsh-Hadamard transform. ECG signals typically are very large and need to be stored for analysis and retrieval at a future time. Walsh-Hadamard transforms are particularly well-suited to this application because they provide compression and thus, require less storage space, plus they also provide rapid signal reconstruction.

Start with an ECG signal. For this example, we replicate it to create a longer signal and insert some additional random noise. Then, transform the signal

using the fast Walsh-Hadamard transform and plot the original signal and the transformed signal.

```
xe = ecg(512);    % Single ecg wave
xr = repmat(xe,1,8); % Replicate it to create more data
x = xr + 0.1.*randn(1,length(xr));  % Add noise

% Fast Walsh-Hadamard transform. Use default values
% for the number of points to use for the transform and
% for the ordering - sequency -- why???

y = fwht(x);
figure('color','white');
subplot(2,1,1);
plot(x);        % Plot original noisy signal
xlabel('Sample index');
ylabel('Amplitude');
title('ECG Signal');
subplot(2,1,2);
plot(abs(y))      % Plot magnitude of transformed signal
xlabel('Sequency index');
ylabel('Magnitude');
title('WHT Coefficients');
```
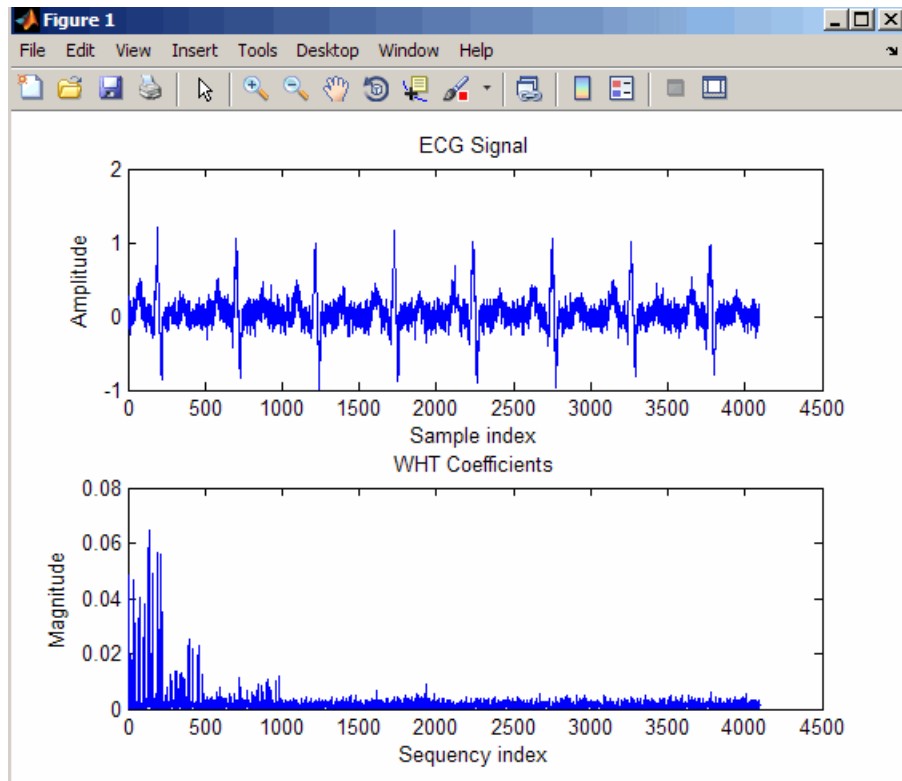
The plot shows that most of the signal energy is in the lower sequency values below approximately 1100. We will store only the first 1024 coefficients and see if the signal can be accurately reconstructed from only these stored coefficients.

```
y(1025:length(x)) = O;          % Zeroing out the higher coefficients
xHat = ifwht(y);                % Signal reconstruction using inverse WHT
figure('color','white');
plot(x);
hold on
plot(xHat,'r');
xlabel('Sample index');
ylabel('ECG signal amplitude');
legend('Original Signal','Reconstructed Signal');
```
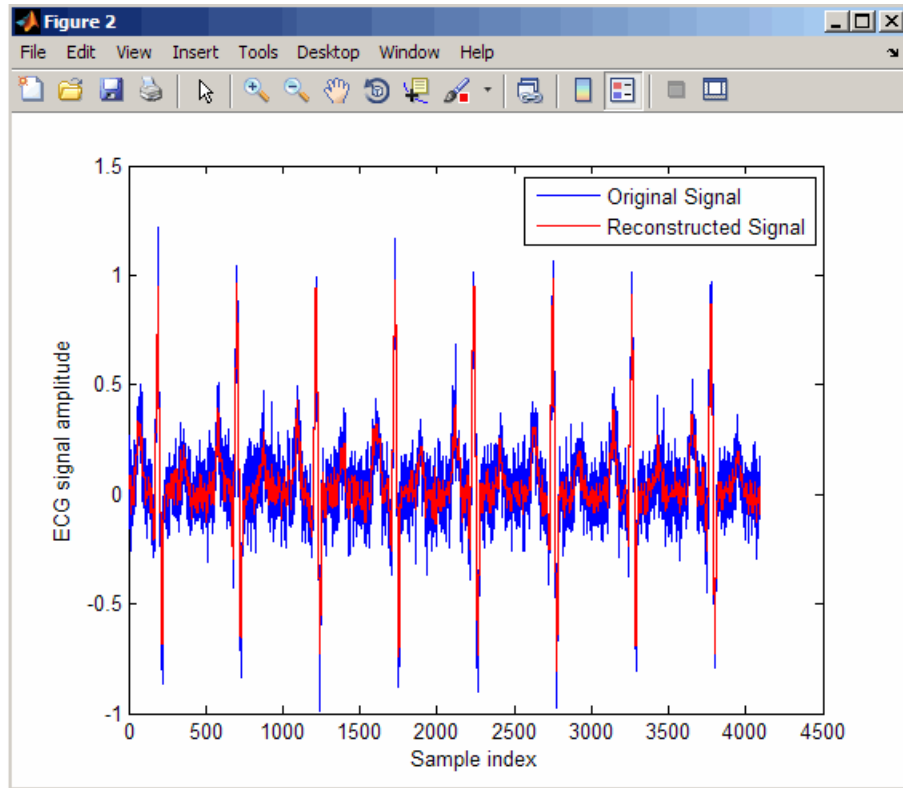
The reproduced signal is very close to the original signal.

To reconstruct the original signal, we stored only the first 1024 coefficients and the ECG signal length. This represents a compression ratio of approximately 4:1. Storing more coefficients is a tradeoff between increased resolution and increased noise, while storing fewer coefficients may cause loss of peaks.

```
req = [length(x) y(1:1024)];
whos x req
```

Name	Size	Bytes	Class	Attributes
req	1x1025	8200	double	
x	1x4096	32768	double	

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] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975, Section 10.5.3.

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

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

8

SPTool: A Signal Processing GUI Suite

SPTool: An Interactive Signal Processing Environment

In this section...
"SPTool Overview" on page 8-2
"SPTool Data Structures" on page 8-3

SPTool Overview

SPTool is an interactive GUI for digital signal processing used to

- Analyze signals

- Design filters

- Analyze (view) filters

- Filter signals

- Analyze signal spectra

You can accomplish these tasks using four GUIs that you access from within SPTool:

- The "Signal Browser" on page 8-7 is for analyzing signals. You can also play portions of signals using your computer's audio hardware.

- The Chapter 5, "FDATool: A Filter Design and Analysis GUI" is available for designing or editing FIR and IIR digital filters. Most Signal Processing Toolbox filter design methods available at the command line are also available in FDATool. Additionally, you can use FDATool to design a filter by using the "Pole/Zero Editor" on page 8-49 to graphically place poles and zeros on the z-plane.

- The "Filter Visualization Tool" on page 8-12 (FVTool) is for analyzing filter characteristics.

- The "Spectrum Viewer" on page 8-14 is for spectral analysis. You can use Signal Processing Toolbox spectral estimation methods to estimate the power spectral density of a signal.

SPTool Data Structures

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

You can bring signals, filters, or spectra from the MATLAB workspace into the SPTool workspace using **File > Import**. For more information, see "Importing Filters and Spectra" on page 8-33. Signals, filters, or spectra that you create in (or import into) the SPTool workspace exist as MATLAB structures. See the MATLAB documentation for more information on MATLAB structures.

When you use **File > Export** to save signals, filters, and spectra that you create or modify in SPTool, these are also saved as MATLAB structures. For more information on exporting, see "Exporting Signals, Filters, and Spectra" on page 8-28.

Opening SPTool

To open SPTool, type

```
sptool
```
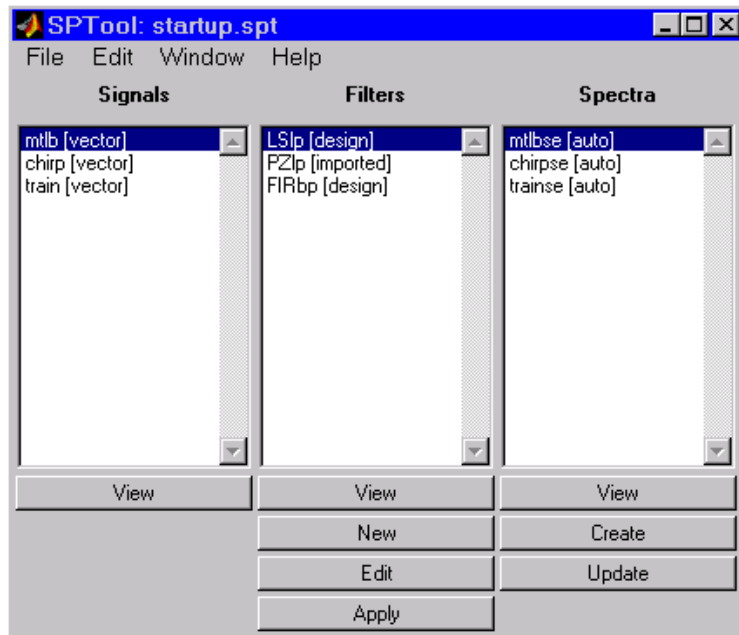

When you first open SPTool, it contains a collection of default signals, filters, and spectra. To specify your own preferences for what signals, filters, and spectra to see when SPTool opens see "Setting Preferences" on page 8-44.

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

- Signal Browser
- Filter Visualization Tool
- Spectrum Viewer

You can access FDATool by clicking **New** to create a new filter or `Edit` to edit a selected filter. Clicking **Apply** applies a selected filter to a selected signal.

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

Getting Context-Sensitive Help

To find information on a particular region of the "Signal Browser" on page 8-7, "FDATool" on page 8-10, or "Spectrum Viewer" on page 8-14:

1 Click **What's this?** .

2 Click on the region of the GUI you want information on.

You can also use **Help > What's This?** to launch context-sensitive help.

Signal Browser

In this section...
"Overview of the Signal Browser" on page 8-7
"Opening the Signal Browser" on page 8-7

Overview of the Signal Browser

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

Using the Signal Browser you can

- Analyze and compare vector or array (matrix) signals.
- Zoom in on portions of signal data.
- Measure a variety of characteristics of signal data.
- Compare multiple signals.
- Play portions of signal data on audio hardware.
- Print signal plots.

Opening the Signal Browser

To open the Signal Browser from SPTool:

1 Select one or more signals in the **Signals** list in SPTool.

2 Click **View** under the **Signals** list.
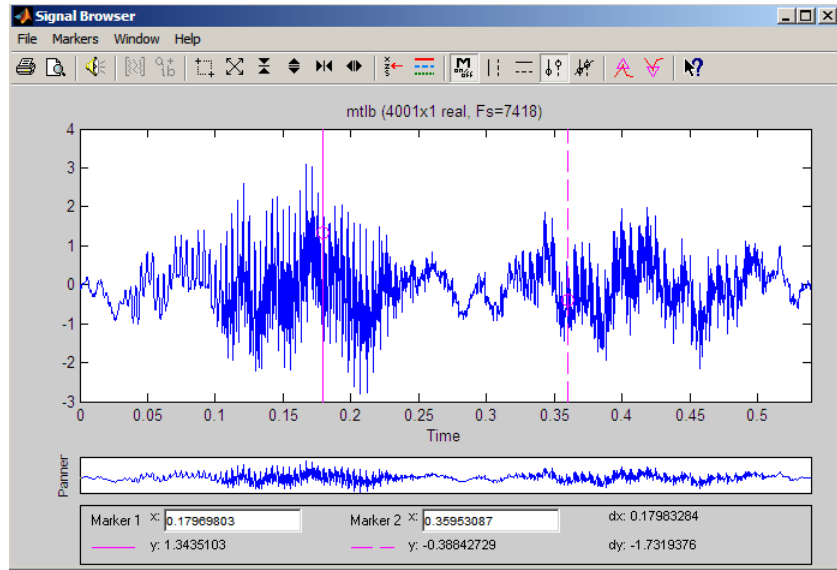
The Signal Browser has the following components:

- A display region for analyzing signals, including markers for measuring, comparing, or playing signals

- A "panner" that displays the entire signal length, highlighting the portion currently active in the display region

- A marker measurements area

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

Icon	Description
	Print and print preview
	Play an audio signal. The function `soundsc` is used to play the signal.
	Display array and complex signals
	Zoom the signal in and out

Icon	Description
	Select one of several loaded signals
	Set the display color and line style of a signal
	Toggle the markers on and off
	Set marker types (See "Making Signal Measurements with Markers" on page 8-42)
	Turn on the What's This help

FDATool

You can use the Filter Design and Analysis Tool (`fdatool`) to design and edit filters.

To open FDATool from SPTool, click **New** under the **Filters** list to create a new filter or select one of the filters in the **Filters** list in SPTool and click **Edit** to edit that filter.
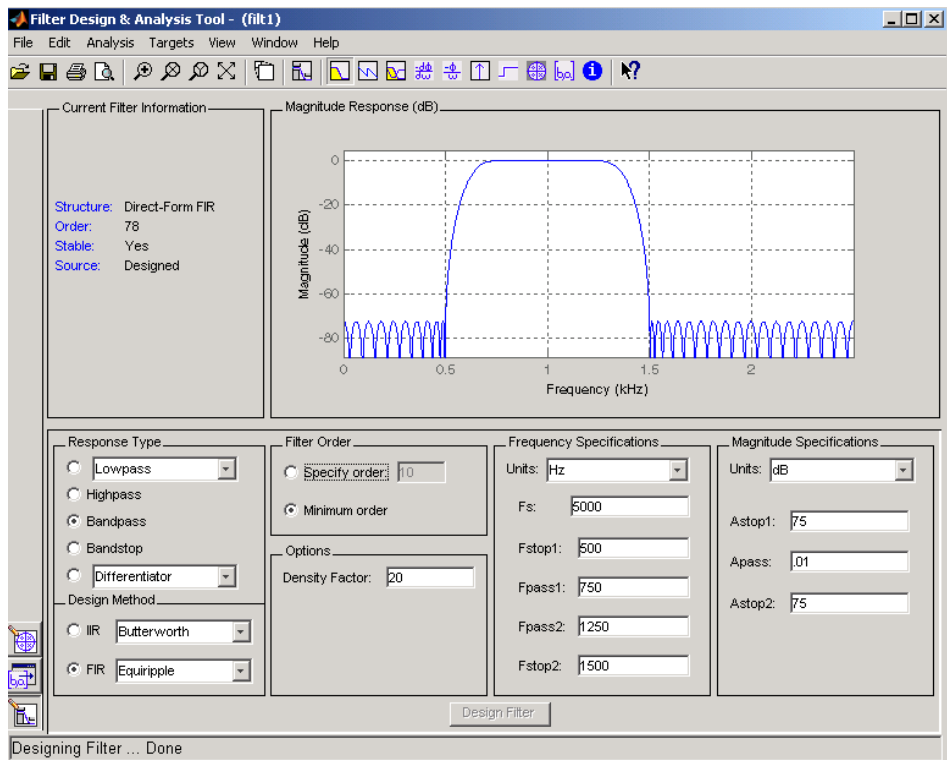
For details about FDATool, see Chapter 5, "FDATool: A Filter Design and Analysis GUI".

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

Filter Visualization Tool

In this section...

Connection between FVTool and SPTool

You can use the Filter Visualization Tool to analyze response characteristics of the selected filter(s). See `fvtool` for detailed information about FVTool.

If you start FVTool by clicking the SPTool **Filter View** button, that FVTool is linked to SPTool. Any changes made in SPTool to the filter are immediately reflected in FVTool. The FVTool title bar includes "SPTool" to indicate the link.

If you start an FVTool by clicking the **New** button or by selecting **File > New** from within FVTool, that FVTool is a standalone version and is not linked to SPTool.

Note Every time you click the **Filter View** button a new, linked FVTool starts. This allows you to view multiple analyses simultaneously.

Opening the Filter Visualization Tool

You open FVTool from SPTool as follows.

1 Select one or more filters in the **Filters** list in SPTool.

2 Click the **View** button under the **Filters** list.

When you first open FVTool, it displays the selected filter's magnitude plot.
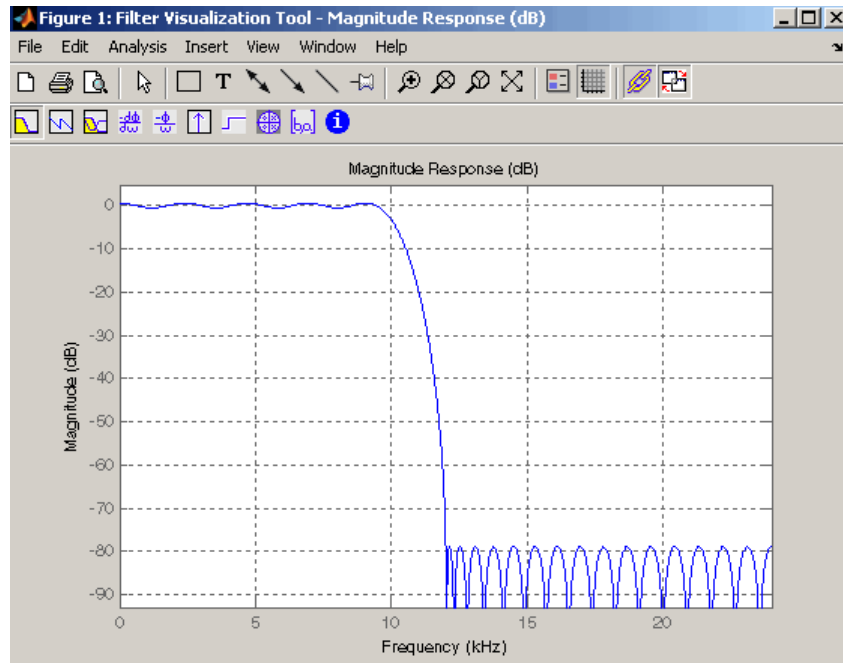
Analysis Parameters

In the plot area of any filter response plot, right-click and select **Analysis Parameters** to display details about the displayed plot. See "Analysis Parameters" in the FDATool online help for more information.

You can change any parameter in a linked FVTool, except the sampling frequency. You can only change the sampling frequency using the SPTool **Edit > Sampling Frequency** or the SPTool **Filters Edit** button.

Spectrum Viewer

In this section...
"Spectrum Viewer Overview" on page 8-14
"Opening the Spectrum Viewer" on page 8-14

Spectrum Viewer Overview

You can use the Spectrum Viewer for estimating and analyzing a signal's power spectral density (PSD). You can use the PSD estimates to understand a signal's frequency content.

The Spectrum Viewer provides the following functionality.

- Analyze and compare spectral density plots.
- Use different spectral estimation methods to create spectra:
  - Burg (`pburg`)
  - Covariance (`pcov`)
  - FFT (`fft`)
  - Modified covariance (`pmcov`)
  - MTM (multitaper method) (`pmtm`)
  - MUSIC (`pmusic`)
  - Welch (`pwelch`)
  - Yule-Walker AR (`pyulear`)
- Modify power spectral density parameters such as FFT length, window type, and sample frequency.
- Print spectral plots.

Opening the Spectrum Viewer

To open the Spectrum Viewer and create a PSD estimate from SPTool:

1 Select a signal from the **Signal** list box in SPTool.

2 Click **Create** in the **Spectra** list.

3 Click **Apply** in the Spectrum Viewer.

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

1 Select a PSD estimate from the **Spectra** list box in SPTool.

2 Click **View** in the Spectra list.

For example:

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

2 Click **Create** in SPTool to open the Spectrum Viewer.
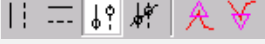
3 Click **Apply** in the Spectrum Viewer to plot the spectrum.
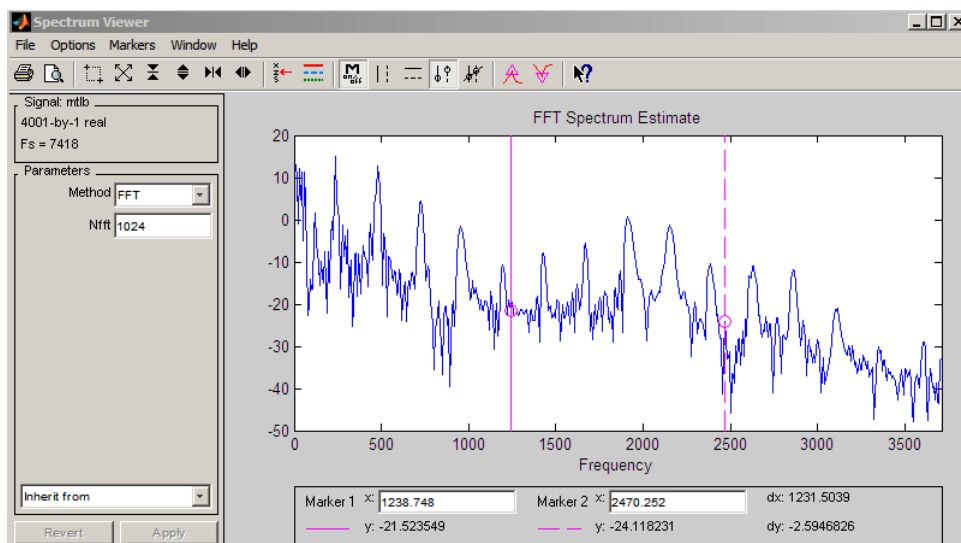
The Spectrum Viewer has the following components:

- A signal identification region that provides information about the signal whose power spectral density estimate is displayed

- A Parameters region for modifying the PSD parameters

- A display region for analyzing spectra and an **Options** menu for modifying display characteristics

- Spectrum management controls

  - **Inherit from** menu to inherit PSD specifications from another PSD object listed in the menu

  - Revert button to revert to the named PSD's original specifications

  - **Apply** button for creating or updating PSD estimates

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

Icon	Description
	Print and print preview
	Zoom the signal in and out
	Select one of several loaded signals
	Set the display color and line style of a signal
	Toggle the markers on and off
	Set marker types
	Turn on the What's This help

Filtering and Analysis of Noise

Overview

The following sections provide an example of using the GUI-based interactive tools to:

- Design and implement an FIR bandpass digital filter
- Apply the filter to a noisy signal
- Analyze signals and their spectra

The steps include:

1 "Importing a Signal into SPTool" on page 8-17

2 Designing a bandpass filter using FDATool

3 Applying the filter to the original noise signal to create a bandlimited noise signal

4 Comparing the time domain information of the original and filtered signals using the Signal Browser

5 Comparing the spectra of both signals using the Spectrum Viewer

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 **File > Import**. The Import to SPTool dialog opens.
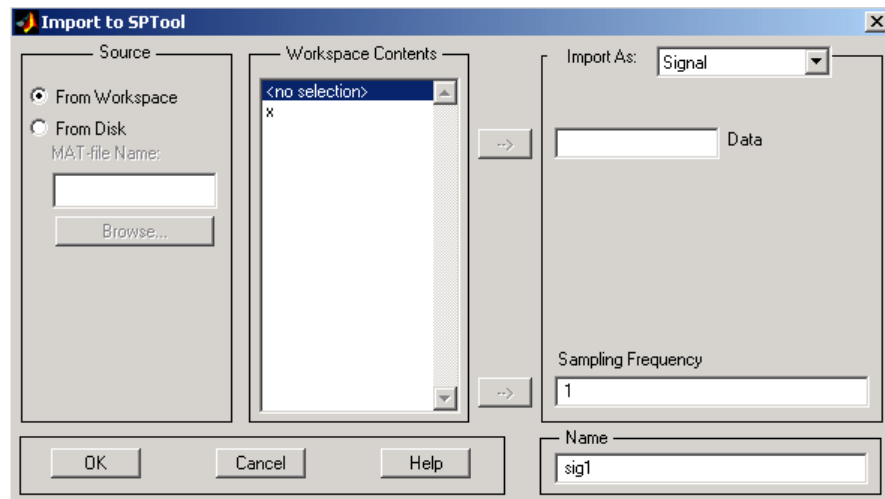
The variable x is displayed in the **Workspace Contents** list. (If it is not, select the **From Workspace** radio button to display the contents of the workspace.)

4 Select the signal and import it into the **Data** field:

  a Select the signal variable x in the **Workspace Contents** list.

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

  c Click on the arrow to the left of the **Data** field or type x in the **Data** field.

  d Type 5000 in the **Sampling Frequency** field.

  e Name the signal by typing noise in the **Name** field.

  f Click **OK**.

  The signal noise[vector] appears and is selected in SPTool's **Signals** list.

Note You can import filters and spectra into SPTool in much the same way as you import signals. See "Importing Filters and Spectra" on page 8-33 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 8-37 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 FDATool.

In this example, you

1 Open a default filter in FDATool.

2 Specify an equiripple bandpass FIR filter.

Opening FDATool

To open FDATool, click **New** in SPTool. FDATool opens with a default filter named filt1.

Specifying the Bandpass Filter

Design an equiripple bandpass FIR filter with the following characteristics:

- Sampling frequency of 5000 Hz

- Stopband frequency ranges of [0 500] Hz and [1500 2500] Hz

- Passband frequency range of [750 1250] Hz

- Ripple in the passband of 0.01 dB

- Stopband attenuation of 75 dB

To modify the filter in FDATool to meet these specifications, you need to

1 Select Bandpass from the **Response Type** list.

2 Verify that **FIR Equiripple** is selected as the **Design Method**.

3 Verify that **Minimum order** is selected as the **Filter Order** and that the **Density Factor** is set to 20.

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

Units	Hz
Fs	5000
Fstop1	500
Fpass1	750
Fpass2	1250
Fstop2	1500

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

Units	dB
Astop1	75
Apass	0.01
Astop2	75

6 Click **Design Filter** to design the new filter. When the new filter is designed, the magnitude response of the filter is displayed.
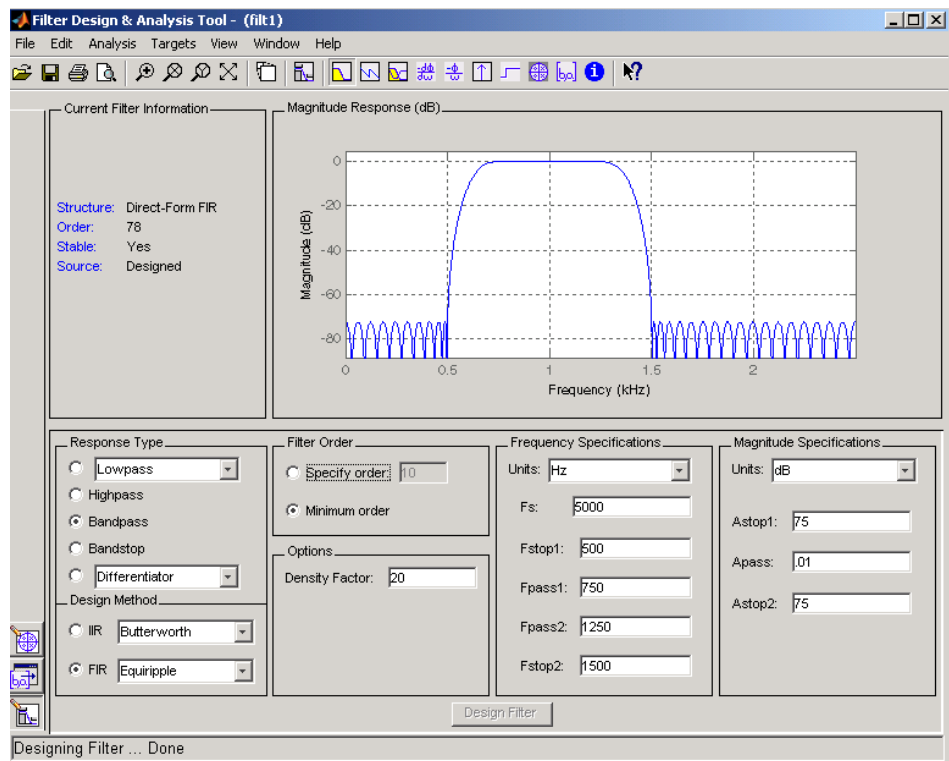
The resulting filter is an order-78 bandpass equiripple filter.

Applying a Filter to a Signal

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

To apply the filter `filt1` you just created to the signal `noise`,

1 In SPTool, select the signal `noise[vector]` from the **Signals** list and select the filter (named `filt1[design]`) from the **Filters** list.
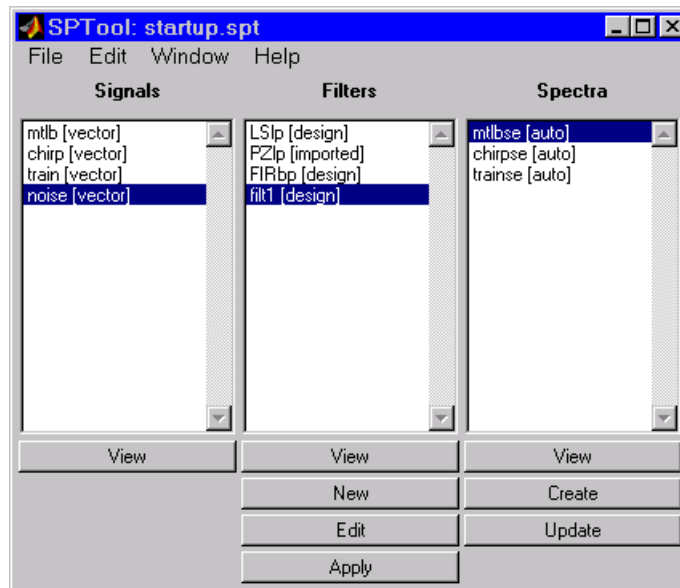
2 Click **Apply** under the **Filters** list.
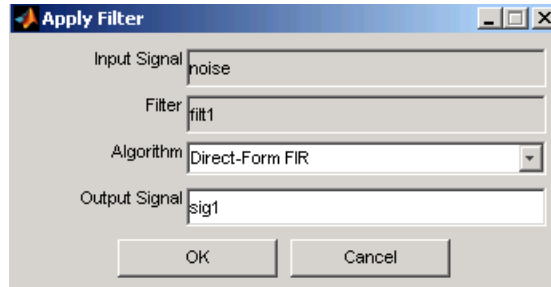
3 Leave the **Algorithm** as `Direct-Form FIR`.

> **Note** You can apply one of two filtering algorithms to FIR filters. The default algorithm is specific to the filter structure, which is shown in the FDATool Current Filter Info frame. Alternately for FIR filters, `FFT based FIR (fftfilt)` uses the algorithm described in `fftfilt`.
>
> For IIR filters, the alternate algorithm is a zero-phase IIR that uses the algorithm described in `filtfilt`.

4 Enter `blnoise` as the **Output Signal** name.

5 Click **OK** to close the Apply Filter dialog box.

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

Analyzing a Signal

You can analyze and print signals using the Signal Browser. You can also play the signals if your computer has audio output capabilities.

For example, compare the signal `noise` to the filtered signal `blnoise`:

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

2 Click **View** under the **Signals** list.

The Signal Browser is activated, and both signals are displayed in the display region. (The names of both signals are shown above the display region.) Initially, the original `noise` signal covers up the bandlimited `blnoise` signal.

3 Push the selection button ![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, ![icon].
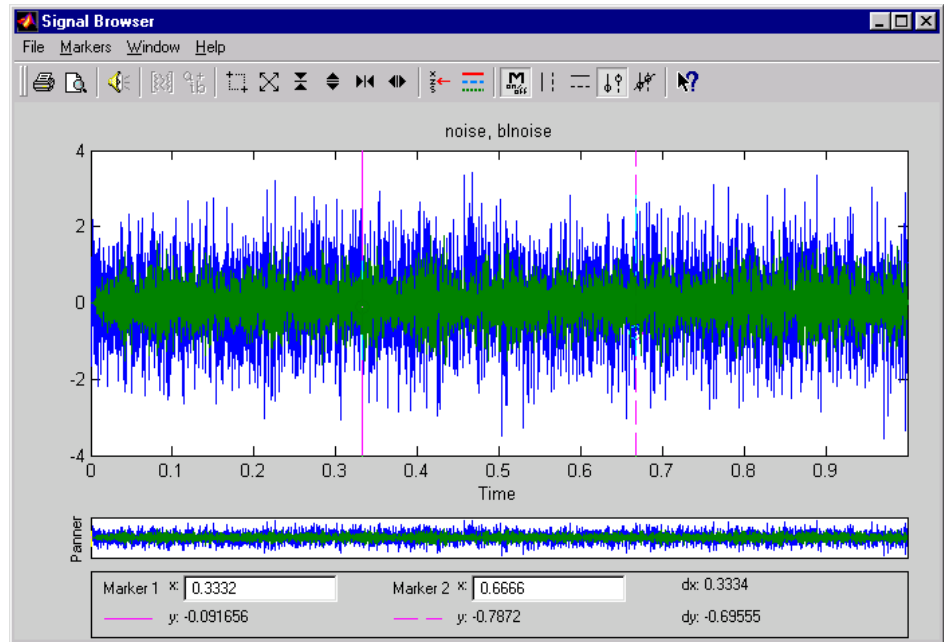
Playing a Signal

When you click **Play** in the Signal Browser toolbar, , the active signal is played on the computer's audio hardware:

1 To hear a portion of the active (selected) signal

   a Use the vertical markers to select a portion of the signal you want to play. Vertical markers are enabled by the and buttons.

   b Click **Play**.

2 To hear the other signal

   a Select the signal as in step above. You can also select the signal directly in the display region.

   b Click **Play** again.

Printing a Signal

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

You can use the line display buttons to maximize the visual contrast between the signals by setting the line color for noise to gray and the line color for blnoise to white. Do this before printing two signals together.

Note You can follow the same rules to print spectra, but you can't print filter responses directly from SPTool.

Use the Signal Browser region in the Preferences dialog box in SPTool to suppress printing of both the panner and the marker settings.

To print both signals, click **Print** in the Signal Browser toolbar.

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 **Window > SPTool** in any active open GUI. SPTool is now the active window.

2 Select the noise[vector] signal in the **Signals** list of SPTool.

3 Click **Create** in the **Spectra** list.

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

4 Click **Apply** in the Spectrum Viewer to compute and display the PSD estimate spect1 using the default parameters.

The PSD of the noise signal is displayed in the display region. The identifying information for the PSD's associated signal (noise) is displayed above the Parameters region.

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

5 Follow steps 1 through 4 for the bandlimited noise signal blnoise to create a second PSD estimate spect2.

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

Opening the Spectrum Viewer with Two Spectra

1 Reactivate SPTool again, as in step 1 above.

2 **Shift+click** on spect1 and spect2 in the **Spectra** list to select them both.

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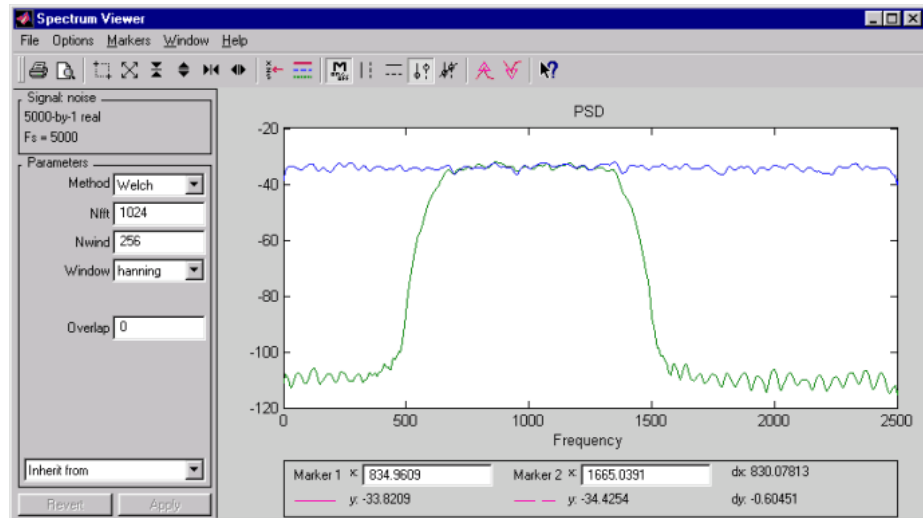
Printing the Spectra

Before printing the two spectra together, use the color and line style selection

button, ▦, to differentiate the two plots by line style, rather than by color.

To print both spectra:

1 Click **Print Preview** ▣ in the toolbar on the Spectrum Viewer.

2 From the Spectrum Viewer Print Preview window, drag the legend out of the display region so that it doesn't obscure part of the plot.

3 Click **Print** in the Spectrum Viewer Print Preview window.

Exporting Signals, Filters, and Spectra

In this section...
"Opening the Export Dialog Box" on page 8-28
"Exporting a Filter to the MATLAB Workspace" on page 8-29

Opening the Export Dialog Box

To save the filter filt1 you just created in this example, open the Export dialog box with filt1 preselected:

1 Select filt1 in the SPTool **Filters** list.

2 Select **File > Export**.

The Export dialog box opens with filt1 preselected.
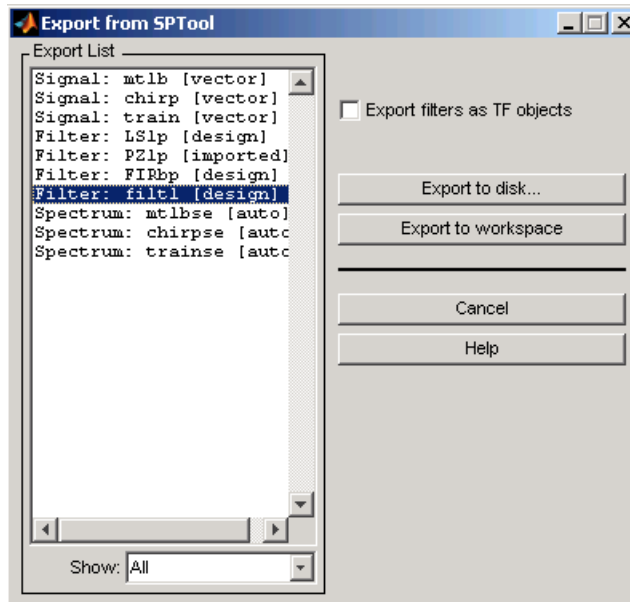
Exporting a Filter to the MATLAB Workspace

To export the filter filt1 to the MATLAB workspace:

1 Select `filt1` from the **Export List** and deselect all other items using **Ctrl+click**.

2 Click **Export to Workspace**.

Accessing Filter Parameters

Accessing Filter Parameters in a Saved Filter

The MATLAB structures created by SPTool have several associated fields, many of which are also MATLAB structures. See the MATLAB documentation for general information about MATLAB structures.

For example, after exporting a filter filt1 to the MATLAB workspace, type

```
filt1
```

to display the fields of the MATLAB filter structure. The tf field of the structure contains information that describes the filter.

The tf Field: Accessing Filter Coefficients

The tf field is a structure containing the transfer function representation of the filter. Use this field to obtain the filter coefficients;

- filt1.tf.num contains the numerator coefficients.
- filt1.tf.den contains the denominator coefficients.

The vectors contained in these structures represent polynomials in descending powers of *z*. The numerator and denominator polynomials are used to specify the transfer function

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

where:

- *b* is a vector containing the coefficients from the tf.num field.

- *a* is a vector containing the coefficients from the `tf.den` field.

- *m* is the numerator order.

- *n* is the denominator order.

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

Note The `FDAspecs` field of your filter contains internal information about FDATool and should not be changed.

Accessing Parameters in a Saved Spectrum

The following structure fields describe the spectra saved by SPTool.

Field	Description
P	The spectral power vector.
f	The spectral frequency vector.
confid	A structure containing the confidence intervals data • 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

Similarities to Other Procedures

The procedures are very similar to those explained in

- "Importing a Signal into SPTool" on page 8-17 for loading variables from the workspace

- "Loading Variables from the Disk" on page 8-37 for loading variables from your disk

Importing Filters

When you import filters, first select the appropriate filter form from the **Form** list. SPTool does not currently support the import of filter objects.
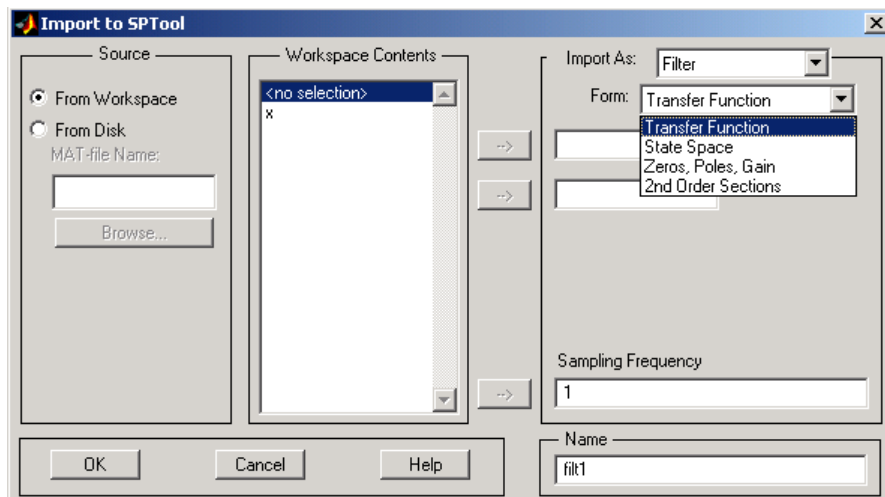
For every filter you specify a variable name or a value for the filter's sampling frequency in the **Sampling Frequency** field. Each filter form requires different variables.

Transfer Function

For `Transfer Function`, you specify the filter by its transfer function representation:

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

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

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

State Space

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

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

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

Zeros, Poles, Gain

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

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

- The **Zeros** field specifies a variable name or value for the zeros vector z, which contains the locations of m zeros.

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

- The **Gain** field specifies a variable name or value for the gain k.

Second Order Sections

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

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

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

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

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

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

Importing Spectra

When you import a power spectral density (PSD), you specify:

- A variable name or a value for the PSD vector in the **PSD** field

- A variable name or a value for the frequency vector in the **Freq. Vector** field

The PSD values in the **PSD** vector correspond to the frequencies contained in the **Freq. Vector** vector; the two vectors must have the same length.
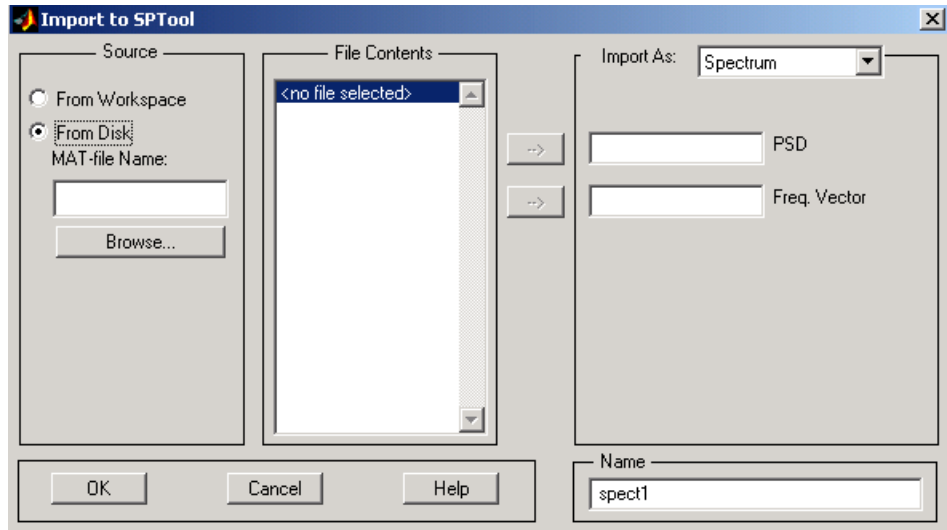
Loading Variables from the Disk

To import variables representing signals, filters, or spectra from a MAT-file on your disk;

1 Select the **From Disk** radio button and do either of the following:

- Type the name of the file you want to import into the **MAT-file Name** field and press either the **Tab** or the **Enter** key on your keyboard.

- Select **Browse**, and then find and select the file you want to import using **Select > File to Open**. Click **OK** to close that dialog.

In either case, all variables in the MAT-file you selected are displayed in the **File Contents** list.

2 Select the variables to be imported into SPTool.

You can now import one or more variables from the **File Contents** list into SPTool, as long as these variables are scalars, vectors, or matrices.

Saving and Loading Sessions

In this section...
"SPTool Sessions" on page 8-38
"Filter Formats" on page 8-38

SPTool Sessions

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

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

Filter Formats

When you start SPTool or open a session, the current filter design format preference is compared to the filter formats in the session. See "Setting Preferences" on page 8-44.

- If the formats match, the session opens.

- If the filter preference is FDATool, but the session contains Filter Designer filters, this warning displays:
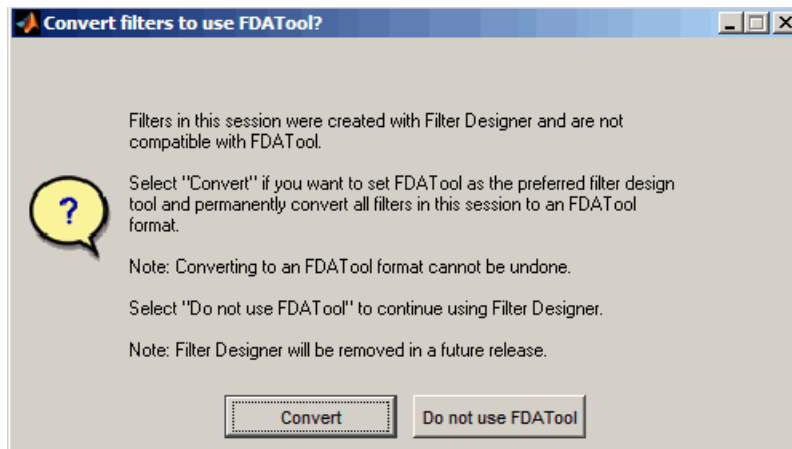
Click **Convert** to convert the filters to FDATool format. Click **Don't Use FDATool** to leave the filters in Filter Designer format and change the preference to **Use Filter Designer**.

- If the filter preference is **Use Filter Designer**, but the session contains FDATool filters, this warning displays:
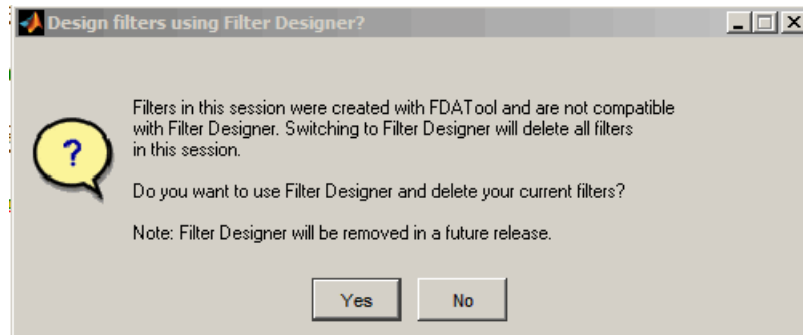
Click **Yes** to remove the current filters. Click **No** to leave the filters in FDATool.

Selecting Signals, Filters, and Spectra

All signals, filters, or spectra listed in SPTool exist as special MATLAB structures. You can bring data representing signals, filters, or spectra into SPTool from the MATLAB workspace. In general, you can select one or several items in a given list box. An item is selected when it is highlighted.

The **Signals** list shows all vector and array signals in the current SPTool session.

The **Filters** list shows all designed and imported filters in the current SPTool session.

The **Spectra** list shows all spectra in the current SPTool session.

You can select a single data object in a list, a range of data objects in a list, or multiple separate data objects in a list. You can also have data objects simultaneously selected in different lists:

- To select a single item, click it. All other items in that list box become deselected.

- To add or remove a range of items, **Shift+click** on the items at the top and bottom of the section of the list that you want to add. You can also drag your mouse pointer to select these items.

- To add a single data object to a selection or remove a single data object from a multiple selection, **Ctrl+click** on the object.

Editing Signals, Filters, or Spectra

You can edit selected items in SPTool by

1 Selecting the names of the signals, filters, or spectra you want to edit.

2 Selecting the appropriate **Edit** menu item:

- **Duplicate** to copy an item in an SPTool list
- **Clear** to delete an item in an SPTool list
- **Name** to rename an item in an SPTool list
- **Sampling Frequency** to modify the sampling frequency associated with either a signal (and its associated spectra) or filter in an SPTool list

The pull-down menu next to each menu item shows the names of all selected items.

You can also edit the following signal characteristics by right-clicking in the display region of the Signal Browser, the Filter Visualization Tool, or the Spectrum Viewer:

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

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

Making Signal Measurements with Markers

You can use the markers on the Signal Browser or the Spectrum Viewer to make measurements on either of the following:

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

The following marker buttons are included

Icon	Description
	Toggle markers on/off
	Vertical markers
	Horizontal markers
	Vertical markers with tracking
	Vertical markers with tracking and slope
	Display peaks (local maxima) You can find peaks in a signal from the command line with `findpeaks`
	Display valleys (local minima)

To make a measurement:

1 Select a line to measure (or play, if you are in the Signal Browser).

2 Select one of the marker buttons to apply a marker to the displayed signal.

3 Position a marker in the main display area by grabbing it with your mouse and dragging:

a Select a marker setting. If you choose the **Vertical**, **Track**, or **Slope** buttons, you can drag a marker to the right or left. If you choose the **Horizontal** button, you can drag a marker up or down.

b Move the mouse over the marker (1 or 2) that you want to drag.

The hand cursor with the marker number inside it 🖐 is displayed when your mouse passes over a marker.

c Drag the marker to where you want it on the signal

As you drag a marker, the bottom of the Signal Browser shows the current position of both markers. Depending on which marker setting you select, some or all of the following fields are displayed — **x1**, **y1**, **x2**, **y2**, **dx**, **dy**, **m**. These fields are also displayed when you print from the Signal Browser, unless you suppress them.

You can also position a marker by typing its **x1** and **x2** or **y1** and **y2** values in the region at the bottom.

Setting Preferences

In this section...

Overview of Setting Preferences

Use **File > Preferences** to customize displays and certain parameters for SPTool and its four component GUIs. If you change any preferences, a dialog box displays when you close SPTool asking if you want to save those changes. If you click **Yes**, the new settings are saved on disk and are used when you restart SPTool from the MATLAB workspace.

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

When you first select **Preferences**, the Preferences dialog box opens with
Markers selected by default.
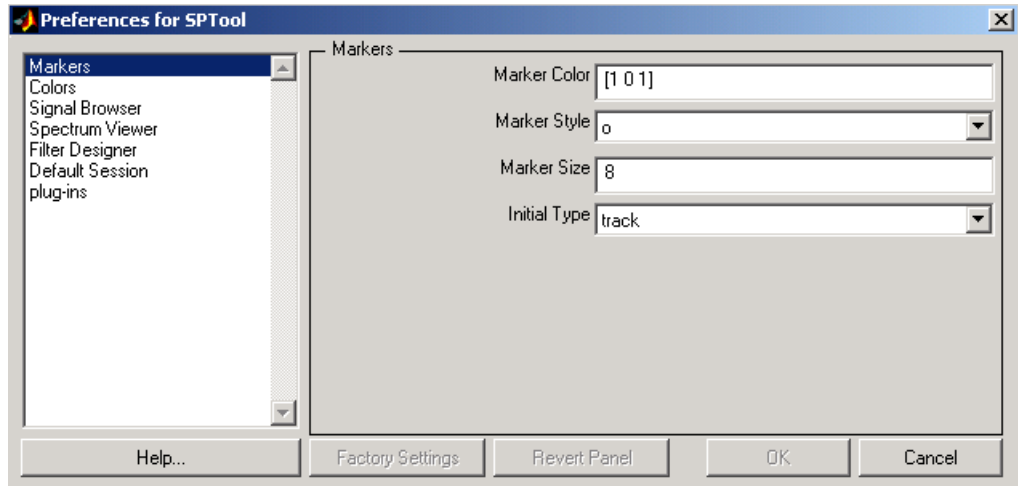
Change any marker settings, if desired. To change settings for another
category, click its name in the category list to display its settings. Most
of the fields are self-explanatory. Details of the Filter Design options are
described below.

Summary of Settable Preferences

In the Preferences regions, you can

- Select colors and markers for all displays.

- Select colors and line styles for displayed signals.

- Configure labels, and enable/disable markers, panner, and zoom in the
  Signal Browser.

- Configure display parameters, and enable/disable markers and zoom in
  the Spectrum Viewer.

- Select whether to use the default FDATool or the Filter Designer to design
  filters. FDATool is the recommended designer.

- Enable/disable use of a default session file.

- Export filters for use with Control System Toolbox software.

- Enable/disable search for plug-ins at startup.

Setting the Filter Design Tool

The Filter Designer options include radio buttons to select the filter design tool.
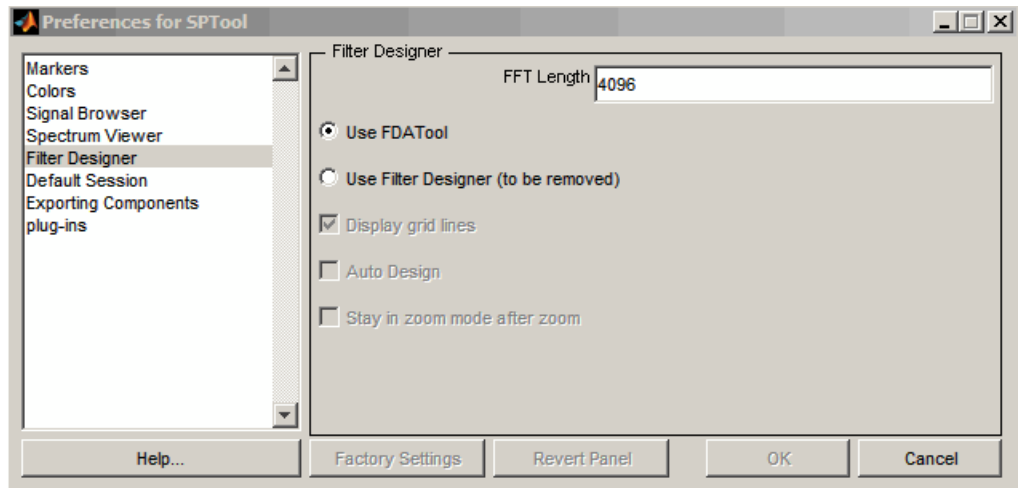
FDATool is the default and recommended tool. You can use Filter Designer, but Filter Designer will be removed in a future release. You cannot change this preference if either FDATool or the Filter Designer is open.

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

If you change the preference from **Use FDATool** to **Use Filter Designer**, a warning message appears. The warning message informs you that filters created with FDATool are not compatible with Filter Designer and that Filter Designer will be removed in the future.
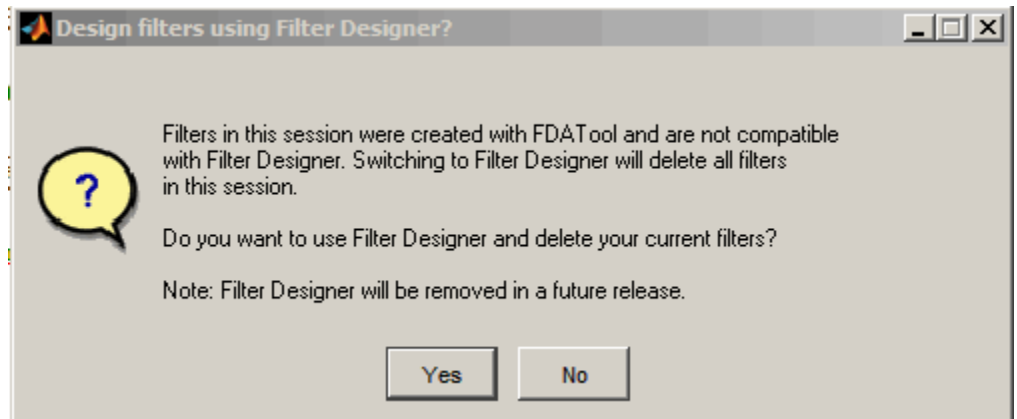
See "Saving and Loading Sessions" on page 8-38 for more information.

When you change the preference from **Use Filter Designer** to **Use FDATool**, a confirmation message appears indicating that switching will convert your filters to FDATool format. See "Saving and Loading Sessions" on page 8-38 for information on this message.

Changes to Filter Designer format are saved only if you save the session. Exiting the session prompts you to save changes to the `sigprefs.mat` and `startup.spt` files. Starting SPTool with Filter Designer specified as the filter design tool results in the warning:

The warning appears each time SPTool initializes unless you opt to not show the warning.

Using the Filter Designer

Filter Designer

"FDATool" on page 8-10 is the recommended filter design tool. Filter Designer will be removed in a future release. The following information is provided for users that choose to use Filter Designer while they transition to FDATool. 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.

Filter Types

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

- Bandpass
- Lowpass
- Bandstop
- Highpass

FIR Filter Methods

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

- Equiripple
- Least squares
- Window

IIR Filter Methods

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

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

Pole/Zero Editor

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

Spectral Overlay Feature

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

Opening the Filter Designer

Open the Filter Designer from SPTool by either:

- Clicking **New** in the **Filters** list in SPTool
- Selecting a filter you want to edit from the **Filters** list in SPTool, and then clicking **Edit**
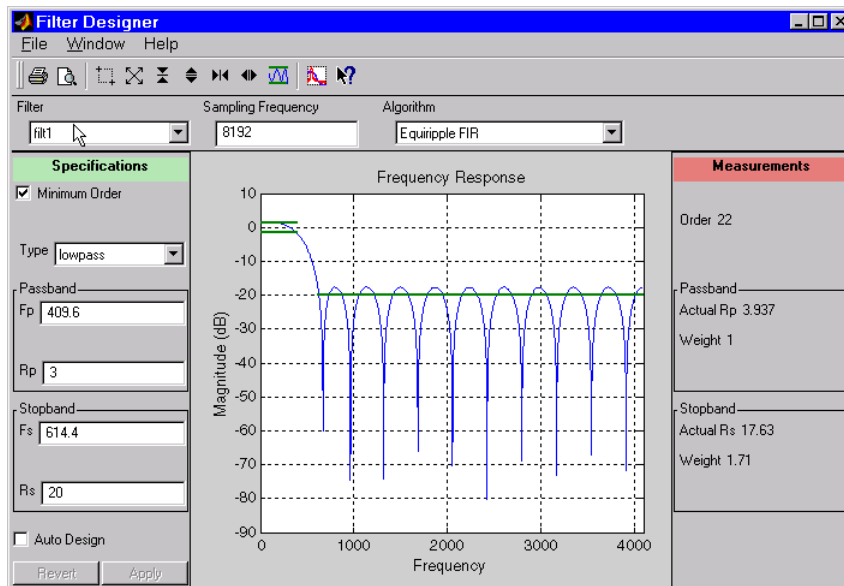
The Filter Designer has the following components:

- A pull-down **Filter** menu for selecting a filter from the list in SPTool

- A **Sampling Frequency** text box

- A pull-down **Algorithm** menu for selecting a filter design method or a pole-zero plot display

- A Specifications area for viewing or modifying a filter's design parameters or pole-zero locations

- A plot display region for graphically adjusting filter magnitude responses or the pole-zero locations

- A Measurements area for viewing the response characteristics and stability of the current filter

- A toolbar with the following buttons

Icon	Description
🖨 📄	Print and print preview
⬚ ⤢ ⤒ ⬍ ⊮ ◀▶	Zoom in and out
〰	Passband view
📉	Overlay spectrum
▶?	Turn on the What's This help

Accessing Filter Parameters in a Saved Filter

The MATLAB structures created by SPTool have several associated fields, many of which are also MATLAB structures. See the MATLAB documentation for general information about MATLAB structures.

For example, after exporting a filter filt1 to the MATLAB workspace, type

```
filt1
```

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

The tf Field: Accessing Filter Coefficients

The tf field is a structure containing the transfer function representation of the filter. Use this field to obtain the filter coefficients;

- filt1.tf.num contains the numerator coefficients.
- filt1.tf.den contains the denominator coefficients.

The vectors contained in these structures represent polynomials in descending powers of z. The numerator and denominator polynomials are used to specify the transfer function

8-51

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

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.

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

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

```
filt1.specs

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

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

The specifications include the parameter values from the Specifications region of the Filter Designer, such as band edges and filter order. For example, the filter above has the following specifications stored in `filt1.specs.fdremez`:

```
filt1.specs.fdremez

ans =
    setOrderFlag: 0
            type: 3
               f: [0 0.2000 0.3000 0.5000 0.6000 1]
               m: [6x1 double]
              Rp: 0.0100
              Rs: 75
              wt: [3.2371 1 3.2371]
           order: 78
```

Because certain filter parameters are unique to a particular design, this structure has a different set of fields for each filter design.

The table below describes the possible fields associated with the filter design specification field (the `specs` field) that can appear in the exported structure.

Parameter	Description
Beta	Kaiser window 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.
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 selected). Contains 0 if the filter order was computed automatically.
type	Contains 1 for lowpass, 2 for highpass, 3 for bandpass, or 4 for bandstop.
w3db	-3 dB frequency for Butterworth IIR designs.
wind	Vector of Kaiser window coefficients.
Wn	Cutoff frequency for the Kaiser window FIR filter when setOrderFlag = 1.
wt	Vector of weights, one weight per frequency band.

Designing a Filter with the Pole/Zero Editor

To design a filter transfer function using the Filter Designer Pole/Zero Editor:

1 Select the Pole/Zero Editor option from the **Algorithm** list to open the Pole/Zero Editor in the Filter Designer display.

```
Equiripple FIR
Least Squares FIR
Kaiser Window FIR
Butterworth IIR
Chebyshev Type 1 IIR
Chebyshev Type 2 IIR
Elliptic IIR
Pole/Zero Editor
```
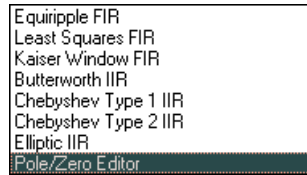
2 Enter the desired filter gain in the **Gain** edit box.

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.

Note Keep the Filter Visualization Tool (FVTool) open while designing a filter with the Pole/Zero Editor. Any changes that you make to the filter transfer function in the Pole/Zero Editor are then simultaneously reflected in the response plots of FVTool.

Positioning Poles and Zeros

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

Icon	Description
⬚	Enable moving poles or zeros by dragging on the plot
⬚	Add pole

Icon	Description
	Add zero
	Erase poles or zeros

You can move both members of a conjugate pair simultaneously by manipulating just one of the poles or zeros.

To ungroup conjugates, select the desired pair and clear **Conjugate pair** in the Specifications region on the Filter Designer.
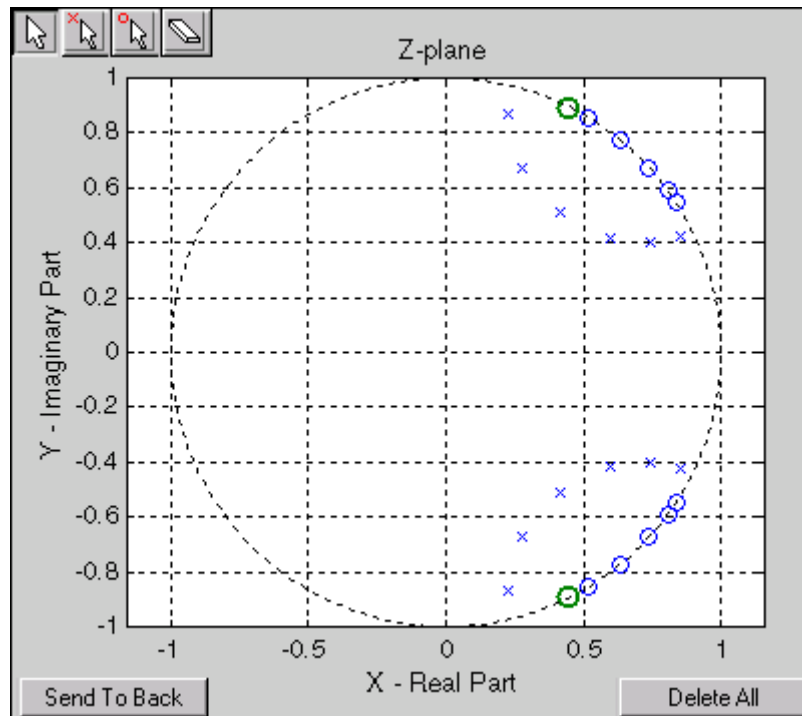
When you place two or more poles (or two or more zeros) directly on top of each other, a number is displayed next to the symbols (on the left for poles, and on the right for zeros) indicating the number of poles or zeros at that location (e.g., $\bigcirc 3$ for three zeros). This number makes it easy 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:

- **Fp** = 800
- **Fs** = 700
- **Rp** = 2.5
- **Rs** = 35

5 Select **Minimum Order** so the Filter Designer can calculate the lowest filter order that produces the desired characteristics.

6 Click **Apply** to compute the filter and update the response plot.

7 Position the cursor over the horizontal filter specification line for the stopband. This is the first (leftmost) horizontal specification line you see.

The cursor changes to the up/down drag indicator.

8 Drag the line until the **Rs** (stopband attenuation) field reads 100.

Note The **Order** value in the Measurements region changes because a higher filter order is needed to meet the new specifications.

9

Code Generation from MATLAB Support in Signal Processing Toolbox

Supported Functions

Code generation from MATLAB is a restricted subset of the MATLAB language that provides optimizations for:

- Generating efficient, production-quality C/C++ code and MEX files for deployment in desktop and embedded applications. For embedded targets, the subset restricts MATLAB semantics to meet the memory and data type requirements of the target environments.

For detailed information about code generation from MATLAB, refer to the "Code Generation from MATLAB" and "MATLAB Coder" documentation. Depending on which feature you wish to use, there are additional required products. For a comprehensive list, see "Which Code Generation Feature to Use".

Code generation from MATLAB supports Signal Processing Toolbox functions listed in the table. You must have the DSP System Toolbox software installed to use this feature. To generate C code, you must have the MATLAB® Coder™ software. If you have the Fixed-Point Toolbox, you can use `fiaccel` to generate MEX code for fixed-point applications.

To follow the examples in this documentation:

- To generate C/C++ code and MEX files with `codegen`, install the MATLAB Coder software, the Signal Processing Toolbox, the DSP System Toolbox, and a C compiler. For the Windows® platform, MATLAB supplies a default C compiler. Run `mex -setup` at the MATLAB command prompt to set up the C compiler.
- Change to a folder where you have write permission.

Note Many Signal Processing Toolbox functions require constant inputs in generated code. To specify a constant input for `codegen`, use `coder.Constant`. For more information, see the "MATLAB Coder" documentation.

Function	Remarks/Limitations
barthannwin	Window length must be a constant. Expressions or variables are allowed if their values do not change.
bartlett	Window length must be a constant. Expressions or variables are allowed if their values do not change.
besselap	Filter order must be a constant. Expressions or variables are allowed if their values do not change.
bitrevorder	—
blackman	Window length must be a constant. Expressions or variables are allowed if their values do not change.
blackmanharris	Window length must be a constant. Expressions or variables are allowed if their values do not change.
bohmanwin	Window length must be a constant. Expressions or variables are allowed if their values do not change.
buttap	Filter order must be a constant. Expressions or variables are allowed if their values do not change.
butter	Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.
buttord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cfirpm	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb1ap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb2ap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb1ord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb2ord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
chebwin	All inputs must be constants. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
cheby1	All Inputs must be constants. Expressions or variables are allowed if their values do not change.
cheby2	All inputs must be constants. Expressions or variables are allowed if their values do not change.
dct	Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.
downsample	—
dpss	All inputs must be constants. Expressions or variables are allowed if their values do not change.
ellip	Inputs must be constant. Expressions or variables are allowed if their values do not change.
ellipap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
ellipord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
filtfilt	Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.
fir1	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fir2	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls1	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firls	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firpm	All inputs must be constants. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
firpmord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firrcos	All inputs must be constants. Expressions or variables are allowed if their values do not change.
flattopwin	All inputs must be constants. Expressions or variables are allowed if their values do not change.
freqz	freqz with no output arguments produces a plot only when the function call terminates in a semicolon. See "freqz With No Output Arguments" on page 9-15.
gaussfir	All inputs must be constant. Expressions or variables are allowed if their values do not change.
gausswin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
hamming	All inputs must be constant. Expressions or variables are allowed if their values do not change.
hann	All inputs must be constant. Expressions or variables are allowed if their values do not change.
idct	Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.
intfilt	All inputs must be constant. Expressions or variables are allowed if their values do not change.
kaiser	All inputs must be constant. Expressions or variables are allowed if their values do not change.
kaiserord	—
levinson	If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.
maxflat	All inputs must be constant. Expressions or variables are allowed if their values do not change.
nuttallwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
parzenwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
rectwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
resample	The upsampling and downsampling factors must be specified as constants. Expressions or variables are allowed if their values do not change.
sgolay	All inputs must be constant. Expressions or variables are allowed if their values do not change.
sosfilt	—
taylorwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
triang	All inputs must be constant. Expressions or variables are allowed if their values do not change.
tukeywin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
upfirdn	• Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change. • Variable-size inputs are not supported.
upsample	Either declare input n as constant, or use the assert function in the calling function to set upper bounds for n. For example, assert(n<10)
xcorr	—
yulewalk	If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.

Specifying Inputs in Code Generation from MATLAB

In this section...

Defining Input Size and Type

When you use Signal Processing Toolbox functions for code generation, you must define the size and type of the function inputs. One way to do this is with the `-args` compilation option. The size and type of inputs must be defined because C is a statically typed language. See "Defining MATLAB Variables for C/C++ Code Generation" in the "Code Generation from MATLAB" documentation for details. To illustrate the need to define input size and type, consider the simplest call to `xcorr` requiring two input arguments. The following demonstrates the differences in the use of `xcorr` in MATLAB and in Code Generation from MATLAB.

Cross correlate two white noise vectors in MATLAB:

```
x = randn(512,1); %real valued white noise
y = randn(512,1); %real valued white noise
[C,lags] = xcorr(x,y);
x_circ = randn(256,1)+1j*randn(256,1); %circular white noise
y_circ = randn(256,1)+1j*randn(256,1); %circular white noise
[C1,lags1] = xcorr(x_circ,y_circ);
```

`xcorr` does not require any information about the size and type of the input arguments. `xcorr` obtains this information at runtime. Contrast this behavior with a MEX-file created with `codegen`. Create the file `myxcorr.m` in a folder where you have read and write permission. Ensure that this folder is in the MATLAB search path. Copy and paste the following two lines of code into `myxcorr.m` and save the file. The compiler tag `%#codegen` must be included in the file.

```
function [C,Lags]=myxcorr(x,y)  %#codegen
[C,Lags]=xcorr(x,y);
```

Enter the following command at the MATLAB command prompt:

```
codegen myxcorr -args {zeros(512,1),zeros(512,1)} -o myxcorr
```

Run the MEX-file:

```
x = randn(512,1); %real valued white noise
y = randn(512,1); %real valued white noise
[C,Lags] = myxcorr(x,y);
```

Define two new inputs x1 and y1 by transposing x and y.

```
x1 = x'; %x1 is 1x512
y1 = y'; %y1 is 1x512
```

Attempt to rerun the MEX-file with the tranposed inputs.

```
[C,Lags] = myxcorr(x1,y1); %Errors
```

The preceding program errors with the message ??? MATLAB expression
'x' is not of the correct size: expected [512x1] found [1x512]..

The error results because the inputs are specified to be 512x1 real-valued
column vectors at compilation. For complex-valued inputs, you must specify
that the input is complex valued. For example:

```
codegen myxcorr -o ComplexXcorr ...
-args {complex(zeros(512,1)),complex(zeros(512,1))}
```

Run the MEX-file at the MATLAB command prompt with complex-valued
inputs of the correct size:

```
x_circ = randn(512,1)+1j*randn(512,1); %circular white noise
y_circ = randn(512,1)+1j*randn(512,1); %circular white noise
[C,Lags] = ComplexXcorr(x_circ,y_circ);
```

Attempting to run ComplexXcorr with real valued inputs results in the error:
??? MATLAB expression 'x' is not of the correct complexness.

Inputs must be Constants

For a number of supported Signal Processing Toolbox functions, the inputs
or a subset of the inputs must be specified as constants at compilation
time. Functions with this behavior are noted in the right column of the

table "Supported Functions" on page 9-2. Use `coder.Type` with the `-args` compilation option, or enter the constants directly in the source code. For a detailed explanation see the "MATLAB Coder" documentation.

Specifying inputs as constants at compilation time results in significant advantages in the speed and efficiency of the generated code. For example, storing filter coefficients or window function values as vectors in the C source code improves performance by avoiding costly computation at runtime. Because a primary purpose of Code Generation from MATLAB is to generate optimized C code for desktop and embedded systems, emphasis is placed on providing the user with computational savings at runtime whenever possible.

To illustrate the constant input requirement with `butter`, create the file `myLowpassFilter.m` in a folder where you have read and write permission. Ensure that this folder is in the MATLAB search path. Copy and paste the following lines of code into `myLowpassFilter.m` and save the file.

```
function output = myLowpassFilter(input,N,Wn) %#codegen
[B,A] = butter(N,Wn,'low');
output = filter(B,A,input);
```

If you have the MATLAB Coder software, enter the following command at the MATLAB command prompt:

```
codegen myLowpassFilter  -o myLowpassFilter ...
-args {zeros(512,1), coder.newtype('constant',5),coder.newtype('constant
```

Once the program compiles successfully, the following message appears in the command window: `Code generation successful:  View report.`

Click on `View report`. Click on the `C code` tab on the top left and open the target source file `myLowpassFilter.c`.

Note that the numerator and denominator filter coefficients are included in the source code.

```
static real_T dv0[6] = { 5.9795780369978346E-5, 0.00029897890184989173
static real_T dv1[6] = { 1.0, -3.9845431196123373, 6.4348670902758709
```

Run the MEX-file without entering the constants:

```
output = myLowpassFilter(randn(512,1));
```

If you attempt to run the MEX-file by inputting the constants, you receive the error ??? Error using ==> myLowpassFilter 1 input required for entry-point 'myLowpassFilter'..

You may also enter the constants in the MATLAB source code directly. Edit the myLowPassFilter.m file and replace the MATLAB code with the lines:

```
function output = myLowpassFilter(input) %#codegen
[B,A] = butter(5,0.1,'low');
output = filter(B,A,input);
```

Enter the following command at the MATLAB command prompt:

```
codegen myLowpassFilter -args {zeros(512,1)} -o myLowpassFilter
```

Run the MEX-file by entering the following at the MATLAB command prompt:

```
output = myLowpassFilter(randn(512,1));
```

See "Apply Window to Input Signal" on page 9-11 ,"Apply Lowpass Filter to Input Signal" on page 9-13, and "Zero Phase Filtering" on page 9-16 for additional examples of the constant input requirement.

Code Generation Examples

Apply Window to Input Signal

In this example, apply a Hamming window to an input data vector of size 512x1.

Create a file called window_data.m by typing

```
>>edit window_data
```

at the MATLAB command prompt.

Copy and paste the code provided into the editor and save the file.

```
function output_data=window_data(input_data,N) %#codegen
Win=hamming(N);
output_data=input_data.*Win;
```

Use codegen to generate a MEX–file window_data.m.

```
codegen window_data -args {zeros(512,1),coder.newtype('constant',512)} -c
```

The -args option defines the input specifications for the MEX –file. input_data is a 512x1 real valued vector. Because the input to hamming must be a constant, coder.newtype is used to specify the window length. In a conventional MATLAB program, you can read the input data length at runtime and construct a Hamming window of the corresponding length.

Alternatively, edit the code for window_data.m as follows:

```
function output_data=window_data(input_data) %#codegen
Win=hamming(512);
output_data=input_data.*Win;
```

The preceding code specifies the length of the Hamming window in the source code as opposed to using `coder.newtype`. Use `codegen` to generate a MEX–file and C code:

```
codegen window_data -args {zeros(512,1)} -o window_data -report
```

The `-report` flag generates a compilation report. If the `codegen` operation is successful, you obtain: `Code generation successful:  View report.`

Click on `View report` to view the Code Generation Report.

Select the `C-code` tab and select `window_data.c` as the `Target Source File`.
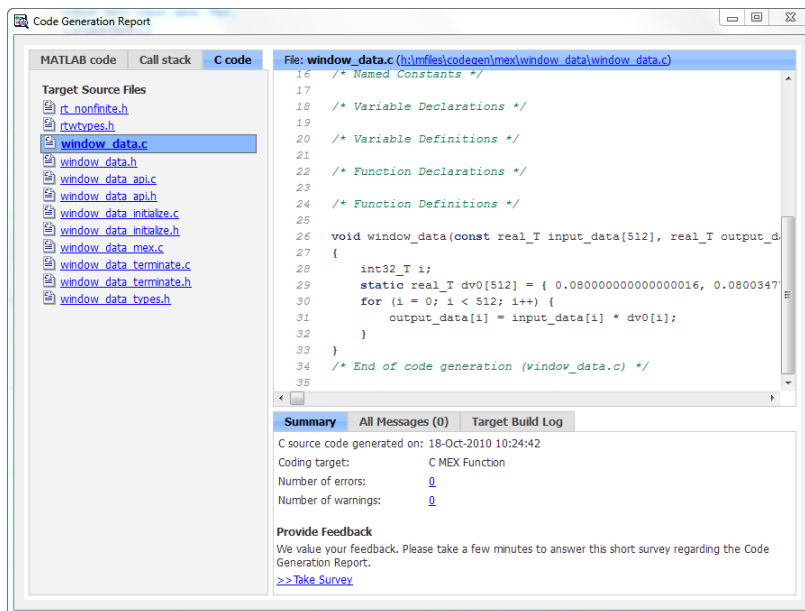
Note from the location bar that the C source code is in the `codegen/mex/<FUNCTION_NAME>` folder. Running `codegen` creates this folder

and places the C source code, C header files, and MEX files in the folder. Each function that you create produces a codegen/mex/<FUNCTION_NAME> folder.

Scroll through the C code to see that the values of the Hamming window are included directly in the C source code.

Run the MEX-file on a white noise input:

```
% Window white noise input
output_data=window_data(randn(512,1));
```

Apply Lowpass Filter to Input Signal

Assuming a sampling frequency of 20 kHz, create a 4–th order Butterworth filter with a 3–dB frequency of 2.5 kHz. Use the Butterworth filter to lowpass filter a 10000x1 input data vector.

Create a file called ButterFilt.m. Copy and paste the following code into the file.

```
function output_data=ButterFilt(input_data) %#codegen
[b,a]=butter(4,0.25);
output_data=filter(b,a,input_data);
```

Run the codegen command to obtain the C source code ButterFilt.c and MEX file:

```
codegen ButterFilt -args {zeros(10000,1)} -o ButterFilt -report
```

The C source code includes the five numerator and denominator coefficients of the 4–th order Butterworth filter as constants.

```
static real_T dv0[5] = { 0.010209480791203124, 0.040837923164812495, 0.06
static real_T dv1[5] = { 1.0, -1.9684277869385174, 1.7358607092088851, -0
```

Apply the filter using the MEX-file:

```
Fs=20000;
%Create 10000x1 input signal
t=0:(1/Fs):0.5-(1/Fs);
input_data=(cos(2*pi*1000*t)+sin(2*pi*500*t)+0.2*randn(size(t)))';
%Filter data
```

```
output_data=ButterFilt(input_data);
```

Cross Correlate or Autocorrelate Input Data

Estimate the cross correlation or autocorrelation of two real-valued input vectors to lag 50. Output the estimate at the nonnegative lags.

Create a file called myxcorr.m. Copy and paste the following code into the file:

```
function [C,Lags]=myxcorr(x,y) %#codegen
[c,lags]=xcorr(x,y,50,'coeff');
C=c(51:end);
Lags=lags(51:end);
```

Run the codegen command at the MATLAB command prompt:

```
codegen myxcorr -args {zeros(512,1), zeros(512,1)} -o myxcorr -report
```

Use the MEX-file to compute and plot the autocorrelation of a white noise input:

```
rng(0,'twister')
%White noise input
input_data=randn(512,1);
%Compute autocorrelation with MEX-file
[C,Lags]=myxcorr(input_data,input_data);
% Plot the result
stem(Lags,C); axis([-0.5 51 -1.1 1.1])
xlabel('Lags'); ylabel('Autocorrelation Function');
```
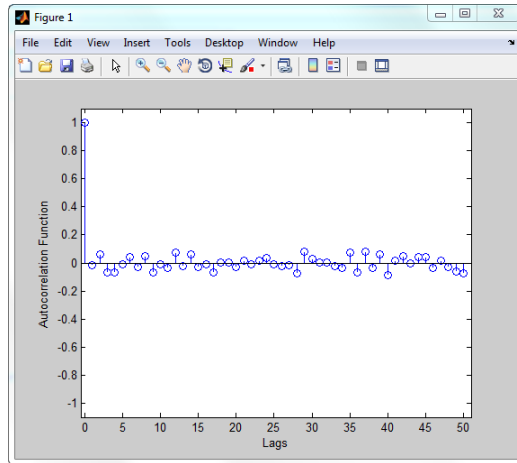
`freqz` **With No Output Arguments**

In Code Generation from MATLAB, `freqz` with no output arguments behaves differently than in the standard MATLAB language. In standard MATLAB, `freqz` with no output arguments produces a plot of the magnitude and phase response of the input filter. The plot is produced regardless of whether the call to `freqz` terminates in a semicolon or not. No frequency response or phase vectors are returned.

`freqz` with no output arguments and no terminating semicolon:

```
B = [0.05 0.9 0.05]; %Numerator coefficients
freqz(B,1) %no semicolon. Plot is produced
```

`freqz` with no output arguments and terminating in a semicolon:

```
B = [0.05 0.9 0.05]; %Numerator coefficients
freqz(B,1); %semicolon. Plot is produced
```

The behavior shown in the preceding examples differs from the expected behavior of a MEX-file using `freqz` with code generation support. To illustrate this difference create a program called `myfreqz.m`.

Copy and paste the following code into the file:

```
function myfreqz(B,A) %#codegen
freqz(B,A)
```

Run the following command at the MATLAB command prompt:

```
codegen myfreqz -args {zeros(1,3), zeros(1,1)} -o myfreqz
```

Calling the MEX-file writes a 512x1 complex-valued vector to the workspace and displays the output. The vector is the frequency response. No plot is produced.

```
myfreqz([0.05 0.9 0.05],1);
```

Change the code in myfreqz.m by adding a terminating semicolon:

```
function myfreqz(B,A) %#codegen
freqz(B,A);
```

Run the following command at the MATLAB command prompt:

```
codegen myfreqz -args {zeros(1,3), zeros(1,1)} -o myfreqz
```

Calling the MEX-file produces a plot of the magnitude and phase response of the filter. The output of the complex-valued frequency response is suppressed.

```
myfreqz([0.05 0.9 0.05],1);
```

Zero Phase Filtering

Design a lowpass Butterworth filter with a 1 kHz 3–dB frequency to implement zero phase filtering on data with a sampling frequency of 20 kHz.

```
[B,A] = butter(20,0.314,'low');
```

Create the program myZerophaseFilt.m.

```
function output  = myZerophaseFilt(input) %#codegen
B=1e-3 *[
    0.0000
    0.0001
    0.0010
    0.0060
```

```
                        0.0254
                        0.0814
                        0.2035
                        0.4071
                        0.6615
                        0.8820
                        0.9702
                        0.8820
                        0.6615
                        0.4071
                        0.2035
                        0.0814
                        0.0254
                        0.0060
                        0.0010
                        0.0001
                        0.0000];
                  A=[1.0000
                       -7.4340
                       28.2476
                      -71.6333
                      134.6222
                     -197.9575
                      235.1628
                     -230.2286
                      188.0901
                     -129.1746
                       74.8284
                      -36.5623
                       15.0197
                       -5.1525
                        1.4599
                       -0.3361
                        0.0613
                       -0.0085
                        0.0009
                       -0.0001
                        0.0000];
                  output = filtfilt(B,A,input);
```

Run the following command at the MATLAB command prompt:

```
codegen myZerophaseFilt -args {zeros(1,20001)} -o myZerophaseFilt
```

Filter input data with `myZerophaseFilt`:

```
Fs = 20000;
t = 0:(1/Fs):1;
Comp500Hz = cos(2*pi*500*t);
Signal = Comp500Hz+sin(2*pi*4000*t)+0.2*randn(size(t));
FilteredData = myZerophaseFilt(Signal);
plot(t(1:500).*1000,Comp500Hz(1:500));
xlabel('msec'); ylabel('Amplitude');
axis([0 25 -1.8 1.8]); hold on;
plot(t(1:500).*1000,FilteredData(1:500),'r','linewidth',2);
legend('500 Hz component','Zero phase lowpass filtered data',...
'Location','NorthWest');
```
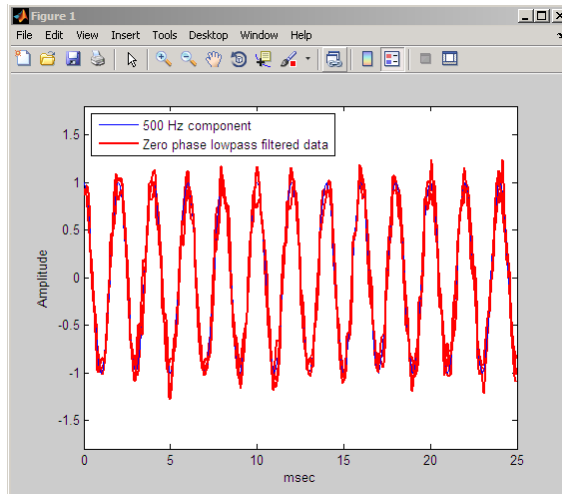
Technical Conventions

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

Term or Symbol	Description
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.

Index