Signal Processing Toolbox™
User's Guide
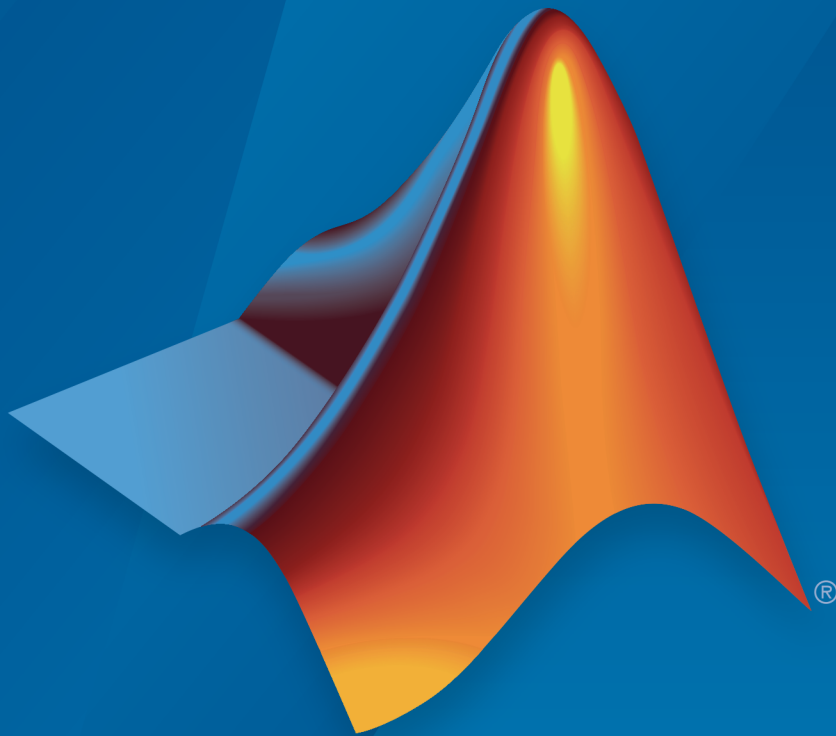
MATLAB®

MathWorks®

How to Contact MathWorks

	Latest news:	www.mathworks.com
	Sales and services:	www.mathworks.com/sales_and_services
	User community:	www.mathworks.com/matlabcentral
	Technical support:	www.mathworks.com/support/contact_us
	Phone:	508-647-7000

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

Revision History

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

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

Convolution and Correlation

10

Multirate Signal Processing

11

Spectral Analysis

12

Linear Prediction

13

Transforms

14

Signal Generation

15

Signal Measurement

16

Spectrum Object to Function Replacement

17

Common Applications

18

Technical Conventions

A

1

Filtering, Linear Systems and Transforms Overview

Filter Implementation and Analysis

In this section...
"Filtering Overview" on page 1-2
"Convolution and Filtering" on page 1-2
"Filters and Transfer Functions" on page 1-3
"Filtering with the filter Function" on page 1-4

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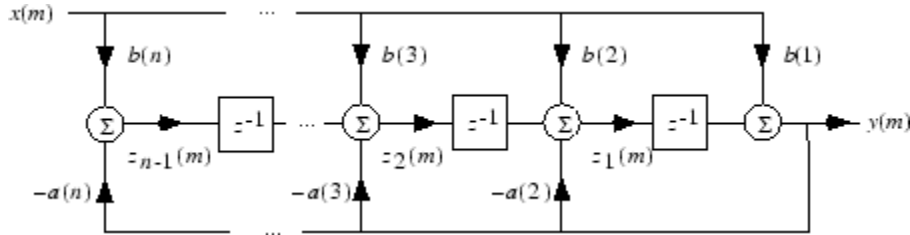
At sample m, `filter` computes the difference equations

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

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.

Multirate Filter Bank Implementation

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

- Upsampling (zero insertion) by integer factor `p`
- Filtering by FIR filter `h`
- Downsampling by integer factor `q`
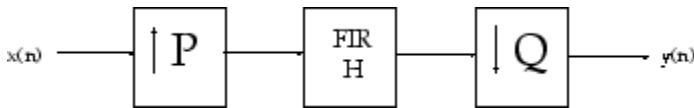
For example, to change the sample rate of a signal from 44.1 kHz to 48 kHz, we first find the smallest integer conversion ratio `p/q`. Set

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

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

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

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

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

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

Frequency Domain Filter Implementation

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

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

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

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

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

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

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

Anti-Causal, Zero-Phase Filter Implementation

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

To see how `filtfilt` does this, recall that if the z-transform of a real sequence $x(n)$ is $X(z)$, the z-transform of the time reversed sequence $x(n)$ is $X(1/z)$. Consider the processing scheme.
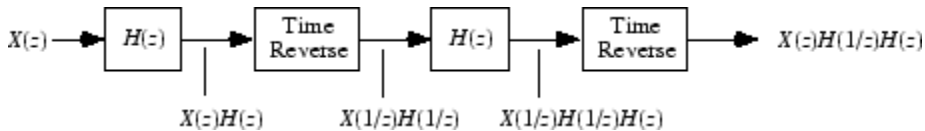
Image of Anti Causal Zero Phase Filter

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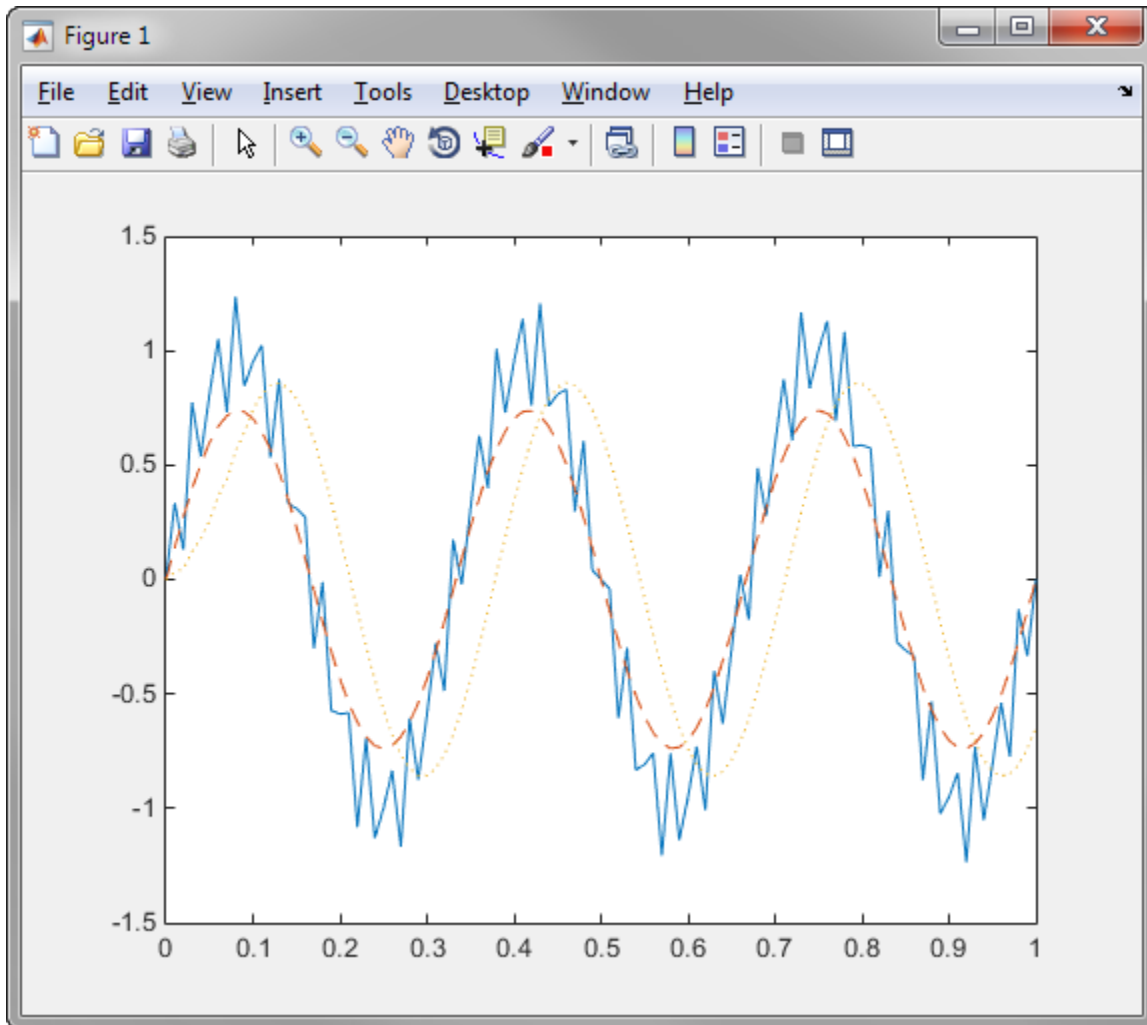
Both filtered versions eliminate the 40 Hz sinusoid evident in the original, solid line. The plot also shows how `filter` and `filtfilt` differ; the dashed (`filtfilt`) line is in phase with the original 3 Hz sinusoid, while the dotted (`filter`) line is delayed by about five samples. Also, the amplitude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

Impulse Response

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

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

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

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

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

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

h = filter(b,a,imp);

stem(0:49,h)
```
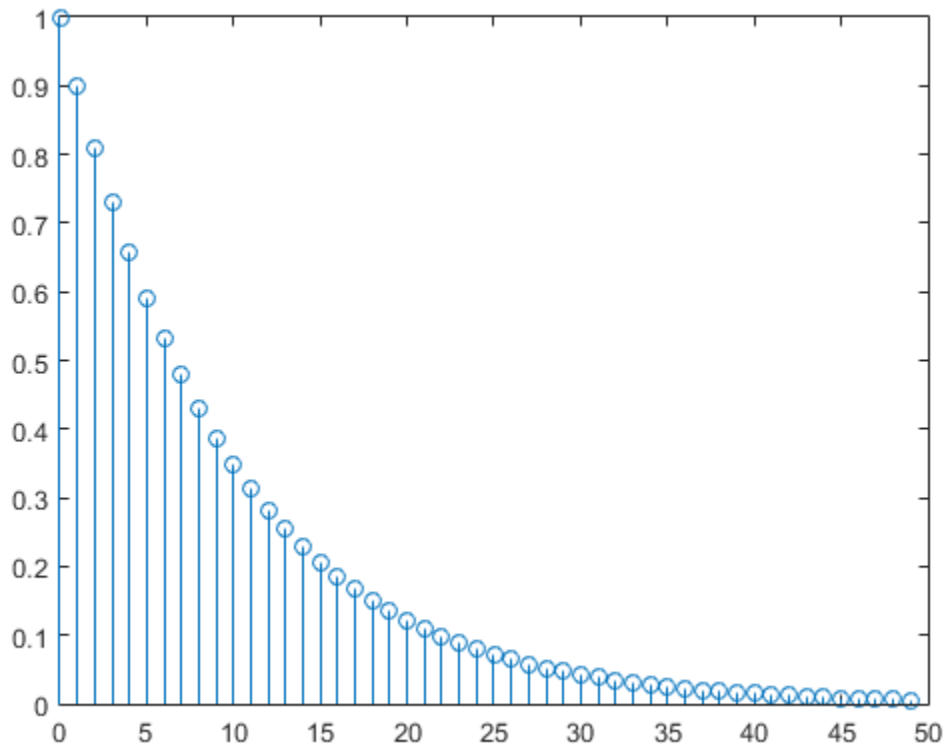
A simple way to display the impulse response is with the Filter Visualization Tool, `fvtool`.

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

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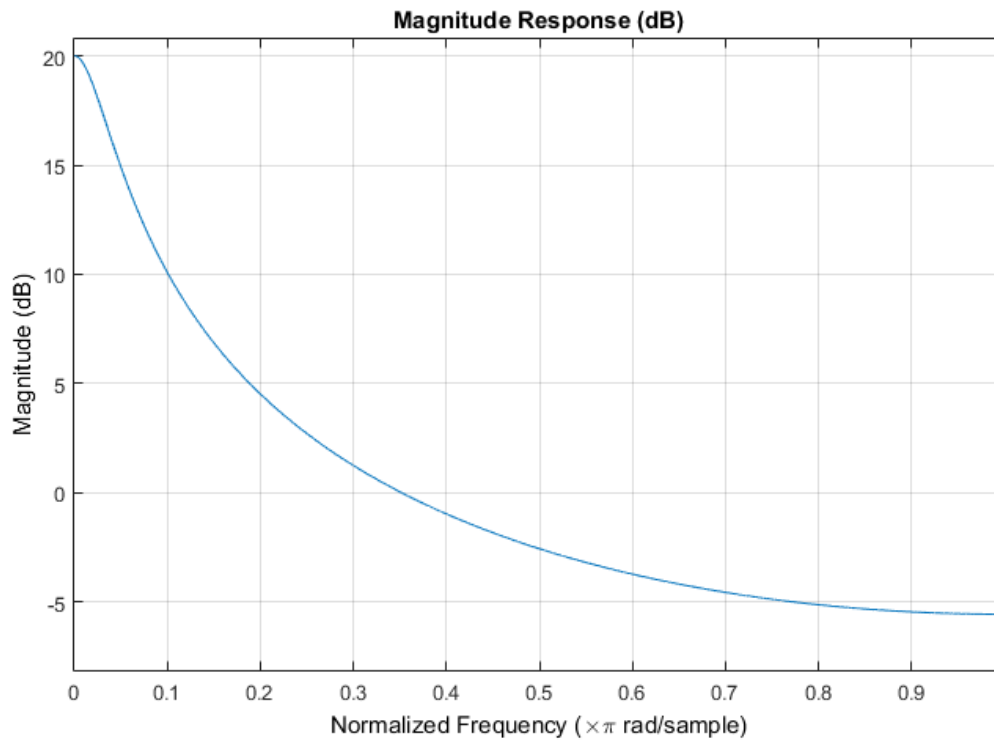
Click the **Impulse Response** button, [↑], on the toolbar, select **Analysis > Impulse Response** from the menu, or type the following code to obtain the exponential decay of the single-pole system.

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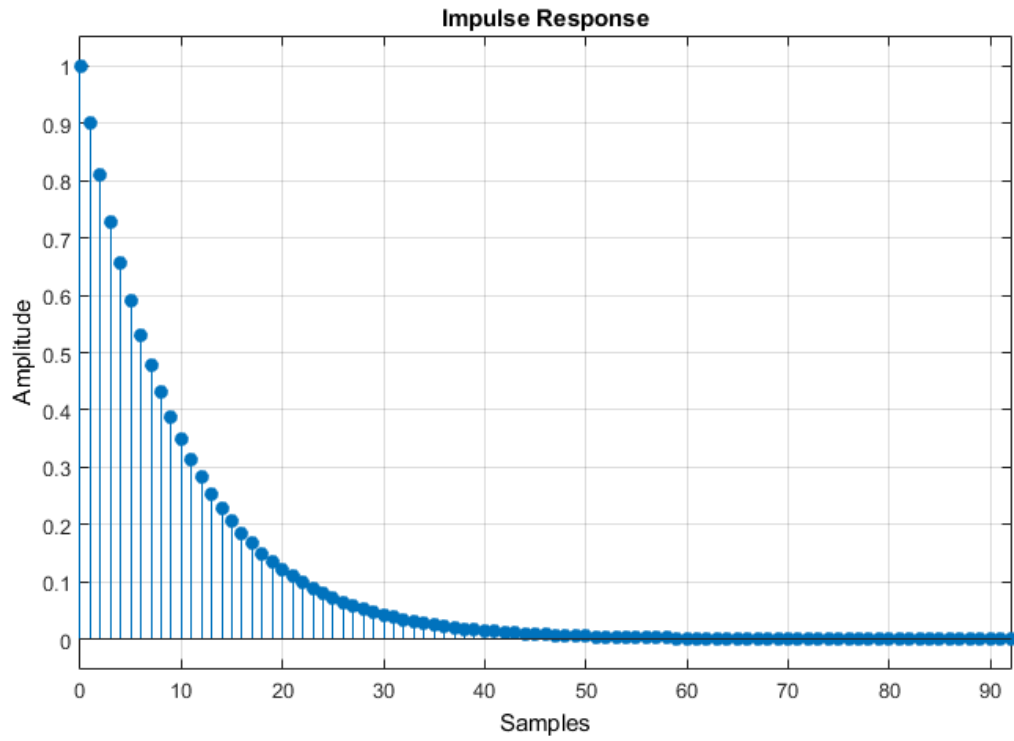
Frequency Response

In this section...
"Digital Domain" on page 1-17
"Analog Domain" on page 1-19

Digital Domain

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

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

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

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

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

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

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

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

Note This toolbox uses the convention that unit frequency is the Nyquist frequency, defined as half the sampling frequency. The cutoff frequency parameter for all basic filter design functions is normalized by the Nyquist frequency. For a system with a

1000 Hz sampling frequency, for example, 300 Hz is 300/500 = 0.6. To convert normalized frequency to angular frequency around the unit circle, multiply by 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)
```
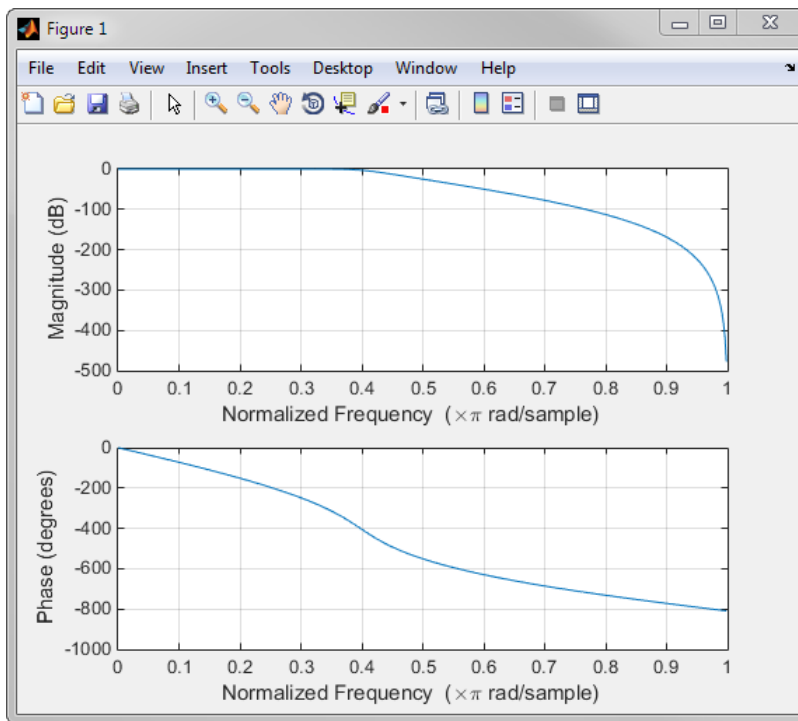

`freqz` can also accept a vector of arbitrary frequency points for use in the frequency response calculation. For example,

```
w = linspace(O,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:

```
[z,p,k] = butter(9,400/1000);
fvtool(zp2sos(z,p,k))
```

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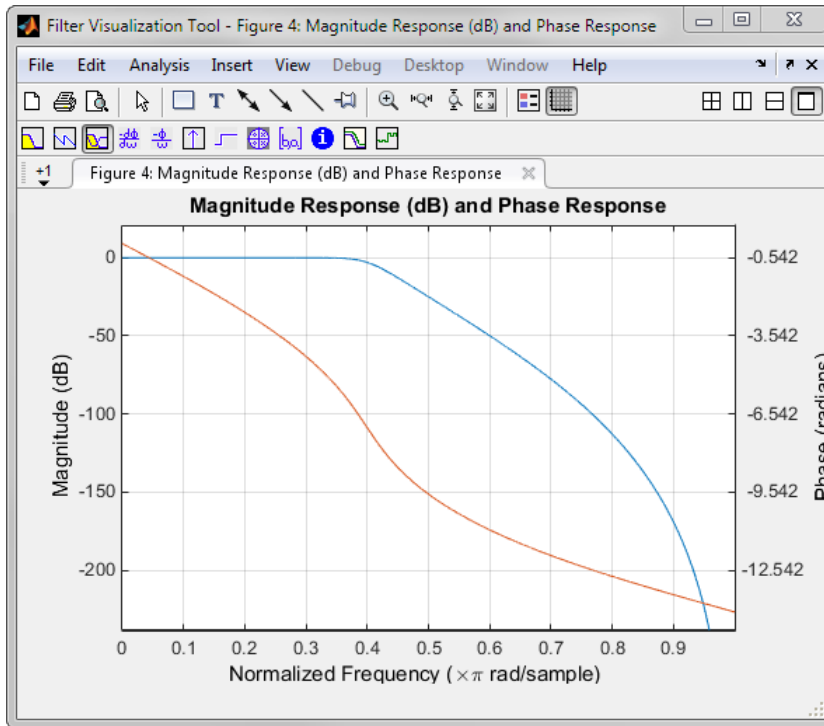
The unwrap function is also useful in frequency analysis. unwrap unwraps the phase to make it continuous across 360° phase discontinuities by adding multiples of ±360°, as needed. To see how unwrap is useful, design a 25th-order lowpass FIR filter:

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

Obtain the 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
```
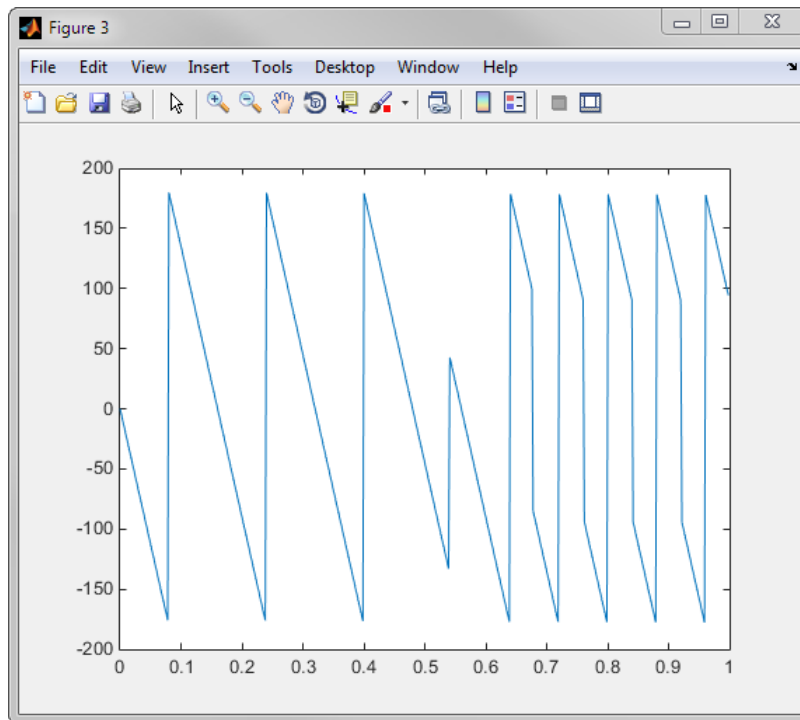

It is difficult to distinguish the 360° jumps (an artifact of the arctangent function inside `angle`) from the 180° jumps that signify zeros in the frequency response.

`unwrap` eliminates the 360° jumps:

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

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

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

1-21
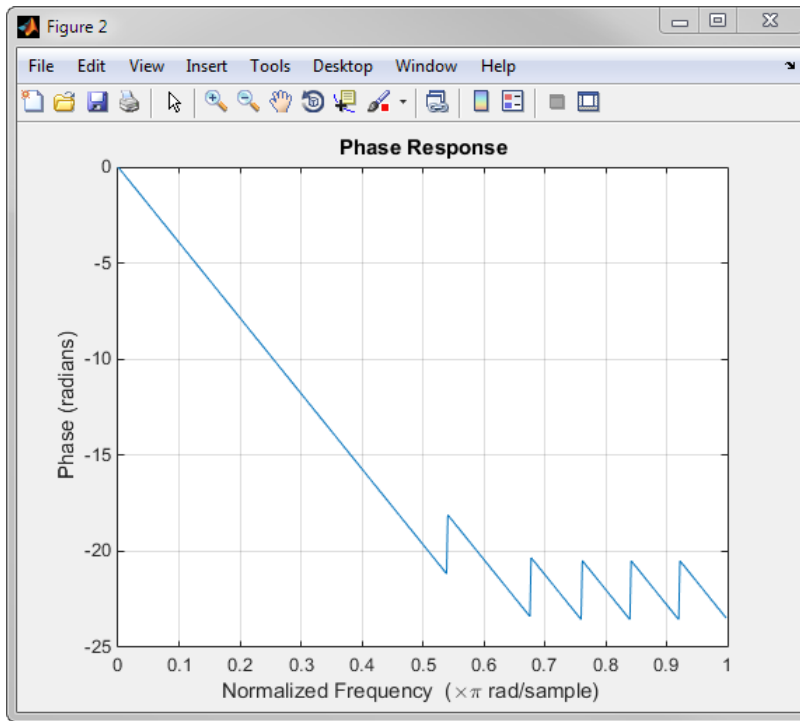
Delay

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

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

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

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

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

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

```
[z,p,k] = butter(10,200/1000);
fvtool(zp2sos(z,p,k),'Analysis','grpdelay', ...
    'OverlayedAnalysis','phasedelay','Legend','on')
```
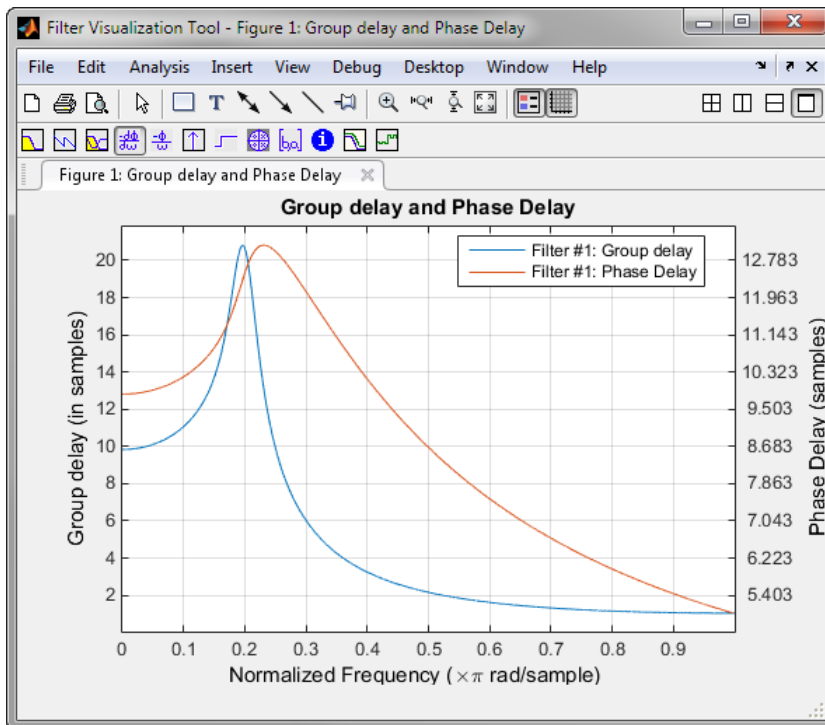
Zero-Pole Analysis

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

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

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

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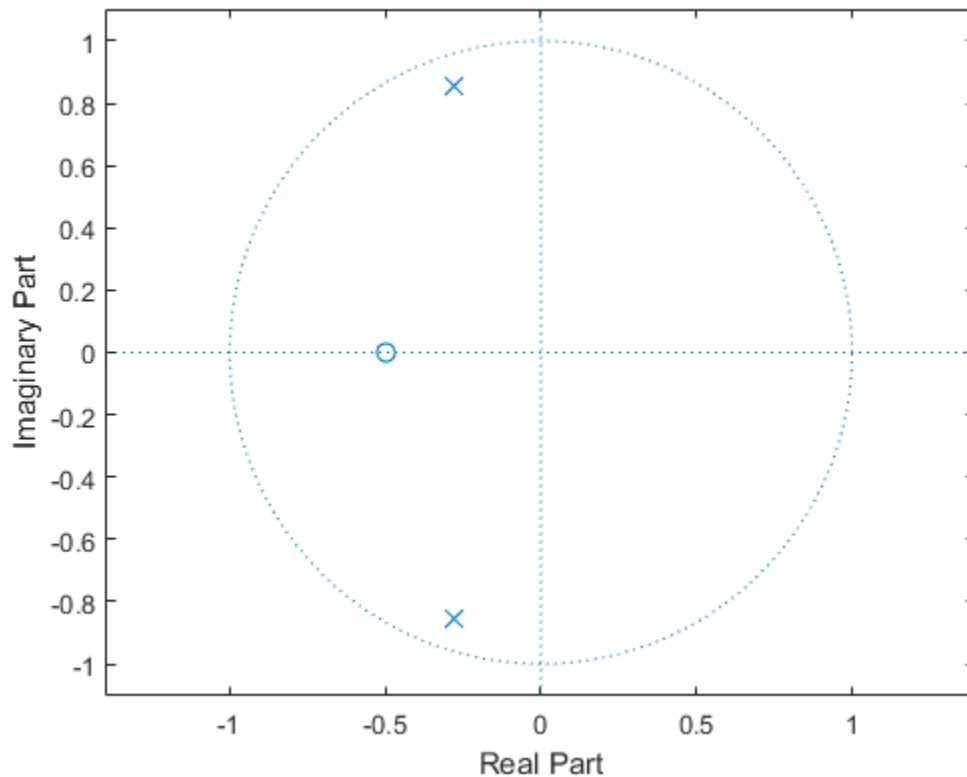
For access to additional tools, use `fvtool`. First convert the poles and zeros to transfer function form, then call `fvtool`.

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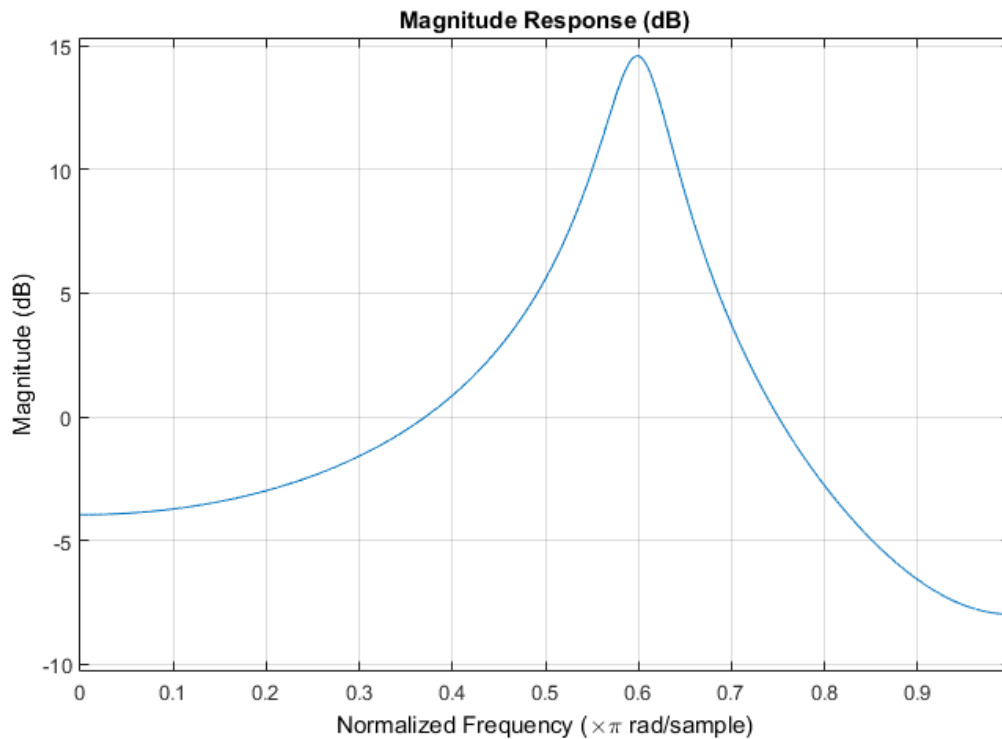

Click the **Pole/Zero Plot** toolbar button, select **Analysis** > **Pole/Zero Plot** from the menu, or type the following code to see the plot.

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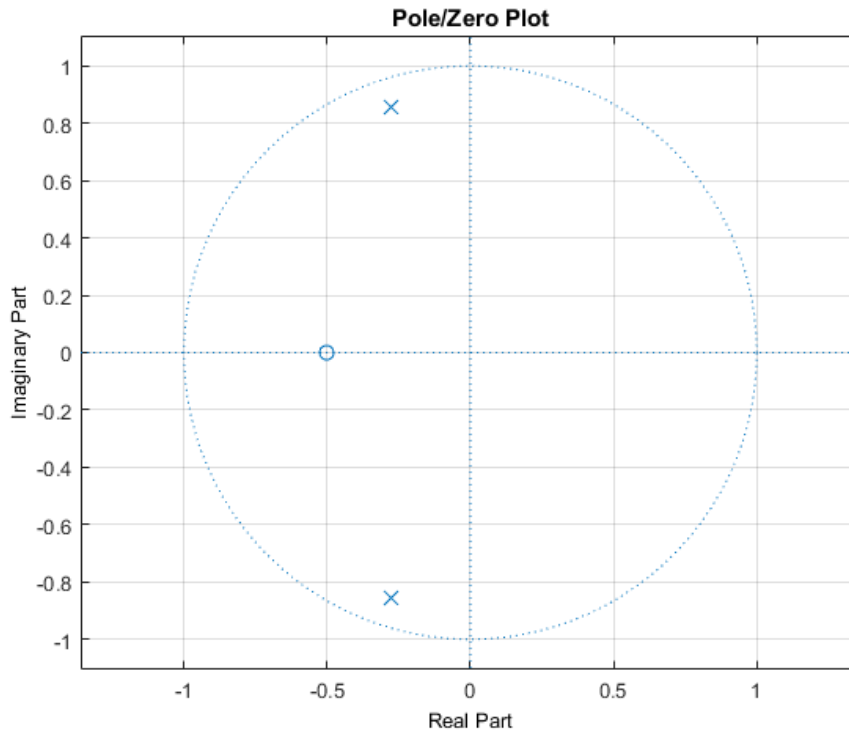
To use `zplane` for a system in transfer function form, supply row vector arguments. In this case, `zplane` finds the roots of the numerator and denominator using the `roots` function and plots the resulting zeros and poles.

```
zplane(b,a)
```
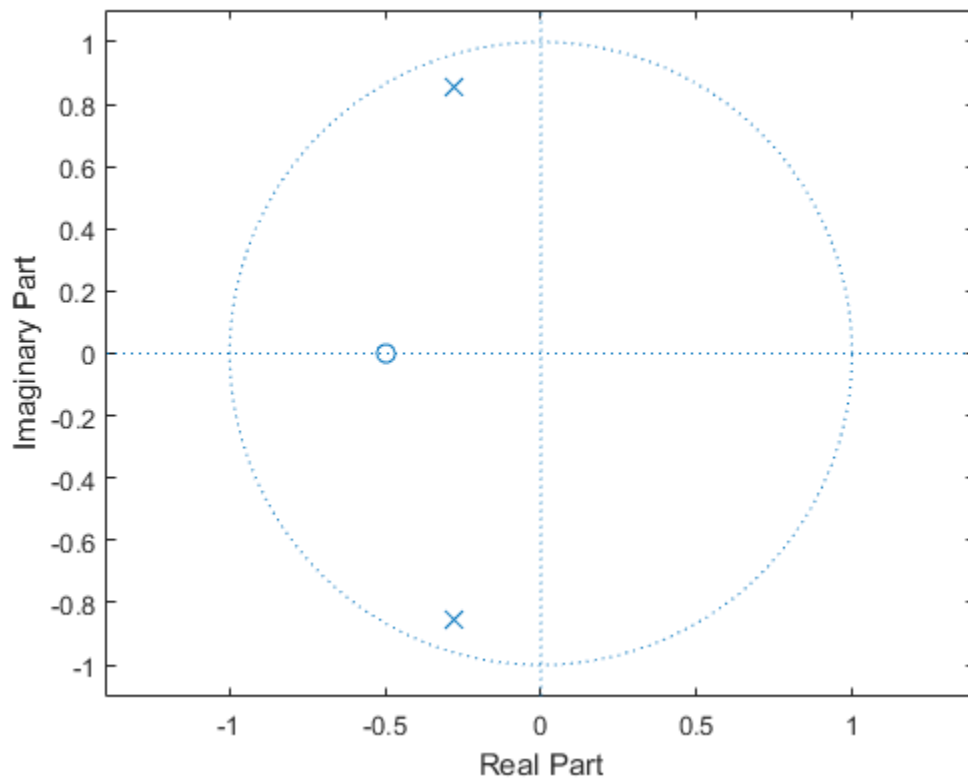
See Linear System Models for details on zero-pole and transfer function representation of systems.

Linear System Models

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 1-30
- "Zero-Pole-Gain" on page 1-30
- "State Space" on page 1-31
- "Partial Fraction Expansion (Residue Form)" on page 1-32
- "Second-Order Sections (SOS)" on page 1-33
- "Lattice Structure" on page 1-34
- "Convolution Matrix" on page 1-36

Transfer Function

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

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

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

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

There are many ways to represent a filter in second-order section form. Through careful pairing of the pole and zero pairs, ordering of the sections in the cascade, and multiplicative scaling of the sections, it is possible to reduce quantization noise gain and avoid overflow in some fixed-point filter implementations. The functions zp2sos and ss2sos, described in "Linear System Transformations" on page 1-40, 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
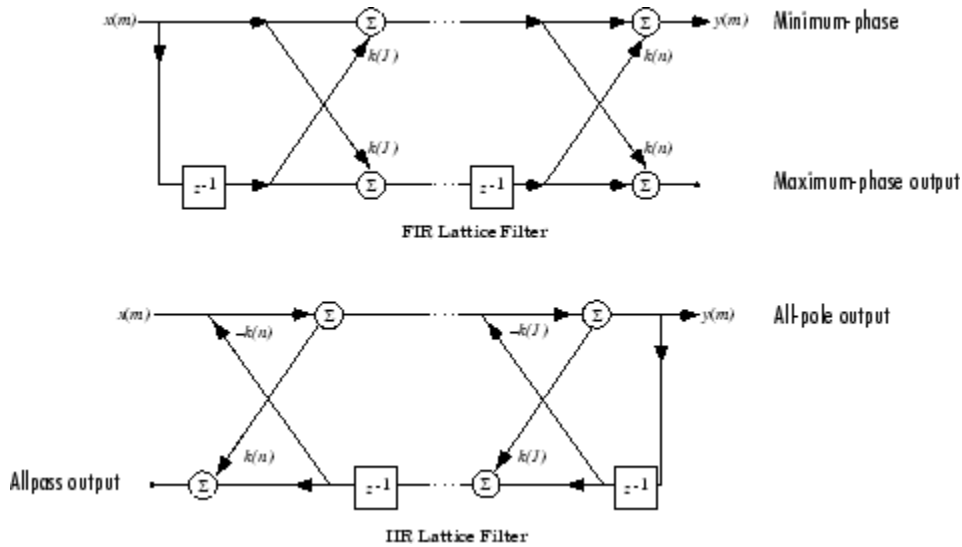
IIR Lattice Filter

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
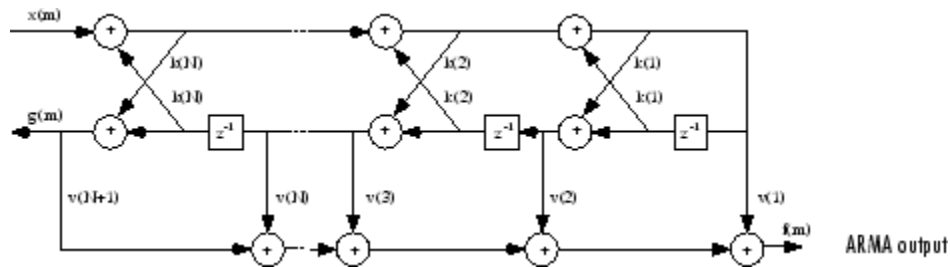
Diagram of lattice/ladder filter

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, 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.. 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
Zero-Pole-Gain	`zp2tf` `poly`	`zp2ss`		none	none	`zp2sos`	none
Partial Fraction	`residuez`	none	none		none	none	none
Lattice Filter	`latc2tf`	none	none	none		none	none
SOS	`sos2tf`	`sos2ss`	`sos2zp`	none	none		none

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

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

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

Discrete Fourier Transform

The discrete Fourier transform, or DFT, is the primary tool of digital signal processing. The foundation of the 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);                            % Magnitude
p = unwrap(angle(y));                  % Phase
```

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

```
f = (0:length(y)-1)*100/length(y);     % Frequency vector
subplot(2,1,1)
plot(f,m)
title('Magnitude')
ax = gca;
ax.XTick = [15 40 60 85];
subplot(2,1,2)
plot(f,p*180/pi)
title('Phase')
ax = gca;
ax.XTick = [15 40 60 85];
```
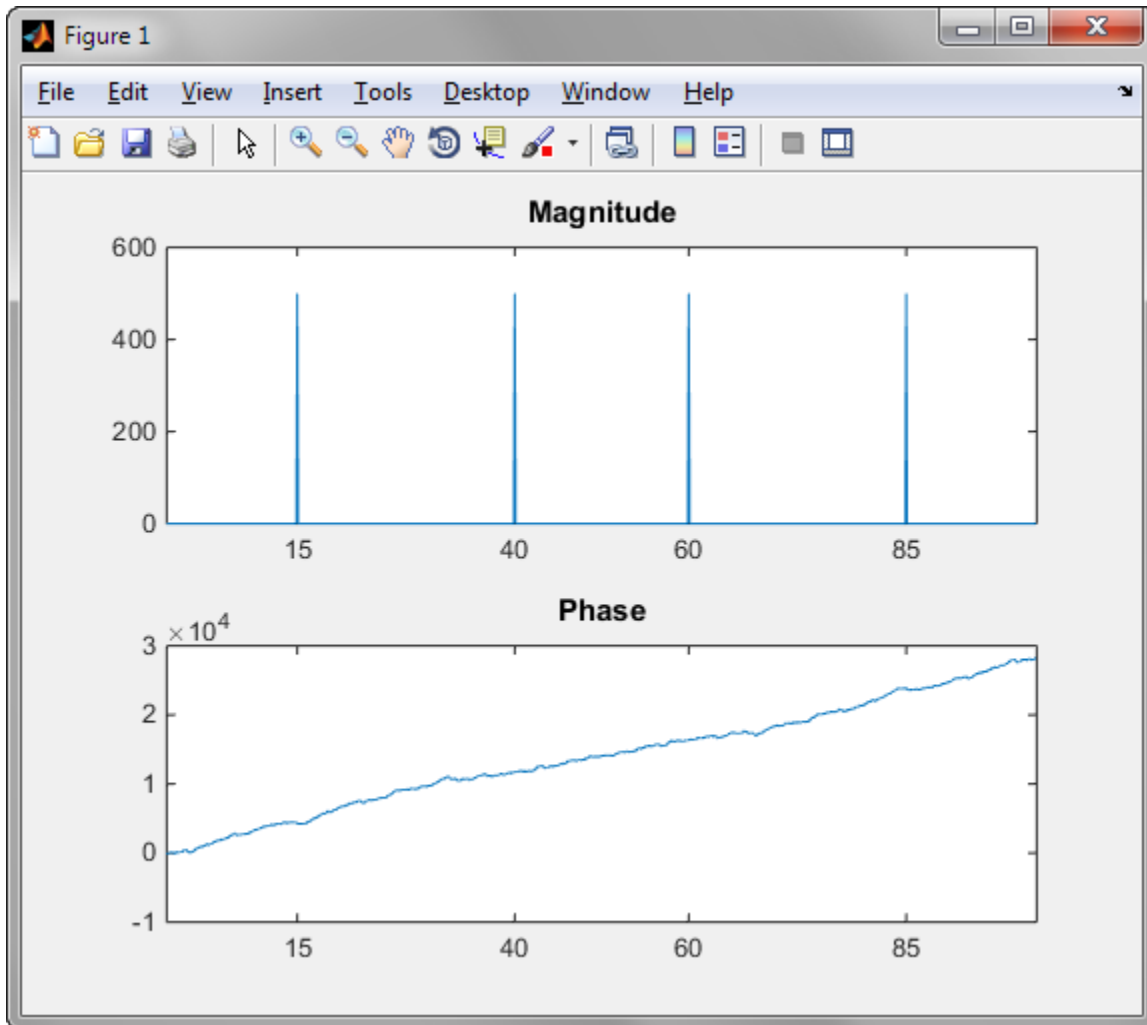
A second argument to `fft` specifies a number of points `n` for the transform, representing DFT length:

```
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
x = cos(2*pi*60*time)+sin(2*pi*120*time)+randn(size(time));   % Data vector
d=fdesign.lowpass('N,F3dB',5,200,Fs); % Lowpass filter specification object
Hd=design(d,'butter');                % Invoke Butterworth design method
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 7-17.

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`
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	Time-domain modeling functions: `lpc`, `prony`, `stmcb` Frequency-domain modeling functions: `invfreqs`, `invfreqz`

Filter Method	Description	Filter Functions
	extensive collection of parametric modeling tools.)	

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 2-39 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`*`)`

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`*`)`

Filter Type	Design Function

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(5,0.4,'s');     % Analog Butterworth filter
```

All filter design functions return a filter in the transfer function, zero-pole-gain, or state-space linear system model representation, depending on how many output arguments are present. In general, you should avoid using the transfer function form because numerical problems caused by 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 to analyze or implement your filter.

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

Designing IIR Filters to Frequency Domain Specifications

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

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

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

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

n =
    12
Wn =
    0.1951    0.4080
```

An elliptic filter that meets the same requirements is given by

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

n =
    5
Wn =
    0.2000    0.4000
```

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

Comparison of Classical IIR Filter Types

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

Butterworth Filter

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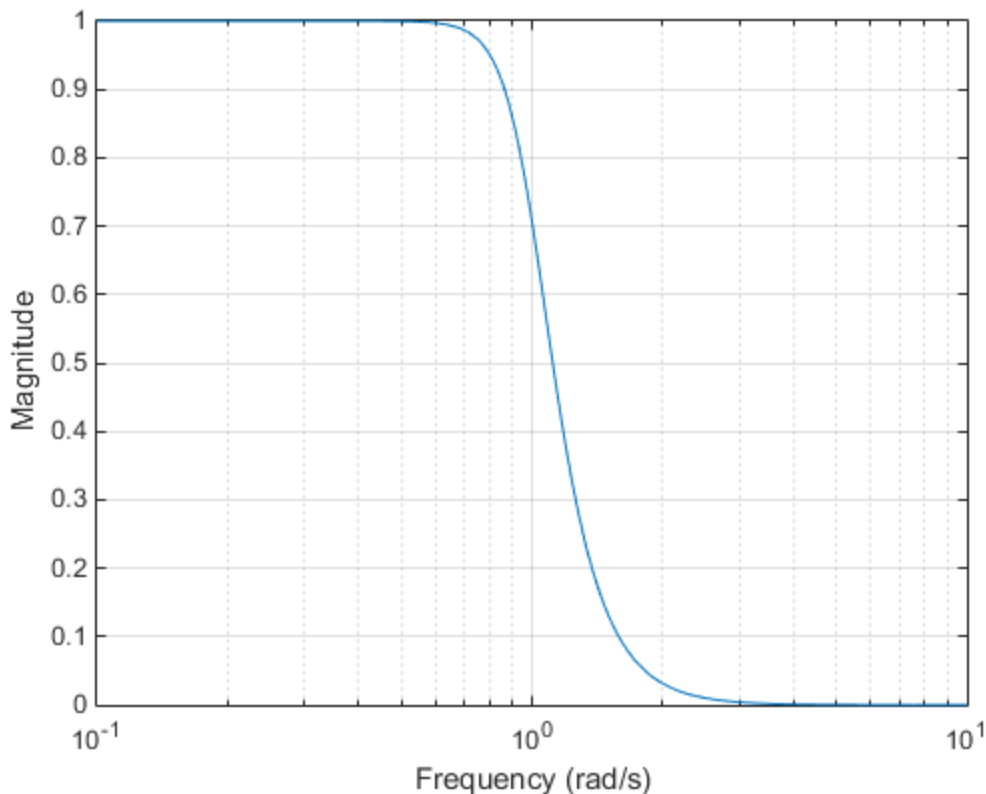
Chebyshev Type I Filter

The Chebyshev Type I filter minimizes the absolute difference between the ideal and actual frequency response over the entire passband by incorporating an equal ripple of Rp dB in the passband. Stopband response is maximally flat. The transition from

passband to stopband is more rapid than for the Butterworth filter. $|H(j\Omega)| = 10^{-Rp/20}$ at $\Omega = 1$.
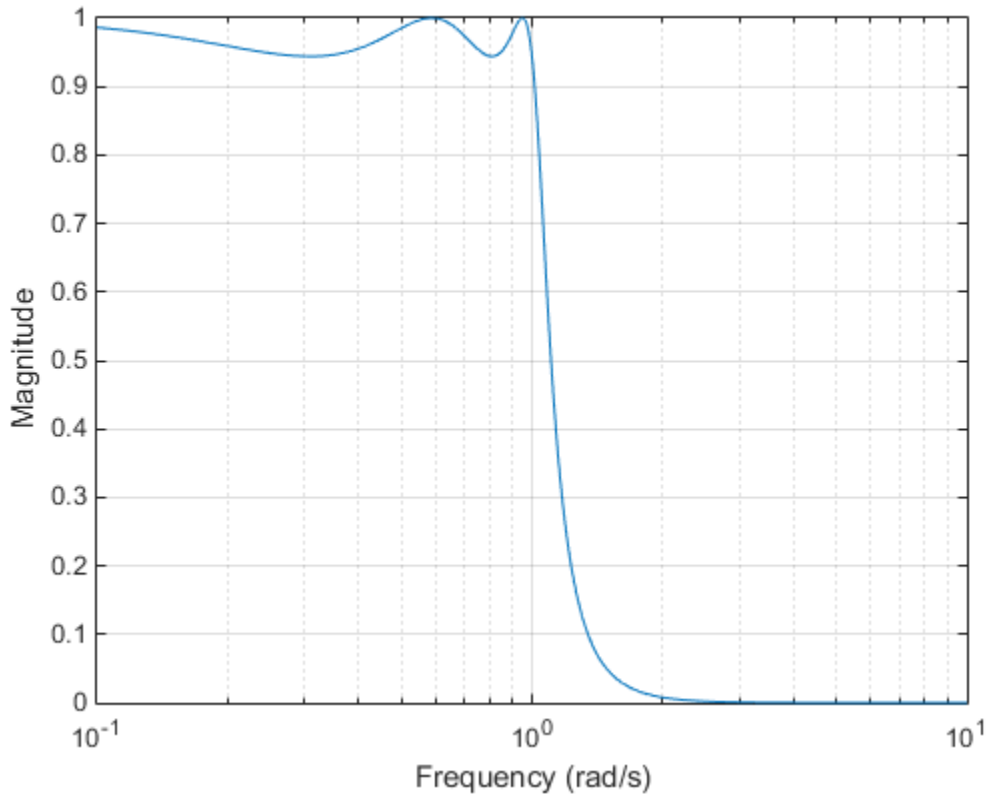
Chebyshev Type II Filter

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 $\Omega = 1$.
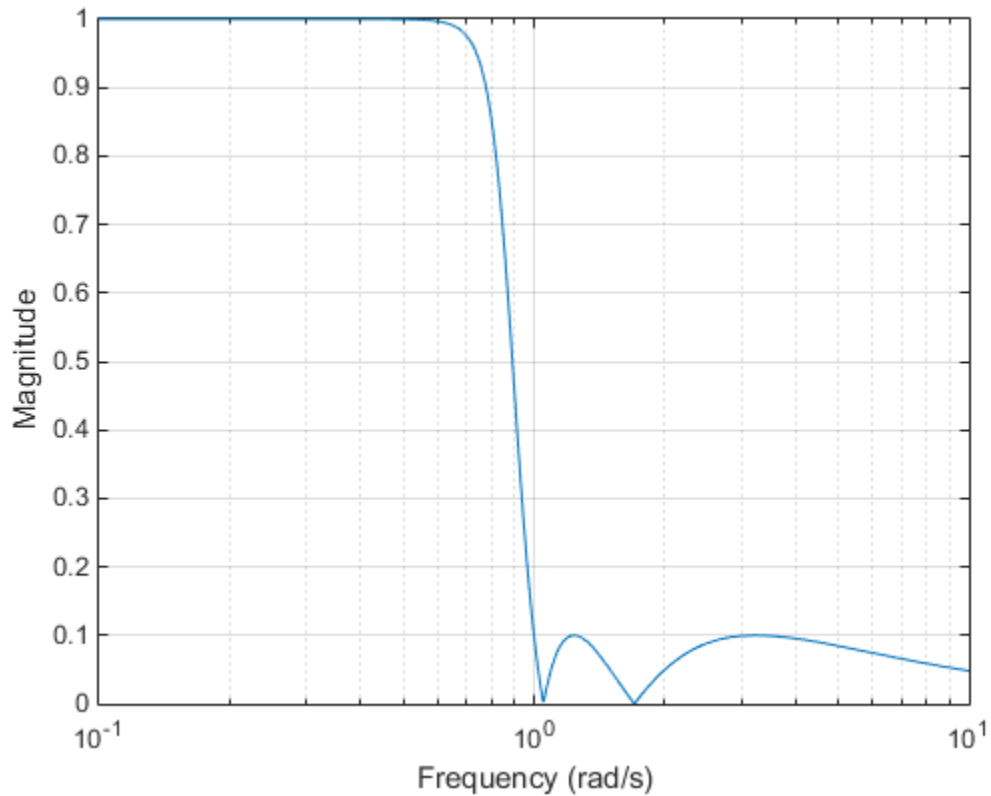
Elliptic Filter

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. $|H(j\Omega)| = 10^{-\text{Rp}/20}$ at $\Omega = 1$.
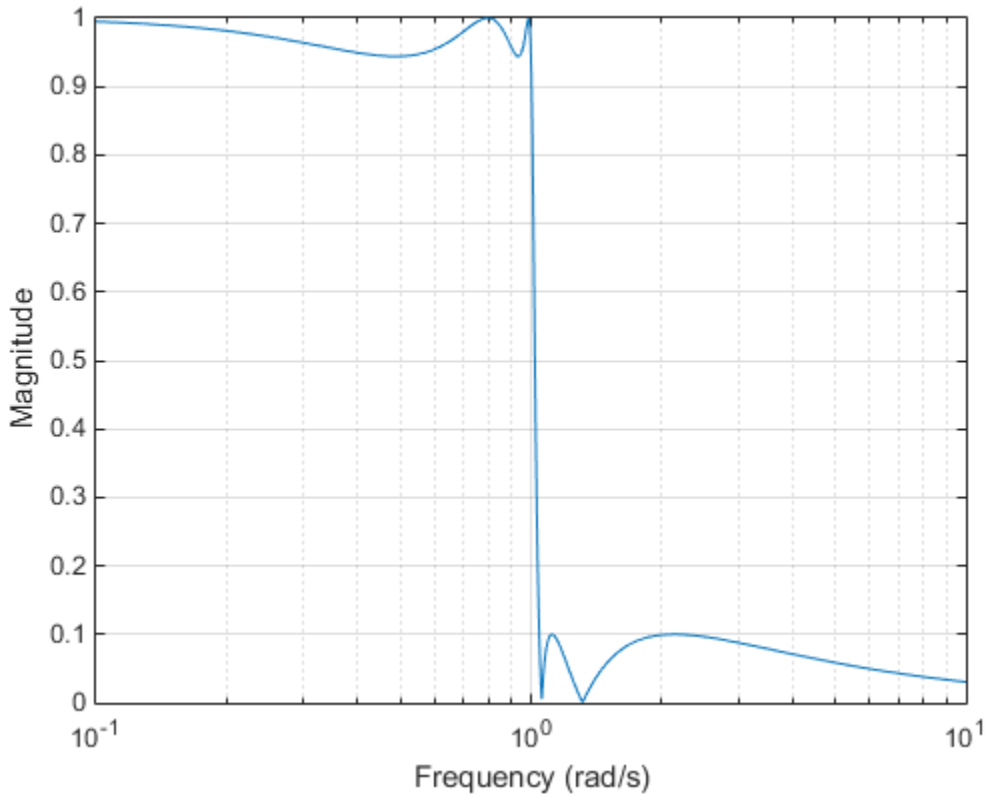
Bessel Filter

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.
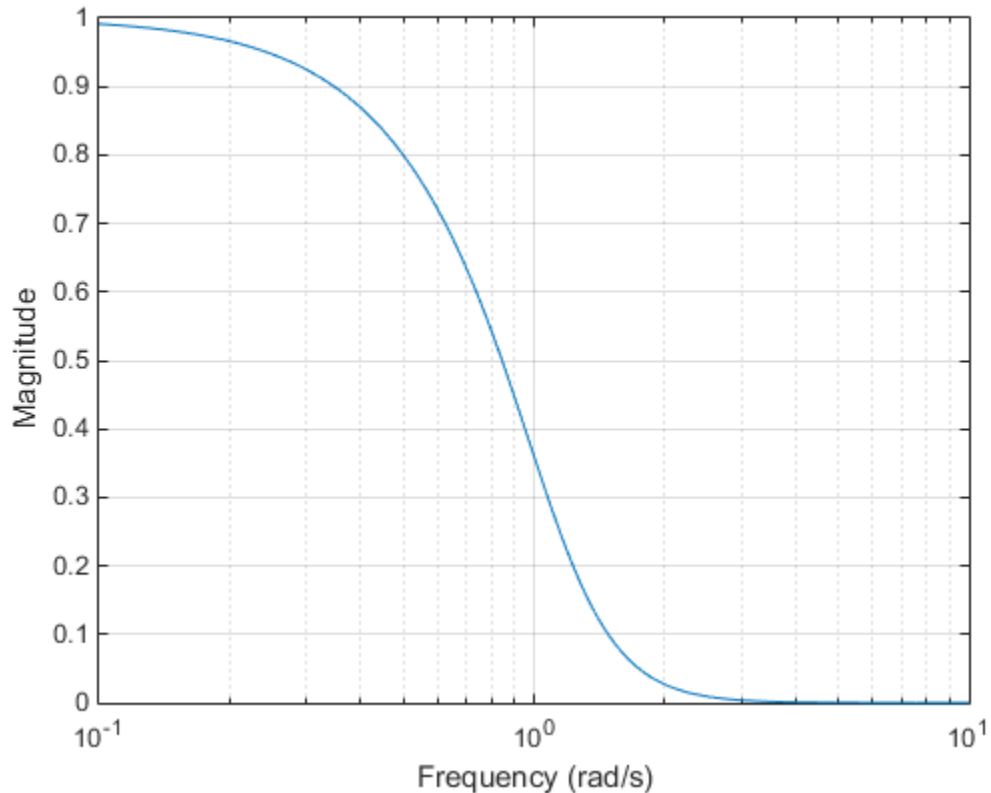
Note The lowpass filters shown above were created with the analog prototype functions `besselap`, `buttap`, `cheb1ap`, `cheb2ap`, and `ellipap`. These functions find the zeros, poles, and gain of an nth-order analog filter of the appropriate type with a cutoff frequency of 1 rad/s. The complete filter design functions (`besself`, `butter`, `cheby1`, `cheby2`, and `ellip`) call the prototyping functions as a first step in the design process. See "Special Topics in IIR Filter Design" on page 2-39 for details.

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

```
[z,p,k] = ellipap(5,0.5,20);
```

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

Direct IIR Filter Design

This toolbox uses the term *direct methods* to describe techniques for IIR design that find a filter based on specifications in the discrete domain. Unlike the analog prototyping method, direct design methods are not constrained to the standard lowpass, highpass, bandpass, or bandstop configurations. Rather, these functions design filters with an arbitrary, perhaps multiband, frequency response. This section discusses the yulewalk function, which is intended specifically for filter design; "Parametric Modeling" on page 7-17 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 nth-order IIR filter whose frequency-magnitude characteristics approximate those given in vectors f and m. f is a vector of frequency points ranging from 0 to 1, where 1 represents the Nyquist frequency. m is a vector containing the 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))
```
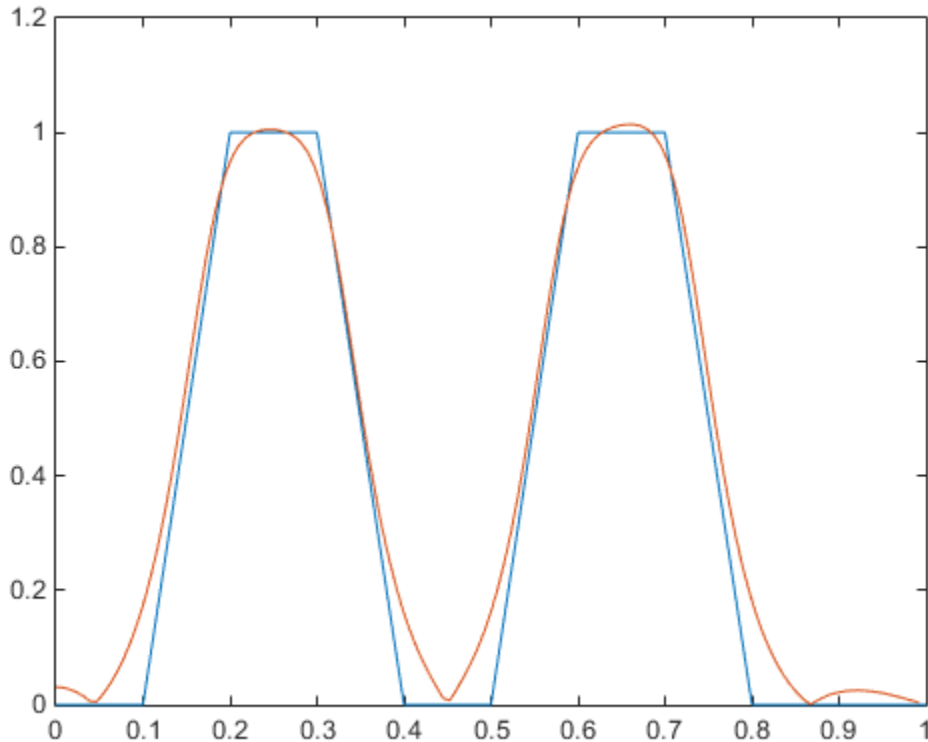

Generalized Butterworth Filter Design

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`

Filter Design Method	Description	Filter Functions
Multiband with Transition Bands	Equiripple or least squares approach over sub-bands of the frequency range	`firls`, `firpm`, `firpmord`
Constrained Least Squares	Minimize squared integral error over entire frequency range subject to maximum error constraints	`fircls`, `fircls1`
Arbitrary Response	Arbitrary responses, including nonlinear phase and complex filters	`cfirpm`
Raised Cosine	Lowpass response with smooth, sinusoidal transition	`rcosdesign`

Linear Phase Filters

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

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

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

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

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

Windowing Method

Consider the ideal, or "brick wall," digital lowpass filter with a cutoff frequency of 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.
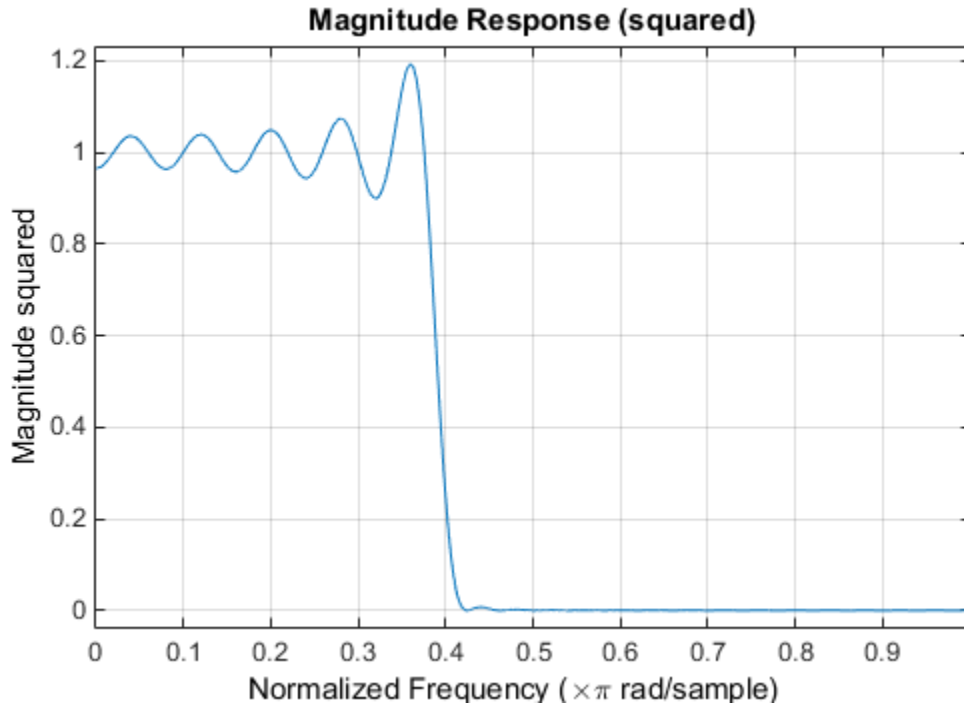
Ringing and ripples occur in the response, especially near the band edge. This "Gibbs effect" does not vanish as the filter length increases, but a nonrectangular window reduces its magnitude. Multiplication by a window in the time domain causes a convolution or smoothing in the frequency domain. Apply 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.

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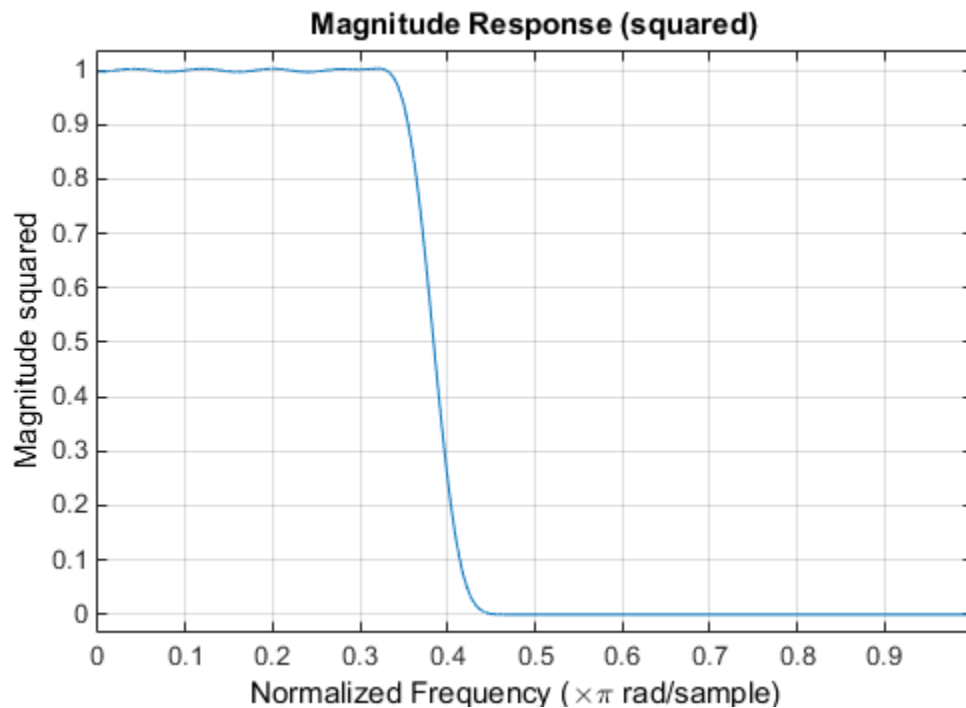
Using a Hamming window greatly reduces the ringing. This improvement is at the expense of transition width (the windowed version takes longer to ramp from passband to stopband) and optimality (the windowed version does not minimize the integrated squared error).

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.

Magnitude Response (squared)
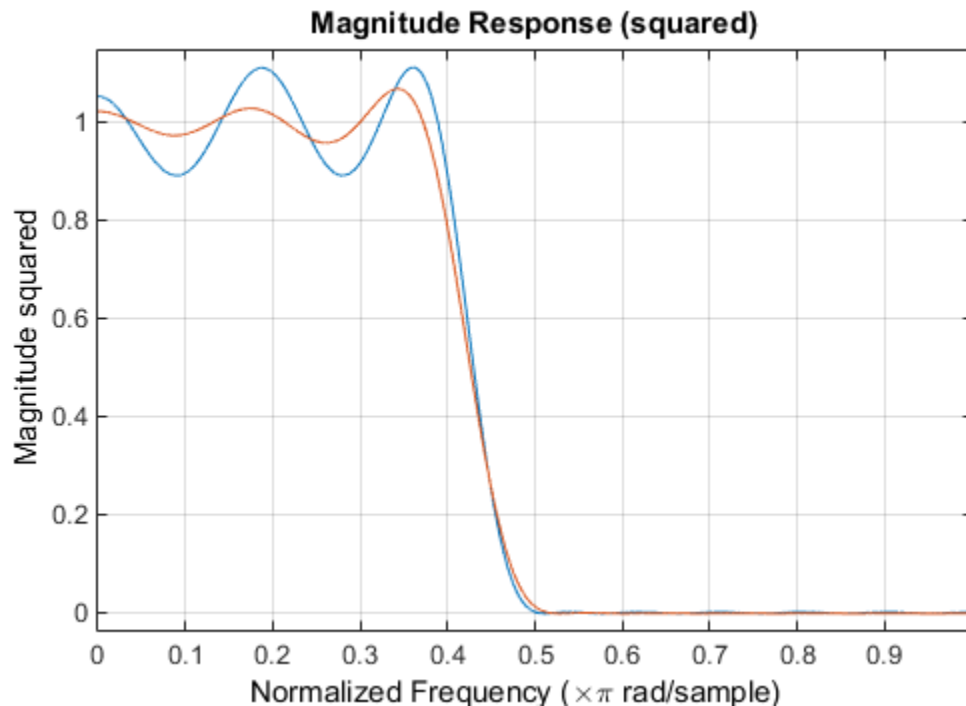
The filter designed with `firpm` exhibits equiripple behavior. Also note that the `firls` filter has a better response over most of the passband and stopband, but at the band edges (`f = 0.4` and `f = 0.5`), the response is further away from the ideal than the `firpm` filter. This shows that the `firpm` 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)
```
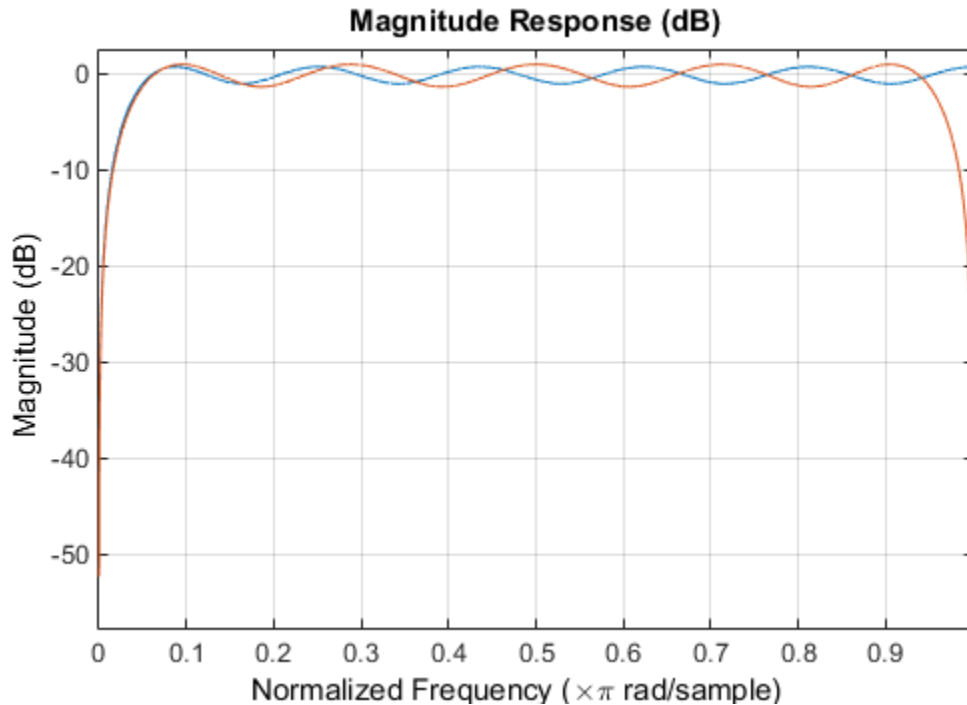
You can find the delayed Hilbert transform of a signal `x` by passing it through these filters.

```
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 "Hilbert Transform" on page 7-47, 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)
```

```
legend('Odd order','Even order','Location','best')
```
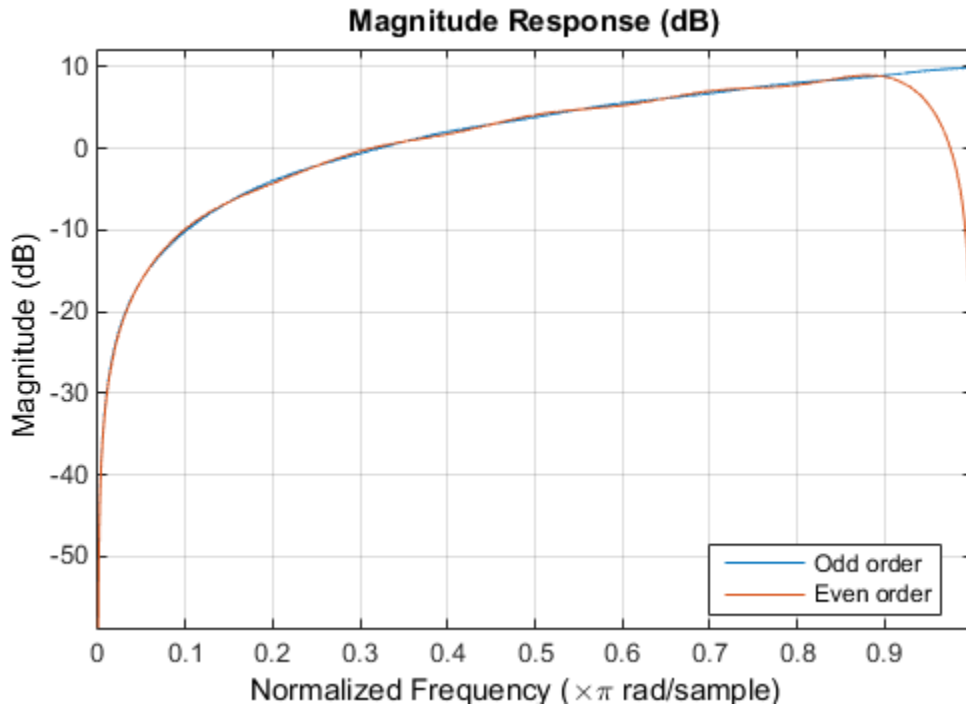

Constrained Least Squares FIR Filter Design

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

Instead of defining passbands, stopbands, and transition regions, the CLS method accepts a cutoff frequency (for the highpass, lowpass, bandpass, or bandstop cases), or passband and stopband edges (for multiband cases), for the 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.
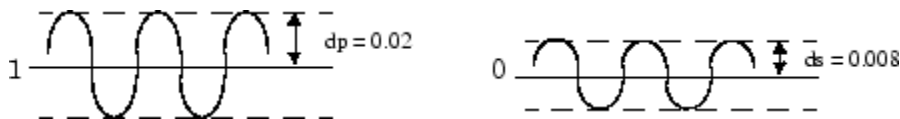
To approach this design problem using `fircls1`, use the following commands:

```
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.

Magnitude Response (squared)
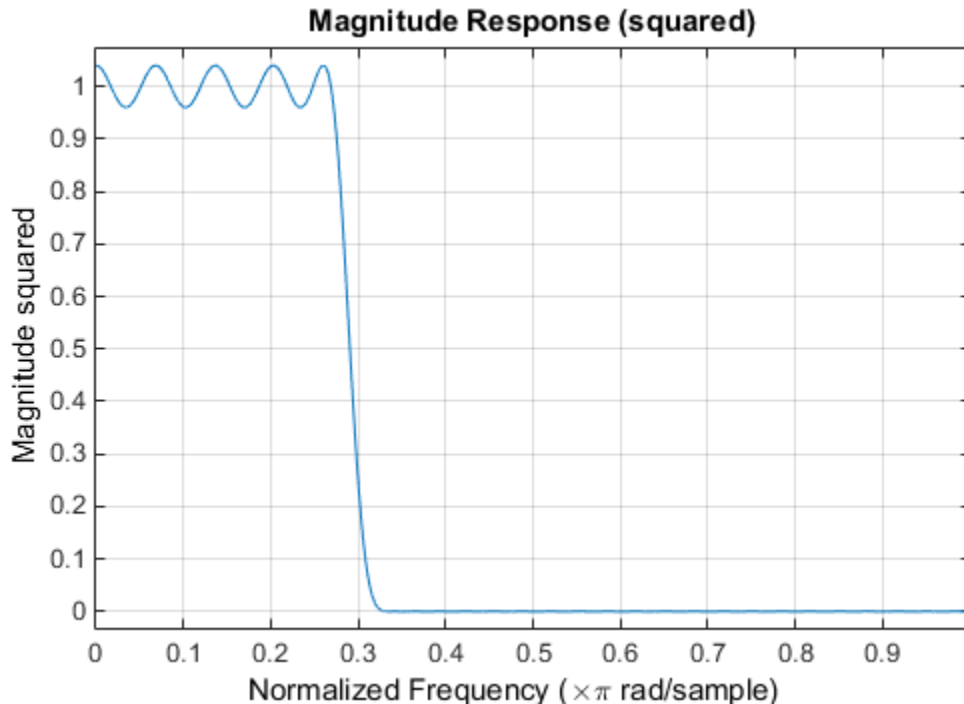
Multiband CLS Filter Design

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.
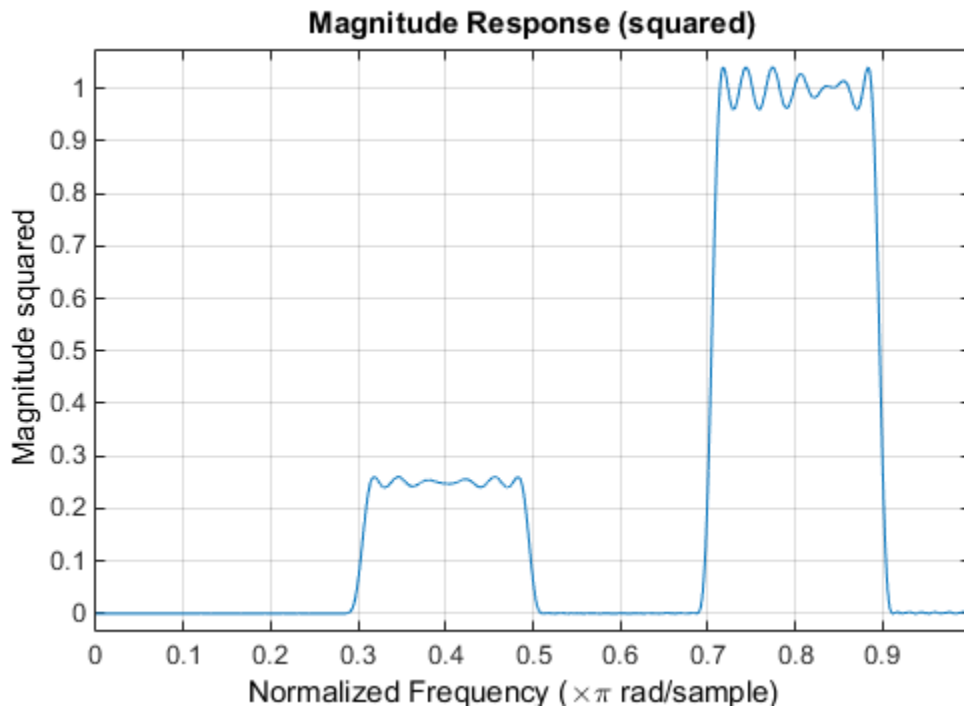
Weighted CLS Filter Design

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.

Magnitude Response (squared)
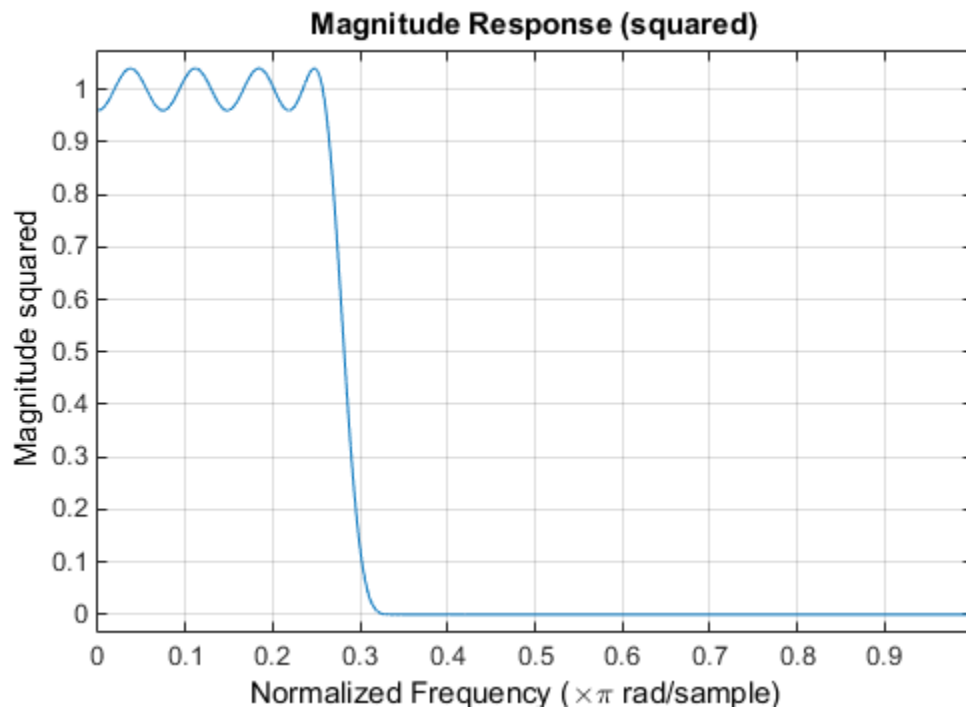
Arbitrary-Response Filter Design

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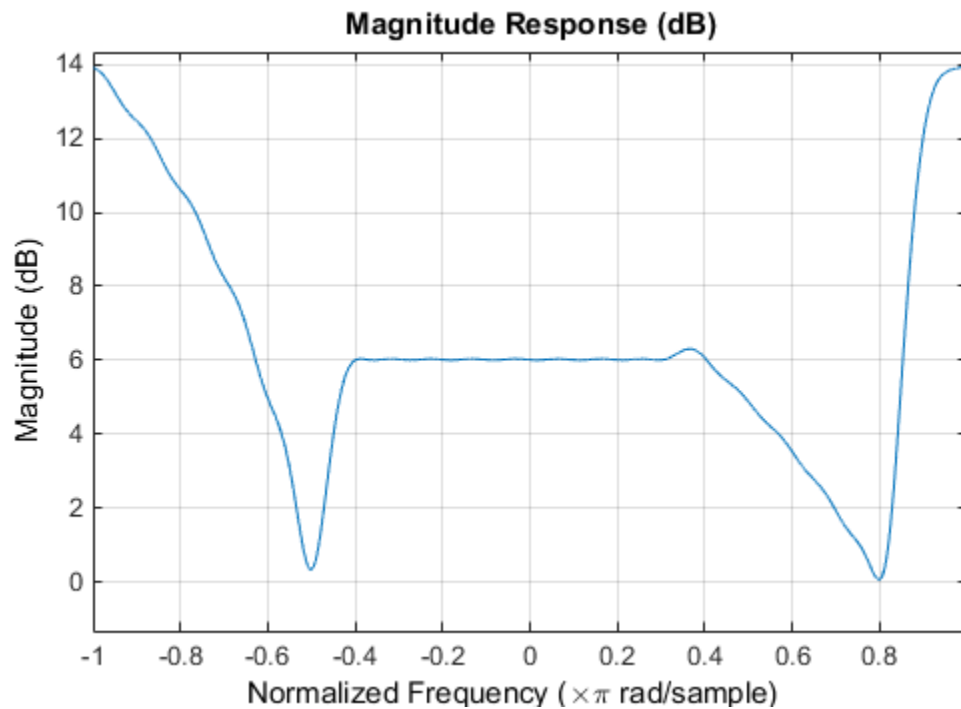
The filter response for this multiband filter is complex, which is expected because of the asymmetry in the frequency domain. The impulse response, which you can select from the FVTool toolbar, is shown below.

Impulse Response
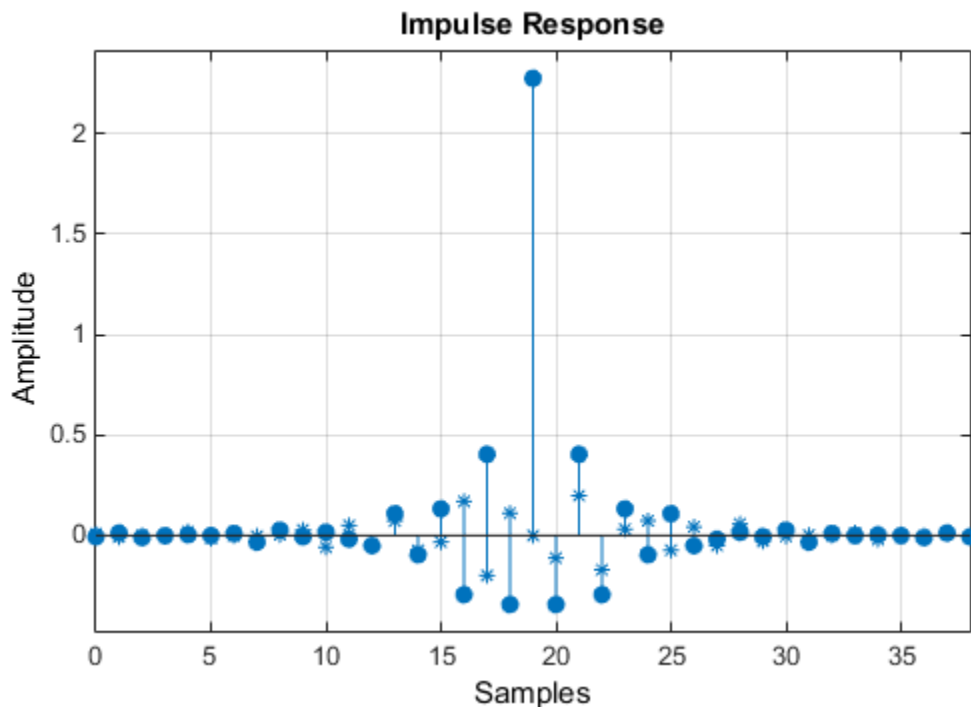
Filter Design with Reduced Delay

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.

Magnitude Response (squared)
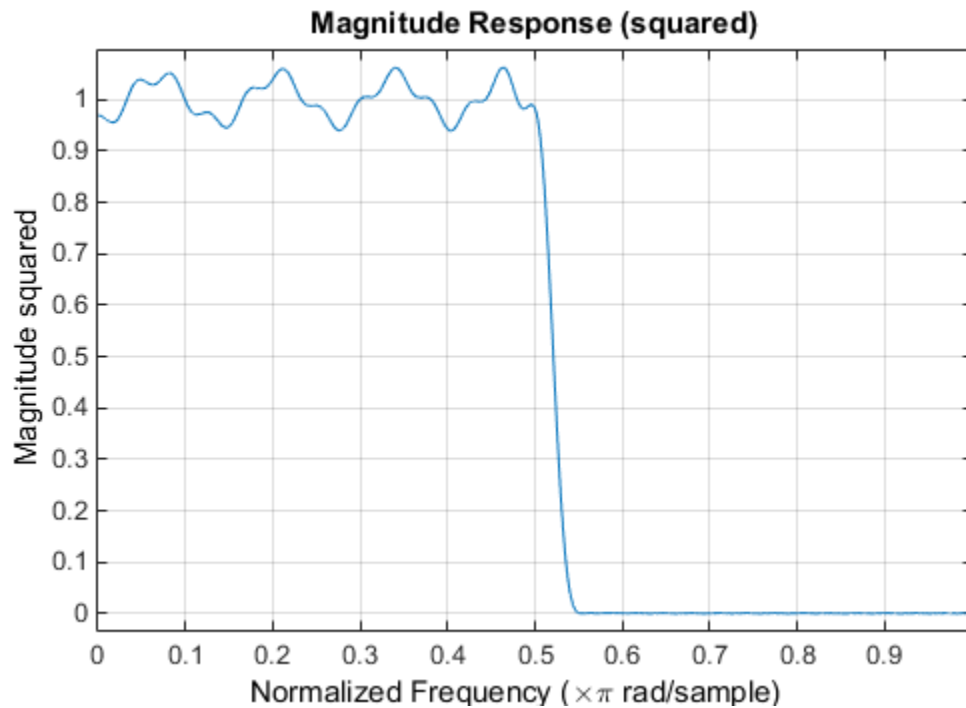
The group delay of the filter reveals that the offset has been reduced from `N/2` to `N/2-16` (i.e., from `30.5` to `14.5`). Now, however, the group delay is no longer flat in the passband region. To create this plot, click the **Group Delay** button on the toolbar.
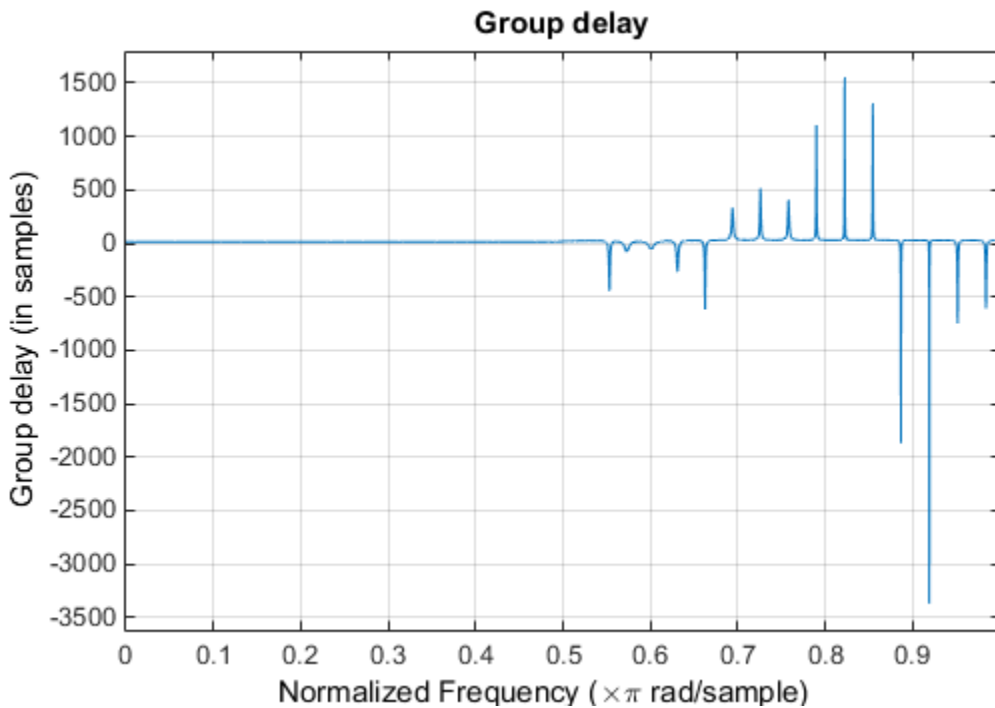
If we compare this nonlinear-phase filter to a linear-phase filter that has exactly 14.5 samples of group delay, the resulting filter is of order 2*14.5, or 29. Using `b = cfirpm(29,[0 0.5 0.55 1],'lowpass')`, the passband and stopband ripple is much greater for the order 29 filter. These comparisons 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" on page 2-39
"Analog Prototype Design" on page 2-39
"Frequency Transformation" on page 2-40
"Filter Discretization" on page 2-42

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

Lowpass to bandpass $$s' = \frac{\omega_0}{B_\omega} \frac{(s/\omega_0)^2 + 1}{s/\omega_0}$$	`[numt,dent]    = lp2bp (num,den,Wo,Bw)`
	
	 `[At,Bt,Ct,Dt] = lp2bp (A,B,C,D,Wo,Bw)`

Lowpass to bandstop $$s' = \frac{B_\omega}{\omega_0} \frac{s/\omega_0}{(s/\omega_0)^2 + 1}$$	`[numt,dent]    = lp2bs (num,den,Wo,Bw)`
	
	 `[At,Bt,Ct,Dt] = lp2bs( A,B,C,D,Wo,Bw)`

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

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

and

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

where ω_1 is the lower band edge and ω_2 is the upper band edge.

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

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

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

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))              % Plot log magnitude vs. freq
xlabel('Frequency (Hz)')
grid
```
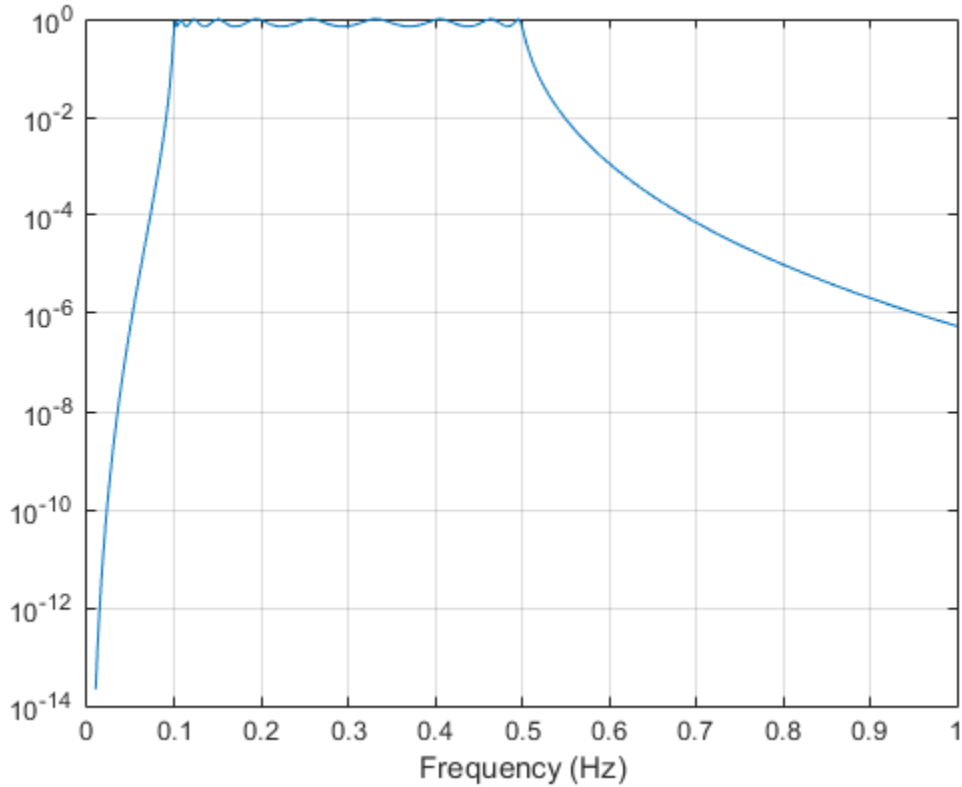
Filter Discretization

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

Analog to Digital Transformation	Transformation Function
Impulse invariance	`[numd,dend] = impinvar (num,den,fs)`
Bilinear transform	`[zd,pd,kd] = bilinear (z,p,k,fs,Fp)`

Analog to Digital Transformation	Transformation Function
	`[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.

Magnitude Response (dB) and Phase Response
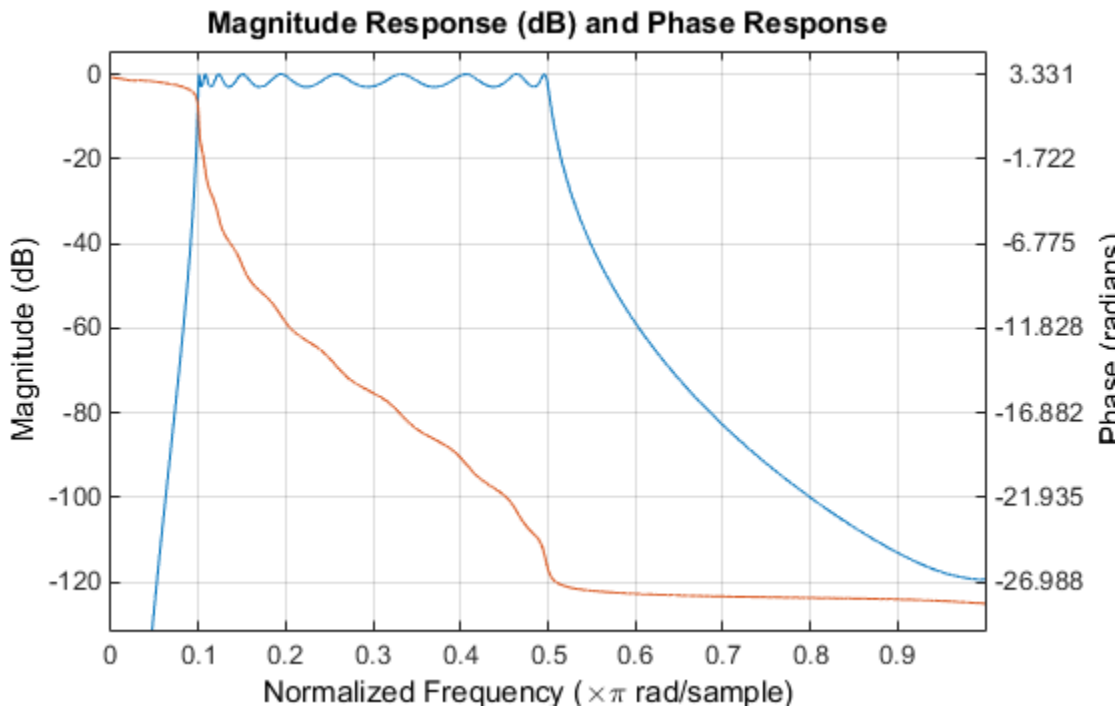
Impulse invariance retains the cutoff frequencies of 0.1 Hz and 0.5 Hz.

Bilinear Transformation

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

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

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

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

The toolbox function `bilinear` implements this operation, where the frequency warping constant k is equal to twice the sampling frequency (`2*fs`) by default, and equal to $2\pi f_p \big/ \tan\left(\pi f_p \big/ f_s\right)$ 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.

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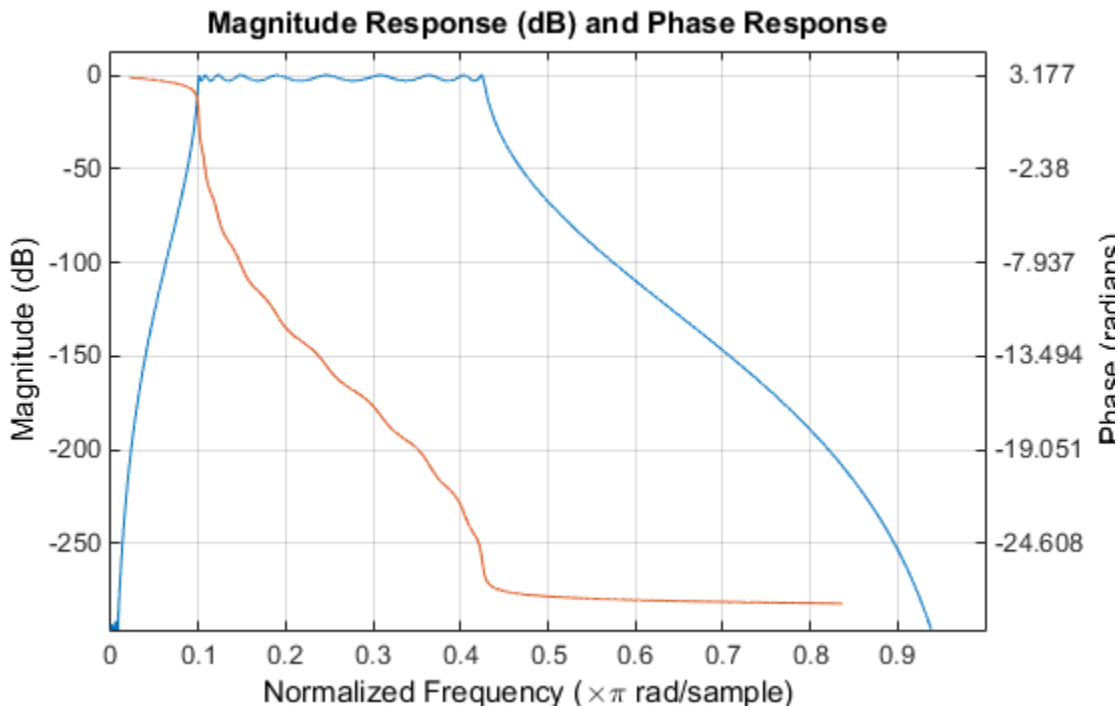
The lower band edge is at 0.1 Hz as expected. Notice, however, that the upper band edge is slightly less than 0.5 Hz, although in the analog domain it was exactly 0.5 Hz. This illustrates the nonlinear nature of the bilinear transformation. To counteract this nonlinearity, it is necessary to create analog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

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

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

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

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

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

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

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

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

Filtering Data With Signal Processing Toolbox Software

Lowpass FIR Filter -- Window Method

This example shows how to design and implement an FIR filter using two command line functions, `fir1` and `designfilt`, and the interactive tool `fdatool`.

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

```
rng default

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

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

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

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

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

fvtool(b,1,'Fs',Fs)
```
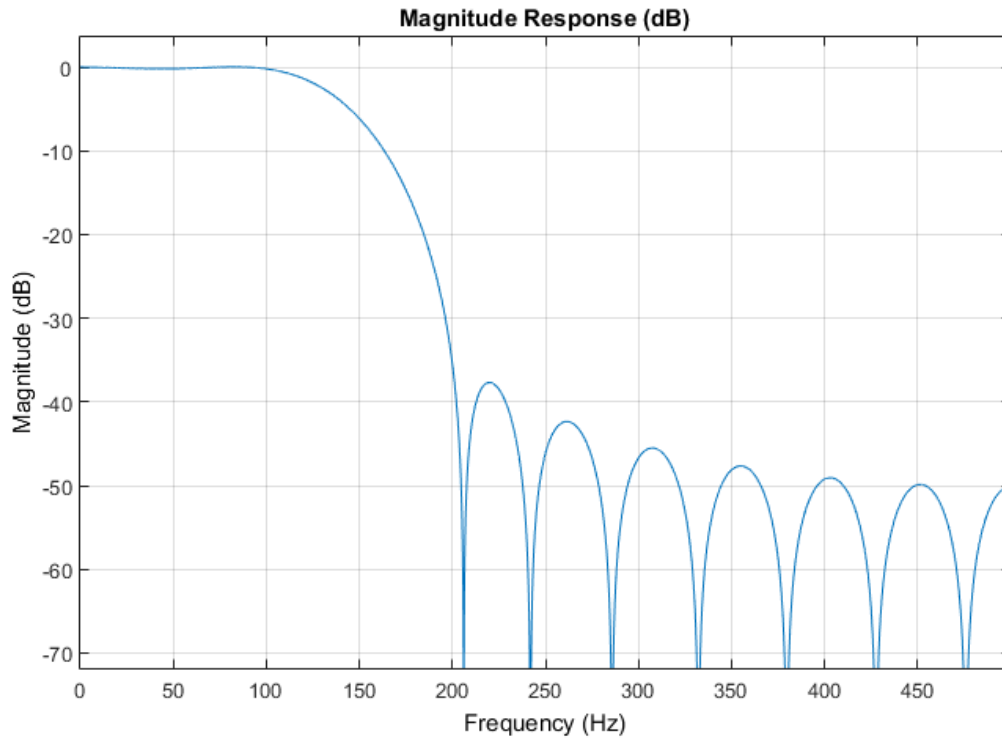
Apply the filter to the signal and plot the result for the first ten periods of the 100 Hz sinusoid.

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

plot(t(1:100),x(1:100))
hold on
plot(t(1:100),y(1:100))

xlabel('Time (s)')
ylabel('Amplitude')
legend('Original Signal','Filtered Data')
```
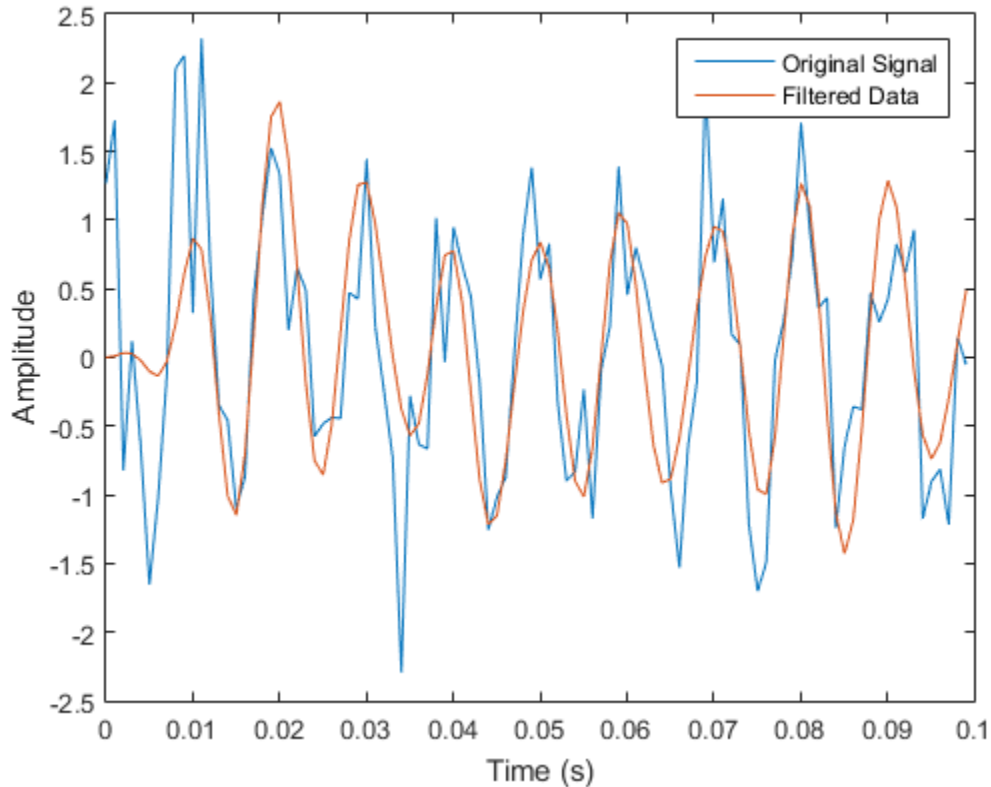
Design the same filter using `designfilt`. Set the filter response to `'lowpassfir'` and input the specifications as `Name,Value` pairs. With `designfilt`, you can specify your filter design in Hz.

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

Filter the data and plot the result.

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

figure
plot(t(1:100),x(1:100))
```

```
hold on
plot(t(1:100),y1(1:100))

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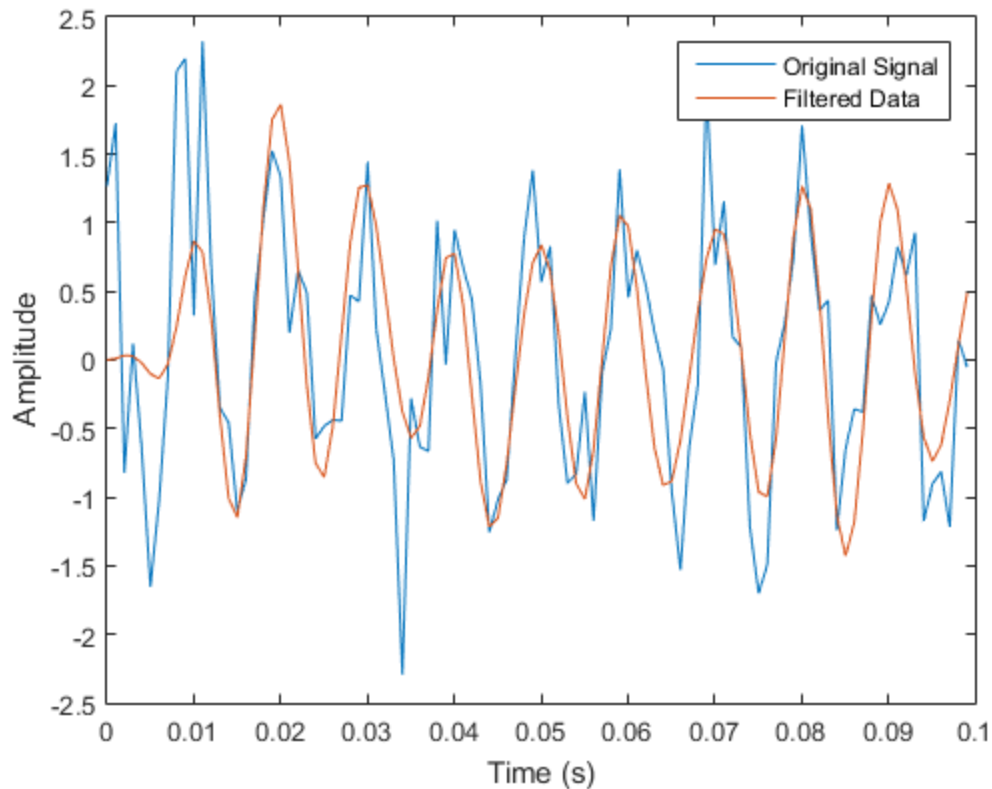

Lowpass FIR Filter with FDATool

This example shows how to design and implement a lowpass FIR filter using the window method with the interactive tool `fdatool`.

- Start FDATool by entering `fdatool` at the command line.

- Set the **Response Type** to **Lowpass**.
- Set the **Design Method** to **FIR** and select the **Window** method.
- Under **Filter Order**, select **Specify order**. Set the order to 20.
- Under **Frequency Specifications**, set **Units** to **Hz**, **Fs** to 1000, and **Fc** to 150.
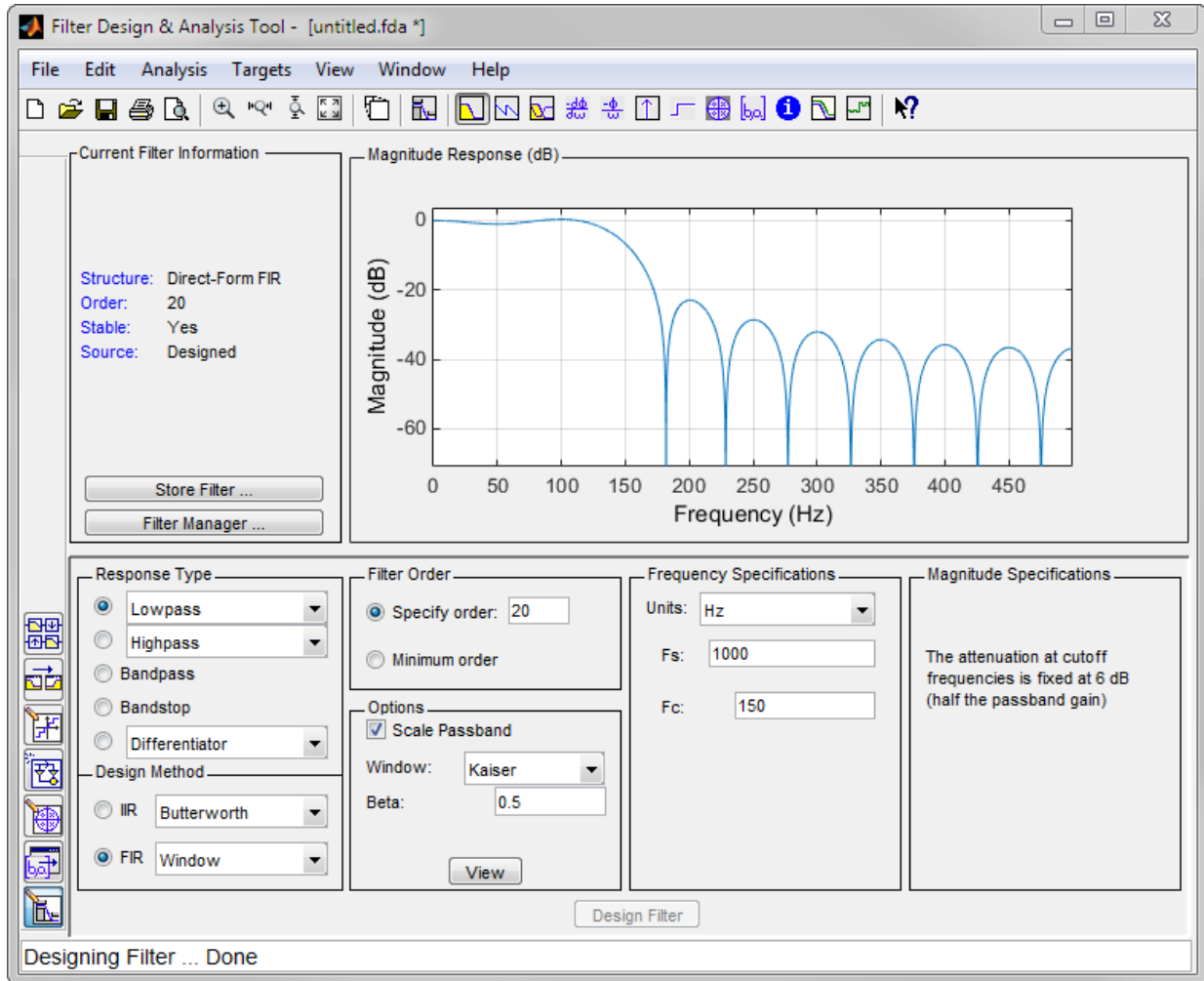
- Click **Design Filter**.

- Select **File** > **Export...** to export your FIR filter to the MATLAB® workspace as coefficients or a filter object. In this example, export the filter as an object. Specify the variable name as `Hd1`.
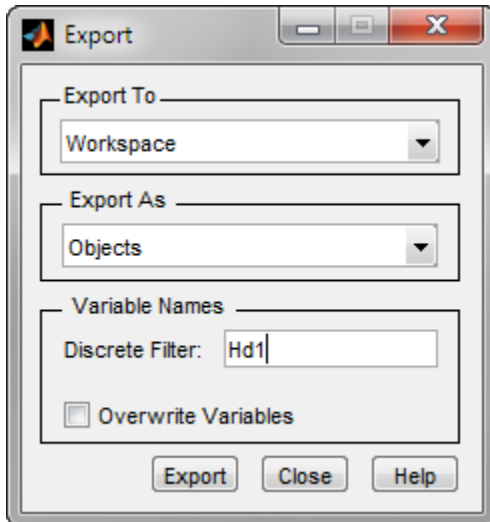
- Click **Export**.
- Filter the input signal in the command window with the exported filter object. Plot the result for the first ten periods of the 100 Hz sinusoid.

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

figure
plot(t(1:100),x(1:100))
hold on
plot(t(1:100),y2(1:100))

xlabel('Time (s)')
ylabel('Amplitude')
legend('Original Signal','Filtered Data')
```
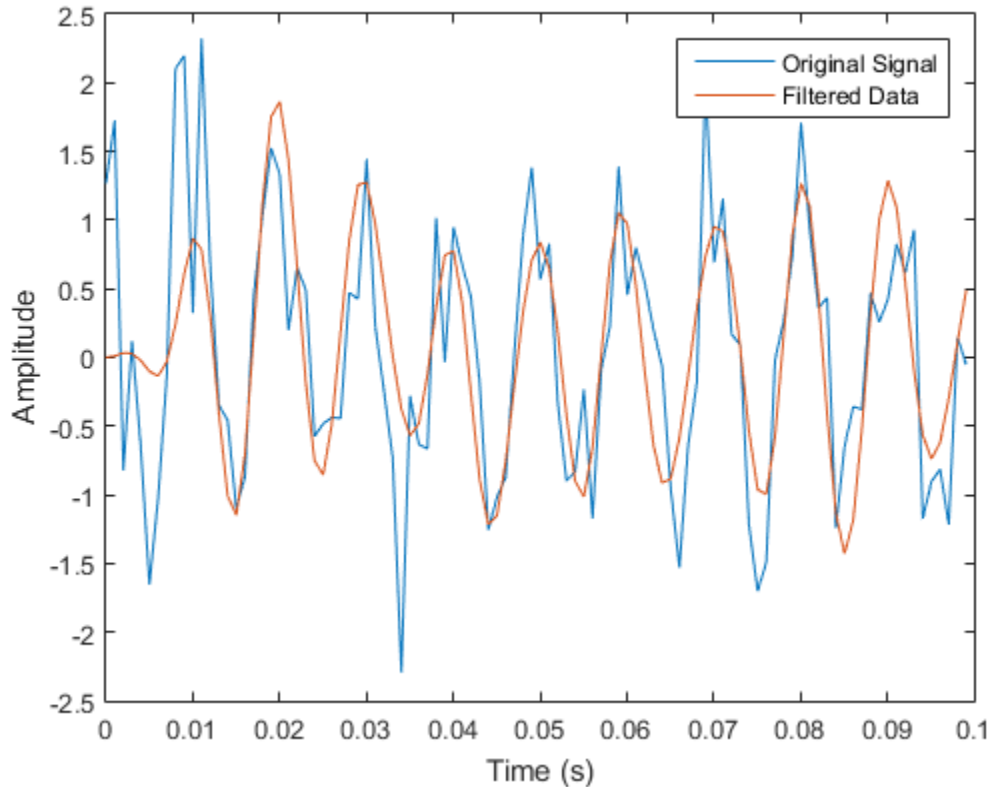
- Select **File** > **Generate MATLAB Code** to generate a MATLAB function to create a filter object using your specifications.

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

Bandpass Filters -- Minimum-Order FIR and IIR Systems

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

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

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

```
rng default

Fs = 1;
n = 1:365;
x = cos(2*pi*(1/7)*n)+cos(2*pi*(1/30)*n-pi/4);
trend = 3*sin(2*pi*(1/1480)*n);
y = x+trend+0.5*randn(size(n));

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

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

subplot(2,1,2)
plot(f,10*log10(pxx))
xlabel('Cycles/day')
ylabel('dB')
grid
```
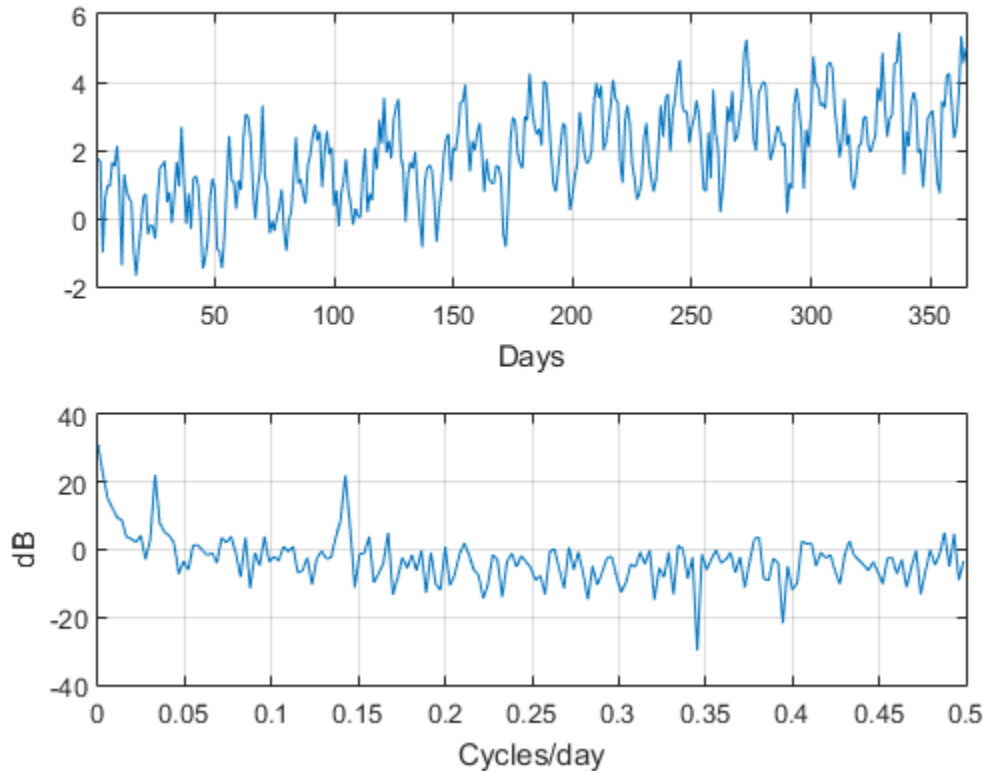
The low-frequency trend appears in the power spectral density estimate as increased low-frequency power. The low-frequency power appears approximately 10 dB above the oscillation at 1/30 cycles/day. Use this information in the specifications for the filter stopbands.

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

```
Hd1 = designfilt('bandpassfir', ...
    'StopbandFrequency1',1/60,'PassbandFrequency1',1/40, ...
    'PassbandFrequency2',1/4 ,'StopbandFrequency2',1/2 , ...
```

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

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

```
fprintf('The order of the FIR filter is %d\n',filtord(Hd1))
fprintf('The order of the IIR filter is %d\n',filtord(Hd2))

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

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

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

The order of the FIR filter is 78
The order of the IIR filter is 8
```
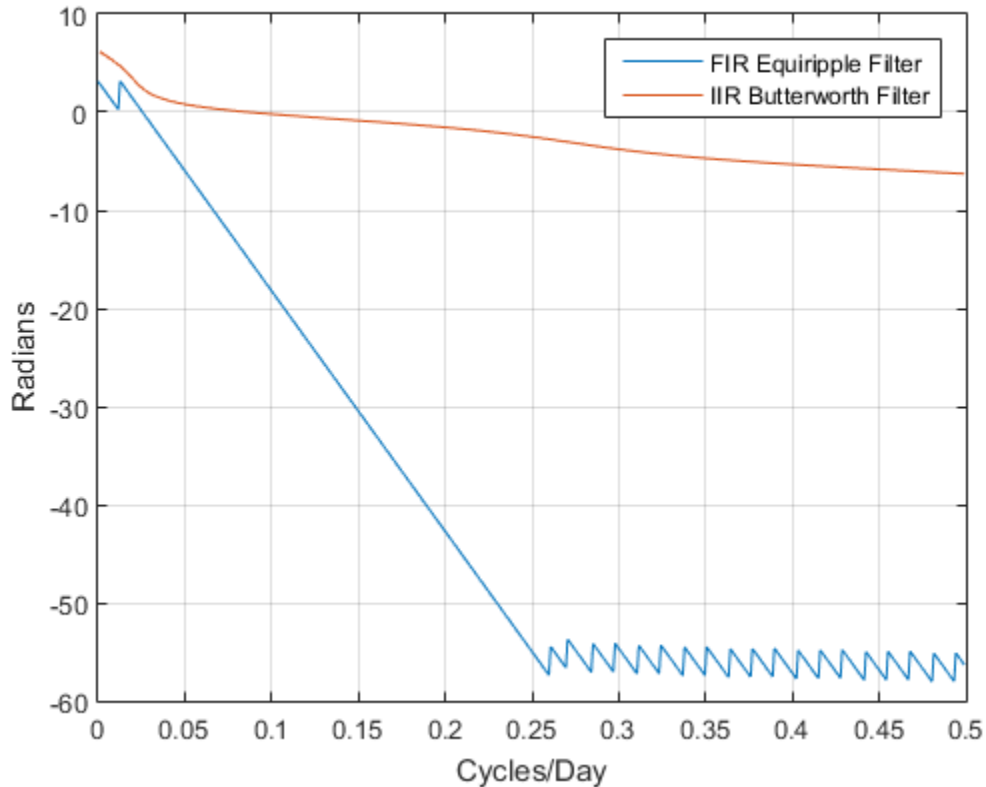
The IIR filter has a much lower order that the FIR filter. However, the FIR filter has a linear phase response over the passband, while the IIR filter does not. The FIR filter delays all frequencies in the filter passband equally, while the IIR filter does not.

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

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

```
Hdlow = designfilt('lowpassfir', ...
    'PassbandFrequency',1/4,'StopbandFrequency',1/2, ...
```

```
    'PassbandRipple',1,'StopbandAttenuation',10, ...
    'DesignMethod','equiripple','SampleRate',1);
```

Filter the data with the bandpass and lowpass filters.

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

Plot the PSD estimate of the bandpass IIR filter output. You can replace `yiir` with `yfir` in the following code to view the PSD estimate of the FIR bandpass filter output.

```
[pxx,f] = periodogram(yiir,[],length(yiir),Fs);

figure
plot(f,10*log10(pxx))

xlabel('Cycles/day')
ylabel('dB')
grid
```
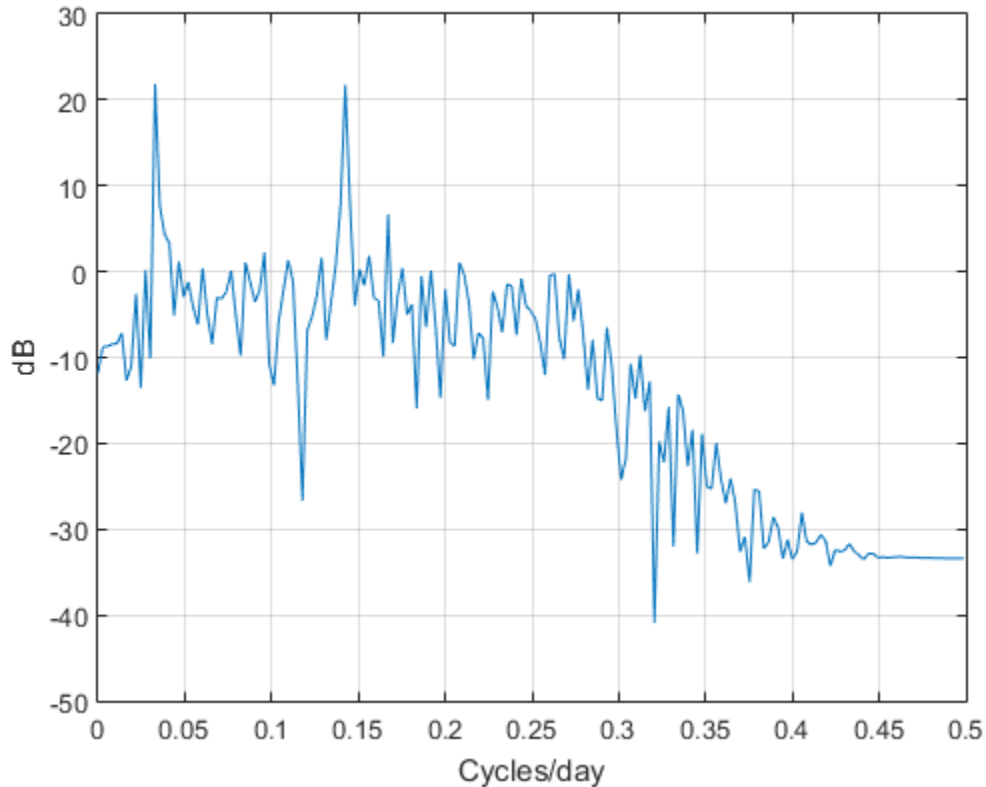
The PSD estimate shows the bandpass filter attenuates the low-frequency trend and high-frequency noise.

Plot the first 120 days of FIR and IIR filter output.

```
figure
plot(n(1:120),yfir(1:120))
hold on
plot(n(1:120),yiir(1:120))

axis([1 120 -2.8 2.8])
xlabel('Days')
legend('FIR bandpass filter output','IIR bandpass filter output', ...
    'Location','SouthEast')
```
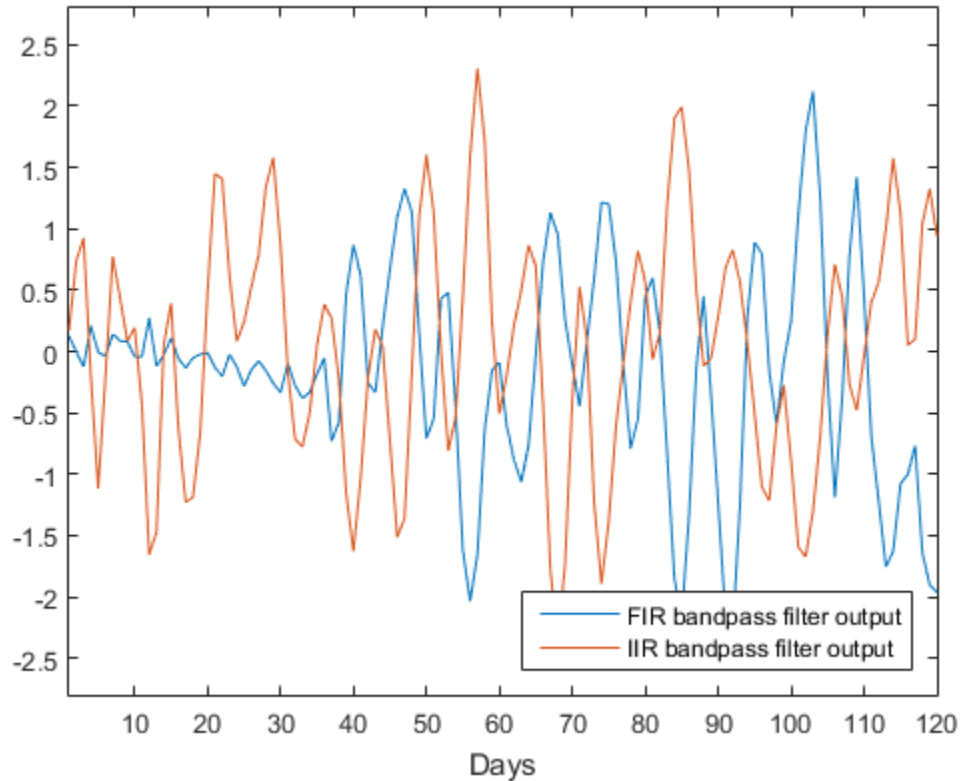
The increased phase delay in the FIR filter is evident in the filter output.

Plot the lowpass FIR filter output superimposed on the superposition of the 7-day and 30-day cycles for comparison.

```
figure
plot(n,x)
hold on
plot(n,ylow)

xlim([1 365])
xlabel('Days')
legend('7-day and 30-day cycles','FIR lowpass filter output', ...
    'Location','NorthWest')
```
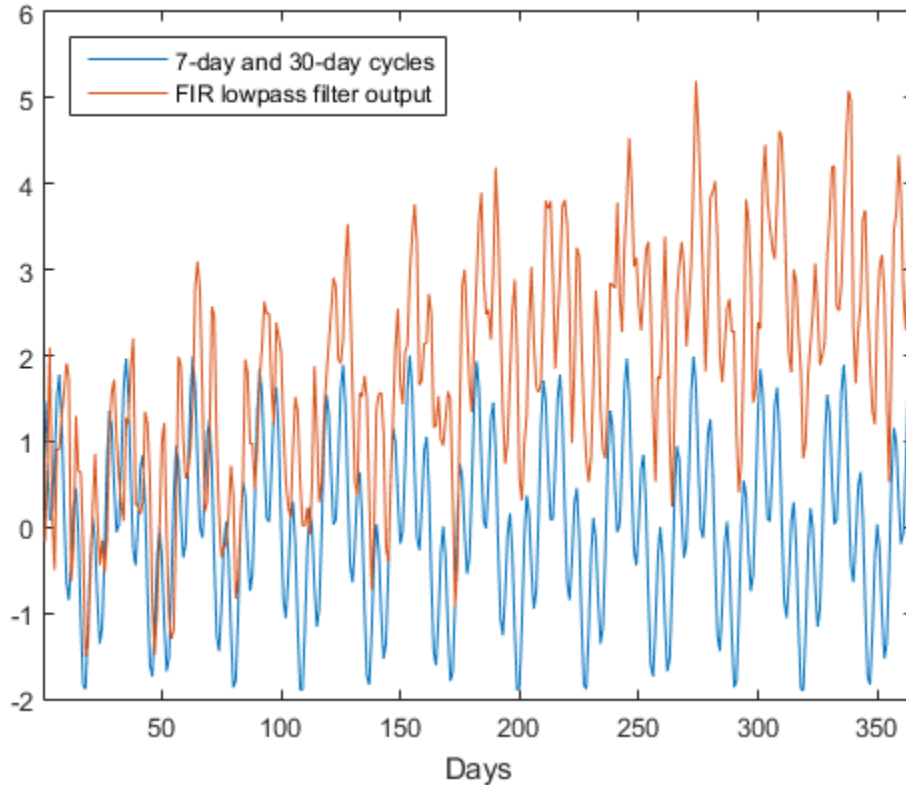
You can see in the preceding plot that the low-frequency trend is evident in the lowpass filter output. While the lowpass filter preserves the 7-day and 30-day cycles, the bandpass filters perform better in this example because the bandpass filters also remove the low-frequency trend.

Zero-Phase Filtering

This example shows how to perform zero-phase filtering.

Repeat the signal generation and lowpass filter design with `fir1` and `designfilt`. You do not have to execute the following code if you already have these variables in your workspace.

```
rng default
```

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

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

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

Filter the data using `filter`. Plot the first 100 points of the filter output along with a superimposed sinusoid with the same amplitude and initial phase as the input signal.

```
yout = filter(Hd,x);
xin = cos(2*pi*100*t);

figure
plot(t(1:100),xin(1:100))
hold on
plot(t(1:100),yout(1:100))

xlabel('Time (s)')
ylabel('Amplitude')
legend('Input Sine Wave','Filtered Data', ...
    'Location','NorthEast')
grid
```
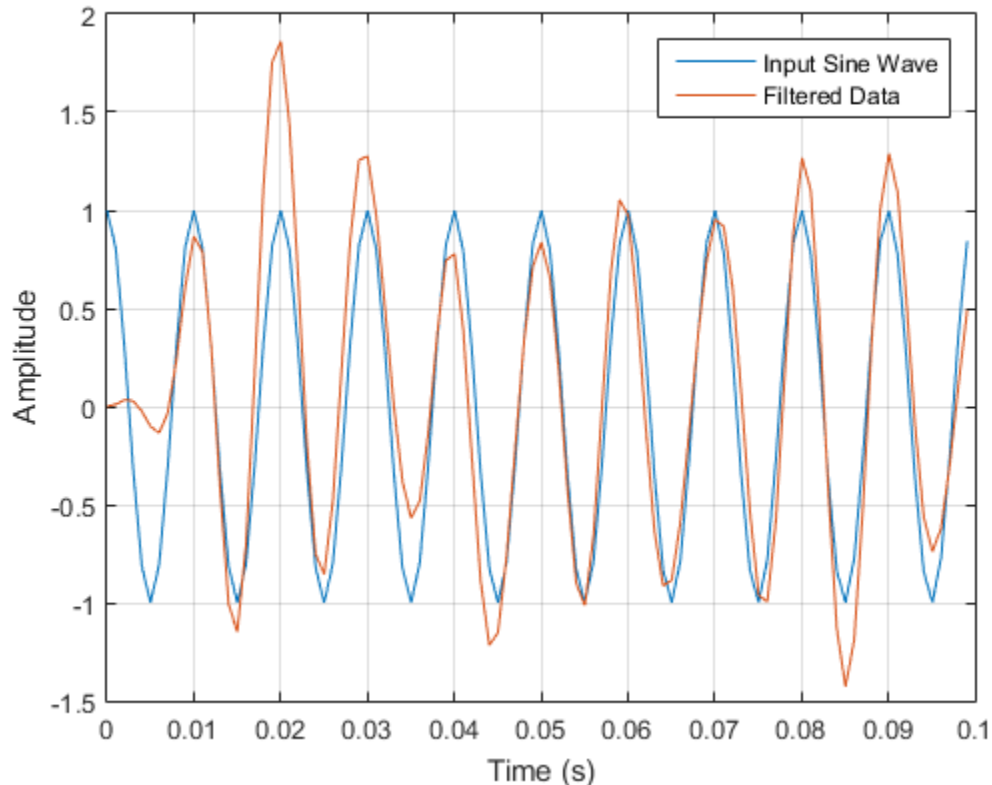
Looking at the initial 0.01 seconds of the filtered data, you see that the output is delayed with respect to the input. The delay appears to be approximately 0.01 seconds, which is almost 1/2 the length of the FIR filter in samples (10×0.001).

This delay is due to the filter's phase response. The FIR filter in these examples is a type I linear-phase filter. The group delay of the filter is 10 samples.

Plot the group delay using `fvtool`.

```
fvtool(Hd,'Analysis','grpdelay');
```
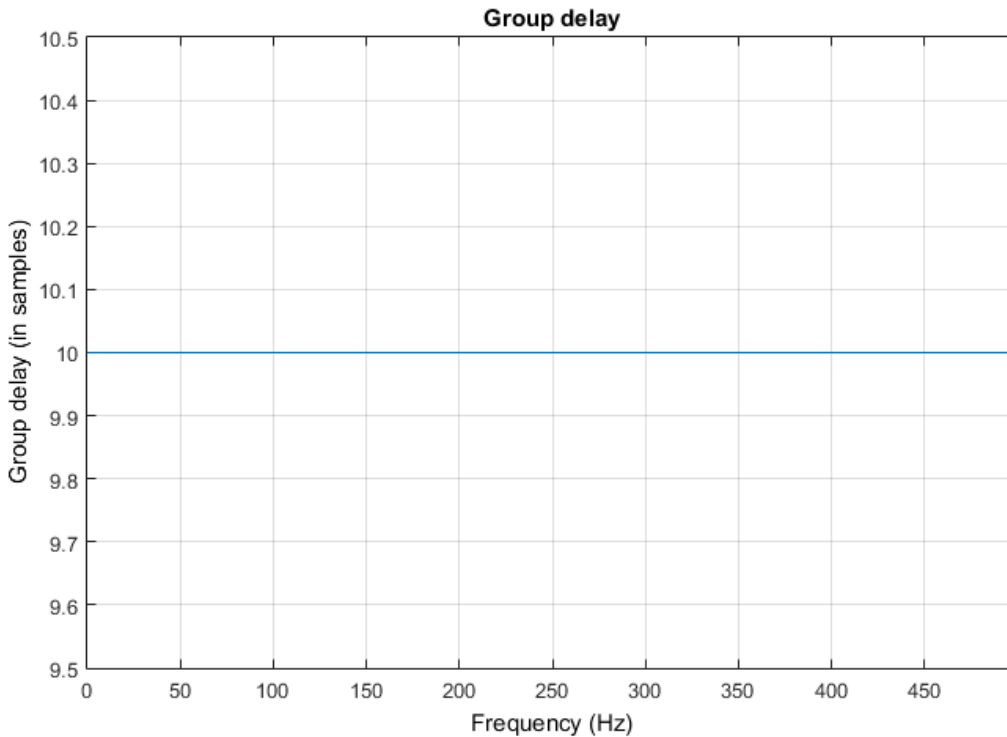
In many applications, phase distortion is acceptable. This is particularly true when phase response is linear. In other applications, it is desirable to have a filter with a zero-phase response. A zero-phase response is not technically possibly in a noncausal filter. However, you can implement zero-phase filtering using a causal filter with `filtfilt`.

Filter the input signal using `filtfilt`. Plot the responses to compare the filter outputs obtained with `filter` and `filtfilt`.

```
yzp = filtfilt(Hd,x);

figure
plot(t(1:100),xin(1:100))
hold on
plot(t(1:100),yout(1:100))
plot(t(1:100),yzp(1:100))
```

```matlab
xlabel('Time (s)')
ylabel('Amplitude')
legend('100-Hz Sine Wave','Filtered Signal','Zero-phase Filtering',...
    'Location','NorthEast')
```
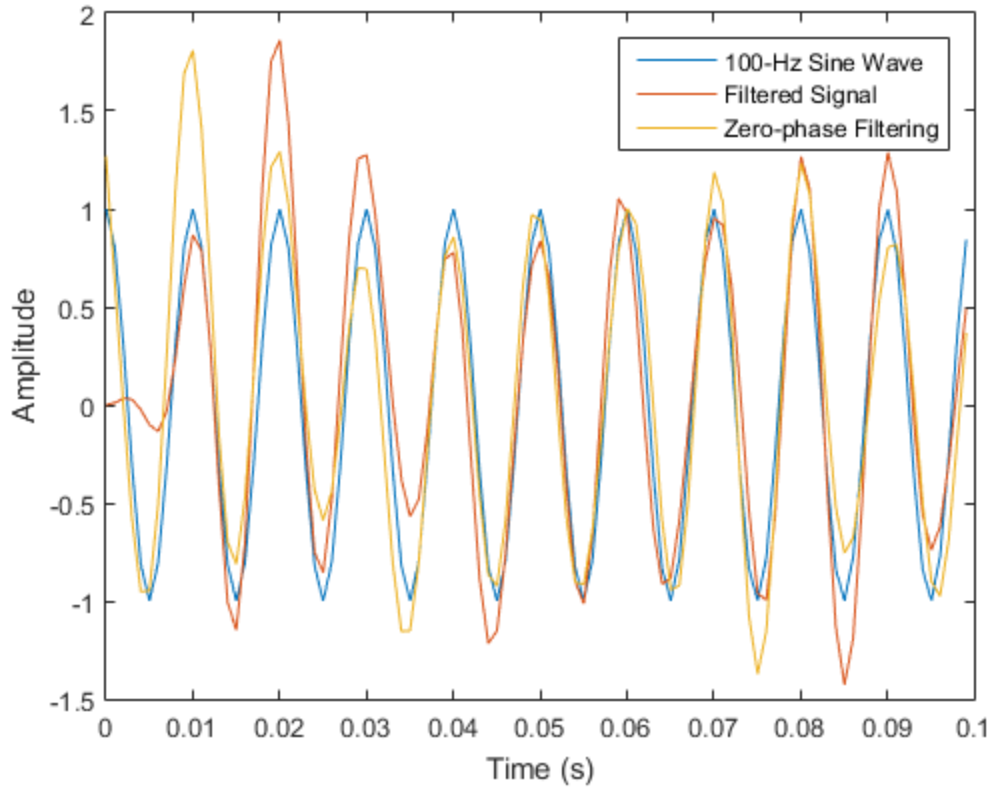

In the preceding figure, you can see that the output of `filtfilt` does not exhibit the delay due to the phase response of the FIR filter.

Practical Introduction to Digital Filtering

This example shows how to design, analyze, and apply a digital filter to your data. It will help you answer questions such as: how do I compensate for the delay introduced by a filter?, How do I avoid distorting my signal?, How do I remove unwanted content from my signal?, How do I differentiate my signal?, and How do I integrate my signal?

Filters can be used to shape the signal spectrum in a desired way or to perform mathematical operations such as differentiation and integration. In what follows you will learn some practical concepts that will ease the use of filters when you need them.

This example focuses on applications of digital filters rather than on their design. If you want to learn more about how to design digital filters see the "Practical Introduction to Digital Filter Design" example.

Compensating for Delay Introduced by Filtering

Digital filters introduce delay in your signal. Depending on the filter characteristics, the delay can be constant over all frequencies, or it can vary with frequency. The type of delay determines the actions you have to take to compensate for it. The **grpdelay** function allows you to look at the filter delay as a function of frequency. Looking at the output of this function allows you to identify if the delay of the filter is constant or if it varies with frequency (i.e. if it is frequency-dependent).

Filter delay that is constant over all frequencies can be easily compensated for by shifting the signal in time. FIR filters usually have constant delay. On the other hand, delay that varies with frequency causes phase distortion and can alter a signal waveform significantly. Compensating for frequency-dependent delay is not as trivial as for the constant delay case. IIR filters introduce frequency-dependent delay.

Compensating for Constant Filter Delay

As mentioned before, you can measure the group of delay of the filter to verify that it is a constant function of frequency. You can use the **grpdelay** function to measure the filter delay, D, and compensate for this delay by appending D zeros to the input signal and shifting the output signal in time by D samples.

Consider a noisy electrocardiogram signal that you want to filter to remove high frequency noise above 75 Hz. You want to apply an FIR lowpass filter and compensate for the filter delay so that the noisy and filtered signals are aligned correctly and can be plotted on top of each other for comparison.

```
Fs = 500;                       % sample rate in Hz
N = 500;                        % number of signal samples
rng default;
x = ecg(N)'+0.25*randn(N,1);    % noisy waveform
t = (0:N-1)/Fs;                 % time vector

% Design a 70th order lowpass FIR filter with cutoff frequency of 75 Hz.

Fnorm = 75/(Fs/2);              % Normalized frequency
df = designfilt('lowpassfir','FilterOrder',70,'CutoffFrequency',Fnorm);
```

Plot the group delay of the filter to verify that it is constant across all frequencies indicating that the filter is linear phase. Use the group delay to measure the delay of the filter.

```
grpdelay(df,2048,Fs)    % plot group delay
D = mean(grpdelay(df))  % filter delay in samples


D =

    35
```
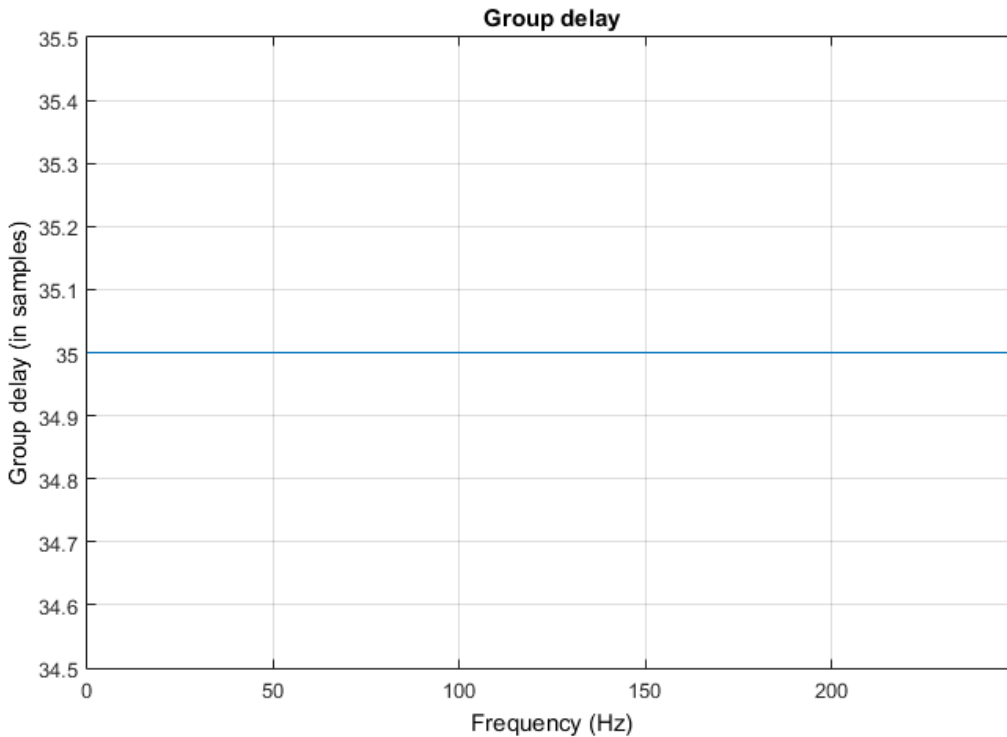
Before filtering, append D zeros at the end of the input data vector, x. This ensures that all the useful samples are flushed out of the filter, and that the input signal and the delay-compensated output signal have the same length. Filter the data and compensate for the delay by shifting the output signal by D samples. This last step effectively removes the filter transient.

```
y = filter(df,[x; zeros(D,1)]); % Append D zeros to the input data
y = y(D+1:end);                  % Shift data to compensate for delay

figure
plot(t,x,t,y,'r','linewidth',1.5);
title('Filtered Waveforms');
xlabel('Time (s)')
legend('Original Noisy Signal','Filtered Signal');
grid on
axis tight
```
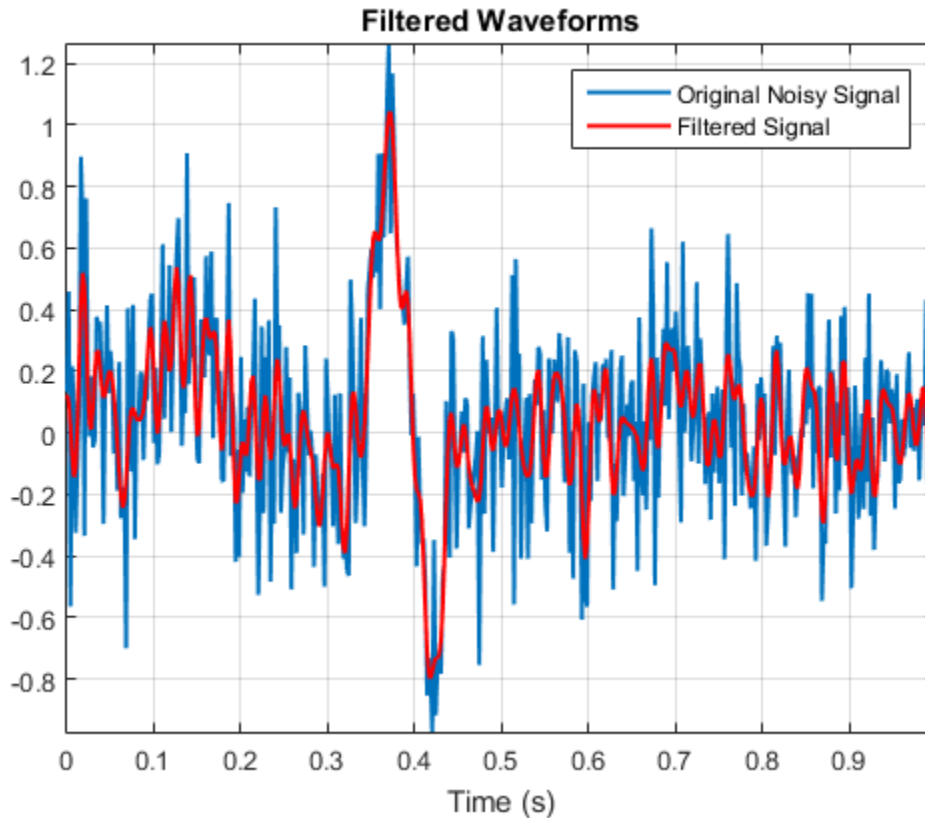
Compensating for Frequency-Dependent Delay

Frequency-dependent delay causes phase distortion in the signal. Compensating for this type of delay is not as trivial as for the constant delay case. If your application allows off-line processing, you can remove the frequency-dependent delay by implementing zero-phase filtering using the **filtfilt** function. **filtfilt** performs zero-phase filtering by processing the input data in both the forward and reverse directions. The main effect is that you obtain zero-phase distortion, i.e., you filter data with an equivalent filter that has a constant delay of 0 samples. Other effects are that you get a filter transfer function which equals the squared magnitude of the original filter transfer function, and a filter order that is double the order of the original filter.

Consider the ECG signal defined in the previous section. Filter this signal with and without delay compensation.

```
% Design a 7th order lowpass IIR elliptic filter with cutoff frequency
% of 75 Hz.

Fnorm = 75/(Fs/2); % Normalized frequency
df = designfilt('lowpassiir',...
                'PassbandFrequency',Fnorm,...
                'FilterOrder',7,...
                'PassbandRipple',1,...
                'StopbandAttenuation',60);
```

Plot the group delay of the filter and notice that it varies with frequency indicating that the filter delay is frequency-dependent.

```
grpdelay(df,2048,'half',Fs)
```
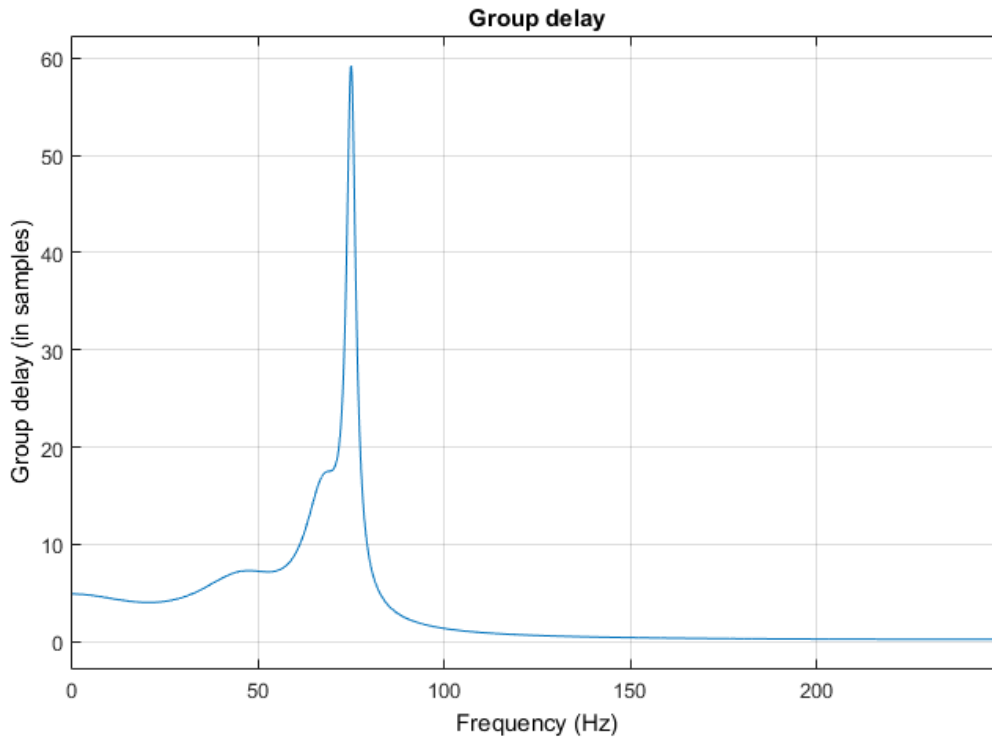
Filter the data and look at the effects of each filter implementation on the time signal.

```
y1 = filter(df,x);    % non-linear phase filter - no delay compensation
y2 = filtfilt(df,x);  % zero-phase implementation - delay compensation

figure
plot(t,x);
hold on
plot(t,y1,'r','linewidth',1.5);
plot(t,y2,'g','linewidth',1.5);
title('Filtered Waveforms');
xlabel('Time (s)')
legend('Original Signal','Non-linear phase IIR output',...
  'Zero-phase IIR output');
ax = axis;
axis([0.25 0.55 ax(3:4)])
grid on
```

Filtered Waveforms
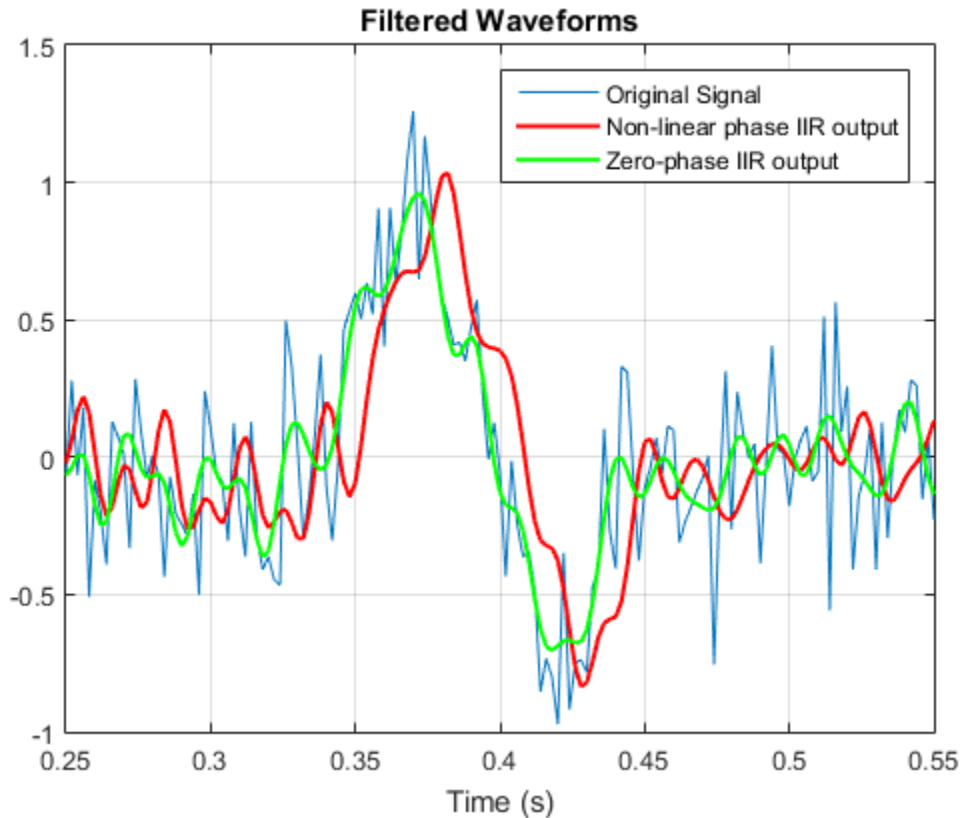
Notice how zero-phase filtering effectively removes the filter delay.

Zero-phase filtering is a great tool if your application allows for the non-causal forward/backward filtering operations, and for the change of the filter response to the square of the original response.

Filters that introduce constant delay are linear phase filters. Filters that introduce frequency-dependent delay are non-linear phase filters.

Removing Unwanted Spectral Content from a Signal

Filters are commonly used to remove unwanted spectral content from a signal. You can choose from a variety of filters to do this. You choose a lowpass filter when you

want to remove high frequency content, or a highpass filter when you want to remove low frequency content. You can also choose a bandpass filter to remove low and high frequency content while leaving an intermediate band of frequencies intact. You choose a bandstop filter when you want to remove frequencies over a given band.

Consider an audio signal that has a power-line hum and white noise. The power-line hum is caused by a 60 Hz tone. White noise is a signal that exists across all the audio bandwidth.

Load the audio signal.

```
Fs = 44100; % Sample rate
y = audioread('noisymusic.wav');
```

Plot the power spectrum of the signal. The red triangular marker shows the strong 60 Hz tone interfering with the audio signal.

```
[P,F] = pwelch(y,ones(8192,1),8192/2,8192,Fs,'power');
helperFilterIntroductionPlot1(F,P,[60 60],[-9.365 -9.365],...
  {'Original signal power spectrum', '60 Hz Tone'})
```
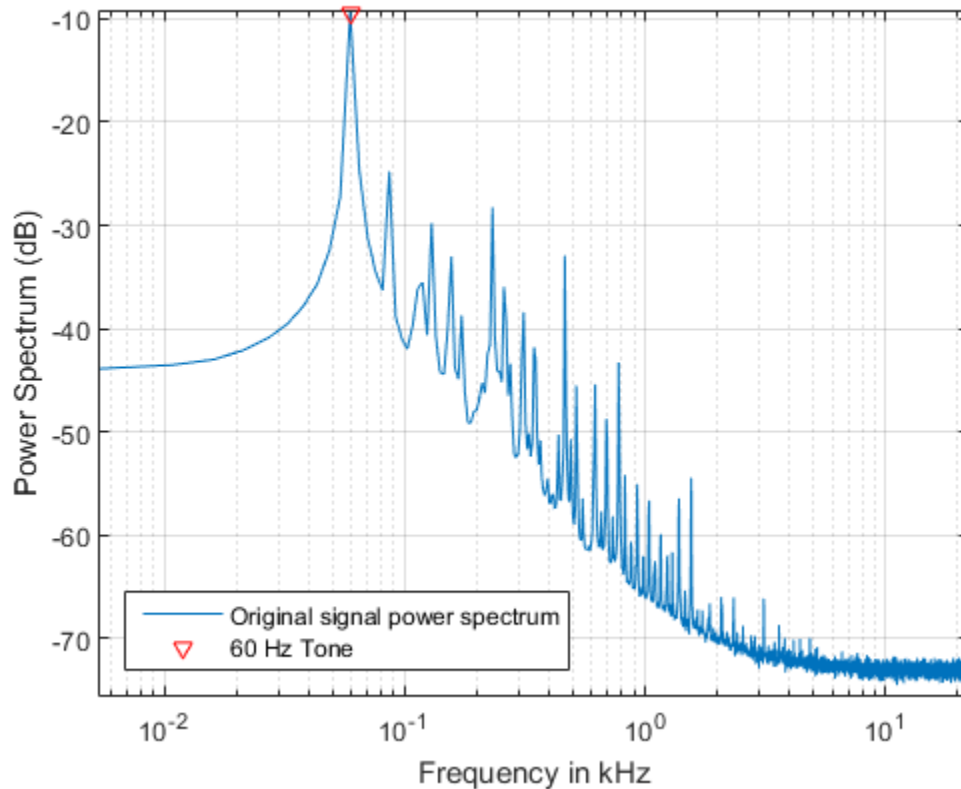
You can first remove as much white noise spectral content as possible using a lowpass filter. The passband of the filter should be set to a value that offers a good trade-off between noise reduction and audio degradation due to loss of high frequency content. Applying the lowpass filter before removing the 60 Hz hum is very convenient since you will be able to downsample the band-limited signal. The lower rate signal will allow you to design a sharper and narrower 60 Hz bandstop filter with a smaller filter order.

Design a lowpass filter with passband frequency of 1 kHz, and stopband frequency of 1.4 kHz. Choose a minimum order design.

```
Fp = 1e3;    % Passband frequency in Hz
Fst = 1.4e3; % Stopband frequency in Hz
Ap = 1;      % Passband ripple in dB
```

```
Ast = 95;     % Stopband attenuation in dB

% Design the filter
df = designfilt('lowpassfir','PassbandFrequency',Fp,...
                'StopbandFrequency',Fst,'PassbandRipple',Ap,...
                'StopbandAttenuation',Ast,'SampleRate',Fs);

% Analyze the filter response
hfvt = fvtool(df,'Fs',Fs,'FrequencyScale','log',...
  'FrequencyRange','Specify freq. vector','FrequencyVector',F);
```
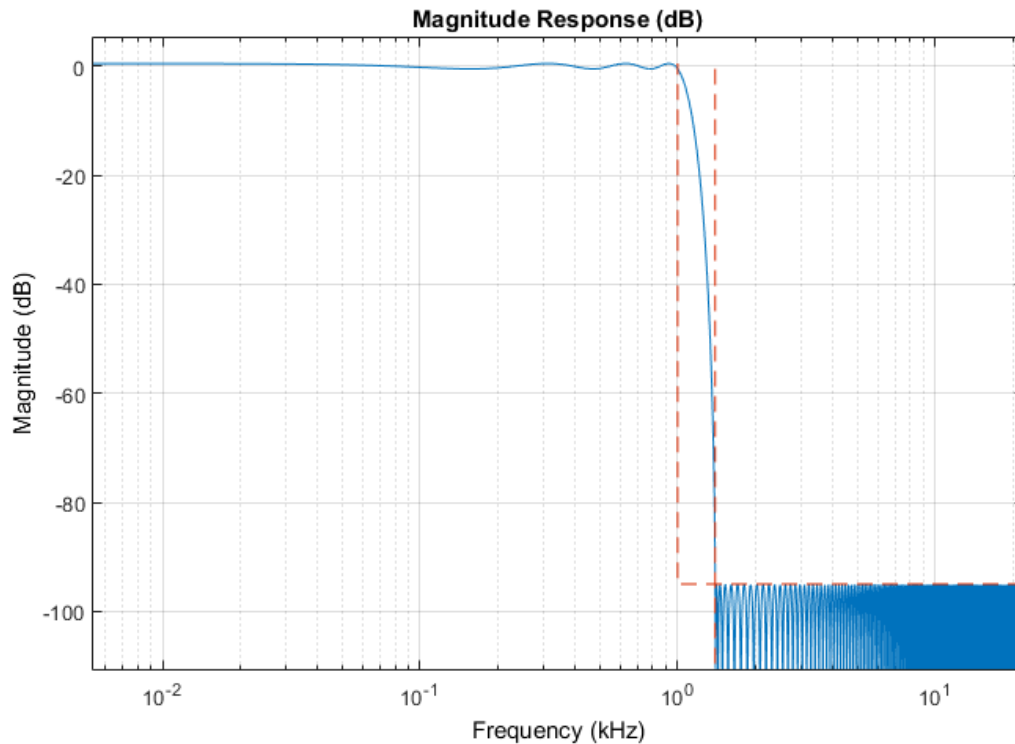


```
% Filter the data and compensate for delay
D = mean(grpdelay(df)); % filter delay
ylp = filter(df,[y; zeros(D,1)]);
ylp = ylp(D+1:end);
```

```
close(hfvt)
```

Look at the spectrum of the lowpass filtered signal. Note how the frequency content above 1400 Hz has been removed.

```
[Plp,Flp] = pwelch(ylp,ones(8192,1),8192/2,8192,Fs,'power');
helperFilterIntroductionPlot1(F,P,Flp,Plp,...
  {'Original signal','Lowpass filtered signal'})
```
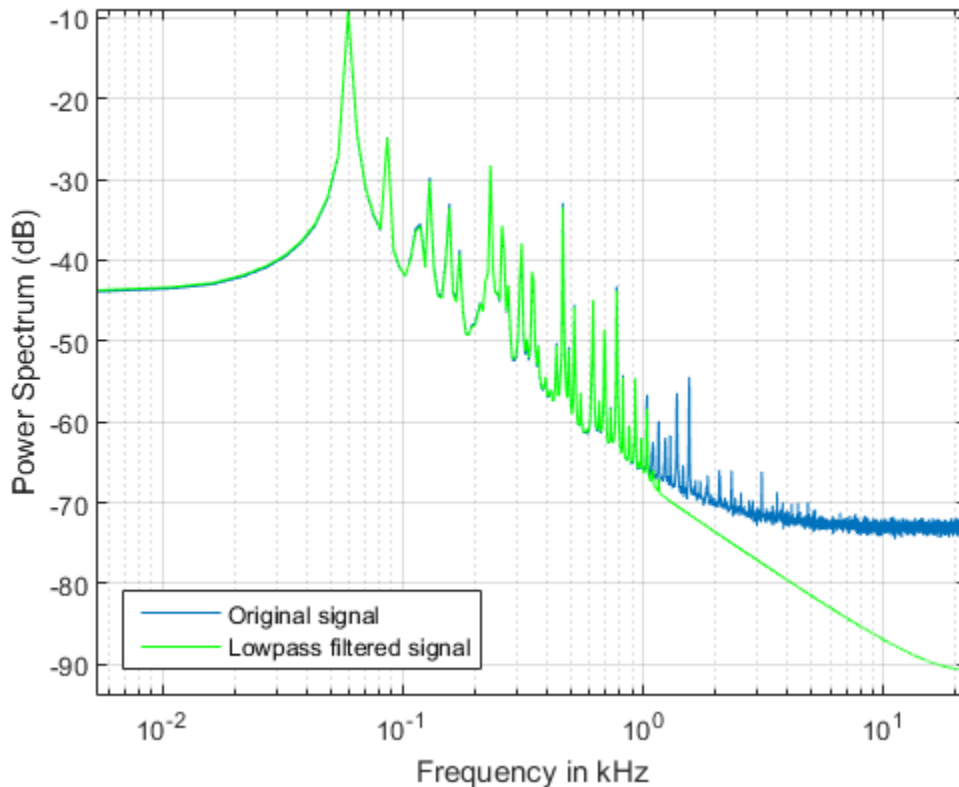

From the power spectrum plot above, you can see that the maximum non-negligible frequency content of the lowpass filtered signal is at 1400 Hz. By the sampling theorem, a sample frequency of 2*1400 = 2800 Hz would suffice to represent the signal correctly, you however, are using a sample rate of 44100 Hz which is a waste since you will need

to process more samples than those necessary. You can downsample the signal to reduce the sample rate and reduce the computational load by reducing the number of samples that you need to process. A lower sample rate will also allow you to design a sharper and narrower bandstop filter, needed to remove the 60 Hz noise, with a smaller filter order.

Downsample the lowpass filtered signal by a factor of 10 to obtain a sample rate of Fs/10 = 4.41 kHz. Plot the spectrum of the signal before and after downsampling.

```
Fs = Fs/10;
yds = downsample(ylp,10);

[Pds,Fds] = pwelch(yds,ones(8192,1),8192/2,8192,Fs,'power');
helperFilterIntroductionPlot1(F,P,Fds,Pds,...
  {'Signal sampled at 44100 Hz', 'Downsampled signal, Fs = 4410 Hz'})
```
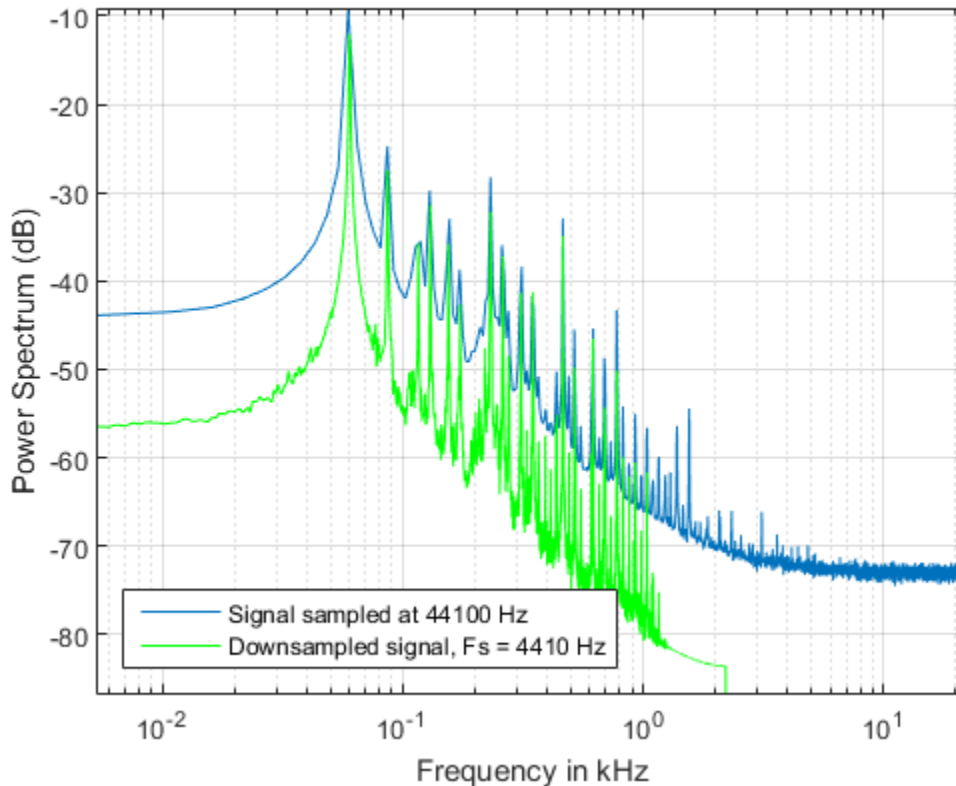
Now remove the 60 Hz tone using an IIR bandstop filter. Let the stopband have a width of 4 Hz centered at 60 Hz. We choose an IIR filter to achieve a sharp frequency notch, small passband ripple, and a relatively low order. Process the data using **filtfilt** to avoid phase distortion.

```matlab
% Design the filter
df = designfilt('bandstopiir','PassbandFrequency1',55,...
                'StopbandFrequency1',58,'StopbandFrequency2',62,...
                'PassbandFrequency2',65,'PassbandRipple1',1,...
                'StopbandAttenuation',60,'PassbandRipple2',1,...
                'SampleRate',Fs,'DesignMethod','ellip');

% Analyze the magnitude response
hfvt = fvtool(df,'Fs',Fs,'FrequencyScale','log',...
  'FrequencyRange','Specify freq. vector','FrequencyVector',Fds(Fds>F(2)));
```
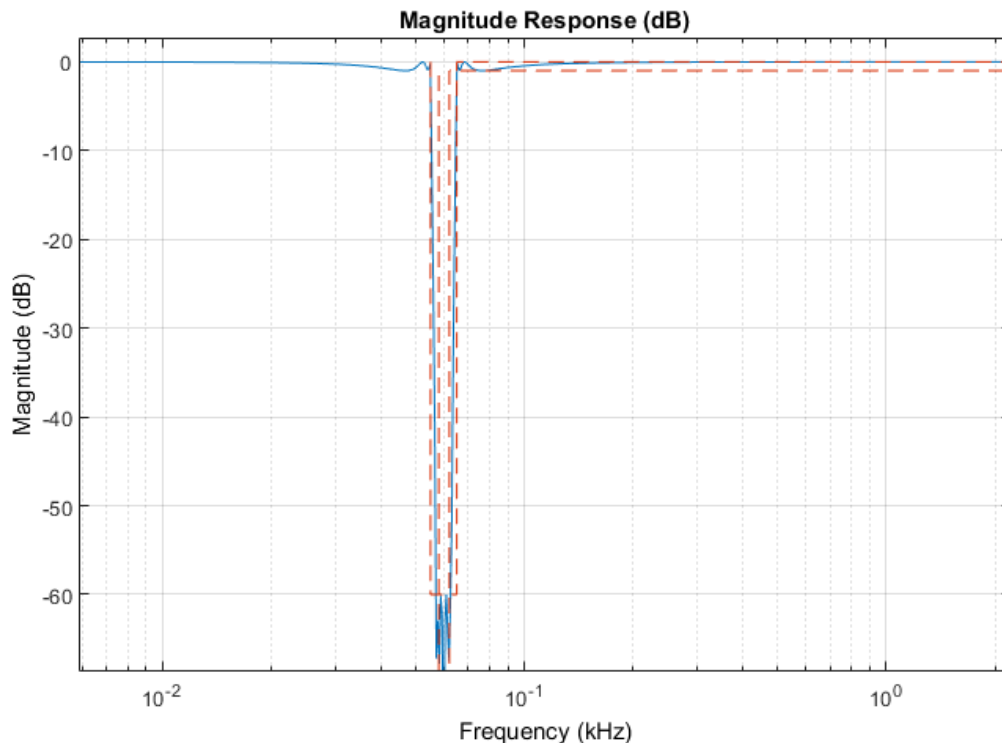
Perform zero-phase filtering to avoid distortion.

```
ybs = filtfilt(df,yds);
```

Finally, upsample the signal to bring it back to the original audio sample rate of 44.1 kHz which is compatible with audio soundcards.

```
yf = interp(ybs,10);
Fs = Fs*10;
```

Take a final look at the spectrum of the original and processed signals. Notice how the high frequency noise floor and the 60 Hz tone have been attenuated by the filters.

```
[Pfinal,Ffinal] = pwelch(yf,ones(8192,1),8192/2,8192,Fs,'power');
close(hfvt)
helperFilterIntroductionPlot1(F,P,Ffinal,Pfinal,...
  {'Original signal','Final filtered signal'})
```
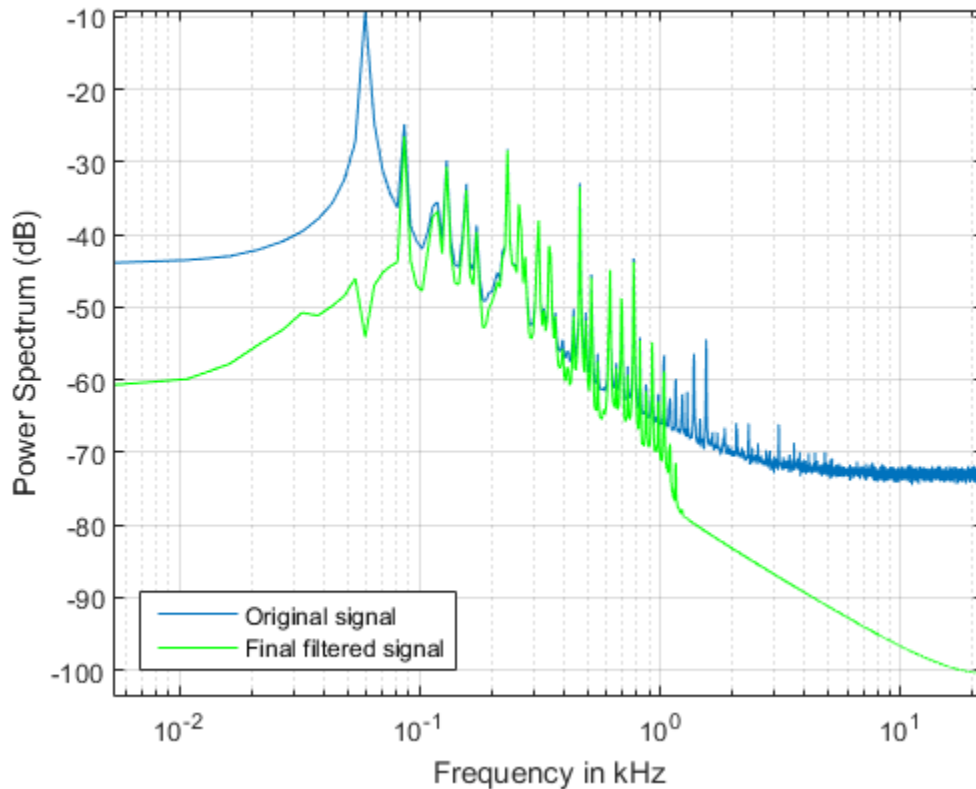
Listen to the signal before and after processing. As mentioned above, the end result is that you have effectively attenuated the 60 Hz hum and the high frequency noise on the audio file.

```
% Play the original signal
hplayer = audioplayer(y, Fs);
play(hplayer);

% Play the noise-reduced signal
hplayer = audioplayer(yf, Fs);
play(hplayer);
```

Differentiating a Signal

The MATLAB **diff** function differentiates a signal with the drawback that you can potentially increase the noise levels at the output. A better option is to use a differentiator filter that acts as a differentiator in the band of interest, and as an attenuator at all other frequencies, effectively removing high frequency noise.

As an example, analyze the speed of displacement of a building floor during an earthquake. Displacement or drift measurements were recorded on the first floor of a three story test structure under earthquake conditions and saved in the quakedrift.mat file. The length of the data vector is 10e3, the sample rate is 1 kHz, and the units of the measurements are cm.

Differentiate the displacement data to obtain estimates of the speed and acceleration of the building floor during the earthquake. Compare the results using diff and an FIR differentiator filter.

```matlab
load quakedrift.mat

Fs  = 1000;                % sample rate
dt = 1/Fs;                 % time differential
t = (0:length(drift)-1)*dt; % time vector
```

Design a 50th order differentiator filter with a passband frequency of 100 Hz which is the bandwidth over which most of the signal energy is found. Set the stopband frequency of the filter to 120 Hz.

```matlab
df = designfilt('differentiatorfir','FilterOrder',50,...
                'PassbandFrequency',100,'StopbandFrequency',120,...
                'SampleRate',Fs);
```

The **diff** function can be seen as a first order FIR filter with response $H(Z) = 1 - Z^{-1}$. Use FVTool to compare the magnitude response of the 50th order differentiator FIR filter and the response of the **diff** function. Clearly, both responses are equivalent in the passband region (from 0 to 100 Hz). However, in the stopband region, the 50th order filter attenuates components while the diff response amplifies components. This effectively increases the levels of high frequency noise.

```matlab
hfvt = fvtool(df,[1 -1],1,'magnitudedisplay','zero-phase','Fs',Fs);
legend(hfvt,'50th order FIR differentiator','Response of diff function');
```
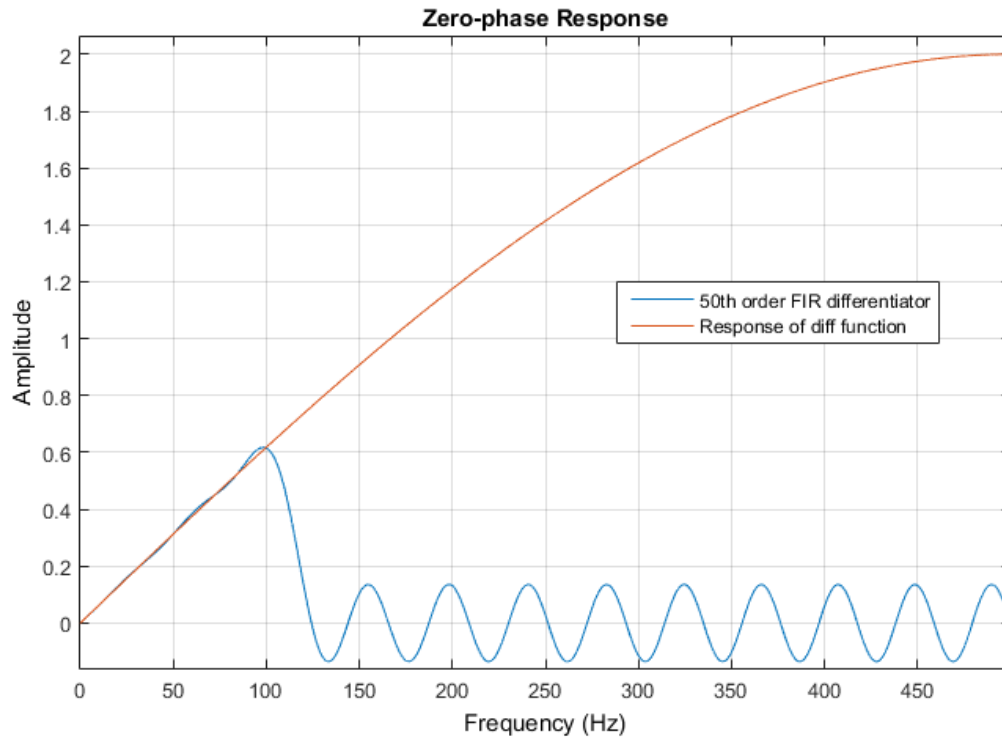
Differentiate using the **diff** function. Add zeros to compensate for the missing samples due to the diff operation.

```
v1 = diff(drift)/dt;
a1 = diff(v1)/dt;

v1 = [0; v1];
a1 = [0; 0; a1];
```

Differentiate using the 50th order FIR filter and compensate for delay.

```
D = mean(grpdelay(df)); % filter delay
v2 = filter(df,[drift; zeros(D,1)]);
v2 = v2(D+1:end);
a2 = filter(df,[v2; zeros(D,1)]);
a2 = a2(D+1:end);
```

2-83

```
v2 = v2/dt;
a2 = a2/dt^2;
```

Plot a few data points of the floor displacement. Plot also a few data points of the speed and acceleration as computed with diff and with the 50th order FIR filter. Notice how the noise has been slightly amplified in the speed estimates and largely amplified in the acceleration estimates obtained with **diff**.

```
helperFilterIntroductionPlot2(t,drift,v1,v2,a1,a2)
```
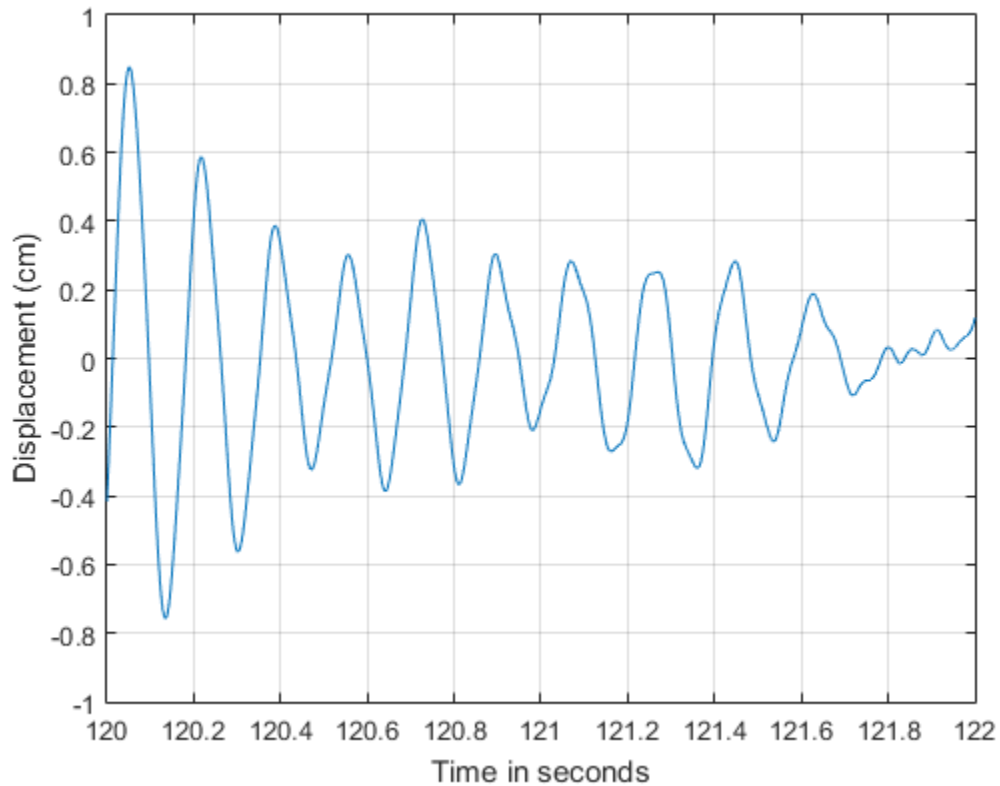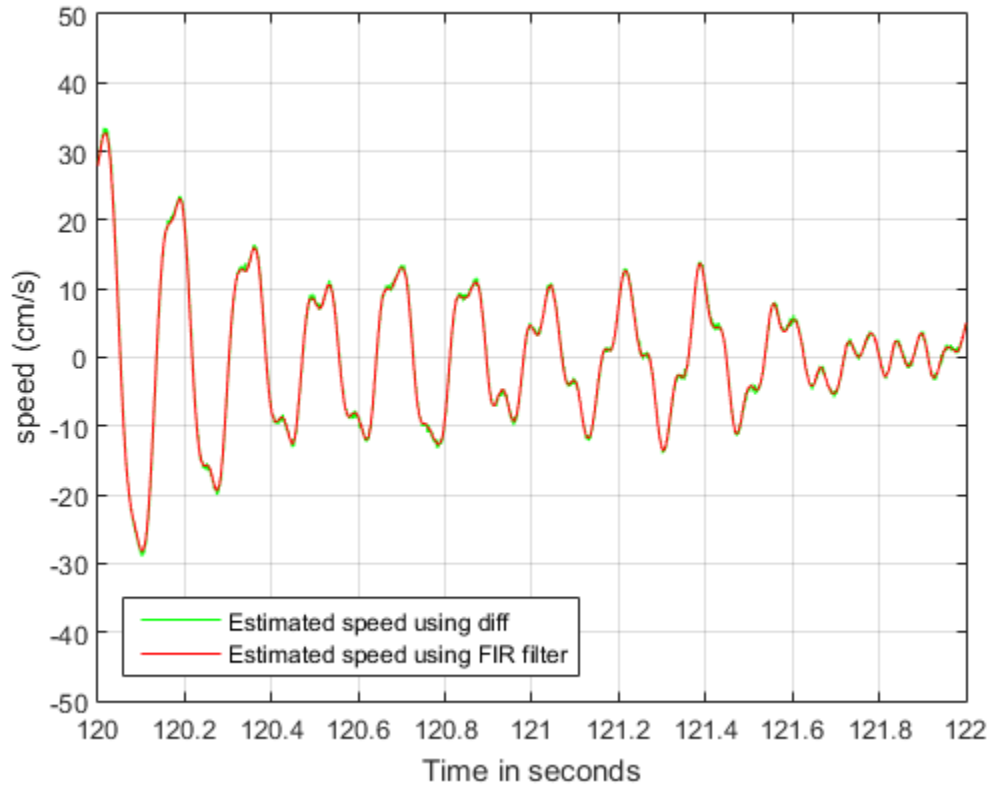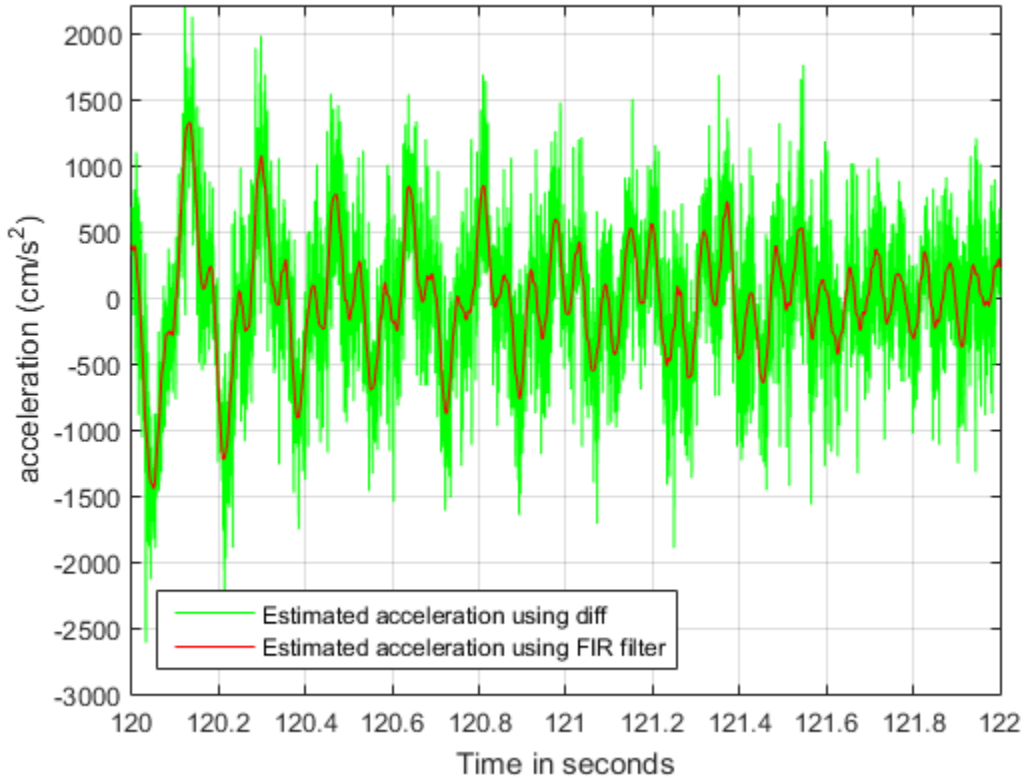
Integrating a Signal

A leaky integrator filter is an all-pole filter with transfer function $H(Z) = 1/[1 - cZ^{-1}]$ where c is a constant that must be smaller than 1 to ensure stability of the filter. It is no surprise that as c approaches one, the leaky integrator approaches the inverse of the **diff** transfer function. Apply the leaky integrator to the acceleration and speed estimates obtained in the previous section to get back the speed and the drift respectively. Use the estimates obtained with the **diff** function since they are noisier.

Use a leaky integrator with $a = 0.999$. Plot the magnitude response of the leaky integrator filter. Notice that the filter acts as a lowpass filter effectively eliminating high frequency noise.

```
close(hfvt)
fvtool(1,[1 -.999],'Fs',Fs)
```
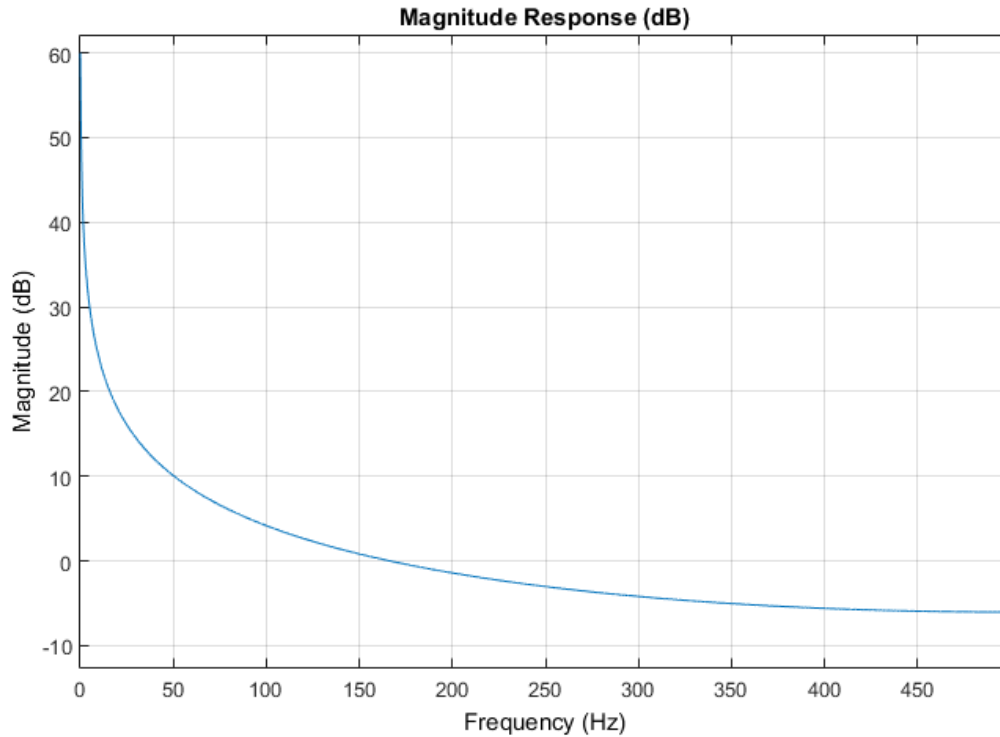

Filter the velocity and acceleration with the leaky integrator.

```
v_original = v1;
a_original = a1;

d_leakyint = filter(1,[1 -0.999],v_original);
v_leakyint = filter(1,[1 -0.999],a_original);

% Multiply by time differential
d_leakyint = d_leakyint * dt;
v_leakyint = v_leakyint * dt;
```

Plot the displacement and speed estimates and compare to the original signals v1 and a1.

```
helperFilterIntroductionPlot3(t,drift,d_leakyint,v_original,v_leakyint)
```
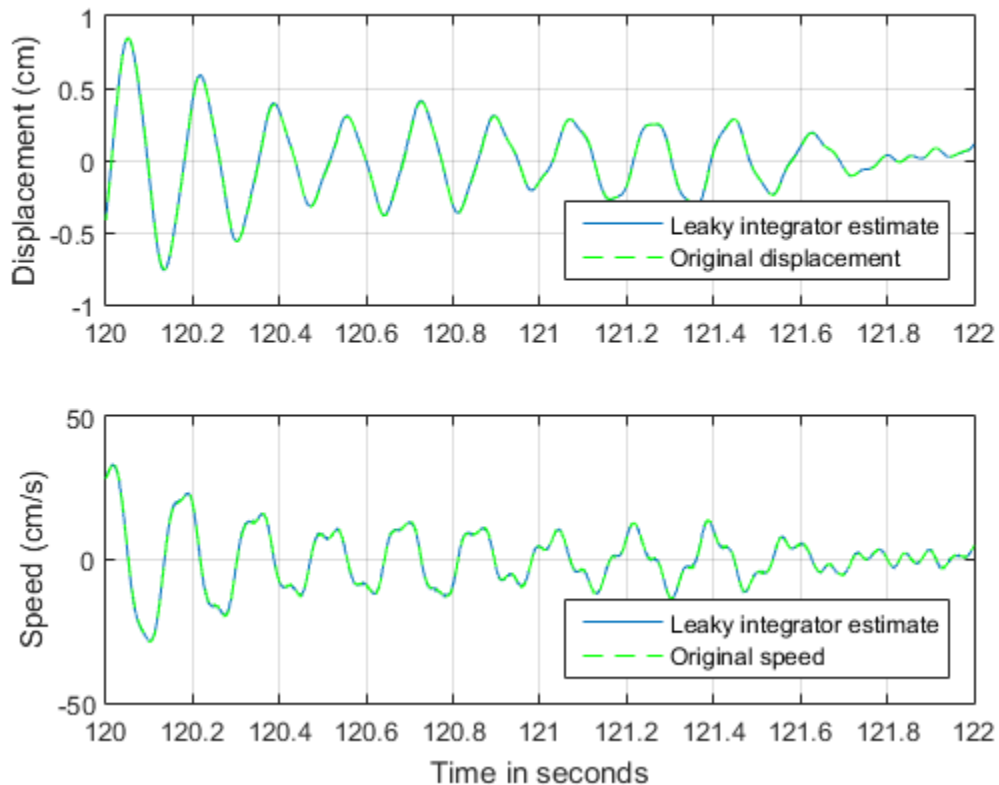

You can also integrate a signal using the **cumsum** and **cumtrapz** functions. Results will be similar to those obtained with the leaky integrator.

Conclusions

In this example you learned about linear and nonlinear phase filters and you learned how to compensate for the phase delay introduced by each filter type. You also learned how to apply filters to remove unwanted frequency components from a signal, and how to downsample a signal after limiting its bandwidth with a lowpass filter. Finally, you learned how to differentiate and integrate a signal using digital filter designs. Throughout the example you also learned how to use analysis tools to look at the response and group delay of your filters.

Further Reading

For more information on filter applications see the Signal Processing Toolbox. For more information on how to design digital filters see the "Practical Introduction to Digital Filter Design" example.

References: J.G. Proakis and D. G. Manolakis, "Digital Signal Processing. Principles, Algorithms, and Applications", Prentice-Hall, 1996.

S. J. Orfanidis, "Introduction To Signal Processing", Prentice-Hall, 1996.

Appendix

The following helper functions are used in this example.

- helperFilterIntroductionPlot1.m
- helperFilterIntroductionPlot2.m
- helperFilterIntroductionPlot3.m

Practical Introduction to Digital Filter Design

This example shows how to design FIR and IIR filters based on frequency response specifications using the `designfilt` function in the Signal Processing Toolbox® product. The example concentrates on lowpass filters but most of the results apply to other response types as well.

This example focuses on the design of digital filters rather than on their applications. If you want to learn more about digital filter applications see the "Practical Introduction to Digital Filtering" example.

FIR Filter Design

Lowpass Filter Specifications

The ideal lowpass filter is one that leaves unchanged all frequency components of a signal below a designated cutoff frequency, ω_c, and rejects all components above ω_c. Because the impulse response required to implement the ideal lowpass filter is infinitely long, it is impossible to design an ideal FIR lowpass filter. Finite length approximations to the ideal impulse response lead to the presence of ripples in both the passband ($\omega < \omega_c$) and the stopband ($\omega > \omega_c$) of the filter, as well as to a nonzero transition width between passband and stopband.

Both the passband/stopband ripples and the transition width are undesirable but unavoidable deviations from the response of an ideal lowpass filter when approximated with a finite impulse response. These deviations are depicted in the following figure:

Zero-Phase Response
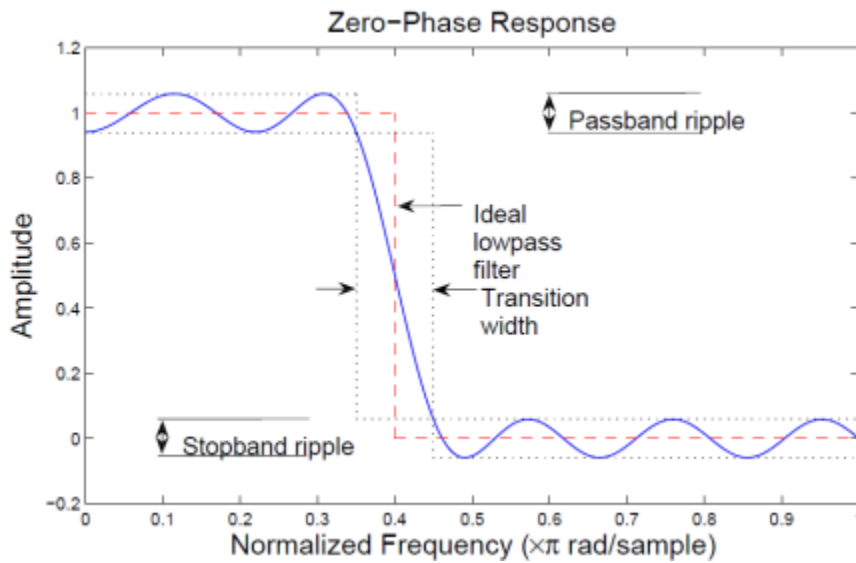
Practical FIR designs typically consist of filters that meet certain design specifications, i.e., that have a transition width and maximum passband and stopband ripples that do not exceed allowable values. In addition, one must select the filter order, or ,equivalently, the length of the truncated impulse response.

A useful metaphor for the design specifications in filter design is to think of each specification as one of the angles in the triangle shown in the figure below.
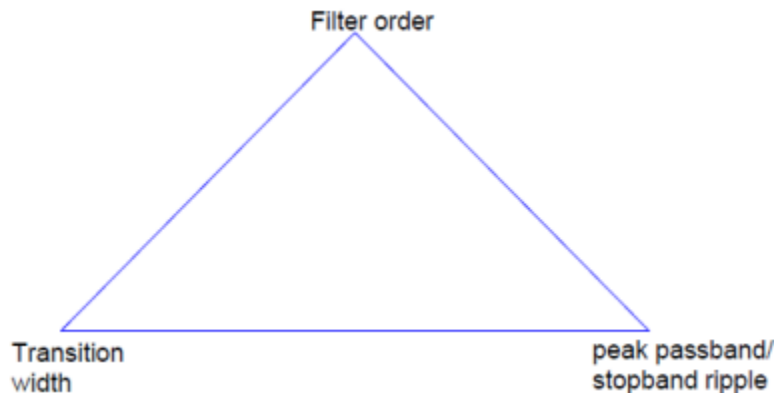
The triangle is used to understand the degrees of freedom available when choosing design specifications. Because the sum of the angles is fixed, one can at most select the values of two of the specifications. The third specification will be determined by the particular design algorithm. Moreover, as with the angles in a triangle, if we make one of the specifications larger/smaller, it will impact one or both of the other specifications.

FIR filters are very attractive because they are inherently stable and can be designed to have linear phase. Nonetheless, these filters can have long transient responses and might prove computationally expensive in certain applications.

Minimum Order FIR Designs

Minimum order designs are obtained by specifying passband and stopband frequencies as well as a passband ripple and a stopband attenuation. The design algorithm then chooses the minimum filter length that complies with the specifications.

Design a minimum order lowpass FIR filter with a passband frequency of 0.37*pi rad/ sample, a stopband frequency of 0.43*pi rad/sample (hence the transition width equals 0.06*pi rad/sample), a passband ripple of 1 dB and a stopband attenuation of 30 dB.

```
Fpass = 0.37;
Fstop = 0.43;
Ap = 1;
Ast = 30;

d = designfilt('lowpassfir','PassbandFrequency',Fpass,...
   'StopbandFrequency',Fstop,'PassbandRipple',Ap,'StopbandAttenuation',Ast);

hfvt = fvtool(d);
```
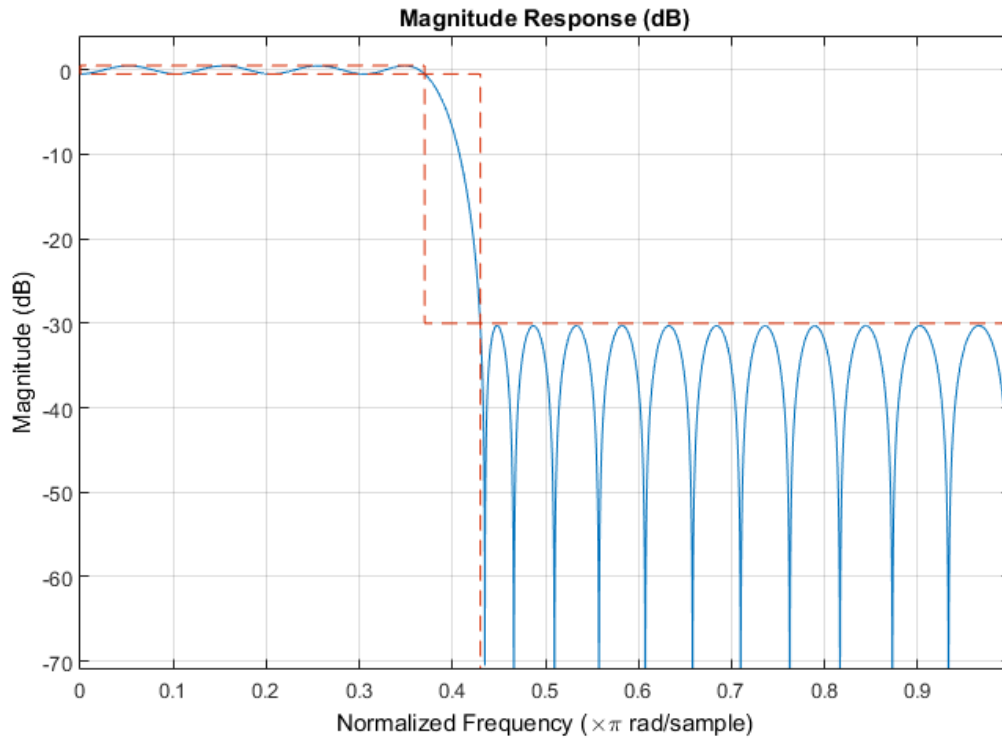
The resulting filter order can be queried using the `filtord` function.

```
N = filtord(d)
```

```
N =

    39
```

You can use the `info` function to get information about the parameters used to design
the filter

```
info(d)
```

```
ans =
```

```
FIR Digital Filter (real)
-------------------------
Filter Length  : 40
Stable         : Yes
Linear Phase   : Yes (Type 2)

Design Method Information
Design Algorithm : Equiripple

Design Specifications
Sample Rate      : 2 (normalized)
Response         : Lowpass
Passband Edge    : 0.37
Stopband Edge    : 0.43
Passband Ripple  : 1 dB
Stopband Atten.  : 30 dB
```

The *designfilt* function chooses an equiripple design algorithm by default. Linear-phase equiripple filters are desirable because for a given order they have the smallest possible maximum deviation from the ideal filter.

Note, however, that minimum-order designs can also be obtained using a Kaiser window. Even though the Kaiser window method yields a larger filter order for the same specifications, the algorithm is less computationally expensive and less likely to have convergence issues when the design specifications are very stringent. This may occur if the application requires a very narrow transition width or a very large stopband attenuation.

Design a filter with the same specifications as above using the Kaiser window method and compare its response to the equiripple filter.

```
dk = designfilt('lowpassfir','PassbandFrequency',Fpass,...
    'StopbandFrequency',Fstop,'PassbandRipple',Ap,...
    'StopbandAttenuation',Ast, 'DesignMethod', 'kaiserwin');

addfilter(hfvt,dk);
legend(hfvt,'Equiripple design', 'Kaiser window design')

N = filtord(dk)
```

```
N =

    52
```


Specifying Frequency Parameters in Hertz

If you know the sample rate at which the filter will operate, you can specify the sample rate and the frequencies in hertz. Redesign the minimum order equiripple filter for a sample rate of 2 kHz.

```
Fpass = 370;
Fstop = 430;
Ap = 1;
Ast = 30;
Fs = 2000;
```

```
d = designfilt('lowpassfir','PassbandFrequency',Fpass,...
  'StopbandFrequency',Fstop,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs);

hfvt = fvtool(d);
```
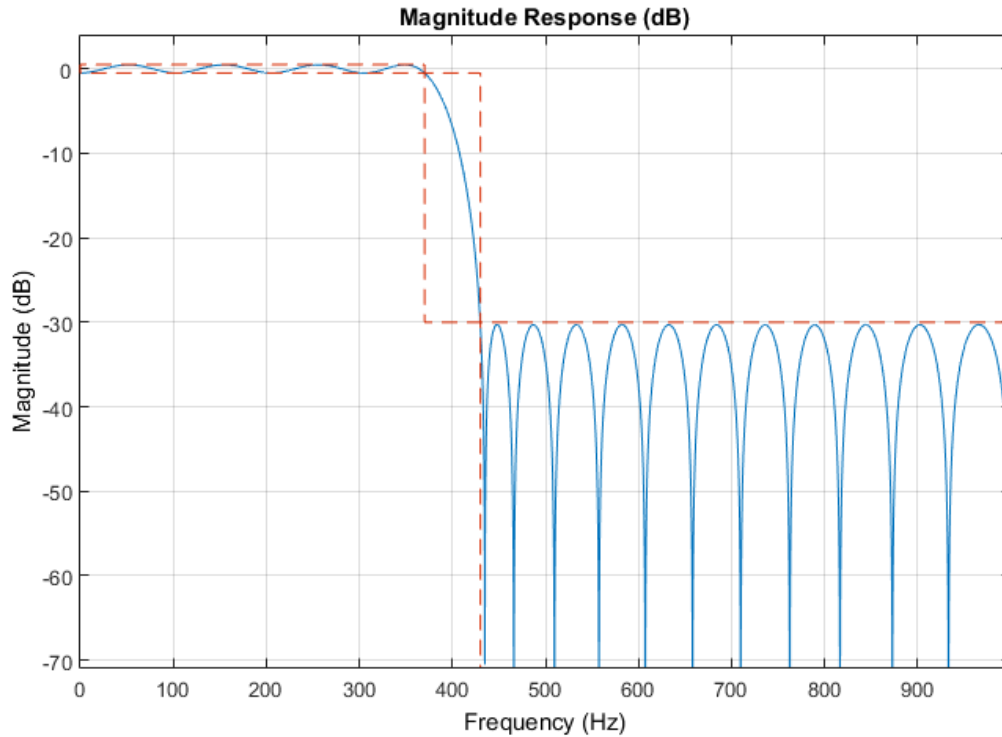

Fixed Order, Fixed Transition Width Designs

Fixed-order designs are useful for applications that are sensitive to computational load or impose a limit on the number of filter coefficients. An option is to fix the transition width at the expense of control over the passband ripple/stopband attenuation.

Consider a 30-th order lowpass FIR filter with a passband frequency of 370 Hz, a stopband frequency of 430 Hz, and sample rate of 2 kHz. There are two design methods available for this particular set of specifications: equiripple and least squares. Let us design one filter for each method and compare the results.

```
N = 30;
Fpass = 370;
Fstop = 430;
Fs = 2000;

% Design method defaults to 'equiripple' when omitted
deq = designfilt('lowpassfir','FilterOrder',N,'PassbandFrequency',Fpass,...
    'StopbandFrequency',Fstop,'SampleRate',Fs);

dls = designfilt('lowpassfir','FilterOrder',N,'PassbandFrequency',Fpass,...
    'StopbandFrequency',Fstop,'SampleRate',Fs,'DesignMethod','ls');

hfvt = fvtool(deq,dls);
legend(hfvt,'Equiripple design', 'Least-squares design')
```
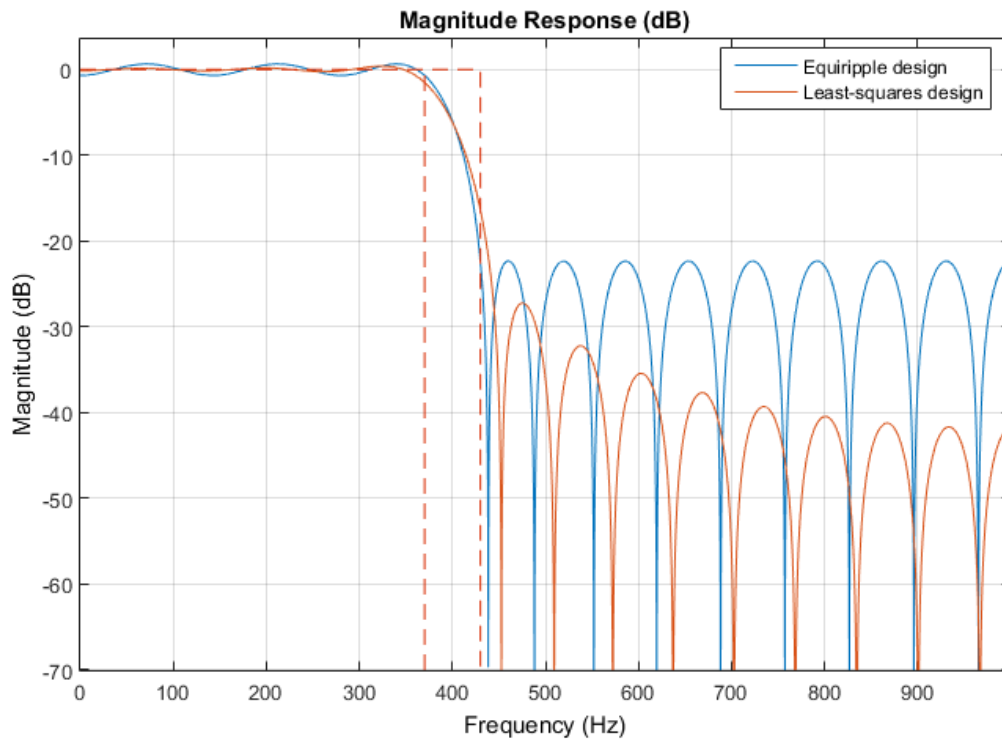
Equiripple filters are ideally suited for applications in which a specific tolerance must be met, such as designing a filter with a given minimum stopband attenuation or a given maximum passband ripple. On the other hand, these designs may not be desirable if we want to minimize the energy of the error (between ideal and actual filter) in the passband/stopband.

> If we want to reduce the energy of a signal as much as possible in a certain frequency band, least squares designs are preferable.

In the examples above, the designed filters had the same ripple in the passband and in the stopband. We can use weights to reduce the ripple in one of the bands while keeping the filter order fixed. For example, if you wish the stopband ripple to be a tenth of that in the passband, you must give the stopband ten times the passband weight. Redesign the equiripple filter using that fact.

```
deqw = designfilt('lowpassfir','FilterOrder',N,'PassbandFrequency',Fpass,...
    'StopbandFrequency',Fstop,'SampleRate',Fs,...
    'PassbandWeight',1,'StopbandWeight',10);

hfvt = fvtool(deq,deqw);
legend(hfvt,'Equiripple design', 'Equiripple design with weighted stopband')
```
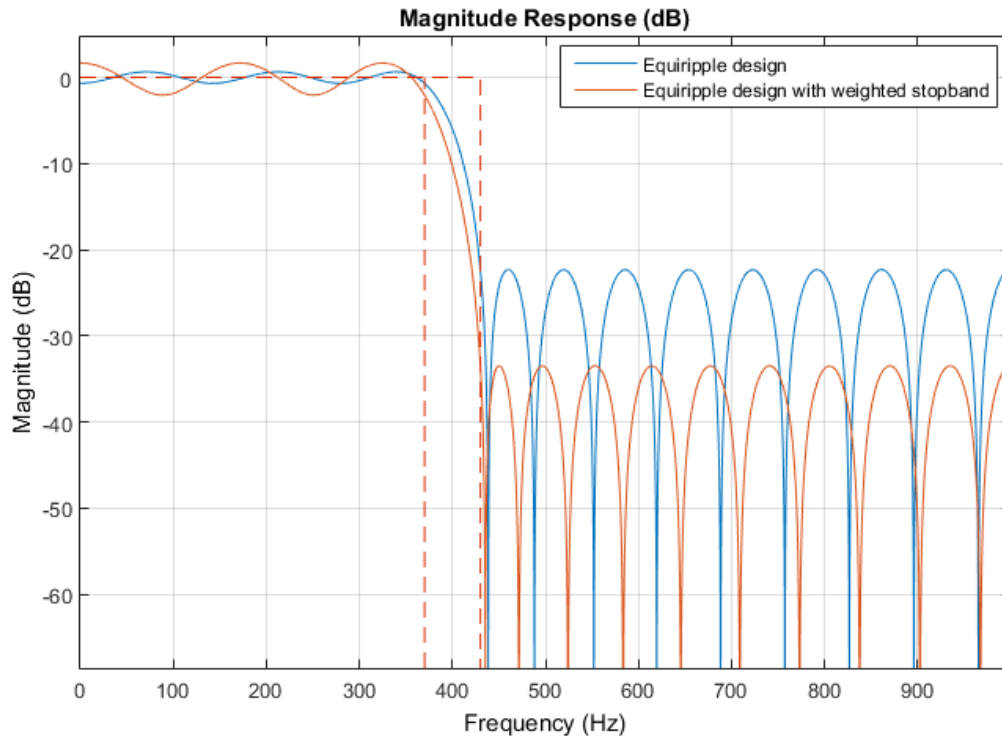
Fixed Order, Fixed Cutoff Frequency

You can design filters with fixed filter order and cutoff frequency using a window design method.

> You can use different windows to control the stopband attenuation while keeping the filter order unchanged.

For example, consider a 100-th order lowpass FIR filter with a cutoff frequency of 60 Hz and a sample rate of 1 kHz. Compare designs that result from using a Hamming window, and a Chebyshev window with 90 dB of sidelobe attenuation.

```
dhamming = designfilt('lowpassfir','FilterOrder',100,'CutoffFrequency',60,...
  'SampleRate',1000,'Window','hamming');
```

```
dchebwin = designfilt('lowpassfir','FilterOrder',100,'CutoffFrequency',60,...
   'SampleRate',1000,'Window',{'chebwin',90});

hfvt = fvtool(dhamming,dchebwin);
legend(hfvt,'Hamming window', 'Chebyshev window')
```
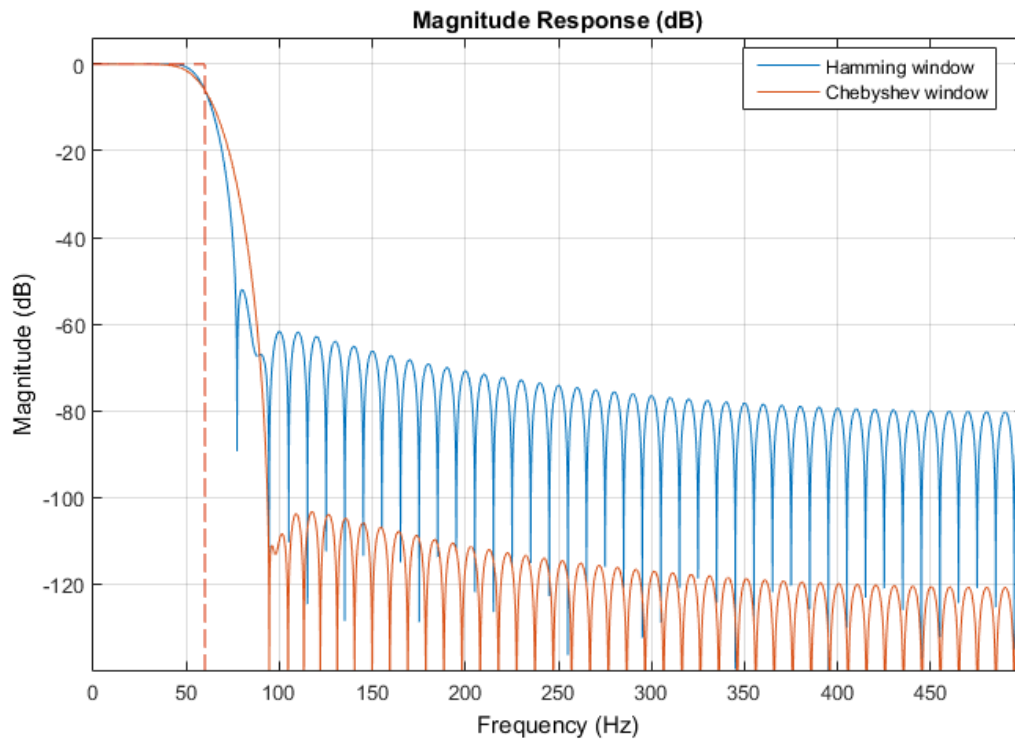

There are other ways in which you can specify a filter with fixed order: fixed cutoff frequency, passband ripple, and stopband attenuation; fixed transition width; and fixed half-power (3dB) frequency.

IIR Filter Design

One of the drawbacks of FIR filters is that they require a large filter order to meet some design specifications. If the ripples are kept constant, the filter order grows inversely

proportional to the transition width. By using feedback, it is possible to meet a set of design specifications with a far smaller filter order. This is the idea behind IIR filter design. The term "infinite impulse response" (IIR) stems from the fact that, when an impulse is applied to the filter, the output never decays to zero.

IIR filters tend to be used when computational resources are at a premium. However, stable, causal IIR filters cannot have perfectly linear phase. IIR filters thus tend to be avoided when linearity of phase is a requirement.

Another important reason for using IIR filters is their small group delay relative to FIR filters, which results in a shorter transient response.

Butterworth Filters

Butterworth filters are maximally flat IIR filters. The flatness in the passband and stopband causes the transition band to be very wide. Large orders are required to obtain filters with narrow transition widths.

Design a minimum order Butterworth filter with passband frequency 100 Hz, stopband frequency 300 Hz, maximum passband ripple 1 dB, and 60 dB stopband attenuation. The sample rate is 2 kHz.

```
Fp = 100;
Fst = 300;
Ap = 1;
Ast = 60;
Fs = 2e3;

dbutter = designfilt('lowpassiir','PassbandFrequency',Fp,...
  'StopbandFrequency',Fst,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs,'DesignMethod','butter');
```

Chebyshev Type I Filters

Chebyshev type I filters attain smaller transition widths than Butterworth filters of the same order by allowing for passband ripple.

The stopband of a Chebyshev type I filter is, as with Butterworth filters, maximally flat. For a given filter order, the tradeoff is thus between passband ripple and transition width.

2-101

Design a Chebyshev type I filter with the same specifications as the Butterworth filter above.
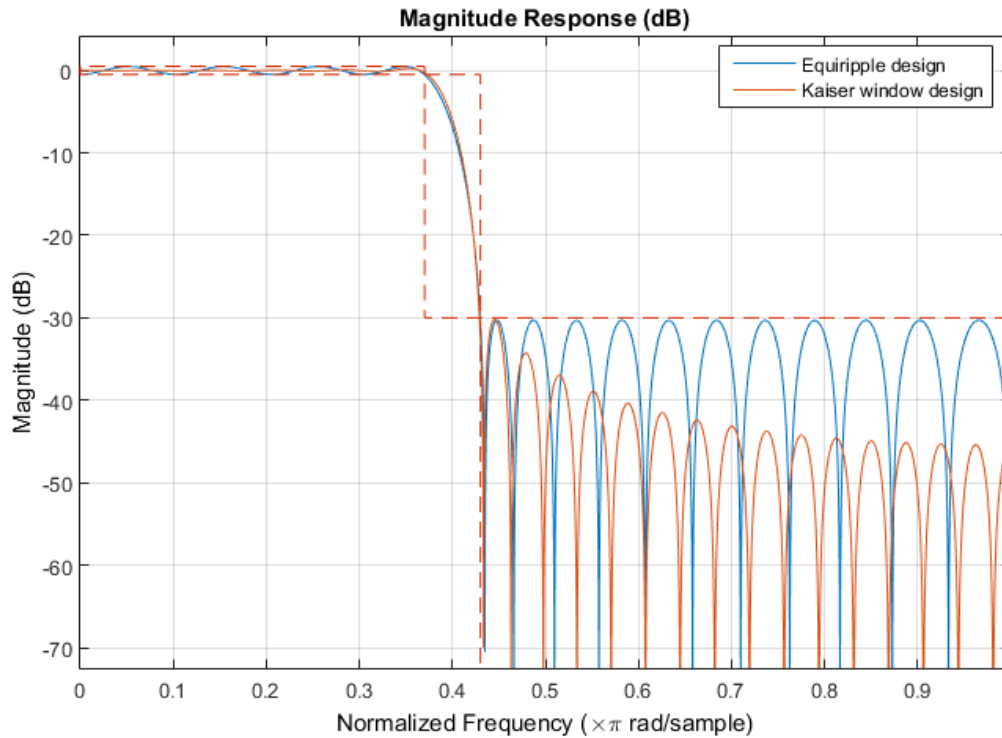
```
dcheby1 = designfilt('lowpassiir','PassbandFrequency',Fp,...
  'StopbandFrequency',Fst,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs,'DesignMethod','cheby1');
```

Chebyshev Type II Filters

Chebyshev type II filters have maximally flat passband and equiripple stopband.

Since extremely large attenuations are typically not required, we may be able to attain the required transition width with a relatively small order by allowing for some stopband ripple.

Design a minimum order Chebyshev type II filter with the same specifications as in the previous examples.

```
dcheby2 = designfilt('lowpassiir','PassbandFrequency',Fp,...
  'StopbandFrequency',Fst,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs,'DesignMethod','cheby2');
```

Elliptic Filters

Elliptic filters generalize Chebyshev and Butterworth filters by allowing for ripple in both the passband and the stopband. As ripples are made smaller, elliptic filters can approximate arbitrarily close the magnitude and phase response of either Chebyshev or Butterworth filters.

The extra degrees of freedom allow elliptic filters to have the smallest order for a given transition width.

```
dellip = designfilt('lowpassiir','PassbandFrequency',Fp,...
  'StopbandFrequency',Fst,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs,'DesignMethod','ellip');
```

Compare the response and the order of the four IIR filters.

For the same specification constraints, the Butterworth method yields the highest order while the elliptic method yields the smallest.

```
FilterOrders = [filtord(dbutter) filtord(dcheby1) filtord(dcheby2) filtord(dellip)]


FilterOrders =

     7     5     5     4


hfvt = fvtool(dbutter,dcheby1,dcheby2,dellip);
axis([0 1e3 -80 2]);
legend(hfvt,'Butterworth', 'Chebyshev type I',...
  'Chebyshev type II','Elliptic', 1)
```
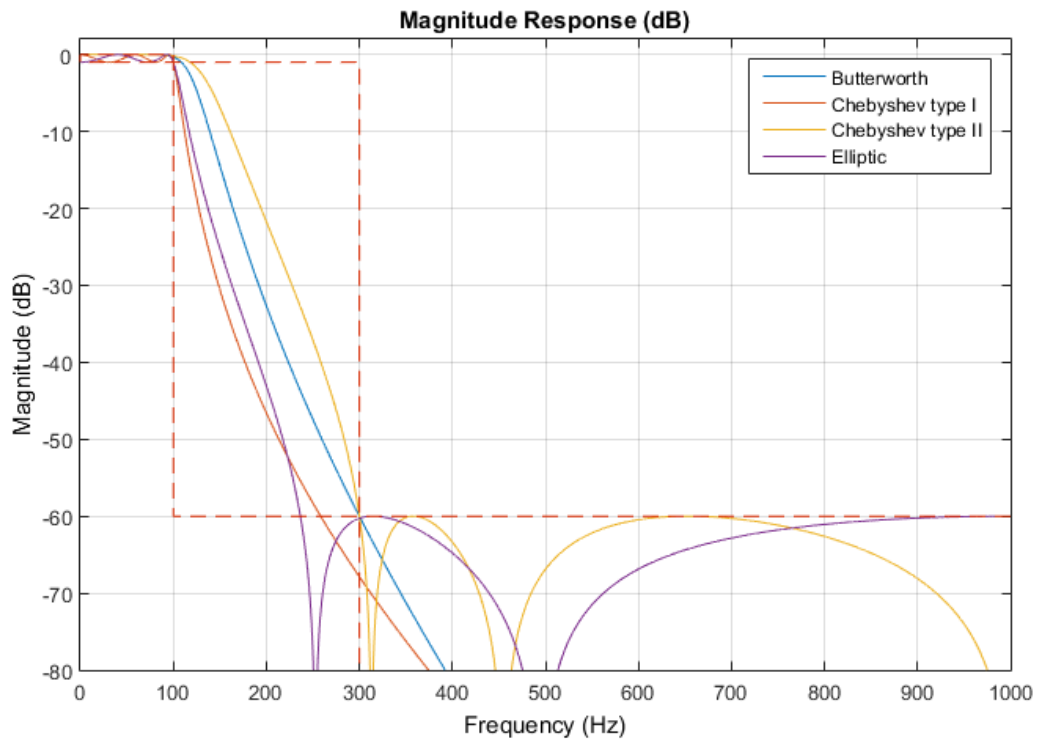

Zoom into the passband to see the ripple differences.

```
axis([0 150 -3 2]);
```
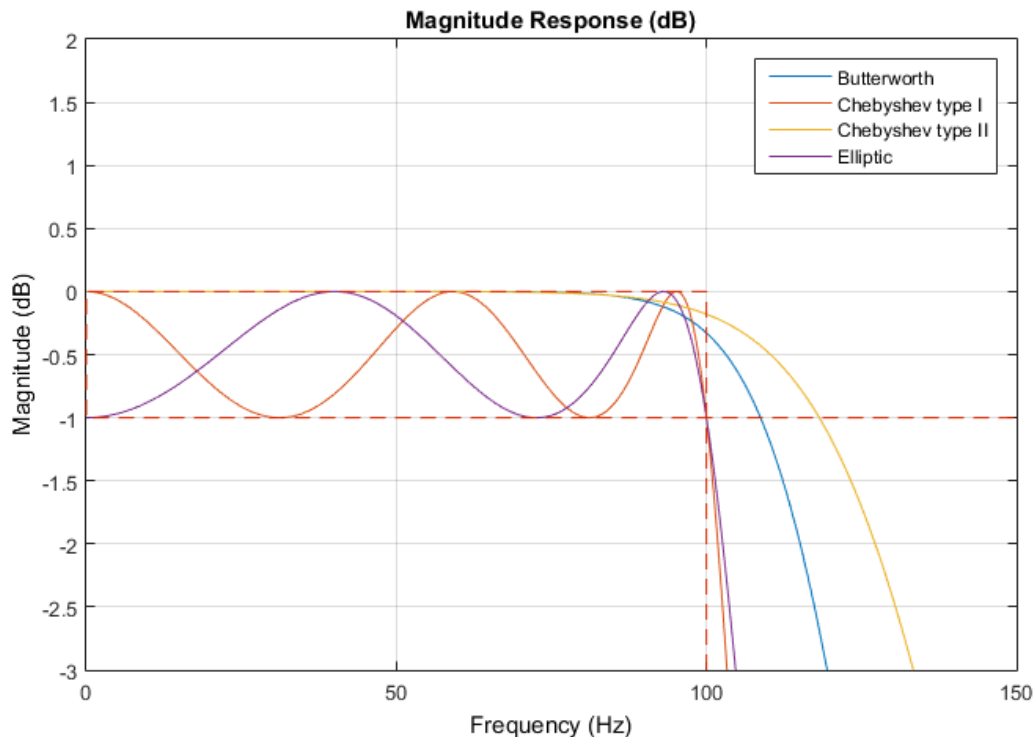
Matching Exactly the Passband or Stopband Specifications

With minimum-order designs, the ideal order needs to be rounded to the next integer. This additional fractional order allows the algorithm to actually exceed the specifications.

> We can use the 'MatchExactly' parameter to constraint the design algorithm to match one band exactly . The other band will exceed its specifications.

By default, Chebyshev Type I designs match the passband, Butterworth and Chebyshev Type II match the stopband, and elliptic designs match both the passband and the stopband (while the stopband edge frequency is exceeded):

```
dellip1 = designfilt('lowpassiir','PassbandFrequency',Fp,...
  'StopbandFrequency',Fst,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs,'DesignMethod','ellip',...
```

```
  'MatchExactly','passband');

dellip2 = designfilt('lowpassiir','PassbandFrequency',Fp,...
  'StopbandFrequency',Fst,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs,'DesignMethod','ellip',...
  'MatchExactly','stopband');

hfvt = fvtool(dellip, dellip1, dellip2);
legend(hfvt,'Matched passband and stopband','Matched passband',...
  'Matched stopband', 1);
axis([0 1e3 -80 2]);
```
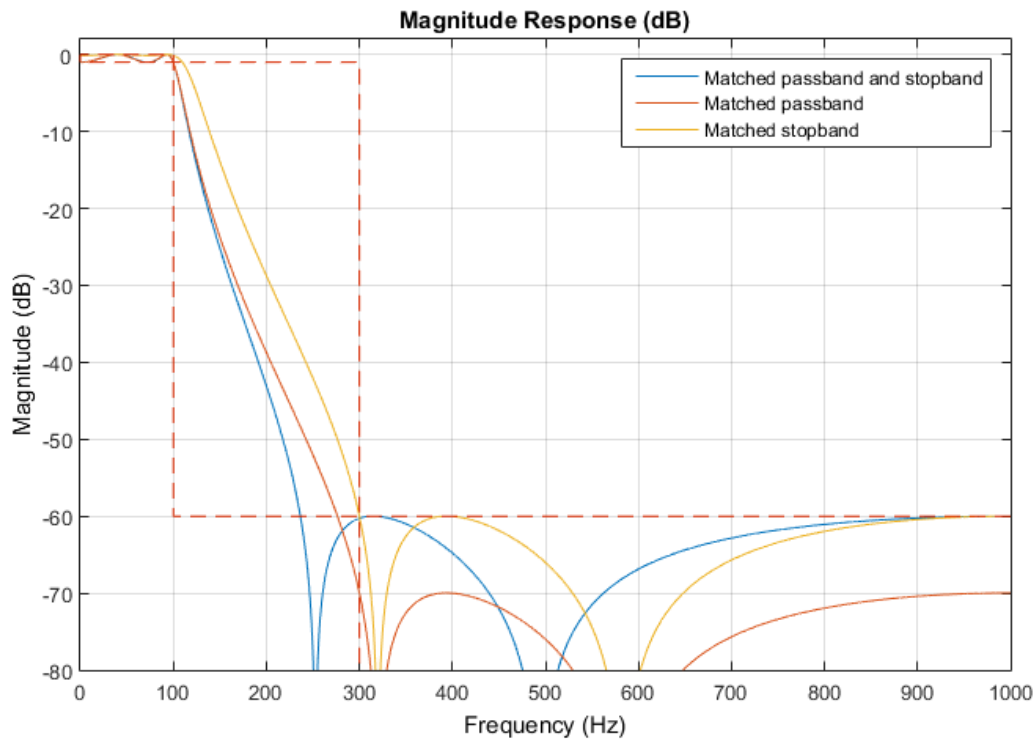

The matched-passband and matched-both designs have a ripple of exactly 1 dB at the passband frequency value of 100 Hz.

Group Delay Comparison

With IIR filters, we need to consider not only the ripple/transition width tradeoff, but also the degree of phase distortion. We know that it is impossible to have linear-phase throughout the entire Nyquist interval. Thus we may want to see how far from linear the phase response is. A good way to do this is to look at the (ideally constant) group delay and see how flat it is."

Compare the group delay of the four IIR filters designed above.

> If phase is an issue, keep in mind that Butterworth and Chebyshev Type II designs have the flattest group delay and thus introduce the least distortion.

```
hfvt = fvtool(dbutter,dcheby1,dcheby2,dellip,'Analysis','grpdelay');
legend(hfvt,'Butterworth', 'Chebyshev type I',...
  'Chebyshev type II','Elliptic', 1)
```
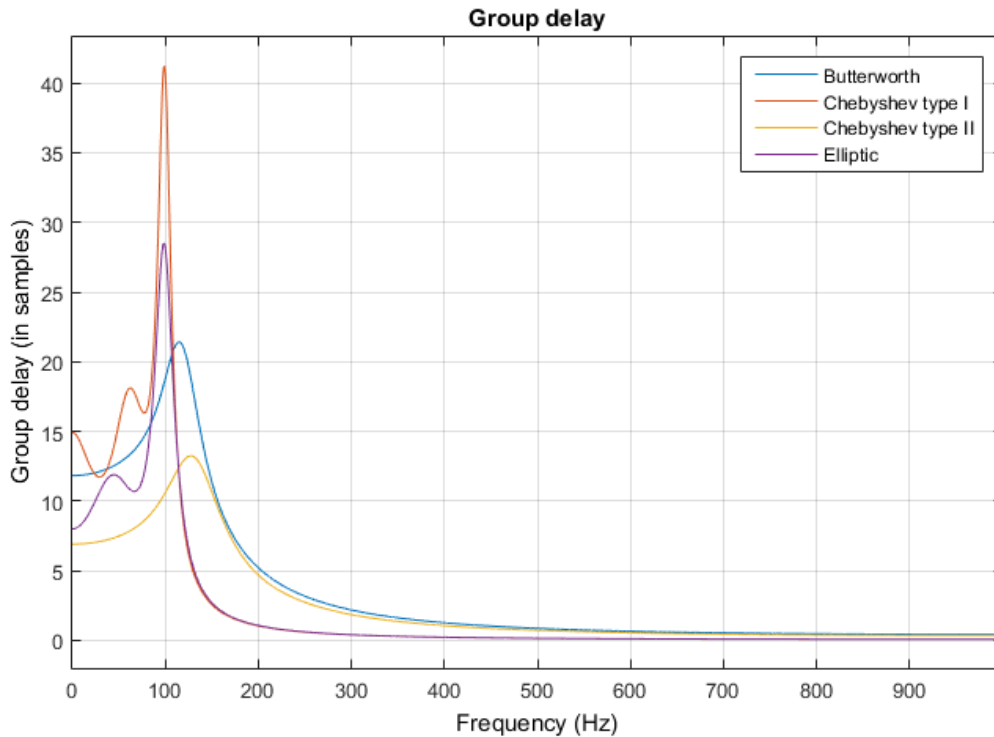
Conclusions

In this example, you learned how to use `designfilt` to obtain a variety of lowpass FIR and IIR filters with different constraints and design methods. `designfilt` can also be used to obtain highpass, bandpass, bandstop, arbitrary-magnitude, differentiator, and Hilbert designs. See the "Filter Design Gallery" example and the documentation to learn more about all the available options.

Further Reading

For more information on filter design and analysis, see the Signal Processing Toolbox® software documentation. For more information on filter applications see the "Practical Introduction to Digital Filtering" example.

Filter Design Gallery

This example shows how to design a variety of FIR and IIR digital filters with the `designfilt` function in the Signal Processing Toolbox® product.

The gallery is designed for you to identify a filter response of interest, view the code, and use it in your own project. It contains examples for each of the available filter responses offered by `designfilt`. Note, however, that these are only a few of the possible ways in which you can design filters for each response type. For an exhaustive list of specification sets, see the Signal Processing Toolbox documentation.

Except when noted otherwise, in this example all frequency units are in hertz, and all ripple and attenuation values are in decibels.

Lowpass FIR Filters

Equiripple Design

```
Fpass = 100;
Fstop = 150;
Apass = 1;
Astop = 65;
Fs = 1e3;

d = designfilt('lowpassfir', ...
  'PassbandFrequency',Fpass,'StopbandFrequency',Fstop, ...
  'PassbandRipple',Apass,'StopbandAttenuation',Astop, ...
  'DesignMethod','equiripple','SampleRate',Fs);

fvtool(d)
```
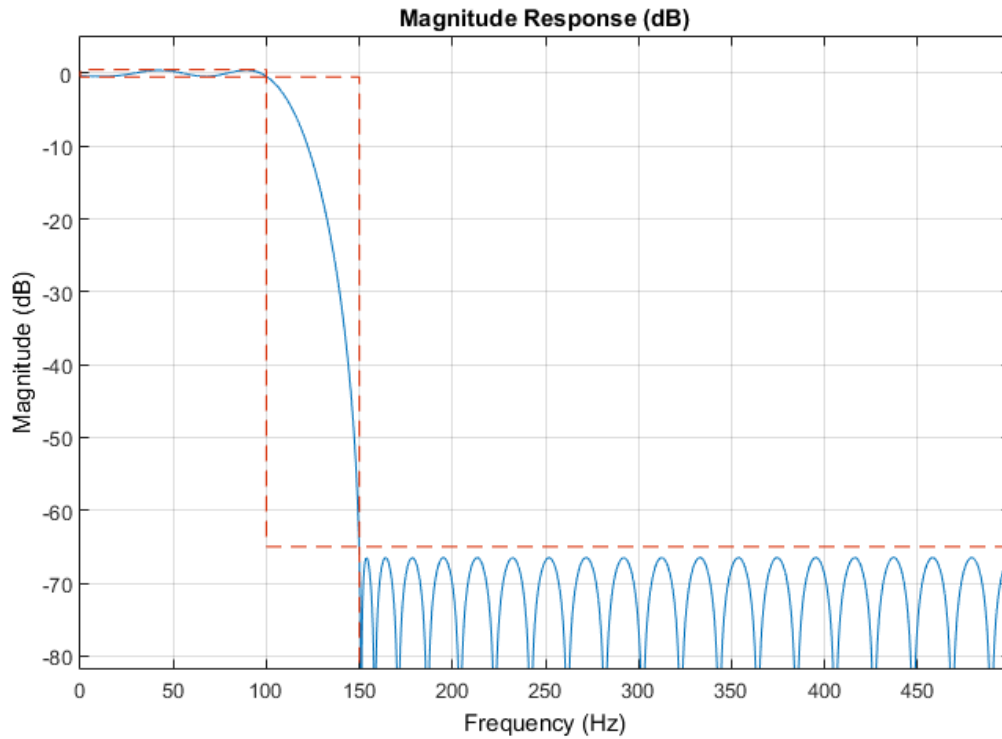
Lowpass IIR Filters

Maximally Flat Design

```
Fpass = 100;
Fstop = 150;
Apass = 0.5;
Astop = 65;
Fs = 1e3;

d = designfilt('lowpassiir', ...
  'PassbandFrequency',Fpass,'StopbandFrequency',Fstop, ...
  'PassbandRipple',Apass,'StopbandAttenuation',Astop, ...
  'DesignMethod','butter','SampleRate',Fs);

fvtool(d)
```

2-109

Magnitude Response (dB)
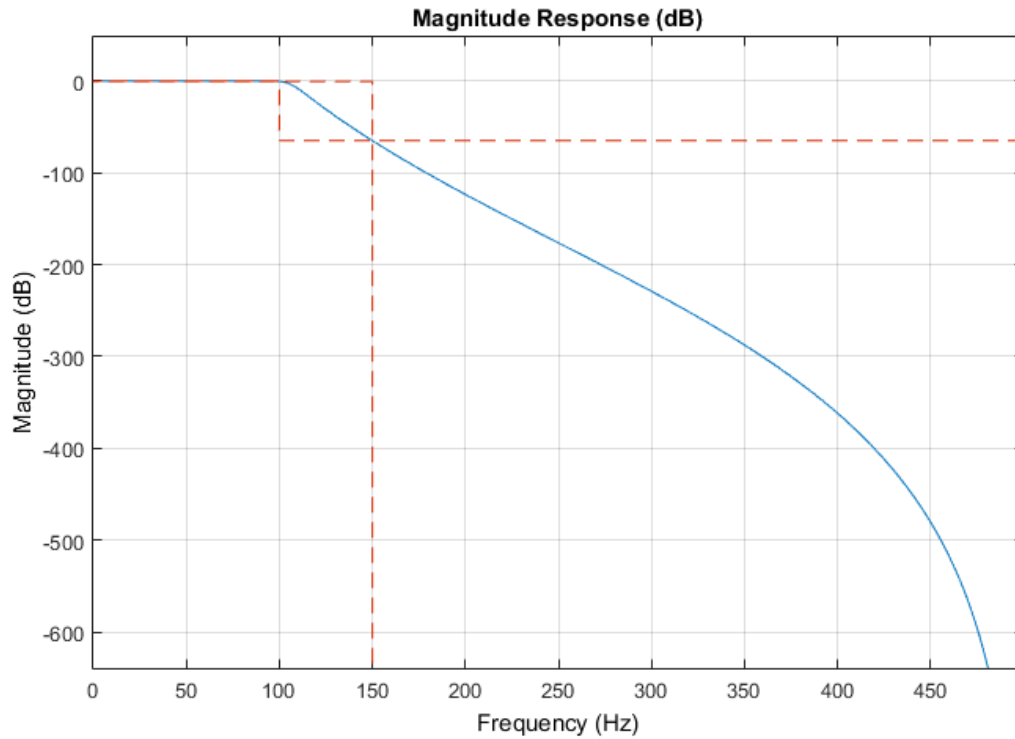
Ripple in Passband and Stopband

```
N = 8;
Fpass = 100;
Apass = 0.5;
Astop = 65;
Fs = 1e3;

d = designfilt('lowpassiir', ...
  'FilterOrder',N, ...
  'PassbandFrequency',Fpass, ...
  'PassbandRipple',Apass,'StopbandAttenuation',Astop, ...
  'SampleRate',Fs);

fvtool(d)
```
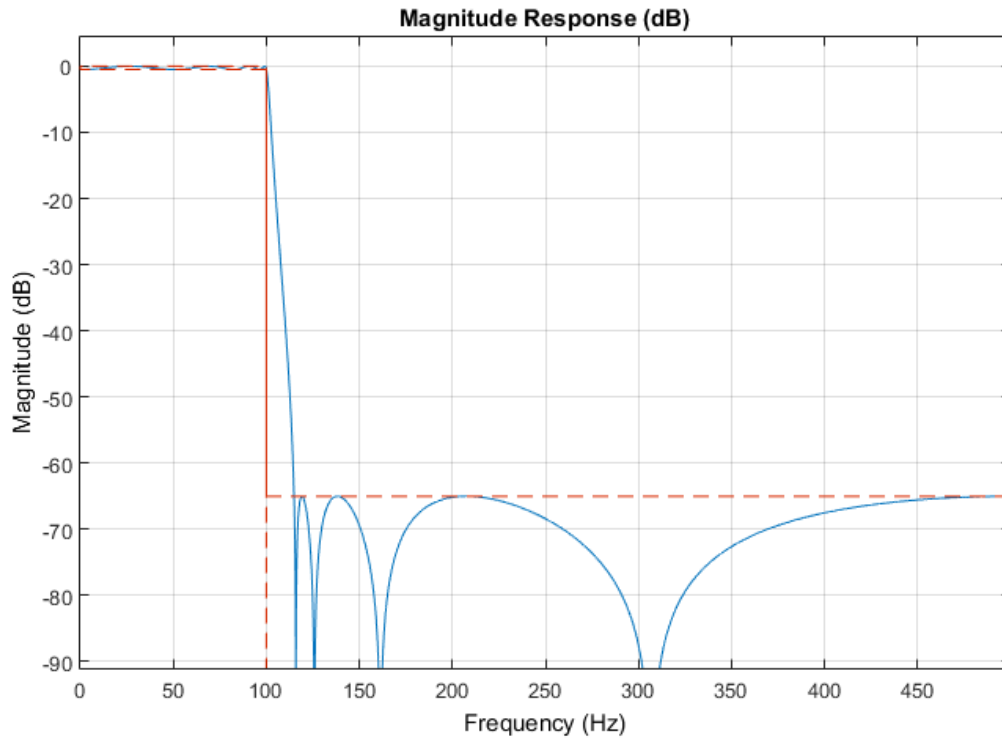
Highpass FIR Filters

Equiripple Design

```
Fstop = 350;
Fpass = 400;
Astop = 65;
Apass = 0.5;
Fs = 1e3;

d = designfilt('highpassfir','StopbandFrequency',Fstop, ...
  'PassbandFrequency',Fpass,'StopbandAttenuation',Astop, ...
  'PassbandRipple',Apass,'SampleRate',Fs,'DesignMethod','equiripple');

fvtool(d)
```
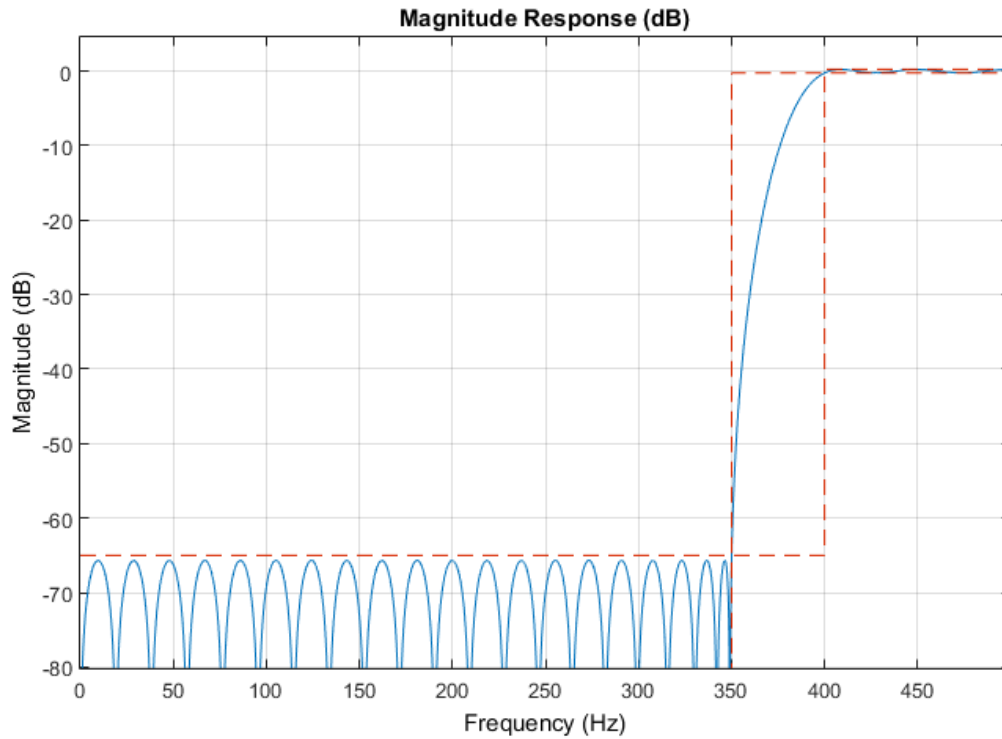
Highpass IIR Filters

Maximally Flat Design

```
Fstop = 350;
Fpass = 400;
Astop = 65;
Apass = 0.5;
Fs = 1e3;

d = designfilt('highpassiir','StopbandFrequency',Fstop ,...
  'PassbandFrequency',Fpass,'StopbandAttenuation',Astop, ...
  'PassbandRipple',Apass,'SampleRate',Fs,'DesignMethod','butter');

fvtool(d)
```
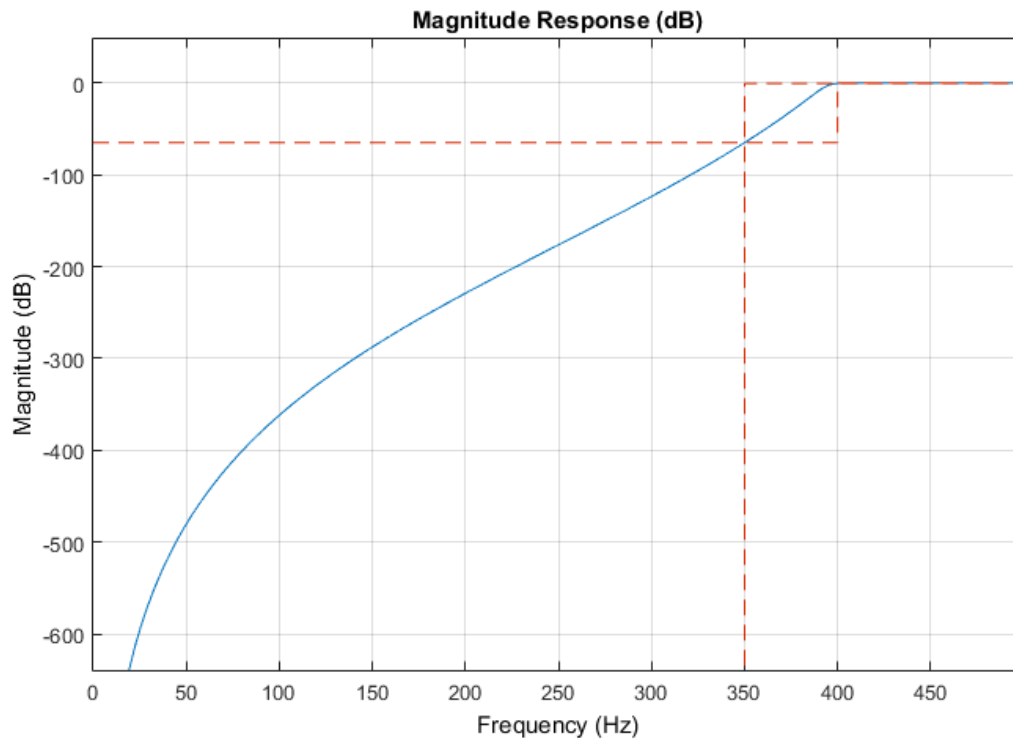
Ripple in Passband and Stopband

```
N = 8;
Fpass = 400;
Astop = 65;
Apass = 0.5;
Fs = 1e3;

d = designfilt('highpassiir', ...
  'FilterOrder',N, ...
  'PassbandFrequency',Fpass, ...
  'StopbandAttenuation',Astop,'PassbandRipple',Apass, ...
  'SampleRate',Fs);

fvtool(d)
```
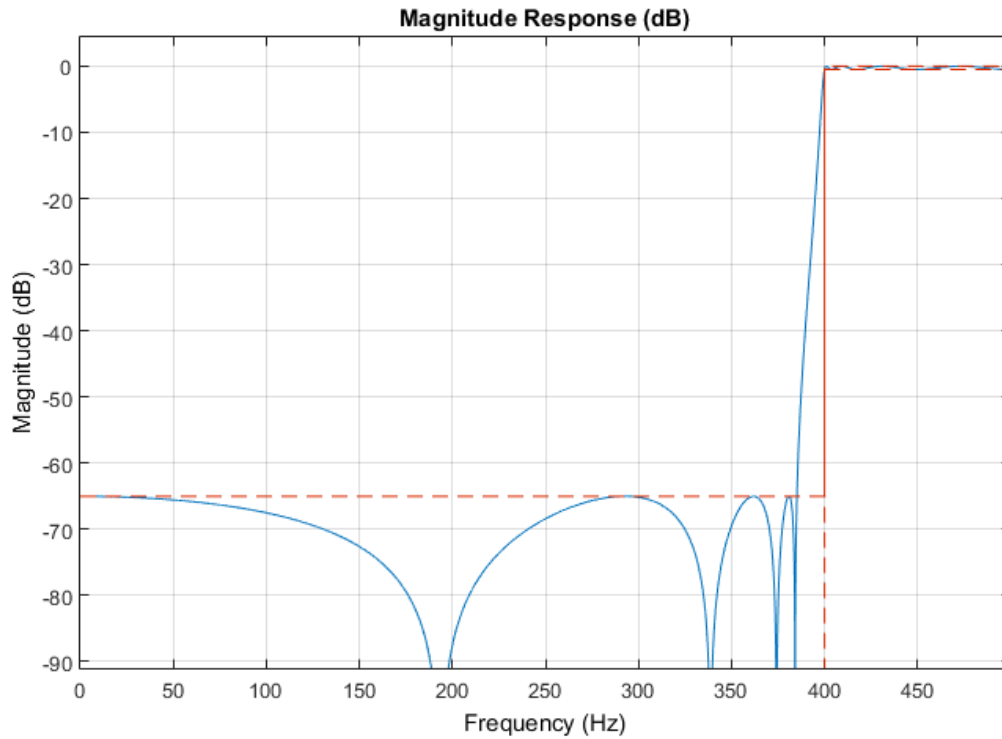
Bandpass FIR Filters

Equiripple Design

```
Fstop1 = 150;
Fpass1 = 200;
Fpass2 = 300;
Fstop2 = 350;
Astop1 = 65;
Apass  = 0.5;
Astop2 = 65;
Fs = 1e3;

d = designfilt('bandpassfir', ...
  'StopbandFrequency1',Fstop1,'PassbandFrequency1', Fpass1, ...
  'PassbandFrequency2',Fpass2,'StopbandFrequency2', Fstop2, ...
```

```
    'StopbandAttenuation1',Astop1,'PassbandRipple', Apass, ...
    'StopbandAttenuation2',Astop2, ...
    'DesignMethod','equiripple','SampleRate',Fs);

fvtool(d)
```
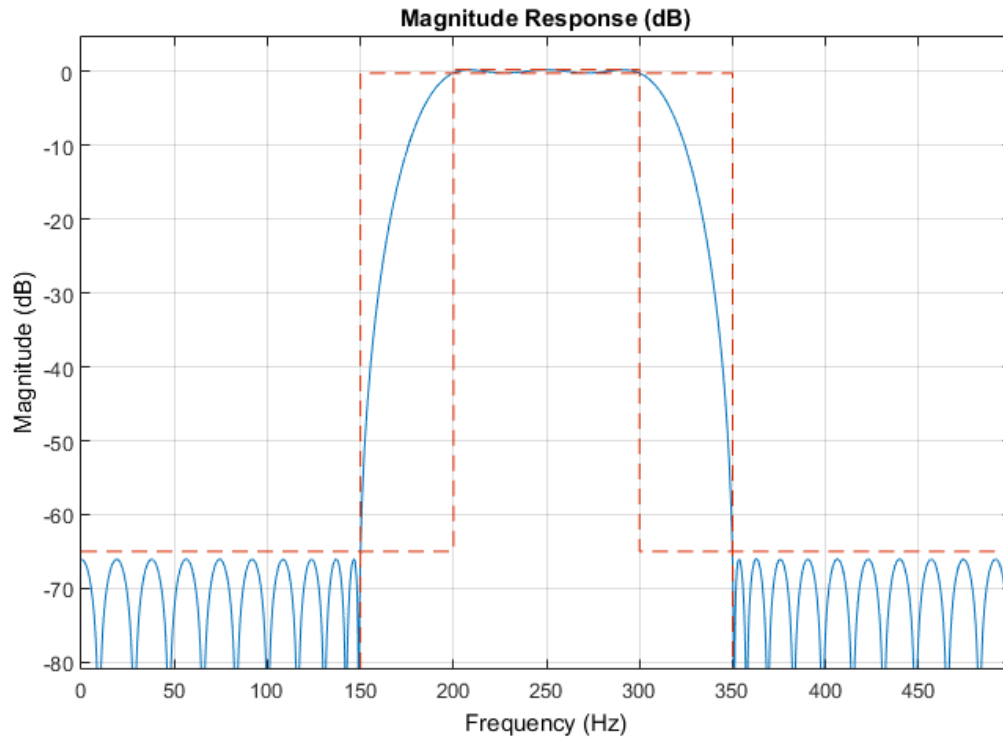

Asymmetric Band Attenuations

```
N = 50;
Fstop1 = 150;
Fpass1 = 200;
Fpass2 = 300;
Fstop2 = 350;
Wstop1 = 3;
Wstop2 = 100;
Fs = 1e3;
```

```matlab
d = designfilt('bandpassfir', ...
  'FilterOrder',N, ...
  'StopbandFrequency1',Fstop1,'PassbandFrequency1', Fpass1, ...
  'PassbandFrequency2',Fpass2,'StopbandFrequency2', Fstop2, ...
  'StopbandWeight1',Wstop1,'StopbandWeight2',Wstop2, ...
   'DesignMethod','equiripple','SampleRate',Fs);

fvtool(d)
```
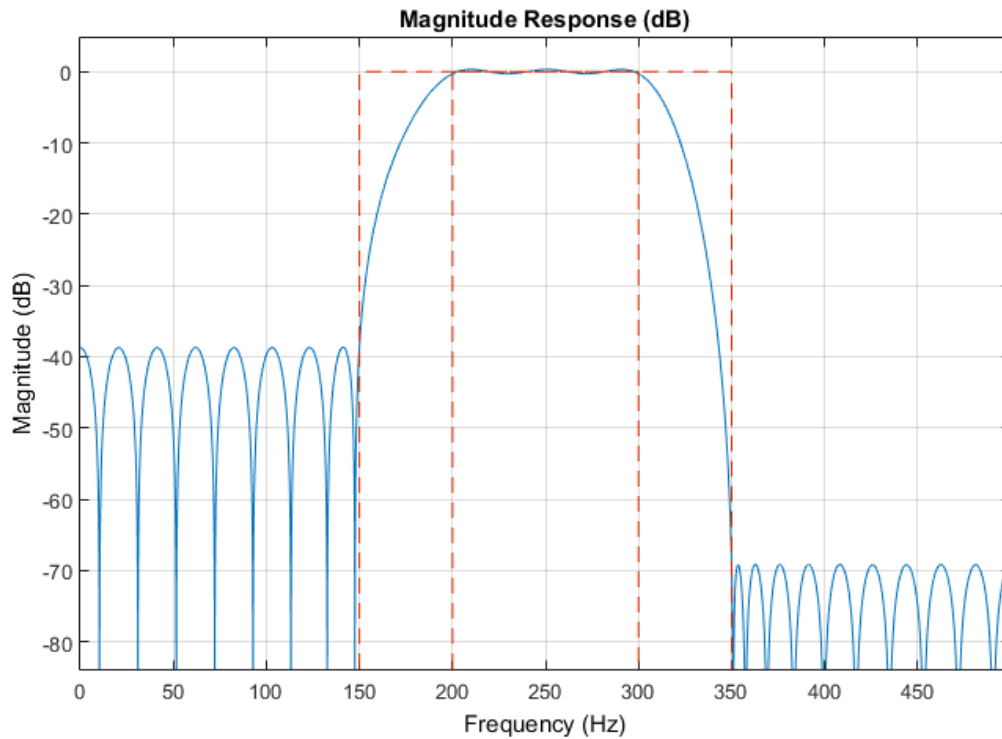

Bandpass IIR Filters

Maximally Flat Design

```matlab
Fstop1 = 150;
Fpass1 = 200;
Fpass2 = 300;
```

```
Fstop2 = 350;
Astop1 = 65;
Apass  = 0.5;
Astop2 = 65;
Fs = 1e3;

d = designfilt('bandpassiir', ...
  'StopbandFrequency1',Fstop1,'PassbandFrequency1', Fpass1, ...
  'PassbandFrequency2',Fpass2,'StopbandFrequency2', Fstop2, ...
  'StopbandAttenuation1',Astop1,'PassbandRipple', Apass, ...
  'StopbandAttenuation2',Astop2, ...
  'DesignMethod','butter','SampleRate', Fs);

fvtool(d)
```
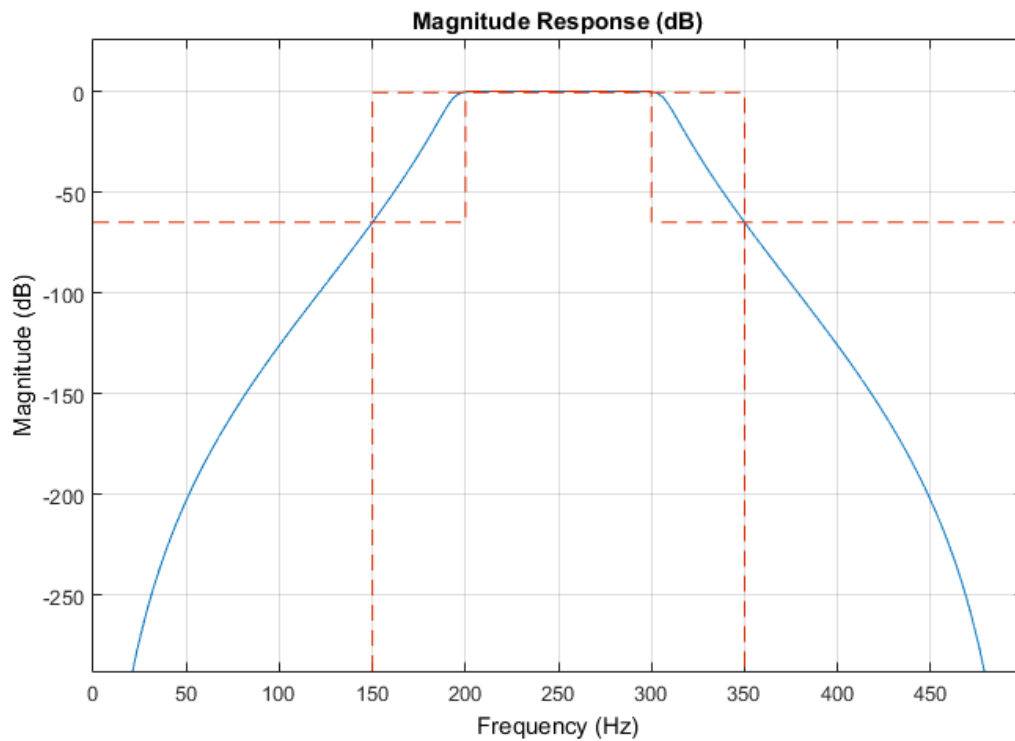

Ripple in Passband and Stopband

```
N = 8;
Fpass1 = 200;
Fpass2 = 300;
Astop1 = 65;
Apass  = 0.5;
Astop2 = 65;
Fs = 1e3;

d = designfilt('bandpassiir', ...
  'FilterOrder',N, ...
  'PassbandFrequency1', Fpass1,'PassbandFrequency2', Fpass2, ...
  'StopbandAttenuation1', Astop1, 'PassbandRipple', Apass, ...
  'StopbandAttenuation2', Astop2, ...
  'SampleRate', Fs);

fvtool(d)
```
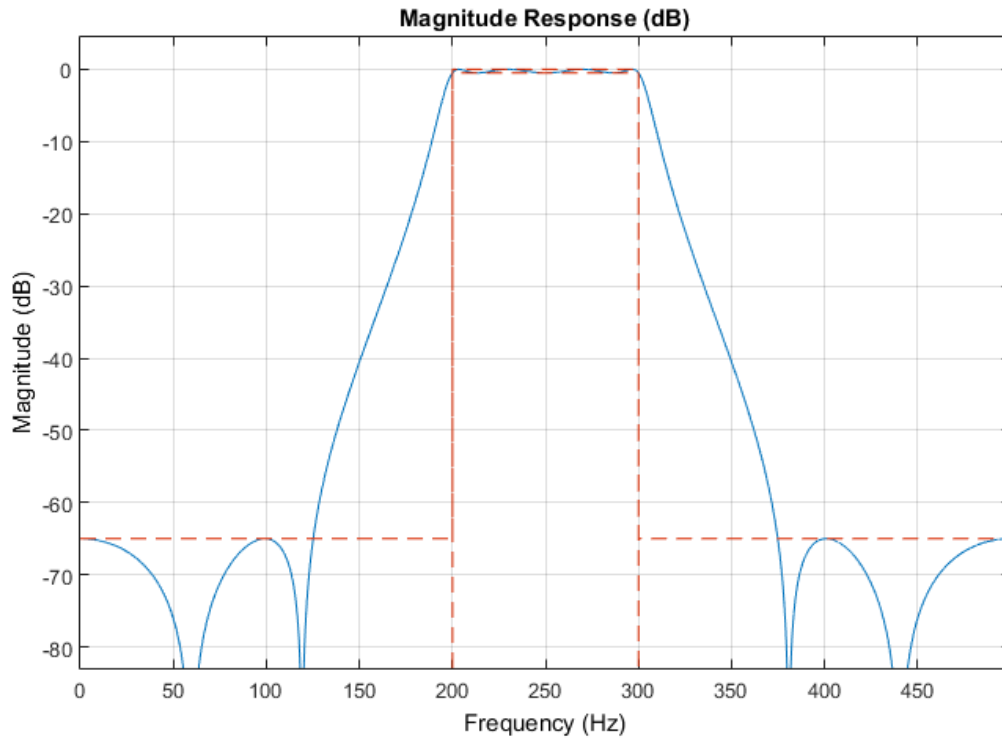
Bandstop FIR Filters

Equiripple Design

```
Fpass1 = 100;
Fstop1 = 150;
Fstop2 = 350;
Fpass2 = 400;
Apass1 = 0.5;
Astop  = 65;
Apass2 = 0.5;
Fs = 1e3;

d = designfilt('bandstopfir', ...
  'PassbandFrequency1',Fpass1,'StopbandFrequency1',Fstop1, ...
  'StopbandFrequency2',Fstop2,'PassbandFrequency2',Fpass2, ...
  'PassbandRipple1',Apass1,'StopbandAttenuation',Astop, ...
  'PassbandRipple2', Apass2, ...
  'DesignMethod','equiripple','SampleRate', Fs);

fvtool(d)
```
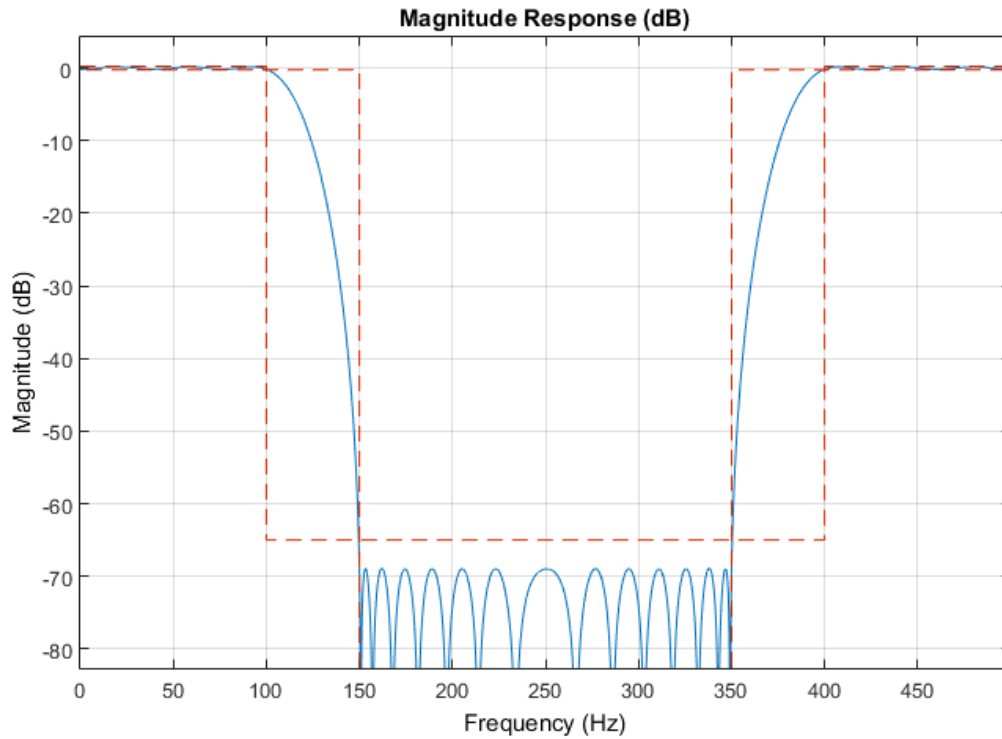
Asymmetric Passband Ripples

```
N = 30;
Fpass1 = 100;
Fstop1 = 150;
Fstop2 = 350;
Fpass2 = 400;
Wpass1 = 1;
Wpass2  = 10;
Fs = 1e3;

d = designfilt('bandstopfir', ...
  'FilterOrder',N, ...
  'PassbandFrequency1',Fpass1,'StopbandFrequency1',Fstop1, ...
  'StopbandFrequency2',Fstop2,'PassbandFrequency2',Fpass2, ...
  'PassbandWeight1',Wpass1,'PassbandWeight2',Wpass2, ...
```

```
'DesignMethod','equiripple','SampleRate', Fs);
```

```
fvtool(d)
```


Magnitude Response (dB)
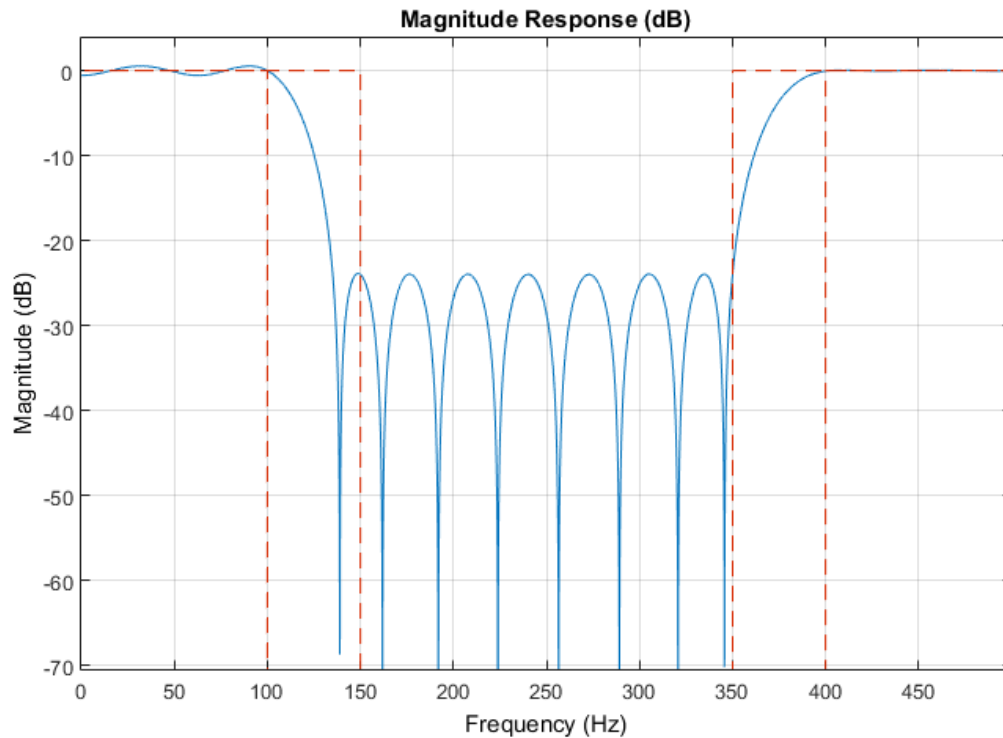
Bandstop IIR Filters

Maximally Flat Design

```
Fpass1 = 100;
Fstop1 = 150;
Fstop2 = 350;
Fpass2 = 400;
Apass1 = 0.5;
Astop  = 65;
Apass2 = 0.5;
Fs = 1e3;
```

```
d = designfilt('bandstopiir', ...
  'PassbandFrequency1',Fpass1,'StopbandFrequency1',Fstop1, ...
  'StopbandFrequency2',Fstop2,'PassbandFrequency2',Fpass2, ...
  'PassbandRipple1',Apass1,'StopbandAttenuation',Astop, ...
  'PassbandRipple2', Apass2, ...
  'DesignMethod','butter','SampleRate', Fs);

fvtool(d)
```
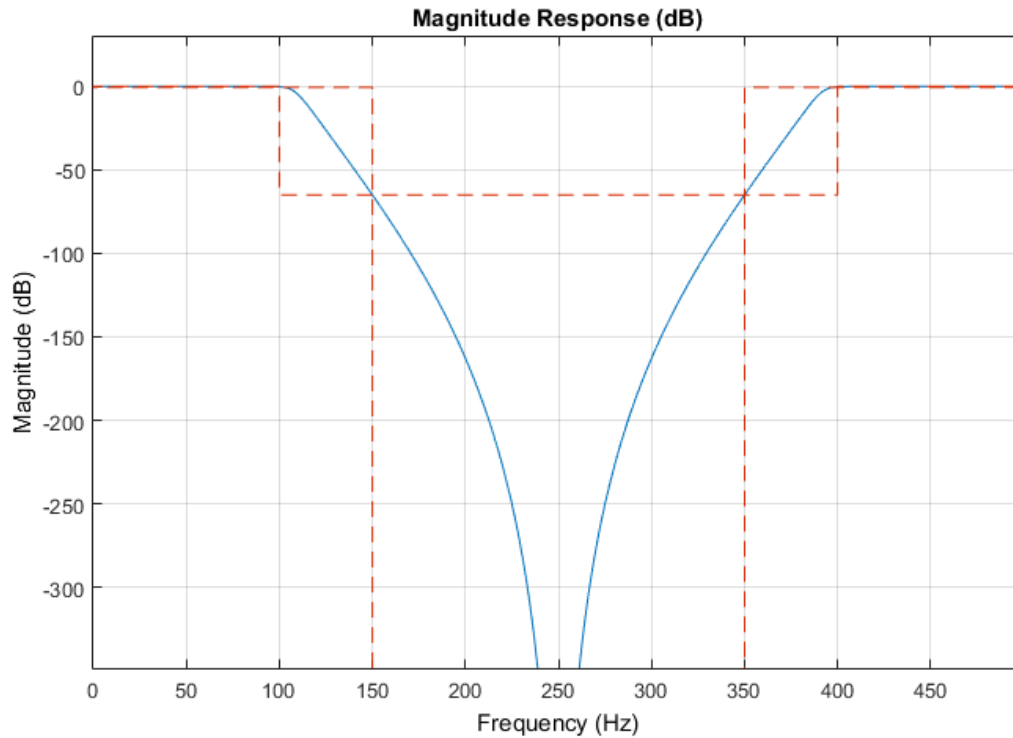

Ripple in Passband and Stopband

```
N = 8;
Fpass1 = 125;
Fpass2 = 375;
Apass = 0.5;
Astop  = 65;
Fs = 1e3;
```

```matlab
d = designfilt('bandstopiir', ...
  'FilterOrder',N, ...
  'PassbandFrequency1',Fpass1,'PassbandFrequency2',Fpass2, ...
  'PassbandRipple',Apass,'StopbandAttenuation', Astop, ...
  'SampleRate',Fs);

fvtool(d)
```
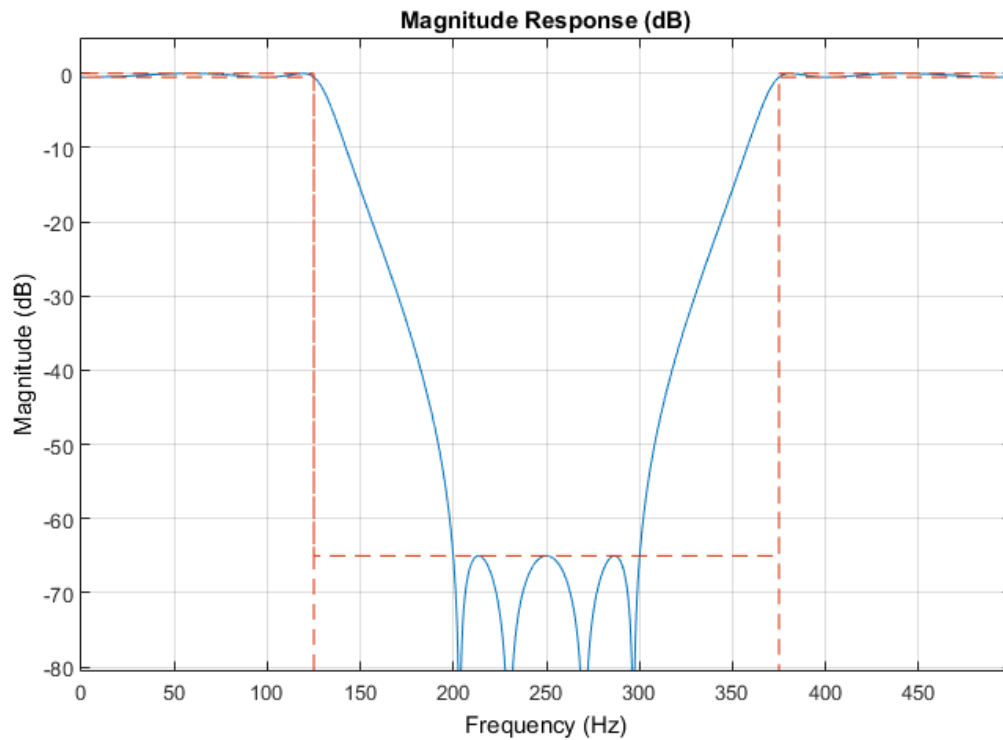

Arbitrary Magnitude FIR Filters

Single-Band Arbitrary Magnitude Design

```matlab
N = 300;

% Frequencies are in normalized units
F1 = 0:0.01:0.18;
```

```
F2 = [.2 .38 .4 .55 .562 .585 .6 .78];
F3 = 0.79:0.01:1;
FreqVect = [F1 F2 F3]; % vector of frequencies

% Define desired response using linear units
A1 = .5+sin(2*pi*7.5*F1)/4;    % Sinusoidal section
A2 = [.5 2.3 1 1 -.2 -.2 1 1]; % Piecewise linear section
A3 = .2+18*(1-F3).^2;          % Quadratic section

AmpVect = [A1 A2 A3];

d = designfilt('arbmagfir',...
   'FilterOrder',N,'Amplitudes',AmpVect,'Frequencies',FreqVect,...
   'DesignMethod','freqsamp');

fvtool(d,'MagnitudeDisplay','Zero-phase')
```
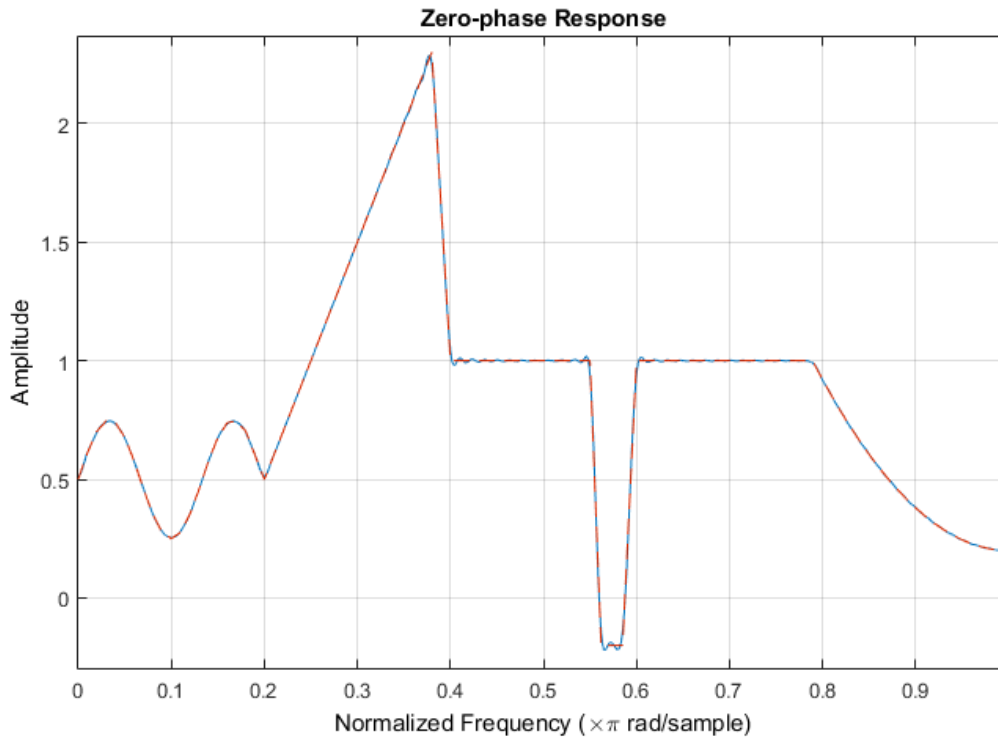
Multiband Lowpass Design with Stepped Attenuation Levels on Stopband

```
N = 150;
B = 2; % Number of bands
% Frequencies are in normalized units
F1 = [0 .25]; % Passband
F2 = [.3 .4 .401 .5 .501 .6 .601 .7 .701 .8 .801 .9 .901 1]; % Stopband
A1 = ones(size(F1));  % Desired amplitudes for band 1 in linear units
A2 = zeros(size(F2)); % Desired amplitudes for band 2 in linear units
% Vector of weights
W = 10.^([0 0 5 5 10 10 15 15 20 20 25 25 30 30 35 35]/20);
W1 = W(1:2);   % Weights for band 1
W2 = W(3:end); % Weights for band 2

d = designfilt('arbmagfir', ...
  'FilterOrder',N,'NumBands',B, ...
  'BandFrequencies1',F1,'BandAmplitudes1',A1, ...
  'BandFrequencies2',F2,'BandAmplitudes2',A2, ...
  'BandWeights1',W1,'BandWeights2', W2);

fvtool(d)
```

Magnitude Response (dB)
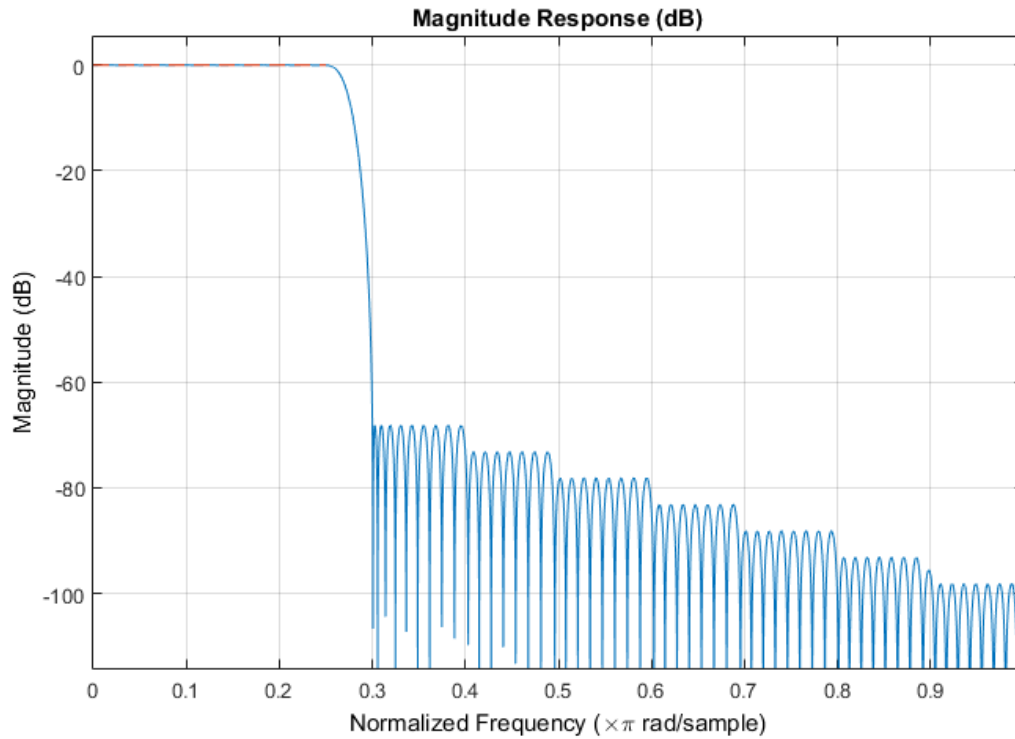
Differentiator FIR Filters

Full Band Design

```
N = 41;
Fs = 1e3;

d = designfilt('differentiatorfir', ...
'FilterOrder',N, 'DesignMethod','equiripple','SampleRate',Fs);

fvtool(d,'MagnitudeDisplay','zero-phase','OverlayedAnalysis','phase')
```

Zero-phase Response and Phase Response
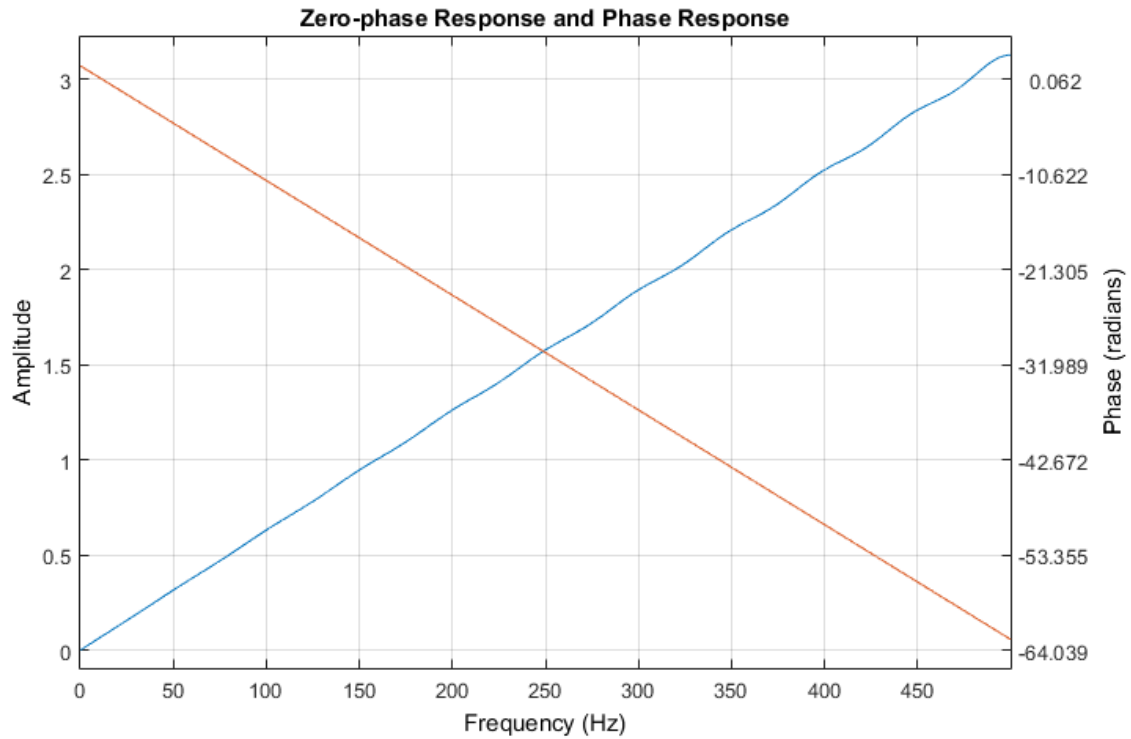
Partial Band Design

```
N = 40;
Fpass = 100;
Fstop = 150;
Fs = 1e3;

d = designfilt('differentiatorfir', ...
'FilterOrder',N, ...
'PassbandFrequency',Fpass,'StopbandFrequency',Fstop, ...
'DesignMethod','equiripple','SampleRate',Fs);

fvtool(d,'MagnitudeDisplay','zero-phase','OverlayedAnalysis','phase')
```

Zero-phase Response and Phase Response
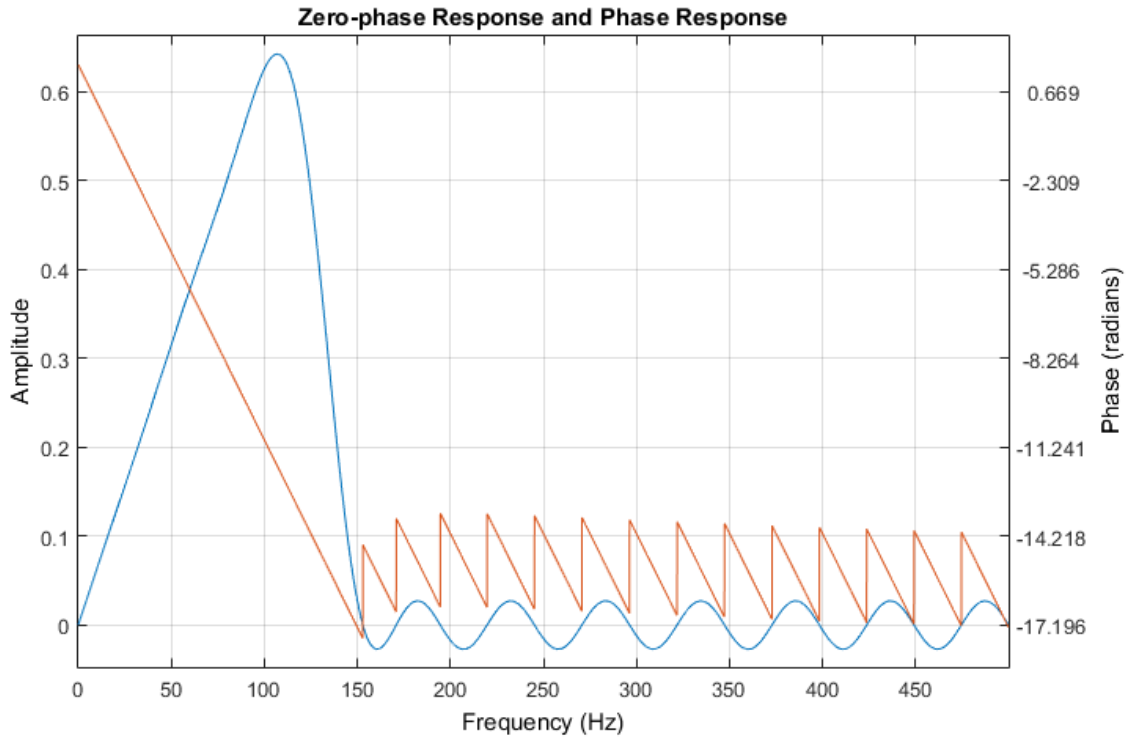
Hilbert FIR Filters

Equiripple Design

```
N = 40;
Tw = 50;
Fs = 1e3;

d = designfilt('hilbertfir', ...
'FilterOrder',N,'TransitionWidth',Tw, ...
'DesignMethod','equiripple','SampleRate',Fs);

fvtool(d,'MagnitudeDisplay','Zero-phase','OverlayedAnalysis','phase')
```
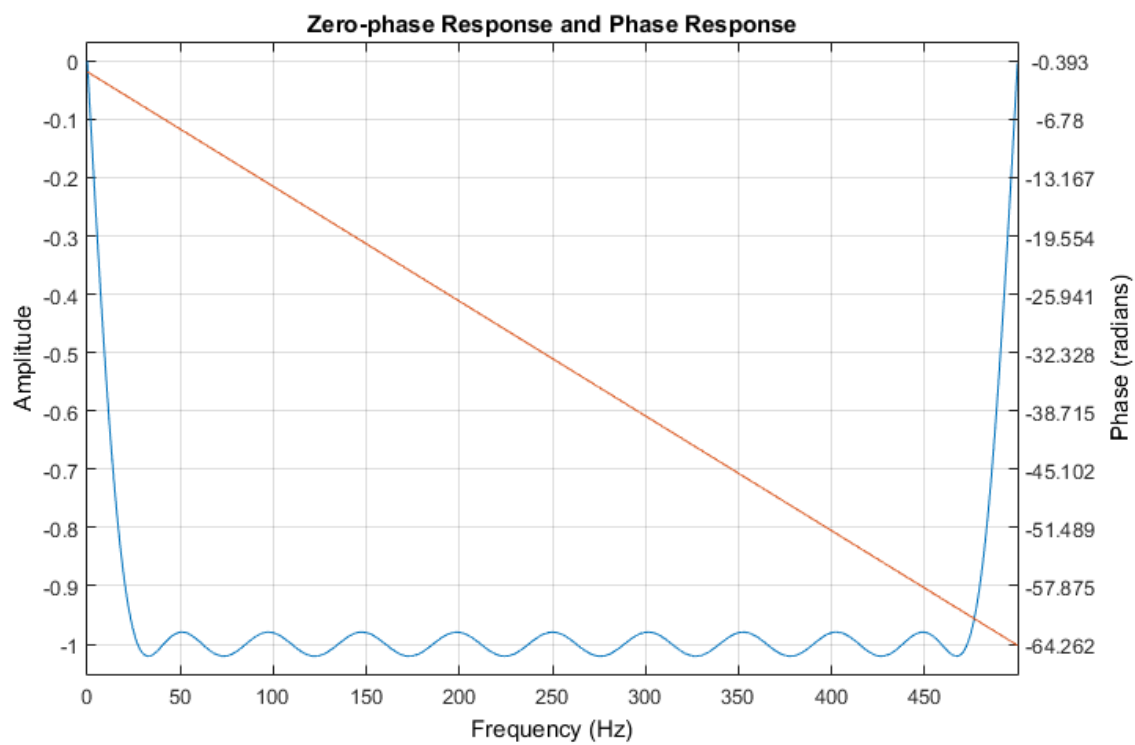
Selected Bibliography

[1] Karam, Lina J., and James H. McClellan. "Complex Chebyshev Approximation for FIR Filter Design." *IEEE® Transactions on Circuits and Systems II: Analog and Digital Signal Processing*. Vol.42, March 1995, pp.207–216.

[2] Selesnick, Ivan W., and C. Sidney Burrus. "Generalized Digital Butterworth Filter Design." *IEEE Transactions on Signal Processing*. Vol.46, June 1998, pp.1688–1694.

[3] Selesnick, Ivan W., Markus Lang, and C. Sidney Burrus. "Constrained Least Square Design of FIR Filters without Specified Transition Bands." *IEEE Transactions on Signal Processing*. Vol.44, August 1996, pp.1879–1892.

Designing a Filter in Fdesign — Process Overview

Process Flow Diagram and Filter Design Methodology

In this section...
"Exploring the Process Flow Diagram" on page 3-2
"Selecting a Response" on page 3-4
"Selecting a Specification" on page 3-4
"Selecting an Algorithm" on page 3-6
"Customizing the Algorithm" on page 3-7
"Designing the Filter" on page 3-8
"Design Analysis" on page 3-9
"Realize or Apply the Filter to Input Data" on page 3-9

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.
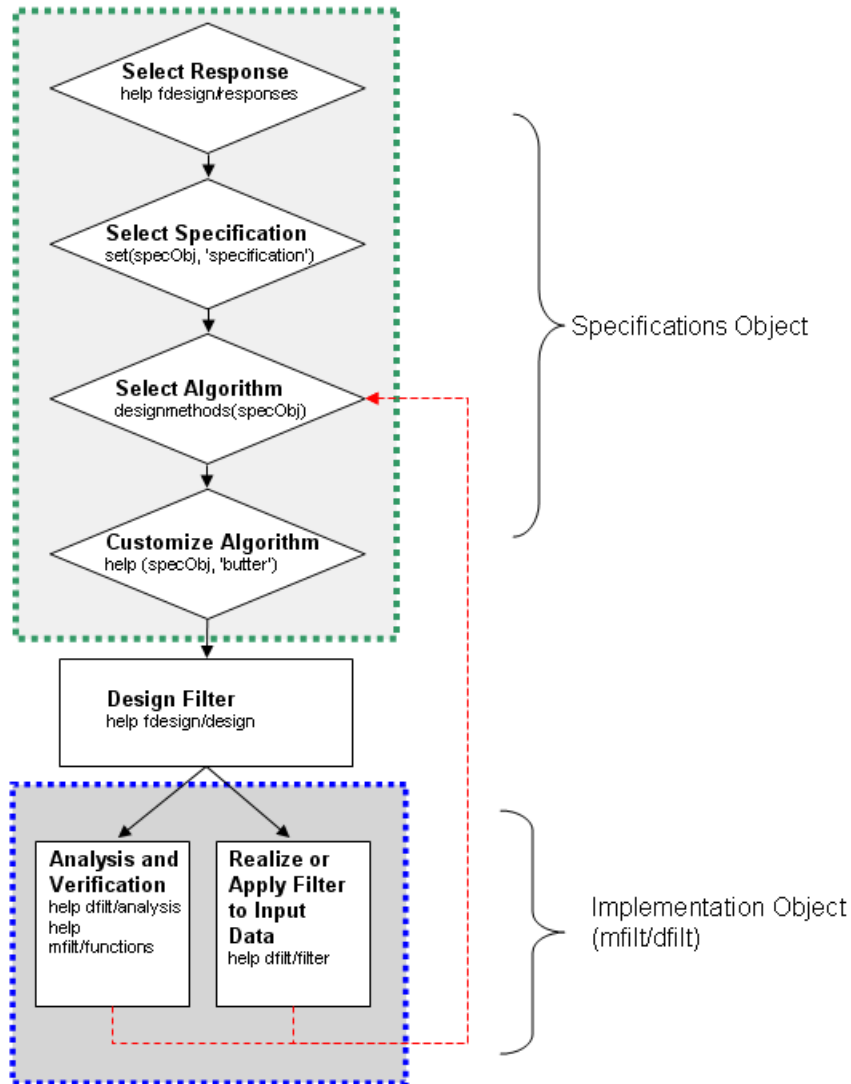
The first four steps of the filter design process relate to the filter Specifications Object, while the last two steps involve the filter Implementation Object. Both of these objects are discussed in more detail in the following sections. Step 5 - the design of the filter, is the transition step from 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 4-2, and provides default
values for all design parameters included in the specification.

Selecting an Algorithm

The availability of algorithms depends the chosen filter response, the design parameters,
and the availability of the DSP System Toolbox. In other words, for the same lowpass
filter, changing the specification 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
```

Designing a Filter in the Filterbuilder GUI

Filterbuilder Design Process

In this section...

Introduction to Filterbuilder

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

Note: `filterbuilder` requires the Signal Processing Toolbox. The 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.

Design a Filter Using Filterbuilder

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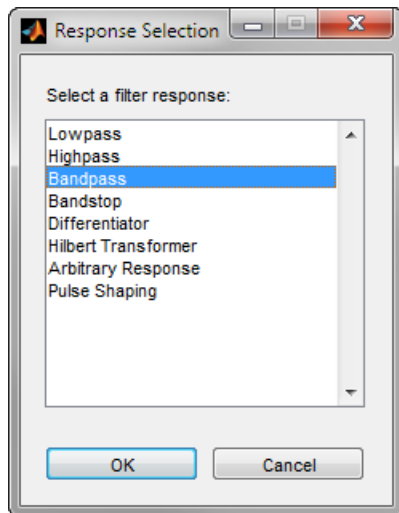
Note This step cannot be skipped because it is not automatically completed for you by the software. You must select a response to initiate the filter design process.

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.
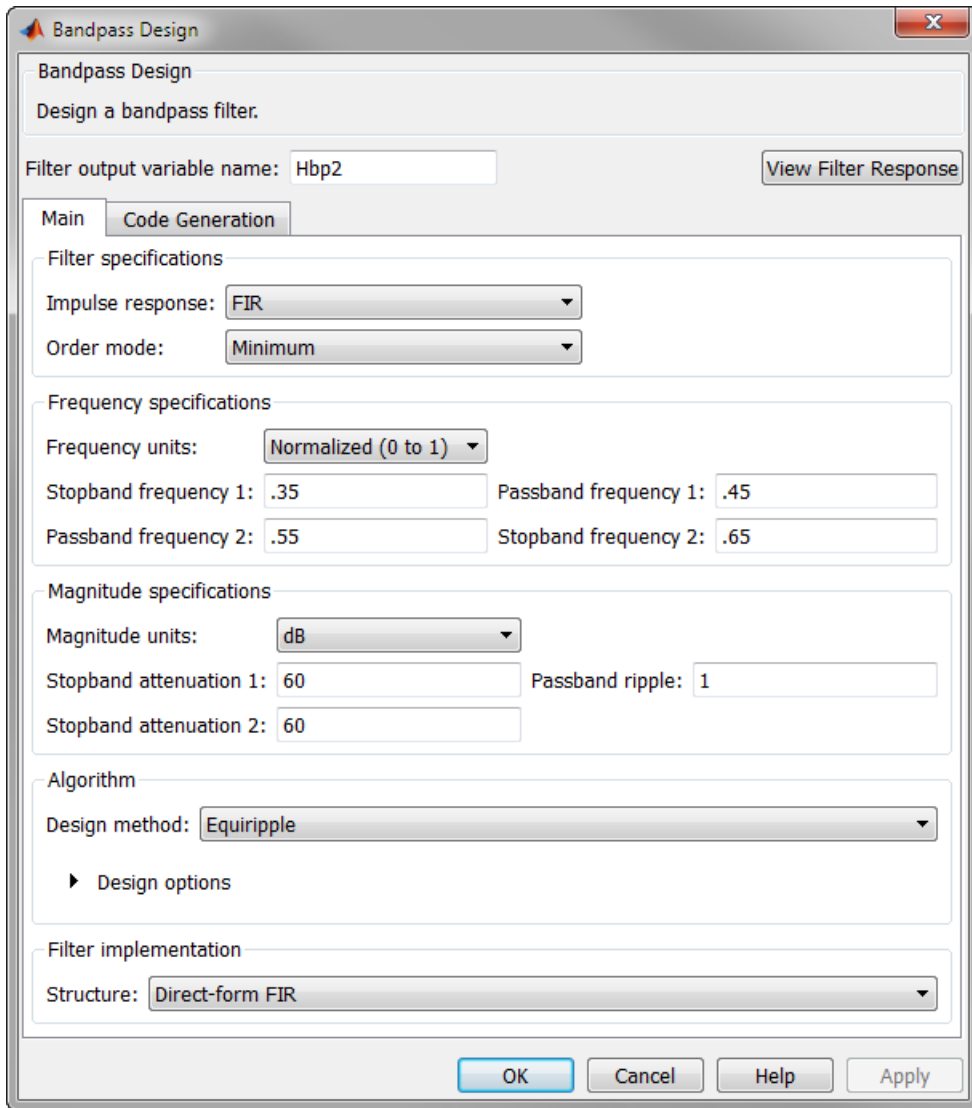
The **Bandpass Design** dialog box contains all the parameters you need to determine the specifications of a bandpass filter. The parameters listed in the **Main** pane depend upon the type of filter you are designing. However, no matter what type of filter you have

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 are `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`.
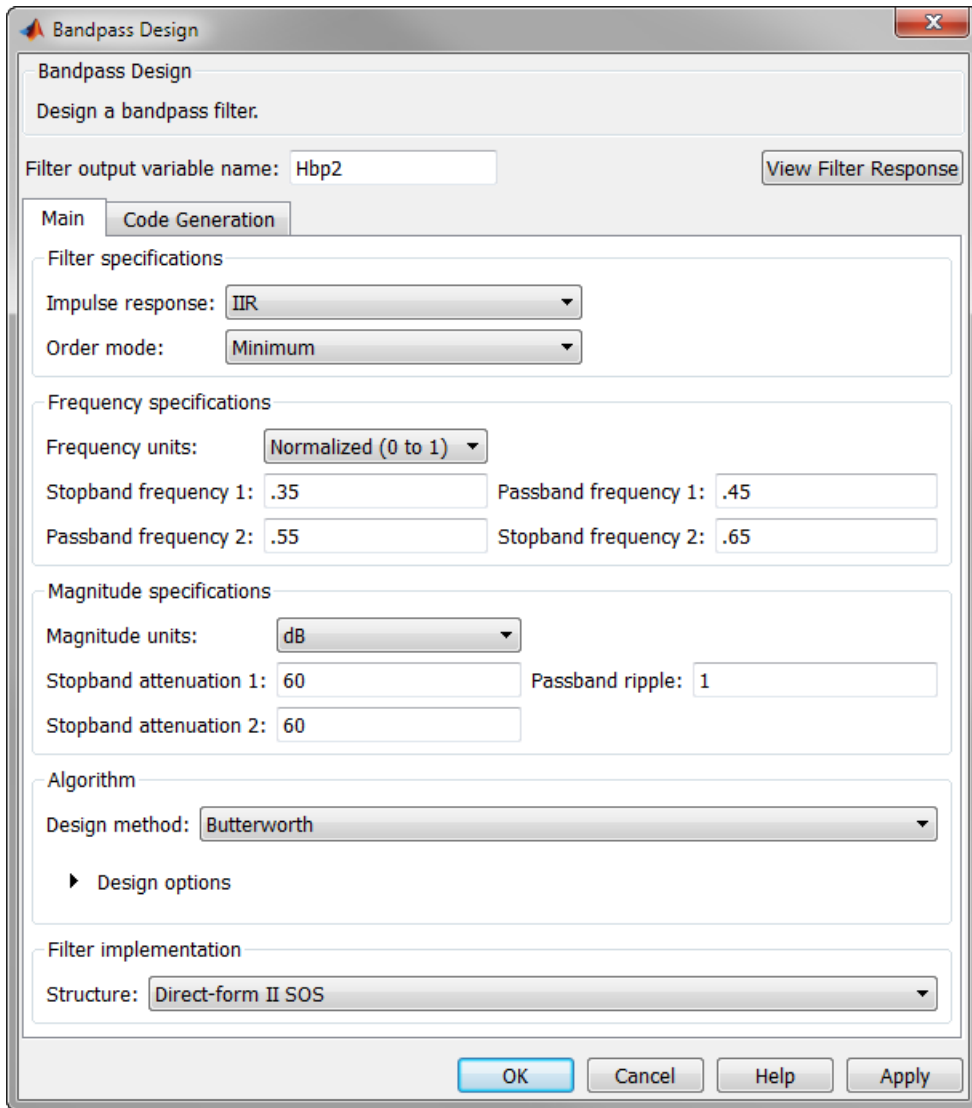
Customize the Algorithm

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.
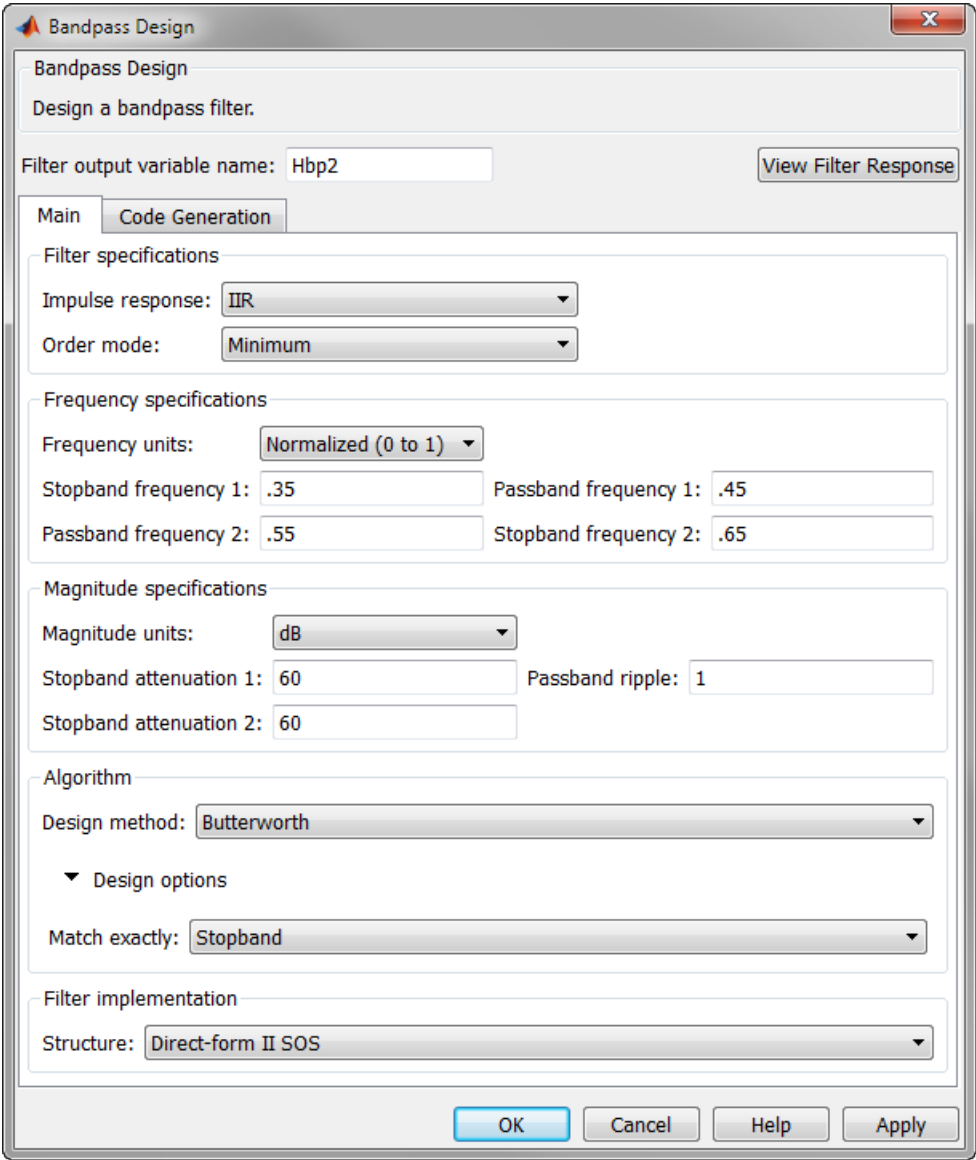
Analyze the Design

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.
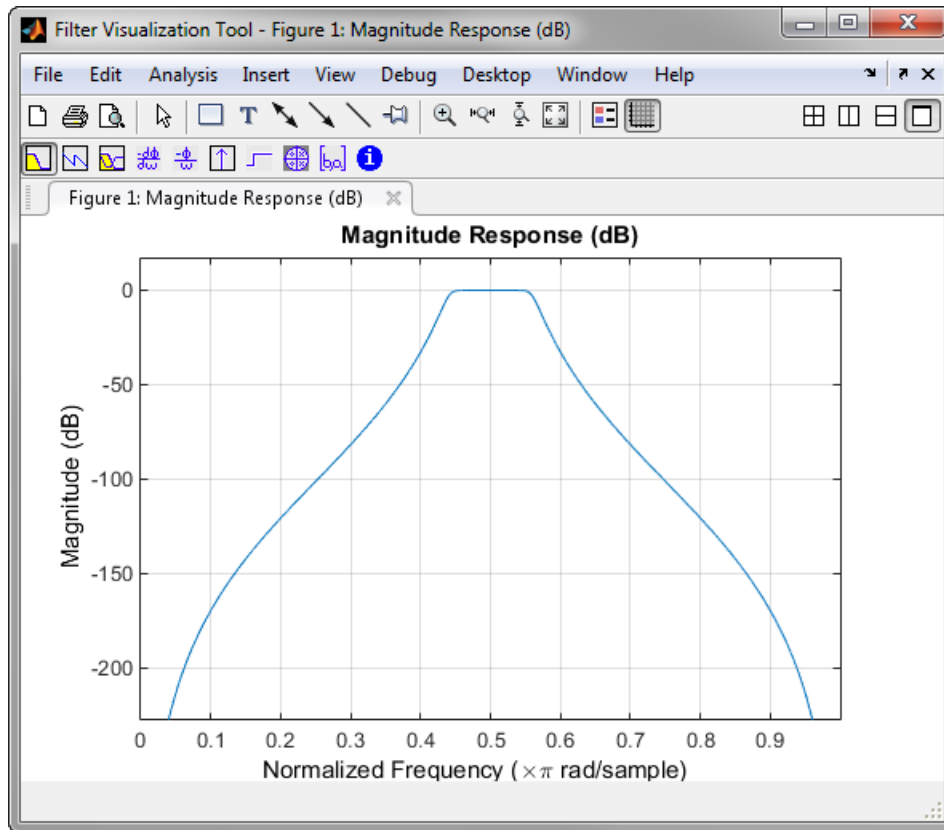
Realize or Apply the Filter to Input Data

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 **Bandpass Design** dialog box, click OK and the Signal Processing Toolbox software creates the filter coefficients and exports it to the MATLAB workspace.

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

```
y = filter(Hbs,x)
```
To understand how the filtering command works, type:

```
help dfilt/filter
```

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

```
help realizemdl
```

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.
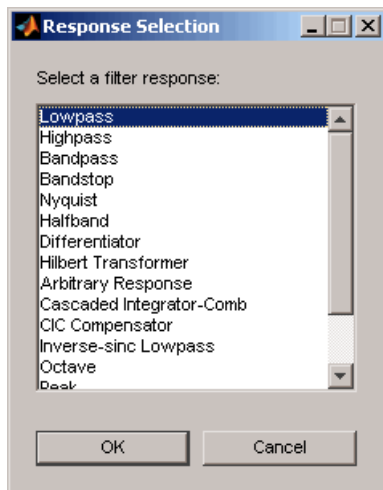
2   Hit the **OK** button. The **Lowpass Design** dialog box opens. Here you can specify the writable parameters of the Lowpass filter object. The components of the **Main** frame of this dialog box are described in the section titled *Lowpass Filter Design Dialog Box — Main Pane*. In the dialog box, 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.
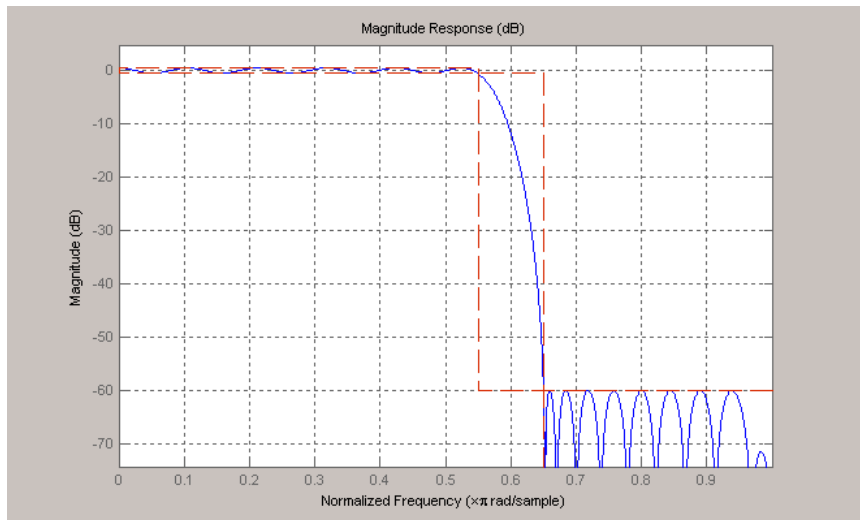
You can change the design and click **Apply**, followed by **View Filter Response**, as many times as needed until your design specifications are met.

Compensate for Delay and Distortion Introduced by Filters

Filtering a signal introduces a delay. This means that the output signal is shifted in time with respect to the input.

When the shift is constant, you can correct for the delay by shifting the signal in time.

Sometimes the filter delays some frequency components more than others. This phenomenon is called phase distortion. To compensate for this effect, you can perform zero-phase filtering using the `filtfilt` function.

Take an electrocardiogram reading sampled at 500 Hz for 1 s. Add random noise. Reset the random number generator for reproducible results

```
Fs = 500;
N = 500;

rng default

xn = ecg(N)+0.1*randn([1 N]);
tn = (0:N-1)/Fs;
```

Remove some of the noise using a filter that stops frequencies above 75 Hz. Use `designfilt` to design an FIR filter of order 70.

```
Nfir = 70;
Fst = 75;

firf = designfilt('lowpassfir','FilterOrder',Nfir, ...
    'CutoffFrequency',Fst,'SampleRate',Fs);
```

Filter the signal and plot it. The result is smoother than the original, but lags behind it.

```
xf = filter(firf,xn);

plot(tn,xn,tn,xf)
title 'Electrocardiogram'
xlabel 'Time (s)'
legend('Original','FIR Filtered')
```
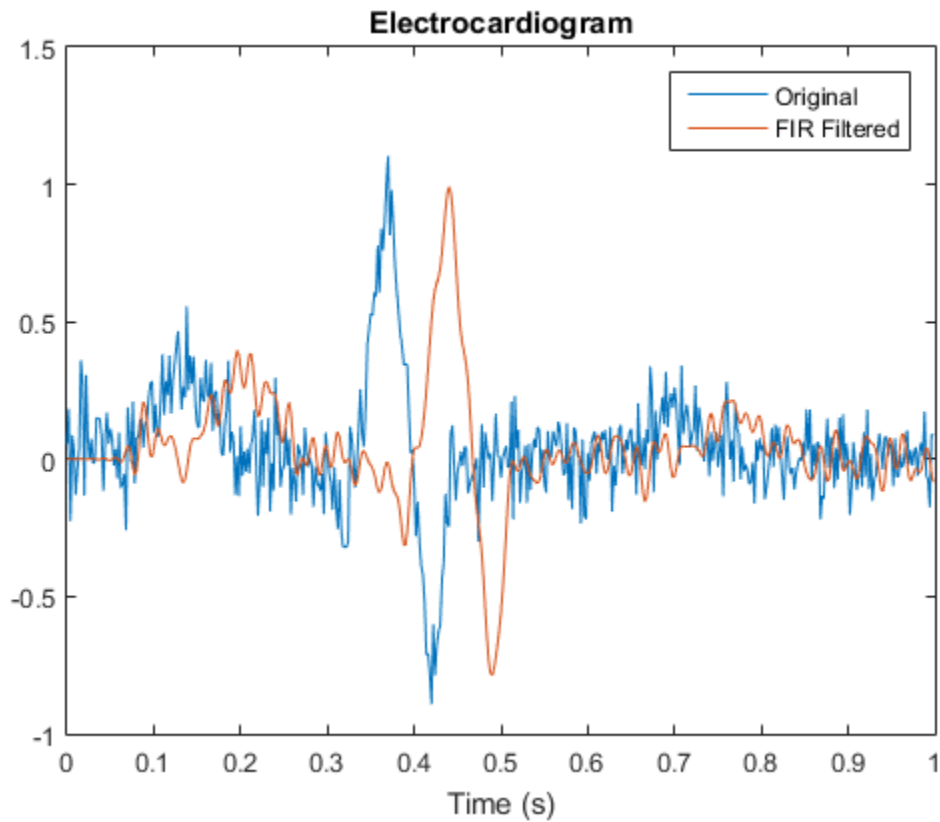
Use `grpdelay` to check that the delay caused by the filter equals half the filter order.

```
grpdelay(firf,N,Fs)
delay = mean(grpdelay(firf))

delay =

    35
```
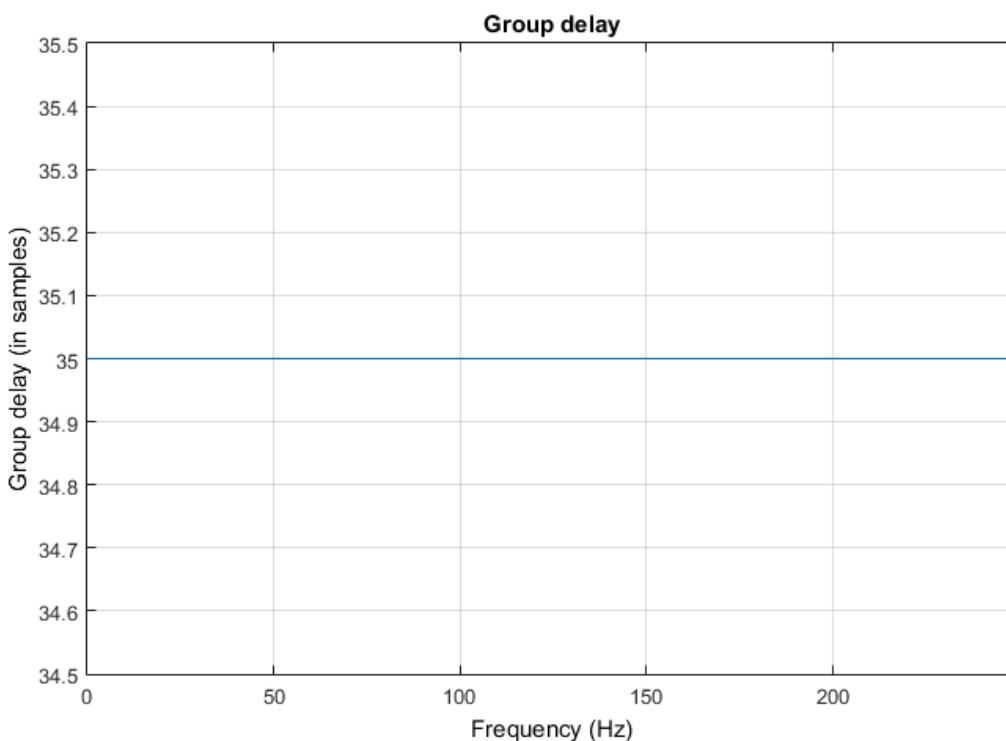
Line up the data. Shift the filtered signal by removing its first `delay` samples. Remove the last `delay` samples of the original and of the time vector.

```
tt = tn(1:end-delay);
sn = xn(1:end-delay);

sf = xf;
sf(1:delay) = [];
```

Plot the signals and verify that they are aligned.

```
plot(tt,sn,tt,sf)
title 'Electrocardiogram'
xlabel('Time (s)'), legend('Original Signal','Filtered Shifted Signal')
```
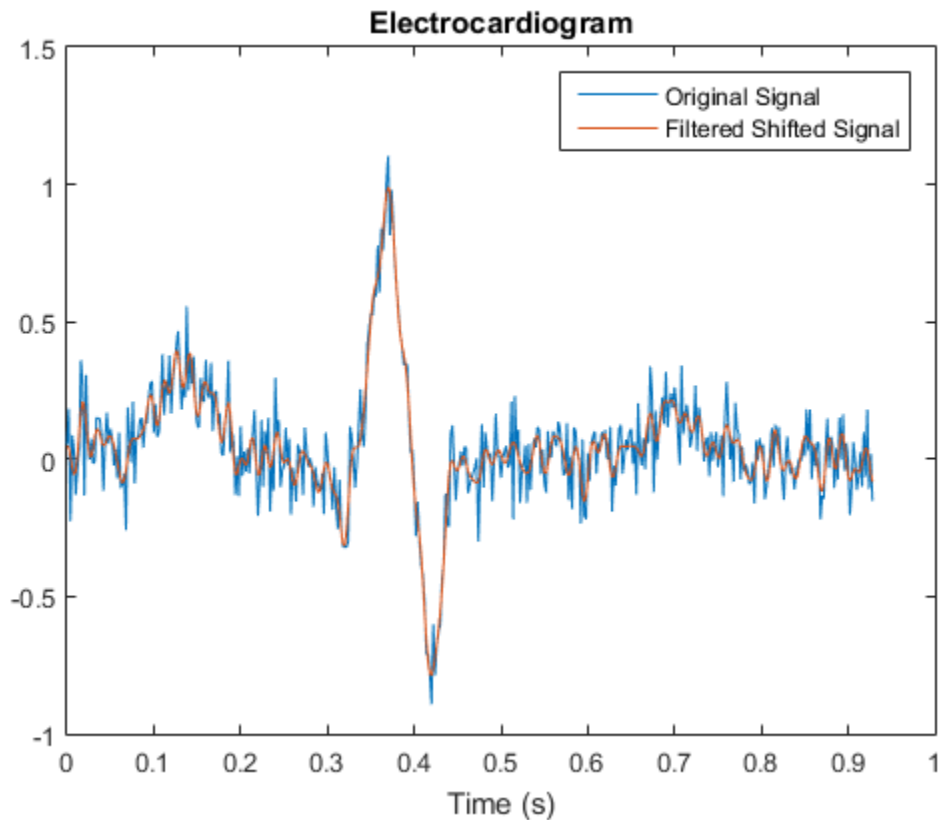
Repeat the computation using a 7th-order IIR filter.

```
Niir = 7;

iir = designfilt('lowpassiir','FilterOrder',Niir, ...
    'HalfPowerFrequency',Fst,'SampleRate',Fs);
```

Filter the signal. The filtered signal is cleaner than the original, but lags in time with respect to it. It is also distorted due to the nonlinear phase of the filter.

```
xfilter = filter(iir,xn);

plot(tn,xn,tn,xfilter)
```

```
title 'Electrocardiogram'
xlabel 'Time (s)'
legend('Original','Filtered')
axis([0.25 0.55 -1 1.5])
```
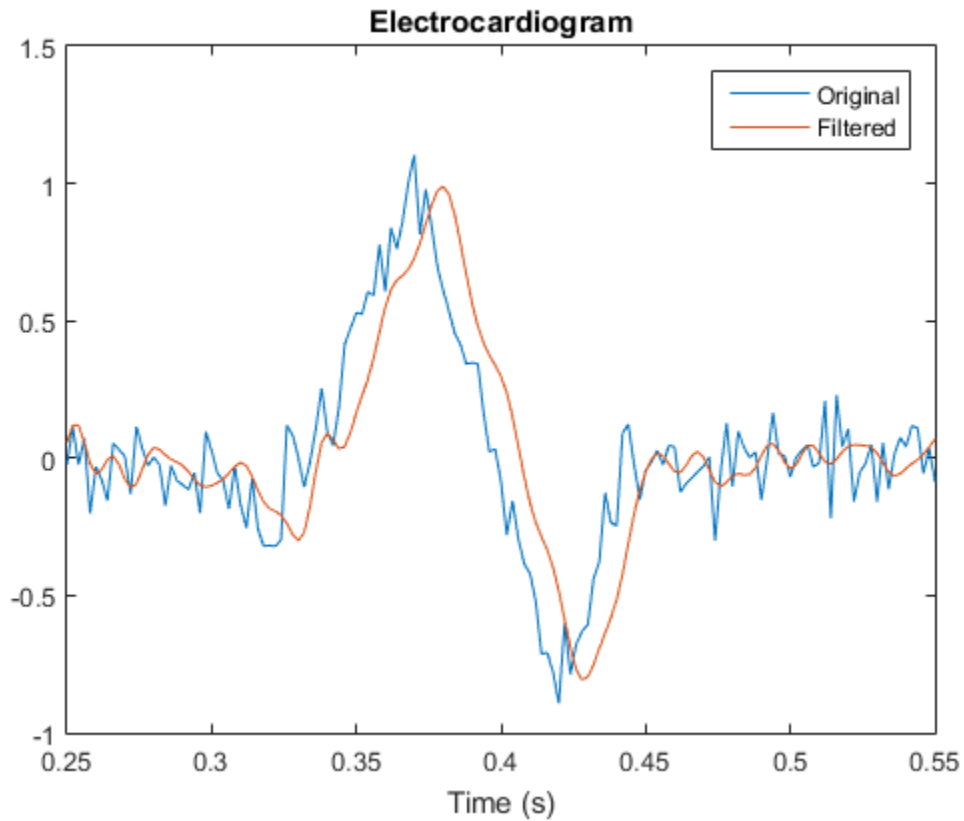

A look at the group delay introduced by the filter shows that the delay is frequency-dependent.

```
grpdelay(iir,N,Fs)
```
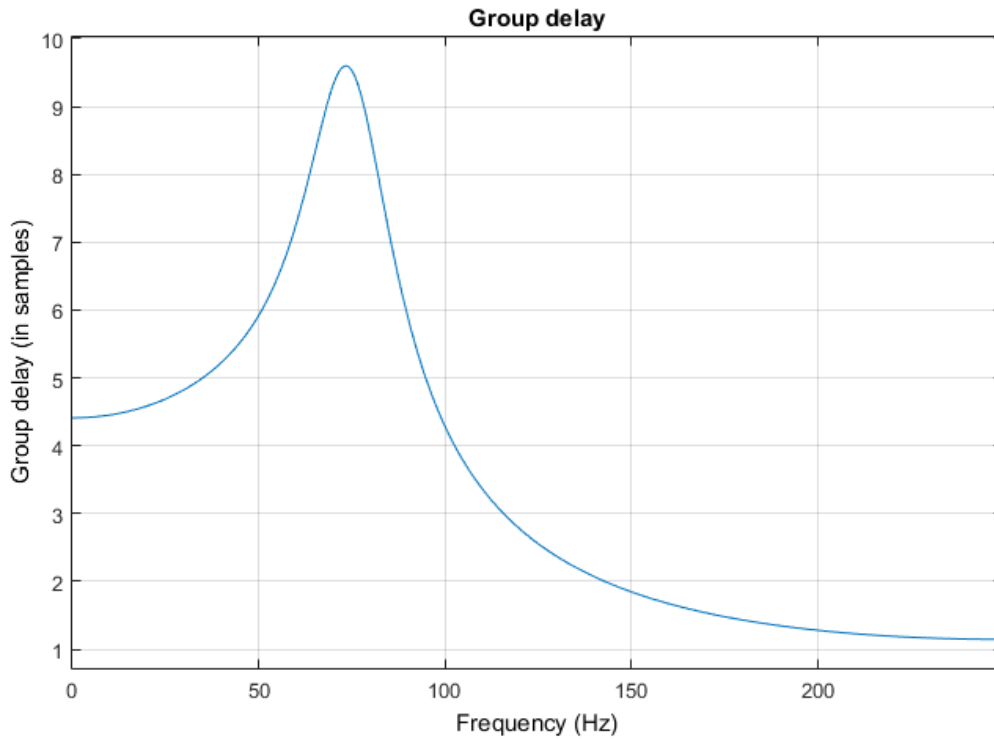
Group delay

Filter the signal using `filtfilt`. The delay and distortion have been effectively removed. Use `filtfilt` when it is critical to keep the phase information of a signal intact.

```
xfiltfilt = filtfilt(iir,xn);

plot(tn,xn)
hold on
plot(tn,xfilter)
plot(tn,xfiltfilt)

title 'Electrocardiogram'
xlabel 'Time (s)'
legend('Original','''filter''','''filtfilt''')
axis([0.25 0.55 -1 1.5])
```
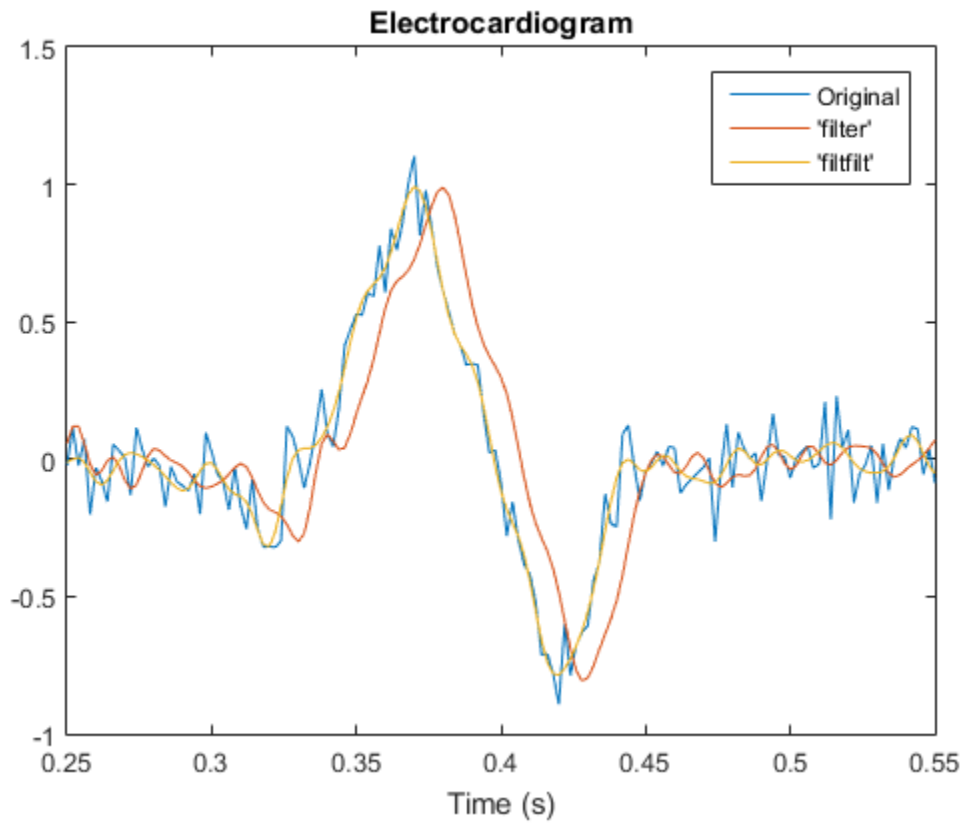
Comparison of Analog IIR Lowpass Filters

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

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

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

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

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

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

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

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

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

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

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

```
ylabel('Attenuation (dB)')
legend('butter','cheby1','cheby2','ellip')
```
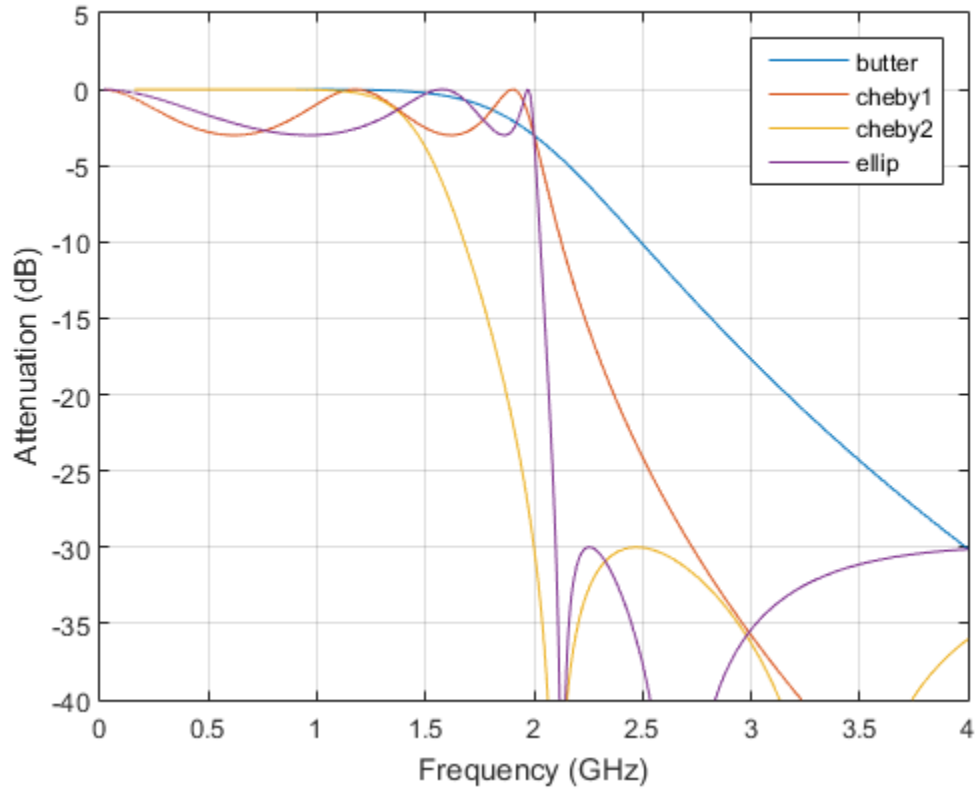

The Butterworth and Chebyshev Type II filters have flat passbands and wide transition bands. The Chebyshev Type I and elliptic filters roll off faster but have passband ripple. The frequency input to the Chebyshev Type II design function sets the beginning of the stopband rather than the end of the passband.

Frequency Response of an Analog Bessel Filter

Design a 5th-order analog lowpass Bessel filter with approximately constant group delay up to 10^4 rad/s. Plot the magnitude and phase responses of the filter using `freqs`.

```
[b,a] = besself(5,10000);
freqs(b,a)
```
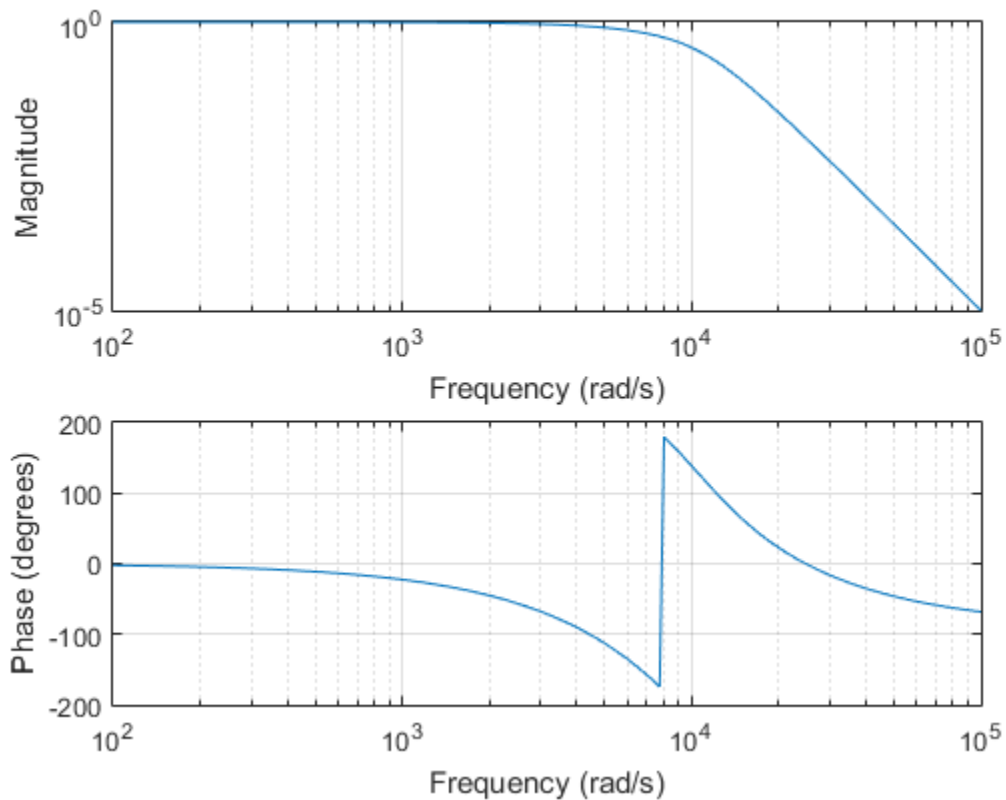
FDATool: A Filter Design and Analysis GUI

FDATool

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.

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`

Design Method	Function
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 5-10 for more information. You use this panel to

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

- Import filter. You use this panel to

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

- Pole/Zero Editor. See "Editing the Filter Using the Pole/Zero Editor" on page 5-20. 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.

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.

Using FDATool

To open FDATool, type

```
fdatool
```
at the MATLAB command prompt.

The Filter Design and Analysis Tool opens with the Design Filter panel displayed.
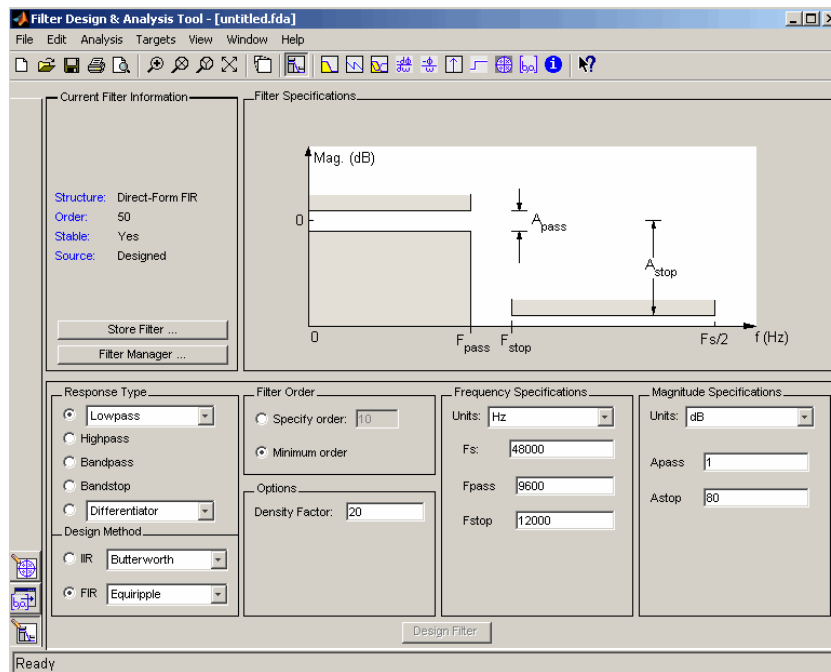
Note that when you open FDATool, **Design Filter** is not enabled. You must make a change to the default filter design in order to enable **Design Filter**. This is true each time you want to change the filter design. Changes to radio button items or drop down menu items such as those under **Response 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 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.
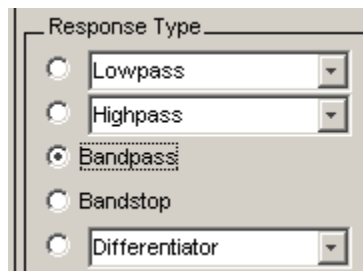
Note: Not all filter design methods are available for all response types. Once you choose your response type, this may restrict the filter design methods available to you. Filter design methods that are not available for a selected response type are removed from the Design Method region of the GUI.

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.
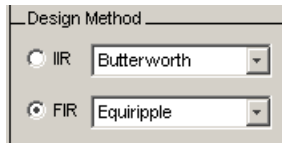
Setting the Filter Design Specifications

- "Viewing Filter Specifications" on page 5-11
- "Filter Order" on page 5-12
- "Options" on page 5-12
- "Bandpass Filter Frequency Specifications" on page 5-13
- "Bandpass Filter Magnitude Specifications" on page 5-14

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**.
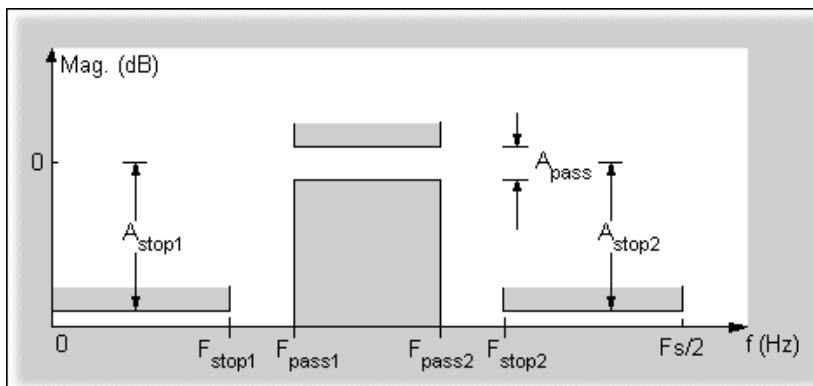
Filter Order

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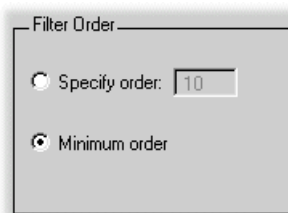
Note that filter order specification options depend on the filter design method you choose. Some filter methods may not have both options available.

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.
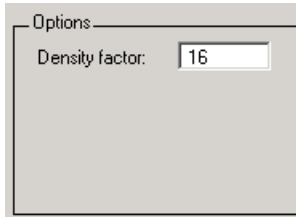
Bandpass Filter Frequency Specifications

For a bandpass filter, you can set

- Units of frequency:

  - Hz
  - kHz
  - MHz
  - Normalized (0 to 1)
- Sampling frequency
- Passband frequencies
- Stopband frequencies
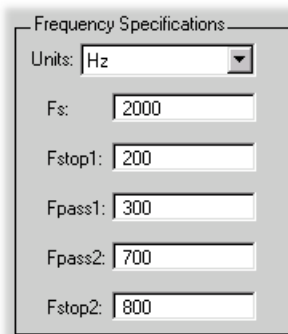
You specify the passband with two frequencies. The first frequency determines the lower edge of the passband, and the second frequency determines the upper edge of the passband.

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.
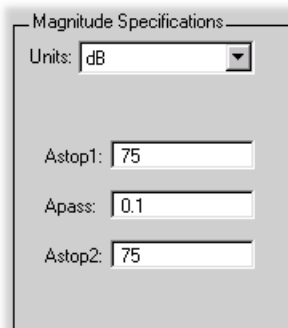
Bandpass Filter Magnitude Specifications

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

- "Displaying Filter Responses" on page 5-15
- "Using Data Tips" on page 5-17
- "Drawing Spectral Masks" on page 5-17
- "Changing the Sampling Frequency" on page 5-19
- "Displaying the Response in FVTool" on page 5-19

Displaying Filter Responses

You can view the following filter response characteristics in the display region or in a separate window.

- Magnitude response
- Phase response
- Magnitude and Phase responses
- Group delay response
- Phase delay response
- Impulse response
- Step response
- Pole-zero plot
- Zero-phase response — available from the *y*-axis context menu in a Magnitude or Magnitude and Phase response plot.

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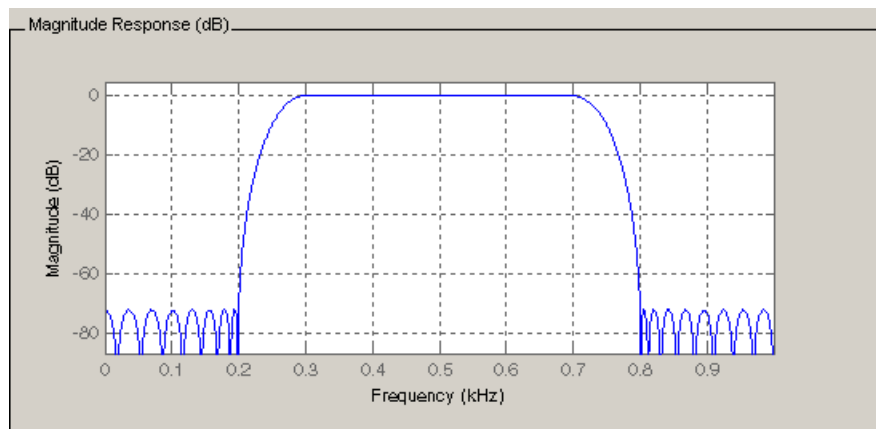
You can also overlay the filter specifications on the Magnitude plot by selecting **View** > **Specification Mask**.

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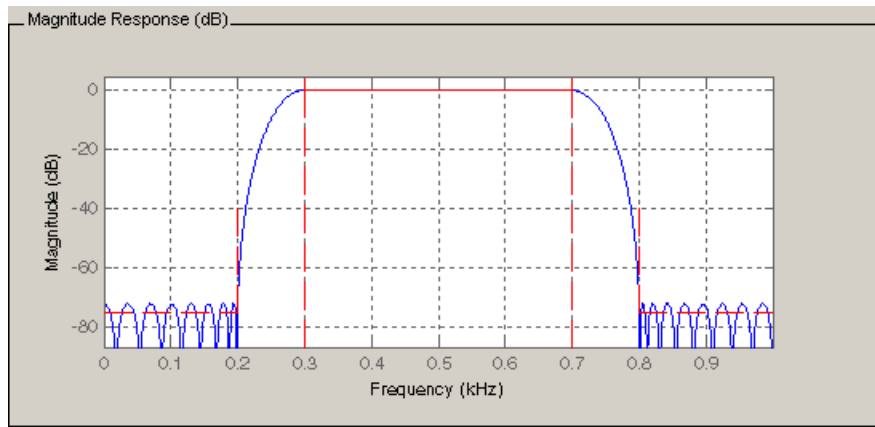
Using Data Tips

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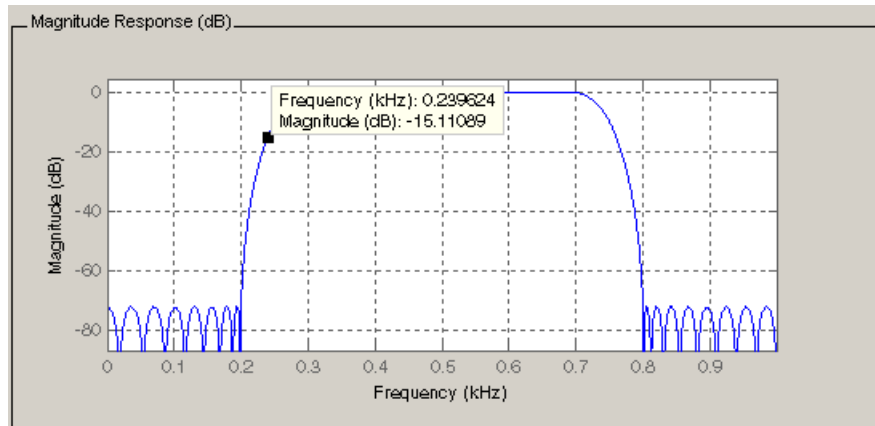
For information on using data tips, see "Display Data Values Interactively" in the MATLAB documentation.

Drawing Spectral Masks

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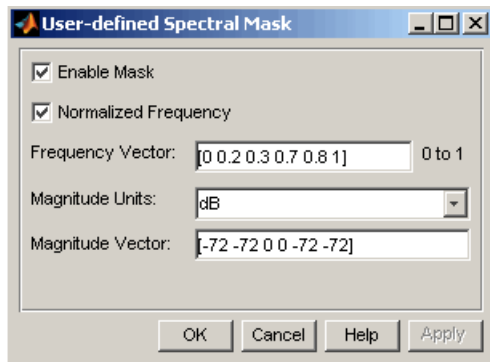
The mask is defined by a frequency vector and a magnitude vector. These vectors must be the same length.

*   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.
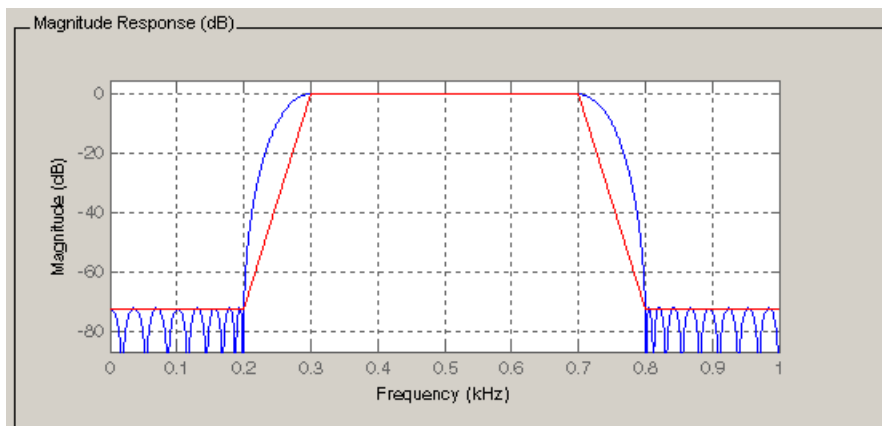
Changing the Sampling Frequency

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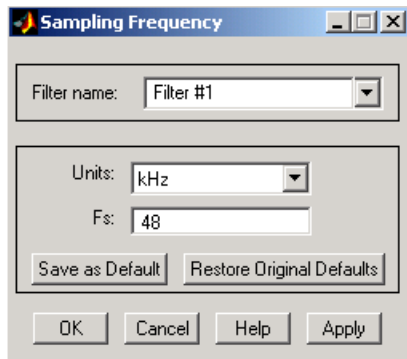
To change the filter name, type the new name in **Filter name**. (In `fvtool`, if you have multiple filters, select the desired filter and then enter the new name.)

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.
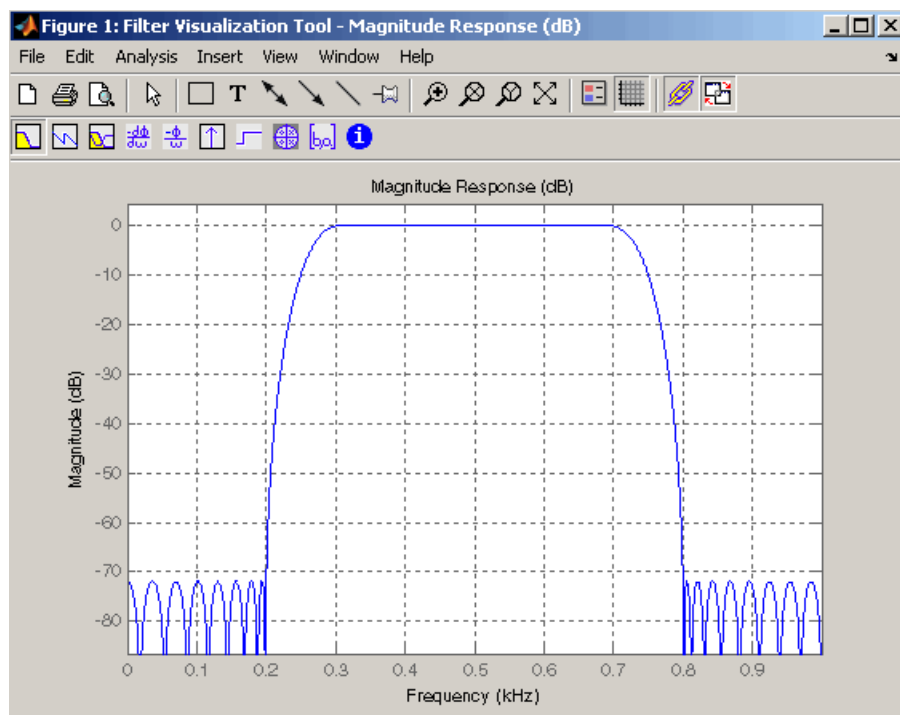
Editing the Filter Using the Pole/Zero Editor

- "Displaying the Pole-Zero Plot" on page 5-20
- "Changing the Pole-Zero Plot" on page 5-21

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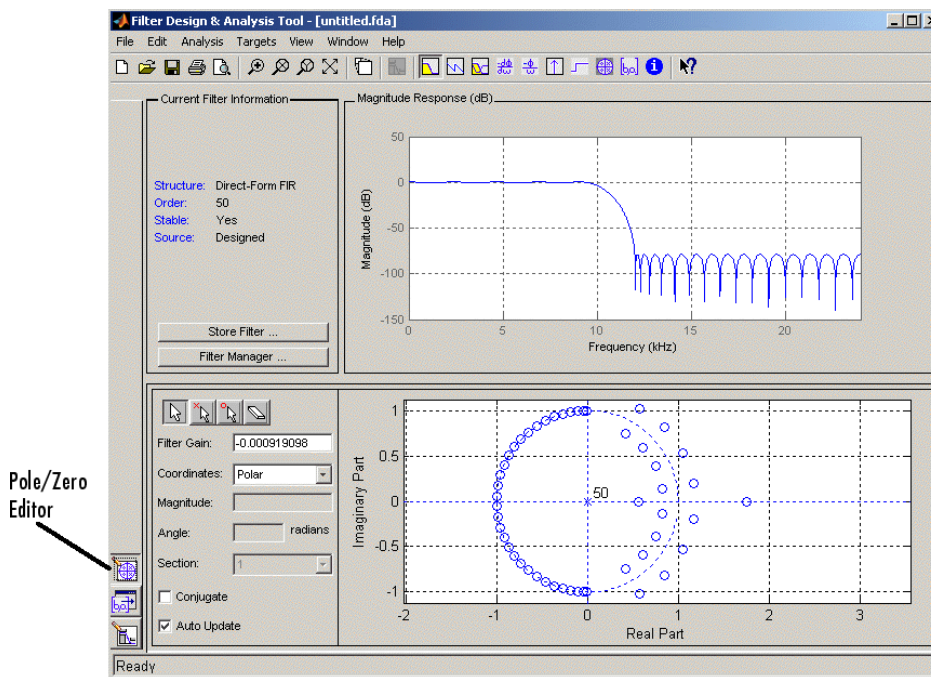
Poles are shown using x symbols and zeros are shown using o symbols.

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.
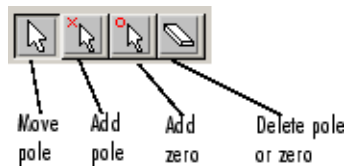
The following plot parameters and controls are located to the left of the pole/zero plot and below the plot mode buttons.

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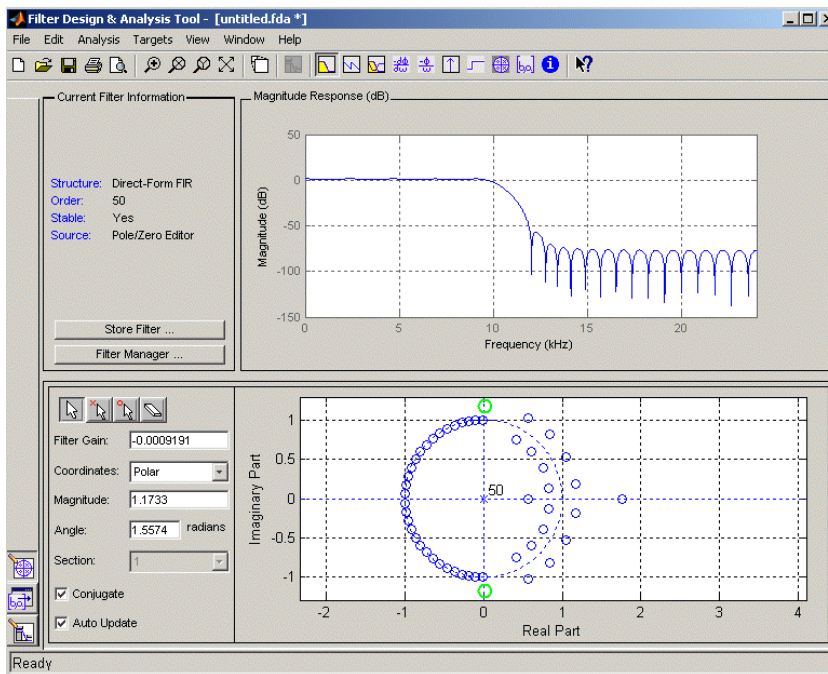
- The selected zero pair is shown in green.
- When you select one of the zeros from a conjugate pair, the Conjugate check box and the conjugate are automatically selected.
- The Magnitude Response plot updates immediately because **Auto update** is active.

Converting the Filter Structure

- "Converting to a New Structure" on page 5-24
- "Converting to Second-Order Sections" on page 5-26

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.
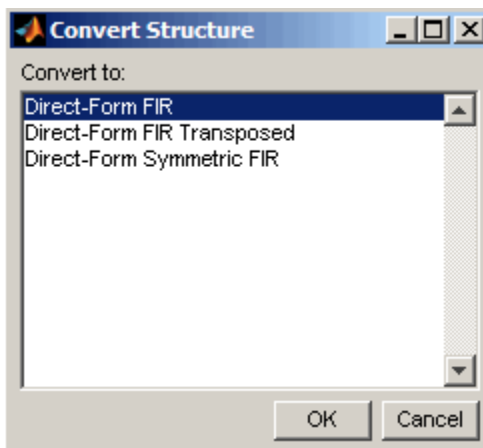
Converting to Second-Order Sections

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**.

Exporting a Filter Design

- "Exporting to a Simulink Model" on page 5-29
- "Other Ways to Export a Filter" on page 5-32

Exporting Coefficients or Objects to the Workspace

You can save the filter either as filter coefficients variables or as a filter object variable. To save the filter to the MATLAB workspace:

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

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

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

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

   For objects, assign the variable name in the **Discrete Filter (**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.

Exporting to SPTool

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

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 Designer™ installed, you can export a CIC filter to a Simulink model.

---

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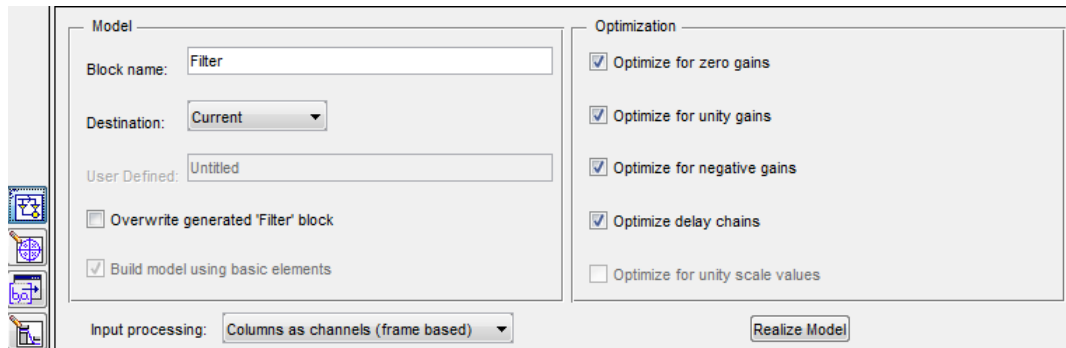
2   Specify the name to use for your block in **Block name**.

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

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

5   If you select the **Build model using basic elements** check box, your filter is created as a subsystem 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:
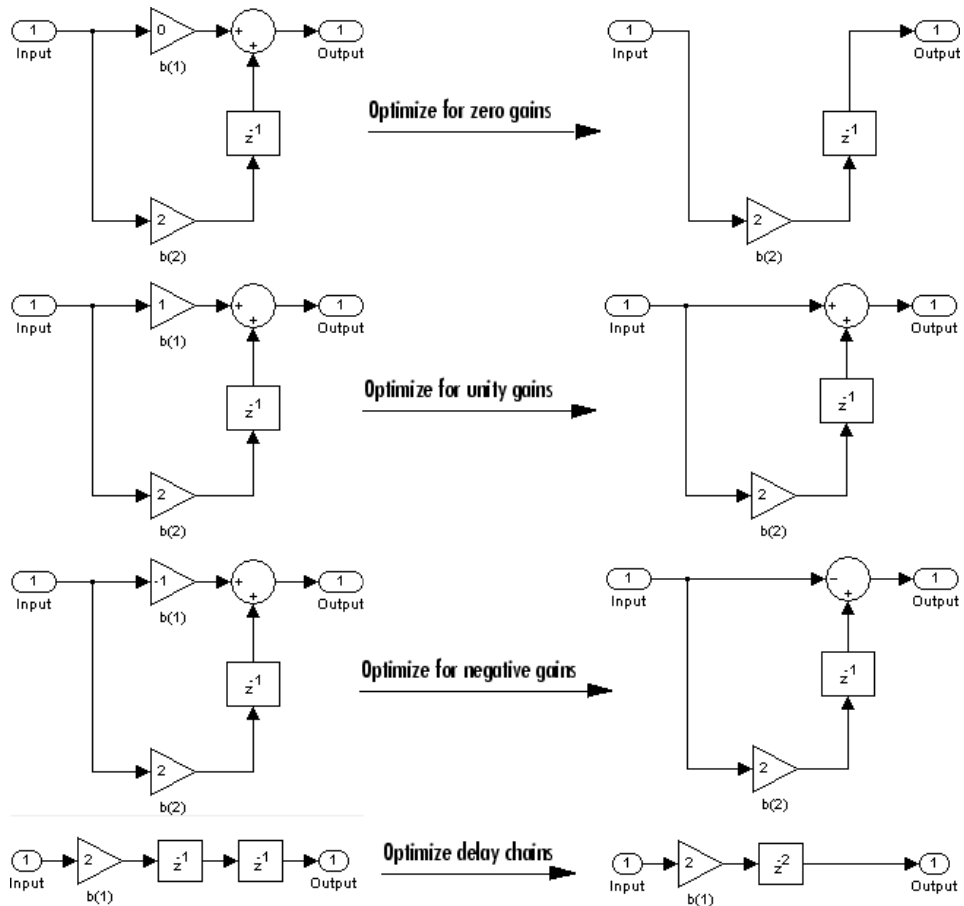
Optimization Effects

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 Biquad Filter block or a Discrete FIR Filter block. For more information, see the Filter Realization Wizard topic in the DSP System Toolbox documentation.

---

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

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

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

7 Click the **Realize Model** button to create the filter block. When the **Build model using basic elements** check box is selected, 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.

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 5-32
- "Generating MATLAB Code" on page 5-34

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)
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** > **Filter Design Function** and specify the filename in the Generate MATLAB code dialog box.

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

The following is generated MATLAB code for the default lowpass filter in 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.
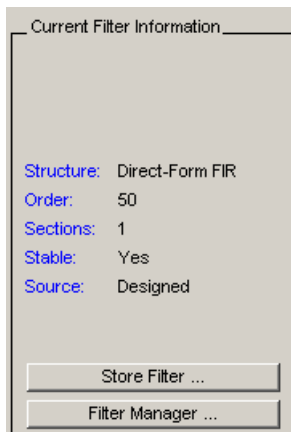
Store Filter — Displays the Store Filter dialog box in which you specify the filter name to use when storing the filter in the Filter Manager. The default name is the type of the filter.
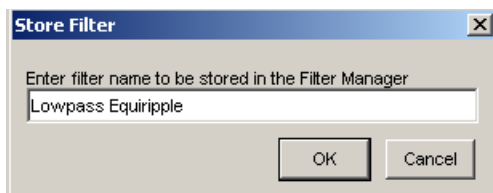
Filter Manager — Opens the Filter Manager.
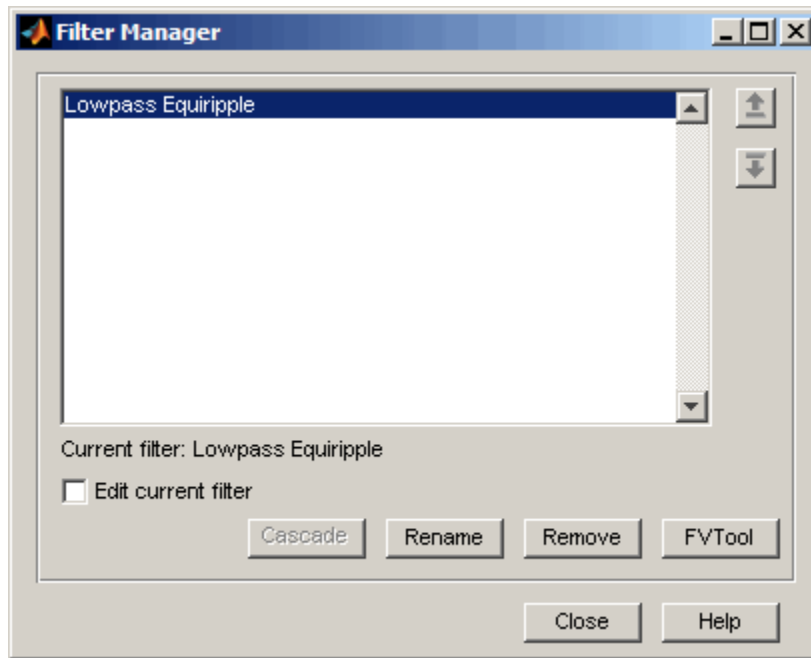
The current filter is listed below the listbox. To change the current filter, highlight the desired filter. If you select **Edit current filter**, FDATool displays the currently selected filter specifications. If you make any changes to the specifications, the stored filter is updated immediately.

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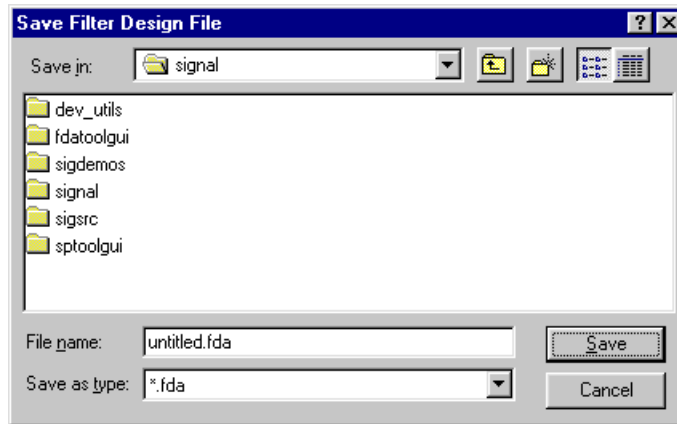
For example, save this design session as `TestFilter.fda` in your current working directory by typing `TestFilter` in the **File name** field.

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.

Importing a Filter Design

In this section...
"Import Filter Panel" on page 5-38
"Filter Structures" on page 5-39

Import Filter Panel

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

The imported filter can be in any of the representations listed in the **Filter Structure** pull-down menu. You can import a filter as second-order sections by selecting the check box.

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

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

Filter Structures

The available filter structures are:

- Direct Form, which includes direct-form I, direct-form II, direct-form I transposed, direct-form II transposed, and direct-form FIR
- Lattice, which includes lattice allpass, lattice MA min phase, lattice MA max phase, and lattice ARMA
- Discrete–time Filter (`dfilt` object)

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

Direct-form

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

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

- 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" for more information.

Importing as second-order sections

For all direct-form structures, except direct-form FIR, you can import the filter in its second-order section representation:

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

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

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

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

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

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

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

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

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

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

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

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

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

where we assume for this discussion that $x(n)$ and $y(n)$ are indexed from 0 to $N-1$, and $\hat{R}_{xy}(m)$ from $-(N-1)$ to $N-1$.

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

In this section...

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:

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

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

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

The average power of the sequence $x(n)$ over the entire Nyquist interval is represented by

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

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

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

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

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

Spectral Estimation Method

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

- Nonparametric methods
- Parametric methods
- Subspace methods

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

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

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

All three categories of methods are listed in the table below with the corresponding toolbox function names. More information about each function is on the corresponding function reference page. See "Parametric Modeling" on page 7-17 for details about `lpc` and other parametric estimation functions.

Spectral Estimation Methods/Functions

Method	Description	Functions
Periodogram	Power spectral density estimate	`periodogram`
Welch	Averaged periodograms of overlapped, windowed signal sections	`pwelch`, `cpsd`, `tfestimate`, `mscohere`

Method	Description	Functions
Multitaper	Spectral estimate from combination of multiple orthogonal windows (or "tapers")	`pmtm`
Yule-Walker AR	Autoregressive (AR) spectral estimate of a time-series from its estimated autocorrelation function	`pyulear`
Burg	Autoregressive (AR) spectral estimation of a time-series by minimization of linear prediction errors	`pburg`
Covariance	Autoregressive (AR) spectral estimation of a time-series by minimization of the forward prediction errors	`pcov`
Modified Covariance	Autoregressive (AR) spectral estimation of a time-series by minimization of the forward and backward prediction errors	`pmcov`
MUSIC	Multiple signal classification	`pmusic`
Eigenvector	Pseudospectrum estimate	`peig`

Nonparametric Methods

The following sections discuss the periodogram, modified periodogram, Welch, and multitaper methods of nonparametric estimation, along with the related 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 using `periodogram`. In this case, the data vector is multiplied by a Hamming window to produce a modified periodogram.

```
[Pxx,F] = periodogram(xn,hamming(length(xn)),length(xn),fs);
plot(F,10*log10(Pxx))
xlabel('Hz')
ylabel('dB')
title('Modified Periodogram Power Spectral Density Estimate')
```

Modified Periodogram Power Spectral Density Estimate
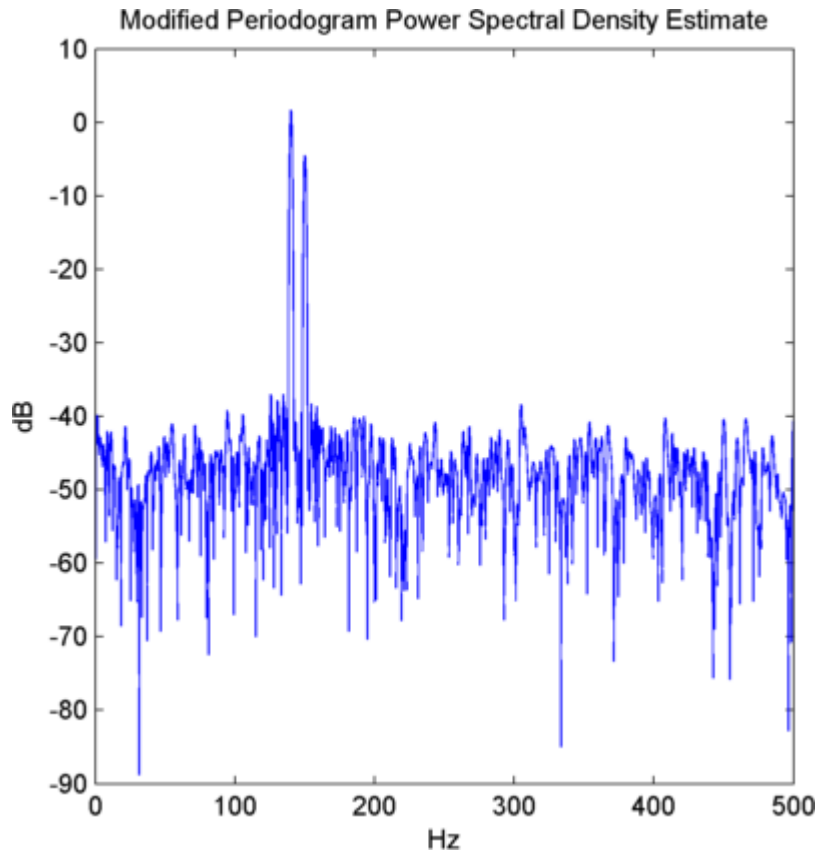
Algorithm

Periodogram computes and scales the output of the FFT to produce the power vs. frequency plot as follows. For a detailed example, see "Power Spectral Density Estimates Using FFT".

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

2   Take the squared magnitudes of the unique FFT values. Scale the squared magnitudes (except for DC) by $2/(F_s N)$, where N is the length of signal *prior* to any zero padding. Scale the DC value by $1/(F_s N)$.

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

4 Plot the resulting magnitude squared FFT against the frequency.

Performance of the Periodogram

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

Spectral Leakage

Consider the PSD of a finite-length (length L) signal $x_L(n)$, 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

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

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

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

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

Its spectrum is

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

which for a finite-length sequence becomes

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

The preceding equation is equal to:

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

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

The frequency response of a rectangular window has the shape of a sinc signal, as shown below.
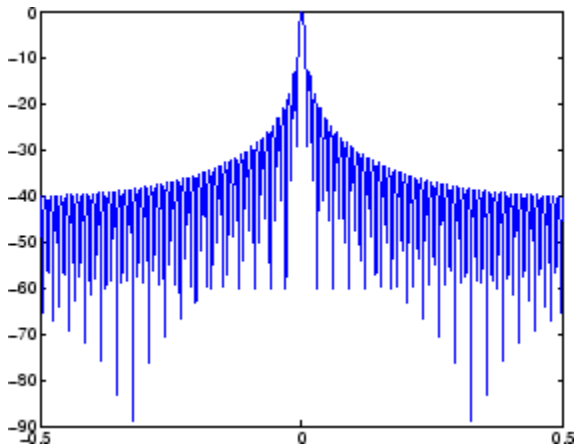
The plot displays a mainlobe and several sidelobes, the largest of which is approximately 13.5 dB below the mainlobe peak. These lobes account for the effect known as *spectral*

leakage. While the infinite-length signal has its power concentrated exactly at the discrete frequency points f_k, the windowed (or truncated) signal has a continuum of power "leaked" around the discrete frequency points f_k.

Because the frequency response of a short rectangular window is a much poorer approximation to the Dirac delta function than that of a longer window, spectral leakage is especially evident when data records are short. Consider the following sequence of 100 samples:

```
fs = 1000;                    % Sampling frequency
t = (0:fs/10)/fs;             % One-tenth second worth of samples
A = [1 2];                    % Sinusoid amplitudes
f = [150;140];                % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
[Pxx,F] = periodogram(xn,rectwin(length(xn)),length(xn),fs);
plot(F,10*log10(Pxx))
```
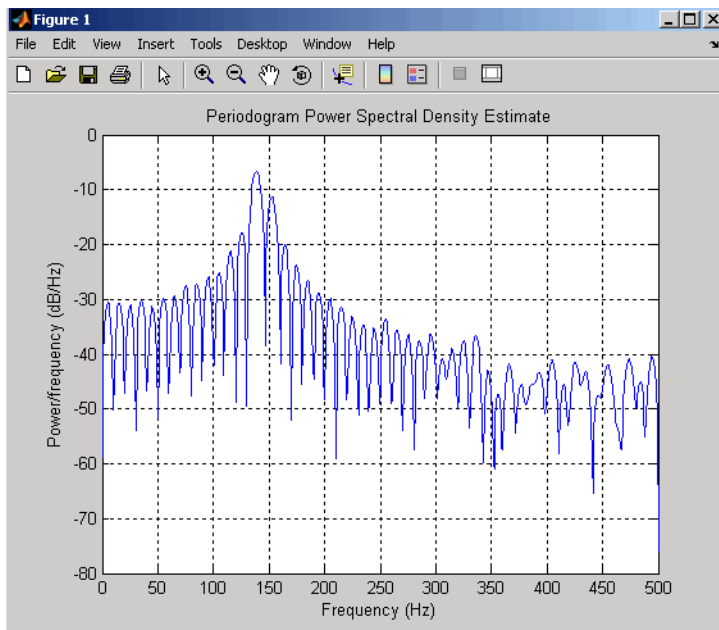

It is important to note that the effect of spectral leakage is contingent solely on the length of the data record. It is *not* a consequence of the fact that the periodogram is computed at a finite number of frequency samples.

Resolution

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

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

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

$$f_2 - f_1 > \frac{F_s}{L}.$$

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

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

```
fs = 1000;                    % Sampling frequency
t = (0:fs/15)/fs;             % 67 samples
A = [1 2];                    % Sinusoid amplitudes
f = [150;140];                % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
[Pxx,F] = periodogram(xn,rectwin(length(xn)),length(xn),fs);
plot(F,10*log10(Pxx))
```
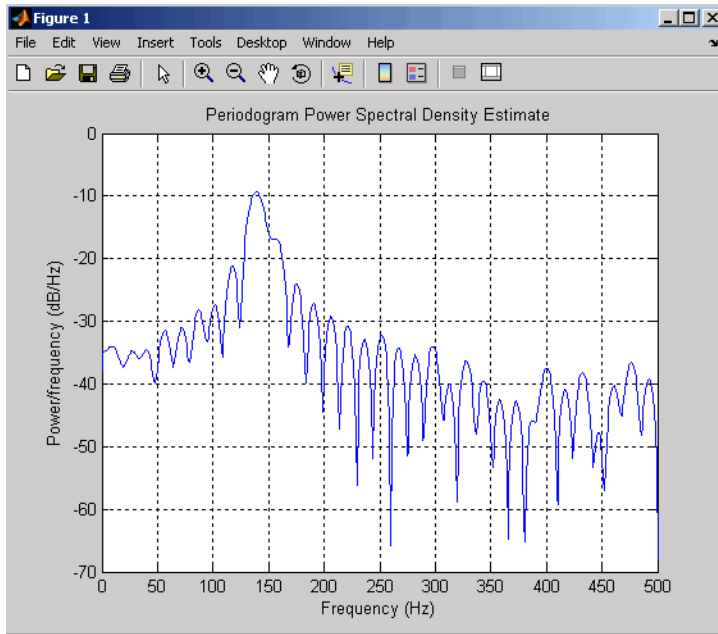
The above discussion about resolution did not consider the effects of noise since the signal-to-noise ratio (SNR) has been relatively high thus far. When the SNR is low, true spectral features are much harder to distinguish, and noise artifacts appear in spectral estimates based on the periodogram. The 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));
[Pxx,F] = periodogram(xn,rectwin(length(xn)),length(xn),fs);
plot(F,10*log10(Pxx))
```

Periodogram Power Spectral Density Estimate
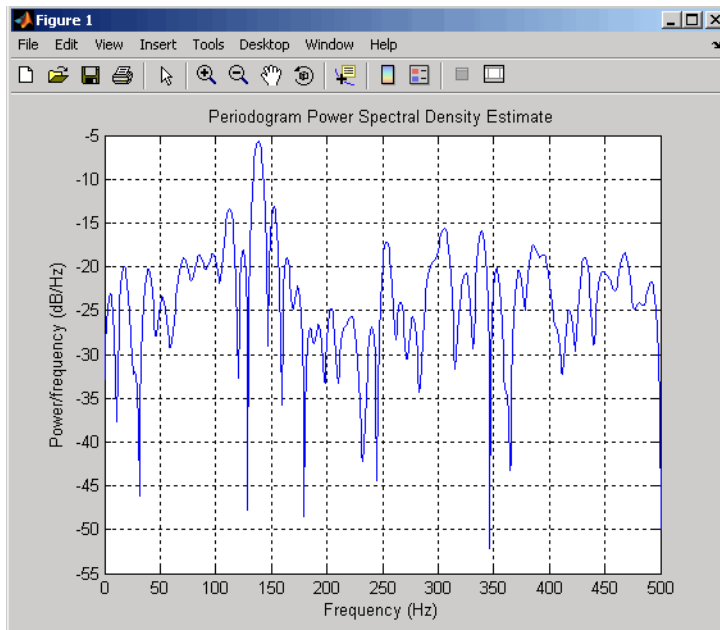
Bias of the Periodogram

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

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

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

Variance of the Periodogram

The variance of the periodogram can be shown to be

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

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

The Modified Periodogram

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

`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));
[Pxx,F] = periodogram(xn,rectwin(length(xn)),length(xn),fs);
plot(F,10*log10(Pxx))
```
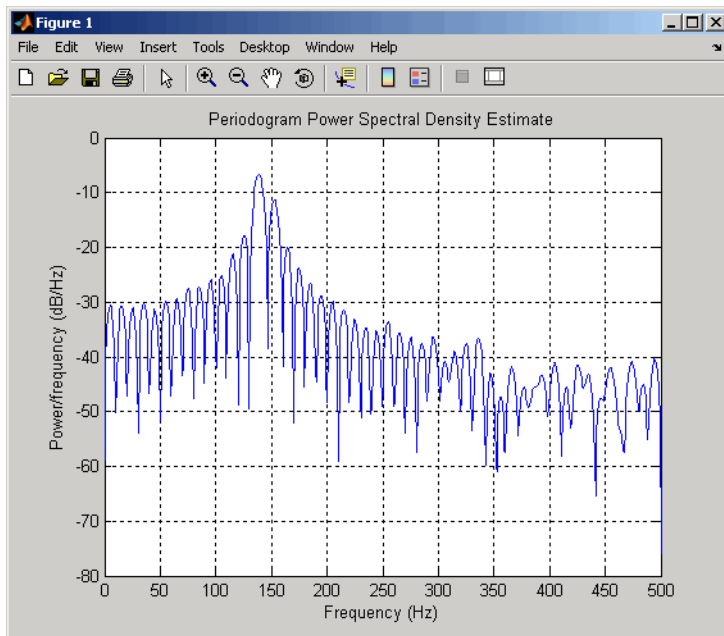
```
[Pxx,F] = periodogram(xn,hamming(length(xn)),length(xn),fs);
plot(F,10*log10(Pxx))
```
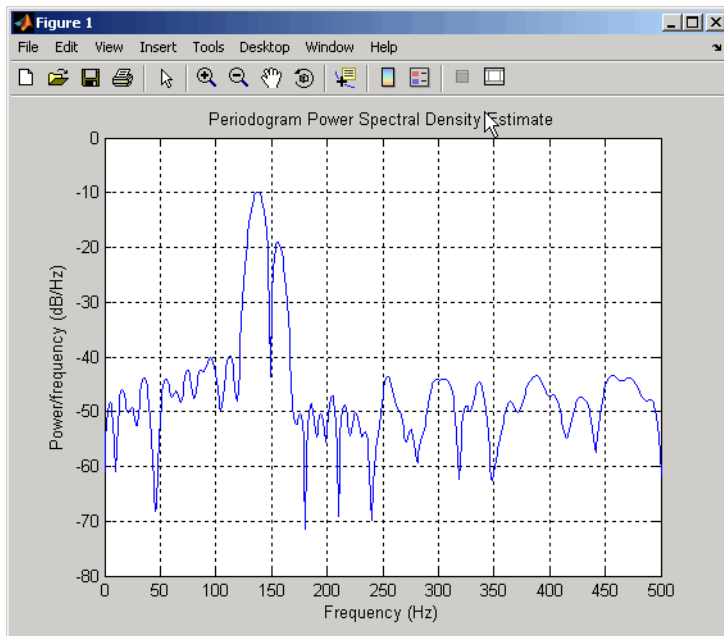
You can verify that although the sidelobes are much less evident in the Hamming-windowed periodogram, the two main peaks are wider. In fact, the 3 dB width of the mainlobe corresponding to a Hamming window is approximately twice that of a rectangular window. Hence, for a fixed data length, the PSD resolution attainable with a Hamming window is approximately half that attainable with a rectangular window. The competing interests of mainlobe width and sidelobe height can be resolved to some extent by using variable windows such as the Kaiser window.

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

The modified periodogram estimate of the PSD is

$$\hat{P}_{xx}(f) = \frac{|X(f)|^2}{F_s L U},$$

where U is the window normalization constant:

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

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

Welch's Method

An improved estimator of the PSD is the one proposed by Welch [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. The toolbox function `pwelch` implements Welch's method.

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

However, as mentioned above, the combined use of short data records and nonrectangular windows results in reduced resolution of the estimator. In summary, there is a 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 1000 samples:

```
fs = 1000;              % Sampling frequency
t = (0:1*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));
[Pxx,F] = periodogram(xn,rectwin(length(xn)),length(xn),fs);
plot(F,10*log10(Pxx))
```
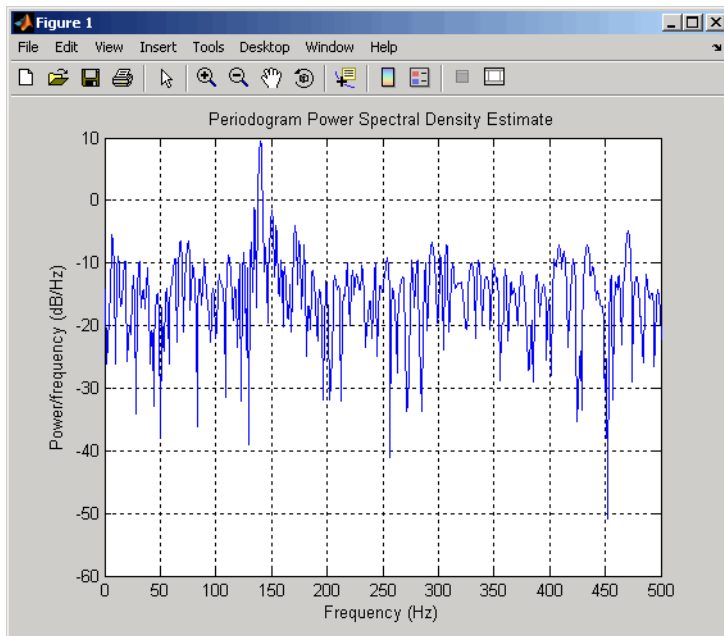
We can obtain Welch's spectral estimate for 3 segments with 50% overlap using a Hamming window.

```
[Pxx,F] = pwelch(xn,hamming(150),75,150,fs);
plot(F,10*log10(Pxx))
xlabel('Hz')
ylabel('dB')
title('Welch''s Overlapped Segment Averaging PSD Estimate')
```
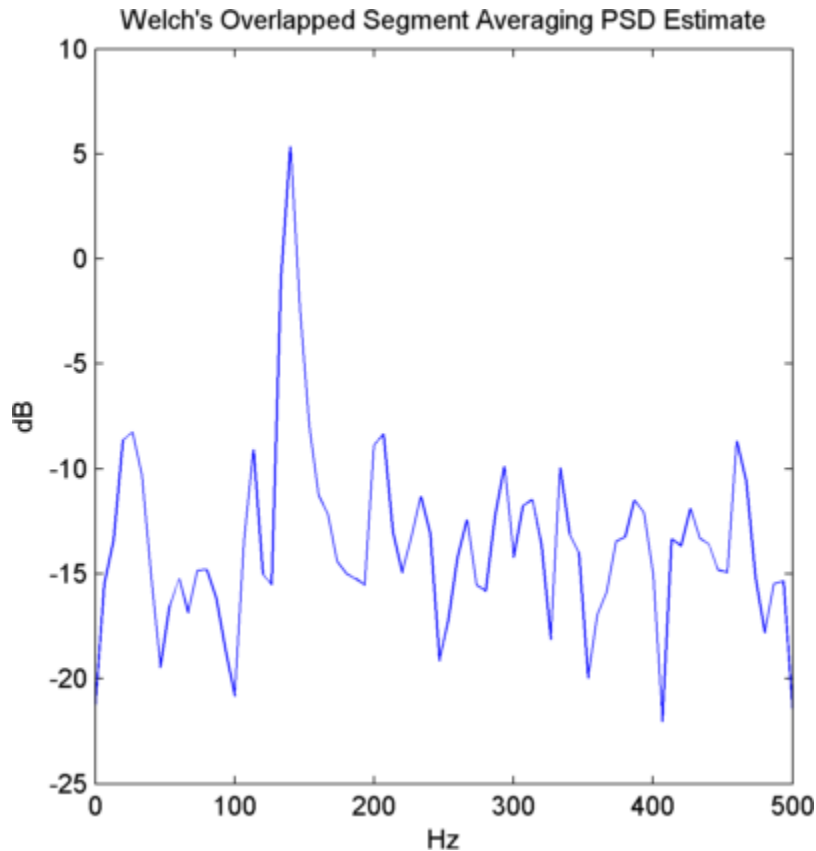
In the periodogram above, noise and the leakage make one of the sinusoids essentially indistinguishable from the artificial peaks. In contrast, although the PSD produced by Welch's method has wider peaks, you can still distinguish the two sinusoids, which stand out from the "noise floor."

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

```
[Pxx,F] = pwelch(xn,rectwin(100),75,512,Fs);
plot(F,10*log10(Pxx))
```
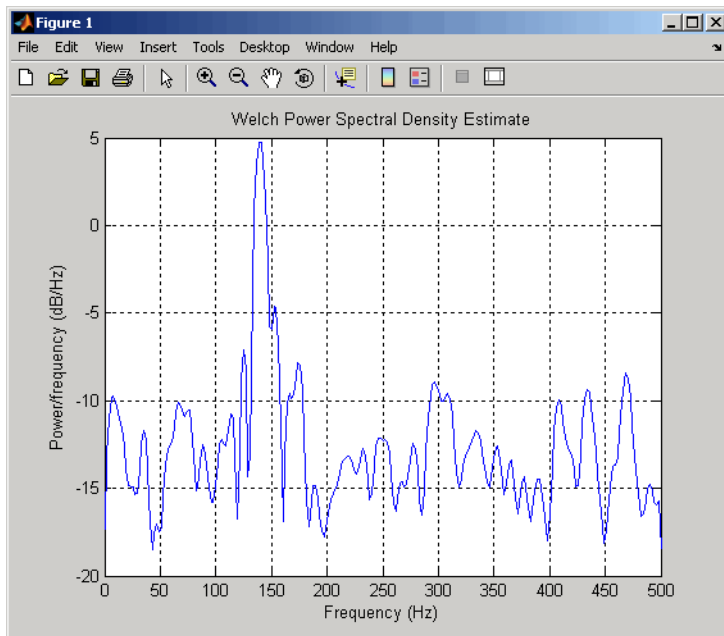
For a more detailed discussion of Welch's method of PSD estimation, see Kay [2] and Welch [8].

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:

$$E\{P_{\text{Welch}}(f)\} = \frac{1}{F_s L U} \int_{-F_s/2}^{F_s/2} |W(f - f')|^2 P_{xx}(f')df',$$

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

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

Multitaper Method

The periodogram can be interpreted as filtering a length L signal, $x_L(n)$, through a filter bank (a set of filters in parallel) of L FIR bandpass filters. The 3 dB bandwidth of each of these bandpass filters can be shown to be approximately equal to f_s / L. The magnitude response of each one of these bandpass filters resembles that of 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 $2NW - 1$ tapers used to form the estimate. This means that, as *NW* increases, there are more estimates of the power spectrum, and the variance of the estimate decreases. However, the bandwidth of each taper is also proportional to *NW*, so as *NW* increases, each estimate exhibits more spectral leakage (i.e., wider peaks) and the overall spectral estimate is more biased. For each data set, there is usually a value for *NW* that allows an optimal trade-off between bias and variance.

The Signal Processing Toolbox function that implements the MTM method is `pmtm`. Use `pmtm` to compute the PSD of `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));
[Pxx1,F1] = pmtm(xn,4,fs);
plot(F1,10*log10(Pxx1))
```
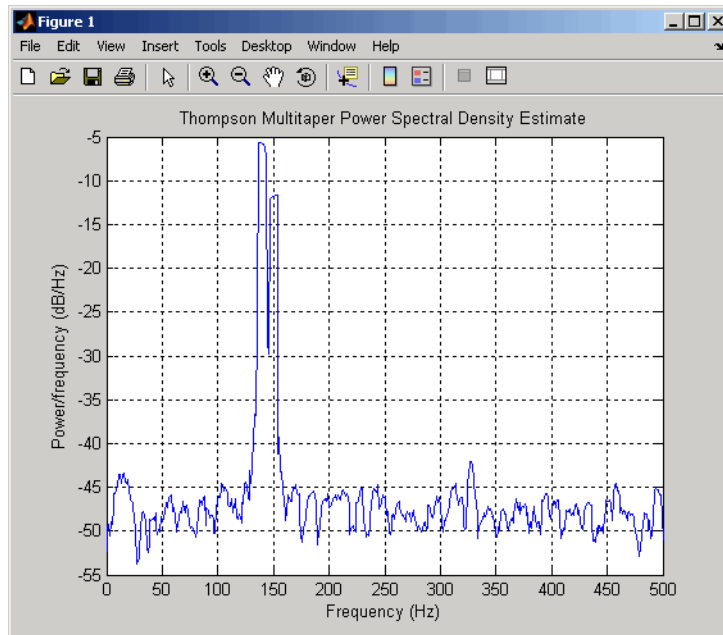

By lowering the time-bandwidth product, you can increase the resolution at the expense of larger variance:

```
[Pxx2,F2] = pmtm(xn,1.5,fs);
plot(F2,10*log10(Pxx2))
```
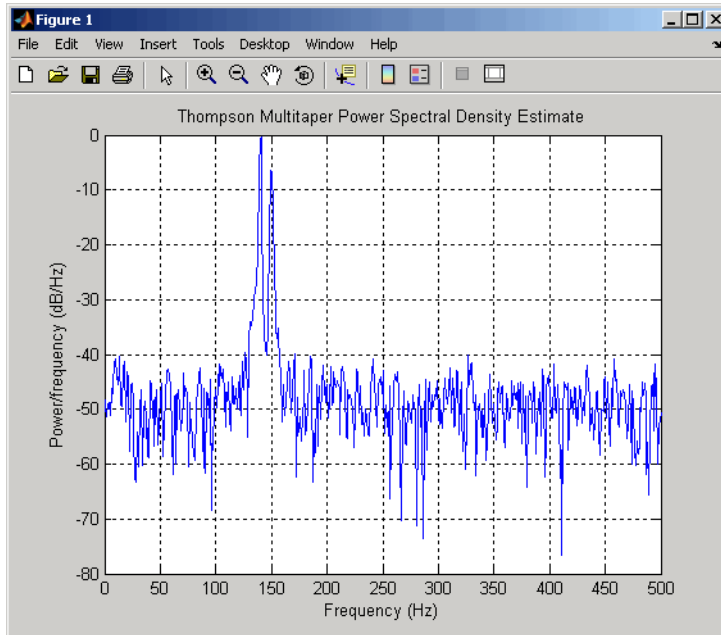

This method is more computationally expensive than Welch's method due to the cost of computing the discrete prolate spheroidal sequences. For long data series (10,000 points or more), it is useful to compute the DPSSs once and save them in a MAT-file. `dpsssave`, `dpssload`, `dpssdir`, and `dpssclear`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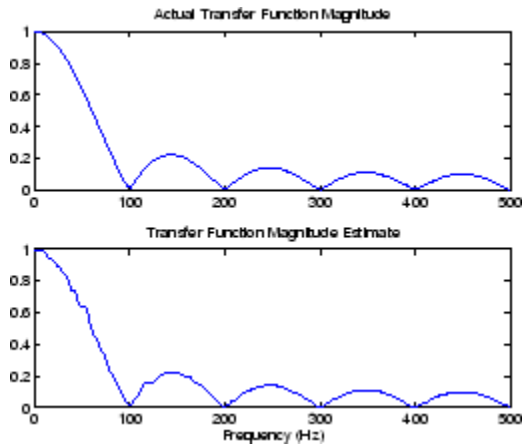

Coherence Function

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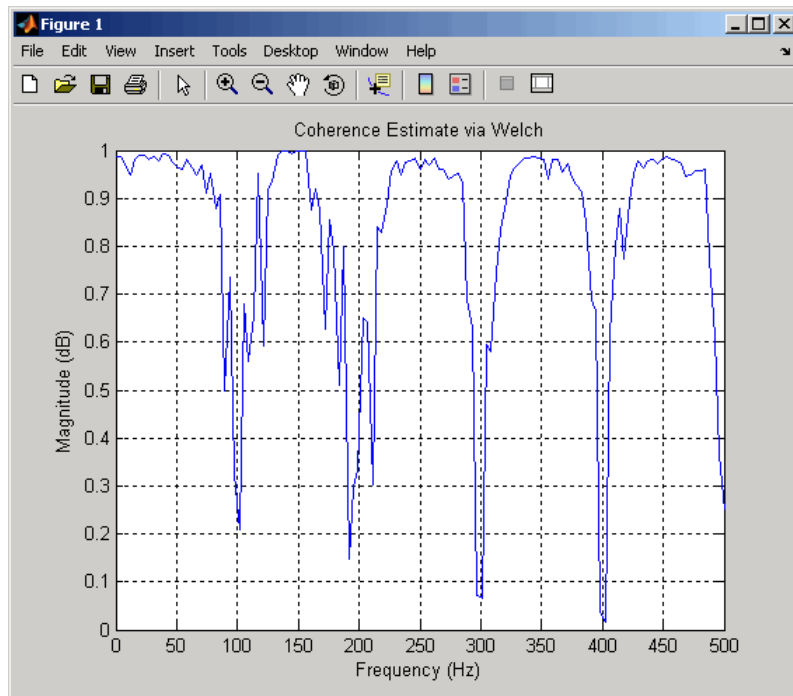
If the input sequence length `nfft`, window length `window`, and the number of overlapping data points in a window, `numoverlap`, are such that `mscohere` operates on only a single record, the function returns all ones. This is because the coherence function for linearly dependent data is one.

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

$$\hat{P}(f) = \frac{1}{F_s} \frac{\varepsilon_p}{\left| 1 - \sum_{k=1}^{p} \hat{a}_p(k) e^{-j2\pi k f / F_s} \right|^2}.$$

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

AR Methods

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

	Burg	Covariance	Modified Covariance	Yule-Walker
	or when order is very large			
	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:

$$
\begin{pmatrix}
r(0) & r^*(1) & \dots & r^*(p-1) \\
r(1) & r(0) & \dots & r^*(p-2) \\
r(p-1) & r(p-2) & \dots & r(0)
\end{pmatrix}
\begin{pmatrix}
a(1) \\
a(2) \\
\vdots \\
a(p)
\end{pmatrix}
=
\begin{pmatrix}
r(1) \\
r(2) \\
\vdots \\
r(p)
\end{pmatrix}.
$$

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 6-44.

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

```
load mtlb
[Pxx,F] = pwelch(mtlb,hamming(256),128,1024,Fs);
plot(F,10*log10(Pxx))
```
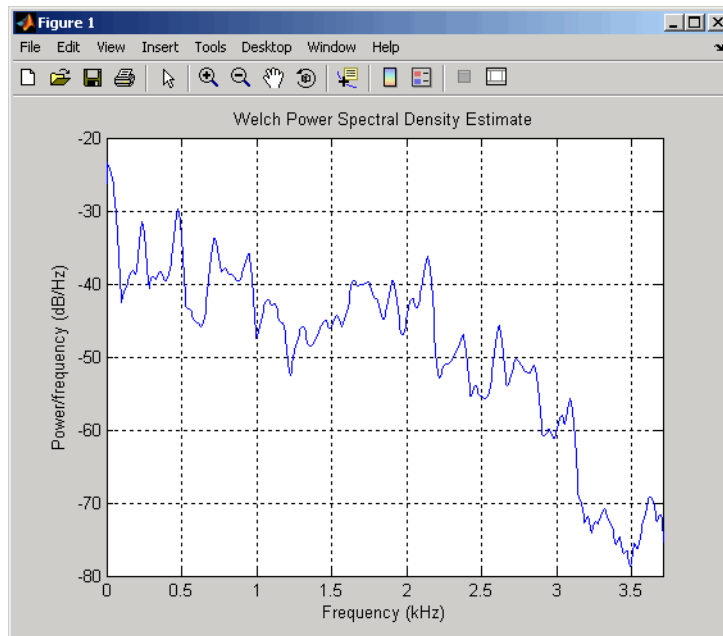


```
order = 14;
[Pxx,F] = pyulear(mtlb,order,1024,fs);
plot(F,10*log10(Pxx))
```
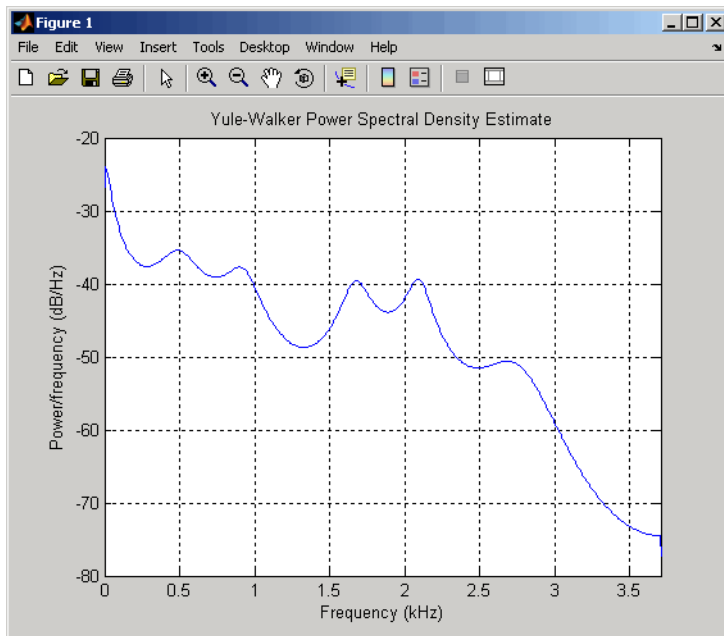
The Yule-Walker AR spectrum is smoother than the periodogram because of the simple underlying all-pole model.

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

```
load mtlb
order = 14;
[Pburg,F] = pburg(mtlb(1:512),order,1024,fs);
plot(F,10*log10(Pburg))
```
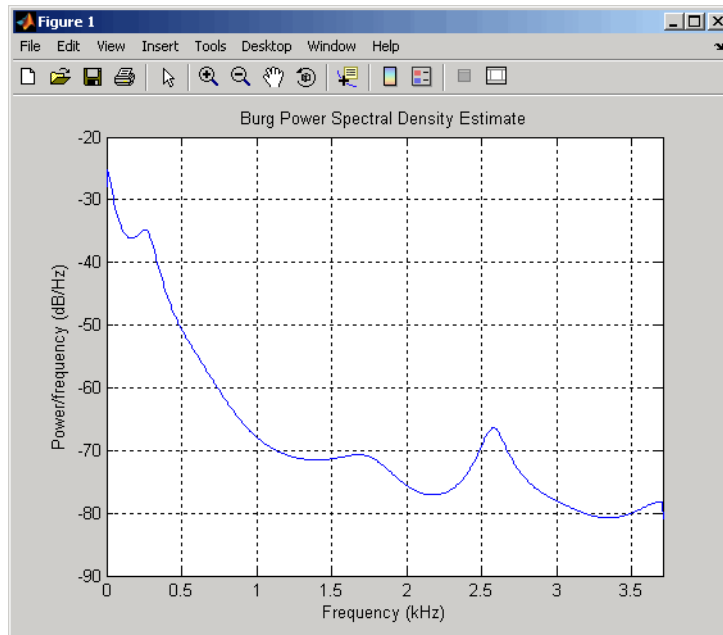


```
[Pxx,F] = pyulear(mtlb(1:512),ORDER,1024,fs);
plot(F,10*log10(Pxx))
```
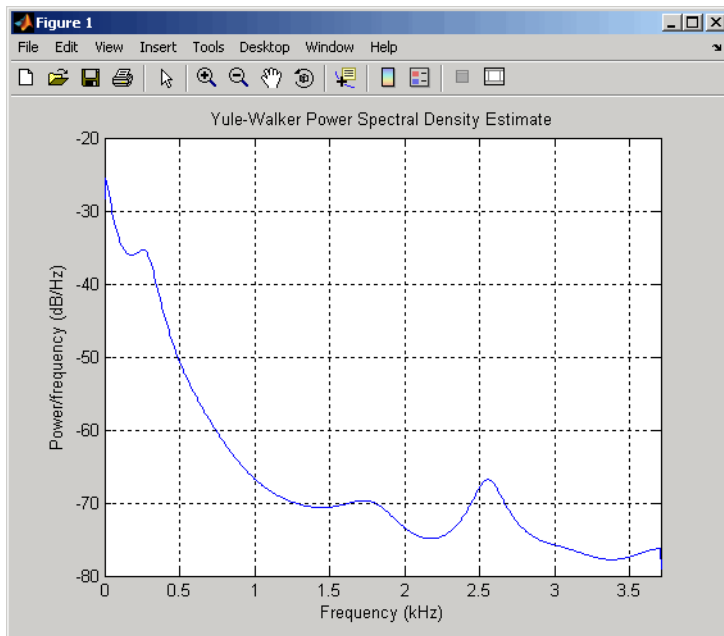
Compare the spectrum of a noisy signal computed using the Burg method and the Welch method:

```
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));
pwelch(xn,hamming(256),128,1024,fs)
```
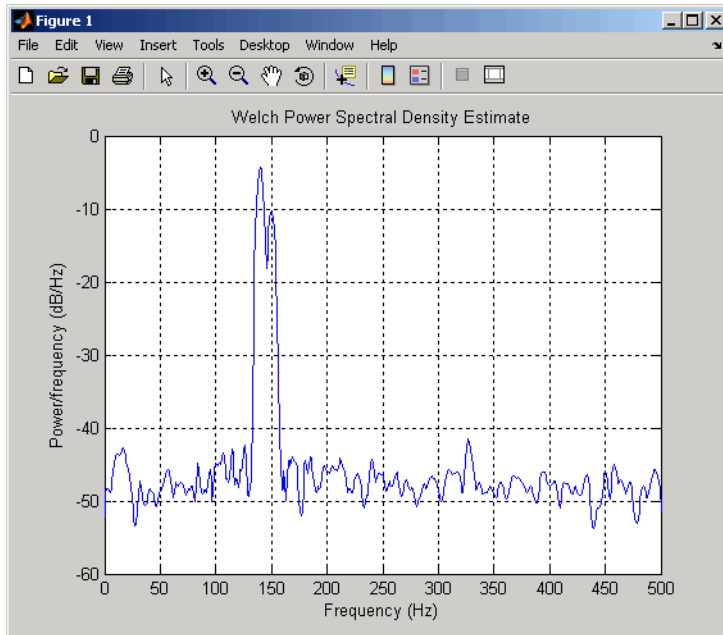
```
pburg(xn,14,1024,fs)
```
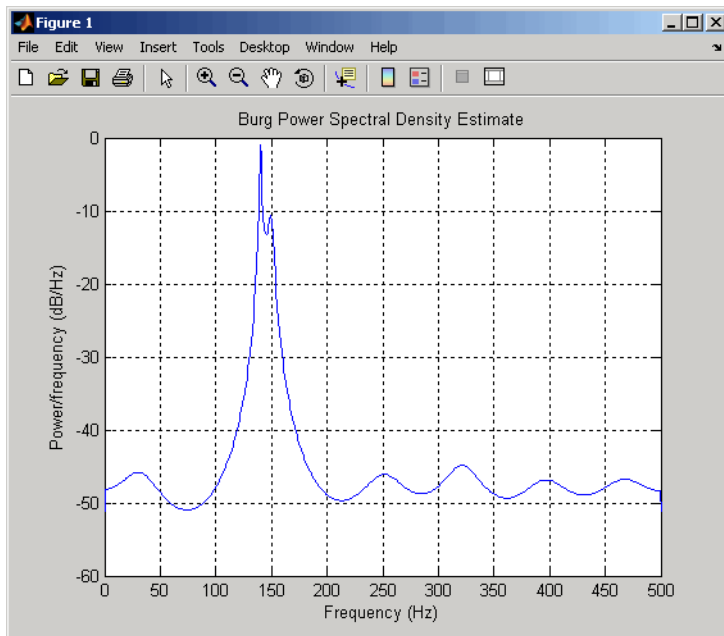
Note that, as the model order for the Burg method is reduced, a frequency shift due to the initial phase of the sinusoids will become apparent.

Covariance and Modified Covariance Methods

The covariance method for AR spectral estimation is based on minimizing the forward prediction error. The modified covariance method is based on minimizing the forward and backward prediction errors. The toolbox functions pcov and pmcov implement the respective methods.

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

```
load mtlb
pcov(mtlb(1:64),14,1024,fs)
```
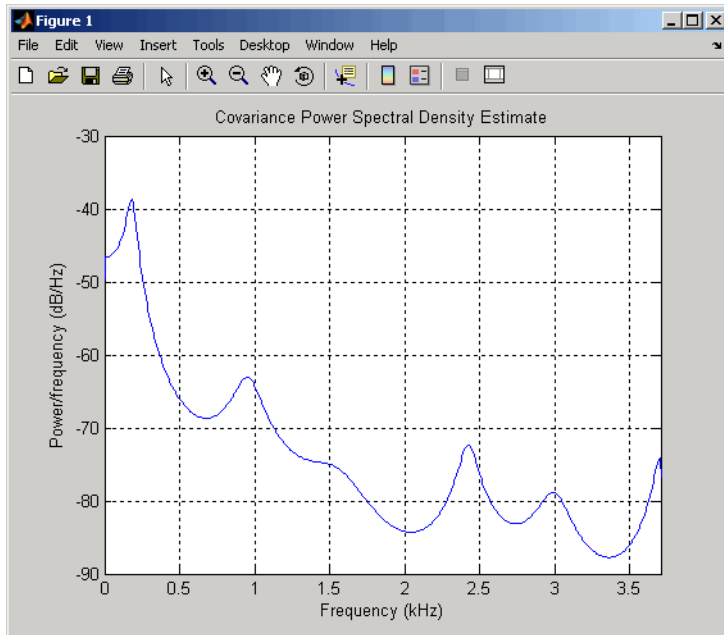
```
pmcov(mtlb(1:64),14,1024,fs)
```
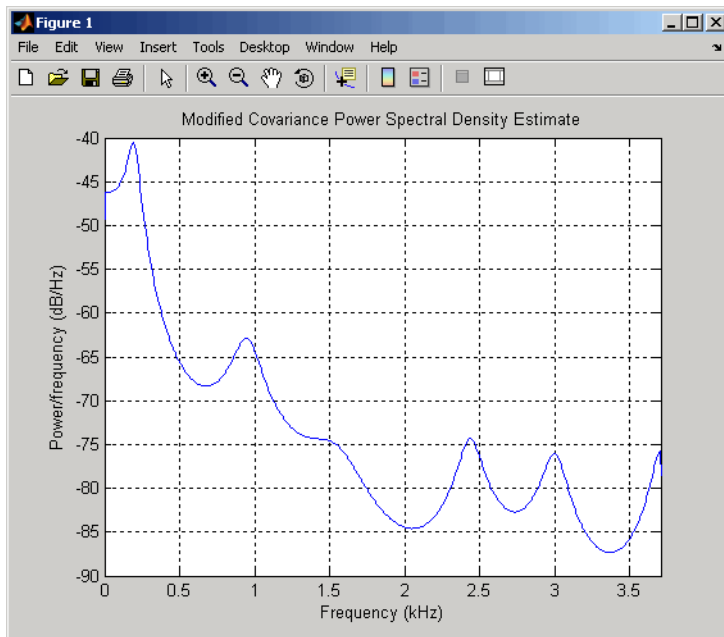
MUSIC and Eigenvector Analysis Methods

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

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

Both of these methods are frequency estimator techniques based on eigenanalysis of the autocorrelation matrix. This type of spectral analysis categorizes the information in a correlation or data matrix, assigning information to either a signal subspace or a noise subspace.

Eigenanalysis Overview

Consider a number of complex sinusoids embedded in white noise. You can write the autocorrelation matrix R for this system as the sum of the signal autocorrelation matrix (S) and the noise autocorrelation matrix (W):

$R = S + W$. There is a close relationship between the eigenvectors of the signal autocorrelation matrix and the signal and noise subspaces. The eigenvectors v of S span the same signal subspace as the signal vectors. If the system contains M complex sinusoids and the order of the autocorrelation matrix is p, eigenvectors v_{M+1} through v_{p+1} span the noise subspace of the autocorrelation matrix.

Frequency Estimator Functions

To generate their frequency estimates, eigenanalysis methods calculate functions of the vectors in the signal and noise subspaces. Both the MUSIC and EV techniques choose a function that goes to infinity (denominator goes to zero) at one of the sinusoidal frequencies in the input signal. Using digital technology, the resulting estimate has sharp peaks at the frequencies of interest; this means that there might not be infinity values in the vectors.

The MUSIC estimate is given by the formula

$$\hat{P}_{\mathrm{MUSIC}}(f) = \frac{1}{\displaystyle\sum_{k=p+1}^{M} \left| e(f)^H v_k \right|^2},$$

where the v_k are the eigenvectors of the noise subspace and $e(f)$ is a vector of complex sinusoids:

$$e(f) = [1 \quad e^{j2\pi f} \quad e^{j4\pi f} \quad \dots \quad e^{j2(M-1)\pi f}]^T.$$

Here 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:

$$\hat{P}_{\mathrm{EV}}(f) = \frac{\lambda_k}{\displaystyle\sum_{k=p+1}^{M} \left| e^H(f) v_k \right|^2}.$$

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.

Selected Bibliography

[1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

[2] Kay, Steven M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[3] Marple, S. Lawrence *Digital Spectral Analysis*. Englewood Cliffs, NJ: Prentice Hall, 1987.

[4] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1996.

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

[6] Proakis, John G., and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1996.

[7] Stoica, Petre, and Randolph Moses. *Spectral Analysis of Signals*. Upper Saddle River, NJ: Prentice Hall, 1997.

[8] Welch, Peter D. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms." *IEEE Trans. Audio Electroacoust.*. Vol. AU-15, 1967, pp.70–73.

7

Special Topics

Windows

In this section...
"Why Use Windows?" on page 7-2
"Available Window Functions" on page 7-2
"Graphical User Interface Tools" on page 7-3
"Basic Shapes" on page 7-3

Why Use Windows?

In both digital filter design and spectral estimation, the choice of a windowing function can play an important role in determining the quality of overall results. The main role of the window is to damp out the effects of the Gibbs phenomenon that results from truncation of an infinite series.

Available Window Functions

Window	Function
Bartlett-Hann window	`barthannwin`
Bartlett window	`bartlett`
Blackman window	`blackman`
Blackman-Harris window	`blackmanharris`
Bohman window	`bohmanwin`
Chebyshev window	`chebwin`
Flat Top window	`flattopwin`
Gaussian window	`gausswin`
Hamming window	`hamming`
Hann window	`hann`
Kaiser window	`kaiser`
Nuttall's Blackman-Harris window	`nuttallwin`

Window	Function
Parzen (de la Vallée-Poussin) window	`parzenwin`
Rectangular window	`rectwin`
Tapered cosine window	`tukeywin`
Triangular window	`triang`

Graphical User Interface Tools

Two graphical user interface tools are provided for working with windows in the Signal Processing Toolbox product:

- Window 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**.
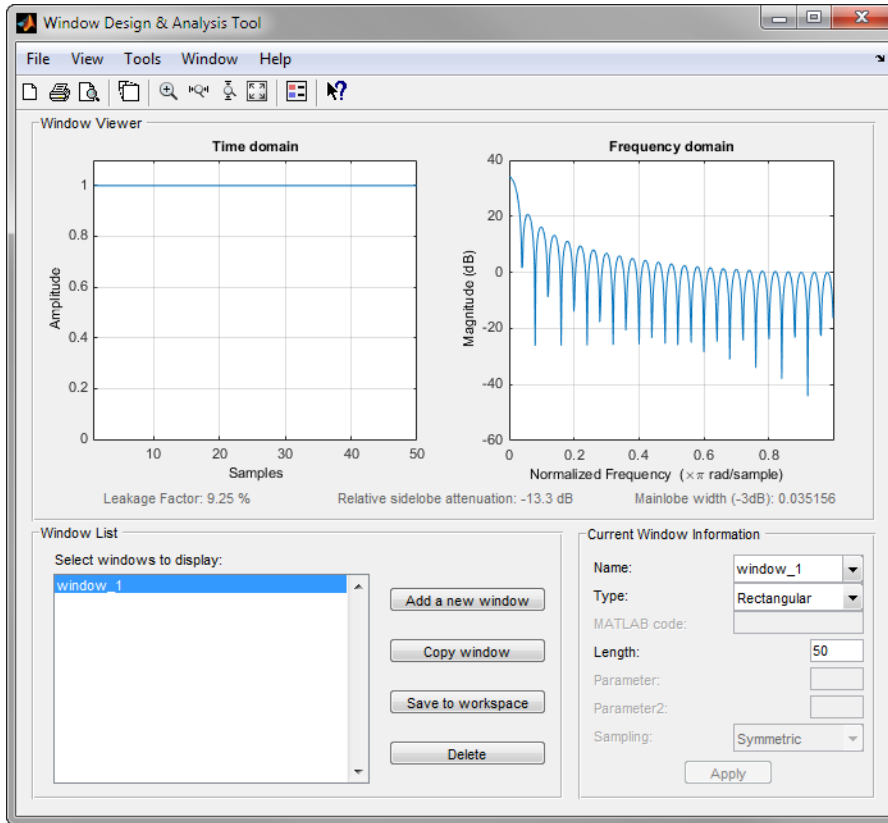
The *Bartlett* (or triangular) *window* is the convolution of two rectangular windows. The functions `bartlett` and `triang` compute similar triangular windows, with three important differences. The `bartlett` function always returns a window with two zeros on the ends of the sequence, so that for `n` odd, the center section of `bartlett(n+2)` is equivalent to `triang(n)`:

```
Bartlett = bartlett(7);
isequal(Bartlett(2:end-1),triang(5))

ans =
     1
```

For `n` even, `bartlett` is still the convolution of two rectangular sequences. There is no standard definition for the triangular window for `n` even; the slopes of the line segments of the `triang` result are slightly steeper than those of `bartlett` in this case:

```
w = bartlett(8);
[w(2:7) triang(6)]
```

You can see the difference between odd and even Bartlett windows in WinTool.
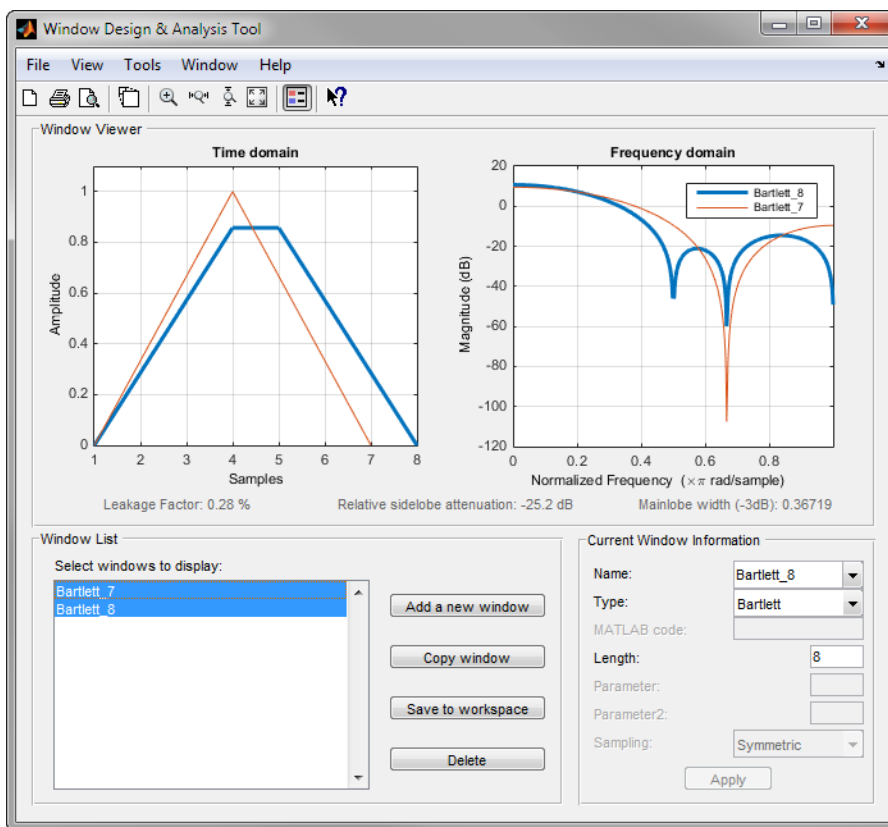
The final difference between the Bartlett and triangular windows is evident in the Fourier transforms of these functions. The Fourier transform of a Bartlett window is negative for `n` even. The Fourier transform of a triangular window, however, is always nonnegative.

The following figure, which plots the zero-phase responses of 8-point Bartlett and Triangular windows, illustrates the difference.

```
zerophase(bartlett(8))
hold on
zerophase(triang(8))
legend('Bartlett','Triangular')
axis([0.3 1 -0.2 0.5])
```
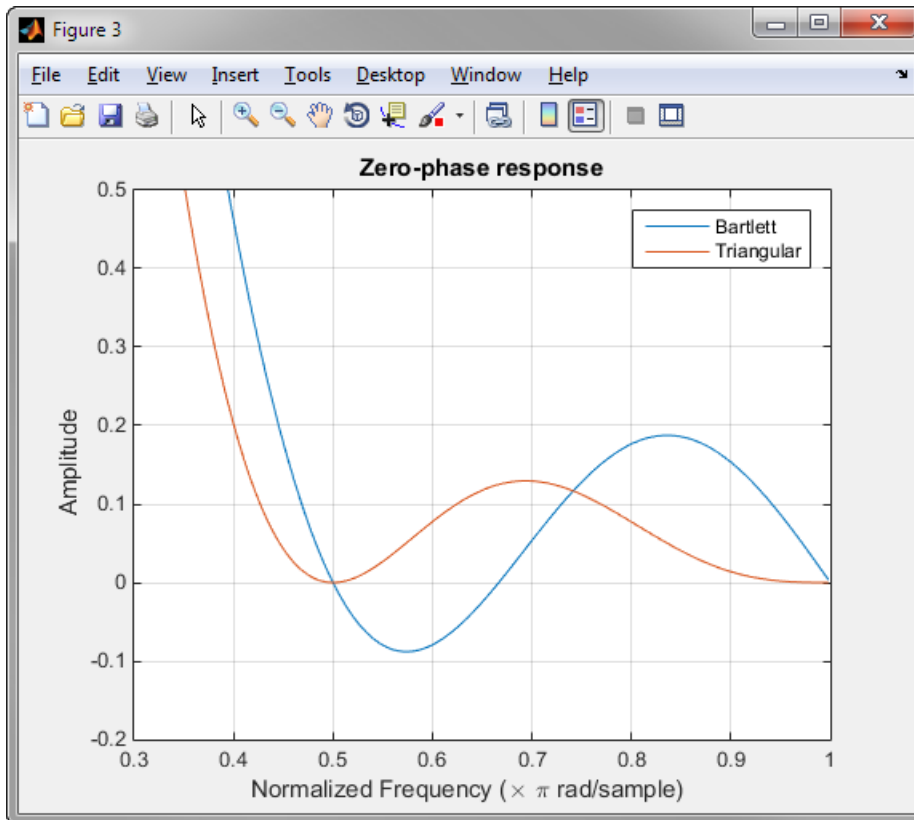

This difference can be important when choosing a window for some spectral estimation techniques, such as the Blackman-Tukey method. Blackman-Tukey forms the spectral estimate by calculating the Fourier transform of the autocorrelation sequence. The resulting estimate might be negative at some frequencies if the window's Fourier transform is negative (see 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 screen shot compares Blackman, Hamming, Hann, and Flat Top windows.
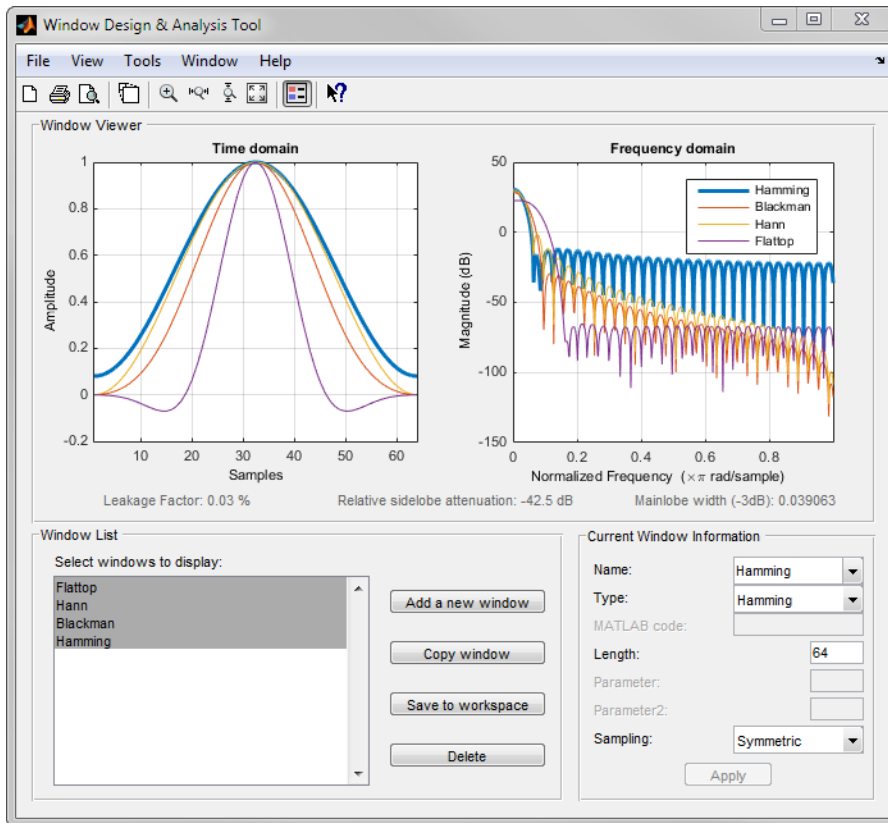
Kaiser Window

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 β parameters of 1, 4, and 9 are shown in this `wintool` example.
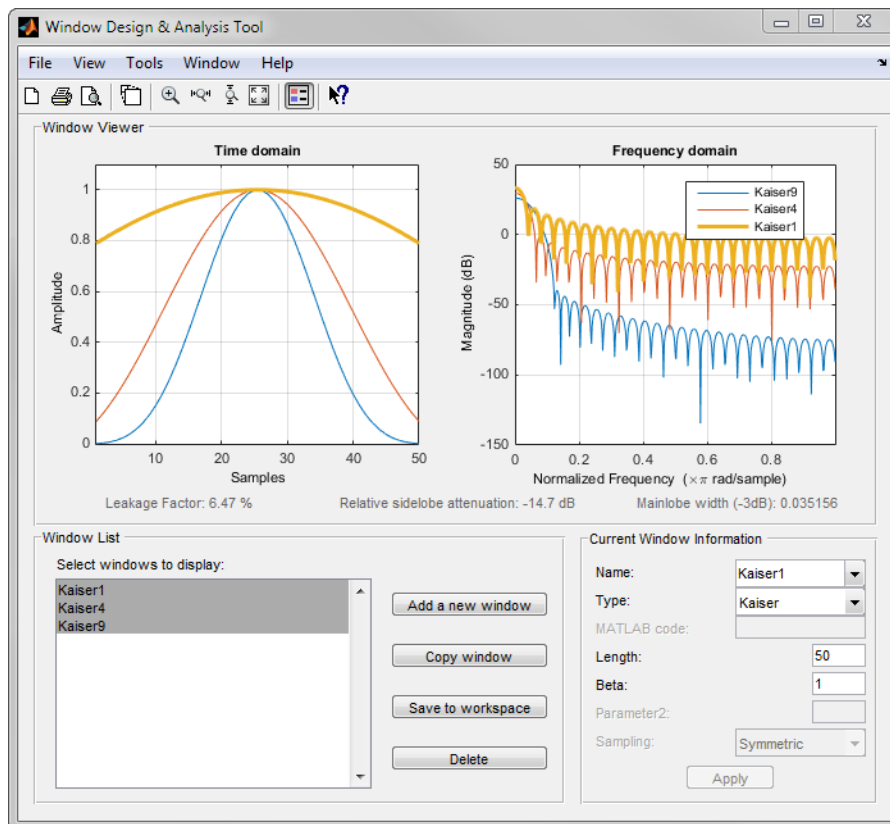
To create these Kaiser windows using the MATLAB command line, type the following:

```
n = 50;
```

```
w1 = kaiser(n,1);
w2 = kaiser(n,4);
w3 = kaiser(n,9);
[W1,f] = freqz(w1/sum(w1),1,512,2);
[W2,f] = freqz(w2/sum(w2),1,512,2);
[W3,f] = freqz(w3/sum(w3),1,512,2);
plot(f,20*log10(abs([W1 W2 W3])))
grid
legend('\beta = 1','\beta = 4','\beta = 9')
```
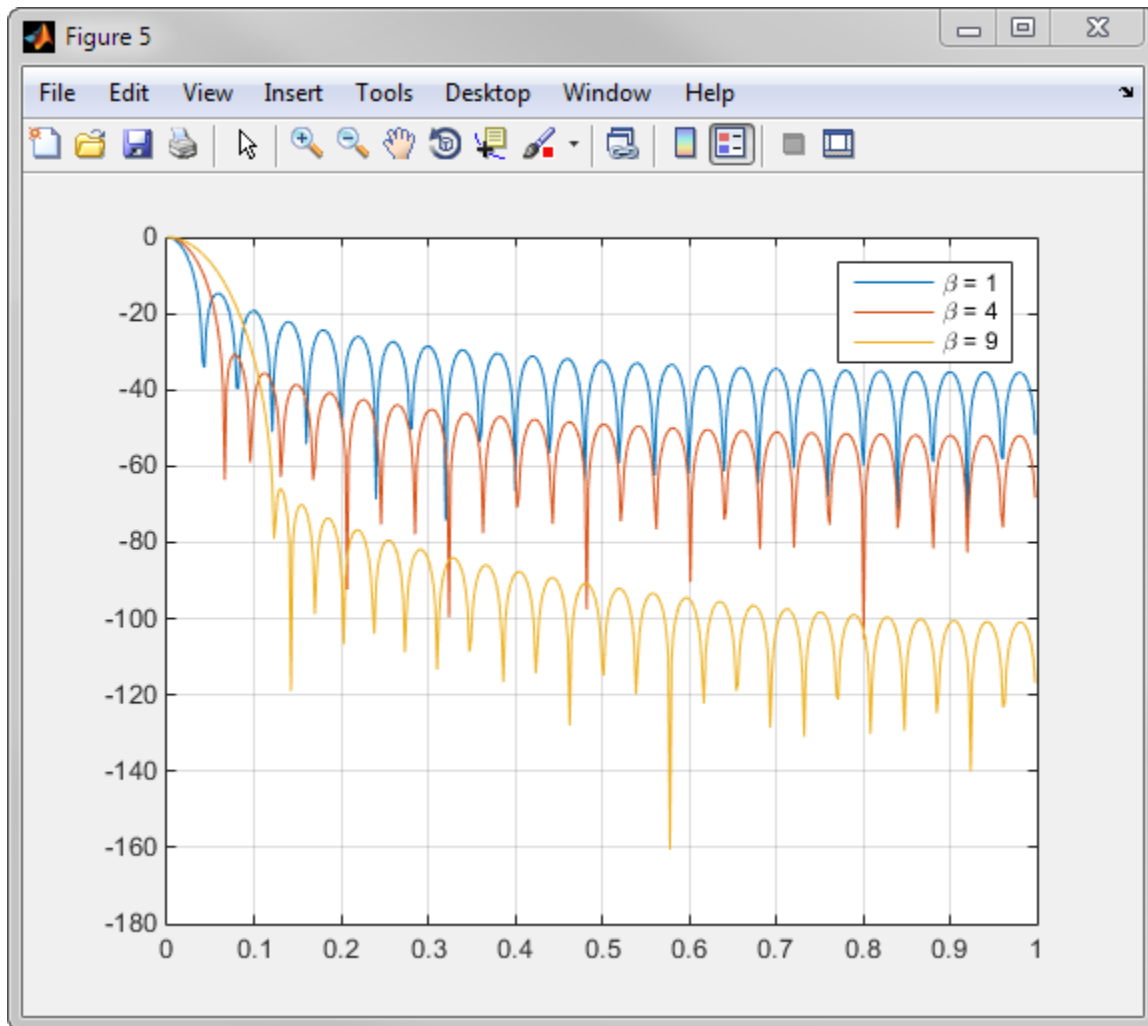
As β increases, the sidelobe height decreases and the mainlobe width increases. This WinTool screen shot shows how the sidelobe height stays the same for a fixed β parameter as the length is varied.
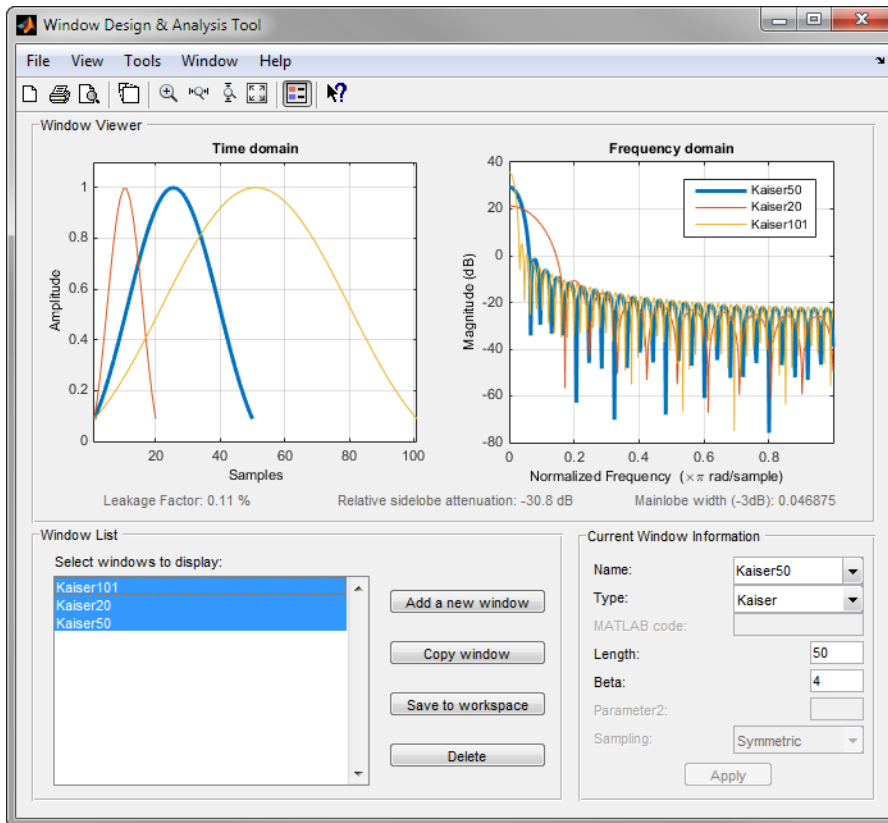
To create these Kaiser windows using the MATLAB command line, type the following:

```
w1 = kaiser(50,4);
w2 = kaiser(20,4);
w3 = kaiser(101,4);
[W1,f] = freqz(w1/sum(w1),1,512,2);
[W2,f] = freqz(w2/sum(w2),1,512,2);
[W3,f] = freqz(w3/sum(w3),1,512,2);
plot(f,20*log10(abs([W1 W2 W3])))
grid
legend('length = 50','length = 20','length = 101')
```
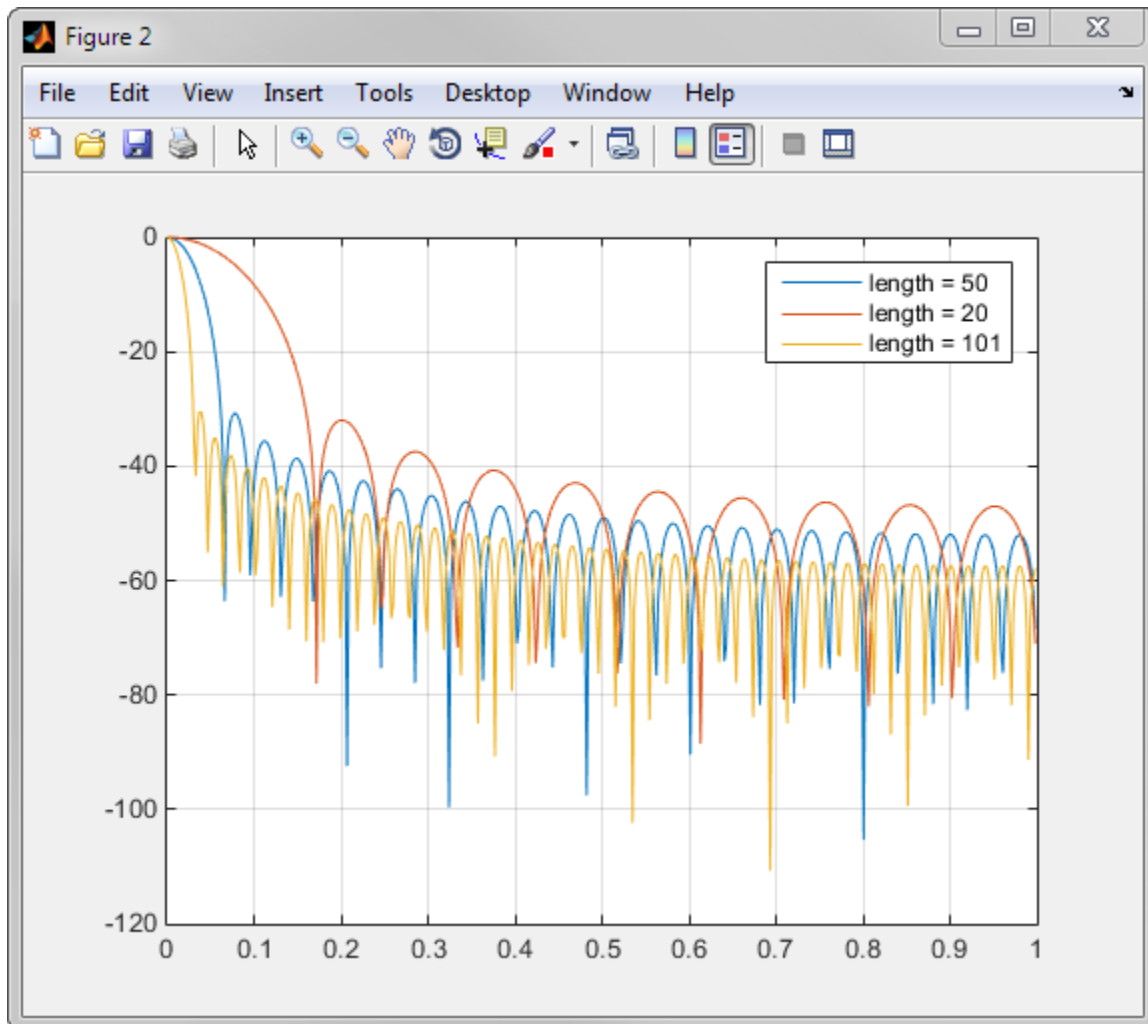
Kaiser Windows in FIR Design

There are two design formulas that can help you design FIR filters to meet a set of filter specifications using a Kaiser window. To achieve a sidelobe height of $-\alpha$ dB, the β 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/sample, use the length

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

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

```
[n,wn,beta] = kaiserord([0.4 0.6]*pi,[1 0],[0.01 0.01],2*pi);
h = fir1(n,wn,kaiser(n+1,beta),'noscale');
```

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

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

```
fvtool(h,1)
```
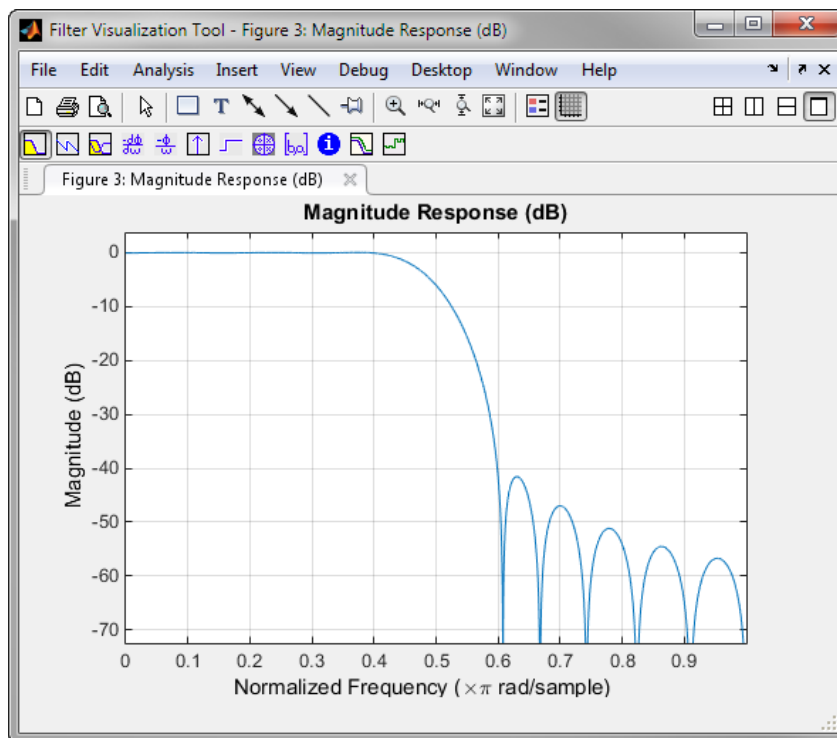
Chebyshev Window

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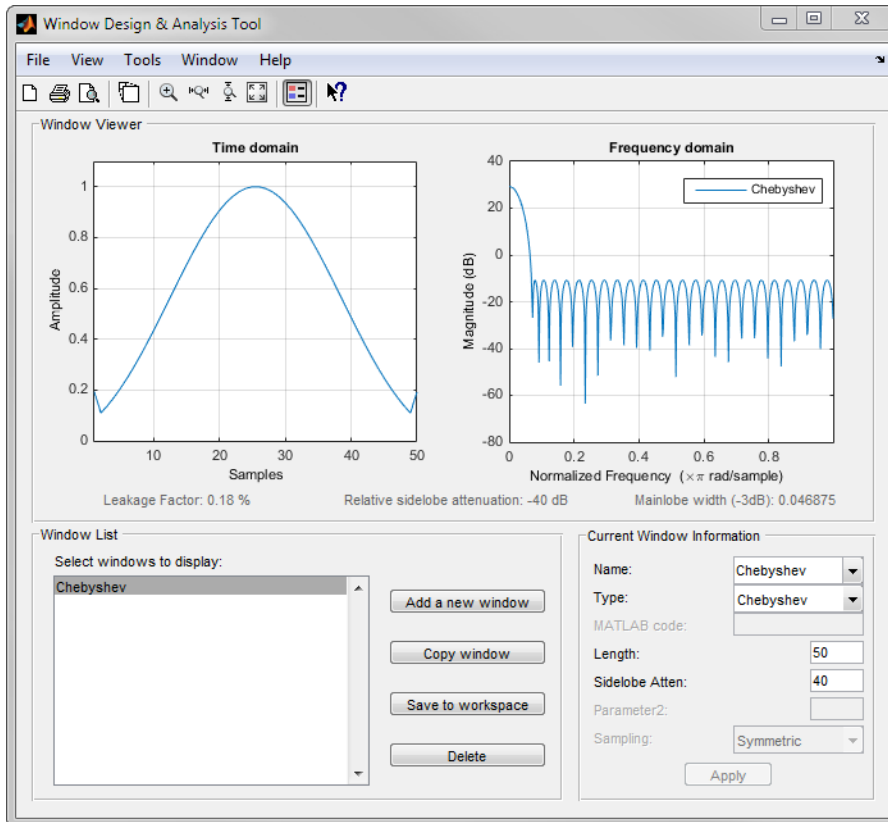
As shown in the Time Domain plot, the Chebyshev window has large spikes at its outer samples.

Parametric Modeling

In this section...
"What is Parametric Modeling" on page 7-17
"Available Parametric Modeling Functions" on page 7-17
"Time-Domain Based Modeling" on page 7-18
"Frequency-Domain Based Modeling" on page 7-21

What is Parametric Modeling

Parametric modeling techniques find the parameters for a mathematical model describing a signal, system, or process. These techniques use known information about the system to determine the model. Applications for parametric modeling include speech and music synthesis, data compression, high-resolution spectral estimation, communications, manufacturing, and simulation.

Available Parametric Modeling Functions

The toolbox parametric modeling functions operate with the rational transfer function model. Given appropriate information about an unknown system (impulse or frequency response data, or input and output sequences), these functions find the coefficients of a linear system that models the system.

One important application of the parametric modeling functions is in the design of filters that have a prescribed time or frequency response.

Here is a summary of the parametric modeling functions in this toolbox.

Domain	Functions	Description
Time	arburg	Generate all-pole filter coefficients that model an input data sequence using the Levinson-Durbin algorithm.
	arcov	Generate all-pole filter coefficients that model an input data sequence by minimizing the forward prediction error.

Domain	Functions	Description
	`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 `n`th-order all-pole model of a signal `x` is

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

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

```
x = impz(1,[1 0.1 0.1 0.1 0.1],10) + randn(10,1)/10;
```

The coefficients for a fourth-order all-pole filter that models the system are

```
a = lpc(x,4)
```

`lpc` first calls `xcorr` to find a biased estimate of the correlation function of `x`, and then uses the Levinson-Durbin recursion, implemented in the `levinson` function, to find the model coefficients `a`. The Levinson-Durbin recursion is a fast algorithm for solving a system of symmetric Toeplitz linear equations. `lpc`'s entire algorithm for `n = 4` is

```
r = xcorr(x);
r(1:length(x)-1) = [];       % Remove corr. at negative lags
a = levinson(r,4)
```

You could form the linear prediction coefficients with other assumptions by passing a different correlation estimate to `levinson`, such as the biased correlation estimate:

```
r = xcorr(x,'biased');
r(1:length(x)-1) = [];       % Remove corr. at negative lags
a = levinson(r,4)
```

Prony's Method (ARMA Modeling)

The `prony` function models a signal using a specified number of poles and zeros. Given a sequence `x` and numerator and denominator orders `n` and `m`, respectively, the statement

```
[b,a] = prony(x,n,m)
```

finds the numerator and denominator coefficients of an IIR filter whose impulse response approximates the sequence `x`.

The `prony` function implements the method described in [4] Parks and Burrus (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)
```

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

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

Notice that the first four samples match exactly. For an example of exact recovery, recover the coefficients of a Butterworth filter from its impulse response:

```
[b,a] = butter(4,.2);
h = impz(b,a,26);
[bb,aa] = prony(h,4,4);
```

Try this example; you'll see that `bb` and `aa` match the original filter coefficients to within a tolerance of 10^{-13}.

Steiglitz-McBride Method (ARMA Modeling)

The `stmcb` function determines the coefficients for the system $b(z)/a(z)$ given an approximate impulse response `x`, as well as the desired number of zeros and poles. This function identifies an unknown system based on both input and output sequences that describe the system's behavior, or just the impulse response of the system. In its default mode, `stmcb` works like `prony`.

```
[b,a] = stmcb(x,3,3)
```

`stmcb` also finds systems that match given input and output sequences:

```
y = filter(1,[1 1],x);      % Create an output signal.
[b,a] = stmcb(y,x,0,1)
```

In this example, `stmcb` correctly identifies the system used to create `y` from `x`.

The Steiglitz-McBride method is a fast iterative algorithm that solves for the numerator and denominator coefficients simultaneously in an attempt to minimize the signal error between the filter output and the given output signal. This algorithm usually converges

rapidly, but might not converge if the model order is too large. As for `prony`, `stmcb`'s resulting filter is not necessarily stable due to its exact modeling approach.

`stmcb` provides control over several important algorithmic parameters; modify these parameters if you are having trouble modeling the data. To change the number of iterations from the default of five and provide an initial estimate for the denominator coefficients:

```
n = 10;              % Number of iterations
a = lpc(x,3);        % Initial estimates for denominator
[b,a] = stmcb(x,3,3,n,a);
```

The function uses an all-pole model created with `prony` as an initial estimate when you do not provide one of your own.

To compare the functions `lpc`, `prony`, and `stmcb`, compute the signal error in each case:

```
a1 = lpc(x,3);
[b2,a2] = prony(x,3,3);
[b3,a3] = stmcb(x,3,3);
[x-impz(1,a1,10)  x-impz(b2,a2,10)  x-impz(b3,a3,10)]
```

In comparing modeling capabilities for a given order IIR model, the last result shows that for this example, `stmcb` performs best, followed by `prony`, then `lpc`. This relative performance is typical of the modeling functions.

Frequency-Domain Based Modeling

The `invfreqs` and `invfreqz` functions implement the inverse operations of `freqs` and `freqz`; they find an analog or digital transfer function of a specified order that matches a given complex frequency response. Though the following examples demonstrate `invfreqz`, the discussion also applies to `invfreqs`.

To recover the original filter coefficients from the frequency response of a simple digital filter:

```
[b,a] = butter(4,0.4)           % Design Butterworth lowpass
[h,w] = freqz(b,a,64);          % Compute frequency response
[b4,a4] = invfreqz(h,w,4,4)     % Model: n = 4, m = 4
```

The vector of frequencies `w` has the units in rad/sample, and the frequencies need not be equally spaced. `invfreqz` finds a filter of any order to fit the frequency data; a third-order example is

```
[b4,a4] = invfreqz(h,w,3,3)    % Find third-order IIR
```

Both `invfreqs` and `invfreqz` design filters with real coefficients; for a data point at positive frequency `f`, the functions fit the frequency response at both `f` and `-f`.

By default `invfreqz` uses an equation error method to identify the best model from the data. This finds b and a in

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

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
```

The resulting filter is always stable.

Graphically compare the results of the first and second algorithms to the original Butterworth filter with FVTool (and select the Magnitude and Phase Responses):

```
fvtool(b,a,b4,a4,b30,a30)
```
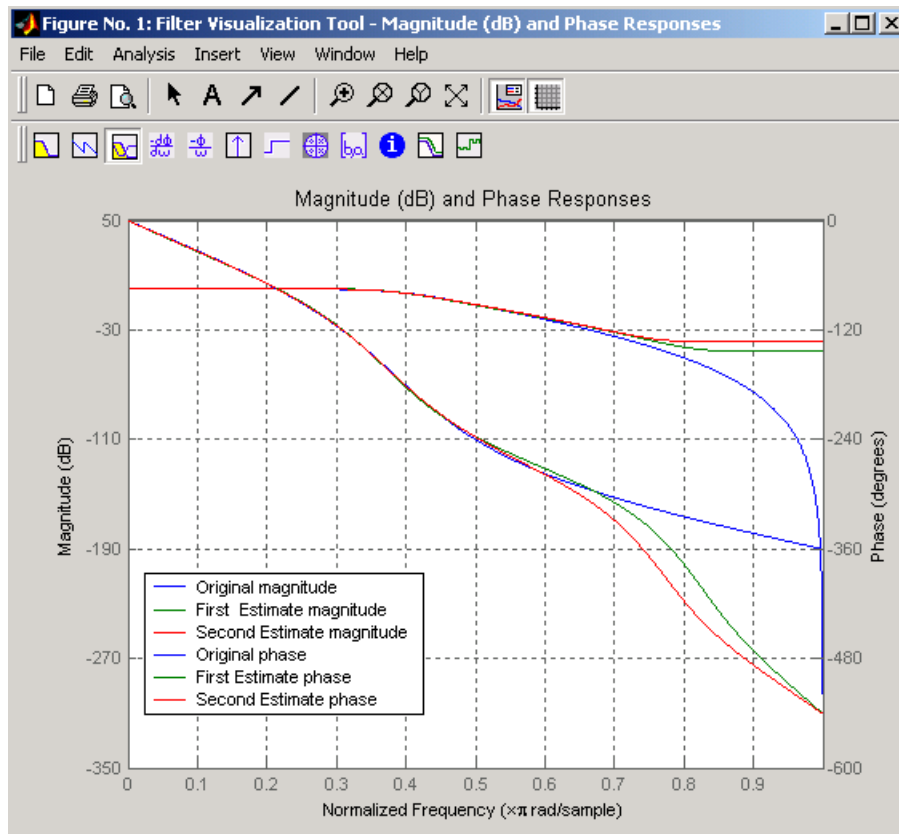
To verify the superiority of the fit numerically, type

```
sum(abs(h-freqz(b4,a4,w)).^2)    % Total error, algorithm 1
sum(abs(h-freqz(b30,a30,w)).^2)  % Total error, algorithm 2
```

Resampling

In this section...
"Available Resampling Functions" on page 7-24
"resample Function" on page 7-24
"decimate and interp Functions" on page 7-25
"upfirdn Function" on page 7-26
"spline Function" on page 7-26

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

What Is a Cepstrum?

Cepstrum analysis is a nonlinear signal processing technique with a variety of applications in areas such as speech and image processing.

The *complex cepstrum* of a sequence *x* is calculated by finding the complex natural logarithm of the Fourier transform of *x*, then the inverse Fourier transform of the resulting sequence:

$$\hat{x} = \frac{1}{2\pi} \int_{-\pi}^{\pi} \log[X(e^{j\omega})] e^{j\omega n} \, d\omega.$$

The toolbox function `cceps` performs this operation, estimating the complex cepstrum for an input sequence. It returns a real sequence the same size as the input sequence.

Try using cceps in an echo detection application. First, create a 45 Hz sine wave sampled at 100 Hz. Add an echo of the signal, with half the amplitude, 0.2 seconds after the beginning of the signal.

```
t = 0:0.01:1.27;
s1 = sin(2*pi*45*t);
s2 = s1 + 0.5*[zeros(1,20) s1(1:108)];
```

Compute and plot the complex cepstrum of the new signal.

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

The complex cepstrum shows a peak at 0.2 seconds, indicating the echo.

The *real cepstrum* of a signal *x*, sometimes called simply the cepstrum, is calculated by determining the natural logarithm of magnitude of the Fourier transform of *x*, then obtaining the inverse Fourier transform of the resulting sequence:

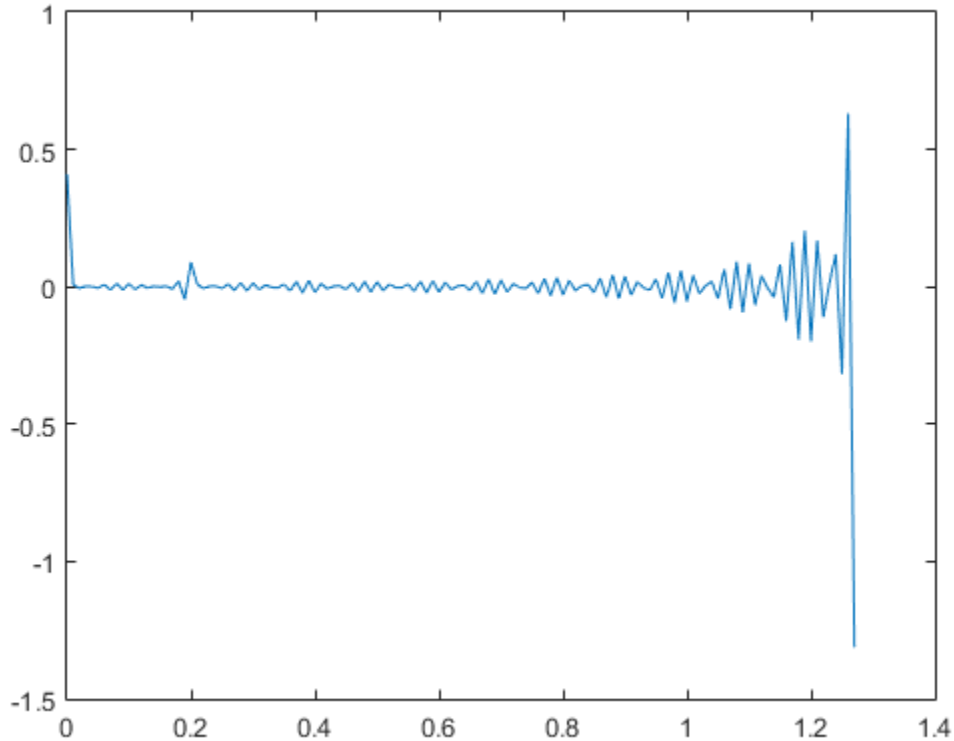$$c_x = \frac{1}{2\pi} \int_{-\pi}^{\pi} \log \left| X(e^{j\omega}) \right| e^{j\omega n} \, d\omega.$$

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

The `rceps` function also returns a unique minimum-phase sequence that has the same real cepstrum as the input. To obtain both the real cepstrum and the minimum-phase reconstruction for a sequence, use `[y,ym] = rceps(x)`, where `y` is the real cepstrum and `ym` is the minimum phase reconstruction of `x`. The following example shows that one output of `rceps` is a unique minimum-phase sequence with the same real cepstrum as `x`.

```
y = [4 1 5];                      % Non-minimum phase sequence
[xhat,yhat] = rceps(y);
xhat2= rceps(yhat);
[xhat' xhat2']


ans =

    1.6225    1.6225
    0.3400    0.3400
    0.3400    0.3400
```

Inverse Complex Cepstrum

To invert the complex cepstrum, use the `icceps` function. Inversion is complicated by the fact that the `cceps` function performs a data-dependent phase modification so that the unwrapped phase of its input is continuous at zero frequency. The phase modification is equivalent to an integer delay. This delay term is returned by `cceps` if you ask for a second output:

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


xhat =

  Columns 1 through 7

    2.2428   -0.0420   -0.0210    0.0045    0.0366    0.0788    0.1386

  Columns 8 through 10

    0.2327    0.4114    0.9249


delay =
```

```
     1
```

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

```
icc = icceps(xhat,2)

icc =

  Columns 1 through 7

    2.0000    3.0000    4.0000    5.0000    6.0000    7.0000    8.0000

  Columns 8 through 10

    9.0000   10.0000    1.0000
```

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

FFT-Based Time-Frequency Analysis

The Signal Processing Toolbox™ product provides a function, `spectrogram`, that returns the time-dependent Fourier transform for a sequence, or displays this information as a spectrogram. The *time-dependent Fourier transform* is the discrete-time Fourier transform for a sequence, computed using a sliding window. This form of the Fourier transform, also known as the short-time Fourier transform (STFT), has numerous applications in speech, sonar, and radar processing. The *spectrogram* of a sequence is the magnitude of the time-dependent Fourier transform versus time.

To display the spectrogram of a linear FM signal:

```
fs = 10000;
t = 0:1/fs:2;
x = vco(sawtooth(2*pi*t,0.75),[0.1 0.4]*fs,fs);
spectrogram(x,kaiser(256,5),220,512,fs,'yaxis')
```
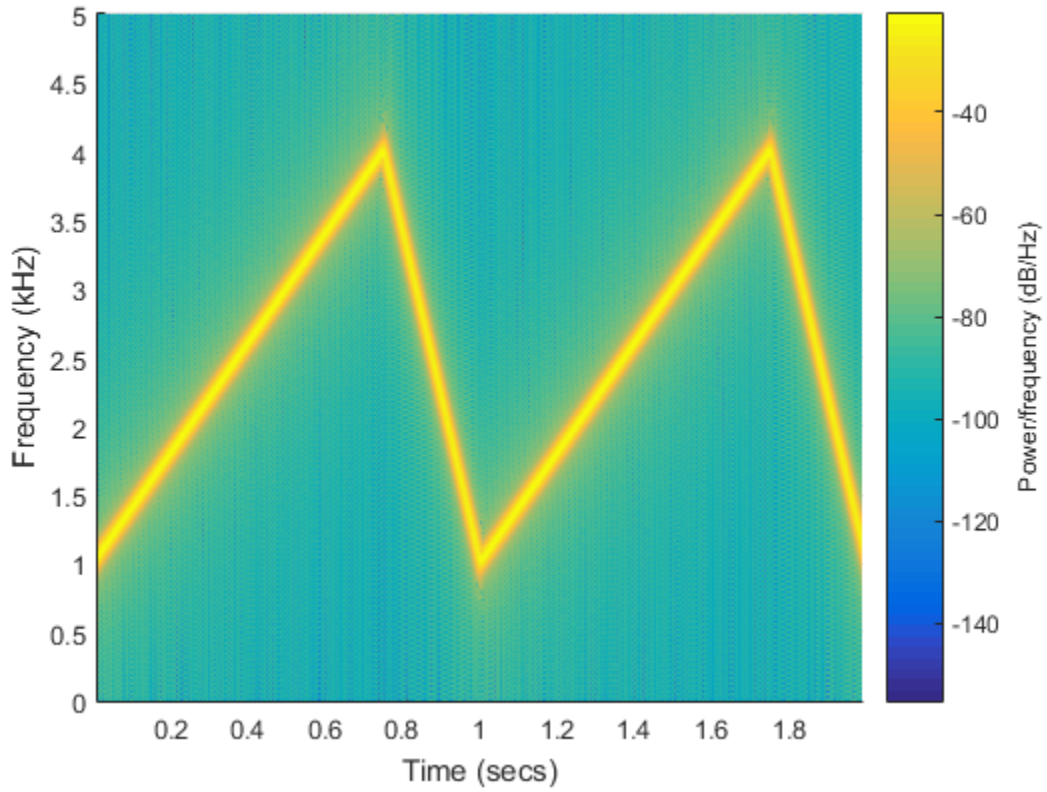
Median Filtering

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™ for information on two-dimensional median filtering.

Communications Applications

In this section...
"Modulation" on page 7-34
"Demodulation" on page 7-35
"Voltage Controlled Oscillator" on page 7-37

Modulation

Modulation varies the amplitude, phase, or frequency of a *carrier signal* with reference to a *message signal*. The `modulate` function modulates a message signal with a specified modulation method.

The basic syntax for the `modulate` function is

```
y = modulate(x,fc,fs,'method',opt)
```

where:

- `x` is the message signal.
- `fc` is the carrier frequency.
- `fs` is the sampling frequency.
- `method` is a flag for the desired modulation method.
- `opt` is any additional argument that the method requires. (Not all modulation methods require an option argument.)

The table below summarizes the modulation methods provided; see the documentation for `modulate`, `demod`, and `vco` for complete details on each.

Method	Description
`amdsb-sc` or `am`	Amplitude modulation, double sideband, suppressed carrier
`amdsb-tc`	Amplitude modulation, double sideband, transmitted carrier
`amssb`	Amplitude modulation, single sideband
`fm`	Frequency modulation
`pm`	Phase modulation
`ppm`	Pulse position modulation

Method	Description
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');
```
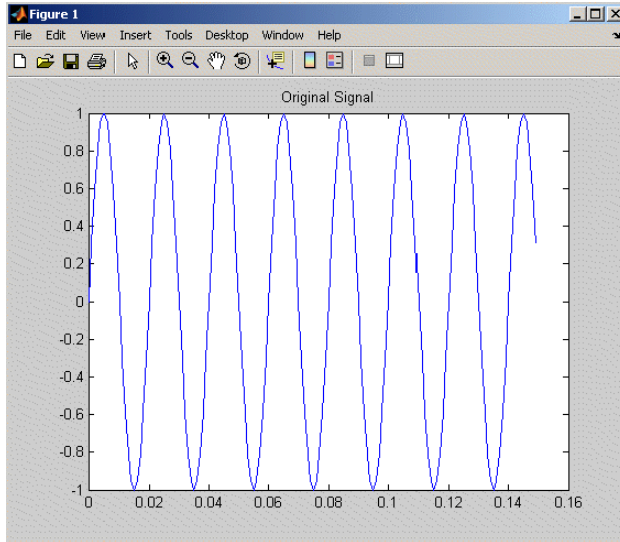

Original Signal

Modulated Signal
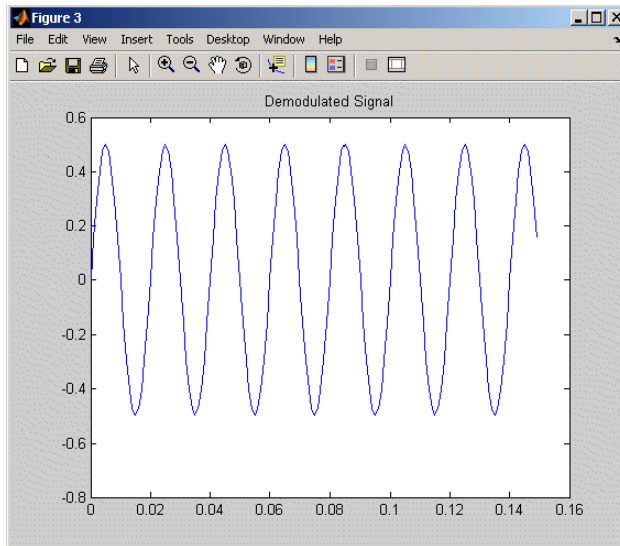
Demodulated Signal

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

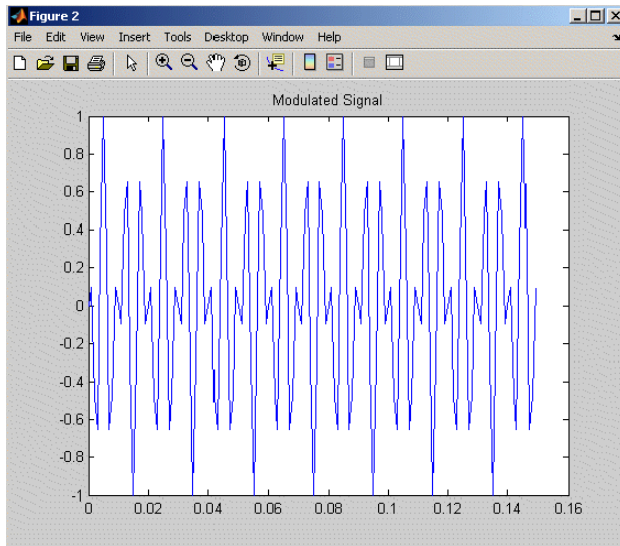The voltage controlled oscillator function `vco` creates a signal that oscillates at a frequency determined by the input vector. The basic syntax for `vco` is

```
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 7-31 for an example using the `vco` function.

Deconvolution

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

The syntax for `deconv` is

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

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

To try `deconv`, first convolve two simple vectors `a` and `b` .

```
a = [1 2 3];
b = [4 5 6];
c = conv(a,b)
c =
    4    13    28    27    18
```

Now use `deconv` to deconvolve `b` from `c`:

```
[q,r] = deconv(c,a)
q =
    4    5    6
r =
    0    0    0    0    0
```

Chirp Z-Transform

The chirp Z-transform (CZT), useful in evaluating the Z-transform along contours other than the unit circle. The chirp Z-transform is also more efficient than the DFT algorithm for the computation of prime-length transforms, and it is useful in computing a subset of the DFT for a sequence. The chirp Z-transform, or CZT, computes the Z-transform along spiral contours in the z-plane for an input sequence. Unlike the DFT, the CZT is not constrained to operate along the unit circle, but can evaluate the Z-transform along contours described by $z_\ell = AW^{-\ell}$, $\ell = 0, ..., M-1$, 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.')
```
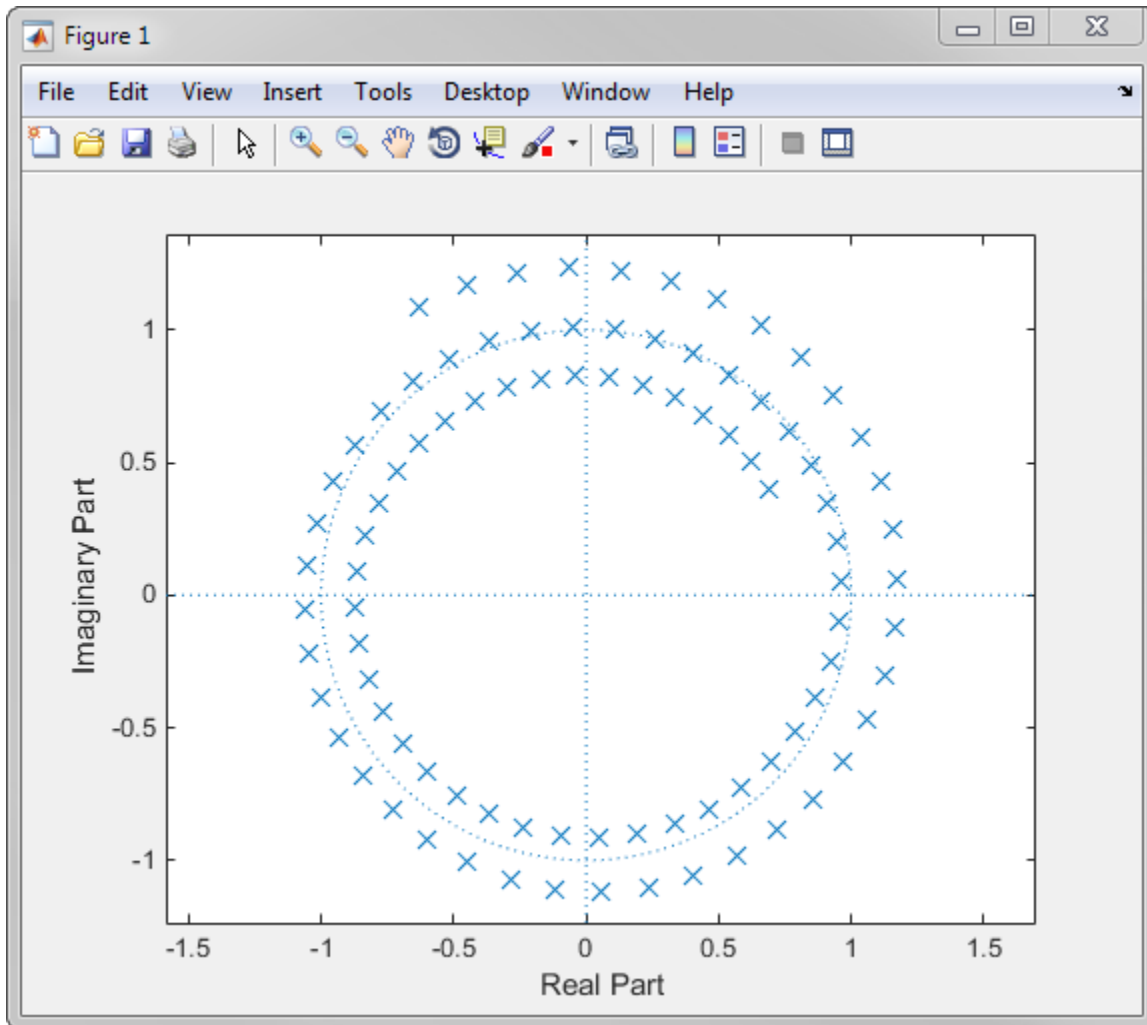
`czt(x,M,W,A)` computes the Z-transform of `x` on these points.

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

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

where

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

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

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

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

where

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

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

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

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

```
y = dct(x)                    % Compute DCT
y2 = find(abs(y) < 0.9);      % Use 17 coefficients
y(y2) = zeros(size(y2));      % Zero out points < 0.9
z = idct(y);                  % Reconstruct signal 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')
```
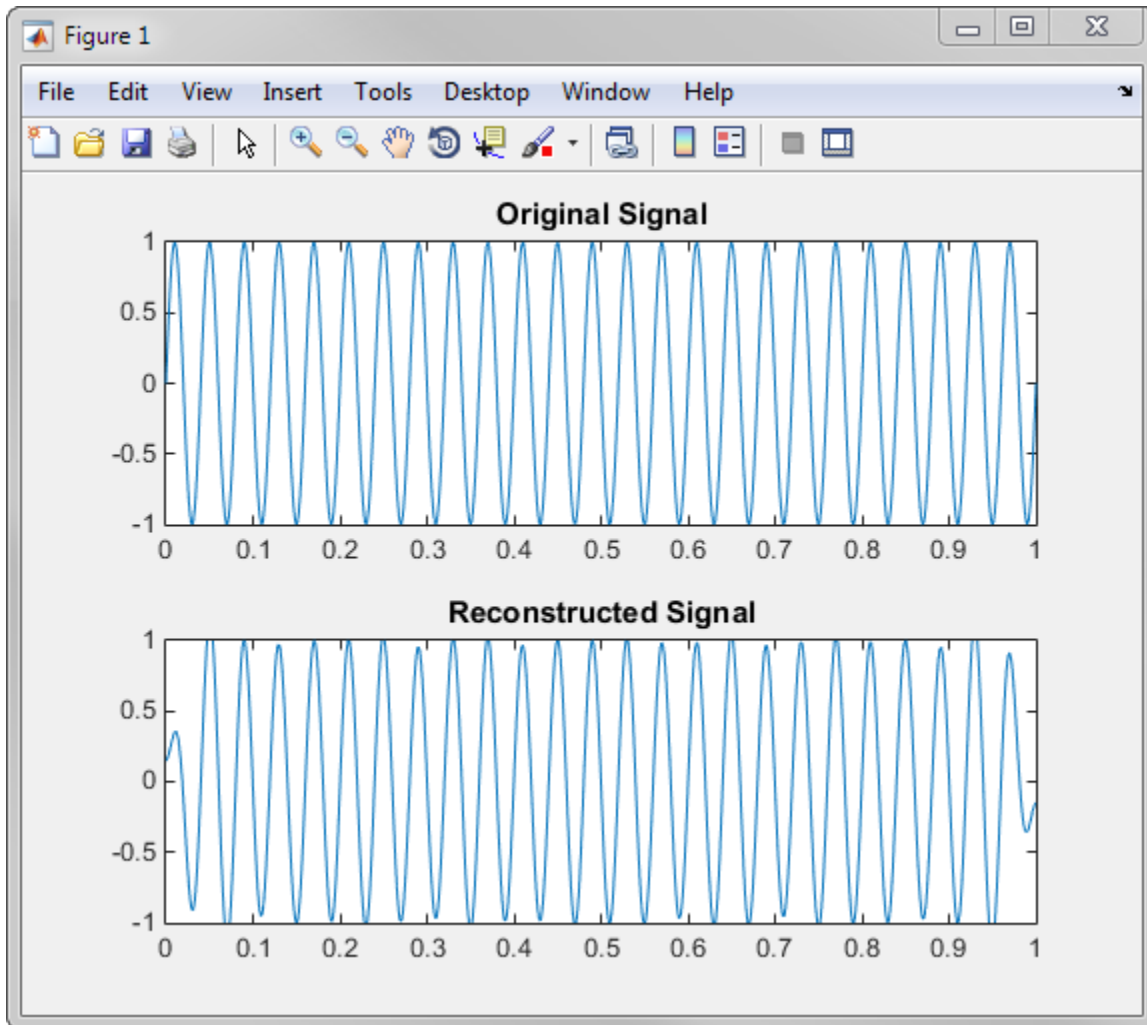
One measure of the accuracy of the reconstruction is

```
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.
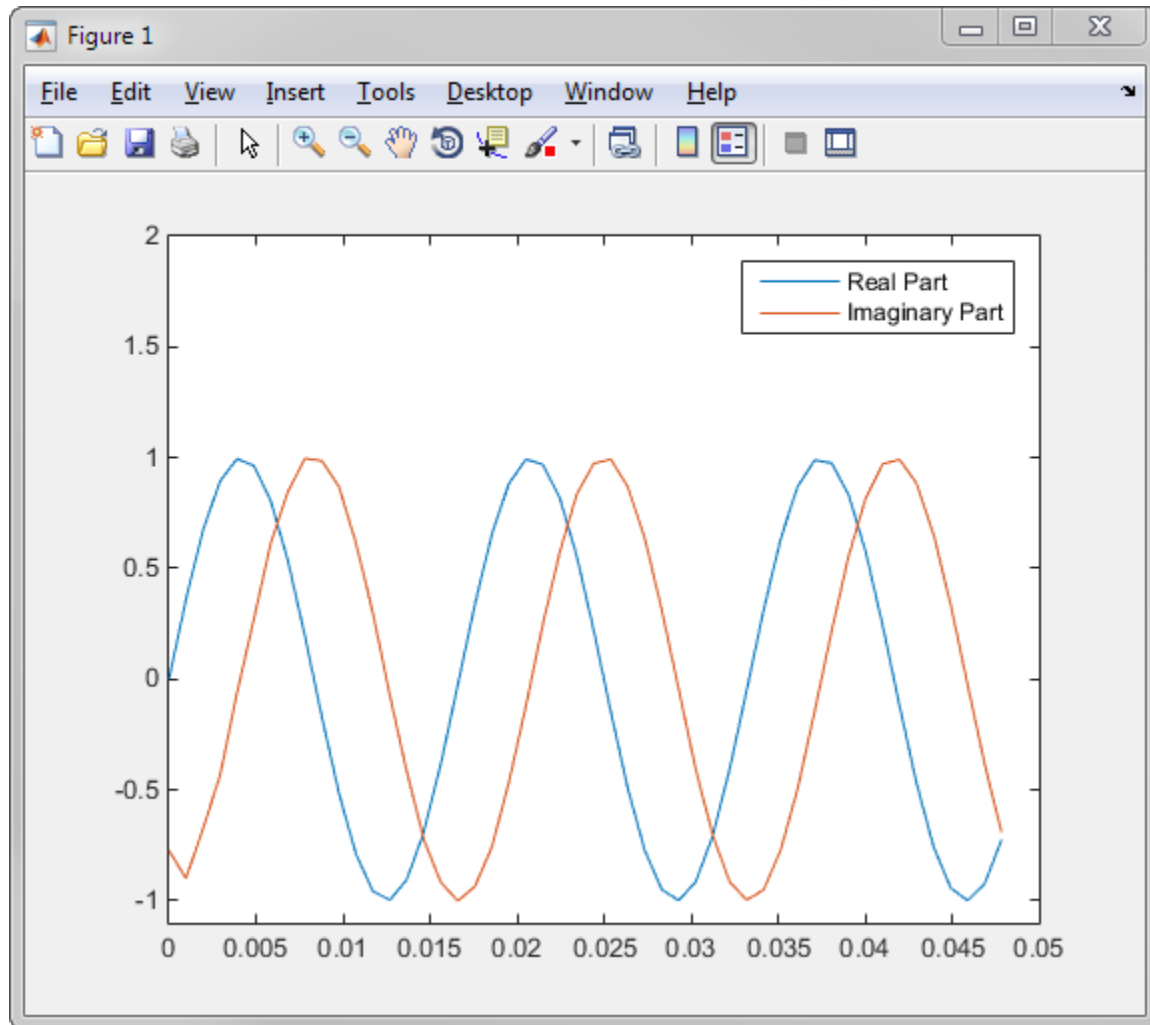
Hilbert Transform

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), use

```
t = 0:1/1024:1;
x = sin(2*pi*60*t);
y = hilbert(x);

plot(t(1:50),real(y(1:50)))
hold on
plot(t(1:50),imag(y(1:50)))
axis([0 0.05 -1.1 2])
legend('Real Part','Imaginary Part')
```

The analytic signal is useful in calculating *instantaneous attributes* of a time series, the attributes of the series at any point in time. The procedure requires that the signal be monocomponent. See "Hilbert Transform and Instantaneous Frequency" for examples.

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:

$$y_n = \frac{1}{N} \sum_{i=0}^{N-1} x_i \, \text{WAL}(n,i),$$

$$x_i = \sum_{i=0}^{N-1} y_n \, \text{WAL}(n,i),$$

where $i = 0,1, \ldots, N-1$ and $\text{WAL}(n,i)$ are Walsh functions. Similar to the Cooley-Tukey algorithm for the FFT, the N elements are decomposed into two sets of $N/2$ elements, which are then combined using a butterfly structure to form the FWHT. For images, where the input is typically a 2-D signal, the FWHT coefficients are calculated by first evaluating across the rows and then evaluating down the columns.

For the following simple signal, the resulting FWHT shows that x was created using Walsh functions with sequency values of 0, 1, 3, and 6, which are the nonzero indices of the transformed x. The inverse FWHT recreates the original signal.

```
x = [4 2 2 0 0 2 -2 0]
y = fwht(x)

x =

    4    2    2    0    0    2   -2    0

y =

    1    1    0    1    0    0    1    0

x1 = ifwht(y)


x1 =

    4    2    2    0    0    2   -2    0
```

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. They also provide rapid signal reconstruction.

Start with an ECG signal. Replicate it to create a longer signal and insert some additional random noise.

```
xe = ecg(512);
xr = repmat(xe,1,8);
x = xr + 0.1.*randn(1,length(xr));
```

Transform the signal using the fast Walsh-Hadamard transform. Plot the original signal and the transformed signal.

```
y = fwht(x);

subplot(2,1,1)
plot(x)
xlabel('Sample index')
ylabel('Amplitude')
title('ECG Signal')

subplot(2,1,2)
plot(abs(y))
xlabel('Sequency index')
ylabel('Magnitude')
title('WHT Coefficients')
```
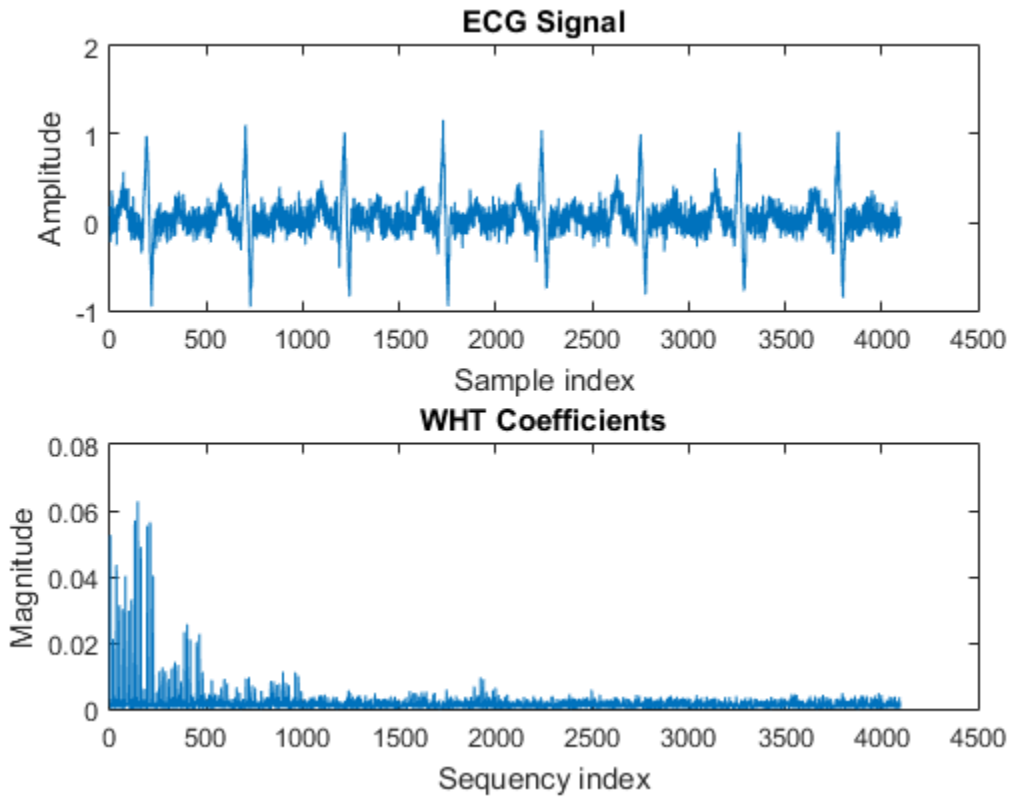
The plot shows that most of the signal energy is in the lower sequency values below approximately 1100. Store only the first 1024 coefficients (out of 4096) and see if the signal can be accurately reconstructed from only these stored coefficients.

```
y(1025:length(x)) = 0;
xHat = ifwht(y);

figure
plot(x)
hold on
plot(xHat)
xlabel('Sample index')
ylabel('ECG signal amplitude')
legend('Original','Reconstructed')
```
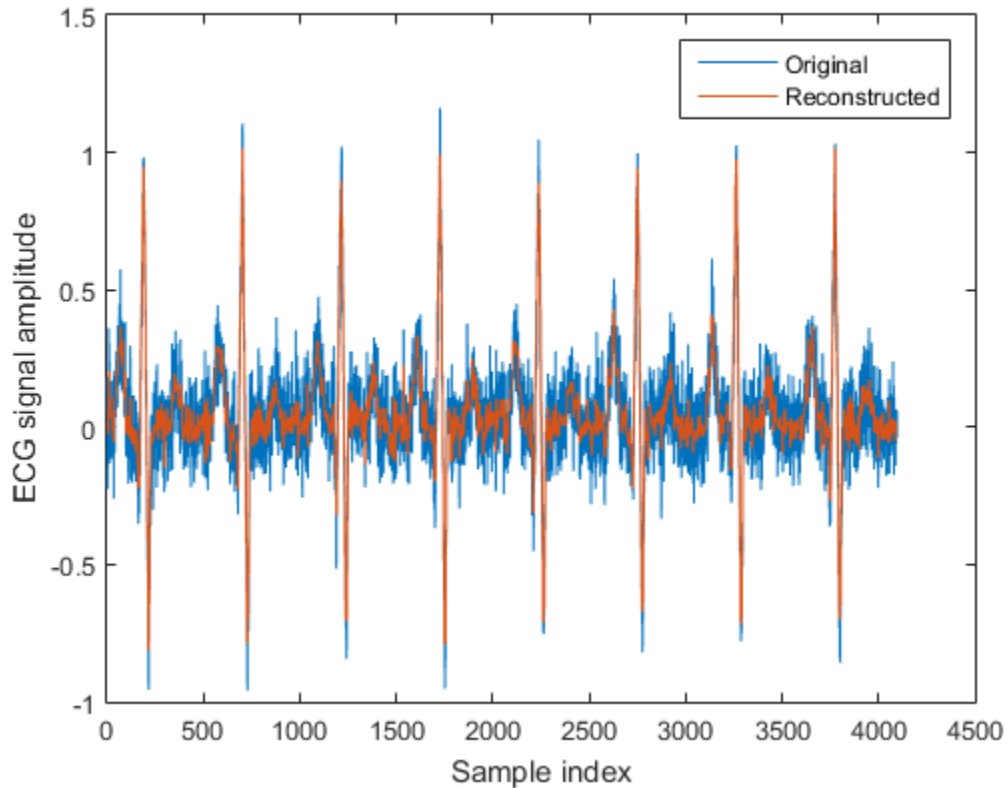
The reproduced signal is very close to the original but has been compressed to a quarter of the size. Storing more coefficients is a tradeoff between increased resolution and increased noise, while storing fewer coefficients may cause loss of peaks.

Selected Bibliography

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

[2] Oppenheim, Alan V., and Ronald W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[3] Oppenheim, Alan V., and Ronald W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975.

[4] Parks, Thomas W., and C. Sidney Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987.

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

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 8-2

SPTool Overview

SPTool is an interactive GUI for digital signal processing used to

- Analyze signals
- Design filters
- Analyze (view) filters
- Filter signals
- Analyze signal spectra

You can accomplish these tasks using four GUIs that you access from within SPTool:

- The "Signal Browser" on page 8-7 is for analyzing signals. You can also play signals using your computer's audio hardware.
- `fdatool` 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 8-47 to graphically place poles and zeros on the *z*-plane.
- The "Filter Visualization Tool" on page 8-11 (FVTool) is for analyzing filter characteristics.
- The "Spectrum Viewer" on page 8-13 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 8-31. 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 8-27.

Opening SPTool

To open SPTool, type

```
sptool
```
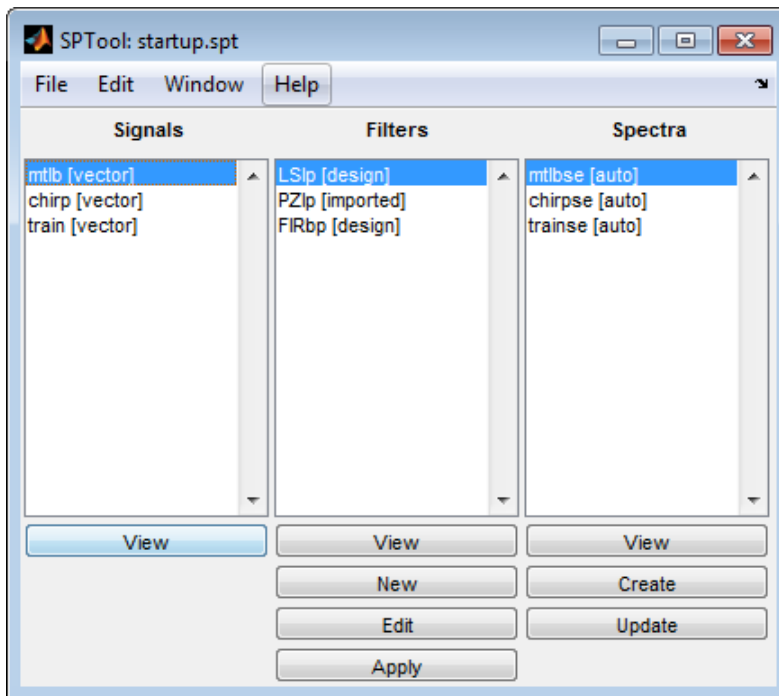

When you first open SPTool, it contains a collection of default signals, filters, and spectra. To specify your own preferences for what signals, filters, and spectra to see when SPTool opens see "Setting Preferences" on page 8-42.

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

- Signal Browser
- Filter Visualization Tool
- Spectrum Viewer

You can access FDATool by clicking **New** to create a new filter or `Edit` to edit a selected filter. Clicking **Apply** applies a selected filter to a selected signal.

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

Getting Context-Sensitive Help

To find information on a particular feature or setting of the "Signal Browser" on page 8-7:

- In any Measurements panel, right-click anywhere on the panel and select **What's this?**.

- In any dialog box where you see the icon in the lower left corner, right-click on any parameter and select **What's this?**.

To find information on a particular region of "FDATool" on page 8-10 or "Spectrum Viewer" on page 8-13:

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.
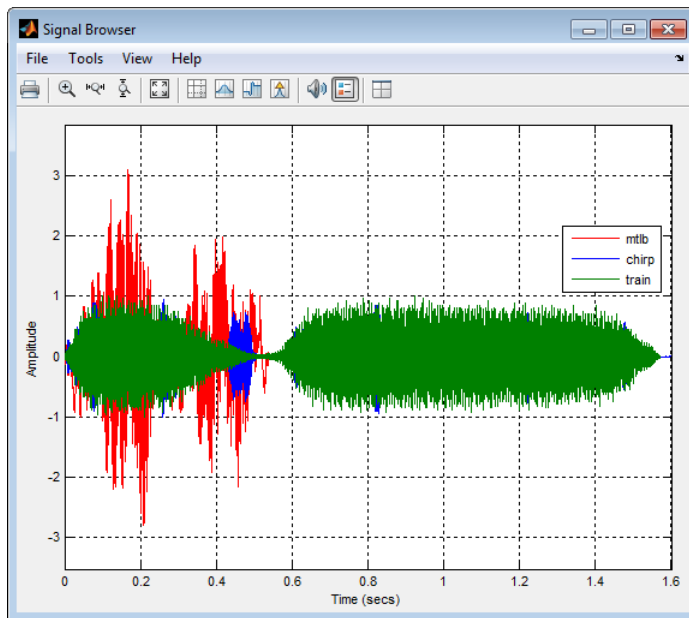
The Signal Browser has the following components:

- A display region for analyzing signals
- A panels section on the right side of the scope window, which shows statistics and information about your signals
- A toolbar with buttons for convenient access to frequently used functions

Icon	Description
	Print the current scope window.
	Play an audio signal. The function soundsc is used to play the signal.
	Show multiple displays of signals.
	Zoom the signal in and out.
	Scale the axes.

Icon	Description
	Toggle the legends on and off.
	Toggle the Cursor Measurements panel. This panel allows you to see screen cursors and get measurements of time and amplitude values at the cursors.
	Toggle the Signal Statistics, Bilevel Measurements, and Peak Finder panels, which display various measurements about the selected signal.

For more information on the Signal Browser, see the `sptool` function reference page.

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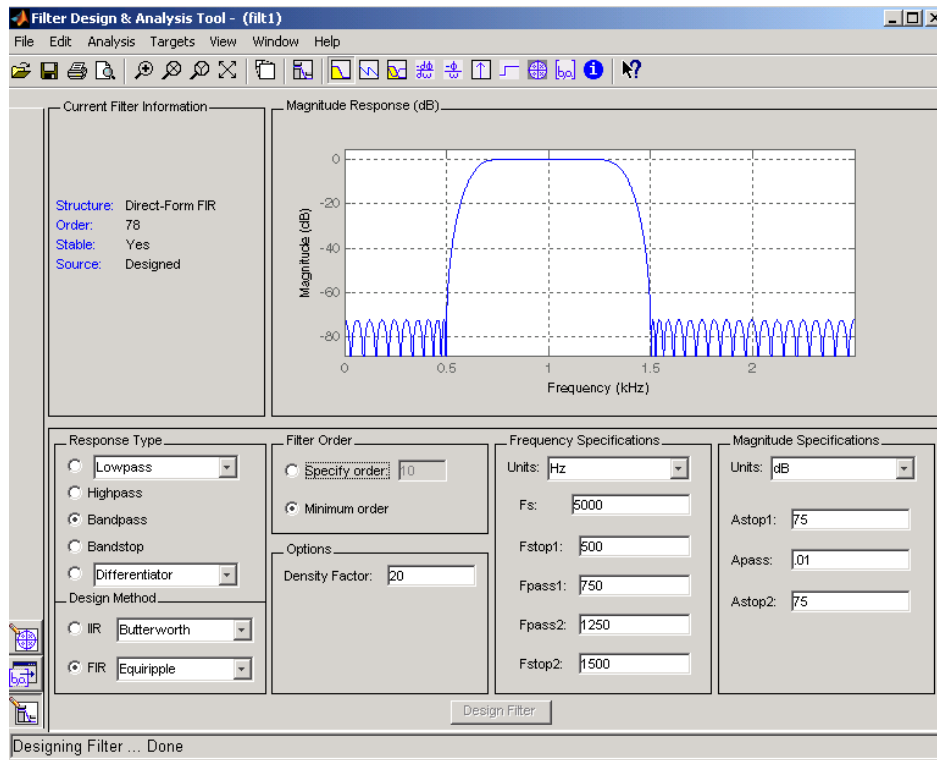
Note When you open FDATool from SPTool, a reduced version of FDATool that is compatible with SPTool opens.

Filter Visualization Tool

In this section...
"Connection between FVTool and SPTool" on page 8-11
"Opening the Filter Visualization Tool" on page 8-11
"Analysis Parameters" on page 8-12

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.
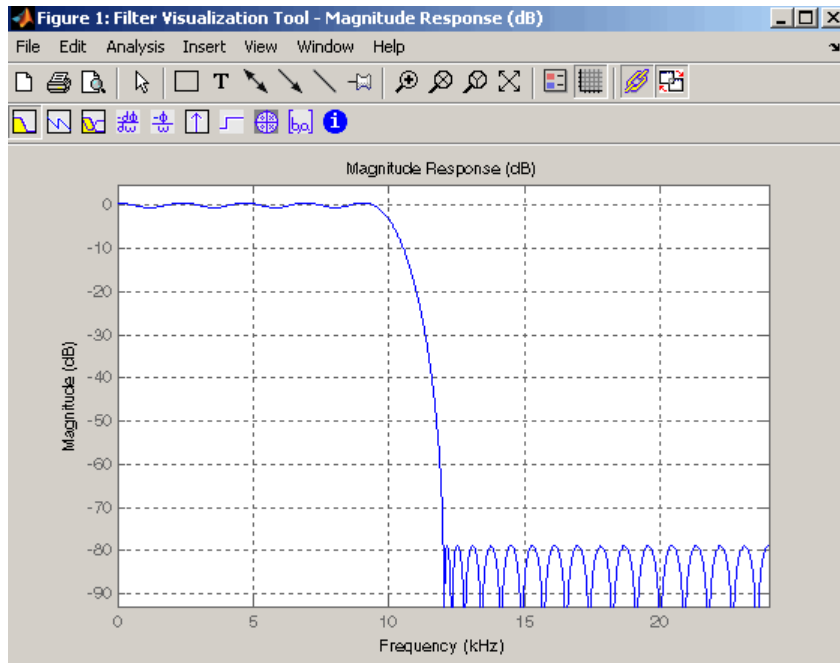
Analysis Parameters

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

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.
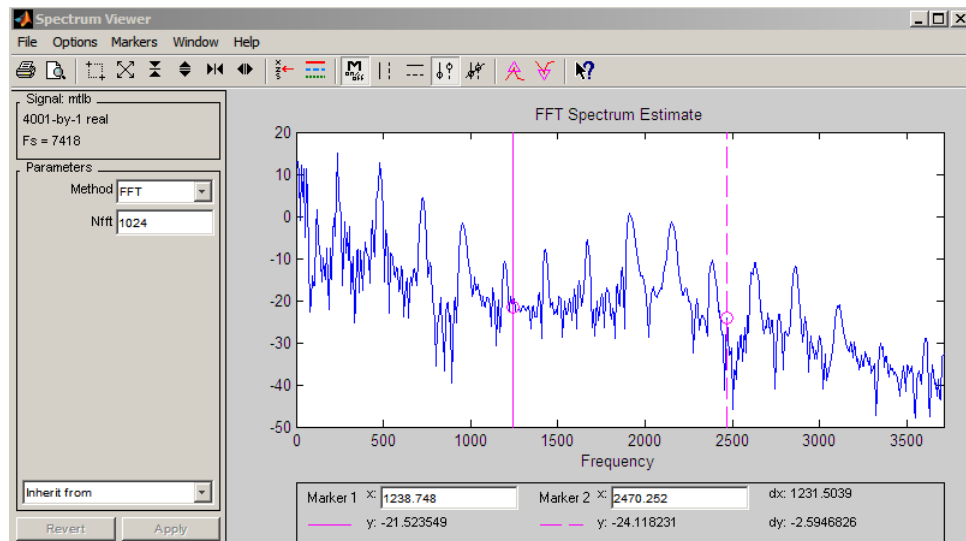
The Spectrum Viewer has the following components:

- 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
M	Toggle the markers on and off
｜¦ ── ↓° ↮ A V	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 8-16

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

```
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.
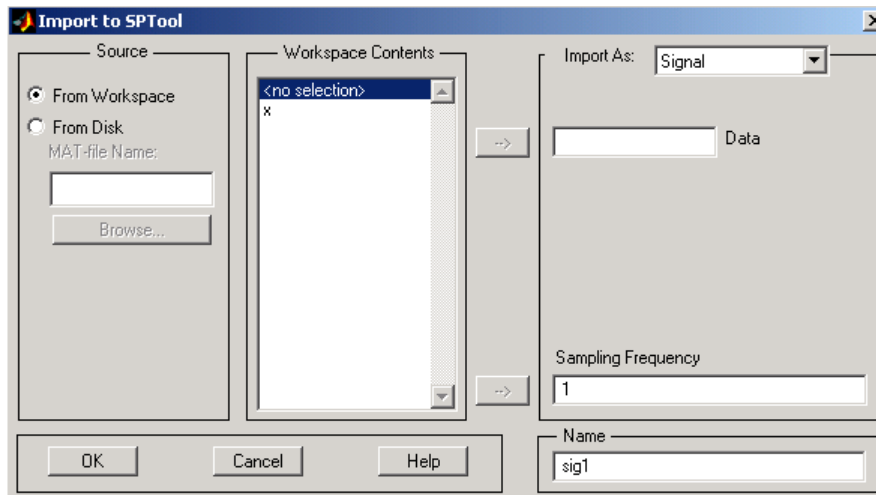
The variable x is displayed in the **Workspace Contents** list. (If it is not, select the **From Workspace** radio button to display the contents of the workspace.)

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 8-31 for specific details.

You can also import signals from MAT-files on your disk, rather than from the workspace. See "Loading Variables from the Disk" on page 8-35 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.
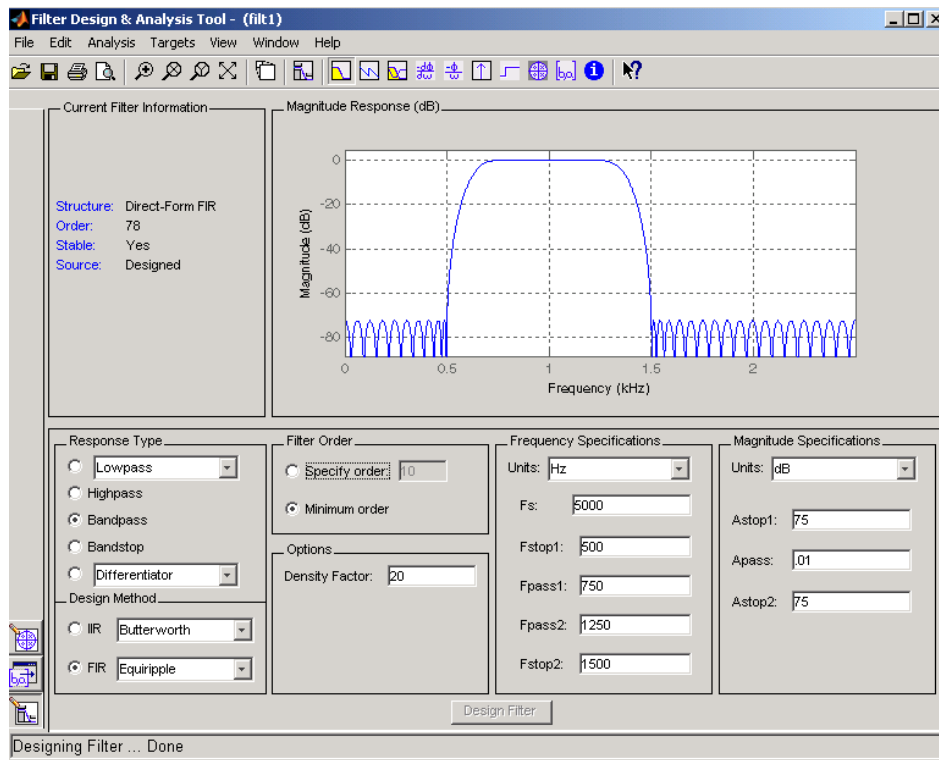
The resulting filter is an order-78 bandpass equiripple filter.

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.
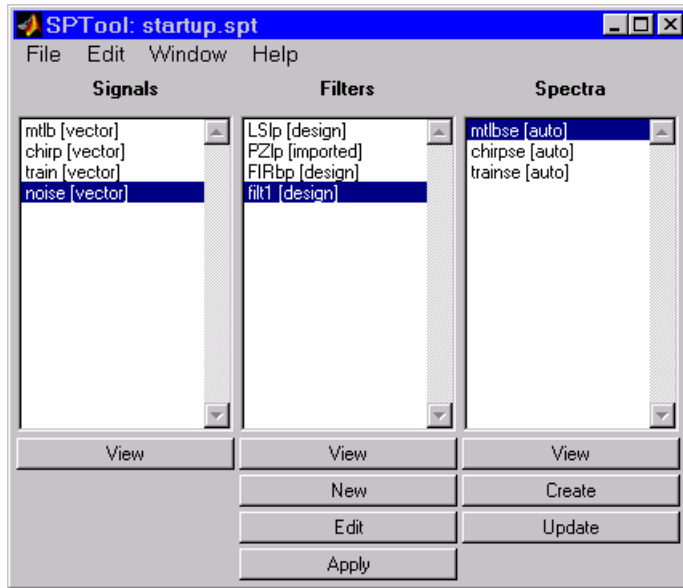
2 Click **Apply** under the **Filters** list.
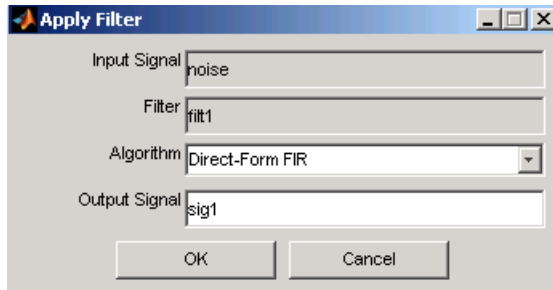
3 Leave the **Algorithm** as `Direct-Form FIR`.

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 on the toolbar to select the `blnoise` signal.

The display area is updated. Now you can see the `blnoise` signal superimposed on top of the `noise` signal. The signals are displayed in different colors in both the display region and the panner. You can change the color of the selected signal using the *Line Properties* button on the toolbar, .
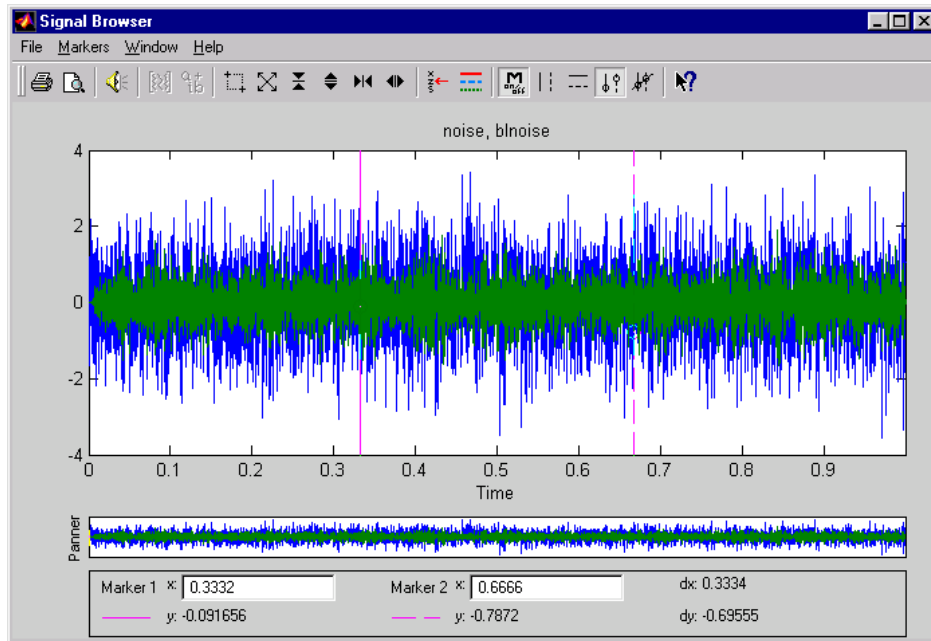
Playing a Signal

When you click **Play** in the Signal Browser toolbar, , the active signal is played on the computer's audio hardware:

1. To hear a portion of the active (selected) signal

    a. Use the vertical markers to select a portion of the signal you want to play.

        Vertical markers are enabled by the and buttons.

    b. Click **Play**.

2. To hear the other signal

    a. Select the signal as in step 3above. 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, ![print icon].

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.
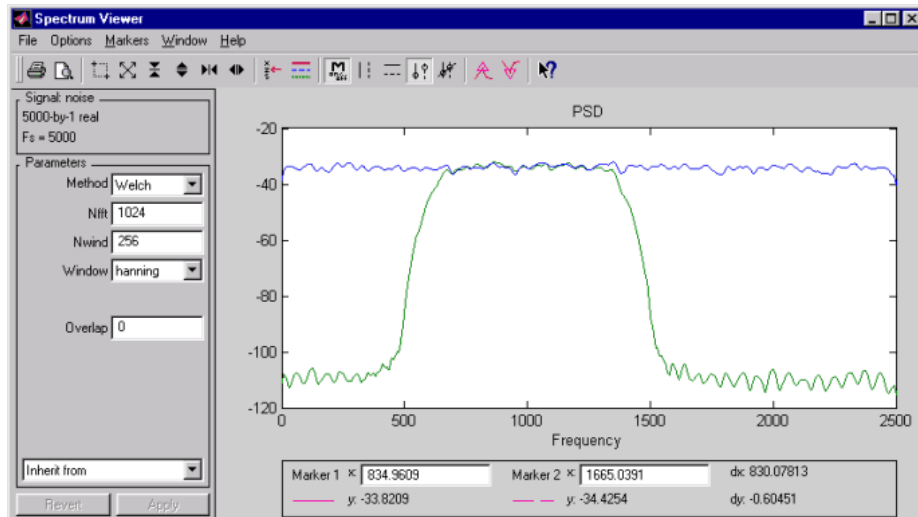
Printing the Spectra

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 8-27
"Exporting a Filter to the MATLAB Workspace" on page 8-27

Opening the Export Dialog Box

To save the filter filt1 you just created in this example, open the Export dialog box with filt1 preselected:

1   Select filt1 in the SPTool **Filters** list.

2   Select **File** > **Export**.

The Export dialog box opens with filt1 preselected.
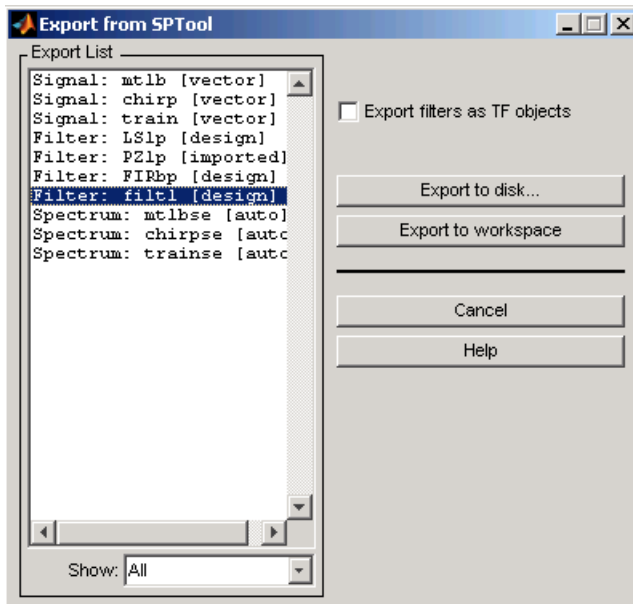
Exporting a Filter to the MATLAB Workspace

To export the filter filt1 to the MATLAB workspace:

1 Select `filt1` from the **Export List** and deselect all other items using **Ctrl+click**.

2 Click **Export to Workspace**.

Accessing Filter Parameters

In this section...
"Accessing Filter Parameters in a Saved Filter" on page 8-29
"Accessing Parameters in a Saved Spectrum" on page 8-30

Accessing Filter Parameters in a Saved Filter

The MATLAB structures created by SPTool have several associated fields, many of which are also MATLAB structures. See the MATLAB documentation for general information about MATLAB structures.

For example, after exporting a filter filt1 to the MATLAB workspace, type

```
filt1
```

to display the fields of the MATLAB filter structure. The `tf` field of the structure contains information that describes the filter.

The tf Field: Accessing Filter Coefficients

The `tf` field is a structure containing the transfer function representation of the filter. Use this field to obtain the filter coefficients;

- `filt1.tf.num` contains the numerator coefficients.
- `filt1.tf.den` contains the denominator coefficients.

The vectors contained in these structures represent polynomials in descending powers of z. The numerator and denominator polynomials are used to specify the transfer function

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

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

In this section...
"Similarities to Other Procedures" on page 8-31
"Importing Filters" on page 8-31
"Importing Spectra" on page 8-33

Similarities to Other Procedures

The procedures are very similar to those explained in

- "Importing a Signal into SPTool" on page 8-16 for loading variables from the workspace
- "Loading Variables from the Disk" on page 8-35 for loading variables from your disk

Importing Filters

When you import filters, first select the appropriate filter form from the **Form** list. SPTool does not currently support the import of filter objects.
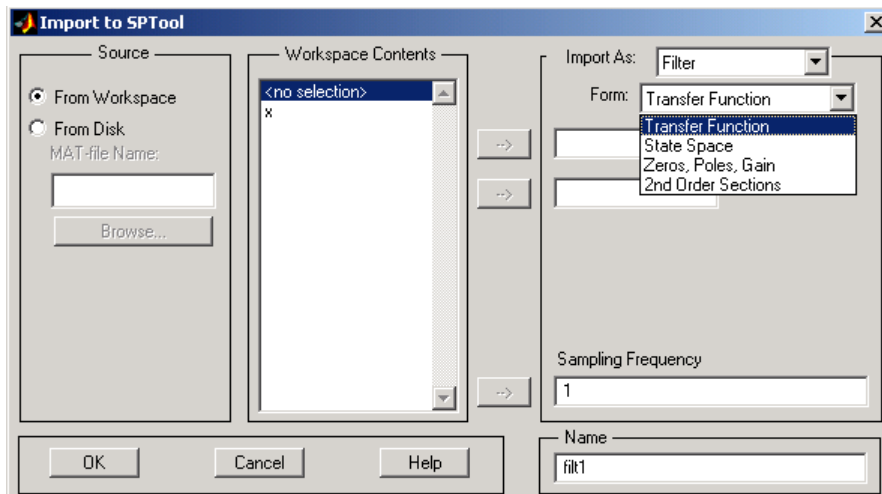
For every filter you specify a variable name or a value for the filter's sampling frequency in the **Sampling Frequency** field. Each filter form requires different variables.

Transfer Function

For `Transfer Function`, you specify the filter by its transfer function representation:

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

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

State Space

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

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

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

Zeros, Poles, Gain

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

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

- The **Zeros** field specifies a variable name or value for the zeros vector z, which contains the locations of m zeros.

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

Second Order Sections

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

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

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

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

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

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

Importing Spectra

When you import a power spectral density (PSD), you specify:

- A variable name or a value for the PSD vector in the **PSD** field
- A variable name or a value for the frequency vector in the **Freq. Vector** field

The PSD values in the **PSD** vector correspond to the frequencies contained in the **Freq. Vector** vector; the two vectors must have the same length.
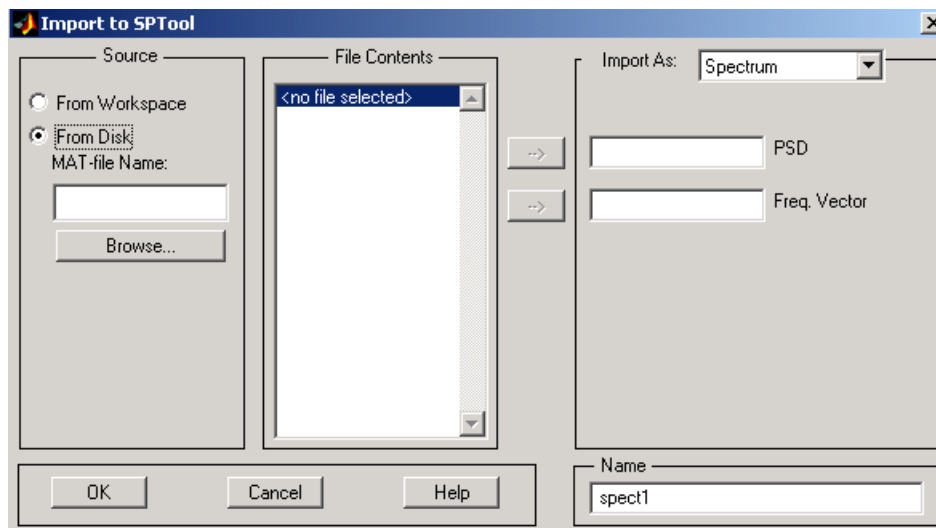
Loading Variables from the Disk

To import variables representing signals, filters, or spectra from a MAT-file on your disk;

1   Select the **From Disk** radio button and do either of the following:

   - Type the name of the file you want to import into the **MAT-file Name** field and press either the **Tab** or the **Enter** key on your keyboard.
   - Select **Browse**, and then find and select the file you want to import using **Select > File to Open**. Click **OK** to close that dialog.

   In either case, all variables in the MAT-file you selected are displayed in the **File Contents** list.

2   Select the variables to be imported into SPTool.

You can now import one or more variables from the **File Contents** list into SPTool, as long as these variables are scalars, vectors, or matrices.

Saving and Loading Sessions

In this section...
"SPTool Sessions" on page 8-36
"Filter Formats" on page 8-36

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 8-42.

- If the formats match, the session opens.
- If the filter preference is FDATool, but the session contains Filter Designer filters, this warning displays:
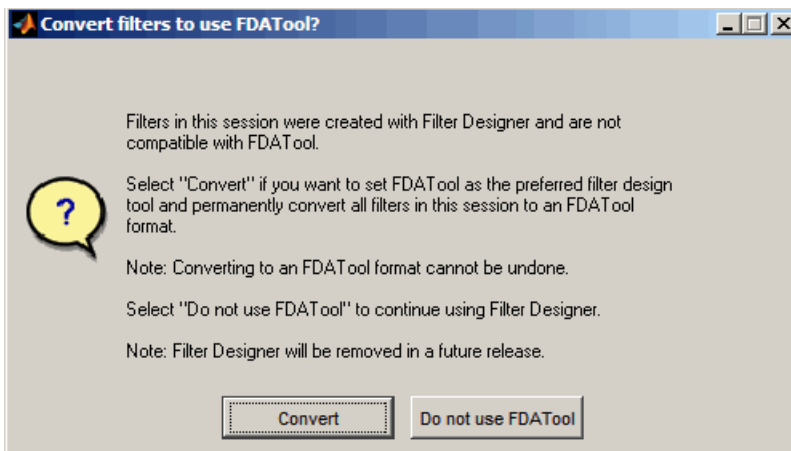
Click **Convert** to convert the filters to FDATool format. Click **Don't Use FDATool** to leave the filters in Filter Designer format and change the preference to **Use Filter Designer**.

- If the filter preference is **Use Filter Designer**, but the session contains FDATool filters, this warning displays:
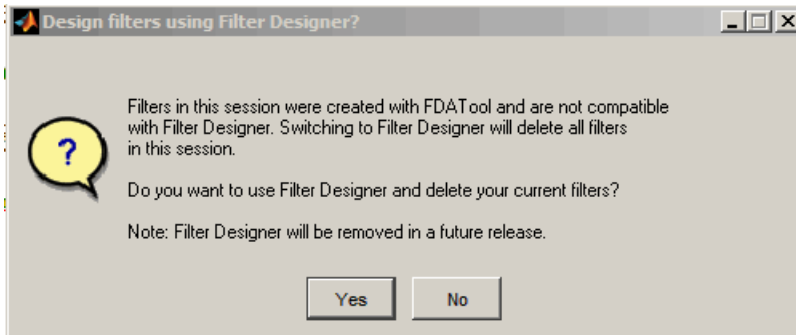
Click **Yes** to remove the current filters. Click **No** to leave the filters in FDATool.

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" on page 8-42
"Summary of Settable Preferences" on page 8-43
"Setting the Filter Design Tool" on page 8-43

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.
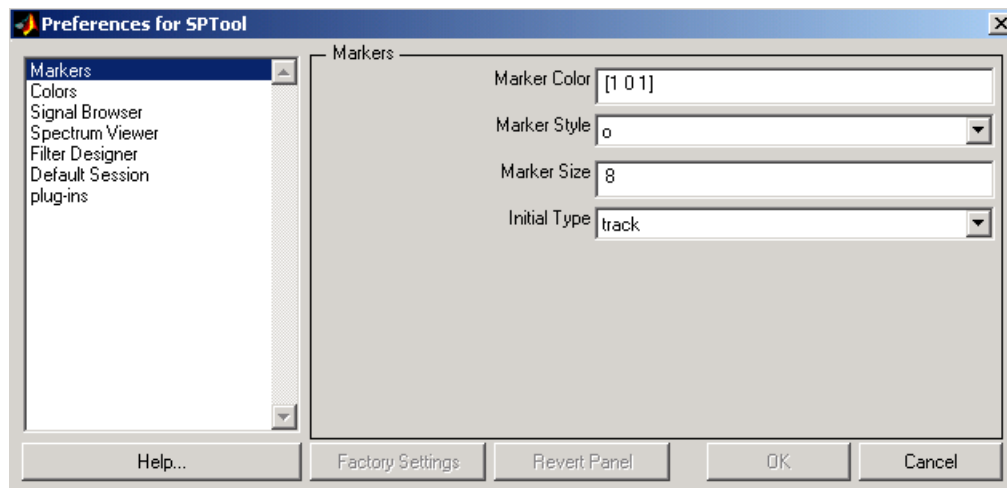
Change any marker settings, if desired. To change settings for another category, click its name in the category list to display its settings. Most of the fields are self-explanatory. Details of the Filter Design options are described below.

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.
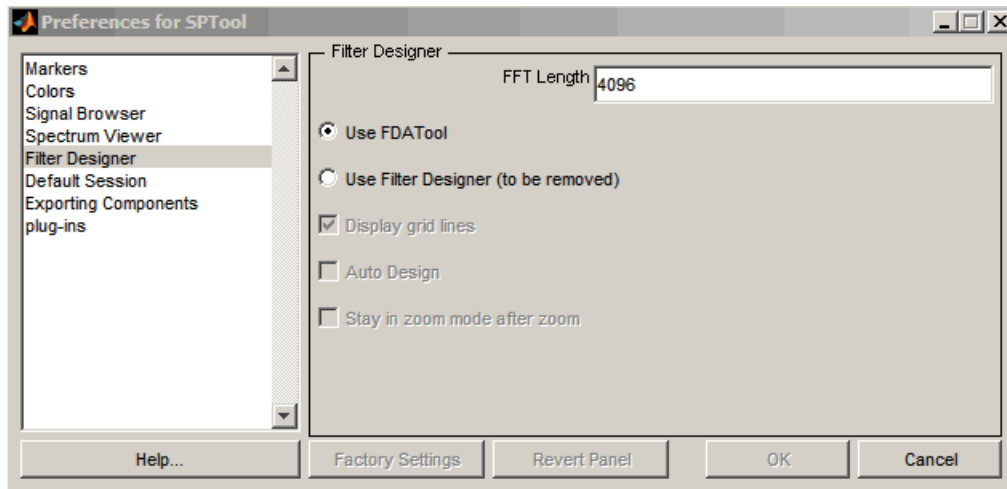
FDATool is the default and recommended tool. You can use Filter Designer, but Filter Designer will be removed in a future release. You cannot change this preference if either FDATool or the Filter Designer is open.

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.
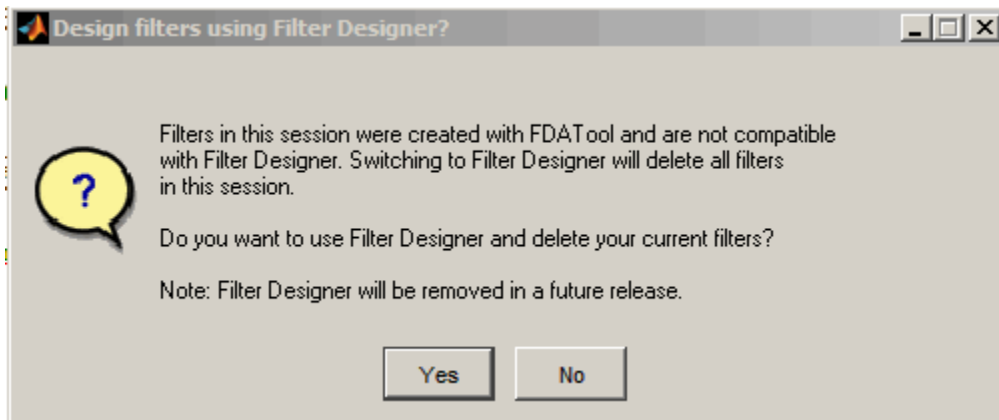
See "Saving and Loading Sessions" on page 8-36 for more information.

When you change the preference from **Use Filter Designer** to **Use FDATool**, a confirmation message appears indicating that switching will convert your filters to FDATool format. See "Saving and Loading Sessions" on page 8-36 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:
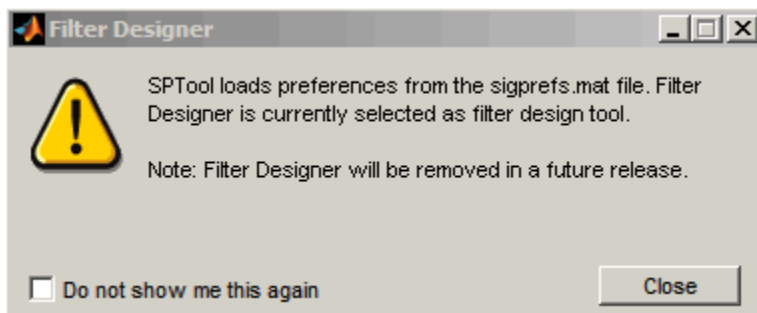
The warning appears each time SPTool initializes unless you opt to not show the warning.

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**
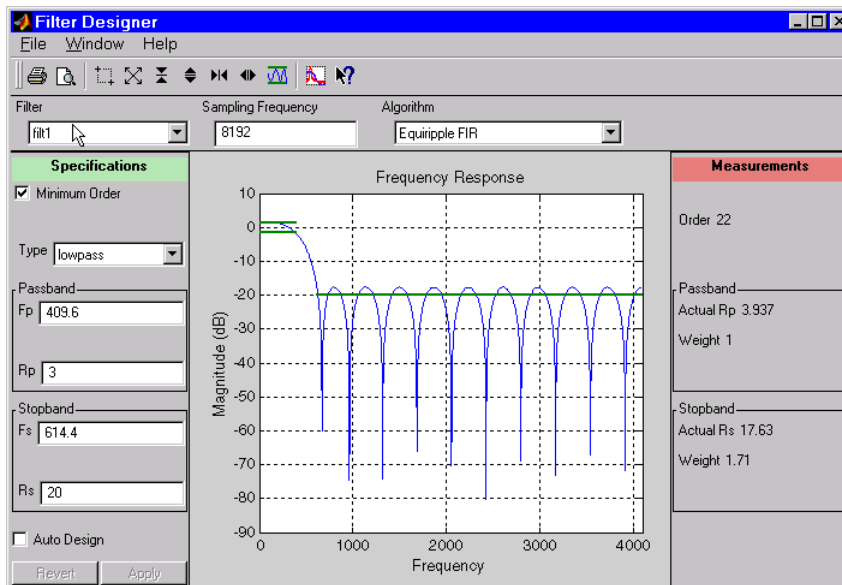
The Filter Designer has the following components:

- 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

Icon	Description
⊡ ⋈ ⊼ ⬍ ⊳⊲	Zoom in and out
〽	Passband view
📈	Overlay spectrum
▶?	Turn on the What's This help

Accessing Filter Parameters in a Saved Filter

The MATLAB structures created by SPTool have several associated fields, many of which are also MATLAB structures. See the MATLAB documentation for general information about MATLAB structures.

For example, after exporting a filter filt1 to the MATLAB workspace, type

```
filt1
```

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

The tf Field: Accessing Filter Coefficients

The `tf` field is a structure containing the transfer function representation of the filter. Use this field to obtain the filter coefficients;

- `filt1.tf.num` contains the numerator coefficients.
- `filt1.tf.den` contains the denominator coefficients.

The vectors contained in these structures represent polynomials in descending powers of *z*. The numerator and denominator polynomials are used to specify the transfer function

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

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)

Parameter	Description
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.
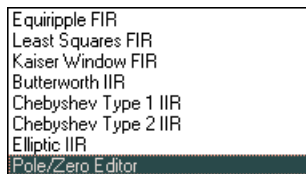
2 Enter the desired filter gain in the **Gain** edit box.

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
	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.
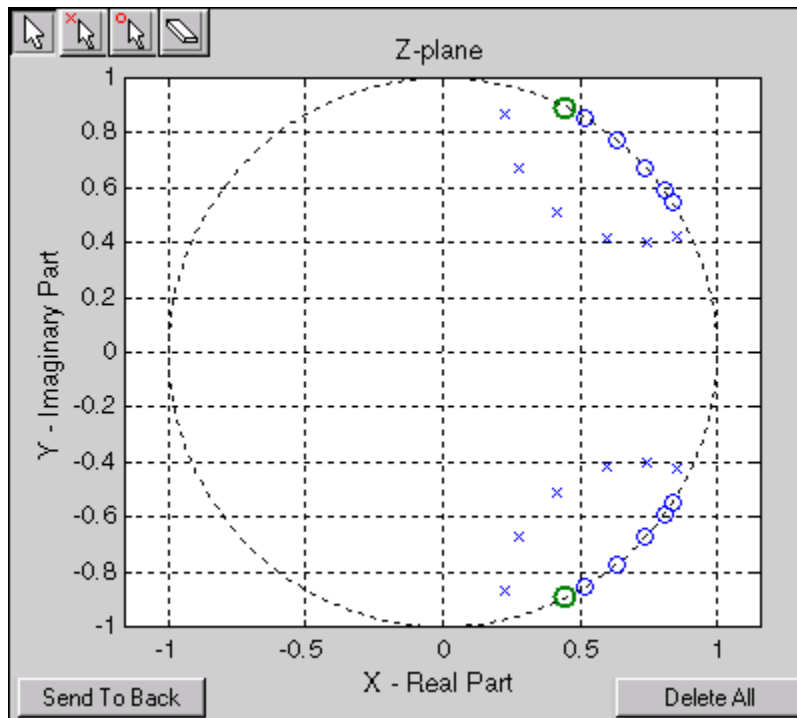
When you place two or more poles (or two or more zeros) directly on top of each other, a number is displayed next to the symbols (on the left for poles, and on the right for zeros) indicating the number of poles or zeros at that location (e.g., ⊙3 for three zeros). This number makes it easy to keep track 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.

Depending on which feature you wish to use, there are additional required products. For a comprehensive list, see "Installing Prerequisite Products" in the MATLAB Coder™ documentation.

Code generation from MATLAB supports Signal Processing Toolbox functions listed in the table. To generate C code, you must have the MATLAB Coder software. If you have the Fixed-Point Designer software, you can use `fiaccel` to generate MEX code for fixed-point applications.

To follow the examples in this documentation:

- To generate C/C++ code and MEX files with `codegen`, install the MATLAB Coder software, the Signal Processing Toolbox, and a C compiler. For the Windows® platform, MATLAB supplies a default C compiler. Run `mex -setup` at the MATLAB command prompt to set up the C compiler.
- Change to a folder where you have write permission.

Note: Many Signal Processing Toolbox functions require constant inputs in generated code. To specify a constant input for `codegen`, use `coder.Constant`.

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.

Function	Remarks/Limitations
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.
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.
db2pow	—

Function	Remarks/Limitations
dct	C and C++ code generation for dct requires DSP System Toolbox software. 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.
findpeaks	—
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.
firpmord	All inputs must be constants. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
flattopwin	All inputs must be constants. Expressions or variables are allowed if their values do not change.
freqz	When called with no output arguments, and without a semicolon at the end, freqz returns the complex frequency response of the input filter, evaluated at 512 points. If the semicolon is added, the function produces a plot of the magnitude and phase response of the filter. See "freqz With No Output Arguments".
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	C and C++ code generation for idct requires DSP System Toolbox software. 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	C and C++ code generation for levinson requires DSP System Toolbox software. 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.

Function	Remarks/Limitations
nuttallwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
parzenwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
pow2db	—
rcosdesign	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	C and C++ code generation for upfirdn requires DSP System Toolbox software. 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	—

Function	Remarks/Limitations
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" on page 9-8
"Inputs must be Constants" on page 9-9

Defining Input Size and Type

When you use Signal Processing Toolbox functions for code generation, you must define the size and type of the function inputs. One way to do this is with the `-args` compilation option. The size and type of inputs must be defined because C is a statically typed language. . To illustrate the need to define input size and type, consider the simplest call to `xcorr` requiring two input arguments. The following demonstrates the differences in the use of `xcorr` in MATLAB and in Code Generation from MATLAB.

Cross correlate two white noise vectors in MATLAB:

```
x = randn(512,1); %real valued white noise
y = randn(512,1); %real valued white noise
[C,lags] = xcorr(x,y);
x_circ = randn(256,1)+1j*randn(256,1); %circular white noise
y_circ = randn(256,1)+1j*randn(256,1); %circular white noise
[C1,lags1] = xcorr(x_circ,y_circ);
```

`xcorr` does not require the size and type of the input arguments. `xcorr` obtains this information at runtime. Contrast this behavior with a MEX-file created with `codegen`. Create the file `myxcorr.m` in a folder where you have read and write permission. Ensure that this folder is in the MATLAB search path. Copy and paste the following two lines of code into `myxcorr.m` and save the file. The compiler tag `%#codegen` must be included in the file.

```
function [C,Lags]=myxcorr(x,y)  %#codegen
[C,Lags]=xcorr(x,y);
```

Enter the following command at the MATLAB command prompt:

```
codegen myxcorr -args {zeros(512,1),zeros(512,1)} -o myxcorr
```

Run the MEX-file:

```
x = randn(512,1); %real valued white noise
```

```
y = randn(512,1); %real valued white noise
[C,Lags] = myxcorr(x,y);
```

Define two new inputs x1 and y1 by transposing x and y.

```
x1 = x'; %x1 is 1x512
y1 = y'; %y1 is 1x512
```

Attempt to rerun the MEX-file with the 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.

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',0.1) } -repor
```

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 9-12 ,"Apply Lowpass Filter to Input Signal" on page 9-14, and "Zero Phase Filtering" on page 9-16 for additional examples of the constant input requirement.

Code Generation Examples

In this section...

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)} -o window_data
```

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`.
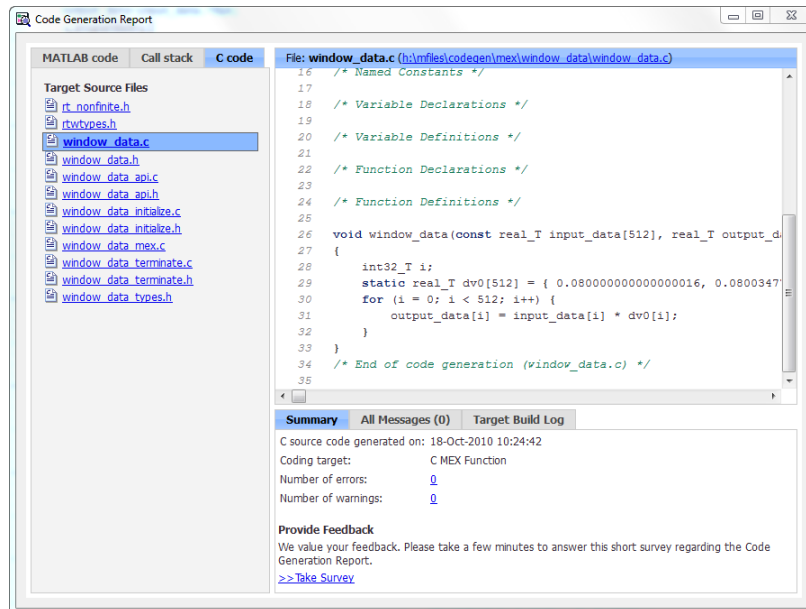
Note from the location bar that the C source code is in the `codegen/mex/<FUNCTION_NAME>` folder. Running `codegen` creates this folder and places the C source code, C header files, and MEX files in the folder. Each function that you create produces a `codegen/mex/<FUNCTION_NAME>` folder.

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, 0.061256884747218
static real_T dv1[5] = { 1.0, -1.9684277869385174, 1.7358607092088851, -0.724470829507
```

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');
```
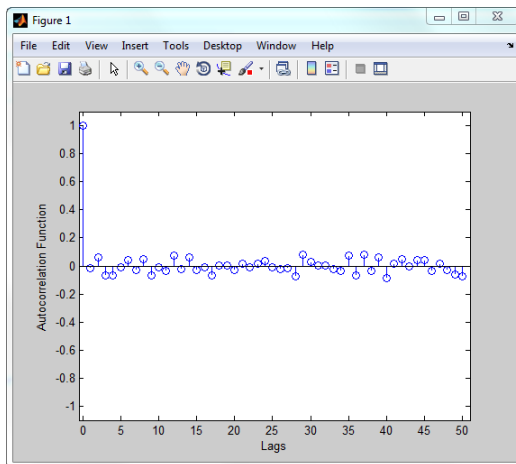

freqz With No Output Arguments

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');
```

10
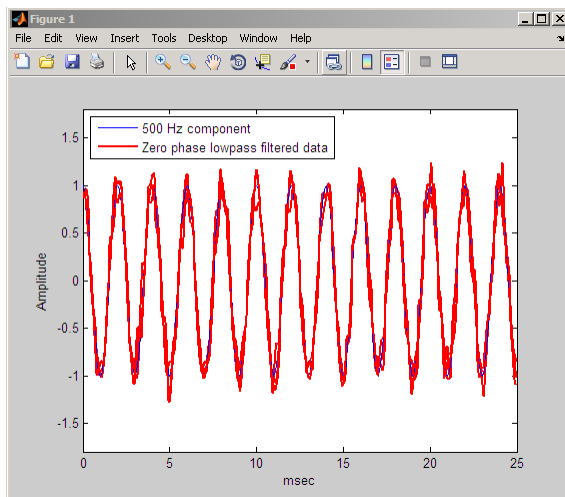
Convolution and Correlation

Linear and Circular Convolution

This example shows how to establish an equivalence between linear and circular convolution.

Linear and circular convolution are fundamentally different operations. However, there are conditions under which linear and circular convolution are equivalent. Establishing this equivalence has important implications. For two vectors, *x* and *y*, the circular convolution is equal to the inverse discrete Fourier transform (DFT) of the product of the vectors' DFTs. Knowing the conditions under which linear and circular convolution are equivalent allows you to use the DFT to efficiently compute linear convolutions.

The linear convolution of an N-point vector, *x*, and a L-point vector, *y*, has length N+L-1.

For the circular convolution of *x* and *y* to be equivalent, you must pad the vectors with zeros to length at least N+L-1 before you take the DFT. After you invert the product of the DFTs, retain only the first N+L-1 elements.

Create two vectors, *x* and *y*, and compute the linear convolution of the two vectors.

```
x = [2 1 2 1];
y = [1 2 3];
clin = conv(x,y);
```

The output has length 4+3–1.

Pad both vectors with zeros to length 4+3–1. Obtain the DFT of both vectors, multiply the DFTs, and obtain the inverse DFT of the product.

```
xpad = [x zeros(1,6-length(x))];
ypad = [y zeros(1,6-length(y))];
ccirc = ifft(fft(xpad).*fft(ypad));
```

The circular convolution of the zero-padded vectors, xpad and ypad, is equivalent to the linear convolution of x and y. You retain all the elements of ccirc because the output has length 4+3–1.

Plot the output of linear convolution and the inverse of the DFT product to show the equivalence.

```
subplot(211)
stem(clin,'markerfacecolor',[0 0 1]);
title('Linear Convolution of x and y');
```

```
set(gca,'ylim',[O 11]);
subplot(212)
stem(ccirc,'markerfacecolor',[O O 1]);
set(gca,'ylim',[O 11]);
title('Circular Convolution of xpad and ypad');
```

Linear Convolution of x and y

Circular Convolution of xpad and ypad
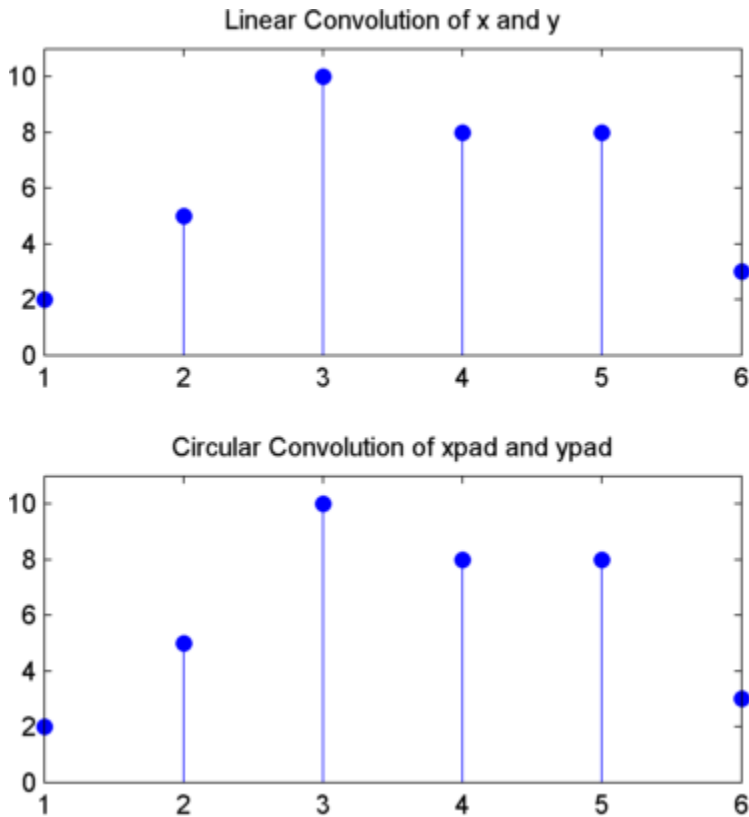
Pad the vectors to length 12 and obtain the circular convolution using the inverse DFT of the product of the DFTs. Retain only the first 4+3−1 elements to produce an equivalent result to linear convolution.

```
N = length(x)+length(y)-1;
xpad = [x zeros(1,12-length(x))];
ypad = [y zeros(1,12-length(y))];
ccirc = ifft(fft(xpad).*fft(ypad));
ccirc = ccirc(1:N);
```

The Signal Processing Toolbox software has a function, `cconv`, that returns the circular convolution of two vectors. You can obtain the linear convolution of `x` and `y` using circular convolution with the following code.

```
ccirc2 = cconv(x,y,6);
```

`cconv` internally uses the same DFT-based procedure illustrated in the previous example.

Confidence Intervals for Sample Autocorrelation

This example shows how to create confidence intervals for the autocorrelation sequence of a white noise process. Create a realization of a white noise process 1000 samples in length. Compute the sample autocorrelation to lag 20. Plot the sample autocorrelation along with the approximate 95%-confidence intervals for a white noise process.

Create the white noise random vector. Set the random number generator to the default settings for reproducible results. Obtain the normalized sampled autocorrelation to lag 20.

```
rng default
x = randn(1000,1);
[xc,lags] = xcorr(x,20,'coeff');
```

Create the lower and upper 95% confidence bounds for the $N(0, 1/N)$ distribution. For a 95%-confidence interval, the critical value is 1.96. The standard deviation is the square root of $1/N$, where N is the length of the input vector.

The confidence interval is

$$0 \pm \frac{1.96}{\sqrt{1000}}.$$

```
lconf = -1.96/sqrt(length(x));
upconf = 1.96/sqrt(length(x));
```

Plot the sample autocorrelation along with the 95%-confidence interval.

```
stem(lags,xc,'filled')
ylim([lconf-0.03 1.05])
hold on
plot(lags,lconf*ones(size(lags)),'r','linewidth',2)
plot(lags,upconf*ones(size(lags)),'r','linewidth',2)
title('Sample Autocorrelation with 95% Confidence Intervals')
```

Sample Autocorrelation with 95% Confidence Intervals
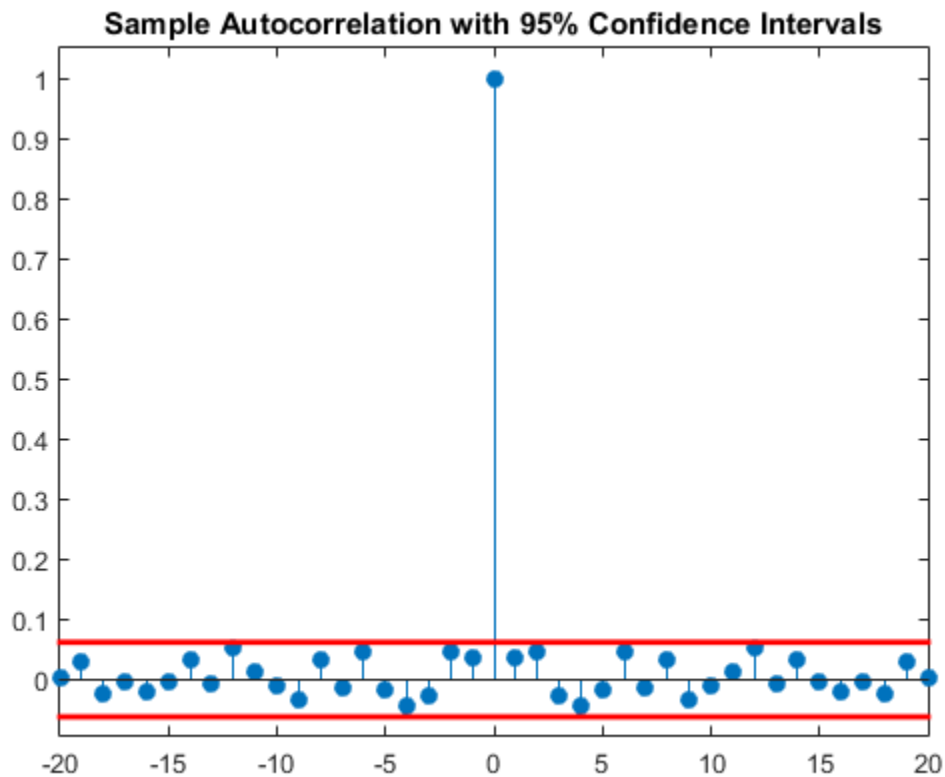
You see in the above figure that the only autocorrelation value outside of the 95%-confidence interval occurs at lag 0 as expected for a white noise process. Based on this result, you can conclude that the data are a realization of a white noise process.

Residual Analysis with Autocorrelation

This example shows how to use autocorrelation with a confidence interval to analyze the residuals of a least-squares fit to noisy data. The residuals are the differences between the fitted model and the data. In a signal-plus-white noise model, if you have a good fit for the signal, the residuals should be white noise.

Create a noisy data set consisting of a 1st-order polynomial (straight line) in additive white Gaussian noise. The additive noise is a sequence of uncorrelated random variables following a $N(0,1)$ distribution. This means that all the random variables have mean zero and unit variance. Set the random number generator to the default settings for reproducible results.

```
x = -3:0.01:3;
rng default
y = 2*x+randn(size(x));
plot(x,y)
```
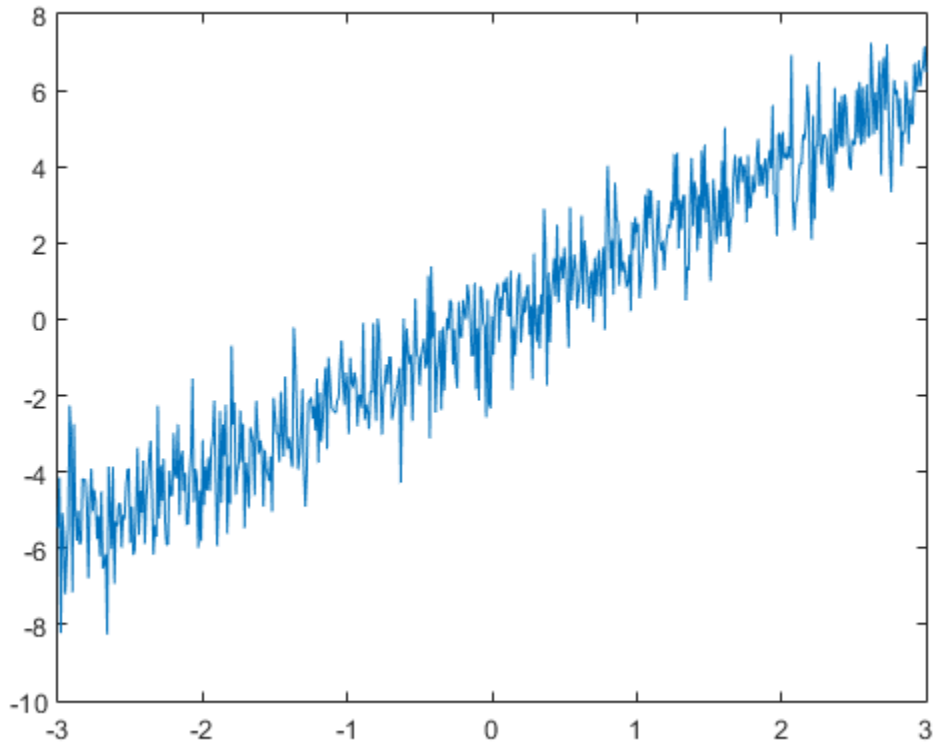
Use `polyfit` to find the least-squares line for the noisy data. Plot the original data along with the least-squares fit.

```
coeffs = polyfit(x,y,1);
yfit = coeffs(2)+coeffs(1)*x;

plot(x,y)
hold on
plot(x,yfit,'linewidth',2)
```
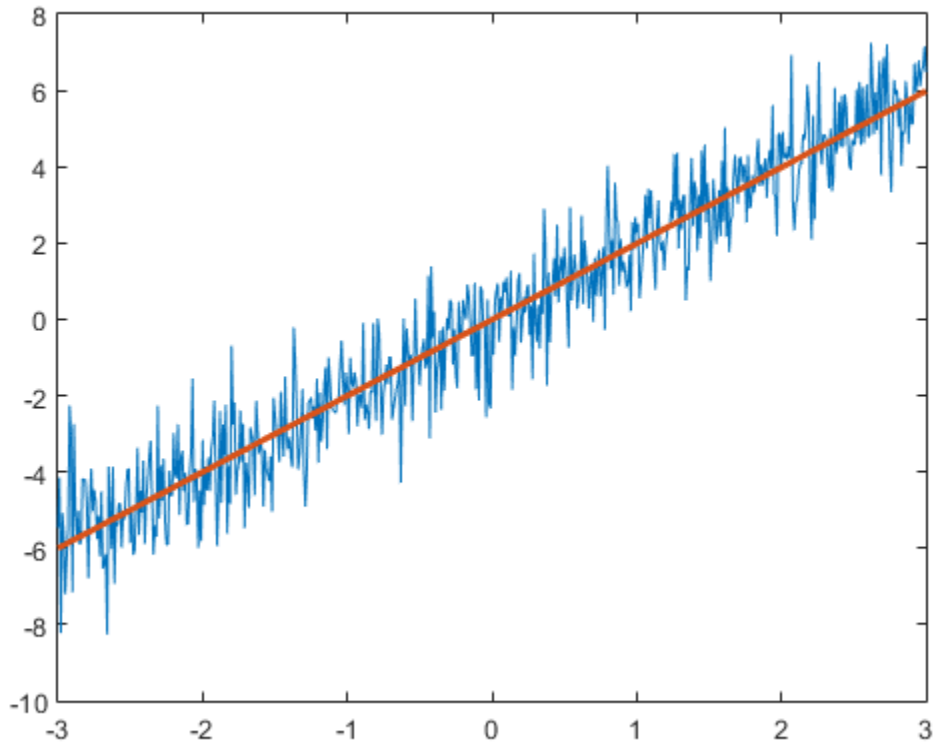
Find the residuals. Obtain the autocorrelation sequence of the residuals to lag 50.

```
residuals = y - yfit;
[xc,lags] = xcorr(residuals,50,'coeff');
```

When you inspect the autocorrelation sequence, you want to determine whether or not there is evidence of autocorrelation. In other words, you want to determine whether the sample autocorrelation sequence looks like the autocorrelation sequence of white noise. If the autocorrelation sequence of the residuals looks like the autocorrelation of a white noise process, you are confident that none of the signal has escaped your fit and ended up in the residuals. In this example, use a 99%-confidence interval. To construct the confidence interval, you need to know the distribution of the sample autocorrelation values. You also need to find the critical values on the appropriate distribution between

which lie 0.99 of the probability. Because the distribution in this case is Gaussian, you can use complementary inverse error function, `erfcinv`. The relationship between this function and the inverse of the Gaussian cumulative distribution function is described on the reference page for `erfcinv`.

Find the critical value for the 99%-confidence interval. Use the critical value to construct the lower and upper confidence bounds.

```
conf99 = sqrt(2)*erfcinv(2*.01/2);
lconf = -conf99/sqrt(length(x));
upconf = conf99/sqrt(length(x));
```

Plot the autocorrelation sequence along with the 99%-confidence intervals.

```
figure

stem(lags,xc,'filled')
ylim([lconf-0.03 1.05])
hold on
plot(lags,lconf*ones(size(lags)),'r','linewidth',2)
plot(lags,upconf*ones(size(lags)),'r','linewidth',2)
title('Sample Autocorrelation with 99% Confidence Intervals')
```

Sample Autocorrelation with 99% Confidence Intervals
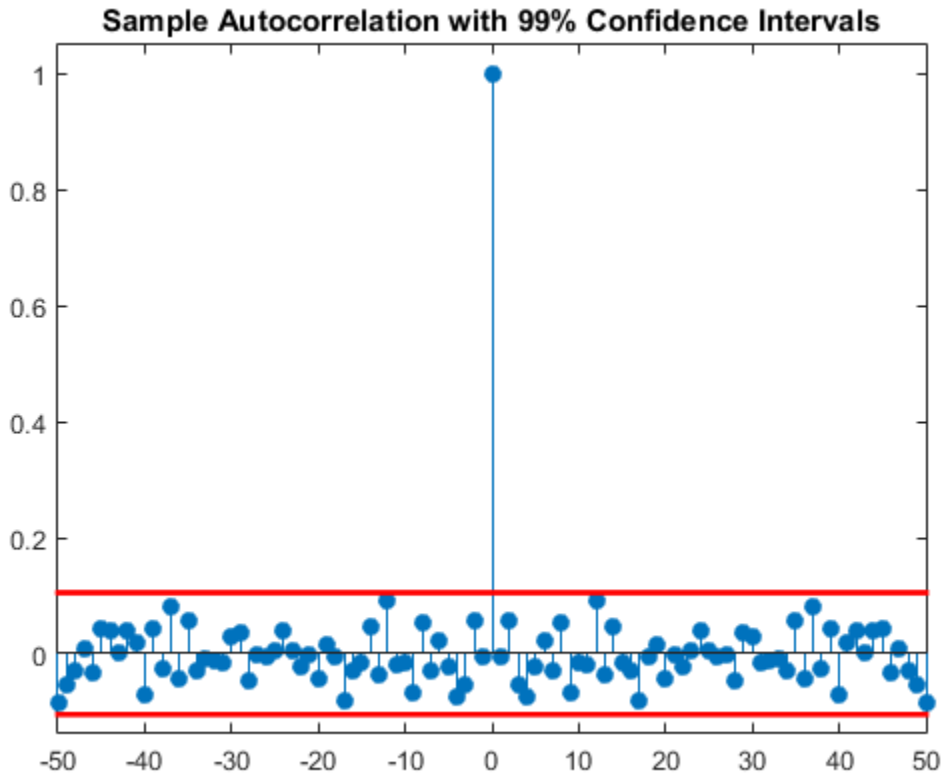
Except at zero lag, the sample autocorrelation values lie within the 99%-confidence bounds for the autocorrelation of a white noise sequence. From this, you can conclude that the residuals are white noise. More specifically, you cannot reject that the residuals are a realization of a white noise process.

Create a signal consisting of a sine wave plus noise. The data are sampled at 1 kHz. The frequency of the sine wave is 100 Hz. Set the random number generator to the default settings for reproducible results.

```
Fs = 1000;
t = 0:1/Fs:1-1/Fs;
rng default
x = cos(2*pi*100*t)+randn(size(t));
```

Use the discrete Fourier transform (DFT) to obtain the least-squares fit to the sine wave at 100 Hz. The least-squares estimate of the amplitude is $2/N$ times the DFT coefficient corresponding to 100 Hz, where N is the length of the signal. The real part is the amplitude of a cosine at 100 Hz and the imaginary part is the amplitude of a sine at 100 Hz. The least-squares fit is the sum of the cosine and sine with the correct amplitude. In this example, DFT bin 101 corresponds to 100 Hz.

```
xdft = fft(x);
ampest = 2/length(x)*xdft(101);
xfit = real(ampest)*cos(2*pi*100*t)+imag(ampest)*sin(2*pi*100*t);

figure

plot(t,x)
hold on
plot(t,xfit,'linewidth',2)
axis([0 0.30 -4 4])
xlabel('Seconds')
ylabel('Amplitude')
```
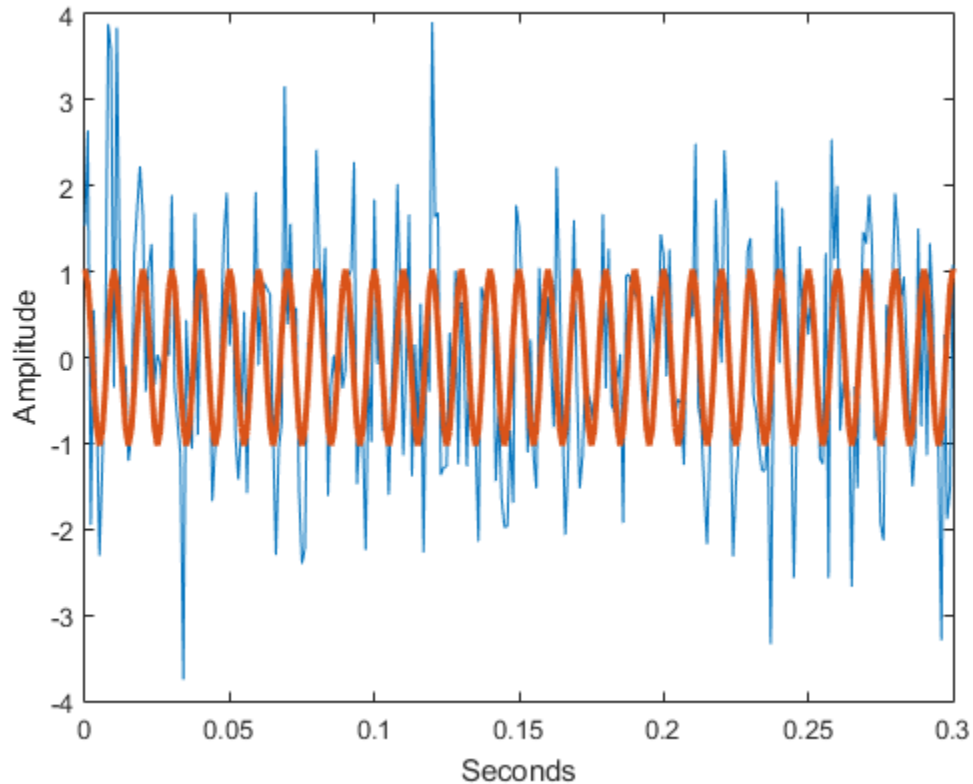
Find the residuals and determine the sample autocorrelation sequence to lag 50.

```
residuals = x-xfit;
[xc,lags] = xcorr(residuals,50,'coeff');
```

Plot the autocorrelation sequence with the 99%-confidence intervals.

```
figure

stem(lags,xc,'filled')
ylim([lconf-0.03 1.05])
hold on
plot(lags,lconf*ones(size(lags)),'r','linewidth',2)
plot(lags,upconf*ones(size(lags)),'r','linewidth',2)
```

```
title('Sample Autocorrelation with 99% Confidence Intervals')
```
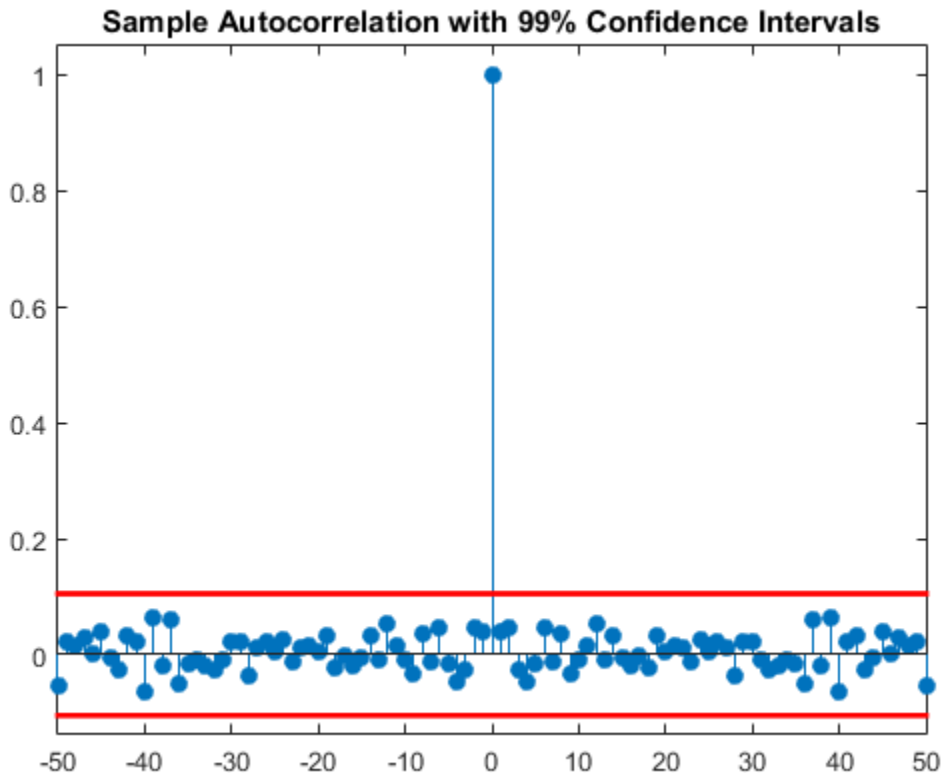

Again, you see that except at zero lag, the sample autocorrelation values lie within the 99%-confidence bounds for the autocorrelation of a white noise sequence. From this, you can conclude that the residuals are white noise. More specifically, you cannot reject that the residuals are a realization of a white noise process.

Finally, add another sine wave with a frequency of 200 Hz and an amplitude of 3/4. Fit only the sine wave at 100 Hz and find the sample autocorrelation of the residuals.

```
x = x+3/4*sin(2*pi*200*t);
xdft = fft(x);
ampest = 2/length(x)*xdft(101);
```

```
xfit = real(ampest)*cos(2*pi*100*t)+imag(ampest)*sin(2*pi*100*t);
residuals = x-xfit;
[xc,lags] = xcorr(residuals,50,'coeff');
```

Plot the sample autocorrelation along with the 99%-confidence intervals.

```
figure

stem(lags,xc,'filled')
ylim([lconf-0.12 1.05])
hold on
plot(lags,lconf*ones(size(lags)),'r','linewidth',2)
plot(lags,upconf*ones(size(lags)),'r','linewidth',2)
title('Sample Autocorrelation with 99% Confidence Intervals')
```
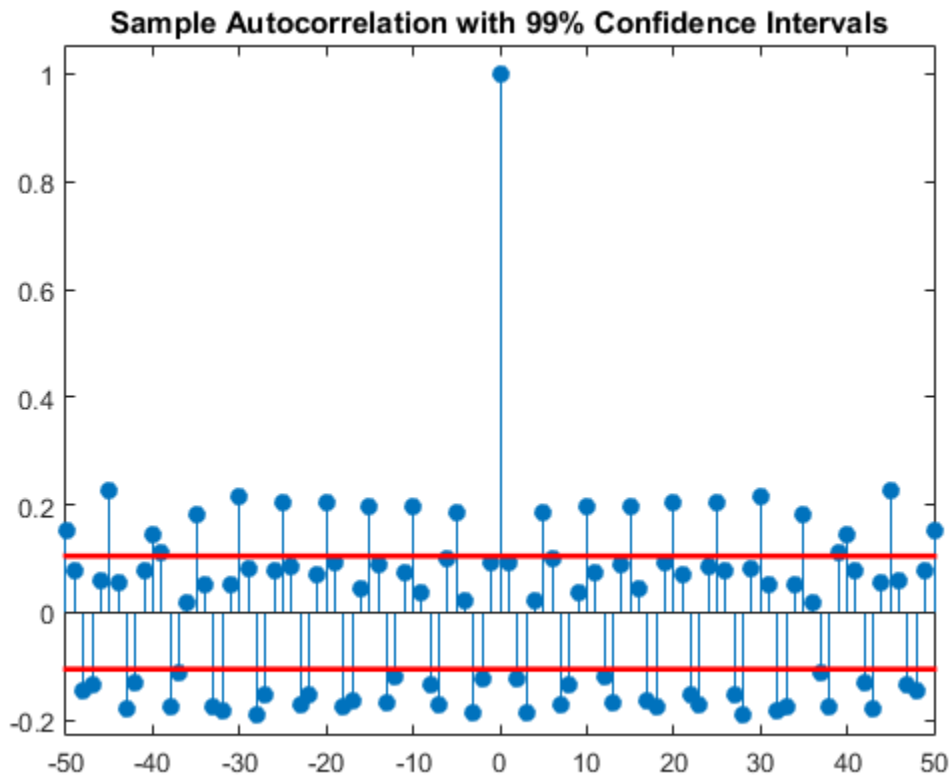
In this case, the autocorrelation values clearly exceed the 99%-confidence bounds for a white noise autocorrelation at many lags. Here you can reject the hypothesis that the residuals are a white noise sequence. The implication is that the model has not accounted for all the signal and therefore the residuals consist of signal plus noise.

Autocorrelation of Moving Average Process

This example shows how to introduce autocorrelation into a white noise process by filtering. When we introduce autocorrelation into a random signal, we manipulate its frequency content. A moving average filter attenuates the high-frequency components of the signal, effectively smoothing it.

Create the impulse response for a 3-point moving average filter. Filter an $N(0,1)$ white noise sequence with the filter. Set the random number generator to the default settings for reproducible results.

```
h = 1/3*ones(3,1);
rng default
x = randn(1000,1);
y = filter(h,1,x);
```

Obtain the biased sample autocorrelation out to 20 lags. Plot the sample autocorrelation along with the theoretical autocorrelation.

```
[xc,lags] = xcorr(y,20,'biased');

Xc = zeros(size(xc));
Xc(19:23) = [1 2 3 2 1]/9*var(x);

stem(lags,xc,'filled')
hold on
stem(lags,Xc,'.','linewidth',2)

lg = legend('Sample autocorrelation','Theoretical autocorrelation');
lg.Location = 'NorthEast';
lg.Box = 'off';
```
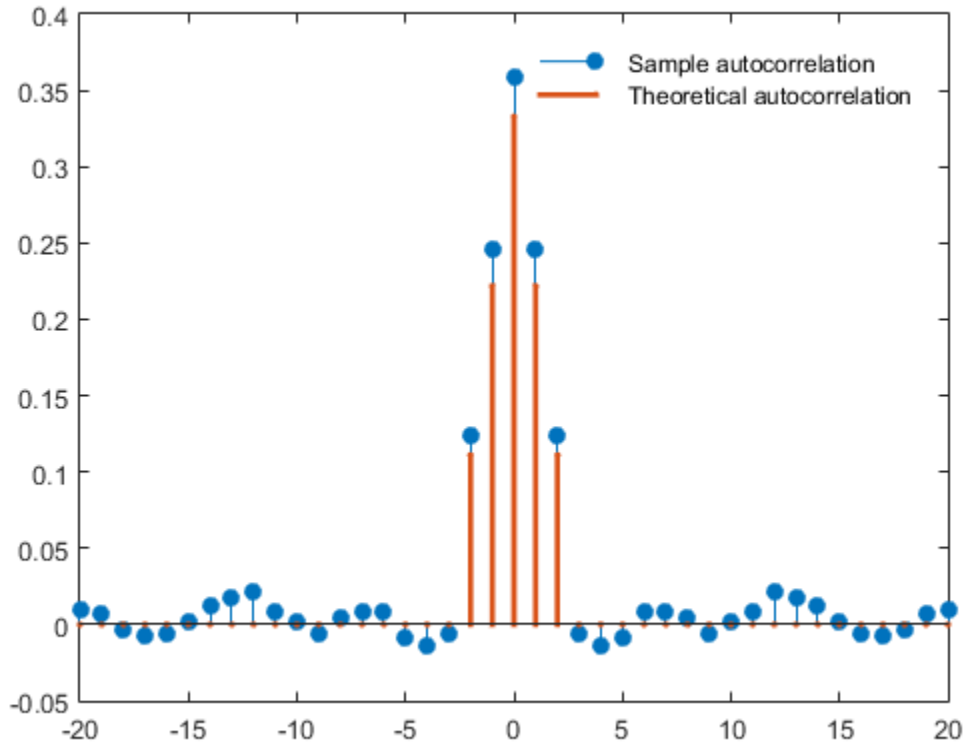
The sample autocorrelation captures the general form of the theoretical autocorrelation, even though the two sequences do not agree in detail.

In this case, it is clear that the filter has introduced significant autocorrelation only over lags [-2,2]. The absolute value of the sequence decays quickly to zero outside of that range.

To see that the frequency content has been affected, plot Welch estimates of the power spectral densities of the original and filtered signals.
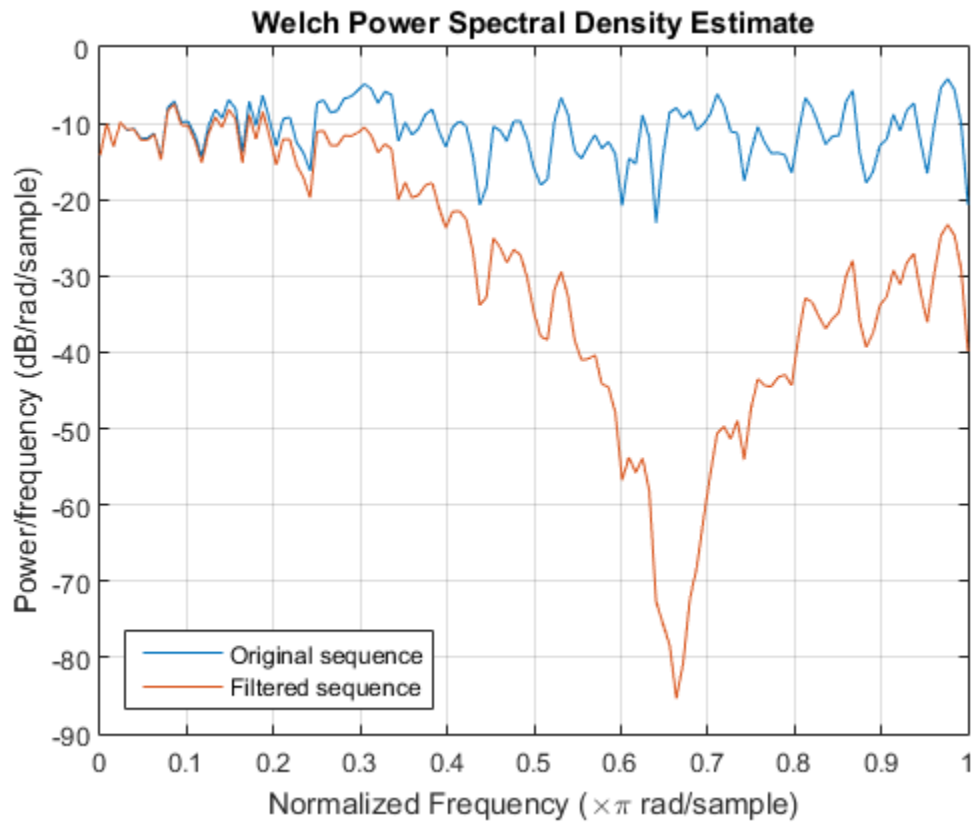
```
[pxx,wx] = pwelch(x);
[pyy,wy] = pwelch(y);
```

```
figure
plot(wx/pi,20*log10(pxx),wy/pi,20*log10(pyy))

lg = legend('Original sequence','Filtered sequence');
lg.Location = 'SouthWest';

xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Power/frequency (dB/rad/sample)')
title('Welch Power Spectral Density Estimate')
grid
```

The white noise has been "colored" by the moving average filter.

External Web Sites

- Ellis, Dan. About Colored Noise. http://www.ee.columbia.edu/~dpwe/noise/

Cross-Correlation of Two Moving Average Processes

This example shows how to find and plot the cross-correlation sequence between two moving average processes. The example compares the sample cross-correlation with the theoretical cross-correlation. Filter an $N(0,1)$ white noise input with two different moving average filters. Plot the sample and theoretical cross-correlation sequences.

Create an $N(0,1)$ white noise sequence. Set the random number generator to the default settings for reproducible results. Create two moving average filters. One filter has impulse response $\delta(n) + \delta(n-1)$. The other filter has impulse response $\delta(n) - \delta(n-1)$.

```
rng default

w = randn(100,1);
x = filter([1 1],1,w);
y = filter([1 -1],1,w);
```

Obtain the sample cross-correlation sequence up to lag 20. Plot the sample cross-correlation along with the theoretical cross-correlation.

```
[xc,lags] = xcorr(x,y,20,'biased');

Xc = zeros(size(xc));
Xc(20) = -1;
Xc(22) = 1;

stem(lags,xc,'filled')
hold on
stem(lags,Xc,'.','linewidth',2)

q = legend('Sample cross-correlation','Theoretical cross-correlation');
q.Location = 'NorthWest';
q.FontSize = 9;
q.Box = 'off';
```
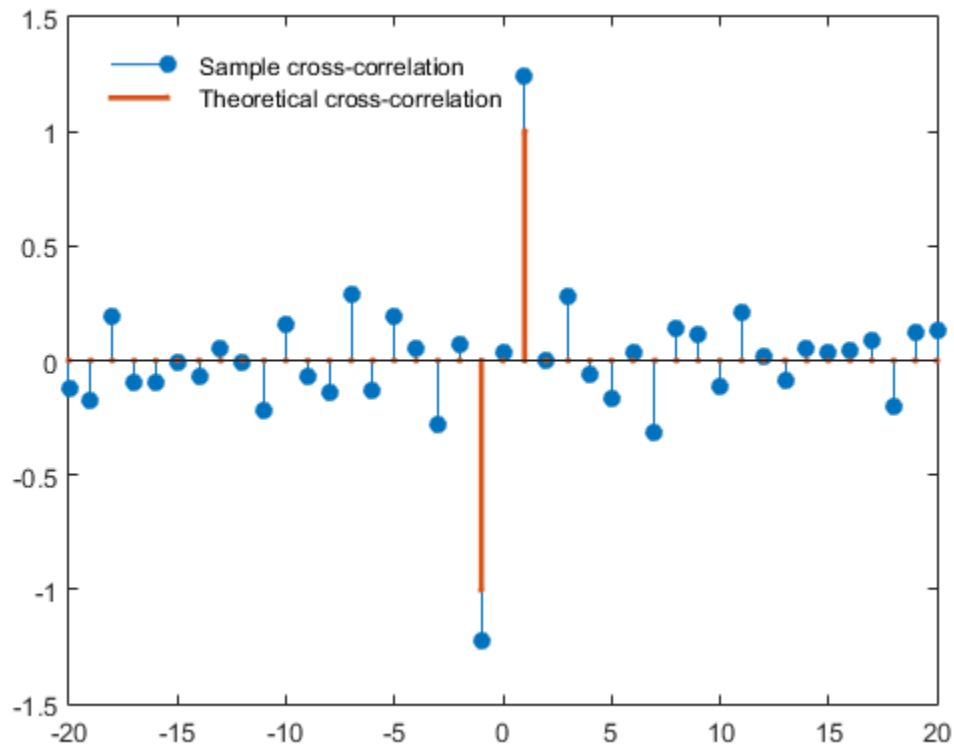
The theoretical cross-correlation is -1 at lag -1, 1 at lag 1, and zero at all other lags. The sample cross-correlation sequence approximates the theoretical cross-correlation.

As expected, there is not perfect agreement between the theoretical cross-correlation and sample cross-correlation. The sample cross-correlation does accurately represent both the sign and magnitude of the theoretical cross-correlation sequence values at lag -1 and lag 1.

Cross-Correlation of Delayed Signal in Noise

This example shows how to use the cross-correlation sequence to detect the time delay in a noise-corrupted sequence. The output sequence is a delayed version of the input sequence with additive white Gaussian noise. Create two sequences. One sequence is a delayed version of the other. The delay is 3 samples. Add $N(0, 0.3^2)$ white noise to the delayed signal. Use the sample cross-correlation sequence to detect the lag.

Create and plot the signals. Set the random number generator to the default settings for reproducible results.

```
rng default

x = triang(20);
y = [zeros(3,1);x]+0.3*randn(length(x)+3,1);

subplot(2,1,1)
stem(x,'filled')
axis([0 22 -1 2])
title('Input Sequence')

subplot(2,1,2)
stem(y,'filled')
axis([0 22 -1 2])
title('Output Sequence')
```

Input Sequence

Output Sequence
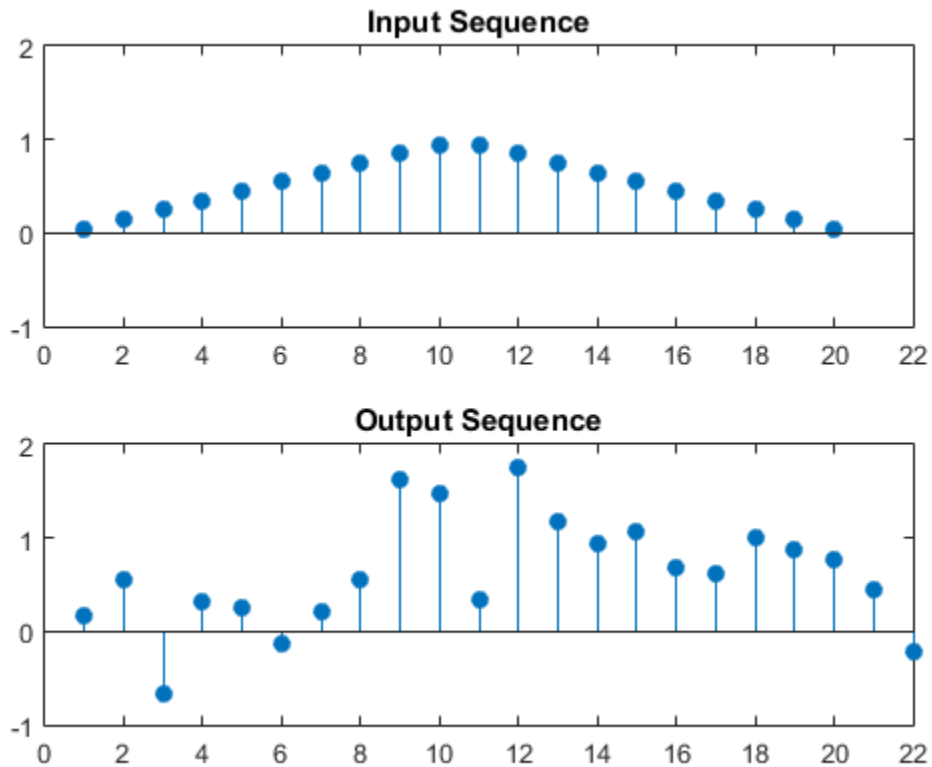
Obtain the sample cross-correlation sequence and use the maximum absolute value to estimate the lag. Plot the sample cross-correlation sequence.

```
[xc,lags] = xcorr(y,x);
[~,I] = max(abs(xc));

figure
stem(lags,xc,'filled')
legend(sprintf('Maximum at lag %d',lags(I)))
title('Sample Cross-Correlation Sequence')
```
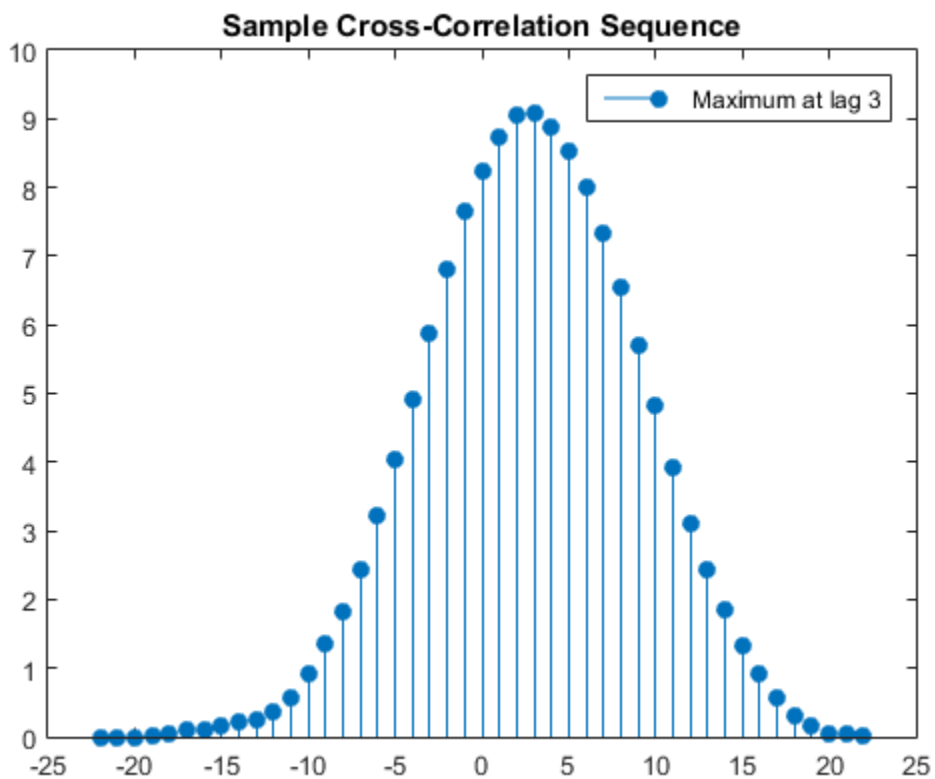
The maximum cross correlation sequence value occurs at lag 3 as expected.

Cross-Correlation of Phase-Lagged Sine Wave

This example shows how to use the cross-correlation sequence to estimate the phase lag between two sine waves. The theoretical cross-correlation sequence of two sine waves at the same frequency also oscillates at that frequency. Because the sample cross-correlation sequence uses fewer and fewer samples at larger lags, the sample cross-correlation sequence also oscillates at the same frequency, but the amplitude decays as the lag increases.

Create two sine waves with frequencies of $2\pi/10$ rad/sample. The starting phase of one sine wave is 0, while the starting phase of the other sine wave is $-\pi$ radians. Add $N(0, 0.25^2)$ white noise to the sine wave with the phase lag of π radians. Set the random number generator to the default settings for reproducible results.

```
rng default
```

```
t = 0:99;
x = cos(2*pi*1/10*t);
y = cos(2*pi*1/10*t-pi)+0.25*randn(size(t));
```

Obtain the sample cross-correlation sequence for two periods of the sine wave (10 samples). Plot the cross-correlation sequence and mark the known lag between the two sine waves (5 samples).

```
[xc,lags] = xcorr(y,x,20,'coeff');
```

```
stem(lags(21:end),xc(21:end),'filled')
```

```
hold on
plot([5 5],[-1 1])
```

```
ax = gca;
ax.XTick = 0:5:20;
```
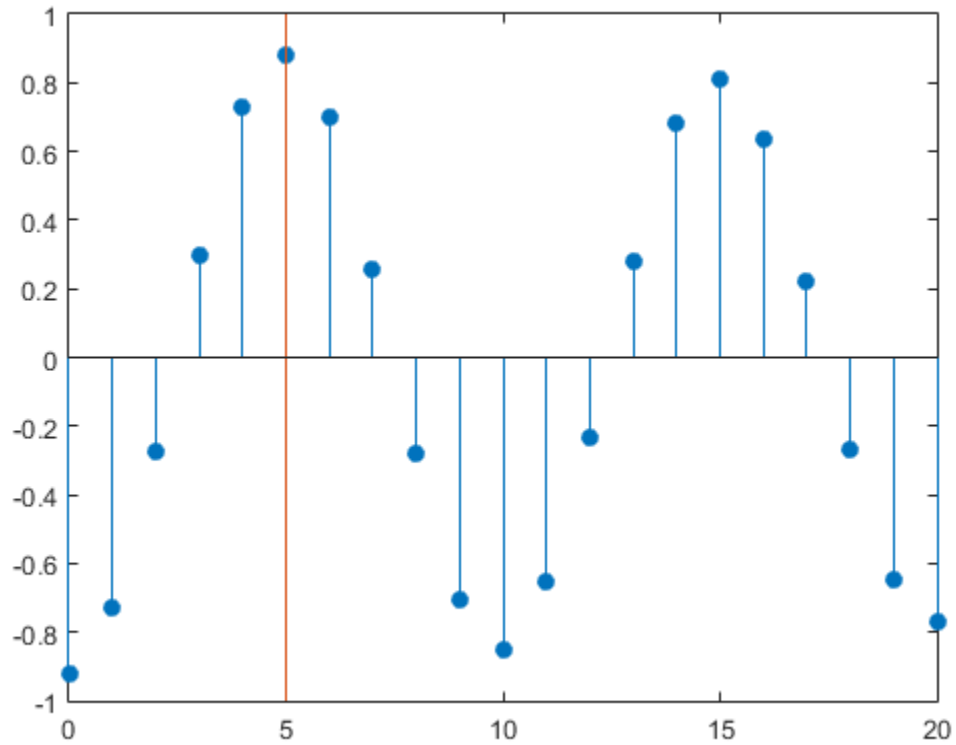
You see that the cross-correlation sequence peaks at lag 5 as expected and oscillates with a period of 10 samples.

Multirate Signal Processing

Downsampling -- Signal Phases

This example shows how to use `downsample` to obtain the *phases* of a signal. Downsampling a signal by M can produce M unique phases. For example, if you have a discrete-time signal, x, with $x(0)$ $x(1)$ $x(2)$ $x(3)$, ..., the M phases of x are $x(nM + k)$ with $k = 0,1, ..., M$-1.

The M signals are referred to as the *polyphase* components of x.

Create a white noise vector and obtain the 3 polyphase components associated with downsampling by 3.

Reset the random number generator to the default settings to produce a repeatable result. Generate a white noise random vector and obtain the 3 polyphase components associated with downsampling by 3.

```
rng default
x = randn(36,1);
x0 = downsample(x,3,0);
x1 = downsample(x,3,1);
x2 = downsample(x,3,2);
```

The polyphase components have length equal to 1/3 the original signal.

Upsample the polyphase components by 3 using `upsample`.

```
y0 = upsample(x0,3,0);
y1 = upsample(x1,3,1);
y2 = upsample(x2,3,2);
```

Plot the result.

```
subplot(4,1,1)
stem(x,'Marker','none')
title('Original Signal')
ylim([-4 4])

subplot(4,1,2)
stem(y0,'Marker','none')
ylabel('Phase 0')
ylim([-4 4])

subplot(4,1,3)
```
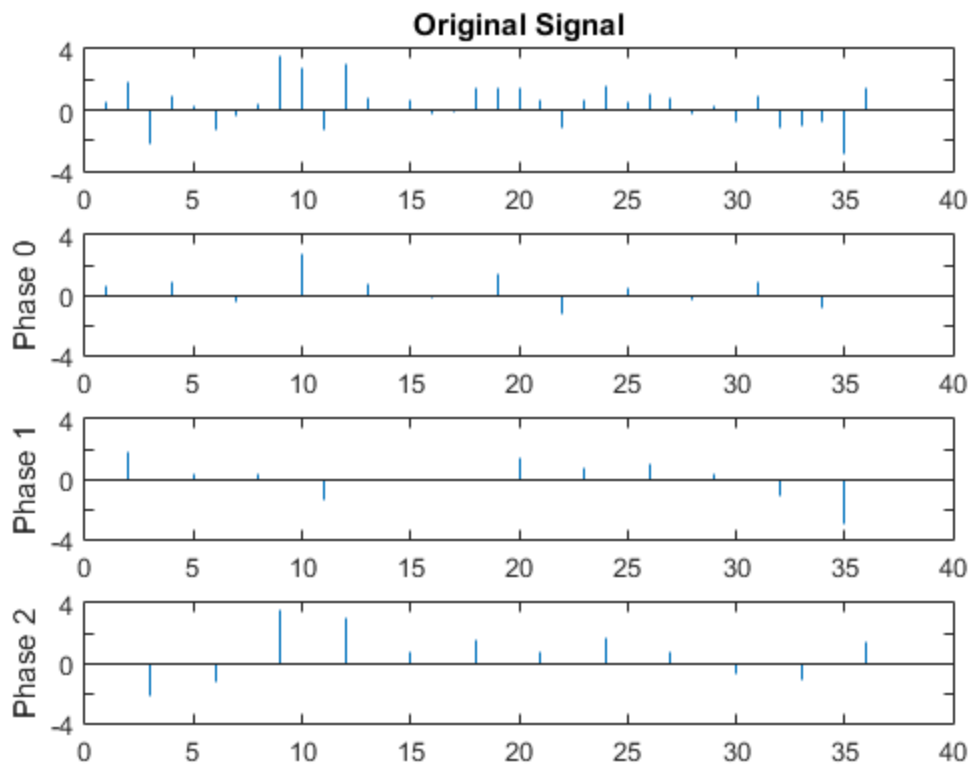
```
stem(y1,'Marker','none')
ylabel('Phase 1')
ylim([-4 4])

subplot(4,1,4)
stem(y2,'Marker','none')
ylabel('Phase 2')
ylim([-4 4])
```


If you sum the upsampled polyphase components you obtain the original signal.

Create a discrete-time sinusoid and obtain the 2 polyphase components associated with downsampling by 2.

Create a discrete-time sine wave with an angular frequency of $\pi/4$ rad/sample. Add a DC offset of 2 to the sine wave to help with visualization of the polyphase components. Downsample the sine wave by 2 to obtain the even and odd polyphase components.

```
n = 0:127;
x = 2+cos(pi/4*n);
x0 = downsample(x,2,0);
x1 = downsample(x,2,1);
```

Upsample the two polyphase components.

```
y0 = upsample(x0,2,0);
y1 = upsample(x1,2,1);
```

Plot the upsampled polyphase components along with the original signal for comparison.

```
subplot(3,1,1)
stem(x,'Marker','none')
ylim([0.5 3.5])
title('Original Signal');

subplot(3,1,2)
stem(y0,'Marker','none')
ylim([0.5 3.5])
ylabel('Phase 0')

subplot(3,1,3)
stem(y1,'Marker','none')
ylim([0.5 3.5])
ylabel('Phase 1')
```
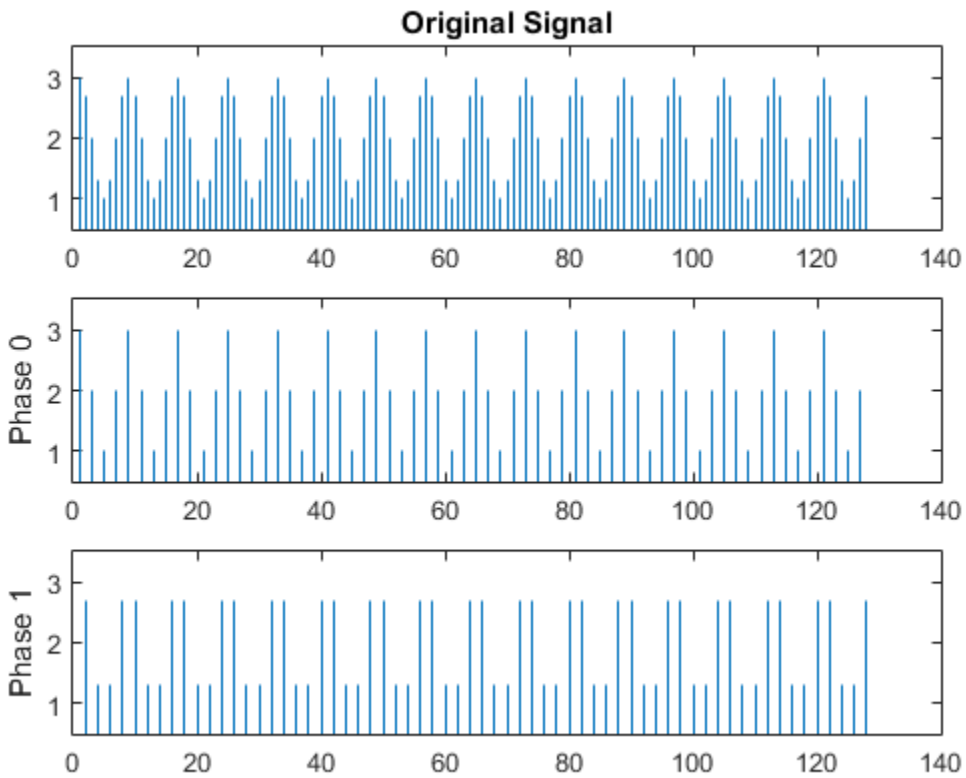
If you sum the two upsampled polyphase components (Phase 0 and Phase 1), you obtain the original sine wave.

Downsampling -- Aliasing

This example shows how to avoid aliasing when downsampling a signal. If a discrete-time signal's baseband spectral support is not limited to an interval of width $2\pi/M$ radians, downsampling by M results in aliasing. Aliasing is the distortion that occurs when overlapping copies of the signal's spectrum are added together. The more the signal's baseband spectral support exceeds $2\pi/M$ radians, the more severe the aliasing. Demonstrate aliasing in a signal downsampled by two. The signal's baseband spectral support exceed π radians in width.

Create a signal with baseband spectral support equal to $3\pi/2$ radians. Use `fir2` to design the signal. Plot the signal's spectrum.

```
F = [0 0.2500 0.5000 0.7500 1.0000];
A = [1.00 0.6667 0.3333 0 0];
Order = 511;
B1 = fir2(Order,F,A);
[Hx,W] = freqz(B1,1,8192,'whole');
Hx = [Hx(4098:end) ; Hx(1:4097)];
omega = -pi+(2*pi/8192):(2*pi)/8192:pi;

plot(omega,abs(Hx))
xlim([-pi pi])
grid
title('Magnitude Spectrum')
xlabel('Radians/Sample')
ylabel('Magnitude')
```
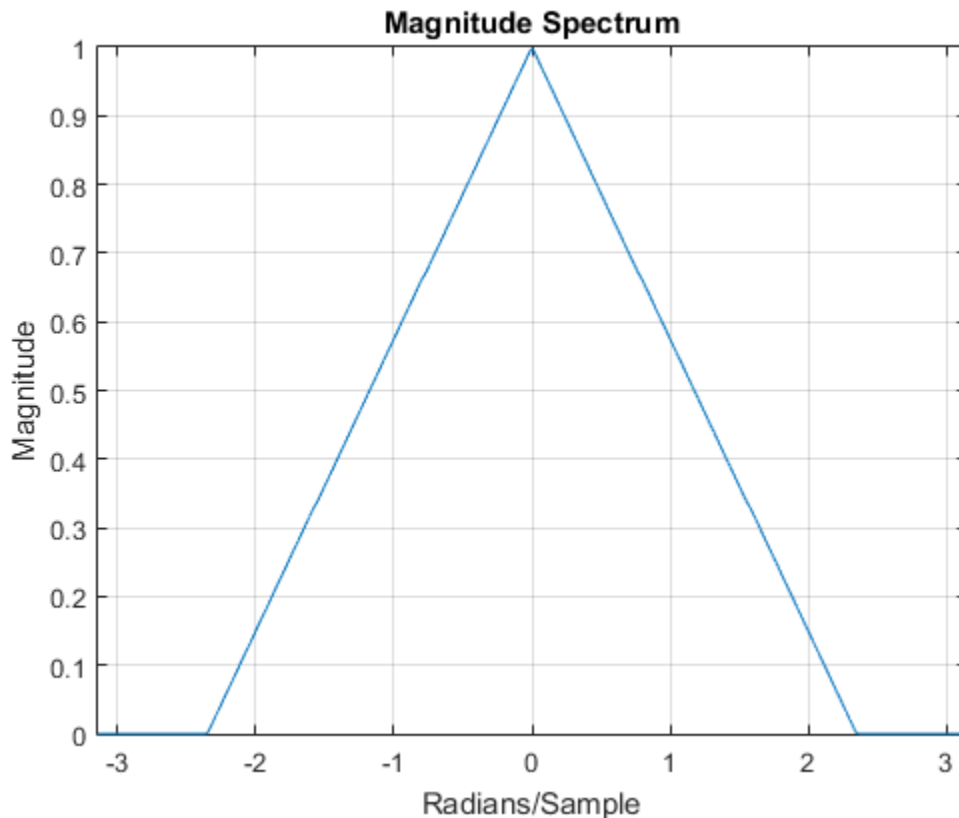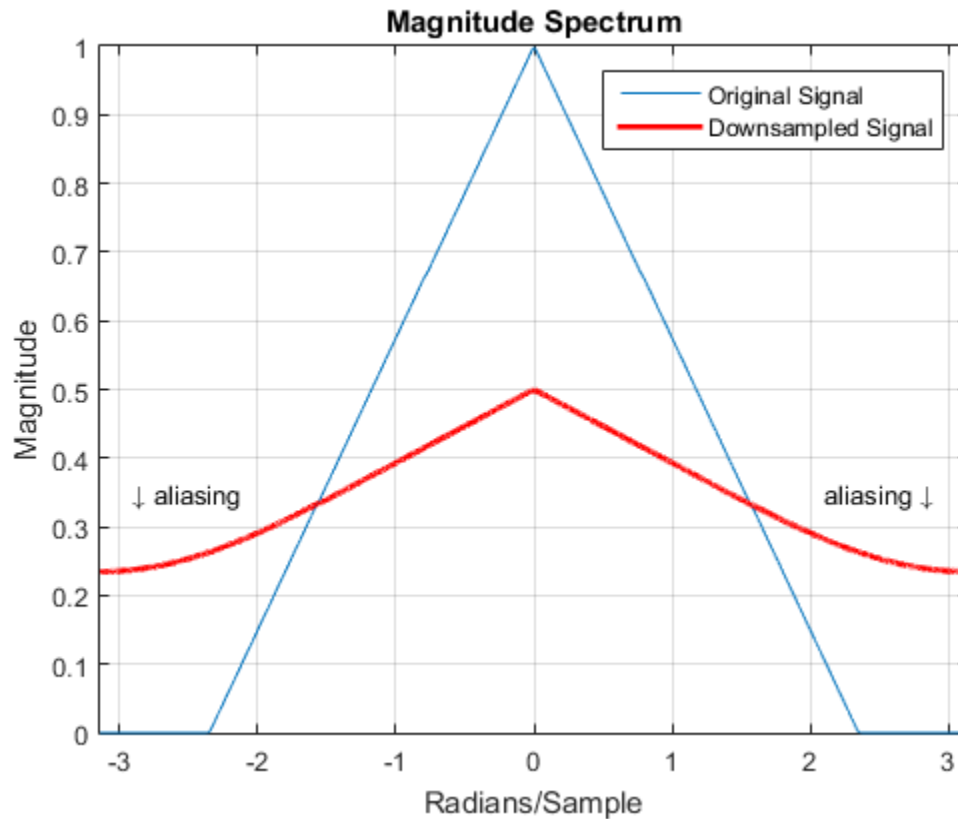
You see that the signal's baseband spectral support exceeds $[-\pi/2, \pi/2]$.

Downsample the signal by a factor of 2 and plot the downsampled signal's spectrum with the spectrum of the original signal.

```
y = downsample(B1,2,0);
[Hy,W] = freqz(y,1,8192,'whole');
Hy = [Hy(4098:end) ; Hy(1:4097)];

hold on
plot(omega,abs(Hy),'r','linewidth',2)
legend('Original Signal','Downsampled Signal')
text(-2.5,0.35,'\downarrow aliasing','HorizontalAlignment','center')
text(2.5,0.35,'aliasing \downarrow','HorizontalAlignment','center')
```

```
hold off
```


Magnitude Spectrum

In addition to an amplitude scaling of the spectrum, the superposition of overlapping spectral replicas causes distortion of the original spectrum for $|\omega| > \pi/2$.

Increase the baseband spectral support of the signal to $[-7\pi/8, 7\pi/8]$ and downsample the signal by 2. Plot the original spectrum along with the spectrum of the downsampled signal.

```
F = [0 0.2500 0.5000 0.7500 7/8 1.0000];
A = [1.00 0.7143 0.4286 0.1429 0 0];
Order = 511;
B2 = fir2(Order,F,A);
```

```
[Hx,W] = freqz(B2,1,8192,'whole');
Hx = [Hx(4098:end) ; Hx(1:4097)];
omega = -pi+(2*pi/8192):(2*pi)/8192:pi;

plot(omega,abs(Hx))
xlim([-pi pi])

y = downsample(B2,2,0);
[Hy,W] = freqz(y,1,8192,'whole');
Hy = [Hy(4098:end) ; Hy(1:4097)];

hold on
plot(omega,abs(Hy),'r','linewidth',2);
grid
legend('Original Signal','Downsampled Signal')
xlabel('Radians/Sample')
ylabel('Magnitude')
hold off
```
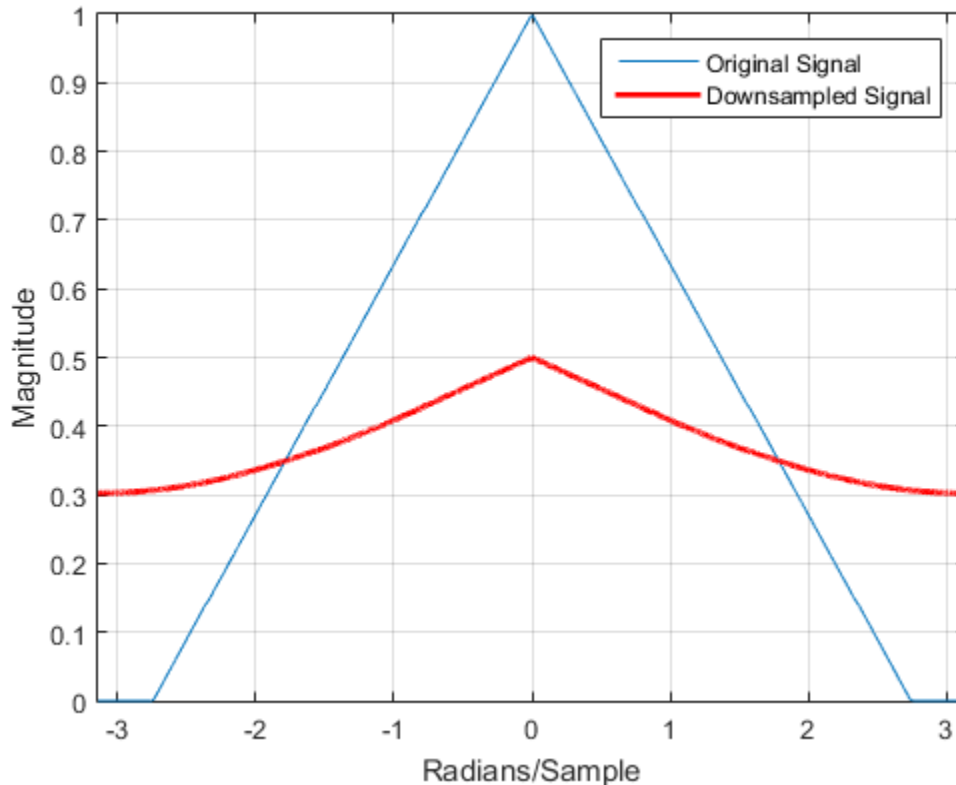
The increased spectral width results in more pronounced aliasing in the spectrum of the downsampled signal because more signal energy is outside $[-\pi/2, \pi/2]$.

Finally, construct a signal with baseband spectral support limited to $[-\pi/2, \pi/2]$. Downsample the signal by a factor of 2 and plot the spectrum of the original and downsampled signals. The downsampled signal is full band, but the shape of the spectrum is preserved because the spectral copies do not overlap. There is no aliasing.

```
F = [0 0.250 0.500 0.7500 1];
A = [1.0000 0.5000 0 0 0];
Order = 511;
B3 = fir2(Order,F,A);
[Hx,W] = freqz(B3,1,8192,'whole');
```

```
Hx = [Hx(4098:end) ; Hx(1:4097)];
omega = -pi+(2*pi/8192):(2*pi)/8192:pi;

plot(omega,abs(Hx))
xlim([-pi pi])

y = downsample(B3,2,0);
[Hy,W] = freqz(y,1,8192,'whole');
Hy = [Hy(4098:end) ; Hy(1:4097)];

plot(omega,abs(Hx))
hold on
plot(omega,abs(Hy),'r','linewidth',2)
grid
legend('Original Signal','Downsampled Signal')
xlabel('Radians/Sample')
ylabel('Magnitude')
hold off
```
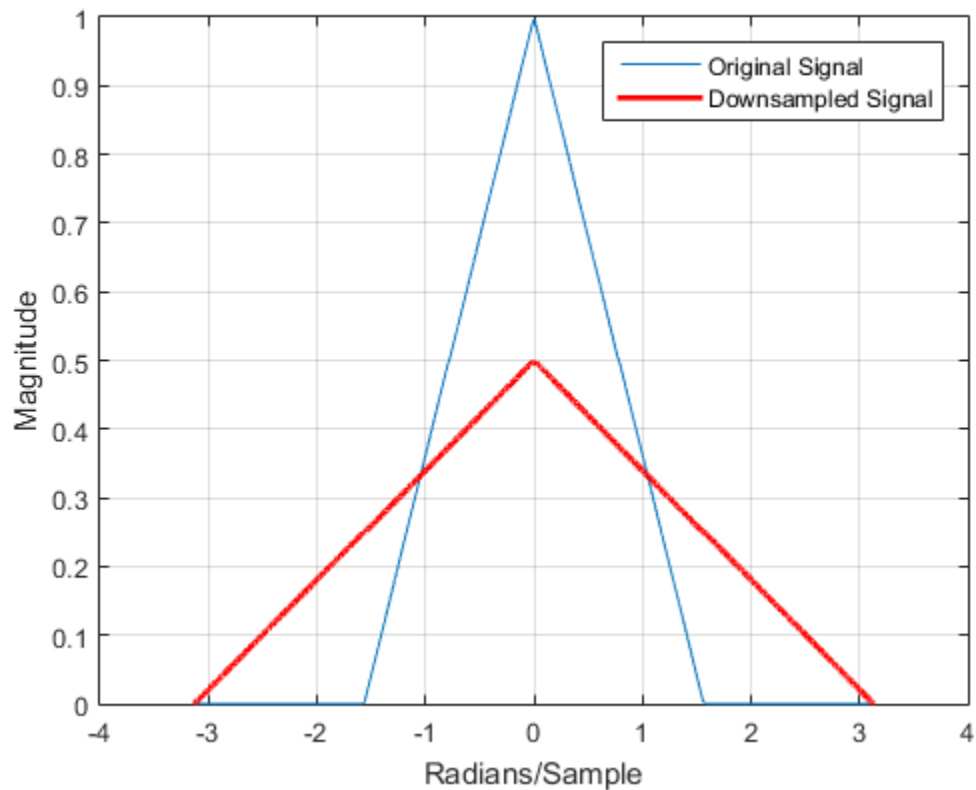
You see in the preceding figure that the shape of the spectrum is preserved. The spectrum of the downsampled signal is a stretched and scaled version of the original signal's spectrum, but there is no aliasing.

Filtering Before Downsampling

This example shows how to filter before downsampling to mitigate the distortion caused by aliasing. You can use `decimate` or `resample` to filter and downsample with one function. Alternatively, you can lowpass filter your data and then use `downsample`. Create a signal with baseband spectral support greater than π radians. Use `decimate` to filter the signal with a 10th-order Chebyshev type I lowpass filter prior to downsampling.

Create the signal and plot the magnitude spectrum.

```
F = [0 0.2500 0.5000 0.7500 1.0000];
A = [1.00 0.6667 0.3333 0 0];
Order = 511;
B = fir2(Order,F,A);
[Hx,W] = freqz(B,1,8192,'whole');
Hx = [Hx(4098:end) ; Hx(1:4097)];
omega = -pi+(2*pi/8192):(2*pi)/8192:pi;

plot(omega,abs(Hx))
xlim([-pi pi])
grid
title('Magnitude Spectrum')
xlabel('Radians/Sample')
ylabel('Magnitude')
```
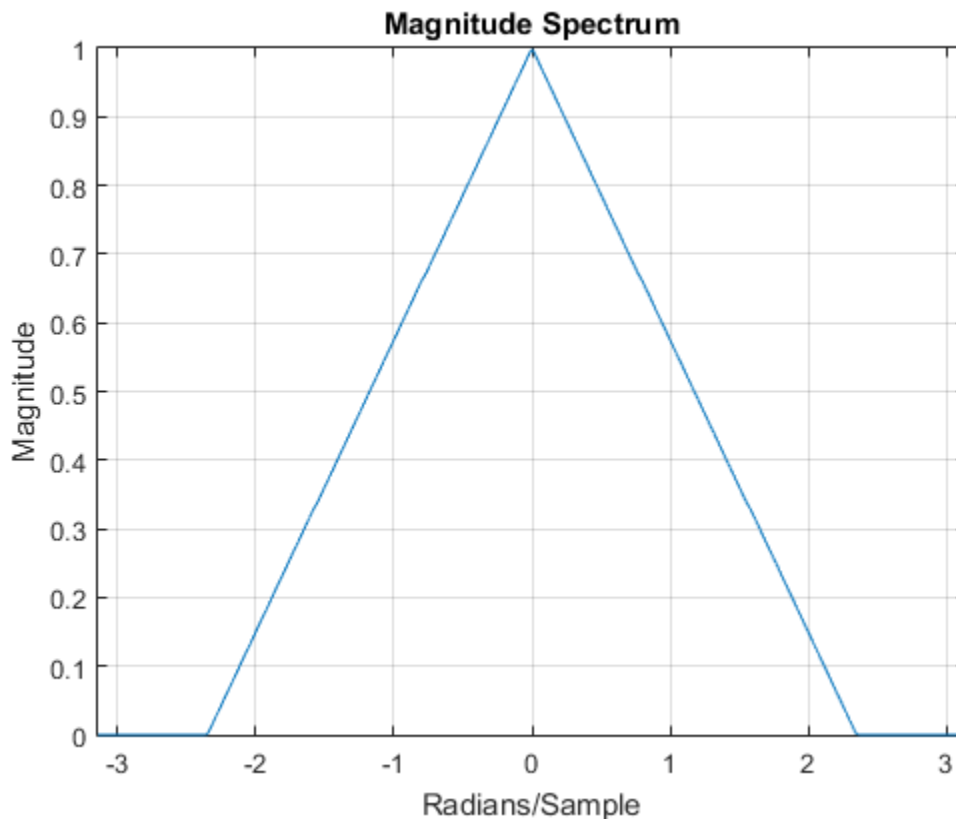
Filter the signal with a 10th-order type I Chebyshev lowpass filter and downsample by 2. Plot the magnitude spectra of the original signal along with the filtered and downsampled signal.

```
y = decimate(B,2,10);
[Hy,W] = freqz(y,1,8192,'whole');
Hy = [Hy(4098:end) ; Hy(1:4097)];

hold on
plot(omega,abs(Hy),'r','linewidth',2)
legend('Original Signal','Downsampled Signal')
```
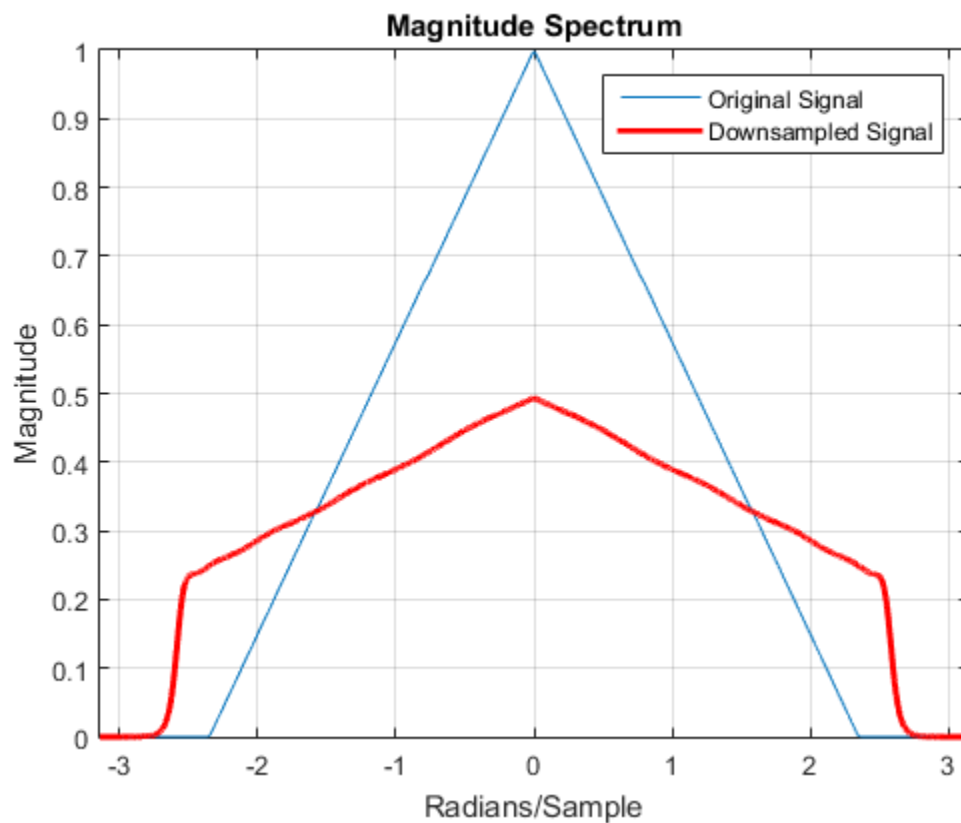
The lowpass filter reduces the amount of aliasing distortion outside the interval $[-\pi/2, \pi/2]$.

Upsampling -- Imaging Artifacts

This example shows how to upsample a signal and how upsampling can result in images. Upsampling a signal contracts the spectrum. For example, upsampling a signal by 2 results in a contraction of the spectrum by a factor of 2. Because the spectrum of a discrete-time signal is 2π-periodic, contraction can cause replicas of the spectrum normally outside of the baseband to appear inside the interval $[-\pi, \pi]$.

Create a discrete-time signal whose baseband spectral support is $[-\pi, \pi]$. Plot the magnitude spectrum.

```
F = [0 0.250 0.500 0.7500 1];
A = [1.0000 0.5000 0 0 0];
Order = 511;
B = fir2(Order,F,A);
[Hx,W] = freqz(B,1,8192,'whole');
Hx = [Hx(4098:end) ; Hx(1:4097)];
omega = -pi+(2*pi/8192):(2*pi)/8192:pi;

plot(omega,abs(Hx))
```
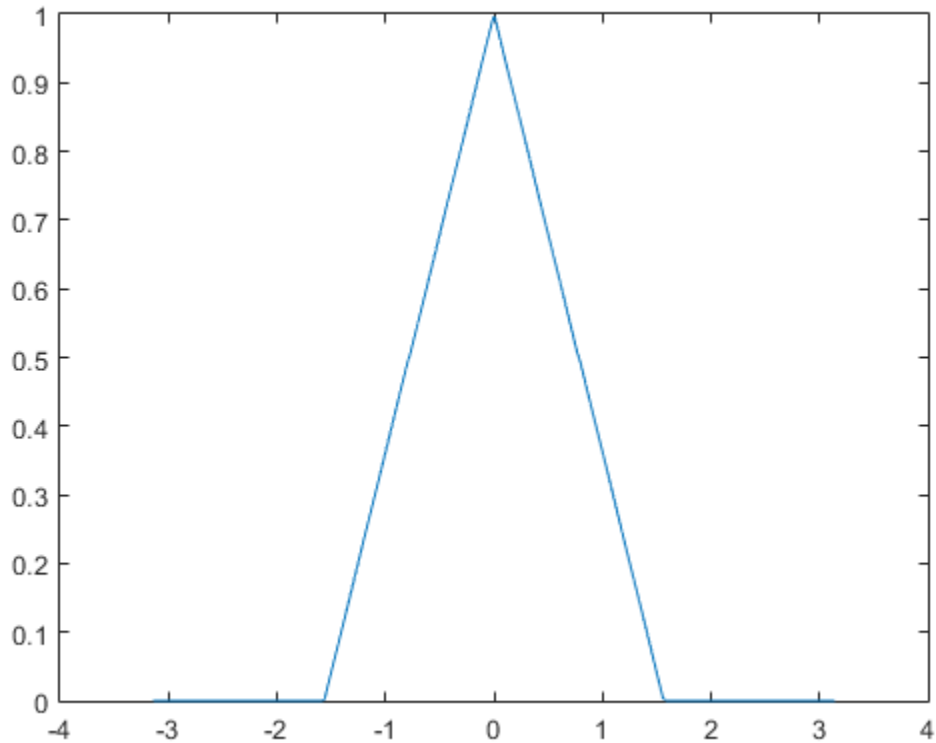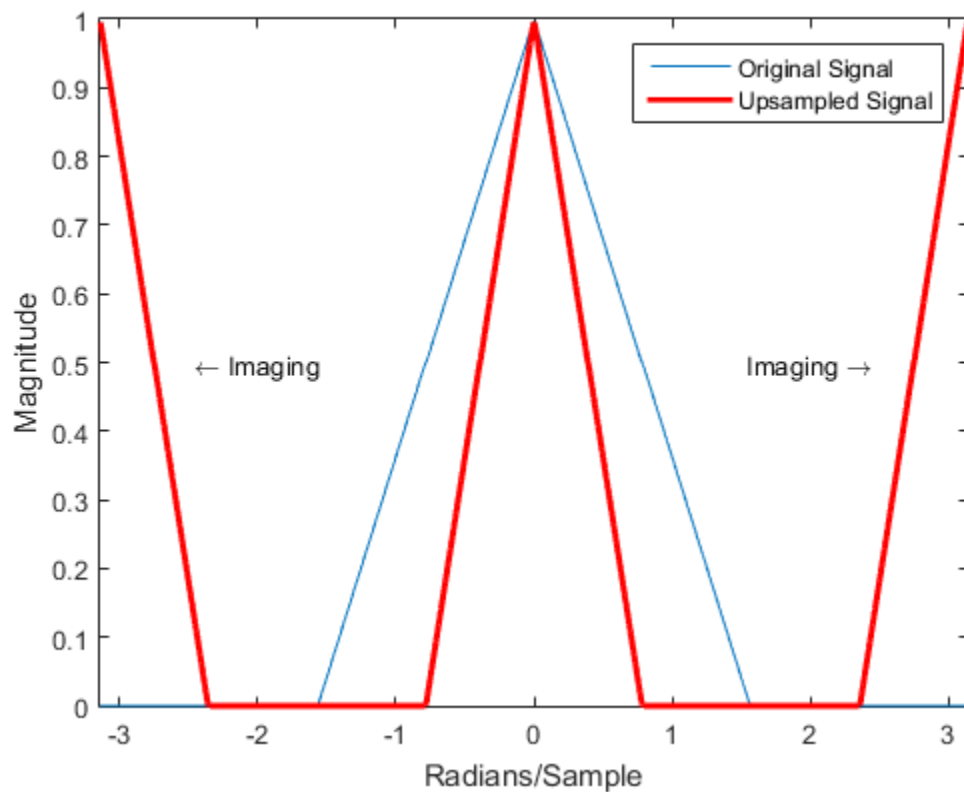
Upsample the signal by 2. Plot the spectrum of the upsampled signal.

```
y = upsample(B,2);
[Hy,W] = freqz(y,1,8192,'whole');
Hy = [Hy(4098:end) ; Hy(1:4097)];

hold on
plot(omega,abs(Hy),'r','linewidth',2)
xlim([-pi pi])
legend('Original Signal','Upsampled Signal')
xlabel('Radians/Sample')
ylabel('Magnitude')
text(-2,0.5,'\leftarrow Imaging','HorizontalAlignment','center')
text(2,0.5,'Imaging \rightarrow','HorizontalAlignment','center')
```

11-17

You can see in the preceding figure that the contraction of the spectrum has drawn subsequent periods of the spectrum into the interval $[-\pi, \pi]$.

Filtering After Upsampling -- Interpolation

This example shows how to upsample a signal and apply a lowpass interpolation filter with `interp`. Upsampling by *L* inserts *L* - 1 zeros between every element of the original signal. Upsampling can create imaging artifacts. Lowpass filtering following upsampling can remove these imaging artifacts. In the time domain, lowpass filtering interpolates the zeros inserted by upsampling.

Create a discrete-time signal whose baseband spectral support is $[-\pi/2, \pi/2]$. Plot the magnitude spectrum.

```
F = [0 0.250 0.500 0.7500 1];
A = [1.0000 0.5000 0 0 0];
Order = 511;
B = fir2(Order,F,A);
[Hx,W] = freqz(B,1,8192,'whole');
Hx = [Hx(4098:end) ; Hx(1:4097)];
omega = -pi+(2*pi/8192):(2*pi)/8192:pi;

plot(omega,abs(Hx))
xlim([-pi pi])
xlabel('Radians/Sample')
ylabel('Magnitude')
```
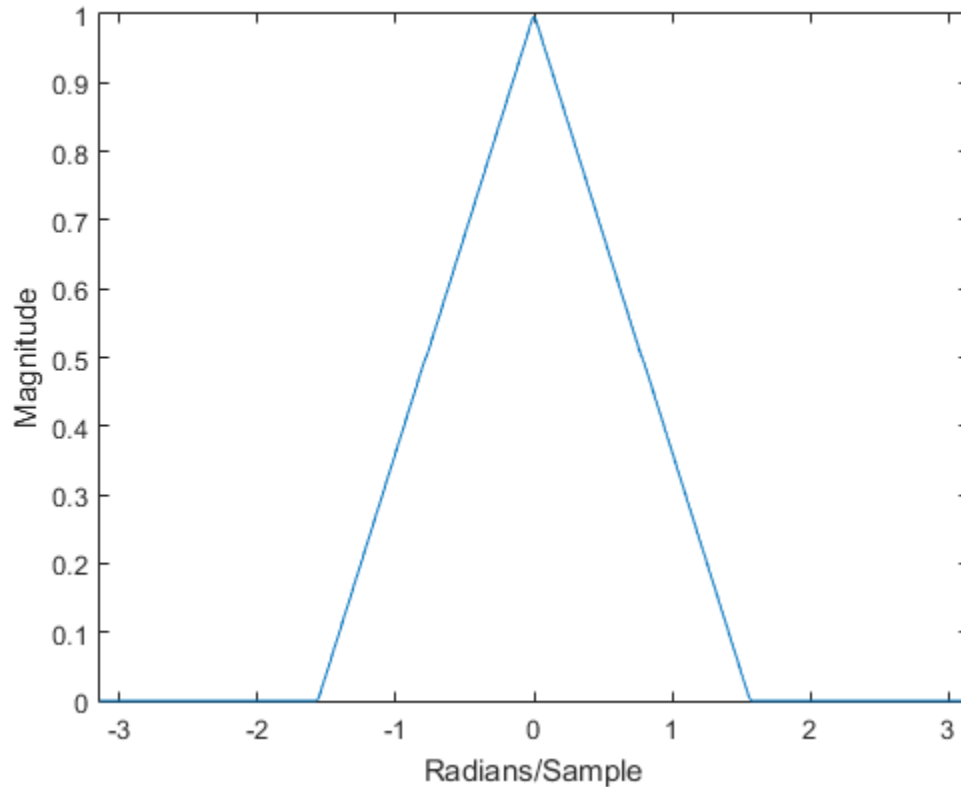
Upsample the signal and apply a lowpass filter to remove the imaging artifacts. Plot the magnitude spectrum.

```
y = interp(B,2);
[Hy,W] = freqz(y,1,8192,'whole');
Hy = [Hy(4098:end) ; Hy(1:4097)];

hold on
plot(omega,abs(Hy),'r','linewidth',2)
legend('Original Signal','Upsampled Signal')
```
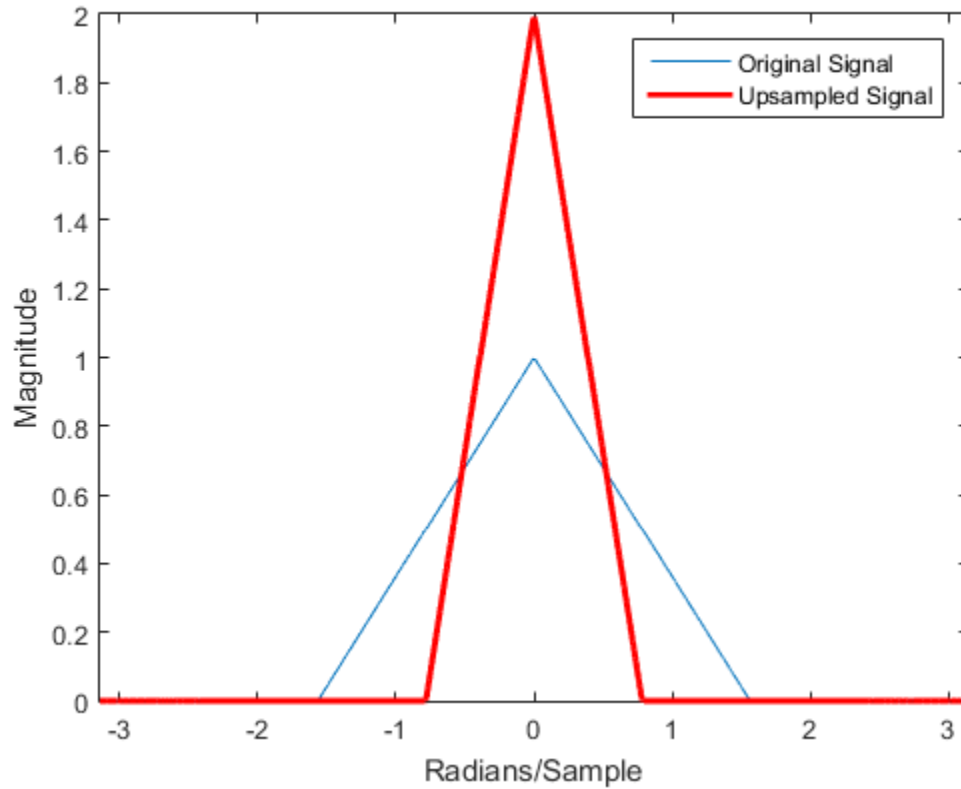
Upsampling still contracts the spectrum, but the imaging artifacts are removed by the lowpass filter.

Simulate a Sample-and-Hold System

This example shows several ways to simulate the output of a sample-and-hold system by upsampling and filtering a signal.

Construct a sinusoidal signal. Specify a sample rate such that 16 samples correspond to exactly one signal period. Draw a stem plot of the signal. Overlay a stairstep graph for sample-and-hold visualization.

```
fs = 16;
t = 0:1/fs:1-1/fs;

x = .9*sin(2*pi*t);

stem(t,x)
hold on
stairs(t,x)
hold off
```
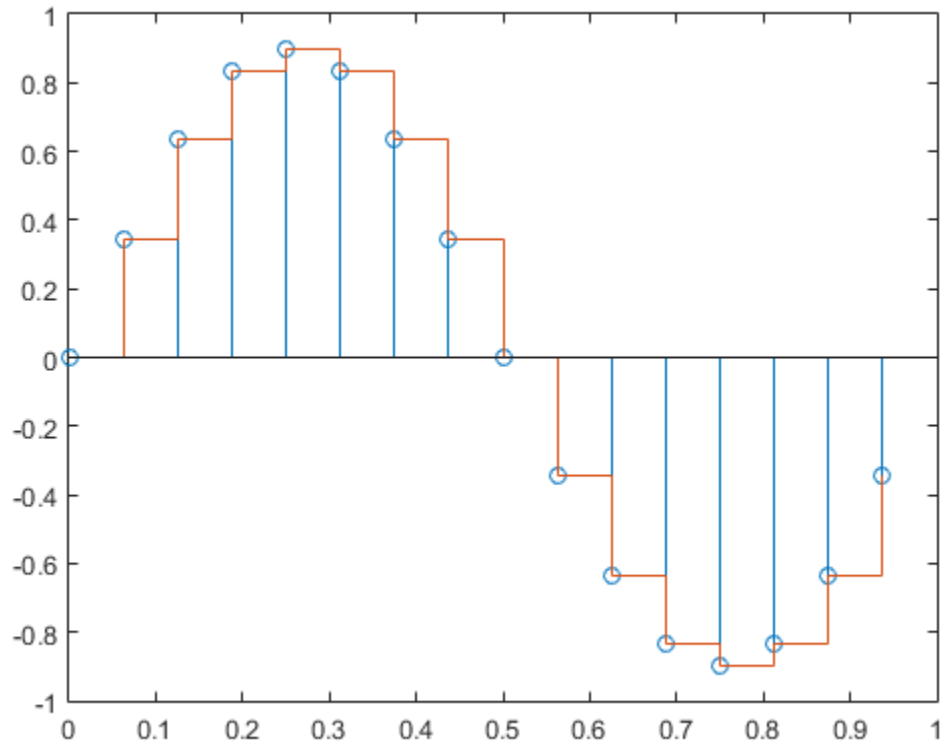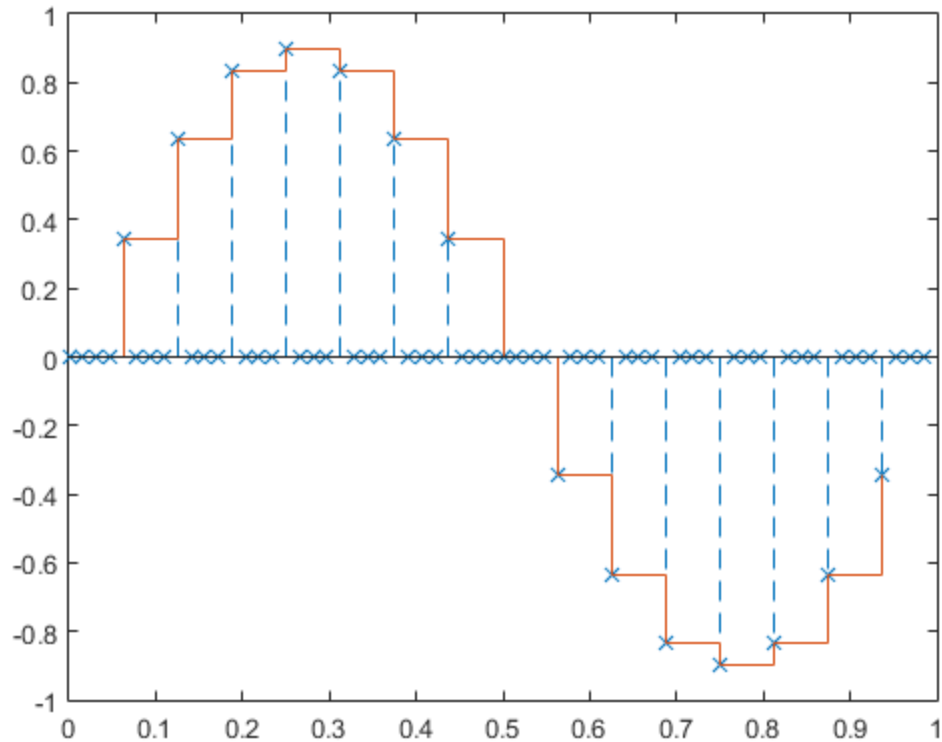
Upsample the signal by a factor of four. Plot the result alongside the original signal. `upsample` increases the sample rate of the signal by adding zeros between the existing samples.

```
ups = 4;

fu = fs*ups;
tu = 0:1/fu:1-1/fu;

y = upsample(x,ups);

stem(tu,y,'--x')

hold on
```
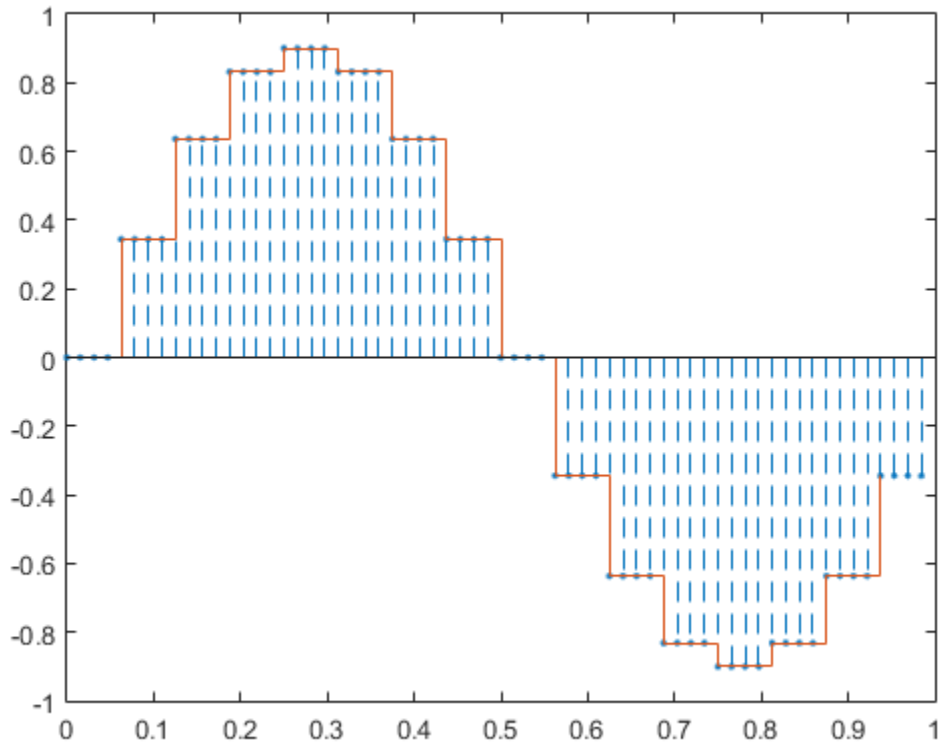
```
stairs(t,x)
hold off
```


Filter with a moving-average FIR filter to fill in the zeros with sample-and-hold values.

```
h = ones(ups,1);

z = filter(h,1,y);

stem(tu,z,'--.')
hold on
stairs(t,x)
hold off
```
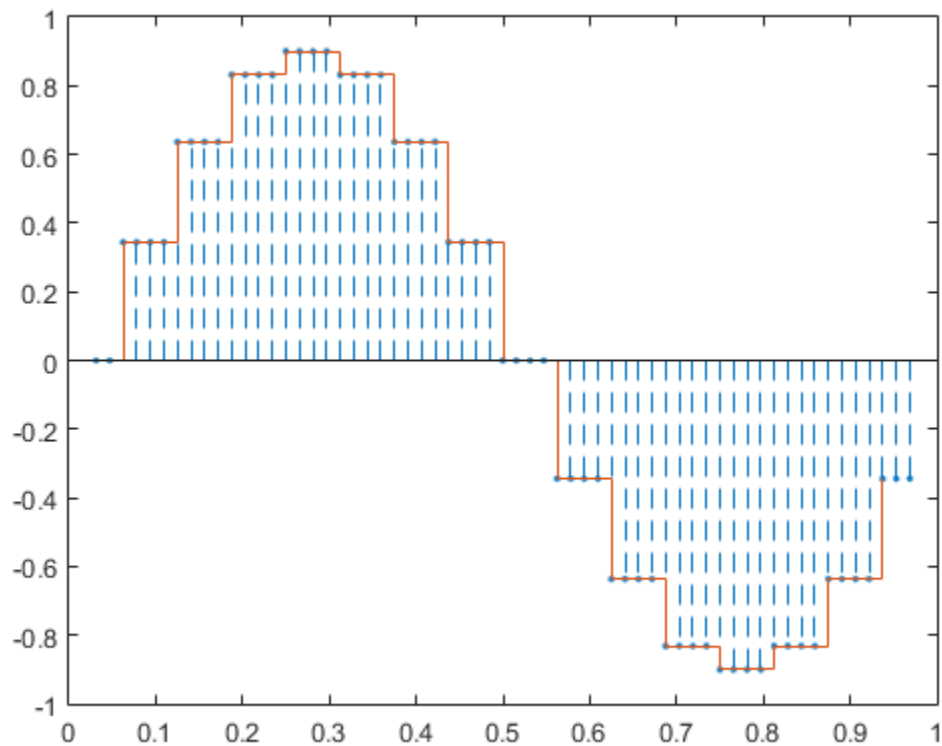
You can obtain the same behavior using the MATLAB® function `interp1` with nearest-neighbor interpolation. In that case, you must shift the origin to line up the sequence.
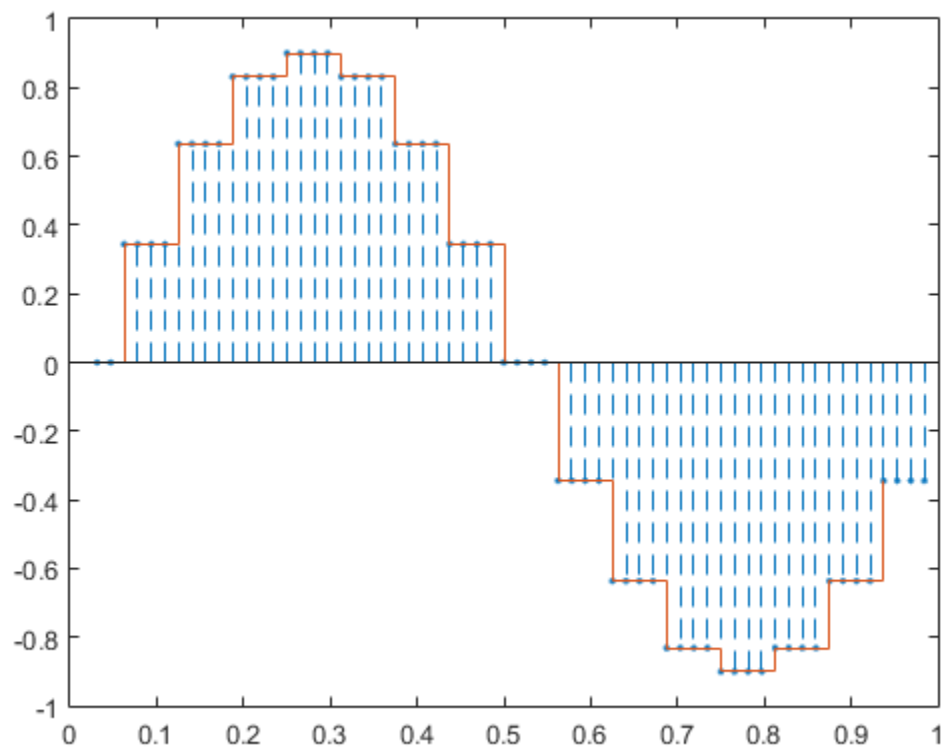
```
zi = interp1(t,x,tu,'nearest');

dl = floor(ups/2);

stem(tu(1+dl:end),zi(1:end-dl),'--.')
hold on
stairs(t,x)
hold off
```

11-25

The function `resample` produces the same result when you set the last input argument to zero.

```
q = resample(x,ups,1,0);

stem(tu(1+dl:end),q(1:end-dl),'--.')
hold on
stairs(t,x)
hold off
```

Changing Signal Sampling Rate

This example shows how to change the sampling rate of a signal. The example has two parts. Part one changes the sampling rate of a sinusoidal input from 44.1 kHz to 48 kHz. This workflow is common in audio processing. The sampling rate used on compact discs is 44.1 kHz, while the sampling rate used on digital audio tape is 48 kHz. Part two changes the sampling rate of a recorded speech sample from 7418 Hz to 8192 Hz.

Create an input signal consisting of a sum of sine waves sampled at 44.1 kHz. The sine waves have frequencies of 2, 4, and 8 kHz.

```
Fs = 44.1e3;
t = 0:1/Fs:1-(1/Fs);
x = cos(2*pi*2000*t)+1/2*sin(2*pi*4000*(t-pi/4))+1/4*cos(2*pi*8000*t);
```

To change the sampling rate from 44.1 to 48 kHz, you have to determine a rational number (ratio of integers), *P/Q*, such that *P/Q* times the original sampling rate, 44100, is equal to 48000 within some specified tolerance.

To determine these factors, use `rat`. Input the ratio of the new sampling rate, 48000, to the original sampling rate, 44100.

```
[P,Q] = rat(48e3/Fs);
abs(P/Q*Fs-48000)
```

You see that `P/Q*Fs` only differs from the desired sampling rate, 48000, on the order of 10^{-12}.

Use the numerator and denominator factors obtained with `rat` as inputs to `resample` to output a waveform sampled at 48 kHz.

```
xnew = resample(x,P,Q);
```

If your computer can play audio, you can play the two waveforms. Set the volume to a comfortable level before you play the signals. Execute the `play` commands separately so that you can hear the signal with the two different sampling rates.

```
P44_1 = audioplayer(x,44100);
P48 = audioplayer(xnew,48000);
play(P44_1)
play(P48)
```

Change the sampling rate of a speech sample from 7418 Hz to 8192 Hz. The speech signal is a recording of a speaker saying "MATLAB".

Load the speech sample.

```
load mtlb
```

Loading the file `mtlb.mat` brings the speech signal, `mtlb`, and the sampling rate, `Fs`, into the MATLAB workspace.

Determine a rational approximation to the ratio of the new sample rate, 8192, to the original sample rate. Use `rat` to determine the approximation.

```
[P,Q] = rat(8192/Fs);
```

Resample the speech sample at the new sampling rate.

```
mtlb_new = resample(mtlb,P,Q);
```

If your computer has audio output capability, you can play the two waveforms at their respective sampling rates for comparison. Set the volume on your computer to a comfortable listening level before playing the sounds. Execute the `play` commands separately to compare the speech samples at the different sampling rates.

```
Pmtlb = audioplayer(mtlb,Fs);
Pmtlb_new = audioplayer(mtlb_new,8192);
play(Pmtlb)
play(Pmtlb_new)
```

12

Spectral Analysis

Power Spectral Density Estimates Using FFT

This example shows how to obtain nonparametric power spectral density (PSD) estimates equivalent to the periodogram using `fft`. The examples show you how to properly scale the output of `fft` for even-length inputs, for normalized frequency and hertz, and for one- and two-sided PSD estimates. Obtain the periodogram for an even-length signal sampled at 1 kHz using both `fft` and `periodogram`. Compare the results.
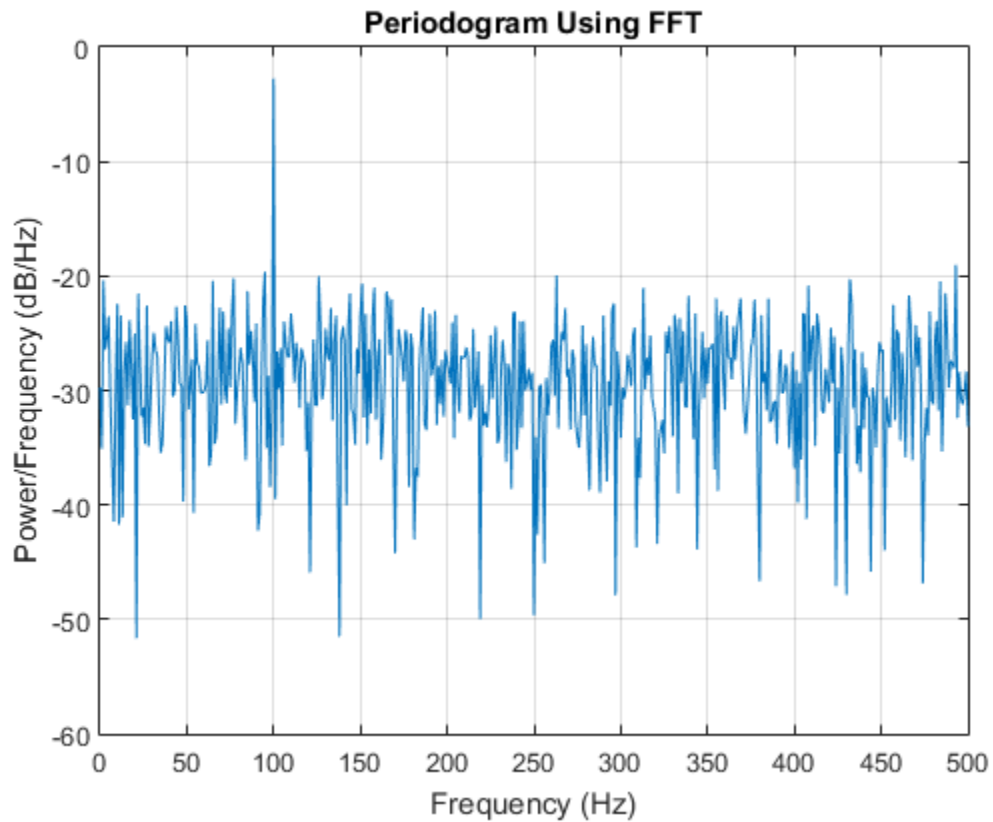
Create a signal consisting of a 100 Hz sine wave in $N(0,1)$ additive noise. The sampling frequency is 1 kHz. The signal length is 1000 samples. Use the default settings of the random number generator for reproducible results.

```
rng default
Fs = 1000;
t = 0:1/Fs:1-1/Fs;
x = cos(2*pi*100*t) + randn(size(t));
```

Obtain the periodogram using `fft`. The signal is real-valued and has even length. Because the signal is real-valued, you only need power estimates for the positive or negative frequencies. In order to conserve the total power, multiply all frequencies that occur in both sets -- the positive and negative frequencies -- by a factor of 2. Zero frequency (DC) and the Nyquist frequency do not occur twice. Plot the result.

```
N = length(x);
xdft = fft(x);
xdft = xdft(1:N/2+1);
psdx = (1/(Fs*N)) * abs(xdft).^2;
psdx(2:end-1) = 2*psdx(2:end-1);
freq = 0:Fs/length(x):Fs/2;

plot(freq,10*log10(psdx))
grid on
title('Periodogram Using FFT')
xlabel('Frequency (Hz)')
ylabel('Power/Frequency (dB/Hz)')
```
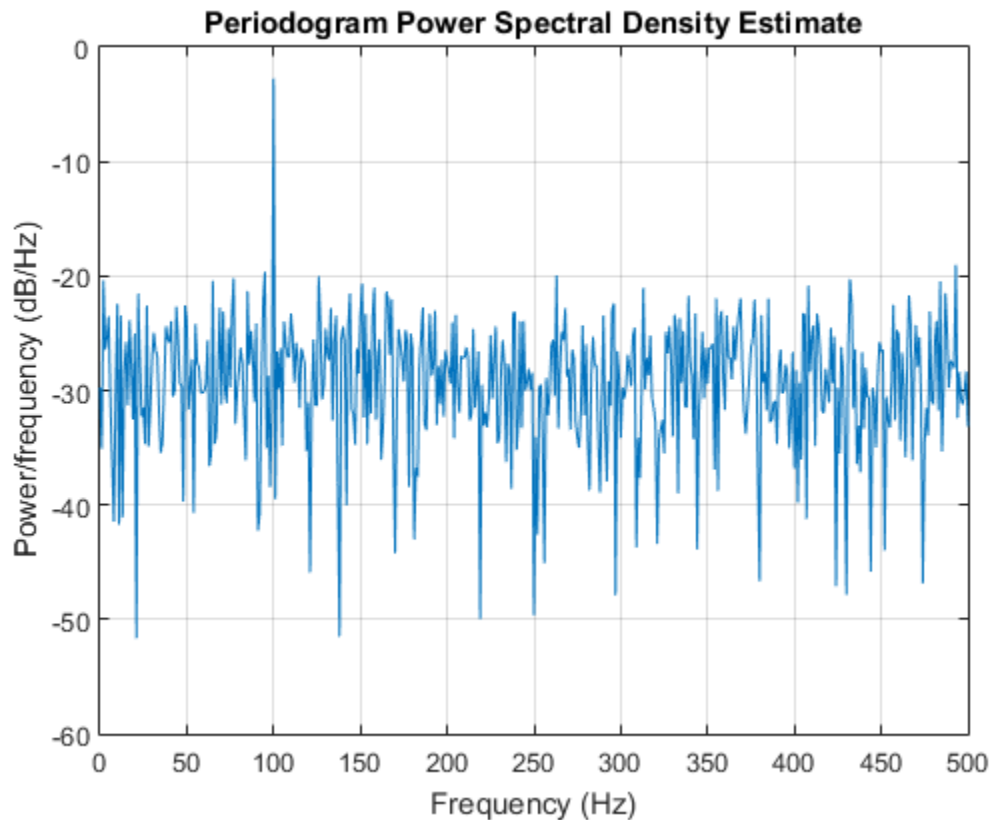
Compute and plot the periodogram using `periodogram`. Show that the two results are identical.

```
periodogram(x,rectwin(length(x)),length(x),Fs)

mxerr = max(psdx'-periodogram(x,rectwin(length(x)),length(x),Fs))


mxerr =

   3.4694e-18
```

Periodogram Power Spectral Density Estimate

Use `fft` to produce a periodogram for an input using normalized frequency. Create a signal consisting of a sine wave in $N(0,1)$ additive noise. The sine wave has an angular frequency of $\pi/4$ rad/sample. Use the default settings of the random number generator for reproducible results.

```
rng default
n = 0:999;
x = cos(pi/4*n) + randn(size(n));
```

Obtain the periodogram using `fft`. The signal is real-valued and has even length. Because the signal is real-valued, you only need power estimates for the positive or negative frequencies. In order to conserve the total power, multiply all frequencies

that occur in both sets -- the positive and negative frequencies -- by a factor of 2. Zero frequency (DC) and the Nyquist frequency do not occur twice. Plot the result.

```
N = length(x);
xdft = fft(x);
xdft = xdft(1:N/2+1);
psdx = (1/(2*pi*N)) * abs(xdft).^2;
psdx(2:end-1) = 2*psdx(2:end-1);
freq = 0:(2*pi)/N:pi;

plot(freq/pi,10*log10(psdx))
grid on
title('Periodogram Using FFT')
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Power/Frequency (dB/rad/sample)')
```
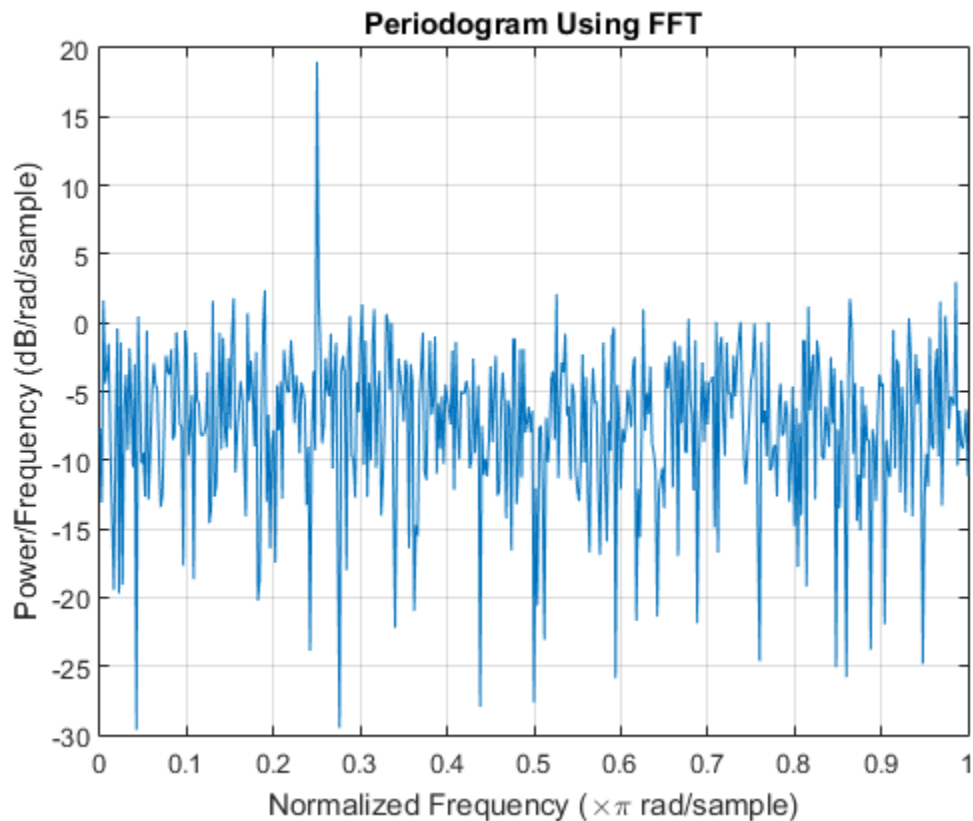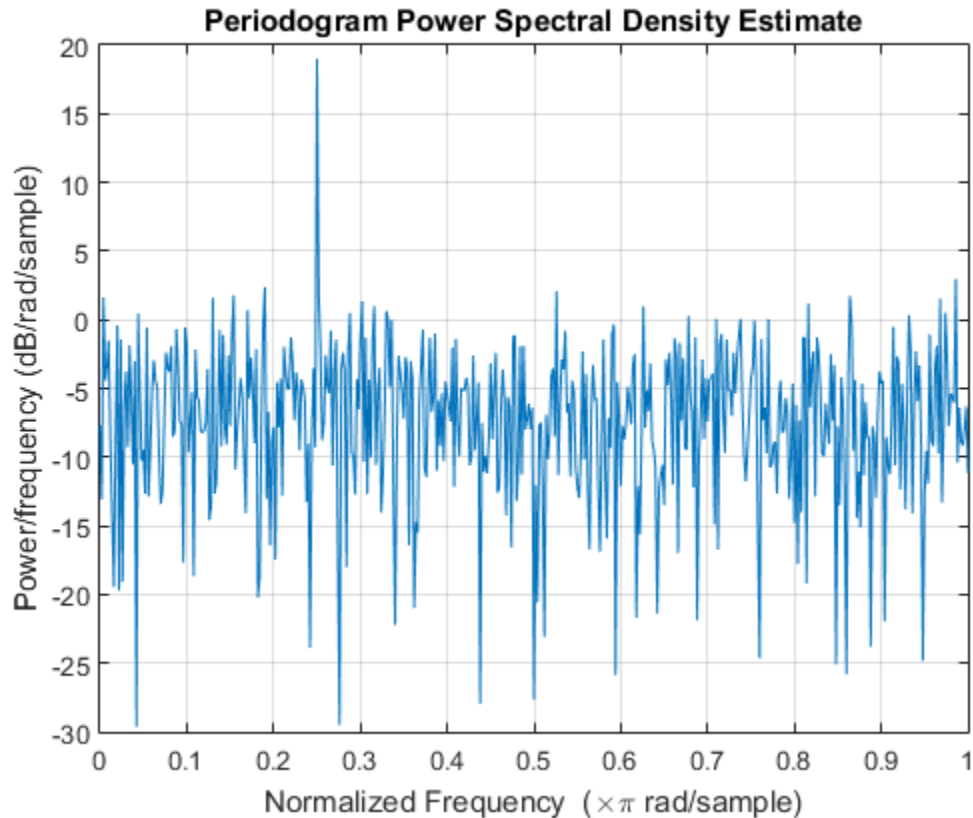
Compute and plot the periodogram using `periodogram`. Show that the two results are identical.

```
periodogram(x,rectwin(length(x)),length(x))

mxerr = max(psdx'-periodogram(x,rectwin(length(x)),length(x)))


mxerr =

   1.4211e-14
```

Periodogram Power Spectral Density Estimate

Use fft to produce a periodogram for a complex-valued input with normalized frequency.

The signal is a complex exponential with an angular frequency of $\pi/4$ rad/sample in complex-valued N(0,1) noise. Set the random number generator to the default settings for reproducible results.

```
rng default
n = 0:999;
x = exp(1j*pi/4*n) + [1 1j]*randn(2,length(n))/sqrt(2);
```

Use `fft` to obtain the periodogram. Because the input is complex-valued, obtain the periodogram from $(-\pi, \pi]$ rad/sample. Plot the result.

```
N = length(x);
```

```
xdft = fft(x);
psdx = (1/(2*pi*N)) * abs(xdft).^2;
freq = 0:(2*pi)/N:2*pi-(2*pi)/N;

plot(freq/pi,10*log10(psdx))
grid on
title('Periodogram Using FFT')
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Power/Frequency (dB/rad/sample)')
```
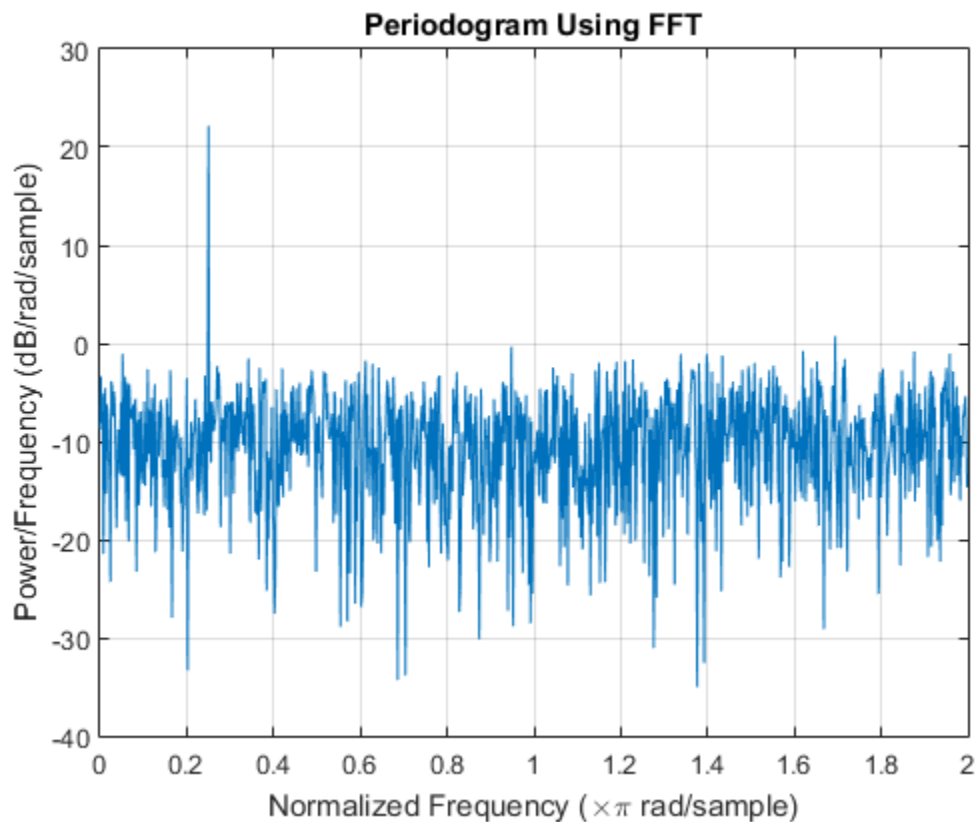

Use `periodogram` to obtain and plot the periodogram. Compare the PSD estimates.
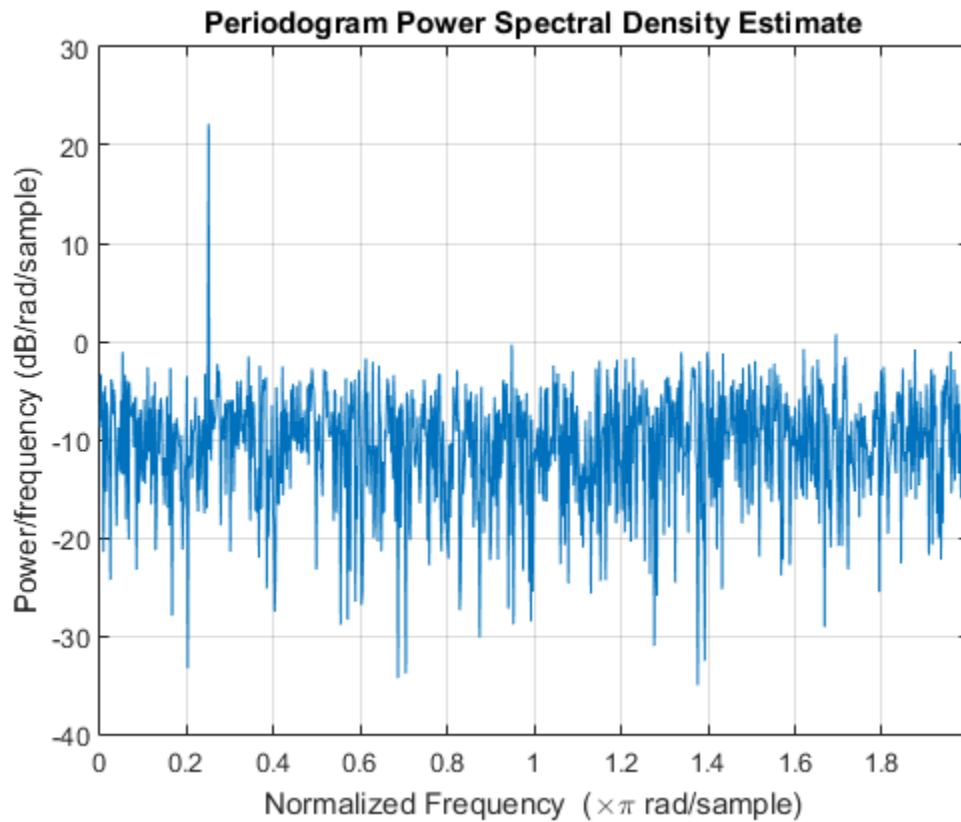
```
periodogram(x,rectwin(length(x)),length(x),'twosided')
```

```
mxerr = max(psdx'-periodogram(x,rectwin(length(x)),length(x),'twosided'))
```

```
mxerr =

   2.8422e-14
```


Periodogram Power Spectral Density Estimate

Bias and Variability in the Periodogram

This example shows how to reduce bias and variability in the periodogram. Using a window can reduce the bias in the periodogram, and using windows with averaging can reduce variability.

Use wide-sense stationary autoregressive (AR) processes to show the effects of bias and variability in the periodogram. AR processes present a convenient model because their PSDs have closed-form expressions. Create an AR(2) model of the following form:

$$y(n) - 0.75y(n-1) + 0.5y(n-2) = \varepsilon(n),$$

where $\varepsilon(n)$ is a zero mean white noise sequence with some specified variance. In this example, assume the variance and the sampling period to be 1. To simulate the preceding AR(2) process, create an all-pole (IIR) filter. View the filter's magnitude response.

```
B2 = 1;
A2 = [1 -0.75 0.5];
fvtool(B2,A2)
```

Magnitude Response (dB)
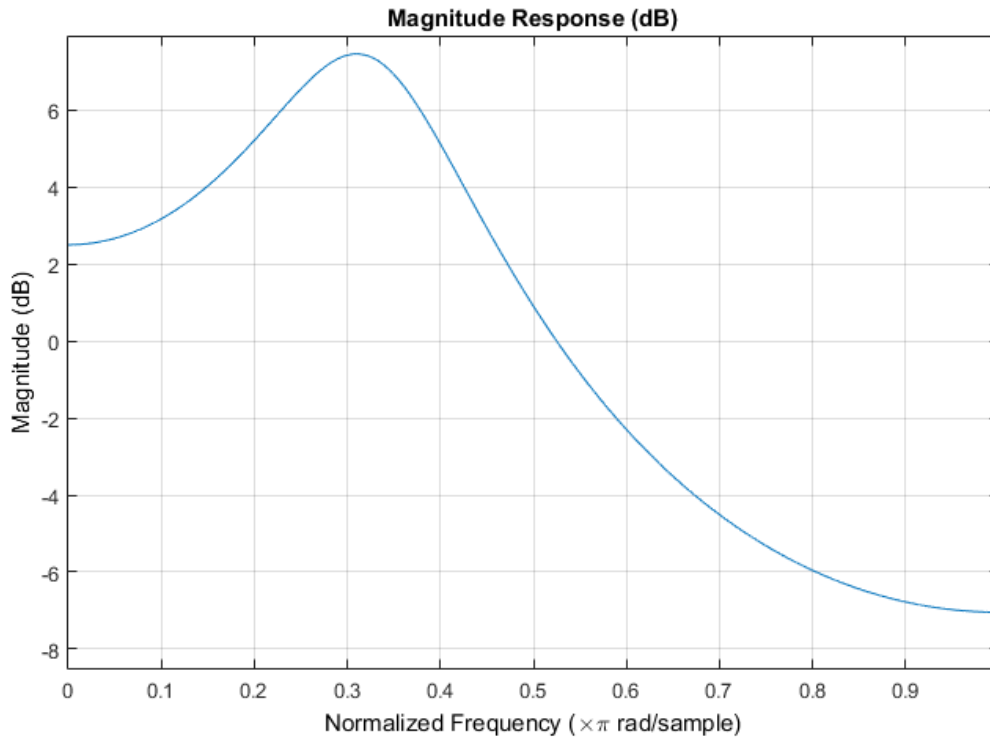
This process is bandpass. The dynamic range of the PSD is approximately 14.5 dB, as you can determine with the following code.
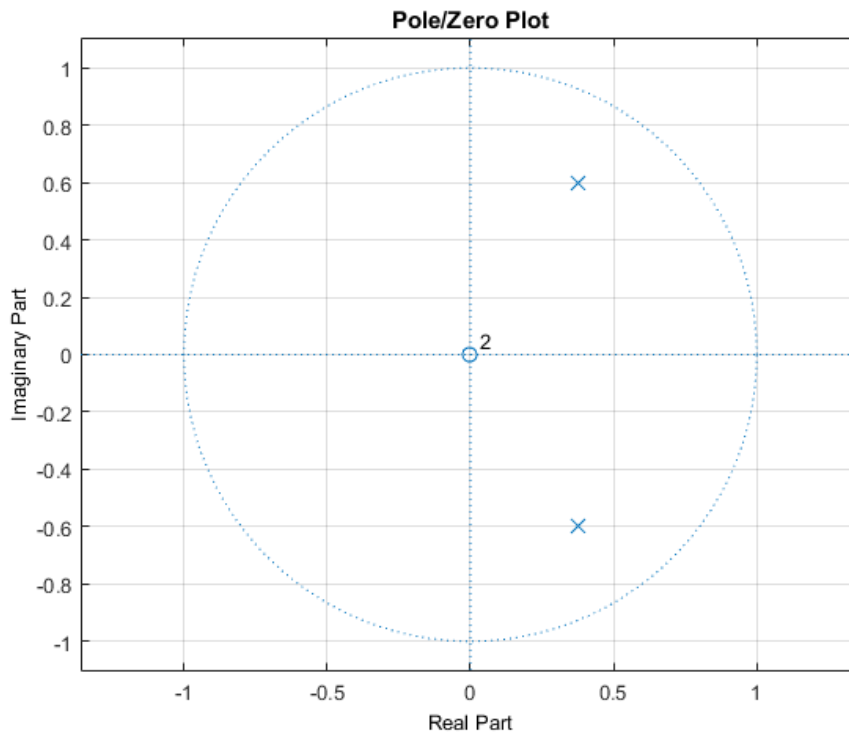
```
[H2,W2] = freqz(B2,A2,1e3,1);
dr2 = max(20*log10(abs(H2)))-min(20*log10(abs(H2)))
```

```
dr2 =

    14.4984
```

By examining the placement of the poles, you see that this AR(2) process is stable. The two poles are inside the unit circle.

```
fvtool(B2,A2,'Analysis','polezero')
```
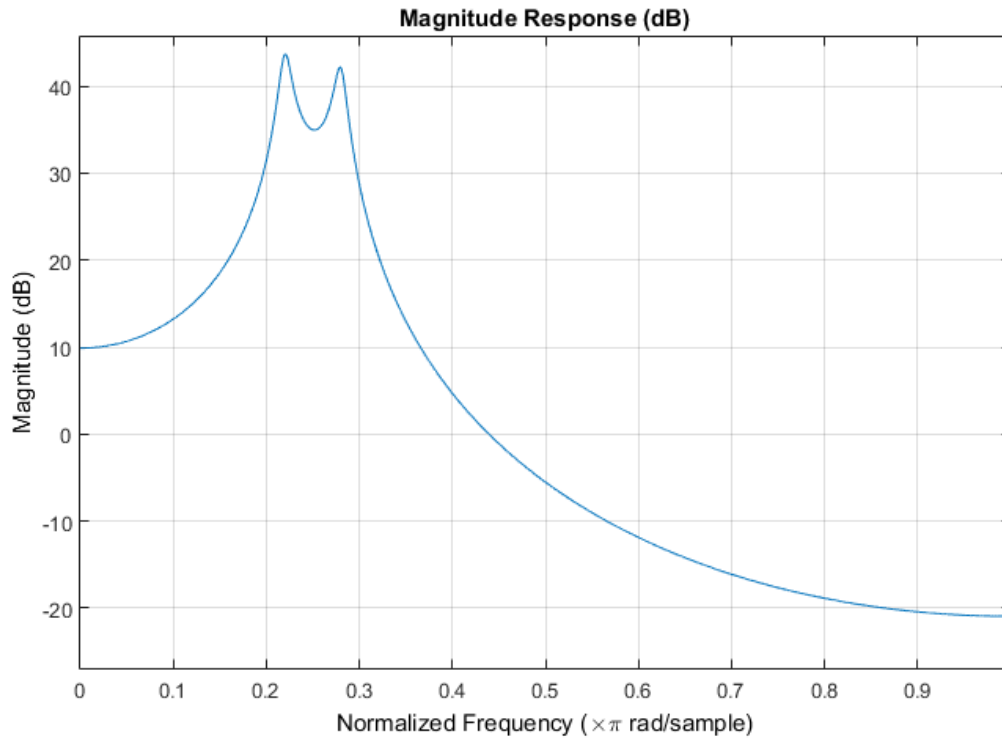
Pole/Zero Plot

Next, create an AR(4) process described by the following equation:

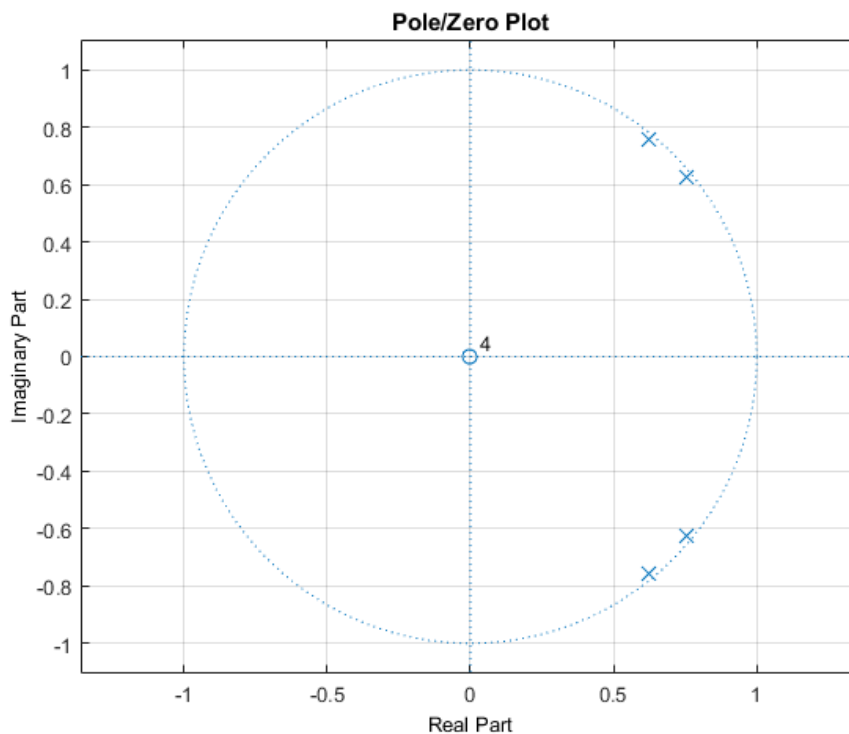$$y(n) - 2.7607y(n-1) + 3.8106y(n-2) - 2.6535y(n-3) + 0.9238y(n-4) = \varepsilon(n).$$

Use the following code to view the magnitude response of this IIR system.

```
B4 = 1;
A4 = [1 -2.7607 3.8106 -2.6535 0.9238];
fvtool(B4,A4)
```

Magnitude Response (dB)

Examining the placement of the poles, you can see this AR(4) process is also stable. The four poles are inside the unit circle.

```
fvtool(B4,A4,'Analysis','polezero')
```

The dynamic range of this PSD is approximately 65 dB, much larger than the AR(2) model.

```
[H4,W4] = freqz(B4,A4,1e3,1);
dr4 = max(20*log10(abs(H4)))-min(20*log10(abs(H4)))
```

```
dr4 =

   64.6213
```

To simulate realizations from these AR(*p*) processes, use `randn` and `filter`. Set the random number generator to the default settings to produce repeatable results. Plot the realizations.

```
rng default
```

```matlab
x = randn(1e3,1);
y2 = filter(B2,A2,x);
y4 = filter(B4,A4,x);

subplot(2,1,1)
plot(y2)
title('AR(2) Process')
xlabel('Time')

subplot(2,1,2)
plot(y4)
title('AR(4) Process')
xlabel('Time')
```
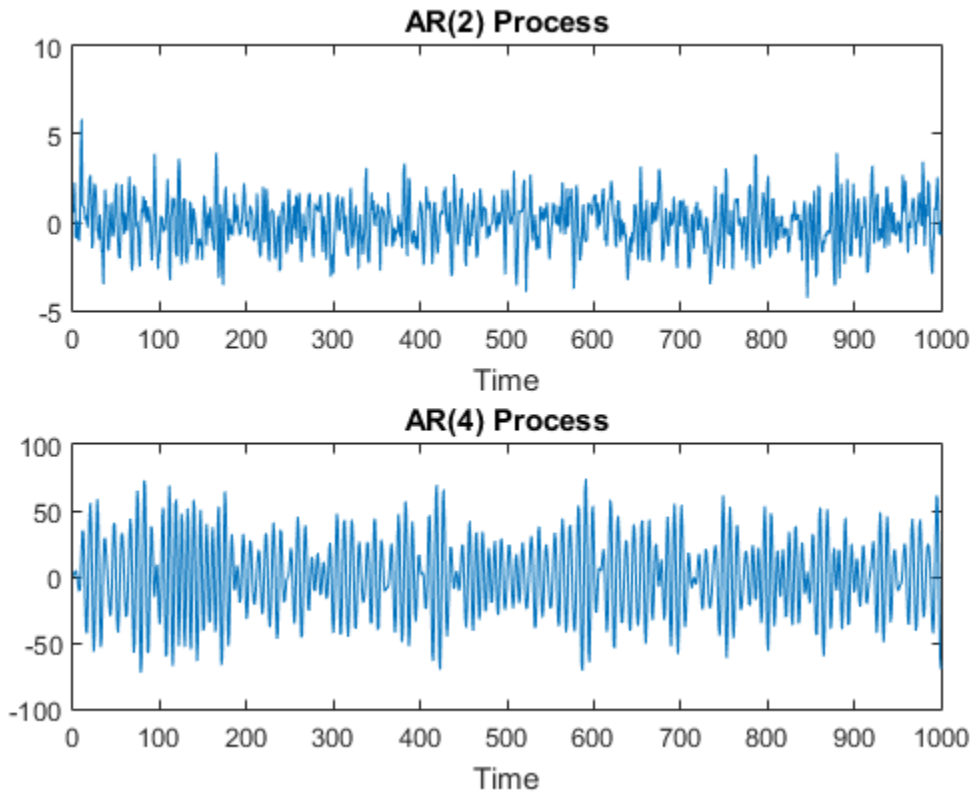
Compute and plot the periodograms of the AR(2) and AR(4) realizations. Compare the results against the true PSD. Note that `periodogram` converts the frequencies to millihertz for plotting.

```
Fs = 1;
NFFT = length(y2);

subplot(2,1,1)
periodogram(y2,rectwin(NFFT),NFFT,Fs)
hold on
plot(1000*W2,20*log10(abs(H2)),'r','linewidth',2)
title('AR(2) PSD and Periodogram')

subplot(2,1,2)
periodogram(y4,rectwin(NFFT),NFFT,Fs)
hold on
plot(1000*W4,20*log10(abs(H4)),'r','linewidth',2)
title('AR(4) PSD and Periodogram')
text(350,20,'\downarrow Bias')
```
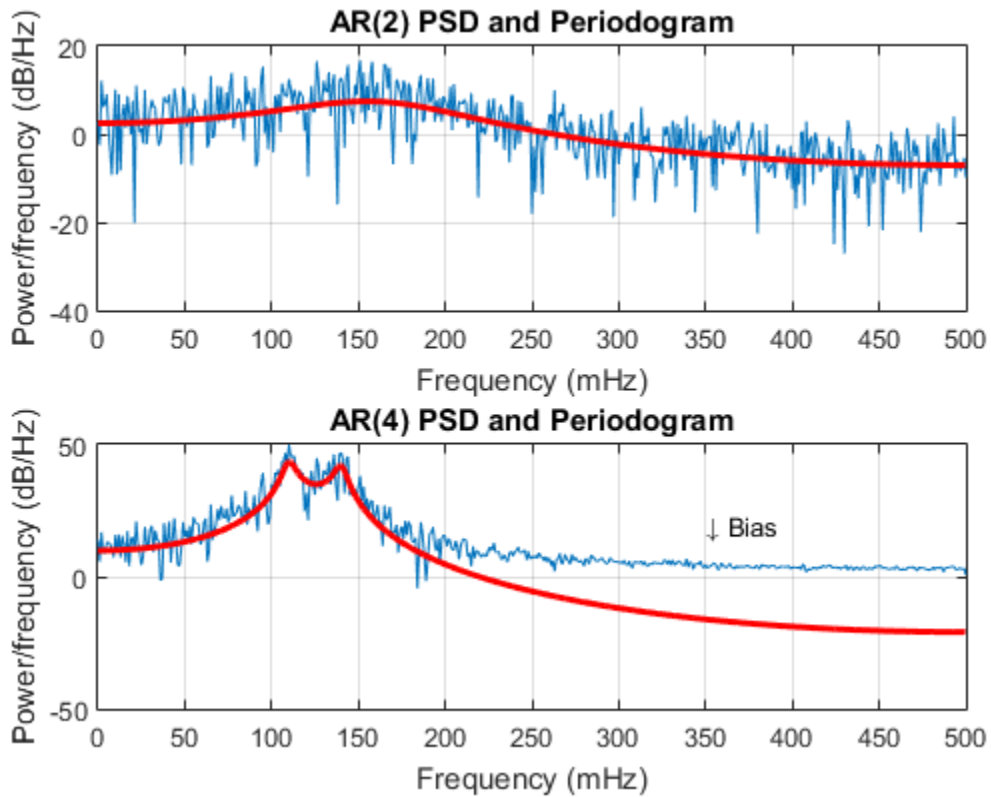
In the case of the AR(2) process, the periodogram estimate follows the shape of the true PSD but exhibits considerable variability. This is due to the low degrees of freedom. The pronounced negative deflections (in dB) in the periodogram are explained by taking the log of a chi-square random variable with two degrees of freedom.

In the case of the AR(4) process, the periodogram follows the shape of the true PSD at low frequencies but deviates from the PSD in the high frequencies. This is the effect of the convolution with Fejer's kernel. The large dynamic range of the AR(4) process compared to the AR(2) process is what makes the bias more pronounced.

Mitigate the bias demonstrated in the AR(4) process by using a taper, or window. In this example, use a Hamming window to taper the AR(4) realization before obtaining the periodogram.

```
figure
periodogram(y4,hamming(length(y4)),NFFT,Fs)
hold on
plot(1000*W4,20*log10(abs(H4)),'r','linewidth',2)
title('AR(4) PSD and Periodogram with Hamming Window')
legend('Periodogram','AR(4) PSD')
```
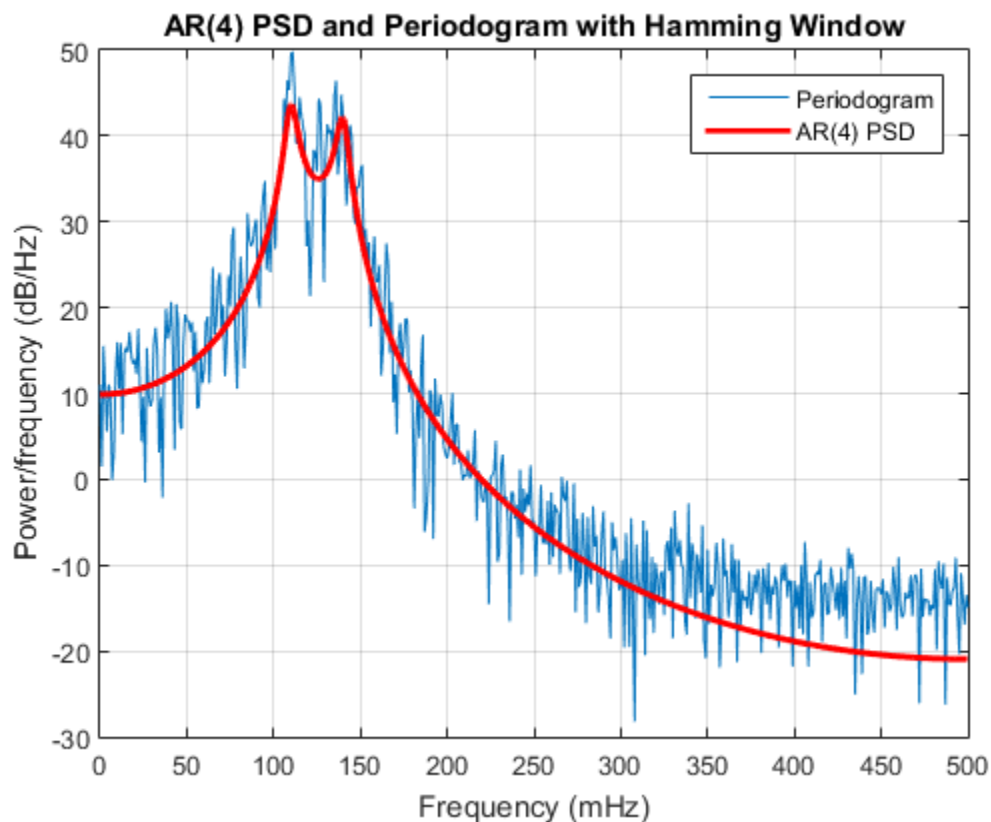

Note that the periodogram estimate now follows the true AR(4) PSD over the entire Nyquist frequency range. The periodogram estimates still only have two degrees of freedom so the use of a window does not reduce the variability of periodogram, but it does address bias.

In nonparametric spectral estimation, two methods for increasing the degrees of freedom and reducing the variability of the periodogram are Welch's overlapped segment averaging and multitaper spectral estimation.

Obtain a multitaper estimate of the AR(4) time series using a time half bandwidth product of 3.5. Plot the result.

```
NW = 3.5;

figure
pmtm(y4,NW,NFFT,Fs)
hold on
plot(1000*W4,20*log10(abs(H4)),'r','linewidth',2)
legend('Multitaper Estimate','AR(4) PSD')
```
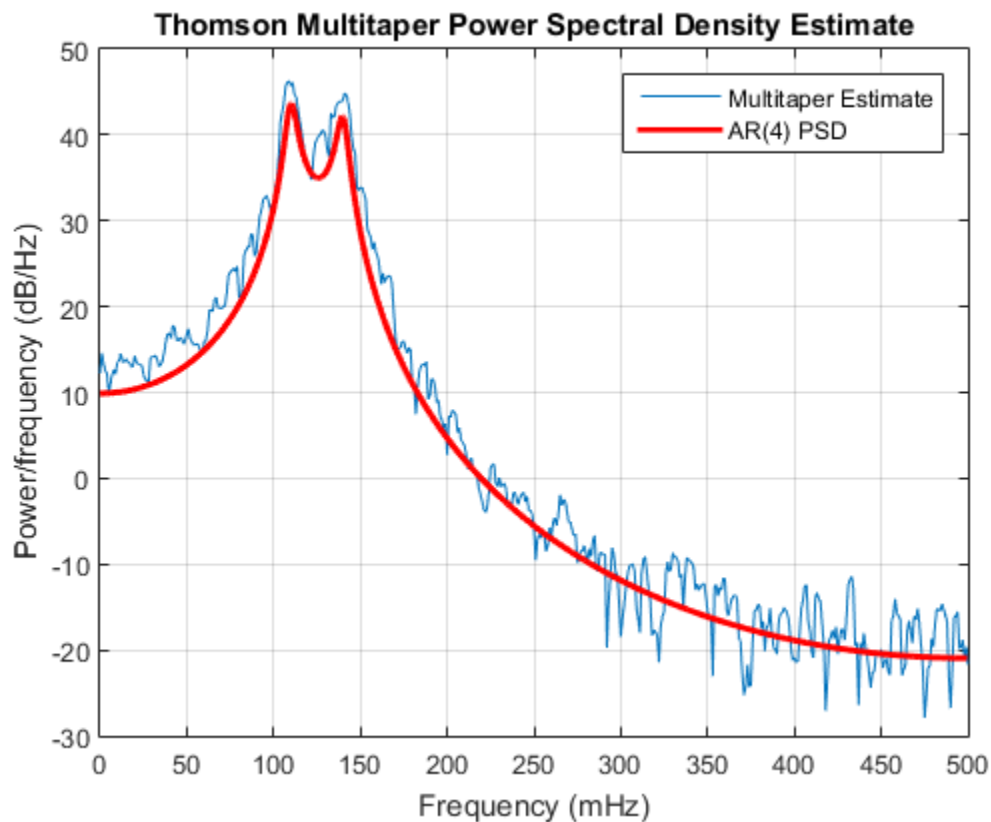
The multitaper method produces a PSD estimate with significantly less variability than the periodogram. Because the multitaper method also uses windows, you see that the bias of the periodogram is also addressed.

Cross Spectrum and Magnitude-Squared Coherence

This example shows how to use the cross spectrum to obtain the phase lag between sinusoidal components in a bivariate time series. The example also uses the magnitude-squared coherence (MSC) to identify significant frequency-domain correlation at the sine wave frequencies.

Create the bivariate time series. The individual series consist of two sine waves with frequencies of 100 and 200 Hz embedded in additive white Gaussian noise and sampled at 1 kHz. The sine waves in the x-series both have amplitudes equal to 1. The 100 Hz sine wave in the y-series has amplitude 0.5 and the 200 Hz sine wave in the y-series has amplitude 0.35. The 100 Hz and 200 Hz sine waves in the y-series are phase-lagged by $\pi/4$ radians and $\pi/2$ radians respectively. You can think of y-series as the noise-corrupted output of a linear system with input x. Set the random number generator to the default settings for reproducible results.

```
rng default

Fs = 1000;
t = 0:1/Fs:1-1/Fs;

x = cos(2*pi*100*t)+sin(2*pi*200*t)+0.5*randn(size(t));
y = 0.5*cos(2*pi*100*t-pi/4)+0.35*sin(2*pi*200*t-pi/2)+ ...
    0.5*randn(size(t));
```

Obtain the magnitude-squared coherence (MSC) for the bivariate time series. The magnitude-squared coherence enables you to identify significant frequency-domain correlation between the two time series. Phase estimates in the cross spectrum are only useful where significant frequency-domain correlation exists.

To prevent obtaining a magnitude-squared coherence estimate, which is identically 1 for all frequencies, you must use an averaged MSC estimator. Both Welch's overlapped segment averaging (WOSA) and mulitaper techniques are appropriate. mscohere implements a WOSA estimator.

Set the window length to 100 samples. This window length contains 10 periods of the 100 Hz sine wave and 20 periods of the 200 Hz sine wave. Use an overlap of 80 samples with the default Hamming window. Input the sample rate explicitly to get the output frequencies in Hz. Plot the magnitude-squared coherence.

```
[Pxy,F] = mscohere(x,y,hamming(100),80,100,Fs);
```

```
plot(F,Pxy)
title('Magnitude-Squared Coherence')
xlabel('Frequency (Hz)')
grid
```


Magnitude-Squared Coherence
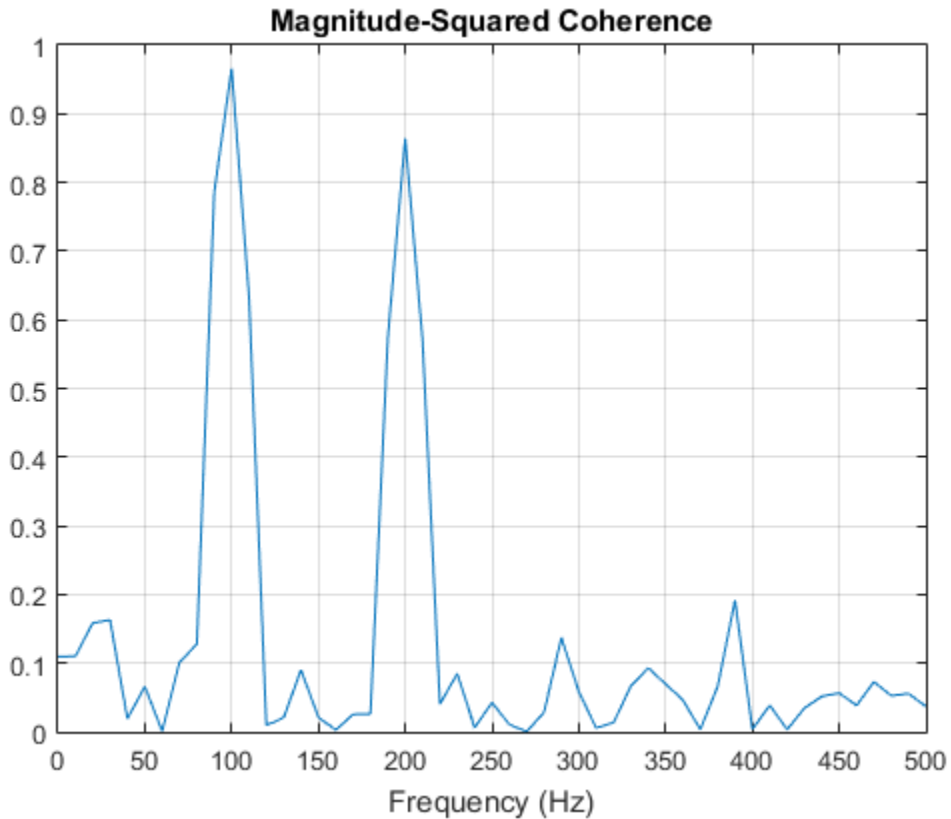
You see that the magnitude-squared coherence is greater than 0.8 at 100 and 200 Hz.

Obtain the cross spectrum of x and y using cpsd. Use the same parameters to obtain the cross spectrum that you used in the MSC estimate. Plot the phase of the cross spectrum and indicate the frequencies with significant coherence between the two times. Mark the known phase lags between the sinusoidal components.

```
[Cxy,F] = cpsd(x,y,hamming(100),80,100,Fs);
```

```
plot(F,-angle(Cxy)/pi)
title('Cross Spectrum Phase')
xlabel('Frequency (Hz)')
ylabel('Lag (\times\pi rad)')

ax = gca;
ax.XTick = [100 200];
ax.YTick = [-1 -1/2 -1/4 0 1/4 1/2 1];
grid
```
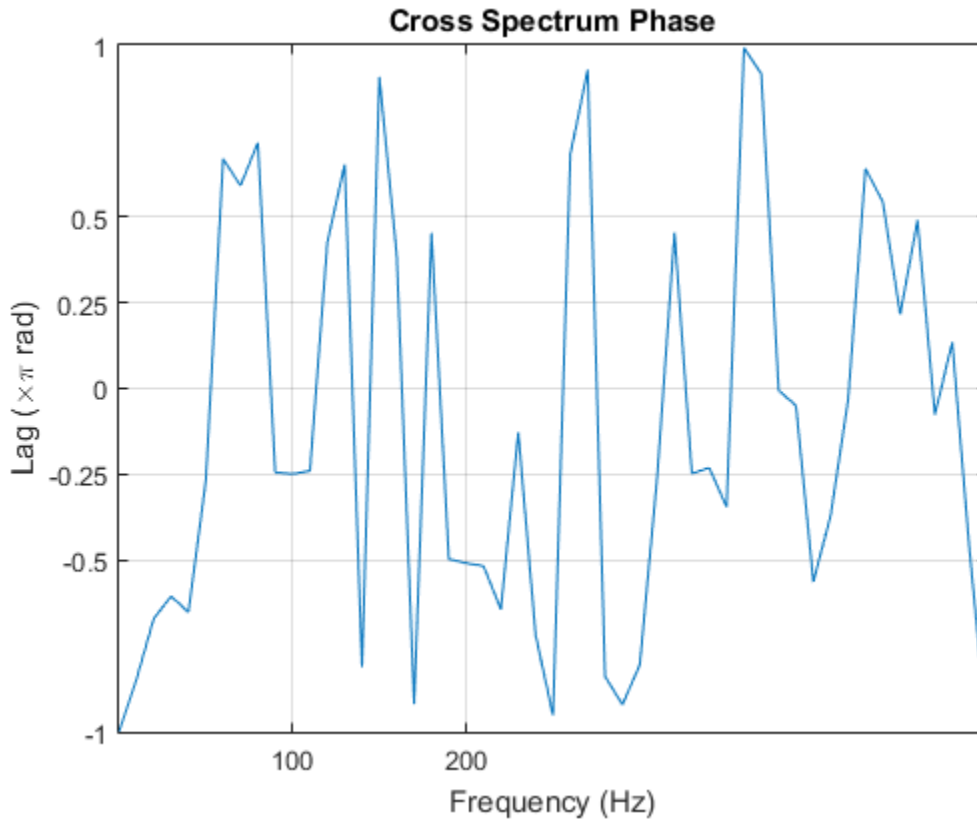

You see that, at 100 Hz and 200 Hz, the phase lags estimated from the cross spectrum are close to the true values.

In this example, the cross spectrum estimates are spaced at $1000/100 = 10$ Hz. You can return the phase estimates at those frequency bins. Keep in mind that the first frequency bin corresponds to 0 Hz, or DC.

```
phi100 = -angle(Cxy(11));
phi200 = -angle(Cxy(21));
```

You see that `phi100` and `phi200` are close to $-\pi/4$ and $-\pi/2$.

```
lag100 = phi100/pi
lag200 = phi200/pi


lag100 =

   -0.2488


lag200 =

   -0.5086
```

Amplitude Estimation and Zero Padding

This example shows how to use zero padding to obtain an accurate estimate of the amplitude of a sinusoidal signal. Frequencies in the discrete Fourier transform (DFT) are spaced at intervals of F_s/N, where F_s is the sample rate and N is the length of the input time series. Attempting to estimate the amplitude of a sinusoid with a frequency that does not correspond to a DFT *bin* can result in an inaccurate estimate. Therefore, zero padding the data before you obtain the DFT can often increase the frequency resolution and thus improve the amplitude estimate.

Create a signal consisting of two sine waves. The two sine waves have frequencies of 100 and 202.5 Hz. The sample rate is 1000 Hz and the signal is 1000 samples in length.

```
Fs = 1e3;
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)+sin(2*pi*202.5*t);
```

Obtain the DFT of the signal. The DFT bins are spaced at 1 Hz. Accordingly, the 100 Hz sine wave corresponds to a DFT bin, but the 202.5 Hz sine wave does not.

Because the signal is real-valued, use only the positive frequencies from the DFT to estimate the amplitude. Scale the DFT by the length of the input signal and multiply all frequencies except 0 and the Nyquist by 2.

Plot the result with the known amplitudes for comparison.

```
xdft = fft(x);
xdft = xdft(1:length(x)/2+1);
xdft = xdft/length(x);
xdft(2:end-1) = 2*xdft(2:end-1);
freq = 0:Fs/length(x):Fs/2;

plot(freq,abs(xdft))
hold on
plot(freq,ones(length(x)/2+1,1),'LineWidth',2)
xlabel('Hz')
ylabel('Amplitude')
```

12-25
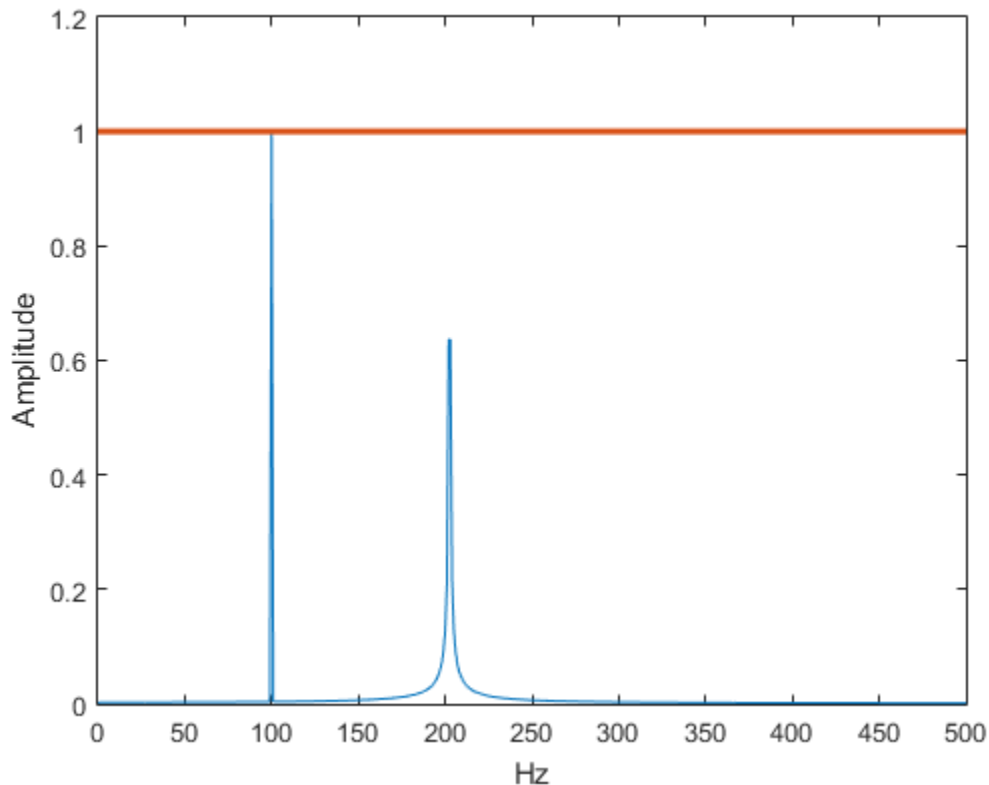
The amplitude estimate at 100 Hz is accurate because that frequency corresponds to a DFT bin. However, the amplitude estimate at 202.5 Hz is not accurate because that frequency does not correspond to a DFT bin.

You can interpolate the DFT by zero padding. While zero padding does not improve the frequency resolution of the DFT, zero padding can allow you to obtain more accurate amplitude estimates of resolvable signal components.

Pad the DFT out to length 2000. With this length, the spacing between DFT bins is $F_s/2000 = 0.5$ Hz. In this case, the energy from the 202.5 Hz sine wave falls directly in a DFT bin. Obtain the DFT and plot the amplitude estimates. Use zero padding out to 2000 samples.

```
xdft = fft(x,2000);
xdft = xdft(1:length(xdft)/2+1);
xdft = xdft/length(x);
xdft(2:end-1) = 2*xdft(2:end-1);
freq = 0:Fs/(2*length(x)):Fs/2;

figure
plot(freq,abs(xdft));
hold on
plot(freq,ones(2*length(x)/2+1,1),'LineWidth',2)
xlabel('Hz')
ylabel('Amplitude')
```
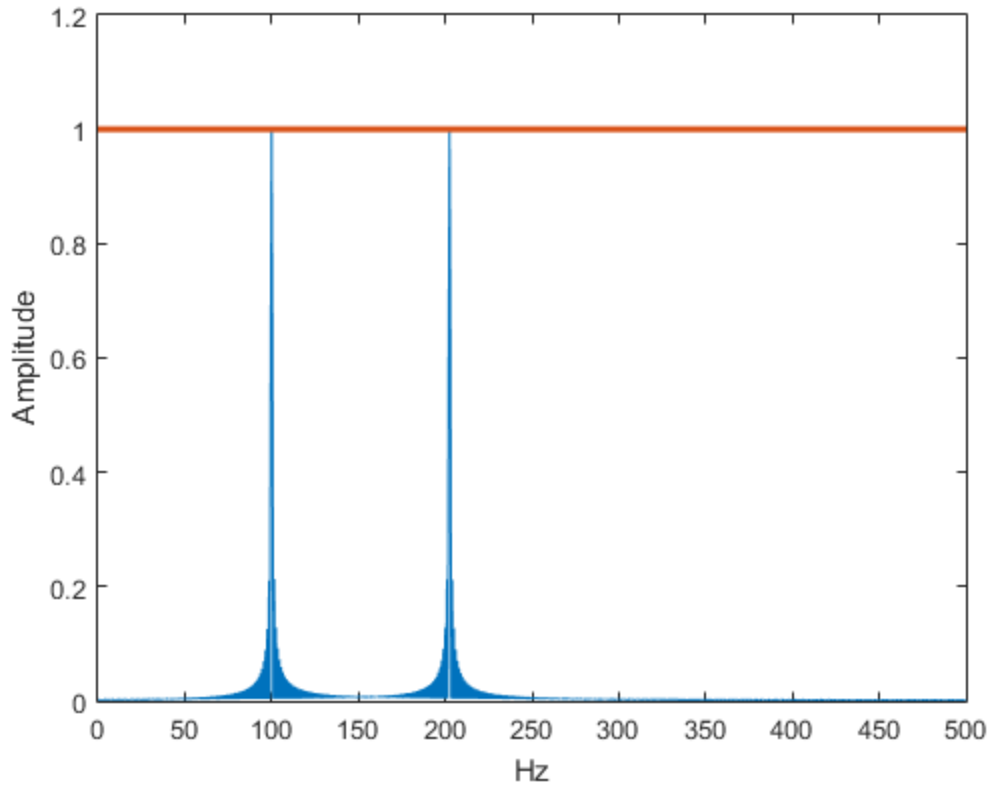
The use of zero padding enables you to estimate the amplitudes of both frequencies correctly.

Significance Testing for Periodic Component

This example shows how to assess the significance of a sinusoidal component in white noise using Fisher's *g*-statistic. Fisher's *g*-statistic is the ratio of the largest periodogram value to the sum of all the periodogram values over 1/2 of the frequency interval, (0,Fs/2). A detailed description of the *g*-statistic and exact distribution can be found in [1] and [2].

Create a signal consisting of a 100-Hz sine wave in white Gaussian noise with zero mean and variance 1. The amplitude of the sine wave is 0.25. The sampling rate is 1 kHz. Set the random number generator to the default settings for reproducible results.

```
Fs = 1e3;
t = 0:0.001:1-0.001;
rng default;
x = 0.25*cos(2*pi*100*t)+randn(size(t));
```

Obtain the periodogram of the signal using `periodogram`. Exclude 0 and the Nyquist frequency (Fs/2).

```
[Pxx,F] = periodogram(x,rectwin(length(x)),length(x),Fs);
Pxx = Pxx(2:length(x)/2);
```

Find the maximum value of the periodogram. Fisher's *g*-statistic is the ratio of the maximum periodogram value to the sum of all periodogram values.

```
[maxval,index] = max(Pxx);
fisher_g = Pxx(index)/sum(Pxx);
```

The maximum periodogram value occurs at 100 Hz, which you can verify by finding the frequency corresponding to the index of the maximum periodogram value.

```
F = F(2:end-1);
F(index)
```

Use the distributional results detailed in [1] and [2] to determine the significance level, `pval`, of Fisher's *g*-statistic. The following MATLAB code implements equation 6 on page 7 in [2].

```
N = length(Pxx);
    upper  = floor(1/fisher_g);
    for nn = 1:3
        I(nn) = ...
            (-1)^(nn-1)*nchoosek(N,nn)*(1-nn*fisher_g)^(N-1);
```

```
    end
pval = sum(I);
```

The *p*-value is less than 0.00001, which indicates a significant periodic component at 100 Hz. The interpretation of Fisher's *g*-statistic is complicated by the presence of other periodicities. See [1] for a modification when multiple periodicities may be present.

References

[1] Percival, Donald B. and Andrew T. Walden. *Spectral Analysis for Physical Applications.* Cambridge, UK: Cambridge University Press, 1993, p. 491.

[2] Wichert, Sofia, Konstantinos Fokianos, and Korbinian Strimmer. "Identifying Periodically Expressed Transcripts in Microarray Time Series Data." *Bioinformatics.* Vol.20, 2004, pp.5–20.

Frequency Estimation by Subspace Methods

This example shows how to resolve closely spaced sine waves using subspace methods. Subspace methods assume a harmonic model consisting of a sum of sine waves, possibly complex, in additive noise. In a complex-valued harmonic model, the noise is also complex-valued.

Create a complex-valued signal 24 samples in length. The signal consists of two complex exponentials (sine waves) with frequencies of 0.50 Hz and 0.52 Hz and additive complex white Gaussian noise. The noise has zero mean and variance 0.2^2. In a complex white noise, both the real and imaginary parts have variance equal to 1/2 the overall variance.

```
n = 0:23;
rng default
x = exp(1j*2*pi*0.5*n)+exp(1j*2*pi*0.52*n)+ ...
    0.2/sqrt(2)*(randn(size(n))+1j*randn(size(n)));
```

Using `periodogram`, attempt to resolve the two sine waves.

```
periodogram(x,rectwin(length(x)),128,1)
```

Periodogram Power Spectral Density Estimate
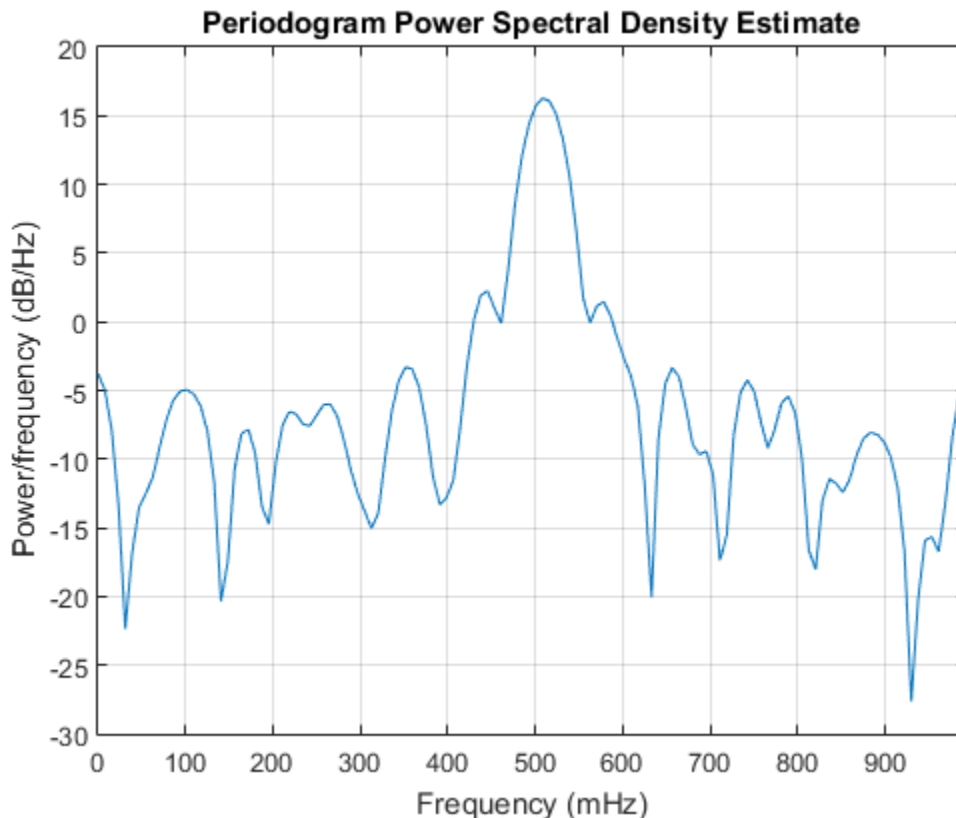
The periodogram shows a broad peak near 1/2 Hz. You cannot resolve the two separate sine waves because the frequency resolution of the periodogram is 1/_N_, where *N* is the length of the signal. In this case, 1/_N_ is greater than the separation of the two sine waves. Zero padding does not help to resolve two separate peaks.
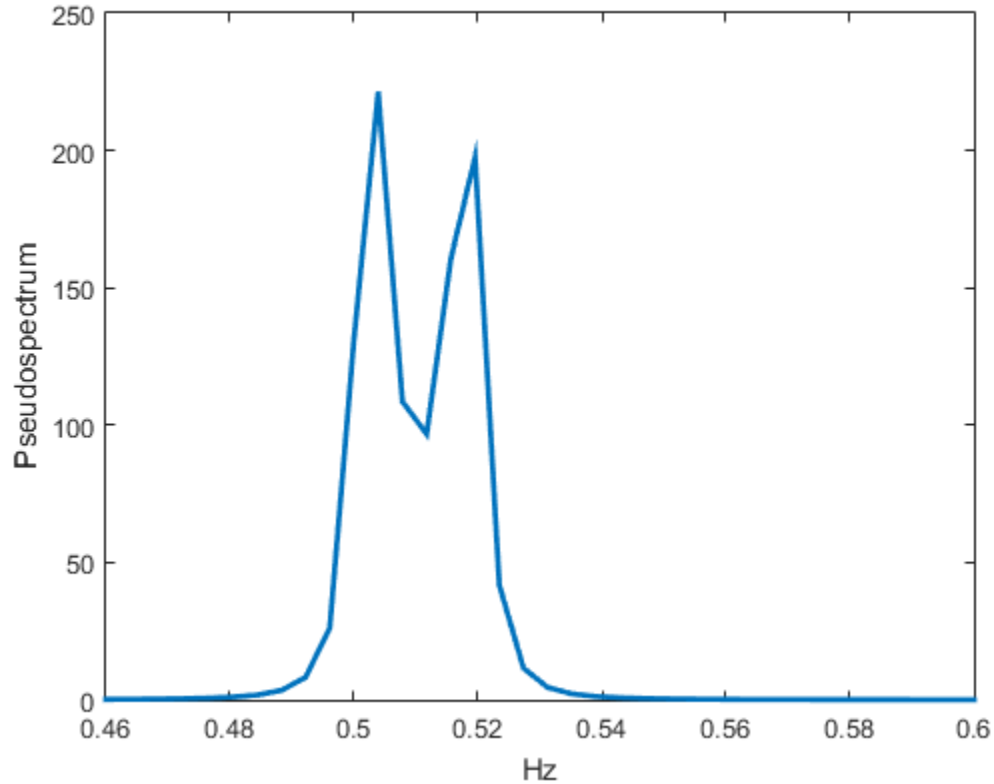
Use a subspace method to resolve the two closely spaced peaks. In this example, use the root-MUSIC method. Estimate the autocorrelation matrix and input the autocorrelation matrix into `pmusic`. Specify a model with two sinusoidal components. Plot the result.

```
[X,R] = corrmtx(x,14,'mod');
[S,F] = pmusic(R,2,[],1,'corr');

plot(F,S,'linewidth',2)
xlim([0.46 0.60])
```

```
xlabel('Hz')
ylabel('Pseudospectrum')
```


The root-MUSIC method is able to separate the two peaks at 0.5 and 0.52 Hz. However, subspace methods do not produce power estimates like power spectral density estimates. Subspace methods are most useful for frequency identification and can be sensitive to model-order misspecification.

Frequency-Domain Linear Regression

This example shows how to use the discrete Fourier transform to construct a linear regression model for a time series. The time series used in this example is the monthly number of accidental deaths in the United States from 1973 to 1979. The data are published in Brockwell and Davis (2006). The original source is the U. S. National Safety Council.

Enter the data. Copy the `exdata` matrix into the MATLAB® workspace.

```
exdata = [
        9007      7750      8162      7717      7792      7836
        8106      6981      7306      7461      6957      6892
        8928      8038      8124      7776      7726      7791
        9137      8422      7870      7925      8106      8129
       10017      8714      9387      8634      8890      9115
       10826      9512      9556      8945      9299      9434
       11317     10120     10093     10078     10625     10484
       10744      9823      9620      9179      9302      9827
        9713      8743      8285      8037      8314      9110
        9938      9129      8433      8488      8850      9070
        9161      8710      8160      7874      8265      8633
        8927      8680      8034      8647      8796      9240];
```

`exdata` is a 12-by-6 matrix. Each column of exdata contains 12 months of data. The first row of each column contains the number of U.S. accidental deaths for January of the corresponding year. The last row of each column contains the number of U.S. accidental deaths for December of the corresponding year.

Reshape the data matrix into a 72-by-1 time series and plot the data for the years 1973 to 1978.

```
ts = reshape(exdata,72,1);
years = linspace(1973,1979,72);

plot(years,ts,'o-','MarkerFaceColor','auto')
xlabel('Year')
ylabel('Number of Accidental Deaths')
```
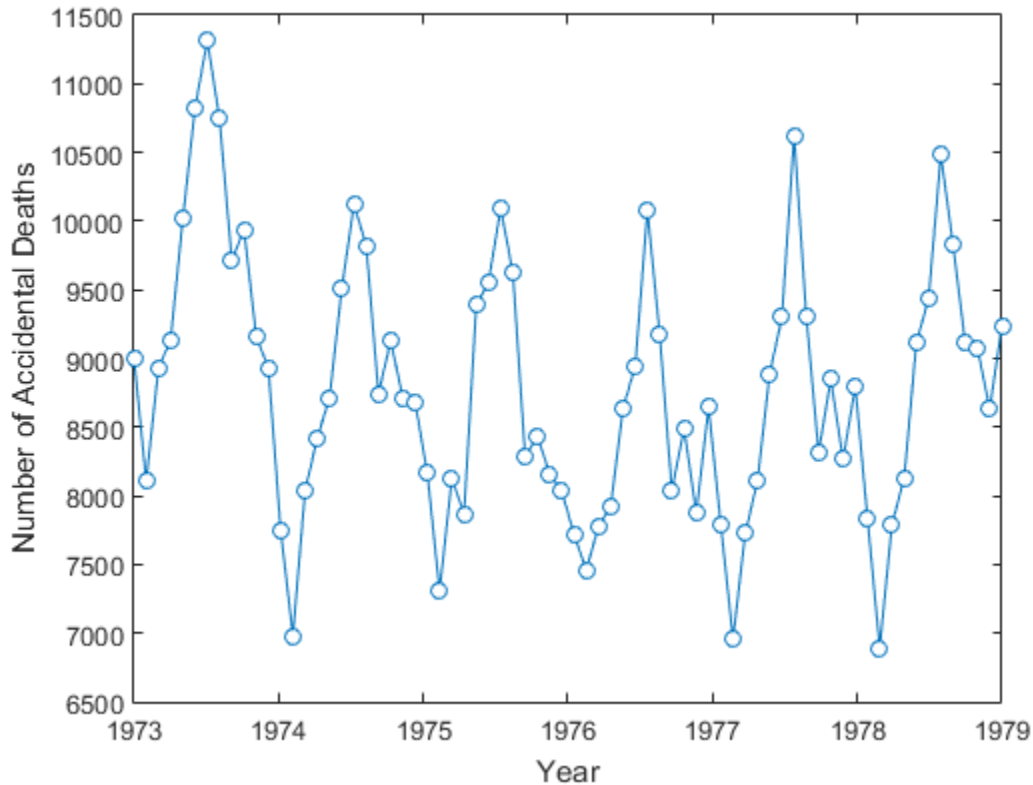
A visual inspection of the data indicates that number of accidental deaths varies in a periodic manner. The period of the oscillation appears to be roughly 1 year (12 months). The periodic nature of the data suggests that an appropriate model may be

$$X(n) = \mu + \sum_k \left(A_k \cos \frac{2\pi kn}{N} + B_k \cos \frac{2\pi kn}{N} \right) + \varepsilon(n),$$

where μ is the overall mean, N is the length of the time series, and $\varepsilon(n)$ is a white noise sequence of independent and identically-distributed (iid) Gaussian random variables with zero mean and some variance. The additive noise term accounts for the randomness inherent in the data. The parameters of the model are the overall mean and the amplitudes of the cosines and sines. The model is linear in the parameters.

To construct a linear regression model in the time domain, you have to specify which frequencies to use for the cosines and sines, form the design matrix, and solve the normal equations in order to obtain the least-squares estimates of the model parameters. In this case, it is easier to use the discrete Fourier transform to detect the periodicities, retain only a subset of the Fourier coefficients, and invert the transform to obtain the fitted time series.

Perform a spectral analysis of the data to reveal which frequencies contribute significantly to the variability in the data. Because the overall mean of the signal is approximately 9,000 and is proportional to the Fourier transform at 0 frequency, subtract the mean prior to the spectral analysis. This reduces the large magnitude Fourier coefficient at 0 frequency and makes any significant oscillations easier to detect. The frequencies in the Fourier transform are spaced at an interval that is the reciprocal of the time series length, 1/72. Sampling the data monthly, the highest frequency in the spectral analysis is 1 cycle/2 months. In this case, it is convenient to look at the spectral analysis in terms of cycles/year so scale the frequencies accordingly for visualization.

```
tsdft = fft(ts-mean(ts));
freq = 0:1/72:1/2;

plot(freq.*12,abs(tsdft(1:length(ts)/2+1)),'o-', ...
    'MarkerFaceColor','auto')
xlabel('Cycles/Year')
ylabel('Magnitude')
ax = gca;
ax.XTick = [1/6 1 2 3 4 5 6];
```
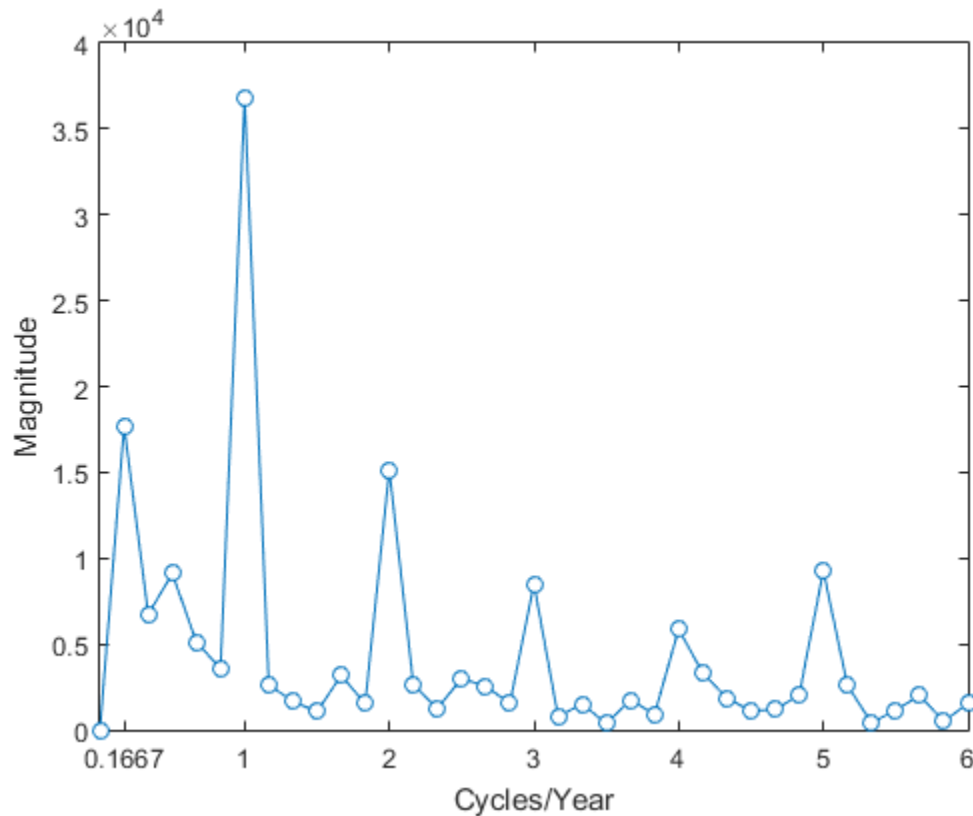
Based on the magnitudes, the frequency of 1 cycle/12 months is the most significant oscillation in the data. The magnitude at 1 cycle/12 months is more than twice as large as any other magnitude. However, the spectral analysis reveals that there are also other periodic components in the data. For example, there appears to be periodic components at harmonics (integer multiples) of 1 cycle/12 months. There also appears to be a periodic component with a period of 1 cycle/72 months.

Based on the spectral analysis of the data, fit a simple linear regression model using a cosine and sine term with a frequency of the most signficant component: 1 cycle/year (1 cycle/12 months).

Determine the frequency bin in the discrete Fourier transform that corresponds to 1 cycle/12 months. Because the frequencies are spaced at 1/72 and the first bin corresponds

to 0 frequency, the correct bin is 72/12+1. This is the frequency bin of the *positive* frequency. You must also include the frequency bin corresponding to the *negative* frequency: -1 cycle/12 months. With MATLAB indexing, the frequency bin of the negative frequency is 72-72/12+3.

Create a 72-by-1 vector of zeros. Fill the appropriate elements of the vector with the Fourier coefficients corresponding to a positive and negative frequency of 1 cycle/12 months. Invert the Fourier transform and add the overall mean to obtain a fit to the accidental death data.

```
N = 72;
freqbin = N/12+1;
freqbins = [freqbin N-freqbin+2];
tsfit = zeros(72,1);
tsfit(freqbins) = tsdft(freqbins);
tsfit = ifft(tsfit,'symmetric');
mu = mean(ts);
tsfit = mu+tsfit;
```

Plot the original data along with the fitted series using two Fourier coefficients.

```
plot(years,ts,'o-','MarkerFaceColor','auto')
xlabel('Year')
ylabel('Number of Accidental Deaths')
hold on
plot(years,tsfit,'linewidth',2)
legend('Data','Fitted Model')
hold off
```
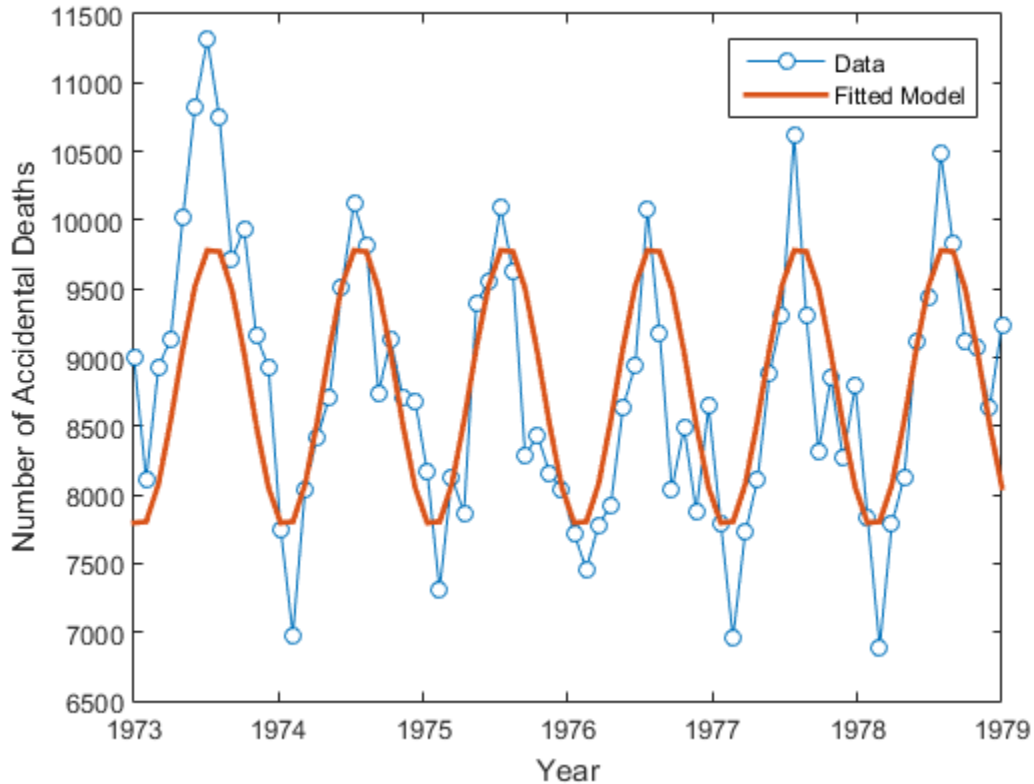
The fitted model appears to capture the general periodic nature of the data and supports the initial conclusion that data oscillate with a cycle of 1 year.

To assess how adequately the single frequency of 1 cycle/12 months accounts for the observed time series, form the residuals. If the residuals resemble a white noise sequence, the simple linear model with one frequency has adequately modeled the time series.

To assess the residuals, use the autocorrelation sequence with 95%-confidence intervals for a white noise.

```
resid = ts-tsfit;
[xc,lags] = xcorr(resid,50,'coeff');
```

```
stem(lags(51:end),xc(51:end),'filled')
hold on
lconf = -1.96*ones(51,1)/sqrt(72);
uconf = 1.96*ones(51,1)/sqrt(72);
plot(lags(51:end),lconf,'r')
plot(lags(51:end),uconf,'r')
xlabel('Lag')
ylabel('Correlation Coefficient')
title('Autocorrelation of Residuals')
hold off
```
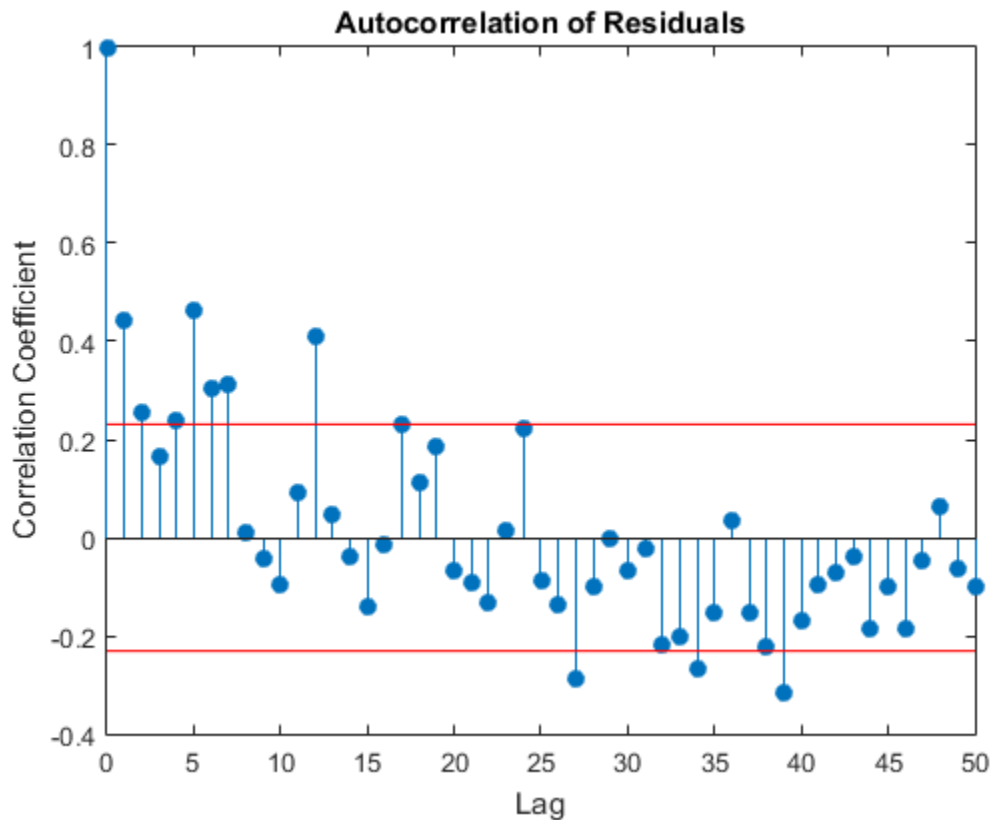

The autocorrelation values fall outside the 95% confidence bounds at a number of lags. It does not appear that the residuals are white noise. The conclusion is that the simple linear model with one sinusoidal component does not account for all the oscillations in

the number of accidental deaths. This is expected because the spectral analysis revealed additional periodic components in addition to the dominant oscillation. Creating a model that incorporates additional periodic terms indicated by the spectral analysis will improve the fit and whiten the residuals.

Fit a model which consists of the three largest Fourier coefficient magnitudes. Because you have to retain the Fourier coefficients corresponding to both negative and positive frequencies, retain the largest 6 indices.

```
tsfit2dft = zeros(72,1);
[Y,I] = sort(abs(tsdft),'descend');
indices = I(1:6);
tsfit2dft(indices) = tsdft(indices);
```

Demonstrate that preserving only 6 of the 72 Fourier coefficients (3 frequencies) retains most of the signal's energy. First, demonstrate that retaining all the Fourier coefficients yields energy equivalence between the original signal and the Fourier transform.

```
norm(1/sqrt(72)*tsdft,2)/norm(ts-mean(ts),2)
```

```
ans =

    1.0000
```

The ratio is 1. Now, examine the energy ratio where only 3 frequencies are retained.

```
norm(1/sqrt(72)*tsfit2dft,2)/norm(ts-mean(ts),2)
```

```
ans =

    0.8991
```

Almost 90% of the energy is retained. Equivalently, 90% of the variance of the time series is accounted for by 3 frequency components.

Form an estimate of the data based on 3 frequency components. Compare the original data, the model with one frequency, and the model with 3 frequencies.

```
tsfit2 = mu+ifft(tsfit2dft,'symmetric');
```

```
plot(years,ts,'o-','markerfacecolor','auto')
xlabel('Year')
ylabel('Number of Accidental Deaths')
hold on
plot(years,tsfit,'linewidth',2)
plot(years,tsfit2,'linewidth',2)
legend('Data','1 Frequency','3 Frequencies')
hold off
```
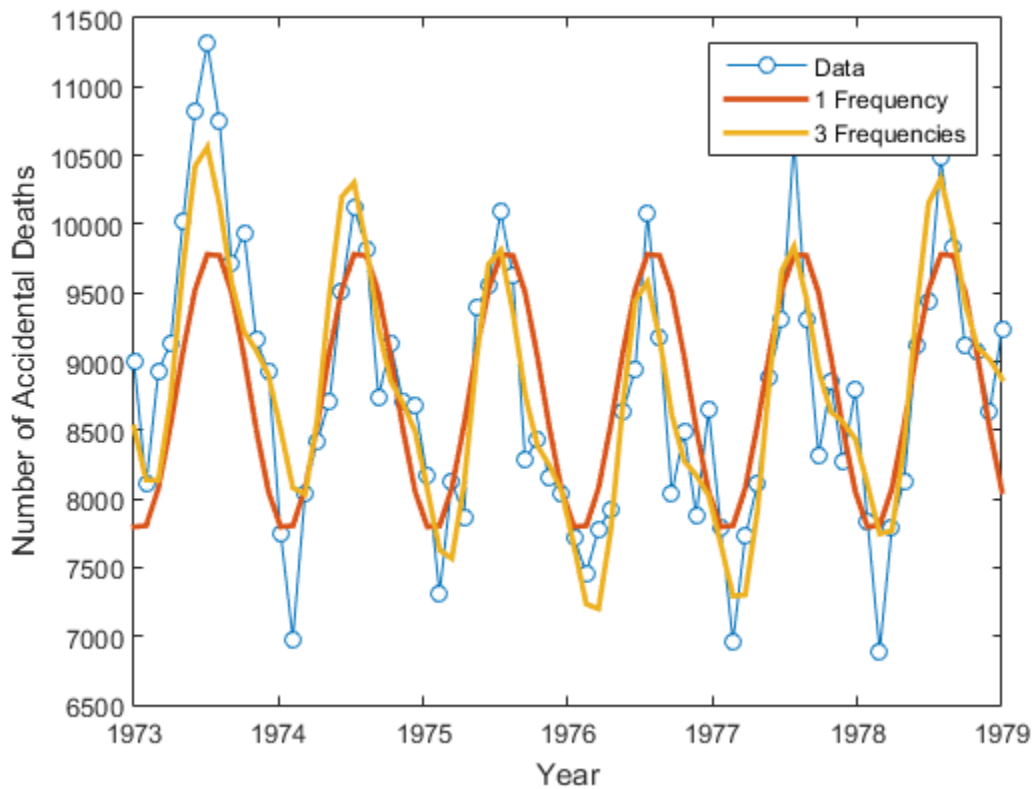

Using 3 frequencies has improved the fit to the original signal. You can see this by examining the autocorrelation of the residuals from the 3-frequency model.

```
resid = ts-tsfit2;
[xc,lags] = xcorr(resid,50,'coeff');
```

```
stem(lags(51:end),xc(51:end),'filled')
hold on
lconf = -1.96*ones(51,1)/sqrt(72);
uconf = 1.96*ones(51,1)/sqrt(72);
plot(lags(51:end),lconf,'r')
plot(lags(51:end),uconf,'r')
xlabel('Lag')
ylabel('Correlation Coefficient')
title('Autocorrelation of Residuals')
hold off
```
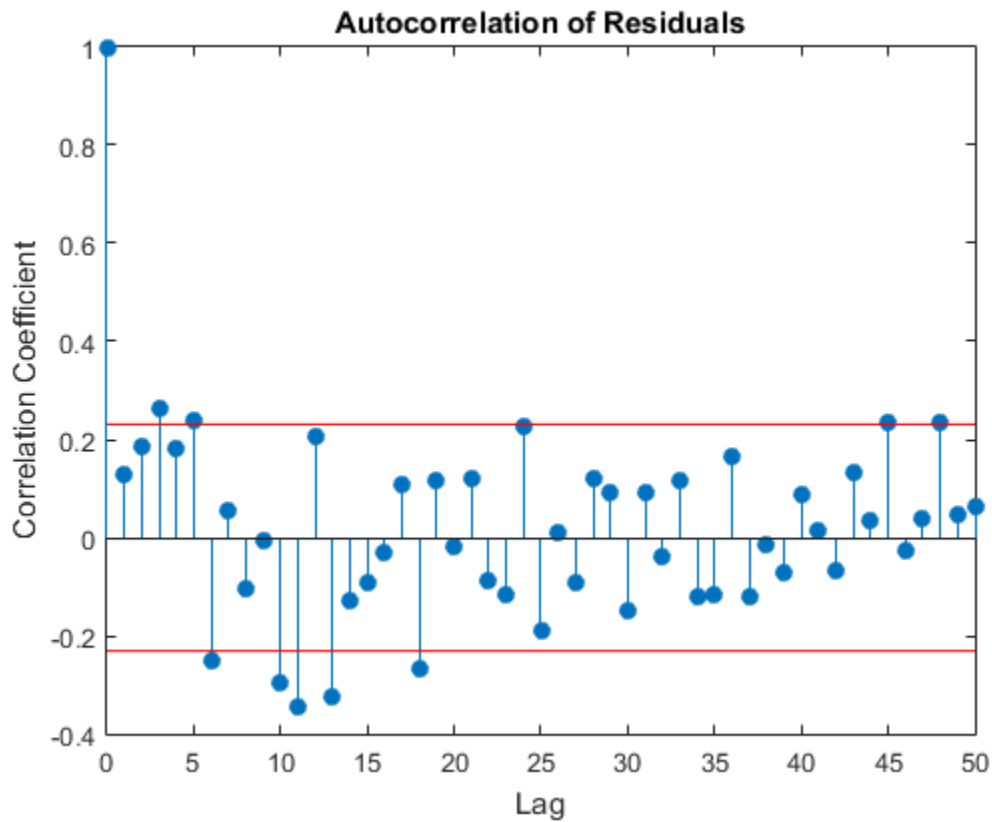

Using 3 frequencies has resulted in residuals that more closely approximate a white noise process.

Demonstrate that the parameter values obtained from the Fourier transform are equivalent to a time-domain linear regression model. Find the least-squares estimates for the overall mean, the cosine amplitudes, and the sine amplitudes for the three frequencies by forming the design matrix and solving the normal equations. Compare the fitted time series with that obtained from the Fourier transform.

```
X = ones(72,7);
X(:,2) = cos(2*pi/72*(0:71))';
X(:,3) = sin(2*pi/72*(0:71))';
X(:,4) = cos(2*pi*6/72*(0:71))';
X(:,5) = sin(2*pi*6/72*(0:71))';
X(:,6) = cos(2*pi*12/72*(0:71))';
X(:,7) = sin(2*pi*12/72*(0:71))';
beta = X\ts;
tsfit_lm = X*beta;
max(abs(tsfit_lm-tsfit2))


ans =

   7.2760e-12
```

The two methods yield identical results. The maximum absolute value of the difference between the two waveforms is on the order of 10-12. In this case, the frequency-domain approach was easier than the equivalent time-domain approach. You naturally use a spectral analysis to visually inspect which oscillations are present in the data. From that step, it is simple to use the Fourier coefficients to construct a model for the signal consisting of a sum cosines and sines.

For more details on spectral analysis in time series and the equivalence with time-domain regression see (Shumway and Stoffer, 2006).

While spectral analysis can answer which periodic components contribute significantly to the variability of the data, it does not explain why those components are present. If you examine these data closely, you see that the minimum values in the 12-month cycle tend to occur in February, while the maximum values occur in July. A plausible explanation for these data is that people are naturally more active in summer than in the winter. Unfortunately, as a result of this increased activity, there is an increased probability of the occurrence of fatal accidents.

References

Brockwell, Peter J., and Richard A. Davis. *Time Series: Theory and Methods*. New York: Springer, 2006.

Shumway, Robert H., and David S. Stoffer. *Time Series Analysis and Its Applications with R Examples*. New York: Springer, 2006.

Measure Total Harmonic Distortion

This example shows shows how to measure the total harmonic distortion (THD) of a sinusoidal signal. The example uses the following scenario: A manufacturer of audio speakers claims the model A speaker produces less than 0.09% harmonic distortion at 1 kHz with a 1 volt input. The harmonic distortion is measured with respect to the fundamental (THD-F).

Assume you record the following data obtained by driving the speaker with a 1 kHz tone at 1 volt. The data is sampled at 44.1 kHz for analysis.

```
Fs = 44.1e3;
t = 0:1/Fs:1;
x = cos(2*pi*1000*t)+8e-4*sin(2*pi*2000*t)+2e-5*cos(2*pi*3000*t-pi/4)+...
    8e-6*sin(2*pi*4000*t);
```

Obtain the total harmonic distortion of the input signal in dB. Specify that six harmonics are used in calculating the THD. This includes the fundamental frequency of 1 kHz. Input the sampling frequency of 44.1 kHz. Determine the frequencies of the harmonics and their power estimates.

```
nharm = 6;
[thd_db,harmpow,harmfreq] = thd(x,Fs,nharm);
```

The function `thd` outputs the total harmonic distortion in dB. Convert the measurement from dB to a percentage to compare the value against the manufacturer's claims.

```
percent_thd = 100*(10^(thd_db/20))
```

```
percent_thd =

    0.0800
```

The value you obtain indicates that the manufacturer's claims about the THD for speaker model A are correct.

You can obtain further insight by examining the power (dB) of the individual harmonics.

```
T = table(harmfreq,harmpow,'VariableNames',{'Frequency','Power'})
```

```
T =
```

```
Frequency      Power
_____      _____

  1000        -3.0103
  2000        -64.949
  3000         -96.99
  4000        -104.95
4997.9        -306.11
5998.9        -310.56
```

The total harmonic distortion is approximately -62 dB. If you examine the power of the individual harmonics, you see that the major contribution comes from the harmonic at 2 kHz. The power at 2 kHz is approximately 62 dB below the power of the fundamental. The remaining harmonics do not contribute significantly to the total harmonic distortion. Additionally, the synthesized signal contains only four harmonics, including the fundamental. This is confirmed by the table, which shows a large power reduction after 4 kHz. Therefore, repeating the calculation with only four harmonics does not change the total harmonic distortion significantly.

Practical Introduction to Frequency-Domain Analysis

This example shows how to perform and interpret basic frequency-domain signal analysis. The example discusses the advantages of using frequency-domain versus time-domain representations of a signal and illustrates basic concepts using simulated and real data. The example answers basic questions such as: what is the meaning of the magnitude and phase of an FFT?, Is my signal periodic?, How do I measure power?, Is there one, or more than one signals in this band?.

Frequency-domain analysis is a tool of utmost importance in signal processing applications. Frequency-domain analysis is widely used in such areas as communications, geology, remote sensing, and image processing. While time-domain analysis shows how a signal changes over time, frequency-domain analysis shows how the signal's energy is distributed over a range of frequencies. A frequency-domain representation also includes information on the phase shift that must be applied to each frequency component in order to recover the original time signal with a combination of all the individual frequency components.

A signal can be converted between the time and frequency domains with a pair of mathematical operators called a transform. An example is the Fourier transform, which decomposes a function into the sum of a (potentially infinite) number of sine wave frequency components. The 'spectrum' of frequency components is the frequency domain representation of the signal. The inverse Fourier transform converts the frequency domain function back to a time function. The **fft** and **ifft** functions in MATLAB allow you to compute the Discrete Fourier transform (DFT) of a signal and the inverse of this transform respectively.

Magnitude and Phase Information of the FFT

The frequency-domain representation of a signal carries information about the signal's magnitude and phase at each frequency. This is why the output of the FFT computation is complex. A complex number, x, has a real, x_r, and an imaginary part, x_i, such that $x = x_r + ix_i$. The magnitude of x is computed as $\sqrt{(x_r^2 + x_i^2)}$, and the phase of x is computed as $\arctan(x_i/x_r)$. You can use MATLAB functions **abs** and **angle** to respectively get the magnitude and phase of any complex number.

Use an audio example to develop some insight on what information is carried by the magnitude and the phase of a signal. To do this, load an audio file containing 15 seconds of acoustic guitar music. The sample rate of the audio signal is 44.1 kHz.

```
Fs = 44100;
y = audioread('guitartune.wav');
```

Use **fft** to observe the frequency content of the signal.

```
NFFT = length(y);
Y = fft(y,NFFT);
F = ((0:1/NFFT:1-1/NFFT)*Fs).';
```

The output of the FFT is a complex vector containing information about the frequency content of the signal. The magnitude tells you the strength of the frequency components relative to other components. The phase tells you how all the frequency components align in time.

Plot the magnitude and the phase components of the frequency spectrum of the signal. The magnitude is conveniently plotted in a logarithmic scale (dB). The phase is unwrapped using the **unwrap** function so that we can see a continuous function of frequency.

```
magnitudeY = abs(Y);        % Magnitude of the FFT
phaseY = unwrap(angle(Y));  % Phase of the FFT

helperFrequencyAnalysisPlot1(F,magnitudeY,phaseY,NFFT)
```

Magnitude response of the audio signal

Phase response of the audio signal
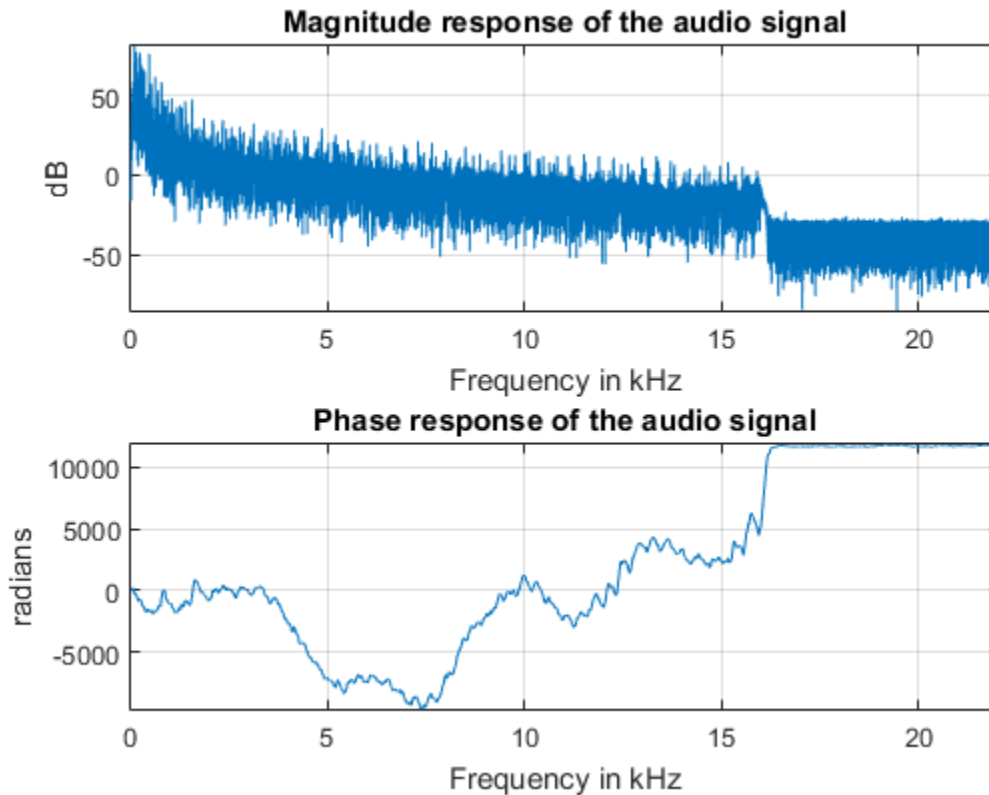
You can apply an inverse Fourier transform to the frequency domain vector, Y, to recover the time signal. The 'symmetric' flag tells **ifft** that you are dealing with a real-valued time signal so it will zero out the small imaginary components that appear on the inverse transform due to numerical inaccuracies in the computations. Notice that the original time signal, y, and the recovered signal, y1, are practically the same (the norm of their difference is on the order of 1e-14). The very small difference between the two is also due to the numerical inaccuracies mentioned above. Play and listen the un-transformed signal y1.

```
y1 = ifft(Y,NFFT,'symmetric');
norm(y-y1)
```

```
ans =
```

```
3.9112e-14
```

```
hplayer = audioplayer(y1, Fs);
play(hplayer);
```

To see the effects of changing the magnitude response of the signal, remove frequency components above 1 kHz directly from the FFT output (by making the magnitudes equal to zero) and listen to the effect this has on the sound of the audio file. Removing high frequency components of a signal is referred to as lowpass filtering.

```
Ylp = Y;
Ylp(F>=1000 & F<=Fs-1000) = 0;

helperFrequencyAnalysisPlot1(F,abs(Ylp),unwrap(angle(Ylp)),NFFT,...
  'Frequency components above 1 kHz have been zeroed')
```
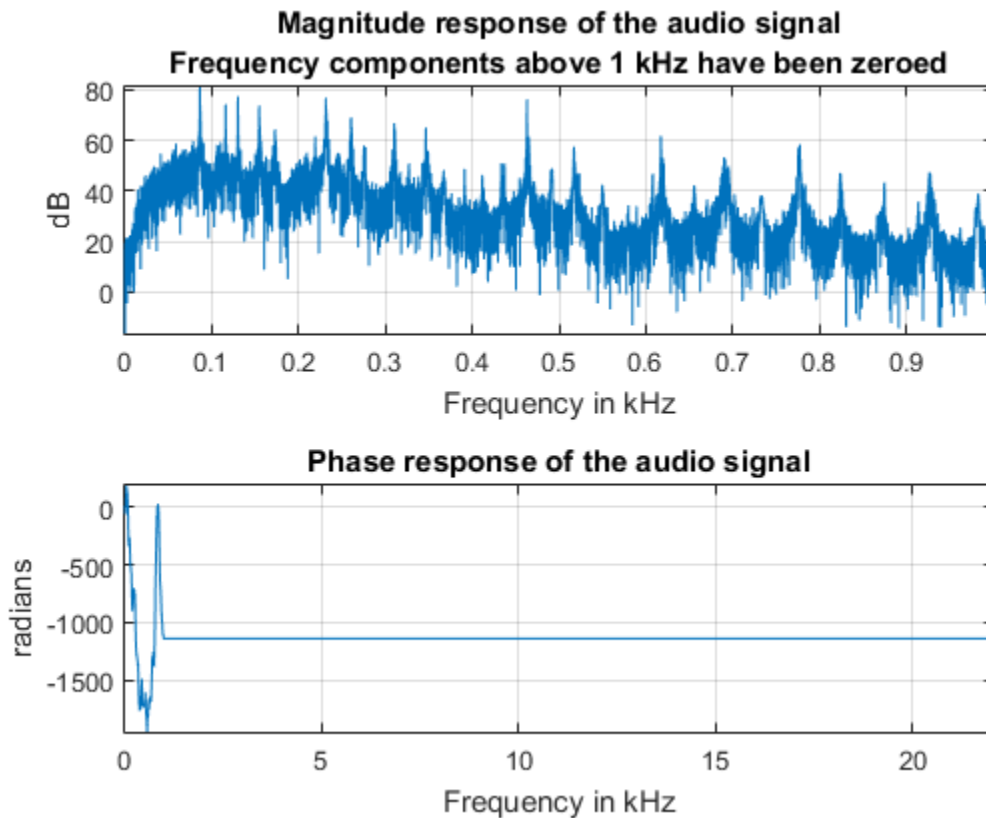
Get the filtered signal back into time domain using **ifft**.

```
ylp = ifft(Ylp,'symmetric');
```

Play the signal. You can still hear the melody but it sounds like if you had covered your ears (you filter high frequency sounds when you do this). Even though guitars produce notes that are between 400 and 1 kHz, as you play a note on a string, the string also vibrates at multiples of the base frequency. These higher frequency components, referred to as harmonics, are what give the guitar its particular tone. When you remove them, you make the sound seem "opaque".

```
hplayer = audioplayer(ylp, Fs);
play(hplayer);
```

The phase of a signal has important information about when in time the notes of the song appear. To illustrate the importance of phase on the audio signal, remove the phase information completely by taking the magnitude of each frequency component. Note that by doing this you keep the magnitude response unchanged.

```
% Take the magnitude of each FFT component of the signal
Yzp = abs(Y);
helperFrequencyAnalysisPlot1(F,abs(Yzp),unwrap(angle(Yzp)),NFFT,[],...
  'Phase has been set to zero')
```
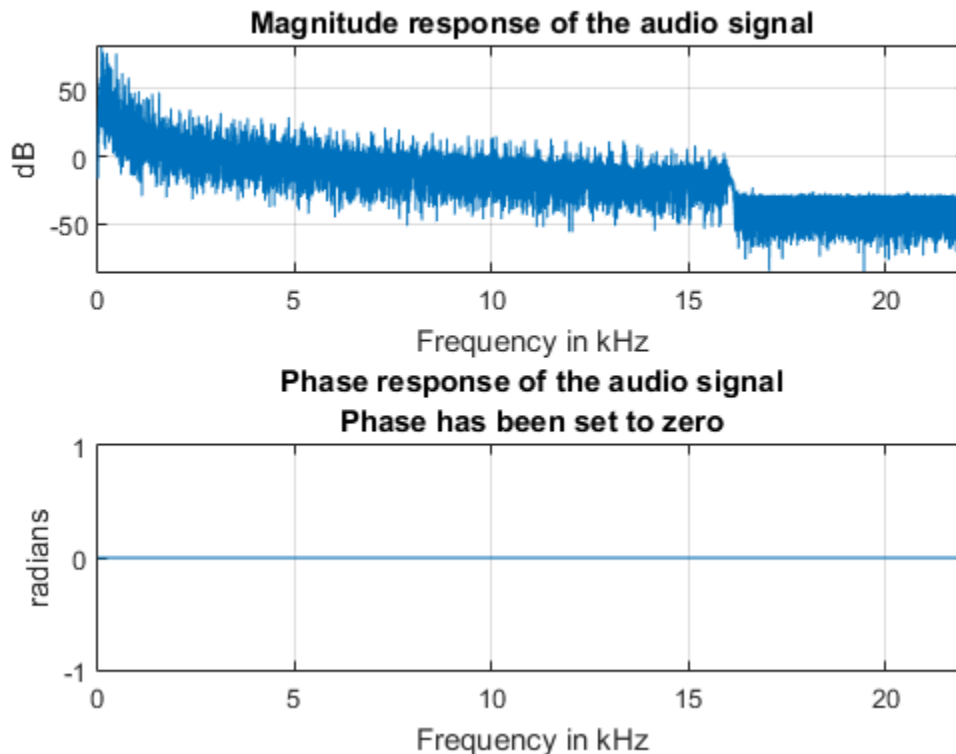

Get the signal back in the time domain and play the audio. You cannot recognize the original sound at all. The magnitude response is the same, no frequency components have been removed this time, but the order of the notes has disappeared completely. The

signal now consists of a group of sinusoids all aligned at time equal to zero. In general, phase distortions caused by filtering can damage a signal to the point of rendering it unrecognizable.

```
yzp = ifft(Yzp,'symmetric');
hplayer = audioplayer(yzp, Fs);
play(hplayer);
```

Finding Signal Periodicities

The frequency domain representation of a signal allows you to observe several characteristics of the signal that are either not easy to see, or not visible at all when you look at the signal in the time domain. For instance, frequency-domain analysis becomes useful when you are looking for cyclic behavior of a signal.

Analyzing Cyclic Behavior of the Temperature in an Office Building

Consider a set of temperature measurements in an office building during the winter season. Measurements were taken every 30 minutes for about 16.5 weeks. Look at the time domain data with the time axis scaled to weeks. Could there be any periodic behavior on this data?

```
load officetemp.mat
Fs = 1/(60*30);    % Sample rate is 1 sample every 30 minutes
t = (0:length(temp)-1)/Fs;

helperFrequencyAnalysisPlot2(t/(60*60*24*7),temp,...
  'Time in weeks','Temperature (Fahrenheit)')
```
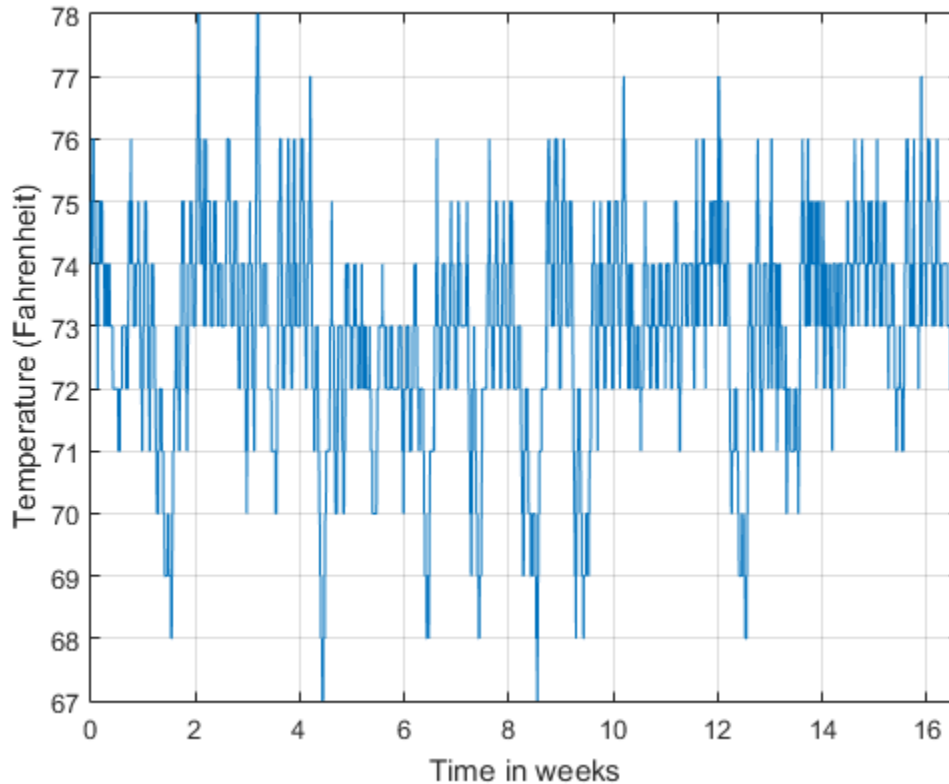
It is almost impossible to know if there is any cyclic behavior on the office temperatures by looking at the time-domain signal. However, the cyclic behavior of the temperature becomes evident if we look at its frequency-domain representation.

Obtain the frequency-domain representation of the signal. If you plot the magnitude of the FFT output with a frequency axis scaled to cycles/week, you can see that there are two spectral lines that are clearly larger than any other frequency component. One spectral line lies at 1 cycle/week, the other one lies at 7 cycles/week. This makes sense given that the data comes from a temperature-controlled building on a 7 day calendar. The first spectral line indicates that building temperatures follow a weekly cycle with lower temperatures on the weekends and higher temperatures during the week. The second line indicates that there is also a daily cycle with lower temperatures during the night and higher temperatures during the day.

```
NFFT = length(temp);                   % Number of FFT points
F = (0 : 1/NFFT : 1/2-1/NFFT)*Fs; % Frequency vector

TEMP = fft(temp,NFFT);
TEMP(1) = 0; % remove the DC component for better visualization

helperFrequencyAnalysisPlot2(F*60*60*24*7,abs(TEMP(1:NFFT/2)),...
  'Frequency (cycles/week)','Magnitude',[],[],[0 10])
```
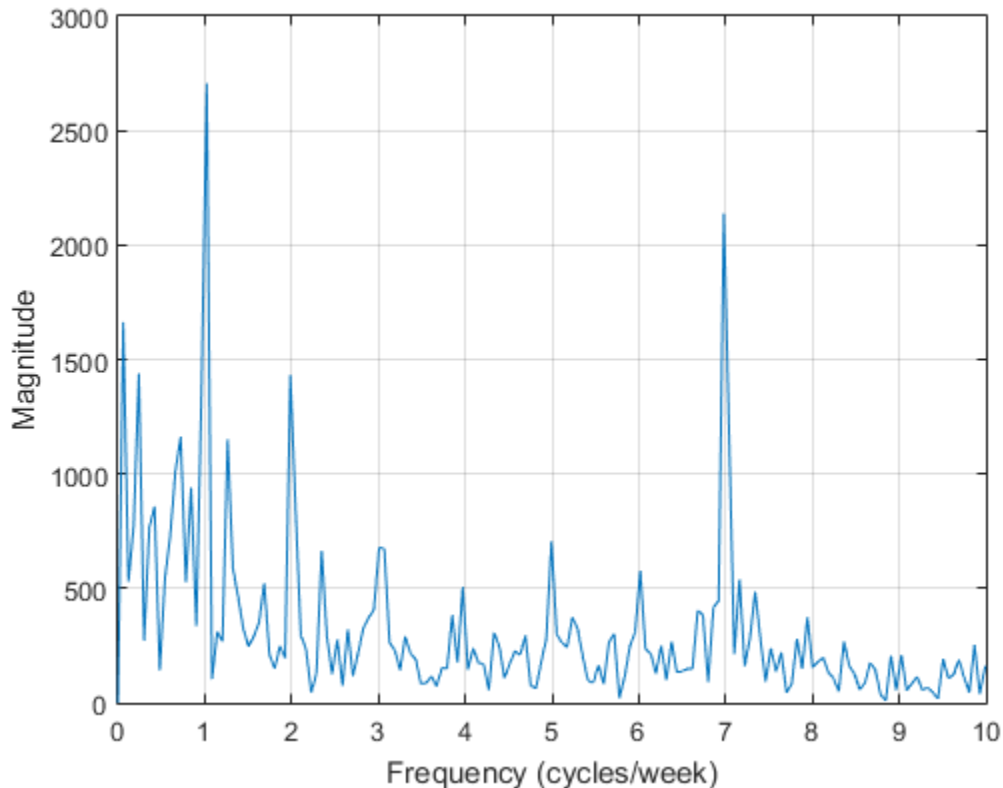

Measuring Power

The **periodogram** function computes the signal's FFT and normalizes the output to obtain a power spectral density, PSD, or a power spectrum from which you can measure

power. The PSD describes how the power of a time signal is distributed with frequency, it has units of watts/Hz. You compute the power spectrum by integrating each point of the PSD over the frequency interval at which that point is defined (i.e. over the resolution bandwidth of the PSD). The units of the power spectrum are watts. You can read power values directly from the power spectrum without having to integrate over an interval. Note that the PSD and power spectrum are real, so they do not contain any phase information.

Measuring Harmonics at the Output of a Non-Linear Power Amplifier

Load the data measured at the output of a power amplifier that has third order distortion of the form $v_o = v_i + 0.75v_i^2 + 0.5v_i^3$, where v_o is the output voltage and v_i is the input voltage. The data was captured with a sample rate of 3.6 kHz. The input v_i consists of a 60 Hz sinusoid with unity amplitude. Due to the nature of the non-linear distortion, you should expect the amplifier output signal to contain a DC component, a 60 Hz component, and second and third harmonics at 120 and 180 Hz.

Load 3600 samples of the amplifier output, compute the power spectrum, and plot the result in a logarithmic scale (decibels-watts or dBW).

```
load ampoutput1.mat
Fs = 3600;
NFFT = length(y);

% Power spectrum is computed when you pass a 'power' flag input
[P,F] = periodogram(y,[],NFFT,Fs,'power');

helperFrequencyAnalysisPlot2(F,10*log10(P),'Frequency in Hz',...
  'Power spectrum (dBW)',[],[],[-0.5 200])
```
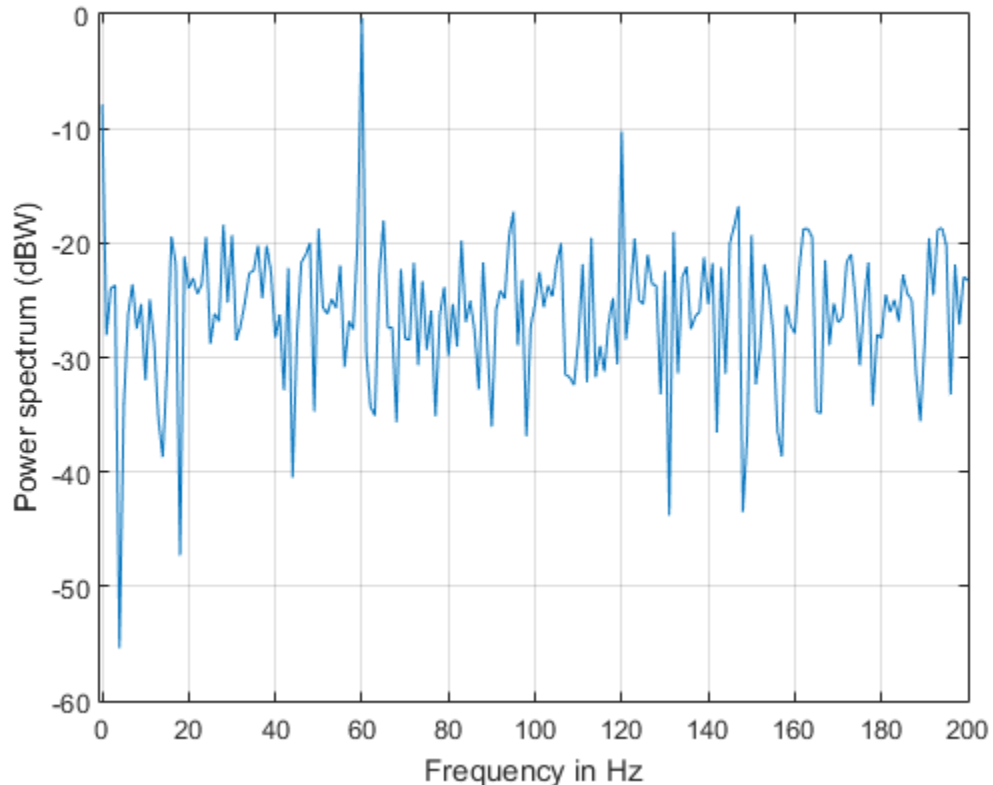
The plot of the power spectrum shows three of the four expected peaks at DC, 60, and 120 Hz. It also shows several more spurious peaks that must be caused by noise in the signal. Note that the 180 Hz harmonic is completely buried in the noise.

Measure the power of the visible expected peaks:

```
PdBW = 10*log10(P);
power_at_DC_dBW = PdBW(F==0)      % dBW

[peakPowers_dBW, peakFreqIdx] = findpeaks(PdBW,'minpeakheight',-11);
peakFreqs_Hz = F(peakFreqIdx)
peakPowers_dBW
```

```
power_at_DC_dBW =

   -7.8873


peakFreqs_Hz =

    60
   120


peakPowers_dBW =

   -0.3175
  -10.2547
```

Improving Power Measurements for Noisy Signals

As seen on the plot above, the periodogram shows several frequency peaks that are not related to the signal of interest. The spectrum looks very noisy. The reason for this is that you only analyzed one short realization of the noisy signal. Repeating the experiment several times and averaging would remove the spurious spectral peaks and yield more accurate power measurements. You can achieve this averaging using the **pwelch** function. This function will take a large data vector, break it into smaller segments of a specified length, compute as many periodograms as there are segments, and average them. As the number of available segments increases, the **pwelch** function will yield a smoother power spectrum (less variance) with power values closer to the expected values.

Load a larger observation consisting of 500e3 points of the amplifier output. Keep the number of points used to perform the FFTs as 3600 so that floor(500e3/3600) = 138 FFTs are averaged to obtain the power spectrum.

```
load ampoutput2.mat
SegmentLength = NFFT;

% Power spectrum is computed when you pass a 'power' flag input
[P,F] = pwelch(y,ones(SegmentLength,1),0,NFFT,Fs,'power');

helperFrequencyAnalysisPlot2(F,10*log10(P),'Frequency in Hz',...
  'Power spectrum (dBW)',[],[],[-0.5 200])
```
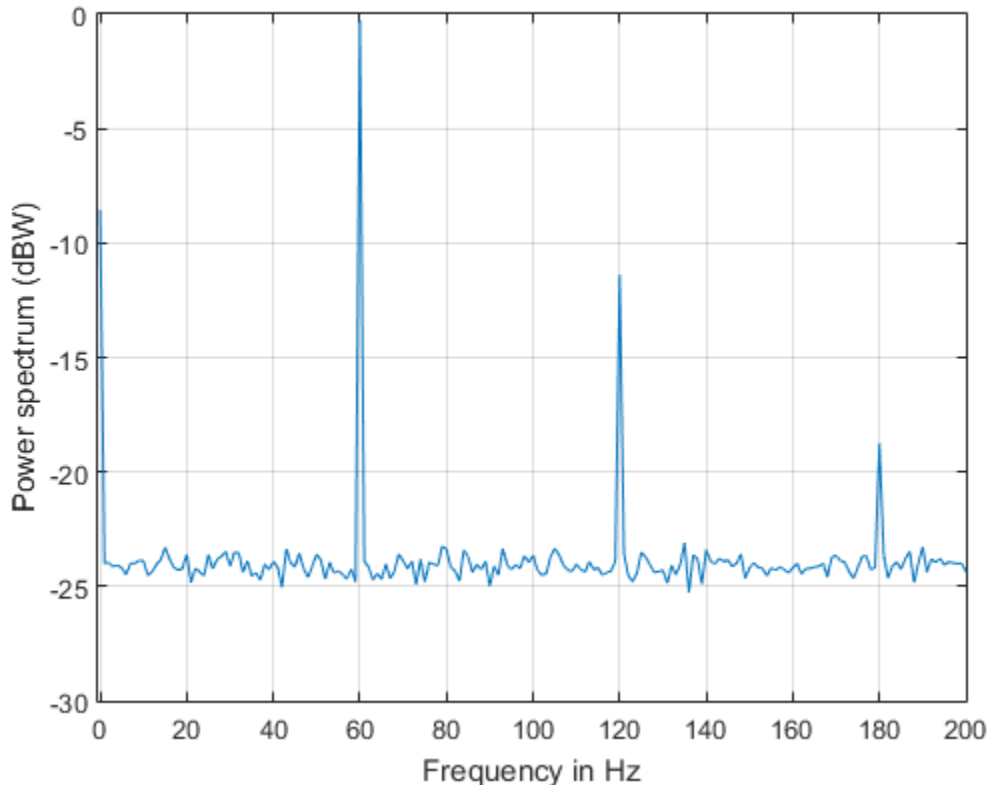
As seen on the plot, **pwelch** effectively removes all the spurious frequency peaks caused by noise. The spectral component at 180 Hz that was buried in noise is now visible. Averaging removes variance from the spectrum and this effectively yields more accurate power measurements.

Measuring Total Average Power and Power Over a Frequency Band

Measuring the total average power of a time-domain signal is an easy and common task. For the amplifier output signal, y, the total average power is computed in the time domain as:

```
pwr = sum(y.^2)/length(y) % in watts
```

```
pwr =

    8.1697
```

In the frequency-domain, the total average power is computed as the sum of the power of all the frequency components of the signal. The value of pwr1 consists of the sum of all the frequency components available in the power spectrum of the signal. The value agrees with the value of pwr computed above using the time domain signal:

```
pwr1 = sum(P) % in watts


pwr1 =

    8.1698
```

But what if you wanted to measure the total power available over a band of frequencies? You can use the **bandpower** function to compute the power over any desired frequency band. You can pass the time-domain signal directly as an input to this function to obtain the power over a specified band. In this case, the function will estimate the power spectrum with the periodogram method.

Compute the power over the 50 Hz to 70 Hz band. The result will include the 60 Hz power plus the noise power over the band of interest:

```
pwr_band = bandpower(y,Fs,[50 70]);
pwr_band_dBW = 10*log10(pwr_band) % dBW


pwr_band_dBW =

    0.0341
```

If you want to control the computation of the power spectrum used to measure the power in a band, you can pass a PSD vector to the **bandpower** function. For instance, you can use the **pwelch** function as you did before to compute the PSD and ensure averaging of the noise effects:

```
% Power spectral density is computed when you specify the 'psd' option
[PSD,F]  = pwelch(y,ones(SegmentLength,1),0,NFFT,Fs,'psd');
pwr_band1 = bandpower(PSD,F,[50 70],'psd');
```

12-61

```
pwr_band_dBW1 = 10*log10(pwr_band1) % dBW


pwr_band_dBW1 =

    0.0798
```

Finding Spectral Components

A signal might be composed of one more frequency components. The ability to observe all the spectral components depends on the frequency resolution of your analysis. The frequency resolution or resolution bandwidth of the power spectrum is defined as R = Fs/N, where N is the length of the signal observation. Only spectral components separated by a frequency larger than the frequency resolution will be resolved.

Analyzing a Building's Earthquake Vibration Control System

Active Mass Driver (AMD) control systems are used to reduce vibration in a building under an earthquake. An active mass driver is placed on the top floor of the building and, based on displacement and acceleration measurements of the building floors, a control system sends signals to the driver so that the mass moves to attenuate ground disturbances. Acceleration measurements were recorded on the first floor of a three story test structure under earthquake conditions. Measurements were taken without the active mass driver control system (open loop condition), and with the active control system (closed loop condition).

Load the acceleration data and compute the power spectrum for the acceleration of the first floor. The length of the data vectors is 10e3 and the sample rate is 1 kHz. Use **pwelch** with segments of length 64 data points to obtain floor(10e3/64) = 156 FFT averages and a resolution bandwidth of Fs/64 = 15.625 Hz. As was shown before, averaging reduces noise effects and yields more accurate power measurements. Use 512 FFT points. Using NFFT > N effectively interpolates frequency points rendering a more detailed spectrum plot (this is achieved by appending NFFT-N zeros at the end of the time signal and taking the NFFT-point FFT of the zero padded vector).

The open loop and close loop acceleration power spectra show that when the control system is active, the acceleration power spectrum decreases between 4 and 11 dB. The maximum attenuation occurs at about 23.44 kHz. An 11 dB reduction means that the vibration power is reduced by a factor of 12.6. The total power is reduced from 0.1670 to 0.059 watts, a factor of 2.83.

```
load quakevibration.mat
```

```
Fs = 1e3;                    % sample rate
NFFT = 512;                  % number of FFT points
segmentLength = 64;          % segment length

% open loop acceleration power spectrum
[P1_OL,F] = pwelch(gfloor1OL,ones(segmentLength,1),0,NFFT,Fs,'power');

% closed loop acceleration power spectrum
P1_CL     = pwelch(gfloor1CL,ones(segmentLength,1),0,NFFT,Fs,'power');

helperFrequencyAnalysisPlot2(F,10*log10([(P1_OL) (P1_CL)]),...
  'Frequency in Hz','Acceleration Power Spectrum in dB',...
  'Resolution bandwidth = 15.625 Hz',{'Open loop', 'Closed loop'},[0 100])
```
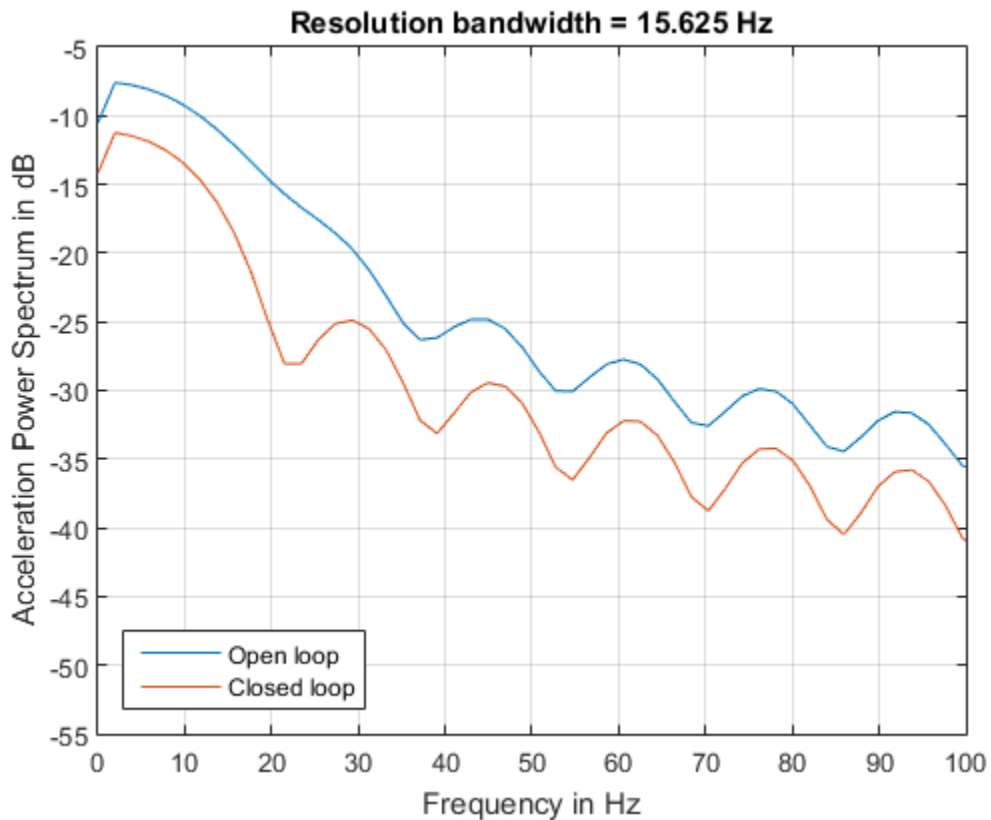
You are analyzing vibration data and you know that vibrations have a cyclic behavior. Then how is it that the spectrum plots shown above do not contain any sharp spectral lines typical of cyclic behavior? Maybe you are missing those lines because they are not resolvable with the resolution obtained with 64 point segment lengths? Increase the frequency resolution to see if there are spectral lines that were not resolvable before. Do this by increasing the data segment length used in the **pwelch** function to 512 points. This yields a new resolution of Fs/512 = 1.9531 Hz. In this case, the number of FFT averages is reduced to floor(10e3/512) = 19. Clearly, there is a trade-off between number of averages and frequency resolution when using **pwelch**. Keep the number of FFT points equal to 512.

```
NFFT = 512;             % number of FFT points
segmentLength = 512;    % segment length

[P1_OL,F] = pwelch(gfloor1OL,ones(segmentLength,1),0,NFFT,Fs,'power');
P1_CL     = pwelch(gfloor1CL,ones(segmentLength,1),0,NFFT,Fs,'power');

helperFrequencyAnalysisPlot2(F,10*log10([(P1_OL) (P1_CL)]),...
  'Frequency in Hz','Acceleration Power Spectrum in dB',...
  'Resolution bandwidth = 1.95 Hz',{'Open loop', 'Closed loop'},[0 100])
```
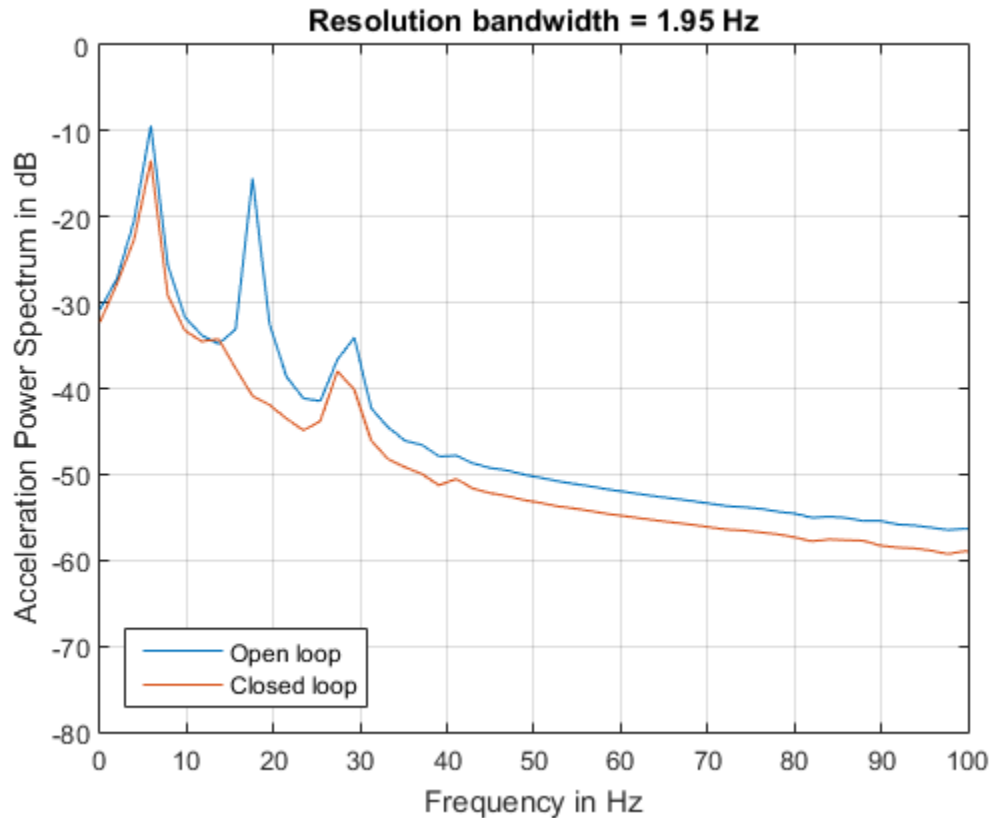
Notice how the increase in frequency resolution allows you to observe three peaks on the open loop spectrum and two on the close loop spectrum. These peaks were not resolvable before. The separation between the peaks on the open loop spectrum is about 11 Hz which is smaller than the frequency resolution obtained with segments of length 64 but larger than the resolution obtained with segments of length 512. The cyclic behavior of the vibrations is now visible. The main vibration frequency is at 5.86 Hz, and the equispaced frequency peaks suggest that they are harmonically related. While it has already been observed that the control system reduces the overall power of the vibrations, the higher resolution spectra shows that another effect of the control system is to notch the harmonic component at 17.58 Hz. So the control system not only reduces the vibration but also brings it closer to a sinusoid.

It is important to note that frequency resolution is determined by the number of signal points, not by the number of FFT points. Increasing the number of FFT points interpolates the frequency data to give you more details on the spectrum but it does not improve resolution.

Conclusions

In this example you learned how to perform frequency-domain analysis of a signal using the **fft**, **ifft**, **periodogram**, **pwelch**, and **bandpower** functions. You understood the complex nature of the FFT and what is the information contained in the magnitude and the phase of the frequency spectrum. You saw the advantages of using frequency domain data when analyzing the periodicity of a signal. You learned how compute the total power or power over a particular band of frequencies of a noisy signal. You understood how increasing the frequency resolution of the spectrum allows you to observe closely spaced frequency components and you learned about the tradeoff between frequency resolution and spectral averaging.

Further Reading

For more information on frequency domain analysis see the Signal Processing Toolbox.

Reference: J.G. Proakis and D. G. Manolakis, "Digital Signal Processing. Principles, Algorithms, and Applications", Prentice Hall, 1996.

Appendix

The following helper functions are used in this example.

- helperFrequencyAnalysisPlot1.m
- helperFrequencyAnalysisPlot2.m

Spectral Analysis of Nonuniformly Sampled Signals

This example shows how to perform spectral analysis on nonuniformly sampled signals. It helps you determine if a signal is uniformly sampled or not, and if not, it shows how to compute its spectrum or its power spectral density.

The example introduces the Lomb-Scargle periodogram, which can compute spectra of nonuniformly sampled signals.

Nonuniformly Sampled Signals

Nonuniformly sampled signals are often found in the automotive industry, in communications, and in fields as diverse as medicine and astronomy. Nonuniform sampling might be due to imperfect sensors, mismatched clocks, or event-triggered phenomena.

The computation and study of spectral content is an important part of signal analysis. Conventional spectral analysis techniques like the periodogram and the Welch method require the input signal to be uniformly sampled. When the sampling is nonuniform, one can resample or interpolate the signal onto a uniform sample grid. This, however, can add undesired artifacts to the spectrum and might lead to analysis errors.

A better alternative is to use the Lomb-Scargle method, which works directly with the nonuniform samples and thus makes it unnecessary to resample or interpolate. The algorithm has been implemented in the `plomb` function.

Spectral Analysis of Signals with Missing Data

Consider a temperature monitoring system in which a microcontroller records the temperature of a room and transmits this reading every 15 minutes to a cloud-based server that stores it. It is known that glitches in internet connectivity prevent the cloud-based system from receiving some of the readings sent by the microcontroller. Also, at least once during the measurement period the microcontroller's battery ran out, leading to a large gap in the sampling.

Load the temperature readings and the corresponding timestamps.

```
load('nonuniformdata.mat','roomtemp','t1')

figure
plot(t1/(60*60*24*7),roomtemp,'LineWidth',1.2)

grid
```

```
xlabel('Time (weeks)')
ylabel('Temperature (\circF)')
```
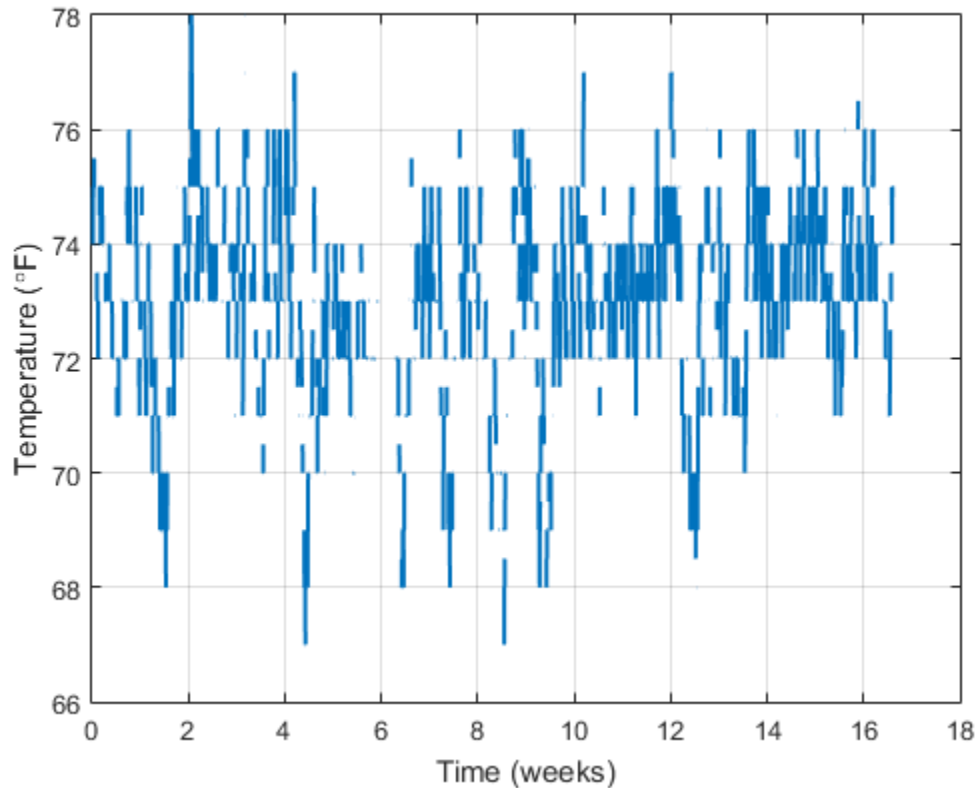

An easy way to determine if a signal is uniformly sampled is to take a histogram of the intervals between successive sample times.

Plot a histogram of sampling intervals (time differences) in minutes. Include only points at which samples are present.

```
tAtPoints = t1(~isnan(roomtemp))/60;
TimeIntervalDiff = diff(tAtPoints);

figure
hist(TimeIntervalDiff,0:100)
grid
```

```
xlabel('Sampling intervals (minutes)')
ylabel('Occurrences')
xlim([10 100])
```
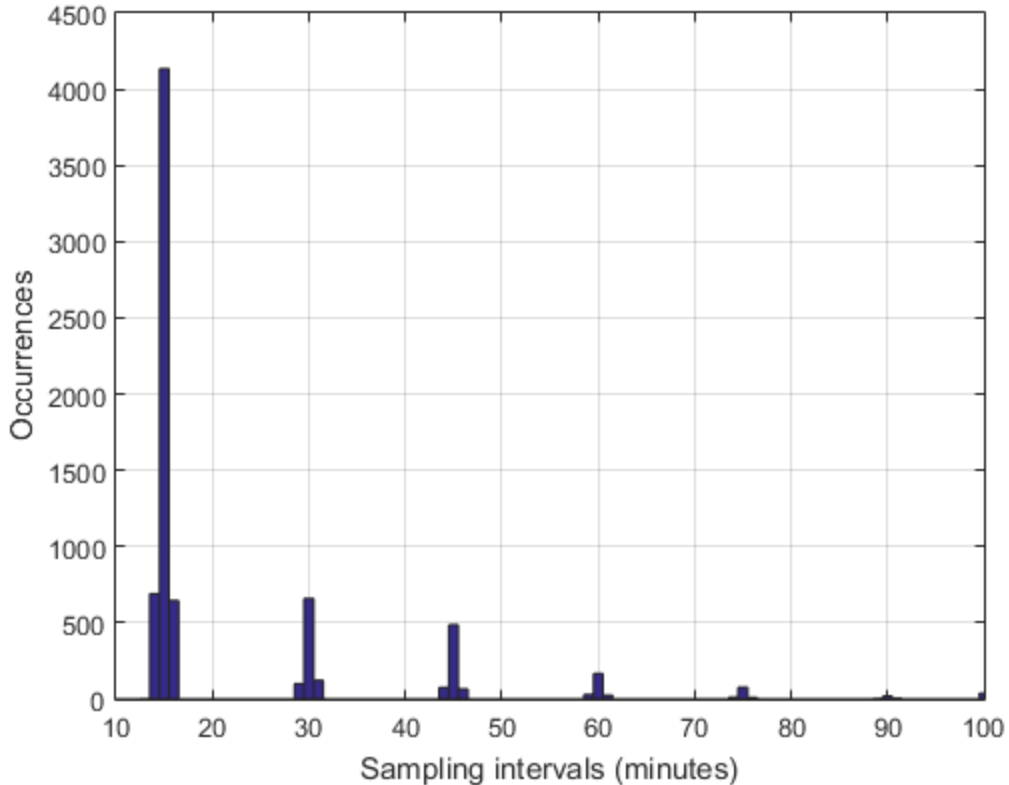

The majority of the measurements are spaced approximately 15 minutes apart, as expected. However, a fair number of occurrences have sampling intervals of around 30 and 45 minutes, which correspond to one or two consecutive dropped samples. This causes the signal to be nonuniformly sampled. Furthermore, the histogram shows some jitter surrounding the bars showing high occurrences. This could relate to TCP/IP latency.

Use the Lomb-Scargle method to compute and visualize the spectral content of the signal. To help visualize the spectrum better, consider frequencies up to 0.02 mHz, which correspond to about 13 cycles per week.

```
[Plomb,flomb] = plomb(roomtemp,t1,2e-5,'power');

figure
plot(flomb*60*60*24*7,Plomb)
grid
xlabel('Frequency (cycles/week)')
ylabel('Power (dBW)')
```
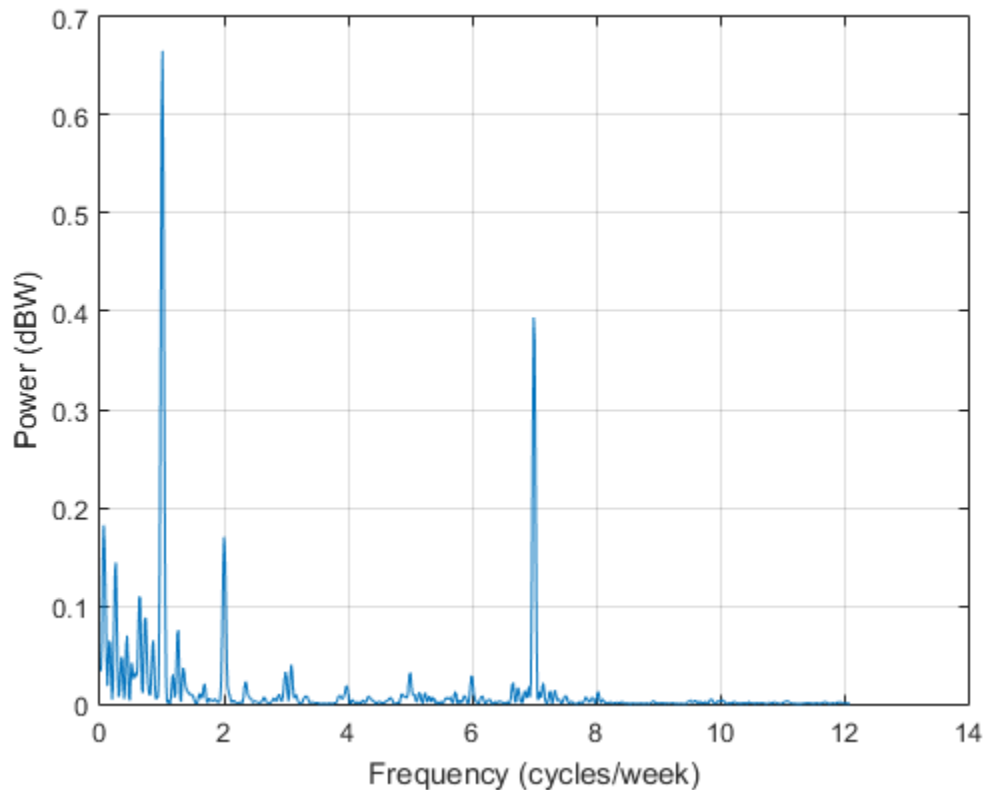

The spectrum shows dominant periodicities at 7 cycles per week and 1 cycle per week. This is understandable, given that the data comes from a temperature-controlled building on a seven-day calendar. The spectral line showing a peak at 1 cycle per week indicates that the temperature in the building follows a weekly cycle, with lower temperatures on weekends and higher temperatures during the week. The spectral line

of 7 cycles per week indicates that there is also a daily cycle with lower temperatures at night and higher temperatures during the day.

Spectral Analysis of Signals with Unevenly Spaced Samples

Heart-rate variability (HRV) signals, which represent the physiological variation in time between heartbeats, are typically unevenly sampled because human heart rates are not constant. HRV signals are derived from electrocardiogram (ECG) readings.

The sample points of an HRV signal are located at the R-Peak times of the ECG. The amplitude of each point is computed as the inverse of the time difference between consecutive R-Peaks and is placed at the instant of the second R-Peak.

```matlab
% Load the signal, the timestamps, and the sample rate
load('nonuniformdata.mat','ecgsig','t2','Fs')

% Find the ECG peaks
[pks,locs] = findpeaks(ecgsig,Fs, ...
    'MinPeakProminence',0.3,'MinPeakHeight',0.2);

% Determine the RR intervals
RLocsInterval = diff(locs);

% Derive the HRV signal
tHRV = locs(2:end);
HRV = 1./RLocsInterval;

% Plot the signals
figure
a1 = subplot(2,1,1);
plot(t2,ecgsig,'b',locs,pks,'*r')
grid
a2 = subplot(2,1,2);
plot(tHRV,HRV)
grid
xlabel(a2,'Time(s)')
ylabel(a1,'ECG (mV)')
ylabel(a2,'HRV (Hz)')
```
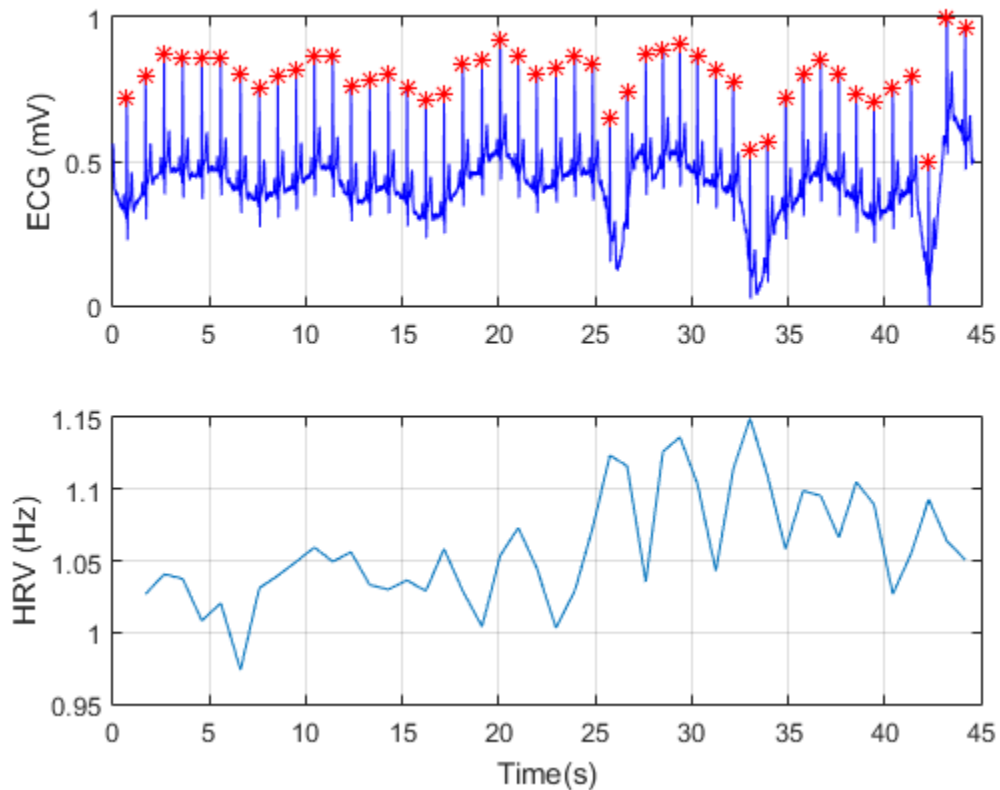
The varying intervals between the R-peaks cause the sample-time nonuniformity in the HRV data. Consider the peak locations of the signal and plot a histogram of their separations in seconds.

```
figure
hist(RLocsInterval)

grid
xlabel('Sampling interval (s)')
ylabel('RR distribution')
```
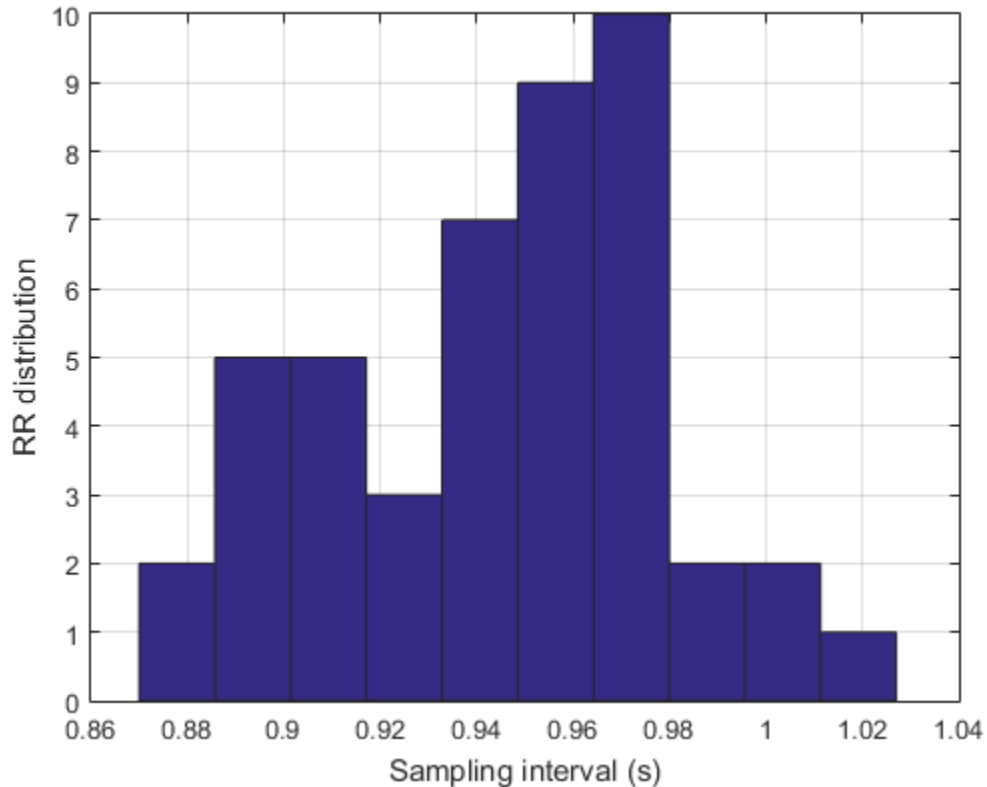
The typical frequency bands of interest in HRV spectra are:

- Very Low Frequency (VLF), from 3.3 to 40 mHz,
- Low Frequency (LF), from 40 to 150 mHz,
- High Frequency (HF), from 150 to 400 mHz.

These bands approximately confine the frequency ranges of the distinct biological regulatory mechanisms that contribute to HRV. Fluctuations in any of these bands have biological significance.

Use `plomb` to calculate the spectrum of the HRV signal.

```
figure
```

```
plomb(HRV,tHRV,'Pd',[0.95, 0.5])
```
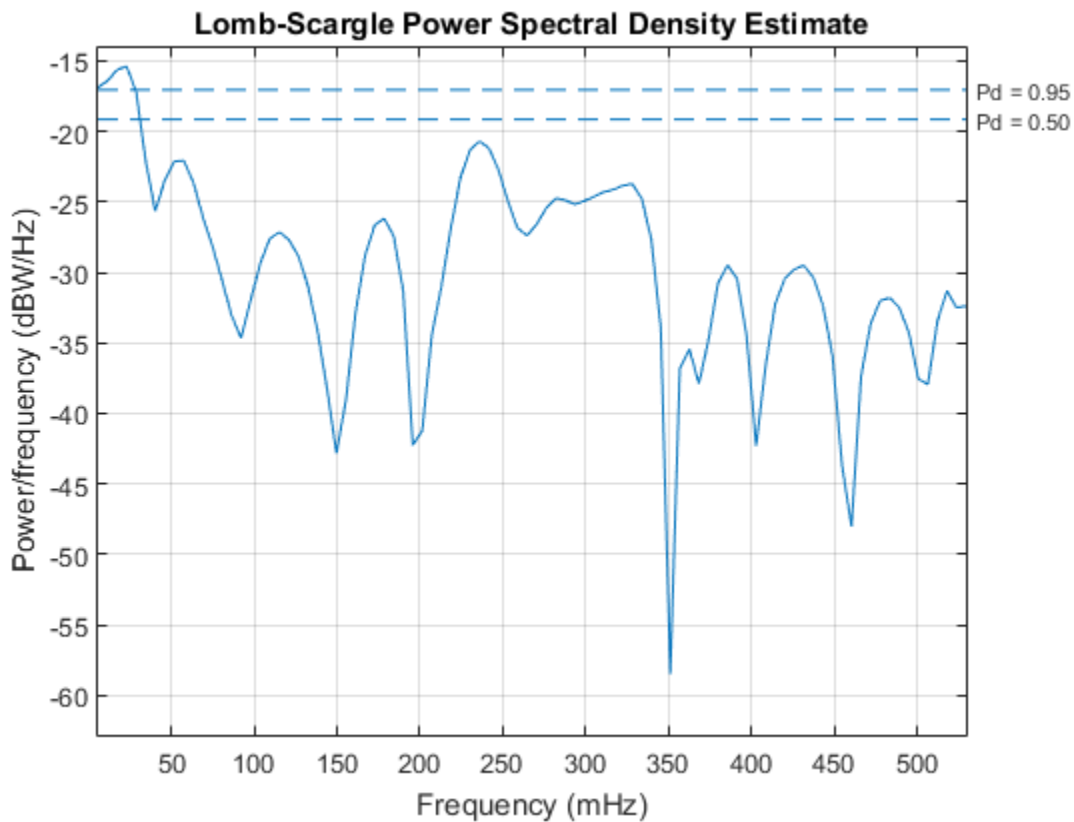

The dashed lines denote 95% and 50% detection probabilities. These thresholds measure the statistical significance of peaks. The spectrum shows peaks in all three bands of interest listed above. However, only the peak located at 23.2 mHz in the VLF range shows a detection probability 95%, while the other peaks have detection probabilities of less than 50%. The peaks lying below 40 mHz are thought to be due to long-term regulatory mechanisms, such as the thermoregulatory system and hormonal factors.

Measure Mean Frequency, Power, Bandwidth

Generate 1024 samples of a chirp sampled at 1024 kHz. The chirp has an initial frequency of 50 kHz and reaches 100 kHz at the end of the sampling. Add white Gaussian noise such that the signal-to-noise ratio is 40 dB.

```
nSamp = 1024;
Fs = 1024e3;
SNR = 40;

t = (0:nSamp-1)'/Fs;

x = chirp(t,50e3,nSamp/Fs,100e3);
x = x+randn(size(x))*std(x)/db2mag(SNR);
```

Estimate the 99% occupied bandwidth of the signal and annotate it on a plot of the power spectral density (PSD).

```
obw(x,Fs);
```
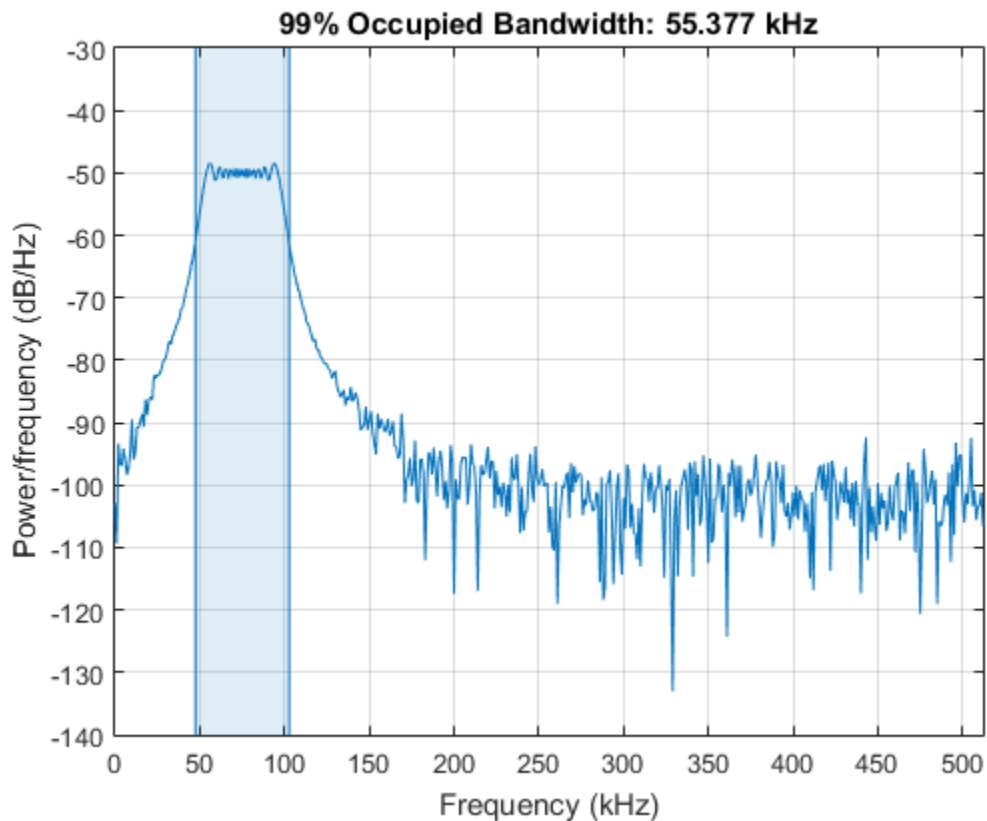
Compute the power in the band and verify that it is 99% of the total.

```
[bw,flo,fhi,powr] = obw(x,Fs);
```

```
pcent = powr/bandpower(x)*100
```

```
pcent =

    99
```

Generate another chirp. Specify an initial frequency of 200 kHz, a final frequency of 300 kHz, and an amplitude that is twice that of the first signal. Add white Gaussian noise.

```
x2 = 2*chirp(t,200e3,nSamp/Fs,300e3);
x2 = x2+randn(size(x2))*std(x2)/db2mag(SNR);
```

Add the two chirps to form a new signal. Plot the PSD of the signal and annotate its median frequency.

```
medfreq([x+x2],Fs);
```
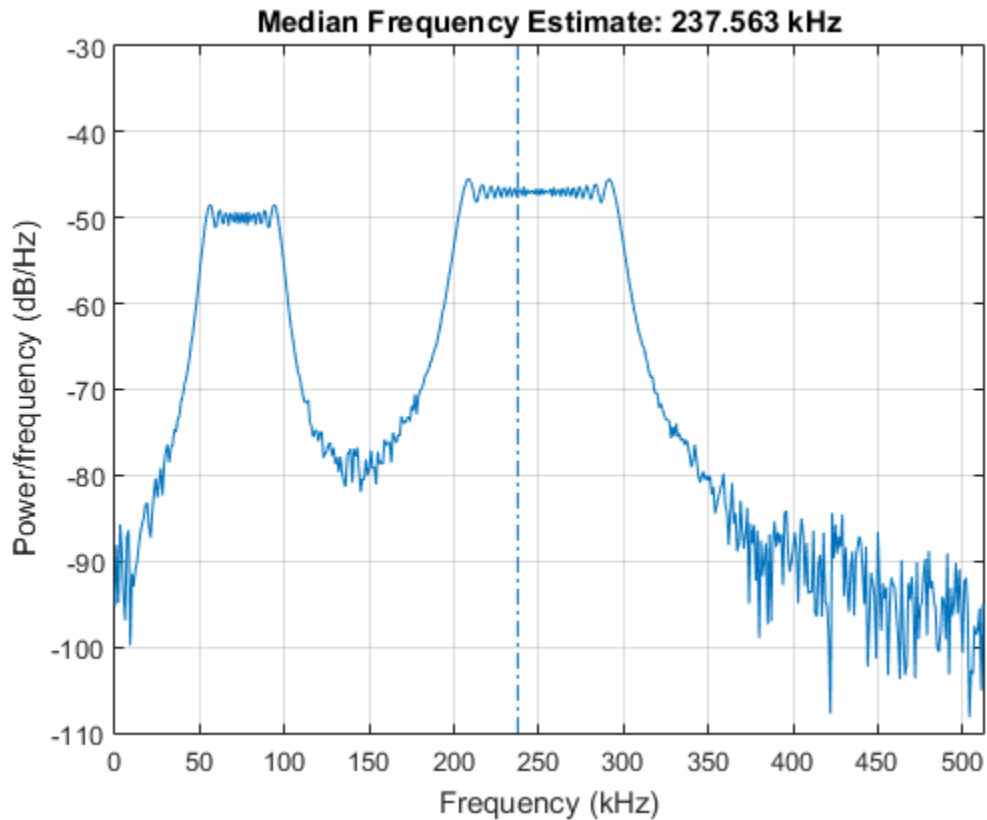

Plot the PSD and annotate the mean frequency.

```
meanfreq([x+x2],Fs);
```
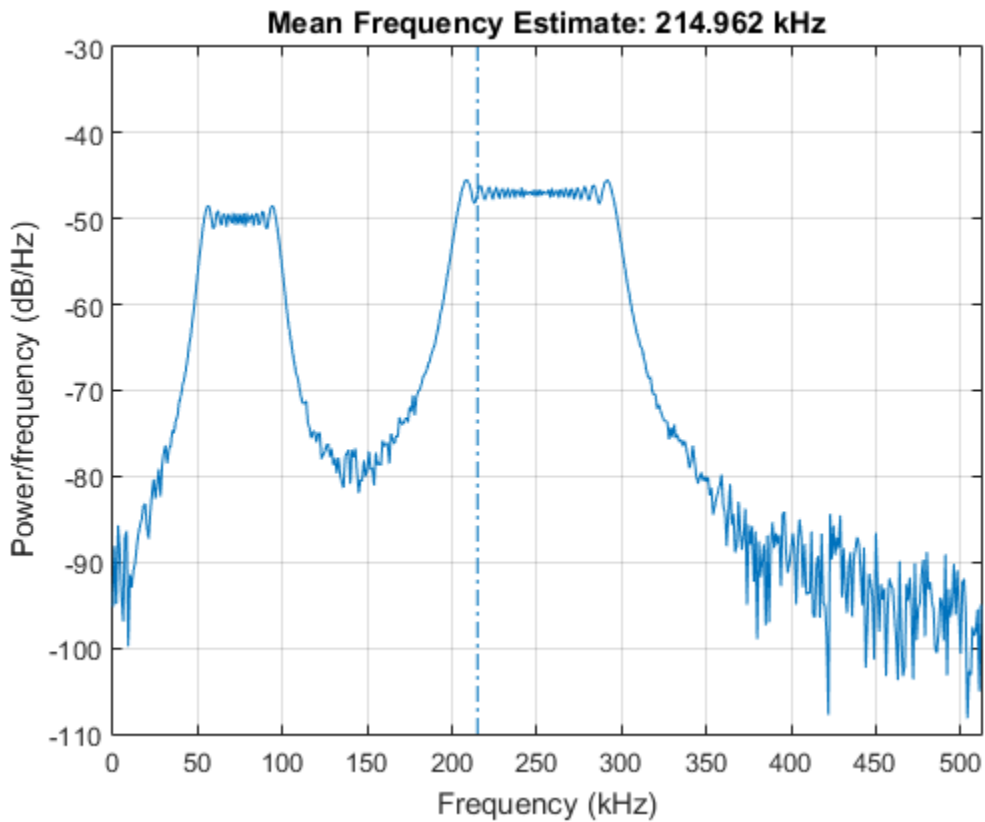
Now consider each chirp to represent a separate channel. Estimate the mean frequency of each channel. Annotate the mean frequencies on a plot of the PSDs.

```
meanfreq([x x2],Fs)
```

```
ans =

   1.0e+05 *

    0.7503    2.4999
```
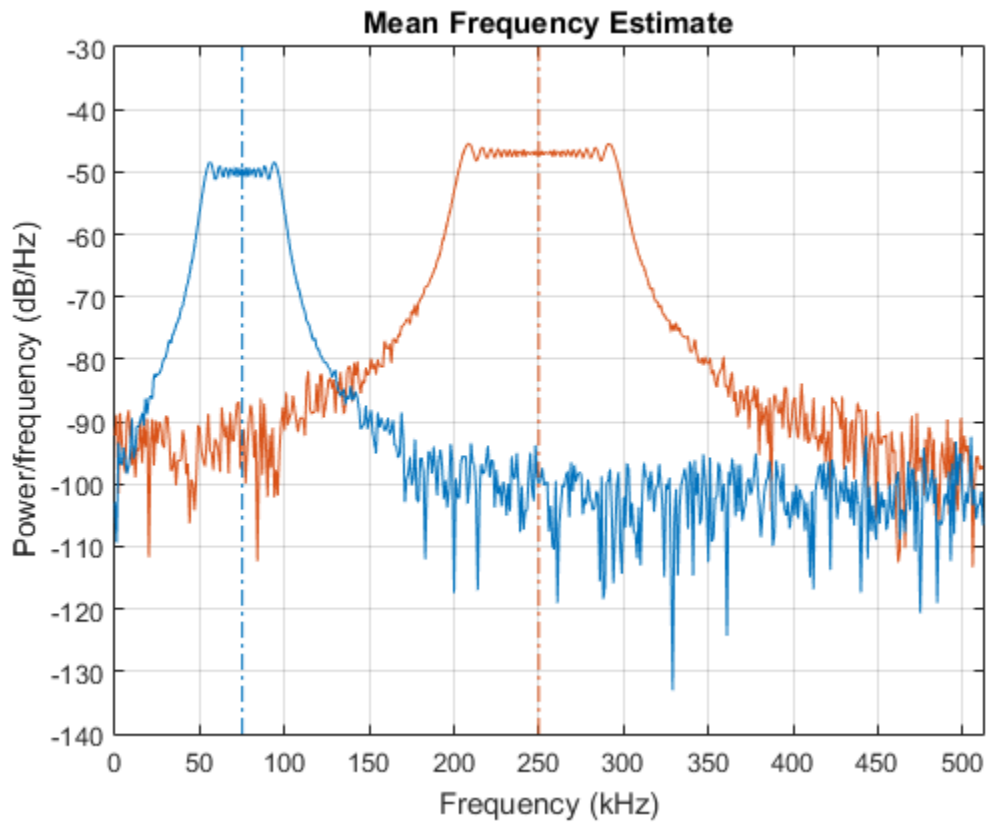
Estimate the half-power bandwidth of each channel. Annotate the 3-dB bandwidths on a plot of the PSDs.

```
powerbw([x x2],Fs)
```

```
ans =

   1.0e+04 *

   4.4386    9.2208
```

3 dB Bandwidth
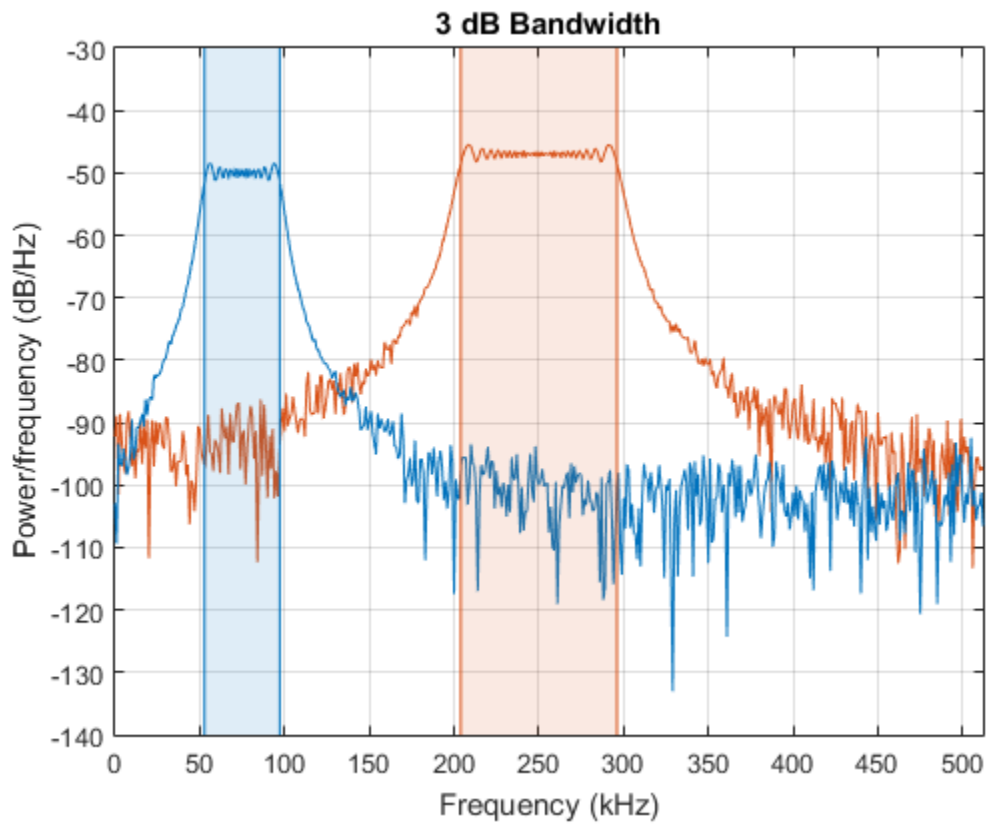
Periodogram of Data Set with Missing Samples

Galileo Galilei observed the motion of Jupiter's four largest satellites during the winter of 1610. When the weather allowed, Galileo recorded the satellites' locations. Use his observations to estimate the orbital period of one of the satellites, Callisto.

Callisto's angular position is measured in minutes of arc. Missing data due to cloudy conditions are specified using NaN's. The first observation is dated January 15.

```
yg = [10.5 NaN 11.5 10.5 NaN NaN NaN -5.5 -10.0 -12.0 -11.5 -12.0 -7.5 ...
    NaN NaN NaN NaN 8.5 12.5 12.5 10.5 NaN NaN NaN -6.0 -11.5 -12.5 ...
    -12.5 -10.5 -6.5 NaN 2.0 8.5 10.5 NaN 13.5 NaN 10.5 NaN NaN NaN ...
    -8.5 -10.5 -10.5 -10.0 -8.0]';

plot(yg,'o')

ax = gca;
nights = [1 18 32 46];
ax.XTick = nights;
ax.XTickLabel = char(datetime(1610,1,nights+14));
```
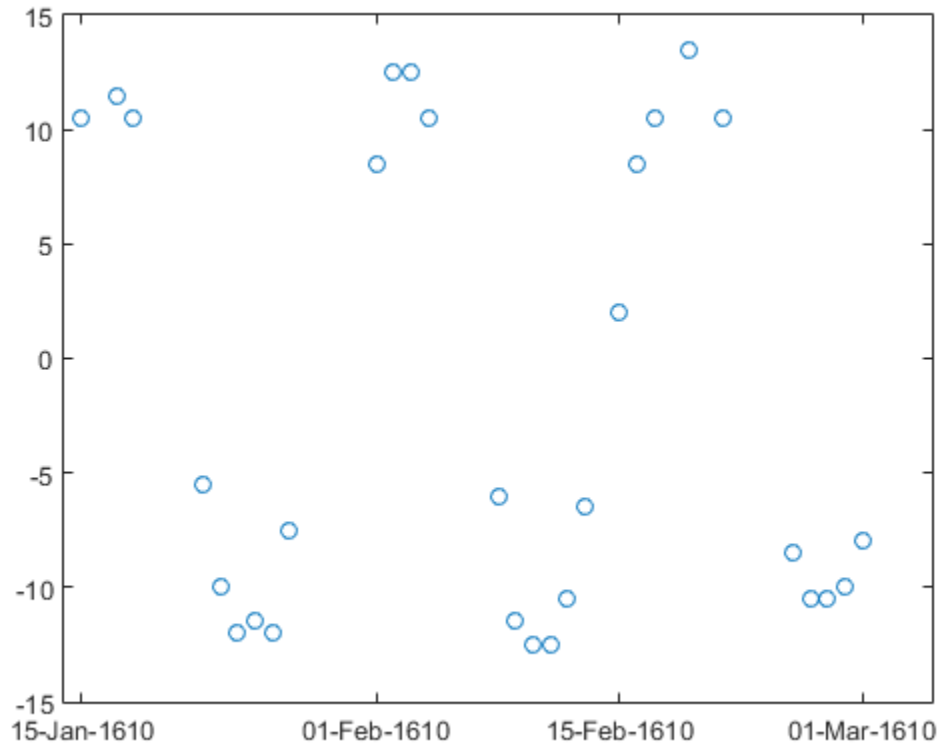
Estimate the power spectrum of the data using `plomb`. Investigate frequencies up to half the sample rate of one observation per day. Specify an oversampling factor of 10.

```
Fs = 1;
Fmax = Fs/2;

[pxx,f] = plomb(yg,Fs,Fmax,10,'power');
```

Use `findpeaks` to determine the location of the only prominent peak of the spectrum. Plot the power spectrum and show the peak.

```
[pk,f0] = findpeaks(pxx,f,'MinPeakHeight',10);

plot(f,pxx,f0,pk,'o')
```

```
xlabel('Frequency (day^{-1})')
```
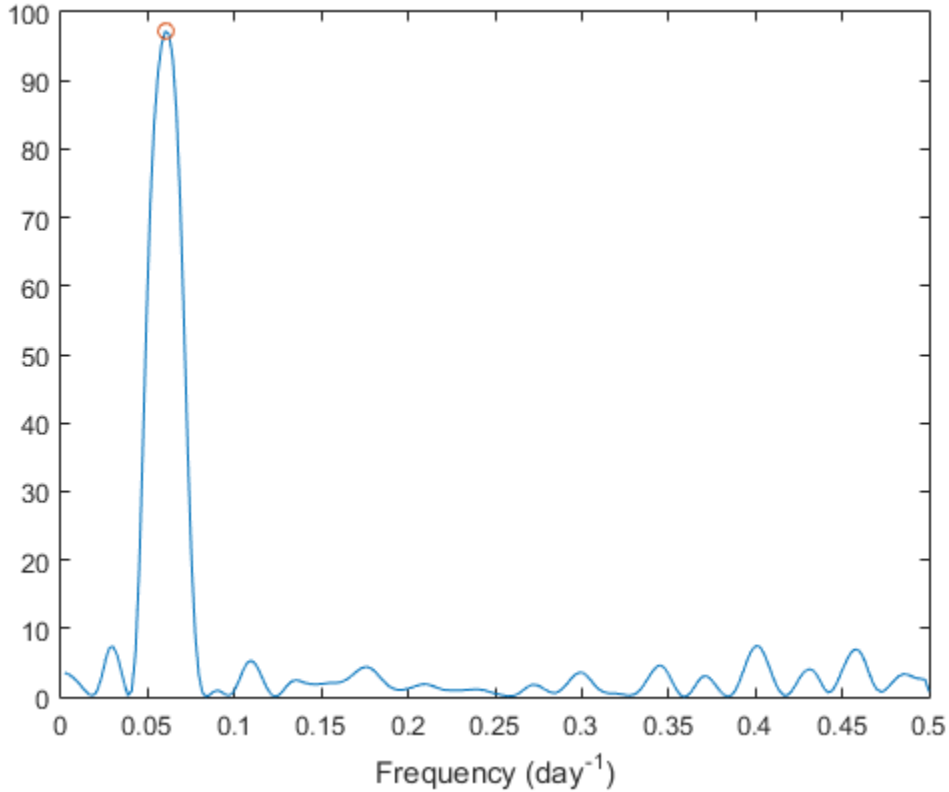

Determine Callisto's orbital period (in days) as the inverse of the frequency of maximum energy. The result differs by less than 1% from the value published by NASA.

```
Period = 1/f0

NASA = 16.6890184;
PercentDiscrep = (Period-NASA)/NASA*100


Period =

    16.6454
```

```
PercentDiscrep =

    -0.2613
```

Welch Spectrum Estimates

Create a signal consisting of three noisy sinusoids and a chirp, sampled at 200 kHz for 0.1 second. The frequencies of the sinusoids are 1 kHz, 10 kHz, and 20 kHz. The sinusoids have different amplitudes and noise levels. The noiseless chirp has a frequency that starts at 20 kHz and increases linearly to 30 kHz during the sampling.

```
Fs = 200e3;
Fc = [1 10 20]'*1e3;
Ns = 0.1*Fs;

t = (0:Ns-1)/Fs;
x = [1 1/10 10]*sin(2*pi*Fc*t)+[1/200 1/2000 1/20]*randn(3,Ns);
x = x+chirp(t,20e3,t(end),30e3);
```

Compute the Welch PSD estimate and the maximum-hold and minimum-hold spectra of the signal. Plot the results.

```
[pxx,f] = pwelch(x,[],[],[],Fs);
pmax = pwelch(x,[],[],[],Fs,'maxhold');
pmin = pwelch(x,[],[],[],Fs,'minhold');

plot(f,pow2db(pxx))
hold on
plot(f,pow2db([pmax pmin]),':')
hold off
xlabel('Frequency (Hz)')
ylabel('PSD (dB/Hz)')
legend('pwelch','maxhold','minhold')
```
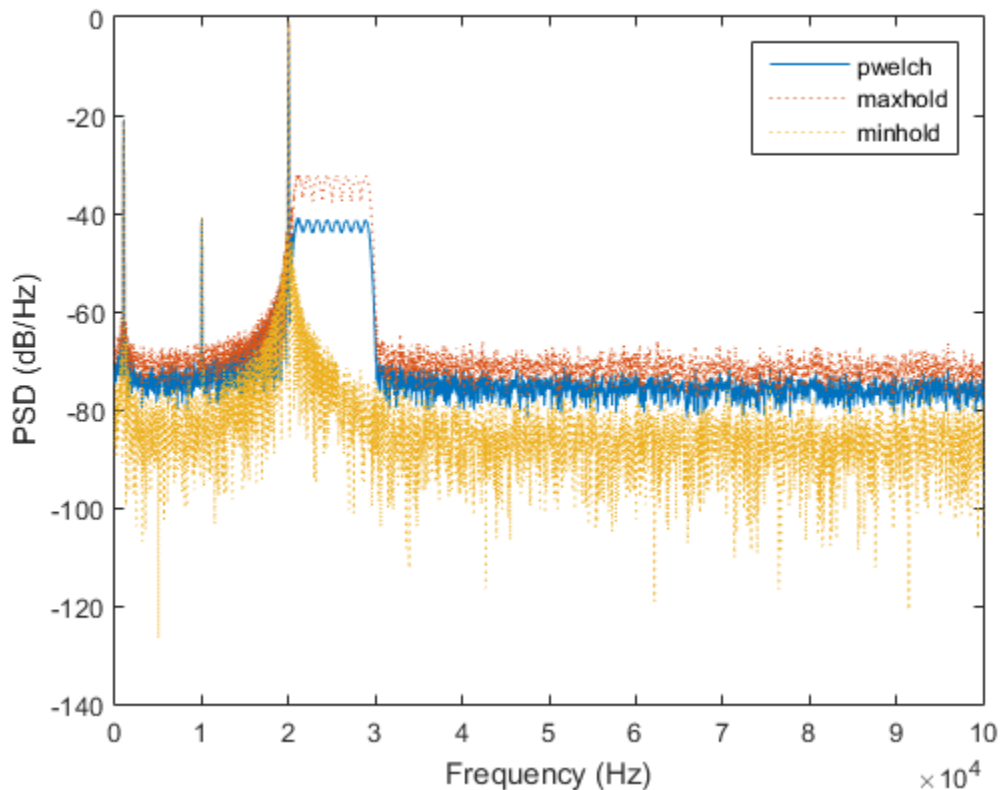
Repeat the procedure, this time computing centered power spectrum estimates.

```
[pxx,f] = pwelch(x,[],[],[],Fs,'centered','power');
pmax = pwelch(x,[],[],[],Fs,'maxhold','centered','power');
pmin = pwelch(x,[],[],[],Fs,'minhold','centered','power');

plot(f,pow2db(pxx))
hold on
plot(f,pow2db([pmax pmin]),':')
hold off
xlabel('Frequency (Hz)')
ylabel('Power (dB)')
legend('pwelch','maxhold','minhold')
```
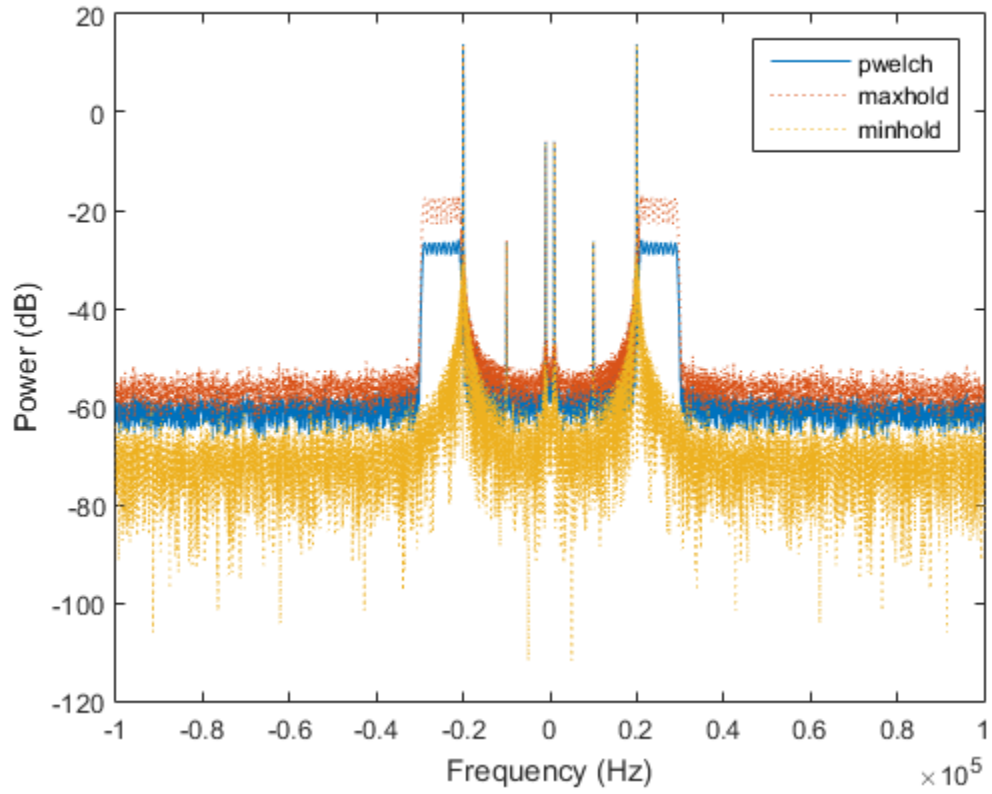
Linear Prediction

Prediction Polynomial

This example shows how to obtain the prediction polynomial from an autocorrelation sequence. The example also shows that the resulting prediction polynomial has an inverse that produces a stable all-pole filter. You can use the all-pole filter to filter a wide-sense stationary white noise sequence to produce a wide-sense stationary autoregressive process.

Create an autocorrelation sequence defined by

$$r(k) = \frac{24}{5} 2^{-|k|} - \frac{27}{10} 3^{-|k|}, \quad k = 0, 1, 2.$$

```
k = 0:2;
rk = (24/5)*2.^(-k)-(27/10)*3.^(-k);
```

Use ac2poly to obtain the prediction polynomial of order 2, which is

$$A(z) = 1 - \frac{5}{6}z^{-1} + \frac{1}{6}z^{-2}.$$

```
A = ac2poly(rk);
```

Examine the pole-zero plot of the FIR filter to see that the zeros are inside the unit circle.

```
zplane(A,1)
```
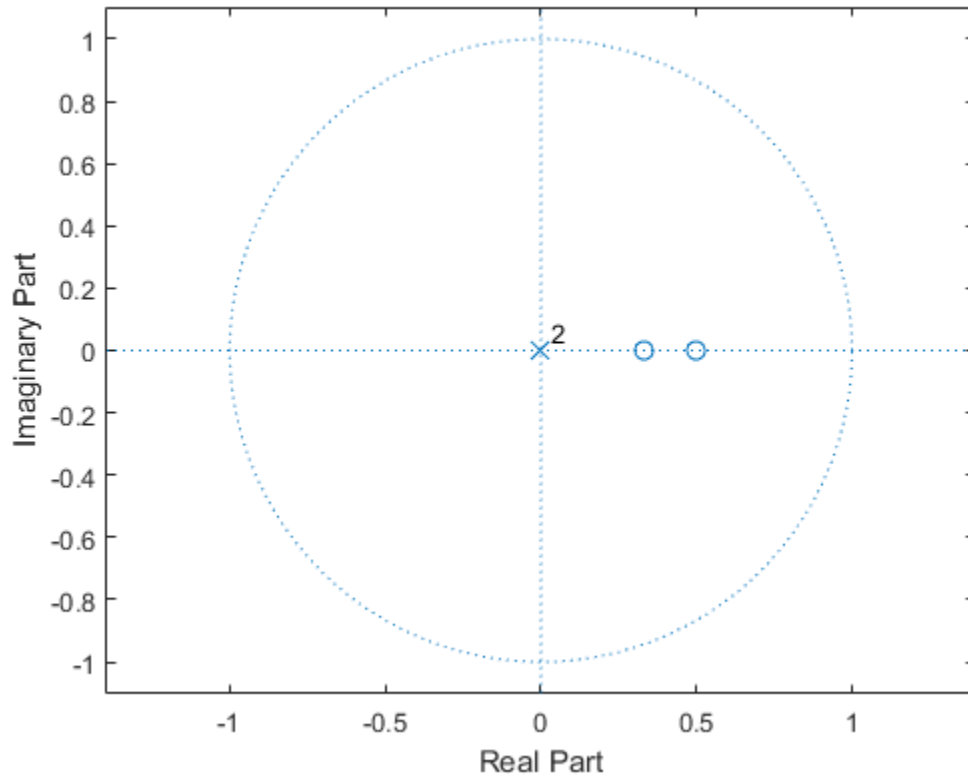
The inverse all-pole filter is stable with poles inside the unit circle.

```
zplane(1,A)
```
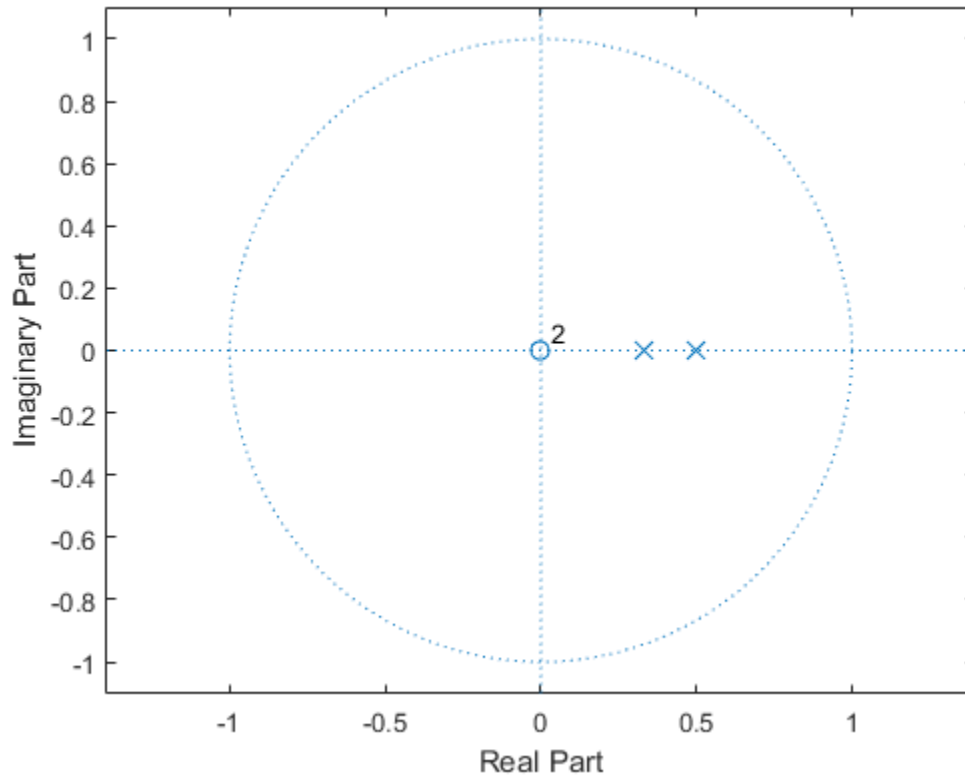
Use the all-pole filter to produce a realization of a wide-sense stationary AR(2) process from a white-noise sequence. Set the random number generator to the default settings for reproducible results.

```
rng default
```

```
x = randn(1000,1);
y = filter(1,A,x);
```

Compute the sample autocorrelation of the AR(2) realization and show that the sample autocorrelation is close to the true autocorrelation.

```
[xc,lags] = xcorr(y,2,'biased');
[xc(3:end) rk']
```

```
ans =

    2.2401    2.1000
    1.6419    1.5000
    0.9980    0.9000
```

Formant Estimation with LPC Coefficients

This example shows how to estimate vowel formant frequencies using linear predictive coding (LPC). The formant frequencies are obtained by finding the roots of the prediction polynomial.

This example uses the speech sample mtlb.mat, which is part of Signal Processing Toolbox™. The speech is lowpass-filtered. Because of the low sampling frequency, this speech sample is not optimal for this example. The low sampling frequency limits the order of the autoregressive model you can fit to the data. In spite of this limitation, the example illustrates the technique for using LPC coefficients to determine vowel formants.

Load the speech signal. The recording is a woman saying "MATLAB". The sampling frequency is 7418 Hz.

```
load mtlb
```

The MAT file contains the speech waveform, mtlb, and the sampling frequency, Fs.

Use the spectrogram function to identify a voiced segment for analysis.

```
segmentlen = 100;
noverlap = 90;
NFFT = 128;

spectrogram(mtlb,segmentlen,noverlap,NFFT,Fs,'yaxis')
```
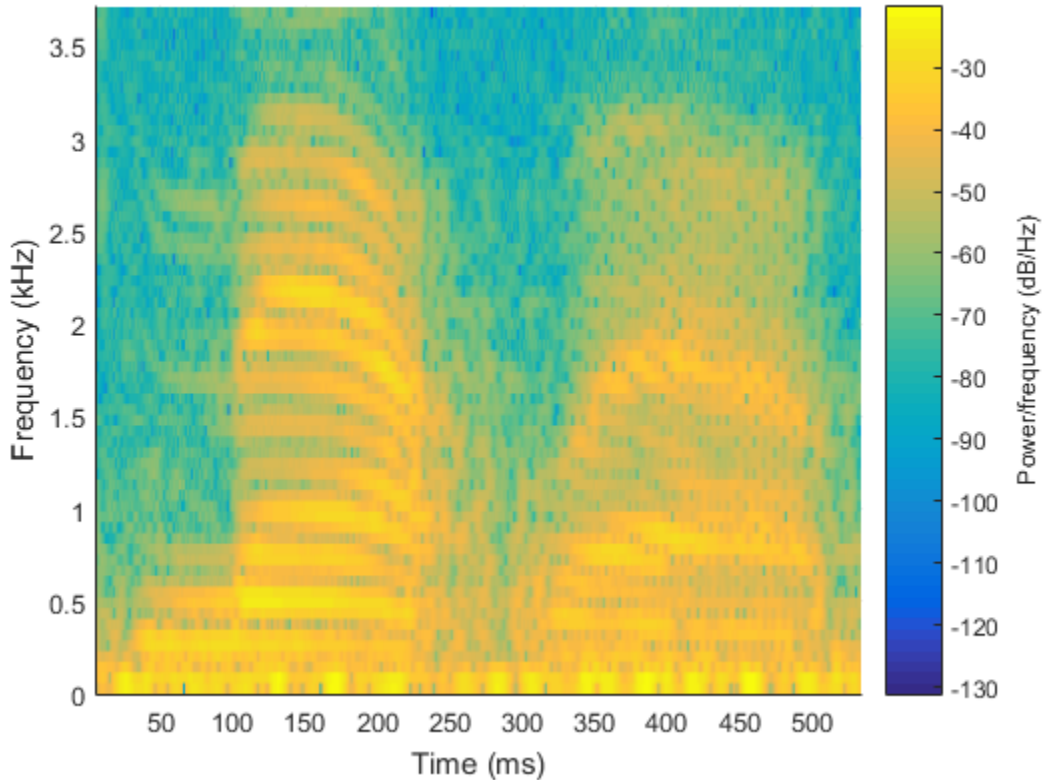
Extract the segment from 0.1 to 0.25 seconds for analysis. The extracted segment corresponds roughly to the first vowel, /ae/, in "MATLAB".

```
dt = 1/Fs;
I0 = round(0.1/dt);
Iend = round(0.25/dt);
x = mtlb(I0:Iend);
```

Two common preprocessing steps applied to speech waveforms before linear predictive coding are windowing and pre-emphasis (highpass) filtering.

Window the speech segment using a Hamming window.

```
x1 = x.*hamming(length(x));
```

Apply a pre-emphasis filter. The pre-emphasis filter is a highpass all-pole (AR(1)) filter.

```
preemph = [1 0.63];
x1 = filter(1,preemph,x1);
```

Obtain the linear prediction coefficients. To specify the model order, use the general rule that the order is two times the expected number of formants plus 2. In the frequency range, $[0, |Fs|/2]$, you expect three formants. Therefore, set the model order equal to 8. Find the roots of the prediction polynomial returned by `lpc`.

```
A = lpc(x1,8);
rts = roots(A);
```

Because the LPC coefficients are real-valued, the roots occur in complex conjugate pairs. Retain only the roots with one sign for the imaginary part and determine the angles corresponding to the roots.

```
rts = rts(imag(rts)>=0);
angz = atan2(imag(rts),real(rts));
```

Convert the angular frequencies in rad/sample represented by the angles to Hz and calculate the bandwidths of the formants.

The bandwidths of the formants are represented by the distance of the prediction polynomial zeros from the unit circle.

```
[frqs,indices] = sort(angz.*(Fs/(2*pi)));
bw = -1/2*(Fs/(2*pi))*log(abs(rts(indices)));
```

Use the criterion that formant frequencies should be greater than 90 Hz with bandwidths less than 400 Hz to determine the formants.

```
nn = 1;
for kk = 1:length(frqs)
    if (frqs(kk) > 90 && bw(kk) <400)
        formants(nn) = frqs(kk);
        nn = nn+1;
    end
end
formants
```

```
formants =
```

```
1.0e+03 *

 0.8697    2.0265    2.7380
```

The first three formants are 869.70, 2026.49, and 2737.95 Hz.

References

[1] Snell, Roy C., and Fausto Milinazzo. "Formant location from LPC analysis data." IEEE® Transactions on Speech and Audio Processing. Vol. 1, Number 2, 1993, pp. 129-134.

[2] Loizou, Philipos C. "COLEA: A MATLAB Software Tool for Speech Analysis."

AR Order Selection with Partial Autocorrelation Sequence

This example shows how to assess the order of an autoregressive model using the partial autocorrelation sequence. For these processes, you can use the partial autocorrelation sequence to help with model order selection. For a stationary time series with values $X(1), X(2), X(3), \ldots, X(k+1)$, the partial autocorrelation sequence at lag k is the correlation between $X(1)$ and $X(k+1)$ after regressing $X(1)$ and $X(k+1)$ on the intervening observations, $X(2), X(3), X(4), \ldots, X(k)$. For a moving average process, you can use the autocorrelation sequence to assess the order. However, for an autoregressive (AR) or autoregressive moving average (ARMA) process, the autocorrelation sequence does not help in order selection. Consider the AR(2) process defined by

$$X(n) + 1.5X(n-1) + 0.75X(n-2) = \varepsilon(n),$$

where $\varepsilon(n)$ is an $N(0, 1)$ Gaussian white noise process. The following example:

- Simulates a realization of the AR(2) process
- Graphically explores the correlation between lagged values of the time series
- Examines the sample autocorrelation sequence of the time series
- Fits an AR(15) model to the time series by solving the Yule-Walker equations (`aryule`)
- Uses the reflection coefficients returned by `aryule` to compute the partial autocorrelation sequence
- Examines the partial autocorrelation sequence to select the model order

Simulate a 1000-sample time series from the AR(2) process defined by the difference equation. Set the random number generator to the default settings for reproducible results.

```
A = [1 1.5 0.75];
rng default
x = filter(1,A,randn(1000,1));
```

View the frequency response of the AR(2) process.

```
freqz(1,A)
```
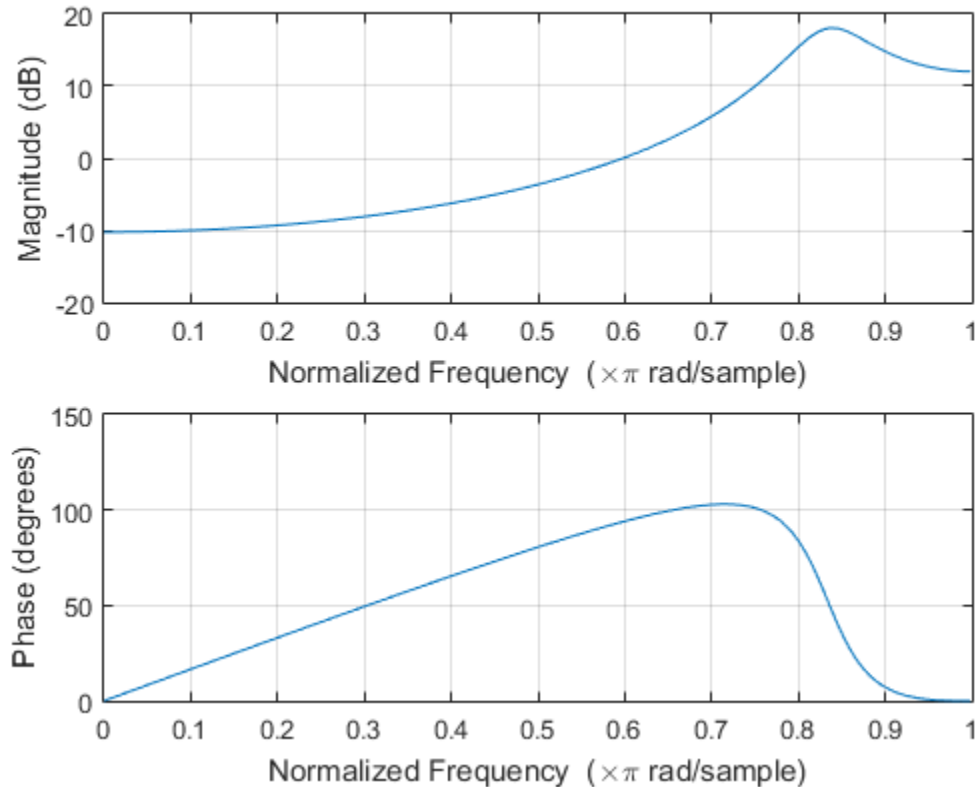
The AR(2) process acts like a highpass filter in this case.

Graphically examine the correlation in x by producing scatter plots of $X(n)$ vs. $X(1)$ for $n = 2, 3, 4, 5$.

```
x12 = x(1:end-1);
x21 = x(2:end);
subplot(2,2,1)
plot(x12,x21,'*')
xlabel('X_1')
ylabel('X_2')

x13 = x(1:end-2);
x31 = x(3:end);
```

```
subplot(2,2,2)
plot(x13,x31,'*')
xlabel('X_1')
ylabel('X_3')

x14 = x(1:end-3);
x41 = x(4:end);
subplot(2,2,3)
plot(x14,x41,'*')
xlabel('X_1')
ylabel('X_4')

x15 = x(1:end-4);
x51 = x(5:end);
subplot(2,2,4)
plot(x15,x51,'*')
xlabel('X_1')
ylabel('X_5')
```
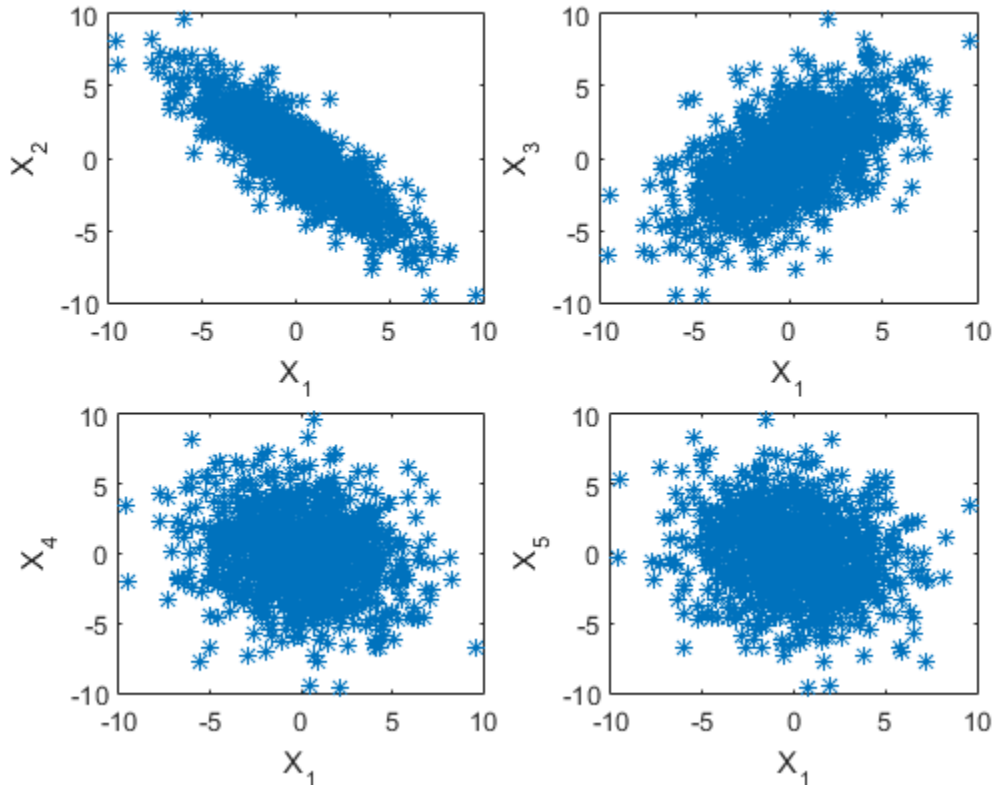
In the scatter plot, you see there is a linear relationship between $X(1)$ and $X(2)$ and between $X(1)$ and $X(3)$, but not between $X(1)$ and either $X(4)$ or $X(5)$.

The points in the top row scatter plots fall approximately on a line with a negative slope in the top left panel and positive slope in the top right panel. The scatter plots in the bottom two panels do not show any apparent linear relationship.

The negative correlation between $X(1)$ and $X(2)$ and the positive correlation between $X(1)$ and $X(3)$ are explained by the highpass-filter behavior of the AR(2) process.

Find the sample autocorrelation sequence out to lag 50 and plot the result.

```
[xc,lags] = xcorr(x,50,'coeff');
```

```
figure
stem(lags(51:end),xc(51:end),'filled')
xlabel('Lag')
ylabel('ACF')
title('Sample Autocorrelation Sequence')
```
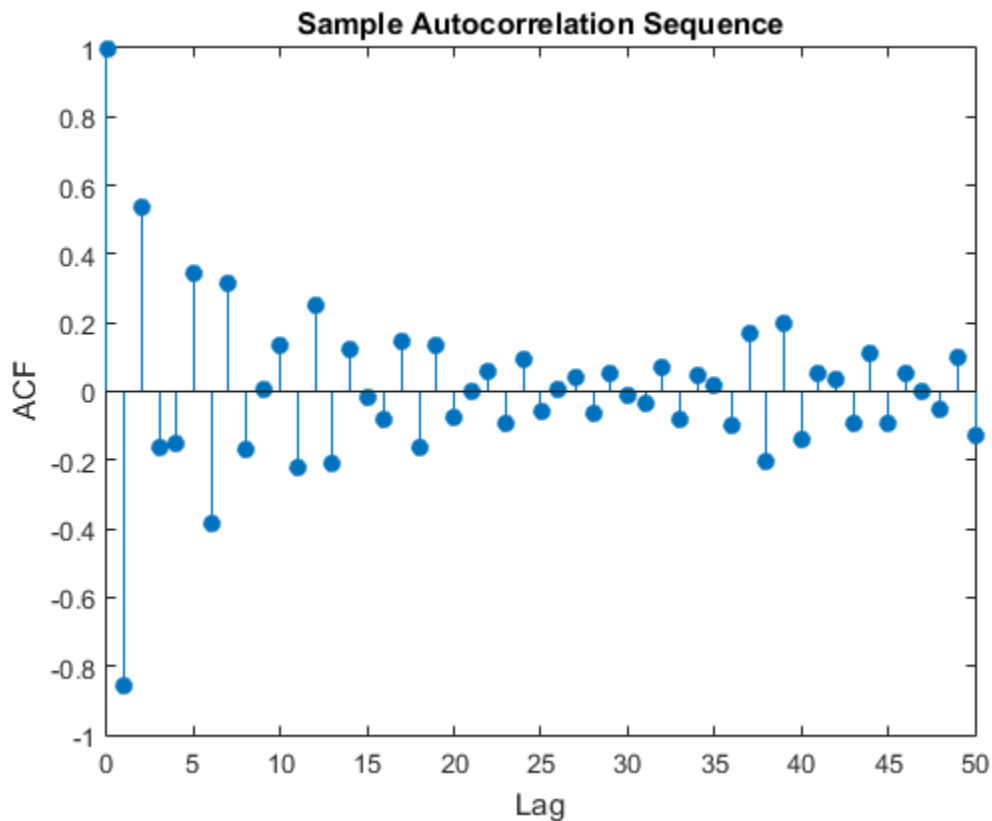

The sample autocorrelation sequence shows a negative value at lag 1 and a positive value at lag 2. Based on the scatter plot, this is the expected result. However, you cannot determine from the sample autocorrelation sequence what order is appropriate for the AR model.

Fit an AR(15) model using `aryule`. Return the reflection coefficients. The negative of the reflection coefficients is the partial autocorrelation sequence.

```
[arcoefs,E,K] = aryule(x,15);
pacf = -K;
```

Plot the partial autocorrelation sequence along with the large-sample 95% confidence intervals. If the data are generated by an autoregressive process of order p, the values of the sample partial autocorrelation sequence for lags greater than p follow a $N(0, 1/N)$ distribution, where N is the length of the time series.

```
stem(pacf,'filled')
xlabel('Lag')
ylabel('Partial Autocorrelation')
xlim([1 15])
uconf = 1.96/sqrt(1000);
lconf = -uconf;
hold on
plot([1 15],[1 1]'*[lconf uconf],'r')
```
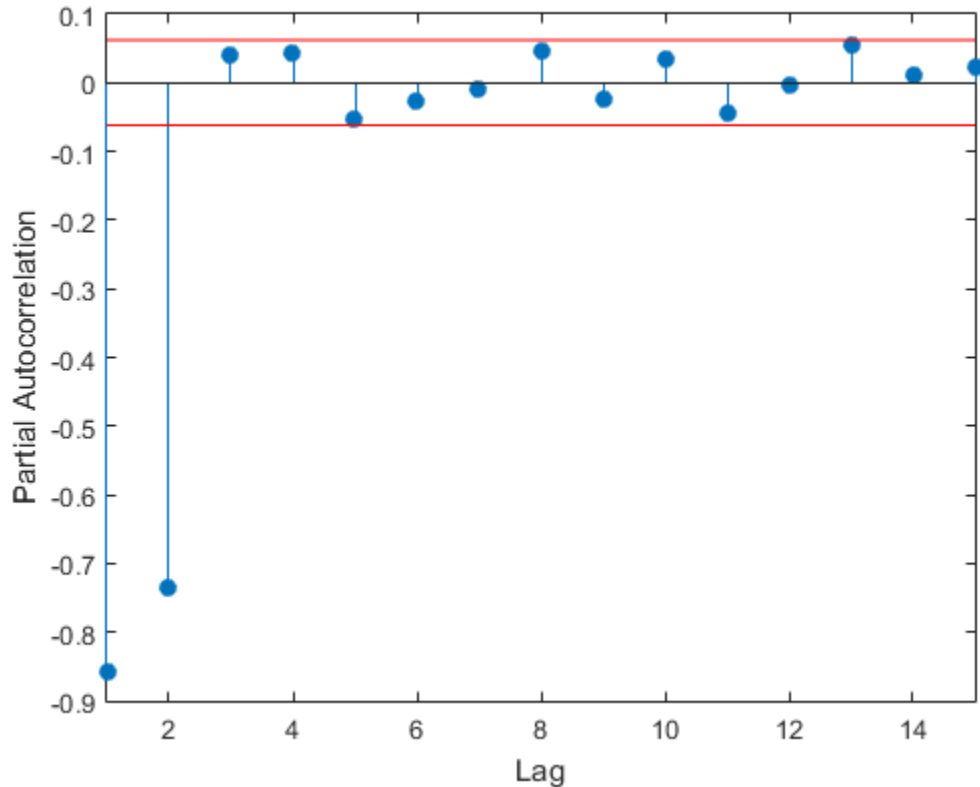
The only values of the partial autocorrelation sequence outside the 95% confidence bounds occur at lags 1 and 2. This indicates that the correct model order for the AR process is 2.

In this example, you generated the time series to simulate an AR(2) process. The partial autocorrelation sequence only confirms that result. In practice, you have only the observed time series without any prior information about model order. In a realistic scenario, the partial autocorrelation is an important tool for appropriate model order selection in stationary autoregressive time series.

14

Transforms

Complex Cepstrum -- Fundamental Frequency Estimation

This example shows how to estimate a speaker's fundamental frequency using the complex cepstrum. The example also estimates the fundamental frequency using a zero-crossing method and compares the results.

Load the speech signal. The recording is a woman saying "MATLAB". The sampling frequency is 7418 Hz. The following code loads the speech waveform, mtlb, and the sampling frequency, Fs, into the MATLAB® workspace.

```
load mtlb
```

Use the spectrogram to identify a voiced segment for analysis.

```
segmentlen = 100;
noverlap = 90;
NFFT = 128;

spectrogram(mtlb,segmentlen,noverlap,NFFT,Fs,'yaxis')
```
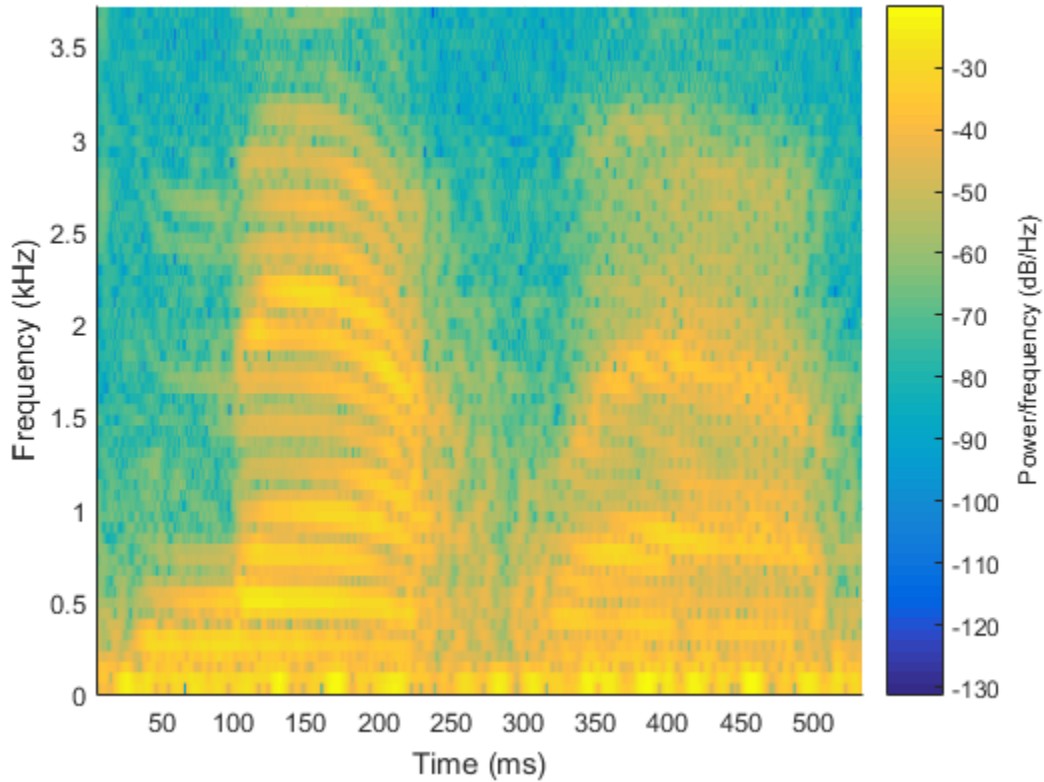
Extract the segment from 0.1 to 0.25 seconds for analysis. The extracted segment corresponds roughly to the first vowel, /ae/, in "MATLAB".

```
dt = 1/Fs;
I0 = round(0.1/dt);
Iend = round(0.25/dt);
x = mtlb(I0:Iend);
```

Obtain the complex cepstrum.

```
c = cceps(x);
```

Plot the cepstrum for times ranging from 2 to 10 ms corresponding to a frequency range of approximately 100 to 500 Hz. Identify the peak in the cepstrum and find the frequency corresponding to the peak. Use the peak as the estimate of the fundamental frequency.

```
t = 0:dt:length(x)*dt-dt;

plot(t(15:75).*1e3,c(15:75))
xlabel('ms')

[~,I] = max(c(15:55));
fprintf('Complex cepstrum F0 estimate is %3.2f Hz.\n', 1/(t(I+15)))
```

```
Complex cepstrum F0 estimate is 231.81 Hz.
```
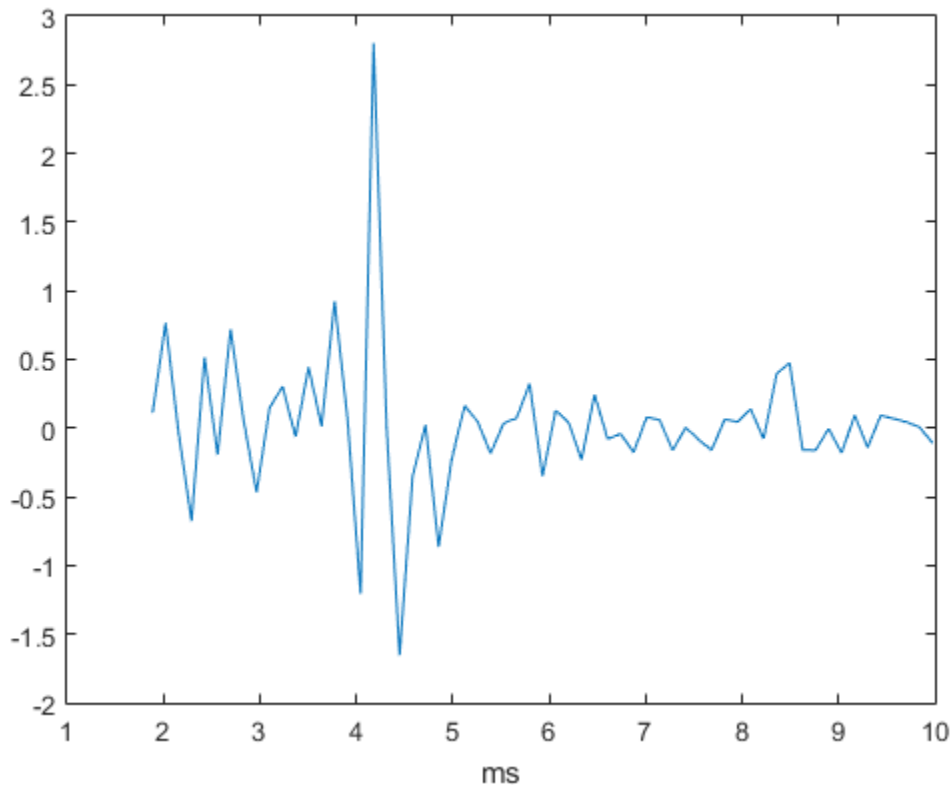
Use a zero-crossing detector on a lowpass-filtered and rectified form of the vowel to estimate the fundamental frequency.

```
[b0,a0]=butter(2,325/(Fs/2));
xin = abs(x);
xin = filter(b0,a0,xin);
xin = xin-mean(xin);
x2 = zeros(length(xin),1);
x2(1:length(x)-1) = xin(2:length(x));
zc=length(find((xin>0 & x2<0) | (xin<0 & x2>0)));
F0=0.5*Fs*zc/length(x);
fprintf('Zero-crossing F0 estimate is %3.2f Hz.\n',F0)

Zero-crossing F0 estimate is 233.27 Hz.
```

The estimate of the fundamental frequency obtained with the complex cepstrum is 231.81 Hz and the estimate with the zero-crossing detector is 233.27 Hz.

Analytic Signal for Cosine

This example shows how to determine the analytic signal. The example also demonstrates that the imaginary part of the analytic signal corresponding to a cosine is a sine with the same frequency. If the cosine has a nonzero mean (DC shift), then the real part of the analytic signal is the original cosine with the same mean, but the imaginary part has zero mean.

Create a cosine with a frequency of 100 Hz. The sampling frequency is 10 kHz. Add a DC offset of 2.5 to the cosine.

```
t = 0:1e-4:1;
x = 2.5+cos(2*pi*100*t);
```

Use the `hilbert` function to obtain the analytic signal. The real part is equal to the original signal. The imaginary part is the Hilbert transform of the original signal. Plot the real and imaginary parts for comparison.

```
y = hilbert(x);

clf
plot(t,real(y))
hold on
plot(t,imag(y))
xlim([0 0.1])
grid on
text([0.015 0.015],[3.7 1.2], ...
    {'Real Part \downarrow';'Imaginary Part \downarrow'})
```
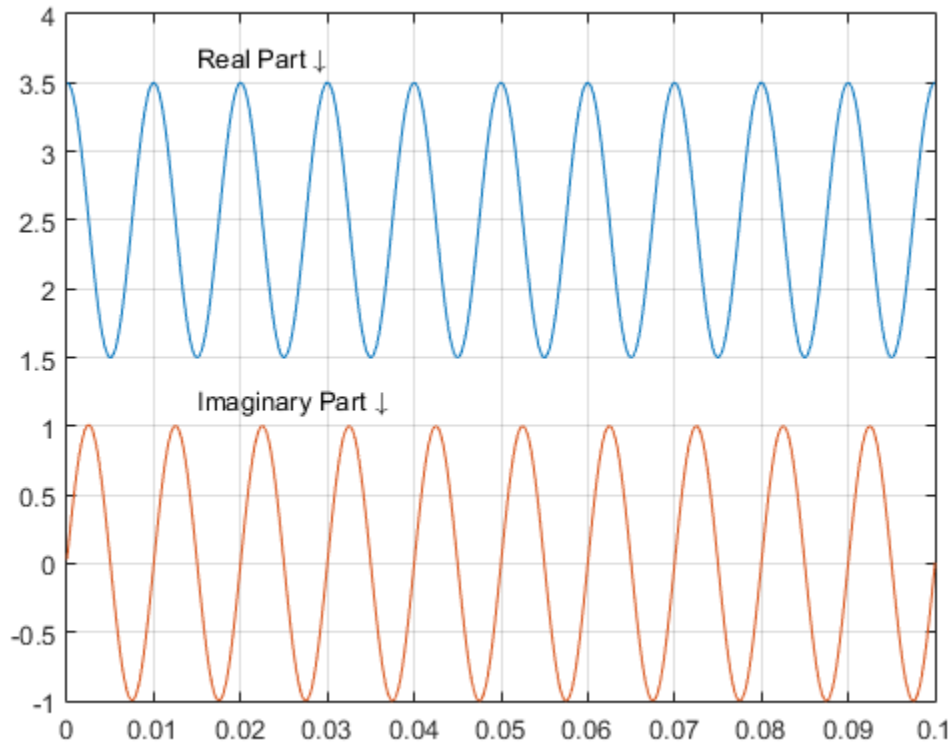
You see that the imaginary part is a sine with the same frequency as the cosine real part. However, the imaginary part has a mean of zero, while the real part has a mean of 2.5.

The original signal is

$$x(t) = 2.5 + \cos(2\pi 1000t).$$

The resulting analytic signal is

$$z(t) = 2.5 + e^{j2\pi 1000t}.$$

Plot 10 periods of the complex-valued analytic signal.

```matlab
prds = 1:1000;

figure
plot3(t(prds),real(y(prds)),imag(y(prds)))

xlabel('Time')
ylabel('Re \{z(t)\}')
zlabel('Im \{z(t)\}')
axis square
```
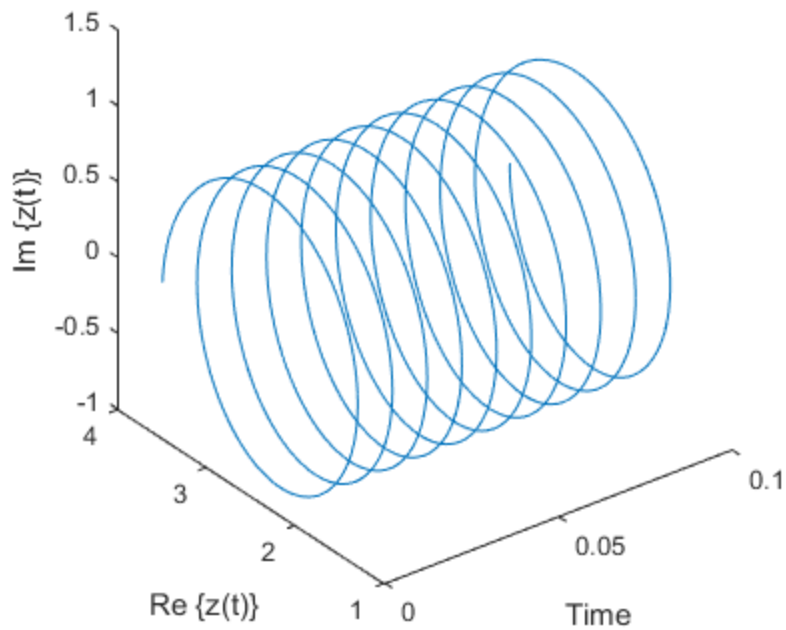
Envelope Extraction Using the Analytic Signal

This example shows how to extract the signal envelope using the analytic signal.

Create a double sideband amplitude-modulated signal. The carrier frequency is 1 kHz. The modulation frequency is 50 Hz. The modulation depth is 100%. The sampling frequency is 10 kHz.

```
t = 0:1e-4:1;
x = [1+cos(2*pi*50*t)].*cos(2*pi*1000*t);

plot(t,x)
xlim([0 0.1])
xlabel('Seconds')
```
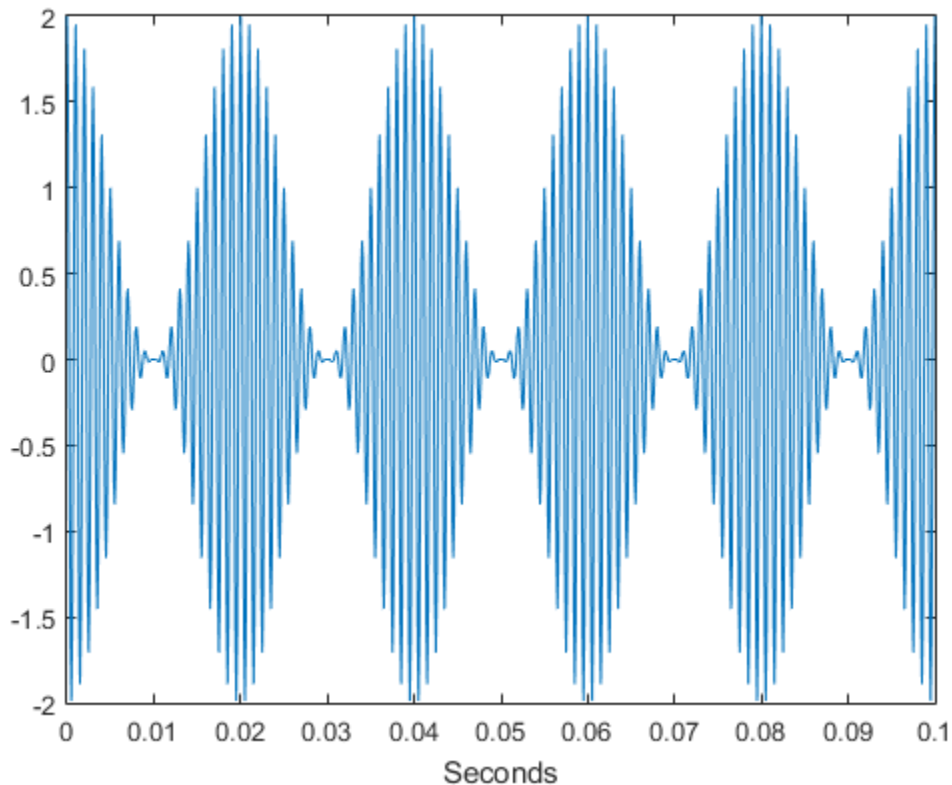
Obtain the analytic signal. Extract the envelope, which is the magnitude (modulus) of the analytic signal. Plot the envelope along with the original signal.

```
y = hilbert(x);
env = abs(y);

plot(t,x)
hold on
plot(t,[-1;1]*abs(y),'r','LineWidth',2)
xlim([0 0.1])
xlabel('Seconds')
```
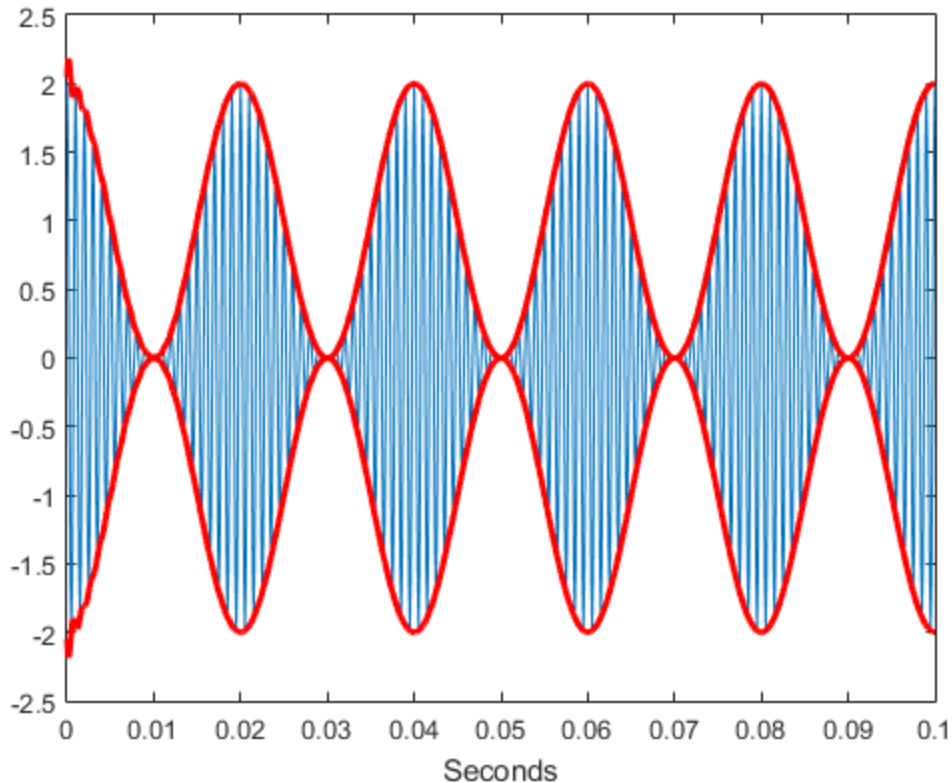

The magnitude of the analytic signal captures the slowly varying features of the signal, while the phase contains the high-frequency information.

Analytic Signal and Hilbert Transform

The `hilbert` function finds the exact analytic signal for a finite block of data. You can also generate the analytic signal by using an FIR Hilbert transformer filter to compute an approximation to the imaginary part.

Generate a sequence composed of three sinusoids with frequencies 203, 721, and 1001 Hz. The sequence is sampled at 10 kHz for about 1 second. Use the `hilbert` function to compute the analytic signal. Plot it between 0.01 seconds and 0.03 seconds.

```
fs = 1e4;
t = 0:1/fs:1;

x = 2.5+cos(2*pi*203*t)+sin(2*pi*721*t)+cos(2*pi*1001*t);

y = hilbert(x);

plot(t,real(y),t,imag(y))
xlim([0.01 0.03])
legend('real','imaginary')
title('hilbert Function')
```

hilbert Function
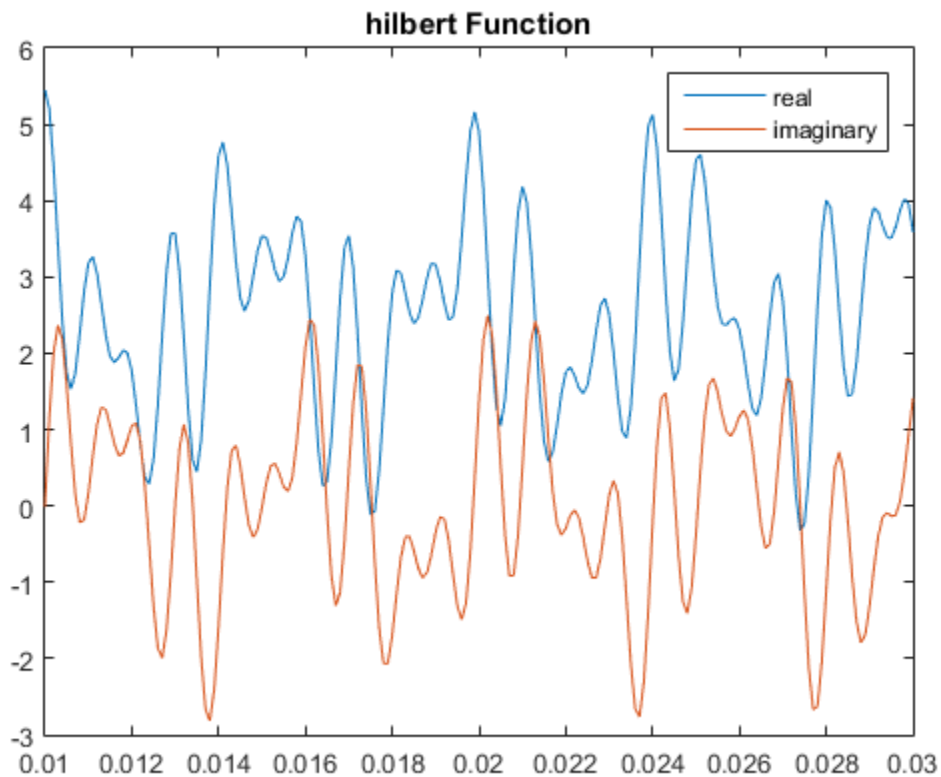
Compute Welch estimates of the power spectral densities of the original sequence and the analytic signal. Divide the sequences into Hamming windowed nonoverlapping sections of length 256. Verify that the analytic signal has no power at negative frequencies.

```
pwelch([x;y].',256,0,[],fs,'centered')
legend('Original','hilbert')
```
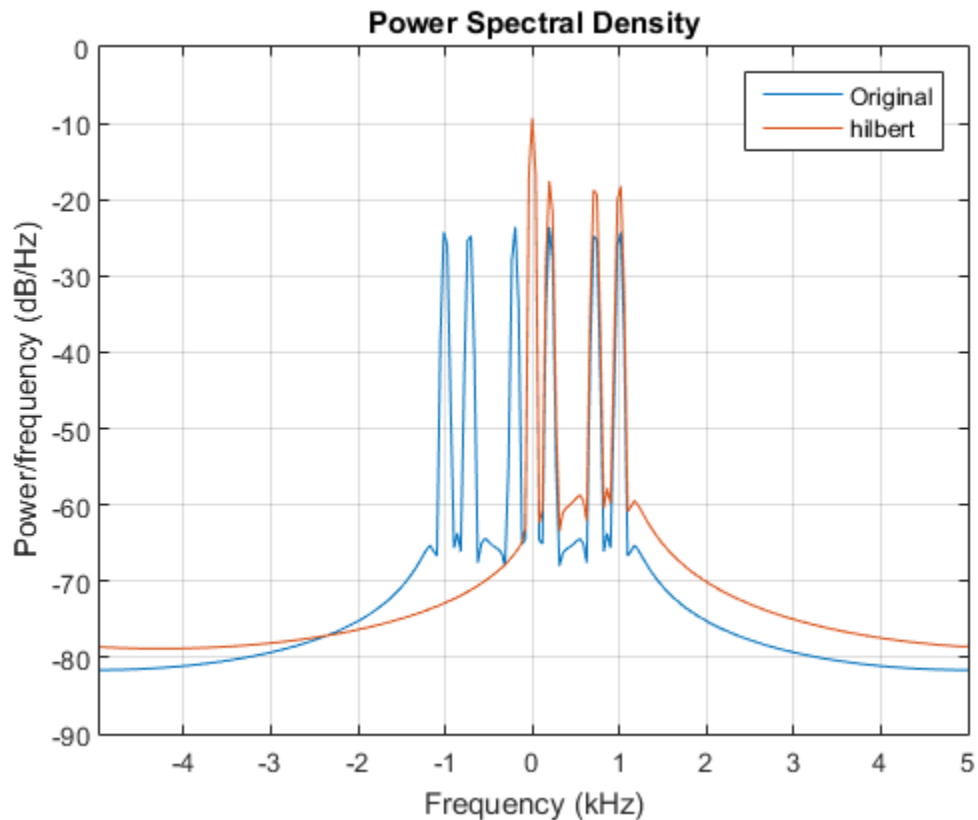
Use the `designfilt` function to design a 60th-order Hilbert transformer FIR filter. Specify a transition width of 400 Hz. Visualize the frequency response of the filter. Filter the sinusoidal sequence to approximate the imaginary part of the analytic signal.

```
fo = 60;

d = designfilt('hilbertfir','FilterOrder',fo, ...
        'TransitionWidth',400,'SampleRate',fs);

freqz(d,1024,fs)

hb = filter(d,x);
```
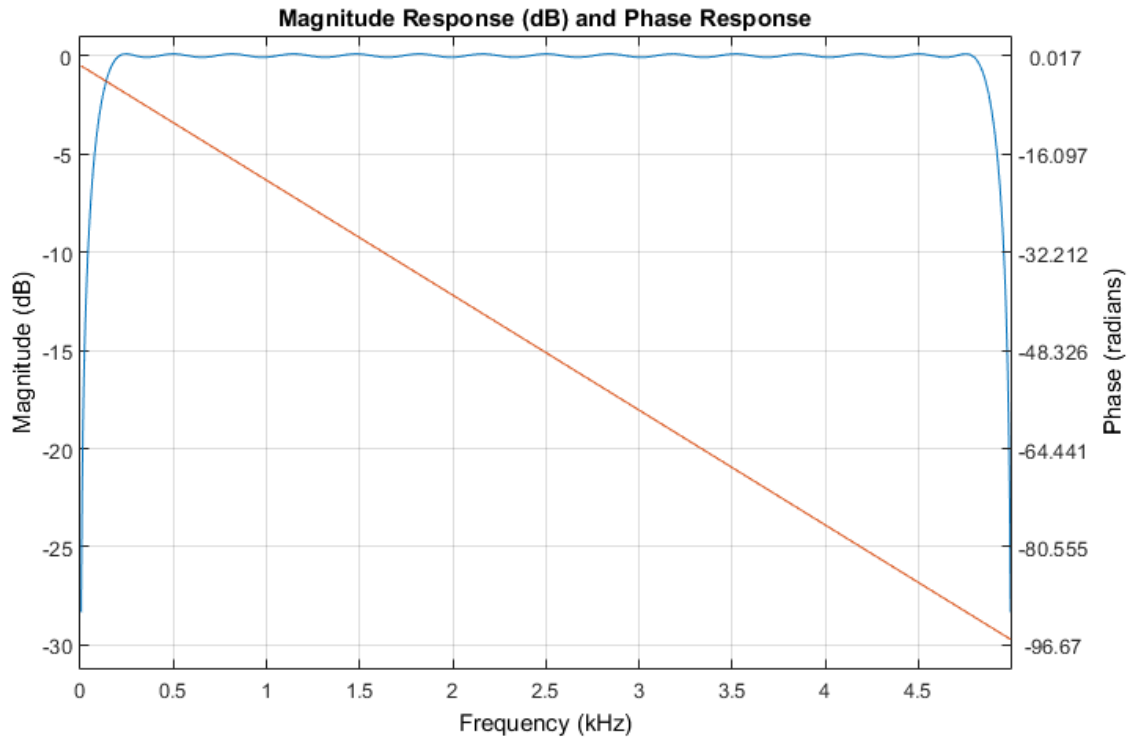
The group delay of the filter, `grd`, is equal to one-half the filter order. Compensate for this delay. Remove the first `grd` samples of the imaginary part and the last `grd` samples of the real part and the time vector. Plot the result between 0.01 seconds and 0.03 seconds.

```
grd = fo/2;

y2 = x(1:end-grd) + 1j*hb(grd+1:end);
t2 = t(1:end-grd);

plot(t2,real(y2),t2,imag(y2))
xlim([0.01 0.03])
legend('real','imaginary')
title('FIR Filter')
```
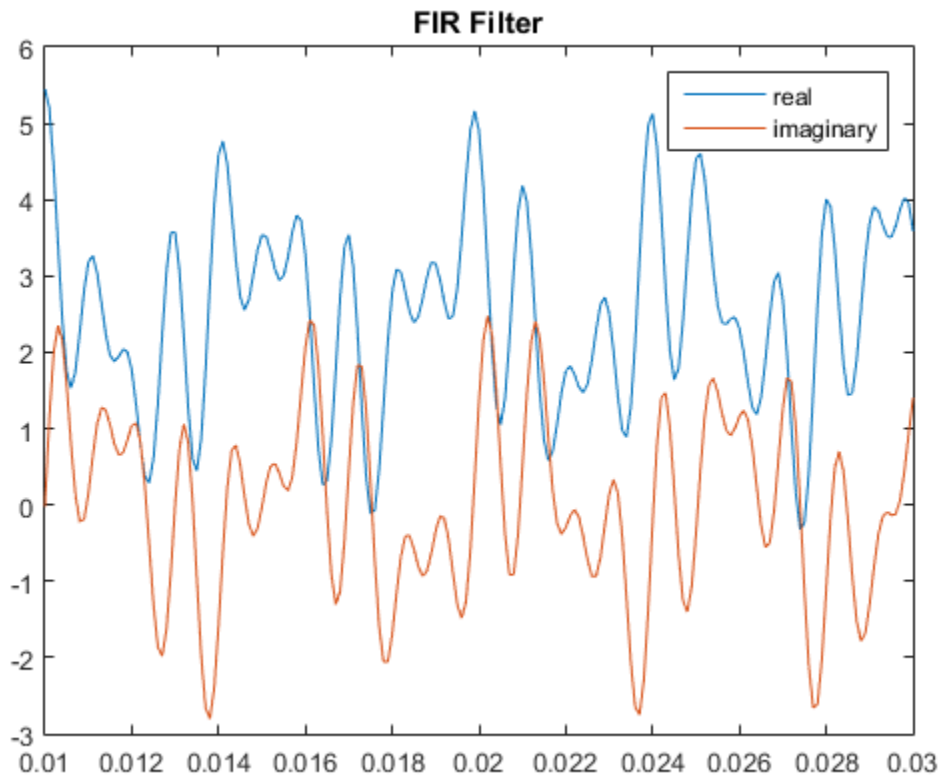
Estimate the PSD of the approximate analytic signal and compare it to the `hilbert` result.

```
pwelch([y;[y2 zeros(1,grd)]].',256,0,[],fs,'centered')
legend('hilbert','FIR Filter')
```

Power Spectral Density
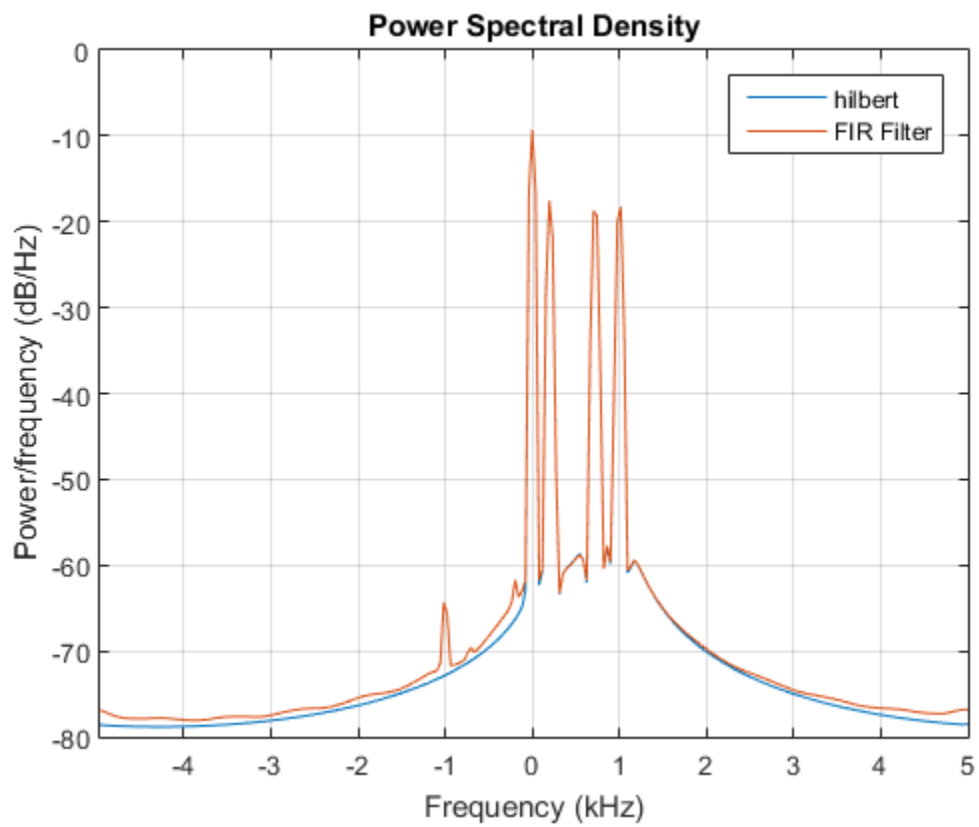
Hilbert Transform and Instantaneous Frequency

The Hilbert transform estimates the instantaneous frequency of a signal for monocomponent signals only. A monocomponent signal is described in the time-frequency plane by a single "ridge." The set of monocomponent signals includes single sinusoids, but also includes signals like chirps.

Generate a chirp sampled at 1 kHz for two seconds. Specify the chirp so its frequency is initially 100 Hz and increases to 200 Hz after one second.

```
Fs = 1000;
t = 0:1/Fs:2-1/Fs;
y = chirp(t,100,1,200);
```

Estimate the spectrum of the chirp using the short-time Fourier transform implemented in the `spectrogram` function. Divide the signal into sections of length 100, windowed with a Hamming window. Specify 80 samples of overlap between adjoining sections and evaluate the spectrum at $\lfloor 100/2 + 1 \rfloor = 51$ frequencies.

```
spectrogram(y,100,80,100,Fs,'yaxis')
view(-27,62)
shading interp
```

14-17
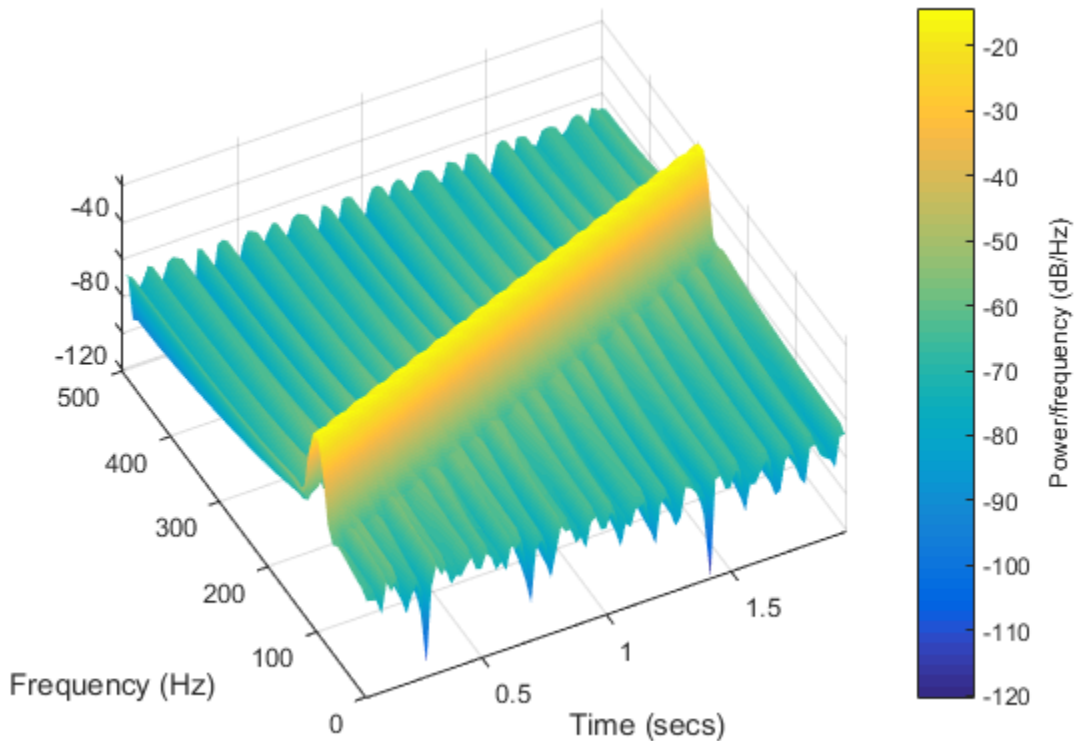
The signal is well described by a single peak frequency at each point in time.

Compute the analytic signal and differentiate its phase to measure the instantaneous frequency. The scaled derivative yields a meaningful estimate.

```
z = hilbert(y);
instfreq = Fs/(2*pi)*diff(unwrap(angle(z)));

plot(t(2:end),instfreq)
xlabel('Time')
ylabel('Hz')
grid on
title('Instantaneous Frequency')
```
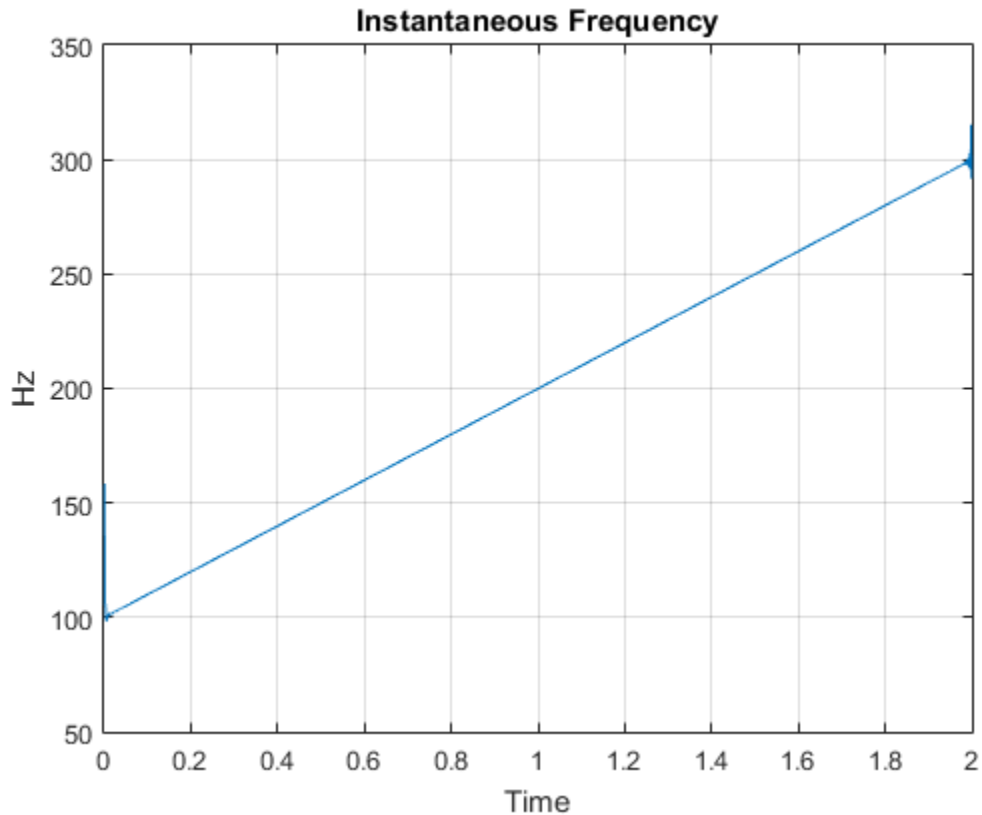
Repeat the procedure, this time for a signal that is not monocomponent.

Generate a sum of two sinusoids of frequencies 60 Hz and 90 Hz, sampled at 1023 Hz for two seconds. Compute and plot the spectrogram. Use Hamming-windowed sections that have lengths of 200 and sample overlaps of 180. Evaluate the spectrum at $\lfloor 256/2 + 1 \rfloor = 129$ frequencies.

```
Fs = 1023;
t = 0:1/Fs:2;
x = sin(2*pi*60*t)+sin(2*pi*90*t);

spectrogram(x,200,180,256,Fs,'yaxis')
ax = gca;
```

```
ax.YTick = [60 90];
shading interp
```
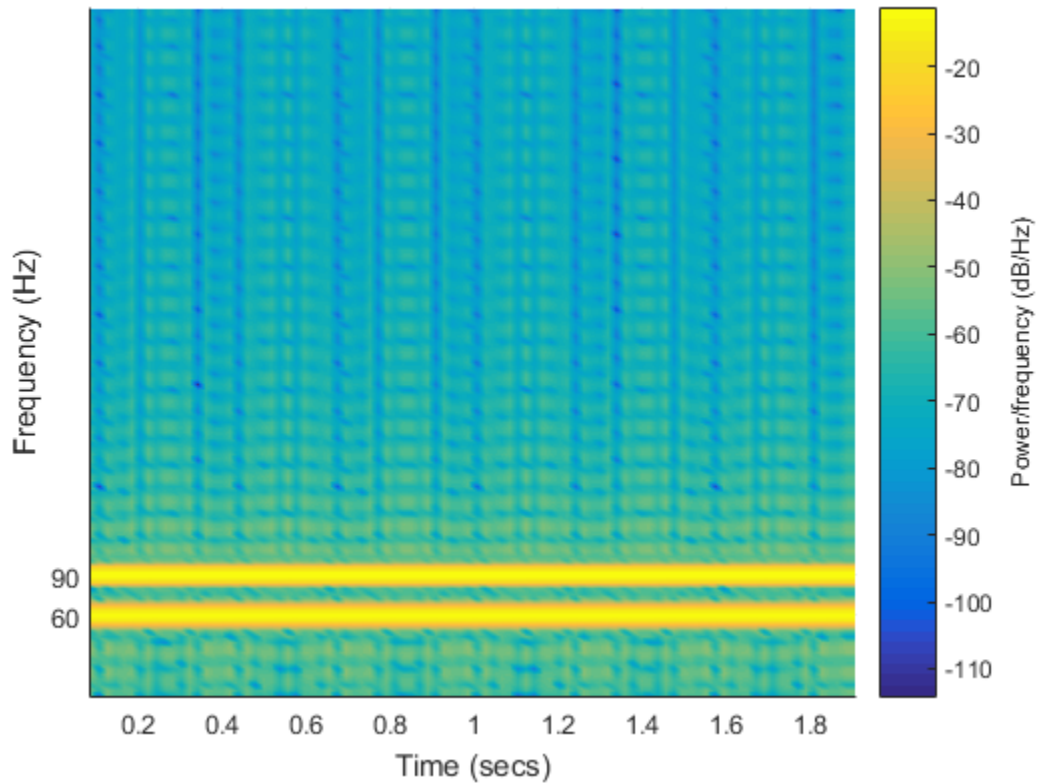

Each time point shows the presence of the two components.

Compute the analytic signal and differentiate its phase. Zoom in to the region enclosing the frequencies of the sinusoids.

```
z = hilbert(x);
instfreq = Fs/(2*pi)*diff(unwrap(angle(z)));
plot(t(2:end),instfreq)

ylim([60 90])
xlabel('Time')
```

```
ylabel('Hz')
```
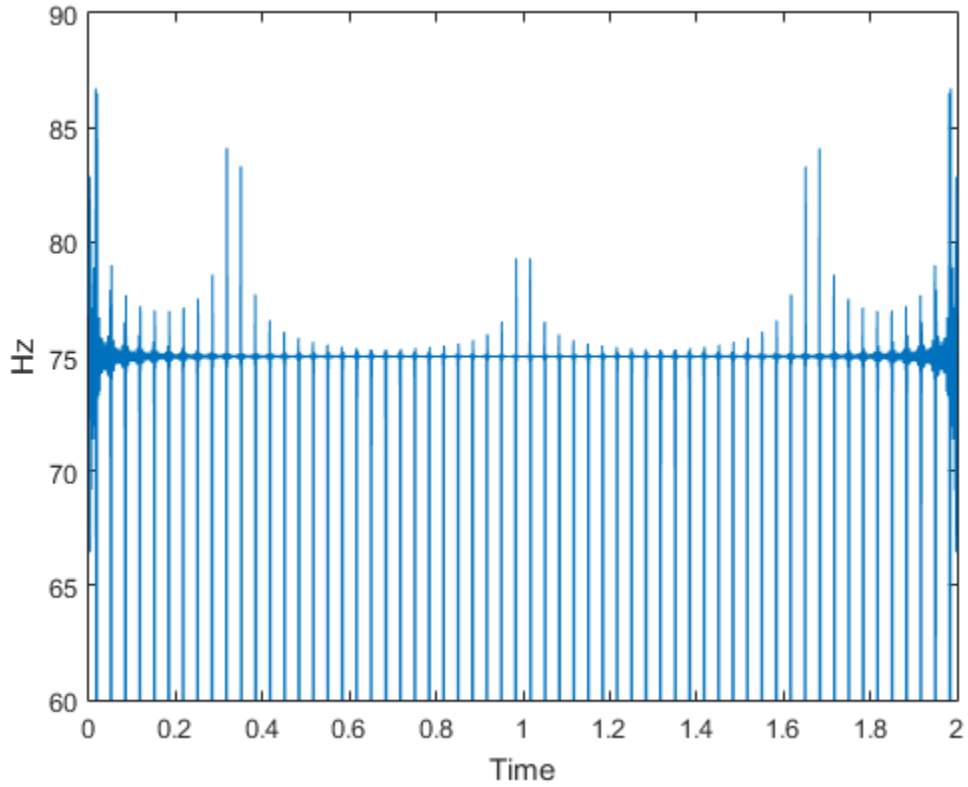

The analytic signal predicts an instantaneous frequency that is the average of the sinusoid frequencies.

Single-Sideband Amplitude Modulation

This example shows how to use the Hilbert transform to carry out single-sideband (SSB) amplitude modulation (AM) of a signal. Single-sideband AM signals have less bandwidth than normal AM signals.

Generate 512 samples of a simulated broadband signal using the `sinc` function. Specify a bandwidth of $\pi/4$ rad/sample.

```
N = 512;
n = 0:N-1;

bw = 1/4;
x = sinc((n-N/2)*bw);
```

Add white Gaussian noise such that the signal-to-noise ratio is 20 dB. Reset the random number generator for reproducible results. Use the `periodogram` function to estimate the power spectral density (PSD) of the signal.

```
rng default

SNR = 20;
noise = randn(size(x))*std(x)/db2mag(SNR);
x = x + noise;

periodogram(x)
```

Periodogram Power Spectral Density Estimate
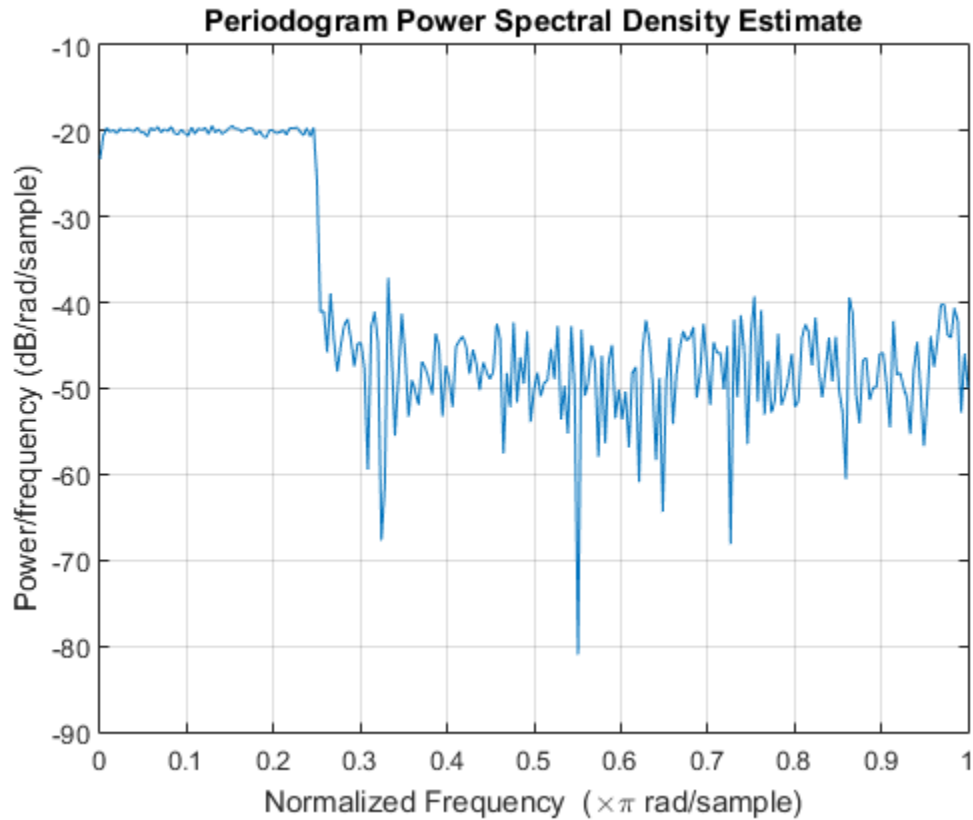
Amplitude modulate the signal using a cosine of carrier frequency $\omega_c = \pi/2$. Multiply by $\sqrt{2}$ so that the power of the modulated signal equals the power of the original signal. Estimate the PSD.

```
wc = pi/2;

x1 = x.*cos(wc*n)*sqrt(2);

periodogram(x1)
legend('Modulated')
```
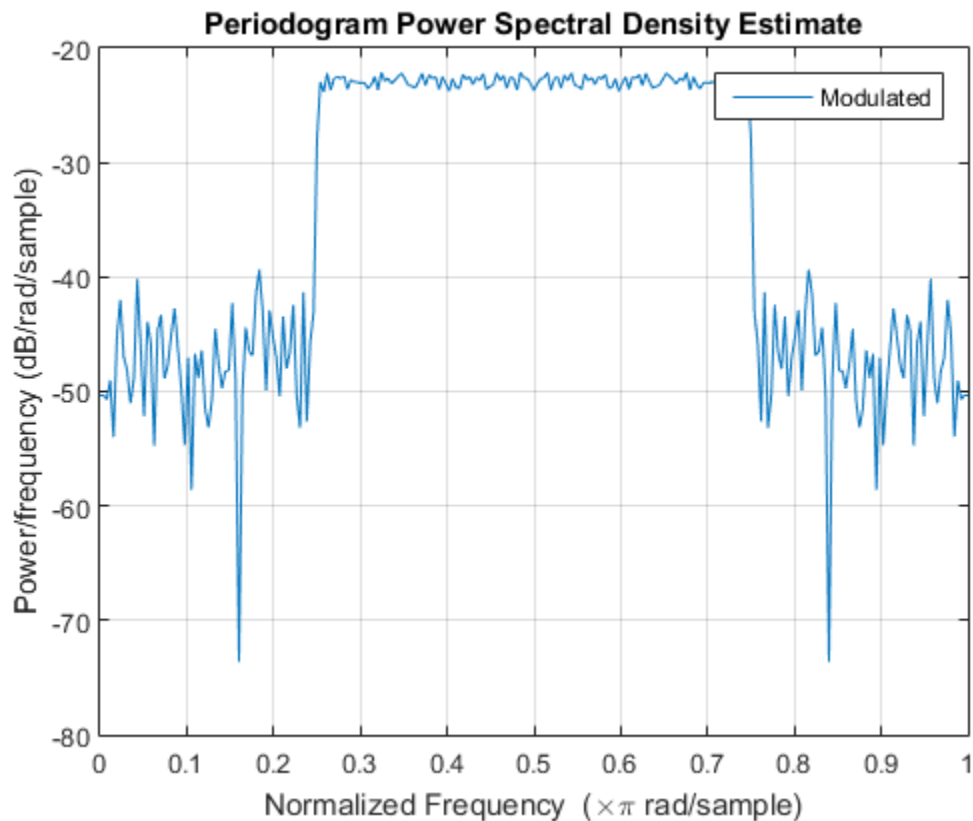
SSB amplitude modulation reduces the bandwidth of the signal by half. To carry out SSB amplitude modulation, you must first compute the Hilbert transform of the signal. Then, amplitude modulate the signal using a sine with the same carrier frequency, ω_c, as before, and add it to the previous signal.
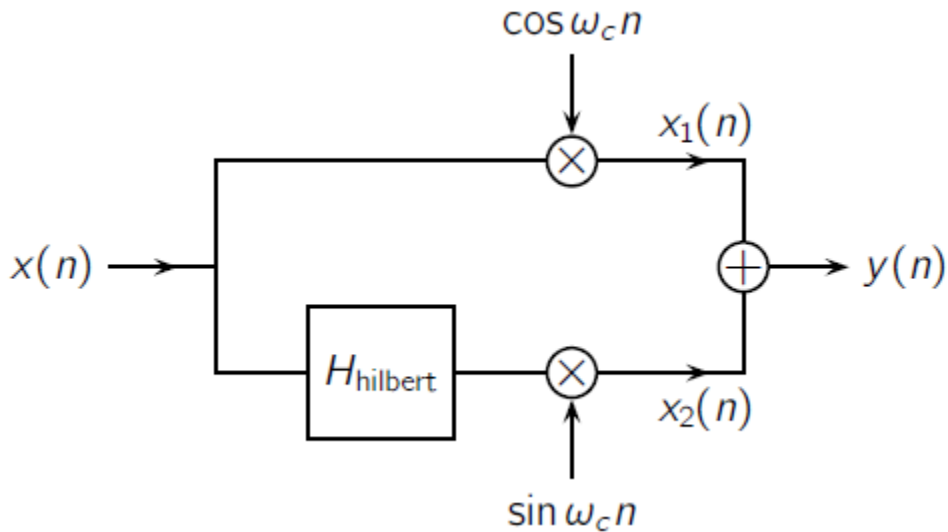
Design a Hilbert transformer using the `designfilt` function. Specify a filter order of 64 and a transition width of 0.1. Filter the signal.

```
Hhilbert = designfilt('hilbertfir','FilterOrder',64, ...
    'TransitionWidth',0.1);

xh = filter(Hhilbert,x);
```

Use the `grpdelay` function to determine the delay, `gd`, introduced by the filter. Compensate for the delay by discarding the first `gd` points of the filtered signal and padding with zeros at the end. Amplitude modulate the result and add it to the original. Compare the PSDs.

```
gd = mean(grpdelay(Hhilbert));
xh = xh(gd+1:end);
eh = zeros(size(x));
eh(1:length(xh)) = xh;

x2 = eh.*sin(wc*n)*sqrt(2);

y = x1+x2;

periodogram([x1;y]')
legend('Modulated','SSB')
```
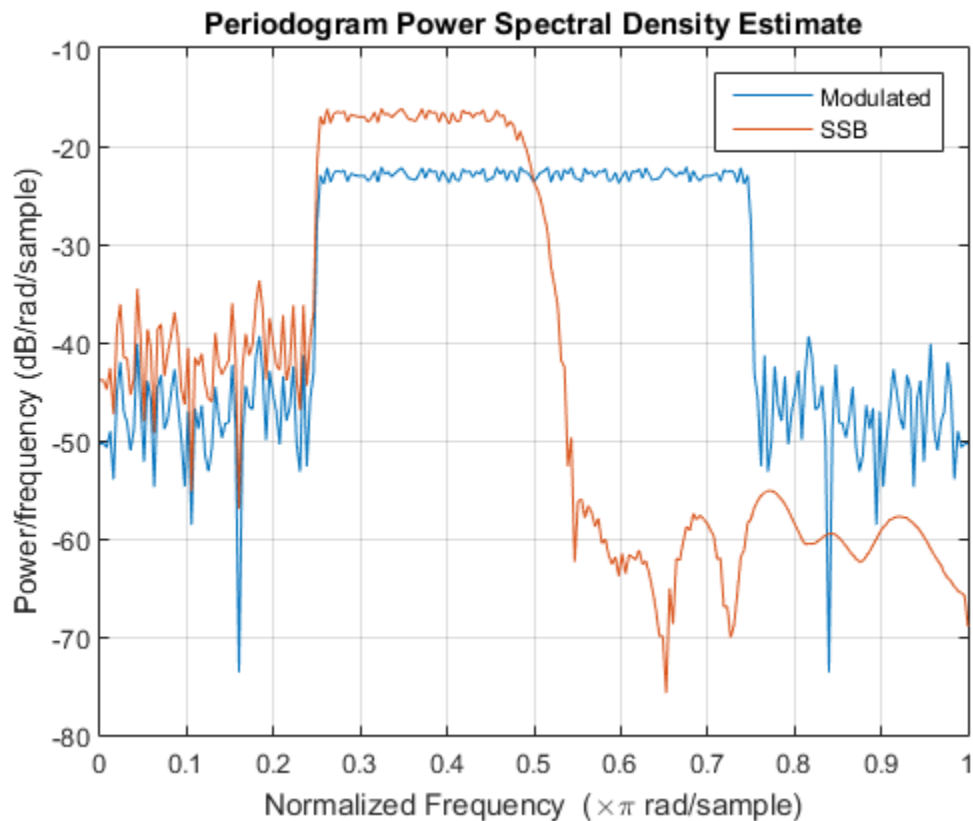
Downconvert the signal and estimate the PSD.

```
ym = y.*cos(wc*n)*sqrt(2);

periodogram(ym)
legend('Downconverted')
```
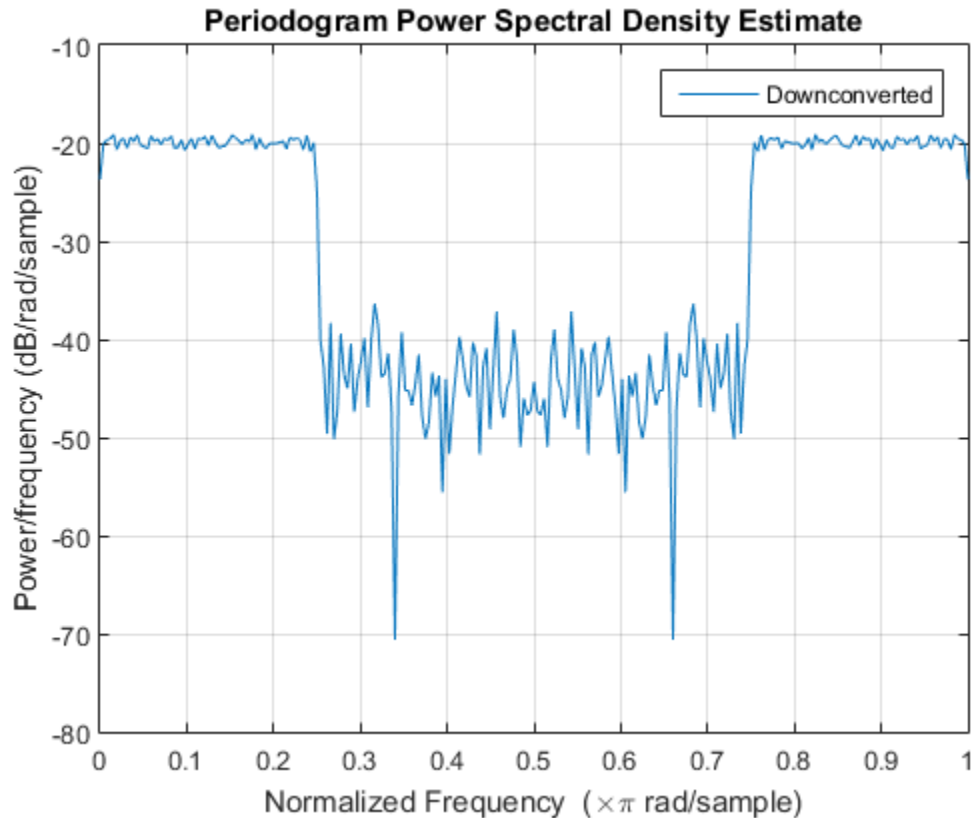
Lowpass filter the modulated signal to recover the original. Specify a 64th-order FIR lowpass filter with a cutoff frequency of $\pi/2$. Compensate for the delay introduced by the filter.

```
d = designfilt('lowpassfir','FilterOrder',64, ...
    'CutoffFrequency',0.5);
dem = filter(d,ym);

gd = mean(grpdelay(d));
dem = dem(gd+1:end);

dm = zeros(size(x));
dm(1:length(dem)) = dem;
```

Estimate the PSD of the filtered signal and compare it to that of the original.

```
periodogram([x;dm]')
legend('Original','Recovered')
```
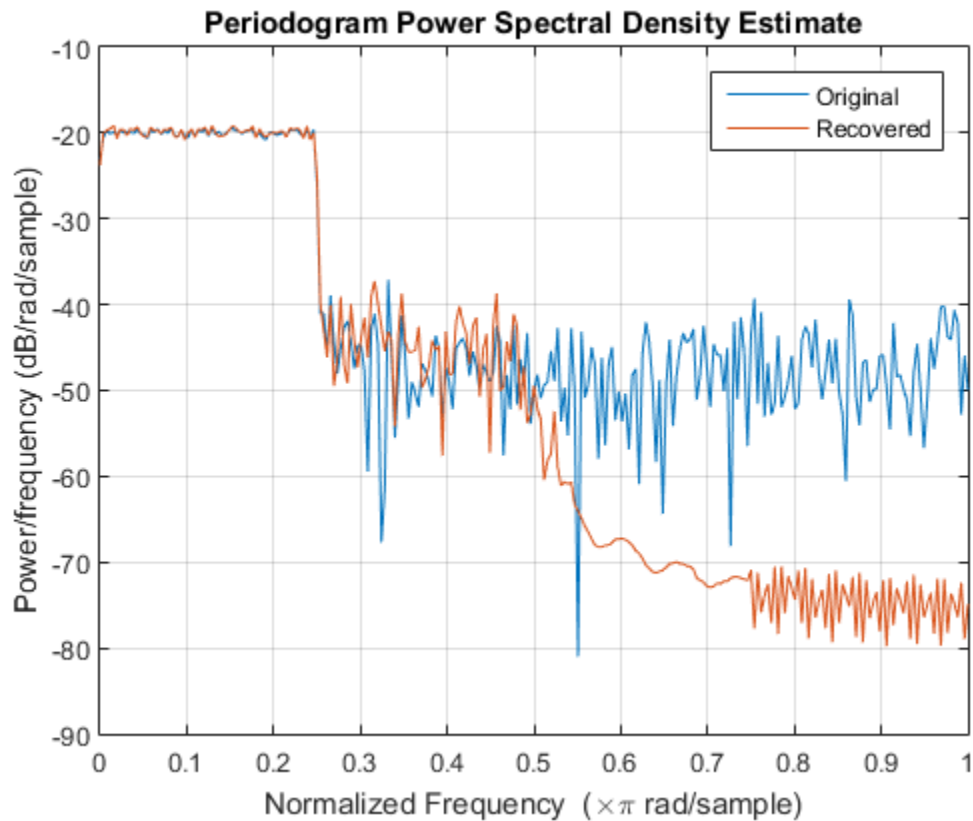

Use the `snr` function to compare the signal-to-noise ratios of the two signals. Plot the two signals in the time domain.

```
snrOrig = snr(x,noise)
snrRecv = snr(dm,noise)

plot(n,[x;dm]')
legend('Original','Recovered')
```

```
axis tight
```

```
snrOrig =

    20.0259


snrRecv =

    20.1373
```
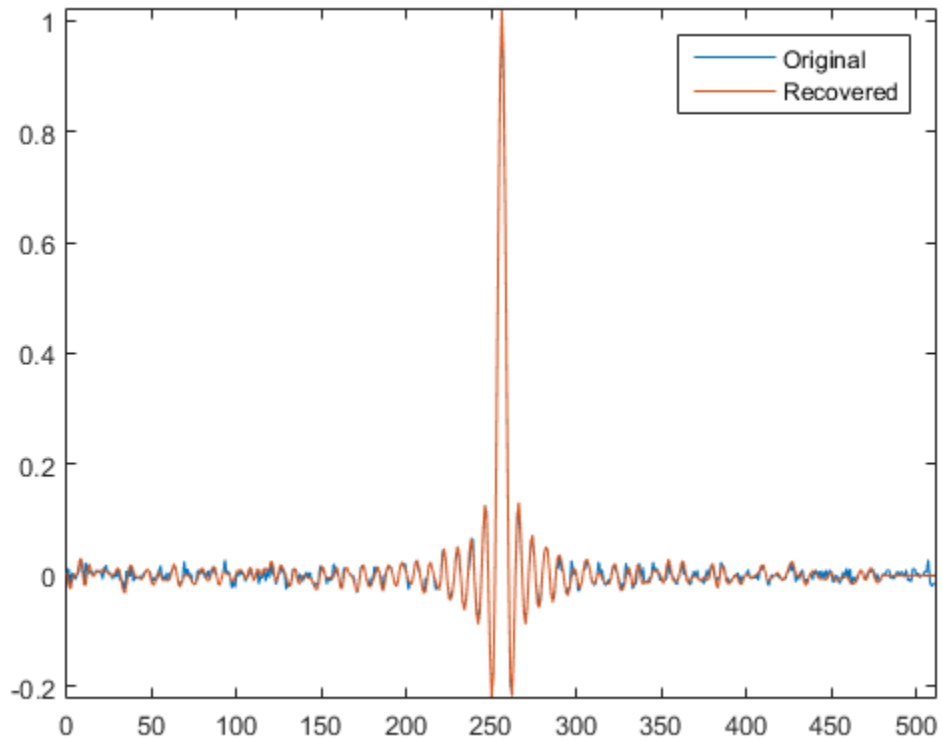

References

Buck, John R., Michael M. Daniel, and Andrew C. Singer. *Computer Explorations in Signals and Systems Using MATLAB*. 2nd Edition. Upper Saddle River, NJ: Prentice Hall, 2002.

Signal Generation

Display Time-Domain Data in Signal Browser

In this section...
"Import and Display Signals" on page 15-3
"Configure the Signal Browser Properties" on page 15-6
"Modify the Signal Browser Display" on page 15-9
"Inspect Your Data (Scaling the Axes and Zooming)" on page 15-10

This example shows how to use and configure the SPTool Signal Browser to display time-domain signals. First, open SPTool by typing

```
sptool
```
at the MATLAB command line. SPTool opens.
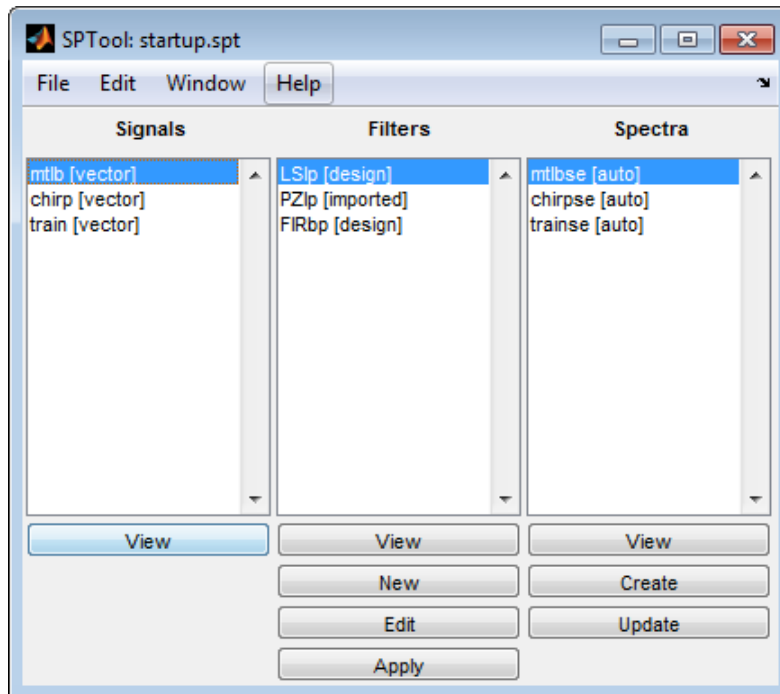
Import and Display Signals

Displaying Multiple Signals

You can display multiple signals in the Signal Browser by selecting more than one entry in the SPTool **Signals** list. In the **Signals** list, first select `mtlb [vector]`. Next, press the **Ctrl** key, and select `chirp [vector]`. Finally, click the **View** button. The Signal Browser opens, displaying both signals.

Any signals that you select in SPTool are now visible in the Signal Browser. You can also select multiple entries using the **Shift** key. In the SPTool **Signals** list, first select `mtlb [vector]`. Next, press the **Shift** key, and select `train [vector]`. The Signal Browser automatically updates to display all three signals, as shown in the following figure.
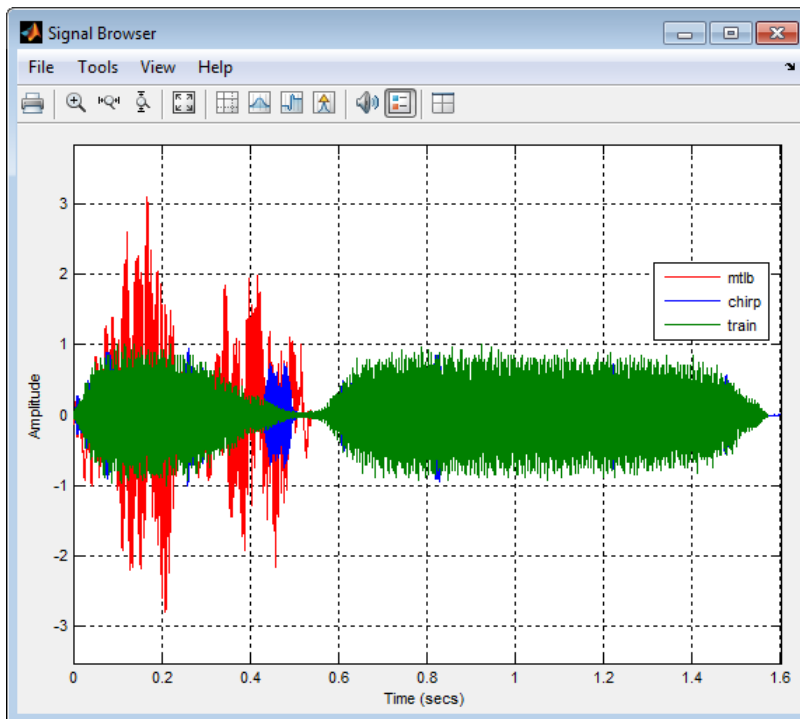
Importing New Signals

Using SPTool, you can import signals from variables in the MATLAB workspace. First, create a 3-second signal, sampled at 10 kHz, that is the sum of two sine waves. At the MATLAB command line, enter the following commands:

```
Fs = 1e4;
t = 0:1/Fs:3;
s = sum(sin(2*pi*[350;440]*t));
```

To import the signal from these variables, in the SPTool menu, select **File** > **Import**. Alternatively, you can press the **Ctrl+I** keyboard shortcut. The Import to SPTool dialog box opens.
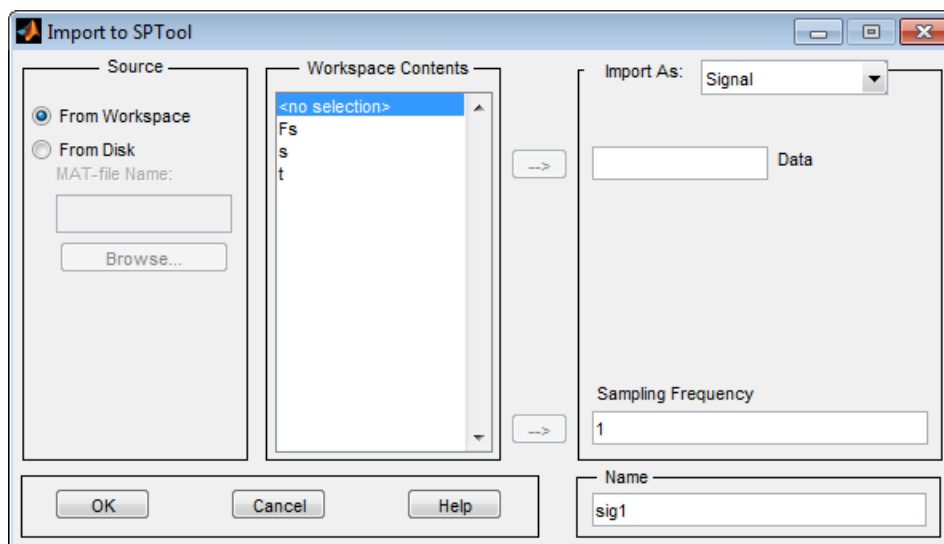
From the **Workspace Contents** list, select s. Click the right arrow (-->) button to the left of the **Data** box. Next, from the **Workspace Contents** list, select Fs. Click the right arrow (-->) button to the left of the **Sampling Frequency** box. You can assign the signal a name in the **Name** box, but since you will rename the signal later in this example, leave it as sig1 for now.

Click **OK**. The SPTool **Signals** list now contains a signal named sig1 [vector].

Selecting a Signal and Playing Audio

In the SPTool **Signals** list, select `sig1 [vector]`, and click the **View** button. The Signal Browser reappears in front. To play audio for the signal, click the Play selected signal (🔊) button. Signal Browser sends the audio signal to the speaker. For more information, see `sound` in the MATLAB documentation.

If you have multiple signals selected in SPTool, you can use the **Trace Selection** panel to choose which signal to make active. In the SPTool **Signals** list, press the **Ctrl** key, and select `mtlb [vector]`. The Signal Browser now displays two signals. In the Signal Browser menu, select **Tools** > **Measurements** > **Trace Selection**. The **Trace Selection** panel appears as shown in the following figure.
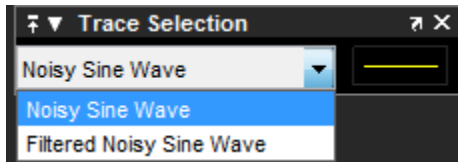
In the **Trace Selection** panel drop-down list, select `sig1`. Then, click the Play selected signal (🔊) button. Signal Browser sends the audio signal to the speaker.

Note: To hear audio when you click the Play selected signal (🔊) button, your computer sound card must be able to support the sample rate of the signal. In this example, the sample rate of the signal, `sig1 [vector]`, is 10 kHz. If your sound card supports this or a greater sample rate, such as 44.1 kHz, then you can hear the audio on your speaker. For more information, see `sound` in the MATLAB documentation.

Change Signal Names from the Legend

When multiple signals are displayed, Signal Browser shows a legend by default. To turn off the legend, click the Show all legends (▦) button. Click the button again to turn the legend back on. You can modify the names of the signal directly in the legend. To do so, when the legend is visible, click and drag it to any location on the display.

You can change the name of any signal directly within the legend. In the legend, double-click the signal name `sig1`. A cursor appears, indicating that you can now change this

name. Highlight the text, and type `Dial Tone`. The legend now shows `Dial Tone` as the name of that signal. In the SPTool **Signals** list, the last item is now also named `Dial Tone`.

Configure the Signal Browser Properties

First, configure the appearance of the Signal Browser window. In the SPTool **Signals** list, first select `mtlb [vector]`. Next, press the **Shift** key, and select `train [vector]`. The Signal Browser automatically updates to display all three signals.

Multiple Displays

You can display multiple signals on different displays in the Signal Browser window. In the SPTool **Signals** list, first ensure all three signals are selected. In the Signal Browser toolbar, click the Layout ( ) button. Select row 3, column 1, as shown in the following figure.

After you make this selection, the Signal Browser is separated into three displays.

Configure Appearance

In the Signal Browser menu, select **View** > **Properties**. The Visuals:Time Domain Options dialog box opens, as shown in the following figure.
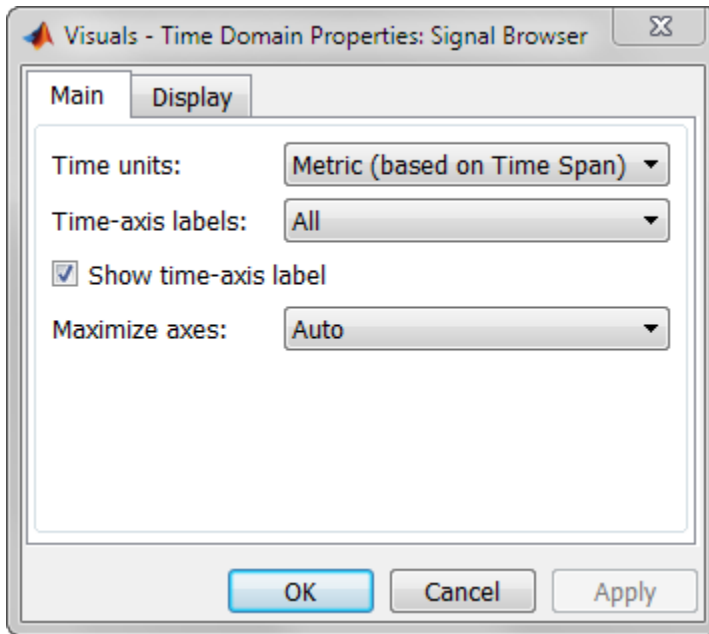
In the Visuals:Time Domain Options dialog box, click the **Main** tab. Choose the appropriate parameter settings for the **Main** tab, as shown in the following table.

| Parameter	Setting
Time units	`Metric (based on Time Span)`
Show time-axis labels	`Bottom Displays Only`
Maximize axes	`On`

When you change the **Maximize axes** parameter to `On`, the axes are expanded to fill the entire display. To conserve space, titles and axis labels are not shown in each display. Click **Apply**.

Set Display Properties

In the Visuals:Time Domain Options dialog box, click the **Display** tab. You can change the value of the **Select display** parameter to make different settings for each display. Set the parameters to the values shown in the following table.

Parameter	Display 1 Setting	Display 2 Setting	Display 3 Setting
Select display	1	2	3
Title	`mtlb`	`chirp`	`train`
Show legend	Selected	Selected	Selected
Show grid	Selected	Selected	Selected
Plot signal(s) as magnitude and phase	Cleared	Cleared	Cleared
Minimum Y-limit	`-2.5`	`-1`	`-1.5`
Maximum Y-limit	`2.5`	`1`	`1.5`
Y-axis label	`Amplitude`	`Amplitude`	`Amplitude`

Click **OK** to save your changes and close the Visuals:Time Domain Options dialog box. The Signal Browser appears as shown in the following figure.
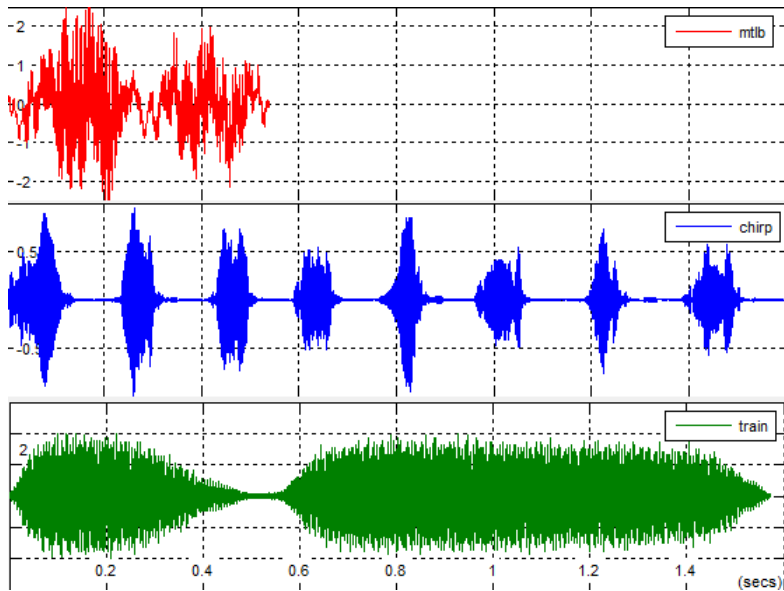
Modify the Signal Browser Display

Use the Style dialog box to modify the appearance of the axes and the lines for each of the selected signals in SPTool. In the Signal Browser menu, select **View > Style**. The Style dialog box opens, as shown in the following figure.
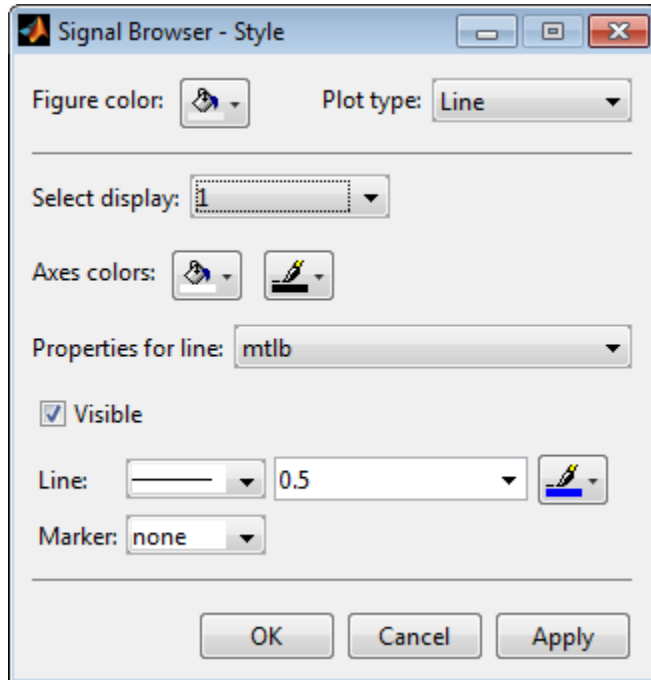
Modify Axes Colors and Line Properties

You can change the value of the **Select display** parameter to make different settings for each display. Set the parameters to the values shown in the following table.

Parameter	Display 1 Setting	Display 2 Setting	Display 3 Setting
Select display	1	2	3
Axes background color	Black	Black	Black
Ticks, labels, and grid colors	White	White	White

Parameter	Display 1 Setting	Display 2 Setting	Display 3 Setting
Line color	Yellow	Cyan	Magenta

These settings enable the Signal Browser to display line colors in the same manner as the Simulink Scope block. Click **OK** to save your changes and close the Style dialog box. The Signal Browser now appears as shown in the following figure.
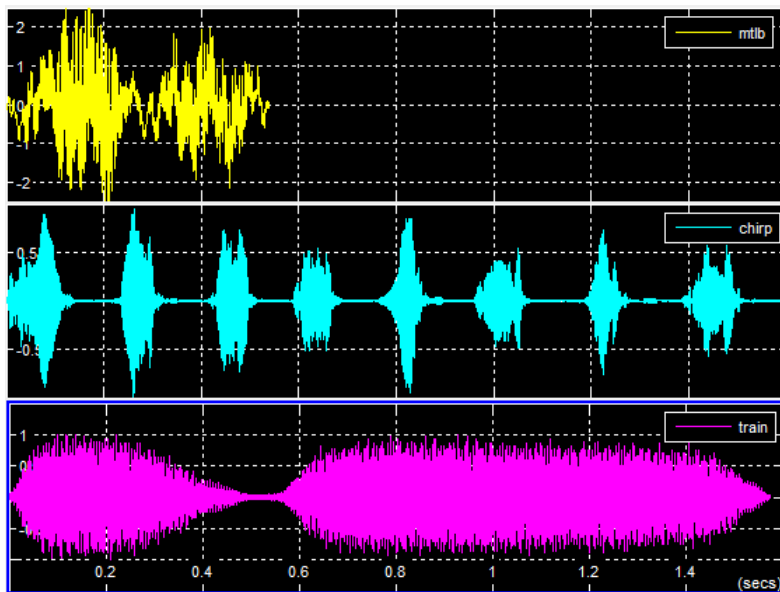
Show and Hide Toolbar

To hide the toolbar, from the Signal Browser menu, select **View** > **Toolbar**. Doing so removes the toolbar from the Signal Browser window and also removes the check mark beside the **Toolbar** option in the **View** menu. You can choose to show the toolbar again at any time by selecting **View** > **Toolbar**.

Inspect Your Data (Scaling the Axes and Zooming)

So far, you have manually set the *y*-axis limits. Use one of the following options to let Signal Browser scale the axes:

- From the Signal Browser menu, select **Tools** > **Scale Axes Limits**.

- From the Signal Browser toolbar, click the Scale Axes Limits (⬍) button.

- With the Signal Browser as your active window, press **Ctrl + A**.

Use the Zoom Tools

The zoom tools allow you to zoom in simultaneously in the directions of both the *x*- and *y*-axes , or in either direction individually. For example, to zoom in on the signal between 0 and 0.5 seconds, you can use the **Zoom X** option.

- To activate the **Zoom X** tool, select **Tools** > **Zoom X**, or press the corresponding toolbar button (ⁱ⁰ᵒⁱ). The Signal Browser indicates that the **Zoom X** tool is active by indenting the toolbar button and placing a check mark next to the **Tools** > **Zoom X** menu option.

- Next, zoom in on the region between 0 and 0.5 seconds. In the Signal Browser window, click on the 0-second mark and drag to the 0.5-second mark. All three displays reflect this new *x*-axis setting, as shown in the following figure.
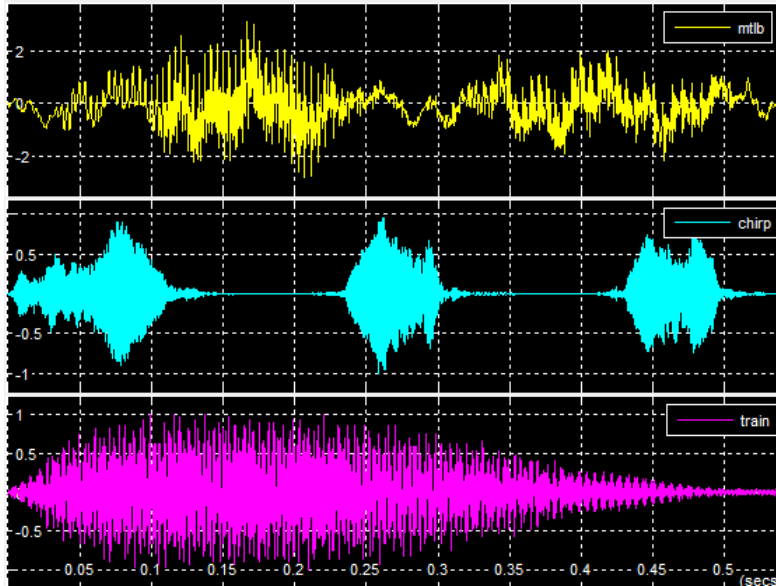
- To zoom out of the Signal Browser window, right-click inside the window, and select **Zoom Out**. Alternatively, you can return to the original view of your signal by right-clicking inside the Signal Browser window and selecting **Reset to Original View**.

Signal Measurement

RMS Value of Periodic Waveforms

This example shows how to find the root mean square (RMS) value of a sine wave, a square wave, and a rectangular pulse train using `rms`. The waveforms in this example are discrete-time versions of their continuous-time counterparts.

Create a sine wave with a frequency of π/4 radians/sample. The length of the sine wave is 16 samples, which equals two periods of the sine wave.
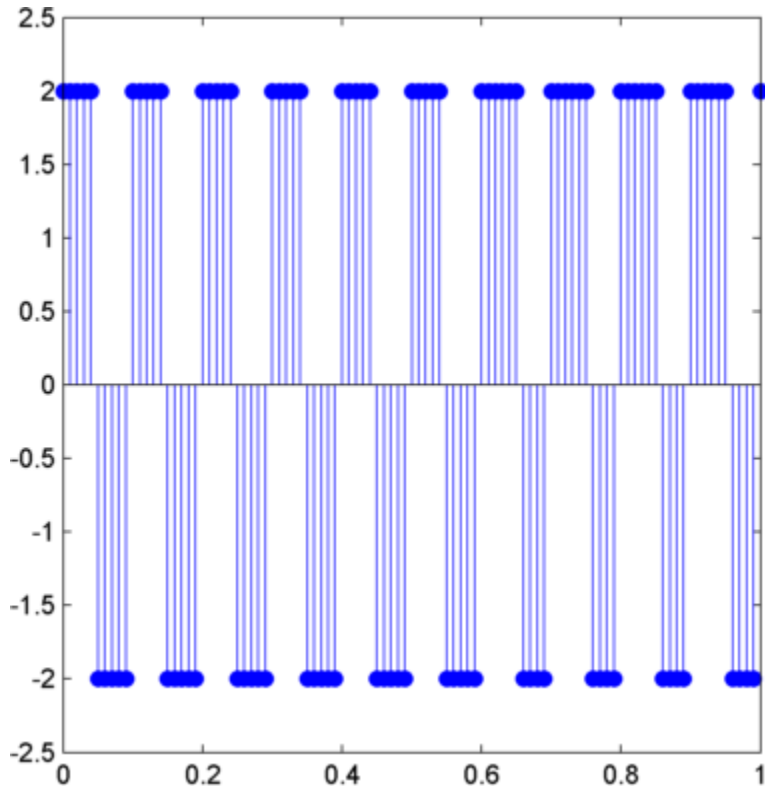
```
n = 0:15;
x = cos(pi/4*n);
```

Compute the RMS value of the sine wave.

```
rmsval = rms(x)
```

The RMS value is equal to 0.7071 as expected.

Create a periodic square wave with a period of 0.1 seconds. The square wave values oscillate between –2 and 2

```
t = 0:0.01:1;
x = 2*square(2*pi*10*t);
stem(t,x,'markerfacecolor',[0 0 1]); axis([0 1 -2.5 2.5]);
```
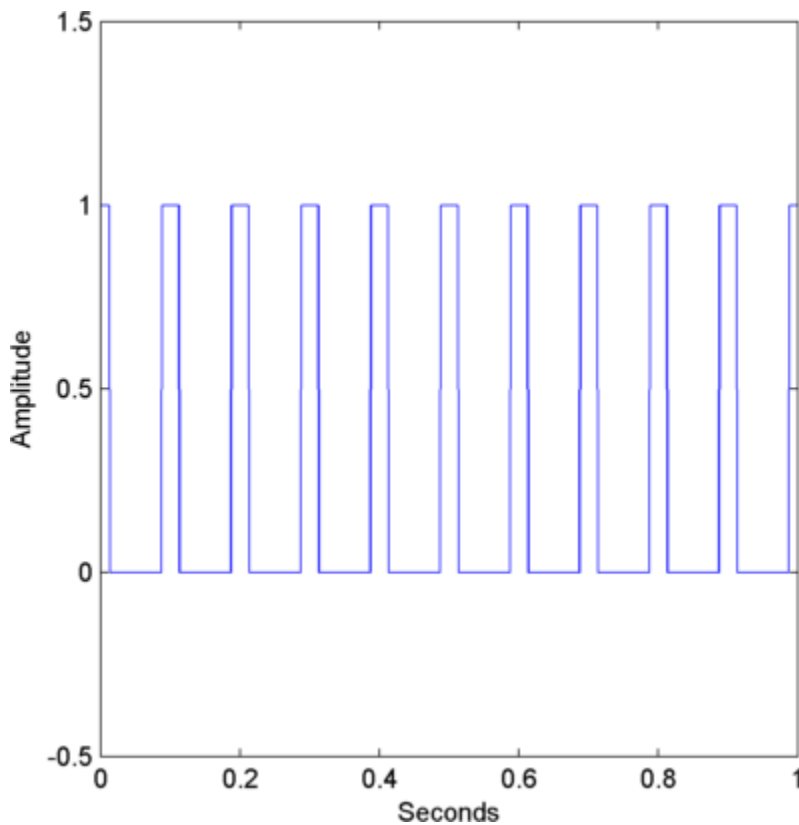
Find the RMS value.

```
rmsval = rms(x)
```

The RMS value agrees with the theoretical value of 2.

Create a rectangular pulse train sampled at 1 kHz with the following parameters: the pulse is on, or equal to 1, for 0.025 seconds and off, or equal to 0, for 0.075 seconds in each 0.1 second interval. This means the pulse period is 0.1 seconds and the pulse is on for 1/4 of that interval. This is referred to as the *duty cycle*. Use `pulstran` to create the rectangular pulse train.

```
t = 0:0.001:(10*0.1);
pulsewidth = 0.025;
pulseperiods = [0:10]*0.1;
x = pulstran(t,pulseperiods,@rectpuls,pulsewidth);
```

```
plot(t,x); axis([0 1 -0.5 1.5]);
xlabel('Seconds'); ylabel('Amplitude');
```


Find the RMS value and compare it to the RMS of a continuous-time rectangular pulse waveform with duty cycle 1/4 and peak amplitude 1.

```
rmsval = rms(x)
thrms = sqrt(1/4)
```

The observed RMS value and the RMS value for a continuous-time rectangular pulse waveform are in good agreement.

Slew Rate of Triangular Waveform

This example shows how to use the slew rate as an estimate of the rising and falling slopes of a triangular waveform. Create three triangular waveforms. One waveform has rising-falling slopes of +/- 2, one waveform has rising-falling slopes of +/- 1/2, and one waveform has a rising slope of +2 and a falling slope of -1/2. Use `slewrate` to find the slopes of the waveforms.

Create a triangular waveform with rising-falling slopes of +/- 2. Set the sampling interval to 0.01 seconds, which corresponds to a sampling frequency of 100 hertz.

```
t = 0:0.01:1;
x = 2*t;
x = [x fliplr(x)];
tnew = [t t+1.01];
plot(tnew,x); xlabel('Time');
ylabel('Amplitude');
```
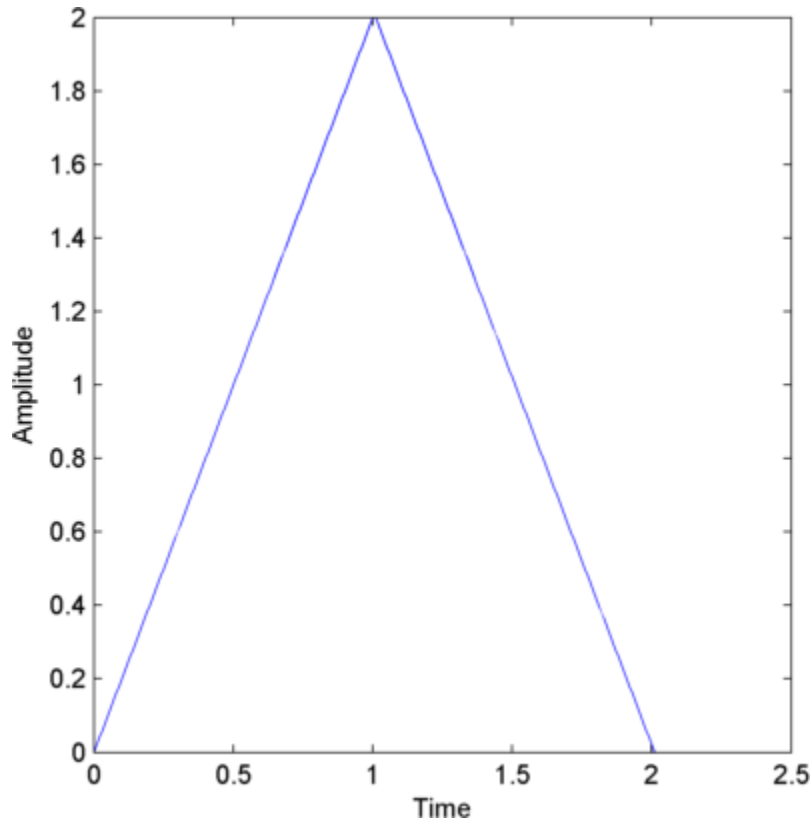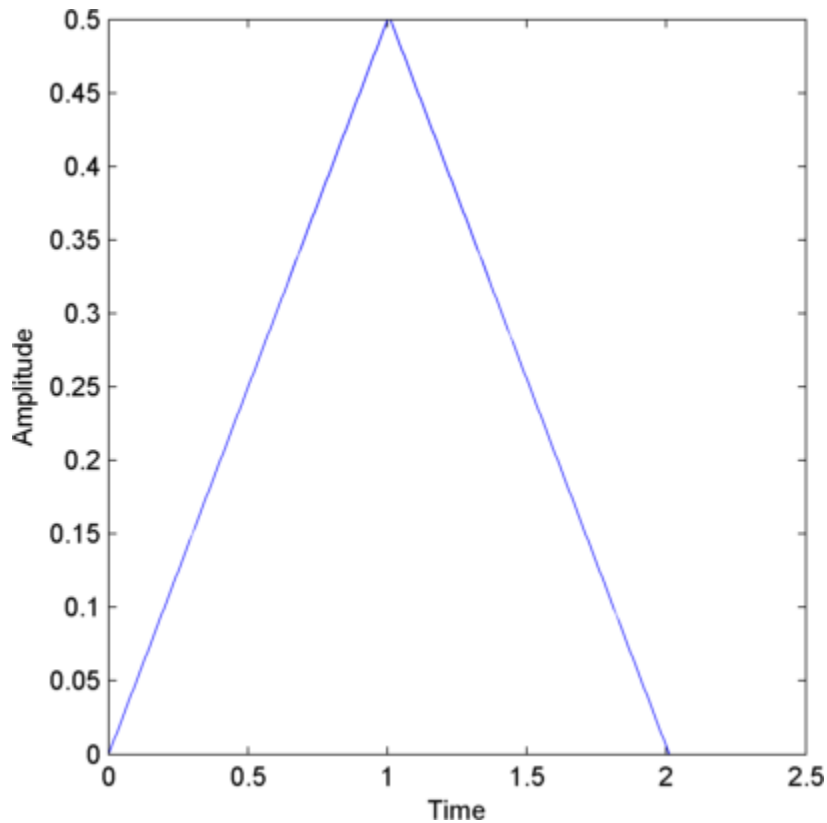
Calculate the slew rate for the triangular waveform. Input the sampling frequency (100 Hz) to obtain the correct positive and negative slope values.

```
s = slewrate(x,100)
```

Create a triangular waveform with slopes of +/- 1/2. Set the sampling interval to 0.01 seconds, which corresponds to a sampling frequency of 100 hertz.

```
t = 0:0.01:1;
x = 1/2*t;
x = [x fliplr(x)];
tnew = [t t+1.01];
plot(tnew,x);   xlabel('Time');
ylabel('Amplitude');
```

Calculate the slew rate for the triangular waveform. Input the sampling frequency (100 Hz) to obtain the correct positive and negative slope values.

```
s = slewrate(x,100)
```

Create a triangular waveform with a rising slope of +2 and a falling slope of -1/2. Set the sampling increment to 0.01 seconds, which corresponds to a sampling frequency of 100 hertz.

```
t = 0:0.01:1;
x = 2*t;
t1 = 1:0.01:5;
x1 = -1/2*(t1-1)+2;
y = [x x1];
tnew = [t t1];
```

```
plot(tnew,y);  xlabel('Time');
ylabel('Amplitude');
```
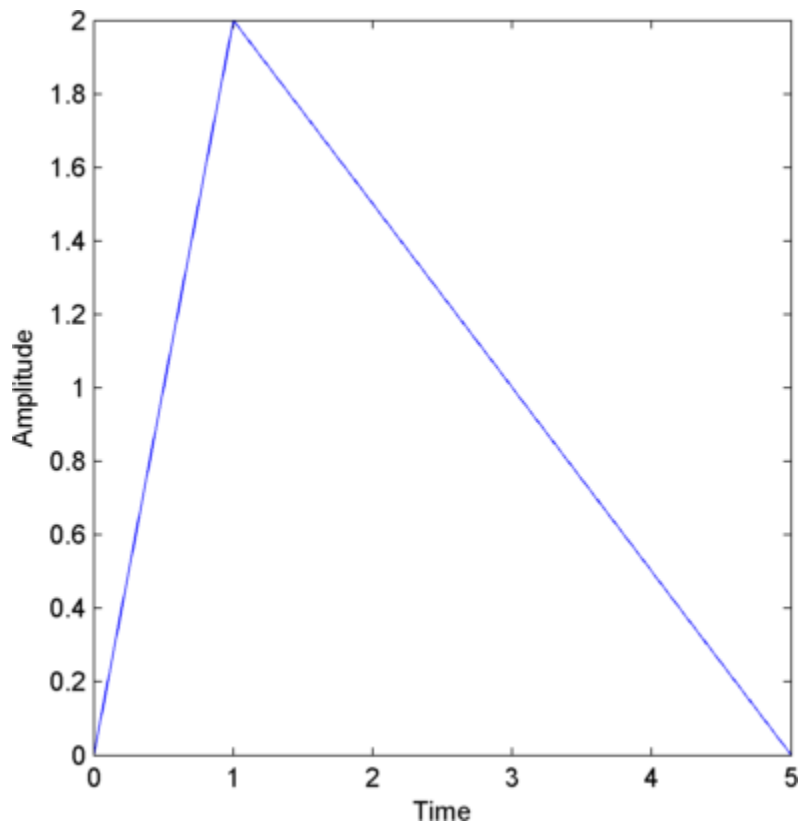

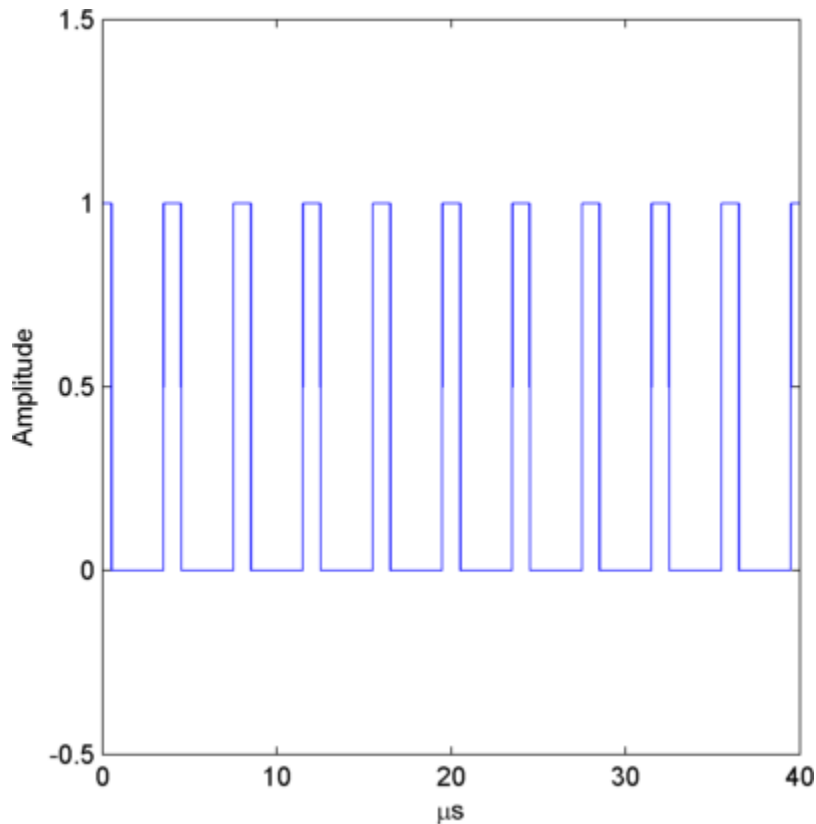Determine the slew rate.

```
s = slewrate(y,100)
```

The first element of s is the rising slope and the second element is the falling slope.

Duty Cycle of Rectangular Pulse Waveform

This example shows how to create a rectangular pulse waveform and measure the duty cycle. You can think of a rectangular pulse waveform as a sequence of *on* and *off* states. One pulse period is the total duration of an *on* and *off* state. The pulse width is the duration of the *on* state. The duty cycle is the ratio of the pulse width to the pulse period. The duty cycle for a rectangular pulse describes the fraction of time that the pulse is *on* in one pulse period.

Create a rectangular pulse sampled at 1 gigahertz. The pulse is on, or equal to 1, for a duration of 1 microsecond. The pulse if off, or equal to 0, for a duration of 3 microseconds. The pulse period is 4 microseconds. Measure the duty cycle of the waveform.

```
Fs = 1e9;
t = 0:1/Fs:(10*4e-6);
pulsewidth = 1e-6;
pulseperiods = [0:10]*4e-6;
x = pulstran(t,pulseperiods,@rectpuls,pulsewidth);
plot(t.*1e6,x); axis([O 40 -O.5 1.5]);
xlabel('\mus'); ylabel('Amplitude');
```

Determine the duty cycle using `dutycycle`. Input both the pulse waveform and the sampling frequency to output the duty cycle. `dutycycle` outputs a duty cycle value for each detected pulse.

```
D = dutycycle(x,Fs)
```

In this example, the duty cycle for each of the 10 detected pulses is identical and equal to 0.25. This is the expected duty cycle because the pulse is on for 1 microsecond and off for 3 microseconds in each 4 microsecond period. Therefore, the pulse is on for 1/4 of each period. Expressed as a percentage, this is equal to a duty cycle of 25%.

Calling `dutycycle` with no output arguments produces a plot with all the detected pulse widths marked.

```
dutycycle(x,Fs);
```

Using the same sampling rate and pulse period, vary the pulse on time (pulse width) from 1 to 3 microseconds in a loop and calculate the duty cycle. Plot the pulse waveforms and display the duty cycle value in the plot title for each step through the loop. You see the duty cycle increase from 0.25 (1/4) to 0.75 (3/4) as the pulse width increases.

```
pulsewidths = 1e-6:1e-6:3e-6;
for nn = 1:length(pulsewidths)
    x = pulstran(t,pulseperiods,@rectpuls,pulsewidths(nn));
    plot(t.*1e6,x); axis([0 40 -0.5 1.5]);
    xlabel('\mus'); ylabel('Amplitude');
    D = dutycycle(x,Fs);
    title(['Duty cycle is ' num2str(mean(D)) ]);
    pause(1);
end
```

Estimate State for Digital Clock

This example shows how to estimate the high and low *state* levels for digital clock data. In contrast to analog voltage signals, signals in digital circuits have only two states: HIGH and LOW. Information is conveyed by the pattern of high and low state levels.

Load `clockex.mat` into the MATLAB workspace. `clockex.mat` contains a 2.3 volt digital clock waveform sampled at 4 megahertz. Load the clock data into the variable, `x`, and the vector of sampling times in the variable, `t`. Plot the data.

```
load('clockex.mat','x','t')
stem(t,x,'markerfacecolor',[0 0 1]); xlabel('Seconds'); ylabel('Volts');
```
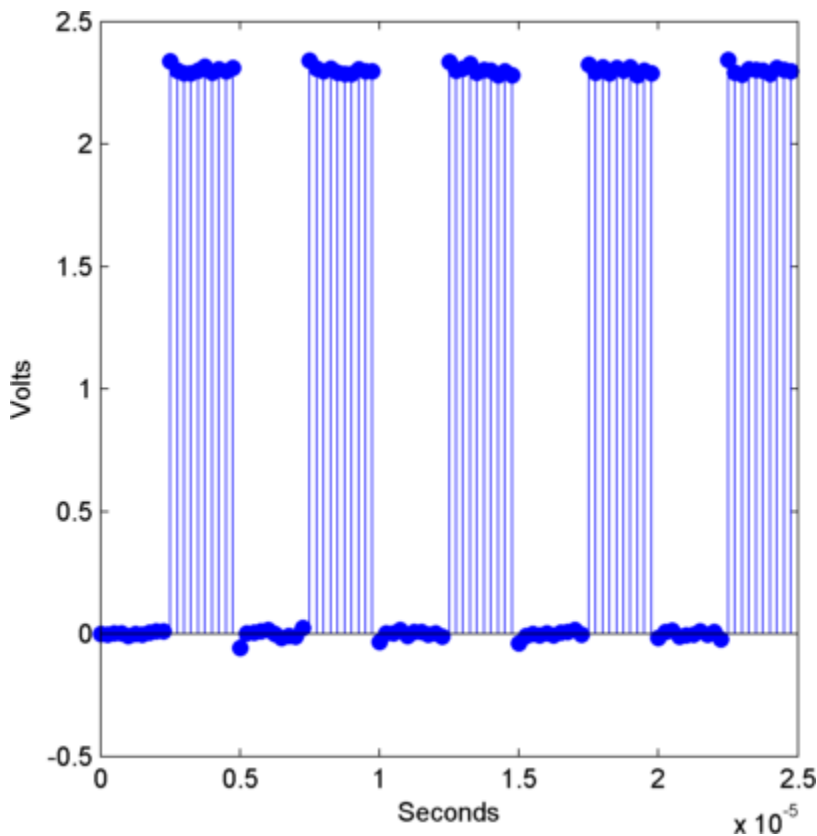

Determine the high and low state levels for the clock data using `statelevels`.

```
levels = statelevels(x)
```

The low-state level estimate is 0.0027 volts and the high-state level estimate is 2.3068 volts. This is the expected result for the 2.3 volt clock data, where the noise-free low-state level is 0 volts, and the noise-free high-state level is 2.3 volts.

Use the estimated state levels to convert the voltages into a sequence of 0s and 1s. The sequence of 0s and 1s is a binary waveform representation of the two states. To make the assignment, use the following decision rule. Assign any voltage within a 3%-tolerance region of the low-state level the value 0, and any voltage within a 3%-tolerance region of the high-state level the value 1.

Determine the 3%-tolerance region around the low-state level.

```
lowtol = [levels(1)-3/100*(levels(2)-levels(1)) ...
    levels(1)+3/100*(levels(2)-levels(1))];
```

Determine the 3%-tolerance region around the high-state level.

```
hightol = [levels(2)-3/100*(levels(2)-levels(1)) ...
    levels(2)+3/100*(levels(2)-levels(1))];
```

Use logical indexing to determine the voltages within a 3%-tolerance region of the low-state level and the voltages within a 3%-tolerance region of the high-state level. Assign the value 0 to the voltages within the tolerance region of the low-state level and 1 to the voltages within the tolerance region of the high-state level. Plot the result.

```
y = zeros(size(x));
y(x>= lowtol(1) & x<= lowtol(2)) = 0;
y(x>= hightol(1) & x<= hightol(2)) = 1;
subplot(211)
stem(t,x,'markerfacecolor',[0 0 1]); ylabel('Volts');
subplot(212)
stem(t,y,'markerfacecolor',[0 0 1]); ylabel('{0,1}'); xlabel('Seconds');
set(gca,'ytick',[0 1])
```

16-13
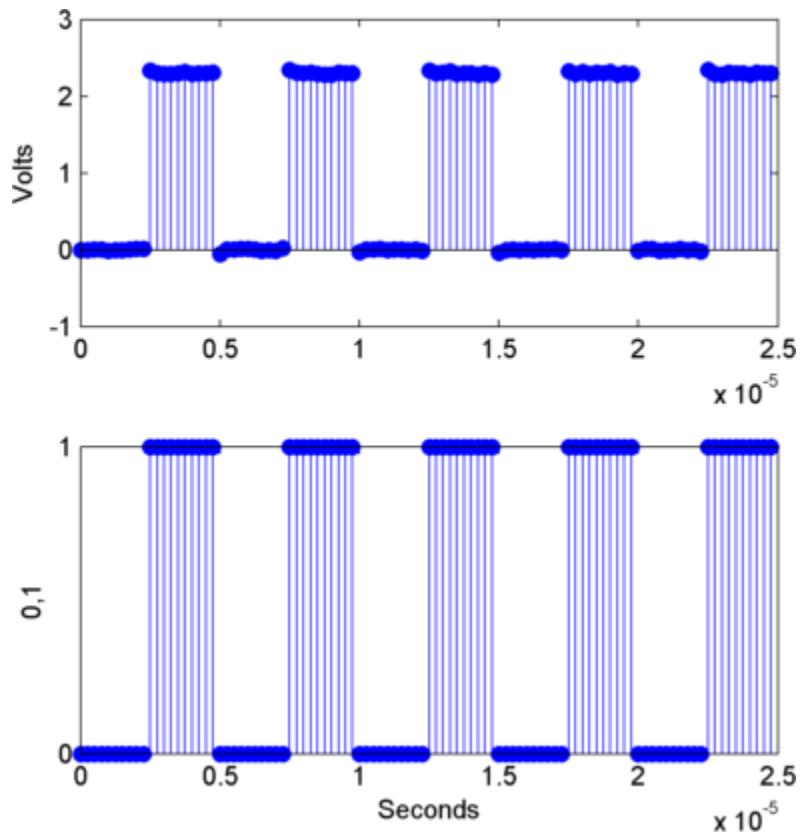
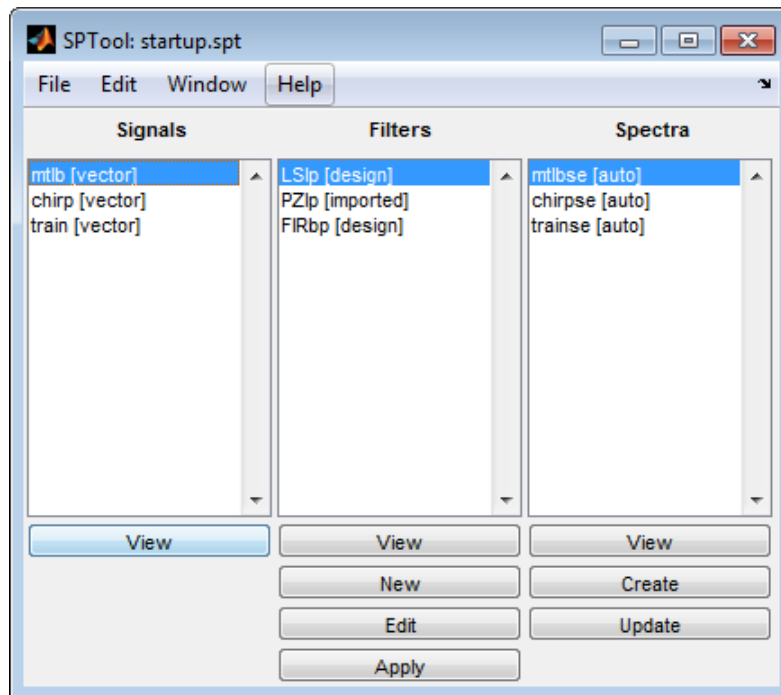The decision rule has assigned all the voltages to the correct state.

Calculate Settling Time with Signal Browser

This example shows how to use the Bilevel Measurements panel in the SPTool Signal Browser to find the settling time of a clock signal.

First, open SPTool by typing the following at the MATLAB command line.

```
sptool
```

SPTool opens.

In this example, you import a clock signal from the MAT-file named `clockex.mat`. In the SPTool menu, select **File > Import**. Alternatively, you can press the **Ctrl+I** keyboard shortcut. The Import to SPTool dialog box opens.

1 Under **Source**, click **From Disk**.
2 In the **MAT-file** box, type `clockex`, and press **Enter**. The variables `x` and `t` appear under **File Contents**.

3 Under **File Contents**, click x. Click the right arrow (⟶) button to the left of the **Data** box.

4 In the **Sampling Frequency** box, type 4000000.

5 In the **Name** box, type Clock.

6 Click **OK**. The SPTool **Signals** list now contains a signal named Clock [vector].

7 In SPTool, in the **Signals** list, select Clock [vector], and click the **View** button. The Signal Browser appears and displays the clock signal.

Because you selected only one signal, the legend is not needed. Turn off the legend by clicking the Show All Legends ( ) button. Then, use the Style dialog box to modify the appearance of the axes and the lines for the signal. In the Signal Browser menu, select **View > Style**.

Parameter	Display 1 Setting
Axes background color	Black
Ticks, labels, and grid colors	Dark Gray
Line color	Yellow

To show the **Bilevel Measurements** panel, in the Signal Browser menu, select **Tools > Measurements > Bilevel Measurements**. To collapse the **Transitions** pane, click the pane collapse button (▼) next to that label. To expand the **Settings** pane and the **Overshoots / Undershoots** pane, click the pane expand button (▶) next to each label. The Signal Browser appears as shown in the following figure.
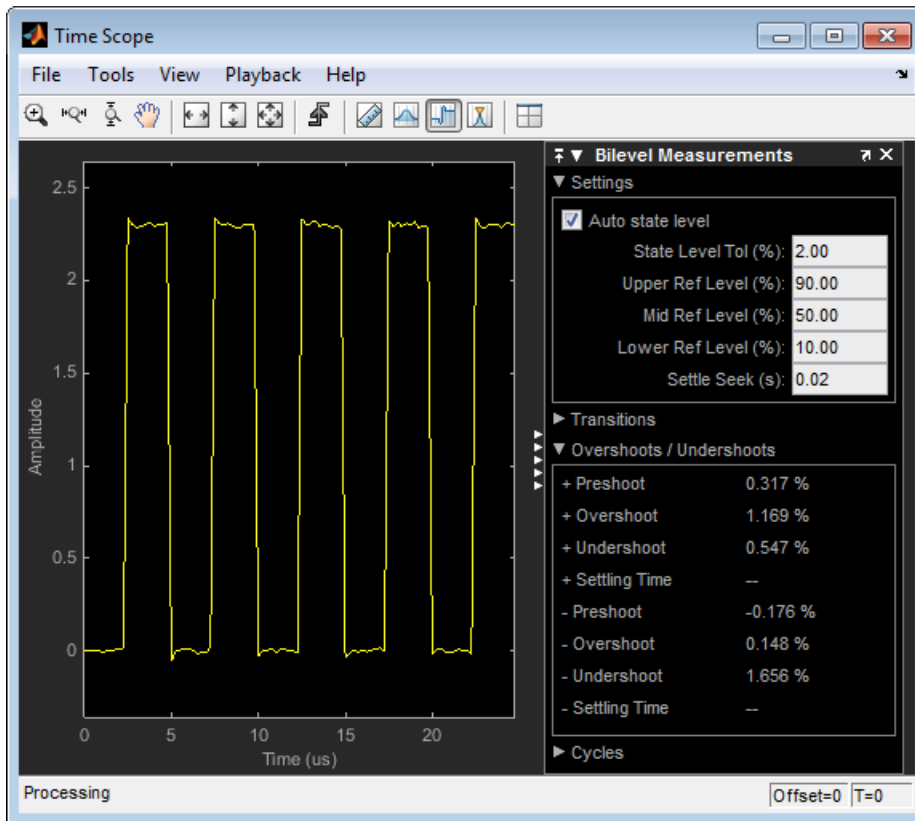
The value for the rising edge **Settling Time** parameter does not appear in the **Overshoots / Undershoots** pane because the **Settle Seek** parameter is too large. The **Settle Seek** value is longer than the entire simulation duration. Enter a value for settle seek of `2e-6`, and press **Enter**. Signal Browser now displays a rising edge settling time value of `118.392 ns`.

This settling time value displayed is actually the statistical average of the settling times for all five rising edges. To display the settling time for only one rising edge, you can zoom in on that transition. In the Signal Browser toolbar, click the Zoom X button ( ). Click the display near a value of 2 microseconds on the *time*-axis. Drag to the right, and release near a value of 4 microseconds on the *time*-axis. Signal Browser updates the rising edge **Settling Time** value to reflect the new time window, as shown in the following figure.
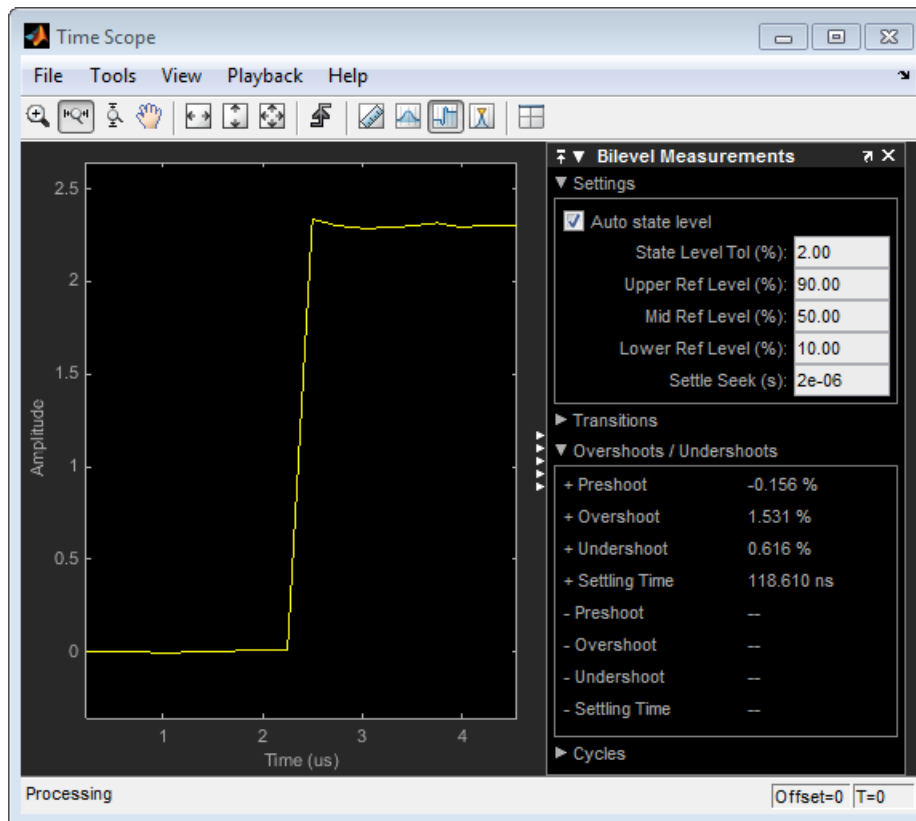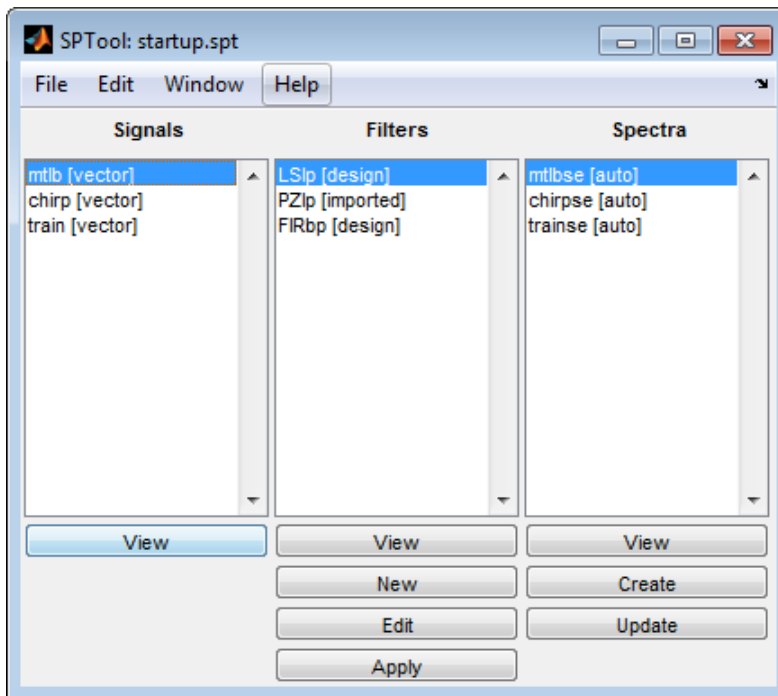
Find Peak Amplitudes in Signal Browser

This example shows how to use the Peak Finder panel in the SPTool Signal Browser to find heart rate, given an electrocardiogram (ECG) signal.

First, open SPTool by typing the following at the MATLAB command line.

```
sptool
```

SPTool opens.

Using SPTool, you can import signals from variables in the MATLAB workspace. First, create an electrocardiogram (ECG) signal, sampled at 4 kHz, and apply the Savitzky-Golay filter. At the MATLAB command line, enter the following commands:

```
x1 = 3.5*ecg(2700).';
y1 = sgolayfilt(kron(ones(1,13),x1),0,21);
n = (1:30000)';
```

```
del = round(2700*rand(1));
mhb = y1(n + del);
ts = 0.00025;
Fs = 1/ts;
```

For more information about the Savitzky-Golay filter, see the sgolayfilt function reference page or run the `sgolaydemo` example.

To import the signal from these variables, in the SPTool menu, select **File** > **Import**. Alternatively, you can press the **Ctrl+I** keyboard shortcut. The Import to SPTool dialog box appears.

1   Under **Workspace Contents**, click mhb. Click the right arrow (→) button to the left of the **Data** box.

2   Under **Workspace Contents**, click Fs. Click the right arrow (→) button to the left of the **Sampling Frequency** box.

3   In the **Name** box, type ECG.

4   Click **OK**. The SPTool **Signals** list now contains a signal named ECG [vector].

5   In SPTool, in the **Signals** list, select ECG [vector], and click the **View** button. The Signal Browser opens and displays the ECG signal.

Because you only selected one signal, the legend is not needed. Turn off the legend by clicking the Show All Legends (⊞) button. Then, use the Style dialog box to modify the appearance of the axes and the lines for the signal. In the Signal Browser menu, select **View** > **Style**.

Parameter	Display 1 Setting
Axes background color	Black
Ticks, labels, and grid colors	Dark Gray
Line color	Yellow

To show the **Peak Finder** panel, in the Signal Browser menu, select **Tools** > **Measurements** > **Peak Finder**. To expand the **Settings** pane, click the pane expand button (▶) next to that label. In the **Max Num of Peaks** box, type 10 and press the **Enter** key. Signal Browser now displays in the **Peaks** pane a list of 10 peak amplitude values, and the times at which they occur, as shown in the following figure.
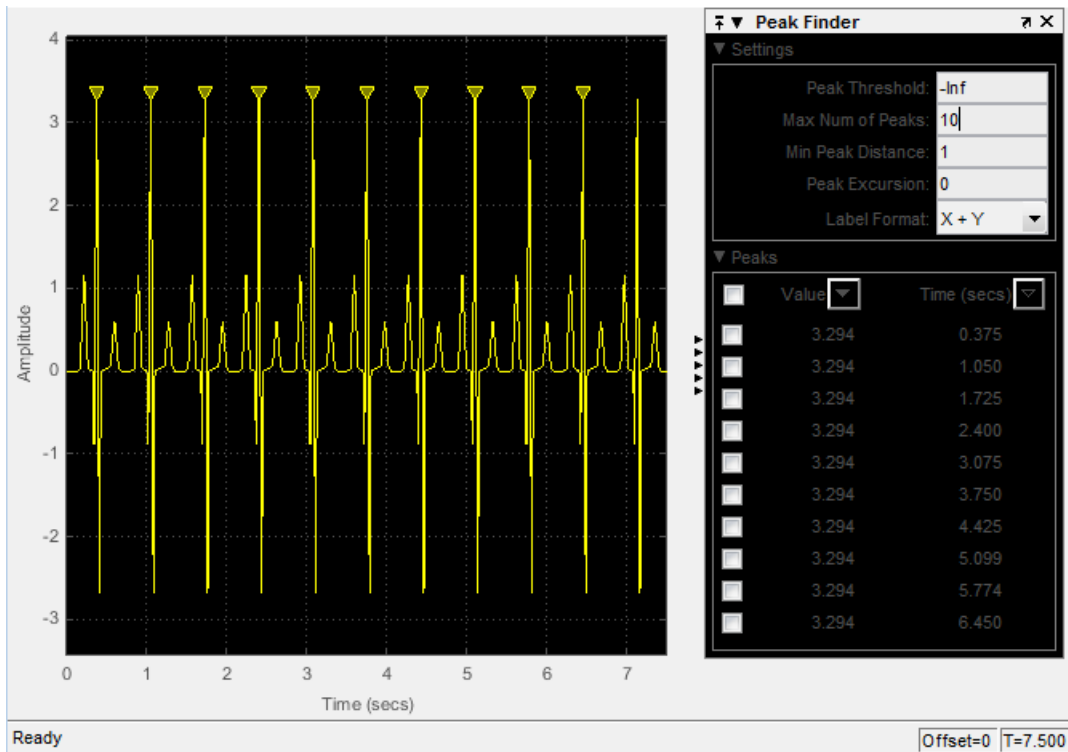
As you can see from the list of peak values, there is a constant time difference of 0.675 seconds between each heartbeat. Therefore, the heart rate detected by the ECG signal is given by the following equation.

$$\frac{60\,\dfrac{\text{sec}}{\text{min}}}{0.675\,\dfrac{\text{sec}}{\text{beat}}} = 88.89\,\frac{\text{beats}}{\text{min}}\,(\text{bpm})$$

16-21

Analyzing Harmonic Distortion

This example shows how to analyze the harmonic distortion of a weakly non-linear system in the presence of noise.

Introduction

In this example we will explore the output of a simplified model of an amplifier that has noise coupled to the input signal and exhibits non-linearity. We will explore how attenuation at the input can reduce harmonic distortion. We will also give an example of how to mathematically correct for the distortion at the output of the amplifier.

Viewing the Effects of Non-Linearity

A convenient way to view the effect of the non-linearity of the amplifier is to view the periodogram of its output when stimulated with a sinusoid. The amplitude of the sinusoid is set to the maximum allowable voltage of the amplifier. (2 Vpk)

In this example we will source a 2 kHz sinusoid for a duration of 50ms.

```
VmaxPk = 2;         % Maximum operating voltage
Fi = 2000;          % Sinusoidal frequency of 2 kHz
Fs = 44.1e3;        % Sample rate of 44.1kHz
Tstop = 50e-3;      % Duration of sinusoid
t = 0:1/Fs:Tstop;   % Input time vector

% Use the maximum allowable voltage of the amplifier
inputVmax = VmaxPk*sin(2*pi*Fi*t);
outputVmax = helperHarmonicDistortionAmplifier(inputVmax);
```

View a zoomed-in region of the output sinusoid. Note that the imperfections of our amplifier are difficult to see visually when plotted with respect to time.
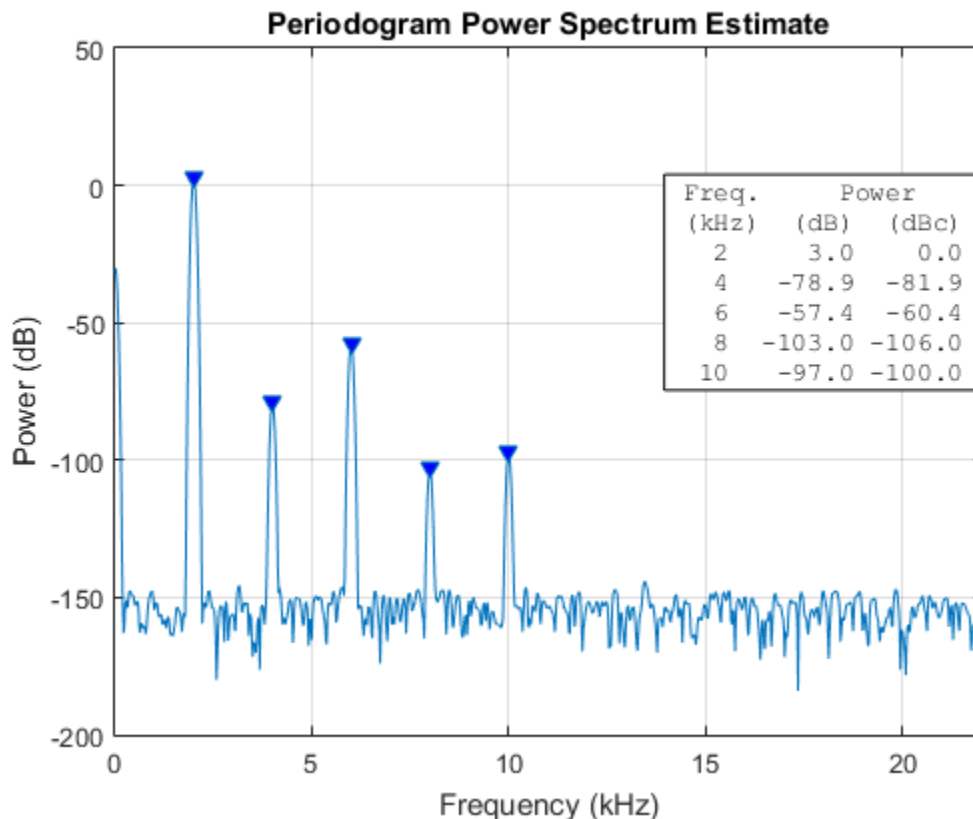
```
plot(t, outputVmax);
xlabel('Time')
ylabel('Output Voltage');
axis([0 5e-3 -2.5 2.5]);
title('Amplifier output');
```
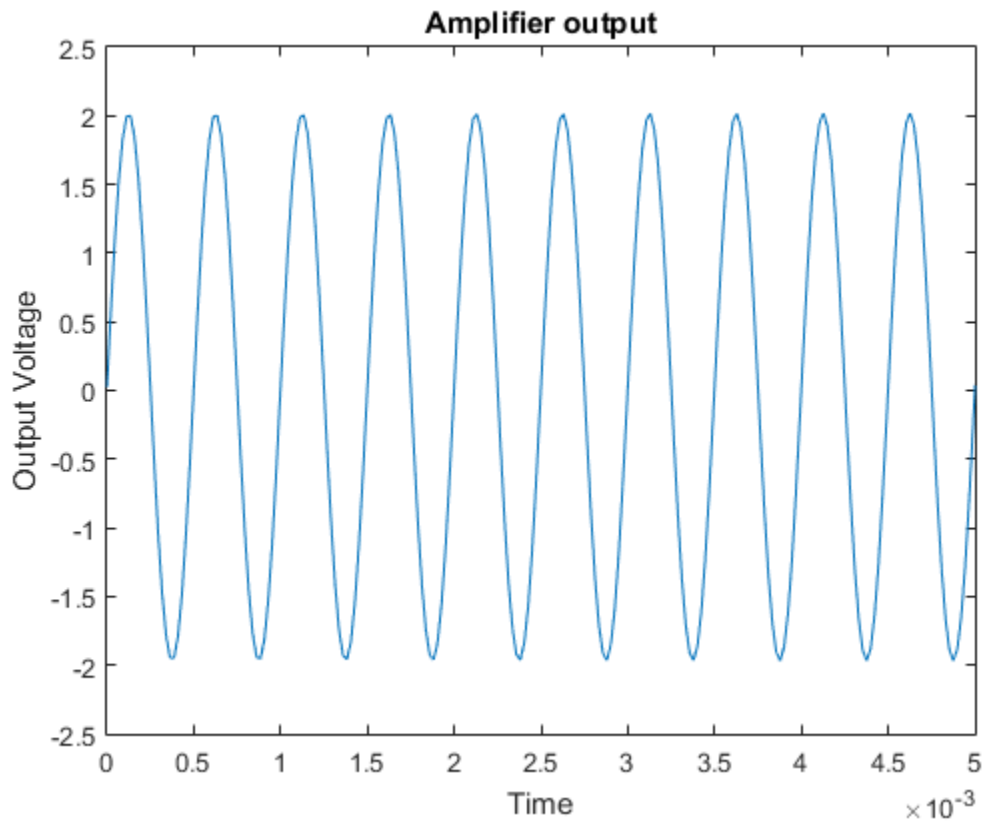
Now let's view the periodogram of our amplifier output.

```
helperPlotPeriodogram(outputVmax, Fs, 'power','annotate');
```

Note that instead of seeing just the 2 kHz sinusoid that we placed at the input, we see other sinusoids at 4 kHz, 6 kHz, 8 kHz, and 10 kHz. These sinusoids are multiples of the fundamental 2 kHz frequency and are due to the non-linearity of the amplifier.

We also see a relatively flat band of noise power.

Quantifying Non-Linear Distortion

Let's examine some common distortion metrics for comparison purposes

Our periodogram shows some very well defined harmonics of the fundamental signal. This suggests we measure the total harmonic distortion of the input signal which returns the ratio of power of all harmonic content to the fundamental signal.

```
thd(outputVmax, Fs)
```
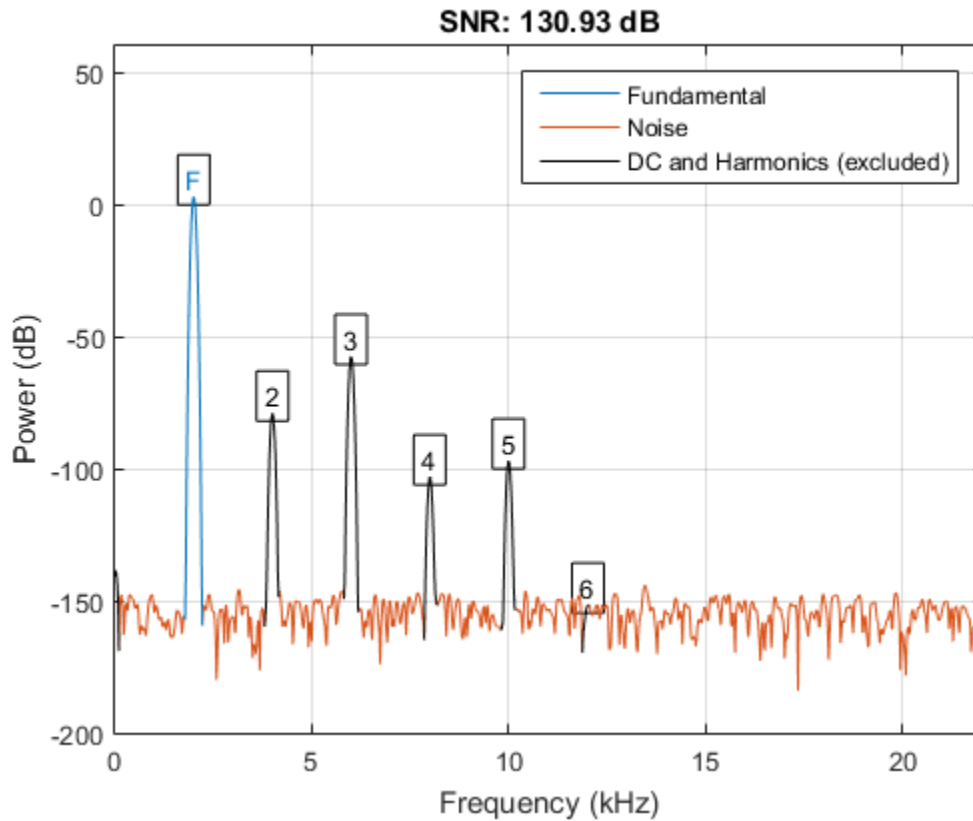
```
ans =

    -60.3888
```
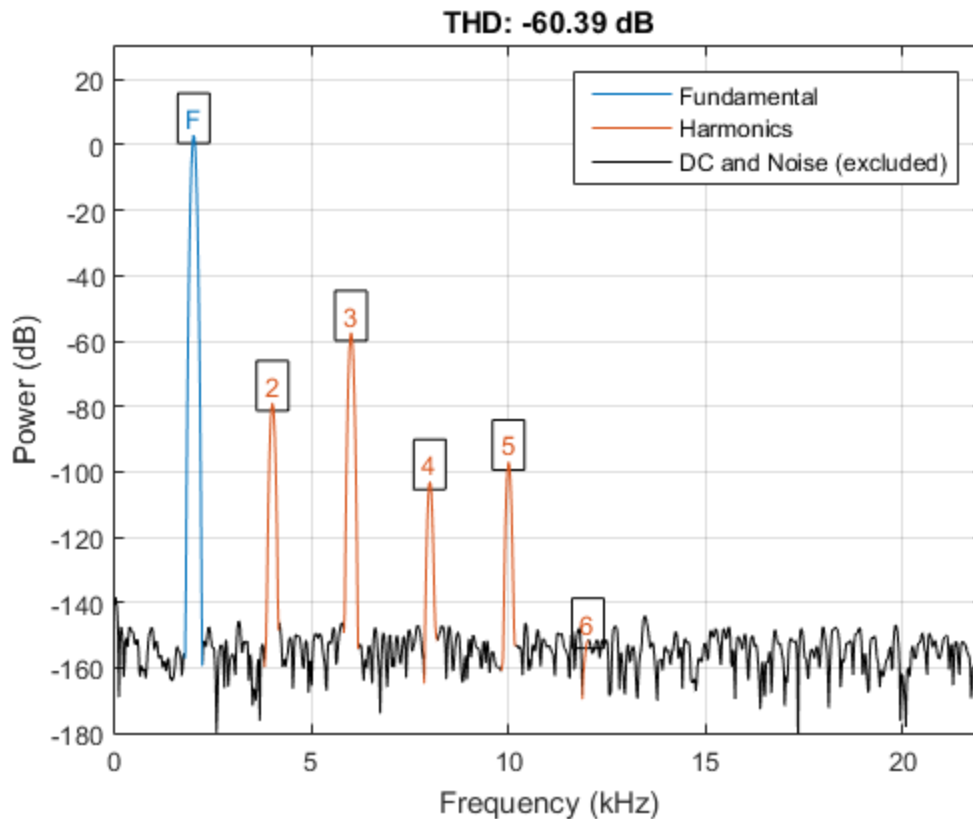

Notice that the third and largest harmonic is about 60 dB down from the fundamental. This is where most of the distortion is occurring.

We can also obtain an estimate of the total noise present in our input. To do this, we call SNR which returns the ratio of the power of the fundamental to the power of all non-harmonic content.

```
snr(outputVmax, Fs)
```
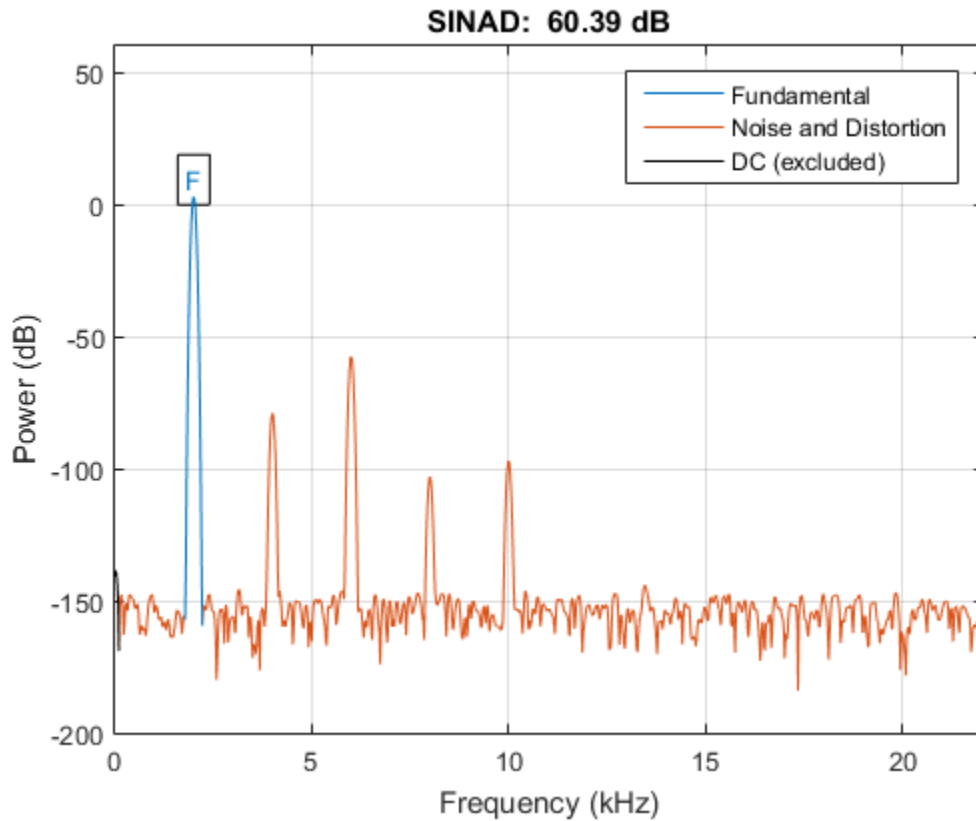
```
ans =

   130.9300
```


Another useful metric to compute is SINAD. This computes the ratio of the power to all other harmonic and noise content in the signal.

```
sinad(outputVmax, Fs)
```

```
ans =

    60.3888
```


The THD, SNR, and SINAD were -60 dB, 131 dB and 60 dB, respectively. Since the magnitude of THD is roughly equal to SINAD, we can attribute that most of the distortion is due to harmonic distortion.

If we inspect the periodogram, we can notice that the third harmonic dominates the distortion of the output.

Input Attenuation to Reduce Harmonic Distortion

Most analog circuitry performing amplification has an inherent trade-off between harmonic distortion and noise power. In our example, our amplifier has relatively low noise power compared to the harmonic distortion. This makes it suitable for detecting low power signals. If our input can be attenuated to enter this low-power region, we can recover some of the harmonic distortion.

Let's repeat the measurements by lowering the input voltage by a factor of two.

```
inputVhalf = (VmaxPk/2) * sin(2*pi*Fi*t);
outputVhalf = helperHarmonicDistortionAmplifier(inputVhalf);
helperPlotPeriodogram(outputVhalf, Fs, 'power','annotate');
```
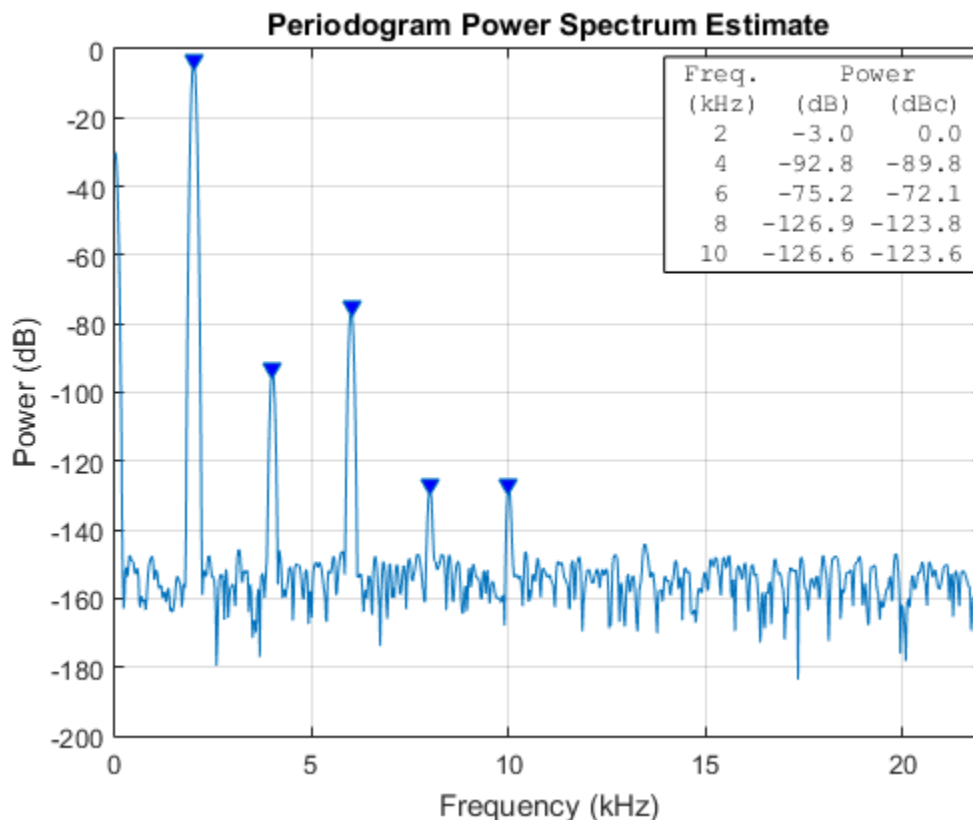
Let's redo our metrics again, this time measuring the effect of lowering the input voltage.

```
thdVhalf = thd(outputVhalf, Fs)
snrVhalf = snr(outputVhalf, Fs)
sinadVhalf = sinad(outputVhalf, Fs)


thdVhalf =

  -72.0676


snrVhalf =

  124.8767


sinadVhalf =

   72.0676
```

Notice that simply attenuating the input power level by 6 dB reduces the harmonic content. The SINAD and THD improved from ~60 dB to ~72 dB. This came at the expense of lowering the SNR from 131 dB to 125 dB.

SNR THD and SINAD as a Function of Input Attenuation

Can further attenuation improve our overall distortion performance? Let's plot THD, SNR and SINAD as a function of input attenuation, sweeping the input attenuator from 1 to 30 dB.

```
% Allocate a table with 30 entries
nReadings = 30;
distortionTable = zeros(nReadings, 3);

% Compute the THD, SNR and SINAD for each of the attenuation settings
for i = 1:nReadings
  inputVbestAtten = db2mag(-i) * VmaxPk * sin(2*pi*Fi*t);
  outputVbestAtten = helperHarmonicDistortionAmplifier(inputVbestAtten);
  distortionTable(i,:) = [abs(thd(outputVbestAtten, Fs))
                          snr(outputVbestAtten, Fs)
                          sinad(outputVbestAtten, Fs)];
end
```

```
% Plot results
plot(distortionTable);
xlabel('Input attenuation (dB)'), ylabel('Dynamic Range (dB)');
legend('|THD|','SNR','SINAD','Location','Best');
title('Distortion Metrics vs. Input Attenuation');
```
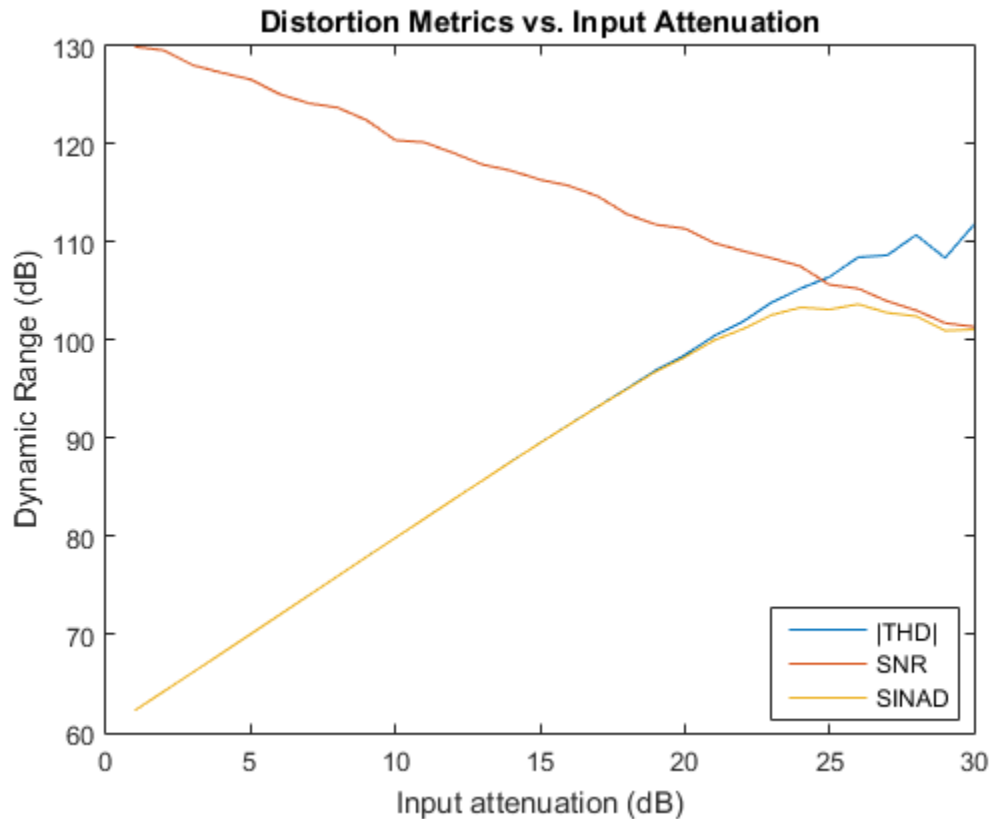

The graph shows the usable dynamic range corresponding to each metric. The *magnitude* of THD corresponds to the range that is free of harmonics. Similarly, SNR corresponds to the dynamic range of that is unaffected by noise; SINAD corresponds to the total dynamic range that is free of distortion.

As can be seen from the graph, SNR degrades as input power attenuation increases. This is because when you attenuate the signal, only the signal is attenuated, but the noise floor of the amplifier stays the same.

Also note that the magnitude of the total harmonic distortion improves steadily until it intersects the SNR curve, after which the measurement becomes unstable. This happens when the harmonics have "disappeared" beneath the noise of the amplifier.
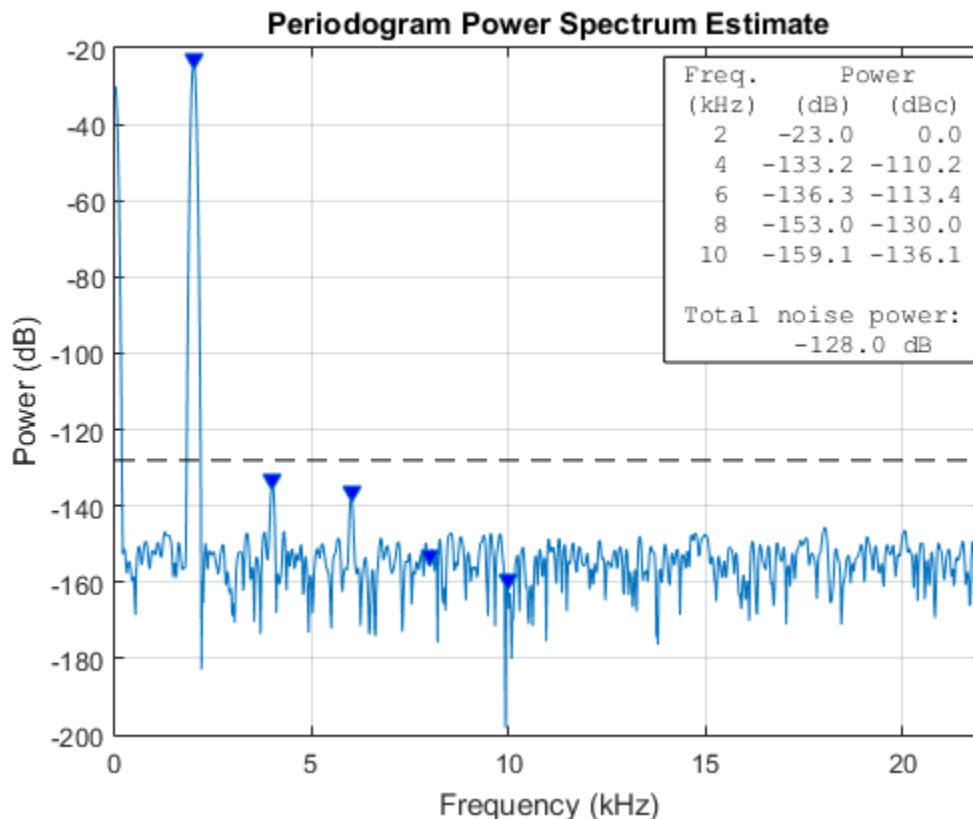
A practical choice of amplifier attenuation for the amplifier would be 26 dB (yielding a SINAD of 103 dB). This would be a reasonable tradeoff between harmonic and noise distortion.

```matlab
% Search the table for the largest SINAD reading
[maxSINAD, iAtten] = max(distortionTable(:,3));
fprintf('Max SINAD (%.1f dB) occurs at %.f dB attenuation\n', ...
  maxSINAD, iAtten);
```

```
Max SINAD (103.7 dB) occurs at 26 dB attenuation
```

Let's plot the periodogram when the attenuator is set to 26 dB.

```matlab
inputVbestAtten = db2mag(-iAtten) * VmaxPk * sin(2*pi*Fi*t);
outputVbestAtten = helperHarmonicDistortionAmplifier(inputVbestAtten);
helperPlotPeriodogram(outputVbestAtten, Fs, 'power','annotate','shownoise');
```

Periodogram Power Spectrum Estimate

Freq.	Power	
(kHz)	(dB)	(dBc)
2	-23.0	0.0
4	-133.2	-110.2
6	-136.3	-113.4
8	-153.0	-130.0
10	-159.1	-136.1
Total noise power:		
	-128.0 dB	

Here we have additionally plotted the level of *total* noise power that is spread across the spectrum. Note that at this attenuation setting, the second and third harmonic are still visible in the spectrum but also considerably less than the total noise power. If we were to have an application that uses a smaller bandwidth of the available spectrum we would benefit from further increasing the attenuation to reduce the harmonic content.

Post-processing to Remove Distortion

Occasionally we can correct for some of the non-linearity of the amplifier. If the output of the amplifier is digitized, we can recover more useful dynamic range by digitally post-processing the captured output and correcting for the non-linearity mathematically.

In our case, we stimulate the input with a linear ramp and fit a third-order polynomial that best fits the input.

```
inputRamp = -2:0.00001:2;
outputRamp = helperHarmonicDistortionAmplifier(inputRamp);
polyCoeff = polyfit(outputRamp, inputRamp, 3)


polyCoeff =

    0.0010   -0.0002    1.0000   -0.0250
```

Now that we have the coefficients we can then perform post-correction at the output and compare side-by-side with our original uncorrected output

```
correctedOutputVmax = polyval(polyCoeff, outputVmax);

helperPlotPeriodogram([outputVmax; correctedOutputVmax], Fs, 'power');
subplot(2,1,1);
title('Uncorrected');
subplot(2,1,2);
title('Polynomial Corrected');
```
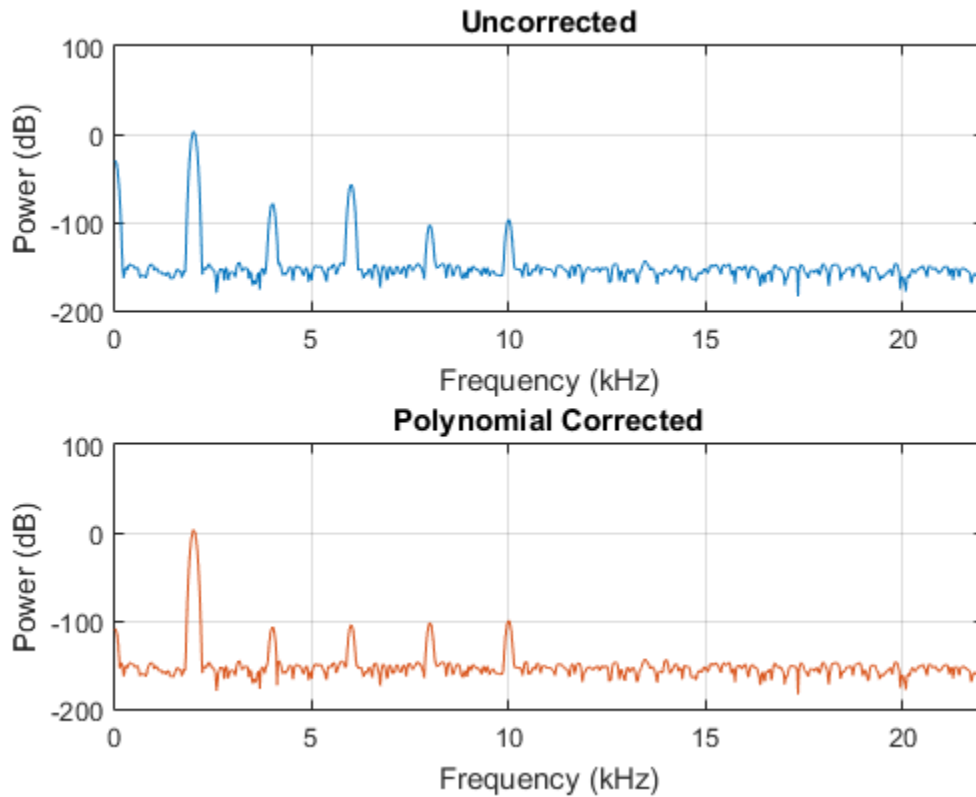
Note that the second and third harmonics are significantly reduced when using polynomial correction.

Let's repeat the measurements again with the corrected output.

```
thdCorrectedVmax = thd(correctedOutputVmax, Fs)
snrCorrectedVmax = snr(correctedOutputVmax, Fs)
sinadCorrectedVmax = sinad(correctedOutputVmax, Fs)
```

```
thdCorrectedVmax =

  -99.6194
```

```
snrCorrectedVmax =

   130.7491


sinadCorrectedVmax =

    99.6162
```

Notice that our SINAD (and THD) dropped from 60 dB down to 99 dB, while preserving our original SNR of 131 dB.

Combining Techniques

We can combine attenuation with polynomial evaluation to find the ideal operating voltage that minimizes the overall SINAD of our system.

```
subplot(1,1,1);
% Add three more columns to our distortion table
distortionTable = [distortionTable zeros(nReadings,3)];
for i = 1:nReadings
  inputVreduced = db2mag(-i) * VmaxPk * sin(2*pi*Fi*t);
  outputVreduced = helperHarmonicDistortionAmplifier(inputVreduced);
  correctedOutput = polyval(polyCoeff, outputVreduced);
  distortionTable(i,4:6) = [abs(thd(correctedOutput, Fs))
                            snr(correctedOutput, Fs)
                            sinad(correctedOutput, Fs)];
end

h = plot(distortionTable);
xlabel('Input attenuation (dB)');
ylabel('Dynamic Range (dB)');
for i=1:3
  h(i+3).Color = h(i).Color;
  h(i+3).LineStyle = '--' ;
end
legend('|THD| (uncorrected)','SNR (uncorrected)','SINAD (uncorrected)', ...
 '|THD| (corrected)','SNR (corrected)','SINAD (corrected)','Location','Best');
title('Distortion Metrics vs. Input Attenuation and Polynomial Correction');
```
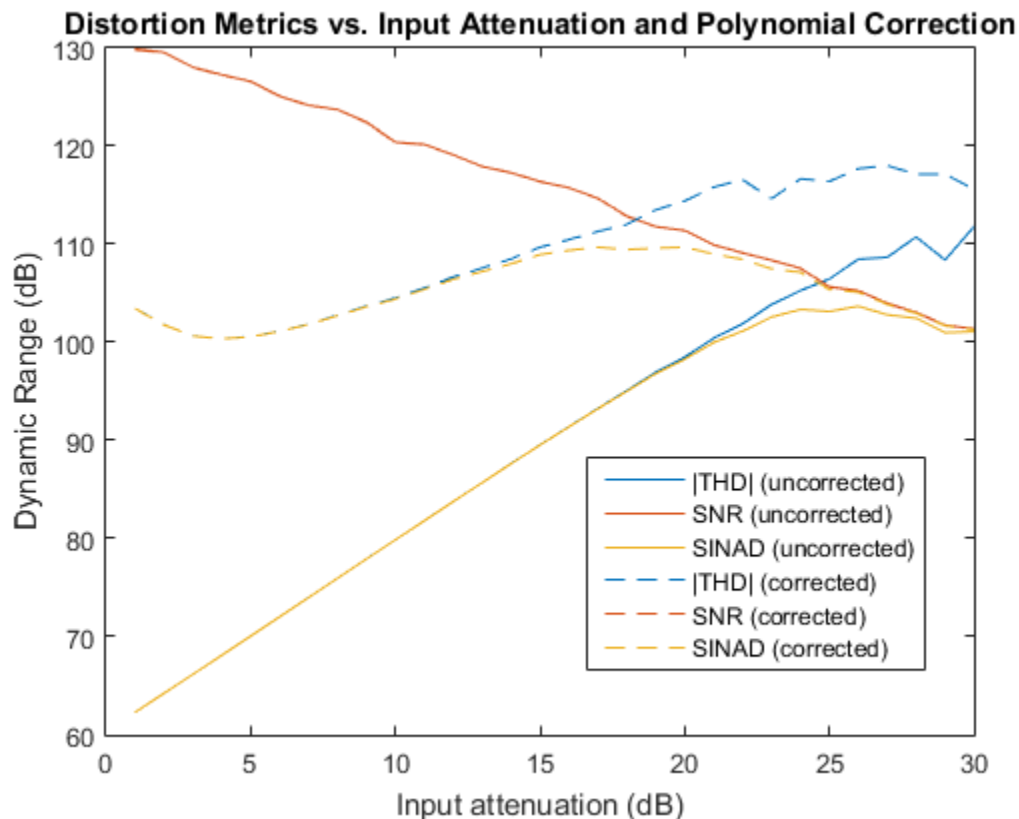
Here, we've plotted all three metrics alongside for both the uncorrected and polynomial corrected amplifier.

As can be seen from the graph, THD has improved considerably, whereas SNR was not affected by polynomial correction. This is to be expected since the polynomial correction only affects the harmonic distortion and not the noise distortion.

Let's show the maximum SINAD possible when corrected by the polynomial

```
[maxSINADcorrected, iAttenCorr] = max(distortionTable(:,6));
fprintf('Corrected:    Max SINAD (%.1f dB) at %.f dB attenuation\n', ...
  maxSINADcorrected, iAttenCorr);
```

```
Corrected:    Max SINAD (109.7 dB) at 17 dB attenuation
```

A good choice of amplifier attenuation for the polynomial corrected amplifier would be 20dB (yielding a SINAD of 109.8 dB).

```
% Recompute amplifier at maximum SINAD attenuation setting with polynomial
inputVreduced = db2mag(-iAttenCorr) * VmaxPk * sin(2*pi*Fi*t);
outputVreduced = helperHarmonicDistortionAmplifier(inputVreduced);
correctedOutputVbestAtten = polyval(polyCoeff, outputVreduced);

helperPlotPeriodogram(correctedOutputVbestAtten, Fs, 'power','annotate','shownoise');
title('Periodogram of attenuated and polynomial corrected amplifier');
```
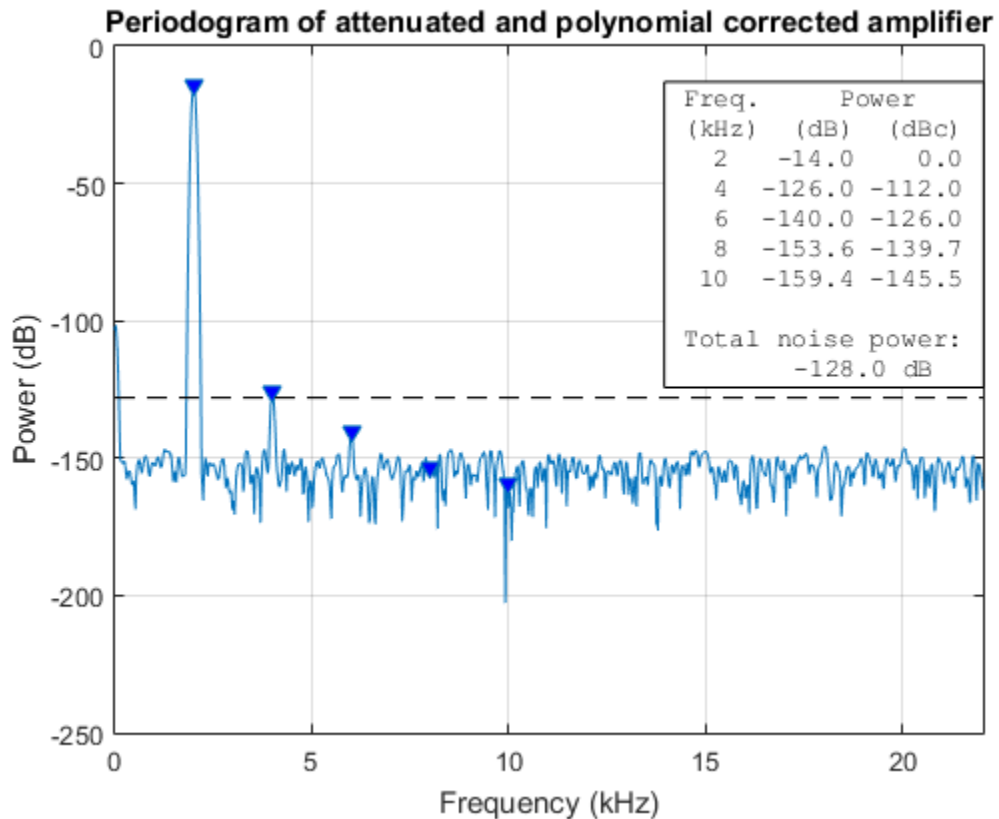

Note that all but the second harmonic disappeared entirely with polynomial correction under the ideal attenuation setting. As noted before the second harmonic appears just

underneath the power level of the total noisefloor. This provides a reasonable tradeoff in applications which use the full bandwidth of the amplifier.

Summary

We have shown how polynomial correction can be applied to the output of an amplifier experiencing distortion and how to pick a reasonable attenuation value to reduce the effects of harmonic distortion.

# Spurious-Free Dynamic Range (SFDR) Measurement

This example shows how to analyze a numerically controlled oscillator (NCO) of a digital down-converter (DDC) implemented in fixed-point arithmetic. The example measures the spurious free dynamic range (SFDR) of the NCO, and explore the effects of adding phase dither. The number of dither bits affects hardware implementation choices. The example shows trade-offs among noise floor level, spurious effects, and number of dither bits. The DDC in the example, designed to meet the GSM specification, models the Graychip 4016.

Introduction

Numerically controlled oscillators (NCOs) are an efficient means of generating sinusoidal signals, and are useful when you require a continuous-phase sinusoidal signal with variable frequency.

A DDC is a key component of digital radios. It translates the high-input sample rates of a digital radio down to lower sample rates (baseband) for further and easier processing. Our DDC has an input rate of 69.333 MHz and is tasked with converting the rate down to 270.833 KHz in accordance with GSM specifications.

The DDC consists of an NCO and a mixer to quadrature down convert the input signal to baseband. A Cascaded Integrator-Comb (CIC) then low-pass filters the baseband signal, and along with two FIR decimating filters downsample the signal to achieve the desired low sample rate, which is then ready for further processing. The final stage, depending on the application, often includes a resampler that interpolates or decimates the signal to achieve the desired sample rate. Further filtering can be achieved with the resampler. See the block diagram of a typical DDC, below. Note that Simulink® handles complex signals, so we don't have to treat the I and Q channels separately.
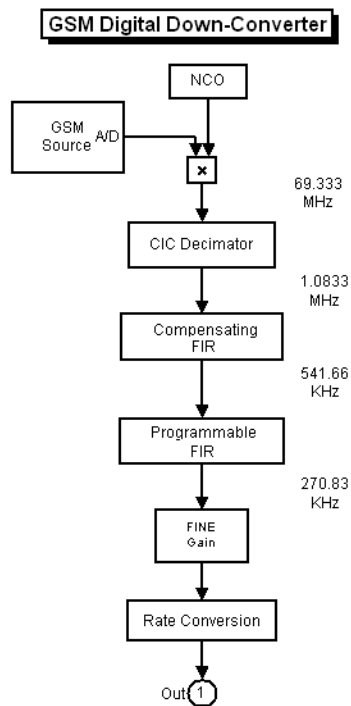
While this example focuses on the analysis of the NCO, an example titled "Implementing the Filter Chain of a Digital Down-Converter", focusing on designing the three-stage, multirate, fixed-point filter chain and HDL code generation is available in the DSP System Toolbox™.

The Numerically Controlled Oscillator

The digital mixer section of the DDC includes a multiplier and an NCO, which provide channel selection or tuning for the radio. The mixer is basically a sine-cosine generator, creating complex values for each sine-cosine pair. The typical NCO has four components: the phase accumulator, the phase adder, the dither generator, and sine-cosine lookup table.

Here is a typical NCO circuit modeled in Simulink, similar to what you might see in the Graychip data sheet.

```
open_system('ddcnco');
```

Numerically Controlled Oscillator (NCO)
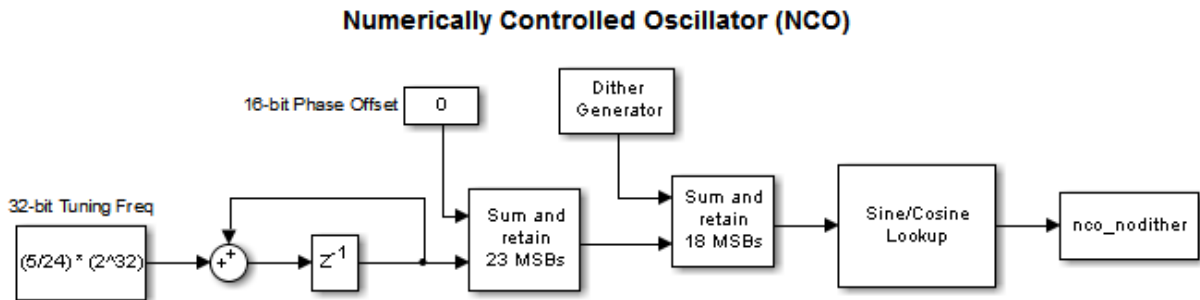
Based on the input frequency, the NCO's phase accumulator produces values that address a sine-cosine lookup table. The phase adder specifies a phase offset that modulates the output of the phase accumulator. The Dither Generator provides phase dithering to reduce amplitude quantization noise, and improving the SFDR of the NCO. The Sine/Cosine Lookup block produces the actual complex sinusoidal signal, and the output is stored in the variable nco_nodither.

In the Graychip, the tuning frequency is specified as a normalized value relative to the chip's clock rate. So for a tuning frequency of F, the normalized frequency is F/Fclk, where Fclk is the chip's clock rate. The phase offset is specified in radians, ranging from 0 to 2pi. In this example the normalized tuning frequency is set to 5/24 while the phase offset is set to 0. The tuning frequency is specified as a 32-bit word and the phase offset is specified as a 16-bit word.

Since the NCO is implemented using fixed-point arithmetic, it experiences undesirable amplitude quantization effects. These numerical distortions are due to the effects of finite word length. Basically, sinusoids are quantized creating cumulative, deterministic, and periodic errors in the time domain. These errors, appear as line spectra or spurs in the frequency domain. The amount of attenuation from the peak of the signal of interest to the highest spur is the SFDR.

The SFDR of the Graychip is 106 dB, but the GSM specification requires that the SFDR be greater than 110 dB. There are several ways to improve the SFDR, and you will explore adding phase dither to the NCO.

The Graychip's NCO contains a phase dither generator which is basically a random integer generator used to improve the oscillator's output purity. Dithering causes the

unintended periodicities of the quantization noise (which causes "spikes" in spectra and thus poor SFDR) to be spread across a broad spectrum, effectively reducing these undesired spectral peaks. Conservation of energy applies, however, so the spreading effectively raises the overall noise floor. That is, the dithering is good for SFDR, but only up to a point.

Let's run the NCO model and analyze its output in the MATLAB workspace. This model does not use dithering.

```
sim('ddcnco');
whos nco*
```

```
  Name              Size                 Bytes  Class     Attributes

  nco_nodither      1x1x20545           328720  double    complex
```

The plot below displays the real part of the first 128 samples of the NCO's output, stored in the variable, nco_nodither.

```
plot(real(squeeze(nco_nodither(1:128)))); grid
title('Real Part of NCO Output - No Dithering')
ylabel('Amplitude');
xlabel('Samples');
```
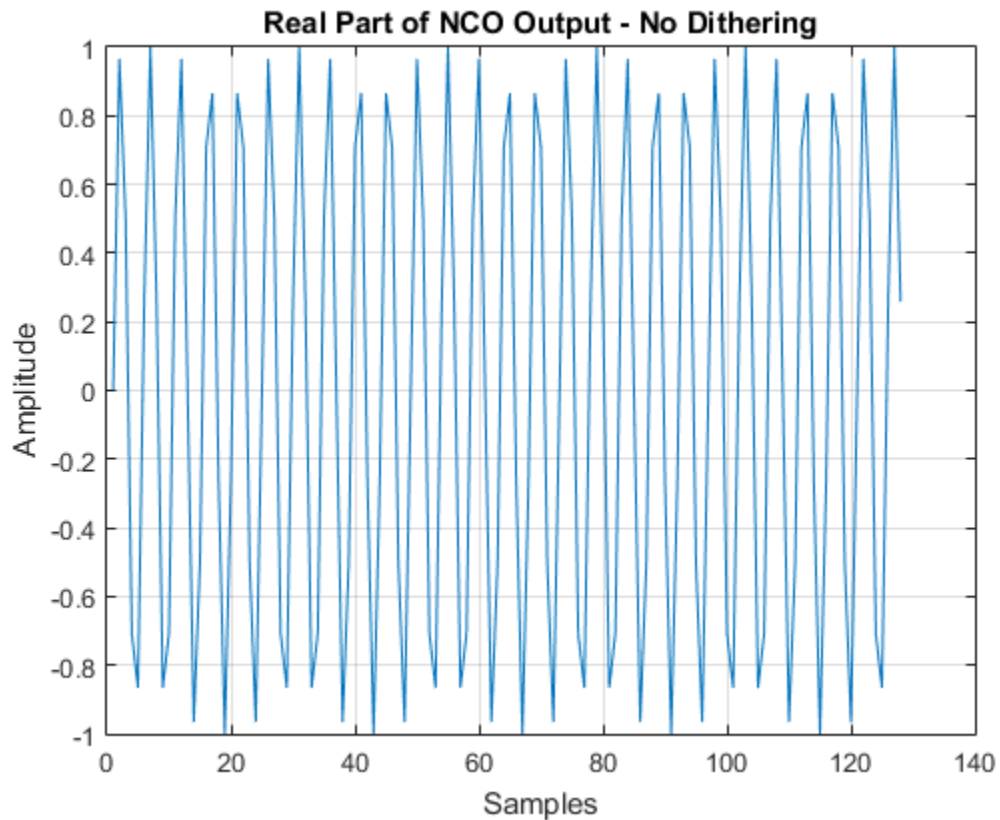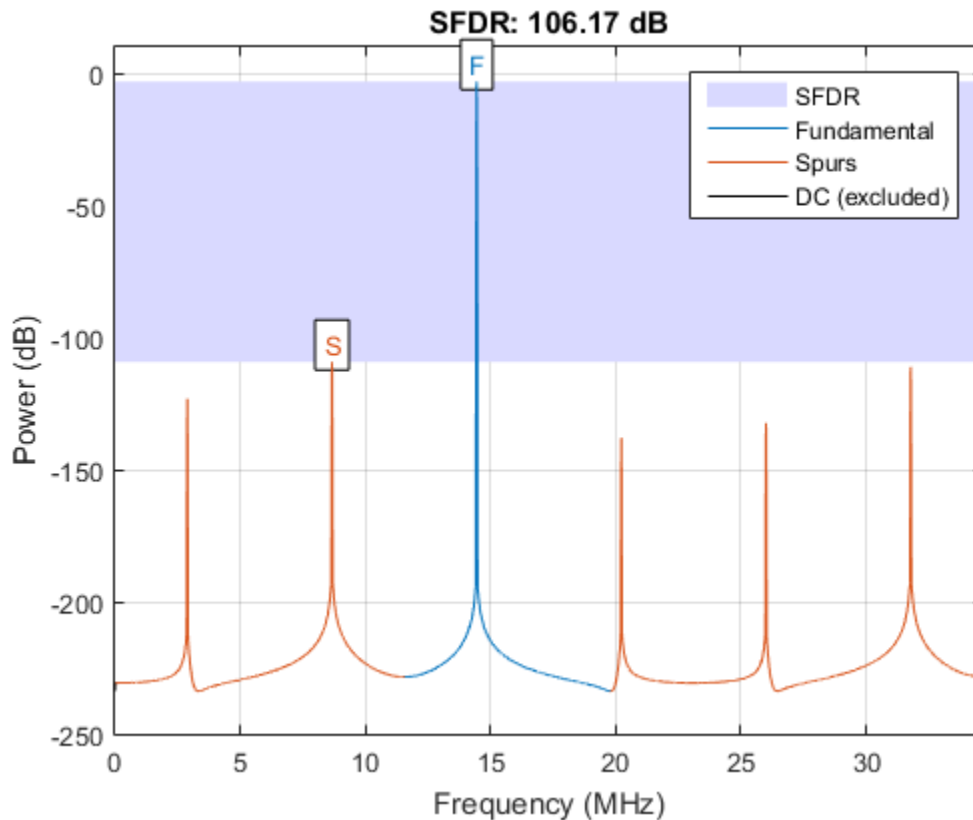
SFDR of NCO Output

Let's take a look at the SFDR of the output of the NCO.

Calculate and plot the SFDR of the NCO output

```
Fs = 69.333e6;
sfdr(real(nco_nodither),Fs);
```

As expected, the power spectrum plot shows a peak at 14.44 MHz, which is the NCO's tuning frequency, 5/24*Fs = 14.444 MHz.

The SFDR, however, is about 106.17 dB, which is too high to meet the GSM specification, which requires 110 dB or more. We can improve this dynamic range by use of phase dithering.

Exploring the Effects of Dithering

To explore adding dither to the NCO, the NCO circuit shown above has been encapsulated in a subsystem and duplicated three times. A different amount of dither was selected for each NCO. Although the NCO allows a range of 1 to 19 bits of dither to

be specified, you will try just few values. Running this model will produce three different NCO outputs based on the amount of dither added.

```
open_system('ddcncowithdither')
```
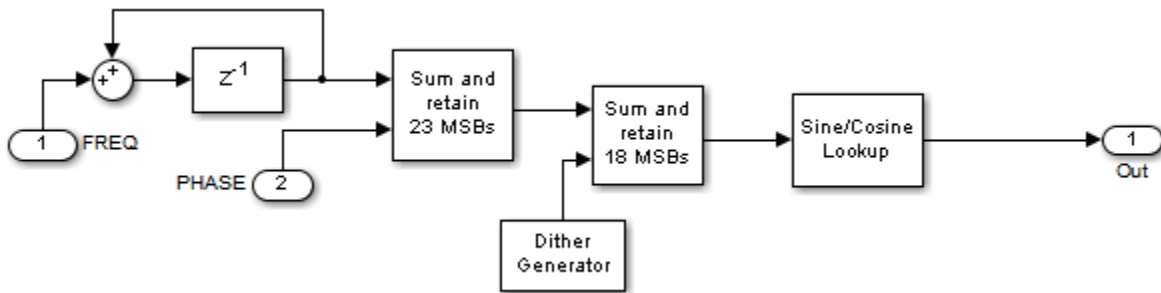

Running the simulation will produce three signals in the MATLAB workspace that you can then analyze using the spectral analysis algorithm defined above. You can run the simulation from the model or from the command line using the sim command.

```
sim('ddcncowithdither')
```

After the simulation completes you are left with the signals that are the NCOs' output. Each signal shows a different amount of dithering.

```
whos nco*
```

```
  Name                Size                  Bytes  Class      Attributes

  nco_dither3         1x1x20501            328016  double     complex
  nco_dither5         1x1x20501            328016  double     complex
  nco_dither7         1x1x20501            328016  double     complex
  nco_nodither        1x1x20545            328720  double     complex
```
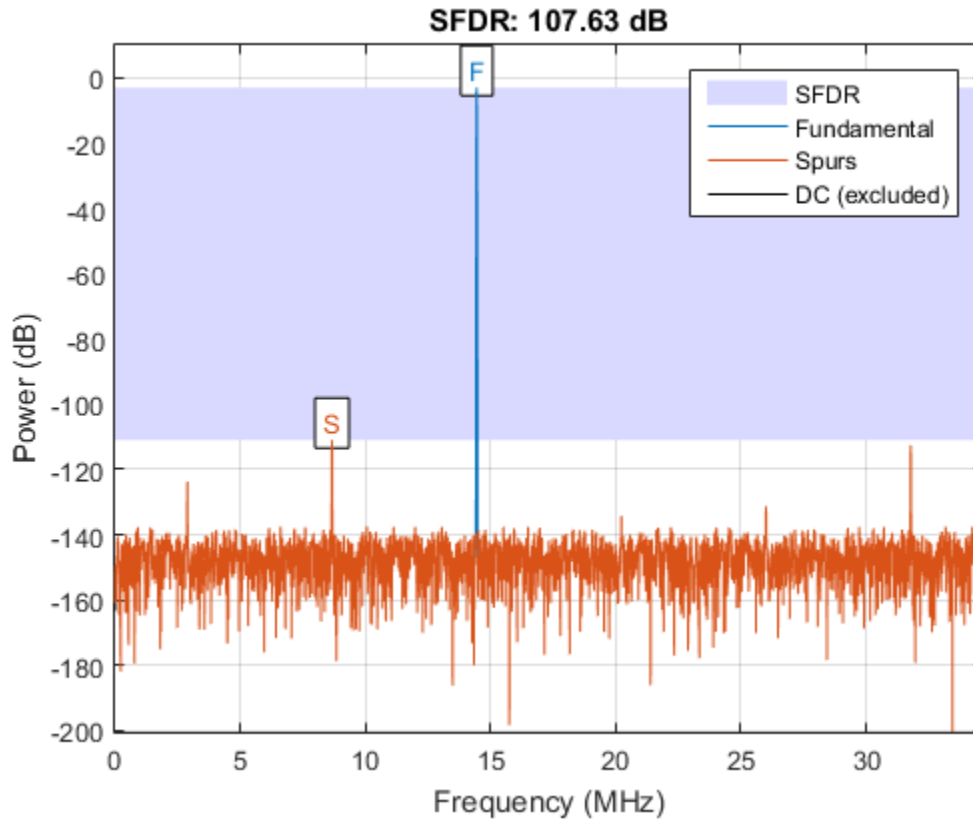
Compute and plot the SFDR after adding 3 bits of dithering.

```
figure,
sfdr(real(nco_dither3),Fs)


ans =
```

```
107.6285
```


With three bits of dither added, the highest spur is now about -112 dB. The SFDR is 107.63 dB. It still fails to meet the GSM specification.
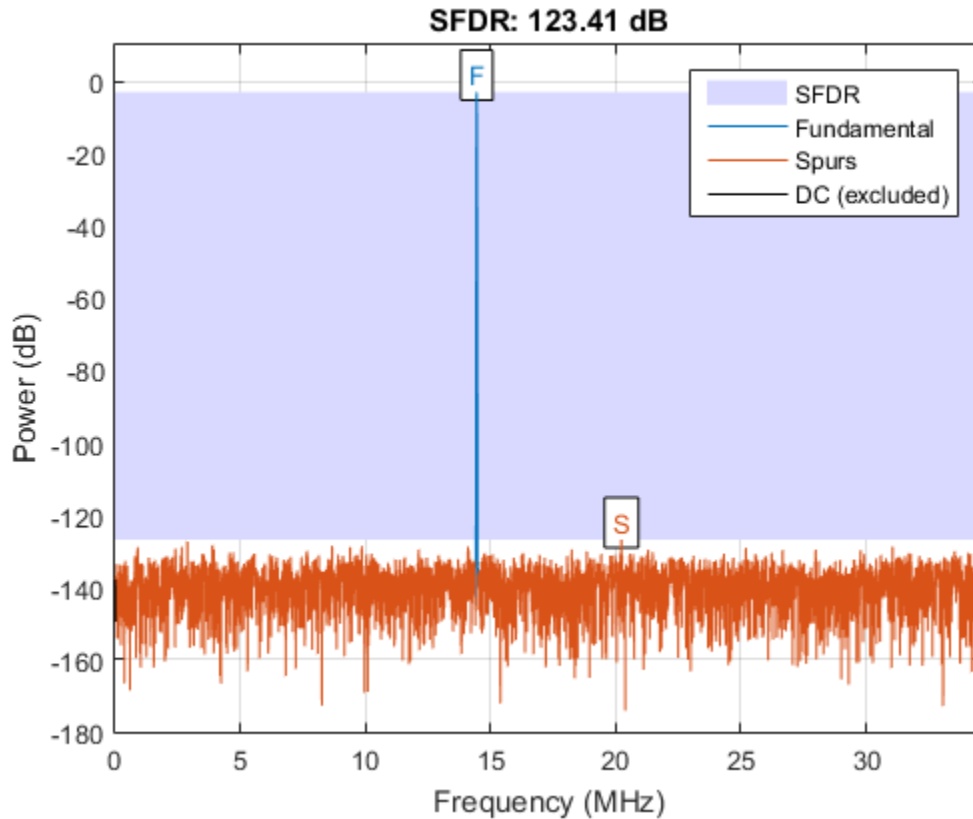
Now add 5 bits of dithering.

```
figure,
sfdr(real(nco_dither5),Fs)


ans =
```

```
123.4065
```


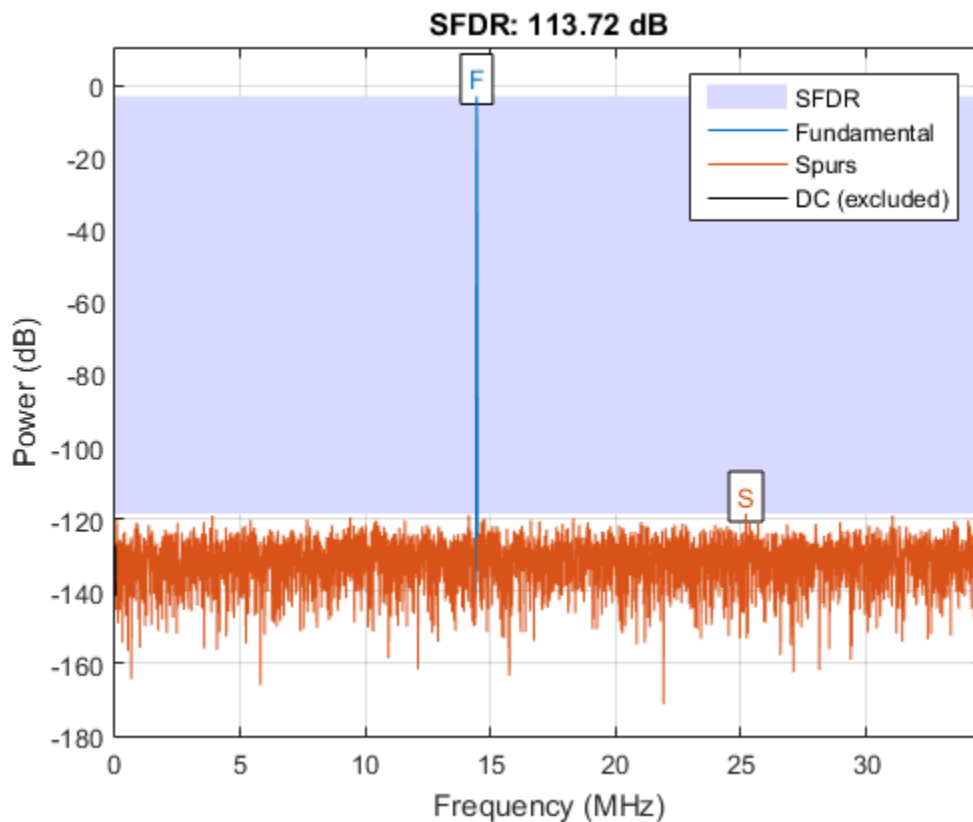With five bits of dither added, the highest spur is now about -127 dB.

The SFDR is 123.41 dB, exceeding the GSM specification.

It appears that more dither gives better results, so add 7 bits of dithering.

```
figure,
sfdr(real(nco_dither7),Fs)
```

```
ans =
```

113.7189

Note that our computed SFDR degraded to 113.72 dB. This is due to the broadband noise generated by dithering starting to overtake the power of the spurious content.

Using 7 bits of dithering meets the GSM specification, but is not as effective as using 5 bits of dithering.

Comparing Results

Tabulate the SFDR for each NCO output against the amounts of dithering for each NCO output.

```
Number of     Spur Free Dynamic
Dither bits   Range(dB)

   0             106.17

   3             107.63

   5             123.41

   7             113.72
```

Summary

This example analyzed the output of an NCO used in a digital down converter for a GSM application. Spectral analysis was used to measure the SFDR, the difference between the highest spur and the peak of the signal of interest. Spurs are deterministic, periodic errors that result from quantization effects. The example also explored the effects of adding dither in the NCO, which adds random data to the NCO to improve its purity. We found that using five bits of dithering achieved the highest SFDR.

Distortion Measurements

Generate 2048 samples of a sinusoid of frequency 2.5 kHz sampled at 50 kHz. Add white Gaussian noise such that the signal-to-noise ratio (SNR) is 80 dB.

```
Fs = 5e4;
f0 = 2.5e3;
N = 2048;
t = (0:N-1)/Fs;
SNR = 80;

x = cos(2*pi*f0*t);
x = x+randn(size(x))*std(x)/db2mag(SNR);
```

Pass the result through a weakly nonlinear amplifier represented by a polynomial. The amplifier introduces spurious tones at the frequencies of the harmonics.

```
amp = [1e-5 5e-6 -1e-3 6e-5 1 25e-3];
x = polyval(amp,x);
```
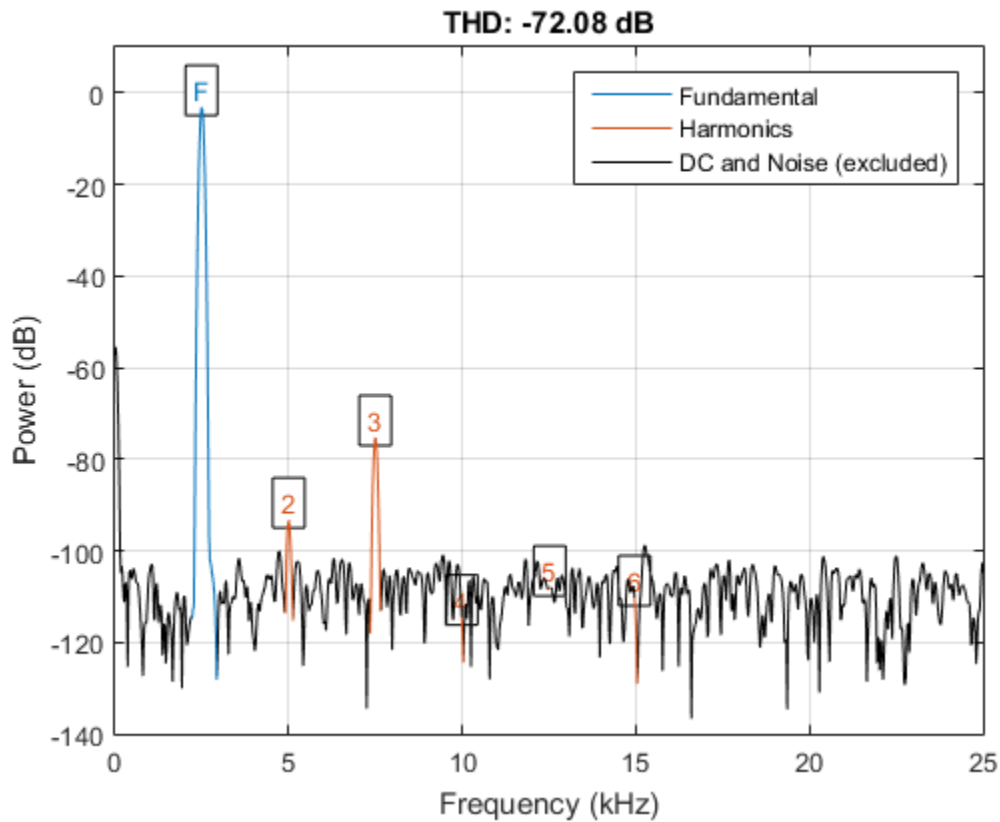
Plot the signal spectrum and annotate the SNR, verifying that it has the expected value. The snr function computes the power ratio of the fundamental to the noise floor and ignores the DC component and the harmonics.

```
snr(x,Fs);
```
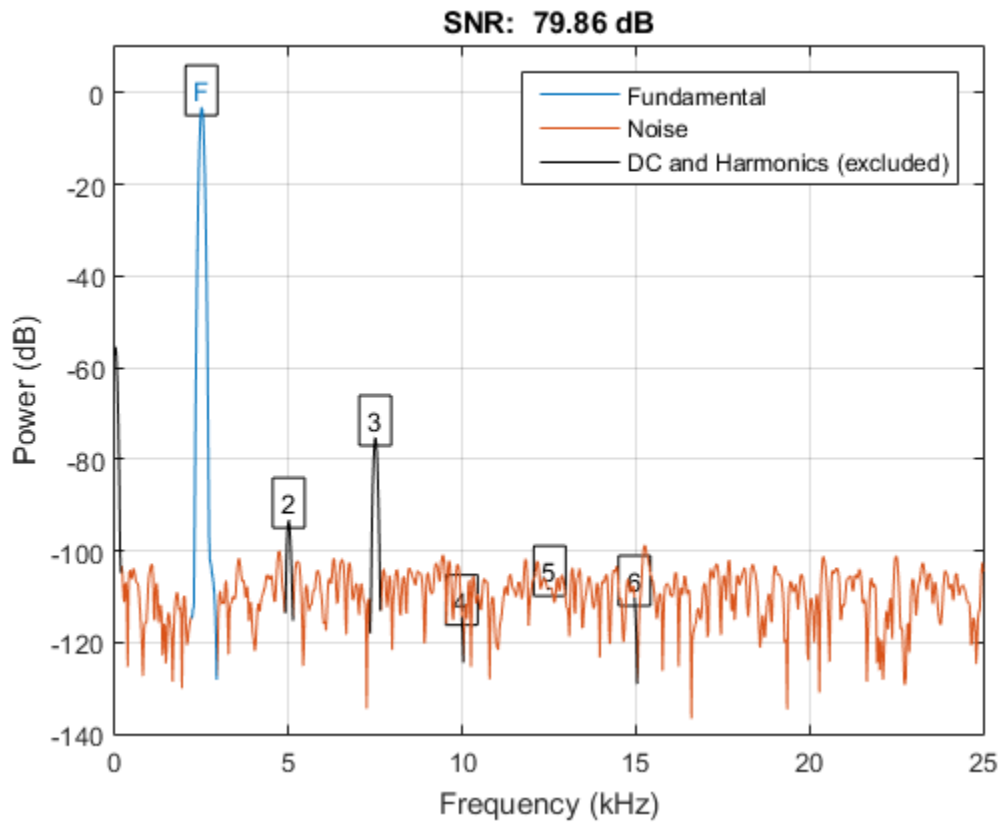
Plot the signal spectrum and annotate the total harmonic distortion (THD). The `thd` function computes the power ratio of the harmonics to the fundamental and ignores the DC component and the noise floor.

```
thd(x,Fs);
```

Plot the signal spectrum and annotate the signal to noise and distortion ratio (SINAD). The `sinad` function computes the power ratio of the fundamental to the harmonics and the noise floor. It ignores only the DC component.

```
sinad(x,Fs);
```
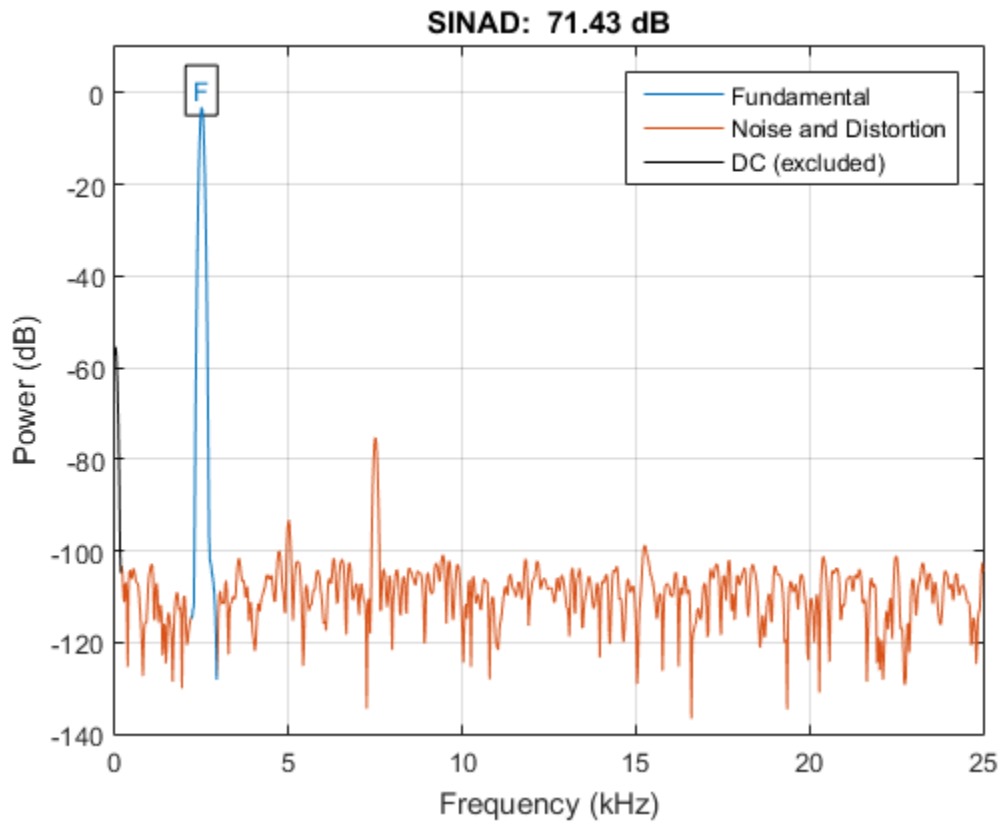
Verify that the SNR, THD, and SINAD obey the equation

$$10^{-\text{SNR}/10} + 10^{\text{THD}/10} = 10^{-\text{SINAD}/10}.$$

```
lhs = 10^(-snr(x,Fs)/10)+10^(thd(x,Fs)/10)
rhs = 10^(-sinad(x,Fs)/10)


lhs =

   7.2203e-08


rhs =
```
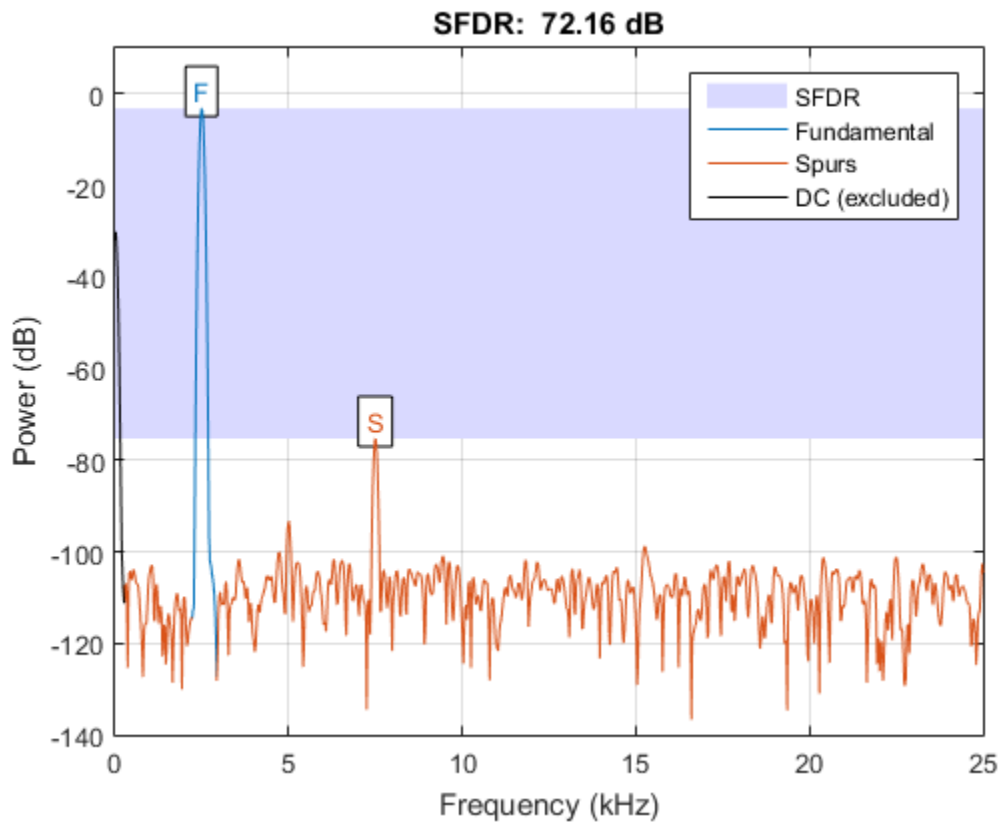
7.1997e-08

Plot the signal spectrum and annotate the spurious-free dynamic range (SFDR). The SFDR is the power ratio of the fundamental to the strongest spurious component ("spur"). In this case, the spur corresponds to the third harmonic.

```
sfdr(x,Fs);
```

Measurement of Pulse and Transition Characteristics

This example shows how to analyze pulses and transitions and compute metrics including rise time, fall time, slew rate, overshoot, undershoot, pulse width, and duty cycle.

Clock Signal with Noise

First let's view the samples from a noisy clock signal.

```
load clocksig clock1 time1 Fs
plot(time1, clock1);
xlabel('Time (secs)'), ylabel('Voltage');
```
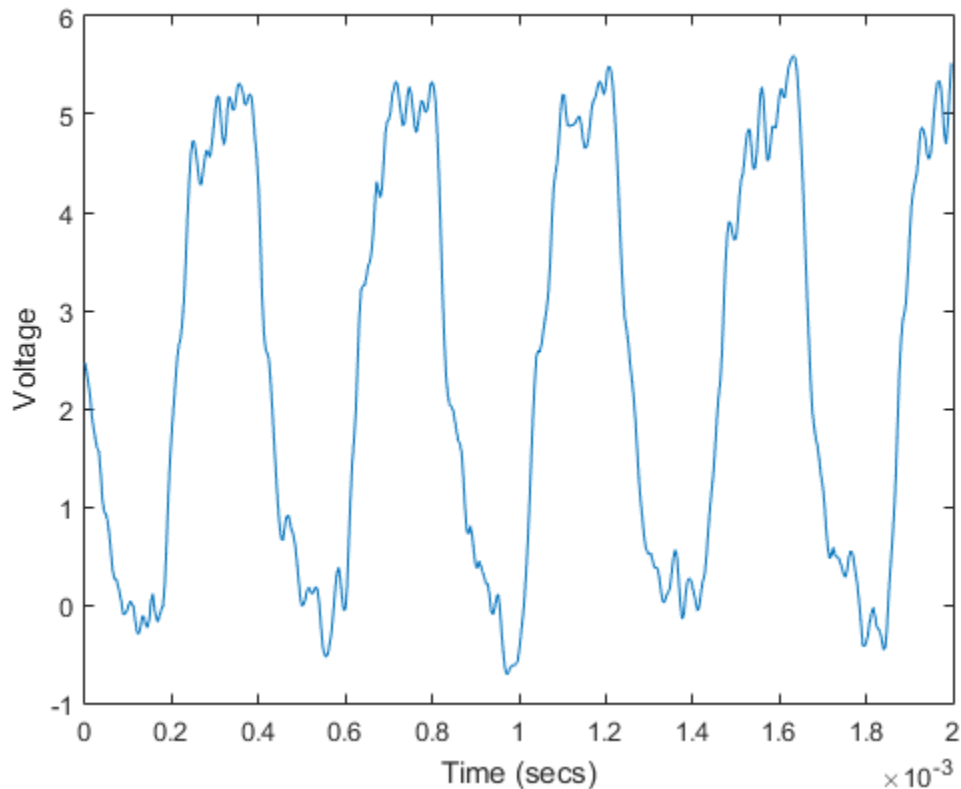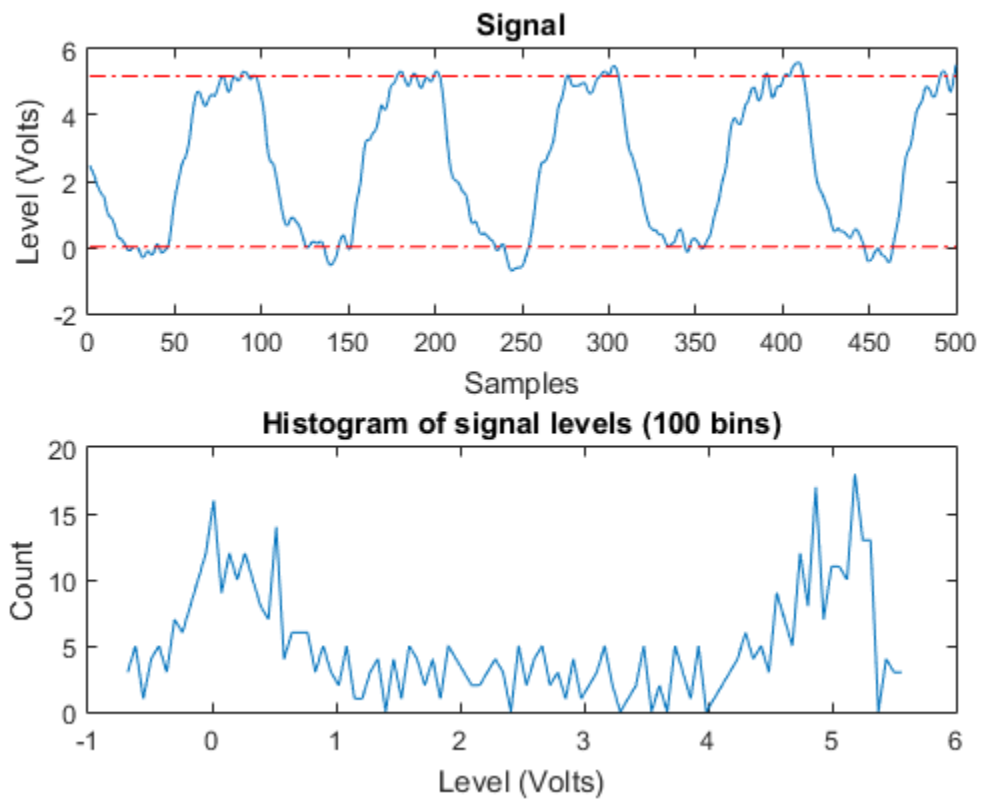
Estimate State Levels

Use `statelevels` with no output argument to visualize the state levels. State levels are estimated via histogram.

```
statelevels(clock1)
```

```
ans =

    0.0138    5.1848
```

The computed histogram is divided into two equal sized regions between the first and last bin. The mode of each region of the histogram is returned as an estimated state level value in the command window.

Use optional input arguments to specify the number of histogram bins, histogram bounds, and the state level estimation method.

Measure Rise Time, Fall Time and Slew Rate

Use `risetime` with no output argument to visualize the rise time of positive-going edges.
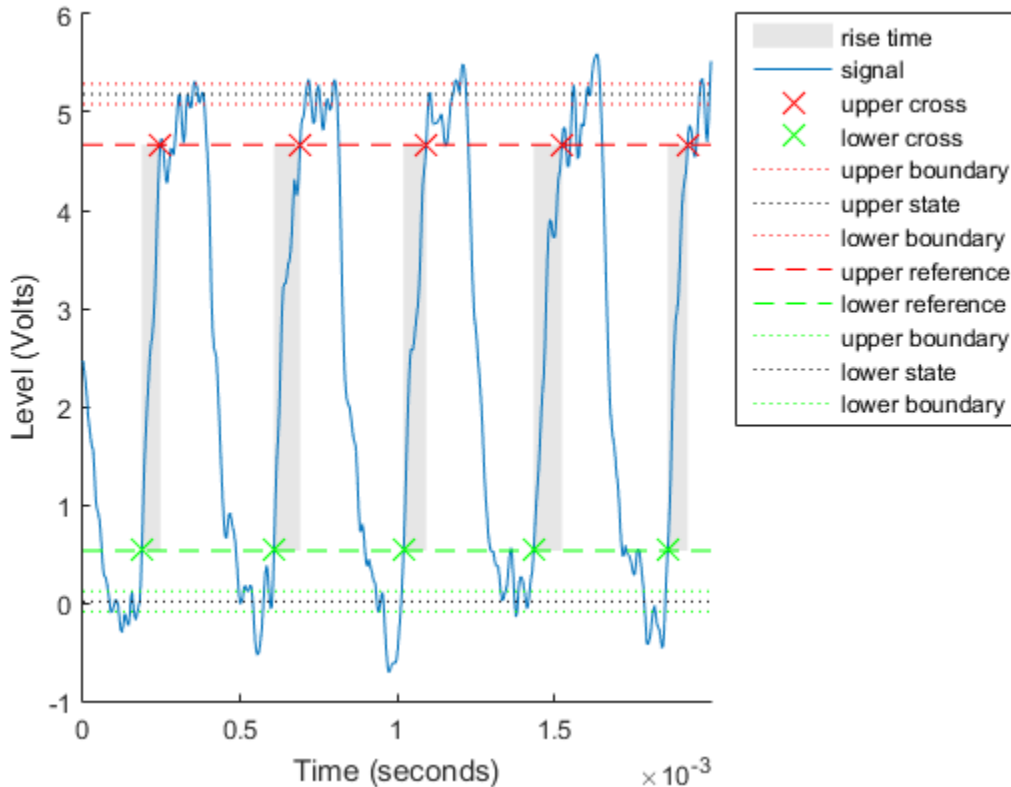
```
risetime(clock1,time1)


ans =

   1.0e-04 *

    0.5919
    0.8344
    0.7185
    0.8970
    0.6366
```

The default reference levels for computing rise time and fall time are set at 10% and 90% of the waveform amplitude.
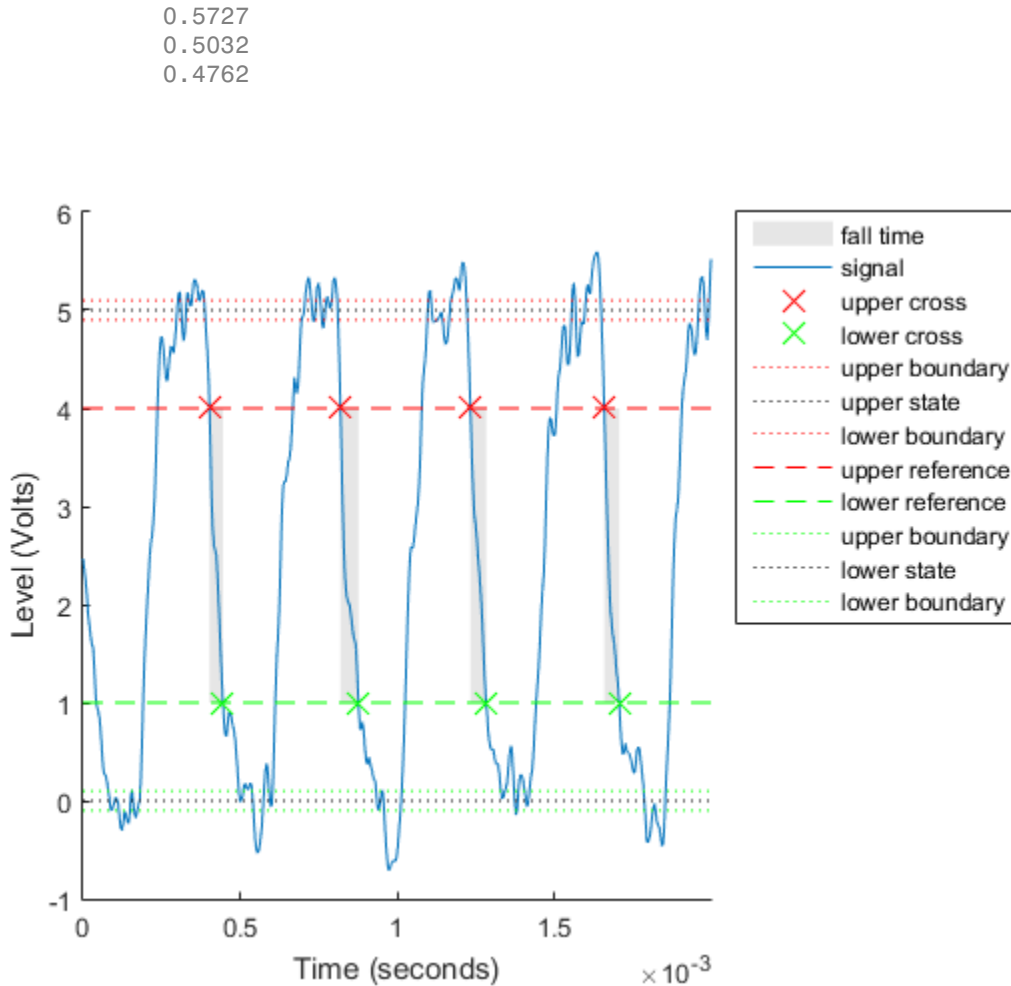
Specify custom reference and state levels via optional input arguments as shown below for a fall time measurement.

```
falltime(clock1,time1,'PercentReferenceLevels',[20 80],'StateLevels',[0 5])
```

```
ans =

   1.0e-04 *

   0.4294
```

```
   0.5727
   0.5032
   0.4762
```


Obtain measurements programmatically by calling functions with one or more output arguments. For uniformly sampled data, you can provide a sample rate in place of the time vector. Use `slewrate` to measure the slope of each positive-going or negative-going edge.

```
sr = slewrate(clock1(1:100), Fs)


sr =
```
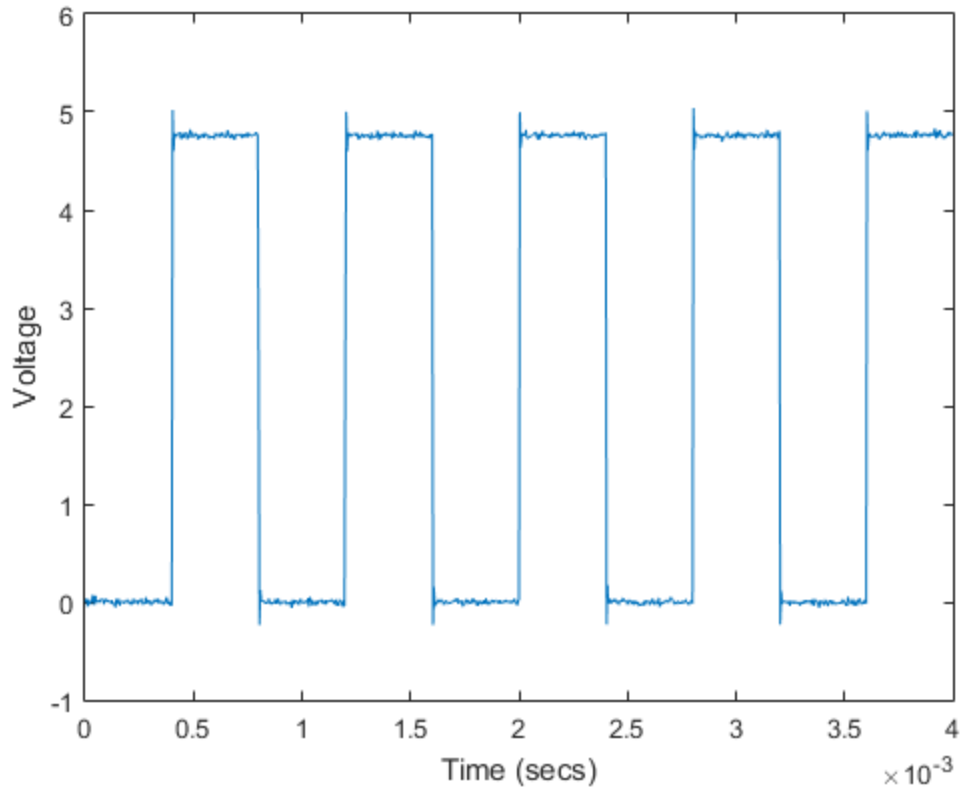
```
7.0840e+04
```

Analyze Overshoot and Undershoot

Now let's view data from a clock with significant overshoot and undershoot.

```
load clocksig clock2 time2 Fs
plot(time2, clock2);
xlabel('Time (secs)'), ylabel('Voltage');
```


Underdamped clock signals have overshoots. Overshoots are expressed as a percentage of the difference between state levels. Overshoots can occur just after an edge, at the start

of the post-transition aberration region. These are called postshoot overshoots. You can measure them by using the `overshoot` function.

```
overshoot(clock2(95:270),Fs)
legend('Location','NorthEast')


ans =

    4.9451
    2.5399
```
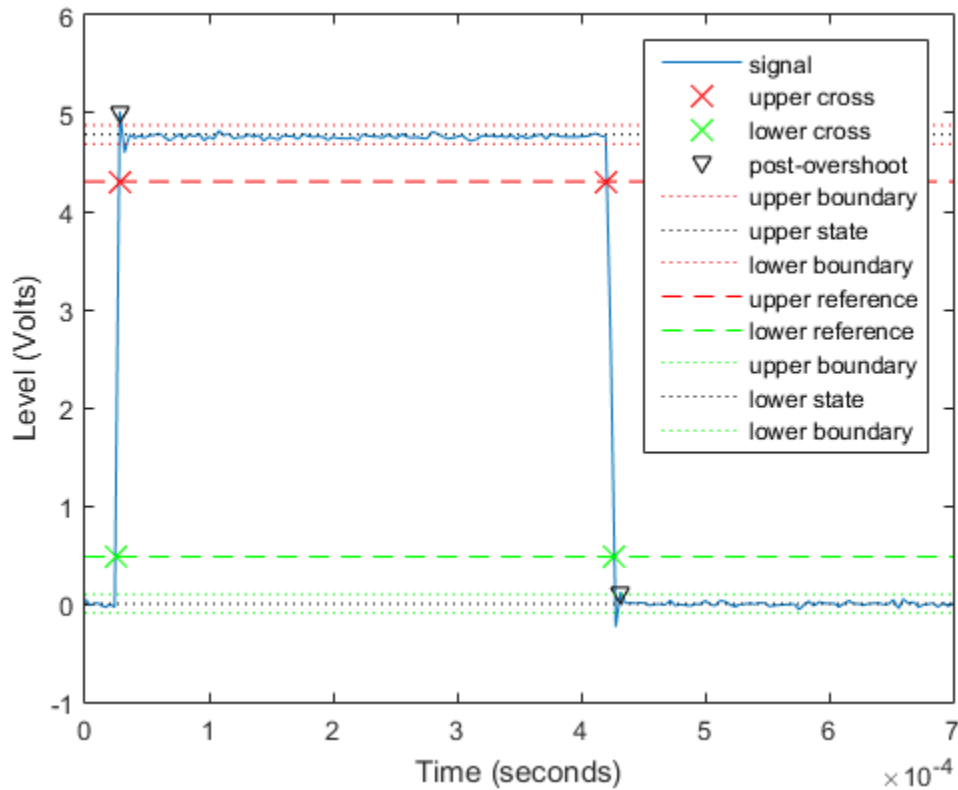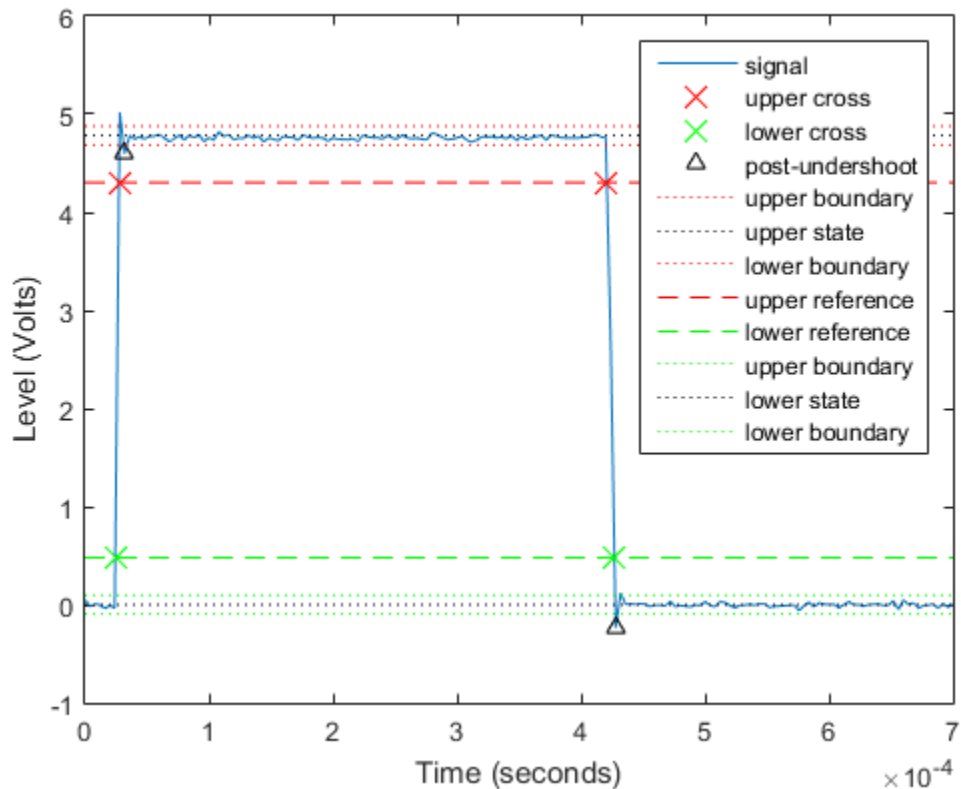

Overshoots may also occur just before an edge, at the end of the pre-transition aberration region. These are called preshoot overshoots. Similarly, you can measure undershoots

in the pre and post-aberration regions. Undershoots are also expressed as a percentage of the difference between the state levels. Use optional input arguments to specify the regions in which to measure aberrations.

```
undershoot(clock2(95:270),Fs,'Region','Postshoot')
legend('Location','NorthEast')
```

```
ans =

    3.8499
    4.9451
```

Measure Pulse Width and Duty Cycle

Use `pulsewidth` with no output argument to plot highlighted pulse widths.

```
pulsewidth(clock2, time2, 'Polarity', 'Positive');
```


This displays pulses of positive polarity. Select negative polarity to see the widths of negative polarity pulses.

Use `dutycycle` to compute the ratio of the pulse width to the pulse period for each positive-polarity or negative-polarity pulse.

```
d = dutycycle(clock2,time2,'Polarity','negative')
```

```
d =

    0.4979
    0.5000
    0.5000
```

Use `pulseperiod` to obtain the periods of each cycle of the waveform. Use this information to compute other metrics like the average frequency of the waveform or the total observed jitter.

```
pp = pulseperiod(clock2, time2);

avgFreq = 1./mean(pp)
totalJitter = std(pp)

avgFreq =

   1.2500e+03


totalJitter =

   1.9866e-06
```

Measuring Signal Similarities

This example shows how to measure signal similarities. It will help you answer questions such as: How do I compare signals with different lengths or different sampling rates? How do I find if there is a signal or just noise in a measurement? Are two signals related? How to measure a delay between two signals (and how do I align them)? How do I compare the frequency content of two signals? Similarities can also be found in different sections of a signal to determine if a signal is periodic.

Comparing Signals with Different Sampling Rates

Consider a database of audio signals and a pattern matching application where you need to identify a song as it is playing. Data is commonly stored at a low sampling rate to occupy less memory.

```
% Load data
load relatedsig.mat;

figure
ax(1) = subplot(311);
plot((0:numel(T1)-1)/Fs1,T1,'k');
ylabel('Template 1');
grid on
ax(2) = subplot(312);
plot((0:numel(T2)-1)/Fs2,T2,'r');
ylabel('Template 2');
grid on
ax(3) = subplot(313);
plot((0:numel(S)-1)/Fs,S);
ylabel('Signal');
grid on
xlabel('Time (secs)');
linkaxes(ax(1:3),'x')
axis([0 1.61 -4 4])
```
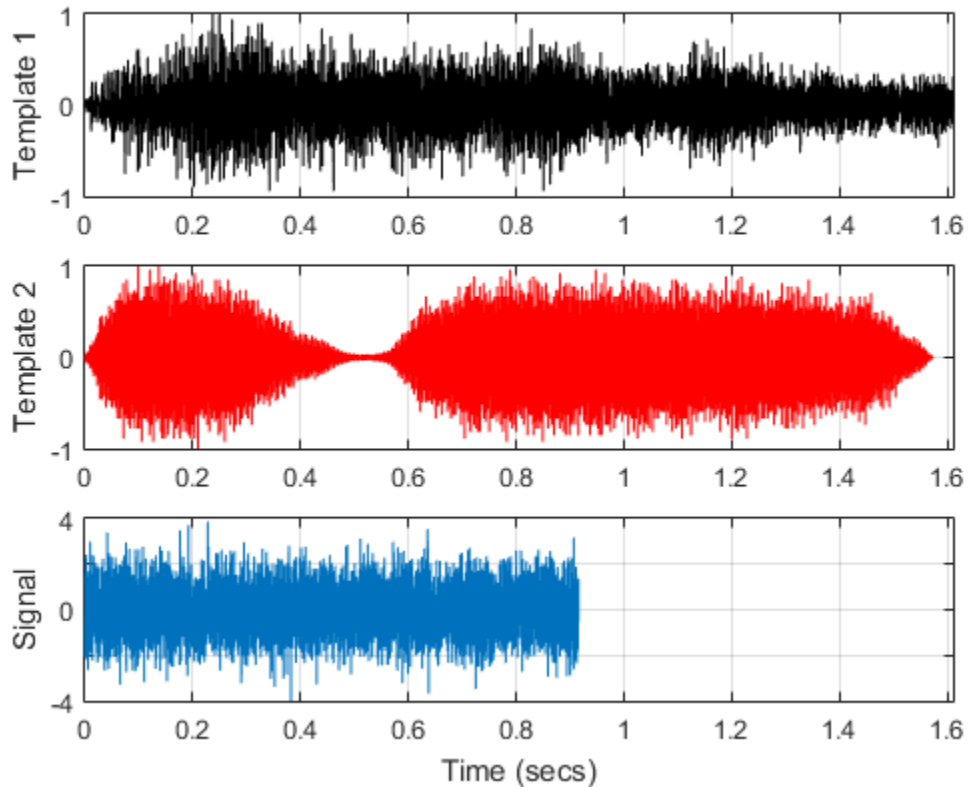
The first and the second subplot show the template signals from the database. The third subplot shows the signal which we want to search for in our database. Just by looking at the time series, the signal does not seem to match to any of the two templates. A closer inspection reveals that the signals actually have different lengths and sampling rates.

```
[Fs1 Fs2 Fs]
```

```
ans =

        4096          4096          8192
```

Different lengths prevent you from calculating the difference between two signals but this can easily be remedied by extracting the common part of signals. Furthermore, it is not always necessary to equalize lengths. Cross-correlation can be performed between signals with different lengths, but it is essential to ensure that they have identical sampling rates. The safest way to do this is to resample the signal with a lower sampling rate. The resample function applies an anti-aliasing(low-pass) FIR filter to the signal during the resampling process.

```
[P1,Q1] = rat(Fs/Fs1);          % Rational fraction approximation
[P2,Q2] = rat(Fs/Fs2);          % Rational fraction approximation
T1 = resample(T1,P1,Q1);        % Change sampling rate by rational factor
T2 = resample(T2,P2,Q2);        % Change sampling rate by rational factor
```

Finding a Signal in a Measurement

We can now cross-correlate signal S to templates T1 and T2 with the xcorr function to determine if there is a match.

```
[C1,lag1] = xcorr(T1,S);
[C2,lag2] = xcorr(T2,S);

figure
ax(1) = subplot(211);
plot(lag1/Fs,C1,'k');
ylabel('Amplitude');
grid on
title('Cross-correlation between Template 1 and Signal')
ax(2) = subplot(212);
plot(lag2/Fs,C2,'r');
ylabel('Amplitude');
grid on
title('Cross-correlation between Template 2 and Signal')
xlabel('Time(secs)');
axis(ax(1:2),[-1.5 1.5 -700 700 ])
```

Cross-correlation between Template 1 and Signal

Cross-correlation between Template 2 and Signal
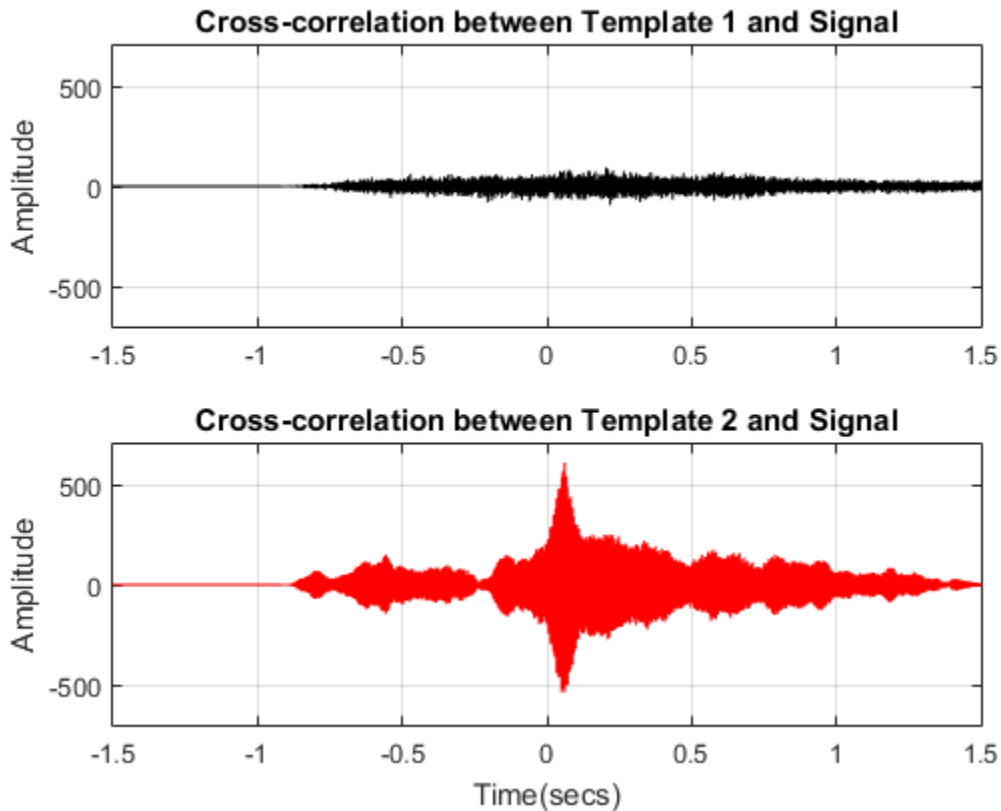
The first subplot indicates that the signal and template 1 are less correlated while the high peak in the second subplot indicates that signal is present in the second template.

```
[~,I] = max(abs(C2));
timeDiff = lag2(I)/Fs
```

```
timeDiff =

    0.0609
```

The peak of the cross correlation implies that the signal is present in template T2 starting after 61 ms.

Measuring Delay Between Signals and Aligning Them

Consider a situation where you are collecting data from different sensors, recording vibrations caused by cars on both sides of a bridge. When you analyze the signals, you may need to align them. Assume you have 3 sensors working at same sampling rates and they are measuring signals caused by the same event.

```
figure,
ax(1) = subplot(311);
plot(s1);
ylabel('s1');
grid on
ax(2) = subplot(312);
plot(s2,'k');
ylabel('s2');
grid on
ax(3) = subplot(313);
plot(s3,'r');
ylabel('s3');
grid on
xlabel('Samples')
linkaxes(ax,'xy')
```
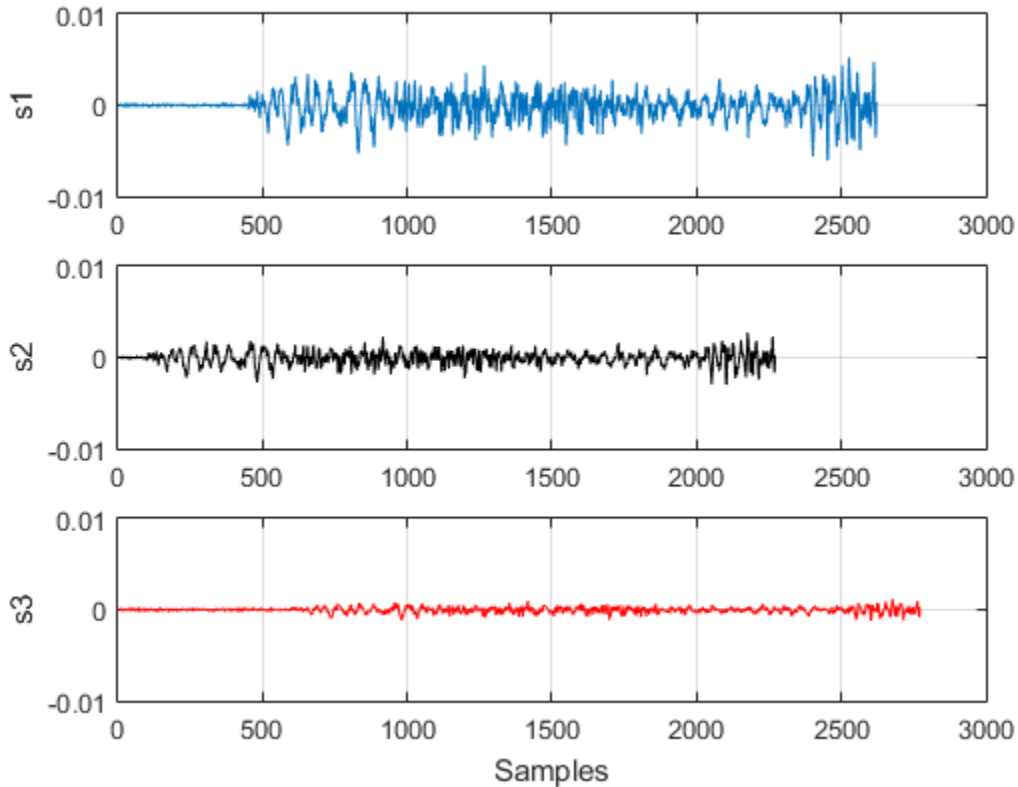
The maximum value of the cross-correlations between s1 and s2 and s1 and s3 indicate time leads/lags.

```
[C21,lag1] = xcorr(s2,s1);
[C31,lag2] = xcorr(s3,s1);

figure
subplot(211);
plot(lag1,C21/max(C21));
ylabel('C21');
grid on
title('Cross-Correlations')
subplot(212);
plot(lag2,C31/max(C31));
```
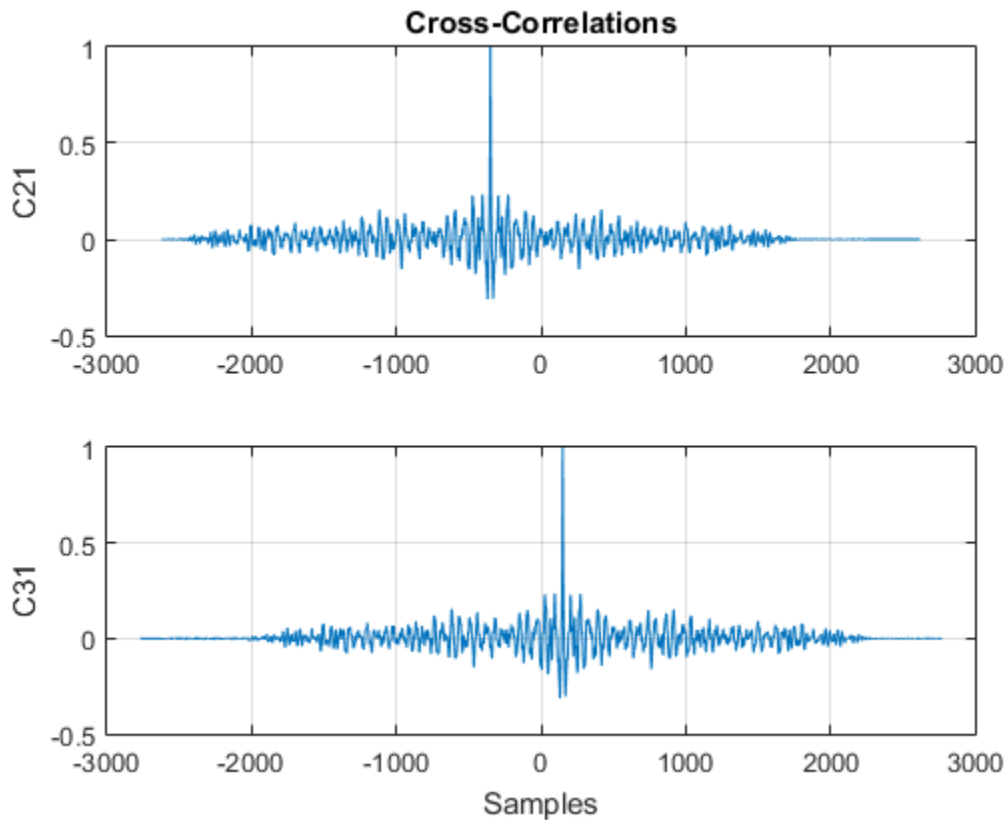
```
ylabel('C31');
grid on
xlabel('Samples')

[~,I1] = max(abs(C21));      % Find the index of the highest peak
[~,I2] = max(abs(C31));      % Find the index of the highest peak
t21 = lag1(I1)               % Time difference between the signals s2,s1
t31 = lag2(I2)               % Time difference between the signals s3,s1


t21 =

  -350


t31 =

   150
```
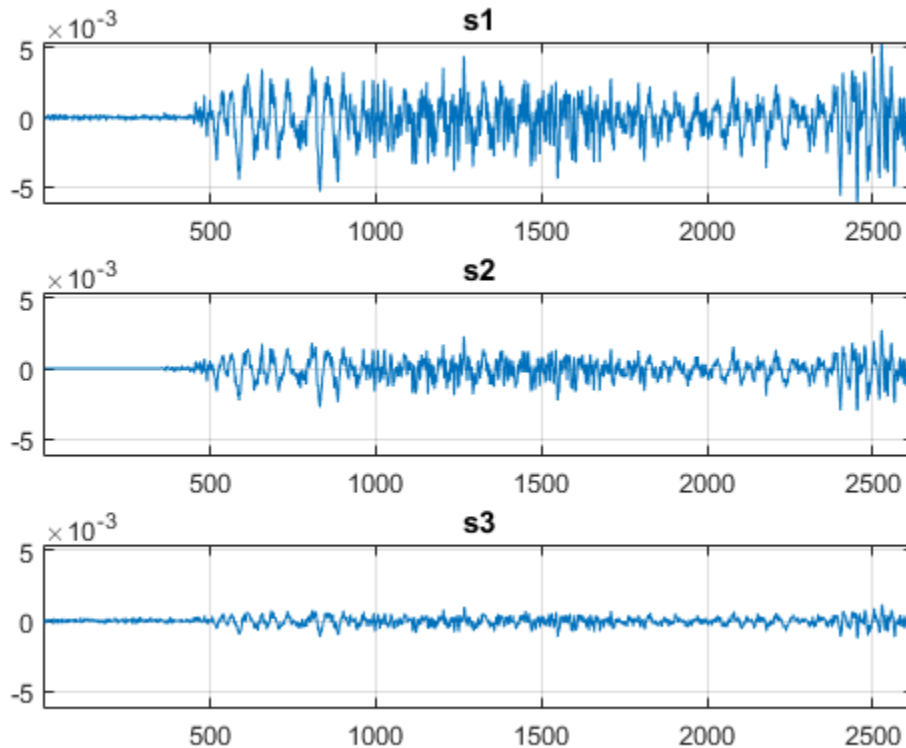
t21 indicates that s2 lags s1 by 350 samples, and t31 indicates that s3 leads s1 by 150 samples. This information can now used to align the 3 signals.

```
s2 = [zeros(abs(t21),1);s2];
s3 = s3(t31:end);

figure
ax(1) = subplot(311);
plot(s1);
grid on;
title('s1');
axis tight
ax(2) = subplot(312);
plot(s2);
```

```
grid on;
title('s2');
axis tight
ax(3) = subplot(313);
plot(s3);
grid on;
title('s3');
axis tight
linkaxes(ax,'xy')
```


## Comparing the Frequency Content of Signals

A power spectrum displays the power present in each frequency. Spectral coherence identifies frequency-domain correlation between signals. Coherence values tending

towards 0 indicate that the corresponding frequency components are uncorrelated while values tending towards 1 indicate that the corresponding frequency components are correlated. Consider two signals and their respective power spectra.

```matlab
Fs = FsSig;              % Sampling Rate

[P1,f1] = periodogram(sig1,[],[],Fs,'power');
[P2,f2] = periodogram(sig2,[],[],Fs,'power');

figure
t = (0:numel(sig1)-1)/Fs;
subplot(221);
plot(t,sig1,'k');
ylabel('s1');
grid on
title('Time Series')
subplot(223);
plot(t,sig2);
ylabel('s2');
grid on
xlabel('Time (secs)')
subplot(222);
plot(f1,P1,'k');
ylabel('P1');
grid on;
axis tight
title('Power Spectrum')
subplot(224);
plot(f2,P2);
ylabel('P2');
grid on;
axis tight
xlabel('Frequency (Hz)')
```
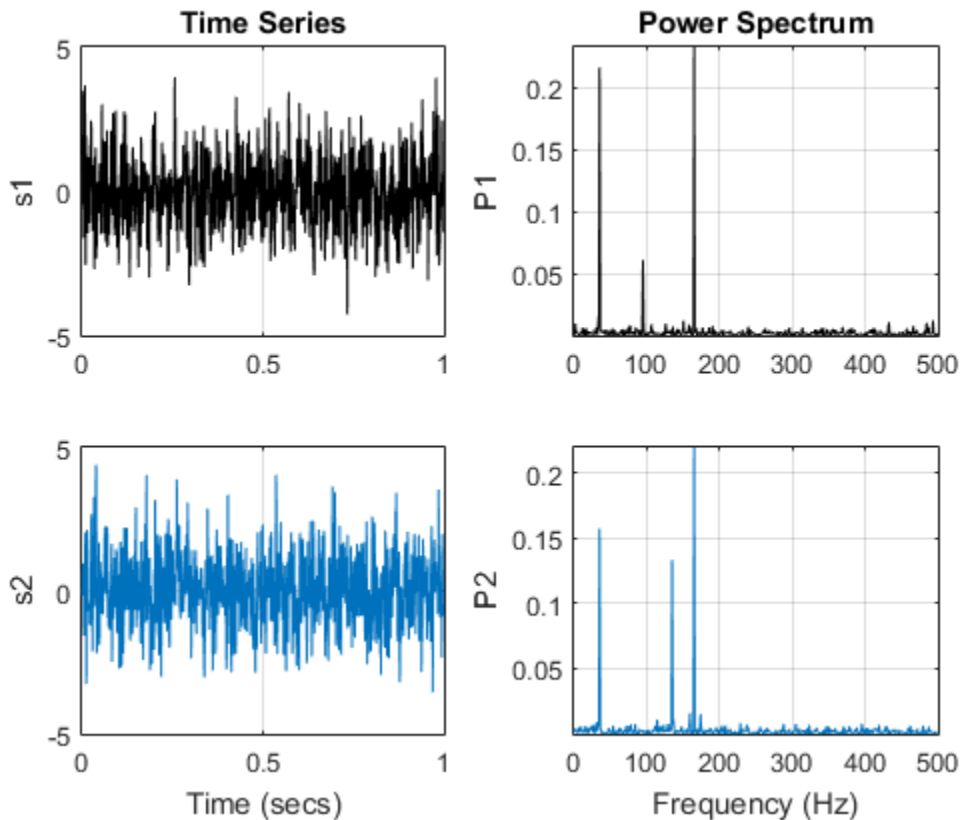
The `mscohere` function calculates the spectral coherence between the two signals. It confirms that sig1 and sig2 have two correlated components around 35 Hz and 165 Hz. In frequencies where spectral coherence is high, the relative phase between the correlated components can be estimated with the cross spectrum phase.

```
[Cxy,f] = mscohere(sig1,sig2,[],[],[],Fs);
Pxy     = cpsd(sig1,sig2,[],[],[],Fs);
phase   = -angle(Pxy)/pi*180;
[pks,locs] = findpeaks(Cxy,'MinPeakHeight',0.75);

figure
subplot(211);
plot(f,Cxy);
title('Coherence Estimate');
```

```
grid on;
hgca = gca;
hgca.XTick = f(locs);
hgca.YTick = .75;
axis([0 200 0 1])
subplot(212);
plot(f,phase);
title('Cross Spectrum Phase (deg)');
grid on;
hgca = gca;
hgca.XTick = f(locs);
hgca.YTick = round(phase(locs));
xlabel('Frequency (Hz)');
axis([0 200 -180 180])
```
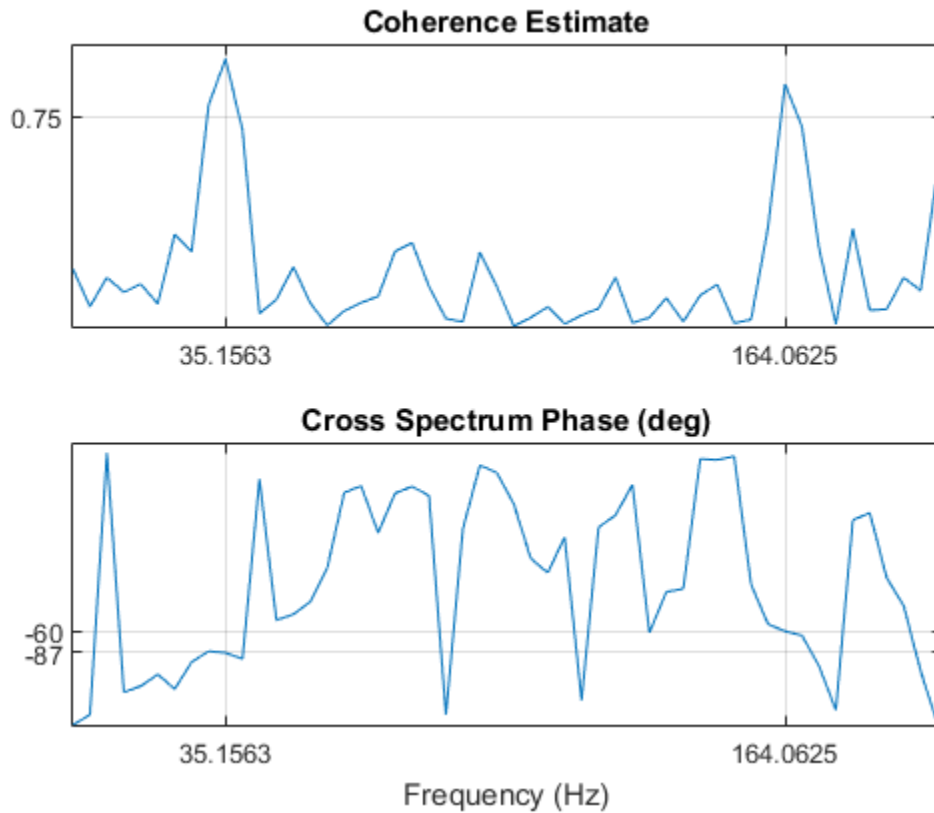
The phase lag between the 35 Hz components is close to -90 degrees, and the phase lag between the 165 Hz components is close to -60 degrees.

Finding Periodicities in a Signal

Consider a set of temperature measurements in an office building during the winter season. Measurements were taken every 30 minutes for about 16.5 weeks.

```
load officetemp.mat  % Load Temperature Data
Fs = 1/(60*30);                  % Sample rate is 1 sample every 30 minutes
days = (0:length(temp)-1)/(Fs*60*60*24);

figure
plot(days,temp)
title('Temperature Data')
xlabel('Time (days)');
ylabel('Temperature (Fahrenheit)')
grid on
```
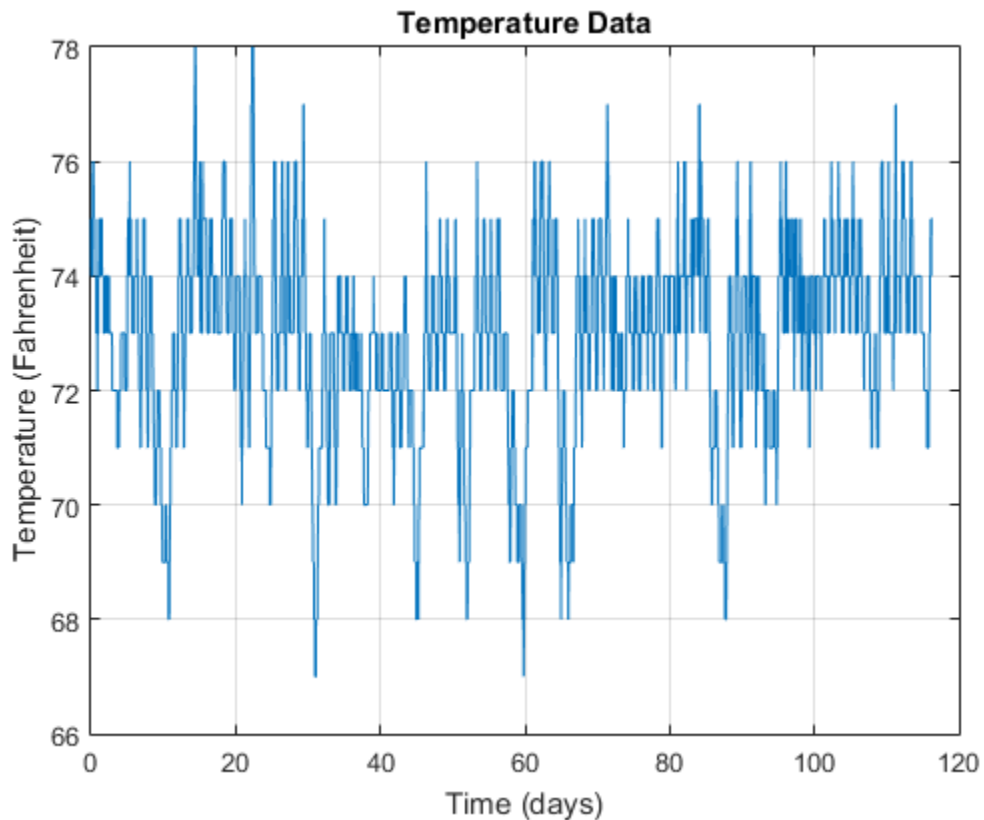
Temperature Data

With the temperatures in the low 70s, you need to remove the mean to analyze small fluctuations in the signal. The `xcov` function removes the mean of the signal before computing the cross-correlation. It returns the cross-covariance. Limit the maximum lag to 50% of the signal to get a good estimate of the cross-covariance.

```
maxlags = numel(temp)*0.5;
[xc,lag] = xcov(temp,maxlags);

[~,df] = findpeaks(xc,'MinPeakDistance',5*2*24);
[~,mf] = findpeaks(xc);

figure
plot(lag/(2*24),xc,'k',...
     lag(df)/(2*24),xc(df),'kv','MarkerFaceColor','r')
```
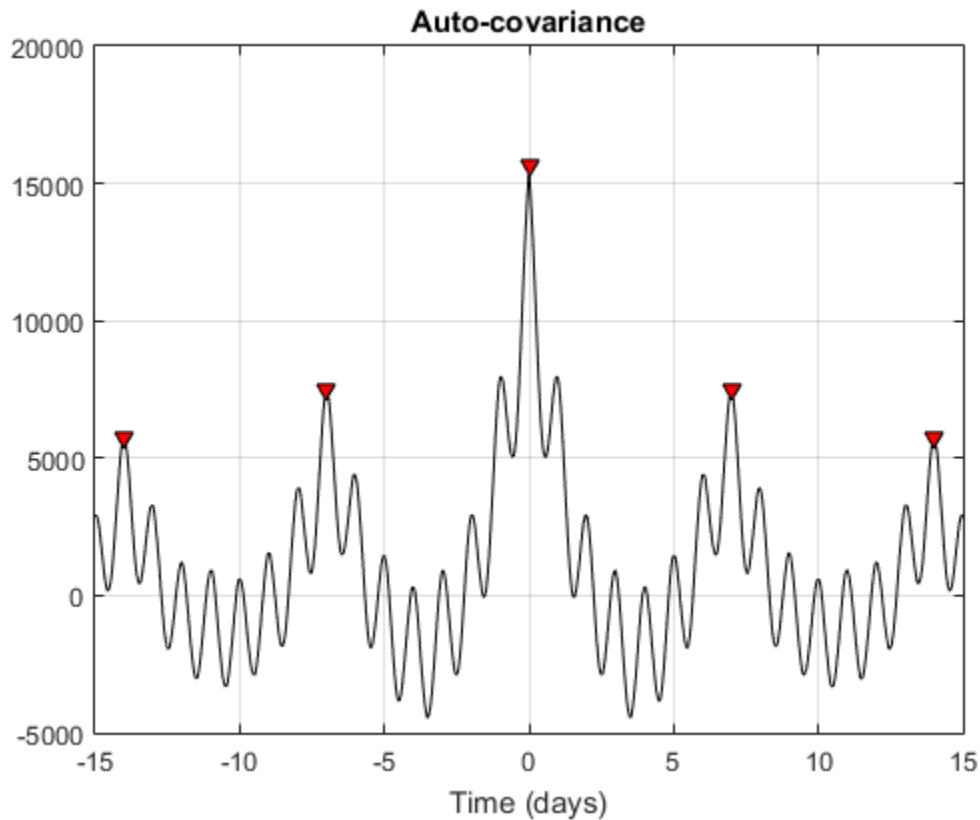
```
grid on
xlim([-15 15]);
xlabel('Time (days)')
title('Auto-covariance')
```


Observe dominant and minor fluctuations in the auto-covariance. Dominant and minor peaks appear equidistant. To verify if they are, compute and plot the difference between the locations of subsequent peaks.

```
cycle1 = diff(df)/(2*24);
cycle2 = diff(mf)/(2*24);

subplot(211);
plot(cycle1);
```
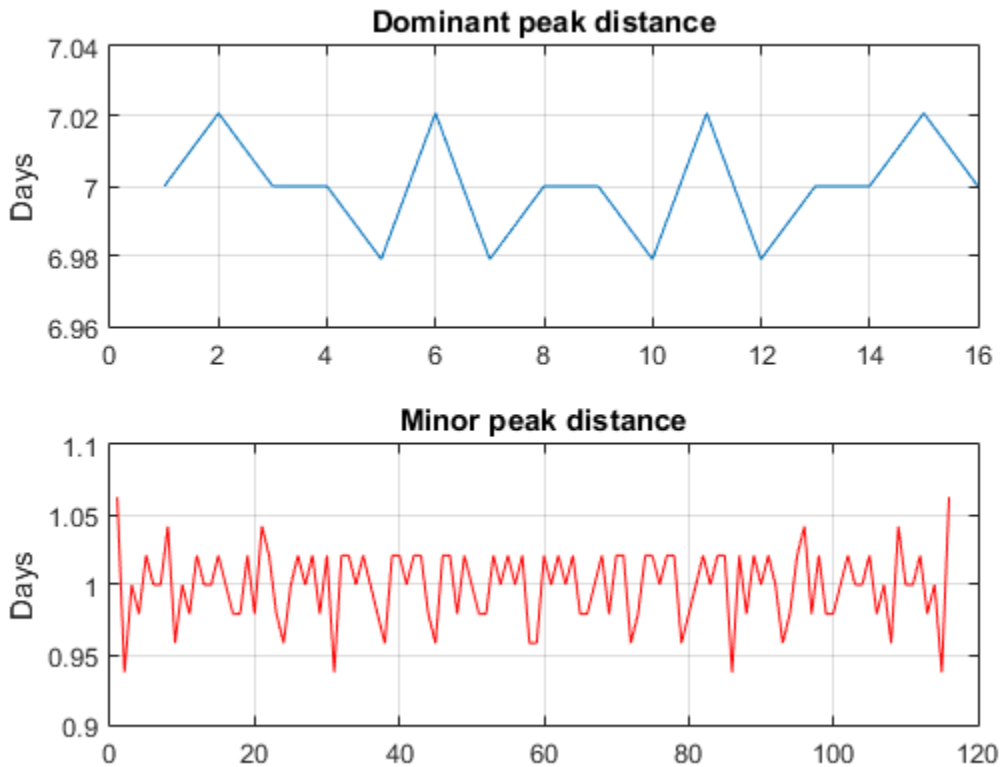
```
ylabel('Days');
grid on
title('Dominant peak distance')
subplot(212);
plot(cycle2,'r');
ylabel('Days');
grid on
title('Minor peak distance')

mean(cycle1)
mean(cycle2)


ans =

     7


ans =

    1.0000
```

The minor peaks indicate 7 cycles/week and the dominant peaks indicate 1 cycle per week. This makes sense given that the data comes from a temperature-controlled building on a 7 day calendar. The first 7-day cycle indicates that there is a weekly cyclic behavior of the building temperature where temperatures lower during the weekends and go back to normal during the week days. The 1-day cycle behavior indicates that there is also a daily cyclic behavior - temperatures lower during the night and increase during the day.

Signal Smoothing

This example shows how to use moving average filters and resampling to isolate the effect of periodic components of the time of day on hourly temperature readings, as well as remove unwanted line noise from an open-loop voltage measurement. The example also shows how to remove noise spikes from a signal and replace the missing sample data by using a median filter.

Motivation

Smoothing is how we discover important patterns in our data while leaving out things that are unimportant (i.e. noise). We use filtering to perform this smoothing. The goal of smoothing is to produce slow changes in value so that it's easier to see trends in our data.

Sometimes when you examine input data you may wish to smooth the data in order to see a trend in the signal. In our example we have a set of temperature readings in Celsius taken every hour at Logan Airport for the entire month of January, 2011.

```
load bostemp
days = (1:31*24)/24;
plot(days, tempC), axis tight;
ylabel('Temp (\circC)');
xlabel('Time elapsed from Jan 1, 2011 (days)');
title('Logan Airport Dry Bulb Temperature (source: NOAA)');
```

Logan Airport Dry Bulb Temperature (source: NOAA)
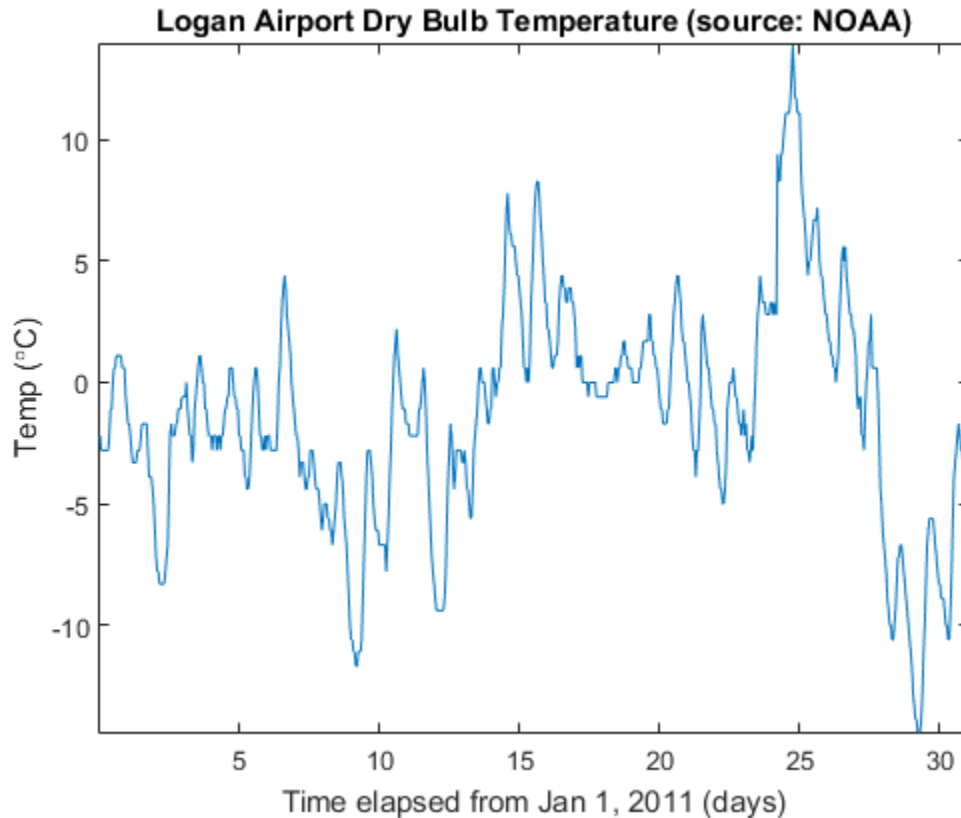
Note that we can visually see the effect that the time of day has upon the temperature readings. If you are only interested in the daily temperature variation over the month, the hourly fluctuations only contribute noise, which can make the daily variations difficult to discern. To remove the effect of the time of day, we would now like to smooth our data by using a moving average filter.
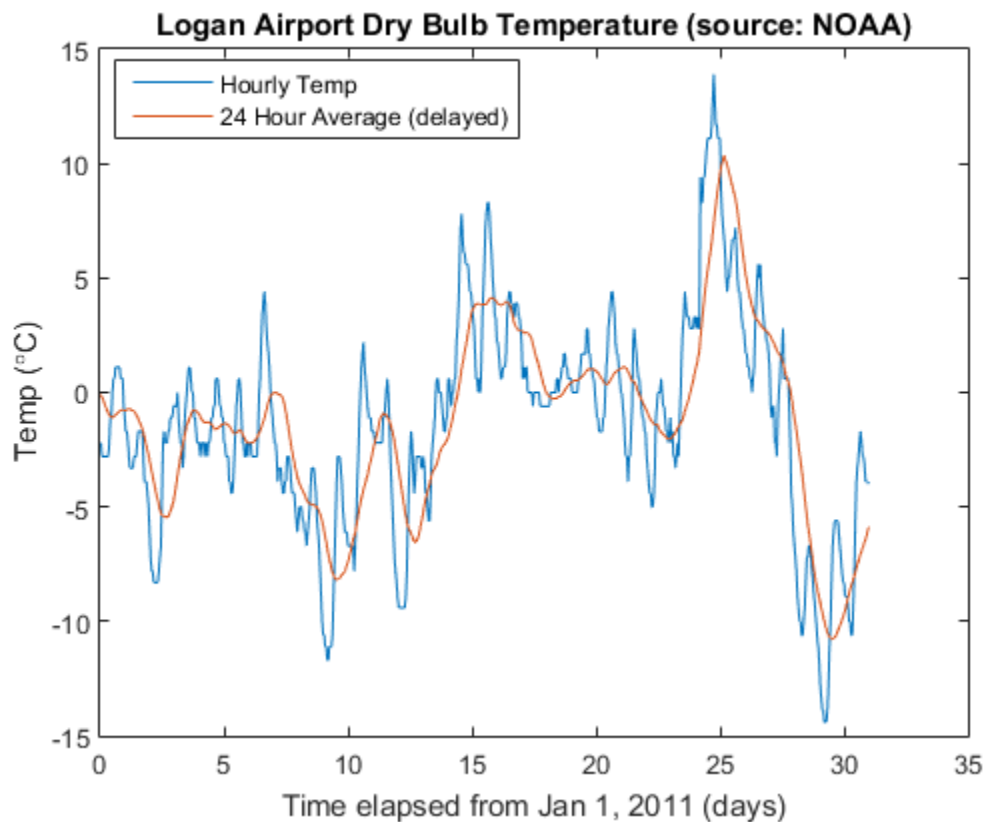
A Moving Average Filter

In its simplest form, a moving average filter of length N takes the average of every N consecutive samples of the waveform.

To apply a moving average filter to each data point, we construct our coefficients of our filter so that each point is equally weighted and contributes 1/24 to the total average. This gives us the average temperature over each 24 hour period.

```
hoursPerDay = 24;
coeff24hMA = ones(1, hoursPerDay)/hoursPerDay;

avg24hTempC = filter(coeff24hMA, 1, tempC);
plot(days, [tempC avg24hTempC]);
legend('Hourly Temp','24 Hour Average (delayed)','location','best');
ylabel('Temp (\circC)');
xlabel('Time elapsed from Jan 1, 2011 (days)');
title('Logan Airport Dry Bulb Temperature (source: NOAA)');
```

Filter Delay

Note that the filtered output is delayed by about twelve hours. This is due to the fact that our moving average filter has a delay.

Any symmetric filter of length N will have a delay of (N-1)/2 samples. We can account for this delay manually.

```
fDelay = (length(coeff24hMA)-1)/2;
plot(days, tempC, 'b', ...
     days-fDelay/24, avg24hTempC, 'g');
axis tight;
legend('Hourly Temp','24 Hour Average','location','best');
ylabel('Temp (\circC)');
xlabel('Time elapsed from Jan 1, 2011 (days)');
title('Logan Airport Dry Bulb Temperature (source: NOAA)');
```

Logan Airport Dry Bulb Temperature (source: NOAA)
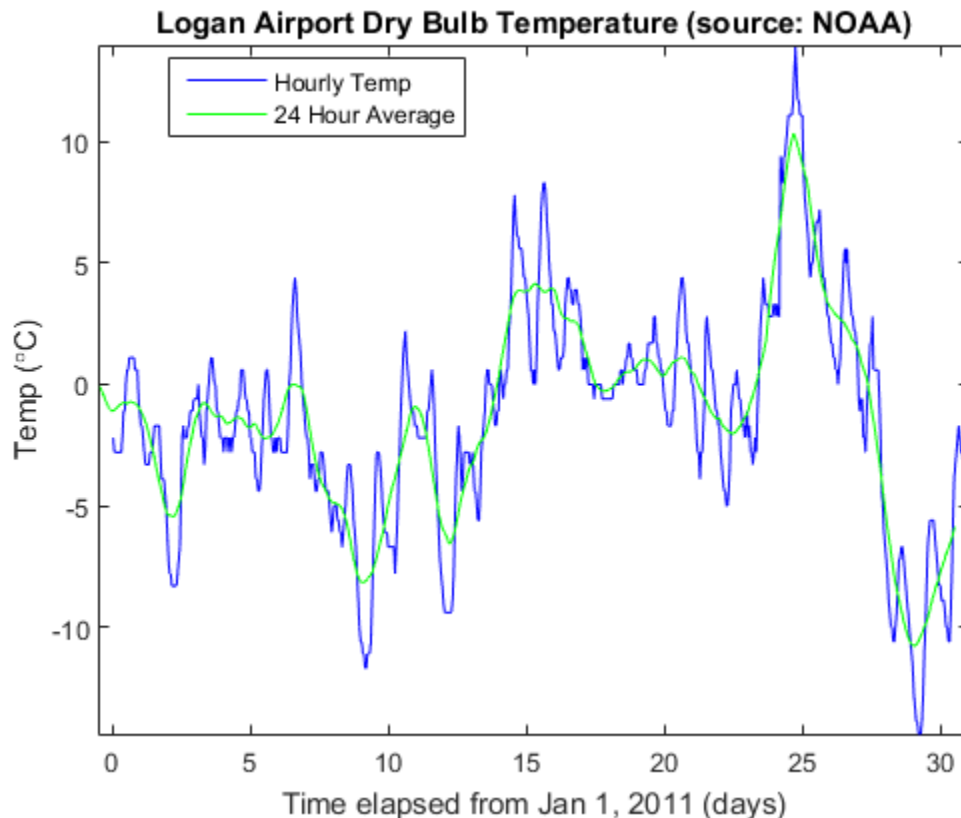
Extracting Average Differences

Alternatively, we can also use the moving average filter to obtain a better estimate of how the time of day affects the overall temperature. To do this, first, subtract the smoothed data from the hourly temperature measurements. Then, segment the differenced data into days and take the average over all 31 days in the month.

```
figure
deltaTempC = tempC - avg24hTempC;
deltaTempC = reshape(deltaTempC, 24, 31).';

plot(1:24, mean(deltaTempC)), axis tight;
title('Mean temperature differential from 24 hour average');
xlabel('Hour of day (since midnight)');
```

```
ylabel('Temperature difference (\circC)');
```


Mean temperature differential from 24 hour average
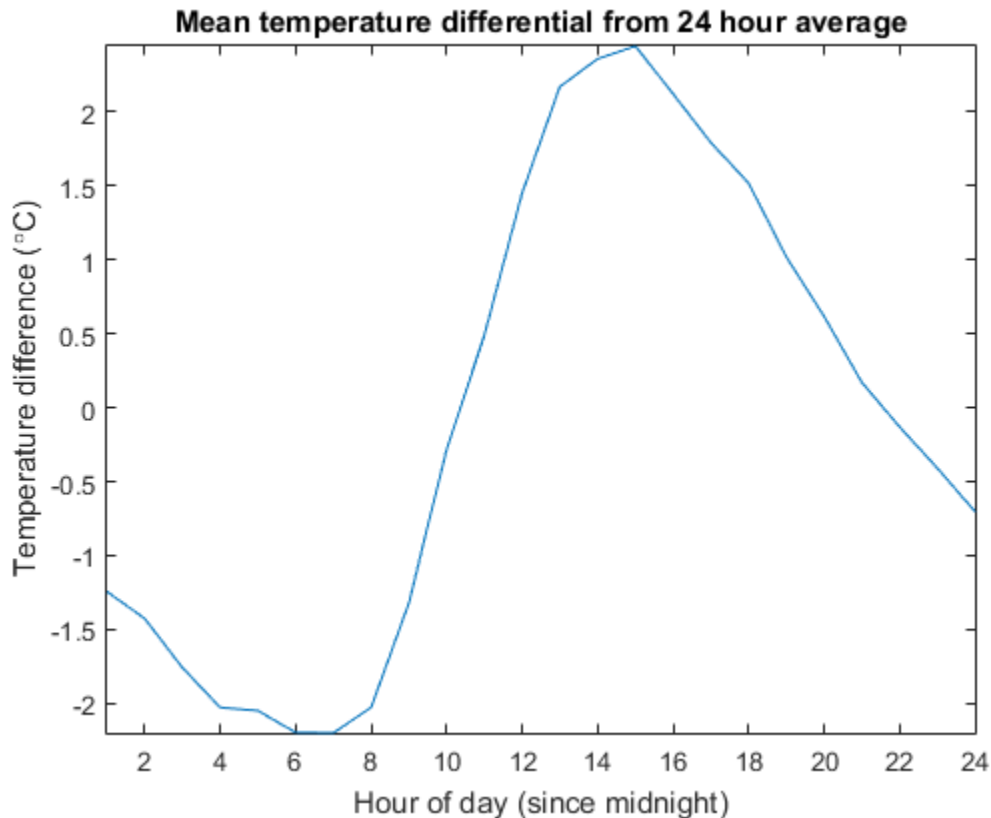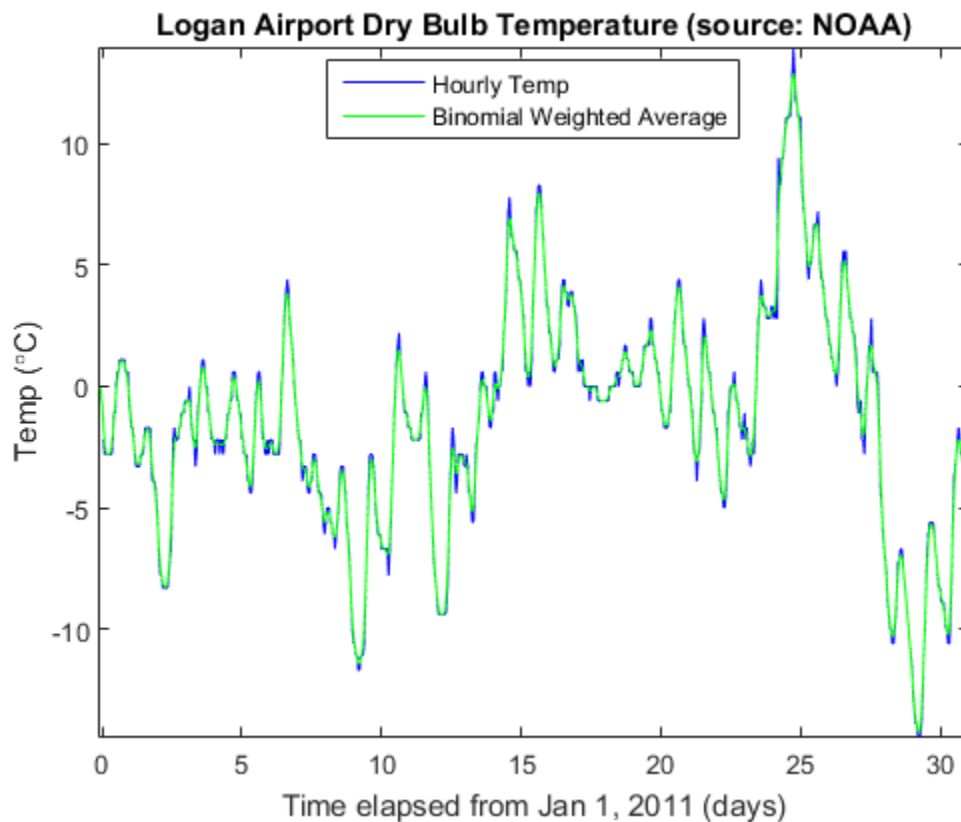
Weighted Moving Average Filters

Other kinds of moving average filters do not weight each sample equally.

Another common filter follows the binomial expansion of $(1/2, 1/2)^n$ This type of filter approximates a normal curve for large values of n. It is useful for filtering out high frequency noise for small n. To find the coefficients for the binomial filter, convolve [1/2 1/2] with itself and then iteratively convolve the output with [1/2 1/2] a prescribed number of times. In this example, use five total iterations.

```
h = [1/2 1/2];
binomialCoeff = conv(h,h);
```

```
for n = 1:4
    binomialCoeff = conv(binomialCoeff,h);
end

figure
fDelay = (length(binomialCoeff)-1)/2;
binomialMA = filter(binomialCoeff, 1, tempC);
plot(days, tempC, 'b', ...
     days-fDelay/24,  binomialMA, 'g');
axis tight;
legend('Hourly Temp','Binomial Weighted Average','location','best');
ylabel('Temp (\circC)');
xlabel('Time elapsed from Jan 1, 2011 (days)');
title('Logan Airport Dry Bulb Temperature (source: NOAA)');
```

Another filter somewhat similar to the Gaussian expansion filter is the exponential moving average filter. This type of weighted moving average filter is easy to construct and does not require a large window size.

You adjust an exponentially weighted moving average filter by an alpha parameter between zero and one. A higher value of alpha will have less smoothing.

```
alpha = 0.45;
exponentialMA = filter(alpha, [1 alpha-1], tempC);
plot(days, tempC, 'b', ...
     days-fDelay/24,  binomialMA, 'g', ...
     days-1/24,       exponentialMA,'r');

axis tight;
legend('Hourly Temp', ...
       'Binomial Weighted Average', ...
       'Exponential Weighted Average','location','best');
ylabel('Temp (\circC)');
xlabel('Time elapsed from Jan 1, 2011 (days)');
title('Logan Airport Dry Bulb Temperature (source: NOAA)');
```

Logan Airport Dry Bulb Temperature (source: NOAA)
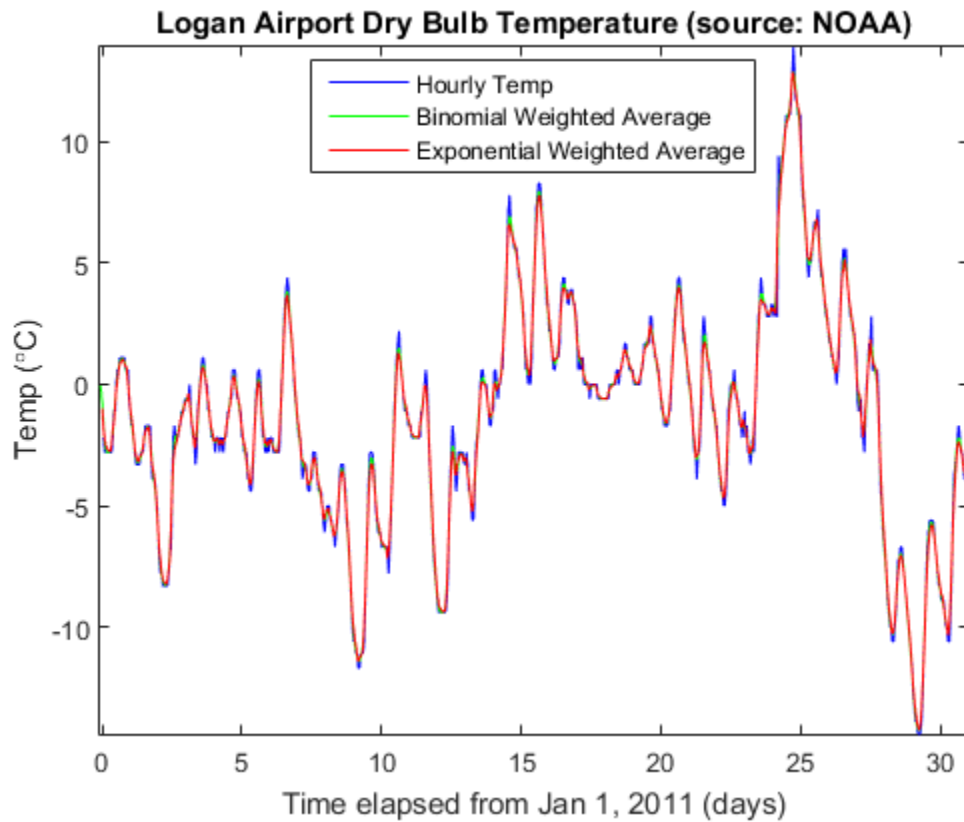
Zoom in on the readings for one day.

```
axis([3 4 -5 2]);
```

**Logan Airport Dry Bulb Temperature (source: NOAA)**
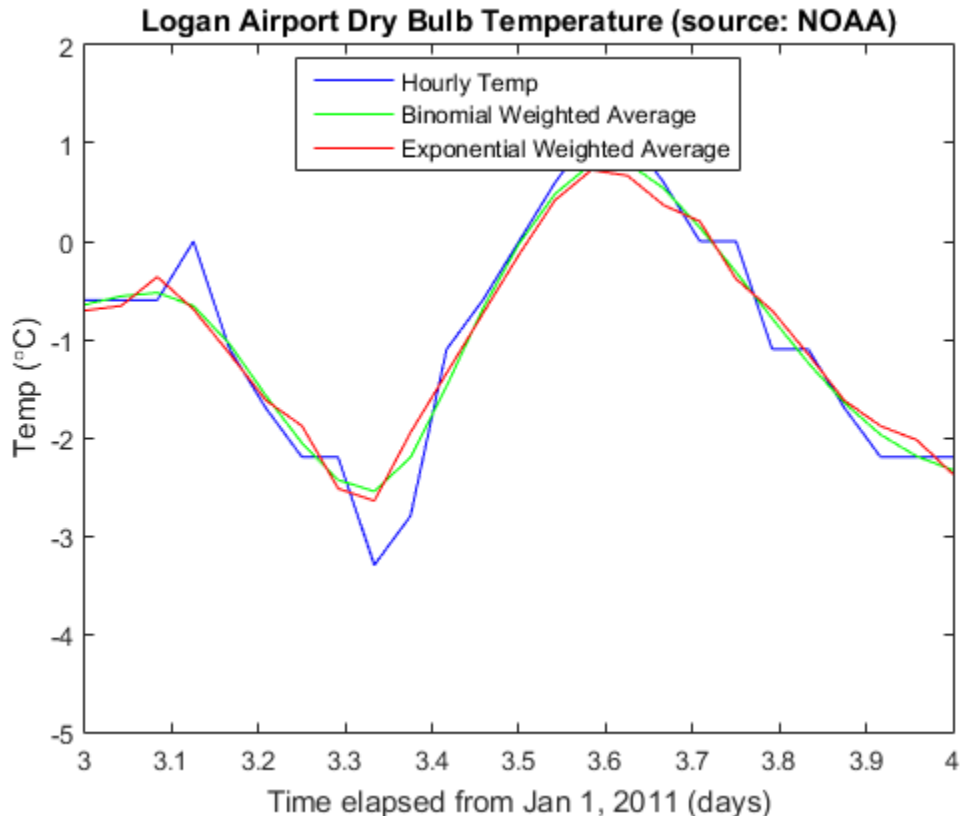
You'll note that by smoothing the data, the extreme values were somewhat clipped.

To track the signal a little more closely, you can use a weighted moving average filter that attempts to fit a polynomial of a specified order over a specified number of samples in a least-squares sense.

As a convenience, you can use the function `sgolayfilt` to implement a Savitzky-Golay smoothing filter. To use `sgolayfilt`, you specify an odd-length segment of the data and a polynomial order strictly less than the segment length. The `sgolayfilt` function internally computes the smoothing polynomial coefficients, performs delay alignment, and takes care of transient effects at the start and end of the data record.

```
cubicMA   = sgolayfilt(tempC, 3, 7);
```

```
quarticMA = sgolayfilt(tempC, 4, 7);
quinticMA = sgolayfilt(tempC, 5, 9);
plot(days, [tempC cubicMA quarticMA quinticMA]);
legend('Hourly Temp','Cubic-Weighted MA', 'Quartic-Weighted MA', ...
       'Quintic-Weighted MA','location','southeast');
ylabel('Temp (\circC)');
xlabel('Time elapsed from Jan 1, 2011 (days)');
title('Logan Airport Dry Bulb Temperature (source: NOAA)');
axis([3 5 -5 2]);
```
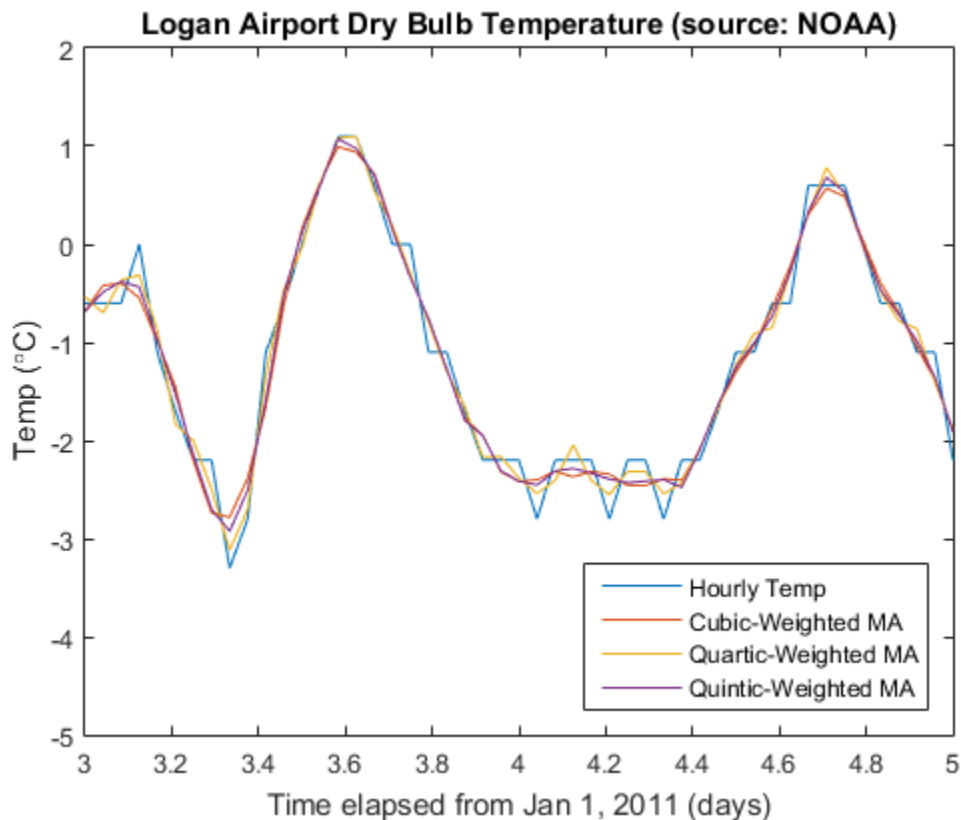

Resampling

Sometimes it is beneficial to resample a signal in order to properly apply a moving average.

In our next example, we sampled the open-loop voltage across the input of an analog instrument in the presence of interference from 60 Hz AC power line noise. We sampled the voltage with a 1 kHz sampling rate.

```
load openloop60hertz
fs = 1000;
t = (0:numel(openLoopVoltage)-1) / fs;
plot(t,openLoopVoltage);
ylabel('Voltage (V)');
xlabel('Time (s)');
title('Open-loop Voltage Measurement');
```
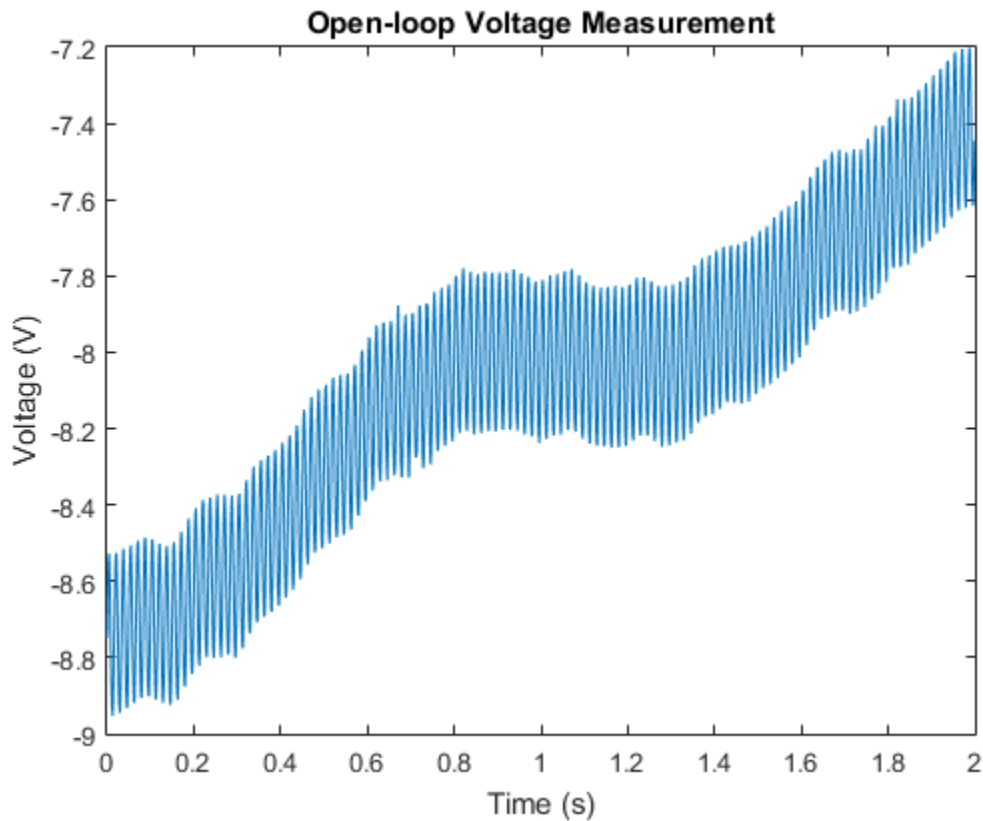

Let's attempt to remove the effect of the line noise by using a moving average filter.

If you construct a uniformly weighted moving average filter, it will remove any component that is periodic with respect to the duration of the filter.

There are roughly 1000 / 60 = 16.667 samples in a complete cycle of 60 Hz when sampled at 1000 Hz. Let's attempt to "round up" and use a 17-point filter. This will give us maximal filtering at a fundamental frequency of 1000 Hz / 17 = 58.82 Hz.

```
plot(t,sgolayfilt(openLoopVoltage,1,17));
ylabel('Voltage (V)');
xlabel('Time (s)');
title('Open-loop Voltage Measurement');
legend('Moving average filter operating at 58.82 Hz', ...
        'location','southeast');
```
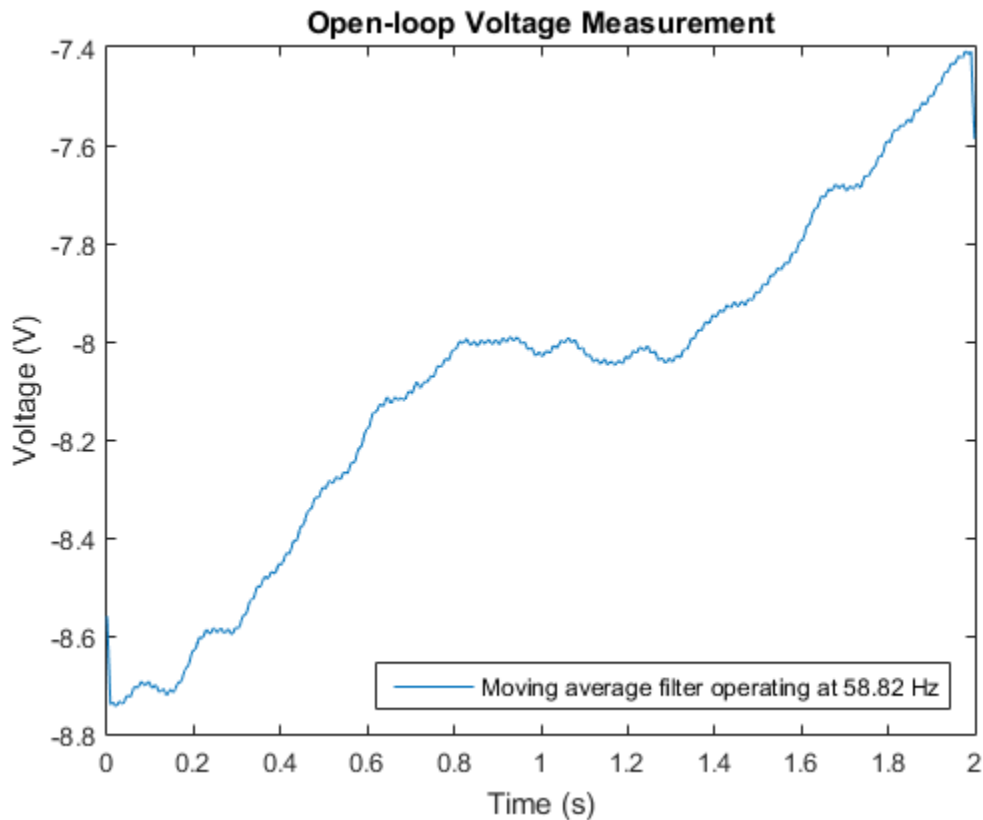
Note that while the voltage is significantly smoothed, it still contains a small 60 Hz ripple.

We can significantly reduce the ripple if we resample the signal so that we capture a complete full cycle of the 60 Hz signal by our moving average filter.

If we resample the signal at 17 * 60 Hz = 1020 Hz, we can use our 17 point moving average filter to remove the 60 Hz line noise.

```
fsResamp = 1020;
vResamp = resample(openLoopVoltage, fsResamp, fs);
tResamp = (0:numel(vResamp)-1) / fsResamp;
vAvgResamp = sgolayfilt(vResamp,1,17);
plot(tResamp,vAvgResamp);
ylabel('Voltage (V)');
xlabel('Time (s)');
title('Open-loop Voltage Measurement');
legend('Moving average filter operating at 60 Hz','location','southeast');
```

Open-loop Voltage Measurement
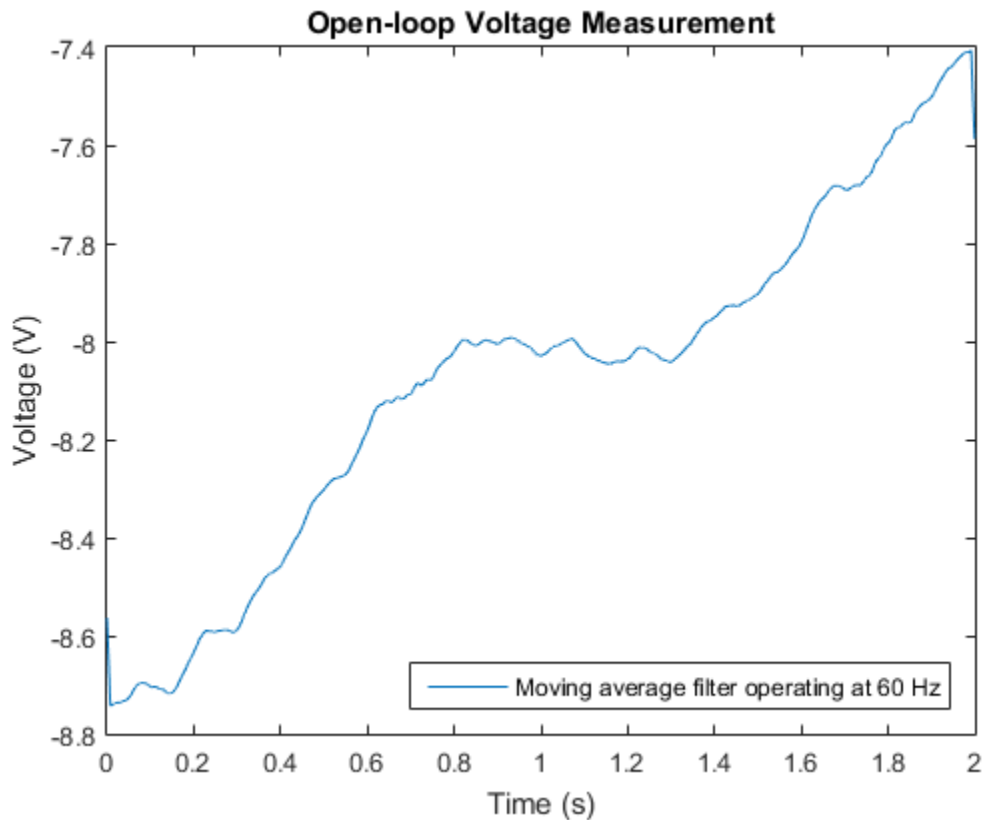
Median Filtering

Sometimes your data can exhibit unwanted noise "spikes" that last for short durations of time. Moving average filters can be unduly influenced by large short-term deviations in the signal value.

We have corrupted our previous example's signal with several large spikes.

```
spikeSignal = zeros(size(openLoopVoltage));
spikeSignal(1:100:2000) = -6;
noisyLoopVoltage = openLoopVoltage + spikeSignal;
plot(t, noisyLoopVoltage)
ylabel('Voltage (V)');
```

```
xlabel('Time (s)');
title('Open-loop Voltage Measurement (spikes added)');
```
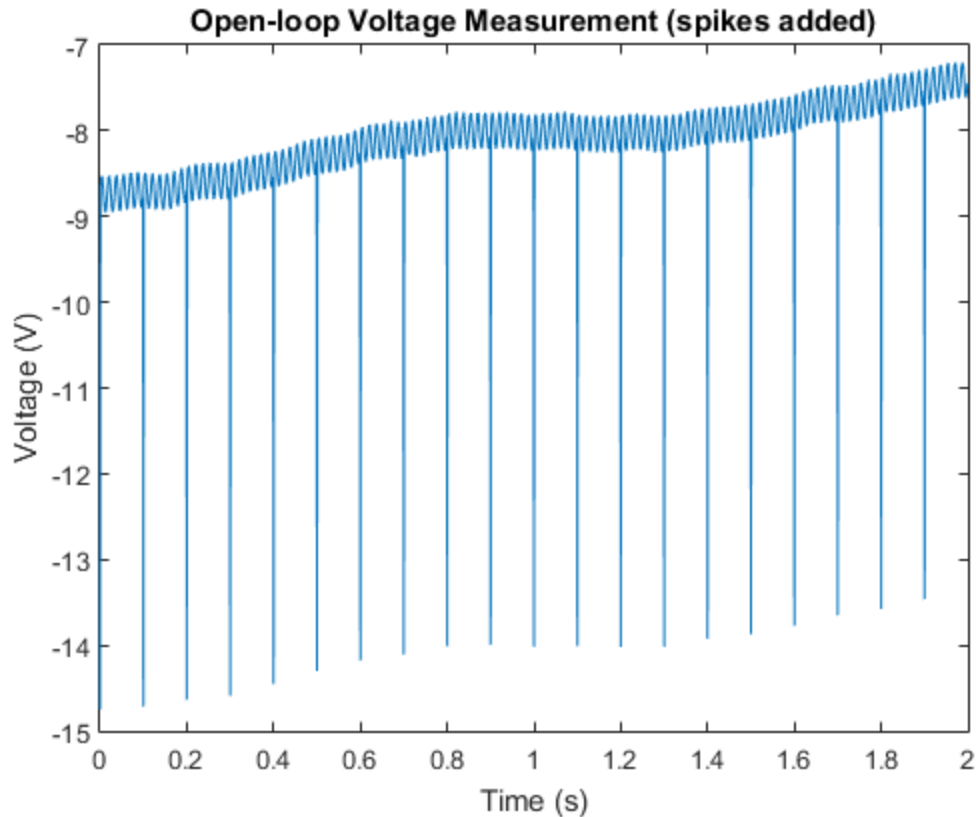

Let's see what the signal looks like after resampling and using the Savitzky-Golay filter.

```
vResamp = resample(noisyLoopVoltage, fsResamp, fs);
tResamp = (0:numel(vResamp)-1) / fsResamp;
vAvgResamp = sgolayfilt(vResamp,1,17);
plot(tResamp,vAvgResamp);
ylabel('Voltage (V)');
xlabel('Time (s)');
title('Open-loop Noisy Voltage Measurement (spikes added)');
legend('Moving average filter operating at 60 Hz','location','southeast');
```

Open-loop Noisy Voltage Measurement (spikes added)
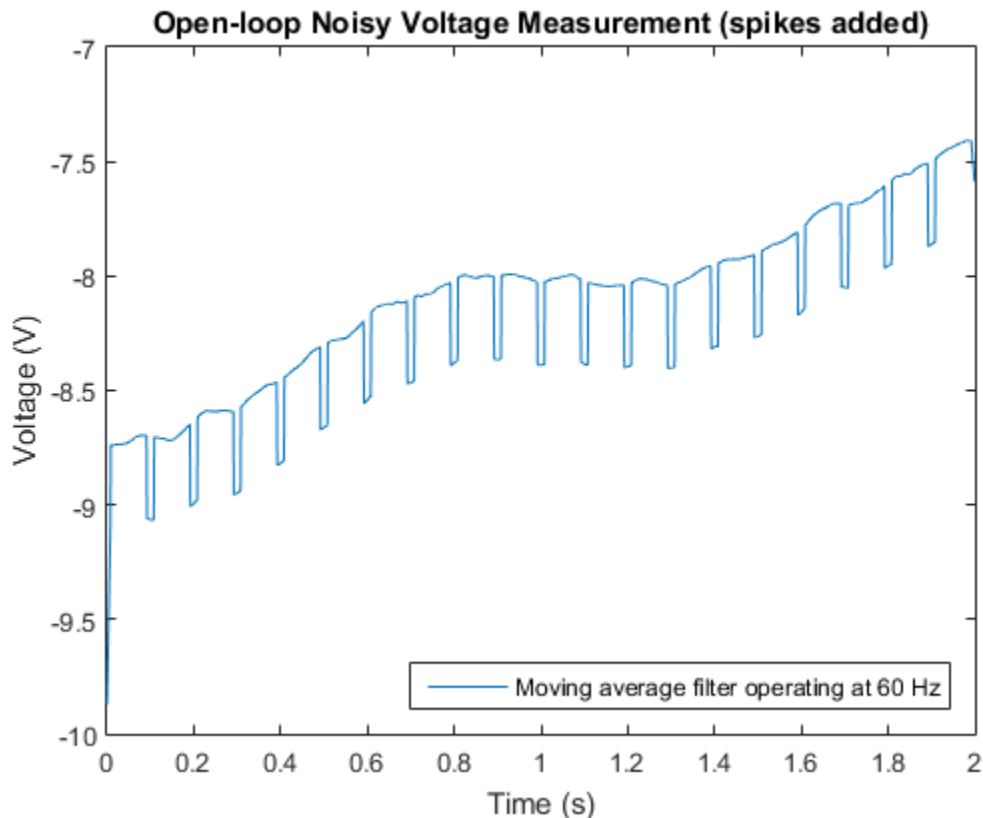
Note that resampling and applying our weighted moving average filter still removes the unwanted 60 Hz data and attenuates the spikes, however the spikes are spread out over the length of the filter.

A median filter replaces each element in the data with the median value over the length of the filter. We can use the filter to remove the spikes.

We will choose our filter order to be 3 (two samples more than the width of our spikes).

```
medfiltLoopVoltage = medfilt1(noisyLoopVoltage, 3);
vResamp = resample(medfiltLoopVoltage, fsResamp, fs);
tResamp = (0:numel(vResamp)-1) / fsResamp;
vAvgResamp = sgolayfilt(vResamp,1,17);
plot(tResamp,vAvgResamp);
```

```
ylabel('Voltage (V)');
xlabel('Time (s)');
title('Open-loop Noisy Voltage Measurement (spikes added)');
legend('Median and moving average filtered','location','southeast');
```
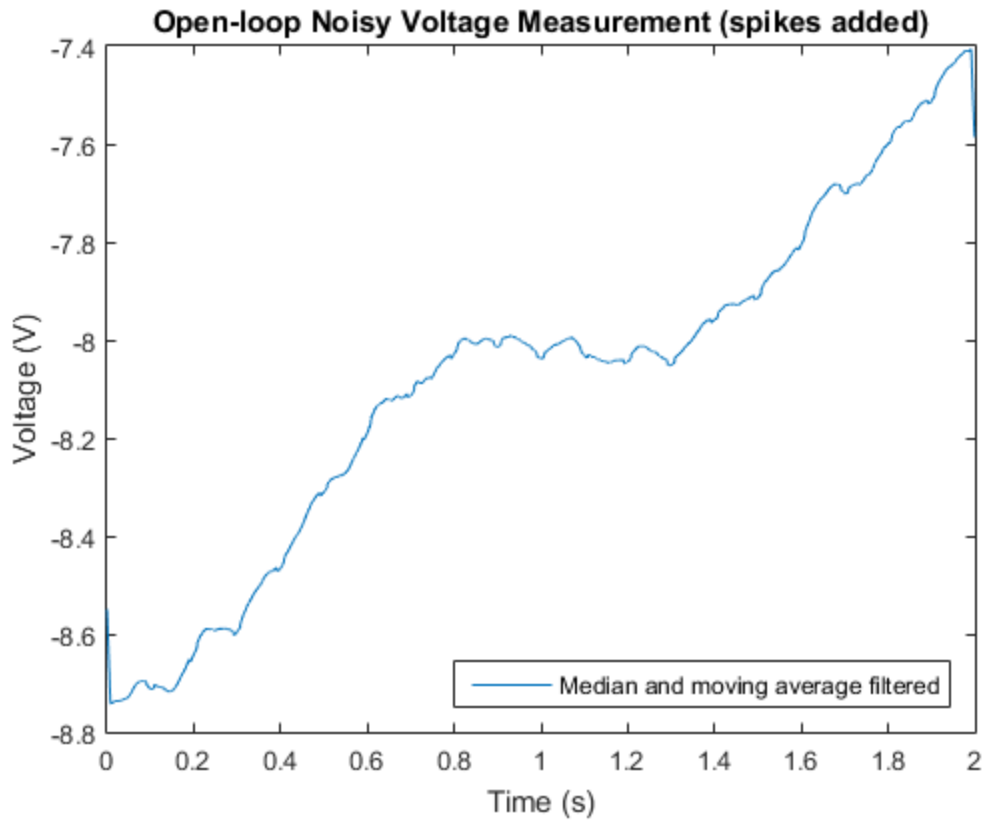

Further Reading

For more information on filtering and resampling see the Signal Processing Toolbox.

Reference: M. Kendall and A. Stuart, "The Advanced Theory of Statistics", Griffin, 1983, Vol 3.

Peak Analysis

This example shows how to perform basic peak analysis. It will help you answer questions such as: How do I find peaks in my signal? How do I measure distance between peaks? How do I measure the amplitude of peaks of a signal which is affected by a trend? How do I find peaks in a noisy signal? How do I find local minima?

Finding Maxima or Peaks

The Zurich sunspot relative number measures both the number and size of sunspots. Use the `findpeaks` function to find the locations and the value of the peaks.

```
load sunspot.dat
year=sunspot(:,1);
relNums=sunspot(:,2);
findpeaks(relNums,year);
xlabel('Year');
ylabel('Sunspot Number')
title('Find All Peaks');
```
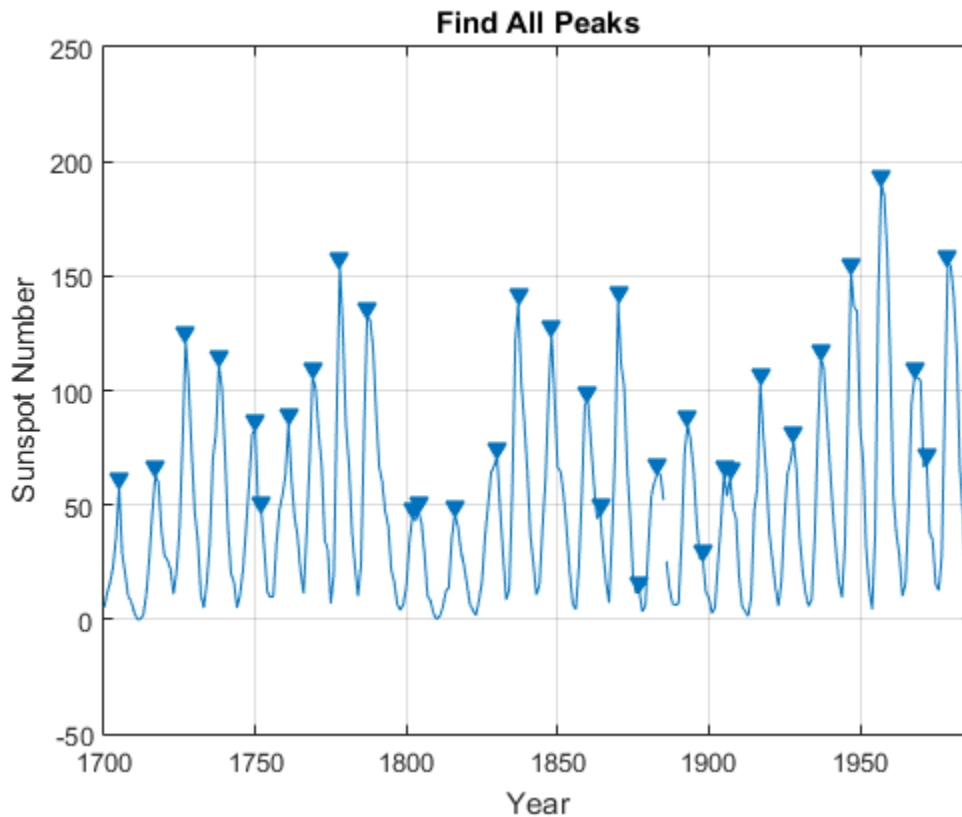
The above plot shows sunspot numbers tabulated over 300 years and labels the detected peaks. The next section shows how to measure distance between these peaks.

Measuring Distance Between Peaks

Peaks in the signal seem to appear at regular intervals. However, some of the peaks are very close to each other. The `MinPeakProminence` property can be used filter out these peaks. Consider peaks that drop off on both sides by at least 40 relative sunspot numbers before encountering a larger value.

```
findpeaks(relNums,year,'MinPeakProminence',40);
xlabel('Year');
ylabel('Sunspot Number')
title('Find Prominent Peaks');
```
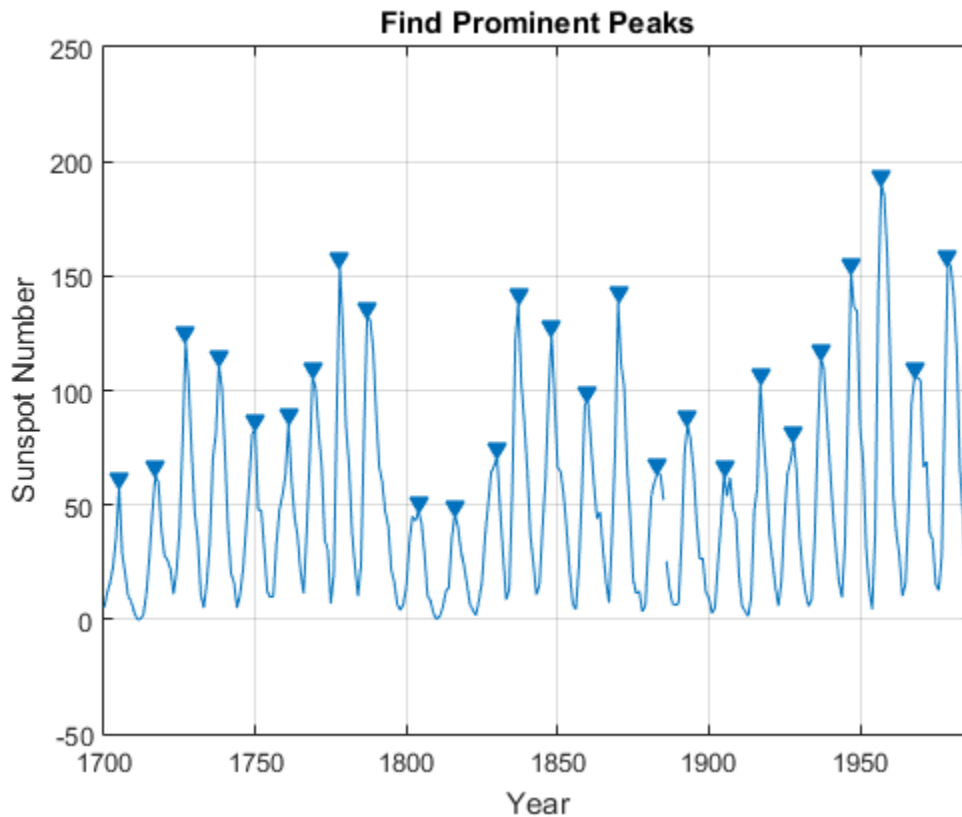
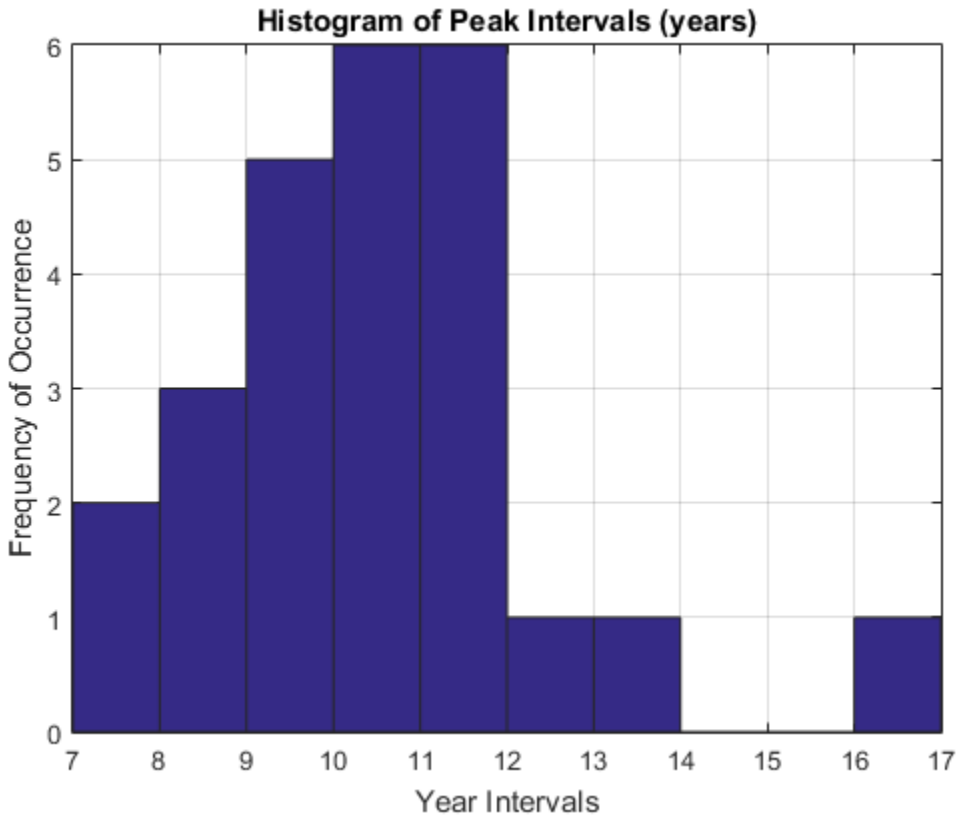The following histogram shows the distribution of peak intervals in years:

```
figure
[pks, locs] = findpeaks(relNums,year,'MinPeakProminence',40);
peakInterval = diff(locs);
hist(peakInterval);
grid on
xlabel('Year Intervals');
ylabel('Frequency of Occurrence')
title('Histogram of Peak Intervals (years)')

AverageDistance_Peaks = mean(diff(locs))
```

```
AverageDistance_Peaks =

    10.9600
```


The distribution shows that majority of peak intervals lie between 10 and 12 years indicating the signal has a cyclic nature. Also, the average interval of 10.96 years between the peaks matches the known cyclic sunspot activity of 11 years.

Finding Peaks in Clipped or Saturated Signals

You may want to consider flat peaks as peaks or exclude them. In the latter case, a minimum excursion which is defined as the amplitude difference between a peak and its immediate neighbors is specified using the `threshold` property.

```matlab
load clippedpeaks.mat

figure

% Show all peaks in the first plot
ax(1) = subplot(2,1,1);
findpeaks(saturatedData);
xlabel('Samples')
ylabel('Amplitude')
title('Detecting Saturated Peaks')

% Specify a minimum excursion in the second plot
ax(2) = subplot(2,1,2);
findpeaks(saturatedData,'threshold',5)
xlabel('Samples');
ylabel('Amplitude')
title('Filtering Out Saturated Peaks')

% link and zoom in to show the changes
linkaxes(ax(1:2),'xy');
axis(ax,[50 70 0 250])
```
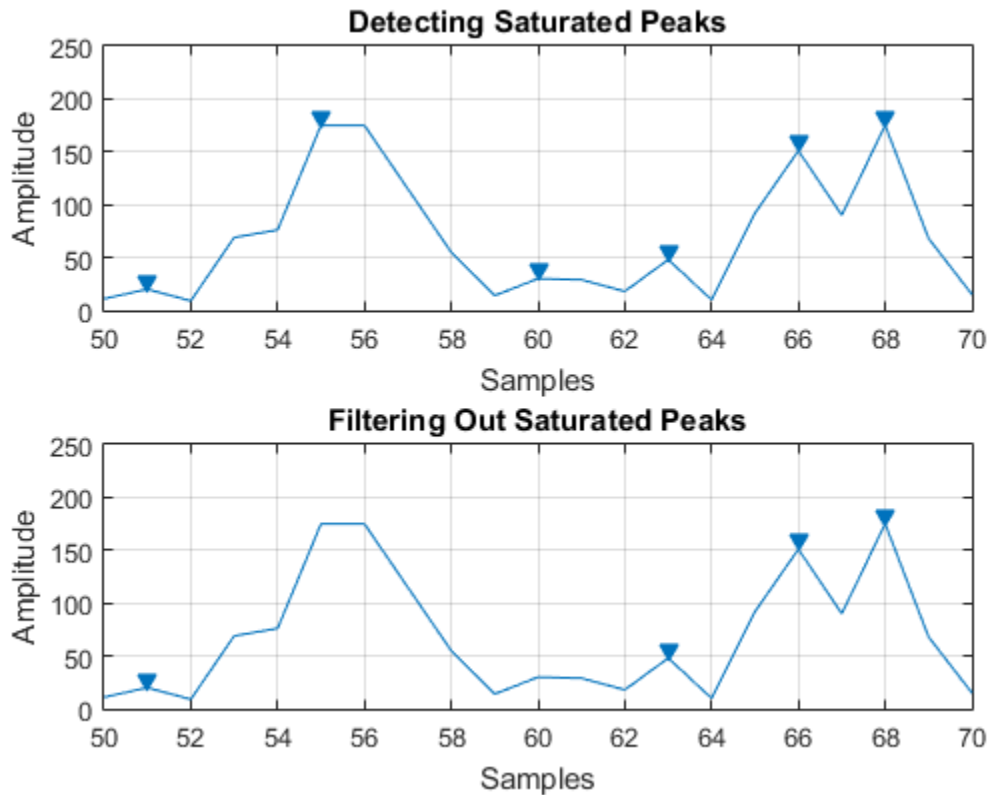
The first subplot shows, that in case of a flat peak, the rising edge is detected as the peak. The second subplot shows that specifying a threshold can help to reject flat peaks.

Measuring Amplitudes of Peaks

This example shows peak analysis in an ECG (Electro-cardiogram) signal. ECG is a measure of electrical activity of the heart over time. The signal is measured by electrodes attached to the skin and is sensitive to disturbances such as power source interference and noises due to movement artifacts.

```
load noisyecg.mat
t = 1:length(noisyECG_withTrend);

figure
```

```
plot(t,noisyECG_withTrend)
title('Signal with a Trend')
xlabel('Samples');
ylabel('Voltage(mV)')
legend('Noisy ECG Signal')
grid on
```
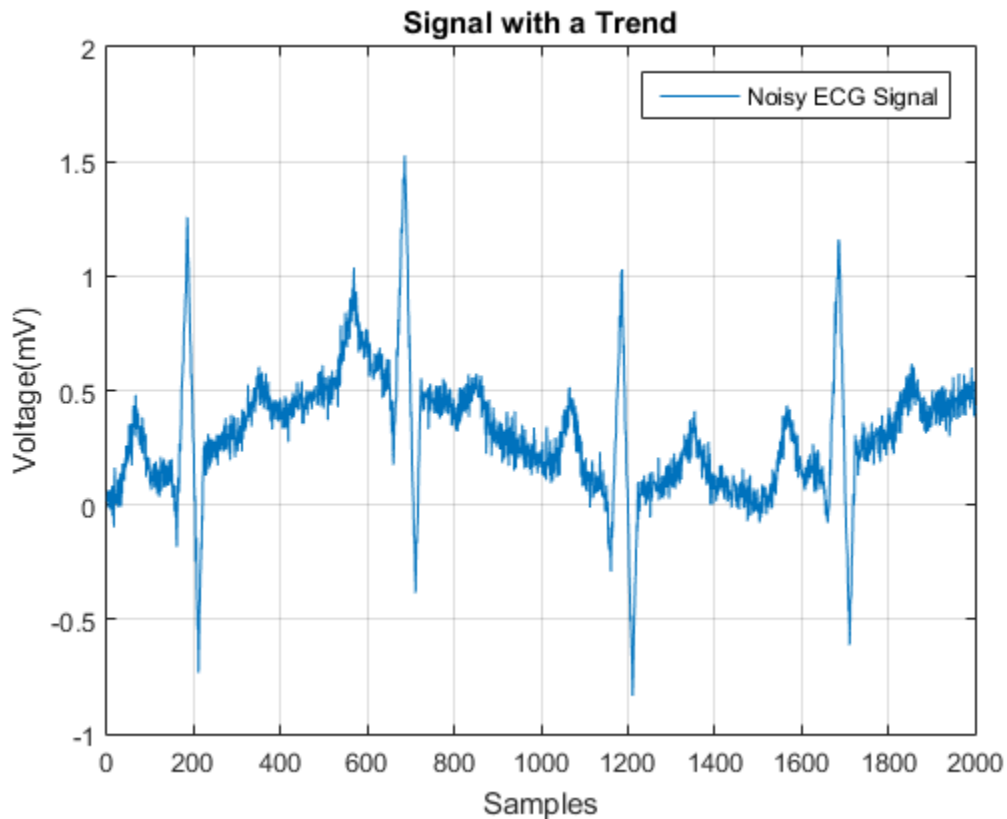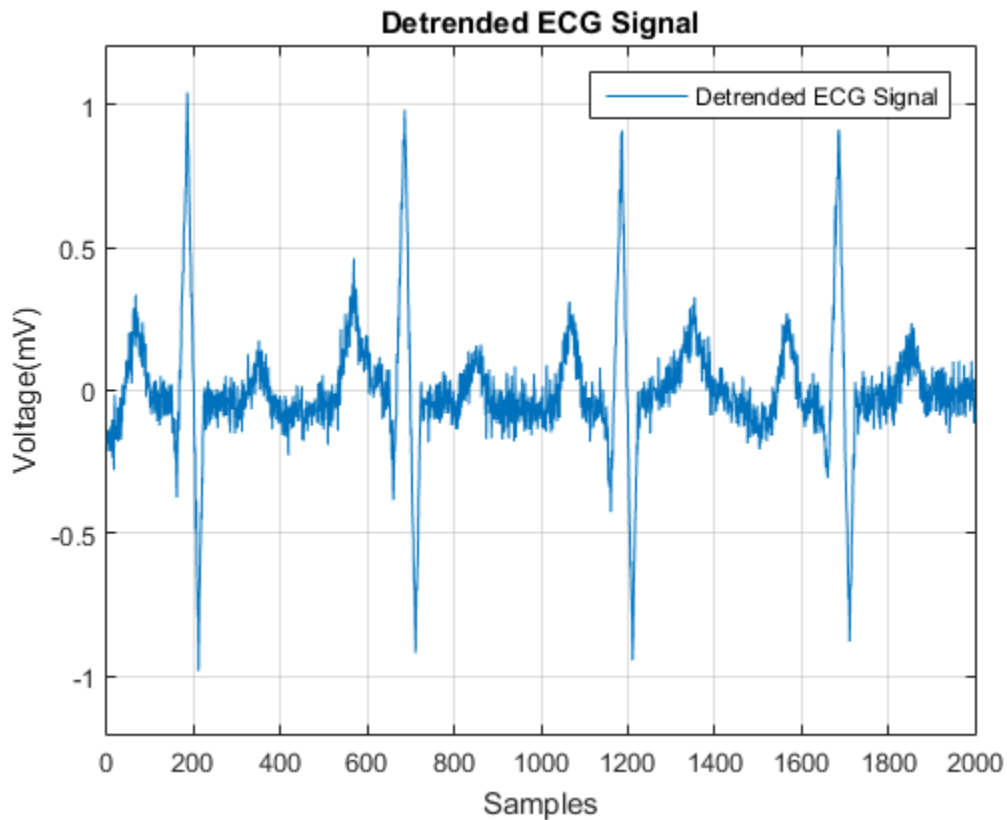

Detrending Data

The above signal shows a baseline shift and therefore does not represent the true amplitude. In order to remove the trend, fit a low order polynomial to the signal and use the polynomial to detrend it.

```
[p,s,mu] = polyfit((1:numel(noisyECG_withTrend))',noisyECG_withTrend,6);
```

```
f_y = polyval(p,(1:numel(noisyECG_withTrend))',[],mu);

ECG_data = noisyECG_withTrend - f_y;          % Detrend data

figure
plot(t,ECG_data); grid on
ax = axis; axis([ax(1:2) -1.2 1.2])
title('Detrended ECG Signal')
xlabel('Samples'); ylabel('Voltage(mV)')
legend('Detrended ECG Signal')
```


After detrending, find the QRS-complex which is the most prominent repeating peak in the ECG signal. The QRS-complex corresponds to the depolarization of the right and

left ventricles of the human heart. It can be used to determine a patient's cardiac rate or predict abnormalities in heart function. The following figure shows the shape of the QRS-complex in an ECG signal.
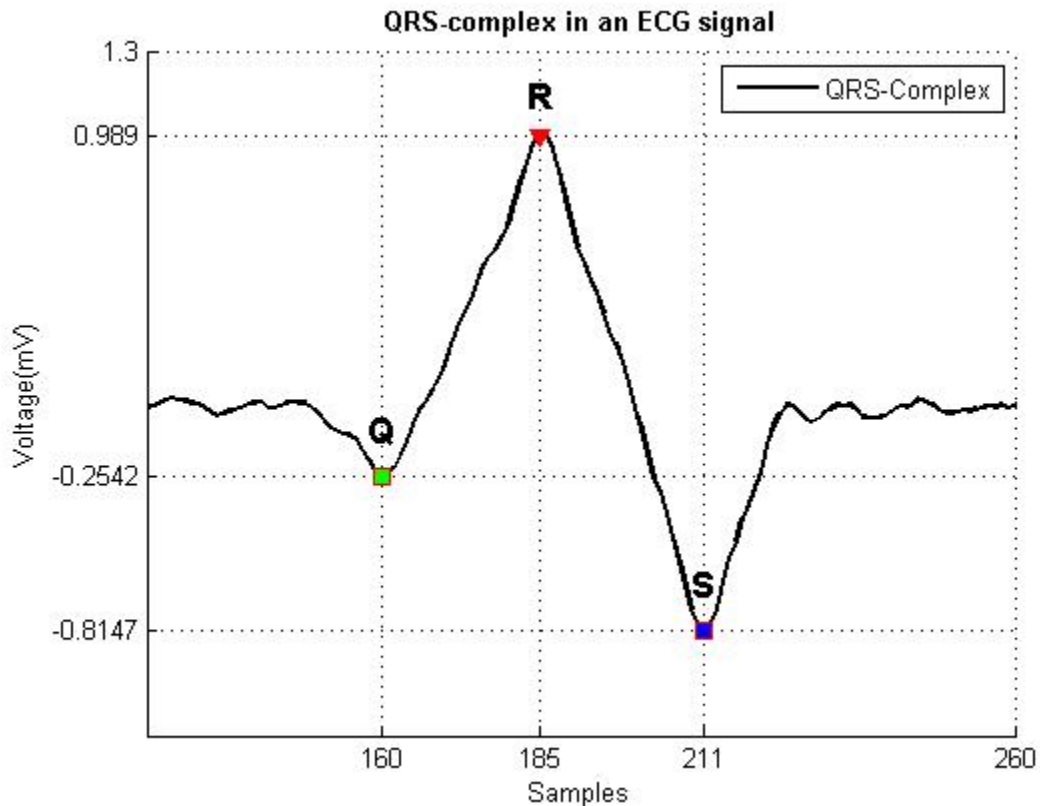
Thresholding to Find Peaks of Interest

The QRS-complex consists of three major components: **Q-wave, R-wave, S-wave**. The R-waves can be detected by thresholding peaks above 0.5mV. Notice that the R-waves are separated by more than 200 samples. Use this information to remove unwanted peaks by specifying a 'MinPeakDistance'.

```
[~,locs_Rwave] = findpeaks(ECG_data,'MinPeakHeight',0.5,...
                                    'MinPeakDistance',200);
```

For detection of the S-waves, find the local minima in the signal and apply thresholds appropriately.

Finding Local Minima in Signal

Local minima can be detected by finding peaks on an inverted version of the original signal.

```
ECG_inverted = -ECG_data;
[~,locs_Swave] = findpeaks(ECG_inverted,'MinPeakHeight',0.5,...
                                        'MinPeakDistance',200);
```

The following plot shows the R-waves and S-waves detected in the signal.

```
figure
hold on
plot(t,ECG_data);
plot(locs_Rwave,ECG_data(locs_Rwave),'rv','MarkerFaceColor','r');
plot(locs_Swave,ECG_data(locs_Swave),'rs','MarkerFaceColor','b');
axis([0 1850 -1.1 1.1]); grid on;
legend('ECG Signal','R-waves','S-waves');
xlabel('Samples'); ylabel('Voltage(mV)')
title('R-wave and S-wave in Noisy ECG Signal')
```
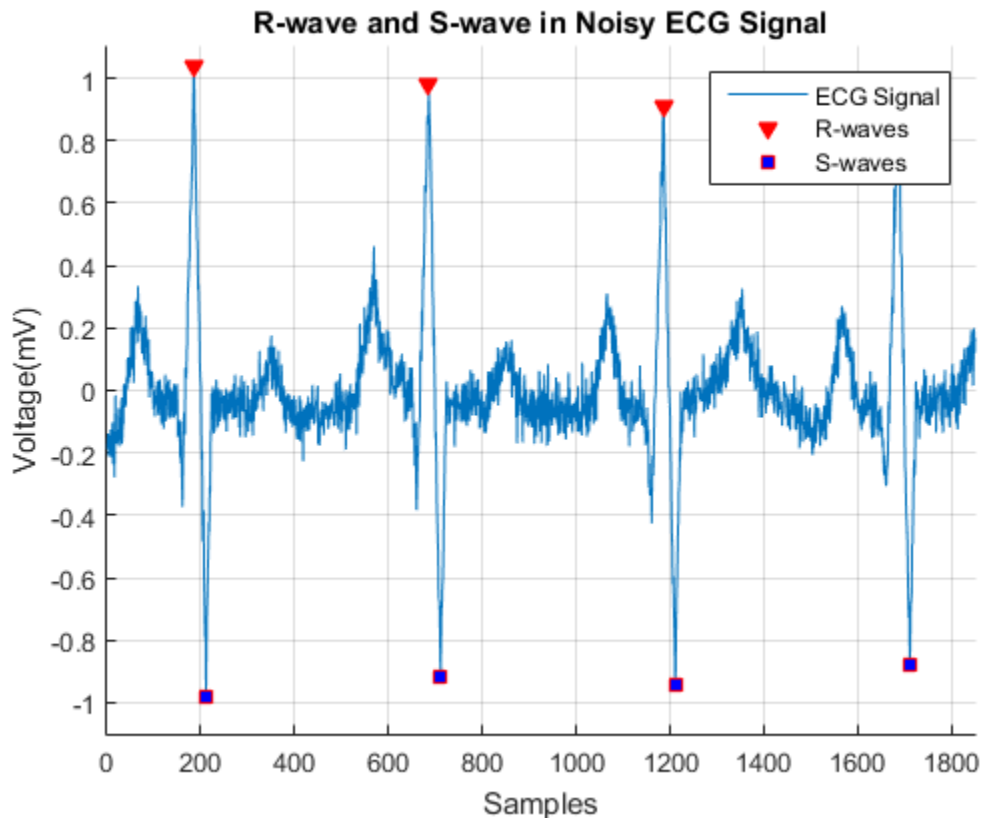
Next, we try and determine the locations of the Q-waves. Thresholding the peaks to locate the Q-waves results in detection of unwanted peaks as the Q-waves are buried in noise. We filter the signal first and then find the peaks. Savitzky-Golay filtering is used to remove noise in the signal.

```
smoothECG = sgolayfilt(ECG_data,7,21);

figure
plot(t,ECG_data,'b',t,smoothECG,'r'); grid on
axis tight;
xlabel('Samples'); ylabel('Voltage(mV)');
legend('Noisy ECG Signal','Filtered Signal')
title('Filtering Noisy ECG Signal')
```

Filtering Noisy ECG Signal
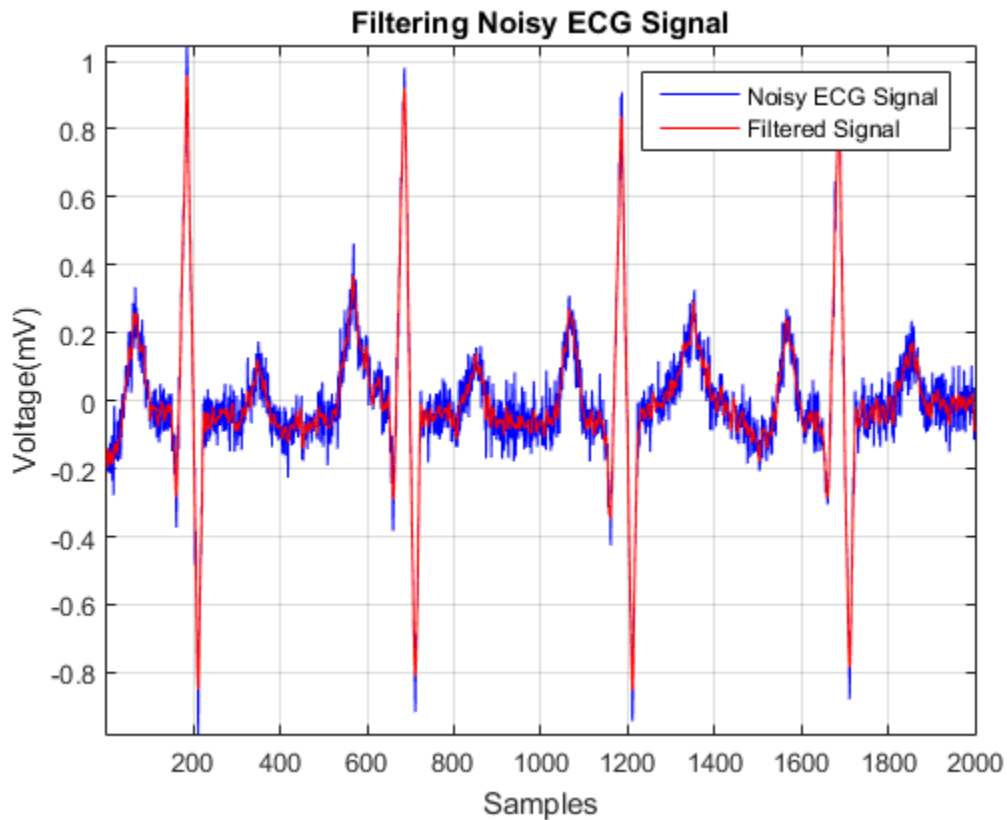
We perform peak detection on the smooth signal and use logical indexing to find the locations of the Q-waves.

```
[~,min_locs] = findpeaks(-smoothECG,'MinPeakDistance',40);

% Peaks between -0.2mV and -0.5mV
locs_Qwave = min_locs(smoothECG(min_locs)>-0.5 & smoothECG(min_locs)<-0.2);

figure
hold on
plot(t,smoothECG);
plot(locs_Qwave,smoothECG(locs_Qwave),'rs','MarkerFaceColor','g');
plot(locs_Rwave,smoothECG(locs_Rwave),'rv','MarkerFaceColor','r');
plot(locs_Swave,smoothECG(locs_Swave),'rs','MarkerFaceColor','b');
```
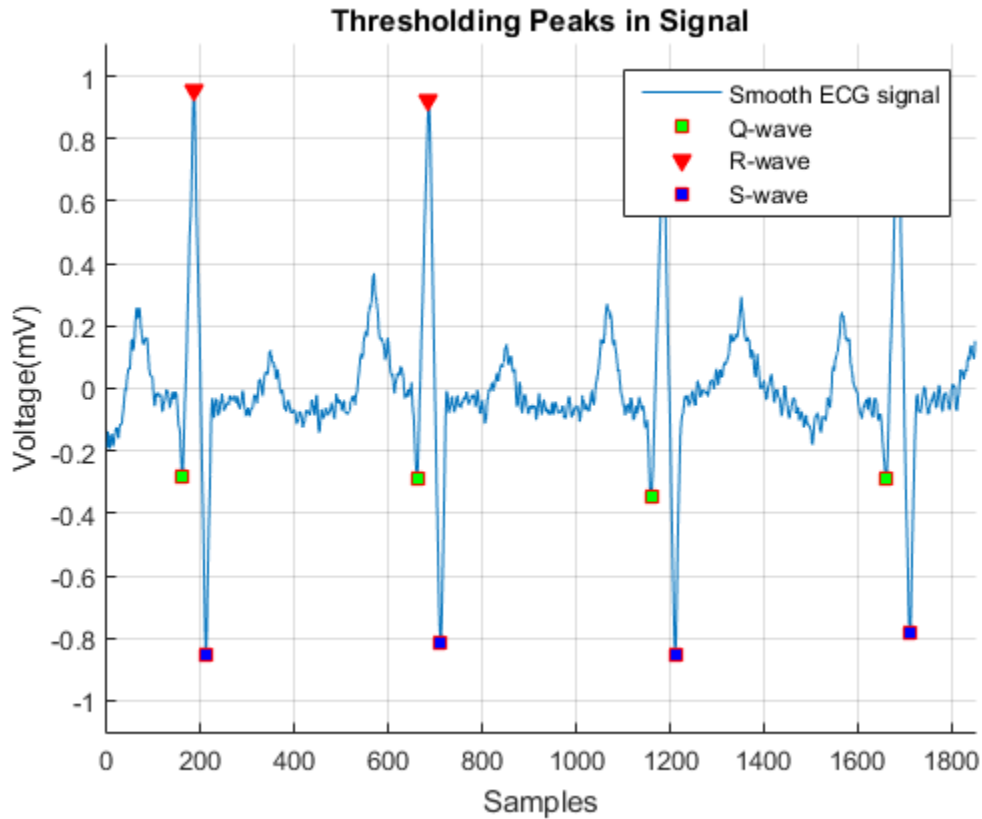
```
grid on
title('Thresholding Peaks in Signal')
xlabel('Samples'); ylabel('Voltage(mV)')
ax = axis; axis([0 1850 -1.1 1.1])
legend('Smooth ECG signal','Q-wave','R-wave','S-wave');
```


The above figure shows that the QRS-complex successfully detected in the noisy ECG signal.

Error Between Noisy and Smooth Signal

Notice the average difference between the QRS-complex in the raw and the detrended filtered signal.

```
% Values of the Extrema
[val_Qwave, val_Rwave, val_Swave] = deal(smoothECG(locs_Qwave), smoothECG(locs_Rwave),

meanError_Qwave = mean((noisyECG_withTrend(locs_Qwave) - val_Qwave))
meanError_Rwave = mean((noisyECG_withTrend(locs_Rwave) - val_Rwave))
meanError_Swave = mean((noisyECG_withTrend(locs_Swave) - val_Swave))


meanError_Qwave =

    0.2771


meanError_Rwave =

    0.3476


meanError_Swave =

    0.1844
```

This demonstrates that it is essential to detrend a noisy signal for efficient peak analysis.

Peak Properties

Some important peak properties involve rise time, fall time, rise level, and fall level. These properties are computed for each of the QRS-complexes in the ECG signal. The average values for these properties are displayed on the figure below.

```
avg_riseTime = mean(locs_Rwave-locs_Qwave); % Average Rise time
avg_fallTime = mean(locs_Swave-locs_Rwave); % Average Fall time
avg_riseLevel = mean(val_Rwave-val_Qwave);  % Average Rise Level
avg_fallLevel = mean(val_Rwave-val_Swave);  % Average Fall Level

helperPeakAnalysisPlot(t,smoothECG,...
                    locs_Qwave,locs_Rwave,locs_Swave,...
                    val_Qwave,val_Rwave,val_Swave,...
                    avg_riseTime,avg_fallTime,...
                    avg_riseLevel,avg_fallLevel)
```

16-113
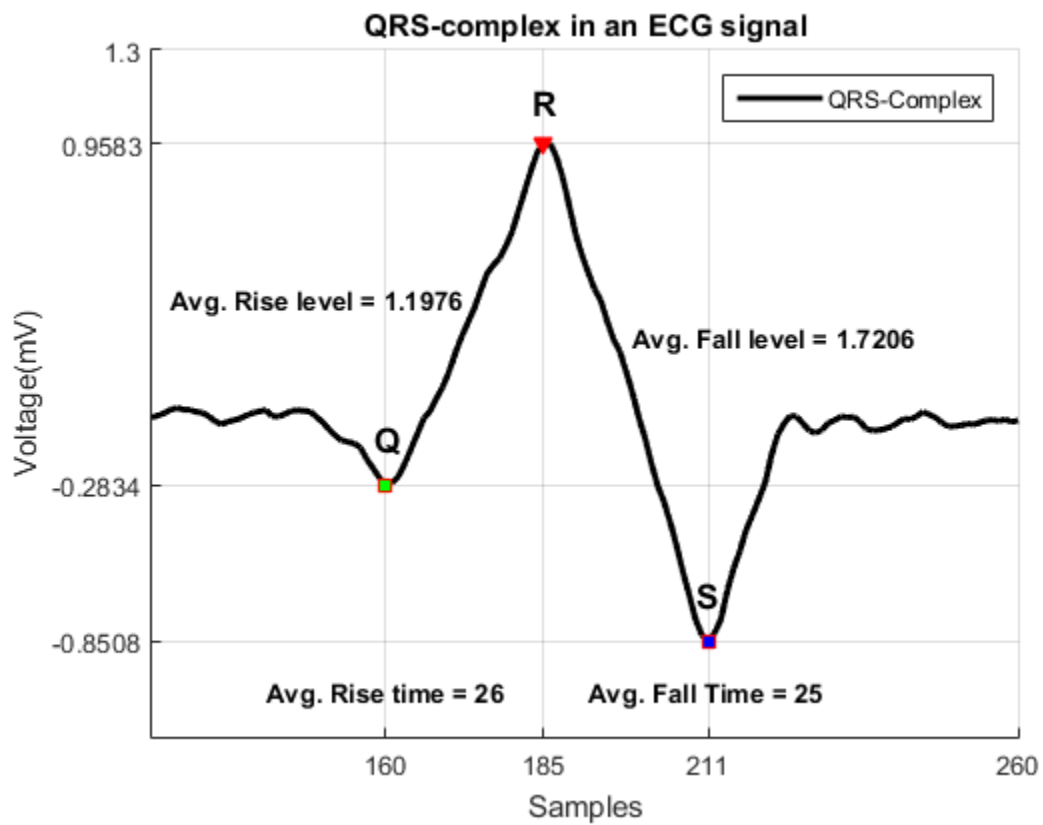
Prominence

The *prominence* of a peak measures how much the peak stands out due to its intrinsic height and its location relative to other peaks. A low isolated peak can be more prominent than one that is higher but is an otherwise unremarkable member of a tall range.

To measure the prominence of a peak:

1 Place a marker on the peak.

2 Extend a horizontal line from the peak to the left and right until the line does one of the following:

  • Crosses the signal because there is a higher peak

  • Reaches the left or right end of the signal

3 Find the minimum of the signal in each of the two intervals defined in Step 2. This point is either a valley or one of the signal endpoints.

4 The higher of the two interval minima specifies the reference level. The height of the peak above this level is its prominence.

`findpeaks` makes no assumption about the behavior of the signal beyond its endpoints, whatever their height. This is reflected in Steps 2 and 4 and often affects the value of the reference level. Consider for example the peaks of this signal:
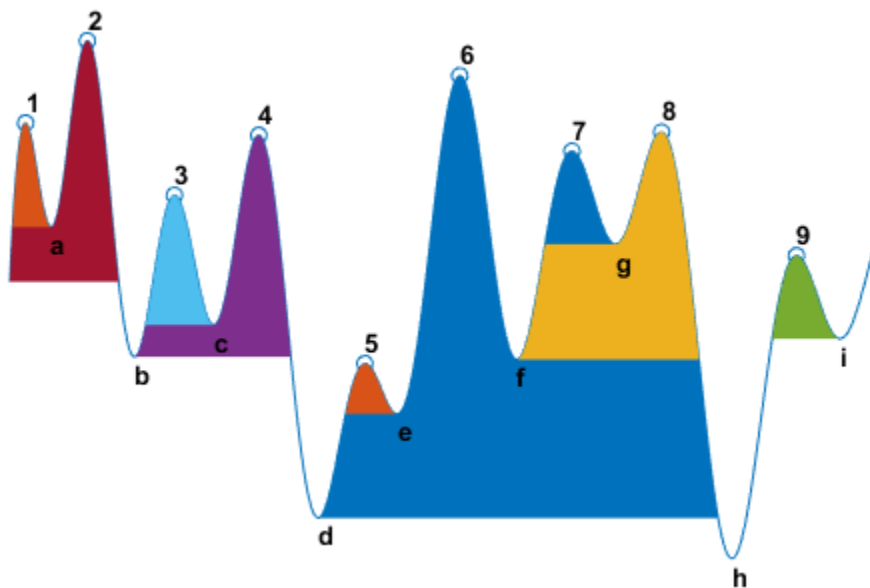
| Peak Number	Left Interval Lies Between Peak and	Right Interval Lies Between Peak and	Lowest Point on the Left Interval	Lowest Point on the Right Interval	Reference Level (Highest Minimum)
1	Left end	Crossing due to peak **2**	Left endpoint	**a**	**a**
2	Left end	Right end	Left endpoint	**h**	Left endpoint
3	Crossing due to peak **2**	Crossing due to peak **4**	**b**	**c**	**c**
4	Crossing due to peak **2**	Crossing due to peak **6**	**b**	**d**	**b**
5	Crossing due to peak **4**	Crossing due to peak **6**	**d**	**e**	**e**
6	Crossing due to peak **2**	Right end	**d**	**h**	**d**
7	Crossing due to peak **6**	Crossing due to peak **8**	**f**	**g**	**g**

Peak Number	Left Interval Lies Between Peak and	Right Interval Lies Between Peak and	Lowest Point on the Left Interval	Lowest Point on the Right Interval	Reference Level (Highest Minimum)
8	Crossing due to peak **6**	Right end	`f`	`h`	`f`
9	Crossing due to peak **8**	Crossing due to right endpoint	`h`	`i`	`i`

Determine Peak Widths

Create a signal that consists of a sum of bell curves. Specify the location, height, and width of each curve.
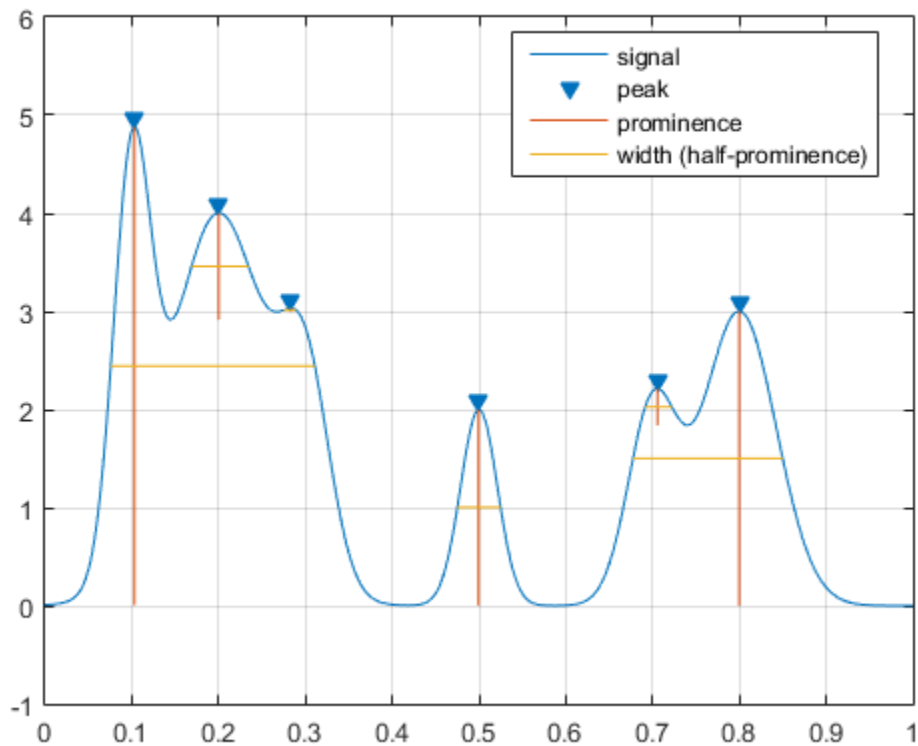
```
x = linspace(0,1,1000);

Pos = [1 2 3 5 7 8]/10;
Hgt = [4 4 2 2 2 3];
Wdt = [3 8 4 3 4 6]/100;

for n = 1:length(Pos)
    Gauss(n,:) =  Hgt(n)*exp(-((x - Pos(n))/Wdt(n)).^2);
end

PeakSig = sum(Gauss);
```

Plot the individual curves and their sum.
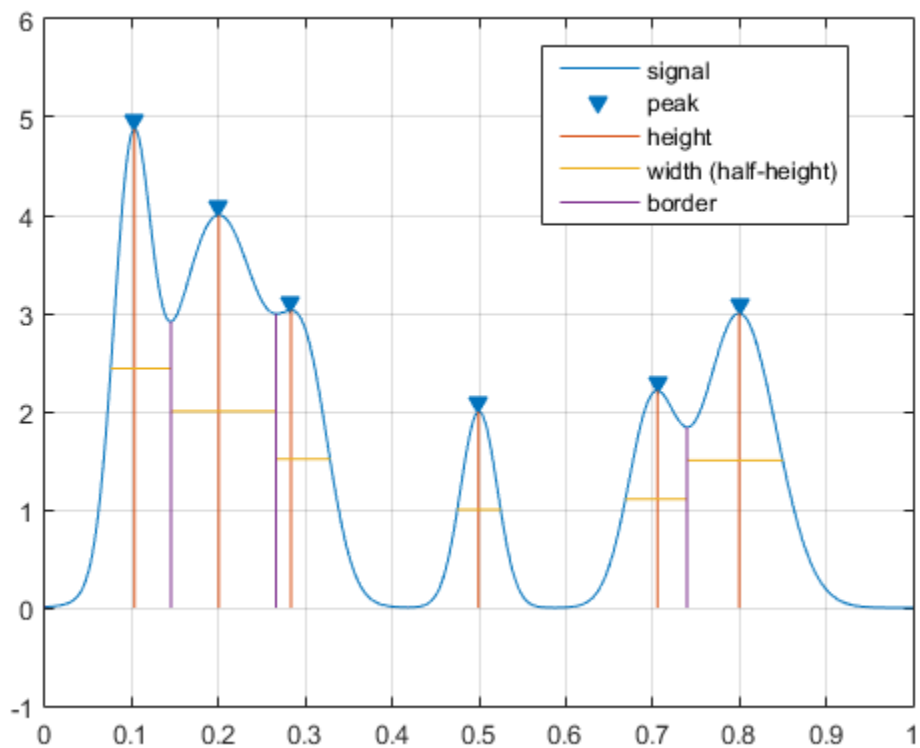
```
plot(x,Gauss,'--',x,PeakSig)
```
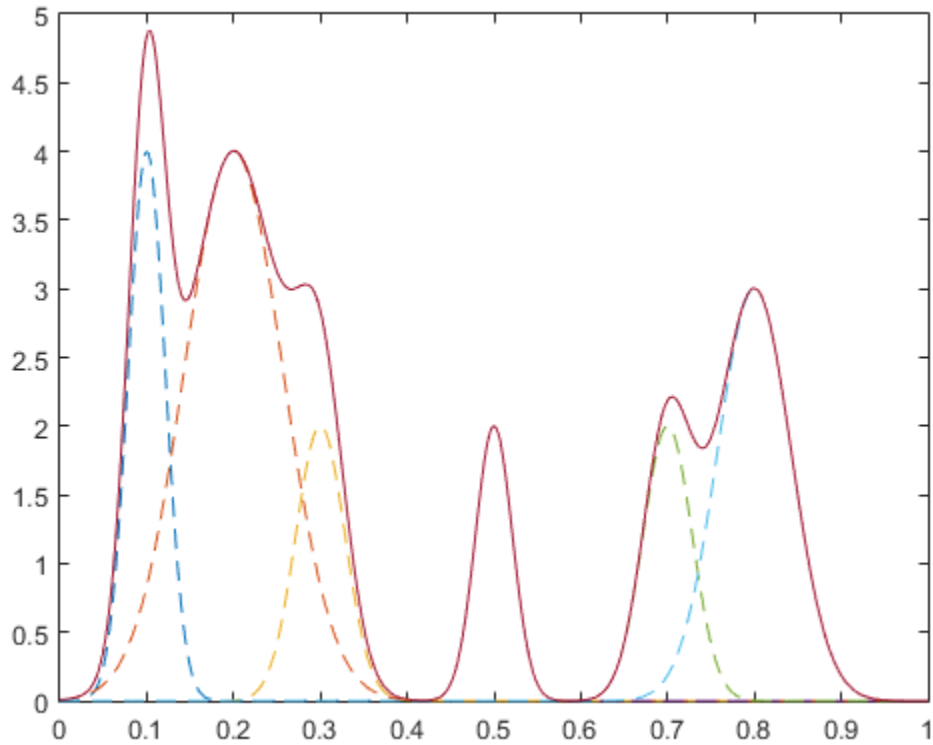
Measure the widths of the peaks using the half prominence as reference.

```
findpeaks(PeakSig,x,'Annotate','extents')
```

Measure the widths again, this time using the half height as reference.

```
findpeaks(PeakSig,x,'Annotate','extents','WidthReference','halfheight')
```

Spectrum Object to Function Replacement

Autoregressive PSD Object to Function Replacement Syntax

The AR PSD object syntax will be removed in the future. The following table gives the equivalent recommended function syntax. The table uses `spectrum.burg` and `pburg` as examples, but the object-to-function replacement syntaxes are valid for all the AR spectral estimators with the appropriate substitution: `spectrum.burg` to `pburg`, `spectrum.cov` to `pcov`, `spectrum.mcov` to `pmcov`, and `spectrum.yulear` to `pyulear`. In each example, `x` is the input signal.

Deprecated Syntax	Replacement Syntax
`hBurg = spectrum.burg;` `psd(hBurg,x);`	`pburg(x,4);`
`hBurg = spectrum.burg(order);` `psd(hBurg,x);`	`pburg(x,order);`
`hBurg = spectrum.burg(order);` `psd(hBurg,x,'NFFT',nfft);`	`pburg(x,order,nfft);`
`hBurg = spectrum.burg(order);` `psd(hBurg,x,'Fs',fs);`	`pburg(x,order,[],fs);`
`hBurg = spectrum.burg(order);` `psd(hBurg,x,'NFFT',nfft,'Fs',fs);`	`pburg(x,order,nfft,fs);`

`hBurg = spectrum.burg(order);`
`psd(hBurg, x,...,'FreqPoints','User Define`
`'FrequencyVector',w);`
	`pburg(x,order,w);`

`hBurg = spectrum.burg(order);`
`psd(hBurg,x,'FreqPoints','User Defined',..`
`'FrequencyVector',f,'Fs',fs);`
	`pburg(x,order,f,fs);`

`hBurg = spectrum.burg` `psd(...,'SpectrumType','TwoSided');`	`pburg(...,'twosided');`
`hBurg = spectrum.burg;` `psd(...,'CenterDC',true);`	`pburg(x,...,'centered');`
`hBurg = spectrum.burg;` `psd(...,'ConfLevel',p);`	`pburg(x,...,'ConfidenceLevel',p);`

`hBurg = spectrum.burg;`
`hPSD = psd(...);`
`Pxx = hPSD.Data;`
`F = hPSD.Frequencies;`
	`[Pxx,F] = pburg(...);`

`hBurg = spectrum.burg;` `hPSD = psd(...,'ConfLevel',p);`	`[Pxx,F,Pxxc] = pburg(...);`

Deprecated Syntax	Replacement Syntax

`Pxx = hPSD.Data;`
`F = hPSD.Frequencies;`
`Pxxc = hPSD.ConfInterval;`	

Multitaper PSD Object to Function Replacement Syntax

The `spectrum.mtm` object syntax will be removed in the future. The following table gives the equivalent recommended function syntax for `pmtm`. In each example, `x` is the input signal.

Deprecated Syntax	Recommended Syntax
`hMTM = spectrum.mtm;` `psd(hMTM,x);`	`pmtm(x,4);`
`hMTM = spectrum.mtm(NW);` `psd(hMTM,x);`	`pmtm(x,NW);`

`[E,V] = dpss(length(x),NW);`
`hMTM = spectrum.mtm(E,V);`
`psd(hMTM,x);`
	`[E,V] = dpss(length(x),NW);`
	`pmtm(x,E,V);`

`hMTM = spectrum.mtm(NW);` `psd(hMTM,x,'Fs',fs);`	`pmtm(x,NW,fs);`
`hMTM = spectrum.mtm(E,V);` `psd(hMTM,x,'Fs',fs);`	`pmtm(x,E,V,fs);`
`hMTM = spectrum.mtm(NW);` `psd(hMTM,x,'Fs',fs,'NFFT',nfft);`	`pmtm(x,NW,nfft,fs);`
`hMTM = spectrum.mtm(E,V);` `psd(hMTM,x,'Fs',fs,'NFFT',nfft);`	`pmtm(x,E,V,nfft,fs);`

`hMTM = spectrum.mtm(NW);`
`psd(hMTM,x,'FreqPoints','User Defined',...`
`'FrequencyVector',w);`
	`pmtm(x,NW,w);`

`hMTM = spectrum.mtm(E,V);`
`psd(hMTM,x,'FreqPoints','User Defined',...`
`'FrequencyVector',w);`
	`pmtm(x,E,V,w);`

`hMTM = spectrum.mtm(NW);`
`psd(hMTM,x,'FreqPoints','User Defined',...`
`'FrequencyVector',f,'Fs',fs);`
	`pmtm(x,E,V,f,fs);`

`hMTM = spectrum.mtm(E,V);`
`psd(hMTM,x,'FreqPoints','User Defined',...`
`'FrequencyVector',f,'Fs',fs);`
	`pmtm(x,E,V,f,fs);`

`hMTM = spectrum.mtm(...,'Adaptive');` `psd(hMTM,...);`	`pmtm(...,'adapt');`
`hMTM = spectrum.mtm(...,'Eigenvalue');` `psd(hMTM,...);`	`pmtm(...,'eigen');`

Deprecated Syntax	Recommended Syntax
`hMTM = spectrum.mtm(...,'Unity');` `psd(hMTM,...);`	`pmtm(...,'unity');`
`hMTM = spectrum.mtm(...);` `psd(hMTM,...,'SpectrumType','twosided');`	`pmtm(...,'twosided');`

`hMTM = spectrum.mtm(...);`
`psd(hMTM,...,'SpectrumType','twosided',...`
`'CenterDC',true);`
	`pmtm(...,'centered');`

`hMTM = spectrum.mtm(...);` `psd(hMTM,...,'ConfLevel',p);`	`pmtm(...,'ConfidenceLevel',p);`

`hMTM = spectrum.mtm(...);`
`hPSD = psd(hMTM,...);`
`Pxx = hPSD.Data;`
`F = hPSD.Frequencies;`
	`[Pxx,F] = pmtm(...);`

`hMTM = spectrum.mtm(...);`
`hPSD = psd(hMTM,x,'ConfLevel',p);`
`Pxx = hPSD.Data;`
`F = hPSD.Frequencies;`
`Pxxc = hPSD.ConfInterval;`
	`[Pxx,F,Pxxc] = pmtm(x,'ConfidenceLevel',p);`

Periodogram PSD Object to Function Replacement Syntax

The `spectrum.periodogram` object syntax will be removed in the future. The following table gives the equivalent recommended function syntax for `periodogram`. In the modified periodogram, you use a window other than the default rectangular window. To illustrate modified periodogram syntaxes, the table uses a specific window. In each example, `x` is the input signal.

Deprecated Syntax	Replacement Syntax
`h = spectrum.periodogram;` `psd(h,x);`	`periodogram(x);`

Modified periodogram with window function:
 	
`h = spectrum.periodogram('hamming');`
`psd(h,x);`
	`win = hamming(length(x));`
	`periodogram(x,win);`

Window function and optional input arguments to window function:
 	
`h = spectrum.periodogram({'Hamming','perio`
`psd(h,x);`
	`win = hamming(length(x),'periodic');`
	`periodogram(x,win);`

Taylor window and multiple optional input arguments:
 	
`nbar = 4;`
`sll = 30;`
`h = spectrum.periodogram({'Taylor',nbar,sl`
`psd(h,x,'Fs',fs,'centerdc',true);`
	`nbar = 4;`
	`sll = -30;`
	`win = taylorwin(length(x),nbar,sll);`
	`periodogram(x,win,[],fs,'centered');`

`h = spectrum.periodogram(...);` `psd(h,x,'NFFT',nfft);`	`win = ...` `periodogram(x,win,nfft);`
`h = spectrum.periodogram(...);` `psd(h,x,'Fs',fs);`	`win = ...` `periodogram(x,win,[],fs);`
`h = spectrum.periodogram(...);` `psd(h,x,'NFFT',nfft,'Fs',fs);`	`win = ...` `periodogram(x,win,nfft,fs);`

`h = spectrum.periodogram(...);`
`psd(h,x,...,'FreqPoints','User Defined',..`
`'FrequencyVector',w);`
	`win = ...`
	`periodogram(x,win,w);`

`h = spectrum.periodogram(...);`
`psd(h,x,'FreqPoints','User Defined',...`
`'FrequencyVector',f,'Fs',fs);`
	`win = ...`
	`periodogram(x,win,f,fs);`

Deprecated Syntax	Replacement Syntax

Two-sided spectrum of a real signal:
 	
`h = spectrum.periodogram(...);`
`psd(h,x,...,'SpectrumType','TwoSided');`
	`win = ...`
	`periodogram(x,win,...,'twosided');`

Two-sided spectrum with DC (0 frequency) in the center:
 	
`h = spectrum.periodogram(...);`
`psd(h,x,...,'CenterDC',true);`
	`win = ...`
	`periodogram(x,win,...,'centered');`

`h = spectrum.periodogram(...);` `psd(h,x,...,'ConfLevel',p);`	`win = ...` `periodogram(x,win,...,'ConfidenceLevel',p);`

`h = spectrum.periodogram(...);`
`hPSD = psd(h,x,...);`
`Pxx = hPSD.Data;`
`F = hPSD.Frequencies;`
	`win = ...`
	`[Pxx,F] = periodogram(x,win,...);`

`h = spectrum.periodogram(...);`
`hPSD = psd(h,x,...,'ConfLevel',p);`
`Pxx = hPSD.Data;`
`F = hPSD.Frequencies;`
`Pxxc = hPSD.ConfInterval;`
	`win = ...`
	`[Pxx,F,Pxxc] = periodogram(x,win,...);`

Welch PSD Object to Function Replacement Syntax

The `spectrum.welch` object syntax will be removed in the future. The following table gives the equivalent recommended function syntax for `pwelch`. To illustrate modified periodogram syntaxes, the table uses a specific window. In each example, `x` is the input signal.

Deprecated Syntax	Replacement Syntax
`h = spectrum.welch;` `psd(h,x);`	`pwelch(x);`
`h = spectrum.welch('Gaussian');` `psd(h,x);`	`win = gausswin(64);` `pwelch(x,win);`

Welch estimate with window function and optional input arguments:
 	
`h = spectrum.welch({'Hamming','periodic'})`
`psd(h,x);`
	`win = hamming(64,'periodic');`
	`pwelch(x,win);`

Taylor window and multiple optional input arguments:
 	
`nbar = 4;`
`sll = 30;`
`h = spectrum.welch({'Taylor', nbar, sll});`
`psd(h,x);`
	`nbar = 4;`
	`sll = -30;`
	`win = taylorwin(64,nbar,sll);`
	`pwelch(x,win);`

`h = spectrum.welch('Hamming',segLen);` `psd(h,x);`	`win = hamming(segLen);` `pwelch(x,win);`

`h = spectrum.welch({'Hamming','periodic'},`
`segLen);`
`psd(h,x);`
	`win = hamming(segLen,'periodic');`
	`pwelch(x,win);`

`nbar = 4;`
`sll = 30;`
`h = spectrum.welch({'Taylor',nbar,sll},...`
`segLen);`
`psd(h,x);`
	`nbar = 4;`
	`sll = -30;`
	`win = taylorwin(segLen,nbar,sll);`
	`pwelch(x,win);`

`h = spectrum.welch('Hamming',segLen,ovlpPc`
`psd(h,x);`
	`win = hamming(segLen);`
	`Noverlap = ceil((ovlpPct/100)*segLen);`
	`pwelch(x,win,Noverlap);`

`h = spectrum.welch({'Hamming','periodic'},` `segLen,ovlpPct);`	`win = hamming(segLen,'periodic');` `Noverlap = ceil((ovlpPct/100)*segLen);`

Deprecated Syntax	Replacement Syntax
`psd(h,x);`	`pwelch(x,win,Noverlap);`

`nbar = 4;`
`sll = 30;`
`h = spectrum.welch({'Taylor',nbar,sll},...`
`segLen,ovlpPct);`
`psd(h,x);`
	`nbar = 4;`
	`sll = -30;`
	`win = taylorwin(segLen,nbar,sll);`
	`Noverlap = ceil((ovlpPct/100)*segLen);`
	`pwelch(x,win,Noverlap);`

`h = spectrum.welch(...);`
`psd(h,x,'NFFT',nfft);`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,nfft);`

`h = spectrum.welch(...);`
`psd(h,x,'Fs',fs);`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,[],fs);`

`h = spectrum.welch(...);`
`psd(h,x,'NFFT',nfft,'Fs',fs);`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,nfft,fs);`

`h = spectrum.welch(...);`
`psd(h,x,...,'FreqPoints','User Defined',...`
`'FrequencyVector',w);`
	`win = ...`
	`periodogram(x,win,w);`

`h = spectrum.periodogram(...);`
`psd(h,x,'FreqPoints','User Defined',...`
`'FrequencyVector',f,'Fs',fs);`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,f,fs);`

Two-sided spectrum of a real signal:
 	
`h = spectrum.welch(...);`
`psd(h,x,...,'SpectrumType','TwoSided');`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,...,'twosided');`

Two-sided spectrum with DC (0 frequency) in the center:
 	
`h = spectrum.welch(...);`
`psd(h,x,...,'CenterDC',true);`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,...,'centered');`

`h = spectrum.welch(...);`
`psd(h,x,...,'ConfLevel',p);`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,...'ConfidenceLevel',p);`

`h = spectrum.welch(...);`
`hPSD = psd(h,x,...);`
`Pxx = hPSD.Data;`
`F = hPSD.Frequencies;`
	`win = ...`
	`Noverlap = ...`
	`[Pxx,F] = pwelch(x,win,Noverlap,...);`

Deprecated Syntax	Replacement Syntax
| ```
h = spectrum.periodogram(...);
hPSD = psd(h,x,...,'ConfLevel',p);
Pxx = hPSD.Data;
F = hPSD.Frequencies;
Pxxc = hPSD.ConfInterval;
``` | ```
win = ...
Noverlap = ...
[Pxx,F,Pxxc] = pwelch(x,win,Noverlap,...
'ConfidenceLevel',p);
``` |

Periodogram MSSPECTRUM Object to Function Replacement Syntax

The `spectrum.periodogram` MSSPECTRUM object syntax will be removed in the future. The following table gives the equivalent recommended function syntax for `periodogram`. In the modified periodogram, you use a window other than the default rectangular window. To illustrate modified periodogram syntaxes, the table uses a specific window. In each example, `x` is the input signal.

Deprecated Syntax	Recommended Syntax
`h = spectrum.periodogram;` `msspectrum(h,x);`	`periodogram(x,'power');`
`h = spectrum.periodogram('Hamming');` `msspectrum(h,x);`	`win = hamming(length(x));` `periodogram(x,win,'power');`
`h = spectrum.periodogram({'Hamming','perio` `msspectrum(h,x);`	`win = hamming(length(x),'periodic');` `periodogram(x,win,'power');`

`nbar = 4;`
`sll = 30;`
`h = spectrum.periodogram({'Taylor',nbar,sl`
`msspectrum(h,x);`
	`nbar = 4;`
	`sll = -30;`
	`win = taylorwin(length(x),nbar,sll);`
	`periodogram(x,win,'power');`

`h = spectrum.periodogram(...);` `msspectrum(h,x,'NFFT',nfft);`	`win= ...` `periodogram(x,win,nfft,'power');`
`h = spectrum.periodogram(...);` `msspectrum(h,x,'Fs',fs);`	`win = ...` `periodogram(x,win,[],fs,'power');`
`h = spectrum.periodogram(...);` `msspectrum(h,x,'NFFT',nfft,'Fs',fs);`	`win = ...` `periodogram(x,win,nfft,fs,'power');`
`h = spectrum.periodogram(...);` `msspectrum(h,x,...,'SpectrumType','TwoSide`	`win = ...` `periodogram(x,win,..., 'twosided','power');`
`h = spectrum.periodogram(...);` `msspectrum(h, x,...,'CenterDC',true);`	`win = ...` `periodogram(x,win,...,'centered','power');`
`h = spectrum.periodogram(...);` `msspectrum(h,x,...,'ConfLevel',p);`	`win = ...` `periodogram(x,win,...,'ConfidenceLevel', p,...'power`

`h = spectrum.periodogram(...);`
`hMS = msspectrum(h,x,...);`
`Sxx = hMS.Data;`
`F = hMS.Frequencies;`
	`win = ...`
	`[Sxx,F] = periodogram(x,win,...,'power');`

`h = spectrum.periodogram(...);`	`win = ...`

Deprecated Syntax	Recommended Syntax

`hMS = msspectrum(h,x,...,'ConfLevel',p);`
`Sxx = hMS.Data;`
`F = hMS.Frequencies;`
`Sxxc = hMS.ConfInterval;`
	`[Sxx,F,Sxxc] = periodogram(x,win,...,'power');`

Welch MSSPECTRUM Object to Function Replacement Syntax

The `spectrum.welch` MSSPECTRUM object syntax will be removed in the future. The following table gives the equivalent recommended function syntax for `pwelch`. In the modified periodogram, you use a window other than the default rectangular window. To illustrate modified periodogram syntaxes, the table uses a specific window. In each example, `x` is the input signal.

Deprecated Syntax	Recommended Syntax
`h = spectrum.welch` `msspectrum(h,x);`	`win = hamming(64);` `pwelch(x,win,[],'power');`
`h = spectrum.welch('Gaussian');` `msspectrum(h,x);`	`win = gausswin(64);` `pwelch(x,win,[],'power');`
`h = spectrum.welch({'Hamming','periodic'})` `msspectrum(h,x);`	`win = hamming(64,'periodic');` `pwelch(x,win,[],'power');`

`nbar = 4;`
`sll = 30;`
`h = spectrum.welch({'Taylor',nbar,sll});`
`msspectrum(h,x);`
	`nbar = 4;`
	`sll = -30;`
	`win = taylorwin(64,nbar,sll);`
	`pwelch(x,win,[],'power');`

`segLen = 128;`
`h = spectrum.welch('Hamming',segLen);`
`msspectrum(h,x);`
	`win = hamming(128);`
	`pwelch(x,win,[],'power');`

`segLen = 128;`
`h = spectrum.welch({'Hamming','periodic'},`
`segLen);`
`msspectrum(h,x);`
	`win = hamming(128,'periodic');`
	`pwelch(x,win,[],'power');`

`nbar = 4;`
`sll = 30;`
`segLen = 128;`
`h = spectrum.welch({'Taylor',nbar,sll},seg`
`msspectrum(h,x);`
	`nbar = 4;`
	`sll = -30;`
	`segLen = 128;`
	`win = taylorwin(segLen,nbar,sll);`
	`pwelch(x,win,[],'power');`

`segLen = 128;`
`ovlpPct = 50;`
`h = spectrum.welch('Hamming',segLen,ovlpPc`
`msspectrum(h,x);`
	`segLen = 128;`
	`win = hamming(segLen);`
	`ovlpPct = 50;`
	`Noverlap = ceil((ovlpPct/100)*segLen);`
	`pwelch(x,win,Noverlap,'power');`

`segLen = 128;`
`ovlpPct = 50;`
`h = spectrum.welch({'Hamming','periodic'},`
`segLen,ovlpPct);`
	`segLen = 128;`
	`ovlpPct = 50;`
	`win = hamming(segLen,'periodic');`
	`Noverlap = ceil((ovlpPct/100)*segLen);`

Deprecated Syntax	Recommended Syntax
`msspectrum(h,x);`	`pwelch(x,win,Noverlap,'power');`

`nbar = 4;`
`sll = 30;`
`segLen = 128;`
`ovlpPct = 50;`
`h = spectrum.welch({'Taylor',nbar,sll},...`
`segLen,ovlpPct);`
`msspectrum(h,x);`
	`nbar = 4;`
	`sll = -30;`
	`segLen = 128;`
	`win = taylorwin(segLen,nbar,sll);`
	`ovlpPct = 50;`
	`Noverlap = ceil((ovlpPct/100)*segLen);`
	`pwelch(x,win,Noverlap,'power');`

`h = spectrum.welch(...);`
`msspectrum(h,x,'NFFT',nfft);`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,nfft,'power');`

`h = spectrum.welch(...);`
`msspectrum(h,x,'Fs',fs);`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,[],fs,'power');`

`h = spectrum.welch(...);`
`msspectrum(h,x,'NFFT',nfft,'Fs',fs);`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,nfft,fs,'power');`

`h = spectrum.welch(...);`
`msspectrum(h, x,...,'FreqPoints','User Def`
`  'FrequencyVector',w);`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,f,fs,'power');`

`h = spectrum.welch(...);`
`msspectrum(h,x,...,'SpectrumType','TwoSide`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,...,'twosided','power');`

`h = spectrum.welch(...);`
`msspectrum(h,x,...,'CenterDC',true);`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,...,'centered','power');`

`h = spectrum.welch(...);`
`msspectrum(h,x,...,'ConfLevel',p);`
	`win = ...`
	`Noverlap = ...`
	`pwelch(x,win,Noverlap,...,'ConfidenceLevel',p,'power`

`h = spectrum.welch(...);`
`hMS = msspectrum(h,x,...);`
`Sxx = hMS.Data;`
`F = hMS.Frequencies;`
	`[Sxx,F] = pwelch(...,'power');`

`h = spectrum.welch(...);`
`hMS = msspectrum(h, x, …, 'ConfLevel', p);`
`Sxx = hMS.Data;`
`F = hMS.Frequencies;`
`Sxxc = hMS.ConfInterval;`
	`[Sxx,F,Sxxc] = pwelch(...,'ConfidenceLevel',p,'power`

Subspace Pseudospectrum Object to Function Replacement Syntax

The pseudospectrum object syntax will be removed in the future. The following table gives the equivalent recommended function syntax. The table uses `spectrum.music` and the functional equivalent, `pmusic`, but the syntax replacements are also valid for `spectrum.eigenvector` to `peig`. In each example, `x` is the input signal.

Deprecated Syntax	Replacement Syntax
`h = spectrum.music(nsinusoids);` `pseudospectrum(h,x);`	`pmusic(x,nsinusoids)`
`h = spectrum.music(nsinusoids);` `pseudospectrum(h,x,'Fs',fs)`	`pmusic(x,nsinusoids,[],fs);`

`h = spectrum.music(nsinusoids,segLen,ovlpP`
`'Hamming');`
`pseudospectrum(h,x)`
	`win = hamming(segLen)`
	`Noverlap = ceil(ovlpPct/100*segLen);`
	`P = nsinusoids;`
	`Fs = 2*pi;`
	`pmusic(x,P,[],Fs,win,Noverlap);`

`h = spectrum.music(nsinusoids,segLen,ovlpP`
`winName, thresh);`
`pseudospectrum(h,x)`
	`win = winfunc(segLen)`
	`Noverlap = ceil(ovlpPct/100*segLen);`
	`P = [nsinusoids thresh];`
	`Fs = 2 *pi;`
	`pmusic(x,P,[],Fs,win,Noverlap);`

`h = spectrum.music(nsinusoids,segLen,...`
`ovlpPct,winName,thresh);`
`pseudospectrum(h,x,'Fs',fs)`
	`win = hamming(segLen)`
	`nfft = max(256,2^nexpow2(segLen));`
	`Noverlap = ceil(ovlpPct/100*segLen);`
	`P = [nsinusoids thresh];`
	`pmusic(x,P,[],Fs,win,Noverlap);`

`h = spectrum.music(nsinusoids,segLen,...`
`ovlpPct,winName,thresh);`
`pseudospectrum(h,x,'Fs',fs,'SpectrumRange'`
	`win = hamming(segLen)`
	`Noverlap = ceil(ovlpPct/100*segLen);`
	`P = [nsinusoids thresh];`
	`pmusic(x,P,[],Fs,range,win,Noverlap);`

`h = spectrum.music(nsinusoids,segLen,...`
`ovlpPct,winName,thresh);`
`pseudospectrum(h,x,'Fs',fs,'SpectrumRange'`
	`win = hamming(segLen)`
	`Noverlap = ceil(ovlpPct/100*segLen);`
	`P = [nsinusoids thresh];`
	`pmusic(x,P,nfft,Fs,range,win,Noverlap);`

`h = spectrum.music(nsinusoids,segLen,...`
`ovlpPct,winName,thresh);`
`pseudospectrum(h,x,...,'FreqPoints','User`
`'Frequency Vector',fVec)`
	`win = hamming(segLen)`
	`Noverlap = ceil(ovlpPct/100*segLen);`
	`P = [nsinusoids thresh];`
	`pmusic(x,P,fVec,Fs,range,win,Noverlap);`

Deprecated Syntax	Replacement Syntax
`h = spectrum.music(...,'DataMatrix');` `pseudospectrum(...)`	`nfft = min(256,2^nextpow2(size(x,1)));` `pmusic(x,P,nfft,Fs,range,win)`

`h = spectrum.music(...,'CorrelationMatrix'`
`pseudospectrum(...)`
	`pmusic(x,P,'corr',nfft,Fs,range,win,Noverlap);`
	or
	
	`pmusic(x,P,'corr',fVec,Fs,range,win,Noverlap);`

`h = spectrum.music(...);` `pseudospectrum(...,'CenterDC',true)`	`pmusic(...,'centered');`
`[Spec,F] = pseudospectrum(...)`	`[Spec,F] = pmusic(...);`

Common Applications

Create Uniform and Nonuniform Time Vectors

You can create uniform and nonuniform time vectors for use in computations involving time series.

Use the colon operator if you know the sampling frequency. If your system samples time at a rate of 15 Hz during one second, you get 16 readings, including the one at zero.

```
Fs = 15;
Ts = 1/Fs;
ts = 0:Ts:1;
```

Use `linspace` if you know the beginning and end of the time interval and the number of samples. Suppose you start a stopwatch and stop it one second later. If you know your instrument took 15 readings, you can generate the time vector.

```
tl = linspace(0,1,15);
```

You can compute the sample rate directly from the samples and use it to reconstruct the time vector.

```
sf = 1/(tl(2)-tl(1));

TL = (0:length(tl)-1)/sf;

ErrorTL = max(abs(tl-TL))
```

```
ErrorTL =

     0
```

You can also reconstruct `ts` using `linspace`.

```
lts = length(ts);
TS = linspace(ts(1),ts(lts),lts);

ErrorTS = max(abs(ts-TS))
```

```
ErrorTS =
```

```
1.1102e-16
```

`linspace` and the colon operator create row vectors by default. Transpose them to obtain column vectors.

```
tcol = tl';
ttrans = ts';
```

Combine `linspace` and the colon operator to generate nonuniform time vectors of arbitrary characteristics.

Suppose you have a Gaussian-modulated sinusoidal pulse that you must sample. The pulse changes rapidly during a one-second interval but slowly during the preceding and following seconds.

Sample the region of interest at 100 Hz and take only five samples before and after. Concatenate the vectors using square brackets.

```
gpl = @(x) 2.1*gauspuls(x-1.5,5,0.4);

Ffast = 100;
Tf = 1/Ffast;
Nslow = 5;
tdisc = [linspace(0,1,Nslow) 1+Tf:Tf:2-Tf linspace(2,3,Nslow)];
```

Generate 20001 samples of the function to simulate the continuous-time pulse. Overlay a plot of the samples defined by `tsf`.

```
Tcont = linspace(0,3,20001)';

plot(Tcont,gpl(Tcont),tdisc,gpl(tdisc),'o','markersize',5)
```

See Also

`gauspuls`

Remove Trends from Data

Measured signals can show overall patterns that are not intrinsic to the data. These trends can sometimes hinder the data analysis and must be removed.

Consider two electrocardiogram (ECG) signals with different trends. ECG signals are sensitive to disturbances such as power source interference. Load the signals and plot them.

```
load(fullfile(matlabroot,'examples','signal','ecgSignals.mat'))
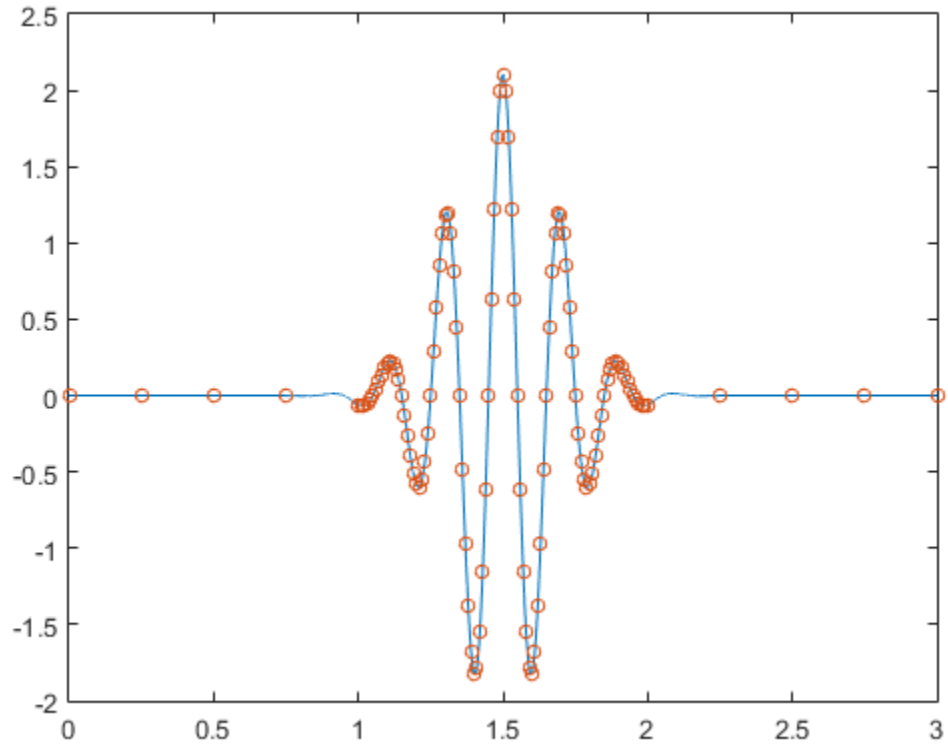
t = (1:length(ecgl))';

subplot(2,1,1)
plot(t,ecgl), grid
title 'ECG Signals with Trends', ylabel 'Voltage (mV)'

subplot(2,1,2)
plot(t,ecgnl), grid
xlabel Sample, ylabel 'Voltage (mV)'
```

The signal on the first plot shows a linear trend. The trend on the second signal is nonlinear. To eliminate the linear trend, use the MATLAB® function detrend.

```
dt_ecgl = detrend(ecgl);
```

To eliminate the nonlinear trend, fit a low-order polynomial to the signal and subtract it. In this case, the polynomial is of order 6. Plot the two new signals.

```
opol = 6;
[p,s,mu] = polyfit(t,ecgnl,opol);
f_y = polyval(p,t,[],mu);
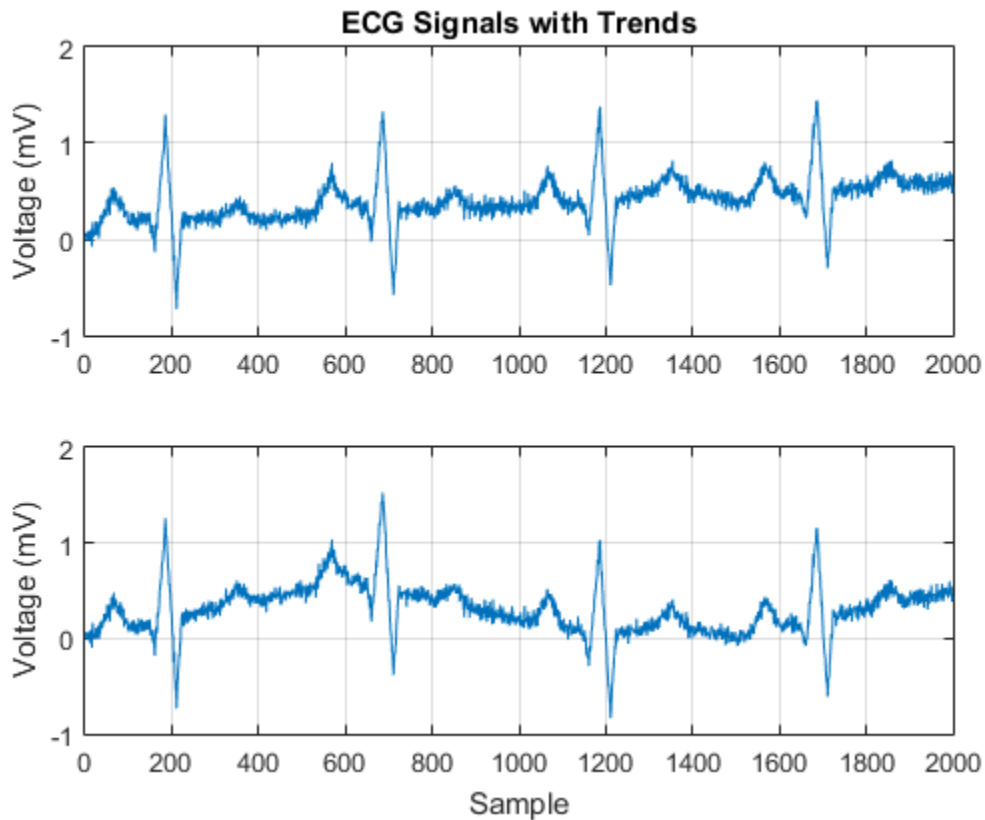
dt_ecgnl = ecgnl - f_y;
```

```
subplot(2,1,1)
plot(t,dt_ecgl), grid
title 'Detrended ECG Signals', ylabel 'Voltage (mV)'

subplot(2,1,2)
plot(t,dt_ecgnl), grid
xlabel Sample, ylabel 'Voltage (mV)'
```


The trends have been effectively removed. Observe how the signals do not show a baseline shift anymore. They are ready for further processing.

See Also
detrend | polyfit | polyval

Related Examples

- "Peak Analysis"

Remove the 60 Hz Hum from a Signal

Alternating current in the United States and several other countries oscillates at a frequency of 60 Hz. Those oscillations often corrupt measurements and have to be subtracted.

Study the open-loop voltage across the input of an analog instrument in the presence of 60 Hz power-line noise. The voltage is sampled at 1 kHz.

```
load openloop60hertz, openLoop = openLoopVoltage;
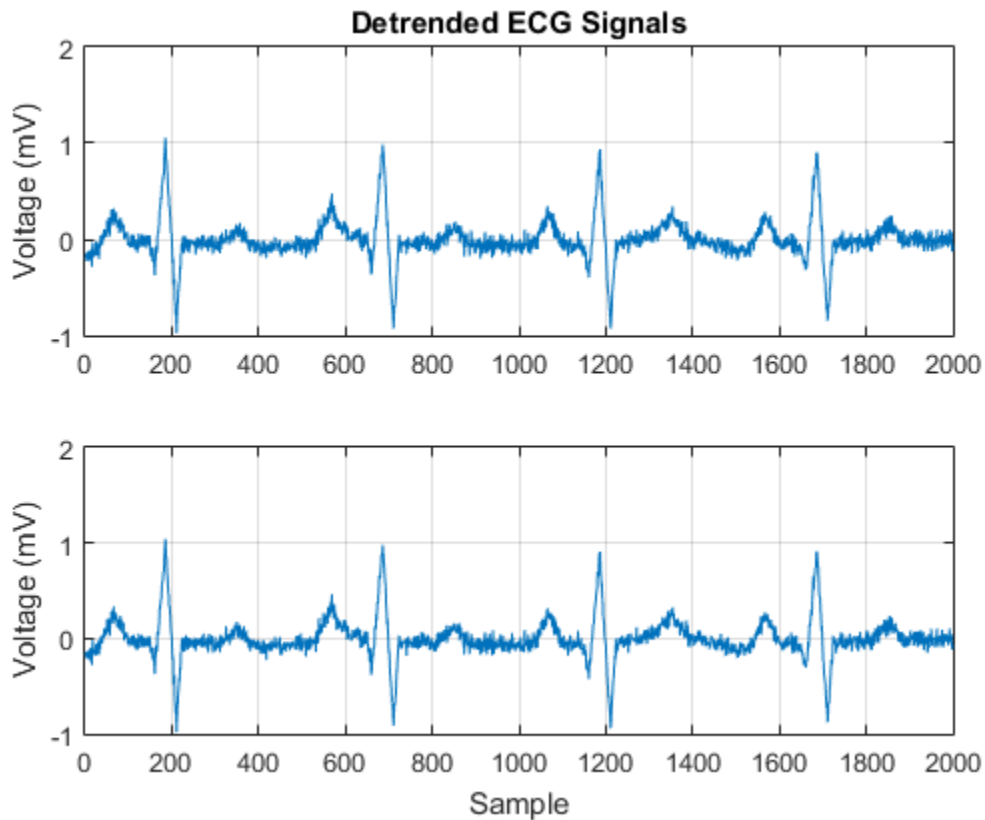
Fs = 1000;
t = (0:length(openLoop)-1)/Fs;

plot(t,openLoop)
ylabel 'Voltage (V)', xlabel 'Time (s)'
title 'Open-Loop Voltage with 60 Hz Noise'
```

Eliminate the 60 Hz noise with a Butterworth notch filter. Use `designfilt` to design it. The width of the notch is defined by the 59 to 61 Hz frequency interval. The filter removes at least half the power of the frequency components lying in that range.

```
d = designfilt('bandstopiir','FilterOrder',2, ...
               'HalfPowerFrequency1',59,'HalfPowerFrequency2',61, ...
               'DesignMethod','butter','SampleRate',Fs);
```

Plot the frequency response of the filter. Note that this notch filter provides up to 45 dB of attenuation.

```
fvtool(d,'Fs',Fs)
```

Filter the signal with `filtfilt` to compensate for filter delay. Note how the oscillations decrease significantly.

```
buttLoop = filtfilt(d,openLoop);
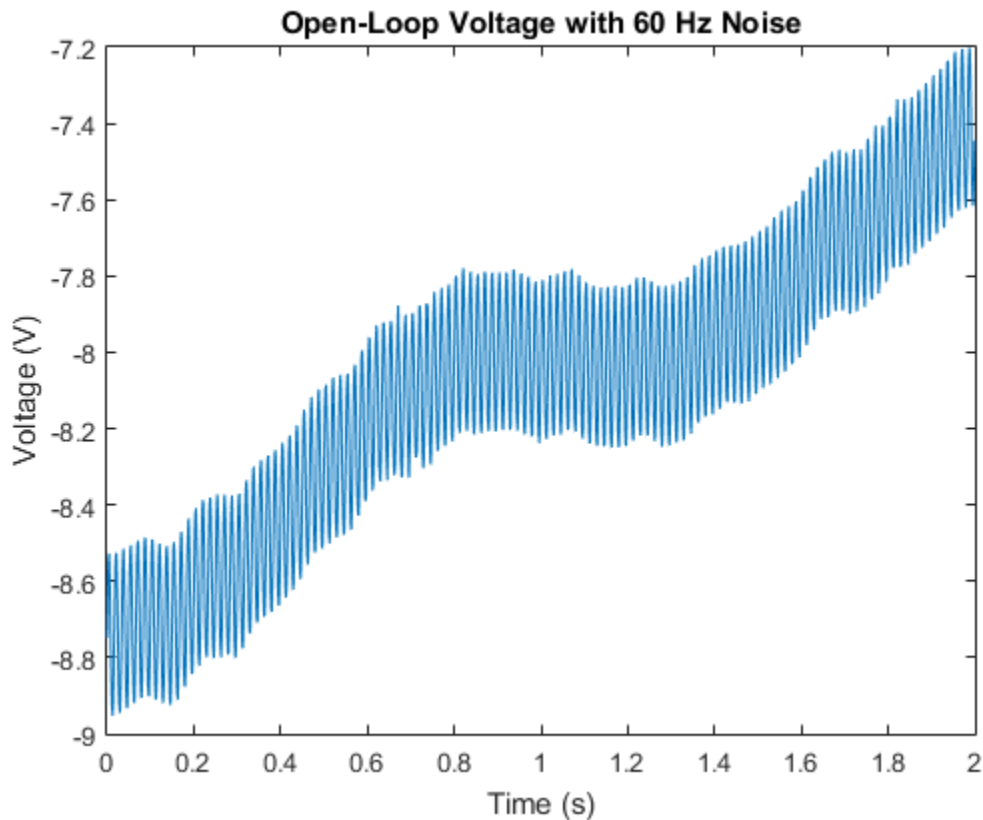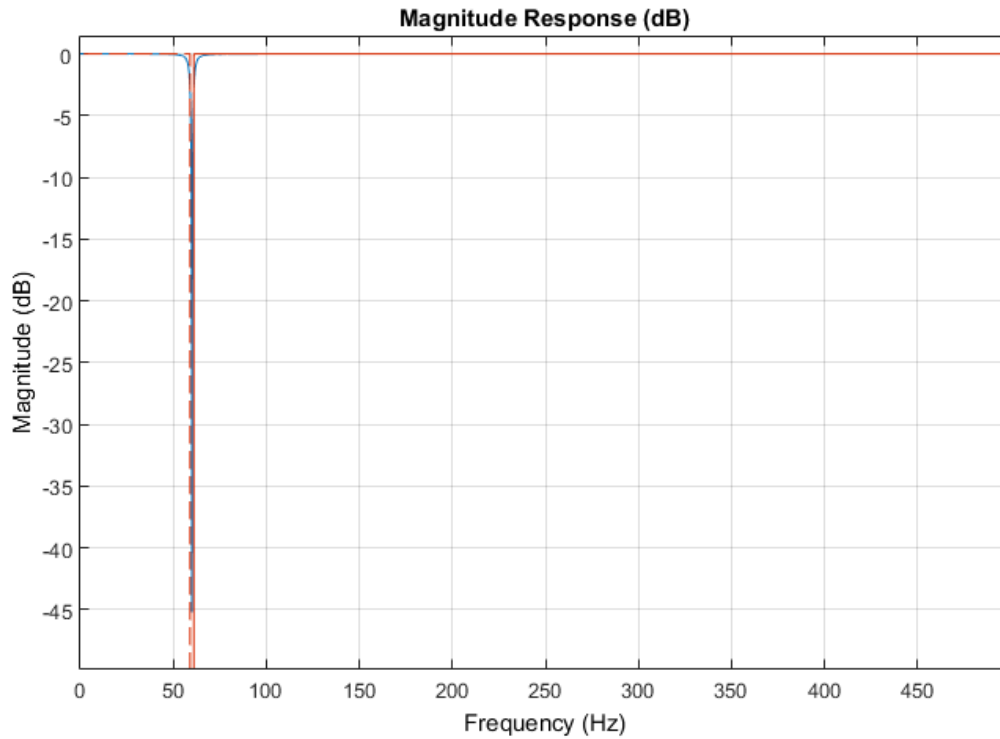
plot(t,openLoop,t,buttLoop)
ylabel 'Voltage (V)', xlabel 'Time (s)'
title 'Open-Loop Voltage', legend('Unfiltered','Filtered')
```

Use the periodogram to see that the "spike" at 60 Hz has been eliminated.

```
[popen,fopen] = periodogram(openLoop,[],[],Fs);
[pbutt,fbutt] = periodogram(buttLoop,[],[],Fs);
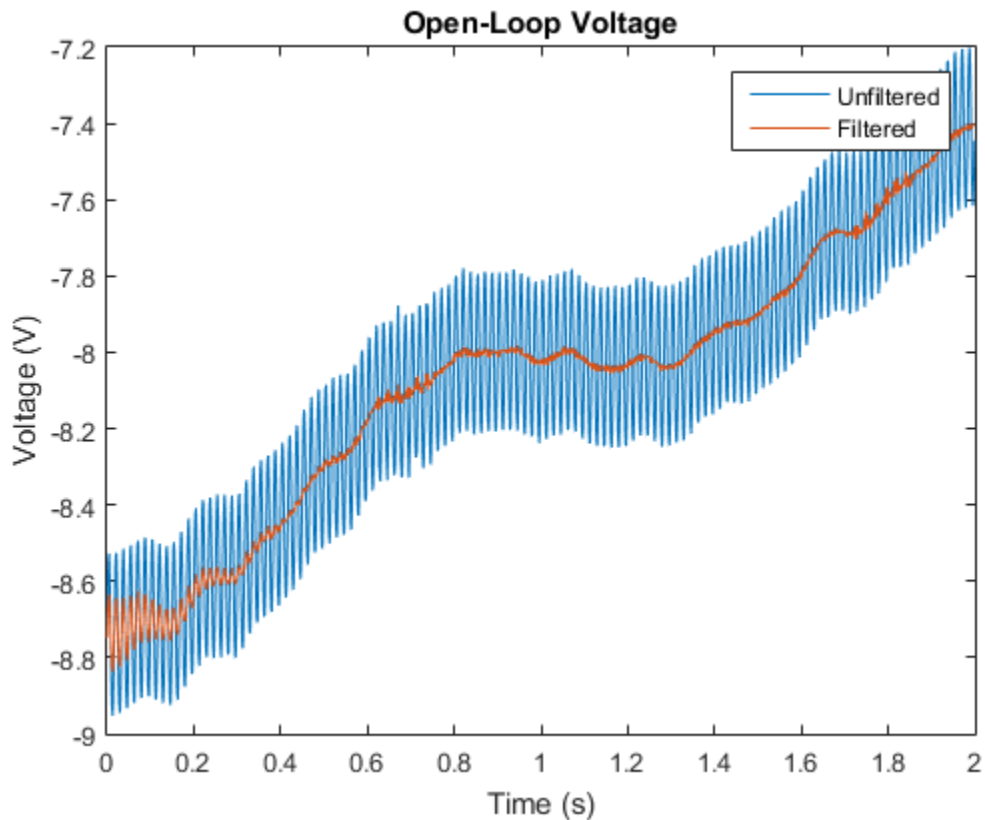
plot(fopen,20*log10(abs(popen)),fbutt,20*log10(abs(pbutt)),'--')
ylabel 'Power/frequency (dB/Hz)', xlabel 'Frequency (Hz)'
title 'Power Spectrum', legend('Unfiltered','Filtered')
```

See Also

designfilt | filtfilt | fvtool | periodogram

Related Examples

- "Signal Smoothing"

Remove Spikes from a Signal

Sometimes data exhibit unwanted transients, or spikes. Median filtering is a natural way to eliminate them.

Consider the open-loop voltage across the input of an analog instrument in the presence of 60 Hz power-line noise. The sampling rate is 1 kHz.

```
load openloop60hertz
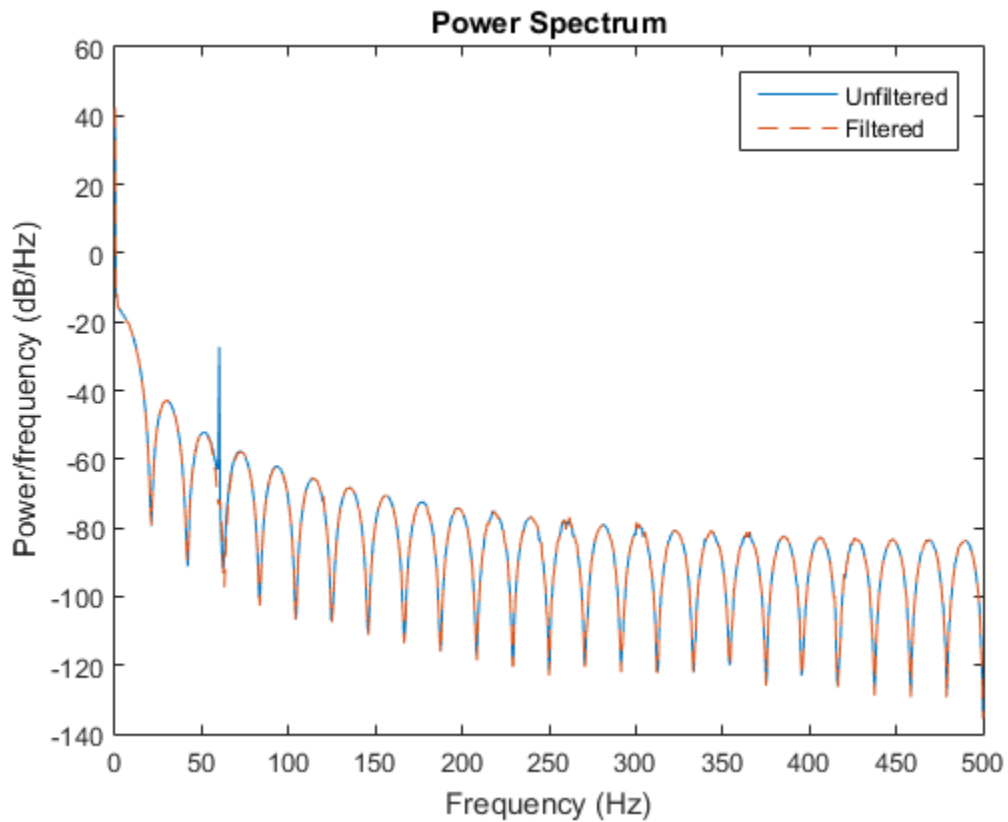
fs = 1000;
t = (0:numel(openLoopVoltage) - 1)/fs;
```

Corrupt the signal by adding transients with random signs at random points. Reset the random number generator for reproducibility.

```
rng default

spikeSignal = zeros(size(openLoopVoltage));
spks = 10:100:1990;
spikeSignal(spks+round(2*randn(size(spks)))) = sign(randn(size(spks)));

noisyLoopVoltage = openLoopVoltage + spikeSignal;

plot(t,noisyLoopVoltage)

xlabel('Time (s)')
ylabel('Voltage (V)')
title('Open-Loop Voltage with Added Spikes')
yax = ylim;
```

The function `medfilt1` replaces every point of a signal by the median of that point and a specified number of neighboring points. Accordingly, median filtering discards points that differ considerably from their surroundings. Filter the signal, using sets of three neighboring points to compute the medians. Note how the spikes vanish.

```
medfiltLoopVoltage = medfilt1(noisyLoopVoltage,3);
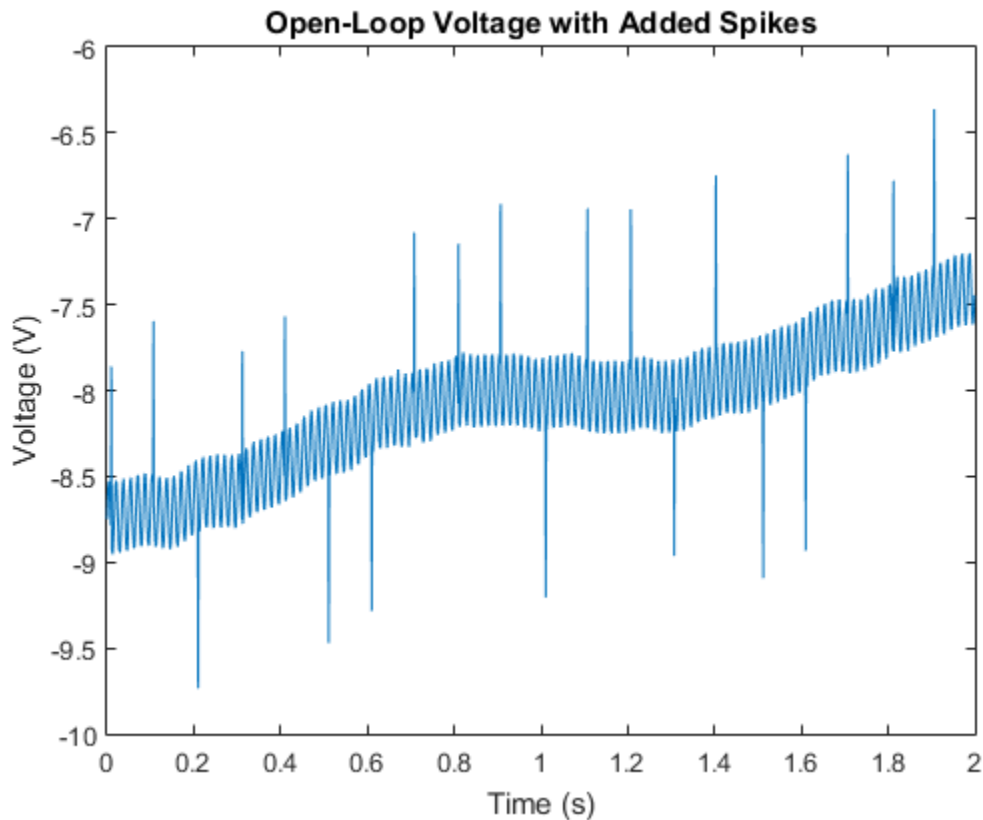
plot(t,medfiltLoopVoltage)

xlabel('Time (s)')
ylabel('Voltage (V)')
title('Open-Loop Voltage After Median Filtering')
ylim(yax)
```

Open-Loop Voltage After Median Filtering

See Also
medfilt1

Related Examples
- "Signal Smoothing"

Process a Signal with Missing Samples

Consider the weight of a person as recorded (in pounds) during the leap year 2012. The person did not record their weight every day. You would like to study the periodicity of the signal, but before you can do so you must take care of the missing data.

Load the data and convert the measurements to kilograms. Missed readings are set to NaN. Determine how many points are missing.

```
load(fullfile(matlabroot,'examples','signal','weight2012.dat'))
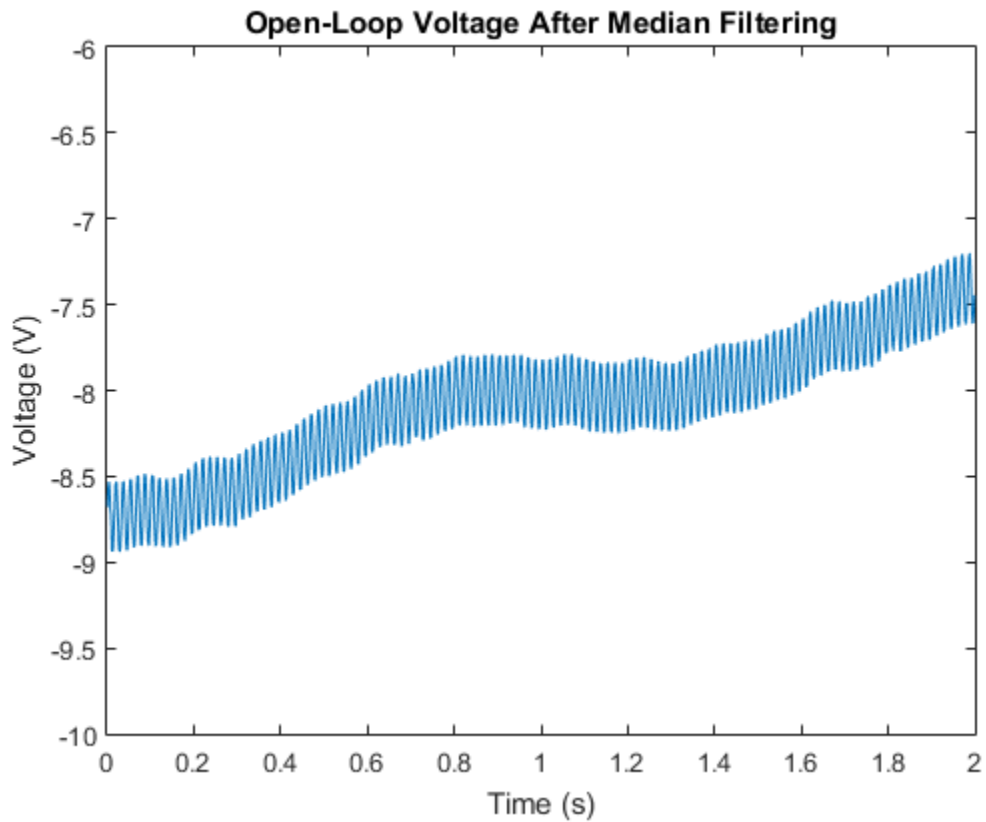
wgt = weight2012(:,2)/2.20462;
daynum = 1:length(wgt);
missing = isnan(wgt);

fprintf('Missing %d samples of %d\n',sum(missing),max(daynum))

Missing 27 samples of 366
```

Assign values to the missing points using `resample`. By default, `resample` makes estimates using linear interpolation. Plot the original and interpolated readings. Zoom in on days 200 through 250, which contain about half of the missing points.

```
wgt_orig = wgt;
wgt = resample(wgt,daynum);

plot(daynum,wgt_orig,'.-',daynum,wgt,'o')
xlabel('Day')
ylabel('Weight (kg)')
axis([200 250 73 77])
legend('Original','Interpolated')
```

Determine if the signal is periodic by analyzing it in the frequency domain. Find the cycle duration, measuring time in weeks. Subtract the mean to concentrate on fluctuations.

```
Fs = 7;
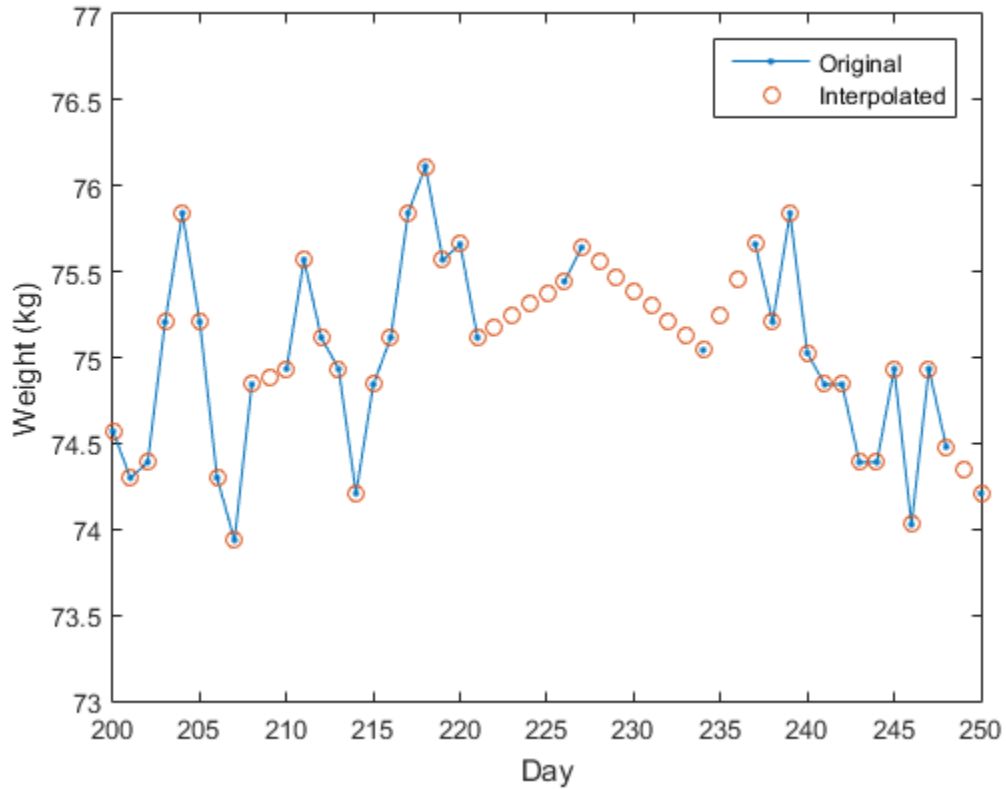
[p,f] = pwelch(wgt-mean(wgt),[],[],[],Fs);

plot(f,p)
xlabel('Frequency (week^{-1})')
```

Notice how the person's weight oscillates weekly. Is there a noticeable pattern from week to week? Eliminate the last two days of the year to get 52 weeks. Reorder the measurements according to the day of the week.

```
wgd = reshape(wgt(1:7*52),[7 52]);

plot(wgd')
xlabel('Week')
ylabel('Weight (kg)')

q = legend(datestr(datenum(2012,1,1:7),'dddd'));
q.Location = 'NorthWest';
```

Smooth out the fluctuations using a filter that fits low-order polynomials to subsets of the data. Specifically, set it to fit cubic polynomials to sets of seven days.

```
wgs = sgolayfilt(wgd',3,7);
```

```
plot(wgs)
xlabel('Week')
ylabel('Smoothed weight (kg)')
```

```
q = legend(datestr(datenum(2012,1,1:7),'dddd'));
q.Location = 'SouthEast';
```

This person tends to eat more, and thus weigh more, during the weekend. Verify by computing the daily means.

```
for jk = 1:7
    fprintf('%3s mean: %5.1f kg\n', ...
        datestr(datenum(2012,1,jk),'ddd')',mean(wgd(jk,:)))
end
```

```
Sun mean:  76.2 kg
Mon mean:  75.7 kg
Tue mean:  75.2 kg
Wed mean:  74.9 kg
Thu mean:  75.1 kg
Fri mean:  75.3 kg
```

```
Sat mean:   75.8 kg
```

See Also
pwelch | sgolayfilt

Related Examples
* "Signal Smoothing"

Reconstruct a Signal from Irregularly Sampled Data

People predisposed to blood clotting are treated with warfarin, a blood thinner. The international normalized ratio (INR) measures the effect of the drug. Larger doses increase the INR and smaller doses decrease it. Patients are monitored regularly, and when their INRs fall out of the target range, their doses and the frequencies of their tests change.

The file `INR.mat` contains the INR measurements performed on a patient over a five-year period. The file includes a `datetime` array with the date and time of each measurement, and a vector with the corresponding INR readings.

Load the file. Use `datenum` to convert the dates to serial date numbers. Center the INR values at 2.5, the middle of the typical target range of 2.0 to 3.0.

```
load(fullfile(matlabroot,'examples','signal','INR.mat'));

dates = datenum(Date);
inr = INR - 2.5;
```

Plot the INR as a function of time. Express the *x*-axis as dates. To avoid clutter, limit the number of ticks to three. Overlay the target INR range on the plot, adding back the 2.5.

```
plot(dates,inr,'o')
axis tight

xl = xlim;
dt = linspace(xl(1),xl(2),3);
ax = gca;
ax.XTick = dt;
ax.XTickLabel = datestr(dt,'mm/dd/yy');

ax.YTick = -1.5:1.5;
ax.YTickLabel = num2str((ax.YTick+2.5)');
hold on
plot([xl;xl]',[-1 1;-1 1]/2,'k:')
```

Resample the data to make the INR readings uniformly spaced. The first reading was taken at 11:28 a.m. on a Friday. Use `resample` to estimate the patient's INR at that time on every subsequent Friday. Specify a sample rate of one reading per week, or equivalently, 1/7 readings per day. Use spline interpolation for the resampling.

```
First = datestr(dates(1),'dddd, mm/dd/yy, HH:MM AM')
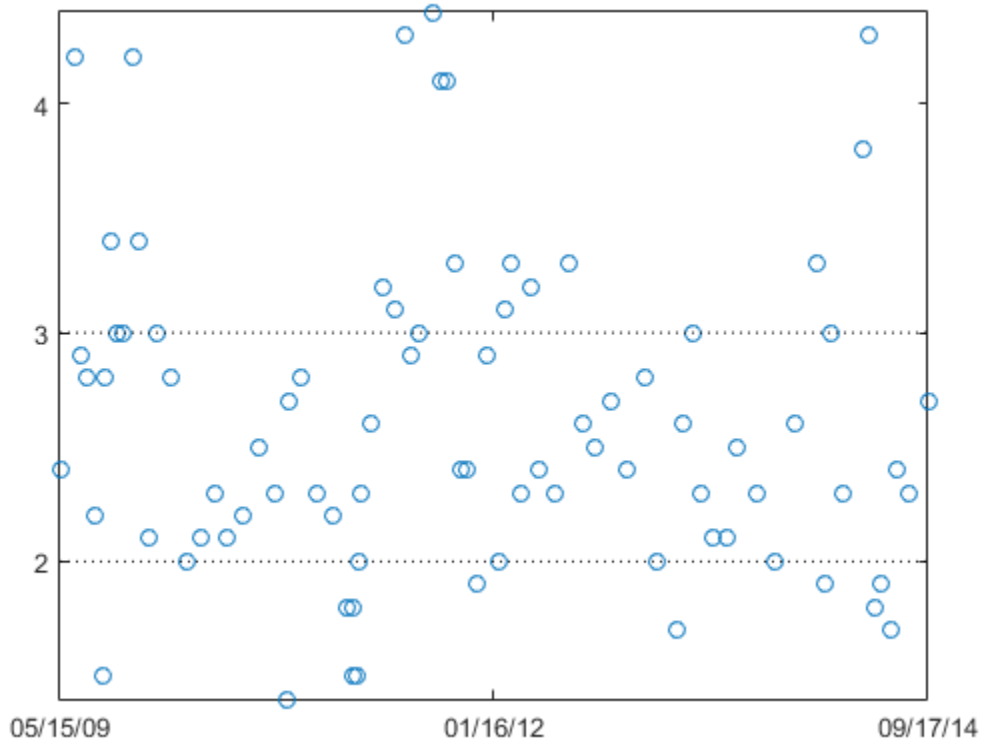
[rum,tee] = resample(inr,dates,1/7,1,1,'spline');

plot(tee,rum,'.-')
title('INR')


First =
```

Friday, 05/15/09, 11:28 AM

Each INR reading determines when the patient must be tested next. Use `diff` to construct a vector of time intervals between measurements. Express the intervals in weeks and plot them using the same *x*-axis as before. For the last point, use the next date prescribed by the anticoagulation nurse.

```
nxt = datenum('10/30/14 07:00 PM');
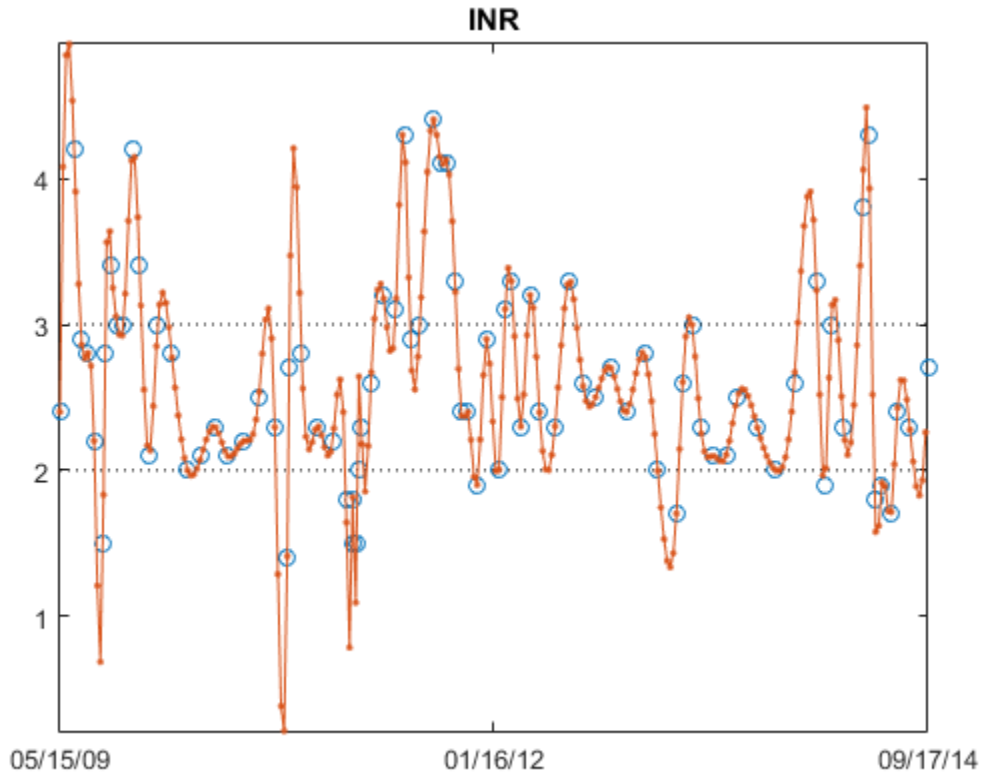
figure
plot(dates,diff([dates;nxt]/7),'o-')
axis tight
```

18-25

```
ax = gca;
ax.XTick = dt;
ax.XTickLabel = datestr(dt,'mm/dd/yy');
title('Time Until Next Reading')
ylabel('Weeks')
```


When the INR is out of range, the times between INR readings remain short. When the INR is too low, the risk of thrombosis is elevated, and the times between readings become especially close. When the patient's INR is in range, the times between readings increase steadily until the ratio becomes too small or too large.

See Also
datenum | datestr | resample

External Web Sites

- National Institutes of Health. Blood Thinners. http://www.nlm.nih.gov/medlineplus/bloodthinners.html

Align Signals with Different Start Times

Many measurements involve data collected asynchronously by multiple sensors. If you want to integrate the signals and study them in tandem, you have to synchronize them. Use `xcorr` for that purpose.

For example, consider a car crossing a bridge. The vibrations it produces are measured by three identical sensors located at different spots. The signals have different arrival times.

Load the signals into the MATLAB® workspace and plot them.

```
load relatedsig
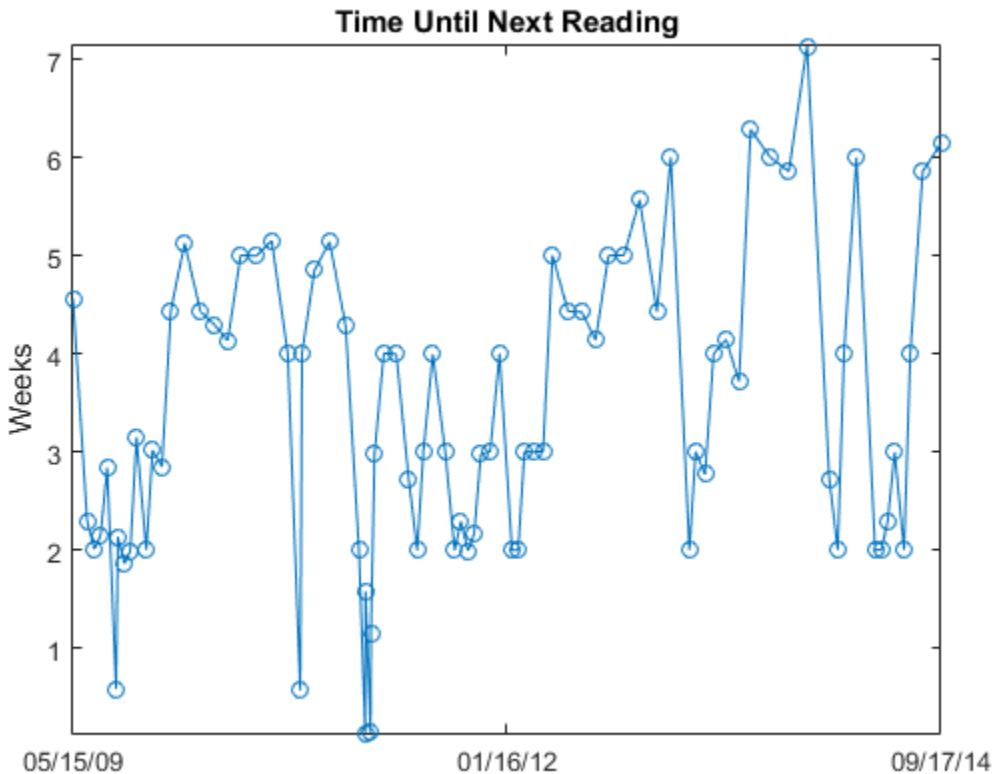
ax(1) = subplot(3,1,1);
plot(s1)
ylabel('s_1')
axis tight

ax(2) = subplot(3,1,2);
plot(s2)
ylabel('s_2')
axis tight

ax(3) = subplot(3,1,3);
plot(s3)
ylabel('s_3')
axis tight
xlabel('Samples')
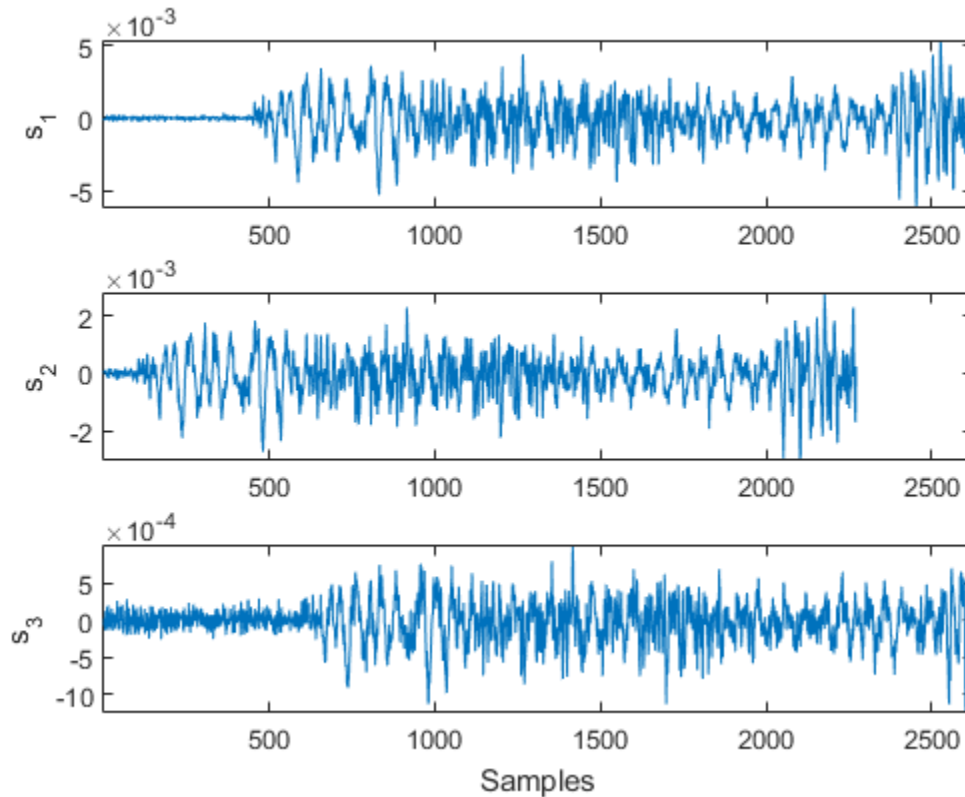
linkaxes(ax,'x')
```

Compute the cross-correlations between the three pairs of signals. Normalize them so their maximum value is one.

```
[C21,lag21] = xcorr(s2,s1);
C21 = C21/max(C21);

[C31,lag31] = xcorr(s3,s1);
C31 = C31/max(C31);

[C32,lag32] = xcorr(s3,s2);
C32 = C32/max(C32);
```

The locations of the maximum values of the cross-correlations indicate time leads or lags.

```
[M21,I21] = max(C21);
t21 = lag21(I21);

[M31,I31] = max(C31);
t31 = lag31(I31);

[M32,I32] = max(C32);
t32 = lag31(I32);
```

Plot the cross-correlations. In each plot display the location of the maximum.

```
subplot(3,1,1)
plot(lag21,C21,[t21 t21],[-0.5 1],'r:')
text(t21+100,0.5,['Lag: ' int2str(t21)])
ylabel('C_{21}')
axis tight
title('Cross-Correlations')

subplot(3,1,2)
plot(lag31,C31,[t31 t31],[-0.5 1],'r:')
text(t31+100,0.5,['Lag: ' int2str(t31)])
ylabel('C_{31}')
axis tight

subplot(3,1,3)
plot(lag32,C32,[t32 t32],[-0.5 1],'r:')
text(t32+100,0.5,['Lag: ' int2str(t32)])
ylabel('C_{32}')
axis tight
xlabel('Samples')
```

Cross-Correlations

s_2 leads s_1 by 350 samples; s_3 lags s_1 by 150 samples. Thus s_2 leads s_3 by 500 samples. Line up the signals by clipping the vectors with longer delays.

```
s1 = s1(-t21:end);
s3 = s3(t32:end);
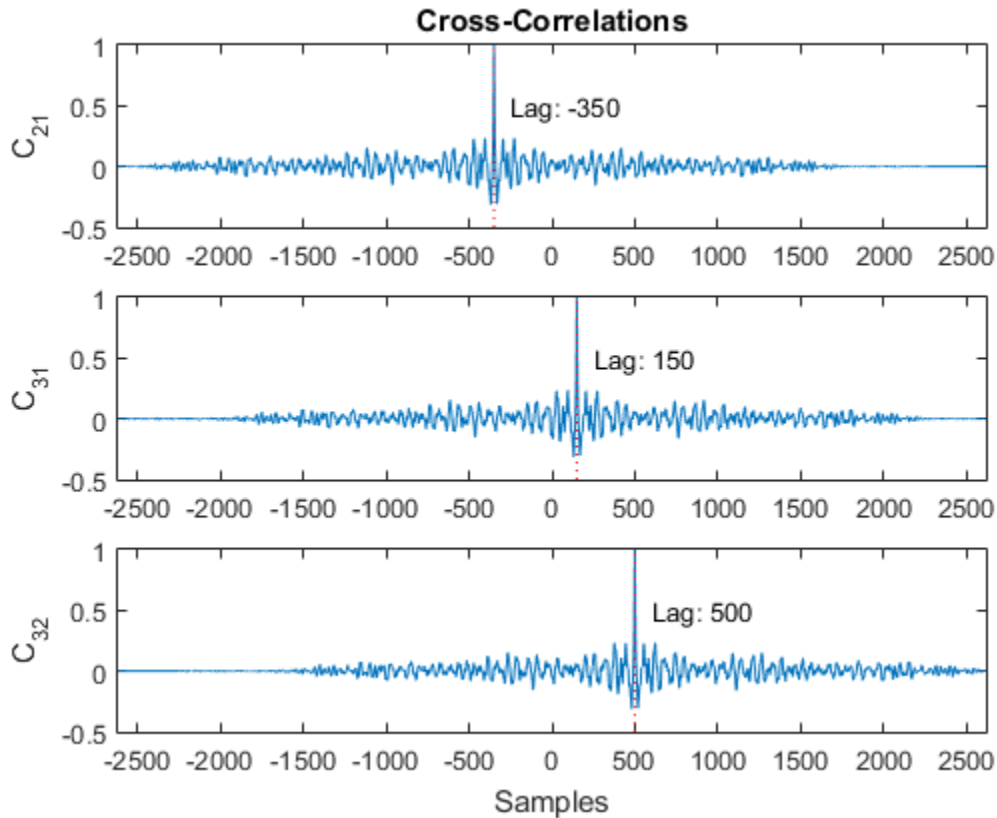
ax(1) = subplot(3,1,1);
plot(s1)
ylabel('s_1')
axis tight

ax(2) = subplot(3,1,2);
plot(s2)
ylabel('s_2')
```

```
axis tight

ax(3) = subplot(3,1,3);
plot(s3)
ylabel('s_3')
axis tight
xlabel('Samples')
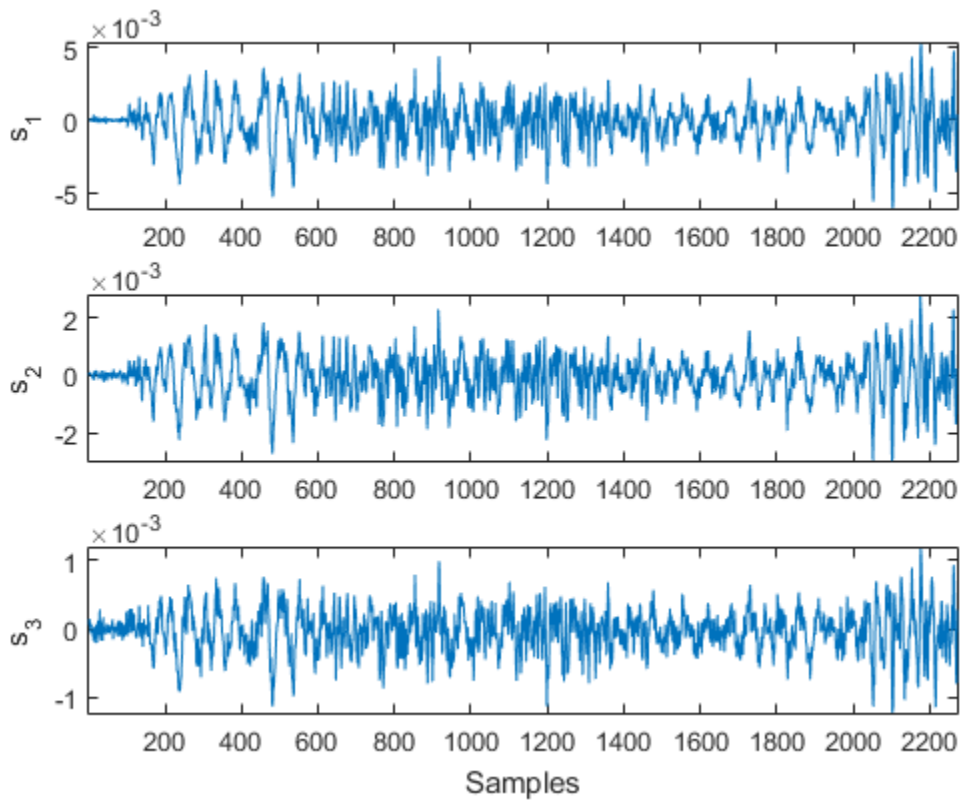
linkaxes(ax,'x')
```

The signals are now synchronized and ready for further processing.

See Also

xcorr

Related Examples

·   "Measuring Signal Similarities"

Find a Signal in a Measurement

You receive some data and would like to know if it matches a longer stream you have measured. Cross-correlation allows you to make that determination, even when the data are corrupted by noise.

Load into the workspace a recording of a ring spinning on a tabletop. Crop a one-second fragment and listen to it.

```matlab
load(fullfile(matlabroot,'examples','signal','Ring.mat'))

Time = 0:1/Fs:(length(y)-1)/Fs;

m = min(y);
M = max(y);

Full_sig = double(y);

timeA = 7;
timeB = 8;
snip = timeA*Fs:timeB*Fs;

Fragment = Full_sig(snip);

% To hear, type soundsc(Fragment,Fs)
```

Plot the signal and the fragment. Highlight the fragment endpoints for reference.

```matlab
plot(Time,Full_sig,[timeA timeB;timeA timeB],[m m;M M],'r--')
xlabel('Time (s)')
ylabel('Clean')
axis tight
```
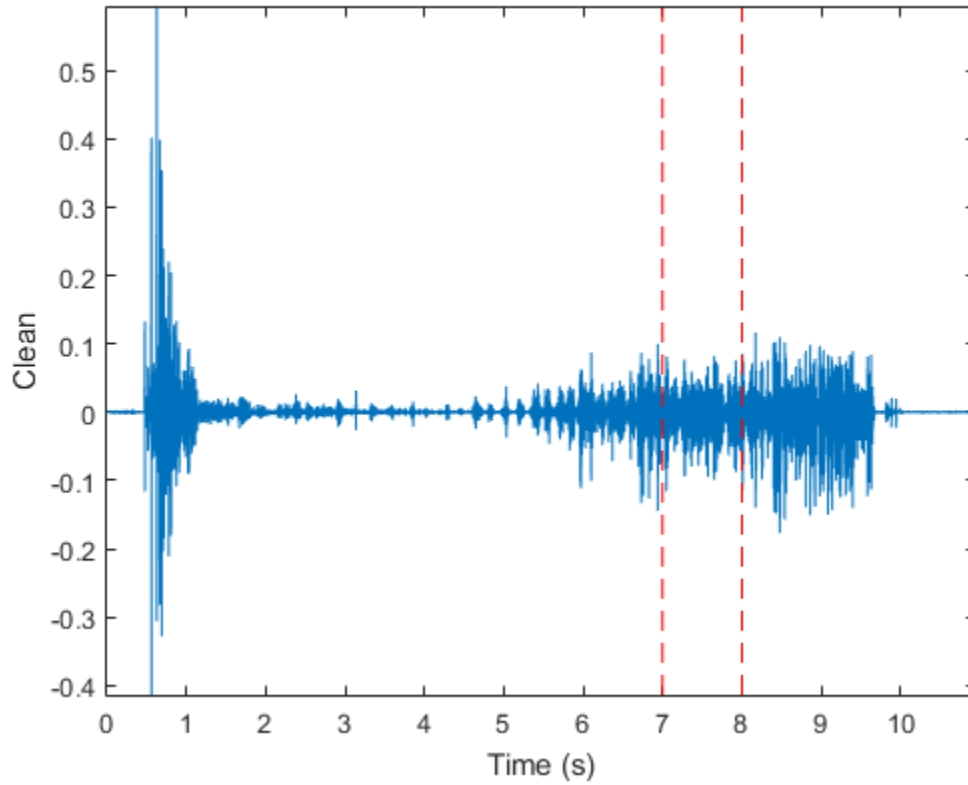
```
plot(snip/Fs,Fragment)
xlabel('Time (s)')
ylabel('Clean')
title('Fragment')
axis tight
```
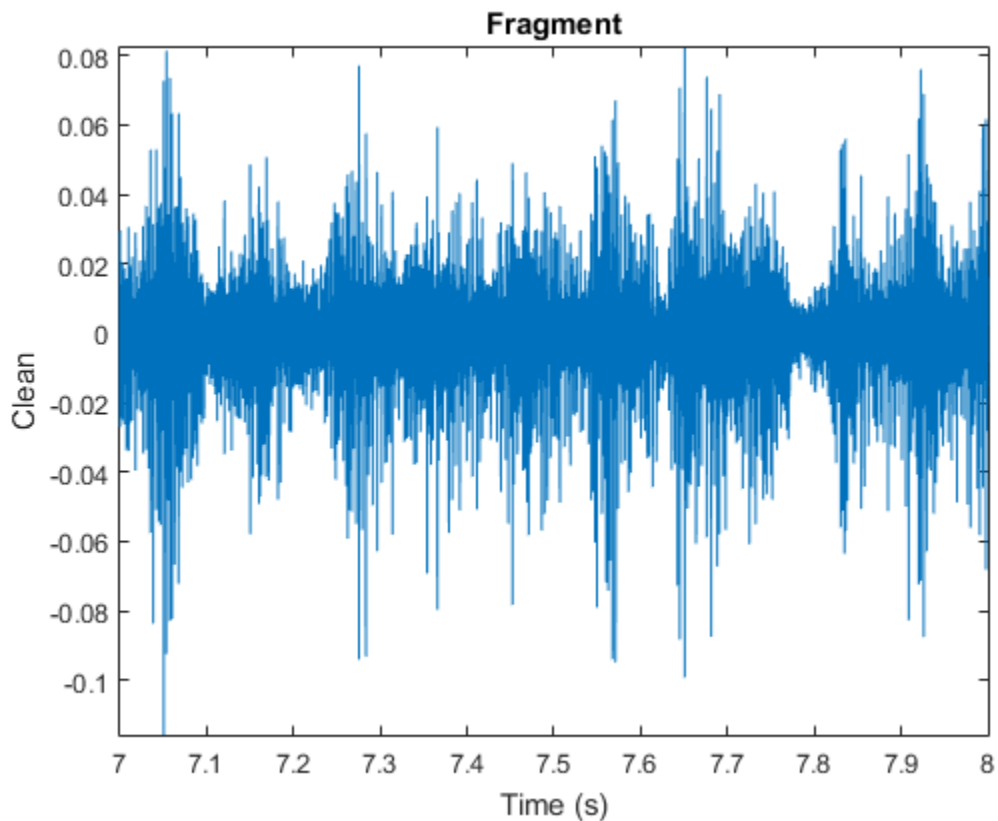
Fragment

Compute and plot the cross-correlation of the full signal and the fragment.

```
[xCorr,lags] = xcorr(Full_sig,Fragment);

plot(lags/Fs,xCorr)
grid
xlabel('Lags (s)')
ylabel('Clean')
axis tight
```
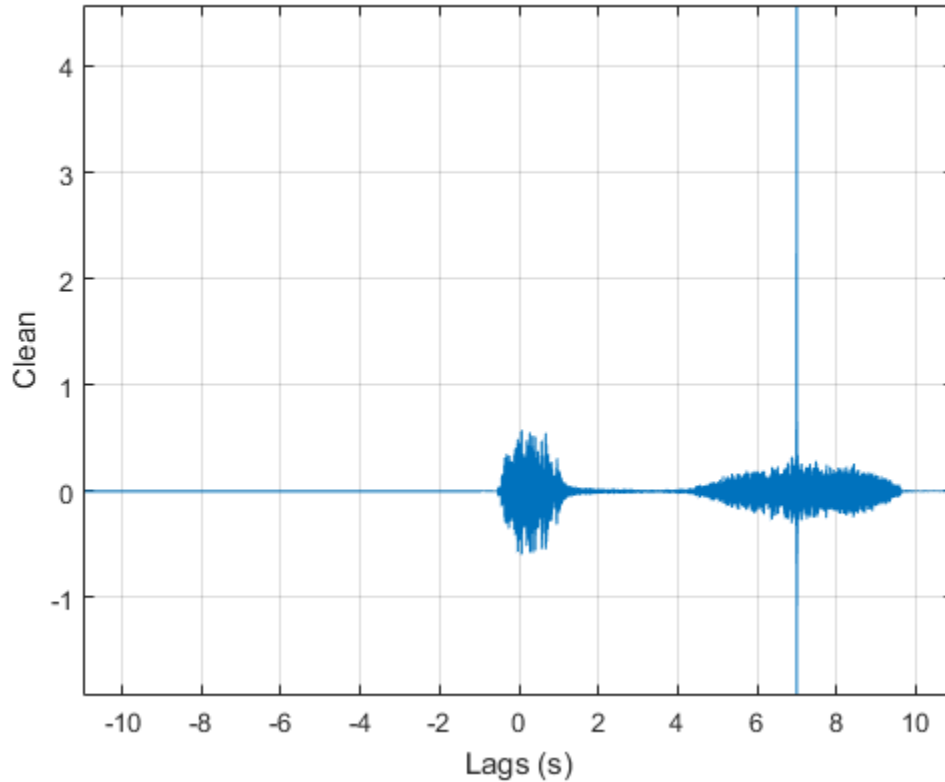
The lag at which the cross-correlation is greatest is the time delay between the signals' starting points. Replot the signal and overlay the fragment.

```
[~,I] = max(abs(xCorr));
maxt = lags(I);

Trial = NaN(size(Full_sig));
Trial(maxt+1:maxt+length(Fragment)) = Fragment;

plot(Time,Full_sig,Time,Trial)
xlabel('Time (s)')
ylabel('Clean')
axis tight
```
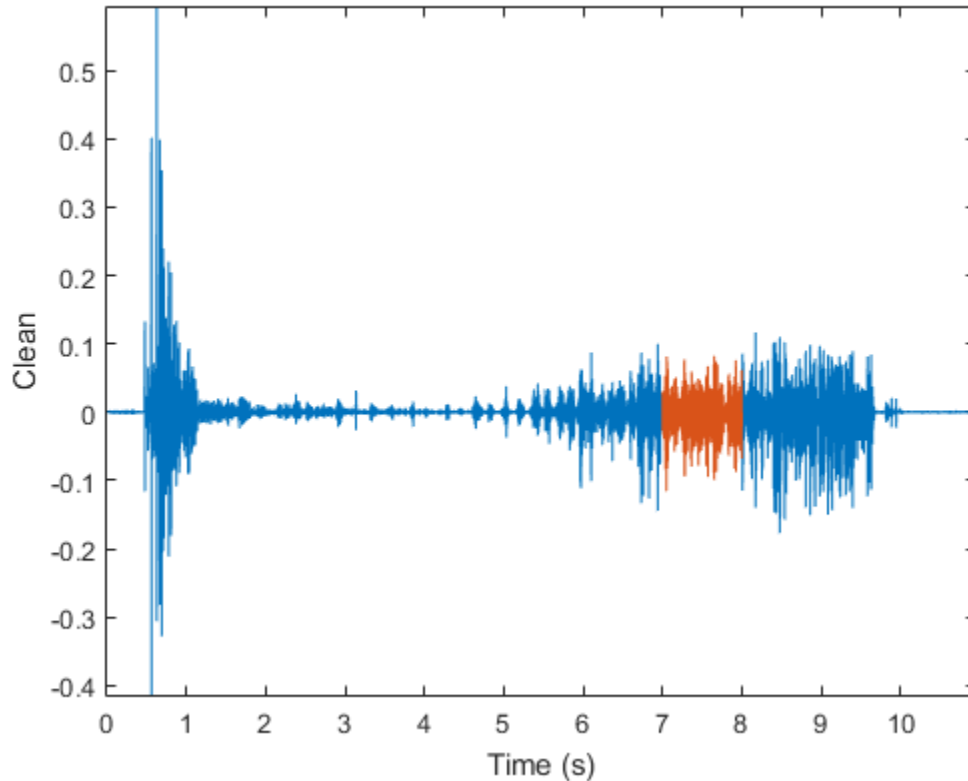
Repeat the procedure, but add noise separately to signal and fragment. The sound cannot be picked out from the noise.

```
NoiseAmp = 0.2*max(abs(Fragment));

Fragment = Fragment+NoiseAmp*randn(size(Fragment));

Full_sig = Full_sig+NoiseAmp*randn(size(Full_sig));

% To hear, type soundsc(Fragment,Fs)

plot(Time,Full_sig,[timeA timeB;timeA timeB],[m m;M M],'r--')
xlabel('Time (s)')
ylabel('Noisy')
```

```matlab
axis tight
```
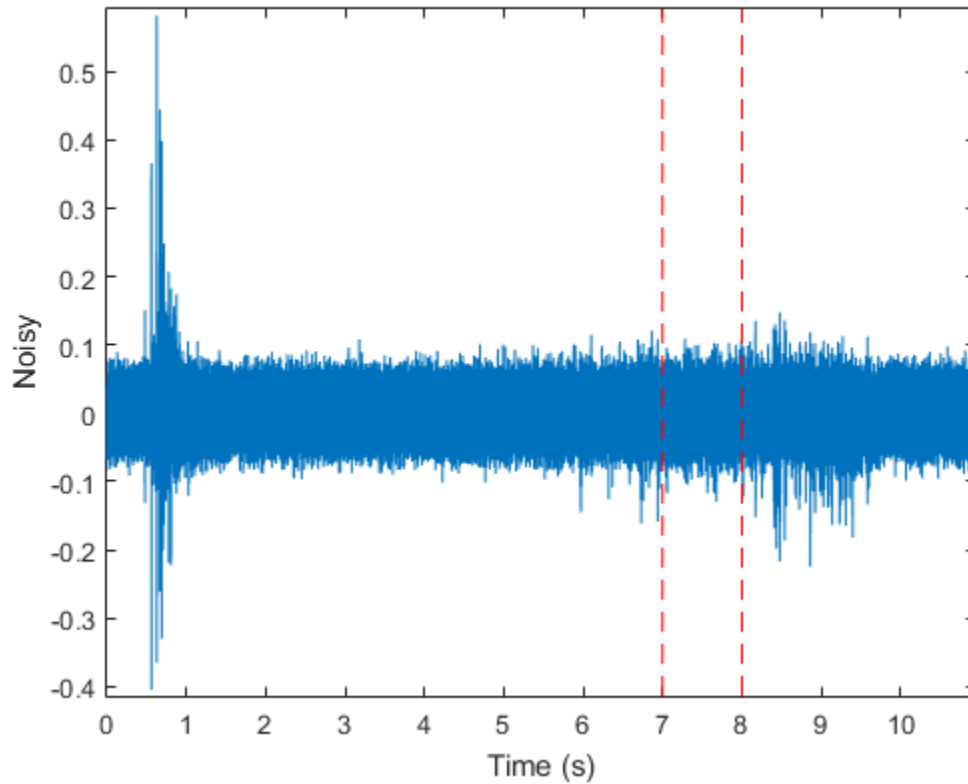

The procedure finds the missing fragment despite the high noise level.

```matlab
[xCorr,lags] = xcorr(Full_sig,Fragment);

plot(lags/Fs,xCorr)
grid
xlabel('Lags (s)')
ylabel('Noisy')
axis tight
```
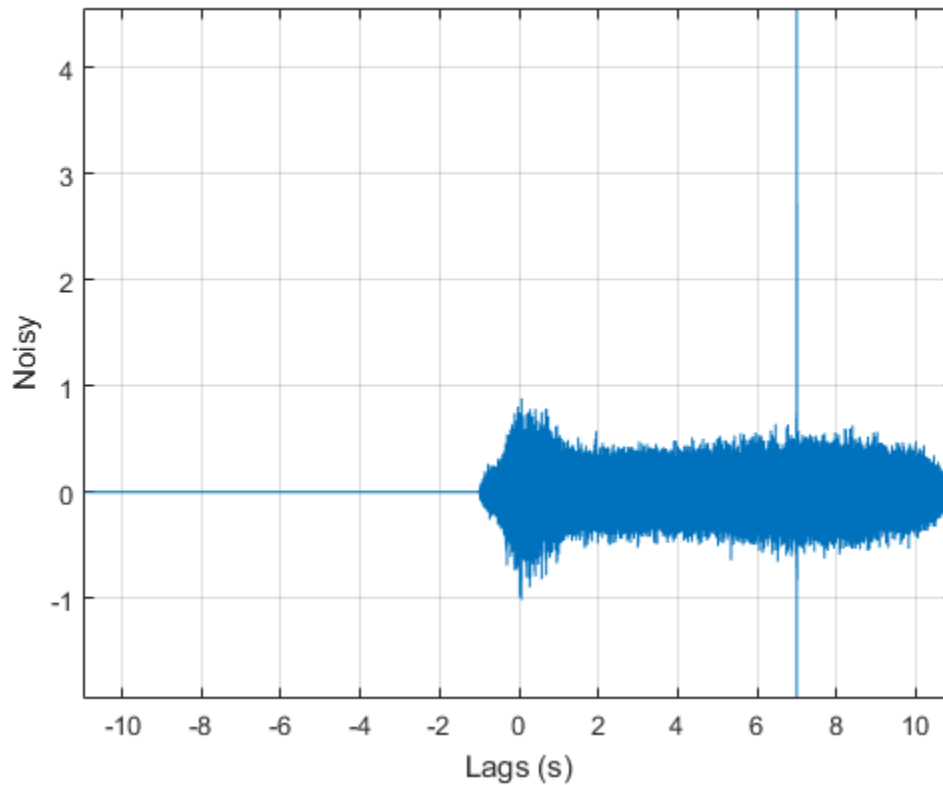
```
[~,I] = max(abs(xCorr));
maxt = lags(I);

Trial = NaN(size(Full_sig));
Trial(maxt+1:maxt+length(Fragment)) = Fragment;

figure
plot(Time,Full_sig,Time,Trial)
xlabel('Time (s)')
ylabel('Noisy')
axis tight
```
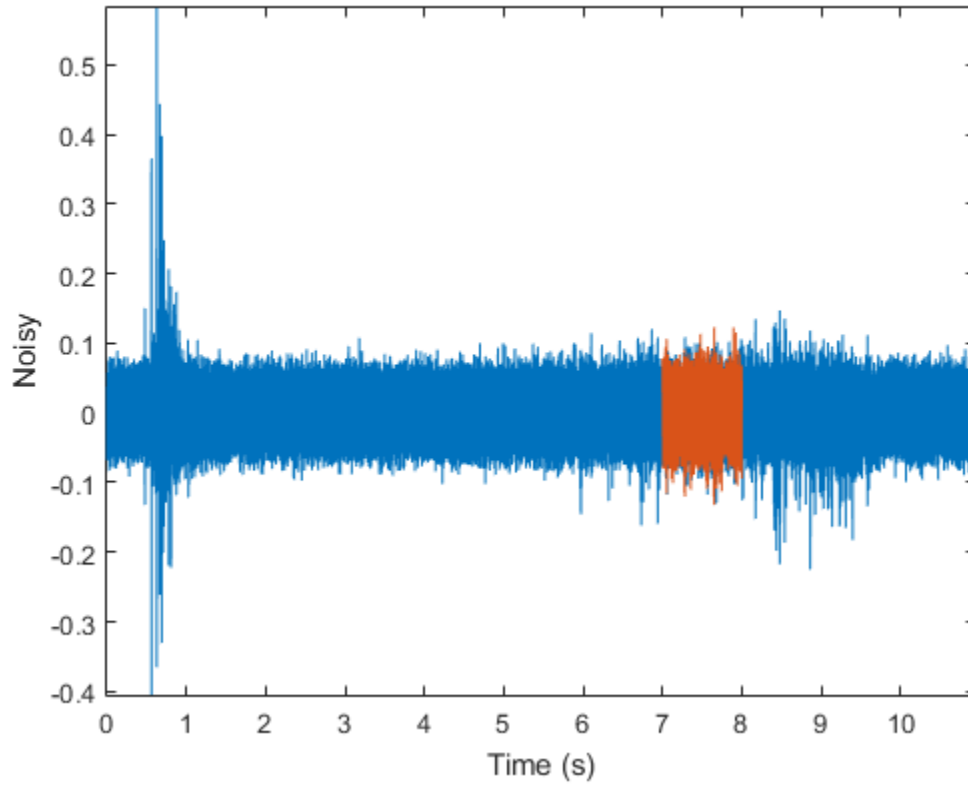
See Also
xcorr

Related Examples
- "Measuring Signal Similarities"

Find Peaks in Data

Use `findpeaks` to find values and locations of local maxima in a set of data.

The file `spots_num.mat` contains the average number of sunspots observed every year from 1749 to 2012. The data are available from NASA.

Find the maxima and their years of occurrence. Plot them along with the data.

```
load(fullfile(matlabroot,'examples','signal','spots_num.mat'))

[pks,locs] = findpeaks(avSpots);

plot(year,avSpots,year(locs),pks,'or')
xlabel('Year')
ylabel('Number')
axis tight
```
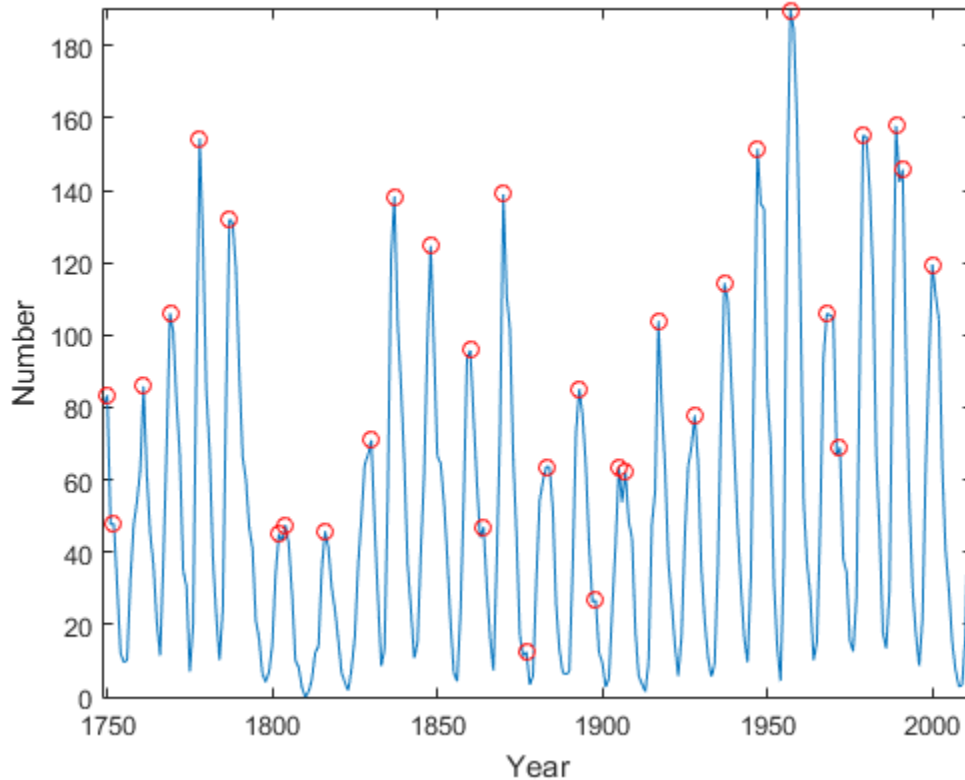
Some peaks are very close to each other. The ones that are not recur at regular intervals. There are roughly five such peaks per 50-year period.

To make a better estimate of the cycle duration, use `findpeaks` again, but this time restrict the peak-to-peak separation to at least six years. Compute the mean interval between maxima.

```
[pks,locs] = findpeaks(avSpots,'MinPeakDistance',6);

plot(year,avSpots,year(locs),pks,'or')
xlabel('Year')
ylabel('Number')
title('Sunspots')
axis tight
```

```matlab
legend('Data','peaks','Location','NorthWest')

cycles = diff(locs);
meanCycle = mean(cycles)
```

```
meanCycle =

    10.8696
```
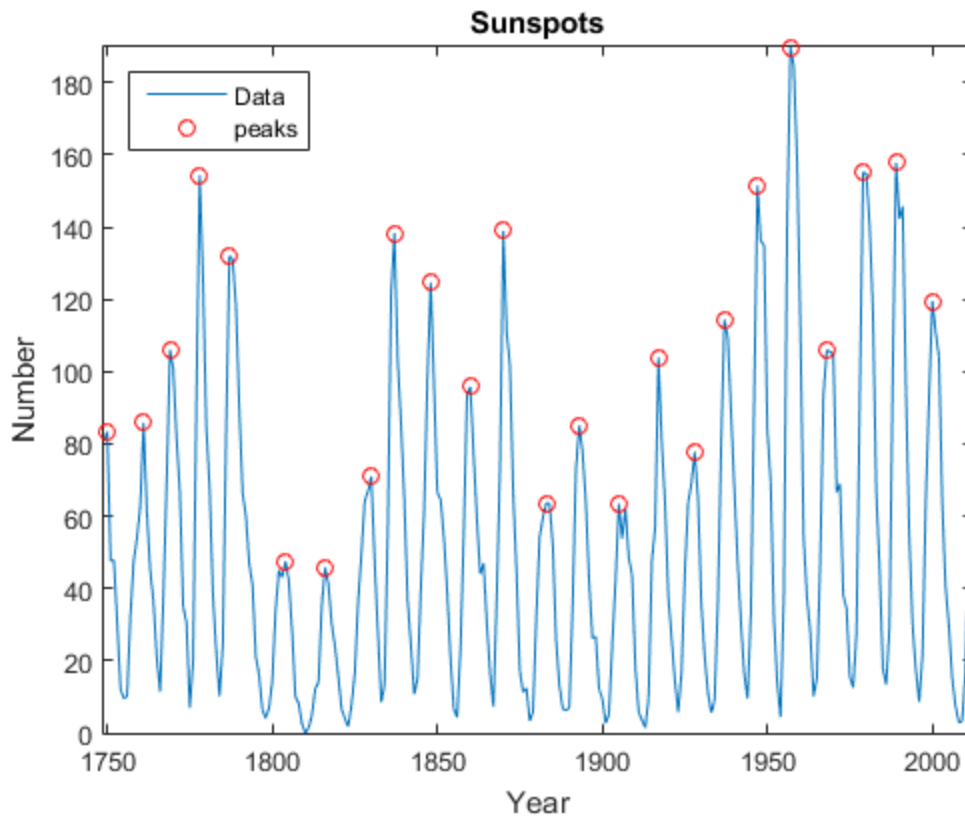

It is well known that solar activity cycles roughly every 11 years. Check by using the Fourier transform. Remove the mean of the signal to concentrate on its fluctuations.

Recall that the sample rate is measured in years. Use frequencies up to the Nyquist frequency.

```
Fs = 1;
Nf = 512;

df = Fs/Nf;
f = 0:df:Fs/2-df;

trSpots = fftshift(fft(avSpots-mean(avSpots),Nf));

dBspots = 20*log10(abs(trSpots(Nf/2+1:Nf)));

yaxis = [20 85];
plot(f,dBspots,1./[meanCycle meanCycle],yaxis)
xlabel('Frequency (year^{-1})')
ylabel('| FFT | (dB)')
axis([0 1/2 yaxis])
text(1/meanCycle + .02,25,['<== 1/' num2str(meanCycle)])
```
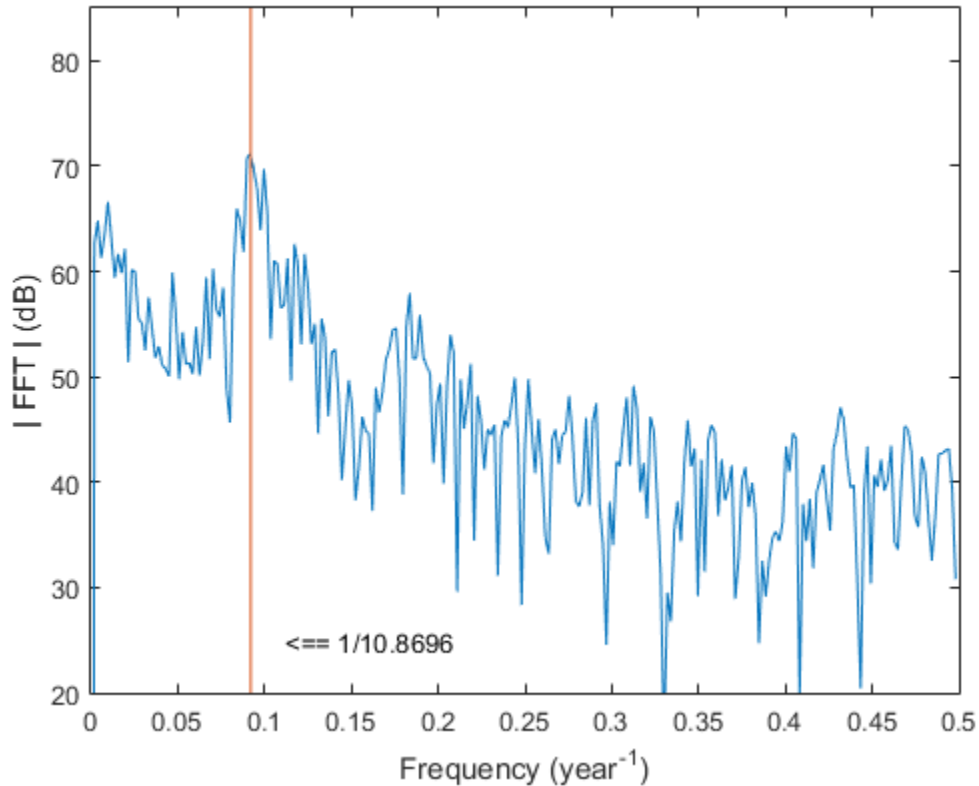
The Fourier transform indeed peaks at the expected frequency, confirming the 11-year conjecture. You also can find the period by locating the highest peak of the Fourier transform.

```
[pk,MaxFreq] = findpeaks(dBspots,'NPeaks',1,'SortStr','descend');

Period = 1/f(MaxFreq)

hold on
plot(f(MaxFreq),pk,'or')
hold off
legend('Fourier transform','1/meanCycle','1/Period')
```

```
Period =

    10.8936
```
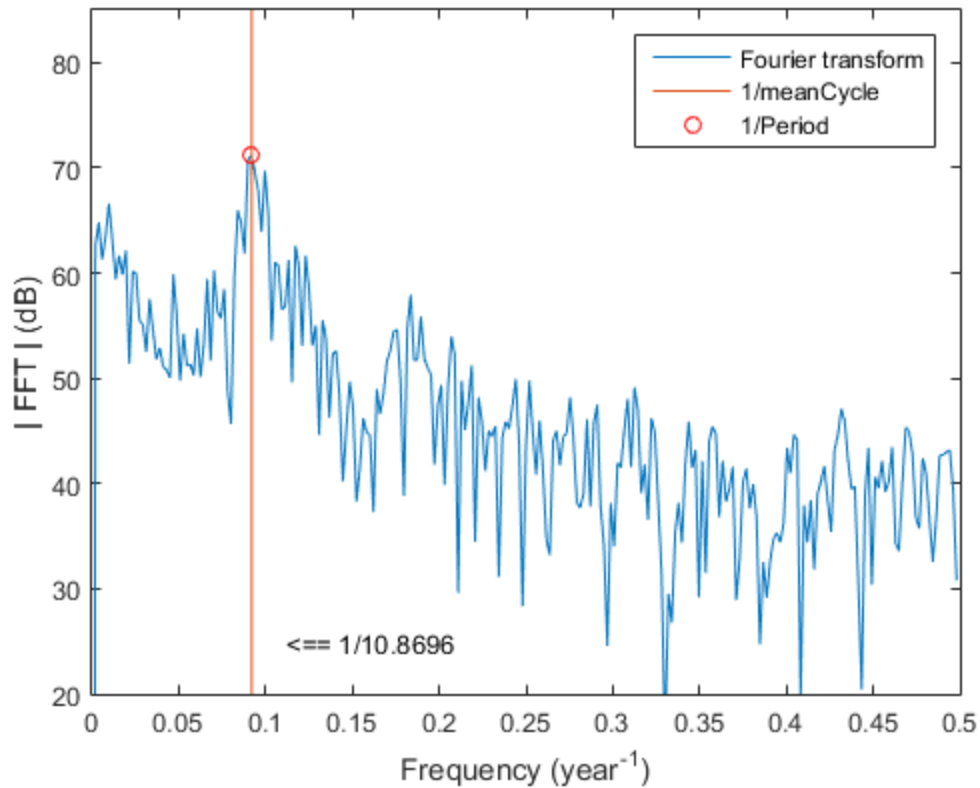

The two estimates coincide quite well.

See Also

`dlmread` | `findpeaks`

Related Examples

- "Peak Analysis"

Find Periodicity Using Autocorrelation

Measurement uncertainty and noise sometimes make it difficult to spot oscillatory behavior in a signal, even if such behavior is expected.

The autocorrelation sequence of a periodic signal has the same cyclic characteristics as the signal itself. Thus, autocorrelation can help verify the presence of cycles and determine their durations.

Consider a set of temperature data collected by a thermometer inside an office building. The device takes a reading every half hour for four months. Load the data and plot it. Measure time in days and temperature in degrees Celsius. Subtract the mean to concentrate on temperature fluctuations.

```
load officetemp

tempC = (temp-32)*5/9;

tempnorm = tempC-mean(tempC);

fs = 2*24;
t = (0:length(tempnorm) - 1)/fs;

plot(t,tempnorm)
xlabel('Time (days)')
ylabel('Temperature ( {}^\circC )')
axis tight
```
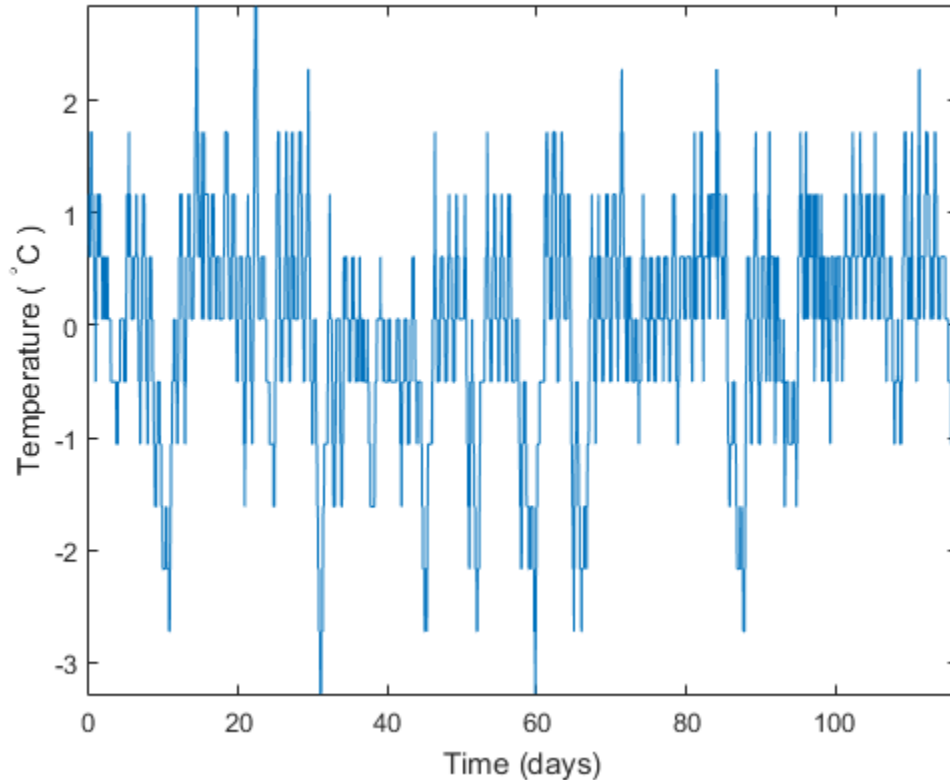
The temperature does seem to oscillate, but the lengths of the cycles cannot be read out easily.

Compute the autocorrelation of the temperature such that it is unity at zero lag. Restrict the positive and negative lags to three weeks. Note the double periodicity of the signal.

```
[autocor,lags] = xcorr(tempnorm,3*7*fs,'coeff');

plot(lags/fs,autocor)
xlabel('Lag (days)')
ylabel('Autocorrelation')
axis([-21 21 -0.4 1.1])
```
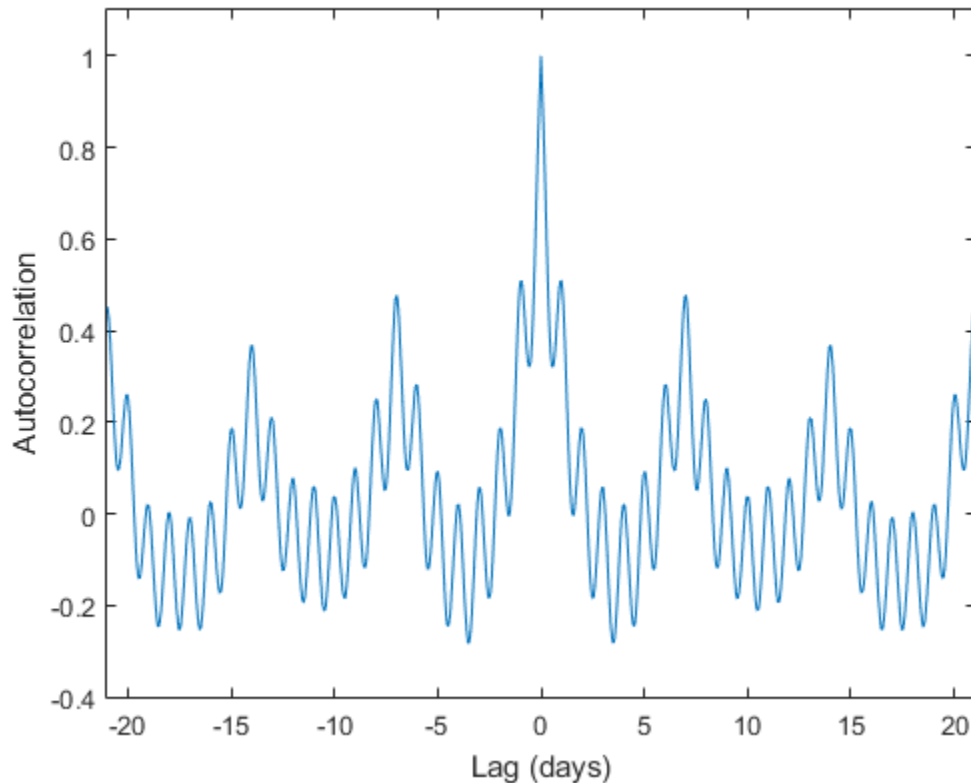
Determine the short and long periods by finding the peak locations and determining the average time differences between them.

To find the long period, restrict `findpeaks` to look for peaks separated by more than the short period and with a minimum height of 0.3.

```
[pksh,lcsh] = findpeaks(autocor);
short = mean(diff(lcsh))/fs

[pklg,lclg] = findpeaks(autocor, ...
    'MinPeakDistance',ceil(short)*fs,'MinPeakheight',0.3);
long = mean(diff(lclg))/fs

hold on
```

```
pks = plot(lags(lcsh)/fs,pksh,'or', ...
    lags(lclg)/fs,pklg+0.05,'vk');
hold off
legend(pks,[repmat('Period: ',[2 1]) num2str([short;long],0)])
axis([-21 21 -0.4 1.1])


short =

    1.0021


long =

    6.9896
```
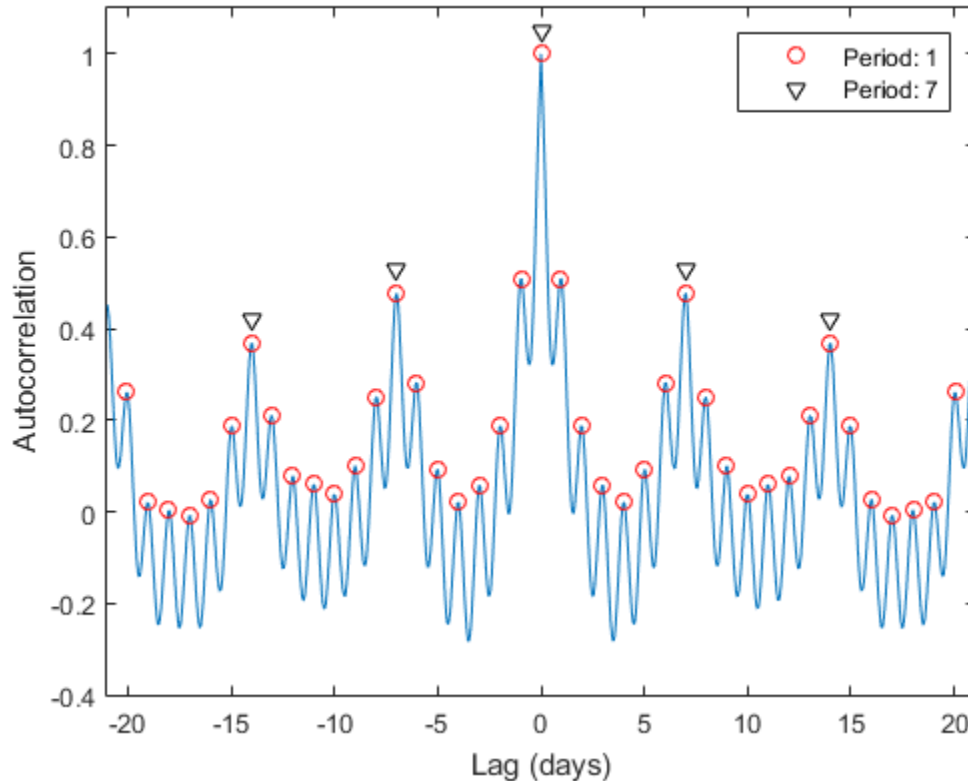
To a very good approximation, the autocorrelation oscillates both daily and weekly. This is to be expected, since the temperature in the building is higher when people are at work and lower at nights and on weekends.

See Also
findpeaks | xcorr

Related Examples
- "Find Periodicity Using Frequency Analysis"

Extract Features of a Clock Signal

How sharply does an on/off signal turn on and off? How often and for how long is it activated? Determine all those characteristics for the output of a clock.

Load the signal and plot it. The time is measured in seconds and the level in volts.

```
load(fullfile(matlabroot,'examples','signal','clock.mat'))

plot(tclock,clocksig)
xlabel('Time (s)')
ylabel('Level (V)')
```
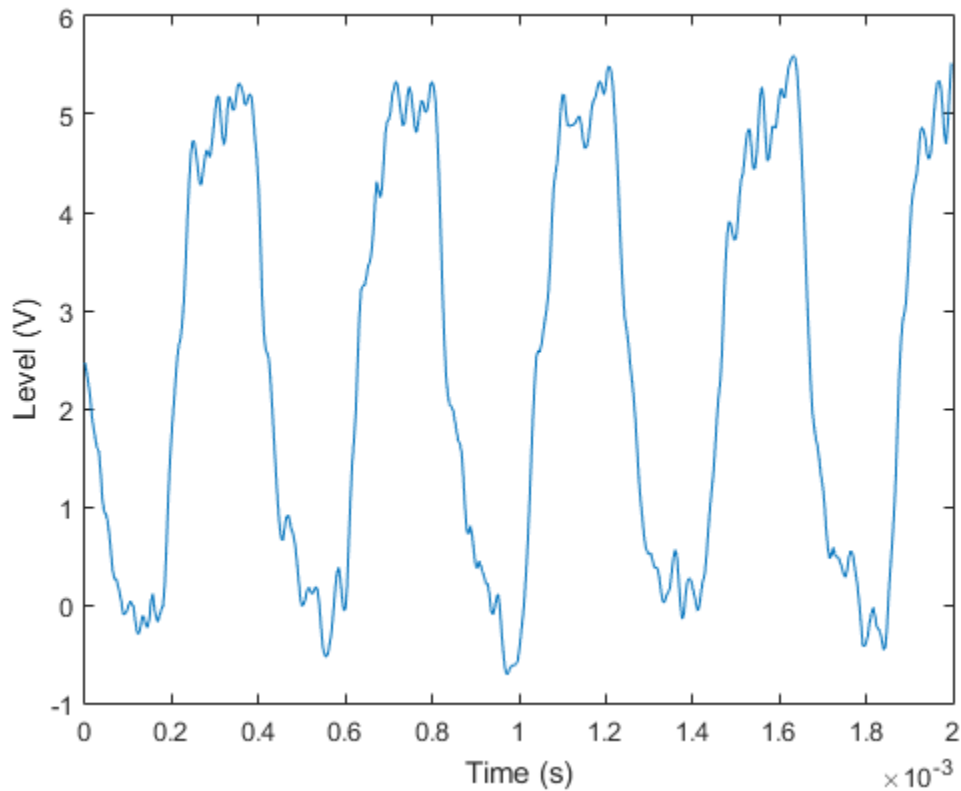
Use `statelevels` to find the lower and upper levels of the signal by means of a histogram. If you do not specify an output, the function plots the signal, marks the levels, and displays the histogram.

```
levels = statelevels(clocksig)

statelevels(clocksig);


levels =

    0.0138    5.1848
```
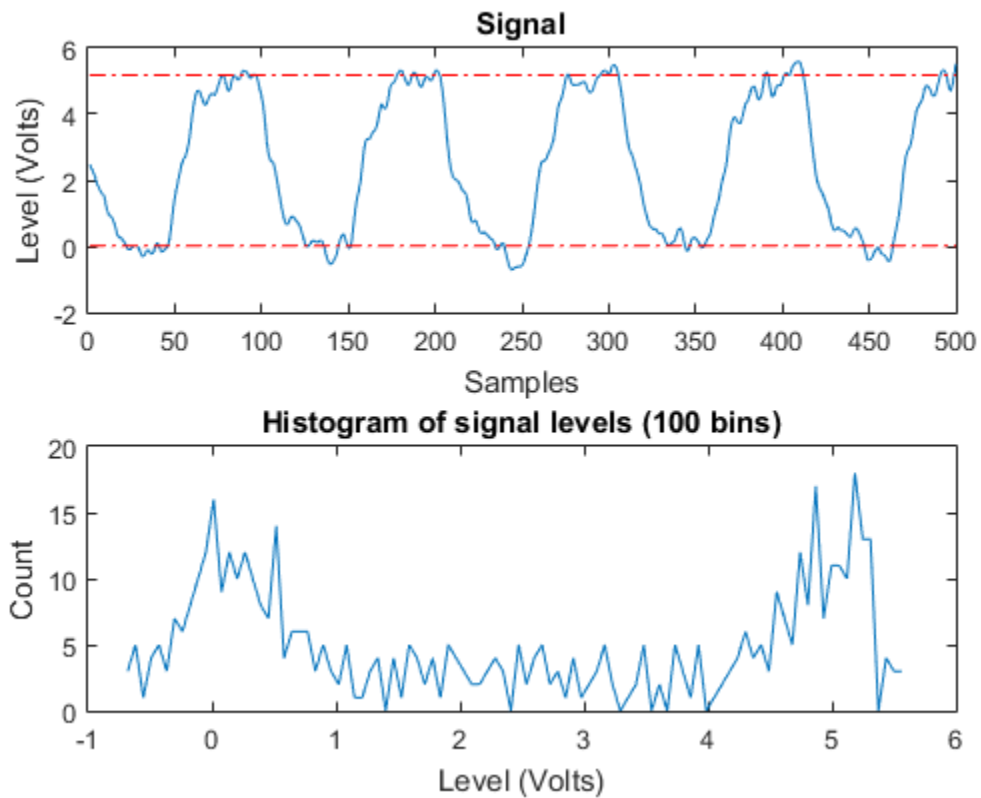
Determine how fast the signal rises at each transition. `risetime` uses the lower and upper levels found by `statelevels`. It defines the rise time as the time it takes the signal to rise from 10% to 90% of the difference between the levels.

```
[Rise,LoTime,HiTime,LoLev,HiLev] = risetime(clocksig,tclock);

Levels = [LoLev HiLev; (levels(2)-levels(1))*[0.1 0.9]+levels(1)]


Levels =

    0.5309    4.6677
    0.5309    4.6677
```

If you call `risetime` without outputs, the function draws an annotated plot of the signal. The rise times are shaded, the crossing points are marked, and the levels are displayed. You can use the time vector or the sample rate as input.

```
risetime(clocksig,Fs);
```

overshoot and undershoot show how far the signal deviates from the state levels at each transition. The results are expressed as percentages of the difference between the levels. Further outputs give the actual times and signal values.

```
overshoot(clocksig,Fs);

[pctgs,values,times] = undershoot(clocksig,Fs);

hold on
text(1.1e-3,2,'     Undershoot','Background','w','Edge','k')
plot([times;1.17e-3],[values;2],'^m')
hold off
```
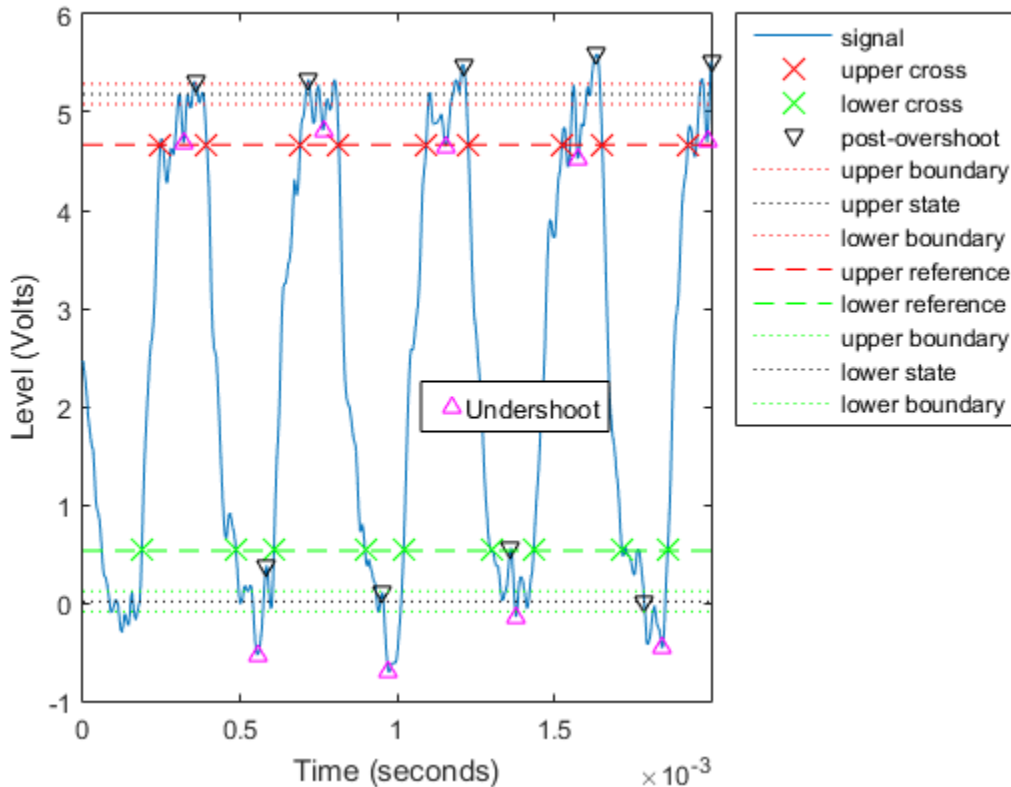
Determine how fast the signal falls using `falltime`. You can set the state levels and the percentage reference levels manually. You can do the same with `risetime`.

```
falltime(clocksig,tclock, ...
    'PercentReferenceLevels',[30 80],'StateLevels',[0 5]);
```
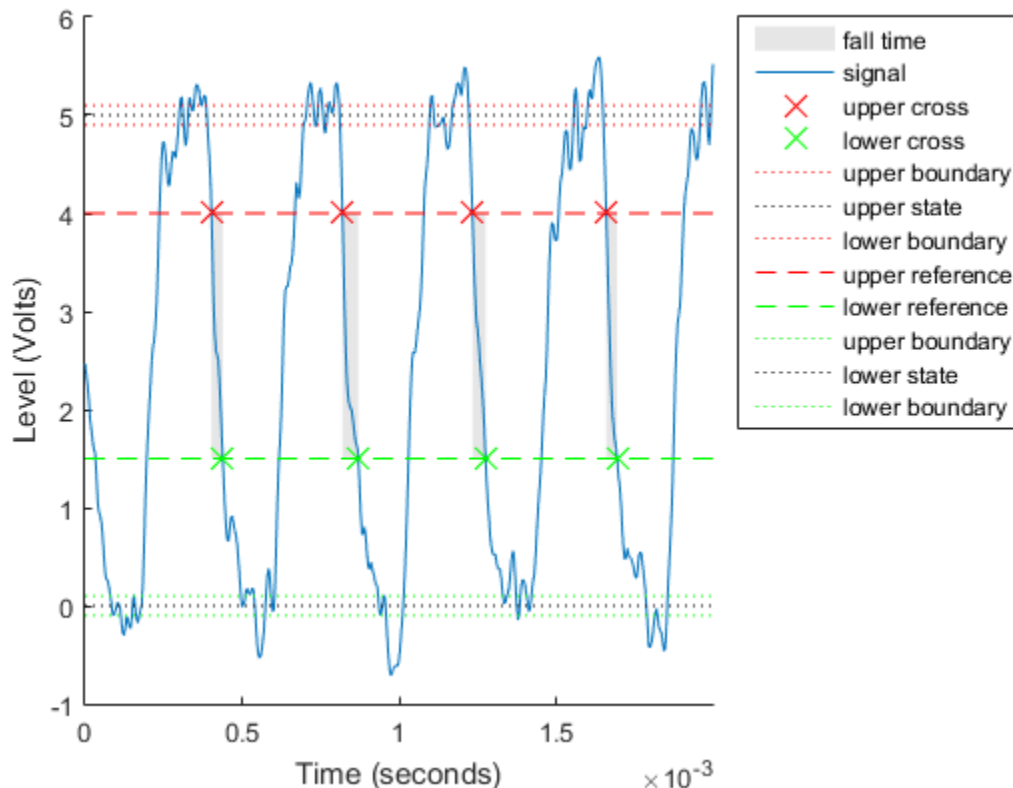
Find the period of the signal. By default, the period is defined as the time elapsed between consecutive rising crossings of the reference level halfway between the state levels. You can change the polarity of the crossings, specify the state levels, or adjust the reference level.

```
per = pulseperiod(clocksig,tclock)

pulseperiod(clocksig,Fs,'Polarity','negative','MidPct',25);


per =

   1.0e-03 *
```

```
        0.4143
        0.4200
        0.4188
        0.4111
```
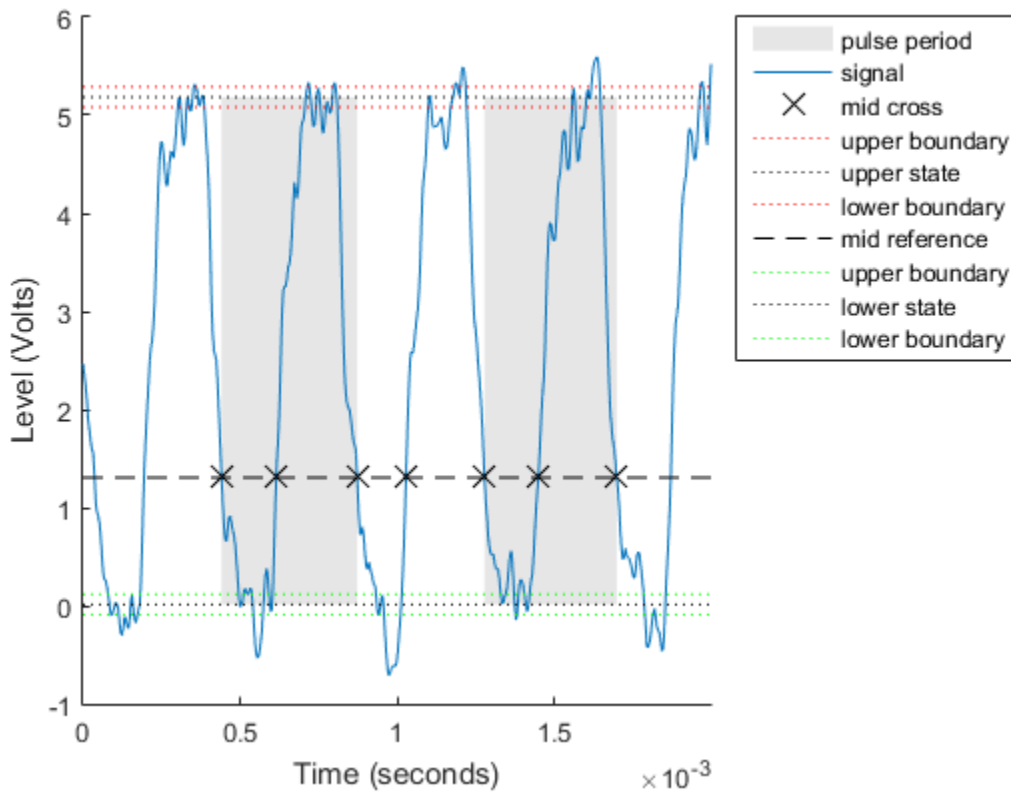

The duty cycle is the ratio of pulse width to pulse period. Determine it directly or using a dedicated function.

```
dut = dutycycle(clocksig,Fs);

wdt = pulsewidth(clocksig,Fs);

compare = [wdt./per dut]
```

```
compare =

    0.4862    0.4862
    0.4756    0.4756
    0.4871    0.4871
    0.4886    0.4886
```

See Also
`dutycycle` | `falltime` | `overshoot` | `pulseperiod` | `pulsewidth` | `risetime` | `slewrate` | `statelevels` | `undershoot`

Related Examples
- "Measurement of Pulse and Transition Characteristics"

Find Periodicity in a Categorical Time Series

This example shows how to perform spectral analysis of categorical-valued time-series data. The spectral analysis of categorical-valued time series is useful when you are interested in cyclic behavior of data whose values are not inherently numerical. This example reproduces in part the analysis reported in Stoffer et al. (1988). The data are taken from Stoffer, Tyler, and Wendt (2000).

The data are from a study of sleep states in newborn children. A pediatric neurologist scored an infant's electroencephalographic (EEG) recording every minute for approximately two hours. The neurologist categorized the infant's sleep state into one of the following:

- qt - Quiet sleep, trace alternant
- qh - Quiet sleep, high voltage
- tr - Transitional sleep
- al - Active sleep, low voltage
- ah - Active sleep, high voltage
- aw - Awake

Enter the data. The infant was never awake during the EEG recording.

```
data = {'ah','ah','ah','ah','ah','ah','ah','ah','tr','ah','tr','ah', ...
    'ah','qh','qt','qt','qt','qt','qt','tr','qt','qt','qt','qt','qt', ...
    'qt','qt','qt','qt','qt','tr','al','al','al','al','al','tr','ah', ...
    'al','al','al','al','al','ah','ah','ah','ah','ah','ah','ah','tr', ...
    'tr','ah','ah','ah','ah','tr','tr','tr','qh','qh','qt','qt','qt', ...
    'qt','qt','qt','qt','qt','qt','qt','qt','qt','qt','qt','qt','qt', ...
    'qt','qt','tr','al','al','al','al','al','al','al','al','al', ...
    'al','al','al','al','al','al','al','ah','ah','ah','ah','ah','ah', ...
    'ah','ah','ah','tr'};

lend = length(data);
t = 1:lend;
```

The easiest way to analyze categorical-valued time series data for cyclic patterns involves assigning numerical values to the categories. There are at least two meaningful ways of assigning values to the infant's sleep states. First, note that you can order the six states from 1 to 6. This assignment makes sense along the scale of least active to most active.

Replace the six sleep states with their numerical equivalents and plot the data.

```
states = ['qt';'qh';'tr';'al';'ah';'aw'];
levelssix = [1 2 3 4 5 6];

for nn = 1:6
    datasix(strcmp(data,states(nn,:))) = levelssix(nn);
end

plot(t,datasix)
axis([0 lend 0 6])
ax = gca;
ax.YTick = [1 2 4 5];
grid
xlabel('Minutes')
ylabel('Sleep State')
```
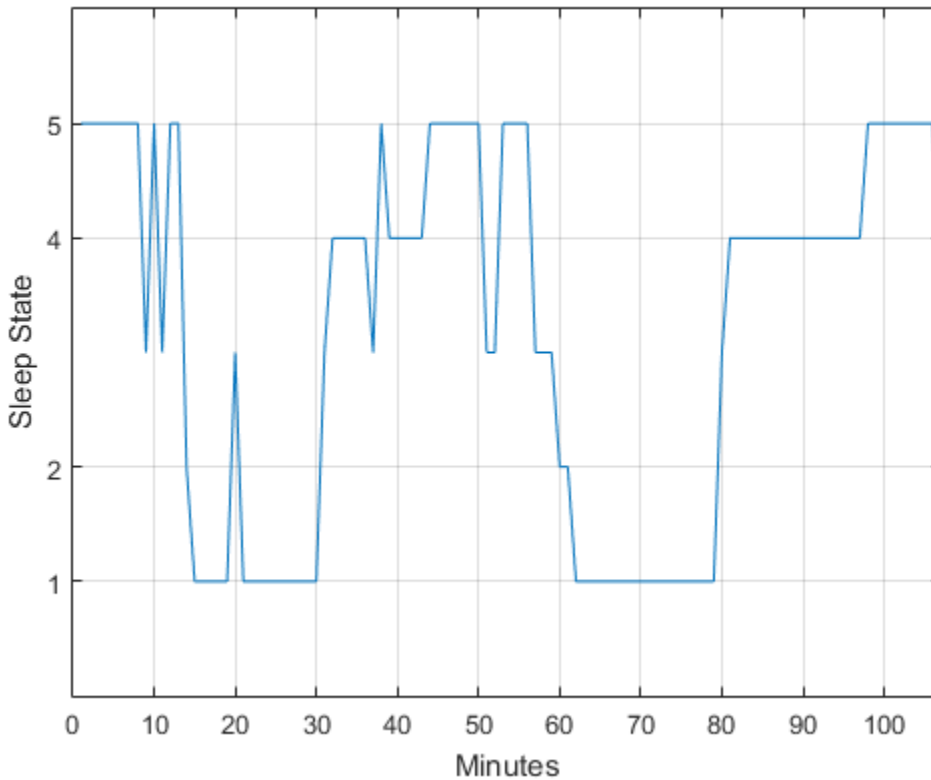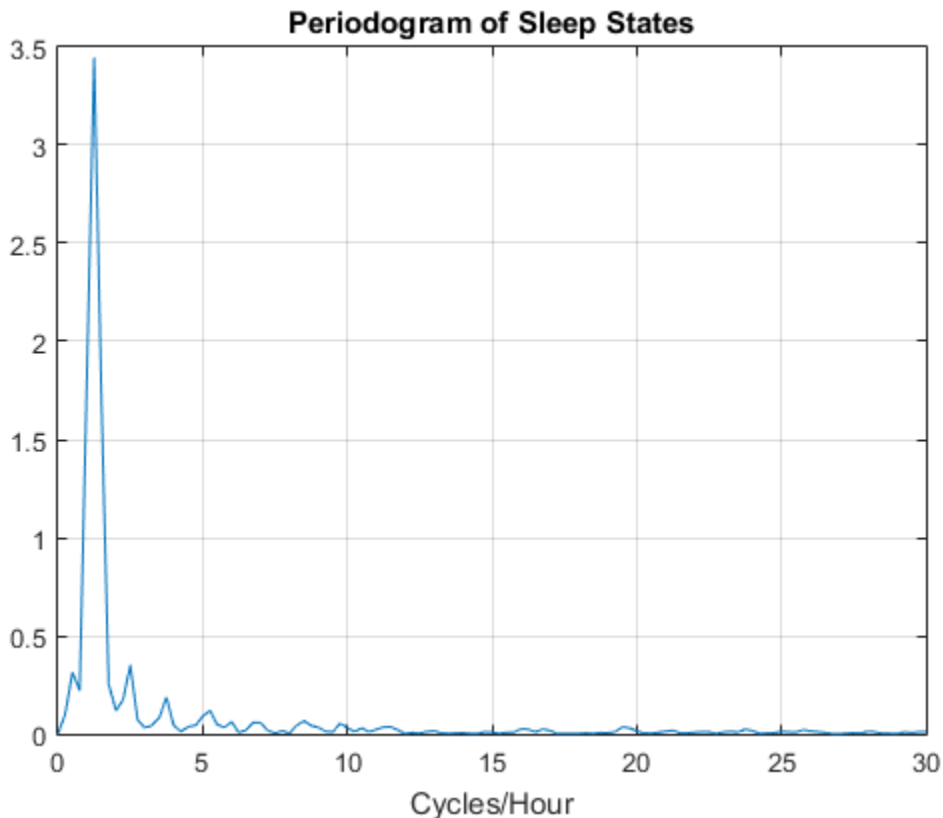
The data exhibit cyclic behavior when you focus on the transitions between the quietest states (1 and 2) and the most active ones (4 and 5). To determine the cycle of that behavior, use spectral analysis. Recall that the sleep states are assigned in one-minute intervals. Sampling the data in one-minute intervals is equivalent to sampling the data 60 times per hour.

```
Fs = 60;
[Pxx,F] = periodogram(detrend(datasix,0),[],240,Fs);

plot(F,Pxx)
grid
xlabel('Cycles/Hour')
title('Periodogram of Sleep States')
```

The spectral analysis shows a clear peak indicating a dominant oscillation, or cycle in the data. Determine the frequency of the peak.

```
[maxval,maxidx] = max(Pxx);
Fsix = F(maxidx)
```

```
Fsix =

    1.2500
```

The infant's sleep states exhibit cyclic behavior with a frequency of approximately 1.25 cycles/hour.

Instead of assigning the sleep states the values 1 to 6, repeat the analysis focusing only on the distinction between quiet and active sleep. Assign the quiet states, `qt` and `qh`, the value 1. Assign the transitional state, `tr`, the value 2. Finally, assign the two active sleep states, `al` and `ah`, the value 3. For completeness, assign the awake state, `aw`, the value 4, even though the state does not occur in the data.

```
states = ['qt';'qh';'tr';'al';'ah';'aw'];
levelsfou = [1 1 2 3 3 4];

for nn = 1:6
    datafou(strcmp(data,states(nn,:))) = levelsfou(nn);
end

plot(t,datafou)
axis([0 lend 0 4])
ax = gca;
ax.YTick = [1 2 3];
grid
xlabel('Minutes')
ylabel('Sleep State')
```
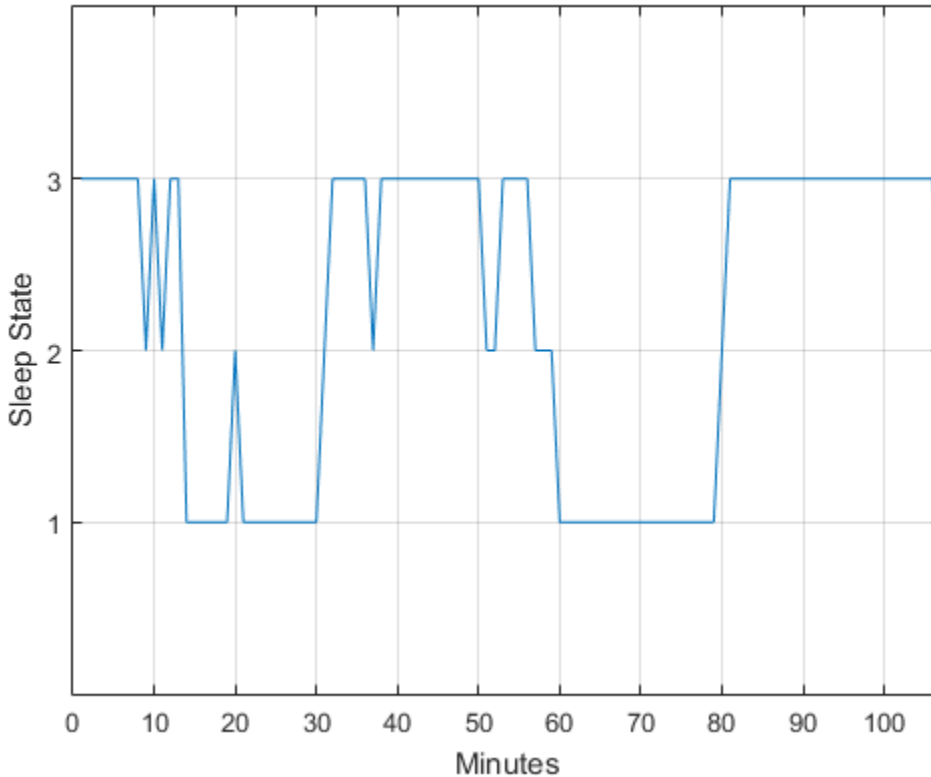
With this rule of assignment between the sleep states and the values 1 to 3, the cyclic behavior of the data is clearer. Repeat the spectral analysis with the new assignment.
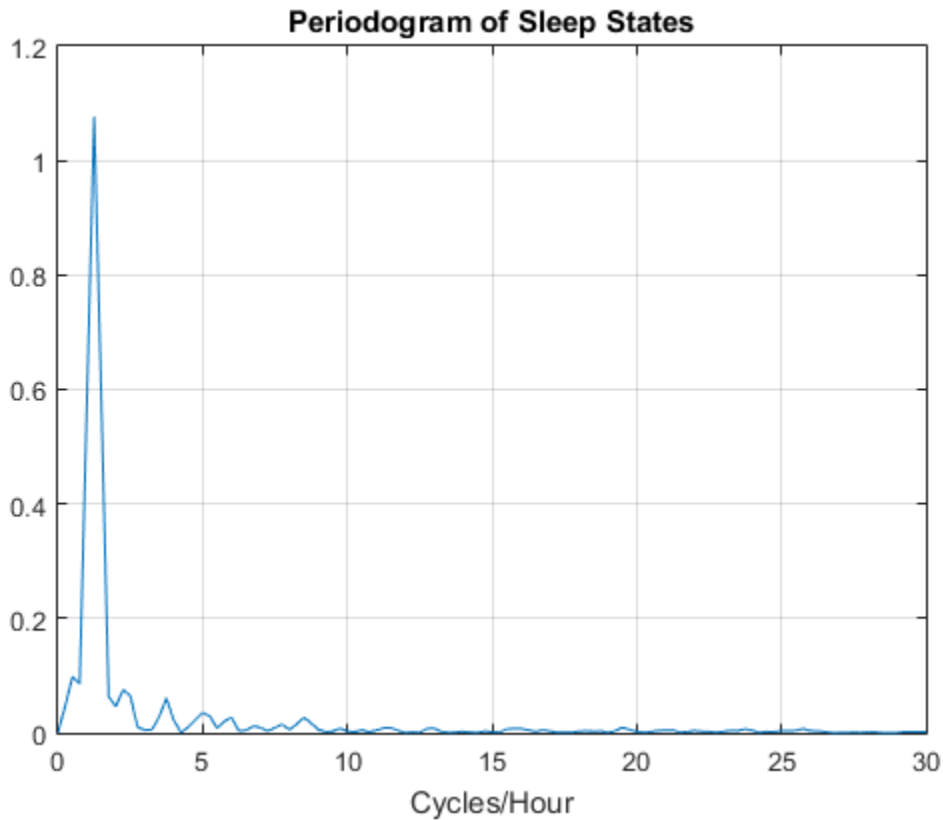
```
[Pxx,F] = periodogram(detrend(datafou,0),[],240,Fs);

plot(F,Pxx)
grid
xlabel('Cycles/Hour')
title('Periodogram of Sleep States')

[maxval,maxidx] = max(Pxx);
F(maxidx)
```

```
ans =

    1.2500
```

Periodogram of Sleep States

The new assignment has not changed the conclusion. The data show a dominant oscillation at 1.25 cyles/hour. Because the mapping between the sleep states and the integers representing those states was consistent, the analysis and conclusions were not affected. Based on a spectral analysis of this categorical data, you conclude that the infant's sleep state cycles between quiet and active sleep approximately once every hour.

References

Stoffer, David S., Mark S. Scher, Gale A. Richardson, Nancy L. Day, and Patricia A. Coble. "A Walsh-Fourier Analysis of the Effects of Moderate Maternal Alcohol Consumption on Neonatal Sleep-State Cycling." *Journal of the American Statistical Association.* Vol. 83, 1988, pp. 954-963.

Stoffer, David S., D. E. Tyler, and D. A. Wendt. "The Spectral Envelope and Its Applications." *Statistical Science.* Vol. 15, 2000, pp. 224-253.

See Also

detrend | periodogram

Compensate for the Delay Introduced by an FIR Filter

Filtering a signal introduces a delay. This means that the output signal is shifted in time with respect to the input. This example shows you how to counteract this effect.

Finite impulse response filters often delay all frequency components by the same amount. This makes it easy to correct for the delay by shifting the signal in time.

Take an electrocardiogram reading sampled at 500 Hz for 1 s. Add random noise. Reset the random number generator for reproducibility.

```
Fs = 500;
N = 500;
rng default

xn = ecg(N)+0.25*randn([1 N]);
tn = (0:N-1)/Fs;
```

Remove some of the noise with a filter that stops frequencies above 75 Hz. Use `designfilt` to design a filter of order 70.

```
nfilt = 70;
Fst = 75;

d = designfilt('lowpassfir','FilterOrder',nfilt, ...
               'CutoffFrequency',Fst,'SampleRate',Fs);
```

Filter the signal and plot it. The result is smoother than the original, but lags behind it.

```
xf = filter(d,xn);

plot(tn,xn)
hold on, plot(tn,xf,'-r','linewidth',1.5), hold off
title 'Electrocardiogram'
xlabel 'Time (s)', legend('Original Signal','Filtered Signal')
```
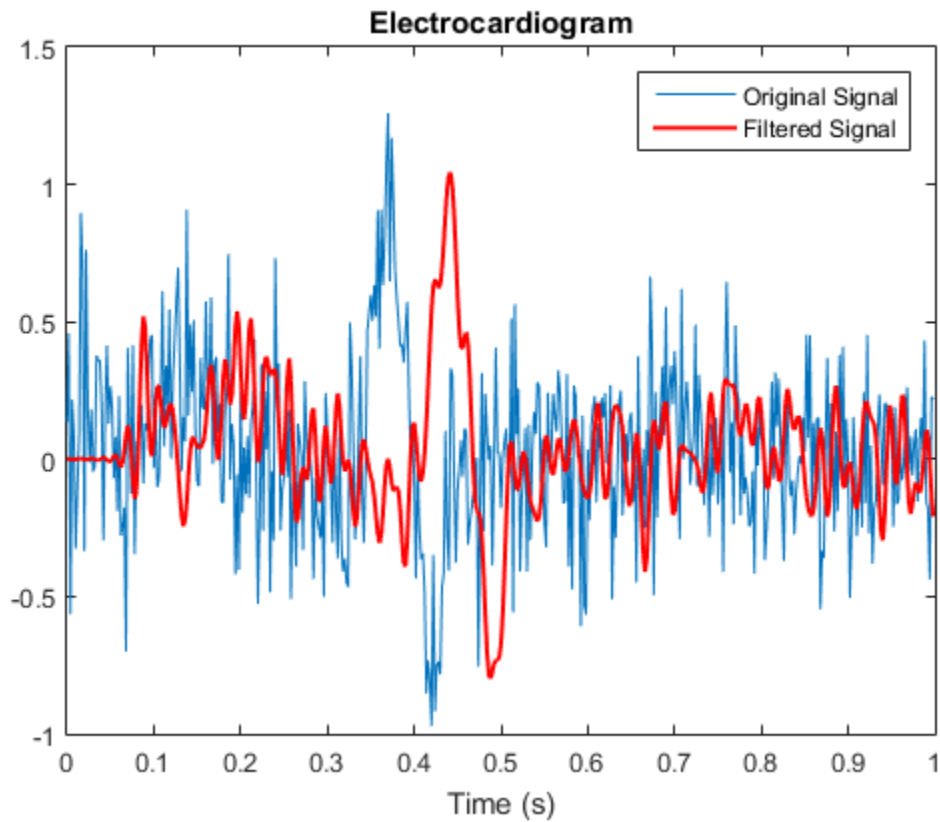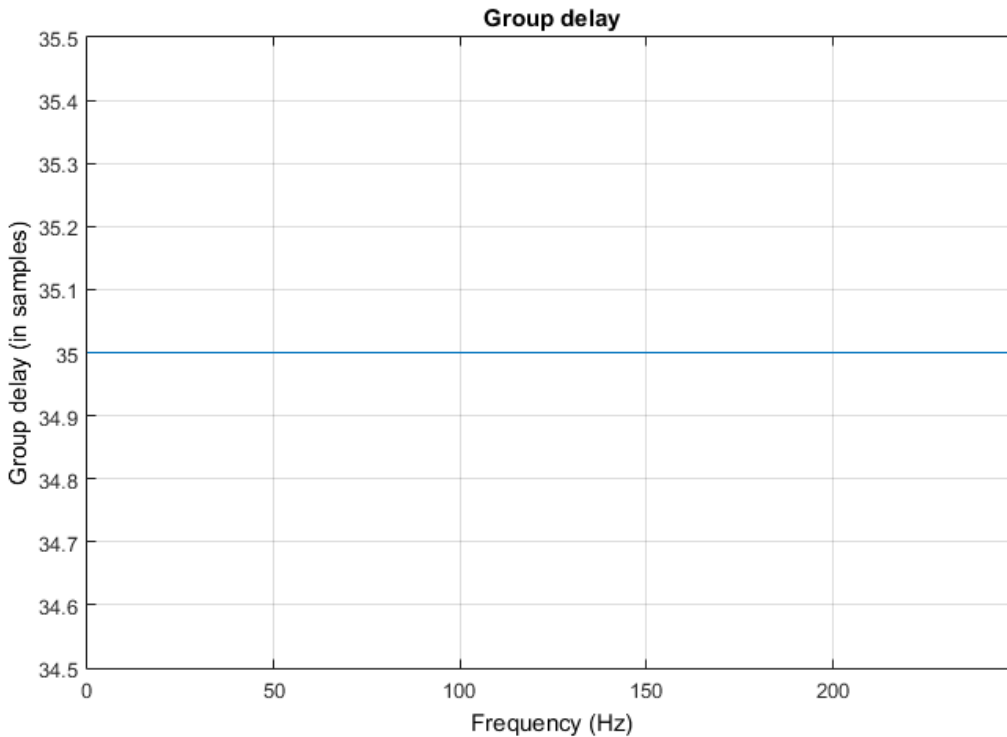
Use `grpdelay` to check that the delay caused by the filter equals half the filter order.

```
grpdelay(d,N,Fs)
delay = mean(grpdelay(d))


delay =

   35
```

Shift the filtered signal to line up the data. Remove its first `delay` samples. Remove the last `delay` samples of the original and of the time vector.

```
tt = tn(1:end-delay);
sn = xn(1:end-delay);

sf = xf;
sf(1:delay) = [];
```

Plot the signals and verify that they are aligned.

```
plot(tt,sn)
hold on, plot(tt,sf,'-r','linewidth',1.5), hold off
title 'Electrocardiogram'
xlabel('Time (s)'), legend('Original Signal','Filtered Shifted Signal')
```
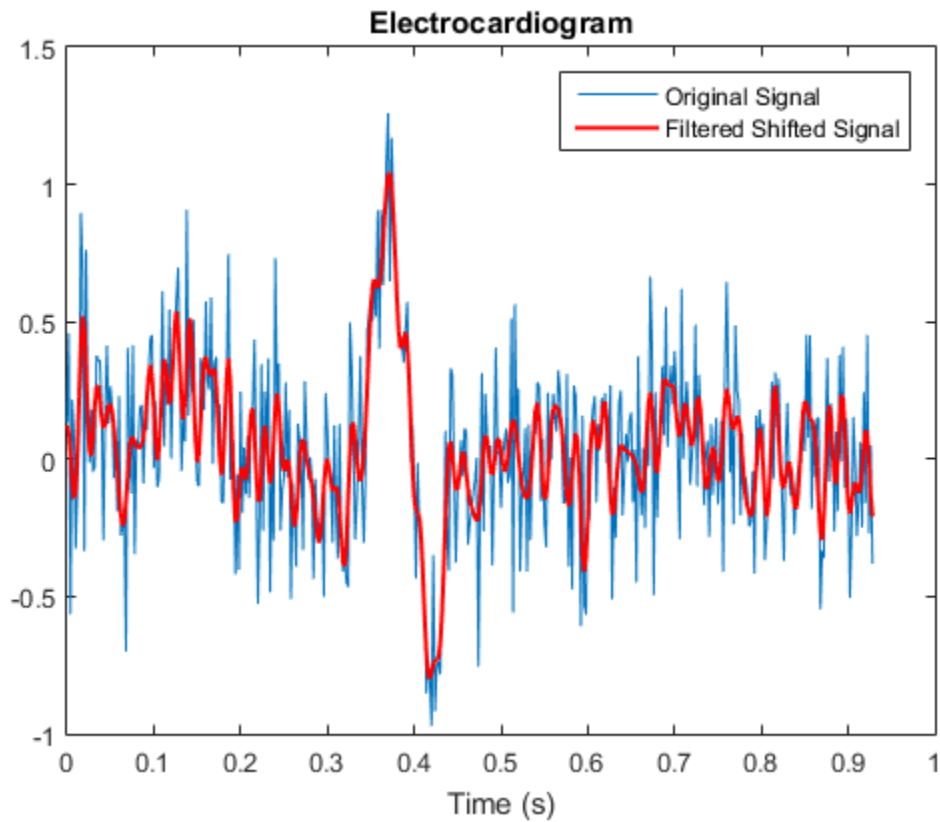
See Also

designfilt | filter | filtfilt | grpdelay

Related Examples

- "Compensate for the Delay Introduced by an IIR Filter"
- "Practical Introduction to Digital Filtering"

Compensate for the Delay Introduced by an IIR Filter

Filtering a signal introduces a delay. This means that the output signal is shifted in time with respect to the input.

Infinite impulse response filters delay some frequency components more than others. They effectively distort the input signal. The function `filtfilt` compensates for the delays introduced by such filters, and thus corrects for filter distortion. This "zero-phase filtering" results from filtering the signal in the forward and backward directions.

Take an electrocardiogram reading sampled at 500 Hz for 1 s. Add random noise.

```
Fs = 500;
N = 500;

rng default
xn = ecg(N) + 0.2*randn([1 N]);
tn = (0:N-1)/Fs;
```

Remove some of the noise with a filter that stops frequencies above 75 Hz. Specify a 7th-order IIR filter with 1 dB of passband ripple and 60 dB of stopband attenuation.

```
Nf = 7;
Fp = 75;
Ap = 1;
As = 60;

d = designfilt('lowpassiir','FilterOrder',Nf,'PassbandFrequency',Fp, ...
    'PassbandRipple',Ap,'StopbandAttenuation',As,'SampleRate',Fs);
```

Filter the signal. The filtered signal is cleaner than the original, but lags in time with respect to it. It is also distorted due to the nonlinear phase of the filter. Zoom in close to the peak.

```
xfilter = filter(d,xn);

plot(tn,xn,tn,xfilter)

title 'Electrocardiogram'
xlabel 'Time (s)', legend('Original Signal','Filtered Signal')
axis([0.25 0.55 -1 1.5])
```
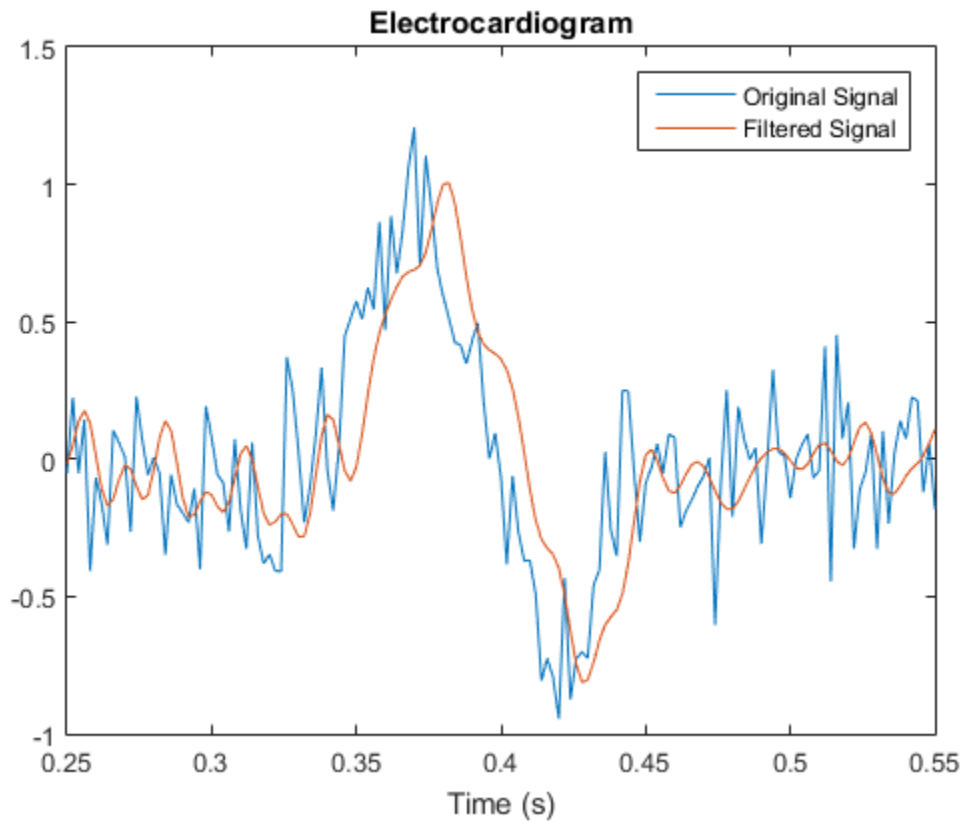
A look at the *group delay* introduced by the filter shows that the delay is frequency-dependent.

```
grpdelay(d,N,Fs)
```
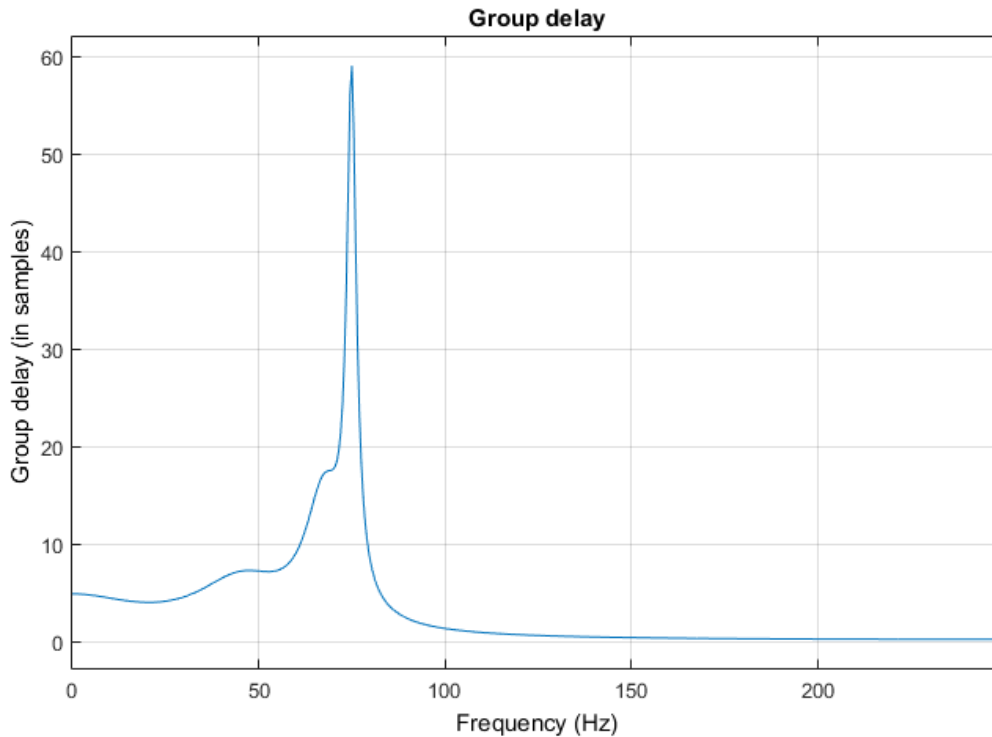
Filter the signal using `filtfilt`. The delay and distortion have been effectively removed. Use `filtfilt` when it is critical to keep the phase information of a signal intact.

```
xfiltfilt = filtfilt(d,xn);

plot(tn,xn,tn,xfilter)
hold on
plot(tn,xfiltfilt,'r','linewidth',2)
hold off

title 'Electrocardiogram'
xlabel 'Time (s)'
legend('Original Signal','Filtered Signal', ...
       'Zero-phase filtered with ''filtfilt''')
axis([0.25 0.55 -1 1.5])
```
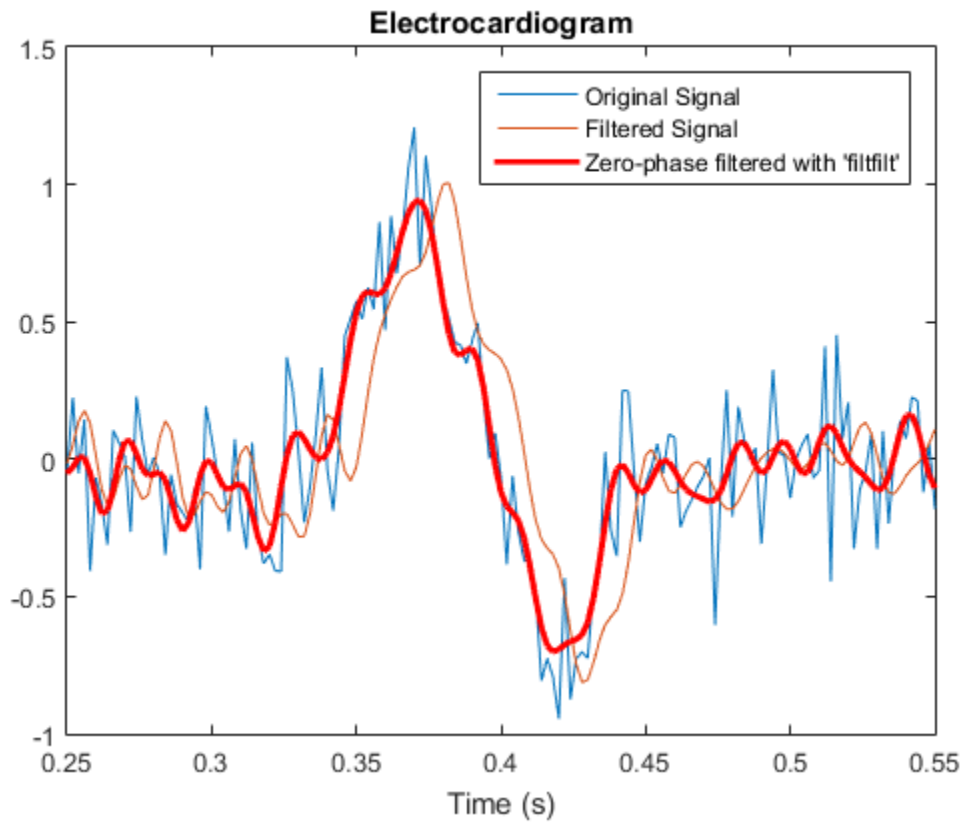
See Also

`designfilt` | `filter` | `filtfilt` | `grpdelay`

Related Examples

- "Compensate for the Delay Introduced by an FIR Filter"
- "Practical Introduction to Digital Filtering"

Take Derivatives of a Signal

You want to differentiate a signal without increasing the noise power. MATLAB®'s function `diff` amplifies the noise, and the resulting inaccuracy worsens for higher derivatives. To fix this problem, use a differentiator filter instead.

Analyze the displacement of a building floor during an earthquake. Find the speed and acceleration as functions of time.

Load the file `earthquake`. The file contains the following variables:

- `drift`: Floor displacement, measured in centimeters
- `t`: Time, measured in seconds
- `Fs`: Sample rate, equal to 1 kHz

```
load(fullfile(matlabroot,'examples','signal','earthquake.mat'))
```

Use `pwelch` to display an estimate of the power spectrum of the signal. Note how most of the signal energy is contained in frequencies below 100 Hz.

```
pwelch(drift,[],[],[],Fs)
```

Welch Power Spectral Density Estimate
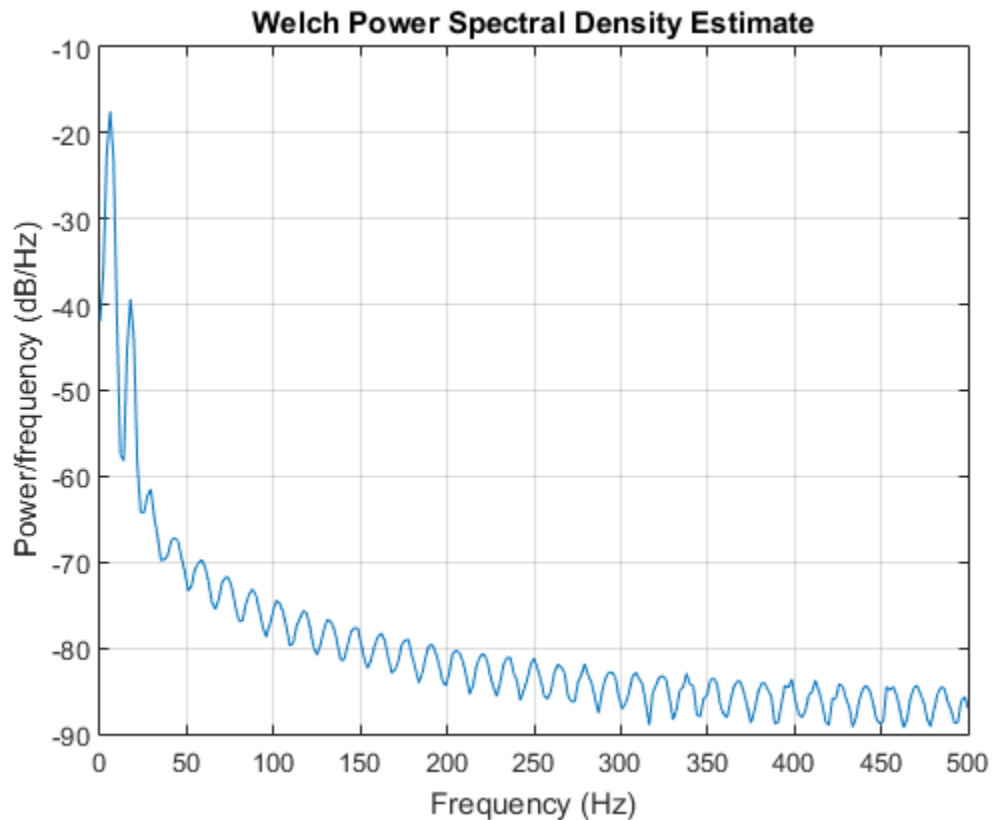
Use `designfilt` to design an FIR differentiator of order 50. To include most of the signal energy, specify a passband frequency of 100 Hz and a stopband frequency of 120 Hz. Inspect the filter with `fvtool`.

```
Nf = 50;
Fpass = 100;
Fstop = 120;

d = designfilt('differentiatorfir','FilterOrder',Nf, ...
    'PassbandFrequency',Fpass,'StopbandFrequency',Fstop, ...
    'SampleRate',Fs);

fvtool(d,'MagnitudeDisplay','zero-phase','Fs',Fs)
```
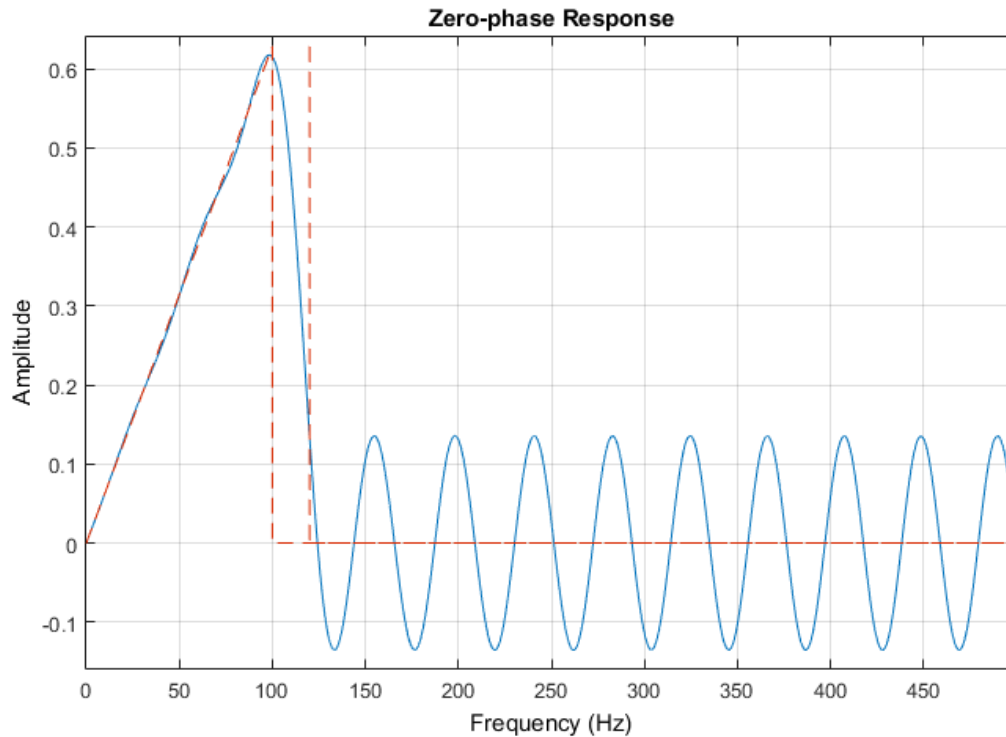
Differentiate the drift to find the speed. Divide the derivative by `dt`, the time interval between consecutive samples, to set the correct units.

```
dt = t(2)-t(1);
```

```
vdrift = filter(d,drift)/dt;
```

The filtered signal is delayed. Use `grpdelay` to determine that the delay is half the filter order. Compensate for it by discarding samples.

```
delay = mean(grpdelay(d))
```

```
tt = t(1:end-delay);
vd = vdrift;
vd(1:delay) = [];
```

```
delay =

    25
```

The output also includes a transient whose length equals the filter order, or twice the group delay. `delay` samples were discarded above. Discard `delay` more to eliminate the transient.

```
tt(1:delay) = [];
vd(1:delay) = [];
```

Plot the drift and the drift speed. Use `findpeaks` to verify that the maxima and minima of the drift correspond to the zero crossings of its derivative.

```
[pkp,lcp] = findpeaks(drift);
zcp = zeros(size(lcp));

[pkm,lcm] = findpeaks(-drift);
zcm = zeros(size(lcm));

subplot(2,1,1)
plot(t,drift,t([lcp lcm]),[pkp -pkm],'or')
xlabel('Time (s)')
ylabel('Displacement (cm)')
grid on

subplot(2,1,2)
plot(tt,vd,t([lcp lcm]),[zcp zcm],'or')
xlabel('Time (s)')
ylabel('Speed (cm/s)')
grid on
```
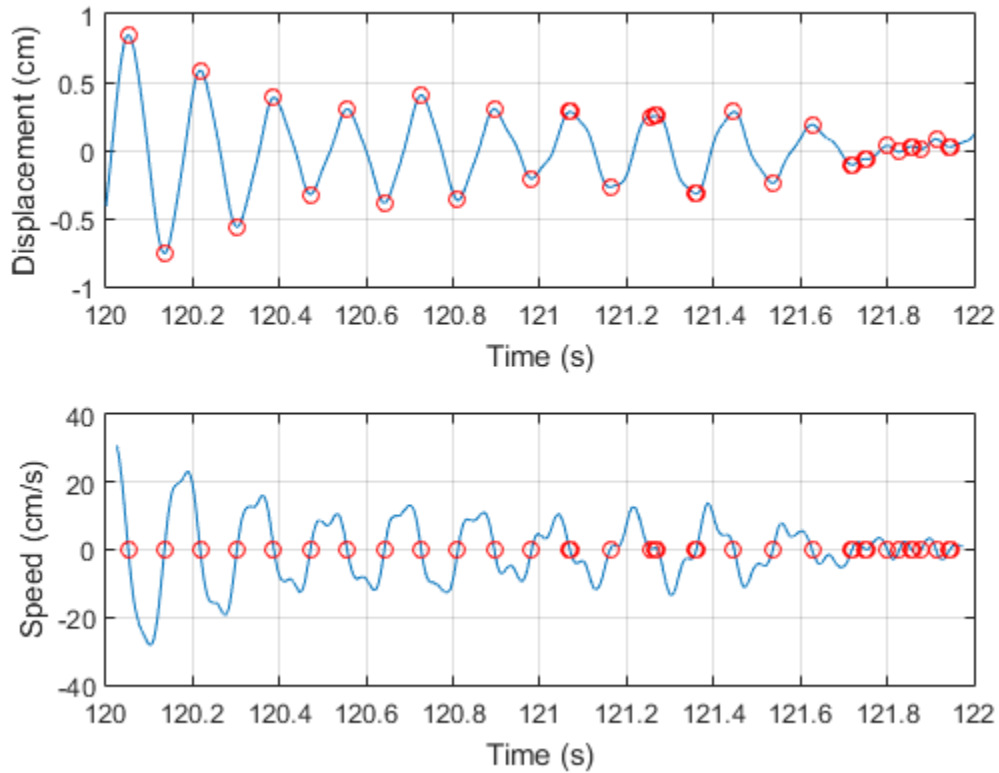
Differentiate the drift speed to find the acceleration. The lag is twice as long. Discard twice as many samples to compensate for the delay, and the same number to eliminate the transient. Plot the speed and acceleration.

```
adrift = filter(d,vdrift)/dt;

at = t(1:end-2*delay);
ad = adrift;
ad(1:2*delay) = [];

at(1:2*delay) = [];
ad(1:2*delay) = [];

subplot(2,1,1)
```
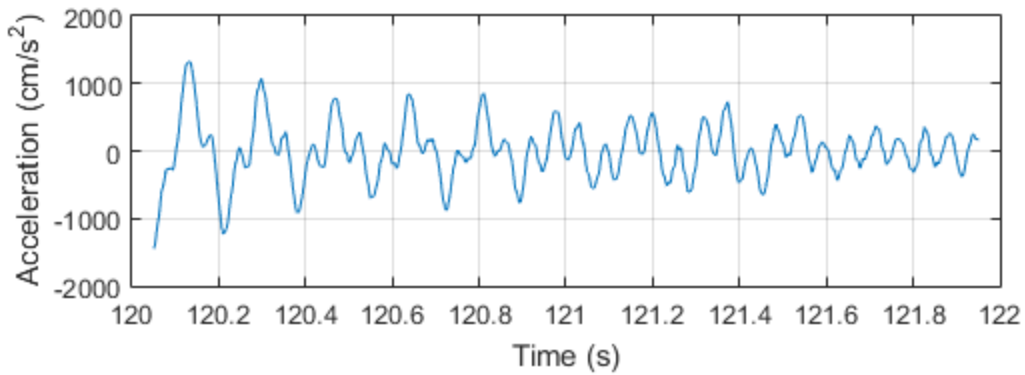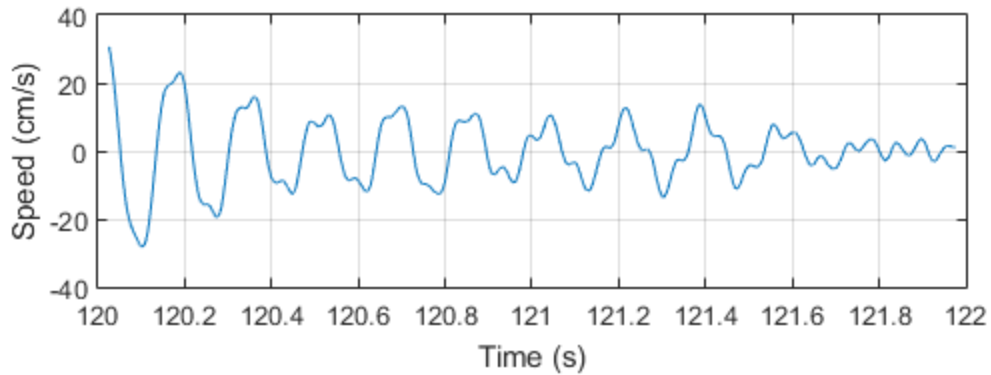
```
plot(tt,vd)
xlabel('Time (s)')
ylabel('Speed (cm/s)')
grid on

subplot(2,1,2)
plot(at,ad)
ax = gca;
ax.YLim = 2000*[-1 1];
xlabel('Time (s)')
ylabel('Acceleration (cm/s^2)')
grid on
```

Compute the acceleration using `diff`. Add zeros to compensate for the change in array size. Compare the result to that obtained with the filter. Notice the amount of high-frequency noise.

```
vdiff = diff([drift;0])/dt;
adiff = diff([vdiff;0])/dt;

subplot(2,1,1)
plot(at,ad)
ax = gca;
ax.YLim = 2000*[-1 1];
xlabel('Time (s)')
ylabel('Acceleration (cm/s^2)')
grid on
legend('Filter')

subplot(2,1,2)
plot(t,adiff)
ax = gca;
ax.YLim = 2000*[-1 1];
xlabel('Time (s)')
ylabel('Acceleration (cm/s^2)')
grid on
legend('diff')
```
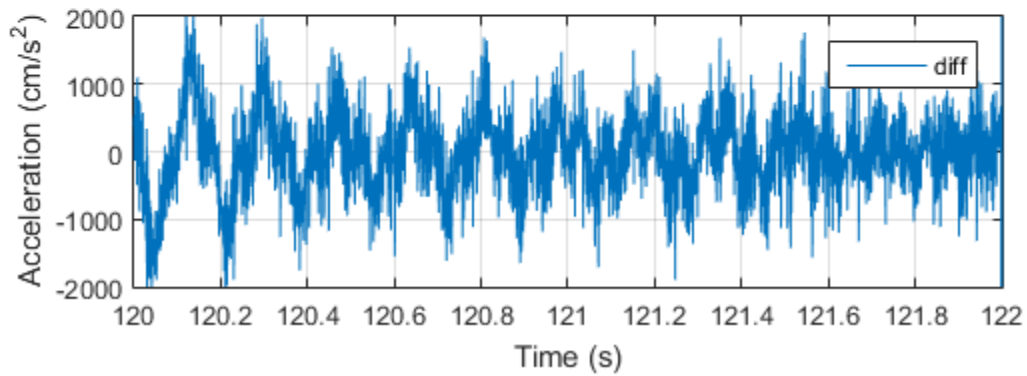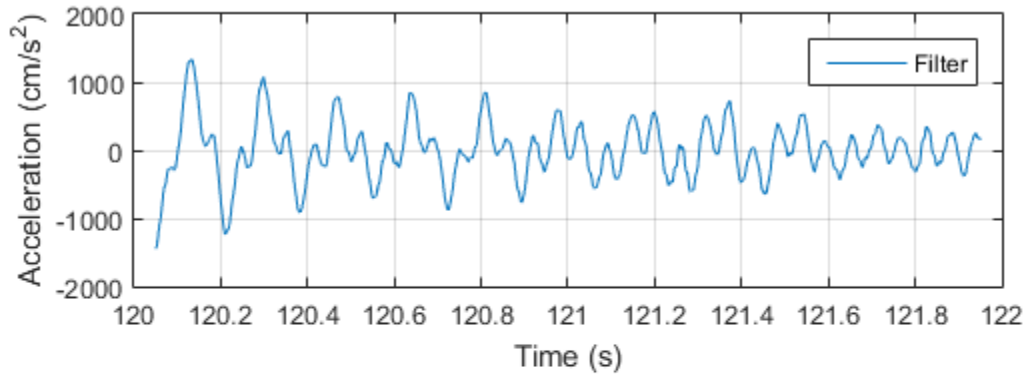
See Also

designfilt | findpeaks | fvtool | grpdelay | periodogram

Related Examples

- "Practical Introduction to Digital Filtering"

Find Periodicity Using Frequency Analysis

It is often difficult to characterize oscillatory behavior in data by looking at time measurements. Spectral analysis can help determine if a signal is periodic and measure the different cycles.

A thermometer in an office building measures the inside temperature every half hour for four months. Load the data and plot it. Measure time in weeks and temperature in degrees Celsius.
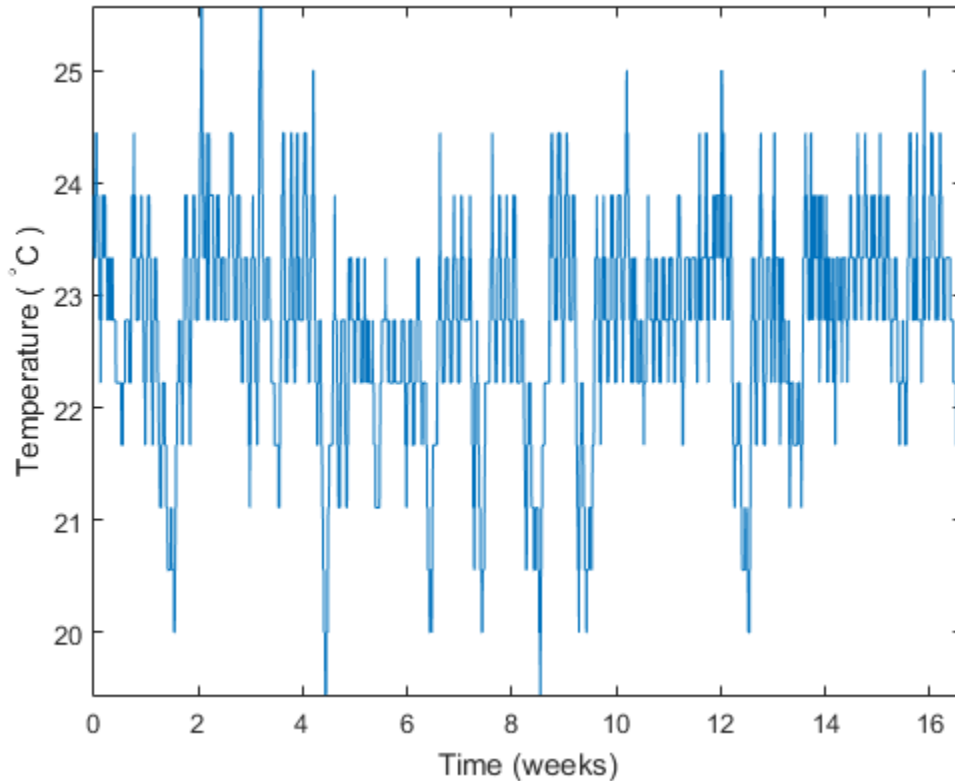
```
load officetemp

tempC = (temp - 32)*5/9;

fs = 2*24*7;
t = (0:length(tempC) - 1)/fs;

plot(t,tempC)
xlabel('Time (weeks)')
ylabel('Temperature ( {}^\circC )')
axis tight
```

The temperature does seem to oscillate, but the lengths of the cycles cannot be determined easily. Look at the signal's frequency content instead.

Subtract the mean to concentrate on temperature fluctuations. Compute and plot the periodogram.

```
tempnorm = tempC - mean(tempC);

[pxx,f] = periodogram(tempnorm,[],[],fs);

plot(f,pxx)
ax = gca;
ax.XLim = [0 10];
xlabel('Frequency (cycles/week)')
```

18-85

```
ylabel('Magnitude')
```
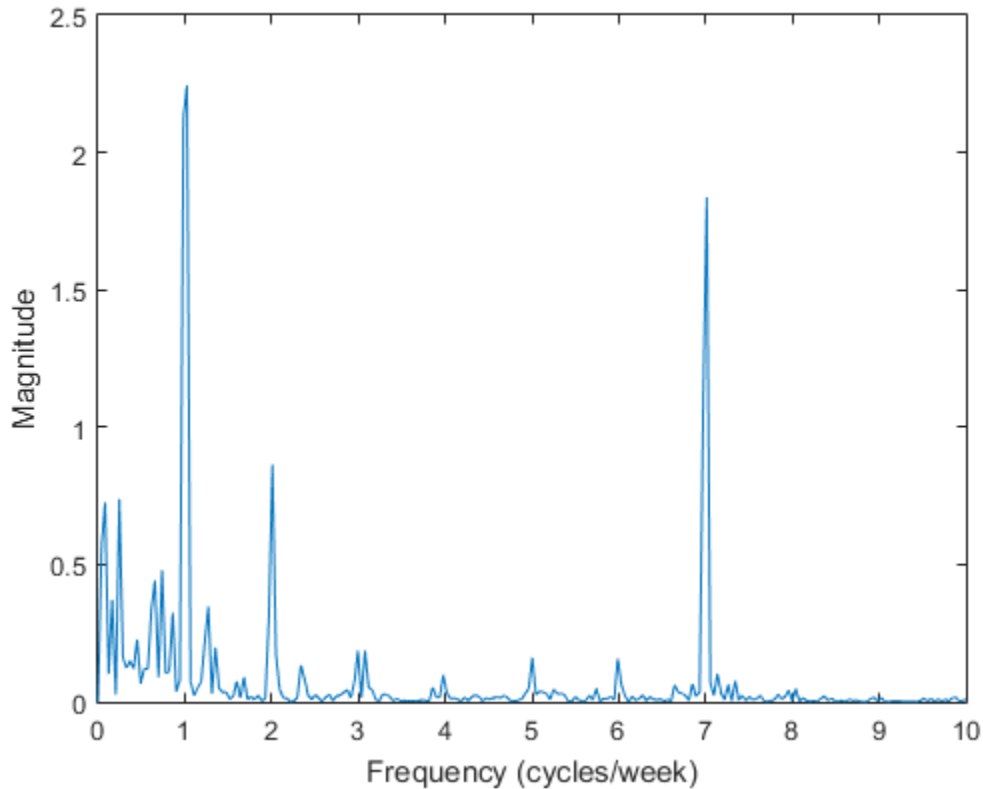

The temperature clearly has a daily cycle and a weekly cycle. The result is not surprising: the temperature is higher when people are at work and lower at nights and on weekends.

See Also
```
findpeaks | periodogram | xcorr
```

Related Examples
- "Find Periodicity Using Autocorrelation"
- "Practical Introduction to Frequency-Domain Analysis"

Detect a Distorted Signal in Noise

The presence of noise often makes it difficult to determine the spectral content of a signal. Frequency analysis can help in such cases.

Consider for example the simulated output of a nonlinear amplifier that introduces third-order distortion.

The input signal is a 180 Hz unit-amplitude sinusoid sampled at 3.6 kHz. Generate 10000 samples.

```
N = 1e4;
n = 0:N-1;
fs = 3600;
f0 = 180;
t = n/fs;
y = sin(2*pi*f0*t);
```

Add unit-variance white noise to the input. Model the amplifier using a third-order polynomial. Pass the input signal through the amplifier using `polyval`. Plot a section of the output. For comparison plot the output of a pure sinusoid.
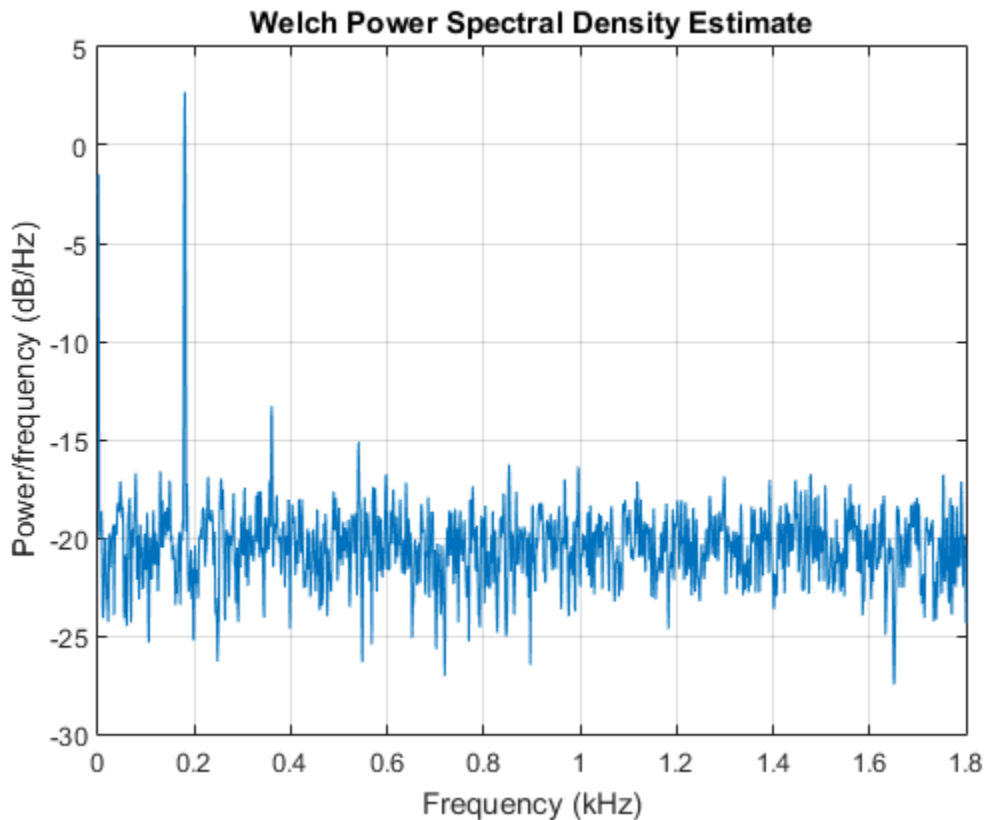
```
rng default
noise = randn(size(y));

dispol = [0.5 0.75 1 0];
out = polyval(dispol,y+noise);

ns = 300:500;

plot(t(ns),[out(ns);polyval(dispol,y(ns))])
xlabel('Time (s)')
ylabel('Signals')
axis tight
legend('With white noise','No white noise')
```
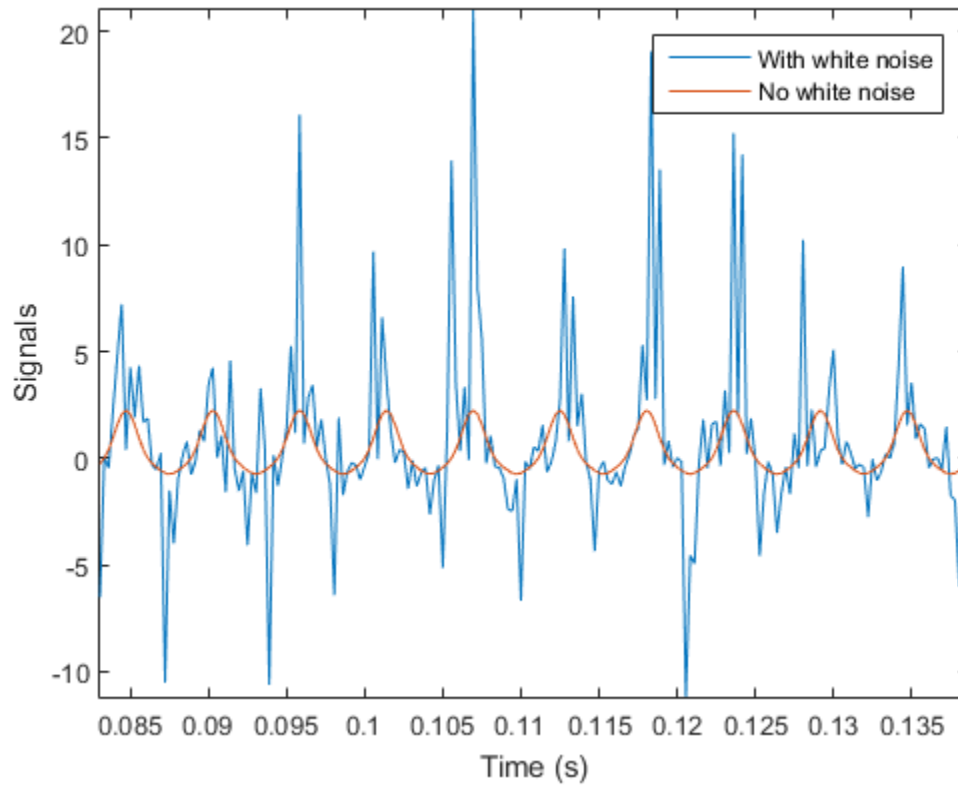
Use `pwelch` to compute and plot the power spectral density of the output.

```
[pxx,f] = pwelch(out,[],[],[],fs);

pwelch(out,[],[],[],fs)
```

Welch Power Spectral Density Estimate

Because the amplifier introduces third-order distortion, the output signal is expected to have:

- A *DC* (zero-frequency) component;
- A *fundamental* component with the same frequency as the input, 180 Hz;
- Two *harmonics* -- frequency components at twice and three times the frequency of the input, 360 and 540 Hz.

Verify that the output is as expected for a cubic nonlinearity.

```
[pks,lox] = findpeaks(pxx,'NPeaks',4,'SortStr','descend');

hold on
```
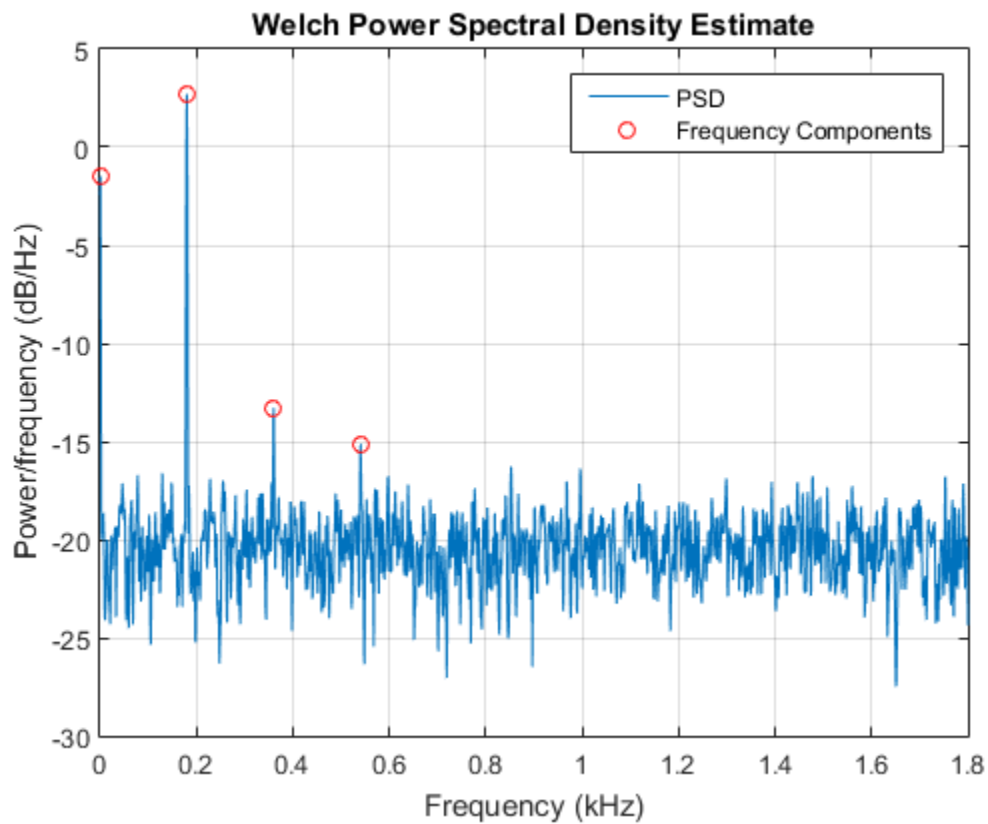
```
plot(f(lox)/1000,10*log10(pks),'or')
hold off

legend('PSD','Frequency Components')

components = sort([f(lox) f0*(0:3)'])'


components =

    0.8789  180.1758  360.3516  540.5273
         0  180.0000  360.0000  540.0000
```

`pwelch` works by dividing the signal into overlapping segments, computing the periodogram of each segment, and averaging. By default, the function uses eight segments with 50% overlap. For 10000 samples, this corresponds to 2222 samples per segment.

Dividing the signal into shorter segments results in more averaging. The periodogram is smoother, but has lower resolution. The higher harmonic cannot be distinguished.

```
pwelch(out,222,[],[],fs)
```
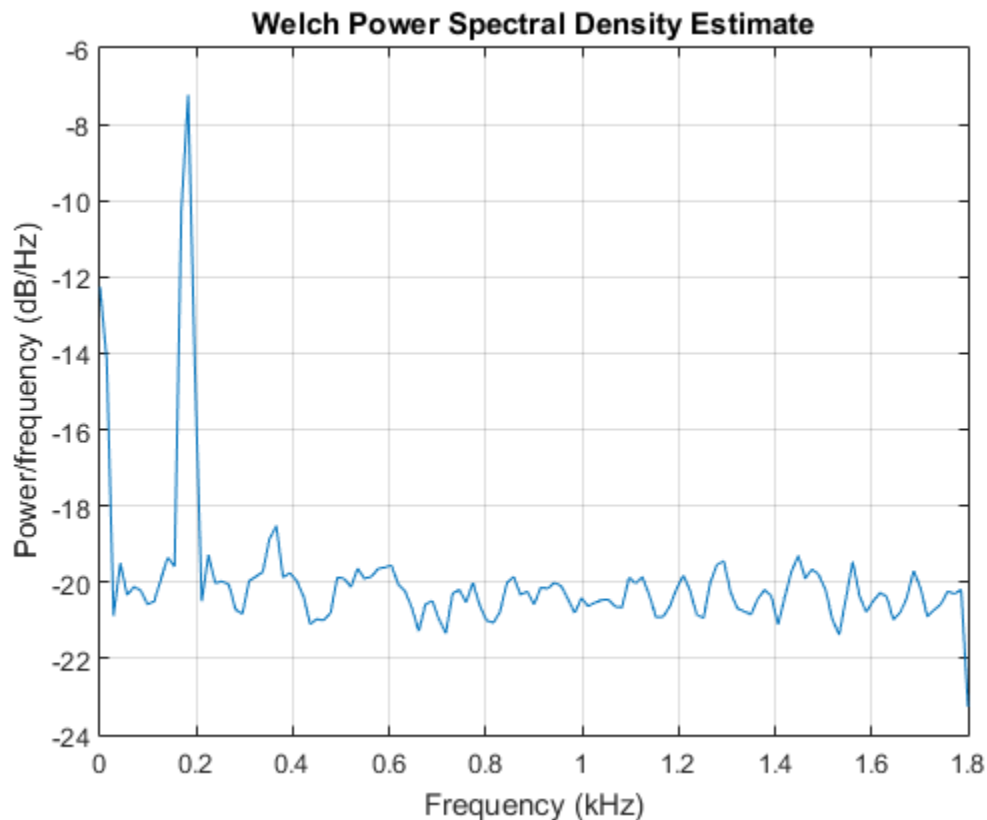

Dividing the signal into longer segments increases the resolution, but also the randomness. The signal and the harmonics are precisely at the expected locations. However, there is at least one spurious high-frequency peak with more power than the higher harmonic.

```
pwelch(out,4444,[],[],fs)
```
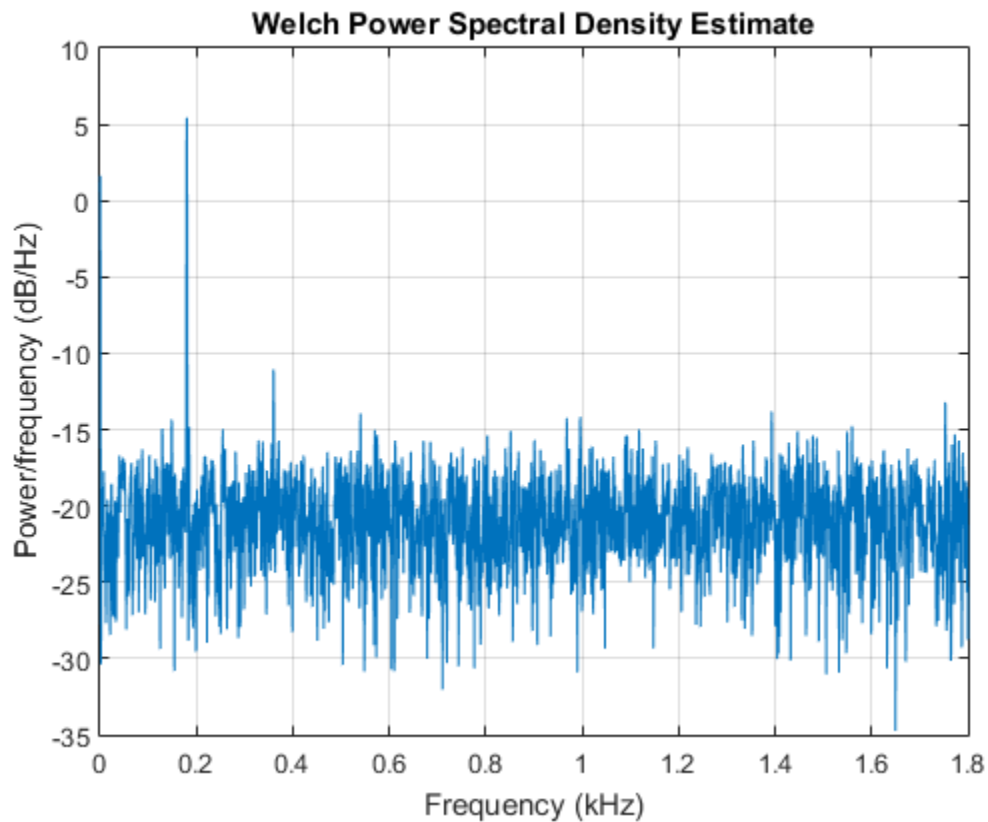

See Also
findpeaks | pwelch

Related Examples

- "Practical Introduction to Frequency-Domain Analysis"

Measure the Power of a Signal

The power of a signal is the sum of the absolute squares of its time-domain samples divided by the signal length, or, equivalently, the square of its RMS level. The function bandpower allows you to calculate signal power in one step.

Initially, consider a noisy sinusoid. Verify that the power is the sum of the power of the signal and of the noise.

```
N = 1200;
Fs = 1000;
t = (0:N-1)/Fs;
w = 124;
A = 2.4;
sigma = 0.13;
s = A*sin(2*pi*w*t)+sigma*randn(size(t));

pTheory = A^2/2+sigma^2
pRMS = rms(s)^2
powbp = bandpower(s,Fs,[0 Fs/2])


pTheory =

    2.8969


pRMS =

    2.8849


powbp =

    2.8775
```

The power of the sinusoid is in the expected frequency band.

```
powband = bandpower(s,Fs,[122 126])


powband =
```

```
          2.8616
```

A nonlinear power amplifier is given a 60 Hz sinusoid as input and outputs a noisy signal with third-order distortion. The sample rate is 3.6 kHz. Subtract the zero-frequency (DC) component to concentrate on the spectral content.

```
load(fullfile(matlabroot,'examples','signal','AmpOutput.mat'))
Fs = 3600;
y = y-mean(y);
```

Because the amplifier introduces third-order distortion, the output signal is expected to have

- A *fundamental* component with the same frequency as the input, 60 Hz;
- Two *harmonics* -- frequency components at twice and three times the frequency of the input, 120 and 180 Hz.

Use bandpower to determine the power stored in the fundamental and the harmonics. Express each value as a percentage of the total power and in decibels. Display the values as a table.

```
pwrTot = bandpower(y,Fs,[0 Fs/2]);

Harmonic = {'Fundamental';'First';'Second'};

Freqs = [60 120 180]';

Power = zeros([3 1]);
for k = 1:3
    Power(k) = bandpower(y,Fs,Freqs(k)+[-10 10]);
end

Percent = Power/pwrTot*100;

inDB = pow2db(Power);

T = table(Freqs,Power,Percent,inDB,'RowNames',Harmonic)


T =

                  Freqs     Power       Percent      inDB
                  _____    _____    _____    _____
```

```
Fundamental      60        1.0079    12.563      0.034136
First           120        0.14483    1.8053     -8.3914
Second          180        0.090023   1.1221    -10.456
```

See Also

bandpower | pow2db | pwelch | snr

Related Examples

- "Practical Introduction to Frequency-Domain Analysis"

Compare the Frequency Content of Two Signals

Spectral coherence helps identify similarity between signals in the frequency domain. Large values indicate frequency components common to the signals.

Load two sound signals into the workspace. They are sampled at 1 kHz. Compute their power spectra using `periodogram` and plot them next to each other.
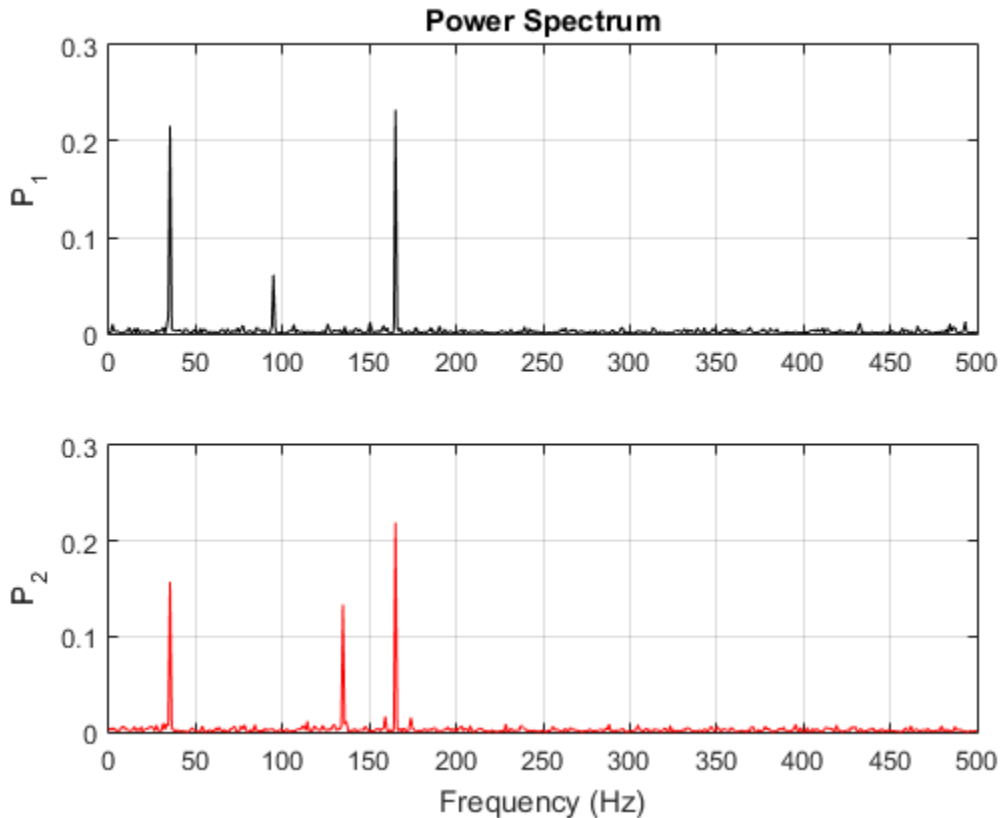
```
load relatedsig

Fs = FsSig;

[P1,f1] = periodogram(sig1,[],[],Fs,'power');
[P2,f2] = periodogram(sig2,[],[],Fs,'power');

subplot(2,1,1)
plot(f1,P1,'k')
grid
ylabel('P_1')
title('Power Spectrum')

subplot(2,1,2)
plot(f2,P2,'r')
grid
ylabel('P_2')
xlabel('Frequency (Hz)')
```

Each signal has three frequency components with significant energy. Two of those components appear to be shared. Find the corresponding frequencies using `findpeaks`.

```
[pk1,lc1] = findpeaks(P1,'SortStr','descend','NPeaks',3);
P1peakFreqs = f1(lc1)

[pk2,lc2] = findpeaks(P2,'SortStr','descend','NPeaks',3);
P2peakFreqs = f2(lc2)


P1peakFreqs =

   165.0391
    35.1563
```

```
    94.7266


P2peakFreqs =

   165.0391
    35.1563
   134.7656
```

The common components are located around 165 and 35 Hz. You can use `mscohere` to find the matching frequencies directly. Plot the coherence estimate. Find the peaks above a threshold of 0.75.

```
[Cxy,f] = mscohere(sig1,sig2,[],[],[],Fs);

thresh = 0.75;
[pks,locs] = findpeaks(Cxy,'MinPeakHeight',thresh);
MatchingFreqs = f(locs)

figure
plot(f,Cxy)
ax = gca;
grid
xlabel('Frequency (Hz)')
title('Coherence Estimate')
ax.XTick = MatchingFreqs;
ax.YTick = thresh;
axis([0 200 0 1])


MatchingFreqs =

    35.1563
   164.0625
```
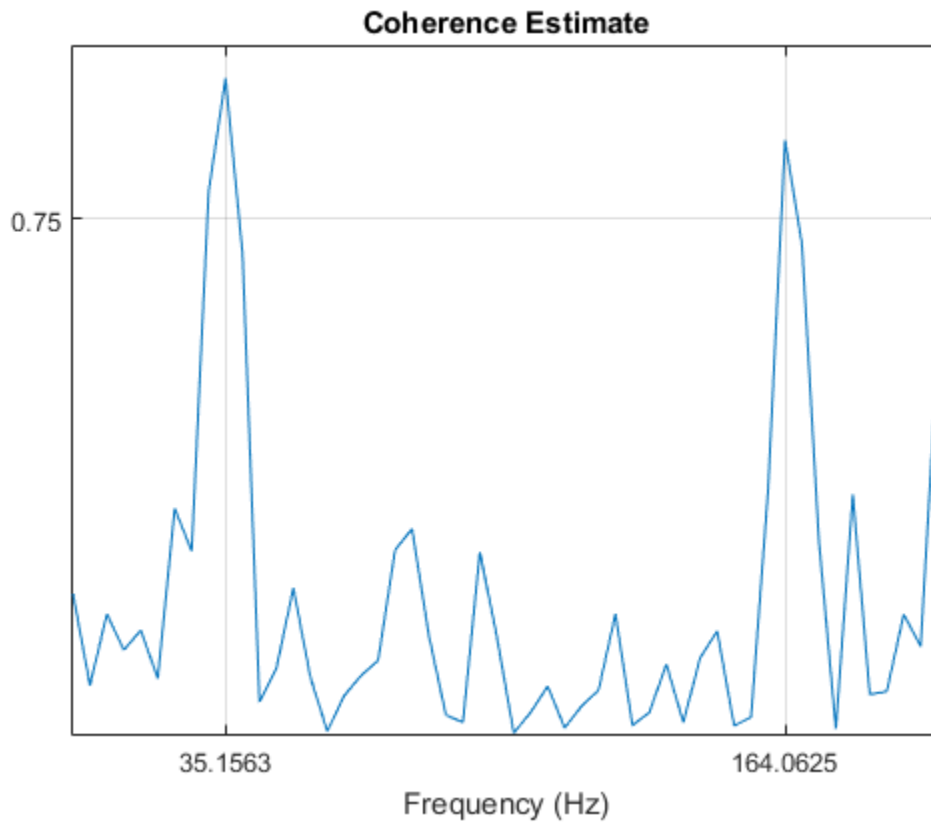
You get the same values as before. You can find the frequency content common to two signals without studying the two signals separately.

See Also
`findpeaks` | `mscohere` | `periodogram`

Related Examples
- "Practical Introduction to Frequency-Domain Analysis"

Detect Periodicity in a Signal with Missing Samples

Consider the weight of a person as recorded (in pounds) during the leap year 2012. The person did not record their weight every day. You would like to study the periodicity of the signal, even though some data points are missing.

Load the data and convert the measurements to kilograms. Missed readings are set to NaN. Determine how many points are missing.
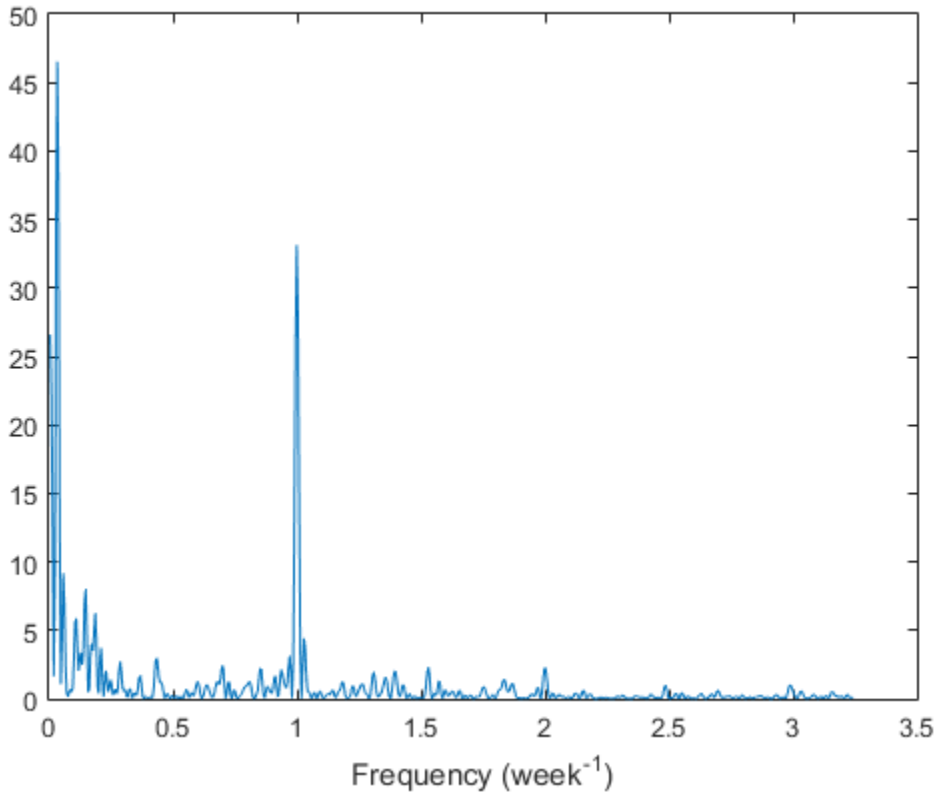
```
load(fullfile(matlabroot,'examples','signal','weight2012.dat'))

wgt = weight2012(:,2)/2.20462;

fprintf('Missing %d samples of %d\n',sum(isnan(wgt)),length(wgt))
```

```
Missing 27 samples of 366
```

Determine if the signal is periodic by analyzing it in the frequency domain. The Lomb-Scargle algorithm is designed to handle data with missing samples or data that have been sampled irregularly.

Find the cycle durations, measuring time in weeks.

```
[p,f] = plomb(wgt,7,'normalized');

plot(f,p)
xlabel('Frequency (week^{-1})')
```
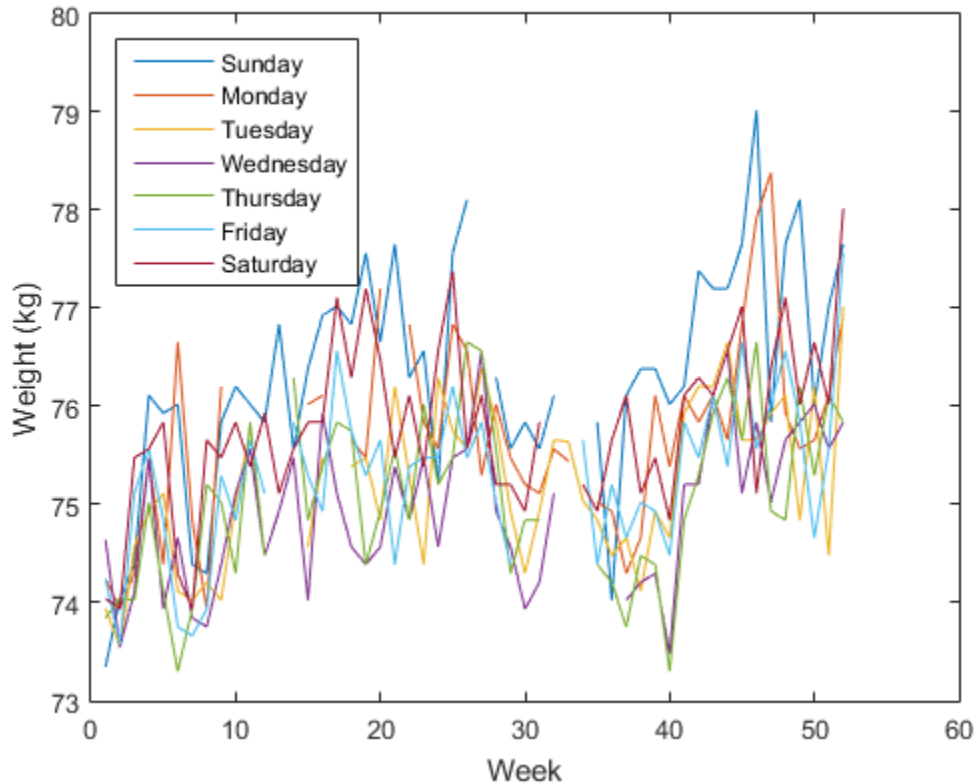
Notice how the person's weight oscillates weekly. Is there a noticeable pattern from week to week? Eliminate the last two days of the year to get 52 weeks. Reorder the measurements according to the day of the week.

```
wgd = reshape(wgt(1:7*52),[7 52])';

plot(wgd)
xlabel('Week')
ylabel('Weight (kg)')

q = legend(datestr(datenum(2012,1,1:7),'dddd'));
q.Location = 'NorthWest';
```
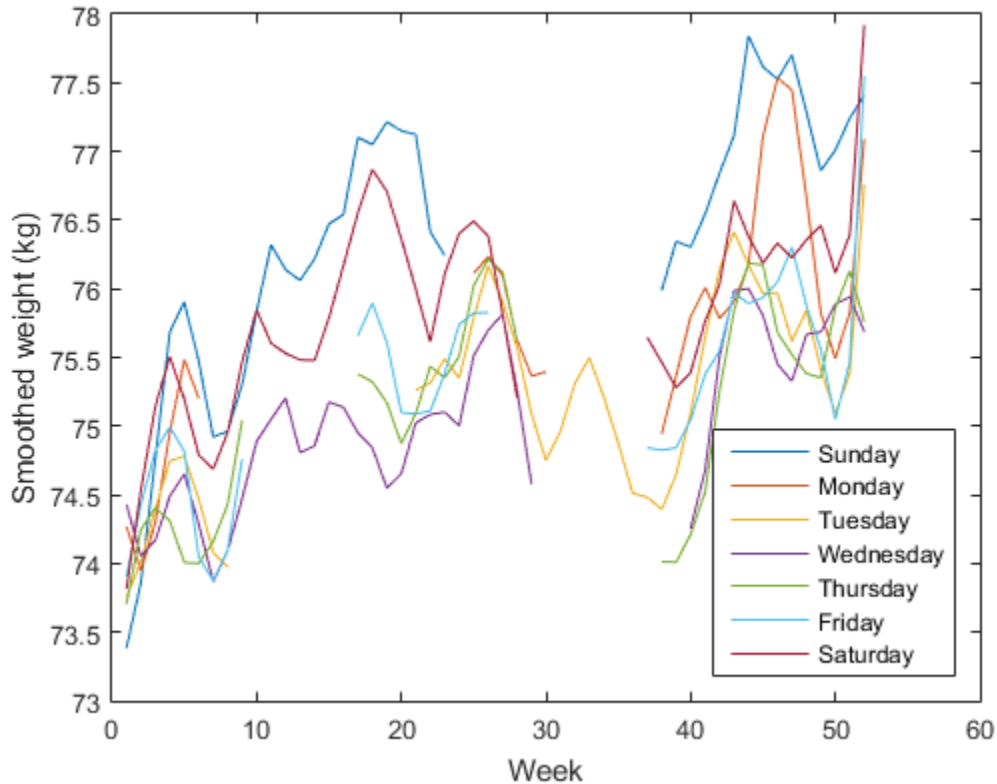
Smooth out the fluctuations using a filter that fits low-order polynomials to subsets of the data. Specifically, set it to fit cubic polynomials to sets of seven days.

```
wgs = sgolayfilt(wgd,3,7);

plot(wgs)
xlabel('Week')
ylabel('Smoothed weight (kg)')

q = legend(datestr(datenum(2012,1,1:7),'dddd'));
q.Location = 'SouthEast';
```

This person tends to eat more, and thus weigh more, during the weekend. Verify by computing the daily means. Exclude the missing values from the calculation.

```
for jk = 1:7
    wgm = find(~isnan(wgd(:,jk)));
    fprintf('%3s mean: %5.1f kg\n', ...
        datestr(datenum(2012,1,jk),'ddd')',mean(wgd(wgm,jk)))
end
```

```
Sun mean:  76.3 kg
Mon mean:  75.7 kg
Tue mean:  75.2 kg
Wed mean:  74.9 kg
Thu mean:  75.1 kg
```

```
Fri mean:   75.3 kg
Sat mean:   75.8 kg
```

See Also
datestr | plomb | sgolayfilt

Related Examples
·   "Signal Smoothing"

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.