Contents

1 Reference

abs 6-11
angle 6-12
bartlett 6-13
besselap 6-14
besself 6-15
bilinear 6-19
blackman 6-24
boxcar 6-25
buttap 6-26
butter 6-27
buttord 6-32
cceps 6-36
cheb1ap 6-37
cheb1ord 6-38
cheb2ap 6-42
cheb2ord 6-43
chebwin 6-47
cheby1 6-48
cheby2 6-53
chirp 6-58
cohere 6-61
conv 6-65
conv2 6-66
convmtx 6-68
corrcoef 6-70
cov 6-71
cplxpair 6-72
cremez 6-73
csd 6-80
czt 6-85
dct 6-87
decimate 6-89
deconv 6-92
demod 6-93
detrend 6-95

dftmtx 6-96
diric 6-97
dpss 6-98
dpssclear 6-100
dpssdir 6-101
dpssload 6-102
dpsssave 6-103
ellip 6-104
ellipap 6-110
ellipord 6-111
fft 6-115
fft2 6-119
fftfilt 6-120
fftshift 6-122
filter 6-123
filter2 6-126
filtfilt 6-127
filtic 6-128
fir1 6-130
fir2 6-133
fircls 6-136
fircls1 6-139
firls 6-142
firrcos 6-147
freqs 6-148
freqspace 6-151
freqz 6-152
gauspuls 6-155
grpdelay 6-157
hamming 6-160
hanning 6-161
hilbert 6-162
icceps 6-164
idct 6-165
ifft 6-166
ifft2 6-167
impinvar 6-168
impz 6-170
interp 6-173
intfilt 6-175

invfreqs 6-177
invfreqz 6-180
kaiser 6-183
kaiserord 6-184
latc2tf 6-189
latcfilt 6-190
levinson 6-191
lp2bp 6-192
lp2bs 6-195
lp2hp 6-197
lp2lp 6-199
lpc 6-201
maxflat 6-204
medfilt1 6-206
modulate 6-207
pmem 6-210
pmtm 6-214
pmusic 6-217
poly2rc 6-223
polystab 6-225
prony 6-226
psd 6-228
pulstran 6-233
rc2poly 6-237
rceps 6-238
rectpuls 6-239
remez 6-240
remezord 6-245
resample 6-249
residuez 6-253
sawtooth 6-256
sinc 6-257
sos2ss 6-260
sos2tf 6-262
sos2zp 6-264
specgram 6-266
sptool 6-271
square 6-275
ss2sos 6-276
ss2tf 6-279

ss2zp 6-280
stmcb 6-282
strips 6-285
tf2latc 6-287
tf2ss 6-288
tf2zp 6-290
tfe 6-292
triang 6-296
tripuls 6-297
unwrap 6-298
upfirdn 6-299
vco 6-303
xcorr 6-305
xcorr2 6-310
xcov 6-311
yulewalk 6-314
zp2sos 6-317
zp2ss 6-320
zp2tf 6-321
zplane 6-322

1

Reference

This section contains detailed descriptions of all Signal Processing Toolbox functions. It begins with a list of functions grouped by subject area and continues with the reference entries in alphabetical order. For more information, see the online MATLAB Function Reference.

Waveform Generation and Plotting	
chirp	Swept-frequency cosine generator.
diric	Dirichlet or periodic sinc function.
gauspuls	Gaussian-modulated sinusoidal pulse generator.
pulstran	Pulse train generator.
rectpuls	Sampled aperiodic rectangle generator.
sawtooth	Sawtooth or triangle wave generator.
sinc	Sinc or sin(πt)/πt function.
square	Square wave generator.
strips	Strip plot.
tripuls	Sampled aperiodic triangle generator.

Filter Analysis and Implementation	
abs	Absolute value (magnitude).
angle	Phase angle.
conv	Convolution and polynomial multiplication.
conv2	Two-dimensional convolution.
fftfilt	FFT-based FIR filtering using the overlap-add method.
filter	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.

Filter Analysis and Implementation	
filter2	Two-dimensional digital filtering.
filtfilt	Zero-phase digital filtering.
filtic	Make initial conditions for filter function.
freqs	Frequency response of analog filters.
freqspace	Frequency spacing for frequency response.
freqz	Frequency response of digital filters.
grpdelay	Average filter delay (group delay).
impz	Impulse response of digital filters.
latcfilt	Lattice and lattice-ladder filter implementation.
unwrap	Unwrap phase angles.
zplane	Zero-pole plot.

Linear System Transformations	
convmtx	Convolution matrix.
latc2tf	Lattice filter to transfer function conversion.
poly2rc	Reflection coefficients from polynomial coefficients.
rc2poly	Polynomial coefficients from reflection coefficients.
residuez	z-transform partial-fraction expansion.
sos2ss	Second-order section to state-space conversion.
sos2tf	Second-order section to transfer function conversion.
sos2zp	Second-order section to zero-pole-gain conversion.
ss2sos	State-space to second-order section conversion.

Linear System Transformations	
ss2tf	State-space to transfer function conversion.
ss2zp	State-space to zero-pole-gain conversion.
tf2latc	Transfer function to lattice filter conversion.
tf2ss	Transfer function to state-space conversion.
tf2zp	Transfer function to zero-pole-gain conversion.
zp2sos	Zero-pole-gain to second-order section conversion.
zp2ss	Zero-pole-gain to state-space conversion.
zp2tf	Zero-pole-gain to transfer function conversion.

IIR Filter Design—Classical and Direct	
besself	Bessel analog filter design.
butter	Butterworth analog and digital filter design.
cheby1	Chebyshev type I filter design (passband ripple).
cheby2	Chebyshev type II filter design (stopband ripple).
ellip	Elliptic (Cauer) filter design.
maxflat	Generalized digital Butterworth filter design.
yulewalk	Recursive digital filter design.

IIR Filter Order Selection	
buttord	Butterworth filter order selection.
cheb1ord	Chebyshev type I filter order selection.
cheb2ord	Chebyshev type II filter order selection.
ellipord	Elliptic filter order selection.

FIR Filter Design	
cremez	Complex and nonlinear-phase equiripple FIR filter design
fir1	Window-based finite impulse response filter design—standard response.
fir2	Window-based finite impulse response filter design—arbitrary response.
fircls	Constrained least square FIR filter design for multiband filters.
fircls1	Constrained least square filter design for lowpass and highpass linear phase FIR filters.
firls	Least square linear-phase FIR filter design.
firrcos	Raised cosine FIR filter design.
intfilt	Interpolation FIR filter design.
kaiserord	Estimate parameters for fir1 with Kaiser window.
remez	Parks-McClellan optimal FIR filter design.
remezord	Parks-McClellan optimal FIR filter order estimation.

Transforms	
czt	Chirp *z*-transform.
dct	Discrete cosine transform (DCT).
dftmtx	Discrete Fourier transform matrix.
fft	One-dimensional fast Fourier transform.
fft2	Two-dimensional fast Fourier transform.
fftshift	Rearrange the outputs of fft and fft2.
hilbert	Hilbert transform.
idct	Inverse discrete cosine transform.
ifft	One-dimensional inverse fast Fourier transform.
ifft2	Two-dimensional inverse fast Fourier transform.

Statistical Signal Processing	
cohere	Estimate magnitude squared coherence function between two signals.
corrcoef	Correlation coefficient matrix.
cov	Covariance matrix.
csd	Estimate the cross spectral density (CSD) of two signals.
pmem	Power spectrum estimate using maximum entropy method (MEM).
pmtm	Power spectrum estimate using the multitaper method (MTM).

Statistical Signal Processing	
pmusic	Power spectrum estimate using MUSIC eigenvector method.
psd	Estimate the power spectral density (PSD) of a signal.
tfe	Transfer function estimate from input and output.
xcorr	Cross-correlation function estimate.
xcorr2	Two-dimensional cross-correlation.
xcov	Cross-covariance function estimate (equal to mean-removed cross-correlation).

Windows	
bartlett	Bartlett window.
blackman	Blackman window.
boxcar	Rectangular window.
chebwin	Chebyshev window.
hamming	Hamming window.
hanning	Hanning window.
kaiser	Kaiser window.
triang	Triangular window.

Parametric Modeling	
invfreqs	Continuous-time (analog) filter identification from frequency data.
invfreqz	Discrete-time filter identification from frequency data.
levinson	Levinson-Durbin recursion.
lpc	Linear prediction coefficients.
prony	Prony's method for time domain IIR filter design.
stmcb	Linear model using Steiglitz-McBride iteration.

Specialized Operations	
cceps	Complex cepstral analysis.
cplxpair	Group complex numbers into complex conjugate pairs.
decimate	Decrease the sampling rate for a sequence (decimation).
deconv	Deconvolution and polynomial division.
demod	Demodulation for communications simulation.
detrend	Remove linear trends.
dpss	Discrete prolate spheroidal sequences (Slepian sequences).
dpssclear	Remove discrete prolate spheroidal sequences from database.
dpssdir	Discrete prolate spheroidal sequences database directory.
dpssload	Load discrete prolate spheroidal sequences from database.
dpsssave	Save discrete prolate spheroidal sequences in database.

Specialized Operations	
icceps	Inverse complex cepstrum.
interp	Increase sampling rate by an integer factor (interpolation).
medfilt1	One-dimensional median filtering.
modulate	Modulation for communications simulation.
polystab	Stabilize polynomial.
rceps	Real cepstrum and minimum phase reconstruction.
resample	Change sampling rate by any factor.
specgram	Time-dependent frequency analysis (spectrogram).
upfirdn	Apply FIR filter and perform sample rate conversion.
vco	Voltage controlled oscillator.

Analog Prototype Design	
besselap	Bessel analog lowpass filter prototype.
buttap	Butterworth analog lowpass filter prototype.
cheb1ap	Chebyshev type I analog lowpass filter prototype.
cheb2ap	Chebyshev type II analog lowpass filter prototype.
ellipap	Elliptic analog lowpass filter prototype.

Frequency Translation	
lp2bp	Lowpass to bandpass analog filter transformation.
lp2bs	Lowpass to bandstop analog filter transformation.
lp2hp	Lowpass to highpass analog filter transformation.
lp2lp	Lowpass to lowpass analog filter transformation.

Filter Discretization	
bilinear	Map variables using bilinear transformation.
impinvar	Impulse invariance method of analog-to-digital filter conversion.

Interactive Tools	
sptool	Interactive signal, filter, and spectrum analysis tool.

Purpose

Absolute value (magnitude).

Syntax

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

Description

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

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

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

This function is part of the standard MATLAB environment.

Example

Calculate the magnitude of the FFT of a sequence:

```
t = (0:99)/100;                          % time vector
x = sin(2*pi*15*t) + sin(2*pi*40*t);     % signal
y = fft(x);                              % compute DFT of x
m = abs(y);                              % magnitude
```

Plot the magnitude:

```
f = (0:length(y)-1)'/length(y)*100;      % frequency vector
plot(f,m)
```

See Also

`angle`	Phase angle.

angle

Purpose

Phase angle.

Syntax

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

Description

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

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

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

To convert back to the original `h` from its magnitude and phase:

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

This function is part of the standard MATLAB environment.

Example

Calculate the phase of the FFT of a sequence:

```
t = (0:99)/100;                          % time vector
x = sin(2*pi*15*t) + sin(2*pi*40*t);     % signal
y = fft(x);                              % compute DFT of x
p = unwrap(angle(y));                    % phase
```

Plot the phase:

```
f = (0:length(y)-1)'/length(y)*100;      % frequency vector
plot(f,p)
```

Algorithm

`angle` can be expressed as:

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

See Also

`abs`	Absolute value (magnitude).

Purpose Bartlett window.

Syntax

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

Description `w = bartlett(n)` returns an `n`-point Bartlett window in the column vector `w`. The coefficients of a Bartlett window are

For `n` odd

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

For `n` even

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

The Bartlett window is very similar to a triangular window as returned by the `triang` function. The Bartlett window always ends with zeros at samples 1 and `n`, however, while the triangular window is nonzero at those points. For `n` odd, the center `n–2` points of `bartlett(n)` are equivalent to `triang(n–2)`.

See Also

`blackman`	Blackman window.
`boxcar`	Rectangular window.
`chebwin`	Chebyshev window.
`hamming`	Hamming window.
`hanning`	Hanning window.
`kaiser`	Kaiser window.
`triang`	Triangular window.

besselap

Purpose

Bessel analog lowpass filter prototype.

Syntax

```
[z,p,k] = besselap(n)
```

Description

`[z,p,k] = besselap(n)` returns the zeros, poles, and gain of an order `n` Bessel analog lowpass filter prototype. It returns the poles in the length `n` column vector `p` and the gain in scalar `k`. `z` is an empty matrix, because there are no zeros. `n` must be less than or equal to 25. The transfer function is

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

`besselap` normalizes the poles and gain so that at low frequency and high frequency the Bessel prototype is asymptotically equivalent to the Butterworth prototype of the same order [1]. The magnitude of the filter is less than `sqrt(1/2)` at the unity cutoff frequency $\Omega_c = 1$.

Analog Bessel filters are characterized by a group delay that is maximally flat at zero frequency and almost constant throughout the passband. The group delay at zero frequency is

$$\left(\frac{(2n)!}{2^n n!}\right)^{1/n}$$

Algorithm

`besselap` finds the filter roots from a look-up table constructed using the Symbolic Math Toolbox.

See Also

`besself`	Bessel analog filter design.
`buttap`	Butterworth analog lowpass filter prototype.
`cheb1ap`	Chebyshev type I analog lowpass filter prototype.
`cheb2ap`	Chebyshev type II analog lowpass filter prototype.
`ellipap`	Elliptic analog lowpass filter prototype.

Also see the *Symbolic Math Toolbox User's Manual.*

References

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

Purpose Bessel analog filter design.

Syntax

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

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

`[b,a] = besself(n,Wn)` designs an order `n` lowpass analog filter with cutoff frequency `Wn`. It returns the filter coefficients in the length `n+1` row vectors `b` and `a`, with coefficients in descending powers of s:

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

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

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

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

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

  The stopband is `w1` < ω < `w2`.

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

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

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

besself returns the zeros and poles in length n or 2*n column vectors z and p and the gain in the scalar k.

To obtain state-space form, use four output arguments:

[A,B,C,D] = besself(n,Wn) or

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

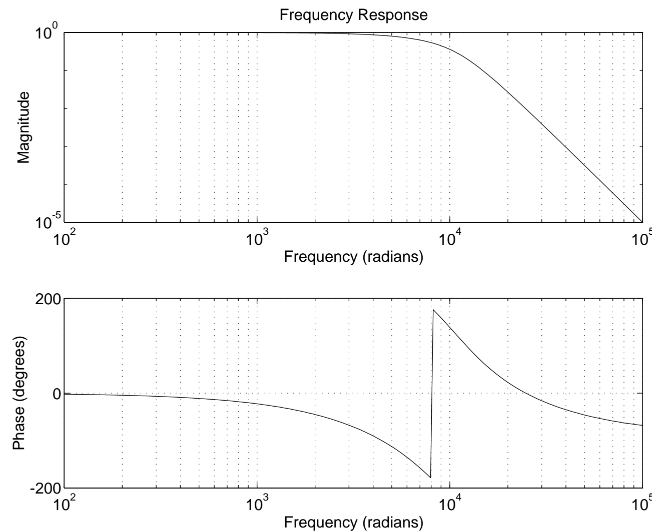
$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

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

Example

Design a fifth-order analog lowpass Bessel filter that suppresses frequencies greater than 10,000 rad/sec and plot the frequency response of the filter using `freqs`:

```
[b,a] = besself(5,10000);
freqs(b,a)                    % plot frequency response
```



Limitations

Lowpass Bessel filters have a monotonically decreasing magnitude response, as do lowpass Butterworth filters. Compared to the Butterworth, Chebyshev, and elliptic filters, the Bessel filter has the slowest rolloff and requires the highest order to meet an attenuation specification.

For high order filters, the state-space form is the most numerically accurate, followed by the zero-pole-gain form. The transfer function coefficient form is the least accurate; numerical problems can arise for filter orders as low as 15.

Algorithm

besself performs a four-step algorithm:

1. It finds lowpass analog prototype poles, zeros, and gain using the besselap function.
2. It converts the poles, zeros, and gain into state-space form.
3. It transforms the lowpass filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies, using a state-space transformation.
4. It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also

besselap	Bessel analog lowpass filter prototype.
butter	Butterworth analog and digital filter design.
cheby1	Chebyshev type I filter design (passband ripple).
cheby2	Chebyshev type II filter design (stopband ripple).
ellip	Elliptic (Cauer) filter design.

Purpose

Map variables using bilinear transformation.

Syntax

```
[zd,pd,kd] = bilinear(z,p,k,Fs)
[zd,pd,kd] = bilinear(z,p,k,Fs,Fp)
[numd,dend] = bilinear(num,den,Fs)
[numd,dend] = bilinear(num,den,Fs,Fp)
[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,Fs)
[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,Fs,Fp)
```

Description

The *bilinear transformation* is a mathematical mapping of variables. In digital filtering, it is a standard method of mapping the *s* or analog plane into the *z* or digital plane. It transforms analog filters, designed using classical filter design techniques, into their discrete equivalents.

The bilinear transformation maps the *s*-plane into the *z*-plane by

$$H(z) = H(s)\Big|_{s=2f_s\frac{z-1}{z+1}}$$

This transformation maps the $j\Omega$ axis (from $\Omega = -\infty$ to $+\infty$) repeatedly around the unit circle ($\exp(j\omega)$, from $\omega = -\pi$ to π) by

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

`bilinear` can accept an optional parameter `Fp` that specifies prewarping. `Fp`, in Hertz, indicates a "match" frequency, that is, a frequency for which the frequency responses before and after mapping match exactly. In prewarped mode, the bilinear transformation maps the *s*-plane into the *z*-plane with

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

With the prewarping option, `bilinear` maps the $j\Omega$ axis (from $\Omega = -\infty$ to $+\infty$) repeatedly around the unit circle ($\exp(j\omega)$, from $\omega = -\pi$ to π) by

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

In prewarped mode, `bilinear` matches the frequency $2\pi f_p$ (in radians per second) in the *s*-plane to the normalized frequency $2\pi f_p/f_s$ (in radians per second) in the *z*-plane.

The `bilinear` function works with three different linear system representations: zero-pole-gain, transfer function, and state-space form.

Zero-Pole-Gain

`[zd,pd,kd] = bilinear(z,p,k,Fs)` and

`[zd,pd,kd] = bilinear(z,p,k,Fs,Fp)` convert the *s*-domain transfer function specified by `z`, `p`, and `k` to a discrete equivalent. Inputs `z` and `p` are column vectors containing the zeros and poles, and `k` is a scalar gain. `Fs` is the sampling frequency in Hertz. `bilinear` returns the discrete equivalent in column vectors `zd` and `pd` and scalar `kd`. `Fp` is the optional match frequency, in Hertz, for prewarping.

Transfer Function

`[numd,dend] = bilinear(num,den,Fs)` and

`[numd,dend] = bilinear(num,den,Fs,Fp)` convert an *s*-domain transfer function given by `num` and `den` to a discrete equivalent. Row vectors `num` and `den` specify the coefficients of the numerator and denominator, respectively, in descending powers of *s*

$$\frac{num(s)}{den(s)} = \frac{num(1)s^{nn} + \cdots + num(nn)s + num(nn+1)}{den(1)s^{nd} + \cdots + den(nd)s + den(nd+1)}$$

`Fs` is the sampling frequency in Hertz. `bilinear` returns the discrete equivalent in row vectors `numd` and `dend` in descending powers of *z* (ascending powers of z^{-1}). `Fp` is the optional match frequency, in Hertz, for prewarping.

State-Space

`[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,Fs)` and

`[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,Fs,Fp)` convert the continuous-time state-space system in matrices A, B, C, D,

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

to the discrete-time system

$$x[n+1] = A_d x[n] + B_d u[n]$$
$$y[n] + C_d x[n] + D_d u[n]$$

Fs is the sampling frequency in Hertz. `bilinear` returns the discrete equivalent in matrices Ad, Bd, Cd, Dd. Fp is the optional match frequency, in Hertz, for prewarping.

Algorithm

`bilinear` uses one of two algorithms, depending on the format of the input linear system you supply. One algorithm works on the zero-pole-gain format and the other on the state-space format. For transfer function representations, `bilinear` converts to state-space form, performs the transformation, and converts the resulting state-space system back to transfer function form.

Zero-Pole-Gain Algorithm

For a system in zero-pole-gain form, `bilinear` performs four steps:

1. If Fp is present, `k = 2*pi*Fp/tan(pi*Fp/Fs)`; otherwise `k = 2*Fs`.
2. It strips any zeros at plus or minus infinity using

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

3 It transforms the zeros, poles, and gain using

```
pd = (1+p/k)./(1-p/k);
zd = (1+z/k)./(1-z/k);
kd = real(k*prod(fs-z)./prod(fs-p));
```

4 It adds extra zeros at -1 so the resulting system has equivalent numerator and denominator order.

State-Space Algorithm

For a system in state-space form, `bilinear` performs two steps:

1 If `Fp` is present, `k = 2*pi*Fp/tan(pi*Fp/Fs)`; else `k = 2*Fs`.

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

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

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

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

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

`bilinear` implements these relations using conventional MATLAB statements. The scalar `r` is arbitrary; `bilinear` uses `sqrt(2/k)` to ensure good quantization noise properties in the resulting system.

Diagnostics

`bilinear` requires that the numerator order be no greater than the denominator order. If this is not the case, `bilinear` displays:

```
Numerator cannot be higher order than denominator.
```

For `bilinear` to distinguish between the zero-pole-gain and transfer function linear system formats, the first two input parameters must be vectors with the same orientation in these cases. If this is not the case, `bilinear` displays:

```
First two arguments must have the same orientation.
```

See Also

`impinvar`	Impulse invariance method of analog-to-digital filter conversion.
`lp2bp`	Lowpass to bandpass analog filter transformation.
`lp2bs`	Lowpass to bandstop analog filter transformation.
`lp2hp`	Lowpass to highpass analog filter transformation.
`lp2lp`	Lowpass to lowpass analog filter transformation.

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Pgs. 209-213.

[2] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. Pgs. 415-430.

blackman

Purpose Blackman window.

Syntax

```
w = blackman(n)
```

Description

`w = blackman(n)` returns the `n`-point Blackman window in the column vector `w`. The equation for a Blackman window is

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

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

Algorithm

```
w = (0.42 - 0.5*cos(2*pi*(0:N-1)/(N-1)) + ...
0.08*cos(4*pi*(0:N-1)/(N-1)))';
```

See Also

`bartlett`	Bartlett window.
`boxcar`	Rectangular window.
`chebwin`	Chebyshev window.
`hamming`	Hamming window.
`hanning`	Hanning window.
`kaiser`	Kaiser window.
`triang`	Triangular window.

Purpose Rectangular window.

Syntax

```
w = boxcar(n)
```

Description `w = boxcar(n)` returns a rectangular window of length `n` in the column vector `w`. This function is provided for completeness; a rectangular window is equivalent to no window at all.

Algorithm

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

See Also

`bartlett`	Bartlett window.
`blackman`	Blackman window.
`chebwin`	Chebyshev window.
`hamming`	Hamming window.
`hanning`	Hanning window.
`kaiser`	Kaiser window.
`triang`	Triangular window.

buttap

Purpose

Butterworth analog lowpass filter prototype.

Syntax

```
[z,p,k] = buttap(n)
```

Description

`[z,p,k] = buttap(n)` returns the zeros, poles, and gain of an order `n` Butterworth analog lowpass filter prototype. It returns the poles in the length `n` column vector `p` and the gain in scalar `k`. `z` is an empty matrix, because there are no zeros. The transfer function is

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

Butterworth filters are characterized by a magnitude response that is maximally flat in the passband and monotonic overall. In the lowpass case, the first 2n–1 derivatives of the squared magnitude response are zero at $\omega = 0$. The squared magnitude response function is

$$|H(\omega)|^2 = \frac{1}{1+(\omega/\omega_0)^{2n}}$$

corresponding to a transfer function with poles equally spaced around a circle in the left half plane. The magnitude response at the cutoff frequency ω_0 is always `1/sqrt(2)`, regardless of the filter order. `buttap` sets ω_0 to 1 for a normalized result.

Algorithm

```
z = [];
p = exp(sqrt(-1)*(pi*(1:2:2*n-1)/(2*n)+pi/2)).';
k = real(prod(-p));
```

See Also

`besselap`	Bessel analog lowpass filter prototype.
`butter`	Butterworth analog and digital filter design.
`cheb1ap`	Chebyshev type I analog lowpass filter prototype.
`cheb2ap`	Chebyshev type II analog lowpass filter prototype.
`ellipap`	Elliptic analog lowpass filter prototype.

References

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

Purpose

Butterworth analog and digital filter design.

Syntax

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

Description

`butter` designs lowpass, bandpass, highpass, and bandstop digital and analog Butterworth filters. Butterworth filters are characterized by a magnitude response that is maximally flat in the passband and monotonic overall.

Butterworth filters sacrifice rolloff steepness for monotonicity in the pass- and stopbands. Unless the smoothness of the Butterworth filter is needed, an elliptic or Chebyshev filter can generally provide steeper rolloff characteristics with a lower filter order.

Digital Domain

`[b,a] = butter(n,Wn)` designs an order `n` lowpass digital Butterworth filter with cutoff frequency `Wn`. It returns the filter coefficients in length `n + 1` row vectors `b` and `a`, with coefficients in descending powers of z:

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

Cutoff frequency is that frequency where the magnitude response of the filter is `sqrt(1/2)`. For `butter`, the cutoff frequency `Wn` must be a number between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency).

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

[b,a] = butter(n,Wn,'*ftype*') designs a highpass or bandstop filter, where *ftype* is

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

  The stopband is $w1 < \omega < w2$.

With different numbers of output arguments, butter directly obtains other realizations of the filter. To obtain zero-pole-gain form, use three output arguments:

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

[z,p,k] = butter(n,Wn,'*ftype*')

butter returns the zeros and poles in length n column vectors z and p, and the gain in the scalar k.

To obtain state-space form, use four output arguments:

[A,B,C,D] = butter(n,Wn) or

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

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] = Cx[n] + Du[n]$$

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

Analog Domain

[b,a] = butter(n,Wn,'s') designs an order n lowpass analog Butterworth filter with cutoff frequency Wn. It returns the filter coefficients in the length n + 1 row vectors b and a, in descending powers of s:

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

butter's cutoff frequency Wn must be greater than 0.

If `Wn` is a two-element vector with `w1 < w2`, `butter(n,Wn,'s')` returns an order `2*n` bandpass analog filter with passband `w1` < ω < `w2`.

`[b,a] = butter(n,Wn,'ftype','s')` designs a highpass or bandstop filter, where *ftype* is

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

  The stopband is `w1` < ω < `w2`.

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

`[z,p,k] = butter(n,Wn,'s')` or

`[z,p,k] = butter(n,Wn,'ftype','s')` returns the zeros and poles in length `n` or `2*n` column vectors `z` and `p` and the gain in the scalar `k`.

To obtain state-space form, use four output arguments:

`[A,B,C,D] = butter(n,Wn,'s')` or

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

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

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

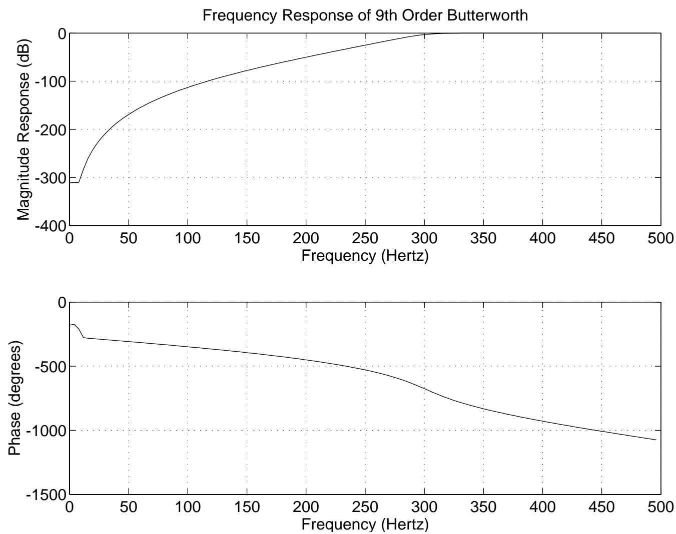
Examples

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

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

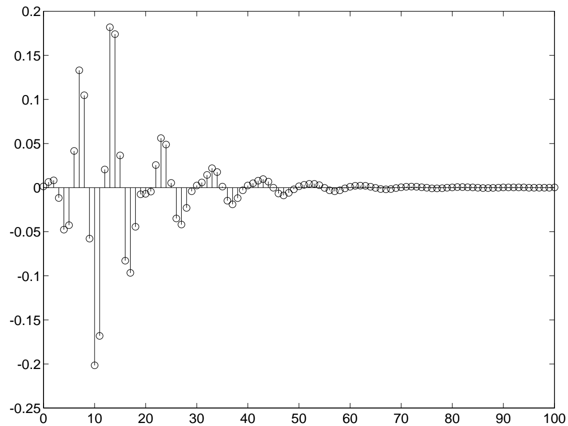
The filter's frequency response is

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



Design a 10th-order bandpass Butterworth filter with a passband from 100 to 200 Hz and plot its impulse response, or *unit sample response*:

```
n = 5; Wn = [100 200]/500;
[b,a] = butter(n,Wn);
[y,t] = impz(b,a,101);
stem(t,y)
```



Limitations

For high order filters, the state-space form is the most numerically accurate, followed by the zero-pole-gain form. The transfer function coefficient form is the least accurate; numerical problems can arise for filter orders as low as 15.

Algorithm

`butter` uses a five-step algorithm:

1. It finds the lowpass analog prototype poles, zeros, and gain using the `buttap` function.
2. It converts the poles, zeros, and gain into state-space form.
3. It transforms the lowpass filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies, using a state-space transformation.
4. For digital filter design, `butter` uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency pre-warping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at `Wn` or `w1` and `w2`.
5. It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also

`besself`	Bessel analog filter design.
`buttap`	Butterworth analog lowpass filter prototype.
`buttord`	Butterworth filter order selection.
`cheby1`	Chebyshev type I filter design (passband ripple).
`cheby2`	Chebyshev type II filter design (stopband ripple).
`ellip`	Elliptic (Cauer) filter design.
`maxflat`	Generalized digital Butterworth filter design.

buttord

Purpose Butterworth filter order selection.

Syntax

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

Description buttord selects the minimum order digital or analog Butterworth filter required to meet a set of filter design specifications:

Wp	Passband corner frequency. Wp, the cutoff frequency, has a value between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency).
Ws	Stopband corner frequency. Ws is in the same units as Wp; it has a value between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency).
Rp	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels. The passband is 0 < w < Wp.
Rs	Stopband attenuation, in decibels. This value is the number of decibels the stopband is down from the passband. The stopband is Ws < w < 1.

Digital Domain

[n,Wn] = buttord(Wp,Ws,Rp,Rs) returns the order n of the lowest order digital Butterworth filter that loses no more than Rp dB in the passband and has at least Rs dB of attenuation in the stopband. The passband runs from 0 to Wp and the stopband runs from Ws to 1, the Nyquist frequency. buttord also returns Wn, the Butterworth cutoff frequency that allows butter to achieve the given specifications (the "3 dB" frequency).

Use buttord for highpass, bandpass, and bandstop filters. For highpass filters, Wp is greater than Ws. For bandpass and bandstop filters, Wp and Ws are two-element vectors that specify the corner frequencies at both edges of the filter, lower frequency edge first. For the band filters, buttord returns Wn as a two-element row vector for input to butter.

If filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design the filter as separate lowpass and highpass sections and cascade the two filters together.

Analog Domain

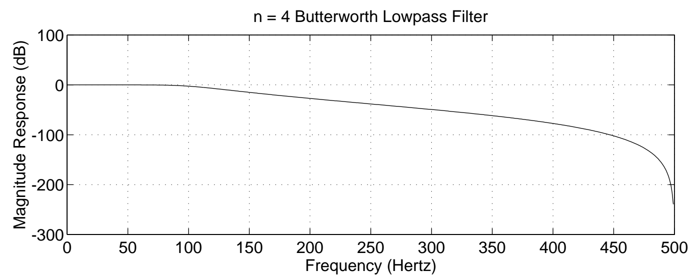
`[n,Wn] = buttord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Wn` for an analog filter. In this case the frequencies in `Wp` and `Ws` are in radians per second and may be greater than 1.

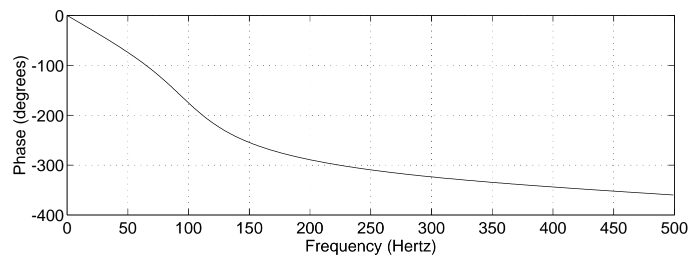
Use `buttord` for highpass, bandpass, and bandstop filters, as described under "Digital Domain."

Examples

For 1000 Hz data, design a lowpass filter with less than 3 dB of attenuation from 0 to 100 Hz, and attenuation at least 15 dB from 150 Hz to the Nyquist frequency. Plot the filter's frequency response:

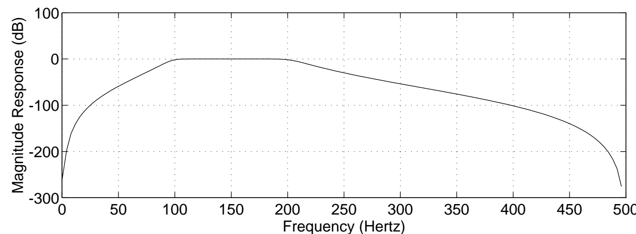
```
Wp = 100/500; Ws = 150/500;
[n,Wn] = buttord(Wp,Ws,3,15)
n =
     4
Wn =
    0.2042
[b,a] = butter(n,Wn);
freqz(b,a,512,1000)
```

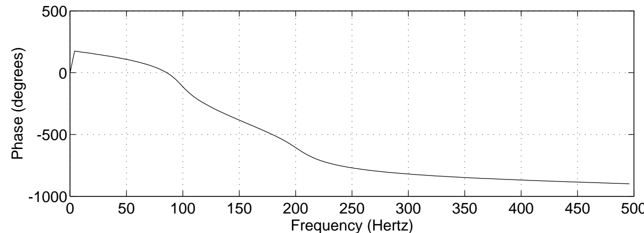




Next design a bandpass filter with passband of 100 Hz to 200 Hz, less than 3 dB of attenuation at the passband corners, and attenuation down 30 dB by 50 Hz out on both sides of the passband:

```
Wp = [100 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 30;
[n,Wn] = buttord(Wp,Ws,Rp,Rs); [b,a] = butter(n,Wn);
freqz(b,a,128,1000)
```





Algorithm

`buttord`'s order prediction formula is described in [1]. It operates in the analog domain for both analog and digital cases. For the digital case, it converts the frequency parameters to the *s*-domain before estimating the order and natural frequency, then converts back to the *z*-domain.

`buttord` initially develops a lowpass filter prototype by transforming the passband frequencies of the desired filter to 1 rad/sec (for low- and highpass filters) and to -1 and 1 rad/sec (for bandpass and bandstop filters). It then computes the minimum order required for a lowpass filter to meet the stopband specification.

See Also

`butter`	Butterworth analog and digital filter design.
`cheb1ord`	Chebyshev type I filter order selection.
`cheb2ord`	Chebyshev type II filter order selection.
`ellipord`	Elliptic filter order selection.

References

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

Purpose Complex cepstral analysis.

Syntax

```
xhat = cceps(x)
[xhat,nd] = cceps(x)
[...] = cceps(x,n)
```

Description Cepstral analysis is a nonlinear signal processing technique that is applied most commonly in speech processing and homomorphic filtering [1].

`xhat = cceps(x)` returns the complex cepstrum of the (assumed real) sequence `x`. The input is altered, by the application of a linear phase term, to have no phase discontinuity at $\pm\pi$ radians. That is, it is circularly shifted (after zero padding) by some samples, if necessary, to have zero phase at π radians.

`[xhat,nd] = cceps(x)` returns the number of samples `nd` of (circular) delay added to `x` prior to finding the complex cepstrum.

`[...] = cceps(x,n)` zero pads `x` to length `n` and returns the length `n` complex cepstrum of `x`.

Algorithm `cceps`, in its basic form, is an M-file implementation of algorithm 7.1 in [2]. A lengthy Fortran program reduces to three lines of MATLAB code:

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

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

See Also

`icceps`	Inverse complex cepstrum.
`hilbert`	Hilbert transform.
`rceps`	Real cepstrum and minimum phase reconstruction.
`unwrap`	Unwrap phase angles.

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[2] IEEE. *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979.

Purpose Chebyshev type I analog lowpass filter prototype.

Syntax

```
[z,p,k] = cheb1ap(n,Rp)
```

Description `[z,p,k] = cheb1ap(n,Rp)` returns the zeros, poles, and gain of an order `n` Chebyshev type I analog lowpass filter prototype with `Rp` dB of ripple in the passband. It returns the poles in the length `n` column vector `p` and the gain in scalar `k`. `z` is an empty matrix, because there are no zeros. The transfer function is

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

Chebyshev type I filters are equiripple in the passband and monotonic in the stopband. The poles are evenly spaced about an ellipse in the left half plane. The Chebyshev type I cutoff frequency ω_0 is set to 1.0 for a normalized result. This is the frequency at which the passband ends and the filter has magnitude response of $10^{-Rp/20}$.

See Also

`besselap`	Bessel analog lowpass filter prototype.
`buttap`	Butterworth analog and digital filter design.
`cheb2ap`	Chebyshev type I analog lowpass filter prototype.
`cheby1`	Chebyshev type I filter design (passband ripple).
`ellipap`	Elliptic analog lowpass filter prototype.

References

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

cheb1ord

Purpose

Chebyshev type I filter order selection.

Syntax

```
[n,Wn] = cheb1ord(Wp,Ws,Rp,Rs)
[n,Wn] = cheb1ord(Wp,Ws,Rp,Rs,'s')
```

Description

cheb1ord selects the minimum order digital or analog Chebyshev type I filter required to meet a set of filter design specifications:

Wp	Passband corner frequency. Wp, the cutoff frequency, has a value between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency).
Ws	Stopband corner frequency. Ws is in the same units as Wp; it has a value between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency).
Rp	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels. The passband is 0 < w < Wp.
Rs	Stopband attenuation, in decibels. This value is the number of decibels the stopband is down from the passband. The stopband is Ws < w < 1.

Digital Domain

[n,Wn] = cheb1ord(Wp,Ws,Rp,Rs) returns the order n of the lowest order Chebyshev filter that loses no more than Rp dB in the passband and has at least Rs dB of attenuation in the stopband. The passband runs from 0 to Wp and the stop-band runs from Ws to 1, the Nyquist frequency. cheb1ord also returns Wn, the Chebyshev type I cutoff frequency that allows cheby1 to achieve the given specifications.

Use cheb1ord for lowpass, highpass, bandpass, and bandstop filters. For high-pass filters, Wp > Ws. For bandpass and bandstop filters, Wp and Ws are two-element vectors that specify the corner frequencies at both edges of the filter, lower frequency edge first. For the band filters, cheb1ord returns Wn as a two-element row vector for input to cheby1.

If filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design the filter as separate lowpass and highpass sections and cascade the two filters together.

Analog Domain

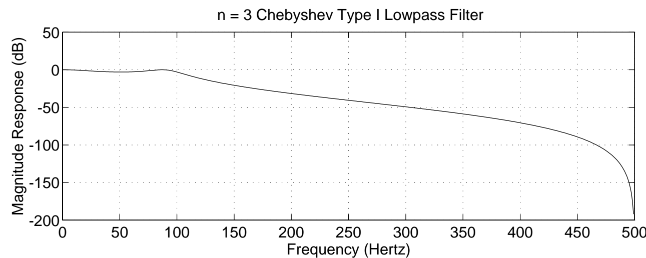
`[n,Wn] = cheb1ord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Wn` for an analog filter. In this case the frequencies in `Wp` and `Ws` are in radians per second and may be greater than 1.

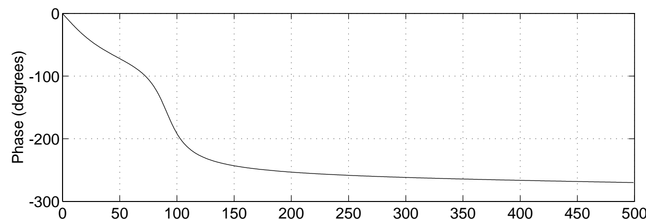
Use `cheb1ord` for lowpass, highpass, bandpass, and bandstop filters, as described under "Digital Domain."

Examples

For 1000 Hz data, design a lowpass filter with less than 3 dB of attenuation from 0 to 100 Hz and attenuation at least 15 dB from 150 Hz to the Nyquist frequency:

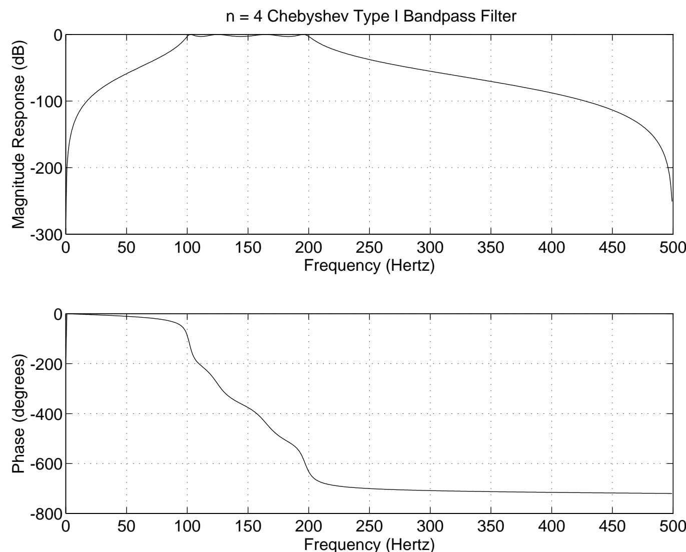
```
Wp = 100/500;  Ws = 150/500;
Rp = 3;  Rs = 15;
[n,Wn] = cheb1ord(Wp,Ws,Rp,Rs)

n =

     3
Wn =

    0.2000

[b,a] = cheby1(n,Rp,Wn);
freqz(b,a,512,1000)
```





Next design a bandpass filter with a passband of 100 Hz to 200 Hz, less than 3 dB of attenuation throughout the passband, and 30 dB stopbands 50 Hz out on both sides of the passband:

```
Wp = [100 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 30;
[n,Wn] = cheb1ord(Wp,Ws,Rp,Rs)

n =

     4

Wn =

    0.2000    0.4000

[b,a] = cheby1(n,Rp,Wn);
freqz(b,a,512,1000)
```



Algorithm

`cheb1ord` uses the Chebyshev lowpass filter order prediction formula described in [1]. The function performs its calculations in the analog domain for both analog and digital cases. For the digital case, it converts the frequency

parameters to the *s*-domain before the order and natural frequency estimation process, then converts them back to the *z*-domain.

`cheb1ord` initially develops a lowpass filter prototype by transforming the passband frequencies of the desired filter to 1 rad/sec (for low- or highpass filters) or to -1 and 1 rad/sec (for bandpass or bandstop filters). It then computes the minimum order required for a lowpass filter to meet the stopband specification.

See Also

`buttord`	Butterworth filter order selection.
`cheby1`	Chebyshev type I filter design (passband ripple).
`cheb2ord`	Chebyshev type II filter order selection.
`ellipord`	Elliptic filter order selection.

References

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

cheb2ap

Purpose

Chebyshev type II analog lowpass filter prototype.

Syntax

```
[z,p,k] = cheb2ap(n,Rs)
```

Description

`[z,p,k] = cheb2ap(n,Rs)` finds the zeros, poles, and gain of an order `n` Chebyshev type II analog lowpass filter prototype with stopband ripple `Rs` dB down from the passband peak value. `cheb2ap` returns the zeros and poles in length `n` column vectors `z` and `p` and the gain in scalar `k`. If `n` is odd, `z` is length `n–1`. The transfer function is

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

Chebyshev type II filters are monotonic in the passband and equiripple in the stopband. The pole locations are the inverse of the pole locations of `cheb1ap`, whose poles are evenly spaced about an ellipse in the left half plane. The Chebyshev type II cutoff frequency ω_0 is set to 1 for a normalized result. This is the frequency at which the stopband begins and the filter has magnitude response of $10^{-Rs/20}$.

Algorithm

Chebyshev type II filters are sometimes called *inverse Chebyshev* filters because of their relationship to Chebyshev type I filters. The `cheb2ap` function is a modification of the Chebyshev type I prototype algorithm:

1. `cheb2ap` replaces the frequency variable ω with $1/\omega$, turning the lowpass filter into a highpass filter while preserving the performance at $\omega = 1$.
2. `cheb2ap` subtracts the filter transfer function from unity.

See Also

`besselap`	Bessel analog lowpass filter prototype.
`buttap`	Butterworth analog lowpass filter prototype.
`cheb1ap`	Chebyshev type I analog lowpass filter prototype.
`cheby2`	Chebyshev type II filter design (stopband ripple).
`ellipap`	Elliptic analog lowpass filter prototype.

References

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

Purpose

Chebyshev type II filter order selection.

Syntax

```
[n,Wn] = cheb2ord(Wp,Ws,Rp,Rs)
[n,Wn] = cheb2ord(Wp,Ws,Rp,Rs,'s')
```

Description

`cheb2ord` selects the minimum order digital or analog Chebyshev type II filter required to meet a set of filter design specifications:

`Wp`	Passband corner frequency. `Wp`, the cutoff frequency, has a value between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency).
`Ws`	Stopband corner frequency. `Ws` is in the same units as `Wp`; it has a value between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency).
`Rp`	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels. The passband is `0 < w < Wp`.
`Rs`	Stopband attenuation, in decibels. This value is the number of decibels the stopband is down from the passband. The stopband is `Ws < w < 1`.

Digital Domain

`[n,Wn] = cheb2ord(Wp,Ws,Rp,Rs)` returns the order `n` of the lowest order Chebyshev filter that loses no more than `Rp` dB in the passband and has at least `Rs` dB of attenuation in the stopband. The passband runs from 0 to `Wp` and the stopband runs from `Ws` to 1, the Nyquist frequency. `cheb2ord` also returns `Wn`, the Chebyshev type II cutoff frequency that allows `cheby2` to achieve the given specifications.

Use `cheb2ord` for lowpass, highpass, bandpass, and bandstop filters. For high-pass filters, `Wp` is greater than `Ws`. For bandpass and bandstop filters, `Wp` and `Ws` are two-element vectors that specify the corner frequencies at both edges of the filter, lower frequency edge first. For the band filters, `cheb2ord` returns `Wn` as a two-element row vector for input to `cheby2`.

If filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design the filter as separate lowpass and highpass sections and cascade the two filters together.

Analog Domain

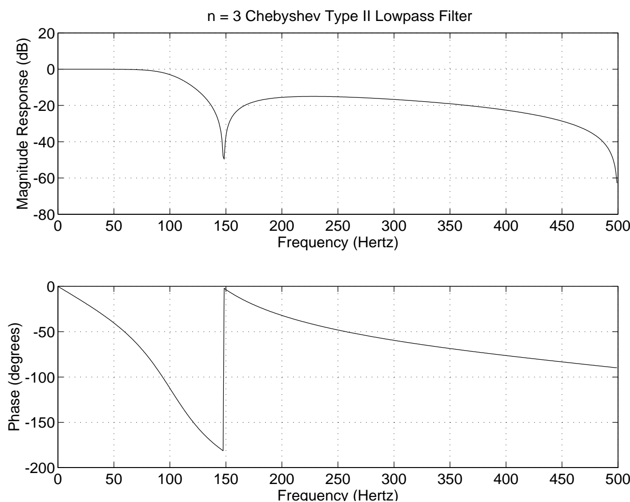
`[n,Wn] = cheb2ord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Wn` for an analog filter. In this case the frequencies in `Wp` and `Ws` are in radians per second and may be greater than 1.

Use `cheb2ord` for lowpass, highpass, bandpass, and bandstop filters, as described under "Digital Domain."

Examples

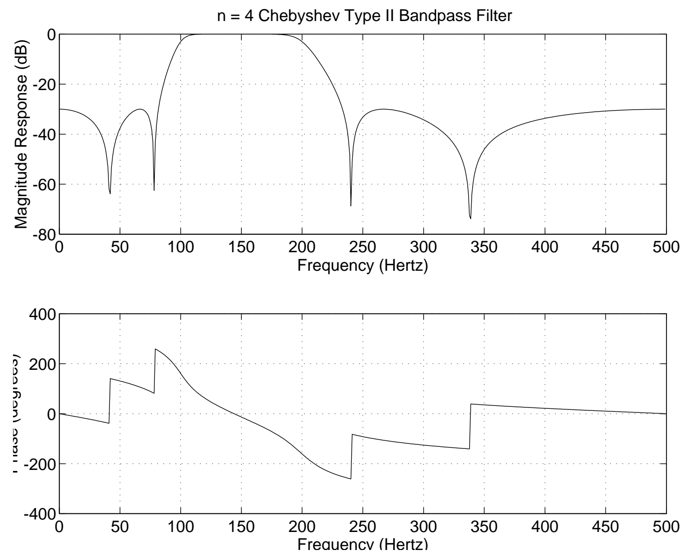
For 1000 Hz data, design a lowpass filter with less than 3 dB of attenuation from 0 to 100 Hz, and attenuation at least 15 dB from 150 Hz to the Nyquist frequency:

```
Wp = 100/500;  Ws = 150/500;
Rp = 3;  Rs = 15;
[n,Wn] = cheb2ord(Wp,Ws,Rp,Rs)
n =

     3

Wn =

    0.2609

[b,a] = cheby2(n,Rs,Wn);
freqz(b,a,512,1000)
```



Next design a bandpass filter with a passband of 100 Hz to 200 Hz, less than 3 dB of attenuation throughout the passband, and 30 dB stopbands 50 Hz out on both sides of the passband:

```
Wp = [100 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 30;
[n,Wn] = cheb2ord(Wp,Ws,Rp,Rs)

n =

     4

Wn =

    0.1633    0.4665

[b,a] = cheby2(n,Rs,Wn);
freqz(b,a,512,1000)
```



Algorithm

cheb2ord uses the Chebyshev lowpass filter order prediction formula described in [1]. The function performs its calculations in the analog domain for both analog and digital cases. For the digital case, it converts the frequency param-

eters to the *s*-domain before the order and natural frequency estimation process, then converts them back to the *z*-domain.

`cheb2ord` initially develops a lowpass filter prototype by transforming the stopband frequencies of the desired filter to 1 rad/sec (for low- and highpass filters) and to -1 and 1 rad/sec (for bandpass and bandstop filters). It then computes the minimum order required for a lowpass filter to meet the passband specification.

See Also

`buttord`	Butterworth filter order selection.
`cheb1ord`	Chebyshev type I filter order selection.
`cheby2`	Chebyshev type II filter design (stopband ripple).
`ellipord`	Elliptic filter order selection.

References

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

Purpose

Chebyshev window.

Syntax

```
w = chebwin(n,r)
```

Description

`w = chebwin(n,r)` returns the column vector `w`, containing the length `n` Chebyshev window whose Fourier transform magnitude sidelobe ripple is `r` dB below the mainlobe magnitude, where `n` is odd. If `n` is even, the window is length `n+1`.

Diagnostics

When `n` is an even number, `chebwin` returns a window of order `n+1` and displays the warning message:

```
chebwin: N must be odd - N is being increased by 1.
```

See Also

`bartlett`	Bartlett window.
`blackman`	Blackman window.
`boxcar`	Rectangular window.
`hamming`	Hamming window.
`hanning`	Hanning window.
`kaiser`	Kaiser window.
`triang`	Triangular window.

References

[1] IEEE. *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979. Program 5.2.

Purpose

Chebyshev type I filter design (passband ripple).

Syntax

```
[b,a] = cheby1(n,Rp,Wn)
[b,a] = cheby1(n,Rp,Wn,'ftype')
[b,a] = cheby1(n,Rp,Wn,'s')
[b,a] = cheby1(n,Rp,Wn,'ftype','s')
[z,p,k] = cheby1(...)
[A,B,C,D] = cheby1(...)
```

Description

`cheby1` designs lowpass, bandpass, highpass, and bandstop digital and analog Chebyshev type I filters. Chebyshev type I filters are equiripple in the passband and monotonic in the stopband. Type I filters roll off faster than type II filters, but at the expense of greater deviation from unity in the passband.

Digital Domain

`[b,a] = cheby1(n,Rp,Wn)` designs an order `n` lowpass digital Chebyshev filter with cutoff frequency `Wn` and `Rp` dB of ripple in the passband. It returns the filter coefficients in the length `n+1` row vectors `b` and `a`, with coefficients in descending powers of z:

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

Cutoff frequency is the frequency at which the magnitude response of the filter is equal to –`Rp` dB. For `cheby1`, the cutoff frequency `Wn` is a number between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency). Smaller values of passband ripple `Rp` lead to wider transition widths (shallower rolloff characteristics).

If `Wn` is a two-element vector, `Wn = [w1 w2]`, `cheby1` returns an order `2*n` bandpass filter with passband `w1` $< \omega <$ `w2`.

`[b,a] = cheby1(n,Rp,Wn,'ftype')` designs a highpass or bandstop filter, where *ftype* is

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

  The stopband is `w1` < ω < `w2`.

With different numbers of output arguments, `cheby1` directly obtains other realizations of the filter. To obtain zero-pole-gain form, use three output arguments:

`[z,p,k] = cheby1(n,Rp,Wn)` or

`[z,p,k] = cheby1(n,Rp,Wn,'ftype')` returns the zeros and poles in length `n` column vectors `z` and `p` and the gain in the scalar `k`.

To obtain state-space form, use four output arguments:

`[A,B,C,D] = cheby1(n,Rp,Wn)` or

`[A,B,C,D] = cheby1(n,Rp,Wn,'ftype')` where `A`, `B`, `C`, and `D` are

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] = Cx[n] + Du[n]$$

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

Analog Domain

`[b,a] = cheby1(n,Rp,Wn,'s')` designs an order `n` lowpass analog Chebyshev type I filter with cutoff frequency `Wn`. It returns the filter coefficients in length `n + 1` row vectors `b` and `a`, in descending powers of s:

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

Cutoff frequency is the frequency at which the magnitude response of the filter is –`Rp` dB. For `cheby1`, the cutoff frequency `Wn` must be greater than 0.

If Wn is a two-element vector, Wn = [w1 w2], with w1 < w2, then cheby1(n,Rp,Wn,'s') returns an order 2*n bandpass analog filter with passband w1 < ω < w2.

[b,a] = cheby1(n,Rp,Wn,'*ftype*','s') designs a highpass or bandstop filter, where *ftype* is

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

  The stopband is w1 < ω < w2.

You can supply different numbers of output arguments for cheby1 to directly obtain other realizations of the analog filter. To obtain zero-pole-gain form, use three output arguments:

[z,p,k] = cheby1(n,Rp,Wn,'s') or

[z,p,k] = cheby1(n,Rp,Wn,'*ftype*','s') returns the zeros and poles in length n or 2*n column vectors z and p and the gain in the scalar k.

To obtain state-space form, use four output arguments:

[A,B,C,D] = cheby1(n,Rp,Wn,'s') or

[A,B,C,D] = cheby1(n,Rp,Wn,'*ftype*','s') where A, B, C, and D are defined as

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

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

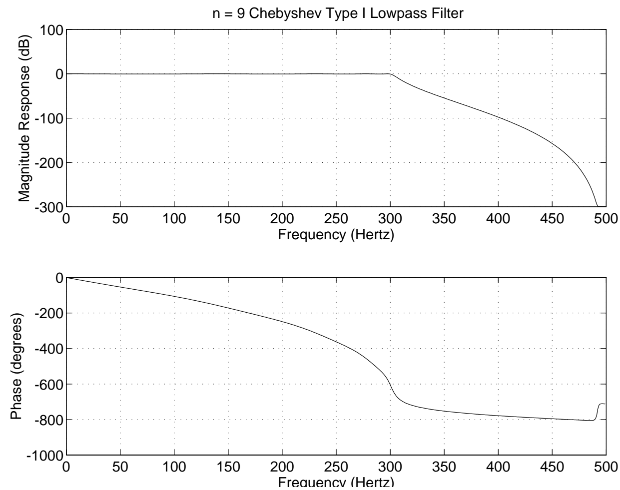
Examples

For data sampled at 1000 Hz, design a 9th-order lowpass Chebyshev type I filter with 0.5 dB of ripple in the passband and a cutoff frequency of 300 Hz:

```
[b,a] = cheby1(9,0.5,300/500);
```

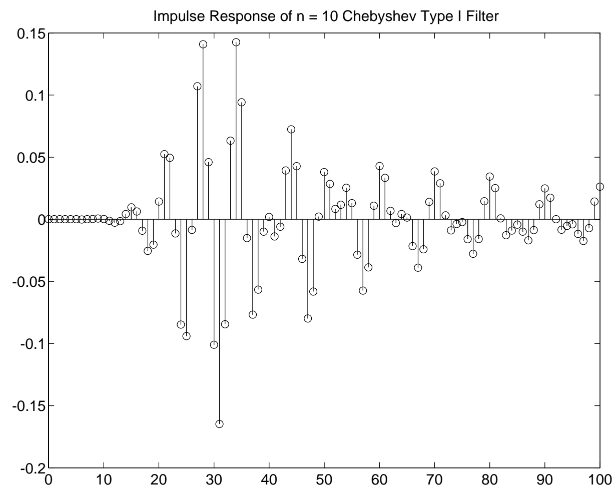
The frequency response of the filter is

```
freqz(b,a,512,1000)
```



Design a 10th-order bandpass Chebyshev type I filter with a passband from 100 to 200 Hz and plot its impulse response:

```
n = 10; Rp = 0.5;
Wn = [100 200]/500;
[b,a] = cheby1(n,Rp,Wn);
[y,t] = impz(b,a,101); stem(t,y)
```



Limitations

For high order filters, the state-space form is the most numerically accurate, followed by the zero-pole-gain form. The transfer function form is the least accurate; numerical problems can arise for filter orders as low as 15.

Algorithm

cheby1 uses a five-step algorithm:

1. It finds the lowpass analog prototype poles, zeros, and gain using the cheb1ap function.
2. It converts the poles, zeros, and gain into state-space form.
3. It transforms the lowpass filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies, using a state-space transformation.
4. For digital filter design, cheby1 uses bilinear to convert the analog filter into a digital filter through a bilinear transformation with frequency pre-warping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at Wn or w1 and w2.
5. It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also

besself	Bessel analog filter design.
butter	Butterworth analog and digital filter design.
cheb1ap	Chebyshev type I analog lowpass filter prototype.
cheb1ord	Chebyshev type I filter order selection.
cheby2	Chebyshev type II filter design (stopband ripple).
ellip	Elliptic (Cauer) filter design.

Purpose

Chebyshev type II filter design (stopband ripple).

Syntax

```
[b,a] = cheby2(n,Rs,Wn)
[b,a] = cheby2(n,Rs,Wn,'ftype')
[b,a] = cheby2(n,Rs,Wn,'s')
[b,a] = cheby2(n,Rs,Wn,'ftype','s')
[z,p,k] = cheby2(...)
[A,B,C,D] = cheby2(...)
```

Description

cheby2 designs lowpass, highpass, bandpass, and bandstop digital and analog Chebyshev type II filters. Chebyshev type II filters are monotonic in the passband and equiripple in the stopband. Type II filters do not roll off as fast as type I filters, but are free of passband ripple.

Digital Domain

[b,a] = cheby2(n,Rs,Wn) designs an order n lowpass digital Chebyshev type II filter with cutoff frequency Wn and stopband ripple Rs dB down from the peak passband value. It returns the filter coefficients in the length n + 1 row vectors b and a, with coefficients in descending powers of z.

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

Cutoff frequency is the beginning of the stopband, where the magnitude response of the filter is equal to –Rs dB. For cheby2, the cutoff frequency Wn is a number between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency). Larger values of stopband attenuation Rs lead to wider transition widths (shallower rolloff characteristics).

If Wn is a two-element vector, Wn = [w1 w2], cheby2 returns an order 2*n bandpass filter with passband w1 < ω < w2.

`[b,a] = cheby2(n,Rs,Wn,'ftype')` designs a highpass or bandstop filter, where *ftype* is

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

  The stopband is `w1` < ω < `w2`.

With different numbers of output arguments, `cheby2` directly obtains other realizations of the filter. To obtain zero-pole-gain form, use three output arguments:

`[z,p,k] = cheby2(n,Rs,Wn)` or

`[z,p,k] = cheby2(n,Rs,Wn,'ftype')` returns the zeros and poles in length `n` column vectors `z` and `p` and the gain in the scalar `k`.

To obtain state-space form, use four output arguments:

`[A,B,C,D] = cheby2(n,Rs,Wn)` or

`[A,B,C,D] = cheby2(n,Rs,Wn,'ftype')` where `A`, `B`, `C`, and `D` are

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] = Cx[n] + Du[n]$$

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

Analog Domain

`[b,a] = cheby2(n,Rs,Wn,'s')` designs an order `n` lowpass analog Chebyshev type II filter with cutoff frequency `Wn`. It returns the filter coefficients in the length `n + 1` row vectors `b` and `a`, with coefficients in descending powers of s:

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

Cutoff frequency is the frequency at which the magnitude response of the filter is equal to –`Rs` dB. For `cheby2`, the cutoff frequency `Wn` must be greater than 0.

If `Wn` is a two-element vector, `Wn = [w1 w2]`, with `w1 < w2`, then `cheby2(n,Rs,Wn,'s')` returns an order `2*n` bandpass analog filter with passband `w1` < ω < `w2`.

`[b,a] = cheby2(n,Rs,Wn,'ftype','s')` designs a highpass or bandstop filter, where *`ftype`* is

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

  The stopband is `w1` < ω < `w2`.

With different numbers of output arguments, `cheby2` directly obtains other realizations of the analog filter. To obtain zero-pole-gain form, use three output arguments:

`[z,p,k] = cheby2(n,Rs,Wn,'s')` or

`[z,p,k] = cheby2(n,Rs,Wn,'ftype','s')` returns the zeros and poles in length `n` or `2*n` column vectors `z` and `p` and the gain in the scalar `k`.

To obtain state-space form, use four output arguments:

`[A,B,C,D] = cheby2(n,Rs,Wn,'s')` or

`[A,B,C,D] = cheby2(n,Rs,Wn,'ftype','s')` where `A`, `B`, `C`, and `D` are

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

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

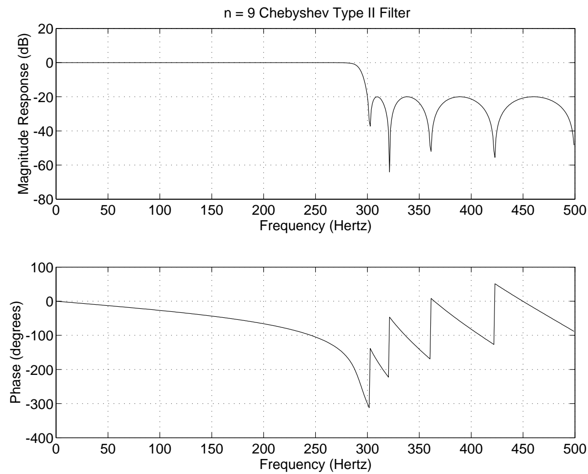
Examples

For data sampled at 1000 Hz, design a ninth-order lowpass Chebyshev type II filter with stopband attenuation 20 dB down from the passband and a cutoff frequency of 300 Hz:

```
[b,a] = cheby2(9,20,300/500);
```

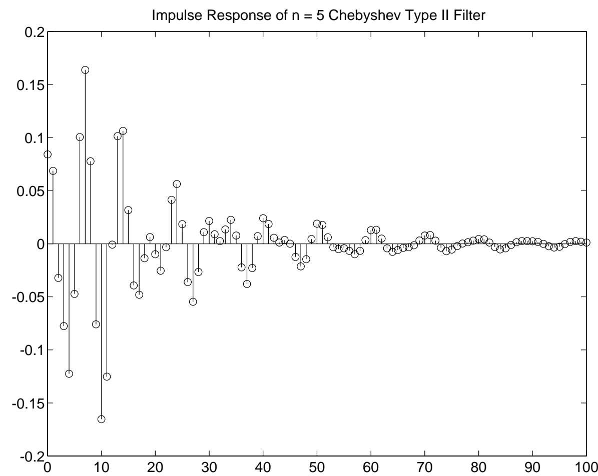
The frequency response of the filter is

```
freqz(b,a,512,1000)
```



Design a fifth-order bandpass Chebyshev type II filter with passband from 100 to 200 Hz and plot the impulse response of the filter:

```
n = 5; r = 20;
Wn = [100 200]/500;
[b,a] = cheby2(n,r,Wn);
[y,t] = impz(b,a,101); stem(t,y)
```



Limitations

For high order filters, the state-space form is the most numerically accurate, followed by the zero-pole-gain form. The transfer function coefficient form is the least accurate; numerical problems can arise for filter orders as low as 15.

Algorithm

`cheby2` uses a five-step algorithm:

1. It finds the lowpass analog prototype poles, zeros, and gain using the `cheb2ap` function.
2. It converts poles, zeros, and gain into state-space form.
3. It transforms the lowpass filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies, using a state-space transformation.
4. For digital filter design, `cheby2` uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at `Wn` or `w1` and `w2`.
5. It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also

`besself`	Bessel analog filter design.
`butter`	Butterworth analog and digital filter design.
`cheb2ap`	Chebyshev type II analog lowpass filter prototype.
`cheb2ord`	Chebyshev type II filter order selection.
`cheby1`	Chebyshev type I filter design (passband ripple).
`ellip`	Elliptic (Cauer) filter design.

chirp

Purpose

Swept-frequency cosine generator.

Syntax

```
y = chirp(t,f0,t1,f1)
y = chirp(t,f0,t1,f1,'method')
y = chirp(t,f0,t1,f1,'method',phi)
```

Description

`y = chirp(t,f0,t1,f1)` generates samples of a linear swept-frequency cosine signal at the time instances defined in array `t`, where `f0` is the instantaneous frequency at time 0, and `f1` is the instantaneous frequency at time `t1`. `f0` and `f1` are both in Hertz. If unspecified, `f0` is 0, `t1` is 1, and `f1` is 100.

`y = chirp(t,f0,t1,f1,'method')` specifies alternative sweep method options, where *`method`* can be

- `linear`, which specifies an instantaneous frequency sweep $f_i(t)$ given by

  $$f_i(t) = f_0 + \beta t$$

  where

  $$\beta = (f_1 - f_0)/t_1$$

  β ensures that the desired frequency breakpoint f_1 at time t_1 is maintained.
- `quadratic`, which specifies an instantaneous frequency sweep $f_i(t)$ given by

  $$f_i(t) = f_0 + \beta t^2$$

  where

  $$\beta = (f_1 - f_0)/t_1$$

- `logarithmic` specifies an instantaneous frequency sweep $f_i(t)$ given by

  $$f_i(t) = f_0 + 10^{\beta t}$$

  where

  $$\beta = [\log_{10}(f_1 - f_0)]/t_1$$

  For a log-sweep, `f1` must be greater than `f0`.

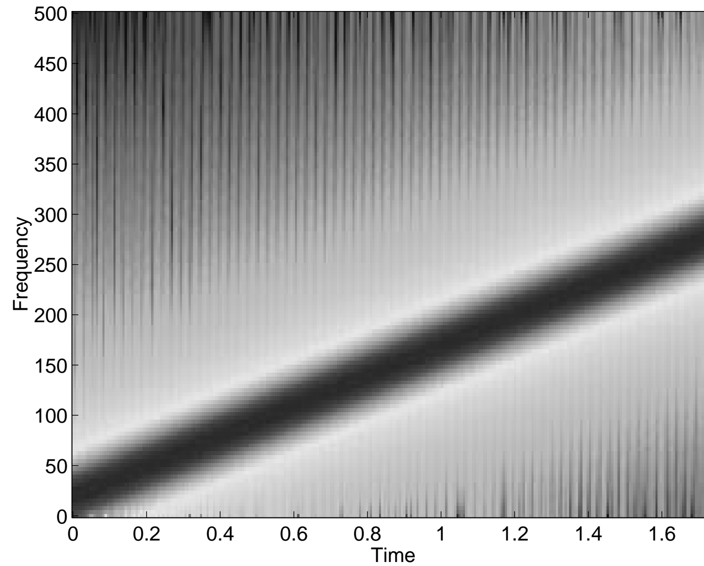
`y = chirp(t,f0,t1,f1,'method',phi)` allows an initial phase `phi` to be specified in degrees. If unspecified, `phi` is 0.

Default values are substituted for empty or omitted trailing input arguments.

Examples

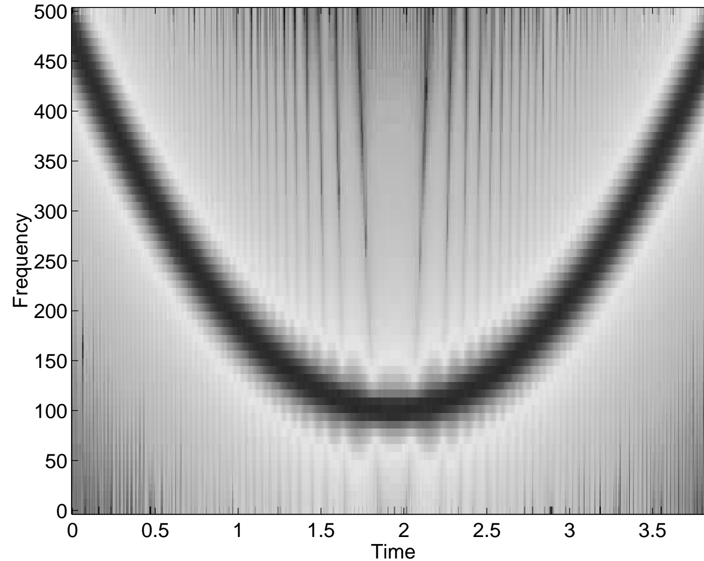
Compute the spectrogram of a linear chirp:

```
t = 0:0.001:2;                % 2 secs @ 1kHz sample rate
y = chirp(t,0,1,150);         % Start @ DC, cross 150Hz at t=1 sec
specgram(y,256,1e3,256,250)  % Display the spectrogram
```



Compute the spectrogram of a quadratic chirp:

```
t = -2:0.001:2;                          % ±2 secs @ 1kHz sample rate
y = chirp(t,100,1,200,'quadratic');      % Start @ 100Hz, cross 200Hz
                                         % at t=1 sec
specgram(y,128,1e3,128,120)              % Display the spectrogram
```



See Also

`cos`	Cosine of vector/matrix elements (see MATLAB Function Reference).
`diric`	Dirichlet or periodic sinc function.
`gauspuls`	Gaussian-modulated sinusoidal pulse generator.
`pulstran`	Pulse train generator.
`rectpuls`	Sampled aperiodic rectangle generator.
`sawtooth`	Sawtooth or triangle wave generator.
`sin`	Sine of vector/matrix elements (see MATLAB Function Reference).
`sinc`	Sinc or $\sin(\pi t)/\pi t$ function.
`square`	Square wave generator.
`tripuls`	Sampled aperiodic triangle generator.

Purpose Estimate magnitude squared coherence function between two signals.

Syntax

```
Cxy = cohere(x,y)
Cxy = cohere(x,y,nfft)
[Cxy,f] = cohere(x,y,nfft,Fs)
Cxy = cohere(x,y,nfft,Fs,window)
Cxy = cohere(x,y,nfft,Fs,window,noverlap)
Cxy = cohere(x,y,...,'dflag')
cohere(x,y)
```

Description

`Cxy = cohere(x,y)` finds the magnitude squared coherence between length `n` signal vectors `x` and `y`. The coherence is a function of the power spectra of `x` and `y` and the cross spectrum of `x` and `y`:

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

`x` and `y` must be the same length. `Cxy = cohere(x,y)` uses the following default values:

- `nfft = min(256,length(x))`
- `Fs = 2`
- `window = hanning(nfft)`
- `noverlap = 0`

`nfft` specifies the FFT length that `cohere` uses. This value determines the frequencies at which the coherence is estimated. `Fs` is a scalar that specifies the sampling frequency. `window` specifies a windowing function and the number of samples `cohere` uses in its sectioning of the `x` and `y` vectors. `noverlap` is the number of samples by which the sections overlap. Any arguments that you omit from the end of the parameter list use the default values shown above.

If `x` is real, `cohere` estimates the coherence function at positive frequencies only; in this case, the output `Cxy` is a column vector of length `nfft/2 + 1` for `nfft` even and `(nfft + 1)/2` for `n` odd. If `x` or `y` is complex, `cohere` estimates the coherence function at both positive and negative frequencies, and `Cxy` has length `nfft`.

`Cxy = cohere(x,y,nfft)` uses the FFT length `nfft` in estimating the power spectrum for `x`. Specify `nfft` as a power of 2 for fastest execution.

`[Cxy,f] = cohere(x,y,nfft,Fs)` returns a vector `f` of frequencies at which the function evaluates the coherence. `Fs` is the sampling frequency. `f` is the same size as `Cxy`, so `plot(f,Cxy)` plots the coherence function versus properly scaled frequency. `Fs` has no effect on the output `Cxy`; it is a frequency scaling multiplier.

`Cxy = cohere(x,y,nfft,Fs,window)` specifies a windowing function and the number of samples per section of the vectors `x` and `y`. If you supply a scalar for `window`, `cohere` uses a Hanning window of that length. The length of the window must be less than or equal to `nfft`; `cohere` zero pads the sections if the window length exceeds `nfft`.

`Cxy = cohere(x,y,nfft,Fs,window,noverlap)` overlaps the sections of `x` by `noverlap` samples.

You can use the empty matrix `[]` to specify the default value for any input argument except `x` or `y`. For example,

```
Cxy = cohere(x,y,[],[],kaiser(128,5));
```

uses 256 as the value for `nfft` and 2 as the value for `Fs`.

`Cxy = cohere(x,y,...,'dflag')` specifies a detrend option, where *dflag* is

- `linear`, to remove the best straight-line fit from the prewindowed sections of `x` and `y`
- `mean`, to remove the mean from the prewindowed sections of `x` and `y`
- `none`, for no detrending (default)

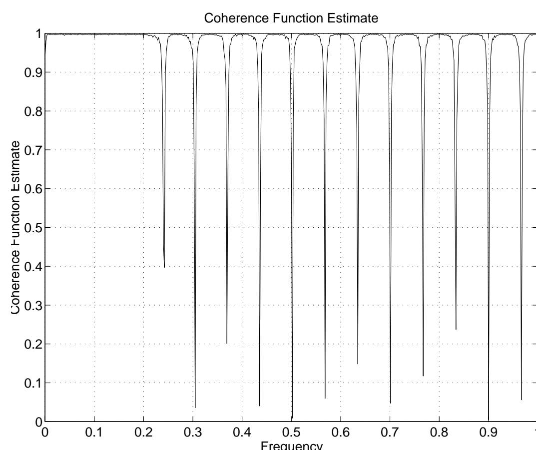
The *dflag* parameter must appear last in the list of input arguments. `cohere` recognizes a *dflag* string no matter how many intermediate arguments are omitted.

`cohere` with no output arguments plots the coherence estimate versus frequency in the current figure window.

Example

Compute and plot the coherence estimate between two colored noise sequences x and y:

```
h = fir1(30,0.2,boxcar(31));
h1 = ones(1,10)/sqrt(10);
r = randn(16384,1);
x = filter(h1,1,r);
y = filter(h,1,x);
cohere(x,y,1024,[],[],512)
```



Diagnostics

An appropriate diagnostic messages is displayed when incorrect arguments are used:

```
Requires window's length to be no greater than the FFT length.
Requires NOVERLAP to be strictly less than the window length.
Requires positive integer values for NFFT and NOVERLAP.
Requires vector (either row or column) input.
Requires inputs X and Y to have the same length.
```

Algorithm

cohere estimates the magnitude squared coherence function [1] using Welch's method of power spectrum estimation (see references [2] and [3]), as follows:

1. It divides the signals x and y into overlapping sections, detrends each section, and multiplies each section by window.
2. It calculates the length nfft fast Fourier transform of each section.
3. It averages the squares of the spectra of the x sections to form Pxx, averages the squares of the spectra of the y sections to form Pyy, and averages the products of the spectra of the x and y sections to form Pxy. It calculates Cxy by

```
Cxy = abs(Pxy).^2/(Pxx.*Pyy)
```

See Also

csd	Estimate the cross spectral density (CSD) of two signals.
psd	Estimate the power spectral density (PSD) of a signal.
tfe	Transfer function estimate from input and output.

References

[1] Kay, S.M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice Hall, 1988. Pg. 454.

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

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

Purpose

Convolution and polynomial multiplication.

Syntax

```
c = conv(a,b)
```

Description

`conv(a,b)` convolves vectors `a` and `b`. The convolution sum is

$$c(n+1) = \sum_{k=0}^{N-1} a(k+1)b(n-k)$$

where N is the maximum sequence length. The series is indexed from $n+1$ and $k+1$ instead of the usual n and k because MATLAB vectors run from 1 to n instead of from 0 to n-1.

Example

The convolution of `a = [1 2 3]` and `b = [4 5 6]` is

```
c = conv(a,b)

c =

     4    13    28    27    18
```

Algorithm

This function is part of the MATLAB environment. It is an M-file that uses the `filter` primitive. `conv` computes the convolution operation as FIR filtering with an appropriate number of zeros appended to the input.

See Also

`conv2`	Two-dimensional convolution.
`convmtx`	Convolution matrix.
`convn`	N-dimensional convolution (see MATLAB Function Reference).
`deconv`	Deconvolution and polynomial division.
`filter`	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.
`residuez`	z-transform partial fraction expansion.
`xcorr`	Cross-correlation function estimate.

conv2

Purpose Two-dimensional convolution.

Syntax

```
C = conv2(A,B)
C = conv2(A,B,'shape')
```

Description

`C = conv2(A,B)` computes the two-dimensional convolution of matrices `A` and `B`. If one of these matrices describes a two-dimensional FIR filter, the other matrix is filtered in two dimensions.

Each dimension of the output matrix `C` is equal in size to the sum of the corresponding dimensions of the input matrices minus 1. For `[ma,na] = size(A)` and `[mb,nb] = size(B)`, then `size(C) = [ma+mb-1,na+nb-1]`.

`C = conv2(A,B,'`*shape*`')` returns a subsection of the two-dimensional convolution with size specified by *shape*, where:

- `full` returns the full two-dimensional convolution (default)
- `same` returns the central part of the convolution that is the same size as `A`
- `valid` returns only those parts of the convolution that are computed without the zero-padded edges. Using this option, `size(C) = [ma-mb+1,na-nb+1]` when `size(A) > size(B)`

`conv2` executes most quickly when `size(A) > size(B)`.

Examples

In image processing, the Sobel edge-finding operation is a two-dimensional convolution of an input array with the special matrix

```
s = [1 2 1; 0 0 0; -1 -2 -1];
```

Given any image, the following line extracts the horizontal edges:

```
h = conv2(I,s);
```

The lines below extract first the vertical edges, then both horizontal and vertical edges combined:

```
v = conv2(I,s');
v2 = (sqrt(h.^2 + v.^2))
```

See Also

conv	Convolution and polynomial multiplication.
convn	*N*-dimensional convolution (see MATLAB Function Reference).
deconv	Deconvolution and polynomial division.
filter2	Two-dimensional digital filtering.
xcorr	Cross-correlation function estimate.
xcorr2	Two-dimensional cross-correlation.

Purpose Convolution matrix.

Syntax

```
A = convmtx(c,n)
A = convmtx(r,n)
```

Description A *convolution matrix* is a matrix, formed from a vector, whose inner product with another vector is the convolution of the two vectors.

`A = convmtx(c,n)` where `c` is a length `m` column vector returns a matrix `A` of size `(m + n-1)`-by-`n`. The product of `A` and another column vector `x` of length `n` is the convolution of `c` with `x`.

`A = convmtx(r,n)` where `r` is a length `m` row vector returns a matrix `A` of size `n`-by-`(m + n-1)`. The product of `A` and another row vector `x` of length `n` is the convolution of `r` with `x`.

Example Generate a simple convolution matrix:

```
h = [1 2 3 2 1];
convmtx(h,7)
ans =

  1   2   3   2   1   0   0   0   0   0   0
  0   1   2   3   2   1   0   0   0   0   0
  0   0   1   2   3   2   1   0   0   0   0
  0   0   0   1   2   3   2   1   0   0   0
  0   0   0   0   1   2   3   2   1   0   0
  0   0   0   0   0   1   2   3   2   1   0
  0   0   0   0   0   0   1   2   3   2   1
```

Note that `convmtx` handles edge conditions by zero padding.

In practice, it is more efficient to compute convolution using

```
y = conv(c,x)
```

than by using a convolution matrix:

```
n = length(x);
y = convmtx(c,n)*x
```

Algorithm `convmtx` uses the function `toeplitz` to generate the convolution matrix.

See Also

`conv`	Convolution and polynomial multiplication.
`convn`	*N*-dimensional convolution (see MATLAB Function Reference).
`conv2`	Two-dimensional convolution.
`dftmtx`	Discrete Fourier transform matrix.

corrcoef

Purpose

Correlation coefficient matrix.

Syntax

```
C = corrcoef(X)
C = corrcoef(X,Y)
```

Description

`corrcoef` returns a matrix of correlation coefficients calculated from an input matrix whose rows are observations and whose columns are variables. If `C = cov(X)`, then `corrcoef(X)` is the matrix whose element (i,j) is

$$\text{corrcoef}(i,j) = \frac{C(i,j)}{\sqrt{C(i,i)C(j,j)}}$$

`C = corrcoef(X)` is the zeroth lag of the covariance function, that is, the zeroth lag of `xcov(x,'coeff')` packed into a square array.

`C = corrcoef(X,Y)` is the same as `corrcoef([X Y])`, that is, it concatenates `X` and `Y` in the row direction before its computation.

`corrcoef` removes the mean from each column before calculating the results. See the `xcorr` function for cross-correlation options.

This function is part of the main MATLAB environment.

See Also

`cov`	Covariance matrix.
`mean`	Average value (see MATLAB Function Reference).
`median`	Median value (see MATLAB Function Reference).
`std`	Standard deviation (see MATLAB Function Reference).
`xcorr`	Cross-correlation function estimate.
`xcov`	Cross-covariance function estimate (equal to mean-removed cross-correlation).

Purpose

Covariance matrix.

Syntax

```
c = cov(x)
c = cov(x,y)
```

Description

cov computes the covariance matrix. If x is a vector, c is a scalar containing the variance. For an array where each row is an observation and each column a variable, cov(X) is the covariance matrix. diag(cov(X)) is a vector of variances for each column, and sqrt(diag(cov(X))) is a vector of standard deviations.

cov(x) is the zeroth lag of the covariance function, that is, the zeroth lag of xcov(x)/(n-1) packed into a square array.

cov(x,y) where x and y are column vectors of equal length is equivalent to cov([x y])), that is, it concatenates x and y in the row direction before its computation.

cov removes the mean from each column before calculating the results.

This function is part of the main MATLAB environment.

Algorithm

```
[n,p] = size(x);
x = x-ones(n,1)*(sum(x)/n);
y = x'*x/(n-1);
```

See Also

corrcoef	Correlation coefficient matrix.
mean	Average value (see MATLAB Function Reference).
median	Median value (see MATLAB Function Reference).
std	Standard deviation (see MATLAB Function Reference).
xcorr	Cross-correlation function estimate.
xcov	Cross-covariance function estimate (equal to mean-removed cross-correlation).

cplxpair

Purpose

Group complex numbers into complex conjugate pairs.

Syntax

```
y = cplxpair(x)
y = cplxpair(x,tol)
```

Description

`y = cplxpair(x)` returns `x` with complex conjugate pairs grouped together. `cplxpair` orders the conjugate pairs by increasing real part. Within a pair, the element with negative imaginary part comes first. The function returns all purely real values following all the complex pairs.

`y = cplxpair(x,tol)` includes a tolerance, `tol`, for determining which numbers are real and which are paired complex conjugates. By default, `cplxpair` uses a tolerance of `100*eps` relative to `abs(x(i))`. `cplxpair` forces the complex conjugate pairs to be exact complex conjugates.

This function is part of the standard MATLAB environment.

Example

Order five poles evenly spaced around the unit circle into complex pairs:

```
cplxpair(exp(2*pi*sqrt(-1)*(0:4)/5)')

ans =
    -0.8090 - 0.5878i
    -0.8090 + 0.5878i
     0.3090 - 0.9511i
     0.3090 + 0.9511i
     1.0000
```

Diagnostics

If there is an odd number of complex numbers, or if the complex numbers cannot be grouped into complex conjugate pairs within the tolerance, `cplxpair` generates the error message:

```
Complex numbers can't be paired.
```

Purpose

Complex and nonlinear-phase equiripple FIR filter design

Syntax

```
b = cremez(n,f,'fresp')
b = cremez(n,f,'fresp',w)
b = cremez(n,f,{'fresp',p1,p2,...},w)
b = cremez(n,f,a,w)
b = cremez(...,'sym')
b = cremez(...,'debug')
b = cremez(...,'skip_stage2')
[b,delta,opt] = cremez(...)
```

Description

`cremez` allows arbitrary frequency-domain constraints to be specified for the design of a possibly complex FIR filter. The Chebyshev (or minimax) filter error is optimized, producing equiripple FIR filter designs.

`b = cremez(n,f,'fresp')` returns a length `n+1` FIR filter with the best approximation to the desired frequency response as returned by function `fresp`. `f` is a vector of frequency bandedge pairs, specified in the range -1 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency). The frequencies must be in increasing order, and `f` must have even length. The frequency bands span `f`*(k)* to `f`*(k+1)* for *k* odd; the intervals `f`*(k+1)* to `f`*(k+2)* for *k* odd are “transition bands” or “don't care” regions during optimization.

`b = cremez(n,f,'fresp',w)` uses the real, non-negative weights in vector `w` to weight the fit in each frequency band. The length of `w` is half the length of `f`, so there is exactly one weight per band.

`b = cremez(n,f,{'fresp',p1,p2,...},...)` supplies optional parameters `p1`, `p2`, ..., to the frequency response function `fresp`. Predefined frequency response functions for *'fresp'* include the following design filters with the appropriate frequency response. For all of the predefined frequency response functions, the symmetry option *'sym'* defaults to `'even'` if no negative frequencies are contained in `f` and `d = 0`; otherwise *'sym'* defaults to `'none'`. (See the *'sym'* option below for details.) For all of the predefined frequency response functions, `d` specifies a group-delay offset such that the filter response

will have a group delay of `n/2+d` in units of the sample interval. Negative values create less delay; positive values create more delay. By default, `d = 0`.

- `lowpass`, `highpass`, `bandpass`, `bandstop`

  These functions share a common syntax, exemplified here by `'lowpass'`:

  `b = cremez(n,f,'lowpass',...)` and

  `b = cremez(n,f,{'lowpass',d},...)` design a linear-phase (`n/2+d` delay) filter.

- `multiband` designs a linear-phase frequency response filter with arbitrary band amplitudes.

  `b = cremez(n,f,{'multiband',a},...)` and

  `b = cremez(n,f,{'multiband',a,d},...)` specify vector `a` containing the desired amplitudes at the band edges in `f`. The desired amplitude at frequencies between pairs of points *(*`f`*(k),* `f`*(k+1))* for *k* odd is the line segment connecting the points *(*`f`*(k),* `a`*(k))* and *(*`f`*(k+1),* `a`*(k+1))*.

- `differentiator` designs a linear-phase differentiator. For these designs, zero-frequency must be in a transition band, and band weighting is set to be inversely proportional to frequency.

  `b = cremez(n,f,{'differentiator',Fs},...)` and

  `b = cremez(n,f,{'differentiator',Fs,d},...)` specify the sample rate `Fs` used to determine the slope of the differentiator response. If omitted, `Fs` defaults to 1.

- `hilbfilt` designs a linear-phase Hilbert transform filter response. For Hilbert designs, zero-frequency must be in a transition band.

  `b = cremez(n,f,'hilbfilt',...)` and

  `b = cremez(N,F,{'hilbfilt',d},...)` design a linear-phase (`n/2+d` delay) Hilbert transform filter.

`b = cremez(n,f,a,w)` is a synonym for `b = cremez(n,f,{'multiband',a},w)`.

`b = cremez(..., 'sym')` imposes a symmetry constraint on the impulse response of the design, where *'sym'* may be

- `none` indicates no symmetry constraint

  This is the default if any negative band edge frequencies are passed, or if *'fresp'* does not supply a default.
- `even` indicates a real and even impulse response

  This is the default for highpass, lowpass, bandpass, bandstop, and multi-band designs.
- `odd` indicates a real and odd impulse response

  This is the default for Hilbert and differentiator designs.
- `real` indicates conjugate symmetry for the frequency response

If any *'sym'* option other than `'none'` is specified, the band edges should only be specified over positive frequencies; the negative frequency region will be filled in from symmetry. If not specified, the *'fresp'* function will be queried for a default setting.

`b = cremez(..., 'skip_stage2')` disables the second-stage optimization algorithm, which executes only when `cremez` determines that an optimal solution has not been reached by the standard Remez error-exchange. Disabling this algorithm may increase the speed of computation, but may incur a reduction in accuracy. By default, the second-stage optimization is enabled.

`b = cremez(..., 'debug')` enables the display of intermediate results during the filter design, where *'debug'* may be one of `'trace'`, `'plots'`, `'both'`, or `'off'`. By default, it is set to `'off'`.

Note that any combination of the *'sym'*, *'debug'*, and `'skip_stage2'` options may be specified.

`[b,delta] = cremez(...)` returns the maximum ripple height `delta`.

`[b,delta,opt] = cremez(...)` returns a structure `opt` of optional results computed by `cremez` and contains the following fields:

`opt.fgrid`, frequency grid vector used for the filter design optimization

`opt.H`, frequency response for each point in `opt.fgrid`

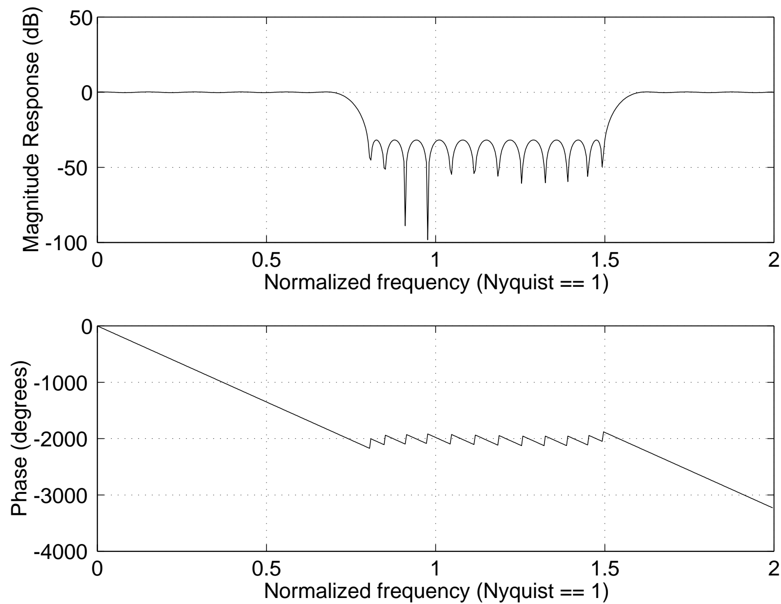
`opt.error`, error at each point in `opt.fgrid`

`opt.fextr`, vector of extremal frequencies

Example

Design a 31-tap, linear-phase, lowpass filter:

```
b = cremez(30,[-1 -0.5 -0.4 0.7 0.8 1],'lowpass');
freqz(b,1,512,'whole');
```



Remarks

User-definable functions may be used, instead of the predefined frequency response functions for `'fresp'`. The function is called from within `cremez` using the following syntax:

```
[dh,dw] = fresp(n,f,gf,w,p1,p2,...)
```
where

- `n` is the filter order.
- `f` is the vector of frequency band edges that appear monotonically between -1 and 1, where 1 is the Nyquist frequency.
- `gf` is a vector of grid points that have been linearly interpolated over each specified frequency band by `cremez`. `gf` determines the frequency grid at which the response function must be evaluated.
- `w` is a vector of real, positive weights, one per band, used during optimization. `w` is optional in the call to `cremez`; if not specified, it is set to unity weighting before being passed to `'fresp'`.
- `dh` and `dw` are the desired complex frequency response and band weight vectors, respectively, evaluated at each frequency in grid `gf`.
- `p1`, `p2`,..., are optional parameters that may be passed to `'fresp'`.

Additionally, a preliminary call is made to `'fresp'` to determine the default symmetry property `'sym'`. This call is made using the syntax:

```
sym = fresp('defaults',{n,f,[],w,p1,p2,...})
```

The arguments may be used in determining an appropriate symmetry default as necessary. The function `private/lowpass.m` may be useful as a template for generating new frequency response functions.

Algorithm

An extended version of the Remez exchange method is implemented for the complex case. This exchange method obtains the optimal filter when the equiripple nature of the filter is restricted to have `n+2` extremals. When it does not converge, the algorithm switches to an ascent-descent algorithm that takes over to finish the convergence to the optimal solution. See the references for further details.

Diagnostics

The following diagnostic messages arise from incorrect usage of `cremez`:

```
Not enough input arguments.
F must contain an even number of band edge entries.
Band edges must be monotonically increasing.
Expecting a string argument.
Invalid argument arg specified.
Invalid default symmetry option sym returned from response.
function fresp. Must be one of 'none', 'real', 'even', or 'odd'.
Frequency band edges must be in the range [-1,+1] for designs with
SYM = 'sym'.
Frequency band edges must be in the range [0,+1] for designs with
SYM = 'sym'.
Incorrect size of results from response function fresp. Sizes must
be the same size as the frequency grid GF.
Both -1 and 1 have been specified as frequencies in F, and the
frequency spacing is too close to move either of them toward its
neighbor.
Internal error: Grid frequencies out of range.
Internal error: domain must be "whole" or "half".
Internal error: obtained a negative bandwidth.
Internal error: two extremal frequecies at the same grid point.
Internal error: dBrange must be > 0.
```

See Also

`fir1`	Window-based finite impulse response filter design—standard response.
`fir2`	Window-based finite impulse response filter design—arbitrary response.
`firls`	Least square linear-phase FIR filter design.
`remez`	Parks-McClellan optimal FIR filter design.
`private/bandpass`	Bandpass filter design function.
`private/bandstop`	Bandstop filter design function.
`private/differentiator`	Differentiator filter design function.
`private/highpass`	Highpass filter design function.
`private/hilbfilt`	Hilbert filter design function.
`private/lowpass`	Lowpass filter design function.
`private/multiband`	Multiband filter design function.

References

[1] Karam, L.J., and J.H. McClellan. "Complex Chebyshev Approximation for FIR Filter Design." *IEEE Trans. on Circuits and Systems II*. March 1995. Pgs. 207-216.

[2] Karam, L.J. *Design of Complex Digital FIR Filters in the Chebyshev Sense*. Ph.D. Thesis, Georgia Institute of Technology, March 1995.

[3] Demjanjov, V.F., and V.N. Malozemov. *Introduction to Minimax*. New York: John Wiley & Sons, 1974.

Purpose

Estimate the cross spectral density (CSD) of two signals.

Syntax

```
Pxy = csd(x,y)
Pxy = csd(x,y,nfft)
[Pxy,f] = csd(x,y,nfft,Fs)
Pxy = csd(x,y,nfft,Fs,window)
Pxy = csd(x,y,nfft,Fs,window,noverlap)
Pxy = csd(x,y,...,'dflag')
[Pxy,Pxyc,f] = csd(x,y,nfft,Fs,window,noverlap,p)
csd(x,y,...)
```

Description

`Pxy = csd(x,y)` estimates the cross spectral density of the length `n` sequences `x` and `y` using the Welch method of spectral estimation. `Pxy = csd(x,y)` uses the following default values:

- `nfft = min(256,length(x))`
- `Fs = 2`
- `window = hanning(nfft)`
- `noverlap = 0`

`nfft` specifies the FFT length that `csd` uses. This value determines the frequencies at which the cross spectrum is estimated. `Fs` is a scalar that specifies the sampling frequency. `window` specifies a windowing function and the number of samples `csd` uses in its sectioning of the `x` and `y` vectors. `noverlap` is the number of samples by which the sections overlap. Any arguments omitted from the end of the parameter list use the default values shown above.

If `x` and `y` are real, `csd` estimates the cross spectral density at positive frequencies only; in this case, the output `Pxy` is a column vector of length `nfft/2 + 1` for `nfft` even and `(nfft + 1)/2` for `nfft` odd. If `x` or `y` is complex, `csd` estimates the cross spectral density at both positive and negative frequencies and `Pxy` has length `nfft`.

`Pxy = csd(x,y,nfft)` uses the FFT length `nfft` in estimating the cross spectral density of `x` and `y`. Specify `nfft` as a power of 2 for fastest execution.

`[Pxy,f] = csd(x,y,nfft,Fs)` returns a vector `f` of frequencies at which the function evaluates the CSD. `f` is the same size as `Pxy`, so `plot(f,Pxy)` plots the

spectrum versus properly scaled frequency. Fs has no effect on the output Pxy; it is a frequency scaling multiplier.

Pxy = csd(x, y, nfft, Fs, window) specifies a windowing function and the number of samples per section of the x vector. If you supply a scalar for window, csd uses a Hanning window of that length. The length of the window must be less than or equal to nfft; csd zero pads the sections if the length of the window is less than nfft. csd returns an error if the length of the window is greater than nfft.

Pxy = csd(x, y, nfft, Fs, window, noverlap) overlaps the sections of x and y by noverlap samples.

You can use the empty matrix [] to specify the default value for any input argument except x or y. For example,

```
csd(x, y, [], 10000)
```

is equivalent to

```
csd(x)
```

but with a sampling frequency of 10,000 Hz instead of the default of 2 Hz.

Pxy = csd(x, y, ..., '*dflag*') specifies a detrend option, where *dflag* is

- linear, to remove the best straight-line fit from the prewindowed sections of x and y
- mean, to remove the mean from the prewindowed sections of x and y
- none, for no detrending (default)

The *dflag* parameter must appear last in the list of input arguments. csd recognizes a *dflag* string no matter how many intermediate arguments are omitted.

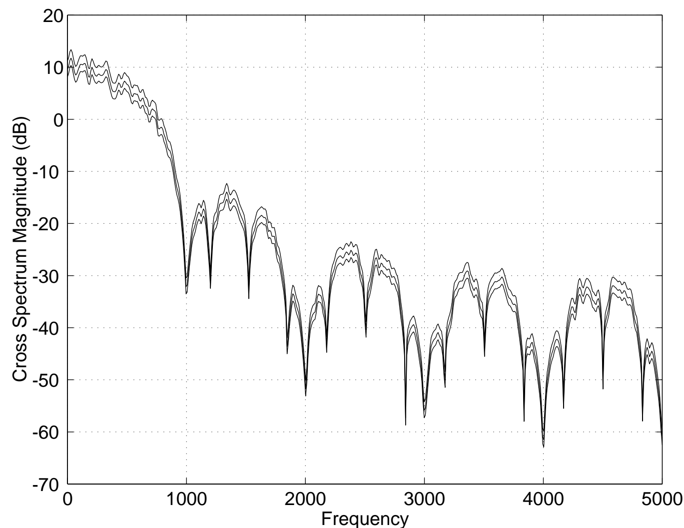
[Pxy, Pxyc, f] = csd(x, y, nfft, Fs, window, noverlap, p) where p is a positive scalar between 0 and 1 returns a vector Pxyc that contains an estimate of the p*100 percent confidence interval for Pxy. Pxyc is a two-column matrix the same length as Pxy. The interval [Pxyc(:, 1), Pxyc(:, 2)] covers the true CSD with probability p. plot(f, [Pxy Pxyc]) plots the cross spectrum inside the p*100 percent confidence interval. If unspecified, p defaults to 0.95.

csd(x,y,...) plots the CSD versus frequency in the current figure window. If the p parameter is specified, the plot includes the confidence interval.

Example

Generate two colored noise signals and plot their CSD with a confidence interval of 95%. Specify a length 1024 FFT, a 500 point triangular window with no overlap, and a sampling frequency of 10 Hz:

```
h = fir1(30,0.2,boxcar(31));
h1 = ones(1,10)/sqrt(10);
r = randn(16384,1);
x = filter(h1,1,r);
y = filter(h,1,x);
csd(x,y,1024,10000,triang(500),0,[])
```



Algorithm

csd implements the Welch method of spectral density estimation (see references [1] and [2]):

1. It applies the window specified by the window vector to each successive detrended section.
2. It transforms each section with an nfft-point FFT.
3. It forms the periodogram of each section by scaling the product of the transform of the x section and the conjugate of the transformed y section.
4. It averages the periodograms of the successive overlapping sections to form Pxy, the cross spectral density of x and y.

The number of sections that csd averages is k, where k is

```
fix((length(x)-noverlap)/(length(window)-noverlap))
```

Diagnostics

An appropriate diagnostic message is displayed when incorrect arguments to csd are used:

```
Requires window's length to be no greater than the FFT length.
Requires NOVERLAP to be strictly less than the window length.
Requires positive integer values for NFFT and NOVERLAP.
Requires vector (either row or column) input.
Requires inputs X and Y to have the same length.
Requires confidence parameter to be a scalar between 0 and 1.
```

See Also

cohere	Estimate magnitude squared coherence function between two signals.
pmem	Power spectrum estimate using maximum entropy method (MEM).
pmtm	Power spectrum estimate using the multitaper method (MTM).
pmusic	Power spectrum estimate using MUSIC eigenvector method.
psd	Estimate the power spectral density (PSD) of a signal.
tfe	Transfer function estimate from input and output.

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975. Pgs. 414-419.

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

[3] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. Pg. 737.

Purpose Chirp *z*-transform.

Syntax

```
y = czt(x,m,w,a)
y = czt(x)
```

Description

`y = czt(x,m,w,a)` returns the chirp *z*-transform of signal `x`. The chirp *z*-transform is the *z*-transform of `x` along a spiral contour defined by `w` and `a`. `m` is a scalar that specifies the length of the transform, `w` is the ratio between points along the *z*-plane spiral contour of interest, and scalar `a` is the complex starting point on that contour. The contour, a spiral or "chirp" in the *z*-plane, is given by

```
z = a*(w.^-(0:m-1))
```

`y = czt(x)` uses the following default values:

- `m = length(x)`
- `w = exp(j*2*pi/m)`
- `a = 1`

With these defaults, `czt` returns the *z*-transform of `x` at `m` equally spaced points around the unit circle. This is equivalent to the discrete Fourier transform of `x`, or `fft(x)`. The empty matrix `[]` specifies the default value for a parameter.

If `x` is a matrix, `czt(x,m,w,a)` transforms the columns of `x`.

Examples

Create a random vector `x` of length 1013 and compute its DFT using `czt`. This is faster than the `fft` function on the same sequence.

```
x = randn(1013,1);
y = czt(x);
```

Use `czt` to zoom in on a narrow-band section (100 to 150 Hz) of a filter's frequency response. First design the filter:

```
h = fir1(30,125/500,boxcar(31));  % filter
```

Establish frequency and CZT parameters:

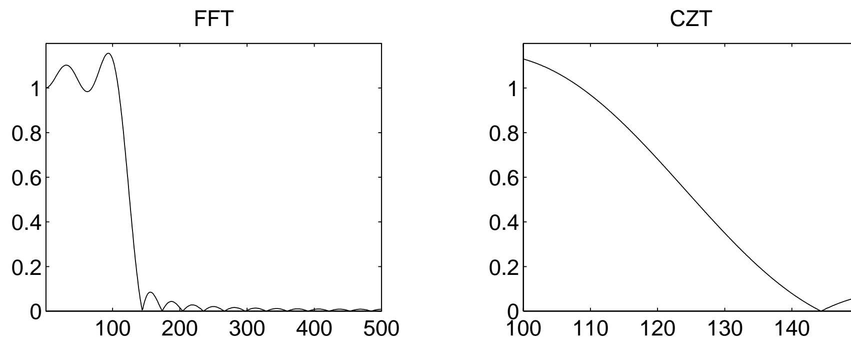
```
Fs = 1000;  f1 = 100;  f2 = 150;   % in Hertz
m = 1024;
w = exp(-j*2*pi*(f2-f1)/(m*Fs));
a = exp(j*2*pi*f1/Fs);
```

Compute both the DFT and CZT of the filter:

```
y = fft(h,1000);
z = czt(h,m,w,a);
```

Create frequency vectors and compare the results:

```
fy = (0:length(y)-1)'*1000/length(y);
fz = ((0:length(z)-1)'*(f2-f1)/length(z)) + f1;
plot(fy(1:500),abs(y(1:500))); axis([1 500 0 1.2])
plot(fz,abs(z)); axis([f1 f2 0 1.2])
```



Algorithm

czt uses the next power-of-2 length FFT to perform a fast convolution when computing the *z*-transform on a specified chirp contour [1]. czt can be significantly faster than fft for large, prime-length sequences.

Diagnostics

If m, w, or a is not a scalar, czt gives the following error message:

```
Inputs M, W, and A must be scalars.
```

See Also

fft	One-dimensional fast Fourier transform.
freqz	Frequency response of digital filters.

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975. Pgs. 393-399.

Purpose Discrete cosine transform (DCT).

Syntax

```
y = dct(x)
y = dct(x, n)
```

Description

`y = dct(x)` returns the discrete cosine transform of `x`

$$y(k+1) = \sum_{n=0}^{N-1} 2x(n+1)\cos\left[\frac{\pi}{2N}(k+1)(2n+1)\right], \qquad k = 0,\ldots,N-1$$

where `N` is the length of `x`, and `x` and `y` are the same size. If `x` is a matrix, `dct` transforms its columns. The series is indexed from $n + 1$ and $k + 1$ instead of the usual n and k because MATLAB vectors run from 1 to N instead of from 0 to N- 1.

`y = dct(x, n)` pads or truncates `x` to length `n` before transforming.

If `x` is a matrix, `dct` transforms its columns.

The DCT is closely related to the discrete Fourier transform. You can often reconstruct a sequence very accurately from only a few DCT coefficients, a useful property for applications requiring data reduction.

Example Find how many DCT coefficients represent 99% of the energy in a sequence:

```
x = (1:100) + 50*cos((1:100)*2*pi/40);
X = dct(x);
[XX,ind] = sort(abs(X)); ind = fliplr(ind);
i = 1;
while (norm([X(ind(1:i)) zeros(1,100-i)])/norm(X)<.99)
 i = i + 1;
end

i =
      3
```

See Also

`fft`	One-dimensional fast Fourier transform.
`idct`	Inverse discrete cosine transform.
`dct2`	Two-dimensional DCT (see *Image Processing Toolbox User's Manual*).
`idct2`	Two-dimensional inverse DCT (see *Image Processing Toolbox User's Manual*).

Purpose

Decrease the sampling rate for a sequence (decimation).

Syntax

```
y = decimate(x,r)
y = decimate(x,r,n)
y = decimate(x,r,'fir')
y = decimate(x,r,n,'fir')
```

Description

Decimation reduces the original sampling rate for a sequence to a lower rate. It is the opposite of interpolation. The decimation process filters the input data with a lowpass filter and then resamples the resulting smoothed signal at a lower rate.

`y = decimate(x,r)` reduces the sample rate of `x` by a factor `r`. The decimated vector `y` is `r` times shorter in length than the input vector `x`. By default, `decimate` employs an eighth-order lowpass Chebyshev type I filter. It filters the input sequence in both the forward and reverse directions to remove all phase distortion, effectively doubling the filter order.

`y = decimate(x,r,n)` uses an order `n` Chebyshev filter. Orders above 13 are not recommended because of numerical instability. MATLAB displays a warning in this case.

`y = decimate(x,r,'fir')` uses a 30-point FIR filter, instead of the Chebyshev IIR filter. Here `decimate` filters the input sequence in only one direction. This technique conserves memory and is useful for working with long sequences.

`y = decimate(x,r,n,'fir')` uses a length `n` FIR filter.

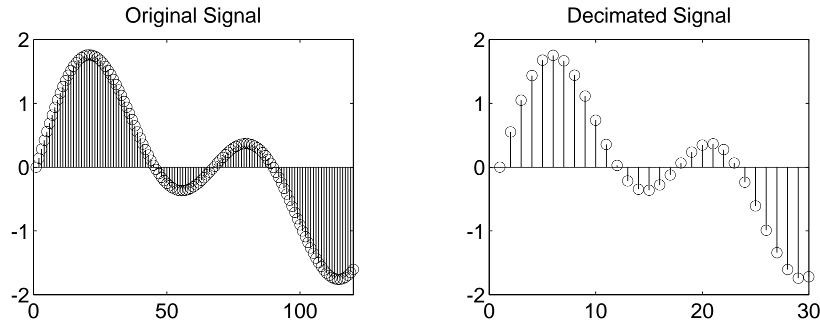
Example

Decimate a signal by a factor of four:

```
t = 0:.00025:1;                          % time vector
x = sin(2*pi*30*t) + sin(2*pi*60*t);
y = decimate(x,4);
```

View the original and decimated signals:

```
stem(x(1:120)), axis([0 120 -2 2])   % original signal
stem(y(1:30))                        % decimated signal
```



Algorithm

`decimate` uses decimation algorithms 8.2 and 8.3 from [1]:

1. It designs a lowpass filter. By default, `decimate` uses a Chebyshev type I filter with normalized cutoff frequency 0.8/`r` and 0.05 dB of passband ripple. For the `fir` option, `decimate` designs a lowpass FIR filter with cutoff frequency `1/r` using `fir1`.
2. For the FIR filter, `decimate` applies the filter to the input vector in one direction. In the IIR case, `decimate` applies the filter in forward and reverse directions with `filtfilt`.
3. `decimate` resamples the filtered data by selecting every `r`–th point.

Diagnostics

If `r` is not an integer, `decimate` gives the following error message:

```
Resampling rate R must be an integer.
```

If `n` specifies an IIR filter with order greater than 13, `decimate` gives the following warning:

```
Warning: IIR filters above order 13 may be unreliable.
```

See Also

`interp`	Increase sampling rate by an integer factor (interpolation).
`resample`	Change sampling rate by any factor.
`spline`	Cubic spline interpolation (see MATLAB Function Reference).
`upfirdn`	Apply FIR filter and perform sample rate conversion.

References

[1] IEEE. *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979. Chapter 8.

deconv

Purpose

Deconvolution and polynomial division.

Syntax

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

Description

`[q,r] = deconv(b,a)` deconvolves vector `a` out of vector `b`, using long division. The result (quotient) is returned in vector `q` and the remainder in vector `r` such that `b = conv(q,a) + r`.

If `a` and `b` are vectors of polynomial coefficients, convolving them is equivalent to polynomial multiplication, and deconvolution is equivalent to polynomial division. The result of dividing `b` by `a` is quotient `q` and remainder `r`.

`deconv` is part of the main MATLAB environment.

Example

The convolution of `a = [1 2 3]` and `b = [4 5 6]` is

```
c = conv(a,b)

c =

     4    13    28    27    18
```

Use `deconv` to divide `b` back out:

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

q =

     4     5     6
r =

     0     0     0     0     0
```

Algorithm

This function is an M-file in the MATLAB environment that uses the `filter` primitive. Deconvolution is the impulse response of an IIR filter.

See Also

`conv`	Convolution and polynomial multiplication.
`filter`	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.
`residuez`	*z*-transform partial fraction expansion.

Purpose

Demodulation for communications simulation.

Syntax

```
x = demod(y,Fc,Fs,'method')
x = demod(y,Fc,Fs,'method',opt)
x = demod(y,Fc,Fs,'pwm','centered')
[x1,x2] = demod(y,Fc,Fs,'qam')
```

Description

demod performs demodulation, that is, it obtains the original signal from a modulated version of the signal. demod undoes the operation performed by modulate.

x = demod(y,Fc,Fs,'*method*',opt) demodulates the real carrier signal y with a carrier frequency Fc and sampling frequency Fs, using one of the options listed below for *method*. (Note that some methods accept an option, opt.)

amdsb-sc or am — **Amplitude demodulation, double side-band, suppressed carrier.** Multiplies y by a sinusoid of frequency Fc and applies a fifth-order Butterworth lowpass filter using filtfilt:

```
x = y.*cos(2*pi*Fc*t);
[b,a] = butter(5,Fc*2/Fs);
x = filtfilt(b,a,x);
```

amdsb-tc — **Amplitude demodulation, double side-band, transmitted carrier.** Multiplies y by a sinusoid of frequency Fc, and applies a fifth-order Butterworth lowpass filter using filtfilt:

```
x = y.*cos(2*pi*Fc*t);
[b,a] = butter(5,Fc*2/Fs);
x = filtfilt(b,a,x);
```

If you specify opt, demod subtracts scalar opt from x. The default value for opt is 0.

amssb — **Amplitude demodulation, single side-band.** Multiplies y by a sinusoid of frequency Fc and applies a fifth-order Butterworth lowpass filter using filtfilt:

```
x = y.*cos(2*pi*Fc*t);
[b,a] = butter(5,Fc*2/Fs);
x = filtfilt(b,a,x);
```

`fm`	**Frequency demodulation.** Demodulates the FM waveform by modulating the Hilbert transform of `y` by a complex exponential of frequency –`Fc` Hz and obtains the instantaneous frequency of the result.
`pm`	**Phase demodulation.** Demodulates the PM waveform by modulating the Hilbert transform of `y` by a complex exponential of frequency –`Fc` Hz and obtains the instantaneous phase of the result.
`ptm`	**Pulse-time demodulation.** Finds the pulse times of a pulse-time modulated signal `y`. For correct demodulation, the pulses cannot overlap. `x` is length `length(t)*Fc/Fs`.
`pwm`	**Pulse-width demodulation.** Finds the pulse widths of a pulse-width modulated signal `y`. `demod` returns in `x` a vector whose elements specify the width of each pulse in fractions of a period. The pulses in `y` should start at the beginning of each carrier period, that is, they should be left justified.
`qam`	**Quadrature amplitude demodulation.** `[x1,x2] = demod(y,Fc,Fs,'qam')` multiplies `y` by a cosine and a sine of frequency `Fc` and applies a fifth-order Butterworth lowpass filter using `filtfilt`:

```
x1 = y.*cos(2*pi*Fc*t);
x2 = y.*sin(2*pi*Fc*t);
[b,a] = butter(5,Fc*2/Fs);
x1 = filtfilt(b,a,x1);
x2 = filtfilt(b,a,x2);
```

The default method is `'am'`. Except for the `'ptm'` and `'pwm'` cases, `x` is the same size as `y`.

If `y` is a matrix, `demod` demodulates its columns.

`x = demod(y,Fc,Fs,'pwm','centered')` finds the pulse widths assuming they are centered at the beginning of each period. `x` is length `length(y)*Fc/Fs`.

See Also

`modulate`	Modulation for communications simulation.
`vco`	Voltage controlled oscillator.

detrend

Purpose Remove linear trends.

Syntax

```
y = detrend(x)
y = detrend(x,0)
```

Description

detrend removes the mean value or linear trend from a vector or matrix, usually for FFT processing.

y = detrend(x) removes the best straight-line fit from vector x and returns it in y. If x is an array, detrend removes the trend from each column.

y = detrend(x,0) removes the mean value from vector x or, if x is an array, from each column of the array.

Algorithm

detrend computes the least-squares fit of a straight line to the data and subtracts the resulting function from the data. The main part of the algorithm is

```
m = length(x);
a = [(1:m)'/m ones(m,1)];
y = x - a*(a\x);
```

To obtain the equation of the straight-line fit, use polyfit.

See Also

polyfit	Polynomial curve fitting (see MATLAB Function Reference).

dftmtx

Purpose

Discrete Fourier transform matrix.

Syntax

```
A = dftmtx(n)
```

Description

A *discrete Fourier transform matrix* is a complex matrix of values around the unit circle, whose matrix product with a vector computes the discrete Fourier transform of the vector.

`A = dftmtx(n)` returns the `n`-by-`n` complex matrix `A` that, when multiplied into a length `n` column vector `x`:

```
y = A*x
```

computes the discrete Fourier transform of `x`.

The inverse discrete Fourier transform matrix is

```
Ai = conj(dftmtx(n))/n
```

Example

In practice, the discrete Fourier transform is computed more efficiently and uses less memory with an FFT algorithm

```
x = 1:256;
y1 = fft(x);
```

than by using the Fourier transform matrix

```
n = length(x);
y2 = x*dftmtx(n);
norm(y1-y2)

ans =

      2.0016e-09
```

Algorithm

`dftmtx` uses an outer product to generate the transform matrix.

See Also

`convmtx`	Convolution matrix.
`fft`	One-dimensional fast Fourier transform.

Purpose

Dirichlet or periodic sinc function.

Syntax

```
y = diric(x,n)
```

Description

`y = diric(x,n)` returns a vector or array `y` the same size as `x`. The elements of `y` are the Dirichlet function of the elements of `x`. `n` must be a positive integer.

The Dirichlet function, or periodic sinc function, is

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

for any nonzero integer `n`. This function has period 2π for `n` odd and period 4π for `n` even. Its peak value is 1, and its minimum value is -1 for `n` even. The magnitude of this function is (1/n) times the magnitude of the discrete-time Fourier transform of the `n`-point rectangular window.

Diagnostics

If `n` is not a positive integer, `diric` gives the following error message:

```
Requires n to be a positive integer.
```

See Also

`cos`	Cosine of vector/matrix elements (see MATLAB Function Reference).
`gauspuls`	Gaussian-modulated sinusoidal pulse generator.
`pulstran`	Pulse train generator.
`rectpuls`	Sampled aperiodic rectangle generator.
`sawtooth`	Sawtooth or triangle wave generator.
`sin`	Sine of vector/matrix elements (see MATLAB Function Reference).
`sinc`	Sinc or sin(πt)/πt function.
`square`	Square wave generator.
`tripuls`	Sampled aperiodic triangle generator.

dpss

Purpose

Discrete prolate spheroidal sequences (Slepian sequences).

Syntax

```
[e,v] = dpss(n,nw)
[e,v] = dpss(n,nw,'calc')
[e,v] = dpss(n,nw,'spline')
[e,v] = dpss(n,nw,'spline',Ni)
[e,v] = dpss(n,nw,'linear')
[e,v] = dpss(n,nw,'linear',Ni)
[e,v] = dpss(n,nw,'trace')
[e,v] = dpss(n,nw,'int','trace')
```

Description

`[e,v] = dpss(n,nw)` generates the first `2*nw` *discrete prolate spheroidal sequences* (DPSS) of length `n`, in the columns of `e`, and their corresponding concentrations in vector `v`. They are generated in the DPSS MAT-file database `dpss.mat`.

- The sequences are generated in the frequency band $|\omega| \le (2\pi W)$, where W = `nw/n` is the half-bandwidth and ω is in radians.
- `e(:,1)` is the length `n` signal most concentrated in the frequency band $|\omega| \le (2\pi W)$ radians, `e(:,2)` is the signal orthogonal to `e(:,1)` that is most concentrated in this band, `e(:,3)` is the signal orthogonal to both `e(:,1)` and `e(:,2)` that is most concentrated in this band, etc.
- For multitaper spectral analysis, typical choices for `nw` are 2, 5/2, 3, 7/2, or 4.

If you specify only `n` and `nw`, DPSS follows these rules for selecting an efficient (although sometimes approximate) algorithm:

1. If `n < 500`, the sequences are calculated directly.
2. If `n ≥ 500`,
   a. look for sequences in the DPSS MAT-file database `dpss.mat`;
   b. if not found, linearly interpolate from sequences with the same `nw` and next largest `n` in the database;
   c. if not found, use spline interpolation from length 64 sequences.

`[e,v] = dpss(n,nw,'calc')` forces DPSS to calculate `e` and `v` directly. This can take a long time for large `n` and `nw`.

`[e,v] = dpss(n,nw,'spline')` uses spline interpolation to compute `e` and `v` from the sequences in `dpss.mat` with length closest to `n`.

`[e,v] = dpss(n,nw,'spline',Ni)` interpolates from existing length `Ni` sequences.

`[e,v] = dpss(n,nw,'linear')` and

`[e,v] = dpss(n,nw,'linear',Ni)` use linear interpolation, which is much faster but less accurate than spline interpolation. `'linear'` requires `Ni > n`.

`[e,v] = dpss(n,nw,'trace')` and

`[e,v] = dpss(n,nw,'int','trace')` use a trailing `'trace'` argument to find out which method DPSS uses, where `'int'` is either `'spline'` or `'linear'`.

See Also

`dpssclear`	Remove discrete prolate spheroidal sequences from database.
`dpssdir`	Discrete prolate spheroidal sequences database directory.
`dpssload`	Load discrete prolate spheroidal sequences from database.
`dpsssave`	Save discrete prolate spheroidal sequences in database.
`pmtm`	Power spectrum estimate using the multitaper method (MTM).

References

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

dpssclear

Purpose Remove discrete prolate spheroidal sequences from database.

Syntax

```
dpssclear(n,nw)
```

Description `dpssclear(n,nw)` removes sequences with length `n` and time-bandwidth product `nw` from the DPSS MAT-file database `dpss.mat`.

See Also

`dpss`	Discrete prolate spheroidal sequences (Slepian sequences).
`dpssdir`	Discrete prolate spheroidal sequences database directory.
`dpssload`	Load discrete prolate spheroidal sequences from database.
`dpsssave`	Save discrete prolate spheroidal sequences in database.

Purpose

Discrete prolate spheroidal sequences database directory.

Syntax

```
dpssdir
dpssdir(n)
dpssdir(nw,'nw')
dpssdir(n,nw)
index = dpssdir
```

Description

`dpssdir` manages the database directory that contains the generated DPSS samples in the DPSS MAT-file database `dpss.mat`.

`dpssdir` lists the directory of saved sequences in `dpss.mat`.

`dpssdir(n)` lists the sequences saved with length `n`.

`dpssdir(nw,'nw')` lists the sequences saved with time-bandwidth product `nw`.

`dpssdir(n,nw)` lists the sequences saved with length `n` and time-bandwidth product `nw`.

`index = dpssdir` is a structure array describing the DPSS database. Pass `n` and `nw` options as for the no output case to get a filtered `index`.

See Also

`dpss`	Discrete prolate spheroidal sequences (Slepian sequences).
`dpssclear`	Remove discrete prolate spheroidal sequences from database.
`dpssload`	Load discrete prolate spheroidal sequences from database.
`dpsssave`	Save discrete prolate spheroidal sequences in database.

dpssload

Purpose Load discrete prolate spheroidal sequences from database.

Syntax `[e,v] = dpssload(n,nw)`

Description `[e,v] = dpssload(n,nw)` loads all sequences with length `n` and time-bandwidth product `nw` in the columns of `e` and their corresponding concentrations in vector `v` from the DPSS MAT-file database `dpss.mat`.

See Also

`dpss`	Discrete prolate spheroidal sequences (Slepian sequences).
`dpssclear`	Remove discrete prolate spheroidal sequences from database.
`dpssdir`	Discrete prolate spheroidal sequences database directory.
`dpsssave`	Save discrete prolate spheroidal sequences in database.

Purpose

Save discrete prolate spheroidal sequences in database.

Syntax

```
dpsssave(nw,e,v)
status = dpsssave(nw,e,v)
```

Description

`dpsssave(nw,e,v)` saves the sequences in the columns of `e` and their corresponding concentrations in vector `v` in the DPSS MAT-file database `dpss.mat`.

- It is not necessary to specify sequence length, because the length of the sequence is determined by the number of rows of `e`.
- `nw` is the *time-bandwidth product* that was specified when the sequence was created using `dpss`.

`status = dpsssave(nw,e,v)` returns 0 if the save was successful and 1 if there was an error.

See Also

`dpss`	Discrete prolate spheroidal sequences (Slepian sequences).
`dpssclear`	Remove discrete prolate spheroidal sequences from database.
`dpssdir`	Discrete prolate spheroidal sequences database directory.
`dpssload`	Load discrete prolate spheroidal sequences from database.

ellip

Purpose

Elliptic (Cauer) filter design.

Syntax

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

Description

`ellip` designs lowpass, bandpass, highpass, and bandstop digital and analog elliptic filters. Elliptic filters offer steeper rolloff characteristics than Butterworth or Chebyshev filters, but are equiripple in both the pass- and stopbands. In general, elliptic filters meet given performance specifications with the lowest order of any filter type.

Digital Domain

`[b,a] = ellip(n,Rp,Rs,Wn)` designs an order `n` lowpass digital elliptic filter with cutoff frequency `Wn`, `Rp` dB of ripple in the passband, and a stopband `Rs` dB down from the peak value in the passband. It returns the filter coefficients in the length `n + 1` row vectors `b` and `a`, with coefficients in descending powers of z.

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

The *cutoff frequency* is the edge of the passband, at which the magnitude response of the filter is –`Rp` dB. For `ellip`, the cutoff frequency `Wn` is a number between 0 and 1, where 1 corresponds to half the sample frequency (Nyquist frequency). Smaller values of passband ripple `Rp` and larger values of stopband attenuation `Rs` both lead to wider transition widths (shallower rolloff characteristics).

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

`[b,a] = ellip(n,Rp,Rs,Wn,'ftype')` designs a highpass or bandstop filter, where *ftype* is

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

  The stopband is `w1` < ω < `w2`.

With different numbers of output arguments, `ellip` directly obtains other realizations of the filter. To obtain zero-pole-gain form, use three output arguments:

`[z,p,k] = ellip(n,Rp,Rs,Wn)` or

`[z,p,k] = ellip(n,Rp,Rs,Wn,'ftype')` returns the zeros and poles in length `n` column vectors `z` and `p` and the gain in the scalar `k`.

To obtain state-space form, use four output arguments:

`[A,B,C,D] = ellip(n,Rp,Rs,Wn)` or

`[A,B,C,D] = ellip(n,Rp,Rs,Wn,'ftype')` where `A`, `B`, `C`, and `D` are

$$\begin{aligned} x[n+1] &= \mathrm{A}x[n] + \mathrm{B}u[n] \\ y[n] &= \mathrm{C}x[n] + \mathrm{D}u[n] \end{aligned}$$

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

Analog Domain

`[b,a] = ellip(n,Rp,Rs,Wn,'s')` designs an order `n` lowpass analog elliptic filter with cutoff frequency `Wn` and returns the filter coefficients in the length `n + 1` row vectors `b` and `a`, in descending powers of s:

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

The *cutoff frequency* is the edge of the passband, at which the magnitude response of the filter is –`Rp` dB. For `ellip`, the cutoff frequency `Wn` must be greater than 0.

If Wn is a two-element vector with w1 < w2, then ellip(n,Rp,Rs,Wn,'s') returns an order 2*n bandpass analog filter with passband w1 < ω < w2.

[b,a] = ellip(n,Rp,Rs,Wn,'*ftype*','s') designs a highpass or bandstop filter, where *ftype* is

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

  The stopband is w1 < ω < w2.

With different numbers of output arguments, ellip directly obtains other realizations of the analog filter. To obtain zero-pole-gain form, use three output arguments:

[z,p,k] = ellip(n,Rp,Rs,Wn,'s') or

[z,p,k] = ellip(n,Rp,Rs,Wn,'*ftype*','s') returns the zeros and poles in length n or 2*n column vectors z and p and the gain in the scalar k.

To obtain state-space form, use four output arguments:

[A,B,C,D] = ellip(n,Rp,Rs,Wn,'s') or

[A,B,C,D] = ellip(n,Rp,Rs,Wn,'*ftype*','s') where A, B, C, and D are

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

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

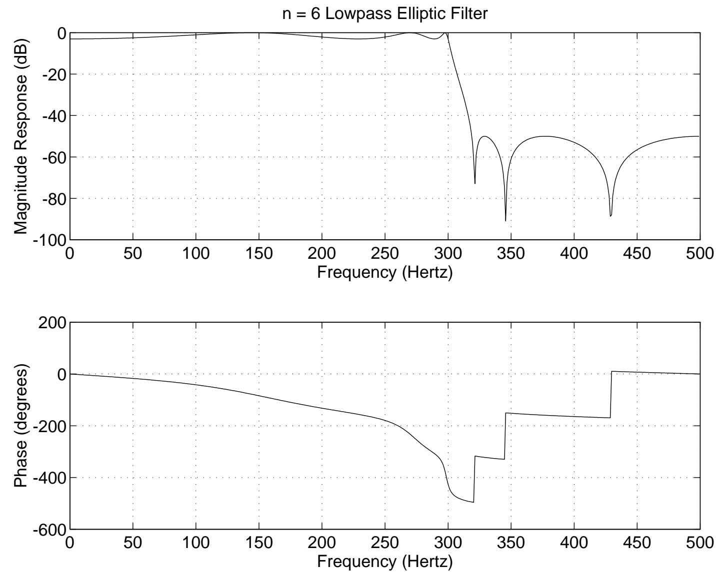
Examples

For data sampled at 1000 Hz, design a sixth-order lowpass elliptic filter with a cutoff frequency of 300 Hz, 3 dB of ripple in the passband, and 50 dB of attenuation in the stopband:

```
[b,a] = ellip(6,3,50,300/500);
```

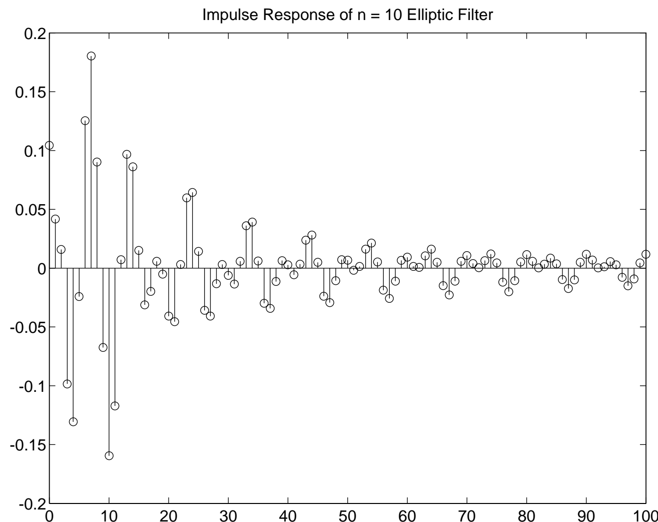
The filter's frequency response is

```
freqz(b,a,512,1000)
```



Design a 20th-order bandpass elliptic filter with a passband from 100 to 200 Hz and plot its impulse response:

```
n = 10; Rp = 0.5; Rs = 20;
Wn = [100 200]/500;
[b,a] = ellip(n,Rp,Rs,Wn);
[y,t] = impz(b,a,101); stem(t,y)
```



Limitations

For high order filters, the state-space form is the most numerically accurate, followed by the zero-pole-gain form. The transfer function form is the least accurate; numerical problems can arise for filter orders as low as 15.

Algorithm

The design of elliptic filters is the most difficult and computationally intensive of the Butterworth, Chebyshev type I and II, and elliptic designs. `ellip` uses a five-step algorithm:

1. It finds the lowpass analog prototype poles, zeros, and gain using the `ellipap` function.
2. It converts the poles, zeros, and gain into state-space form.
3. It transforms the lowpass filter to a bandpass, highpass, or bandstop filter with the desired cutoff frequencies using a state-space transformation.
4. For digital filter design, `ellip` uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at `Wn` or `w1` and `w2`.
5. It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also

`besself`	Bessel analog filter design.
`butter`	Butterworth analog and digital filter design.
`cheby1`	Chebyshev type I filter design (passband ripple).
`cheby2`	Chebyshev type II filter design (stopband ripple).
`ellipap`	Elliptic analog lowpass filter prototype.
`ellipord`	Elliptic filter order selection.

ellipap

Purpose

Elliptic analog lowpass filter prototype.

Syntax

```
[z,p,k] = ellipap(n,Rp,Rs)
```

Description

`[z,p,k] = ellipap(n,Rp,Rs)` returns the zeros, poles, and gain of an order `n` elliptic analog lowpass filter prototype, with `Rp` dB of ripple in the passband, and a stopband `Rs` dB down from the peak value in the passband. The zeros and poles are returned in length `n` column vectors `z` and `p` and the gain in scalar `k`. If `n` is odd, `z` is length `n – 1`. The transfer function is

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

Elliptic filters are equiripple in both the passband and stopband. They offer steeper rolloff characteristics than Butterworth and Chebyshev filters, but they are equiripple in both the passband and the stopband. Of the four classical filter types, elliptic filters usually meet a given set of filter performance specifications with the lowest filter order.

`ellip` sets the cutoff frequency ω_0 of the elliptic filter to 1 for a normalized result. The *cutoff frequency* is the frequency at which the passband ends and the filter has a magnitude response of $10^{-Rp/20}$.

Algorithm

`ellipap` uses the algorithm outlined in [1]. It employs the M-file `ellipk` to calculate the complete elliptic integral of the first kind and the M-file `ellipj` to calculate Jacobi elliptic functions.

See Also

`besselap`	Bessel analog lowpass filter prototype.
`buttap`	Butterworth analog lowpass filter prototype.
`cheb1ap`	Chebyshev type I analog lowpass filter prototype.
`cheb2ap`	Chebyshev type II analog lowpass filter prototype.
`ellip`	Elliptic (Cauer) filter design.

References

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

Purpose Elliptic filter order selection.

Syntax

```
[n,Wn] = ellipord(Wp,Ws,Rp,Rs)
[n,Wn] = ellipord(Wp,Ws,Rp,Rs,'s')
```

Description `ellipord` selects the minimum order digital or analog elliptic filter required to meet a set of lowpass filter design specifications:

`Wp`	Passband corner frequency. `Wp`, the cutoff frequency, has a value between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency).
`Ws`	Stopband corner frequency. `Ws` is in the same units as `Wp`; it has a value between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency).
`Rp`	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels. The passband is `0 < w < Wp`.
`Rs`	Stopband attenuation, in decibels. This value is the number of decibels the stopband is down from the passband. The stopband is `Ws < w < 1`.

Digital Domain

`[n,Wn] = ellipord(Wp,Ws,Rp,Rs)` returns the order `n` of the lowest order elliptic filter that loses no more than `Rp` dB in the passband and has at least `Rs` dB of attenuation in the stopband. The passband runs from 0 to `Wp` and the stopband extends from `Ws` to 1, the Nyquist frequency. `ellipord` also returns `Wn`, the cutoff frequency that allows `ellip` to achieve the given specifications.

Use `ellipord` for lowpass, highpass, bandpass, and bandstop filters. For highpass filters, `Wp` is greater than `Ws`. For bandpass and bandstop filters, `Wp` and `Ws` are two-element vectors that specify the corner frequencies at both edges of the filter, lower frequency edge first. For the band filters, `ellipord` returns `Wn` as a two-element row vector for input to `ellip`.

If filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design the filter as separate lowpass and highpass sections and cascade the two filters together.

Analog Domain

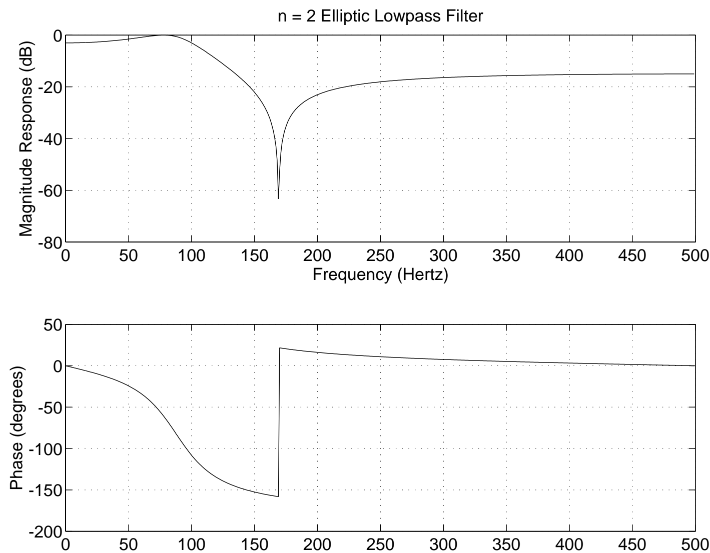
`[n,Wn] = ellipord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Wn` for an analog filter. In this case the frequencies in `Wp` and `Ws` are in radians per second and may be greater than 1.

Use `ellipord` for lowpass, highpass, bandpass, and bandstop filters as described under "Digital Domain."

Examples

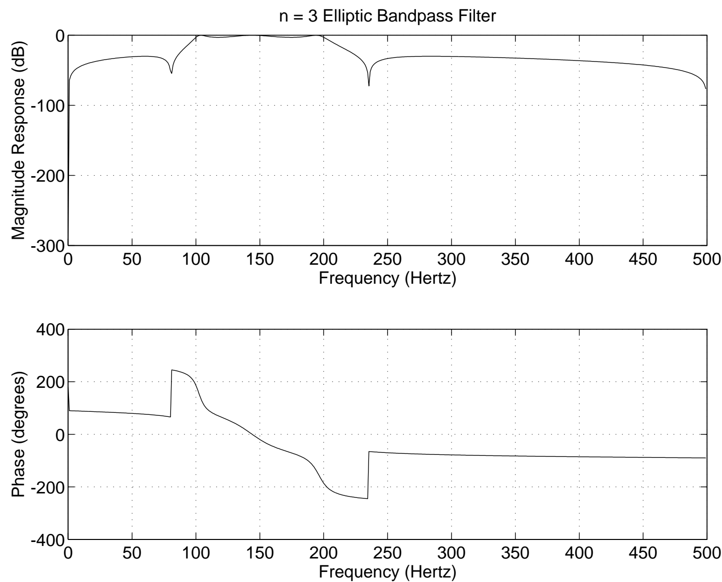
For 1000 Hz data, design a lowpass filter with less than 3 dB of attenuation from 0 to 100 Hz and at least 15 dB of attenuation from 150 Hz to the Nyquist frequency:

```
Wp = 100/500;  Ws = 150/500;
Rp = 3;  Rs = 15;
[n,Wn] = ellipord(Wp,Ws,Rp,Rs)
n =

     2
Wn =

    0.2000
[b,a] = ellip(n,Rp,Rs,Wn);
freqz(b,a,512,1000)
```



Now design a bandpass filter with a passband from 100 Hz to 200 Hz, less than 3 dB of ripple throughout the passband, and 30 dB stopbands 50 Hz out on both sides of the passband:

```
Wp = [100 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 30;
[n,Wn] = ellipord(Wp,Ws,Rp,Rs)

n =

     3

Wn =

    0.2000    0.4000

[b,a] = ellip(n,Rp,Rs,Wn);
freqz(b,a,512,1000)
```



Algorithm

`ellipord` uses the elliptic lowpass filter order prediction formula described in [1]. The function performs its calculations in the analog domain for both the analog and digital cases. For the digital case, it converts the frequency parameters to the *s*-domain before the order and natural frequency estimation process, then converts them back to the *z*-domain.

`ellipord` initially develops a lowpass filter prototype by transforming the passband frequencies of the desired filter to 1 rad/sec (for low- and highpass filters) and to -1 and 1 rad/sec (for bandpass and bandstop filters). It then computes the minimum order required for a lowpass filter to meet the stopband specification.

See Also

`buttord`	Butterworth filter order selection.
`cheb1ord`	Chebyshev type I filter order selection.
`cheb2ord`	Chebyshev type II filter order selection.
`ellip`	Elliptic (Cauer) filter design.

References

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

Purpose One-dimensional fast Fourier transform.

Syntax

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

Description `fft` computes the discrete Fourier transform of a vector or matrix. This function implements the transform given by

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

where $W_N = e^{-j(2\pi/N)}$ and N = `length(x)`. Note that the series is indexed as $n + 1$ and $k + 1$ instead of the usual n and k because MATLAB vectors run from 1 to N instead of from 0 to N-1.

`y = fft(x)` is the discrete Fourier transform of vector `x`, computed with a fast Fourier transform (FFT) algorithm. If `x` is a matrix, `y` is the FFT of each column of the matrix. The `fft` function employs a radix-2 fast Fourier transform algorithm if the length of the sequence is a power of two, and a slower algorithm if it is not; see the “Algorithm” section for details.

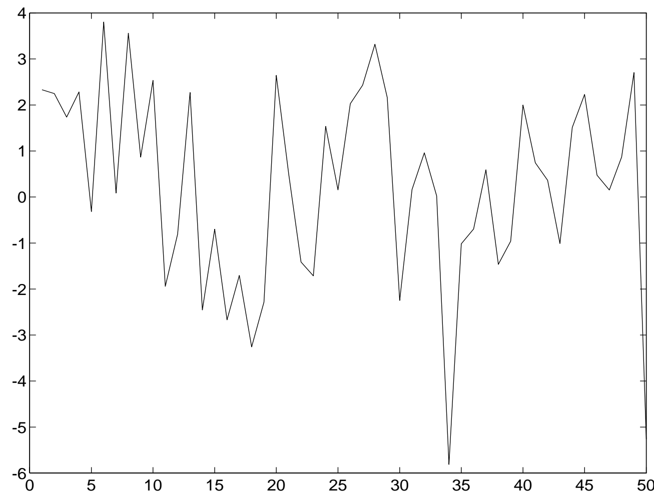
`y = fft(x,n)` is the `n`-point FFT. If the length of `x` is less than `n`, `fft` pads `x` with trailing zeros to length `n`. If the length of `x` is greater than `n`, `fft` truncates the sequence `x`. If `x` is an array, `fft` adjusts the length of the columns in the same manner.

`fft` is part of the standard MATLAB environment.

Example A common use of the Fourier transform is to find the frequency components of a time-domain signal buried in noise. Consider data sampled at 1000 Hz. Form

a signal consisting of 50 Hz and 120 Hz sinusoids and corrupt the signal with zero-mean random noise:

```
t = 0:0.001:0.6;
x = sin(2*pi*50*t) + sin(2*pi*120*t);
y = x + 2*randn(1,length(t));
plot(y(1:50))
```



It is difficult to identify the frequency components by studying the original signal. Convert to the frequency domain by taking the discrete Fourier transform of the noisy signal y using a 512-point fast Fourier transform (FFT):

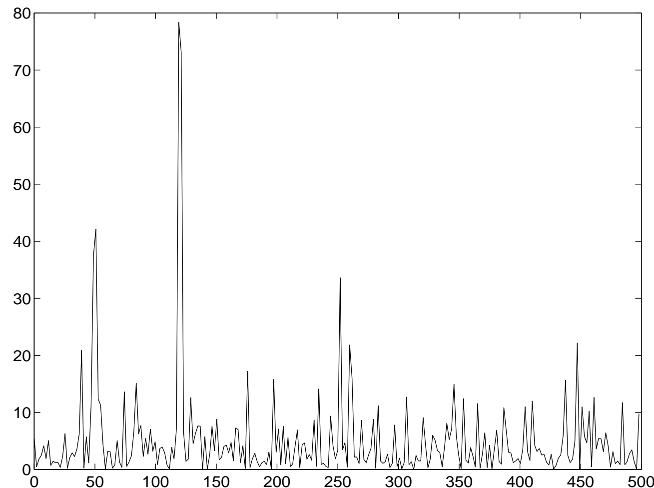
```
Y = fft(y,512);
```

The power spectral density, a measurement of the energy at various frequencies, is

```
Pyy = Y.*conj(Y) / 512;
```

Graph the first 256 points (the other 256 points are symmetric) on a meaningful frequency axis:

```
f = 1000*(0:255)/512;
plot(f,Pyy(1:256))
```



See the psd function for details on calculating spectral density.

Sometimes it is useful to normalize the output of `fft` so that a unit sinusoid in the time domain corresponds to unit amplitude in the frequency domain. To produce a normalized discrete-time Fourier transform in this manner, use

```
Pn = abs(fft(x))*2/length(x)
```

Algorithm

`fft` is a built-in MATLAB function. When the sequence length is a power of two, `fft` uses a high-speed radix-2 fast Fourier transform algorithm. The radix-2 FFT routine is optimized to perform a real FFT if the input sequence is purely real; otherwise it computes the complex FFT. This causes a real power-of-two FFT to be about 40% faster than a complex FFT of the same length.

When the sequence length is not an exact power of two, a separate algorithm finds the prime factors of the sequence length and computes the mixed-radix discrete Fourier transforms of the shorter sequences.

The execution time for `fft` depends on the sequence length. If the length of a sequence has many prime factors, the function computes the FFT quickly; if the length has few prime factors, execution is slower. For sequences whose lengths are prime numbers, `fft` uses the raw (and slow) DFT algorithm. For this reason it is usually better to use power-of-two FFTs, if this is supported by your application. For example, on one machine a 4096-point real FFT takes 2.1 seconds and a complex FFT of the same length takes 3.7 seconds. The FFTs of neighboring sequences of length 4095 and 4097, however, take 7 seconds and 58 seconds, respectively.

Suppose a sequence x of N points is obtained at a sample frequency of f_s. Then, for up to the Nyquist frequency, or point $n = N/2 + 1$, the relationship between the actual frequency and the index k into x (out of N possible indices) is

$$f = (k-1) * f_s / N$$

See Also

`dct`	Discrete cosine transform (DCT).
`dftmtx`	Discrete Fourier transform matrix.
`fft2`	Two-dimensional fast Fourier transform.
`fftshift`	Rearrange the outputs of `fft` and `fft2`.
`filter`	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.
`freqz`	Frequency response of digital filters.
`ifft`	One-dimensional inverse fast Fourier transform.
`psd`	Estimate the power spectral density (PSD) of a signal.

Purpose

Two-dimensional fast Fourier transform.

Syntax

```
Y = fft2(X)
Y = fft2(X,m,n)
```

Description

`Y = fft2(X)` performs a two-dimensional FFT, producing a result `Y` the same size as `X`. If `X` is a vector, `Y` has the same orientation as `X`.

`Y = fft2(X,m,n)` truncates or zero pads `X`, if necessary, to create an `m`-by-`n` array before performing the FFT. The result `Y` is also `m`-by-`n`.

`fft2` is part of the standard MATLAB environment.

Algorithm

`fft2(x)` is simply

```
fft(fft(x).').'
```

This computes the one-dimensional `fft` of each column of `x`, then of each row of the result. The time required to compute `fft2(x)` depends on the number of prime factors in `[m,n] = size(x)`. `fft2` is fastest when `m` and `n` are powers of 2.

See Also

`fft`	One-dimensional fast Fourier transform.
`fftshift`	Rearrange the outputs of `fft` and `fft2`.
`ifft`	One-dimensional inverse fast Fourier transform.
`ifft2`	Two-dimensional inverse fast Fourier transform.

Purpose FFT-based FIR filtering using the overlap-add method.

Syntax

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

Description `fftfilt` filters data using the efficient FFT-based method of *overlap-add*, a frequency domain filtering technique that works only for FIR filters.

`y = fftfilt(b,x)` filters the data in vector `x` with the filter described by coefficient vector `b`. It returns the data vector `y`. The operation performed by `fftfilt` is described in the *time domain* by the difference equation

$$y(n) = b(1)x(n) + b(2)x(n-1) + \cdots + b(nb+1)x(n-nb)$$

An equivalent representation is the *z*-transform or *frequency domain* description

$$Y(z) = (b(1) + b(2)z^{-1} + \cdots + b(nb+1)z^{-nb})X(z)$$

By default, `fftfilt` chooses an FFT length and data block length that guarantee efficient execution time.

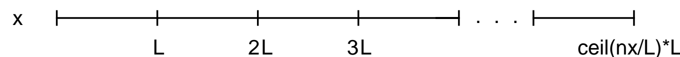
`y = fftfilt(b,x,n)` uses an FFT length of `nfft = 2^nextpow2(n)` and a data block length of `nfft - length(b) + 1`.

`fftfilt` works for both real and complex inputs.

Example Show that the results from `fftfilt` and `filter` are identical:

```
b = [1 2 3 4];
x = [1 zeros(1,99)]';
norm(fftfilt(b,x) - filter(b,1,x))

ans =
    9.5914e-15
```

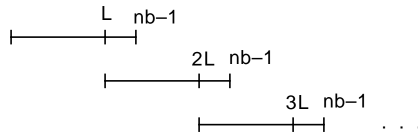
Algorithm `fftfilt` uses `fft` to implement the *overlap-add method* [1], a technique that combines successive frequency domain filtered blocks of an input sequence. `fftfilt` breaks an input sequence `x` into length `L` data blocks:



and convolves each block with the filter `b` by

```
y = ifft(fft(x(i:i+L-1),nfft).*fft(b,nfft));
```

where `nfft` is the FFT length. `fftfilt` overlaps successive output sections by `nb-1` points, where `nb` is the length of the filter, and sums them:



`fftfilt` chooses the key parameters `L` and `nfft` in different ways, depending on whether you supply an FFT length `n` and on the lengths of the filter and signal. If you do not specify a value for `n` (which determines FFT length), `fftfilt` chooses these key parameters automatically:

- If `length(x) > length(b)`, `fftfilt` chooses values that minimize the number of blocks times the number of flops per FFT.
- If `length(b) >= length(x)`, `fftfilt` uses a single FFT of length

  ```
  2^nextpow2(length(b) + length(x) - 1)
  ```

  This essentially computes

  ```
  y = ifft(fft(B,nfft).*fft(X,nfft))
  ```

If you supply a value for `n`, `fftfilt` chooses an FFT length, `nfft`, of `2^nextpow2(n)` and a data block length of `nfft - length(b) + 1`. If `n` is less than `length(b)`, `fftfilt` sets `n` to `length(b)`.

See Also

`filter`	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.
`filtfilt`	Zero-phase digital filtering.

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

fftshift

Purpose Rearrange the outputs of `fft` and `fft2`.

Syntax

```
y = fftshift(x)
```

Description `y = fftshift(x)` rearranges the outputs of `fft` and `fft2` by moving the zero frequency component to the center of the spectrum, which is sometimes a more convenient form.

For vectors, `fftshift(x)` returns a vector with the left and right halves swapped.

For arrays, `fftshift(x)` swaps quadrants one and three with quadrants two and four.

This function is part of the main MATLAB environment.

Example For any array `X`,

```
Y = fft2(x)
```

has `Y(1,1) = sum(sum(X))`; the DC component of the signal is in the upper-left corner of the two-dimensional FFT. For

```
Z = fftshift(Y)
```

the DC component is near the center of the matrix.

See Also

`fft`	One-dimensional fast Fourier transform.
`fft2`	Two-dimensional fast Fourier transform.

Purpose Filter data with a recursive (IIR) or nonrecursive (FIR) filter.

Syntax

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

Description `filter` is part of the MATLAB environment. It filters data using a digital filter. The filter realization is the *transposed direct form II* structure [1], which can handle both FIR and IIR filters.

If $a(1) \neq 1$, `filter` normalizes the filter coefficients by $a(1)$. If $a(1) = 0$, the input is in error.

`y = filter(b,a,x)` filters the data in vector `x` with the filter described by coefficient vectors `a` and `b` to create the filtered data vector `y`. When `x` is a matrix, filter operates on the columns of `x`. When `x` is an *N*-dimensional array, filter operates on the first non-singleton dimension.

`[y,zf] = filter(b,a,x)` returns the final values of the states in the vector `zf`.

`[...] = filter(b,a,x,zi)` specifies initial state conditions in the vector `zi`.

The size of the initial/final condition vector is `max(length(b),length(a))-1`. `zi` or `zf` can also be an array of such vectors, one for each column of `x` if `x` is a matrix. If `x` is a multidimensional array, `filter` works across the first nonsingleton dimension of `x` by default.

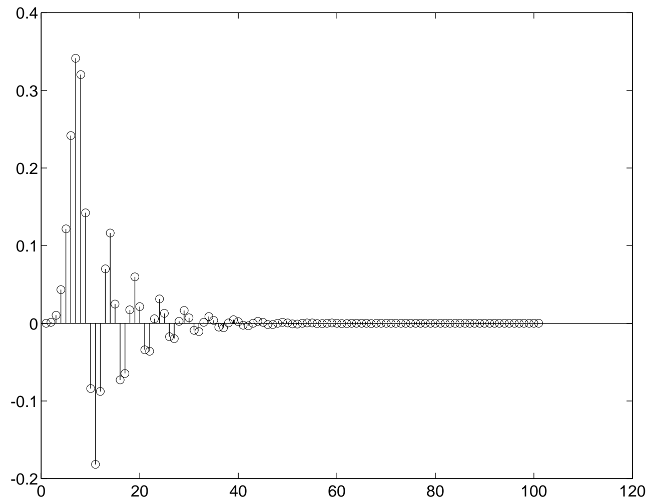
`[...] = filter(b,a,x,zi,dim)` works across the dimension `dim` of `x`. Set `zi` to empty to get the default initial conditions.

`filter` works for both real and complex inputs.

Example

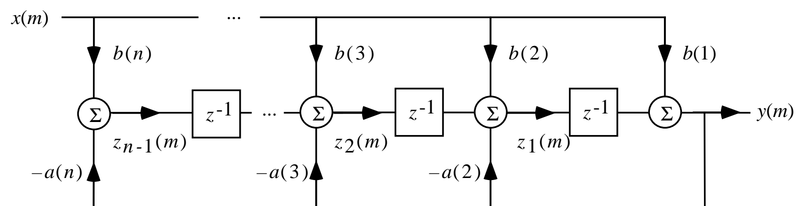
Find and graph the 100-point unit impulse response of a digital filter:

```
x = [1 zeros(1,100)];
[b,a] = butter(12,400/1000);
y = filter(b,a,x);
stem(y)
```



Algorithm

`filter` is a built-in MATLAB function. `filter` is implemented as a transposed direct form II structure



where n-1 is the filter order.

The operation of `filter` at sample m is given by the time domain difference equations for y and the states z_i :

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

You can use `filtic` to generate the state vector $z_i(0)$ from past inputs and outputs.

The input-output description of this filtering operation in the z-transform domain is a rational transfer function:

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

Diagnostics

If $a(1) = 0$, `filter` gives the following error message:

```
First denominator coefficient must be nonzero.
```

If the length of the initial condition vector is not the greater of *na* and *nb*, `filter` gives the following error message:

```
Initial condition vector has incorrect dimensions.
```

See Also

`fftfilt`	FFT-based FIR filtering using the overlap-add method.
`filter2`	Two-dimensional digital filtering.
`filtfilt`	Zero-phase digital filtering.
`filtic`	Make initial conditions for `filter` function.

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. Pgs. 311-312.

filter2

Purpose

Two-dimensional digital filtering.

Syntax

```
Y = filter2(B,X)
Y = filter2(B,X,'shape')
```

Description

`Y = filter2(B,X)` filters the two-dimensional data in `X` with the two-dimensional FIR filter in the matrix `B`. The result, `Y`, is computed using two-dimensional convolution and is the same size as `X`.

`Y = filter2(B,X,'shape')` returns `Y` computed with size specified by *shape*:

- `same` returns the central part of the convolution that is the same size as `X` (default).
- `full` returns the full two-dimensional convolution, `size(Y) > size(X)`.
- `valid` returns only those parts of the convolution that are computed without the zero-padded edges, `size(Y) < size(X)`.

Algorithm

`filter2` is part of the MATLAB environment. It uses `conv2` to compute the full two-dimensional convolution of the FIR filter with the input matrix. By default, `filter2` extracts and returns the central part of the convolution that is the same size as the input matrix. Use the *shape* parameter to specify an alternate part of the convolution for return.

See Also

`conv2`	Two-dimensional convolution.
`filter`	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.

Purpose

Zero-phase digital filtering.

Syntax

```
y = filtfilt(b,a,x)
```

Description

`y = filtfilt(b,a,x)` performs zero-phase digital filtering by processing the input data in both the forward and reverse directions (see problem 5.39 in [1]). After filtering in the forward direction, it reverses the filtered sequence and runs it back through the filter. The resulting sequence has precisely zero-phase distortion and double the filter order. `filtfilt` minimizes startup and ending transients by matching initial conditions.

`filtfilt` works for both real and complex inputs.

Algorithm

`filtfilt` is an M-file that uses the `filter` function. In addition to the forward-reverse filtering, it attempts to minimize startup transients by adjusting initial conditions to match the DC component of the signal and by prepending several filter lengths of a flipped, reflected copy of the input signal.

See Also

`fftfilt`	FFT-based FIR filtering using the overlap-add method.
`filter`	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.
`filter2`	Two-dimensional digital filtering.

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. Pgs. 311-312.

filtic

Purpose Make initial conditions for `filter` function.

Syntax

```
z = filtic(b,a,y,x)
z = filtic(b,a,y)
```

Description `z = filtic(b,a,y,x)` finds the initial conditions `z` for the delays in the *transposed direct form II* filter implementation given past outputs `y` and inputs `x`. The vectors `b` and `a` represent the numerator and denominator coefficients, respectively, of the filter's transfer function.

The vectors `x` and `y` contain the most recent input or output first, and oldest input or output last:

$$x = \{x(-1), x(-2), x(-3), \ldots, x(-nb), \ldots\}$$

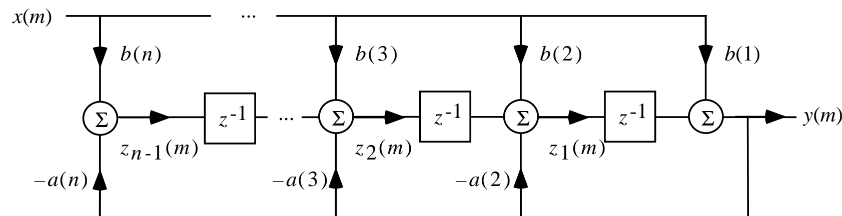
$$y = \{y(-1), y(-2), y(-3), \ldots, y(-na), \ldots\}$$

where `nb` is `length(b)-1` (the numerator order) and `na` is `length(a)-1` (the denominator order). If `length(x)` is less than `nb`, `filtic` pads it with zeros to length `nb`; if `length(y)` is less than `na`, `filtic` pads it with zeros to length `na`. Elements of `x` beyond `x(nb-1)` and elements of `y` beyond `y(na-1)` are unnecessary so `filtic` ignores them.

Output `z` is a column vector of length equal to the larger of *nb* and *na*. `z` describes the state of the delays given past inputs `x` and past outputs `y`.

`z = filtic(b,a,y)` assumes that the input `x` is 0 in the past.

The transposed direct form II structure is



where *n*-1 is the filter order.

`filtic` works for both real and complex inputs.

Algorithm

`filtic` performs a reverse difference equation to obtain the delay states `z`.

Diagnostics

If any of the input arguments `y`, `x`, `b`, or `a` is not a vector (that is, if any argument is a scalar or array), `filtic` gives the following error message:

```
Requires vector inputs.
```

See Also

`filter`	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.
`filtfilt`	Zero-phase digital filtering.

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. Pgs. 296, 301-302.

fir1

Purpose

Window-based finite impulse response filter design—standard response.

Syntax

```
b = fir1(n,Wn)
b = fir1(n,Wn,'ftype')
b = fir1(n,Wn,window)
b = fir1(n,Wn,'ftype',window)
b = fir1(...,'noscale')
```

Description

`fir1` implements the classical method of windowed linear-phase FIR digital filter design [1]. It designs filters in standard lowpass, bandpass, highpass, and bandpass configurations. (For windowed filters with arbitrary frequency response, use `fir2`.)

`b = fir1(n,Wn)` returns row vector `b` containing the `n + 1` coefficients of an order `n` lowpass FIR filter. This is a Hamming-windowed, linear-phase filter with cutoff frequency `Wn`. The output filter coefficients, `b`, are ordered in descending powers of *z*.

$$b(z) = b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}$$

`Wn`, the cutoff frequency, is a number between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency).

If `Wn` is a two-element vector, `Wn = [w1 w2]`, `fir1` returns a bandpass filter with passband `w1 < ω < w2`.

If `Wn` is a multi-element vector, `Wn = [w1 w2 w3 w4 w5 ... wn]`, `fir1` returns an order `n` multiband filter with bands `0 < w < w1`, `w1 < w < w2`, ..., `wn < w < 1`.

By default, the filter is scaled so that the center of the first passband has magnitude exactly 1 after windowing.

`b = fir1(n,Wn,'ftype')` specifies a filter type, where *ftype* is

- `high` for a highpass filter with cutoff frequency `Wn`
- `stop` for a bandstop filter, if `Wn = [w1 w2]`
  The stopband is `w1 < ω < w2`.
- `'DC-1'` to make the first band of a multiband filter a passband
- `'DC-0'` to make the first band of a multiband filter a stopband

`fir1` always uses an even filter order for the highpass and bandstop configurations. This is because for odd orders, the frequency response at the Nyquist frequency is 0, which is inappropriate for highpass and bandstop filters. If you specify an odd-valued `n`, `fir1` increments it by 1.

`b = fir1(n,Wn,window)` uses the window specified in column vector `window` for the design. The vector `window` must be `n+1` elements long. If no window is specified, `fir1` employs a Hamming window.

`b = fir1(n,Wn,'ftype',window)` accepts both *`ftype`* and *`window`* parameters.

`b = fir1(...,'noscale')` turns off the default scaling.

The group delay of the FIR filter designed by `fir1` is `n/2`.

Algorithm

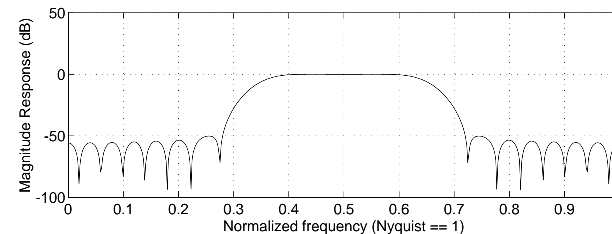
`fir1` uses the window method of FIR filter design [1]. If *w(n)* denotes a window, where $1 \le n \le N$, and the impulse response of the ideal filter is *h(n)*, where *h(n)* is the inverse Fourier transform of the ideal frequency response, then the windowed digital filter coefficients are given by

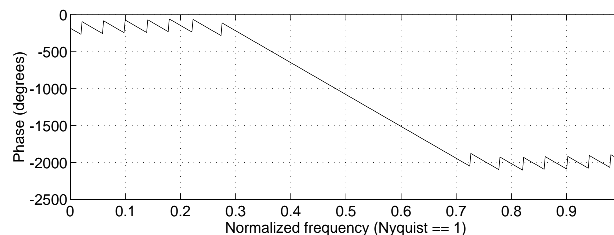
$$b(n) = w(n)h(n), \quad 1 \le n \le N$$

Examples

Design a 24th-order FIR bandpass filter with passband $0.35 \le w \le 0.65$:

```
b = fir1(48,[0.35 0.65]);
freqz(b,1,512)
```





Design a 34th order FIR highpass filter with a cutoff frequency of 0.48, using a Chebyshev window with 30 dB of ripple:

```
b = fir1(34,0.48,'high',chebwin(35,30));
xfilt = filter(b,1,x);
```

Diagnostics

If n is odd and you specify a bandstop or highpass filter, fir1 gives the following warning message:

```
For highpass and bandstop filters, N must be even.
Order is being increased by 1.
```

See Also

filter	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.
fir2	Window-based finite impulse response filter design—arbitrary response.
fircls	Constrained least square FIR filter design for multiband filters.
fircls1	Constrained least square filter design for lowpass and highpass linear phase FIR filters.
firls	Least square linear-phase FIR filter design.
freqz	Frequency response of digital filters.
kaiserord	Estimate parameters for fir1 with Kaiser window.
remez	Parks-McClellan optimal FIR filter design.

References

[1] IEEE. *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979. Algorithm 5.2.

Purpose

Window-based finite impulse response filter design—arbitrary response.

Syntax

```
b = fir2(n,f,m)
b = fir2(n,f,m,window)
b = fir2(n,f,m,npt)
b = fir2(n,f,m,npt,window)
b = fir2(n,f,m,npt,lap)
b = fir2(n,f,m,npt,lap,window)
```

Description

`fir2` designs windowed digital FIR filters with arbitrarily shaped frequency response. (For standard lowpass, bandpass, highpass, and bandstop configurations, use `fir1`.)

`b = fir2(n,f,m)` returns row vector `b` containing the `n+1` coefficients of an order `n` FIR filter. The frequency-magnitude characteristics of this filter match those given by vectors `f` and `m`:

- `f` is a vector of frequency points in the range from 0 to 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency). The first point of `f` must be 0 and the last point 1. The frequency points must be in increasing order.
- `m` is a vector containing the desired magnitude response at the points specified in `f`.
- `f` and `m` must be the same length.
- Duplicate frequency points are allowed, corresponding to steps in the frequency response.

Use `plot(f,m)` to view the filter shape.

The output filter coefficients, `b`, are ordered in descending powers of z:

$$b(z) = b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}$$

`b = fir2(n,f,m,window)` uses the window specified in column vector `window` for the filter design. The vector `window` must be `n+1` elements long. If no window is specified, `fir2` employs a Hamming window.

`b = fir2(n,f,m,npt)` and

`b = fir2(n,f,m,npt, window)` specify the number of points `npt` for the grid onto which `fir2` interpolates the frequency response, with or without a `window` specification.

`b = fir2(n,f,m,npt,lap)` and

`b = fir2(n,f,m,npt,lap,window)` specify the size of the region, `lap`, that `fir2` inserts around duplicate frequency points, with or without a `window` specification.

See the "Algorithm" section for more on `npt` and `lap`.

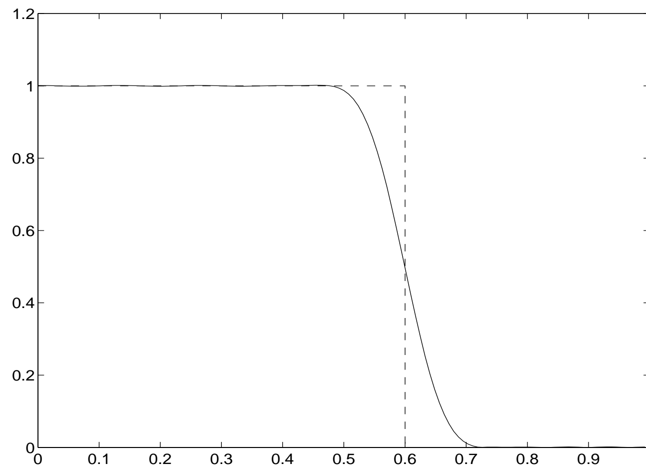
Algorithm

The desired frequency response is interpolated onto a dense, evenly spaced grid of length `npt`. `npt` is 512 by default. If two successive values of `f` are the same, a region of `lap` points is set up around this frequency to provide a smooth but steep transition in the requested frequency response. By default, `lap` is 25. The filter coefficients are obtained by applying an inverse fast Fourier transform to the grid and multiplying by a window; by default, this is a Hamming window.

Example

Design a 30th-order lowpass filter and overplot the desired frequency response with the actual frequency response:

```
f = [0 0.6 0.6 1]; m = [1 1 0 0];
b = fir2(30,f,m);
[h,w] = freqz(b,1,128);
plot(f,m,w/pi,abs(h))
```



See Also

`butter`	Butterworth analog and digital filter design.
`cheby1`	Chebyshev type I filter design (passband ripple).
`cheby2`	Chebyshev type II filter design (stopband ripple).
`ellip`	Elliptic (Cauer) filter design.
`fir1`	Window-based finite impulse response filter design—standard response.
`maxflat`	Generalized digital Butterworth filter design.
`remez`	Parks-McClellan optimal FIR filter design.
`yulewalk`	Recursive digital filter design.

fircls

Purpose

Constrained least square FIR filter design for multiband filters.

Syntax

```
b = fircls(n,f,amp,up,lo)
fircls(n,f,amp,up,lo,'design_flag')
```

Description

`b = fircls(n,f,amp,up,lo)` generates a length `n+1` linear phase FIR filter `b`. The frequency-magnitude characteristics of this filter match those given by vectors `f` and `amp`:

- `f` is a vector of transition frequencies in the range from 0 to 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency). The first point of `f` must be 0 and the last point 1. The frequency points must be in increasing order.
- `amp` is a vector describing the piecewise constant desired amplitude of the frequency response. The length of `amp` is equal to the number of bands in the response and should be equal to `length(f)-1`.
- `up` and `lo` are vectors with the same length as `amp`. They define the upper and lower bounds for the frequency response in each band.

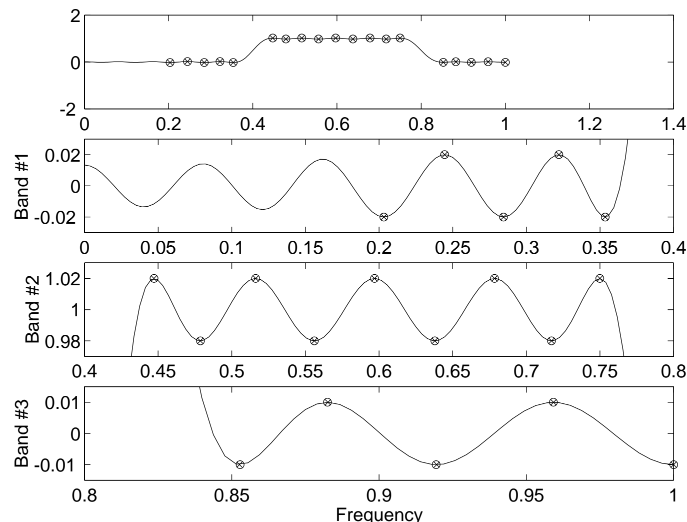
`fircls(n,f,amp,up,lo,'design_flag')` enables you to monitor the filter design, where *`design_flag`* can be

- `trace`, for a textual display of the design table used in the design
- `plots`, for plots of the filter's magnitude, group delay, and zeros and poles
- `both`, for both the textual display and plots

Example

Design an order 50 bandpass filter:

```
n = 50;
f = [0 0.4 0.8 1];
amp = [0 1 0];
up = [0.02 1.02 0.01];
lo = [-0.02 0.98 -0.01];
b = fircls(n,f,amp,up,lo,'plots') %plots magnitude response
```



NOTE Normally, the lower value in the stopband will be specified as negative. By setting `lo` equal to 0 in the stopbands, a nonnegative frequency response amplitude can be obtained. Such filters can be spectrally factored to obtain minimum phase filters.

Algorithm

The algorithm is a multiple exchange algorithm that uses Lagrange multipliers and Kuhn-Tucker conditions on each iteration.

See Also

`fircls1`	Constrained least square filter design for lowpass and highpass linear phase FIR filters.
`firls`	Least square linear-phase FIR filter design.
`remez`	Parks-McClellan optimal FIR filter design.

References

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

[2] Selesnick, I.W., M. Lang, and C.S. Burrus. “Constrained Least Square Design of FIR Filters without Specified Transition Bands.” *IEEE Transactions on Signal Processing*, Vol. 44, No. 8 (August 1996).

Purpose

Constrained least square filter design for lowpass and highpass linear phase FIR filters.

Syntax

```
b = fircls1(n,wo,dp,ds)
b = fircls1(n,wo,dp,ds,'high')
b = fircls1(n,wo,dp,ds,wt)
b = fircls1(n,wo,dp,ds,wt,'high')
b = fircls1(n,wo,dp,ds,wp,ws,k)
b = fircls1(n,wo,dp,ds,wp,ws,k,'high')
b = fircls1(n,wo,dp,ds,...,'design_flag')
```

Description

`b = fircls1(n,wo,dp,ds)` generates a lowpass FIR filter `b`. `n+1` is the filter length, `wo` is the normalized cutoff frequency in the range between 0 and 1 (where 1 corresponds to half the sampling frequency, that is, the Nyquist frequency), `dp` is the maximum passband deviation from 1 (passband ripple), and `ds` is the maximum stopband deviation from 0 (stopband ripple).

`b = fircls1(n,wo,dp,ds,'high')` generates a highpass FIR filter `b`.

`b = fircls1(n,wo,dp,ds,wt)` and

`b = fircls1(n,wo,dp,ds,wt,'high')` specify a frequency `wt` above which (for `wt>wo`) or below which (for `wt<wo`) the filter is guaranteed to meet the given band criterion. This will help you design a filter that meets a passband or stopband edge requirement. There are four cases:

- Lowpass:
  - `0<wt<wo<1`: the amplitude of the filter is within `dp` of 1 over the frequency range $0 < \omega <$ `wt`.
  - `0<wo<wt<1`: the amplitude of the filter is within `ds` of 0 over the frequency range `wt` $< \omega < 1$.
- Highpass:
  - `0<wt<wo<1`: the amplitude of the filter is within `ds` of 0 over the frequency range $0 < \omega <$ `wt`.
  - `0<wo<wt<1`: the amplitude of the filter is within `dp` of 1 over the frequency range `wt` $< \omega < 1$.

`b = fircls1(n,wo,dp,ds,wp,ws,k)` generates a lowpass FIR filter `b` with a weighted function. `n+1` is the filter length, `wo` is the normalized cutoff frequency, `dp` is the maximum passband deviation from 1 (passband ripple), and `ds` is the maximum stopband deviation from 0 (stopband ripple). `wp` is the passband edge of the L2 weight function and `ws` is the stopband edge of the L2 weight function, where `wp` < `wo` < `ws`. `k` is the ratio (passband L2 error)/(stopband L2 error):

$$\frac{\int_0^{W_p} |A(\omega) - D(\omega)|^2 d\omega}{\int_{W_s}^{\pi} |A(\omega) - D(\omega)|^2 d\omega} = k$$

`b = fircls1(n,wo,dp,ds,wp,ws,k,'high')` generates a highpass FIR filter `b` with a weighted function, where `ws` < `wo` < `wp`.

`b = fircls1(n,wo,dp,ds,...,'design_flag')` enables you to monitor the filter design, where *`design_flag`* can be

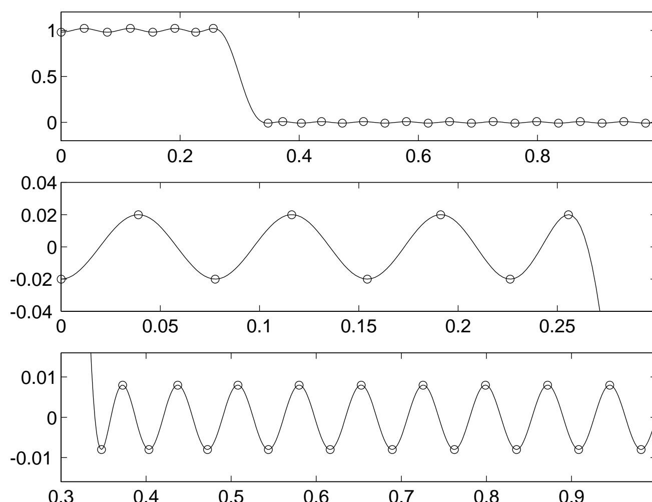
- `trace`, for a textual display of the design table used in the design
- `plots`, for plots of the filter's magnitude, group delay, and zeros and poles
- `both`, for both the textual display and plots

NOTE In the design of very narrow band filters with small `dp` and `ds`, there may not exist a filter of the given length that meets the specifications.

Example

Design an order 55 lowpass filter with a cutoff frequency at 0.3:

```
n = 55; wo = 0.3;
dp = 0.02; ds = 0.008;
b = fircls1(n,wo,dp,ds,'plots'); %plot magnitude response
```



Algorithm

The algorithm is a multiple exchange algorithm that uses Lagrange multipliers and Kuhn-Tucker conditions on each iteration.

See Also

`fircls`	Constrained least square FIR filter design for multiband filters.
`firls`	Least square linear-phase FIR filter design.
`remez`	Parks-McClellan optimal FIR filter design.

References

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

[2] Selesnick, I.W., M. Lang, and C.S. Burrus. "Constrained Least Square Design of FIR Filters without Specified Transition Bands." *IEEE Transactions on Signal Processing*, Vol. 44, No. 8 (August 1996).

firls

Purpose Least square linear-phase FIR filter design.

Syntax

```
b = firls(n,f,a)
b = firls(n,f,a,w)
b = firls(n,f,a,'ftype')
b = firls(n,f,a,w,'ftype')
```

Description `firls` designs a linear-phase FIR filter that minimizes the weighted, integrated squared error between an ideal piecewise linear function and the magnitude response of the filter over a set of desired frequency bands.

`b = firls(n,f,a)` returns row vector `b` containing the `n+1` coefficients of the order `n` FIR filter whose frequency-amplitude characteristics approximately match those given by vectors `f` and `a`. The output filter coefficients, or "taps," in `b` obey the symmetry relation

$$b(k) = b(n+2-k), \qquad k = 1, \ldots, n+1$$

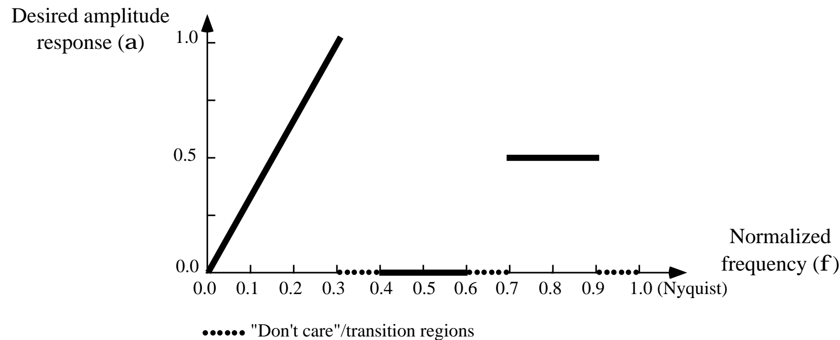
These are type I (`n` odd) and type II (`n` even) linear-phase filters. Vectors `f` and `a` specify the frequency-amplitude characteristics of the filter:

- `f` is a vector of pairs of frequency points, specified in the range between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency). The frequencies must be in increasing order. Duplicate frequency points are allowed and, in fact, can be used to design a filter exactly the same as those returned by the `fir1` and `fir2` functions with a rectangular or `boxcar` window.
- `a` is a vector containing the desired amplitude at the points specified in `f`.

  The desired amplitude function at frequencies between pairs of points ($f(k)$, $f(k+1)$) for k odd is the line segment connecting the points ($f(k)$, $a(k)$) and ($f(k+1)$, $a(k+1)$).

  The desired amplitude function at frequencies between pairs of points ($f(k)$, $f(k+1)$) for k even is unspecified. These are transition or "don't care" regions.
- `f` and `a` are the same length. This length must be an even number.

The relationship between the `f` and `a` vectors in defining a desired amplitude response is



`b = firls(n,f,a,w)` uses the weights in vector `w` to weight the fit in each frequency band. The length of `w` is half the length of `f` and `a`, so there is exactly one weight per band.

`b = firls(n,f,a,'ftype')` and

`b = firls(n,f,a,w,'ftype')` specify a filter type, where *`ftype`* is

- `hilbert` for linear-phase filters with odd symmetry (type III and type IV)

  The output coefficients in `b` obey the relation $b(k) = -b(n + 2 - k)$, $k = 1, ..., n + 1$. This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.

- `differentiator` for type III and type IV filters, using a special weighting technique

  For nonzero amplitude bands, the integrated squared error has a weight of $(1/f)^2$ so that the error at low frequencies is much smaller than at high frequencies. For FIR differentiators, which have an amplitude characteristic proportional to frequency, the filters minimize the relative integrated squared error (the integral of the square of the ratio of the error to the desired amplitude).

Examples

Design an order 255 lowpass filter with transition band:

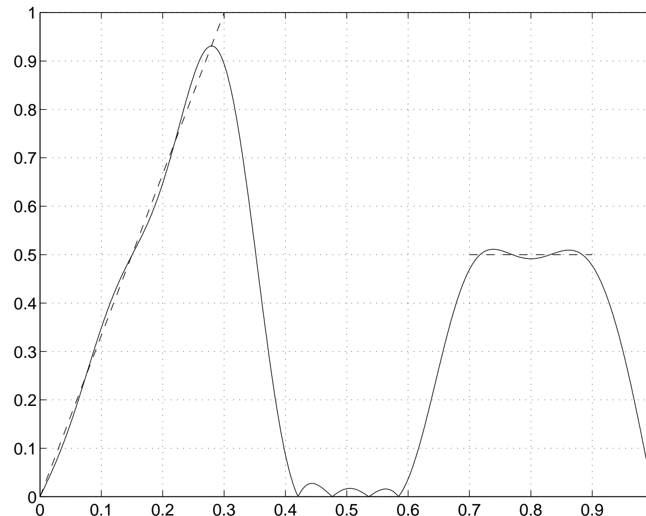
```
b = firls(255,[0 0.25 0.3 1],[1 1 0 0]);
```

Design a 31 coefficient differentiator:

```
b = firls(30,[0 0.9],[0 0.9],'differentiator');
```

Design a 24th-order anti-symmetric filter with piecewise linear passbands and plot the desired and actual frequency response:

```
F = [0 0.3  0.4 0.6  0.7 0.9];
A = [0  1   0  0   0.5 0.5];
b = firls(24,F,A,'hilbert');
for i=1:2:6,
   plot([F(i) F(i+1)],[A(i) A(i+1)],'--'), hold on
end
[H,f] = freqz(b,1,512,2);
plot(f,abs(H)), grid on, hold off
```



Algorithm

Reference [1] describes the theoretical approach that `firls` takes. The function solves a system of linear equations involving an inner product matrix of size roughly `n/2` using MATLAB's \ operator.

This function designs type I, II, III, and IV linear-phase filters. Type I and II are the defaults for `n` even and odd respectively, while the `'hilbert'` and `'differentiator'` flags produce type III (`n` even) and IV (`n` odd) filters. The various filter types have different symmetries and constraints on their frequency responses (see [2] for details).

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

Diagnostics

An appropriate diagnostic message is displayed when incorrect arguments are used:

```
F must be even length.
F and A must be equal lengths.
Requires symmetry to be 'hilbert' or 'differentiator'.
Requires one weight per band.
Frequencies in F must be nondecreasing.
Frequencies in F must be in range [0,1].
```

A more serious warning message is

```
Warning: Matrix is close to singular or badly scaled.
```

This tends to happen when the filter length times the transition width grows large. In this case, the filter coefficients `b` might not represent the desired filter. You can check the filter by looking at its frequency response.

See Also

`fir1`	Window-based finite impulse response filter design—standard response.
`fir2`	Window-based finite impulse response filter design—arbitrary response.
`firrcos`	Raised cosine FIR filter design.
`remez`	Parks-McClellan optimal FIR filter design.

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Pgs. 54-83.

[2] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. Pgs. 256-266.

Purpose

Raised cosine FIR filter design.

```
b = firrcos(n,F0,df,Fs)
b = firrcos(n,F0,df)
```

Description

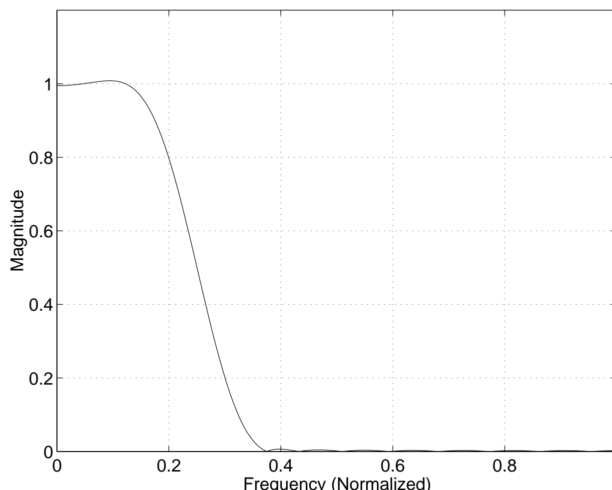
`firrcos(n,F0,df,Fs)` returns an order `n` lowpass linear-phase FIR filter with a raised cosine transition band. The filter has cutoff frequency `F0`, transition width `df`, and sampling frequency `Fs`, all in Hertz. `F0` must be between 0 and *Fs/2*. `df` must be small enough so that `F0` ± `df/2` is between 0 and *Fs*/2. The filter order `n` must be even.

`firrcos(n,F0,df)` uses a default sampling frequency of `Fs = 2`.

Example

Design an order 20 raised cosine FIR filter with cutoff frequency 0.25 of the Nyquist frequency and a transition width of 0.25:

```
h = firrcos(20,0.25,0.25);
H = fft(h,1024);
plot((0:1023)/1024*2,abs(H)),axis([0 1 0 1.2]),grid
```



Remarks

`firrcos` minimizes the integral squared error in the frequency domain.

See Also

`firls`	Least square linear-phase FIR filter design.
`remez`	Parks-McClellan optimal FIR filter design.

Purpose

Frequency response of analog filters.

Syntax

```
h = freqs(b,a,w)
[h,w] = freqs(b,a)
[h,w] = freqs(b,a,n)
freqs(b,a)
```

Description

`freqs` returns the complex frequency response *H(jw)* (Laplace transform) of an analog filter:

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

given the numerator and denominator coefficients in vectors `b` and `a`.

`h = freqs(b,a,w)` returns the complex frequency response of the analog filter specified by coefficient vectors `b` and `a`. `freqs` evaluates the frequency response along the imaginary axis in the complex plane at the frequencies specified in real vector `w`.

`[h,w] = freqs(b,a)` automatically picks a set of 200 frequency points `w` on which to compute the frequency response `h`.

`[h,w] = freqs(b,a,n)` picks `n` frequencies on which to compute the frequency response `h`.

`freqs` with no output arguments plots the magnitude and phase response versus frequency in the current figure window.

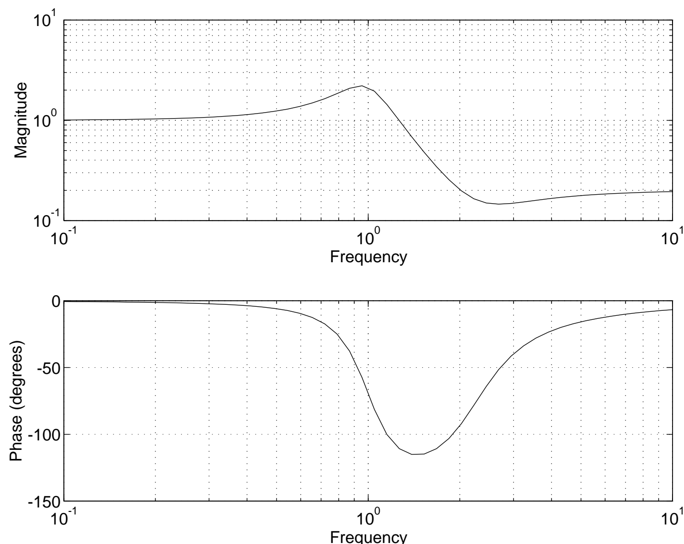
`freqs` works only for real input systems and positive frequencies.

Example

Find and graph the frequency response of the transfer function given by

$$H(s) = \frac{0.2s^2 + 0.3s + 1}{s^2 + 0.4s + 1}$$

```
a = [1 0.4 1];
b = [0.2 0.3 1];
w = logspace(-1,1);
freqs(b,a,w)
```



You can also create the plot with

```
h = freqs(b,a,w);
mag = abs(h);
phase = angle(h);
subplot(2,1,1), loglog(w,mag)
subplot(2,1,2), semilogx(w,phase)
```

To convert to Hertz, degrees, and decibels, use

```
f = w/(2*pi);
mag = 20*log10(mag);
phase = phase*180/pi;
```

Algorithm

`freqs` evaluates the polynomials at each frequency point, then divides the numerator response by the denominator response:

```
s = i*w;
h = polyval(b,s)./polyval(a,s);
```

See Also

`abs`	Absolute value (magnitude).
`angle`	Phase angle.
`freqz`	Frequency response of digital filters.
`invfreqs`	Continuous-time (analog) filter identification from frequency data.
`logspace`	Generate logarithmically spaced vectors (see MATLAB Function Reference).
`polyval`	Polynomial evaluation (see MATLAB Function Reference).

Purpose

Frequency spacing for frequency response.

Syntax

```
f = freqspace(n)
f = freqspace(n,'whole')
[f1,f2] = freqspace(n)
[f1,f2] = freqspace([m n])
[x1,y1] = freqspace(n,'meshgrid')
[x1,y1] = freqspace([m n],'meshgrid')
```

Description

`freqspace` returns the implied frequency range for equally spaced frequency responses. This is useful when creating frequency vectors for use with `freqz`.

`f = freqspace(n)` returns the frequency vector `f` assuming `n` evenly spaced points around the unit circle. For `n` even or odd, `f` is `(0:2/n:1)`. For `n` even, `freqspace` returns `(n + 2)/2` points. For `N` odd, it returns `(n + 1)/2` points.

`f = freqspace(n,'whole')` returns `n` evenly spaced points around the whole unit circle. In this case, `f` is `0:2/n:2*(n–1)/n`.

`[f1,f2] = freqspace(n)` returns the two-dimensional frequency vectors `f1` and `f2` for an `n`-by-`n` matrix. For `n` odd, both `f1` and `f2` are `[–1 + 1/n:2/n:1–1/n]`. For `n` even, both `f1` and `f2` are `[–1:2/n:1–2/n]`.

`[f1,f2] = freqspace([m n])` returns the two-dimensional frequency vectors `f1` and `f2` for an `m`-by-`n` matrix.

`[x1,y1] = freqspace(n,'meshgrid')` and

`[x1,y1] = freqspace([m n],'meshgrid')` are equivalent to

```
[f1,f2] = freqspace(...);
[x1,y1] = meshgrid(f1,f2);
```

See the MATLAB Function Reference for details on the `meshgrid` function.

See Also

`freqz`	Frequency response of digital filters.
`invfreqz`	Discrete-time filter identification from frequency data.

freqz

Purpose

Frequency response of digital filters.

Syntax

```
[h,w] = freqz(b,a,n)
[h,f] = freqz(b,a,n,Fs)
[h,w] = freqz(b,a,n,'whole')
[h,f] = freqz(b,a,n,'whole',Fs)
h = freqz(b,a,w)
h = freqz(b,a,f,Fs)
freqz(b,a)
```

Description

`freqz` returns the complex frequency response $H(e^{jw})$ of a digital filter, given the numerator and denominator coefficients in vectors `b` and `a`.

`[h,w] = freqz(b,a,n)` returns the `n`-point complex frequency response of the digital filter

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

given the coefficient vectors `b` and `a`. `freqz` returns both `h`, the complex frequency response, and `w`, a vector containing the `n` frequency points. `freqz` evaluates the frequency response at `n` points equally spaced around the upper half of the unit circle, so `w` contains `n` points between 0 and π.

It is best, although not necessary, to choose a value for `n` that is an exact power of two, because this allows fast computation using an FFT algorithm. If you do not specify a value for `n`, it defaults to 512.

`[h,f] = freqz(b,a,n,Fs)` specifies a positive sampling frequency `Fs`, in Hertz. It returns a vector `f` containing the actual frequency points between 0 and `Fs/2` at which it calculated the frequency response. `f` is of length `n`.

`[h,w] = freqz(b,a,n,'whole')` and

`[h,f] = freqz(b,a,n,'whole',Fs)` use `n` points around the whole unit circle (from 0 to 2π, or from 0 to `Fs`).

`h = freqz(b,a,w)` returns the frequency response at the frequencies in vector `w`. These frequencies must be between 0 and 2π.

`h = freqz(b,a,f,Fs)` returns the frequency response at the frequencies in vector `f`, where the elements of `f` are between 0 and `Fs`.

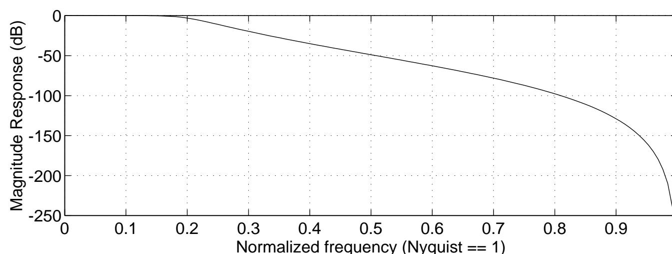
`freqz` with no output arguments plots the magnitude and phase response versus frequency in the current figure window.

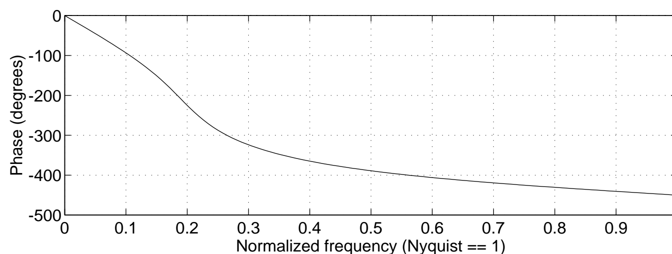
`freqz` works for both real and complex input systems.

Example

Plot the magnitude and phase response of a Butterworth filter:

```
[b,a] = butter(5,0.2);
freqz(b,a,128)
```





Algorithm

`freqz` uses an FFT algorithm when argument `n` is present. It computes the frequency response as the ratio of the transformed numerator and denominator coefficients, padded with zeros to the desired length:

```
h = fft(b,n)./fft(a,n)
```

If `n` is not a power of two, the FFT algorithm is not as efficient and may cause long computation times.

When a frequency vector `w` or `f` is present, or if `n` is less than `max(length(b),length(a))`, `freqz` evaluates the polynomials at each frequency point using Horner's method of polynomial evaluation and then divides the numerator response by the denominator response.

See Also

`abs`	Absolute value (magnitude).
`angle`	Phase angle.
`fft`	One-dimensional fast Fourier transform.
`filter`	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.
`freqs`	Frequency response of analog filters.
`impz`	Impulse response of digital filters.
`invfreqz`	Discrete-time filter identification from frequency data.
`logspace`	Generate logarithmically spaced vectors (see MATLAB Function Reference).

Purpose Gaussian-modulated sinusoidal pulse generator.

Syntax

```
yi = gauspuls(t,fc,bw)
yi = gauspuls(t,fc,bw,bwr)
[yi,yq] = gauspuls(...)
[yi,yq,ye] = gauspuls(...)
tc = gauspuls('cutoff',fc,bw,bwr,tpe)
```

Description `gauspuls` generates Gaussian-modulated sinusoidal pulses.

`yi = gauspuls(t,fc,bw)` returns a unity-amplitude Gaussian RF pulse at the times indicated in array `t`, with a center frequency `fc` in Hertz and a fractional bandwidth `bw`, which must be greater than 0. The default value for `fc` is 1000 Hz and for `bw` is 0.5.

`yi = gauspuls(t,fc,bw,bwr)` returns a unity-amplitude Gaussian RF pulse with a bandwidth of `100*bw` as measured at a level of `bwr` dB with respect to the normalized signal peak. The fractional bandwidth reference level `bwr` must be less than 0, because it indicates a reference level less than the peak (unity) envelope amplitude. The default value for `bwr` is -6 dB.

`[yi,yq] = gauspuls(...)` returns both the in-phase and quadrature pulses.

`[yi,yq,ye] = gauspuls(...)` returns the RF signal envelope.

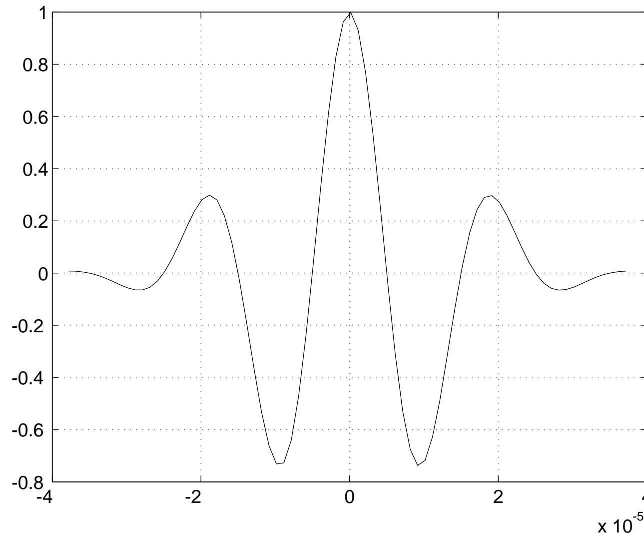
`tc = gauspuls('cutoff',fc,bw,bwr,tpe)` returns the cutoff time `tc` (greater than or equal to 0) at which the trailing pulse envelope falls below `tpe` dB with respect to the peak envelope amplitude. The trailing pulse envelope level `tpe` must be less than 0, because it indicates a reference level less than the peak (unity) envelope amplitude. The default value for `tpe` is -60 dB.

Remarks Default values are substituted for empty or omitted trailing input arguments.

Example

Plot a 50 kHz Gaussian RF pulse with 60% bandwidth, sampled at a rate of 1 MHz. Truncate the pulse where the envelope falls 40 dB below the peak:

```
tc = gauspuls('cutoff',50e3,0.6,[],-40);
t = -tc : 1e-6 : tc;
yi = gauspuls(t,50e3,0.6);
plot(t,yi)
```



See Also

`chirp`	Swept-frequency cosine generator.
`cos`	Cosine of vector/matrix elements (see MATLAB Function Reference).
`diric`	Dirichlet or periodic sinc function.
`pulstran`	Pulse train generator.
`rectpuls`	Sampled aperiodic rectangle generator.
`sawtooth`	Sawtooth or triangle wave generator.
`sin`	Sine of vector/matrix elements (see MATLAB Function Reference).
`sinc`	Sinc or $\sin(\pi t)/\pi t$ function.
`square`	Square wave generator.
`tripuls`	Sampled aperiodic triangle generator.

Purpose Average filter delay (group delay).

Syntax

```
[gd,w] = grpdelay(b,a,n)
[gd,f] = grpdelay(b,a,n,Fs)
[gd,w] = grpdelay(b,a,n,'whole')
[gd,f] = grpdelay(b,a,n,'whole',Fs)
gd = grpdelay(b,a,w)
gd = grpdelay(b,a,f,Fs)
grpdelay(b,a)
```

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

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

where ω is frequency and θ is the phase angle of $H(e^{jw})$.

`[gd,w] = grpdelay(b,a,n)` returns the `n`-point group delay, $\tau_g(\omega)$, of the digital filter

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

given the numerator and denominator coefficients in vectors `b` and `a`. `grpdelay` returns both `gd`, the group delay, and `w`, a vector containing the `n` frequency points in radians. `grpdelay` evaluates the group delay at `n` points equally spaced around the upper half of the unit circle, so `w` contains `n` points between 0 and π. A value for `n` that is an exact power of two allows fast computation using an FFT algorithm.

`[gd,f] = grpdelay(b,a,n,Fs)` specifies a positive sampling frequency `Fs` in Hertz. It returns a length `n` vector `f` containing the actual frequency points at which the group delay is calculated, also in Hertz. `f` contains `n` points between 0 and `Fs/2`.

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

`[gd,f] = grpdelay(b,a,n,'whole',Fs)` use `n` points around the whole unit circle (from 0 to 2π, or from 0 to `Fs`).

`gd = grpdelay(b,a,w)` and

`gd = grpdelay(b,a,f,Fs)` return the group delay evaluated at the points in `w` (in radians) or `f` (in Hertz), respectively, where `Fs` is the sampling frequency in Hertz.

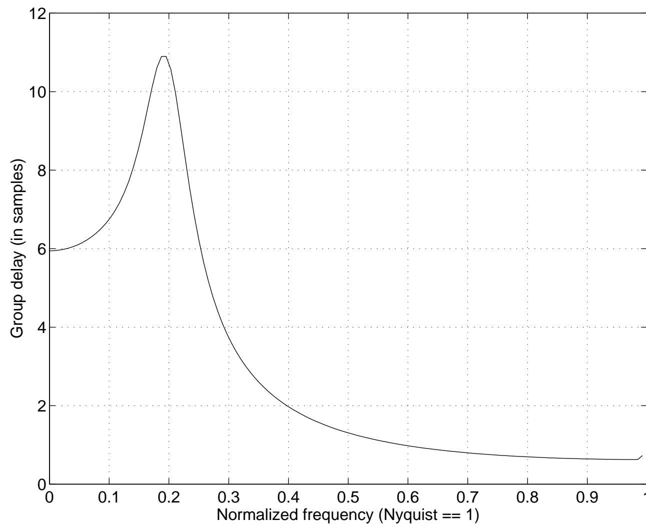
`grpdelay` with no output arguments plots the group delay versus frequency in the current figure window.

`grpdelay` works for both real and complex input systems.

Examples

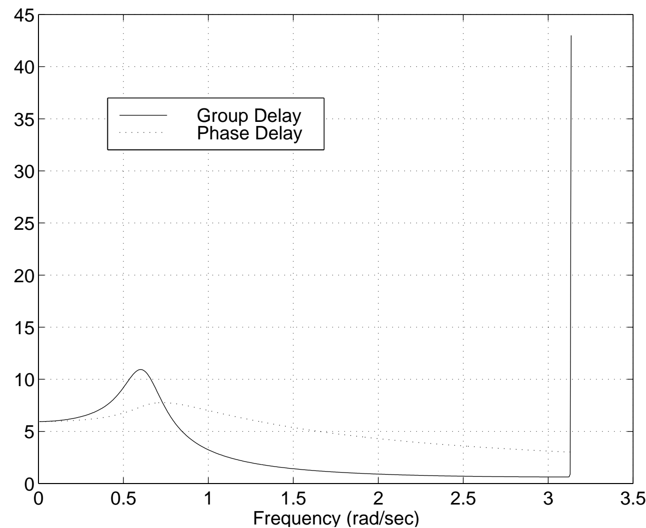
Plot the group delay of Butterworth filter *b(z)/a(z)*:

```
[b,a] = butter(6,0.2);
grpdelay(b,a,128)
```



Plot both the group and phase delays of a system on the same graph:

```
gd = grpdelay(b,a,512);
gd(1) = []; % avoid NaNs
[h,w] = freqz(b,a,512); h(1) = []; w(1) = [];
pd = -unwrap(angle(h))./w;
plot(w,gd,w,pd,':')
```



Algorithm

`grpdelay` multiplies the filter coefficients by a unit ramp. After Fourier transformation, this process corresponds to differentiation.

See Also

`cceps`	Complex cepstral analysis.
`fft`	One-dimensional fast Fourier transform.
`freqz`	Frequency response of digital filters.
`hilbert`	Hilbert transform.
`icceps`	Inverse complex cepstrum.
`rceps`	Real cepstrum and minimum phase reconstruction.

hamming

Purpose Hamming window.

Syntax

```
w = hamming(n)
```

Description

`w = hamming(n)` returns an `n`-point Hamming window in the column vector `w`. The coefficients of a Hamming window are

$$w[k+1] = 0.54 - 0.46\cos\left(2\pi\frac{k}{n-1}\right), \quad k = 0, \ldots, n-1$$

See Also

`bartlett`	Bartlett window.
`blackman`	Blackman window.
`boxcar`	Rectangular window.
`chebwin`	Chebyshev window.
`hanning`	Hanning window.
`kaiser`	Kaiser window.
`triang`	Triangular window.

Purpose Hanning window.

Syntax

```
w = hanning(n)
```

Description `w = hanning(n)` returns an `n`-point Hanning window in the column vector `w`. The coefficients of a Hanning window are

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

See Also

`bartlett`	Bartlett window.
`blackman`	Blackman window.
`boxcar`	Rectangular window.
`chebwin`	Chebyshev window.
`hamming`	Hamming window.
`kaiser`	Kaiser window.
`triang`	Triangular window.

hilbert

Purpose Hilbert transform.

Syntax

```
y = hilbert(x)
```

Description

`y = hilbert(x)` returns a complex helical sequence, sometimes called the *analytic signal*, from a real data sequence. The analytic signal has a real part, which is the original data, and an imaginary part, which contains the Hilbert transform. The imaginary part is a version of the original real sequence with a 90° phase shift. Sines are therefore transformed to cosines and vice versa. The Hilbert transformed series has the same amplitude and frequency content as the original real data and includes phase information that depends on the phase of the original data.

If `x` is a matrix, `y = hilbert(x)` operates columnwise on the matrix, finding the Hilbert transform of each column.

The Hilbert transform is useful in calculating instantaneous attributes of a time series, especially the amplitude and frequency. The instantaneous amplitude is the amplitude of the complex Hilbert transform; the instantaneous frequency is the time rate of change of the instantaneous phase angle. For a pure sinusoid, the instantaneous amplitude and frequency are constant. The instantaneous phase, however, is a sawtooth, reflecting the way in which the local phase angle varies linearly over a single cycle. For mixtures of sinusoids, the attributes are short term, or local, averages spanning no more than two or three points.

Reference [1] describes the Kolmogorov method for minimum phase reconstruction, which involves taking the Hilbert transform of the logarithm of the spectrum of a time series. The toolbox function `rceps` performs this reconstruction.

Algorithm

The analytic signal for a sequence `x` has a *one-sided Fourier transform*, that is, negative frequencies are 0. To approximate the analytic signal, `hilbert` calculates the FFT of the input sequence, replaces those FFT coefficients that correspond to negative frequencies with zeros, and calculates the inverse FFT of the result.

In detail, `hilbert` uses a four-step algorithm:

1. It calculates the FFT of the input sequence, storing the result in a vector `y`. Before transforming, it zero pads the input sequence so its length `n` is the closest power of two, if necessary. This ensures the most efficient FFT computation.
2. It creates a vector `h` whose elements `h(i)` have the values
   - 1 for `i` = 1, `(n/2) + 1`
   - 2 for `i` = 2, 3, ..., `(n/2)`
   - 0 for `i` = `(n/2) + 2`, ... , `n`
3. It calculates the element-wise product of `y` and `h`.
4. It calculates the inverse FFT of the sequence obtained in step 3 and returns the first `n` elements of the result.

If the input data `x` is a matrix, `hilbert` operates in a similar manner, extending each step above to handle the matrix case.

See Also

`fft`	One-dimensional fast Fourier transform.
`ifft`	One-dimensional inverse fast Fourier transform.
`rceps`	Real cepstrum and minimum phase reconstruction.

References

[1] Claerbout, J.F. *Fundamentals of Geophysical Data Processing*. New York: McGraw-Hill, 1976. Pgs. 59-62.

icceps

Purpose Inverse complex cepstrum.

Syntax

```
x = icceps(xhat,nd)
```

Description `x = icceps(xhat,nd)` returns the inverse complex cepstrum of the (assumed real) sequence `xhat`, removing `nd` samples of delay. If `xhat` was obtained with `cceps(x)`, then the amount of delay that was added to `x` was the element of `round(unwrap(angle(fft(x)))/pi)` corresponding to π radians.

See Also

`cceps`	Complex cepstral analysis.
`hilbert`	Hilbert transform.
`rceps`	Real cepstrum and minimum phase reconstruction.
`unwrap`	Unwrap phase angles.

References

[1] Oppenheim, A.V., and R.W. Schafer. *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975.

Purpose

Inverse discrete cosine transform.

Syntax

```
x = idct(y)
x = idct(y,n)
```

Description

The inverse discrete cosine transform reconstructs a sequence from its discrete cosine transform (DCT) coefficients. The `idct` function is the inverse of the `dct` function.

`x = idct(y)` returns the inverse discrete cosine transform of `y`

$$x(n+1) = \frac{1}{N}\sum_{k=0}^{N-1} w(k)y(k+1)\cos\frac{\pi}{2N}(k+1)(2n+1)\ , \qquad n = 0, \ldots, N-1$$

where

$$w(k) = \begin{cases} \frac{1}{2}, & k = 0 \\ 1, & 1 \le k \le N-1 \end{cases}$$

and `N = length(x)`. `x` is the same size as `y`. The series is indexed from $n + 1$ and $k + 1$ instead of the usual n and k because MATLAB vectors run from 1 to N instead of from 0 to N-1.

`x = idct(y,n)` appends zeros or truncates the vector `y` to length `n` before transforming.

If `y` is a matrix, `idct` transforms its columns.

See Also

`dct`	Discrete cosine transform (DCT).
`dct2`	Two-dimensional DCT (see *Image Processing Toolbox User's Manual*).
`idct2`	Two-dimensional inverse DCT (see *Image Processing Toolbox User's Manual*).
`ifft`	One-dimensional inverse fast Fourier transform.

Purpose One-dimensional inverse fast Fourier transform.

Syntax

```
y = ifft(x)
y = ifft(x,n)
```

Description `ifft` computes the inverse Fourier transform of a vector or array. This function implements the inverse transform given by

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

where $W_N = e^{-j(2p/N)}$ and N = `length(x)`. Note that the series is indexed as $n + 1$ and $k + 1$ instead of the usual n and k because MATLAB vectors run from 1 to N instead of from 0 to N-1.

`y = ifft(x)` is the inverse Fourier transform of vector `x`. If `x` is an array, `y` is the inverse FFT of each column of the matrix.

`y = ifft(x,n)` is the `n`-point inverse FFT. If the length of `x` is less than `n`, `ifft` pads `x` with trailing zeros to length `n`. If the length of `x` is greater than `n`, `ifft` truncates the sequence `x`. When `x` is an array, `ifft` adjusts the length of the columns in the same manner.

`ifft` is part of the standard MATLAB environment.

Algorithm `ifft` is an M-file. The algorithm for `ifft` is the same as that for `fft`, except for a sign change and a scale factor of `n = length(x)`. The execution time is fastest when `n` is a power of two and slowest when `n` is a large prime.

See Also

`fft`	One-dimensional fast Fourier transform.
`fft2`	Two-dimensional fast Fourier transform.
`fftshift`	Rearrange the outputs of `fft` and `fft2`.
`ifft2`	Two-dimensional inverse fast Fourier transform.

Purpose

Two-dimensional inverse fast Fourier transform.

Syntax

```
Y = ifft2(X)
Y = ifft2(X,m,n)
```

Description

`Y = ifft2(X)` returns the two-dimensional inverse fast Fourier transform (FFT) of the array `X`. If `X` is a vector, `Y` has the same orientation as `X`.

`Y = ifft2(X,m,n)` truncates or zero pads `X`, if necessary, to create an `m`-by-`n` array before performing the inverse FFT. The result `Y` is also `m`-by-`n`.

For any `X`, `ifft2(fft2(X))` equals `X` to within roundoff error. If `X` is real, `ifft2(fft2(X))` may have small imaginary parts.

`ifft` is part of the standard MATLAB environment.

Algorithm

The algorithm for `ifft2` is the same as that for `fft2`, except for a sign change and scale factors of `[m n]=size(X)`. The execution time is fastest when `m` and `n` are powers of two and slowest when they are large primes.

`ifft2` is part of the standard MATLAB environment.

See Also

`fft`	One-dimensional fast Fourier transform.
`fft2`	Two-dimensional fast Fourier transform.
`fftn`	*N*-dimensional fast Fourier transform (see MATLAB Function Reference).
`fftshift`	Rearrange the outputs of `fft` and `fft2`.
`ifft`	One-dimensional inverse fast Fourier transform.
`ifftn`	*N*-dimensional inverse fast Fourier transform (see MATLAB Function Reference).

impinvar

Purpose

Impulse invariance method of analog-to-digital filter conversion.

Syntax

```
[bz,az] = impinvar(b,a,Fs)
[bz,az] = impinvar(b,a)
```

Description

`[bz,az] = impinvar(b,a,Fs)` creates a digital filter with numerator and denominator coefficients `bz` and `az`, respectively, whose impulse response is equal to the impulse response of the analog filter with coefficients `b` and `a`, scaled by `1/Fs`.

`[bz,az] = impinvar(b,a)` uses the default value of 1 Hz for `Fs`.

Example

Convert an analog lowpass filter to a digital filter using `impinvar` with a sampling frequency of 10 Hz:

```
[b,a] = butter(4,0.3,'s');
[bz,az] = impinvar(b,a,10);
real(bz)

ans =

   1.0e-06 *

        0         0    0.1324    0.5192    0.1273

real(az)

ans =

    1.0000   -3.9216    5.7679   -3.7709    0.9246
```

Algorithm

`impinvar` performs the impulse-invariant method of analog-to-digital transfer function conversion discussed in reference [1]:

1. It finds the partial fraction expansion of the system represented by `b` and `a`.
2. It replaces the poles `p` by the poles `exp(p/Fs)`.
3. It finds the transfer function coefficients of the system from the residues from step 1 and the poles from step 2.

See Also

`bilinear`	Map variables using bilinear transformation.
`lp2bp`	Lowpass to bandpass analog filter transformation.
`lp2bs`	Lowpass to bandstop analog filter transformation.
`lp2hp`	Lowpass to highpass analog filter transformation.
`lp2lp`	Lowpass to lowpass analog filter transformation.

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Pgs. 206-209.

impz

Purpose

Impulse response of digital filters.

Syntax

```
[h,t] = impz(b,a)
[h,t] = impz(b,a,n)
[h,t] = impz(b,a,n,Fs)
impz(b,a)
impz(...)
```

Description

`[h,t] = impz(b,a)` computes the impulse response of the filter with numerator coefficients `b` and denominator coefficients `a`. `impz` chooses the number of samples and returns the response in column vector `h` and times (or sample intervals) in column vector `t` (where `t = (0:n-1)'` and `n` is the computed impulse response length).

`[h,t] = impz(b,a,n)` computes `n` samples of the impulse response. If `n` is a vector of integers, `impz` computes the impulse response at those integer locations where 0 is the starting point of the filter.

`[h,t] = impz(b,a,n,Fs)` computes `n` samples and scales `t` so that samples are spaced `1/Fs` units apart. `Fs` is 1 by default.

`[h,t] = impz(b,a,[],Fs)` chooses the number of samples for you and scales `t` so that samples are spaced `1/Fs` units apart.

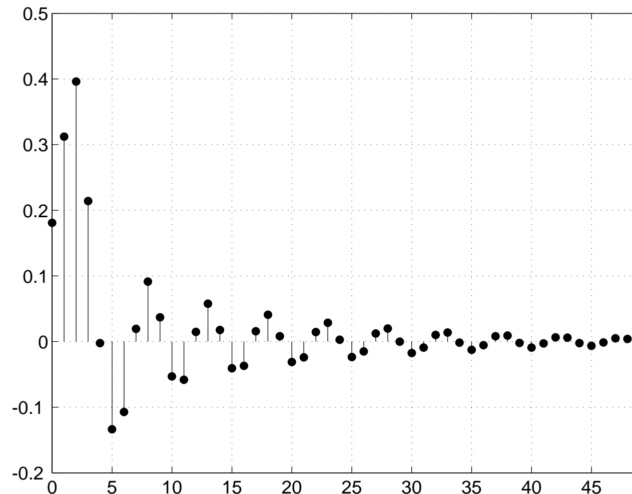
`impz` with no output arguments plots the impulse response in the current figure window using `stem(t,h)`.

`impz` works for both real and complex input systems.

Example

Plot the first 50 samples of the impulse response of a fourth-order lowpass elliptic filter with cutoff frequency of 0.4 times the Nyquist frequency:

```
[b,a] = ellip(4,0.5,20,0.4);
impz(b,a,50)
```



Algorithm

`impz` filters a length `n` impulse sequence using

```
filter(b,a,[1 zeros(1,n-1)])
```

To compute `n` in the auto-length case, `impz` either uses `n = length(b)` for the FIR case or first finds the poles using `p = roots(a)`, if `length(a)` is greater than 1.

If the filter is unstable, `n` is chosen to be the point at which the term from the largest pole reaches `10^6` times its original value.

If the filter is stable, `n` is chosen to be the point at which the term due to the largest amplitude pole is `5*10^-5` of its original amplitude.

If the filter is oscillatory (poles on the unit circle only), `impz` computes five periods of the slowest oscillation.

If the filter has both oscillatory and damped terms, `n` is chosen to equal five periods of the slowest oscillation or the point at which the term due to the

largest (nonunity) amplitude pole is 5*10^–5 of its original amplitude, whichever is greater.

impz also allows for delay in the numerator polynomial, which it adds to the resulting n.

See Also

impulse	Unit impulse response (see *Control Toolbox User's Guide*).
stem	Plot discrete sequence data (see MATLAB Function Reference).

Purpose

Increase sampling rate by an integer factor (interpolation).

Syntax

```
y = interp(x,r)
y = interp(x,r,l,alpha)
[y,b] = interp(x,r,l,alpha)
```

Description

Interpolation increases the original sampling rate for a sequence to a higher rate. `interp` performs lowpass interpolation by inserting zeros into the original sequence and then applying a special lowpass filter.

`y = interp(x,r)` increases the sampling rate of `x` by a factor of `r`. The interpolated vector `y` is `r` times longer than the original input `x`.

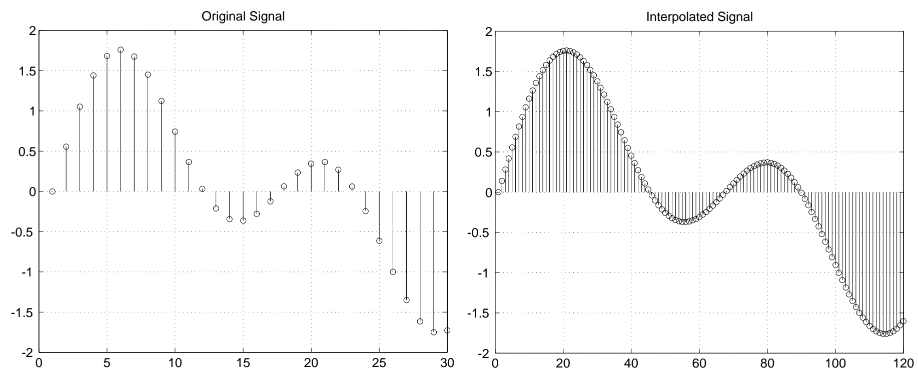
`y = interp(x,r,l,alpha)` specifies `l` (filter length) and `alpha` (cut-off frequency). The default value for `l` is 4 and the default value for `alpha` is 0.5.

`[y,b] = interp(x,r,l,alpha)` returns vector `b` containing the filter coefficients used for the interpolation.

Example

Interpolate a signal by a factor of four:

```
t = 0:0.001:1; % time vector
x = sin(2*pi*30*t) + sin(2*pi*60*t);
y = interp(x,4);
stem(x(1:30))
stem(y(1:120)), axis([0 120 -2 2])
```



Algorithm

`interp` uses the lowpass interpolation Algorithm 8.1 described in [1]:

1. It expands the input vector to the correct length by inserting zeros between the original data values.
2. It designs a special symmetric FIR filter that allows the original data to pass through unchanged and interpolates between so that the mean-square errors between the interpolated points and their ideal values are minimized.
3. It applies the filter to the input vector to produce the interpolated output vector.

The length of the FIR lowpass interpolating filter is `2*l*r+1`. The number of original sample values used for interpolation is `2*l`. Ordinarily, `l` should be less than or equal to 10. The original signal is assumed to be band limited with normalized cutoff frequency 0 ≤ `alpha` ≤ 1, where 1 is half the original sampling frequency (the Nyquist frequency). The default value for `l` is 4 and the default value for `alpha` is 0.5.

Diagnostics

If `r` is not an integer, `interp` gives the following error message:

```
Resampling rate R must be an integer.
```

See Also

`decimate`	Decrease the sampling rate for a sequence (decimation).
`interp1`	1-D data interpolation (table lookup) (see MATLAB Function Reference).
`resample`	Change sampling rate by any factor.
`spline`	Cubic spline interpolation (see MATLAB Function Reference).
`upfirdn`	Apply FIR filter and perform sample rate conversion.

References

[1] IEEE. *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979. Algorithm 8.1.

Purpose Interpolation FIR filter design.

Syntax

```
b = intfilt(r,l,alpha)
b = intfilt(r,n,'Lagrange')
```

Description `b = intfilt(r,l,alpha)` designs a linear phase FIR filter that performs ideal bandlimited interpolation using the nearest `2*l-1` nonzero samples, when used on a sequence interleaved with `r-1` consecutive zeros every `r` samples. It assumes an original bandlimitedness of `alpha` times the Nyquist frequency. The returned filter is identical to that used by `interp`.

`b = intfilt(r,n,'Lagrange')` or `b = intfilt(r,n,'l')` designs an FIR filter that performs `n`th-order Lagrange polynomial interpolation on a sequence interleaved with `r-1` consecutive zeros every `r` samples. `b` has length `(n + 1)*r` for `n` even, and length `(n + 1)*r-1` for `n` odd.

Both types of filters are basically lowpass and are intended for interpolation and decimation.

Examples Design a digital interpolation filter to upsample a signal by four, using the bandlimited method:

```
alpha = 0.5; % "bandlimitedness" factor
h1 = intfilt(4,2,alpha); % bandlimited interpolation
```

The filter `h1` works best when the original signal is bandlimited to `alpha` times the Nyquist frequency. Create a bandlimited noise signal:

```
randn('seed',0)
x = filter(fir1(40,0.5),1,randn(200,1)); % bandlimit
```

Now zero pad the signal with three zeros between every sample. The resulting sequence is four times the length of `x`:

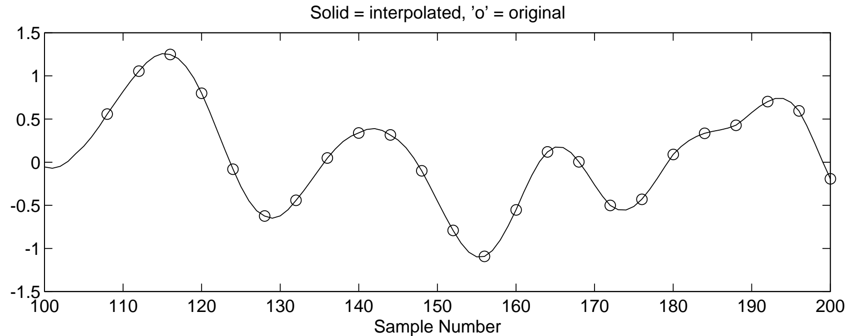
```
xr = reshape([x zeros(length(x),3)]',4*length(x),1);
```

Interpolate using the `filter` command:

```
y = filter(h1,1,xr);
```

y is an interpolated version of x, delayed by seven samples (the group-delay of the filter). Zoom in on a section to see this:

```
plot(100:200,y(100:200),7+(101:4:196),x(26:49),'o')
```



intfilt's other type of filter performs Lagrange polynomial interpolation of the original signal. For example, first-order polynomial interpolation is just linear interpolation, which is accomplished with a triangular filter:

```
h2 = intfilt(4,1,'l')    % Lagrange interpolation
h2 =
    0.2500    0.5000    0.7500    1.0000    0.7500    0.5000   0.2500
```

Algorithm

The bandlimited method uses firls to design an interpolation FIR equivalent to that presented in [1]. The polynomial method uses Lagrange's polynomial interpolation formula on equally spaced samples to construct the appropriate filter.

See Also

decimate	Decrease the sampling rate for a sequence (decimation).
interp	Increase sampling rate by an integer factor (interpolation).
resample	Change sampling rate by any factor.

References

[1] Oetken, Parks, and Schüßler. "New Results in the Design of Digital Interpolators." IEEE *Trans. Acoust., Speech, Signal Processing*. Vol. ASSP-23 (June 1975). Pgs. 301-309.

Purpose

Continuous-time (analog) filter identification from frequency data.

Syntax

```
[b,a] = invfreqs(h,w,nb,na)
[b,a] = invfreqs(h,w,nb,na,wt)
[b,a] = invfreqs(h,w,nb,na,wt,iter)
[b,a] = invfreqs(h,w,nb,na,wt,iter,tol)
[b,a] = invfreqs(h,w,nb,na,wt,iter,tol,'trace')
```

Description

`invfreqs` is the inverse operation of `freqs`; it finds a continuous-time transfer function that corresponds to a given complex frequency response. From a laboratory analysis standpoint, `invfreqs` is useful in converting magnitude and phase data into transfer functions.

`[b,a] = invfreqs(h,w,nb,na)` returns the real numerator and denominator coefficient vectors `b` and `a` of the transfer function

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^{nb} + b(2)s^{(nb-1)} + \ldots + b(nb+1)}{a(1)s^{na} + a(2)s^{(na-1)} + \ldots + a(na+1)}$$

whose complex frequency response is given in vector `h` at the frequency points specified in vector `w`. Scalars `nb` and `na` specify the desired orders of the numerator and denominator polynomials.

Frequency is specified in radians and the length of `h` must be the same as the length of `w`.

`[b,a] = invfreqs(h,w,nb,na,wt)` weights the fit-errors versus frequency. `wt` is a vector of weighting factors the same length as `w`.

`invfreqs(h,w,nb,na,wt,iter)` and

`invfreqs(h,w,nb,na,wt,iter,tol)` provide a superior algorithm that guarantees stability of the resulting linear system and searches for the best fit using a numerical, iterative scheme. The `iter` parameter tells `invfreqs` to end the iteration when the solution has converged, or after `iter` iterations, whichever comes first. `invfreqs` defines convergence as occurring when the norm of

the (modified) gradient vector is less than `tol`. `tol` is an optional parameter that defaults to 0.01. To obtain a weight vector of all ones, use

```
invfreqs(h,w,nb,na,[],iter,tol)
```

`invfreqs(h,w,nb,na,wt,iter,tol,'trace')` displays a textual progress report of the iteration.

Remarks

When building higher order models using high frequencies, it is important to scale the frequencies, dividing by a factor such as half the highest frequency present in `w`, so as to obtain well conditioned values of `a` and `b`. This corresponds to a rescaling of time.

Examples

Convert a simple transfer function to frequency response data and then back to the original filter coefficients:

```
a = [1 2 3 2 1 4]; b = [1 2 3 2 3];
[h,w] = freqs(b,a,64);
[bb,aa] = invfreqs(h,w,4,5)

bb =

    1.0000    2.0000    3.0000    2.0000    3.0000

aa =

    1.0000    2.0000    3.0000    2.0000    1.0000    4.0000
```

Notice that `bb` and `aa` are equivalent to `b` and `a`, respectively. However, `aa` has poles in the left half-plane and thus the system is unstable. Use `invfreqs`'s iterative algorithm to find a stable approximation to the system:

```
[bbb,aaa] = invfreqs(h,w,4,5,[],30)

bbb =

    0.6602    2.0058    2.3589    1.0837   –0.1337

aaa =

    1.0000    3.2986    7.0223    5.7543    2.9219    0.0002
```

Suppose you have two vectors, mag and phase, that contain magnitude and phase data gathered in a laboratory, and a third vector w of frequencies. You can convert the data into a continuous-time transfer function using invfreqs:

```
[b,a] = invfreqs(mag.*exp(j*phase),w,2,3);
```

Algorithm

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

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

by creating a system of linear equations and solving them with MATLAB's \ operator. Here $A(w(k))$ and $B(w(k))$ are the Fourier transforms of the polynomials a and b, respectively, at the frequency $w(k)$, and n is the number of frequency points (the length of h and w). This algorithm is based on Levi [1]. Several variants have been suggested in the literature, where the weighting function wt gives less attention to high frequencies.

The superior ("output-error") algorithm uses the damped Gauss-Newton method for iterative search [2], with the output of the first algorithm as the initial estimate. This solves the direct problem of minimizing the weighted sum of the squared error between the actual and the desired frequency response points:

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

See Also

freqs	Frequency response of analog filters.
freqz	Frequency response of digital filters.
invfreqz	Discrete-time filter identification from frequency data.
prony	Prony's method for time domain IIR filter design.

References

[1] Levi, E.C. "Complex-Curve Fitting." *IRE Trans. on Automatic Control*. Vol. AC-4 (1959). Pgs. 37-44.

[2] Dennis, J.E., Jr., and R.B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, NJ: Prentice Hall, 1983.

invfreqz

Purpose

Discrete-time filter identification from frequency data.

Syntax

```
[b,a] = invfreqz(h,w,nb,na)
[b,a] = invfreqz(h,w,nb,na,wt)
[b,a] = invfreqz(h,w,nb,na,wt,iter)
[b,a] = invfreqz(h,w,nb,na,wt,iter,tol)
[b,a] = invfreqz(h,w,nb,na,wt,iter,tol,'trace')
```

Description

`invfreqz` is the inverse operation of `freqz`; it finds a discrete-time transfer function that corresponds to a given complex frequency response. From a laboratory analysis standpoint, `invfreqz` can be used to convert magnitude and phase data into transfer functions.

`[b,a] = invfreqz(h,w,nb,na)` returns the real numerator and denominator coefficients in vectors `b` and `a` of the transfer function

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

whose complex frequency response is given in vector `h` at the frequency points specified in vector `w`. Scalars `nb` and `na` specify the desired orders of the numerator and denominator polynomials.

Frequency is specified in radians between 0 and π, and the length of `h` must be the same as the length of `w`.

`[b,a] = invfreqz(h,w,nb,na,wt)` weights the fit-errors versus frequency. `wt` is a vector of weighting factors the same length as `w`.

`invfreqz(h,w,nb,na,wt,iter)` and

`invfreqz(h,w,nb,na,wt,iter,tol)` provide a superior algorithm that guarantees stability of the resulting linear system and searches for the best fit using a numerical, iterative scheme. The `iter` parameter tells `invfreqz` to end the iteration when the solution has converged, or after `iter` iterations, whichever comes first. `invfreqz` defines convergence as occuring when the norm of

the (modified) gradient vector is less than `tol`. `tol` is an optional parameter that defaults to 0.01. To obtain a weight vector of all ones, use

```
invfreqz(h,w,nb,na,[],iter,tol)
```

`invfreqz(h,w,nb,na,wt,iter,tol,'trace')` displays a textual progress report of the iteration.

Example

Convert a simple transfer function to frequency response data and then back to the original filter coefficients:

```
a = [1 2 3 2 1 4]; b = [1 2 3 2 3];
[h,w] = freqz(b,a,64);
[bb,aa] = invfreqz(h,w,4,5)

bb =
    1.0000    2.0000    3.0000    2.0000    3.0000

aa =
    1.0000    2.0000    3.0000    2.0000    1.0000    4.0000
```

Notice that `bb` and `aa` are equivalent to `b` and `a`, respectively. However, `aa` has poles outside the unit circle and thus the system is unstable. Use `invfreqz`'s iterative algorithm to find a stable approximation to the system:

```
[bbb,aaa] = invfreqz(h,w,4,5,[],30)

bbb =
    0.2427    0.2788    0.0069    0.0971    0.1980

aaa =
    1.0000   –0.8944    0.6954    0.9997   –0.8933    0.6949
```

Algorithm

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

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

by creating a system of linear equations and solving them with MATLAB's \ operator. Here $A(w(k))$ and $B(w(k))$ are the Fourier transforms of the

polynomials a and b, respectively, at the frequency $w(k)$, and n is the number of frequency points (the length of h and w). This algorithm is a based on Levi [1].

The superior ("output-error") algorithm uses the damped Gauss-Newton method for iterative search [2], with the output of the first algorithm as the initial estimate. This solves the direct problem of minimizing the weighted sum of the squared error between the actual and the desired frequency response points:

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

See Also

freqs	Frequency response of analog filters.
freqz	Frequency response of digital filters.
invfreqs	Continuous-time (analog) filter identification from frequency data.
prony	Prony's method for time domain IIR filter design.

References

[1] Levi, E.C. "Complex-Curve Fitting." *IRE Trans. on Automatic Control.* Vol. AC-4 (1959). Pgs. 37-44.

[2] Dennis, J.E., Jr., and R.B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, NJ: Prentice Hall, 1983.

Purpose Kaiser window.

Syntax

```
w = kaiser(n,beta)
```

Description `w = kaiser(n,beta)` returns an `n`-point Kaiser (I_0 - sinh) window in the column vector `w`. `beta` is the Kaiser window β parameter that affects the sidelobe attenuation of the Fourier transform of the window.

To obtain a Kaiser window that designs an FIR filter with sidelobe height -α dB, use the following β:

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

Increasing `beta` widens the mainlobe and decreases the amplitude of the sidelobes (increases the attenuation).

See Also

`bartlett`	Bartlett window.
`blackman`	Blackman window.
`boxcar`	Rectangular window.
`chebwin`	Chebyshev window.
`hamming`	Hamming window.
`hanning`	Hanning window.
`kaiserord`	Estimate parameters for `fir1` with Kaiser window.
`triang`	Triangular window.

References

[1] Kaiser, J.F. “Nonrecursive Digital Filter Design Using the I_0 - sinh Window Function.” *Proc. 1974 IEEE Symp. Circuits and Syst.* (April 1974). Pgs. 20-23.

[2] IEEE. *Digital Signal Processing II.* IEEE Press. New York: John Wiley & Sons, 1975.

kaiserord

Purpose

Estimate parameters for `fir1` with Kaiser window.

Syntax

```
[n,Wn,beta,ftype] = kaiserord(f,a,dev)
[n,Wn,beta,ftype] = kaiserord(f,a,dev,Fs)
c = kaiserord(f,a,dev,Fs,'cell')
```

Description

`kaiserord` returns a filter order `n` and `beta` parameter to specify a Kaiser window for use with the `fir1` function. Given a set of specifications in the frequency domain, `kaiserord` estimates the minimum FIR filter order that will approximately meet the specifications. `kaiserord` converts the given filter specifications into passband and stopband ripples and converts cutoff frequencies into the form needed for windowed FIR filter design.

NOTE If the band ripples are specified as unequal, the smallest one is used, since the Kaiser window method is constrained to give filters with equal ripple heights in all the passbands and stopbands.

`[n,Wn,beta,ftype] = kaiserord(f,a,dev)` finds the approximate order `n`, normalized frequency band edges `Wn`, and weights that meet input specifications `f`, `a`, and `dev`. `f` is a vector of band edges and `a` is a vector specifying the desired amplitude on the bands defined by `f`. The length of `f` is twice the length of `a`, minus 2. Together, `f` and `a` define a desired piecewise constant response function. `dev` is a vector the same size as `a` that specifies the maximum allowable error or deviation between the frequency response of the output filter and its desired amplitude, for each band.

`fir1` can use the resulting order `n`, frequency vector `Wn`, multiband magnitude type `ftype`, and the Kaiser window parameter `beta`. The `ftype` string is intended for use with `fir1`; it is equal to `'high'` for a highpass filter and `'stop'` for a bandstop filter. For multiband filters, it can be equal to `'dc-0'` when the first band is a stopband (starting at $f = 0$) or `'dc-1'` when the first band is a passband.

To design a filter `b` that approximately meets the specifications given by `kaiser` parameters `f`, `a`, and `dev`:

```
b = fir1(n,Wn,kaiser(n+1,beta),ftype,'noscale')
```

`[n,Wn,beta,ftype] = kaiserord(f,a,dev,Fs)` specifies a sampling frequency `Fs`. If not present, `Fs` defaults to 2 Hz, implying a Nyquist frequency of 1 Hz. You can therefore specify band edges scaled to a particular application's sampling frequency.

`c = kaiserord(f,a,dev,Fs,'cell')` is a cell-array whose elements are the parameters to `fir1`.

NOTE In some cases, `kaiserord` underestimates or overestimates the order `n`. If the filter does not meet the specifications, try a higher order such as `n+1`, `n+2`, and so on, or a lower order.

NOTE Results are inaccurate if the cutoff frequencies are near 0 or the Nyquist frequency or if `dev` is large (greater than 10%).

Algorithm

`kaiserord` uses empirically derived formulas for estimating the orders of lowpass filters, as well as differentiators and Hilbert transformers. Estimates for multiband filters (such as bandpass filters) are derived from the lowpass design formulas.

The design formulas that underlie the Kaiser window and its application to FIR filter design are

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

where $\alpha = -20\log_{10}\delta$ is the stopband attenuation expressed in decibels (recall that $\delta_p = \delta_s$ is required). The design formula is:

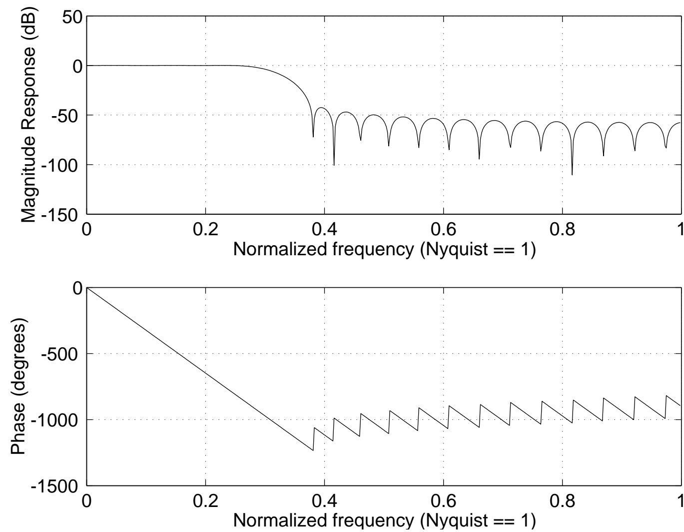
$$n = \frac{\alpha - 8}{2.285\,\Delta\omega}$$

where n is the filter order.

Examples

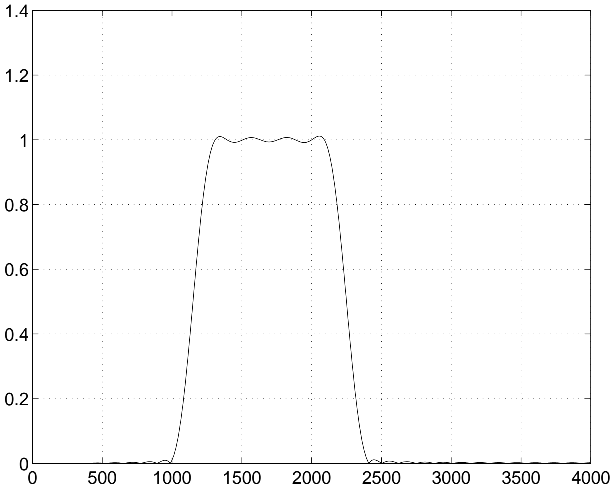
Design a lowpass filter with passband from 0 to 1 kHz and stopband from 1500 Hz to 4 kHz. Specify passband ripple of 5% and stopband attenuation of 40 dB:

```
fsamp = 8000;
fcuts = [1000 1500];
mags = [1 0];
devs = [0.05 0.01];
[n,Wn,beta,ftype] = kaiserord(fcuts,mags,devs,fsamp);
hh = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
freqz(hh)
```



Design an odd-length bandpass filter (note that odd length means even order, so the input to `fir1` must be an even integer):

```
fsamp = 8000;
fcuts = [1000 1300 2410];
mags = [0 1 0];
devs = [0.01 0.05 0.01];
[n,Wn,beta,ftype] = kaiserord(fcuts,mags,devs,fsamp);
n = n + rem(n,2);
hh = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
[H,f] = freqz(hh,1,1024,fsamp);
plot(f,abs(H)),grid on
```



Design a lowpass filter with a passband cutoff of 1500 Hz, a stopband cutoff of 2000 Hz, passband ripple of 0.01, stopband ripple of 0.1, and a sampling frequency of 8000 Hz:

```
[n,Wn,beta,ftype] = kaiserord([1500 2000],[1 0],[0.01 0.1],8000);
b = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
```

This is equivalent to

```
c = kaiserord([1500 2000],[1 0],[0.01 0.1],8000,'cell');
b = fir1(c{:});
```

Remarks

Be careful to distinguish between the meanings of filter length and filter order. The filter *length* is the number of impulse response samples in the FIR filter. Generally, the impulse response is indexed from $n = 0$ to $n = L-1$, where L is the filter length. The filter *order* is the highest power in a z-transform representation of the filter. For an FIR transfer function, this representation is a polynomial in z, where the highest power is z^{L-1} and the lowest power is z^0. The filter order is 1 less than the length (L-1) and is also equal to the number of zeros of the z polynomial.

Diagnostics

The lengths of the frequency, magnitude, and deviation vectors must match; otherwise, `kaiserord` gives the following error message:

```
Requires M and DEV to be the same length.
Length of F must be 2*length(M)-2.
```

When magnitudes are not specified correctly, `kaiserord` gives the following error messages:

```
Stopbands must be zero.
All passbands must have the same height.
```

If the band edges are not strictly increasing, `kaiserord` gives the following error message:

```
Bandedges must be strictly increasing.
```

See Also

`fir1`	Window-based finite impulse response filter design—standard response.
`kaiser`	Kaiser window.
`remezord`	Parks-McClellan optimal FIR filter order estimation.

References

[1] Kaiser, J.F. "Nonrecursive Digital Filter Design Using the I_0 - sinh Window Function." P*roc. 1974 IEEE Symp. Circuits and Syst.* (April 1974). Pgs. 20-23.

[2] IEEE. *Digital Signal Processing II*. IEEE Press. New York: John Wiley & Sons, 1975. Pgs. 123-126.

Purpose Lattice filter to transfer function conversion.

Syntax

```
[num,den] = latc2tf(k,v)
[num,den] = latc2tf(k,'iir')
num = latc2tf(k,'fir')
num = latc2tf(k)
```

Description

`[num,den] = latc2tf(k,v)` finds the transfer function numerator `num` and denominator `den` from the IIR lattice coefficients `k` and ladder coefficients `v`.

`[num,den] = latc2tf(k,'iir')` assumes that `k` is associated with an all-pole IIR lattice filter.

`num = latc2tf(k,'fir')` and

`num = latc2tf(k)` find the transfer function numerators from the FIR lattice coefficients specified by `k`.

See Also

`latcfilt`	Lattice and lattice-ladder filter implementation.
`tf2latc`	Transfer function to lattice filter conversion.

latcfilt

Purpose Lattice and lattice-ladder filter implementation.

Syntax

```
[f,g] = latcfilt(k,x)
[f,g] = latcfilt(k,v,x)
[f,g] = latcfilt(k,1,x)
```

Description

`[f,g] = latcfilt(k,x)` filters `x` with the FIR lattice coefficients in vector `k`. `f` is the forward lattice filter result and `g` is the backward filter result.

If `k` and `x` are vectors, the result is a (signal) vector.

Matrix arguments are permitted under the following rules:

- If `x` is a matrix and `k` is a vector, each column of `x` is processed through the lattice filter specified by `k`.
- If `x` is a vector and `k` is a matrix, each column of `k` is used to filter `x`, and a signal matrix is returned.
- If `x` and `k` are both matrices with the same number of columns, then the *i*-th column of `k` is used to filter the *i*-th column of `x`. A signal matrix is returned.

`[f,g] = latcfilt(k,v,x)` filters `x` with the IIR lattice coefficients `k` and ladder coefficients `v`. `k` and `v` must be vectors, while `x` may be a signal matrix.

`[f,g] = latcfilt(k,1,x)` filters `x` with the IIR all-pole lattice specified by `k`. `k` and `x` may be vectors or matrices according to the rules given for the FIR lattice.

See Also

`filter`	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.
`latc2tf`	Lattice filter to transfer function conversion.
`tf2latc`	Transfer function to lattice filter conversion.

Purpose

Levinson-Durbin recursion.

Syntax

```
a = levinson(r,n)
```

Description

The Levinson-Durbin recursion is an algorithm for finding an all-pole IIR filter with a prescribed deterministic autocorrelation sequence. It has applications in filter design, coding, and spectral estimation. The filter that `levinson` produces is minimum phase.

`a = levinson(r,n)` finds the coefficients of an nth-order autoregressive linear process which has `r` as its autocorrelation sequence. `r` is a real deterministic autocorrelation sequence (a vector), and `n` is the order of denominator polynomial $a(z)$, that is, `a = [1 a(2) ... a(n+1)]`. The filter coefficients are ordered in descending powers of z:

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

Algorithm

`levinson` solves the symmetric Toeplitz system of linear equations:

$$\begin{bmatrix} R(1) & R(2) & \cdots & R(n) \\ R(2) & R(1) & \cdots & R(n-1) \\ \vdots & \vdots & \ddots & \vdots \\ R(n) & R(n-1) & \cdots & R(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(n+1) \end{bmatrix} = \begin{bmatrix} -R(2) \\ -R(3) \\ \vdots \\ -R(n+1) \end{bmatrix}$$

where `r = [`$R(1) \ldots R(n+1)$`]` is the input autocorrelation vector. The algorithm requires $O(n^2)$ flops and is thus much more efficient than the MATLAB `\` command for large `n`. However, the `levinson` function uses `\` for low orders to give the fastest possible execution.

See Also

`lpc`	Linear prediction coefficients.
`prony`	Prony's method for time domain IIR filter design.
`stmcb`	Linear model using Steiglitz-McBride iteration.

References

[1] Ljung, L. *System Identification: Theory for the User*. Englewood Cliffs, NJ: Prentice Hall, 1987. Pgs. 278-280.

lp2bp

Purpose Lowpass to bandpass analog filter transformation.

Syntax

```
[bt,at] = lp2bp(b,a,Wo,Bw)
[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw)
```

Description `lp2bp` transforms analog lowpass filter prototypes with a cutoff frequency of 1 rad/sec into bandpass filters with desired bandwidth and center frequency. The transformation is one step in the digital filter design process for the `butter`, `cheby1`, `cheby2`, and `ellip` functions.

`lp2bp` can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

Transfer Function Form (Polynomial)

`[bt,at] = lp2bp(b,a,Wo,Bw)` transforms an analog lowpass filter prototype given by polynomial coefficients into a bandpass filter with center frequency `Wo` and bandwidth `Bw`. Row vectors `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of s:

$$\frac{b(s)}{a(s)} = \frac{b(1)s^{nn} + \cdots + b(nn)s + b(nn+1)}{a(1)s^{nd} + \cdots + a(nd)s + a(nd+1)}$$

Scalars `Wo` and `Bw` specify the center frequency and bandwidth in units of radians/second. For a filter with lower band edge `w1` and upper band edge `w2`, use `Wo = sqrt(w1*w2)` and `Bw = w2-w1`.

`lp2bp` returns the frequency transformed filter in row vectors `bt` and `at`.

State-Space Form

`[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw)` converts the continuous-time state-space lowpass filter prototype in matrices `A`, `B`, `C`, `D` :

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

into a bandpass filter with center frequency `Wo` and bandwidth `Bw`. For a filter with lower band edge `w1` and upper band edge `w2`, use `Wo = sqrt(w1*w2)` and `Bw = w2-w1`.

The bandpass filter is returned in matrices `At`, `Bt`, `Ct`, `Dt`.

Algorithm

`lp2bp` is a highly accurate state-space formulation of the classic analog filter frequency transformation. Consider the state-space system:

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

where u is the input, x is the state vector, and y is the output. The Laplace transform of the first equation is

$$sx = Ax + Bu$$

Now if a bandpass filter is to have center frequency ω_0 and bandwidth B_w, the standard s-domain transformation is

$$s = Q(p^2 + 1)/p$$

where $Q = \omega_0/B_w$ and $p = s/\omega_0$. Substituting this for s in the Laplace transformed state-space equation, and considering the operator p as d/dt:

$$Q\ddot{x} + Qx = A\dot{x} + B\dot{u}$$

or

$$Q\ddot{x} - A\dot{x} - B\dot{u} = -Qx$$

Now define

$$Q\dot{\omega} = -Qx$$

which, when substituted, leads to

$$Q\dot{x} = Ax + Q\omega + Bu$$

The last two equations give equations of state. Write them in standard form and multiply the differential equations by ω_0 to recover the time/frequency scaling represented by p and find state matrices for the bandpass filter:

```
Q = Wo/Bw; [ma,na] = size(A);
At = Wo*[A/Q eye(ma,na);-eye(ma,na) zeros(ma,na)];
Bt = Wo*[B/Q; zeros(ma,nb)];
Ct = [C zeros(mc,ma)];
Dt = d;
```

If the input to lp2bp is in transfer function form, the function transforms it into state-space form before applying this algorithm.

See Also

bilinear	Map variables using bilinear transformation.
impinvar	Impulse invariance method of analog-to-digital filter conversion.
lp2bs	Lowpass to bandstop analog filter transformation.
lp2hp	Lowpass to highpass analog filter transformation.
lp2lp	Lowpass to lowpass analog filter transformation.

Purpose

Lowpass to bandstop analog filter transformation.

Syntax

```
[bt,at] = lp2bs(b,a,Wo,Bw)
[At,Bt,Ct,Dt] = lp2bs(A,B,C,D,Wo,Bw)
```

Description

`lp2bs` transforms analog lowpass filter prototypes with a cutoff frequency of 1 rad/sec into bandstop filters with desired bandwidth and center frequency. The transformation is one step in the digital filter design process for the `butter`, `cheby1`, `cheby2`, and `ellip` functions.

`lp2bs` can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

Transfer Function Form (Polynomial)

`[bt,at] = lp2bs(b,a,Wo,Bw)` transforms an analog lowpass filter prototype given by polynomial coefficients into a bandstop filter with center frequency `Wo` and bandwidth `Bw`. Row vectors `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of *s*:

$$\frac{b(s)}{a(s)} = \frac{b(1)s^{nn} + \cdots + b(nn)s + b(nn+1)}{a(1)s^{nd} + \cdots + a(nd)s + a(nd+1)}$$

Scalars `Wo` and `Bw` specify the center frequency and bandwidth in units of radians/second. For a filter with lower band edge `w1` and upper band edge `w2`, use `Wo = sqrt(w1*w2)` and `Bw = w2-w1`.

`lp2bs` returns the frequency transformed filter in row vectors `bt` and `at`.

State-Space Form

`[At,Bt,Ct,Dt] = lp2bs(A,B,C,D,Wo,Bw)` converts the continuous-time state-space lowpass filter prototype in matrices `A`, `B`, `C`, `D`:

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

into a bandstop filter with center frequency `Wo` and bandwidth `Bw`. For a filter with lower band edge `w1` and upper band edge `w2`, use `Wo = sqrt(w1*w2)` and `Bw = w2-w1`.

The bandstop filter is returned in matrices `At`, `Bt`, `Ct`, `Dt`.

Algorithm

`lp2bs` is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a bandstop filter is to have center frequency ω_0 and bandwidth B_w, the standard s-domain transformation is

$$s = \frac{p}{Q(p^2+1)}$$

where $Q = \omega_0/B_w$ and $p = s/\omega_0$. The state-space version of this transformation is

```
Q = Wo/Bw;
At = [Wo/Q*inv(A) Wo*eye(ma);-Wo*eye(ma) zeros(ma)];
Bt = -[Wo/Q*(A B); zeros(ma,nb)];
Ct = [C/A zeros(mc,ma)];
Dt = D - C/A*B;
```

See `lp2bp` for a derivation of the bandpass version of this transformation.

See Also

`bilinear`	Map variables using bilinear transformation.
`impinvar`	Impulse invariance method of analog-to-digital filter conversion.
`lp2bp`	Lowpass to bandpass analog filter transformation.
`lp2hp`	Lowpass to highpass analog filter transformation.
`lp2lp`	Lowpass to lowpass analog filter transformation.

Purpose

Lowpass to highpass analog filter transformation.

Syntax

```
[bt,at] = lp2hp(b,a,Wo)
[At,Bt,Ct,Dt] = lp2hp(A,B,C,D,Wo)
```

Description

`lp2hp` transforms analog lowpass filter prototypes with a cutoff frequency of 1 rad/sec into highpass filters with desired cutoff frequency. The transformation is one step in the digital filter design process for the `butter`, `cheby1`, `cheby2`, and `ellip` functions.

The `lp2hp` function can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

Transfer Function Form (Polynomial)

`[bt,at] = lp2hp(b,a,Wo)` transforms an analog lowpass filter prototype given by polynomial coefficients into a highpass filter with cutoff frequency `Wo`. Row vectors `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of *s*:

$$\frac{b(s)}{a(s)} = \frac{b(1)s^{nn} + \cdots + b(nn)s + b(nn+1)}{a(1)s^{nd} + \cdots + a(nd)s + a(nd+1)}$$

Scalar `Wo` specifies the cutoff frequency in units of radians/second. The frequency transformed filter is returned in row vectors `bt` and `at`.

State-Space Form

`[At,Bt,Ct,Dt] = lp2hp(A,B,C,D,Wo)` converts the continuous-time state-space lowpass filter prototype in matrices `A`, `B`, `C`, `D`:

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

into a highpass filter with cutoff frequency `Wo`. The highpass filter is returned in matrices `At`, `Bt`, `Ct`, `Dt`.

Algorithm

`lp2hp` is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a highpass filter is to have cutoff frequency ω_0, the standard s-domain transformation is

$$s = \frac{\omega_0}{p}$$

The state-space version of this transformation is

```
At = Wo*inv(A);
Bt = -Wo*(A\B);
Ct = C/A;
Dt = D - C/A*B;
```

See `lp2bp` for a derivation of the bandpass version of this transformation.

See Also

`bilinear`	Map variables using bilinear transformation.
`impinvar`	Impulse invariance method of analog-to-digital filter conversion.
`lp2bp`	Lowpass to bandpass analog filter transformation.
`lp2bs`	Lowpass to bandstop analog filter transformation.
`lp2lp`	Lowpass to lowpass analog filter transformation.

Purpose

Lowpass to lowpass analog filter transformation.

Syntax

```
[bt,at] = lp2lp(b,a,Wo)
[At,Bt,Ct,Dt] = lp2lp(A,B,C,D,Wo)
```

Description

`lp2lp` transforms an analog lowpass filter prototype with a cutoff frequency of 1 rad/sec into a lowpass filter with any specified cutoff frequency. The transformation is one step in the digital filter design process for the `butter`, `cheby1`, `cheby2`, and `ellip` functions.

The `lp2lp` function can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

Transfer Function Form (Polynomial)

`[bt,at] = lp2lp(b,a,Wo)` transforms an analog lowpass filter prototype given by polynomial coefficients into a lowpass filter with cutoff frequency `Wo`. Row vectors `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of s:

$$\frac{b(s)}{a(s)} = \frac{b(1)s^{nn} + \cdots + b(nn)s + b(nn+1)}{a(1)s^{nd} + \cdots + a(nd)s + a(nd+1)}$$

Scalar `Wo` specifies the cutoff frequency in units of radians/second. `lp2lp` returns the frequency transformed filter in row vectors `bt` and `at`.

State-Space Form

`[At,Bt,Ct,Dt] = lp2lp(A,B,C,D,Wo)` converts the continuous-time state-space lowpass filter prototype in matrices `A`, `B`, `C`, `D`:

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

into a lowpass filter with cutoff frequency `Wo`. `lp2lp` returns the lowpass filter in matrices `At`, `Bt`, `Ct`, `Dt`.

Algorithm

lp2lp is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a lowpass filter is to have cutoff frequency ω_0, the standard *s*-domain transformation is

$$s = p / \omega_o$$

The state-space version of this transformation is

```
At = Wo*A;
Bt = Wo*B;
Ct = C;
Dt = D;
```

See lp2bp for a derivation of the bandpass version of this transformation.

See Also

bilinear	Map variables using bilinear transformation.
impinvar	Impulse invariance method of analog-to-digital filter conversion.
lp2bp	Lowpass to bandpass analog filter transformation.
lp2bs	Lowpass to bandstop analog filter transformation.
lp2hp	Lowpass to highpass analog filter transformation.

Purpose Linear prediction coefficients.

Syntax

```
[a,g] = lpc(x,n)
```

Description Linear prediction models each sample of a signal as a linear combination of previous samples, that is, as the output of an all-pole IIR filter. It has applications in filter design, speech coding, spectral analysis, and system identification.

`[a,g] = lpc(x,n)` finds the coefficients and gain of an `n`th-order auto-regressive linear process that models the time series `x` as

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

`x` is the real input time series (a vector), and `n` is the order of the denominator polynomial *a(z)*, that is, `a = [1 a(2) ... a(n+1)]`. The filter coefficients are ordered in descending powers of *z*.

If `n` is unspecified, `lpc` uses as a default `n = length(x)-1`.

If `x` is a matrix containing a separate signal in each column, `lpc` returns a model estimate for each column in the rows of `a` and a row vector of gains `g`.

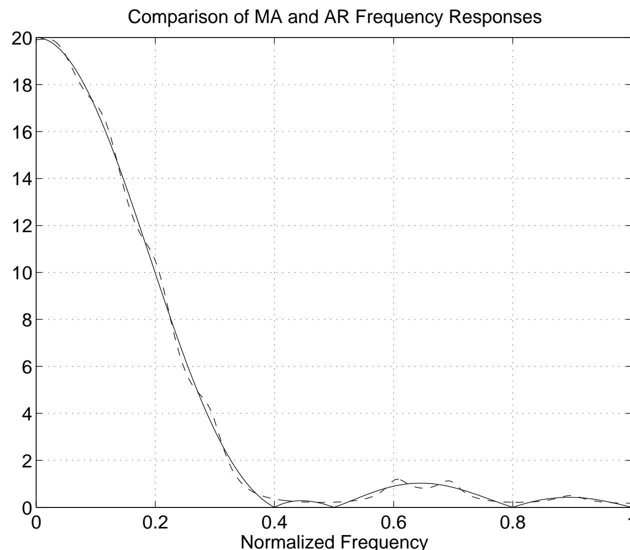
Example

Model a nonrecursive (FIR) filter with an all-pole IIR filter using lpc:

```
x = [1:4 4:-1:1];
[a,g] = lpc(x,15);
[H,w] = freqz(x,1,512); [H1,w] = freqz(g,a,512);
```

Plot the FIR response with a solid line and the IIR response with a dashed line:

```
plot(w/pi,abs(H),w/pi,abs(H1),'--')
```



Algorithm

lpc uses the autocorrelation method of autoregressive (AR) modeling to find the filter coefficients. This technique is also called the maximum entropy method (MEM) of spectral estimation. The filter generated is stable. However, the generated filter might not model the process exactly even if the data sequence is truly an AR process of the correct order. This is because the autocorrelation method implicitly windows the data, that is, it assumes that signal samples beyond the length of x are 0.

`lpc` solves the following system of equations using the equivalent of MATLAB's \ operator:

$$\begin{bmatrix} 0 & 0 & \cdots & 0 \\ x(1) & 0 & \cdots & 0 \\ x(2) & x(1) & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ x(l) & x(l-1) & \ddots & x(1) \\ 0 & x(l) & \ddots & \vdots \\ 0 & 0 & \cdots & x(l-1) \\ 0 & 0 & \cdots & x(l) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(n+1) \end{bmatrix} = \begin{bmatrix} -x(1) \\ -x(2) \\ \vdots \\ -x(l) \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

It first forms the deterministic autocorrelation of `x` with `xcorr(x)` and then calls `levinson` with the resulting autocorrelation as input.

See Also

`ar`	Compute autoregressive models of signals (see *System Identification Toolbox User's Guide*).
`levinson`	Levinson-Durbin recursion.
`pmem`	Power spectrum estimate using maximum entropy method (MEM).
`prony`	Prony's method for time domain IIR filter design.
`stmcb`	Linear model using Steiglitz-McBride iteration.

References

[1] Jackson, L.B. *Digital Filters and Signal Processing*. Second Ed. Boston: Kluwer Academic Publishers, 1989. Pgs. 255-257.

maxflat

Purpose

Generalized digital Butterworth filter design.

Syntax

```
[b,a,] = maxflat(nb,na,Wn)
b = maxflat(nb,'sym',Wn)
[b,a,b1,b2] = maxflat(nb,na,Wn)
[...] = maxflat(nb,na,Wn,'design_flag')
```

Description

`[b,a,] = maxflat(nb,na,Wn)` is a lowpass Butterworth filter with numerator and denominator coefficients `b` and `a` of orders `nb` and `na` respectively. `Wn` is the cutoff frequency at which the magnitude response of the filter is equal to $1/\sqrt{2}$ (approx. -3 dB). `Wn` must be between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency).

`b = maxflat(nb,'sym',Wn)` is a symmetric FIR Butterworth filter. `nb` must be even, and `Wn` is restricted to a subinterval of [0,1]. The function raises an error if `Wn` is specified outside of this subinterval.

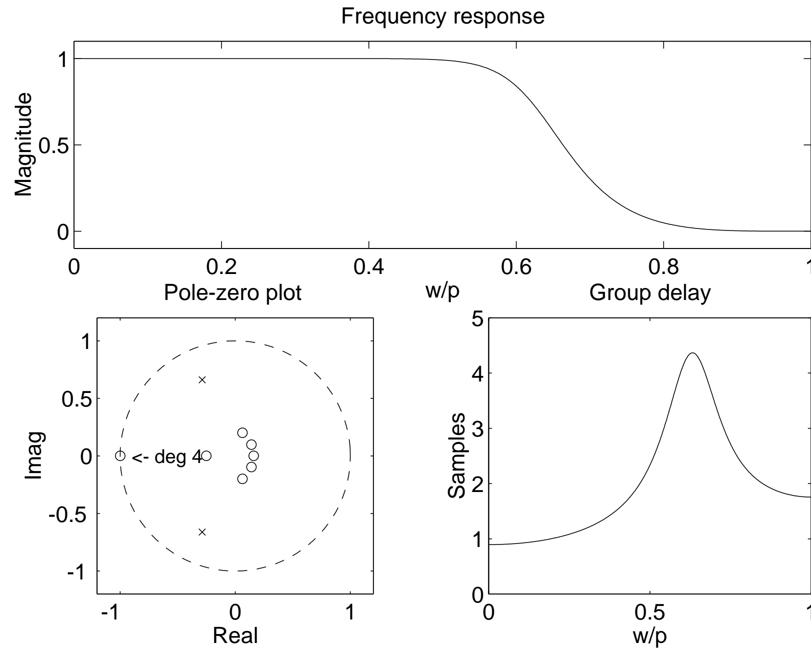
`[b,a,b1,b2] = maxflat(nb,na,Wn)` returns two polynomials `b1` and `b2` whose product is equal to the numerator polynomial `b` (that is, `b = conv(b1,b2)`). `b1` contains all the zeros at $z = -1$, and `b2` contains all the other zeros.

`[...] = maxflat(nb,na,Wn,'design_flag')` enables you to monitor the filter design, where *design_flag* is

- `trace`, for a textual display of the design table used in the design
- `plots`, for plots of the filter's magnitude, group delay, and zeros and poles
- `both`, for both the textual display and plots

Examples

```
nb = 10; na = 2; Wn = 0.2*pi;
[b,a,b1,b2] = maxflat(nb,na,Wn,'plots')
```



Algorithm

The method consists of the use of formulae, polynomial root finding, and a transformation of polynomial roots.

See Also

`butter`	Butterworth analog and digital filter design.
`filter`	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.
`freqz`	Frequency response of digital filters.

References

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

medfilt1

Purpose

One-dimensional median filtering.

Syntax

```
y = medfilt1(x,n)
y = medfilt1(x,n,blksz)
```

Description

`y = medfilt1(x,n)` applies an order `n`, one-dimensional median filter to vector `x`. `y` is the same length as `x`; the function treats the signal as if it is 0 beyond the end points.

For `n` odd, `y(k)` is the median of `x(k-(n-1)/2:k+(n-1)/2)`.

For n even, `y(k)` is the median of `x(k-n/2)`,`x(k-(n/2)+1)`,...,`x(k+(n/2)-1)`. In this case, `medfilt1` sorts the numbers, then takes the average of the `(n-1)/2` and `((n-1)/2)+1` elements.

The default for `n` is 3.

`y = medfilt1(x,n,blksz)` uses a `for`-loop to compute `blksz` (block size) output samples at a time. Use `blksz << length(x)` if you are low on memory, since `medfilt1` uses a working matrix of size `n`-by-`blksz`. By default, `blksz = length(x)`; this is the fastest execution if you have sufficient memory.

If `x` is a matrix, `medfilt1` median filters its columns using

```
y(:,i) = medfilt1(x(:,i),n,blksz)
```

in a loop over the columns of `x`.

See Also

`filter`	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.
`medfilt2`	Two-dimensional median filtering (see *Image Processing Toolbox User's Guide*).
`median`	Median value (see MATLAB Function Reference).

References

[1] Pratt,W.K. *Digital Image Processing*. New York: John Wiley & Sons, 1978. Pgs. 330-333.

Purpose

Modulation for communications simulation.

Syntax

```
y = modulate(x,Fc,Fs,'method')
y = modulate(x,Fc,Fs,'method',opt)
[y,t] = modulate(x,Fc,Fs)
```

Description

`y = modulate(x,Fc,Fs,'method')` and

`y = modulate(x,Fc,Fs,'method',opt)` modulate the real message signal `x` with a carrier frequency `Fc` and sampling frequency `Fs`, using one of the options listed below for *`method`*. Note that some methods accept an option, `opt`.

`amdsb-sc` or `am` — **Amplitude modulation, double side-band, suppressed carrier.** Multiplies `x` by a sinusoid of frequency `Fc`:

```
y = x.*cos(2*pi*Fc*t)
```

`amdsb-tc` — **Amplitude modulation, double side-band, transmitted carrier.** Subtracts scalar `opt` from `x` and multiplies the result by a sinusoid of frequency `Fc`:

```
y = (x-opt).*cos(2*pi*Fc*t)
```

If the `opt` parameter is not present, `modulate` uses a default of `min(min(x))` so that the message signal `(x-opt)` is entirely non-negative and has a minimum value of 0.

`amssb` — **Amplitude modulation, single side-band.** Multiplies `x` by a sinusoid of frequency `Fc` and adds the result to the Hilbert transform of `x` multiplied by a phase shifted sinusoid of frequency `Fc`:

```
y = x.*cos(2*pi*Fc*t)+imag(hilbert(x)).*sin(2*pi*Fc*t)
```

This effectively removes the upper sideband.

`fm` **Frequency modulation.** Creates a sinusoid with instantaneous frequency that varies with the message signal `x`:

```
y = cos(2*pi*Fc*t + opt*cumsum(x))
```

`cumsum` is a rectangular approximation to the integral of `x`. `modulate` uses `opt` as the constant of frequency modulation. If `opt` is not present, `modulate` uses a default of

```
opt = (Fc/Fs)*2*pi/(max(max(x)))
```

so the maximum frequency excursion from `Fc` is `Fc` Hz.

`pm` **Phase modulation.** Creates a sinusoid of frequency `Fc` whose phase varies with the message signal `x`:

```
y = cos(2*pi*Fc*t + opt*x)
```

`modulate` uses `opt` as the constant of phase modulation. If `opt` is not present, `modulate` uses a default of

```
opt = pi/(max(max(x)))
```

so the maximum phase excursion is π radians.

`pwm` **Pulse-width modulation.** Creates a pulse-width modulated signal from the pulse widths in `x`. The elements of `x` must be between 0 and 1, specifying the width of each pulse in fractions of a period. The pulses start at the beginning of each period, that is, they are left justified.

```
modulate(x,Fc,Fs,'pwm','centered')
```

yields pulses centered at the beginning of each period. `y` is length `length(x)*Fs/Fc`.

`ptm` **Pulse time modulation.** Creates a pulse time modulated signal from the pulse times in `x`. The elements of `x` must be between 0 and 1, specifying the left edge of each pulse in fractions of a period. `opt` is a scalar between 0 and 1 that specifies the length of each pulse in fractions of a period. The default for `opt` is 0.1. `y` is length `length(x)*Fs/Fc`.

`qam` **Quadrature amplitude modulation.** Creates a quadrature amplitude modulated signal from signals `x` and `opt`:

```
y = x.*cos(2*pi*Fc*t) + opt.*sin(2*pi*Fc*t)
```

`opt` must be the same size as `x`.

If you do not specify *method*, then modulate assumes am. Except for the pwm and ptm cases, y is the same size as x.

If x is an array, modulate modulates its columns.

[y,t] = modulate(x,Fc,Fs) returns the internal time vector t that modulate uses in its computations.

See Also

demod	Demodulation for communications simulation.
vco	Voltage controlled oscillator.

pmem

Purpose

Power spectrum estimate using maximum entropy method (MEM).

Syntax

```
[Pxx,freq] = pmem(x,p)
[Pxx,freq] = pmem(x,p,nfft,Fs,'corr')
[Pxx,freq,a] = pmem(x,p,nfft,Fs,'corr')
```

Description

`pmem` estimates the power spectral density (PSD) of the signal vector $x[n]$ or correlation matrix **R** using the maximum entropy method (MEM) described in [1]. It derives an all-pole model to represent the spectrum, so the correct choice of the model order `p` is crucial.

`[Pxx,freq] = pmem(x,p)` returns `Pxx`, the power spectrum estimate, and `freq`, a vector of frequencies at which the PSD was estimated. `x` is the input signal, or the input correlation matrix, where

- A row or column vector represents one signal
- A square, Hermitian symmetric matrix represents a correlation matrix (when `'corr'` is used)
- A rectangular array assumes that each column of `x` is a separate "look" at the signal (as in array processing)

`p` is the model order for the all-pole filter.

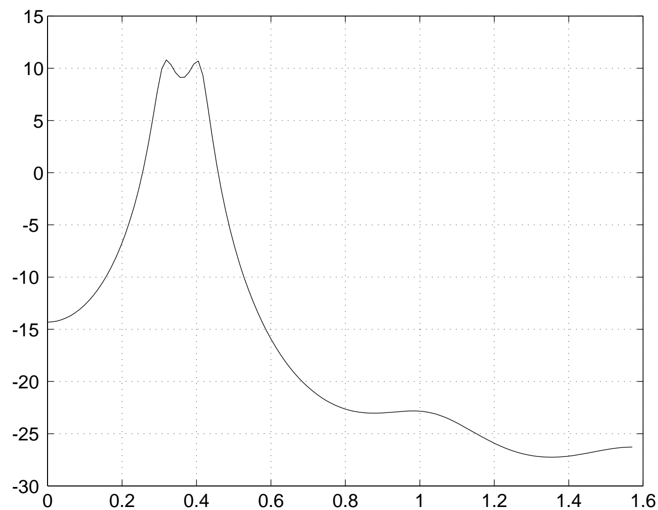
`[Pxx,freq] = pmem(x,p,nfft,Fs,'corr')` specifies the FFT length `nfft` (default is 256) and the sampling frequency for the signal `Fs` (default is 1). If `Fs` is specified, the output frequency vector `freq` is scaled by this value. If the input signal is real-valued, `freq` ranges from 0 to `Fs/2`. If the input signal is complex, `freq` ranges from 0 to `Fs`. `'corr'` is a text string to specify a correlation option. Specifying `'corr'` forces `x` to be taken as a correlation matrix. `'corr'` must appear at the end of the argument list.

`[Pxx,freq,a] = pmem(x,p,nfft,Fs,'corr')` returns vector `a` of filter coefficients for the all-pole filter model.

Examples

This example analyzes a sequence $x[n]$, assuming that two real signals are present in the signal subspace. In this case, the model order must be four or larger, because each real sinusoid is the sum of two complex exponentials. Experience shows that taking a larger model order than the minimum seems to work better.

```
% Create xx as a signal vector.

nn = 0:199;
randn('seed',0)
xx = cos(0.257*pi*nn) + sin(0.2*pi*nn) + 0.01*randn(size(nn));
[PP,ff,aa] = pmem(xx,7); % 7th order model

plot(ff*pi,10*log10(PP)) % Plot the pole locations.
```



The following examples use x as a correlation matrix and a data matrix:

```
% Assume that RR is a square corr. matrix (for example, 7 by 7).

RR = toeplitz(cos(0.1*pi*[0:6])) + 0.1*eye(7);
[PP,ff] = pmem(RR,4,'corr');
% Make a Look Matrix: xx is rectangular (100 by 7).
%
randn('seed',0)
xx = reshape(cos(0.257*pi*(0:699)),7,100) + 0.1*randn(7,100);
[PP,ff] = pmem(xx,4);

% Same (100 by 7) data matrix as before, but with a longer FFT.

[PP,ff] = pmem(xx,4,512);
```

Algorithm

The MEM estimate is given by the formula:

$$P_{mem}(f) = \frac{1}{\left|\mathbf{a}^H \mathbf{e}(f)\right|^2}$$

where the vector of all-pole filter coefficients **a** is the solution of the autocorrelation normal equation:

$$\mathbf{Ra} = \mathbf{r}$$

The matrix **R** is the autocorrelation matrix, which should be Toeplitz [1]. The elements of the vector **r** are also correlations.

Diagnostics

There must be at least one output argument and at least two inputs; otherwise, pmem stops and gives one of the following error messages:

```
Must have at least 1 output argument.
Must have at least 2 input arguments.
```

The first argument must be a full matrix, otherwise pmem gives the following error message:

```
Input signal or correlation cannot be sparse.
```

If you specify an empty matrix for the second argument, `pmem` gives the following error message:

```
Model order must be given, empty not allowed.
```

If the final argument is the string `'corr'`, then the first input must be a square correlation matrix that is also Hermitian symmetric, otherwise `pmem` gives the following error messages:

```
Correlation matrix (R) is not square.
Correlation matrix (R) is not Hermitian symmetric.
```

See Also

`lpc`	Linear prediction coefficients.
`pmtm`	Power spectrum estimate using the multitaper method (MTM).
`pmusic`	Power spectrum estimate using MUSIC eigenvector method.
`prony`	Prony's method for time domain IIR filter design.
`psd`	Estimate the power spectral density (PSD) of a signal.

References

[1] Marple, S.L. *Digital Spectral Analysis*. Englewood Cliffs, NJ: Prentice Hall, 1987. Chapter 7.

pmtm

Purpose

Power spectrum estimate using the multitaper method (MTM).

Syntax

```
Pxx = pmtm(x)
Pxx = pmtm(x,nw)
Pxx = pmtm(x,nw,nfft)
[Pxx,f] = pmtm(x,nw,nfft,Fs)
[Pxx,f] = pmtm(x,nw,nfft,Fs,'method')
[Pxx,Pxxc,f] = pmtm(x,nw,nfft,Fs,'method')
[Pxx,Pxxc,f] = pmtm(x,nw,nfft,Fs,'method',p)
[Pxx,Pxxc,f] = pmtm(x,e,v,nfft,Fs,'method',p)
[Pxx,Pxxc,f] = pmtm(x,dpss_params,nfft,Fs,'method',p)
```

Description

`pmtm` estimates the power spectral density (PSD) of the real time series *x* using the multitaper method (MTM), described in [1].

`Pxx = pmtm(x,nw)` estimates the PSD using `nw` as the time-bandwidth product for the discrete prolate spheroidal sequences (Slepian sequences) that are used as data windows. The default for `nw` is 4; other typical choices are 2, 5/2, 3, 7/2. The number of sequences used to form `Pxx` is `2*nw-1`.

`Pxx = pmtm(x,nw,nfft)` defines the frequency grid as length `nfft`. When `x` is real, `Pxx` is length `(nfft/2+1)` for `nfft` even and `(nfft+1)/2` for `nfft` odd; when `x` is complex, `Pxx` is length `nfft`. The default for `nfft` is 256 or the next power of 2 greater than the length of `x`, whichever is larger.

`[Pxx,f] = pmtm(x,nw,nfft,Fs)` returns `f`, the vector of frequencies at which the PSD is estimated, for the sampling frequency `Fs`. The default for `Fs` is 2 Hz.

`[Pxx,f] = pmtm(x,nw,nfft,Fs,'method')` specifies the algorithm used for combining the individual spectral estimates, where method is

- `adapt`, to specify Thomson's adaptive nonlinear combination (default)
- `unity`, to specify a linear combination with unity weights
- `eigen`, to specify a linear combination with eigenvalue weights

[Pxx,Pxxc,f] = pmtm(x,nw,nfft,Fs,'*method*') returns Pxxc, the 95% confidence interval for Pxx, and

[Pxx,Pxxc,f] = pmtm(x,nw,nfft,Fs,'*method*',p) returns Pxxc, the p*100% confidence interval for Pxx, where p is a scalar between 0 and 1. Confidence intervals are computed using a chi-squared approach, where Pxxc(:,1) is the lower bound and Pxxc(:,2) is the upper bound of the confidence interval.

[Pxx,Pxxc,f] = pmtm(x,e,v,nfft,Fs,'*method*',p) returns the PSD estimate Pxx, the confidence interval Pxxc, and the frequency vector f from the data tapers in e and their concentrations v.

[Pxx,Pxxc,f] = pmtm(x,dpss_params,nfft,Fs,'*method*',p) returns the PSD estimate Pxx, the confidence interval Pxxc, and the frequency vector f from the data tapers computed using dpss with parameters from the cell array dpss_params, starting with the second element of the array. The first parameter is set by the length of x. For example, pmtm(x,{3.5,'calc','trace'},512,Fs) forces direct calculation of the Slepian sequences. See dpss for options.

Remarks

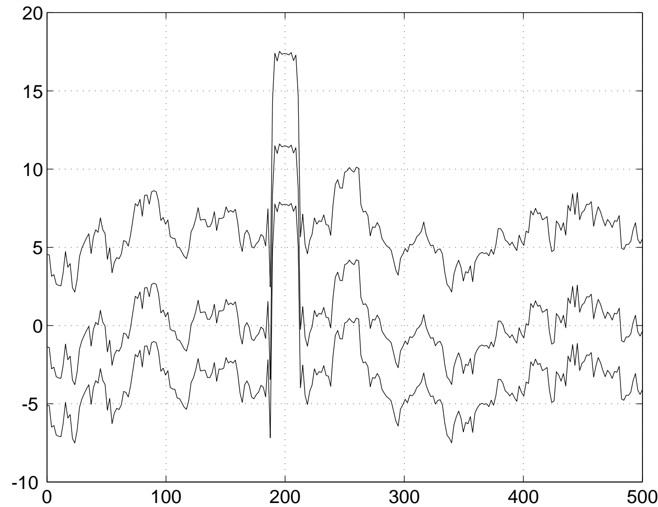
pmtm with no output arguments plots the PSD in the current or next available figure, with confidence intervals.

To use default parameters for any argument in an expression, insert an empty matrix []. For example, pmtm(x,[],[],1000) uses defaults for the second and third elements, in this case, nw and nfft.

Example

This example analyzes a sinusoid in white noise:

```
Fs = 1000;  t = 0:1/Fs:0.3;
x = cos(2*pi*t*200) + randn(size(t));
[Pxx,Pxxc,f] = pmtm(x,3.5,512,Fs,[],0.99);
plot(f,10*log10([Pxx Pxxc]))
```



See Also

`dpss`	Discrete prolate spheroidal sequences (Slepian sequences).
`pmem`	Power spectrum estimate using maximum entropy method (MEM).
`pmusic`	Power spectrum estimate using MUSIC eigenvector method.
`psd`	Estimate the power spectral density (PSD) of a signal.

References

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

[2] Thomson, D.J. “Spectrum estimation and harmonic analysis.” In *Proceedings of the IEEE*. Vol. 70 (1982). Pgs. 1055-1096.

Purpose

Power spectrum estimate using MUSIC eigenvector method.

Syntax

```
[Pxx,f] = pmusic(x,p)
[Pxx,f] = pmusic(x,[p thresh])
[Pxx,f] = pmusic(x,[p thresh],nfft,Fs,window,noverlap)
[Pxx,f] = pmusic(x,...,'corr')
[Pxx,f] = pmusic(x,...,'ev')
[Pxx,f,evects,svals] = pmusic(x,...)
```

Description

`pmusic` estimates the power spectral density (PSD) of a signal or correlation matrix using Schmidt's eigen-analysis method [1]. The name MUSIC is an acronym for MUltiple SIgnal Classification. The *eigenvector method*, which uses eigenvalue weighting, is also supported [2]. The calling syntax is similar to that of `psd`, which also performs spectrum estimation. `psd` uses the classical FFT-based approach while `pmusic` performs eigen-analysis of the signal's correlation matrix.

`[Pxx,f] = pmusic(x,p)` and

`[Pxx,f] = pmusic(x,[p thresh])` return `Pxx`, the power spectrum estimate, and `f`, a vector of frequencies at which the PSD is estimated. `x` is the input signal, where

- A row or column vector represents one observation of the process output (for example, one "signal")
- A rectangular (possibly square) array assumes that each column of `x` is a separate observation of the process output (for example, each column is one output of an array of sensors, as in array processing)
- A square matrix, given the trailing argument `'corr'`, represents a correlation matrix

The second argument is a one- or two-element vector, either `p` or `[p thresh]`. If only `p` is specified, the signal subspace dimension is `p`. If `[p thresh]` is specified, `thresh` is multiplied by λ_{min}, the smallest eigenvalue; eigenvalues below the threshold λ_{min}`*thresh` are assigned to the noise subspace. In this case, `p` is the maximum dimension of the signal subspace.

WARNING `pmusic` must assign eigenvectors to the noise and signal subspaces, but this is very difficult to do in practice. The two parameters `p` and `thresh` are provided for flexibility and control.

`[Pxx,f] = pmusic(x,[p thresh],nfft,Fs,window,noverlap)` specifies the FFT length `nfft` (default is 256) and the sampling frequency for the signal `Fs` (default is 2). If `Fs` is specified, the output frequency vector `f` is scaled by this value. If the input signal is real-valued, the frequency range is 0 to `Fs/2`; for the complex case, it is 0 to `Fs`. `window` is a scalar specifying the rectangular window length, or a vector giving the actual window coefficients. `noverlap`, used in conjunction with `window`, is a scalar that gives the number of points by which to overlap successive windows.

`[Pxx,f] = pmusic(x,...,'corr')` forces `x` to be taken as a correlation matrix. In this case, the arguments `window` and `noverlap` are ignored.

`[Pxx,f] = pmusic(x,...,'ev')` selects the eigenvector variant of the MUSIC estimator. See the "Algorithm" section below for an explanation of how this is different from the MUSIC method.

`[Pxx,f,evects,svals] = pmusic(x,...)` returns two additional arguments. `evects` is a matrix of eigenvectors spanning the noise subspace (one per column). `svals` is either a vector of singular values (squared) from `svd` or a vector of eigenvalues of the correlation matrix when the `'corr'` option is present.

Remarks

The input `x` can be a vector or a matrix. `x` can be interpreted as signal data or as a correlation matrix, in one of three ways:

- `x` is a vector of signal values (row or column). In this case, the dimension of the eigenvectors must be given. This is done either by taking the default value of `2*p` or by specifying a window length using `window`.
- `x` is a rectangular (`m`-by-`n`, possibly square) matrix. In this case, each column of `x` is a separate observation signal that enters into the SVD analysis, `n` is

the number of observations, and the dimension of the eigenvectors is equal to m, the length of a column.

- x is a square matrix and the trailing 'corr' is present. x is treated as a correlation matrix. In this case, the matrix must have only real, nonnegative eigenvalues.

The inputs p and thresh can determine the number of noise eigenvectors in one of three ways:

- If thresh < 1, or if it is unspecified, the number of eigenvectors spanning the signal subspace will be equal to p. p must be an integer satisfying 0 ≤ p < n, where n is the dimension of the eigenvectors. This dimension n is the column length in the data matrix case, the matrix size in the correlation matrix case, or the window length for signal vectors. The value of thresh is unused.
- If p ≥ n, thresh must be at least 1. thresh is used as the multiplier to determine an absolute threshold for splitting the eigenvalues between the signal and noise subspaces:

$$\lambda_k \le (\mathrm{thresh})\min\{\lambda_k\} \quad \Rightarrow \{\lambda_k, v_k\} \text{ belong to noise subspace}$$

  If thresh < 1, there will be no noise eigenvectors. This case is not allowed and gives the following error message:

  ```
  Noise subspace dimension cannot be zero.
  ```

- When p < n and thresh ≥ 1, p specifies the maximum number of signal eigenvectors. However, the threshold test specified by thresh can also take eigenvectors from the signal subspace and assign them to the noise subspace.

Examples

This example analyzes a signal vector xx, assuming that two real signals are present in the signal subspace. In this case, the dimension of the signal subspace is 4 because each real sinusoid is the sum of two complex exponentials:

```
nn = 0:199;
xx = cos(0.257*pi*nn) + sin(0.2*pi*nn) + 0.01*randn(size(nn));
[PP,ff] = pmusic(xx,4);
```

This example analyzes the same signal vector xx with an eigenvalue cutoff of 10% above the minimum. Setting p = Inf forces the signal/noise subspace deci-

sion to be based on thresh. Use eigenvectors of dimension 7 and a sampling frequency Fs of 8 kHz:

```
[PP,ff] = pmusic(xx,[Inf,1.1],[],8000,7); % window length = 7
```

With the third and fourth outputs, by plotting the zeros of the noise-eigenvector polynomials, it is possible to create a "Root-MUSIC" algorithm, as the following zplane plot illustrates:

```
[PP,ff,v_noise] = pmusic(xx,4);
for kk = 1:size(v_noise,2)
   rr(:,kk) = roots(v_noise(:,kk));
end
zplane(rr)
```

Assume that RR is a square correlation matrix (for example, 7-by-7):

```
RR = toeplitz(cos(0.1*pi*[0:6])) + 0.1*eye(7);
[PP,ff] = pmusic(RR,4,'corr');
```

Make an observation matrix xx that is rectangular (100-by-7):

```
xx = reshape(cos(0.257*pi*(0:699)),7,100) + 0.1*randn(7,100);
[PP,ff] = pmusic(xx,4);
```

Use the same signal, but let pmusic form the 100-by-7 data matrix using its window and overlap inputs. In addition, use a longer FFT:

```
yy = xx(:);
[PP,ff] = pmusic(yy,4,512,[],7,0);
```

If we set p = 0, all the eigenvectors are assigned to the noise subspace. 'ev' specifies the eigenvector weighting. This turns out to be equivalent to MVDL (Capon's MLM):

```
[PP,ff] = pmusic(RR,0,'ev','corr');
```

Algorithm

The MUSIC estimate is given by the formula

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

where N is the dimension of the eigenvectors and $\mathbf{v}_k$ is the k-th eigenvector of the correlation matrix of the input signal. The integer p is the dimension of the signal subspace, so the eigenvectors $\mathbf{v}_k$ used in the sum correspond to the smallest eigenvalues and also span the noise subspace. The vector $\mathbf{e}(f)$ consists of complex exponentials, so the inner product

$$\mathbf{v}_k^H \mathbf{e}(f)$$

amounts to a Fourier transform. The second form is preferred for computation because the FFT is computed for each $\mathbf{v}_k$ and then the squared magnitudes are summed.

In the eigenvector method, the summation is weighted by the eigenvalues λ_k of the correlation matrix:

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

The function relies on the `svd` matrix decomposition in the signal case, and it uses the `eig` function for analyzing the correlation matrix. If SVD is used, the correlation matrix is never explicitly computed, but the singular values are the λ_k.

Diagnostics

There must be at least one output argument and at least two inputs; otherwise, `pmusic` stops and gives one of the following error messages:

```
Must have at least 1 output argument.
Must have at least 2 input arguments.
```

The first argument must be a full matrix, otherwise `pmusic` gives the following error message:

```
Input signal or correlation cannot be sparse.
```

If the second argument was entered as an empty matrix, or if it has more than two elements, or if it has negative or non-integer elements, `pmusic` gives one of the following error messages:

```
P cannot be empty.
Second input must have only 1 or 2 elements.
P must be an integer.
Second input must contain non-negative entries.
```

If the value of `p` is too large with respect to the eigenvector dimension, and `thresh` is less than 1, no eigenvectors can be assigned to the noise subspace and the algorithm fails. In this case, `pmusic` gives the following error message:

```
Noise subspace dimension cannot be zero.
```

If the `'corr'` parameter is used, then the first input must be a square correlation matrix. If it is not, `pmusic` gives the following error message:

```
Correlation matrix (R) is not square.
```

The correlation matrix is then checked for validity; if it fails, `pmusic` gives the following error message:

```
Correlation matrix (R) has negative or complex eigenvalue.
```

See Also

`lpc`	Linear prediction coefficients.
`pmem`	Power spectrum estimate using maximum entropy method (MEM).
`pmtm`	Power spectrum estimate using the multitaper method (MTM).
`prony`	Prony's method for time domain IIR filter design.
`psd`	Estimate the power spectral density (PSD) of a signal.

References

[1] Schmidt, R.O. “Multiple Emitter Location and Signal Parameter Estimation.” *IEEE Trans. Antennas Propagation*. Vol. AP-34 (March 1986). Pgs. 276-280.

[2] Marple, S.L. *Digital Spectral Analysis*. Englewood Cliffs, NJ: Prentice Hall, 1987. Pgs. 373-378.

Purpose Reflection coefficients from polynomial coefficients.

Syntax

```
k = poly2rc(a)
```

Description `k = poly2rc(a)` finds the reflection coefficients of the lattice structure of the discrete filter `a`. `a` must be real, and `a(1)` cannot be 0. `k` is a row vector of size `length(a)-1`.

A simple, fast way to check if `a` has all of its roots inside the unit circle is to check if each of the elements of `k` have magnitude less than 1:

```
stable = all(abs(poly2rc(a))<1)
```

Example Consider an IIR filter given by

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

Its reflection coefficient representation is

```
k = poly2rc(a)

k =

        0.3090     0.9800     0.0031     0.0082    -0.0082
```

Limitations If `abs(k(i)) == 1` for any `i`, finding the reflection coefficients is an ill-conditioned problem. `poly2rc` will return some `NaN`s and provide a warning message in this case.

Algorithm `poly2rc` implements the recursive relationship:

$$k(n) = a_n(n)$$

$$a_{n-1}(m) = \frac{a_n(m) - k(n)a_n(n-m)}{1 - k(n)^2}, \quad m = 1, 2, \ldots n-1$$

This relationship is based on Levinson's recursion [1]. To implement it, `poly2rc` loops through `a` in reverse order after discarding its first element. For each loop iteration `i`, the function

1. sets `k(i)` equal to `a(i)`
2. applies the second relationship above to elements 1 through `i` of the vector `a`:

```
a = (a-k(i)*fliplr(a))/(1-k(i)^2);
```

See Also

`latc2tf`	Lattice filter to transfer function conversion.
`latcfilt`	Lattice and lattice-ladder filter implementation.
`rc2poly`	Polynomial coefficients from reflection coefficients.
`tf2latc`	Transfer function to lattice filter conversion.

References

[1] McClellan, J. "Parametric Signal Modeling." *Advanced Topics in Signal Processing*. Oppenheim, A.V., and J.S. Lim, ed. Englewood Cliffs, NJ: Prentice Hall, 1988. Pgs. 20-21.

Purpose

Stabilize polynomial.

Syntax

```
b = polystab(a)
```

Description

`polystab` stabilizes a polynomial with respect to the unit circle; it reflects roots with magnitudes greater than 1 inside the unit circle.

`b = polystab(a)` where a is a vector of polynomial coefficients normally in the *z*-domain:

$$a(z) = a(1) + a(2)z^{-1} + \cdots + a(na+1)z^{-na}$$

returns a row vector containing the stabilized polynomial.

Example

`polystab` can convert a linear-phase filter into a minimum-phase filter with the same magnitude response:

```
h = fir1(25,0.4);
hmin = polystab(h)*norm(h)/norm(polystab(h));
```

Algorithm

`polystab` finds the roots of the polynomial and maps those roots found outside the unit circle to the inside of the unit circle:

```
v = roots(a);
vs = 0.5*(sign(abs(v)-1)+1);
v = (1-vs).*v + vs./conj(v);
b = a(1)*poly(v);
```

See Also

`roots`	Polynomial roots (see MATLAB Function Reference).

prony

Purpose

Prony's method for time domain IIR filter design.

Syntax

```
[b,a] = prony(h,nb,na)
```

Description

Prony's method is an algorithm for finding an IIR filter with a prescribed time domain impulse response. It has applications in filter design, exponential signal modeling, and system identification (parametric modeling).

`[b,a] = prony(h,nb,na)` finds a filter with numerator order `nb`, denominator order `na`, and the time domain impulse response in `h`. `prony` returns the filter coefficients in row vectors `b` and `a`, of length `nb + 1` and `na + 1`, respectively. The filter coefficients are in descending powers of z:

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

Example

Recover the coefficients of a Butterworth filter from its impulse response:

```
[b,a] = butter(4,0.2)

b =
      0.0048      0.0193      0.0289      0.0193      0.0048

a =
      1.0000     –2.3695      2.3140     –1.0547      0.1874

h = filter(b,a,[1 zeros(1,25)]);
[bb,aa] = prony(h,4,4)

bb =
      0.0048      0.0193      0.0289      0.0193      0.0048

ab =
      1.0000     –2.3695      2.3140     –1.0547      0.1874
```

Algorithm

`prony` implements the method described in reference [1]. This method uses a variation of the covariance method of AR modeling to find the denominator coefficients `a` and then finds the numerator coefficients `b` for which the impulse response of the output filter matches exactly the first `nb + 1` samples of `x`. The

filter is not necessarily stable, but potentially can recover the coefficients exactly if the data sequence is truly an autoregressive moving average (ARMA) process of the correct order.

See Also

`butter`	Butterworth analog and digital filter design.
`cheby1`	Chebyshev type I filter design (passband ripple).
`cheby2`	Chebyshev type II filter design (stopband ripple).
`ellip`	Elliptic (Cauer) filter design.
`invfreqz`	Discrete-time filter identification from frequency data.
`levinson`	Levinson-Durbin recursion.
`lpc`	Linear prediction coefficients.
`stmcb`	Linear model using Steiglitz-McBride iteration.

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Pgs. 226-228.

Purpose Estimate the power spectral density (PSD) of a signal.

Syntax

```
Pxx = psd(x)
Pxx = psd(x,nfft)
[Pxx,f] = psd(x,nfft,Fs)
Pxx = psd(x,nfft,Fs,window)
Pxx = psd(x,nfft,Fs,window,noverlap)
Pxx = psd(x,...,'dflag')
[Pxx,Pxxc,f] = psd(x,nfft,Fs,window,noverlap,p)
psd(x,...)
```

Description

`Pxx = psd(x)` estimates the power spectrum of the sequence `x` using the Welch method of spectral estimation. `Pxx = psd(x)` uses the following default values:

- `nfft = min(256,length(x))`
- `Fs = 2`
- `window = hanning(nfft)`
- `noverlap = 0`

`nfft` specifies the FFT length that `psd` uses. This value determines the frequencies at which the power spectrum is estimated. `Fs` is a scalar that specifies the sampling frequency. `window` specifies a windowing function and the number of samples `psd` uses in its sectioning of the `x` vector. `noverlap` is the number of samples by which the sections overlap. Any arguments that you omit from the end of the input parameter list use the default values shown above.

If `x` is real, `psd` estimates the spectrum at positive frequencies only; in this case, the output `Pxx` is a column vector of length `nfft/2+1` for `nfft` even and `(nfft+1)/2` for `nfft` odd. If `x` is complex, `psd` estimates the spectrum at both positive and negative frequencies and `Pxx` has length `nfft`.

`Pxx = psd(x,nfft)` uses the specified FFT length `nfft` in estimating the power spectrum for `x`. Specify `nfft` as a power of 2 for fastest execution.

`[Pxx,f] = psd(x,nfft,Fs)` returns a vector `f` of frequencies at which the function evaluates the PSD. `f` is the same size as `Pxx`, so `plot(f,Pxx)` plots the power spectrum versus properly scaled frequency. `Fs` has no effect on the output `Pxx`; it is a frequency scaling multiplier.

`Pxx = psd(x,nfft,Fs,window)` specifies a windowing function and the number of samples per section of the x vector. If you supply a scalar for window, psd uses a Hanning window of that length. The length of the window must be less than or equal to nfft; psd zero pads the sections if the length of the window is less than nfft. psd returns an error if the length of the window is greater than nfft.

`Pxx = psd(x,nfft,Fs,window,noverlap)` overlaps the sections of x by noverlap samples.

You can use the empty matrix [] to specify the default value for any input argument except x. For example,

```
psd(x,[],10000)
```

is equivalent to

```
psd(x)
```

but with a sampling frequency of 10,000 Hz instead of the default of 2 Hz.

`Pxx = psd(x,...,'dflag')` specifies a detrend option, where *dflag* is

- linear, to remove the best straight-line fit from the prewindowed sections of x
- mean, to remove the mean from the prewindowed sections of x
- none, for no detrending (default)

The *dflag* parameter must appear last in the list of input arguments. psd recognizes a *dflag* string no matter how many intermediate arguments are omitted.

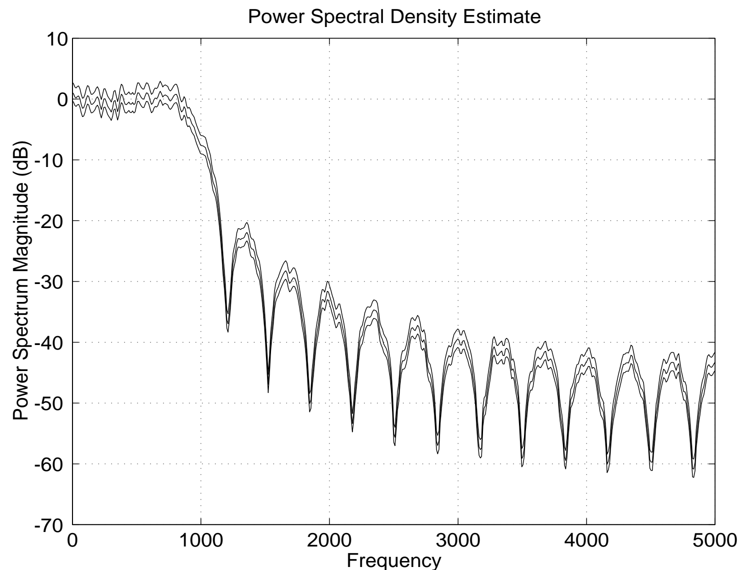
`[Pxx,Pxxc,f] = psd(x,nfft,Fs,window,noverlap,p)` where p is a positive scalar between 0 and 1 returns a vector Pxxc that contains an estimate of the p*100 percent confidence interval for Pxx. Pxxc is a two-column matrix that is the same length as Pxx. The interval [Pxxc(:,1),Pxxc(:,2)] covers the true PSD with probability p. plot(f,[Pxx Pxxc Pxxc]) plots the power spectrum inside the p*100 percent confidence interval. If unspecified, p defaults to 0.95.

`psd(x,...)` with no output arguments plots the PSD versus frequency in the current figure window. If the `p` parameter is specified, the plot includes the confidence interval.

Example

Generate a colored noise signal and plot its PSD with a confidence interval of 95%. Specify a length 1024 FFT, a 512-point Kaiser window with no overlap, and a sampling frequency of 10 kHz:

```
h = fir1(30,0.2,boxcar(31));      % design a lowpass filter
r = randn(16384,1);               % white noise
x = filter(h,1,r);                % color the noise
psd(x,1024,10000,kaiser(512,5),0,0.95)
```



Algorithm

psd calculates the power spectral density using Welch's method (see references [1] and [2]):

1. It applies the window specified by the window vector to each successive de-trended section of x.
2. It transforms each section with an nfft-point FFT.
3. It forms the periodogram of each section by scaling the magnitude squared of each transform.
4. It averages the periodograms of the overlapping sections to form Pxx, the power spectrum of x.

The number of sections that psd averages is

```
k = fix((length(x)-noverlap)/(length(window)-noverlap))
```

Diagnostics

An appropriate diagnostic message is displayed when incorrect arguments to psd are used:

```
Requires window's length to be no greater than FFT length.
Requires NOVERLAP to be strictly less than the window length.
Requires positive integer values for NFFT and NOVERLAP.
Requires confidence parameter to be a scalar between 0 and 1.
Requires vector input.
```

See Also

cohere	Estimate magnitude squared coherence function between two signals.
csd	Estimate the cross spectral density (CSD) of two signals.
pmem	Power spectrum estimate using maximum entropy method (MEM).
pmtm	Power spectrum estimate using the multitaper method (MTM).
pmusic	Power spectrum estimate using MUSIC eigenvector method.
specgram	Time-dependent frequency analysis (spectrogram).
tfe	Transfer function estimate from input and output.

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975. Pgs. 399-419.

[2] Welch, P.D. “The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms.” *IEEE Trans. Audio Electroacoust.* Vol. AU-15 (June 1967). Pgs. 70-73.

[3] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. Pgs. 311-312.

Purpose

Pulse train generator.

Syntax

```
y = pulstran(t,d,'func')
y = pulstran(t,d,'func',p1,p2,...)
y = pulstran(t,d,p,Fs)
y = pulstran(t,d,p)
```

Description

`pulstran` generates pulse trains from continuous functions or sampled prototype pulses.

`y = pulstran(t,d,'func')` generates a pulse train based on samples of a continuous function, `'func'`, where *func* is

- `gauspuls`, for Gaussian-modulated sinusoidal pulse generator
- `rectpuls`, for sampled aperiodic rectangle generator
- `tripuls`, for sampled aperiodic triangle generator

`pulstran` is evaluated `length(d)` times and returns the sum of the evaluations `y = func(t-d(1)) + func(t-d(2)) + ...`.

The function is evaluated over the range of argument values specified in array `t`, after removing a scalar argument offset taken from the vector `d`. Note that *func* must be a vectorized function that can take an array `t` as an argument.

An optional gain factor may be applied to each delayed evaluation by specifying `d` as a two-column matrix, with the offset defined in column 1 and associated gain in column 2 of `d`. Note that a row vector will be interpreted as specifying delays only.

`pulstran(t,d,'func',p1,p2,...)` allows additional parameters to be passed to '*func*' as necessary. For example,

```
func(t-d(1),p1,p2,...) + func(t-d(2),p1,p2,...) + ...
```

`pulstran(t,d,p,Fs)` generates a pulse train that is the sum of multiple delayed interpolations of the prototype pulse in vector `p`, sampled at the rate `Fs`, where `p` spans the time interval `[0, (length(p)-1)/Fs]`, and its samples are identically 0 outside this interval. By default, linear interpolation is used for generating delays.

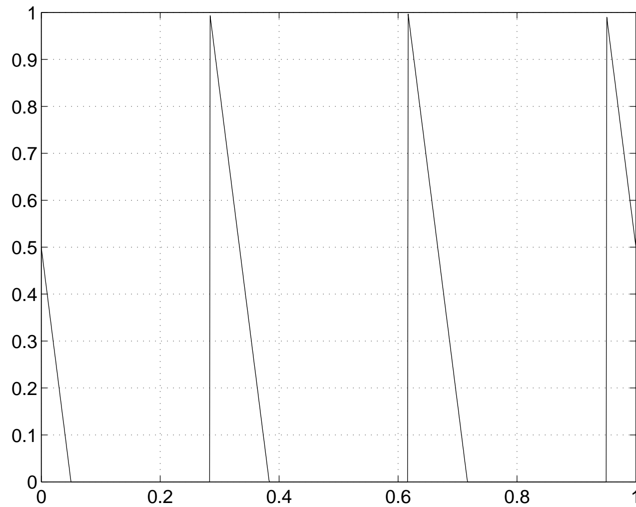
`pulstran(t,d,p)` assumes that the sampling rate `Fs` is equal to 1 Hz.

`pulstran(...,'func')` specifies alternative interpolation methods. See `interp1` for a list of available methods.

Examples

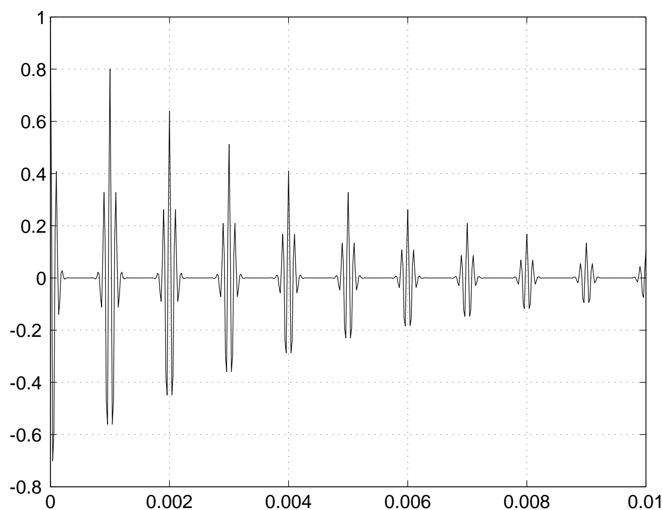
This example generates an asymmetric sawtooth waveform with a repetition frequency of 3 Hz and a sawtooth width of 0.1 sec. It has a signal length of 1 sec and a 1 kHz sample rate:

```
t = 0 : 1/1e3 : 1;          % 1 kHz sample freq for 1 sec
d = 0 : 1/3 : 1;            % 3 Hz repetition freq
y = pulstran(t,d,'tripuls',0.1,-1);
plot(t,y)
```



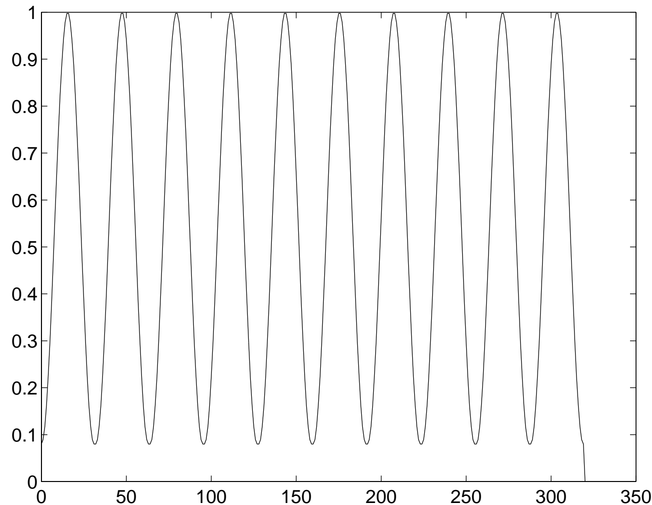
This example generates a periodic Gaussian pulse signal at 10 kHz, with 50% bandwidth. The pulse repetition frequency is 1 kHz, sample rate is 50 kHz, and pulse train length is 10 msec. The repetition amplitude should attenuate by 0.8 each time:

```
t = 0 : 1/50E3 : 10e-3;
d = [0 : 1/1E3 : 10e-3 ; 0.8.^(0:10)]';
y = pulstran(t,d,'gauspuls',10e3,0.5);
plot(t,y)
```



This example generates a train of 10 Hamming windows:

```
p = hamming(32);
t = 0:320;  d = (0:9)'*32;
y = pulstran(t,d,p);
plot(t,y)
```



See Also

`chirp`	Swept-frequency cosine generator.
`cos`	Cosine of vector/matrix elements (see MATLAB Function Reference).
`diric`	Dirichlet or periodic sinc function.
`gauspuls`	Gaussian-modulated sinusoidal pulse generator.
`rectpuls`	Sampled aperiodic rectangle generator.
`sawtooth`	Sawtooth or triangle wave generator.
`sin`	Sine of vector/matrix elements (see MATLAB Function Reference).
`sinc`	Sinc or $\sin(\pi t)/\pi t$ function.
`square`	Square wave generator.
`tripuls`	Sampled aperiodic triangle generator.

Purpose

Polynomial coefficients from reflection coefficients.

Syntax

```
a = rc2poly(k)
```

Description

`a = rc2poly(k)` finds the filter coefficients `a`, with `a(1) = 1`, from the real reflection coefficients `k` of the lattice structure of a discrete filter. `a` is row vector of length `k + 1`.

Example

Consider an IIR filter given by reflection coefficients `k`:

```
k = [0.3090    0.9800    0.0031    0.0082   -0.0082];
```

Its polynomial representation is

```
a = rc2poly(k)

a =

1.0000    0.6149    0.9899    0.0000    0.0031   -0.0082
```

Algorithm

`rc2poly` computes `a` using Levinson's recursion [1]. The function

1. sets the output vector `a` to the first element of `k`
2. loops through the remaining elements of `k`

   For each loop iteration `i`, `a = [a + a(i-1:-1:1)*k(i)  k(i)]`.
3. implements `a = [1 a]`

See Also

`latc2tf`	Lattice filter to transfer function conversion.
`latcfilt`	Lattice and lattice-ladder filter implementation.
`poly2rc`	Reflection coefficients from polynomial coefficients.
`tf2latc`	Transfer function to lattice filter conversion.

References

[1] McClellan, J. "Parametric Signal Modeling." *Advanced Topics in Signal Processing*. Oppenheim, A.V., and J.S. Lim, ed. Englewood Cliffs, NJ: Prentice Hall, 1988. Pgs. 20-21.

rceps

Purpose

Real cepstrum and minimum phase reconstruction.

Syntax

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

Description

The *real cepstrum* is the inverse Fourier transform of the real logarithm of the magnitude of the Fourier transform of a sequence.

`rceps(x)` returns the real cepstrum of the real sequence `x`. The real cepstrum is a real-valued function.

`[y,ym] = rceps(x)` returns both the real cepstrum `y` and a minimum phase reconstructed version `ym` of the input sequence.

Algorithm

`rceps` is an M-file implementation of algorithm 7.2 in [2], that is:

```
y = real(ifft(log(abs(fft(x)))));
```

Appropriate windowing in the cepstral domain forms the reconstructed minimum phase signal:

```
w = [1; 2*ones(n/2-1,1); ones(1 - rem(n,2),1); zeros(n/2-1,1)];
ym = real(ifft(exp(fft(w.*y))));
```

See Also

`cceps`	Complex cepstral analysis.
`fft`	One-dimensional fast Fourier transform.
`hilbert`	Hilbert transform.
`icceps`	Inverse complex cepstrum.
`unwrap`	Unwrap phase angles.

References

[1] Oppenheim, A.V., and R.W. Schafer. *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975.

[2] IEEE. *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979.

rectpuls

Purpose Sampled aperiodic rectangle generator.

Syntax

```
y = rectpuls(t)
y = rectpuls(t,w)
```

Description `y = rectpuls(t)` returns a continuous, aperiodic, unity-height rectangular pulse at the sample times indicated in array `t`, centered about `t = 0` and with a default width of 1. Note that the interval of non-zero amplitude is defined to be open on the right, that is, `rectpuls(-0.5) = 1` while `rectpuls(0.5) = 0`.

`y = rectpuls(t,w)` generates a rectangle of width `w`.

`rectpuls` is typically used in conjunction with the pulse train generating function, `pulstran`.

See Also

`chirp`	Swept-frequency cosine generator.
`cos`	Cosine of vector/matrix elements (see MATLAB Function Reference).
`diric`	Dirichlet or periodic sinc function.
`gauspuls`	Gaussian-modulated sinusoidal pulse generator.
`pulstran`	Pulse train generator.
`sawtooth`	Sawtooth or triangle wave generator.
`sin`	Sine of vector/matrix elements (see MATLAB Function Reference).
`sinc`	Sinc or $\sin(\pi t)/\pi t$ function.
`square`	Square wave generator.
`tripuls`	Sampled aperiodic triangle generator.

remez

Purpose

Parks-McClellan optimal FIR filter design.

Syntax

```
b = remez(n,f,a)
b = remez(n,f,a,w)
b = remez(n,f,a,'ftype')
b = remez(n,f,a,w,'ftype')
```

Description

`remez` designs a linear-phase FIR filter using the Parks-McClellan algorithm [1]. The Parks-McClellan algorithm uses the Remez exchange algorithm and Chebyshev approximation theory to design filters with an optimal fit between the desired and actual frequency responses. The filters are optimal in the sense that the maximum error between the desired frequency response and the actual frequency response is minimized. Filters designed this way exhibit an equiripple behavior in their frequency responses and hence are sometimes called *equiripple* filters.

`b = remez(n,f,a)` returns row vector `b` containing the `n+1` coefficients of the order `n` FIR filter whose frequency-amplitude characteristics match those given by vectors `f` and `a`.

The output filter coefficients (taps) in `b` obey the symmetry relation

$$b(k) = b(n+2-k), \quad k = 1, \ldots, n+1$$

Vectors `f` and `a` specify the frequency-magnitude characteristics of the filter:

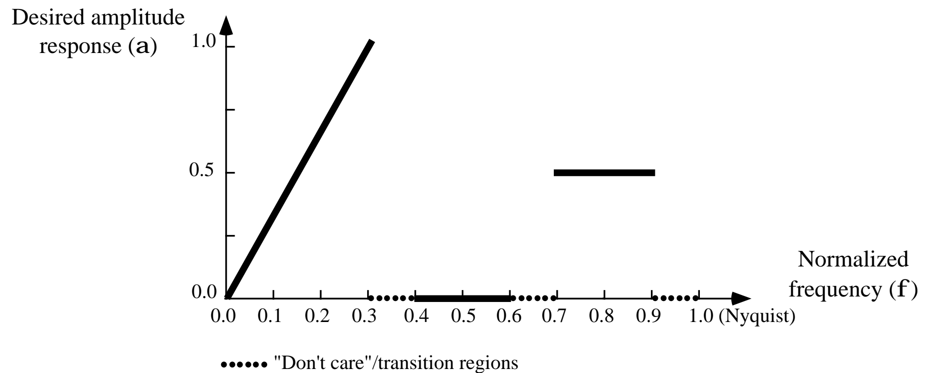
- `f` is a vector of pairs of frequency points, specified in the range between 0 and 1, where 1 corresponds to half the sampling frequency (the Nyquist frequency). The frequencies must be in increasing order.
- `a` is a vector containing the desired amplitudes at the points specified in `f`.

  The desired amplitude at frequencies between pairs of points $(f(k), f(k+1))$ for k odd is the line segment connecting the points $(f(k), a(k))$ and $(f(k+1), a(k+1))$.

  The desired amplitude at frequencies between pairs of points $(f(k), f(k+1))$ for k even is unspecified. The areas between such points are transition or "don't care" regions.
- `f` and `a` must be the same length. The length must be an even number.

The relationship between the f and a vectors in defining a desired frequency response is

```
f = [0 .3 .4 .6 .7 .9]
a = [0  1  0  0 .5 .5]
```



remez(n,f,a,w) uses the weights in vector w to weight the fit in each frequency band. The length of w is half the length of f and a, so there is exactly one weight per band.

b = remez(n,f,a,'*ftype*') and

b = remez(n,f,a,w,'*ftype*') specify a filter type, where *ftype* is

- hilbert, for linear-phase filters with odd symmetry (type III and type IV)

  The output coefficients in b obey the relation $b(k) = -b(n+2-k)$, $k = 1,...,n+1$. This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.

  For example,

  ```
  h = remez(30,[0.1 0.9],[1 1],'Hilbert');
  ```

  designs an approximate FIR Hilbert transformer of length 31.

- differentiator, for type III and IV filters, using a special weighting technique

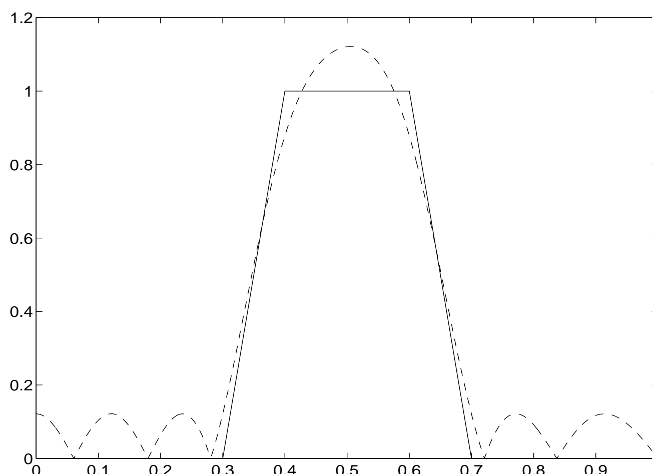
  For nonzero amplitude bands, it weights the error by a factor of $1/f$ so that the error at low frequencies is much smaller than at high frequencies. For

FIR differentiators, which have an amplitude characteristic proportional to frequency, these filters minimize the maximum relative error (the maximum of the ratio of the error to the desired amplitude).

Example

Graph the desired and actual frequency responses of a 17th-order Parks-McClellan bandpass filter:

```
f = [0 0.3 0.4 0.6 0.7 1]; a = [0 0 1 1 0 0];
b = remez(17,f,a);
[h,w] = freqz(b,1,512);
plot(f,a,w/pi,abs(h))
```



Algorithm

`remez` is a MEX-file version of the original FORTRAN code from [1], altered to design arbitrarily long filters with arbitrarily many linear bands.

`remez` designs type I, II, III, and IV linear-phase filters. Type I and Type II are the defaults for `n` even and `n` odd, respectively, while Type III (`n` even) and Type IV (`n` odd) are obtained with the `'hilbert'` and `'differentiator'` flags. The different types of filters have different symmetries and certain constraints on their frequency responses (see reference [5] for more details):

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

Diagnostics

An appropriate diagnostic message is displayed if incorrect arguments are used:

```
Filter order must be 3 or more.
There should be one weight per band.
Frequency and amplitude vectors must be the same length.
The number of frequency points must be even.
Frequencies must lie between 0 and 1.
Frequencies must be specified in bands.
Frequencies must be nondecreasing.
Adjacent bands not allowed.
```

A more serious warning message is

```
-- Failure to Converge --
Probable cause is machine rounding error.
```

In the rare event that you see this message, it is possible that the filter design may still be correct. Verify the design by checking its frequency response.

See Also

`butter`	Butterworth analog and digital filter design.
`cheby1`	Chebyshev type I filter design (passband ripple).
`cheby2`	Chebyshev type II filter design (stopband ripple).
`cremez`	Complex and nonlinear-phase equiripple FIR filter design
`ellip`	Elliptic (Cauer) filter design.
`fir1`	Window-based finite impulse response filter design—standard response.
`fir2`	Window-based finite impulse response filter design—arbitrary response.
`fircls`	Constrained least square FIR filter design for multiband filters.
`fircls1`	Constrained least square filter design for lowpass and highpass linear phase FIR filters.
`firls`	Least square linear-phase FIR filter design.
`firrcos`	Raised cosine FIR filter design.
`remezord`	Parks-McClellan optimal FIR filter order estimation.
`yulewalk`	Recursive digital filter design.

References

[1] IEEE. *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979. Algorithm 5.1.

[2] IEEE. *Selected Papers in Digital Signal Processing, II*. IEEE Press. New York: John Wiley & Sons, 1979.

[3] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987. Pg. 83.

[4] Rabiner, L.R., J.H. McClellan, and T.W. Parks. “FIR Digital Filter Design Techniques Using Weighted Chebyshev Approximations.” *Proc. IEEE* 63 (1975).

[5] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. Pgs. 256-266.

Purpose Parks-McClellan optimal FIR filter order estimation.

Syntax

```
[n,fo,ao,w] = remezord(f,a,dev)
[n,fo,ao,w] = remezord(f,a,dev,Fs)
c = remezord(f,a,dev,Fs,'cell')
```

Description `[n,fo,ao,w] = remezord(f,a,dev)` finds the approximate order, normalized frequency band edges, frequency band amplitudes, and weights that meet input specifications `f`, `a`, and `dev`, to use with the `remez` command.

- `f` is a vector of band edges, and `a` is a vector specifying the desired amplitude on the bands defined by `f`. The length of `f` is twice the length of `a`, minus 2. The desired function is piecewise constant.
- `dev` is a vector the same size as `a` that specifies the maximum allowable deviation or ripples between the frequency response and the desired amplitude of the output filter, for each band.

Use `remez` with the resulting order `n`, frequency vector `fo`, amplitude response vector `ao`, and weights `w` to design the filter `b` which approximately meets the specifications given by `remezord` input parameters `f`, `a`, and `dev`:

```
b = remez(n,fo,ao,w)
```

`[n,fo,ao,w] = remezord(f,a,dev,Fs)` specifies a sampling frequency `Fs`. `Fs` defaults to 2 Hz, implying a Nyquist frequency of 1 Hz. You can therefore specify band edges scaled to a particular application's sampling frequency.

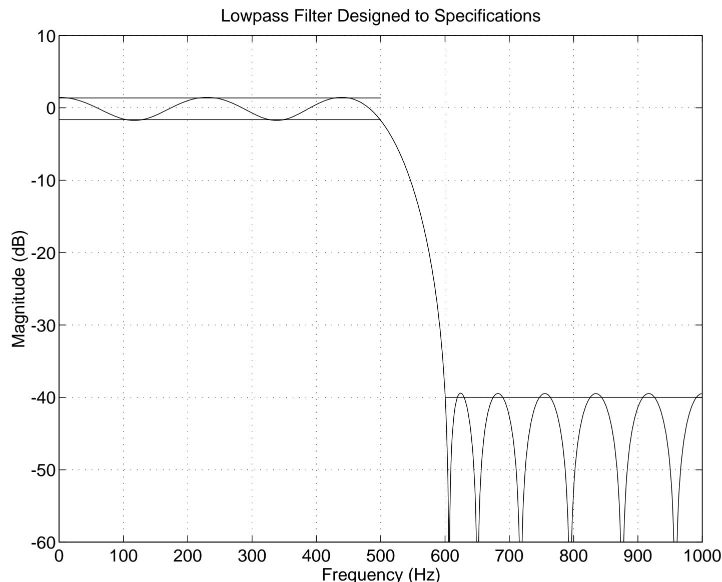
In some cases `remezord` underestimates the order `n`. If the filter does not meet the specifications, try a higher order such as `n+1` or `n+2`.

`c = remezord(f,a,dev,Fs,'cell')` specifies a cell-array whose elements are the parameters to `remez`.

Examples

Design a minimum-order lowpass filter with a 500 Hz passband cutoff frequency and 600 Hz stopband cutoff frequency, with a sampling frequency of 2000 Hz), at least 40 dB attenuation in the stopband, and less than 3 dB of ripple in the passband:

```
rp = 3;            % passband ripple
rs = 40;           % stopband ripple
Fs = 2000;         % sampling frequency
f = [500 600];     % cutoff frequencies
a = [1 0];         % desired amplitudes
% compute deviations
dev = [(10^(rp/20)-1)/(10^(rp/20)+1)  10^(-rs/20)];
[n,fo,ao,w] = remezord(f,a,dev,Fs);
b = remez(n,fo,ao,w);
[h,f] = freqz(b,1,1024,Fs);
plot(f,20*log10(abs(h)))
```



Note that the filter falls slightly short of meeting the specifications. Using `n+1` in the call to `remez` instead of `n` achieves the desired amplitude characteristics.

Design a lowpass filter with a 1500 Hz passband cutoff frequency and 2000 Hz stopband cutoff frequency, with a sampling frequency of 8000 Hz, a maximum stopband amplitude of 0.1, and a maximum passband error (ripple) of 0.01:

```
[n,fo,ao,w] = remezord( [1500 2000], [1 0], [0.01 0.1], 8000 );
b = remez(n,fo,ao,w);
```

This is equivalent to

```
c = remezord( [1500 2000], [1 0], [0.01 0.1], 8000, 'cell');
b = remez(c{:});
```

NOTE In some cases, `remezord` underestimates or overestimates the order `n`. If the filter does not meet the specifications, try a higher order such as `n+1` or `n+2`.

NOTE Results are inaccurate if the cutoff frequencies are near 0 or the Nyquist frequency.

Algorithm

`remezord` uses the algorithm suggested in [1]. This method is inaccurate for band edges close to either 0 or the Nyquist frequency (`Fs/2`).

Diagnostics

If the input parameter lengths are not consistent, `remezord` gives the following error messages:

```
Requires M and DEV to be the same length.
Length of F must be length(M)-2.
```

See Also

`buttord`	Butterworth filter order selection.
`cheb1ord`	Chebyshev type I filter order selection.
`cheb2ord`	Chebyshev type II filter order selection.
`ellipord`	Elliptic filter order selection.
`kaiserord`	Estimate parameters for fir1 with Kaiser window.
`remez`	Parks-McClellan optimal FIR filter design.

References

[1] Rabiner, L.R., and O. Herrmann. “The Predictability of Certain Optimum Finite Impulse Response Digital Filters.” *IEEE Trans. on Circuit Theory*. Vol. CT-20, No. 4 (July 1973). Pgs. 401-408.

[2] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975. Pgs. 156-157.

Purpose Change sampling rate by any factor.

Syntax

```
y = resample(x,p,q)
y = resample(x,p,q,n)
y = resample(x,p,q,n,beta)
y = resample(x,p,q,b)
[y,b] = resample(x,p,q)
```

Description

`y = resample(x,p,q)` resamples the sequence in vector `x` at `p/q` times the original sampling rate, using a polyphase filter implementation. `y` is `p/q` times the length of `x`. `p` and `q` must be positive integers. If `x` is a matrix, `resample` works down the columns of `x`.

`resample` applies an anti-aliasing (lowpass) FIR filter to `x` during the resampling process. It designs the filter using `fir1` with a Kaiser window.

`y = resample(x,p,q,n)` uses `n` terms on either side of `x[n]` to perform the resampling. The length of the FIR filter `resample` uses is proportional to `n`; larger values of `n` provide better accuracy at the expense of more computation time. The default for `n` is 10. If you let `n = 0`, `resample` uses a zero-order hold.

`y = resample(x,p,q,n,beta)` uses `beta` as the design parameter for the Kaiser window that `resample` uses in designing the lowpass filter. The default for `beta` is 5.

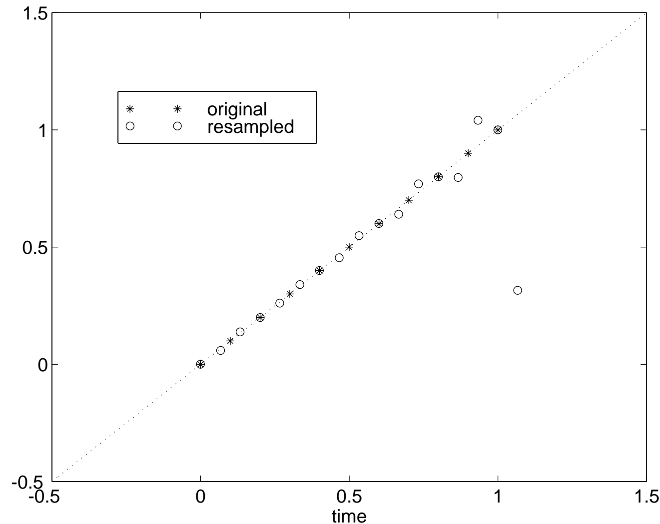
`y = resample(x,p,q,b)` filters `x` with `b`, a vector of filter coefficients.

`[y,b] = resample(x,p,q)` returns the vector `b`, which contains the coefficients of the filter applied to `x` during the resampling process.

Examples

Resample a simple linear sequence at 3/2 the original rate:

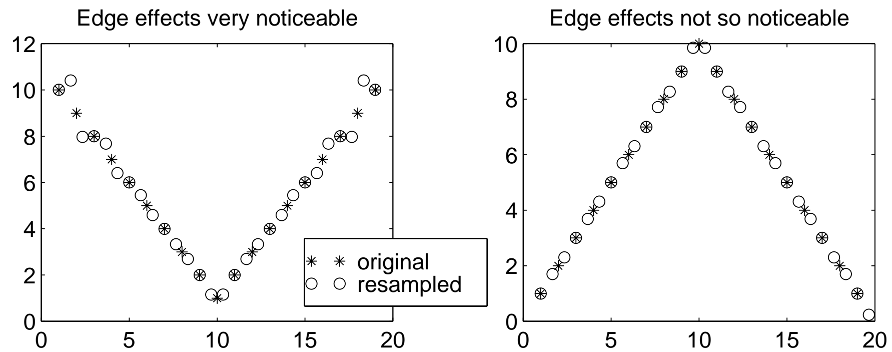
```
Fs1 = 10;               % original sampling freq. in Hz
t1 = 0:1/Fs1:1;         % time vector
x = t1;                 % define a linear sequence
y = resample(x,3,2);  % now resample it
t2 = (0:(length(y)-1))*2/(3*Fs1);  % new time vector
plot(t1,x,'*',t2,y,'o',-0.5:0.01:1.5,-0.5:0.01:1.5,':')
```



Notice that the last few points of the output y are inaccurate. In its filtering process, resample assumes the samples at times before and after the given samples in x are equal to zero. Thus large deviations from zero at the end

points of the sequence x can cause inaccuracies in y at its end points. The following two plots illustrate this side effect of resample:

```
x = [1:10 9:-1:1]; y = resample(x,3,2);
plot(1:19,x,'*',(0:28)*2/3 + 1,y,'o')
x = [10:-1:1 2:10]; y = resample(x,3,2);
plot(1:19,x,'*',(0:28)*2/3 + 1,y,'o')
```



Diagnostics

If p or q are not positive integers, resample gives the appropriate error message:

```
P must be a positive integer.
Q must be a positive integer.
```

If x is not a vector, resample gives the following error message:

```
Input X must be a vector.
```

See Also

decimate	Decrease the sampling rate for a sequence (decimation).
fir1	Window-based finite impulse response filter design—standard response.
interp	Increase sampling rate by an integer factor (interpolation).
interp1	1-D data interpolation (table lookup) (see MATLAB Function Reference).
intfilt	Interpolation FIR filter design.
kaiser	Kaiser window.
spline	Cubic spline interpolation (see MATLAB Function Reference).
upfirdn	Apply FIR filter and perform rate changing.

Purpose z-transform partial-fraction expansion.

Syntax

```
[r,p,k] = residuez(b,a)
[b,a] = residuez(r,p,k)
```

Description `residuez` converts a discrete time system, expressed as the ratio of two polynomials, to partial fraction expansion, or residue, form. It also converts the partial fraction expansion back to the original polynomial coefficients.

`[r,p,k] = residuez(b,a)` finds the residues, poles, and direct terms of a partial fraction expansion of the ratio of two polynomials, *b(z)* and *a(z)*. Vectors `b` and `a` specify the coefficients of the polynomials of the discrete-time system *b(z)/a(z)* in descending powers of z:

$$b(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_m z^{-m}$$

$$a(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_n z^{-n}$$

If there are no multiple roots and `a > n-1`,

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

The returned column vector `r` contains the residues, column vector `p` contains the pole locations, and row vector `k` contains the direct terms. The number of poles is

```
n = length(a)-1 = length(r) = length(p)
```

The direct term coefficient vector `k` is empty if `length(b) < length(a)`; otherwise

```
length(k) = length(b) - length(a) + 1
```

If `p(j) = ... = p(j+s-1)` is a pole of multiplicity `s`, then the expansion includes terms of the form

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

`[b,a] = residuez(r,p,k)` with three input arguments and two output arguments, converts the partial fraction expansion back to polynomials with coefficients in row vectors `b` and `a`.

The `residue` function in the MATLAB environment is very similar to `residuez`. It computes the partial fraction expansion of continuous-time systems in the Laplace domain (see reference [1]), rather than discrete-time systems in the z-domain as does `residuez`.

Algorithm

`residuez` applies standard MATLAB functions and partial fraction techniques to find `r`, `p`, and `k` from `b` and `a`:

1. It finds the direct terms `a` using `deconv` (polynomial long division) when `length(b)>length(a)-1`.
2. It finds the poles using `p = roots(a)`. `mpoles` finds repeated poles and reorders the poles according to their multiplicities.
3. It finds the residue for each nonrepeating pole p_i by multiplying $b(z)/a(z)$ by $1/(1-p_i z^{-1})$ and evaluating the resulting rational function at $z = p_i$.
4. It finds the residues for the repeated poles by solving

   ```
   S2*r2 = h - S1*r1
   ```

   for `r2` using `\`. `h` is the impulse response of the reduced $b(z)/a(z)$, `S1` is a matrix whose columns are impulse responses of the first-order systems made up of the nonrepeating roots, and `r1` is a column containing the residues for the nonrepeating roots. Each column of matrix `S2` is an impulse response. For each root p_j of multiplicity s_j, `S2` contains s_j columns representing the impulse responses of each of the following systems:

   $$\frac{1}{1-p_j z^{-1}}, \frac{1}{(1-p_j z^{-1})^2}, \ldots, \frac{1}{(1-p_j z^{-1})^{s_j}}$$

The vector `h` and matrices `S1` and `S2` have `n + xtra` rows, where `n` is the total number of roots and the internal parameter `xtra`, set to 1 by default, determines the degree of overdetermination of the system of equations.

Diagnostics

If `a(1) == 0` while computing the partial fraction decomposition using `[r,p,k] = residuez(b,a)`, `residuez` gives the following error message:

```
First coefficient in A vector must be nonzero.
```

If the number of residues `r` and poles `p` is not the same, `residuez` gives the following error message:

```
R and P vectors must be the same size.
```

See Also

`convmtx`	Convolution matrix.
`deconv`	Deconvolution and polynomial division (see MATLAB Function Reference).
`poly`	Polynomial with specified roots (see MATLAB Function Reference).
`prony`	Prony's method for time domain IIR filter design.
`residue`	Partial fraction expansion (see MATLAB Function Reference).
`roots`	Polynomial roots (see MATLAB Function Reference).
`ss2tf`	State-space to zero-pole-gain conversion.
`tf2ss`	Transfer function to state-space conversion.
`tf2zp`	Transfer function to zero-pole-gain conversion.
`zp2ss`	Zero-pole-gain to state-space conversion.

References

[1] Oppenheim, A.V., and R.W. Schafer. *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975. Pgs. 166-170.

sawtooth

Purpose

Sawtooth or triangle wave generator.

Syntax

```
x = sawtooth(t)
x = sawtooth(t,width)
```

Description

`sawtooth(t)` generates a sawtooth wave with period 2π for the elements of time vector `t`. `sawtooth(t)` is similar to `sin(t)`, but it creates a sawtooth wave with peaks of -1 and 1 instead of a sine wave. The sawtooth wave is defined to be -1 at multiples of 2π and to increase linearly with time with a slope of $1/\pi$ at all other times.

`sawtooth(t,width)` generates a modified triangle wave where `width`, a scalar parameter between 0 and 1, determines the fraction between 0 and 2π at which the maximum occurs. The function increases from -1 to 1 on the interval 0 to 2π*`width`, then decreases linearly from 1 to -1 on the interval 2π*`width` to 2π. Thus a parameter of 0.5 specifies a standard triangle wave, symmetric about time instant π with peak-to-peak amplitude of 1. `sawtooth(t,1)` is equivalent to `sawtooth(t)`.

Diagnostics

If the `width` parameter is not a scalar, `sawtooth` gives the following error message:

```
Requires WIDTH parameter to be a scalar.
```

See Also

`chirp`	Swept-frequency cosine generator.
`cos`	Cosine of vector/matrix elements (see MATLAB Function Reference).
`diric`	Dirichlet or periodic sinc function.
`gauspuls`	Gaussian-modulated sinusoidal pulse generator.
`pulstran`	Pulse train generator.
`rectpuls`	Sampled aperiodic rectangle generator.
`sin`	Sine of vector/matrix elements (see MATLAB Function Reference).
`sinc`	Sinc or $\sin(\pi t)/\pi t$ function.
`square`	Square wave generator.
`tripuls`	Sampled aperiodic triangle generator.

Purpose

Sinc or sin(πt)/πt function.

Syntax

```
y = sinc(x)
```

Description

`sinc` computes the sinc function of an input vector or array, where the sinc function is

$$\mathrm{sinc}(t) = \begin{cases} 1, & t = 0 \\ \dfrac{\sin(\pi t)}{\pi t}, & t \neq 0 \end{cases}$$

This function is the continuous inverse Fourier transform of the rectangular pulse of width 2π and height 1:

$$\mathrm{sinc}(t) = \frac{1}{2\pi}\int_{-\pi}^{\pi} e^{j\omega t} d\omega$$

`y = sinc(x)` returns an array `y` the same size as `x`, whose elements are the `sinc` function of the elements of `x`.

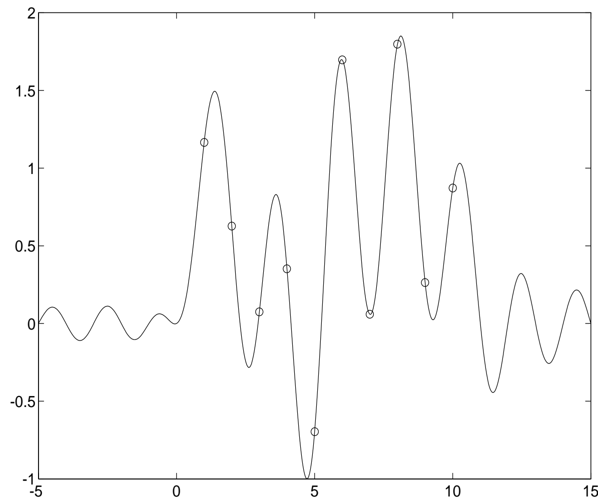
The space of functions bandlimited in the frequency band $\omega \in [-\pi,\pi]$ is spanned by the infinite (yet countable) set of sinc functions shifted by integers. Thus any such bandlimited function *g(t)* can be reconstructed from its samples at integer spacings:

$$g(t) = \sum_{n=-\infty}^{\infty} g(n)\mathrm{sinc}(t-n)$$

Example

Perform ideal bandlimited interpolation by assuming that the signal to be interpolated is 0 outside of the given time interval and that it has been sampled at exactly the Nyquist frequency:

```
t = (1:10)';              % a column vector of time samples
x = randn(size(t));       % a column vector of data
% ts is times at which to interpolate data
ts = linspace(-5,15,600)';
y = sinc(ts(:,ones(size(t))) - t(:,ones(size(ts)))')*x;
plot(t,x,'o',ts,y)
```



See Also

chirp	Swept-frequency cosine generator.
cos	Cosine of vector/matrix elements (see MATLAB Function Reference).
diric	Dirichlet or periodic sinc function.
gauspuls	Gaussian-modulated sinusoidal pulse generator.
pulstran	Pulse train generator.
rectpuls	Sampled aperiodic rectangle generator.
sawtooth	Sawtooth or triangle wave generator.
sin	Sine of vector/matrix elements (see MATLAB Function Reference).
square	Square wave generator.
tripuls	Sampled aperiodic triangle generator.

sos2ss

Purpose

Second-order section to state-space conversion.

Syntax

```
[A,B,C,D] = sos2ss(sos)
```

Description

`sos2ss` converts a second-order section representation of a given system to an equivalent state-space representation.

`[A,B,C,D] = sos2ss(sos)` converts the system `sos`, in second-order section form, to a single-input, single-output state-space representation:

$$\begin{aligned} x[n+1] &= \mathrm{A}x[n] + \mathrm{B}u[n] \\ y[n] &= \mathrm{C}x[n] + \mathrm{D}u[n] \end{aligned}$$

The discrete transfer function in second-order section form is given by

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

where L is the number of rows in `sos`. `sos` is a L-by-6 matrix organized as

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

The entries of `sos` must be real for proper conversion to state space. The returned matrix `A` is size N-by-N, where $N = 2L$-1, `B` is a length N-1 column vector, `C` is a length N-1 row vector, and `D` is a scalar.

Example

Compute the state-space representation of a simple second-order section form system:

```
sos = [1   1   1   1   0 -1;  -2   3   1   1  10   1];
[A,B,C,D] = sos2ss(sos)

A =

  -10.8990    -3.1463         0         0
    3.1463          0         0         0
   -9.8990    -2.8284    0.8990    0.3178
         0          0    0.3178         0

B =

     1
     0
     1
     0

C =

   19.7980     5.6569    1.2020    2.5106

D =

    -2
```

Algorithm

sos2ss first finds the zeros and poles of the second-order sections using roots, then uses zp2ss to find a state-space representation of the system.

See Also

sos2tf	Second-order section to transfer function conversion.
sos2zp	Second-order section to zero-pole-gain conversion.
ss2sos	State-space to second-order section conversion.
zp2sos	Zero-pole-gain to second-order section conversion.

sos2tf

Purpose

Second-order section to transfer function conversion.

Syntax

```
[b,a] = sos2tf(sos)
```

Description

`sos2tf` converts a second-order section representation of a given system to an equivalent transfer function representation.

`[b,a] = sos2tf(sos)` returns the numerator coefficients `b` and denominator coefficients `a` of the transfer function that describes a discrete-time system given by `sos` in second-order section form. The second-order section format of *H(z)* is given by

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

where L is the number of rows of `sos`. `sos` is an L-by-6 matrix which contains the coefficients of each second-order section stored in its rows:

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

Row vectors `b` and `a` contain the numerator and denominator coefficients of *H(z)* stored in descending powers of z:

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

Algorithm

`sos2tf` uses the `conv` function to multiply all of the numerator and denominator second-order polynomials together.

Example

Compute the transfer function representation of a simple second-order section form system:

```
sos = [1   1   1   1   0 -1;  -2   3   1   1  10   1];
[b,a] = sos2tf(sos)

b =

   -2      1      2      4      1

a =

     1     10      0    -10     -1
```

See Also

sos2ss	Second-order section to state-space conversion.
sos2zp	Second-order section to zero-pole-gain conversion.
ss2sos	State-space to second-order section conversion.
zp2sos	Zero-pole-gain to second-order section conversion.

sos2zp

Purpose

Second-order section to zero-pole-gain conversion.

Syntax

```
[z,p,k] = sos2zp(sos)
```

Description

`sos2zp` converts a second-order section representation of a given system to an equivalent zero-pole-gain representation.

`[z,p,k] = sos2zp(sos)` returns the zeros `z`, poles `p`, and gain `k` of the system given by `sos` in second-order section form. The second-order section format of *H(z)* is given by

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

where L is the number of rows of `sos`. `sos` is an L-by-6 matrix which contains the coefficients of each second-order section stored in its rows:

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

Column vectors `z` and `p` contain the zeros and poles of the transfer function *H(z)*:

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

where the orders N and M are determined by the matrix `sos`.

Example

Compute the poles, zeros, and gain of a simple system in second-order section form:

```
sos = [1  1  1  1  0 -1; -2  3  1  1 10  1];
[z,p,k] = sos2zp(sos)

z =

  -0.5000 + 0.8660i
  -0.5000 - 0.8660i
   1.7808
  -0.2808

p =

   -1.0000
    1.0000
   -9.8990
   -0.1010

k =

    -2
```

Algorithm

sos2zp finds the roots and poles of each second-order section using the roots command. sos2zp returns the roots and poles with conjugate pairs in consecutive locations, with the order of the pairs determined by their row in the sos matrix. The gain k is the product of the gains of the sections:

$$k = \prod_{k=1}^{L} \frac{b_{0k}}{a_{0k}}$$

See Also

sos2ss	Second-order section to state-space conversion.
sos2tf	Second-order section to transfer function conversion.
ss2sos	State-space to second-order section conversion.
zp2sos	Zero-pole-gain to second-order section conversion.

specgram

Purpose

Time-dependent frequency analysis (spectrogram).

Syntax

```
B = specgram(a)
B = specgram(a,nfft)
[B,f] = specgram(a,nfft,Fs)
[B,f,t] = specgram(a,nfft,Fs)
B = specgram(a,nfft,Fs,window)
B = specgram(a,nfft,Fs,window,noverlap)
specgram(a)
B = specgram(a,f,Fs,window,noverlap)
```

Description

`specgram` computes the windowed discrete-time Fourier transform of a signal using a sliding window. The spectrogram is the magnitude of this function.

`B = specgram(a)` calculates the spectrogram for the signal in vector `a`. This syntax uses the default values:

- `nfft = min(256,length(a))`
- `Fs = 2`
- `window = hanning(nfft)`
- `noverlap = length(window)/2`

`nfft` specifies the FFT length that `specgram` uses. This value determines the frequencies at which the discrete-time Fourier transform is computed. `Fs` is a scalar that specifies the sampling frequency. `window` specifies a windowing function and the number of samples `specgram` uses in its sectioning of vector `a`. `noverlap` is the number of samples by which the sections overlap. Any arguments that you omit from the end of the input parameter list use the default values shown above.

If `a` is real, `specgram` computes the discrete-time Fourier transform at positive frequencies only. If `n` is even, `specgram` returns `nfft/2+1` rows (including the zero and Nyquist frequency terms). If `n` is odd, `specgram` returns `nfft/2` rows. The number of columns in `B` is

```
k = fix((n-noverlap)/(length(window)-noverlap))
```

If `a` is complex, `specgram` computes the discrete-time Fourier transform at both positive and negative frequencies. In this case, `B` is a complex matrix with `nfft`

rows. Time increases linearly across the columns of B, starting with sample 1 in column 1. Frequency increases linearly down the rows, starting at 0.

B = specgram(a,nfft) uses the specified FFT length nfft in its calculations. Specify nfft as a power of 2 for fastest execution.

[B,f] = specgram(a,nfft,Fs) returns a vector f of frequencies at which the function computes the discrete-time Fourier transform. Fs has no effect on the output B; it is a frequency scaling multiplier.

[B,f,t] = specgram(a,nfft,Fs) returns frequency and time vectors f and t respectively. t is a column vector of scaled times, with length equal to the number of columns of B. t(j) is the earliest time at which the *j*-th window intersects a. t(1) is always equal to 0.

B = specgram(a,nfft,Fs,window) specifies a windowing function and the number of samples per section of the x vector. If you supply a scalar for window, specgram uses a Hanning window of that length. The length of the window must be less than or equal to nfft; specgram zero pads the sections if the length of the window exceeds nfft.

B = specgram(a,nfft,Fs,window,noverlap) overlaps the sections of x by noverlap samples.

You can use the empty matrix [] to specify the default value for any input argument. For example,

```
B = specgram(x,[],[],[],10000)
```

is equivalent to

```
B = specgram(x)
```

but with a sampling frequency of 10,000 Hz instead of the default 2 Hz.

specgram with no output arguments displays the scaled logarithm of the spectrogram in the current figure window using

```
imagesc(t,f,20*log10(abs(b))),axis xy,colormap(jet)
```

The `axis xy` mode displays the low-frequency content of the first portion of the signal in the lower-left corner of the axes. `specgram` uses `Fs` to label the axes according to true time and frequency.

`B = specgram(a,f,Fs,window,noverlap)` computes the spectrogram at the frequencies specified in `f`, using either the chirp *z*-transform (for more than 20 evenly spaced frequencies) or a polyphase decimation filterbank. `f` is a vector of frequencies in Hertz; it must have at least two elements.

Algorithm

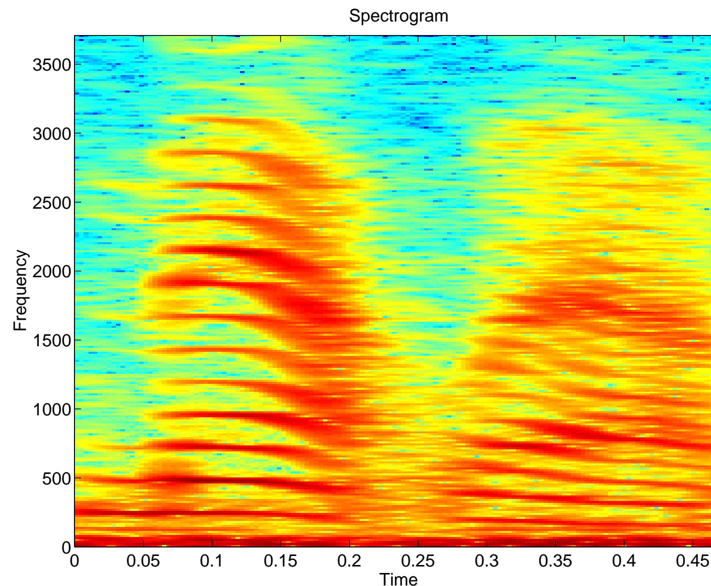
`specgram` calculates the spectrogram for a given signal as follows:

1. It splits the signal into overlapping sections and applies the window specified by the `window` parameter to each section.
2. It computes the discrete-time Fourier transform of each section with a length `nfft` FFT to produce an estimate of the short-term frequency content of the signal; these transforms make up the columns of `B`. `specgram` zero pads the windowed sections if `nfft > length(window)`, so the quantity `(length(window) - noverlap)` specifies by how many samples `specgram` shifts the window.
3. For real input, `specgram` truncates the spectrogram to the first `nfft/2 + 1` points for `nfft` even and `(nfft + 1)/2` for `nfft` odd.

Example

Plot the spectrogram of a digitized speech signal:

```
load mtlb
specgram(mtlb,512,Fs,kaiser(500,5),475)
```



Diagnostics

An appropriate diagnostic message is displayed when incorrect arguments are used:

```
Requires window's length to be no greater than the FFT length.
Requires NOVERLAP to be strictly less than the window length.
Requires positive integer values for NFFT and NOVERLAP.
Requires vector input.
```

See Also

`cohere`	Estimate magnitude squared coherence function between two signals.
`csd`	Estimate the cross spectral density (CSD) of two signals.
`psd`	Estimate the power spectral density (PSD) of a signal.
`tfe`	Transfer function estimate from input and output.

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. Pgs. 713-718.

[2] Rabiner, L.R., and R.W. Schafer. *Digital Processing of Speech Signals*. Englewood Cliffs, NJ: Prentice Hall, 1978.

Purpose Interactive digital signal processing tool.

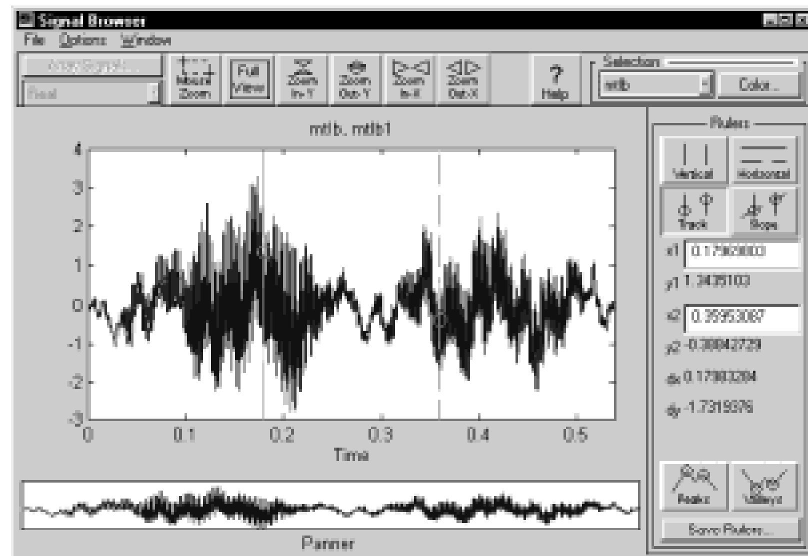
Syntax

```
sptool
```

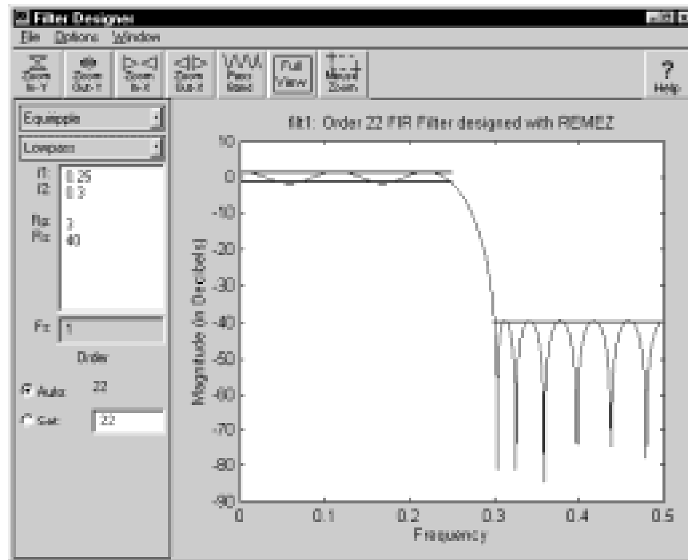
Description The `sptool` command invokes a suite of graphical user interface (GUI) tools that provides access to many of the signal, filter, and spectral analysis functions in the toolbox in a powerful, easy-to-use interactive signal display and exploration environment.

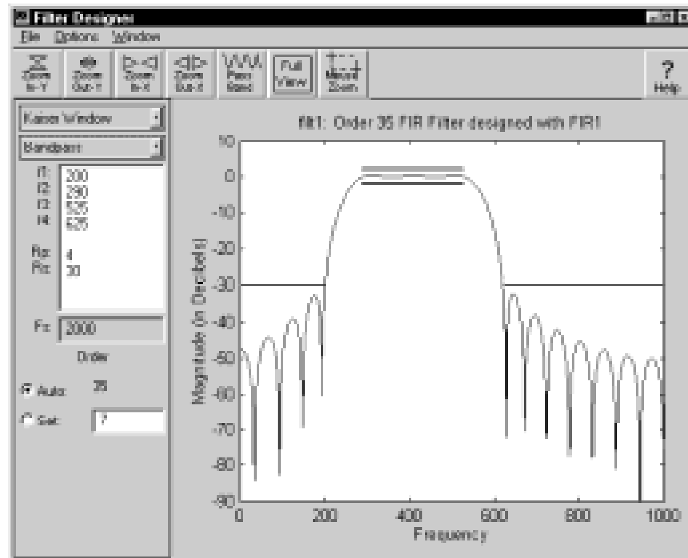
Using `sptool`, you can import, export, and manage signals, filters, and spectra. From `sptool`, you can activate its four integrated signal processing tools:

- The *Signal Browser*, for viewing, measuring, and analyzing time-domain information of imported signals:

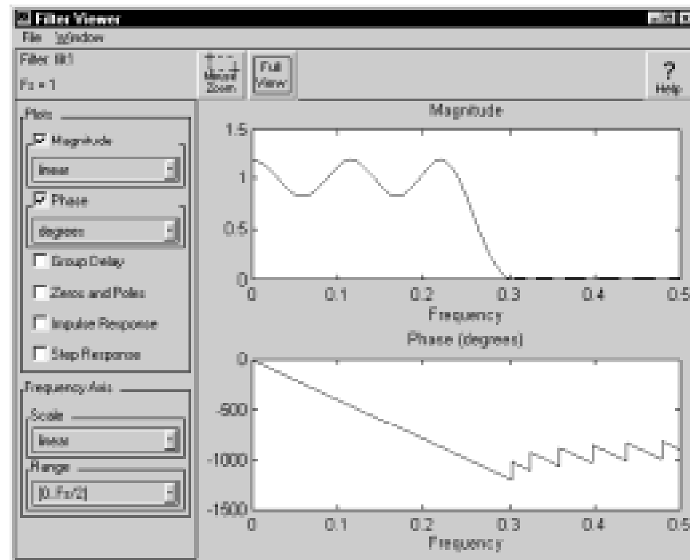


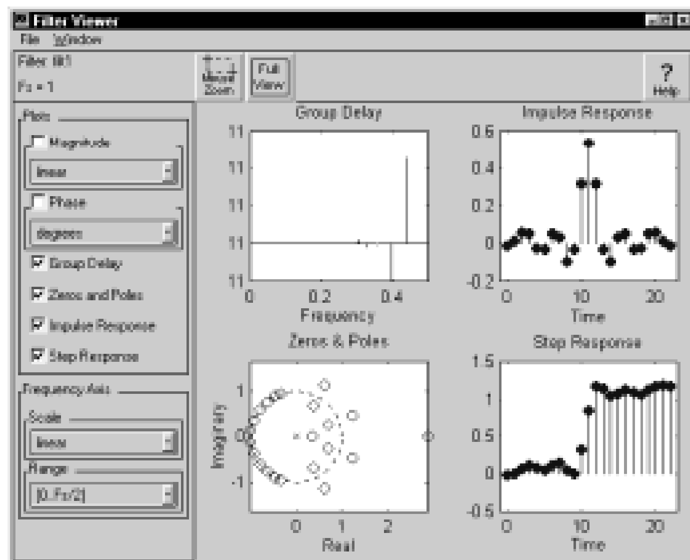
- The *Filter Designer*, for designing and editing FIR and IIR filters of various lengths and types, with standard (lowpass, highpass, bandpass, and bandstop) configurations:



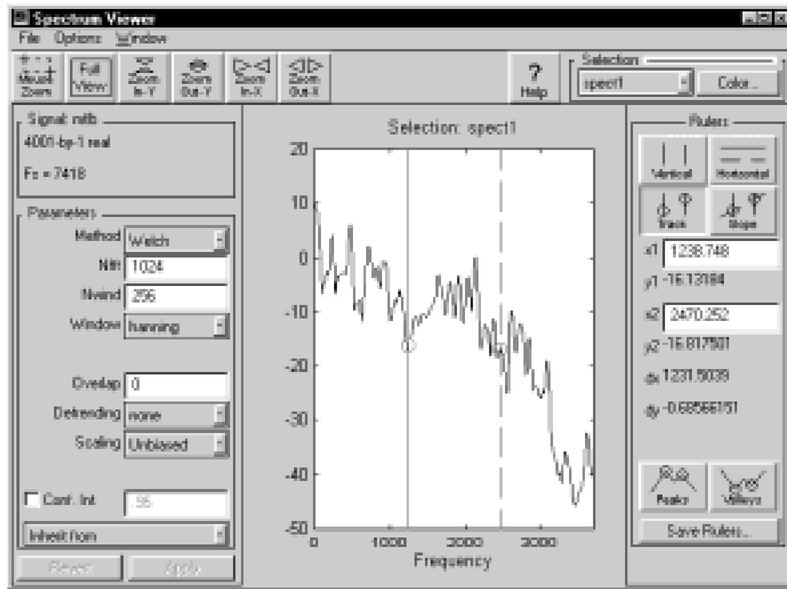


- The *Filter Viewer*, for viewing the characteristics of a designed or imported filter, including its magnitude response, phase response, group delay, pole-zero plot, impulse response, and step response:





- The *Spectrum Viewer*, for graphical analysis of frequency-domain data using a variety of methods of spectral density estimation, including Welch's method (PSD and CSD), the maximum entropy method (MEM), the multitaper method (MTM), and the MUSIC eigenvector method:



`sptool` opens an empty signal, filter, and spectrum management window. See Chapter 5, "Interactive Tools," for a full discussion of how to use `sptool`.

Purpose

Square wave generator.

Syntax

```
x = square(t)
x = square(t,duty)
```

Description

`square(t)` generates a square wave with period 2π for the elements of time vector `t`. `square(t)` is similar to `sin(t)`, but it creates a square wave with peaks of ± 1 instead of a sine wave.

`square(t,duty)` generates a square wave with specified duty cycle, `duty`. The *duty cycle* is the percent of the period in which the signal is positive.

See Also

`chirp`	Swept-frequency cosine generator.
`cos`	Cosine of vector/matrix elements (see MATLAB Function Reference).
`diric`	Dirichlet or periodic sinc function.
`gauspuls`	Gaussian-modulated sinusoidal pulse generator.
`pulstran`	Pulse train generator.
`rectpuls`	Sampled aperiodic rectangle generator.
`sawtooth`	Sawtooth or triangle wave generator.
`sin`	Sine of vector/matrix elements (see MATLAB Function Reference).
`sinc`	Sinc or $\sin(\pi t)/\pi t$ function.
`tripuls`	Sampled aperiodic triangle generator.

ss2sos

Purpose

State-space to second-order section conversion.

Syntax

```
sos = ss2sos(A,B,C,D)
sos = ss2sos(A,B,C,D,iu)
sos = ss2sos(A,B,C,D,'order')
sos = ss2sos(A,B,C,D,iu,'order')
```

Description

`ss2sos` converts a state-space representation of a given system to an equivalent second-order section representation.

`sos = ss2sos(A,B,C,D)` finds a matrix `sos` in second-order section form that is equivalent to the state-space system represented by input arguments `A`, `B`, `C` and `D`. The input system must be single input and real. `sos` is an L-by-6 matrix

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

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

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

`sos = ss2sos(A,B,C,D,iu)` specifies a scalar `iu` that determines which output of the state-space system `A`, `B`, `C`, `D` is used in the conversion. The default for `iu` is 1.

`sos = ss2sos(A,B,C,D,'order')` and

`sos = ss2sos(A,B,C,D,iu,'order')` specify the order of the rows in `sos`, where *order* is

- `down`, to order the sections so the first row of `sos` contains the poles closest to the unit circle
- `up`, to order the sections so the first row of `sos` contains the poles farthest from the unit circle (default)

Example

Find a second-order section form of a Butterworth lowpass filter:

```
[A,B,C,D] = butter(5,0.2);
sos = ss2sos(A,B,C,D)

sos =

    0.2330    0.2329         0    1.0000   -0.5095         0
    0.0647    0.1294    0.0647    1.0000   -1.0966    0.3554
    0.0851    0.1701    0.0851    1.0000   -1.3693    0.6926
```

Algorithm

ss2sos uses a four-step algorithm to determine the second-order section representation for an input state-space system:

1. It finds the poles and zeros of the system given by A, B, C and D.
2. It uses the function zp2sos, which first groups the zeros and poles into complex conjugate pairs using the cplxpair function. zp2sos then forms the second-order sections by matching the pole and zero pairs according to the following rules:
   a. Match the poles closest to the unit circle with the zeros closest to those poles.
   b. Match the poles next closest to the unit circle with the zeros closest to those poles.
   c. Continue until all of the poles and zeros are matched.

   ss2sos groups real poles into sections with the real poles closest to them in absolute value. The same rule holds for real zeros.
3. It orders the sections according to the proximity of the pole pairs to the unit circle. ss2sos normally orders the sections with poles closest to the unit circle last in the cascade. You can tell ss2sos to order the sections in the reverse order by specifying the 'down' flag.

   Putting "high Q" sections at the beginning of the cascade, by specifying the 'down' flag, reduces the sensitivity of the filter response to quantization noise near those poles. Putting "high Q" sections at the end of the cascade (the default) prevents reduction in signal power level early in the cascade. ss2sos orders all-zero sections according to the minimum of $|z_i|$ and $|z_i^{-1}|$,

where z_i, $i = 1, 2$, are the zeros in the section.References [1] and [2] provide a detailed discussion of section ordering.

4 ss2sos scales the sections so the maximum of the magnitude of the transfer function of the first N sections in cascade is less than 1:

$$\max_{|\omega| \le \pi} \left| \prod_{i=1}^{N} H_i(e^{j\omega}) \right| < 1, \quad N = 1, \ldots, L-1$$

subject to the constraint that the overall gain, k, stays the same:

$$k = \prod_{k=1}^{L} \frac{b_{0k}}{a_{0k}}$$

This scaling is an attempt to minimize overflow in some standard fixed point implementations of filtering.

Diagnostics

If there is more than one input to the system, ss2sos gives the following error message:

```
State-space system must have only one input.
```

See Also

sos2ss	Second-order section to state-space conversion.
sos2tf	Second-order section to transfer function conversion.
sos2zp	Second-order section to zero-pole-gain conversion.
zp2sos	Zero-pole-gain to second-order section conversion.

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. Pgs. 363-370.

[2] Jackson, L.B. *Digital Filters and Signal Processing*. Second Ed. Boston: Kluwer Academic Publishers, 1989. Pgs. 319-324.

Purpose State-space to transfer function conversion.

Syntax

```
[num,den] = ss2tf(a,b,c,d,iu)
```

Description `ss2tf` converts a state-space representation of a given system to an equivalent transfer function representation.

`[num,den] = ss2tf(a,b,c,d,iu)` returns the transfer function

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

of the system

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

from the `iu`-th input. Vector `den` contains the coefficients of the denominator in descending powers of *s*. The numerator coefficients are returned in array `num` with as many rows as there are outputs *y*. The function `tf2ss` is the inverse of `ss2tf`. `ss2tf` also works with systems in discrete time, in which case it returns the *z*-transform representation.

Algorithm `ss2tf` uses `poly` to find the characteristic polynomial *det(sI-A)* and the equality

$$H(s) = c(sI - A)^{-1}b = \frac{\det(sI - A + bc) - \det(sI - A)}{\det(sI - A)}$$

See Also

`ss2zp`	State-space to zero-pole-gain conversion.
`tf2ss`	Transfer function to state-space conversion.
`tf2zp`	Transfer function to zero-pole-gain conversion.
`zp2ss`	Zero-pole-gain to state-space conversion.
`zp2tf`	Zero-pole-gain to transfer function conversion.

ss2zp

Purpose

State-space to zero-pole-gain conversion.

Syntax

`[Z,p,k] = ss2zp(A,B,C,D,iu)`

Description

`ss2zp` converts a state-space representation of a given system to an equivalent zero-pole-gain representation. The zeros, poles, and gains of state-space systems represent the transfer function in factored form.

`[Z,p,k] = ss2zp(A,B,C,D,iu)` calculates the transfer function in factored form

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

of the system

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

from the `iu`-th input. Returned column vector `p` contains the pole locations of the denominator coefficients of the transfer function. Array `Z` contains the numerator zeros in its columns, with as many columns as there are outputs y. Column vector `k` contains the gains for each numerator transfer function.

The function `zp2ss` is the inverse of `ss2zp`. `ss2zp` also works with systems in discrete time, in which case it returns the z-transform representation. The input state-space system must be real.

Example

Here are two ways of finding the zeros, poles, and gains of a system:

```
num = [2 3];
den = [1 0.4 1];
[z,p,k] = tf2zp(num,den)

z =

   -1.5000
```

```
p =

    -0.2000 + 0.9798i
    -0.2000 - 0.9798i
k =

    2

[A,B,C,D] = tf2ss(num,den);
[z,p,k] = ss2zp(A,B,C,D,1)

z =

    -1.5000

p =

    -0.2000 + 0.9798i
    -0.2000 - 0.9798i

k =

    2
```

Algorithm

ss2zp finds the poles from the eigenvalues of the A array. The zeros are the finite solutions to a generalized eigenvalue problem:

```
z = eig([A B;C D], diag([ones(1,n) 0]);
```

In many situations this algorithm produces spurious large, but finite, zeros. ss2zp interprets these large zeros as infinite.

ss2zp finds the gains by solving for the first nonzero Markov parameters.

See Also

ss2tf	State-space to transfer function conversion.
tf2ss	Transfer function to state-space conversion.
zp2ss	Zero-pole-gain to state-space conversion.

References

[1] Laub, A.J., and B.C. Moore. “Calculation of Transmission Zeros Using QZ Techniques.” *Automatica* 14 (1978). Pg. 557.

stmcb

Purpose

Linear model using Steiglitz-McBride iteration.

Syntax

```
[b,a] = stmcb(x,nb,na)
[b,a] = stmcb(x,u,nb,na)
[b,a] = stmcb(x,nb,na,niter)
[b,a] = stmcb(x,u,nb,na,niter)
[b,a] = stmcb(x,nb,na,niter,ai)
[b,a] = stmcb(x,u,nb,na,niter,ai)
```

Description

Steiglitz-McBride iteration is an algorithm for finding an IIR filter with a prescribed time domain impulse response. It has applications in both filter design and system identification (parametric modeling).

`[b,a] = stmcb(x,nb,na)` finds the coefficients `b` and `a` of the system *b(z)/a(z)* with approximate impulse response `x`, exactly `nb` zeros, and exactly `na` poles.

`[b,a] = stmcb(x,u,nb,na)` finds the system coefficients `b` and `a` of the system that, given `u` as input, has `x` as output. `x` and `u` must be the same length.

`[b,a] = stmcb(x,nb,na,niter)` and

`[b,a] = stmcb(x,u,nb,na,niter)` use `niter` iterations. The default for `niter` is 5.

`[b,a] = stmcb(x,nb,na,niter,ai)` and

`[b,a] = stmcb(x,u,nb,na,niter,ai)` use the vector `ai` as the initial estimate of the denominator coefficients. If `ai` is not specified, `stmcb` uses the output argument from `[b,ai] = prony(x,0,na)` as the vector `ai`.

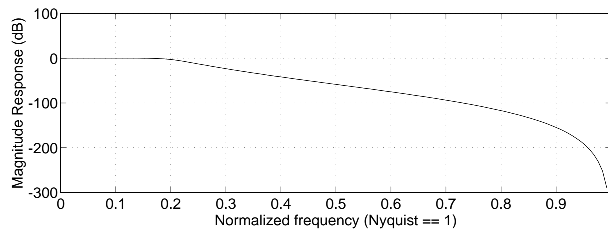
`stmcb` returns the IIR filter coefficients in length `nb+1` and `na+1` row vectors `b` and `a`. The filter coefficients are ordered in descending powers of *z*:

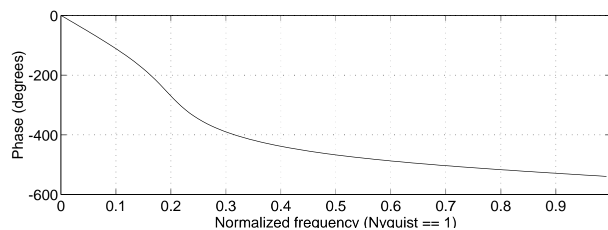
$$H(z)=\frac{B(z)}{A(z)}=\frac{b(1)+b(2)z^{-1}+\cdots+b(nb+1)z^{-nb}}{a(1)+a(2)z^{-1}+\cdots+a(na+1)z^{-na}}$$

Example

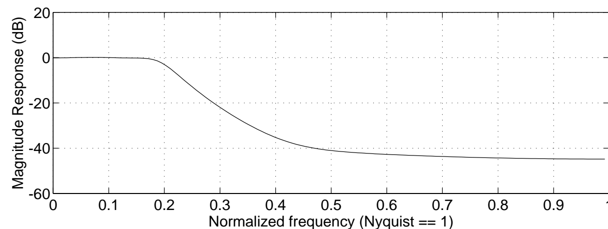
Approximate the impulse response of a Butterworth filter with a system of lower order:

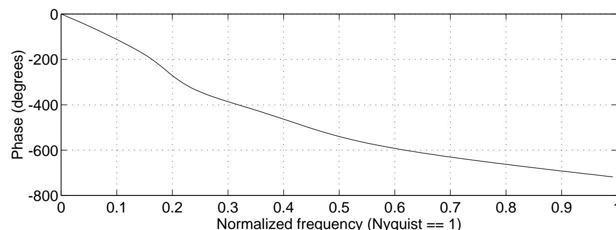
```
[b,a] = butter(6,0.2);
h = filter(b,a,[1 zeros(1,100)]);
freqz(b,a,128)
```






```
[bb,aa] = stmcb(h,4,4);
freqz(bb,aa,128)
```





Algorithm

`stmcb` attempts to minimize the squared error between the impulse response x' of $b(z)/a(z)$ and the input signal `x`:

$$\min_{a,b} \sum_{i=0}^{\infty} | x(i) - x'(i) |^2$$

`stmcb` iterates using two steps:

1. It prefilters `x` and `u` using $1/a(z)$.
2. It solves a system of linear equations for `b` and `a` using `\`.

`stmcb` repeats this process `niter` times. No checking is done to see if the `b` and `a` coefficients have converged in fewer than `niter` iterations.

Diagnostics

If `x` and `u` have different lengths, `stmcb` gives the following error message:

```
X and U must have same length.
```

See Also

`lpc`	Linear prediction coefficients.
`oe`	Compute the prediction error estimate of an output-error model (see *System Identification Toolbox User's Manual*).
`prony`	Prony's method for time domain IIR filter design.

References

[1] Steiglitz, K., and L.E. McBride. "A Technique for the Identification of Linear Systems." *IEEE Trans. Automatic Control*. Vol. AC-10 (1965). Pgs. 461-464.

[2] Ljung, L. *System Identification: Theory for the User*. Englewood Cliffs, NJ: Prentice Hall, 1987. Pg. 297.

Purpose Strip plot.

Syntax

```
strips(x)
strips(x,n)
strips(x,sd,Fs)
strips(x,sd,Fs,scale)
```

Description

`strips(x)` plots vector `x` in horizontal strips of length 250. If `x` is a matrix, `strips(x)` plots each column of `x`. The left-most column (column 1) is the top horizontal strip.

`strips(x,n)` plots vector `x` in strips that are each `n` samples long.

`strips(x,sd,Fs)` plots vector `x` in strips of duration `sd` seconds, given a sampling frequency of `Fs` samples per second.

`strips(x,sd,Fs,scale)` scales the vertical axes.

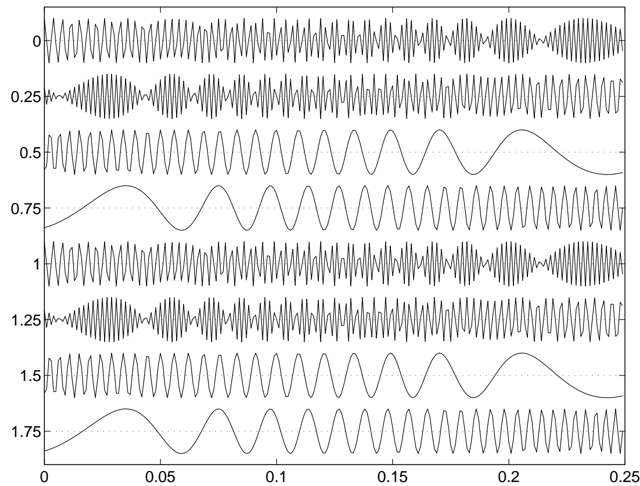
If `x` is a matrix, `strips(x,n)`, `strips(x,sd,Fs)`, and `strips(x,sd,Fs,scale)` plot the different columns of `x` on the same strip plot.

`strips` ignores the imaginary part of `x` if it is complex.

Example

Plot two seconds of a frequency modulated sinusoid in 0.25 second strips:

```
Fs = 1000;                            % sampling frequency
t = 0:1/Fs:2;                         % time vector
x = vco(sin(2*pi*t),[10 490],Fs);     % FM waveform
strips(x,0.25,Fs)
```



See Also

`plot`	Linear 2-D plot (see MATLAB Function Reference).
`stem`	Plot discrete sequence data (see MATLAB Function Reference).

Purpose Transfer function to lattice filter conversion.

Syntax

```
[k,v] = tf2latc(num,den)
k = tf2latc(1,den)
[k,v] = tf2latc(1,den)
k = tf2latc(num)
```

Description `[k,v] = tf2latc(num,den)` finds the lattice parameters `k` and the ladder parameters `v` for an IIR (ARMA) lattice-ladder filter, normalized by `den(1)`. Note that an error will be generated if any poles of the transfer function lie on the unit circle.

`k = tf2latc(1,den)` finds the lattice parameters `k` for an IIR all-pole (AR) lattice filter.

`[k,v] = tf2latc(1,den)` returns a scalar ladder coefficient `v`.

`k = tf2latc(num)` finds the lattice parameters `k` for an FIR (MA) lattice filter, normalized by `num(1)`.

The function `latc2tf` is the inverse of `tf2latc`.

See Also

`latc2tf`	Lattice filter to transfer function conversion.
`latcfilt`	Lattice and lattice-ladder filter implementation.
`poly2rc`	Reflection coefficients from polynomial coefficients.
`rc2poly`	Polynomial coefficients from reflection coefficients.

tf2ss

Purpose Transfer function to state-space conversion.

Syntax

```
[A,B,C,D] = tf2ss(num,den)
```

Description `tf2ss` converts a transfer function representation of a given system to an equivalent state-space representation.

`[A,B,C,D] = tf2ss(num,den)` finds a state-space representation:

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

given a system in transfer function form:

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

from a single input. Input vector `den` contains the denominator coefficients in descending powers of s. Array `num` contains the numerator coefficients with as many rows as there are outputs y. `tf2ss` returns the `A`, `B`, `C`, and `D` matrices in controller canonical form.

`tf2ss` also works for discrete systems, but you must pad the numerator with trailing zeros to make it the same length as the denominator.

The function `ss2tf` is the inverse of `tf2ss`.

Example Consider the system

$$H(s) = \frac{\begin{bmatrix} 2s+3 \\ s^2+2s+1 \end{bmatrix}}{s^2+0.4s+1}$$

To convert this system to state-space:

```
num = [0 2 3; 1 2 1];
den = [ 1 0.4 1];
[A,B,C,D] = tf2ss(num,den)
```

```
A =

   -0.4000   -1.0000
    1.0000         0

B =

     1
     0

C =

    2.0000    3.0000
    1.6000         0

D =

     0
     1
```

There is disagreement in the literature on naming conventions for the canonical forms. It is easy, however, to generate similarity transformations that convert to other forms. For example:

```
T = fliplr(eye(n));
A = T\A*T;
```

Algorithm

`tf2ss` writes the output in controller canonical form by inspection.

See Also

`ss2tf`	State-space to transfer function conversion.
`ss2zp`	State-space to zero-pole-gain conversion.
`tf2zp`	Transfer function to zero-pole-gain conversion.
`zp2ss`	Zero-pole-gain to state-space conversion.
`zp2tf`	Zero-pole-gain to transfer function conversion.

tf2zp

Purpose

Transfer function to zero-pole-gain conversion.

Syntax

```
[z,p,k] = tf2zp(num,den)
```

Description

`tf2zp` finds the zeros, poles, and gains of a system in polynomial transfer function form.

`[z,p,k] = tf2zp(num,den)` finds the single-input, multi-output (SIMO) factored transfer function form:

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

given a SIMO system in polynomial transfer function form:

$$\frac{num(s)}{den(s)} = \frac{num(1)s^{nn-1} + \cdots + num(nn-1)s + num(nn)}{den(1)s^{nd-1} + \cdots + den(nd-1)s + den(nd)}$$

Vector `den` specifies the coefficients of the denominator in descending powers of s. Matrix `num` indicates the numerator coefficients with as many rows as there are outputs. The zero locations are returned in the columns of matrix `z`, with as many columns as there are rows in `num`. The pole locations are returned in column vector `p` and the gains for each numerator transfer function in vector `k`.

`tf2zp` also works for discrete systems. The function `zp2tf` is the inverse of `tf2zp`.

Example

Find the zeros, poles, and gains of the system

$$H(s) = \frac{2s+3}{s^2+0.4s+1}$$

```
num = [2 3];
den = [1 0.4 1];
[z,p,k] = tf2zp(num,den)

z =

    -1.5000

p =

    -0.2000 + 0.9798i
    -0.2000 - 0.9798i

k =

    2
```

Algorithm

The system is converted to state-space using `tf2ss` and then to zeros, poles, and gains using `ss2zp`.

See Also

`ss2tf`	State-space to transfer function conversion.
`ss2zp`	State-space to zero-pole-gain conversion.
`tf2ss`	Transfer function to state-space conversion.
`zp2ss`	Zero-pole-gain to state-space conversion.
`zp2tf`	Zero-pole-gain to transfer function conversion.

tfe

Purpose

Transfer function estimate from input and output.

Syntax

```
Txy = tfe(x,y)
Txy = tfe(x,y,nfft)
[Txy,f] = tfe(x,y,nfft,Fs)
Txy = tfe(x,y,nfft,Fs,window)
Txy = tfe(x,y,nfft,Fs,window,noverlap)
Txy = tfe(x,y,...,'dflag')
tfe(x,y)
```

Description

`Txy = tfe(x,y)` finds a transfer function estimate `Txy` given input signal vector `x` and output signal vector `y`. The *transfer function* is the quotient of the cross spectrum of `x` and `y` and the power spectrum of `x`:

$$T_{xy}(f) = \frac{P_{xy}(f)}{P_{xx}(f)}$$

The relationship between the input `x` and output `y` is modeled by the linear, time-invariant transfer function `Txy`.

Vectors `x` and `y` must be the same length. `Txy = tfe(x,y)` uses the following default values:

- `nfft = min(256,(length(x))`
- `Fs = 2`
- `window = hanning(nfft)`
- `noverlap = 0`

`nfft` specifies the FFT length that `tfe` uses. This value determines the frequencies at which the power spectrum is estimated. `Fs` is a scalar that specifies the sampling frequency. `window` specifies a windowing function and the number of samples `tfe` uses in its sectioning of the `x` and `y` vectors. `noverlap` is the number of samples by which the sections overlap. Any arguments that omitted from the end of the parameter list use the default values shown above.

If `x` is real, `tfe` estimates the transfer function at positive frequencies only; in this case, the output `Txy` is a column vector of length `nfft/2+1` for `nfft` even and `(nfft+1)/2` for `n` odd. If `x` or `y` is complex, `tfe` estimates the transfer function for both positive and negative frequencies and `Txy` has length `nfft`.

`Txy = tfe(x,y,nfft)` uses the specified FFT length `nfft` in estimating the transfer function. Specify `nfft` as a power of 2 for fastest execution.

`[Txy,f] = tfe(x,y,nfft,Fs)` returns a vector `f` of frequencies at which `tfe` estimates the transfer function. `Fs` is the sampling frequency. `f` is the same size as `Txy`, so `plot(f,Txy)` plots the transfer function estimate versus properly scaled frequency. `Fs` has no effect on the output `Txy`; it is a frequency scaling multiplier.

`Txy = tfe(x,y,nfft,Fs,window)` specifies a windowing function and the number of samples per section of the `x` vector. If you supply a scalar for `window`, `Txy` uses a Hanning window of that length. The length of the window must be less than or equal to `nfft`; `tfe` zero pads the sections if the length of the window exceeds `nfft`.

`Txy = tfe(x,y,nfft,Fs,window,noverlap)` overlaps the sections of `x` by `noverlap` samples.

You can use the empty matrix `[]` to specify the default value for any input argument except `x` or `y`. For example,

```
Txy = tfe(x,y,[],[],kaiser(128,5))
```

uses 256 as the value for `nfft` and 2 as the value for `Fs`.

`Txy = tfe(x,y,...,'dflag')` specifies a detrend option, where *dflag* is

- `linear`, to remove the best straight-line fit from the prewindowed sections of `x` and `y`
- `mean`, to remove the mean from the prewindowed sections of `x` and `y`
- `none`, for no detrending (default)

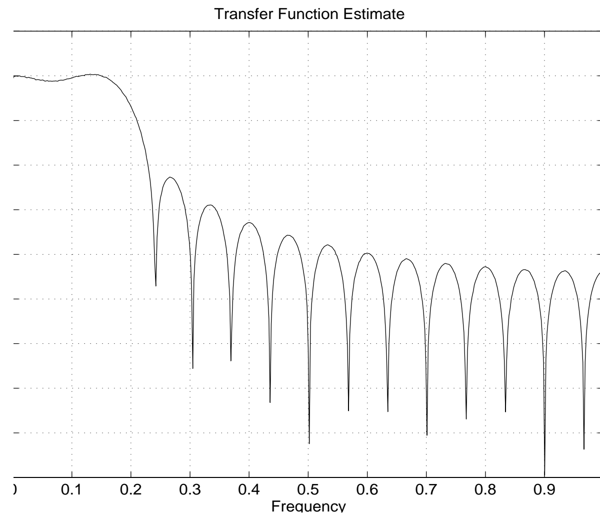
The *dflag* parameter must appear last in the list of input arguments. `tfe` recognizes a *dflag* string no matter how many intermediate arguments are omitted.

`tfe` with no output arguments plots the magnitude of the transfer function estimate in decibels versus frequency in the current figure window.

Example

Compute and plot the transfer function estimate between two colored noise sequences x and y:

```
h = fir1(30,0.2,boxcar(31));
x = randn(16384,1);
y = filter(h,1,x);
tfe(x,y,1024,[],[],512)
```



Algorithm

tfe uses a four-step algorithm:

1. It multiplies the detrended sections by window.
2. It takes the length nfft FFT of each section.
3. It averages the squares of the spectra of the x sections to form Pxx and averages the products of the spectra of the x and y sections to form Pxy.
4. It calculates Txy:

   ```
   Txy = Pxy./Pxx
   ```

Diagnostics

An appropriate diagnostic message is displayed when incorrect arguments are used:

```
Requires window's length to be no greater than the FFT length.
Requires NOVERLAP to be strictly less than the window length.
Requires positive integer values for NFFT and NOVERLAP.
Requires vector (either row or column) input.
Requires inputs X and Y to have the same length.
```

See Also

`etfe`	Compute empirical transfer function estimate and periodogram (see *System Identification Toolbox User's Guide*).
`cohere`	Estimate magnitude squared coherence function between two signals.
`csd`	Estimate the cross spectral density (CSD) of two signals.
`psd`	Estimate the power spectral density (PSD) of a signal.
`spa`	Perform spectral analysis for input-output data (see *System Identification Toolbox User's Guide*).

triang

Purpose

Triangular window.

Syntax

```
w = triang(n)
```

Description

`triang(n)` returns an `n`-point triangular window in the column vector `w`. The coefficients of a triangular window are

For `n` odd:

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

For `n` even:

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

The triangular window is very similar to a Bartlett window. The Bartlett window always ends with zeros at samples 1 and `n`, while the triangular window is nonzero at those points. For `n` odd, the center `n–2` points of `triang(n–2)` are equivalent to `bartlett(n)`.

See Also

`bartlett`	Bartlett window.
`blackman`	Blackman window.
`boxcar`	Rectangular window.
`chebwin`	Chebyshev window.
`hamming`	Hamming window.
`hanning`	Hanning window.
`kaiser`	Kaiser window.

Purpose Sampled aperiodic triangle generator.

Syntax

```
y = tripuls(T)
y = tripuls(T,w)
y = tripuls(T,w,s)
```

Description `y = tripuls(T)` returns a continuous, aperiodic, symmetric, unity-height triangular pulse at the times indicated in array `T`, centered about `T=0` and with a default width of 1.

`y = tripuls(T,w)` generates a triangle of width `w`.

`y = tripuls(T,w,s)` generates a triangle with skew `s`, where $-1 < s < 1$. When `s` is 0, a symmetric triangle is generated.

See Also

`chirp`	Swept-frequency cosine generator.
`cos`	Cosine of vector/matrix elements (see MATLAB Function Reference).
`diric`	Dirichlet or periodic sinc function.
`gauspuls`	Gaussian-modulated sinusoidal pulse generator.
`pulstran`	Pulse train generator.
`rectpuls`	Sampled aperiodic rectangle generator.
`sawtooth`	Sawtooth or triangle wave generator.
`sin`	Sine of vector/matrix elements (see MATLAB Function Reference).
`sinc`	Sinc or $\sin(\pi t)/\pi t$ function.
`square`	Square wave generator.

unwrap

Purpose Unwrap phase angles.

Syntax

```
p = unwrap(p)
```

Description `p = unwrap(p)` corrects the phase angles in vector `p` by adding multiples of $\pm 2\pi$, where needed, to smooth the transitions across branch cuts. When `p` is a matrix, `unwrap` corrects the phase angles down each column. The phase must be in radians.

This function is part of the standard MATLAB environment.

Limitations `unwrap` tries to detect branch cut crossings, but it can be fooled by sparse, rapidly changing phase values.

See Also

`angle`	Phase angle.

Purpose Apply FIR filter and perform sample rate conversion.

Syntax

```
yout = upfirdn(xin,h)
yout = upfirdn(xin,h,p)
yout = upfirdn(xin,h,p,q)
```

Description `upfirdn` performs a cascade of three operations:

1. Upsampling by `p` (zero inserting)
2. FIR filtering with the impulse response given in `h`
3. Downsampling by `q` (throwing away samples)

`upfirdn` has been implemented as a MEX-file for maximum speed, so only the outputs actually needed are computed. The FIR filter is usually a lowpass filter, which you must design using another function such as `remez` or `fir1`.

NOTE The function `resample` performs an FIR design using `fir1`, followed by rate changing implemented with `upfirdn`.

`yout = upfirdn(xin,h)` returns the output signal `yout`. If `yout` is a row or column vector, then it represents one signal; if `yout` is an array, then each column is a separate output. `xin` is the input signal. If `xin` is a row or column vector, then it represents one signal; if `xin` is an array, then each column is filtered. `h` is the impulse response of the FIR filter. If `h` is a row or column vector, then it represents one filter; if `h` is an array, then each column is a separate impulse response.

`yout = upfirdn(xin,h,p)` specifies the upsampling factor `p`. `p` is an integer with a default of 1.

`yout = upfirdn(xin,h,p,q)` specifies the downsampling factor `q`. `q` is an integer with a default of 1.

NOTE Since upfirdn performs convolution and rate changing, the yout signals have a different length than xin. The length of $y[n]$ is approximately p/q times the length of $x[n]$.

Remarks

Usually the inputs xin and the filter h are vectors, in which case only one output signal is produced. However, when these arguments are arrays, each column is treated as a separate signal or filter. Valid combinations are

1. xin is a vector and h is a vector.

   There is one filter and one signal, so the function convolves xin with h. The output signal yout is a row vector if xin is a row; otherwise, it is a column vector.
2. xin is an array and h is a vector.

   There is one filter and many signals, so the function convolves h with each column of xin. The resulting yout will be an array with the same number of columns as xin.
3. xin is a vector and h is an array.

   There are many filters and one signal, so the function convolves each column of h with xin. The resulting yout will be an array with the same number of columns as h.
4. xin is an array and h is an array, both with the same number of columns.

   There are many filters and many signals, so the function convolves corresponding columns of xin and h. The resulting yout is an array with the same number of columns as xin and h.

Examples

If both p and q are equal to 1 (that is, there is no rate changing), the result is ordinary convolution of two signals (equivalent to conv):

```
yy = upfirdn(xx,hh);
```

This example implements a seven-channel filter bank by convolving seven different filters with one input signal, then downsamples by five:

```
% Assume that hh is an L-by-7 array of filters.
yy = upfirdn(xx,hh,1,5);
```

Implement a rate change from 44.1 kHz (CD sampling rate) to 48 kHz (DAT rate), a ratio of 160/147. This requires a lowpass filter with cutoff frequency at $\omega_c = 2\pi/160$:

```
% Design lowpass filter with cutoff at 1/160th of Fs.
hh = fir1(300,2/160); % need a very long lowpass filter
yy = upfirdn(xx,hh,160,147);
```

In this example, the filter design and resampling are separate steps. Note that `resample` would do both steps as one.

Algorithm

`upfirdn` uses a polyphase interpolation structure. The number of multiply-add operations in the polyphase structure is approximately $(L_h L_x - pL_x)/q$ where L_h and L_x are the lengths of $h[n]$ and $x[n]$, respectively.

A more accurate flops count is computed in the program, but the actual count is still approximate. For long signals $x[n]$, the formula is quite often exact.

Diagnostics

There must be one output argument and at least two input arguments. If either of these conditions are violated, `upfirdn` gives the appropriate error message:

```
UPFIRDN needs at least two input arguments.
UPFIRDN should have exactly one output argument.
```

If the arrays are sparse, `upfirdn` gives the error message

```
H must be full numeric matrix.
```

When the input signals are in the columns of a matrix and there are multiple filters also in the columns of a matrix, the number of signals and filters must be the same. If they are not, `upfirdn` gives the error message

```
X and H must have the same number of columns, if more than one.
```

The arguments `p` and `q` must be integers. If they are not, `upfirdn` gives the error message

```
P and/or Q must be greater than zero
```

If the arguments `p` and `q` are not relatively prime, `upfirdn` gives the warning message

```
WARNING (upfirdn) p & q have common factor
```

See Also

`conv`	Convolution and polynomial multiplication.
`decimate`	Decrease the sampling rate for a sequence (decimation).
`filter`	Filter data with a recursive (IIR) or nonrecursive (FIR) filter.
`interp`	Interpolation FIR filter design.
`resample`	Change sampling rate by any factor.

References

[1] Crochiere, R.E., and L.R. Rabiner. *Multi-Rate Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1983. Pgs. 88-91.

[2] Crochiere, R.E. "A General Program to Perform Sampling Rate Conversion of Data by Rational Ratios." In *Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979. Pgs. 8.2-1 to 8.2-7.

Purpose

Voltage controlled oscillator.

Syntax

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

Description

`y = vco(x,Fc,Fs)` creates a signal that oscillates at a frequency determined by the real input vector or array `x` with sampling frequency `Fs`. `Fc` is the carrier or reference frequency; when `x` is 0, `y` is an `Fc` Hz cosine with amplitude 1 sampled at `Fs` Hz. `x` ranges from -1 to 1, where -1 corresponds to a 0 frequency output, 0 to `Fc`, and 1 to `2*Fc`. `y` is the same size as `x`.

`y = vco(x,[Fmin Fmax],Fs)` scales the frequency modulation range so that -1 and 1 values of `x` yield oscillations of `Fmin` Hz and `Fmax` Hz respectively. For best results, `Fmin` and `Fmax` should be in the range 0 to `Fs/2`.

By default, `Fs` is 1 and `Fc` is `Fs/4`.

If `x` is a matrix, `vco` produces a matrix whose columns oscillate according to the columns of `x`.

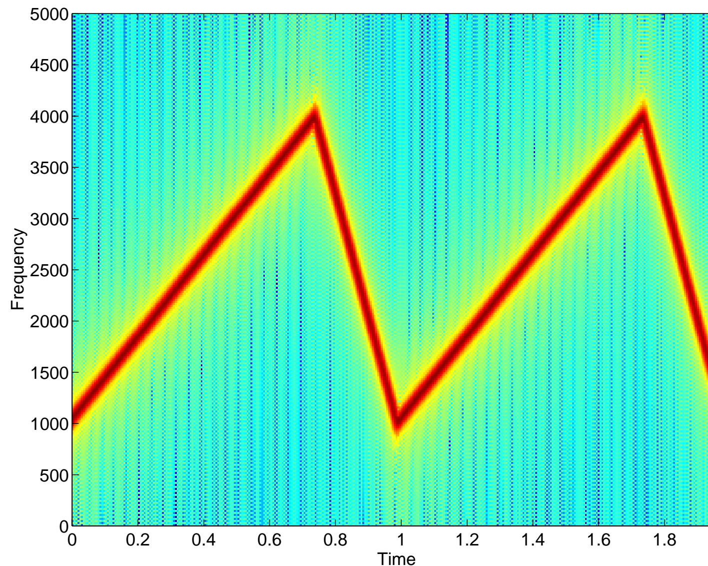
Example

Generate two seconds of a signal sampled at 10,000 samples/second whose instantaneous frequency is a triangle function of time:

```
Fs = 10000;
t = 0:1/Fs:2;
x = vco(sawtooth(2*pi*t,0.75),[.1 0.4]*Fs,Fs);
```

Plot the spectrogram of the generated signal.

```
specgram(x,512,Fs,kaiser(256,5),220)
```



Algorithm

vco performs FM modulation using the modulate function.

Diagnostics

If any values of x lie outside [-1,1], vco gives the following error message:

```
X outside of range [-1,1].
```

See Also

demod	Demodulation for communications simulation.
modulate	Modulation for communications simulation.

Purpose

Cross-correlation function estimate.

Syntax

```
c = xcorr(x,y)
c = xcorr(x)
c = xcorr(x,y,'option')
c = xcorr(x,y,maxlags)
c = xcorr(x,y,maxlags,'option')
[c,lags] = xcorr(x,y)
[c,lags] = xcorr(x,y,maxlags)
[c,lags] = xcorr(x,y,maxlags,'option')
```

Description

`xcorr` estimates the cross-correlation sequence of random processes. Autocorrelation is handled as a special case.

The true cross-correlation sequence is

$$\gamma_{xy}(m) = E\{x_n y^*_{n+m}\}$$

where x_n and y_n are stationary random processes, $-\infty < n < \infty$, and $E\{\}$ is the expected value operator. `xcorr` must estimate the sequence because, in practice, access is available to only a finite segment of the infinite-length random process.

`c = xcorr(x,y)` returns the cross-correlation sequence in a length $2N$-1 vector, where `x` and `y` are length N vectors.

`c = xcorr(x)` is the autocorrelation sequence for the vector `x`. Where `x` is an N-by-P matrix, `c = xcorr(X)` returns a matrix with $2N$-1 rows whose P^2 columns contain the cross-correlation sequences for all combinations of the columns of `X`.

By default, `xcorr` computes raw correlations with no normalization. For a length N vector:

$$c_{xy}(m) = \begin{cases} \sum_{n=0}^{N-|m|-1} x_{n+1} y^*_{n+m+1} & m \geq 0 \\ c^*_{yx}(-m) & m < 0 \end{cases}$$

The output vector `c` has elements given by `c(m)` = $c_{xy}(m\text{-}N)$, m=1,...,2N-1.

The correlation function requires normalization to estimate the function properly.

`c = xcorr(x,y,'option')` specifies a scaling option, where `'option'` is

- `biased`, for biased estimates of the cross-correlation function:

$$c_{xy,biased}(m) = \frac{1}{N} c_{xy}(m)$$

- `unbiased`, for unbiased estimates of the cross-correlation function:

$$c_{xy,unbiased}(m) = \frac{1}{N - |m|} c_{xy}(m)$$

- `coeff`, to normalize the sequence so the autocorrelations at zero lag are identically 1.0
- `none`, to use the raw, unscaled cross-correlations (default)

See reference [1] for more information on the properties of biased and unbiased correlation estimates.

`[c,lags] = xcorr(x,y)` where `x` and `y` are length `N` vectors, returns the cross-correlation sequence in a length `2N–1` vector and the output lags in the vector `[–N+1:N–1]`. That is, the maximum lag is `N–1`.

`[c,lags] = xcorr(x,y,maxlags)` where `x` and `y` are length `N` vectors, returns the cross-correlation sequence in a length `2*maxlags+1` vector `c`. `lags` is a vector of the lag indices where `c` was estimated, that is, `[–maxlags:maxlags]`.

`[c,lags] = xcorr(x,maxlags)` returns the autocorrelation sequence over the lag range `[–maxlags:maxlags]`.

`[c,lags] = xcorr(X,maxlags)` where `x` is an M-by-P matrix, is a matrix with `2*maxlags+1` rows whose P^2 columns contain the cross-correlation sequences for all combinations of the columns of `X`.

`[c,lags] = xcorr(x,maxlags,'option')` and

`[c,lags] = xcorr(x,y,maxlags,'option')` specifies both a maximum number of lags and a scaling option.

In all cases, `xcorr` gives an output such that the zeroth lag of the correlation vector is in the middle of the sequence, at element or row `maxlags+1` or at `N`.

Examples

The second output `lags` is useful when plotting. For example, the estimated autocorrelation of zero-mean Gaussian white noise $c_{ww}(m)$ can be displayed for $-10 \le m \le 10$ using

```
ww = randn(1000,1);
[c_ww,lags] = xcorr(ww,10,'coeff');
stem(lags,c_ww)
```

Swapping the `x` and `y` input arguments reverses (and conjugates) the output correlation sequence. For row vectors, the resulting sequences are reversed left to right; for column vectors, up and down. The following example illustrates this property (`mat2str` is used for a compact display of complex numbers):

```
x = [1,2i,3];  y = [4,5,6];
[c1,lags] = xcorr(x,y);
c1 = mat2str(c1,2), lags

c1 =

    [12+i*4.2e-16 15-i*8  22-i*10  5-i*12  6-i*1.4e-15]

lags =

    -2    -1     0     1     2

c2 = conj(fliplr(xcorr(y,x)));
c2 = mat2str(c2,2)

c2 =

    [12+i*4.2e-16 15-i*8  22-i*10  5-i*12  6-i*1.4e-15]
```

For the case where input argument `x` is a matrix, the output columns are arranged so that extracting a row and rearranging it into a square array

produces the cross-correlation matrix corresponding to the lag of the chosen row. For example, the cross-correlation at zero lag can be retrieved by

```
randn('seed',0)
X = randn(2,2);
[M,P] = size(X);
c = xcorr(X);
c0 = zeros(P); c0(:) = c(M,:)   % Extract zero-lag row

c0 =

    1.7500    0.3079
    0.3079    0.1293
```

You can calculate the matrix of correlation coefficients that the MATLAB function `corrcoef` generates by substituting

```
c = xcov(X,'coef')
```

in the last example. The function `xcov` subtracts the mean and then calls `xcorr`.

Use `fftshift` to move the second half of the sequence starting at the zeroth lag to the front of the sequence. `fftshift` swaps the first and second halves of a sequence.

Algorithm

For more information on estimating covariance and correlation functions, see [1] and [2].

Diagnostics

There must be at least one vector input argument; otherwise, `xcorr` gives the following error message:

```
1st arg must be a vector or matrix.
```

The string '*option*' must be the last argument; otherwise, `xcorr` gives the following error message:

```
Argument list not in correct order.
```

If the second argument was entered as a scalar, it is taken to be `maxlag` and no succeeding input can be a scalar. When the second argument is a vector, the first must also be a signal vector. The third argument, when present, must be

a scalar or a string. If they are not, `xcorr` gives the appropriate error message(s):

```
3rd arg is maxlag, 2nd arg cannot be scalar.
When b is a vector, a must be a vector.
Maxlag must be a scalar.
```

Normally the lengths of the vector inputs should be the same; if they are not, then the only allowable scaling option is `'none'`. If it is not, `xcorr` gives the following error message:

```
OPTION must be 'none' for different length vectors A and B.
```

See Also

`conv`	Convolution and polynomial multiplication.
`corrcoef`	Correlation coefficient matrix.
`cov`	Covariance matrix.
`xcorr2`	Two-dimensional cross-correlation.
`xcov`	Cross-covariance function estimate (equal to mean-removed cross-correlation).

References

[1] Bendat, J.S., and A.G. Piersol. *Random Data: Analysis and Measurement Procedures*. New York: John Wiley & Sons, 1971. Pg. 332.

[2] Oppenheim, A.V., and R.W. Schafer. *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975. Pgs. 63-67, 746-747, 839-842.

xcorr2

Purpose

Two-dimensional cross-correlation.

Syntax

```
C = xcorr2(A)
C = xcorr2(A, B)
```

Description

`C = xcorr2(A, B)` returns the cross-correlation of matrices `A` and `B` with no scaling. `xcorr2` is the two-dimensional version of `xcorr`. It has its maximum value when the two matrices are aligned so that they are shaped as similarly as possible.

`xcorr2(A)` is the autocorrelation matrix of input matrix `A`. It is identical to `xcorr2(A, A)`.

See Also

`conv2`	Two-dimensional convolution.
`filter2`	Two-dimensional digital filtering.
`xcorr`	Cross-correlation function estimate.

Purpose

Cross-covariance function estimate (equal to mean-removed cross-correlation).

Syntax

```
v = xcov(x,y)
v = xcov(x)
v = xcov(x,'option')
[c,lags] = xcov(x,y,maxlags)
[c,lags] = xcov(x,maxlags)
[c,lags] = xcov(x,y,maxlags,'option')
```

Description

xcov estimates the cross-covariance sequence of random processes. Autocovariance is handled as a special case.

The true cross-covariance sequence is the mean-removed cross-correlation sequence

$$\phi_{xy}(m) = E\{(x_n - m_x)(y_{n+m} - m_y)^*\}$$

where m_x and m_y are the mean values of the two stationary random processes, and $E\{\}$ is the expected value operator. xcov estimates the sequence because, in practice, access is available to only a finite segment of the infinite-length random process.

`v = xcov(x,y)` returns the cross-covariance sequence in a length $2N$-1 vector, where x and y are length N vectors.

`v = xcov(x)` is the autocovariance sequence for the vector x. Where x is an N-by-P array, `v = xcov(X)` returns am array with $2N$-1 rows whose P^2 columns contain the cross-covariance sequences for all combinations of the columns of X.

By default, xcov computes raw covariances with no normalization. For a length N vector:

$$c_{xy}(m) = \begin{cases} \displaystyle\sum_{n=0}^{N-|m|-1}\left(x(n) - \frac{1}{N}\sum_{i=0}^{N-1} x_i\right)\left(y^*_{n+m} - \frac{1}{N}\sum_{i=0}^{N-1} y^*_i\right) & m \ge 0 \\ c^*_{yx}(-m) & m < 0 \end{cases}$$

The output vector c has elements given by c(m) = $c_{xy}(m-N)$, $m=1,...,2N-1$.

The covariance function requires normalization to estimate the function properly.

`v = xcov(x,'option')` specifies a scaling option, where *option* is

- biased, for biased estimates of the cross-covariance function
- unbiased, for unbiased estimates of the cross-covariance function
- coeff, to normalize the sequence so the auto-covariances at zero lag are identically 1.0
- none, to use the raw, unscaled cross-covariances (default)

See [1] for more information on the properties of biased and unbiased correlation and covariance estimates.

`[c,lags] = xcov(x,y,maxlags)` where x and y are length m vectors, returns the cross-covariance sequence in a length 2*maxlags+1 vector c. lags is a vector of the lag indices where c was estimated, that is, [–maxlags:maxlags].

`[c,lags] = xcov(x,maxlags)` is the autocovariance sequence over the range of lags [–maxlags:maxlags].

`[c,lags] = xcov(x,maxlags)` where x is an m-by-p array, returns array c with 2*maxlags+1 rows whose P^2 columns contain the cross-covariance sequences for all combinations of the columns of X.

`[c,lags] = xcov(x,y,maxlags,'option')` specifies a scaling option, where *option* is the last input argument.

In all cases, xcov gives an output such that the zeroth lag of the covariance vector is in the middle of the sequence, at element or row maxlag+1 or at m.

Examples

The second output lags is useful when plotting. For example, the estimated autocovariance of uniform white noise $c_{ww}(m)$ can be displayed for $-10 \leq m \leq 10$ using

```
ww = randn(1000,1)      % generate uniform noise with mean = 1/2
[cov_ww,lags] = xcov(ww,10,'coeff');
stem(lags,cov_ww)
```

Algorithm

xcov computes the mean of its inputs, subtracts the mean, and then calls xcorr. For more information on estimating covariance and correlation functions, see [1] and [2].

Diagnostics

xcov does not check for any errors other than the correct number of input arguments. Instead, it relies on the error checking in xcorr, which it calls.

See Also

conv	Convolution and polynomial multiplication.
corrcoef	Correlation coefficient matrix.
cov	Covariance matrix.
xcorr	Cross-correlation function estimate.
xcorr2	Two-dimensional cross-correlation.

References

[1] Bendat, J.S., and A.G. Piersol. *Random Data: Analysis and Measurement Procedures*. New York: John Wiley & Sons, 1971. Pg. 332.

[2] Oppenheim, A.V., and R.W. Schafer. *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975. Pgs. 63-67, 746-747, 839-842.

yulewalk

Purpose Recursive digital filter design.

Syntax

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

Description

`yulewalk` designs recursive IIR digital filters using a least-squares fit to a specified frequency response.

`[b,a] = yulewalk(n,f,m)` returns row vectors `b` and `a` containing the `n+1` coefficients of the order `n` IIR filter whose frequency-magnitude characteristics approximately match those given in vectors `f` and `m`:

- `f` is a vector of frequency points, specified in the range between 0 and 1, where 1 corresponds to half the sample frequency (the Nyquist frequency).The first point of `f` must be 0 and the last point 1, with all intermediate points in increasing order. Duplicate frequency points are allowed, corresponding to steps in the frequency response.
- `m` is a vector containing the desired magnitude response at the points specified in `f`.
- `f` and `m` must be the same length.
- `plot(f,m)` displays the filter shape.

The output filter coefficients are ordered in descending powers of z,

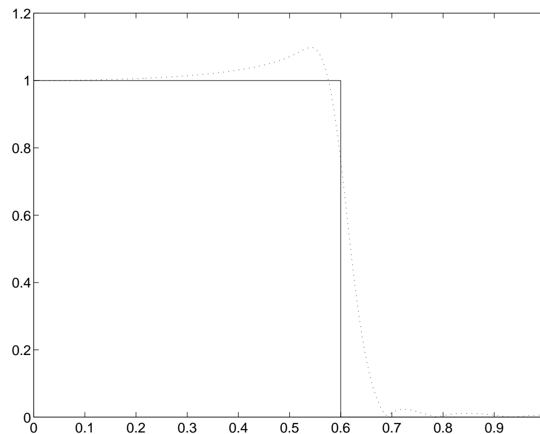
$$\frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}$$

When specifying the frequency response, avoid excessively sharp transitions from passband to stopband. You may need to experiment with the slope of the transition region to get the best filter design.

Example

Design an 8th-order lowpass filter and overplot the desired frequency response with the actual frequency response:

```
f = [0 0.6 0.6 1];
m = [1 1 0 0];
[b,a] = yulewalk(8,f,m);
[h,w] = freqz(b,a,128);
plot(f,m,w/pi,abs(h),'--')
```



Algorithm

yulewalk performs a least-squares fit in the time domain. It computes the denominator coefficients using modified Yule-Walker equations, with correlation coefficients computed by inverse Fourier transformation of the specified frequency response. To compute the numerator, yulewalk takes the following steps:

1. Computes a numerator polynomial corresponding to an additive decomposition of the power frequency response.
2. Evaluates the complete frequency response corresponding to the numerator and denominator polynomials.
3. Uses a spectral factorization technique to obtain the impulse response of the filter.
4. Obtains the numerator polynomial by a least-squares fit to this impulse response.

See Also

`butter`	Butterworth analog and digital filter design.
`cheby1`	Chebyshev type I filter design (passband ripple).
`cheby2`	Chebyshev type II filter design (stopband ripple).
`ellip`	Elliptic (Cauer) filter design.
`fir2`	Window-based finite impulse response filter design—arbitrary response.
`firls`	Least square linear-phase FIR filter design.
`maxflat`	Generalized digital Butterworth filter design.
`remez`	Parks-McClellan optimal FIR filter design.

References

[1] Friedlander, B., and B. Porat. “The Modified Yule-Walker Method of ARMA Spectral Estimation.” *IEEE Transactions on Aerospace Electronic Systems*. AES-20, No. 2 (March 1984). Pgs. 158-173.

Purpose

Zero-pole-gain to second-order section conversion.

Syntax

```
sos = zp2sos(z,p,k)
sos = zp2sos(z,p,k,'order')
```

Description

`zp2sos` converts a zero-pole-gain representation of a given system to an equivalent second-order section representation.

`sos = zp2sos(z,p,k)` finds a matrix `sos` in second-order section form equivalent to the zero-pole-gain system represented by input arguments `z`, `p`, and `k`. Vectors `z` and `p` contain the zeros and poles of the system *H(z)*, not necessarily in any order:

$$H(z) = k\frac{(z-z(1))(z-z(2))\dots(z-z(N))}{(p-p(1))(p-p(2))\dots(p-p(M))}$$

`k` is a scalar gain. The zeros and poles must be real or come in complex conjugate pairs. `sos` is an L-by-6 matrix:

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

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

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

The number of rows L of matrix `sos` is the maximum of the ceiling of $N/2$ and the ceiling of $M/2$, where N and M are the lengths of `z` and `p`, respectively.

`sos = zp2sos(z,p,k,'order')` specifies the order of the rows in `sos`, where *order* is

- `down`, to order the sections so the first row of `sos` contains the poles closest to the unit circle
- `up`, to order the sections so the first row of `sos` contains the poles farthest from the unit circle (default)

Example

Find a second-order section form of a Butterworth lowpass filter:

```
[z,p,k] = butter(5,0.2);
sos = zp2sos(z,p,k);
```

Algorithm

`zp2sos` uses a four-step algorithm to determine the second-order section representation for an input zero-pole-gain system:

1. It groups the zeros and poles into complex conjugate pairs using the `cplxpair` function.
2. It forms the second-order section by matching the pole and zero pairs according to the following rules:
   a. Match the poles closest to the unit circle with the zeros closest to those poles.
   b. Match the poles next closest to the unit circle with the zeros closest to those poles.
   c. Continue until all of the poles and zeros are matched.

   `zp2sos` groups real poles into sections with the real poles closest to them in absolute value. The same rule holds for real zeros.
3. It orders the sections according to the proximity of the pole pairs to the unit circle. `zp2sos` normally orders the sections with poles closest to the unit circle last in the cascade. You can tell `zp2sos` to order the sections in the reverse order by specifying the `down` flag.

   Putting "high Q" sections at the beginning of the cascade, by specifying the `down` flag, reduces the response sensitivity of the filter to quantization noise near those poles. Putting "high Q" sections at the end of the cascade (the default) prevents reduction in signal power level early in the cascade. `zp2sos` orders all zero sections according to the minimum of $|z_i|$ and $|z_i^{-1}|$ where

z_i (for $i = 1, 2$) are the zeros in the section. References [1] and [2] provide detailed discussions of section ordering.

4 `zp2sos` scales the sections so the maximum of the magnitude of the transfer function of the first N sections in cascade is less than 1:

$$\max_{|\omega| \le \pi} \left| \prod_{i=1}^{N} H_i(e^{j\omega}) \right| < 1, \quad N = 1, \ldots, L-1$$

subject to the constraint that the overall gain, `k`, stays the same:

$$\prod_{k=1}^{L} \frac{b_{0k}}{a_{0k}} = k$$

This scaling is an attempt to minimize overflow in some standard fixed point implementations of filtering.

See Also

`sos2ss`	Second-order section to state-space conversion.
`sos2tf`	Second-order section to transfer function conversion.
`sos2zp`	Second-order section to zero-pole-gain conversion.
`ss2sos`	State-space to second-order section conversion.

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. Pgs. 363-370.

[2] Jackson, L.B. *Digital Filters and Signal Processing*. Third Ed. Boston: Kluwer Academic Publishers, 1989. Pgs. 319-324.

zp2ss

Purpose

Zero-pole-gain to state-space conversion.

Syntax

```
[A,B,C,D] = zp2ss(Z,p,k)
```

Description

`zp2ss` converts a zero-pole-gain representation of a given system to an equivalent state-space representation. It forms state-space models from the zeros, poles, and gains of systems in transfer function form.

`[A,B,C,D] = zp2ss(Z,p,k)` finds a single input, multiple output, state-space representation

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

given a system in factored transfer function form

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

Column vector `p` specifies the pole locations, and array `Z` the zero locations with as many columns as there are outputs. The gains for each numerator transfer function are in vector `k`. The `A`, `B`, `C`, and `D` matrices are returned in controller canonical form.

`Inf` values may be used as place holders in `Z` if some columns have fewer zeros than others.

Algorithm

`zp2ss`, for single-input systems, groups complex pairs together into two-by-two blocks down the diagonal of the `A` matrix. This requires the zeros and poles to be real or come in complex conjugate pairs.

See Also

`ss2tf`	State-space to transfer function conversion.
`ss2zp`	State-space to zero-pole-gain conversion.
`tf2ss`	Transfer function to state-space conversion.
`tf2zp`	Transfer function to zero-pole-gain conversion.
`zp2tf`	Zero-pole-gain to transfer function conversion.

Purpose

Zero-pole-gain to transfer function conversion.

Syntax

```
[num,den] = zp2tf(Z,p,k)
```

Description

`zp2tf` forms transfer function polynomials from the zeros, poles, and gains of a system in factored form.

`[num,den] = zp2tf(z,p,k)` finds a rational transfer function:

$$\frac{num(s)}{den(s)} = \frac{num(1)s^{nn-1} + \ldots + num(nn-1)s + num(nn)}{den(1)s^{nd-1} + \ldots + den(nd-1)s + den(nd)}$$

given a system in factored transfer function form:

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

Column vector `p` specifies the pole locations, and array `Z` the zero locations, with as many columns as there are outputs. The gains for each numerator transfer function are in vector `k`. The zeros and poles must be real or come in complex conjugate pairs. The polynomial coefficients are returned in vectors: the denominator coefficients in row vector `den` and the numerator coefficients in matrix `num`, with as many rows as there are columns of `z`.

`Inf` values can be used as place holders in `Z` if some columns have fewer zeros than others.

Algorithm

The system is converted to transfer function form using `poly` with `p` and the columns of `Z`.

See Also

`ss2tf`	State-space to transfer function conversion.
`ss2zp`	State-space to zero-pole-gain conversion.
`tf2ss`	Transfer function to state-space conversion.
`tf2zp`	Transfer function to zero-pole-gain conversion.
`zp2ss`	Zero-pole-gain to state-space conversion.

zplane

Purpose

Zero-pole plot.

Syntax

```
zplane(z,p)
zplane(b,a)
[hz,hp,ht] = zplane(z,p)
```

Description

This function displays the poles and zeros of discrete-time systems.

`zplane(z,p)` plots the zeros specified in column vector `z` and the poles specified in column vector `p` in the current figure window. The symbol `'o'` represents a zero and the symbol `'x'` represents a pole. The plot includes the unit circle for reference. If `z` and `p` are arrays, `zplane` plots the poles and zeros in the columns of `z` and `p`, respectively, in different colors.

You can override the automatic scaling of `zplane` using

```
axis([xmin xmax ymin ymax])
```

or

```
set(gca,'ylim',[ymin ymax])
```

or

```
set(gca,'xlim',[xmin xmax])
```

after calling `zplane`. This is useful in the case where one or a few of the zeros or poles have such a large magnitude that the others are grouped around the origin and are thus hard to distinguish.

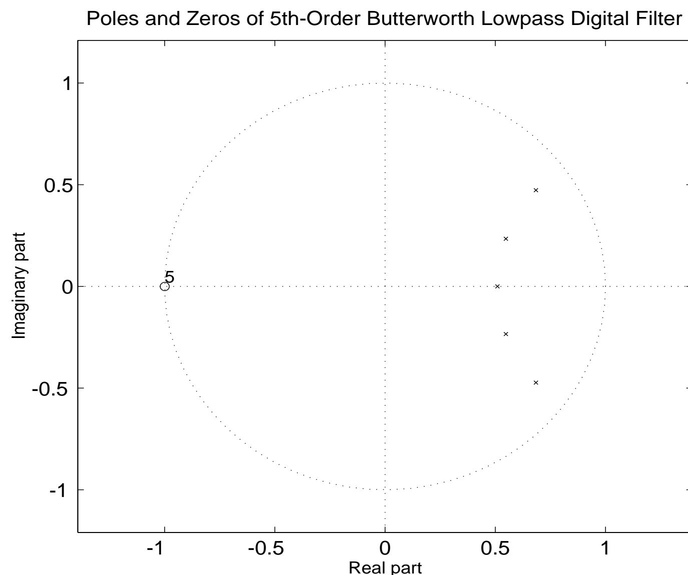
`zplane(b,a)` where `b` and `a` are row vectors, first uses `roots` to find the zeros and poles of the transfer function represented by numerator coefficients `b` and denominator coefficients `a`.

`[hz,hp,ht] = zplane(z,p)` returns vectors of handles to the zero lines, `hz`, and the pole lines, `hp`. `ht` is a vector of handles to the axes /unit circle line and to text objects, which are present when there are multiple zeros or poles. If there are no zeros or no poles, `hz` or `hp` is set to the empty matrix `[]`.

Examples

Plot the poles and zeros of a 5th-order Butterworth lowpass digital filter with cutoff frequency of 0.2:

```
[z,p,k] = butter(5,0.2);
zplane(z,p)
```



To generate the same plot with a transfer function representation of the filter:

```
[b,a] = butter(5,0.2);    % transfer function
zplane(b,a)
```

See Also

`freqz`	Frequency response of digital filters.

Index

Symbols

../signal/fft.htm 6-154, 6-159
../signal/filter.htm 6-154
.c.ss2tf 6-279
.c.tf2zp 6-290
.c.zp2tf 6-321

A

abs 6-2, **6-11**, 6-12, 6-150, 6-154
amdsb–sc 6-207
amdsb–tc 6-207
amplitude demodulation
  double side-band, suppressed carrier 6-93
  double side-band, transmitted carrier 6-93
  single side-band 6-93
amplitude modulation
  double side-band, suppressed carrier 6-207
  double side-band, transmitted carrier 6-207
  single side-band 6-207
amssb 6-207
analog filter
  Bessel 6-15
  Butterworth 6-28
  Chebyshev type I 6-49
  Chebyshev type II 6-54
  converting to digital 6-168
  design
    Bessel 6-15
    Butterworth 6-27
    Chebyshev type I 6-48
    Chebyshev type II 6-54
    elliptic 6-104
    inverse 6-177
  elliptic 6-105
  frequency response 6-148
  order estimation
    Butterworth 6-33
    Chebyshev type I 6-39
    Chebyshev type II 6-44
    elliptic 6-112
analog prototype
  Bessel filter 6-14
  Butterworth filter 6-26
  Chebyshev type I filter 6-37
  Chebyshev type II filter 6-42
  conversion to bandpass 6-192
  conversion to bandstop 6-195
  conversion to highpass 6-197
  conversion to lowpass 6-199
  elliptic filter 6-110
analytic signal 6-162
angl e 6-2, 6-11, **6-12**, 6-150, 6-154, 6-298
ar 6-203
attributes
  instananeous 6-162
autocorrelation 6-305
  two-dimensional 6-310
autocovariance 6-311

B

bandlimited interpolation 6-258
bandpass filter
  Bessel 6-15
  Butterworth 6-27, 6-29
  Chebyshev type I 6-48, 6-50
  Chebyshev type II 6-53, 6-55
  elliptic 6-104, 6-106
  FIR design
    with window method 6-131
  transformation from lowpass to 6-192

bandstop filter
Bessel 6-15
Butterworth 6-28, 6-29
Chebyshev type I 6-49, 6-50
Chebyshev type II 6-54, 6-55
elliptic 6-105, 6-106
FIR design
with window method 6-131
transformation from lowpass to 6-195
bartlett 6-7, **6-13**, 6-24, 6-25, 6-47, 6-160, 6-161, 6-183, 6-296
compared to triang 6-13
Bartlett window
coefficients 6-13
Bessel filter
analog 6-15
analog prototype 6-14
bandpass configuration
analog 6-15
bandstop configuration
analog 6-15
highpass configuration
analog 6-15
limitations 6-17
lowpass configuration
analog 6-15
besselap 6-9, **6-14**, 6-18, 6-26, 6-37, 6-42, 6-110
besself 6-4, 6-14, **6-15**, 6-31, 6-52, 6-57, 6-109
bilinear 6-10, **6-19**, 6-169, 6-194, 6-196, 6-198, 6-200
bilinear transformation 6-19
output representation 6-20
prewarping 6-19
blackman 6-7, 6-13, **6-24**, 6-25, 6-47, 6-160, 6-161, 6-183, 6-296
Blackman window 6-24
boxcar 6-7, 6-13, 6-24, **6-25**, 6-47, 6-160, 6-161, 6-183, 6-296
buttap 6-9, 6-14, **6-26**, 6-31, 6-37, 6-42, 6-110
butter 6-4, 6-18, 6-26, **6-27**, 6-35, 6-52, 6-57, 6-109, 6-135, 6-205, 6-227, 6-244, 6-316
Butterworth filter
analog 6-28
analog prototype 6-26
bandpass configuration
analog 6-29
digital 6-27
bandstop configuration
analog 6-29
digital 6-28
design 6-27
digital 6-27
highpass configuration
analog 6-29
digital 6-28
limitations 6-31
lowpass configuration
analog 6-28
digital 6-27
order estimation 6-32
buttord 6-5, 6-31, **6-32**, 6-41, 6-46, 6-114, 6-247

C

canonical forms 6-289
carrier frequency 6-207, 6-303
carrier signal 6-93
Cauer filter. *See* elliptic filter
cceps 6-8, **6-36**, 6-159, 6-238
cheb1ap 6-9, 6-14, 6-26, **6-37**, 6-42, 6-52, 6-110
cheb1ord 6-5, 6-35, **6-38**, 6-46, 6-52, 6-114, 6-247
cheb2ap 6-9, 6-14, 6-26, 6-37, **6-42**, 6-57, 6-110

cheb2ord 6-5, 6-35, 6-41, **6-43**, 6-57, 6-114, 6-247
chebwin 6-7, 6-13, 6-24, 6-25, **6-47**, 6-160, 6-161, 6-183, 6-296
cheby1 6-4, 6-18, 6-31, 6-37, 6-41, 6-46, **6-48**, 6-57, 6-109, 6-135, 6-227, 6-244, 6-316
cheby2 6-4, 6-18, 6-31, 6-42, 6-52, **6-53**, 6-109, 6-135, 6-227, 6-244, 6-316
Chebyshev error minimization 6-240
Chebyshev type I filter
analog 6-49
analog prototype 6-37
bandpass configuration
analog 6-50
digital 6-48
bandstop configuration
analog 6-50
digital 6-49
design 6-48
digital 6-48
highpass configuration
analog 6-50
digital 6-49
lowpass configuration
analog 6-49
digital 6-48
order estimation 6-38
Chebyshev type II filter
analog 6-54
analog prototype 6-42
bandpass configuration
analog 6-55
digital 6-53
bandstop configuration
analog 6-55
digital 6-54
design
Chebyshev type II 6-53
digital 6-53
highpass configuration
analog 6-55
digital 6-54
limitations 6-52
lowpass configuration
analog 6-54
digital 6-53
order estimation 6-43
Chebyshev window 6-47
chirp 6-2, **6-58**, 6-156, 6-236, 6-239, 6-256, 6-259, 6-275, 6-297
chirp *z*-transform 6-85
for narrowband frequency analysis 6-85
coefficients
correlation 6-70
linear prediction
reflection 6-223, 6-237
cohere 6-6, **6-61**, 6-83, 6-231, 6-269, 6-295
coherence 6-61
communications simulation 6-93, 6-207
complex conjugate 6-72
complex numbers
grouping by conjugate 6-72
control systems 6-172
conv 6-2, **6-65**, 6-67, 6-69, 6-92, 6-302, 6-309, 6-313
conv2 6-2, 6-65, **6-66**, 6-69, 6-126, 6-310
convmtx 6-3, 6-65, **6-68**, 6-96, 6-255
convn 6-65, 6-67, 6-69
convolution
and filtering 6-126
convolution matrix 6-68
defined 6-65
two-dimensional 6-66
obtaining subsection 6-66
convolution matrix 6-68

example 6-68
corrcoef 6-6, **6-70**, 6-71, 6-309, 6-313
correlation
coefficient matrix 6-70
cos 6-156, 6-236, 6-239, 6-256, 6-259, 6-275, 6-297
cov 6-6, 6-70, **6-71**, 6-309, 6-313
covariance
matrix 6-71
cplxpair 6-8, **6-72**
cremez 6-5, **6-73**, 6-244
cross spectral density 6-80
cross-correlation 6-305
two-dimensional 6-310
cross-covariance 6-311
csd 6-6, 6-64, **6-80**, 6-231, 6-269, 6-295
cutoff frequency
defined 6-15
czt 6-6, **6-85**

D

dct 6-6, **6-87**, 6-118, 6-165
dct2 6-88, 6-165
decimate 6-8, **6-89**, 6-174, 6-176, 6-252, 6-302
decimation 6-89
FIR filter for 6-175
deconv 6-8, 6-65, 6-67, **6-92**, 6-255
deconvolution 6-92
demod 6-8, **6-93**, 6-209, 6-304
demodulation 6-93
methods 6-93
detrend 6-8, **6-95**
dftmtx 6-6, 6-69, **6-96**, 6-118
differentiator 6-143, 6-241
digital filter
Butterworth 6-27
Chebyshev type I 6-48
Chebyshev type II 6-53
elliptic 6-104
group delay 6-157
identification from frequency data 6-180
implementation 6-120, 6-123
FFT-based (FIR) 6-120
impulse response 6-170
order estimation
Butterworth 6-32
Chebyshev type I 6-38
Chebyshev type II 6-43
elliptic 6-111
equiripple FIR 6-245
phase delay 6-157
two-dimensional 6-126
zero-phase 6-127
diric 6-2, 6-60, **6-97**, 6-156, 6-236, 6-239, 6-256, 6-259, 6-275, 6-297
Dirichlet function 6-97
discrete cosine transform 6-87
inverse 6-165
discrete Fourier transform 6-115
applications 6-115
inverse 6-166
matrix 6-96
two-dimensional 6-167
matrix 6-96
two-dimensional 6-119
discretization 6-168
dpss 6-8, **6-98**, 6-100, 6-101, 6-102, 6-103, 6-216
dpssclear 6-8, 6-99, **6-100**, 6-101, 6-102, 6-103
dpssdir 6-8, 6-99, 6-100, **6-101**, 6-102, 6-103
dpssload 6-8, 6-99, 6-100, 6-101, **6-102**, 6-103
dpsssave 6-8, 6-99, 6-100, 6-101, 6-102, **6-103**

E

`ellip` 6-4, 6-18, 6-31, 6-52, 6-57, **6-104**, 6-110, 6-114, 6-135, 6-227, 6-244, 6-316
`ellipap` 6-9, 6-14, 6-26, 6-37, 6-42, 6-109, **6-110**
`ellipord` 6-5, 6-35, 6-41, 6-46, 6-109, **6-111**, 6-247
elliptic filter
- analog 6-105
- analog prototype 6-110
- bandpass configuration
  - analog 6-106
  - digital 6-104
- bandstop configuration
  - analog 6-106
  - digital 6-105
- design 6-104
- digital 6-104
- highpass configuration
  - analog 6-106
  - digital 6-105
- limitations 6-108
- lowpass configuration
  - analog 6-105
  - digital 6-104
- order estimation 6-111

equiripple characteristics
- elliptic filter 6-104, 6-110
- from Parks-McClellan design 6-240

`etfe` 6-295

F

FFT 6-115
- prime factor algorithm 6-117
- radix-2 algorithm 6-117
- two-dimensional 6-119

`fft` 6-6, 6-36, 6-86, 6-88, 6-96, **6-115**, 6-119, 6-122, 6-154, 6-159, 6-163, 6-164, 6-166, 6-167, 6-238
- execution time 6-118
- prime factor algorithm 6-117
- radix-2 algorithm 6-117
- rearranging output 6-122

`fft2` 6-6, 6-118, **6-119**, 6-122, 6-166, 6-167
`fftfilt` 6-2, **6-120**, 6-125, 6-127
- compared to `filter` 6-120

`fftn` 6-167
`fftshift` 6-6, 6-118, 6-119, **6-122**, 6-166, 6-167
filter
- analog prototype 6-14, 6-26, 6-37, 6-42, 6-110
- Butterworth 6-27
- Chebyshev type I 6-48
- Chebyshev type II 6-53
- design
  - FIR 6-240
  - inverse 6-177, 6-180
- elliptic 6-104
- identification from frequency data 6-177
- implementation 6-120, 6-123
- median 6-206
- minimum phase 6-225
- order 6-32, 6-38, 6-43, 6-111
- two-dimensional 6-126

`filter` 6-2, 6-65, 6-92, 6-118, 6-121, **6-123**, 6-127, 6-129, 6-132, 6-154, 6-190, 6-205, 6-206, 6-302
- compared to `fftfilt` 6-120
- initial conditions 6-128

Filter Designer 6-272
Filter Viewer 6-273
`filter2` 6-3, 6-67, 6-125, **6-126**, 6-127, 6-310
`filtfilt` 6-3, 6-121, 6-125, **6-127**, 6-129
`filtic` 6-3, 6-125, **6-128**

FIR filter
arbitrary frequency response 6-133
design
decimation 6-175
interpolation 6-175
least squares method 6-142
linear phase 6-142
multiband frequency response 6-133
Parks-McClellan method 6-240
window method 6-130
differentiator 6-143, 6-241
Hilbert transformer 6-143, 6-241
implementation 6-123
FFT-based 6-120
overlap-add method 6-120
linear phase 6-240
order estimation, `remez` function 6-245
types 6-145, 6-242
FIR filter design
arbitrary responses
complex filters 6-73
nonlinear phase 6-73
`fir1` 6-5, **6-130**, 6-135, 6-146, 6-188, 6-244, 6-252
fir2 6-244
`fir2` 6-5, 6-132, **6-133**, 6-146, 6-244, 6-316
`fircls` 6-5, 6-132, **6-136**, 6-141, 6-244
`fircls1` 6-5, 6-132, 6-138, **6-139**, 6-244
`firls` 6-5, 6-132, 6-138, 6-141, **6-142**, 6-147, 6-244, 6-316
filter characteristics 6-145
`firrcos` 6-5, 6-146, **6-147**, 6-244
Fourier transform. *See* discrete Fourier transform, FFT
`freq2` 6-154
`freqs` 6-3, **6-148**, 6-179, 6-182
`freqspace` **6-151**
frequency 6-240
carrier 6-207, 6-303
prewarping 6-19
transformation 6-192, 6-195, 6-197, 6-199
vector 6-133, 6-136, 6-314
frequency analysis
time-dependent 6-266
frequency demodulation 6-94
frequency modulation 6-208
frequency response
arbitrary 6-133
inverse 6-177
spacing 6-151
frequency transformation
lowpass to bandpass 6-192
lowpass to bandstop 6-195
lowpass to highpass 6-197
lowpass to lowpass 6-199
frequency vector 6-240
`freqz` 6-3, 6-86, 6-118, 6-132, 6-150, 6-151, **6-152**, 6-159, 6-179, 6-182, 6-205, 6-323
spacing 6-151

G

`gauspuls` 6-2, 6-60, 6-97, **6-155**, 6-236, 6-239, 6-256, 6-259, 6-275, 6-297
Gauss-Newton method 6-179, 6-182
group delay 6-157
`grpdelay` 6-3, **6-157**

H

`hamming` 6-7, 6-13, 6-24, 6-25, 6-47, **6-160**, 6-161, 6-183, 6-296
Hamming window 6-160

hanning 6-7, 6-13, 6-24, 6-25, 6-47, 6-160, **6-161**, 6-183, 6-296
Hanning window 6-161
highpass filter
  Bessel 6-15
  Butterworth 6-28, 6-29
  Chebyshev type I 6-49, 6-50
  Chebyshev type II 6-54, 6-55
  elliptic 6-105, 6-106
  FIR design
    with window method 6-132
  transformation from lowpass to 6-197
hilbert 6-6, 6-36, 6-159, **6-162**, 6-164, 6-238
Hilbert transform 6-162
Hilbert transformer 6-143, 6-241

I

icceps 6-9, 6-159, **6-164**, 6-238
idct 6-6, 6-88, **6-165**
idct2 6-88, 6-165
ifft 6-6, 6-118, 6-119, 6-163, 6-165, **6-166**, 6-167
ifft2 6-6, 6-119, 6-166, **6-167**
ifftn 6-167
IIR filter
  design
    Levinson-Durbin recursion 6-191, 6-201
    Prony's method 6-226
    Steiglitz-McBride iteration 6-282
    Yule-Walker 6-314
  implementation 6-123
image processing 6-66
impinvar 6-10, 6-23, **6-168**, 6-194, 6-196, 6-198, 6-200
impulse 6-172
impulse invariance 6-168
impulse response 6-170
impz 6-3, 6-154, **6-170**
initial conditions 6-128
instantaneous attributes 6-162
interactive tools
  Filter Designer 6-272
  Filter Viewer 6-273
  Signal Browser 6-271
  Spectrum Viewer 6-274
  sptool 6-271
interp 6-9, 6-91, **6-173**, 6-176, 6-252, 6-302
interp1 6-174, 6-252
interpolation 6-173
  FIR filter design 6-175
intfilt 6-5, **6-175**, 6-252
inverse discrete cosine transform 6-165
inverse discrete Fourier transform 6-166
  matrix 6-96
  two-dimensional 6-167
inverse filter design 6-180, 6-226
  analog 6-177
  digital 6-180
invfreqs 6-8, 6-150, **6-177**, 6-182
invfreqz 6-8, 6-151, 6-154, 6-179, **6-180**, 6-227

K

kaiser 6-7, 6-13, 6-24, 6-25, 6-47, 6-160, 6-161, **6-183**, 6-188, 6-252, 6-296
Kaiser window 6-183
  beta parameter 6-183
kaiserord 6-5, 6-132, 6-183, **6-184**, 6-247

L

Lagrange interpolation filter 6-175
latc2tf 6-3, **6-189**, 6-190, 6-224, 6-237, 6-287
latcfilt 6-3, 6-189, **6-190**, 6-224, 6-237, 6-287

least squares method, FIR filter design 6-142
    filter characteristics 6-145
levinson 6-8, **6-191**, 6-203, 6-227
Levinson-Durbin recursion 6-191
linear phase 6-142
    filter design 6-240
linear prediction coefficients 6-201
linear trend
    removing from sequence 6-95
logspace 6-150, 6-154
lowpass filter
    Bessel 6-15
    Butterworth 6-27, 6-28
    Chebyshev type I 6-48, 6-49
    Chebyshev type II 6-53, 6-54
    elliptic 6-104, 6-105
    for decimation 6-89
    for interpolation 6-173
    translation of cutoff frequency 6-199
lp2bp 6-10, 6-23, 6-169, **6-192**, 6-196, 6-198, 6-200
lp2bs 6-10, 6-23, 6-169, 6-194, **6-195**, 6-198, 6-200
lp2hp 6-10, 6-23, 6-169, 6-194, 6-196, **6-197**, 6-200
lp2lp 6-10, 6-23, 6-169, 6-194, 6-196, 6-198, **6-199**
lpc 6-8, 6-191, **6-201**, 6-213, 6-222, 6-227, 6-284
LPC. *See* linear prediction coefficients

M

magnitude
    vector 6-133, 6-136, 6-314
match frequency (for prewarping) 6-19
matrices
    convolution 6-68
    correlation coefficient 6-70
    covariance 6-71
    discrete Fourier transform 6-96
    inverse discrete Fourier transform 6-96
maxflat 6-4
maxflat 6-4, 6-31, 6-135, **6-204**, 6-316
mean 6-70, 6-71
medfilt1 6-9, **6-206**
medfilt2 6-206
median 6-70, 6-71, 6-206
median filter 6-206
message signal 6-207
minimum phase filter 6-225
modulate 6-9, 6-94, **6-207**, 6-304
modulation 6-207
    methods 6-207

N

normalization
    correlation 6-306

O

oe 6-284
order estimation 6-245
    Butterworth 6-32
    Chebyshev type I 6-38
    Chebyshev type II 6-43
    elliptic 6-111
oscillator
    voltage controlled 6-303
overlap-add method, filter implementation 6-120
overlap-add method, FIR filter implementation 6-120

P

parametric modeling 6-180
Parks-McClellan method, FIR filter design 6-240
partial fractions form 6-253
periodic sinc function 6-97
phase
  computing with angle 6-12
  unwrapping 6-298
phase delay 6-157
phase demodulation 6-94
phase modulation 6-208
plot
  strip plot 6-285
  zero-pole 6-322
plot 6-286
pmem 6-6, 6-83, 6-203, **6-210**, 6-216, 6-222, 6-231
pmtm 6-6, 6-83, 6-99, 6-213, **6-214**, 6-222, 6-231
pmusic 6-7, 6-83, 6-213, 6-216, **6-217**, 6-231
poly 6-255
poly2rc 6-3, **6-223**, 6-237, 6-287
polyfit 6-95
polynomial
  stabilization 6-225
polynomial division 6-92
polynomial multiplication 6-65
polystab 6-9, **6-225**
polyval 6-150
power spectral density 6-228
prewarping 6-19
  defined 6-19
prony 6-8, 6-179, 6-182, 6-191, 6-203, 6-213, 6-222, **6-226**, 6-255, 6-284
Prony's method 6-226
prototype
  Bessel filter 6-14
  Butterworth filter 6-26
  Chebyshev type I filter 6-37
  Chebyshev type II filter 6-42
  elliptic filter 6-110
psd 6-7, 6-64, 6-83, 6-118, 6-213, 6-216, 6-222, **6-228**, 6-269, 6-295
pulse time demodulation 6-94
pulse time modulation 6-208
pulse train generator 6-233
pulse width demodulation 6-94
pulse width modulation 6-208
pulstran 6-2, 6-58, 6-60, 6-97, 6-98, 6-100, 6-101, 6-102, 6-103, 6-156, **6-233**, 6-239, 6-256, 6-259, 6-275, 6-287, 6-297

Q

quadrature amplitude demodulation 6-94
quadrature amplitude modulation 6-208
quantization noise 6-318

R

rc2poly 6-3, 6-224, **6-237**, 6-287
rceps 6-9, 6-36, 6-159, 6-163, 6-164, **6-238**
real cepstrum 6-238
rectangular window 6-25
rectpuls 6-2, 6-60, 6-97, 6-156, 6-236, **6-239**, 6-256, 6-259, 6-275, 6-297
reflection coefficients 6-223, 6-237
remez 6-5, 6-132, 6-135, 6-138, 6-141, 6-146, 6-147, **6-240**, 6-247, 6-316
  filter characteristics 6-242
  order estimation 6-245
Remez exchange algorithm 6-240
remezord 6-5, 6-188, 6-244, **6-245**
resample 6-9, 6-91, 6-174, 6-176, **6-249**, 6-302
resampling 6-249
residue 6-255

`residuez` 6-3, 6-65, 6-92, **6-253**
roots
 of Bessel filter 6-14
`roots` 6-225, 6-255

S

sampling rate
 changing by non-integer factor 6-249
 decreasing by integer factor 6-89
 increasing by integer factor 6-173
`sawtooth` 6-2, 6-60, 6-97, 6-156, 6-236, 6-239, **6-256**, 6-259, 6-275, 6-297
second-order sections form
 converting to state-space 6-260
 converting to transfer function 6-262
 converting to zero-pole-gain 6-264
signal
 analytic 6-162
 carrier 6-93
 message 6-207
 reconstruction
  minimum phase 6-238
Signal Browser 6-271
`sin` 6-60, 6-97, 6-156, 6-236, 6-239, 6-256, 6-259, 6-275, 6-297
`sinc` 6-2, 6-60, 6-97, 6-156, 6-236, 6-239, 6-256, **6-257**, 6-275, 6-297
 bandlimited interpolation example 6-258
sinc function 6-257
 and bandlimited interpolation 6-258
`sos2ss` 6-3, **6-260**, 6-263, 6-265, 6-278, 6-319
`sos2tf` 6-3, 6-261, **6-262**, 6-265, 6-278, 6-319
`sos2zp` 6-3, 6-261, 6-263, **6-264**, 6-278, 6-319
spa 6-295
`specgram` 6-9, 6-231, **6-266**
 example 6-303
spectrogram 6-266
 example 6-303
Spectrum Viewer 6-274
`spline` 6-91, 6-174, 6-252
`sptool` 6-10, **6-271**
`square` 6-2, 6-60, 6-97, 6-156, 6-236, 6-239, 6-256, 6-259, **6-275**, 6-297
`ss2sos` 6-3, 6-261, 6-263, 6-265, **6-276**, 6-319
`ss2tf` 6-255, **6-279**, 6-281, 6-289, 6-291, 6-320, 6-321
`ss2zp` 6-4, 6-279, **6-280**, 6-289, 6-291, 6-320, 6-321
stabilization, polynomial 6-225
startup transients
 reducing 6-127
state-space form
 converting to second-order section 6-276
 converting to zero-pole-gain 6-280
`std` 6-70, 6-71
Steiglitz-McBride iteration 6-282
`stem` 6-172, 6-286
`stmcb` 6-8, 6-191, 6-203, 6-227, **6-282**
strip plot 6-285
 defined 6-285
`strips` 6-2, **6-285**
swept-frequency cosine generator. *See* chirp

T

`tf2latc` 6-4, 6-189, 6-190, 6-224, 6-237, **6-287**
`tf2ss` 6-4, 6-255, 6-279, 6-281, **6-288**, 6-291, 6-320, 6-321
`tf2zp` 6-255, 6-279, 6-280, 6-289, **6-290**, 6-320, 6-321
`tfe` 6-7, 6-64, 6-83, 6-231, 6-269, **6-292**
toolbox
 controls 6-172
 image processing 6-88, 6-165, 6-206

symbolic math 6-14
system identification 6-203, 6-284, 6-295
transfer function
converting to state-space 6-288
estimate from input and output 6-292
transform
chirp *z*-transform 6-85
discrete cosine 6-87
discrete Fourier 6-115
Hilbert 6-162
inverse discrete cosine 6-165
inverse discrete Fourier 6-166
transformations
bilinear 6-19
frequency 6-192, 6-195, 6-197, 6-199
transposed direct form II 6-123
initial conditions 6-128
trend
removing 6-95
`triang` 6-7, 6-13, 6-24, 6-25, 6-47, 6-160, 6-161, 6-183, **6-296**
compared to `bartlett` 6-13
triangular window 6-296
`tripuls` 6-2, 6-60, 6-97, 6-156, 6-236, 6-239, 6-256, 6-259, 6-275, **6-297**
two-dimensional operations
autocorrelation 6-310
convolution 6-66
obtaining subsection 6-66
cross-correlation 6-310
discrete Fourier transform 6-119
filtering 6-126
inverse discrete Fourier transform 6-167

U

unit circle 6-225
`unwrap` 6-3, 6-36, 6-164, 6-238, **6-298**
`upfirdn` 6-9, 6-91, 6-174, 6-252, **6-299**

V

`vco` 6-9, 6-94, 6-209, **6-303**
vector
frequency 6-133, 6-136, 6-240, 6-314
magnitude 6-133, 6-136, 6-314
weighting 6-143, 6-241
voltage controlled oscillator 6-303

W

waveform
sawtooth 6-256
square 6-275
triangle 6-256
Welch's method
for cross spectral density estimation 6-83
for power spectral density estimation 6-64, 6-231
window
Bartlett 6-13
Blackman 6-24
Chebyshev 6-47
Hamming 6-160
Hanning 6-161
Kaiser 6-183
rectangular 6-25
triangular 6-296
window method, FIR filter design
bandpass configuration 6-130
bandstop configuration 6-130
highpass configuration 6-130
lowpass configuration 6-130

X

xcorr 6-7, 6-65, 6-67, 6-70, 6-71, **6-305**, 6-310, 6-313
xcorr2 6-7, 6-67, 6-309, **6-310**, 6-313
xcov 6-7, 6-70, 6-71, 6-309, **6-311**

Y

yulewalk 6-4, 6-135, 6-244, **6-314**
Yule-Walker filter design 6-314

Z

zero-order hold 6-249
See also averaging filter
zero-phase filtering 6-127
zero-pole analysis
zero-pole plots 6-322
zero-pole-gain form
converting to second-order section 6-317
converting to state-space 6-320
zp2sos 6-4, 6-261, 6-263, 6-265, 6-278, **6-317**
zp2ss 6-4, 6-255, 6-279, 6-281, 6-289, 6-291, **6-320**, 6-321
zp2tf 6-279, 6-289, 6-290, 6-291, 6-320, **6-321**
zplane 6-3, **6-322**
z-transform
chirp *z*-transform 6-85
discrete Fourier transform 6-115