

MATLAB TRAINING SESSION VII

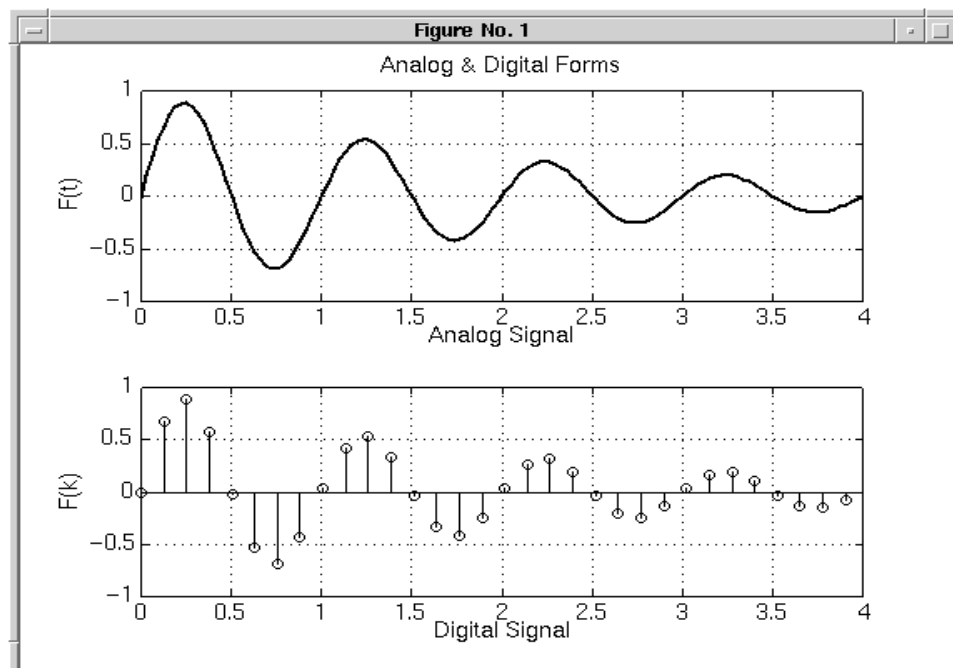
BASIC SIGNAL PROCESSING

Frequency Domain Analysis:

While we will discuss both analog and digital signal processing, the focus will be on digital signal processing (**DSP**). Recall that an analog signal is a continuous function, usually of time, that represents information. In order to process this information with the computer, an analog signal can be sampled every T seconds, thus generating a digital signal that is a sequence of values from the original analog signal. We represent a digital signal that has been sampled from a continuous signal $f(t)$ using the following notation: $f_k = f(kT)$. The digital signal is the sequence represented by $[f_k]$.

The time that we start collecting the samples is usually assumed to be zero, and thus the first sample is usually referred to as f_0 . Hence, if a signal is sampled at 100 Hz (100 times per second), the first three values of the digital signal correspond to the following analog signal values $f_0 = f(0T) = f(0.00)$; $f_1 = f(1T) = f(0.01)$; and $f_2 = f(2T) = f(0.02)$. To see the relationship between an analog signal and its corresponding digital signal. At the command prompt try:

```
rupp
>> t = linspace(0,4,128);
>> f = exp(-0.5*t).*sin(2*pi*t);
>> subplot(2,1,1), plot(t,f), title('Analog & Digital Forms')
>> xlabel('Analog Signal'), ylabel('F(t)')
>> subplot(2,1,2), stem(t(1:4:128),f(1:4:128))
>> xlabel('Digital Signal'), ylabel('F(k)')
>>
```



Often, we will associate two vectors with a signal. One vector will signify the subscript or the sequence number and the other will represent the signal value. This will be needed so that we can use the equations from signal processing without conflict with MATLAB's convention for numbering the elements of a vector.

Usually, a signal is analysed in two domains—the time domain and the frequency domain. The time domain signal is represented by the data values; the frequency domain signal is represented by complex number values that represent the sinusoidal frequency components that compose the signal. Some types of information are most evident from the time domain representation of the signal. For example, by looking at the time domain plot, we can usually determine if the signal is periodic, or if it is a random signal. From the time domain values, we can also easily compute additional values such as mean, variance, and power. Other types of information, such as the frequency content of a signal, are not usually evident from the time domain, and thus we may need to compute the frequency content of the signal in order to determine if it is band-limited or if it contains certain frequencies. The frequency content of a signal is also called the frequency spectrum of the signal.

Discrete Fourier Transform:

The discrete Fourier transform (**DFT**) algorithm is used to convert a digital signal in the time domain into a set of points in the frequency domain. The input to the DFT algorithm is a set of N values $[f_k]$; the algorithm then computes a set of N complex values $[F_k]$ that represents the frequency domain information. The DFT can be very computationally intensive, however if the number of points is a power of two ($N = 2^M$), then a special algorithm called the fast Fourier transform (**FFT**) can be used; the FFT greatly reduces the number of computations.

Since a digital signal is sampled every T seconds, there are $1/T$ samples per second; and thus the sampling rate is $1/T$ Hz. The selection of the sampling rate for generating a digital signal must be done carefully to avoid aliasing, a problem that is caused by sampling too slowly. To avoid aliasing, the signal must be sampled at a sampling rate that is greater than twice the frequency of the highest sinusoid content of the signal. The **Nyquist** frequency is equal to half the sampling rate, and represents the upper limit of the frequencies that should be contained in the digital signal.

Fast Fourier Transform:

The MATLAB function for computing the frequency content of a digital signal is the `fft` function. If a single input vector argument is used, the digital time domain signal, the output vector will be the same size and will contain complex values representing the frequency content of the signal. When the input vector size is a power of two, the `fft` algorithm is used, otherwise the DFT algorithm is used. When two input argument vectors are used, the first vector is the time signal and the second argument is an integer (L) representing the desired number of points for the output vector. When $L < N$, then the first L values are used and when $L > N$, then $L - N$ zeros are appended to the input vector.

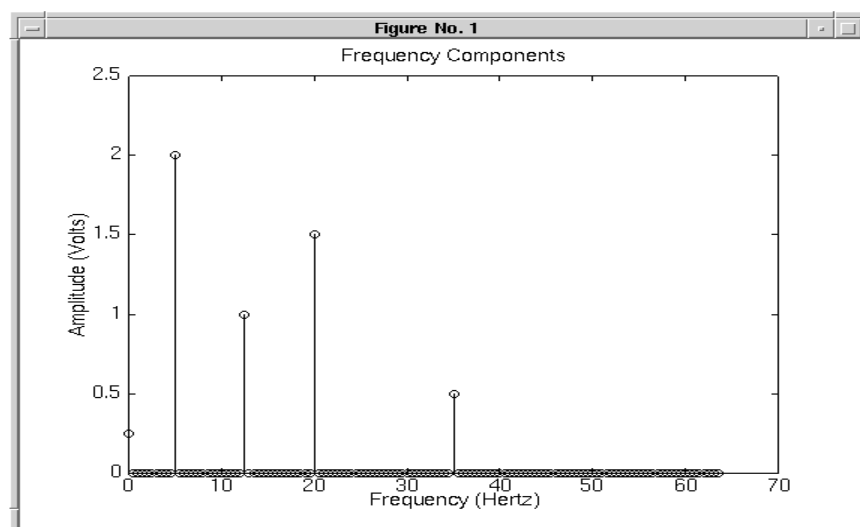
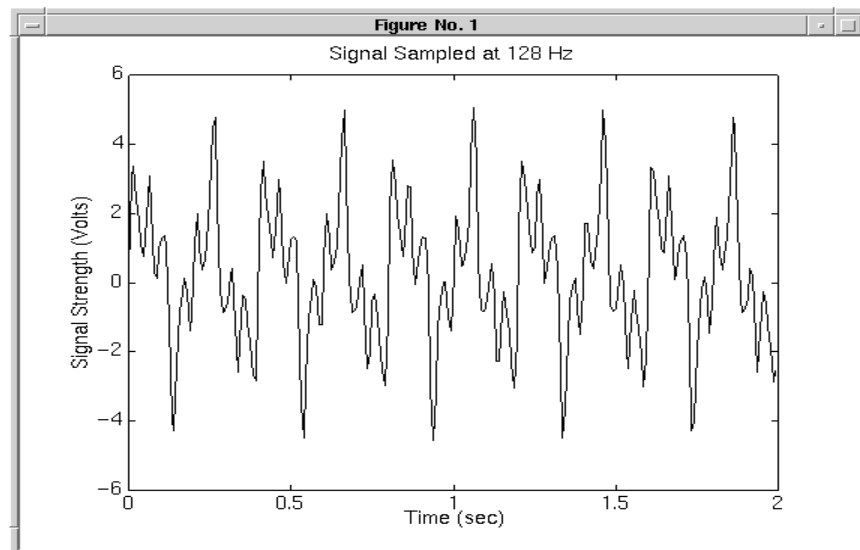
The frequency domain values computed by the `fft` function correspond to the frequencies separated by $1/(N*T)$ Hz. Thus, if we have 32 samples of a time signal that was sampled at 1000 Hz, then the frequency values computed by the `fft` correspond to $F_i = i*31.25$ Hz where $i = 0, 1, \dots, 31$. The Nyquist frequency is equal to $1/2T$ (500 Hz), and thus corresponds to F_{16} . Since

the discrete Fourier transform is a periodic function, the computed values above F_{16} are merely the complex conjugates of the values below, no new information is given so usually only the first half of the output of the `fft` is plotted.

```

>> N=256: % # of samples
>> T=1/128: % sampling time interval
>> k=0:N-1: time = k*T: % Setup discrete time
>> %Use the first 1/2 of the data and normalize
>> f = .25 + 2*sin(2*pi*5*k*T) + 1*sin(2*pi*12.5*k*T) + ...
      1.5*sin(2*pi*20*k*T) + .5*sin(2*pi*35*k*T):
>> plot(time,f), title('Signal Sampled at 128 Hz')
>> xlabel('Time (sec)'), ylabel('Signal Strength (Volts)')
>> % The fft routine
>> F=fft(f);
>> magF=abs([F(1)/N,F(2:N/2)/(N/2)]);
>> hertz=k(1:N/2)*(1/(N*T));
>> stem(hertz,magF), title('Frequency Components')
>> xlabel('Frequency (Hertz)'), ylabel('Amplitude (Volts)')
>>

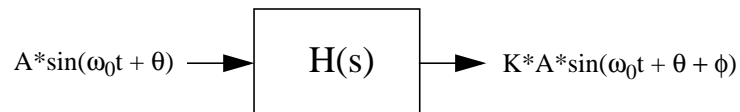
```



One more point is the concept of leakage. If there is a frequency in the time domain signal that falls in the interval between two of the discrete frequencies used in the `fft`, it will be detected as a component in both frequencies, but with reduced amplitude. In MATLAB an approximation of the time domain signal can be recovered using the `ifft` function. Try: `plot(time,ifft(F))`

Filter Analysis:

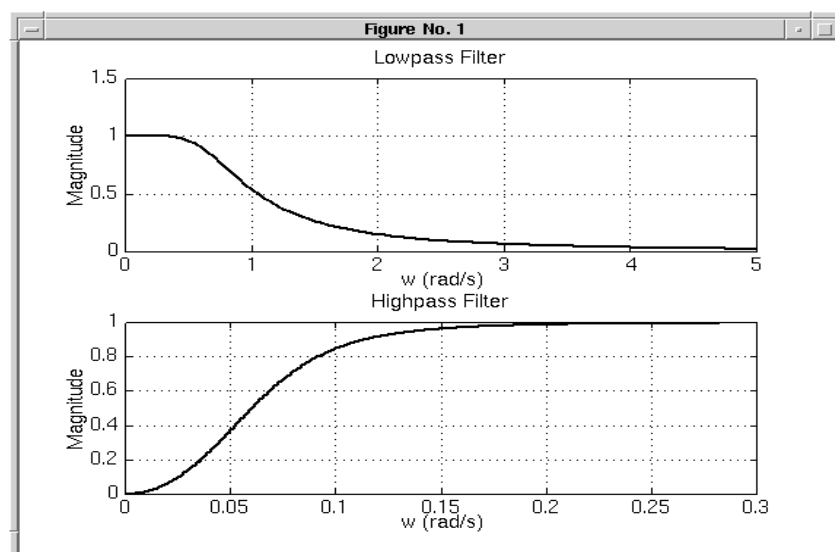
The transfer function of an analog system is represented by a complex function $H(s)$, and a transfer function of a digital system is represented by a complex function $H(z)$. These transfer functions describe the effect of the system on an input signal, termed filtering. For a given frequency ω_0 the filter has a magnitude K and a phase ϕ . Then, if the input to the filter contains a sinusoid with frequency ω_0 , the output of that component will be multiplied by K and its phase will be incremented by ϕ .



The transfer function of a filter can also be described in terms of the band of frequencies that it passes. For example, a *lowpass* filter will pass (unalter) frequencies below a cutoff frequency and remove (attenuate) the frequencies above. There are also *highpass*, *bandpass*, and *bandstop* (*notch*) filters. In order to determine the characteristics of a transfer function, we need to plot the corresponding magnitude and phase functions. When we have a transfer function representation of the filter (i.e. a numerator polynomial and a denominator polynomial) we can look at the magnitude characteristic with the MATLAB function `freqs` and `freqz`. Try:

```

>> w1 = 0:0.05:5; w2 = 0:0.001:0.3;
>> F1num = [0.5979]; F1den = [1 1.0275 0.5979];
>> Hs1 = freqs(F1num,F1den,w1);
>> F2num = [1 0 0]; F2den = [1 0.1117 0.0062];
>> Hs2 = freqs(F2num,F2den,w2);
>> subplot(2,1,1), plot(w1,abs(Hs1)), title('Lowpass Filter')
>> xlabel('w (rad/s)'), ylabel('Magnitude'), grid
>> subplot(2,1,2), plot(w2,abs(Hs2)), title('Highpass Filter')
>> xlabel('w (rad/s)'), ylabel('Magnitude'), grid
>>
  
```



When determining the characteristics of a digital filter using the `freqz` function, 3 input arguments are required. The third argument differs from `freqs` in the sense that instead of specifying the frequency range, you specify the number of normalized frequency values to use over the interval $[0, \pi]$. Since $H(z)$ is applied to input signals with a sampling time of T , the appropriate range of frequencies is from 0 to the Nyquist frequency, which is π/T rps or $1/(2T)$ Hz. When we assume that z is used as a function of normalized frequency, then $H(z)$ has a corresponding range of frequencies from 0 to π .

The phase of a filter can be plotted using the `angle` function or the `unwrap` function. Since the phase of a complex number is an angle in radians, the angle is only unique for a 2π interval. The output of the `angle` function maps the phase values to values between $-\pi$ and π , and thus can create discontinuities at points where the phase exceeds these bounds. The `unwrap` function removes the 2π discontinuities introduced by the `angle` function when used in the reference `unwrap(angle(H))`.

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_n z^{-n}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_n z^{-n}}$$

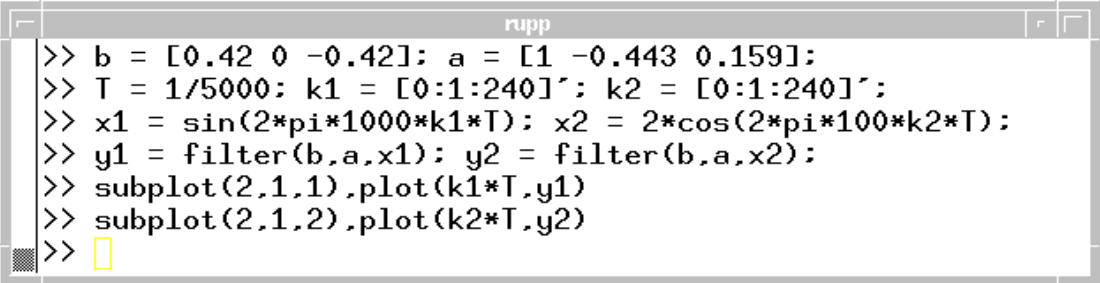
$$y_n = \sum_{k=-N1}^{N2} b_k x_{n-k} - \sum_{k=1}^{N3} a_k y_{n-k}$$

A digital filter can also be specified using a standard difference equation, **SDE**, which has the general form shown above. When $N_1 = 0$, the coefficients $[b_k]$ and $[a_k]$ are precisely the same coefficients in the transfer function $H(z)$. If the denominator transfer function is equal to 1, the filter is an finite impulse response filter (**FIR**); If the denominator transfer function is not equal to 1, the filter is an infinite impulse response filter (**IIR**). Both types of filters are commonly used in digital signal processing.

Digital Filter Implementation:

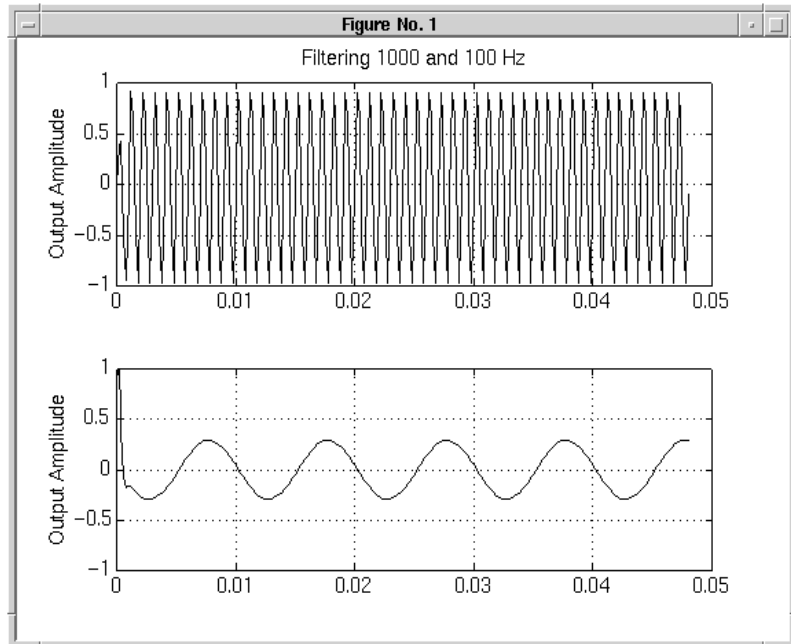
A digital filter can be defined in terms of either a transfer function $H(z)$ or a standard difference equation. The input to the filter is a digital signal, and the output of the filter is another digital signal. The difference equation shown above defines the steps involved in obtaining y at each discrete time n . The simplest way to apply a digital filter to an input signal in MATLAB is with the `filter` function. The `filter` function assumes that the filter is defined as the transfer function $H(z)$. The first two arguments to the filter function are the vector of coefficients for the numerator and denominator. The third argument is the vector representing the digital input x .

The following transfer function was designed to pass frequencies between 500 and 1500 Hz



```

>> b = [0.42 0 -0.42]; a = [1 -0.443 0.159];
>> T = 1/5000; k1 = [0:1:240]'; k2 = [0:1:240]';
>> x1 = sin(2*pi*1000*k1*T); x2 = 2*cos(2*pi*100*k2*T);
>> y1 = filter(b,a,x1); y2 = filter(b,a,x2);
>> subplot(2,1,1).plot(k1*T,y1)
>> subplot(2,1,2).plot(k2*T,y2)
>>
  
```



Digital Filter Design:

IIR Filter Design using Analog Prototypes:

MATLAB contains functions for designing four types of digital filters based on analog filter designs. Butterworth filters have maximally flat passbands and stopbands, Chebyshev Type I filters have ripple in the passband, Chebyshev Type II filters have ripple in the stopband, and elliptic filters have ripple in both bands. However, for a given filter order, elliptic filters have the sharpest transition of all these filters. The functions for designing digital lowpass IIR filters using analog prototypes have the following format:

<code>[B,A] = butter(N,Wn)</code>	N = filter order
<code>[B,A] = cheby1(N,Rp,Wn)</code>	R_p = pass ripple
<code>[B,A] = cheby2(N,Rs,Wn)</code>	R_s = stop ripple
<code>[B,A] = ellip(N,Rp,Rs,Wn)</code>	W_n = normalized cutoff frequency

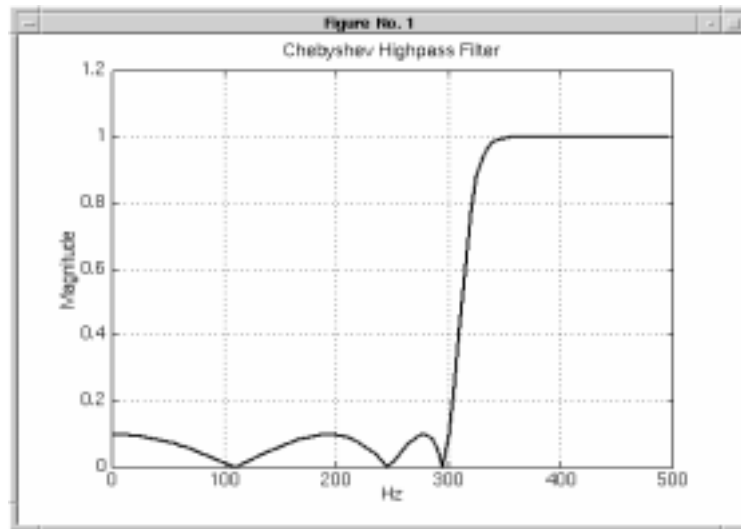
To design a bandpass filter, W_n should be a vector containing W_{p1} and W_{p2} . To design a highpass filter, an additional parameter 'high' should be added as the last function parameter. To design a bandstop filter, the additional parameter 'stop' should be added as the last function parameter.

```

>> [B,A] = cheby2(6,20,0.6,'high')
B =
    0.1587   -0.1665    0.3482   -0.3256    0.3482   -0.1665    0.1587
A =
    1.0000    0.7265    1.2535    0.2506    0.3309   -0.0367    0.0282

>> [H,wT] = freqz(B,A,100);
>> T = 0.001; hertz = wT/(2*pi*T);
>> plot(hertz,abs(H)),title('Chebyshev Highpass Filter')
>> xlabel('Hz'), ylabel('Magnitude'), grid

```



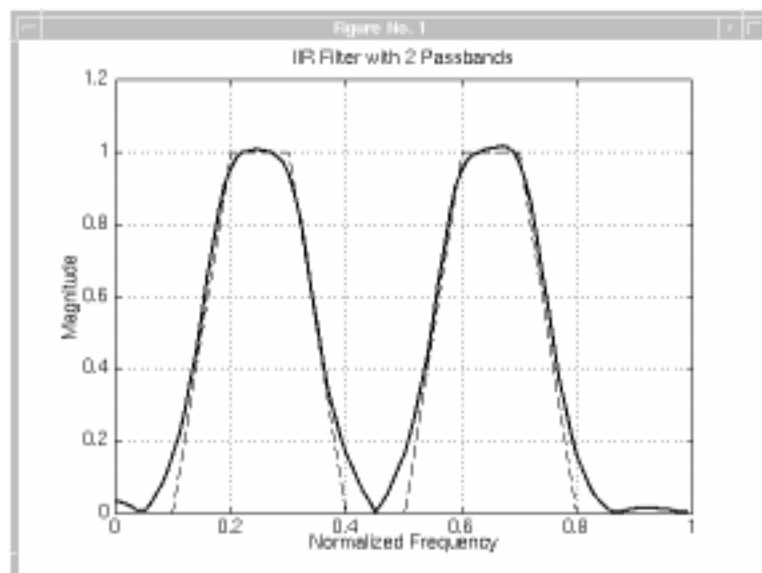
Direct IIR Filter Design:

MATLAB has a function for performing Yule-Walker filter designs. The command is:

```
[B,A] = yulewalk(n,f,m)
```

The output vectors B and A contain the coefficients of the nth order IIR filter. The vectors f and m specify the frequency-magnitude characteristics of the filter over the normalized frequency range 0 to 1, which represents the Nyquist frequency range. The frequencies in f must begin with 0 and end with 1, and be increasing. The magnitudes of m must correspond to the frequencies in f, and represent the desired filter magnitude for each f.

```
rupp
>> m = [0 0 1 1 0 0 1 1 0 0];
>> f = [0 .1 .2 .3 .4 .5 .6 .7 .8 1];
>> [B,A] = yulewalk(12,f,m);
>> [H,wT] = freqz(B,A,100);
>> plot(f,m,'--',wT/pi,abs(H)). title('IIR Filter with 2 Passbands')
>> xlabel('Normalized Frequency'), ylabel('Magnitude'), grid
```



Direct FIR Filter Design:

FIR filters are designed in MATLAB using the Parks-McClellan filter design algorithm that uses the Remez exchange algorithm. Recall that FIR filters require only a B vector because the denominator polynomial of $H(z)$ is equal to 1. The command is:

$$B = \text{remez}(n, f, m)$$

The arguments play the same role and have the same constraints as with the `yulewalk` function given above. An additional rule is that the number of points in `f` and `m` must be an even number. To obtain desirable characteristics from the FIR filter, it is not unusual for the filter order to be large.

```

>> m = [0 0 1 1 0 0 1 1 0 0];
>> f = [0 .1 .2 .3 .4 .5 .6 .7 .8 1];
>> [B,A] = remez(50,f,m);
>> [H,wT] = freqz(B,[1],100);
>> plot(f,m,'--',wT/pi,abs(H)), title('FIR Filter with 2 Passbands')
>> xlabel('Normalized Frequency'), ylabel('Magnitude'), grid
>>

```

