Matlab Signal Processing Examples

This document provides some example code which implements some common signal processing tasks, such as synthesising signals, filtering signals and designing systems.

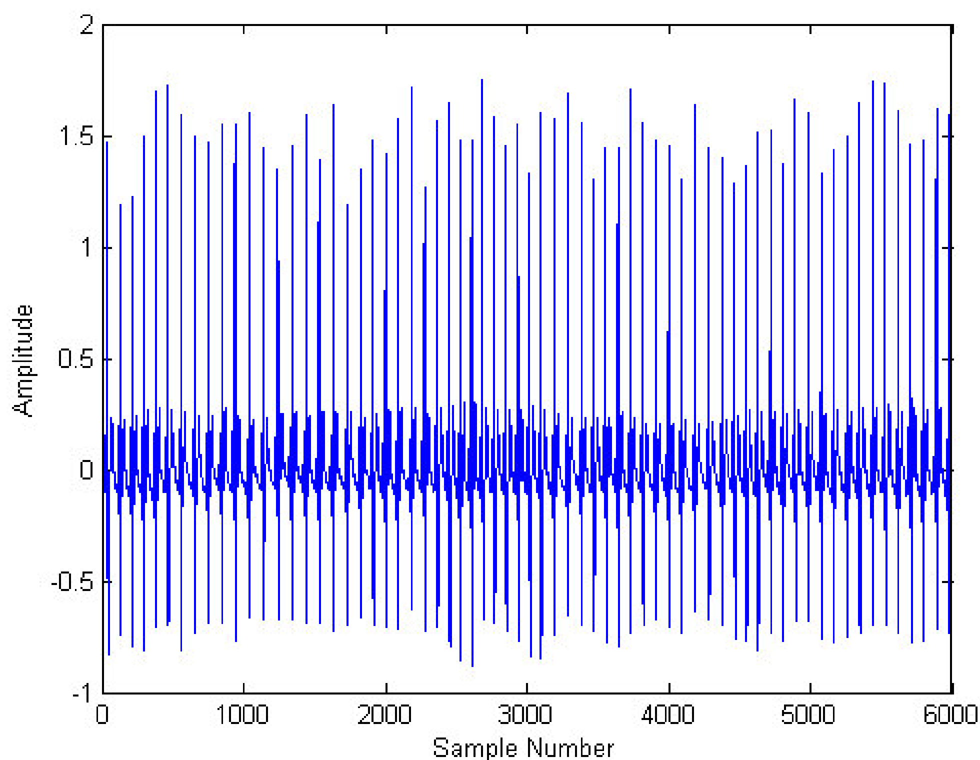
Contents

- Reading data from files
- Writing data to files
- The Colon (:) Operator - a really important feature in Matlab
- Creating/Synthesing Signals
- Analysing Frequency Content of a Signal
- Filtering Signals / Determining the Output of a System
- Determining a systems frequency response
- Designing Filters

Reading data from files

Signal processing involves analysing, manipulating and synthesising signals. The starting point for doing any of these tasks is often to read in a previously recorded signal of interest. The most general function to read in a signal is the load function; while functions like wavread and imread, which read in audio an images, respectively are also used frequently.

```
%load in a previously downloaded ecg signal (from physionet.org)
ecg_sig = load('ecg.txt'); % http://eleceng.dit.ie/dorran/matlab/ecg.txt
% It is known that this signal was sampled at 100Hz from physionet website
plot(ecg_sig);
ylabel('Amplitude')
xlabel('Sample Number')
```



The load command can also be used to read in previously saved matlab workspaces

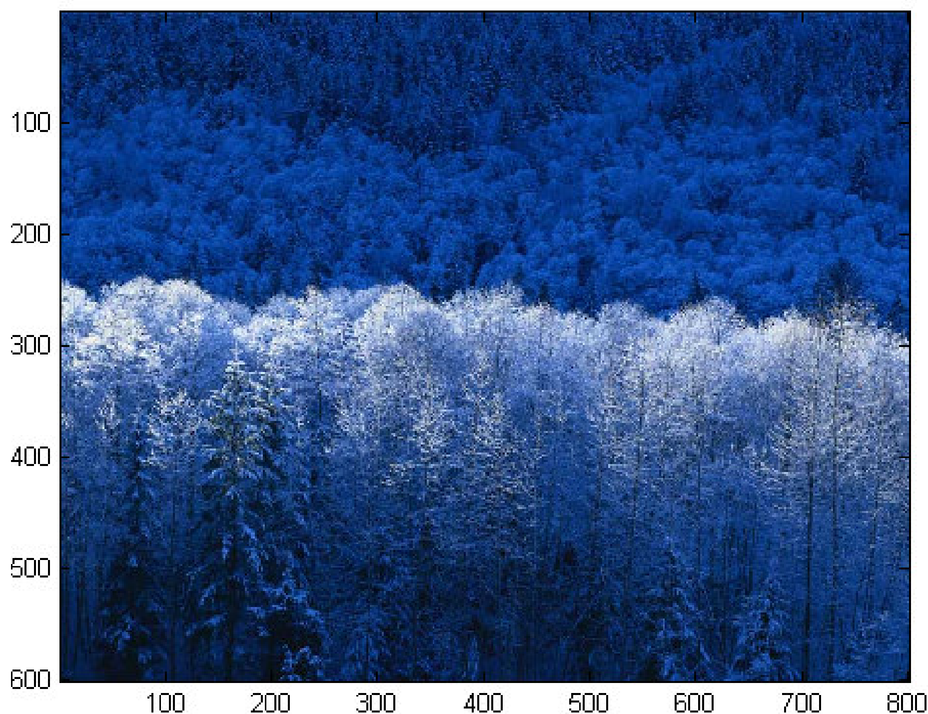
```
len_ecg = length(ecg_sig);
```

```
save('ecg_data.mat', 'ecg_sig', 'len_ecg');
clear all; %clear all variables from the workspace.

whos %show that there are no variables


load('ecg_data.mat') % load the variables back in
whos % show that the variables have been reloaded
```

```
  Name            Size            Bytes  Class     Attributes

  ecg_sig      6000x1             48000  double
  len_ecg         1x1                 8  double
```

```
[sound_sig fs] = wavread('drums.wav'); %http://eleceng.dit.ie/dorran/matlab/drums.wav
sound('drums.wav', fs);

image_signal = imread('Winter.jpg');
image(image_signal)
```

```
Warning: No sound hardware detected.
```



Writing data to files

This section focuses on writing plain text to files - this is a common task for recording the results of analysis/testing, for example. Please note that there are a number of specialised functions to create image files and audio files i.e. imwrite and wavwrite - you can find out more by looking up the help on these files.

Before writing a file you need to "open it". When you open a file for the purpose of writing to it you need to specify if you intend to overwrite or append to the file. The fopen function returns a file ID which can

then used to actual write data to the file using the fprintf function. Finally the file should be closed.

```
fid = fopen('test.txt', 'w' ); % open for overwriting
a = 5;
fprintf(fid, 'The value of the variable a is %d', a);
fprintf(fid, '\n\tBye Bye!!'); % put a newline (\n) and tab (\t) before Bye Bye
fclose(fid);
```

A demo of appending information to a file

```
file_id = fopen('test.txt', 'a' ); % open for appending
b = 22.233;
fprintf(file_id, 'The value of the variable a is %2.8f', b);
fprintf(file_id, '\n\n variable a is still a value of %2.2f \n\n Bye again!', a); % put a new
fclose(file_id);
```

A demo of writing a vector to a file

```
fid = fopen('vector_data.txt', 'w' ); % open for overwriting
vec = rand( 1, 1000);
fprintf(fid, '%2.4f\n', vec);
fclose(fid);
```

The Colon (:) Operator - a really important feature in Matlab

The colon operator is used both to create a sequence of numbers (signal generation) and to select a range of values from a discrete signal (segment selection). It is very important to become comfortable with both uses of the colon (:) operator.

This section is broken into two parts; part 1 deals with basic signal generation and part 2 with segment selection. Signal generation/synthesis is also dealt with more completly in a separate section.

Part 1 - Signal Generation The basic format for using the colon operator is the following: `starting_value : step_size : last_value_limit` however `starting_value : last_value_limit` is also used. Here are four examples of signal generation using the colon operator. A description of each example is contained in the following paragraph.

- Ex. 1

```
x = 0 : 7


x =

     0     1     2     3     4     5     6     7
```

- Ex. 2

```
x = 0 : 1 : 7


x =

     0     1     2     3     4     5     6     7
```

- Ex. 3

```
x = 1 : 0.25 : 2
```

```
x =

    1.0000    1.2500    1.5000    1.7500    2.0000
```

- Ex. 4

```
x = 100 : -0.33 : 99


x =

  100.0000   99.6700   99.3400   99.0100
```

As stated above the basic format for using the colon operator is the following:

```
starting_value : step_size : last_value_limit
```

This format is used for Ex. 2, 3 and 4, however if the `step_size` is ommitted (as is the case in Ex. 1) then the step_size defaults to 1. It should be appreciated that Ex. 2 produces the same values as Ex.1 since the `step_size` is explicitly set to 1.

Part 2 - Segment Selection

The colon operator is used frequently to select a range of values from a discrete signal. Here are a few examples.

Ex. 1

```
x = [12 34 4 5 6 78 8 9 2];
seg = x(3:5) % select from sample 3 to sample 5


seg =

     4     5     6
```

Ex. 2

```
x = [12 34 4 5 6 78 8 9 2];
seg = x(3:end) % select from sample 3 to the last sample


seg =

     4     5     6    78     8     9     2
```

Ex. 3 - Combination of signal generation and segement selection.

```
T = 0.01; %samling period
n = 0 : 300; % sample number
frequency = 2; % frequency of a sinusoid in hertz
x = cos(2*pi*frequency*n*T); % create three seconds of a cosine sinusoid
Xmags = abs(fft(x)); % get the magnitudes of the Discrete Fourier Transform
first_half_Xmags = Xmags(1 : round(length(Xmags)/2));
```

Creating/Synthesing Signals

A separate document deals with creating sequences of ones, zeros and random numbers (see

document on common signal processing functions). The previous section (the colon operator) also showed how to create a ramp signal and a sinuoid. This section is divided into four parts: Part 1 shows how to create a signal given any arbitrary mathemtical expression for a signal; Part 2 shows how to create combine signals through concatenation and summing; Part 3 shows how to create a periodic signal given any arbitrary period of a signal

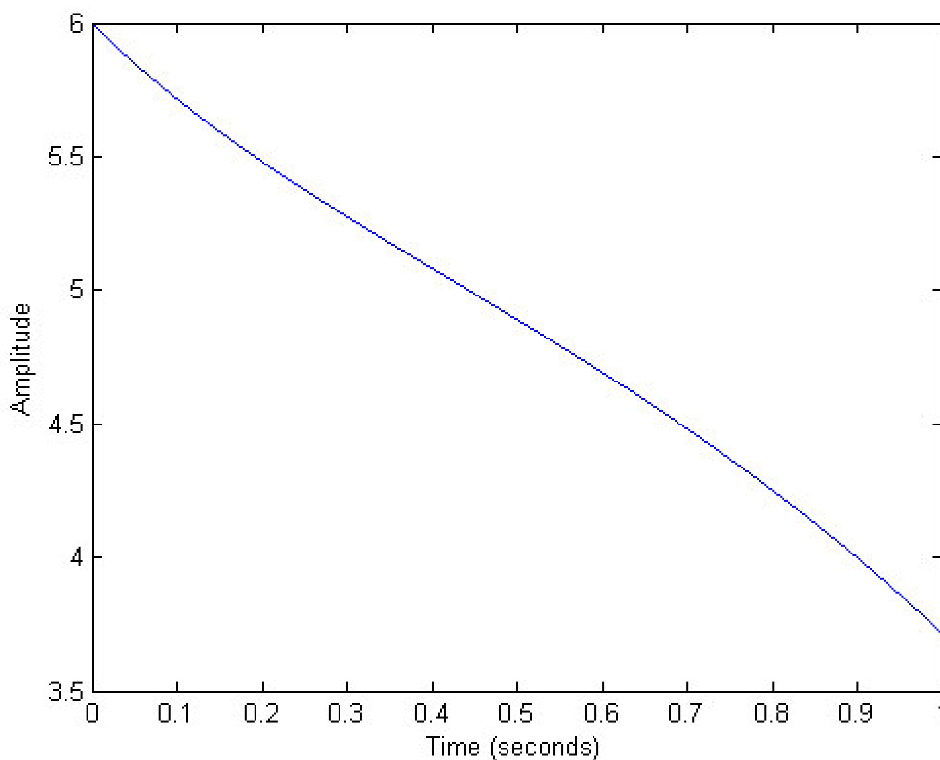
Part 1 - Synthesis Using a Mathematical Expression

Given a mathematical expression for a signal x(t) then the discrete version of this signal is given by x[n] = x(nT) where n is the sample number and T is sampling period.

Here are two examples of this formula being applied:

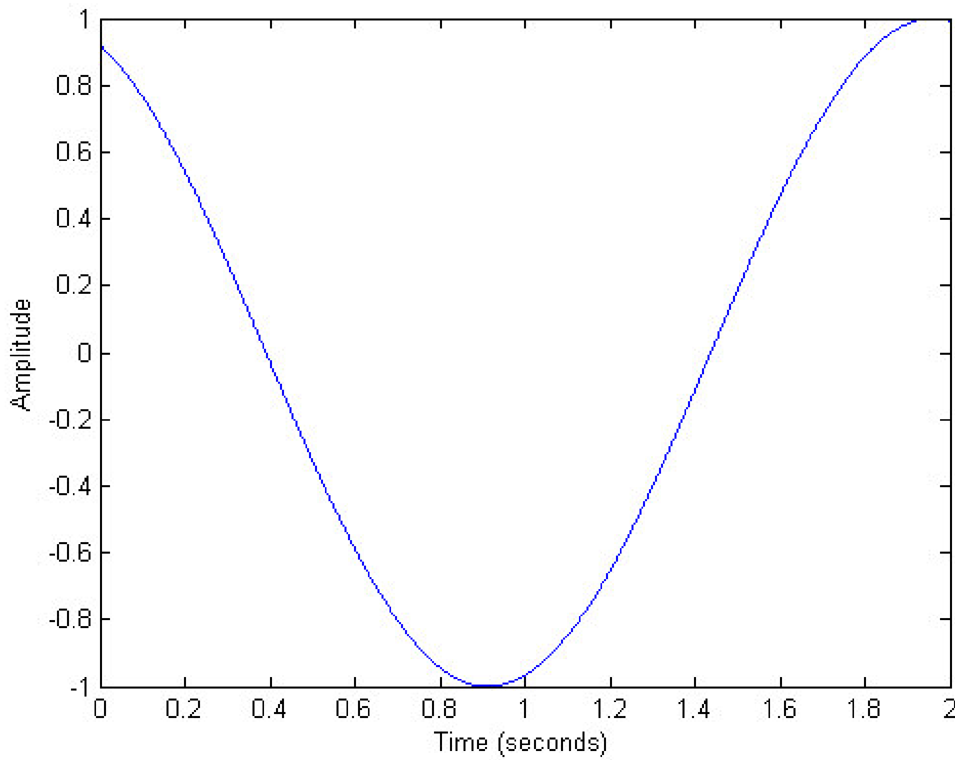
Ex. 1 - $x(t) = e^{-\pi t} + 5\sqrt[2]{cos(t)}$

```
T = 0.0001; % sampling period in seconds
n = 0 : 10000; %n is the sample number
x = exp(-pi*n*T) + 5*sqrt(cos(n*T));
t = n*T;
plot(t,x)
xlabel('Time (seconds)');
ylabel('Amplitude');
```



Ex. 2 - $x(t) = 4cos(\omega t + \phi)$ where $\phi = 0.4$ and $\omega = 3$

```
T = 0.001; % sampling period in seconds
n = 0 : 2000; %n is the sample number
omega = 3;
x = cos(omega*n*T + 0.4);
t = n*T;
plot(t,x)
xlabel('Time (seconds)');
ylabel('Amplitude');
```



Part 2 - Summing and Concatenating Signals

If you have two discrete signals (variables/sequences of numbers) you can add them together to produce a new signal. You can also append (concatenate) one signal on to the end of another one.

It should be noted that when you sum two signals they must have the same number of rows and columns, since each sample is added to the corresponding sample of the other signal. **Ex. 1** - adding two signals

```
x = [3 4 5 3 5];
y = [5 6 8 3 5 ];
w = x + y
```

```
w =

     8    10    13     6    10
```

Ex. 2 - concatenating two signals (rows)

```
x = [1 2 3 3];
y = [3 6 7 ];
w = [x y]
```

```
w =

     1     2     3     3     3     6     7
```

Ex. 3 - concatenating two signals (columns)

```
x = [3 ; 2 ; 6];
y = [1 ; 9];
w = [x ; y]
```

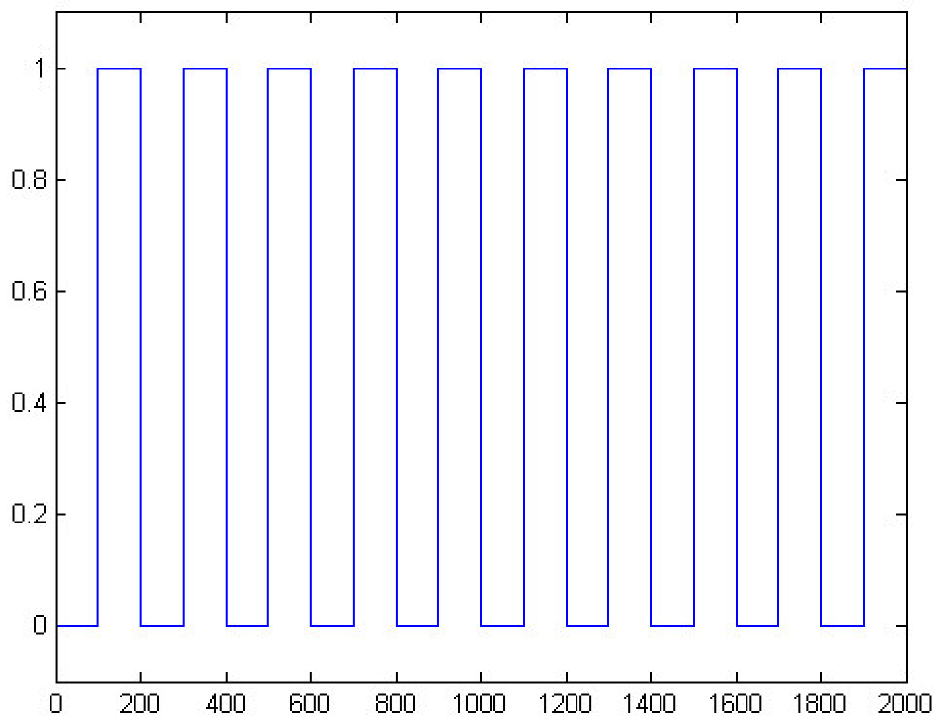
```
w =

     3
     2
     6
     1
     9
```

Part 3 - Creating an arbitrary periodic signal

If you have synthesised a signal using any of the methods described above you can use that signal as the repeating segment in a periodic signal. This particulary useful for signals that cannot be described mathematically without the use of piecewise functions

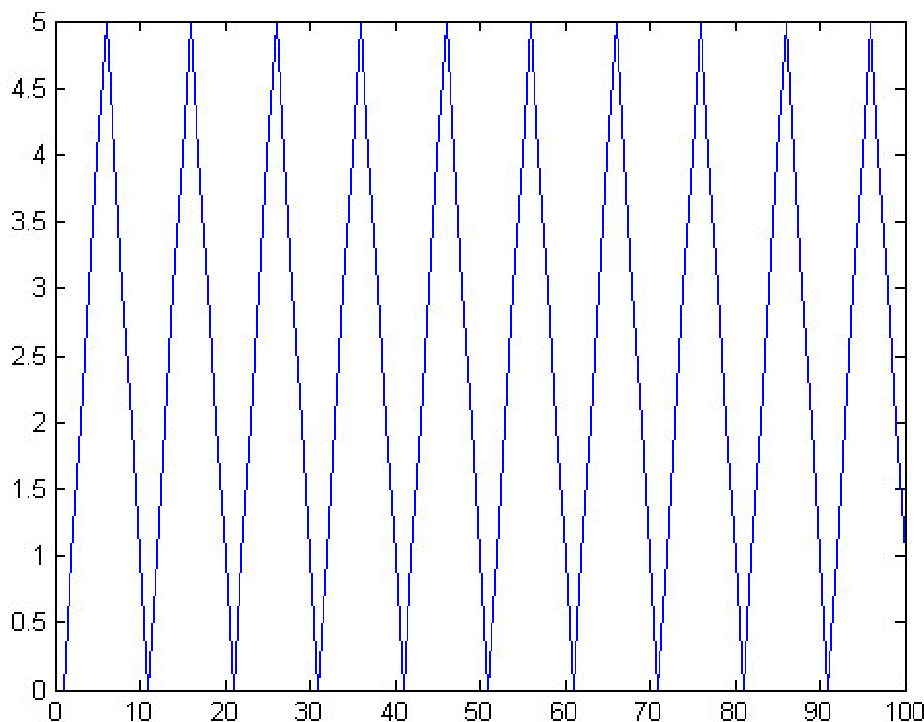
Ex. 1 - square waveform

```
repeating_segment = [ zeros(1, 100) ones(1, 100)];
num_periods = 10;
square_waveform = []; % empty variable
for k = 1 : num_periods
  square_waveform = [square_waveform repeating_segment];
end
plot(square_waveform);
ylim([-0.1 1.1])
```



Ex. 2 - triangular waveform

```
repeating_segment = [ 0:5 4:-1:1]; % same as [ 0 1 2 3 4 5 4 3 2 1 ]
num_periods = 10;
trangular_waveform = []; % empty variable
for k = 1 : num_periods
  trangular_waveform = [trangular_waveform repeating_segment];
end
plot(trangular_waveform);
```



Analysing Frequency Content of a Signal

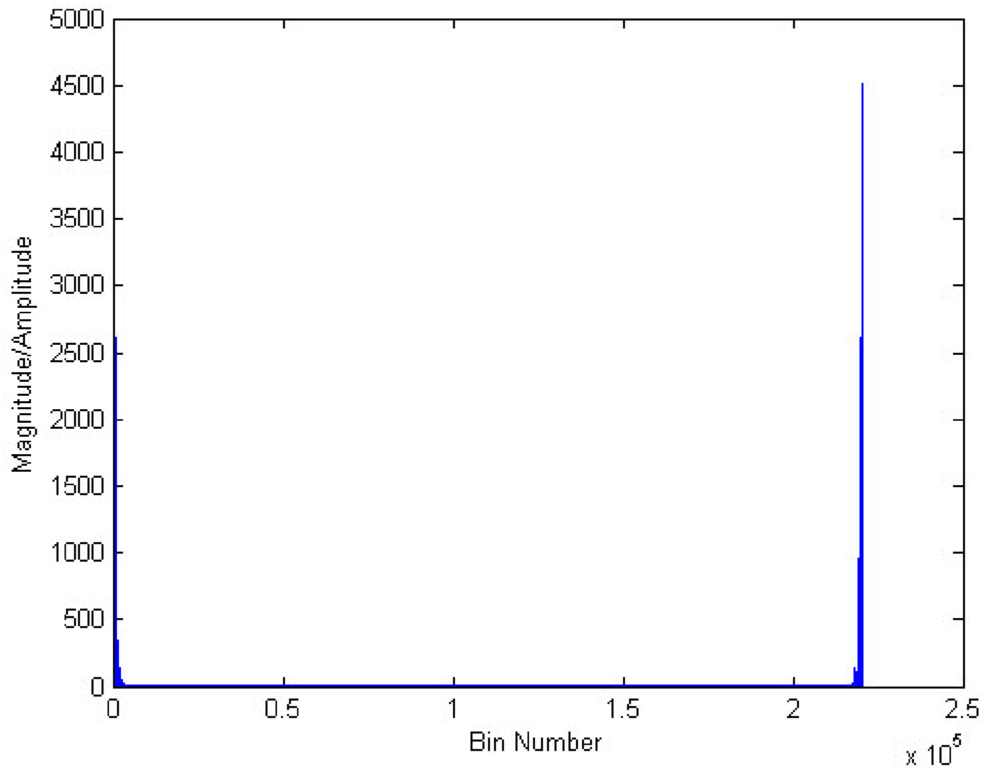
The `fft` is a function which calculates the Discrete Fourier Transform (DFT) of a signal. Interpreting what the `fft` function returns takes some practice, but it is one of the most commonly used functions in the DSP and signals & systems modules and any student of the topic needs to get comfortable with it, very quickly!

In this example the frequency content of a bass guitar signal will be analysed. Available to download at http://eleceng.dit.ie/dorran/matlab/bass.wav

```
fs = 44100; % the signal is known to have been recorded at this rate
bass_guitar = wavread('bass.wav', 5*fs); %read in first 5 seconds; fs = 44100Hz
ft = fft(bass_guitar);
mag_ft = abs(ft);
```

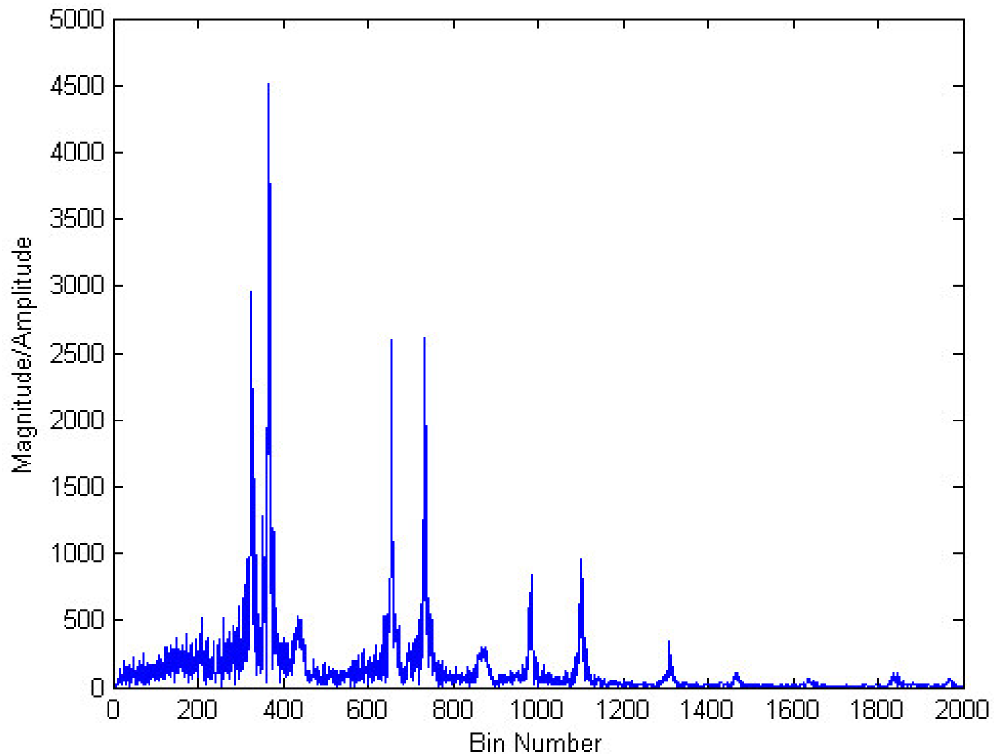
The following plot will be difficult to interpret as most of the frequency content is in the low frequency region, as expected for a bass guitar. The plot after this one will just show the low frequency content.

```
plot(mag_ft) % plot magnitudes versus frequency bins
ylabel('Magnitude/Amplitude')
xlabel('Bin Number')
```



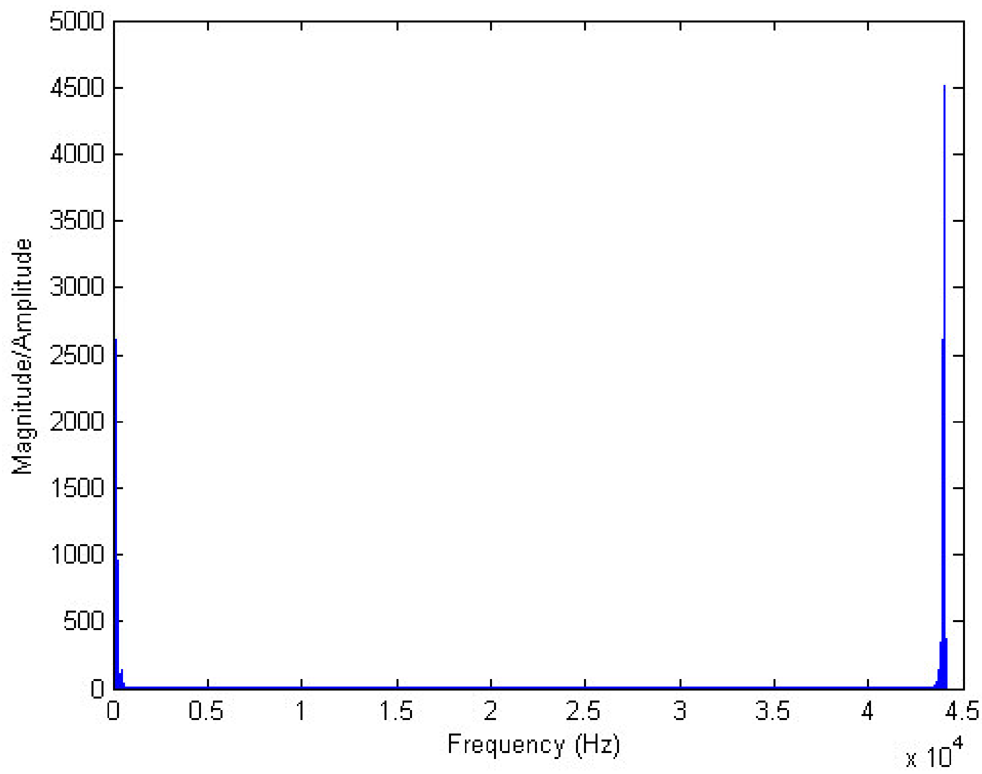
The following plot shows that there regions of strong spectral energy. These regions correspond to the frequencies of the fundamentals and harmonics of the notes being played on the guitar.

```
low_freq_mags = mag_ft(1:2000); %the magnitudes of the low freq content are stored in a new v
%Note: 2000 in the above command corresponds to 2000/length(mag_ft)*fs = 400Hz
plot(low_freq_mags);
ylabel('Magnitude/Amplitude')
xlabel('Bin Number')
```



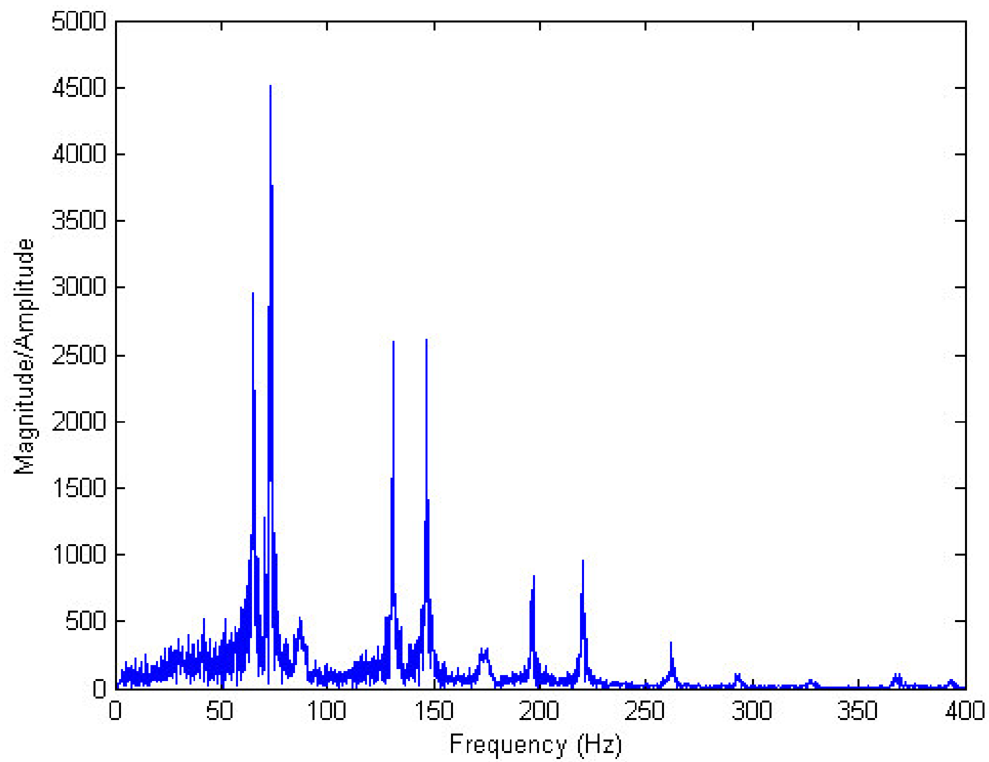
Oftentimes its useful to show the actual frequencies rather than bin numbers on the x axis. After taking the fft of a signal each frequency bin is separated by fs/(N-1) Hz where fs is the sampling frequency (Hz) and N is the number of frequency bins i.e. length(mag_ft) in the example above.

```
N = length(mag_ft);
freq_scale = 0: fs/(N-1) : fs;
plot(freq_scale, mag_ft);
ylabel('Magnitude/Amplitude')
xlabel('Frequency (Hz)')
```



Since low_freq_mags was extracted from mag_ft the same frequency spacing exists between the bins of low_freq_mags. Note that the length of the frequency scale must be the same as the length of the magnitude vector being plotted.

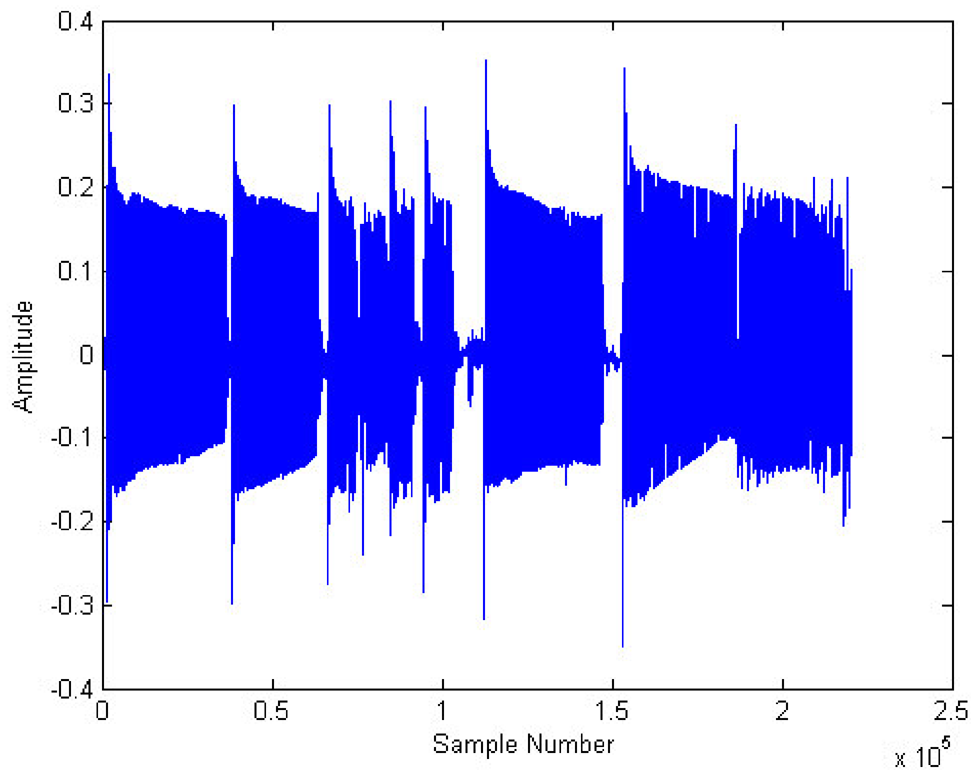
```
low_mag_freq_scale = freq_scale(1:length(low_freq_mags));
plot(low_mag_freq_scale, low_freq_mags);
ylabel('Magnitude/Amplitude')
xlabel('Frequency (Hz)')
```



Here's an example of analysing the frequency content of the first note in the bass guitar recording.

First analyse the time-domain signal to determine where the first note ends

```
plot(bass_guitar);

ylabel('Amplitude')
xlabel('Sample Number')
```



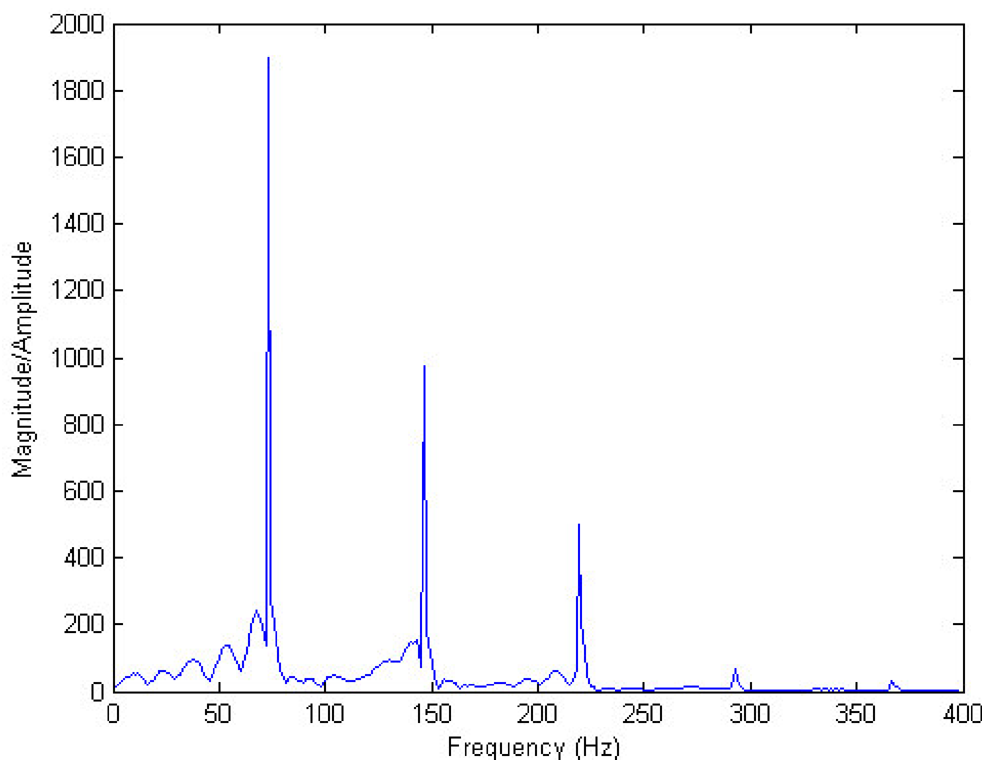
It can be seen that the first note ends at around sample 38000

```
first_note = bass_guitar(1:38000); %exrtact the first note from the signal
ft_first_note = fft(first_note);
mags = abs(ft_first_note);
freq_scale = 0: fs/(length(mags) -1) : fs;
```

determine the bin corresponding to 400Hz. Just display those frequency bins. The plot shows the fundamental at 73Hz with two strong harmonics at 146Hz and 219Hz

```
low_freq_bin_range = length(mags)*(400/44100);
low_freq_mags = mags(1:low_freq_bin_range);
low_freq_scale = freq_scale(1: low_freq_bin_range);
plot(low_freq_scale, low_freq_mags)
ylabel('Magnitude/Amplitude')
xlabel('Frequency (Hz)')
```

```
Warning: Integer operands are required for colon operator when used as index
Warning: Integer operands are required for colon operator when used as index
```



Filtering Signals / Determining the Output of a System

Before begining this section the reader must appreciate that a discrete system can be described by a set of `b` and `a` coefficients i.e. the coefficients associated with the systems transfer function.

There are two very straightforward ways of determining the output of a system when a given input is applied: 1. using the `filter` function 2. using convolution i.e. the `conv` function

A third way is to use frequency-domain multiplication - this way is less commonly used but is useful to be aware of.

Using the `filter` function

Once the `b` and `a` coefficients of a system are availbale it is a trivial excercise to determine the output of a system to any input. Here is an example for a system described by the following difference equation:

y[n] = 0.3x[n] + 1.5x[n-1] + 0.6y[n-1]

the `b` coefficients of this system are 0.3 and 1.5; the `a` coefficients are 1 and -0.6.

```
x = [ 2 4 5 23 1 34 5] ; % an example signal
b = [0.3 1.5];
a = [1 -0.6];

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

Using the `conv` function

Convolution is a mathematical process in which the output of a system can be determined if you have the systems impulse response `y = conv(x,h);`

This technique is particularly useful for the situation where it is a relatively easy to determine a systems impulse response, but rather more difficult to try an model the system in question i.e. determine a set of `b` and `a` coeffients that model the behaviour of the system. Perhaps one good example of this for audio processing, whereby a recording is changed by adding 'reverberation' effect which makes the recording sound as if it was recorded in a different environment. For example, when creating a music recording it

is often the case that each instrument is recorded in isolation in a sound proof anechoic (echo free) environment. Then the individual recordings are added together (mixed) before finally adding an effect which makes the music sound as if it were recorded in a large venue which has musically pleasing reverberation. This is done by convolving the music with the impulse response of appropriate venues; for example Carnegie Hall or the Sydney Opera House. This makes the music sound as if it was actually recorded in these venues. The impulse response of these venues is effectively just a recording of the venue after a pseudo audio impulse is played i.e. a loud short bang and they are easily obtained. Here's some example code to try out

```
[church_impulse fs] = wavread('church.wav'); % read in a previously recorded

impulse response of a church available from http://eleceng.dit.ie/dorran/matlab/church.wav

my_speech = wavrecord(2*fs, fs, 1); % record 2 seconds of speech

my_speech_in_church = conv(my_speech, church_impulse);

sound(my_speech_in_church, fs); %play back the recording. It should sound like

%you are speaking in a church
```

In this second example it'll be shown that convolution produces the same output as the `filter` function above. To do this the impulse response of the above system will first be determined using the `impz` fucntion.

```
x = [ 2 4 5 23 1 34 5] ; % an example signal
b = [0.3 1.5];
a = [1 -0.6];

h = impz(b,a);
y2 = conv(x,h);
% compare y2 with y above. They should be the same.
```

Using frequency-domain multiplication

The frequency content of the output of a system is equal to the fequency content of the input multiplied by the frequency response of the system i.e. Y(k) = X(k).H(k), where k represents the bin number.

Once Y(k) is determined the time-domain output can be obtained by taking the inverse DFT using the `ifft` function. It is important to note that in order for this technique to work the number of frequency bins used in determining Y(k) must be at least as long as the number of non-zero values in y[n] i.e. the length of the input plus the length of the impulse response. If this rule is not observed unusual (but explainable) results will occur.

Note that the frequency response of a system might be obtained in any number of ways

```
%(see section on determing a systems frequency response). In this example the frequency
%response of the example system above will be determined by taking the DFT of the systems
%impulse response, simply to show that the technique of frequency domain multiplication works
x = [ 2 4 5 23 1 34 5] ; % an example signal
b = [0.3 1.5];
a = [1 -0.6];
h = filter(b,a, [ 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]); %determine the impulse response
% alternatively use |impz| to calculate h

num_bins = length(x) + length(h); % this is minimum number of bins that should be uesed to de
H = fft(h, num_bins);
X = fft(x, num_bins);
Y = X.*H;
y3 = ifft(Y);
y3 = real(y3); % usually there will be some very small imaginary terms in the y value - these
% introduced due to rounding errors and should be ignored.
%
% compare y3 with y and y2 above. They should be the same.
```

Determining a systems frequency response

Three ways will be shown to determine a systems frequency response. The most straighforward way is using the `freqz` function. The two other ways shown here are based on signal processing theory; they are the impulse response method and the method of evaluating the systems transfer function 'around the unit circle' (rarely used, but important theory for designing filters)

Using `freqz` to determine a systems frequency response

If you have the `b` and `a` coefficients of a system then obtaining the frequency response is trivial using the `freqz` function.

Here is an example for a system described by the following difference equation:

y[n] = x[n] + 2x[n-1] + y[n-1] - 0.3y[n-2]

the `b` coefficients of this system are 1 and 2; the `a` coefficients are 1, -1 and 0.3.

```
b = [1 2];
a = [1 -1 0.3];

H = freqz(b,a);
plot(abs(H)); % show magnitude response against bin number
plot(angle(H));% show phase response against bin number
close all
```

The frequency scale can be updated to show normalised frequency using the following:

```
[H, w] = freqz(b,a);
plot(w, abs(H)); % show magnitude response against normalised frequency
plot(w, angle(H));% show phase response against normalised frequency
close all
```

Using the systems impulse response to determine a systems frequency response

An impulse contains all frequency components of the sample amplitude and 0 phase shift. This makes it an ideal signal to use to determine a systems frequency response, since if you pass an impulse through a system the output shows how all sinusoids are modified by the system.

Perhaps a good example of where this thoery can be used in practice is in when sound engineers need to analyse a music venue's frequency response in order to ensure certain frequencies are not over amplified by the venue, in order to ensure that the music sounds as the artist intends. Once analysed the frequency response of the venue can be altered by, for example, placing large curtains along a wall.

As an example to determine the frequency response of a church, the impulse response can be obtained. The frequency response is simply obtained by taking the fft of the churches impulse response.

```
[church_impulse fs] = wavread('church.wav'); % read in a previously recorded
% impulse response of a church available from http://eleceng.dit.ie/dorran/wav/church.wav
H = fft(church_impulse);
plot(abs(H)); % show magnitude response against bin number
plot(angle(H));% show phase response against bin number
close all
```

The frequency scale can be updated to show frequencies following:

```
[church_impulse fs] = wavread('church.wav'); % church impulse response
H = fft(church_impulse);
N = length(church_impulse);
freq_scale = 0 : fs/(N-1) : fs;
plot(freq_scale, abs(H)); % show magnitude response against normalised frequency
plot(freq_scale, angle(H));% show phase response against normalised frequency
close all
```

Evaluate H(z) at 'unit circle' to determine a systems frequency response

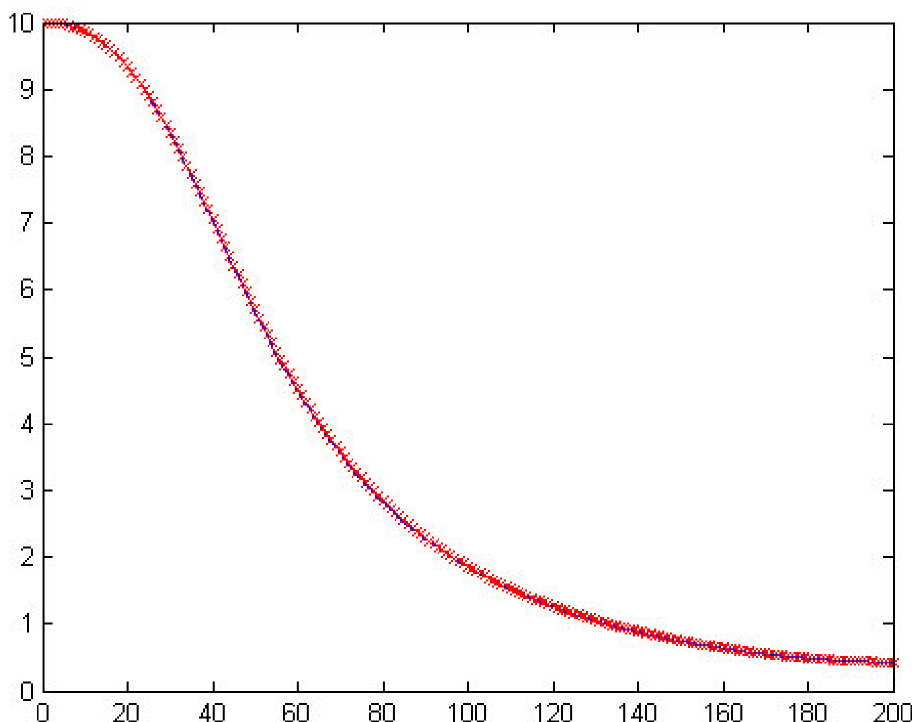
A systems transfer function H(z) evaluated around the 'unit circle' is a measure of a systems frequency response since H(w) is a subset of H(z). On a pole-zero diagram the unit circle occurs at values of $z = e^{j\theta}$, where θ corresponds to the frequency which has a normalised range of 0 to π radians/sample (or 0 to fs/2 Hz).

Using an earlier example, a system with `b` coefficients at 1 and 2; and `a` coefficients at 1, -1 and 0.3 will have a transfer function given by:

$$H(z) = \frac{z+2}{z^2 - z + 0.3}$$

The following evaluates this expression for 200 points around the unit circle i.e. theta is varied from 0 to pi in steps of pi/(200-1);

```
theta = 0 : pi/(200-1) : pi;
z = exp(j*theta);
H = (z + 2)./(z.^2 -z + 0.3);

%compare with freqz evaluated for 200 bins
b = [1 2];
a = [1 -1 0.3];
H2 = freqz(b,a, 200);
plot(abs(H));
hold on
plot(abs(H2),'rx')
hold off
```



Designing Filters

There are lots of filter design techniques available (fir1, chebyshev, butterworth, elliptical) and matlab has functions which design filters using these techniques.

Each filter design technique has its own set of pro's and con's (see http://answers.yahoo.com/question/index?qid=20101017195425AAxy47e for info on IIR filters) and the

choice of filter depends on the application of the filter.

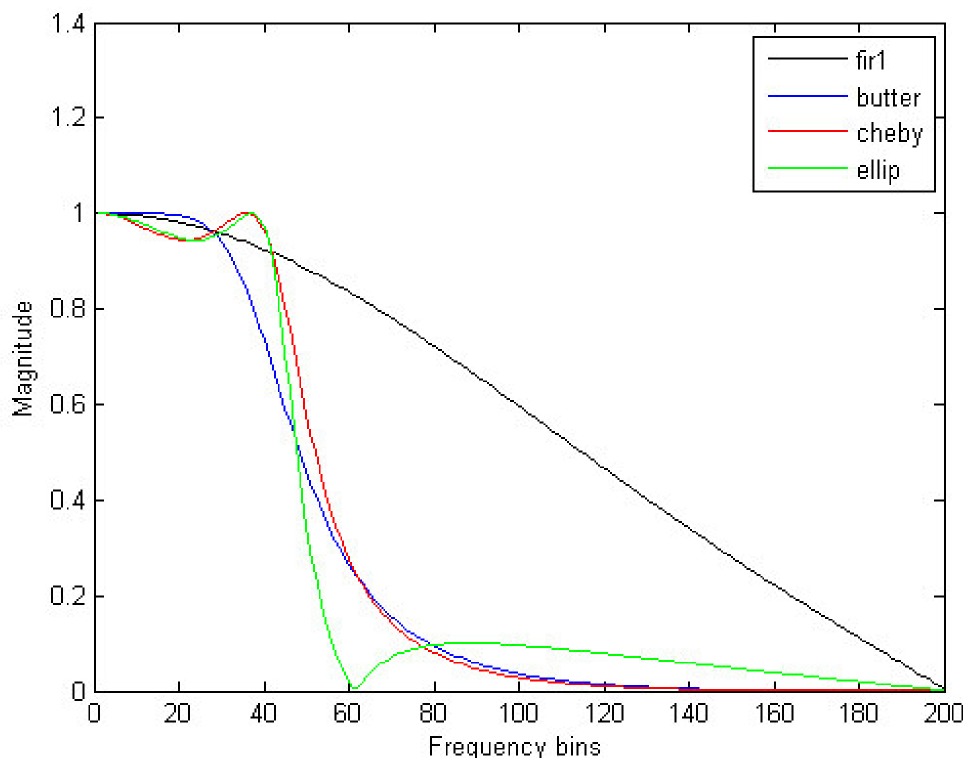
All the filter design functions return a set of `b` and `a` coefficients which define the system

Example 1

Low pass filter, cutoff 0.2 rads/sample, 3rd order. Designed using 3 different IIR design techniques, and an FIR technique. From the plots below it can be seen that Butterworth is maximally flat i.e. no ripple in the passband; chebyshev has ripple in the passband but none in the stopband; while elliptical has ripple in both the stopband and passband. With some application ripple may or may not be allowed. The benefit of allowing ripple is that the cutoff can be much sharper (with a corresponding trade-off in time-domain response). The FIR filter provides the worst response in terms of roll-off rate.

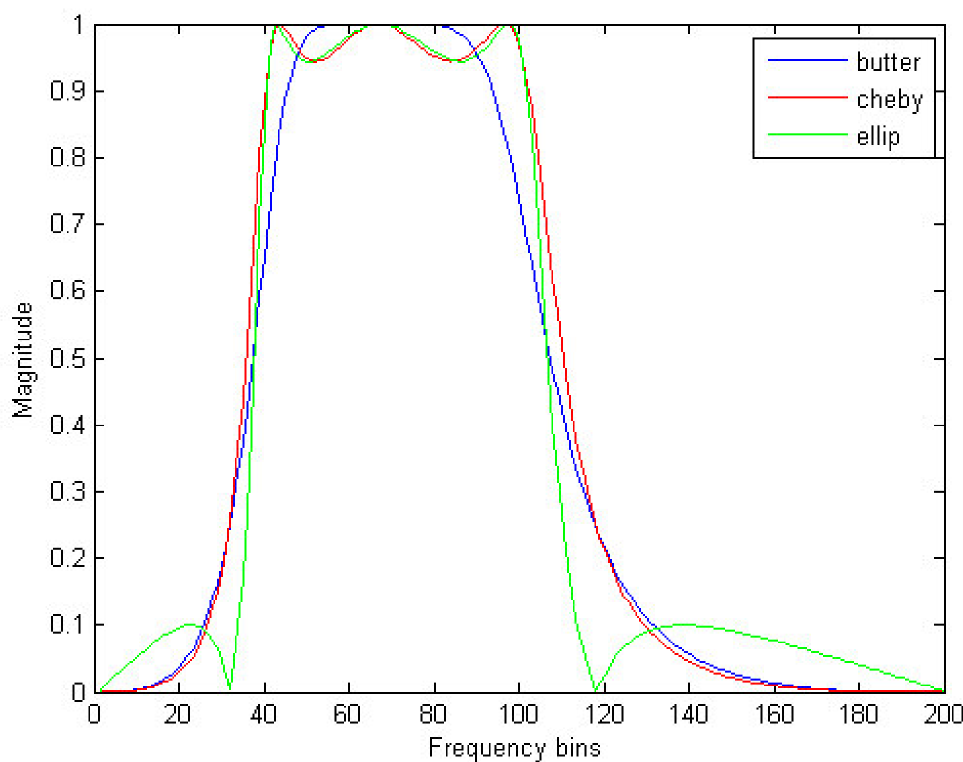
The order specifies the maximum number of `b` and `a` coefficients returned. Compare the two band pass filters below; the 6th order achieves a response that has much sharper cutoff

```
[b_fir1] = fir1(3, 0.2);
[b_butter, a_butter] = butter(3, 0.2);
[b_cheby, a_cheby] = cheby1(3, 0.5, 0.2); % 0.5 paramter specfies passband ripple
[b_ellip, a_ellip] = ellip(3,0.5, 20, 0.2); % 20 parameter specifies stopband ripple

H_fir1 = freqz(b_fir1, 1, 200);
H_butter = freqz(b_butter, a_butter, 200);
H_cheby = freqz(b_cheby, a_cheby, 200);
H_ellip = freqz(b_ellip, a_ellip,200);
plot(abs(H_fir1),'k')
hold on
plot(abs(H_butter))
plot(abs(H_cheby), 'r');
plot(abs(H_ellip), 'g');
xlabel('Frequency bins');
ylabel('Magnitude');
legend('fir1','butter', 'cheby', 'ellip')
hold off
```



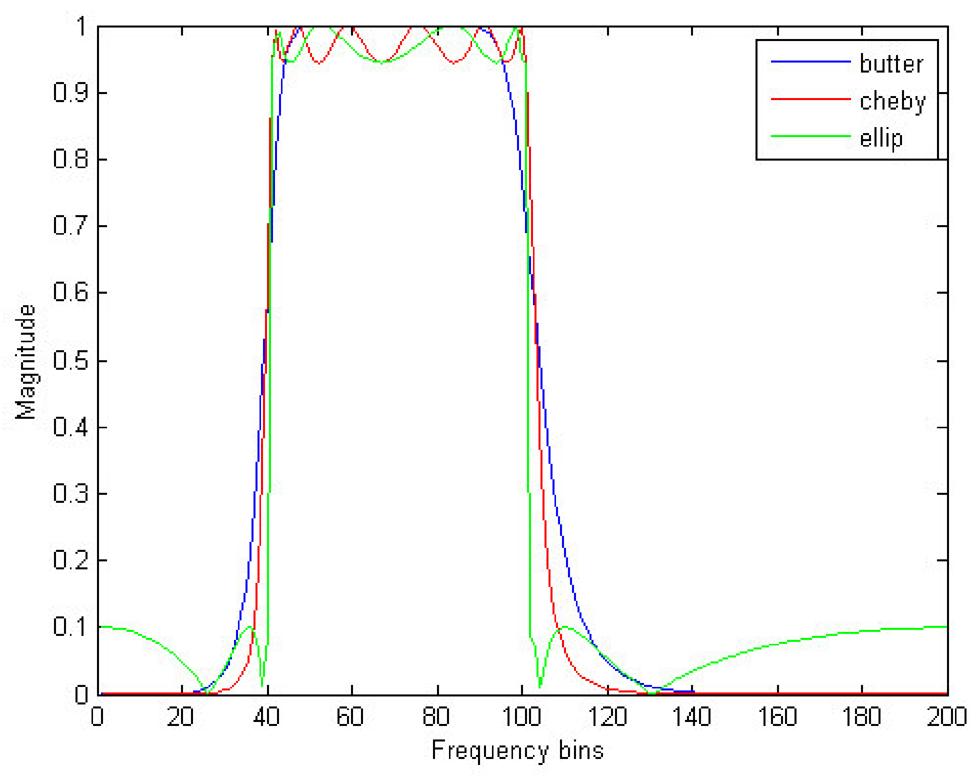
Design a bandpass filter using the three IIR techniques - 3rd order

```
[b_butter, a_butter] = butter(3, [0.2 0.5]);
[b_cheby, a_cheby] = cheby1(3, 0.5, [0.2 0.5]); % 0.5 paramter specfies passband ripple
[b_ellip, a_ellip] = ellip(3,0.5, 20, [0.2 0.5]); % 20 parameter specifies stopband ripple

H_butter = freqz(b_butter, a_butter, 200);
H_cheby = freqz(b_cheby, a_cheby, 200);
H_ellip = freqz(b_ellip, a_ellip,200);
plot(abs(H_butter))
hold on
plot(abs(H_cheby), 'r');
plot(abs(H_ellip), 'g');
xlabel('Frequency bins');
ylabel('Magnitude');
legend('butter', 'cheby', 'ellip')
hold off
```



Design a bandpass filter using the three IIR techniques - 6th order

```
[b_butter, a_butter] = butter(6, [0.2 0.5]);
[b_cheby, a_cheby] = cheby1(6, 0.5, [0.2 0.5]); % 0.5 paramter specfies passband ripple
[b_ellip, a_ellip] = ellip(6 ,0.5, 20, [0.2 0.5]); % 20 parameter specifies stopband ripple

H_butter = freqz(b_butter, a_butter, 200);
H_cheby = freqz(b_cheby, a_cheby, 200);
H_ellip = freqz(b_ellip, a_ellip,200);
plot(abs(H_butter))
hold on
plot(abs(H_cheby), 'r');
plot(abs(H_ellip), 'g');
xlabel('Frequency bins');
ylabel('Magnitude');
legend('butter', 'cheby', 'ellip')
hold off
```



Published with MATLAB® 7.11