WHOI Matlab Tutorials Summer 2020, Day 4: Signal Processing

07/07/2020

Goals:

1) Read in some audio data & plot it in the time domain

2) Analyze frequency components of the data using Fast Fourier Transform

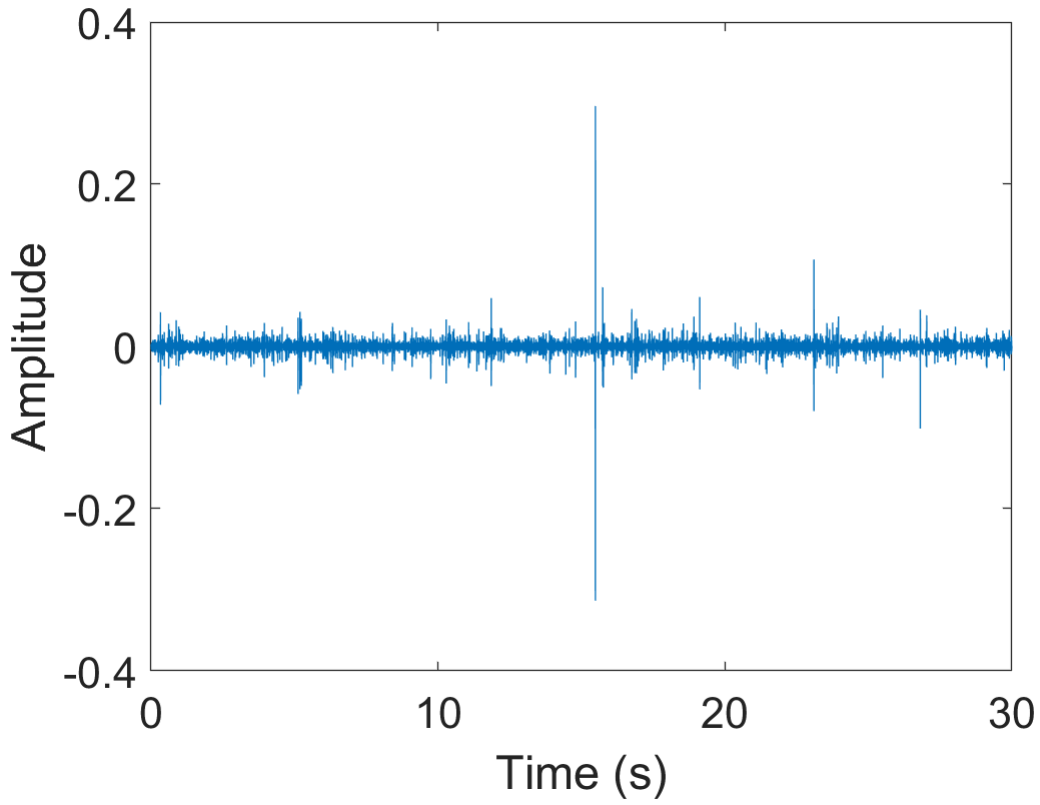
3) Plot data in frequency domain as a power spectrum (with pwelch function)

4) Make a spectrogram (shows data in time AND frequency domain)

5) Filter data (lowpass, highpass, bandpass)

1) Read in some audio data & plot in the time domain

```
clear
close all
clc

% Add folder .wav file is in to path:
addpath('C:\Users\Ian\Documents\MIT WHOI JP\MATLAB_WHOIcourse_Summer2020')

% Read in audio data:
[data, fs] = audioread('RamHeadReef_130802235647.wav'); % "data" here is your raw data,
                                                        % "fs" is the sample rate
N = length(data); % N is the total number of samples

% What are the units of data? Matlab by default reads in .wav data
% normalized from -1 to 1. In this case, these are 'raw' voltage data from
% a hydrophone, that has a certain sensitivity where X micropascals
% (a unit for sound pressure) are transduced to 1 V.
% But we won't worry about correcting the amplitude units for this example.

% Time vector for data, in seconds:
tvec = 0:1/fs:(N-1)/fs;


% Plot the waveform (amplitude vs. time):
figure
plot(tvec,data)
xlabel('Time (s)')
ylabel('Amplitude')
set(gca,'FontSize',16)
ylim([-0.4 0.4])
```



We can listen to the data using the soundsc function

```
soundsc(data,fs) % fs input makes sure it plays at the correct sample rate (speed)

% type:
%   clear sound
% in the command window if you want to stop it
```

2) Analyze frequency components of data using Fast Fourier Transform

According to the Nyquist Theorem, you can only resolve frequencies up to half of the sample rate, fs.

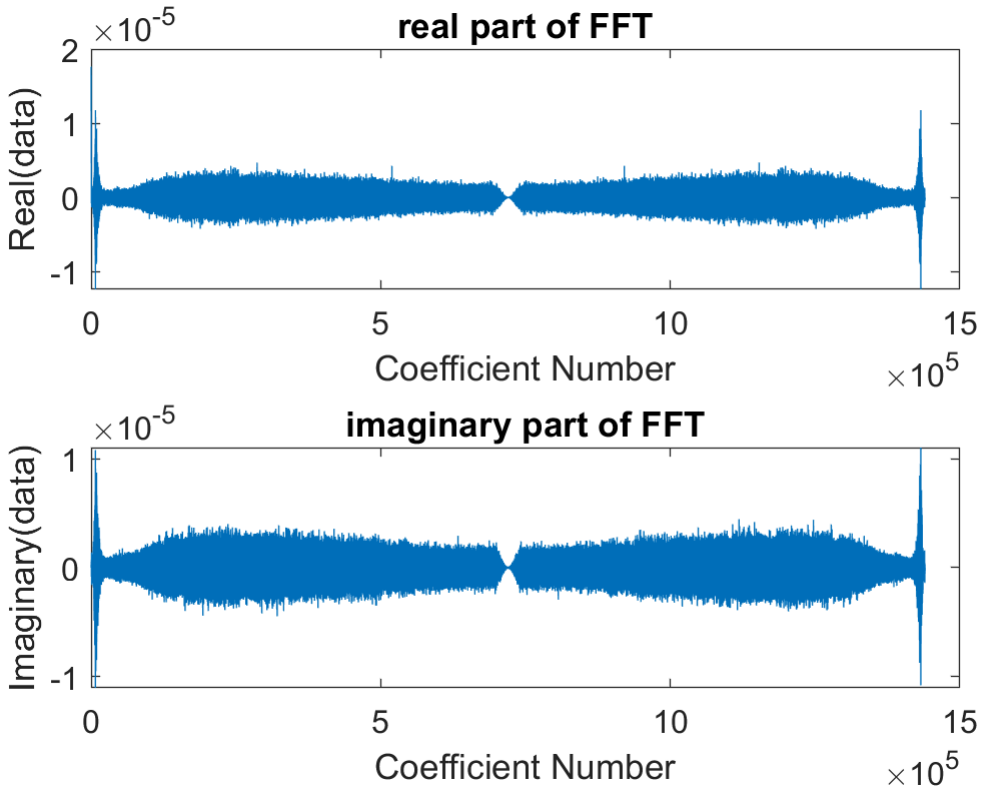
Thus for fs = 48000, we can only analyze frequencies up to 24000 Hz.

```
% Fast Fourier Transform:
FTdata = fft(data)/N; % normalization by N is by convention
```

The output of fft is supposed to give us amplitudes for each frequency, but its raw output is complex, i.e. it has real and imaginary values. Why? A Fourier Transform essentially breaks down your signal into individual sines and cosines for each frequency component (bin). The real part indicates how much of the frequency component represents a cosine wave, and the imaginary component indicates how much of it represents a sine wave.

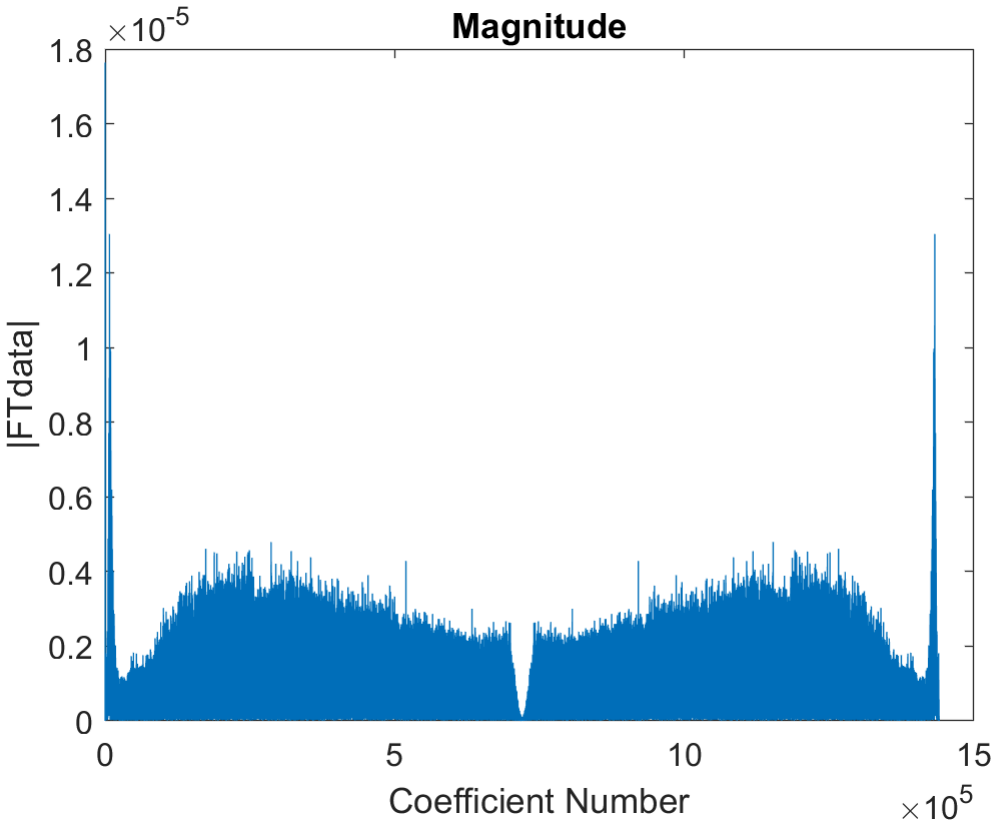
```
figure

subplot(2,1,1)
plot(real(FTdata))
```

```
xlabel('Coefficient Number')
ylabel('Real(data)')
set(gca,'FontSize',12)
title('real part of FFT')

subplot(2,1,2)
plot(imag(FTdata))
xlabel('Coefficient Number')
ylabel('Imaginary(data)')
set(gca,'FontSize',12)
title('imaginary part of FFT')
```



Plotting the real and imaginary components in this way is not how we usually look at frequencies of a signal. Usually we are interested in the magnitude (and sometimes also the phase). Let's focus just on the magnitude, and plot that next:

```
figure
plot(abs(FTdata)) % Taking the absolute value gives us the (complex) magnitude
title('Magnitude')
set(gca,'FontSize',12)
xlabel('Coefficient Number')
ylabel('|FTdata|')
```

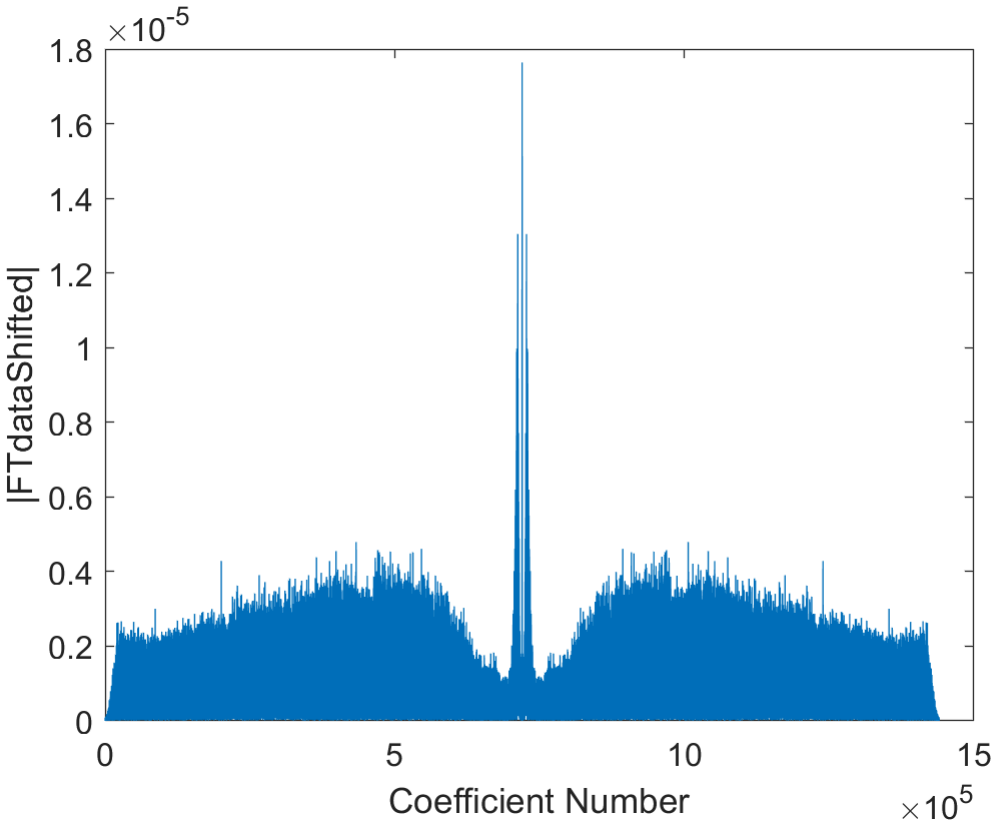


So now we have the magnitude, but how are the FFT coefficients on the X axis organized?

The fft output has a DC component at 0, and negative and positive frequencies. But it places them in a weird order: DC, followed by increasing positive frequencies, followed by increasing negative frequencies, like in this example with N = 16 samples:

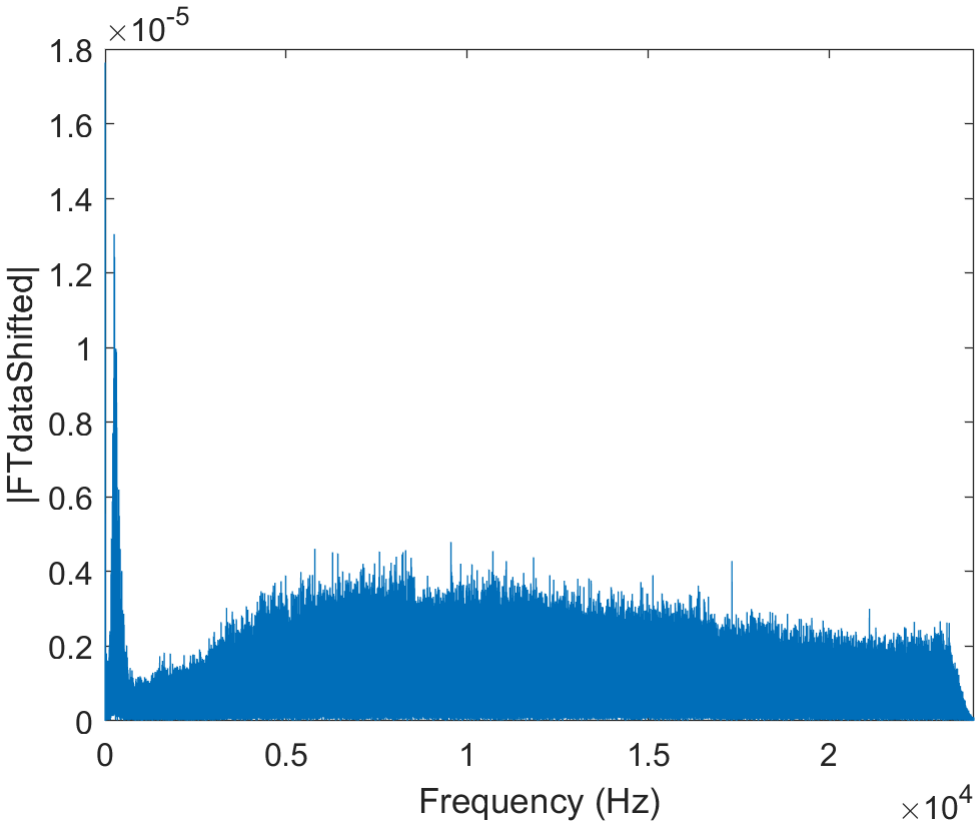
Use fftshift to get a more intuitive frequency order: increasing negative frequencies, followed by DC component, followed by increasing positive frequenices:


```
figure
FTdataShifted = fftshift(FTdata);
plot(abs(FTdataShifted));
set(gca,'FontSize',12)
xlabel('Coefficient Number')
ylabel('|FTdataShifted|')
```



We still have the x-axis units in 'Coefficient Number' though. Let's change them to Hz:

```
% Correct the x-axis to frequency unit Hz:
df = fs/N; % Default frequency resolution of fft function, in Hz
Fvec = (-floor(N/2):ceil(N/2)-1)*df;
figure
plot(Fvec,abs(FTdataShifted))
set(gca,'FontSize',12)
xlabel('Frequency (Hz)')
ylabel('|FTdataShifted|')

% For real-world applications, we just want to report the positive
% frequencies:
xlim([0 Fvec(end)]);
```



Using fft in this way gave us very small frequency bins (0.0333 Hz), and as a result the magnitude vs. frequency is pretty 'noisy'.

One solution to this would be to use 'movmean' function to smooth out the data.

Typically we would report magnitude vs. frequency in what's called a "power spectrum", in larger frequency bins; for acoustics, 1 Hz is a common bin size. All this can actually be done in just one line of code using a function like "pwelch":

3) Plot data in frequency domain as a power spectrum:

pwelch computes Fourier Transforms internally and outputs 'power' (i.e. amplitude squared) for a series of time windows you define, then averages them. This is a good way to show the average frequency content of a signal. Inputs to the pwelch function are:

1) Your data to input

2) Window length, in number of samples, to divide the signal over. You can also specify a window type; here "hann" is the window type, short for "Hanning" window,... it's just one of many possible window types. Here the window length is 1 s = 48000 samples = fs

3) Number of samples of overlap between windows (using overlap gives you a "sliding window" average). Here, overlap is set to 0.5 s (0.5*fs), i.e. 50% overlap.

4) Specify "nfft", aka your FFT size, in samples:

fs/nfft = frequency resolution (in Hz);

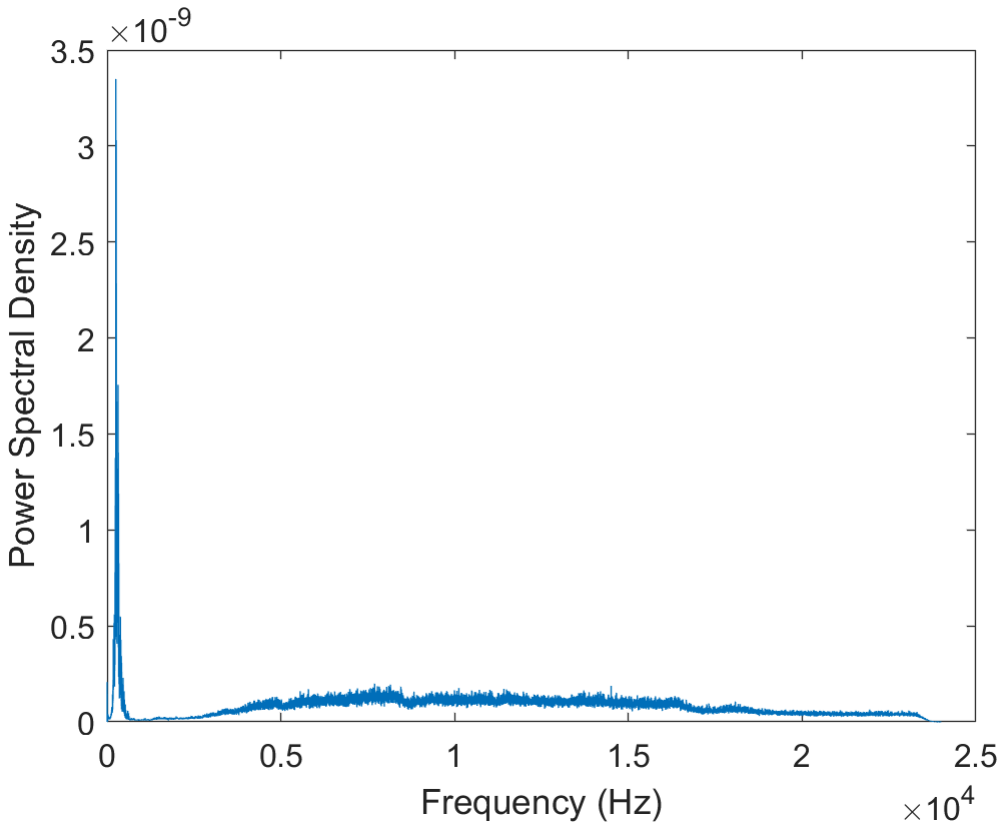
nfft/fs = temporal resolution (in s);

nfft must be a positive integer scalar. If nfft = fs, you will get output in 1 Hz bins.

5) Sample rate of your data, fs

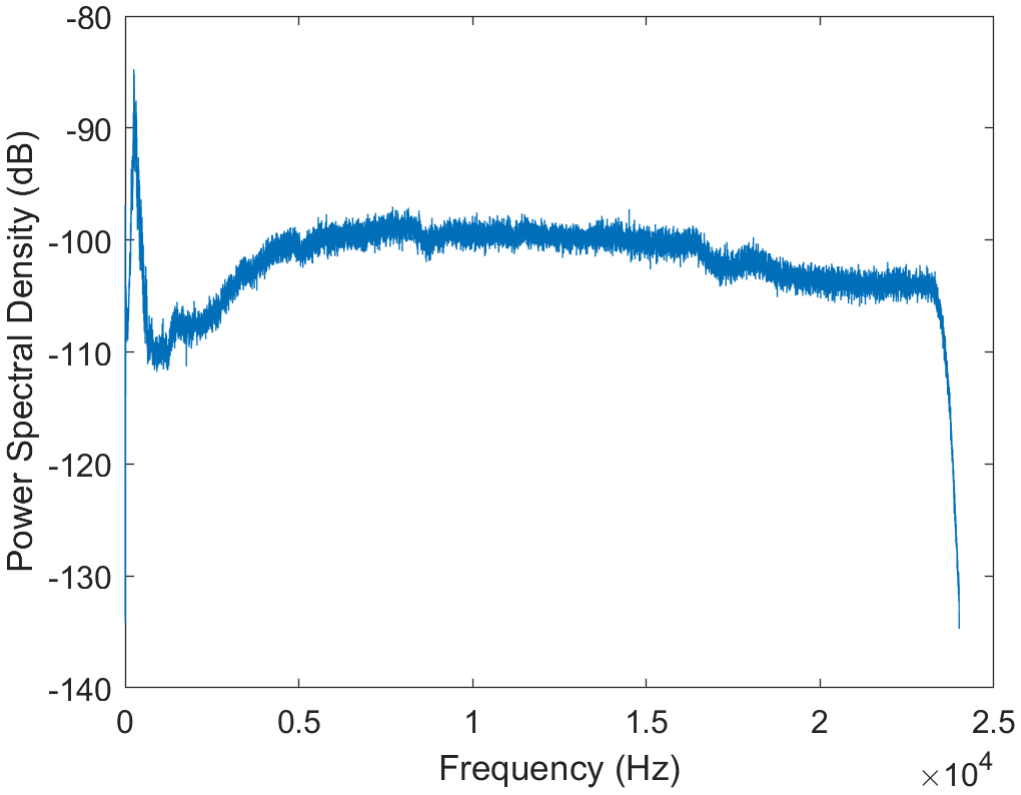
6) input 'onesided' to get data for just the positive frequencies.

```
[PSD,f] = pwelch(data,hann(fs),0.5*fs,fs,fs,'onesided');

figure
plot(f,PSD)
set(gca,'FontSize',12)
ylabel('Power Spectral Density')
xlabel('Frequency (Hz)')
```



In the case of audio data, amplitudes will span many orders of magnitude, so it's standard to report magnitudes on a dimensionless log scale in order to compress those values to a narrower range. In acoustics, we use the 'decibel', or dB scale:

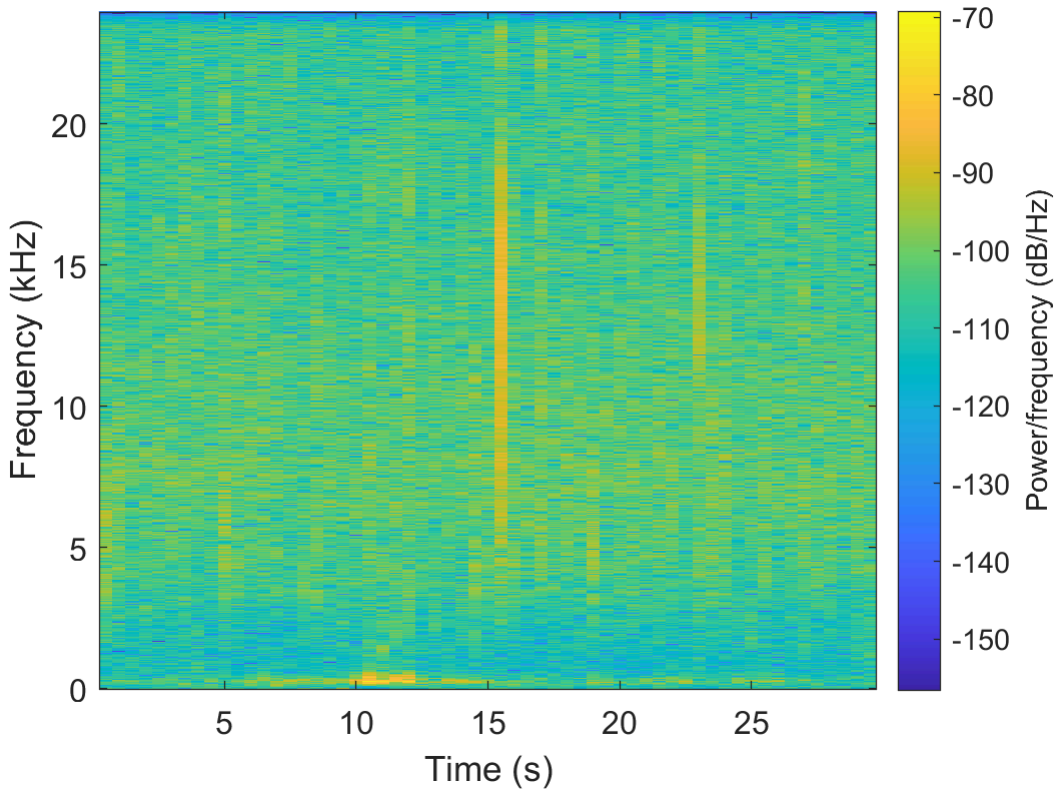
```
figure
plot(f,10*log10(PSD)) % 10*log10(x) is a standard equation to convert power to dB scale
set(gca,'FontSize',12)
ylabel('Power Spectral Density (dB)')
xlabel('Frequency (Hz)')
```



4) Make a spectrogram (shows data in time AND frequency domain)

The spectrogram function works in a similar way as pwelch, where FFTs are calculated for discrete time windows (usually with overlap), but shows the frequency content across time as well, instead of averaging across all time windows.

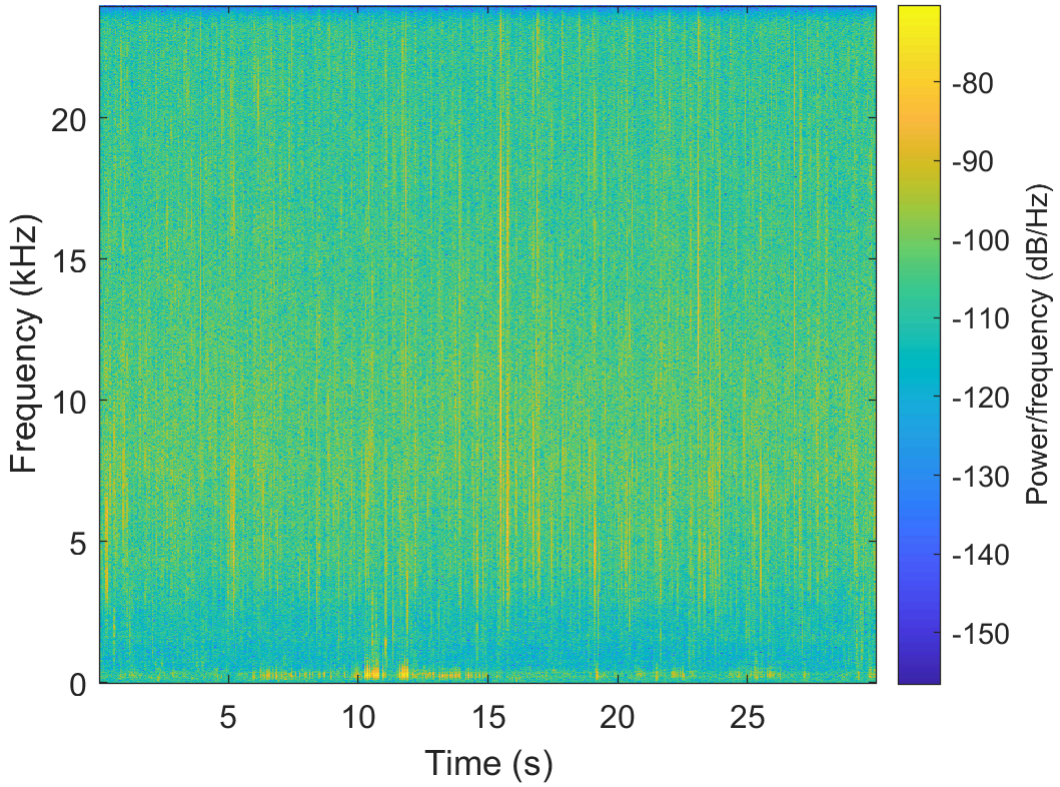
```
% Inputs to spectrogram here are the same as pwelch, plus the 'yaxis' option
% which puts the frequency axis as the yaxis (as is standard)
figure
spectrogram(data,hann(fs),fs*0.5,fs,fs,'onesided','yaxis');
set(gca,'FontSize',12)
```



With these options, the frequency resolution is pretty fine (1 Hz bins), and with 50% overlap we actually have 0.5 s bins (with no overlap we would have 1 s bins). If we want to improve the temporal resolution it will be at the expense of frequency resolution (and vice versa).

Let's make another spectrogram with finer time resolution by using a smaller nfft and window size:

```
figure
spectrogram(data,hann(fs/10),(fs/10)*0.5,fs/10,fs,'onesided','yaxis');
set(gca,'FontSize',12)
```

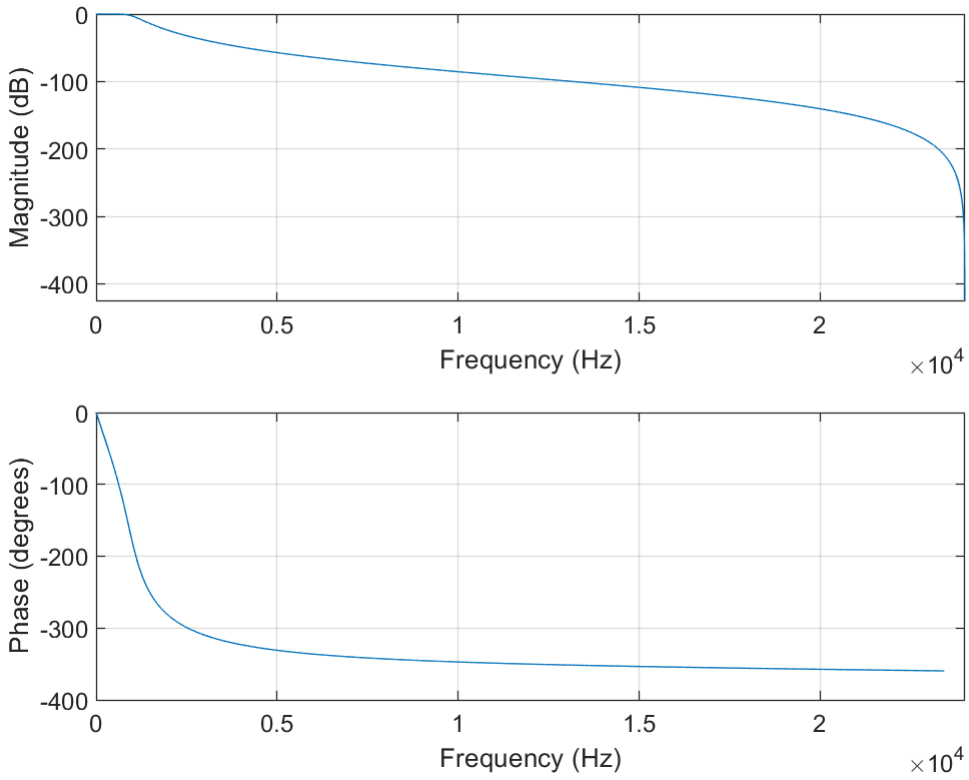


Now we can see the finer temporal structure of short duration sounds, like broadband (i.e. spanning many frequencies) pulses, which are snapping shrimp snaps. If we zoom in on the lowest 1000 Hz, we can can get a better look at the fish sounds that show up in yellow around 10-15 s and at around 300 Hz.

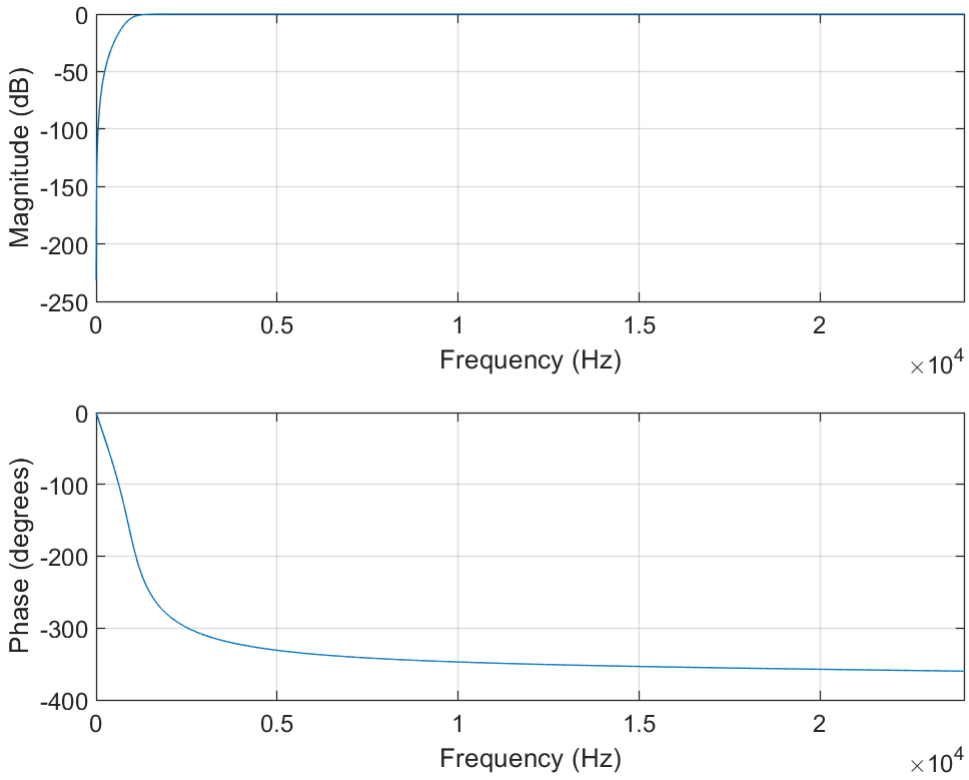
5) Basic filtering with a Butterworth filter

These examples use a filter function where you input your time series data, so you don't have to worry about calculating FFT as a separate step.

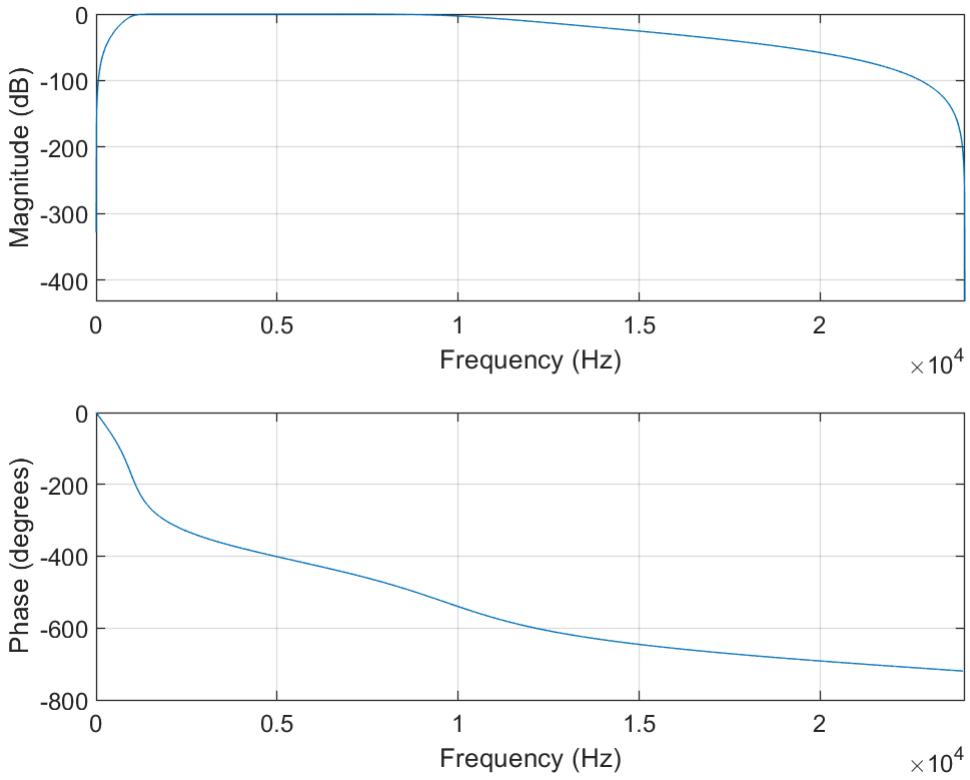
```
% Low pass filter:
Wn = 1000/(fs/2);  % Filter cutoff frequencies normalized from 0 to 1, where
                   % the maximum is the Nyquist frequency (fs/2, the highest
                   % frequency we can resolve).
[b1,a1] = butter(4,Wn,'low');  % b1 and a1 are transfer function coefficients,
                               % 1st input is filter order.

% Use the freqz function to visualize the filter:
figure
freqz(b1,a1,fs,fs) % 3rd and 4th inputs of give the sample rate,
                   % and show the x-axis units in Hz.
```



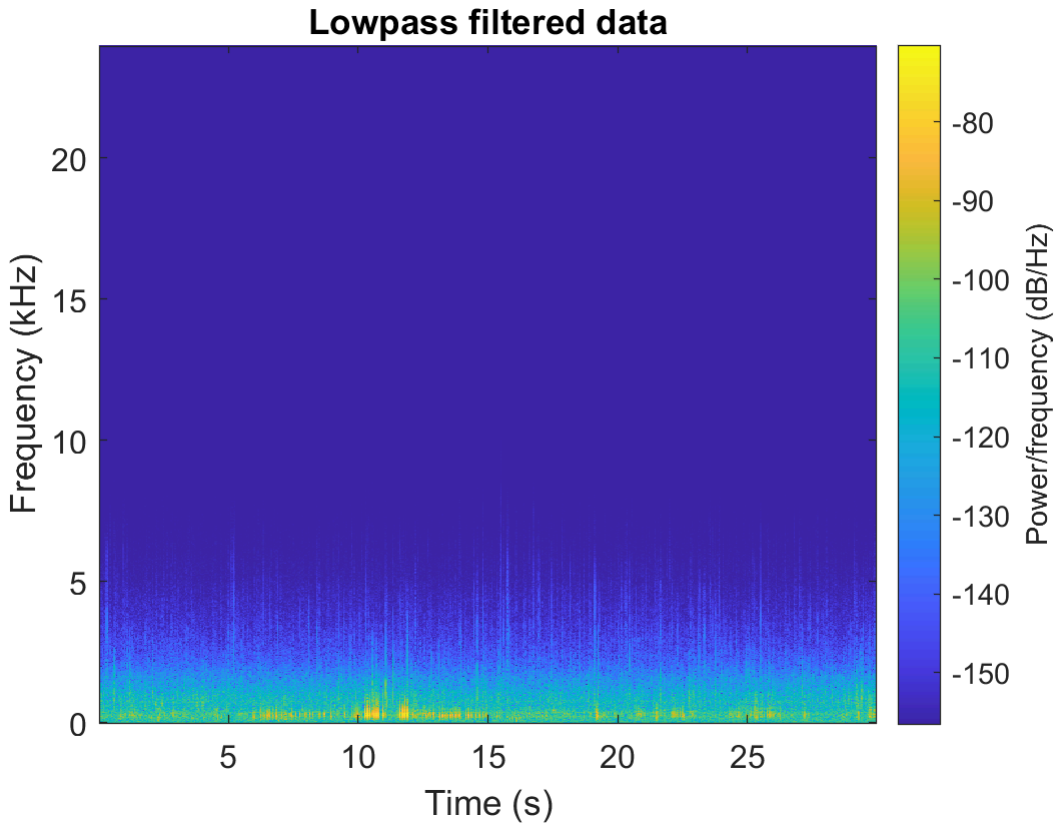

```
% High pass filter:
Wn = 1000/(fs/2);
[b2,a2] = butter(4,Wn,'high');
figure
freqz(b2,a2,fs,fs)
```



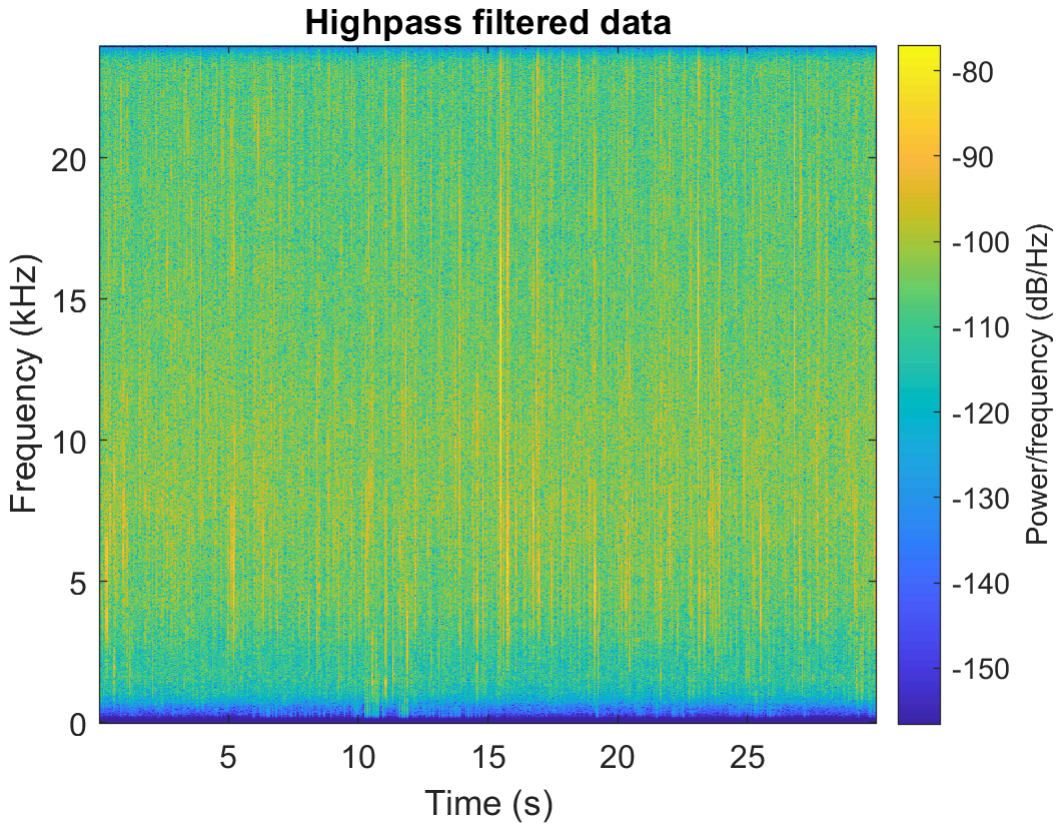

```
% Bandpass filter:
Wn = [1000/(fs/2) 10000/(fs/2)];
[b3,a3] = butter(4,Wn,'bandpass');
figure
freqz(b3,a3,fs,fs)
```



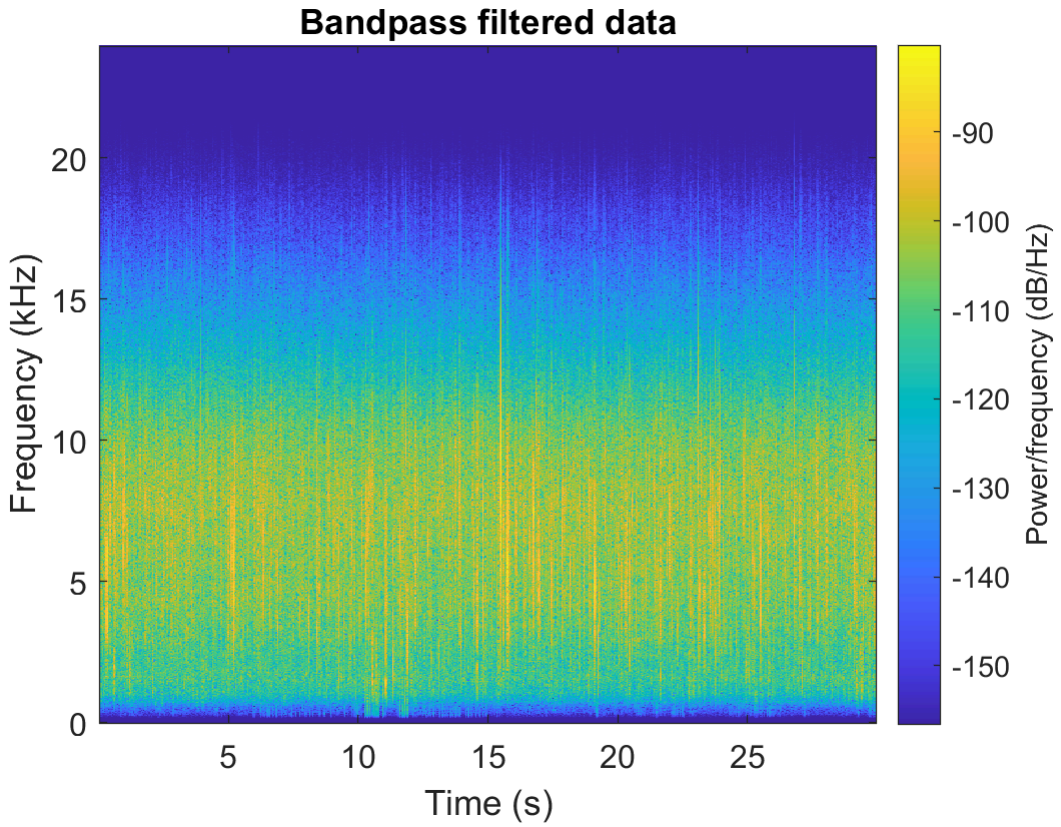

```
% Use the filter function to filter your data
dataFiltlow = filter(b1,a1,data);
dataFilthigh = filter(b2,a2,data);
dataFiltband = filter(b3,a3,data);

% Let's visualize how the filters change the spectrograms:
spectrogram(dataFiltlow,hann(fs/10),(fs/10)*0.5,fs/10,fs,'onesided','yaxis');
set(gca,'FontSize',12)
title('Lowpass filtered data')
```




```
spectrogram(dataFilthigh,hann(fs/10),(fs/10)*0.5,fs/10,fs,'onesided','yaxis');
set(gca,'FontSize',12)
title('Highpass filtered data')
```




```
spectrogram(dataFiltband,hann(fs/10),(fs/10)*0.5,fs/10,fs,'onesided','yaxis');
set(gca,'FontSize',12)
title('Bandpass filtered data')
```



6) Other useful signal processing functions:

'ifft' performs the inverse FFT (getting data back to the time domain). You may want to get back to the time domain for example, after you filtered the data in the frequency domain. If you used fftshift, you'll have to use ifftshift first:

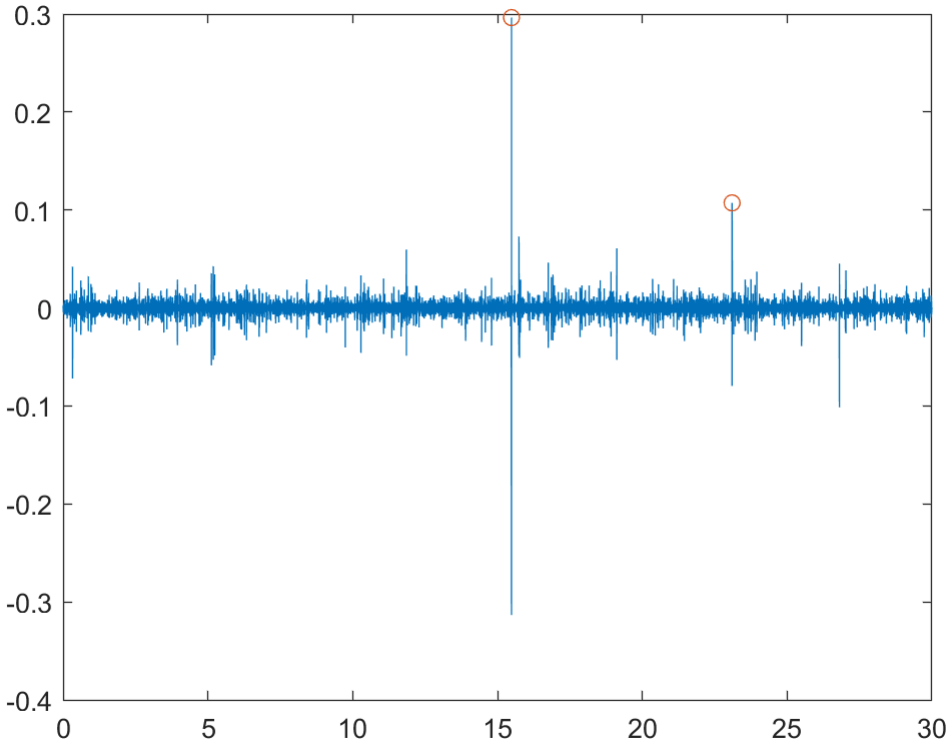
```
FTdataNew = ifftshift(FTdataShifted); % Undo shift
dataNew = ifft(FTdataNew*N); % send to time domain
```

'resample' allows you to change the sample rate of your data: useful if you sampled at a much higher rate than needed and want to reduce the sample rate to make spectral analysis faster:

```
fs_new = fs/2;
data_downsampled = resample(data,fs_new,fs);
```

'findpeaks' is useful for finding local maxima in your time series:

```
[pks,locs] = findpeaks(data,'MinPeakDistance',fs*1,'MinPeakHeight',0.1);
figure
plot(tvec,data)
hold on
scatter(locs/fs,pks) % plot the peaks
```



Search "Signal Processing Toolbox" in the Matlab Documentation for more functions and tutorials!