# Armstrong Atlantic State University
# Engineering Studies
# MATLAB Marina – Sound Processing Primer

**Prerequisites**

The Sound Processing Primer assumes knowledge of the MATLAB IDE, MATLAB help, arithmetic operations, built in functions, scripts, variables, arrays, logic expressions, conditional structures, iteration, functions, debugging, characters and strings, cell arrays, structures, and file input and output. Material on these topics is covered in the MATLAB Marina Introduction to MATLAB module, MATLAB Marina Variables module, MATLAB Marina Arrays module, MATLAB Marina Logic Expressions module, MATLAB Marina Conditional Structures module, MATLAB Marina Iteration module, MATLAB Marina Functions module, MATLAB Marina debugging module, MATLAB Marina Character and Strings module, MATLAB Marina Cell Arrays module, MATLAB Marina Structures module, and MATLAB Marina File Input and Output module.

**Learning Objectives**

1. Be able to use MATLAB to load and save audio signals (.wav files).
2. Be able to generate audio signals that are tones and sums of tones.
3. Be able to implement signal processing operations.
4. Be able to use MATLAB to determine the spectrum of sampled signals.

**Terms**

sinusoid, amplitude, frequency, phase, sampling, sampling frequency, filtering, signal processing, spectrum, spectrogram, Fourier transform, FFT

**MATLAB Functions, Keywords, and Operators**

wavread, wavwrite, audioread, audiowrite, audioinfo, conv, filter, fft, fftshift, sound, soundsc, audioplayer, play

**Sinusoidal Signals**

The general form of a sinusoid with constant frequency is $x(t) = A\cos(\omega t + \phi)$, where A is the amplitude, $\omega$ is the radian frequency (radians/sec), and $\phi$ is the phase (radians). The frequency f in Hertz (cycles/sec) can be found from the radian frequency via $\omega = 2\pi f$. The period T of a sinusoid is the inverse of the frequency, $T = \frac{1}{f} = \frac{2\pi}{\omega}$. The sinusoid $x(t) = A\cos(\omega t + \phi)$ with phase shift $\phi$ can also be represented by a sinusoid with time shift $t_d$, $x(t) = A\cos(\omega(t - t_d))$, where $t_d = -\frac{\phi}{\omega}$.

The general form for a signal that is a sum of sinusoids is $x(t) = A_0 + \sum_{k=1}^{N} A_k \cos(\omega_k t + \phi_k)$, where $A_0$ is the DC term and $A_k$, $\omega_k$, and $\phi_k$ are the amplitude, radian frequency and phase shift of the kth sinusoid in the sum of sinusoids.

**Plotting Sinusoidal Signals**

To obtain a smooth plot in MATLAB of a sinusoidal signal one needs 10 to 25 points per period of the signal. For signals composed of a sum of sinusoids, one needs 10 to 25 points per period of the highest frequency sinusoid to obtain a smooth plot.

For example to plot two periods of the signal $x(t) = \cos(200\pi t)$:

- The frequency of the signal is $\omega = 200\pi$ radians/sec or $f = 100$ Hz.
- The amplitude is 1 and there is no phase shift.
- The period of the sinusoid is $T = \frac{1}{f} = 0.01$ seconds and to obtain 25 points per period, the interval between points should be $T/25 = 0.0004$ seconds.

```
f = 100;    % frequency of sinusoid
T = 1/f;    % period of sinusoid
deltat = T/25;
% generate time vector and sinusoidal signal
t = 0.0:deltat:2*T;
x = cos(2*pi*f*t);
% plot sinusoid
figure(1)
plot(t, x), grid, xlabel('t (sec)'), ylabel('x(t)');
```

Figure 1, MATLAB Code to Plot Sinusoid

**Sampling**

Sampling involves periodically measuring a signal. The rate of measurement is called the sampling frequency. The sampling frequency $f_s$ is the number of samples (measurements) taken per second. To properly represent a signal composed of a sum of sinusoids, one must sample at least twice the highest frequency in the signal, i.e. $f_s \geq 2 f_{\max}$ (Nyquist, Shannon).

A signal $x(t) = 3\cos(200\pi t) + \cos(500\pi t) + \cos(2000\pi t)$ has a maximum frequency of $f_{\max} = 1000$ Hz. To properly sample the signal (signal is unambiguously represented by its samples), one must sample the signal at $f_s \geq 2 \cdot 1000 = 2000$ Hz.

The process of sampling takes a continuous time signal $x(t)$ and generates a sequence of numbers $x[n]$ corresponding to the samples. The entire sequence is referred to as $x[n]$ and the kth sample, denoted $x[k]$, is the value $x(kT_s) = x(k / f_s)$.

Consider the previous plotting example of Figure 1. The MATLAB signal x is really a sampled signal and the sampling frequency used to obtain a smooth plot is 10 to 25 times the highest frequency of the signal. The time vector used to evaluate the sinusoid of the plotting example of Figure 1 could also be created using sampling notation.

```
f = 100;    % frequency of sinusoid
T = 1/f;    % period of sinusoid
fs = 25*f;    % sampling frequency
% generate time vector and sinusoidal signal
t = 0.0:1/fs:2*T;
x = cos(2*pi*f*t);
% plot sinusoid
figure(1)
plot(t, x), grid, xlabel('t (sec)'), ylabel('x(t)');
```

Figure 2, MATLAB Code to Plot Sinusoid (Time Vector via Sampling)

Because most plotting software linearly interpolates between points to generate the plot curves, we need a higher sample rate to obtain smooth plots than is needed to accurately represent the signal according to the sampling theorem.

**Audio Signals**

Digital audio signals can be obtained by sampling and quantizing analog audio signals or by generating them directly using computer hardware/software.

Common digital audio formats are:

- Uncompressed formats: WAV, AU, PCM
- Lossless compressed formats: FLAC, ALAC (Apple lossless),
- Lossy compressed formats: MP3, AAC

Along with the audio samples, digital audio files typically contain information about the sampling frequency, number of channels, and bits per channel. WAV files (waveform audio file format) are audio files that contain a header that provides information on the number of channels (1 for mono, 2 for stereo), sample rate $f_s$, and bits per sample (8 or 16). The remainder of the wave file is the audio samples. WAV files can store compressed data but are more commonly used to store uncompressed audio data. WAV files support multiple channels, sample rates up to the GHz range, and data sizes from 8 bits to 32 bits.

CD quality audio is stereo (two channel), 44.1 kHz, and 16-bit LPCM data (LPCM stands for linear pulse channel modulation). For LPCM, data is represented as a sequence of amplitude values using linear quantization. The 16-bit data for CD audio is stored as a two's complement signed integers. A CD quality audio stream requires a 44.1k*2*16 = 1411.2 kbits/sec audio stream.

The MATLAB functions `wavread` and `wavwrite` can be used to read audio data from WAV files and write audio data to WAV files. The MATLAB functions `audioread` and `audiowrite` can be used to read audio data from a variety of audio file formats including WAV and MP3.

The MATLAB code segment of Figure 3 gives an example of reading from a WAV file and plotting the time domain samples. Typical use of `wavread` involves three return values. In the example of Figure 3, `dg` is a column vector of the audio samples, `fs` is the sampling frequency, and `numBits` is the number of bits per sample. If the WAV file has multiple channels, the audio sample variable will be a multidimensional array. MATLAB will typically represent the data read from a wave file as double precision real numbers, although using the optional `fmt` string `'native'` in `wavread`, one can read in the data as `uint8` or `int16` data. The `audioread` function operates similar to `wavread` but it only returns the samples and sampling frequency. The `audioinfo` function can be used to obtain the number of bits per sample.

```
% dog growl audio signal, samples come in as column vector
[dg, fs, numBits] = wavread('doggrowl.wav');
% create time vector from number of samples and sample rate
t = ((0:1:length(dg)-1)/fs)';
% plot dog growl signal
figure(1)
plot(t,dg),xlabel('t (sec)'), ylabel('x(t)'), title('Dog Growl')
```

Figure 3, MATLAB Code to Read from WAV File

The MATLAB functions `wavwrite` and `audiowrite` allow one to save data to an audio file. The typical use of `wavwrite` and `audiowrite` are

```
wavwrite(sampleData, fs, numBits, filename);
audiowrite(filename, sampleData, fs, 'BitsPerSample', numBits);
```

These MATLAB statements will write the data stored in the variable `sampleData` to the specified filename. The data to be written should be column data.

The MATLAB functions `sound`, `soundsc`, and `audioplayer` are used for playing audio data. The `sound` function plays the signal by sending it directly to the computer's sound card. Data is assumed to be real numbers in the range of -1 to 1 and data outside this range is clipped. The `soundsc` function works the same as the `sound` function except the data is scaled so that the signal is played as loud as possible without clipping. The `audioplayer`

function returns an audioplayer object which can be played using the `play` function. There is also a legacy `wavplay` function that only works in 32-bit Windows operating systems and will be eliminated in future MATLAB releases.

**Signal Processing**

A filter is a system designed to modify the components of signals. A filter takes a sequence of input samples $x[n]$ and transforms them into a sequence of output samples $y[n]$ according to some rule. One commonly used filter is a finite impulse response (FIR) filter. FIR filters are typically represented by a difference equation or convolution sum.

$$y[n]=\sum_{k=0}^{M} b_k x[n-k]$$

Where $x[n]$ is the input sequence, $y[n]$ is the output sequence, $x[n-D]$ is $x[n]$ delayed by D samples, and $b_k$ are the filter weights or filter coefficients. The number of nonzero weights is $L=M+1$, where M is the filter order. The filter order corresponds to the maximum delay of the system.

Computing the output of a FIR filter operation requires a nested loop, an outer loop to run through the samples n and an inner loop to run through the summation index of the FIR computation for sample n. If the filter order is small, for example 2, the inner loop can be omitted and the filtering for each sample implemented with a single line of code

$$y[n]=b_0 x[n]+b_1 x[n-1]+b_2 x[n-2]$$

MATLAB has several built in functions for working with discrete-time systems (a FIR filter is a discrete-time system). MATLAB has built in functions: `conv` and `filter` for operating on one dimensional signals and `conv2` and `filter2` for operating on two dimensional signals like images. Figures 4a and 4b show MATLAB code for computing the output of an order 2 FIR filter $y[n]=\frac{1}{3}x[n]+\frac{1}{3}x[n-1]+\frac{1}{3}x[n-2]$ to the input x[n].

```
% input signal
fs = 10000; % sampling frequency
nn = 0:1:99;
x = 5*cos(2*pi*(500/fs)*nn) + cos(2*pi*(3000/fs)*nn);
% filter coefficients
b = ones(1,3)/3;
% filter using MATLAB filter function
y = zeros(1,length(x));
y = filter(b,1,x);
```

Figure 4a, FIR Filtering using `filter` Function

MATLAB's `filter` function, `y = filter(b,a,x)`, filters the input x with the filter described by the coefficients in the vectors b and a and generates the output y. For FIR filters, the b vector is the vector of FIR coefficients and the a vector should be a single one (a = 1).

```
% FIR filtering using nested loop, first few samples
% are special cases as have delay of 2
yloop = zeros(1,length(x));
yloop(1) = b(1)*x(1);
yloop(2) = b(1)*x(2) + b(2)*x(1);
for n = 3:1:length(x)
    for k = 1:1:length(b)
        yloop(n) = yloop(n) + b(k)*x(n-(k-1));
    end
end
```

Figure 4b, FIR Filtering Implementation Evaluating Function first M Times Outside of Loop to avoid Using Negative Indices, Same x[n] as Figure 4a

For high order filters is not practical to compute the first M outputs separately outside the loop. Figure 4c shows an alternative to the code of Figure 4b using a conditional statement to avoid computation using inputs prior to n = 0 which for this example is when the input starts.

```
yloop2 = zeros(1,length(x));
for n = 1:1:length(x)
    for k = 1:1:length(b)
        if (n-k >= 0)
            yloop2(n) = yloop2(n) + b(k)*x(n-(k-1));
        end
    end
end
```

Figure 4c, FIR Filtering Implementation using Conditional Structure to Avoid Negative Indices, Same x[n] as Figure 4a

Since MATLAB does not support zero or negative indices the loops start at 1 rather than 0 and go one higher than in the formulas shown previously (k runs from 1 to M+1 rather than 0 to M).

A more efficient way of handling the problem of negative indices is to pad the input with M zeros and use the padded input in place of the input. One must then shift the indices in the padded input by M when using them in the FIR filtering code as xpad[n+M] corresponds to x[n]. The implementation of Figure 4d also omits the inner loop as the filter length is small and the filter formula can be coded in one line.

```
% FIR filtering, padded input
xpad = [zeros(1,length(b)-1), x];
y2 = zeros(1,length(x));
for n = 3:1:length(x)+2
    y2(n-2) = b(1)*xpad(n) + b(2)*xpad(n-1) + b(3)*xpad(n-2);
end
```

Figure 4d, FIR Filtering Implementation using Padding, Same x[n] as used in Figure 4a

**Spectrum**

Physical world signals such as music or speech are composed of many frequency components and it is usually easier to characterize complex signals such as music and speech in terms of their frequency content rather than time domain characteristics. The analysis and design of signal processing systems is often done in the frequency domain as it if often much easier to describe the effect of a system on the signal's spectrum.

The spectrum of a signal is its frequency content, given as set of complex-amplitude frequency pairs or as a plot of amplitude and phase versus frequency. For signals that are a sum of sinusoids, the spectrum can be found via inspection. For most periodic signals, the spectrum of the signal can be determined from the signal's Fourier series representation. For signals that are not sums of sinusoids nor periodic, we can usually use the Fourier transform to determine the spectrum of the signal.

For signals that are a sum of sinusoids $x(t) = A_0 + \sum_{k=1}^{N} A_k \cos(\omega_k t + \phi_k)$ the spectrum is the frequency, amplitude, and phase information of each sinusoid in the signal.

For example, the spectrum of $x(t) = 2 + 3\cos(200\pi t) + \cos(500\pi t) + \cos(2000\pi t)$ is $\{(0\text{ Hz}, 2)(100\text{ Hz}, 3), (250\text{ Hz}, 1), (1000\text{ Hz}, 1)\}$.

The spectrum can also be represented as a plot of the complex amplitude of each sinusoidal component versus frequency. For signals that are a sum of sinusoids $x(t) = A_0 + \sum_{k=1}^{N} A_k \cos(\omega_k t + \phi_k)$, this will give N+1 lines, where each line corresponds to one of the sinusoidal components of the signal and the height of line will be equal to the magnitude of that sinusoidal component's complex amplitude. Often the magnitude and phase of the complex amplitude will be plotted separately.

The magnitude plot for the signal $x(t) = 2 + 3\cos(200\pi t) + \cos(500\pi t) + \cos(2000\pi t)$ is shown in Figure 5.

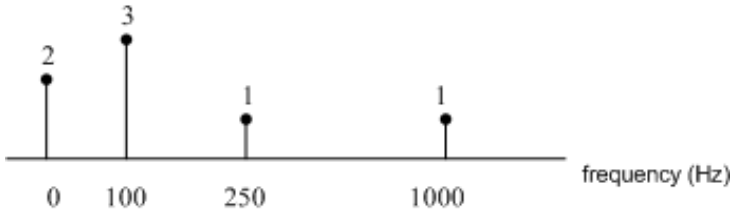

Figure 5, Spectrum Plot, Magnitude Portion

**DFT and FFT**

The discrete Fourier Transform (DFT) takes a sequence of numbers and transforms it into complex numbers corresponding to the complex amplitudes of the sinusoidal components of the sequence of numbers. The sequence of numbers is typically the samples of a sampled signal and the DFT gives the frequency content (spectrum) of the signal represented by the samples. In practice the Fast Fourier Transform (FFT) rather than the DFT is used for computing the spectrum of sampled signals. The FFT is a more efficient version of the DFT and in some cases generates less numerical error.

The discrete spectrum Xk produced by the FFT is the same length as the signal, i.e. the FFT will produce an N point discrete spectrum for a signal of length N. The FFT produces two copies the spectrum, the first N/2 values of Xk correspond to the complex amplitudes for the positive frequencies (0 to fmax) and the remaining values correspond to the negative frequencies (-fmax to 0). The maximum frequency that can be represented in the spectrum of a signal with N samples and duration T is N/(2T). The frequency resolution (how close of frequencies can be distinguished from each other) depends on the duration of the signal. The longer the duration of the signal, the better the frequency resolution in the computed spectrum and the more points (higher sample rate), the higher the maximum frequency in the spectrum is.

The MATLAB `fft` function will compute the FFT. The `fft` function takes a time domain signal (sampled signal) and returns the discrete spectrum. Figure 6a shows a MATLAB program that computes the spectrum of the signal $x(t) = 2 + 3\cos(200\pi t) + \cos(500\pi t) + \cos(2000\pi t)$ using the `fft` function. Figure 6b shows the signal and Figure 6c shows the spectrum of the signal.

The maximum frequency represented in the computed spectrum of the program of Figure 6a is `specmax` (25k Hz). The maximum frequency that can be represented is often higher than the highest frequency of the signal and it may make sense to zoom in on the frequency range of interest, in this case only the spectrum from 0 to 3000 Hz is plotted. The frequency resolution of the spectrum computed by the program of Figure 6a is `df` (1.529 Hz). Note that the amplitudes in the spectrum are scaled but the relative amplitudes are what we expect for the spectrum of this signal. The DC term is scaled by N and the other terms by N/2.

The FFT is more efficient if the number of points is an integer power of 2. The MATLAB function `nextpow2` determines the next higher power of 2 which is useful for FFT operations.

```
% sampled signal with duration one second
fmax = 2000;
fs = 25*fmax;
t = 0 : 1/fs : 1.0;
x = 2 + 3*cos(2*pi*100*t) + cos(2*pi*250*t) + cos(2*pi*1000*t);

% spectrum of signal
N = length(x);
df = 1/t(end);      % frequency interval in the spectrum
specmax = (df*N/2); % maximum frequency in the spectrum
f = [0:1:N-1]*df;   % vector of frequency values for plotting
Xk = fft(x);        % spectrum values

% plot signal and magnitude of spectrum
timerange = 1:1:floor(length(x)/10);
freqrange = find(f <= 3000);
figure(1)
subplot(2,1,1), plot(t(timerange),x(timerange)), ...
    xlabel('t (sec)'), ylabel('x(t)')
subplot(2,1,2),plot(f(freqrange),abs(Xk(freqrange))), ...
    xlabel('f (Hz)'), ylabel('|Xk(jw)|')
```

Figure 6a, MATLAB Code for Computing Spectrum

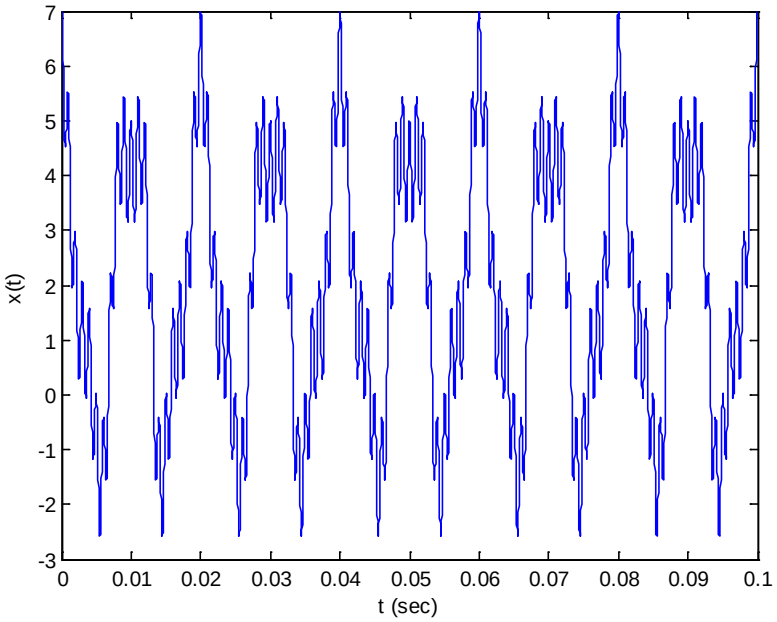


Figure 6b, Signal (0 to 0.1 seconds)

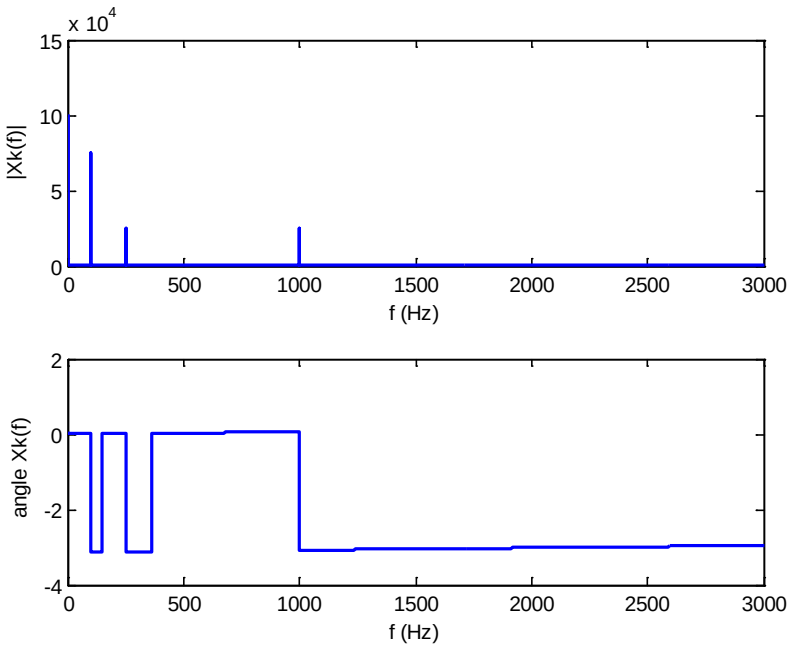


Figure 6c, Spectrum of Signal (0 to 3000 Hz)

The MATLAB program of Figure 7a illustrates determining the spectrum of a signal read in from a WAV file. Figure 7b shows the time domain plot and Figure 7c the magnitude portion of the spectrum of the signal from the doggrowl.wav file. Note in the spectrum of Figure 7c that both copies of the spectrum are being displayed (spectrum from 0 to 12.5k Hz and mirrored spectrum from 12.5 to 25k Hz). Also notice in the time domain plot the signal varies with time and we should expect the spectrum to vary with the signal variation.

```
% dog growl audio signal, samples come in as column vector
[dg, fsdg, numBits] = wavread('doggrowl.wav');
% create time vector from number of samples and sample rate
L = length(dg);
tdg = ((0:1:L-1)/fs)';
% spectrum of dog growl
N = 2^nextpow2(L);
df = fs/N;                  % frequency resolution in Hz
fdg = (0:1:N-1)*df;
DGk = fft(dg,N)/L;          % N point fft with magnitudes scaled
% plot dog growl signal and spectrum
figure(2)
subplot(2,1,1), plot(tdg,dg), ...
    xlabel('t (sec)'), ylabel('x(t)'), title('Dog Growl')
subplot(2,1,2), plot(fdg,abs(DGk),'LineWidth',2), ...
    xlabel('f (Hz)'), ylabel('|DGk(f)|'), title('Dog Growl Spectrum')
```

Figure 7a, MATLAB Program for Spectrum of Dog Growl

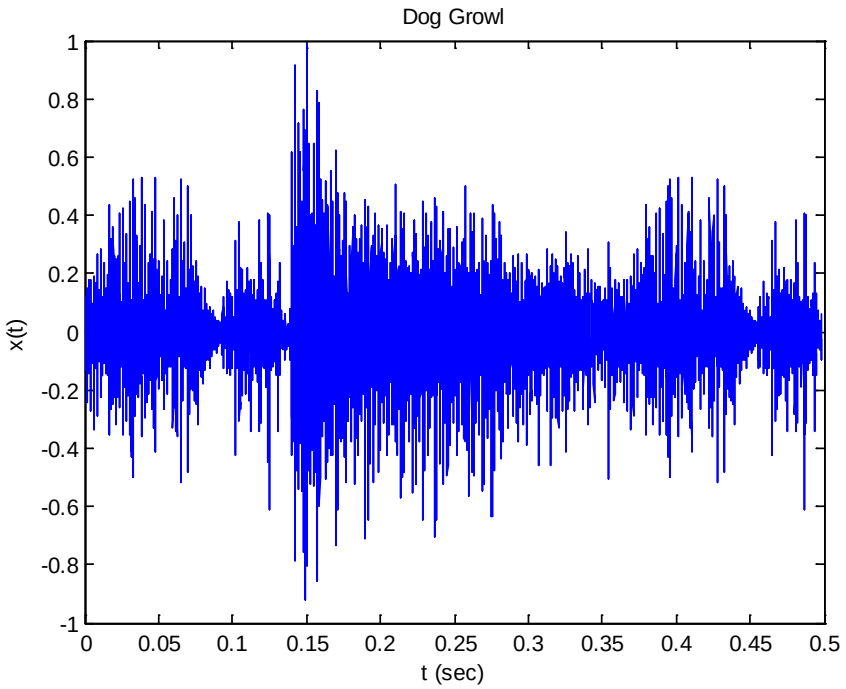


Figure 7b, Dog Growl Signal

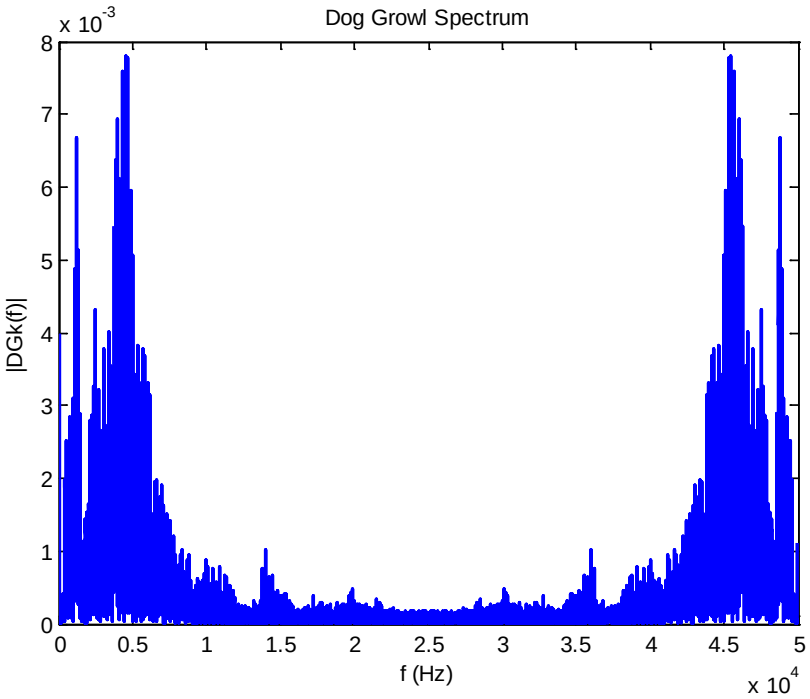


Figure 7c, Magnitude Spectrum of Dog Growl Signal

**Spectrogram**

Time varying spectrums are typically handled by performing frequency analysis on a windowed signal. The window is moved along the signal and the frequency analysis is repeated for the portion of the signal in the window. This is continued for the entire time range of the signal. A plot of a time varying spectrum is called a spectrogram. Typically in a spectrogram, the x-axis will be time, the y-axis frequency and either color or intensity used to indicate the energy at the frequency.

MATLAB has a built in function `spectrogram` (part of the Signal Processing toolbox) for computing the spectrogram of signals. The `spectrogram` function takes four or five arguments in the typical use

```
S = spectrogram(xx, window, noverlap, nfft)
```

or

```
S = spectrogram(xx, window, noverlap, nfft, fs)
```

Where `xx` is the signal, `window` is either the window length if a single value or the window values if a vector, `noverlap` is number of points of overlap for each segment, `nfft` is number of points for the fixed FFT of each segment, `fs` is the sampling frequency in Hz, and `S` is a 2D array with each column holding the short time spectrum X[k] at time n (time increases across columns and frequency increases down rows. The window length and `nfft` are typically the same value and the default window is a Hamming window. The number of points of overlap, `noverlap`, must be less than the window length and is typically 50% to 80% of the window length. The default for MATLAB's spectrogram plots is to display time on the y axis and frequency on the x axis. This can be switched by using an additional argument `'yaxis'`, `spectrogram(xx, windowlength, noverlap, nfft, fs,'yaxis')` to get time on the x-axis and frequency on the y-axis.

The window length and `nfft` should be an integer power of 2 (the windowed spectrum is computing using the FFT) for the best performance and the sampling frequency should be the same as used to generate the discrete-time signal.

Last modified Friday, November 01, 2013