CENG 384 - Programming Assignment

In this homework, you will display and analyze sound signals in the time domain and in the frequency domain. The tasks and explanations are provided in the form of a Jupyter Notebook file, which you can download from here (https://colab.research.google.com/drive/19TGCQ1FdgSVQNX1PFnKhoS3uxSoJ_UnB). To see it with the drawings, you should run it in the directory you can download from the COW page of the course.

Sound samples and the Python file

The sound samples that you can use in this assignment and a Python starter file are provided to you at the COW page of the course.

Submission instructions

Run file `collect_submission.sh` , which will create you a `ceng384_pHW.zip` file. Upload this file at the COW page of the course.

Deadline

17th of May, 23:55

Testing Environment

Your solutions will be tested on inek machines (running Ubuntu 18.04) with Python v2.7. For the libraries, minimum requirements are: Numpy v1.11 , Matplotlib v1.5.1, Scipy 0.17.

In [12]:

```
# Some setup utils
import numpy as np
import matplotlib.pyplot as plt
import scipy

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipytho
n
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Warm-up: Generating, Plotting and Analyzing a Simple Signal

Let us start warming up by generating a simple signal with the numpy library (check this (https://docs.scipy.org/doc/numpy/user/quickstart.html) for a tutorial). Numpy is a very talented library for working with numerical data structures, like complex numbers, vectors, matrices etc.

In the following cell, we will first construct a simple continuous-time signal:

$$x(t) = \cos(2\pi t) + \sin(40\pi t).$$

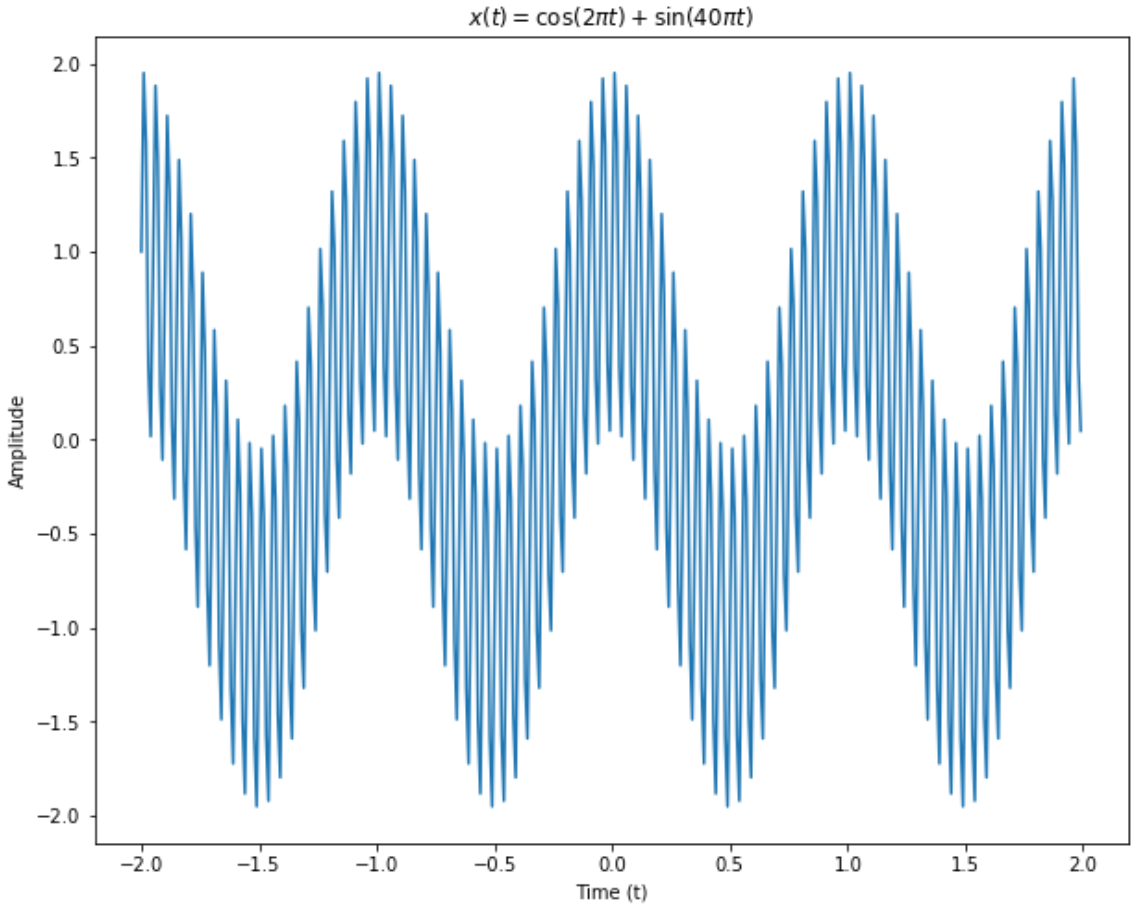
For plotting the signal, we will use the matplotlib library (check this (https://matplotlib.org/users/pyplot_tutorial.html) for a tutorial).

In [13]:

```
t = np.arange(-2, 2, .01) # Sample a time interval between -2 and 2
x = np.cos(2*np.pi*t)+np.sin(40*np.pi*t)

plt.plot(t, x)
plt.xlabel('Time (t)')
plt.ylabel('Amplitude')
plt.title('$x(t) = \cos(2\pi t) + \sin(40\pi t)$')
```

Out[13]:

```
Text(0.5,1,'$x(t) = \\cos(2\\pi t) + \\sin(40\\pi t)$')
```



See, that's very easy. You can just construct any signal in this manner and plot it.

In the previous step, we actually constructed a discrete-time signal and drew that as if it were a continuous-time signal. Now, let us directly work with a discrete-time signal.

In the following cell, we will construct and plot a simple discrete-time signal:

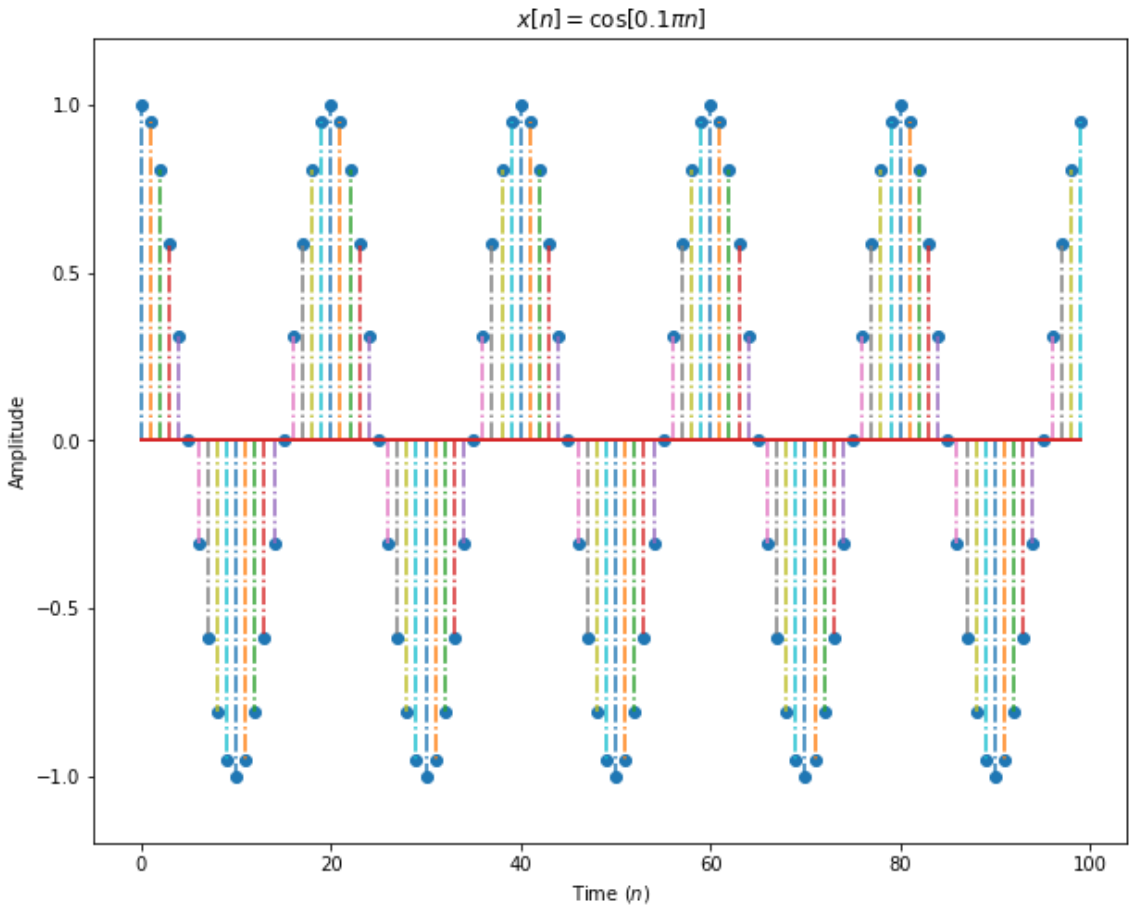
$$x[n] = \cos[0.1\pi n].$$

In [14]:

```
n = np.array(range(0, 100))
# np.linspace(0.1, 2*np.pi, 10)
markerline, stemlines, baseline = plt.stem(n, np.cos(0.1 * np.pi * n), '-.')
plt.setp(baseline,'linewidth', 2)
plt.xlabel('Time ($n$)')
plt.ylabel('Amplitude')
plt.title('$x[n]=\cos[0.1 \pi n]$')
plt.ylim(-1.2, 1.2)
```

Out[14]:

```
(-1.2, 1.2)
```



Nice! So colorful and periodic :)

Serious Stuff: Load and Plot a WAV file

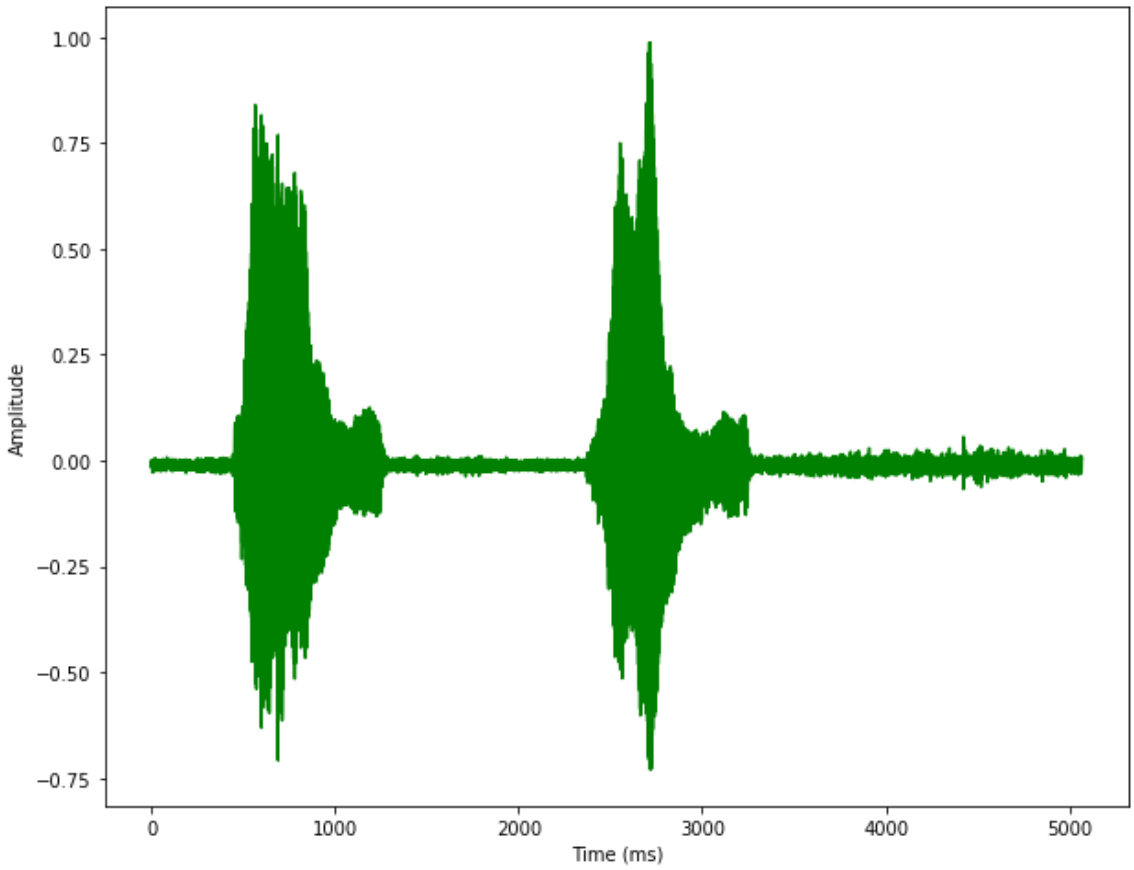
A standard format for storing audio/sound signals is the Waveform Audio Format (WAV in short -- see here (https://en.wikipedia.org/wiki/WAV) for more info). Don't worry, we will not ask you to load a WAV file from scratch. In the following step, we provide you a routine for loading and plotting a WAV file.

Under `"./sound_samples"` directory, there are many other sound samples. Load them and check how they look.

In [15]:

```
from ceng384_prog_hw import *

filename = 'sound_samples/cat.wav'

# Create a Sound object
signal = Sound(filename, verbose=True)

# Let us plot our signal
signal.plot_sound()
```

```
Sound() is finishing with:
         source file:  sound_samples/cat.wav
         sampling rate (Hz):  44100
         duration (s):  5.06777777778
         number of samples:  223489
```



The Discrete Fourier Transform

The sound signal that you have just loaded and plotted is a discrete-time, finite-interval signal. We can transform such signals into the frequency domain using Discrete Fourier Transform, defined as follows:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-jk\frac{2\pi}{N}n}, \qquad k = 0, 1, 2, \ldots, N-1$$

where N is the length (the number of samples) of our signal x.

Task 1: Implement Fourier Transform

Open file `"ceng384_prog_hw.py"` and complete the implementation of function "dft".

In [16]:

```
signal = Sound(filename)

# Let us test your function with the first 100 values of the sound signal
x = signal.data[0:100]

dft_naive = dft(x)
```

Check your implementation

Using the Fast Fourier Transform (FFT) implementation of the Numpy library, let us check how good your implementation was.

In [17]:

```
# FFT from Numpy:
tic()
dft_fast = np.fft.fft(x)
toc()
tic()
dft_naive = dft(x) # No need to recalculate, but let's repeat it just for clarit
y
toc()

# You should see a total difference less than 10^-10
error = 0
for i in range(len(dft_naive)):
    error += np.abs(dft_naive[i]-dft_fast[i])

print "The difference (Euclidean distance) is: ", error
print "Your are good to go!" if error < 10**-10 else "FAILED! Your implementatio
n has too much error. Go back and check for errors"
```

```
Elapsed time is 0.000288009643555 seconds.
Elapsed time is 0.0527560710907 seconds.
The difference (Euclidean distance) is:  1.8173222994242019e-13
Your are good to go!
```

Spectograms

Since sound signals can be too long, composed of many small-segment units of sounds, analyzing a whole sound signal with a single set of spectrum is not practical. A good spectral tool for sound signals is to use a spectogram.

A spectogram is essentially a matrix composed of DFT of small intervals of the input. Formally, the spectogram $\mathcal{S}\{x[n]\}$ can be calculated as follows:

$$\mathcal{S}\{x[n]\}[i] = \mathcal{F}\{x[i \times s : i \times s + W]\},$$

where W is the length of the interval for which we are calculating the DFT, and $\mathcal{F}\{x[n]\}$ is the DFT of $x[n]$. This is illustrated in the following drawing.

According to this definition, $\mathcal{S}[0]$ is a vector composed of the DFT of $x[0 : W - 1]$, and $\mathcal{S}[1]$ is then the DFT of $x[s : s + W - 1]$.

The "jump" amount between window positions, s, is called the stride.

Task 2: Calculate Spectogram

Open file `"ceng384_prog_hw.py"` again and complete the implementation of function "calculate_spectogram". Note that the function takes a DFT calculating function as an argument. When you pass your dft() function as an argument, you will realize that it is very very slow. To speed things up, we will continue with the FFT function from numpy.

In [18]:

```
window_size = 500
stride = 100
x = signal.data
Fs = signal.sampling_rate
duration = signal.duration

tic()
S = calculate_spectogram(x, window_size=window_size, stride=stride, dft_function
=np.fft.fft)
toc()

# Let's print the shape of spectogram just to make things clear
print "S.shape: ", S.shape
```

```
Elapsed time is 1.045317173 seconds.
S.shape:  (2230, 499)
```

Plotting the Spectograms

Now let us plot the spectogram. For visualization purposes, we will do many tricks:

1) Since the Fourier Transform is complex, we will just look at the magnitude.

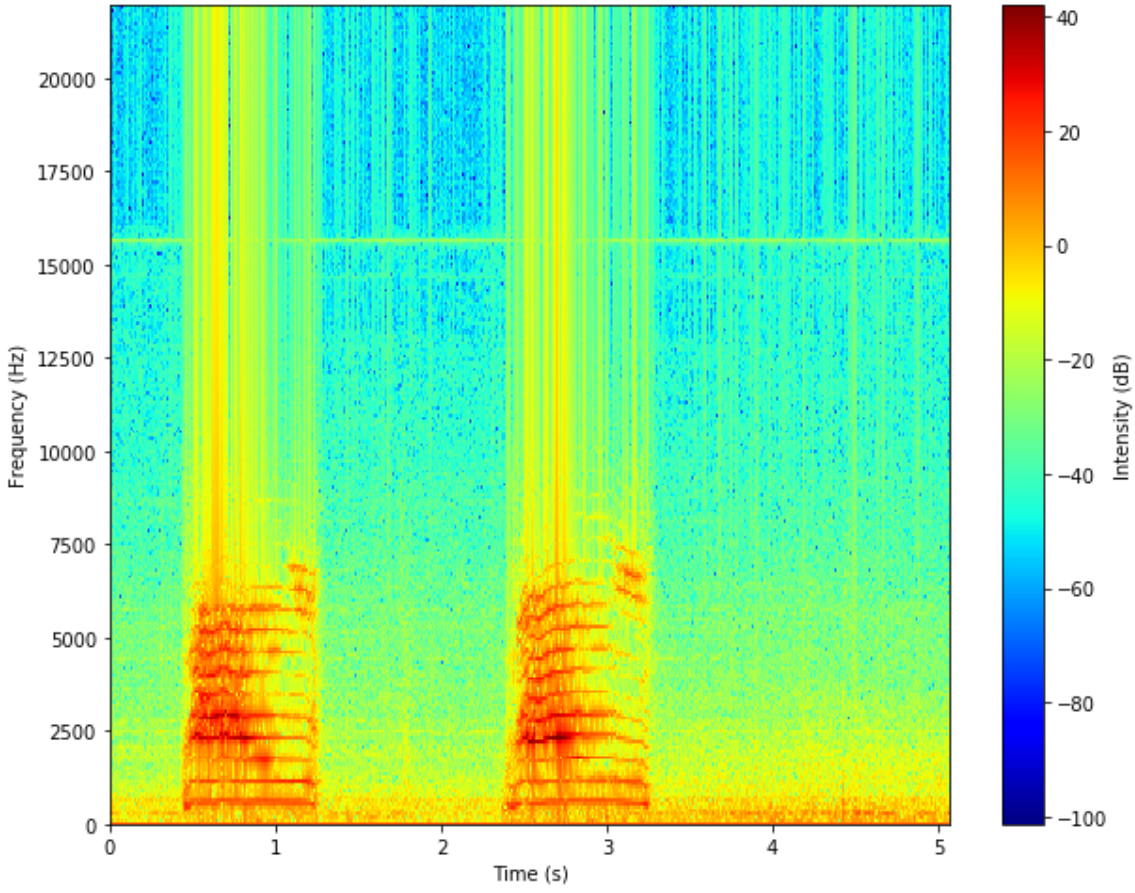
2) Since the Discrete Fourier Transform is symmetric, we will just visualize a half.

3) We will use a logarithmic scale (in terms of decibels) for the magnitude, to better see the whole range.

Although these steps make the main frequencies/components more visible, they may also exaggerate the small components which may appear as artifacts.

In [19]:

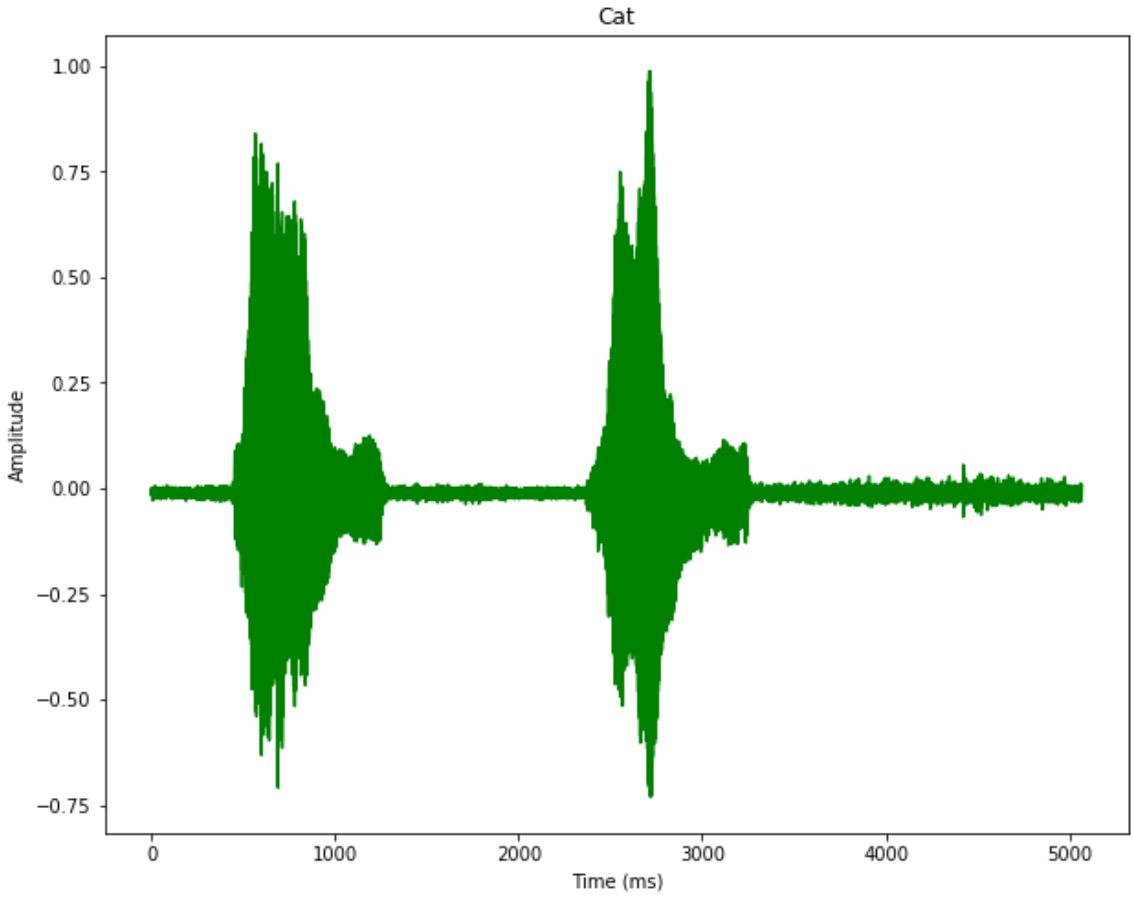
```
plot_spectogram(S, stride, window_size, duration=duration, Fs=Fs)
```

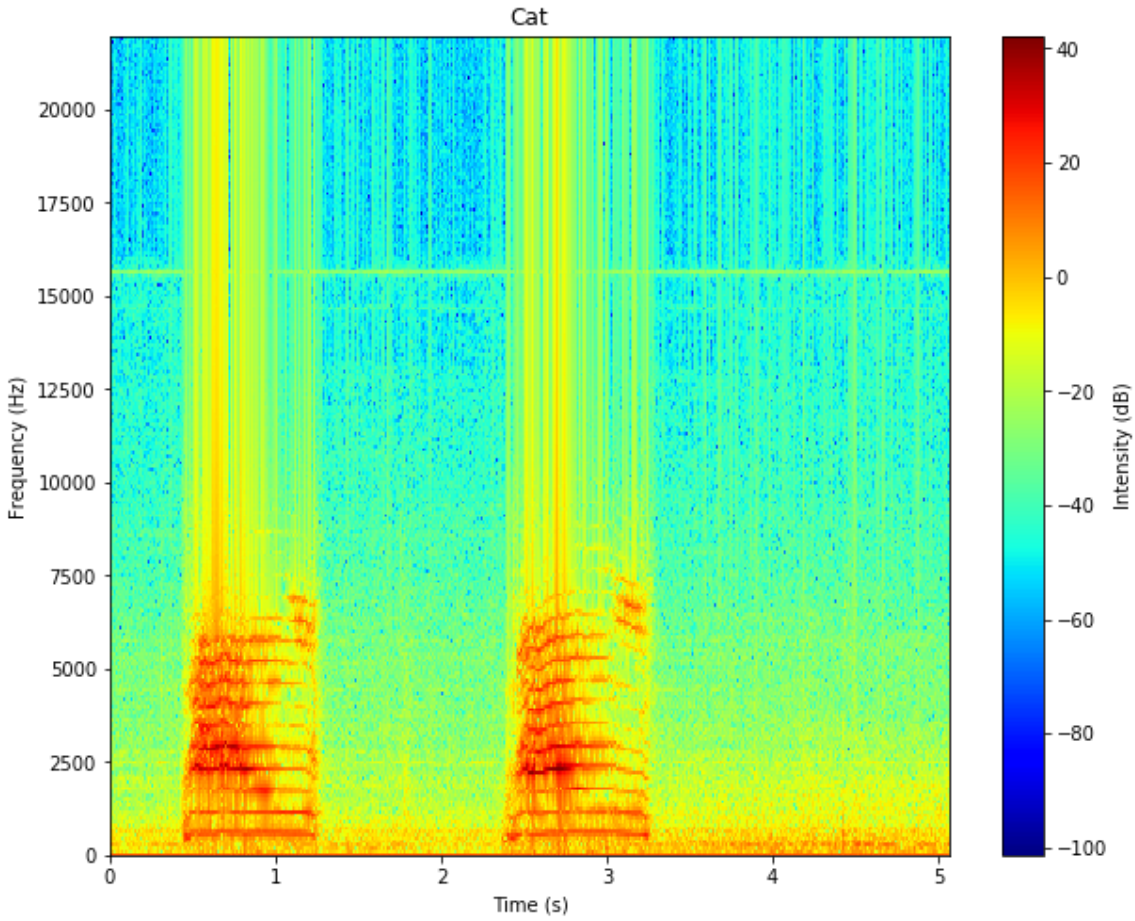


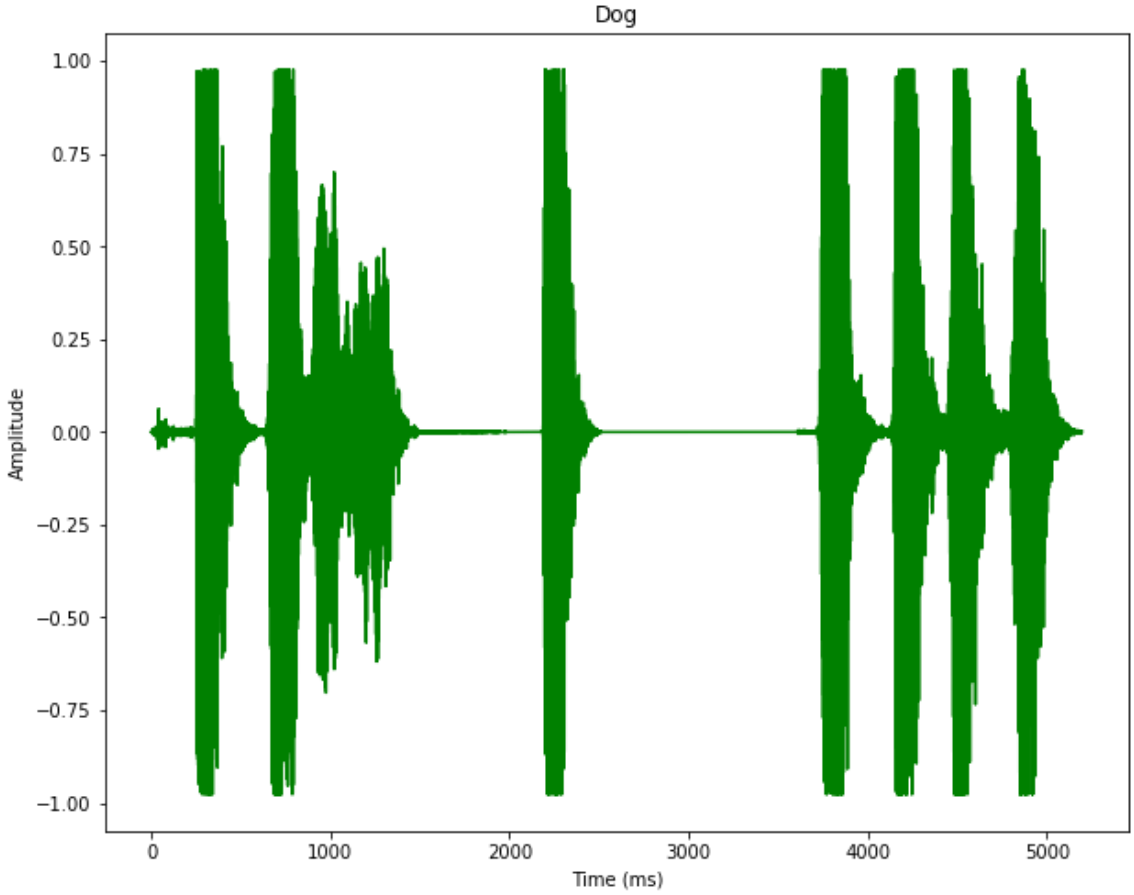
Analyze some signals

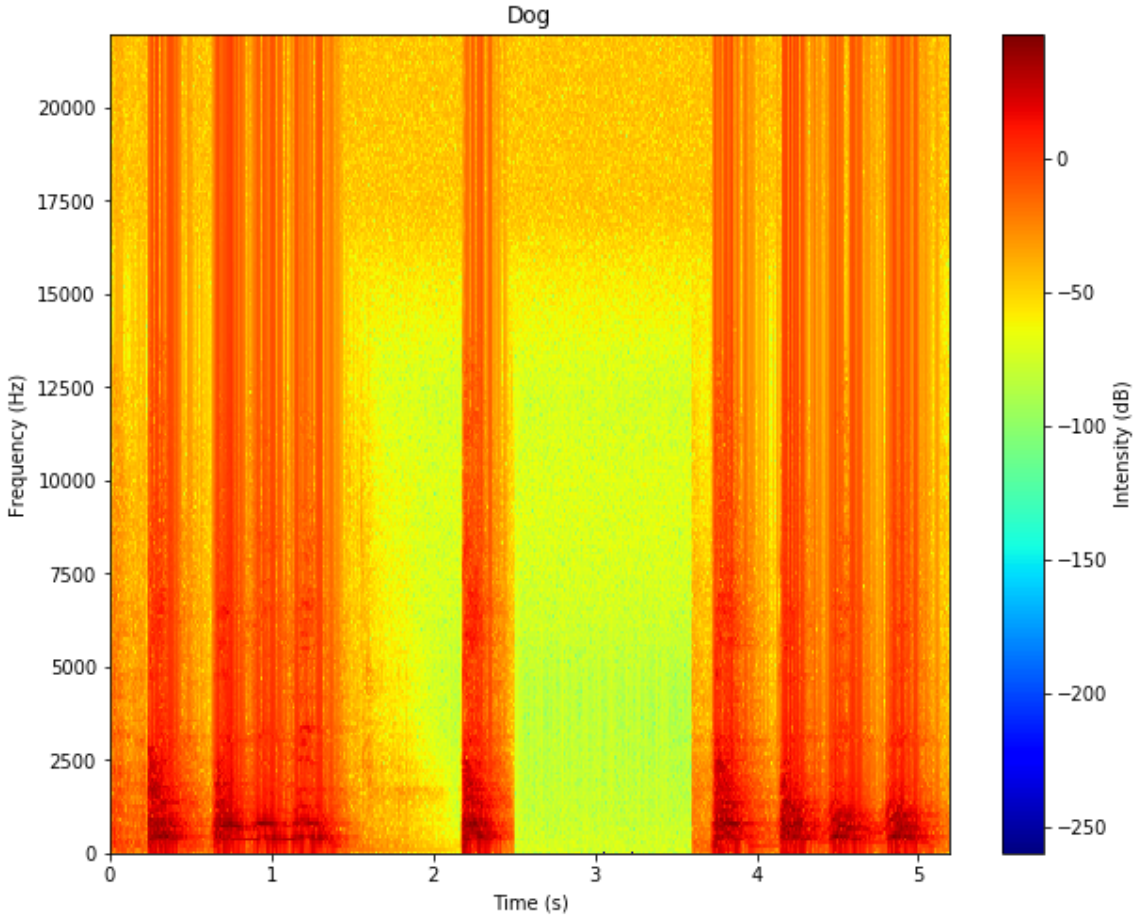
In [20]:

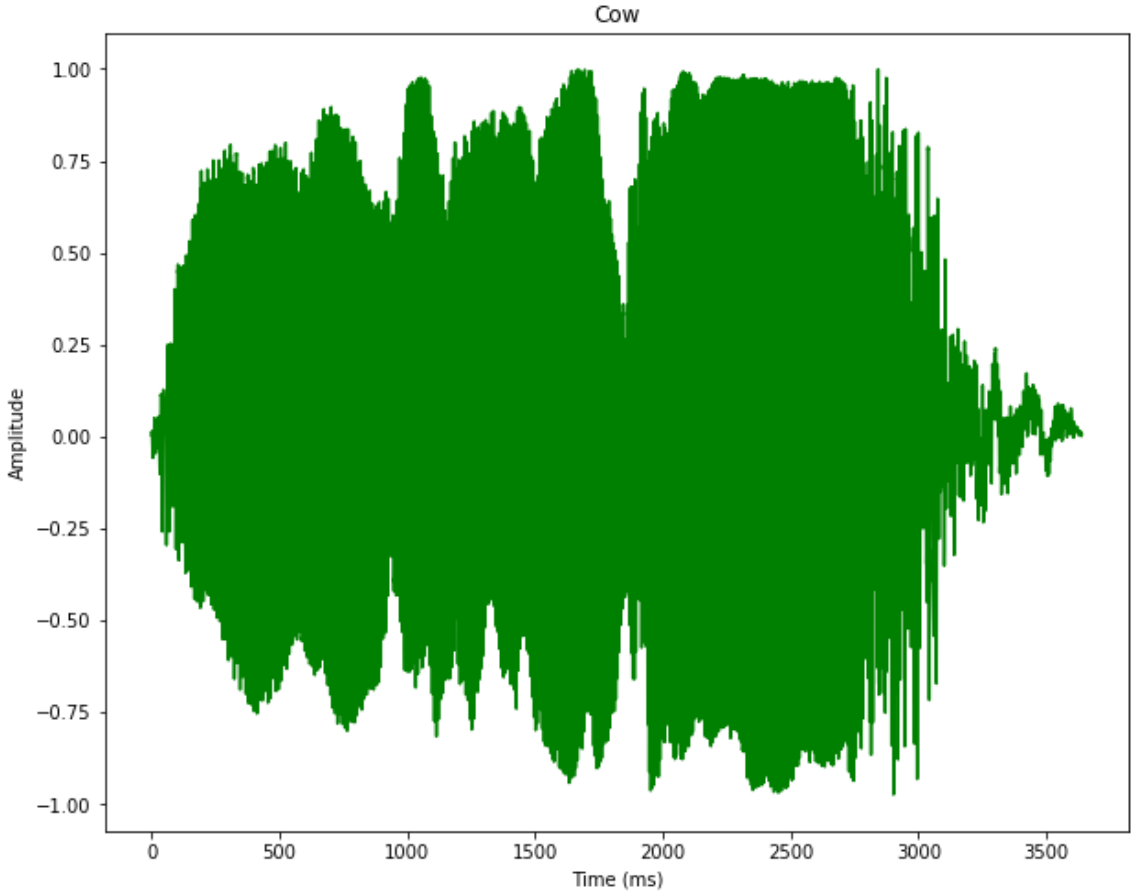
```
def analyze_sound(filename, name):
    signal = Sound(filename)
    signal.plot_sound(name)
    x = signal.data
    Fs = signal.sampling_rate
    duration = signal.duration

    S = calculate_spectogram(x, window_size=window_size, stride=stride, dft_func
tion=np.fft.fft)
    plot_spectogram(S, stride, window_size, duration=duration, Fs=Fs, name=name)

window_size = 500
stride = 100

# Cat sound
cat_sound_filename = 'sound_samples/cat.wav'
analyze_sound(cat_sound_filename, "Cat")

# Dog sound
cow_sound_filename = 'sound_samples/dog.wav'
analyze_sound(cow_sound_filename, "Dog")

# Cow sound
cow_sound_filename = 'sound_samples/cow.wav'
analyze_sound(cow_sound_filename, "Cow")
```

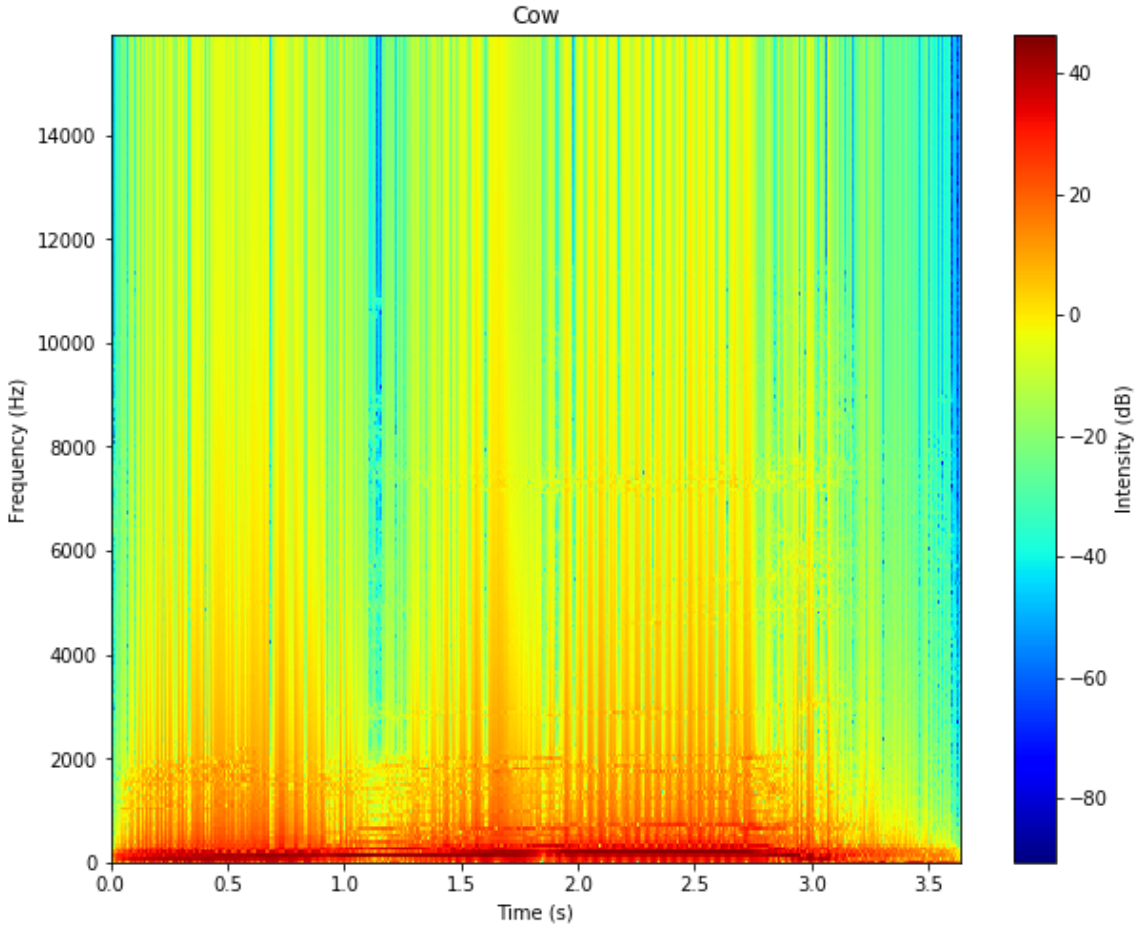

Cat
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
Amplitude
0
1000
2000
3000
4000
5000
Time (ms)


Cat
Frequency (Hz)
20000
17500
15000
12500
10000
7500
5000
2500
0
0
1
2
3
4
5
Time (s)
40
20
0
−20
−40
−60
−80
−100
Intensity (dB)


Dog
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Amplitude
0
1000
2000
3000
4000
5000
Time (ms)


Dog
Frequency (Hz)
20000
17500
15000
12500
10000
7500
5000
2500
0
0
1
2
3
4
5
Time (s)
0
−50
−100
−150
−200
−250
Intensity (dB)

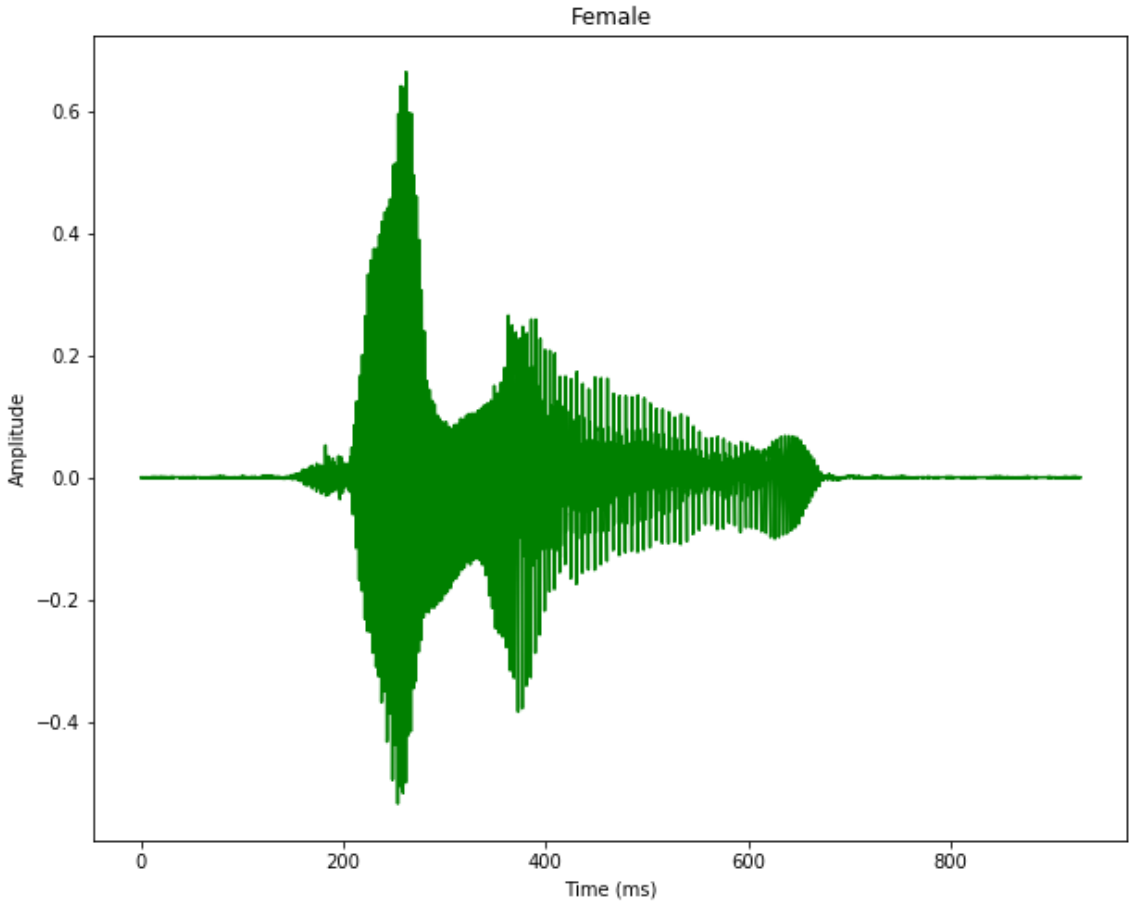

Cow
Amplitude
Time (ms)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
500
1000
1500
2000
2500
3000
3500

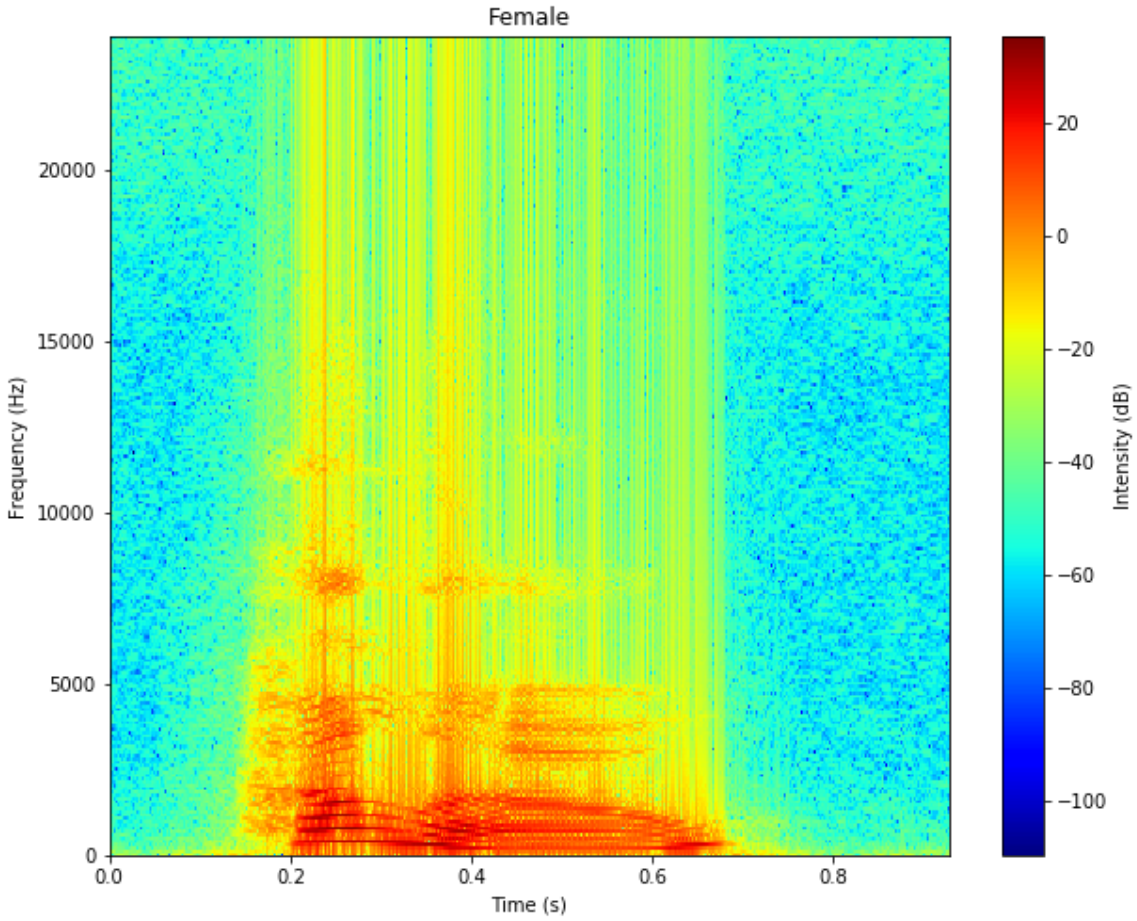


Let's look at a human sound sample

In [21]:

```
# Female hello sound
female_sound_filename = 'sound_samples/hello_female.wav'
analyze_sound(female_sound_filename, "Female")
```


Female
Amplitude
0.6
0.4
0.2
0.0
−0.2
−0.4
0
200
400
600
800
Time (ms)


Female
Frequency (Hz)
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
Time (s)
Intensity (dB)
20
0
−20
−40
−60
−80
−100

Task 3: Discussion

1) What do you see in the time domain for the sample sounds that you have analyzed above?

It shows how the sample sounds change over time. Amplitude of a signal can be examined because this is the only information that is shown in time domain.

In the time domain of cat sound, we can observe that at the time cat meows amplitude of the signal gets higher. In the given sound example cat meows twice so amplitude of the signal changes twice exactly when cat meows. In other time intervals, the amplitude equals to zero.

In the dog sound example, when the dog barks amplitude of the signal gets higher. Dog barks multiple times in this given sound example so the amplitude goes high in these time intervals. In other time intervals, the amplitude is zero.

In the cow sound, amplitude of the signal is almost non-zero for the given time interval because the cow moos during the whole recording.

In the female sound, amplitude is non-zero when the women is speaking. Amplitude of the signal changes because the loudness of the women changes during the recording.

2) What do you see in the frequency domain for the sample sounds that you have analyzed above?

Frequency-domain analyzes the sound over a range of frequencies. It shows how much of the sound lies within each given frequency band. When we analyze the frequency domain graphs for the sample sounds we obtain the following results.

In low frequency, when there is sound like the cat meows or the dog barks the intensity is high for that time interval. However, when there is no sound intensity is low in low frequency. Also in high frequency, when there is sound the intensity is high, when there is not intensity is low.

If we compare the intensities for high and low frequencies, at a certain time we can see that intensity gets lower when we go to high frequency from low frequency. For example, in frequency domain for cat sound at time=1s when the frequency is between 0-2500 Hz, the intensity is around 20-40 dB. However, again at time=1s when the frequency is between 17500-20000 Hz, the intensity is around (-20)-0 dB.

In [ ]: