Lab 4: Digital Signal Processing
Date Posted: October 9, 2023

1 Objectives

- Understanding signals in the time and frequency domain
- Understanding sampling
- Plotting in Matplotlib
- Using FFT

2 Resources

- Numpy Fourier Transform: `https://docs.scipy.org/doc/numpy/reference/routines.fft.html`
- Scipy Documentation: `https://docs.scipy.org/doc/`
- Matplotlib Documentation: `https://matplotlib.org/`

3 Libraries

Before you get started, you might find it helpful to include a few more libraries. In particular, you will want to download numpy, scipy, and matplotlib.

- `https://numpy.org/install/`
- `https://scipy.org/install/`
- `https://matplotlib.org/stable/users/installing/index.html`

4 Creating a sine wave

In lab today you're going to create a sine wave and save it to a wav file. But first, some definitions.

Frequency: The frequency is the number of times a sine wave repeats per second. If none stated, then in this lab we will be using 1 kHz.

Sampling rate: Real world signals are analog, while computers are digital. To convert an analog signal to a digital one, you need an analog to digital converter. The sampling rate is the number of times per second that the converter takes a sample of the analog signal. Most professional audio equipment uses a value of 48000.

Sine Wave Formula: Our trusty sine/cosine formula is

$$y(t) = A * sin(2 * \pi * f * t) \tag{1}$$

where $y(t)$ is the y-axis sample we want to calculate for x-axis sample t. A is the amplitude, f is the frequency, and t is our sample. Because we need to convert it to digital, we divide it by the sampling rate.

[Note: In most books, the value for Amplitude is usually 1. But that won't work for us. The sine wave we generate will be in floating point, and while that will be good enough for drawing a graph, it won't work when we write to a file. The reason being that we are dealing with integers. If you look at wave files, they are written as 16 bit short integers. If we write a floating point number, it will not be represented right. To get around this, we have to convert our floating point number to fixed point. One of the ways to do so is to multiply it with a fixed constant. How do we calculate this constant? Well, the maximum value of signed 16 bit number is $32767(2^{15}–1)$.

(Because the left most bit is reserved for the sign, leaving 15 bits. We raise 2 to the power of 15 and then subtract one, as computers count from 0). So we want full scale audio, we'd multiply it with 32767. But I want an audio signal that is half as loud as full scale, so I will use an amplitude of 16000.]

Now take a look at the following code found in sine.py:

```
1  import numpy as np
2  import wave
3  import struct
4  import matplotlib.pyplot as plt
5
6  # frequency is the number of times a wave repeats a second
7  frequency = 1000
8  num_samples = 48000
9
10 # The sampling rate of the analog to digital convert
11 sampling_rate = 48000.0
12 amplitude = 16000
13 file = ”test.wav”
14
15 sine_wave = [np.sin(2 * np.pi * frequency * x/sampling_rate) for x in range(
       num_samples)]
16
17 nframes=num_samples
18 comptype=”NONE”
19 compname=”not compressed”
20 nchannels=1
21 sampwidth=2
22
23 wav_file=wave.open(file, ’w’)
24 wav_file.setparams((nchannels, sampwidth, int(sampling_rate), nframes, comptype,
       compname))
25
26 for s in sine_wave:
27     wav_file.writeframes(struct.pack(’h’, int(s*amplitude)))
```

- Lines 1-3 just set up the variables we've discussed before.
- Line 15 defines the sine wave. The equation is in brackets [], which means that the final answer will be in the form of a list. This line says generate x in the range of 0 to num_samples, and for each of those x values, generate a value that is the sine of that.
- Lines 17-21 set the parameters for writing the sine wave to a file. *nframes* is the number of frames or samples. *comptype* and *compname* both signify that the data isn't compressed. *nchannels* says the number of channel is 1. *sampwidth* is the sample width in bytes. Wave files are usually 16 bits or 2 bytes per sample.
- Lines 23-24 open the file and set the parameters.
- Lines 26-27 write the sine wave sample by using *sample.writeframes*. You can ignore the meaning inside struct.pack for now, but if you're curious, go here for an explainer: `https://www.cameronmacleod.com/blog/reading-wave-python`.

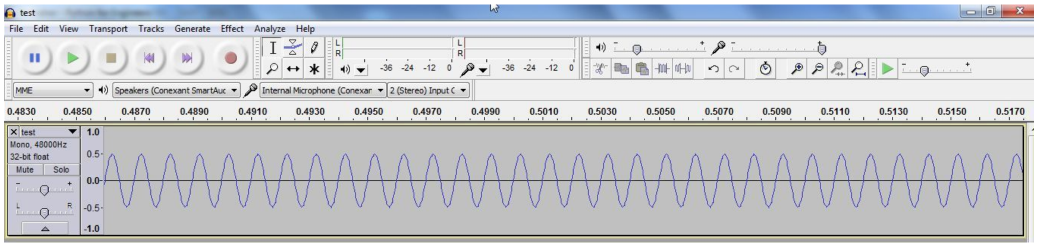
Run the code and you should get a sine wave sample called *test.wav*, packed as 16 bit audio. You can now play it in any audio player you have and hear a short tone. If you want to check the frequency of the tone, you can use a program. Below is a snapshot of the file open in Audacity, an open source audio player. Here is what the file looks like in Audacity.

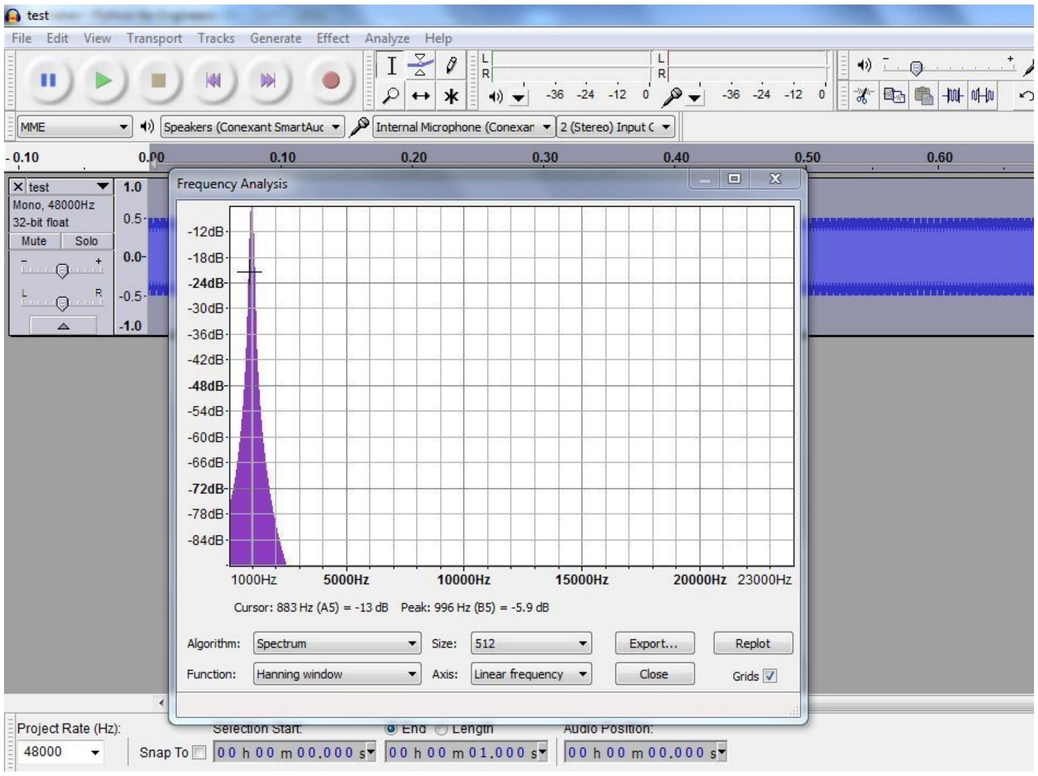
To find the frequency, press Ctrl+A, then Analyze -¿ Plot Spectrum.

The peak is around 1000 Hz, which is exactly how we created the wave file.

5 Getting the frequency

A signal in the time domain will change over time. But, a 1000 Hz audio tone will have the same frequency no matter how long you look at it. The Discrete Fourier Transform (DFT) was used in





Cursor: 883 Hz (A5) = -13 dB Peak: 996 Hz (B5) = -5.9 dB

Algorithm: Spectrum Size: 512
Function: Hanning window Axis: Linear frequency
Grids ✓

Project Rate (Hz): 48000

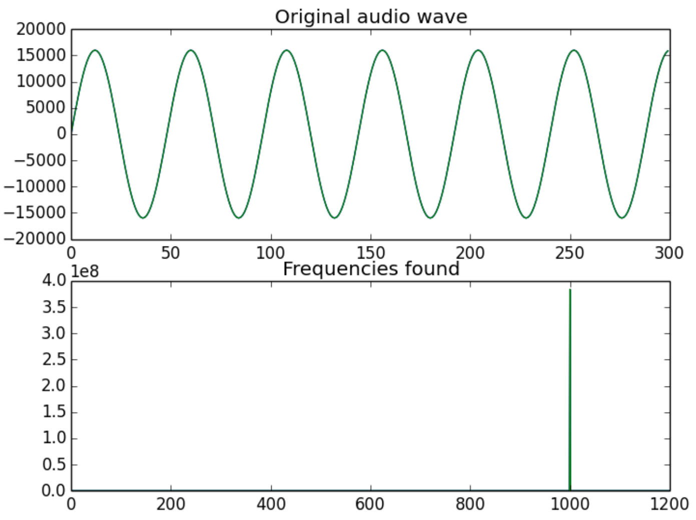
the 70s to convert signals in the time domain to the frequency domain. This took a loooong time to do, so the Fast Fourier Transform (FFT) was invented, and is what is normally used nowadays. You take a signal, run the FFT on it, and get the frequency of the signal back.

Now let's take a look at how to do this. In the function get_freq(), the code looks like this:

```python
import numpy as np
import wave
import struct
import matplotlib.pyplot as plt

frame_rate = 48000.0
infile = "test.wav"
num_samples = 48000
wav_file = wave.open(infile, 'r')
data = wav_file.readframes(num_samples)
wav_file.close()

data = struct.unpack('{n}h'.format(n=num_samples), data)

data = np.array(data)

data_fft = np.fft.fft(data)

frequencies = np.abs(data_fft)

print("The frequency is {} Hz".format(np.argmax(frequencies)))
```

```
22
23 plt.subplot(2,1,1)
24 plt.plot(data[:300])
25 plt.title("Original audio wave")
26 plt.subplot(2,1,2)
27 plt.plot(frequencies)
28 plt.title("Frequencies found")
29 plt.xlim(0,1200)
30 plt.show()
```

- **Lines 6-11 read the wave file generated in the previous example.**
- **Recall in the previous example we glossed over packing while writing the wave file. We'll do the same and just say that we have to undo the packing in Line 13.**
- **Line 15 converts the data into a numpy array.**
- **Line 17 takes the FFT of the data, creating an array with all the frequencies present in the signal.**
- **The FFT returns an array of complex numbers that don't tell us anything. Yet. Go ahead and test this out by printing out some of the values of the FFT. You'll see an array of complex values. So how do you convert complex numbers to real values? Numpy to the rescue. Line 19 uses the abs() function to take the complex signal and generate the real part of it.**
- **The FFT returns all possible frequencies in the signal. The output of data_fft[1] gives you the frequency part of 1 Hz, while data_fft[1000] gives you the frequency part of 1000 Hz. If the signal does not contain that particular frequency, you're going to see a very small number output (i.e., 10^{-12}). To find the array element with the highest value, you can use argmax, as shown in Line 21.**
- **Lines 23-30 use matplotlib to plot the data. Be sure you go through and understand how plots and subplots work.**



6 Homework Questions

6.1 Inserting Noise

Now it's your turn to play with noise! In this problem, you'll generate a sine wave, add noise to it, and then filter out the noise. In the function noisy() in Lab 4.py, do the following:

1. **Generate one sine wave that contains a main frequency of 1000 Hz and noise at a frequency of 50 Hz.**
2. **In one plot, graph the sine wave containing only the 1000 Hz, a sine wave with only 50 Hz, and then the combined sine wave containing both.**
3. **Find and graph the frequencies contained in this combined wave.**
4. **Filter the signal to remove the noise. You don't need to know anything about filtering aside from the notion that you're going to remove values of the frequency that you know do not belong in the original signal. Plot the filtered frequency.**
5. **Take the inverse FFT to find what the original signal should look like without noise. In one plot, graph the original sine wave of 1000 Hz, the noisy combined wave, and then sine wave after it has been filtered.**

6.2 Pitch

Download the file "trumpet.wav."

1. **Use the existing code from the function pitch() to read the wave file, and then plot the entire signal.**
2. **Select a segment of the signal with a somewhat constant pitch and plot this.**
3. **Plot the spectrum for this segment of the signal**
4. **Output the three highest peaks (and their frequencies).**
5. **Filter out the highest frequencies**
6. **Plot the filtered signal (in time domain)**
7. **Save your segment of the trumpet as segment.wav and the filtered one as segment_filter.wav**

6.3 Mixing Signals

Create a mixed signal composed of at least three signals of different frequencies. Evaluate the signal to get a wave and listen to it. Compute its spectrum and plot it. What happens if you add frequency components that are not multiples of the fundamental frequency? Implement your code in the function mix_signals() in Lab4.py.

6.4 Stretch

Write a function in stretch() in Lab4.py that takes a wave and a stretch factor and speeds up or slows down the wave. Output the modified wave into a new wav file.

6.5 Sampling

If you sample a signal at too low a framerate, frequencies above the folding frequency (aka the sampling rate $f_s/2$) get aliased. Once that happens, it is no longer possible to filter out these components, because they are indistinguishable from lower frequencies.

It is a good idea to filter out these frequencies before sampling; a low-pass filter used for this purpose is called an anti-aliasing filter. Apply a low-pass filter to the file drum.wav before sampling, and then apply the low-pass filter again to remove spectral copies introduced by sampling. The result should be identical to the filtered signal.

Well, let's simulate a simple signal that consists of the superposition of two cosine functions. We will simulate this signal at high sampling rate (as if it was continuous), and see what happens when we downsample it.

1. **Begin by plotting the signal and its frequency spectrum in the function.**
2. **Next sub-sample the signal to create a new signal. If the original sampling rate is 1000, name the new sampling rate F. If, for example, F were 500, then the sampled signal would take every other point from the original signal.**
3. **Plot the spectrum for this new sub-sampled signal, as well as the reconstructed signal.**
4. **At what values of F does the signal start degrading? Why?**

7 Submitting Homework to ELMS

List all assumptions, and use comments to notate your code whenever possible. Clarity and style will matter. Your TF should be able to read your code and understand everything. You will be submitting one python file containing your code/functions. You will submit a pdf document containing your answers for 6.2 and 6.3, and 6.5.

1. **directoryID_lab4.py**
2. **directoryID_lab4_writeup.pdf**