Lab 9

Fourier I: The Discrete Fourier Transform

Lab Objective: *The analysis of periodic functions has many applications in pure and applied mathematics, especially in settings dealing with sound waves. The Fourier transform provides a way to analyze such periodic functions. In this lab, we implement the discrete Fourier transform and explore digital audio signals.*

Sound Waves

Sound is the way that we perceive vibrations in matter. These vibrations travel in waves. Sound waves have two important characteristics that determine what we hear, or whether or not we can hear it. *Frequency* is a measurement of the number of occurrences in a certain time, and determines the pitch of the sound. Only certain frequencies are perceptible to the human ear. The second characteristic is *intensity* or *amplitude*, and determines the volume of the sound. Sounds waves correspond physically to continuous functions, but computers can approximate sound waves using discrete measurements. Indeed, discrete measurements can be made indistinguishable to the human ear from a truly continuous wave. Usually, sound waves are of a sinusoidal nature (with some form of decay), and the frequency is related to the wavelength, and the intensity to the wave amplitude.

Digital Audio Signals

Digital Audio Signals are how computers can approximate sound waves, and have two key components that relate to the frequency and amplitude of sound waves: samples, and sampling rate. A sample is a measurement of the amplitude of a sound wave at a specific instant in time. The sampling rate corresponds to the sound frequency.

If we know at what rate a set of samples were taken, then we can reconstruct the wave exactly as it was recorded. If we don't know the sampling rate, then our frequencies will be unknown. In most applications, this sample rate will be measured in the number of samples taken per second, Hertz (Hz). The standard rate for high quality audio is 44100 equally spaced samples per second, or 44.1 kHz.

Problem 1. Write a class called `Signal` for storing digital audio signals. The constructor should accept a sample rate (an integer) and an array of samples (a NumPy array). Store these inputs as attributes.

Write a method called `plot()` that generates the graph of the sound wave. Use the sample rate to label the x-axis in terms of seconds. See Figure 9.1 for an example.

Wave File Format

One of the most common audio file formats across operating systems is the *wave* format, also called `wav` after its file extension. It is a lightweight, common standard that is in wide use. SciPy has built-in tools to read and create `wav` files. To read in a `wav` file, we can use the `read()` function that returns the file's sample rate and samples. See Figure 9.1.

```
# Read from the sound file.
>>> from scipy.io import wavfile
>>> rate, wave = wavfile.read('tada.wav')

# To visualize the data, use the Signal class's plot function.
>>> sig = Signal(rate, wave)
>>> sig.plot()
```

Writing a signal to a file is also simple. We use `wavfile.write()`, specifing the name of the new file, the sample rate, and the array of samples.

```
# Write a random signal sampled at a rate of 44100 Hz to my_sound.wav.
>>> wave = sp.random.randint(-32767, 32767, 30000)
>>> samplerate = 44100
>>> wavfile.write('my_sound.wav', samplerate, wave)
```

Scaling

The `wavfile.write()` function expects an array of 16 bit integers for the samples (whole numbers between -32767 and 32767). Therefore, waves may need to be scaled and converted to integers before being written to a file.

```
# Generate random samples between -0.5 and 0.5.
>>> samples = sp.random.random(30000)-.5
# Scale the wave so that the samples are between -32767 and 32767.
>>> samples *= 32767*2
# Cast the samples as 16 bit integers.
>>> samples = sp.int16(samples)
```

The scaling technique in the above example works, but only because we knew beforehand that the values were in the interval $[-\frac{1}{2}, \frac{1}{2}]$. If the entries of a wave are not scaled properly, the operating system may not know how to play the file.

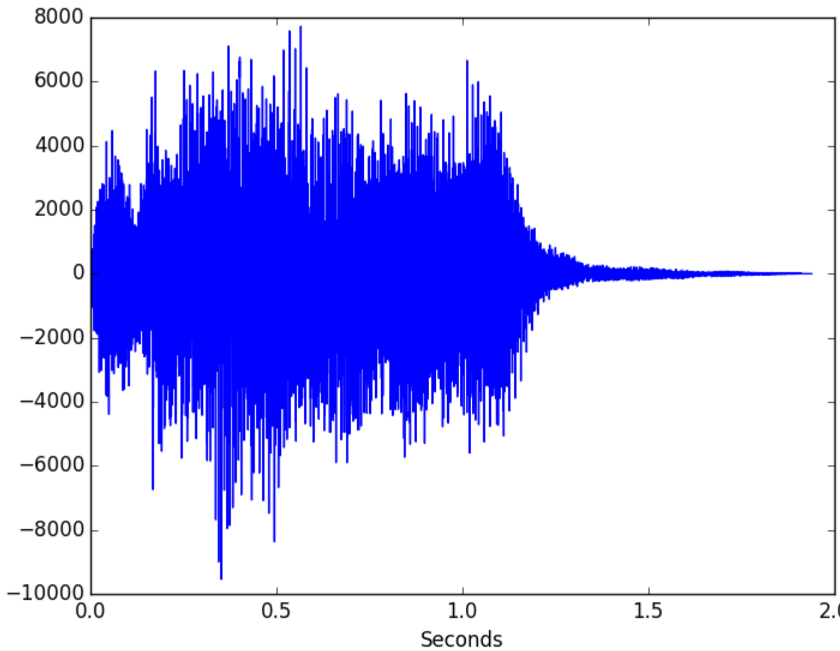

Figure 9.1: The soundwave of `tada.wav`.

Problem 2. Add a method to the `Signal` class called `export()` that accepts a file name and generates a `.wav` file from the sample rate and the array of samples. Scale the array of samples appropriately before writing to the output file. Ensure that your scaling technique is valid for arbitrary arrays of samples. Note that some arrays will not need to be scaled.

Creating Sounds in Python

In order to generate a sound in python, we need to sample the corresponding sinusoidal wave and then save it as an audio file. For example, suppose that we want to generate a sound with a frequency of 500 Hertz for 10 seconds.

```
>>> samplerate = 44100
>>> frequency = 500
>>> length = 10         # Length in seconds of the desired sound.
```

Recall the the function $\sin(x)$ has a period of 2π. To create sounds, however, we want the period of our wave to be 1, corresponding to 1 second. Thus, we will sample from the function

$$\sin(2\pi x f)$$

where f is our desired frequency.

```
# The lambda keyword is a shortcut for creating a one-line function.
>>> wave_function = lambda x: sp.sin(2*sp.pi*x*frequency)
```

In the following code, we generate a signal using three steps: first, we find the correct step size given the sample rate. Next, we generate the points at which we wish to sample the wave. Finally, we sample the wave by passing the sample points to `wave_function`. Then we can use our `Signal` class to plot the soundwave or write it to a file.

```
# Calculate the step size, the sample points, and the sample values.
>>> stepsize = 1./samplerate
>>> sample_points = sp.arange(0, length, stepsize)
>>> samples = wave_function(sample_points)

# Use the Signal class to write the sound to a file.
>>> sinewave = Signal(samplerate, samples)
>>> sinewave.export("sine.wav")
```

The `export()` method should take care of scaling and casting the entries as 16-bit integers.

Problem 3. The 'A' note occurs at a frequency of 440 Hertz. Generate the sine wave that corresponds to an 'A' note being played for 5 seconds.

Once you have successfully generated the 'A' note, experiment with different frequencies to generate different notes. The following table shows some frequencies that correspond to common notes. Octaves of these notes are obtained by doubling or halving these frequencies.

Note	Frequency
A	440
B	493.88
C	523.25
D	587.33
E	659.25
F	698.46
G	783.99
A	880

Implement a function outside of the `Signal` class that accepts a frequency and a duration and returns an instance of the `Signal` class corresponding to the desired soundwave. Sample at a rate of 44100 samples per second to create these sounds.

Discrete Fourier Transform

Some Technicalities

Under the right conditions, a continuous periodic function may be represented as a sum of sine waves:

$$f(x) = \sum_{k=-\infty}^{\infty} c_k \sin kx$$

where the constants c_k are called the *Fourier coefficients*.

Such a transform also exists for discrete periodic functions. Whereas the frequencies present in the continuous case are multiples of a sine wave with a period of 1, the discrete case is somewhat different. The Fourier coefficients in the discrete case represent the amplitudes of sine waves whose periods are multiples of a "fundamental frequency." The fundamental frequency is a sine wave with a period length equal to the amount of time of the signal.

The k^{th} coefficient of a signal $\{x_0, .., x_{N-1}\}$ is calculated with the following formula:

$$c_k = \sum_{n=0}^{N-1} x_n e^{\frac{2\pi ikn}{N}} \tag{9.1}$$

where i is the square root of -1. This process is done for each k from 0 to $N-1$. Thus there are just as many Fourier coefficients as samples from the orginal signal.

Problem 4. Write a function that accepts a NumPy array and computes the discrete Fourier transform of the array using Equation 9.1. Return the array of calculated coefficients.

SciPy has several methods for calculating the DFT of an array. Use `scipy.fft()` or `scipy.fftpack.fft()` to check your implementation. The naive method is significantly slower than SciPy's implementation, so test your function only on small arrays. When you have your method working, try to optimize it so that you can calculate each coefficient c_k in just one line of code.

Plotting the DFT

The graph of the Fourier transform of a sound file is useful in a variety of applications. While the graph of the original signal gives information about the amplitude of a soundwave at certain points, the graph of the discrete Fourier transform shows which frequencies are present in the signal. Often, this information is of greater importance than how the wave changes in time. Frequencies present in the signal have non-zero coefficients. The magnitude of these coefficients corresponds to how influential the frequency is in the signal. For example, the sounds that we generated

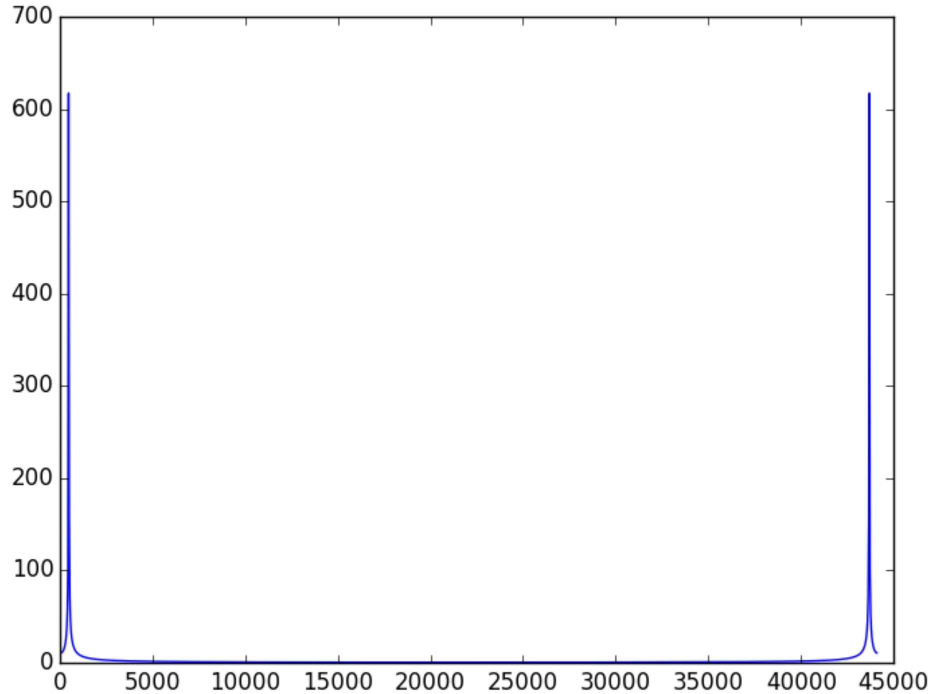

Figure 9.2: The magnitude of the coefficients of the discrete Fourier transform of an 'A' note. Notice that there are two spikes in the graph, the first around 440 on the x-axis. This second spike is due to symmetries inherent in the DFT. For our purposes we will mostly be concerned with the left side of the DFT plot.

in the previous section contained only one frequency. If we created an 'A' note at 440 Hz, then the graph of the DFT would appear as in Figure 9.2.

On the other hand, the DFT of a more complicated sound wave will have many frequencies present. Some of these frequencies correspond to the different tones present in the signal. See Figure 9.3 for an example.

Fixing the x-axis

If we take the DFT of a signal and then plot it without any other considerations, the x-axis will correspond to the index of the coefficients in the DFT and not their frequencies. In a previous section, we mention that the "fundamental frequency" for the DFT corresponds to a sine wave whose period is the same as the length of the signal. Thus, if unchanged, the x-axis gives us the number of times a particular sine wave cycles throughout the whole signal. If we want to label the x-axis with the frequencies measured in Hertz, or cycles per second, we will need to convert the

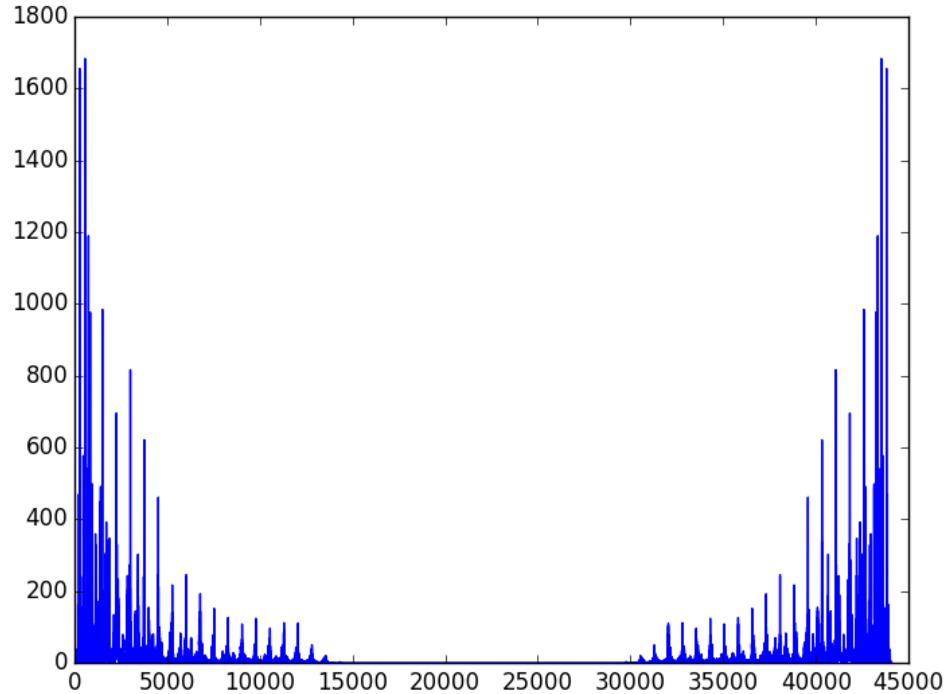

Figure 9.3: The discrete Fourier transform of `tada.wav`. Each spike in the graph corresponds to a frequency that is present in the signal.

units. Fortunately, the bitrate is measured in samples per second. Therfore, if we divide the frequency (given by the index) by the number of samples, and multiply by the sample rate, we end up with cycles per second, or Hertz.

$$\frac{\text{cycles}}{\text{samples}} \times \frac{\text{samples}}{\text{second}} = \frac{\text{cycles}}{\text{second}}$$

```
# Calculate the DFT and the x-values that correspond to the coefficients. Then
# convert the x-values so that they measure frequencies in Hertz.
>>> dft = sp.fft(signal)
>>> x_vals = sp.arange(1,len(dft)+1, 1)*1. # Make them floats

# x_vals now corresponds to frequencies measured in cycles per signal length.
>>> x_vals = x_vals/len(signal)
>>> x_vals = x_vals*rate
```

Problem 5. Update the `plot()` method in the `Signal` class so that it generates a single plot with two subplots: the original soundwave, and the magnitude of the coefficients of the DFT (as in Figure 9.3). Use one of SciPy's

FFT implementations to calculate the DFT.

Problem 6. A chord is a conjunction of several notes played together. We can create a chord in Python by adding several sound waves together. For example, to create a (minor) chord with 'A', 'C', and 'E' notes, we generate the sound waves for each, as in the prior problem, and then add them together.

Create several chords and observe the plot of their DFT. There should be as many spikes as there are notes in the plot. Then create a sound that changes over time.

(Hints: you may consider implementing the `__add__()` magic method for the `Signal` class. NumPy's `np.hstack()` and `np.vstack()` may also be helpful.)