# 1

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 introduce how to work with digital audio signals in Python, implement the discrete Fourier transform, and use the Fourier transform to detect the frequencies present in a given sound wave. We strongly recommend completing the exercises in a Jupyter Notebook.*

Digital Audio Signals

Sound waves have two important characteristics: *frequency*, which determines the pitch of the sound, and *intensity* or *amplitude*, which determines the volume of the sound. Computers use *digital audio signals* to approximate sound waves. These signals have two key components: *sample rate*, which relates to the frequency of sound waves, and *samples*, which measure the amplitude of sound waves at a specific instant in time.

To see why the sample rate is necessary, consider an array with samples from a sound wave. The sound wave can be arbitrarily stretched or compressed to make a variety of sounds. If compressed, the sound becomes shorter and has a higher pitch. Similarly, the same set of samples with a lower sample rate becomes stretched and has a lower pitch.
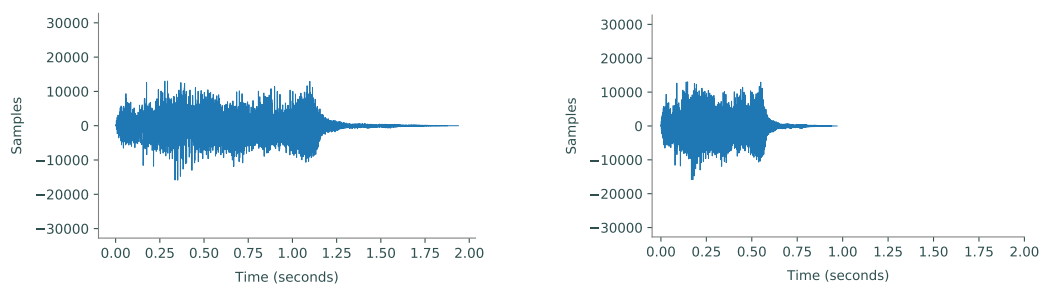
(a) The plot of `tada.wav`.   (b) Compressed plot of `tada.wav`.

Figure 1.1: Plots of the same set of samples from a sound wave with varying sample rates. The plot on the left is the plot of the samples with the original sample rate. The sample rate of the plot on the right has been doubled, resulting in a compression of the actual sound when played back.

1

Given the rate at which a set of samples is taken, the wave can be reconstructed exactly as it was recorded. In most applications, this sample rate is measured in *Hertz* (Hz), the number of samples taken per second. The standard rate for high quality audio is 44100 equally spaced samples per second, or 44.1 kHz.

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. SciPy has built-in tools to read and create `wav` files. To read a `wav` file, use `scipy.io.wavfile.read()`. This function returns the signal's sample rate and its samples.

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

Sound waves can be visualized by plotting time against the amplitude of the sound, as in Figure 1.1. The amplitude of the sound at a given time is just the value of the sample at that time. Since the sample rate is given in samples per second, the length of the sound wave in seconds is found by dividing the number of samples by the sample rate:

$$\frac{\text{num samples}}{\text{sample rate}} = \frac{\text{num samples}}{\text{num samples/second}} = \text{second.} \tag{1.1}$$

Problem 1. Write a `SoundWave` class for storing digital audio signals.

1. The constructor should accept an integer sample rate and an array of samples. Store each input as an attribute.

2. Write a method that plots the stored sound wave. Use (1.1) to correctly label the x-axis in terms of seconds, and set the y-axis limits to $[-32768, 32767]$ (the reason for this is discussed in the next section).

Use SciPy to read `tada.wav`, then instantiate a corresponding `SoundWave` object and display its plot. Compare your plot to Figure 1.1a.

Scaling

To write to a `wav` file, use `scipy.io.wavfile.write()`. This function accepts the name of the file to write to, the sample rate, and the array of samples as parameters.

```
>>> import numpy as np

# Write a 2-second random sound wave sampled at a rate of 44100 Hz.
>>> samples = np.random.randint(-32768, 32767, 88200, dtype=np.int16)
>>> wavfile.write("white_noise.wav", 44100, samples)
```

For `scipy.io.wavfile.write()` to correctly create a `wav` file, the samples must be one of four numerical datatypes: 32-bit floating point (`np.float32`), 32-bit integers (`np.int32`), 16-bit integers

(`np.int16`), or 8-bit unsigned integers (`np.uint8`). If samples of a different type are passed into the function, it may still write a file, but the sound will likely be distorted in some way. In this lab, we only work with 16-bit integer samples, unless otherwise specified.

A 16-bit integer is an integer between -32768 and 32767, inclusive. If the elements of an array of samples are not all within this range, the samples must be scaled before writing to a file: multiply the samples by 32767 (the largest number in the 16-bit range) and divide by the largest sample magnitude. This ensures the most accurate representation of the sound and sets it to full volume.

$$\texttt{np.int16}\left(\left(\frac{\text{original samples}}{\max(|\text{original samples}|)}\right) \times 32767\right) = \text{scaled samples} \tag{1.2}$$

Because 16-bit integers can only store numbers within a certain range, it is important to multiply the original samples by the largest number in the 16-bit range *after* dividing by the largest sample magnitude. Otherwise, the results of the multiplication may be outside the range of integers that can be represented, causing overflow errors. Also, samples may sometimes contain complex values, especially after some processing. Make sure to scale and export only the real part (use the `real` attribute of the array).

NOTE

The IPython API includes a tool for embedding sounds in a Jupyter Notebook. The function `IPython.display.Audio()` accepts either a file name or a sample rate (`rate`) and an array of samples (`data`); calling the function generates an interactive music player in the Notebook.

```
In [1]: import IPython
        from scipy.io import wavfile

        # Embed tada.wav straight from the file.
        IPython.display.Audio(filename="tada.wav")

        # Alternatively, embed tada.wav using the raw data.
        # rate, samples = wavfile.read("tada.wav")
        # IPython.display.Audio(rate=rate, data=samples)
```

Out[1]: ▶ 0:00 ●━━━━━━━━━━ -0:01

ACHTUNG!

Turn the volume down before listening to any of the sounds in this lab.

Problem 2. Add a method to the `SoundWave` class that accepts a file name and a boolean `force`. Write to the specified file using the stored sample rate and the array of samples. If the array of samples does not have `np.int16` as its data type, or if `force` is `True`, scale the samples as in (1.2) before writing the file.

Use your method to create two new files that contains the same sound as `tada.wav`: one

> without scaling, and one with scaling (use `force=True`). Use `IPython.display.Audio()` to
> display `tada.wav` and the new files. All three files should sound identical, except the scaled file
> should be louder than the other two.

Generating Sounds

Sinusoidal waves correspond to pure frequencies, like a single note on the piano. Recall that the
function $\sin(x)$ has a period of 2π. To create a specific tone for 1 second, we sample from the
sinusoid with period 1,

$$f(x) = \sin(2\pi x k),$$

where k is the desired frequency. According to (1.1), generating a sound that lasts for s seconds at
a sample rate r requires rs equally spaced samples in the interval $[0, s]$.

> **Problem 3.** Write a function that accepts floats k and s. Create a `SoundWave` instance con-
> taining a tone with frequency k that lasts for s seconds. Use a sample rate of $r = 44100$.
>
> 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 (Hz) |
> |------|----------------|
> | A | 440 |
> | B | 493.88 |
> | C | 523.25 |
> | D | 587.33 |
> | E | 659.25 |
> | F | 698.46 |
> | G | 783.99 |
> | A | 880 |

Use your function to generate an A tone lasting for 2 seconds.

> **Problem 4.** Digital audio signals can be combined by addition or concatenation. Adding
> samples overlays tones so they play simultaneously; concatenated samples plays one set of
> samples after the other with no overlap.
>
> 1. Implement the `__add__()` magic method for the `SoundWave` class so that if `A` and `B`
>   are `SoundWave` instances, `A + B` creates a new `SoundWave` object whose samples are the
>   element-wise sum of the samples from `A` and `B`. Raise a `ValueError` if the sample arrays
>   from `A` and `B` are not the same length.
>
>   Use your method to generate a three-second A minor chord (A, C, and E together).
>
> 2. Implement the `__rshift__()` magic method[a] for the `SoundWave` class so that if `A` and `B`
>   are `SoundWave` instances, `A >> B` creates a new `SoundWave` object whose samples are the
>   concatenation of the samples from `A`, then the samples from `B`. Raise a `ValueError` if the
>   sample rates from the two objects are not equal.
>   (Hint: `np.concatenate()`, `np.hstack()`, and/or `np.append()` may be useful.)

Use your method to generate the arpeggio A → C → E, where each pitch lasts one second.

Consider using these two methods to produce elementary versions of some simple tunes.

---

[a]The >> operator is a *bitwise shift operator* and is usually reserved for operating on binary numbers.

The Discrete Fourier Transform

As with the chords generated above, all sound waves are sums of varying amounts of different frequencies (pitches). In the case of the discrete samples $\mathbf{f} = \begin{bmatrix} f_0 & f_1 & \cdots & f_{n-1} \end{bmatrix}^\mathsf{T}$ that we have worked with thus far, each f_i gives information about the amplitude of the sound wave at a specific instant in time. However, sometimes it is useful to find out what frequencies are present in the sound wave and in what amount.

We can write the sound wave sample as a sum

$$\mathbf{f} = \sum_{k=0}^{n-1} c_k \mathbf{w}_n^{(k)}, \tag{1.3}$$

where $\{\mathbf{w}_n^{(k)}\}_{k=0}^{n-1}$, called the *discrete Fourier basis*, represents various frequencies. The coefficients c_k represent the amount of each frequency present in the sound wave.

The *discrete Fourier transform (DFT)* is a linear transformation that takes $\mathbf{f}$ and finds the coefficients $\mathbf{c} = \begin{bmatrix} c_0 & c_1 & \cdots & c_{n-1} \end{bmatrix}^\mathsf{T}$ needed to write $\mathbf{f}$ in this frequency basis. Later in the lab, we will convert the index k to a value in Hertz to find out what frequency c_k corresponds to.

Because the sample $\mathbf{f}$ was generated by taking n evenly spaced samples of the sound wave, we generate the basis $\{\mathbf{w}_n^{(k)}\}_{k=0}^{n-1}$ by taking n evenly spaced samples of the frequencies represented by the oscillating functions $\{e^{-2\pi i k t/n}\}_{k=0}^{n-1}$. (Note that $i = \sqrt{-1}$, the imaginary unit, is represented as 1j in Python). This yields

$$\mathbf{w}_n^{(k)} = \begin{bmatrix} \omega_n^0 & \omega_n^{-k} & \cdots & \omega_n^{-(n-1)k} \end{bmatrix}^\mathsf{T}, \tag{1.4}$$

where $\omega_n = e^{2\pi i/n}$.

The DFT is then represented by the change of basis matrix

$$F_n = \frac{1}{n}\begin{bmatrix} \mathbf{w}_n^0 & \mathbf{w}_n^1 & \mathbf{w}_n^2 & \cdots & \mathbf{w}_n^{n-1} \end{bmatrix} = \frac{1}{n}\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \cdots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \cdots & \omega_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \cdots & \omega_n^{-(n-1)^2} \end{bmatrix}, \tag{1.5}$$

and we can take the DFT of f by calculating

$$\mathbf{c} = F_n \mathbf{f}. \tag{1.6}$$

Note that the DFT depends on the number of samples n, since the discrete Fourier basis we use depends on the number of samples taken. The larger n is, the closer the frequencies approximated by the DFT will be to the actual frequencies present in the sound wave.

> **ACHTUNG!**
>
> There are several different conventions for defining the DFT. For example, instead of (1.6), `scipy.fftpack.fft()` uses the formula
>
> $$\mathbf{c} = nF_n\mathbf{f},$$
>
> where F_n is as given (1.5). Denoting this version of the DFT as $\hat{F}_n\mathbf{f} = \hat{\mathbf{c}}$, we have $nF_n = \hat{F}_n$ and $n\mathbf{c} = \hat{\mathbf{c}}$. The conversion is easy, but it is very important to be aware of which convention a particular implementation of the DFT uses.

> **Problem 5.** Write a function that accepts an array $\mathbf{f}$ of samples. Use 1.6 to calculate the coefficients $\mathbf{c}$ of the DFT of $\mathbf{f}$. Include the $1/n$ scaling in front of the sum.
>
>   Test your implementation on small, random arrays against `scipy.fftpack.fft()`, scaling your output $\mathbf{c}$ to match SciPy's output $\hat{\mathbf{c}}$. Once your function is working, try to optimize it so that the entire array of coefficients is calculated in the one line.
> (Hint: Use array broadcasting.)

The Fast Fourier Transform

Calculating the DFT of a vector of n samples using only (1.6) is at least $O(n^2)$, which is incredibly slow for realistic sound waves. Fortunately, due to its inherent symmetry, the DFT can be implemented as a recursive algorithm by separating the computation into even and odd indices. This method of calculating the DFT is called the *fast Fourier transform* (FFT) and runs in $O(n \log n)$ time.

Algorithm 1.1 The fast Fourier transform for arrays with 2^a entries for some $a \in \mathbb{N}$.

1: **procedure** SIMPLE_FFT($\mathbf{f}$, N)
2:      **procedure** SPLIT($\mathbf{g}$)
3:          $n \leftarrow \text{size}(\mathbf{g})$
4:          **if** $n \leq N$ **then**
5:              **return** $nF_n\mathbf{g}$ ▷ Use the function from Problem 5 for small enough $\mathbf{g}$.
6:          **else**
7:              even $\leftarrow$ SPLIT($\mathbf{g}_{::2}$)   ▷ Get the DFT of every other entry of $\mathbf{g}$, starting from 0.
8:              odd $\leftarrow$ SPLIT($\mathbf{g}_{1::2}$)   ▷ Get the DFT of every other entry of $\mathbf{g}$, starting from 1.
9:              $\mathbf{z} \leftarrow \text{zeros}(n)$
10:             **for** $k = 0, 1, \ldots, n-1$ **do**   ▷ Calculate the exponential parts of the sum.
11:                 $z_k \leftarrow e^{-2\pi i k/n}$
12:             $m \leftarrow n \mathbin{//} 2$ ▷ Get the middle index for $\mathbf{z}$ (// is integer division).
13:             **return** [even $+$ $\mathbf{z}_{:m}\odot$odd, even $+$ $\mathbf{z}_{m:}\odot$odd] ▷ Concatenate two arrays of length m.
14:     **return** SPLIT($\mathbf{f}$) $/\,\text{size}(\mathbf{f})$

Note that the base case in lines 4–5 of Algorithm 1.1 results from setting $n = 1$ in (1.6), yielding the single coefficient $c_0 = g_0$. The $\odot$ in line 13 indicates the component-wise product

$$\mathbf{f} \odot \mathbf{g} = \begin{bmatrix} f_0g_0 & f_1g_1 & \cdots & f_{n-1}g_{n-1} \end{bmatrix}^{\mathsf{T}},$$

which is also called the *Hadamard product* of **f** and **g**.

This algorithm performs significantly better than the naïve implementation of the DFT, but the simple version described in Algorithm 1.1 only works if the number of original samples is exactly a power of 2. SciPy's FFT routines avoid this problem by padding the sample array with zeros until the size is a power of 2, then executing the remainder of the algorithm from there. Of course, SciPy also uses various other tricks to further speed up the computation.

Problem 6. Write a function that accepts an array **f** of n samples where n is a power of 2. Use Algorithm 1.1 to calculate the DFT of **f**.
(Hint: eliminate the loop in lines 10–11 with `np.arange()` and array broadcasting, and use `np.concatenate()` or `np.hstack()` for the concatenation in line 13.)

Test your implementation on random arrays against `scipy.fftpack.fft()`, scaling your output **c** to match SciPy's output **ĉ**. Time your function from Problem 5, this function, and SciPy's function on an array with 8192 entries.
(Hint: Use `%time` in Jupyter Notebook to time a single line of code.)

Visualizing the DFT

The graph of the DFT of a sound wave is useful in a variety of applications. While the graph of the sound in the time domain gives information about the amplitude (volume) of a sound wave at a given time, the graph of the DFT shows which frequencies (pitches) are present in the sound wave. Plotting a sound's DFT is referred to as plotting in the *frequency domain*.

As a simple example, the single-tone notes generated by the function in Problem 3 contain only one frequency. For instance, Figure 1.2a graphs the DFT of an A tone. However, this plot shows two frequency spikes, despite there being only one frequency present in the actual sound. This is due to symmetries inherent to the DFT; for frequency detection, the second half of the plot can be ignored as in Figure 1.2b.
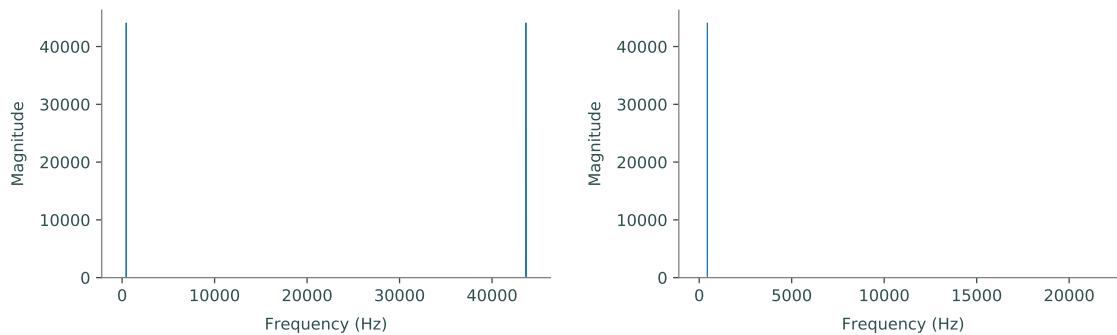
(a) The DFT of an A tone with symmetries.   (b) The DFT of an A tone without symmetries.

Figure 1.2: Plots of the DFT with and without symmetries. Notice that the x-axis of the symmetrical plot on the left goes up to 44100 (the sample rate of the sound wave) while the x-axis of the non-symmetric plot on the right goes up to only 22050 (half the sample rate). Also notice that the spikes occur at 440 Hz and 43660 Hz (which is $44100 - 440$).

The DFT of a more complicated sound wave has many frequencies, each of which corresponds to

a different tone present in the sound wave. The magnitude of the coefficients indicates a frequency's influence in the sound wave; a greater magnitude means that the frequency is more influential.
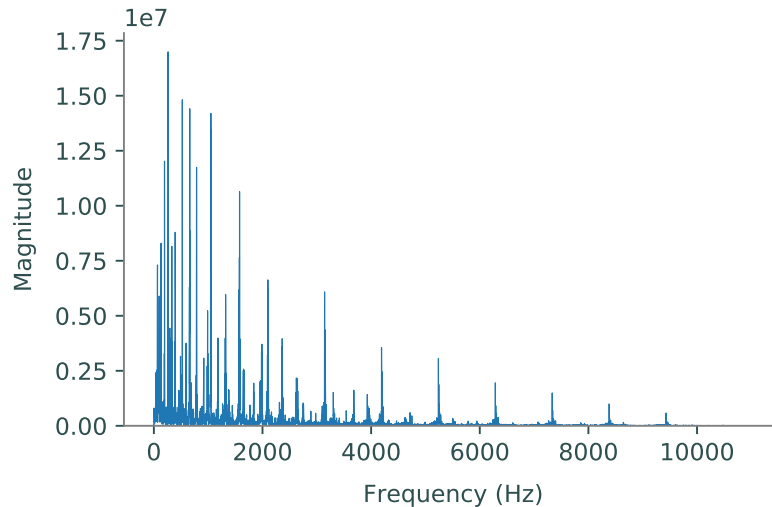

Figure 1.3: The discrete Fourier transform of `tada.wav`. Each spike in the graph corresponds to a frequency present in the sound wave. Since the sample rate of `tada.wav` is 22050 Hz, the plot of its DFT without symmetries only goes up to 11025 Hz, half of its sample rate.

Plotting Frequencies

Since the DFT represents the frequency domain, the x-axis of a plot of the DFT should be in terms of Hertz, which has units $1/s$. In other words, to plot the magnitudes of the Fourier coefficients against the correct frequencies, we must convert the frequency index k of each c_k to Hertz. This can be done by multiplying the index by the sample rate and dividing by the number of samples:

$$\frac{k}{\text{num samples}} \times \frac{\text{num samples}}{\text{second}} = \frac{k}{\text{second}}. \tag{1.7}$$

In other words, $kr/n = v$, where r is the sample rate, n is the number of samples, and v is the resulting frequency.

Problem 7. Modify your `SoundWave` plotting method from Problem 1 so that it accepts a boolean defaulting to `False`. If the boolean is `True`, take the DFT of the stored samples and plot—in a new subplot—the frequencies present on the x-axis and the magnitudes of those frequencies (use `np.abs()` to compute the magnitude) on the y-axis. Only display the first half of the plot (as in Figures 1.2b and 1.2b), and use (1.7) to adjust the x-axis so that it correctly shows the frequencies in Hertz. Use SciPy to calculate the DFT.

Display the DFT plots of the A tone and the A minor chord from Problem 4. Compare your results to Figures 1.2a and 1.4.
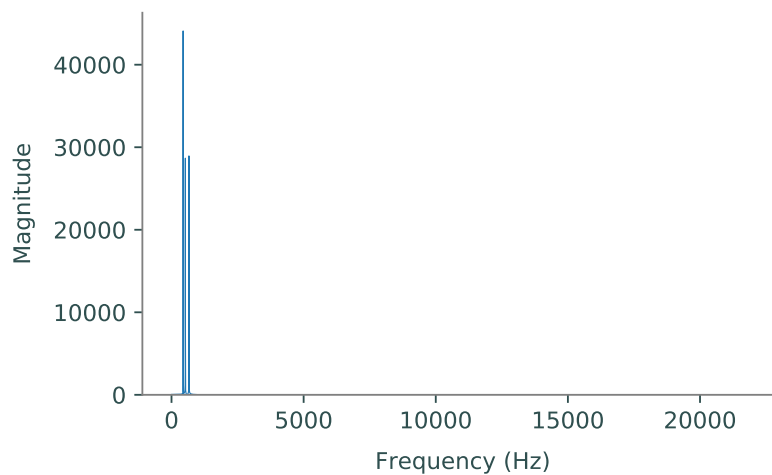
Figure 1.4: The DFT of the A minor chord.

If the frequencies present in a sound are already known before plotting its DFT, the plot may be interesting, but little new information is actually revealed. Thus, the main applications of the DFT involve sounds in which the frequencies present are unknown. One application in particular is sound filtering, which will be explored in greater detail in a subsequent lab. The first step in filtering a sound is determining the frequencies present in that sound by taking its DFT.

Consider the DFT of the A minor chord in Figure 1.4. This graph shows that there are three main frequencies present in the sound. To determine what those frequencies are, find which indices of the array of DFT coefficients have the three largest values, then scale these indices the same way as in (1.7) to translate the indices to frequencies in Hertz.

Problem 8. The file `mystery_chord.wav` contains an unknown chord. Use the DFT and the frequency table in Problem 3 to determine the individual notes that are present in the sound. (Hint: `np.argsort()` may be useful.)