A2-FourierTransform

July 13, 2019

- Ronald Matthew Montehermoso
- A13732474

- COGS118C - Assignment 2

1 This notebook has [45 + 5 bonus] points in total

The number of points for each question is denoted by []. Make sure you've answered all the questions and that the point total add up.

2 Lab 2 - Fourier Analysis & Coding Your Own DFT

In this lab, we will learn about Fourier analysis the hands-on way: through building your own Discrete Fourier transform (DFT). You will employ complex sinusoids and dot products from Lab 0, and apply your DFT to a real signal to measure neural oscillations via the power spectrum. There is a bonus section at the end to explain aliasing in the frequency domain.

Key concepts:

- Cosine and Sine waves
- Dot product, revisited
- Complex exponential, magnitude and phase
- Wave number and frequency
- Power spectrum
- Aliasing

3 Discrete Fourier Transform

It's useful to keep the formula for DFT in mind, as we step through each component for building it in code.

Definition [edit]

The *discrete Fourier transform* transforms a sequence of *N* complex numbers $\{\mathbf{x_n}\} := x_0, x_1, \ldots, x_{N-1}$ into another sequence of complex numbers, $\{\mathbf{X_k}\} := X_0, X_1, \ldots, X_{N-1}$, which is defined by

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N} kn} \qquad \text{(Eq.1)}$$
$$= \sum_{n=0}^{N-1} x_n \cdot [\cos(2\pi kn/N) - i \cdot \sin(2\pi kn/N)],$$

from Wikipedia.

To unpack that scary-looking equation, notice first that the summation expression is performing a dot product between two discrete-time signals: our signal $x(n)$ and the complex exponential in polar form.

The second line simply rewrites the complex exponential using Euler's formula, and because of the linearity of the dot product, we can treat the sine and cosine separately.

k is the wave number, which is analogous to frequency.

** At its core, the DFT turns an array of numbers, our signal in time, into another array of numbers, its "frequency domain representation".**

```
In [1]: # make the imports
        %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
        from scipy import io
```

4 Load the LFP signal

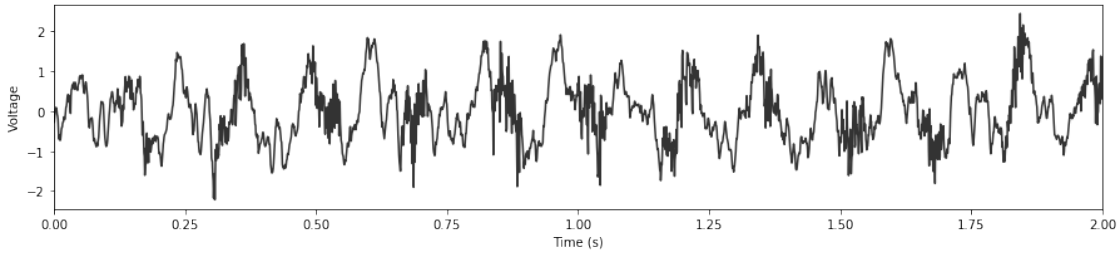
Today, we'll be working with LFP recorded in the rat hippocampus. This dataset comes from an openly accessible neuroscience database. For more information on this particular dataset, see here.

As was with last assignment, we will load the .mat file into python, and unpack the dictionary (I've done all this for you). In this file, LFP values are actually stored as integers (representing ADC levels), so we'll just label the y-axis as voltage without specifying a particular unit (like mV).

```
In [2]: data = io.loadmat('./data/LFP.mat', squeeze_me=True)
        print(data.keys())

        # unpack the variables
        fs = data['fs'] # sampling rate
        print('Sampling rate = %iHz'%fs)

        lfp = data['lfp'][0,:]/1000 # this file contains two channels, we'll only work with th
        lfp_short = lfp[:int(2*fs)] # make a variable that has only the first two seconds of t
```

```
        t_short = np.arange(0, len(lfp_short)/fs, 1/fs) # create the corresponding time vector

        plt.figure(figsize=(15,3))
        plt.plot(t_short,lfp_short, 'k', alpha=0.8)
        plt.xlim([0,2])
        plt.xlabel('Time (s)');plt.ylabel('Voltage');
```

```
dict_keys(['__header__', '__version__', '__globals__', 'lfp', 'fs', 'spike_indices', 'spike_fs
Sampling rate = 1250Hz
```



5 Neural Oscillations

You will notice that the LFP signal has a very prominent rhythmic component, with peaks near, for example, 0.25, 0.37, and 0.5 seconds. Therefore, it has a **periodicity of about 0.125 seconds**, or a **frequency of 8Hz**. This is a very famous brain oscillation, called the theta oscillation.

Fourier, or frequency domain analysis, is particularly useful for characterizing this type of rhythmic brain signal. The method has a long history from electrical and systems engineering, both for describing signals and processes that generate signals (linear time invariant systems). But first, we'll construct the primary ingredient of the Fourier Transform: **sine waves**.

6 [10] Q1: Sine and Cosine Waves

Before we start analyzing the neural data, we have to go over the concept of sine waves. As we alluded to above, the (Discrete) Fourier Transform is a series of dot products, between sines/cosines of different frequencies and our signal. This might be a little confusing, if you are mainly familiar with computing the cosine of an angle, which gives you a single number back. For example, the cosine of 90 degrees (or $\pi/2$) is 0.

Here, we want a sine *wave*, i.e., to compute cosine and sine over time to produce a vector, and you can think of that as computing the cosine of different "angles" at the same time points as your LFP signal. But what are these angles?

It's helpful to picture a clock in your head. A clock's seconds hand rotates around the circle, which has 360 degrees, or 2π radians. Every second, it moves clockwise by a little bit. $1/60$th of 2π, to be exact. Therefore, at every time point, you can compute the cosine of the **angle between the seconds hand and 12 o'clock** (due North).

(Note that, mathematically, the reference for 0 degree is actually 3 o'clock, or due East. But we'll use 12 o'clock for this question because it's a little more intuitive.)

---

[1] Let `t=0` be the start of the minute (pointing at 12 o'clock). What is the angle (in radians) between the seconds hand and the reference (12 o'clock) at `t=15` seconds? Save that in the variable `angle_t15`. Hint: `np.pi` has the value for π.

[1] What is the cosine and sine of that angle? Answer this here without coding.

ANSWER: sine = 1, cosine = 0

[1] Complete the code to compute and print the quantities above and make sure this is what you expected.

[1] What is the angle at `t=30` seconds? What is the cosine and sine of that angle?

ANSWER: The angle is a 180 degree angle, or pi in radians. sine = 0, cosine = -1

[1] After how many seconds will the angle be 2π? In other words, in how many seconds will the seconds hand have completed a full revolution around the clock?

ANSWER: The clock should complete a full revolution in 60 seconds.

[1] What is the frequency of rotation of the seconds hand on a clock (in cycles per second, Hz)? Store that in the variable `clock_freq`.

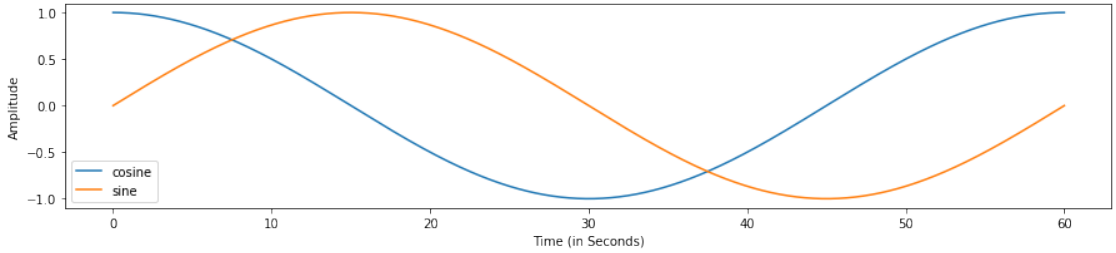
[4] The above questions gave you two points where time is mapped to angle. In code below, compute the corresponding angle (**in radians**) for all time points in the time vector `t_clock`, and store it in the variable `angle_clock`. Note, you should use the variables `t_clock`, `clock_freq`, and the constant π. Then, using this angle array, compute the cosine and sine of the entire vector and store them in `cos_clock` and `sin_clock`. Finally, plot both the cosine and sine values over time. Remember to label your lines and axes. Hint: both cosine and sine should go through just a single cycle.

```
In [3]: print(np.pi) # print the value for pi

        # compute the cosine and sine at t=15 seconds
        angle_t15 = (15 / 60) * (2 * np.pi)

        # print the cosine and sine of that angle
        print(np.sin(angle_t15))
        print(np.cos(angle_t15))

        clock_freq = 1/60

        # creates a vector that represents the discussed "clock" example
        t_clock= np.arange(0,60,1/fs)

        # takes the clock representation and converts it into a vector of angles that correspo
        angle_clock = t_clock * (clock_freq * (2 * np.pi))

        # finds the corersponding cosine and sine values of corresponding values with regards
        cos_clock = np.cos(angle_clock)
        sin_clock = np.sin(angle_clock)

        print(cos_clock)
        print(sin_clock)
```

```
        plt.figure(figsize=(15,3))

        plt.plot(t_clock, cos_clock, label = 'cosine')
        plt.plot(t_clock, sin_clock, label = 'sine')
        plt.legend()
        plt.xlabel('Time (in Seconds)')
        plt.ylabel('Amplitude')
```

```
3.141592653589793
1.0
6.123233995736766e-17
[1.          1.          0.99999999 ... 0.99999997 0.99999999 1.        ]
[ 0.00000000e+00  8.37758040e-05  1.67551607e-04 ... -2.51327410e-04
 -1.67551607e-04 -8.37758040e-05]
```

```
Out[3]: Text(0, 0.5, 'Amplitude')
```



7 [5] Q2: Cosine and Sine Dot Product

You should be familiar with how to construct a cosine and sine wave of a particular frequency at this point. To start analyzing our brain data, we'll use a cosine/sine of a different frequency: 8Hz.

[1] Make an 8Hz cosine and sine wave (store in variables `cos` and `sin`) using the time vector `t_short` and the frequency variable `freq`. This should be analogous to what you did above.

[1] Plot, in the same graph, the 2 seconds of LFP signal (already done), as well as the cosine and sine you have just generated. Remember to label the traces and axes.

[1] How many cycles should an 8Hz oscillation go through in 2 seconds? Count the peaks in your plot to make sure the answer is what you expect.

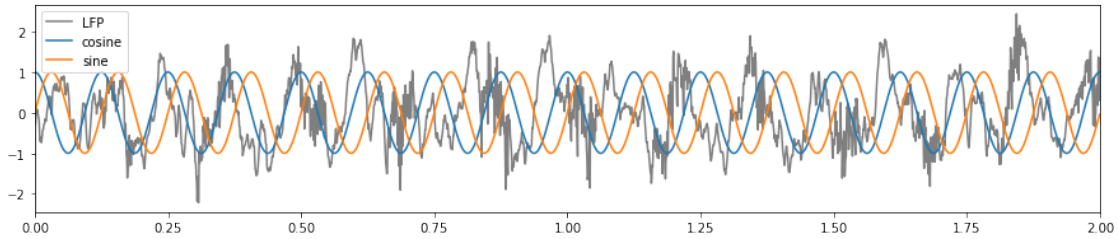
ANSWER:

[2] Compute the dot product between the 2 second chunk of LFP and the cosine vector, and store in the variable `a_coef`; repeat for the LFP and the sine vector, store in variable `b_coef`.

```
In [4]: freq = 8 # Hz
        angle = t_short * (8 * (2 * np.pi))

        # using the numpy functions to find the cosine and sine of a given angle
        cos = np.cos(angle)
```

```
sin = np.sin(angle)

plt.figure(figsize=(15,3))

# plotting code for signal, cosine, and sine waves
plt.plot(t_short, lfp_short, 'k', label='LFP', alpha=0.5)
plt.plot(t_short, cos, label = 'cosine')
plt.plot(t_short, sin, label = 'sine')
plt.xlim([0,2]); plt.legend()

a_coef = np.dot(lfp_short, cos)
b_coef = np.dot(lfp_short, sin)
print(a_coef, b_coef)
```

```
355.11525874773554 -663.1592906945903
```



8 [5] Q2 - continued

We'll be generating sine waves and performing the dot product very often from this point on, so we better make them into functions. Complete the functions below. Note that you essentially already wrote all the code in the cell above, we just want to wrap them inside functions for convenience.

[1] For the function `make_cos_sin`: given a time vector `t`, and a frequency of interest `freq`, return the corresponding cosine and sine waves.

[2] For the function `dot_cos_sin`: given a signal, the associated time vector, and the frequency of interest, first make cosine and sine waves by calling the function `make_cos_sin`, then compute and return the dot product between `signal` and the cosine and sine wave.

[1] Call your function `dot_cos_sin` to repeat the above analysis (for 8Hz waves). Confirm that you get the same answer as above. Print the value.

[1] Perform the same computation, but for 30Hz cosine/sine waves.

```
In [5]: # implement a function that calculates the cosine and sine given a time (or time array)
        def make_cos_sin(t, freq):

            angle = t * (freq * (2 * np.pi))
            cos = np.cos(angle)
```

```
        sin = np.sin(angle)

        return cos, sin
    # creating the function that caluclates the a and b fourier coefficients based on a gi
    def dot_cos_sin(signal, t, freq):

        # call the created make_cos_sin function to reduce the amount of code that we need
        cos, sin = make_cos_sin(t, freq)

        # caluclate the a_coefficent and b_coefficient using the numpy dot products using
        # make_cos_sin function
        a_coef = np.dot(signal, cos)
        b_coef = np.dot(signal, sin) * (-1)

        return a_coef, b_coef

    # compute the dot products for 8Hz cosine and sine waves
    freq = 8
    print(dot_cos_sin(lfp_short, t_short, freq))

    # compute the dot products for 30Hz cosine and sine waves
    freq=30
    print(dot_cos_sin(lfp_short, t_short, freq))
```

```
(355.11525874773554, 663.1592906945903)
(19.802550750977826, -4.756835689910389)
```

9 [3 + 2 Bonus] Q3: Complex Exponential, Power (Amplitude), and Phase

Why did we just do that? One way to conceptualize the DFT is that it's a "sinusoidal sieve": by passing a signal through the DFT, we split up the signal into different frequencies, and measure the **independent** contribution of each of those component frequencies. This is what a prism does, separating white light into lights of different frequencies. What we did in Q2 was to construct a small part of the sieve, creating the "hole" at 8Hz and 30Hz. Passing the signal through (via dot product) "filters" out everything except the component of the signal at 8Hz. It's like wearing red goggles: it blocks/absorbs all other colors except red.

[BONUS: 2] Construct the complex exponential using `np.exp`, and perform the dot product directly with the signal `lfp_short`, confirm that you get the same complex number as `(a_coef + i*b_coef)`, which I've printed for you.

[2] Complete the function `compute_amp_phase` for computing the amplitude (magnitude) and phase (angle) from the coefficients. You may use the numerical formula for magnitude and angle in A0, or use `np.abs()` and `np.angle()` on the complex number itself.

[1] Using the above function, find the magnitude and phase of the 8Hz component of the signal. Store them in the variables `amp` and `phase`.

```
In [6]: # BONUS: computing the dot product from the complex exponential
        exp = (complex(0,1)) * (-(2 * np.pi) * freq) * t_short
        complex_exp = np.exp(exp)

        # print the dot product
        print(np.dot(complex_exp, lfp_short)) # _FILL_IN_YOUR_CODE_HERE

        # computing the dot product separately
        freq = 8
        a_coef, b_coef = dot_cos_sin(lfp_short, t_short, freq)
        print(a_coef + 1j*b_coef) # printing the coefficients as a complex number, note the mi

        def compute_amp_phase(a_coef, b_coef):

            # initialize an array that has a column for squared a coefficients (cosine) and sq
            amp_array = [a_coef**2, b_coef**2]
            amp = np.sum(amp_array)

            # to find the phase shift we must caluclate the inverse tangent of b_coefficient d
            phase = np.arctan(b_coef / a_coef)

            return amp, phase

        amp, phase = compute_amp_phase(a_coef, b_coef)
        print(amp,phase)
```

```
(19.802550750977833-4.756835689910385j)
(355.11525874773554+663.1592906945903j)
565887.0918300232 1.0791613329579512
```

10 [8] Q4: Wave Number & Frequencies of DFT

In the questions above, we've cherrypicked a particular frequency (8Hz), and used that sieve to query for the contribution of the 8Hz component in our signal. To build the full DFT, you have to complete the sieve. A sieve merely separates components of a material, it does not make anything disappear. The same is true for the DFT. The real ingenuity behind the Fourier Transform is that a signal - *any signal* - can be decomposed into contributions from sinusoidal components (or bases), if you pick the right frequencies. In other words, from the signal's Fourier transform, you can reconstruct the original signal with 100% accuracy, using the *inverse* Fourier Transform.

One idea is to simply do this for every single possible frequency, 8Hz, 8.1Hz, 8.01Hz, etc. But we quickly realize that there are an infinite number of frequencies along the real number line, and that's probably an overkill anyway. If our discrete time signal is only 1000-points long, using a million sinusoids to represent that isn't very clever.

The intuitive answer (and the right answer) is that **we need as many frequencies as we have samples in the signal itself**. In the LFP signal above (`lfp_short`), there are 2500 points, so we need 2500 frequencies of sine waves to deconstruct that signal.