Lab 9: Discrete Fourier Transforms and Optimizing Code

Objectives

By the end of this lab you should be able to:

- Read data from a text file into an array.
- Implement a straightforward (but slow) Fourier transform using loops in Python.
- Create two plots in the same plot window.
- Replace loops with **`numpy`** array operations.

Prelab Questions: *Read the rest of this lab* and Sections 7.1 and 7.2 of your book, and only then answer the following questions. Although the book uses **`cmath`** for exponentials that can handle complex numbers we will simply use numpy. Please *do not* use **`cmath.`**

1. Read about **`time.time()`** at *https://docs.python.org/3.7/library/time.html#time.time*. What does **`time.time()`** return? Explain in words how you could use **`time.time()`** to measure the time it takes a Python function to run. **HINT***:* You may need to call it more than once.
2. Imagine you are working with data in which there are 2000 data points recorded for one period, all real numbers.
    1. How many *k* values are there?
    2. What is the largest value of *k*?
    3. If the input data is periodic with period 0.1999 sec, what is the largest frequency, *f?*

Background

General form for complex input

Recall Fourier's assertion that any periodic function y*(x),* periodic over *x*=0 to *L and sampled at N evenly spaced points,* can be written as a sum of sinusoidal basis functions

$$y(x) = \frac{1}{N}\sum_{k=0}^{N-1} c_k \exp\left(i\frac{2\pi k x}{L}\right). \qquad \text{Eqn. (0)}$$

The coefficients in the sum, c_k, are called the *discrete Fourier transform* of the function *y(x).*

If the function is known at positions $x_n = nL/N$ then this can be written without any reference to the period *L:*

$$y(x_n) = \frac{1}{N}\sum_{k=0}^{N-1} c_k \exp\left(i\frac{2\pi k n}{N}\right). \qquad \textit{Eqn. (0.5)}$$

If we write the value of the function at those positions as $y_n = y(x_n)$, then the coefficients c_k are given by

$$c_k = \sum_{n=0}^{N-1} y_n \exp\left(-i\frac{2\pi k n}{N}\right), \qquad \text{Eqn. (1)}$$

where $0 \leq k \leq N-1$. *Much faster* methods exist (and we will explore them shortly), but this method is the most direct. **NOTE:** The data can be evenly spaced in time (t) or position (x). Which kind of function you are sampling affects the meaning of "frequency" in the DFT as either a (temporal) frequency (cycles/time) or spatial frequency (cycles/length).

Special case: real function

A special case that often applies in physics is that the input data or function *y(x)* is real-valued (i.e. it is not complex). In that case it can be shown that only *half* of the frequencies are independent. For $k < N/2$ the values are calculated using Eqn. (1). For values of k that are larger than N/2, the c_k are complex conjugates of the c_k from the smaller values of k ($c_{N-k} = c_k^* = c_k$).

In this special case, the formula for the c_k is the same as before, Eqn. (1), but the maximum value of k is *N/2* (if N is even); the book explains the condition for the odd case and how to write both compactly in Python. The formula for y changes a little because the sum only goes up to N/2. It turns out that

$$y(x_n) = \frac{1}{2N}\left[2c_0 + (-2)^n c_{N/2} + \sum_{k=1}^{N/2-1} \Re\left(c_k \exp\left(i\frac{2\pi k n}{N}\right)\right)\right], \qquad \textit{Eqn. (2)}$$

where $\Re(z)$ means the real part of the complex number z.

Important note about units

The book (and the discussion above) presents both k and n as unitless (because they are) but we typically want to know positions in physical units like seconds (or meters, depending on the application) and frequencies in units like Hz (or inverse meters, if appropriate).

Converting is, fortunately, straightforward. Multiply n by L/N to get the position x_n and multiply the wave number k by N/L to get the frequency f.

Part 0: Download the Starting Code From GitHub as a zip file

Seriously, you will save yourself time later. Just do it.

Part 1: Pressure versus Time Graph

- You will write a program to compute the acoustic spectrum of an audio data file.
- A data file of a recorded sound (**`c4.txt`**) is available on GitHub. Start by downloading the data file **`c4.txt`** to the directory in which your Python program will be written. The data is stored in columns. The first column is time (s), and the second is pressure (with arbitrary units). The data was taken using a microphone and LoggerPro, then saved in text format. Look at your homework with the sunspots problem for details of how to read a text file with Python (or google it).

- Create a Python program to read the first column as an array of times **`time`**, and the second column as an array of pressures **`pressure`**.
- Graph the pressure versus time.
- From the graph estimate the dominant period of the oscillation, and calculate the dominant frequency component in Hz.

Part 2: Acoustic Power Spectrum 1.0: (slow)

- Write a function called **`dft_two_loop`** which takes only one argument, the data, **`y`**, and calculates the DFT using equation (1). One loop should be over *k.* Inside that loop write a second loop to sum over *n.* The code for doing this is in Sections 7.2 of the book. Remember, the maximum value of *k* is **`N//2`** since the input data is real.

- Produce an acoustic power spectrum which is a plot of the modulus of the Fourier coefficients ($|c_k|^2$) versus the frequency *f*. Compare the frequency at which the power is a maximum to the value you estimated in Part 1.

Part 3: Acoustic Power Spectrum 2.0: (faster)

- Create a copy of your **`dft_two_loop`** function and rename it **`dft_one_loop`**. In your new function, replace the innermost loop over **`n`** with a **`numpy`** expression that creates an array of **`n`** elements containing all the Fourier terms in the sum.
    - This can be done in a single operation by replacing **`n`** with something like **`np.arange(N)`** and **`y[n]`** with **`y.`**
    - Once you have created an array of all the Fourier terms, you can sum over the array using a single **`np.sum`**.
    - When you are done, you should have replaced the entire **`for n in range(N):`** loop with a single line.

- Add the output of **`dft_one_loop`** to your plot of the power spectrum obtained the **`dft_two_loop`** (and, of course, add a legend).

- Compare the speed of **`dft_one_loop`** with **`dft_two_loop`**. Though there are much better ways to measure the time a function takes to execute, feel free to estimate it by printing out the time before and after the function runs.

- Comment in the notebook on the difference in time between the two implementations.

Part 4: Acoustic Power Spectrum 3.0: use numpy.fft.rfft, no loops, (fastest)

- Create a copy of your **`dft_one_loop`** function and rename it **`dft`**. In your new function, replace the current code with a call to the function from numpy for calculating the FFT of a real function, **`np.fft.rfft`**. It returns the DFT in the same format as the one we have been calculating ourselves.

- Compare the speed of **`dft`** with the previous two methods. Though there are much better ways to measure the time a function takes to execute, feel free to estimate it by printing out the time before and after the function runs.

- Add the output of **`dft`** to your plot of the power spectrum.

Part 5: Inverse Transform

- Calculate the inverse Fourier transform, i.e. calculate y_n from the c_k using Eqn. (2).
- Compare the y_n with the input data (by graphing or taking the difference) and comment.

OPTIONAL [Extra Credit][but honestly not worth the time]

Part 6: Acoustic Power Spectrum 4.0: no loops, but is it faster?

- **`numpy`** has a convenient way of repeating arrays, called broadcasting, that can help eliminate loops.
- Read about **`numpy`** array broadcasting at https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html then answer the questions below. Feel free to try them out in python!
    - If **`k = np.array([1, 2])`** and **`n = np.array([0, -1, -2])`**, what will you get when you try **`k + n`**?
    - What is the shape of **`k[:, np.newaxis]`**?
    - What is the result of **`k[:, np.newaxis] + n`**? Does it make sense? Explain.

- The version of the function you have from Part 3 has a loop over k in which you always perform the same action, once for each of the **`N // 2 + 1`** values of k. If we had a way of telling **`numpy`** to repeat the values of n for each of the values of k we could do the whole calculation without loops.

- **Array broadcasting:** removing loops using **`numpy`**
    - Array broadcasting works by lining up multi-dimensional arrays from right to left. If k is an array that is **`N/2+1`** × **`1`** and n is a vector of length **`N`**, and you ask **`numpy`** to do an operation with them, it first lines them up like this:

Variable	Dimensions (before broadcasting)		
`k`	**`N//2+1`**	×	**`1`**
`n`			**`N`**

    - After aligning, "missing" dimensions or dimensions of size 1 are repeated to make the dimensions match. Once aligned by broadcasting, the two quantities can be combined.

Variable	Dimensions (after broadcasting)			
`k`	**`N//2+1`**	×	**`N`**	Make **`N`** copies of **`k`**, one for each **`n`**.
`n`	**`N//2+1`**	×	**`N`**	Make **`N/2+1`** copies of **`n`**, one for each **`k`**.

- Start a new function called **`dft_no_loop`** and use array broadcasting to write it without loops. An outline is below.
  - Create variables for storing **`n`** and **`k`**

    ```
    n = np.arange(N)
    k = np.arange(N // 2 + 1)
    k = k[:, np.newaxis]    # Add extra dimension to k of size 1
    ```
  - Compute the Fourier terms

    ```
    fourier_terms = ?
    ```

    This expression should look identical or nearly identical to the expression you used to compute the Fourier terms in Part 3, an array containing all **`n`** Fourier terms you need to sum. However in this context, if you have defined **`k`** and **`n`** properly, it will be an **`N//2+1`** × **`N`** array.
  - Finally, compute the sum over the Fourier terms using

    ```
    c = fourier_terms.sum(axis=1)
    ```

    This uses the **`sum`** method applied to the array **`fourier_terms.`** The **`axis`** argument controls which direction to perform the sum (otherwise all the elements in the entire 2D array are summed over). In this case we want to sum over *n*, which is the second dimension, so we use **`axis=1.`**
- Add the output of **`dft_no_loop`** to your plot of the power spectrum.
- Compare the time it takes to run this version to the other two. Comment in the code on whether there was a speed improvement as suggested by the title of this part of the lab.

For whole program (all parts)	
`dt = ?`	This is the time interval between the equally spaced data samples in the file. You should figure out a way to make the program determine this from the data and **NOT** hard code a single number. **HINT:** All the data points are evenly spaced in time, including the first 2 data points.
`N = ?`	Define **`N`** as the number of data samples in the array **`t`**. How do you determine the size of **`t`**?
`f = ?`	Calculate the frequencies at which we will obtain the Fourier transform. You may *not* use a loop. Recall that the expression in Python for obtaining the number of frequencies in the Fourier transform of a real function is **`N // 2 + 1.`**
Part 2: `dft_two_loop – see book, but don't use cmath`	

Part 3: `Determining the run time`	
`import time`	Imports library for time commands, useful for determining how long commands took to execute.
`print time.time()`	Prints the time in seconds since "epoch".

What to turn in at the end of lab:

- Upload the Python code and the notebook containing your write-up to GitHub.

You should save a copy of the code on a flash drive or email the program to yourself. Your work will be evaluated and returned by Friday. Revisions will be due the beginning of next lab.