Fun with Fourier transforms

H. Cynthia Chiang
CRAQ summer school 2022

1 Introduction

The Fourier transform is one of the most important mathematical tools used in signal processing, both in astronomy and a wide range of other fields. There are many online resources describing Fourier transforms, including the following websites:

- http://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/
- http://www.thefouriertransform.com/

This tutorial will cover some of the basics of Fourier transforms, but I encourage you to consult the above references for more in-depth information.

The Fourier transform is a method of decomposing a function into sines and cosines of different periodicities. For a continuous function $f(x)$, the forward and inverse transforms can be written respectively as

$$F(k) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i k x} dx \tag{1}$$

$$f(x) = \int_{-\infty}^{\infty} F(k) e^{2\pi i k x} dk. \tag{2}$$

The first equation gives us $F(k)$, the Fourier transform of $f(x)$: this quantity represents the amplitudes of all the sines and cosines that need to be combined to reconstruct our original function. (Recall that the complex exponential $e^{-2\pi i k x}$ can be expressed in terms of sine and cosine according to Euler's formula.) Reconstructing $f(x)$, or performing the inverse Fourier transform, is given by the second equation: the $F(k)$ amplitudes are used as the coefficients of the sines and cosines, and integrating over all k yields the original $f(x)$.

Continuous Fourier transforms assume that the function $f(x)$ exists for all infinite values of x. However, real astronomical signals that are recorded as data aren't infinite (unless you fill an infinite number of computer hard drives). Because real data have finite lengths, we use the *discrete* Fourier transform (DFT) when processing signals. A data set, recorded signal, etc. is simply a series of numbers (samples) that we can denote as f_x, where x is the sample number, and the total number of samples is N (so that $x = 0, 1, 2, ..., N-1$). The DFT of f_x and the inverse transform F_k can be written respectively as

$$F_k = \sum_{x=0}^{N-1} f_x e^{-2\pi i x k / N} \tag{3}$$

$$f_x = \frac{1}{N} \sum_{k=0}^{N-1} F_k e^{2\pi i x k / N}. \tag{4}$$

As is the case with the continuous Fourier transform, the first equation decomposes the original data series f_x into its constituent sines and cosines, and the amplitudes of these constituents are given by F_k. The second equation shows that f_x can be reconstructed by summing sines and cosines with the appropriate F_k coefficients. Because N is a finite number, we can evaluate these sums on a computer. As a side note, computer implementations of this calculation generally employ *fast Fourier transform*

(FFT) algorithms. Because FFTs are used so often for calculating DFTs, the two acronyms are often used interchangeably. For the rest of this tutorial, I'll use FFT as my terminology of choice.

Before we continue on with the mathematical properties of FFTs, let's pause and talk about why we care in the first place. The variables x and k can represent a variety of physical quantities in real life, and one very common pair is *time* and *frequency*. In other words, a signal that's sampled at constant intervals in time can be represented by a sum of sines and cosines at different frequencies. You already know one real-world application of this: music. Music is a signal that varies with time, and it's composed of tones at different *pitches* or frequencies. For example, if you hear many high pitches, you might guess that you're listening to a bird or a flute; if you hear many low pitches, you might guess that you're listening to techno with a whomping bass. In other words, analyzing the pitches can reveal some sort of information about the nature of the music. Similarly, analyzing the frequency content of time-ordered astronomical data can also reveal hidden information and periodicities.

Exercise 0: Constructing a square wave
Directory location: `square_wave`

The python script `square_wave.py` in the above directory demonstrates that sinusoidal functions can be added together to construct an arbitrary function. Here, we will build a square wave one piece at a time. As the plots update, the left subplot shows overplotted sine waves that are summed to form the graph shown in the right subplot. As more and more sine waves at are added together, the sum looks increasingly like a square wave. All of the plots are saved to disk for convenience, and you can increase the number of sines by changing the value of `i_max` inside the code.

2 Properties of FFTs

We've already mentioned that astronomical data are finite in length, and there's one other important property that's relevant for FFTs: *data are real numbers*. If f_x are real, it can be shown that F_k obeys the following symmetry:

$$F_{N-k} = \bar{F}_k, \tag{5}$$

where $k = 0, 1, 2, ..., N-1$, and the overbar denotes the complex conjugate. Because of this symmetry, *the FFT of f_x is described completely by only $N/2 + 1$ complex numbers*.

Another important property of FFTs is the *shift theorem*. If f_x is multiplied by a linear phase $e^{2\pi i x m/N}$, where m is an integer, then F_k is shifted so that each F_k value is replaced by $F_k - m$. The subscript is interpreted modulo N so that the shift is "circular" or periodic. Similarly, a circular shift in f_x results in F_k being multiplied by a linear phase.

Exercise 1: One-dimensional FFTs of common functions
Directory location: `one_dimensional`

Open the python script in an editor and *read the contents carefully*. Once you have an idea of what the code does, run it. As the plots appear, you can use the various buttons to zoom and pan the images. The plots are also automatically saved to disk as png files.

(1a) Use Equation 3 to calculate the FFT of a delta function that has an amplitude of one and that is centered on the zeroth sample number. Compare your answer to the plot that you generated with the python script.

(1b) Edit the script so that the delta function is centered at an arbitrary non-zero sample number of your choosing. Try several other center locations, and examine how the location affects the FFT magnitude and phase.

(1c) Use Equation 1 to calculate the continuous Fourier transform of a boxcar function that is zero everywhere except $-L/2 < x < L/2$, where it has an amplitude of one. Here L is a constant that defines the width of the boxcar. Check that your answer is correct by comparing with the script output.

(1d) The script calculates the full FFT of the input function, assuming that it could be complex; however, we know that our inputs are real. The `numpy` module also includes FFT routines that take explicitly real-valued inputs. Find the appropriate real-valued FFT function by searching online, replace the full complex FFT in the script, and rerun the script. How do the outputs of the real and complex FFT differ?

3 Generalizing to two dimensions

As you well know, much of astronomy deals with images, which are two-dimensional arrays of data. It's therefore useful to discuss how Fourier transforms can be extended beyond one dimension. The two-dimensional FFT and inverse FFT are defined respectively as

$$F_{u,v} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f_{x,y} e^{-2\pi i u x/M} e^{-2\pi i v y/N} \tag{6}$$

$$f_{x,y} = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F_{u,v} e^{2\pi i u x/M} e^{2\pi i v y/N}. \tag{7}$$

The basic principle is still the same as the one-dimensional transform, but we now use sines and cosines in two dimensions to construct an arbitrary set of data $f_{x,y}$ that are sampled at equally spaced intervals in x and y.

Exercise 2: Exploring two-dimensional FFTs
Directory location: `two_dimensional`

Open the python script in an editor and *read the contents carefully* before running. In each of the exercises below, make the described edits to the code and *try to predict the result* before you run the code. Do the results match what you guessed?

(2a) Use the script to calculate the 2D FFT of a rectangle with dimensions of your choosing.

(2b) Move the same rectangle from part (a) to several different locations, and compare the FFT results. Do the magnitudes and/or phases change?

When you computed the 2D FFTs in the previous exercises, the output $F_{u,v}$ were arranged so that the origin is at the center of the plot. In the following exercise, you will experiment with *filtering* images by taking the FFT, excising a portion of $F_{u,v}$, and then taking the inverse transform.

Exercise 3: Filtering images
Directory location: `filtering`

Open the python script in an editor and *read the contents carefully* before running. In each of the exercises below, make the described edits to the code and *try to predict the result* before you run the code. Do the results match what you guessed?

(3a) There are two masking options given in the code, one of which is commented out. Run the code twice, using one option at a time. How do the two masking options impact the final images?

(3b) Try running the code with different values for the mask radius. How does changing the mask size affect the filtered image?

(3c) Replace the circular mask with a new mask that consists of a centered vertical stripe (suggested width of 6 pixels, but you're free to choose otherwise). You will need to write your own code to construct this mask. What happens when you use this mask to create a filtered image? Repeat the calculation with a centered horizontal stripe.

You should now have a rough sense of what information is contained in the magnitudes and phases of Fourier transforms. Let's take this concept one step further. In the following exercise, you will take the FFTs of two different images, and you will swap and modify the magnitudes and phases to create a "Franken-FFT." You will then take the inverse FFT of this Franken-object to reconstruct a new, mystery image.

Exercise 4: Swapping magnitudes and phases
Directory location: `phase_swap`

Once again, open the python script in an editor and *read the contents carefully* before running. In each of the exercises below, make the described edits to the code and *try to predict the result* before you run the code. Do the results match what you guessed?

(4a) When you first run the python script with its default settings, the code takes the FFTs of both images, swaps the *phases*, and then takes the inverse transforms to create "Franken-images."

(4b) By uncommenting the appropriate lines of code, repeat (4a) but this time swapping the *magnitudes* instead of the phases.

(4c) Uncomment the appropriate lines of code to give both images *random magnitudes*.

(4d) Uncomment the appropriate lines of code to give both images *random phases*.