The Fast Fourier transform

How to use it, and common pitfalls

Joëlle Begin, July 2020

Working in cosmology, you'll be using the Fourier transform *a lot*. We all know the functional form of the Fourier transform,

$$\tilde{f}(k) = \int_{-\infty}^{\infty} dx\ f(x) e^{-ikx}$$

But although the things we study are in reality continuous, when we're dealing with real experimental data or numerical simulations we usually work with discrete **samples** of the continuous thing. So, you have to be able to do a numerical Fourier transform for all your discrete data!

Luckily, you don't have to re-invent the wheel. Some clever programmers already came up with an algorithm for computing the discrete Fourier transform, which numpy has helpfully wrapped in the function `fft`. Let's see the FFT in practice with a simple example. Let

$$f(x) = \sin\left(\frac{2\pi x}{L}\right)$$

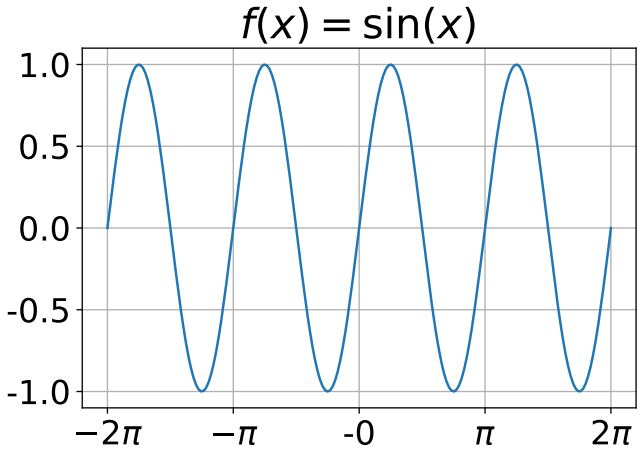
Then the Fourier transform will be

$$\tilde{f}(k) = \frac{1}{2i}\left[\delta\left(k + \frac{2\pi}{L}\right) - \delta\left(k - \frac{2\pi}{L}\right)\right]$$

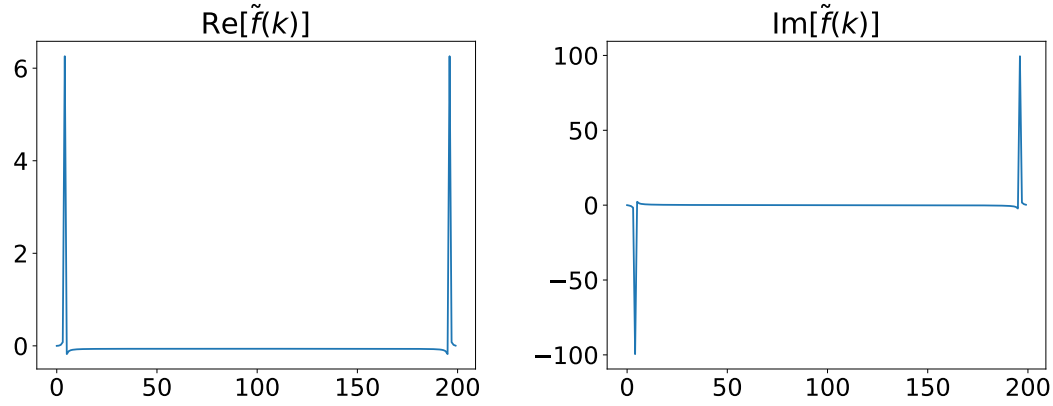
So we should expect a completely imaginary signal, with spikes at $k = 2\pi/L$ and $k = -2\pi/L$. Let's see what the code does!

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.fft import fft

def function(x, L):
    return np.sin(2*np.pi*x/L)

L = np.pi
x = np.linspace(-2*np.pi,2*np.pi,200)

f = function(x, L)
f_tilde = fft(f)
```

So we have a pure sine wave as our function



and when we plot `f_tilde`, we get the following:



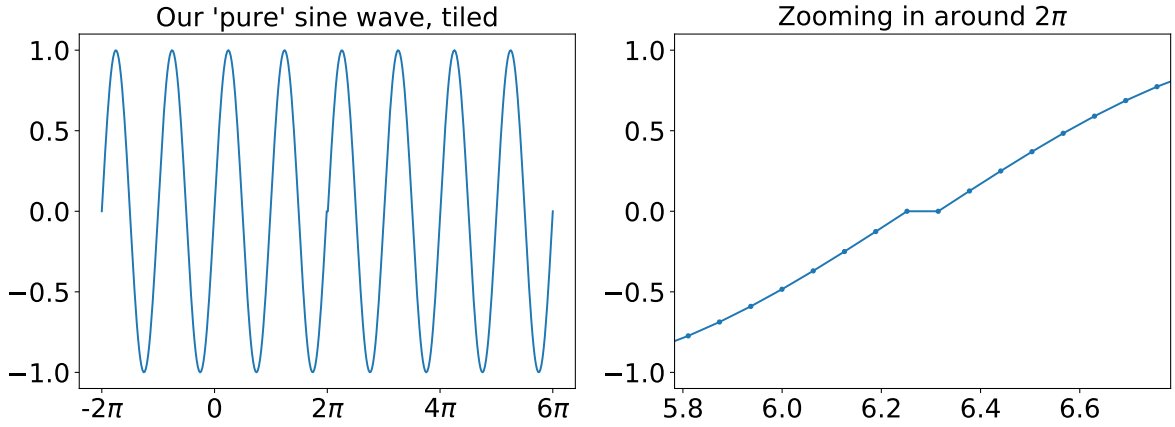
Uh-oh. Something strange is going on. The first thing that seems weird is that the real part is not zero, as we would have expected. What's happening here?

FFT PITFALL: The FFT algorithm assumes periodic boundary conditions.

We're giving the FFT what we think is a pure sine wave, but the FFT effectively tiles this signal we're giving it. So, after tiling, is it a pure sine wave? Let's try the following:

```
x_tiled = np.linspace(0,8*np.pi,400)
tiled = [list(f)*2][0]
```

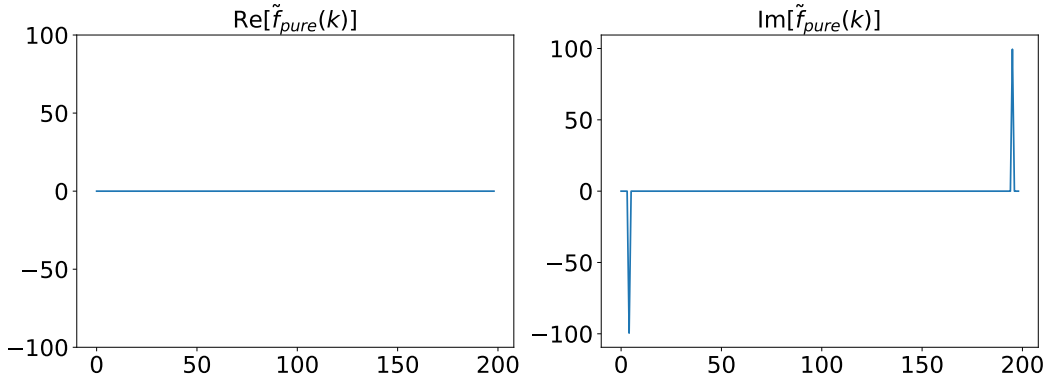
and plot what the tiled function looks like.



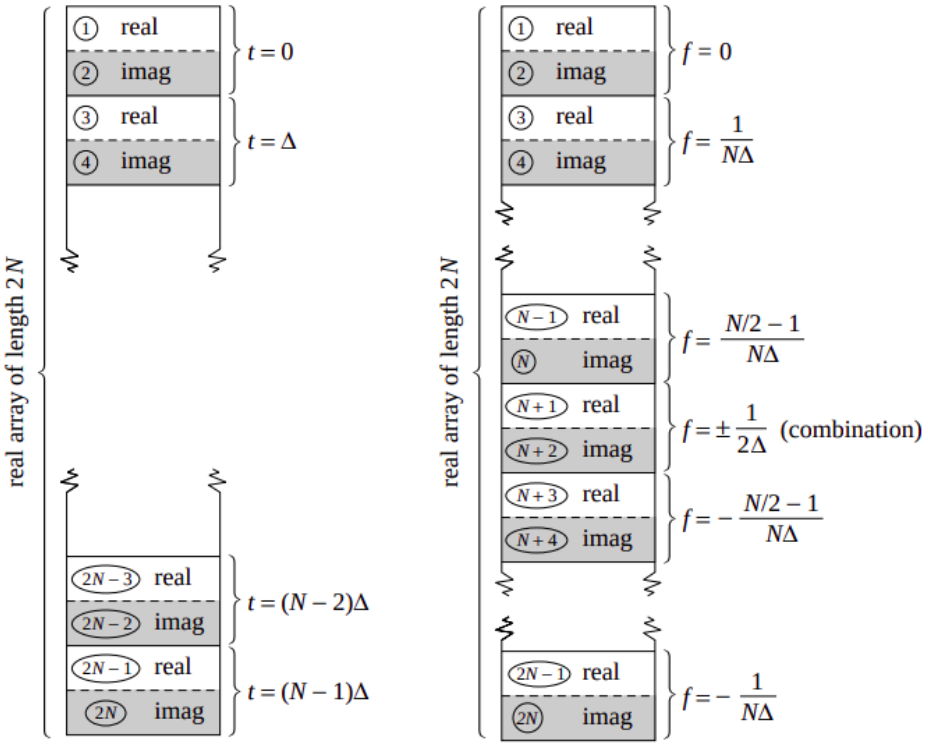
Aha! There's a little hump right where we tile the two sine waves. This is because if you look at how we defined our range of angles above, we did `np.linspace(-2*np.pi,2*np.pi,200)`. This means that `f[0] = 0` and `f[-1] = 0`, so when the FFT algorithm applies periodic boundary conditions, there will be two zeros in series. That is, in FFT's eyes, this is not a pure sine wave! Let's give FFT a pure sine wave now.

```
pure_sine = f[:-1] #popping out last element that would give repeated zeros
pure_sine_tilde = fft(pure_sine)
```

Now, plotting `pure_sine_tilde`:



Exactly as we expected! Wonderful! We've dodged the first FFT pitfall. But let's keep going, cause trust me, there are many more. How about the location of the peaks? We expect spikes at $k = 2\pi/L$ and $k = -2\pi/L$. Note I haven't plotted anything on the x-axis yet, so let's see how FFT decides where to put the peaks. Let's look at this picture from the fantastic book *Numerical Recipes*.



There's a lot to unpack here. The panel on the left is a representation of the input that FFT expects. The algorithm assumes an input of N elements[1], sampled at a resolution of Δ. Then, after the FFT has been computed, the output in fourier space has a resolution of

$$\Delta f = \frac{1}{N\Delta}$$

[1]Here they have $2N$ elements because in the programming language this textbook is written there is no complex data type, so they represent a complex array of some signal S by letting `array[i] =` $\mathrm{Re}(S_i)$ and `array[i+1] =` $\mathrm{Im}(S_i)$