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*. Here is the functional form of the Fourier transform,

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

But although the things we study are in reality continuous, when we're dealing with real experimental data or numerical simulations we 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

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

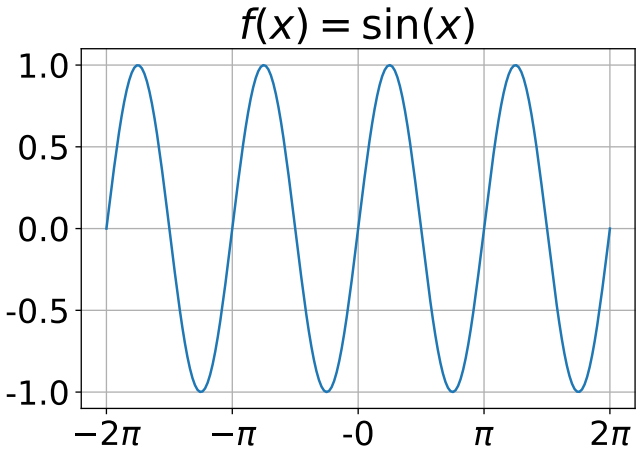
Then the Fourier transform will be

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

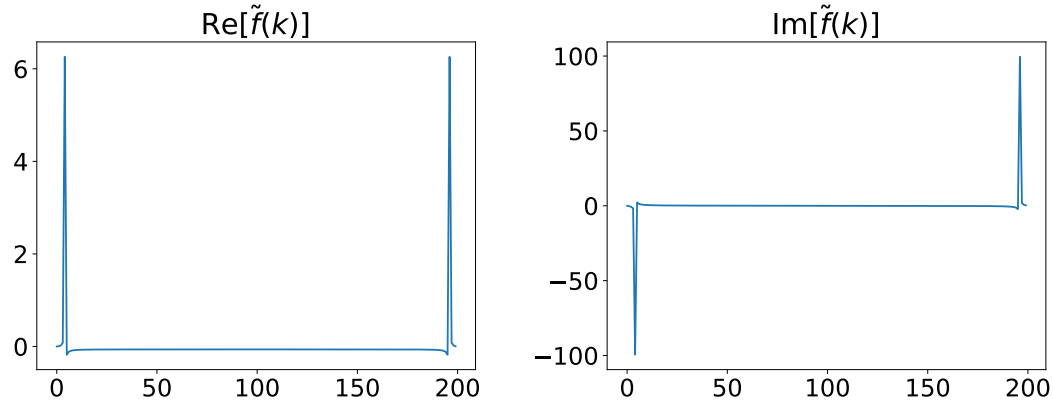
So we should expect a completely imaginary signal, with spikes at $k = 2\pi/L$ and $k = -2\pi/L$. Let's see what the code does!

In [1]:
```
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:



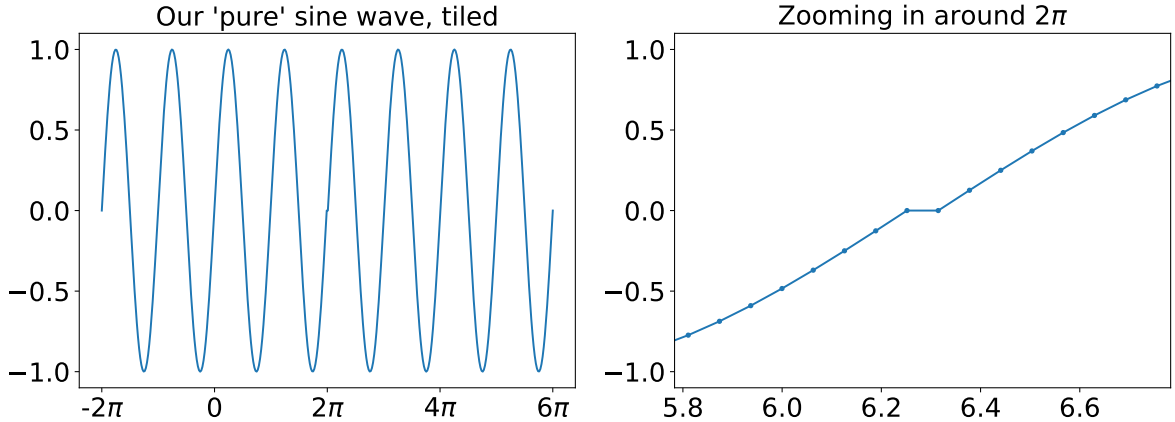
Uh-oh. Something strange is going on. The first thing that seems weird is that the real part is not zero, as we would have expected. What's happening here?

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. It assumes your input is a sample of exactly one period of an infinite periodic signal. So, after tiling, is it a pure sine wave? Let's try the following:

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

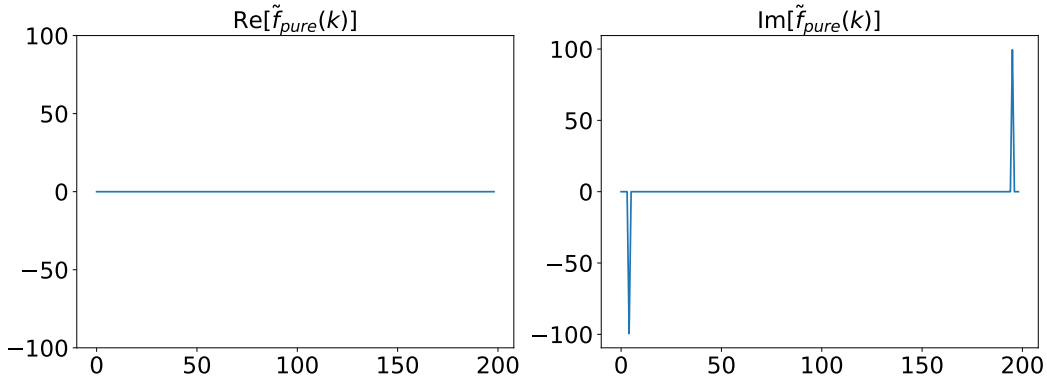
and plot what the tiled function looks like.



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.

In [3]:
```
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 that I haven't told `matplotlib` what goes 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*.

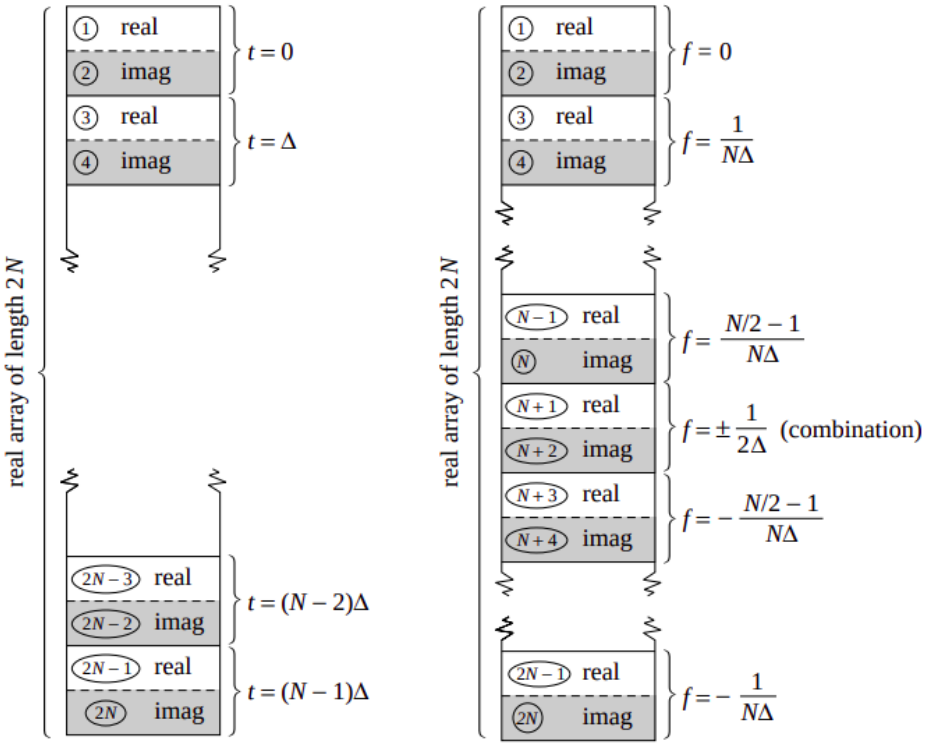


Figure 1: FFT diagram from *Numerical Recipes*

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

[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] = Re(S_i) and array[i+1] = Im(S_i).

output in Fourier space has a resolution of

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

Which is known as the *nyquist frequency*. So the values returned by `fft` correspond to the Fourier transform sampled at $1/N\Delta$, $2/N\Delta$, and so on. Again, numpy has a nice function that takes care of this for us.

In [4]:
```
from numpy.fft import fftfreq

Delta = x[1] - x[0]
freqs = fftfreq(len(pure_sine), Delta)

nyquist_theory = 1/(len(pure_sine)*Delta)
nyquist_fft = freqs[1]

print("Theoretical nyquist frequency: ", nyquist_theory)
print("fftfreqs nyquist frequency: ", nyquist_fft)
```

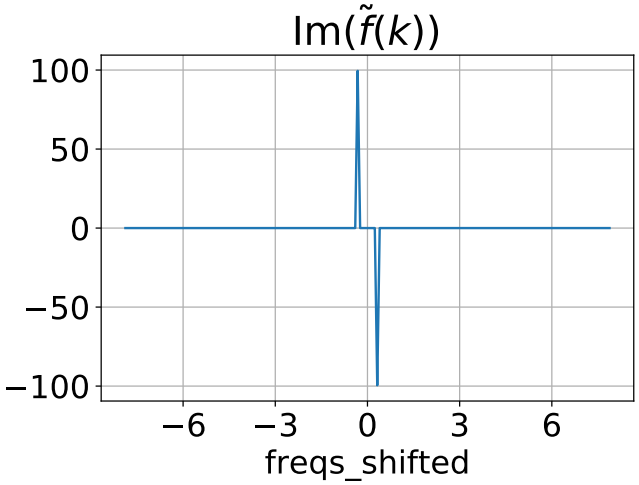
```
Theoretical nyquist frequency:  0.07957747154594787
fftfreqs nyquist frequency:  0.07957747154594787
```

Now, let's plot our FFT with the proper axis. But first...

FFT PITFALL: Origin placement. The FFT algorithm has a weird idea of how the data is ordered.

The astute reader will notice from figure 1 that the ordering of the frequencies is strange. It goes from the zero frequency, to the largest *positive* frequency, to the *smallest* negative frequency, then finally to the *largest* negative frequency. If you're a normal person and like to order your data like $-\infty \to 0 \to \infty$, you have to account for this. As always, numpy to the recue!

In [5]:
```
from numpy.fft import fftshift

'''technically I should shift the input as well, but in this case
we don't have to because of periodicity'''
pure_sine_tilde_shifted = fftshift(fft(pure_sine))
freqs_shifted = fftshift(freqs)
```



ALright! Looking good. Now, let's locate the peaks.

In [6]:
```
peak_ind = np.where(pure_sine_tilde_shifted == np.max(pure_sine_tilde_shifted))[0][0]
theoretical_peak = 2*np.pi/L

print("Theoretical peak: ", theoretical_peak)
print("fft peak: ",np.abs(freqs_shifted[peak_ind]))
```

```
Theoretical peak:  2.0
fft peak:  0.31830988618379147
```

Looks like we have more to do. You guessed it, here comes another...

FFT PITFALL: Why can't everyone just use the same conventions?

The reason that we aren't getting the correct locations for the peaks is that we haven't scaled the horizontal axis properly. The reason we have to scale this axis is because the Fourier convention that we use in cosmology is different than the FFT convention. Here is the FFT convention, from numpy documentation:

$$A_j = \sum_{m=0}^{N-1} a_m \ \exp\left[-i2\pi \frac{mj}{N}\right] \tag{$\star$}$$

Now let's take the cosmology convention, and try to get it in a similar form.

$$\begin{aligned}\tilde{f}(k) &= \int dx\ e^{-ikx} f(x) \\ &\approx \sum_{m=0}^{N-1} (\Delta x) e^{-ikx_m} f(x_m) \\ &= \sum_{m=0}^{N-1} (\Delta x) f(\Delta m) e^{-ikm\Delta}\end{aligned} \tag{$\star\star$}$$

Here we've used

$$x_m = \Delta m$$

since the mth sample of the smooth function will correspond to the sampling resolution Δ times the index m. We can set $(\star) = (\star\star)$, and comparing the exponents:

$$k_j m\Delta = \frac{2\pi mj}{N}$$

and so

$$k_j = 2\pi \frac{j}{N\Delta}$$

notice that $j/N\Delta$ is exactly the expression we would expect for the jth Fourier number, but we have an extra factor of 2π due to the different Fourier conventions. Then, if we account for this:

In [7]:
```
k_array = freqs_shifted*2*np.pi
peak_scaled = k_array[peak_ind]

print("Theoretical peak: ", theoretical_peak)
print("fft peak with scaled axis: ",np.abs(peak_scaled))
```

```
Theoretical peak:  2.0
fft peak with scaled axis:  2.000000000000005
```

And we're done! Exactly as we expected! Actually, one last comment.

FFT PITFALL: On scaling the amplitudes.

In this example I didn't bother with the amplitudes since it's delta functions, but in general you have to correct for amplitudes as well. Notice from ($\star$) that this is a dimensionless expression. When we're doing Fourier transforms of physical things, the integrand has dimensions! Again comparing ($\star$) and ($\star\star$),

$$a_m = (\Delta x) f(\Delta m)$$

So if I have my array `f` corresponding to the discretely sampled function, in order to have a proper normalization I would have to do:

In [8]:
```
f_tilde = fftshift(fft(fftshift(f*delta_x)))
```

where `delta_x` is my real space resolution. **Units matter here!!!**. If I have some volume of length L Mpc, with resolution Δ Mpc and N samples, then my k-space resolution will be

$$\Delta k = \frac{2\pi}{L} = \frac{2\pi}{N\Delta}$$

so it is very important to be watchful of units here.

Some closing thoughts: This was long, but I hope having read this will save you some of the long hours I've spent trying to figure out FFT. You will probably make some mistakes; I've been working with FFT for over a year now, and still learned some new things in writing this. But don't feel bad, it is a *very* counter-intuitive algorthim, riddled with pitfalls. So, best of luck!