Fast Fourier Transform

Carver J. Bierson

July 17, 2017

May processes in nature repeat on some timescale. As scientists, we want to be able to take a data set and ask, "is there a periodic signal in here?" Fourier transforms are one way to answer that question and are used in a wide variety of other applications (including image compression)

1 Some Theory

Let us say we have some data, $h(t)$. This could be a sound recording, the amount of CO_2 in the atmosphere, or temperature[1]. It is true that for any continuous signal, it can be represented as an infinite sum of sine and cosine functions with different amplitudes and wavelengths. This can represented by the Fourier transform

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{-2\pi f i t} df \tag{1}$$

$$= \int_{-\infty}^{\infty} H(\omega) \left(\cos(2\pi f t) - i \sin(2\pi f t)\right) df \tag{2}$$

Here f is the angular frequency. This frequency can be related to the period of a wave via

$$P = \frac{1}{f}. \tag{3}$$

$H(f)$ is a complex function that contains information about the amplitude and phase of each sine/cosine in the summation. If we have our data in the time domain we can solve for this function via

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{2\pi f i t} dt \tag{4}$$

Similar to when we did numerical integration we are going to approximate these integrals by a sum.

$$Hn = \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} \tag{5}$$

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-2\pi i k n/N} \tag{6}$$

[1] We could even consider a spatial signal, $h(x)$, like topography.

Instead of integrating over all time we will also be limited to summing over our data points. In doing so we have some limitations that arise. Because we are approximating our domain by a set of periodic functions, we have to assume our data set repeats before the start and after the end. In reality this is often not the case. Because of this we usually only want to interpret signals in our data that are much smaller than our domain.

There are limits on the highest and lowest frequency that the FFT will return based on your data itself. Lets say our data is evenly spaced in time at a regular interval of Δt. The highest frequency (shortest period) signal that we can resolve is known as the Nyquist critical frequency.

$$f_c = \frac{1}{2\Delta t} \tag{7}$$

This equation is saying that you need at least 2 points per period to know that there is a periodic signal. The lowest frequency signal we can resolve depends on the total time interval we sample for,

$$f_L = \frac{1}{2N\Delta t} \tag{8}$$

This is saying we need at least half of a wavelength within our domain in order to resolve it. The FFT will always return a value with $f = 0$. $H(f = 0)$ is always the mean of our data points.

There is much more subtly in using FFTs that are beyond the scope of this class.

2 FFT in python

Let us make a data set where we know the input frequencies, and try to recover those.

```python
import matplotlib.pyplot as plt
import numpy as np


time= np.linspace(0,100) # array of time values (s)
dt= time[1]-time[0] # find timestep

input_f=[1/50.,1/20. ] # list of input frequencies
input_A=[5,20] # list of input amplitudes

# build signal array
signal = 0*time
for i in range(len(input_f)):
    # I will only do sins but you could also do cosines and add a
    phase shift
    signal+=input_A[i]*np.sin(2*np.pi*input_f[i]*time)

# do fft
H=np.fft.fft(signal)
# calculate frequencies for each H(f)
f=np.fft.fftfreq(len(time), dt)

# Plot FFT power spectrum
plt.figure()

plt.plot(f, np.absolute(H), 'rs', label='fft') # H is complex so we
    will plot the absolute

plt.xlabel('f (1/s)')
plt.ylabel('|H(f)|')

# invert FFT to recover the original signal
yifft=np.fft.ifft(H) # recover our original signal

# Plot original signal and ifft result
plt.figure()

plt.plot(time, signal, 'r-', label='Original')
plt.plot(time, yifft, 'sb', label='Inverted')

plt.xlabel('time (s)')
plt.ylabel('h(t)')
plt.legend()
```

Now lets take a square wave (a box in the time domain), do an FFT, and then slowly reconstruct the square wave a few components at a time.

```

import matplotlib.pyplot as plt
import numpy as np

time= np.linspace(0,100, 100) # array of time values (s)
dt= time[1]-time[0] # find timestep

# Make signal the same size and time, but 0 everywhere
signal = 0*time
# make the signal equal to 1 between 30<t<70
signal[np.logical_and(time<70, time>30)]=1.0

# Do FFT
H=np.fft.fft(signal)
f=np.fft.fftfreq(len(time), dt)


plt.figure()

plt.plot(f, np.absolute(H), 'rs-', label='fft') # H is complex so
    we will plot the absolute

plt.xlabel('f (1/s)')
plt.ylabel('|H(f)|')

# sort H(f) values (and f) by the amplituide of H(f)
# Filp the index so I have the largest values first
sort_index=np.flipud(np.argsort(np.absolute(H)))

H_sort=H[sort_index] # sort the H(f)'s
f_sort=f[sort_index] # sort the f's


plt.figure()

plt.plot(time, signal, 'k-', label='Original')


N_stops=[1,2,3,5,10, 50] # steps to stot at and print
for N in N_stops:
    # To do this I am going to 0 out all values that are not in the
     N largest
    # Because of the symmetry in H(f) I will keep 2*N values
    H_temp=H.copy() # copy ensures we do not change H
    for i in range(len(H)):
        if not(H_temp[i] in H_sort[:2*N]):
            H_temp[i]=0

    yifft=np.fft.ifft(H_temp) # recover our original signal
    plt.plot(time, yifft, '-', label='{:d} feqs'.format(N))

plt.xlabel('time (s)')
plt.ylabel('h(t)')

plt.legend()
```

Note here that as we add more frequency components the area overshooting our box shrinks, but the amplitude of the offshoot does not. No matter how many frequencies you use this overshoot will have approximately the same amplitude. This is known as Gibb's overshoot if you want to lookup more on this.