Discrete Fourier Transform and Fast Fourier Transforms

Sriram Vadlamani

The fourier series

The Discrete Fourier Transforms acronymed as DFT can be derived from the formula of the fourier series.
Let's have a look at the general formula of the Fourier Series:

$$f(x) = a_0 + \sum_{n=1}^{+\infty}(a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L}))$$

Arriving at the Fourier series from the DFT

We will try to prove that the DFT are just the same as the Fourier series general formula. We can start by looking at the general DFT formula
$\forall f_k$ where $k \in \mathbb{N}$ are vectors and $\hat{f}_k$ are the frequencies of the each f_k vectors i.e, the fourier transform.

$$\hat{f}_k = \sum_{j=0}^{n-1} f_j e^{-\iota 2\pi jk/n}$$

From here, let's use $e^{\iota\theta} = \cos(\theta) + \iota \sin(\theta)$

Then, we have:

$$\hat{f}_k = \sum_{j=0}^{n-1}\{(f_j \cos(\frac{2\pi jk}{n})) - (f_j \iota \sin(\frac{2\pi jk}{n}))\}$$

For this proof, let's assume $k = 1$

For $j = 0$ the cosine term equals 1 and the sine term nullifies and hence becomes f_0

simplifying on the same basis, we get:

$$f_0 + f_1(\cos(\tfrac{2\pi}{n}) - \iota \sin(\tfrac{2\pi}{n})) + f_2(\cos(\tfrac{4\pi}{n}) - \iota \sin(\tfrac{4\pi}{n})) + \cdots + f_n(\cos(\tfrac{2(n-1)\pi}{n}) - \iota \sin(\tfrac{2(n-1)\pi}{n}))$$

Now, factoring the cosines and sines i.e, rearranging them, we get:

$$\hat{f}_1 = f_0 + \sum_{j=1}^{n} f_n(\cos(\frac{2\pi j}{n}) - \iota \sin(\frac{2\pi j}{n}))$$

We can see a clear relation with the general fourier expression with $a_0 = f_0$, $a_n = f_n$ and $b_n = -\iota f_n$

DFT matrix

From the general formula of the DFT $\hat{f}_k = \sum_{j=0}^{n-1} f_j e^{-\iota 2\pi jk/n}$ we can say that

$\hat{f}_k$ are multiples of $e^{\frac{-2\pi\iota}{n}}$

So let's say $\omega_n = e^{\frac{-2\pi\iota}{n}}$

let the vector of f_k be

$$\begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_k \end{bmatrix}$$

and let the $\hat{f}_k$ vector be

$$\begin{bmatrix} \hat{f}_1 \\ \hat{f}_2 \\ \vdots \\ \hat{f}_k \end{bmatrix}$$

and the DFT matrix would hence be:

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n & \cdots & \omega_n^{n-1} \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{2(n-1)} \\ 1 & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}$$

Therefore we can say that:
$\hat{f}_k = M \cdot f_k$

The above matrix multiplication gives us the $\hat{f}_k$ vectors which are nothing but the frequency of the of each of the f_k vectors.

Fast Fourier Transforms (FFT)

The FFT is an algorithm that performs the DFT in a more efficient way. Let's assume the number of vectors f_k are a clean power of two. let k = 1024.

in that case the DFT can be written as:

$\hat{f}_k = M_{1024} \cdot f_k$ we can split the M_{1024} matrix as the following:

$$M_{1024} = \begin{bmatrix} I_{512} & D_{512} \\ I_{512} & -D_{512} \end{bmatrix} \cdot \begin{bmatrix} F_{512} & 0 \\ 0 & F_{512} \end{bmatrix} \cdot \begin{bmatrix} f_{even} \\ f_{odd} \end{bmatrix}$$

Where I is the identity matrix and D is the diagonal matrix of the given order.

Now, the F_{512} matrix can be further split into 2 F_{256} and 4 F_{128} and so on until we reach a 2 x 2 matrix, whose value can be stored to compute the other values, this is also known as dynamic programming.

A simple noise reduction program in python

```
# Import statements
import os
import sys
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = [16, 12]
plt.rcParams.update({'font.size': 18})

# create a simple signal with two frequencies
dt = 0.001
t = np.arange(0,1,dt)
f = np.sin(2*np.pi*50*t) + np.sin(2*np.pi*120*t) # some random frequencies
f_clean = f
f = f + 2.5*np.random.randn(len(t))

plt.plot(t,f,color='orange',LineWidth=1.5,label='Noisy')
plt.plot(t,f_clean,color='black',LineWidth=2,label='Clean')
plt.xlim(t[0], t[-1])
plt.legend()

<matplotlib.legend.Legend at 0x7f17636c9d60>

n = len(t)
fhat = np.fft.fft(f,n)  # It's that simple in numpy!
PSD = fhat * np.conj(fhat) / n # PSD = Power spectral density
freq = (1/(dt*n)) * np.arange(n)
L = np.arange(1,np.floor(n/2),dtype='int')
```

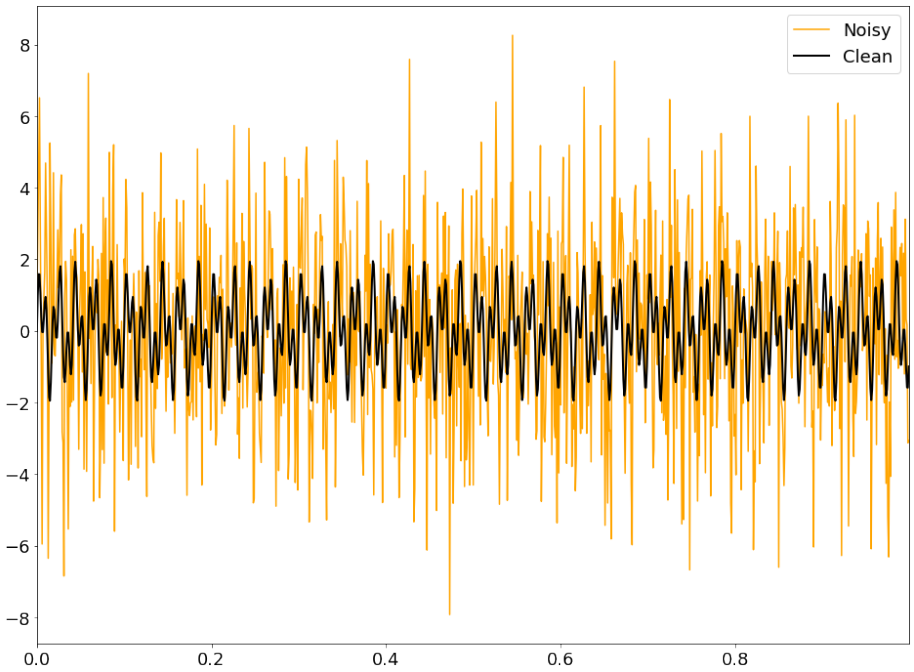

Figure 1: png

```
fig ,axs = plt.subplots(2,1)
plt.sca(axs[0])
plt.plot(t,f,color='c',LineWidth='1.5',label='Noisy')
plt.plot(t,f_clean,color='k',LineWidth='1.5',label='clean')
plt.xlim(t[0],t[-1])
plt.legend()

plt.sca(axs[1])
plt.plot(freq[L],PSD[L],color='c',LineWidth='1.5',label='noisy')
plt.xlim(freq[L[0]], freq[L[-1]])
plt.xlabel('frequency')
plt.ylabel('PSD')
plt.legend()

plt.show()
```

```
/usr/lib/python3.8/site-packages/numpy/core/_asarray.py:85: ComplexWarning: Cast
  return array(a, dtype, copy=False, order=order)
```

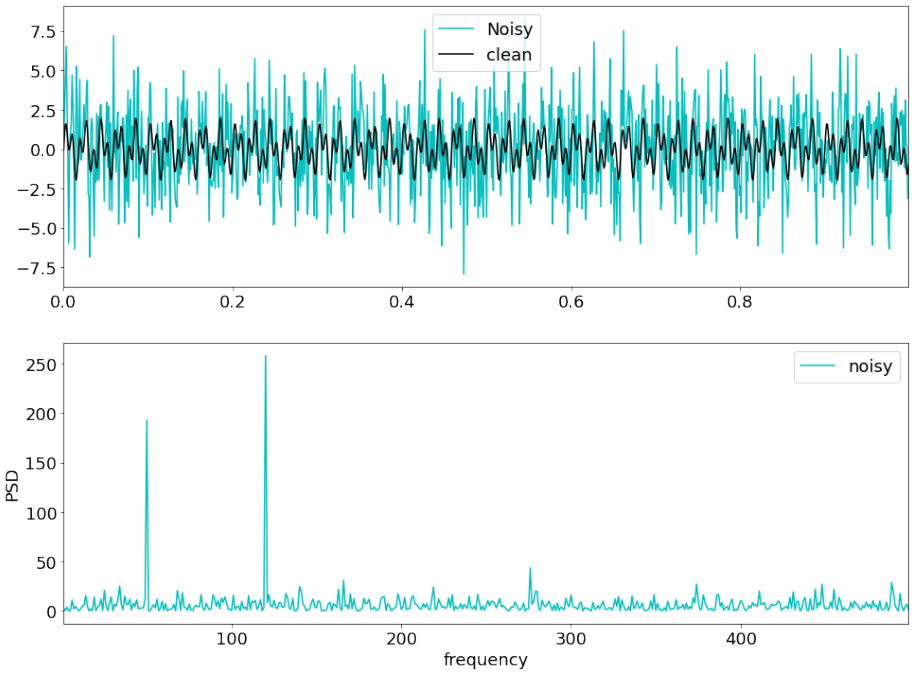

Figure 2: png

'''

We can clearly see two spikes in the above graph that are 50hz and 120hz, which

```
frequencies that we constructed. So we can choose a number, say 100 and any Fou
to give us pure frequencies of 50 and 120hz.
'''

'\nWe can clearly see two spikes in the above graph that are 50hz and 120hz, whi

# Filtering out the noise using the PSD
indices = PSD > 100
PSD_clean = PSD * indices # zeroing out all the PSD below 100
fhat = indices * fhat
f_filter = np.fft.ifft(fhat) # Inverse FFT

# Now plotting the filtered data
fig, axs = plt.subplots(3,1)
plt.sca(axs[0])
plt.plot(t,f_filter,color='c',LineWidth='1.5',label='filtered signal')
plt.xlim(t[0],t[-1])
plt.legend()

plt.sca(axs[1])
plt.plot(freq[L],PSD_clean[L],color='c', LineWidth='1.5',label='PSD filtered')
plt.xlim(freq[L[0]], freq[L[-1]])
plt.legend()
plt.show()

/usr/lib/python3.8/site-packages/numpy/core/_asarray.py:85: ComplexWarning: Cast
  return array(a, dtype, copy=False, order=order)
/usr/lib/python3.8/site-packages/numpy/core/_asarray.py:85: ComplexWarning: Cast
  return array(a, dtype, copy=False, order=order)
```

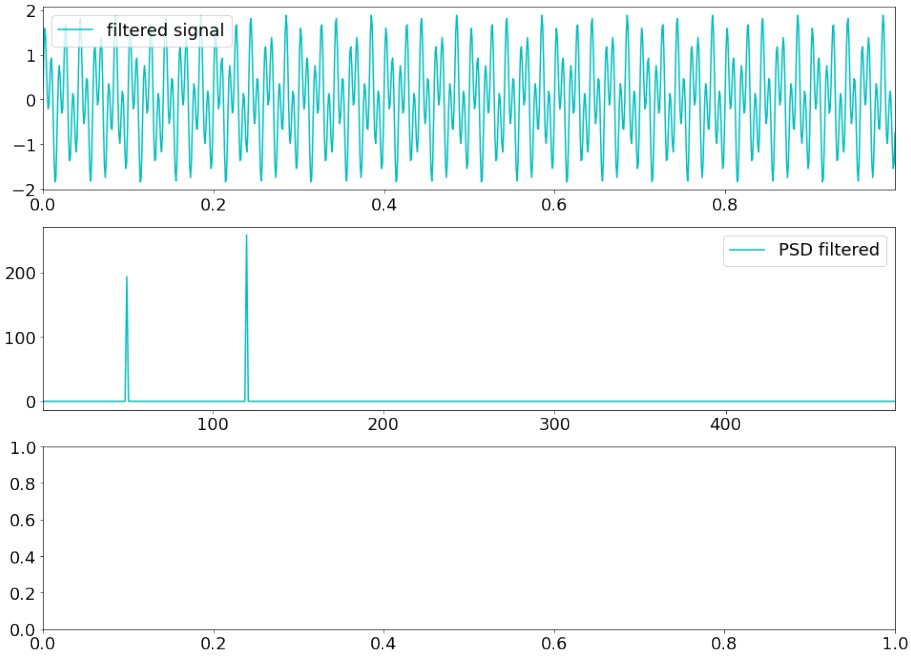

Figure 3: png