

## Final Project: EEG Visualizer – FFT Algorithms

Eddie Guo

March 15, 2020

### Discrete Fourier Transform

The Fourier transform can be written in a few forms with forward and inverse transforms:

#### Hertz Frequency

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-i2\pi ft} dt$$

$$x(t) = \int_{-\infty}^{\infty} X(f)e^{i2\pi ft} df$$

#### Radian Frequency

$$X(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x(t)e^{-i\omega t} dt$$

$$x(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} X(\omega)e^{i\omega t} d\omega$$

The discrete Fourier transform (DFT) is more useful to us as programmers. The naïve method is an  $O(n^2)$  computation which also has a forward and inverse form. A few notes on notation:

- $N$  – the number of time samples.
- $n$  – the current sample considered (0, 1, ...,  $N-1$ ).
- $x_n$  – the value of the signal at time  $n$ .
- $k$  – the current frequency (0 Hz to  $N-1$  Hz).
- $X_k$  – the complex number representing amplitude and phase.

#### Inverse DFT

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i2\pi kn/N}$$

#### Forward DFT

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}$$

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}$$

$$\mathbf{X} = \mathbf{x} \cdot M$$

Where the matrix  $M$  is given by

$$M_{kn} = e^{-i2\pi kn/N}$$

We can compute the DFT using matrix multiplication as shown below.

```

1 import numpy as np
2
3 def dft(x):
4     """Computes the discrete Fourier transform of the 1-D array x"""
5     x = np.asarray(x, dtype=float)
6     n = np.arange(N)
7     N = x.shape[0]
8     k = n.reshape((N, 1))
9     M = np.exp(-2j * np.pi * k * n / N)
10
11     return np.dot(x, M)

```

Listing 1: Naïve implementation of the DFT

## Cooley-Tukey FFT Algorithm

This algorithm exploits the symmetry in the DFT. Let us start by re-expressing the DFT in terms of  $X_{N+k}$ .

$$\begin{aligned}
 X_{N+k} &= \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N} \\
 &= \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi n} \cdot e^{-i2\pi kn/N}
 \end{aligned}$$

We can now use the identity  $e^{2\pi i} = 1 \Rightarrow e^{2\pi in} = (e^{2\pi i})^n = 1^n = 1$ .

$$= \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}$$

However, this is just the original DFT. Thus, symmetry exists such that  $X_{N+k} = X_k$ . We can take this one step further  $X_{k+iN} = X_k$ , for any integer  $i \in \mathbb{R}$ .

$$\begin{aligned}
 X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N} \\
 &= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i2\pi k(2m)/(N/2)} + \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i2\pi k(2m+1)/(N/2)} \\
 &= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i2\pi k(2m)/(N/2)} + e^{-i2\pi k/N} \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i2\pi km/(N/2)}
 \end{aligned}$$

Here, the DFT is split into two terms: one for even-numbered values and one for odd-numbered values. This still gives  $(N/2) * N$  computations giving the same time complexity of  $O(n^2)$ . However, we notice that  $0 \leq k < N$  and  $0 \leq n < M \equiv N/2$ . From the symmetric properties shown, there is only need to perform half the computations for each partition of the original  $N$ -dimensional vector. Thus, the algorithm is converted from  $O(n^2)$  to  $O(M^2)$  where  $M = N/2$ .

It is clear here that we are cooking up a divide-and-conquer approach: partition the vector until the partitioning no longer provides any reasonable time benefits, say  $N \leq 32$ . We can recursively apply all computations such that  $O(N^2)$  becomes  $O(\frac{N}{2} \log_2 N) = O(N \log N)$ .<sup>1</sup>

<sup>1</sup>Note: this algorithm works  $\iff$  the input vector's size is a power of 2.