1 DFT

DFT 其实就是使用有限个数的基 $\{1, e^{ix}, ..., e^{i(N-1)x}\}$ 以及有限个数的插值节点 $\{x_0, x_1, x_{N-1}\}, x_j = \frac{2\pi}{N}j$ 去逼近函数 $f(x)$:

$$f(x) = \sum_{j=0}^{N-1} \hat{f}_j e^{-ijx} \tag{1}$$

很容易证明，这些基底在离散情形下也是正交的:

$$\langle e^{ikx}, e^{ilx} \rangle = \sum_{j=0}^{N-1} e^{i(k-l)\frac{2\pi}{N}j} = \begin{cases} 0 & k = l \\ N & k \neq l \end{cases} \tag{2}$$

那么根据正交性，结合公式 (1)，就能得到

$$\hat{f}_k = \frac{\langle f(x), e^{ikx} \rangle}{\langle e^{ikx}, e^{ikx} \rangle} = \frac{1}{N} \sum_{j=0}^{N-1} f_j e^{-i\frac{2\pi}{N}j} \tag{3}$$

其实前面的系数 $\frac{1}{N}$ 都是无关紧要的，所以为了下面讨论的方便，我们直接写为

$$\hat{f}_k = \sum_{j=0}^{N-1} f_j e^{-i\frac{2\pi}{N}j} \tag{4}$$

2 FFT

从上一节可以知道，DFT 其实就是矩阵与向量的乘积运算，算法复杂度为 $O(n^2)$。能不能做到更好呢？事实上是可以的，启发就来源于乘法的结合律 $ab + bc = a(b + c)$，等式左边进行了 2 次乘法操作，而等式右边其实只进行了 1 次乘法操作，于是计算的复杂度得以降低。所以，FFT 就出现了，可以从后面的讲解中看到，其实 FFT 就是避免了大量的重复计算，然后把算法的复杂度降低到了 $O(n \log n)$。

首先我们考虑要对一个长度为 $2N$ 的序列做 DFT，那么有:

$$\hat{f}_k = \sum_{j=0}^{2N-1} f_j e^{-i\frac{2\pi}{2N}j} \tag{5}$$

记 $\bar{w} = e^{-i\frac{2\pi}{2N}}$，并且把下标为偶数和奇数的部分分割开来，那么有:

$$\hat{f}_k = \sum_{j=0}^{N-1} f_{2j} \bar{w}^{k(2j)} + \sum_{j=0}^{N-1} f_{2j+1} \bar{w}^{k(2j+1)} \tag{6}$$

我们已经把奇数和偶数部分拆分开了，现在希望这两部分分别做一个长度为 N 的 DFT。怎么做呢？我们记 $\bar{W}=e^{-\frac{2\pi}{N}}$，那么容易推导出 $\bar{W}=\bar{w}^2$，于是上面的式子可以进一步写为

$$\hat{f}_k=\sum_{j=0}^{N-1} f_{2j}\bar{W}^{kj}+\bar{w}^k\sum_{j=0}^{N-1} f_{2j+1}\bar{W}^{kj} \tag{7}$$

容易观察出 $\sum_{j=0}^{N-1} f_{2j}\bar{W}^{kj}$ 和 $\sum_{j=0}^{N-1} f_{2j+1}\bar{W}^{kj}$ 就是分别对序列 $\{f_0,f_2,...,f_{2N-2}\}$ 和 $\{f_1,f_3,...,f_{2N-1}\}$ 做 DFT，于是可以记为：

$$\hat{f}_k=F(f_0,f_2,...,f_{2N-2})_k+\bar{w}^kF(f_1,f_3,...,f_{2N-1})_k \tag{8}$$

也就是说，我们可以从两个长度为 N 的子序列的 DFT 结果求得长度为 $2N$ 的序列的 DFT 结果。有人会问了，其实这样做了以后复杂度还是没有降低啊？的确，我们还要推导一下 $\hat{f}_{k+N}$ 与这两个子序列的关系，下面给出推导：

$$\begin{aligned}
\hat{f}_{k+N} &= \sum_{j=0}^{N-1} f_{2j}\bar{W}^{(k+N)j}+\bar{w}^{(k+N)}\sum_{j=0}^{N-1} f_{2j+1}\bar{W}^{(k+N)j} \\
&= \sum_{j=0}^{N-1} f_{2j}\bar{W}^{kj}+\bar{w}^{(k+N)}\sum_{j=0}^{N-1} f_{2j+1}\bar{W}^{kj} \quad (\bar{W}^{Nj}=1) \\
&= \sum_{j=0}^{N-1} f_{2j}\bar{W}^{kj}-\bar{w}^{k}\sum_{j=0}^{N-1} f_{2j+1}\bar{W}^{kj} \quad (\bar{w}^{N}=-1)
\end{aligned} \tag{9}$$

现在有点意思了，我们发现其实 f_k 和 f_{k+N} 就只差了一个加减号而已。这样的话，我们做两个序列长度为 N 的 DFT 就有意义了：原来求一个 $\hat{f}_k$ 的时候大概要消耗 $2N$ 的计算量，但是经过公式的变换以后求两个 $\hat{f}_k$ 才消耗了 $2N$ 的计算量 (每个序列长度为 N 的计算消耗 N)。哈哈，是不是很神奇！

当然，上面的分析只是把 $2N$ 的序列分成两个 N 的序列，我们可以接着这样做下去，$N\rightarrow\frac{N}{2},...\ 4\rightarrow 2$。于是就有了著名的快速傅里叶变换 FFT。现在我们来分析一下 FFT 的时间复杂度：

$$T(N)=2T(N/2)+O(N) \tag{10}$$

$2T(N/2)$ 为计算两个序列长度为 $N/2$ 的 FFT 时间，得到了这两个序列的 FFT 以后，我们可以重构序列长度为 N 的 FFT，需花费 $O(N)$ 时间。稍微学习过算法的同学都会知道，上面的递推关系给出的时间复杂度为

$$T(N)=O(N\log N) \tag{11}$$

3 Implementation of FFT

其实如果序列长度是 2 次幂的话，每次分割都会很顺利，这时的 FFT 时间效率是最高的。自己实现了这种情形的 FFT

```python
import numpy as np


def my_fft(y):
    """
    Recursive implementation of FFT.

    :param y: np.array, with length equal to `2^L`.

    :return
    yh: np.array, same length as `y`.
    """
    n = len(y)
    N = n // 2
    if n == 2:
        y0, y1 = y
        yh0 = y0 + y1
        yh1 = y0 - y1
        return np.array([yh0, yh1])
    yeven = my_fft(y[::2])
    yodd = my_fft(y[1::2])
    wk = np.exp(-np.arange(N)*np.pi/N * 1j)
    yh = np.zeros(n, dtype=np.complex128)
    yh[:n//2] = yeven + wk * yodd
    yh[n//2:] = yeven - wk * yodd
    return yh
```