The Fast Fourier Transform (FFT) algorithm

We consider the case of the Finite Fourier Transform (discrete, periodic) when \( N = 2^D \) is a power of 2. The definition of Fourier transform is

\[
F_{D}f(j) = \sum_{k=0}^{2^D-1} f(k)\exp\left(-2\pi i \frac{j k}{2^D}\right), \quad j = 0, ..., 2^D - 1,
\]

where we have highlighted the dependence of \( F = F_{D} \) on the parameter \( D \).

The complexity of an algorithm for computing \( F_{D}f \), which only makes use of the four operations, will be defined as the (approximate) number of multiplications used. The underlying assumption is that sums are relatively inexpensive in terms of computational time.

When we write that the complexity is \( O(N) \), we mean that there is a constant \( C \) independent of \( N \) such that no more that \( C \cdot N \) operations have to be carried out if we have \( N \) data to process (in the case of the FT, the data are the \( N \) values \( f(j), j = 0, ..., N - 1 \)).

If our algorithm merely consists of applying the definition of \( F_{D}f \), then we have a total number of \( 2^D \times 2^D = N^2 \) multiplications, as for each \( j \) among \( 2^D \) possible ones, we have \( 2^D \) multiplications in the sum.

We will see that it is in fact possible to do the calculation with complexity \( O(N \log N) \).

Here is the basic recursion step. Let \( f_{e}(j) = f(2j) \) and \( f_{o}(j) = f(2j + 1), \quad j = 0, ..., 2^{D-1} - 1 \) (\( e \) and \( o \) refer to \( \text{even} \) and \( \text{odd} \), respectively). Then we can split the sum defining \( F_{D}f \) is even and odd summands:

\[
F_{D}f(j) = \sum_{k=0}^{2^{D-1}-1} f(2k)\exp\left(-2\pi i \frac{2k j}{2^D}\right) + \sum_{l=0}^{2^{D-1}-1} f(2l + 1)\exp\left(-2\pi i \frac{j (2l+1)}{2^D}\right)
\]

\[
= \sum_{k=0}^{2^{D-1}-1} f_{e}(k)\exp\left(-2\pi i \frac{jk}{2^{D-1}}\right) + \exp\left(-2\pi i \frac{j}{2^D}\right) \sum_{l=0}^{2^{D-1}-1} f_{o}(l)\exp\left(-2\pi i \frac{j l}{2^{D-1}}\right)
\]

\[
= F_{D-1}f_{e}(j) + \exp\left(-2\pi i \frac{j}{2^D}\right)F_{D-1}f_{o}(j).
\]

Let \( C(D) \) be the computational complexity for \( N = 2^D \). Observe that \( f_{e} \) and \( f_{o} \) have both \( 2^{D-1} \) entries, and that hence \( F_{D-1}f_{e}(j), F_{D-1}f_{o}(j) \), need only be computed for \( 2^{D-1} \) values of \( j: \ j = 0, ..., 2^{D-1} - 1 \). For the other values of \( j \), we can use periodicity: \( F_{D-1}f_{o}(j + 2^{D-1}) = F_{D-1}f_{o}(j) \), and the same holds for \( f_{e} \). Thus, we need \( 2C(D-1) \) multiplications to compute \( F_{D-1}f_{e}(j), F_{D-1}f_{o}(j) \).

On the other hand, in the last line of the calculations, we have to perform the multiplication times \( \exp\left(-2\pi i \frac{j}{2^D}\right) \), and this has to be done for \( j = 0, ..., 2^D - 1 \).

Overall, we have to perform: \( C(D) = 2C(D-1) + 2^D \) multiplications.
We can proceed the same way to reduce calculations from \(\mathcal{F}_{D-1}\) to \(\mathcal{F}_{D-2}\), and so on. We obtain:

\[
C(D) = 2C(D - 1) + 2^D
\]

\[
= 2[C(D - 2) + 2^{D-1}] + 2^D
\]

\[
= 2C(D - 2) + 2 \times 2^D
\]

\[
= \ldots
\]

\[
= 2^{m-1}C(D - m) + m \times 2^D
\]

\[
= 2^{D-1}C(0) + D \times 2^D
\]

\[
= D \times 2^D
\]

since \(C(0) = 0\)

\[
= N \log_2 N.
\]

Recursion is applied to \(f_e\) and \(f_o\), leading to \(f_{ee}, f_{eo}, f_{oe}, f_{oo},\) and so on. It is easy to make this into an explicit algorithm (exercise for engineers!). Such algorithm can be interpreted as writing the matrix \(\mathcal{F}_D\) representing the Fourier transform in \(\mathbb{C}^{2^D}\) as the product of “sparese matrices” (matrices in which a large number on entries vanish).

We will work out in detail the case \(D = 2\).

\[
\mathcal{F}_2 f(j) = \begin{pmatrix}
1 & 1 & 1 & 1 \\
1 & -1 & i & -i \\
1 & -1 & -1 & 1 \\
i & -1 & -1 & 1
\end{pmatrix}\begin{pmatrix}
f(0) \\
f(1) \\
f(2) \\
f(3)
\end{pmatrix}
\]

\[
= \begin{cases}
f(0) + f(1) + f(2) + f(3) & \text{if } j = 0 \\
f(0) + f(1)i - f(2) - f(3)i & \text{if } j = 1 \\
f(0) - f(1) + f(2) - f(3) & \text{if } j = 2 \\
f(0) - f(1)i - f(2) + f(3)i & \text{if } j = 3
\end{cases}
\]

\[
= \begin{pmatrix}
1010 \\
010i \\
10-10 \\
010-i
\end{pmatrix}\begin{pmatrix}
1010 \\
10-10 \\
0101 \\
010-1
\end{pmatrix}\begin{pmatrix}
f(0) \\
f(1) \\
f(2) \\
f(3)
\end{pmatrix}
\]

The interested student might want to work out the \(D = 3\) case, to see the 3 matrices factoring \(\mathcal{F}_3\) becoming more and more spare.

It is not known whether there is an algorithm for computing \(\mathcal{F}_D f\), and having a smaller computational complexity than the FFT.