Shiraz University

Cooley-Tukey FFT Algorithm

Seyed Mahdi Mahdavi Mortazavi[1]

[1] *Department of Electrical and Computer Engineering, Shiraz University*

Abstract

The Cooley-Tukey algorithm is a widely used method for efficiently computing the Fast Fourier Transform, a crucial tool in signal processing and data analysis. This review article, explores the development and workings of the Cooley-Tukey algorithm, providing a detailed explanation of its recursive divide-and-conquer approach that enable it to reduce the computation time of the FFT. Additionally, a practical test is provided to understand the effects of this optimization.

Keywords: DFT, FFT, Cooley-Tukey Algorithm, Radix-2 DIT, Divide-And-Conquer.

1. Introduction

Often in Digital Signal Processing applications, it is necessary to estimate the signal spectrum or vice versa, knowing the signal spectrum calculate the signal itself. For this purpose, digital technology uses the Discrete Fourier Transform (DFT) [1].

DFT operation is useful in many fields, but computing it directly from the definition is often too slow to be practical. A Fast Fourier Transform (FFT) algorithm, rapidly computes such transformations by factorizing the DFT matrix into a product of sparse factors, many of which are zero [2].

By far, the most commonly used FFT is the Cooley–Tukey algorithm. This is a divide-and-conquer algorithm that recursively breaks down a DFT of any composite size $n = n_1 n_2$ into many smaller DFTs of sizes n_1 and n_2, along with $O(n)$ multiplications [3]. As a result, it reduces the complexity of computing the DFT from $O(n^2)$, which arises if one simply applies the definition of DFT, to $O(nlogn)$, where n is the data size. [4].

2. Cooley-Tukey Algorithm

The Cooley–Tukey algorithm, named after J. W. Cooley and John Tukey, is the most common FFT algorithm. It transforms the DFT of an arbitrary composite size $N = N_1 N_2$ in terms of N1 smaller DFTs of sizes $N2$, recursively, to reduce the computation time to $O(NlogN)$ for highly composite N.

2.1 The radix-2 DIT case

A radix-2 decimation-in-time (DIT) FFT is the simplest and most common form of the Cooley–Tukey algorithm, although highly optimized Cooley–Tukey implementations typically use other forms of the algorithm. Radix-2 DIT divides a DFT of size N into two interleaved DFTs of size $N/2$ with each recursive stage. The DFT is defined by the formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}$$

Where k is an integer ranging from 0 to $N-1$.

Radix-2 DIT computes the DFTs of the even-indexed inputs ($x_{2m} = x_0, x_2, \ldots, x_{N-2}$) and of the odd-indexed inputs ($x_{2m+1} = x_1, x_3, \ldots, x_{N-1}$), and then combines those two results to produce the DFT of the whole sequence. This idea can then be performed recursively to reduce the overall runtime to $O(NlogN)$. This simplified form assumes that N is a power of two.

The radix-2 DIT algorithm rearranges the DFT of the function x_n into two parts: a sum over the even-numbered indices $n = 2m$ and a sum over the odd-numbered indices $n = 2m + 1$:

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k}$$

One can factor a common multiplier $e^{-\frac{2\pi i}{N}k}$ out of the second sum, as shown in the equation below. It is then clear that the two sums are the DFT of the even-indexed part x_{2m} and the DFT of odd-indexed part x_{2m+1} of the function x_n. Denoting the DFT of the Even-indexed inputs x_{2m} by E_k and for the Odd-indexed inputs x_{2m+1} by O_k and we obtain:

$$X_k = \underbrace{\sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2} mk}}_{\text{DFT of even-indexed part of } x_n} + e^{-\frac{2\pi i}{N}k} \underbrace{\sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2} mk}}_{\text{DFT of odd-indexed part of } x_n} = E_k + e^{-\frac{2\pi i}{N}k} O_k$$

Using periodicity of the complex exponential, $X_{k+\frac{N}{2}}$ is also obtained from E_k and O_k:

$$\begin{aligned}
X_{k+\frac{N}{2}} &= \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2} m(k+\frac{N}{2})} + e^{-\frac{2\pi i}{N}(k+\frac{N}{2})} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2} m(k+\frac{N}{2})} \\
&= \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2} mk} e^{-2\pi mi} + e^{-\frac{2\pi i}{N}k} e^{-\pi i} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2} mk} e^{-2\pi mi} \\
&= \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2} mk} - e^{-\frac{2\pi i}{N}k} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2} mk} \\
&= E_k - e^{-\frac{2\pi i}{N}k} O_k
\end{aligned}$$

We can rewrite X_k and $X_{k+\frac{N}{2}}$ as [5]:

$$\begin{aligned} X_k &= E_k + e^{-\frac{2\pi i}{N}k} O_k \\ X_{k+\frac{N}{2}} &= E_k - e^{-\frac{2\pi i}{N}k} O_k \end{aligned}$$

2.2 Comparison of calculation speed

Table 1 presents a comparison of computational speeds between the FFT using the Cooley-Tukey algorithm and the DFT. This comparison highlights the superior computational efficiency of FFT over DFT.

Table 1: Comparing the speed of FFT and DFT calculations (approximate) [6]

Input size	$N = 2^{10}$	$N = 2^{15}$	$N = 2^{20}$
DFT	1.553 seconds	26 minutes	5 hours
FFT	0.055 seconds	1.564 seconds	52.218 seconds

3. Conclusion

Expressing the DFT of length N recursively in terms of two DFTs of size $N/2$, is the core of the radix-2 DIT Fast Fourier Transform. The algorithm gains its speed by re-using the results of intermediate computations to compute multiple DFT outputs. Note that final outputs are obtained by a plus and minus combination of E_k and $O_k \exp(-2\pi ik/N)$, which is simply a size-2 DFT (sometimes called a butterfly in this context); when this is generalized to larger radices, the size-2 DFT is replaced by a larger DFT. This process is an example of the general technique of divide-and-conquer algorithms [5].

References

[1] https://www.dsp-weimich.com/digital-signal-processing/fast-fourier-transform-fft-and-their-c-realizations-using-the-octave-gnu-tool/

[2] Van Loan, Charles (1992). Computational Frameworks for the Fast Fourier Transform.

[3] Gentleman, W. Morven; Sande, G. (1966). "Fast Fourier transforms—for fun and profit".

[4] https://en.wikipedia.org/wiki/Fast_Fourier_transform

[5] https://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm

[6] https://github.com/theMHD-120/fft