CS260 PROJECT FINAL REPORT: COMPARISON OF COOLEY–TUKEY ALGORITHM AND RADIX-4 FFT ALGORITHM FOR FFT

Hao Liu(185896) Mohamed Bouaziz(184732) Tongzhou Gu(186085) Juyi Lin(187176)
Jinjie Mai(179794)
{hao.liu, mohamed.bouaziz, tongzhou.gu, juyi.lin, jinjie.mai}@kaust.edu.sa

1 Introduction

Fourier Transform (FT) is one of the most well-known mathematical transforms that convert a continuous input function from the time domain into the frequency domain. From the perspective of an engineer, with Fourier Transform, a function of analog signal strength against time could be converted into a function against frequency, or vice versa, which means any signal could be seen as a combination of an infinite number of sine waves at different frequencies. For a discrete input function, such as a function describing a digital signal waveform, there is a "discrete version" of the Fourier Transform named Discrete Fourier Transform (DFT), and a discrete function could be regarded as the sum of a finite number of sine waves, depending on the number of the sampling points. In the practical use of Discrete Fourier Transform, we also have its efficient implementation known as Fast Fourier Transform (FFT).

The industry has widely adopted fast Fourier Transform as one of the most commonly-used algorithms. It has demonstrated its massive value in signal processing, especially for wireless communication, as well as digital image and audio processing. For example, Orthogonal frequency division multiplexing (OFDM)[1] and its derived method Orthogonal Frequency Division Multiple Access (OFDMA)[2] is used by modern wireless communication technology such as LTE Mobile Network[3] and Wi-Fi 6 Wireless LAN[4] to improve bandwidth utilization and achieve a higher transmission rate. This method heavily utilizes FFT to modulate and demodulate the signal. Another famous application is image and audio compression. Derived from FFT, Discrete Cosine Transform (DCT) is another transform that can deliver a higher compression ratio than DFT when used by compression algorithms. One Fast DCT implementation is based on FFT with additional pre- and post-processing[5]. So far, many mainstream image and audio compression formats, including MP3 for audio, JPEG for images, and MPEG for video, are DCT-based[6].

Among the FFT implementations, Cooley-Tukey FFT[7] is considered the standard implementation, and there are also several variants, including Radix-4 FFT[8], Rader's FFT[9], Winograd FFT[10], Quantum Fourier transform (QFT)[11], and Split-Radix FFT[**?**]. This project aims to analyze and re-implement Cooley-Tukey FFT and Radix-4 FFT algorithms. Then we are devoted to making a detailed comparison between them through theoretical and experimental analysis.

2 Discrete Fourier Transform (DFT)

2.1 Mathematical Problem Formulation

In this section, we give a mathematical formulation of discrete Fourier transform (DFT) [12] and explain the naive DFT algorithm in detail. For brevity, we consider the one-dimensional case of DFT in the following.

Suppose that we have a discrete sequence of length N $(x(0), x(1), x(2), ...x(N-1))$.

DFT is a function $\mathcal{F}$, which will transform the given sequence x_n to a new sequence of length N,

$(X(0), X(1), X(2), ...X(N-1))$**.**

Formally, DFT function $\mathcal{F}$ is defined by:

$$\mathcal{F}[x(.)](k) = \frac{1}{N}\sum_{n=0}^{N-1} x(n) \cdot e^{-i2\pi\frac{kn}{N}}, \quad k \in \{0, \ldots, N-1\}, \tag{1}$$

where i is the imaginary entity whose square is -1.

The inverse DFT $\mathcal{F}^{-1}$, alias Inverse Discrete Fourier Transform (IDFT), is the function that can reconstruct x_n from X_n with no loss of information. Similarly, we can write this process as follows:

$$x(k) = \sum_{n=0}^{N-1} X(n) \cdot e^{i2\pi\frac{kn}{N}}, \quad k \in \{0, \ldots, N-1\}, \tag{2}$$

2.2 Time Complexity Analysis of Naive DFT

Algorithm 1 A naive implementation for DFT

Data: Input discrete sequence of length N: $(x(.)) = x(0), x(1), x(2), ...x(N-1)$
Result: Output sequence of length N: $(X(.)) = X(0), X(1), X(2), ...X(N-1)$
Init: Initialize $X_n = X(0), X(1), X(2), ...X(N-1)$ as zeros

```
for k ← 0 to N − 1 do
    for n ← 0 to N − 1 do
        X(k) = X(k) + x(n) · e^{−i2π kn/N}
    end
end
return X_n
```

The algorithm procedure can be found in Alg. 1. Obviously, we have two loops in length N in our implementation. For every output element, we need to traverse every element in the input array. So the time complexity of DFT in 1D case will be:

$$N \times N = N^2 = O(N^2)$$

2.3 DFT Matrix

Considering Eq. 1, if we denote the last term as:

$$W_N = e^{\frac{-i2\pi}{N}} = \cos(\frac{2\pi}{N}) - i\sin(\frac{2\pi}{N}) \tag{3}$$

$$W_N^{kn} = e^{-i2\pi\frac{kn}{N}} = \cos(2\pi\frac{kn}{N}) - i\sin(2\pi\frac{kn}{N}) \tag{4}$$

We can rewrite Eq. 1 using Eq. 4 as:

$$X_k = \frac{1}{N}\sum_{n=0}^{N-1} x_n \cdot W_N^{kn}, \quad n \in \mathbb{Z}, \tag{5}$$

Using the notation in Eq. 4, we can explicitly write DFT as a linear transformation. For instance, if we have $(x(1), \ldots, x(N-1))$ with $N = 5$, then we can formulate the DFT as follows:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ X(4) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & W_5 & W_5^2 & W_5^3 & W_5^4 \\ 1 & W_5^2 & W_5^4 & W_5^6 & W_5^8 \\ 1 & W_5^3 & W_5^6 & W_5^9 & W_5^{12} \\ 1 & W_5^4 & W_5^8 & W_5^{12} & W_5^{16} \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \end{bmatrix} \tag{6}$$

Another interesting fact is that $g(k) = W_N^{nk}$ is a periodic function with period N. We will find writing DFT in matrix form is more intuitive to lead us to go from DFT to FFT.

3 DFT Implementation and Validation

For a fair comparison with our FFT implementation, we implement Alg. 1 with C++, the same programming language we will use for our FFT.

After that, we think it's important to verify the correctness of our implementation, otherwise, our implementation and complexity analysis rely on DFT as a baseline may go wrong.

For this, we manually design a periodic function (Fig. 1):

$$f(t) = 10\sin(2\pi 10t) + 9\sin(2\pi 9t) + ... + \sin(2\pi t)$$

and sample some discrete data points from it. Then we perform DFT to these samples by our implementation and official Numpy [13]. We verify the correctness by comparing the results qualitatively and quantitatively.

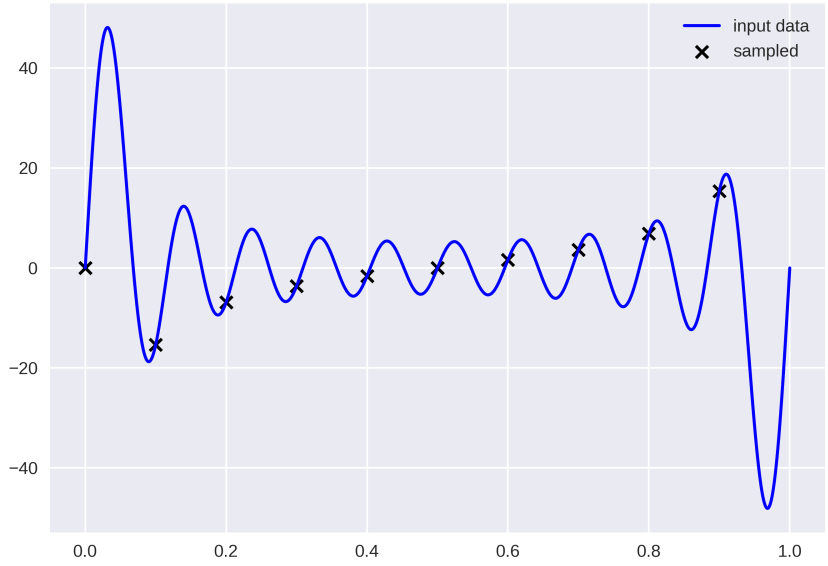

Figure 1: An example of simple synthetic signal for sampling

From the visualization in Fig. 1, we can find that the DFT outputs are very close. Moreover, we increase the data size, try various data with gaussian noise and variable density (as shown in Fig. 3), and compute the L2 error between ours and Numpy's. As shown in Table 1, although the errors are increasing with the data size, the absolute error is still in a relatively small order of magnitude, which we think is acceptable. We validate this more concretely through the visualization in Fig. 2 where we can see that the amplitudes of our implementation and the official one have the same shape and values.

Data Size	L2 Norm
10	7.47×10^{-15}
100	5.88×10^{-12}
1,000	1.98×10^{-9}
10,000	5.73×10^{-7}
100,000	2.37×10^{-4}

Table 1: L2 Error between the implemented and the official averaged on three runs

Since we have verified our DFT implementation qualitatively and quantitatively, in the following experiments and implementation, to perform a fair comparison, we will use our DFT as the baseline for correctness validation and complexity analysis. We will use the same mechanism to generate test input data and validate the implementation correctness.

(a) DFT result of our DFT

(b) DFT result from Numpy

Figure 2: Visualization of DFT results in amplitudes

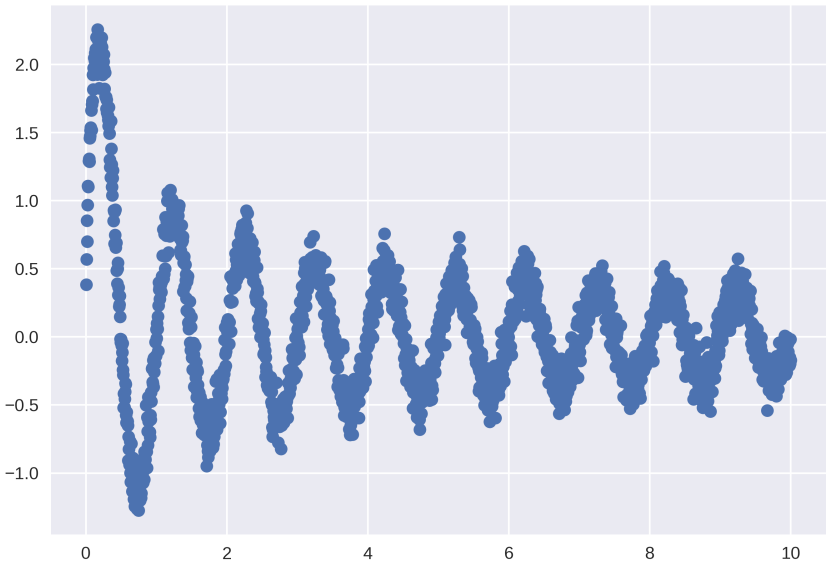

Figure 3: An example of large-size data with random noise for correctness verification

4 Description of Algorithms

4.1 Algorithms 1: Cooley–Tukey algorithm

4.1.1 Description

This famous algorithm was invented by Cooley and Tukey [7], engineers at the IBM research center in the early 1960s. It has had a considerable impact on the development of digital signal processing applications due to its efficiency [14]. A discrete Fourier transform calculation is a product of a matrix and a vector. It, therefore, requires N^2 multiplications/additions of complex numbers (N is the order of the matrix).

Assuming that a computer performs 10^9 operations per second, a transform calculation on a signal of $N = 10^3$ samples will require 10^{-3}s. A calculation on an image of size $N \times N = 10^6$ will require N^4 or 10^{12} operations which costs fifteen minutes. If we consider processing data in a three-dimensional domain (on vectors of size $N \times N \times N$), we would need N^6 or 10^{18} operations, which require a few decades.

The fast Fourier transform considerably reduces the number of operations: instead of performing N^2 operations, it will suffice to perform $N \log_2 N$. In the three previous examples, we will have to perform 10^4, 2×10^7 and 3×10^{10} operations which will require respectively $10^{-5}s$, $2 \times 10^{-2}s$ and $30s$. To explain this algorithm, we will use recursion by showing that the computation of a Fourier transform of size N comes down to the computation of two Fourier transforms of size $N/2$ followed by $N/2$ multiplications.

We want to calculate for $k = 0, \cdots, N-1$

$$X(k) = \sum_{t=0}^{N-1} x(t) \exp(-2\pi j \frac{k.t}{N}) \tag{7}$$

We pose $t = 2n$ if t is even and $t = 2n+1$ if t is odd. $X(k)$ is then written by posing $M = N/2$

$$X(k) = \sum_{n=0}^{M-1} x(2n) \exp(-2\pi j \frac{k.2n}{N}) + \sum_{n=0}^{M-1} x(2n+1) \exp(-2\pi j \frac{k.(2n+1)}{N}) \tag{8}$$

Let us name the sequences.

$$\begin{gathered} t = 0, \cdots, 2M-1 : x_{2M}(t) = x(t) \\ n = 0, \cdots, M-1 : x_N^{even}(n) = x(2n) \\ n = 0, \cdots, M-1 : x_N^{odd}(n) = x(2n+1) \\ k = 0, \cdots, 2M-1 : X_{2M}(k) = X(k) \end{gathered}$$

With these notations, we get

$$X_{2M}(k) = \sum_{n=0}^{M-1} x_M^{even}(n) \exp\left(-2\pi j \frac{k \cdot n}{M}\right) + \sum_{n=0}^{M-1} x_M^{odd}(n) \exp\left(-2\pi j \frac{k \cdot n}{M}\right) \exp\left(-2\pi j \frac{k}{2M}\right) \tag{9}$$

In the second summation of the right-hand side of the equation, the factor $exp(-2\pi j \frac{k}{2M})$ does not depend on n. We, therefore, have the following equation for $k = 0, \cdots, 2M-1$.

$$X_{2M}(k) = \left[\sum_{n=0}^{M-1} x_M^{even}(n) \exp\left(-2\pi j \frac{k \cdot n}{M}\right)\right] + \exp\left(-2\pi j \frac{k}{2M}\right) \left[\sum_{n=0}^{M-1} x_M^{odd}(n) \exp\left(-2\pi j \frac{k \cdot n}{M}\right)\right] \tag{10}$$

If $0 \leq k \leq M-1$, we recognize in the two expressions between square brackets the discrete Fourier transforms of the sequences of the even-numbered samples $x_N^{even}(n)$ and the odd-numbered samples $x_N^{odd}(n)$ which we call $X_N^{even}(k)$ and $X_N^{odd}(k)$ respectively.

Therefore, for $k = 0, \cdots, M-1$

$$X_{2M}(k) = X_N^{even}(k) + \exp(-\pi j \frac{k}{M}) X_N^{odd}(k) \tag{11}$$

When $M \leq k \leq 2M-1$, we can write

$$k = \ell + M \tag{12}$$

and notice that

$$\exp(-\pi j \frac{k}{M}) = -\exp(-\pi j \frac{\ell}{M}) \tag{13}$$

Therefore, for $\ell = 0, \cdots, M-1$, the following equation is used

$$X_{2M}(\ell+M) = \left[\sum_{n=0}^{M-1} X_N^{even}(n) \exp(-2\pi j \frac{(\ell+M).n}{M})\right] + \exp\left(-2\pi j \frac{\ell+M}{2M}\right) \left[\sum_{n=0}^{M-1} X_N^{odd}(n) \exp(-2\pi j \frac{(\ell+M).n}{M})\right] \tag{14}$$

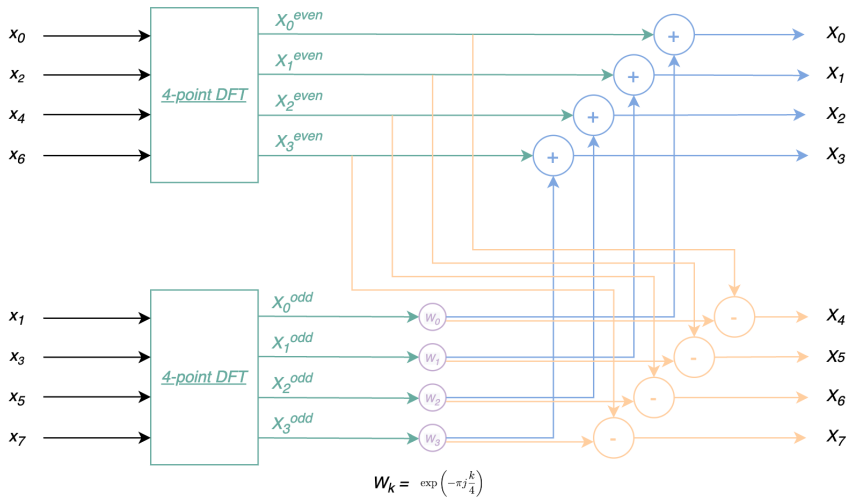

Figure 4: Cooley-Tukey functional diagram

and noting that

$$\exp(-2\pi j\frac{(\ell+M).n}{M}) = \exp(-2\pi j\frac{\ell.n}{M}) \tag{15}$$

we have an analogous writing for $\ell = 0, \cdots, M-1$

$$X_{2M}(\ell+M) = X_N^{even}(\ell) - exp(-\pi j\frac{\ell}{M})X_N^{odd}(\ell) \tag{16}$$

We can change the name of the variable ℓ to k and combine the two equations $X_{2M}(k)$ and $X_{2M}(k+M)$ for $k = 0, \cdots, M-1$

$$X_{2M}(k) = X_N^{even}(k) + \exp\left(-\pi j\frac{k}{M}\right)X_N^{odd}(k) \tag{17}$$

$$X_{2M}(k+M) = X_N^{even}(k) - \exp\left(-\pi j\frac{k}{M}\right)X_N^{odd}(k) \tag{18}$$

4.1.2 Functionality principal

It is clear that the calculation of the DFT components $X(k)$ in Eq. 7 for $k = 0, \cdots, 2M-1$ (in other terms $k = 0, \cdots, N-1$) could be transformed into the calculation of Eq. 17 for $k = 0, \cdots, M-1$ (in other terms $k = 0, \cdots, N/2-1$) and Eq. 18 and for $k = M, \cdots, 2M-1$ (in other terms $k = N/2, \cdots, N-1$). The benefit from this method is the reuse of the sub-components $X_N^{even}(k)$ and $X_N^{odd}(k)$ for every $k = 0, \cdots, N/2$ and a corresponding $k + N/2$ as graphically illustrated in Fig. 4.

In addition to reusing the sub-components $X_N^{even}(k)$ and $X_N^{odd}(k)$, the same principle applies to their internal calculation as they are DFTs per se. Accordingly, the problem gets transformed into following the divide and conquer algorithmic paradigm.

4.1.3 Algorithm pseudo-code and complexity analysis

Cooley–Tukey algorithm is an example of a divide-and-conquer algorithm. The time complexity is reduced from $O(N^2)$ of the original DFT to $O(Nlog(N))$.

Cooley–Tukey algorithm divides the N length input into two parts with the length $N/2$. The updating of the result requires $O(N)$ operations. So we have $L(N) = 2L(\frac{N}{2}) + O(N)$. According to Master Theorem, the time complexity is $O(Nlog(N))$ [15]. The algorithm can be seen as given in Alg. 2:

Algorithm 2 Cooley–Tukey FFT algorithm

Input: An array from $x(0)$ to $x(N-1)$
Output: An array which is the results of DFT of the array from $x(0)$ to $x(N-1)$
if $N = 1$ **then**
  return $x[0]$
else
  $x^{even} = (x_0, x_2, \cdots, x_{N-2})$
  $x^{odd} = (x_1, x_3, \cdots, x_{N-1})$
  $X_N^{even} = FFT(x_0, x_2, \cdots, x_{N-2})$
  $X_N^{odd} = FFT(x_1, x_3, \cdots, x_{N-1})$
  for $k = 0$ to $N/2 - 1$ **do**
    $X_N(k) = X_N^{even}(k) + e^{-j2\pi k/N} X_N^{odd}(k)$
    $X_N(k + N/2) = X_N^{even}(k) - e^{-j2\pi k/N} X_N^{odd}(k)$
  end for
end if
return X_N

4.2 Algorithms 2: Radix-4 FFT Algorithm

4.2.1 Description

The first idea is decimation in time. It means to divide the sequence $(x(0), \ldots, x(N-1))$ in the time domain. Cooley-Tukey is the Radix-2 time-drawn FFT algorithm, we know that the Radix-2 algorithm divides $(x(0), \ldots, x(N-1))$ into two groups of parity according to the value of N. The purpose is to transform the DFT of computing N points into the DFT of computing 2 points at $N/2$ points. And then divide into the DFT of computing 4 points at $N/4$ points. Such operation will be repeated until the sequence is decomposed into the operation of two-point DFT, which is simple addition and subtraction [8]. Therefore, it is easy to extend the conclusion that $(x(0), \ldots, x(N-1))$ can be decomposed into more groups.

Radix-4 FFT algorithm divides the sequence whose length $N = 4^l$ into 4 sequences. As we have explained above, we can transform the DFT of computing N points into the DFT of computing 4 points at $N/4$ point. Then we transform the DFT of computing $N/4$ points into the DFT of computing 4 points at $N/16$ point. In this way, finally, we get the operation of Radix-4 FFT.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{i2\pi \frac{kn}{N}} \tag{19}$$

By the definition of DFT Eq.19, we could rewrite the formula to the following Eq. 20.

$$\begin{aligned} X(k) = &\sum_{n=0}^{\frac{N}{4}-1} x(4n) e^{-\left(i\frac{2\pi\cdot(4n)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+1) e^{-\left(i\frac{2\pi(4n+1)k}{N}\right)} \\ &+ \sum_{n=0}^{\frac{N}{4}-1} x(4n+2) e^{-\left(i\frac{2\pi(4n+2)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+3) e^{-\left(i\frac{2\pi(4n+3)k}{N}\right)} \end{aligned} \tag{20}$$

Thus, DFT on a considerable input can be solved by performing DFT on partitioned inputs four times, and the inputs are split this way.

$$\mathrm{DFT}_N[x] = \mathrm{DFT}_{\frac{N}{4}}[x_0] + W_N^k \cdot \mathrm{DFT}_{\frac{N}{4}}[x_1] + W_N^{2k} \cdot \mathrm{DFT}_{\frac{N}{4}}[x_2] + W_N^{3k} \cdot \mathrm{DFT}_{\frac{N}{4}}[x_3] \tag{21}$$

$$
\begin{aligned}
x &= x(0), x(1), x(2), \cdots, x(N-1) \\
x_0 &= x(0), x(4), x(8), \cdots, x(N-4) \\
x_1 &= x(1), x(5), x(9), \cdots, x(N-3) \\
x_2 &= x(3), x(6), x(10), \cdots, x(N-2) \\
x_3 &= x(4), x(7), x(11), \cdots, x(N-1)
\end{aligned} \tag{22}
$$

The Radix-4 butterfly are as in Eq. 23 where $0 <= k <= N/4 - 1$ and $X_0 = \text{DFT}_{\frac{N}{4}}[x_0], \cdots, X_3 = \text{DFT}_{\frac{N}{4}}[x_3]$.

$$
\begin{aligned}
X(k + 0N/4) &= X_0(k) + W_N^k X_1(k) + W_N^{2k} X_2(k) + W_N^{3k} X_3(k) \\
X(k + 1N/4) &= X_0(k) + W_N^{k+N/4} X_1(k) + W_N^{2(k+N/4)} X_2(k) + W_N^{3(k+N/4)} X_3(k) \\
X(k + 2N/4) &= X_0(k) + W_N^{k+N/2} X_1(k) + W_N^{2(k+N/2)} X_2(k) + W_N^{3(k+N/2)} X_3(k) \\
X(k + 3N/4) &= X_0(k) + W_N^{k+3N/4} X_1(k) + W_N^{2(k+3N/4)} X_2(k) + W_N^{3(k+3N/4)} X_3(k)
\end{aligned} \tag{23}
$$

We can simplify Eq. 23 to:

$$
\begin{aligned}
X(k) &= X_0(k) + W_N^k X_1(k) + W_N^{2k} X_2(k) + W_N^{3k} X_3(k) \\
X(k + N/4) &= X_0(k) - jW_N^k X_1(k) - W_N^{2k} X_2(k) + jW_N^{3k} X_3(k) \\
X(k + N/2) &= X_0(k) - W_N^k X_1(k) + W_N^{2k} X_2(k) - W_N^{3k} X_3(k) \\
X(k + 3N/4) &= X_0(k) + jW_N^k X_1(k) - W_N^{2k} X_2(k) - jW_N^{3k} X_3(k)
\end{aligned} \tag{24}
$$

4.2.2 Analysis

From the above, we can find that there are 12 complex addition operations in Radix-4 FFT.

If the FFT length $N = 4^l$, DFT could continue recursively decomposed. To determine the total computational cost, considering that the number of stages is $l = log_4(N) = \frac{log_2(N)}{2}$, and each stage needs $\frac{N}{4}$ butterflies, the complexity should be:

$$
\text{Complex Multiplies} = 3\frac{N}{4}\frac{log_2(N)}{2} = \frac{3}{8}Nlog_2(N) \tag{25}
$$

$$
\text{Complex Add} = 8\frac{N}{4}\frac{log_2(N)}{2} = Nlog_2(N) \tag{26}
$$

The algorithm can be seen in Alg. 3:

5 Experimental setup and results

5.1 Experimental setup

In this experiment, the two fast algorithms, along with the baseline DFT, were all implemented following the same programming model. The codes are all written in C++ and compiled using GCC 9.4.0, which can be found in the following GitHub repository in [16]. The input signals are randomly generated using the same seed, and their lengths are all powers of 4. The experiment was performed on Linux Ubuntu 20.04 running on an AMD Ryzen Threadripper PRO 3975WX 32-Cores (2 threads for each core) with 512GB System Memory. The processes ran on different cores during the experiment to avoid any overlap of memory access and competition for computational resources.

5.2 Experimental results

We experimented on different input sizes ranging from $4^1 = 4$ to $4^{11} = 4194304$. The reason behind choosing the powers of 4 is that Radix-4 requires it and it conforms with the condition of Cooley-Tukey, which is the power 2. For

Algorithm 3 Radix-4 FFT algorithm

Input: An array from $x(0)$ to $x(N-1)$
Output: An array which is the results of DFT of the array from $x(0)$ to $x(N-1)$
if $N = 1$ **then**
  return $x(0)$
else
  $x_0 = (x(0), x(4), \cdots, x(N-4))$
  $x_1 = (x(1), x(5), \cdots, x(N-3))$
  $x_2 = (x(2), x(6), \cdots, x(N-2))$
  $x_3 = (x(3), x(7), \cdots, x(N-1))$
  $X_0 = FFT(x_0)$
  $X_1 = FFT(x_1)$
  $X_2 = FFT(x_2)$
  $X_3 = FFT(x_3)$
  $W_N^{kn} = e^{-i2\pi \frac{kn}{N}}$
  $W_N^{2kn} = e^{-i2\pi \frac{2kn}{N}}$
  $W_N^{3kn} = e^{-i2\pi \frac{3kn}{N}}$
  for $k = 0$ to $N/4 - 1$ **do**
    $X(k) = X_0(k) + W_N^k X_1(k) + W_N^{2k} X_2(k) + W_N^{3k} X_3(k)$
    $X(k + N/4) = X_0(k) - jW_N^k X_1(k) - W_N^{2k} X_2(k) + jW_N^{3k} X_3(k)$
    $X(k + N/2) = X_0(k) - W_N^k X_1(k) + W_N^{2k} X_2(k) - W_N^{3k} X_3(k)$
    $X(k + 3N/4) = X_0(k) + jW_N^k X_1(k) - W_N^{2k} X_2(k) - jW_N^{3k} X_3(k)$
  end for
end if
return X

each algorithm with each input size, 1000 runs were performed for the sake of high accuracy, and the mean results are reported in seconds in Table 2.

Size of input	Cooley-Tukey	Radix-4	DFT
4	0.000003689	0.000002867	1.90500e-16
16	0.000016361	0.000013618	1.87670e-05
64	0.000054113	0.000042901	2.14548e-04
256	0.000236232	0.000177538	3.24741e-03
1024	0.001060170	0.000815169	5.28836e-02
4096	0.004668360	0.003652710	8.48808e-01
16384	0.021242800	0.016467100	1.33362e+01
65536	0.095119600	0.072782500	2.10358e+02
262144	0.417862000	0.319617000	CPU time exceeded
1048576	1.796190000	1.365360000	CPU time exceeded
4194304	7.883840000	6.0079200000	CPU time exceeded

Table 2: Execution time needed for every algorithm (each result is an average of 1000 runs)

In order to exploit those results for comparison with the theory, we start by analyzing Fig. 5, where the abscissa axis is the logarithmic of the size and the ordinate axis is the logarithmic of the execution times.

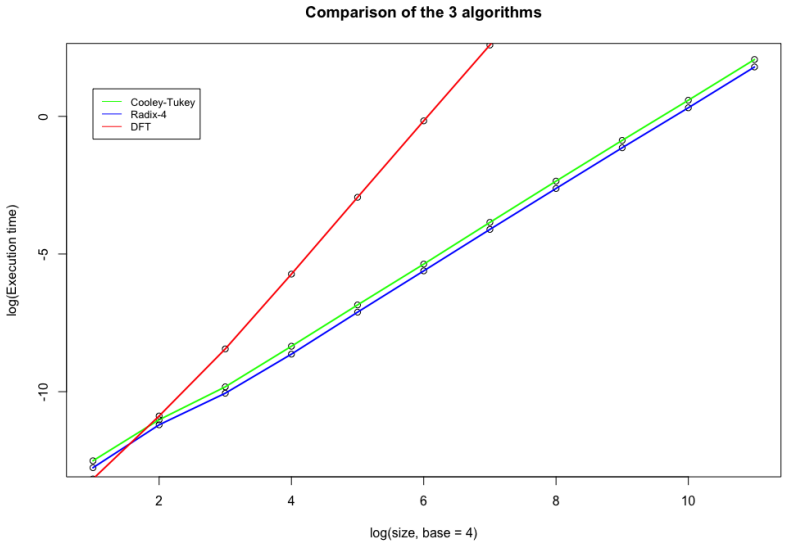

Figure 5: Comparison of the three algorithms

Theoretically, the two algorithms are of complexity $O(Nlog(N))$. We visualize the same behavior in the figure showed above with the green and blue linear lines linking the $log(size)$ and the $log(execution\_time)$. In fact, since the abscissa axis is in logarithmic scale, and the ordinate one as well, it is as if we are comparing $log(N)$ to $Nlog(N)$, which justifies the linear trend for both curves of the fast algorithms. As for the DFT curve, the one in red, it is clear that it performs worse than the fast algorithms and jumps to high values very quickly, which validates the fact that it has higher complexity of $O(N^2)$ compared to the fast ones.

However, we first notice that although the two fast algorithms confirm the theoretical results, they do have a difference in the overall speed. In addition, we can see that DFT performs worse than them, but not how much (the question is, is it really $O(N^2)$?).

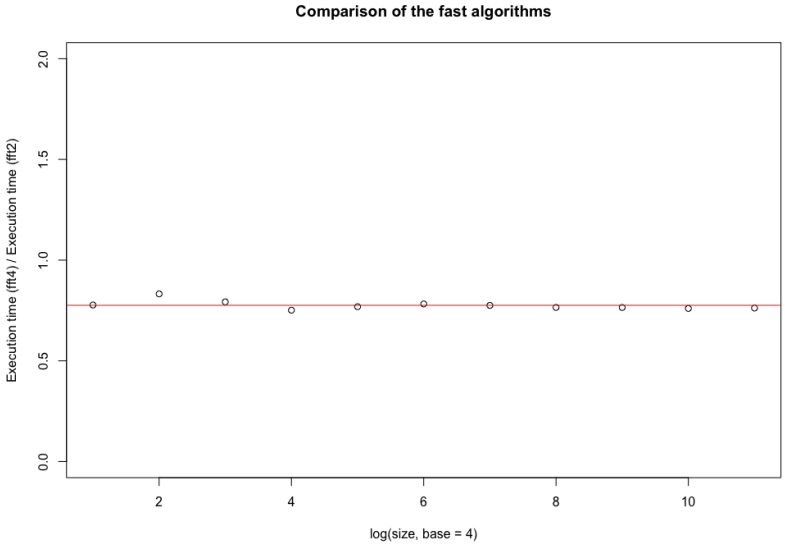

Figure 6: Comparison of the fast algorithms

We start with the first problem. Fig. 6 shows a comparison of the fast algorithms by dividing the execution time of the Radix-4 algorithm by the Cooley-Tukey one. It is clear that the execution time range varies closely to a horizontal line

of a value between 0.7 and 0.8. That gave us an observation that Radix-4 is X1.24-X1.43 faster than Cooley-Tukey. This result confirms the theoretical result about the reduction of the number of operations required in Radix-4.

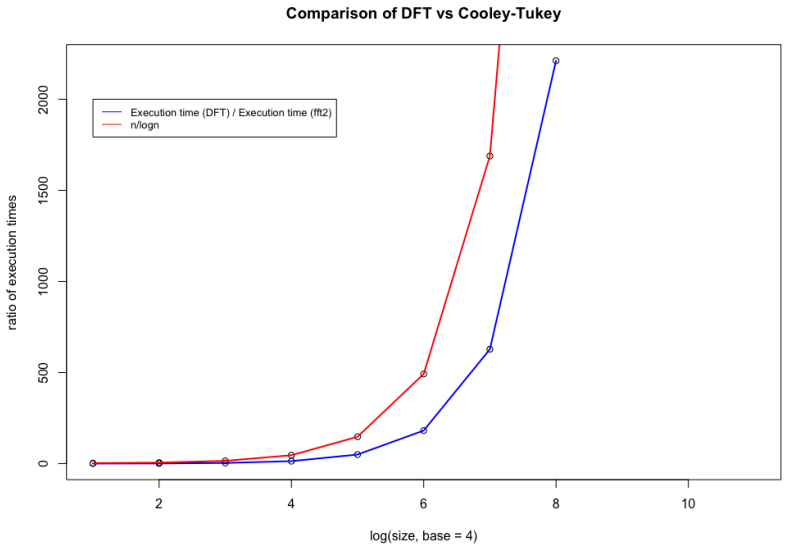

Figure 7: Comparison of the fast algorithms

Similarly, for the second question, we compare the fast algorithms to DFT. As the fast algorithms have the same complexity and the same execution time raised to a multiplicative factor, we only compare DFT to Cooley-Tukey as the result will be the same with Radix-4. Fig. 7 shows the execution time of DFT divided by the execution time of the Cooley-Tukey algorithm in the blue curve. Theoretically, DFT has a complexity of $O(N^2)$ whereas Cooley-Tukey has a complexity of $O(Nlog(N))$, which means that dividing the execution times will give a trend similar to $N^2/Nlog(N) = N/log(N)$. In the same Fig. 7 we visualize the curve of that $N/log(N)$ in red that follows the blue curve of the actual execution times ratio to some multiplicative factor, which confirms the last theoretical result.

Overall, all the theoretical results are checked and confirmed practically.

6 Conclusion

In this project, we tackled the Fast Fourier Transform problem using two well-known algorithms, Cooley-Tukey, and Radix-4. We went through the theoretical analysis from a mathematical and complexity perspective. We showed in the complexity analysis that both are faster than the traditional Discrete Fourier Transform calculation process and that the Radix-4 is faster than Cooley-Tukey by some multiplicative factor. We implemented those algorithms with the traditional one using the same programming model, and the last one served as a baseline. We ran every experiment on different input sizes with identical values for each case 1000 times. We analyzed the reported mean results. We confirmed their complexities experimentally by looking at the fast algorithms' behavior against the baseline DFT. Following that, the two fast algorithms were compared, confirming that the Radix-4 is the faster one, as the theory dictates. Lastly, we checked whether the DFT, when compared to the fast algorithms, behaves as the theory indicates and confirmed that it was the case.

References

[1] Jean Armstrong. OFDM for optical communications. *Journal of Lightwave Technology*, 27(3):189–204, February 2009.

[2] Harri Holma and Antti Toskala. *LTE for UMTS: OFDMA and SC-FDMA Based Radio Access*. John Wiley & Sons, April 2009.

[3] Innovations. LTE in a nutshell. `https://rintintin.colorado.edu/~gifford/5830-AWL/LTE%20in%20a%20Nutshell%20-%20Physical%20Layer.pdf`, 2010.

[4] Boris Bellalta. IEEE 802.11ax: High-efficiency WLANS. *IEEE Wireless Communications*, 23(1):38–46, February 2016.

[5] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, USA, 1997.

[6] Wikipedia contributors. Discrete cosine transform. `https://en.wikipedia.org/w/index.php?title=Discrete_cosine_transform&oldid=1100242965`, July 2022.

[7] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[8] Douglas L Jones. Digital signal processing: A user's guide, radix-4 fft algorithms. pages 149–153, 2006.

[9] C.M. Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968.

[10] S. Winograd. On computing the discrete fourier transform. *Mathematics of Computation*, 32(141):175–199, 1978.

[11] D. Coppersmith. An approximate fourier transform useful in quantum factoring, 2002.

[12] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Prentice Hall, Upper Saddle River, N.J., 2008.

[13] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[14] Wikipedia contributors. Cooley–Tukey FFT algorithm. `https://en.wikipedia.org/w/index.php?title=Cooley%E2%80%93Tukey_FFT_algorithm&oldid=1111684678`, September 2022.

[15] Cooley-Tukey FFT algorithms. `http://people.scs.carleton.ca/~maheshwa/courses/5703COMP/16Fall/FFT_Report.pdf`.

[16] Fast fourier transform. `https://github.com/medbzkst/FFT`.