Digital Signal Processing Laboratory Report - Lab5: Fast Fourier Transform

Haodong Zhang, 12113010

***Abstract*—In this experiment, we introduced and explained the process and expressions of the Discrete Fourier Transform (DFT). We implemented the DFT process using Matlab, employing direct expression methods. Several signals were chosen as examples to test our code implementation, and their plots were adjusted and shifted. The plots were presented with ω as the horizontal axis in rad/s units, displayed within the range of $-\pi$ to π, rather than using index k as the horizontal coordinate.**

Subsequently, we investigated the impact of zero padding and studied how the length of the DFT affects the spectrum. Due to the high complexity of the DFT, we introduced and explained the Fast Fourier Transform (FFT) algorithm. We provided a detailed explanation of its principles and implemented the FFT process using Matlab. Both divide-and-conquer and iterative methods were used to implement the FFT. Initially, we took an 8-point FFT as an example, writing three functions: FFT2, FFT4, and FFT8. We achieved iteration by calling FFT4 from FFT8 and FFT2 from FFT4. We then improved this process by encapsulating it into a function and implementing the iteration through self-calling.

Finally, we analyzed the number of multiplication operations, i.e., the complexity, used by the FFT algorithm, comparing its efficiency with the direct implementation of the DFT.

***Index Terms*—Digital Signal Processing, Matlab, Discrete Fourier Transform (DFT), Fast Fourier Transform (FFT)**

I. Introduction

THIS experiment aims to use Matlab tools to implement and analyze the discrete fourier transform (DFT). And then further introduce and analyze the fast fourier transform (FFT) algorithm on the basis of the DFT.

DTFT (Discrete-Time Fourier Transform) is the Fourier transform for discrete signals. To better utilize Fourier transforms in computer systems, the Discrete Fourier Transform (DFT) was introduced as a discrete version of DTFT, better suited for processing discrete signals in digital computer systems. The DFT process can be directly implemented in Matlab using its expressions, and the results can be visualized. However, the images obtained directly using the formula have index k as the horizontal coordinate. To better represent the spectrum of the DFT in the image, the horizontal coordinate should be adjusted to the frequency ω. Therefore, it is necessary to adjust the order and index of the DFT when plotting.

Additionally, the length N of the DFT also affects the results. Increasing N, which involves zero-padding the original signal to extend its length, theoretically refines the DFT spectrum. This process is known as zero-padding. However, directly calculating the DFT using its expression can lead to high complexity due to the numerous multiplication operations involved.

To address this issue, the Fast Fourier Transform (FFT) algorithm was proposed. The FFT algorithm significantly reduces the number of multiplication operations, thereby lowering the overall algorithm complexity. Its core idea involves divide-and-conquer and iterative methods. These processes can also be implemented using Matlab code. The FFT process can be visualized using a butterfly diagram, and the number of multiplication operations can be calculated for comparison with the direct DFT method.

This is a lab report of the Digital Signal Processing (EE323) in the Southern University of Science and Technology (SUSTech).

Haodong Zhang is a undergraduate student majoring in communication engineering, Department of Electronic and Electrical Engineering, Southern University of Science and Technology (SUSTech), Shenzhen 518055, China (email: 12113010@mail.sustech.edu.cn).

II. Discrete Fourier Transform Analysis

A. The DFT and Shifting of the Frequency Range

The Discrete Fourier Transform (DFT) is a crucial mathematical tool in the field of signal processing, employed to convert a discrete sequence or signal from the time domain to the frequency domain. The discrete version of the Fourier Transform is usually Discrete-Time Fourier Transform (DTFT) which is very useful analytically but it usually cannot be exactly evaluated on a computer because it requires an infinite sum and its inverse transform requires the evaluation of an integral which is difficult to implement in computer discrete system. Therefore, DFT is used to solve this problem which is a sampled version of the DTFT, hence it is better suited for numerical evaluation on computers. The transform and inverse transform expression are shown as below:

$$\text{(DFT)} \quad \mathrm{X_N[k]} = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N} \tag{1a}$$

$$\text{(Inverse DFT)} \quad \mathrm{x[n]} = \frac{1}{N} \sum_{k=0}^{N-1} X_N[k] e^{j2\pi kn/N} \tag{1b}$$

Where $\mathrm{X_N[k]}$ is an N point DFT of x[n] and k ranges from 0 to N-1. We can write a Matlab function to implement the DFT.

$DFTsum.m$:

```
function X = DFTsum(x)
N = length(x);
n = 0:N-1;
X = zeros(1,N);
for k = 0:N-1
    X(k+1) = x*(exp(-1j*2*pi*k*n/N)).';
end
end
```

where x is an N point vector containing the values x(0),...,x(N-1) and X is the corresponding DFT. We will test and verify this function by creating a Hamming window x of length N = 20 using the Matlab command x = hamming(20). And then we use the function above to compute the 20 points DFT of x and plot the magnitude of the DFT, $|X_{20}[k]|$, versus the index k. The corresponding figure is shown in Fig. 1.

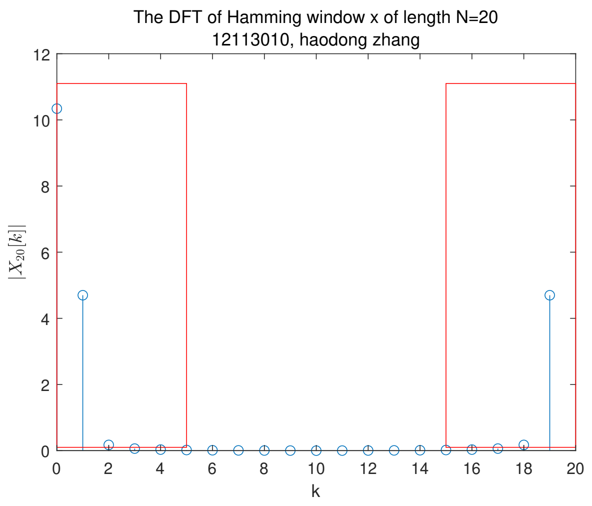

Fig. 1: The magnitude of the DFT for a Hamming window x of length 20

In this figure, we use a red rectangle to box in the regions corresponding to low frequency components. Because in discrete frequency domain, the low-frequency regions are located near $0, 2\pi, ..., or\ 2k\pi$, and the high frequency regions are located near $\pm\pi, ..., or\ (2k+1)\pi$. So, we can find that the Hamming window signal is with low frequency components. The corresponding codes are:

```
N = 20;
x = hamming(20);
k = 0:N−1;
X = DFTsum(x');
stem(k,abs(X));
hold on
rectangle('Position', [0, 0.1, 5, 11], 'EdgeColor', 'r');
hold on
rectangle('Position', [15, 0.1, 5, 11], 'EdgeColor', 'r');
hold off;
title(sprintf('The DFT of Hamming window x of length
    N=20\n12113010, haodong zhang'))
xlabel('k');
ylabel('$\left | X_{20}[k] \right | $','
    Interpreter','latex')
```

This method to plot the DFT has some disadvantages. First, the DFT values are plotted against k rather than the frequency ω. Second, the arrangement of frequency samples in the DFT goes from 0 to 2π rather than from $-\pi$ to π, as is conventional with the DTFT. In order to plot the DFT values similar to a conventional DTFT plot, we must compute the vector of frequencies in radians per sample, and then "rotate" the plot to produce the more familiar range, $-\pi$ to π. Each element of ω should be the frequency of the corresponding DFT sample X[k], which can be computed by $\omega = 2\pi k/N, k \in [0, ..., N-1]$. Therefore, we can do the process: if $\omega \geq \pi$, then it should be set to $\omega - 2\pi$. Then the resulting vectors X and ω are correct, but out of order. To reorder them, we must swap the first and second halves of the vectors using the Matlab function "fftshift". The functions are as follows:

$DTFTsamples.m$:

```
function [X,w] = DTFTsamples(x)
X = DFTsum(x);
N = length(x);
k = 0:N−1;
w = 2*pi*k/N;
w(w>=pi) = w(w>=pi)−2*pi;
w = sort(w);
X = fftshift (X);
end
```

And the figure after adjusted is shown in Fig. 2.

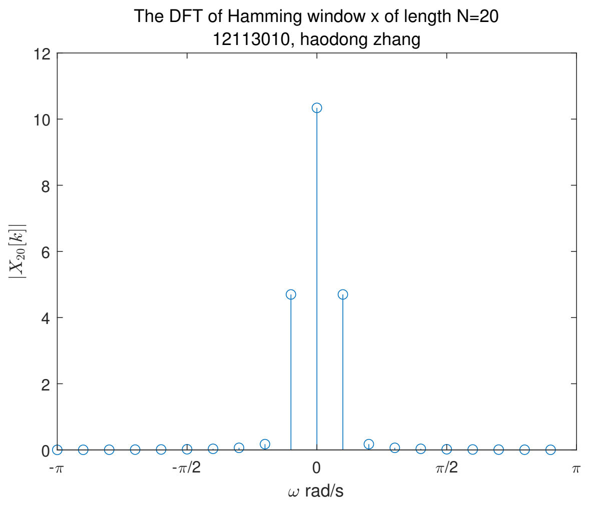

Fig. 2: The magnitude of the DFT for a Hamming window x of length 20

B. Zero Padding

The spacing between samples of the DTFT is determined by the number of points in the DFT. In this section, we will explore and illustrate this effect. We will consider the finite-duration signal:

$$x[n] = \begin{cases} sin(0.1\pi n) & 0 \leq n \leq 49 \\ 0 & otherwise \end{cases} \tag{2}$$

And then, we will compute the DTFT samples of $x[n]$ using both N = 50 and N = 200 point FT's, then plot them and compare the difference between them. Notice that when N = 200, most of the samples of $x[n]$ will be zeros because $x[n] = 0$ for N $\geq$ 50. The results are shown in Fig. 3.

From Fig. 3, it can be observed that when the length is 200, it is closer to the true Discrete-Time Fourier Transform (DTFT) compared to the case with a length of 50 (where there are two impulse functions at -0.1π and 0.1π). At the same time, it can be seen that with the increase in length, the plot of DTFT samples becomes more refined, containing more points and details. This phenomenon can be deduced from the principles of the Discrete Fourier Transform (DFT), as the frequency domain of the DFT is a sampling of the DTFT frequency domain. Therefore, a larger value of N implies more sampling points, resulting in a more refined image that

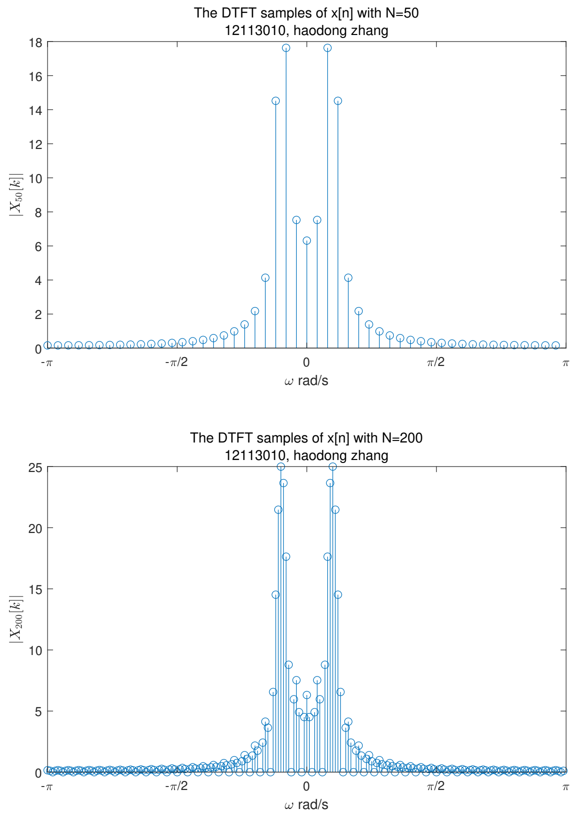

Fig. 3: The magnitude of the DFT for x[n] of length 50 and 200

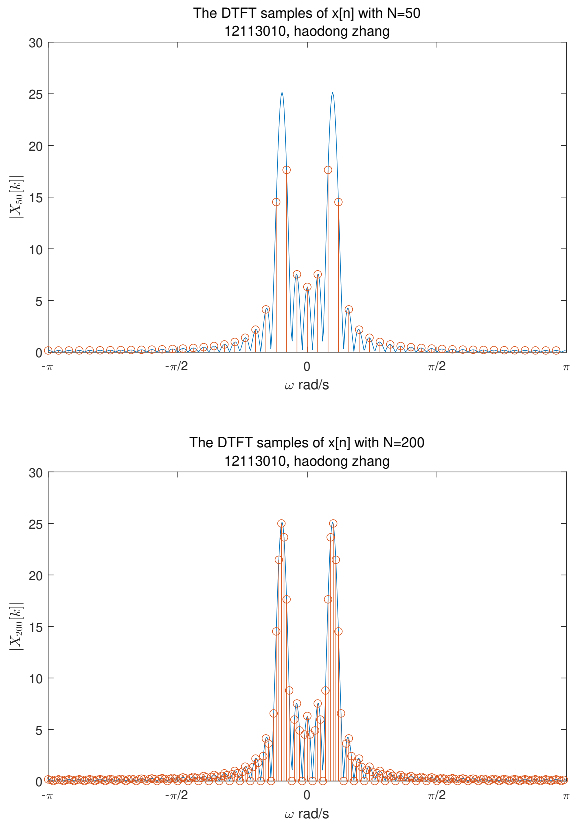

Fig. 4: The samping process: the magnitude of the DFT for x[n] of length 50 and 200

better reflects the true DTFT. The sampling process can be described more vividly in the Fig. 4 where the envelop is the DTFT of x[n] using the function in lab4. This technique of increasing length is known as “zero padding”, and may be used to produce a finer sampling of the DTFT.

The codes are as follows:

```
figure;
N = 50;
x = zeros(1,N);
for n = 0:49
    x(n+1) = sin(0.1*pi*n);
end
[X,w] = DTFTsamples(x);
subplot(2,1,1);
stem(w,abs(X));
title(sprintf('The DTFT samples of x[n] with N=50\
    n12113010, haodong zhang'))
xlabel('\omega rad/s');
ylabel('$\left | X_{50}[k] \right | $','
    Interpreter','latex')
xlim([-pi,pi])
xticks([-pi -pi/2 0 pi/2 pi]);
xticklabels({'-\pi' '-\pi/2' '0' '\pi/2' '\pi'});

N = 200;
x = zeros(1,N);
for n = 0:49
    x(n+1) = sin(0.1*pi*n);
end
[X,w] = DTFTsamples(x);
subplot(2,1,2);
stem(w,abs(X));
title(sprintf('The DTFT samples of x[n] with N=200\
    n12113010, haodong zhang'))
xlabel('\omega rad/s');
ylabel('$\left | X_{200}[k] \right | $','
    Interpreter','latex')
xlim([-pi,pi])
xticks([-pi -pi/2 0 pi/2 pi]);
xticklabels({'-\pi' '-\pi/2' '0' '\pi/2' '\pi'});
```

III. The Fast Fourier Transform Algorithm

A. The Basic Principle of FFT

In the preceding sections, we can find that the DFT is a very computationally intensive operation. Therefore, the Fast Fourier Transform (FFT) comes into play; it can provide a computationally efficient implementation of the Discrete Fourier Transform (DFT). As we saw, a straightforward implementation of the DFT can be computationally expensive because the number of multiplies grows as the square of the input length (i.e. N^2 for an N point DFT). The FFT reduces this computation using two simple but important concepts. The first concept, known as divide-and-conquer, splits the problem into two smaller problems. The second concept, known as recursion, applies this divide-and-conquer method repeatedly until the problem is solved. Now we will introduce it clearly in the following.

We should assume that N is even, so that N/2 is an integer. Then we can break the sum in equation (1a) into two sums, one containing all the terms for which n is even, and one containing all the terms for which n is odd:

$$\begin{aligned} X[k] &= \sum_{\substack{n=0\\ n\ even}}^{N-1} x[n]e^{-j2\pi k\frac{n}{N}} + \sum_{\substack{n=0\\ n\ odd}}^{N-1} x[n]e^{-j2\pi k\frac{n}{N}} \\ &= \sum_{m=0}^{N/2-1} x[2m]e^{-j2\pi k\frac{2m}{N}} + \sum_{m=0}^{N/2-1} x[2m+1]e^{-j2\pi k\frac{2m+1}{N}} \\ &= \sum_{m=0}^{N/2-1} x[2m]e^{-j2\pi k\frac{m}{N/2}} + e^{-j2\pi\frac{k}{N}} \sum_{m=0}^{N/2-1} x[2m+1]e^{-j2\pi k\frac{m}{N/2}} \end{aligned} \tag{3}$$

Thus the first sum in equation(3) is an N/2 point DFT of the even-numbered data points in the original sequence. Similarly, the second sum is an N/2 point DFT of the odd-numbered data points in the original sequence. To obtain the N point DFT of the complete sequence, we multiply the DFT of the odd-numbered data points by the complex exponential factor $e^{-j2\pi\frac{k}{N}}$, and then simply sum the two N/2 point DFTs. Then, we can define two new N/2 point data sequences $x_0[n]$ and $x_1[n]$, which contain the even and odd-numbered data points from the original N point sequence. This separation of even and odd points is called decimation in time. The N point DFT of x[n] is then given by:

$$X[k] = X_0[k] + e^{-j2\pi\frac{k}{N}} X_1[k] \ for\ k = 0, ..., N-1 \tag{4}$$

where $X_0[k]$ and $X_1[k]$ are the N/2 point DFT's of the even and odd points. And then equation(4) requires less computation than the original N point DFT because we need to only compute $X_0[k]$ and $X_1[k]$ for N/2 values of k rather than the N values. Furthermore, the complex exponential factor $e^{-j2\pi\frac{k}{N}}$ has the property that $-e^{-j2\pi\frac{k}{N}} = e^{-j2\pi\frac{k+N/2}{N}}$. Then these two facts may be combined to yield a simpler expression for the N point DFT:

$$\begin{aligned} X[k] &= X_0[k] + W_N^k X_1[k] \\ X[k+N/2] &= X_0[k] - W_N^k X_1[k] \end{aligned} \tag{5}$$

where the complex constants defined by $W_N^k = e^{-j2\pi\frac{k}{N}}$ are commonly known as the twiddle factors. For each odd and even part, we can continue to subdivide them, forming a recursive process. Next, we will implement it first using a simple divide-and-conquer approach, and then employ a recursive divide-and-conquer method.

B. Implementation of Divide-and-Conquer DFT

In this section, we will implement the DFT transformation using the equation(5) by writing a Matlab function.

$dcDFT.m$:

```
function X = dcDFT(x)
N = length(x);
X = zeros(1,N);
x0 = x(1:2:N);
x1 = x(2:2:N);
X0 = DFTsum(x0);
X1 = DFTsum(x1);
for i=1:N/2
    X(i) = X0(i)+exp(-1j*2*pi*(i-1)/N)*X1(i);
    X(i+N/2) = X0(i)-exp(-1j*2*pi*(i-1)/N)*X1(i);
end
end
```

In this function, we separate the samples of x into even and odd points and use the function DFTsum mentioned above to compute the two N/2 point DFT's, and then multiply it by the twiddle factors. And at last, we can combine the two DFT's to form X. We can test this function by using it to compute the DFT's of the following signals:

1. $x[n] = \delta[n]$ for N = 10.
2. $x[n] = 1$ for N = 10.
3. $x[n] = e^{j2\pi n/N}$ for N = 10.

The results are shown in Fig. 5. And the results from direct DFT of these siganl using the function DFTsum are shown in Fig. 6. From these two figures, we can see that the results of Divide-and-Conquer DFT are correct whcih indict the algorithm is correct.

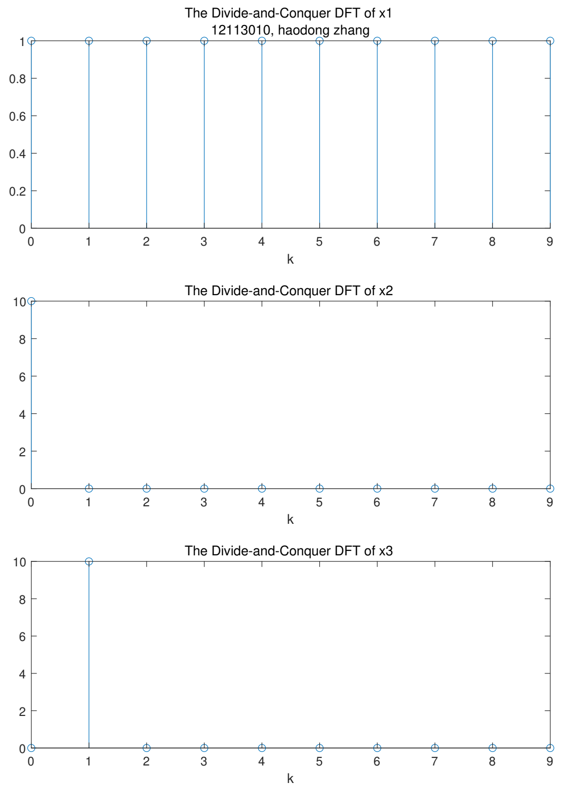

Fig. 5: The Divide-and-Conquer DFT of different signals x[n]

The codes are as following:

```
figure;
n = -5:4;
k=0:9;
x1 = (n==0);
X1 = dcDFT(x1);
x2 = ones(1,10);
X2 = dcDFT(x2);
```

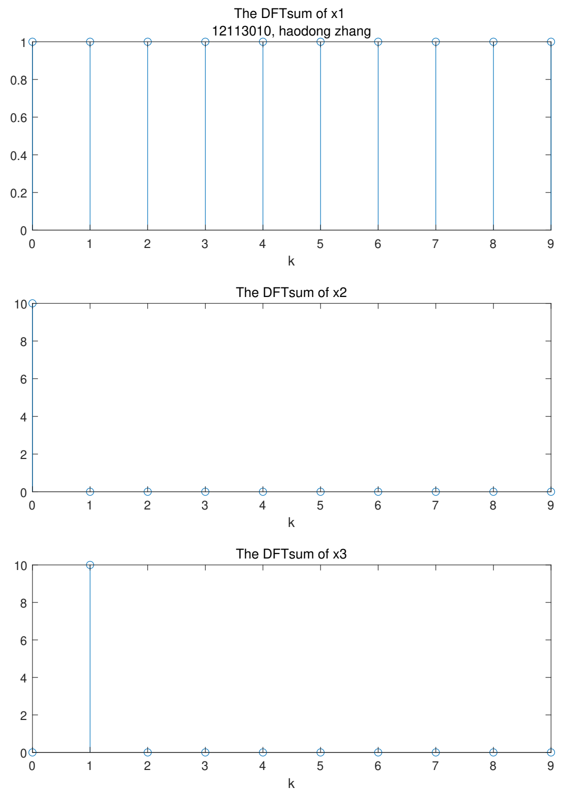

Fig. 6: The direct DFT of different signals x[n]

```
x3 = exp(1j*2*pi*n/10);
X3 = dcDFT(x3);
subplot (3,1,1)
stem(k,abs(X1));
 title (sprintf('The Divide-and-Conquer DFT of x1 \
     n12113010, haodong zhang'))
xlabel('k');
subplot (3,1,2)
stem(k,abs(X2));
 title (sprintf('The Divide-and-Conquer DFT of x2 '))
xlabel('k');
subplot (3,1,3)
stem(k,abs(X3));
 title (sprintf('The Divide-and-Conquer DFT of x3 '))
xlabel('k');
figure;
X11 = DFTsum(x1);
X22 = DFTsum(x2);
X33 = DFTsum(x3);
subplot (3,1,1)
stem(k,abs(X11));
 title (sprintf('The DFTsum of x1 \n12113010, haodong
     zhang'))
xlabel('k');
subplot (3,1,2)
stem(k,abs(X22));
 title (sprintf('The DFTsum of x2'))
xlabel('k');
subplot (3,1,3)
stem(k,abs(X33));
 title (sprintf('The DFTsum of x3'))
xlabel('k');
```

We can calculate that the number of multiplies that are required in this approach is that: For each N/2 DFT of the odd and even parts, $\mathrm{N}^2/4$ multiplication operations are required. Combining the odd and even parts involves an additional $2 * (\mathrm{N}/2) = \mathrm{N}$ multiplication operations. Therefore, the total number of operations needed is $2 * \mathrm{N}^2/4 + \mathrm{N} = \mathrm{N} + \mathrm{N}^2/2$ which is less than the number of direct DFT, N^2 for $\mathrm{N} \geq 3$.

C. *Implementation of Recursive Divide-and-Conquer DFT*

The second basic concept underlying the FFT algorithm is that of recursion. As mentioned above as, if N/2 is also even. Then we may apply the same decimation-in-time idea to the computation of each of the N/2 point DFT's. Usually, we can suppose N is a power of 2, i.e. $\mathrm{N} = 2^p$ for some integer p. We can then repeatedly decimate the sequence until each subsequence contains only two points. And the last 2 point DFT is a simple sum and difference of values as below:

$$X[0] = x[0] + x[1]$$
$$X[1] = x[0] - x[1] \qquad (6)$$

Firstly, we can take 8-points DFT as example to illustrate the recursive process. The flow diagram that results for an 8-point DFT is shown in Fig. 7. There are 3 stages of twiddle factors (in the first stage, the twiddle factors simplify to "1"). The fundamental idea depicted in the graph is that the computation of the third-level FFT8 utilizes the results of FFT4, FFT4 directly uses the results of FFT2, and this process continues until the first-level FFT2 is computed directly. This process represents a recursive procedure. The diagram is called a butterfly due to its resemblance, and thus, this algorithm is also referred to as butterfly operation. We can write three Matlab

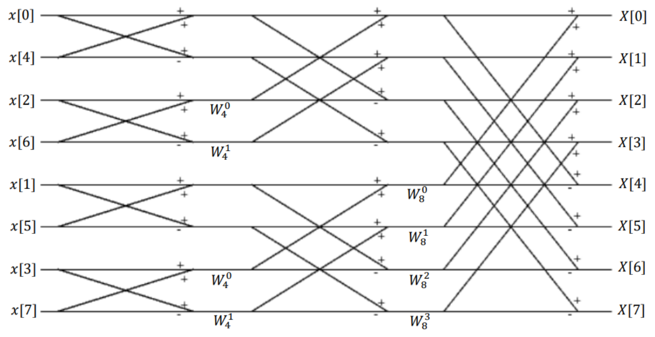

Fig. 7: The recursive process of 8-Point FFT

functions to compute the 2-, 4-, and 8-point FFT's:

$FFT2.m$:

```
function X = FFT2(x)
N = length(x);
X = zeros(1,N);
X(1) = x(1)+x(2);
X(2) = x(1)−x(2);
end
```

$FFT4.m$:

```
function X = FFT4(x)
N = length(x);
X = zeros(1,N);
x0 = x(1:2:N);
x1 = x(2:2:N);
X0 = FFT2(x0);
```

```
X1 = FFT2(x1);
for  i=1:N/2
    X(i)  = X0(i)+exp(−1j∗2∗pi∗(i−1)/N)∗X1(i);
    X(i+N/2) = X0(i)−exp(−1j∗2∗pi∗(i−1)/N)∗X1(i);
end
end
```

$FFT8.m$:

```
function  X = FFT8(x)
N = length(x);
X = zeros(1,N);
x0 = x (1:2: N);
x1 = x (2:2: N);
X0 = FFT4(x0);
X1 = FFT4(x1);
for  i=1:N/2
    X(i)  = X0(i)+exp(−1j∗2∗pi∗(i−1)/N)∗X1(i);
    X(i+N/2) = X0(i)−exp(−1j∗2∗pi∗(i−1)/N)∗X1(i);
end
end
```

In the FFT2 function, we directly compute the 2-point DFT using equation(6). And then the functions FFT4 and FFT8 compute their respective FFT's using the divide and conquer strategy which means that FFT8 calls FFT4, and FFT4 calls FFT2. This process embodies the concept of recursion. Then we can test our function FFT8 by using it to compute the DFT's of the following signals and then compare these results to the previous ones.

1. $x[n] = \delta[n] \quad \text{for N} = 8.$
2. $x[n] = 1 \quad \text{for N} = 8.$
3. $x[n] = e^{j2\pi n/N} \quad \text{for N} = 8.$

The results are shown in Fig. 8. From the figure, it can be seen that the results we calculated are consistent with those obtained previously. This indicates that our program is correct, and our recursive approach and algorithm are also correct. And we can list the output of FFT8 for the second signal $x[n] = 1, forN = 8$ as follows:

$$y[n] = [\ 8\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ ],\ n = 0, 1, ..., 7$$

Now, we can calculate the total number of multiplies by twiddle factors required for this 8-points FFT according to flow diagram in Fig. 7. If we include all twiddle factors in each stage, there are 8 multiplications in each stage, so there are 3*8=24 multiplications. We can extend it to $\text{N} = 2^{\text{p}}$ point FFT, there are p stages of computation, and there are N multiplications in each stage, so the total number of multiplies is $\text{Np} = \text{Nlog}_2\text{N}$. We can compare it with the direct implementation of DFT. For example, When p=10 it requires $2^{10} * 10 = 10240$ multiplies, but it requires $(2^{10})^2 = 1048576$ from which we can find obviously the FFT can reduce the complexity dramatically. We can further simplify the butterfly computation using $W_N^{r+N/2} = -W_N^r$, and then we reduce it to $\frac{\text{N}}{2}\text{p} = \frac{\text{N}}{2}\text{log}_2\text{N}$. And then we can continue to exclude the trivial multiplications with $W_N^0 = 1$, $W_N^{N/2} = -1$ and $W_N^{N/4} = -j$ the exact count of non-trivial complex multiplications are even less, given by $\frac{\text{N}}{2}(\text{log}_2\text{N}-2)-(\frac{\text{N}}{2}-2)$.

The codes are as follows:

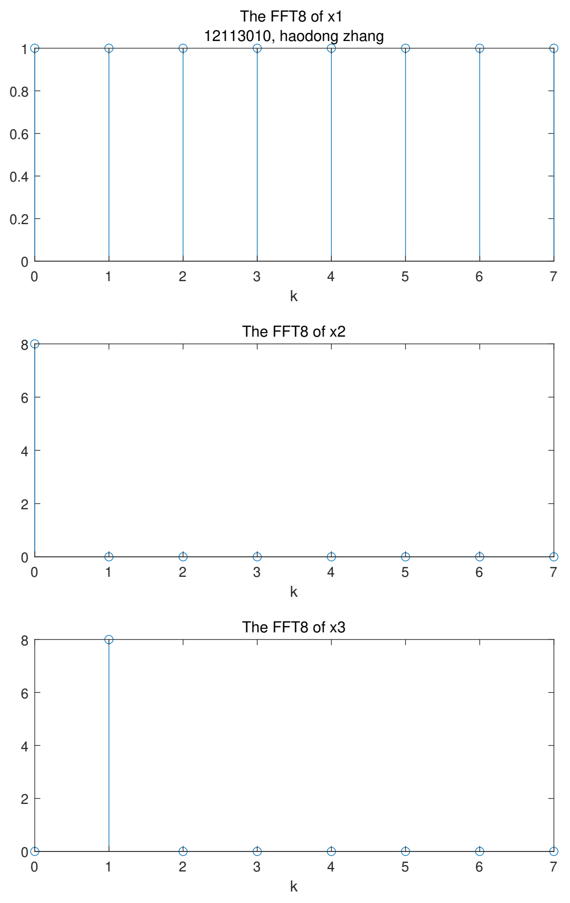

Fig. 8: The FFT8 of different signals

```
figure ;
n = −3:4;
k=0:7;
x1 = (n==0);
X1 = FFT8(x1);
x2 = ones (1,8) ;
X2 = FFT8(x2);
x3 = exp(1j∗2∗pi∗n/8);
X3 = FFT8(x3);
subplot (3,1,1)
stem(k,abs(X1));
 title ( sprintf ('The FFT8 of x1 \n12113010, haodong
    zhang'))
xlabel ('k');
subplot (3,1,2)
stem(k,abs(X2));
 title ( sprintf ('The FFT8 of x2'))
xlabel ('k');
subplot (3,1,3)
stem(k,abs(X3));
 title ( sprintf ('The FFT8 of x3'))
xlabel ('k');
```

After that, we can find that the FFT4.m and FFT8.m have almost the exact same form. The only difference between them is the length of the input signal, and the function called to compute the (N/2)-point DFTs. Obviously, it's redundant to write a separate function for each specific length DFT when they each have the same form. The preferred method is to write a recursive function, which means that the function calls itself within the body. It is imperative that a recursive function has a condition for exiting without calling itself, i.e. N=2, otherwise it would never terminate. So, we write a recursive function to achieve the calling itself process.

$fft_{stage}.m$:

```
function X = fft_stage (x)
    N = length(x);
    X = zeros(1,N);
    x0 = x(1:2:N);
    x1 = x(2:2:N);
    if N == 2
        X0 = x0;
        X1 = x1;
    else
        X0 = fft_stage (x0);
        X1 = fft_stage (x1);
    end
    for i=1:N/2
        X(i) = X0(i)+exp(−1j∗2∗pi∗(i−1)/N)∗X1(i);
        X(i+N/2) = X0(i)−exp(−1j∗2∗pi∗(i−1)/N)∗X1(i);
    end
end
```

Its structure is similar to the functions mentioned earlier, and its concept is straightforward. If N is not equal to 2, it remains in a recursive state, continually calling itself and using odd-even computation methods for calculation. It continues until N becomes 2, at which point a simple summation and subtraction of two terms occur, thus completing the recursion and obtaining the final result. Similarly, we can test fft_{stage} on the three 8 points signals given above, and verify that whether it returns the same results as FFT8 or not. The results are shown in Fig. 9 which is same as the results of FFT8 in Fig. 8. So, the program and recursive algorithm are also correct. It is worth noting that the signal lengths required here are all in the form of 2^n. For signals that do not meet this length criterion, the method of zero-padding mentioned earlier can be employed. This involves extending the signal length to the nearest 2^n by appending zeros. Importantly, zero-padding does not alter the results of the DFT, and it can refine the frequency spectrum, enabling further down-sampling to obtain the DFT of the original signal. Hence, the FFT algorithm is universally applicable to any signal, with the flexibility to accommodate non-power-of-2 signal lengths through zero-padding.

The codes are as follows:

```
figure;
n = −3:4;
k=0:7;
x1 = (n==0);
X1 = fft_stage (x1);
x2 = ones(1,8);
X2 = fft_stage (x2);
x3 = exp(1j∗2∗pi∗n/8);
X3 = fft_stage (x3);
subplot (3,1,1)
stem(k,abs(X1));
title (sprintf('The FFT of x1 \n12113010, haodong
    zhang'))
xlabel('k');
subplot (3,1,2)
stem(k,abs(X2));
title (sprintf('The FFT of x2'))
xlabel('k');
subplot (3,1,3)
stem(k,abs(X3));
title (sprintf('The FFT of x3'))
xlabel('k');
```

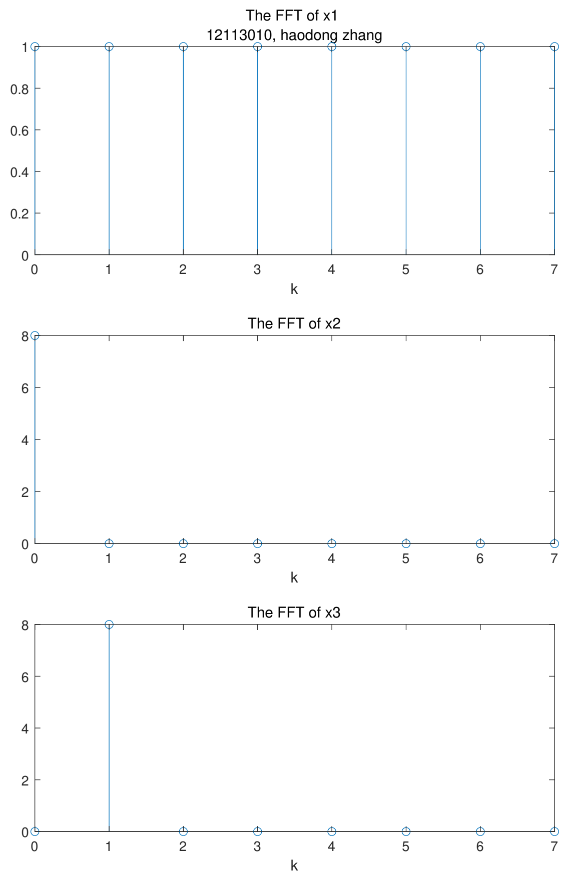

Fig. 9: The FFT of different signals

IV. Conclusion

Through this experiment, we gained a profound understanding of the calculation methods of the Discrete Fourier Transform (DFT) and made adjustments to its images. On one hand, we adjusted the horizontal coordinate to the frequency ω, and modified the display range to $-\pi$ to π. To achieve this, we first mapped k to ω, and then applied the fftshift function to shift the signal, meeting the specified requirements. Additionally, we explored the impact of zero-padding and found that increasing the length of the DFT through zero-padding results in a finer DFT spectrum. This is because the DFT itself is a sampling in the frequency domain of the DTFT, and increasing N essentially increases the sampling rate, providing a more detailed frequency domain representation.

Furthermore, to reduce the complexity of the DFT, we introduced the basic principles of the Fast Fourier Transform (FFT) algorithm. We used butterfly diagrams to visually demonstrate the FFT process, highlighting its core idea of employing divide-and-conquer and recursive methods. We implemented these concepts using Matlab. In the end, we used a self-calling approach in a program to implement the recursive method. We also analyzed the complexities of both methods, specifically the number of multiplication operations involved. For N point FFT, the total number of multiplies is $\mathrm{N}\log_2\mathrm{N}$ including all twiddle factors in each stage. To further simplify the butterfly computation using $W_N^{r+N/2} = -W_N^r$, and then the total numbers are reduced to $\frac{\mathrm{N}}{2}\log_2\mathrm{N}$. And then after excluding the trivial multiplications with $W_N^0 = 1$, $W_N^{N/2} = -1$ and $W_N^{N/4} = -j$, the exact count of non-trivial complex multiplications are even less, given by $\frac{\mathrm{N}}{2}(\log_2\mathrm{N}-2)-(\frac{\mathrm{N}}{2}-2)$. Therefore, the FFT algorithm can reduce the complexity dramatically compared to the direct DFT method.

Acknowledgments

Thanks to the teachers and teaching assistants for their guidance. Thank the teacher for teaching us theoretical knowledge and providing the lab manual for us, which raised many meaningful questions for us and guided and inspired us to think deeply.