Digital System Design Course Labora (March 2024)

12110405 Zhewei Chen

Abstract—This lab report will continue the discussion of the discrete fourier transform DFT and introduce fast fourier transform(FFT).

Index Terms—Full-adder, Vivado, VHDL, Simulation

I. Introduction

THIS laboratory will continue the discussion of the DFT and will introduce the Fast Fourier Transform (FFT). Firstly, we try to shift the frequency range and horizontal axis to make our DFT values plot more similar to the conventional DTFT plot. Then, we researched how the zero padding affects our DFT values plot. When studying FFT, we first explored the divide and conquer method to accelerate DFT. Then, by employing a recursive approach, we implemented the aforementioned method to achieve an FFT of lengths 2, 4, 8. Additionally, we will study how to use zero padding to compute the FFT of lengths N, where $N = 1, 2, 3 \dots$.

II. Experimental Contents

A. Continuation of DFT Analysis

This section continues the analysis of the DFT started in the previous week's laboratory.

DFT: $$X_N[k] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi kn/N}$$

1) Shifting the Frequency Range :

First, create a Hamming window x of length $N = 20$, then use the Matlab function DFTsum to compute the 20-point DFT of x. Plot the magnitude of the DFT, $|X_{20}[k]|$, versus the index k.

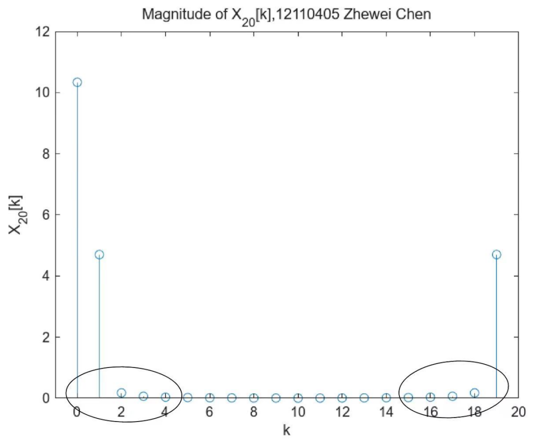


Fig. 1: Plot of $|X_{20}[k]|$ versus the index k

Analysis:

1 - The regions of low frequency component is shown in the circles of Fig. 1.

2 - There exists two disadvantages of our DFT values plots.

2.1 The frequency range of the DFT is from 0 to 2π, which is different from the conventional DTFT plot, which is from $-\pi$ toπ.

2.2 The horizontal axis of the DFT values plot is against to k rather than ω.

Then we can write a MATLAB fourier called **DTFTsamples.m** to get the plot of DTFT samples using **DFTsum.m** and **fftshift**() to resolve the two problems above.

DTFTsamples.m

```
function [X,w] = DTFTsamples(x)
N = length(x);
k = 0:N-1;
w = 2*pi*k/N;
w(w>=pi)=w(w>=pi)-2*pi;% shift the
    range from [0,2*pi] to [-pi,pi]
w = fftshift(w);
X = fftshift(DFTsum(x));
display(w);display(X);
end
```

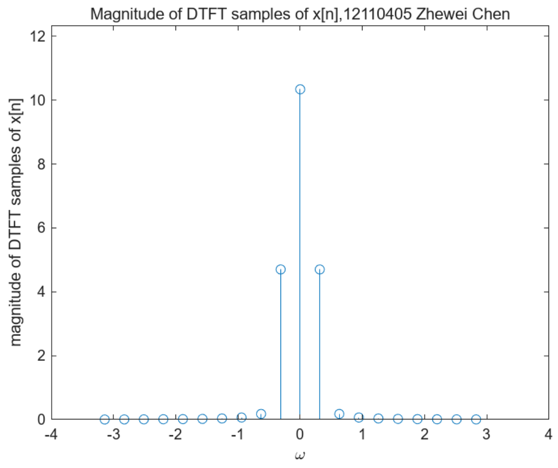


Fig. 2: Plot of DTFTsamples of hamming window with N = 20

Analysis:

1 - By transforming k to ω by $\omega = 2\pi k/N$,the plot is against ω now.

2 - Using function **fftshift**() to shift both ω and DTFTsamples, we can get the plot in the range of $[-\pi, \pi]$.

2) Zero Padding:

This section we will use finite duration sequence $x[n]$ to illustrate the effect of zero padding on the DFT.

$$x[n] = \begin{cases} \sin(0.1\pi n), & 0 \le n \le 49 \\ 0, & \text{otherwise} \end{cases}$$

$$\text{length}(x[n]) = N$$

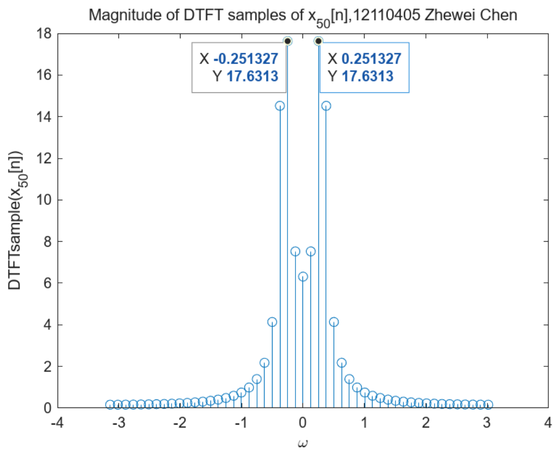


Fig. 3: Plot of DTFTsamples when N = 50

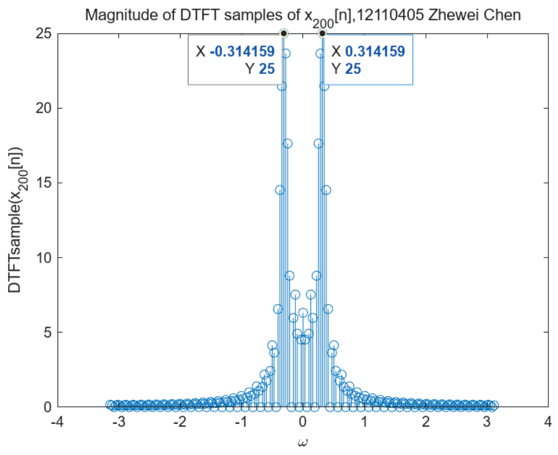


Fig. 4: Plot of DTFTsamples when N = 200(with zero padding)

Analysis:

1 - Plot of case **N = 200** (with zero padding) looks more similar to the true DTFT plot than case $N = 50$.

2 - The reasons why two plots look so different:

Firstly, we need to derive how the true DTFT plot looks like.According to the last DFT lab, we can know that $x[n]$ can be a truncated function of $y[n] = \sin(0.1\pi n), -\infty < n < \infty$,with window function $\omega[n] = 1, 0 \le n \le 49$.

$$\begin{aligned} X(e^{j\omega}) &= Y(e^{j\omega}) * W(e^{j\omega}) \\ &= (\pi\delta(\omega - 0.1\pi) + \pi\delta(\omega + 0.1\pi))W(e^{j\omega}) \\ &= \pi W(e^{j(\omega-0.1\pi)}) + \pi W(e^{j(\omega+0.1\pi)}) \end{aligned}$$

Again, according to the last lab,

$$W\left(e^{j\omega}\right) = e^{-j\omega(N-1)/2}\frac{\sin(\omega N/2)}{\sin(\omega/2)}$$

So we can derive that $X(e^{j\omega})$ is composed of two shifted sinc functions, centered at $\omega = 0.1\pi$ and $\omega = -0.1\pi$ respectively.

Both Fig. 3 and Fig. 4 are the frequency samples of $X(e^{j\omega})$. However, Fig.4 show more details(for example, the two peak of DTFTsamples is closer to the true DTFT) of $X(e^{j\omega})$ than Fig.3, because the **frequency resolution** of Fig.4 is higher than Fig.3.

3 - The reasons why case N =200 (with zero padding) can provide higher **frequency resolution**:

Recall the mathematical definition of DFT and DTFT, we can know that DFT is frequency sampling of DTFT at $\omega = 2\pi k/N, k = 0, 1, ..., N-1$, which also means DFT has a frequency resolution of $\omega = 2\pi k/N$.

When we increase the length of $x[n]$ with zero padding, it will **not** change the value of $X_N[k]$.Because the data we add into $x_N[k]$ is zero. However the $\omega = 2\pi k/N$ will be more dense, which means the frequency resolution will be **higher**.

So the case N =200 (with zero padding) can provide higher frequency resolution and is similar to the true DTFT plot.

B. The Fast Furier Transform

1) Implementation of Divide-and-Conquer DFT:

This section we will implement the divide-and-conquer DFT algorithm in Matlab. According to the lab manual, the divide-and-conquer DFT algorithm is as follows:

$$\left.\begin{aligned} X[k] &= X_0[k] + W_N^k X_1[k] \\ X[k+N/2] &= X_0[k] - W_N^k X_1[k] \end{aligned}\right\} \text{for } k = 0, ..\frac{N}{2} - 1$$

where the complex constants defined by $W_N^k = e^{-j2\pi k/N}$ are called twiddle factors. We can use digram to illustrate the divide-and-conquer DFT algorithm as Fig. 5.

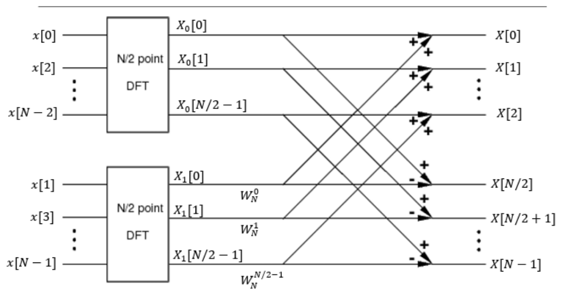


Fig. 5: DIgram of divide and conquer DFT of N-point, computed using the two N/2-point DFT's $X_0^{N/2}[k]$ and$X_1^{N/2}[k]$

dcDFT.m

```
function X = dcDFT(x)
```

```
N=length(x);
x0=x(1:2:N);% even part
x1=x(2:2:N);% odd part
X0=DFTsum(x0);
X1=DFTsum(x1);
k=0:1:N/2-1;
twiddle_factor=exp(-1i*2*pi*k/N);
X1=twiddle_factor.*X1;
X=[X0+X1 X0-X1];
end
```

DFTsum.m

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

Then we test the functions by input signal

$x_1[n] = \delta[n]$, for $N = 10$
$x_2[n] = 1$, for $N = 10$
$x_3[n] = e^{j2\pi n/N}$, for $N = 10$

and compare the results computed by **DFTsum.m** and **dcDFT.m** (The **DFTsum.m** is a direct implementation of DFT and it is from the last lab.)

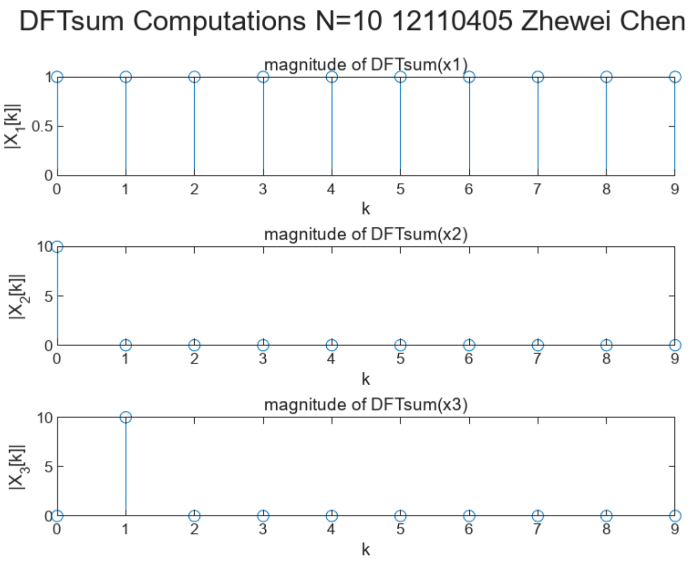


Fig. 6: Plot of the calculation results of **DFTsum.m** for $x_1[n]$,$x_2[n]$,$x_3[n]$ with N=10

Analysis:

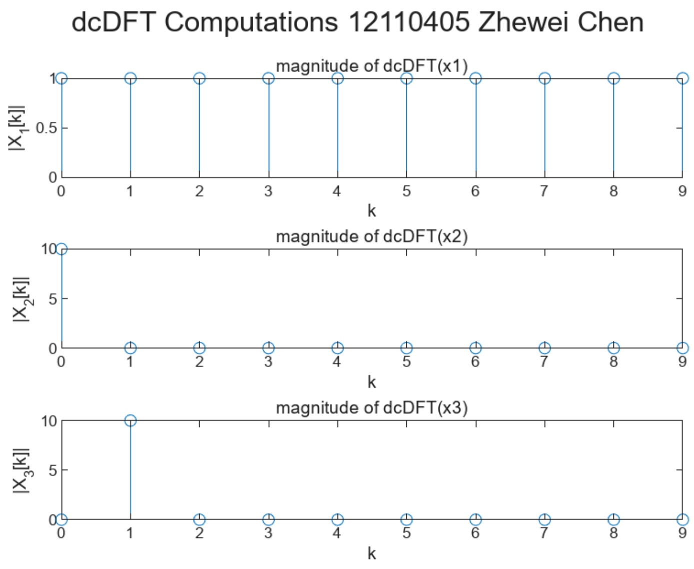


Fig. 7: Plot of the calculation results of **dcDFT.m** for $x_1[n]$,$x_2[n]$,$x_3[n]$ with N=10

1 -The results computed by **DFTsum.m** and **dcDFT.m** are the same. In function **dcDFT.m**, we divide the input signal into even and odd parts, and then compute the DFT of each part. Finally, we combine the results of the two parts to get the result.

2 - The number of multiplies that are required in this approach to computing an N point DFT is $(\mathbf{N}+\mathbf{N^2})/\mathbf{2}$. (Consider a multiply to be one multiplication of real or complex numbers.)

So the total number of multiplications is $(N + N^2)/2$. For two parts $N/2$-point DFT, we need two of $(N/2)^2$ multiplications. To simplify, we also consider constant twiddle factor (for example,1,-1) as one multiplication. So we need $N/2$ multiplications for twiddle factor. Because the in **dcDFT.m** we only need to compute the twiddle factor in the range of $k = 0, 1, ..., N/2 - 1$. So the total number is $(N/2)^2 \times 2 + N/2 = (N + N^2)/2$.

2) Recursive Divide and Conquer :
In this section we will implement the divide-and-conquer algorithm recursively, that means recursively divide the DFT computations until the length of the sequence is 2 and then combine the results. Using this algorithm, we write MATLAB functions **FFT2.m**, **FFT4.m**, **FFT8.m** to compute the FFT of lengths 2, 4, 8 respectively.

FFT2.m

```
function X = FFT2(x)
X(1)=x(1)+x(2);
X(2)=x(1)-x(2);
end
```

FFT4.m

```
function X = FFT4(x)
```

```
N=4;%N=length(x);
x_even=x(1:2:N);    X_even=FFT2(x_even);
x_odd=x(2:2:N);     X_odd=FFT2(x_odd);
k=0:N/2-1;
twiddle_factor=exp(-1i*2*pi*k/N);
X_odd=twiddle_factor.*X_odd;
X=[X_even+X_odd     X_even-X_odd];
end
```

FFT8.m

```
function X = FFT8(x)
N=8;%N=length(x);
x_even=x(1:2:N);    X_even=FFT4(x_even);
x_odd=x(2:2:N);     X_odd=FFT4(x_odd);
k=0:N/2-1;
twiddle_factor=exp(-1i*2*pi*k/N);
X_odd=twiddle_factor.*X_odd;
X=[X_even+X_odd     X_even-X_odd];
end
```

fftstage.m

```
function X = fftstage(x)
N=length(x);
x_even=x(1:2:N);
X_even=fftstage(x_even);
x_odd=x(2:2:N); X_odd=fftstage(x_odd);
k=0:N/2-1;
twiddle_factor=exp(-1i*2*pi*k/N);
X_odd=twiddle_factor.*X_odd;
X=[X_even+X_odd     X_even-X_odd];
end
```

Then we test the functions by input signal :

$x_1[n] = \delta[n]$, for $N = 8$
$x_2[n] = 1$, for $N = 8$
$x_3[n] = e^{j2\pi n/N}$, for $N = 8$
and compare the results computed by **DFTsum.m** and **FFT8.m**

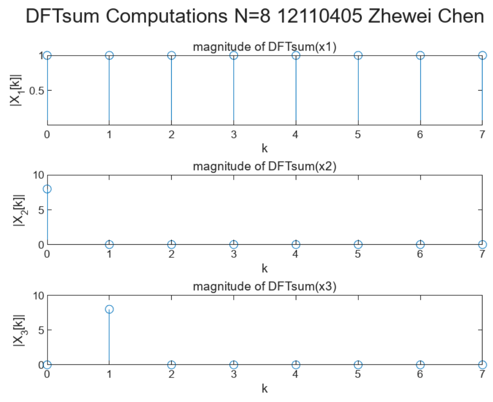


Fig. 8: Plot of the calculation results of **DFTsum.m** for $x_1[n]$,$x_2[n]$,$x_3[n]$ with N=8

Analysis:

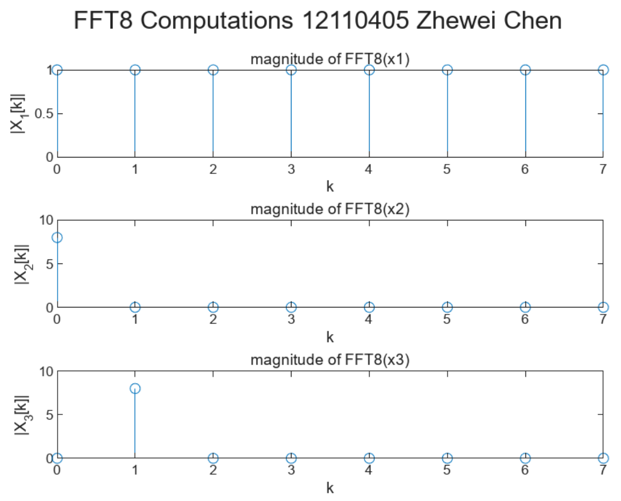


Fig. 9: Plot of the calculation results of **FFT8.m** for $x_1[n]$,$x_2[n]$,$x_3[n]$ with N=8

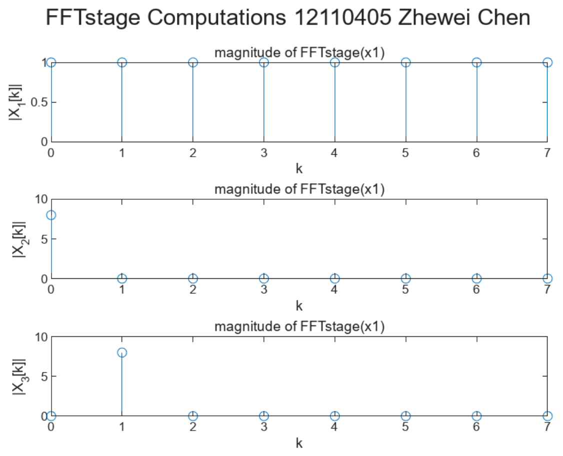
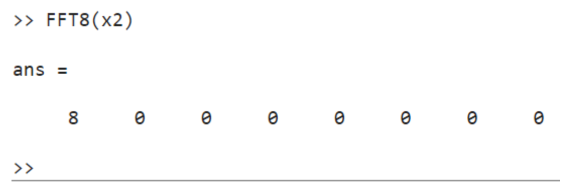


Fig. 10: Plot of the calculation results of **fftstage.m** for $x_1[n]$,$x_2[n]$,$x_3[n]$ with N=8

1 -The results computed by **DFTsum.m**, **FFT8.m** and **fftstage** are the same. In function **FFT8.m**, we compute the half DFTs with **FFT4.m**, and in function **FFT4.m**, we compute the half DFTs with **FFT2.m**. In function **fftstage.m**, we compute the half DFTs with **fftstage.m** recursively.

2 -



```
>> FFT8(x2)

ans =

     8     0     0     0     0     0     0     0

>>
```

Fig. 11: Result of **FFT8.m** with input signal $x_2[n]$,for N=8

3 -The total number of multiplies by twiddle factors

required for 8-point FFT(A multiply is a multiplication by a real or complex number.) : $8/2\times\log_2 8 =$**12**.(Including considering $W_N^0 = 1$ and $W_2^1 = -1$ as one multiplication respectively)

4 - Including considering constant twiddle factors like $W_N^0 = 1$ as one multiplication:

A formula for the number of multiples required for an $N = 2^p$ point FFT : $(\mathbf{N\log_2 N})/\mathbf{2}$ or $\mathbf{p2^{p-1}}$

As we derive the number of multiples required **dcDFT.m** with length N input signal. We do the $log_2 N$ times divisions, each division increase $N/2$ multiples. So the total number of multiples is $(Nlog_2 N)/2$ or $p2^{p-1}$.

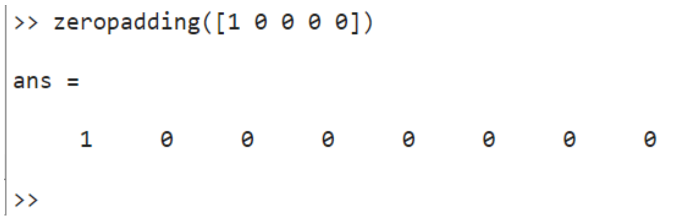
Consider a $p = 10$ DFT.Compared to number of multiplies required for direct implementation needs $N^2 = 2^{2p}$=1048576 multiplications, FFT just needs $10 \times 2^9 = 5120$ multiplications

III. Expasion Content

If a signal is not in the form of $N = 2^p$, we can use zero padding to expand the signal to the form of $N = 2^p$.

zeropadding.m

```
 function [y] = zeropadding(x)
N=length(x);
 nextPowerOf2 = nextpow2(N);
 N1 = 2^nextPowerOf2;
 y = [x, zeros(1, N1 - N)];
 end
```



```
>> zeropadding([1 0 0 0 0])

ans =

     1     0     0     0     0     0     0     0

>>
```

Fig. 12: Test for **zeropadding.m**

Then we can use our **fftstage.m** to compute the FFT of lengths N, where $N = 1, 2, 3 \ldots$.

Analysis:

Though zero padding means more computation, it has fewer effect on complexity than the decrease of the number of multiplications that FFT provided. At the same time, zero padding can increase the frequency resolution to make the DFT image more closely resemble the actual DTFT image. So it is worthy using zero padding to expand the signal to implement FFT.

IV. Conclusion

During this lab, we study that we can use function to shift DFT frequency domain and change the scale of the horizontal axis from k to ω of DFT plots. By zero-padding, we can increase the frequency resolution to make the DFT image more closely resemble the actual DTFT image. We researched how to using the method of divide and conquer to recursively implement the Fast Fourier Transform (FFT), which could lead to a extremely decrease the number of multiplications. Also we learned how to use zero padding to expand the signal to the form of $N = 2^p$ to compute the FFT of lengths N, where $N = 1, 2, 3 \ldots$. In this lab, we tried a method to **combine** mathematical derivation and MATLAB programming to study ot plot the results of DFT and FFT, which helps us to understand the computation of DFT and FFT **theriotically** and **practically**.