3.1 N-point FFT

- **실습** 1.2.3절에 설명한 시분할 알고리즘을 이용해 N-point FFT 함수를 작성하라. (m-file function으로 구현할 것)
  - 입력
    - x : 이산신호 $x[n]$, 신호의 길이는 N, $n = 0, 1, \cdots, N-1$
  - 출력
    - f_hat : 이산주파수 $\hat{f}$, $\hat{f} = 0, \frac{1}{N}, \frac{2}{N}, \cdots, \frac{N-1}{N}$
    - Xk : 스펙트럼 X_k, 복소수이며 길이는 N
    - N_mult : 곱셈 연산의 횟수

```
function [ F_hat,X_k,N_mult,N ] = my_fft_2( x )
N = length(x);
n = 0 : N-1;
New_index = n;
New_index = dec2bin(New_index,log2(N));
New_index = fliplr(New_index);
New_index = bin2dec(New_index);
Temp_X = x(New_index+1);
N_mult = 0;
for stage = 1:log2(N)
    Temp_out=[];
    w_N = 2^stage;
    w_n = 0:((w_N)/2)-1;
    W2 = exp(-1j*2*pi*w_n/w_N);
    A = 1:2^(stage-1);
    for T = 1 : (N/2^stage)
        Not_w = ((2^stage)*(T-1)) + A;
        w = Not_w + 2^(stage-1);
        Temp_A = Temp_X(w).*W2;
        Temp_A = [Temp_X(Not_w ) + Temp_A, Temp_X(Not_w ) - Temp_A];
        N_mult = N_mult + length(w);
        Temp_out = [Temp_out,Temp_A];
    end;
    Temp_X = Temp_out;
end
X_k = [Temp_X((N/2)+1:end) , Temp_X(1:(N/2))];
F_hat = linspace(-0.5,0.5,N+1);
F_hat = F_hat(1:end-1);
end
```

샘플 개수가 2^q 인 신호를 입력으로 받아서 point 개수와 해당하는 구간을 정의해줍니다.

강의자료에 있는 재배열 알고리즘을 구현하는 부분입니다. New_index를 n으로 초기화해주고 log2(N) 비트로 표현되는 2진수로 변환, 비트 반전 후 다시 10진수로 변환해줍니다. 주의할 부분은 log2(N)비트로 정의해주지 않으면 똑같은 10진수 2가 반전 시 다른 값이 나오게 됩니다.
ex) 0010 -> 0100 , 10->01

강의자료의 버터플라이 알고리즘을 그대로 구현하려고 했습니다. 각 단계를 구현한 외부 for문, 회전인자 정리를 구현한 내부 for문이 있습니다. 시분할 선도를 보면 직전 단계 Xk(n)에서 다음 단계 Xk(n+1)로 이동할 때 회전인자가 곱해지는 원소와 곱해지지 않는 원소가 있습니다. 이러한 원소들의 인덱스 규칙을 찾아서 내부for문에서 구하고 회전인자 정리의 특성을 이용하여 덧샘과 뺄셈으로 곱셈연산 횟수를 줄였습니다. N_mult 부분에서 곱셈연산의 횟수를 세어주는데 내부포문 한 번에 w 벡터와 직전단계 Xk 가 곱해지는 횟수를 세줍니다.
ex)8-point fft 기준으로 계산해보면
- 1단계에서는 w벡터의 길이는 1개, 내부for문의 반복 횟수는 4회 이기 때문에 N_mult = 4 가 됩니다.
- 2단계에서는 w벡터의 길이는 2개, 내부for문의 반복 횟수는 4회 이기 때문에 N_mult = 이전값 + 4가 됩니다.
- 마지막 단계(log2(8)를 거치면 최종 N_mult값은 12가 됩니다.

$\hat{f}$의 범위를 [0~1]에서 [-0.5 ~ 0.5]로 보기 위해 Xk를 y축 대칭으로 이동시킵니다.
-0.5 ~ 0.5를 N+1등분 해주고 0.5만 제거해주면 정확히 0.5를 포함하지않는 N등분이 됩니다.

3.1_1) 주어진 조건으로 함수를 구현한 뒤 DEMO 과정에서 조교님의 추천으로 새로운 알고리즘에 도전하게 되었습니다.

```
1    function [ F_hat,X_k,N_mult,N ] = my_fft_2( x )
2    N = length(x);
3    q = log2(N);
4    MODY = mod(q,1);
5    if(MODY ~= 0)
6        if(MODY >= 0.5)
7            MODY = 1- MODY;
8            Temp_log = q + MODY;
9        else
10           Temp_log = q - MODY;
11       end
12       N = 2^(Temp_log);
13       x = [x,zeros(1,N-length(x))];
14   end
15   n = 0 : N-1;
16   New_index = n;
17   New_index = dec2bin(New_index,log2(N));
18   New_index = fliplr(New_index);
19   New_index = bin2dec(New_index);
20   Temp_X = x(New_index+1);
21   N_mult = 0;
22   for stage = 1:log2(N)
```

- 22번째 줄 아랫부분은 직전에 첨부했던 코드와 동일 합니다.

- 2~4번째 줄에서 입력값의 샘플 개수를 확인 후 log2를 씌워 소수점 부분을 확인합니다.

- MODY변수는 N이 2의 거듭제곱 수 면 0이 되고, 그렇지않으면 소수점 나머지를 같게 됩니다.

- 5~9번째 줄은 가까운 2의 거듭제곱 수만큼 zero-padding 이나 truncation을 진행합니다.
ex) 입력 신호의 길이가 17이면 16-point FFT를 수행, 25이면 32-point FFT를 수행합니다.

- 마지막으로 출력결과에 N을 반환함으로써 만약 2의 거듭제곱 수가 아닌 입력이 들어왔을 때 몇 포인트 FFT를 수행하는지 알 수 있다.

최종적으로 구현한 FFT함수를 이용하여 실행속도를 비교해봤습니다.

N = 256*256;

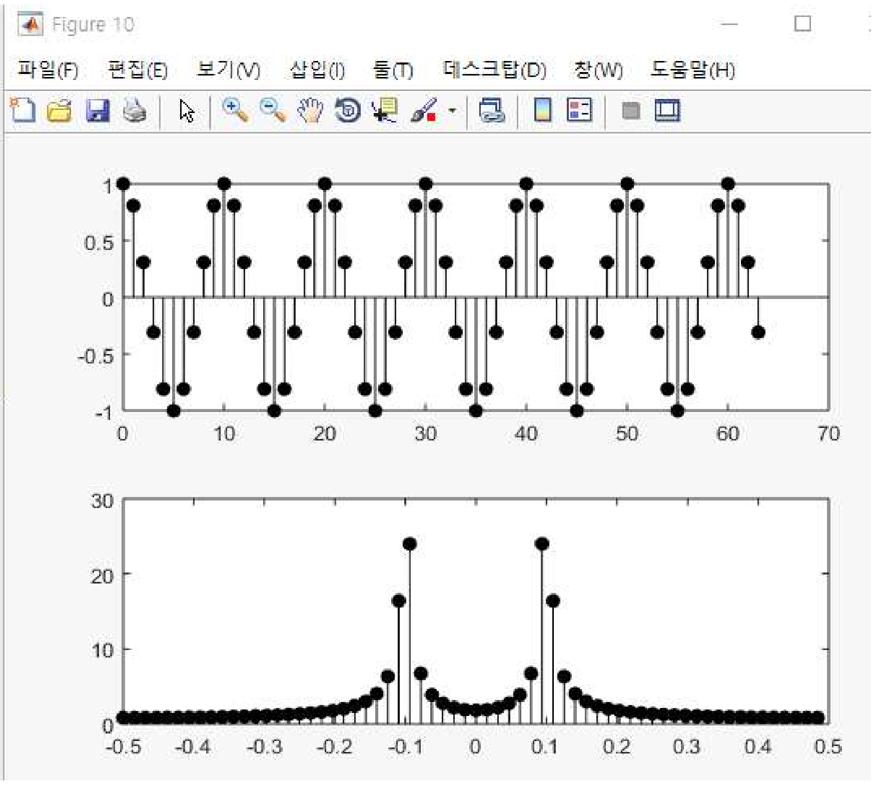
함수 이름	호출	총 시간	셀프 타임*	총 시간 플롯 (짙은 띠 = 셀프 타임)
my_dft	1	112.485 s	112.483 s	
my_fft_2	1	0.485 s	0.390 s	
my_fft	1	3.611 s	0.802 s	

N = 65536 (2의 16제곱) 일 때 각 함수의 실행속도만 측정해보았습니다.
실수로 회전인자를 제대로 고려하지 않았던 my_fft함수는 회전인자까지 제대로 고려해준 my_fft_2보다 성능이 안 좋은 것을 확인 할 수 있습니다.
직접 구현했던 my_dft 함수가 연산량 때문에 112초라는 시간이 소모되는 것과 비교해보면 my_fft_2는 속도를 상당히 빠르게 잘 구현한 것 같다는 생각이 들었습니다.

- **실습** **DEMO** 위에서 구현한 N-point FFT를 이용해 다음 이산신호 $x[n]$의 크기 스펙트럼을 구하고 그래프에 표시하라. (단, 이산주파수 $\hat{f} \in [-\frac{1}{2}, \frac{1}{2}]$에 대해 그려라.) (그림 7 참고)
  - $x[n] = \cos(2\pi \hat{f}_0 n), \; n = 0, 1, 2, \cdots, N-1$
  - $\hat{f}_0 = 0.1, \; N = 64$

```
clc;
clear;
N = 64;
n = 0: N -1;
f = 0.1;
x = cos(2*pi*f*n);

[F_hat,Xk,N_mult,New_N] = my_fft_2(x);
figure(10)
subplot(2,1,1)
stem(n,x,'k','MarkerFaceColor','k');
subplot(2,1,2)
stem(F_hat,abs(Xk),'k','MarkerFaceColor',
```



주어진 조건을 대입하고 my_fft_2에 신호를 대입하여 그대로 그려준 결과입니다.

```
>> N_mult

N_mult =

   192
```

곱셈 연산의 횟수를 측정해보면 위와 같은 결과가 나오는데, $\frac{N}{2} * \log_2(N)$ 공식에 N = 64를 넣었을 때와 같은 결과가 나옵니다.

- 실습 위 신호를 FFT하는데 필요한 곱셈 연산의 횟수는 얼마인가? 신호 $x[n]$을 DFT했을 때 필요한 곱셈 횟수와 비교하라.

이전 실습을 통해 DFT의 곱셈 연산량은 N^2라는 것을 확인했었다. 64-point-DFT의 곱셈 연산량은 수식에 의해 64의 제곱, 즉 4096개가 나오게 된다. 직전에 64-point-FFT의 곱셈 연산량은 N_mult 값으로 확인해본 결과 192개였는데, N = 64 기준으로 FFT보다 DFT 곱셈 연산량이 20배보다 큰 많은 것을 확인할 수 있다.

3.2 DFT와의 비교 – 스펙트럼

- 실습 DEMO 다음과 같은 신호 $x[n]$을 N-point DFT와 N-point FFT를 이용해 크기 스펙트럼을 구하라. (그림 8 ∼ 11 참고)

$$x[n] = 0.3\cos(2\pi \hat{f}_1 n) + 0.8\sin(2\pi \hat{f}_2 n)$$

  - $n = 0, 1, 2, \cdots, N-1$
  - $\hat{f}_1 = 0.1,\ \hat{f}_2 = 0.3$
  - $N = 16,\ 32,\ 64,\ 128$

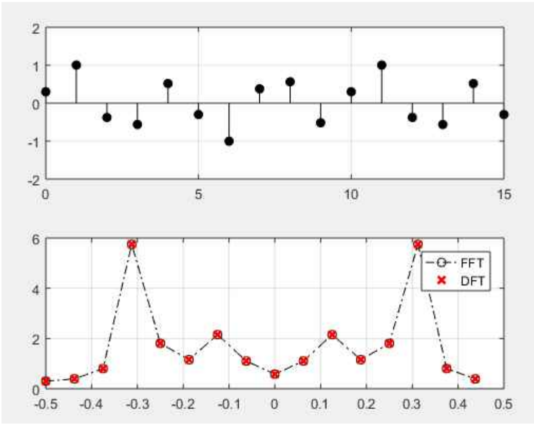
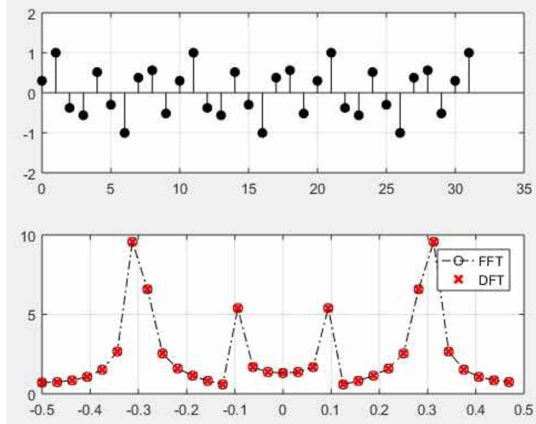
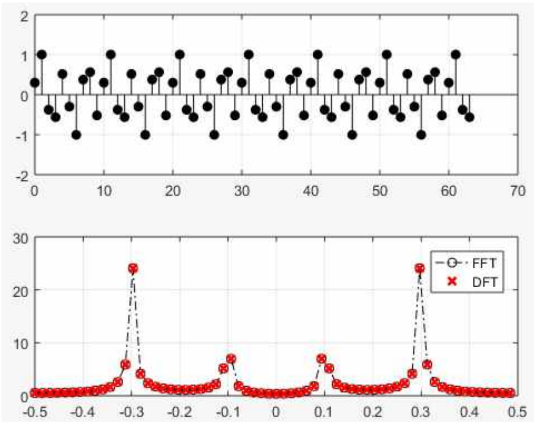
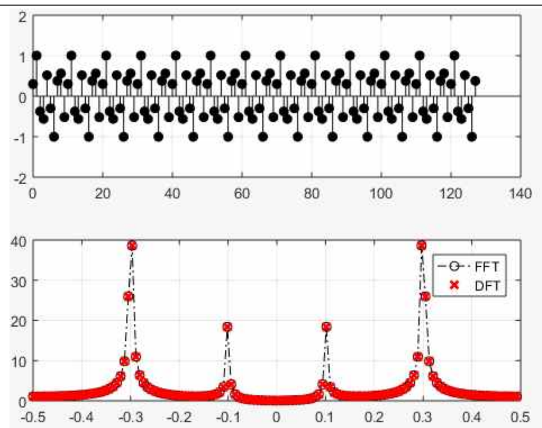
```
clc;
clear;
N = 16;
n = 0: N -1;
f1 = 0.1;
f2 = 0.3;
x = 0.3*cos(2*pi*f1*n) + 0.8*sin(2*pi*f2*n);

[ F_hat_fft,Xk_fft,N_mult_fft_16,New_N_16] = my_fft_2( x);
[ F_hat_dft,Xk_dft, N_mult_dft_16] = my_dft( x );

figure(1)
subplot(2,1,1)
stem(n,x,'k','MarkerFaceColor','k');
grid on

subplot(2,1,2)
plot(F_hat_fft,abs(Xk_fft),'k-.o');
hold on;
plot(F_hat_dft,abs(Xk_dft),'rx','LineWidth', 2);
legend('FFT','DFT')
grid on
```

N값에 변화만 주면서 그래프에 표시해보는 실습이기 때문에 N = 16일 때 코드만 첨부하고 나머지 결과는 다음 페이지에 그래프로만 정리했습니다. my_dft() 함수는 이전 3조(임준호, 주성민)에서 구현했던 함수를 가져와서 사용하였습니다.

N = 16	N = 32
	
N = 64	N = 128
	

네 개의 그래프 모두 DFT와 FFT의 결과가 같은 것을 볼 수 있습니다. N의 값과 상관없이 $\hat{f}$ 이 -0.3, -0.1, 0.1, 0.3 근처에서 0이 아닌 값을 가지고 있고, N이 커질수록 $\hat{f}$의 값이 강조됩니다. N이 증가할수록 $\hat{f}$[-0.5 ~ 0.5] 범위 안에 많은 점이 찍히기 때문에 DTFT의 결과와 비슷해집니다. 또한 두 알고리즘 모두 N이 증가할수록 더해지는 횟수가 많아져서 결과값들의 전체적인 크기가 증가하는 것을 발견했습니다. 다음 과정인 이미지의 압축에서, 시간축 이미지를 주파수축 스펙트럼을 볼 때 스케일링해주는 이유(원인)라고 추측해 보았습니다. 이번 실습을통해 DFT와 FFT는 알고리즘의 차이는 있지만, 결과 자체는 똑같다는 것을 확인하였습니다. 다음 실습 문제를 통해 N값을 비교해보면서 실습을 마치 겠습니다.

3.3 DFT와의 비교 – 연산복잡도

- 실습 DEMO 다음과 같은 신호 $x[n]$을 N-point DFT와 N-point FFT를 이용해 스펙트럼을 구하고 N에 따른 곱셈 연산횟수, FFT와 DFT의 곱셈 연산 횟수의 비율을 측정해 그래프에 표시하라. (그림 12 참고)

$$x[n] = 0.3\cos(2\pi\hat{f}_1 n) + 0.8\sin(2\pi\hat{f}_2 n)$$

  - $n = 0, 1, 2, \cdots, N-1$
  - $\hat{f}_1 = 0.1,\ \hat{f}_2 = 0.3$
  - $N = 16,\ 32,\ 64,\ 128$

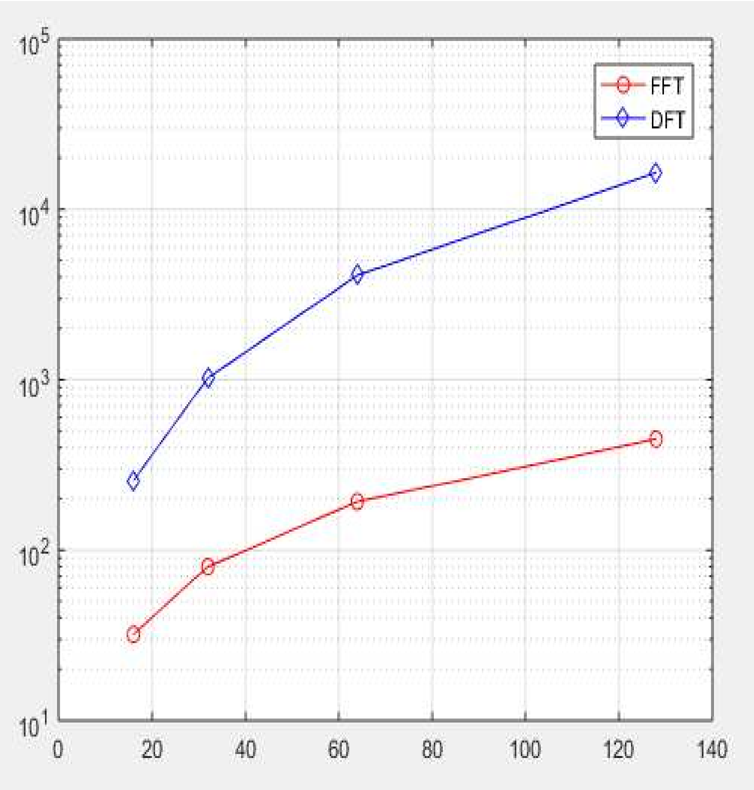
다음은 직전 문제의 소스코드 중 일부입니다.

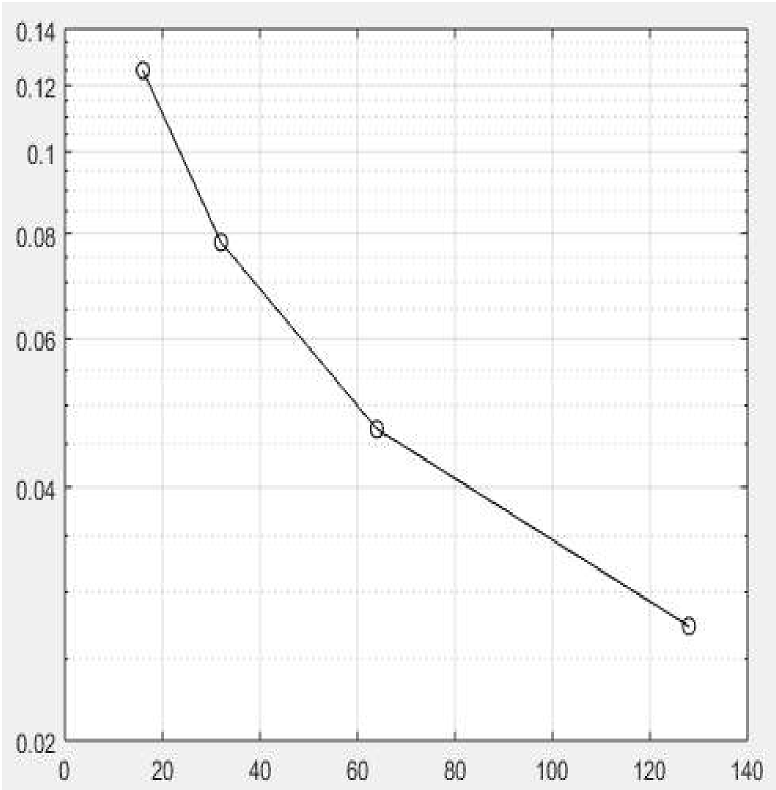
```
[ F_hat_fft,Xk_fft,N_mult_fft_16,New_N_16] = my_fft_2( x);
[ F_hat_dft,Xk_dft, N_mult_dft_16] = my_dft( x );
```

N값이 변화할 때 마다 곱셈 연산의 수를 받아주는 변수 N_mult_fft_x와 N_mult_dft_x를 만들었었습니다. 다음 코드는 위의 변수를 사용하여 3.3번 실습을 수행하는 코드입니다.

```
n_mult = [16,32,64,128];
N_mult_fft = [N_mult_fft_16,N_mult_fft_32,N_mult_fft_64,N_mult_fft_128];
N_mult_dft = [N_mult_dft_16,N_mult_dft_32,N_mult_dft_64,N_mult_dft_128];
figure(5)
semilogy(n_mult,N_mult_fft,'r-o')
grid on
hold on
semilogy(n_mult,N_mult_dft,'b-d')
legend('FFT','DFT')
ratio = N_mult_fft./N_mult_dft;
figure(6)
semilogy(n_mult,ratio,'k-o')
grid on
axis([0,140,0.02,0.14])
```

FFT의 곱셈연산 횟수와 DFT 의 곱셈연산의 횟수들을 하나의 벡터로 합쳐준 뒤 횟수의 비율을 확인했습니다. 다음페이지에 결과가 있습니다.





좌측 그래프는 FFT와 DFT의 곱셈 연산량을 보여주는 그래프이고 우측 그래프는 FFT연산량을 DFT연산량으로 나눠준 값을 보여주는 그래프입니다. 좌측 그래프에서 N값이 커질수록 그래프 간격이 더 벌어지는데, N이 증가할 때마다 FFT의 효율이 DFT보다 훨씬 좋아진다는 것을 알 수 있습니다. 우측의 그래프는 N이 증가함에 따라 분모에 있는 DFT의 곱셈 연산량이 분자에 있는 FFT 곱셈 연산량보다 증가량이 훨씬 커져서, 분수값이 점점 작아지는 것을 볼 수 있습니다. 우측 그래프의 결과값을 ratio라는 변수에 넣었는데 (1 - ratio) x 100을 해주면 N-point-FFT가 N-point-DFT보다 얼마나 더 효율이 좋은지 퍼센트 값으로 알 수 있습니다.