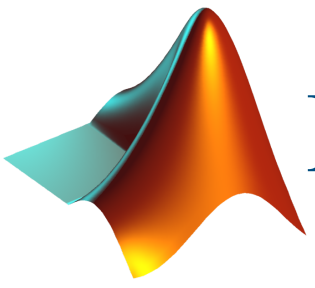
# 임베디드 신호처리 실습

FFT 실습 결과보고서

전자공학부 임베디드시스템

2014146004 김 민 섭



MATLAB

## 1-1 시분활 알고리즘을 이용해 N-Point FFT 함수를 작성하라.

본 함수를 작성함에 있어 입렵과 출력은 다음과 같다.

- **입력**

  x : 이산신호 x[n], 신호의 길이는 N, n= 0, 1, 2, … N-1

- **출력**

  f_hat : 이산 주파수 $\hat{f}, \hat{f} = 0,1/N,2/N,\ldots N-1/N$

  XK : 스펙트럼 $X_k$ , 복소수이며 길이는 N

  N_Mult : 곱셈 연산의 횟수

시분활 FFT 알고리즘을 구현하기 위해서 크게 생각해야 할 사항은 3가지 정도 있었다.

1. **입력신호 x[n]의 재배열**
2. **각 연산 단계에 맞는 회전인자 생성**
3. **시분활 선도에 맞도록 구현되는 알고리즘 구현**

위 세가지가 큰 해결과제였다. 그중 1번과 2번은 순조롭게 진행된 방면 3번 과정은 시간이 다소 투입되는 과정이었다. 위 사항들을 어떻게 해결했는지 알아보기 이전에 초기 설정 코드부터 살펴보도록 하자. 초기 설정과 관련된 코드는 다음과 같다.

```
function [f_hat , Xk,N_mult] = myfun_N_Point_FFT2(x)

x_len = length(x);
bit_value= log2(x_len);
N_to_order = 0 : 1:(x_len-1);
f_mid =  [zeros(1,x_len)];
X_mid=zeros(bit_value+1,x_len);
N_mult = 0;
```

N 을 구하기 위해서 입력신호 x의 길이를 구하였다. 그리고 FFT 알고리즘은 $2^q$ 개의 입력이 들어온다. q는 FFT 알고리즘을 구현할때 반드시 필요한 것중 하나이다. 0 부터 시작되는 값들과 계산이 있기에 N_to_order 라는 행렬을 생성해 주었다. f_mid 는 f_hat에 들어가기 전의 중간 값이다.
X_mid 도 마찬가지로 Xk 로 들어가기 전의 중간값이다. 하지만 여러 단계가 있는 만큼 2차원 행렬로 만들어 주었다. 변수명 및 초기 설정 값들을 알아보았다. 이제부터 어떻게 FFT를 구현했는지 살펴보자.

### 1. 입력신호 x[n]의 재배열

시분할 알고리즘을 구현하기 위해 가장 먼저 해야 할 일이다. 하지만 알고리즘의 방식의 따라 제일 마지막에 해줄 수도 있다. 재배열을 하는 방법은 십진으로 정리된 순서를 이진수로 변환한 후 이진수를 뒤집는다. 뒤집은 2진수를 다시 10진수로 바꾸면 우리가 원하는 순서를 구할 수 있다.
이제부터 어떻게 구현했는지 코드를 보면서 같이 살펴보도록 하자.

```
bin_num = fliplr(dec2bin(N_to_order,bit_value));
new_num = bin2dec(bin_num);
new_num = rot90(new_num);
```

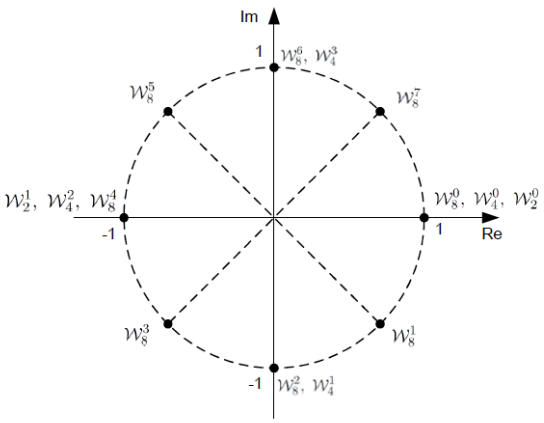
재배열 코드는 MARLAB의 내장 함수를 이용하여서 비교적 간단하게 구현 할 수 있었다.
0~N-1까지의 수를 2진수로 표현해주고 이를 뒤집는다. 그리고 이를 다시 10진수로 바꾸어 준다.
rot90은 90도로 행렬을 돌려주는 건데 내장함수를 사용한 후 결과가 열로 저장이 되기 때문에 필자가 사용하기 쉬운 백터로 바꾸어 준 것이다. 이로서 입력신호 x[n]의 재배열을 손쉽게 진행 할 수 있었다.

**2. 각 연산 단계에 맞는 회전인자 구성.**

어떻게 이 작업을 진행할지에 앞서 FFT의 전신인 DFT 식을 한번 살펴보자.

$$X_k = \sum_{n=0}^{N-1} x[n]e^{-j2\pi(k/N)n}, k = 0,1,2...N-1$$

위 식은 DFT의 식이다. 여기서의 회전인자는 $e^{-j2\pi(k/N)n}$ 이다. 이를 오일러의 공식을 사용하여 변환하면 다음과 같다. $e^{-j2\pi(k/N)n} = cos(2\pi nk/N) - jsin(2\pi nk/N)$ 이는 실수부와 허수부로 나누어서 2차원 좌표계에 표시할 수 있는데 N에 따라서 다음과 같이 표현 할 수 있다.




N = 2, 4, 8 일때의 회전인자의 위치이다. 0 ≦ w < $\pi$ 만 보았을때 부호만 바꾸어 주면 같은 값들이 나오는 것을 볼 수 있다. 이를 이용하면 회전인자는 절반만 구하고 부호만 바꾸어 주면 된다.
이제 어떻게 구현하였는지 코드를 살펴보도록 하자.

```
W_turn = zeros(bit_value,(x_len/2));
for col = 1:bit_value
    for row = 1:2^(N_to_order(col))
    W_turn(col, row) = exp(-1i*((2*pi)/(2.^col))*N_to_order(row));
    end
end
```

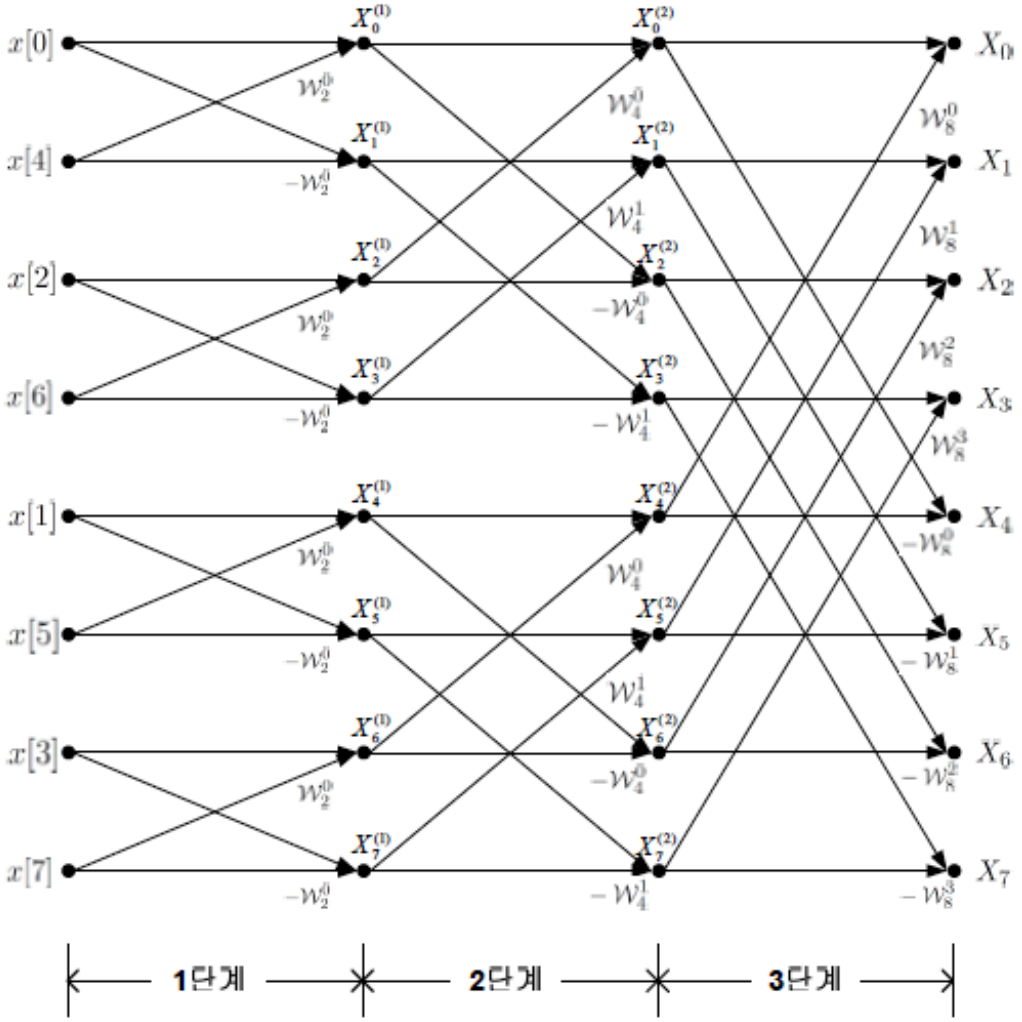
먼저 w_turn 이라는 2차원 영행렬을 만들어 준다. 비트값만큼의 열을 만든다. 이는 시분활 선도의 알고리즘에서 각 단계별 회전인자가 다르기 때문이다. 그리고 총길이의 절반 만큼의 행을 만든다.
이 방법은 데이터를 다소 낭비하긴 하지만 직관적인 표현이므로 사용하였다.그 이후 비트별 회전인자를 이중 포문을 이용하여서 구해주었다.다은은 N = 8 일때의 회전인자의 모습들이다.

```
W_turn =

   1.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   1.0000 + 0.0000i   0.0000 - 1.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   1.0000 + 0.0000i   0.7071 - 0.7071i   0.0000 - 1.0000i  -0.7071 - 0.7071i
```

위의 열부터 N = 2, 4, 8 일때의 회전인자가 구해졌다. 아까 말했듯이 마지막 단계인 N =8 일때를 최종 기준으로 만든 것이다 보니 1,2열에서 0의 값만 있는 데이터 낭비 부분이 존재한다. 구해진 회전인자에 대해서 음수를 취해주면 전체 회전인자를 구할 수 있다. 위와 같은 방식으로 각 연산 단계에 맞는 회전인자를 구현하였다.

### 3. 시분활 선도에 맞도록 구현되는 알고리즘 구현

이 부분을 가장 해결하는데 오랜 시간이 소모되었다. 먼저 시분활 선도를 한번 살펴보자.




N =8 일때를 기준으로 하여 만든 시분활 선도이다. 1단계부터 살펴보자. 재배열된 입력신호가 $2^1$을 기준으로 묶여있다. 그리고 묶여진 것의 처음 $2^0$ 개 까지는 회전인자가 곱해지지 않은 채로 출력값에 더해지고 다음번의 출력값에도 더해진다. 그리고 $2^0$개 이후의 것들은 음의 회전인자가 곱해진 채로 다음 출력값에 더해지고 양의 회전인자가 곱해진건 이전 출력값에 더해진다. 이런 규칙성을 가지고 1단계에서는 $2^1$ 개로 묶고 $2^0$를 기준으로 다른 연산을 수행하게 된다. 즉 N 단계에서는 $2^N$개로 묶여지고 $2^{N-1}$를 기준으로 다른 연산을 수행하게 된다. 그리고 2단계부터는 회전인자의 순서도 달라지게 되는데 코드를 보면서 한번 더 이해해 보도록 하자.

```
for stage=1 : bit_value
    w_value = 1 : 1 : 2.^(stage-1); %이는 단계별 회전인자에 따른 백터이다.
    In_To_W = 1;                    %어떤 회전인자가 들어가야 하는지를 알려주는 지표이다. 초기값은 1로 설정했다.
    for k =1 : ( x_len - 2.^(N_to_order(stage)))
        if or((mod(k,2.^(stage)) > (2.^stage)/2),(mod(k,2.^(stage)) == 0))
        else
            same_W = W_turn(stage,w_value(In_To_W))*X_mid(stage,k+2.^N_to_order(stage));
            X_mid(stage+1,k) = X_mid(stage,k)+ same_W;
            X_mid(stage+1,k+2.^N_to_order(stage)) = X_mid(stage,k)- same_W;
            N_mult = N_mult +1;
        end
    In_To_W = In_To_W +1;
    if In_To_W >2.^(N_to_order(stage))   In_To_W = 1;   end
    end
end
```

본 코드는 총 2중 for 문으로 구성되어 있다. 첫번째 for 문은 단계를 나타내고 두번째 for 문은 각 단계별 연산을 진행한다. 각 단계별 연산할 시에 마지막 2.^(N_to_order(stage)) 번째 이후에는 필요 없으므로 for 문은 x_len - 2.^(N_to_order(stage)) 까지 돌려주었다. 우리는 $2^N$ 개씩 묶은 것 중 절반만 연산하면 나머지 절반의 값은 덧셈으로 구할 수 있다는 것을 이용하였다.

or((mod(k,2.^(stage)) > (2.^stage)/2),(mod(k,2.^(stage)) == 0)) 를 풀이해보면 $2^N$ 묶은 절반 이후의 값들을 의미한다. 그래서 우리는 이 값들 이외의 것들을 연산해 주기로 하였다.
연산의 위의 코드대로 진행하면 되며 한번 진행할 때마다 곱하기 연산 횟수를 세어준다.
다소 복잡하지만 규칙성을 찾으면서 시분활 선도에 맞도록 구현되는 알고리즘을 구현할 수 있었다.

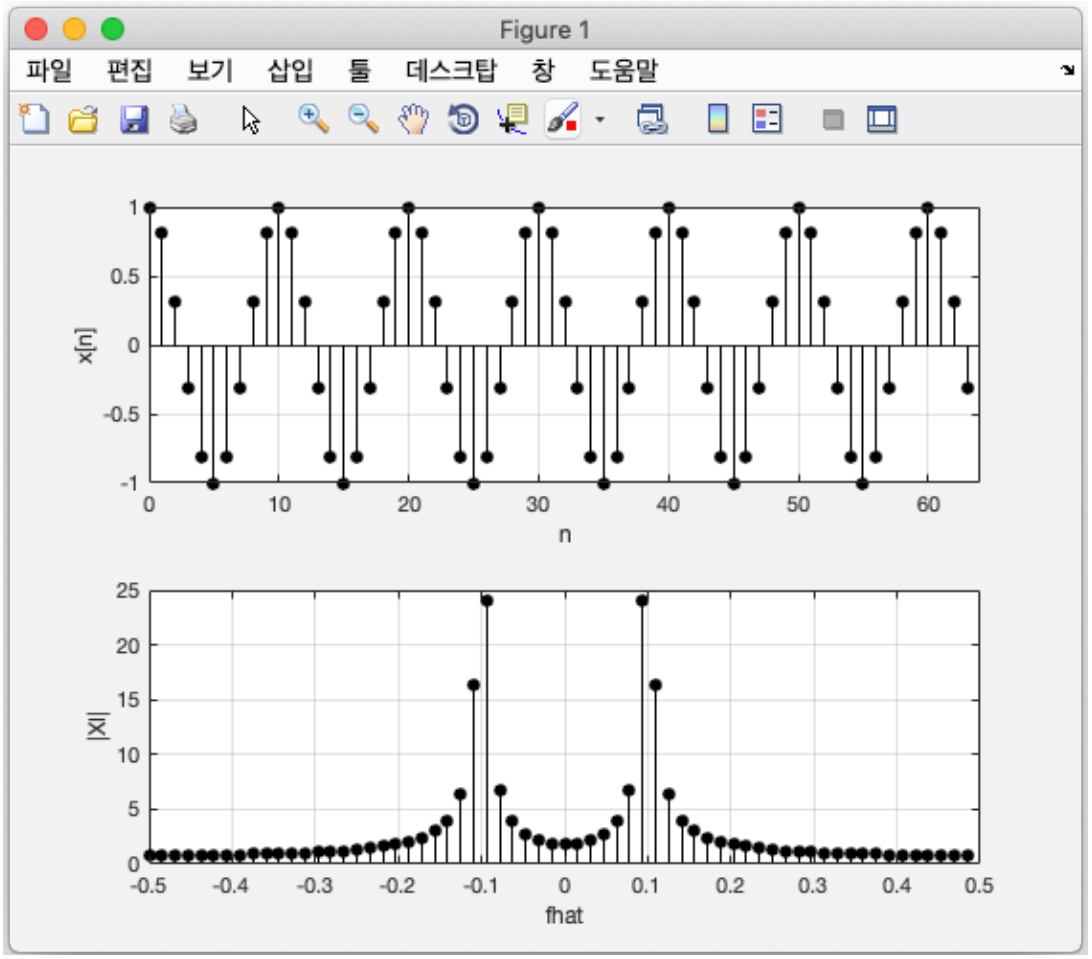
1. **입력신호 x[n]의 재배열**
2. **각 연산 단계에 맞는 회전인자 생성**
3. **시분활 선도에 맞도록 구현되는 알고리즘 구현**

우리는 위 세가지 것을 차례대로 해결해 나아가면서 FFT 시분활 알고리즘을 구현 할 수 있었다.
전체적인 코드는 다음과 같다.

```
function [f_hat , Xk,N_mult] = myfun_N_Point_fft3(x)

x_len = length(x);
bit_value= log2(x_len);
N_to_order = 0 : 1:(x_len-1);
f_mid =  [zeros(1,x_len)];
X_mid=zeros(bit_value+1,x_len);
N_mult = 0;

W_turn = zeros(bit_value,(x_len/2));
for col = 1:bit_value
    for row = 1:2^(N_to_order(col))
    W_turn(col, row) = exp(-1i*((2*pi)/(2.^col))*N_to_order(row));
    end
end

new_num = rot90(bin2dec(fliplr(dec2bin(N_to_order,bit_value))));

for i = 1 : x_len
    In_To_x = new_num(i)+1;
    X_mid(1,i) = x(In_To_x);
end

for stage=1 : bit_value
    w_value = 1 : 1 : 2.^(stage-1);
    In_To_W = 1;
    for k =1 : ( x_len - 2.^(N_to_order(stage)))
        if or((mod(k,2.^(stage)) > (2.^stage)/2),(mod(k,2.^(stage)) == 0))
        else
           same_W =
W_turn(stage,w_value(In_To_W))*X_mid(stage,k+2.^N_to_order(stage));

           X_mid(stage+1,k) = X_mid(stage,k)+ same_W;
           X_mid(stage+1,k+2.^N_to_order(stage)) = X_mid(stage,k)- same_W;
           N_mult = N_mult +1;
        end
    In_To_W = In_To_W +1;
    if In_To_W >2.^(N_to_order(stage))   In_To_W = 1;   end
    end
end

for k =1 :  x_len
    if k < (x_len/2 +1)
        f_mid(k) = N_to_order(k)/x_len;
    else
        f_mid(k) = N_to_order(k)/x_len-1;
    end
end
Xk = [zeros(1,x_len)];
for a=1 :x_len
    Xk(a)= X_mid(bit_value+1,a);
end
f_hat =f_mid;
```

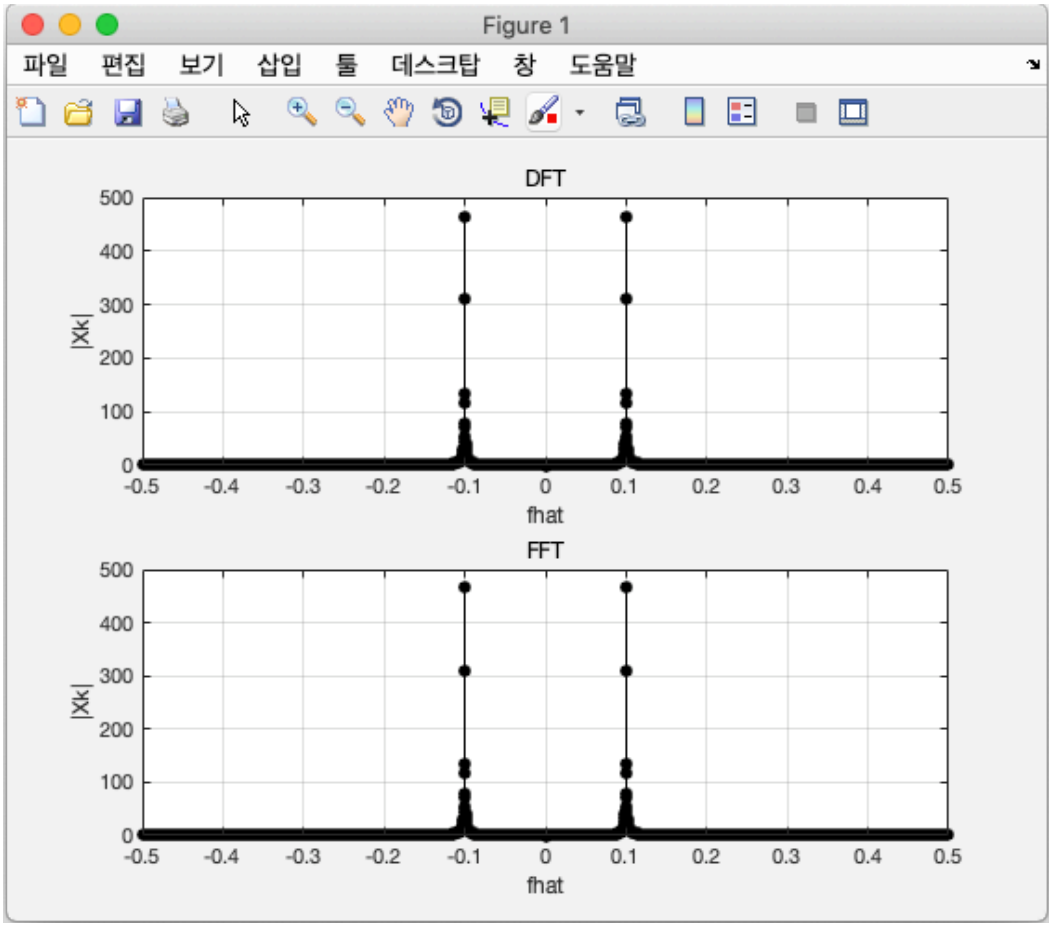
**1-2 위에서 구현한 N - Point FFT를 이용해 다음 이산신호 x[n]의 크기 스펙트럼을 구하고 그래프에 표시하라.**

- $x[n] = 0.3cos(2\pi\hat{f}_0 n), n = 0,1,2,...N-1$
- $\hat{f}_0 = 0.1, N = 64$



## 1-3 위 신호를 FFT 하는데 필요한 곱셈 연산 횟수는 얼마인가? 신호 x[n]을 DFT 했을 때 필요한 곱셈 횟수와 비교하라.

이론적으로 N-Point FFT의 곱셈연산 횟수는 $(N/2)log_2N$ 회 나오게 된다. DFT의 곱셈 연산 횟수는 $N^2$번 나오게 된다. 이 이론이 실제로 맞는지 MATLAB 을 활용하여서 비교해 보았다.




먼저 1.2의 신호 x[n]을 4096-Point DFT와 FFT의 결과값이다. 예상대로 두 신호의 결과 값은 동일하다. 곱셈 연산 횟수를 살펴봤더니 확연한 차이가 있었다.

```
N_mult1_DFT =     N_mult =

    16777216           24576
```

곱셈 연산 한 결과값은 DFT 가 FFT 보다 훨씬 많은 곱셈 연산 횟수를 진행했어야 했다.
이론대로 $N^2$,$(N/2)log_2N$ 번의 곱셈 연산이 진행된 모습을 살펴 볼 수 있었다.
곱셈연산 횟수가 실행시간에 미치는 영향도 컸다. 다음은 DFT와 FFT의 실행시간이다.

| | | | | |
|---|---|---|---|---|
| myfun_N_Point_DFT | 1 | 3.869 s | 3.869 s | |
| myfun_N_Point_FFT2 | 1 | 0.065 s | 0.057 s | |

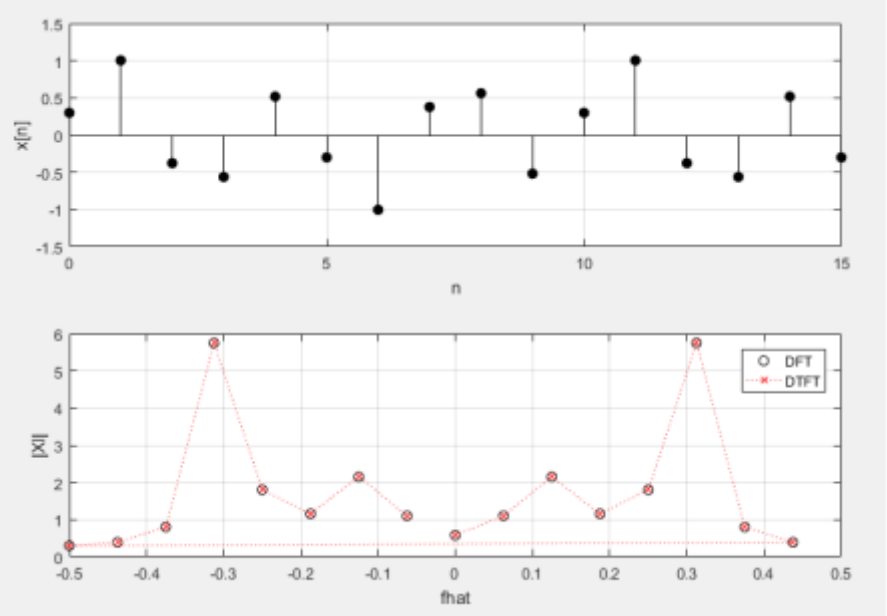
실행시간 역시 확연한 차이가 일어났다. FFT가 확실히 실행시간이 많이 단축되는 모습을 살펴 볼 수 있었다. 이로서 실제로 주파수 영역에서 관찰하거나 무엇인가를 조정하려 할때 DFT 보다 FFT를 사용하는것이 훨씬더 빠르고 효율적이라고 할 수 있다.

**2-1 다음 신호 x[n]을 N-point DFT와 N-point FFT를 이용하여 스펙트럼을 구하여 비교해라.**

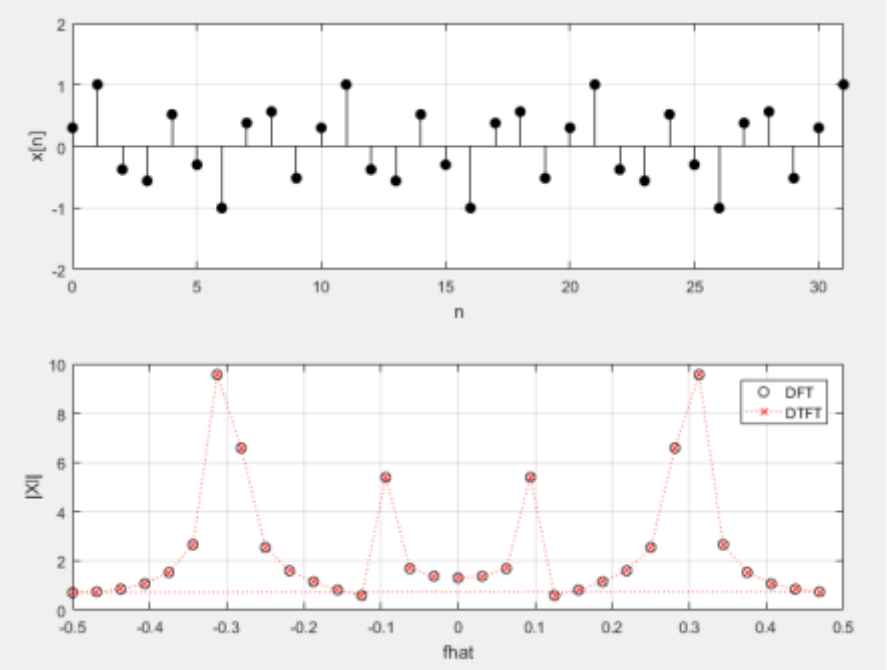
$$x[n] = 0.3\cos(2\pi\hat{f}_1 n) + 0.8\sin(2\pi\hat{f}_2 n)$$

$\hat{f}_1$ = 0.1, $\hat{f}_2$ =0.3 N = 16, 32, 64, 128

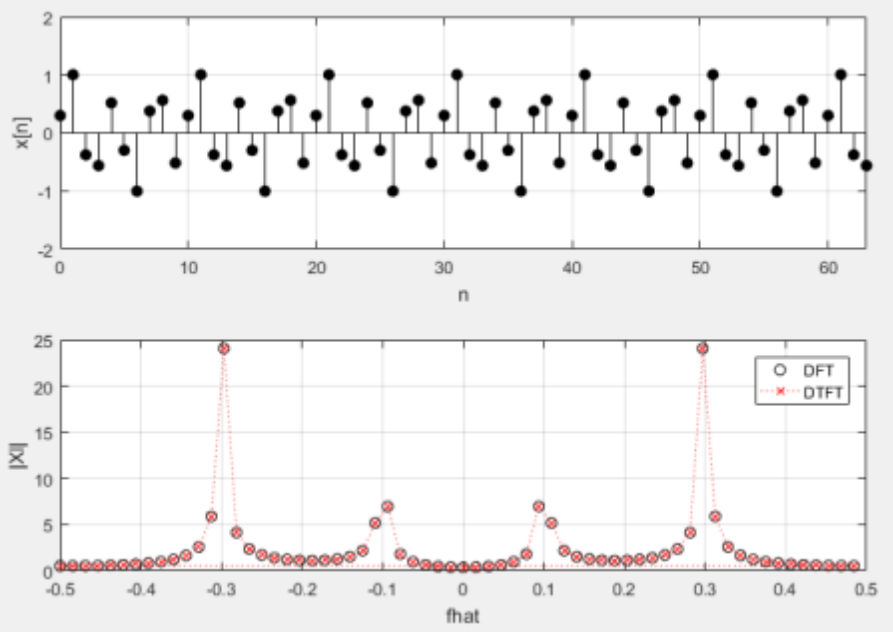
N=16 일 때의 16-point DFT와 FFT 그래프



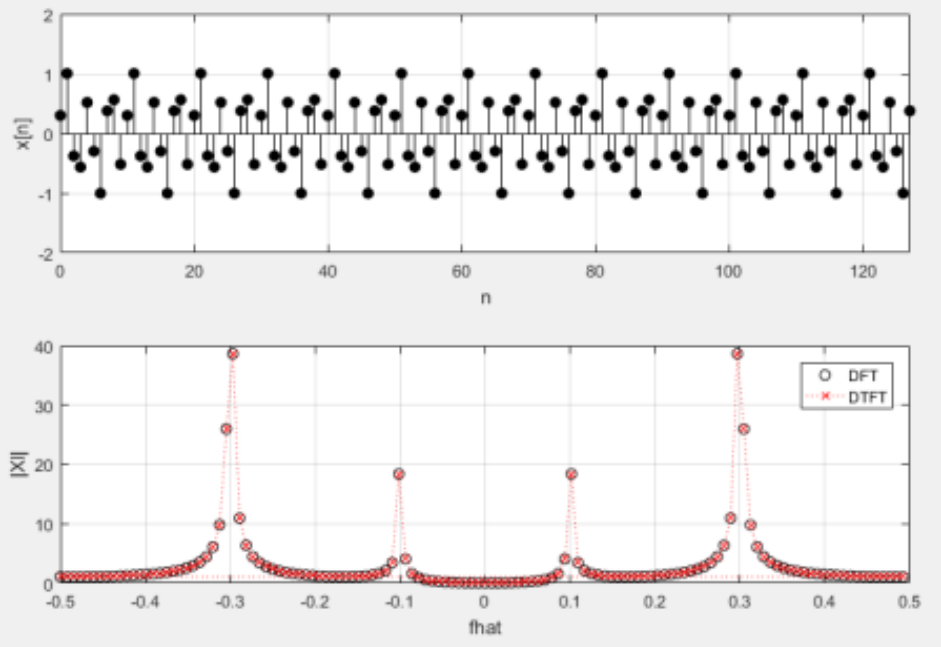

N=32 일 때의 32-point DFT와 FFT 그래프



N=64 일 때의 64-point DFT와 FFT 그래프



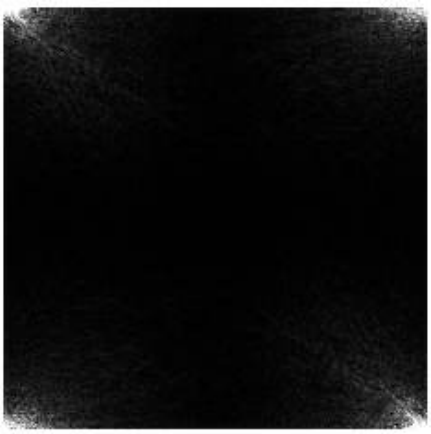

N=128 일 때의 128-point DFT와 FFT 그래프




x[n]의 함수를 N-point DFT, N-point FFT 한 결과이다. FFT는 DFT에 비해 곱셈연산 횟수를 줄인 방법이기 때문에 전체적으로 봤을 때 FFT결과 값이랑 DFT 결과 값이랑 완벽히 일치한다. 또한 N이 커질수록 기존 주어진 함수에 근접하는 모습을 보이고 있다. N = 16 일 때의 그래프를 보면 값이 듬성듬성 주어져서 제대로 된 모양을 갖추지 못한다. 하지만 N = 128까지 증가함에 따라 같은 영역 안에 전 보다 많은 값을 받아와서 원하는 모양에 근접하는 모습을 볼 수 있다.

## 3. DFT와 DCT의 상호관계

우리가 배우는 DFT와 DCT 시간영역에서의 이산신호를 주파수 영역에서 이산신호로 변환하여 보여준다는 공통점이 있다. DFT는 여러 푸리에 변환 중 DTFT 변환을 먼저 해서 이산신호를 푸리에 변환하여 주파수 영역으로 나타내준다. 이때 주파수 영역에서 결과로 연속신호가 나오는데 이산신호로 바꿔주기 위해 샘플링을 추가로 한번 더 해준다. 여기까지가 DFT의 과정이다. DFT는 시간 축에서의 이산신호를 주파수영역에서 이산신호로 나타내준다. DCT는 Descrete Cosine Transform으로 말 그대로 서로 다른 주파수를 가잔 코사인함수들의 합으로 구성되어진 이산변환이다. DCT는 JPEG의 손실 있는 압축 등 여러 영상처리 활용에 쓰인다. 유독 영상처리에 이 DCT가 자주 쓰이는 이유는 DCT한 결과값이 모두 실수로 나오기 때문이다. DFT인 경우를 생각해 보자. DFT변환의 식을 보게 되면 매번 $e^{-j2\pi nk/n}$이 각 x[n]에 곱해지고 곱한 값들을 더하는 방식이다. $e^{-j2\pi nk/n}$을 오일러 공식을 사용해 풀어보면 $e^{-j2\pi nk/n}$는 $\cos(-2\pi\frac{nk}{N}) + jsin(-2\pi\frac{nk}{N})$ 와 같은 식이다. 우측 sin()함수에 복소수 j가 붙어있는 것을 볼 수 있다. 즉 DFT는 매 계산마다 복소수를 포함한 식을 곱해주고 더해줌으로써 결과 값도 복소수 값이 나오게 된다. 이 복소수 값으로 인해 불필요한 저장 공간을 차지하며 계산을 복잡하게 만든다. 이것에 비해 DCT는 모든 결과값이 DFT에서 sin함수 부분, 즉 복소수 부분만 제거하고 남은 실수 부분에 상수 배해준 실수 값이다. 아래의 두 사진은 왼쪽이 주파수 영역에서의 DFT 결과값, 오른쪽이 DCT결과값이다.

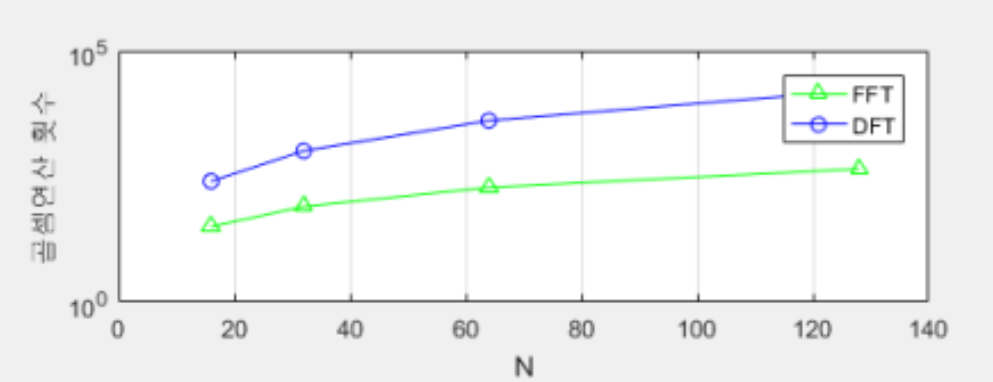


두 사진의 다른 점은 어두운 영상에서 밝은 부분이 한쪽 모서리에만 있는 것과 네 개의 모서리에 있다는 것이다. DFT변환의 결과값이 4개 모서리에서 밝은 부분이 생기는데 이는 결과값 안의 복소수 값 때문이다. DCT변환은 복소수 값을 배제하고 계산하였기에 훨씬 용이하다고 볼 수 있다. DCT 변환은 DFT변환과 비슷한 푸리에 관련 변환이고 코사인함수가 우함수이기 때문에 DCT함수 결과값도 우함수가 나오게 된다. 이러한 우함수 대칭 성질 때문에 DCT변환은 DFT변환의 2배 길이로 표현할 수 있다. 단, 다른 점이라면 결과값이 실수로 나온다는 것인데 이 특성 때문에 현재까지도 영상처리뿐만 아니라 여러 방면에서 DCT변환 방식을 많이 쓰고 있다.

## 3-1 다음과 같은 신호 x[n]을 N-Point DFT, FFT를 이용해 스펙트럼을 구하고 N에 따른 곱셈 연산횟수, FFT 와 DFT의 곱셈연산 횟수의 비율을 측정해 그래프에 표시하라.

N-point DFT의 수식은 다음과 같다. $X_k = \sum_{n=0}^{N-1} x[n]e^{-j2\pi(k/N)n}, k = 0,1,2...N-1$

k가 0부터 N-1까지 총 N번 바뀜에 따라서 n도 0부터 N-1까지 바뀌면서 연산을 하게 된다.한번 연산할 때 곱셈이 한번 실행되기 때문에 총 곱셈 연산 횟수는 N*N 이다. N-point FFT의 경우는 N-point FFT 선도를 그려서 계산을 할 경우 행의 개수는 N개, 열의 개수는 $log_2N$+ 1개다.곱셈 연산은 열과 열 사이에서 실행이 되어 총 곱셈 연산 횟수는 (행의 개수) * (열의 개수 -1) 인$Nlog_2N$이다. 여기서 곱해지는 회전인자의 대칭성을 이용하게 되면 반은 그대로 연산하고 나머지 반은 (-)만 붙이게 되면 된다. 결과적으로 연산횟수가 반으로 줄어들게 되어 총 곱셈 연산 횟수는 $N/2log_2N$이 된다. N-point DFT와 N-point FFT의 곱셈 연산 횟수를 비교 했을 때 아래 그래프와 같이 N*N인 DFT 곱셈횟수에 비해 $N/2log_2N$ 인 FFT 곱셈횟수가 더 적다.
곱셈연산 횟수의 비율 rate = (FFT의 곱셈횟수 / DFT의 곱셈횟수) 라고 하면




rate = $\frac{\frac{1}{2}Nlog_2N}{N^2} = \frac{\frac{1}{2}log_2N}{N}$ 이 된다. $N = 2^q$ 라고 하면 $\frac{\frac{1}{2}log_2 2^q}{2^q} = \frac{q}{2^{q+1}}$ 이 된다.

아래 그래프를 봐도 알 수 있듯이 N값이 커질수록 rate 값은 점점 작아지게 된다.
결과적으로 FFT와 DFT의 곱셈 횟수 비율만 봤을 때도 FFT의 곱셈 연산 횟수가 DFT의 곱셈 연산 횟수보다 N이 커질수록 줄어드는 것을 알 수 있다.