

---

고속 푸리에 변환 알고리즘의 이해와 이용

Understanding and Using the Fast Fourier Transform Algorithm

---

**작성자** 정지혁  
202102934  
인천대학교 정보통신공학과  
jihyuk2291@naver.com  
010-2291-3052

**참고** 본 글은 인천대학교 컴퓨터 알고리즘  
강의 中 개인의 목적을 위하여 만들  
어짐.

---

## 차 례

### 1 FFT Algorithm의 설명

1.1 Fourier Transform을 왜?	1
1.2 Discrete Fourier Transform	
1.3 Fast Fourier Transform	2
1.4 FFT의 다양한 알고리즘	2

### 2 FFT 알고리즘의 구현.

2.1 알고리즘 구현	6
2.2 알고리즘 성능 평가	18

### 3 FFT 알고리즘의 적용

3.1 알고리즘의 적용	19
--------------	----

### 4 결론

# I FFT Algorithm 이란?

## Fourier Transform 을 왜?

푸리에 변환은 임의의 입력 신호를 다양한 주파수를 갖는 주기 함수들의 합으로 나타내어 준다.

일반적으로 시간 도메인에서 모든 시간대의 정보를 표현할 수 없다. 하지만 시간 영역의 정보를 주파수 영역의 정보로 변환시킨다면 주파수는 전 구간에서 해석이 가능하기 때문에 정보의 해석과 분석이 용이하게 된다. 또한, 시간 도메인에서 미분은 주파수 영역에서 주파수와의 곱셈이 되고, 컨볼루션은 평범한 곱셈이 된다. 이런 경우에는 정보를 푸리에 변환하여 연산을 하고, 다시 역변환하는 식으로 연산을 더 쉽게 처리 할 수 있다.

정보 신호를 푸리에 변환 하더라도 단점이 존재하는데, 시간에 대한 연속성이 고려되지 않으며, 신호에 고주파 성분이 포함되어 있을 경우에 신호의 특징 분석이 어려워진다는 것이다.

	Finite	Infinite
Discrete time domain	Discrete Fourier Transform (DFT) $X(k) = \sum_{n=0}^{N-1} x(n)e^{-j\omega_k n}$ $k = 0, 1, \dots, N-1$	Discrete Time Fourier Transform (DTFT) $X(\omega) = \sum_{n=-\infty}^{+\infty} x(n)e^{-j\omega n}$ $\omega \in (-\pi, +\pi)$
Continuous time domain	Fourier Series $X(k) = \int_0^T x(t)e^{-j\omega_k t} dt$ $k = -\infty, \dots, +\infty$	Fourier Transform $X(\omega) = \int_{-\infty}^{+\infty} x(t)e^{-j\omega t} dt$ $\omega \in (-\infty, +\infty)$

다음과 같이 이산 시간 도메인과 연속 시간 도메인으로 구분 나눌 수 있다. 보통 정보를 A/D 변환하여 이산 시간 도메인(Discrete time domain)에서 정보를 해석한다.

## Discrete Fourier Transform

FFT 는 결국 컴퓨터 구조와 효율에 맞게 DFT 를 효율적으로 운용하는 알고리즘이다.

FFT 에 대해 알기 전에 DFT 를 먼저 알아보자.

컴퓨터는 이산 신호만을 처리 할 수 있고, 신호 구간이 무한할 수 없다. 때문에 유한 구간의 이산 신호의 Fourier Transform 을 해야한다. 일반적인 Fourier Transform 과 다르게  $\pm\infty$  적분 형태의 무한 적분이 아닌 유한 합으로 대체된다.

$$X(\omega_k) = \sum_{n=0}^{N-1} x(t_n)e^{-i\omega_k t_n}, \quad k = 0, \dots, N-1$$

DFT 공식 (출처: Wiki백과)

위의 식은 DFT 의 식이다. 푸리에 변환보다 수학적으로 간단하고 실제 계산과 관계가 깊다. 위에서 N 은 정보 샘플의 개수이다. 현재에는 컴퓨터의 성능이 좋아져 DFT 와 FFT 의 경계가 많이 허물어 졌다. 과거, 변환 길이 N 의 증가로 계산 량이 급증했고, 이를 해결하기 위해 제안된 것이 고속 푸리에 변환(Fast Fourier Transform) 이다. 위의 식에서 보면 알다시피, DFT 알고리즘을 프로그래밍 한다면  $O(N^2)$ 의 시간복잡도를 가지게 된다.

\*샘플 시간에서의 입력 시간의 진폭  $x(t_n)$

과 회전인자  $e^{-i\omega_k t_n}$ 의 곱을 (N-1)번 반복한다. 진폭 하나의 계산은, N 개의 진폭 데이터를 이용하여 N 개 항의 곱셈이 필요하다.

## Fast Fourier Transform

DFT 에서 어떤 과정을 통하여 계산량을 줄일 수 있었을까? 바로 회전인자의 주기성 및 대칭성을 이용하여 계산량을 줄이는 것이다. 단위 원을 한바퀴 도는데 필요한 서로 다른 N 만 계산한다 DFT 를 길이가 짧은 여러 DFT 로 계속 분할시켜 곱셈량을 N 에 비례하도록 만드는 것이다. 쉽게 말하자면 샘플링 과정에서 필요한 신호만을 골라내어 빠르게 연산을 하는 것이다.

1942 년 FFT 의 기초가 되는 다닐슨-란츠 보조정리(Danielson-Lanczos Lemma)가 발견되었다.

N(단 짝수인 경우)의 DFT 는 각각의 길이가 N/2 인 두개의 DFT 의 합으로 다시 쓸 수

있다는 것이다. 회전자  $W_N = e^{-2\pi i/N}$  를 이용하여 DFT 수식을 정의하면 다음과 같다.

$$f_k = \sum_{n=0}^{N-1} x_n W_N^{kn} \quad k = 0, \dots, N-1$$

회전자는 N 에 관한 주기성으로 인해 다음과 같은 성질이 있다.

$$\begin{aligned} W_N^{k+N} &= W_N^k \quad \because (W_N^{k+N}) = e^{-\frac{2\pi i}{N}(k+N)} = e^{-\frac{2\pi i}{N}k} e^{-2\pi i} = e^{-\frac{2\pi i}{N}k} (1) = W_N^k \\ W_N^{k+N/2} &= -W_N^k \quad \because (W_N^{k+N/2}) = e^{-\frac{2\pi i}{N}(k+\frac{N}{2})} = e^{-\frac{2\pi i}{N}k} e^{-\pi i} = e^{-\frac{2\pi i}{N}k} (-1) = -W_N^k \\ W_a^b &= W_{na}^{nb} \quad n: \text{자연수} \end{aligned}$$

이를 이용하여 다음과 같이 회전자를 볼 수 있다.

$$W_4^6 = W_4^2, \quad W_2^1 = W_4^2 = W_8^4, \quad W_4^{(k+4/2)} = W_4^k W_4^2 = -W_4^k$$

만약 길이가 N(짝수)인 DFT 는 다음과 같은 방법으로 각각 길이가 N/2 인 두개의 DFT 로 분해가 가능하다.

$$\begin{aligned} f_k &= [x_0 W_N^0 + x_2 W_N^{2k} + x_4 W_N^{4k} + \dots + x_{(N-4)} W_N^{k(N-4)} + x_{(N-2)} W_N^{k(N-2)}] \\ &\quad + [x_1 W_N^k + x_3 W_N^{3k} + x_5 W_N^{5k} + \dots + x_{(N-3)} W_N^{k(N-3)} + x_{(N-1)} W_N^{k(N-1)}] \end{aligned}$$

위의 DFT 에서 짝수 번째 항과 홀수 번째 항의 묶은 후에 재정의 한다면 아래와 같아진다.

$$f_k = \sum_{n=0}^{N/2-1} x_{2n} W_N^{2kn} + W_N^k \sum_{n=0}^{N/2-1} x_{2n+1} W_N^{2kn}$$

여기서  $W_N^2 = e^{-4\pi i/N} = e^{-2\pi i/(N/2)} = W_{N/2}$  가 성립하므로

$$\begin{aligned} f_k &= \sum_{n=0}^{N/2-1} x_{2n} W_{N/2}^{kn} + W_N^k \sum_{n=0}^{N/2-1} x_{2n+1} W_{N/2}^{kn} \\ &= E_k + W_N^k O_k \quad k = 0, \dots, N/2-1 \end{aligned}$$

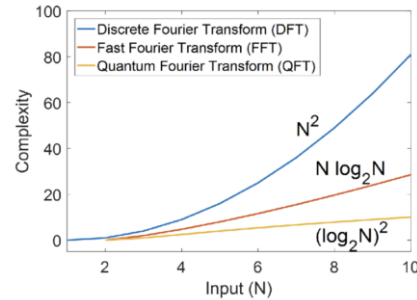
위의 식으로 부터  $f_{k+N/2}$  를 구해본다면

$$\begin{aligned} f_{k+N/2} &= \sum_{n=0}^{N/2-1} x_{2n} W_{N/2}^{(k+N/2)n} + W_N^{k+N/2} \sum_{n=0}^{N/2-1} x_{2n+1} W_{N/2}^{(k+N/2)n} \\ &= \sum_{n=0}^{N/2-1} x_{2n} W_{N/2}^{kn} - W_N^k \sum_{n=0}^{N/2-1} x_{2n+1} W_{N/2}^{kn} \\ &= E_k - W_N^k O_k \quad k = 0, \dots, N/2-1 \text{ 와 같아진다} \end{aligned}$$

(E 는 Even, 0 는 Odd) 위의 식을 구현하면  $O(N \log N)$ 의 시간 복잡도인 것을 수식을 통해 알 수 있다. 이것이 FFT 의 기본적인 이론이다.

위 식에서  $E_k$  와  $O_k$  는 각각 N/2 개의 원소에 대한 DFT 식이다. 따라서 복소지수함수와 DFT 의 주기성을 이용한다면, 좀 더 단순화 시켜  $N/2 \log N$  의 횟수로 계산을 할 수 있다.

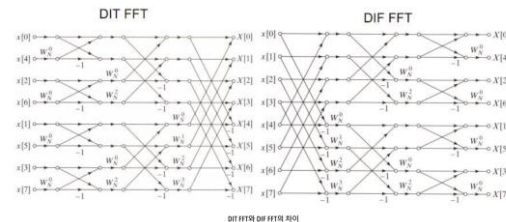
(그림 1)



DFT(Discrete Fourier Transform)과 FFT(Fast Fourier Transform)의 계산 효율 비교

## FFT 의 다양한 알고리즘

FFT 알고리즘은 DIF(Decimation in Frequency)와 DIT(Dicimation in Time)알고리즘으로 분류된다. DIF 는 말그대로 주파수분할이며 출력을 나누는 방식이며, DIT 는 시분할로 입력을 나누는 방식이다. 또 DIF 와 DIT 각각 Radix 2/4/8 알고리즘으로 분류할 수 있다.



**Coolley-Tukey Algorithm(Radix-2)**은 가장 일반적으로 사용되는 FFT 알고리즘으로, 분할 정복 기반이다. Radix-2 알고리즘은 다른 FFT 알고리즘의 기초가 되는 알고리즘이므로 본 글에서는 Radix-2 알고리즘에 대해 알아보고 구현 해보겠다. 전에서 도출한 다음의 식은 N 개의 데이터에 대한 DFT 를 N/2 개의 데이터에 대한 두 DFT 로 분할 한 것이다.

$$f_k = \sum_{n=0}^{N/2-1} x_{2n} W_N^{2kn} + W_N^k \sum_{n=0}^{N/2-1} x_{2n+1} W_N^{2kn}$$

위의 식과 같이 FFT에서는 재귀적으로 절반의 크기를 구하기 때문에 기본적으로 수

열의 길이를  $2^N$ 로 맞춰놓고 시작한다.

$$f_k = \sum_{n=0}^{N-1} x_n W_N^{kn} \quad k = 0, \dots, N-1$$

추가로 설명하자면, 위의 식에서 k=0에서 N-1까지 각 항의 계산에 N회의 곱셈이 필요하므로  $N^2$ 회의 곱셈이 필요하게 된다.

$$\begin{aligned} f_k &= E_k + W_N^k O_k & k = 0, \dots, N/2-1 \\ f_{k+N/2} &= E_k - W_N^k O_k & k = 0, \dots, N/2-1 \end{aligned}$$

하지만 위 식에서 DFT는  $(N/2)^2 = N^2/4$ 회의 곱셈이 필요하다. 이러한 식의 분할에 의해 계산량이 절반으로 줄어든 것이다. 이 분할을 2, 3. 회 반복하다 보면 계산량이 1/4, 1/8.로 감소하는데 이것이 Cooley-Tukey 알고리즘의 기본이다.

앞서 말한 것처럼 N을 재귀적으로 2등분하여 N=2인 신호를 얻어 계산하므로

$N = 2^n$ 인 경우가 대부분이다. (N=2인 신호는 DFT 계산이 쉽다.)

또, 알고리즘 구현 중 알아야 할 것은

$N = 2^n$ 인 경우에 Bit Reversal이 성립된다는 것이다. 처음 원래의 신호와 최종적으로 나누어진 신호의 순서를 이진수로 나타내면 아래 그림과 같이 대응하는 신호의 모든 비트들을 거꾸로 배열한 것과 같다.

	stage 1	stage 2	stage 3	
initial	N=8	N=4	N=2	final
000	0	0	0	000
001	1	2	4	100
010	2	4	2	010
011	3	6	6	110
100	4	1	1	001
101	5	3	5	101
110	6	5	3	011
111	7	7	7	111

bit reversed order (  $b_3 b_2 b_1 \rightarrow b_1 b_2 b_3$  )

시분할 Radix-2(Radix-2 DIT) 알고리즘에 대하여 알아보자. N=4인 경우 DFT 행렬 식은

(그림2)

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & W^2 & W^3 \\ W^0 & W^2 & W^4 & W^6 \\ W^0 & W^3 & W^6 & W^9 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

위와 같다. 지수의 짝홀을 기준으로 열의 순서를 변경한다면 아래와 같이 변형(앞에서 말한 분할)이 가능하다.

$$W^4 = W^0, \quad W^6 = W^{2+4} = W^{2+0} = W^2$$

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^2 & W^1 & W^3 \\ W^0 & W^4 & W^2 & W^6 \\ W^0 & W^6 & W^3 & W^9 \end{bmatrix} \begin{bmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^2 & W^1 & W^3 \\ W^0 & W^2 & W^1 & W^3 \\ W^0 & W^2 & W^1 & W^3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{bmatrix}$$

위의 식을 회전자의 성질을 이용하여 변형하면

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & W^0 & 0 \\ 0 & 1 & 0 & W^1 \\ 1 & 0 & W^2 & 0 \\ 0 & 1 & 0 & W^3 \end{bmatrix} \begin{bmatrix} W^0 & W^0 & 0 & 0 \\ W^0 & W^2 & 0 & 0 \\ 0 & 0 & W^0 & W^0 \\ 0 & 0 & W^0 & W^2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & W^0 & 0 \\ 0 & 1 & 0 & W^1 \\ 1 & 0 & W^2 & 0 \\ 0 & 1 & 0 & W^3 \end{bmatrix} \begin{bmatrix} \begin{bmatrix} W^0 & W^0 \\ W^0 & W^2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_2 \end{bmatrix} \\ \begin{bmatrix} W^0 & W^0 \\ W^0 & W^2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_3 \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & W_4^0 & 0 \\ 0 & 1 & 0 & W_4^1 \\ 1 & 0 & W_4^2 & 0 \\ 0 & 1 & 0 & W_4^3 \end{bmatrix} \begin{bmatrix} \begin{bmatrix} W_2^0 & W_2^0 \\ W_2^0 & W_2^1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_2 \end{bmatrix} \\ \begin{bmatrix} W_2^0 & W_2^0 \\ W_2^0 & W_2^1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_3 \end{bmatrix} \end{bmatrix}$$

이때,  $W_4^2 \rightarrow -W_4^0, W_4^3 \rightarrow -W_4^1, W_2^1 \rightarrow -W_2^0$ 이므로 최종식은 아래와 같아진다.

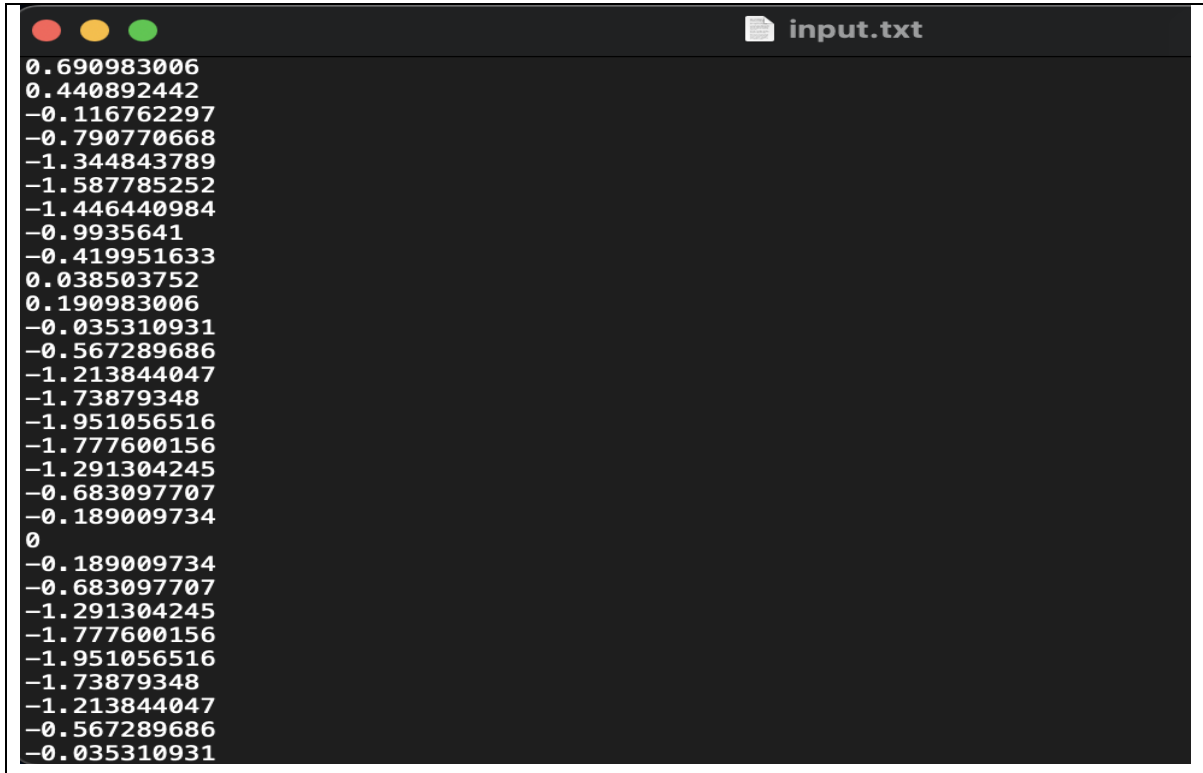
$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & W_4^0 & 0 \\ 0 & 1 & 0 & W_4^1 \\ 1 & 0 & -W_4^0 & 0 \\ 0 & 1 & 0 & -W_4^1 \end{bmatrix} \begin{bmatrix} \begin{bmatrix} W_2^0 & W_2^0 \\ W_2^0 & -W_2^0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_2 \end{bmatrix} \\ \begin{bmatrix} W_2^0 & W_2^0 \\ W_2^0 & -W_2^0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_3 \end{bmatrix} \end{bmatrix}$$

즉, N=4인 DFT는 N=2인 DFT 두 개를 이용하여 계산할 수 있다.

## II FFT Algorithm 구현

### 1. 알고리즘 구현

- **InputData** “./data/input.txt”에 데이터를 저장했다. 결과로 “output\_fft.txt”파일이 같은 경로에 저장된다.



```
0.690983006
0.440892442
-0.116762297
-0.790770668
-1.344843789
-1.587785252
-1.446440984
-0.9935641
-0.419951633
0.038503752
0.190983006
-0.035310931
-0.567289686
-1.213844047
-1.73879348
-1.951056516
-1.777600156
-1.291304245
-0.683097707
-0.189009734
0
-0.189009734
-0.683097707
-1.291304245
-1.777600156
-1.951056516
-1.73879348
-1.213844047
-0.567289686
-0.035310931
```

FFT 알고리즘의 특성을 유의하며 Java 로 구현해 보겠다. 사용한 클래스는 아래와 같다.

- **Complex** 복소수형 데이터를 처리하기 위한 클래스, 실수부 *re* 와 허수부 *im* 을 가지는 형태

```
package fourier;

public class Complex {

    public double re = 0;
    public double im = 0;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // a+bi 형태의 복소수를 문자열로 반환한다.
    public String toString() {
        return toString("i");
    }
}
```

```
// toString("j") --> a+bj 형태의 복소수를 문자열로 반환한다.
public String toString(String im_ch) {
    String s = "";
    if(this.re==0 && this.im==0) {
        s = "0";
    }else if(this.re == 0) {
        s = this.im + im_ch;
    }else if(this.im == 0) {
        s = "" + this.re;
    }else if(this.im > 0) {
        s = this.re + "+" + this.im + im_ch;
    }else {
        s = "" + this.re + this.im + im_ch;
    }
    return s;
}
}
```

#### ■ CMath 복소수끼리의 연산이 구현된 클래스

```
package fourier;
public class CMath {
    public static Complex add(Complex z1, Complex z2) {
        return new Complex(z1.re + z2.re, z1.im + z2.im);
    }

    public static Complex subtract(Complex z1, Complex z2) {
        return new Complex(z1.re - z2.re, z1.im - z2.im);
    }

    public static Complex multiply(Complex z1, Complex z2) {
        return new Complex(z1.re * z2.re - z1.im * z2.im, z1.re * z2.im + z1.im * z2.re);
    }

    public static Complex divideR(Complex z, double r) {
        return new Complex(z.re / r, z.im / r);
    }
}
```

```

    public static double magnitude(Complex z) {
        return Math.sqrt(z.re * z.re + z.im * z.im);
    }
}

```

#### ■ FFT fft와 ifft를 가지고 있는 핵심 클래스

```

package fourier;

public class FFT {

    private static double log2(double a) {
        return Math.log(a) / Math.log(2);
    }

    /**
     * Check the number is power of 2 or not.
     * @param N
     * @return
     * 0 : it is power of 2
     * padding_count: if it is now power of 2, required count number to be power of 2
     * @throws Exception
     */
    private static int checkPowerOfTwo(int N) throws Exception {
        int result = -1;
        if (N == 0) throw new Exception("Input size is zero.");
        double l,c,f;
        l=FFT.log2(N);
        c=Math.ceil(l);
        f=Math.floor(l);
        if(c==f) result =0;
        else result = (int)Math.pow(2, c)-N;
        return result;
    }

    private static Complex[] makePowerOfTwoSize(Complex[] arr, int fillCount) {
        Complex[] newArr = new Complex[arr.length + fillCount]; //initial value is null
        System.arraycopy(arr, 0, newArr, 0, arr.length);
        for(int i=arr.length;i<newArr.length;++i) {

```



```

        newArr[i] = new Complex(0,0);
    }
    return newArr;
}

// getReversedArr(13,4) -> {1,0,1,1} where, 13=1101
private static int getReversedNum(int num, int numOfBits) {
    int i, j, k;
    //1. Decimal number -> Reversed Binary array
    byte a;
    int d=0;
    //num=13, numOfBits=4
    for (i=0;i<numOfBits;++i) {        // 0      1      2      3
        j = numOfBits - 1 - i;        // 3      2      1      0
        k = (int)(num >> j);          // 13>>3=1  13>>2=3  13>>1=6  13>>0=13
        a = (byte)(k & 1);             // arr[3]=1  arr[2]=1  arr[1]=0  arr[0]=1
        d = d + (a<<i);
    }
    return d;
}

private static Complex[] arrange4fft(Complex[] src, int numOfBits) throws Exception{
    int i,j;
    Complex[] arrangedSrc = new Complex[src.length];
    for(i=0;i<src.length;++i) {
        j = FFT.getReversedNum(i, numOfBits); //{000,001,010, 011, 100, 101, 110, 111} -> {0, 4,
2, 6, 1, 5, 3, 7}
        arrangedSrc[j] = src[i];
    }
    return arrangedSrc;
}

/**
 * Calculate W(Convolution ring)
 *
 *  $W[k]$  for FFT =  $\cos(\theta) - j\sin(\theta)$ , where  $\theta = (2\pi*k/N)$ 

```

```

*  $W[k]$  for IFFT =  $\cos(-\theta)$  -  $i\sin(-\theta)$ , where  $\theta = (2\pi k/N)$ 
* @param N
* @param isInverse true: for IFFT, false: for FFT
* @return calculated W array
*/

private static Complex[] calculateW(int N, boolean isInverse) {
    Complex[] W = null;
    double T = 0;
    int N2 = N/2;
    W = new Complex[N2];
    T = 2 * Math.PI / N;
    double theta = 0;
    if (isInverse) { //IFFT
        for(int i=0;i<N2;++i) {
            theta = -(T * i);
            W[i] = new Complex(Math.cos(theta), -Math.sin(theta));
        }
    }else { //FFT
        for(int i=0;i<N2;++i) {
            theta = (T * i);
            W[i] = new Complex(Math.cos(theta), -Math.sin(theta));
        }
    }

    return W;
}

/**
 *
 * @param X 2D array. X[0][i] and X[1][i] change it's role each other by the src and tgt number
 *
 * @param src source index of the array X. 0 or 1
 * @param tgt target index of the array X. 0 or 1
 * @param s Starting index of the data in the array
 * @param size Region size to be calculated: 2 -> 4 -> 8 -> ...
 * @param klump k's jumping value to use right value of W

```

```

* @param W Convolution ring
*/
private static void regionFFT(
    Complex[][] X, int src, int tgt, int s, int size, int kJump, Complex[] W) {

    // Xm+1[i] = Xm[i] + Xm[i+half]W[k]
    // Xm+1[i+half] = Xm[i] - Xm[i+half]W[k]
    int k=0;
    int e = s + (size/2)-1;
    Complex T=null;
    int half = (int)(size / 2);
    for(int i=s;i<=e;++i) {
        T = CMath.multiply(X[src][i+half], W[k]);
        X[tgt][i] = CMath.add(X[src][i], T);
        X[tgt][i+half] = CMath.subtract(X[src][i], T);
        k += kJump;
    }
}

////////////////////////////////////
private static Complex[] fft_forward(Complex[] src, boolean isInverse) throws Exception {
    int N = src.length;
    //1. check src data size. If it's not 2^n size then make it to be 2^n size with zero padding
    int fillCount = 0;
    fillCount = checkPowerOfTwo(N);
    if(fillCount > 0) {
        src = makePowerOfTwoSize(src, fillCount);
    }else if(fillCount < 0) {
        throw new Exception("Something wrong: while calculateing filling count from the input
data");
    }

    //2. calculate loop count: 2-> 1, 4->2, 8->3, 16->4
    N = src.length; //input array can be changed. so, read one more time.
    int totalLoop = (int)FFT.log2(N); //it is guaranteed that N is 2^n number, therefore log2(N)
will be int value.
    //3. arrange src index to calculate FFT as bottom-up style

```

```

Complex[] arrangedSrc = arrange4fft(src, totalLoop);
//4. calculate W: 0 to (N/2-1)
Complex[] W = calculateW(N, isInverse); //for fft. for ifft:calculateW(N, true)
//5. use 2-dimensional array to save memory space. X[0, ] <-> X[1, ]
Complex[][] X = new Complex[2][N];
System.arraycopy(arrangedSrc, 0, X[0], 0, N);
//6. calculate FFT by bottom-up style
int srcIdx=0, tgtIdx=0;
int kJump=0, regionSize =0;
for(int curLoop=0; curLoop<totalLoop;++curLoop) {
    tgtIdx = (tgtIdx+1) % 2;
    srcIdx = (tgtIdx+1) % 2;
    regionSize = 1 << (curLoop+1);    //if N=8: 2 -> 4 -> 8
    kJump = 1 << (totalLoop - curLoop - 1);    //if N=8: 4 -> 2 -> 1
    int i=0;
    while(i<N) {
        regionFFT(X, srcIdx, tgtIdx, i, regionSize, kJump, W);
        i += regionSize;
    }
}

//7. find final output index in the X array
int resultIdx = ((totalLoop & 1) == 0) ? 0 : 1;
//8. in case of IFFT, divide N
if(isInverse) {
    for(int i=0;i<N;++i) X[resultIdx][i] = CMath.divideR(X[resultIdx][i], N);
}

//9. return the result as 1d array
return X[resultIdx];
}

public static Complex[] fft(double[] src) throws Exception{
    return FFT.fft(src, -1);
}

```

```

public static Complex[] fft(double[] src, int roundDigit) throws Exception{
    Complex[] complexSrc = new Complex[src.length];
    for(int i=0;i<src.length;++i) complexSrc[i] = new Complex(src[i],0);
    Complex[] X = fft_forward(complexSrc, false);
    // round up at roundDigit
    if(roundDigit >= 0) {
        double d = Math.pow(10.0, roundDigit);
        for(int i=0;i<X.length;++i) {
            X[i] = new Complex(Math.round(X[i].re * d)/d, Math.round(X[i].im * d)/d);
        }
    }
    return X;
}

public static double[] ifft(Complex[] src) throws Exception{
    return FFT.ifft(src,-1);
}

public static double[] ifft(Complex[] src, int roundDigit) throws Exception{
    Complex[] complexArr = fft_forward(src, true);
    double[] x = new double[complexArr.length];
    if(roundDigit >= 0) {
        double d = Math.pow(10.0, roundDigit); // round up at roundDigit
        for(int i=0;i<x.length;++i) {
            x[i] = Math.round(complexArr[i].re * d) / d ;
        }
    }else {
        for(int i=0;i<x.length;++i) {
            x[i] = complexArr[i].re;
        }
    }

    return x;
}
}

```

■ **Main** main 함수가 있는 클래스, 클래스 및 함수 호출

```

import java.io.*;
import java.util.Vector;
import fourier.*;
public class Main {

    public static void main(String[] args) {
        try {
            test_simple_fft();
            test_not_power_of_2();
            test_measure_time();
            test_use_file();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // 1. test for simple a few data
    private static void test_simple_fft() throws Exception{
        double[] x = {0,0,0,0,1,1,1,1}; //input
        Complex[] X; //result value after FFT calculation
        System.out.println("*** test_simple_fft ***");
        //1. calculate FFT for a few data. (count = 8)
        X = FFT.fft(x);
        for(int i=0;i<X.length;++i) System.out.print(X[i].toString()+" ", );
        System.out.println("");
        //2. obtain FFT value with rounded format. and use "j" for imaginary character
        X = FFT.fft(x,2); //rounded num_digit is 2
        for(int i=0;i<X.length;++i) System.out.print(X[i].toString("j")+ " ", );
        System.out.println("");
    }

    // 2. in case of 2^n count of data
    private static void test_not_power_of_2() throws Exception{
        double[] x = {0,0,0,0,1,1}; //input size is 6 that is not 2^n count
        Complex[] X; //result value after FFT calculation
        System.out.println("\n*** test_not_power_of_2 (auto zero padding) ***");
    }
}

```

```

//1. in case of the input data size is not 2^n count.
// --> automatically fill zero and make input size 2^n count
X = FFT.fft(x,3);
for(int i=0;i<X.length;++i) System.out.print(X[i].toString()+" ");
    System.out.println("");
}

// 3. measure execution time to 32k, 64k size data
private static void test_measure_time() throws Exception{
    long beforeT, diffT;
    int N = 0;
    System.out.println("\n** test_measure_time **");
    //1) 32kB data
    N = (int)Math.pow(2, 15);
    double[] data32 = new double[N];
    for(int i=0;i<N;++i) data32[i] = Math.cos(2*Math.PI*0.004*i);
    beforeT = System.currentTimeMillis();
    Complex[] result32 = FFT.fft(data32);
    diffT = System.currentTimeMillis() - beforeT;
    System.out.println("Execution Time(32kB): " + diffT + "ms");
    for(int i=0;i<4;++i) System.out.print("[ "+i+" "+result32[i].toString()+" ");
    System.out.println(".....");
    for(int i=N-4;i<N;++i) System.out.print("[ "+i+" "+result32[i].toString()+" ");
    System.out.println("");
    //2) 64kB data
    N = (int)Math.pow(2, 16);
    double[] data64 = new double[N];
    for(int i=0;i<N;++i) data64[i] = Math.cos(2*Math.PI*0.004*i);
    beforeT = System.currentTimeMillis();
    Complex[] result64 = FFT.fft(data64);
    diffT = System.currentTimeMillis() - beforeT;
    System.out.println("Execution Time(64kB): " + diffT + "ms");
    for(int i=0;i<4;++i) System.out.print("[ "+i+" "+result64[i].toString()+" ");
    System.out.println(".....");
    for(int i=N-4;i<N;++i) System.out.print("[ "+i+" "+result64[i].toString()+" ");
}

```

```

private static void test_use_file() throws Exception {
    System.out.println("\n** test_use_file **");
    //1) read file
    Vector<String> v = new Vector<String>();
    File f = new File("./data/input.txt");
    BufferedReader br = new BufferedReader(new FileReader(f));
    String line="";
    while((line=br.readLine()) != null) {
        line = line.trim();
        if(line.length()!=0) v.add(line);
    }
    br.close();
    //2) parse string to double
    double[] x = new double[v.size()];
    for(int i=0;i<x.length;++i)    x[i] = Double.parseDouble(v.get(i));
    //3) do fft
    long beforeT = 0, diffT=0;
    beforeT = System.currentTimeMillis();
    Complex[] X = FFT.fft(x);
    diffT = System.currentTimeMillis() - beforeT;
    System.out.println("fft calculation time: "+diffT+"ms");
    //4. save Y to the file
    f = new File("./data/outptu_fft.txt");
    BufferedWriter bw = new BufferedWriter(new FileWriter(f));
    if(f.isFile() && f.canWrite()) {
        for(int i=0;i<X.length;++i) {
            bw.write(X[i].toString());
            bw.newLine();
        }
    }
    System.out.println("fft result is written to the file");
    bw.close();
}
}

```



- **OutputData** 프로그램 실행 시 “./data/output\_fft.txt”에 데이터가 저장된다.

```
** test_measure_time **
Execution Time(32kB): 32ms
[0]17.441665612932592,
[1]17.442678036016932+0.03054032554147773i,
[2]17.445716012601977+0.06109131908951394i,
[3]17.450781665925863+0.09166366108017976i,
.....
[32764]17.457878538842998-0.1222680568280476i,
[32765]17.450781665924485-0.09166366107969492i,
[32766]17.445716012601984-0.06109131908950698i,
[32767]17.442678036016957-0.03054032554145092i,

Execution Time(64kB): 24ms
[0]31.474846045104538,
[1]31.475301314548666+0.058005922412330935i,
[2]31.4766672023788+0.11601690943639723i,
[3]31.478943947152842+0.17403802715017125i,
.....
[65532]31.48213194657478-0.2320743446023137i,
[65533]31.47894394715285-0.1740380271501539i,
[65534]31.476667202378863-0.11601690943633183i,
[65535]31.47530131454865-0.05800592241228639i,

** test_use_file **
fft calculation time: 12ms
fft result is written to the file
```

## 2. 알고리즘 성능 평가

64kBytes 와 32kBytes 에서 각각 32ms, 24ms 가 나왔다. 알고리즘 측면에서 FFT 를 잘 구현했지만 불필요한 클래스 또는 메서드들을 간소화 한다면 시간을 더 단축 시킬 수 있을 것 같다. 예를 들어, 복소수 형태의 계산 결과를 Complex 용 클래스를 사용하는 것이 아니라 실수부, 허수부를 나누어 각각의 배열로 만들면 공간적인 측면에서 시간이 절약 될 것 같다(데이터가 충분히 많다면) 또, 회전자  $W^k$  를 매 라운드마다 달라지는 N 값에 대해  $W_N^k$  를 계산 하는 것이 아니라  $W^k$  를 한번 구한 후 사용하고  $W^0 \sim W^{\frac{N}{2}-1}$  까지만 구하여 재활용하면 계산이 조금 더 줄어들 것이다. 데이터를 정렬하는 과정(Bit reversal)이나 데이터 입출력 과정은 구현한 코드로부터 개선의 여지가 눈에 띄일만큼 보이지는 않는다.

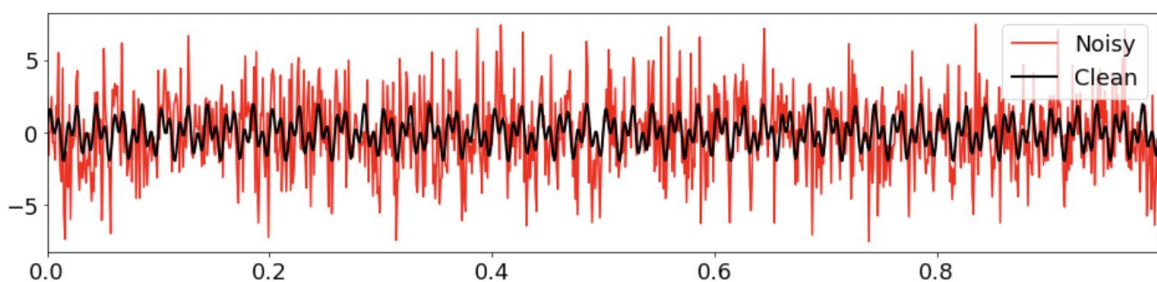
### III 알고리즘의 적용

#### 1. 알고리즘의 적용

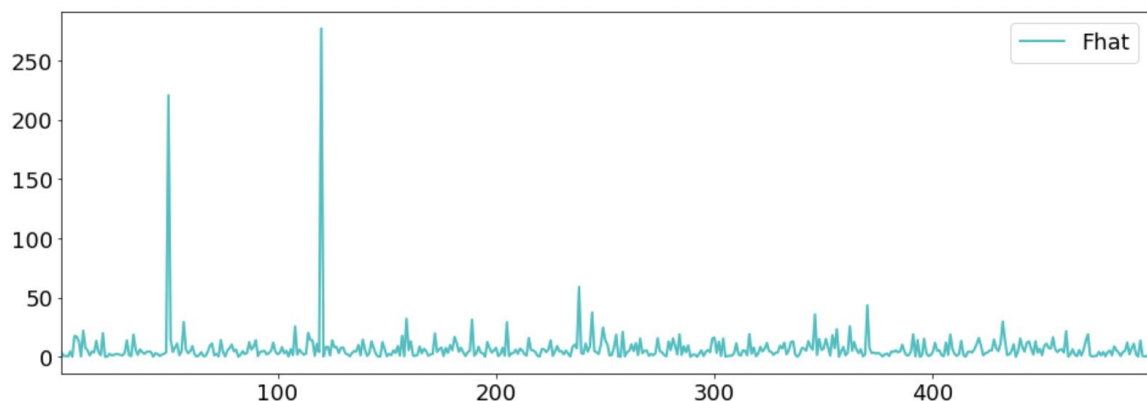
FFT 알고리즘은 오디오 및 음향 측정뿐 아니라 다양한 분야에서 활용된다. 신호를 개별 스펙트럼 구성 요소로 변환하여 신호에 대한 주파수 정보를 제공한다. FFT는 기계 또는 시스템의 결함 분석, 품질 관리 및 상태 모니터링에도 사용된다. 본인이 평소 관심이 많던 통신분야의 noise 제거에도 사용되는데 이를 알아보자.

신호를 측정할 시에는 측정 대상이 되는 신호 외에 여러 noise 들이 끼어 들어온다. 수신 측에서는 이러한 noise 들을 제거해야하는데, 송신 데이터에 FFT를 적용하여 시간 domain에서 주파수(진동수) domain으로 바꾸면 각 데이터 점들이 어떠한 진동수를 가지고 있는지 알 수 있게 된다. noise 들은 각 점에 합쳐지는데 원래의 신호가 가진 진동수와 같이 모든 데이터에 포함 합쳐지지는 않는다. 즉, noise가 섞인 신호를 받아도 이를 진동수 영역으로 넘겨서 분석해보면 신호가 갖는 진동수가 다른 진동수보다 크다. 이를 이용해 일정 크기보다 작은 진동수를 모두 제거하고 푸리에 역변환 한다면 원래 신호의 진동수로 구성된 파형을 얻을 수 있다.

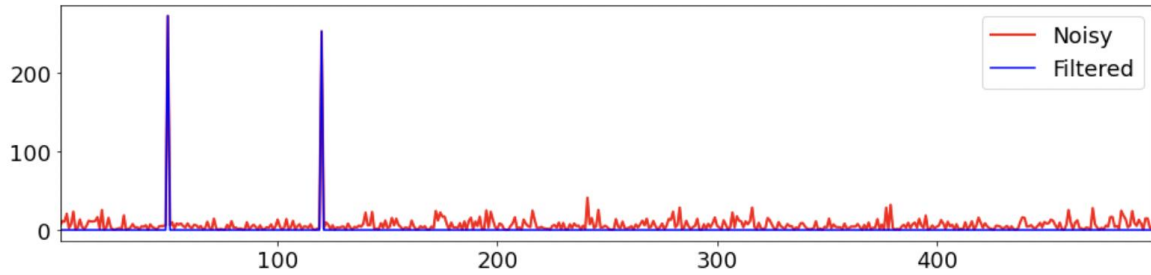
원래의 신호가 진동수 50과 120으로 구성되어 있으며, 이 신호에 임의의 noise를 구했을 때, 수신단에서 받는 신호는 아래와 같다.



검정색으로 나타난 것이 송신 신호, 빨간색이 noise이다. 위의 신호를 FFT



위의 사진은 송신 신호를 진동수 domain에서 나타낸 것이다. 원래 신호의 진동수가 50, 120이므로 위의 그림처럼 나타난다. PSD를 이용하여 filtering된 푸리에 계수를 역변환하면 기존 신호를 거의 동일하게 복원 가능하다.



빨간색과 파란색은 각각 filtering 하기 전의 푸리에 계수, filtering 이후의 계수이다. filtering 을 통해 원래 파형을 구성하고 있는 진동수 푸리에 계수만 남기고 noise 는 모두 제거한 것이다. 남은 푸리에 계수를 역변환하여 측정하면 원래의 파형을 구할 수있다.

## IV 결론

고속 푸리에 이론에 대하여 알아보았다. 통신공학이론에서 DFT 에 대하여 미리 공부하였기 때문에 흥미를 가지고 알고리즘을 대할 수 있었다. 비록 FFT 유도 과정에서 수학적 이론과 여러 이론들이 필요하지만, 통신공학에서 뺄 수 없는 알고리즘이라는 것을 알았다. 수업 시간 외에서 noise 제거에 관한 FFT 를 따로 공부해야겠다.

## 참고문헌(기타 제목)

### 단행본

- 위키백과 이산 푸리에 변환 中
- 위키백과 고속 푸리에 변환 中
- heejin\_park. 가장 빠른 java 용 FFT 구현해보기, <https://infograph.tistory.com/349>
- heejin\_park, java 로 FFT 알고리즘 충실히 구현해보기, <https://infograph.tistory.com/352>
- Math/Fourier Analysis, FFT를 이용한 noise 제거하기 <https://people-analysis.tistory.com/152>

### 논문

- Young-Beom Jang&Sang-Woo Lee, DIT Radix-4 FFT 구현을 위한 저전력 Butterfly 구조, 한국통신학회논문지(J - KICS) '13-12 Vol. 38A NO. 12 <https://doi.org/10.7840/kics>, 2013. 38A. 12. 1145

## 그림

- (그림 1) 음향/소음/진동 전문기업 “아이티에스” 공식 블로그 - 고속 푸리에 변환과 주파수 분석 시 스펙트럼의 종류 中
- (그림 2) 위키백과 - 고속 푸리에 변환 中 발체