REPORT

인천대학교

과목명: 컴퓨터알고리즘

담당교수: 김동훈 교수님

학과: 정보통신공학과

학년: 4학년

학번: 201501596

이름: 이규명

FFT(Fast Fourier Transform) Algorithm

서론

푸리에 변환(Fourier Transform)은 매우 많은 분야에서 다양하게 활용하고 있는 중요한 개념이다. 신호를 주파수 성분으로 변환하여 다양한 분석 처리를 할 수 있고, 임의의 필터링 연산도 가능하게 한다. 이러한 푸리에 변환 중에서도 디지털 도메인 입력 신호를 받아 디지털 도메인 형식으로 결과값을 반환하는 이산 푸리에 변환(`Discrete Fourier Transform`, `DFT`) 및 그의 역변환을 빠르게 계산할 수 있는 알고리즘인 고속 푸리에 변환(`Fast Fourier Transform`, `FFT`)에 대하여 알아보도록 하겠다.

DFT

FFT에 대해 이해하기 위해서는 먼저 DFT를 알아야 한다.

DFT란 디지털 신호를 받아 디지털 주파수로 바꾸어 주는 푸리에 변환을 말한다. 즉 신호 값이 연속적이지 않고 띄엄 띄엄 존재하며 이 값들을 제외한 시간이나 주파수 부분은 값이 0인 것이다.

DFT의 공식은 다음과 같다. j는 0부터 n-1 까지의 표본 순번이 되다. 즉 n은 표본의 개수가 되며 이 표본의 값이 주파수 함수 값을 구하고자 하는 시간 함수의 입력값이 된다. 2pi/n은 시간함수에서의 각 표본 간의 간격이며 k는 변환 후 주파수함수의 순번이다.

$$F_k = \sum_{j=0}^{n-1} f_j e^{-ikj2\pi/n}$$

여기서 입력값인 시간함수의 표본은 실수 값인 반면 변환된 주파수 함수의 값은 복소수 값을 얻게 된다는 점을 유의해야 한다.

FFT

1) 동작 방식

그렇다면 FFT의 동작에 대하여 알아보자.

기본적으로 FFT는 분할 정복 알고리즘을 사용하여 재귀적으로 n 크기를 n = n1*n2 식이 성립하는 n1, n2크기의 두 DFT로 나눈 후 각각의 결과를 다시 합치는 방식으로 구현하는 형태가 많다.

여기서 FFT의 많은 알고리즘들이 존재하는데 이 중 Cooley–Tukey 알고리즘의 가장 기본적인 형태인 *기수-2 DIT 형태(radix-2 DIT case)*의 알고리즘으로 알아보겠다.

Cooley–Tukey 알고리즘은 보통 n을 2분할하여 분할 정복을 수행하기 때문에 n이 2의 제곱수인 경우에 많이 쓰인다. 2분할은 보통 짝수 순번과 홀수 순번의 형태로 분할하는데 이를 이해하기 쉽게 풀어 설명하면 `n=8`이고 `[f1,f2,f3,f4,f5,f6,f7,f8]` 와 같이 크기 8인 배열 안에 시간 함수의 값이 주어져 있다고 가정해보자. 이 배열은 짝수 순번과 홀수 순번의 배열로 2분할이 되는데 짝수 순번: `[f2, f4, f6, f8]` 홀수 순번: `[f1, f3, f5, f7]`으로 분할이 되는 것이다. 또 각 배열들은 또다시 분할이 되는데 `[f2, f4, f6, f8]` -> 짝수 순번: `[f4, f8]` 홀수 순번: `[f2, f6]` 으로 분할되고 `[f1, f3, f5, f7]` -> 짝수 순번: `[f3, f7]` 홀수 순번: `[f1, f5]` 와 같이 분할되어 각각 DFT를 한 결과를 합치는 방식인 것이다.

2) 계산 속도

이렇게 동작하는 FFT는 DFT와 계산 속도 면에서 얼만큼의 차이가 있을까?

이를 알기 위해서는 convolution을 곱으로 바꿔주는 규칙을 적용하면 알 수 있다.

$$\mathcal{F}\{a * b\} = \mathcal{F}\{a\}\mathcal{F}\{b\}$$

이는 푸리에변환한 함수의 곱은 convolution한 값과 같다는 의미이다.

먼저 일반적으로 DFT한 수열식 a, b의 convolution을 구하고자 할 경우에 대해 보자면, 주기가 N인 수열 a, b의 convolution c는 다음과 같이 나타낼 수 있다.

$$c_n = \sum_{j=0}^{N-1} a_j b_{n-j}$$

$$(c_i = a_0 b_i + a_1 b_{i-1} + \ldots + a_i b_0)$$

위 방법으로 convolution c를 직접 계산하면 최종적으로 `O(N^2)`의 시간이 걸린다.

하지만 DFT를 빠르게 구할 수 있다면 더 빠르게 결과를 계산해낼 수 있을 것이다. 바로 FFT가 DFT를 빠르게 구하게 해주는 알고리즘이다.

FFT의 알고리즘대로 DFT를 구해본다면 짝수 순번, 홀수 순번으로 분할한 후 분할한 짝수번째 항들의 DFT와 홀수번째 항의 DFT를 계산하는 데 `O(log N)`의 시간이 걸리고, 이를 합치면서 전체 DFT를 계산하는 데 `O(N)`의 시간이 걸리게 되어 총 `O(Nlog N)`의 시간으로 DFT를 계산할 수 있게 된다.

3) 계산식

FFT를 수학적 계산식으로 나타내어 알아보자. 먼저 위에서 다루었던 DFT의 공식을 이용해 짝수 순번, 홀수 순번으로 분할하는 식을 보자면 다음과 같다. 여기서 짝수 순번은 j = 2m으로, 홀수 순번은 j = 2m+1로 치환하였고, 2분할이 되었으니 각 수열은 n/2 - 1까지의 범위가 된다.

$$F_k = \sum_{j=0}^{n-1} f_j e^{-ikj2\pi/n}$$

$$F_k = \sum_{m=0}^{n/2-1} f_{2m} e^{-ik2\pi(2m)/n} + \sum_{m=0}^{n/2-1} f_{2m+1} e^{-ik2\pi(2m+1)/n}$$

위 식을 다시 정리하면 다음과 같아진다. 여기서 f2m부분이 짝수 순번 파트, f2m+1부분이 홀수 순번 파트로 볼 수 있다. 또한 홀수 파트 앞에 곱해진 지수함수 e는 오일러 공식을 이용해 sin, cos관련 식으로 바꾸어 계산할 수 있다. 여기서 cos값은 실수부, sin값은 허수부임을 인지해야 한다.

$$F_k = \sum_{m=0}^{n/2-1} f_{2m} e^{-\frac{2\pi i}{n/2}mk} + e^{-\frac{2\pi i}{n}k} \sum_{m=0}^{n/2-1} f_{2m+1} e^{-\frac{2\pi i}{n/2}mk}$$

$$= (\text{짝수 파트})E_k + e^{-\frac{2\pi i}{n}k}(\text{홀수 파트})O_k$$

$$e^{-\frac{2\pi i}{n}k} = cos(-\frac{2\pi k}{n}) + isin(-\frac{2\pi k}{n})$$

그런데 `k`대신 `k+n/2`의 값을 넣어서 식을 세워보면 k식과 가운데 `+, -`부호만 다른 점을 알 수 있다. 이 점을 이용해 이전에 이용했던 식을 재활용할 수 있으므로 알고리즘을 단순화할 수 있다는 것을 알 수 있다.

$$F_{k+\frac{n}{2}} = \sum_{m=0}^{n/2-1} f_{2m} e^{-\frac{2\pi i}{n/2} m(k+\frac{n}{2})} + e^{-\frac{2\pi i}{n}(k+\frac{n}{2})} \sum_{m=0}^{n/2-1} f_{2m+1} e^{-\frac{2\pi i}{n/2} m(k+\frac{n}{2})}$$

$$= \sum_{m=0}^{n/2-1} f_{2m} e^{-\frac{2\pi i}{n/2} mk} e^{-2\pi im} + e^{-\frac{2\pi i}{n} k} e^{-\pi i} \sum_{m=0}^{n/2-1} f_{2m+1} e^{-\frac{2\pi i}{n/2} mk} e^{-2\pi im}$$

$$= \sum_{m=0}^{n/2-1} f_{2m} e^{-\frac{2\pi i}{n/2} mk} - e^{-\frac{2\pi i}{n} k} \sum_{m=0}^{n/2-1} f_{2m+1} e^{-\frac{2\pi i}{n/2} mk}$$

$$= E_k - e^{-\frac{2\pi i}{n} k} O_k$$

4) 코드 구현 (Java)

```
public class FFT {

    // x[]의 fft 계산하는 함수, x의 길이 n은 2의 제곱수
    public static Complex[] fft(Complex[] x) {
        int n = x.length;
        Complex[] y = new Complex[n];

        // base case
        if (n == 1) return new Complex[] { x[0] };

        //  Cooley-Tukey radix 2 DIT case FFT 에서 n은 2의 제곱수여야 한다
        if (n % 2 != 0) {
            throw new IllegalArgumentException("n이 2의 제곱수가 아니다");
        }

        // 입력x를 짝수 구간으로 나눈 뒤 fft 계산
        // even[]: f2j, evenFFT[]: f2j에 exp(-ikj2PI/(n/2)) 곱한 값에 시그마한 값의
배열
        Complex[] even = new Complex[n/2];
        for (int k = 0; k < n / 2; k++) {
            even[k] = x[2 * k];
        }
        Complex[] evenPart = fft(even);

        // x를 홀수 구간으로 나눈 뒤 fft 계산
        // odd[]: f2j+1
        Complex[] odd  = even;  // n/2의 같은 크기 배열 재사용
        for (int k = 0; k < n / 2; k++) {
            odd[k] = x[2 * k + 1];
        }
        Complex[] oddPart = fft(odd);

        // 정복
        for (int k = 0; k < n / 2; k++) {
            double kth = -2 * k * Math.PI / n;  // 공식에서 exp(-ik2PI/n)을 오일러
공식으로 sin,cos관한 식으로 만들기 위한 변수
```

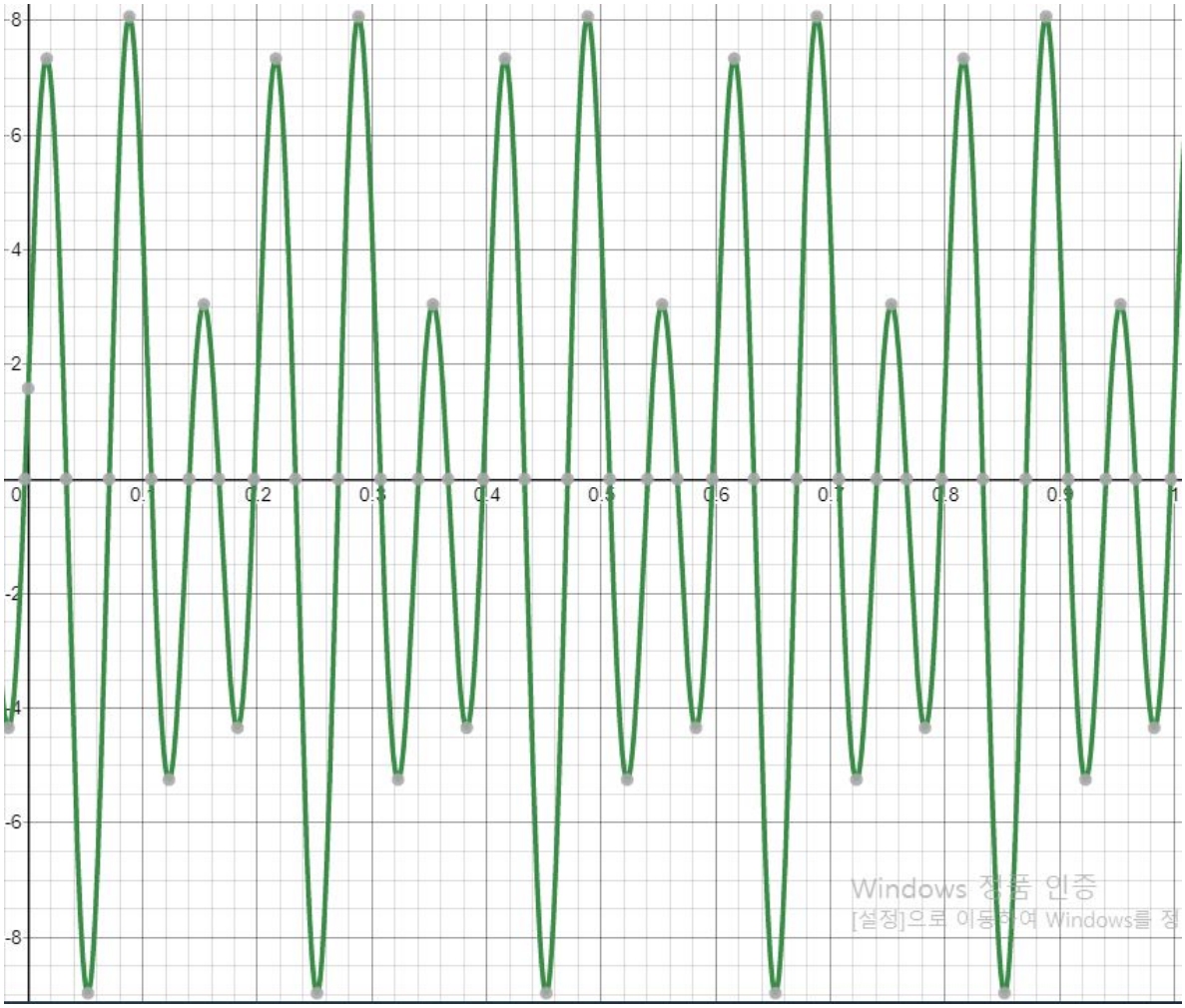
```
            Complex wk = new Complex(Math.cos(kth), Math.sin(kth)); // 오일러 공
식으로 만든 sin,cos 합성수
            y[k]       = evenPart[k].plus (wk.times(oddPart[k]));
            y[k + n/2] = evenPart[k].minus(wk.times(oddPart[k]));
        }
        return y;
    }
}

//복소수 클래스
class Complex {
    double re;
    double im;

    public Complex() {
        this(0, 0);
    }

    public Complex(double r, double i) {
        re = r;
        im = i;
    }

    public Complex plus(Complex c) {
        return new Complex(this.re + c.re, this.im + c.im);
    }

    public Complex minus(Complex c) {
        return new Complex(this.re - c.re, this.im - c.im);
    }

    public Complex times(Complex c) {
        return new Complex(this.re * c.re - this.im * c.im, this.re * c.im +
this.im * c.re);
    }
}
```

5) Example

$$x(t) = 3cos(20\pi t) + 6sin(30\pi t - 3/(4\pi)), 0 <= t <= 1$$

위 식에 대한 신호를 주파수 변환 후 `x(f)`의 그래프 그리기

우선 x(t)의 그래프를 보면 다음과 같다.



그래프를 보면 T=0.2 주기를 가지고 계속 반복하는 형태를 보인다. 따라서 x(t)의 표본을 따야 하는데 이는 n=8이라고 하면 0부터 0.2/8 = 0.025의 간격으로 표본을 구하고, 그에 대한 배열을 FFT함수에 입력값으로 넣은 후 반환된 복소수 배열의 값을 출력한 뒤 출력값을 이용해 그래프를 그리면 되겠다.

```
public static void show(Complex[] y) {
        for (int i = 0; i < y.length; i++) {
            System.out.println(("" + Math.round(y[i].re * 1000)/1000.0 +","+
Math.round(y[i].im * 1000)/1000.0 + "i"));
        }
        System.out.println();
    }


    public static void main(String[] args) {

        int n = 8;
        double t = 0.0;
        Complex[] x = new Complex[n];

        for(int i = 0; i < n; i++) {
            double cosF = 20 * Math.PI * t;
            double sinF = 30 * Math.PI * t - ((3 /(4 * Math.PI)) + (3 % (4 *
Math.PI)));
            double xt = (3 * Math.cos(cosF)) + (6 * Math.sin(sinF));

            x[i] = new Complex(xt, 0);
            t += 0.025;     //표본의 간격
        }
```

```
        Complex[] fftY = fft(x);

        show(fftY);

    }
```

실행 결과

```
0.0,0.0i
0.0,0.0i
12.0,0.0i
2.328,23.887i
0.0,0.0i
2.328,-23.887i
12.0,0.0i
0.0,0.0i
```

`n = 8` 일 때 X(f) 값의 결과를 얻을 수 있었다. 해당 결과값은 x(t)의 주기 `T=0.2` 내에서의 값으로만 구한 값이기 때문에 t의 범위가 `0<=t<=0.175` 가 된다. 주어진 문제에서 t의 범위는 `0<=t<=1` 이므로 해당 범위의 결과값은 주어진 결과값이 `t=1` 때까지 반복하는 값을 가질 것이다.

결과값 그래프 표시