알고리즘 중간 과제

서론

전문적인 개념에 대한 보고서를 작성하는 것이 처음이라서 어려움을 많이 겪었다. 공업수학에서 이제 막 해당 개념에 대한 진도를 나가고 있기 때문에 수학적인 방면으로는 매우 기초적인 개념만 이해하고 있다. 코딩 실력이 좋지 않아 스스로 모든 코드를 제작할 정도의 능력도 없다. 때문에 인터넷에서 해당 자료를 많이 참고했는데, 신뢰할 수 있는 자료인지 판별하는 것이 꽤나 버거웠다. 특히 응용 부분에 대해 작성하면서 해당 문제로 어려움을 많이 느꼈다. 때문에 이 글은 내 동기나 후배가 봐도 이해할 수 있을 정도로 간단하게 쓰는 것을 목표로 했다. 이러한 이유로 어려운 개념이 나오면 이에 대한 설명을 추가하려고 노력했다. 비록 본인도 배경지식이 부족한 상태이지만, 본인의 이해력과 전달력을 믿고 글을 써보려고 한다. 참고로 본인은 해당 개념이 왜 생기게 되었는지 앞부분에 역사를 조금 다룰 계획이다. 푸리에 변환에 대한 본격적인 이야기는 1장 2절에서부터 다루기 때문에 오직 개념에만 관심이 있는 독자들은 참고하길 바란다.

차례

---

1. Fourier Transform의 역사

핵실험금지를 위한 노력

유일하게 전쟁에 핵무기를 사용한 국가답게 처음으로 핵실험을 한 국가는 미국이다. 핵 개발이 1945년부터 본격적으로 진행되면서 전 세계적으로 국민들은 핵 전쟁에 대한 두려움을 느꼈다. 이에 세계적 평화를 위하여 국가들은 핵실험 금지에 대한 회의를 여러 차례 진행했다. 비록 소련과 미국의 의견 차이가 심했지만, 소련의 핵실험 정지 선언으로 1958년에 제네바에서 당시의 핵보유국인 미국, 영국 그리고 소련이 핵실험 금지 회의를 가졌다. 그러나 회의를 하면서도 실험소 관리 및 사찰 문제 등으로 1961년에 소련이 핵실험을 재개했고 1962년에 소련이 미사일을 쿠바에 배치하려는 것으로 냉전이 극에 달하게 된다. 이러한 냉전 속에서 핵전쟁 발발을 우려한 국가들은 1963년에 부분적인 핵실험 금지 조약(PTBT)을 체결한다. 그 내용 중 우리가 눈여겨볼 조항은 1조에 관한 내용이다.

> 대기권과 외기권 그리고 수중에서의 핵실험을 금지한다.

대기권과 외기권에서의 핵실험은 핵분열로 생성되는 동위원소를 통해 감지가 가능하다. 또한, 수중에서의 핵실험은 폭발로 인한 특수한 소음이 물을 매질로 하여 퍼지기 때문에 청음기를 통해 이를 감지할 수 있다. 하지만, 지하의 경우는 다르다. 아무리 좋은 지진계라도 핵실험과 지진을 구분하기에는 한계가 있었기 때문에 부분적인 핵실험 금지 조약을 체결할 때에는 지하에서의 핵실험을 금지할 수가 없었다. 이를 감시할 방도가 없었기 때문

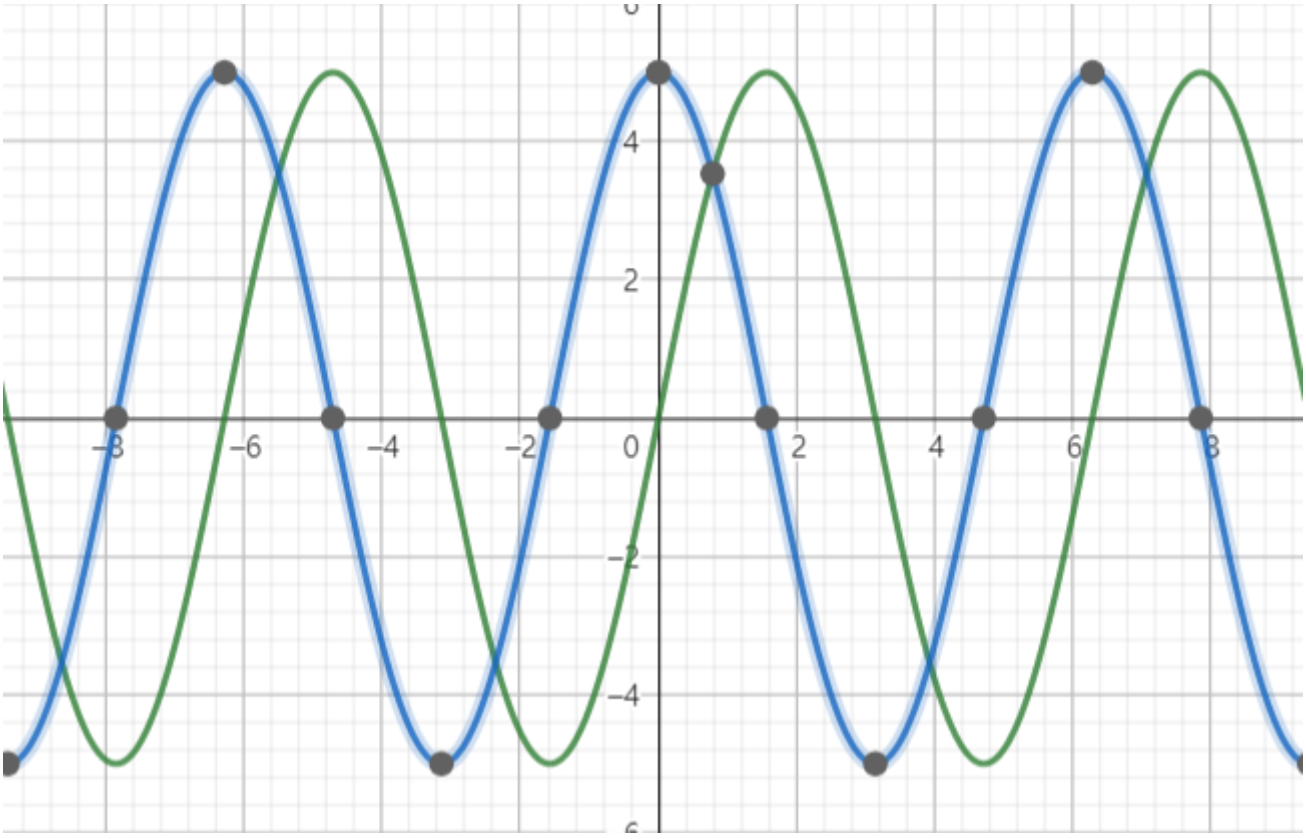
이다. 당시 시대는 2차 세계대전과 쿠바 미사일 위기로 국가 간에 냉전의 기류가 돌았기 때문에 모호한 조항이 전쟁의 빌미를 제공하는 일은 피해야 했다. 때문에 판별이 불가능한 지하에서의 핵실험을 섣부르게 금지할 수가 없었다.

신호를 쉽게 분석하기 위한 방법

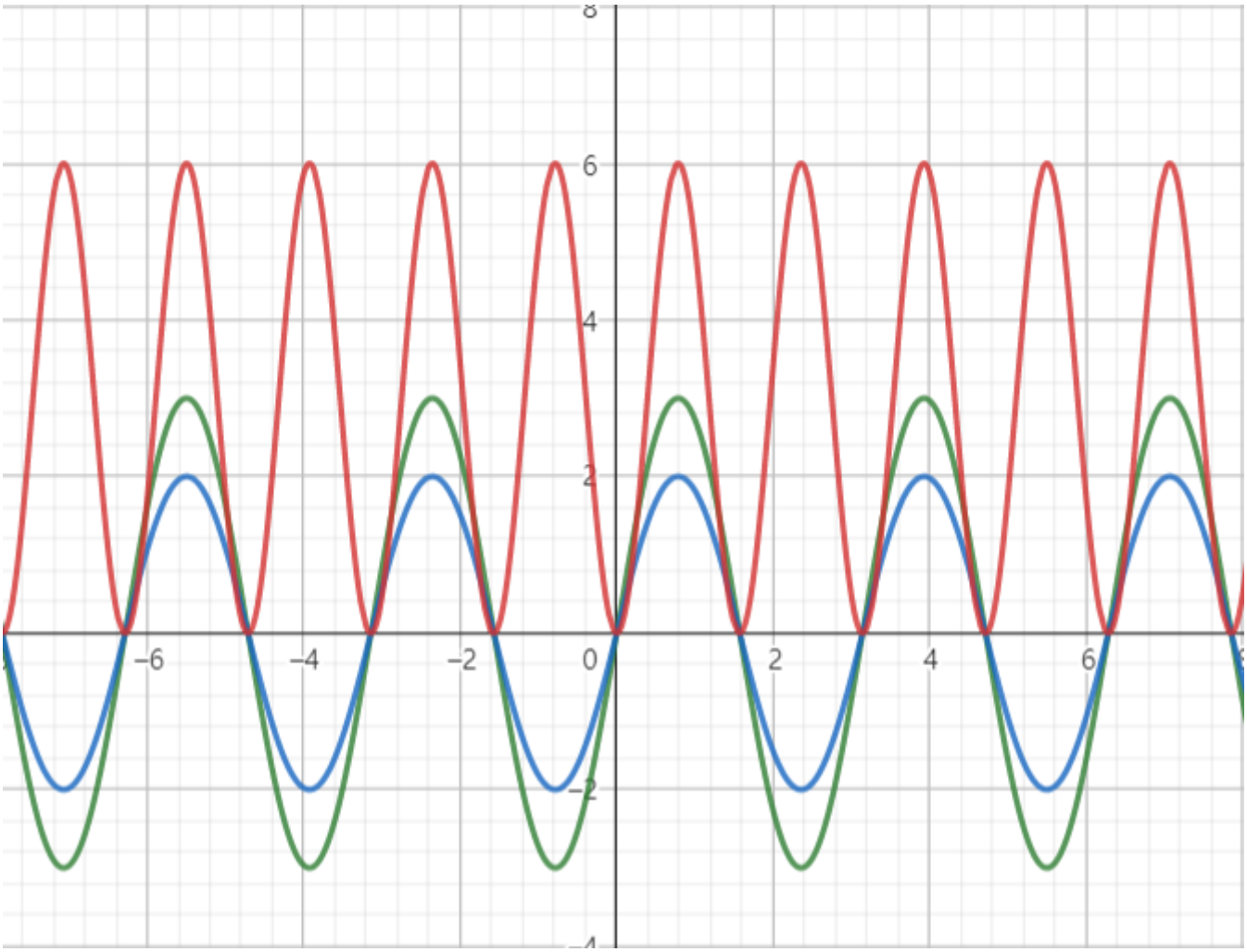
지진계로 핵실험을 감지하기 위해서는 우선 해당 신호의 근원이 무엇인지 파악해야 한다. 그 후에는 신호를 분석하여 핵실험의 정확한 위치와 규모에 대해 파악을 할 수 있어야 한다. 이러한 목표를 이루기 위하여 과학자들은 신호를 주파수에 따라 분석하려는 방법을 강구하기 시작했다.

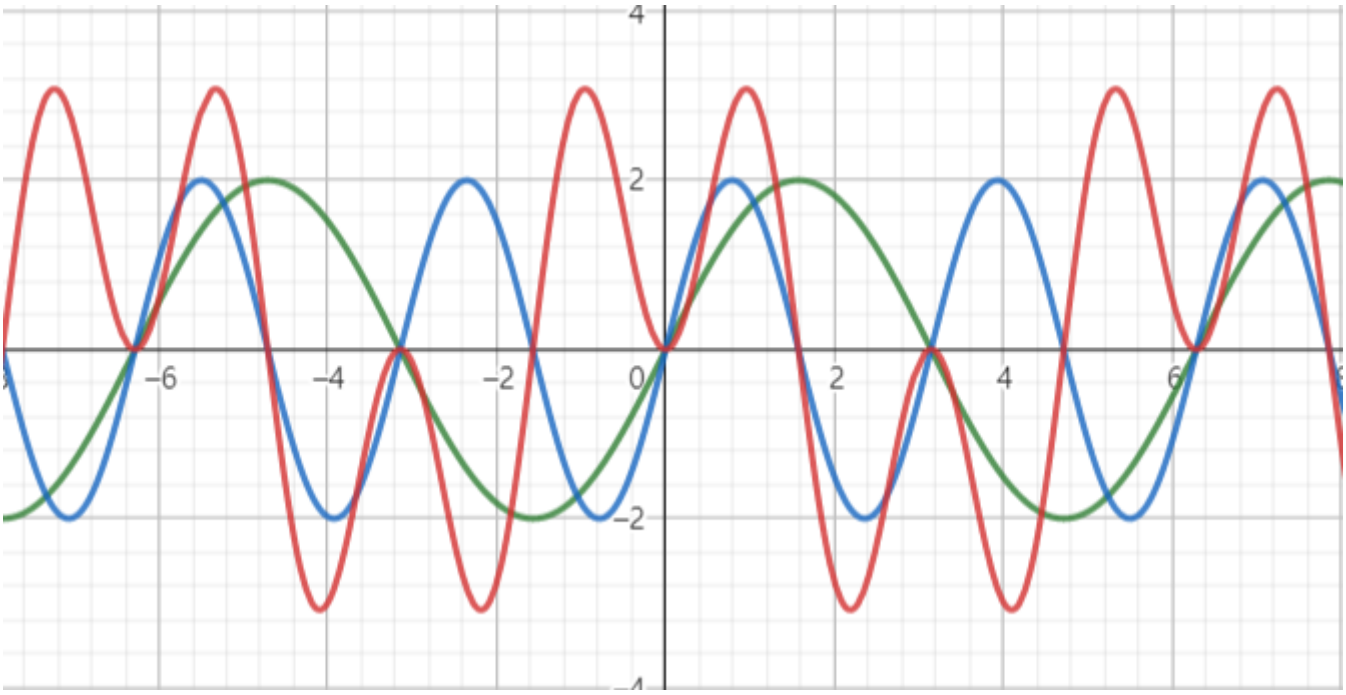
그렇게 고안한 방법이 **Fourier Transform(푸리에 변환)**이다. 푸리에 변환은 복잡한 신호, 즉 주기적이지 않은 신호를 선형적인 특성을 이용하여 여러 사인파와 코사인파로 분할하는 것이다.*(참고로 주기적인 신호는 **Fourier Series**를 통해 구한다. 당시 과학자들이 분석하려는 신호는 비주기 신호인데 이는 주기를 무한으로 취급하면 **Fourier Series**를 적용할 수 있다. 다만 **Fourier Series**에 대해서는 다루지 않을 것이다. 내용이 궁금하다면, 해당 문서를 참고하길 바란다.)*

최근에 선형대수학에서 어떠한 집합 S에 대한 1차 조합이 집합 V의 모든 요소를 표현 가능하고 S의 요소들이 서로 선형 독립이면 S는 V의 Basis(기저)라는 것을 배웠다. 선형 독립의 조건은 집합 안에 존재하는 요소들이 서로의 1차 조합으로 표현이 불가능하면 선형 독립이라고 한다. 이를 적용하기 위해서 어떠한 신호 V(signal)가 있고 집합 S(basis)가 있다고 가정하자. S = {cos a, sin a}이라고 할 때에, cos a와 sin a는 서로 선형 독립이다. 코사인과 사인은 어떤 상수배를 취해도 서로를 표현할 수 없다. 즉, mcos a ≠ nsin a이므로 독립이다.(단, m, n이 동시에 0인 경우는 제외한다.)



그리고 사인파와 코사인파가 신호의 주파수에 얼마나 기여를 하는지는 사인파 혹은 코사인파를 신호와 곱하여 나오는 파동을 적분하여 알 수 있다. 아래의 그래프에 어떠한 신호와 주파수가 다른 사인파 그리고 그들의 곱을 표현했다. 이를 통해서 주파수가 다르다면 적분값이 0이 된다는 사실을 알 수 있다. 이는 해당 사인파가 신호에 전혀 기여를 하지 않는 것을 의미한다. 반대로 주파수가 같으면 그 값은 항상 양수가 되면서 적분값이 존재하게 된다.





신호는 여러 주파수를 가지고 있으므로 실질적으로는 무수히 많은 사인파와 코사인파로 이뤄지게 된다. 한 주파수에 대응하는 사인파와 코사인파가 존재할 수도 있고 그렇지 않은 경우도 있다. 만약 사인파와 코사인파가 동시에 존재하는 경우에는 이들의 진폭을 고려해야 한다. 이 진폭은 각각 계산을 해도 되지만, 오일러 공식을 이용하면 하나의 지수항으로 처리가 가능하다.

$$e^{i\pi} = \cos x + i \sin x$$

$$Z = \cos x + i \sin x, \quad x = 0 \rightarrow Z = 1$$

$$\frac{dZ}{dx} = -\sin x + i \cos x$$

$$= i^2 \sin x + i \cos x = i(i \sin x + \cos x)$$

$$= iZ \rightarrow \frac{1}{Z} dZ = i dx$$

$$\int \frac{1}{Z} dZ = \int i dx, \quad \ln Z = ix + C$$

$$C = 0 \quad (\because \ln 1 = 0 = i \cdot 0 + C)$$

$$\therefore \ln Z = ix \rightarrow Z = e^{ix}$$

즉, 오일러 공식을 통하여 주파수에 대한 사인파와 코사인파, 각각의 파동의 진폭을 구할 수 있다. 이때에 실수부는 코사인파의 진폭이고 허수부는 사인파의 진폭이다.

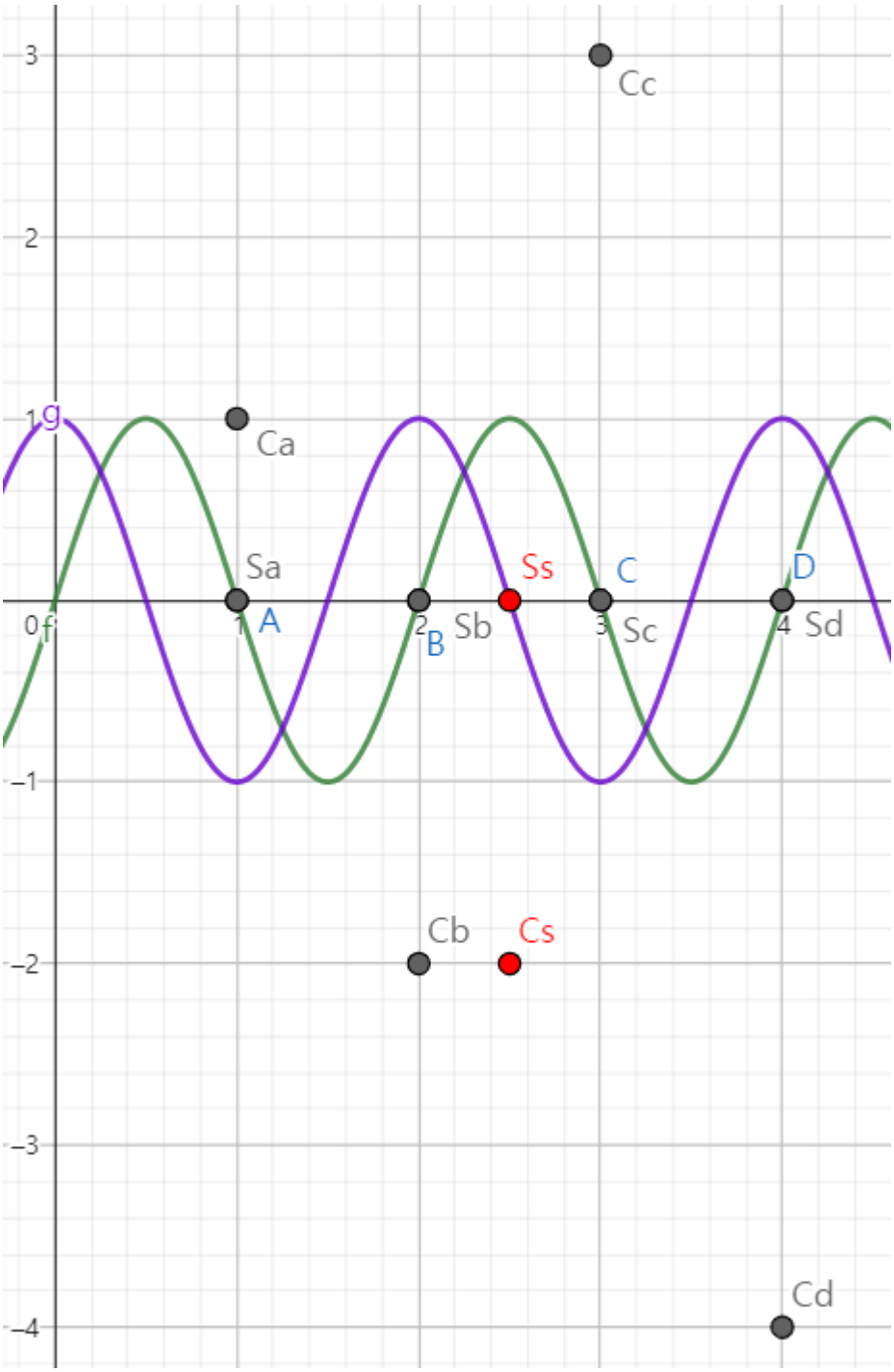
2. Discrete Fourier Transform란 무엇인가?

데이터 측정의 한계를 극복하려면

푸리에 변환의 발명으로 과학자들은 주파수를 기준으로 특정 신호를 사인파와 코사인의 집합으로 분할이 가능해졌다. 덕분에 신호를 분석하기에 매우 용이해졌다. 한편, 지진계와 같은 측정기가 감지한 신호는 언뜻 연속적으로 보이지만, 이산적인 여러 데이터 포인트의 집합일 뿐이다. 그저 데이터 포인트 사이의 간격이 0에 수렴하기 때문에 연속적으로 인식하게 되는 것이다. 즉, 우리가 얻은 신호는 연속적이지 못하다. 게다가 과학자들은 신호에서 분석하고 싶은 일부분만을 추출하여 분석한다. 불필요한 연산을 줄여 적은 노력으로 빠른 분석을 마치기 위함이다. 따라서 분석할 신호는 유한적이다. 분석할 신호가 이산적이고 유한적이므로, 주파수 또한 이산적이고 유한적인 상태가 된다. 이러한 상태에서는 푸리에 변환을 이상적으로 적용하기 어렵다.

때문에 과학자들은 푸리에 변환을 적용하기 위하여 **Discrete Fourier Transform(이산 푸리에 변환)**을 사용한다. **이산 푸리에 변환**을 적용하기 위해서는 한 가지 특성에 대해서 이해할 필요가 있다. 사인파와 코사인파는 선형성으로 인하여 각각의 배수, 즉 진폭만큼 측정한 신호에 기여한다. 따라서 하나의 주파수에 대해서 사인파와 코사인파가 대응하므로 데이터 포인트의 간격이 0으로 수렴하여 연속된 것처럼 보일수록 주파수 영역에서 주파수의 간격도 좁아진다. 이를 통해 고주파수에 대한 분석이 가능하게 된다. 반대로 데이터 포인트의 간격이 넓을 경우에는 미세한 변화까지 감지하지 못하게 되고 결과적으로 주파수 영역에서 주파수의 간격도 넓어진다. 정리하자면 주파수의 간격, 즉 정확도는 데이터 포인트의 간격과 비례한다. 추가로 주파수는 선형성으로 인해 가장 낮은 주파수의 정수배로 표현이 가능하고 주파수 영역에서의 포인트의 개수는 데이터 포인트의 개수와 같다.

본격적으로 **이산 푸리에 변환**을 시작하면, 주파수 영역에서의 포인트를 x좌표로 나열하고 그에 대한 값을 y좌표로 한다. 이때에 y좌표의 값은 데이터 포인트의 값에 해당 주파수의 사인파와 코사인파를 각각 곱한 것에 대한 합으로 나타난다. 예를 들어, 주기가 2인 사인파와 코사인 파에 대해 신호를 분석해보자. 신호를 (1, 0), (2, 0), (3, 0), (4, 0)이라는 데이터 포인트로 볼 수 있을 때, 각 점에 대한 사인파와 코사인파의 기여도를 동일한 x축에 찍는다. 점 A(1, 0)에 대해서는 사인파가 0, 코사인파가 1만큼 기여를 한다. 이처럼 각각의 점에 대해 기여도를 계산한 뒤, 그들의 총합을 한눈에 보기 좋도록 x=2.5에서 붉은 점으로 나타냈다. 결과적으로 이산 푸리에 변환을 실행하면 결과는 -2+0i가 된다. 왜냐하면, 실수부가 코사인파의 기여도이고 허수부가 사인파의 기여도이기 때문이다.



이때에 시간 복잡도는 주파수의 영역에 대한 포인트의 수 × 데이터 포인트의 수이다.(*정확히는 데이터 포인트의 수 +1의 제곱이다.*) 이 둘의 수는 같기 때문에 시간 복잡도는 $O(n^2)$이다. 해당 코드를 살펴보면 더 쉽게 알 수 있다.

```
package midTest.DFT;

public class DFT {
    static class Complex {
        private double real;
        private double imag;

        public Complex(double real, double imag) {
            this.real = real;
            this.imag = imag;
```

```
        }

        public double getReal() {
            return real;
        }

        public double getImag() {
            return imag;
        }

        public Complex add(Complex other) {
            return new Complex(real + other.getReal(), imag + other.getImag());
        }

        public Complex subtract(Complex other) {
            return new Complex(real - other.getReal(), imag - other.getImag());
        }

        public Complex multiply(Complex other) {
            double realPart = real * other.getReal() - imag * other.getImag();
            double imagPart = real * other.getImag() + imag * other.getReal();
            return new Complex(realPart, imagPart);
        }

        @Override
        public String toString() {
            if (imag >= 0)
                return String.format("%.4f + %.4fi", real, imag);
            else
                return String.format("%.4f - %.4fi", real, Math.abs(imag));
        }
    }

    public static Complex[] dft(Complex[] x) {
        int n = x.length;
        Complex[] y = new Complex[n];

        for (int k = 0; k < n; k++) {
            y[k] = new Complex(0, 0);
            for (int j = 0; j < n; j++) {
                double theta = 2 * Math.PI * j / n * k;
                y[k] = y[k].add(x[j].multiply(new Complex(Math.cos(theta),
Math.sin(theta))));
            }
        }

        return y;
    }

    public static void main(String[] args) {
        Complex[] x = { new Complex(1.0, 0.0), new Complex(2.0, 0.0), new
Complex(3.0, 0.0), new Complex(4.0, 0.0) };
        Complex[] y = dft(x);
        for (Complex c : y) {
```

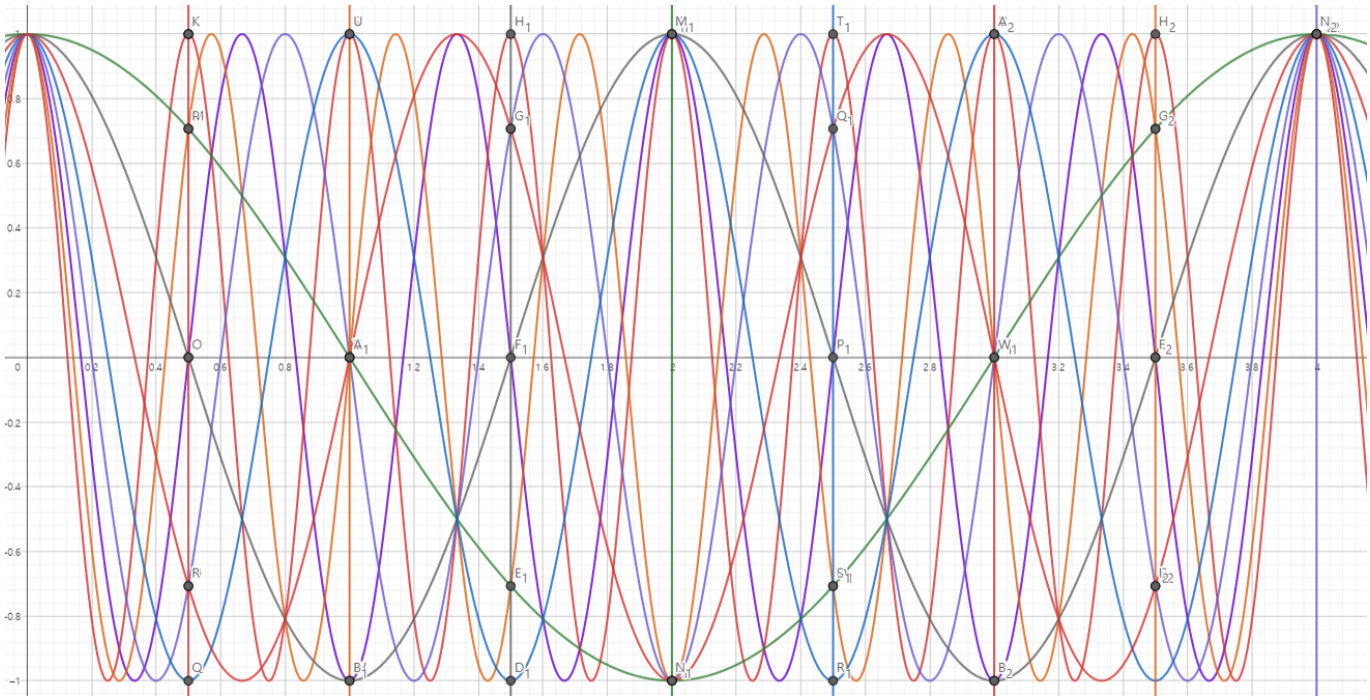
```
                System.out.println(c);
            }
        }
    }
```

dft 함수 안에는 이중 for문이 존재한다. 부수적인 연산을 제외하면 이중 for문으로 인하여 데이터 포인트의 수에 대한 제곱만큼의 시간이 걸리게 된다. 예를 들어, 데이터 포인트가 2,048개이면 연산은 4,194,304 + a이다. 하나의 연산에 0.1초가 걸린다면 419,430.4초, 쉽게 말해서 약 116.5시간 정도가 걸린다.

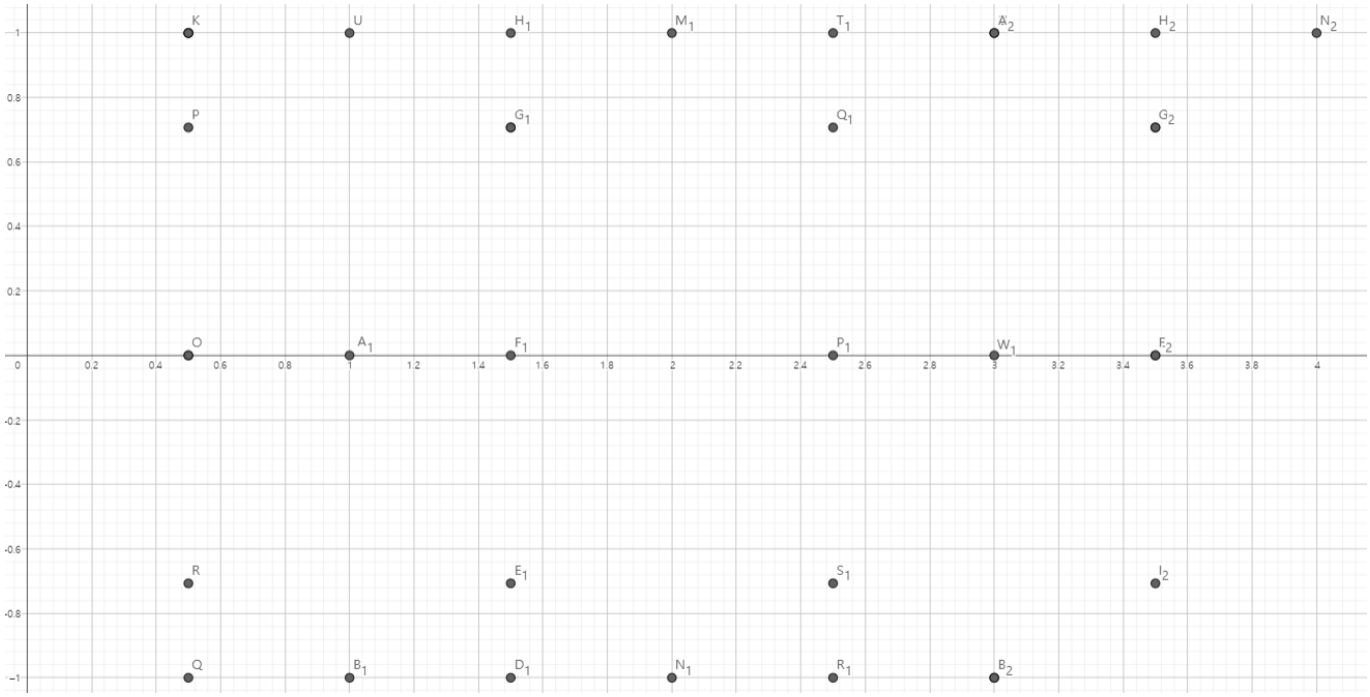
3. Fast Fourier Transform란 무엇인가?

시간 복잡도의 한계를 극복하려면

드디어 과학자들은 신호에서 주파수에 대한 정보를 추출하여 분석할 수 있게 됐다. 하지만, 데이터 포인트가 많아질수록 연산의 회수가 크게 증가한다. 정확도를 높이기 위해서는 많은 데이터 포인트에 대해 분석을 할 필요가 있었기 때문에 과학자들은 더욱 효율적인 계산 방법이 필요했다. 그리고 여러 시행착오 끝에 한 사람이 이산 푸리에 변환에서 한 가지의 특징을 알아낸다. 바로 주파수가 특정한 지점에서 같은 값을 가진다는 것이다. 예를 들어 이산 푸리에 변환을 한다면 아래와 같은 위치에서 연산이 발생한다.

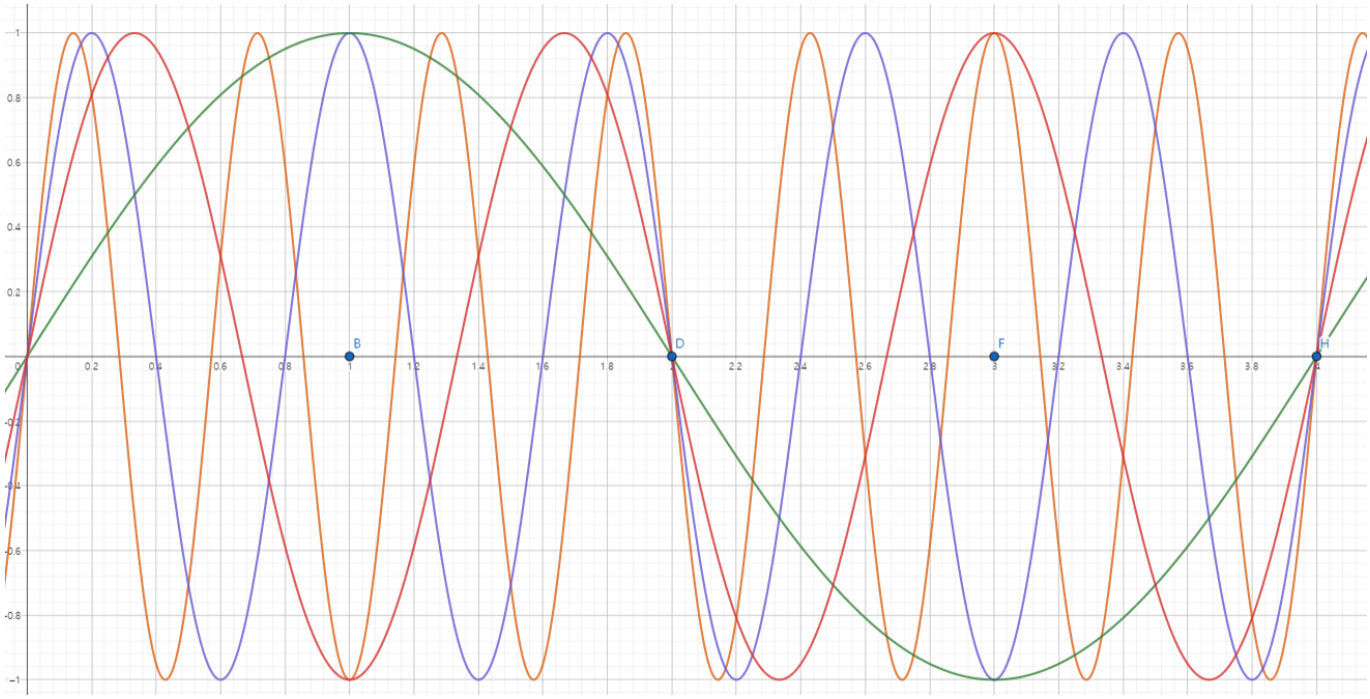


한눈에 보기 좋도록 그래프를 지우고 접점만을 남기면 다음과 같은 점들을 얻을 수 있다.



이산 푸리에 변환이라면 9×9로 총 81번의 연산이 필요하다. 하지만 연산 중에 겹치는 부분이 존재한다. 예를 들어, (2, 0)에서 홀수번째의 주파수와 짝수번째의 주파수끼리 같은 값을 가진다. 결국 2번의 연산만이 필요하다는 것을 알 수 있다. 심지어 두 결과값의 절대값은 같다. 따라서 연산을 1번만 하면 그 값에 -를 붙여 모든 결과값을 도출해낼 수 있다. 게다가 (4, 0)에서는 연산이 1번으로 끝난다. 결과적으로 연산의 회수는 28회이고 시간 복잡도가 $O(n\log_2 n)$이다. 예를 들어, 데이터 포인트가 2,048개이면 22,528 + a의 연산이 필요하고 연산 1회에 0.1초가 필요하다면 2,252.8초가 필요하다. 즉, 37.55분 정도가 걸리게 된다. 이산 푸리에 변환에서는 약 4~5일이 걸리지만, 고속 푸리에 변환으로는 1시간도 걸리지 않는다.

고속 푸리에 변환은 분할 정복 알고리즘이다. 그 이유는 홀수번째와 짝수번째의 데이터 포인트에서 결과가 겹치기 때문이다. 아래는 짝수번째 데이터 포인트에서 사인파에 대한 분석이다. 사인파의 짝수번째 주파수에서도 다시 순서를 매기면, 홀수번째 주파수와 짝수번째 주파수의 값끼리 겹치는 부분이 생긴다는 것을 알 수 있다. 특히 홀수번째의 경우, 양의 값과 음의 값은 정확하게 -1배이다. 즉, 1번 계산을 한 뒤에는 이 값을 참조하면 된다. 이를 통하여 계산을 절반으로 줄일 수 있다. 이를 계속해서 반복하면 주파수에 대한 연산을 n번에서 $\log_2 n$번으로 줄일 수 있다. 이를 데이터 포인터만큼, n번 반복하므로 결과적으로는 시간 복잡도가 $O(n\log_2 n)$이다.




```java
package midTest.FFT;

public class FFT {
    static class Complex {
        private double real;
        private double imag;

        public Complex(double real, double imag) {
            this.real = real;
            this.imag = imag;
        }

        public double getReal() {
            return real;
        }

        public double getImag() {
            return imag;
        }

        public Complex add(Complex other) {
            return new Complex(real + other.getReal(), imag + other.getImag());
        }

        public Complex subtract(Complex other) {
            return new Complex(real - other.getReal(), imag - other.getImag());
        }

        public Complex multiply(Complex other) {
            double realPart = real * other.getReal() - imag * other.getImag();
            double imagPart = real * other.getImag() + imag * other.getReal();
            return new Complex(realPart, imagPart);
        }
```

```
        @Override
        public String toString() {
            if (imag >= 0)
                return String.format("%.4f + %.4fi", real, imag);
            else
                return String.format("%.4f - %.4fi", real, Math.abs(imag));
        }
    }

    public static Complex[] fft(Complex[] x) {
        int n = x.length;

        if (n == 1) {
            return x;
        }

        Complex[] even = new Complex[n / 2];
        Complex[] odd = new Complex[n / 2];
        for (int i = 0; i < n / 2; i++) {
            even[i] = x[2 * i];
            odd[i] = x[2 * i + 1];
        }

        Complex[] evenResult = fft(even);
        Complex[] oddResult = fft(odd);

        Complex[] result = new Complex[n];
        for (int k = 0; k < n / 2; k++) {
            double theta = 2 * Math.PI / n * k;
            Complex t = new Complex(Math.cos(theta), Math.sin(theta));
            result[k] = evenResult[k].add(t.multiply(oddResult[k]));
            result[k + n / 2] = evenResult[k].subtract(t.multiply(oddResult[k]));
        }

        return result;
    }

    public static void main(String[] args) {
        Complex[] x = { new Complex(1.0, 0.0), new Complex(2.0, 0.0), new
Complex(3.0, 0.0), new Complex(4.0, 0.0) };
        Complex[] y = fft(x);
        for (Complex c : y) {
            System.out.println(c);
        }
    }
}
```

위의 코드를 통하여 분할 정복이 가능하다. 주파수가 홀수번째인 경우와 짝수번째인 경우로 분할하는 과정에서 재귀 함수가 사용된다. 여기서 주의할 점은 입력되는 데이터 포인트의 개수가 2^n개일 때에만 알고리즘을 돌릴 수 있다는 점이다. 그렇지 않으면 NullPointerException이 발생하기 때문이다. 때문에 홀짝으로 완벽하게 나눌 수 있도록 2^n개의 데이터를 입력받아야 한다. 만약 데이터가 2^n개가 아니라면, 아무런 의미가 없는 (0, 0)에 대한

데이터를 삽입한다. 데이터 포인터에 사인파와 코사인파를 곱하면 값은 결과적으로 0이 되기 때문에 의미가 없다.

FFT가 적용되는 사례에는 어떤 것이 있을까?

너무나도 늦은 발견, 그러나 중요한 발전

```
Run | Debug
74      public static void main(String[] args) {
75          Complex[] x = new Complex[2048];
76          Random r = new Random();
77          for (int i = 0; i < x.length; i++) {
78              x[i] = new Complex(r.nextDouble(), imag:0.0);
79          }
80          long startTime = System.currentTimeMillis();
81          fft(x);
82          long endTime = System.currentTimeMillis();
83          long time = endTime - startTime;
84          System.out.printf(format:"실행 시간 : %d", time);
```

```
TERMINAL  PROBLEMS  DEBUG CONSOLE  OUTPUT

새로운 크로스 플랫폼 PowerShell 사용 https://aka.ms/pscore6

PS D:\pyfile>  & 'C:\Program Files\Java\jdk-14.0.2\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '
-cp' 'C:\Users\user\AppData\Roaming\Code\User\workspaceStorage\3c8bb79d3793f9c3b07bff478ef3cb3a\redhat.java\
jdt_ws\pyfile_676e5ebe\bin' 'midTest.FFT.FFTTest'
실행 시간 : 59
PS D:\pyfile>
```

```
Run | Debug
61      public static void main(String[] args) {
62          Complex[] x = new Complex[2048];
63          Random r = new Random();
64          for (int i = 0; i < x.length; i++) {
65              x[i] = new Complex(r.nextDouble(), imag:0.0);
66          }
67          long startTime = System.currentTimeMillis();
68          dft(x);
69          long endTime = System.currentTimeMillis();
70          long time = endTime - startTime;
71          System.out.printf(format:"실행 시간 : %d", time);
72      }
73  }
```

```
TERMINAL  PROBLEMS  DEBUG CONSOLE  OUTPUT

새로운 크로스 플랫폼 PowerShell 사용 https://aka.ms/pscore6

PS D:\pyfile>  & 'C:\Program Files\Java\jdk-14.0.2\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '
-cp' 'C:\Users\user\AppData\Roaming\Code\User\workspaceStorage\3c8bb79d3793f9c3b07bff478ef3cb3a\redhat.java\
jdt_ws\pyfile_676e5ebe\bin' 'midTest.FFT.FFTTest'
실행 시간 : 59
PS D:\pyfile>
```

위의 사진을 통하여 고속 푸리에 변환이 매우 효율적이라는 사실을 알 수 있다. 따라서 과학자들은 고속 푸리에 변환을 통해 지하에서의 핵실험을 빠른 시간으로 감지할 수 있게 되었다. 이를 통하여 1996년에 포괄적 핵실험 금지 조약(CTBT)을 체결하면서 모든 핵실험이 금지되었다. 물론 북한과 같이 조약에 협조하지 않는 국가도 있지만, 미국과 소련에서 주로 이뤄지던 핵실험의 통제는 고속 푸리에 변환의 가장 큰 수확이라 볼 수 있다. 다만 아쉬운 점은 부분적인 핵실험 금지 조약으로부터 푸리에 변환을 발명까지 30년 정도의 시간이 필요했고 그 사

이에 핵실험이 약 1400회 정도 있었다는 점이다.(*핵실험 연대기 (참고)*, *CTBTO 에서 측정한 핵실험 회수 (2000회)*) 그럼에도 더 이상의 발전을 막아주는 고속 푸리에 변환은 핵전쟁으로부터 우리를 안전하게 해주고 있다.

정보화 시대에서의 응용

고속 푸리에 변환은 지하에서의 핵실험을 감지하는 것 이외에도 여러 방면에서 응용된다. 가장 널리 적용된 사례는 신호를 분석하는 데에 사용되는 것이다. 예를 들어, 소리에 대한 신호를 분석하여 특정 주파수를 감지할 수 있다. 이를 통하여 음향 관련 종사자들이 믹싱을 더 효율적으로 할 수 있도록 도울 수 있다. 당연히 특정 주파수를 감지하는 것이 가능하므로 음성 인식 및 변조 등에도 활용할 수 있다. 그리고 뇌파 및 맥박을 분석하여 문제점이 있는지 파악이 가능하다. CT나 MRI 그리고 초음파 검사와 같이 주파수를 이용한 의료 기구에 대하여 고속 푸리에 변환을 적용하여 결과를 분석하는 것이 용이하다.

위를 뒷바침하는 논문들이 몇 가지 있다. 일반적으로 레이더는 주파수를 감지하는 기기이므로 이를 구현하기 위해서는 고속 푸리에 변환이 필수적이다. 해당 논문은 레이더를 구현하는 방식에 대해 설명하고 있다. 또한, 철도의 위치를 파악하는 기기인 지상자에도 고속 푸리에 변환이 사용된다. 이 논문을 참고하면, 철도 특성상 주변의 소음이 신호 감지를 방해하기 때문에 잡음의 간섭을 최소화하도록 고속 푸리에 변환을 적용하고 있다. 그 외에도 제품의 제조 진동수를 분석 경우가 있다. 이 논문은 고속 푸리에 변환을 통해서 제품별 진동 데이터를 주파수 영역을 변환하여 비교한다. 이를 통해 두 제품의 상관계수를 분석할 수 있고 머신러닝과 딥러닝의 입력 데이터로 사용할 수 있다.

앞서 언급한 것처럼 고속 푸리에 변환은 딥러닝과도 연관이 깊다. 이에 대한 예시로는 컨벌루션 연산이 있는데, 컨벌루션 신경망(CNN)은 이미지 처리에서 우수한 성능을 보인다. 해당 논문에서는 컨벌루션 연산을 최적화하기 위하여 고속 푸리에 변환을 적용시킨다. 고속 푸리에 변환을 적용한 경우를 Fourier CNN이라고 부른다. CNN과 Fourier CNN의 정확도는 약 2%의 차이로 오차가 크지 않은 것에 비해서 연산 속도는 약 7%의 차이를 보인다고 논문의 결론부에 정리되어 있다. 즉, 고속 푸리에 변환은 연산을 줄이는 방식으로 많이 사용되기 때문에 데이터 분석 및 딥러닝을 최적화하기 위하여 사용할 수 있다.

게다가 고속 푸리에 변환은 압축 및 복원에도 쓰일 수 있다. 예를 들어, 이미지의 픽셀에 대한 정보를 주파수처럼 취급하면 이를 고속 푸리에 변환으로 단순화시킬 수 있다. 이를 역으로 복호화를 하는 데에 적용해도 원본을 잘 표현할 수 있다. 이는 음성에 대해서도 적용이 가능하다. 따라서 고속 푸리에 변환을 통해 분석과 압축 등을 빠르게 할 수 있다는 점에서 고속 푸리에 변환은 매우 유용하다.

마지막으로 양자후암호화(PQC)에 대한 알고리즘 중 하나인 FALCON 알고리즘에서 고속 푸리에 연산이 사용된다. 비록 해당 논문에서 고속 푸리에 변환으로도 연산을 가속하는 것이 한계가 있어 이를 더욱 가속시키기 우한 HW를 설계하지만, 기본적으로 알고리즘을 수행하기 위하여 고속 푸리에 연산을 사용한다.