**Maciej Kozub**

# Laboratorium 8 – Szybka transformata Fouriera

## 1. DFT

Klasa reprezentująca DFT składa się z wektora danych wejściowych, wektora wyników oraz metody obliczającej ciąg wynikowy.

Metoda calculate() dla każdego elementu ciągu wejściowego wykonuje sumowanie zgodnie ze wzorem:

$$
\begin{aligned}
X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N} \\
&= \sum_{n=0}^{N-1} x_n \cdot [\cos(2\pi kn/N) - i \cdot \sin(2\pi kn/N)]
\end{aligned}
$$

Kod źródłowy:

```
void DFT::init(std::vector<double>& input){
    this->samples = input;
    this->results.resize((unsigned) input.size());
}

void DFT::calculate(){
    auto size = (unsigned) samples.size();
    for(int i = 0; i < size; i++){
        std::complex<double> sum( r: 0,  i: 0);
        for(int j = 0; j < size; j++){
            sum += (std::polar( rho: 1.0,  theta: (-2 * M_PI * i * j) / size) * samples[j]);
        }
        results[i] = sum;
    }
}

std::vector<std::complex<double>> DFT::getResults(){
    return results;
}
```

Teoretyczna złożoność obliczeniowa tego algorytmu to O(n2), gdzie n to liczba elementów ciągu wejściowego.

## 2. FFT

Modyfikacja metody DFT wykorzystująca symetrię do ulepszenia złożoności obliczeniowej problemu dzięki zastosowaniu algorytmu Cooleya-Tukeya. Ciąg obliczeń zostaje rozbity na dwa pod problemy, a te dalej na dwa - i tak aż do pewnej liczby elementów transformowanego ciągu, dla której stosujemy zwykły DFT - dzięki temu stosujemy podejście rekurencyjne i znacząco ulepszamy złożoność obliczeniową.

Metoda ta bazuje na symetrii:

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-i\,2\pi\,k\,n\,/\,N} \\ &= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i\,2\pi\,k\,(2m)\,/\,N} + \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i\,2\pi\,k\,(2m+1)\,/\,N} \\ &= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i\,2\pi\,k\,m\,/\,(N/2)} + e^{-i\,2\pi\,k\,/\,N} \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i\,2\pi\,k\,m\,/\,(N/2)} \end{aligned}$$

Kluczowa dla klasy FFT jest metoda fft, która dokonuje podziału na dwa podproblemy, wykonuje dla nich obliczenia (poprzez wywołanie rekurencyjne), a następnie łączy otrzymane wyniki w zwracany ciąg wynikowy.

Kod źródłowy:

```
void FFT::init(std::vector<double>& input){
    this->results.resize((unsigned) input.size());

    for(unsigned i = 0; i < input.size(); i++){
        results[i] = std::complex<double>(input[i], 0);
    }
}

void FFT::calculate(){
    fft(results);
}
```

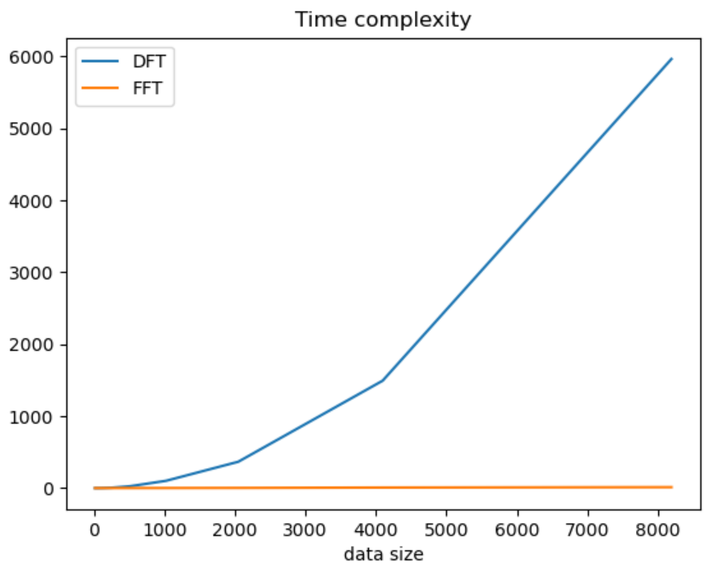
```cpp
void FFT::fft(std::vector<std::complex<double>>& values){
    const unsigned N = values.size();
    if (N <= 1)
        return;

    std::vector<std::complex<double>> even, odd;
    for(int i = 0; i < N; i++){
        if(i % 2 == 0)
            even.push_back(values[i]);
        else
            odd.push_back(values[i]);
    }

    fft(even);
    fft(odd);

    for (unsigned i = 0; i < N/2; i++){
        std::complex<double> tmp = std::polar(1.0, -2 * M_PI * i / N) * odd[i];
        values[i] = even[i] + tmp;
        values[i + N / 2] = even[i] - tmp;
    }
}

std::vector<std::complex<double>> FFT::getResults(){
    return results;
}
```

```cpp
std::vector<double> FFT::getSpectrum(){
    std::vector<double> spectrum;
    spectrum.resize(results.size());
    for(int i = 0; i < spectrum.size(); i++){
        spectrum[i] = abs(results[i]);
    }
    return spectrum;
}
```

Teoretyczna złożoność obliczeniowa FFT to O(n log n) - dzięki zastosowaniu rekurencji zamiast obliczania wprost z definicji (jak w przypadku DFT).

## 3. Porównanie metod

```
Time for 16 samples:
        DFT: 0.0227 ms
        FFT: 0.0213 ms
Time for 32 samples:
        DFT: 0.0906 ms
        FFT: 0.0419 ms
Time for 64 samples:
        DFT: 0.3666 ms
        FFT: 0.0765 ms
Time for 128 samples:
        DFT: 1.4825 ms
        FFT: 0.1656 ms
Time for 256 samples:
        DFT: 6.0015 ms
        FFT: 0.5052 ms
Time for 512 samples:
        DFT: 26.0882 ms
        FFT: 0.745 ms
Time for 1024 samples:
        DFT: 101.843 ms
        FFT: 1.5511 ms
Time for 2048 samples:
        DFT: 366.502 ms
        FFT: 2.9006 ms
Time for 4096 samples:
        DFT: 1493.3 ms
        FFT: 8.1241 ms
Time for 8192 samples:
        DFT: 5964.59 ms
        FFT: 13.0623 ms
```



Powyższy wykres obrazuje rozbieganie się czasu wykonania metod DFT  i  FFT.

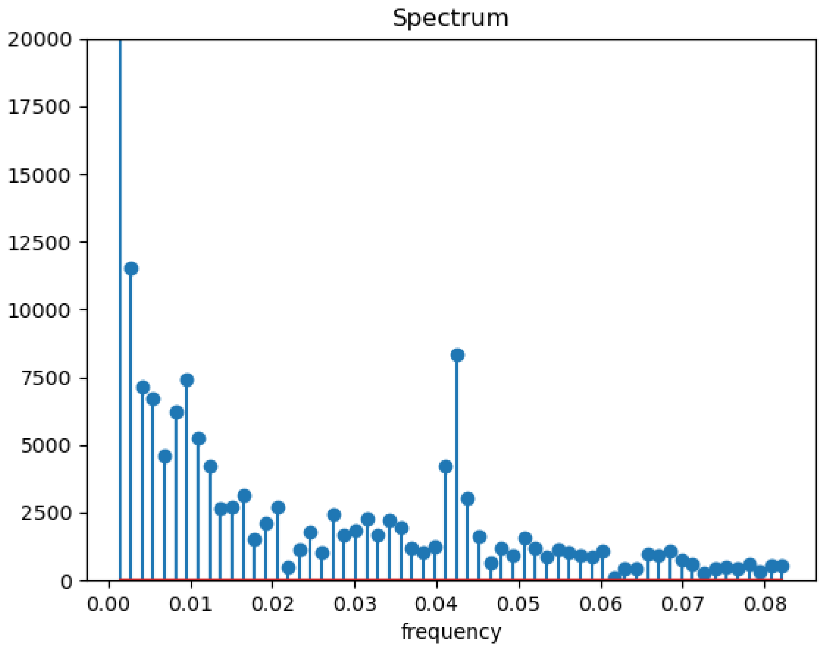
Widać, że złożoność czasowa metody DFT ma charakterystyczną tendencje wzrostową paraboli, a FFT jest praktycznie liniowa (n log n). Otrzymane wyniki są zgodne z założeniami teoretycznymi.

Metoda DFT wykonuje obliczenia naiwnie, zgodnie z definicją transformaty. Metoda FFT wykorzystuję symetrię i stosując metodę „dziel i zwyciężaj", dzieli problem na pod problemy rekurencyjnie - osiągając dużą lepszą złożoność czasową.

## 4. Analiza szeregu czasowego

a) Wykorzystałem dane pozyskane z Instytutu Meteorologii i Gospodarki Wodnej (https://dane.imgw.pl/datastore). Dane prezentują historyczne pomiary temperatury (co 10 minut) z wybranej stacji pomiarowej w lipcu 2019 roku.

b) Uzyskany wykres (do rysowania wykorzystałem Python'a)




c) Jak widać dane powtarzają się z częstotliwością nieco ponad 0,04 1/h co oznacza, że zbliżona temperatura powtarza się co około 24 godziny. Ma to jak najbardziej sens. Dodatkowo widzimy duże zagęszczenie w pobliży 0 – co sugeruje, że spora część danych jest losowa – dane nie powtarzają się.