Laboratorium 8: Szybka transformata Fouriera

Metody Obliczeniowe w Nauce i Technice

Wiktor Tarsa

1. **Dyskretna transformata Fouriera**

Zarówno dziedziną jak i zbiorem wartości DFT jest zbiór liczb zespolonych. Użyłem biblioteki complex do przedstawienia liczb zespolonych w programie.
Zaimplementowałem funkcję, która wykonuje DFT na podanym zbiorze liczb zespolonych.

```
std::vector<std::complex<double>>
computeDFT(std::vector<std::complex<double>> x) {
    std::vector<std::complex<double>> X;
    std::complex<double> zero(0, 0);
    int N = x.size();
    X.resize(N, zero);


    for (int k = 0; k < N; k++) {
        for (int n = 0; n < N; n++) {
            std::complex<double> f(cos(2 * M_PI * k * n / N), -sin(2 * M_PI
*   k * n / N));
            X[k] += x[n] * f;
        }
    }

    return X;
}
```

Funkcja computeDFT korzysta z zależności opisanej we wzorze Eulera. Główna pętla funkcji to tak naprawdę zależność opisana poniżej:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i 2\pi k n / N}$$
$$= \sum_{n=0}^{N-1} x_n \cdot [\cos(2\pi k n / N) - i \cdot \sin(2\pi k n / N)],$$

Do weryfikacji poprawności funkcji użyłem danych z przykładu z wikipedii. W tym celu zaimplementowałem funkcję która zwraca wektor z tymi danymi. Utworzyłem też prostą funkcję która wypisuje zawartość wektora liczb zespolonych.

```
std::vector<std::complex<double>> wikipediaExample() {
    std::vector<std::complex<double>> numbers;
    numbers.resize(4);
    numbers[0] = std::complex<double>(1, 0);
    numbers[1] = std::complex<double>(2, -1);
    numbers[2] = std::complex<double>(0, -1);
    numbers[3] = std::complex<double>(-1, 2);
    return numbers;
}

void printComplexVector(std::vector<std::complex<double>> x) {
    for (int i = 0; i < x.size(); i++) {
        if (x[i].real() != 0) std::cout << x[i].real();
        if (x[i].real() != 0 and x[i].imag() != 0) std::cout << " + ";
        if (x[i].imag() != 0) {
            std::cout << x[i].imag() << "i";
        }
        std::cout << std::endl;
    }
}
```

Otrzymane wyniki są prawie takie same jak oczekiwane:

```
2
-2 + -2i
8.88178e-16 + -2i
4 + 4i
```

$$\mathbf{X} = \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -2 - 2i \\ -2i \\ 4 + 4i \end{pmatrix}$$

Trzecia liczba powinna składać się tylko z części urojonej. Drobna niezgodność wynika prawdopodobnie z błędu przybliżenia liczby PI lub błędu zaokrąglenia w trakcie operacji na liczbach zmiennoprzecinkowych.

2. **FFT**

Zaimplementowałem funkcję, która realizuje algorytm FFT:

```
void FFT(std::complex<double> *X, int N) {
   if (N >= 2) {

       std::vector<std::complex<double>> copy;
       copy.resize(N / 2);
       for (int i = 0; i < N / 2; i++) {
           copy[i] = X[i * 2 + 1];
       }
       for (int i = 0; i < N / 2; i++) {
           X[i] = X[i * 2];
       }
       for (int i = 0; i < N / 2; i++) {
           X[i + N / 2] = copy[i];
       }

       FFT(X, N / 2);
       FFT(X + N / 2, N / 2);

       for (int k = 0; k < N / 2; k++) {
           std::complex<double> e = X[k];
           std::complex<double> o = X[k + N / 2];
           std::complex<double> w = std::complex<double>(cos(2. * M_PI * k
/ N), -sin(2. * M_PI * k / N));
           X[k] = e + w * o;
           X[k + N / 2] = e - w * o;
       }
   }
}
```

Powyższa funkcja jest rekurencyjna. Aby ułatwić operacje na wektorze X - jest on przedstawiony w formie wskaźnika. Taka forma umożliwia funkcji FFT na łatwe operowanie częścią danych(po kilkukrotnym wywołaniu rekurencyjnym) bez konieczności kopiowania danych do innego wektora.

Schemat działania funkcji:
Na początku kopiuję wszystkie elementy o parzystych indeksach do pierwszej połowy wektora, natomiast te o nieparzystych indeksach - do drugiej. W ten sposób trzymam dane na których są wykonywane operacje obok siebie.
Następnie wykonuję krok rekurencyjny - wywołuję funkcję FFT dwukrotnie, na każdej z wcześniej oddzielonych połówek X.
Gdy wektor osiągnie zbyt małą długość(czyli 2) nie dochodzi do podziału. Wtedy wyznaczam wartości poszczególnych liczb.

3. **Pomiar i analiza czasów wykonania algorytmów DFT i FFT.**

Zaimplementowałem funkcję, która jako argumenty przyjmuje liczby: 1) typu int będącą seedem do generatora liczb pseudolosowych oraz 2) wielkość zbioru danych. Następnie wylosowałem zbiór o podanej wielkości i zmierzyłem czas wykonania algorytmów DFT i FFT dla tych danych.

```
void DFTandFFTtest(int seed, int n){
    srand(seed);
    std::vector<std::complex<double>> X;
    std::complex<double> X2[n];
    for(int i = 0; i < n; i++){
        auto c = std::complex<double>(1+rand()%100, 1+rand()%100);
        X.push_back(c);
        X2[i] = c;
    }

    std::cout << "wielkość zbioru: " << n << std::endl;
    auto start = std::chrono::steady_clock::now();
    computeDFT(X);
    auto end = std::chrono::steady_clock::now();
    std::cout << "DFT: "
              << std::chrono::duration_cast<std::chrono::microseconds>(end -
start).count()
              << " µs" << std::endl;

    start = std::chrono::steady_clock::now();
    FFT(X2, n);
    end = std::chrono::steady_clock::now();
    std::cout << "FFT: "
              << std::chrono::duration_cast<std::chrono::microseconds>(end -
start).count()
              << " µs" << std::endl;
}
```

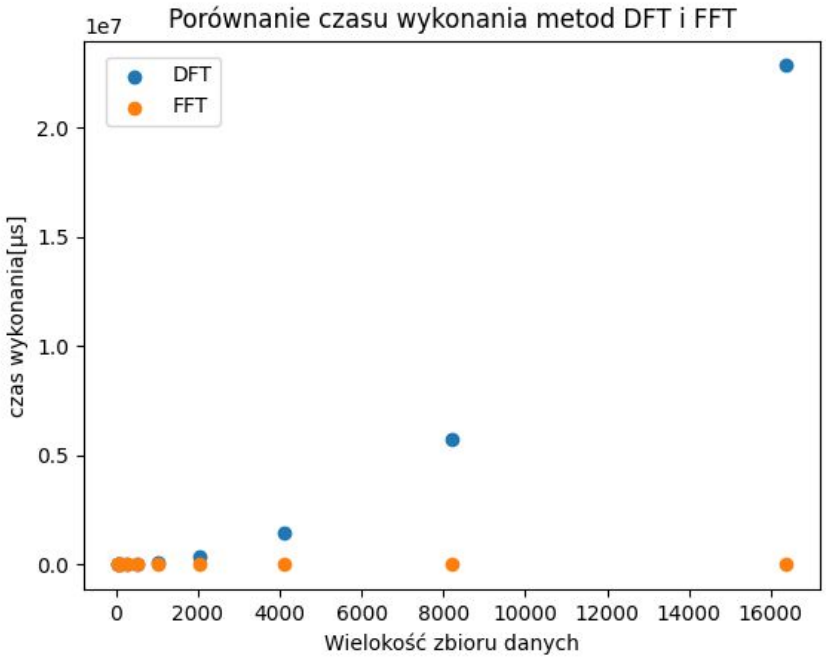
```
wielkość zbioru: 32
DFT: 160 µs
FFT: 19 µs
wielkość zbioru: 64
DFT: 660 µs
FFT: 75 µs
```

```
wielkość zbioru: 128
DFT: 1691 µs
FFT: 94 µs
wielkość zbioru: 256
DFT: 6132 µs
FFT: 201 µs
```

```
wielkość zbioru: 512
DFT: 23437 µs
FFT: 431 µs
wielkość zbioru: 1024
DFT: 93208 µs
FFT: 917 µs
```

```
wielkość zbioru: 2048
DFT: 361608 µs
FFT: 2031 µs
wielkość zbioru: 4096
DFT: 1437868 µs
FFT: 5551 µs
```

```
wielkość zbioru: 8192
DFT: 5756818 µs
FFT: 9035 µs
wielkość zbioru: 16384
DFT: 22833367 µs
FFT: 18986 µs
```

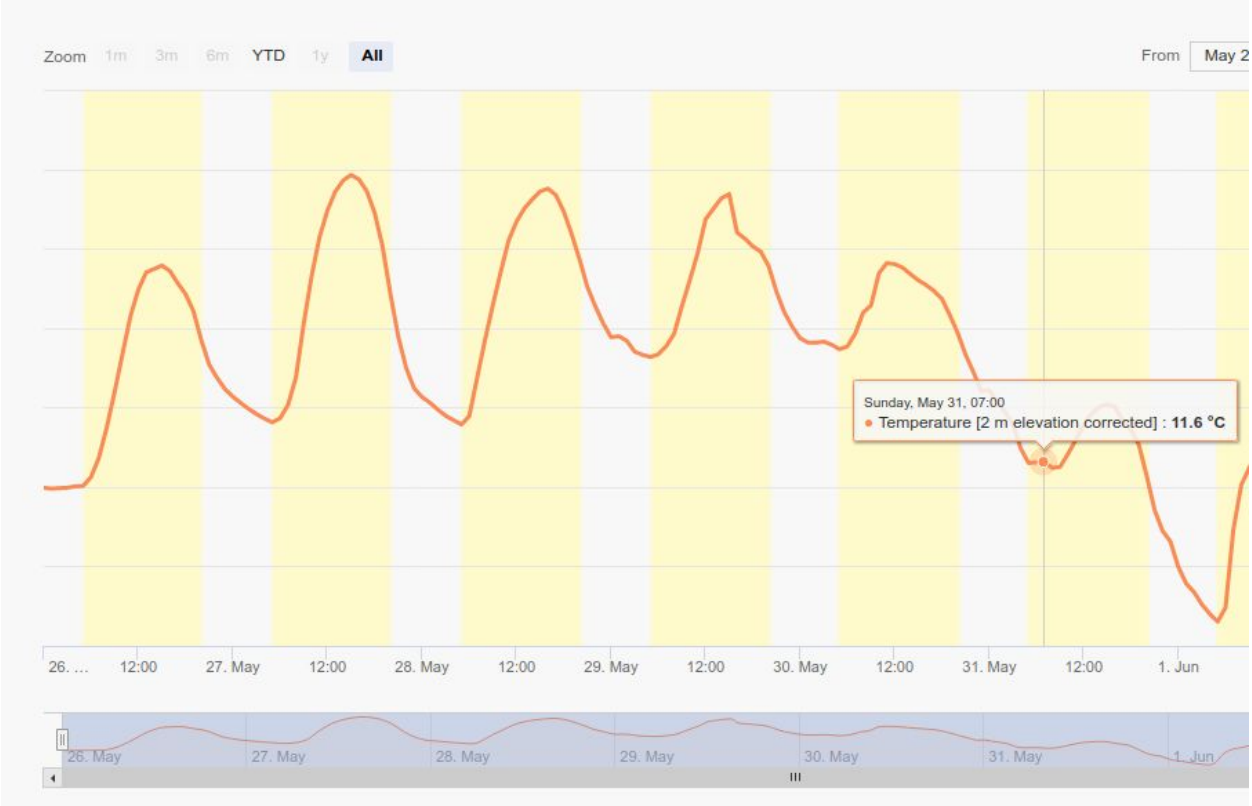


Wykonałem wykres który porównuje czasy wykonania algorytmów dla tych samych danych o różnej wielkości. Z wykresu wynika, że czas wykonania algorytmu DFT rośnie wielomianowo (połączenie kropek przypominałoby parabolę), dlatego uważam, że wynik ten jest zgodny z oczekiwaniami.
Algorytm FFT jest znacznie szybszy. Żółte kropki reprezentujące czas wykonania FFT na wykresie tworzą linię prostą. W rzeczywistości czas wykonania w zależności od ilości danych jest nieliniową relacją(szacowana złożoność to O(NlogN)) jednak dla małej ilości danych może to być niezauważalne.

Główna przewaga algorytmu FFT nad DFT to czas działania. Złożoność pojedynczej iteracji funkcji realizującej FFT jest dwa razy mniejsza od DFT. Jest tak dlatego, że obliczając jedną wartość możemy ją wykorzystać do wyznaczenia dwóch wyrazów X[k]. Takie postępowanie często nazywa się operacjami motylkowymi. Jednak diametralna różnica w czasie wykonania zależy przede wszystkim od sposobu wyznaczania kolejności wyliczania wyrazów. Stosując metodę dziel i zwyciężaj ulepszamy złożoność poprzez rekurencyjne rozbicie problemu na wiele innych, mniejszych problemów.
Każde wywołanie rekurencyjne funkcji FFT zmniejsza stałą N o połowę - a właśnie od tej stałej zależy złożoność pojedynczej iteracji.

4. **Analiza średniej temperatury w Polsce od 26.05.20 0:00 do 31.05.20 7:00**

Na stronie meteoblue.com znalazłem zestawienie średniej temperatury w Polsce. Pomiary są dokonywane co godzinę. Pobrałem 128 pomiarów od 26.05.20 0:00 do 31.05.20 7:00:



Następnie zaimplementowałem funkcję, która wykonuje algorytm FFT na podanych danych. Funkcja wypisuje też wartości możliwych częstotliwości.

```
void tempData(){
   std::complex<double> X[128] =
           {9.946629, 9.896628, 9.926628, 9.956628, 10.036628, 10.066628,
            10.626628, 11.846628, 13.766628, 15.996628,
            18.396627, 20.746628, 22.416628, 23.506628, 23.716629,
            23.936628, 23.596628, 22.826628, 22.13663, 21.056627,
            19.226627, 17.696629, 16.866629, 16.156628, 15.686628,
            15.286628, 14.916628, 14.586628, 14.306628, 14.066628,
            14.326628, 15.166628, 16.88663, 20.296629, 23.286629,
25.716629,
            27.396627, 28.586628, 29.286629, 29.63663,
            29.356628, 28.616629, 27.246628, 25.13663, 22.13663, 19.456629,
            17.486628, 16.216629, 15.666628, 15.306628,
            14.876628, 14.486629, 14.206628, 13.936628, 14.446629,
            16.856628, 19.246628, 21.446629, 23.536629, 25.506628,
            26.706629, 27.526628, 28.096628, 28.596628, 28.776628,
            28.366629, 27.336628, 25.916628, 24.326628, 22.586628,
```

```
            21.376629, 20.306627, 19.426628, 19.486628, 19.196629,
            18.516628, 18.316628, 18.176628, 18.346628, 18.856628,
            19.63663, 21.38663, 23.016628, 24.716629, 26.846628, 27.516628,
            28.166628, 28.436628, 26.006628, 25.626629,
            25.13663, 24.806627, 23.906628, 22.286629, 20.986628,
20.086628,
            19.356628, 19.096628, 19.076628, 19.146627,
            18.956629, 18.666628, 18.846628, 19.656628, 20.976627,
            21.426628, 23.446629, 24.086628, 24.026628, 23.806627,
            23.396627, 23.006628, 22.706629, 22.336628, 21.846628,
            20.816628, 19.696629, 18.326628, 17.226627, 16.026628,
            16.106628, 15.336628, 14.736629, 13.986629, 12.426628,
            11.506628, 11.556628, 11.586628};

   int n = 128;
   double seconds = 3600.0;
   double sampleRate = n / seconds;
   double freqResolution = sampleRate / n;
   std::complex<double> x[n];

   FFT(X, n);

//    for (int i = 0; i < n; i++) {
//        std::cout << abs(X[i]) << std::endl;
//    }
//
   for (int i = 0; i < n; i++) {
       std::cout << i * freqResolution << std::endl;
   }

}
```

Sprawdziłem poprawność wyników korzystając z funkcji fft biblioteki numpy. Wyniki obu algorytmów są zgodne.

```
x = []
with open('temp.txt', 'r') as f:
      for item in f:
      x.append(float(item));

myfft = []
with open('fft.txt', 'r') as ft:
      for item in ft:
      myfft.append(float(item));
```

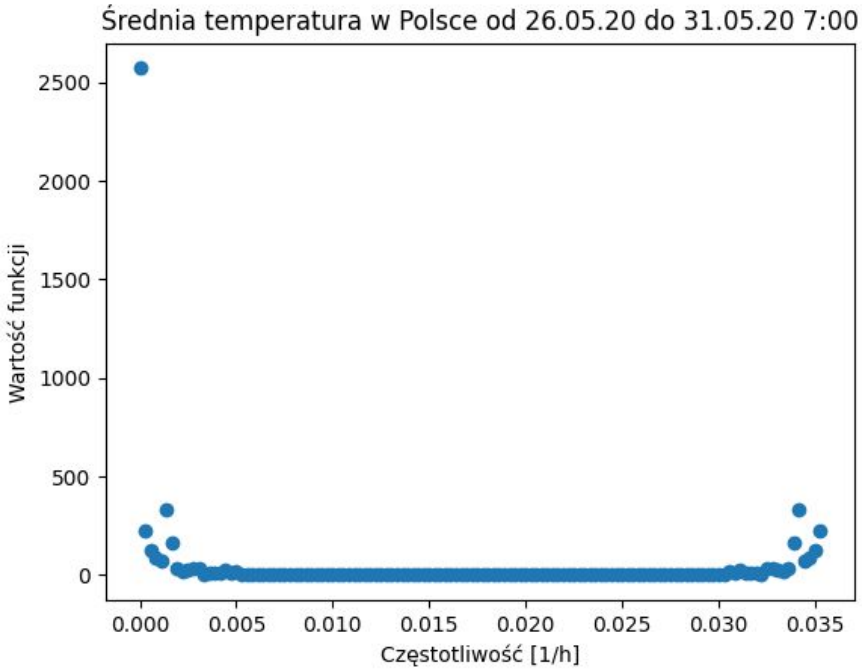
```
result = np.fft.fft(x)

result_abs = []
for item in result:
    result_abs.append(abs(item))

print(np.allclose(myfft, result_abs))
```

```
wiktor@wiktor-ThinkPad-T470s:~/studia/mownit/laborki/lab8$ python3 dft.py
True
```

Następnie utworzyłem wykres średniej temperatury w Polsce w dziedzinie częstotliwości:



Dominująca wartość występuje w x = 0. Pozostałe wartości rozkładają się w bardzo podobny sposób. Wartości na krańcach dziedziny są dominujące. Na wykresie nie występuje jednak wartość dominująca różna od 0, dlatego uważam, że analiza Fouriera dla tych danych ma średni sens.