Julia Plewa
MOwNiT
Laboratorium 4

1. **Napisz klasę realizującą DFT. Proszę opisać kolejno realizowane na danych operacje posługując się teorią.**

Kod:

```
std::vector <std::complex<T>> calculate() override
{
    // prepare result vector
    std::vector <std::complex<T>> transformedDataset(rawDataset.size(),
                                                     std::complex<T>(0, 0));
    // compute result vector elements one by one
    for (int k = 0; k < rawDataset.size(); k++)
    {
        // each result vector element is a sum
        for (int n = 0; n < rawDataset.size(); n++)
        {
            // using Euler's formula
            std::complex <T> tmp(cos(2 * M_PI * k * n / (T) rawDataset.size()),
                                 -(sin(2 * M_PI * k * n / (T) rawDataset.size())));
            transformedDataset[k] += rawDataset[n] * tmp;
        }
    }
    return transformedDataset;
}
```

Opis:

Dyskretna transformata Fouriera (DFT) polega na przekształceniu ciągu N liczb zespolonych $x_0, x_1, \dots, x_{N-1}$ w nowy ciąg $X_0, X_1, \dots, X_{N-1}$, gdzie $\forall\ 0 \le k \le N-1$:

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

W powyższym kodzie w zewnętrznej pętli po kolei obliczane są kolejne wartości X_k, a dla każdej z nich w wewnętrznej pętli obliczana jest suma iloczynów.

Przykład działania:

wejście	**wyjście**
`{1, 2 - 1i, -1i, -1 + 2i}`	`{2, -2 - 2i, 8.88178e-16 - 2i, 4 + 4i}`

Warto zauważyć, że w związku z brakiem dokładności operacji zmiennoprzecinkowych zamiast wartości 0 w wynikowym wektorze pojawia się tak naprawdę wartość $8.88178e-16 \approx 0$. Wynik jest zgodny z przykładem z Wikipedii.

2. **Korzystając z implementacji stworzonej w zadaniu 1 napisz klasę realizującą FFT (korzystając z algorytmu Cooleya-Tukeya). Implementacje poprzyj stosownym materiałem teoretycznym.**

Kod:

```
// recursive inner function
void cooleyTurkey(std::vector <std::complex<T>> &transformedDataset, int index,
                  int begin, int N, int step)
{
    if (N <= CUT_OFF_SIZE)     // base case DFT
    {
        std::vector <std::complex<T>> sliceCopy(N);    // copy data into a new vector
        for (int i = 0; i < N; i++)
        {
            sliceCopy[i] = rawDataset[begin + i * step];
        }
       // calculate the DFT
       std::unique_ptr <fourier::DFT<double>> dft =
           td::make_unique<fourier::DFT<double>>(fourier::DFT<double>(sliceCopy));
       std::vector <std::complex<double>> dft_result = dft -> calculate();
       for (int i = 0; i < N; i++)    // copy results back into the result vector
       {
           transformedDataset[index + i] = dft_result[i];
       }
    }
    else
    {
        // calculate left- and right-half FFT
        // each sub-vector has half less elements and includes every other element
        // from the currect vector (with an offset in case of the right-half vector)
        cooleyTurkey(transformedDataset, index, begin, N / 2, 2 * step);
        cooleyTurkey(transformedDataset, index + N / 2, begin + step, N / 2, 2 * step);
        // use symmetry to calculate final results
        for (int k = index; k < index + (N / 2); k++)
        {
            std::complex<T> t = transformedDataset[k];
            std::complex<T> tmp(cos(2 * M_PI * k / N), -sin(2 * M_PI * k / N));
            transformedDataset[k]          = t + transformedDataset[k + N / 2] * tmp;
            transformedDataset[k + N / 2] = t - transformedDataset[k + N / 2] * tmp;
        }
    }
}
```

```
std::vector <std::complex<T>> calculate() override
{
    if (ceil(log2(rawDataset.size())) != floor(log2(rawDataset.size())))
    {
        throw exceptions::IncorrectVectorSizeException();
    }
    std::vector <std::complex<T>> transformedDataset(rawDataset.size());
    // out-of-place radix-2 DIT (decimation-in-time) FFT
    cooleyTurkey(transformedDataset, 0, 0, rawDataset.size(), 1);
    return transformedDataset;
}
```

Opis:

W ogólnej postaci algorytm Cooleya-Tukeya polega na wyrażeniu dyskretnej transformaty Fouriera wektora o dowolnie złożonej wielkości $N = N_1 \cdot N_2$ w członach mniejszych DFT wielkości N_1 i N_2. Najpowszechniejszą wersją algorytmu jest *radix-2 DIT* (decimation-in-time), w której wektor jest zawsze rozbijany na dwie mniejsze części i właśnie na tę wersję zaimplementowano w trakcie realizacji zadania.

Istotną cechą algorytmu DFT jest jego symetria, tj. dla wektora o długości N $\forall\ 0 \le k \le N-1$ zachodzi:

$$X_{N+k} = \sum_{n=0}^{N-1} x_n \cdot e^{-i\,2\pi\,(N+k)\,n\,/\,N} = \sum_{n=0}^{N-1} x_n \cdot e^{-i\,2\pi\,n} \cdot e^{-i\,2\pi\,k\,n\,/\,N} =$$
$$= \sum_{n=0}^{N-1} x_n \cdot e^{-i\,2\pi\,k\,n\,/\,N} = X_k\,.$$

Ponadto z definicji DFT można wykazać, że:

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

gdzie A_k i B_k to wyniki DFT dla kolejno wartości wektora wejściowego z parzystych oraz nieparzystych pozycji.

Na podstawie pokazanej wcześniej symetrii łatwo jest wyprowadzić również następującą zależność:

$$X_{k+N/2} = A_k - e^{-i\,2\pi\,k\,/\,N} \cdot B_k \,.$$

Wynika z tego, że każda para obliczonych rekurencyjnie wektorów może być wykorzystana podwójnie.

Przykład działania:

wejście	**wyjście**
`{1, 2 - 1i, -1i, -1 + 2i}`	`{2, -2 - 2i, 8.88178e-16 - 2i, 4 + 4i}`

Wynik jest zgodny z przykładem z Wikipedii.

3. **Dokonaj pomiarów czasu wykonywania obu transformat dla danych o różnym rozmiarze. Pomiaru dokonaj dla kilku wielkości danych (min. 10). Na tej podstawie dokonaj analizy czasowej złożoności obliczeniowej obu algorytmów i porównaj je ze sobą. Sprawdź czy zgadzają się z rządami teoretycznymi i opisz różnicę w algorytmie, która generuje różnicę w złożoności.**

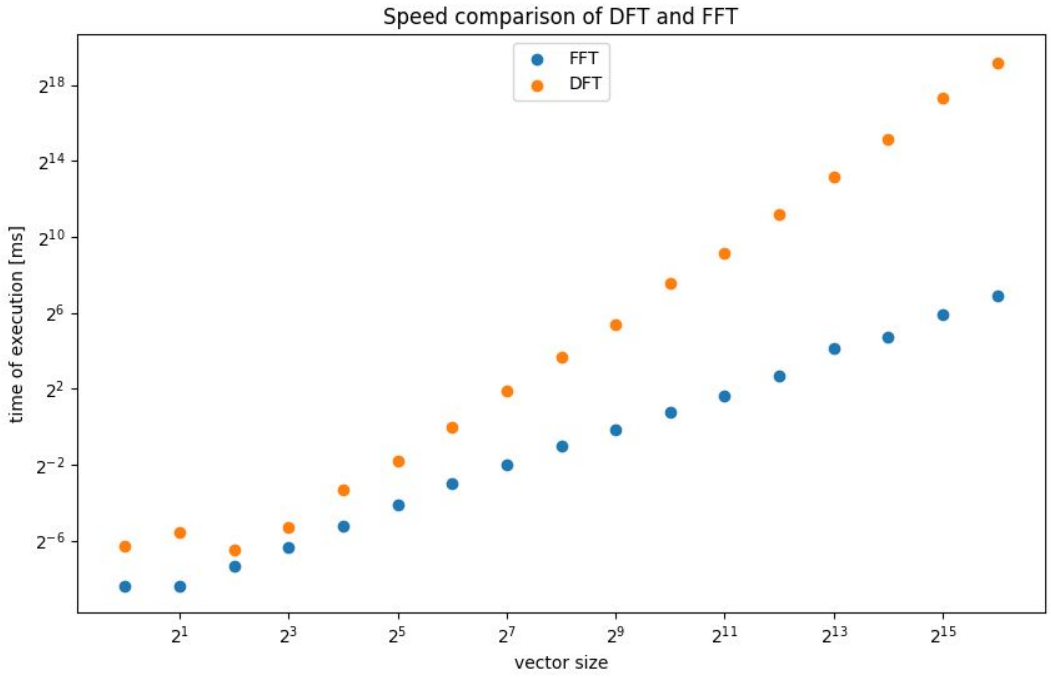
Wyniki:

rozmiar wektora	**czas wykoniania** $[ms]$	
	DFT	**FFT**
1	0.013	0.003
2	0.021	0.003
4	0.011	0.006
8	0.025	0.012
16	0.098	0.027
32	0.291	0.058
64	0.996	0.124

128	3.716	0.248
256	12.532	0.492
512	42.414	0.875
1024	187.661	1.712
2048	562.695	3.067
4096	2280.47	6.39
8192	9226.73	17.73
16384	35473.6	26.491
32768	165294	58.779
65536	595048	116.3

Wykres:

Dla zwiększenia czytelności obie osie zostały przedstawione w skali logarytmicznej.



Porównanie:

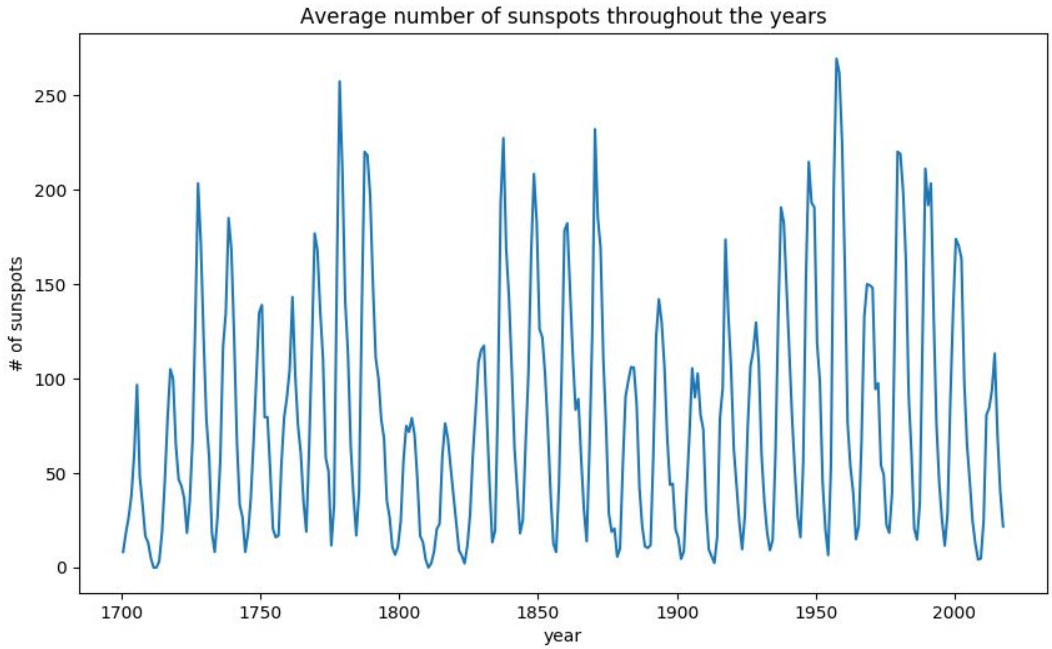
Algorytm FFT opiera się na DFT, jednak wprowadza dodatkowe optymalizacje. Jego wyprowadzenie zostało przedstawione w poprzednim podpunkcie. Dzięki rekurencyjnemu podziałowi wektora na mniejsze części oraz wykorzystaniu symetrii obecnej w DFT, FFT pozwala na poprawę złożoności z $\Theta(n^2)$ na $\Theta(n\ log(n))$.

Poprawa ta jest widoczna w uzyskanych wynikach. W przypadku DFT podwojenie ilości danych prowadzi w przybliżeniu do czterokrotnego zwiększenia czasu działania. Pomiary czasów działania zaimplementowanego FFT sugerują, że jego złożoność faktycznie jest zbliżona do teoretycznego $\Theta(n\ log(n))$, a być może nawet delikatnie lepsza. W wybranej implementacji FFT nie jest dokonywane w miejscu, zatem nie ma konieczności permutowania kolejności elementów wektora.

4. Przetestuj implementację z zadania 2. do wykonania analizy szeregu czasowego:

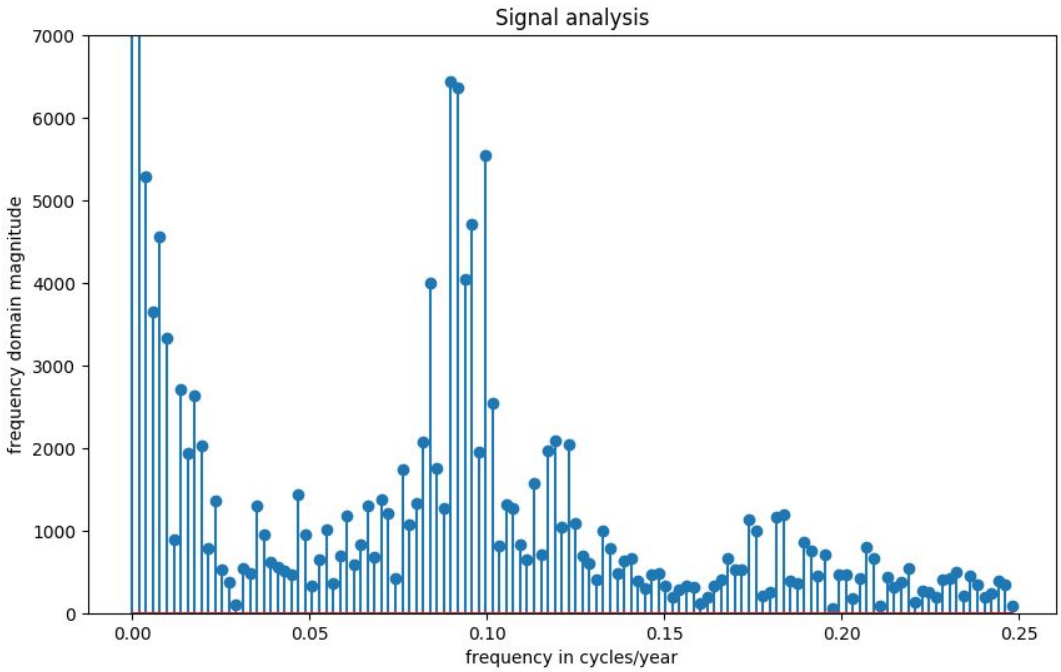
a. Znajdź dane przedstawiające jakiś szereg czasowy.

Do analizy wybrano zbiór danych *Yearly mean total sunspot number* udostępniony przez Royal Observatory of Belgium przedstawiający średnią roczną liczbę plam słonecznych.



b. **Załaduj je do programu. Zobacz, czy wykonanie analizy Fouriera na tych danych ma sens - być może trzeba pogrupować je w równe odstępy.**

c. **Narysuj wykres w dziedzinie częstotliwości i postaraj się opisać zależności jakie możemy na nim dostrzec.**



Z wykresu wynika, że maksymalna aktywność plam słonecznych ma miejsce mniej więcej 1 / 0.09 razy do roku, czyli w przybliżeniu co 11 lat.

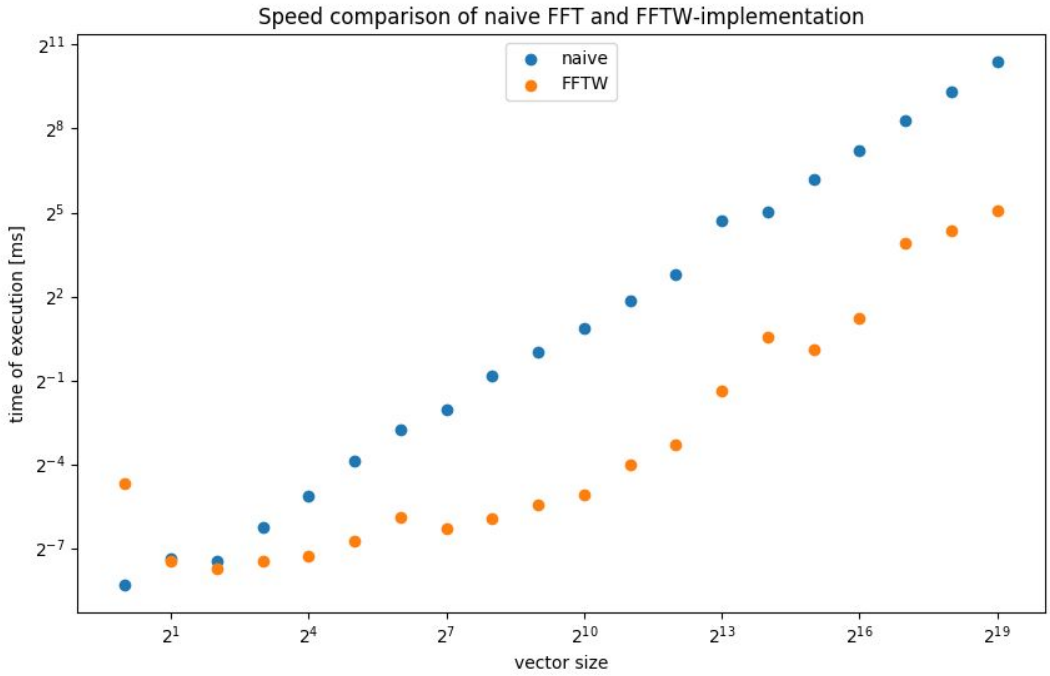
5. **(*) Wykonaj ponownie analizę FFT wykorzystując w tym celu bibliotekę KFR / FFTW / bibliotekę Aquila C++. Wykorzystaj dane z których korzystałeś w poprzednich zadaniach (analogiczne rozmiary w kolejnych iteracjach). Zestaw na wykresie wyniki pomiarów wybranej biblioteki oraz swojej własnej implementacji. Przedstaw możliwy sposób jakiejkolwiek optymalizacji swojego algorytmu, aby zbliżyć się do testowanych.**

Wykorzystana biblioteka: FFTW

Wyniki:

rozmiar wektora	**czas wykoniania** [*ms*]	
	własne FFT	**FFT z FFTW**
1	0.0031	0.0387
2	0.006	0.0056
4	0.0057	0.0047
8	0.0133	0.0056
16	0.0285	0.0064
32	0.0675	0.0093
64	0.1453	0.0169
128	0.2438	0.0129
256	0.5602	0.0164
512	1.0124	0.0226
1024	1.8371	0.0296
2048	3.5461	0.0612
4096	6.9025	0.1027
8192	26.5546	0.3846
16384	32.762	1.4657
32768	73.547	1.066
65536	149.261	2.3441
131072	314.171	14.7457
262144	641.955	20.6442
524288	1364.51	33.6974

Wykres:



Możliwość optymalizacji:

Implementacja FFT z biblioteki FFTW prezentuje się dużo bardziej wydajnie niż implementacja stworzona na potrzeby zadania 2 (choć nie dla wektora o rozmiarze 1-2). W celu zwiększenie wydajności FFT często używa się przypadku bazowego większego niż $N = 1$ do amortyzowania narzutu rekursji - jest to funkcjonalność uwzględniona w stworzonej implementacji. Inną możliwą optymalizacją jest obliczenie współczynnika fazowego z wyprzedzeniem.

Biblioteka FFTW wybiera algorytm FFT na podstawie rozmiaru zadanego wektora, a następnie wykorzystuje mechanizmy takie jak *loop unrolling* czyli odwijanie pętli (generowane "dynamicznie" w trakcie kompilacji) oraz optymalizuje dostęp do pamięci cache. Warto również zauważyć, że stworzona implementacja obsługuje jedynie przypadek *radix2*, podczas gdy warianty *radix4* i *radix8* gwarantują często dużo lepsze wyniki.