MOwNiT2 Lab4

Wszystkie zestawy punktów znajdują się w takiej strukturze:

```
typedef std::vector<std::complex<double>> FourierData;
```

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

```
class DiscreteFourier
    {
    public:
       // construction methods
       DiscreteFourier(const FourierData data) : values(data) {}
        // methods
        FourierData dft() {
           int N = values.size();

           FourierData results;

           for (int k = 0; k < N; k++) {  // For each output element
               std::complex<double> sum = 0;
               std::complex<double> tmp;
               for (int n = 0; n < N; n++) {  // For each input element
                   double angle = 2 * M_PI * n * k / N;
                   tmp.real(values[n].real() * cos(angle) +
values[n].imag() * sin(angle));
                   tmp.imag(-values[n].real() * sin(angle) +
values[n].imag() * cos(angle));
                   sum = sum + tmp;
               }
               results.push_back(sum);
           }
           return results;
       }
        // private members
    private:
        FourierData values;
    };
```

DFT wyraża się wzorem:

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

Wewnętrzna pętla programu dokonuje sumowania.
Zewnętrzna pętla wywołuje sumowanie i po jego wykonaniu dodaje wyliczoną wartość do zbioru wynikowego.

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

```
class FastFourier
    {
    public:
       // construction methods
       FastFourier(const FourierData data) : values(data) {}
        // methods
        FourierData fft() {
           if(values.size()%2 != 0){
               std::string txt = "Size must be a power of two.";
               throw std::invalid_argument(txt);
           }
           FourierData result(values.size());
           computeFFT(result, 0, 0, values.size(), 1);
           return result;
       }
   private:
       void computeFFT(FourierData &data, int index, int from, int size, int
stride){
           if(size == 1){
               data[index] = values[from];
               // one point DFT equals given point, no need to invoke dft()
           }
           else{
               computeFFT(data, index, from, size/2, 2*stride);
                   // even-indexed elements
               computeFFT(data, index+size/2, from+stride, size/2,
2*stride);         // odd-indexed elements

               for(int k=index; k<index+size/2; k++){
                   std::complex<double> even = data[k];
                   std::complex<double> odd = data[k+size/2];
                   std::complex<double> tf =
exp(std::complex<double>(0,-2.*M_PI*k/size));
                   data[k] = even + tf*odd;
                   data[k+size/2] = even - tf*odd;
               }
           }
       }
        // private members
    private:
       FourierData values;
    };
```

- Idea FFT polega na podziale obliczania DFT na dwie części:

$$X_k = \sum_{n=0}^{N-1} x_n * e^{-i2\pi kn/N} = \sum_{m=0}^{N/2-1} x_{2m} * e^{-i2\pi k(2m)/N} + \sum_{m=0}^{N/2-1} x_{2m+1} * e^{-i2\pi k(2m+1)/N}$$

  Teraz można wykonać DFT osobno dla elementów o parzystych i nieparzystych indeksach.
- Tę operację można powtarzać aż do osiągnięcia preferowanego rozmiaru lub aż dotrzemy do wywoływania DFT dla pojedynczych elementów.
- Ze względu na opisany podział rozmiar danych powinien być potęgą dwójki (aczkolwiek jest to tylko uproszczenie).
- W swoim rozwiązaniu dokonuje podziału danych aż do rozmiaru równego 1. Korzystając z faktu, że dla pojedynczego punktu DFT(x) = x metoda obliczająca DFT nie musi być wywoływana wcale.

Przykładowe dane z wikipedii:

```
const FourierData test4 =
   {
       std::complex<double>(1,0),
       std::complex<double>(2,-1),
       std::complex<double>(0,-1),
       std::complex<double>(-1,2)
   };
```

Powinny dać wynik:

```
(2,0)
(-2,-2)
(0,-2)
(4,4)
```

Wynik wywołania obu algorytmów:

DFT	FFT
(2,0) (-2,-2) (8.88178e-016,-2) (4,4)	(2,-2.22045e-016) (-2,-2) (4.44089e-016,-2) (4,4)

Minimalne różnice przy wartościach 0 mogą wynikać ze skończonej dokładności liczb zmiennoprzecinkowych.

Zadanie 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.

Czas mierzyłem za pomocą:

```
std::chrono::high_resolution_clock::time_point start =
                                std::chrono::high_resolution_clock::now();
   // wywołanie FFT lub DFT
std::chrono::high_resolution_clock::time_point end =
                                std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::microseconds>
                        (end - start).count()/1000.0;
```

DFT	FFT
512 wartości	
61.488ms	0.504ms
1024 wartości	
251.446ms	0.992ms
2048 wartości	
1001.92ms	2.021ms
4096 wartości	
3924.35ms	4.934ms
Czasowa złożoność obliczeniowa	
$O(N^2)$	$O(Nlog_2N)$

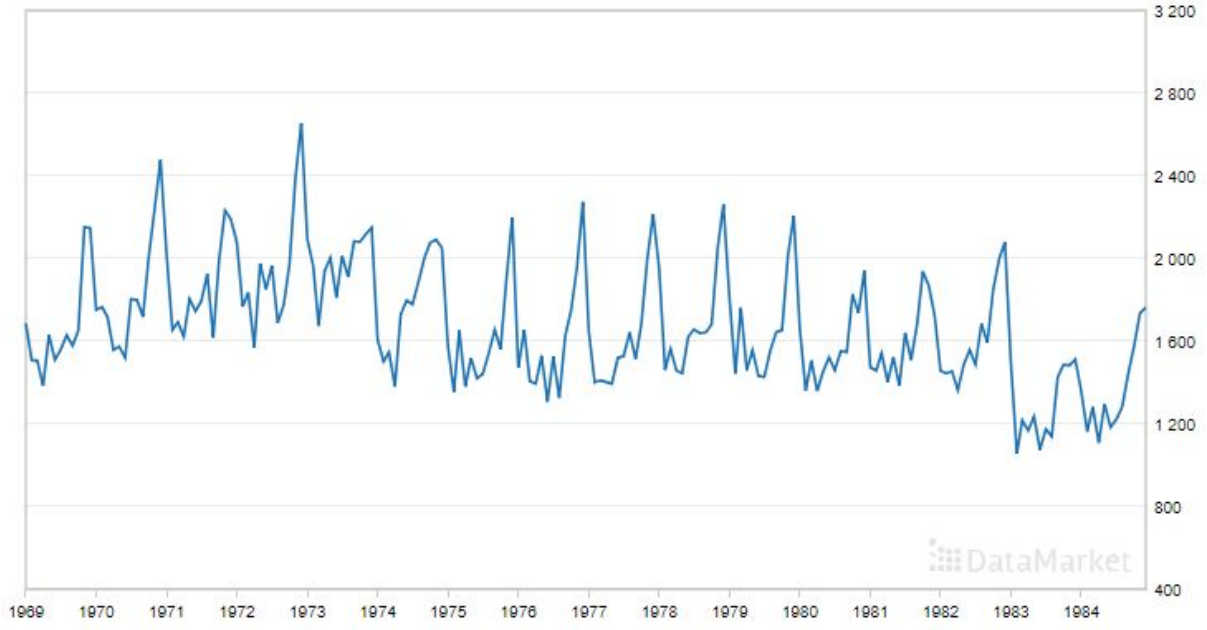
Przy podwojeniu rozmiaru próbki algorytm DFT pracuje ~4 razy dłużej,
a algorytm FFT ~2.4 raza dłużej.
Pokrywa się to ze złożonościami odpowiednio: N^2 i $Nlog_2N$.

Pierwszy algorytm zawiera dwie zagnieżdżone pętle przechodzące przez cały rozmiar N, a więc: $N * N = N^2$.

Drugi algorytm dokonuje podziału próbki N na dwie równe aż otrzyma pojedynczą wartość i zawiera jedną pętlę, toteż: $log_2N * N = Nlog_2N$.

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

Dane przedstawiają ilość poważnie rannych / zmarłych w wypadkach samochodowych w Wielkiej Brytanii w każdym miesiącu na przestrzeni 15lat (1969-1984).



Widać tutaj wysokie skoki w grudniu każdego roku.

Skorzystam z pierwszych 128 wartości (ponad 10lat).

```
FourierData time_series;
std::ifstream infile("time_series_data.csv");
std::string line;
std::getline(infile, line); // skip first line
for(int i=0;i<128;i++){
    std::getline(infile, line);
    std::string value = line.substr(line.find(','));
    value = value.substr(1);
    value = value.substr(0,value.find(','));
    double d = std::stod(value,NULL);
    time_series.push_back(std::complex<double>(d,0));
}
```

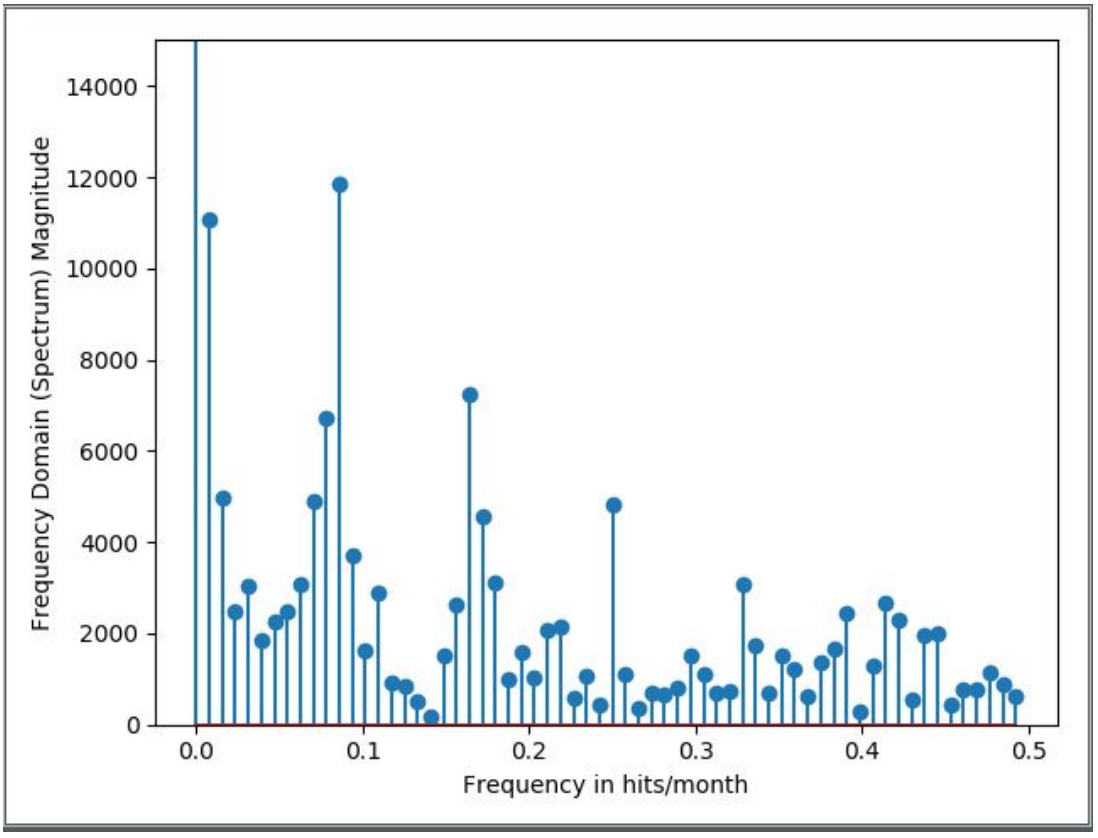
Po wykonaniu FFT na danych zapisałem rezultat do pliku csv.

Kod wykresu w dziedzinie częstotliwości:

```
import csv
import math as m
import matplotlib.pyplot as plt
import scipy.fftpack as fft

data = []
with open('computed_data.csv') as csvfile:
   reader = csv.reader(csvfile)
   for row in reader:
       re = float(row[0])
       im = float(row[1])
       data.append(m.sqrt(re*re+im*im))

f_s = 1
freqs = fft.fftfreq(len(data)) * f_s
fig, ax = plt.subplots()
ax.stem(freqs[:64], data[:64])
ax.set_xlabel('Frequency in hits/month')
ax.set_ylabel('Frequency Domain (Spectrum) Magnitude')
ax.set_ylim(-1, 15000)
plt.show()
```



Dominującą częstotliwością jest ~0.85/mies co oznacza że dane mają wzorce powtarzające się co ~12 miesięcy (czego można było się spodziewać).