Dyskretna transformata Fouriera

Michał Kazanecki, Krzysztof Kubień, Mikołaj Wielebnowski

30 czerwca 2020

1 Wstęp

Dyskretna transformata Fouriera (DFT) jest reprezentacją wejściowego ciągu w dziedzinie częstotliwości. Jest ona jedną z najważniejszych dyskretnych transformat, mającą wiele zastosowań praktycznych. DFT znajduje zastosowanie między innymi w analizie i przetwarzaniu sygnałów, przetwarzaniu obrazów, kompresji danych, rozwiązywaniu równań różniczkowych cząstkowych, mnożeniu dużych liczb całkowitych czy mnożeniu wielomianów. Dyskretna forma transformaty umożliwia jej implementację na komputerze w postaci algorytmów numerycznych. Wiele zastosowań DFT wynika z dostępności szybkiego algorytmu obliczającego DFT zwanego szybką transformatą Fouriera (FFT)[2]. W poniższym artykule zostanie przedstawiony algorytm FFT i jego implementacja w języku C++ i Python.

2 DFT i FFT

2.1 Pierwiastki zespolone z jedności

Znim przejdziemy do wzoru i algorytmu na dyskretną transformatę Fouriera zdefiniujemy kilka użytecznych pojęć. Zespolony pierwiastek z jedności to taka liczba zespolona ω, że

$$\omega^n = 1 \tag{1}$$

Głównym pierwiastkiem zespolonym z jedności nazywamy wartość

$$\omega_n = e^{2\pi i/n} \tag{2}$$

Pierwiastki zespolone z jedności mają następujące własności:

$$\omega_{dn}^{dk} = \omega_n^k \tag{3}$$

$$\omega_n^{n/2} = -1 \tag{4}$$

$$(\omega_n^{k+n/2})^2 = (\omega_n^k)^2 \tag{5}$$

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0 \tag{6}$$

Z równania (5) wynika, że dla parzystego $n > 0$ zbiór kwadratów n zespolonych pierwiastków z 1 stopnia n jest zboirem n/2 zesplonych pierwiastków z 1 stopnia n/2

2.2 Transformacja prosta i odwrotna

Dyskretna transformacja Fouriera dana jest wzorem

$$y_k = \sum_{j=0}^{n-1} a_j \omega_n^{-kj}, \quad 0 \leqslant k \leqslant n-1 \tag{7}$$

Równanie to można przedstawić w postaci macierzowej $y = V_n a$, gdzie macierz V_n to macierz Vandermonde'a zawierająca odpowiednie potęgi ω_n:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \dots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \dots & \omega_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \dots & \omega_n^{-(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} \tag{8}$$

Aby otrzymać transforację odwrotną należy pomnożyć powyższe wyrażenie przez V_n^{-1}. Nie trudno sprawdzić, że prowadzi to do wzoru

$$a_j = \frac{1}{n} \sum_{j=0}^{n-1} y_k \omega_n^{kj} \tag{9}$$

Jak widać, obliczenie transformacji z definicji sprowadza się do mnożenia macierzy przez wektor. Jest to algorytm o złożoności czasowej $O(n^2)$, gdyż dla każdego z n współczynników wykonujemy n operacji mnożeia zespolonego. Z powyższych wzorów widać też, że obliczenie transformacji odwrotnej odbywa się podobnie do obliczenia transformacji prostej. Bardzo podobny będzie też algorytm pozwalający obliczyć obie transformacje w szybki sposób. Z tego powodu w dalszej części skupimy się tylko na transformacji prostej. Warto zauważyć też, że definicja prostej i odwrotnej transformacji zależy od przyjętej konwencji (por. [1, 2])

2.3 Szybka transforacja Fouriera

Rozwiązanie, które zostanie przedstawione poniżej opiera się na metodzie "dziel i zwyciężaj". Algorytm ten został przedstawiony przez Jamesa Cooleya i Johna Tuckeya w pracy z 1965 roku, jednak był znany już wcześniej (został wynaleziony przez Gaussa ok. 1805 roku)[3]. Korzysta on ze szczególnych własności pierwiastków zesplonych z jedności i pozwala obliczyć DFT w czasie $O(n \, log \, n)$. Przedstawiony poniżej opis oparty został na opisie zawartym w [1].

Bez utraty ogólności możemy założyć, że ilość współczynników n jest potęgą dwójki (dany ciąg można uzupełnić zerami). Obliczenie współczynnika y_k jest niczym innym jak obliczeniem wartości wielomianu $A(x) = \sum_{j=0}^{n-1} a_j x^j$ mającego ograniczenie stopnia n, w punkcie ω_n^{-k}. Algorytm opiera się na podzieleniu współczynników na parzyste i nieparzyste i zdefiniowaniu nowych wielomianów o ograniczeniu stopnia n/2.

$$A^{(0)}(x) = a_0 + a_2 x + a_4 x^2 + ... a_{n-2} x^{n/2-1} \tag{10}$$

$$A^{(1)}(x) = a_1 + a_3 x + a_5 x^2 + ... a_{n-1} x^{n/2-1} \tag{11}$$

Wielomian $A(x)$ dany jest wtedy wzorem:

$$A(x) = A^{(0)}(x^2) + xA^{(1)}(x^2) \tag{12}$$

Sprowadzamy więc problem obliczenia wartości wielomianu $A(x)$ w punktach ω_n^0, ω_n^{-1}, ω_n^{-2}, ..., $\omega_n^{-(n-1)}$ do ewaluacji wielomianów $A^{(0)}(x)$ oraz $A^{(1)}(x)$ w punktach $(\omega_n^0)^2$, $(\omega_n^{-1})^2$, $(\omega_n^{-2})^2$, ..., $(\omega_n^{-(n-1)})^2$ i połączenia wyników według wzoru (12). Jak wynika ze wzoru (5) ilość punktów, w których dokonujemy ewaluacji zmniejsza się o połowę, ponieważ po podniesieniu pierwiastków z jedności do kwadratu każdy z punktów występuje dwukrotnie. Problem obliczenia n-elementowej DFT zamieniamy więc na dwa problemy tej samej postaci jednak dwukrotnie mniejszego rozmiaru. Dzieląc kolejne problemy na pół otrzymujemy rekurencyjny algorytm na obliczenie dyskretnej transformacji Fouriera. Warunek stopu dla rekurencji to obliczenie DFT z ciągu jednoelementowego. Jest ona wtedy równa temu elementowi:

$$y = a_0 \omega_1^0 = a_0 \cdot 1 = a_0 \tag{13}$$

Wprowadzając oznaczenia dla $k \leqslant n/2 - 1$:

$$y_k = A(\omega_n^{-k}) \tag{14}$$

$$y_k^{(0)} = A^{(0)}(\omega_{n/2}^{-k}) = A^{(0)}(\omega_n^{-2k}) \tag{15}$$

$$y_k^{(1)} = A^{(1)}(\omega_{n/2}^{-k}) = A^{(1)}(\omega_n^{-2k}) \tag{16}$$

Możemy obliczyć wartości wektora y:

$$y_k = A(\omega_n^{-k}) = A^{(0)}(\omega_n^{-2k}) + \omega_n^{-k} A^{(1)}(\omega_n^{-2k}) = y_k^{(0)} + \omega_n^{-k} y_k^{(1)} \tag{17}$$

$$y_{k+n/2} = A(\omega_n^{-k}) = A^{(0)}(\omega_n^{-2k}) + \omega_n^{-k+n/2} A^{(1)}(\omega_n^{-2k}) = y_k^{(0)} - \omega_n^{-k} y_k^{(1)} \tag{18}$$

Implementacja algorytmu w językach Python oraz C++ znajduje się odpowiednio w dodatku A oraz B. Kod napisany był w oparciu o podobny, podany w [4].

Aby oszacować złożoność czasową algorytmu warto zauważyć, że oprócz wywołań rekurencyjnych wykonanie procedury zajmuje czas $O(n)$. Prowadzi to równania rekurencyjnego:

$$T(n) = 2T(n/2) + O(n) = O(n \, log \, n) \tag{19}$$

3 Działanie programów

Na rysunku 1 przedstawione zostały wyniki obliczeń szybkiej transformacji Fouriera przy uzyciu zaimplementowanego algorytmu oraz implementacji z pakietu Numpy dla przykładowego sygnału w dziedzinie czasu $a = sin(2\pi t)$ o częstotliwości próbkowania 8 Hz na przedziale 4 sekund.

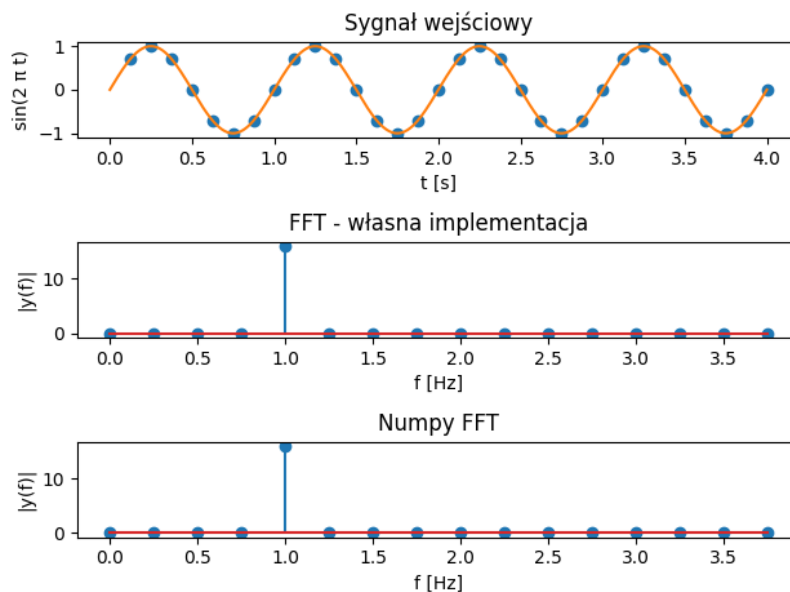
```
FFT obliczona przy pomocy zaimplementowanego algorytmu:
[-9.33947930e-16+0.00000000e+00j -1.18159308e-15+4.34808508e-17j
 -1.45201602e-15+4.30650238e-17j -3.53061466e-15+1.43797042e-15j
  1.13137085e+01-1.13137085e+01j  1.72626616e-15-5.04473351e-17j
  1.96527281e-15-4.17986656e-16j  8.83914898e-16-1.18108949e-15j
 -4.44089210e-16-4.89858720e-16j  6.77033607e-16+1.41027008e-16j
  1.43505355e-15-8.62075866e-16j  3.03936888e-16-2.07821884e-15j
  7.99360578e-15+4.44089210e-15j -1.40194848e-15-1.56411563e-17j
 -1.71953496e-16+4.87154234e-16j -1.41530595e-16+1.63400441e-16j
  4.57695098e-17+0.00000000e+00j -1.41530595e-16-1.63400441e-16j
 -1.71953496e-16-4.87154234e-16j -1.40194848e-15+1.56411563e-17j
  4.44089210e-15-3.55271368e-15j  3.03936888e-16+2.07821884e-15j
  1.43505355e-15+8.62075866e-16j  6.77033607e-16-1.41027008e-16j
 -4.44089210e-16+4.89858720e-16j  8.83914898e-16+1.18108949e-15j
  1.96527281e-15+4.17986656e-16j  1.72626616e-15+5.04473351e-17j
  1.13137085e+01+1.13137085e+01j -3.53061466e-15-1.43797042e-15j
 -1.45201602e-15-4.30650238e-17j -1.18159308e-15-4.34808508e-17j]

FFT obliczona przy pomocy funkcji z pakietu numpy:
[-9.33947930e-16+0.00000000e+00j -1.18159308e-15+4.34808508e-17j
 -1.45201602e-15+4.30650238e-17j -3.53061466e-15+1.43797042e-15j
  1.13137085e+01-1.13137085e+01j  1.72626616e-15-5.04473351e-17j
  1.96527281e-15-4.17986656e-16j  8.83914898e-16-1.18108949e-15j
 -4.44089210e-16-4.89858720e-16j  6.77033607e-16+1.41027008e-16j
  1.43505355e-15-8.62075866e-16j  3.03936888e-16-2.07821884e-15j
  4.44089210e-15+2.66453526e-15j -1.40194848e-15-1.56411563e-17j
 -1.71953496e-16+4.87154234e-16j -1.41530595e-16+1.63400441e-16j
  4.57695098e-17+0.00000000e+00j -1.41530595e-16-1.63400441e-16j
 -1.71953496e-16-4.87154234e-16j -1.40194848e-15+1.56411563e-17j
  4.44089210e-15-2.66453526e-15j  3.03936888e-16+2.07821884e-15j
  1.43505355e-15+8.62075866e-16j  6.77033607e-16-1.41027008e-16j
 -4.44089210e-16+4.89858720e-16j  8.83914898e-16+1.18108949e-15j
  1.96527281e-15+4.17986656e-16j  1.72626616e-15+5.04473351e-17j
  1.13137085e+01+1.13137085e+01j -3.53061466e-15-1.43797042e-15j
 -1.45201602e-15-4.30650238e-17j -1.18159308e-15-4.34808508e-17j]
```

Rysunek 1: Wyniki działania FFT zaimlementowanej w języku Python oraz zawartej w pakiecie Numpy

Na rysunku 2 można zobaczyć FFT obliczoną dla tego samego ciągu przy użyciu programu napisanego w języku C++. Jak widać każda z metod daje ten sam wynik. DFT możemy traktować jako przedstawienia wektora a w bazie rozpiętej przez sinusy i cosinusy o częstotliwościach: $0, \frac{1}{n \cdot dt}, \frac{2}{n \cdot dt}, ..., \frac{n/2-1}{n \cdot dt}$, gdzie dt to interwał czasowy pomiędzy kolejnymi próbkami [5]. Na tej podstawie możemy przypuszczać, że niezerowe współczynniki wektora $y = DFT(a)$ będą odpowiadać częstotliwości 1 Hz. Przypuszczenia te potwierdzają wykresy przedstawione na rysunku 3.

```
FFT
(-9.3395e-16,0)                (-1.1816e-15,4.3481e-17)
(-1.452e-15,4.3065e-17)        (-3.5306e-15,1.438e-15)
(11.314,-11.314)               (1.7263e-15,-5.0447e-17)
(1.9653e-15,-4.1799e-16)       (8.8391e-16,-1.1811e-15)
(-4.4409e-16,-4.8986e-16)      (6.7703e-16,1.4103e-16)
(1.4351e-15,-8.6208e-16)       (3.0394e-16,-2.0782e-15)
(7.9936e-15,4.4409e-15)        (-1.4019e-15,-1.5641e-17)
(-1.7195e-16,4.8715e-16)       (-1.4153e-16,1.634e-16)
(4.577e-17,0)                  (-1.4153e-16,-1.634e-16)
(-1.7195e-16,-4.8715e-16)      (-1.4019e-15,1.5641e-17)
(4.4409e-15,-3.5527e-15)       (3.0394e-16,2.0782e-15)
(1.4351e-15,8.6208e-16)        (6.7703e-16,-1.4103e-16)
(-4.4409e-16,4.8986e-16)       (8.8391e-16,1.1811e-15)
(1.9653e-15,4.1799e-16)        (1.7263e-15,5.0447e-17)
(11.314,11.314)                (-3.5306e-15,-1.438e-15)
(-1.452e-15,-4.3065e-17)       (-1.1816e-15,-4.3481e-17)
```

Rysunek 2: Wyniki działania FFT zaimlementowanej w języku C++



Rysunek 3: Sygnał wejściowy i jego transformacja Fouriera

4 Podsumowanie

W pracy udało się skutecznie zaimplementować algorytm obliczający dyskterną transformację Fouriera w czasie $O(n \, log \, n)$. Algorytm posiada jednak ograniczenie polegające na mozliwości obliczania FFT tylko dla ciągów, których długość jest potęgą dwójki. Dla autorów stanowi to motywację do dalszych poszukiwań.

A FFT - Python

```
 #!/usr/bin/env python3
import numpy as np

def FFT(a):
    n = len(a)
    y = np.zeros([n], dtype = complex)
    if n == 1: return a
    wn = np.exp(-2 * np.pi * 1j / n)
    w = 1
    a_0 = a[::2]
    a_1 = a[1::2]
    y_0 = FFT(a_0)
    y_1 = FFT(a_1)
    for k in range(n//2):
        y[k] = y_0[k] + w * y_1[k]
        y[k + n//2] = y_0[k] - w * y_1[k]
        w *= wn
    return y
```

B FFT - C++

```
#include <iostream>
#include <complex>
#include <math.h>
#include <vector>

using namespace std;

using cd = complex<double>;
const double pi = acos(-1);

vector<cd> fft(vector<cd> a)
{
    int n = a.size();
    vector<cd> y(n);
    if (n == 1)
        return a;
    const cd ang = -2 * pi * 1i / n;
    cd wn = exp(ang);
    cd w = 1;
    vector<cd> a_0(n / 2), a_1(n / 2);
    for (int i = 0; i < n/2; i++)
    {
```

```
        a_0[i] = a[2*i];
        a_1[i] = a[2*i+1];
    }
    vector<cd> y_0 = fft(a_0);
    vector<cd> y_1 = fft(a_1);
    for (int i = 0; i < n/2; i++)
    {
        y[i] = y_0[i] + w * y_1[i];
        y[i + n/2] = y_0[i] - w * y_1[i];
        w *=wn;
    }
    return y;
}
```

Literatura

[1] Cormen C. H., Leiserson, C. E., Rivest R. L., *Wprowadzenie do Aagorytmów*, Warszawa: WNT 2001

[2] https://en.wikipedia.org/wiki/Discrete_Fourier_transform (Dostęp: 28.06.2020)

[3] https://pl.wikipedia.org/wiki/Szybka_transformacja_Fouriera (Dostęp: 28.06.2020)

[4] https://cp-algorithms.com/algebra/fft.html (Dostęp: 29:06.2020)

[5] http://th-www.if.uj.edu.pl/zfs/gora/timeseries08/wyklad01.pdf (Dostęp: 29.06.2020)