Metody Numeryczna, kierunek informatyka,
Lab15: Szybka transformacja Fouriera (FFT)

DYSKRETNA TRANSFORMACJA FOURIERA (DFT, ang. *discrete Fourier transform*)

Niech $(y_j)_{j=0}^{n-1} \subset \mathbb{C}$ będzie danym ciągiem liczb zespolonych o długości n. *Dyskretną transformacją Fouriera* ciągu $(y_j)_{j=0}^{n-1}$ nazywamy ciąg liczb zespolone $(\hat{y})_{k=0}^{n-1}$ określony są wzorami

$$\hat{y}_k = \sum_{j=0}^{n-1} y_j e^{-2\pi i k j/n}, \qquad 0 \le k \le n-1, \tag{1}$$

gdzie $i \in \mathbb{C}$ jest jednostką urojoną (pierwiastkiem kwadratowym z -1). Dla ułatwienia dalszych rachunków definicję (1) zapisuje się zazwyczaj wprowadzając symbol ω_n oznaczający pierwszy zespolony pierwiastek n-tego stopnia z 1: $\omega_n = e^{\frac{2\pi i}{n}}$. Poniżej są zebrane podstawowe własności tego pierwiastka, które będą wykorzystywane w dalszej części (ponadto korzystamy ze znanej tożsamości Eulera, $e^{ix} = \cos x + i \sin x$ dla dowolnego $x \in \mathbb{R}$):

$$\begin{aligned} &\omega_n := e^{2\pi i/n} = \cos\frac{2\pi}{n} + i \sin\frac{2\pi}{n}, \\ &\omega_n^k = (\omega_n)^k = e^{2\pi k i/n}, \quad \overline{\omega_n^k} = e^{-2\pi k i/n}, \quad k \in \mathbb{Z}, \\ &(\omega_n)^n = (e^{2\pi i/n})^n = e^{2\pi i} = 1, \end{aligned} \tag{2}$$

gdzie $\overline{z}$ oznacza sprzężenie zespolone liczby $z \in \mathbb{C}$.

Definicję DFT daną wzorem (1) przedstawimy teraz następująco

$$\hat{y}_k = \sum_{j=0}^{n-1} y_j \omega_n^{-kj}, \qquad 0 \le k \le n-1. \tag{3}$$

Dyskretną transformację Fouriera możemy także wyrazić w formie mnożenia macierzowego

$$\hat{\boldsymbol{y}} = \begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_{n-2} \\ \hat{y}_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-2} & \omega_n^{2(n-2)} & \omega_n^{3(n-2)} & \cdots & \omega_n^{(n-1)(n-2)} \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{bmatrix}, \qquad \hat{\boldsymbol{y}} = \overline{\mathrm{DFT}}_n \cdot \boldsymbol{y}, \tag{4}$$

gdzie $\mathrm{DFT}_n \in \mathbb{C}^{n \times n}$ jest zespoloną macierzą kwadratową rozmiaru n zdefiniowaną w (4), a $\overline{z} = x - iy$ oznacza sprzężenie zespolone liczby $z = x + iy \in \mathbb{C}$. Zauważmy, że elementami tej macierzy są po prostu odpowiednie potęgi pierwiastka ω_n: $(\mathrm{DFT}_n)_{k,j} = \omega_n^{kj}$ dla $k, j = 0, 1, \ldots, n-1$.

Transformacja DFT jest odwracalna. Na podstawie ciągu $(\hat{y}_k)_{k=0}^{n-1}$ można *jednoznacznie* odtworzyć oryginalny ciąg $(y)_{j=0}^{n-1}$. W zasadzie opiera się to na znanej (szkolnej?) tożsamości: dla dowolnego $z \in \mathbb{C}$ zachodzi

$$\sum_{j=0}^{n-1} z^j = 1 + z + z^2 + \ldots + z^{n-1} = \begin{cases} \dfrac{1-z^n}{1-z}, & \text{gdy } z \ne 1, \\ n, & \text{gdy } z = 1. \end{cases} \tag{5}$$

Z tożsamości tej wynika

$$\sum_{j=0}^{n-1}\omega_n^{k(s-j)}=\sum_{j=0}^{n-1}(\omega_n^{s-j})^k=\begin{cases}\dfrac{1-\omega_n^{-(s-j)n}}{1-\omega_n^{s-j}}, & \text{gdy } s\neq j,\\ n, & \text{gdy } s=j,\end{cases}=\begin{cases}\dfrac{1-1}{1-\omega_n^{s-j}}, & \text{gdy } s\neq j,\\ n, & \text{gdy } s=j,\end{cases}=\begin{cases}0, & \text{gdy } s\neq j,\\ n, & \text{gdy } s=j,\end{cases} \quad (6)$$

gdzie wykorzystano $\omega_n^{-(s-j)n}=e^{(2\pi i/n)(s-j)n}=e^{2\pi i(s-j)}=1.$

Możemy teraz pokazać odwracalność DFT: ustalmy indeks $s\in\{0,1,\ldots,n-1\}$, pomnóżmy wyrazy ciągu (3) przez współczynniki ω_n^{ks} i zsumujmy po k:

$$\sum_{k=0}^{n-1}\hat{y}_k\omega_n^{ks}=\sum_{k=0}^{n-1}\left(\sum_{j=0}^{n-1}y_j\omega_n^{-kj}\right)\omega_n^{ks}=\sum_{k=0}^{n-1}\sum_{j=0}^{n-1}y_j\omega_n^{-kj}\omega_n^{ks}=\sum_{j=0}^{n-1}\sum_{k=0}^{n-1}y_j\omega_n^{ks-kj}$$
$$=\sum_{j=0}^{n-1}y_j\sum_{k=0}^{n-1}\omega_n^{k(s-j)}=\sum_{j=0}^{n-1}y_jn\delta_{js}=ny_s,$$

skąd

$$y_s=\frac{1}{n}\sum_{k=0}^{n-1}\hat{y}_k\omega_n^{sk},\quad 0\le s\le n-1. \quad (7)$$

Wykonanie mnożenia macierzowego (4), czyli obliczenie n współczynników wg wzorów (1) wymaga n^2 mnożeń liczb zespolonych (zatem $4n^2$ mnożeń rzeczywistych). Tak więc złożoność takiej procedury jest $O(n^2)$. W zastosowaniach, na przykład przy przetwarzaniu sygnałów w czasie rzeczywistym, gdzie trzeba w krótkim czasie dokonywać wielu DFT na ciągach liczb o długości tysięcy, taka złożoność praktycznie uniemożliwiałaby wykorzystanie DFT. Okazuje się jednak, że można *znacznie* efektywniej obliczyć DFT, czyli ciąg $(\hat{y}_k)$, stosując pewien pomysłowy sposób obliczania sum (3), który wykorzystuje szczególne własności macierzy DFT_n. Metodę taką zaproponowali w 1965 r. James Cooley i John Tukey i nosi ona nazwę *szybkiej transformacji Fouriera*, w skrócie FFT (ang. *Fast Fourier Transform*).[1] Obecnie istnieje wiele wariantów tej metody, bardziej optymalnych niż oryginalny algorytm Tukeya–Cooley'ego, choć asymptotyczna złożoność wszystkich wariantów jest tego samego rzędu $O(n\log n)$.

Aby zrozumieć dlaczego można tak drastycznie zredukować złożoność obliczeniową (z n^2 do $n\log n$) zobaczmy jak można obliczać DFT dla ciągu o długości $2n$. Mamy więc dany ciąg $(y_j)_{j=0}^{2n-1}$ i obliczamy DFT, czyli

$$\hat{y}_k=\sum_{j=0}^{2n-1}y_j\omega_{2n}^{-kj},\qquad 0\le k\le 2n-1,$$

co rozbijamy na sumę dwóch składników: po indeksach parzystych $(2j)$ oraz nieparzystych $(2j+1)$:

$$\hat{y}_k=\sum_{j=0}^{2n-1}y_j\omega_{2n}^{-kj}=\sum_{j=0}^{n-1}y_{2j}\omega_{2n}^{-2kj}+\sum_{j=0}^{n-1}y_{2j+1}\omega_{2n}^{-k(2j+1)}=\sum_{j=0}^{n-1}y_{2j}\omega_{2n}^{-2kj}+\sum_{j=0}^{n-1}y_{2j+1}\omega_{2n}^{-2kj}\omega_{2n}^{-k}$$
$$=\sum_{j=0}^{n-1}y_{2j}(\omega_{2n}^2)^{-kj}+\omega_{2n}^{-k}\sum_{j=0}^{n-1}y_{2j+1}(\omega_{2n}^2)^{-kj}.$$

Ponieważ $\omega_{2n}^2=(e^{-2\pi i/2n})^2=e^{-2\pi i/n}=\omega_n$, więc powyższe sumy możemy przepisać tak

[1] J.W. Cooley, J.W. Tukey, An Algorithm for the Machine Calculation of Complex Fourier Series, *AMS Math. Comp*. (**19**), 1965, 297–301.

$$\hat{y}_k = \sum_{j=0}^{n-1} y_{2j}\,\omega_n^{-kj} \;+\; \omega_{2n}^{-k} \sum_{j=0}^{n-1} y_{2j+1}\,\omega_n^{-kj}, \qquad 0 \le k \le 2n-1. \tag{8}$$

Zauważmy, że dla $k,\ j$ całkowitych zachodzą tożsamości

$$\omega_n^{-(k+n)j} = \omega_n^{-kj}, \quad \omega_{2n}^{-(k+n)} = \omega_{2n}^{-k}, \quad \omega_{2n}^{-n} = -1,$$

które pozwalają wyrazić sumy (8) następująco: dla $0 \le k \le n-1$ nic nie zmieniamy, ale dla $n \le k \le 2n-1$ zamieniamy indeks k na $k+n$, gdzie $0 \le k \le n-1$:

$$\hat{y}_{k+n} = \sum_{j=0}^{n-1} y_{2j}\,\omega_n^{-(k+n)j} \;+\; \omega_{2n}^{-(k+n)} \sum_{j=0}^{n-1} y_{2j+1}\,\omega_n^{-(k+n)j} = \sum_{j=0}^{n-1} y_{2j}\,\omega_n^{-kj}\,\underbrace{\omega_n^{-nj}}_{=1} \;+\; \omega_{2n}^{-k}\,\underbrace{\omega_{2n}^{-n}}_{-1} \sum_{j=0}^{n-1} y_{2j+1}\,\omega_n^{-kj}\,\underbrace{\omega_n^{-nj}}_{=1}$$

$$= \sum_{j=0}^{n-1} y_{2j}\,\omega_n^{-kj} \;-\; \omega_{2n}^{-k} \sum_{j=0}^{n-1} y_{2j+1}\,\omega_n^{-kj}.$$

Tak więc wzory (8) sprowadzają się do

$$\begin{aligned} \hat{y}_k &= \sum_{j=0}^{n-1} y_{2j}\,\omega_n^{-kj} \;+\; \omega_{2n}^{-k} \sum_{j=0}^{n-1} y_{2j+1}\,\omega_n^{-kj}, \qquad 0 \le k \le n-1, \\ \hat{y}_{k+n} &= \sum_{j=0}^{n-1} y_{2j}\,\omega_n^{-kj} \;-\; \omega_{2n}^{-k} \sum_{j=0}^{n-1} y_{2j+1}\,\omega_n^{-kj}, \qquad 0 \le k \le n-1. \end{aligned} \tag{9}$$

Z powyższych równości wynika, że dla $0 \le k \le n-1$ obliczenie wyrazów $\hat{y}_k$ oraz $\hat{y}_{k+n}$ wymaga użycia dokładnie tych samych dwóch sum

$$\sum_{j=0}^{n-1} y_{2j}\,\omega_n^{-kj} \quad \text{oraz} \quad \sum_{j=0}^{n-1} y_{2j+1}\,\omega_n^{-kj}.$$

Łatwo jest teraz podać zależność rekurencyjną pomiędzy N_{2n} – liczbą mnożeń dla FFT_{2n} a N_n – liczbą mnożeń dla FFT_n. Ze wzorów (9) wynika bowiem, że aby otrzymać FFT_{2n} musimy obliczyć dwie transformacje FFT_n (liczba mnożeń $2N_n$) oraz dodatkowo wykonać n mnożeń $\omega_{2n}^{-k} \cdot \sum_{j=0}^{n-1} y_{2j+1}\,\omega_n^{-kj}$, gdyż $k = 0,\ldots,n-1$. Zatem otrzymujemy

$$N_{2n} = 2N_n + n. \tag{10}$$

Najprostszą wersję FFT otrzymujemy gdy przyjmiemy, że liczba wartości w próbce jest potęgą dwójki, $n = 2^m$. Wtedy wzór (10) przepisany dla n zamiast $2n$ przyjmie postać

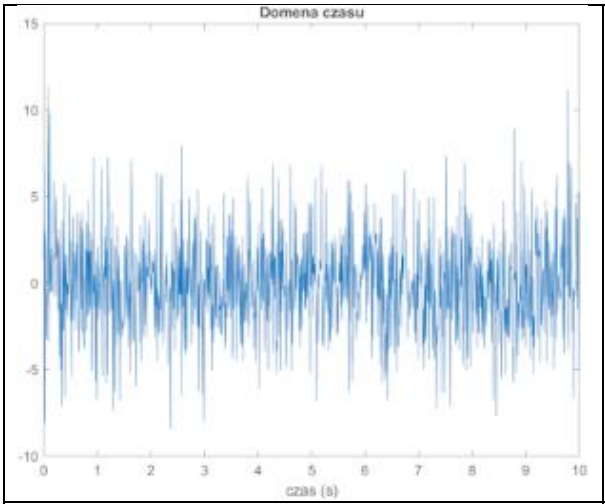
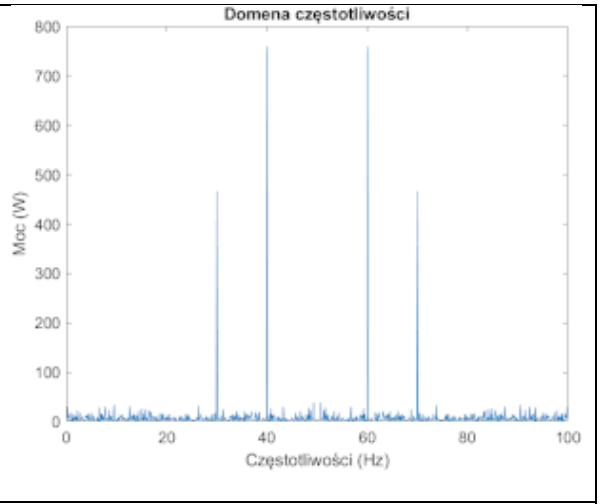
$$N_n = 2N_{n/2} + n/2$$

skąd podstawiając $n = 2^m$

$$N_n = N_{2^m} = 2N_{2^{m-1}} + 2^{m-1} = 2(2N_{2^{m-2}} + 2^{m-2}) + 2^{m-1} = 2^2 N_{2^{m-2}} + 2\cdot 2^{m-1} = 2^2(2N_{2^{m-3}} + 2^{m-3}) + 2\cdot 2^{m-1}$$

$$= 2^3 N_{2^{m-3}} + 3\cdot 2^{m-1} = \ldots = 2^m N_1 + m\cdot 2^{m-1} = 0 + \tfrac{1}{2} m\cdot 2^m = \boxed{\tfrac{1}{2} n \log_2 n.}$$

Przykład. Jednym z podstawowych zastosowań FFT jest przetwarzanie sygnałów. Może to na przykład oznaczać filtrowanie sygnału czasowego, aby wydobyć w czasie rzeczywistym główne składowe częstotliwości. Poniższy obrazek ilustruje graficznie taka sytuację.

	
Zaszumiony sygnał (domena czasowa)	DFT sygnału (domena częstotliwości)

Biblioteka GSL dostarcza wielu funkcji wykonujących szybką transformację Fouriera. Należą one do dwóch grup: (i) funkcje *„radix2”*, które działają na ciągach o długości będącej potęgą dwójki $(n = 2^m)$, (ii) funkcje, które działają na ciągach o dowolnej długości.

Na zajęciach wykorzystamy dwie z nich:

```
gsl_fft_complex_radix2_forward();
gsl_fft_complex_forward();
```

Wykorzystanie tych funkcji wymaga włączenia odpowiednich plików nagłówkowych zawierających prototypy tych funkcji:

```
#include "/usr/include/gsl/gsl_errno.h"
#include "/usr/include/gsl/gsl_fft_complex.h"
```

Funkcja `gsl_fft_complex_radix2_forward()` jest wyspecjalizowana do danych o długości, które są potęgą dwójki: $n = 2^m$. Dzięki takiemu rozmiarowi można zaimplementować bardziej wydajną wersję algorytmu FFT – tzw. *split-radix FFT*. Opiera się ona na rekurencyjnym rozbiciu problemu rozmiaru n na jeden pod-problem rozmiaru $n/2$ i dwóch rozmiaru $n/4$. Natomiast funkcja `gsl_fft_complex_forward()` jest bardziej ogólna (i mniej efektywna), i może być stosowana dla danych o dowolnym rozmiarze n.

Przykładowe wykorzystanie funkcji podane jest poniżej. Proszę zwrócić uwagę, że długość danych jest równa $n = 128 = 2^7$, oraz że funkcja wymaga w ogólności danych zespolonych. W tym celu deklarujemy tablicę rozmiaru $2n$, w której wejściowe dane zespolone $(y_j)_{j=0}^{n-1} \subset \mathbb{C}$ przechowywane są parami: $\mathrm{Re}\, y_j, \mathrm{Im}\, y_j$ w sposób następujący: `y[2j]`$= \mathrm{Re}\, y_j$, `y[2j+1]`$= \mathrm{Im}\, y_j$ dla $0 \le j \le n-1$. To znaczy na pozycjach o parzystych indeksach znajdują się części rzeczywiste, a na pozycjach o nieparzystych indeksach znajdują się części urojone danych. Jeżeli dane wejściowe są rzeczywiste, to oczywiście wypełniamy `y[2j+1]`$= 0$.

```c
#include <stdio.h>
#include "/usr/include/gsl/gsl_math.h"
#include "/usr/include/gsl/gsl_errno.h"
#include "/usr/include/gsl/gsl_fft_complex.h"
#define n (128)
#define pi (3.14159265)

int main() {
    int i;
    double span = 6, t, y[2*n];

    for (i=0; i < n; i++) {
        t = (span/n)*i;
        y[2*i] = 3.4*sin(2*pi*t)+4.9*sin(5*pi*t)-8.1*sin(9*pi*t);
        y[2*i+1] = 0.0;
    }

    gsl_fft_complex_radix2_forward(y, 1, n);

    for (i=0; i < n; i++)
        printf("Moc[%d] = %.2f\n", i, pow(y[2*i],2)+pow(y[2*i+1],2));
}
```

Użycie funkcji `gsl_fft_complex_forward` wymaga dwóch pomocniczych deklaracji:

```
gsl_fft_complex_wavetable *wavetable;
gsl_fft_complex_workspace *workspace;
```

oraz inicjalizacji tych dwóch obiektów:

```
wavetable = gsl_fft_complex_wavetable_alloc(n);
workspace = gsl_fft_complex_workspace_alloc(n);
```

Tablicę y[2n] używamy jak dla radix2, ale wywołanie funkcji ma teraz postać:

```
gsl_fft_complex_forward(y, 1, n, wavetable, workspace);
```

ZADANIA DO WYKONANIA W PROJEKCIE

1) Wygenerować dane dla funkcji

$$f(t) = 5{,}3\sin(2\pi t) - 6{,}7\sin(2\pi 4t) + 2{,}1\sin(2\pi 7t),$$

na przedziale $0 \le t \le 4\pi$ z próbkowaniem $n = 64$ i $n = 128$. Dane zapisać do pliku. Następnie napisać program, który będzie odczytywał dane z tego pliku do odpowiedniej tablicy, wykonywał FFT algorytmem radix2, wypisywał na ekranie spektrum mocy sygnałów oraz zapisywał spektrum do innego pliku.

2) Wygenerować dane jak w punkcie 1), ale dodatkowo zniekształcić je dodając szum. W celu zaszumienia można dodać losowe wartości z przedziału $[-1, 1]$ wykorzystując jakąś funkcję zwracającą liczby losowe o rozkładzie jednostajnym. Może to wymagać odpowiedniego przeskalowania tych liczb, aby wchodziły do przedziału $[-1, 1]$. Dane zapisać do pliku i sporządzić ich wykres (dowolne narzędzie: może być MATLAB, Ms Excel, Gnuplot …). Następnie

przeprowadzić przekształcenie FFT algorytmem radix2, zapisać spektrum mocy sygnałów do pliku i na ekran.

3) Wykonać program analogiczny do p.2) ale wykorzystać własna funkcję okresowa o czterech częstotliwościach, dodać szum, a następnie wykonać FFT funkcją

```
gsl_fft_complex_forward();
```

Dodać filtr, który wypisze na ekranie tylko te częstotliwości, których moc jest co najmniej równa 70% wartości średniej mocy w sygnale.