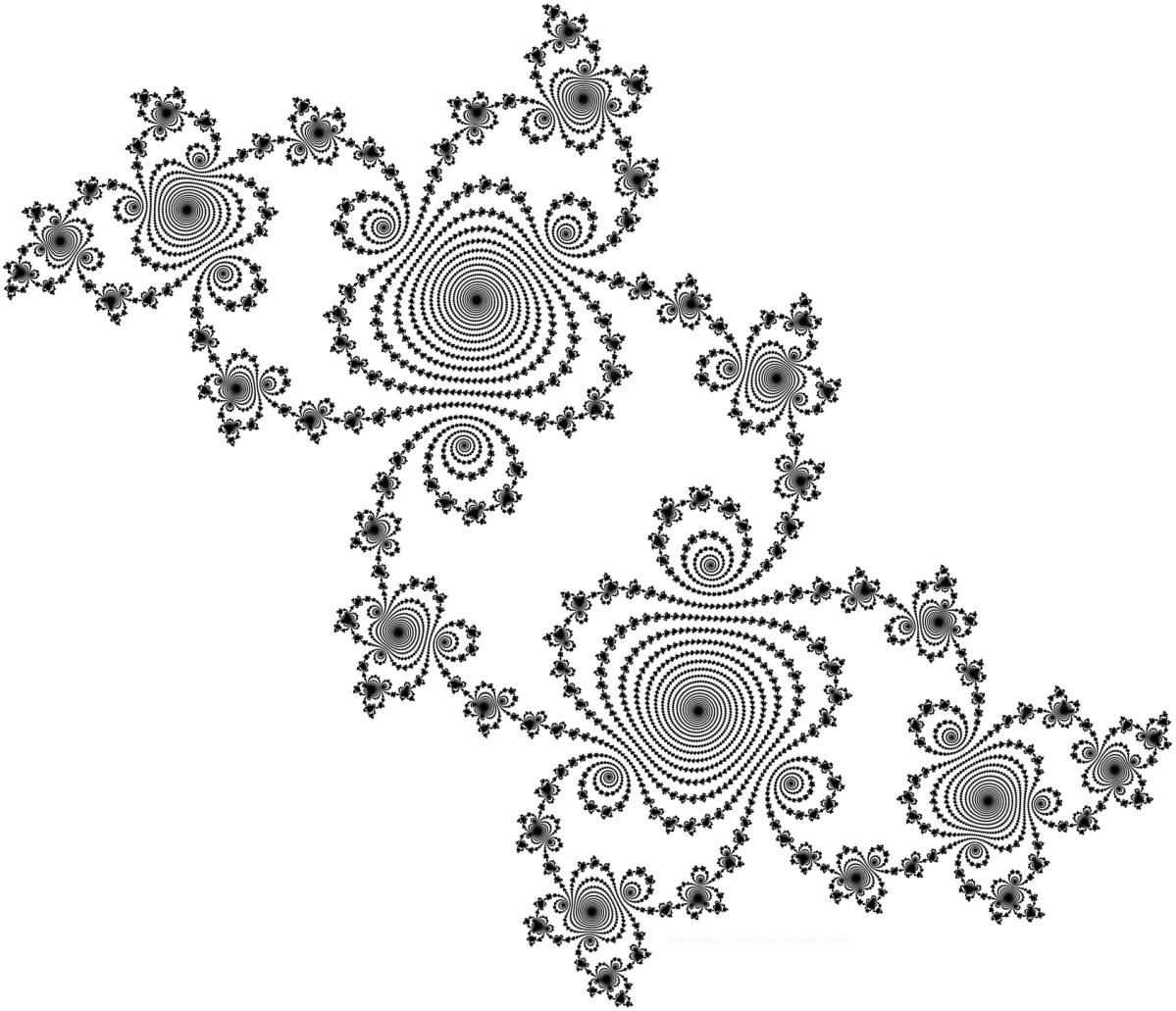
Zbiór problemów i pytań bez odpowiedzi



Szymon M. Woźniak
https://s.zymon.org

Spis treści

1. Problemy na rozruch

Problem 1.1: Zaimplementuj funkcję $\mathbb{N} \to \mathbb{N}^+$, która zwróci wartość funkcji silnia $n! \in \mathbb{N}^+$ dla liczny naturalne $n \in \mathbb{N}$. Obliczenia zaimplementuj rekurencyjnie.

Problem 1.2: Zaimplementuj funkcję $\mathbb{N} \to \mathbb{N}^+$, która zwróci wartość funkcji silnia $n! \in \mathbb{N}^+$ dla liczny naturalne $n \in \mathbb{N}$. Obliczenia zaimplementuj iteracyjnie.

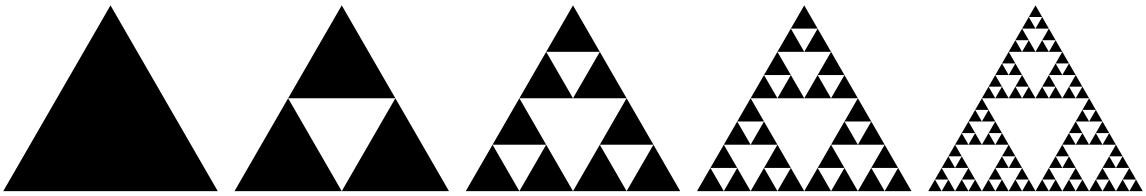
Problem 1.3: Zaimplementuj funkcję $\mathbb{N} \to \{0, 1\}$, która zwróci logiczną wartość prawda jeżeli liczba $N \in \mathbb{N}$ jest parzysta.

Problem 1.4: Zaimplementuj funkcję $\mathbb{N} \to \{0, 1\}$, która zwróci logiczną wartość prawda jeżeli $N \in \mathbb{N}$ jest liczbą pierwszą.

Problem 1.5: Zaimplementuj funkcję $S \to S$, która zwróci wejściowy ciąg znakowy $s \in S$ w odwrotnej kolejności. Symbol S oznacza zbiór wszystkich ciągów znakowych.

Problem 1.6: Zaimplementuj funkcję $S \to \{0, 1\}$, która zwróci logiczną wartość prawda jeżeli ciąg znaków $a \in S$ jest palindromem. Symbol S oznacza zbiór wszystkich ciągów znakowych.

Problem 1.7: Zaimplementuj funkcję $\mathbb{N} \to \mathbb{R}^+$, która zwróci pole powierzchni trójkąta Sierpińskiego rzędu $N \in \mathbb{N}$. Załóż że pole powierzchni trójkąta Sierpińskiego rzędu 0 wynosi 1.0 i maleje wraz ze zwiększającym się rzędem.



Rysunek 1: Trójkąty Sierpińskiego następujących rzędów (od lewej): 0, 1, 2, 3, 4.

Problem 1.8: Zaimplementuj funkcję $\mathbb{R}^+ \to \mathbb{R}^+$, która obliczy $\sqrt{a}$ dla $a \geq 0$ z wykorzystaniem metody Newtona.

Problem 1.9: Dany jest rekurencyjny ciąg $z_{n+1} = z_n^2 + p$, gdzie $z_n, p \in \mathbb{C}$ oraz $z_0 = 0$. Zbadaj i określ maksymalną K-zbieżność ciągu z_n dla punktów p takich że,

$$-1 < \mathfrak{R}(p) < 2,$$

$$-1 < \mathfrak{I}(p) < 1.$$

Zbiór K-zbieżnych punktów p ciągu z_n zdefiniowany jest jako:

$$M_K = \{p \in \mathbb{C} : \forall_{n<K} |z_n| < 2\}.$$

Wyrysuj płaszczyznę zespoloną, gdzie dla każdego zbadanego punktu p, przypisz maksymalne K, dla którego ciąg jest K-zbieżny.

Problem 1.10: Liczbę pierwszą nazwiemy cykliczną, jeżeli wszystkie cykliczne przesunięcia tej liczby są też liczba pierwszą. Cykliczne przesunięcia liczby pierwszej 197 to 971 i 719. Istnieje trzynaście cyklicznych liczb pierwszych poniżej stu: 2, 3, 5, 7, 11, 13, 17, 31, 37, 71, 73, 79, 97. Ile takich liczb jest poniżej miliona?

2. Modelowanie sygnałów ciągłych

Problem 2.1: Oblicz wektor $x \in \mathbb{R}^{256}$, zawierający kolejne próbki pobrane z szybkością 1000 próbek na sekundę ciągłego rzeczywsitego syganłu harmonicznego o amplitudzie 2, częstotliwości oscylacji 25 Hz, oraz przesunięciu fazowym $\frac{\pi}{4}$. Pierwsza próbka x_1 powinna być pobrana w momencie 0.25 s.

Problem 2.2: Oblicz wektor $x \in \mathbb{C}^N$, zawierający kolejne próbki pobrane z szybkością 2048 próbek na sekundę ciągłego zespolonego syganłu harmonicznego o amplitudzie 0.25, częstotliwości oscylacji $\frac{\pi}{2}$ Hz, oraz przesunięciu fazowym π. Pierwsza próbka x_1 powinna być pobrana w momencie 5.0 s, natomiast ostatnia próbka x_N powinna zostać pobrana w momencie 10.0 s.

Problem 2.3: Wygeneruj wektor $x \in \mathbb{R}^{1000}$, zawierający próbki szumu białego o mocy 0.25.

Problem 2.4: Wygeneruj wektor $x \in \mathbb{C}^{1000}$, zawierający próbki zespolonego szumu białego o mocy 2.

Problem 2.5: Zaimplementuj funkcję `cw_rectangular`: $\mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}$, zwracającą wartości impulsu prostokątnego o szerokości $T \in \mathbb{R}^+$ w chwili $t \in \mathbb{R}$.

Problem 2.6: Zaimplementuj funkcję `cw_triangle`: $\mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}$, zwracającą wartości impulsu trójkątnego o szerokości $T \in \mathbb{R}^+$ w chwili $t \in \mathbb{R}$.

Problem 2.7: Zaimplementuj funkcję `cw_literka_M`: $\mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}$, zwracającą impuls przypominający literę `M` o szerokości $T \in \mathbb{R}^+$ w chwili $t \in \mathbb{R}$.

Problem 2.8: Zaimplementuj funkcję `cw_literka_U`: $\mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}$, zwracającą impuls przypominający literę `U` o szerokości $T \in \mathbb{R}^+$ w chwili $t \in \mathbb{R}$.

Problem 2.9: Zaimplementuj funkcję `ramp_wave`: $\mathbb{R} \to \mathbb{R}$, zwracającą wartości okresowego sygnału fali piłokształtnej z narastającym zboczem w chwili $t \in \mathbb{R}$. Sygnał powinien posiadać następujące parametry: amplituda 1, okres 1 sekunda, składowa stała 0, $\text{ramp\_wave}(0) = 0$, oraz $\frac{\partial\ \text{ramp\_wave}}{\partial t}\ |_{t=0} = 1$.

Problem 2.10: Zaimplementuj funkcję `sawtooth_wave`: $\mathbb{R} \to \mathbb{R}$, zwracającą wartości okresowego sygnału fali piłokształtnej z opadającym zboczem w chwili $t \in \mathbb{R}$. Sygnał powinien posiadać następujące parametry: amplituda 1, okres 1 sekunda, składowa stała 0, $\text{sawtooth\_wave}(0) = 0$, oraz $\frac{\partial\ \text{sawtooth\_wave}}{\partial t}\ |_{t=0} = -1$.

Problem 2.11: Zaimplementuj funkcję `triangular_wave`: $\mathbb{R} \to \mathbb{R}$, zwracającą wartości okresowego sygnału fali trójkątnej w chwili $t \in \mathbb{R}$. Sygnał powinien posiadać następujące parametry: amplituda 1, okres 1 sekunda, składowa stała 0, $\text{triangular\_wave}(0) = 0$, oraz $\frac{\partial\ \text{triangular\_wave}}{\partial t}\ |_{t=0} = 4$.

Problem 2.12: Zaimplementuj funkcję `square_wave`: $\mathbb{R} \to \mathbb{R}$, zwracającą wartości okresowego sygnału bipolarnej fali prostokątnej w chwili $t \in \mathbb{R}$. Sygnał powinien posiadać następujące parametry: amplituda 1, okres 1 sekunda, składowa stała 0, oraz $\text{square\_wave}(t) = 1$ dla $t \in \left(0, \frac{1}{2}\right)$.

Problem 2.13: Zaimplementuj funkcję `pulse_wave`: $\mathbb{R} \times [0, 1] \to \mathbb{R}$ zwracającą wartości okresowego sygnału (unipolarnego) impulsu prostokątnego w chwili $t \in \mathbb{R}$ o współczynniku wypełnienia $\rho \in [0, 1]$. Sygnał powinien posiadać następujące parametry: okres 1 s, składowa stała ρ, oraz $\text{pulse\_wave}(t) = 1$ dla $t \in (0, \rho)$.

Problem 2.14: Zaimplementuj funkcję `impulse_reapeter`: $F \times \mathbb{R} \times \mathbb{R} \to F$, która zwróci funkcję $f \in F$, gdzie F oznacza zbiór funkcji $\mathbb{R} \to \mathbb{R}$. Funkcja f powinna zwracać wartość sygnału $f(t) \in \mathbb{R}$ w momencie $t \in \mathbb{R}$. Sygnał $f(t)$ jest sygnałem okresowym o okresie $T = t_2 -$

t_1, gdzie $t_1, t_2 \in \mathbb{R}$ oraz $t_1 < t_2$. Funkcja f jest związany z funkcją wymuszającą $g \in F$ poprzez warunek

$$\int_{t_1}^{t_2} (f(t) - g(t))^2 \, \mathrm{dt} = 0.$$

Problem 2.15: Zaimplementuj funkcję `ramp_wave_bl`: $\mathbb{R} \times \mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}$. Funkcja powinna zwracać wartość sygnału fali piłokształtnej z narastającmym zboczem w chwili $t \in \mathbb{R}$ (podobnie jak w Problem 2.9). Dodatkowo zwracany sygnał powinien mieć amplitudę $A \in \mathbb{R}^+$, okres trwający $T \in \mathbb{R}^+$ sekund oraz pasmo tego sygnału powinno być ograniczone od dołu i góry przez częstotliwości $|B| \in \mathbb{R}^+$.

Problem 2.16: Zaimplementuj funkcję `sawtooth_wave_bl`: $\mathbb{R} \times \mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}$. Funkcja powinna zwracać wartość sygnału fali piłokształtnej z opadającym zboczem w chwili $t \in \mathbb{R}$ (podobnie jak w Problem 2.10). Dodatkowo zwracany sygnał powinien mieć amplitudę $A \in \mathbb{R}^+$, okres trwający $T \in \mathbb{R}^+$ sekund oraz pasmo tego sygnału powinno być ograniczone od dołu i góry przez częstotliwości $|B| \in \mathbb{R}^+$.

Problem 2.17: Zaimplementuj funkcję `triangular_wave_bl`: $\mathbb{R} \times \mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}$. Funkcja powinna zwracać wartość sygnału fali trójkątne w chwili $t \in \mathbb{R}$ (podobnie jak w Problem 2.11). Dodatkowo zwracany sygnał powinien mieć amplitudę $A \in \mathbb{R}^+$, okres trwający $T \in \mathbb{R}^+$ sekund oraz pasmo tego sygnału powinno być ograniczone od dołu i góry przez częstotliwości $|B| \in \mathbb{R}^+$.

Problem 2.18: Zaimplementuj funkcję `square_wave_bl`: $\mathbb{R} \times \mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}$. Funkcja powinna zwracać wartość sygnału bipolarnej fali prostokątne w chwili $t \in \mathbb{R}$ (podobnie jak w Problem 2.12). Dodatkowo zwracany sygnał powinien mieć amplitudę $A \in \mathbb{R}^+$, okres trwający $T \in \mathbb{R}^+$ sekund oraz pasmo tego sygnału powinno być ograniczone od dołu i góry przez częstotliwości $|B| \in \mathbb{R}^+$.

Problem 2.19: Zaimplementuj funkcję `pulse_wave_bl`: $\mathbb{R} \times [0, 1] \times \mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}$. Funkcja powinna zwracać wartość sygnału impulsu prostokątne w chwili $t \in \mathbb{R}$ o współczynniku wypełnienia $\rho \in [0, 1]$ (podobnie jak w Problem 2.13). Dodatkowo zwracany sygnał powinien mieć amplitudę $A \in \mathbb{R}^+$, okres trwający $T \in \mathbb{R}^+$ sekund oraz pasmo tego sygnału powinno być ograniczone od dołu i góry przez częstotliwości $|B| \in \mathbb{R}^+$.

Problem 2.20: Zaimplementuj funkcję `impulse_reapeter_bl`: $F \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}^+ \to F$, która zwróci funkcję $f_B \in F$, gdzie F oznacza zbiór funkcji $\mathbb{R} \to \mathbb{R}$. Funkcja f_B powinna zwracać wartość sygnału $x(t) \in \mathbb{R}$ w momencie $t \in \mathbb{R}$. Sygnał x jest sygnałem okresowym o okresie $T = t_2 - t_1$, gdzie $t_1, t_2 \in \mathbb{R}$ oraz $t_1 < t_2$. Pasmo sygnału x powinno być ograniczone od dołu i góry przez częstotliwość $|B| \in \mathbb{R}^+$. Funkcja f_B jest związany z funkcją wymuszającą $g \in F$ poprzez warunek

$$\lim_{B \to \infty} \int_{t_1}^{t_2} (f_B(t) - g(t))^2 \, \mathrm{dt} = 0.$$

Problem 2.21: Zaimplementuj funkcję `rand_signal_bl`: $\mathbb{R}^+ \times \mathbb{R}^+ \to F$, która zwróci funkcję $f \in F$, gdzie F oznacza zbiór funkcji $\mathbb{R} \to \mathbb{R}$. Funkcja f powinna zwracać wartość sygnału $x(t) \in \mathbb{R}$ w momencie $t \in \mathbb{R}$. Sygnał x jest pseudolosowym sygnałem którego pasmo jest ograniczone przez częstotliwości $f_1, f_2 \in \mathbb{R}$, gdzie $f_1 < f_2$. Moc sygnału x powinna wynosić w przybliżeniu 1.0.

Problem 2.22: Zaimplementuj funkcję kronecker : $\mathbb{Z} \to \mathbb{R}$, zwracającą wartość dyskretnego impulsu jednostkowego (delta Kroneckera) dla $n \in \mathbb{Z}$.

Problem 2.23: Zaimplementuj funkcję heaviside : $\mathbb{Z} \to \mathbb{R}$, zwracającą wartość dyskretnego skoku jednostkowego (funkcja skokowa Heaviside'a) dla $n \in \mathbb{Z}$.

Problem 2.24: Zaimplementuj funkcję rect : $\mathbb{N}^+ \to \mathbb{R}^N$, zwracającą wektor $x \in \mathbb{R}^N$ z próbkami dyskretnego okna prostokątnego o długości $N > 0$.

Problem 2.25: Zaimplementuj funkcję triang : $\mathbb{N}^+ \to \mathbb{R}^N$, zwracającą wektor $x \in \mathbb{R}^N$ z próbkami dyskretnego okna trójkątnego (Barletta) o długości $N > 0$.

Problem 2.26: Zaimplementuj funkcję hanning : $\mathbb{N}^+ \to \mathbb{R}^N$, zwracającą wektor $x \in \mathbb{R}^N$ z próbkami dyskretnego okna Hanninga o długości $N > 0$.

Problem 2.27: Zaimplementuj funkcję hamming : $\mathbb{N}^+ \to \mathbb{R}^N$, zwracającą wektor $x \in \mathbb{R}^N$ z próbkami dyskretnego okna Hamminga o długości $N > 0$.

Problem 2.28: Zaimplementuj funkcję blackman : $\mathbb{N}^+ \to \mathbb{R}^N$, zwracającą wektor $x \in \mathbb{R}^N$ z próbkami dyskretnego okna Blackmana o długości $N > 0$.

3. Parametry sygnałów

Problem 3.1: Zaimplementuj funkcję `mean`: $\mathbb{C}^N \to \mathbb{C}$ zwracającą wartość średnią dyskretnego sygnału $x \in \mathbb{C}^N$.

Problem 3.2: Zaimplementuj funkcję `peak2peak`: $\mathbb{R}^N \to \mathbb{R}$ zwracającą wartość międzyszczytową dyskretnego sygnału $x \in \mathbb{R}^N$.

Problem 3.3: Zaimplementuj funkcję `energy`: $\mathbb{C}^N \to \mathbb{R}$ zwracającą energię dyskretnego sygnału $x \in \mathbb{C}^N$.

Problem 3.4: Zaimplementuj funkcję `power`: $\mathbb{C}^N \to \mathbb{R}$ zwracającą moc dyskretnego sygnału $x \in \mathbb{C}^N$.

Problem 3.5: Zaimplementuj funkcję `rms`: $\mathbb{C}^N \to \mathbb{R}$ zwracającą wartość skuteczną dyskretnego sygnału $x \in \mathbb{C}^N$.

Problem 3.6: Zaimplementuj funkcję `running_mean`: $\mathbb{C}^N \times \mathbb{N}^+ \to \mathbb{C}^N$ zwracającą dyskretny sygnał średniej bierzącej dyskretnego sygnału $x \in \mathbb{C}^N$ z wycinków sygnału o szerokości $2M + 1$ próbek.

Problem 3.7: Zaimplementuj funkcję `running_energy`: $\mathbb{C}^N \times \mathbb{N}^+ \to \mathbb{R}^N$ zwracającą dyskretny sygnał energii bierzącej dyskretnego sygnału $x \in \mathbb{C}^N$ z wycinków sygnału o szerokości $2M + 1$ próbek.

Problem 3.8: Zaimplementuj funkcję `running_power`: $\mathbb{C}^N \times \mathbb{N}^+ \to \mathbb{R}^N$ zwracającą dyskretny sygnał mocy bierzącej dyskretnego sygnału $x \in \mathbb{C}^N$ z wycinków sygnału o szerokości $2M + 1$ próbek.

4. Próbkowanie

Problem 4.1: Dany jest sygnał $x(t) = \sin(200\pi t)$ z którego zostały pobrane próbki z częstotliwością $f_p = 250$ Hz. Znajdź wynikowy ciąg $x[n]$.

Problem 4.2: Dany jest sygnał $s(t) = \cos(\omega t), t \in \mathbb{R}$. W wyniku operacji próbkowania został otrzymany ciąg $s[n] = \cos(\gamma \pi n)$, gdzie $\gamma \in \mathbb{R}$ i $n \in \mathbb{Z}$. Znajdź ω przy założeniu że częstotliwość próbkownia wynosi $f_p \in \mathbb{R}^+$.

Problem 4.3: Określ czy dane poniżej ciągi są okresowe. Jeżeli tak, wyznacz okres.
a) $x[n] = \cos\left(\frac{3\pi}{7}n - \frac{\pi}{8}\right)$
b) $g[n] = \exp\left(j\left(\frac{n}{8} - \pi\right)\right)$

Problem 4.4: Dany jest ciągły sygnał $s(t) = g(t)\cos(1000\pi t)$ z którego zostały pobrane próbki $s[n] = s\left(\frac{n}{f_s}\right)$. Jeżeli $f_s > 1300$, to ciąg $s[n]$ jest jednoznaczy z ciągłym sygnałem $s(t)$, tzn. możliwa jest perfekcyjna rekonsturkcja sygnału ciągłego $s(t)$ na podstawie dyskretnych próbek $s[n]$. Jaki warunek należy postawić dla częstotliwości próbkowania f_g, aby ciąg $g[n] = g\left(\frac{n}{f_g}\right)$ był jednoznaczy z ciągłym sygnałem $g(t)$?

Problem 4.5: Zaimplementuj funkcję `interpolate`: $\mathbb{R}^N \times \mathbb{R}^N \times F_s \to F$, która zwróci funckje $f \in F$, gdzie F to zbiór funkcji $\mathbb{R} \to \mathbb{R}$, zwracającą wartość $f(x) = \hat{x}(t)$, gdzie $\hat{x}(t) \in \mathbb{R}$ to interpolacją ciągłego sygnału $x(t) \in \mathbb{R}$ w chwili $t \in \mathbb{R}$. Do interpolacji wykorzystaj metodę funkcji sklejnych z funkcją $k \in F_s$, gdzie F_s jest zbiorem wszystkich funkcji sklejnych $\mathbb{R} \to \mathbb{R}$. Węzły interpolacyjne są dane przez dwa wektory $\boldsymbol{m} \in \mathbb{R}^N$ i $\boldsymbol{s} \in \mathbb{R}^N$, gdzie $\boldsymbol{m} = [t_1, t_2, ..., t_N]$, a $\boldsymbol{s} = [x(t_1), x(t_2), ..., x(t_N)]$. Dodatkowo załóż że $t_{n+1} - t_n = \Delta t$.

Problem 4.6: Zbadaj jak kształtuje się średnią kwadratową błedu rekonstrukcji sygnału o ograniczonym paśmie w zależności od częstotliwości próbkowania. Do rekonstrukcji wykorzystaj metodę funkcji sklejnych oraz funckje sinc : $\mathbb{R} \to \mathbb{R}$ jako bazową funkcję sklejną. Wykreś uzyskane charakterystyki wyjaśnij krzywe.

Problem 4.7: Zbadaj błąd rekonstrukcji sygnału $x(t) = \cos\left(\pi f_p * t + \varphi\right)$ próbkowanego z tak zawaną krytyczną częstotliwością próbkowania $f_p \in \mathbb{R}$, w zależności od przesunięcia fazowego $\varphi \in [0, 2\pi]$. Do rekonstrukcji wykorzystaj metodę funkcji sklejnych oraz funckje sinc : $\mathbb{R} \to \mathbb{R}$ jako bazową funkcję sklejną. Wykreś badaną charakterystykę oraz uzasadnij jej przebieg.

5. Kwantyzacja

Problem 5.1: Zaimplementuj funkcję `quantize`: $\mathbb{R}^N \to F$, która zwróci funkcje $f_L \in F$, gdzie F to zbiór funkcji $\mathbb{R} \to \mathbb{R}$. Funkcja $f_L(x)$ powinna zwracać dla wartości $x \in \mathbb{R}$ poziom kwantyzacji $l' \in L$ poprzez rozwiązanie problemu

$$l' = \arg\min_{l \in L} \|x - l\|_2,$$

gdzie $L = [l_1, l_2, ..., l_N]^T \in \mathbb{R}^N$ zawiera poziomy kwantyzacji które spełniają warunek $l_i \neq l_j$ dla $i \neq j$.

Problem 5.2: Zaimplementuj funkcję `SQNR`: $\mathbb{N}^+ \to R$, która obliczy teoretyczny stosunek mocy sygnału do mocy szumu kwantyzacji wyrażony w decybelach dla idealnego $N \in \mathbb{N}^+$ bitowego przetwornika analogowo cyfrowego.

Problem 5.3: Zaimplementuj funkcję `SNR`: $\mathbb{R}^+ \times \mathbb{R}^+ \to R$, która obliczy stosunek mocy sygnału do mocy szumu wyrażony w decybelach.

Problem 5.4: Zbadaj zależność między teoretycznymi i rzeczywistymi wartościami SQNR dla róznych sygnałów.

6. Obliczanie dyskretnej transformacji Fouriera

Problem 6.1: Oblicz dwu punktową dyskretną transformacje Fouriera (2-DFT) sygnału $x \in \mathbb{R}^2$.

$$x = (20, 5)$$

Problem 6.2: Oblicz cztero punktową dyskretną transformacje Fouriera (4-DFT) sygnału $x \in \mathbb{R}^4$.

$$x = (3, 2, 5, 1)$$

Problem 6.3: Zaimplementuj funkcję `dtft`: $\mathbb{R} \times \mathbb{C}^N \times \mathbb{R}^+ \to \mathbb{C}$, która obliczy wartość ciągłego widma częstotliwosciowego dyskretnego sygnału $x \in \mathbb{C}^N$ dla częstotliwości $f \in \mathbb{R}$, przy złożeniu że próbki sygnału były pobrane z częstotliwością $f_s \in \mathbb{R}^+$. Do obliczenia tego przekształcenia wykorzystaj dyskretną w czasie transformację Fouriera oraz algorytm naiwny (tzn. bezpośrednio ze wzoru).

Problem 6.4: Zaimplementuj funkcję `dft`: $\mathbb{C}^N \to \mathbb{C}^N$ dla $N \in \mathbb{N}^+$, która przekształci dyskretny sygnał $x \in \mathbb{C}^N$ z domeny czasu do odpowiadającego mu dyskretnego sygnału $X \in \mathbb{C}^N$ w domenie częstotliwości (dwustronne dyskretne widmo częstotliwościowe). Do obliczenia tego przekształcenia wykorzystaj dyskretną transformację Fouriera oraz algorytm naiwny (tzn. bezpośrednio ze wzoru).

Problem 6.5: Zaimplementuj funkcję `idft`: $\mathbb{C}^N \to \mathbb{C}^N$ dla $N \in \mathbb{N}^+$, która przekształci dyskretny sygnał $X \in \mathbb{C}^N$ z domeny częstotliwości (dwustronne dyskretne widmo częstotliwościowe) do odpowiadającego mu dyskretnego sygnału $x \in \mathbb{C}^N$ w domenie czasu. Do obliczenia tego przekształcenia wykorzystaj odwrotną dyskretną transformację Fouriera oraz algorytm naiwny (tzn. bezpośrednio ze wzoru).

Problem 6.6: Zaimplementuj funkcję `rdft`: $\mathbb{R}^N \to \mathbb{C}^{\lfloor \frac{N}{2} \rfloor + 1}$ dla $N \in \mathbb{N}^+$, która przekształci dyskretny sygnał $x \in \mathbb{R}^N$ z domeny czasu do odpowiadającego mu dyskretnego sygnału $X \in \mathbb{C}^{\lfloor \frac{N}{2} \rfloor + 1}$ w domenie częstotliwości (jednostronne dyskretne widmo częstotliwościowe). Do obliczenia tego przekształcenia wykorzystaj dyskretną transformację Fouriera oraz algorytm naiwny (tzn. bezpośrednio ze wzoru).

Problem 6.7: Zaimplementuj funkcję `irdft`: $\mathbb{C}^{\lfloor \frac{N}{2} \rfloor + 1} \to \mathbb{R}^N$ dla $N \in \mathbb{N}^+$, która przekształci dyskretny sygnał $X \in \mathbb{C}^{\lfloor \frac{N}{2} \rfloor + 1}$ z domeny częstotliwości (jednostronne dyskretne widmo częstotliwościowe) do odpowiadającego mu dyskretnego sygnału $x \in \mathbb{R}^N$ w domenie czasu. Do obliczenia tego przekształcenia wykorzystaj odwrotną dyskretną transformację Fouriera oraz naiwny algorytm (tzn. bezpośrednio ze wzoru).

Problem 6.8: Wyprowadź algorytm Cooley–Tukey w wersji radix-2 do oblicznia N-punktowej dyskretnej transformacji Fouriera $\mathbb{C}^N \to \mathbb{C}^N$, gdzie $N = 2^K$ dla $K \in \mathbb{N}^+$. Wyprowadzenie zacznij od wzoru na dyskretną transformacje Fouriera.

Problem 6.9: Zaimplementuj funkcję `fft_radix2_dit_r`: $\mathbb{C}^{2^K} \to \mathbb{C}^{2^K}$ dla $K \in \mathbb{N}^+$, która przekształci dyskretny sygnał $x \in \mathbb{C}^{2^K}$ z domeny czasu do odpowiadającego mu dyskretnego sygnału $X \in \mathbb{C}^{2^K}$ w domenie częstotliwości (dwustronne dyskretne widmo częstotliwościowe). Do obliczenia tego przekształcenia wykorzystaj dyskretną transformację Fouriera oraz algorytm *Cooleya-Tukeya* w wersji radix-2 z decymacją w dziedzinie czasu. Obliczenia zaimplementuj z wykorzystaniem rekurencji.

Problem 6.10: Zaimplementuj funkcję `ifft_radix2_dif_r`: $\mathbb{C}^{2^K} \to \mathbb{C}^{2^K}$ dla $K \in \mathbb{N}^+$, która przekształci dyskretny sygnał $X \in \mathbb{C}^{2^K}$ z domeny częstotliwości (dwustronne dyskretne widmo

częstotliwościowe) do odpowiadającego mu dyskretnego sygnału $x \in \mathbb{C}^{2^K}$ w domenie czasu. Do obliczenia tego przekształcenia wykorzystaj odwrotną dyskretną transformację Fouriera oraz algorytm *Cooleya-Tukeya* w wersji radix-2 z decymacją w dziedzinie częstotliwości. Obliczenia zaimplementuj z wykorzystaniem rekurencji.

Problem 6.11: Wyprowadź algorytm Cooley–Tukey w wersji radix-3 do oblicznia N-punktowej dyskretnej transformacji Fouriera $\mathbb{C}^N \to \mathbb{C}^N$, gdzie $N = 3^K$ dla $K \in \mathbb{N}^+$. Wyprowadzenie zacznij od wzoru na dyskretną transformacje Fouriera.

Pytanie 6.1: Dla jakiej liczby $N \in \mathbb{N}^+$ nie istnieje algorytm radix-N o mniejszej złożonośćić obliczeniowej niż algorytm bezpośredni?

Problem 6.12: Wyprowadź algorytm Cooley–Tukey w wersji radix-M do oblicznia N-punktowej dyskretnej transformacji Fouriera $\mathbb{C}^N \to \mathbb{C}^N$, gdzie $N = M^K$ dla $K \in \mathbb{N}^+$. Wyprowadzenie zacznij od wzoru na dyskretną transformacje Fouriera.

Problem 6.13: Zaimplementuj funkcję `fft`: $\mathbb{C}^N \to \mathbb{C}^N$ dla $N \in \mathbb{N}^+$, która przekształci dyskretny sygnał $x \in \mathbb{C}^N$ z domeny czasu do odpowiadającego mu dyskretnego sygnału $X \in \mathbb{C}^N$ w domenie częstotliwości (dwustronne dyskretne widmo częstotliwościowe). Do obliczenia tego przekształcenia wykorzystaj dyskretną transformację Fouriera oraz połącz znane Ci algorytmy tak aby zaimplementowana funkcja miała jak najmniejszą złożoność obliczeniową.

Problem 6.14: Zaimplementuj funkcję `ifft`: $\mathbb{C}^N \to \mathbb{C}^N$ dla $N \in \mathbb{N}^+$, która przekształci dyskretny sygnał $X \in \mathbb{C}^N$ z domeny częstotliwości (dwustronne dyskretne widmo częstotliwościowe) do odpowiadającego mu dyskretnego sygnału $x \in \mathbb{C}^N$ w domenie czasu. Do obliczenia tego przekształcenia wykorzystaj odwrotną dyskretną transformację Fouriera oraz połącz znane Ci algorytmy tak aby zaimplementowana funkcja miała jak najmniejszą złożoność obliczeniową.

7. Analiza częstotliwościowa sygnałów dyskretnych

7.1. Problemy tablicowe

Problem 7.1.1: Niech wektor $\mathbf{x} \in \mathbb{R}^{5000}$ zawiera w sobie 5000 próbek pobranych z ciągłego sygnału $x(t) \in \mathbb{R}$. Odstęp czasu między dwiema kolejnymi próbkami wynosi $\Delta t = 50 \cdot 10^{-6}$. Następnie, korzystając z wektora $\mathbf{x}$ zostało obliczone widmo częstotliwościowe z wykorzystaniem 8192-punktowej dyskretnej transformaty Fouriera. Jaki będzie odstęp częstotliwościowy Δf między dwoma sąsiadującymi próbkami w wynikowej transformacie?

Problem 7.1.2: Ciągły sygnał $x(t) \in \mathbb{C}$ ma ograniczone pasmo od góry przez częstotliwość 5000, tzn. $X(j\omega) = 0, \ \forall \omega : |\omega| > 10000\pi$. Następnie z tego sygnału zostały pobrane próbki z okresem Δt, przez co powstał ciąg $x[n] = x(\Delta t \cdot n)$. W celu wykoniania analizy częstotliwościowej tego sygnału oblicznamy N-punktową dyskretną transformacji Foureira z N kolejnych wartości ciągu $x[n]$. Jako że potrzebujemy obliczyć w jak najkrótszym czasie wartości dyskretnej transformacji Fouriera, to wykorzystujemy do tego celu algorytm szybkiej transformacji Fouriera *radix-2*. Wyznacz minimalną wartość N oraz przedział do którego musi należeć częstotliwosc próbkowania, tak aby nie wystąpił aliasing oraz odstęp częstotliwościowy między próbkami (Δf) dyskretnego widma częstotliwościwego był mniejszy niż 5.

Problem 7.1.3: Sygnał $x(t) \in \mathbb{C}$ ma ograniczone pasmo przez częstotliwość 500 Hz, tzn. $X(j\omega) = 0, \ \forall \omega : |\omega| \geq 1000\pi$. Następnie z sygnału $x(t)$ zostają pobrane próbki, które tworzą następującą ciąg $x[n] = x(\Delta t \cdot n), \ n \in \mathbb{Z}$, gdzie $\Delta t = 0.001$ sekundy. Niech ciąg $X[k] \in \mathbb{C}$, $k = 0, ..., 1999$ będzie dyskretną transformatą Fouriera dowolnego podciągu $x[n]$ o długości 2000.

a) Z jaką ciągłą częstotliwością widma $X(j\omega)$ będzie korespondował prążek $X[k]$ dla $k = 0$?
b) Z jaką ciągłą częstotliwością widma $X(j\omega)$ będzie korespondował prążek $X[k]$ dla $k = 200$?
c) Z jaką ciągłą częstotliwością widma $X(j\omega)$ będzie korespondował prążek $X[k]$ dla $k = 1000$?
d) Z jaką ciągłą częstotliwością widma $X(j\omega)$ będzie korespondował prążek $X[k]$ dla $k = 1600$?

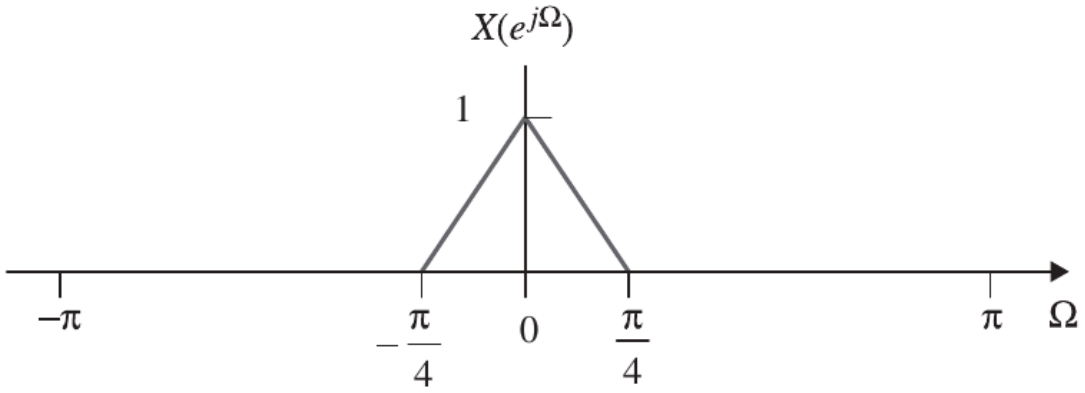
Problem 7.1.4: Dany jest dyskretny w czasie sygnał

$$x[n] = \cos\left(\frac{\pi}{3}n\right) + 2\sin\left(\frac{\pi}{4}n\right).$$

Przyjmij że częstotliwość próbkowania jest znormalizowana ($f_p = 1$). Wykreśl widmo dyskretnej w czasie transformacji Fouriera $X\left(e^{j\Omega}\right)$ w zakresie od $-\pi$ do π dla poniższych sygnałów. Przypisz wartości charakterystycznym punktom na wykresie.

a) $y[n] = x[n]\cos\left(\frac{\pi}{8}n\right)$
b) $y[n] = x[n]\exp\left(j\frac{\pi}{8}n\right)$
c) $y[n] = x[n]\cos\left(\frac{\pi}{2}n\right)$
d) $y[n] = x[n]\exp\left(j\frac{\pi}{2}n\right)$
e) $y[n] = x[n]\cos\left(\frac{7\pi}{8}n\right)$
f) $y[n] = x[n]\exp\left(j\frac{7\pi}{8}n\right)$

Problem 7.1.5: Na poniższym wykresie dane jest widmo amplitudowe dyskretnej w czasie transformacji Fouriera $X\left(e^{j\Omega}\right)$ pewnego sygnału $x[n]$.



Wykreśl widmo dyskretnej w czasie transformacji Fouriera w zakresie od $-\pi$ do π dla poniższych sygnałów. Przypisz wartości charakterystycznym punktom na wykresie.

a) $y[n] = x[n]\cos\left(\frac{\pi}{8}n\right)$
b) $y[n] = x[n]\exp\left(j\frac{\pi}{8}n\right)$
c) $y[n] = x[n]\cos\left(\frac{\pi}{2}n\right)$
d) $y[n] = x[n]\exp\left(j\frac{\pi}{2}n\right)$
e) $y[n] = x[n]\cos\left(\frac{7\pi}{8}n\right)$
f) $y[n] = x[n]\exp\left(j\frac{7\pi}{8}n\right)$

Problem 7.1.6: Rozważ sytuację w której szacujemy widmo amplitudowe dyskretngo w czasie sygnału $x[n]$ z wykorzystaniem dyskretnej transformacji Fouriera. Do analizy częstotliwościowej wykorzystane jest dyskretne okno Hamminga o długości $L = 2^k$, które jest nałożone na sygnał przed obliczeniem transformacji. Należy znaleźć minimalną długość okna (tzn. k), aby możliwe było rozróżnienie w widmie amplitudowym dwóch sygnałów harmonicznych, których częstotliwość różnią się od siebie zaledwie o $\frac{\pi}{100}$ w Ω (znormalizowana częstość kołowa).

Problem 7.1.7: Poniżej dane są trzy różne sygnały $x_{i[n]}$, które charakteryzują się tym że składają się sumy dwóch sygnałów harmonicznych.

$$x_1[n] = \cos\left(\frac{\pi}{4}n\right) \qquad + \cos\left(\frac{17\pi}{64}n\right)$$
$$x_2[n] = \cos\left(\frac{\pi}{4}n\right) \quad + 0.8\cos\left(\frac{21\pi}{64}n\right)$$
$$x_3[n] = \cos\left(\frac{\pi}{4}n\right) + 0.001\cos\left(\frac{21\pi}{64}n\right)$$

Na sygnały te zostaje nałożone okna prostokątne o długościu 64 próbek, a następnie zostaje oszacowane widmo częstotliwościowego z wykorzystaniem 64 punktowej dyskretnej transformacji Foureira. W których widmach amplitudowych, będą widoczne dwie wartości szczytowe, korespondujące z sygnałami harmonicznymi? Uzasadnij.

Problem 7.1.8: Niech $x[n] = \cos\left(\frac{2\pi}{5}n\right)$, a $v[n]$ będzie ciągiem który powstał po wymnożeniu ciągu $x[n]$ z 32 punktowym oknem prostokątnym. Wykreśl krzywą przedstawiającą w przybliżeniu widmo amplitudowe $\left|V\left(e^{j\Omega}\right)\right|$ na przedziale $-\pi < \Omega < \pi$. Oblicz częstotliwości w których występują maksymalne wartości listków tego widma amplitudowego, oraz częstotliwości miejsc zerwoych występujących przy listkach głównych. Zaznacz te częstotliwości na wykresie. Oblicz amplitudę lista głównego oraz pierwszego lista bocznego wykorzystanego okna, oraz zaznacz punkty przyjmujące te wartości na wykresie.

7.2. Problemy implementacyjne

Problem 7.2.1: Zaimplementuj funkcję `fftfreq`: $\mathbb{N}^+ \times \mathbb{R}^+ \to \mathbb{R}^N$, która obliczy wektor $f \in \mathbb{R}^N$ zawierający częstotliwości próbek dyskretnej transformaty Fouriera (dwustronne dyskretne widmo częstotliwościowe) obliczonej z dyskretnego sygnału o długości $N \in \mathbb{N}^+$ przy założeniu że próbki sygnału w domenie czasu zostały pobrane z częstotliwością $f_p \in \mathbb{R}^+$.

Problem 7.2.2: Zaimplementuj funkcję `rfftfreq`: $\mathbb{N}^+ \times \mathbb{R}^+ \to \mathbb{R}^{\lfloor \frac{N}{2} \rfloor + 1}$, która obliczy wektor $f \in \mathbb{R}^{\lfloor \frac{N}{2} \rfloor + 1}$ zawierający częstotliwości próbek dyskretnej transformaty Fouriera (jednostronne dyskretne widmo częstotliwościowe) obliczonej z dyskretnego sygnału o długości $N \in \mathbb{N}^+$ przy założeniu że próbki sygnału w domenie czasu zostały pobrane z częstotliwością $f_p \in \mathbb{R}^+$.

Problem 7.2.3: Zaimplementuj funkcję `amplitude_spectrum`: $\mathbb{C}^N \times \mathbb{R}^N \to \mathbb{R}^N$, która oblicz widmo amplitudowe $A \in \mathbb{R}^N$, dyskretnego sygnału $x \in \mathbb{C}^N$, który powstał poprzez nałożenie dyskretnego okna $w \in \mathbb{R}^N$ na ciąg $x[n] \in \mathbb{C}$.

Problem 7.2.4: Zaimplementuj funkcję `power_spectrum`: $\mathbb{C}^N \times \mathbb{R}^N \to \mathbb{R}^N$, która oblicz widmo mocy $P \in \mathbb{R}^N$ dyskretnego sygnału $x \in \mathbb{C}^N$, który powstał poprzez nałożenie dyskretnego okna $w \in \mathbb{R}^N$ na ciąg $x[n] \in \mathbb{C}$.

Problem 7.2.5: Zaimplementuj funkcję `psd`: $\mathbb{C}^N \times \mathbb{R}^N \times \mathbb{R}^+ \to \mathbb{R}^N$, która oblicz widmową gęstość mocy $G \in \mathbb{R}^N$ dyskretnego sygnału $x \in \mathbb{C}^N$, który powstał poprzez nałożenie dyskretnego okna $w \in \mathbb{R}^N$ na ciąg $x[n] \in \mathbb{C}$. Ciąg $x[n]$ powstał poprzez spróbkowanie ciagłego sygnału z szybkością $f_p \in \mathbb{R}^+$ próbke na sekundę.

Pytanie 7.2.1: Jak obliczyć moc sygnału $x \in \mathbb{C}^N$ na podstawie widma mocy $P \in \mathbb{C}^N$?

Pytanie 7.2.2: Jak obliczyć moc sygnału $x \in \mathbb{C}^N$ na podstawie widmowej gęstości mocy $G \in \mathbb{C}^N$?

Problem 7.2.6: Zaimplementuj funkcję `periodogram`: $\mathbb{C}^N \times \mathbb{R}^K \times \mathbb{N} \times \mathbb{R}^+ \to \mathbb{R}^K$, która oszacuje widmową gęstość mocy dyskretnego sygnału $x \in \mathbb{C}^N$, próbkowaniego z szybkościa $f_p \in \mathbb{R}^+$ próbek na sekundę, z wykorzystaniem metody Welcha. Do szacowania widmowej gęstości mocy, wykorzystaj wycinki sygnału uzyskane przez nakładanie dyskretnego okna czasowego $w \in \mathbb{R}^K$ na sygnał, gdzie $K \ll N$. Kolejne wycinki sygnału powinna nachodzić na siebie, na $L \in \mathbb{N}$ próbkach.

Problem 7.2.7: Dany jest dyskretny sygnał $x \in \mathbb{C}^N$ (FM.wav), który został pobrany przy pomocy radia programowalnego z szybkością 3 MHz w paśmie radiowym. Oszacuj widmową gęstość mocy tego sygnału i zidentyfikuj liczbę stacji radiowych oraz ich pasma. Stacje radiowe nadają sygnał, który jest zmodulowany częstotliwościowo (modulacja FM). Korzystając z dyskretnej transformaty Fouriera, wydziel pasmowy sygnał danej stacji radiowej, przesuń do pasma podstawowego (pasmo stacji radiowej powinno zostać przesuniętę na częstotliwość zero), a następnie użyj demodulacji kwadraturowej do odzyskania sygnału modulowanego.

DODATEK A: Wstęp do języka programowania Julia

Jest to krótki, ale powinien być to wystarczający, wstęp do języka programowania Julia na potrzeby tego laboratorium. Celem tego dodatku jest przedstawienie minimalnego podzbioru składni języka, aby można było jak najszybciej być w stanie sprawnie implementować programy, które będą realizować proste obliczenia numeryczne. Natomiast chciałbym też podkreślić, że nie jest to wstęp do programowania jako takiego, to już powinieneś umieć. Oficjalny podręcznik do języka Julia, który w miarę całościowo wprowadza do języka, można znaleźć tutaj, natomiast zawiera on zdecydowanie nadmiar informacji, niż jest to potrzebne. Jako lekturę dodatkową, przed przejściem dalej, można sobie pobieżnie przeglądnąć:

- Noteworthy Differences from other Languages.
- MATLAB–Python–Julia cheatsheet.

A.1 Zmienne

Zmienna, czyli nazwa (ciąg znaków), do której przyporządkowana jest jakaś wartość, można tworzyć poprzez operator przypisania = (znak równa się).

```
a1 = 1     # Przypisanie liczby całkowitej (typ Int)
b2 = 1.0   # Przypisanie liczby zmiennoprzecinkowej (typ Float64)
c3 = 'a'   # Przypisanie znaku (typ Char)
d4 = "abc" # Przypisanie ciągu znaków (typ String)
```

Reguły dotyczące nazw zmiennych są bardzo podobne jak w innych popularnych językach programowania, więc nie będziemy się tutaj zbytnio w to zagłębiać. Przykładowo nazwa zmiennej nie może zaczynać się od cyfry, ale może zaczynać się od znaku podkreślenia.

```
4zmienna = 1 # Błąd skladni
_zmienna = 1 # Brak błędu składni
```

W nazwie zmiennej mogą występować znaki Unicode, jak na przykład znaki alfabetu greckiego.

```
_α = 1.0
d_β = 2.0
Γ_1 = 3.0
```

W REPL (Read-Eval-Print-Loop) i edytorach tekstu, które wspierają programowanie w języku Julia, znaki takie można szybko wprowadzać poprzez wpisanie `\kod_znaku` i naciśnięcie klawisza tabulacji.

```
\pi<TAB>
\beta<TAB>
\Gamma<TAB>
```

Spis wszystkich takich znaków można znaleźć w rozdziale dokumentacji Julia Unicode input. Niektóre symbole, jak na przykład π, mogą mieć specjalnie znacznie. Pamiętaj, że każda zmienna ma jakiś typ. Żeby sprawdzić typ zmiennej, można skorzystać z funkcji `typeof`, co może przydać się czasem podczas analizy napisanego kodu.

```

```

```
typeof(a1) # Int
typeof(b2) # Float64
typeof(c3) # Char
typeof(d4) # String
```

Przydane rozdziały z oficjalnego podręcznka języka programowania Julia:
- Variables

A.2 Liczby

W zasadzie powinno się wiedzieć dwie rzeczy o liczbach: typy i operacje. Najbardziej istotnymi dla nas typami numerycznymi są:

- Typ `Int` (a właściwe `Int64`) reprezentujący liczbę całkowitą: `1`, `2`, `10000`, `10_000`.
- Typ `Float64` reprezentujący liczbę zmiennoprzecinkową: `1.0`, `2.0`, `1e-3`, `1e+6`.
- Typ `ComplexF64` reprezentujący zespoloną liczba zmiennoprzecinkową: `1.0 + im*2.0`.

W bibliotece standardowej istnieją oczywiście też inne typy numeryczne (patrz tutaj), natomiast nie są one raczej w centrum naszego zainteresowania, przynajmniej, póki co.

Co można robić z liczbami? Wykonywać na nich działania arytmetyczne.

Wyrażenie	Operacja	Opis
`x + y`	dodawanie	dodanie x do y
`x - y`	odejmowanie	odjęcie y od x
`x * y`	mnożenie	przemonżenie x przez y
`x / y`	dzielenie	dzielenie x przez y
`x ÷ y`	dzielenie bez reszty	równoważne z funckja `div(x,y)` (`\div<TAB>` -> ÷)
`x ^ y`	potęgowanie	podnieś x do potęgi y
`x % y`	reszta z dzielenia	równoważne z funckją `rem(x,y)`

W bibliotece standardowej są zdefiniowane pewne istotne stałe numeryczne, takie jak,
- Liczba urojona $i^2 = -1$, kryje pod zmienną `im`.
- Liczba $\pi = 3.1415926535897...$, kryje się pod zmienną `pi` oraz `π` (`\pi<TAB>` -> π).

Przydatne funkcje dla liczb zespolonych:
- `abs(z)` - moduł liczby zespolonej `z`.
- `angle(z)` - argument (faza) liczby zespolonej `z`.
- `real(z)` - część rzeczywista liczby zespolonej `z`.
- `imag(z)` - część urojona liczby zespolonej `z`.

Przydane rozdziały z oficjalnego podręcznka języka programowania Julia:
- Integers and Floating-Point Numbers
- Complex and Rational Numbers
- Mathematical Operations and Elementary Functions

A.3 Wartości logiczne

Typ `Bool` to typ reprezentujący binarną wartość logiczną, która może przyjąć wartość prawda (`true`), bądź fałsz (`false`). Głównie ma on zastosowanie przy instrukcjach warunkowych, o których będzie za chwile. Podstawowe operacje na wartościach logicznych, które można wykorzystać do budowania wyrażeń logicznych to:

Wyrażenie	Operacja
`!x`	negacja
`x && y`	koniunkcja
`x || y`	alternatywa

Przykład wyrażenia logicznego:

```
!(true && false) || true
```

Wyrażenia logicznie najczęściej buduje się z wykorzystaniem operacji porównywania dwóch wartości.

Wyrażenie	Opis
`x == y`	prawda jeżeli x jest równe y
`x != y`	prawda jeżeli x jest różne od y
`x < y`	prawda jeżeli x jest mniejsze niż y
`x <= y`	prawda jeżeli x jest mniejsze niż lub równe y
`x > y`	prawda jeżeli x jest większe niż y
`x >= y`	prawda jeżeli x jest większe niż lub równe y

```
a = 1
b = 2.0
c = 1.0

!(a == c) && (a > b) || !(a >= b) || (a < b) && !((a <= b) && !(a <= c))
```

Kolejność wykonywania działań logicznych: negacja, koniunkcja, alternatywa. Zawsze można też skorzystać z nawiasów, aby wymusić pożądaną kolejność.

A.4 Sterowanie przepływem

Sterowanie przepływem, na potrzeby tego wstępu, to zbiór instrukcji, które pozwalają na warunkowe wykonywanie lub powtarzanie bloków instrukcji. Interesują nas tutaj w zasadzie dwie rzeczy: instrukcje warunkowe oraz pętle. Omówimy sobie dwie instrukcje warunkowe: `if` i `?:` (ternary operator), oraz dwa pętle: `while` i `for`.

Operator warunkowy `?:` ma następującą składnię:

```
warunek ? "zwróc wartość jeżeli prawda" : "zwróc wartość jeżeli fałsz"
```

Parę przykładów wykorzystania tego operatora:

```
1+1 == 2 ? "prawda" : "fałsz" # Zwraca "prawda"
1+1 == 3 ? "prawda" : "fałsz" # Zwraca "fałsz"

x = 0.5
x = x > 1.0 ? x^2 : 2x
```

Druga instrukcja warunkowa, to instrukcja `if`, która daję odrobinę więcej możliwości. Instrukcji `if` mogą akompaniować, o ile jest taka potrzeba, również instrukcje `elseif` i `else`. Składnia jest standardowa, nie ma co tutaj za bardzo się rozwodzić, więc przejdźmy od razu do trzech przykładów, które pokazują trzy warianty wykorzystania tych instrukcji.

```
a = 1
if _WARUNEK # Jeżeli _WARUNEK jest prawdziwy to wykonaj ten blok kodu
  a += 1
end
@show a
```

```
a = 1
if _WARUNEK # Jeżeli _WARUNEK jest prawdziwy to wykonaj ten blok kodu
  a += 1
else # Jeżeli _WARUNEK jest fałszwy to wykonaj ten blok kodu
  a -= 1
end
@show a
```

```
a = 1
if _WARUNEK # Jeżeli _WARUNEK jest prawdziwy to wykonaj ten blok kodu
  a += 1
elseif _WARUNEK2 # Jeżeli poprzednie warunki są fałszye,
  a *= 10        # ale ten nie, to wykonaj ten blok kodu
else # Jeżeli wszystkie warunki są fałszwy to wykonaj ten blok kodu
  a -= 1
end
@show a
```

Jeżeli jest taka potrzeba, to oczywiście można umieścić więcej niż jedną instrukcję `elseif` w ramach takiego wyrażenia.

Pętla `while` służy do warunkowego powtarzania bloku instrukcji. Instrukcje są powtarzane, dopóki warunek jest spełniony. Warunek jest sprawdzany przed każdym wykonaniem bloku instrukcji. Przykład:

```
i = 0
while i < 10
```

```
  @show i
  i += 1
end
```

Pętla `for` służy głównie do iterowania po iterowalnych strukturach i wykonania bloku instrukcji dla aktualnie wygenerowanego elementu przez iterator. Dla zobrazowania przykładową strukturą, po której możliwa jest iteracja, będzie tablica `[3, 1, 4]`, natomiast do omówienia tablic przejdziemy za niedługo. Przykładowe wykorzystane pętli `for` poniżej.

```
a = 1
for i in 1:10
  a += i
  @show i, a
end
```

Skupmy się teraz na wyrażeniu `1:10`, które to tworzy iterowalną strukturę `UnitRange{Int64}`, która zwraca kolejne liczby od 1 do 10 gdy się po niej iteruje. W ogólności wyrażenie `start:krok:stop` generuje iterator, który będzie zwracał wartości wypisane przez poniższy kod:

```
i = start
while i <= stop
    @show i
    i += krok
end
```

Jeżeli w wyrażeniu `start:krok:stop` nie podamy wartości `krok`, czyli wywołamy wyrażenie `start:stop`, to wartość domyślna `krok = 1`. Do generowania takich iteratorów istnieje także funkcja `range`, która może być czasem wygodniejsza niż wyrażenie `start:krok:stop`. Zapoznaj się z dokumentacją tej funkcji poprzez wpisanie w REPL `?range`, i sam oceń.

Oczywiście dostępne są także dwie specjalne instrukcje `break` oraz `continue`, które to można umieścić w pętli. Dla przypomnienia wywołanie instrukcji `break` przerwie wykonywanie pętli, natomiast wywołanie instrukcji `continue` wywołuje przejście do następnej iteracji pętli.

Inne przydane funkcje dla pętli `for`, z którymi polecam się zapoznać to `enumerate` oraz `zip`.

Przydane rozdziały z oficjalnego podręcznka języka programowania Julia:
- Control flow

A.5 Funkcje

W języku programowania Julia, funkcje możemy definiować na dwa sposoby. Jako że każdy z tych sposób może być przydatny, to omówmy sobie oba. Zanim przejdziemy sobie do omówienia składni, to nadmienie, że argumenty przekazywane do funkcji są zawsze przekazywane przez wartość. Zacznijmy od omówienia sobie najbardziej ogólnego przypadku na poniższym przykładzie.

```
function foo(a, b, c=10.0; klucz1, klucz2=1)
  return klucz1 > klucz2 ? a*b*c : a+b+c
end
```

Definicje funkcji zaczynamy od słowa kluczowego `function`. Następnie powinna pojawić się nazwa funkcji, w przykładzie jest to `foo`. W nawiasach podajemy argumenty, które funkcja przyjmuje. W przykładzie mamy trzy argumenty pozycyjne: `a`, `b`, `c` oraz dwa argumenty przez słowo kluczowe: `klucz1`, `klucz2`. Argumenty przez słowo kluczowe są oddzielone od argumentów pozycyjnych znakiem `;`. Różnica między tymi dwoma rodzajami argumentów jest podczas wywołania funkcji. Argumenty przez słowo kluczowe trzeba nazwać podczas wywołania funkcji i kolejność ich podawania nie ma znaczenia. W przypadku argumentów pozycyjnych nie podajemy ich nazw i kolejność ma znaczenie. Dodatkowo każdy argument może przyjąć wartość domyślną poprzez operator przypisania wartości. Co przekłada się na to że jeżeli nie podamy danego argumentu podczas wywołania, to przyjmuje on w ten czas domyślną wartość (o ile jest podana). Wartość zwracana przez funkcje jest wskazana przez instrukcję `return`. Przeanalizuj poniższe przykłady wywołania funkcji `foo`.

```
foo(1, 2, 3; klucz1=10, klucz2=20)
foo(1, 2; klucz1=10)
```

Akurat przykładowa funkcja jest dość prosta i można wykorzystać tutaj drugi (jedno linijkowy) sposób definiowania funkcji, który jest następujący:

```
bar(a, b, c=10.0; klucz1, klucz2=1) = klucz1 > klucz2 ? a + b * c : a * b + c
```

Istnieje możliwość jawnej specyfikacji typów dla argumentów oraz wartości zwracanej przez funkcje. Składnia jest jak w przykładzie poniżej, natomiast nie trzeba tego robić.

```
suma_int(a::Int, b::Int)::Int = a + b
```

Przydane rozdziały z oficjalnego podręcznka języka programowania Julia:
- Functions

A.6 Tablice

Tablica bądź lista, to taka struktura danych, która reprezentuje skończony i uporządkowanego zbioru elementów, który można modyfikować (ang. *mutable*). Elementami tablicy w zasadzie może być dowolna wartość, natomiast z punktu widzenia obliczeń numerycznych, interesują nas głównie tablice, które zawierają w sobie liczby. Aby stworzyć tablicę, trzeba zbudować wyrażenie z wykorzystaniem nawiasów kwadratowych, przykładowo:

```
julia> a = [1, 2, 3]
> 3-element Vector{Int64}:
 1
 2
 3
```

W rezultacie otrzymaliśmy coś, co jest typu `Vector{Int64}`. I tak typ `Vector` to jest właśnie jednowymiarowa tablica, a w zasadzie alias na parametryczny typ `Array`.

```
julia> Vector
> Vector (alias for Array{T, 1} where T)
```

Typ `Int64`, który pojawił się w nawiasach `{}` mówi jakiego typu elementy przechowuje ten `Vector`. Nazwa `Vector` nie jest przypadkowa i wrócimy do omówienia tego później. Natomiast na początku skupmy się na jedno wymiarowych tablicach.

Jeżeli w wyrażeniu znajdą się literały o różnych typach, to interpreter spróbuje rzutować je do wspólnego typu (o ile jest to możliwe).

```
julia> b = [1.0, 2, 3]
> 3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

Jeżeli nie jest to możliwe, to powstanie tablica przechowująca elementy typu `Any`, który oznacza co do zasady dowolny typ.

```
julia> b = [1.0, 2, 3, "tekst"]
> 4-element Vector{Any}:
 1.0
 2
 3
  "tekst"
```

Jak stworzyć pustą jednowymiarową tablicę? Przykładowo w taki sposób:

```
a = []           # Pusta tablica przechowująca elementy o dowolnym typie `Any`
b = Any[]        # To samo co `a = []`, tylko bardziej jawnie
c = Int[]        # Pusta tablica przechowująca elementy o typie `Int`
d = Float64[]    # Pusta tablica przechowująca elementy o typie `Float64`
e = ComplexF64[] # Pusta tablica przechowująca elementy o typie `ComplexF64`
```

Podstawowe operacje na jednowymiarowych tablicach:

```
push!(a, 3)        # Dodaj na koniec tablcy wartość `3`
last = pop!(a)     # Usuń ostatni element tablicy i zwróc go
append!(a, [4, 5]) # Do tablicy `a` doklej tablice `[4, 5]`
length(a)          # Zwróć ilość elementów w tablicy (nie tylko jednowymiarowej)
```

W języku Julia istnieje konwencja w której to funkcje których nazwa kończy się znakiem `!`, modyfikują argument funkcji. Przeważnie modyfikowany jest w ten czas pierwszy argument pozycyjny.

Aby dostać się bezpośrednio to i-tego elementu tablicy, można wykorzystać nawiasy kwadratowe.

```
a = [1, 2, 3, 4]
a[2] # Zwraca drugi element tablicy a
a[2] = 10 # Przypisanie do drugiego elementu nowej wartości
```

W języku programowania Julia pierwszy element tablicy (czy w ogólności uporządkowanych kolekcji) ma indeks 1. Więc specyfikujemy pozycję w tablicy, a nie przesunięcie względem pierwszego elementu jak np. w języku C. Jeżeli potrzebujesz operować na przesunięciach, to zawsze możesz napisać wyrażenie w następujący sposób `a[1+i]`, gdzie i to przesunięcie.

Jeżeli chcemy wybrać podzbiór elementów z tablicy, to można skorzystać z iteratora `start:krok:stop` albo wyspecyfikować indeksy przez tablicę zawierającą dodatnie liczby całkowite. Zapoznaj się z poniższymi przykładami.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
a[1:2:9]        # Zwróci elementy o nieparzystych indeksach
a[begin:2:end]
a[3:7]          # Zwróć wycinek tablicy
a[9:-1:1]       # Zwróci tablice w odwróconej kolejności
a[end:-1:begin] # Tak też zadziała
a[[3, 8]]       # Zwrócy 3 i 8 element tablicy
```

No i tyle o tablicach jednowymiarowych. Przejdźmy do tablic wielowymiarowych.

Tablice dwuwymiarowe można stworzyć następującą składnią:

```
A = [1 2; 3 4]
```

W rezultacie, jak może się domyślić, powstał macierz, czyli typ , będący także aliasem na parametryczny typ `Array`.

```
julia> Matrix
> Matrix (alias for Array{T, 2} where T)
```

Dla tablic co najmniej trzy wymiarowych nie ma już specjalnych aliasów i tworzenie ich w taki sposób jest już wysoce niepraktyczne. Przeważnie korzysta się w ten czas z funkcji do alokacji tablicy o danym rozmiarze jak `zeros`. Natomiast omówmy sobie składnie tego wyrażenia. Elementy w wierszach oddzielamy odstępem (spacja), natomiast kolejne wiersze oddzielamy średnikiem (;). Możemy w tym stylu zbudować też macierz z wektorów. Przeanalizuj, co generuje poniższy kod.

```
a = [3, 2, 1]

A = [a a]
b = [a; a]
C = [A; b b]
```

Zasady dostępu do elementów z tablic wielowymiarowych są analogiczne jak w przypadku tablic jednowymiarowych. Jedyna różnica jest podanie indeksu (bądź iteratora) dla kolejnych wymiarów.

```
A = [1 2 3; 4 5 6; 7 8 9]
A[1,3] # Który element zostanie zwróci?
A[1:2, 2:3] # Jaka macierz zostanie zwrócona?
A[[1,3], [2, 3]] # A tutaj czego się spodziewasz?
```

No dobra, to, w jakim celu jednowymiarowa tablica jest typu `Vector`, a nie po prostu `Array`? Powód jest prosty, zmienne typu `Vector` traktowane są jako wektory, a oznacza to, że co do zasady można na nich wykonywać podstawowe działania arytmetyczne znane z algebry liniowej.

```
a = [3, 2, 1]
b = [1, 2, 3]
```

```
c = a'        # Sprzężenie hermitowskie (transpzycja i sprzężenie zespolone)
d = 3*a + 2*b # Liniowa kombinajca wektorów
e = a' * b    # Iloczny skalarny dwóch wektorów
f = a  * b'   # Iloczny zewnętrzny dwóch wektorów
```

Podobnie jest w przypadku tablic dwuwymiarowych, które są traktowane jako macierze.

```
A = [1 2 3; 4 5 6; 7 8 9]
b = [1, 2, 3]
c = A*b       # Przekształcenie liniowe
d = b'*A*b    # Forma kwadratowa
e = A' * A
```

Natomiast błąd zwróci takie wyrażenie jak `[1, 2, 3] + 3`, bo nie da się dodać wartości skalarnej i wektora, tak jak nie da się dodać macierzy i wektora. Można to rozwiązać przez mechanizm rozgłaszania, o którym będzie krótko w następnej sekcji. Zapoznaj się, co robią następujące funkcje, bo pewnie się przydadzą:

```
zeros(4)
zeros(ComplexF64, 6, 4)
zeros(ComplexF64, (5, 10))

size(zeros(4, 8))
length(zeros(4, 8))
reshape(zeros(ComplexF64, 4, 8), 2, 16)

eltype(zeros(4, 8))
eltype(zeros(ComplexF64, 4, 8))

ones(4)
ones(ComplexF64, 4, 4)
ones(ComplexF64, (4, 4))

rand(4)
rand(ComplexF64, 4, 4)
rand(ComplexF64, (4, 4))

randn(4)
randn(ComplexF64, 4, 4)
randn(ComplexF64, (4, 4))
```

Przydane rozdziały z oficjalnego podręcznka języka programowania Julia:
- Single- and multi-dimensional Arrays

A.7 Rozgłaszanie

Rozgłaszanie (ang. broadcasting) to prosty mechanizm, który ma za zadanie rozwiązać następujący problem. Jeżeli mam funkcję `abs(z)`, która oblicza moduł liczby zespolonej `z`, a chciałbym obliczyć moduł dla każdej liczby zespolonej w tablicy `zt = randn(ComplexF64, 100)`, to jak to mogę zrobić? Na przykład pętlą `for`, ale wygodniejszym rozwiązaniem jest rozgłoszenie funkcji `abs` na elementy tablicy `zt` w następujący sposób `abs.(zt)`. Symbol `.` (kropka) jest operatorem rozgłaszania. Więc jeżeli chcemy dodać skalara do wektora, to musimy rozgłosić skalar na elementy wektora.

```
a = [1, 2, 3]
b = a .+ 1
```

Dwóch wektorów kolumnowych nie da się przez siebie przemnożyć, natomiast możemy przemnożyć przez siebie odpowiednie elementy.

```
a = [1, 2, 3]
b = a * a  # Błąd
b = a .* a # Sukces
```

Sam mechanizm rozgłaszania jest trochę bardziej skomplikowany, niż zostało to tutaj przedstawione, natomiast takie zastosowanie rozgłaszania powinno wystarczyć na potrzeby implementacji prostych obliczeń.

A.8 Krotka

Krotka (ang. *tuple*), to taka jednowymiarowa lista, tyle że jak się ją stworzy, to nie można zmieniać wartość jej elementów (ang. *immutable type*). Aby stworzyć krotkę, trzeba wykorzystać nawiasy okrągłe. Przykład

```
krotka = (1, 2, 3)   # Stwórz krotkę i przypisz ją do zmiennej
krotka[1]            # Dobierz się do pierwszego elemetu
a, b, c = krotka     # Można łatwo i szybko rozpakować sobie krotkę
@show typeof(krotka) # Zauważ że typ krotki jest bardzo konkretny
krotka[2] = 55       # Nie można tego zrobić
```

Krotki występują przy opazji zwracania wielu wartości z funkcji.

```
function foobar(a, b)
  c = a*b
  d = a^b
  return c, d # Tutaj zwracana jest krotka, mimo że nie ma nawiasów
end

prod, power = foobar(10, 4) # A tutaj krotka jest od razu rozpakowana
```

Czy też podczas iterowania pętlą `for` z wykorzystaniem funkcji `enumerate`

```
for (i, x) in enumerate([11, 22, 33, 44])
  @show i
  @show x
end
```

A.9 Przydate funkcje z biblioteki standardowej

```
N, R = 10, 3
a, b, c, d = 1.9, 2.7, 3.5, 4.3
z = 1.0 + im*1.0
T = [a, b, c, d]
```

```
# Operacja na liczbach całkowitych
div(N, R)    # Dzielenie całkowite
rem(N, R)    # Reszta z dzielenia
divrem(N, R) # Dzielenie i reszta za jednym razem

# Podstawowe funkcje matematyczne
sqrt(d)  # Pierwiastek kwadratowy liczby x
exp(d)   # Eksponenta liczy x
cos(d)   # Cosinus kąta x (rad)
sin(d)   # Sinus kąta x (rad)
sinc(d)  # Funkcja sinc z liczby x
log(d)   # Logarytm naturalny z liczby x
log2(d)  # Logarytm binarny z liczby x
log10(d) # Logarytm dziesiętny z liczby x

abs(z)   # Moduł liczby zespolone
angle(z) # Argument liczby zespolonej
real(z)  # Część rzeczywista
imag(z)  # Część urojona

floor(Int, d) # Podłoga z liczby x (zrzutowany na typ Int)
ceil(Int, d)  # Sufit z liczby x (zrzutowany na typ Int)

max(a, b, c, d) # Wartość maksymalna argumentów
min(a, b, c, d) # Wartość minimalna argumentów

# Funkcje dla tablic
length(T)   # Ilość elementów w tablicy
size(T)     # Wymiar tablicy
sum(T)      # Suma wszystkich elementów
maximum(T)  # Maksymalna wartość w tablicy
minimum(T)  # Minimalna wartość w tablicy
argmax(T)   # Indeks maksymalne wartość w tablicy
argmin(T)   # Indeks minimalne wartość w tablicy

# Alokowanie tablicy o zadanym wymiarze
# Wymiarów można dodawać w opór. Wymiar można alternatywnie podać w postaci krotki.
# Domyślnie typ Float64, można też podać inny jako pierwszy argument.
zeros(N)                    # Wektor o długości N o typie Float64
zeros(N, N)                 # Macierz o wymiarze N x N o typie Float64
zeros(N, N, N)              # Tablica o wymiarze N x N x N o typie Float64
zeros(ComplexF64, N)        # Wektor o długości N o typie ComplexF64
zeros(ComplexF64, N, N)     # Macierz o wymiarze N x N o typie ComplexF64
zeros(ComplexF64, N, N, N)  # Tablica o wymiarze N x N x N o typie ComplexF64

# Iterator
start, krok, stop = 10, 0.1, 100
start:stop
start:krok:stop
range(start, stop; step=krok)
range(start, stop; length=100)
```

A.10 Ciągi znakowe

Do reprezentacji ciągów znakowych służy niezmienny typ `String`.

```
# Stwórz pusty ciąg znakowy
a = ""
@show a

# Przyłącz do ciągu `a` drugi ciąg znaków
a = a * "Już " * "nie taki pusty"
@show a

# Wypisz podciąg
@show a[3:10]
```

A.11 Wypisywanie na standardowe wyjście

```
a = "to jest"
b = "przykład"
x = randn(4, 4)

# Wypisz podane wartości
print(a, b, x)

# Wypisz podane wartości ze znakiem nowej linii na końcu
println(a, b, x)

# Wyświelt ładnie wartość (tylko jedne argument)
display(x)

# Wypisywanie wartości w ustrukturyzowany sposób
@show a b x
```

A.12 Wykresy

Istnieje co najmniej parę sensowych bibliotek do generowania wykresów w ekosystemie Julii, natomiast skupimy się wyłącznie na Makie. Makie jest wysokopoziomową biblioteką, która co do zasady definiuje ujednolicony interfejs programistyczny między różnymi silnikami renderującymi. My zainteresujemy się silnikiem *Cairo* (grafika wektorowa) i *OpenGL*. Moduł Makie dla silnik renderującego *Cairo* to `CaiorMakie.jl`, natomiast dla silnika renderującego *OpenGL* to `GLMakie.jl`. Aby zainstalować paczki wpisz w REPL:

```
] add CairoMakie GLMakie
```

i jak się skonczy proces instalacji (kompilacji), przyciśnij klawisz *backspace*, aby wyjść z menadzera paczek. Aby załadować paczkę `CairoMakie.jl` do użytku, należy wywołać instrukcję `using CairoMakie`. `CairoMakie.jl` jest wspierane przez rozszerzenie `julialang` w VSCode i będzie automatycznie wyświetlało wygenerowane obrazki w okienku, w VSCode. Podstawy Makie, które będą wystarczające na potrzeby labów, omówione są w Makie.org - Basic Tutorial. Proszę się zapoznać z tym materiałem. Poniżej parę przepisów jak rysować wykresy w Makie.

```
using CairoMakie

fs = 100
dt = 1/fs
t_1, t_2 = 0.0, 1.0
t = range(t_1, t_2; step=dt)
x = sin.(2*pi*10*t)

lines(t, x)
```

```
using CairoMakie

fs = 256
dt = 1/fs
t_1, t_2 = 0.0, 1.0
t = range(t_1, t_2; step=dt)
x = sin.(2*pi*10*t)

scatter(t, x)
```

```
using CairoMakie

fs = 256
dt = 1/fs
t_1, t_2 = 0.0, 1.0
t = range(t_1, t_2; step=dt)
x = sin.(2*pi*10*t)

lines(t, x)
scatter!(t, x)
xlims!(0.2, 0.3)
ylims!(-2.2, 2.2)
current_figure()
```

```
using CairoMakie

fs = 256
dt = 1/fs
t_1, t_2 = 0.0, 1.0
t = range(t_1, t_2; step=dt)
x = sin.(2*pi*10*t)

fig = Figure(size=(800, 400))
ax1 = Axis(fig[1,1], aspect=1)
ax2 = Axis(fig[1,2], aspect=1)
lines!(ax1, t, x)
scatter!(ax2, t, x)
current_figure()
```

```
using CairoMakie

```

```
fs = 256
dt = 1/fs
t_1, t_2 = 0.0, 1.0
t = range(t_1, t_2; step=dt)
x = sin.(2*pi*10*t)

fig = Figure(size=(800, 400))
ax1 = Axis(fig[1,1], aspect=1)
ax2 = Axis(fig[1,2], aspect=1)

lines!(ax1, t, x; label="sinus kreski")
scatter!(ax2, t, x; label="sinus kropki")

axislegend(ax1)
axislegend(ax2)

xlims!(ax1, 0.2, 0.3)
ylims!(ax1, -2.2, 2.2)

xlims!(ax2, 0.2, 0.3)
ylims!(ax2, -2.2, 2.2)
current_figure()
```

```
using CairoMakie

x = 0:10
y = 10:40

fig = Figure(size=(800, 400))
ax = Axis(fig[1,1], aspect=1)

heatmap!(ax, x, y, rand(length(x),length(y)))
current_figure()
```