Obliczenia Naukowe - L1

Piotr Maciejończyk

17 października 2023

1 Rozpoznanie arytmetyki

1.1 Epsilon maszynowy

Należało znaleźć tzw. epsilon maszynowy - najmniejszą liczbę $macheps > 0$, taką że:

$$fl(1.0 + macheps) > 1.0 \textbf{ oraz } fl(1.0 + macheps) = 1 + macheps$$

Wartości epsilonu maszynowego dla typów zmiennopozycyjnych w języku Julia otrzymałem na dwa sposoby:

a) Korzystając z wbudowanej funkcji *eps()*.
b) Metodą iteracyjną **(plik ex-1.1.jl)**.

W języku C natomiast, znalazłem wartości dla odpowiednio *Float32*(float) oraz *Float64*(double) w pliku nagłówkowym **float.h**. Oto przedstawienie moich wyników:

macheps	*eps()*	iteracyjnie	float.h
Float16	0.000977	0.000977	———
Float32	$1.1920929 \cdot 10^{-7}$	$1.1920929 \cdot 10^{-7}$	$1.192093 \cdot 10^{-7}$
Float64	$2.220446049250313 \cdot 10^{-16}$	$2.220446049250313 \cdot 10^{-16}$	$2.220446 \cdot 10^{-16}$

Jak można zauważyć powyżej, metoda iteracyjna wyliczająca epsilon maszynowy dla typów zmiennopozycyjnych jest bardzo precyzyjna i zwraca odpowiednie wartości.

Natomiast, jaki dokładnie związek ma liczba macheps z precyzją arytmetyki ϵ?

Precyzja arytmetyki (ϵ) jest górnym oszacowaniem błędu względnego zaokrąglenia liczby i w standardzie IEEE 754 przyjmuje formę:

$$\epsilon = 2^{-t} \geq \frac{|rd(x)-x|}{|x|} = |\delta|,$$

gdzie: t - liczba bitów mantysy (precyzja), x - liczba, której błąd względny chcemy ocenić.

Epsilon maszynowy (*macheps*) jest najmniejszą możliwą dodatnią liczbą większą od 1. Zgodnie ze standardem IEEE 754, na część ułamkową mantysy każdej liczby rzeczywistej jest przeznaczone $t-1$ bitów. Zatem, aby otrzymać najmniejszą liczbę większą od 1.0, trzeba zwiększyć 1.0 o dokładnie $2^{-(t-1)}$, co jest szukaną przez nas wartością *macheps*. Można zatem zauważyć, że:

$$macheps = 2^{-(t-1)} = 2 \cdot 2^{-t} = 2 \cdot \epsilon$$

1.2 Maszynowe η

Zadaniem było iteracyjnie wyznaczyć dla wszystkich typów zmiennopozycyjnych liczbę maszynową η, czyli najmniejszą liczbę większą od 0 w danej arytmetyce *fl*.

Wartości epsilonu maszynowego dla typów zmiennopozycyjnych w języku Julia otrzymałem na dwa sposoby:

a) Korzystając z wbudowanej funkcji *nextfloat(T(0.0))* (T - typ zmiennopozycyjny).
b) Metodą iteracyjną szukając najmniejszej dodatniej liczby, takiej że $fl(\eta + 0.0) > 0.0$ **(plik ex-1.2.jl)**.

Oto tabela zawierająca moje wyniki:

η	*nextfloat(T(0.0))*	iteracyjnie
Float16	$6.0 \cdot 10^{-8}$	$6.0 \cdot 10^{-8}$
Float32	$1.0 \cdot 10^{-45}$	$1.0 \cdot 10^{-45}$
Float64	$5.0 \cdot 10^{-324}$	$5.0 \cdot 10^{-324}$

Jak można zauważyć powyżej, metoda iteracyjna wyliczająca liczbę maszynową η dla typów zmiennopozycyjnych jest bardzo precyzyjna i zwraca odpowiednie wartości.

Jaki związek ma liczba η z liczbą MIN_{sub}?

Liczba MIN_{sub} to najmniejsza dodatnia liczba zmiennoprzecinkowa, która może być reprezentowana w danym systemie arytmetyki zmiennoprzecinkowej. Ta definicja tyczy się również maszynowej liczby η, zatem oba te terminy są synonimiczne. Liczbę MIN_{sub} można zapisać jako:

$$MIN_{sub} = 2^{1-t} \cdot 2^{c_{min}},$$

gdzie: t - liczba bitów mantysy, c_{min} - najmniejsza wartość cechy.

1.3 Liczba (MAX)

Poleceniem było wyznaczyć iteracyjnie liczbę (MAX) dla wszystkich typów zmiennopozycyjnych w języku Julia oraz otrzymane wyniki porównać z wartościami zwracanymi przez funkcję *floatmax(T)* i z wartościami zawartymi w pliku nagłówkowym **float.h** języka C.

Moja funkcja iteracyjnie szukała liczby MAX (czyli największej dodatniej wartości znormalizowanej dla danego typu zmiennoprzecinkowego), poprzez:

a) Utworzenie mantysy dodając do siebie kolejno potęgi dwójki
b) Iteracyjnie mnożąc wynik przez dwa, aż do uzyskania nieskończoności w danej arytmetyce
c) Zwrócenie ostatniej liczby przed nieskończonością

Rozwiązanie znajduje się w pliku **ex-1.3.jl**. Znalazłem również wartości MAX dla typów *float* oraz *double* w pliku nagłówkowym języka C (są to wartości z dokładnością do 10 cyfry po przecinku). W tabeli poniżej umieściłem moje wyniki:

MAX	*floatmax(T)*	iteracyjnie	float.h
Float16	$6.55 \cdot 10^4$	$6.55 \cdot 10^4$	———
Float32	$3.4028235 \cdot 10^{38}$	$3.4028235 \cdot 10^{38}$	$3.4028234664 \cdot 10^{-38}$
Float64	$1.7976931348623157 \cdot 10^{308}$	$1.7976931348623157 \cdot 10^{308}$	$1.7976931349 \cdot 10^{308}$

Jak można zauważyć powyżej, metoda iteracyjna wyliczająca liczbę (MAX) dla typów zmiennopozycyjnych jest bardzo precyzyjna i zwraca odpowiednie wartości.

Co zwracają funkcje floatmin(Float32) i floatmin(Float64) i jaki jest związek zwracanych wartości z liczbą MIN_{nor}*?*

Zgodnie z dokumentacją języka Julia, funkcja *floatmin(T)* zwraca odpowiednio najmniejszą dodatnią liczbę znormalizowaną dla danego typu zmiennoprzecinkowego T. Zatem można powiedzieć, że wartością zwracaną przez tę funkcję jest $MIN_{nor} = 2^{c_{min}} = -2^{d-1} + 2$.

2 Metoda Kahana

Według Kahana, wartość *macheps* (czyli epsilonu maszynowego) można uzyskać za pomocą równania:

$$macheps = fl(3 \cdot (\tfrac{4}{3} - 1) - 1)$$

Celem tego zadania było skorzystać z powyższej metody do obliczenia epsilonu maszynowego oraz porównać otrzymane wartości z wartościami rzeczywistymi dla typów zmiennoprzecinkowych w języku Julia. Mój kod źródłowy zawarłem w pliku **ex-2.jl**, a tabela poniżej prezentuje moje wyniki:

macheps	*eps()*	Kahan
Float16	0.000977	-0.000977
Float32	$1.1920929 \cdot 10^{-7}$	$1.1920929 \cdot 10^{-7}$
Float64	$2.220446049250313 \cdot 10^{-16}$	$-2.220446049250313 \cdot 10^{-16}$

Jak można zauważyć powyżej, metoda Kahana wyliczająca epsilon maszynowy dla typów zmiennopozycyjnych jest bardzo precyzyjna i zwracająca poprawny, co do wartości bezwzględnej, wynik.

3 Rozmieszczenie liczb zmiennopozycyjnych

Zadanie polegało na eksperymentalnym sprawdzeniu równomiernego rozmieszczenia liczb zmiennopozycyjnych w zadanych przedziałach w arytmetyce **double**.

3.1 Przedział [1, 2]

Należało sprawdzić, czy w przedziale **[1, 2]** każda liczba zmiennopozycyjna x może być przedstawiona następująco:

$$x = 1 + k \cdot \delta,$$

gdzie:
$k \in \{0, 1, 2, \ldots, 2^{52}\}$, $\delta = 2^{-52}$.

Do rozwiązania tego zadania skorzystałem z funkcji *bitstring()*, która m.in. zwraca bitową reprezentację otrzymanej liczby, aby sprawdzić, czy przy podanych parametrach liczby zmiennopozycyjne są równomiernie rozmieszczone w przedziale **[1, 2]**. Stworzyłem też funkcję *checkcorrectness()*, która na końcach przedziału sprawdza, czy dana wartość kroku jest dla niego prawidłowa. Zastosowałem również podejście iteracyjne i po kolei drukowałem kolejne liczby z zadanego przedziału:

Value of k	sign (1 bit)	exponent (11 bits)	fraction (52 bits)
0	0	01111111111	0000000000000000000000000000000000000000000000000000**0**
1	0	01111111111	0000000000000000000000000000000000000000000000000001
2	0	01111111111	0000000000000000000000000000000000000000000000000010
3	0	01111111111	0000000000000000000000000000000000000000000000000011
4	0	01111111111	0000000000000000000000000000000000000000000000000100
5	0	01111111111	0000000000000000000000000000000000000000000000000101
6	0	01111111111	0000000000000000000000000000000000000000000000000110
7	0	01111111111	0000000000000000000000000000000000000000000000000111
8	0	01111111111	0000000000000000000000000000000000000000000000001000
9	0	01111111111	0000000000000000000000000000000000000000000000001001

W powyższej tabeli można zauważyć, że w każdej iteracji bitowa reprezentacja liczby zwiększa się o 1. Na podstawie tych wyników można wnioskować że liczby zmiennopozycyjne w przedziale **[1, 2]** są rozmieszczone równomiernie przy dobranych odpowiednio parametrach k oraz δ. Mój kod źródłowy znajduje się w pliku **ex-3.1.jl**.

3.2 Przedział $[\frac{1}{2}, 1]$

Jak dla przedziału [1, 2], wykonujemy dokładnie takie same obliczenia dla przedziału $[\frac{1}{2}, 1]$, natomiast musimy zmienić parametr δ. Jako, że teraz działamy na przedziale dwukrotnie mniejszym, to $\delta = 2^{-53}$ (oraz oczywiście na początku $x = \frac{1}{2}$). Kod źródłowy dla tej części zadania znajduje się w pliku **ex-3.2.jl**. Tak samo jak wcześniej, iteracyjnie drukowałem bitową reprezentację kolejnych liczb:

Zatem można wywynioskować, że w przedziale $[\frac{1}{2}, 1]$ liczby zmiennoprzecinkowe również są równomiernie rozmieszczone.

Value of k	sign (1 bit)	exponent (11 bits)	fraction (52 bits)
0	0	01111111110	0000000000000000000000000000000000000000000000000000**0**
1	0	01111111110	0000000000000000000000000000000000000000000000000001
2	0	01111111110	0000000000000000000000000000000000000000000000000010
3	0	01111111110	0000000000000000000000000000000000000000000000000011
4	0	01111111110	0000000000000000000000000000000000000000000000000100
5	0	01111111110	0000000000000000000000000000000000000000000000000101
6	0	01111111110	0000000000000000000000000000000000000000000000000110
7	0	01111111110	0000000000000000000000000000000000000000000000000111
8	0	01111111110	0000000000000000000000000000000000000000000000001000
9	0	01111111110	0000000000000000000000000000000000000000000000001001

3.3 Przedział [2, 4]

Tym razem przedział jest dwa razy dłuższy niż [1, 2] oraz cztery razy dłuższy niż $[\frac{1}{2}, 1]$, zatem ustawiamy krok jako $\delta = 2^{-51}$ oraz na początku $x = 2$ (kod źródłowy w pliku **ex-3.3.jl**). Tak jak w poprzednich przedziałach, poniżej zamieszczam tabelę z wynikami:

Value of k	sign (1 bit)	exponent (11 bits)	fraction (52 bits)
0	0	10000000000	0000000000000000000000000000000000000000000000000000**0**
1	0	10000000000	0000000000000000000000000000000000000000000000000001
2	0	10000000000	0000000000000000000000000000000000000000000000000010
3	0	10000000000	0000000000000000000000000000000000000000000000000011
4	0	10000000000	0000000000000000000000000000000000000000000000000100
5	0	10000000000	0000000000000000000000000000000000000000000000000101
6	0	10000000000	0000000000000000000000000000000000000000000000000110
7	0	10000000000	0000000000000000000000000000000000000000000000000111
8	0	10000000000	0000000000000000000000000000000000000000000000001000
9	0	10000000000	0000000000000000000000000000000000000000000000001001

W tym przypadku liczby zmiennoprzecinkowe także są równomierne rozmieszczone w zadanym przedziale.

3.4 Dla rozpatrywanego przedziału

W arytmetyce *Float64* wartość kroku δ można obliczyć na podstawie wzoru:

$$\delta = 2^c \cdot 2^{1-t} = 2^{c_{KN}-(2^{d-1}-1)} \cdot 2^{1-t} \overset{Float64}{=} 2^{c_{KN}-1023} \cdot 2^{-52},$$

ponieważ dla *Float64*: $t = 53$, $d = 11$ (liczba bitów przeznaczona na eksponentę).

4 Błędy w arytmetyce

Celem tego zadania było znalezienie w arytmetyce fl = *Float64*(**double**) liczby zmiennopozycyjnej x w przedziale $1 < x < 2$, takiej że:

$$fl(x \cdot fl(\tfrac{1}{x})) \neq 1,$$

a także znalezienie najmniejszej takiej liczby x.

Mój program (**plik ex-4.jl**) od razu znajduje najmniejszą taką liczbę iteracyjnie. Do znalezienia rozwiązania posłużyłem się wartością *macheps*. W moim kodzie źródłowym pętla *while* wykonuje się do momentu, aż $fl(x \cdot fl(\frac{1}{x})) \neq 1$, zaczynając od $x = 1 +$ *macheps*. Dla każdej kolejnej iteracji liczba x zwiększa się o *macheps*.

Liczba, którą podaje program, ma wartość: **1.000000057228997**.

5 Obliczanie iloczynu skalarnego dwóch wektorów

W tym zadaniu należało stworzyć cztery algorytmy w języku Julia obliczające w arytmetyce *Float32* oraz *Float64* iloczyn skalarny dwóch wektorów o rozmiarach n:

a) "w przód"
b) "w tył"
c) "od największego do najmniejszego"
d) "od najmniejszego do największego"

Prawidłowość działania algorytmów należało przetestować dla dwóch wektorów x, y dla $n = 5$. Wektory x i y są postaci:

$$x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$$
$$y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$$

Wyniki trzeba było również porównać z rzeczywistą wartością iloczynu skalarnego wektorów x i y, która wynosi: $-1.00657107000000 \cdot 10^{-11}$. Wartość ta jest bardzo bliska zeru, co pozwala wnioskować, że "prawie"$x \perp y$ i wyniki algorytmów będą błędne. Kod źródłowy znajduje się w pliku **ex-5.jl**, a poniżej znajduje się tabela z wynikami zwróconymi przez mój program:

	algorytm a)	algorytm b)	algorytm c)	algorytm d)
Float32	-0.4999443	-0.4543457	-0.5	-0.5
Float64	$1.0251881368296672 \cdot 10^{-10}$	$-1.5643308870494366 \cdot 10^{-10}$	0.0	0.0

Jak można zauważyć powyżej, wyniki dla arytmetyki *Float32* znacząco odbiegają od rzeczywistej wartości iloczynu skalarnego wektorów x i y. To samo dotyczy arytmetyki *Float64*, w której jedynie wynik z wykorzystaniem algorytmu "w tył" jest jakkolwiek bliski dobrej odpowiedzi (lecz mimo wszystko podaje wynik około 15 razy mniejszy). Wniosek jaki można wyciągnąć, to taki, że kolejność wykonywania działań ma wpływ na precyzję obliczeń oraz że liczenie iloczynu skalarnego dla dwóch wektorów (prawie) prostopadłych jest zadaniem źle uwarunkowanym.

6 Błędy w arytmetyce ciąg dalszy

Zadanie dotyczyło policzenia w języku Julia w arytmetyce *Float64* wartości następujących funkcji

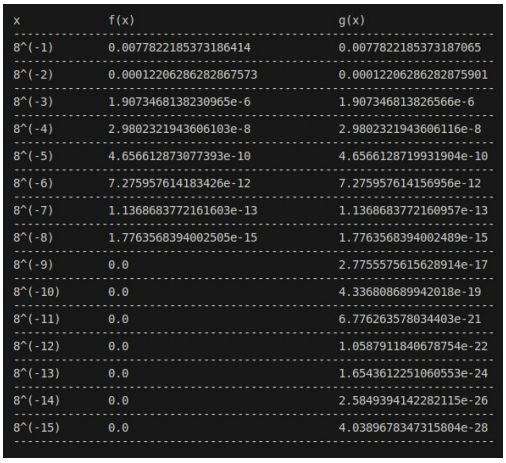
$$f(x) = \sqrt{x^2+1} - 1 \textbf{ oraz } g(x) = \frac{x^2}{\sqrt{x^2+1}+1},$$

dla $x \in \{8^{-1}, 8^{-2}, 8^{-3}, \ldots\}$.

Na podstawie uzyskanych rezultatów trzeba było również stwierdzić, która z tych funkcji zwraca bardziej wiarygodne wyniki. Na samym dole tej strony przedstawiłem otrzymane przeze mnie wyniki programu **ex-6.jl**, w którym zwracałem rezultaty funkcji f oraz g dla x ze zbioru podanego wyżej:

Co pierwsze rzuca się w oczy, to fakt, iż funkcja f dla wartości x mniejszych od 8^{-8} zwraca 0.0 jako wynik, gdzie funkcja g zachowuje się "normalnie" i dalej drukuje wiarygodne wyniki. Może to być spowodowane tym, iż należy unikać odejmowania od siebie liczb podobnej wielkości, gdyż taki proces może prowadzić do utraty liczb znaczących oraz precyzji.

W funkcji f, przy x^2 dążącym do zera, zmierzamy do sytuacji, w której odejmujemy jeden od "wartości coraz to bliższej jedynce". Natomiast w funkcji g działamy na dodawaniu i owy problem nie występuje, dlatego zwraca ona wiarygodniejsze wyniki.



x	f(x)	g(x)
8^(-1)	0.0077822185373186414	0.0077822185373187065
8^(-2)	0.00012206286282867573	0.00012206286282875901
8^(-3)	1.9073468138230965e-6	1.907346813826566e-6
8^(-4)	2.9802321943606103e-8	2.9802321943606116e-8
8^(-5)	4.656612873077393e-10	4.6566128719931904e-10
8^(-6)	7.275957614183426e-12	7.2759576141569560e-12
8^(-7)	1.1368683772161603e-13	1.1368683772160957e-13
8^(-8)	1.7763568394002505e-15	1.7763568394002489e-15
8^(-9)	0.0	2.7755575615628914e-17
8^(-10)	0.0	4.336808689942018e-19
8^(-11)	0.0	6.776263578034403e-21
8^(-12)	0.0	1.0587911840678754e-22
8^(-13)	0.0	1.6543612251060553e-24
8^(-14)	0.0	2.5849394142282115e-26
8^(-15)	0.0	4.0389678347315804e-28

Rysunek 1: Wyniki programu **ex-6.jl**

7 Przybliżanie pochodnej funkcji

Zadanie dotyczyło problemu przybliżania wartości pochodnej *f(x)* w punkcie x wzorem:

$$f'(x_0) \approx \tilde{f}'(x_0) = \frac{f(x_0+h)-f(x_0)}{h}.$$

Należało skorzystać z tego wzoru w języku Julia w arytmetyce *Float64* do przybliżenia wartości pochodnej funkcji:

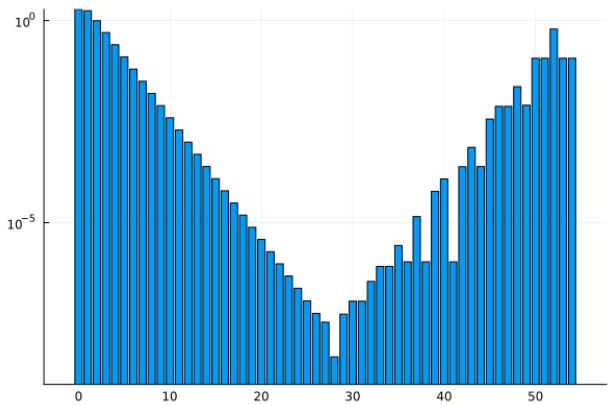
$$f(x) = sinx + cos3x,$$

w punkcie $x_0 = 1$.
Poniżej znajdują się wyniki mojego programu **ex-7.jl**, w którym to dla h $= 2^{-n}$, gdzie $n \in \{0, 1, 2, \ldots, 54\}$, obliczyłem przybliżoną wartość pochodnej funkcji f oraz błędy $|f'(x_0) - \tilde{f}'(x_0)|$.

Z wykresu na Rysunku 2. można wywnioskować, że najmniejszy błąd w aproksymacji pochodnej funkcji f występuje dla $n = 28$ (czyli dla $h = 2^{-28}$). Dla $n > 28$ błąd znowu staje się coraz większy.

Na Rysunku 3. w kolumnie "$1+h$" można zauważyć, że wartość h stopniowo zbliża się do zera. Powinno się unikać odejmowania od siebie bardzo mocno przybliżonych wartości, co dzieje się w przypadku kiedy h przyjmuje wartości coraz to bliższe zeru. Wtedy $f(x_0+h) \approx f(x_0)$ dla $x_0 = 1$ i następuje wymieniony wyżej problem.

Dla wartości $h < 2^{-52}$ zmienna h przyjmuje nawet wartość 0. Zatem, do pewnego momentu precyzja arytmetyki osiąga swój maksymalny poziom, po czym (poprzez przykładowo odejmowanie od siebie dwóch bardzo mocno zbliżonych do siebie liczb), ta precyzja jest tracona przez utratę liczb znaczących w rachunkach komputera.



Rysunek 2: Wartość błędu aproksymacji pochodnej dla $x_0 = 1$.

h	1 + h	approximation	error
2^(0)	2.0	2.0179892252685967	1.9010469435800585
2^(-1)	1.5	1.8704413979316472	1.753499116243109
2^(-2)	1.25	1.1077870952342974	0.9908448135457593
2^(-3)	1.125	0.6232412792975817	0.5062989976090435
2^(-4)	1.0625	0.3704000662035192	0.253457784514981
2^(-5)	1.03125	0.24344307439754687	0.1265007927090087
2^(-6)	1.015625	0.1800975633073278 5	0.063155281618789 7
2^(-7)	1.0078125	0.1484913953710958	0.0315491136825576 4
2^(-8)	1.0039625	0.1327091142805159	0.0157668325919777 53
2^(-9)	1.001953125	0.1248236929407085	0.0078814112521703 45
2^(-10)	1.0009765625	0.1208824768110616 8	0.0039401951225235 265
2^(-11)	1.00048828125	0.1189122504688384 7	0.0019699687803003 13
2^(-12)	1.000244140625	0.1179272337390102 6	0.0009849520504721 099
2^(-13)	1.0001220703125	0.1174347496107657 2	0.0004924679222275 685
2^(-14)	1.00006103515625	0.1171885136209311 9	0.0002462319323930 373
2^(-15)	1.00003051757812 5	0.1170653971457795 7	0.0001231154572414 1837
2^(-16)	1.00001525878906 25	0.1170038392883725 5	6.1557599834394 24e-5
2^(-17)	1.00000762939453 12	0.1169730604597134 5	3.0778771175299 37e-5
2^(-18)	1.00000381469726 56	0.1169576710672117 8	1.5389378673624 776e-5
2^(-19)	1.00000190734863 28	0.1169499763636849 8	7.6946751468298 66e-6
2^(-20)	1.00000095367431 64	0.1169461290119215 8	3.8473233834324 105e-6

(a) Dla $0 \leq n \leq 20$

2^(-21)	1.0000004768371582	0.1169442052487284	1.9235601902423127e-6
2^(-22)	1.0000002384185791	0.1169432429595967 17	9.6127114002086 96e-7
2^(-23)	1.0000001192092896	0.1169427623972296 7	4.8070869151928 26e-7
2^(-24)	1.0000000596046448	0.1169425211846828 5	2.3949614469387 37e-7
2^(-25)	1.0000000298023224	0.1169423982501 03	1.1656156484463 054e-7
2^(-26)	1.0000000149011612	0.1169423386454582 2	5.6956920069239 914e-8
2^(-27)	1.0000000074505806	0.1169423162937164 3	3.4605178278468 43e-8
2^(-28)	1.0000000037252903	0.1169422864913940 4	4.8028558907731 17e-9
2^(-29)	1.0000000018626451	0.1169422268867492 7	5.4801788884617 51e-8
2^(-30)	1.0000000009313226	0.1169421672821044 9	1.1440643366000 813e-7
2^(-31)	1.0000000004656613	0.1169421672821044 9	1.1440643366000 813e-7
2^(-32)	1.0000000002328306	0.1169419288635253 9	3.5282501276157 063e-7
2^(-33)	1.0000000001164153	0.1169414520263671 9	8.2966217096469 56e-7
2^(-34)	1.0000000000582077	0.1169414520263671 9	8.2966217096469 56e-7
2^(-35)	1.0000000000291038	0.1169395446777343 8	2.7370108037771 956e-6
2^(-36)	1.000000000014552	0.116943359375	1.0776864618478 044e-6
2^(-37)	1.000000000007276	0.1169281005859375	1.4181102600652 196e-5
2^(-38)	1.000000000003638	0.116943359375	1.0776864618478 044e-6
2^(-39)	1.000000000001819	0.1168823242187 5	5.9957469788152 196e-5
2^(-40)	1.0000000000009095	0.1168212890625	0.0001209926260381522
2^(-41)	1.0000000000004547	0.116943359375	1.0776864618478 044e-6
2^(-42)	1.0000000000002274	0.11669921875	0.0002430629385381522

(b) Dla $21 \leq n \leq 42$

2^(-43)	1.0000000000001137	0.1162109375	0.0007313441885381522
2^(-44)	1.0000000000000568	0.1171875	0.0002452183114618478
2^(-45)	1.0000000000000284	0.11328125	0.0033661031688538152
2^(-46)	1.0000000000000142	0.109375	0.007567281688538152
2^(-47)	1.000000000000007	0.109375	0.007567281688538152
2^(-48)	1.0000000000000036	0.09375	0.023192281688538152
2^(-49)	1.0000000000000018	0.125	0.008057718311461848
2^(-50)	1.0000000000000009	0.0	0.11694228168853815
2^(-51)	1.0000000000000004	0.0	0.11694228168853815
2^(-52)	1.0000000000000002	-0.5	0.6169422816885382
2^(-53)	1.0	0.0	0.11694228168853815
2^(-54)	1.0	0.0	0.11694228168853815

(c) Dla $43 \leq n \leq 54$

Rysunek 3: Wyniki programu **ex-7.jl** dla $x_0 = 1$.