-->

Sprawozdanie z Listy nr 1

Sara Żyndul

279686

Zadanie 1

1.1 Opis i rozwiązanie

Rozwiązanie obejmujące iteracyjne wyznaczanie epsilona maszynowego macheps, liczby maszynowej eta oraz liczby MAX znajduję się w pliku `task1.jl`

1.2 Wyniki

Typ	Iteracyjnie	eps(Typ)	Wartość z `float.h`
Float16	$9.765625000000 \times 10^{-4}$	$9.765625000000 \times 10^{-4}$	N/A
Float32	$1.192092895508 \times 10^{-7}$	$1.192092895508 \times 10^{-7}$	$1.1920928955078125 \times 10^{-7}$
Float64	$2.220446049250 \times 10^{-16}$	$2.220446049250 \times 10^{-16}$	$2.2204460492503131 \times 10^{-16}$

Tabela 1: Porównanie wyników eksperymentalnego wyznaczenia macheps z wartościami zwracanymi przez funkcję eps() oraz danymi zawartymi w pilku nagłówkowym `float.h` dowolnej instalacji języka C.

Typ	Iteracyjnie	nextFloat()
Float16	$5.960464477539 \times 10^{-8}$	$5.960464477539 \times 10^{-8}$
Float32	$1.401298464325 \times 10^{-45}$	$1.401298464325 \times 10^{-45}$
Float64	$4.940656458412 \times 10^{-324}$	$4.940656458412 \times 10^{-324}$

Tabela 2: Porównanie wyników eksperymentalnego wyznaczenia eta z wartościami zwracanymi przez funkcję `nextfloat(typ(0.0))`.

Typ	`floatmin()`	MIN_{nor}
Float32	$1.175494350822 \times 10^{-38}$	1.2×10^{-38}
Float64	$2.225073858507 \times 10^{-308}$	2.2×10^{-308}

Tabela 3: Porównanie wyników zwracanych przez funkcję floatmin() z wartościami liczby MIN_{nor} podanej na wykładzie.

Typ	Iteracyjnie	`floatmax()`	MAX (wykład)
Float16	$6.550400000000 \times 10^{4}$	$6.550400000000 \times 10^{4}$	N/A
Float32	$3.402823466385 \times 10^{38}$	$3.402823466385 \times 10^{38}$	3.4×10^{38}
Float64	$1.797693134862 \times 10^{308}$	$1.797693134862 \times 10^{308}$	1.8×10^{308}

Tabela 4: Porównanie wyników eksperymentalnego wyznaczenia MAX z wartościami zwracanymi przez funkcję `floatmax()` oraz danymi z wykładu.

1.3 Interpretacja wyników oraz wnioski

- **Zgodność wyników:**
  Eksperymentalne wyniki iteracyjne zgadzają się (do dokładności wydruku) z funkcjami wbudowanymi (`eps` , `nextfloat` , `floatmin` , `floatmax`) oraz z wartościami z `float.h` (gdzie dostępne).

- **Epsilon maszynowy (macheps):**

  macheps jest stałą wprost wynikającą z liczby bitów mantysy — jest to odległość między 1.0 a następną reprezentowalną liczbą. $\mathrm{macheps} = 2^{1-t}$, gdzie t to liczba bitów znaczących (mantysa + 1). Możemy zauważyć następujący związek między precyzją arytmetyki ϵ a epsilonem maszynowym:

  $$macheps = 2 * \epsilon$$

- **Eta:** eta to najmniejsza dodatnia liczba możliwa do reprezentacji (zwykle podnormalna). `nextfloat(0.0)` zwraca dokładnie tę wartość.

- **Związek eta z MIN_{sub}:** eta ≈ MIN_{sub} — czyli minimalna wartość zdenormalizowana.

- **Związek floatmin z MIN_{nor}**: Wartość MIN_{nor}, czyli najmniejszej znormalizowanej liczby w danym typie zmiennopozycyjnym jest w przybliżeniu równa odpowiednim wartościom zwracanym przez `floatmin()` .

- **MAX (największa skończona wartość):** `prevfloat(Inf(T))` oraz `floatmax(T)` dają tę samą wartość. Do wykrywania overflowu użyteczne jest `isinf(T(value))` .

Zadanie 2

2.1 Opis i rozwiązanie

Rozwiązanie obejmujące wyznaczanie 3(4/3–1)–1 w arytmetyce zmiennopozycyjnej, które

według Kahan'a przybliża epsilon maszynowy (macheps) znajduję się w pliku `task2.jl`.

2.2 Wyniki

Typ	Wartość wzoru Kahan'a	eps()
Float16	-9.77×10^{-4}	9.77×10^{-4}
Float32	1.1920929×10^{-7}	1.1920929×10^{-7}
Float64	$-2.220446049250313 \times 10^{-16}$	$-2.220446049250313 \times 10^{-16}$

Tabela 1: Porównanie wyników wyliczeń wyrażenia Kahan'a $K = 3(4/3 - 1) - 1$ z wartościami zwracanymi przez funkcję eps().

2.3 Wnioski

Można zauważyć że, aby dla badanych typów wyrażenie Kahan'a poprawnie wyznaczało epsilon maszynowy trzeba na wynik operacji nałożyć moduł, a więc $macheps = |3(4/3 - 1) - 1|$. Błędy w bicie znaku wynikają z różnej ostatniej cyfry mantysy w reprezentacji fl(4/3).

$K = 3(fl(4/3) - 1) - 1 = 3(4/3 + \delta - 1) - 1 = 3(1/3 + \delta) - 1 = 1 + 3\delta - 1 = 3\delta$

Jeśli $fl(4/3) < 4/3$ to $\delta < 0$ i $K < 0$.

Jeśli $fl(4/3) > 4/3$ to $\delta > 0$ i $K > 0$.

Zadanie 3

3.1 Opis i Rozwiązanie

Zadanie polega na eksperymentalnym sprawdzeniu, czy liczby są równomiernie rozłożone w odpowiednich przedziałach ($[1, 2], [1/2, 1]$ oraz $[2, 4]$) i sprawdzeniu ile wynosi krok w przedziale (odległość między następnymi liczbami). Rozwiązanie znajduje się w pliku task3.jl.

3.2 Wyniki

k	x ($1 + k * 2^{-52}$)	20 dolnych bitów mantysy	ok?
0	1.000000000000000000	00000000000000000000	TAK
1	1.000000000000000222	00000000000000000001	TAK
2	1.000000000000000444	00000000000000000010	TAK

k	x ($1 + k * 2^{-52}$)	20 dolnych bitów mantysy	ok?
3	1.0000000000000666	00000000000000000011	TAK
4	1.0000000000000888	00000000000000000100	TAK
5	1.0000000000001110	00000000000000000101	TAK
6	1.0000000000001332	00000000000000000110	TAK
7	1.0000000000001554	00000000000000000111	TAK
8	1.0000000000001776	00000000000000001000	TAK
9	1.0000000000001998	00000000000000001001	TAK
10	1.0000000000002220	00000000000000001010	TAK

Tabela 1: Sprawdzenie, czy wzór $1 + k \times 2^{-52}$ odwzorowuje kolejne liczby Float64 w przedziale $[1, 2]$. Możemy zauważyć, że dla kolejnych wartości k mantysa pokazuje, że kroki między liczbami są pojedyncze. Kolumna 'ok?' sprawdza, czy mantysa odzwierciedla liczbę k oraz czy x jest równy dokładnie $1 + k \times 2^{-52}$.

x	nextfloat(x) – x	krok teoretyczny (2^{e-52})	wykładnik e
1.0000000000000000	$2.220446049250313081 * 10^{-16}$	$2.220446049250313081 * 10^{-16}$	0
1.0000000000000022	$2.220446049250313081 * 10^{-16}$	$2.220446049250313081 * 10^{-16}$	0
1.0000000000002220	$2.220446049250313081 * 10^{-16}$	$2.220446049250313081 * 10^{-16}$	0
1.5000000000000000	$2.220446049250313081 * 10^{-16}$	$2.220446049250313081 * 10^{-16}$	0
1.9999999999999978	$2.220446049250313081 * 10^{-16}$	$2.220446049250313081 * 10^{-16}$	0

Tabela 2: Próbkowanie na przedziale $[1, 2]$. Tabela przedstawia wybrane wartości x i pozwala sprawdzić, że odległość między następnymi liczbami maszynowymi jest równa 2^{e-52}, gdzie e to rzeczywisty wykładniki liczby x (tutaj $e = 0$).

x	nextfloat(x) – x	krok teoretyczny (2^{e-52})	wykładnik e
$5.0000000000000000 * 10^{-1}$	$1.110223024625156540 * 10^{-16}$	$1.110223024625156540 * 10^{-16}$	-1
$5.0000000000000111 * 10^{-1}$	$1.110223024625156540 * 10^{-16}$	$1.110223024625156540 * 10^{-16}$	-1
$7.5000000000000000 * 10^{-1}$	$1.110223024625156540 * 10^{-16}$	$1.110223024625156540 * 10^{-16}$	-1
$9.9999999999999889 * 10^{-1}$	$1.110223024625156540 * 10^{-16}$	$1.110223024625156540 * 10^{-16}$	-1

Tabela 3: Próbkowanie na przedziale $[1/2, 1]$. Reszta j.w..

x	nextfloat(x) – x	krok teoretyczny (2^{e-52})	wykładnik e
2.0000000000000000	$4.440892098500626162 * 10^{-16}$	$4.440892098500626162 * 10^{-16}$	1

x	nextfloat(x) – x	krok teoretyczny (2^{e-52})	wykładnik e
2.0000000000000044	$4.440892098500626162 * 10^{-16}$	$4.440892098500626162 * 10^{-16}$	1
3.0000000000000000	$4.440892098500626162 * 10^{-16}$	$4.440892098500626162 * 10^{-16}$	1
3.9999999999999956	$4.440892098500626162 * 10^{-16}$	$4.440892098500626162 * 10^{-16}$	1

Tabela 4: Próbkowanie na przedziale $[2, 4]$. Reszta j.w..

3.3 Wnioski

Możemy zauważyć, że każda liczba zmiennopozycyjna na przedziale $[1, 2]$ jest równomiernie rozmieszczona z krokiem $\delta = 2^{-52}$. Po wykonaniu eksperymentów na przedziałach $[1, 2]$, $[1/2, 1]$ oraz $[2, 4]$ możemy stwierdzić, że liczby na przedziale $[2^e, 2^{e+1}]$ są rozmieszczone równomiernie z krokiem 2^{e-52}.

Zadanie 4

4.1 Opis i Rozwiązanie

Rozwiązanie obejmujące znalezienie pierwszego takiego x w arytmetyce Float64, że $1 < x < 2$ oraz $x \times (1/x) \neq 1$ znajduje się w pliku task4.jl.

4.2 Wyniki

Najmniejszą taką liczbą jest $x = 1.000000057228997$, dla której $x * (1/x) = 0.9999999999999999$.

4.3 Wnioski

Nawet przy prostych, pojedynczych operacjach w arytmetyce zmiennopozycyjnej trzeba brać pod uwagę możliwość pojawienia się błędu.

Zadanie 5

5.1 Opis i Rozwiązanie

Zadanie polega na wykorzystaniu czterech algorytmów sumowania do obliczenia iloczynu skalarnego dwóch wektorów:

$x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$

$y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$.

Rozwiązanie znajduje się w pliku task5.jl.

5.2 Wyniki

Wynik dokładny iloczynu skalarnego: $-1.00657107000000 \times 10^{-11}$

Algorytm	Float32	Float64
a	-4.999442994594573975	$1.025188136829667182 \times 10^{-10}$
b	-4.543457031250000000	$-1.564330887049436569 \times 10^{-10}$
c	-5.000000000000000000	0.000000000000000000
d	-5.000000000000000000	0.000000000000000000

Tabela 1: Porównanie obliczania iloczynu skalarnego w precyzji pojedynczej i podwójnej poprzez algorytmy dodawania: a) po kolei w przód, b) po kolei od tyłu, c) od największego do najmniejszego oraz d) od najmniejszego do największego.

5.3 Wnioski

Float32 ma za małą precyzję, by uzyskać dobre przybliżenie tego iloczynu skalarnego i daje bardzo duży błąd. Float64 pozwala uzyskać o wiele lepsze przybliżenie iloczynu skalarnego, przy czym algorytmy a i b dają błąd bezwzględny rzędu 10^{-10}, a algorytmy c i d dają mniejszy błąd rzędu 10^{-11}. Grupowanie zgodnie ze znakiem pozwala więc zmniejszyć anulację.

Zadanie 6

6.1 Cel i Rozwiązanie

Zadanie polegało na obliczaniu kolejnych wartości funkcji f i g:
$f(x) = \sqrt{(x^2+1)} - 1$, $g(x) = x^2/(\sqrt{(x^2+1)} + 1)$ i określeniu, która z nich jest bardziej wiarygodna (matematycznie $f = g$).

Rozwiązanie znajduje się w pliku task6.jl

6.2 Wyniki

n	Wartość $f(8^{-n})$	Wartość $g(8^{-n})$
1	$7.782218537318641438 * 10^{-3}$	$7.782218537318706490 * 10^{-3}$
2	$1.220628628286757333 * 10^{-4}$	$1.220628628287590136 * 10^{-4}$
3	$1.907346813823096454 * 10^{-6}$	$1.907346813826565901 * 10^{-6}$

n	Wartość $f(8^{-n})$	Wartość $g(8^{-n})$
4	$2.980232194360610265 * 10^{-8}$	$2.980232194360611588 * 10^{-8}$
5	$4.656612873077392578 * 10^{-10}$	$4.656612871993190406 * 10^{-10}$
6	$7.275957614183425903 * 10^{-12}$	$7.275957614156956124 * 10^{-12}$
7	$1.136868377216160297 * 10^{-13}$	$1.136868377216095674 * 10^{-13}$
8	$1.776356839400250465 * 10^{-15}$	$1.776356839400248887 * 10^{-15}$
9	0.000000000000000000	$2.775557561562891351 * 10^{-17}$
...	...	...
176	0.000000000000000000	$6.475817233170386704 * 10^{-319}$
177	0.000000000000000000	$1.011846442682872922 * 10^{-320}$
178	0.000000000000000000	$1.581010066691988941 * 10^{-322}$
179	0.000000000000000000	0.000000000000000000

Tabela 1: Porównanie wartości funkcji f i g dla kolejnych wartości argumentu $x = 8^{-1}, 8^{-2}, ...$

6.3 Wnioski

Obie funkcje są matematycznie równe, ale w arytmetyce zmiennoprzecinkowej dają różne wyniki z powodu zaokrągleń.

$f(x) = \sqrt(x^2 + 1) - 1$ traci dokładność przez odejmowanie dwóch bliskich liczb - stąd już dla umiarkowanie małych x wyniki są mniej dokładne, a dla wszystkich $i \geq 9$ funkcja daje dokładne 0 w Float64.

$g(x) = x^2/(\sqrt(x^2 + 1) + 1)$ jest numerycznie stabilniejsza - unika bezpośredniej anulacji i daje poprawne małe wartości (w tym podnormalne) do momentu rzeczywistego underflow w arytmetyce.

g kontynuuje malejącą skalę aż do bardzo małych rzędów równych 10^{-322} zanim zaniknie; f „zeruje się” dużo wcześniej - to pokazuje, że g zachowuje znaczące cyfry tam, gdzie f ich nie zachowuje.

Zadanie 7

7.1 Cel i Rozwiązanie

Celem zadania było eksperymentalne sprawdzenie jakie wartości h pozwalają najlepiej przybliżyć wartość pochodnej wyliczonej ze wzoru $(f(x_0 + h) - f(x_0))/h$. Rozwiązanie zadania znajduję się w pliku task7.jl.

7.2 Wyniki

n	Przybliżenie pochodnej f	Błąd bezwzględny	Wartość (h+1)
0	$2.0179892252685967 * 10^{0}$	$1.9010469435800585 * 10^{0}$	2.0000000000000000
1	$1.8704413979316472 * 10^{0}$	$1.7534991162431091 * 10^{0}$	1.5000000000000000
2	$1.1077870952342974 * 10^{0}$	$9.9084481354575926 * 10^{-1}$	1.2500000000000000
3	$6.2324127929758166 * 10^{-1}$	$5.0629899760904351 * 10^{-1}$	1.1250000000000000
4	$3.7040006620351917 * 10^{-1}$	$2.5345778451498102 * 10^{-1}$	1.0625000000000000
5	$2.4344307439754687 * 10^{-1}$	$1.2650079270900871 * 10^{-1}$	1.0312500000000000
6	$1.8009756330732785 * 10^{-1}$	$6.3155281618789694 * 10^{-2}$	1.0156250000000000
...	...	...	...
26	$1.1694233864545822 * 10^{-1}$	$5.6956920069239914 * 10^{-8}$	1.0000000149011612
27	$1.1694231629371643 * 10^{-1}$	$3.4605178278468429 * 10^{-8}$	1.0000000074505806
28	$1.1694228649139404 * 10^{-1}$	$4.8028558907731167 * 10^{-9}$	1.0000000037252903
29	$1.1694222688674927 * 10^{-1}$	$5.4801788884617508 * 10^{-8}$	1.0000000018626451
30	$1.1694216728210449 * 10^{-1}$	$1.1440643366000813 * 10^{-7}$	1.0000000009313226
...	...	...	...
49	$1.2500000000000000 * 10^{-1}$	$8.0577183114618478 * 10^{-3}$	1.0000000000000018
50	$0.0000000000000000 * 10^{0}$	$1.1694228168853815 * 10^{-1}$	1.0000000000000009
51	$0.0000000000000000 * 10^{0}$	$1.1694228168853815 * 10^{-1}$	1.0000000000000004
52	$-5.0000000000000000 * 10^{-1}$	$6.1694228168853815 * 10^{-1}$	1.0000000000000002
53	$0.0000000000000000 * 10^{0}$	$1.1694228168853815 * 10^{-1}$	1.0000000000000000
54	$0.0000000000000000 * 10^{0}$	$1.1694228168853815 * 10^{-1}$	1.0000000000000000

Tabela 1: Tabela przedstawia wyliczone w arytmetyce Float64 przybliżone wartości pochodnej dla $h = 2^{-n}$ wraz z błędem bezwzględnym obliczonej wartości z wartością dokładną w arytmetyce Float64 równą 0.11694228168853815.

7.3 Wnioski

Gdy nie bierzemy pod uwagę błędów wynikających z arytmetyki zmiennopozycyjnej, przy przybliżaniu wartości pochodnej wzorem podanym w zadaniu im mniejsza jest wartość h, tym bardziej dokładna jest wartość pochodnej.

W arytmetyce Float64 błąd bezwzględny stale się zmniejsza od $n = 0$ aż do $n = 28$, dla którego wartość pochodnej jest najbardziej dokładna. Dla większych n z powodu błędów zaokrągleń i w końcu efektów takich jak $1 + h == 1$ dla $n >= 53$ dokładność spada.