Obliczenia naukowe - Lista 1

Wojciech Pakulski (250350)

1 Rozpoznanie arytmetyki

1.1 Wyznaczanie epsilonów maszynowych

Epsilonem maszynowym macheps (ang. machine epsilon) nazywamy najmniejszą liczbę macheps > 0 taką, że:

(1) $\mathrm{fl}(1.0 + \mathrm{macheps}) > 1.0$
(2) $\mathrm{fl}(1.0 + \mathrm{macheps}) = 1 + \mathrm{macheps}.$

Zadanie polegało na tym, żeby w języku Julia wyznaczyć iteracyjnie epsilony maszynowe dla typów $\mathrm{Float16}$, $\mathrm{Float32}$ i $\mathrm{Float64}$ zgodnych ze standardem $\mathrm{IEEE}\ 754$ i porównać z wartościami zwracanymi przez funkcję *eps*.

Kod źródłowy programu znajduje się w pliku zad1.jl – jako funkcja o nazwie macheps. Algorytm polega na:

- Wartości macheps przyporządkowana jest wartość 1
- Wartość macheps jest dzielona na 2, dopóki spełniony jest warunek:

$$\left(1 + \frac{macheps}{2}\right) \neq 1$$

Wyniki działania napisanego programu:

	Wyliczony macheps	Funkcja *eps*	Wartość z float.h
Float16	0.000977	0.000977	-
Float32	1.1920929e-7	1.1920929e-7	1.1920928955078125e-7
Float64	2.220446049250313e-16	2.220446049250313e-16	2.2204460492503131e-16

1.2 Wyznaczanie liczby eta

Zadanie polegało na wyznaczeniu iteracyjnie liczby maszynowej eta, dla wszystkich typów zmiennopozycyjnych zgodnych ze standardem $\mathrm{IEEE}\ 754$, takiej, że:

$$\mathrm{eta} > 0.0$$

i porównaniu otrzymanych wyników z działaniem funkcji *nextfloat*.

Kod źródłowy programu znajduje się w pliku zad1.jl – jako funkcja o nazwie eta. Algorytm polega na:

- Wartości eta przyporządkowana jest wartość 1
- Wartość macheps jest dzielona na 2, dopóki spełniony jest warunek:

$$\frac{eta}{2} > 0$$

Wyniki działania napisanego programu:

	Wyliczona eta	Wynik funkcji *nextfloat*
Float16	6.0e-8	6.0e-8
Float32	1.0e-45	1.0e-45
Float64	5.0e-324	5.0e-324

1.3 Jaki związek ma liczba macheps z precyzją arytmetyki (oznaczaną na wykładzie przez ε)?

Wielkość macheps jest równa różnicy między liczbą 1, a następną większą liczbą, którą można zapisać w systemie zmiennopozycyjnym. Następstwem tego jest przypadek, jeśli liczba rzeczywista nie może zostać zapisana na określonej liczbie bitów (w systemie dwójkowym), to jest przybliżana. Powoduje to błąd względny.

Wniosek: im mniejsza jest wartość epsilona maszynowego, tym większa jest precyzja obliczeń.

1.4 Jaki związek ma liczba eta z liczbą MIN_{sub}?

Liczba eta jest równa liczbie MIN_{sub}. Obie są najmniejszą wartością zmiennoprzecinkową większą od 0, którą można zapisać w danym systemie. Są liczbami zdenormalizowanymi, czyli wszystkie bity cechy wynoszą 0, a ostatni bit mantysy wynosi 1.

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

Funkcje *floatmin(Float32)* i *floatmin(Float64)* zwracają najmniejsze dodatnie liczby zmiennoprzecinkowe podanych typów w postaci znormalizowanej. Czyli są równe z wartością liczby MIN_{nor}.

1.6 Wyznaczanie liczby MAX

Zadanie polegało na iteracyjnym wyznaczeniu liczby MAX dla wszystkich typów zmiennopozycyjnych zgodnych ze standardem IEEE 754 (half, single, double) i porównać je z wartościami zwracanymi przez funkcję *floatmax* oraz z danymi zawartymi w pliku nagłówkowym float.h instancji języka C.

Kod źródłowy programu znajduje się w pliku zad1.jl – jako funkcja o nazwie max.

Algorytm polega na:

- Przyporządkowujemy 1 wartości test
- Mnożymy ją razy dwa dopóki: $isinf(2 \cdot test) == false$
- Zachowując wartość test w zmiennej pomocniczej prev, dodajemy do wartości test wartość adder (równą wartości $\frac{test}{2}$). Jeśli okaże się, że dodanie jej sprawia, że isinf(test) == true, to dzielimy adder na dwa dopóki isinf(test + adder) == true. Gdy się uda to powtarzamy ten punkt

Wyniki działania napisanego programu:

	Wyliczony MAX	Funkcja *floatmax*	Wartość z float.h
Float16	6.55e4	6.55e4	-
Float32	3.4028235e38	3.4028235e38	3.40282347e+28
Float64	1.7976931348623157e308	1.7976931348623157e308	1.7976931348623157e+308

2 Epsilon maszynowy Kahana

Kahan stwierdził, że epsilon maszynowy (macheps) można otrzymać obliczając wartość wyrażenia: $3 \cdot \left(\frac{4}{3} - 1\right) - 1$ w arytmetyce zmiennopozycyjnej.

Zadanie polegało na sprawdzeniu słuszności stwierdzenia dla wszystkich typów zmiennopozycyjnych Float16, Float32 i Float64.

Kod źródłowy programu znajduje się w pliku zad2.jl – jako funkcja o nazwie kahan.

Wyniki działania napisanego programu:

	Eps Kahana	Eps maszynowy
Float16	- 0.000977	0.000977
Float32	1.1920929e-7	1.1920929e-7
Float64	- 2.220446049250313e-16	2.220446049250313e-16

Można zauważyć, że wynik epsilona maszynowego, w Float32, nie różni się niczym od tego obliczonego przez Kahana. Mały problem pojawia się przy wyliczonych wartościach dla Float16 i Float64, gdyż wartość bezwzględna z wyniku się zgadza, jednak znak wychodzi nie odpowiedni. Jeśli będzie się brało poprawkę na ten błąd sposób można uznać za użyteczny.

3 Rozmieszczenie liczb zmiennopozycyjnych na przedziałach

Zadanie polegało na eksperymentalnym sprawdzeniu, że w arytmetyce Float64 liczby zmiennopozycyjne są równomiernie rozmieszczone w [1, 2] z krokiem $\delta = 2^{-52}$.

Najprostszą metodą jest sprawdzenie zapisu bitowego kolejnych liczb zmiennoprzecinkowych po 1.0 używając funkcji *prevfloat*, *nextfloat* i *bitstring.*

	Liczby z przedziału [1, 2]
1.0	001111111111|0000000000000000000000000000000000000000000000000000
	001111111111|0000000000000000000000000000000000000000000000000001
	001111111111|0000000000000000000000000000000000000000000000000010
	001111111111|0000000000000000000000000000000000000000000000000011
	001111111111|0000000000000000000000000000000000000000000000000100
	…
	001111111111|1111111111111111111111111111111111111111111111111100
	001111111111|1111111111111111111111111111111111111111111111111101
	001111111111|1111111111111111111111111111111111111111111111111110
	001111111111|1111111111111111111111111111111111111111111111111111
2.0	010000000000|0000000000000000000000000000000000000000000000000000

Jak widać liczby zmiennoprzecinkowe w arytmetyce Float64, są równooddalone od siebie, odległość między nimi można policzyć licząc liczbę kombinacji bitów mantysy, która ma 52 bity. Daje nam to krok 2^{-52}. Można zauważyć, że każdą kolejną liczbę można obliczyć rekurencyjnie: następna = poprzednia + 1

3.1 Jak rozmieszczone są liczby zmiennopozycyjne w przedziale [0.5, 1]?

Spróbujmy podejścia jak dla poprzedniego przedziału.

	Liczby z przedziału [0.5, 1]
0.5	001111111110|0000000000000000000000000000000000000000000000000000
	001111111110|0000000000000000000000000000000000000000000000000001
	001111111110|0000000000000000000000000000000000000000000000000010
	001111111110|0000000000000000000000000000000000000000000000000011
	…
	001111111110|1111111111111111111111111111111111111111111111111110
	001111111110|1111111111111111111111111111111111111111111111111111
1.0	001111111111|0000000000000000000000000000000000000000000000000000

Jak widać rozmiar mantysy się nie zmienił i mamy do wykorzystania 52 bity, czyli liczba możliwych liczb zmiennoprzecinkowych do przedstawienia jest taka sama jak na

przedziale [1, 2], jednak rozmiar przedziału jest dwa razy mniejszy co znaczy, że gęstość liczb jest dwa razy większa. Daje to krok $\frac{2^{-52}}{2} = 2^{-53}$.

3.2 Jak rozmieszczone są liczby zmiennopozycyjne w przedziale [2, 4]?

	Liczby z przedziału [2, 4]
2.0	010000000000|0000000000000000000000000000000000000000000000000000
	010000000000|0000000000000000000000000000000000000000000000000001
	010000000000|0000000000000000000000000000000000000000000000000010
	010000000000|0000000000000000000000000000000000000000000000000011
	...
	010000000000|1111111111111111111111111111111111111111111111111110
	010000000000|1111111111111111111111111111111111111111111111111111
4.0	010000000001|0000000000000000000000000000000000000000000000000000

Znów mamy sytuację analogiczną do poprzednich dwóch przykładów. Liczb rzeczywistych zmiennoprzecinkowych na przedziale [2, 4] jest tyle samo co na przedziałach [1, 2] i [0.5, 1], natomiast przedział jest dwa razy większy, co przekłada się na dwa razy mniejszą gęstość. Daje to krok $2^{-52} \cdot 2 = 2^{-51}$.

4 Wyznaczanie liczby z przedziału (1, 2), takiej że $x \cdot \left(\frac{1}{x}\right) \neq 1$

Zadanie polegało na eksperymentalnym znalezieniu najmniejszej liczby zmiennopozycyjnej w przedziale (1, 2), w arytmetyce Float64 spełniającej warunek:

$$\mathrm{x} \cdot \left(\frac{1}{\mathrm{x}}\right) \neq 1$$

Kod źródłowy programu znajduje się w pliku zad4.jl – jako funkcja o nazwie *zad4.*

Idea algorytmu:

- Ustawiamy x na wartość 1.
- Dopóki warunek wymieniony wyżej nie zostanie spełniony dodajemy wartość epsilona maszynowego do x.
- Pierwsza liczba, która spełni warunek jest naszą szukaną liczbą.

Okazuje się, że najmniejszą taką liczbą zmiennoprzecinkową jest:

1.000000057228997

To, że taka liczba istnieje spowodowane jest tym, że operacje na liczbach zmiennoprzecinkowych są obarczone błędem przybliżenia, bo nie każdą liczbę rzeczywistą można przedstawić na określnej liczbie bitów w systemie binarnym. Dlatego pracując na liczbach zmiennoprzecinkowych zawsze trzeba mieć na uwadze

możliwość nawarstwienia się tych błędów, co może skutkować problemami takimi jak ten.

5 Porównywanie skuteczności liczenia iloczynów skalarnych wektorów w zależności od sposobu sumowania

Zadanie polegało na zaimplementowaniu programów liczących iloczyn skalany dwóch wektorów:

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

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

Trzeba było napisać poniższe algorytmy, licząc sumę na cztery sposoby:

(a) "w przód" $\sum_{i=1}^{n} x_i y_i$, tj. algorytm

S := 0
for i := 1 to n do
S := S + $x_i \cdot y_i$
end for

(b) "w tył" $\sum_{i=1}^{n} x_i y_i$, tj. algorytm

S := 0
for i := n downto 1 do
S := S + $x_i \cdot y_i$
end for

(c) od największego do najmniejszego (dodaj dodatnie liczby w porządku od największego do najmniejszego, dodaj ujemne liczby w porządku od najmniejszego do największego, a następnie daj do siebie obliczone sumy częściowe)

(d) od najmniejszego do największego (przeciwnie do metody (c)).

Mój program działa dla wektora o długości n. Wybrany wektor należy edytować w kodzie, który znajduje się w pliku zad5.jl – jako funkcja o nazwie *zad5.*

Wyniki działania napisanego programu:

	Wyniki w Float32	Wyniki w Float64
Algorytm (a)	- 0.4999443	1.0251881368296672e-10
Algorytm (b)	- 0.4543457	- 1.5643308870494366e-10
Algorytm (c)	- 0.5	0.0
Algorytm (d)	- 0.5	0.0

Prawidłowa wartość (przy dokładności do 15 cyfr) wynosi:

$$-1.00657107000000 \cdot 10^{-11}.$$

Jak widzimy żadne z sumowań nie daje nam niestety całkowitej zgodności. Zaokrąglenia, które robił komputer okazują się za duże. Jednak możemy zauważyć, że większa precyzja (Float64) pozwala uzyskać nam dokładniejszy wynik ze względu na większą mantysę.

Ważne co należy wywnioskować z tego zadania to fakt, że w arytmetyce zgodnej z IEEE 754 dodawanie nie jest przemienne, a kolejność wykonywanych obliczeń może znacząco wpłynąć na wynik. Należy unikać odejmowaniu od siebie liczb bliskich siebie, bo powoduje to duże błędy.

6 Porównywanie wartości dwóch równych funkcji dla małych x

Zadanie polegało na porównaniu ze sobą wartości następujących funkcji

$$f(x) = \sqrt{x^2 + 1} - 1$$

$$g(x) = \frac{x^2}{\sqrt{x^2 + 1} + 1}$$

dla kolejnych wartości argumentu $x = 8^{-1}, 8^{-2}, \ldots$ w arytmetyce Float64.

Kod źródłowy programu znajduje się w pliku zad6.jl – jako funkcja o nazwie *zad6.*

Wyniki działania napisanego programu:

Wartość 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	7.275957614156956e-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

8^{-16}	0.0	6.310887241768095e-30
8^{-17}	0.0	9.860761315262648e-32
8^{-18}	0.0	1.5407439555097887e-33
8^{-19}	0.0	2.407412430484045e-35
8^{-20}	0.0	3.76158192263132e-37
8^{-21}	0.0	5.877471754111438e-39
8^{-22}	0.0	9.183549615799121e-41
8^{-23}	0.0	1.4349296274686127e-42
8^{-24}	0.0	2.2420775429197073e-44

Jak można zauważyć w tabeli, wyniki dla danych x są różne mimo, że funkcje są takie same. Im wartość x jest mniejsza, tym większe są różnice między wynikami funkcji *f(x)* i *g(x)*.

6.1 Chociaż f = g komputer daje różne wyniki. Które z nich są wiarygodne, a które nie?

Od wartości argumentu równej 8^{-9} wartość funkcji *f(x)* jest równa 0. Jest to spowodowane tym, że człon $\sqrt{x^2+1}$ jest bardzo zbliżony do 1, dla małych x. Tu widać, że należy unikać odejmowania liczb biskich sobie, bo zaokrąglenie wyliczone przez komputer sprawiło, że $\sqrt{x^2+1}-1 = 0$

Dlatego *g(x)* jest bardziej wiarygodna, gdyż nie jest przeprowadzane odejmowanie liczb zbliżonych do siebie.

Na przyszłość lepiej tak przekształcać funkcję, żeby uniknąć takiego odejmowania. Poprawność obliczeń mogłaby się również znacząco poprawić, gdyby liczba cyfr znaczących, podczas wykonywania działań, była zbliżona.

7 Porównanie przybliżonej pochodnej z dokładną

To zadanie polegało na obliczeniu przybliżenia pochodnej używając wzoru:

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

dla funkcji $f(x) = sin\,x + cos\,3x$ w punkcie $x_0 = 1$ oraz błędów $\left|f'(x_0) - \tilde{f}'(x_0)\right|$ porównując dla dokładnej wartości pochodnej dla $h = 2^{-n}$ (n = 0, 1, 2, …, 54).

Kod źródłowy programu znajduje się w pliku zad7.jl – jako funkcja o nazwie *zad7.*

Wyniki działania napisanego programu:

Dokładna (przy dokładności 16 cyfr po przecinku) wartość pochodnej:

0.11694228168853815

n	1 + h	Błąd	$\tilde{f}'(x_0)$
0	2.0	1.9010469435800585	2.0179892252685967
1	1.5	1.753499116243109	1.8704413979316472
2	1.25	0.9908448135457593	1.1077870952342974
3	1.125	0.5062989976090435	0.6232412792975817
4	1.0625	0.253457784514981	0.3704000662035192
5	1.03125	0.1265007927090087	0.24344307439754687
6	1.015625	0.0631552816187897	0.18009756330732785
7	1.0078125	0.03154911368255764	0.1484913953710958
8	1.00390625	0.015766832591977753	0.1327091142805159
9	1.001953125	0.007881411252170345	0.1248236929407085
10	1.0009765625	0.0039401951225235265	0.12088247681106168
11	1.00048828125	0.001969968780300313	0.11891225046883847
12	1.000244140625	0.0009849520504721099	0.11792723373901026
…	…	…	…
24	1.0000000596046448	2.394961446938737e-7	0.11694252118468285
25	1.0000000298023224	1.1656156484463054e-7	0.116942398250103
26	1.0000000149011612	5.6956920069239914e-8	0.11694233864545822
27	1.0000000074505806	3.460517827846843e-8	0.11694231629371643
28	1.0000000037252903	4.802855890773117e-9	0.11694228649139404
29	1.0000000018626451	5.480178888461751e-8	0.11694222688674927
30	1.0000000009313226	1.1440643366000813e-7	0.11694216728210449
31	1.0000000004656613	1.1440643366000813e-7	0.11694216728210449
…	…		
42	1.0000000000002274	0.0002430629385381522	0.11669921875
43	1.0000000000001137	0.0007313441885381522	0.1162109375
44	1.0000000000000568	0.0002452183114618478	0.1171875
45	1.0000000000000284	0.003661031688538152	0.11328125
46	1.0000000000000142	0.007567281688538152	0.109375
47	1.000000000000007	0.007567281688538152	0.109375
48	1.0000000000000036	0.023192281688538152	0.09375
49	1.0000000000000018	0.008057718311461848	0.125
50	1.0000000000000009	0.11694228168853815	0.0
51	1.0000000000000004	0.11694228168853815	0.0
52	1.0000000000000002	0.6169422816885382	-0.5
53	1.0	0.11694228168853815	0.0
54	1.0	0.11694228168853815	0.0

Można zauważyć, że dla dużego h wynik jest bardzo niedokładny, jego dokładność poprawia się, aż do momentu szczytowego, dla $h = 2^{-28}$, gdzie wartość błędu jest jedynie rzędu 10^{-9} . Następnie błąd naszego przybliżenia rośnie.

7.1 Jak wytłumaczyć, że od pewnego momentu zmniejszanie wartości *h* nie poprawia przybliżenia wartości pochodnej? Jak zachowują się wartości *1+h*?

Z matematycznego punktu widzenia im bardziej *h* zbliża się do 0, wynik powinien być dokładniejszy. Nie dzieje się tak, przez to że operujemy w arytmetyce IEEE 754 i błąd spowodowany przybliżaniem *h* przewyższa korzyści zbliżania się go do zera. Wpływ na pogarszanie się wyniku ma także znowu odejmowanie bliskich sobie wartości liczbowych. Błąd się powiększa, aż do momentu, gdy $x_o + h = x_o$ (błąd podobny do zadania 4).