Sprawozdanie

Obliczenia naukowe, lista 1

Aleksandra Malinowska, WPPT INF, semestr 5, październik 2019

1. Wstęp

Sprawozdanie to zawiera analizę problemów oraz wyniki ich rozwiązań z zadań z listy nr 1 profesora Pawła Zielińskiego z przedmiotu Obliczenia naukowe. Wszystkie programy potrzebne do rozwiązania zadań zostały napisane w języku Julia.

2. Lista zadańe

2.1. Zadanie 1

2.1.1. Epsilon maszynowy

2.1.1.1. Opis problemu i rozwiązanie

Pierwszym problemem w zadaniu jest eksperymentalne sprawdzenie, jaki jest epsilon maszynowy dla arytmetyk zmiennopozycyjnych Float16, Float32 oraz Float64. Do wykonania tego doświadczenia napisałam funkcję `findMacheps(type)`, która iteracyjnie sprawdza kolejne potencjalne wartości poprzez obliczenie nierówności warunkującej epsilon.

2.1.1.2. Wyniki i wnioski

Wyniki eksperymentu wraz z porównaniem do prawidłowych wartości znajdują się w tabeli poniżej.

Arytmetyka	**Float16**	**Float32**	**Float64**
findMacheps(type)	`0.000977`	`1.1920929e-7`	`2.220446049250313e-16`
eps(type)	`0.000977`	`1.1920929e-7`	`2.220446049250313e-16`
float.h	`-`	`1.192093e-7`	`2.220446e-16`

Doświadczenie dowodzi, że algorytm znajdowania liczby `macheps` jest poprawny. Znaleziony w ten sposób epsilon maszynowy jest jednocześnie precyzją danej arytmetyki.

2.1.2. Eta

2.1.2.1. Opis problemu i rozwiązanie

Kolejnym problemem w zadaniu jest znalezienie liczby `eta` dla kolejnych arytmetyk. Aby przeprowadzić eksperyment szukający tej liczby, napisałam funkcję `findEta(type)`, która iteracyjnie sprawdza, czy kolejne liczby następujące po 0 spełniają warunek.

2.1.2.2. Wyniki i wnioski

Wyniki działania programu wraz z porównaniem do rzeczywistych wartości znajdują się w poniższej tabeli.

Arytmetyka	**Float16**	**Float32**	**Float64**
findEta(type)	`6.0e-8`	`1.0e-45`	`5.0e-324`
nextFloat(0.0)	`6.0e-8`	`1.0e-45`	`5.0e-324`
MIN_{sub}	`5.96e-8`	`1.4e-45`	`4.9e-324`

W tabeli wyraźnie widać, że stworzony przeze mnie algorytm poprawnie znajduję liczbę `eta`. Liczba ta jest maszynową reprezentacją liczby $MIN_{sub} = 2^{-(t-1)}2^{Cmin}$.

2.1.3. Floatmin

W poniższej tabeli znajduje się porównanie liczby `floatmin()` z wartością MIN_{nor} w zależności od arytmetyki.

Arytmetyka	**Float32**	**Float64**
floatmin(type)	`1.1754944e-38`	`2.2250738585072014e-308`
MIN_{nor}	`1.2e-38`	`2.2e-308`

Wyraźnie widać, że funkcja `floatmin()` zwraca najmniejszą liczbę normalną w danej arytmetyce, definiowaną przez MIN_{nor}.

2.1.4. Floatmax

2.1.4.1. Opis problemu i rozwiązanie

Ostatnim problemem w tym zadaniu jest znalezienie największej liczby w danej arytmetyce. Do wykonania tego zadania napisałam program, który iteracyjnie wyznaczył największą możliwą liczbę całkowitą, a następnie powiększył o jej największą możliwą część dziesiętną.

2.1.4.2. Wyniki i wnioski

W tabeli poniżej znajdują się wyniki działania programu zestawione z rzeczywistymi wartościami.

Arytmetyka	**Float16**	**Float32**	**Float64**
findMax(type)	`6.55e4`	`3.4028235e38`	`1.7976931348623157e308`
floatmax(type)	`6.55e4`	`3.4028235e38`	`1.7976931348623157e308`
float.h	`-`	`3.402823e38`	`1.797693e308`

Dane zgromadzone w powyższej tabeli dowodzą, że algorytm jest poprawny.

2.2. Zadanie 2

2.2.1. Opis problemu i rozwiązanie

Problemem tego zdania jest sprawdzenie, czy twierdzenie Kahana o sposobie wyliczania epsilonu maszynowego jest prawdziwe. Stwierdził on, że jest to wynik wyrażenia $3(\frac{4}{3} - 1) - 1$ i jest prawidłowy dla każdego typu zmiennopozycyjnego. Aby to sprawdzić napisałam prostą funkcję `countKahan(type)`, która oblicza wartość tego wyrażenia w zależności od typu.

2.2.2. Wyniki i wnioski

Wyniki działania mojego programu wraz z porównaniem do rzeczywistych wartości znajdują się w poniższej tabeli.

Typ	**Float16**	**Float32**	**Float64**
countKahan(type)	`-0.000977`	`1.1920929e-7`	`-2.220446049250313e-16`
eps(type)	`0.000977`	`1.1920929e-7`	`2.220446049250313e-16`

Jak widać w tabeli, wartości bezwzględne otrzymanych wartości są identyczne względem typu. Jednak dla typów Float16 i Float64 wyniki różnią się znakiem. Dlatego też nasuwa się wniosek, że twierdzenie Kahana jest prawdziwe tylko dla arytmetyki pojedynczej precyzji.

2.3. Zadanie 3

2.3.1. Opis problemu i rozwiązanie

W tym zadaniu należało sprawdzić równomierność rozmieszczenia liczb w przedziale `[1,2]`, a następnie w przedziałach `[0.5, 1]` i `[2,4]` w arytmetyce Float64. W treści zadania jest napisane, żeby sprawdzić, czy odstępem między kolejnymi liczbami w przedziale `[1,2]` jest liczba $\delta = 2^{-52}$. Sposobem na łatwe sprawdzenie tego jest iteracyjne wyświetlanie kolejnych (tzn. oddalonych o δ) liczb z tego przedziału w postaci bitowej (dzięki użyciu funkcji `bitstring()`) zaczynając od początku przedziału, czyli w tym wypadku liczby 1, następnie 0.5 a następnie 2.

2.3.2. Wyniki

W poniższej tabeli znajdują się wyniki działania programu, który wyświetla pierwsze dziesięć liczb od początku przedziału oddalone od siebie o $\delta = 2^{-52}$.

Początek przedziału	**1**	$\mathbf{\frac{1}{2}}$	**2**
	`00111111111100...0000000`	`00111111111000...0000000`	`010000000000...0000000`
	`00111111111100...0000001`	`00111111111000...0000010`	`010000000000...0000000`
	`00111111111100...0000010`	`00111111111000...0000100`	`010000000000...0000001`
	`00111111111100...0000011`	`00111111111000...0000110`	`010000000000...0000010`
	`00111111111100...0000100`	`00111111111000...0001000`	`010000000000...0000010`
	`00111111111100...0000101`	`00111111111000...0001010`	`010000000000...0000010`
	`00111111111100...0000110`	`00111111111000...0001100`	`010000000000...0000011`
	`00111111111100...0000111`	`00111111111000...0001110`	`010000000000...0000100`
	`00111111111100...0001000`	`00111111111000...0010000`	`010000000000...0000100`
	`00111111111100...0001001`	`00111111111000...0010010`	`010000000000...0000100`
	`00111111111100...0001010`	`00111111111000...0010100`	`010000000000...0000101`

W pierwszej kolumnie tabeli widać, że liczby wypisane przez program są kolejnymi liczbami maszynowymi w tym przedziale. W kolejnej kolumnie, czyli dla przedziału od 0.5 widać, że występują już liczby tylko parzyste, a zatem co druga liczba. Prosty wniosek to konieczność zmniejszenia kroku między poprzez podzielenie go na 2, zatem dla przedziału `[0.5, 1]` $\delta = 2^{-53}$. Po doświadczalnym sprawdzeniu tego wniosku w programie, można stwierdzić, że był on prawdziwy.

W ostatniej kolumnie natomiast widać, że liczby powtarzają się, co sugeruje, że odstęp między liczbami jest zbyt mały, dlatego należy go dwukrotnie zwiększyć. Ponownie eksperyment, w którym $\delta = 2^{-51}$, potwierdził słuszność tego wniosku.

Z powyższych doświadczeń oraz faktu z podręcznika „Analiza numeryczna” (D. Kincaid, W. Cheney):

> *„Rozkład liczb zmiennopozycyjnych w komputerze jest nierównomierny. Między kolejnymi potęgami dwójki znajduje się tyle samo liczb maszynowych.”*

można wyprowadzić wniosek, że liczby zmiennopozycyjne w komputerze w przedziale `[a,b]` są oddalone od siebie zgodnie ze wzorem:

$$\delta = 2^{-52 + \lfloor \log_2 a \rfloor}$$

gdzie a jest początkiem przedziału, a $b \leq 2^{\lfloor \log_2 a \rfloor + 1}$.

2.4. Zadanie 4

2.4.1. Opis problemu i rozwiązanie

W zadaniu należało znaleźć liczbę z przedziału `[1,2]`, która nie spełnia równania $x * \frac{1}{x} = 1$. Następnie należało znaleźć najmniejszą taką liczbę w arytmetyce Float64. Sposobem na rozwiązanie tego problemu jest program, który zaczynając od początku przedziału, bierze kolejne liczby (dzięki funkcji `nextfloat(x)`) i sprawdza, czy spełniają równanie.

2.4.2. Wyniki i wnioski

Wynik działania tego programu znajduje się w poniższej tabeli.

Przedział	**Wynik**
$\langle 1, 2 \rangle$	`1.000000057228997`
$(-\infty, \infty)$	`-1.7976931348623157 * 10^308`

W obu przypadkach wynikiem są liczby, których odwrotności są zbyt małe, by zapisać je w podwójnej precyzji, dlatego też liczby są zaokrąglane, skąd wynikają błędy w obliczeniach.

2.5. Zadanie 5

2.5.1. Opis problemu i rozwiązanie

Problemem tego zadania było obliczenie iloczynu skalarnego dwóch wektorów czterema sposobami podanymi w treści i porównanie wyników z prawidłową wartością. Aby to zrobić napisałam cztery funkcje, każda odpowiadająca jednej z metod, które liczą sumę iloczynów kolejnych współrzędnych.

2.5.2. Wyniki i wnioski

Wyniki eksperymentu, polegającego na uruchomieniu wyżej wspomnianych funkcji w dwóch arytmetykach (Float32 i Float64), znajdują się w poniższej tabeli.

Metoda	**Float32**	**Float64**
1 („w przód”)	`-0.4999443`	`1.0251881368296672e-10`
2 („w tył”)	`-0.4543457`	`-1.5643308870494366e-10`
3	`-0.5`	`0.0`
4	`-0.5`	`0.0`

Prawidłowym wynikiem tego działania jest -1.00657107000000 · 10^{-11}, zatem żadna z metod nie pozwoliła na dokładne policzenie wyniku. Prostym wnioskiem jest, że zadanie jest źle uwarunkowane. Wymaga ono mnożenia liczb bardzo oddalonych od siebie, a następnie dodawania ich do siebie, co prowadzi do nawarstwiania się kolejnych błędów w przybliżeniach i skutkuje niepoprawnym wynikiem, w szczególności w arytmetyce pojedynczej precyzji.

2.6. Zadanie 6

2.6.1. Opis problemu i rozwiązanie

W tym zadaniu należało sprawdzić, jak komputer policzy wartość tej samej funkcji przedstawionej w dwóch postaciach. Rozwiązaniem jest program, który oblicza wartości funkcji dla kolejnych wartości dwoma sposobami, a następnie porównuje je ze sobą.

2.6.2. Wyniki i wnioski

Poniżej przedstawiam wyniki wywołania funkcji dla kolejnych wartości.

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^{-8}	1.7763568394002505e-15	1.7763568394002489e-15
8^{-9}	0.0	2.7755575615628914e-17
...	...	...
8^{-178}	0.0	1.6e-322
8^{-179}	0.0	0.0
...	0.0	0.0

Wyraźnie widać, że funkcja `g(x)`, przedstawiona w postaci ułamka, jest dużo bardziej precyzyjna od funkcji w postaci `f(x)`. Prawidłowe wartości funkcji dla zadanych `x`, które w każdej kolejnej iteracji maleją ośmiokrotnie, to liczby bardzo małe i w każdej iteracji malejące. Dlatego też, w przypadku funkcji w postaci `f(x)` zakres szybko się kończy, ponieważ ostatnią operacją jest odjęcie 1 od liczby postaci `(1.0.....)` , po czym następuje zaokrąglenie. W przypadku funkcji postaci `g(x)` ostatnim działaniem jest dzielenie liczby, zatem mała liczba, która jest wynikiem tego dzielenia może być od razu poprawnie zapisana na całej długości mantysy. Stąd też wniosek, że wartości otrzymywane w wyniku działania funkcji `g(x)` są bardziej wiarygodne.

2.7. Zadanie 7

2.7.1. Opis problemu i rozwiązanie

W tym zadaniu należało porównać wyniki obliczania pochodnej w punkcie z dwóch wzorów. Do wykonania tego zadania napisałam program, który iteracyjnie obliczał kolejne (przybliżone) wartości pochodnej w punkcie, w zależności od zmiennej h.

2.7.2. Wyniki i wnioski

Wyniki eksperymentu przedstawiam w poniższej tabeli.

h	**~f'(1)**	**|f'(1) - ~f'(1)|**
2^{0}	2.0179892252685967	1.9010469435800585
2^{-1}	1.8704413979316472	1.753499116243109
2^{-2}	1.1077870952342974	0.9908448135457593
...	…	…
2^{-27}	0.11694231629371643	3.460517827846843e-8
2^{-28}	0.11694228649139404	4.802855890773117e-9
2^{-29}	0.11694222688674927	5.480178888461751e-8
...	…	…
2^{-54}	0.0	0.11694228168853815
f'(1) = 0.11694228168853815		

Z powyższych danych wyraźnie wynika, że zmniejszanie odległości h od punktu x0 = 1 do pewnego momentu powoduje poprawę dokładności przybliżenia pochodnej, jednak od wartości $h = 2^{-29}$ ponownie następuje jej spadek. Wynika to prawdopodobnie z konieczności wykonywania działań na bardzo małych liczbach, co prowadzi do zaokrąglania wyników i powiększania się błędu.

W zadaniu należało również sprawdzić, jak zachowują się wartości 1+h. W tabeli poniżej widać wyniki tego eksperymentu.

n (h = 2^{-n})	**h**	**1 + h**
0	1.0	2.0
1	0.5	1.5
2	0.25	1.25
...	...	...
16	1.52587890625e-5	1.0000152587890625
17	7.62939453125e-6	1.0000076293945312
18	3.814697265625e-6	1.0000038146972656
...	...	...
51	4.440892098500626e-16	1.0000000000000004
52	2.220446049250313e-16	1.0000000000000002
53	1.1102230246251565e-16	1.0
54	5.551115123125783e-17	1.0

Tu wyraźnie widać, w którym momencie zaczynają pojawiać się zaokrąglenia powodujące błąd. We wzorze na przybliżenie pochodnej funkcji występuje składnik $f(x_0 + h)$, który w tym wypadku wynosi $f(1 + h)$, stąd wniosek, że h nie może być liczbą zbyt małą ze względu na ograniczoną długość mantysy. Gdy h będzie liczbą względnie małą w stosunku do x_0, wtedy rozwinięcie dziesiętne sumy $x_0 + h$ może nie zmieścić się w mantysie.

3. Podsumowanie

Przeprowadzone eksperymenty miały na celu wyszczególnienie błędów obliczeniowych powstałych w wyniku zapisywania liczb rzeczywistych w ograniczonej długością arytmetyce zmiennopozycyjnej zgodnej ze standardem IEEE 754. Nieświadomość powstawania tych błędów w komputerze może grozić poważnymi błędami przy programowaniu wymagającym dokładnych obliczeń.