Sprawozdanie

Obliczenia naukowe 2019 – lista 1

Tomasz Janik, WPPT INF, sem. 5

1. Wstęp

Jest to sprawozdanie, w którym zaprezentowane są rozwiązania zadań z pierwszej listy z kursu Obliczenia naukowe oraz wnioski jakie zostały z nich wyciagnięte. Zadania zostały rozwiązane z pomocą programów napisanych w języku Julia.

2. Rozwiązania zadań

2.1. Zadanie 1

2.1.1. Epsilon maszynowy

2.1.1.1. Opis problemu i rozwiązanie

Na początku naszym zadaniem było iteracyjne wyznaczenie epsilonu maszynowego. Dla arytmetyk Float16, Float32, Float64. W tym celu została napisana funkcja macheps(), która sprawdza zadany warunek episilonu dla kolejnych wartości, startując od 1, a następnie dzieląc wartość przez 2.

2.1.1.2. Wyniki i wnioski

Poniżej w tabeli przedstawiono wyniki zadania oraz wartości prawidłowe.

Arytmetyka	*Float16*	*Float32*	*Float64*
macheps()	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

Jak można zauważyć funkcja prawidłowo wyznaczyła epsilon maszynowy. Owa liczba jest też podwojoną precyzją danej arytmetyki.

2.1.2. Eta

2.1.2.1. Opis problemu i rozwiązanie

Następnie należało znaleźć liczbę eta dla tych samych arytmetyk. W tym celu została napisana funkcja eta(), która sprawdza zadany warunek eta dla kolejnych wartości, startując od 1, a następnie dzieląc wartość przez 2.

2.1.2.2. Wyniki i wnioski

Poniżej w tabeli przedstawiono wyniki zadania oraz wartości prawidłowe.

Arytmetyka	*Float16*	*Float32*	*Float64*
eta()	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

Jak można zauważyć funkcja prawidłowo wyznaczyła liczbę eta. Liczba ta jest maszynową reprezentacją $MIN_{sub} = 2^{-(t-1)}2^{Cmin}$.

2.1.3. Floatmin

Poniżej w tabeli znajduje się porównanie liczby floatmin() z wartością MIN_{nor} dla danych arytmetyk

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

Można zauważyć, że funkcja floatmin() zwraca najmniejszą liczbę normalną w danej arytmetyce, czyli MIN_{nor}.

2.1.4. Floatmax

2.1.4.1. Opis problemu i rozwiązanie

Na koniec należało znaleźć największą liczbę w danej arytmetyce. W tym celu została napisana funkcja max(), która wyznacza iteracyjnie maksymalną część całkowitą, a następnie część dziesiętną.

2.1.4.2. Wyniki i wnioski

Poniżej w tabeli znajdują się wyniki działania programu zestawione z prawidłowymi wartościami.

Arytmetyka	*Float16*	*Float32*	*Float64*
Max()	6.55e4	3.4028235e38	1.7976931348623157e308
floatmax(type)	6.55e4	3.4028235e38	1.7976931348623157e308
MAX z wykładu	-	3.4e38	1.8e308

Jak można zauważyć funkcja prawidłowo wyznaczyła liczbę MAX.

2.2. Zadanie 2

2.2.1. Opis problemu i rozwiązanie

W tym zadaniu należało poprawność twierdzenia Kahana o sposobie wyliczania epsilonu maszynowego. Stwierdził, że epsilon maszynowy można otrzymać obliczając wyrażenie $3(\frac{4}{3} - 1) - 1$ w arytmetyce zmiennopozycyjnej. W tym celu została napisana funkcja kahan(), która oblicza to wyrażenie w 3 arytmetykach, Float16, Float32 oraz Float64.

2.2.2. Wyniki i wnioski

Poniżej w tabeli znajdują się wyniki działania programu zestawione z prawidłowymi wartościami.

Arytmetyka	*Float16*	*Float32*	*Float64*
kahan()	-0.000977	1.1920929e-7	-2.220446049250313e-16
eps(type)	0.000977	1.1920929e-7	2.220446049250313e-16

Jak widać w tabeli, dla typów Float16 oraz Float64 uzyskaliśmy wartości ujemne. Stąd można wywnioskować, że twierdzenie Kahana jest poprawne, jeśli popatrzymy na wartość bezwzględną otrzymanego wyniku.

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. Na początku należy sprawdzić czy w przedziale [1,2] liczby są oddalone o $\delta = 2^{-52}$. Możemy to zrobić wyświetlając kilka kolejnych liczb w postaci bitowej, przy użyciu funkcji bitstring(), zwiększając je o δ. Możemy zacząć od początków przedziałów, zatem kolejno od 1, $\frac{1}{2}$, oraz 2.

2.3.2. Wyniki

Poniżej w tabeli znajduje się pierwsze kilka liczb od początku przedziału oddalone od siebie o $\delta = 2^{-52}$.

Początek przedziału	*1*	$\frac{1}{2}$	*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

Jak widać w pierwszej kolumnie tabeli otrzymujemy kolejne liczby maszynowe, zatem krok ten jest w rzeczywistości odległością między dwoma kolejnymi liczbami.

W kolejnej kolumnie widać, że przeskakujemy od razu co 2 liczby, zatem wnioskujemy, że krok jest w tym przedziale 2 razy mniejszy i równy 2^{-53}.

W ostatniej kolumnie natomiast widać, że potrzebujemy 2 kroków, aby otrzymać kolejną liczbę, zatem krok w tym przedziale wynosi $2 * 2^{-52} = 2^{-51}$.

Wynika to z tego, gdyż z każdą kolejną potęgą dwójki zmienia się cecha liczby, lecz mantysa zawsze musi być z przedziału [1,2), zatem ogólny wzór na rozkład liczb w zdanym przedziale [a,b] wynosi:

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

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 $x * \frac{1}{x} = 1$ w arytmetyce Float64. Następnie należało znaleźć najmniejszą taką liczbę. W tym celu zostały napisane funkcje, które sprawdzają równanie dla kolejnych liczb startując w pierwszym przypadku od 1, a w drugim od floatmin(Float64).

2.4.2. Wyniki i wnioski

Wynik działania tego programu znajduje się w poniższej tabeli.

Przedział	*Wynik*
$[\mathbf{1}, \mathbf{2}]$	1.000000057228997
$(\mathbf{0}, \infty)$	2.2250738585072014e-308

Obecny błąd obliczeń wynika z tego, że odwrotność liczb jest zaokrąglana, a następnie to zaokrąglenie przenosi się na błąd wyniku całego wyrażenia.

2.5. Zadanie 5

2.5.1. Opis problemu i rozwiązanie

W tym zadaniu należało obliczyć iloczyn skalarny dwóch zadanych wektorów 4 metodami, a następnie sprawdzić z rzeczywistym wynikiem. W tym celu zostały napisane funkcje liczące tymi metodami w arytmetyce Float32, a następnie we Float64.

2.5.2. Wyniki i wnioski

W poniżej tabeli zaprezentowane zostały wyniki obliczeń funkcji.

Metoda	*Float32*	*Float64*
a („w przód”)	-0.4999443	1.0251881368296672e-10
b („w tył”)	-0.4543457	-1.5643308870494366e-10
c	-0.5	0.0
d	-0.5	0.0
Wartość rzeczywista	$-1.00657107000000 \cdot 10^{-11}$	

Jak można zauważyć żadna z metod nie pozwoliła na dokładne policzenie wyniku. Można z tego wywnioskować, że zadanie jest źle uwarunkowane i przez mnożenie liczb znacznie oddalonych od siebie, a następnie ich dodawanie, generuje błędy zaokrągleń, które się nawarstwiają, co z kolei prowadzi do błędnego wyniku.

2.6. Zadanie 6

2.6.1. Opis problemu i rozwiązanie

W tym zadaniu należało sprawdzić wyniki obliczeń w arytmetyce Float64 tej samej funkcji, zapisanej w dwóch różnych postaciach. W tym celu napisana została funkcja, która oblicza obie formuły dla zadanych argumentów i wypisuje wyniki.

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

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

2.6.2. Wyniki i wnioski

W poniżej tabeli znajdują się wyniki funkcji w wybranych iteracjach.

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

Jak łatwo można zauważyć funkcja g(x) oferuje znacznie większą precyzję od funkcji f(x), która to już przy argumencie 8^{-9} daje wynik równy 0,a funkcja g dopiero przy 8^{-179}. Różnica wynika z tego, że w pierwszej postaci odejmujemy od siebie coraz to bliższe sobie liczby, ponieważ wyrażenie pod pierwiastkiem zbliża się do 1 z każdą kolejną iteracją. A jak pokazaliśmy na wykładzie im bliższe od siebie odejmujemy liczby tym większy błąd otrzymamy.

Z tego możemy wywnioskować, że wyniki 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 obliczanej za pomocą zadanego wzoru z rzeczywistą jej wartością dla funkcji $sinx + cos3x$. W tym celu została napisana funkcja, która iteracyjnie oblicza kolejne przybliżone wartości pochodnej w punkcie w zależności od h.

2.7.2. Wyniki i wnioski

W poniżej tabeli zaprezentowane są wyniki obliczeń funkcji oraz rzeczywista wartość pochodnej.

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		

Jak można zauważyć, zmniejszanie h powoduje poprawę dokładności tylko do pewnego momentu. Od wartości h = 2^{-29} 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 zaprezentowane są te wartości.

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
...		
27	7.450580596923828e-9	1.0000000074505806
28	3.725290298461914e-9	1.0000000037252903
29	1.862645149230957e-9	1.0000000018626451
...		
51	4.440892098500626e-16	1.0000000000000004
52	2.220446049250313e-16	1.0000000000000002
53	1.1102230246251565e-16	1.0
54	5.551115123125783e-17	1.0

Tutaj widać kiedy zaczynają się zaokrąglenia, które powodują błędy w obliczeniach pochodnej. Kiedy h staje się wystarczająco małe, przy obliczaniu 1+h dochodzi do zaokrąglenia. Stąd wniosek, że h nie może być zbyt małą liczbą z uwagi na ograniczoną precyzję w arytmetyce.

3. Podsumowanie

Rozwiązania zadań miały na celu ukazać na jakie błędy obliczeniowe możemy natknąć się, kiedy liczby możemy zapisywać tylko z zadaną precyzją, tutaj standard IEEE 754 oraz arytmetyki Float16, Float32 i Float64. Na podstawie otrzymanych wyników można śmiało stwierdzić, że nie poświęcenie wystarczającej uwagi tej kwestii przy tworzeniu algorytmów, może doprowadzić do poważnych błędów.