Sprawozdanie z listy nr 1 Obliczenia Naukowe

Marek Świergoń (261750)

24 października 2022, PWr, WIiT INA

1 Zadanie 1

1.1 Cele zadania

W zadaniu należało wyznaczyć iteracyjnie:

- epsilon maszynowy *macheps*, czyli najmniejszą liczbę większą od zera taką, że $fl(1.0 + macheps) > 1.0$ i $fl(1.0 + macheps) = 1 + macheps$,
- liczbę maszynową *eta*, czyli najmniejszą liczbę dodatnią reprezentowaną w danym systemie zmiennoprzecinkowym,
- liczbę *MAX*, największą liczbę dodatnią reprezentowaną w danym systemie zmiennoprzecinkowym.

1.2 Rozwiązanie zadania

Metody wyznaczające iteracyjnie powyższe liczby znajdują się w pliku *zadanie1.jl*.

1.3 Wyniki

Typ danych	Wynik eksperymentalny	Wartość *eps()*	Wartość dokładna z **float.h**
Float16	0.000977	0.000977	b.d.
Float32	$1.1920929 * 10^{-7}$	$1.1920929 * 10^{-7}$	$1.192092896 * 10^{-7}$
Float64	$2.220446049250313 * 10^{-16}$	$2.220446049250313 * 10^{-16}$	$2.2204460492503131 * 10^{-16}$

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

Typ danych	Wynik eksperymentalny	Wartość zwracana przez *nextfloat()*
Float16	$6 * 10^{-8}$	$6 * 10^{-8}$
Float32	$1 * 10^{-45}$	$1 * 10^{-45}$
Float64	$5 * 10^{-324}$	$5 * 10^{-324}$

Tabela 2: Porównanie wyników ekspeymentalnego wyznaczenia *eta* z wartościami zwracanymi przez funkcję *nextfloat(TYP(0.0))*.

Typ danych	Wartość *floatmin()*	Wartość MIN_{nor}
Float32	$1.1754944 * 10^{-38}$	$1.2 * 10^{-38}$
Float64	$2.2250738585072014 * 10^{-308}$	$2.2 * 10^{-308}$

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

Typ danych	Wynik eksperymentalny	Wartość *MAX*	Wartość dokładna z **float.h**
Float16	$6.55 * 10^4$	$6.55 * 10^4$	b.d.
Float32	$3.4028235 * 10^{38}$	$3.4028235 * 10^{38}$	$3.402823466 * 10^{38}$
Float64	$1.7976931348623157 * 10^{308}$	$1.7976931348623157 * 10^{308}$	$1.7976931348623158 * 10^{308}$

Tabela 4: Porównanie wyników ekspeymentalnego wyznaczenia *MAX* z wartościami zwracanymi przez funkcję *floatmax()* oraz danymi zawartymi w pilku nagłówkowym **float.h** dowolnej instalacji języka C.

1.4 Interpretacja wyników i wnioski

Z powyższych tabeli wynika, że udało się w sposób eksperymentalny wyznaczyć dokładnie liczby *macheps*, *eta* i *MAX*. Zarówno Julia, jak i C korzystają z typów zmiennoprzecinkowych zgodnych ze standardem IEEE 754, stąd zbieżności wyników w wierszach tabel. Ponadto można wyciągnąć następujące wnioski:

- Gdy porównamy wyniki z wartościami precyzji arytmetyki ϵ podanymi na wykładzie, to możemy zauważyć następującą zależność
$$macheps = 2 * \epsilon.$$
- Wyznaczona eksperymentalnie liczba *eta* jest przybliżeniem podanej na wykładzie liczby MIN_{sub}, będącej najmniejszą dodatnią liczbą zdenormalizowaną reprezentowaną w danym typie zmiennopozycyjnym.
- Wartość podanej na wykładzie liczby MIN_{nor}, będącej najmniejszą dodatnią liczbą znormalizowaną reprezentowaną w danym typie zmiennopozycyjnym, jest w przybliżeniu równa wartościom zwracanym przez funkcję *floatmin()*.

2 Zadanie 2

2.1 Cel zadania

Sprawdzić eksperymentalnie (dla typów zmiennopozycyjnych Float16, Float32, Float64) słuszność stwierdzenia podanego przez Kahana, według którego epsilon maszynowy można wyznaczyć, obliczając wyrażenie 3(4/3-1)-1 w danej arytmetyce zmiennopozycyjnej.

2.2 Rozwiązanie

W pliku źródłowym *zadanie2.jl* znajduje się rozwiązanie polegające na zestawieniu wyniku otrzymanego z obliczenia wyrażenia (uważając na konwersję typów) podanego przez Kahana i wartości zwracanej przez funkcję *eps()*.

2.3 Wyniki

Typ danych	Wynik eksperymentalny (Kahan)	Wartość zwracana przez *eps()*
Float16	-0.000977	0.000977
Float32	$1.1920929 * 10^{-7}$	$1.1920929 * 10^{-7}$
Float64	$-2.220446049250313 * 10^{-16}$	$2.220446049250313 * 10^{-16}$

Tabela 5: Porównanie wyników ekspeymentalnego wyznaczenia *macheps* (ze wzoru Kahana) z wartościami zwracanymi przez funkcję *eps()*.

2.4 Interpretacja wyników i wnioski

Aby wyrażenie Kahana wyznaczało poprawnie epsilon maszynowy dla wszystkich typów zmiennopozycyjnych, należy nałożyć na wynik funkcję wartości bezwzględnej *abs()*. Błędy w bicie znaku wynikają

z reprezentacji rozwinięcia binarnego ułamka (4/3) (= 1.(10)), konkretniej z różnej ostatniej cyfry mantysy dla różnych typów zmiennopozycyjnych. Zatem

$$eps() = |kahaneps()|$$

3 Zadanie 3

3.1 Cele zadania

Sprawdzenie eksperymentalnie w języku Julia, że w arytmetyce Float64 liczby zmiennopozycyjne są równomiernie rozmieszczone w [1, 2] z krokiem $\delta = 2^{-52}$, tj. $x = 1 + k\delta$ w tej arytmetyce, gdzie $k = 0, 1, 2, ..., 252 - 1$. Następnie sprawdzenie, czy na przedziałach $[\frac{1}{2}, 1]$ oraz $[2, 4]$ liczby również są równomiernie rozmieszczone i jeśli tak, to z jakim krokiem δ.

3.2 Rozwiązanie

Metody służące do eksperymentalnego odkrycia sposobu rozmieszczenia liczb w zadanych przedziałach znajdują się w pliku *zadanie3.jl*. Zostały zaimplementowane dwie funkcje, jedna (*printSteps(a::Float64, b::Float64)*) w sposób bruteforce sprawdza i wyświetla wszystkie unikatowe wartości odstępów pomiędzy liczbami w przedziale [a, b]. Ze względu na swój charakter funkcja działa bardzo wolno i należy ją tylko stosować dla małych przedziałów i do orientacyjnego wyznaczenia wartości kroku. Druga funkcja (*printBits(start::Float64, step::Float64, n::Int)*) bierze jako argument liczbę startową *start*, krok *step* i liczbę iteracji *n*, następnie wyświetla z użyciem funkcji *bitstring()* zapis bitowy *n* kolejnych liczb (różniących się o zadany w parametrze krok) w arytmetyce Float64.

3.3 Wyniki

3.3.1 Przedział [1, 2]

Wyznaczony krok $\delta = 2.220446049250313080847263336181640625 * 10^{16} \approx 2^{-52}$.

Numer iteracji i	Wartość zwracana przez *bitstring(a_i)* (podzielona na znak, cechę i mantysę)
1	0 | 01111111111 | 0000000000000000000000000000000000000000000000000000
2	0 | 01111111111 | 0000000000000000000000000000000000000000000000000001
3	0 | 01111111111 | 0000000000000000000000000000000000000000000000000010
4	0 | 01111111111 | 0000000000000000000000000000000000000000000000000011
5	0 | 01111111111 | 0000000000000000000000000000000000000000000000000100
6	0 | 01111111111 | 0000000000000000000000000000000000000000000000000101
7	0 | 01111111111 | 0000000000000000000000000000000000000000000000000110
8	0 | 01111111111 | 0000000000000000000000000000000000000000000000000111
9	0 | 01111111111 | 0000000000000000000000000000000000000000000000001000
10	0 | 01111111111 | 0000000000000000000000000000000000000000000000001001

Tabela 6: Wywołanie funkcji *bistring()* dla pierwszych dziesięciu liczb w arytmetyce Float64 z przedziału $[1, 2]$ wyznaczonych wzorem $x = 1 + \delta(i - 1)$ (ponieważ numeracja iteracji od 1)

3.3.2 Przedział $[\frac{1}{2}, 1]$

Wyznaczony krok $\delta = 1.110223024625156540423631668090820312 * 10^{16} \approx 2^{-53}$.

Numer iteracji i	Wartość zwracana przez *bitstring(a_i)* (podzielona na znak, cechę i mantysę)
1	0 | 01111111110 | 0000000000000000000000000000000000000000000000000000
2	0 | 01111111110 | 0000000000000000000000000000000000000000000000000001
3	0 | 01111111110 | 0000000000000000000000000000000000000000000000000010
4	0 | 01111111110 | 0000000000000000000000000000000000000000000000000011
5	0 | 01111111110 | 0000000000000000000000000000000000000000000000000100
6	0 | 01111111110 | 0000000000000000000000000000000000000000000000000101
7	0 | 01111111110 | 0000000000000000000000000000000000000000000000000110
8	0 | 01111111110 | 0000000000000000000000000000000000000000000000000111
9	0 | 01111111110 | 0000000000000000000000000000000000000000000000001000
10	0 | 01111111110 | 0000000000000000000000000000000000000000000000001001

Tabela 7: Wywołanie funkcji *bistring()* dla pierwszych dziesięciu liczb w arytmetyce Float64 z przedziału $[\frac{1}{2}, 1]$ wyznaczonych wzorem $x = \frac{1}{2} + \delta(i-1)$ (ponieważ numeracja iteracji od 1)

3.3.3 Przedział [2, 4]

Wyznaczony krok $\delta = 4.440892098500626161694526672363281250 * 10^{16} \approx 2^{-51}$.

Numer iteracji i	Wartość zwracana przez *bitstring(x)* (podzielona na znak, cechę i mantysę)
1	0 | 10000000000 | 0000000000000000000000000000000000000000000000000000
2	0 | 10000000000 | 0000000000000000000000000000000000000000000000000001
3	0 | 10000000000 | 0000000000000000000000000000000000000000000000000010
4	0 | 10000000000 | 0000000000000000000000000000000000000000000000000011
5	0 | 10000000000 | 0000000000000000000000000000000000000000000000000100
6	0 | 10000000000 | 0000000000000000000000000000000000000000000000000101
7	0 | 10000000000 | 0000000000000000000000000000000000000000000000000110
8	0 | 10000000000 | 0000000000000000000000000000000000000000000000000111
9	0 | 10000000000 | 0000000000000000000000000000000000000000000000001000
10	0 | 10000000000 | 0000000000000000000000000000000000000000000000001001

Tabela 8: Wywołanie funkcji *bistring()* dla pierwszych dziesięciu liczb w arytmetyce Float64 z przedziału $[2, 4]$ wyznaczonych wzorem $x = 2 + \delta(i-1)$ (ponieważ numeracja iteracji od 1)

3.4 Interpretracja wyników i wnioski

Wyniki potwierdzają hipotezę postawioną w celu zadania, tj. w arytmetyce Float64 na przedziale [1,2] liczby zmiennopozycyjne są równomiernie rozmieszczone w [1, 2] z krokiem $\delta = 2^{-52}$.
Na przedziale $[\frac{1}{2}, 1]$ liczby również są rozmieszczone równomiernie, z krokiem $\delta = 2^{-53}$.
Taka sama własność zachodzi też dla liczb z przedziału [2,4], z krokiem $\delta = 2^{-51}$.
Wartości zwracane przez *bitstring()* pokazują, że liczby sąsiadujące ze sobą zgodnie ze wzorem $x = \frac{1}{2} + k\delta, k = 0, 1, 2, ..., 251$ różnią się w mantysie o jeden (gdyby mantysę traktować jako liczbę całkowitą zapisaną binarnie), zatem rzeczywiście nie istnieje żadna liczba znajdująca się pomiędzy nimi, tj. nie da się takiej liczby zapisać z takim rozmiarem mantysy. Co więcej, można wyciągnąć wniosek, że taka własność zachodzi **dla wszystkich przedziałów [a, b], gdzie $a < b$ oraz a i b są sąsiadującymi potęgami dwójki**. Wynika to z faktu, że liczby z takiego przedziału zapisane w arytmetyce Float64 mają wspólną cechę (różnią się tylko na mantysie). Wyjątkiem jest liczba prawego krańca b, która ma cechę większą o 1, ale mantysę całkowicie wyzerowaną, gdzie sąsiadująca z lewej strony liczba ma mantysę w pełni zapisaną jedynkami, dlatego jeszcze w dalszym ciągu zachowany zostaje podany wcześniej wzór.

4 Zadanie 4

4.1 Cel zadania

Znaleźć eksperymentalnie w arytmetyce Float64 (w Julii) najmniejszą liczbę zmiennopozycyjną w przedziale $1 < x < 2$ taką, że $x * (1/x) \neq 1$, tj. $fl(xfl(1/x)) \neq 1$.

4.2 Rozwiązanie

Metoda znajdująca taką liczbę znajduje się w pliku *zadanie4.jl*. Metoda zaczyna od $x = 1.0$ i sprawdza kolejno wszystkie liczby zmiennopozycyjne (z użyciem *nextfloat()*) aż do momentu, gdy $fl(xfl(1/x)) \neq 1$.

4.3 Wyniki

Najmniejszą taką liczbą jest x = 1.000000057228997, dla niej $fl(xfl(1/x)) = 0.9999999999999999$.

4.4 Wniosek

Działania w arytmetyce zmiennopozycyjnej obarczone są błędem, który należy brać pod uwagę nawet przy podstawowych operacjach.

5 Zadanie 5

5.1 Cel zadania

Obliczenie iloczynu skalarnego dwóch wektorów x i y w typach Float32 i Float64, gdzie

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

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

Iloczyn zostanie policzony z wykorzystaniem czterech algorytmów różniących się kolejnością sumowania:

1. "w przód", tj. $\sum_{i=1}^{n} x_i y_i$
2. "w tył", tj. $\sum_{i=n}^{1} x_i y_i$
3. dodając osobno dodatnie iloczyny składowe w porządku od największego do najmniejszego i ujemne iloczyny składowe w porządku od najmniejszego do największego, następnie dodając obliczone sumy częściowe
4. przeciwnie do metody 3.

5.2 Rozwiązanie

W pliku *zadanie5.jl* zostały zaimplementowane po dwie wersje każdego z algorytmów: jedna operująca na Float32, druga na Float64. W dalszej części skryptu wywoływane są metody z algorytmami, a wyniki są wyświetlane na standardowe wyjście.

5.3 Wyniki

Numer algorytmu	Wynik w Float32	Wynik w Float64
1	-0.4999443	$1.0251881368296672 * 10^{-10}$
2	-0.4543457	$-1.5643308870494366 * 10^{-10}$
3	0.0	-0.5
4	0.0	-0.5

Tabela 9: Porównanie wyniku obliczania iloczynu skalarnego dwóch wektorów z użyciem różnych algorytmów. Prawidłowy wynik to $-1.00657107000000 * 10^{-11}$.

5.4 Wniosek

Kolejność sumowania ma znaczenie, wielkość błędu poszczególnych działań różni się w zależności od zastosowanego algorytmu. Zadanie obliczenia iloczynu sklaranego dla podanych wektorów jest obarczone dużym błędem.

6 Zadanie 6

6.1 Cel zadania

Należy policzyć w języku Julia w arytmetyce Float64 wartości funkcji

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

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

dla kolejnych wartości $x = 8^{-1}, 8^{-2}, 8^{-3}, \ldots$.

6.2 Rozwiązanie

Obie funkcje zostały wprost zaimplementowane w pliku *zadanie6.jl*. Następnie w pętli funkcje są wywoływane dla $x = 8^{-1}, 8^{-2}, 8^{-3}, \ldots$, dopóki x w reprezentacji Float64 jest różne od zera.

6.3 Wyniki

x	f(x)	g(x)
8^{-1}	0.0077822185373186414	0.0077822185373187065
8^{-2}	0.00012206286282867573	0.00012206286282875901
8^{-3}	$1.9073468138230965 * 10^{-6}$	$1.907346813826566 * 10^{-6}$
8^{-4}	$2.9802321943606103 * 10^{-8}$	$2.9802321943606116 * 10^{-8}$
8^{-5}	$4.656612873077393 * 10^{-10}$	$4.6566128719931904 * 10^{-10}$
8^{-6}	$7.275957614183426 * 10^{-12}$	$7.275957614156956 * 10^{-12}$
8^{-7}	$1.1368683772161603 * 10^{-13}$	$1.1368683772160957 * 10^{-13}$
8^{-8}	$1.7763568394002505 * 10^{-15}$	$1.7763568394002489 * 10^{-15}$
8^{-9}	0.0	$2.7755575615628914 * 10^{-17}$
8^{-10}	0.0	$4.336808689942018 * 10^{-19}$
...	...	...
8^{-170}	0.0	$4.450147717014403 * 10^{-308}$
8^{-171}	0.0	$6.953355807835 * 10^{-310}$
8^{-172}	0.0	$1.086461844974 * 10^{-311}$
8^{-173}	0.0	$1.69759663277 * 10^{-313}$
8^{-174}	0.0	$2.65249474 * 10^{-315}$
8^{-175}	0.0	$4.144523 * 10^{-317}$
8^{-176}	0.0	$6.4758 * 10^{-319}$
8^{-177}	0.0	$1.012 * 10^{-320}$
8^{-178}	0.0	$1.6 * 10^{-322}$
8^{-179}	0.0	0.0

Tabela 10: Porównanie wartości zwracanych (w typie Float64) przez funkcje f i g.

6.4 Interpretacja wyników i wnioski

Należy podkreślić, że z matematycznego punktu widzenia $f = g$. Dla paru pierwszych iteracji funkcje f i g dają zbliżone wyniki, jednak dla $x \leqslant 8^{-9}$ funkcja f zaczyna zwracać wyniki znacząco odchylone od rzeczywistej wartości dla tego wyrażenia. Funkcja g zachowuje się lepiej i zwraca w miarę precyzyjnie wartości dla stosunkowo małego x (z błędem wynikającym głównie z precyzji arytmetyki).
Wiarygodne są wyniki uzyskane z funkcji g. Zauważmy, że dla $x \to 0$ $\sqrt{x^2+1} \approx 1$, zaś zadanie odejmowania liczb w przybliżeniu równych jest obarczone dużym błędem, zależnym od samych argumentów działania. Stąd wynika złe zachowanie funkcji f; w funkcji g problem ten nie występuje, dzięki prostym przekształceniom matematycznym nie jest wykonywane odejmowanie.

7 Zadanie 7

7.1 Cel zadania

Obliczenie w języku Julia w arytmetyce Float64 przybliżonej wartości pochodnej funkcji $f(x) = \sin x + \cos 3x$ w punkcie $x_0 = 1$ oraz błędów $|f'(x_0) - \tilde{f}'(x_0)|$ dla $h = 2^{-n}$ $(n = 0, 1, 2, ..., 54)$. Do wyliczenia przybliżenia należało użyć wzoru

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

.

7.2 Rozwiązanie

W pliku *zadanie7.jl* wprost zaimplementowane zostały:

- funkcja f,
- metoda przybliżająca pochodną funkcji f zgodnie z podanym wzorem i zadanym w parametrach punktem x i krokiem h,
- pochodna $f'(x) = \cos x - 3\sin 3x$, dająca dokładne wartości i będąca punktem odniesienia przy wyliczaniu błędów metody przybliżającej.

W pętli wywoływane są te funkcje dla zadanych w celu x_0 i h oraz obliczany jest błąd $|f'(x_0) - \tilde{f}'(x_0)|$.

7.3 Wyniki

h	$\tilde{f}'(x_0)$	$\lvert f'(x_0) - \tilde{f}'(x_0)\rvert$	$1+h$
2^{-0}	2.0179892252685967	1.9010469435800585	2.0
2^{-1}	1.8704413979316472	1.753499116243109	1.5
2^{-2}	1.1077870952342974	0.9908448135457593	1.25
2^{-3}	0.6232412792975817	0.5062989976090435	1.125
...	...	...	...
2^{-26}	0.11694233864545822	$5.6956920069239914 * 10^{-8}$	1.0000000149011612
2^{-27}	0.11694231629371643	$3.460517827846843 * 10^{-8}$	1.0000000074505806
2^{-28}	0.11694228649139404	$4.802855890773117 * 10^{-9}$	1.0000000037252903
2^{-29}	0.11694222688674927	$5.480178888461751 * 10^{-8}$	1.0000000018626451
2^{-30}	0.11694216728210449	$1.1440643366000813 * 10^{-7}$	1.0000000009313226
...	...	...	...
2^{-35}	0.11693954467773438	1.9010469435800585	1.0000000000291038
2^{-36}	0.116943359375	1.9010469435800585	1.000000000014552
2^{-37}	0.1169281005859375	1.9010469435800585	1.000000000007276
...	...	...	...
2^{-49}	0.125	0.008057718311461848	1.0000000000000018
2^{-50}	0.0	0.11694228168853815	1.0000000000000009
2^{-51}	0.0	0.11694228168853815	1.0000000000000004
2^{-52}	-0.5	0.6169422816885382	1.0000000000000002
2^{-53}	0.0	0.11694228168853815	1.0
2^{-54}	0.0	0.11694228168853815	1.0

Tabela 11: Wynik aproksymacji pochodnej funkcji f w punkcie $x_0 = 1$ wraz z błędem i wielkością przesunięcia $1 + h$. Wartość pochodnej $f'(x_0) \approx 0.11694228168853815$.

7.4 Interpretacja wyników i wnioski

W przypadku aproksymowania pochodnej bez błędów wynikających z zastosowania arytmetyki zmiennopozycyjnej łatwo zauważyć, że im mniejszy jest krok h zastosowany we wzorze, tym dokładniejsze przybliżenie $f'(x_0)$ uzyskamy.
Przechodząc do arytmetyki zmiennopozycyjnej, w której daną liczbę musimy zapisać na ograniczonej liczbie bitów, należy pamiętać, że odejmowanie liczb x i y takich, że $x \approx y$ wiąże się z dużym błędem, znacząco przewyższającym rzędowo błąd reprezentacji liczby w danym systemie. W związku z tym, gdy korzystamy z aproksymacji pochodnej według powyższego wzoru w arytmetyce Float64, to istnieje takie h, że dla każdego $h_{new} < h$ błąd aproksymacji pochodnej z użyciem h_{new} jest większy niż z użyciem h. Dla podanego zakresu wartości $h = 2^{-n}$ $(n = 0, 1, 2, ..., 54)$ takim h dającym najlepsze przybliżenie pochodnej $f'(1)$ jest $h = 2^{-28}$.
Problem wielkości tego błędu występuje w szczególności, gdy wartość bezwzględna pochodnej w danym punkcie jest mała (funkcja rośnie bądź maleje powoli), tak jak w przykładzie funkcji podanej w tym zadaniu.