Sprawozdanie z Laboratorium
Obliczenia Naukowe - Lista 1

Karol Wziątek

27 października 2025

Zadanie 1

0.1 Epsilon maszynowy (*macheps*)

Epsilonem maszynowym *macheps* nazywamy najmniejszą liczbę dodatnią taką, że w arytmetyce zmiennoprzecinkowej zachodzi $1.0 + \text{macheps} > 1.0$.

Jest to miara precyzji obliczeń, która określa odległość od liczby 1.0 do następnej reprezentowalnej liczby maszynowej. Im mniejszy epsilon, tym więcej bitów przeznaczonych na mantysę w danym typie zmiennoprzecinkowym, co z kolei przekłada się na wyższą precyzję tej arytmetyki.

Typ danych	Wartość z `float.h` (GCC)	Wartość z `eps(T)` (Julia)	Wartość wyznaczona iteracyjnie
`Float16`	$9.7656e{-4}$	$9.77e{-4}$	$9.77e{-4}$
`Float32`	$1.192\,093e{-7}$	$1.192\,092\,9e{-7}$	$1.192\,092\,9e{-7}$
`Float64`	$2.220\,446e{-16}$	$2.220\,446\,049\,250\,313e{-16}$	$2.220\,446\,049\,250\,313e{-16}$

Wartości *macheps* wyznaczone iteracyjnie są zgodne z wynikami funkcji `eps(T)`. Wyznaczone zostały poprzez dzielenie przez 2 liczy 1.0 w danej arytmetyce aż T(1.0) + eta == T(1.0), gdzie T to kolejno Float16, Float32, Float64.

Związek między macheps a ϵ: $\text{macheps} = 2 \cdot \epsilon$ - na podstawie wartości podanych na wykładzie.

0.2 Najmniejsza dodatnia liczba maszynowa (*eta*)

Liczba *eta* (η) to najmniejsza dodatnia wartość, jaką można reprezentować w danym standardzie zmiennoprzecinkowym.

- **Związek z MIN_{sub}**: Liczba *eta* jest tożsama z MIN_{sub}, czyli najmniejszą możliwą do reprezentowania dodatnią liczbą subnormalną. W języku Julia wartość tę można uzyskać za pomocą funkcji `nextfloat(T(0.0))`.
- **Związek z MIN_{nor}**: Funkcja `floatmin(T)` zwraca najmniejszą dodatnią liczbę **znormalizowaną**, znaną jako MIN_{nor}. Dla odpowiednio `Float32` i `Float64` wartości te wynoszą: $1.175\,494\,4e{-38}$ i $2.225\,073\,858\,507\,201\,4e{-308}$, co zgadza się z wartościami podanymi na wykładzie.

Typ danych	Wartość z `nextfloat(T(0.0))`	Wartość wyznaczona iteracyjnie
`Float16`	$6.0e{-8}$	$6.0e{-8}$
`Float32`	$1.0e{-45}$	$1.0e{-45}$
`Float64`	$5.0e{-324}$	$5.0e{-324}$

Wartości *eta* wyznaczone iteracyjnie są zgodne z wynikami funkcji `nextfloat(T(0.0))`. Wyznaczone zostały poprzez dzielenie przez 2 liczy 1.0 w danej arytmetyce aż T(0.0) + eta == T(0.0), gdzie T to kolejno Float16, Float32, Float64

0.3 Największa wartość skończona (MAX)

Liczba MAX to najwyższa wartość, którą można zapisać w danym typie zmiennoprzecinkowym.

Typ danych	Wartość z `float.h` (GCC)	Wartość wyznaczona iteracyjnie	Wartość z `floatmax(T)` (Julia)
`Float16`	—	$6.55e4$	$6.55e4$
`Float32`	$3.402\,823\,466\,385\,288\,6e38$	$3.402\,823\,5e38$	$3.402\,823\,5e38$
`Float64`	$1.797\,693\,134\,862\,315\,7e308$	$1.797\,693\,134\,862\,315\,7e308$	$1.797\,693\,134\,862\,315\,7e308$

Jak widać w tabeli, wartości wyznaczone iteracyjnie są zgodne z pozostałymi źródłami. Aby doświadczalnie wyznaczyć MAX trzeba mnożyć T(1.0) przez dwa aż do uzyskania INF. Bierzemy ostatnią wartość przed uzyskaniem INF i zamieniamy wszystkie jej 0 na 1, gdzie T to kolejno Float16, Float32, Float64.

Zadanie 2

W tym zadaniu należy sprawdzić, czy metoda Kahana poprawnie wyznacza epsilon maszynowy dla typów Float16, Float32, Float64.

Typ danych	Wartość z metody Kahana	Wartość z `eps(T)` (Julia)
`Float16`	$-9.77e-4$	$9.77e-4$
`Float32`	$1.192\,092\,9e-7$	$1.192\,092\,9e-7$
`Float64`	$-2.220\,446\,049\,250\,313e-16$	$2.220\,446\,049\,250\,313e-16$

Wnioski Z powyższej tabeli widzimy, że wyrażenie Kahana niepoprawnie wyznaczało epsilon maszynowy dla wszystkich typów zmiennopozycyjnych. W celu otrzymania prawidłowego rozwiązania należy na wynik nałożyć wartość bezwzględną. Błędy w bicie znaku wynikają z reprezentacji rozwinięcia binarnego ułamka $\frac{4}{3}$.

Zadanie 3

Sprawdź eksperymentalnie w języku Julia, że w arytmetyce Float64 (arytmetyce double w standarcie IEEE 754) liczby zmiennopozycyjne są równomiernie rozmieszczone w [1, 2] z krokiem = 252. Innymi słowy, każda liczba zmiennopozycyjna x pomiędzy 1 i 2 może być przedstawione następująco x = 1 + k w tej arytmetyce, gdzie k = 1, 2, . . . , 252 1 i = 252.

- **Przedział** $[1, 2]$**:** W arytmetyce `double`, liczby zmiennoprzecinkowe są rozmieszczone równomiernie na przedziale $[1, 2]$ z krokiem równym $\delta = 2^{-52}$. Oznacza to, że każda kolejna liczba na tym przedziale różni się od poprzedniej o dokładnie δ. Sprawdzono to eksperymentalnie: 1000-krotnie generując losową liczbę z tego przedziału i wyznaczając kolejną liczbę maszynową za pomocą funkcji `nextfloat()`, różnica między nimi zawsze była równa δ.
- **Przedział** $[0.5, 1]$**:** Dla tego przedziału krok wynosi $\delta = 2^{-53}$. Każda liczba może być przedstawiona jako: $x = 1 + k \cdot \delta$, gdzie k jest liczbą całkowitą, a $\delta = 2^{-53}$.
- **Przedział** $[2, 4]$**:** W tym przypadku krok jest większy i wynosi $\delta = 2^{-51}$. Dla tego przedziału każda liczba może być przedstawiona jako: $x = 2 + k \cdot \delta$, gdzie $\delta = 2^{-51}$.

Powyższe eksperymenty potwierdzają, że w arytmetyce zmiennoprzecinkowej liczby są rozmieszczone gęściej bliżej zera i rzadziej w miarę oddalania się od niego. Zjawisko to nie jest przypadkowe, lecz wynika bezpośrednio ze sposobu, w jaki liczby są reprezentowane w formacie IEEE 754.

Zadanie 4

W tym zadaniu trzeba znaleść eksperymentalnie najmniejszą liczbę x w arytmetyce $Float64$, $1 < x < 2$, taką że:

$$x \otimes (1/x)) \neq 1$$

Aby ją wyznaczyć, zaczynamy od liczby 1 i zwiększamy ją przy użyciu funkcji $nextfloat$ aż do momentu, gdy warunek przestanie być spełniony. W ten sposób otrzymujemy:

$$x = 1.000000057228997$$

Jest to najmniejsza liczba w arytmetyce $Float64$, spełniająca podany warunek.

Zadanie 5

W tym zadaniu należy zaimplementować 4 różne algorytmy obliczania iloczynu skalarnego dwóch wektorów w arytmetyce $Float32$ oraz $Float64$ i porównać ich wyniki między sobą oraz z wynikiem dokładnym. Implementacja algorytmów:

1. **w przód** – $\sum_{i=1}^{n} x_i y_i$
2. **w tył** – $\sum_{i=n}^{1} x_i y_i$
3. **sortowanie rosnąco** – sortujemy iloczyny $x_i y_i$ rosnąco (w zależności od wartości bezwzględnej, osobno dodajemy ujemne i dodatnie)
4. **sortowanie malejąco** – analogicznie jak wyżej, ale sortujemy malejąco

Po przeprowadzeniu eksperymentów na wektorach:

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

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

oraz obliczeniu dokładnego wyniku iloczynu skalarnego (wynoszącego $-1.00657107000000 \cdot 10^{-11}$) otrzymujemy następujące wyniki:

Algorytm	Wynik Float32	Wynik Float64
1.	$-4.9994430 \cdot 10^{-1}$	$1.025188136829667 \cdot 10^{-10}$
2.	$-4.5434570 \cdot 10^{-1}$	$-1.564330887049437 \cdot 10^{-10}$
3.	$-5.0000000 \cdot 10^{-1}$	$0.000000000000000 \cdot 10^{0}$
4.	$-5.0000000 \cdot 10^{-1}$	$0.000000000000000 \cdot 10^{0}$

Jak widać, wyniki są obarczone dużym błędem, a różne algorytmy dają różne wyniki, co potwierdza, że kolejność wykonywania działań ma znaczenie. Błąd jest tym większy im mniejsza jest mantysa - Float64 lepiej na tym wychodzi.

Zadanie 6

W tym zadaniu należało obliczyć wartości funkcji: $f(x) = \sqrt{x^2+1} - 1$ oraz $g(x) = \frac{x^2}{\sqrt{x^2+1}+1}$ Iterując po wartościach $x = 8^{-1}, 8^{-2}, 8^{-3}, \ldots$ aż do momentu, gdy wynik obliczeń przestanie się zmieniać (w arytmetyce $Float64$) otrzymujemy następujące wyniki:

x	f(x)	g(x)
8^{-1}	$7.7822185373186414 \cdot 10^{-3}$	$7.7822185373187065 \cdot 10^{-3}$
8^{-2}	$1.2206286282867573 \cdot 10^{-4}$	$1.2206286282875901 \cdot 10^{-4}$
8^{-3}	$1.9073468138230965 \cdot 10^{-6}$	$1.9073468138265659 \cdot 10^{-6}$
8^{-4}	$2.9802321943606103 \cdot 10^{-8}$	$2.9802321943606116 \cdot 10^{-8}$
8^{-5}	$4.6566128730773926 \cdot 10^{-10}$	$4.6566128719931904 \cdot 10^{-10}$
8^{-6}	$7.2759576141834259 \cdot 10^{-12}$	$7.2759576141569561 \cdot 10^{-12}$
8^{-7}	$1.1368683772161603 \cdot 10^{-13}$	$1.1368683772160957 \cdot 10^{-13}$
8^{-8}	$1.7763568394002505 \cdot 10^{-15}$	$1.7763568394002489 \cdot 10^{-15}$
8^{-9}	$0.0000000000000000 \cdot 10^{0}$	$2.7755575615628914 \cdot 10^{-17}$
...	...	...
8^{-170}	$0.0000000000000000 \cdot 10^{0}$	$4.4501477170144028 \cdot 10^{-308}$
8^{-171}	$0.0000000000000000 \cdot 10^{0}$	$6.9533558078350043 \cdot 10^{-310}$
8^{-172}	$0.0000000000000000 \cdot 10^{0}$	$1.0864618449742194 \cdot 10^{-311}$
8^{-173}	$0.0000000000000000 \cdot 10^{0}$	$1.6975966327722179 \cdot 10^{-313}$
8^{-174}	$0.0000000000000000 \cdot 10^{0}$	$2.6524947387065904 \cdot 10^{-315}$
8^{-175}	$0.0000000000000000 \cdot 10^{0}$	$4.1445230292290475 \cdot 10^{-317}$
8^{-176}	$0.0000000000000000 \cdot 10^{0}$	$6.4758172331703867 \cdot 10^{-319}$
8^{-177}	$0.0000000000000000 \cdot 10^{0}$	$1.0118464426828729 \cdot 10^{-320}$
8^{-178}	$0.0000000000000000 \cdot 10^{0}$	$1.5810100666919889 \cdot 10^{-322}$
8^{-179}	$0.0000000000000000 \cdot 10^{0}$	$0.0000000000000000 \cdot 10^{0}$

Mimo, że funkcje f oraz g są algebraicznie takie same, to Julia zwraca inne rezultaty. Dzieje się tak, ponieważ odejmowanie od siebie dwóch bardzo bliskich liczb generuje duży błąd.

Im mniejszy x, tym wartość $\sqrt{x^2+1}$ jest bliższa 1. Tak więc operacja $\sqrt{x^2+1}-1$ jest operacją odejmowania dwóch bardzo bliskich sobie liczb, co powoduje powstanie dużego błędu numerycznego. W przypadku funkcji g nie występuje odejmowanie bliskich sobie liczb, przez co błąd numeryczny jest znacznie mniejszy.

Zadanie 7

W zadaniu należało obliczyć przybliżoną wartość pochodnej funkcji $f(x) = \sin x + \cos 3x$ w punkcie $x_0 = 1$ za pomocą wzoru:

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

Obliczając wartości funkcji dla kolejnych wartości $h = 2^{-n}, n \in \{0, 1, 2, \ldots, 54\}$ otrzymujemy następujące wyniki:
Początkowo, gdy zmniejszamy wartość h, błąd w przybliżeniu pochodnej również maleje. Jednak po pewnym punkcie (około $h = 2^{-27}$) zaczyna on ponownie rosnąć, mimo że h staje się mniejsze. Dzieje się tak, ponieważ w tym zakresie zaczyna przeważać błąd zaokrągleń. Przy bardzo małych h różnica $f(x0+h) - f(x0)$ jest tak mała, że ograniczona precyzja arytmetyki zmiennoprzecinkowej powoduje znaczne zniekształcenia, co skutkuje większym błędem w obliczeniach.

h	$\tilde{f}'(x_0)$	Błąd bezwzględny
2^{0}	$8.694677 \cdot 10^{-1}$	$7.525254 \cdot 10^{-1}$
2^{-1}	$4.730729 \cdot 10^{-1}$	$3.561306 \cdot 10^{-1}$
2^{-2}	$2.998405 \cdot 10^{-1}$	$1.828982 \cdot 10^{-1}$
2^{-3}	$2.507786 \cdot 10^{-1}$	$1.338363 \cdot 10^{-1}$
2^{-4}	$2.381337 \cdot 10^{-1}$	$1.211914 \cdot 10^{-1}$
2^{-5}	$2.349485 \cdot 10^{-1}$	$1.180062 \cdot 10^{-1}$
2^{-6}	$2.341506 \cdot 10^{-1}$	$1.172084 \cdot 10^{-1}$
2^{-7}	$2.339511 \cdot 10^{-1}$	$1.170088 \cdot 10^{-1}$
2^{-8}	$2.339012 \cdot 10^{-1}$	$1.169589 \cdot 10^{-1}$
2^{-9}	$2.338887 \cdot 10^{-1}$	$1.169464 \cdot 10^{-1}$
2^{-10}	$2.338856 \cdot 10^{-1}$	$1.169433 \cdot 10^{-1}$
2^{-11}	$2.338848 \cdot 10^{-1}$	$1.169425 \cdot 10^{-1}$
2^{-12}	$2.338846 \cdot 10^{-1}$	$1.169423 \cdot 10^{-1}$
2^{-13}	$2.338846 \cdot 10^{-1}$	$1.169423 \cdot 10^{-1}$
...	...	...
2^{-50}	$1.250000 \cdot 10^{-1}$	$8.057718 \cdot 10^{-3}$
2^{-51}	$2.500000 \cdot 10^{-1}$	$1.330577 \cdot 10^{-1}$
2^{-52}	$-5.000000 \cdot 10^{-1}$	$6.169423 \cdot 10^{-1}$
2^{-53}	$1.000000 \cdot 10^{0}$	$8.830577 \cdot 10^{-1}$
2^{-54}	$0.000000 \cdot 10^{0}$	$1.169423 \cdot 10^{-1}$