Obliczenia Naukowe - laboratorium 1

Łukasz Machnik

23 października 2023

1 Zadanie 1

1.1 Epsilon maszynowy *macheps*

Epsilon maszynowy to najmniejsza liczba większa od 0 która może być reprezentowana w danej arytmetyce i dla której zachodzi $fl(1.0 + macheps) > 1.0$ i $fl(1.0 + macheps) = 1 + macheps$. A zatem *macheps* jest odległością liczby 1.0 od kolejnej liczby większej od 1 reprezentowanej w danej arytmetyce.

Epsilon maszynowy musi być potęgą dwójki przez sposób zapisu liczb w systemie zmiennopozycyjnym - jest to najmniejsza możliwa liczba którą można zapisać w części ułamkowej mantysy. Aby znaleźć wartość liczby *macheps* zaimplementowałem następujący algorytm:

1. Deklaracja zmiennej x danego typu
2. Przypisanie $x = 1.0$
3. Dopóki $x + 1.0 > 1.0$
   (a) Zapamiętanie obecnej wartości x jako x_p
   (b) Przypisanie $x = \frac{x}{2}$
4. Zwrócenie wartości zmiennej x_p

Następująca tabela prezentuje porównanie wyników zwróconych przez powyższy algorytm, wyników zwróconych przez wbudowaną w język Julia funkcję *eps(Typ)* oraz danych z pliku float.h mojej instalacji języka C.

Typ	Float16	Float32 (float)	Float64 (double)
macheps	0.000977	$1.1920929 * 10^{-7}$	$2.220446049250313 * 10^{-16}$
eps(Typ)	0.000977	$1.1920929 * 10^{-7}$	$2.220446049250313 * 10^{-16}$
float.h	N/A	$1.1920928955078125 * 10^{-7}$	$2.22044604925031308084726333618164062 5 * 10^{-16}$

Można stwierdzić że eksperyment został przeprowadzony poprawnie gdyż otrzymane wartości są takie same jak wartości zwracane przez funkcję *eps(Typ)*. Widać też że w języku C epsilon maszynowy ma większą precyzję niż w Julii.

Precyzja arytmetyki to najmniejsza taka liczba ϵ że $\frac{|fl(x)-x|}{|x|} \leqslant |\epsilon|$. A skoro odległość między 1.0 a następną liczbą maszynową wynosi *macheps* to $\epsilon = \frac{macheps}{2}$

1.2 Liczba maszynowa *eta*

Liczba maszynowa *eta* to najmniejsza liczba większa od 0.0, która ma swoją reprezentację w danej arytmetyce. Można zatem doszukać się sporej analogii do poprzedniego podpunktu, a więc i algorytm wyszukiwania tej liczby jest bardzo podobny:

1. Deklaracja zmiennej x danego typu
2. Przypisanie $x = 1.0$
3. Dopóki $x > 0.0$
   (a) Zapamiętanie obecnej wartości x jako x_p
   (b) Przypisanie $x = \frac{x}{2}$

4. Zwrócenie wartości zmiennej x_p

Wyniki zwrócone przez ten algorytm w porównaniu z wynikami wywołania funkcji *nextfloat(0.0)* (zwracającej następną liczbę maszynową w danej arytmetyce) przedstawia poniższa tabela:

Typ	Float16	Float32	Float64
eta	$6.0 * 10^{-8}$	$1.0 * 10^{-45}$	$5.0 * 10^{-324}$
nextfloat(0.0)	$6.0 * 10^{-8}$	$1.0 * 10^{-45}$	$5.0 * 10^{-324}$

Eksperyment ponownie można uznać za udany - liczba wyznaczona eksperymentalnie pokrywa się z liczbą zwróconą przez funkcję wbudowaną.

Liczbę MIN_{sub} definiujemy jako $MIN_{sub} = 2^{-(t-1)}2^{c_{min}}$ gdzie t oznacza długość mantysy, a $c_{min} = -2^{d-1} + 2$ gdzie d ilość bitów przeznaczonych na zapis cechy. Po znormalizowaniu do danego typu liczba MIN_{sub} jest równa liczbie *eta*.

1.3 Floatmin

Zadanie polega na sprawdzeniu co wyświetla funkcja *floatmin(T)* dla Float32 i Float64 i porównanie tych wartości z liczbą $MIN_{nor} = 2^{c_{min}}$ dla $c_{min} = -2^{d-1} + 2$

Typ	Float32	Float64
floatmin()	$1.1754944 * 10^{-38}$	$2.2250738585072014 * 10^{-308}$
$fl(MIN_{nor})$	$1.1754944 * 10^{-38}$	$2.2250738585072014 * 10^{-308}$

Można zauważyć że funkcja floatmin() zwraca wartość znormalizowaną wartość MIN_{nor} dla danego typu danych

1.4 Floatmax

Ostatnim podpunktem jest iteracyjne wyznaczenie liczby *fmax* będącej największą liczbą jaką można zapisać w danej arytmetyce i porównanie tej liczby z wartością zwracaną przez wbudowaną funkcję *floatmax()* oraz podanymi na wykładzie wartościami $MAX = (2 - 2^{-(t-1)})2^{c_{max}}$ dla każdego typu danych.

Algorytm wyznaczający *fmax* wykorzystuje wyliczoną wcześniej wartość *eps* i działa w następujący sposób:

1. Deklaracja zmiennej x danego typu
2. Przypisanie $x = 2.0 - eps()$
3. Dopóki $x < \infty$
   (a) Zapamiętanie obecnej wartości x jako x_p
   (b) Przypisanie $x = x * 2$
4. Zwrócenie wartości zmiennej x_p

Algorytm ten na początku ustawia mantysę na największą możliwą liczbę a następnie iteracyjnie zwiększa cechę dopóki liczba jest mniejsza niż nieskończoność. Wyniki uzyskane w ten sposób wraz z porównaniem wymaganym w zadaniu prezentuje poniższa tabela:

Typ	Float16	Float32	Float64
fmax	$6.55 * 10^4$	$3.4028235 * 10^{38}$	$1.7976931348623157 * 10^{308}$
floatmax()	$6.55 * 10^4$	$3.4028235 * 10^{38}$	$1.7976931348623157 * 10^{308}$
$fl(MAX)$	N/A	$3.4028235 * 10^{38}$	$1.7976931348623157 * 10^{308}$

Jak widać eksperyment można uznać za udany gdyż eksperymentalnie wyznaczone wartości są takie same jak wartości zwrócone przez funkcję *floatmax()* oraz jak znormalizowane wartości MAX obliczone wg. wzoru z wykładu.

2 Zadanie 2

Niech $f() = 3(\frac{4}{3}-1)-1$. Wartość funkcji $f()$ w naturalnej arytmetyce wynosi 0 jednak w arytmetyce zmiennopozycyjnej sprawa wygląda inaczej ponieważ wynik odejmowania w nawiasie to $\frac{1}{3}$ a jest to liczba której nie da się dokładnie reprezentować w systemie binarynm. Efekty zaokrąglenia tej liczby kumuluje się przy mnożeniu przez 3 oraz przy odejmowaniu dwóch bardzo podobnych siebie liczbco skutkuje otrzymaniem następujących wyników:

Typ	Float16	Float32	Float64
$f()$	-0.000977	$1.1920929 * 10^{-7}$	$-2.220446049250313 * 10^{-16}$
$eps()$	0.000977	$1.1920929 * 10^{-7}$	$2.220446049250313 * 10^{-16}$

Jak widać wartość funkcji $f()$ jest równa $\pm eps()$, a więc potwierdziła się teza że za pomocą funkcji $f()$ można obliczyć wartość $eps()$

3 Zadanie 3

W arytmetyce double w standardzie IEEE 754 (w języku Julia typ Float64) liczby zmiennopozycyjne w przedziale $[1; 2]$ są rozmieszczone równomiernie z krokiem $\delta = 2^{-52}$. Zatem każda liczba może być przedstawiona jako $x = 1 + k\delta$ dla $k \in \{0, 1, 2, \ldots, 2^{52}\}$ i $\delta = 2^{-52}$. Moim zadaniem było potwierdzić ten fakt eksperymentalnie oraz sprawdzić rozmieszczenie liczb w przedziałach $[\frac{1}{2}; 1]$ oraz $[2; 4]$

W celu wykonania zadania wykorzystałem funkcję *bitstring()* wyświetlającą daną liczbę jako ciąg bitów. Zaczynając od $k = 0$ wyświetlałem kolejno reprezentacje binarne liczb $x = 1 + k\delta$.

k	*bitstring(x)*
0	0 01111111111 0000000000000000000000000000000000000000000000000000
1	0 01111111111 0000000000000000000000000000000000000000000000000001
2	0 01111111111 0000000000000000000000000000000000000000000000000010
3	0 01111111111 0000000000000000000000000000000000000000000000000011
4	0 01111111111 0000000000000000000000000000000000000000000000000100
...	...
$2^{52} - 4$	0 01111111111 1111111111111111111111111111111111111111111111111100
$2^{52} - 3$	0 01111111111 1111111111111111111111111111111111111111111111111101
$2^{52} - 2$	0 01111111111 1111111111111111111111111111111111111111111111111110
$2^{52} - 1$	0 01111111111 1111111111111111111111111111111111111111111111111111
2^{52}	0 10000000000 0000000000000000000000000000000000000000000000000000

Zapis binarny liczb Float64 można podzielić na 3 części:

- 1 bit przeznaczony na zapis znaku
- 11 bitów reprezentujących cechę
- 52 bity zapisujące mantysę

Jak widać zwiększając k o 1 zwiększamy mantysę o 1 a cecha pozostaje bez zmian. Cecha zmienia się dopiero dla $x = 2.0$. Można stąd wysnuć wniosek że liczby na przedziale $[1; 2]$ są rozmieszczone równomiernie z krokiem $\delta = 2^{-52}$

Analogicznie sprawdziłem jak wygląda reprezentacja bitowa liczb w przedziale $[\frac{1}{2}; 1]$ eksperymentalnie sprawdzając że dla równomiernego rozmieszczenia krok powinien wynosić $\delta = 2^{-53}$ a $k \in \{0, 1, 2, \ldots, 2^{51}\}$:

k	*bitstring(x)*
0	0 01111111110 0000000000000000000000000000000000000000000000000000
1	0 01111111110 0000000000000000000000000000000000000000000000000001
2	0 01111111110 0000000000000000000000000000000000000000000000000010
3	0 01111111110 0000000000000000000000000000000000000000000000000011
4	0 01111111110 0000000000000000000000000000000000000000000000000100
...	...
$2^{51} - 4$	0 01111111110 1111111111111111111111111111111111111111111111111100
$2^{51} - 3$	0 01111111110 1111111111111111111111111111111111111111111111111101
$2^{51} - 2$	0 01111111110 1111111111111111111111111111111111111111111111111110
$2^{51} - 1$	0 01111111110 1111111111111111111111111111111111111111111111111111

Oraz w przedziale $[2;4]$ z krokiem $\delta = 2^{-51}$ oraz $k \in \{0, 1, 2, \ldots, 2^{52}\}$:

k	*bitstring(x)*
0	0 10000000000 0000000000000000000000000000000000000000000000000000
1	0 10000000000 0000000000000000000000000000000000000000000000000001
2	0 10000000000 0000000000000000000000000000000000000000000000000010
3	0 10000000000 0000000000000000000000000000000000000000000000000011
4	0 10000000000 0000000000000000000000000000000000000000000000000100
...	...
$2^{52} - 4$	0 10000000000 1111111111111111111111111111111111111111111111111100
$2^{52} - 3$	0 10000000000 1111111111111111111111111111111111111111111111111101
$2^{52} - 2$	0 10000000000 1111111111111111111111111111111111111111111111111110
$2^{52} - 1$	0 10000000000 1111111111111111111111111111111111111111111111111111

Wniosek jest następujący: dla każdego przedziału $[2^n; 2^{n+1}]$ dla $n \in \mathbb{Z}$ liczby w arytmetyce double w standardzie IEEE 754 są rozmieszczone równomiernie z krokiem $\delta = 2^{-52+n}$. Dla liczb ujemnych działa to analogicznie - jedyna różnica to bit znaku ustawiony na 1

4 Zadanie 4

Polecenie każe znaleźć najmniejszą liczbę z przedziału $(1; 2)$ w arytmetyce Float64 taką że $x * \frac{1}{x} \neq 1$.

Wykorzystując wiedzę z poprzedniego zadania stworzyłem bruteforce'owy program przeglądający wszystkie reprezentowalne w tym przedziale liczby od najmniejszej $(fl(1.0 + 2^{-52}))$ aż do natrafienia na liczbę spełniającą powyższe równanie.

Najmniejszą taką liczbą jest 1.000000057228997.

Należy pamiętać że działając w arytmetyce Float64 nie zawsze otrzymujemy dokładny wynik - często jest to tylko przybliżenie z jakąś dokładnością. W przypadku liczby 1.000000057228997 błąd przybliżenia jest na tyle niefortunny, że wykonując zadane działanie nie otrzymujemy oczekiwanego wyniku.

5 Zadanie 5

Mamy zadane dwa wektory:

$$x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$$
$$y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$$

I chcemy policzyć ich iloczyn skalarny:

$$S = \sum_{i=1}^{5} x[i] * y[i]$$

W tym celu używam czterech różnych algorytmów:

- S_a: oblicza sumę idąc w wektorach od początku do końca
- S_b: oblicza sumę idąc w wektorach od końca do początku
- S_c: zapamiętuje wszystkie iloczyny częściowe a potem dodaje je: osobno ujemne od najmniejszego do największego, osobno nieujemne od największego do najmniejszego, a na końcu dodaje te dwie sumy częściowe
- S_d: zapamiętuje wszystkie iloczyny częściowe a potem dodaje je: osobno ujemne od największego do najmniejszego, osobno nieujemne od najmniejszego do największego, a na końcu dodaje te dwie sumy częściowe

Prawidłowa wartość $S = -1.00657107 * 10^{-11}$ Wyniki każdego z tych algorytmów oraz błąd bezwzględny względem wartości dodatniej (Δ) przedstawia tabela poniżej:

Typ	Float32	Float64
S_a	-0.4999443	$1.0251881368296672 * 10^{-10}$
S_b	-0.4543457	$-1.5643308870494366 * 10^{-10}$
S_c	-0.5	0.0
S_d	-0.5	0.0
Δ_a	0.49994429944939167	$1.1258452438296672 * 10^{-10}$
Δ_b	0.4543457031149343	$1.4636737800494365 * 10^{-10}$
Δ_c	0.4999999999899343	$1.00657107 * 10^{-11}$
Δ_d	0.4999999999899343	$1.00657107 * 10^{-11}$

Niezależnie od algorytmu występują większe lub mniejsze błędy. Dla Float32 najbliżej poprawnej wartości był algorytm S_b a dla Float64 mimo pozornie najmniejszej precyzji najbliżej były algorytmy S_c i S_d z wynikiem 0.0

6 Zadanie 6

Zadane są dwie funkcje:

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

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

Teoretycznie te funkcje są sobie równe jednak w arytmetyce zmiennopozycyjnej zwracane przez nie wartości mogą się różnić. Poniżej tabela wartości zwracanych przez funkcje dla $x = 8^{-n}$

n	1	2	3	...
$f(x)$	0.0077822185373186414	0.00012206286282867573	$1.9073468138230965e-6$	...
$g(x)$	0.0077822185373187065	0.00012206286282875901	$1.907346813826566e-6$	...

Dla małych n różnica jest niewielka ale nawet dla $x = \frac{1}{8}$ różnica w uzyskanych wynikach występuje

n	...	9	...	178	$\geqslant 179$
$f(x)$	...	0.0	...	0.0	0.0
$g(x)$	...	$2.7755575615628914e-17$	...	$1.6e-322$	0.0

Dla $n \geqslant 9$ wartość funkcji $f(x)$ wynosi 0.0 podczas gdy wartość funkcji $g(x)$ jest rózna od 0 aż do $n = 178$. Funkcji $g(x)$ można ufać bardziej nie dlatego że dłużej daje wyniki różne od 0 (wszak nie muszą być to wyniki bliższe poprawnym) ale dlatego żę z punktu widzenia obliczeń arytmetyki zmiennoprzecinkowej wykonujemy w niej "bezpieczne" działania. x jest liczbą bardzo małą dlatego $\sqrt{x^2 + 1}$ jest bliskie 1. zatem w funkcji $f(x)$ odejmujemy dwie bliskie sobie liczby co podwyższa prawdopodobieństwo wystąpienia błędu

7 Zadanie 7

Mamy zadaną funkcje:

$$f(x) = sin(x) + cos(3x)$$

Wiadomo że pochodna tej funkcji jest następująca:

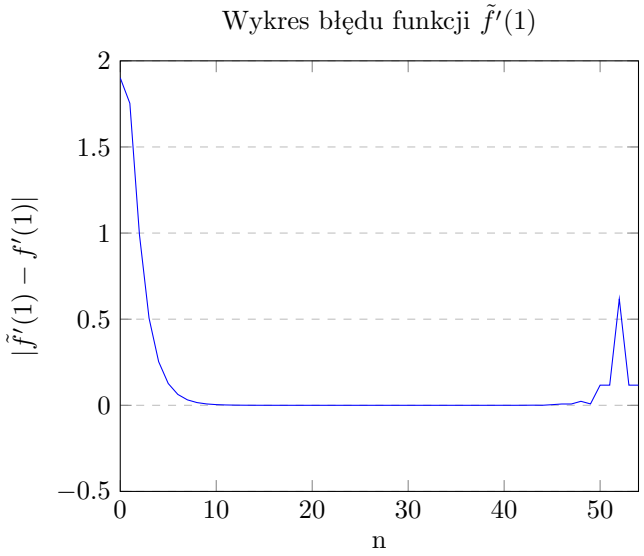
$$f'(x) = cos(x) - 3sin(3x)$$

Przybliżoną wartość pochodnej w punkcie x_0 można obliczyć za pomocą następującej funkcji:

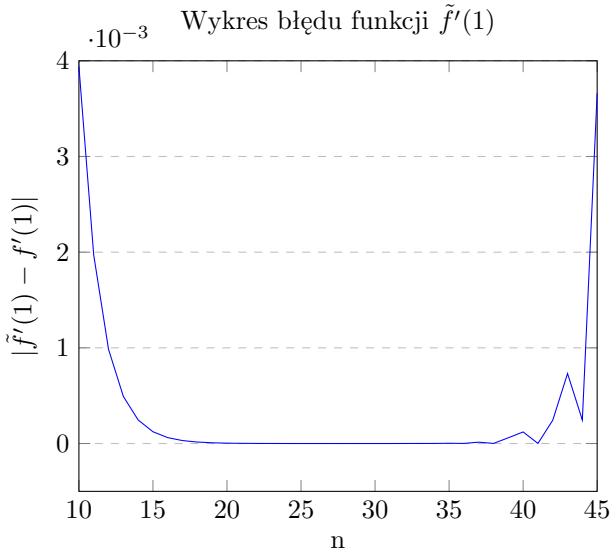
$$\tilde{f}'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h}$$

Napisałem program obliczający przybliżoną wartość pochodnej w punkcie $x_0 = 1$ dla $h = 2^{-n}$, $n \in \{0, 1, 2, \ldots, 54\}$ oraz odległość tego wyniku od wartość funkcji $f'(1) = 0.11694228168853815$

Poniższy wykres przedstawia błąd bezwzględny otrzymanego wyniku w zależności od parametru h dla $h = 2^{-n}$



Poniższy wykres przedstawia przybliżenie na wartości tej funkcji dla $10 \leqslant n \leqslant 45$



Błąd bardzo szybko maleje do wartości bliskich 0 już dla $n = 17$ jednak dla wartości $n > 40$ znowu rośnie. Jest to spowodowane tym że jeśli weźmiemy zbyt małe h to w liczniku będziemy odejmować bardzo bliskie siebie liczby co może spowodować że zaokrąglenie wyniku będzie mocno odbiegać od wyniku realnego.