Poniedziałek 7:30, TN

Obliczenia naukowe: sprawozdanie z laboratorium

LISTA NR 1

JAKUB IWON

1. Zadanie 1

1.1. Opis:

Celem zadania było wyznaczenie w sposób iteracyjny epsilonu maszynowego oraz znalezienie minimalnej i maksymalnej liczby dodatniej dla typów Float16, Float32, Float64.

1.2. Rozwiązanie i wnioski:

Epsilon maszynowy wyznaczamy dzieląc zmienną `macheps = 1` przez 2, sprawdzając jednocześnie czy zachodzi warunek `1.0 + macheps > 1`. W ten sposób dokonujemy przesunięcia bitowego w prawo. Wynikiem jest liczba, która w reprezentacji Float64 wygląda następująco:

`0011110010110000000000000000000000000000000000000000000000000000`

Wartość wykładnika jest równa -52 czyli tyle ile bitów ma mantysa. Wykładnik ten przesuwa nam "ukrytą" jedynkę z początku mantysy na miejsce najmniej znaczącego bitu. Jest to minimalna wartość jaką może mieć mantysa przez co dodanie wartości mniejszej do liczby 1.0 nie wpłynie na wynik.

Proces obliczania epsilonu maszynowego dla pozostałych typów jest analogiczny.

Wyniki:

`Obliczony epsilon maszynowy Float64:`	`2.220446049250313e-16`
`Obliczony epsilon maszynowy Float32:`	`1.1920929e-7`
`Obliczony epsilon maszynowy Float16:`	`0.000977`
`Eps(Float64):`	`2.220446049250313e-16`
`Eps(Float32):`	`1.1920929e-7`
`Eps(Float16):`	`0.000977`

Jak widać wyniki zgadzają się z wartościami zwracanymi przez funkcję `eps()`.

Wartości epsilonu maszynowego w pliku nagłówkowym `float.h` języka C:

`FLT_EPSILON = 1.192093e-07`
`DBL_EPSILON = 2.220446e-16`
`LDBL_EPSILON = 1.084202e-19`

Obliczona wartość `macheps` jest równa podwójnej wartości precyzji arytmetyki zadanej wzorem

$\varepsilon = \frac{1}{2}\ \beta^{1-t}$ (podstawiając dla typu double wartość t = 53, dla single t = 24, dla half t = 11).

Liczbę `eta` wyznacza się w podobny sposób jak epsilon maszynowy z tą różnicą, że nie przestajemy dzielić przez 2 dopóki `eta > 0.0`. W ten sposób uzyskujemy liczbę o najmniejszej możliwej wartości wykładnika i mantysy w standardzie IEEE754.

Wyniki:

`Eta64:`	`5.0e-324`
`Eta32:`	`1.0e-45`
`Eta16:`	`6.0e-8`
`nextfloat(Float64(0.0)):`	`5.0e-324`
`nextfloat(Float32(0.0)):`	`1.0e-45`
`nextfloat(Float16(0.0)):`	`6.0e-8`

Obliczone wartości `eta` oraz funkcji `nextfloat` dla argumentu 0.0 zgadzają się. Są one równe wartości

MIN_{Sub} zadanej wzorem $2^{-(t-1)} * 2^{cmin}$ (podstawiając odpowiednie wartości t dla poszczególnych typów).

Funkcja `floatmin` zwraca najmniejszą liczbę w postaci znormalizowanej dla poszczególnych typów i jest równa wartości MIN_{Nor}.

Maksymalną liczbę dla typów Float64, Float32 oraz Float16 możemy wyznaczyć iteracyjnie, mnożąc zmienną `max` przez 2 zaczynając od liczby zwracanej przez funkcję `prevfloat(1.0)`. W ten sposób uzyskujemy liczbę o maksymalnej wartości mantysy. Mnożąc przez 2 zwiększamy wartość wykładnika aż otrzymamy maksymalną wartość.

Wyniki:

Obliczona maksymalna wartość Float64:	1.7976931348623157e308
Obliczona maksymalna wartość Float32:	3.4028235e38
Obliczona maksymalna wartość Float16:	6.55e4
floatmax(Float64):	1.7976931348623157e308
floatmax(Float32):	3.4028235e38
floatmax(Float16):	6.55e4

Maksymalne wartości liczb zmiennopozycyjnych w pliku nagłówkowym `float.h` języka C:

```
FLT_MAX: 1E+37
DBL_MAX: 1E+37
LDBL_MAX: 1E+37
```

2. Zadanie 2

2.1. Opis

Celem zadania było obliczenie wartości wyrażenia $3 * \left(\frac{4}{3} - 1\right) - 1$ arytmetyce zmiennopozycyjnej i sprawdzenie czy jest ona równa wartości epsilonu maszynowego.

2.2. Rozwiązanie i wnioski

Wyniki:

Wartość wyrażenia dla Float64:	-2.220446049250313e-16
eps(Float64):	2.220446049250313e-16
Wartość wyrażenia dla Float32:	1.1920929e-7
eps(Float32):	1.1920929e-7
Wartość wyrażenia dla Float16:	-0.000977
eps(Float16):	0.000977

Jak widzimy obliczone wyrażenie jest równe co do wartości bezwzględnej epsilonowi maszynowemu dla poszczególnych typów zmiennych.

Dokładna wartość wyrażenia $3 * \left(\frac{4}{3} - 1\right) - 1$ wynosić 0. Uzyskany wynik różni się od dokładnego z uwagi na brak możliwości dokładnej reprezentacji liczby $\frac{4}{3}$ w komputerze. Pomimo, że następne obliczenia są dokładne początkowy błąd przekłada się na wynik końcowy. Jako, że popełniony zostaje tylko jeden błąd w obliczeniach, wartość absolutna błędu nie przekracza wartości epsilonu maszynowego.

3. Zadanie 3

3.1. Opis

Celem zadania było sprawdzenie w jakich odstępach rozmieszczone są liczby w arytmetyce zmiennopozycyjnej w poszczególnych przedziałach.

3.2. Rozwiązanie i wnioski

Aby sprawdzić rozmieszczenie liczb w przedziale [1, 2], zmienną początkowo równą 1 iteracyjnie zwiększamy o wartość 2^{-52} i dla każdego wyniku wywołujemy funkcję bitstring zwracającą binarną reprezentację liczby.

Fragment wyniku programu:

```
0011111111110000000000000000000000000000000000000000000000000000
0011111111110000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000011
0011111111110000000000000000000000000000000000000000000000000100
```

Można zauważyć, że za każdym razem zwiększamy mantysę o wartość, którą reprezentuje jej najmniej znaczący bit przy czym cecha pozostaje bez zmian. Teraz spójrzmy na binarną reprezentację liczby 2 oraz liczby poprzedzającej ją:

```
0011111111111111111111111111111111111111111111111111111111111111
0100000000000000000000000000000000000000000000000000000000000000
```

Widzimy tutaj, że aby osiągnąć wartość 2 (zmienić potęgę dwójki) musimy "wyzerować" mantysę. Aby to zrobić musimy powyżej opisaną czynność powtórzyć 2^{52} razy. Wnioskujemy stąd, że liczby w przedziale [1, 2] rozmieszczone są równomiernie z krokiem $2^{52} * 2^{0}$.

Podobne rozumowanie można przeprowadzić dla przedziałów $[\frac{1}{2}, 1]$ oraz [2, 4]. W obydwóch tych przedziałach mieści się 2^{52} liczb jednak rozmiar przerw między nimi zmienia się w zależności od wartości cechy. W przypadku przedziału $[\frac{1}{2}, 1]$ liczby rozmieszczone są z krokiem $2^{-52} * 2^{-1} = 2^{-53}$ natomiast w przedziale [2, 4] rozmiar przerwy to $2^{-52} * 2^{1} = 2^{-51}$.

4. Zadanie 4

4.1. Opis

Celem zadania było znalezienie w przedziale (1, 2) takiej liczby x, dla której wartość wyrażenia $x * (\frac{1}{x})$ była różna od 1. W drugiej części zadania należało znaleźć najmniejszą taką liczbę.

4.2. Rozwiązanie i wnioski

Szukaną liczbę znajdujemy w sposób iteracyjny podstawiając do wzoru każdą kolejną liczbę z przedziału (1, 2) i sprawdzając czy wynik równa się 1. Najmniejszą liczbę znajdujemy podobnie, z tą różnicą, że zaczynamy od pierwszej reprezentatywnej liczby po 0.

Wynik działania programu:

```
Znalezione x w przedziale (1, 2):   1.000000057228997
Najmniejsze znalezione x:           5.0e-324
```

Przyczyną niepoprawnie obliczonej wartości wyrażenia $x * (\frac{1}{x})$ jest brak możliwości dokładnej reprezentacji liczby $\frac{1}{x}$ w arytmetyce Float64.

Dokładna wartość (dokładność do 25 cyfr): `0.9999999427710062751579102`

Obliczona wartość: `0.9999999999999999`

W drugim przypadku wartość argumentu x była tak mała, że wynik $\frac{1}{x}$ został obliczony jako nieskończoność.

5. Zadanie 5

5.1. Opis

Celem zadania było policzenie iloczynu skalarnego na cztery sposoby: "w przód", "w tył", od największego do najmniejszego oraz od najmniejszego do największego a następnie porównanie poszczególnych wyników z prawidłową wartością.

5.2. Rozwiązanie i wnioski

Wyniki:

Algorytm	Wynik Float32	Wynik Float64	Błąd bezwzględny Float32	Błąd bezwzględny Float64	Błąd względny Float32	Błąd względny Float64
Algorytm A	-0.16756826908253317	1.0251881368296672e-10	0.1675682690724674593	1.1258452438296672e-10	1664743544362%	1118%
Algorytm B	-0.16756826941855252	-1.5643308870494366e-10	0.1675682694084868093	1.4636737800494366e-10	1664743547700%	1454%
Algorytm C	-0.16756826918572187	0.0	0.1675682691756561593	1.00657107e-11	1664743545387%	100%
Algorytm D	0.16756826918572187	0.0	0.1675682691756561593	1.00657107e-11	1664743545387%	100%

Patrząc na powyższe dane można zauważyć, że algorytmy C i D zwracały wyniki obarczone najmniejszym błędem natomiast algorytm B zwracał wartość najbardziej różniącą się od prawidłowego wyniku.

Wnioskować można stąd, że kolejność w jakiej dodajemy do siebie liczby ma wpływ na wynik. Stosując algorytmy C i D częściowo unikamy dodawania liczb o bardzo różnym rzędzie wielkości co mogłoby prowadzić do błędów.

Z powyższych danych widać również, że błędy bezwzględne są małe (mamy do czynienia z małymi liczbami) natomiast błędy względne są duże. Zmiana arytmetyki z typu `Double` na `Single` powoduje bardzo duży spadek precyzji obliczeń.

6. Zadanie 6

6.1. Opis

Celem zadania było wytłumaczenie różnicy wyników zwracanych przez komputer dla dwóch matematycznie równych funkcji $f(x) = \sqrt{x^2+1} - 1$ oraz $g(x) = \frac{x^2}{\sqrt{x^2+1}+1}$ dla poszczególnych argumentów.

6.2. Rozwiązanie i wnioski

Fragment uzyskanych wyników:

x	f(x)	g(x)
1.9721522630525295e-31	-1.0	1.9446922743316068e-62
2.465190328815662e-32	-1.0	3.0385816786431356e-64
3.0814879110195774e-33	-1.0	4.7477838728798994e-66
3.851859888774472e-34	-1.0	7.418412301374843e-68
4.81482486096809e-35	-1.0	1.1591269220898192e-69
6.018531076210112e-36	-1.0	1.8111358157653425e-71
7.52316384526264e-37	-1.0	2.8298997121333476e-73

Można zauważyć, że funkcja g(x) zwraca precyzyjne wyniki podczas gdy wyniki zwracane przez f(x) są obarczone dużym błędem.

Wyjaśnieniem tego zjawiska jest fakt, że w funkcji f(x) do czynienia mamy z utratą cyfr znaczących dokonując odejmowania dla argumentów bliskich zeru. Dzieje się tak ponieważ wartość $\sqrt{x^2+1}$ dla $x \approx 0$ zbliża się do 1. Dochodzi więc do odejmowania dwóch bardzo bliskich sobie liczb co może prowadzić do utraty cyfr znaczących.

Przekształcając wyrażenie $\sqrt{x^2+1}-1$ do postaci $\frac{x^2}{\sqrt{x^2+1}+1}$ unikamy tego problemu co przekłada się na bardziej precyzyjne wyniki.

7. Zadanie 7

7.1. Opis

Celem zadania było policzenie przybliżonej wartości pochodnej funkcji $f(x) = \sin x + \cos 3x$ za pomocą wzoru $f'(x) \approx \frac{f(x_0+h)-f(x_0)}{h}$ dla coraz mniejszych wartości h.

7.2. Rozwiązanie i wnioski

Fragment wyników zwracanych przez programu:

h	f'(1.0)	f'~(1.0)	Błąd bezwzględny
2^{-26}	0.11694228168853815	0.11694233864545822	5.6956920069239914e-8
2^{-27}	0.11694228168853815	0.11694231629371643	3.460517827846843e-8
2^{-28}	0.11694228168853815	0.11694228649139404	4.802855890773117e-9
2^{-29}	0.11694228168853815	0.11694222688674927	5.480178888461751e-8
2^{-30}	0.11694228168853815	0.11694216728210449	1.1440643366000813e-7
2^{-31}	0.11694228168853815	0.11694216728210449	1.1440643366000813e-7
2^{-32}	0.11694228168853815	0.11694192886352539	3.5282501276157063e-7

Jak widać od pewnego momentu mimo iż wartość h zmniejsza się, różnica między przybliżoną a dokładną wartością pochodnej zwiększa się.

Dzieje się tak ponieważ dla bardzo małych h, wartości f(x + h) oraz f(x) są bardzo bliskie siebie co może prowadzić do utraty cyfr znaczących. Gdy h osiągnie wartość mniejszą od ε dla typu Float64 (2^{-52}) liczby 1 oraz 1 + h są reprezentowane w komputerze przez tą samą wartość co prowadzi do przybliżenia pochodnej równego zero.