Sprawozdanie z listy pierwszej

Karolina Bąk

Październik 2019

1 Zadanie 1

Celem pierwszego zadania było iteracyjne wyznaczenie macheps, czyli najmniejszej liczby, która spełnia równanie:

$$fl(1.0 + macheps) = 1.0$$

dla typów Float16, Float32 oraz Float64. Aby obliczyć tą liczbę dzieliłam 1 w poszczególnych arytmetykach przez 2 aż równanie zostało spełnione. Otrzymane wyniki porównałam z machepsem otrzymanym z dostępnej w Julii funkcji eps() i z wartościami w pliku nagłówkowym float.h z języka C.

	pętla	eps()	float.h
Float16	0.000977	0.000977	-
Float32	$1.1920929 * 10^{-7}$	$1.1920929 * 10^{-7}$	$1.192092896 * 10^{-7}$
Float64	$2.220446049250313 * 10^{-16}$	$2.220446049250313 * 10^{-16}$	$2.2204460492503131 * 10^{-16}$

Wartość macheps jest ważna w arytmetyce zmiennoprzecinkowej, ponieważ określa górną granicę błędu względnego przy zaokrąglaniu liczby rzeczywistej do wartości akceptowanej przez komputer. Następnym celem zadania, było wyznaczenie, również iteracyjnie, liczby eta, czyli najmniejszej liczby powyżej zera, która spełnia równanie

$$eta > 0.0$$

dla typów z poprzedniej części. Algorytm działania był jedynie lekką modyfikacją pierwszego. Wyniki porównałam z funkcją nextfloat(*typ*(0.0)).

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

Liczba eta jest najmniejszą liczbą nieuznawaną przez arytmetykę za zero maszynowe. Podobnie jak SUB_{MIN}, które jest najmniejszą subnormalną liczbą w IEEE 754. Liczby te są sobie równe.

Przy skorzystaniu z funkcji floatmin() uzyskane MIN różni się od liczby eta. Dzieje się tak, ponieważ owa funkcja zwraca MIN_{NOR}, które również jest minimum ale dla znormalizowanych liczb IEEE 754, czyli takich, których cecha nie posiada najmniejszego możliwego wykładnika i wiodący bit w mantysie nie jest zerem.

	MIN_{NOR}
Float32	$1.1754944 * 10^{-38}$
Float64	$2.2250738585072014 * 10^{-308}$

Ostatnią częścią zadania było iteracyjne obliczenie MAX dla typów zmiennoprzecinkowych. Aby obliczyć tą liczbę zbliżałam się maksymalnie do maxa poprzez mnożenie przez dwa, a następnie zbliżałam się do maksimum liczba po liczbie. Wyniki porównałam z bazową funkcją Julii floatmax(*typ*) oraz z wartościami z pliku nagłówkowego float.h.

	pętla	floatmax()	float.h
Float16	$6.55 * 10^4$	$6.55 * 10^4$	-
Float32	$3.4028235 * 10^{38}$	$3.4028235 * 10^{38}$	$3.402823466 * 10^{38}$
Float64	$1.7976931348623157 * 10^{308}$	$1.7976931348623157 * 10^{308}$	$1.7976931348623158 * 10^{308}$

2 Zadanie 2

Celem tego zadania było sprawdzenie czy obliczając wyrażenie $3 * (3/4 - 1) - 1$ można uzyskać macheps dla danej arytmetyki zmiennoprzecinkowej. Używając REPL Julii uzyskałam następujące wyniki:

	$3 * (3/4 - 1) - 1$
Float16	-0.000977
Float32	$1.1920929 * 10^{-7}$
Float64	$-2.220446049250313 * 10^{-16}$

Zgadzają się one z uzyskanymi wcześniej wynikami. W przypadku Float16 i Float64 nastąpiła zmiana znaku na ujemny.

3 Zadanie 3

Celem tego zadania było sprawdzenie czy liczby w arytmetyce Float64 są rozłożone równomiernie w przedziale [1,2] ze skokiem $\delta = 2^{-52}$. Sprawdzenie każdej liczby po kolei potrzebowałoby sporo czasu, więc sprawdziłam jak zachowują się liczby w środku i na obrzeżach przedziału. Liczba 1.0 w zapisie binarnym (z podziałem na bit znaku, cechę i mantysę dla lepszej widoczności) wygląda tak:

0|01111111111|0000000000000000000000000000000000000000000000000000

Następna po niej liczba z tej arytmetyki wygląda tak:

0|01111111111|0000000000000000000000000000000000000000000000000001

Jej wartość to

$$(1 + 1/2^{52}) * 2^{1023-1023} = 1 + 1/2^{52}$$

2^{52} zostało przeze mnie użyte, ponieważ mantysa zawiera w Float64 52 bity. Zmiana wartości cechy następuje dopiero przy liczbie Float64(2.0), więc można wywnioskować, że do tego momentu każda liczba w tej arytmetyce występuje co 2^{-52}.

Następnie zbadałam rozłożenie liczb w przedziale [0.5,1.0]. 0.5 w zapisie binarnym wygląda tak:

0|01111111110|0000000000000000000000000000000000000000000000000000

Aby policzyć krok użyłam tej samej metody co wyżej

$$(1 + 1/2^{52}) * 2^{1022-1023} = (1 + 2^{52})/2^{53} = (1/2) + 1/2^{53}$$

A więc krok w tym przedziale wynosi 2^{-53}. Dla przedziału [2.0,4.0] obliczenia wyglądały następująco

$$(1 + 1/2^{52}) * 2^{1024-1023} = (1 + 2^{52})/2^{51} = 2 + 1/2^{51}$$

Krok dla tego przedziału wynosi 2^{-51}.

Można zauważyć pewną prawidłowość w rozmieszczeniu liczb zmiennoprzecinkowych Float64. W przedziałach typu $[2^n, 2^{n+1}]$, gdzie $n \in \mathbb{Z}$, odstęp między liczbami jest stały i wynosi

$$2^{-52+n}$$

4 Zadanie 4

Celem zadania było eksperymentalne wyznaczenie w arytmetyce Float64 takiej liczby x, że $1 < x < 2$ i $x*(1/x) \neq 1$. Aby wyznaczyć x iteracyjnie sprawdzałam po kolei czy liczby Float64 z tego przedziału spełniają to równanie i dla pierwszej napotkanej przerwałam pętlę. Uzyskałam wynik

1.000000057228997

Jest to najmniejsza liczba w tym przedziale co spełnia to równanie. Z kolei najmniejszą liczbą (w sensie najbliższą zera), która również spełnia ten warunek jest wyznaczona wcześniej $eta = 5.0 * 10^{-324}$.

5 Zadanie 5

Celem zadania jest implementacja iloczynu skalarnego na 4 sposoby i porównanie otrzymanych wyników w arytmetykach Float32 i Float64. 1. sposób to liczenie sumy w kolejności występowania, 2. sposób liczył sumę w odwrotnej kolejności, 3. sposób sumował sumy częściowe osobno dla dodatnich i ujemnych. Dodatnie od największej do najmniejszej, a ujemne od najmniejszej do największej. Następnie obie sumy częściowe były dodawane. 4. sposób był odwrotnością 3. Po wykonaniu tych algorytmów otrzymałam następujące wyniki.

	Float32	Float64
1	-0.4999443	$1.0251881368296672 * 10^{-10}$
2	-0.4543457	$-1.5643308870494366 * 10^{-10}$
3	-0.5	0
4	-0.5	0

Najbliżej wyniku $-1.00657107000000*10^{-11}$ była arytmetyka Float64 w sposobie 1. i 2. Reszta jest obarczona ogromnym błędem. Dzieje się tak, ponieważ dla iloczynu skalarnego z wektorami, które mają różne znaki przy współrzędnych zadanie jest źle uwarunkowane. Kolejność wykonania działań ma także duże znaczenie przy kumulacji błędu.

6 Zadanie 6

Celem zadania było obliczenie wartości dwóch równoważnych funkcji w Float64.

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

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

Obie funkcje należało przetestować dla $8^{-1}, 8^{-2}, 8^{-3}$ itd.

	f(x)	g(x)
1	0.0077822185373186414	0.0077822185373187065
2	0.00012206286282867573	0.00012206286282875901
3	$1.9073468138230965 * 10^{-6}$	$1.907346813826566 * 10^{-6}$
4	$2.9802321943606103 * 10^{-8}$	$2.9802321943606116 * 10^{-8}$
5	$4.656612873077393 * 10^{-10}$	$4.6566128719931904 * 10^{-10}$
6	$7.275957614183426 * 10^{-12}$	$7.275957614156956 * 10^{-12}$
7	$1.1368683772161603 * 10^{-13}$	$1.1368683772160957 * 10^{-13}$
8	$1.7763568394002505 * 10^{-15}$	$1.7763568394002489 * 10^{-15}$
9	0	$2.7755575615628914 * 10^{-17}$
20	0	$3.76158192263132 * 10^{-37}$
40	0	$2.8298997121333476 * 10^{-73}$
60	0	$2.1289799200040754 * 10^{-109}$

80	0	$1.6016664761464807 * 10^{-145}$
120	0	$9.065110999561118 * 10^{-218}$
160	0	$5.1306710016229703 * 10^{-290}$
178	0	$1.6 * 10^{-322}$
179	0	0

Funkcja f bardzo szybko zaczyna zwracać wartości równe zeru podczas, gdy funkcja g zbliża się do minimalnych dla Float64 wartości. Dzieje się tak, ponieważ przy odejmowaniu tak bliskich sobie liczb szybko tracone są znaczące bity przez wyrównywanie liczb. Jeśli przekształcimy funkcję tak, aby zastąpić odejmowanie dodawaniem precyzja działań znacząco wzrasta. Dlatego więc bardziej wiarygodne wyniki daje funkcja g(x).

7 Zadanie 7

Celem zadania było skorzystanie z wzoru na przybliżoną wartość pochodnej

$$f'(x_0) = (f(x_0 + h) - f(x_0))/h$$

w celu zbadania zachowania obliczania pochodnej w punkcie z coraz większą precyzją. Do testu użyłam zadanej funkcji $f(x) = sinx + cos3x$. Jej pochodna wynosi $f'(x) = cosx - 3sin3x$. Poniżej przedstawiam otrzymane przeze mnie wyniki dla $h = 2^{-i}$, gdzie $i = 0, 1, ...54$.

h	1+h	$fp'(x_0)$	$\lvert f'(x_0) - fp'(x_0)\rvert$
2^0	2.0	2.0179892252685967	1.9010469435800585
2^{-1}	1.5	1.8704413979316472	1.753499116243109
2^{-2}	1.25	1.1077870952342974	0.9908448135457593
2^{-3}	1.125	0.6232412792975817	0.5062989976090435
2^{-4}	1.0625	0.3704000662035192	0.253457784514981
2^{-5}	1.03125	0.24344307439754687	0.1265007927090087
2^{-6}	1.015625	0.18009756330732785	0.0631552816187897
2^{-7}	1.0078125	0.1484913953710958	0.03154911368255764
2^{-8}	1.00390625	0.1327091142805159	0.015766832591977753
2^{-9}	1.001953125	0.1248236929407085	0.007881411252170345
2^{-10}	1.0009765625	0.12088247681106168	0.0039401951225235265
2^{-11}	1.00048828125	0.11891225046883847	0.001969968780300313
2^{-12}	1.000244140625	0.11792723373901026	0.0009849520504721099
2^{-13}	1.0001220703125	0.11743474961076572	0.0004924679222275685
2^{-14}	1.00006103515625	0.11718851362093119	0.0002462319323930373
2^{-15}	1.000030517578125	0.11706539714577957	0.00012311545724141837
2^{-16}	1.0000152587890625	0.11700383928837255	$6.155759983439424 * 10^{-5}$
2^{-17}	1.0000076293945312	0.11697306045971345	$3.077877117529937 * 10^{-5}$
2^{-18}	1.0000038146972656	0.11695767106721178	$1.5389378673624776 * 10^{-5}$

2^{-19}	1.0000019073486328	0.11694997636368498	$7.694675146829866 * 10^{-6}$
2^{-20}	1.0000009536743164	0.11694612901192158	$3.8473233834324105 * 10^{-6}$
2^{-21}	1.0000004768371582	0.1169442052487284	$1.9235601902423127 * 10^{-6}$
2^{-22}	1.000000238418579	0.11694324295967817	$9.612711400208696 * 10^{-7}$
2^{-23}	1.0000001192092896	0.11694276239722967	$4.807086915192826 * 10^{-7}$
2^{-24}	1.0000000596046448	0.11694252118468285	$2.394961446938737 * 10^{-7}$
2^{-25}	1.0000000298023224	0.116942398250103	$1.1656156484463054 * 10^{-7}$
2^{-26}	1.0000000149011612	0.11694233864545822	$5.6956920069239914 * 10^{-8}$
2^{-27}	1.0000000074505806	0.11694231629371643	$3.460517827846843 * 10^{-8}$
2^{-28}	1.0000000037252903	0.11694228649139404	$4.802855890773117 * 10^{-9}$
2^{-29}	1.0000000018626451	0.11694222688674927	$5.480178888461751 * 10^{-8}$
2^{-30}	1.0000000009313226	0.11694216728210449	$1.1440643366000813 * 10^{-7}$
2^{-31}	1.0000000004656613	0.11694216728210449	$1.1440643366000813 * 10^{-7}$
2^{-32}	1.0000000002328306	0.11694192886352539	$3.5282501276157063 * 10^{-7}$
2^{-33}	1.0000000001164153	0.11694145202636719	$8.296621709646956 * 10^{-7}$
2^{-34}	1.0000000000582077	0.11694145202636719	$8.296621709646956 * 10^{-7}$
2^{-35}	1.0000000000291038	0.11693954467773438	$2.7370108037771956 * 10^{-6}$
2^{-36}	1.000000000014552	0.116943359375	$1.0776864618478044 * 10^{-6}$
2^{-37}	1.000000000007276	0.1169281005859375	$1.4181102600652196 * 10^{-5}$
2^{-38}	1.000000000003638	0.116943359375	$1.0776864618478044 * 10^{-6}$
2^{-39}	1.000000000001819	0.11688232421875	$5.9957469788152196 * 10^{-5}$
2^{-40}	1.0000000000009095	0.1168212890625	0.0001209926260381522
2^{-41}	1.0000000000004547	0.116943359375	$1.0776864618478044 * 10^{-6}$
2^{-42}	1.0000000000002274	0.11669921875	0.0002430629385381522
2^{-43}	1.0000000000001137	0.1162109375	0.0007313441885381522
2^{-44}	1.0000000000000568	0.1171875	0.0002452183114618478
2^{-45}	1.0000000000000284	0.11328125	0.003661031688538152
2^{-46}	1.0000000000000142	0.109375	0.007567281688538152
2^{-47}	1.000000000000007	0.109375	0.007567281688538152
2^{-48}	1.0000000000000036	0.09375	0.023192281688538152
2^{-49}	1.0000000000000018	0.125	0.008057718311461848
2^{-50}	1.0000000000000009	0.0	0.11694228168853815
2^{-51}	1.0000000000000004	0.0	0.11694228168853815
2^{-52}	1.0000000000000002	-0.5	0.6169422816885382
2^{-53}	1.0	0.0	0.11694228168853815
2^{-54}	1.0	0.0	0.11694228168853815

Zauważyłam, że błąd jest najmniejszy przy h rzędu 2^{-28}, gdzie błąd jest rzędu 10^{-9}. Po tej wartości błąd jedynie wzrasta by na koniec wynieść 100 procent. Dzieje się tak przez to, że małe liczby zmiennoprzecinkowe mają niewiele cyfr znaczących w zapisie, więc wraz z maleniem ich spada dokładność obliczeń.