Sprawozdanie

Nie interesuj się

15 kwietnia 2020

1 Zadanie Pierwsze

1.1 Epsilon maszynowy

Cel: Doświadczalne obliczenie epsilonu maszynowego (oznaczonej dalej jako *macheps*) oraz porównanie go z wynikami z innych źródeł.

Teoria: Zdefiniujmy zbiór F jako skończony zbiór liczb zmiennoprzecinkowych wtedy **epsilon maszynowy (macheps):** to:

$$(macheps, n \in F)(\forall n > 0)(\exists m > 0)(f(1 + macheps) > 1) \wedge (f(1 + n) > 1) \Rightarrow macheps \leq n$$

Przyrządy: Do wykonania doświadczenia użyję funkcji macheps zlokalizowanej w pliku *zadanie_ 1.jl*. Funkcja działa poprzez porównywanie w pętli sumy liczby 1 oraz zmiennej $\frac{macheps}{2}$, do liczby 1, aż do momentu w którym bedą one równe. Co iterację zmienna *macheps* jest dzielona przez 2. Charakterystyka funkcji wygląda następująco:

Wejście:	Typ zmiennoprzecinkowy na którym chcemy operować (w przypadku naszego doświadczenia Float16, Float32 lub Float64)
Wyjście:	Otrzymany epsilon maszynowy, z dokładnością do określonego na wejściu typu

Wyniki:

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

Wnioski:
Wyniki algorytmu pokrywają się z z wynikami otrzymanymi za pomocą funkcji bibliotecznych. Dowodzi to, że algorym działa poprawnie. Wraz ze wzrostem oczekiwanej precyzji, maleje także epsilon maszynowy. Precyzję arytmetyki wyraża się za pomocą wzoru $2^{-(t+1)}$, gdzie t to długość mantysy, podczas gdy epsilon maszynowy jest dwukrotnie większy i wynosi 2^{-t}.

1.2 Liczba η

Cel: Doświadczalne otrzymanie liczby η, czyli najmniejszej liczby dodatniej dla określonego typu zmiennoprzecinkowego

Teoria: Zdefiniujmy zbiór F jako skończony zbiór liczb zmiennoprzecinkowych wtedy η to:

$$(\eta, n \in F)(\forall n > 0)(\exists \eta > 0)(\eta > 0) \wedge (n > 0) \Rightarrow \eta \leq n$$

Przyrządy: Do wykonania doświadczenia użyję funkcji *etagen* zlokalizowanej w pliku *zadanie_ 1.jl*. Funkcja ta, zaczynając od liczby 1 i w kolejnych iteracjach pętli dzieląc ją na 2, szuka ostatniej liczby dodatniej różnej od zera. Charakterystyka funkcji wygląda następująco:

Wejście:	Typ zmiennoprzecinkowy na którym chcemy operować (w przypadku naszego doświadczenia Float16, Float32 lub Float64)
Wyjście:	Otrzymana liczba η, z dokładnością do określonego na wejściu typu

Wyniki:

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

Wnioski: Wyniki funkcji z biblioteki języka C oraz mojego algorytmu są takie same, co dowodzi jego poprawności. Obliczona przez nas wartość to według nazewnictwa z wykładu MIN_{sub}, podczas gdy MIN_{nor} jest od niej większe. Zachodzi relacja $MIN_{sub} = 2^{-(t-1)} * MIN_{nor}$.

1.3 Liczba *MAX*

Cel: Doświadczalne obliczenie największej liczby możliwej do zapisania (oznaczonej dalej jako MAX) w danym typie.

Teoria: Zdefiniujmy zbiór F jako skończony zbiór liczb zmiennoprzecinkowych wtedy $\boldsymbol{MAX}$ to:

$$(max, n \in F)(\forall n)(\exists max)(max \geq n)$$

Przyrządy: Do wykonania doświadczenia użyję funkcji *max_finder* zlokalizowanej w pliku *zadanie_1.jl*. Funkcja działa poprzez stopniowe mnożenie początkowej liczby dwa razy, a następnie po osiągnięciu $\frac{inf}{2}$, dodawaniu liczb kolejno liczb $\frac{max}{2}$, $\frac{max}{4}$, ...; aż do osiągnięcia *inf* w n+1 iteracji, wtedy funkcja przeywa działanie w n iteracji . Charakterystyka funkcji wygląda następująco:

Wejście:	Typ zmiennoprzecinkowy na którym chcemy operować (w przypadku naszego doświadczenia Float16, Float32 lub Float64)
Wyjście:	Otrzymana liczba MAX, z dokładnością do określonego na wejściu typu

Wyniki:

Typ	**algorytm**	**floatmax()**	**float.h**
Float16:	$6.55 * 10^{4}$	$6.55 * 10^{4}$	-
Float32:	$3.4028235 * 10^{38}$	$3.4028235 * 10^{38}$	$3.40282347 * 10^{38}$
Float64:	$1.7976931348623157 * 10^{308}$	$1.7976931348623157 * 10^{308}$	$1.7976931348623157 * 10^{308}$

Wnioski: Dane otrzymane za pomocą mojego algorytmu pokrywają się z danymi otrzymanymi z innych źródeł. Dowodzi to poprawności algorytmu.

2 Epsilon Maszynowy Kohena (Zadanie Drugie)

Cel: Doświadczalne udowodnienie twierdzenia Kohena

Teoria: Twierdzenie Kohena - epsilon maszynowy można otrzymać poprzez równanie:

$$macheps = 3 * (\frac{4}{3} - 1) - 1$$

Przyrządy: Do przeprowadzenia doświadczenia używał będę funkcji macheps_check zlokalizowanej w pliku zadanie_2.jl. Funkcja ta liczy ww. równamie krok po kroku. Następnie porównam je z wynikami funkcji *eps()* dołączonej do podstawowej biblioteki języka *julia*. Charakterystyka mojej funkcji wygląda następująco:

Wejście:	Typ zmiennoprzecinkowy na którym chcemy operować (w przypadku naszego doświadczenia Float16, Float32 lub Float64)
Wyjście:	Otrzymany epsilon Kehena, z dokładnością do określonego na wejściu typu

Wyniki:

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

Wnioski: Wyniki doświadczenia w większości nie pokrywają się z poprawnymi. Numeryczne wyniki się zgadzają, aczkolwiek dla typów Float16 oraz Float64 otrzymaliśmy liczbę przeciwną (tj. z innym znakiem, ujemną). Wynika to z faktu braku reprezentacji liczby $\frac{4}{3}$ w systemie binarnym, w związku z czym komputer musi je zaokrąglić zgodnie z zasadą *roundtoeven*, co skutkuje błędnym bitem znaku. Wynik byłby poprawny gdybyśmy używali wzoru:

$$macheps = |3 * (\frac{4}{3} - 1) - 1|$$

3 Rozmieszczenie liczb zmiennopozycyjnych (Zadanie Trzecie)

Cel: Badanie rozmieszczenia liczb zmiennopozycyjnych

Teoria: Między kolejnymi potęgami liczby 2, znajduje się zawsze tyle samo liczb zmiennoprzecinkowych, tj. Dla zbiorów X i Y będących podzbiorami zbioru liczb F - float:

$$(\forall n, m \in C)((X = [2^n, 2^{n+1}]) \wedge (Y = [2^m, 2^{m+1}])) \Rightarrow |X| = |Y|$$

Przyrządy: W celu udowodnienia prawdziwości tezy, użyję funkcji *float_ check* zlokalizowanej w pliku zadanie_3.jl. Funkcja ta na podstawie wybranej potęgi dwójki (n) iteruje po kolejnych liczbach z przedziału $[2^n, 2^{n+1}]$, w kierunku rosnącym, krokiem każdego przejścia jest liczba δ. Wyświetla ona także do konsoli liczby po których iteruje, zarówno w systemie dziesiętnym , jak i dwójkowym.

Wyniki:

Dla n = -1, przedział $[\frac{1}{2}, 1]$, $\delta = 2^{-53}$:

Bitstring	**Faktyczna wartość**
0011111111100000000000000000000000000000000000000000000000000000	0.5
0011111111100000000000000000000000000000000000000000000000000001	0.5000000000000001
0011111111100000000000000000000000000000000000000000000000000010	0.5000000000000002
0011111111100000000000000000000000000000000000000000000000000011	0.5000000000000003
0011111111100000000000000000000000000000000000000000000000000100	0.5000000000000004
0011111111100000000000000000000000000000000000000000000000000101	0.5000000000000006
0011111111100000000000000000000000000000000000000000000000000110	0.5000000000000007
0011111111100000000000000000000000000000000000000000000000000111	0.5000000000000008
...	...
0011111111101111111111111111111111111111111111111111111111111010	0.9999999999999993
0011111111101111111111111111111111111111111111111111111111111100	0.9999999999999996
0011111111101111111111111111111111111111111111111111111111111110	0.9999999999999998
0011111111110000000000000000000000000000000000000000000000000000	1.0

Dla n = 0, przedział $[1, 2]$, $\delta = 2^{-52}$:

Bitstring	**Faktyczna wartość**
0011111111110000000000000000000000000000000000000000000000000000	1.0
0011111111110000000000000000000000000000000000000000000000000001	1.0000000000000002
0011111111110000000000000000000000000000000000000000000000000010	1.0000000000000004
0011111111110000000000000000000000000000000000000000000000000011	1.0000000000000007
0011111111110000000000000000000000000000000000000000000000000100	1.0000000000000009
0011111111110000000000000000000000000000000000000000000000000101	1.000000000000001
0011111111110000000000000000000000000000000000000000000000000110	1.0000000000000013
0011111111110000000000000000000000000000000000000000000000000111	1.0000000000000016
...	...
0011111111111111111111111111111111111111111111111111111111111010	1.9999999999999987
0011111111111111111111111111111111111111111111111111111111111100	1.9999999999999991
0011111111111111111111111111111111111111111111111111111111111110	1.9999999999999996
0100000000000000000000000000000000000000000000000000000000000000	2.0

Dla n = 1, przedział $[2, 4]$, $\delta = 2^{-51}$:

Bitstring	**Faktyczna wartość**
0100000000000000000000000000000000000000000000000000000000000000	2.0
0100000000000000000000000000000000000000000000000000000000000001	2.0000000000000004
0100000000000000000000000000000000000000000000000000000000000010	2.000000000000001
0100000000000000000000000000000000000000000000000000000000000011	2.0000000000000013
0100000000000000000000000000000000000000000000000000000000000100	2.0000000000000018
0100000000000000000000000000000000000000000000000000000000000101	2.000000000000002
0100000000000000000000000000000000000000000000000000000000000110	2.0000000000000027
0100000000000000000000000000000000000000000000000000000000000111	2.000000000000003
...	...
0100000000001111111111111111111111111111111111111111111111111010	3.9999999999999973
0100000000001111111111111111111111111111111111111111111111111100	3.9999999999999982
0100000000001111111111111111111111111111111111111111111111111110	3.999999999999999
0100000000010000000000000000000000000000000000000000000000000000	4.0

Wnioski: Każdy krok δ zwiększa mantysę o 1 w systemie binarnym. Na tej podstawie można stwierdzić że gęstości liczb zmiennoprzecinkowych maleje wraz we zrostem przedziału między potęgami dwójki, ponieważ tyle samo liczb musi pokryć coraz większą powierzchnię. Dla każdego przedziału jest to jednak inny krok, określony wzorem $\delta = 2^{-52+n}$ gdzie n jest potęgą dwójki określającą dolną granicę przedziału.

4 Wpływ zakrągleń na wynik (Zadanie Czwarte)

Cel: Doświadczalne udowodnienie, że większość działań na typach zmiennoprzecinkowych obarczone jest błedem

Teoria: Operując na typie Float64, znalezienie liczby zmiennoprzecikowej spełniającej równanie:

$$x * \frac{1}{x} \neq 1, x \in [1, 2]$$

Przyrządy: W celu udowodnienia tezy, posłużą się funkcjami *check_ eq_ first* oraz *check_ eq_ last* zlokalizowanymi w pliku zadanie_4.jl. Funkcje te, poprzez sprawdzanie w pętli ww. równania i za każdym razem gdy ono zachodzi, przejście na następną lub poprzednią liczbę float, wskazaną przez funkcję nextfloat lub prevfloat.

Wyniki: Poniższa tabela ukazuje wynik otrzymany z pomocą tej funkcji:

Najmniejsza liczba:	**Największa liczba:**
1.000000057228997	1.9999999850988384

Wnioski: Działania na typach zmiennoprzecinkowych, takich jak Float64, związane są z zaokrągleniami, które powodują błędy. W tym przypadku największą rolę odegrał ułamek, który w niekórych przypadkach jest niemożliwy do zapisania w systemie dwójkowym, co spowodowało zaokrąglenie odbijające się potem na wyniku. W celu uniknięcia takich błędów należy obliczenia jak najmocniej uprościć, lub gdy to nie możliwe postarać się zwiększyć dokładność.

5 Obliczanie iloczynu skalarnego (Zadanie Piąte)

Cel: Zbadanie zależności wyników obliczeń od sposobu obliczeń.

Teoria: Iloczyn skalarny S dwóch wektorów A i B:

$$S = \sum_{i=1}^{n} a_i * b_i$$

Przyrządy: Doświadczenie polega na obliczeniu iloczynu skalarnego dla dwóch zadanych wektorów

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

za pomocą czterech różnych algorytmów: "w przód", "w tył", "od najmniejszego do największego" i "od największego do najmniejszego". Można jednak zauważyć że mnożenie poszczególnych składowych nie ma wpływu na wynik obliczeń, ponieważ przebiega on tak samo dla każdego algorytmu. Funkcja implementująca wszystkie ww. algorytmy nazywa się *scalar_ mult* i jest zlokalizowana w pliku zadanie_5.jl.

Opis algorytmów:

1. Algorytm 1 ("w przód") - przejście wprost po wektorze, sumując kolejne wyrazy, zaczynając od początku tablicy
2. Algorytm 2 ("w tył") - przejście wprost po wektorze, sumując kolejne wyrazy, zaczynając od końca tablicy
3. Algorytm 3 ("od najmniejszego do największego") - posortowanie iloczynów rosnąco a następnie dodanie ich po kolei do dwóch akumulatorów, jeden dla liczb dodatnich, drugi dla liczb ujemnych a ostatecznie zsumowanie obu akumulatorów.

4. Algorytm 4 (“od największego do najmniejszego”) - posortowanie iloczynów malejąco a następnie dodanie ich po kolei do dwóch akumulatorów, jeden dla liczb dodatnich, drugi dla liczb ujemnych a ostatecznie zsumowanie obu akumulatorów.

Charakterystyka formalna funkcji wygląda następująco:

Wejście:	Typ zmiennoprzecinkowy na którym chcemy operować (w przypadku naszego doświadczenia Float32 lub Float64)
Wyjście:	Otrzymane sumy dla każdego algorytmu, z dokładnością do określonego na wejściu typu

Wyniki:

Algorytm	**Float32**	**Float64**	**poprawna wartość**
1	-0.4999443	$1.0251881368296672 * 10^{-10}$	$-1.00657107000000 * 10^{-11}$
2	-0.4543457	$-1.5643308870494366 * 10^{-10}$	$-1.00657107000000 * 10^{-11}$
3	-0.5	0.0	$-1.00657107000000 * 10^{-11}$
4	-0.5	0.0	$-1.00657107000000 * 10^{-11}$

Wnioski: Jak pokazały wyniki, sposób wykonywania obliczeń, a konkretnie kolejność teoretycznie odwracalnych operacji ma ogromne znaczenie dla precyzji obliczeń. Suma cyfr znacząco od siebie różnych generuje większe błędy niż liczb sobie bliższych. Użycie Float64 znacząco zwiększa naszą precyzję, jednak nie pozwala osiągnąć prawidłowego wyniku. Algorytmy 3 i 4 zostały skonstruowane tak by wygenerować największy błąd (suma liczb najbardziej oddalonych od siebie).

6 Wpływ zmiany sposobu obliczeń na wynik (Zadanie Szóste)

Cel: Zbadanie zachowania dwóch funkcji, które są własnymi wyrażeniami alternatywnymi.

Teoria: Mamy dane funkcje *f(x)* oraz *g(x)*

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

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

Przyrządy: Do przeprowadzenia koniecznych obliczeń wykorzystam funkcję *test_functions* która automatyzuje funkcje *f1* oraz *f2* dla podanych wartości argumentu $x = 8^{-1}, 8^{-2}, 8^{-3}$.... Wszystkie funcjie znajdują się w pliku zadanie_6.jl.

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^{-0}	0.0	$2.7755575615628914 * 10^{-17}$

Wnioski: Wartości obu funkcji zmierzają do zera. Jednakże w przypadku funkcji *f*, jej dokładność szybko spada, ponieważ obejmuje ona bardzo bliskie sobie wartości $\sqrt{x^2 + 1}$ oraz 1. Funkcja g unika takich operacji i popełnia tylko błędy wynikające z określonej arytmetyki Float.

7 Obliczanie wartości pochodnej w punkcie (Zadanie Siódme)

Cel: Obliczenie wartości pochodnej wyrażonej wzorem

$$f(x) = sinx + cos3x$$

w punkcie $x = 1$

Teoria: Jesteśmy w stanie obliczyć przybliżoną wartość pochodnej w punkcie za pomocą wzoru:

$$\widetilde{f}'(x) = \frac{f(x_0 + h) - f(x_0)}{h}, h = 2^{-n}, n \in \{1, 2, 3, .., 54\}$$

Dodatkowo jestesmy w stanie oszacować błąd (β) powstały w wyniku takich obliczeń:

$$\beta = |f'(x_0) - \widetilde{f}(x_0)|$$

Przyrządy: Do obliczeń wykorzysta funkcję *diriv* znajdującą się pliku zadanie_7.jl

Wyniki:

h	$f'(x)$	$\widetilde{f}(x)$	β	$h+1$
2^0	0.11694228168853815	2.0179892252685967	1.9010469435800585	2.0
2^{-1}	0.11694228168853815	1.8704413979316472	1.753499116243109	1.5
2^{-3}	0.11694228168853815	0.6232412792975817	0.5062989976090435	1.125
2^{-5}	0.11694228168853815	0.24344307439754687	0.1265007927090087	1.03125
2^{-7}	0.11694228168853815	0.1484913953710958	0.03154911368255764	1.0078125
2^{-9}	0.11694228168853815	0.1248236929407085	0.007881411252170345	1.001953125
2^{-11}	0.11694228168853815	0.11891225046883847	0.001969968780300313	1.00048828125
2^{-13}	0.11694228168853815	0.11743474961076572	0.0004924679222275685	1.0001220703125
2^{-15}	0.11694228168853815	0.11706539714577957	0.00012311545724141837	1.000030517578125
2^{-17}	0.11694228168853815	0.11697306045971345	$3.077877117529937 * 10^{-5}$	1.0000076293945312
2^{-19}	0.11694228168853815	0.11694997636368498	$7.694675146829866 * 10^{-6}$	1.0000019073486328
2^{-21}	0.11694228168853815	0.1169442052487284	$1.9235601902423127 * 10^{-6}$	1.0000004768371582
2^{-23}	0.11694228168853815	0.11694276239722967	$4.807086915192826 * 10^{-7}$	1.0000001192092896
2^{-25}	0.11694228168853815	0.116942398250103	$1.1656156484463054 * 10^{-7}$	1.0000000298023224
2^{-26}	0.11694228168853815	0.11694233864545822	$5.6956920069239914 * 10^{-8}$	1.0000000149011612
2^{-27}	0.11694228168853815	0.11694231629371643	$3.460517827846843 * 10^{-8}$	1.0000000074505806
2^{-28}	0.11694228168853815	0.11694228649139404	$4.802855890773117 * 10^{-9}$	1.0000000037252903
2^{-29}	0.11694228168853815	0.11694222688674927	$5.480178888461751 * 10^{-8}$	1.0000000018626451
2^{-30}	0.11694228168853815	0.11694216728210449	$1.1440643366000813 * 10^{-7}$	1.0000000009313226
2^{-32}	0.11694228168853815	0.11694192886352539	$3.5282501276157063 * 10^{-7}$	1.0000000002328306
2^{-34}	0.11694228168853815	0.11694145202636719	$8.296621709646956 * 10^{-7}$	1.0000000000582077
2^{-36}	0.11694228168853815	0.116943359375	$1.0776864618478044 * 10^{-6}$	1.000000000014552
2^{-38}	0.11694228168853815	0.116943359375	$1.0776864618478044 * 10^{-6}$	1.000000000003638
2^{-40}	0.11694228168853815	0.1168212890625	0.0001209926260381522	1.0000000000009095
2^{-42}	0.11694228168853815	0.11669921875	0.0002430629385381522	1.0000000000002274
2^{-44}	0.11694228168853815	0.1171875	0.0002452183114618478	1.0000000000000568
2^{-46}	0.11694228168853815	0.109375	0.007567281688538152	1.0000000000000142
2^{-48}	0.11694228168853815	0.09375	0.023192281688538152	1.0000000000000036
2^{-50}	0.11694228168853815	0.0	0.11694228168853815	1.0000000000000009
2^{-53}	0.11694228168853815	0.0	0.11694228168853815	1.0

Wnioski: Można zauważyć, że algorytm najlepsze wyniki otrzymał przy $h = 2^{-28}$ (najmniejszy błąd), a dalej dokładność ulega coraz większemu pogorszeniu. Jest to związane z '*pochłanianiem*' liczby h przez jedynkę, co najlepiej obrazuje zbiżanie się do $h = 2^{-54}$. Dadatkowo, należy wziąć pod uwagę utratę cyfr znaczących przu obliczaniu błędu, co jest spowodowane odejmowaniem bardzo bliskich sobie liczb.