Sprawozdanie
Obliczenia naukowe - lista 1

Kamil Król

Zadanie 1

MachEps

Epsilonem maszynowym macheps (ang. machine epsilon) nazywamy najmniejszą liczbę macheps większą od 0 taką, że fl(1.0 + macheps)>1.0.

W celu wyznaczenia metodą iteracyjną *macheps* zgodnego z powyższą definicją napisałem program, a jego wyniki zamieściłem w poniższej tabeli. Zgodnie z treścią zadania program uruchomiłem dla typów Float16, Float32 oraz Float64 i porównałem z wartościami zwracanymi przez funkcję *eps* dla każdego z typów.

Iteracyjne wyznaczanie macheps

	obliczony *macheps*	eps(type)	wartość z pliku float.h
Float16	0.000977	0.000977	xd
Float32	1.1920929e-7	1.1920929e-7	xd
Float64	2.220446049250313e-16	2.220446049250313e-16	xd

Okazało się, że wartości *macheps* wyznaczone przeze mnie są równe wartościom zwracanym przez wbudowaną w język Julia funkcją *eps*.

W treści zadania pojawia się pytanie: jaki związek ma liczba *macheps* z precyzją arytmetyki (oznaczaną na wykładzie przez ϵ)? W celu odpowiedzi na to pytanie przytoczę najpierw definicję precyzji arytmetyki - ϵ. Jest to największy błąd względny reprezentacji liczby jaki możemy popełnić i dla liczb reprezentowanych zgodnie ze standardem IEEE-754 wyraża się on wzorem: 2^{-t}. Podstawiając do wzoru dla arytmetyki Float32 mamy:

$$\epsilon = 2^{-24} = 0.5 \cdot 2^{-23} = \frac{1}{2} \cdot macheps$$

Wartość *macheps* dla Float32 w tabeli tj. 1.1920929e-7 jest zaokrąglona. Jej dokładna wartość wynosi: 1.1920928955078125e-7 co jest równe 2^{-23} (stąd równość). *Macheps* jest w komputerze przechowywany dokładnie. Wykonując to rozumowanie dla wszystkich typów widzimy zgodność i otrzymujemy zależność: $macheps = 2\epsilon$.

Eta

Kolejnym zadaniem jest wyznaczenie liczby *eta* takiej, że $eta > 0.0$ dla wszystkich typów zmiennopozycyjnych Float16, Float32, Float64. Wyniki napisanego przeze mnie programu, który iteracyjnie wyznacza te liczby, umieściłem w poniższej tabeli. Ponadto wartości otrzymanych liczb *eta* porównałem z wartościami zwracanymi przez funkcje: *nextfloat*(Float16(0.0)), *nextfloat*(Float32(0.0)), *nextfloat*(Float64(0.0)

Iteracyjne wyznaczanie eta

	obliczona *eta*	nextfloat(type(0))
Float16	6.0e-8	6.0e-8
Float32	1.0e-45	1.0e-45
Float64	5.0e-324	5.0e-324

Wartości obliczone przeze mnie okazały się takie same jak zwrócone przez funkcje wbudowane w język Julia. Kolejnym pytaniem jest: Jaki związek ma liczba *eta* z liczbą MIN_{sub}?
MIN_{sub} jest najmniejszą liczbą zdenormalizowaną (subnormalną), tzn. taką gdzie cecha liczby jest wypełniona zerami. Inaczej najmniejsza możliwa do przechowania w danym systemie liczba. Liczba *eta* jest równa liczbie MIN_{sub}, są one tożsame. Są to liczby tak małe, że nie da się ich pomniejszyć manipulując cechą. TO DO
Innym pytaniem z treści zadania jest: co zwracają funkcje *floatmin*(Float32) i *floatmin*(Float64) i jaki jest związek zwracanych wartości z liczbą MIN_{nor}?
TO DO Funkcje te zwracają najmniejsze liczby znormalizowane dla danego typu, a jest to dokładnie MIN_{nor}. Liczby znromalizowane to takie gdzie w mantysie zakładamy niepisaną jedynkę, tzn. wartość mantysy '0011...' oznacza '1.0011...' (inaczej niż w subnormalnych) i cecha nie jest zerem. Inaczej są to liczby, które można pomniejszyć zmniejszając wartość cechy.

Liczba MAX

Kolejnym zadaniem do zrobienia było wyznaczenie (iteracyjnie) liczby MAX dla wszystkich typów zmiennopozycyjnych Float16, Float32, Float64 i porównanie wyników z wartościami zwracanymi przez funkcje: *floatmax*(Float16), *floatmax*(Float32), *floatmax*(Float64) oraz z danymi zawartymi w pliku nagłówkowym float.h dowolnej instalacji języka C. Liczbę MAX interpretuję jako największą wartość jaką można przechować w danym typie zmiennoprzecinkowym. Przy wyznaczaniu tej wartości musiałem pamiętać aby mantysa była wypełniona jedynkami. By to uzyskać postanowiłem wziąć liczbę *zaraz przed* liczbą 2.0 czyli 2.0 - *macheps*. Ten rezultat mogłem uzyskać też biorąc liczbę *zaraz przed* 1.0, wtedy byłoby to $1.0 - \frac{macheps}{2}$.

Ponownie wartości wyznaczone przeze mnie okazały się takie same jak te wyznaczone przez funkcje z języka Julia.

Iteracyjne wyznaczanie liczby MAX

	Obliczony MAX	maxfloat(type)	wartość z pliku float.h
Float16	6.55e4	6.55e4	xd
Float32	3.4028235e38	3.4028235e38	xd
Float64	1.7976931348623157e308	1.7976931348623157e308	xd

Zadanie 2

Zadanie to dotyczyło sprawdzenia eksperymentalnie w języku Julia słuszności tezy Kahana dla wszystkich typów zmiennopozycyjnych Float16, Float32, Float64. Wyniki programu zamieściłem w poniższej tabeli:

	Obliczony eps wg. wzoru Kahana	Wartość funkcji eps(type)
Float16	-2.220446049250313e-16	2.220446049250313e-16
Float32	1.1920929e-7	1.1920929e-7
Float64	-2.220446049250313e-16	2.220446049250313e-16

Ponadto mój program sprawdzał, czy wartości bezwzględne otrzymanych eps są równe i okazało się że tak jest. Wnioskiem z doświadczenia jest to, że teza Kahana jest słuszna - macheps można obliczyć stosując zaproponowany przez niego wzór. W celu otrzymania macheps należy na wynik otrzymany ze wzoru $3(4/3 - 1) - 1$ nałożyć wartość bezwzględną.

Zadanie 3

Celem tego zadania było eksperymentalne sprawdzenie, że w arytmetyce Float64 liczby zmiennopozycyjne są równomiernie rozmieszczone w [1, 2] z krokiem $\delta = 2^{-52}$. Mój program zaczynał od liczby 1.0 i z każdą iteracją dodawał do niej liczbę δ. W celu zidentyfikowania czy wybrany krok jest poprawny drukowałem kolejne wartości liczb i obserwowałem jak się zmieniają. Rezultaty w tabeli poniżej.

	bitowa reprezentacja
$1.0 + 1\delta$	0—01111111111—0000000000000000000000000000000000000000000000000001
$1.0 + 2\delta$	0—01111111111—0000000000000000000000000000000000000000000000000010
$1.0 + 3\delta$	0—01111111111—0000000000000000000000000000000000000000000000000011
$1.0 + 4\delta$	0—01111111111—0000000000000000000000000000000000000000000000000100
$1.0 + 5\delta$	0—01111111111—0000000000000000000000000000000000000000000000000101
$1.0 + 6\delta$	0—01111111111—0000000000000000000000000000000000000000000000000110
$1.0 + 7\delta$	0—01111111111—0000000000000000000000000000000000000000000000000111
$1.0 + 8\delta$	0—01111111111—0000000000000000000000000000000000000000000000001000
...	...

Widać w kolejnych rekordach, że wybrany krok - δ jest poprawny (dla przedziału [1,2]), ponieważ wartości zmieniają się na najmniej znaczących bitach (na końcu) w sposób umożliwiający przejście przez wszystkie możliwe wartości mantysy.
Pojawia się pytanie: Jak rozmieszczone są liczby zmiennopozycyjne w przedziale $[\frac{1}{2}, 1]$, a jak w przedziale [2, 4] i jak mogą być przedstawione dla rozpatrywanego przedziału?
Liczbę δ dla przedziału $[\frac{1}{2}, 1]$ będziemy nazywać δ_1, a dla [2, 4] δ_4. Liczb we wszystkich trzech wspomianych przedziałach jest tyle samo (w tej reprezentacji). Np. między 1 i 2 jest tyle samo liczb co między 2 i 4, a tych jest tyle samo co w przedziale [4,8] itd. Granicami tych przedziałów są potęgi liczby 2. Jak można z tą wiedzą wyznaczyć δ dla przedziałów $[\frac{1}{2}, 1]$ i [2, 4]? Wiemy, że δ dla przedziału [1,2] wynosi $\delta = 2^{-52}$. Długość przedziału $[\frac{1}{2}, 1]$ jest 2 razy mniejsza od długości [1,2], więc naturalnym kandydatem dla liczby δ_1 (dla przedziału $[\frac{1}{2}, 1]$) wydaje się liczba δ podzielona przez 2. Zatem mamy $\delta_1 = 2^{-53}$.
Przedział [2, 4] ma długość dwa razy większą niż [1, 2], więc kandydatem na liczbę δ_4 będzie liczba 2 razy więszka od δ. W związku z tym mamy $\delta_4 = 2^{-51}$.
Pozostaje te wartości eksperymentalnie sprawdzić. W poniższej tabeli znajdują się wyniki programu, który sprawdza liczby δ_1 i δ_4 w analogiczny sposób jak przy sprawdzaniu δ dla przedziału [1,2].

$\delta_1 = 2^{-53}$

	bitowa reprezentacja
$0.5 + 1\delta$	0—01111111110—0000000000000000000000000000000000000000000000000001
$0.5 + 2\delta$	0—01111111110—0000000000000000000000000000000000000000000000000010
$0.5 + 3\delta$	0—01111111110—0000000000000000000000000000000000000000000000000011
$0.5 + 4\delta$	0—01111111110—0000000000000000000000000000000000000000000000000100
$0.5 + 5\delta$	0—01111111110—0000000000000000000000000000000000000000000000000101
$0.5 + 6\delta$	0—01111111110—0000000000000000000000000000000000000000000000000110
$0.5 + 7\delta$	0—01111111110—0000000000000000000000000000000000000000000000000111
$0.5 + 8\delta$	0—01111111110—0000000000000000000000000000000000000000000000001000
$0.5 + 9\delta$	0—01111111110—0000000000000000000000000000000000000000000000001001

$\delta_4 = 2^{-51}$

	bitowa reprezentacja
$2.0 + 1\delta$	0—10000000000—0000000000000000000000000000000000000000000000000001
$2.0 + 2\delta$	0—10000000000—0000000000000000000000000000000000000000000000000010
$2.0 + 3\delta$	0—10000000000—0000000000000000000000000000000000000000000000000011
$2.0 + 4\delta$	0—10000000000—0000000000000000000000000000000000000000000000000100
$2.0 + 5\delta$	0—10000000000—0000000000000000000000000000000000000000000000000101
$2.0 + 6\delta$	0—10000000000—0000000000000000000000000000000000000000000000000110
$2.0 + 7\delta$	0—10000000000—0000000000000000000000000000000000000000000000000111
$2.0 + 8\delta$	0—10000000000—0000000000000000000000000000000000000000000000001000
$2.0 + 9\delta$	0—10000000000—0000000000000000000000000000000000000000000000001001

Widać, że wartości δ_1 i δ_4 są poprawne z tego samego powodu co poprzednio - wartości zmieniają się na najmniej znaczących bitach (na końcu) w sposób umożliwiający przejście przez wszystkie możliwe wartości mantysy.

Zadanie 4

Celem tego zadania było eksperymentalne znalezienie w arytmetyce Float64 (double) najmniejszej liczby zmiennopozycyjnej x w przedziale (1,2) takiej, że $x \cdot \frac{1}{x} \neq 1$. Wynik mojego programu to: 1.000000057228997. Reprezentacja tej liczby w arytmetyce Float64 to:
0 01111111111 0000000000000000000000011110101110010111111100101010.
Mantysa zinterpretowana jako samodzielna liczba to 257736490. Oznacza to, że program musiał wykonać aż 257736490 - 1 = 257736480 iteracji. TO DO
Wnioski?

Zadanie 5

Zadanie to polegało na eksperymentalnym obliczeniu iloczynu skalarnego dwóch wektorów x i y.
x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]
y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049].
W treści zadania są podane 4 algorytmy:
a (forward),b (backward),c (descending),d (ascending), które zaimplementowałem. Algorytmy są opisane w treści zadania, więc nie będę ich tu opisywać. Kolejnym krokiem w zadaniu było porównanie otrzymanych wartości z wartościa prawdziwą, zrealizowałem to licząc błąd bezwzględny i względny. Przypomnijmy - wartość dokładna (z dokładnością do 15 cyfr, według treści zadania) wynosi $-1.00657107 \cdot 10^{-11}$. Wyniki mojego programu znajdują się w tabeli poniżej:

Dla Float64

alogrytm	obliczona wartość	błąd bezwzględny	błąd względny
Forward	1.0251881368296672e-10	1.1258452438296672e-10	11.184955313981627
Backward	-1.5643308870494366e-10	1.4636737800494365e-10	14.541186645165915
Descending	0.0	1.00657107e-11	1.0
Ascending	0.0	1.00657107e-11	1.0

Dla Float32

alogrytm	obliczona wartość	błąd bezwzględny	błąd względny
Forward	-0.4999443	0.49994429944939167	4.9668057661282845e10
Backward	-0.4543457	0.4543457031149343	4.51379655800096e10
Descending	-0.5	0.4999999999899343	4.967359135306107e10
Ascending	-0.5	0.4999999999899343	4.967359135306107e10

Patrząc na wyniki nasuwa się kilka ważnych wniosków. Wartości obliczone przez program nie są równe wartości dokładnej bez względu na użyty algorytm. Ponadto różnice w wartościach obliczonych przez poszczególne algorytmy okazały się kompletnie nieintuicyjne. Algorytm c (descending), który wydaje się *najgorszy* okazał się widocznie lepszy od algorytmu a i b (dla Float64) i na dodatek dał taki sam wynik jak algorytm d (ascending) (dla Float64 i Float32), który wydaje się *najlepszy*. Przy czym przez *najlepszy/najgorszy* rozumiem taki który oblicza wartość odpowiednio najdokładniej/najmniej dokładnie.
Inną rzeczą wartą zauważenia jest też fakt, że różnice w błędach dla dwóch testowanych typów okazały się bardzo duże - błędy dla Float64 były znacznie mniejsze. Kolejnym wnioskiem jest to, że licząc iloczyn skalarny dla Float64 otrzymaliśmy w 2 przypadkach wartość 0.0, co nieuważnemu użytkownikowi dałoby podstawy żeby twierdzić, że wekotry x i y są prostopadłe, kiedy w rzeczywistości tak nie jest.

Second Section

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisissem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi necante...