Sprawozdanie

OBLICZENIA NAUKOWE, LISTA NR 1

ŁUKASZ KLEKOWSKI 229738

Wrocław 21.10.2017

1. Zadanie pierwsze

1.1.Opis problemu

Polecenie polegało na napisaniu w języku Julia programów iteracyjnie obliczających epsilonu maszynowego (macheps), najmniejszej dodatniej liczby rzeczywistej (eta) oraz wartości maksymalnej dla trzech licz typów zmiennopozycyjnych : Float16, Float32, Float64. Należało także porównać otrzymane wartości ze stałymi ze środowiska Julia: eps(), nextfloat(0.0) oraz realmax(). Epsilon maszynowy oraz wartości maksymalne należało porównać także z danymi zawartymi w pliku nagłówkowym float.h w dowolnej instalacji języka C.

1.2.Rozwiązanie problemu

Do obliczenia zera maszynowego użyłem pętli while która dzieliła liczbę 1.0 dopóki nie był spełniony dany warunek:

$$1.0 + \frac{macheps}{2} = 1.0$$

Podobnie wyliczyłem najmniejszą dodatnią liczbę rzeczywistą. Dzieliłem w pętli liczbę 1.0 do momentu gdy:

$$\frac{eta}{2} \leq 0.0$$

Natomiast do obliczenia największej wartości użyłem pętli która mnożyła liczbę 1.0 przez 2 do momentu gdy funkcja isinf(), która sprawdza czy dana liczba to nieskończoność zwróci true. Następnie przy pomocy funkcji prevfloat() odczytujemy poprzednią wartość floata, a w naszym przypadku jest to wartość maksymalna.

1.3.Wyniki

1.3.1. Macheps

Julia

	machepsFloat()	eps()
Float16	9.765625e-4	9.765625e-4
Float32	1.1920928955078125e-7	1.1920928955078125e-7
Float64	2.22044604925031308e-16	2.22044604925031308e-16

Język C

	FLT_EPSILON / DBL_EPSILON
Float (Float32)	1.1920928955078125e-7
Double (Float64)	2.2204460492503131e-16

Epsilony obliczone w pętlach są takie same jak te uzyskane za pomocą funkcji eps(). W pliku nagłówkowym float.h nie ma informacji na temat epsilona dla arytmetyki Float16, natomiast dla Float32 epsilon jest taki sam jak w Julii, a dla Float64 jest trochę zaokrąglone.

1.3.2. Eta

	eta()	nextfloat(0.0)
Float16	5.9604644775390625e-8	5.9604644775390625e-8
Float32	1.40129846432e-45	1.40129846432e-45
Float64	4.9e-324	4.9e-324

Obliczone eta zgadza się z liczbą uzyskaną za pomocą funkcji nextfloat(0.0)

1.3.3. MAX

Julia

	inf()	realmax()
Float16	65504.0	65504.0
Float32	3.4028234664e38	3.4028234664e38
Float64	1.7976931348623157e308	1.7976931348623157e308

Język C

	FLT_MAX / DBL_MAX
Float	3.40282346638528860000000000000e38
Double	1.79769313486231570000000000000000e308

Wyniki z pętli są takie same jak liczby otrzymane z funkcji realmax(). W pliku nagłówkowym float.h również nie ma informacji co do Float16. Natomiast Float i Double są bardzo zaogrąglone w porównaniu do wyników otrzymywanych w Julii.

1.4. Wnioski

W zadaniu było zawarte pytanie jaki związek ma liczba *macheps* z precyzją arytmetyki oraz jaki związek ma liczba eta z liczbąMIN_{sub}.

Precyzja arytmetyki oraz epsilon maszynowy jest to maksymalny błąd względny, który może zajść podczas zaokrąglania liczby rzeczywistej do liczby zmiennoprzecinkowej.

Liczba MIN_{sub} jest tym samy co epsilon czyli najmniejszą możliwą liczbą większą od 0.

2. Zadanie drugie

2.1. Opis zadania

Polecenie polegało na sprawdzeniu słuszności stwierdzenia Kahana, czy epsilon maszynowy można uzyskać za pomocą wzoru $3 * \left(\frac{4}{3} - 1\right) - 1$ w precyzji Float16, Float32, oraz Float64.

2.2. Rozwiązanie

Napisałem w języku Julia funkcję która liczy podane wyrażenie

2.3. Wyniki

Float16	Float32	Float64
-9.765625e-4	1.1920928955078125e-7	-2.22044604925031308e-16

2.4.Wnioski

Nie otrzymaliśmy zera choć teoretycznie powinniśmy. Komputer nie jest w stanie dokładnie zapisać liczby 4/3 więc musi zaokrąglić ją do najbliższej liczby w arytmetyce zmiennopozycyjnej. Powstały błąd to właśnie epsilon maszynowy który może powstać podczas zaokrąglania liczby.

3. Zadanie trzecie

3.1.Opis zadania

Polecenie polegało na sprawdzeniu rozmieszczenia liczb zmiennopozycejnych w precyzji Float64 dla przedziałów [0.5-1], [1-2] [2-4].

3.2.Opis rozwiązania i wyniki

Użyłem tutaj funkcji bits() dzięki której dostałem binarną reprezentację danej liczby.
W pętli dla danych przedziałów zwiększałem liczbę o dany krok i sprawdzałem jak będzie wyglądała ich reprezentacja binarna, jednocześnie porównując je z nextfloat()

[1-2]

	bits()
1.0	0011111111110000000000000000000000000000000000000000000000000000
1.0 + 2-52	0011111111110000000000000000000000000000000000000000000000000001
nextfloat(1.0)	0011111111110000000000000000000000000000000000000000000000000001
1.0+2*2-52	0011111111110000000000000000000000000000000000000000000000000010

[0.5-1]

	bits()
0.5	0011111111100000000000000000000000000000000000000000000000000000
0.5 + 2-53	0011111111100000000000000000000000000000000000000000000000000001
nextfloat(0.5)	0011111111100000000000000000000000000000000000000000000000000001
0.5+2*2-53	0011111111100000000000000000000000000000000000000000000000000010

[2-4]

	bits()
2.0	0100000000000000000000000000000000000000000000000000000000000000
2.0 + 2-51	0100000000000000000000000000000000000000000000000000000000000001
nextfloat(2.0)	0100000000000000000000000000000000000000000000000000000000000001
2.0+2*2-51	0100000000000000000000000000000000000000000000000000000000000010

3.3.Wnioski

Z przeprowadzonego eksperymentu wynika że liczby w przedziale [1-2] są rozłożone równomiernie z krokiem 2-52, w przedziale [0.5-1] z krokiem 2-53, a w przedziale [2-4] z krokiem 2-51.

Wynika to oczywiście z charakterystyki tej arytmetyki ponieważ liczb w takich przedziałach jest tyle samo, a te przedziały się zwiększają więc są one rozłożone bardziej rzadko lub gęściej.

4. Zadanie czwarte

4.1. Opis zadania

Polecenie polegało na znalezienie najmniejszej takiej liczby w przedziale (1-2) w precyzji Float64 która spełnia warunek x*x!=1

4.2. Rozwiązanie

W pętli do liczby 1.0 był dodawany epsilon maszynowy i następnie za pomocą powstałej liczby liczony był dany wzór. Gdy wynik był różny od 1 to dana liczba jest drukowana na ekranie.

4.3. Wyniki

Najmniejsza liczba która nie spełnia tego równania to :

1.000000057228997

4.4. Wnioski

Takie liczby są znajdowane dlatego, że komputer nie potrafi zapisać dokładnie każdej liczby w postaci 1/x w arytmetyce zmiennopozycyjnej.

5. Zadanie piąte

5.1. Opis zadania

Polecenie polegało na obliczeniu iloczynu skalarnego dwóch wektorów za pomocą czterech algorytmów używając precyzji Float32 oraz Float64.

5.2. Rozwiązanie

Pierwszy algorytm polegał na zwykłym obliczeniu produktu skalarnego sumując po kolei iloczyny. Drugi algorytm robił to w odwrotnej kolejności. Trzeci algorytm polegał na dodawaniu wyników mnożenia składowych wektora od najmniejszej wartości do największej, a czwarty algorytm robił to na odwrót.

5.3. Wyniki

Algorytm	Float32	Float64	Błąd względny F32	Błąd względny F64
I	-4.9994429945e-1	1.02518813e-10	4.996159e10	1.1245e1
II	-4.5434570312500e-1	-1.56433088e-10	4.540473e10	1.463e1
III	-5.000e-1	0	4.996716e10	-
IV	-5.000e-1	0	-4.9967165	-

5.4. Wnioski

Żaden z algorytmów nie dał wyniku choć przybliżonego do wartości poprawnej. W przypadku precyzji Float32 wyniki sa bardzo dalekie od rzeczywistych. W przypadku precyzji Float64 pierwszy oraz drugi algorytm dawał wyniki bliskie do wyniku precyzyjnego, a trzeci oraz czwarty algorytm dawał wartości 0. Okazuje się że kolejność wykonywania działań w komputerze ma ogromne znaczenie.

6. Zadanie szóste

6.1.Opis zadania

Polecenie polegało na obliczeniu wartości dwóch funkcji dla argumentów 8^{-1} , 8^{-2} itd.

6.2.Wyniki

x	f(x)	g(x)
8^{-1}	0.007782218537318641438105 3	0.007782218537318706490235 6
8^{-2}	0.000122062862828675733339 9	0.000122062862828759013619 3
8^{-3}	0.000001907346813823096454 1	0.000001907346813826565901 1
8^{-4}	0.000000029802321943606102 6	0.000000029802321943606115 9
8^{-5}	0.000000000465661287307739 3	0.000000000465661287199319 0
8^{-6}	0.000000000007275957614183 4	0.000000000007275957614157 0
8^{-7}	0.000000000000113686837721 6	0.000000000000113686837721 6
8^{-8}	0.000000000000001776356839 4	0.000000000000001776356839 4
8^{-9}	0.000000000000000000000000	0.000000000000000027755575 6
8^{-10}	0.000000000000000000000000	0.000000000000000000433680 9

6.3.Wnioski

Po sprawdzeniu otrzymanych wyników w pakiecie matematycznym wolfram można stwierdzić że wyniki otrzymane za pomocą funkcji g(x) są dokładniejsze. Funkcja f od argumentu 8-9 zwraca 0. To zadanie również pokazuje że obliczenia wykonywane w różny sposób mogą dawać różne wyniki.

7. Zadanie siódme

7.1.Opis zadania

Polecenie polegało na obliczeniu wartości pochodnej danej funkcji w punkcie xo=1. Należało użyć wzoru do przybliżonej wartości pochodnej oraz porównać ją z dokładną pochodną. Obliczenia były przeprowadzane w precyzji Float64.

7.2.Wyniki

h	Przybliżona pochodna	Błąd
2^{0}	2.01798922526859669801524 43	1.90104694358005854581961 10
2^{-1}	1.87044139793164720941831 55	1.75349911624310905722268 22
2^{-2}	1.10778709523429741423683 48	0.99084481354575926204120 15
2^{-3}	0.62324127929758166288820 58	0.50629899760904351069257 25
2^{-4}	0.37040006620351917376865 46	0.25345778451498102157302 13
2^{-10}	0.12088247681106167874531 82	0.00394019512252352654968 49
2^{-11}	0.11891225046883846516720 95	0.00196996878030031297157 62
2^{-12}	0.11792723373901026207022 37	0.00098495205047210987459 04
2^{-13}	0.11743474961076572071760 89	0.00049222816885381521956 33
2^{-14}	0.11718851362093118950724 60	0.00024623193233930373116 27
2^{-35}	0.11693954467773437500000 00	0.00000273701080377719563 33
2^{-40}	0.11682128906250000000000 00	0.00012099262603815219563 33

2^{-50}	0.00000000000000000000000000	0.11694228168853815219563333
2^{-51}	0.00000000000000000000000000	0.11694228168853815219563333
2^{-54}	0.00000000000000000000000000	0.11694228168853815219563333

7.3.Wnioski

Zmniejszanie wartości h od pewnego momentu 2-35 przestało przybliżać nas do prawidłowego wyniku i następne przybliżenia były obarczone coraz większym błędem.