Obliczenia Naukowe Lista 1

Kacper Majkowski

25 października 2023

1 Zadanie 1

1.1 Macheps

Problem:

Znaleść epsilon maszynowy (macheps) dla różnych typów liczbowych
- Float16, Float32, Float64

Rozwiązanie:

Program początkowo przyjjmuje zmienną x = 2. Następnie wylicza on średnią z 1 i x, a wynik przyjmuje jako nowy x. Proces ten jest powtarzany aż średnia z 1 i x bedzie równa 1. Oznacza to że nie ma żadnych dostępnych liczb pomiędzy 1 a wyliczonym x. Wtedy macheps, czyli odległość pomiędzy 1 a następną liczbą jest równy x - 1.

Wyniki:

Porównanie wyliczonych epsilonów maszynowych z funkcją eps() oraz biblioteką float.h w C:

Typ liczbowy	Wyliczony macheps	Wynik funkcji eps()	Wartość w bibliotece float.h
Float16	0.000977	0.000977	-
Float32	1.1920929e-7	1.1920929e-7	1.19209290e-07
Float64	2.220446049250313e-16	2.220446049250313e-16	2.2204460492503131e-16

Wnioski:

Jeżeli obliczymy precyzje arytmetyki dla tych typów to okaże się że są one wartościami równe z ich epsilonami maszynowymi. Dla przykładu, precyzja

Float16, gdzie na część ułamkową przeznaczone jest 10 bitów wynosi $(1/2) * 2^9 = 0.0009765625$, co po zaokrągleniu daje nam 0.000977, czyli wyliczony macheps dla Float16. Podobnie wygląda sytuacja z precyzjami Float32 oraz Float64. Wynika to z faktu, że liczba 1 w formacie float jest zapisana jako (...)|100...000 (liczba zer zależy od typu float). Aby otrzymać następną liczbę trzeba zmienić ostatni bit z 0 na 1, czyli innymi słowy dodać najmniejszą liczbę na jaką pozwala precyzja arytmetyki.

1.2 Eta

Problem:

Znaleść najmniejszą liczbę większą od 0 (eta) dla różnych typów liczbowych - Float16, Float32, Float64

Rozwiązanie:

Rozwiązanie wygląda bardzo podobnie do wyliczania machepsu. Program początkowo przyjmuje zmienną x = 1. Dzieli on x przez 2, a wynik przyjmuje jako nowy x. Proces ten jest powtarzany x/2 bedzie równa 0. Oznacza to że nie ma żadnych dostępnych liczb pomiędzy 0 a wyliczonym x. Wtedy eta, czyli odległość pomiędzy 0 a następną liczbą jest równy x.

Wyniki:

Porównanie wyliczonych eta z funkcją nextfloat() i MINsub :

Typ liczbowy	Wyliczony eta	Wynik funkcji nextfloat()	MINsub
Float16	6.0e-8	6.0e-8	5.96e-8
Float32	1.0e-45	1.0e-45	1.4e-45
Float64	5.0e-324	5.0e-324	4.9e-324

Wnioski:

Wyliczone przez nas eta są bardzo zbliżone do wartości MINsub dla danych typów float (równe z dokładnością do zaokrąglenia). Nie powinno być to zaskoczenie gdyż MINsub (Minimal subnormal) z definicji jest najmniejszą liczbą większą od zera, czyli tym co chcieliśmy wyliczyć. Z drugie strony mamy również MINnor (Minimal Normal), czyli najmniejszą liczbę znormalizowaną. Jest to liczba którą dostajemy po wywołaniu funkcji floatmin():

Typ liczbowy	Wynik funkcji floatmin()	MINnor
Float32	1.1754944e-38	1.2 E-38
Float64	2.2250738585072014e-308	2.2 E-308

Wartości MINnor zostały wzięte z raportu IEEE Standard 754 for Binary Floating-Point Arithmetic

Problem:

Znaleść maksymalną dostępną liczbę dla różnych typów liczbowych
- Float16, Float32, Float64

Rozwiązanie:

Program przyjmuje zmienną x = 1. Następnie mnoży x razy 2 aż do momentu gdy następne takie mnożenie dałoby wynik równy nieskończoności. W ten sposób wyliczyliśmy największą potęgę 2 mniejszą od maxfloat'a. Program tworzy wtedy zmienną adder równą x/2. Dodaje ją do x aż takie dodawanie nie dałoby nieskończoności. Wtedy dzieli adder przez 2 i powtarza dodawanie. Robi tak dopóki dodawanie addera zmienia wartość x, czyli nie jest on pomijalnie mały. Gdy adder jest już pomijalnie mały, wiemy że nie można dodać żadnej liczby do x, tak aby nie uzyskać nieskończoności, czyli a jest to maksymalna dostępna liczba.

Wyniki:

Porównanie wyliczonych maksymalnych liczb z funkcją floatmax() oraz biblioteką float.h w C:

Typ liczbowy	Wyliczony max float	Wynik funkcji floatmax()	Wartość w bibliotece float.h
Float16	6.55e4	6.55e4	-
Float32	3.4028235e38	3.4028235e38	3.402823466e38
Float64	1.7976931348623157e308	1.7976931348623157e308	1.7976931348623158e308

Wnioski:

Porównując otrzymane watrości z wynikami funkcji floatmax() oraz watrościami w bibliotece widać że program poprawnie oblicza maksymalne liczby odpowiednich typów float.

2 Zadanie 2

Problem:

Sprawdzić czy obliczając wyrażenie $3*(4/3-1)-1$ można wyznaczyć macheps dla każdego typu float

Rozwiązanie:

Program oblicza dane wyrażenie używając kolejno liczb w formatach Float16, Float32, Float64

Wyniki:

Typ liczbowy	Wynik wyrażenia	Macheps dla danego typu
Float16	-0.000977	0.000977
Float32	1.1920929e-7	1.1920929e-7
Float64	-2.220446049250313e-16	2.220446049250313e-16

Wnioski:
Dana formuła oblicza macheps dokładnie dla Float32, a dla Float16 oraz Float64 otrzymany macheps ma poprawną wartość, lecz przeciwny znak.

3 Zadanie 3

Problem:
Sprawdzić odstępy pomiędzy liczbami float w zakresie (1.0, 2.0), (0.5, 1.0) oraz (2.0, 4.0)

Rozwiązanie:
Program wylicza różnicę pomiędzy kolejnymi liczbami w danym zakresie. Aby to zrobić przechodzi po danym zakresie, analizując pary kolejnych liczb oraz obliczając ich różnicę. Można zauważyć, analizując te liczby funkcją bitstring(), że różnice między kolejnymi liczbami muszą być potęgami 2, gdyż dodając 1 do mantysy liczby w formacie float zwiększamy jej wartość o $2^{cecha-dugoscmantysy}$ (tyle wynosi wartość ostatniego bitu mantysy). Oznacza to że różnica między kolejnymi liczbami w tym zakresie wynosi 2 do potęgi pewnego wykładnika. Program wyznacza ten wykładnik wyliczając $log_2 roznica = wykadnik$.

Wyniki:

Zakres	Otrzymany wykładnik	Różnica
(1.0, 2.0)	-52	2^{-52}
(0.5, 1.0)	-53	2^{-53}
(2.0, 4.0)	-51	2^{-51}

Wnoiski:

Uruchamiając tę funkcję dla zakresu (1.0, 2.0) faktycznie otrzymujemy wynik -52 dla każdej pary liczb, więc odstępy w tym zakresie wynoszą 2^{-52}. Dla zakresu (0.5, 1.0) otrzymujemy wykładnik -53 więc odstępy w tym zakresie wynoszą 2^{-53}. Wynika to z faktu, że liczby w zakresie (0.5, 1.0) w reprezentacji float mają mniejszą o 1 cechę niż liczby z zakresu (1.0, 2.0). Oznacza to, że ostatni bit mantysy ma 2 razy mniejszą wartość, co przekłada się na 2 razy mniejsze odstępy między liczbami. Analogicznie dla liczb z zakresu (2.0, 4.0) cecha jest o 1 większa niż dla liczb z zakresu (1.0, 2.0), więc różnica między liczbami w tym zakresie wynosi 2^{-51}.

4 Zadanie 4

Problem:
Znaleść najmniejszą liczbę w zakresie (1.0, 2.0) która pomnożona przez swoją odwrotność nie daje wyniku 1.

Rozwiązanie:
Program przechodzi kolejno po liczbach w zakresie (1.0, 2.0), wyliczając iloczyn ich oraz ich odwrotności. Jeżeli znajdzie liczbę, dla której ta wyliczona wartość nie wynosi 1, przerywa pracę i ją wypisuje, wraz z otrzymanym wynikiem. Znaleziona liczba będzie najmniejsza, gdyż program przeszukuje zakres od 1.0 w kolejności rosnącej.

Wyniki:
Znaleziona liczba: 1.000000057228997
Iloczyn liczby i jej odwrotności: 0.9999999999999999

Wnoiski:
Istnieją liczby w zakresie (1.0, 2.0) dla których błędy spowodowane precyzją arytmetyki sprawiają, że nawet takie podstawowe operacje jak pomnożenie liczby przez jej odwrotność dają błędne wyniki.

5 Zadanie 5

Problem:
Przeanalizować różne sposoby obliczania iloczynu skalarnego dwóch wektorów.

Rozwiązanie:

Program realizuje 4 sposoby obliczania iloczynu skalarnego dwóch wektorów: 1) Od początku do końca, 2) Od końca do początku, 3) Od największego do najmniejszego, 4) Od najmniejszego do największego, Podane wektory to:
V1 = $[2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$
V2 = $[1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$
Faktyczna wartość iloczynu = $-1.00657107000000 * 10^{-11}$

Wyniki:

Dla arytmetyki Float32:

Sposób obliczeń	Otrzymany iloczyn	Błąd względny w %
Od początku do końca	-0.4999443	$4.9678e12$
Od końca do początku	-0.4543457	$4.5138e12$
Od największego do najmniejszego	-0.5	$4.9673e12$
Od najmniejszego do największego	-0.5	$4.9673e12$

Dla arytmetyki Float64:

Sposób obliczeń	Otrzymany iloczyn	Błąd względny w %
Od początku do końca	1.0251881368296672e-10	1118.4955
Od końca do początku	-1.5643308870494366e-10	1454.1187
Od największego do najmniejszego	0.0	100
Od najmniejszego do największego	0.0	100

Wnoiski:

Można zauważyć, że używając arytmetyki Float32 popełniamy ogromny błąd licząc iloczyn tych wektorów. Spowodowane jest to różnicą rzędów wyników mnożeń. Z powodu ograniczonej liczby bitów przeznaczonych na ich reprezentacje, nie pozwala dokładnie przedstawiać małych różnic w wynikach, przez co tracimy dokładność. Sytuacja poprawia się, gdy użyjemy arytmetyki Float64, lecz błąd względny nadal jest znaczący.

6 Zadanie 6

Problem:

Zbadać wyniki funkcji $f(x) = \sqrt{x^2+1} - 1$ oraz $g(x) = x^2/\sqrt{x^2+1} + 1$ dla kolejnych odwrotności potęg 8 (8^{-1}, 8^{-2}, 8^{-3}... itd).

Rozwiązanie:

Program implementuje funkcje f oraz g w arytmetyce Float64 i dla argumentów 8^{-1}, 8^{-2}, 8^{-3}, 8^{-4}, 8^{-5} wylicza ich wartości.

Wyniki:

x	f(x)	g(x)	różnica f(x) - g(x)
8^{-1}	0.0077822185373186414	0.0077822185373187065	-6.505213034913027e-17
8^{-2}	0.00012206286282867573	0.00012206286282875901	-8.328027937404281e-17
8^{-3}	1.9073468138230965e-6	1.907346813826566e-6	-3.469446951953614e-18
8^{-4}	2.9802321943606103e-8	2.9802321943606116e-8	-1.3234889800848443e-23
8^{-5}	4.656612873077393e-10	4.6566128719931904e-10	1.0842021724855044e-19

Porównajmy otrzymane wartości z faktycznymi wartościamy wyliczonymi za pomocą programu wolfram alpha (Oznaczmy je w(x)):

x	f(x)-w(x)	g(x)-w(x)
8^{-1}	-6.505213034913027e-17	0.0
8^{-2}	-8.329383190119888e-17	-1.3552527156068805e-20
8^{-3}	-3.469446951953614e-18	0.0
8^{-4}	-1.3234889800848443e-23	0.0
8^{-5}	1.0842016554976216e-19	-5.169878828456423e-26

Wnoiski:

Funkcja g(x) jest bardziej wiarygodna. Dzieje się tak przez błąd który zachodzi podczas obliczania funkcji f(x). W ostatnim kroku gdy odejmujemy 1 od wyliczonego pierwiastka odejmujemy od siebie bardzo zbliżone wartościami liczby. W arytmetyce float powoduje to sporą stratę precyzji. Natomiast w funkcji g(x), dzięki przekształceniu wzoru, unikamy tego problemu. W żadnym momencie nie odejmujemy podobnych liczb, dzięki czemu nie popełniamy tego błędu, co daje bardziej wiarygodny wynik.

7 Zadanie 7

Problem:

Zbadać dokładność przybliżenia pochonej funkcji wzorem $f'(x) \approx \frac{f(x_0+h)-f(x_0)}{h}$ w arytmetyce Float64 dla funkcji $f(x) = sinx + cos3x$ w puncie $x_0 = 1$

Rozwiązanie:

Program oblicza pochodną, wartość 1+h oraz błąd względny według przybliżającego wzoru dla kolejnych wartości $h = 2^{-n}$ gdzie $n = 1, 2, 3..., 54$

Wyniki:

n	Przybliżenie f'(x)	1+h	Błąd bezwzględny	Rząd błędu
0	2.0179892252685967	2.0	1.9010469435800585	0
10	0.12088247681106168	1.0009765625	0.003940195122523624	-3
20	0.11694612901192158	1.0000009536743164	3.847323383529555e-6	-6
30	0.11694216728210449	1.0000000009313226	1.1440643356286362e-7	-7
40	0.1168212890625	1.0000000000009095	0.00012099262603805505	-4
45	0.11328125	1.0000000000000284	0.003661031688538055	-3
50	0.0	1.0000000000000009	0.11694228168853806	-1
54	0.0	1.0	0.11694228168853806	-1

Wnoiski:

Można zauważyć, że najdokładniejsze wyniki dostajemy dla $n \approx 30$. Przy dalszym zwiększaniu n, błąd staje się coraz większy. Dzieje się tak, ponieważ musimy obliczyć wartość $f(x + h)$ dla x = 1. Dla coraz mniejszych wartości h, wartość 1+h, przez precyzję arytmetyki float, traci dokładność. Sprawia to mniejszą dokładność wartości funkcji $f(x + h)$, a co za tym idzie dokładność całego przybliżenia.