Obliczenia naukowe

Laboratorium

Lista 1

Kinga Majcher
272354

28 października 2024

1 Zadanie 1

1.1 Wyznaczanie epsilona maszynowego (*macheps*)

1.1.1 Opis problemu

Epsilonem maszynowym *macheps* (ang. machine epsilon) nazywamy najmniejszą liczbę $macheps > 0$ taką, że $fl(1.0 + macheps) > 1.0$ i $fl(1.0 + macheps) = 1 + macheps$. Innymi słowy *macheps* jest odległością 1.0 od kolejnej liczby x, $x > 1$, reprezentowanej w arytmetyce zmiennopozycyjnej (kolejnej liczby maszynowej). Problemem zadania jest napisanie programu wyznaczającego iteracyjnie wartość *macheps* dla Float16, Float32 oraz Float64.

1.1.2 Rozwiązanie

W celu obliczenia wartości *macheps* ustawiamy początkowo wartość *epsilon* na 1, a następnie wykonujemy pętle while, która dzieli *epsilon* przez 2 dopóki $1 + epsilon > 1$. Jako, że po ostatnim wykonaniu pętli warunek nie był już spełniony, to zwracamy ostatnią wartość, która tę nierówność spełniała, w tym przypadku jest to $2 \cdot epsilon$.

1.1.3 Wyniki oraz ich interpretacja

Typ	**Wyznaczona wartość**	**Wartość zwracana przez *eps(type)***	**Wartość zawarta w pliku *float.h***
Float16	0.000977	0.000977	-
Float32	1.1920929e-7	1.1920929e-7	1.1920929e-07
Float64	2.220446049250313e-16	2.220446049250313e-16	2.2204460492503131e-16

Tabela 1: Wyniki obliczania wartości *epsilon* i porównanie ze znanymi wartościami

W pliku *float.h* nie ma podanej bezpośrednio wartości dla typu *Float16 (half)*.

Wartości *macheps* wyznaczone dla Float16, Float32 oraz Float64 są zgodne z wartościami zwracanymi przez *eps(type)*. Dla Float32 oraz Float64 są one również zgodne z wartościami zawartymi w pliku nagłówkowym *float.h*.

1.1.4 Wnioski

Metoda obliczania wartosci *macheps* jest poprawna. Program poprawnie oblicza wartości *macheps* dla wszystkich testowanych typów.

1.1.5 Związek *macheps* z precyzją arytmetyki

Precyzja arytmetyki jest na wykładzie oznaczana literą ϵ, obliczana jest z wzoru $\epsilon = 0.5 \cdot \beta^{1-t}$, gdzie β to wartość podstawy arytmetyki (w tym przypadku $\beta = 2$), a t to liczba cyfr w mantysie. Po przekształceniu dochodzimy więc do wzoru $\epsilon = 2^{-t}$ jako precyzja arytmetyki o podstawie 2. Po dokonaniu obliczeń dochodzimy do wyników:

$$\epsilon = 2^{-11} = 4.8828125000 \cdot 10^{-4}, \text{ dla formatu Float16}$$

$$\epsilon = 2^{-24} = 5.9604644775390625 \cdot 10^{-8}, \text{ dla formatu Float32}$$

$$\epsilon = 2^{-53} = 1.1102230246251565404236316680908203125 \cdot 10^{-16}, \text{ dla formatu Float64}$$

Jak można łatwo zauważyć otrzymane wyniki są dwa razy mniejsze od wartości *macheps* wyliczonej w zadaniu. Jest to spowodowane tym, że precyzja arytmetyki jest maksymalną możliwą różnicą między realną wartością liczby, a jej przybliżeniem. *macheps* jest to natomiast odległość między dwoma kolejnymi liczbami. Odległość między 1, a $1 + macheps$ musi być podzielona na dwa, tak aby każda z liczb znajdujących się w tym przedziale mogła być jakoś przybliżona, stąd $macheps = 2 \cdot \epsilon$.

1.2 Wyznaczanie najmniejszej liczby maszynowej (eta)

1.2.1 Opis problemu

Liczbą maszynową $eta > 0.0$ nazywamy pierwszą liczbę maszynową po 0.0, która jest od niego różna. Problemem zadania jest napisanie programu wyznaczającego iteracyjnie wartość *eta* dla Float16, Float32 oraz Float64.

1.2.2 Rozwiązanie

W celu obliczenia wartości *eta* ustawiamy początkowo wartość *eta* na 1, a następnie wykonujemy pętle while, która dzieli *eta* przez 2 dopóki $\frac{eta}{2} > 0$. Podczas przechodzenia przez pętle program zmniejsza za każdym razem o 1 wartość zmiennej *exp*, w celu znania dokładnej wartości *eta* jako 2^{exp}. Po wyjściu z pętli program zwraca wartość *eta*.

1.2.3 Wyniki oraz ich interpretacja

Typ	Wyznaczona wartość	Wielkość wykładnika	Wartość zwracana przez *nextfloat(type(0.0))*
Float16	6.0e-8	-24	6.0e-8
Float32	1.0e-45	-126	1.0e-45
Float64	5.0e-324	-1074	5.0e-324

Tabela 2: Wyniki obliczania wartości *eta* i porównanie ze znanymi wartościami

Wartości *eta* wyznaczone dla Float16, Float32 oraz Float64 są zgodne z wartościami zwracanymi przez *nextfloat(type(0.0))*.

1.2.4 Wnioski

Metoda obliczania wartosci *eta* jest poprawna. Program poprawnie oblicza wartości *eta* dla wszystkich testowanych typów.

1.3 Wartości MIN_{sub} oraz MIN_{nor}

1.3.1 MIN_{sub}

Liczba MIN_{sub} jest to najmniejsza liczba subnormalna możliwa do reprezentowania w IEEE 754. Jako liczba subnormalna ma ona wykładnik w IEEE 754 postaci 00. . . 0. Wartość MIN_{sub} można obliczyć z wzoru $MIN_{sub} = 2^{-(t-1)} \cdot 2^{C_{min}}$, gdzie t to liczba cyfr w mantysie, a C_{min} to minimalna wartość cechy c. Po wykonaniu obliczeń otrzymujemy:

$$MIN_{sub} = 2^{-(11-1)} \cdot 2^{-14} = 2^{-24} = 5.9604644775390625 \cdot 10^{-8}, \text{ dla formatu Float16}$$

$$MIN_{sub} = 2^{-(24-1)} \cdot 2^{-126} = 2^{-149} = 1.401298464324817 \cdot 10^{-45}, \text{ dla formatu Float32}$$

$$MIN_{sub} = 2^{-(53-1)} \cdot 2^{-1022} = 2^{-1074} = 4.94065645841246 \cdot 10^{-324}, \text{ dla formatu Float64}$$

Wartości te są równe wartości *eta* dla każdego z typów.

1.3.2 MIN_{nor}

Liczba MIN_{nor} jest to najmniejsza liczba znormalizowana możliwa do reprezentacji w IEEE 754. Posiada ona wykładnik postaci 00...1. Wartość MIN_{nor} można obliczyć z wzoru: $MIN_{nor} = 2^{C_{min}}$, gdzie C_{min} to minimana wartość cechy c. Są to wartości:

$$MIN_{nor} = 2^{-126} = 1.1754943508222875079 68 \cdot 10^{-38}, \text{ dla formatu Float32}$$

$$MIN_{nor} = 2^{-1022} = 2.2250738585072013830 \cdot 10^{-308}, \text{ dla formatu Float64}$$

Funkcja *floatmin(type)* zwraca wartość najmniejszej dodatniej liczby znormalizowanej możliwej do reprezentacji w podanym typie. Są to wartości:

$$floatmin(Float32) = 1.1754944e - 38, \text{ dla formatu Float32}$$

$$floatmin(Float64) = 2.2250738585072014e - 308, \text{ dla formatu Float64}$$

Jak można więc zauważyć, wartości MIN_{nor} oraz *floatmin* są bardzo zbliżone. Są one bowiem tożsame - oznaczają najmniejszą możliwą liczbę znormalizowaną, a niewielka różnica w reprezentacji ułamkowej jest związana z porównywaniem wartości policzonej ręcznie z wartością zwracaną przez komputer.

1.4 Wyznaczanie liczby MAX

1.4.1 Opis problemu

Liczba MAX jest to największa możliwa reprezentowalna wartość w IEEE 754. Problemem zadania jest wyznaczenie tej wartości iteracyjnie.

1.4.2 Rozwiązanie

W systemie binarnym mnożenie przez 2 można interpretować jako przesuwanie przecinka w liczbie o jedno miejsce w prawo. Rozważając tę samą operację w notacji wykładniczej o podstawie 2 można zauważyć, że $x \cdot 2 = (\pm m \cdot 2^C) \cdot 2 = \pm m \cdot 2^{C+1}$. W wyniku operacji mnożenia przez 2 wartość mantysy nie ulega zmianie, a jedynie rośnie potęga przy 2. Standard IEEE 754 bazuje na notacji wykładniczej o podstawie 2, stąd pomysłem na znalezienie największej reprezentowalnej w tym systemie liczby jest znalezienie liczby, która w części ułamkowej mantysy ma $11\ldots1$, a następnie przemnażanie jej kolejno przez 2. Początkowo wartości *max* oraz *temp* ustawiam na 1.0. Pierwsza pętla *while* zmniejsza dwukrotnie wartość zmiennej *temp* dopóki spełniona jest nierówność 1 - *temp* < 1. Następnie zmniejszamy wartość zmiennej *max* o dwukrotność wartości temp - w ten sposób otrzymujemy poprzednią liczbę maszynową względem liczby 1.0. Liczba ta ma część ułamkową mantysy o wartości $11\ldots1$. W drugiej pętli natomiast przemnażamy znaleziona liczbę przez 2 do momentu, gdy nieprawdą jest, przestaje być to, że jej dwukrotność jest różna od nieskończoności. W ten sposób otrzymujemy maksymalną liczbę reprezentowalną w IEEE 754 dla danego typu.

1.4.3 Wyniki oraz ich interpretacja

Typ	**Wyznaczona wartość**	**Wartość zwracana przez *floatmax(type)***	**Wartość zawarta w pliku *float.h***
Float16	6.55e4	6.55e4	-
Float32	3.4028235e38	3.4028235e38	3.40282347e+38
Float64	1.7976931348623157e308	1.7976931348623157e308	1.7976931348623157e+308

Tabela 3: Wyniki obliczania wartości MAX i porównanie ze znanymi wartościami

W pliku *float.h* nie ma podanej bezpośrednio wartości dla typu *Float16 (half)*.

Wartości MAX wyznaczone dla Float16, Float32 oraz Float64 są zgodne z wartościami zwracanymi przez *floatmax(type)*. Dla Float32 oraz Float64 są one również zgodne z wartościami zawartymi w pliku nagłówkowym *float.h*.

1.4.4 Wnioski

Metoda obliczania wartosci MAX jest poprawna. Program poprawnie oblicza wartości MAX dla wszystkich testowanych typów.

2 Zadanie 2

2.1 Opis problemu

Według W. Kahana wartość epsilonu maszynowego (*macheps*) można uzyskać wykonując działanie $3 \cdot (\frac{4}{3} - 1) - 1$. Problemem zadania jest sprawdzenie słuszności tego stwierdzenia dla różnych typów zmiennopozycyjnych.

2.2 Rozwiązanie

W celu obliczenia wartości wyrażenia tworzymy funkcję, która przyjmuje typ jako parametr, a następnie wykonujemy działanie $3 \cdot (\frac{4}{3} - 1) - 1$ pamiętając o tym, że każda z liczb ma mieć ustalony typ. Następnie porównujemy uzyskane wartości z wartościami uzyskiwanymi przez wywołanie wbudowanej funkcji *eps(type)*.

2.3 Wyniki oraz ich interpretacja

Typ	**Wartość wyznaczona metodą Kahana**	**Wartość zwracana przez *eps(type)***
Float16	-0.000977	0.000977
Float32	1.1920929e-7	1.1920929e-7
Float64	-2.220446049250313e-16	2.220446049250313e-16

Tabela 4: Wyniki obliczania wartości *macheps* według algorytmu Kahana i porównanie ze znanymi wartościami zwracanymi przez funkcje *eps(type)*.

Wartość wyznaczona metodą Kahana jest zgodna z realną wartością uzyskaną przez *eps(type)* tylko dla Float32. Dla Float16 oraz Float64 wartości te są zgodne co do modułu. Jest ona spowodowana różnicami w zaokrągleniu wartości liczby $\frac{4}{3}$ dla różnych typów. Poniżej przedstawiam zapis bitowy po kolejnych operacjach w poszczególnych typach:

Działanie	**Zapis bitowy**
$\frac{4}{3}$	0 01111 0101010101
$\frac{4}{3} - 1$	0 01101 0101010100
$3 \cdot (\frac{4}{3} - 1)$	0 01110 1111111110
$3 \cdot (\frac{4}{3} - 1) - 1$	1 00101 0000000000

Tabela 5: Zapisy bitowe po poszczególnych działaniach dla Float16

Działanie	**Zapis bitowy**
$\frac{4}{3}$	0 01111111 01010101010101010101011
$\frac{4}{3} - 1$	0 01111101 01010101010101010101100
$3 \cdot (\frac{4}{3} - 1)$	0 01111111 00000000000000000000001
$3 \cdot (\frac{4}{3} - 1) - 1$	0 01101000 00000000000000000000000

Tabela 6: Zapisy bitowe po poszczególnych działaniach dla Float32

Działanie	**Zapis bitowy**
$\frac{4}{3}$	0 01111111111 0101010101010101010101010101010101010101010101010101
$\frac{4}{3} - 1$	0 01111111101 0101010101010101010101010101010101010101010101010100
$3 \cdot (\frac{4}{3} - 1)$	0 01111111110 1111111111111111111111111111111111111111111111111110
$3 \cdot (\frac{4}{3} - 1) - 1$	1 01111001011 0000000000000000000000000000000000000000000000000000

Tabela 7: Zapisy bitowe po poszczególnych działaniach dla Float64

Jak można zauważyć różnice w zaokrągleniu $\frac{4}{3}$ (0101 . . . 0101 dla Float16 oraz Float64 i 0101 . . . 1011 dla Float32) mają znaczący wpływ na wyniki kolejnych działań co w efekcie prowadzi do zmiany znaku przy końcowym wyniku.

2.4 Wnioski

Metoda Kahana może być wykorzystywana do obliczania wartości *macheps* tylko jeśli bierzemy pod uwagę moduł wyników jakie zwraca. Pod względem samej poprawności obliczeń jest ona problematyczna bo wynik jest zależny od tego jakie przybliżenie zostanie zastosowane w przypadku danej zmiennej - czy w górę, czy w dół.

3 Zadanie 3

3.1 Opis problemu

Problemem zadania jest sprawdzenie eksperymentalnie, że w arytmetyce Float64 w IEEE 754 liczby zmiennopozycyjne są równomiernie rozmieszczone w $[1, 2]$ z krokiem $\delta = 2^{-52}$, czyli, że każda liczba x w tym przedziale może być przedstawiona jako $x = 1 + k\delta$, gdzie $k = 1, 2, \ldots, 2^{52} - 1$ i $\delta = 2^{-52}$. Ponadto problemem jest sprawdzenie jak liczby są rozmieszczone w przedziale $[\frac{1}{2}, 1]$ oraz $[2, 4]$.

3.2 Rozwiązanie

Rozwiązanie polega na przyjrzeniu się zapisowi bitowemu, wyświetlanemu za pomocą funkcji *bitstring* oraz zauważeniu zależności na temat tego, jak działa mnożenie przez 2 w IEEE 754. Weźmy przykład:

$$100.000001 \cdot 2 = 1.00000001 \cdot 2^2 \cdot 2 = 1.00000001 \cdot 2^3 = 1000.00001$$

W wyniku operacji mnożenia przez 2 wartość mantysy nie ulega zmianie, a jedynie rośnie potęga przy 2. Można zauważyć, że poprzez operacje mnożenia długość części ułamkowej liczby zmalała o 1, podczas gdy jej długość w zapisie notacji wykładniczej pozostała taka sama. W standardzie IEEE 754 dla dowolnego typu liczba dostępnych cyfr w części ułamkowej jest ściśle określona. Wartość $\delta = 2^{-52}$ jest wartością domyślną dla przedziału $[1, 2]$, jest to spowodowane tym, ze dowolną liczbę z przedziału $[1, 2)$ można zapisać w postaci notacji wykładniczej z potęgą przy 2 o wartości 0. Przez to w części ułamkowej zapisu IEEE 754 dla Float64 można zapisać 2^{52} różnych liczb znajdujących się w przedziale $[1, 2)$, stąd odległość miedzy dwoma kolejnymi musi wynosić 2^{-52}. W wyniku przemnażania przez 2 liczba liczb możliwych do zapisania rośnie (np. dla przedziału $[2, 4)$ są już to liczby w postaci 2. ... i 3. ..., podczas gdy dla przedziału $[1, 2)$ są to tylko liczby postaci 1. ...), naturalnym jest więc, że jeśli liczba pozycji, na których liczbę można zapisać jest stała to cierpiała będzie na tym dokładność. Podobnie jest z dzieleniem przez 2, wtedy dokładność będzie rosła. Przez to udało mi się zauważyć, że dla każdego przedziału postaci $[2^t, 2^{t+1}]$, gdzie $t \in$ Z, wartość δ wynosi $\delta = 2^{-52+t}$.
Dla zachowania ogólności program został napisany dla dowolnego przedziału $[2^t, 2^{t+1}]$, gdzie $t \in$ Z. Funkcja *ieee_with_spaces(number)* jest funkcją pomocniczą, która wyświetla zapis bitowy podanej liczby zwrócony przez *bitstring* podzielony na części znaku, wykładnika oraz mantysy w celu łatwiejszej analizy bitów. Kolejna funkcja jako parametr przyjmuje wartości lewej *(left)* i prawej *(right)* granicy przedziału. Następnie liczy deltę poprzez dzielenie wartości *calculated_delta* początkowo ustawionej na 1.0 kolejno przez 2, dopóki spełniona jest nierówność $left + \frac{calculated\_delta}{2} > left$. W kolejnej pętli liczona jest wartość delty poprzez skorzystanie z wcześniej wyprowadzonej zależności. Funkcja drukuje obie wartości wyznaczonej delty w celu porównania ich, a następnie drukuje wartości kolejnej liczby maszynowej po *left*, czyli $left + delta$ oraz dla porównania wartość kolejnej liczby maszynowej otrzymanej przez $nextfloat(left)$, a także wartość poprzedniej liczby maszynowej przed *right*, czyli $left + (2^{52} - 1) \cdot delta$ i dla porównania wartość zwracaną przez $prevfloat(right)$. Jeśli obliczona delta jest poprawna liczby w obu parach powinny mieć tę samą reprezentację bitową.

3.3 Wyniki oraz ich interpretacja

Liczba	**Zapis bitowy**
1.0	0 01111111111 0000000000000000000000000000000000000000000000000000
$1.0 + delta$	0 01111111111 0000000000000000000000000000000000000000000000000001
$nextfloat(1.0)$	0 01111111111 0000000000000000000000000000000000000000000000000001
$1.0 + (2^{52} - 1) \cdot delta$	0 01111111111 1111111111111111111111111111111111111111111111111111
$prevfloat(2.0)$	0 01111111111 1111111111111111111111111111111111111111111111111111
2.0	0 10000000000 0000000000000000000000000000000000000000000000000000

Tabela 8: Zapisy bitowe dla przedziału $[1, 2]$

Liczba	Zapis bitowy
0.5	0 01111111110 0000000000000000000000000000000000000000000000000000
$0.5 + delta$	0 01111111110 0000000000000000000000000000000000000000000000000001
$nextfloat(0.5)$	0 01111111110 0000000000000000000000000000000000000000000000000001
$0.5 + (2^{52} - 1) \cdot delta$	0 01111111110 1111111111111111111111111111111111111111111111111111
$prevfloat(1.0)$	0 01111111110 1111111111111111111111111111111111111111111111111111
1.0	0 01111111111 0000000000000000000000000000000000000000000000000000

Tabela 9: Zapisy bitowe dla przedziału $[0.5, 1]$

Liczba	Zapis bitowy
2.0	0 10000000000 0000000000000000000000000000000000000000000000000000
$2.0 + delta$	0 10000000000 0000000000000000000000000000000000000000000000000001
$nextfloat(2.0)$	0 10000000000 0000000000000000000000000000000000000000000000000001
$2.0 + (2^{52} - 1) \cdot delta$	0 10000000000 1111111111111111111111111111111111111111111111111111
$prevfloat(4.0)$	0 10000000000 1111111111111111111111111111111111111111111111111111
4.0	0 10000000001 0000000000000000000000000000000000000000000000000000

Tabela 10: Zapisy bitowe dla przedziału $[2, 4]$

Dla każdego z rozpatrywanych przedziałów obliczone wartości $left+delta$ oraz $left+(2^{52}-1)\cdot delta$ są poprawne.

3.4 Wnioski

Dla dowolnego przedziału $[2^t, 2^{t+1}]$, gdzie $t \in$ Z wartość delty można obliczyć z wzoru $\delta = 2^{-52+t}$, a każdą liczbę wewnątrz tego przedziału można zapisać jako $2^t + k \cdot \delta$, gdzie $k \in 1, 2, \ldots, 2^{52} - 1$. Wraz z wzrostem wykładnika wartości części ułamkowych liczb stają się coraz mniej dokładne.

4 Zadanie 4

4.1 Opis problemu

Problemem zadania jest znalezienie w arytmetyce Float64 w sposób iteracyjny najmniejszej liczby x z przedziału $1 < x < 2$, takiej że $x \cdot \frac{1}{x} \neq 1$.

4.2 Rozwiązanie

Z poprzedniego zadania wiemy, że w przedziale $[1, 2]$ liczby są rozmieszczone równomiernie z krokiem $\delta = 2^{-52}$. W celu znalezienia takiej wartości x wykonujemy pętle while, która zaczynając od $x = 1.0$ za każdym razem sprawdza, czy $x \cdot \frac{1}{x} \neq 1$. Jeśli tak to funkcja zwraca wartość x. W przeciwnym wypadku zwiększa wartość x o δ. Jeśli program nie znajdzie liczby spełniającej warunek, zwraca wartość -1.

4.3 Wyniki oraz ich interpretacja

Najmniejszą liczbą x w przedziale $(1, 2)$, która spełnia $x \cdot \frac{1}{x} \neq 1$ jest 1.000000057228997. Wartość $x \cdot \frac{1}{x}$ dla tego x wynosi 0.9999999999999999.

4.4 Wnioski

Przez to, że w arytmetyce IEEE 754 niemożliwe jest przedstawienie każdej dowolnej liczby rzeczywistej to liczby ulegają zaokrąglaniu. Dzielenie nie jest operacją odwracalną w arytmetyce zmiennopozycyjnej. Wszystkie działania obarczone są pewnym błędem, o którym należy pamiętać dokonując obliczeń.

5 Zadanie 5

5.1 Opis problemu

Problemem zadania jest napisanie programu dokonującego obliczeń iloczynu skalarnego wektorów x oraz y na 4 różne sposoby:

1. obliczenie kolejnych iloczynów i sumowanie ich po kolei
2. obliczenie kolejnych iloczynów i sumowanie ich od końca
3. obliczenie kolejnych iloczynów, podzielenie ich na iloczyny dodatnie i ujemne, posortowanie obu grup malejąco względem modułu, policzenie osobnych podsum dla iloczynów dodatnich i ujemnych i na końcu dodanie podsum
4. obliczenie kolejnych iloczynów, podzielenie ich na iloczyny dodatnie i ujemne, posortowanie obu grup rosnąco względem modułu, policzenie osobnych podsum dla iloczynów dodatnich i ujemnych i na końcu dodanie podsum

Obliczeń dokonujemy dla wektorów:

$$x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$$

$$y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$$

Prawidłowa wartość sumy wynosi $-1.00657107000000 \cdot 10^{-11}$.

5.2 Rozwiązanie

5.2.1 Metoda 1

Inicjujemy wartość zmiennej *sum* na 0.0. Następnie w pętli *for i in 1:length(x)* zwiększamy wartość sumy o iloczyn $x[i] \cdot y[i]$ dla kolejnych i.

5.2.2 Metoda 2

Inicjujemy wartość zmiennej *sum* na 0.0. Następnie w pętli *for i = length(x):-1:1* zwiększamy wartość sumy o iloczyn $x[i] \cdot y[i]$ dla kolejnych i.

5.2.3 Metoda 3

Inicjujemy wartości zmiennych *positive_sum* oraz *negative_sum* na 0.0. Tworzymy tablicę, w której obliczamy poszczególne iloczyny $x[i] \cdot y[i]$ dla każdego i. Tworzymy tablicę *positive_products*, w której przechowujemy tylko iloczyny dodatnie posortowane malejąco oraz tablicę *negative_products*, w której przechowujemy tylko iloczyny ujemne posortowane rosnąco. Następnie w pętli zwiększamy *positive_sum* o kolejne wartości dodatnich iloczynów, a następnie w drugiej pętli zwiększamy *negative_sum* o kolejne wartości ujemnych iloczynów. Na samym końcu wartość *sum* ustawiamy na sumę *positive_sum* oraz *negative_sum*.

5.2.4 Metoda 4

Inicjujemy wartości zmiennych *positive_sum* oraz *negative_sum* na 0.0. Tworzymy tablicę, w której obliczamy poszczególne iloczyny $x[i] \cdot y[i]$ dla każdego i. Tworzymy tablicę *positive_products*, w której przechowujemy tylko iloczyny dodatnie posortowane rosnąco oraz tablicę *negative_products*, w której przechowujemy tylko iloczyny ujemne posortowane malejąco. Następnie w pętli zwiększamy *positive_sum* o kolejne wartości dodatnich iloczynów, a następnie w drugiej pętli zwiększamy *negative_sum* o kolejne wartości ujemnych iloczynów. Na samym końcu wartość *sum* ustawiamy na sumę *positive_sum* oraz *negative_sum*.

5.3 Wyniki oraz ich interpretacja

Typ	**Wartość dla metody 1**	**Wartość dla metody 2**	**Wartość dla metody 3**	**Wartość dla metody 4**
Float32	-0.4999443	-0.4543457	-0.5	-0.5
Float64	1.0251881368296672e-10	-1.5643308870494366e-10	0.0	0.0

Tabela 11: Wyniki obliczania wartości iloczynu skalarnego wektorów x i y dla czterech różnych metod i dla typów Float32 i Float64

Typ	Wartość błędu względnego dla metody 1	Wartość błędu względnego dla metody 2	Wartość błędu względnego dla metody 3	Wartość błędu względnego dla metody 4
Float32	-4.9668057e12%	-4.5137967e12%	-4.9673593e12%	-4.9673593e12%
Float64	-1118.4955313981627%	-1454.1186645165915%	-100.0%	-100.0%

Tabela 12: Wartości błędów względnych dla wszystkich metod i typów obliczone z wzoru $\frac{|s-\tilde{s}|}{s} \cdot 100\%$, gdzie s to rzeczywista wartość iloczynu, a $\tilde{s}$ to obliczona daną metodą wartość iloczynu skalarnego

Wyniki obliczeń dla każdej z metod znacząco odbiegają od realnej wartości iloczynu skalarnego. Wartości błędu względnego dla prawie każdej z metod przekraczają znacząco 100%.

5.4 Wnioski

Żadna z metod obliczania iloczynu skalarnego nie przyniosła wyniku zbliżonego do prawdziwego, choć patrząc na wartości błędów względnych można zauważyć, że dla Float64 wartości te są nieco lepsze, jednak nadal niewystarczająco dokładne. Błędy te są spowodowane tym, że w arytmetyce zmiennoprzecinkowej, aby dodać dwie liczby musimy wyrównać ich cechy. W przypadku wektorów x oraz y różnice w wielkości dodawanych wartości są bardzo duże, przez co po wyrównaniu ich cech precyzja tej mniejszej znacząco spada. Tak więc na wartość otrzymanego wyniku wpływają zarówno wartości na jakich wykonujemy działania jak i kolejność z jaką to robimy.

6 Zadanie 6

6.1 Opis problemu

Problemem zadania jest policzenie wartości funkcji:

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

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

dla kolejnych wartości argumentu $x = 8^{-1}, 8^{-2}, \ldots$ w artymetyce Float64 i wyjaśnienie dlaczego komputer daje różne wyniki, choć funkcje są tożsame. Ponadto należy stwierdzić, która z funkcji daje bardziej wiarygodne wyniki.

6.2 Rozwiązanie

Zaimplementujemy dwie funkcje - $f(x)$ i $g(x)$, które dla podanego x zwracają wartość funkcji zgodnie z odpowiadającymi wzorami. Następnie w pętli sprawdzamy jak zachowują się wartości obu funkcji.

6.3 Wyniki oraz ich interpretacja

Wartość x	**Wartość zwracana przez $f(x)$**	**Wartość zwracana przez $g(x)$**
8^{-1}	0.0077822185373186414	0.0077822185373187065
8^{-2}	0.00012206286282867573	0.00012206286282875901
8^{-3}	1.9073468138230965e-6	1.907346813826566e-6
8^{-4}	2.9802321943606103e-8	2.9802321943606116e-8
8^{-5}	4.656612873077393e-10	4.6566128719931904e-10
8^{-6}	7.275957614183426e-12	7.275957614156956e-12
8^{-7}	1.1368683772161603e-13	1.1368683772160957e-13
8^{-8}	1.7763568394002505e-15	1.7763568394002489e-15
8^{-9}	0.0	2.7755575615628914e-17
...	...	...
8^{-178}	0.0	1.6e-322
8^{-179}	0.0	0.0

Tabela 13: Wyniki funkcji $f(x)$ i $g(x)$

Jak można zauważyć dla początkowych wartości x obie funkcje zwracają podobne wielkości, później wartość $f(x)$ zaczyna nieco odbiegać od wartości $g(x)$, aż w końcu całkowicie się zeruje dla $x = 8^{-9}$ i mniejszych wartości x. Funkcja $g(x)$ zeruje się dopiero dla $x = 8^{-179}$.

6.4 Wnioski

Dla $x \to 0$ $\sqrt{x^2+1} \to 1$. W przypadku funkcji $f(x)$ skutkuje to tym, że dość szybko od wartości nieznacznie większej od 1 odejmujemy 1. Takie odejmowanie bliskich sobie liczb powoduje spadek precyzji arytmetyki, przez co w konsekwencji już dla $x = 8^{-9}$ funkcja zwraca wartość 0.0. W przypadku funkcji $g(x)$ unikamy tego odejmowania przez co wyniki przez nią zwracane są bardziej bliskie prawdy, choć nadal przez ograniczenia arytmetyki nie są równe realnym wartościom. Jako, że $\sqrt{x^2+1} \to 1$ dla $x \to 0$ to wartość funkcji $g(x)$ od pewnego momentu wynosi $\frac{x^2}{2}$, jest to mimo wszystko i tak dużo lepsze oszacowanie realnej wartości niż to zwracane przez $f(x)$. Jak więc można zauważyć, coś co jest matematyczną tożsamością nie zawsze będzie nią w arytmetyce zmiennopozycyjnej. Czasem warto przekształcić funkcję, której wartości szukamy do równoważnej postaci, aby otrzymać bardziej wiarygodne wyniki.

7 Zadanie 7

7.1 Opis problemu

Przybliżoną wartość pochodnej $f(x)$ w danym punkcie x można obliczyć z wzoru

$$f'(x_0) \approx \tilde{f}'(x_0) = \frac{f(x_0+h) - f(x_0)}{h}, \text{ gdzie } h \to 0$$

Korzystając z tego wzoru należy obliczyć w arytmetyce Float64 przybliżoną wartość pochodnej dla funkcji $f(x) = \sin x + \cos 3x$ w punkcie $x_0 = 1$ oraz błędów $|f'(x_0) - \tilde{f}'(x_0)|$ dla $h = 2^{-n}$, gdzie $n \in 0, 1, 2, \ldots 54$.

7.2 Rozwiązanie

Zaimplementujemy dwie funkcje. Funkcja f jako parametr przyjmuje x i zwraca jego wartość. Funkcja *derivative* przyjmuje dwa parametry - wartość x i obecną wartość h. Następnie w pętli obliczamy wartość pochodnej dla danego h oraz błąd bezwzględny. Dokładna wartość pochodnej została policzona poza programem i wynosi $f'(x) = 0.11694228168853815$.

7.3 Wyniki oraz ich interpretacja

h	$h+1$	$\tilde{f}(x_0)$	$\lvert f'(x_0) - \tilde{f}'(x_0)\rvert$
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^{-27}	1.0000000074505806	0.11694231629371643	3.460517827846843e-8
2^{-28}	1.0000000037252903	0.11694228649139404	4.802855890773117e-9
2^{-29}	1.0000000018626451	0.11694222688674927	5.480178888461751e-8
...	...	...	...
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

Tabela 14: Wyniki obliczania wartości pochodnej i błędy bezwzględne

Przybliżenie staje się coraz lepsze wraz z zmniejszaniem się h do pewnego momentu. Najlepsze jest ono dla $h = 2^{-28}$.

7.4 Wnioski

Z matematycznego punktu widzenia wraz z zmniejszaniem się wartości h przybliżenie wartości pochodnej powinno stawać się coraz bardziej precyzyjne. Tak jednak nie jest w tym przypadku, gdzie otrzymujemy je dla $h = 2^{-28}$. Jest to spowodowane oczywiście błędami zaokrągleń. Gdy h staje się bardzo małe to obliczamy różnicę $f(x_0 + h) - f(x_0)$ dla bardzo podobnych wartości. Operacja ta prowadzi do znacznego zmniejszenia dokładności wyniku, co w konsekwencji sprawia, że wartość pochodnej zamiast być dokładniejsza, to od realnej wartości odbiega.