Obliczenia naukowe Lista 1

Radosław Wojtczak [1]

23.10.2021

[1]Numer indeksu: 254607

Spis treści

Zadanie 1

1.1 Krótki opis problemu

Główną problematyką zadania pierwszego było wyznaczenie w zadanych standardach reperezentacji liczb zmiennoprzeciwnkowych:

1. Epsilon maszynowego (ang. machine epsilon, zwanego również jako *macheps*), czyli najmniejszej takiej liczby, że $macheps > 0$ oraz

$$1.0 + macheps > 1.0 \tag{1.1}$$

   czyli najmniejszej takiej liczby, która dodana do 1 ciągle zmienia jej wartość

2. Liczby maszynowej *eta*, takiej że $eta > 0.0$, czyli najmniejszej liczby, która w danym standardzie jest ciągle różna od 0

3. Liczbę MAX, czyli największą liczbę zmiennoprzecinkową w danym standarcie

Rozpatrywane standardy w tym zadaniu to:

1. Half, czyli Float16

2. Single, czyli Float32

3. Double, czyli Float64

1.2 Rozwiazanie

Rozwiązanie zadania znajduje się w pliku zad1.jl. Składa się z 7 funkcji:

- main, która odpowiedzialna jest za poprawną kolejność wykonywanych operacji

- my_eps(), która inicjalizuje zmienną *macheps* na 1.0 w zadanym standarcie, następnie generuje wartości epsilon maszynowego w sposób iteracyjny przy pomocy pętli while, która powtarza się aż do momentu, gdy $1 + \frac{macheps}{2} = 1$ (czyli do momentu, gdy liczba jest na tyle mała, że jej dodanie do jedynki nic nie zmienia). W ów pętli dochodzi do dekrementacji wartości epsilon maszynowego o połowę ($\frac{macheps}{2}$). Gdy pętla ukończy swoje działanie, otrzymujemy odpowiednią wartość *macheps*, którą wypisujemy na standardowy strumień wyjścią funkcją println().

- built_in_eps(), która wypisuje wyniki otrzymane przy skorzystaniu z wbudowanej funkcji w Julii o nazwie eps() (zgodnie z poleceniem zadania)
- my_eta(), która inicjalizuje zmienną *eta* na 1.0 w zadanym standarcie, następnie w pętli dochodzi do zmniejszenia tej wartości o połowę ($\frac{eta}{2}$) aż do momentu, gdy $\frac{eta}{2} = 0.0$ (w podanym standarcie). Po zakończeniu pracy pętli wypisujemy wynik na standardowym strumień wyjścia przy pomocy wbudowanej funkcji println()
- built_in_eta(), która wypisuje wyniki otrzymane przy skorzystaniu z wbudowanej funkcji w Julii o nazwie nextfloat (zgodnie z poleceniem zadania)
- my_max(), która ma za zadanie wyznaczenie największej liczby zmiennoprzeciwnkowej dla podanego standardu. Funkcja ów rozpoczyna od inicjalizacji zmiennej *max* jedynką (dla podanego standardu) i zwiększa ją dwukrotnie (wykonuje operacje $max = max * 2$) aż do momentu, gdy $max * 2 = \infty$. W tej sytuacji zauważamy, że nie możemy dalej postępować zaplanowaną strategia. Z tego powodu kreujejmy nową zmienną- *underflow*, która jest równa połowie maxa. Następnie przeprowadzamy dodawanie w pętli do momentu, aż nie otrzymamy nieskończoności lub zmienna *underflow* będzie nierozróżnialna od jedynki. Gdy pętla ukończy swoje działanie, otrzymujemy odpowiednią wartość *max*, którą wypisujemy na standardowy strumień wyjścią funkcją println().
- built_in_max(), która wypisuje wyniki otrzymane przy skorzystaniu z wbudowanej funkcji w Julii o nazwie floatmax (zgodnie z polecenim zadania)

Dodatkowym plikiem w tym zadaniu jest plik o nazwie test.c, którego zadaniem jest wypisanie wartości epsilon maszynowego oraz maksymalnej liczby zmiennoprzecinkowej w języku C korzystając ze stałych zawartych w pliku nagłówkowym "float.h"

1.3 Wyjaśnienie poprawności funkcji my_max()

Przeformułujmy lekko to zadanie w celu łatwiejszego przedstawienia rozumowania- Niech wartością startową będzie 1, a naszym zadaniem będzie znalezienie wartości, która jest najbliżej liczby 4. Za pierwszym razem zwiększamy wartość liczby 1 dwukrotnie otrzymując liczbę 2(za to odpowiedzialna jest pierwsza pętla while w funkcji my_max()), natomiast kolejne wykonanie takiej operacji poskutkuje otrzymaniem liczby 4, dlatego musimy zmienić naszą strategię. Inicjujemy nową zmienną *underflow*, która będzie równa połowie aktualne maxa (w tym przypadku połowa z 2 to 1). W pętli iteracyjnie dodajemy *underflow* do *max* przy okazji zmniejszając wartośc *underflow* dwukrotnie. Dla wyżej zaprezentowanego przykładu zmienna *underflow* będzie otrzymywała następujące wartości: $\{1, 0.5, 0.25, 0.125, ...\}$ natomiast zmienna *max* $\{2, 3, 3.5, 3.75, ...\}$. Zauważamy, że granica takiego ciągu dąży do liczby 4. Ze względu na skończoną reprezentację liczb zmiennoprzecinkowych w komputerze w pewnym momencie zbliżymy się na tyle

do liczby 4, że będzie ona dla komputera nierożróżnialna od samej czwórki. Rozwiązaniem, będzie liczba, którą komputer rozróżni ostatnią przed liczbą 4. Powyższe rozumowanie rozszerza się do przypadku z polecenia zadania.

1.4 Wyniki oraz ich interpetacje

Prezentacja otrzymanych wyników w tabelach:

Typ zmiennoprzecinkowy	my_eps()	eps()	float.h
Float 16	0.000977	0.000977	BRAK
Float 32	1.1920929e-7	1.1920929e-7	1.1920928955e-07
Float 64	2.220446049250313e-16	2.220446049250313e-16	2.2204460493e-16

Table 1.1: Wyniki dla epsilon maszynowego

Typ zmiennoprzeciwnkowy	my_eta()	nextfloat(0.0)
Float16	6.0e-8	6.0e-8
Float32	1.0e-45	1.0e-45
Float64	5.0e-324	5.0e-324

Table 1.2: Wyniki dla ety

Typ zmiennoprzeciwnkowy	my_max()	floatmax()	float.h
Float16	6.55e4	6.55e4	BRAK
Float32	3.4028235e38	3.4028235e38	3.4028234664e+38
Float64	1.7976931348623157e308	1.7976931348623157e308	1.7976931349e+308

Table 1.3: Wyniki dla max

Interpetacja: Wyniki otrzymane przez moje funkcje jak i przez funkcje wbudowane są sobie równe, co oznacza, iż zostały one zaimplementowane w sposób poprawny. Drobna różnica występująca w kolumnach zawierających informację na temat stałych w bibliotece float.h wynika z zaokrąglenia.

1.5 Wnioski

Liczby przedstawione w komputerze tworzą siatkę, przez co ich liczba jest skończona. Skończoność dotyczy nie tylko nieskończoności, ale także liczb występujących w dowolnych przedziałach. Naturalnie zauważamy, iż im więcej bitów poświęcimy na reprezentację liczby zmiennoprzecinkowej otrzymujemy dokładniejsze reprezentacje (bardziej zbliżone do siebie) oraz większe wartości maksymalne.

1.6 Odpowiedzi na pytania z treści zadania

1. Jaki związek ma liczba *macheps* z precyzją arytmetyki?

2. Jaki związek ma liczba *eta* z liczbą MIN_{sub}?

1. Epsilon maszynowy z definicji jest wartością określającą precyzję obliczeń numerycznych. Dodatkowo na wykładzie precyzję arytmetki (oznaczaną jako ε) wykorzystywaliśmy w równaniu:

$$rd(x) = x(1+\delta), |\delta| \leq \varepsilon \tag{1.2}$$

które jest bliźniaczo podobne do równania, które wykorzystywaliśmy w celu obliczenia wartości macheps 1.1. Oznacza to, że te dwie wartości są tożsame. Gdyby tak nie było otrzymywane wyniki obliczeń były fałszowane. W hipotetycznej sytuacji, w której epsilon masynowy byłby większy niż precyzja arytmetyki nasze obliczenia nie byłby dokładne ze względu na zbyt ubogą reprezentację liczb, natomiast w drugą strone- gdyby precyzja arytmetyki była większa niż epsilon maszynowy, nie bylibyśmy w stanie wykorzystać części przechowywanych liczb ze względu na to, iż nigdy nie otrzymalibyśmy ich w obliczanych równaniach.

2.

Typ zmiennoprzeciwnkowy	MIN_{sub}	MIN_{nor}
Float32	$1.4 * 10^{-45}$	$1.2 * 10^{-38}$
Float64	$4.9 * 10^{-324}$	$2.2 * 10^{-308}$

Table 1.4: Wyniki dla max

Z wykładu wiemy, iż wartości MIN_{sub} to $1.4 * 10^{-45}$ (Float32) oraz $4.9 * 10^{-324}$ co pokrywa się w wynikami otrzymanymi prez powyższe funkcje (jedyna różnica występuja zaokrągleniach). To co należy zauważyć to fakt, iż liczby zwracane przez funkcje my_eta() oraz nextfloat() są liczbami zdenormalizowanymi. Dla porównania funkcja floatmin() zwraca wartości znormalizowane, która dla Float32 jest równa $1.2 * 10^{-38}$ a dla Float64: $2.2 * 10^{-308}$, które są bardziej oddalone od zera niż wartości zdenormalizowane. Przeprowadzając dokładne obliczenia na bardzo małych liczbach należy pamiętać o istnieniu oby reprezentacji liczb.

Zadanie 2

2.1 Krótki opis problemu

Celem zadania 2 była weryfikacja metody Kahana odnośnie wyznaczania epsilon maszynowego. Zaproponowany przez niego sposób polegał na obliczeniu wyrażenia:

$$3 * (\frac{4}{3} - 1) - 1 \tag{2.1}$$

W arytmetyce zmiennopozycyjnej

2.2 Rozwiazanie

Plik zad2.jl zawiera krótką implementację wzoru 2.1 znajdującą się w funkcji KahanMethod(). Dodatkowo dla formalności została umieszczona funkcja Verify(), która funkcjonuje identycznie jak funkcja my_eps() z poprzedniego zadania.

2.3 Wyniki oraz ich interpetacje

Prezentacja otrzymanych wyników w tabelach:

Typ zmiennoprzecinkowy	KahanMethod()	Verify()
Float 16	-0.000977	0.000977
Float 32	1.1920929e-7	1.1920929e-7
Float 64	-2.220446049250313e-16	2.220446049250313e-16

Table 2.1: Porównanie funkcji eps() i metody Kahana 2.1

Interpetacja: Łatwo zauważyć, że powyższe wyniki są równe co do wartości bezwzględnej. Zmiana znaku może występować, ze względu na to, iż ułamek $\frac{4}{3}$ jest ułamkiem okresowym, którego okresować w zależności od liczby bitów przeznaczonych na liczbę, może wpływać na zmianę znaku.

2.4 Wnioski

Ze względu na skończoną reprezentację liczb w komputerze występują swego rodzaju paradoksy, gdzie mimo odejmowania, którego wynik powinien być do-

datni, otrzymujemy wynik ze znakiem przeciwnym do tego, co podpowiada nam matematyka.

Zadanie 3

3.1 Krótki opis problemu

Celem tego zadania było sprawdzenie gęstości rozłożenia liczb zmiennoprzecinkowych w odpowiednich przedziałach.

3.2 Rozwiazanie

W pliku zad3.jl w funkcji *zad3()* zostało zaimplementowane rozwiązanie mające na celu dla podanych argumentów *start* i *stop* (odpowiednio- dla początku i końca przedziału) obliczyć odległość między następnymi liczbami w zadanym przedziale. Rozwiązanie to opiera się o fakt, iż przy stałej eksponencie to, ile liczb jesteśmy w stanie przedstawić, zależy tylko i wyłącznie od liczby bitów jaka została przeznaczona na mantysę (która różni się w zależności od typu reprezentacji liczb zmiennoprzecinkowych). Oznacza to, iż gęstość rozłożenia liczb w przedziale [1,2] jest większa niż w przedziale [2,4]. To absurdalne na pierwszy rzut oka stwierdzenie zaczyna nabierać sensu gdy zdamy sobie sprawę, że poza ostatnimi cyframi wyżej wymienionych przedziałów (które możemy wyeliminować z rozpatrywania, gdyż odległość między liczbami jest stała) każda liczba, która się w nich zawiera, ma taką samą eksponantę. Łatwiej to zauważyć zapisując te przedziały w następujący sposób: $[2^0, 2^1]$ i $[2^1, 2^2]$. Dla całego pierwszego przedziału eksponanta jest równa 2^0, natomiast dla drugiego- 2^1. Zauważamy więc, że mamy tyle samo miejsca na przedstawienie liczb z przedziału [1,2] jak i [2,4], co automatycznie musi wpłynąć na różnicę gęstości rozłożenia. Korzystając ze wzoru (b- tzw. frakcja dla podanego typu, e- expontent bias:

$$(-1)^{znak}\left(1+\sum_{i=1}^{52} b_{52-i}2^{-i}\right)\times 2^{e-1023} \tag{3.1}$$

który możemy przekształcić na

$$2^{-52}\times 2^{e-1023} \tag{3.2}$$

funkcja *zad3()* oblicza gęstośc rozłożenia liczb zmiennoprzecinkowych w podanym przedziale. Wystąpienie liczby 52 wynika z faktu, iż w typie podwójnej precyzji do reprezentacji mantysy wykorzystujemy 52 bity (pozostałe 9 bitów rozkłada się w odpowiedni sposób: 1- bit znaku, 8-eksponanta, co od razu warunkuje nam liczbę występującą w wykładniku elementu 2^{e-1023}, gdyż w tym typie bias=1023)

3.3 Wyniki oraz ich interpetacje

Wyniki wywołań funkcji zad3() przedstawia tabela:

Przedział $[2^n, 2^{n+1}]$	zad3()	2^{n-52}
$[\frac{1}{2},1]$	1.1102230246251565e-16	1.1102230246251565e-16
$[1,2]$	2.220446049250313e-16	2.220446049250313e-16
$[2,4]$	4.440892098500626e-16	4.440892098500626e-16

Table 3.1: Porównanie wykorzystania wzoru 3.2 i 2^{n-52} w celu obliczenia odległości między liczbami dla podanych przedziałów

3.4 Wnioski

Na podstawie znajomości standardu IEEE754 oraz wykonanych obliczeń zauważamy, iż odległośc między dwoma następnymi liczb zmiennoprzecinkowych w typie precyzji double w przedziale, który możemy zapisać jako $[2^n, 2^{n+1}]$ wynosi 2^{n-52}. Zauważamy również, iż sprawdzanie ów odległości dla przedziałów, których wartości granicznych nie da się przedstawić w wyżej wymieniony sposób, jest bezcelowa, gdyż w takim przedziale liczby nie są rozłożone równomiernie. Dochodzimy do wniosku, iż liczby w komputerze przedstawiane są swego rodzaju logarytmicznie- najwięcej liczb znajduje się jak najbliżej zera, natomiast im dalej od zera tym "symetryczna siatka", którą tworzą liczby zmiennoprzeciwnkowe, staje się coraz rzadsza, co wpływa na różnice dokładności reprezentacji w zalezności od przedziału.

Zadanie 4

4.1 Krótki opis problemu

Główną problematyką zadania było znalezienie takiej liczby $x \in (1, 2)$, że

$$x * \frac{1}{x} \neq 1 \tag{4.1}$$

co ma pokazać niedokładność reprezentacji liczb w komputerze.

4.2 Rozwiazanie

W swoim rozwiązaniu znajdującym się w pliku zad4.jl postanowiłem iteracyjnie sprawdzać podany warunek 4.1 zaczynając od liczby 1 (typu Float64) i przechodzić podany zbiór korzystając z funkcji nextfloat() aż do momentu, gdy ów warunek nie będzie spełniony. W ten sposób otrzymana liczba spełnia warunki a) i b) zadania. Dodatkowo w pliku zostało zawarte sprawdzenie wypisujące wynik działania dla znalezionej liczby.

4.3 Wyniki oraz ich interpetacje

- Otrzymany wynik: 1.000000057228997
- Wartość 4.1 dla otrzymanego wyniku: 0.9999999999999999

4.4 Wnioski

Ze względu na skończoność reprezentacji liczb zmiennoprzecinkowych w komputerze dochodzi do licznych anomalii, których czasami moglibyśmy nawet nie rozpatrywać ze względu na trywialność równania czy obliczeń, co mogłoby doprowadzić do nielada tragedii.

Zadanie 5

5.1 Krótki opis problemu

Celem zadania było zaimplementowanie czterech różnych sposobów obliczania iloczynu skalarnego dwóch wektorów:

1. Dodawanie "w przód"
2. Dodawanie "w tył"
3. Dodawanie "od największego do najmniejszego"
4. Dodawanie "od najmniejszego do największego"

Oraz sprawdzenie jak kolejność dodawania (względnie odejmowania) składników różnej wielkości wpływa na wynik końcowy

5.2 Rozwiazanie

Implementacja powyższych algorytmów znajduje się w pliku zad5.jl. Funkcja a() odpowiada za implementację pierwszego algorytmu, który polega na iteracyjnym dodawaniu wartości $x_i * y_i$ do globalnej sumy dla $i \in 1, ..., n$, gdzie n jest równe długości wektora.
Funkcja b() odpowiada za implementację drugiego algorytmu, który działa analogicznie do pierwszego z drobną różnicą- przechodzone indeksy w tablicy są w odwortnej kolejności (czyli przechodzimy od $i \in n, ..., 1$, stąd pochodzi różnica w nazewnictwie dwóch algorytmów- w algorytmie 1. z indeksem i idziemy do przodu, natomiast w algorytmie 2. cofamy się).

Funkcja c() odpowiada za implementację algorytmu 3, który polega na dodawaniu "od największego do najmniejszego". W swojej implementacji postanowiłem utworzyć dwa nowe wektory- wektor o nazwie *positive_ array* przechowujący jedynie dodatnie iloczyny $x_i * y_i$ oraz *negative_ array*, który przechowuje jedynie ujemne iloczyny. Następnie dochodzi do osobnego sortowania tablic według odpowiedniego klucza (tablice z wartościami pozytywnymi sortuję od największego korzystając z flagi *rev=true*). Dodatkowo jeśli otrzymany typ to Float32 dokonuję konwersji dodatkowych wektorów na typ Float32.
Ostatecznie dochodzi do zsumowania elementów znajdujących się w *positive_ array*

do zmiennej *positive_sum*. Analogicznie dla wektora *negative_array* oraz zmiennej *negative_sum*.
W ostatnim kroku dodajemy sumy częściowe i otrzymujemy wynik, który przechowujemy w zmiennej *sum*.

Funkcja d() odpowiada za implementację algorytmu 4, który polega na "dodawaniu od najmniejszego do największego". Tutaj należy zauważyć, iż cały proces myślowy odnośnie kroków, które trzeba wykonać, aby otrzymać wynik, jest analogiczny do algorytmu numer 3. Z tego powodu implementacyjnie jedyną różnicą między funkcją d() a c() jest sortowanie- tutaj flaga *rev=true* jest użyta dla *negative_array*.

5.3 Wyniki oraz ich interpetacje

Wyniki działania powyższych algorytmów dla podanych wektorów znajdują się w poniższych tabelach:

Algorytm	Wartość funkcji	Dokładny wynik
1.	-0.4999443	-1.00657107000000e-11
2.	-0.4543457	-1.00657107000000e-11
3.	-0.5	-1.00657107000000e-11
3.	-0.5	-1.00657107000000e-11

Table 5.1: Otrzymane wyniki dla typu danych Float32

Algorytm	Wartość funkcji	Dokładny wynik
1.	1.0251881368296672e-10	-1.00657107000000e-11
2.	-1.5643308870494366e-10	-1.00657107000000e-11
3.	0.0	-1.00657107000000e-11
3.	0.0	-1.00657107000000e-11

Table 5.2: Otrzymane wyniki dla typu danych Float64

Interpretacja: W zależności od użytego algorytmu oraz użytego typu danych otrzymujemy różne wyniki. Spodziewanym efektem było otrzymanie różnic w wynikach, ze względu na rosnący błąd przy każdym dodawaniu, jednakże nie jestem w stanie dokładnie wytłumaczyć skąd biorą się aż takie różnice w otrzymanych wynikach.

5.4 Wnioski

W odróżnieniu od matematyki którą znamy, w komputerze kolejność wykonywania, nawet działań przemiennych, ma duże znaczenie, ze względu na skończoność reprezentacji liczb w komputerze oraz na ich nierówne rozłożenie (co zostało pokazane w Zadaniu 3).

Zadanie 6

6.1 Krótki opis problemu

Problem polega na implementacji dwóch algorytmów:

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

Oraz sprawdzenia różnic w wynikach dla wartości $8^i, i \in [1, n]$, gdzie n jest ustaloną wartością

6.2 Rozwiazanie

Implementacja powyższych funkcji znajduje się w pliku zad6.jl.

6.3 Wyniki oraz ich interpetacje

Otrzymane wyniki zostały zaprezentowane w poniższej tabeli: Interpretacja: Dokładniejsze wyniki działania otrzymujemy dla funkcji *g(x)*, gdyż wiemy, że wynikiem tego działania powinna być liczba większa niż 0 dla każdej potęgi ósemki. Funkcja *f(x)* pokazuje pierwsze 0.0 dla $\frac{1}{8^{-9}}$, gdzie funkcja *g(x)* przedstawia ciągle wartości liczbowe różne od 0 nawet dla $\frac{1}{8^{-100}}$.

6.4 Wnioski

Wykonując działania w artymetyce zmiennoprzecinkowej należy uważac na liczbę cyfr znaczących znajdujących się w rozpatrywanych składnikach działania. Zbyt duża różnica cyfr znaczących może powodować znaczny błąd dokładności. Dodatkowym wnioskiem płynącym w wykonanego zadania jest fakt, iż odejmowanie bardzo małych wartości nie sprawdza się za dobrze, więc w miarę możliwości powinniśmy tego unikać. (Dobrym przykładem jak to robić jest to zadanie- przekształciliśmy wzór funkcji *f(x)* tak, że funkcja *g(x)* nie zawiera w sobie żadnego odejmowania)

x	$f(8^{-x})$	$g(8^{-x})$
1	0.0077822185373186414	0.0077822185373187065
2	0.00012206286282867573	0.00012206286282875901
3	1.9073468138230965e-6	1.907346813826566e-6
4	2.9802321943606103e-8	2.9802321943606116e-8
5	4.656612873077393e-10	4.6566128719931904e-10
6	7.275957614183426e-12	7.275957614156956e-12
7	1.1368683772161603e-13	1.1368683772160957e-13
8	1.7763568394002505e-15	1.7763568394002489e-15
9	0.0	2.7755575615628914e-17
10	0.0	4.336808689942018e-19
20	0.0	3.76158192263132e-37
30	0.0	3.262652339992623e-55
40	0.0	2.8298997121333476e-73
50	0.0	2.454546732648633e-91
60	0.0	2.1289799200040754e-109
70	0.0	1.8465957235571472e-127
80	0.0	1.6016664761464807e-145
90	0.0	1.3892242184281734e-163
100	0.0	1.204959325514 42e-181

Table 6.1: Otrzymane wyniki zaimplementowanych funkcji dla wybranych wartości x

Zadanie 7

7.1 Krótki opis problemu

Celem zadania było zaimplementowanie funkcji, która porzystając ze wzoru

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h} \tag{7.1}$$

Obliczy wartośc pochodnej dla $h \to 0$ oraz $x_0 = 1$ dla funkcji

$$f(x) = \sin(x) + \cos(3x) \tag{7.2}$$

7.2 Rozwiazanie

Korzystając z naszej wiedzy matematycznej wiemy, iż

$$\widetilde{f'}(x) = \cos(x) - 3 * \sin(3x) \tag{7.3}$$

Obliczenie tej funkcji dla x_0 będzie stanowiło podstawę do sprawdzenia wyniku implementacji funkcji 7.1. W pliku zad7.jl zaimplementowano 3 funkcje, które odpowiadają odpowiednio wzorom 7.2 7.3 i 7.1. Następnie w pętli dla podanych w zadaniu wartości n obliczamy h, *1+h* oraz pochodną ze wzoru 7.1 którą porównujemy z 7.3.

7.3 Wyniki oraz ich interpetacje

W tabeli poniżej zostały zawarte wyniki odpowiednich działań:

7.4 Wnioski

Przeprowadzanie obliczeń na liczbach, których wartośc oscyluje w okolicach zera może być niebezpieczne ze względu na niedokładną ich reprezentację (koło samego zera nie znajduje się aż tak dużo liczb ze względu na to, iż komputer potrzebuje "jednoznaczną" reprezentację zera).

h^{-n}	$1+h$	$f'(x)$	$\widetilde{f'}(x)$	różnica
h^{-0}	2.0	0.11694228168853815	2.0179892252685967	1.9010469435800585
h^{-1}	1.5	0.11694228168853815	1.8704413979316472	1.753499116243109
h^{-2}	1.25	0.11694228168853815	1.1077870952342974	0.9908448135457593
h^{-3}	1.125	0.11694228168853815	0.6232412792975817	0.5062989976090435
h^{-4}	1.0625	0.11694228168853815	0.3704000662035192	0.253457784514981
:	:	:	:	:
h^{-10}	1.0009765625	0.11694228168853815	0.12088247681106168	0.0039401951225235265
:	:	:	:	:
h^{-20}	1.0000009536743164	0.11694228168853815	0.11694612901192158	3.8473233834324105e-6
:	:	:	:	:
h^{-25}	1.0000000298023224	0.11694228168853815	0.116942398250103	1.1656156484463054e-7
h^{-26}	1.0000000149011612	0.11694228168853815	0.116942338645458	5.6956920069239914e-8
h^{-27}	1.0000000074505806	0.11694228168853815	0.11694231629371643	3.460517827846843e-8
h^{-28}	1.0000000037252903	0.11694228168853815	0.11694228649139404	4.802855890773117e-9
h^{-29}	1.0000000018626451	0.11694228168853815	0.11694222688674927	5.480178888461751e-8
h^{-30}	1.0000000009313226	0.11694228168853815	0.11694216728210449	1.1440643366000813e-7
h^{-31}	1.0000000004656613	0.11694228168853815	0.11694216728210449	1.1440643366000813e-7
:	:	:	:	:
h^{-35}	1.0000000000291038	0.11694228168853815	0.11693954467773438	2.7370108037771956e-6
:	:	:	:	:
h^{-40}	1.0000000000009095	0.11694228168853815	0.1168212890625	0.0001209926260381522
:	:	:	:	:
h^{-48}	1.0000000000000036	0.11694228168853815	0.09375	0.023192281688538152
h^{-49}	1.0000000000000018	0.11694228168853815	0.125	0.008057718311461848
h^{-50}	1.0000000000000009	0.11694228168853815	0.0	0.11694228168853815
h^{-51}	1.0000000000000004	0.11694228168853815	0.0	0.11694228168853815
h^{-52}	1.0000000000000002	0.11694228168853815	-0.5	0.6169422816885382
h^{-53}	1.0	0.11694228168853815	0.0	0.11694228168853815
h^{-54}	1.0	0.11694228168853815	0.0	0.11694228168853815

Table 7.1: Otrzymane wyniki zaimplementowanych funkcji dla wybranych wartości x