Sprawozdanie Obliczenia Naukowe

Piotr Zapała

October 2023

Spis treści

1 Zadanie 1

1.1 Opis problemu

Pierwsze zadanie polega na iteracyjnym wyznaczeniu epsilonów maszynowych dla wszystkich typów zmiennopozycyjnych zgodnych ze standardem **IEEE 754**, następnie należy porównać z wartościami zwracanymi przez funkcje: `eps(Float16), eps(Float32), eps(Float64)` oraz z danymi zawartymi w pliku `float.h`.

Następnie musimy wyznaczyć liczbę maszynową *eta* dla wszystkich typów zmiennopozycyjnych zgodnych ze standardem **IEEE 754**, następnie należy porównać z wartościami zwracanymi przez funkcje: `nextfloat(Float16(0.0)), nextfloat(Float32(0.0)), nextfloat(Float64(0.0))`.

Ostatnią rzeczą o którą jesteśmy proszeni w zadaniu pierwszym, jest iteracyjne wyznaczenie liczby (MAX) dla wszystkich typów zmiennopozycyjnych zgodnych ze standardem **IEEE 754**, następnie należy porównać z wartościami zwracanymi przez funkcje: `floatmax(Float16), floatmax(Float32), floatmax(Float64)` oraz z danymi zawartymi w pliku `float.h` lub danymi z wykładu.

1.2 Rozwiązanie

1.machine epsilon

Aby wyznaczyć ***epsilon maszynowy*** dla danego typu zmiennopozycyjnego, należy posłużyć się funkcjami **type(x)** oraz **one(type)**. Funkcja **type(x)** zwraca nam **x** w danym typie zmiennopozycyjnym, a funkcja **one(type)** zwraca jedynkę w typie podanym jako argument. Początkowo liczba **epsilon** jest równa **one(type)**, następnie iteracyjnie jest dzielona przez dwa do momentu, aż **one(type) + epsilon / 2 = one(type)**.

2.eta

W celu wyznaczenia liczby ***eta*** ponownie wykorzystujemy funkcje **type(x)** oraz **one(type)**. Pierwszym krokiem jest inicjalizacja liczby ***eta*** jako jedynke w danym typie zmiennopozycyjnym, następnie iteracyjnie dzielimy ją przez dwa, do momentu aż **eta / 2 = 0**.

3.max

Do wyznaczenia liczby ***max*** wykorzystujemy funkcje znane z poprzednich przykładów oraz **prevfloat(x)**, zwracającą największą poprzedzającą **x** liczbę posiadającą reprezentacje w danej arytmetyce. Początkowo ***max*** wynosi **prevfloat(one(type))**, następnie z każdą iteracją jej wartość jest podwajana. Dzieje się to do momentu, aż wartość zwraca przez funkcję **isinf(2*max)**, wynosi nieskończoność.

1.3 Wyniki

type	Float16	Float32	Float64
machEps(type)	0.000977	1.1920929f-7	2.220446049250313e-16
eps(type)	0.000977	1.1920929f-7	2.220446049250313e-16
float.h	———	1.192093e-07	2.220446e-16

Tabela 1: machine epsilon.

W ogólnym modelu precyzję arytmetki zwyczajowo oznaczamy przez ϵ, która wyraża się wzorem

$$\epsilon = \frac{1}{2} * \beta^{1-t} \text{ , gdzie } t \in [\frac{1}{\beta}, 1).$$

Natomiast w standardzie **IEEE 754**, precyzja danej arytmetyki wynosi $\epsilon = 2^{-t}$.

type	`Float16`	`Float32`	`Float64`
ϵ	2^{-11}	2^{-24}	2^{-53}
type(ϵ)	0.0004883	5.9604645f-8	1.1102230246251565e-16
eps(type)	0.000977	1.1920929f-7	2.220446049250313e-16

Tabela 2: precyzja arytmetyki a `machine epsilon`.

Zatem precyzja danej arytmetki w standardzie **IEEE 754** jest równa $\frac{1}{2}$*`machine epsilon`.

type	`Float16`	`Float32`	`Float64`
eta(type)	6.0e-8	1.0f-45	5.0e-324
nextfloat(type(0))	6.0e-8	1.0f-45	5.0e-324
float.h	———	1.175494e-38	2.225074e-308

Tabela 3: eta.

MIN_{sub} jest najmniejszą dla danej arytmetyki liczbą w postaci nieznormalizowanej.
$MIN_{sub} = 2^{1-t} * 2^{c_{min}}$, t jest liczbą cyfr mantysy, przy czym $t \in [1, 2)$,
a $c_{min} = -2^{d-1} + 2$, gdzie d oznacza liczbę bitów mantysy.

type	`Float16`	`Float32`	`Float64`
c_{min}	-14	-126	-1022
MIN_{sub}	2^{-24}	2^{-149}	2^{-1074}
$type(MIN_{sub})$	6.0e-8	1.0e-45	5.0e-324
eta(type)	6.0e-8	1.0e-45	5.0e-324

Tabela 4: MIN_{sub}.

Zatem MIN_{sub} jest równe liczbie **eta(type)** w danej arytmetyce.

MIN_{nor} jest to najmniejszą liczbą jaką można zapisać w danej arytmetyce w postaci znormalizowanej.
$MIN_{nor} = 2^{c_{min}}$, a $c_{min} = -2^{d-1} + 2$.

type	`Float16`	`Float32`	`Float64`
c_{min}	-14	-126	-1022
MIN_{nor}	2^{-14}	2^{-126}	2^{-1022}
$type(MIN_{nor})$	6.104e-5	1.1754944e-38	2.2250738585072014e-308
floatmin(type)	6.104e-5	1.1754944e-38	2.2250738585072014e-308

Tabela 5: MIN_{nor}.

Zatem MIN_{nor} jest równe **floatmin(type)** w danej arytmetyce.

type	`Float16`	`Float32`	`Float64`
maxFloat(type)	6.55e4	3.4028235f38	1.7976931348623157e308
floatmax(type)	6.55e4	3.4028235f38	1.7976931348623157e308
float.h	———	3.402823e+38	1.797693e+308

Tabela 6: float max

1.4 Wnioski

Wyniki przedstawione w tabelach, jednoznacznie wskazują na to, że standard **IEEE 754** ma pewne ogarniczenia. Wraz ze wzrostem liczby bitów jakie przeznaczamy na zapis danej liczby, pewne niedogodności arytmetyki są mniej zauważalne. Zwiększanie liczby bitów, pozwala nam na zwiększanie zakresu liczb na których możemy operować.

2 Zadanie 2

2.1 Opis problemu

W zadaniu drugim jesteśmy proszeni o przeprowadzenie serię eksperymentów, mających na celu sprawdzenie, czy dla wszystkich typów zmiennopozycyjnych, obliczenie wyrażenia **3(4/3-1)-1** daje *epsilon maszynowy*.

2.2 Rozwiązanie

W celu rozwiązania zadania drugiego należy posłużyć się funkcjami **one(type)** oraz **type(x)**.

2.3 Wyniki

type	Float16	Float32	Float64
eps(type)	0.000977	1.1920929f-7	2.220446049250313e-16
kahanEps(type)	-0.000977	1.1920929f-7	-2.220446049250313e-16

Tabela 7: kahan eps.

2.4 Wnioski

Przez ogarniczoną dokładność arytmetyki zmiennopozycyjnej, jesteśmy narażeni na błędy nawet w bardzo trywialnych przykładach.

3 Zadanie 3

3.1 Opis problemu

W zadaniu trzecim musimy sprawdzić, czy liczby zmiennopozycyjne w arytmetyce `Float64` są równo rozmieszczone z następującym krokiem $\delta = 2^{-54}$ w przedziałach: [1,2], [$\frac{1}{2}$,1] oraz [2,4].

3.2 Rozwiązanie

Korzystamy ze wskazówki i używamy funkcji **bitstring**, dzięki niej jesteśmy w stanie wypisać reprezentacje binarne liczb. Sprawdzamy po 10 pierwszych liczb z danego przedziału, gdyż sprawdzenie całości mogłoby okazać się kłopotliwe.

3.3 Wyniki

Pierwsze 10 liczb w przedziale [2, 4]
0011111111110000000000000000000000000000000000000000000000000000
0011111111110000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000011
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000101

Tabela 8: przedział [2, 4].

Pierwsze 10 liczb w przedziale [1, 2]
0011111111110000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000011
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000101
0011111111110000000000000000000000000000000000000000000000000110
0011111111110000000000000000000000000000000000000000000000000111
0011111111110000000000000000000000000000000000000000000000001000
0011111111110000000000000000000000000000000000000000000000001001
0011111111110000000000000000000000000000000000000000000000001010

Tabela 9: przedział [1, 2]

Pierwsze 10 liczb w przedziale [1/2, 1]
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000110
0011111111110000000000000000000000000000000000000000000000001000
0011111111110000000000000000000000000000000000000000000000001010
0011111111110000000000000000000000000000000000000000000000001100
0011111111110000000000000000000000000000000000000000000000001110
0011111111110000000000000000000000000000000000000000000000010000
0011111111110000000000000000000000000000000000000000000000010010
0011111111110000000000000000000000000000000000000000000000010100

Tabela 10: przedział [1/2, 1]

Możemy zauważyć, iż gęstość rozmieszenia liczb w przedziale [2, 4] jest dwukrotnie większa niż w przypadku przedziału [1, 2].

3.4 Wnioski

Z powyższego przykładu widzimy, iż to jaka odległość dzieli kolejne liczby (jaka jest precyzja arytmetyki) mające reprezentację w danej arytmetyce, zależy od przedziału w którym znajdują się te liczby.

4 Zadanie 4

4.1 Opis problemu

W pierwszym podpunkcie zadania czwartego jesteśmy proszeni o to, aby w arytmetyce `Float64` zgodnej ze standardem **IEEE 754** znaleźć taką liczbę x w przedziale $1 < x < 2$, że $x * (\frac{1}{x}) \neq 1$. Natomiast podpunkt drugi polega na wyznaczeniu najmniejszej takiej liczby.

4.2 Rozwiązanie

W tym zadaniu należy się posłużyć funkcjami **one(type)**, **zero(type)** oraz funkcją **nextfloat(x)**. Funkcja **nextfloat(x)**, zwaraca najmniejszą liczbę, większą od **x**, która posiada reprezentację w danej arytmetyce zmiennopozycyjnej. W przykładzie pierwszym zaczynamy od jedynki, następnie wyznaczamy iteracyjnie następną liczbę posiadającą reprezentację w naszej arytmetyce. Czynność powtarzamy do momentu, aż nie znajdziemy liczby która nie spełnia danej równość $x * (\frac{1}{x}) = 1$. Przykład drugi jest analogiczny, z taką różnicą, że tym razem zaczynamy od zera.

4.3 Wyniki

Wyniki prezentują się następująco:
1. $x = 1.000000057228997$
2. $x = 1.0e - 323$

4.4 Wnioski

Analizując podane przykłady możemy wywnioskować, iż **IEEE 754** ma ograniczoną dokładność i nawet bardzo proste operacje mogą dawać fałszywe wyniki.

5 Zadanie 5

5.1 Opis problemu

W zadaniu piątym jesteśmy proszeni o zaimplemetowanie, na cztery różne sposoby, algorytm obliczania iloczynu skalarnego wektorów, a następnie mamy obliczyć ten iloczyn dla następującej pary wektorów:

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

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

(a) "w przód" $\sum_{i=1}^{n} x_i y_i$, tj. algorytm

$S := 0$
for i:=1 **to** n **do**
    $S := S + x_i * y_i$
end for

(b) "w tył" $\sum_{i=n}^{1} x_i y_i$, tj. algorytm

$S := 0$
for i:=n **downto** 1 **do**
    $S := S + x_i * y_i$
end for

(c) dodatnie dodajemy w porządku od największego do najmniejszego, a ujemne w porządku od najmniejszego do największego, następnie dodajemy do siebie obliczone sumy częściowe.
(d) przeciwnie do metody (c).

5.2 Rozwiązanie

Rozwiązanie polega na zaimplementowaniu podanych algorytmów, a następnie obliczeniu iloczynu skalarnego dla każdego z osobna.

5.3 Wyniki

type	Float32	Float64
example1	-0.49999443f0	-5.0134667188046684e-5
example2	-0.4543457f0	-5.013464760850184e-5
example3	-0.5f0	-5.013449117541313e-5
example5	-0.5f0	-5.013495683670044e-5

Tabela 11: iloczyn skalarny.

5.4 Wnioski

Przyglądając się wynikom, nasuwają się dwa wnioski, pierwszym jest fakt, iż na wynik wpływa typ arytmetyki, którą się posługujemy. Drugą rzeczą jest to, iż na końcowy wynik ma również wpływ kolejność wykonywania działań.

6 Zadanie 6

6.1 Opis problemu

W zadaniu szóstym jesteśmy proszeni o to, aby w arytmetyce `Float64` obliczyć wartości następujących funkcji:

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

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

dla kolejnych wartości argumentu $x = 8^{-1}, 8^{-2}, 8^{-3}, \ldots$. Następnie należy odpowiedzieć, które wyniki są wiarygodne, bo pomimo iż $f = g$, komputer daje nam różne rozwiązania.

6.2 Rozwiązanie

Rozwiązanie polega na zaimplementowaniu podanych równań, a następnie obliczaniu ich wartości dla kolejnych argumentów $x = 8^{-1}, 8^{-2}, 8^{-3}, \ldots$.

6.3 Wyniki

Pomimo równości f i g, widzimy znaczącą różnicę w otrzymywanych wynikach. W tym przypadku jest to spowodowane odejmowaniem we wzorze funkcji f. Dla bardzo małych wartości x, mamy $\sqrt{x^2 + 1} \approx 1$, powoduje to utratę cyfr znaczących z wyniku pierwiastka.

6.4 Wnioski

W przypadkach w których może wystąpić odjemowanie bliskich sobie liczb, należy się zastanowić, czy nie jesteśmy w stanie wykonać przekształcenia algebraicznego. Gdyż może to pozwolić na prowadzenie bardziej dokładnych obliczeń.

x	f(x)	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^{-10}	0.0	4.336808689942018e-19
8^{-20}	0.0	3.76158192263132e-37
8^{-30}	0.0	3.2626522339992623e-55
8^{-40}	0.0	2.8298997121333476e-73
8^{-50}	0.0	2.4545467326488633e-91
8^{-60}	0.0	2.1289799200040754e-109
8^{-70}	0.0	1.8465957235571472e-127
8^{-80}	0.0	1.6016664761464807e-145
8^{-90}	0.0	1.3892242184281734e-163
8^{-100}	0.0	1.204959932551442e-181

Tabela 12: porównanie wartości funkcji f i g dla $x = 8^{-1}, 8^{-2}, 8^{-3}, \ldots$.

7 Zadanie 7

7.1 Opis problemu

Korzystając ze wzoru na przybliżoną wartość pochodnej $f(x)$ w punkcie x,

$$f'(x) \approx \tilde{f}'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h}$$

w arytmetyce `Float64` należy obliczyć przybliżoną wartość pochodnej funkcji $f(x) = \sin x + \cos 3x$ w punkcie $x_0 = 1$ oraz błędów $|f'(x_0) - \tilde{f}'(x_0)|$ dla $h = 2^{-n}$ $(n = 0, 1, 2, \ldots, 54)$.
Następnie należy wyjaśnić dlaczego od pewnego momentu zmniejszanie wartości h nie poprawia przybliżenia wartości pochodnej, sprawdzić jak zachowują się wartości $1 + h$ oraz porównać obliczone przybliżenia pochodnej z jej dokładną wartością. Pochodna funkcji f ma następującą postać $f'(x) = \cos x - 3 \sin 3x$.

7.2 Rozwiązanie

Rozwiązanie polega na zaimplementowaniu podanych równań, a następnie obliczaniu ich wartości dla kolejnych h oraz $1 + h$.

7.3 Wyniki

Dokładna wartość pochodnej w punkcie wynosi 0.11694228168853815

h	derivative(f, h, x0)	relativeError(f, g, h, x0)
1.0	2.0179892252685967	1.9010469435800585
0.5	1.8704413979316472	1.753499116243109
0.25	1.1077870952342974	0.9908448135457593
0.125	0.6232412792975817	0.5062989976090435
0.0625	0.3704000662035192	0.253457784514981
0.03125	0.24344307439754687	0.1265007927090087
0.015625	0.18009756330732785	0.0631552816187897
0.0078125	0.1484913953710958	0.03154911368255764
0.00390625	0.1327091142805159	0.015766832591977753
0.001953125	0.1248236929407085	0.007881411252170345
0.0009765625	0.12088247681106168	0.0039401951225235265
0.000244140625	0.11792723373901026	0.0009849520504721099
0.0001220703125	0.11743474961076572	0.0004924679222275685
6.103515625e-5	0.11718851362093119	0.0002462319323930373
3.0517578125e-5	0.11706539714577957	0.0001231154572414183 7
1.52587890625e-5	0.11700383928837255	6.155759983439424e-5
7.62939453125e-6	0.11697306045971345	3.077877117529937e-5
3.814697265625e-6	0.11695767106721178	1.5389378673624776e-5
1.9073486328125e-6	0.11694997636368498	7.694675146829866e-6
9.5367431640625e-7	0.11694612901192158	3.8473233834324105e-6
4.76837158203125e-7	0.1169442052487284	1.9235601902423127e-6
2.384185791015625e-7	0.11694324295967817	9.612711400208696e-7
1.1920928955078125e-7	0.11694276239722967	4.807086915192826e-7
5.960464477539063e-8	0.11694252118468285	2.394961446938737e-7
2.9802322387695312e-8	0.116942398250103	1.1656156484463054e-7
1.4901161193847656e-8	0.11694233864545822	5.6956920069239914e-8
7.450580596923828e-9	0.11694231629371643	3.460517827846843e-8
3.725290298461914e-9	0.11694228649139404	4.802855890773117e-9
1.862645149230957e-9	0.11694222688674927	5.480178888461751e-8
9.313225746154785e-10	0.11694216728210449	1.1440643366000813e-7
4.656612873077393e-10	0.11694216728210449	1.1440643366000813e-7
2.3283064365386963e-10	0.11694192886352539	3.5282501276157063e-7
1.1641532182693481e-10	0.11694145202636719	8.296621709646956e-7
5.820766091346741e-11	0.11694145202636719	8.296621709646956e-7
2.9103830456733704e-11	0.11693954467773438	2.7370108037771956e-6
1.4551915228366852e-11	0.116943359375	1.0776864618478044e-6
7.275957614183426e-12	0.1169281005859375	1.4181102600652196e-5
3.637978807091713e-12	0.116943359375	1.0776864618478044e-6
1.8189894035458565e-12	0.11688232421875	5.9957469788152196e-5
9.094947017729282e-13	0.1168212890625	0.000120992626038152
4.547473508864641e-13	0.116943359375	1.0776864618478044e-6
2.2737367544323206e-13	0.11669921875	0.0002430629385381522
1.1368683772161603e-13	0.1162109375	0.0007313441885381522
5.684341886080802e-14	0.1171875	0.0002452183114618478
2.842170943040401e-14	0.11328125	0.003661031688538152
1.4210854715202004e-14	0.109375	0.007567281688538152
7.105427357601002e-15	0.109375	0.007567281688538152
3.552713678800501e-15	0.09375	0.023192281688538152
1.7763568394002505e-15	0.125	0.008057718311461848
8.881784197001252e-16	0.0	0.11694228168853815
4.440892098500626e-16	0.0	0.11694228168853815
2.220446049250313e-16	-0.5	0.6169422816885382
1.1102230246251565e-16	0.0	0.11694228168853815
5.551115123125783e-17	0.0	0.11694228168853815

Tabela 13: porównanie wyników dla h.

Najlepsze przybliżenie pochodnej w punkcie $x_0 = 1$ otrzymujemy do $h = 2^{-28}$. Zatem zmniejszanie wartości h od pewnego wcale nie poprawia jakości naszych obliczeń, wręcz przeciwnie. Jest to spowodowane odejmowaniem dwóch bliskich sobie liczb.

1 + h	derivative(f, 1 + h, x0)	relativeError(f, g, 1 + h, x0)
2.0	-0.3107443710161304	0.42768665270466855
1.5	0.7290859824876875	0.6121437007991494
1.25	1.4556808426956374	1.3387385610070992
1.125	1.7730038158791617	1.6560615341906235
1.0625	1.90633124732842	1.7893889656398818
1.03125	1.9650746526601908	1.8481323709716526
1.015625	1.992284872504413	1.875342590815875
1.0078125	2.0053282295936543	1.8883859479051162
1.00390625	2.011706884941135	1.8947646032525967
1.001953125	2.0148601393872734	1.8979178576987352
1.0009765625	2.016427708987274	1.8994854272987358
1.000244140625	2.0175994143067255	1.9006571326181874
1.0001220703125	2.0177943671554126	1.9008520854668745
1.00006103515625	2.017891808055301	1.9009495263667628
1.000030517578125	2.0179405196229427	1.9009982379344046
1.0000152587890625	2.01796487318604	1.9010225914975019
1.0000076293945312	2.017977049412388	1.90103476772385
1.0000038146972656	2.0179831373867603	1.9010408556982221
1.0000019073486328	2.0179861813392455	1.9010438996507073
1.0000009536743164	2.0179877033068125	1.9010454216182744
1.0000004768371582	2.017988464288428	1.9010461825998899
1.000000238418579	2.0179888447786927	1.9010465630901545
1.0000001192092896	2.0179890350236898	1.9010467533351516
1.0000000596046448	2.0179891301461548	1.9010468484576166
1.0000000298023224	2.0179891777073786	1.9010468960188405
1.0000000149011612	2.0179892014879885	1.9010469197994504
1.0000000074505806	2.0179892133782924	1.9010469316897542
1.0000000037252903	2.0179892193234443	1.9010469376349062
1.0000000018626451	2.0179892222960207	1.9010469406074826
1.0000000009313226	2.0179892237823087	1.9010469420937706
1.0000000004656613	2.017989224525453	1.9010469428369148
1.0000000002328306	2.0179892248970246	1.9010469432084864
1.0000000001164153	2.0179892250828106	1.9010469433942725
1.0000000000582077	2.0179892251757034	1.9010469434871653
1.0000000000291038	2.0179892252221503	1.9010469435336121
1.000000000014552	2.0179892252453735	1.9010469435568353
1.000000000007276	2.017989225256985	1.901046943568447
1.000000000003638	2.017989225262791	1.901046943574253
1.000000000001819	2.0179892252656937	1.9010469435771555
1.0000000000009095	2.017989225267145	1.9010469435786068
1.0000000000004547	2.017989225267871	1.901046943579333
1.0000000000002274	2.017989225268234	1.9010469435796957
1.0000000000001137	2.017989225268415	1.901046943579877
1.0000000000000568	2.017989225268506	1.901046943579968
1.0000000000000284	2.0179892252685514	1.9010469435800132
1.0000000000000142	2.017989225268574	1.901046943580036
1.000000000000007	2.017989225268585	1.901046943580047
1.0000000000000036	2.017989225268591	1.9010469435800528
1.0000000000000018	2.017989225268594	1.9010469435800559
1.0000000000000009	2.0179892252685954	1.9010469435800572
1.0000000000000004	2.017989225268596	1.9010469435800577
1.0000000000000002	2.0179892252685963	1.901046943580058
1.0	2.0179892252685967	1.9010469435800585
1.0	2.0179892252685967	1.9010469435800585

Tabela 14: porównanie wyników dla 1 + h.

7.4 Wnioski

Prawadząc tego typu obliczenia, powinniśmy się wystrzegać wartości bliskich zera.