Sprawozdanie z listy 1 na Obliczenia Naukowe

Adrian Herda, 268449

October 22, 2023

1 Rozpoznanie arytmetyki

Zadanie to polegało na obliczeniu *epsilonu maszynowego*, *ety maszynowej* oraz *MAX* dla wszystkich typów zmiennopozycyjnych, w standarzie **IEEE 754**, dostepnych w jezyku *Julia*. W jezyku tym mamy dostepne 3 takie typy: Float16, Float32, Float64. Typy te reprezuntuja precyzje 16-, 32- oraz 64-bitowa.

- Epsilon maszynowy - najmniejesza liczba spełniajaca równanie $fl(1.0 + macheps) > 1.0$, która da rade reprezentować w arytmetyce zmiennopozycyjnej
- Eta maszynowa - najmniejsza liczba wieksza od zera, która da rade reprezentować w arytmetyce zmiennopozycyjnej
- MAX - najwieksza liczba, która można reprezentować w arytmetyce zmiennopozycyjnej

1.1 Rozwiazania

Rozwiazania polegaja na iteracyjnym zmniejszaniu (lub powiekszaniu w przypadku funkcji MAX) wybranej liczby aż do granicy w której nie da rady mieć bardziej prezycyjnej liczby. Wtedy liczba jest zwracana jako wynik.

1.1.1 Epsilon maszynowy

```
# Funkcja znajduje najmniejsza liczbe wieksza od 1 dla typu T i zwraca jej
    roznice od 1
function machine_epsilon(T)
    result = T(2.0)
    epsilon = T(1.0)

    while one(T) + epsilon > one(T)
        result /= T(2.0)
        epsilon /= T(2.0)
    end

    return result
end
```

Listing 1: Obliczanie epsilonu maszynowego

1.1.2 Eta maszynowa

```
# Funkcja znajduje najmniejsza liczbe wieksza od 0 dla typu T
function machine_eta(T)
    result = T(2.0)
    eta = T(1.0)

    while eta > zero(T)
        result /= T(2.0)
        eta /= T(2.0)
    end
```

```
10
11      return result
12  end
```

Listing 2: Obliczanie ety maszynowej

1.1.3 MAX

```
1   # Funkcja pomocnicza
2   # Funckja znajduje najwieksza liczbe mniejsza od liczby x dla typu T
3   function find_previous_float(x, T)
4       x = T(x)
5       diff = T(1.0)
6       while x - diff < x
7           diff /= T(2.0)
8       end
9
10      diff *= T(2.0)
11
12      return x - diff
13  end
14
15  # Funckcja znajduje najwi ksza mo liwa liczbe
16  function max(T)
17      max = find_previous_float(1.0, T)
18
19      while !isinf(max * T(2.0))
20          max *= T(2.0)
21      end
22
23      return max
24  end
```

Listing 3: Obliczanie MAX

1.2 Wyniki oraz ich interpretacja

1.2.1 Epsilon maszynowy

	Float16	Float32	Float64
machine_eps()	0.000977	1.1920929e-7	2.220446049250313e-16
eps()	0.000977	1.1920929e-7	2.220446049250313e-16
float.h		1.19209e-07	2.22045e-16

Wyniki pokazuja że iteracyjny sposób jest dobry, a plik nagłówkowy jezyka C nie jest aż tak dokładny.

1.2.2 Eta maszynowa

	Float16	Float32	Float64
machine_eta()	6.0e-8	1.0e-45	5.0e-324
nextfloat(zero())	6.0e-8	1.0e-45	5.0e-324

Ponownie widzimy że iteracyjny sposób daje nam poprawne wyniki.

1.2.3 MAX

	Float16	Float32	Float64
max()	6.55e4	3.4028235e38	1.7976931348623157e308
floatmax()	6.55e4	3.4028235e38	1.7976931348623157e308
float.h		3.40282e38	1.79769e308

Po raz trzeci upewniamy sie że nasz sposób wyliczania charakterystycznych liczb jest poprawny. Plik nagłówkowy jezyka C nie daje aż tak dokładnych wyników.

1.2.4 Precyzja arytmetyki

Precyzja arytmetyki jest dokładnościa reprezentowania liczb w danym systemie liczbowym. Czesto jest reprezentowana przez grecka litere ϵ i w naszym przypadku jest obliczana wzorem

$$\epsilon = 2^{-t}$$

gdzie t to liczba cyfr mantysy. Po obliczeniu wartości ϵ dla typów Float16, Float32 oraz Float64 i porównaniu z obliczonymi przez nas epsilonami maszynowymi zauważymy że wyniki sa takie same. Wnioskujemy zatem że ϵ jest równy epsilonowi maszynowemu.

1.2.5 $\mathbf{MIN}_{sub}$

Liczba ta jest najmnijesza nieznormalizowana liczba, która można reprezentować za pomoca danej arytmetyki. Po porównaniu jej z eta maszynowa, która jest nastepna liczba po zerze, zauważymy że sa to te same liczby.

1.2.6 $\mathbf{MIN}_{nor}$

Funkcja floatmin() zwraca najmniejsza znormalizowana liczbe dla danego typu. Jest to definicja liczby MIN_{nor}, wiec te dwie liczby sa sobie równe.

1.3 Wnioski

Pierwszy i najważniejszy wniosek to że arytmetyka i reprezentacje liczb w komputerach nie zawsze sa precyzyjne, liczby nie sa ciagłe i nie każda liczbe rezeczywista możemy dokładnie zapisać w pamieci komputera. Tu z pomoca przychodzi nam standard IEEE 754 który pozwala na dość dokładne reprezentowanie liczb ale cały czas ma swoje ograniczenia. Należy również pamietać o tym że liczby moga być znormalizowane i nieznormalizowane, gdyż czasami może to przypożyć problemów.

2 Wyrażenie Kahana

Zadanie 2 polegało na eksperymentalnym przetestowaniu stwierdzenia Kahana. Według niego epsilon maszynowy można obliczyć za pomoca wzoru

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

Naszym zadaniem było sprawdzić to dla wszystkich typów zmiennopozycyjnych w jezyku Julia

2.1 Rozwiazanie

Funkcja napisana przez mnie zmienia wszystkie liczby na podany w parametrze T typ i oblicza wynik podanego przez Kahana wyrażenia.

```
# Funkcja znajdujaca epsilon maszynowy metoda Kahana dla typu T
function macheps(T)
    return T(3) * (T(4) / T(3) - T(1)) - T(1)
end
```

Listing 4: Stwierdzenie Kahana

2.2 Wyniki oraz ich interpretacja

	Float16	Float32	Float64
Kahan	-0.000977	1.1920929e-7	-2.220446049250313e-16
eps()	0.000977	1.1920929e-7	2.220446049250313e-16

Wartości wyników zgadzaja sie, problem natomiast jest ze znakami w wynikach. Powodem wydaje sie być przybliżenie wyniku dzielenia $\frac{4}{3}$ które dla Float32 jest przybliżeniem w góre a dla pozostałych typów jest przybliżeniem w dół.

2.3 Wnioski

Wniosków z tego zadania może być kilka. Jednym z nich może być że istnieja łatwiejsze sposoby na obliczanie epsilonu maszynowego niż w pierwszym zadaniu. Kolejnym może być że nie każde obliczenia, mimo że dla nas proste dadza dobry wynik przy skończonej dokładności.

3 Równomierne rozmieszczenie liczb

W tym zadaniu mieliśmy sprawdzić eksperymentalnie czy liczby w arytmetyce Float64 w jezyku Julia liczby sa rozmieszone równomiernie w zakresie [1, 2). Każda liczba w podanym zakresie powinna móc być przedstawiona w postaci $x = 1 + k\delta$ gdzie $k = 1, 2, ..., 2^{52} - 1$ oraz $\delta = 2^{-52}$.

3.1 Rozwiazanie

Z racji na ogromna ilość liczb w tym przedziale, nie jest możliwe sprawdzenie wszystkich liczb po kolei. Możemy natomiast sprawdzić poczatek i koniec przedziału aby sprzawdzić czy tam zachowana została własność równomiernego rozmieszczenia z róznicami pomiedzy liczbami równymi δ. Innym sposobem sprawdzenia równomierności rozmieszczenia jest porównanie eksponenty, pierwszej i ostaniej liczby z przedziału ponieważ jeśli sa takie same to jedyne co mogło sie zmieniać to mantysa a wtedy róznice pomiedzy liczbami sa na 100% takie same

```
# Funkcja sprawdzjaca rownomiernie rozmieszone dla 100 pierwszych liczb wyznaczanych przez start jako pierwsza liczbe oraz func jako funkcja wyznaczjaca kolejna liczbe
# delta to oczekiwana roznica pomiedzy liczbami
# funkcja zwraca true jesli zachowane jest rownomiernie rozmieszenie i false jesli nie jest zachowane
function rownomiernie_rozmieszone_na_ogonach(start, delta, func)
    rownomiernie_rozmieszone = true
    for i in 0:100
        if func(start) != start + delta
            rownomiernie_rozmieszone = false
            break
        end
        start = func(start)
    end

    return rownomiernie_rozmieszone
end

# Sprawdzanie poczatku przedzialu [1, 2)
println("Rownomierne rozmieszenie na poczatku przedzialu [1, 2): ", rownomiernie_rozmieszone_na_ogonach(one(Float64), eps(Float64), nextfloat))
# Sprawdzanie konca przedzialu [1, 2)
println("Rownomierne rozmieszenie na koncu przedzialu [1, 2): ", rownomiernie_rozmieszone_na_ogonach(one(Float64) + 1, -eps(Float64), prevfloat))

# Porownywanie bit w liczb
x = one(Float64)

println(prevfloat(x), " - " , bitstring(prevfloat(x)))
println(x, " - " , bitstring(x))
println(nextfloat(x), " - " , bitstring(nextfloat(x)))
println(nextfloat(nextfloat(x)), " - " , bitstring(nextfloat(nextfloat(x))))
println(prevfloat(x+1), " - " , bitstring(prevfloat(x+1)))
println(x+1, " - " , bitstring(x+1))
```

Listing 5: Równomierne rozmieszczenie liczb

3.2 Wyniki oraz ich interpretacja

“Równomierne rozmieszenie na poczatku przedziału [1, 2): true
Równomierne rozmieszenie na końcu przedziału [1, 2): true“

x	liczba	bitstring(x)
prevfloat(1.0)	0.9999999999999999	0**01111111110**1111111111111111111111111111111111111111111111111111
1.0	1.0	0**01111111111**0000000000000000000000000000000000000000000000000000
nextfloat(1.0)	1.0000000000000002	0**01111111111**0000000000000000000000000000000000000000000000000001
nextfloat(nextfloat(1.0))	1.0000000000000004	0**01111111111**0000000000000000000000000000000000000000000000000010
prevfloat(2.0)	1.9999999999999998	0**01111111111**1111111111111111111111111111111111111111111111111111
2.0	2.0	0**10000000000**0000000000000000000000000000000000000000000000000000

Wyniki stworzonej przeze mnie potwierdzaja że rozmieszczenie jest równomierne. Podobnie porównanie bitów pokazuje, że eksponenta sie nie zmieniła a wiec rozmieszenie jest równomierne.

3.3 Wnioski

Liczby pomiedzy kolejnymi potegami dwójki sa równomiernie rozmieszczone. Im wieksze potegi dwójki tym wieksze różnice pomiedzy kolejnymi liczbami.

Zakres	δ
$[\frac{1}{2}, 1)$	1.1102230246251565e-16
$[1, 2)$	2.220446049250313e-16
$[2, 4)$	4.440892098500626e-16

4 Liczba nie spełniajaca równania

W zadaniu 4 mieliśmy za zadanie znaleźć jakakolwiek liczbe zmiennopozycyjna typu Float64 taka, że:

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

$$1 < x < 2$$

a także znaleźć najmniejsza taka liczbe.

4.1 Rozwiazanie

Funkcja, która napisałem, za parametry przyjmuje poczatek i koniec przeszukiwanego przedziału oraz funkcje, która wyznacza nastepna liczbe do sprawdzenia. Wewnatrz mojej funkcji jest petla która przechodzi po kolejnych liczbach zmiennopozycyjnych i sprawdza czy spełniaja one podane w zadaniu równanie. Jeśli tak to swraca odpowiedź czy znaleziono oraz znaleziona liczbe, jeśli nie znaleziono to zwraca odpowiedź i koniec przedziału.

```
1  # Funkcja znajduje liczbe nie spelniajaca rownania x * 1/x = 1 w przedziale
       (start, end_)
2  # parametr func to funkcja ktora wyzancza kolejna sprawdzana liczbe
3  # zwraca: znaleziono - true jesli znaleziono liczbe nie spelniajaca rownania
       lub false jesli nie znaleziono
4  #         start - jesli znaleziono jest true to bedzie to liczba nie
       spelniajaca rowania
5  function find_ne(start, end_, func)
6      flag = true
7      if start > end_
8          flag = false
9      end
10     znaleziono = false
11     while (flag && start < end_) || (!flag && end_ < start)
12         if Float64(start * Float64(1 / start)) != 1
13             znaleziono = true
14             break
```

```
        end
        start = func(start)
    end
    return (start, znaleziono)
end

println("Szukanie liczby nie spelniajacej dzialania x * 1/x = 1 na poczatku przedzialu (1, 2): ", find_ne(nextfloat(Float64(1)), prevfloat(Float64(2)), nextfloat))
println("Szukanie liczby nie spelniajacej dzialania x * 1/x = 1 na ko cu przedzialu (1, 2): ", find_ne(prevfloat(Float64(2)), nextfloat(Float64(1)), prevfloat))
println("Najmniejsza liczba nie spelniajaca rownania x * 1/x = 1 w przedziale (1, 2): ", find_ne(nextfloat(Float64(1)), prevfloat(Float64(2)), nextfloat))
```

Listing 6: Szukanie liczby x nie spelniajacej równaia

4.2 Wyniki i interpretacja

Wystarczy znaleźć najmniejsza liczbe, żeby wykonać oba podpunkty zdania. Liczba taka jest:

$$x = 1.000000057228997$$

4.3 Wnioski

Arytmetyka liczb zmiennopozycyjnych ze wzgledu no swoja niedokładność potrafi źle obliczyć nawet najprostsze działania.

5 Iloczyn skalarny wektorów

W tym zadaniu mieliśmy przetestować rózne sposoby obliczania iloczynu skalarnego dla pojedynczej i podwójnej precyzji (typów Float32 oraz Float64). Wybranymi sposobami liczenia iloczynu skalarnego były:

1. "W przód" - $\sum_{i=1}^{n} x_i y_i$
2. "W tył" - $\sum_{i=n}^{1} x_i y_i$
3. Najpierw zsumować liczby dodatnie od najwiekszej do najmniejszej a potem liczby ujemne od najmniejszej do najwiekszej
4. Njapierw zsumować liczby dodatnie od najmnijeszej do najwiekszej a potem liczby ujemne od najwiekszej do najmnijeszej

5.1 Rozwiazanie

Moje funkcje przyjumja 3 argumenty:

- x - pierwszy wektor
- y - drugi wektor
- T - typ danych które ma liczyć funkcja

oblicza iloczyn skalarny a nastepnie sumuje i zwraca wynik.

5.1.1 Sposób "W przód"

```
# Funkcja zwraca iloczyn skalarany x oraz y liczony "w przod"
# T to typ zmiennej
function a(x, y, T)
    result = T(0)
    n = length(x)

    for i in 1:n
        result += x[i] * y[i]
    end

    return result
end
```

Listing 7: Sposób nr. 1

5.1.2 Sposób "W tył"

```
# Funkcja zwraca iloczyn skalarany x oraz y liczony "w ty "
# T to typ zmiennej
function b(x, y, T)
    result = T(0)
    n = length(x)

    for i in n:-1:1
        result += (x[i] * y[i])
    end

    return result
end
```

Listing 8: Sposób nr. 2

5.1.3 Sposób "Najwieksze -¿ najmniejszych"

```
# Funkcja zwraca iloczyn skalarany x oraz y liczony:
#       Dodatnie od najwiekszego do najmniejszego
#       Ujemne od najmnijeszego do najwiekszego
# T to typ zmiennej
function c(x, y, T)
    to_sum = x .* y

    sum_positive = filter(a -> a >= 0, to_sum)
    sum_negative = filter(a -> a < 0, to_sum)

    sum_positive = sort(sum_positive, rev=true)
    sum_negative = sort(sum_negative)

    sum = T(0)
    for i in eachindex(sum_positive)
        sum += sum_positive[i]
    end
    for i in eachindex(sum_negative)
        sum += sum_negative[i]
    end

    return sum
end
```

Listing 9: Sposób nr. 3

5.1.4 Sposób "Najmniejsze do najwiekszych"

```
# Funkcja zwraca iloczyn skalarany x oraz y liczony:
#       Dodatnie od najmnijeszego do najwiekszego
#       Ujemne od najwiekszego do najmniejszego
# T to typ zmiennej
function d(x, y, T)
    to_sum = x .* y

    sum_positive = filter(a -> a >= 0, to_sum)
    sum_negative = filter(a -> a < 0, to_sum)

    sum_positive = sort(sum_positive)
    sum_negative = sort(sum_negative, rev=true)

    sum = T(0)
    for i in eachindex(sum_positive)
        sum += sum_positive[i]
    end
    for i in eachindex(sum_negative)
        sum += sum_negative[i]
    end

    return sum
end
```

Listing 10: Sposób nr. 4

5.2 Wyniki oraz ich interpretacja

	Float32	Float64	Prawdziwy wynik
Sposób nr. 1	-0.4999443	1.0251881368296672e-10	-1.00657107000000e-11
Sposób nr. 2	-0.4543457	-1.5643308870494366e-10	-1.00657107000000e-11
Sposób nr. 3	-0.39291382	1.4068746168049984e-12	-1.00657107000000e-11
Sposób nr. 4	-0.5	0.0	-1.00657107000000e-11

Precyzja pojedyncza wykazuje wieksze błedy niż przecycja podwójna. W precyzji podwójnej pojawiły sie wyniki które maja w sobie nawet błedny znak. Sposób nr. 4 wydaje sie być najgorszym za wszystkich algorytmów.

5.3 Wnioski

Wyniki działań, mimo że nie powinny, moga zależeć od kolejności ich wykonywania. Różnice jakie wynikna z tego typu problemu moga być zadziwiajaco duże.

6 Porównanie f(x)=g(x)

W zadaniu nr. 6 należało porównwać, w podójnej precyzji, dwie nastepujace funkcje:

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

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

Funkcjce te wydaja sie dla nas ludzi być takie same, ponieważ takie sa. Ale jak w poprzednim zadaniu było widać czasami kolejność działań bardzo wpływa na wyniki działań.

6.1 Rozwiazanie

Funkcje które napisałem sa bardzo prosto napisane, na koniec skryptu zostaja one porównane dla pierwszych 10 liczb a nastepnie dla wielokrotności 20 aż do 100.

```
# Funkcja podana w zadaniu przyjmuj ca x jako liczb
f(x) = sqrt(x^2 + 1.0) - 1.0

# Funkcja podana w zadaniu przyjmuj ca x jako liczb
g(x) = x^2 / (sqrt(x^2 + 1.0) + 1.0)

# Ta cz      pokazuje r   nice pomiedzy warto ciami funkcji wy ej
    zdefiniowanymi
for i in 1:10
    println("f(8^-$i)  =   ", f(Float64(8)^-i))
    println("g(8^-$i)  =   ", g(Float64(8)^-i), "\n")
end
for i in [20, 40, 60, 80, 100]
    println("f(8^-$i)  =   ", f(Float64(8)^-i))
    println("g(8^-$i)  =   ", g(Float64(8)^-i), "\n")
end
```

Listing 11: Porównanie f(x) oraz g(x)

6.2 Wyniki oraz ich interpretacja

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^{-40}	0.0	2.8298997121333476e-73
8^{-60}	0.0	2.1289799200040754e-109
8^{-80}	0.0	1.6016664761464807e-145
8^{-100}	0.0	1.204959932551442e-181

Dla wiekszych wartości x widać poodbieństwa w wynikach, nie sa one takie same ale sa na tyle podobne że cieżko byłoby powiedzieć które sa blizej prawdy. Patrzac na mniejsze x widać już, że funkcja f(x) nie daje sobie rady z tak małymi wartościami i w porównaniu do g(x) szybko traci wiarygodność zwracajac wartości 0.0. Dzieje sie tak ze wzgledu na odejmowanie 1 od pierwiastka. Funkcja g(x) omija ten problem, dzieki temu zwraca wiarygodne wyniki jeszcze dużo dłużej niż f(x).

6.3 Wnioski

Kolejność działań potrafi sprawić że wyniki staja sie szybko dalekie od prawdy. Ale z drugiej strony możemy skonstruować działania z taka kolejnościa działań, które beda z duża dokładnościa podawały nam wyniki.

7 Przybliżanie wartości pochodnej

W tym zadaniu naszym celem jest porównanie przybliżoncyh wartości pochodznej:

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

z prawdziwymi wartościami pochodnej dla $h = 2^{-n}(n = 0, 1, 2, ..., 54)$ Funkcja któej pochodna mamy policzyć jest:

$$f(x) = sin(x) + cos(3x)$$

7.1 Rozwiazanie

Rozwiazanie zawiera zaimplementowane funkcje podane w treści zadania. Funckja oblicczajaca prawdziwa wartość zpochodnej w punkcie x została policzona przeze mnie.

$$\frac{d}{dx}(sin(x) + cos(3x)) = cos(x) - 3sin(3x)$$

```
# Funckja oblicza wartosc funkcji podanej w zadaniu w punkcie x
f(x) = sin(x) + cos(3x)
# Funckja oblicza wartosc pochodnej funkcji podanej w zadaniu w punkcie x
dfdx(x) = cos(x) - 3sin(3x)
# Aproksymacja pochodnej funkcji podanej w parametrze f w punkcie x i z
    precyzj  h
dfdx_approx(x, h, f) = (f(x + h) - f(x)) / h

# Prawdziwa wartosc pochodnej funckcji f w punkcie x = 1
actual = dfdx(1)


for n in 0:54
    println("\nh = 2^-$n:")
    println("~f'(1)             = ", dfdx_approx(1.0, Float64(2)^-n, f))
    println(" f'(1)             = ", actual)
    println("|f'(1) - ~f'(1)|   = ", abs(dfdx_approx(1.0, Float64(2)^-n, f) -
        actual))
end
```

Listing 12: Obliczanie przyblizonej pochodnej

7.2 Wyniki oraz ich interpretacja

Prawdziwy wynik pochodnej w punkcie $x_0 = 1$ wynosi:

$$f'(x_0) = 0.11694228168853815$$

h	$h+1$	$f'(x_0)$	$\lvert f'(x_0) - f'(x_0)\rvert$
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^{-7}	1.0078125	0.1484913953710958	0.03154911368255764
2^{-8}	1.00390625	0.1327091142805159	0.015766832591977753
2^{-9}	1.001953125	0.1248236929407085	0.007881411252170345
2^{-10}	1.0009765625	0.12088247681106168	0.0039401951225235265
2^{-11}	1.00048828125	0.11891225046883847	0.001969968780300313
2^{-12}	1.000244140625	0.11792723373901026	0.0009849520504721099
2^{-13}	1.0001220703125	0.11743474961076572	0.0004924679222275685
2^{-14}	1.00006103515625	0.11718851362093119	0.0002462319323930373
2^{-15}	1.000030517578125	0.11706539714577957	0.00012311545724141837
2^{-16}	1.0000152587890625	0.11700383928837255	6.155759983439424e-5
2^{-17}	1.0000076293945312	0.11697306045971345	3.077877117529937e-5
2^{-18}	1.0000038146972656	0.11695767106721178	1.5389378673624776e-5
2^{-19}	1.0000019073486328	0.11694997636368498	7.694675146829866e-6
2^{-20}	1.0000009536743164	0.11694612901192158	3.8473233834324105e-6

2^{-20}	1.0000009536743164	0.11694612901192158	3.8473233834324105e-6
2^{-21}	1.0000004768371582	0.1169442052487284	1.9235601902423127e-6
2^{-22}	1.000000238418579	0.11694324295967817	9.612711400208696e-7
2^{-23}	1.0000001192092896	0.11694276239722967	4.807086915192826e-7
2^{-24}	1.0000000596046448	0.11694252118468285	2.394961446938737e-7
2^{-25}	1.0000000298023224	0.116942398250103	1.1656156484463054e-7
2^{-26}	1.0000000149011612	0.11694233864545822	5.6956920069239914e-8
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^{-30}	1.0000000009313226	0.11694216728210449	1.1440643366000813e-7
2^{-31}	1.0000000004656613	0.11694216728210449	1.1440643366000813e-7
2^{-32}	1.0000000002328306	0.11694192886352539	3.5282501276157063e-7
2^{-33}	1.0000000001164153	0.11694145202636719	8.296621709646956e-7
2^{-34}	1.0000000000582077	0.11694145202636719	8.296621709646956e-7
2^{-35}	1.0000000000291038	0.11693954467773438	2.7370108037771956e-6
2^{-36}	1.000000000014552	0.116943359375	1.0776864618478044e-6
2^{-37}	1.000000000007276	0.1169281005859375	1.4181102600652196e-5
2^{-38}	1.000000000003638	0.116943359375	1.0776864618478044e-6
2^{-39}	1.000000000001819	0.11688232421875	5.9957469788152196e-5
2^{-40}	1.0000000000009095	0.1168212890625	0.0001209926260381522

2^{-41}	1.0000000000004547	0.116943359375	1.0776864618478044e-6
2^{-42}	1.0000000000002274	0.11669921875	0.0002430629385381522
2^{-43}	1.0000000000001137	0.1162109375	0.0007313441885381522
2^{-44}	1.0000000000000568	0.1171875	0.0002452183114618478
2^{-45}	1.0000000000000284	0.11328125	0.003661031688538152
2^{-46}	1.0000000000000142	0.109375	0.007567281688538152
2^{-47}	1.000000000000007	0.109375	0.007567281688538152
2^{-48}	1.0000000000000036	0.09375	0.023192281688538152
2^{-49}	1.0000000000000018	0.125	0.008057718311461848
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

Po wynikach widać że na poczatek dokładność przybliżenia rośnie a nastepnie zaczyna maleć. Najlepsze przybliżenie daje $h = 2^{-28}$. Wynika to z błedów zwiazanych z wartościami h coraz bliższymi do 0.

7.3 Wnioski

Przy prowadzeniu obliczeń przybliżajacych wynik najlepiej nie korzystać z liczb tak bliskich do zera. Istnieja liczby które wbrew intucji beda lepiej przybliżać niż te bliskie zeru.