Obliczenia Naukowe - Laboratorium 1

Michał Waluś 279695

Październik 2025

1 Zadanie 1

1.1 Epsilon maszynowy

Opis problemu: Używając języka Julia, iteracyjne wyznaczenie epsilona maszynowego *macheps* (najmniejszej liczby większej od 1 reprezentowalnej w arytmetyce zmiennopozycyjnej) dla typów `Float16`, `Float32` i `Float64`, oraz porównanie wyznaczonych wartości z wartościami zwracanymi przez funkcję wbudowaną `eps` oraz stałymi zawartymi w pliku `float.h` dla języka C.

Rozwiązanie: Ustalamy liczbę $x = 1.0$ i iteracyjnie dzielimy ją przez 2, dopóki wartość wyrażenia $1.0 + x$ nie będzie równa 1.0. Epsilon maszynowy to dwukrotność liczby x otrzymanej w ten sposób.

Wyniki:
Wartości zwrócone przez program w języku Julia:

```
Float16
Calculated macheps: 0.000977
1 + cal. macheps: 1.001
Macheps: 0.000977

Float32
Calculated macheps: 1.1920929e-7
1 + cal. macheps: 1.0000001
Macheps: 1.1920929e-7

Float64
Calculated macheps: 2.220446049250313e-16
1 + cal. macheps: 1.0000000000000002
Macheps: 2.220446049250313e-16
```

Gdzie "Calculated macheps"to wartość epsilona maszynowego wyznaczona iteracyjnie, a "macheps"to wartość zwrócona przez funkcję `eps()`.

Wartości zawarte w pliku `float.h`:

```
Float32 macheps: 1.192093e-07
Float64 macheps: 2.220446e-16
```

Wnioski: Wyniki wyznaczone iteracyjnie tą metodą są zgodne zarówno z wartościami zwracanymi przez funkcję wbudowaną, jak i ze stałymi z języka C, co dodatkowo potwierdza zgodność typów zmiennopozycyjnych ze standardem IEEE 754.

Dodatkowo, widzimy, że precyzja typu `Float64` jest ponad dwukrotnie większe niż typu `Float32` (i analogicznie z `Float32` w stosunku do `Float16`), więc możemy zaobserwować, że stosunek długości mantysy do liczby bitów typu jest większy dla bardziej precyzyjnych typów.

Otrzymany wyniki pokazują, że dla wartości bliskich 1.0 typ **Float16** umożliwia dość dokładne obliczenia dla wartości o 4 cyfrach znaczących, za to dla typu `Float32` jest to już 8 cyfr znaczących, a dla `Float64` aż 16, co jest znacząco większą precyzją niż mają niemalże wszystkie rzeczywiste dane, a przy obliczeniach z mniej dokładnych pomiarów (2-3 liczby znaczące), nawet typ `Float16` jest wystarczający.

Ponieważ wartość $1.0 + macheps$ jest reprezentowana dokładnie, a nie z zaokrągleniem, jest ona równa dwukrotności *precyzji arytmetyki*.

1.2 Eta

Opis problemu: Używając języka Julia, iteracyjne wyznaczenie liczby maszynowej *eta* - najmniejszej liczby dodatniej, której dokładna reprezentacja jest możliwa dla typów zmiennopozycyjnych `Float16`, `Float32` i `Float64` oraz porównanie otrzymanego wyniku z wartościami zwracanymi przez funkcję `nextfloat()`.

Rozwiązanie: Zaczynając od wartości $x = 1.0$, iteracyjnie dzielimy x przez 2, dopóki wartość wyrażenia $x/2$ nie będzie równa 0. Otrzymana w ten sposób wartość x to nasza *eta*.

Wyniki:
Wartości zwrócone przez program w języku Julia:

```
Float16
Calculated eta: 6.0e-8
Eta: 6.0e-8

Float32
Calculated eta: 1.0e-45
Eta: 1.0e-45

Float64
Calculated eta: 5.0e-324
Eta: 5.0e-324
```

Gdzie "Calculated eta"to wartość wyznaczona iteracyjnie, a "Eta"to wartość zwrócona przez funkcję `nextfloat()` dla wartości 0.0 odpowiedniego typu.

Wnioski: Wartości wyznaczone tą metodą są zgodne z wartościami zwracanymi przez funkcję wbudowaną. Dla kolejnych typów liczba *eta* maleje bardzo szybko, co obrazuje bardzo duży wzrost precyzji obliczeń dla większych typów - o ile wartości rzędu 10^{-8} można napotkać relatywnie często dla rzeczywistych danych, tak liczby mniejsze od 10^{-45} są bardzo rzadkie, a mniejsze od 10^{-300} nie pojawiają się prawie nigdy, co obrazuje, że typ `Float32` nadaje się do większości obliczeń na realnych danych o bardzo małych wartościach, a przy użyciu typu `Float64` precyzji może zabraknąć tylko dla danych specjalnie dobranych w tym celu.

Ponieważ *eta* jest równa najmniejszej dodatnej liczbie mającej dokładną reprezentację w arytmetyce zmiennopozycyjnej, jest ona równa liczbie MIN_{sub}.

1.3 floatmin

Opis problemu: Sprawdzenie wartości zwracanych przez funkcję wbudowaną `floatmin()` dla typów `Float32` oraz `Float64` w języku Julia i porównanie ich z liczbą MIN_{nor}.

Rozwiązanie: Wyznaczamy wartości MIN_{nor} dla liczby zmiennopozycyjnych 32- i 64-bitowych zgodnych z IEEE 754.

$$32 \text{ bit: } 2^{c_{min}} = 2^{-126} \approx 1.17549 \cdot 10^{-38}$$

$$64 \text{ bit: } 2^{c_{min}} = 2^{-1022} \approx 2.22507 \cdot 10^{-308}$$

Wyniki:

Wyniki zwrócone przez program w języku Julia:

```
Float32 floatmin: 1.1754944e-38

Float64 floatmin: 2.2250738585072014e-308
```

Wnioski: Wartości zwracane przez funkcję `floatmin()` są równe liczbie MIN_{nor}, czyli funkcja ta nie zwraca najmniejszej dodatniej wartości możliwej do zapisania przy użyciu typu zmiennopozycyjnego, w przeciwieństwie do tego co mogliby oczekiwać użytkownicy.

1.4 MAX

Opis problemu: Używając języka Julia iteracyjnie wyznaczyć wartość liczby MAX dla typów zmiennopozycyjnych `Float16`, `Float32` i `Float64`, porównanie otrzymanych wyników dla tych zwracanych przez funkcję `floatmax()` oraz z wartościami zawartymi w pliku nagłówkowym `float.h` dla języka C.

Rozwiąznie: Najpierw wyznaczamy iteracyjnie najmniejszą taką liczbę α, że $1.0 - \alpha \neq 1.0$, robimy to w sposób podobny do wyznaczania *macheps* w 1.1, czyli zaczynając od $\alpha = 1.0$ dzielimy ją przez 2, dopóki nie zajdzie $1.0 - \frac{\alpha}{2} - 1.0$. Następnie wyliczamy $x = 1.0 - \alpha$, otrzymując liczbę której wszystkie bity mantysy to jedynki. Finalnie iteracyjnie podwajamy x dopóki funkcja `isinf()` nie zwróci wartości `true` dla $2x$.

Wyniki:

Wyniki zwrócone przez program w języku Julia:

```
Calculated Float16 max: 6.55e4
Float16 max: 6.55e4

Calculated Float32 max: 3.4028235e38
Float32 max: 3.4028235e38

Calculated Float64 max: 1.7976931348623157e308
Float64 max: 1.7976931348623157e308
```

Gdzie "Calculated [float] max"to obliczona wartość, a "[float] max"to wartość zwrócona przez funkcję `floatmax()`.

Wartości z pliku `float.h`:

```
Float32 max: 3.402823e+38
Float64 max: 1.797693e+308
```

Wnioski: Obliczona wartość jest zgodna z wartością zwracaną przez funkcję i zawartą w języku C, co jest kolejnym dowodem na zgodność implementacji typów w języku Julia ze standardem IEEE 754.

Wartość MAX dla 16-bitowych liczb jest relatywnie mała i może zostać przekroczona dla wielu rzeczywistych danych. Dla `Float32` ta wartość jest już duża większa i dla większości obliczeń powinna być wystarczająco, za to wartość 64-bitowego MAX jest zdecydowanie większa od nawet liczby atomów we Wszechświecie i nawet dla ekstremalnie dużych rzeczywistych danych powinna zachowywać wysoką precyzję.

1.5 Wnioski nt. arytmetyki `float`

Typ `Float16` pozwala na zapisywanie liczb do 4 cyfr znaczących i w relatywnie niewielkim zakresie od około $6 \cdot 10^{-8}$ do $6,5 \cdot 10^{4}$ i powinnien być tylko używany, gdy bardzo nam zależy na oszczędzaniu pamięci, a dane są odpowiedniej wielkości.

`Float32` pozwala już na zapisanie liczb z 8 cyframi znaczącymi o wielkościach od 10^{-45} do 10^{38}, co pozwala na stosunkowo dokładne obliczenia na liczbach z przedziału od 10^{-30} do 10^{30}, co powinno być wystarczające do większości zastosowań.

W kontraście do tego, typ `Float64`, pozwala na obliczenia z nawet 16 cyframi precyzji na przedziale 10^{-300} do 10^{300} i powinien być wystarczający do niemalże wszystkich rzeczywistych zastosowań.

2 Zadanie 2

Opis problemu: Sprawdzenie poprawności metody Kahana na obliczanie epsilona maszynowego, za pomocą wyrażenia $3(\frac{4}{3} - 1) - 1$ za pomocą języka Julia dla typów `Float16`, `Float32` oraz `Float64`.

Roziązanie: Obliczenie wyrażenia $3(\frac{4}{3} - 1) - 1$.

Wyniki:

Wyniki zwrócone przez program:

```
Float16
Kahan's method: -0.000977
Macheps: 0.000977

Float32
Kahan's method: 1.1920929e-7
Macheps: 1.1920929e-7

Float64
Kahan's method: -2.220446049250313e-16
Macheps: 2.220446049250313e-16
```

Gdzie "Kahan's method"to wartość wyliczona z ww. wyrażenia, a "Macheps"to rzeczywista wartość epsilona maszynowego (patrz 1.1).

Wnioski: Metoda Kahana daje wynik prawdziwy pod względem wartości, ale dla liczb 16- i 64-bitowych z niewłaściwym znakiem. Oznacza to, że w zależności od liczby bitów mantysy $3(\frac{4}{3} - 1)$ może być równe $1.0 + \epsilon$ lub $1.0 - \epsilon$, gdzie ϵ to epsilon maszynowy.

3 Zadanie 3

Opis problemu: Używając języka Julia, sprawdzenie, że liczby zmiennopozycyjne typu `Float64` są równomiernie rozmieszczone w przedziale $[1, 2]$ z krokiem $\delta = 2^{-52}$, a także sprawdzenie jak

rozmieszone są liczby w przedziałach $[0.5, 1]$ i $[2, 4]$.

Rozwiązanie: Tworzymy funkcję `nextFloat()` wyznaczającą kolejną liczbę zmiennopozycyjną (tak jak w 1.1) oraz funkcję `prevFloat()` wyznaczającą poprzednią liczbę pozycyjną (tak jak w 1.4). Z pomocą tych funkcji i funkcji wbudowanej `bitstring()` możemy sprawdzić wielkość kroku oraz zapis w formacie binarnym ostatniej liczby mającej tą samą cechę, co pierwsza liczba w przedziale.

Dodatkowo używając funkcji, która zaczyna od `x = nextFloat(start)` (gdzie start to początek przedziału) i `count = 2`, iteracyjnie wykonując $x = 2 \cdot x - start$ (czyli podwajając wartość części ułamkowej mantysy), dopóki $x < 2$, jednocześnie podwajając *count*, możemy zliczyć ilość liczb zmienno przecinkowych w danym przedziale.

Obie te metody zakładają, że dla każdego ciągu 52 bitów, istnieje liczba zmiennopozycyjna o takiej części ułamkowej mantysy i jest właściwie interpretowana, ale sprawdzenie tego na standardowym komputrze wymagałoby kilku miesięcy obliczeń.

Wyniki:

Wyniki zwrócone przez program w języku Julia:

```
Representation of 1.0:
0011111111110000000000000000000000000000000000000000000000000000
Representation of nextFloat(1.0):
0011111111110000000000000000000000000000000000000000000000000001
Step: -52.0
Representation of prevFloat(2.0):
0011111111111111111111111111111111111111111111111111111111111111
Representation of 2.0:
0100000000000000000000000000000000000000000000000000000000000000
Step: -52.0

Number of Float64 numbers in [0.5, 1.0): 4503599627370496, log2: 52.0
Representation of 0.5:
0011111111100000000000000000000000000000000000000000000000000000
Representation of nextFloat(0.5):
0011111111100000000000000000000000000000000000000000000000000001
Step: -53.0
Representation of prevFloat(1.0):
0011111111101111111111111111111111111111111111111111111111111111
Representation of 1.0:
0011111111110000000000000000000000000000000000000000000000000000
Step: -53.0

Number of Float64 numbers in [2.0, 4.0): 4503599627370496, log2: 52.0
Representation of 2.0:
0100000000000000000000000000000000000000000000000000000000000000
Representation of nextFloat(2.0):
0100000000000000000000000000000000000000000000000000000000000001
Step: -51.0
Representation of prevFloat(4.0):
0100000000011111111111111111111111111111111111111111111111111111
Representation of 4.0:
0100000000010000000000000000000000000000000000000000000000000000
Step: -51.0
```

Gdzie "log2"to logarytm o podstawie 2 z ilości liczb, "Representation"to liczba przedstawiona w formie binarnej, a "Step"oznacza logarytm o podstawie 2 z różnicy pomiędzy dwiema poprzednimi liczbami.

Wnioski:

W obrębie każdego z tych przedziałów znajduje się dokładnie 2^{52} liczb (zakładając przedział prawostronnie otwarty), gdzie część ułamkowa mantysy pierwszej składa się z samych 0, a ostatniej z samych 1. Zarówno na początku jak i na końcu przedziału pozostaje taki sam.

Porównanie uzyskanych wyników jest przedstawione w Tabeli 1, gdzie postać ogólna, to forma w jakiej może zostać zapisana każda liczba z przedziału, dla δ takiej samej jak w drugiej kolumnie.

Przedział	Krok (δ)	Postać ogólna
$[0.5, 1.0]$	2^{-53}	$0.5 + k\delta,\ k = 0, 1, \ldots, 2^{52}$
$[1.0, 2.0]$	2^{-52}	$1.0 + k\delta,\ k = 0, 1, \ldots, 2^{52}$
$[2.0, 4.0]$	2^{-51}	$2.0 + k\delta,\ k = 0, 1, \ldots, 2^{52}$

Tabela 1: Porównanie uzyskanych wyników dla zadanych przedziałów

4 Zadanie 4

Opis problemu: Znalezienie eksperymentalnie w języku Julia, najmniejszą liczbę x typu `Float64`, taką że $1 < x < 2$ i $x \cdot (\frac{1}{x}) \neq 1$

Rozwiązanie: Iterujemy po kolejnych liczbach zmiennopozycyjnych, zaczynając od 1.0, dopóki nie znajdziemy najmniejszej spełniającej.

Wyniki:

Wyniki zwrócone przez program:

```
Smallest x such that x * (1/x) != 1: 1.000000057228997
Value: 0.9999999999999999
```

Gdzie value to obliczona wartość $x \cdot (1/x)$

Wnioski: Wyrażenia, które matematycznie się redukują, nie zawsze będą miały tą własność gdy wyliczone w kodzie, zatem przed wyliczaniem powinniśmy maksymalnie uprościć wszelkie wyrażenia algebraiczne.

5 Zadanie 5

Opis problemu: Używając języka Julia i typów `Float32` i `Float64`, obliczenie iloczynu skalarnego wektorów x i y na cztery sposoby:

1. "w przód od najmniejszego do największego indeksu
2. "w tył w odwrotnej kolejności niż "w przód"
3. od największego do najmniejszego, dodając dodatnie i ujemne składniki osobno
4. odwrotnie niż w 3.

oraz porównanie uzyskanych wartości z rzeczywistą wartością: $-1.00657107000000 \cdot 10^{-11}$

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

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

Rozwiązanie: Dla 1. i 2. policzenie sumy z użyciem pętli, dla 3. i 4. wyliczenie najpierw $x_i y_i$ dla wszystkich współrzędnych, podzielenie na dwie tablice, posortowanie ich (odpowiednio, w zależności od punktu), obliczenie sum tablic i dodanie ich.

Otrzymane wyniki:

Wyniki zwrócone przez program:

```
Float32 forward: -0.4999443
Float32 backward: -0.4543457
Float32 from biggest: -0.5
Float32 from smallest: -0.5

Float64 forward: 1.0251881368296672e-10
Float64 backward: -1.5643308870494366e-10
Float64 from biggest: 0.0
Float64 from smallest: 0.0
```

Dla `Float32`, wszystkie wyniki są blisko -0.5, co jest około 10 rzędów wielkości większe niż prawdziwa wartość.

Dla `Float64` otrzymany wynik w metodzie 1 ma przeciwny znak, a zarówno w metodach 1. i 2. otrzymany wynik różni się o rząd wielkości. Dla metod 3. i 4. program zwrócił 0.

Wnioski: Wartość iloczynu skalarnego jest znacznie mniejsza niż wartości składowych wektorów, w związku z czym tracimy precyzję odejmując liczby mające podobne wartości. Lepszy wynik otrzymalibyśmy, gdybyśmy dodawali składniki $x_i y_i$ w kolejności od największych względem wartości, wtedy mniejsze składniki byłyby bardziej zauważalne.

6 Zadanie 6

Opis problemu: Używając języka Julia, dla kolejnych wartości $x = 8^{-1}, 8^{-2}, 8^{-3}, \ldots$ obliczenie wartości funkcji:

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

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

Rozwiązanie: Wyliczenie wartości funkcji dla pierwszych 180 wartości (liczba wyznaczona eksperymentalnie, dla której obie funkcje zwracają 0).

Wyniki: Wyniki zwrócone przez program:

```
n=1, f(x)=0.0077822185373186414, g(x)=0.0077822185373187065
n=2, f(x)=0.00012206286282867573, g(x)=0.00012206286282875901
n=3, f(x)=1.9073468138230965e-6, g(x)=1.907346813826566e-6
n=4, f(x)=2.9802321943606103e-8, g(x)=2.9802321943606116e-8
n=5, f(x)=4.656612873077393e-10, g(x)=4.6566128719931904e-10
n=6, f(x)=7.275957614183426e-12, g(x)=7.275957614156956e-12
n=7, f(x)=1.1368683772161603e-13, g(x)=1.1368683772160957e-13
```

```
n=8, f(x)=1.7763568394002505e-15, g(x)=1.7763568394002489e-15
n=9, f(x)=0.0, g(x)=2.7755575615628914e-17
n=10, f(x)=0.0, g(x)=4.336808689942018e-19
...
n=166, f(x)=0.0, g(x)=7.466108948025751e-301
n=167, f(x)=0.0, g(x)=1.1665795231290236e-302
n=168, f(x)=0.0, g(x)=1.8227805048890994e-304
n=169, f(x)=0.0, g(x)=2.848094538889218e-306
n=170, f(x)=0.0, g(x)=4.450147717014403e-308
n=171, f(x)=0.0, g(x)=6.953355807835e-310
n=172, f(x)=0.0, g(x)=1.086461844974e-311
n=173, f(x)=0.0, g(x)=1.69759663277e-313
n=174, f(x)=0.0, g(x)=2.65249474e-315
n=175, f(x)=0.0, g(x)=4.144523e-317
n=176, f(x)=0.0, g(x)=6.4758e-319
n=177, f(x)=0.0, g(x)=1.012e-320
n=178, f(x)=0.0, g(x)=1.6e-322
n=179, f(x)=0.0, g(x)=0.0
n=180, f(x)=0.0, g(x)=0.0
```

Gdzie $x = 8^{-n}$. Dla czytelności większość wyników ze środka została wycięta.

Dla funkcji $f(x)$ już dla $n = 9$ program zwraca 0, jest to spowodowane tym, że wartości $\sqrt{x^2+1}$ są bardzo bliskie 1 dla małych x i odejmując te dwie liczby od siebie tracimy precyzję. Dla kontrastu dodając te dwie liczby do siebie otrzymujemy coś bliskiego dwukrotności dowolnej z nich i tracimy niewiele precyzji, co pozwala funkcji $g(x)$ zwracać sensowne wyniki nawet w okolicach $n = 165$, a 0 dopiero, gdy wartość spadnie poniżej liczby maszynowej *eta* (patrz 1.2).

Wnioski: Formy wyrażeń algebraicznych, które są bardziej skomplikowane dla ludzi, jeśli dobrze dobrane, potrafią zwracać znacznie lepsze resultaty podczas obliczeń za pomocą komputera.

7 Zadanie 7

Opis problemu: Używając języka Julia i liczb typu `Float64`, przybliżenie wartości pochodnej funkcji $f(x) = \sin x + \cos 3x$ w punkcie $x_0 = 1$, używając wzoru

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

a następnie porównanie różnicy między wartością rzeczywistą, dla $h = 2^{-n}$, gdzie $n = 0, 1, 2, \ldots, 54$.

Rozwiązanie: Wyznaczamy dokładną wartość pochodnej:

$$f'(x_0) = \cos x - 3 \sin 3x$$

i używamy jej do obliczenia wartości dokładnej, którą porównujemy z wartością przybliżoną.

Wyniki:

Tabela 2: Wyniki zwrócone przez program

n	$\tilde{f}'(x_0)$	$f'(x_0)$	$\lvert\tilde{f}'(x_0) - f'(x_0)\rvert$
0	2.0179892252685967	0.11694228168853815	1.9010469435800585
1	1.8704413979316472	0.11694228168853815	1.753499116243109
2	1.1077870952342974	0.11694228168853815	0.9908448135457593
3	0.6232412792975817	0.11694228168853815	0.5062989976090435
4	0.3704000662035192	0.11694228168853815	0.253457784514981
5	0.24344307439754687	0.11694228168853815	0.1265007927090087
6	0.18009756330732785	0.11694228168853815	0.0631552816187897
7	0.1484913953710958	0.11694228168853815	0.03154911368255764
8	0.1327091142805159	0.11694228168853815	0.015766832591977753
9	0.1248236929407085	0.11694228168853815	0.007881411252170345
10	0.12088247681106168	0.11694228168853815	0.0039401951225235265
11	0.11891225046883847	0.11694228168853815	0.001969968780300313
12	0.11792723373901026	0.11694228168853815	0.0009849520504721099
13	0.11743474961076572	0.11694228168853815	0.0004924679222275685
14	0.11718851362093119	0.11694228168853815	0.0002462319323930373
15	0.11706539714577957	0.11694228168853815	0.00012311545724141837
16	0.11700383928837255	0.11694228168853815	6.155759983439424e-5
17	0.11697306045971345	0.11694228168853815	3.077877117529937e-5
18	0.11695767106721178	0.11694228168853815	1.5389378673624776e-5
19	0.11694997636368498	0.11694228168853815	7.694675146829866e-6
20	0.11694612901192158	0.11694228168853815	3.8473233834324105e-6
21	0.1169442052487284	0.11694228168853815	1.9235601902423127e-6
22	0.11694324295967817	0.11694228168853815	9.612711400208696e-7
23	0.11694276239722967	0.11694228168853815	4.807086915192826e-7
24	0.11694252118468285	0.11694228168853815	2.394961446938737e-7
25	0.116942398250103	0.11694228168853815	1.1656156484463054e-7
26	0.11694233864545822	0.11694228168853815	5.6956920069239914e-8
27	0.11694231629371643	0.11694228168853815	3.460517827846843e-8
28	0.11694228649139404	0.11694228168853815	4.802855890773117e-9
29	0.11694222688674927	0.11694228168853815	5.480178888461751e-8
30	0.11694216728210449	0.11694228168853815	1.1440643366000813e-7
31	0.11694216728210449	0.11694228168853815	1.1440643366000813e-7
32	0.11694192886352539	0.11694228168853815	3.5282501276157063e-7
33	0.11694145202636719	0.11694228168853815	8.296621709646956e-7
34	0.11694145202636719	0.11694228168853815	8.296621709646956e-7
35	0.11693954467773438	0.11694228168853815	2.7370108037771956e-6
36	0.116943359375	0.11694228168853815	1.0776864618478044e-6
37	0.1169281005859375	0.11694228168853815	1.4181102600652196e-5
38	0.116943359375	0.11694228168853815	1.0776864618478044e-6
39	0.11688232421875	0.11694228168853815	5.9957469788152196e-5
40	0.1168212890625	0.11694228168853815	0.0001209926260381522
41	0.116943359375	0.11694228168853815	1.0776864618478044e-6
42	0.11669921875	0.11694228168853815	0.0002430629385381522
43	0.1162109375	0.11694228168853815	0.0007313441885381522
44	0.1171875	0.11694228168853815	0.0002452183114618478
45	0.11328125	0.11694228168853815	0.003661031688538152
46	0.109375	0.11694228168853815	0.007567281688538152
47	0.109375	0.11694228168853815	0.007567281688538152

48	0.09375	0.11694228168853815	0.023192281688538152
49	0.125	0.11694228168853815	0.008057718311461848
50	0.0	0.11694228168853815	0.11694228168853815
51	0.0	0.11694228168853815	0.11694228168853815
52	-0.5	0.11694228168853815	0.6169422816885382
53	0.0	0.11694228168853815	0.11694228168853815
54	0.0	0.11694228168853815	0.11694228168853815

Dla $n = 28$ otrzymane wartości są najbardziej zbliżone do siebie, co spowodowane jest zmniejszaniem wartości h, co powoduje lepsze przybliżenie wartości pochodnej, jednak dla co raz większych n, wartość h zaczyna się zbliżać do epsilona maszynowego i wartość $1 + h$ oddala się od prawdziwej. Dodatkowo, anomalia jest widoczna dla $n = 52$, gdzie wartość przybliżenia wynosi -0.5.

Wnioski: Podczas pisania programu należy zwrócić uwagę, by wartości na których się operuje nie stały się zbyt małe w porównaniu do stałych lub innych zmiennych, co skutkuje zmniejszeniem dokładności przybliżenia, zamiast poprawienia.