Obliczenia naukowe. Sprawozdanie z listy 1.

Maciej Bazela 261743

21 października 2022

Spis treści

1 Zadanie 1 **2**
1.1 Opis problemu . . . 2
1.2 Rozwiązanie . . . 2
1.3 Wyniki . . . 2
1.4 Wnioski . . . 3

2 Zadanie 2 **3**
2.1 Opis problemu . . . 3
2.2 Rozwiązanie . . . 3
2.3 Wyniki . . . 3
2.4 Wnioski . . . 4

3 Zadanie 3 **4**
3.1 Opis problemu . . . 4
3.2 Rozwiązanie . . . 4
3.3 Wyniki . . . 5
3.4 Wnioski . . . 5

4 Zadanie 4 **5**
4.1 Opis problemu . . . 5
4.2 Rozwiązanie . . . 5
4.3 Wyniki . . . 6
4.4 Wnioski . . . 6

5 Zadanie 5 **6**
5.1 Opis problemu . . . 6
5.2 Rozwiązanie . . . 6
5.3 Wyniki . . . 7
5.4 Wnioski . . . 7

6 Zadanie 6 **7**
6.1 Opis problemu . . . 7
6.2 Rozwiązanie . . . 7
6.3 Wyniki . . . 8
6.4 Wnioski . . . 8

7 Zadanie 7 **8**
7.1 Opis problemu . . . 8
7.2 Rozwiązanie . . . 9
7.3 Wyniki . . . 9
7.4 Wnioski . . . 9

1 Zadanie 1

1.1 Opis problemu

Zadanie 1. polegało na wyznaczeniu poniższych liczb:

1. *macheps*: najmniejszej liczby, dla której $fl(1.0 + macheps) > 1.0$
2. *eta*: najmniejszej reprezentowanej liczby
3. MAX: największej reprezentowanej liczby

dla wszystkich dostępnych typów zmiennopozycyjnych w języku *Julia* (Float16, Float32, Float64).

Dodatkowo w treści zadania zostały zadane poniższe pytania:

1. Jaki związek ma liczba *macheps* z precyzją arytmetyki (oznaczaną na wykładzie przez ϵ)?
2. Jaki związek ma liczba eta z liczbą MIN_{sub} (zob. wykład lub raport)?
3. Co zwracają funkcje *floatmin(Float32)* i *floatmin(Float64)* i jaki jest związek zwracanych wartości z liczbą MIN_{nor} (zob. wykład lub raport)?

1.2 Rozwiązanie

Kod źródłowy rozwiązań do tego zadania znajduje się w pliku zad1.jl.

Iteracyjne wyznaczanie *macheps* oraz *eta* polegało na kolejno mnożeniu i dzieleniu przez dwa danej liczby startowej aż do osiągnięcia warunku stopu:

1. *macheps*: $1.0 + (\frac{macheps}{2}) = 1$
2. *eta*: $0.0 + (\frac{eta}{2}) = 0.0$

Wyznaczenie MAX polegało na znalezieniu liczby x o maksymalnej *cesze* dla danego typu zmiennopozycyjnego i dodawaniu $\frac{x}{2^k}$ dla $k = 1, 2, \ldots$, dopóki $x + \frac{x}{2^1} + \frac{x}{2^2} + \cdots < \text{inf}$, gdzie inf odpowiada oznaczeniu nieskończoności w *IEEE 754*.

1.3 Wyniki

Poniżej znajdują się tabele z wynikami obliczeń *macheps*, *eta*, MAX dla zadanych typów zmiennopozycyjnych:

Typ float	**Moja implementacja**	**eps (Julia)**	**float.h**
Float16	0.000977	0.000977	brak
Float32	$1.1920929e-7$	$1.1920929e-7$	$1.19209290e-7F$
Float64	$2.220446049250313e-16$	$2.220446049250313e-16$	$2.2204460492503131e-16$

Tabela 1: Wartości *macheps*, *eps* i odpowiadające wartości z *float.h*.

Typ float	**Moja implementacja**	**nextfloat (Julia)**
Float16	$6.0e-8$	$6.0e-8$
Float32	$1.0e-45$	$1.0e-45$
Float64	$5.0e-324$	$5.0e-324$

Tabela 2: Wartości *eta* i *nextfloat()*.

Typ float	Moja implementacja	floatmax (Julia)	float.h
Float16	$6.55e4$	$6.55e4$	brak
Float32	$3.4028235e38$	$3.4028235e38$	$3.402823e+38$
Float64	$1.7976931348623157e308$	$1.7976931348623157e308$	$1.79769e+308$

Tabela 3: Wartości *MAX*, *floatmax()* i odpowiadających wartości z *float.h*.

1.4 Wnioski

Wyliczone przeze mnie wartości pokrywają się z liczbami zwracanymi przez implementacje ze standardowej biblioteki *Julii (Base)* oraz z definicjami z plików nagłówkowego *float.h*.
Tak naprawdę w tym zadaniu staraliśmy się otrzymać specyficzną reprezentację w standardzie *IEEE 754*, więc skoro dostałem takie same wyniki, to znaczy, że zaimplementowany przeze mnie algorytm działa poprawnie.

W standardzie *IEEE 754* wyróżniamy liczby znormalizowane ($\pm MIN_{nor}$) i zdenormalizowane ($\pm MIN_{sub}$). Liczby zdenormalizowane to takie, których wszystkie bity *cechy* są ustawione na 0, a *mantysa* zawiera chociaż jedną jedynkę. Wyliczana przez nas liczba *eta* (w *Julii*: *nextfloat(0.0)*) jest dokładnie najmniejszą taką liczbą (dodatnią).
Liczby zwracane przez *floatmin(FloatX)* są najmniejszymi liczbami znormalizowanymi typu *FloatX* (ich *cechy* mają chociaż jeden niezerowy bit).

Macheps jest górną granicą błędu względnego, powstałego w wyniku zaokrąglania w arytmetyce zmiennopozycyjnej. Jego wartość to dwukrotność precyzji arytmetyki (ϵ) wyznaczonej na wykładzie:

$$\epsilon = \frac{1}{2}\beta^{(1-t)}$$
$$macheps = \beta^{(1-t)}$$

Gdzie β - baza rozwinięcia, t - liczba cyfr mantysy.

2 Zadanie 2

2.1 Opis problemu

Zadanie 2. polegało na sprawdzeniu, czy wynik poniższego wyrażenia odpowiada *macheps* dla danego typu zmiennopozycyjnego:

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

2.2 Rozwiązanie

Kod źródłowy rozwiązań do tego zadania znajduje się w pliku zad2.jl.

W załączonym pliku źródłowym można znaleźć funkcję obliczającą powyższe wyrażenie (w funkcji *kahan()*) dla danego typu Float.
Obliczenie powyższego wyrażenia możnaby równie dobrze wykonać w *REPL Julii*, ale wolałem zaimplementować to zadanie w osobnym pliku dla czystej czytelności.

2.3 Wyniki

Typ float	kahan()	eps (Julia)
Float16	-0.000977	0.000977
Float32	$1.1920929e-7$	$1.1920929e-7$
Float64	$-2.220446049250313e-16$	$2.220446049250313e-16$

Tabela 4: Wartości wyrażenia $3 \cdot (\frac{4}{3}-1)+1$ i *eps()*.

2.4 Wnioski

Jak widać, dla *Float16* i *Float64* otrzymaliśmy wyniki, które różnią się znakiem od wartości zwracanej przez funkcję *eps()*.
Powodem jest rozwinięcie ułamka $\frac{4}{3}$ w systemie binarnym: $\frac{4}{3} = (1.(10))_2$.
Typ *Float16* i *Float64* mają parzystą ilość bitów mantysy (kolejno 10 i 52), natomiast *Float32* ma 23 bity.
Ostatnią cyfrą mantysy w *Float16* i *Float64* będzie 0, w *Float32* - 1. Zaokrąglenie nieskończonego ułamka $\frac{4}{3}$ spowoduje, że w *Float16* i *Float64* zaokrąglamy do najbliższej liczby ujemnej ($-macheps$), w *Float32* zaokrąglamy do liczby dodatniej ($+macheps$).

3 Zadanie 3

3.1 Opis problemu

Zadanie 3. polegało na sprawdzeniu, że w arytmetyce *double* (*Float64*) liczby zmiennopozycyjne są równomiernie rozmieszczone w przedziale $[1, 2]$ z krokiem $\delta = 2^{-52}$. Trzeba było także sprawdzić, rozmieszczenie liczb z przedziałów: $[\frac{1}{2}, 1]$ i $[2, 4]$.

3.2 Rozwiązanie

Na początku powinniśmy zauważyć, że podane przedziały są dobrane tak, że *cechy w kodzie z nadmiarem* (w reprezentacji *IEEE 754*) liczb granicznych różnią się o 1:

$$[\frac{1}{2}, 1] = [2^{-1}, 2^0]$$
$$[1, 2] = [2^0, 2^1]$$
$$[2, 4] = [2^1, 2^2]$$

W arytmetyce *Float64* dla ustalonego *bitu znaku* i *cechy* można zapisać 2^{52} różnych wartości, ponieważ mamy 52 bity mantysy, na których liczby mogą się od siebie różnić.
Ale skoro dla danego przedziału, który zawiera liczby o ustalonych *bitach znaku i cechy* możemy zapisać skończoną liczbę różnych liczb zmiennopozycyjnych, to znaczy, że dystans pomiędzy kolejnymi liczbami z zadanego przedziału będzie różny. Wyniesie on dokładnie $2^{c_{start}} \cdot 2^{-52}$, gdzie c_{start} to *cecha w kodzie z nadmiarem* dla początku zadanego przedziału w arytmetyce *double*.

Aby wyliczyć dokładnie jaki jest dystans pomiędzy liczbami w przedziale, który spełnia wyżej opisane warunki, stworzyłem prostą funkcję o nazwie $floating\_distance$:

Algorytm 1: Algorytm do wyliczania dystansu pomiędzy liczbami w arytmetyce *double* z zadanego przedziału

```
function floating_distance (_start, _end);
Input  : Przedział (_start, _end)
Output: Różnica pomiędzy kolejnymi dwiema liczbami w arytmetyce double w przedziale
        [_start, _end]
if (1 + exponent(_start)) ≠ exponent(_end) then
  | return 'Cechy _start i _end różnią się o więcej niż 1. Przedział może być nierówny';
else
  | return fl(2)^(exponent(_start)) · fl(2)^(-52);
end
```

Funkcja *exponent* w $Julli(Base)$ zwraca *cechę w kodzie z nadmiarem* dla danej znormalizowanej liczby zmiennopozycyjnej.

3.3 Wyniki

Przedział $[start, end]$	$\boldsymbol{floating\_distance(start, end)}$
$[\frac{1}{2}, 1]$	1.1102230246251565e-16
$[1, 2]$	2.220446049250313e-16
$[2, 4]$	4.440892098500626e-16

Tabela 5: Wyniki wywołania funkcji spread dla danego przedziału.

3.4 Wnioski

W standardzie *IEEE 754* możemy zapisać skończoną liczbę liczb zmiennopozycyjnych. Zapisując w nim bardzo małe liczby musimy mieć świadomość, że będą one zaokrąglone do wartości, które mogą być reprezentowane przez komputer.

Im mniejsza *cecha* danej liczby w *IEEE 754* tym mniejsza odległość pomiędzy kolejnymi liczbami zmiennopozycyjnymi (większa precyzja). Im większa *cecha* tym większe odległości (mniejsza precyzja).

4 Zadanie 4

4.1 Opis problemu

W zadaniu 4. mamy znaleźć taką liczbę (w arytmetyce *Float64*) x:

1. $x \in (1, 2) : fl(x \cdot fl(\frac{1}{x})) \neq fl(1)$
2. najmniejszą liczbę, taką że $fl(x \cdot fl(\frac{1}{x})) \neq fl(1)$

4.2 Rozwiązanie

Kod źródłowy rozwiązań do tego zadania znajduje się w pliku zad4.jl.

Znalezienie takich liczb oparłem na algorytmie brute-force. Moja funkcja przyjmuje argumenty *_start* i *limit*, które określają kolejno: od jakiej wartości rozpoczynamy iterowanie oraz jaki jest górny limit poszukiwań.
W pierwszym przypadku wywołujemy funkcję *smallest_not_equal_to_one* z argumentami *one(Float64)* i *Float64(2)*.
Aby znaleźć najmniejszą taką liczbę spełniającą warunki zadania, za *_start* przyjmujemy *zero(Float64)*, za *limit* nieskończoność w *IEEE 754* dla *Float64* (*Inf64*)

Algorytm przechodzi po kolejnych liczbach zmiennopozycyjnych (z wykorzystaniem funkcji z *Base Julii*: *nextfloat()*), aż do osiągnięcia warunku stopu $nextfloat(curr) \cdot \frac{fl(1)}{nextfloat(curr)} \neq fl(1)$.

Algorytm 2: Algorytm brute-force do wyznaczania liczby x z zadanego przedziału $(\_start, limit)$

function smallest_not_equal_to_one $(\_start, limit)$;
Input : Przedział $(\_start, limit)$
Output: Najmniejsza liczba x, taka że: $x \in (\_start, limit) : fl(x \cdot fl(\frac{1}{x})) \neq fl(1)$
$curr \leftarrow \_start$;
$next \leftarrow nextfloat(curr)$;
while $next \cdot \frac{fl(1)}{next} == fl(1)$ ***and*** $curr < limit$ **do**
    $curr = next$;
    $next = nextfloat(curr)$;
end
return $next$

4.3 Wyniki

Dla podanych warunków otrzymałem poniższe wyniki:

1. $x = 1.000000057228997$ ($x \cdot \frac{fl(1)}{x} = 0.9999999999999999$)

2. $x = 5.0e - 324$ ($x \cdot \frac{fl(1)}{x} = Inf$)

4.4 Wnioski

Zaokrąglenia w arytmetyce zmiennopozycyjnej mogą prowadzić do niespodziewanych dla człowieka wyników. Mimo, że dla ludzi banalne jest policzenie, że $x \cdot \frac{1}{x} = 1$, niezawsze otrzymamy taki wynik korzystając z typów *Float* na komputerze. Dlatego ważne jest, aby uważnie sprawdzać otrzymywane wyniki, szczególnie gdy porówunujemy liczby zmiennopozycyjne.

W treści zadania w podpunkcie 2. nie było określone czy liczba którą szukamy ma być znormalizowana, czy nie. Wynik który otrzymałem jest oczywiście najmniejszą możliwą liczbą zdenormalizowaną różną od $fl(0)$ (składa się z 63 zer i ostatniego bitu ustawionego na 1). Gdybyśmy chcieli znaleźć taką liczbę x, że nasz warunek nie równa się nieskończoności, trzeba lekko zmodyfikować algorytm.

5 Zadanie 5

5.1 Opis problemu

Zadanie 5. polega na policzeniu iloczynu skalarnego dwóch wektorów x i y:

$$x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$$
$$y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$$

korzystając z różnych algorytmów sumowania:

1. ‘w przód’

2. ‘w tył’

3. ‘od największego do najmniejszego’ (dodając dodatnie elementy w porządku od największego do najmniejszego, ujemne od najmniejszego do największego, na końcu dodając obie sumy częściowe)

4. ‘od najmniejszego do największego’ (dodając dodatnie elementy w porządku od najmniejszego do największego, ujemne od największego do najmniejszego, na końcu dodając obie sumy częściowe)

Obliczenia powinniśmy wykonać na typach *Float32* i *Float64* oraz porównać z dokładną wartością: $-1.00657107000000 \cdot 10^{-11}$.

5.2 Rozwiązanie

Kod źródłowy rozwiązań do tego zadania znajduje się w pliku zad3.jl.

Rozwiązanie tego zadania nie jest niczym innym niż zaimplementowaniem podanych algorytmów i policzeniu wartości dla danych typów *Float*.

5.3 Wyniki

Algorytm sumowania	Float32	Float64
W przód	-0.4999443	$1.0251881368296672e-10$
W tył	-0.4543457	$-1.5643308870494366e-10$
Od największego do najmniejszego	-0.5	0.0
Od najmniejszego do największego	-0.5	0.0

Tabela 6: Porównanie wyników iloczynu skalarnego x i y dla *Float32*, *Float64* i różnych algorytmów sumowania.

5.4 Wnioski

Sumowanie liczb strategią od największego elementu do najmniejszego lub na odwrót, powoduje największą utratę precyzji obliczeń.

Najlepiej zachował się algorytm sumowania 'w tył' dla typu *Float64*.

Drastyczne różnice w wynikach iloczynu skalarnego dla różnych typów i algorytmów sumowania pokazuje jak ważny jest dobór odpowiedniej strategii przy wykonywaniu obliczeń, których wynik może być bliski zeru. Przy wyliczaniu takich wartości warto jest używać liczb zmiennopozycyjnych podwójnej precyzji (*Float64*), mimo wolniejszego działania programu.

6 Zadanie 6

6.1 Opis problemu

Zadanie 6. polega na implementacji funkcji $f(x)$ i $g(x)$:

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

oraz wyliczeniu wartości obu funkcji dla $x = 8^{-k}, k = 1, 2, \ldots$, w artymetyce *Float64*.

6.2 Rozwiązanie

Kod źródłowy rozwiązań do tego zadania znajduje się w pliku zad3.jl.

W załączonym pliku *zad6.jl* znajduje się implementacja obu funkcji oraz wyliczenie ich wartości dla zakresu $[1, k]$. Liczbę k podaje się jako argument linii komend.

Przykładowe wywołanie programu *zad6.jl* dla $k = 180$:

```
$ julia zad6.jl 180
```

6.3 Wyniki

k	$f(8^{-k})$	$g(8^{-k})$
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	$3.2626522339992623e-55$
40	0.0	$2.8298997121333476e-73$
50	0.0	$2.4545467326488633e-91$
70	0.0	$1.8465957235571472e-127$
100	0.0	$1.204959932551442e-181$
160	0.0	$5.1306710016229703e-290$
175	0.0	$4.144523e-317$
176	0.0	$6.4758e-319$
177	0.0	$1.012e-320$
178	0.0	$1.6e-322$
179	0.0	0.0
180	0.0	0.0

Tabela 7: Wartości funkcji f i g dla danego 8^{-k}.

6.4 Wnioski

Funkcja f bardzo szybko zbiegła do $fl(0)$, natomiast funkcja g daje bardzo dokładne wyniki nawet dla bardzo małych argumentów.

Powodem niedokładnych wyników dla funkcji f jest odejmowanie bliskich sobie liczb zmiennopozycyjnych. Zaokrąglenie wyniku odejmowania sprawia, że wystarczy tylko 9 kroków, aby dojść do wartości $fl(0)$.

Funkcja g omija ten problem poprzez przekształcenie zadanej funkcji f. Pozwala to na dużo dokładniejsze obliczenia, ponieważ ani mianownik, ani licznik nie są liczbami bliskimi $fl(0)$.

7 Zadanie 7

7.1 Opis problemu

W zadaniu 7. mieliśmy zaimplementować funkcję wyliczającą przybliżenie wartości pochodnej $f(x_0)$ w punkcie x_0:

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

oraz wyliczyć wartości pochodnej funkcji $f(x) = \sin x + \cos 3x$ w punkcie $x_0 = 1$ i błędów $|f'(x_0) - \tilde{f}'(x_0)|$ dla $h = 2^{-n}, n = 0, 1, 2, \ldots, 54$ w arytmetyce *double* (*Float64*).

7.2 Rozwiązanie

Kod źródłowy rozwiązań do tego zadania znajduje się w pliku zad3.jl.

Dokładna pochodna dla funkcji $f(x) = \sin x + \cos 3x$ to $f'(x) = \cos x - 3 \sin 3x$.

Dla każdego h z zadanego zakresu i $x_0 = fl(1)$ wyliczyłem $x_0 + h$, przybliżoną wartość pochodnej z podanego wzoru oraz błąd $|f'(x_0) - \tilde{f}'(x_0)|$. Wyniki wydrukowałem do konsoli.

7.3 Wyniki

Dokładna wartość pochodnej f w punkcie x_0 wynosi: $f'(x_0) = 0.11694228168853815$
Tabela z wynikami znajduje się na nastepnej stronie.

7.4 Wnioski

Błąd względny $|f'(x_0) - \tilde{f}'(x_0)|$ maleje aż do $h = 2^{-28}$. Potem zaczyna on rosnąć. Jest to spowodowane tym, że dla coraz mniejszych h odejmujemy liczby coraz bliższe sobie $(f(x_0 + h) - f(x_0))$. Zachodzi tutaj takie same zjawisko jak w zadaniu 6. Gdy dochodzimy do wartości h mniejszych niż $h = 2^{-28}$ tracimy dokładność przez zaokrąglenie odejmowania w liczniku.

h	$x_0 + h$	$\tilde{f}'(x_0)$	$\lvert f'(x_0) - \tilde{f}'(x_0)\rvert$
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^{-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^{-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

Tabela 8: Wyliczone wartości $x_0 + h$ $(1 + h)$, przybliżenia pochodnej $f'(x_0)$ i błędu $|f'(x_0) - \tilde{f}'(x_0)|$ dla danego $h = 2^{-i}$.