Sprawozdanie
Obliczenia naukowe - lista 1

Kamil Król

Zadanie 1

MachEps

Epsilonem maszynowym macheps (ang. machine epsilon) nazywamy najmniejszą liczbę macheps większą od 0 taką, że fl(1.0 + macheps)>1.0.

W celu wyznaczenia metodą iteracyjną *macheps* zgodnego z powyższą definicją napisałem program, a jego wyniki zamieściłem w poniższej tabeli. Zgodnie z treścią zadania program uruchomiłem dla typów Float16, Float32 oraz Float64 i porównałem z wartościami zwracanymi przez funkcję *eps* dla każdego z typów.

Iteracyjne wyznaczanie macheps

	obliczony *macheps*	eps(type)	wartość z pliku float.h
Float16	0.000977	0.000977	xd
Float32	1.1920929e-7	1.1920929e-7	xd
Float64	2.220446049250313e-16	2.220446049250313e-16	xd

Okazało się, że wartości *macheps* wyznaczone przeze mnie są równe wartościom zwracanym przez wbudowaną w język Julia funkcją *eps*.

W treści zadania pojawia się pytanie: jaki związek ma liczba *macheps* z precyzją arytmetyki (oznaczaną na wykładzie przez ϵ)? W celu odpowiedzi na to pytanie przytoczę najpierw definicję precyzji arytmetyki - ϵ. Jest to największy błąd względny reprezentacji liczby jaki możemy popełnić i dla liczb reprezentowanych zgodnie ze standardem IEEE-754 wyraża się on wzorem: 2^{-t}. Podstawiając do wzoru dla arytmetyki Float32 mamy:

$$\epsilon = 2^{-24} = 0.5 \cdot 2^{-23} = \frac{1}{2} \cdot macheps$$

Wartość *macheps* dla Float32 w tabeli tj. 1.1920929e-7 jest zaokrąglona. Jej dokładna wartość wynosi: 1.1920928955078125e-7 co jest równe 2^{-23} (stąd równość). *Macheps* jest w komputerze przechowywany dokładnie. Wykonując to rozumowanie dla wszystkich typów widzimy zgodność i otrzymujemy zależność: $macheps = 2\epsilon$.

Eta

Kolejnym zadaniem jest wyznaczenie liczby *eta* takiej, że $eta > 0.0$ dla wszystkich typów zmiennopozycyjnych Float16, Float32, Float64. Wyniki napisanego przeze mnie programu, który iteracyjnie wyznacza te liczby, umieściłem w poniższej tabeli. Ponadto wartości otrzymanych liczb *eta* porównałem z wartościami zwracanymi przez funkcje: *nextfloat*(Float16(0.0)), *nextfloat*(Float32(0.0)), *nextfloat*(Float64(0.0)

Iteracyjne wyznaczanie eta

	obliczona *eta*	nextfloat(type(0))
Float16	6.0e-8	6.0e-8
Float32	1.0e-45	1.0e-45
Float64	5.0e-324	5.0e-324

Wartości obliczone przeze mnie okazały się takie same jak zwrócone przez funkcje wbudowane w język Julia. Kolejnym pytaniem jest: Jaki związek ma liczba *eta* z liczbą MIN_{sub}?
MIN_{sub} jest najmniejszą liczbą zdenormalizowaną (subnormalną), tzn. taką gdzie cecha liczby jest wypełniona zerami. Inaczej najmniejsza możliwa do przechowania w danym systemie liczba. Liczba *eta* jest równa liczbie MIN_{sub}, są one tożsame. Są to liczby tak małe, że nie da się ich pomniejszyć manipulując cechą.
Innym pytaniem z treści zadania jest: co zwracają funkcje *floatmin*(Float32) i *floatmin*(Float64) i jaki jest związek zwracanych wartości z liczbą MIN_{nor}?
Funkcje te zwracają najmniejsze liczby znormalizowane dla danego typu, a jest to dokładnie MIN_{nor}. Liczby znromalizowane to takie gdzie w mantysie zakładamy niepisaną jedynkę, tzn. wartość mantysy '0011...' oznacza '1.0011...' (inaczej niż w subnormalnych) i cecha nie jest zerem. Inaczej są to liczby, które można pomniejszyć zmniejszając wartość cechy.

Liczba MAX

Kolejnym zadaniem do zrobienia było wyznaczenie (iteracyjnie) liczby *MAX* dla wszystkich typów zmiennopozycyjnych Float16, Float32, Float64 i porównanie wyników z wartościami zwracanymi przez funkcje: *floatmax*(Float16), *floatmax*(Float32), *floatmax*(Float64) oraz z danymi zawartymi w pliku nagłówkowym float.h dowolnej instalacji języka C. Liczbę *MAX* interpretuję jako największą wartość jaką można przechować w danym typie zmiennoprzecinkowym. Przy wyznaczaniu tej wartości musiałem pamiętać aby mantysa była wypełniona jedynkami. By to uzyskać postanowiłem wziąć liczbę *zaraz przed* liczbą 2.0 czyli 2.0 - *macheps*. Ten rezultat mogłem uzyskać też biorąc liczbę *zaraz przed* 1.0, wtedy byłoby to $1.0 - \frac{macheps}{2}$.

Iteracyjne wyznaczanie liczby *MAX*

	Obliczony *MAX*	maxfloat(type)	wartość z pliku float.h
Float16	6.55e4	6.55e4	xd
Float32	3.4028235e38	3.4028235e38	xd
Float64	1.7976931348623157e308	1.7976931348623157e308	xd

Ponownie wartości wyznaczone przeze mnie okazały się takie same jak te wyznaczone przez funkcje z języka Julia.

Zadanie 2

Zadanie to dotyczyło sprawdzenia eksperymentalnie w języku Julia słuszności tezy Kahana dla wszystkich typów zmiennopozycyjnych Float16, Float32, Float64. Wyniki programu zamieściłem w poniższej tabeli:

	Obliczony eps wg. wzoru Kahana	Wartość funkcji eps(type)
Float16	-2.220446049250313e-16	2.220446049250313e-16
Float32	1.1920929e-7	1.1920929e-7
Float64	-2.220446049250313e-16	2.220446049250313e-16

Ponadto mój program sprawdzał, czy wartości bezwzględne otrzymanych eps są równe i okazało się że tak jest. Wnioskiem z doświadczenia jest to, że teza Kahana jest słuszna - macheps można obliczyć stosując zaproponowany przez niego wzór. W celu otrzymania macheps należy na wynik otrzymany ze wzoru $3(4/3 - 1) - 1$ nałożyć wartość bezwzględną.

Zadanie 3

Celem tego zadania było eksperymentalne sprawdzenie, że w arytmetyce Float64 liczby zmiennopozycyjne są równomiernie rozmieszczone w [1, 2] z krokiem $\delta = 2^{-52}$. Mój program zaczynał od liczby 1.0 i z każdą iteracją dodawał do niej liczbę δ. W celu zidentyfikowania czy wybrany krok jest poprawny drukowałem kolejne wartości liczb i obserwowałem jak się zmieniają. Rezultaty w tabeli poniżej.

	bitowa reprezentacja
$1.0 + 1\delta$	0—01111111111—0000000000000000000000000000000000000000000000000001
$1.0 + 2\delta$	0—01111111111—0000000000000000000000000000000000000000000000000010
$1.0 + 3\delta$	0—01111111111—0000000000000000000000000000000000000000000000000011
$1.0 + 4\delta$	0—01111111111—0000000000000000000000000000000000000000000000000100
$1.0 + 5\delta$	0—01111111111—0000000000000000000000000000000000000000000000000101
$1.0 + 6\delta$	0—01111111111—0000000000000000000000000000000000000000000000000110
$1.0 + 7\delta$	0—01111111111—0000000000000000000000000000000000000000000000000111
$1.0 + 8\delta$	0—01111111111—0000000000000000000000000000000000000000000000001000

...

Widać w kolejnych rekordach, że wybrany krok - δ jest poprawny (dla przedziału [1,2]), ponieważ wartości zmieniają się na najmniej znaczących bitach (na końcu) w sposób umożliwiający przejście przez wszystkie możliwe wartości mantysy.
Pojawia się pytanie: Jak rozmieszczone są liczby zmiennopozycyjne w przedziale $[\frac{1}{2}, 1]$, a jak w przedziale [2, 4] i jak mogą być przedstawione dla rozpatrywanego przedziału?
Liczbę δ dla przedziału $[\frac{1}{2}, 1]$ będziemy nazywać δ_1, a dla [2, 4] δ_4. Liczb we wszystkich trzech wspomianych przedziałach jest tyle samo (w tej reprezentacji). Np. między 1 i 2 jest tyle samo liczb co między 2 i 4, a tych jest tyle samo co w przedziale [4,8] itd. Granicami tych przedziałów są potęgi liczby 2. Jak można z tą wiedzą wyznaczyć δ dla przedziałów $[\frac{1}{2}, 1]$ i [2, 4]? Wiemy, że δ dla przedziału [1,2] wynosi $\delta = 2^{-52}$. Długość przedziału $[\frac{1}{2}, 1]$ jest 2 razy mniejsza od długości [1,2], więc naturalnym kandydatem dla liczby δ_1 (dla przedziału $[\frac{1}{2}, 1]$) wydaje się liczba δ podzielona przez 2. Zatem mamy $\delta_1 = 2^{-53}$.
Przedział [2, 4] ma długość dwa razy większą niż [1, 2], więc kandydatem na liczbę δ_4 będzie liczba 2 razy więszka od δ. W związku z tym mamy $\delta_4 = 2^{-51}$. Pozostaje te wartości eksperymentalnie sprawdzić. W poniższej tabeli znajdują się wyniki programu, który sprawdza liczby δ_1 i δ_4 w analogiczny sposób jak przy sprawdzaniu δ dla przedziału [1,2].

$\delta_1 = 2^{-53}$

	bitowa reprezentacja
$0.5 + 1\delta$	0—01111111110—0000000000000000000000000000000000000000000000000001
$0.5 + 2\delta$	0—01111111110—0000000000000000000000000000000000000000000000000010
$0.5 + 3\delta$	0—01111111110—0000000000000000000000000000000000000000000000000011
$0.5 + 4\delta$	0—01111111110—0000000000000000000000000000000000000000000000000100
$0.5 + 5\delta$	0—01111111110—0000000000000000000000000000000000000000000000000101
$0.5 + 6\delta$	0—01111111110—0000000000000000000000000000000000000000000000000110
$0.5 + 7\delta$	0—01111111110—0000000000000000000000000000000000000000000000000111
$0.5 + 8\delta$	0—01111111110—0000000000000000000000000000000000000000000000001000
$0.5 + 9\delta$	0—01111111110—0000000000000000000000000000000000000000000000001001

Widać, że wartości δ_1 i δ_4 są poprawne z tego samego powodu co poprzednio - wartości zmieniają się na najmniej znaczących bitach (na końcu) w sposób umożliwiający przejście przez wszystkie możliwe wartości mantysy.

$\delta_4 = 2^{-51}$

	bitowa reprezentacja
$2.0 + 1\delta$	0—10000000000—0000000000000000000000000000000000000000000000000001
$2.0 + 2\delta$	0—10000000000—0000000000000000000000000000000000000000000000000010
$2.0 + 3\delta$	0—10000000000—0000000000000000000000000000000000000000000000000011
$2.0 + 4\delta$	0—10000000000—0000000000000000000000000000000000000000000000000100
$2.0 + 5\delta$	0—10000000000—0000000000000000000000000000000000000000000000000101
$2.0 + 6\delta$	0—10000000000—0000000000000000000000000000000000000000000000000110
$2.0 + 7\delta$	0—10000000000—0000000000000000000000000000000000000000000000000111
$2.0 + 8\delta$	0—10000000000—0000000000000000000000000000000000000000000000001000
$2.0 + 9\delta$	0—10000000000—0000000000000000000000000000000000000000000000001001

Zadanie 4

Celem tego zadania było eksperymentalne znalezienie w arytmetyce Float64 (double) najmniejszej liczby zmiennopozycyjnej x w przedziale (1,2) takiej, że $x \cdot (\frac{1}{x}) \neq 1$. Wynik mojego programu to: 1.000000057228997. Reprezentacja tej liczby w arytmetyce Float64 to:

0011111111110000000000000000000000011110101110010111111100101010

Mantysa zinterpretowana jako samodzielna liczba to 257736490. Oznacza to, że program musiał wykonać aż 257736490 - 1 = 257736480 iteracji. Kolejnym krokiem w zadaniu było znalezienie najmniejszej takiej liczby x. Przez najmniejszą taką liczbę rozumiem tu najmniejszą liczbę >0. Wynik działania programu to $5.0e-324$. Jest to wartość MIN_{sub}. Reprezentacja tej liczby w arytmetyce Float64 to:

0000000000000000000000000000000000000000000000000000000000000001

Matematycznie mamy $MIN_{sub} \cdot (\frac{1}{MIN_{sub}}) = 1$, a program stwierdził, że tak nie jest. Dlaczego? Przyjrzyjmy się obliczonej przez komputer wartości: $\frac{1}{MIN_{sub}} = Inf$. Stąd później $MIN_{sub} \cdot (\frac{1}{MIN_{sub}}) = MIN_{sub} \cdot Inf = Inf \neq 1$. Stąd właśnie wziął się taki wynik programu. Wnioskiem z zadania jest to, że ze względu na ograniczenia reprezentacji liczb, niektóre działania (nawet te podstawowe) mogą dawać błędne wyniki.

Zadanie 5

Zadanie to polegało na eksperymentalnym obliczeniu 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].
W treści zadania są podane 4 algorytmy:
a (forward),b (backward),c (descending),d (ascending), które zaimplementowałem. Algorytmy są opisane w treści zadania, więc nie będę ich tu opisywać. Kolejnym krokiem w zadaniu było porównanie otrzymanych wartości z wartościa prawdziwą, zrealizowałem to licząc błąd bezwzględny i względny. Przypomnijmy - wartość dokładna (z dokładnością do 15 cyfr, według treści zadania) wynosi $-1.00657107 \cdot 10^{-11}$. Wyniki mojego programu znajdują się w tabeli poniżej:

Dla Float64

alogrytm	obliczona wartość	błąd bezwzględny	błąd względny
Forward	1.0251881368296672e-10	1.1258452438296672e-10	11.184955313981627
Backward	-1.5643308870494366e-10	1.4636737800494365e-10	14.541186645165915
Descending	0.0	1.00657107e-11	1.0
Ascending	0.0	1.00657107e-11	1.0

Dla Float32

alogrytm	obliczona wartość	błąd bezwzględny	błąd względny
Forward	-0.4999443	0.49994429944939167	4.9668057661282845e10
Backward	-0.4543457	0.4543457031149343	4.513796558000096e10
Descending	-0.5	0.4999999999899343	4.967359135306107e10
Ascending	-0.5	0.4999999999899343	4.967359135306107e10

Patrząc na wyniki nasuwa się kilka ważnych wniosków. Wartości obliczone przez program nie są równe wartości dokładnej bez względu na użyty algorytm. Ponadto różnice w wartościach obliczonych przez poszczególne algorytmy okazały się kompletnie nieintuicyjne. Algorytm c (descending), który wydaje się *najgorszy* okazał się widocznie lepszy od algorytmu a i b (dla Float64) i na dodatek dał taki sam wynik jak algorytm d (ascending) (dla Float64 i Float32), który wydaje się *najlepszy*. Przy czym przez *najlepszy/najgorszy* rozumiem taki który oblicza wartość odpowiednio najdokładniej/najmniej dokładnie.
Inną rzeczą wartą zauważenia jest też fakt, że różnice w błędach dla dwóch testowanych typów okazały się bardzo duże - błędy dla Float64 były znacznie mniejsze. Kolejnym wnioskiem jest to, że licząc iloczyn skalarny dla Float64 otrzymaliśmy w 2 przypadkach wartość 0.0, co nieuważnemu użytkownikowi dałoby podstawy żeby twierdzić, że wekotry x i y są prostopadłe, kiedy w rzeczywistości tak nie jest.

Zadanie 6

Celem zadania było policzenie i porównanie wartości dwóch matematycznie równych funkcji (f i g) dla argumentów: $8^{-1}, 8^{-2}, 8^{-3}, ...$ i określenie które z otrzymanych wyników są wiarygodne.

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

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

W tabeli poniżej znajdują się wyniki mojego programu.

x	f(x)	g(x)	f = g
8^{-1}	0.0077822185373186414	0.0077822185373187065	false
8^{-2}	0.00012206286282867573	0.00012206286282875901	false
8^{-3}	1.9073468138230965e-6	1.907346813826566e-6	false
8^{-4}	2.9802321943606103e-8	2.9802321943606116e-8	false
8^{-5}	4.656612873077393e-10	4.6566128719931904e-10	false
8^{-6}	7.275957614183426e-12	7.275957614156956e-12	false
8^{-7}	1.1368683772161603e-13	1.1368683772160957e-13	false
8^{-8}	1.7763568394002505e-15	1.7763568394002489e-15	false
8^{-9}	0.0	2.7755575615628914e-17	false
8^{-10}	0.0	4.336808689942018e-19	false
8^{-11}	0.0	6.776263578034403e-21	false
8^{-12}	0.0	1.0587911840678754e-22	false
8^{-13}	0.0	1.6543612251060553e-24	false
8^{-14}	0.0	2.5849394142282115e-26	false
8^{-15}	0.0	4.0389678347315804e-28	false
8^{-16}	0.0	6.310887241768095e-30	false
8^{-17}	0.0	9.860761315262648e-32	false
8^{-18}	0.0	1.5407439555097887e-33	false
8^{-19}	0.0	2.407412430484045e-35	false
⋮			
8^{-23}	0.0	1.4349296274686127e-42	false
8^{-24}	0.0	2.2420775429197073e-44	false
8^{-25}	0.0	3.503246160812043e-46	false
8^{-26}	0.0	5.473822126268817e-48	false
8^{-27}	0.0	8.552847072295026e-50	false
8^{-28}	0.0	1.3363823550460978e-51	false
8^{-29}	0.0	2.088097429759528e-53	false
8^{-30}	0.0	3.2626522339992623e-55	false

Widać, że dla żadnego z badanych argumentów funkcje nie są równe. Widać również, że począwszy od 8^{-9} funkcja f daje wartość równą 0.0 podczas kiedy funkcja g nie. Zastanówmy się skąd mogła się wziąć ta wartość 0.0. Rozważmy sytuację kiedy liczba $x \ll 1.0$. W takim przypadku może wystąpić zjawisko tzw. pochłonięcia. Przy wykonywaniu dodawania cechy liczb są wyrównywane i w przypadku znacznej różnicy w wartościach liczb, cyfry liczby x^2 mogą zostać całkiem pochłonięte gdyż przy takiej samej cesze mantysy nie są w stanie uwzględnić cyfr znaczących dla obu liczb. W rezultacie pod pierwiastkiem pojawia się wartość 1.0. Stąd po wykonaniu dodawania otrzymujemy $\sqrt{1} - 1 = 0$.
Zobaczmy jak to wygląda dla x = 8^{-9}. Pierwsze co robimy to podnosimy x do kwadratu. Mamy: $(8^{-9})^2 = 8^{-18} = (2^3)^{-18} = 2^{3\cdot(-18)} = 2^{-54} = x_0$. Nasza mantysa może przechować maksymalnie 52 cyfry. Reprezentacje liczb, które będziemy do siebie dodawać są dokładne. (Przez zapis 000...0 rozumiemy 52 zera) $1.0 = 2^0 \cdot 1.000...0; 2^{-54} = 2^{-54} \cdot 1.000...0$. (W obu przypadkach mantysy wypełnione zerami). Teraz chcemy wyrównać cechy do większej z nich, zatem mantysę (wraz z niepisaną jedynką z przodu - 1.mantysa) liczby 2^{-54} przesuwamy o 54 miejsca. Teraz przystępujemy do dodawania (dodajemy w dwa razy większej precyzji) $x^2 + 1.0 = 1.\underbrace{000...001}_{54 cyfry}$. W rezultacie ostatnia jedynka uciekła poza 52. pozycję i zostanie ucięta - otrzymujemy 1.000...0 - same zera po przecinku. Następnie liczmy z tego pierwiastek i odejmujemy od tego 1.0: $\sqrt{1.0} - 1.0 = 1.0 - 1.0 = 0.0$. Stąd właśnie wzięło się nasze zero w wyniku.
Rodzi się pytanie, dlaczego funkcja g nie zachowała się tak jak f. Widać, że w mianowniku wystąpi podobny problem - cały pierwiastek w pewnym momencie będzie równy 1.0, a co za tym idzie wartość całego mianownika przyjmie wartość 2.0. (Warto nadmienić, że dzielenie przez 2 jest wykonywane dokładnie). Jednak pozostanie tu licznik, który nadal będzie wpływał na wynik funkcji. Zwróćmy uwagę na to, że w momencie w którym pochłonęliśmy x obliczając wartość *f(x)* straciliśmy całkiem informację o argumencie - jego wartość straciła znaczenie i nie mogła już wpłynąć na wartość funkcji. Inaczej jest z fukcją g, w momencie pochłonięcia x-a w mianowniku, x nie stracił możliwości wpływania na wartość funkcji. Podsumowując, funkcja g jest odporniejsza na zjawisko pochłaniania niż funkcja f. Stąd wartości funkcji g są bardziej wiarygodne.

Zadanie 7

Ostatnie zadanie polegało na obliczeniu przybliżonej wartości funkcji $f(x) = sin(x) + cos(3x)$ w punkcie $x_0 = 1$ korzystając ze wzoru:

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

Ponadto dla $h = 2^{-n}(n = 0, 1, 2, 3, 4, ..., 54)$ trzeba było obliczyć błędy $|f'(x) - \widetilde{f}'(x)|$. Dokładną wartość pochodnej funkcji możemy obliczyć z matematycznego wzoru: $f'(x) = cos(x) - 3sin(3x)$. Wyniki działania mojego programu znajdują się w tabeli poniżej.

n	$\widetilde{f}'(1)$	$\lvert f'(x) - \widetilde{f}'(x)\rvert$	h+1
0	2.0179892252685967	1.9010469435800585	2.0
1	1.8704413979316472	1.753499116243109	1.5
2	1.1077870952342974	0.9908448135457593	1.25
3	0.6232412792975817	0.5062989976090435	1.125
4	0.3704000662035192	0.253457784514981	1.0625
5	0.24344307439754687	0.1265007927090087	1.03125
6	0.18009756330732785	0.0631552816187897	1.015625
		...	
26	0.11694233864545822	5.6956920069239914e-8	1.0000000149011612
27	0.11694231629371643	3.460517827846843e-8	1.0000000074505806
28	0.11694228649139404	4.802855890773117e-9	1.0000000037252903
29	0.11694222688674927	5.480178888461751e-8	1.0000000018626451
30	0.11694216728210449	1.1440643366000813e-7	1.0000000009313226
31	0.11694216728210449	1.1440643366000813e-7	1.0000000004656613
32	0.11694192886352539	3.5282501276157063e-7	1.0000000002328306
33	0.11694145202636719	8.296621709646956e-7	1.0000000001164153
34	0.11694145202636719	8.296621709646956e-7	1.0000000000582077
35	0.11693954467773438	2.7370108037771956e-6	1.0000000000291038
36	0.116943359375	1.0776864618478044e-6	1.000000000014552
		...	
44	0.1171875	0.0002452183114618478	1.0000000000000568
45	0.11328125	0.003661031688538152	1.0000000000000284
46	0.109375	0.007567281688538152	1.0000000000000142
47	0.109375	0.007567281688538152	1.000000000000007
48	0.09375	0.023192281688538152	1.0000000000000036
49	0.125	0.008057718311461848	1.0000000000000018
50	0.0	0.11694228168853815	1.0000000000000009
51	0.0	0.11694228168853815	1.0000000000000004
52	-0.5	0.6169422816885382	1.0000000000000002
53	0.0	0.11694228168853815	1.0
54	0.0	0.11694228168853815	1.0

Widać, że dla n = 53 i 54 wartość $\widetilde{f}'(1)$ wyniosła 0 (dla dalszych wartości też tak jest), a wartość $h+1 = 1$. Patrząc na definicję *macheps* z zadania 1. możemy stwierdzić, że liczba h wraz ze wzrostem liczby n zbliżała się do *macheps*. (Zwróćmy uwagę, że dla $n = 52$, h wyniosło $2^{-52} = macheps$ i na to, że wartość $x_0 = 1$ jest tu kluczowa). Skutkowało to, że od wartości $n = 53$ wyrażenie $f(1+h) - f(1)$ przybierało postać $f(1) - f(1) = 0$. Dla $n > 2^{-52}$, h zostaje pochłonięte podczas dodawania w liczniku i przez to później licznik przyjmuje wartość 0. Wyjaśnia to dlaczego wartości h+1 od pewnego momentu są równe dokładnie 1.

Kiedy przyjrzymy się dokładniej wartościom błędów bezwzględnych widzimy, że dokładność przybliżenia zaczeła się pogarszać dużo wcześniej niż przy 2^{-53}. Wartość dla funkcji 2^{-28} wydaje się najdokładniejsza, gdyż błąd bezwzględny jest wtedy najmniejszy. Dalsze wartości mają już większy błąd. Uzasadnieniem dlaczego się tak dzieje jest zjawisko utraty cyfr znaczących. Zwróćmy uwagę,

że kiedy komputer oblicza przybliżoną wartość pochodnej w punkcie $x_0 = 1$ według wzoru podanego powyżej, liczby w liczniku tj. $f(x_0 + h)$ i $f(x_0)$ mają wartości bardzo bliskie sobie. Kiedy następuje ich odejmowanie następuje zjawisko utraty cyfr znaczących. Wyjaśnia to dlaczego malejące h przestaje od pewnego momentu zwiększać dokładność przybliżenia wartości pochodnej. Wynika z tego, że należy unikać odejmowania liczb bliskich sobie, ponieważ narażamy się na wyżej opisane zjawisko.