Sprawozdanie
obliczenia naukowe 1

Autor: Jan Sieradzki
Nr indeksu: 236441

Zadanie 1

A)

I. Krótki opis problemu :

Napisać program, który wyznaczy iteracyjnie epsilony maszynowe dla typów zmiennopozycyjnych Float16,Float32,Float64, zgodnych ze standardem IEEE754, porownać rezultaty z wartościami zwracanymi przez odpowiednie funkcje, podające epsilony maszynowe, w Juli.

II. Rozwiązanie :

```
function macheps16()
        x = Float16(1.0)
        y = Float16(1.0)
        while Float16(1.0) + y > 1.0
            x = y
            y = y / Float16(2.0)
        end
        return x
end
```

Aby znaleźć epsilony maszynowe, dzielę x przez 2 (czyli przesuwam liczbę o bit), dopóki wartość y (to jest x/2) zwiększona o 1 nie będzie dawać 1 (ponieważ y będze tak małe, że komputer pominie ją, gdyż będzie tak małe). Po wykonaniu pętli, funkcja zwraca x, czyli ostatnią zapamientaną liczbę zanim x/2 komputer pomija.

III. Wyniki oraz ich interpretacja :

```
eps(Float16)=0.000977,                  moja funkcja macheps16=0.000977
eps(Float32)=1.1920929e-7,              moja funkcja macheps32=1.1920929e-7
eps(Float64)=2.220446049250313e-16, moja funkcja macheps64=2.220446049250313e-16
```

Jak widać, wniki uzyskane z mojej funkcji pokrywają się z wynikiem funkcji wbudowanej eps(), co oznacza, że udało mi się znaleźć epsilon maszynowy dla typów zmiennopozycyjnych Float16,Float32,Float64.

IV. Wnioski :

Udało mi się obliczyć iteracyjnie epsilon maszynowy (wynosi 2^{1-t} gdzie t to ilość bitów przeznaczonych na mantysę), który jest dwukrotnie większy niż precyzja arytmetki, którą oznaczaliśmy na wykładzie epsilonem. Innym wnioskiem jest, iż im więcej bitowy Float, tym dokładniejszą jet arytmetyką.

B)

I. Krótki opis problemu :

Napisać program, który wyznaczy iteracyjnie liczbę eta, taką , że eta > 0.0 dla typów zmiennopozycyjnych Float16,Float32,Float64, zgodnych ze standardem IEEE754, porownać rezultaty z wartościami zwracanymi przez odpowiednie funkcje w Juli.

II. Rozwiązanie :

```
function eta16()
        x = Float16(1.0)
        while (x/Float16(2.0) > 0.0)
           x = x/Float16(2.0)
        end
        return x
end
```

Aby znaleźć eta, dzielę x przez 2 (czyli przesuwam liczbę o bit), dopóki wartość x/2 będzie większe od zera . Po wykonaniu pętli, funkcja zwraca x, czyli ostatnią zapamientaną liczbę zanim x/2 dało zero maszynowe.

III. Wyniki oraz ich interpretacja :

```
nextfloat(Float16(0.0))=6.0e-8,     moja funkcja eta16()=6.0e-8
nextfloat(Float32(0.0))=1.0e-45,   moja funkcja eta32()=1.0e-45
nextfloat(Float64(0.0))=5.0e-324, moja funkcja eta64()=5.0e-324
```

Jak widać, wniki uzyskane z mojej funkcji pokrywają się z wynikiem funkcji wbudowanej nextfloat(0.0), która zwraca MIN_{sub} co oznacza, że udało mi się znaleźć MIN_{sub} dla typów zmiennopozycyjnych Float16,Float32,Float64.

IV. Wnioski :

Istnieje najmniejsza liczba MIN arytmetyki, dla której liczby których moduł jest mniejszy nniż |MIN| są rozumiane przez komputer jako zero maszynowe. Taką liczbą jest właśnie szukana ETA, która jest właściwie MIN_{sub}.

C)

I. Krótki opis problemu :

Napisać program, który wyznaczy iteracyjnie liczbę MAX dla typów zmiennopozycyjnych Float16,Float32,Float64, zgodnych ze standardem IEEE754, porownać rezultaty z wartościami zwracanymi przez odpowiednie funkcje w Juli.

II. Rozwiązanie :

```
function max16()
        x = Float16(1.0)
        while !isinf(x*Float16(2.0))
                x = x * Float16(2.0)
        end
        i = eta16()
        while isinf(x*(Float16(2.0)-i))
                i = i*Float16(2.0)
        end
        return x*(Float16(2.0)-i)
end
```

Jako, że w momencie, gdy liczba w danej arytmetyce przekroczy MAX, jest traktowana jako nieskończoność, podwajam w pętli x, dopóki x nie będzie czytane jako nieskończoność przez funkcję isinf(). Pobieram ostatnie x, które przed podwojeniem nie było nieskończone I chcę je wymnożyć razy $(2\text{-}2^{-(t-1)})$, gdyż MAX liczy się : $(2\text{-}2^{-(t-1)})* 2^{Cmax}$. X to właśnie 2^{Cmax}, a aby zdobyć $(2\text{-}2^{-(t-1)})$ odejmuję od 2 liczbę I, którą konstruuję przypisując do niej liczbę MIN, a ponieważ MIN = $(2^{-(t-1)})* 2^{Cmin}$, pozbywam się 2^{Cmin} mnożąc w pętli i*2, dopóki nie osiągnę celu. W ten sposób otrzymuję liczbę Max = $(2\text{-}2^{-(t-1)})* 2^{Cmax}$.

III. Wyniki oraz ich interpretacja :

realmax(Float16)=6.55e4,	*moja funckja max16()=6.55e4*
realmax(Float32)=3.4028235e38,	*moja funckja max32()=3.4028235e38*
realmax(Float64)=1.7976931348623157e308,	*moja funkcja max64()=1.7976931348623157e308*

Jak widać, wniki uzyskane z mojej funkcji pokrywają się z wynikiem funkcji wbudowanej realmax(), która zwraca MAX co oznacza, że udało mi się znaleźć MAX dla typów zmiennopozycyjnych Float16,Float32,Float64.

IV. Wnioski :

Jeśli liczba przekroczy MAX, jest traktowana jako nieskończoność, oraz Max = $(2\text{-}2^{-(t-1)})* 2^{Cmax}$.

Zadanie 2

I. Krótki opis problemu :

Sprawdzić eksperymentalnie czy prawdą jest stwierdzenie Kahan’a, że 3(4/3-1)-1 = macheps.

II. Rozwiązanie :

Wykonuję działanie w każdej z trzech arytmetyk I porównuję z wynikiem funkcji eps(), która zwraca macheps (epsilon maszynowy).

III. Wyniki oraz ich interpretacja :

Float16	
Kahan macheps: -0.000977,	**eps(Float16)=0.000977**
Float32	
Kahan macheps: 1.1920929e-7,	**eps(Float32)=1.1920929e-7**
Float64	
Kahan macheps: -2.220446049250313e-16	**eps(Float64)=2.220446049250313e-16**

Wynik ze wzoru zgadza się z macheps tylko w Float32, w pozostałych dwóch rytmetykach liczby różnią się znakami.

IV. Wnioski :

Stwierdzenie Kahana dokładnie sprawdza się tylko w arytmetyce Float32, ponieważ macheps powinnien być nieujemny, a we Float16 I Float64 tlko moduł rezultatów jest zgodny z epsilonami maszynowymi.

Zadanie 3

I. Krótki opis problemu :

Sprawdź eksperymentalnie , że w arytmetyce Float64 liczby zmiennopozycyjne są równomiernie rozmieszczone w [1,2] z krokiem $\delta = 2^{-52}$. Innymi słowy, każda liczba zmiennopozycyjna x pomiędzy 1 i 2 może być przedstawione następująco $x = 1 + k\delta$ w tej arytmetyce, gdzie $k = 1, 2, \ldots, 2^{52} - 1$ i $\delta = 2^{-52}$. Jak rozmieszczone są liczby zmiennopozycyjne w przedziale [0.5 , 1], jak w przedziale [2, 4] i jak mogą być przedstawione dla rozpatrywanego przedziału?

II. Rozwiązanie :

```
delta = Float64(2)^-52
println("przedzial [1,2]\n")
x = Float64(1)
for i = 1 : 6
        x += delta
        println(x,"  ",bits(x))
end
```

Analogicznie dla każdego przedziału 6 razy dodaję δ I po każdym dodaniu wyświetlam w bitach wynik. Dla przedziału [1,2] początkowe x to 1, dla [0.5,1] x to 0.5, a dla [2,4] x to 2.

III. Wyniki oraz ich interpretacja :

```
przedzial [1,2]
1.0000000000000002  0011111111110000000000000000000000000000000000000000000000000001
1.0000000000000004  0011111111110000000000000000000000000000000000000000000000000010
1.0000000000000007  0011111111110000000000000000000000000000000000000000000000000011
1.0000000000000009  0011111111110000000000000000000000000000000000000000000000000100
1.000000000000001  0011111111110000000000000000000000000000000000000000000000000101
1.0000000000000013  0011111111110000000000000000000000000000000000000000000000000110
przedzial [0.5,1]
0.5000000000000002  0011111111100000000000000000000000000000000000000000000000000010
0.5000000000000004  0011111111100000000000000000000000000000000000000000000000000100
0.5000000000000007  0011111111100000000000000000000000000000000000000000000000000110
0.5000000000000009  0011111111100000000000000000000000000000000000000000000000001000
0.5000000000000011  0011111111100000000000000000000000000000000000000000000000001010
0.5000000000000013  0011111111100000000000000000000000000000000000000000000000001100
przedzial [2,4]
2.0  0100000000000000000000000000000000000000000000000000000000000000
2.0  0100000000000000000000000000000000000000000000000000000000000000
2.0  0100000000000000000000000000000000000000000000000000000000000000
2.0  0100000000000000000000000000000000000000000000000000000000000000
2.0  0100000000000000000000000000000000000000000000000000000000000000
2.0  0100000000000000000000000000000000000000000000000000000000000000
```

Dla przedziału [1,2] każde dodanie δ zwiększa x o 1 bit, czyli liczby są równomiernie rozmieszczone I można je zapisać : x=1 + k*δ. Dla przedziału [0.5, 1] dodanie δ dodje do zapisu 2 bity, a dla przedziału [2,4] w ogóle nie obserwujemy zmian w zapisie bitowym, mimo dodania kolejnych δ.

IV. Wnioski :

Tylko dla przedziału “domyślnego” dla arytmetyki IEEE754, liczby są rozmieszczane równomiernie. Jeżeli chodzi o przedziały [1,2] I [2,4] arytmetyka traci swoją precyzję I nie jest rozmieszczana równomiernie.

Zadanie 4

I. Krótki opis problemu :

Znajdź eksperymentalnie w języku Julia w arytmetyce Float64 zgodnej ze standardem IEEE 754 liczbę zmiennopozycyjną x = przedziale 1 < x < 2, taką, że $x * 1/x \neq 1$.

II. Rozwiązanie :

```
function zad4a()
x = Float64(1.0)

while x*Float64(1/x) == 1.0
        x = nextfloat(Float64(x))
end
println("Taka liczba to x = ", x)
end
```

W pętli while iteracyjnie działałem na x (wartość początkowa 1.0) funkcją nextfloat, dopóki nie znajdę pierwszej liczby x ze zbioru [1 , 2] spełniającej $x * 1/x \neq 1$. Anlogicznie zrobiłem szukając najmniejszej liczby, tutaj za x startowe przyjąłem – realmax(Float64), czyli najmniejszą liczbę arytmetyki.

III. Wyniki oraz ich interpretacja :

```
Taka liczba to x = 1.000000057228997
Taka liczba to x = -1.7976931348623157e308
```

Znalezione liczby spełniają warunek zadania.

IV. Wnioski :

Istnieją takie liczby w arytmetyce Float64, że $x * 1/x \neq 1$.

Zadanie 5

I. Krótki opis problemu :

Napisz program w języku Julia realizujących eksperyment obliczenia iloczynu skalarnego dwóch wektorów
x = [2.718281828, −3.141592654, 1.414213562, 0.5772156649, 0.3010299957]
y = [1486.2497, 878366.9879, −22.37492, 4773714.647, 0.000185049].

II. Rozwiązanie :

Dodaję iteracyjnie odpowiadające sobie wartośći wektorów I tworzę małe sumy :

```
T1a = Float32(x[1] * y[1])
T2a = Float32(x[2] * y[2])
T3a = Float32(x[3] * y[3])
T4a = Float32(x[4] * y[4])
T5a = Float32(x[5] * y[5])
T1b = Float64(x[1] * y[1])
T2b = Float64(x[2] * y[2])
T3b = Float64(x[3] * y[3])
T4b = Float64(x[4] * y[4])
T5b = Float64(x[5] * y[5])
```

następnie te sumy dodaję na cztery sposoby:
A. Po kolei je dodaję.
B. Dodaję je od tyłu.
C. Ustawiam sumy od najmniejszego do największego I je kolejno dodaję.
D. Ustawiam sumy od największego do najmniejszego I je kolejno dodaję.

Po wszystkim wyświetlam wyniki.

III. Wyniki oraz ich interpretacja :

```
Dokładny wynik: Suma = -1.00657107000000e-11
Algorytm A:
 single: Suma = -0.4999443
 double: Suma = 1.0251881368296672e-10
Algorytm B:
 single: Suma = -0.4543457
 double: Suma: S = -1.5643308870494366e-10
Algorytm C:
 single: Suma = -0.5
 double: Suma = 0.0
Algorytm D:
 single: Suma = -0.5
 double: Suma = 0.0
```

Widzimy, że Float64 jest dużo dokładniejszy niż Float32, oraz że najbliżej prawidłowej sumy były algorytmy A I B, a C I D okazały się niefektywne.

IV. Wnioski :

Sposób wykonywania obliżeń ma bardzo duże znaczenie w arytmetykach Float, co jest spowodowane ograniczoną precyzją obliczeń.

Zadanie 6

I. Krótki opis problemu :

Policz w języku Julia w arytmetyce Float64 wartości następujących funkcji: $f(x) = \sqrt{x^2 + 1} - 1$ $g(x) = x\,2\sqrt{x^2+1}+1$ dla kolejnych wartości argumentu $x = 8^{-1}$, 8^{-2} ,8^{-3} ,….,8^{-n}. Otrzymujemy różne wyniki, które są wiarygodne, a które nie?

II. Rozwiązanie :

```
function zad6()
        for i=1:13
                println("f(8^(-$i)) = ", f(8.0^(-i)))
                println("g(8^(-$i)) = ", g(8.0^(-i)))
        end
end
```

Tworzę obie funkcje I iteracyjnie wstawiam do nich 8^{-n} , wyświetlam wyniki.

III. Wyniki oraz ich interpretacja :

```
f(8^(-1)) = 0.0077822185373186414
g(8^(-1)) = 0.0077822185373187065
f(8^(-2)) = 0.00012206286282867573
g(8^(-2)) = 0.00012206286282875901
f(8^(-3)) = 1.9073468138230965e-6
g(8^(-3)) = 1.907346813826566e-6
f(8^(-4)) = 2.9802321943606103e-8
g(8^(-4)) = 2.9802321943606116e-8
f(8^(-5)) = 4.656612873077393e-10
g(8^(-5)) = 4.6566128719931904e-10
f(8^(-6)) = 7.275957614183426e-12
g(8^(-6)) = 7.275957614156956e-12
f(8^(-7)) = 1.1368683772161603e-13
g(8^(-7)) = 1.1368683772160957e-13
f(8^(-8)) = 1.7763568394002505e-15
g(8^(-8)) = 1.7763568394002489e-15
f(8^(-9)) = 0.0
g(8^(-9)) = 2.7755575615628914e-17
f(8^(-10)) = 0.0
g(8^(-10)) = 4.336808689942018e-19
f(8^(-11)) = 0.0
g(8^(-11)) = 6.776263578034403e-21
f(8^(-12)) = 0.0
g(8^(-12)) = 1.0587911840678754e-22
f(8^(-13)) = 0.0
g(8^(-13)) = 1.6543612251060553e-24
```

Jak widać bardziej przekonująca jest funkcja g(x), ponieważ już przy 8^{-10} funkcja f(x) zaczyna zerować wynik, chociaż dokładny rezultat nie może wynosić zero (chociaż jest bardzo bliski zera).

IV. Wnioski :

Nawet jeśli tą samą funkcję przedstawimy inaczej, ma to znaczenie dla arytmetyki Float, gdyż ze względu na ograniczoną precyzję, możemy w jednej formie wzoru stracić więcej na dokładności, niż w innej formie, tak jak w przypadku tego zadania.

Zadanie 7

I. Krótki opis problemu :

Skorzystać z wzoru na przybliżoną wartość pochodnej f(x) w punkcie x i obliczyć w arytmetyce Float64 przybliżone wartości pochodnych funkcji oraz moduł z różnicy przybliżonej pochodnej I dokładnej.

II. Rozwiązanie :

```
f(x) = sin(x) + cos(Float64(3.0)*x)
g(x) = cos(x) - Float64(3.0)*sin(Float64(3.0)*x)


function przyblizenie(h,x)
         return (f(x+h)-f(x))/h
end

function blad(h,x)
   return abs(g(x) - przyblizenie(h,x))
end
```

Zapisujemy funkcje f(x), której pochodną będziemy przybliżać. Liczmy pochodną funkcji f(x) i zapisujemy ją jako g(x). Następnie podstawiamy do wzoru na przybliżoną pochodną odpowiednie h I funkcję f, liczymy pochodną dla różnych $h=2^{-n}$ (n= 0,1,2…,54) I liczymy błąd, tzn moduł z różnicy przybliżonej pochodnej I dokładnej.

III. Wyniki oraz ich interpretacja :

Jak widać błąd przybliżenia zmniejsza się wraz z wzrostem “i”, jednak tylko do i=28, potem dla i=[29,54] błąd znowu rośnie. Warto zwrócić uwagę na sumę (1+h) która z każdą iteracją “i” jest przesuwana o bit, aż dla i=53 suma daje wartość 1.0 (ponieważ jest tak małe, że precyzja arytmetyki ją pomija), co uniemożliwia szacowanie, gdyż górna część wzoru :

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

zeruje się.

i=0	(1+h)= 2.0	szacowanie 2.0179892252685967	blad przyb. 1.9010469435800585
i=1	(1+h)= 1.5	szacowanie 1.8704413979316472	blad przyb. 1.753499116243109
i=2	(1+h)= 1.25	szacowanie 1.1077870952342974	blad przyb. 0.9908448135457593
i=3	(1+h)= 1.125	szacowanie 0.6232412792975817	blad przyb. 0.5062989976090435
i=4	(1+h)= 1.0625	szacowanie 0.3704000662035192	blad przyb. 0.253457784514981
i=5	(1+h)= 1.03125	szacowanie 0.24344307439754687	blad przyb. 0.1265007927090087
i=6	(1+h)= 1.015625	szacowanie 0.18009756330732785	blad przyb. 0.0631552816187897
i=7	(1+h)= 1.0078125	szacowanie 0.1484913953710958	blad przyb. 0.03154911368255764
i=8	(1+h)= 1.00390625	szacowanie 0.1327091142805159	blad przyb. 0.015766832591977753
i=9	(1+h)= 1.001953125	szacowanie 0.1248236929407085	blad przyb. 0.007881411252170345
i=10	(1+h)= 1.0009765625	szacowanie 0.12088247681106168	blad przyb. 0.0039401951225235265
i=11	(1+h)= 1.00048828125	szacowanie 0.11891225046883847	blad przyb. 0.001969968780300313
i=12	(1+h)= 1.000244140625	szacowanie 0.11792723373901026	blad przyb. 0.0009849520504721099
i=13	(1+h)= 1.0001220703125	szacowanie 0.11743474961076572	blad przyb. 0.0004924679222275685
i=14	(1+h)= 1.00006103515625	szacowanie 0.11718851362093119	blad przyb. 0.0002462319323930373
i=15	(1+h)= 1.000030517578125	szacowanie 0.11706539714577957	blad przyb. 0.00012311545724141837
i=16	(1+h)= 1.000015258789625	szacowanie 0.11700383928837255	blad przyb. 6.155759983439424e-5
i=17	(1+h)= 1.0000076293945312	szacowanie 0.11697306045971345	blad przyb. 3.077877117529937e-5
i=18	(1+h)= 1.0000038146972656	szacowanie 0.11695767106721178	blad przyb. 1.5389378673624776e-5
i=19	(1+h)= 1.0000019073486328	szacowanie 0.11694997636368498	blad przyb. 7.694675146829866e-6
i=20	(1+h)= 1.0000009536743164	szacowanie 0.11694612901192158	blad przyb. 3.8473233834324105e-6
i=21	(1+h)= 1.0000004768371582	szacowanie 0.1169442052487284	blad przyb. 1.9235601902423127e-6
i=22	(1+h)= 1.000000238418579	szacowanie 0.11694324295967817	blad przyb. 9.612711400208696e-7
i=23	(1+h)= 1.0000001192092896	szacowanie 0.11694276239722967	blad przyb. 4.807086915192826e-7
i=24	(1+h)= 1.0000000596046448	szacowanie 0.11694252118468285	blad przyb. 2.394961446938737e-7
i=25	(1+h)= 1.0000000298023224	szacowanie 0.116942398250103	blad przyb. 1.1656156484463054e-7
i=26	(1+h)= 1.0000000149011612	szacowanie 0.11694233864545822	blad przyb. 5.6956920069239914e-8
i=27	(1+h)= 1.0000000074505806	szacowanie 0.11694231629371643	blad przyb. 3.460517827846843e-8
i=28	(1+h)= 1.0000000037252903	szacowanie 0.11694228649139404	blad przyb. 4.802855890773117e-9
i=29	(1+h)= 1.0000000018626451	szacowanie 0.11694222688674927	blad przyb. 5.480178888461751e-8
i=30	(1+h)= 1.0000000009313226	szacowanie 0.11694216728210449	blad przyb. 1.1440643366000813e-7
i=31	(1+h)= 1.0000000004656613	szacowanie 0.11694216728210449	blad przyb. 1.1440643366000813e-7
i=32	(1+h)= 1.0000000002328306	szacowanie 0.11694192886352539	blad przyb. 3.5282501276157063e-7
i=33	(1+h)= 1.0000000001164153	szacowanie 0.11694145202636719	blad przyb. 8.296621709646956e-7
i=34	(1+h)= 1.0000000000582077	szacowanie 0.11694145202636719	blad przyb. 8.296621709646956e-7
i=35	(1+h)= 1.0000000000291038	szacowanie 0.11693954467773438	blad przyb. 2.7370108037771956e-6
i=36	(1+h)= 1.000000000014552	szacowanie 0.116943359375	blad przyb. 1.0776864618478044e-6
i=37	(1+h)= 1.000000000007276	szacowanie 0.1169281005859375	blad przyb. 1.4181102600652196e-5
i=38	(1+h)= 1.000000000003638	szacowanie 0.116943359375	blad przyb. 1.0776864618478044e-6
i=39	(1+h)= 1.000000000001819	szacowanie 0.11688232421875	blad przyb. 5.995746978815219 6e-5
i=40	(1+h)= 1.0000000000009095	szacowanie 0.1168212890625	blad przyb. 0.0001209926260381522
i=41	(1+h)= 1.0000000000004547	szacowanie 0.116943359375	blad przyb. 1.0776864618478044e-6
i=42	(1+h)= 1.0000000000002274	szacowanie 0.11669921875	blad przyb. 0.0002430629385381522
i=43	(1+h)= 1.0000000000001137	szacowanie 0.1162109375	blad przyb. 0.0007313441885381522
i=44	(1+h)= 1.0000000000000568	szacowanie 0.1171875	blad przyb. 0.0002452183114618478
i=45	(1+h)= 1.0000000000000284	szacowanie 0.11328125	blad przyb. 0.003661031688538152
i=46	(1+h)= 1.0000000000000142	szacowanie 0.109375	blad przyb. 0.007567281688538152
i=47	(1+h)= 1.000000000000007	szacowanie 0.109375	blad przyb. 0.007567281688538152
i=48	(1+h)= 1.0000000000000036	szacowanie 0.09375	blad przyb. 0.023192281688538152
i=49	(1+h)= 1.0000000000000018	szacowanie 0.125	blad przyb. 0.008057718311461848
i=50	(1+h)= 1.0000000000000009	szacowanie 0.0	blad przyb. 0.11694228168853815
i=51	(1+h)= 1.0000000000000004	szacowanie 0.0	blad przyb. 0.11694228168853815
i=52	(1+h)= 1.0000000000000002	szacowanie -0.5	blad przyb. 0.6169422816885382
i=53	(1+h)= 1.0 szacowanie 0.0	szacowanie 0.0	blad przyb. 0.11694228168853815
i=54	(1+h)= 1.0 szacowanie 0.0	szacowanie 0.0	blad przyb. 0.11694228168853815

IV. Wnioski :

Wzor na pochodną najlepiej działa dla i=28, ponieważ jest to kompromis między utratą dokładności przy dzieleniu przez h, a wzrostem dokładności wzoru ze względu na mniejsze h.