Eliza Zawisza, 244967

SPRAWOZDANIE 1

ZADANIE 1

Opis problemu:

W zadaniu należało zapoznać się z arytmetyką zmiennopozycyjną. Zadanie składało się z trzech części, w których należało wyznaczyć odpowiednio:

1. liczbę macheps, czyli najmniejszą liczbę macheps > 0 taka, że że 1.0 + macheps > 1.0.
2. liczbę eta, czyli najmniejszą liczbę > 0.0
3. liczbę MAX, czyli największą liczbę mniejszą od nieskończoności

dla arytmetyki Float16, Float32 i Float64 oraz porównać je z liczbami zwracanymi przez funkcje wbudowane w bibliotekach Julii: *eps()*, *nextfloat()*, *floatmax()* oraz z danymi zawartymi w pliku nagłówkowym float.h dowolnej instancji języka C (dotyczny tylko macheps i MAX).

Rozwiązania:

I. EPSILON MASZYNOWY

W celu znalezienia epsilona maszynowego, dla zadanych arytmetyk użyłam algorytmu, który w pętli dzieli przez dwa do momentu, gdy 1 + szukana liczba jest większa od 1. Poniżej kod algorytmu, który można znaleźć w pliku *zadanie1.jl*

```
function findMacheps(Type)
  macheps = Type(1.0)
  divider = Type(2.0)
  while Type(1.0) + macheps/divider > Type(1.0)
    macheps = macheps/divider
  end
return macheps
end
```

II. LICZBA ETA

W celu wyznaczenia liczby eta zastosowałam algorytm polegający na dzieleniu w pętli liczby 1 przez 2 do momentu, w którym wynik dzielenia jest większy od 0. Poniżej kod algorytmu, który można znaleźć w pliku *zadanie1.jl*

```
function findEta(Type)
  eta = Type(1.0)
  divider = Type(2.0)
  while eta/divider > Type(0.0)
    eta = eta/divider
  end
return eta
end
```

III. LICZBA MAX

W celu wyznaczenia liczby max zastosowałam algorytm, który w pętli dokonuje mnożenia liczby 1 przez 2, dopóki liczba ta nie jest nieskończonością. Jako warunek sprawdzający użyłam wbudowanej funkcji bibliotecznej *!isinf*. Następnie znaleziony wynik mnożę razy różnicę liczby 2 i espilonu maszynowego zadanego typu arytmetyki.

```
function findMax(Type)
    max = Type(1.0)
  multiplier = Type(2.0)

  while !isinf(max*multiplier)
     max = max*multiplier
  end
  max = max * (Type(2.0)-eps(Type))
 return max
end
```

Wyniki oraz ich interpretacja:

I. MACHEPS – wartości

Typ arytmetyki zmiennopozycyjnej	**Obliczony wynik**	**Wynik metody eps()**	**Wynik z pliku float.h**
Float16	0.000977	0.000977	brak danych
Float32	1.1920929e-7	1.1920929e-7	1.1920928955e-07
Float64	2.220446049250313e-16	2.220446049250313e-16	2.2204460493e-16

Zestawienie wyników wartości epsilonu maszynowego dla zadanego typu arytmetyki oraz porównanie z wartościami z pliku nagłówkowego float.h

II. ETA – wartości

Typ arytmetyki zmiennopozycyjnej	**Obliczony wynik**	**Wynik metody nextfloat()**
Float16	6.0e-8	6.0e-8
Float32	1.0e-45	1.0e-45
Float64	5.0e-324	5.0e-324

Zestawienie wyników wartości epsilonu maszynowego dla zadanego typu arytmetyki.

III. MAX – wartości

Typ arytmetyki zmiennopozycyjnej	**Obliczony wynik**	**Wynik metody floatmax()**	**Wynik z piku float.h**
Float16	6.55e4	6.55e4	brak danych
Float32	3.4028235e38	3.4028235e38	3.4028234664e+38
Float64	1.7976931348623157e308	1.7976931348623157e308	1.7976931348623157e+308

Zestawienie wyników wartości epsilonu maszynowego dla zadanego typu arytmetyki oraz porównanie z wartościami z pliku nagłówkowego float.h

Wnioski:

Z powyższych tabel wynika, że algorytmy użyte przez mnie do obliczanie poszczególnych liczb dla konkretnych arytmetyk są prawidłowe, ponieważ ich wyniki zgadzają się z wynikami funkcji wbudowanych w bibliotekach Julia.

Epsilon maszynowy jest dwukrotnie większy od precyzji arytmetyki. Precyzja arytmetyki wg wzoru wynosi 2^{-t-1}, gdzie t oznacza długość mantysy, natomiast $\epsilon = 2^{-t}$.

Eta jest najmniejsza liczbą większą od 0. Jej zapis bitowy składający się z samych 0 i ostatniej 1 pokazuje, że jest to liczba zdenormalizowana(subnormal).

Funkcja floatmin() zwraca najmniejszą znormalizowaną liczbę dla zadanej arytmetyki.

Analizując tabele można w łatwy sposób dostrzec, że wraz ze wzrostem precyzji arytmetyki zmniejsza się epsilon maszynowy, dzięki czemu liczby są dokładniej reprezentowane, a obliczenia, które wykonujemy mają dokładniejsze wyniki.

ZADANIE 2

Opis problemu:

W zadaniu należało obliczyć epsilon maszynowy używając wzoru Kahana do obliczania macheps: $3\left(\frac{4}{3}-1\right)-1$. Oraz sprawdzić jego poprawność dla zadanych typów arytmetyki: Float16, Float32, Float64.

Rozwiązanie:

W celu obliczenia epsilonu maszynowego dla podanych arytmetyk użyłam funkcji zawartej w pliku zadanie2.jl:

```
function Kahan_Macheps(Type)
   return Type(3.0) * (Type(4.0) / Type(3.0) - Type(1.0)) - Type(1.0)
end
```

Wyniki oraz ich interpretacja:

Typ arytmetyki zmiennopozycyjnej	**Wynik obliczeń**	**Wartość funkcji eps()**
Float16	-0.000977	0.000977
Float32	1.1920929e-7	1.1920929e-7
Float64	-2.220446049250313e-16	2.220446049250313e-16

Zestawienie wyników wartości macheps wyliczonych za pomocą wzoru Kahana oraz zwracanych przez funkcję eps().

Wnioski:

Z powyższej tabeli wynika, że sposób obliczania wartości epsilonu maszynowego dla zadanych typów arytmetyki daje wyniki poprawne z dokładnością do znaku. Błąd ten wynika z reprezentacji liczby $4/3$ w systemie dwójkowym, której część ułamkowa jest okresowa, przez co musimy zastosować zaokrąglenie liczby i następuje zmiana znaku. Można łatwo zauważyć, że wzór Kahana byłby prawidłowy, gdybyśmy nałożyli na niego wartość bezwzględną.

ZADANIE 3

Opis problemu:

W zadaniu należało sprawdzić eksperymentalnie rozmieszczenie liczb zmiennopozycyjnych w zakresach: [1,2], $[\frac{1}{2}, 1]$ oraz [2,4] w arytmetyce double w standardzie IEEE754. Wiedząc, że liczby zmiennopozycyjne są równomiernie rozmieszczone w [1, 2] z krokiem $\delta = 2^{-52}$.

Rozwiązanie:

W celu sprawdzenia rozłożenia liczb użyłam funkcji *bitstring()*, która zwraca reprezentację binarną danej liczby. W pętli wypisywałam kolejne 8 liczb z zadaną deltą i początkiem przedziału sprawdzania, by znaleźć odpowiednią deltę – rozmieszczenie liczb, dla poszczególnych zakresów. Do sprawdzania używałam delt: 2^{-51} i 2^{-53}, gdyż zakresy te rozpoczynają się kolejnymi potęgami dwójek.

Kod programu znajduje się w pliku *zadanie3.jl*

```
function check_delta_correctness(delta, beginning)
   for k in 1:8
      println(bitstring(Float64(beginning + k*delta)))
   end
end
```

Wyniki oraz ich interpretacja:

Range [1, 2]:
Delta = 2^(-52)
0011111111110000000000000000000000000000000000000000000000000**1**
0011111111110000000000000000000000000000000000000000000000000**10**
0011111111110000000000000000000000000000000000000000000000000**11**
0011111111110000000000000000000000000000000000000000000000000**100**
0011111111110000000000000000000000000000000000000000000000000**101**
0011111111110000000000000000000000000000000000000000000000000**110**
0011111111110000000000000000000000000000000000000000000000000**111**
0011111111110000000000000000000000000000000000000000000000000**1000**

Dla zakresu [1,2] delta wynosi 2^{-52}, gdyż ostatnie bity zmieniają się poprawnie o 1.

Range [1/2, 1]:
Delta = 2^(-52)
0011111111100000000000000000000000000000000000000000000000000**10**
0011111111100000000000000000000000000000000000000000000000000**100**
0011111111100000000000000000000000000000000000000000000000000**110**
0011111111100000000000000000000000000000000000000000000000000**1000**
0011111111100000000000000000000000000000000000000000000000000**1010**
0011111111100000000000000000000000000000000000000000000000000**1100**
0011111111100000000000000000000000000000000000000000000000000**1110**
0011111111100000000000000000000000000000000000000000000000000**10000**

Range [1/2, 1]:
Delta = 2^(-51)
0011111111100000000000000000000000000000000000000000000000000**100**
0011111111100000000000000000000000000000000000000000000000000**1000**
0011111111100000000000000000000000000000000000000000000000000**1100**
0011111111100000000000000000000000000000000000000000000000000**10000**
0011111111100000000000000000000000000000000000000000000000000**10100**

0011111111100000000000000000000000000000000000000000000000**11000**
0011111111100000000000000000000000000000000000000000000000**11100**
0011111111100000000000000000000000000000000000000000000000**100000**
Range [1/2, 1]: **Delta = 2^(-53)** 0011111111100000000000000000000000000000000000000000000000000**1** 0011111111100000000000000000000000000000000000000000000000000**10** 0011111111100000000000000000000000000000000000000000000000000**11** 0011111111100000000000000000000000000000000000000000000000000**100** 0011111111100000000000000000000000000000000000000000000000000**101** 0011111111100000000000000000000000000000000000000000000000000**110** 0011111111100000000000000000000000000000000000000000000000000**111** 0011111111100000000000000000000000000000000000000000000000000**1000**

Dla zakresu $[\frac{1}{2}, 1]$ delta wynosi 2^{-53} , gdyż ostatnie bity zmieniają się poprawnie o 1.

Range [2, 4]: **Delta = 2^(-52)** 0100000000000000000000000000000000000000000000000000000000000**0** 0100000000000000000000000000000000000000000000000000000000000**1** 0100000000000000000000000000000000000000000000000000000000000**10** 0100000000000000000000000000000000000000000000000000000000000**10** 0100000000000000000000000000000000000000000000000000000000000**10** 0100000000000000000000000000000000000000000000000000000000000**11** 0100000000000000000000000000000000000000000000000000000000000**100** 0100000000000000000000000000000000000000000000000000000000000**100**
Range [2, 4]: **Delta = 2^(-51)** 0100000000000000000000000000000000000000000000000000000000000**1** 0100000000000000000000000000000000000000000000000000000000000**10** 0100000000000000000000000000000000000000000000000000000000000**11** 0100000000000000000000000000000000000000000000000000000000000**100** 0100000000000000000000000000000000000000000000000000000000000**101** 0100000000000000000000000000000000000000000000000000000000000**110** 0100000000000000000000000000000000000000000000000000000000000**111** 0100000000000000000000000000000000000000000000000000000000000**1000**
Range [2, 4]: **Delta = 2^(-53)** 0100000000000000000000000000000000000000000000000000000000000**0** 0100000000000000000000000000000000000000000000000000000000000**0** 0100000000000000000000000000000000000000000000000000000000000**1** 0100000000000000000000000000000000000000000000000000000000000**1** 0100000000000000000000000000000000000000000000000000000000000**1** 0100000000000000000000000000000000000000000000000000000000000**10** 0100000000000000000000000000000000000000000000000000000000000**10** 0100000000000000000000000000000000000000000000000000000000000**10**

Dla zakresu [1,2] delta wynosi 2^{-51}, gdyż ostatnie bity zmieniają się poprawnie o 1.

Wnioski:

Reprezentacja zmiennopozycyjna w standardzie IEEE 754 double składa się z 64 bitów:

- 1 bit to bit znaku, jeśli liczba jest dodatnia jest 0, w przeciwnym razie jest 1;
- kolejne 11 bitów odpowiada za cechę
- ostatnie 52 bity to mantysa liczby

Jak widać na poniższych przykładach wszystkie liczby mają taką samą cechę i znak, różnią się jedynie mantysą, która zwiększa się o 1. Dla podanych przedziałów delty wynoszą odpowiednio 2^{-51}, 2^{-52}, 2^{-53}. Ilość liczb w każdym z przedziałów: [1,2], $[\frac{1}{2}, 1]$ oraz [2,4] jest taka sama, jednakże ich gęstość rozłożenia jest różna. Im większa odległość przedziału od 0.0, tym odstępy między liczbami w zadanym przedziale są większe.

ZADANIE 4

Opis problemu:

Celem zadania było znalezienie dwóch liczb zmiennopozycyjnych w arytmetyce Float64, które spełniają warunek: $x * (1/x) = 1$. Pierwsza liczba miała być z przedziału $1<x<2$, druga liczba miała być najmniejszą liczbą spełniającą to równanie.

Rozwiązanie:

W celu rozwiązania zadania użyłam algorytmu, który pobierał odpowiednio najmniejszą liczbę większą od 1 i sprawdzał warunek zadania oraz najmniejsza liczbę w zadanej arytmetyce i sprawdzał warunek zadania. Poniższe kody pochodzą z pliku *zadanie4.jl*

```
function findSmallestInRange()
        x = nextfloat(Float64(1.0))
        while Float64(x * Float64(1 / x)) == 1
                x = nextfloat(Float64(x))
        end
return x
end
```

```
function findSmallest()
        x = -floatmax(Float64)
        while Float64(x * Float64(1 / x)) == 1
                x = nextfloat(Float64(x))
        end
return x
end
```

Wyniki oraz ich interpretacja:

Wynikiem pierwszej funkcji jest liczba: 1.000000057228997, która przy sprawdzeniu z warunkiem zadania daje wynik: $0.9999999999999999 \neq 1$

Wynikiem drugiej funkcji jest liczba: -1.7976931348623157e308, która przy sprawdzeniu z warunkiem zadania daje wynik: $0.9999999999999999 \neq 1$

Wnioski:

Biorąc pod uwagę powyższe wyniki można zauważyć, że błędy zaokrągleń a arytmetyce zmiennopozycyjnej sprawiają, że równanie $x * \frac{1}{x} \neq 1$ może zostać spełnione.

ZADANIE 5

Opis problemu:

Celem zadania było obliczenie sumy iloczynu skalarnego dwóch zadanych wektorów:

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

na 4 różne sposoby: w przód, w tył, od największego do najmniejszego oraz od najmniejszego do największego w arytmetyce Float32 i Float64.

Rozwiązanie:

W celu obliczenia sum iloczynów skalarnych należało zaimplementować 4 funkcje obliczające sumę na każdy z 4 sposobów. Do obliczenia sum od najmniejszego do największego oraz od największego do najmniejszego zaimplementowałam dwie pomocnicze funkcje: *computeSortSum(negativeElements, positiveElements, Type)* – funkcja zwracała sumę z posortowanych tablic zawierających odpowiednio iloczyny wektorów > 0 – *positiveElements* oraz iloczyny wektorów < 0 – *negativeElements; computeScalarProduct(Type)* - funkcja zwracała tablicę danego typu obliczonych wektorów skalarnych. Poniżej przestawiam implementacje funkcji, zawarte w pliku *zadanie5.jl.*

```
function a(Type)
   sum = Type(0.0)
   for i = 1:n
      sum+= Type(x[i])*Type(y[i])
   end
   return sum
end
```

```
function b(Type)
   sum = Type(0.0)
   i = n
   while i != 0
      sum+= Type(x[i])*Type(y[i])
      i = i - 1
   end
   return sum
end
```

```
function c(Type)
   productArray = computeScalarProduct(Type)
   negativeElements=[]
   positiveElements=[]

   for i = 1:length(productArray)
      if productArray[i] > 0.0
         push!(positiveElements, productArray[i])
      else
         push!(negativeElements, productArray[i])
      end
   end

   sort!(negativeElements)
   sort!(positiveElements, rev = true)

   sum = computeSortSum(negativeElements, positiveElements, Type)
   return sum
end
```

```
function d(Type)
   productArray = computeScalarProduct(Type)
   negativeElements=[]
   positiveElements=[]
```

```
    for i = 1:length(productArray)
        if productArray[i] > 0.0
            push!(positiveElements, productArray[i])
        else
            push!(negativeElements, productArray[i])
        end
    end

    sort!(negativeElements, rev = true)
    sort!(positiveElements)

    sum = computeSortSum(negativeElements, positiveElements, Type)
    return sum
end
```

```
function computeSortSum(negativeElements, positiveElements,
Type)
    sumNeagtive = Type(0.0)
    sumPositive = Type(0.0)

    for i in 1:length(negativeElements)
        sumNeagtive += negativeElements[i]
    end

    for i in 1:length(positiveElements)
        sumPositive += positiveElements[i]
    end

   return sumPositive + sumNeagtive
end
```

```
function computeScalarProduct(Type)
    array = []
    for i = 1:n
        push!(array, Type(x[i])*Type(y[i]))
    end
    return array
end
```

Wyniki oraz ich interpretacja:

Sposób	**Float32**	**Float64**	**Prawidłowy wynik**
A – w przód	-0.4999443	1.0251881368296672e-10	−1.0065710700000010 − 11
B – w tył	-0.4543457	-1.5643308870494366e-10	
C – od największego do najmniejszego	-0.5	0.0	
D – od najmniejszego do największego	-0.5	0.0	

Jak widać w powyższej tabeli wszystkie wyliczone wartości zarówno w arytmetyce Float32, jak i Float64 są różne od prawidłowego wyniku.

Wnioski:

Wyniki w przedstawionej tabeli pokazują, że ilość działań, jaką wykonujemy w czasie policzenia w tym przypadku sumy iloczynu skalarnego dwóch wektorów ma znaczenie. Ponadto istotnym czynnikiem

wpływającym na błąd wyniku jest rząd wielkości dodawanych liczb, gdy podczas dodawania liczby będą skrajnie różnić się od siebie rzędem wielkości błąd obliczeń będzie większy. Dlatego też w celu najmniejszego obarczenia błędem wyniku powinno się sumować liczby od najmniejszej do największej.

ZADANIE 6

Opis problemu:

Celem zadania było policzyć w arytmetyce Float64 wartości funkcji:

$f(x) = \sqrt{x^2+1} - 1$, $g(x) = \frac{x^2}{\sqrt{x^2+1}+1}$ dla kolejnych wartości argumentów: x = 8^{-1}, 8^{-2}, 8^{-3}, . . .

Rozwiązanie:

W celu obliczenia wartości funkcji zastosowałam poniższe algorytmy, dla argumentów: 8^{-1}, 8^{-2}, 8^{-3}, . . ., 8^{-100}. Kody źródłowe dostępne są w pliku *zadanie6.jl*

function **f**(x)
  return Float64(sqrt(x^2 + 1) - 1)
end
	function **g**(x)
	  return Float64(x^2/(sqrt(x^2 + 1) + 1))
	end

Wyniki oraz ich interpretacja:

Porównanie pierwszych 15 wyników:

Wartość 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^{-11}	0.0	6.776263578034403e-21
8^{-12}	0.0	1.0587911840678754e-22
8^{-13}	0.0	1.6543612251060553e-24
8^{-14}	0.0	2.5849394142282115e-26
8^{-15}	0.0	4.0389678347315804e-28

Jak widać z powyższej tabeli funkcja f(x) i g(x) do pewnego momentu zwracają takie same wyniki. Niestety później funkcja f(x) zaczyna zwracać wynik 0.0. Pomimo tego, że funkcje są sobie równe.

Wnioski:

Z matematycznego punktu widzenia zarówno funkcja f(x) i g(x) mają granicę $\lim_{x\to 0} f(x)$ i $\lim_{x\to 0} g(x)$ równą 0, tzn, że wartości funkcji powinny zbliżać się do 0, ale nigdy jej nie osiągnąć. Funkcja f(x) przy obliczaniu wartości f(x) daje wyniki 0, ze względu na odejmowanie od siebie dwóch bardzo zbliżonych liczb, co powoduje utratę cyfr znaczących. W funkcji g(x) nie mamy do czynienia z tym zjawiskiem, ponieważ nie występuje odejmowanie.

ZADANIE 7

Opis problemu:

Celem zadania było obliczyć wartość pochodnej funkcji $f(x) = \sin x + cos3x$ w punkcie $x_0 = 1$, wykorzystując do tego wzór na przybliżoną wartość pochodnej $\frac{f(x0+h)-f(x0)}{h}$, dla $h = 2^{-n}$, ($n = 0,1,2,...,54$). Oraz obliczyć błąd względny przybliżenia dla poszczególnych h i prawdziwej wartości pochodnej w punkcie.

Rozwiązanie:

W celu obliczenia zadania należało napisać funkcje, obliczające przybliżoną wartość pochodnej oraz błąd względny przybliżenia. Poniżej algorytmy wykorzystane do obliczenia pochodnej, które dostępne są w pliku *zadanie7.jl*

```
function f(x)
   return sin(x) + cos(3x)
end
```

```
function fApxDerivative(x0, h)
   return (f(x0 + h) - f(x0)) / h
end
```

```
function fDerivative(x)
   return cos(x) - 3sin(3x)
end
```

```
function approximationError(realValue, approximateValue)
   return abs(realValue-approximateValue)
end
```

Wyniki oraz ich interpretacja:

Wartości pochodnej oraz błędu przybliżenia dla wybranych h.

Wartość h	Przybliżona wartość pochodnej	Błąd przybliżenia	1+h
2^{-0}	2.0179892252685967	1.9010469435800585	2.0
2^{-1}	1.8704413979316472	1.753499116243109	1.5
2^{-2}	1.1077870952342974	0.9908448135457593	1.25
2^{-3}	0.6232412792975817	0.5062989976090435	1.125
2^{-4}	0.3704000662035192	0.253457784514981	1.0625
2^{-5}	0.24344307439754687	0.1265007927090087	1.03125
2^{-10}	0.12088247681106168	0.0039401951225235265	1.0009765625
2^{-15}	0.11706539714577957	0.00012311545724141837	1.000030517578125
2^{-20}	0.11694612901192158	3.8473233834324105e-6	1.0000009536743164
2^{-25}	0.116942398250103	1.1656156484463054e-7	1.0000000298023224
2^{-30}	0.11694216728210449	1.1440643366000813e-7	1.0000000009313226
2^{-35}	0.11693954467773438	2.7370108037771956e-6	1.0000000000291038
2^{-40}	0.1168212890625	0.0001209926260381522	1.0000000000009095
2^{-45}	0.11328125	0.003661031688538152	1.0000000000000284
2^{-50}	0.0	0.11694228168853815	1.0000000000000009
2^{-51}	0.0	0.11694228168853815	1.0000000000000004
2^{-52}	-0.5	0.6169422816885382	1.0000000000000002
2^{-53}	0.0	0.11694228168853815	1.0

2^{-54}	0.0	0.11694228168853815	1.0

Z powyższych wyników można zobaczyć, że najlepsze przybliżenie pochodnej, otrzymujemy dla wartości $h=2^{-30}$, gdyż prawidłowy wynik pochodnej wynosi: 0.11694228168853815.

Wnioski:

Z tabeli łatwo można wywnioskować, że wartość 1+h zmniejsza się wraz ze zmniejszaniem się wartości h, do momentu całkowitego pochłonięcia wartości przez 1. Wynika to z dużej różnicy rzędu wielkości dwóch dodawanych do siebie wartości. Błąd przybliżenia, który w pewnym momencie ponownie zaczyna wzrastać wynika właśnie z różnic rzędów wielkości odejmowanych od siebie wartości. Aby otrzymywać wyniki obarczone najmniejszym błędem powinno się unikać działań na liczbach będących bardzo blisko 0.