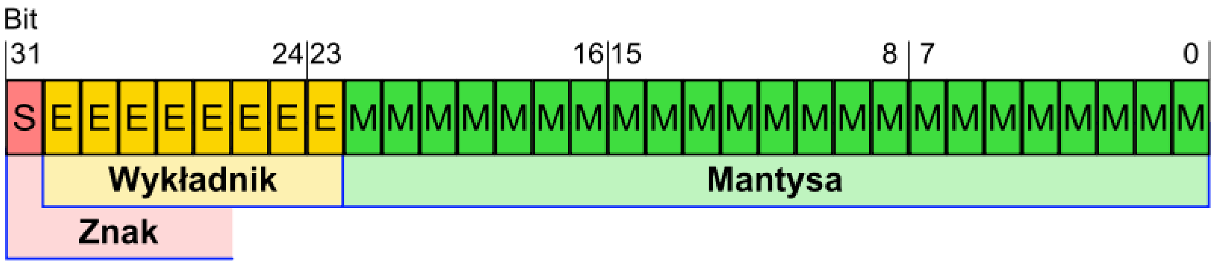
OBLICZENIA NAUKOWE

Lista 1

NIEBIESKI KAPTUREK
PPT-FIFA-TEAM®
PAŹDZIERNIK 2018

1. Cel:

Przedmiotem pierwszej listy z przedmiotu Obliczenia naukowe pod przewodnictwem dr hab. Pawła Zielińskiego było zapoznanie się z językiem Julia oraz reprezentacją liczb zmiennoprzecinkowych w standardzie IEEE754 (Rysunek 1). W dziewięciu zadaniach, które były na tej liście mieliśmy wyznaczyć epsilon maszynowy, minimum i maksimum we wszystkich dostępnych typach zmiennopozycyjnych (half, single, double), a także eksperymentalnie sprawdzić słuszność twierdzenia o epsilonie maszynowym, rozmieszczenia liczb w arytmetyce Float64 i obliczenia iloczynu skalarnego oraz pochodnej funkcji.



Rysunek 1: Schemat 32-bitowej liczby zmiennoprzecinkowej w standardzie IEEE754.[1]

2. Realizacja:

2.1 Zadanie 1 - Rozpoznanie arytmetyki

Pierwszy podpunkt zadania polega na wyznaczeniu epsilonu maszynowego *macheps*[2] dla wszystkich dostępnych typów zmiennopozycyjnych Float16, Float32, Float64, zgodnych ze standardem IEEE754 (half, single, double), i porównać z wartościami zwracanymi przez funkcje: eps(Float16), eps(Float32), eps(Float64) oraz z danymi zawartymi w pliku nagłówkowym float.h dowolnej instalacji języka C.

```
# funkcja rekurencyjna, wyliczająca epsilon maszynowy w Float16
function macheps16(x)
        macheps = x/2
        if 1 + macheps > 1
                macheps16(Float16(macheps))
        else
                return x
        end
end
```

Listing1: Funkcja, wyliczająca epsilon maszynowy w Float16, znajdująca się w zad1.jl

[1] Źródło: https://pl.wikipedia.org/wiki/Plik:IEEE-754-single1.svg

[2] Epsilonem maszynowym (ang. machine epsilon) - najmniejszą liczba dodatnia taka, że *fl*(1.0 + *macheps*) > 1.0.

Dzielimy wartość zmiennej *macheps* przed dwa, do momentu, gdy 1 + *macheps* > 1, wtedy funkcja zwróci połowę tej liczby, czyli najmniejszą dodatnią, która spełnia warunki zadania, co oznacza, że jest to epsilon maszynowy wyznaczony iteracyjnie w podanej arytmetyce, w tym przypadku half ze standardu IEEE754, ale dla innych robi się analogicznie (zamiast funkcji Float16 używamy Float32 lub Float64).

```
# funkcja, pokazująca wyniki podpunktu a)
function zad1a()
println("eps(Float16)=",eps(Float16),", macheps16()=",macheps16(1))
println("eps(Float32)=",eps(Float32),", macheps32()=",macheps32(1))
println("eps(Float64)=",eps(Float64),", macheps64()=",macheps64(1))
end
```

Listing2: Funkcja, pokazująca wyniki podpunktu a) oraz epsilony maszynowe, znajdująca się w zad1.jl

```
julia> zad1a()
eps(Float16)=0.000977, macheps16=0.000977
eps(Float32)=1.1920929e-7, macheps32=1.1920929e-7
eps(Float64)=2.220446049250313e-16, macheps64=2.220446049250313e-16
```

Listing3: Wynik działania funkcji *zad1a()*, znajdującej się w zad1.jl

name	value	stands for	expresses
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1E-5 or smaller 1E-9 or smaller 1E-9 or smaller	EPSILON	Difference between 1 and the least value greater than 1 that is representable.

Tabela1: Dane z pliku nagłówkowego float.h

Drugi podpunkt zadania polegał na iteracyjnym wyznaczeniu liczby *eta* takiej, że *eta* > 0.0 dla wszystkich typów zmiennopozycyjnych Float16, Float32, Float64, zgodnych ze standardem IEEE 754 (half, single, double), i porównaniu z wartościami zwracanymi przez funkcje: nextfloat(Float16(0.0)), nextfloat(Float32(0.0)), nextfloat(Float64(0.0))

```
#funkcja rekurencyjna, wyznaczająca liczbę eta w Float16
function eta16(x)
	eta = x/2
	if eta > 0
		eta16(Float16(eta))
	else
		return x
	end
end
```

Listing4: Funkcja, wyliczająca liczbę eta, znajdująca się w zad1.jl

Problem z podpunktu drugiego rozwiązujemy podobnie do tego wcześniejszego. Tworzymy funkcję, która rozwiąże to w sposób rekurencyjny. Dzielimy wartość zmiennej *eta* przed dwa, do momentu, gdy *eta* > 0, wtedy funkcja zwróci połowę tej liczby, czyli najmniejszą dodatnią, która spełnia warunki zadania, co oznacza, że jest to szukana liczba *eta* w podanej arytmetyce, w tym przypadku half ze standardu IEEE754, ale dla innych robi się analogicznie (zamiast funkcji Float16 używamy Float32 lub Float64).

```
# funkcja, pokazująca wyniki podpunktu b)

function zad1b()
println("nextfloat(Float16(0.0))=",nextfloat(Float16(0.0)), ", eta16()=",eta16(1))
println("nextfloat(Float32(0.0))=",nextfloat(Float32(0.0)), ", eta32()=",eta32(1))
println("nextfloat(Float64(0.0))=",nextfloat(Float64(0.0)), ", eta64()=",eta64(1))
end
```

Listing5: Funkcja, pokazująca wyniki podpunktu b) oraz następne liczby po zerze, znajdująca się w zad1.jl

```
julia> zad1b()
nextfloat(Float16(0.0))=6.0e-8, eta16()=6.0e-8
nextfloat(Float32(0.0))=1.0e-45, eta32()=1.0e-45
nextfloat(Float64(0.0))=5.0e-324, eta64()=5.0e-324
```

Listing6: Wynik działania funkcji *zad1b()*, znajdującej się w zad1.jl

Ostatni problem w zadaniu pierwszym polegało na napisaniu programu w języku Julia, wyznaczającego liczbę (*MAX*) dla wszystkich typów zmiennopozycyjnych Float16, Float32, Float64, zgodnych ze standardem IEEE 754 (half, single, double), i porównaniu z wartościami zwracanymi przez funkcje: realmax(Float16), realmax(Float32), realmax(Float64) oraz z danymi zawartymi w pliku nagłówkowym float.h dowolnej instalacji języka C lub z danymi z wykładu.

```
function max16(x)
	max = x*2
	if !isinf(max)
		max16(Float16(max))
	else
		return x
	end
end
```

Listing7: Funkcja pomocnicza, pomagająca wyliczyć max we Float16, znajdująca się w zad1.jl

```
function mymax16(x)
        i = eta16(1)
        while !isinf(x*(2-i))
                i=i*2
        else
                return x*(2-i)
        end
end
```

Listing8: Druga funkcja, pomagająca wyliczyć max we Float16, znajdująca się w zad1.jl

Rozwiązanie problemu wydaje się bardziej skomplikowane, niż w poprzednich podpunktach. Najpierw trzeba użyć funkcji *max16(x)*, wylicza ona największą liczbę, będącą w postaci $2^k, k\epsilon\mathbb{Z}$, która w danej arytmetyce nie jest jeszcze nieskończonością. Następnie przekazujemy tę liczbę, do drugiej funkcji, która szuka właściwego maksimum, będącego w postaci $max16(x)(2-i)$, gdzie i, to najmniejsza możliwa liczba, żeby to wyrażenie nie było nieskończonością w danej arytmetyce zmiennopozycyjnej. Wartość i rozpoczynam od wartości minimalnej *eta*, którą wyliczyliśmy w poprzednim podpunkcie.

```
# funkcja, pokazująca wyniki podpunktu c)
function zad1c()
println("realmax(Float16)=", realmax(Float16), ", mymax16()=",
mymax16(max16(1)))
println("realmax(Float32)=", realmax(Float32), ", mymax32()=",
mymax32(max32(1)))
println("realmax(Float64)=", realmax(Float64), ", mymax64()="
```

Listing9: Funkcja, pokazująca wyniki podpunktu c) oraz maksima w danych arytmetykach, znajdująca się w zad1.jl

```
julia> zad1c()
realmax(Float16)=6.55e4, mymax16()=6.55e4
realmax(Float32)=3.4028235e38, mymax32()=3.4028235e38
realmax(Float64)=1.7976931348623157e308,
mymax64()=1.7976931348623157e308
```

Listing10: Wynik działania funkcji *zad1c()*, znajdującej się w zad1.jl

format	MIN_{sub}	MIN_{nor}	MAX
single	1.4_{10} - 45	1.2_{10} - 38	3.4_{10} - 38
double	4.9_{10} - 324	2.2_{10} - 324	1.8_{10} - 308

Tabela2: Dane z wykładu

Wnioski z zadania pojawią się w punkcie 3.1 sprawozdania.

2.2 Zadanie 2 - Stwierdzenie Kahana o epsilonie maszynowym

Drugie zadanie polega na tym, żeby obliczyć wyrażenie $3(\frac{4}{3} - 1) - 1$, w arytmetyce wszystkich typów zmiennopozycyjnych Float16, Float32 i Float64, oraz porównać je z epsilonami maszynowymi.

```
function zad2()
   kahanmacheps16 =
                  Float16(
                        Float16(
                              3 * Float16(
                                    Float16(
                                          4/3
                                          )
                                    - 1
                                    )
                              )
                        - 1
                        )

         kahanmacheps32 =
               Float32(
                     Float32(
                           3 * Float32(
                                 Float32(
                                       4/3
                                       )
                                 - 1
                                 )
                           )
                     - 1
                     )
         kahanmacheps64 =
               Float64(
                     Float64(
                           3 * Float64(
                                 Float64(
                                       4/3
                                       )
                                 - 1
                                 )
                           )
                     - 1
                     )

         println("kahanmacheps16=",kahanmacheps16,", eps(Float16)=",eps(Float16))
         println("kahanmacheps32=",kahanmacheps32,", eps(Float32)=",eps(Float32))
         println("kahanmacheps64=",kahanmacheps64,", eps(Float64)=",eps(Float64))

end
```

Listing11: Pełny kod zadania 2, znajdującego się w zad2.jl

Rozwiązaniem problemu jest dobre zdefiniowanie, jak obliczyć pożądany wynik. Każde obliczenia powinny być w odpowiedniej arytmetyce pozycyjnej.

```
julia> zad2()
kahanmacheps16=-0.000977, eps(Float16)=0.000977
kahanmacheps32=1.1920929e-7, eps(Float32)=1.1920929e-7
kahanmacheps64=-2.220446049250313e-16, eps(Float64)=2.220446049250313e-16
```

Listing12: Wynik działania funkcji *zad2()*, znajdującej się w zad2.jl

Wnioski z zadania pojawią się w punkcie 3.2 sprawozdania.

2.3 Zadanie 3 - Równomierne rozmieszczenie liczb

W trzecim zadaniem mierzymy się z problemem równomiernego rozmieszczenia liczb w arytmetyce Float64 (arytmetyce double w standardzie IEEE754). Mamy sprawdzić, że na przedziale [1,2] są rozmieszczone z krokiem $\partial = 2^{-52}$.

W celu sprawdzenia wykonałem różne działania, wykorzystującą funkcję bits(), która zwraca reprezentację bitową danej liczby.

0	0	1	1	1	1	1	1	1	1	1	1	0	...	0	0	0

Rysunek2: Reprezentacja bitowa liczby 1.0 we Float64, wynik funkcji *bits(1.0)*

Widzimy na wyżej załączonym rysunku, że mantysa wypełniona jest samymi zerami. Teraz chciałbym porównać reprezentację dwóch liczb: $1.0 + \partial$ oraz następnej po 1 we Float64, czyli korzystamy z funkcji *nextfloat(Float64(1.0))*.

0	0	1	1	1	1	1	1	1	1	1	1	0	...	0	0	**1**

Rysunek3: Reprezentacja bitowa następnej liczby po 1.0 we Float64, czyli 1.0+∂

Obliczenia potwierdzają, że te dwie liczby mają taką samą bitową reprezentację, czyli w arytmetyce Float64 są sobie równe. Dla pewności sprawdźmy jeszcze, jak wygląda reprezentacja bitowa liczb: 2.0, $2.0 - \partial$, $1.0 + \partial(2^{52} - 1)$.

0	1	0	0	0	0	0	0	0	0	0	0	0	...	0	0	0

Rysunek4: Reprezentacja bitowa liczby 2.0 we Float64, wynik funkcji *bits(2.0)*

0	0	1	1	1	1	1	1	1	1	1	1	1	...	1	1	1

Rysunek5: Reprezentacja bitowa liczby 2.0-∂ we Float64, a także 1.0+∂(2^{52}-1)

Obliczenia potwierdzają również, że i te liczby mają taką samą bitową reprezentację, czyli w arytmetyce Float64 są sobie równe. Z treści zadania wiemy, że:
$x = 1 + k\partial,\ k = 1, 2, \ldots, 2^{52} - 1$ i $\partial = 2^{-52}$, czyli liczb pomiędzy 1 a 2 powinno być $(2^{52} - 1) - 1 + 1$ (k_{MAX} - k_{MIN} + 1), czyli dokładnie $2^{52} - 1$.
Teraz policzmy, ile możliwości stwarza reprezentacja bitowa. Z wyżej sprawdzonych przykładów widzimy, że bit znaku oraz wykładnik jest taki sam dla: 1.0, 1.0+∂ i 2.0-∂. Oznacza to, że różnią się tylko rozmieszczeniem bitów w mantysie. Zatem liczb pomiędzy 1 a 2 jest tyle ile możliwości zapełnienia 52 bitów mantysy i odjęciu przypadek z samymi zerami, bo tak oznaczamy liczbę 1.0. Nietrudno, więc zauważyć, że tych liczb jest tyle samo, czyli dokładnie $2^{52} - 1$.

Sprawdźmy teraz podpunkt b) tzn, ile wynosi reprezentacja bitowa dla liczby 0.5, 0.5+∂ i *nextfloat(Float64(0.5))*:

0	0	1	1	1	1	1	1	1	1	1	**0**	0	...	0	**0**	**0**

Rysunek6: Reprezentacja bitowa liczby 0.5 we Float64

0	0	1	1	1	1	1	1	1	1	1	**0**	0	...	0	**0**	**1**

Rysunek7: Reprezentacja bitowa liczby *nextfloat(Float64(0.5))*

0	0	1	1	1	1	1	1	1	1	1	**0**	0	...	0	**1**	**0**

Rysunek8: Reprezentacja bitowa liczby 0.5 + ∂

Widzimy tutaj, że w tym przedziale ∂ zwiększa zapis bitowy o 2 bity.

W podpunkcie c musimy natomiast sprawdzić, ile wynosi reprezentacja bitowa dla liczby 2.0, 2.0+∂, 2.0+2∂ i *nextfloat(Float64(2.0))*:

0	1	0	0	0	0	0	0	0	0	0	0	0	...	0	**0**	**0**

Rysunek9: Reprezentacja bitowa liczby 2.0 we Float64

0	1	0	0	0	0	0	0	0	0	0	0	0	...	0	**0**	**1**

Rysunek10: Reprezentacja bitowa liczby *nextfloat(Float64(2.0))*

0	1	0	0	0	0	0	0	0	0	0	0	0	...	0	**0**	**0**

Rysunek11: Reprezentacja bitowa liczby 2.0 + ∂

0	1	0	0	0	0	0	0	0	0	0	0	0	...	0	**0**	**1**

Rysunek12: Reprezentacja bitowa liczby 2.0 + 2∂

Widzimy tutaj, że w tym przedziale 2∂ zwiększa zapis bitowy o bit.

2.4 Zadanie 4 - "x*(1/x) ≠ 1"

Czwarte zadanie polega na znalezieniu eksperymentalnie w arytmetyce Float64 zgodnej ze standardem IEEE 754 (double) liczby zmiennopozycyjnej x w przedziale $1 < x < 2$, takiej, że $x * \frac{1}{x} \neq 1$, tj. fl(xfl(1/x))≠1, a później znalezieniu najmniejszej takiej liczby.

```
function zad4a()
        x = Float64(1.0)
        while Float64(x*Float64(1/x)) == 1.0
        x = nextfloat(Float64(x))
        end
println("x = ", x)
end
```

Listing13: Funkcja, szukająca najmniejszą liczbę spełniającą początkowe warunki zadania, znajdująca się w zad4.jl

Napisana funkcja zwraca liczbę, która nie spełnia warunków zadania. Mój program zwraca x = 1.000000057228997, jest to najmniejsza liczba spełniająca wszystkie warunki zadania, a ponieważ sprawdzamy od najmniejszej liczby i nie omijamy żadnej, to uzyskany wynik jest na pewno najmniejszy.

Drugi podpunkt zadania można zrozumieć na kilka sposobów, ja rozróżniłem trzy z nich:

1. najmniejsza liczba z przedziału 1 < x < 2 (jest to rozwiązanie omówione wyżej)
2. najmniejsza liczba, patrząc na moduł tej liczby (będzie ona bliska minimum)
3. najmniejsza liczba, patrząc na wartość tej liczby (będzie ona bliska - MAX)

```
function zad4b()
        x = Float64(0.0)
        while Float64(x*Float64(1/x)) == 1.0
        x = nextfloat(Float64(x))
        end
println("x = ", x)
end
```

Listing14: Funkcja, szukająca najmniejszą liczbę - drugie rozumowanie, znajdująca się w zad4.jl

```
function zad4c()
        x = - realmax(Float64)
        while Float64(x*Float64(1/x)) == 1.0
        x = nextfloat(Float64(x))
        end
println("x = ", x)
end
```

Listing15: Funkcja, szukająca najmniejszą liczbę - trzecie rozumowanie, znajdująca się w zad4.jl

W obu przypadkach już pierwsza liczba nie spełnia warunków pętli while.

2.5 Zadanie 5 - Iloczyn skalarny dwóch wektorów

W zadaniu piątym mamy napisać program w języku Julia realizujący następujący eksperyment obliczania 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].

Zaimplementować mamy cztery algorytmy i policzyć sumę na cztery sposoby:

(a) "w przód" $\sum_{i=1}^{n} x_i \, y_i$, tj. algorytm

S := 0
for *i* := 1 to *n* **do**
    S := *S* + $x_i * y_i$
end for

(b) "w tył" $\sum_{i=n}^{1} x_i \, y_i$, tj. algorytm

S := 0
for *i* := n **downto** *1* **do**
    S := *S* + $x_i * y_i$
end for

(c) od największego do najmniejszego (dodać dodatnie liczby w porządku od największego do najmniejszego, dodaj ujemne liczby w porządku od najmniejszego do największego, a następnie daj do siebie obliczone sumy częściowe),

(d) od najmniejszego do największego (przeciwnie do metody (c)).

Zacząłem implementację algorytmów od stworzenia wektorów i wyliczenie pojedynczych mnożeń, co widać na załączonym listingu16.

```
# Tworzenie wektorów
x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]
y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]

# Wyliczanie pojedynczych mnożeń (podpunkt c) i d))
S1s = Float32(Float32(x[1]) * Float32(y[1]))
...
S5d = Float64(Float64(x[5]) * Float64(y[5]))
```

Listing16: Początek kodu z zad5.jl

Pierwsze dwa algorytmy zaimplementowałem w następujący sposób:

```
# Podpunkt a) "w przód"
function zad5a32()
	S = Float32(0.0)
	for i=1:5
		S=Float32(S+Float32(Float32(x[i])*Float32(y[i])))
	end
	return S
end
```

Listing17: Implementacja algorytmu A, funkcja z zad5.jl

```
# Podpunkt b) "w tył"
function zad5b32()
	S = Float32(0.0)
	for i=1:5
		S=Float32(S+Float32(Float32(x[6-i])*Float32(y[6-i])))
	end
	return S
end
```

Listing18: Implementacja algorytmu B, funkcja z zad5.jl

Podane są wersje dla arytmetyki Float32, jednakże dla Float64 robi się je analogicznie. Do napisania następnych dwóch algorytmów wykonałem mnożenia z listingu16, po których można było dokonać następujących porównań:

$$S_4 > S_1 > S_5 > 0 > S_3 > S_2$$

Te natomiast pozwoliły na implementację dwóch kolejnych algorytmów:

```
# Podpunkt c) "od największego do najmniejszego"
function zad5c32()
	Splus = Float32(Float32(S4s+S1s)+S5s)
	Sminus = Float32(S2s+S3s)
	S = Float32(Splus+Sminus)
	return S
end
```

Listing19: Implementacja algorytmu C, funkcja z zad5.jl

```
# Podpunkt d) "od najmniejszego do największego"
function zad5d32()
	Splus = Float32(Float32(S5s+S1s)+S4s)
	Sminus = Float32(S3s+S2s)
	S = Float32(Splus+Sminus)
	return S
end
```

Listing20: Implementacja algorytmu D, funkcja z zad5.jl

Podane są wersje dla arytmetyki Float32, jednakże dla Float64 robi się je analogicznie.

```
julia> zad5()
Dokładny wynik: S = -1.00657107000000e-11
Algorytm A:
 pojedyncza precyzja: S = -0.4999443
 podwójna precyzja: S = 1.0251881368296672e-10
Algorytm B:
 pojedyncza precyzja: S = -0.4543457
 podwójna precyzja: S = -1.5643308870494366e-10
Algorytm C:
 pojedyncza precyzja: S = -0.5
 podwójna precyzja: S = 0.0
Algorytm D:
 pojedyncza precyzja: S = -0.5
 podwójna precyzja: S = 0.0
```

Listing21: Wyniki algorytmów, funkcja-odpowiedź z zad5.jl

2.6 Zadanie 6 - Funkcje z pierwiastkiem

Przedostatnie zadanie polega na policzeniu w języku Julia w arytmetyce Float64 wartości następujących funkcji:

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

dla kolejnych wartości argumentu $8^{-1}, 8^{-2}, 8^{-3}$, ...

Chociaż *f* = *g* komputer daje rożne wyniki, należy sprawdzić, które z nich są wiarygodne, a które nie.

Dla zaimplementowanych funkcji w arytmetyce Float64 (listing22), otrzymujemy następujące wyniki (dla argumentów $8^{-i}, i = 1, 2, \ldots, 15$):

```
function kwadrat(x)
        return x*x
end
function f(x)
        return Float64(Float64(sqrt(Float64(kwadrat(Float64(x)))+1.0))-1.0)
end
function g(x)
        return Float64(Float64(kwadrat(Float64(x))) /
        Float64((Float64(sqrt(Float64(kwadrat(Float64(x)))+1.0))+1.0)))
end
function zad6()
        for i=1:15 println("f(8^(-$i)) = ", f(8.0^(-i)))
                println("g(8^(-$i)) = ", g(8.0^(-i)))
        end
end
```

Listing22: Pełny kod zadania6, znajdujący się w zad6.jl

```
julia> zad6()
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
f(8^(-14)) = 0.0
g(8^(-14)) = 2.5849394142282115e-26
f(8^(-15)) = 0.0
g(8^(-15)) = 4.0389678347315804e-28
```

Listing23: Wyniki dwóch funkcji, z zad6.jl

Można zauważyć, że funkcja f już dla argumentu 8^{-9} w arytmetyce Float64 wynosi 0.0.

Wnioski z zadania pojawią się w punkcie 3.6 sprawozdania.

2.7 Zadanie 7 - Pochodna funkcji

W ostatnim zadaniu musimy policzyć pochodną funkcji $f(x) = sin(x) + cos(3x)$ w arytmetyce Float64, korzystając z definicji pochodnej: $f'(x_0) = \lim_{h \to 0} \frac{f(x_0+h) - f(x_0)}{h}$, tzn. $f'(x_0) \approx \widetilde{f}'(x_0) = \frac{f(x_0+h) - f(x_0)}{h}$, gdzie $x_0 = 1$ i $h = 2^{-n}$ $(n = 0, 1, 2, .., 54)$.

Na początku zacznijmy od policzenia dokładnej wartości pochodnej funkcji f, w punkcie $x_0 = 1$:

$f(x) = sin(x) + cos(3x) \Rightarrow f'(x) = (sin(x))' + (cos(3x))' = cos(x) - 3sin(3x)$, czyli

$f'(x_0 = 1) = 0.1169422818853815$

Następnie stwórzmy funkcję w języku Julia i arytmetyce Float64, która wyliczy i pokaże nam pochodną funkcji f, dla coraz mniejszego $h = 2^{-n}$ $(n = 0, 1, 2, .., 54)$.

```
# Zadana funkcja f(x)=sin(x)+cos(3x)
function f(x)
        return Float64(Float64(sin(x)) + Float64(cos(3.0 * x)))
end

# Pochodna z definicji
function pochodnaf(x,h)
        return Float64( Float64(f(Float64(x+h)) - f(x)) / Float64(h) )
end

# Liczenie przybliżonych wartości pochodnej
function wynikpochodnaf()
        for n = 0:54
                println("h = 2^(-$n), f'(1) = ", pochodnaf(1,2.0^-n))
        end
end
```

Listing24: Fragment programu zad7.jl

Pierwsza funkcja, nazywająca się po prostu *f(x)*, wylicza i zwraca wartość funkcji f w arytmetyce Float64. Na wejściu podaje się jej argument x.

Funkcja *pochodnaf(x,h)* zwraca wartość naszej pochodnej w arytmetyce Float64, liczonej w sposób opisany już wyżej. Na wejściu podaje jej się wartość x (w naszym przypadku $x_0 = 1$) oraz h (wyliczamy je w następnej funkcji).

Trzecia funkcja, nazywająca się *wynikpochodnaf()* wylicza przybliżone wartości pochodnej. Znajduje się w niej pętla for, która inkrementuje wartość n $(n \in [0, 54])$, która natomiast potrzebna nam jest do określenia $h = 2^{-n}$.

Następnie zajmijmy się obliczeniem błędu względnego:

$|\, f'(x_0) - \widetilde{f}'(x_0)\, |\ dla\ h = 2^{-n} (n = 0, 1, \ldots, 54).$

Napisałem do tego specjalną funkcję *blad()*:

```
# Liczenie błędu
function blad()
 for n = 0:54
        println("Błąd dla h=2^(-$n): ð = ", abs(pochodnaf(1,2.0^-n)-dokladnapochodnaf(1)))
end
end
```

Listing25: Funkcja, wyliczająca i wyświetlająca błąd względny dla danego h, fragment programu zad7.jl

```
julia> blad()
Błąd dla h=2^(-0): ð = 1.9010469435800585
Błąd dla h=2^(-1): ð = 1.753499116243109
Błąd dla h=2^(-2): ð = 0.9908448135457593
Błąd dla h=2^(-3): ð = 0.5062989976090435
...
Błąd dla h=2^(-23): ð = 4.807086915192826e-7
Błąd dla h=2^(-24): ð = 2.394961446938737e-7
Błąd dla h=2^(-25): ð = 1.1656156484463054e-7
Błąd dla h=2^(-26): ð = 5.6956920069239914e-8
Błąd dla h=2^(-27): ð = 3.460517827846843e-8
Błąd dla h=2^(-28): ð = 4.802855890773117e-9
Błąd dla h=2^(-29): ð = 5.480178888461751e-8
Błąd dla h=2^(-30): ð = 1.1440643366000813e-7
Błąd dla h=2^(-31): ð = 1.1440643366000813e-7
Błąd dla h=2^(-32): ð = 3.5282501276157063e-7
Błąd dla h=2^(-33): ð = 8.296621709646956e-7
Błąd dla h=2^(-34): ð = 8.296621709646956e-7
Błąd dla h=2^(-35): ð = 2.7370108037771956e-6
Błąd dla h=2^(-36): ð = 1.0776864618478044e-6
Błąd dla h=2^(-37): ð = 1.4181102600652196e-5
...
Błąd dla h=2^(-48): ð = 0.023192281688538152
Błąd dla h=2^(-49): ð = 0.008057718311461848
Błąd dla h=2^(-50): ð = 0.11694228168853815
Błąd dla h=2^(-51): ð = 0.11694228168853815
Błąd dla h=2^(-52): ð = 0.6169422816885382
Błąd dla h=2^(-53): ð = 0.11694228168853815
Błąd dla h=2^(-54): ð = 0.11694228168853815
```

Listing26: Fragment wyniku funkcji *blad()* z programu zad7.jl

Wnioski z zadania pojawią się w punkcie 3.7 sprawozdania.

3. Wnioski:

3.1 Zadanie 1 - Rozpoznanie arytmetyki

Z pierwszego podpunktu dowiedzieliśmy się, czym jest epsilon maszynowy *macheps*, czyli najmniejszą liczbą w danym typie zmiennopozycyjnym, taką że: $fl(1.0 + macheps) > 1.0$, a także jakie wartości przyjmuje dla danej arytmetyki:

Nazwa arytmetyki	Wartość epsilonu maszynowego
half	0.000977
single	1.1920929e-7
double	2.220446049250313e-16

Tabela3: Wartości epsilonu maszynowego dla wszystkich typów zmiennopozycyjnych zgodnych ze standardem IEEE754

Z drugiego podpunktu dowiedzieliśmy się, jak wyznaczyć najmniejszą liczbę dodatnią w danym typie zmiennopozycyjnym, która jest równa liczbie $\mathrm{MIN}_{\mathrm{SUB}}$.

Nazwa arytmetyki	Wartość eta	$\mathrm{MIN}_{\mathrm{SUB}}$
half	6.0e-8	$2^{-24} \approx 5.96 \times 10^{-8}$
single	1.0e-45	$2^{-149} \approx 1.4 \times 10^{-45}$
double	5.0e-324	$2^{-1074} \approx 4.94 \times 10^{-324}$

Tabela4: Otrzymane wartości liczby eta oraz $\mathrm{MIN}_{\mathrm{SUB}}$ dla wszystkich typów zmiennopozycyjnych zgodnych ze standardem IEEE754

Z trzeciego podpunktu dowiedzieliśmy się, jak wyznaczyć największą liczbę w danym typie zmiennopozycyjnym.

$$MAX = (2 - 2^{-(t-1)})2^{Cmax} = (2 - 2^{-(t-1)})2^{2^{d-1}-1}$$

Nazwa arytmetyki	MAX
half	$(2 - 2^{-10}) \times 2^{15} = 65504$
single	$(2 - 2^{-23}) \times 2^{127} \approx 3.40 \times 10^{38}$
double	$(2 - 2^{-52}) \times 2^{1023} \approx 1.80 \times 10^{308}$

Tabela5: Wartości MAX dla wszystkich typów zmiennopozycyjnych zgodnych ze standardem IEEE754

3.2 Zadanie 2 - Stwierdzenie Kahana o epsilonie maszynowym

Wykonując wyrażenie $3(\frac{4}{3}-1)-1$ w arytmetyce zmiennopozycyjnej otrzymałem następujące wyniki:

	Kahan	*macheps*
Float16	-0.000977	0.000977
Float32	1.1920929×10^{-7}	1.1920929×10^{-7}
Float64	$-2.2204460492503 \times 10^{-16}$	$2.220446049250313 \times 10^{-16}$

Tabela6: Wyniki z zadania 2

Oznacza to, że w arytmetyce Float32 liczba uzyskana z powyższego wyrażenia jest równa epsilonowi maszynowemu. Natomiast w pozostałych arytmetykach liczby są równe, co do modułu.

Wynika to z tego, że w systemie dwójkowym liczba $\frac{4}{3}$ ma rozwinięcie nieskończone okresowe, co sprawdzaliśmy na liście pierwszej w zadaniu czwartym na ćwiczeniach, więc w zależności od parzystości liczby bitów, składających się na część ułamkową mantysy, dokonujemy zaokrąglenia w dół lub górę. We Float16 i Float64 (odpowiednio t-1=10 i t-1=52) parzystość jest inna niż we Float32 (t-1=23), stąd różnica, co do znaku.

3.3 Zadanie 3 - Równomierne rozmieszczenie liczb

Wykonując wszystkie działania zawarte w punkcie 2.3 sprawozdania, doszedłem do wniosku, że dla kroku $\partial = 2^{-52}$:

- w przedziale [1,2] - ∂ zwiększa zapis bitowy o bit
- w przedziale [0,5; 1] - ∂ zwiększa zapis bitowy o 2 bity
- w przedziale [2,4] - 2∂ zwiększa zapis bitowy o bit
- w przedziale $[2^n, 2^{n+1}]$ - ∂ zwiększa zapis bitowy o 2^{-n} bity

Zatem liczby we Float64 nie są równomiernie rozmieszczone. Czym odległość od zera się zwiększa, tym rozmieszczone są rzadziej.

3.4 Zadanie 4 - "x*(1/x) ≠ 1"

Wykonując wszystko, co było zawarte w punkcie 2.4 sprawozdania, dochodzimy do wyniku, że najmniejszą liczbą, spełniającą wszystkie warunki zadania jest: 1.000000057228997.

3.5 Zadanie 5 - Iloczyn skalarny dwóch wektorów

Nazwa algorytmu	Float32	Float64
A	-0.4999443	1.0251881368296672e-10
B	-0.4543457	-1.5643308870494366e-10
C	-0.5	0.0
D	-0.5	0.0
Dokładny wynik	-1.00657107000000e-11	

Tabela7: Wyniki z zadania 5

Pierwszym, bardzo oczywistym wnioskiem jest fakt, że wyniki w arytmetyce Float64 są bliższe dokładnemu wynikowi, niż te z Float32.

Drugim wnioskiem, otrzymanym też w zadaniu 1 pierwszej listy z ćwiczeń, jest to, że kolejność wykonywania dodawań ma znaczenie, żeby błąd był jak najmniejszy powinno dodawać się w kolejności rosnącej.

3.6 Zadanie 6 - Funkcje z pierwiastkiem

$\boldsymbol{n}$	$\boldsymbol{f(8^{-n})}$	$\boldsymbol{g(8^{-n})}$	dokładny wynik
1	0.0077822185373186414	0.0077822185373187065	0.0077822185373187065
2	0.00012206286282867573	0.00012206286282875901	0.00012206286282875902
3	1.9073468138230965e-6	1.907346813826566e-6	1.90734681382656e-6
4	2.9802321943606103e-8	2.9802321943606116e-8	2.9802321943606116e-8
5	4.656612873077393e-10	4.6566128719931904e-10	4.6566128719931904e-10
6	7.275957614183426e-12	7.275957614156956e-12	7.275957614156956e-12
7	1.1368683772161603e-13	1.1368683772160957e-13	1.1368683772160957e-13
8	1.7763568394002505e-15	1.7763568394002489e-15	1.7763568394002489e-15
9	0.0	2.7755575615628914e-17	2.7755575615628914e-17
10	0.0	4.336808689942018e-19	4.336808689942018e-19
11	0.0	6.776263578034403e-21	6.776263578034403e-21
12	0.0	1.0587911840678754e-22	1.0587911840678754e-22
13	0.0	1.6543612251060553e-24	1.6543612251060553e-24
14	0.0	2.5849394142282115e-26	2.5849394142282115e-26
15	0.0	4.0389678347315804e-28	4.0389678347315804e-28

Tabela8: Wyniki z zadania 6

Po dokonaniu niezbędnych obliczeń, znajdujących się w punkcie 2.6 sprawozdania, można zauważyć, że zdecydowanie bliższe wyniki otrzymuje się z funkcji g, a funkcja f już dla $x = 8^{-9} = 0.0$.

2.7 Zadanie 7 - Pochodna funkcji

```
julia> blad()
Błąd dla h=2^(-0): ð = 1.9010469435800585
Błąd dla h=2^(-1): ð = 1.753499116243109
Błąd dla h=2^(-2): ð = 0.9908448135457593
Błąd dla h=2^(-3): ð = 0.5062989976090435
...
Błąd dla h=2^(-23): ð = 4.807086915192826e-7
Błąd dla h=2^(-24): ð = 2.394961446938737e-7
Błąd dla h=2^(-25): ð = 1.1656156484463054e-7
Błąd dla h=2^(-26): ð = 5.6956920069239914e-8
Błąd dla h=2^(-27): ð = 3.460517827846843e-8
Błąd dla h=2^(-28): ð = 4.802855890773117e-9
Błąd dla h=2^(-29): ð = 5.480178888461751e-8
Błąd dla h=2^(-30): ð = 1.1440643366000813e-7
Błąd dla h=2^(-31): ð = 1.1440643366000813e-7
Błąd dla h=2^(-32): ð = 3.5282501276157063e-7
Błąd dla h=2^(-33): ð = 8.296621709646956e-7
Błąd dla h=2^(-34): ð = 8.296621709646956e-7
Błąd dla h=2^(-35): ð = 2.7370108037771956e-6
Błąd dla h=2^(-36): ð = 1.0776864618478044e-6
Błąd dla h=2^(-37): ð = 1.4181102600652196e-5
...
Błąd dla h=2^(-48): ð = 0.023192281688538152
Błąd dla h=2^(-49): ð = 0.008057718311461848
Błąd dla h=2^(-50): ð = 0.11694228168853815
Błąd dla h=2^(-51): ð = 0.11694228168853815
Błąd dla h=2^(-52): ð = 0.6169422816885382
Błąd dla h=2^(-53): ð = 0.11694228168853815
Błąd dla h=2^(-54): ð = 0.11694228168853815
```

Listing26: Fragment wyniku funkcji *blad()* z programu zad7.jl

Z listingu 26 widać, że najmniejszy błąd ∂ jest dla $h = 2^{-28}$ i wynosi on: $\partial = 4.802855890773117 \times 10^{-9}$ jest to spowodowane tym, że zmniejszenia h przybliżają wartość funkcji, ale równocześnie tracimy coraz więcej na przybliżeniu wyników. Wynik ten jest zatem jest pośrodku tych dwóch procesów.