# Obliczenia Naukowe

Laboratorium 1

Bartosz Grelewski

October 23, 2022

# 1 Zadanie 1

## 1.1 Podpunkt 1

### 1.1.1 Opis Problemu

W tym zadaniu naszym celem było wyznaczenie iteracyjnie przy użyciu języka Julia epsilonu maszynowego dla niektórych wartości naszej arytmetyki. Epsilon jest odległością od 1 do najmniejszej liczby większej od 1 możliwej do zapisania w danej arytmetyce. Liczba 1 ma w zapisie dwójkowym mantysę wypełnioną zerami. Czyli najmniejsza liczba spełniająca fl(1.0 + macheps) > 1.0. Dla typów zmiennoprzecinkowych float16, float32 i float64.

### 1.1.2 Rozwiązanie

Aby wygenerować dane służące później do porównania z gotowymi funkcjami eps(float16) itd. należy zastosować algorytm generujący nam owe wartości iteracyjnie.

```
# liczenie macheps
function Macheps_Algorithm(type)
    epsilon = type(1.0)
    while type(1.0 + 0.5*epsilon) > type(1.0)
        epsilon = type(0.5 * epsilon)
    end
    return epsilon
end
```

Algorytm działa w taki sposób, że sprawdzanie machepsów zaczynamy od najmłodszego wyniku 1, zaś kolejne liczby będziemy generować poprzez iloraz 2, co w zapisie binarnym odpowiada przesuwaniu jedynej jedynki w mantysie coraz bardziej w prawo. W algorytmie **type** oznacza arytmetykę.

### 1.1.3 Interpretacja wyniku

| Typ danych | Algorytm | funkcja eps | float.h |
|---|---|---|---|
| Float16 | 0.000977 | 0.000977 | nie ma |
| Float32 | 1.1920929e-7 | 1.1920929e-7 | 1.19209e-07 |
| Float64 | 2.220446049250313e-16 | 2.220446049250313e-16 | 2.22045e-16 |

## 1.2 Podpunkt 2

### 1.2.1 Opis problemu

Kolejny podpunkt to program w języku Julia wyznaczający iteracyjnie liczbę maszynową eta taką, że eta > 0.0 dla wszystkich typów zmiennopozycyjnych Float16, Float32, Float64, zgodnych ze standardem IEEE 754 (half, single, double), i porównać z wartościami zwracanymi przez funkcje: nextfloat(Float16(0.0)), nextfloat(Float32(0.0)), nextfloat(Float64(0.0)). Szukamy możliwie najbliższej liczby większej od 0 w zapisie binarnym.

### 1.2.2 Rozwiązanie

Stosujemy przerobiony algorytm z punktu powyżej. Jedyna różnica to brak dodawania 0.5 i zaczynamy od wartości 1.0 z iloczynem eta w całości większe od 0 w pętli while.

```
function eta(type)
    eta = type(1.0)
    while type(0.5*eta) > 0
        eta = type(0.5 * eta)
    end
    return eta
end
```

### 1.2.3 Interpretacja wyniku

| Typ danych | Algorytm | eta |
|---|---|---|
| Float16 | 0.6.0e-8 | 0.6.0e-8 |
| Float32 | 1.0e-45 | 1.0e-45 |
| Float64 | 5.0e-324 | 5.0e-324 |

## 1.3 Podpunkt 3

### 1.3.1 Opis problemu

Napisać program w języku Julia wyznaczający iteracyjnie liczbę (MAX) dla wszyst- kich typów zmiennopozycyjnych Float16, Float32, Float64, zgodnych ze standardem IEEE 754 (half, single, double), i porównać z wartościami zwracanymi przez funkcje: floatmax(Float16), floatmax(Float32), floatmax(Float64) oraz z danymi zawartymi w pliku nagłówkowym float.h dowolnej instalacji języka C.

### 1.3.2 Rozwiązanie

Piszemy algorytm generujący nam iteracyjnie liczbę MAX dla wszystki typów zmiennopozycyjnych. Posługujemy się funkcjami takimi jak **!isinf()** czy już wcześniej napisaną **eps**.

```
function find_max(type)
    max = type(1.0)

    while !isinf(max * T(2.0))
        max *= type(2.0)
    end
    max *= (type(2.0) - eps(type))
    return max
end
```

Na początku ustawiamy nasze MAX jako arytmetykę pod zmienną **type**. Za pomocą funkcji **isinf** sprawdzamy, czy kolejna inkrementacja nie spowoduje utworzenia liczby spoza dostępnego zakresu. Jest to warunek końcowy pętli, więc później przemnażamy przez *(type(2.0) - eps(type))*.

### 1.3.3 Interpretacja wyniku

| Typ danych | Algorytm | funkcja floatmax | float.h |
|---|---|---|---|
| Float16 | 6.55e4 | 6.55e4 | 6.55e4 |
| Float32 | 3.4028235e38 | 3.4028235e38 | 3.4028235e38 |
| Float64 | 1.7976931348623157e308 | 1.7976931348623157e308 | 1.7976931348623157e308 |

## 1.4 Wnioski

W podpunkcie 1 metoda iteracyjna epsilonu jest jes poprawna ze względu na wyniki skonfrontowane z prawidłowymi wartościami. Im większa precyzja arytmetyki tym mniejszy jest epsilon maszynowy. W podpunkcie 2 liczba eta jest najmniejszą możliwą do zapisania liczbą dodatnią. Algorytm iteracyjny jest poprawny, ponieważ wyniki pokrywają się z wartościami wbudowanych metod. W podpunkcie 3 wzrost wartości MAX wraz ze wziększeniem się precyzji arytmetyki wynika z faktu, że im większa jest precyzja tym więcej liczb możemy zapisać.

### 1.4.1 Odpowiedzi na pytania

*Jaki związek ma liczba macheps z precyzją arytmetyki?*

Wraz ze wzrostem precyzji arytmetyki maleje wartość *macheps*.

*Jaki związek ma liczba eta z liczbą $MIN_{sub}$?*

Wraz ze wzrostem prezycji arytmetyki maleje wartość *eta*. Liczba *eta* ma związek z liczbą $MIN_{sub}$. Jak porównamy sobie np. arytmetykę float32 z float64 to zauważymy, że jest to liczba zdenormalizowana.

*Co zwracają funkcje floatmin(Float32) i floatmin(Float64) i jaki jest związek zwraca- nych wartości z liczbą $MIN_{nor}$*

# 2 Zadanie 2

## 2.1 Opis Problemu

Kahan stwierdził, że epsilon maszynowy (macheps) można otrzymać obliczając wyrażenie

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

w arytmetyce zmiennopozycyjnej. Sprawdzić eksperymentalnie w języku Julia słuszność tego stwierdzenia dla wszystkich typów zmiennopozycyjnych Float16, Float32, Float64.

## 2.2 Rozwiązanie

Program polega na użyciu funkcji one do otrzymania 1 w danej arytmetyce, a następnie podstawienia do wzoru przy uwzględnienia faktu korzystania z liczb opartych na tej funkcji.

```
function kahan(type)
    type(3.0) * (type(4.0/3.0) - type(1.0)) - type(1.0)
end
```

## 2.3 Interpretacja wyniku

| Typ danych | kahan | eps |
|---|---|---|
| Float16 | -0.000977 | 0.000977 |
| Float32 | 1.1920929e-7 | 1.1920929e-7 |
| Float64 | -2.220446049250313e-16 | 2.220446049250313e-16 |

## 2.4 Wnioski

Wartość bezwzględna wyników zgadza się z epsilonem zwracanym przez funkcję języka. Zmiana znaku spowodowana jest możliwą ilością bitów np. dla Float32 ostatnią cyfrą mantysy będzie -1, a Float16 0. Czyli dla wszystkich typów będzie działać jeśli nałożymy na całe równanie wartość bezwzględną.

# 3 Zadanie 3

## 3.1 Opis problemu

Sprawdzenie czy w przedziałach liczbowych liczby zmiennopozycyjne są rozmieszczone równomiernie.

## 3.2 Rozwiązanie

Aby sprawdzić czy liczby są rozmieszczone równomiernie sprawdzam wykładnik, a odległości między liczbami odejmując a od prevfloat(a) gdzie a jest dowolną liczbą zmiennopozycyjną z danego przedziału.

### 3.3 Wyniki

Liczby w przedziale [1, 2] są rozmieszczone w odstępach $2^{-52}$, w przedziale $[\frac{1}{2}, 1]$ $2^{-53}$, a w przedziale [2, 4] $2^{-51}$.

### 3.4 Wnioski

Im blizej 0 tym gęściej są rozstawione liczby zmiennopozycyjne.

## 4 Zadanie 4

### 4.1 Opis problemu

Znajdź eksperymentalnie w arytmetyce Float64 zgodnej ze standardem IEEE 754 (double) liczbę zmiennopozycyjną x w przedziale 1 < x < 2, taką, że

$$x \cdot \frac{1}{x} \neq 1$$

tj. fl(xfl(1/x)) = 1 (napisz program w języku Julia znajdujący tę liczbę). Znajdź najmniejszą taką liczbę.

### 4.2 Rozwiązanie

```
#Finding the smallest number
function lower(y)
    while Float64(y*(Float64(1.0)/y)) == Float64(1.0) && y < Float64(2.0)
        y = nextfloat(y)
    end
    println("The smallest x which fulfill the equation is: $(y)")
end

function main()
    y = nextfloat(Float64(1.0))
    lower(y)
end
main()
```

Tak jak w treści polecenia, wyznaczyłem eksperymentalnie za pomocą arytmetyki Float64 liczbę, która spełnia założenia zadania. Od razu wyznaczyłem najmniejszą wartość.

### 4.3 Interpretacja wyniku

Wynosi ona 1.000000057228997.

### 4.4 Wnioski

Nasz wynik nie wyszedł równo 1. Spowodowane jest to błędem zaokrągleń, które powstały na skutek operacji obliczeniowych. Arytmetyka ma skończoną dokładność dlatego też nie zawsze zwraca dobre wyniki.

# 5 Zadanie 5

## 5.1 Opis problemu

Napisz 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].
Należało zaimplementować 4 podane algorytmy:
1. *"W przód."*
2. *"W tył."*
3. *"Od największego do najmniejszego."*
4. *"Od najmniejszego do największego."*

## 5.2 Rozwiązanie

Algorytm pierwszy obejmuje policzenie sigmy od 1 do n z iloczynu. **Type** - typ Float32 albo Float64. Pętla wykonuje się od 1 do długości pierwszego wektora (x). W pętli natomiast następuje suma iloczynu dwóch elementów wektorów w pozycji zależnej od iteratora (i).

```
function forward(x, y, type)
    sum = type(0.0)
    for i in 1:length(x)
        sum += x[i] * y[i]
    end
    return sum
end
```

Alogrytm drugi oblicza iloczyn skalarny dwóch wektorów x i y, względem poprzedniego algorytmu zmienia się tylko sigma, która jest teraz od n do 1.

```
function backward(x, y, type)
    sum = type(0.0)
    #tzn. ze odejmujemy 1 co iteracje wiec iterujemy
    #do 1 ale odejmujac 1 od dlugosci x
    for i in length(x):-1:1
        sum += x[i] * y[i]
    end
    return sum
end
```

Algorytm trzeci mnoży wartości o odpowiadających indeksach. Następnie tworzy *partial sum* wyników dodatnich dodawanych w kolejności rosnącej. Zwraca sumę sum.

```
function partial_desc(x, y)
    #mnozenie tablic
    p = x .* y

    sumOfPositiveNumbers = sum(sort(filter(a -> a>0, p), rev=true))
    sumOfNegativeNumbers = sum(sort(filter(a -> a<0, p)))
    return sumOfPositiveNumbers+sumOfNegativeNumbers
end
```

Ostatni algorytm analogiczny do trzeciego, ale sumy częściowe tworzone są w odwrotnym porządku.

```
function partial_asc(x, y)
    p = x .* y
    sumOfPositiveNumbers = sum(sort(filter(a -> a>0, p)))
    sumOfNegativeNumbers = sum(sort(filter(a -> a<0, p), rev=true))
    return sumOfPositiveNumbers+sumOfNegativeNumbers
end
```

## 5.3 Interpretacja wyniku

Wyniki dla Float32

| Algorytm | Wynik |
|---|---|
| 1 | -0.4999443 |
| 2 | -0.4543457 |
| 3 | -0.5 |
| 4 | -0.5 |

Wyniki dla Float64

| Algorytm | Wynik |
|---|---|
| 1 | 1.0251881368296672e-10 |
| 2 | -1.5643308870494366e-10 |
| 3 | 0.0 |
| 4 | 0.0 |

## 5.4 Wnioski

Konfrontując uzyskane wyniki z faktycznym prawidłowym wynikiem:

$$-1.00657107000000 \cdot 10^{-11}$$

nie uzyskaliśmy żadnego poprawnego wyniku. Na błędy obliczeniowe ma wpływ nie tylko arytmetyka. Również kolejność wykonywania działań.

# 6 Zadanie 6

## 6.1 Opis problemu

Funkcje równe sobie daja inne wyniki.

## 6.2 Rozwiazanie

Implementacja funkcji f i g podanych w treści zadania.

## 6.3 Wyniki

$8^{-1}$ : $g(x) = 0.0077822185373187065$, $f(x) = 0.0077822185373186414$
$8^{-2}$ : $g(x) = 0.00012206286282875901$, $f(x) = 0.00012206286282867573$
$8^{-3}$ : $g(x) = 1.907346813826566e-6$, $f(x) = 1.9073468138230965e-6$
$8^{-4}$ : $g(x) = 2.9802321943606116e-8$, $f(x) = 2.9802321943606103e-8$
$8^{-5}$ : $g(x) = 4.6566128719931904e-10$, $f(x) = 4.656612873077393e-10$
$8^{-6}$ : $g(x) = 7.275957614156956e-12$, $f(x) = 7.275957614183426e-12$
$8^{-7}$ : $g(x) = 1.1368683772160957e-13$, $f(x) = 1.1368683772161603e-13$
$8^{-8}$ : $g(x) = 1.7763568394002489e-15$, $f(x) = 1.7763568394002505e-15$
$8^{-9}$ : $g(x) = 2.7755575615628914e-17$, $f(x) = 0.0$
$8^{-10}$ : $g(x) = 4.336808689942018e-19$, $f(x) = 0.0$

## 6.4 Wnioski

Bardziej wiarygodna jest funkcja g, poniewaz liczby im są dalsze od 0 tym tracą mniej znaczących bitów.

# 7 Zadanie 7

## 7.1 Opis problemu

Obliczanie pochodnej i przyblizonej pochodnej oraz róznicy miedzy nimi.

## 7.2 Rozwiazanie

Obliczenie pochodnej z definicji - przyblizona, oraz obliczenie funkcji pochodnej.

## 7.3 Wyniki

n = 0 approximate derivative = 2.0179892252685967 derivative = 0.11694228168853815 difference = -1.9010469435800585
n = 1 approximate derivative = 1.8704413979316472 derivative = 0.11694228168853815 difference = -1.753499116243109
n = 2 approximate derivative = 1.1077870952342974 derivative = 0.11694228168853815 difference = -0.9908448135457593
n = 3 approximate derivative = 0.6232412792975817 derivative = 0.11694228168853815 difference = -0.5062989976090435

n = 45 approximate derivative = 0.11328125 derivative = 0.11694228168853815 difference = 0.003661031688538152
n = 46 approximate derivative = 0.109375 derivative = 0.11694228168853815 difference = 0.007567281688538152
n = 47 approximate derivative = 0.109375 derivative = 0.11694228168853815 difference = 0.007567281688538152
n = 48 approximate derivative = 0.09375 derivative = 0.11694228168853815 difference = 0.023192281688538152
n = 49 approximate derivative = 0.125 derivative = 0.11694228168853815 difference = -0.008057718311461848
n = 50 approximate derivative = 0.0 derivative = 0.11694228168853815 difference = 0.11694228168853815
n = 51 approximate derivative = 0.0 derivative = 0.11694228168853815 difference = 0.11694228168853815
n = 52 approximate derivative = -0.5 derivative = 0.11694228168853815 difference = 0.6169422816885382
n = 53 approximate derivative = 0.0 derivative = 0.11694228168853815 difference = 0.11694228168853815
n = 54 approximate derivative = 0.0 derivative = 0.11694228168853815 difference = 0.11694228168853815

## 7.4 Wnioski

Podobnie jak w poprzednim zadaniu - im blizej zera tym są większe błędy.