# Sprawozdanie z listy 1. - Obliczenia naukowe

Jakub Bachanek

25 października 2020

## 1 Zadanie 1.

### 1.1 Opis problemu

Celem zadania jest iteracyjne wyznaczenie następujących liczb:

- epsilon maszynowy *macheps*
- liczba maszynowa *eta*
- liczba maszynowa *MAX*

dla typów `Float16`, `Float32` oraz `Float64`, a następnie porównanie ich z wartościami zwracanymi przez funkcje wbudowane języka `Julia` oraz danymi zawartymi w pliku nagłówkowym `float.h`.

### 1.2 Rozwiązanie

*Macheps* wyznaczamy iteracyjnie poprzez dzielenie jedynki przez 2 dopóki warunek jest spełniony.

```
function calc(type)
    macheps = type(1.0)
    previous_macheps = type(1.0)

    while type(1.0 + macheps) > 1
        previous_macheps = macheps
        macheps = macheps / 2
    end

    println("My result macheps ", type, ": ",  previous_macheps)
end
```

Liczba *macheps* jest dwa razy większa od precyzji arytmetyki $\epsilon$, ponieważ jeśli odległość między dwoma kolejnymi liczbami maszynowymi wynosi *macheps*, to maksymalny błąd reprezentacji liczby w danej arytmetyce jaki można popełnić, to połowa tej odległości.

Liczbę maszynową *eta* wyznaczamy iteracyjnie poprzez dzielenie jedynki przez 2 dopóki *eta* jest większa od zera.

```
function calc(type)
    previous_eta = type(1.0)
    eta = type(1.0)

    while eta > 0
        previous_eta = eta
        eta = eta / 2
    end

    println("My result eta ", type, ": ",  previous_eta)
end
```

Liczba *eta* jest równa liczbie $MIN_{sub}$, ponieważ *eta* jest największą liczbą większą od zera, która ma reprezentację w danej arytmetyce.

Funkcje floatmin(Float32) oraz floatmin(Float64) zwracają najmniejszą, znormalizowaną dodatnią liczbę reprezentowaną w danej arytmetyce. Jest ona równa liczbie $MIN_{nor}$.

Liczbę maszynową $MAX$ wyznaczamy iteracyjnie poprzez mnożenie liczby maszynowej poprzedzającej jedynkę (wtedy mantysa składa się z samych jedynek) przez 2 dopóki $MAX$ jest mniejszy od nieskończoności.

```
function calc(type)
    max = prevfloat(type(1.0))
    previous_max = max

    while !isinf(max)
        previous_max = max
        max = max * 2
    end

    println("My result max ", type, ": ", previous_max)
end
```

## 1.3 Wyniki i interpretacja

Poniższe tabele przedstawiają uzyskane przeze mnie wyniki.

| Typ | My result *macheps* | `eps(type)` | `float.h` |
|---|---|---|---|
| Float16 | 0.000977 | 0.000977 | — |
| Float32 | 1.1920929e-7 | 1.1920929e-7 | 1.1920928955e-07 |
| Float64 | 2.220446049250313e-16 | 2.220446049250313e-16 | 2.2204460493e-16 |

| Typ | My result *eta* | `nextfloat(type(0.0))` |
|---|---|---|
| Float16 | 6.0e-8 | 6.0e-8 |
| Float32 | 1.0e-45 | 1.0e-45 |
| Float64 | 5.0e-324 | 5.0e-324 |

| Typ | My result *MAX* | `floatmax(type)` | `float.h` |
|---|---|---|---|
| Float16 | 6.55e4 | 6.55e4 | — |
| Float32 | 3.4028235e38 | 3.4028235e38 | 3.4028234664e+38 |
| Float64 | 1.7976931348623157e308 | 1.7976931348623157e308 | 1.7976931349e+308 |

Uzyskane przeze mnie wyniki zgadzają się z tymi zwracanymi przez wbudowane funkcje języka `Julia` oraz zawartymi w pliku `float.h`.

### 1.4 Wnioski

Metody iteracyjne obliczania *macheps*, *eta* oraz *MAX* okazały się prawidłowe. Arytmetyka zmiennoprzecinkowa niesie ze sobą szereg ograniczeń, o których należy zawsze pamiętać.

## 2 Zadanie 2.

### 2.1 Opis problemu

Celem zadania jest sprawdzenie poprawności stwierdzenia Kahana, że epsilon maszynowy można otrzymać obliczając wyrażenie:

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

w arytmetyce zmiennoprzecinkowej.

### 2.2 Rozwiązanie

Obliczamy wyrażenie prostą funkcją dla każdego typu.

```
function calc(type)
    value = type(3.0) * (type(4.0)/type(3.0) - type(1.0)) - type(1.0)

    println("My result ", type, ": ",  value)
end
```

### 2.3 Wyniki i interpretacja

Poniższa tabela przedstawia uzyskany wynik.

| Typ | My result *macheps* | `eps(type)` |
|---|---|---|
| Float16 | -0.000977 | 0.000977 |
| Float32 | 1.1920929e-7 | 1.1920929e-7 |
| Float64 | -2.220446049250313e-16 | 2.220446049250313e-16 |

Wartości się zgadzają z dokładnością do wartości bezwzględnej. Różnice wynikają z reprezentacji nieskończonego rozwinięcia liczby $\frac{4}{3}$ na parzystej, bądź nieparzystej liczbie bitów mantysy tych typów.

### 2.4 Wnioski

Z uwagi na skończoną dokładność arytmetyki zmiennoprzecinkowej, wyniki obliczeń mogą być różne od naszych oczekiwań.

## 3 Zadanie 3.

### 3.1 Opis problemu

Celem zadania jest sprawdzenie rozmieszczenia liczb zmiennoprzecinkowych w arytmetyce `Float64` w przedziałach $[\frac{1}{2}, 1]$, $[1, 2]$, $[2, 4]$.

### 3.2 Rozwiązanie

Za pomocą funkcji `bitstring` możemy zobaczyć jak wyglądają bity liczby w arytmetyce `Float64`

Dla przedziału $[1, 2]$:

```
bitstring(Float64(1.0))
0011111111110000000000000000000000000000000000000000000000000000

bitstring(nextfloat(Float64(1.0)))
0011111111110000000000000000000000000000000000000000000000000001

bitstring(prevfloat(Float64(2.0)))
0011111111111111111111111111111111111111111111111111111111111111

bitstring(Float64(2.0))
0100000000000000000000000000000000000000000000000000000000000000
```

Dla przedziału $[\frac{1}{2}, 1]$:

```
   bitstring(Float64(0.5))
0011111111100000000000000000000000000000000000000000000000000000

   bitstring(nextfloat(Float64(0.5)))
0011111111100000000000000000000000000000000000000000000000000001

   bitstring(prevfloat(Float64(1.0)))
0011111111101111111111111111111111111111111111111111111111111111

   bitstring(Float64(1.0))
0011111111110000000000000000000000000000000000000000000000000000
```

Dla przedziału $[2, 4]$:

```
   bitstring(Float64(2.0))
0100000000000000000000000000000000000000000000000000000000000000

   bitstring(nextfloat(Float64(2.0)))
0100000000000000000000000000000000000000000000000000000000000001

   bitstring(prevfloat(Float64(4.0)))
0100000000001111111111111111111111111111111111111111111111111111

   bitstring(Float64(4.0))
0100000000010000000000000000000000000000000000000000000000000000
```

Liczba bitów mantysy w `Float64` jest stała i wynosi 52. Za ich pomocą możemy wyrazić $2^{52}$ różnych liczb, które gdy cecha jest stała, są równoodległe. Zatem krok między liczbami o tej samej cesze wynosi $2^{c-1023} * 2^{-52}$

## 3.3 Wyniki i interpretacja

Każda liczba zmiennopozycyjna w `Float64` między $\frac{1}{2}$ i 1 może być przedstawiona jako $x = \frac{1}{2} + k\delta$, gdzie $k = 1, 2, ..., 2^{52} - 1$ i $\delta = 2^{-53}$.

Podobnie dla przedziału $[2, 4]$, tylko wtedy $\delta = 2^{-51}$.

## 3.4 Wnioski

Rozkład liczb maszynowych jest równomierny dla liczb o takiej samej cesze. Krok między kolejnymi liczbami wzrasta natomiast wraz z cechą.

# 4 Zadanie 4.

## 4.1 Opis problemu

Celem zadania jest znalezienie w arytmetyce `Float64` liczby zmiennoprzecinkowej $x$ w przedziale $(1, 2)$ takiej, że $x * \frac{1}{x} \neq 1$. Następnie należy znaleźć najmniejszą taką liczbę.

## 4.2 Rozwiązanie

Iterujemy po kolejnych liczbach, aż znajdziemy taką która nie spełnia warunku.

```
function search_number(num)
    x = nextfloat(Float64(num))

    while x * (Float64(1.0) / x) == Float64(1.0)
        x = nextfloat(x)
    end

    println("My result = ", x)
end

search_number(Float64(1.0))
search_number(Float64(0.0))
```

## 4.3 Wyniki i interpretacja

Tabela przedstawia otrzymaną liczbę dla a) $1 < x < 2$ oraz b) najmniejszą taką, że $x > 0$

| Punkt | My result $x$ |
|---|---|
| a | 1.000000057228997 |
| b | 5.0e-324 |

## 4.4 Wnioski

Przez niedokładne reprezentowanie liczb w arytmetyce `Float64`, obliczając nawet proste wyrażenia możemy otrzymać błędne wyniki.

# 5 Zadanie 5.

## 5.1 Opis problemu

Celem zadania jest obliczenie iloczynu skalarnego dwóch wektorów w arytmetyce `Float32` oraz `Float64` za pomocą czterech różnych algorytmów sumowania:

- "w przód"
- "w tył"
- od największego do najmniejszego
- od najmniejszego do największego

## 5.2 Rozwiązanie

Poniższe funkcje odpowiadają kolejnym algorytmom sumowania.

```
function sum_a(x, y, type)
    s::type = 0.0

    for i = 1:length(x)
        s = s + type(x[i]) * type(y[i])
    end

    return s
end


function sum_b(x, y, type)
    s::type = 0.0

    for i = length(x):-1:1
        s = s + type(x[i]) * type(y[i])
    end

    return s
end


function sum_c(x, y, type)
    positive = zeros(type, 0)
    negative = zeros(type, 0)

    for i = 1:length(x)
        result::type = type(x[i]) * type(y[i])

        if result > 0.0
            append!(positive, result)
        else
            append!(negative, result)
        end
    end
```

```
    part_sum_positive = foldl(+, sort(positive, rev=true))
    part_sum_negative = foldl(+, sort(negative))

    return part_sum_positive + part_sum_negative
end


function sum_d(x, y, type)
    positive = zeros(type, 0)
    negative = zeros(type, 0)

    for i = 1:length(x)
        result::type = type(x[i]) * type(y[i])

        if result > 0.0
            append!(positive, result)
        else
            append!(negative, result)
        end
    end

    part_sum_positive = foldl(+, sort(positive))
    part_sum_negative = foldl(+, sort(negative, rev=true))

    return part_sum_positive + part_sum_negative
end
```

### 5.3 Wyniki i interpretacja

Poniższa tabela przedstawia uzyskane wyniki w poszczególnych algorytmach.

| Punkt | Result `Float32` | Result `Float64` |
|---|---|---|
| a | -0.4999443 | 1.0251881368296672e-10 |
| b | -0.4543457 | -1.5643308870494366e-10 |
| c | -0.5 | 0.0 |
| d | -0.5 | 0.0 |

Prawidłowa wartość = -1.00657107000000e-11.

Żaden z algorytmów w tych arytmetykach nie obliczył prawidłowej wartości. W obu arytmetykach nastąpiła utrata dokładności ze względu na błędy zaokrągleń, ponadto w przykładach c) oraz d) nastąpiła redukcja cyfr znaczących.

### 5.4 Wnioski

Zarówno kolejność obliczeń jak i same dane znacząco wpływają na dokładność wyników. Zawsze należy pamiętać o zjawisku redukcji cyfr znaczących i o tym,

że arytmetyka zmiennoprzecinkowa nie jest przemienna.

# 6 Zadanie 6

## 6.1 Opis problemu

Celem zadania jest obliczanie wartości poniższych funkcji dla kolejnych wartości argumentu $x = 8^{-1}, 8^{-2}, 8^{-3}, ....$

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

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

## 6.2 Rozwiązanie

Iteracyjnie wyznaczamy wartości funkcji dla kilkunastu kolejnych argumentów i obserwujemy jak te wartości się zmieniają.

```
function f(x::Float64)
    return sqrt(x^2 + 1) - 1
end

function g(x::Float64)
    return x^2 / (sqrt(x^2 + 1) + 1)
end


function calc(n)
    functions = [f, g]

    for func in functions
        for i = 1:n
            x = Float64(1/(8^i))
            println("Function ", func, "(8^(-", i, ")) = ", func(x))
        end
        println()
    end
end

calc(13)
```

## 6.3 Wyniki i interpretacja

Poniższa tabela przedstawia wyniki obliczeń.

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

Już od $n = 9$ zauważamy, że dla funkcji $f$ nastąpił znaczny spadek dokładności, natomiast $g$ nadal podaje wiarygodne wyniki. Jest to spowodowane operacją odejmowania w funkcji $f$, ponieważ podczas niej dochodzi do redukcji cyfr znaczących. W przypadku funkcji $g$ to zjawisko nie występuje.

### 6.4 Wnioski

Chcąc otrzymywać jak najbardziej dokładne wyniki obliczeń należy unikać między innymi odejmowania liczb o bardzo zbliżonych wartościach, ponieważ wtedy możemy spowodować redukcję cyfr znaczących.

## 7 Zadanie 7.

### 7.1 Opis problemu

Celem zadania jest obliczenie przybliżonej wartości pochodnej funkcji $f(x) = \sin x + \cos 3x$ w punkcie $x_0 = 1$ za pomocą wzoru:

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

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

### 7.2 Rozwiązanie

Iteracyjnie obliczamy wartości ze wzorów dla kolejnych $n$.

```
function calc_derivative(f, x0::Float64, h::Float64)
    return (f(x0 + h) - f(x0)) / h
end
```

```
function f(x::Float64)
    return sin(x) + cos(3 * x)
end

function f_der(x::Float64)
    return cos(x) - 3 * sin(3 * x)
end

function calc()
    for n = 0:54
        h = Float64(2.0^(-n))
        println("Derivative value (for n = ", n, ") = ", calc_derivative(f, 1.0, h))
        println("Error (for n = ", n, ") = ", abs(calc_derivative(f, 1.0, h) - f_der(1.0)))
        println()
    end
end

calc()
```

### 7.3 Wyniki i interpretacja

Poniższa tabela przedstawia obliczone wyniki dla kolejnych $n$.

| n | $\tilde{f}'(x_0)$ | $\lvert f'(x_0) - \tilde{f}'(x_0)\rvert$ |
|---|---|---|
| 0 | 2.0179892252685967 | 1.9010469435800585 |
| 1 | 1.8704413979316472 | 1.753499116243109 |
| 2 | 1.1077870952342974 | 0.9908448135457593 |
| 3 | 0.6232412792975817 | 0.5062989976090435 |
| 4 | 0.3704000662035192 | 0.253457784514981 |
| 5 | 0.24344307439754687 | 0.1265007927090087 |
| 6 | 0.18009756330732785 | 0.0631552816187897 |
| 7 | 0.1484913953710958 | 0.03154911368255764 |
| 8 | 0.1327091142805159 | 0.015766832591977753 |
| 9 | 0.1248236929407085 | 0.007881411252170345 |
| 10 | 0.12088247681106168 | 0.0039401951225235265 |
| 11 | 0.11891225046883847 | 0.001969968780300313 |
| 12 | 0.11792723373901026 | 0.0009849520504721099 |
| 13 | 0.11743474961076572 | 0.0004924679222275685 |
| 14 | 0.11718851362093119 | 0.0002462319323930373 |
| 15 | 0.11706539714577957 | 0.00012311545724141837 |
| 16 | 0.11700383928837255 | 6.155759983439424e-5 |
| 17 | 0.11697306045971345 | 3.077877117529937e-5 |
| 18 | 0.11695767106721178 | 1.5389378673624776e-5 |
| 19 | 0.11694997636368498 | 7.694675146829866e-6 |

| | | |
|---|---|---|
| 20 | 0.11694612901192158 | 3.8473233834324105e-6 |
| 21 | 0.1169442052487284 | 1.9235601902423127e-6 |
| 22 | 0.11694324295967817 | 9.612711400208696e-7 |
| 23 | 0.11694276239722967 | 4.807086915192826e-7 |
| 24 | 0.11694252118468285 | 2.394961446938737e-7 |
| 25 | 0.116942398250103 | 1.1656156484463054e-7 |
| 26 | 0.11694233864545822 | 5.695692006923991 4e-8 |
| 27 | 0.11694231629371643 | 3.460517827846843e-8 |
| 28 | 0.11694228649139404 | 4.802855890773117e-9 |
| 29 | 0.11694222688674927 | 5.480178888461751e-8 |
| 30 | 0.11694216728210449 | 1.1440643366000813e-7 |
| 31 | 0.11694216728210449 | 1.1440643366000813e-7 |
| 32 | 0.11694192886352539 | 3.5282501276157063e-7 |
| 33 | 0.11694145202636719 | 8.296621709646956e-7 |
| 34 | 0.11694145202636719 | 8.296621709646956e-7 |
| 35 | 0.11693954467773438 | 2.7370108037771956e-6 |
| 36 | 0.116943359375 | 1.0776864618478044e-6 |
| 37 | 0.1169281005859375 | 1.4181102600652196e-5 |
| 38 | 0.116943359375 | 1.0776864618478044e-6 |
| 39 | 0.11688232421875 | 5.9957469788152196e-5 |
| 40 | 0.1168212890625 | 0.0001209926260381522 |
| 41 | 0.116943359375 | 1.0776864618478044e-6 |
| 42 | 0.11669921875 | 0.0002430629385381522 |
| 43 | 0.1162109375 | 0.0007313441885381522 |
| 44 | 0.1171875 | 0.0002452183114618478 |
| 45 | 0.11328125 | 0.003661031688538152 |
| 46 | 0.109375 | 0.007567281688538152 |
| 47 | 0.109375 | 0.007567281688538152 |
| 48 | 0.09375 | 0.023192281688538152 |
| 49 | 0.125 | 0.008057718311461848 |
| 50 | 0.0 | 0.11694228168853815 |
| 51 | 0.0 | 0.11694228168853815 |
| 52 | -0.5 | 0.6169422816885382 |
| 53 | 0.0 | 0.11694228168853815 |
| 54 | 0.0 | 0.11694228168853815 |

Dokładna wartość = 0.11694228168853805

Od pewnego momentu zmniejszanie wartości $h$ nie poprawia przybliżenia wartości pochodnej, ponieważ w liczniku odejmujemy coraz bliższe sobie liczby, co pociąga za sobą redukcję cyfr znaczących. Wartości $1 + h$ są coraz bliżej 1, aż w pewnej chwili ją osiągają - dzieje się to dla $n > 52$. Najlepsze przybliżenie otrzymujemy dla $n = 28$.

## 7.4 Wnioski

Ograniczona dokładność arytmetyki zmiennoprzecinkowej nie pozwala nam obliczyć dokładnych wartości pewnych wyrażeń. Redukcja cyfr znaczących przy odejmowaniu liczb bliskich sobie może bardzo negatywnie wpłynąć na dokładność wyniku.