# Obliczenia naukowe

Sprawozdanie nr 1

Barbara Banaszak
236514

16-10-2018

# 1 Zadanie 1

## 1.1 Opis problemu

W zadaniu mamy wyznaczyć iteracyjnie epsilon maszynowy (ang.macheps), czyli liczbę taką, że $fl(1.0 + macheps) > 1.0$, liczbę $eta$ ($eta > 0.0$) oraz liczbę ($MAX$) dla wszystkich typów zmiennopozycyjnych w standardzie IEEE 754, używając języka Julia oraz porównać wyliczone wartości z funkcjami wbudowanymi i danymi zawartymi w pliku nagłówkowym C.

## 1.2 Rozwiązanie

Aby obliczyć epsilon maszynowy posługujemy się następującym algorytmem:

$macheps \leftarrow 1$
**while** $(1.0 + macheps > 1.0)$ **do**
  $macheps \leftarrow macheps * 0.5$
**end while**

podobnie będziemy wyznaczać liczbę $eta$:

$eta \leftarrow 1$
**while** $(0.0 + eta > 0.0)$ **do**
  $eta \leftarrow eta * 0.5$
**end while**

a także największą liczbę możliwą do zapisania daną precyzją ($MAX$), do wyznaczenia tej liczby użyjemy także wbudowanej funkcji **isinf**:

$max \leftarrow 1$
**while** !**isinf**$(max * 2.0)$ **do**
  $max \leftarrow max * 2$
**end while**
$max \leftarrow max * (2.0 - macheps)$

## 1.3 Wyniki

| *Macheps* | | | |
|---|---|---|---|
| Sposób wyliczenia | Float16 | Float32 | Float64 |
| Interacyjnie | 0.000977 | 1.1920929e-7 | 2.220446049250313e-16 |
| Funkcja **eps** | 0.000977 | 1.1920929e-7 | 2.220446049250313e-16 |
| float.h | - | 1.19209e-07 | 2.22045e-16 |

| *Eta* | | | |
|---|---|---|---|
| Sposób wyliczenia | Float16 | Float32 | Float64 |
| Interacyjnie | 6.0e-8 | 1.0e-45 | 5.0e-324 |
| Funkcja **nextfloat** | 6.0e-8 | 1.0e-45 | 5.0e-324 |

| $MAX$ | | | |
|---|---|---|---|
| Sposób wyliczenia | Float16 | Float32 | Float64 |
| Interacyjnie | 65504.0 | 3.4028235e38 | 1.7976931348623157e308 |
| Funkcja **realmax** | 6.55e4 | 3.4028235e38 | 1.7976931348623157e308 |
| float.h | - | 3.40282e+38 | 1.79769e+308 |

### 1.4 Wnioski

Pierwsze co zauważymy, gdy przyjrzymy się otrzymanym wynikom to, że pokrywają się one z wartościami rzeczywistymi, a więc stosując metodę iteracyjną zostały wyliczone poprawnie. Możemy też zauważyć, że wartość epsilona maszynowego jest powiązana z precyzją arytmetyki, im mniejsza wartość *macheps* tym precyzja arytmetyki będzie większa. Wartość *macheps* jest również ściśle związana z precyzją arytmetyki $2^{-t-1}$, jego wartość jest dokładnie dwa razy większa ($2^{-t}$).

Liczba *eta* natomiast jak wskazuje jej definicja jest to najmniejsza liczba większa od zera możliwa do zapisania w danej precyzji. Gdy popatrzymy na jej zapis bitowy okaże się, że w dowolnej arytmetyce wszystkie bity cechy będą zerami, oznacza to, że jest to liczba zdenormalizowana ($MIN_{sub}$).

## 2 Zadanie 2

### 2.1 Opis problemu

Zadanie polega na dowiedzeniu słuszności twierdzenia Kahana, że epsilon maszynowy możemy otrzymać obliczając w danej arytmetyce wyrażenie $3(\frac{4}{3}-1)-1$.

### 2.2 Rozwiązanie

Wykonujemy podane działanie dla Float16, Float32 i Float64 odpowiednio rzutując liczby.

### 2.3 Wyniki

| Arytmetyka | Float16 | Float32 | Float64 |
|---|---|---|---|
| Wynik | -0.000977 | 1.1920929e-7 | -2.220446049250313e-16 |
| Poprawna wartość | 0.000977 | 1.1920929e-7 | 2.220446049250313e-16 |

### 2.4 Wnioski

Wartości *macheps* zgadzają się dla wszystkich arytmetyk co do wartości bezwzględnej, jednak dla precyzji Float16 i Float64 nie zgadzają się znaki.

# 3 Zadanie 3

## 3.1 Opis problemu

Mamy za zadanie sprawdzić, że w arytmetyce Float64 liczby zmiennopozycyjne są rozmieszczone w przedziale [1,2] równomiernie z krokiem $\delta = 2^{-52}$ oraz wyznaczyć eksperymentalnie rozmieszczenie liczb w przedziałach [$\frac{1}{2}$,1] i [2,4].

## 3.2 Rozwiązanie

Aby sprawdzić rozmieszczenie liczb posłużymy się funkcją **bits** oraz następującym wzorem, którym możemy przedstawiać liczby zmiennopozycyjne $liczba = 1 + k\delta$, gdzie $k = 1, 2, ..., 2^{-52} - 1$:

$\delta \leftarrow 2^{-52}$
**for** $i = 1$ **to** $(2^{52} - 1)$ **do**
  $number \leftarrow 0.5 + i * \delta$
  $bits(number)$
**end for**

## 3.3 Wyniki

| Przedział [1,2] $\delta = 2^{-52}$ |
|---|
| 0011111111110000000000000000000000000000000000000000000000000001 |
| 0011111111110000000000000000000000000000000000000000000000000010 |
| 0011111111110000000000000000000000000000000000000000000000000011 |
| 0011111111110000000000000000000000000000000000000000000000000100 |
| 0011111111110000000000000000000000000000000000000000000000000101 |

| Przedział [$\frac{1}{2}$,1] $\delta = 2^{-53}$ |
|---|
| 0011111111100000000000000000000000000000000000000000000000000001 |
| 0011111111100000000000000000000000000000000000000000000000000010 |
| 0011111111100000000000000000000000000000000000000000000000000011 |
| 0011111111100000000000000000000000000000000000000000000000000100 |
| 0011111111100000000000000000000000000000000000000000000000000101 |

| Przedział [2,4] $\delta = 2^{-51}$ |
|---|
| 0100000000000000000000000000000000000000000000000000000000000001 |
| 0100000000000000000000000000000000000000000000000000000000000010 |
| 0100000000000000000000000000000000000000000000000000000000000011 |
| 0100000000000000000000000000000000000000000000000000000000000100 |
| 0100000000000000000000000000000000000000000000000000000000000101 |

## 3.4 Wnioski

W arytmetyce Float64 liczby z przedziału są rozmieszczone z krokiem $\delta = 2^{-52}$, co sprawdziliśmy korzystając z funkcji **bits** i wyświetlając pokolei liczby z tego przedziału z krokiem $\delta$. Widzimy, że reprezentacja bitowa kolejnych liczb różni się o 1, zatem jest to poprawny krok. Tym samym sposobem wyznaczyliśmy

$\delta = 2^{-53}$ dla przedziału $[\frac{1}{2}$,1] i $\delta = 2^{-51}$ dla przedziału [2,4]. Liczby w kolejnych przedziałach będących potęgami dwójki, są zawsze rozmieszczone z krokiem $\delta = 2^n$, im większy przedział tym n będzie większe. Można także zauważyć, że między kolejnymi potęgami dwójki liczby posiadają tą samą cechę, zmienia się tylko mantysa.

# 4 Zadanie 4

## 4.1 Opis problemu

Mamy za zadanie znaleźć najmniejszą liczbę $x$ w arytmetyce Float64, taką że $1 < x < 2$, która spełnia zależność $x * \frac{1}{x} \neq 1$

## 4.2 Rozwiązanie

Do znalezienia rozwiązania posłużymy się następującym algorytmem:

$x \leftarrow 1.0$
**while** $x < 2.0$ **do**
  **if** $x * \frac{1}{x} \neq 1$ **then**
    **print** x
  **end if**
  $x \leftarrow$ **nextfloat**$(x)$
**end while**

## 4.3 Wyniki

Najmniejszą taką liczbą jest: $x = 1.000000057228997$

## 4.4 Wnioski

Zadanie to pokazuje, że działania w arytmetyce zmiennoprzecinkowej mogą generować błędy spowodowane niedokładnością zaokrągleń, matematycznie przecież dana zależność nie może być spełniona przez żadną liczbę rzeczywistą, a jednak udało nam się taką liczbę znaleźć.

# 5 Zadanie 5

## 5.1 Opis problemu

Zadanie polega na obliczeniu iloczynu skalarnego dwóch wektorów $x = [2.718281828, -3.141592654, 1.414213562$ i $y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$ na cztery różne sposoby i porównanie wyników z poprawnym wynikiem.

## 5.2 Rozwiązanie

Posłużyliśmy się następującymi algorytmami:

1. $\sum_{i=1}^{5} x_i y_i$
2. $\sum_{i=5}^{1} x_i y_i$
3. dodajemy dodatnie liczby w porządku od najmniejszej do największej, ujemne w porządku od najmniejszej do największej, poczym dodajemy do siebie sumy częściowe
4. odwrotnosć punktu 3.

## 5.3 Wyniki

| Sposób | Float32 | Float64 |
|---|---|---|
| 1. | -0.4999443 | 1.0251881368296672e-10 |
| 2. | -0.4543457 | -1.5643308870494366e-10 |
| 3. | 0.0 | 0.0 |
| 4. | -0.5 | -0.5 |

Prawidłowy wynik to $1.00657107000000_{10}11$, jak widać żadnym ze sposobów nie udało się go wyliczyć.

## 5.4 Wnioski

Zadanie to pokazuje, podobnie jak poprzednie, że wykonując działania na liczbach zmiennoprzecinkowych możemy spodziewać się błędów wynikających z niedokładności zaokrąglania. Możemy zauważyć także, że kolejność wykonywania działań ma spore znaczenie, w zależności od wybranego sposobu otrzymywaliśmy różne wyniki. Nie bez znaczenia jest także precyzja arytmetyki, używając większej- Float64 i używając 1. sposobu, otrzymaliśmy wynik najbliższy prawdy.

# 6 Zadanie 6

## 6.1 Opis problemu

Zadanie polega na obliczeniu wartości dwóch funkcji $f(x) = \sqrt{x^2+1} - 1$ i $g(x) = \frac{x^2}{\sqrt{x^2+1}+1}$ dla $x = 8^{-1}, 8^{-2}, 8^{-3}, ...$ w arytmetyce Float64.

## 6.2 Rozwiązanie

Liczymy wartości poszczególnych funkcji dla $x = 8^{-1}, 8^{-2}, 8^{-3}, ..., 8^{-10}$

### 6.3 Wyniki

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

Widzimy, że początkowo wyniki w miarę się pokrywają, natomiast od $x = 8^{-9}$ funkcja $f(x)$ zaczyna zwracać 0.

### 6.4 Wnioski

Zdecydowanie bardziej wiarygodne są wyniki jakie daję funkcja $g(x)$. Obie te funkcje będą dażyć do 0, natomiast nigdy nie powinny go osiągnąć, co w przypadku funkcji $f(x)$ dzieję się bardzo szybko.

## 7 Zadanie 7

### 7.1 Opis problemu

Zadanie polega na obliczeniu przyliżonej wartości pochodnej funkcji $f(x) = \sin(x) + \cos(3x)$ w punkcie $x_0 = 1$, korzystając ze wzoru $\widetilde{f}'(x_0) = \frac{f(x_0+h)-f(x_0)}{h}$ oraz błędu $|f'(x_0) - \widetilde{f}'(x_0)|$ dla $h = 2^{-n}, n = 0, 1, 2, ..., 54$.

### 7.2 Rozwiązanie

W zadaniu liczymy przybliżone pochodne zgodnie z podanym wyżej wzorem dla danych h, właściwą wartość pochodnej w punkcie $x_0 = 1$ zgodnie ze wzorem $f'(x) = \cos(x) - sin(3x)$, błędy dla poszczególnych przybliżeń pochodnych oraz wyliczamy wartości $h + 1$ dla poszczególnych h.

### 7.3 Wyniki

Prawidłowa wartośc $f'(x) = 0.11694228168853815$.

| $h$ | $\widetilde{f'}(x_0)$ | $h+1$ | $\lvert f'(x_0) - \widetilde{f'}(x_0)\rvert$ |
|---|---|---|---|
| $h = 1$ | 1.8704413979316472 | 1.5 | 1.753499116243109 |
| $h = 2$ | 1.1077870952342974 | 1.25 | 0.9908448135457593 |
| $h = 3$ | 0.6232412792975817 | 1.125 | 0.5062989976090435 |
| $h = 4$ | 0.3704000662035192 | 1.0625 | 0.253457784514981 |
| . | . | . | . |
| . | . | . | . |
| . | . | . | . |
| $h = 26$ | 0.11694233864545822 | 1.0000000149011612 | 5.6956920069239914e-8 |
| $h = 27$ | 0.11694231629371643 | 1.0000000074505806 | 3.460517827846843e-8 |
| $h = 28$ | 0.11694228649139404 | 1.0000000037252903 | 4.802855890773117e-9 |
| $h = 29$ | 0.11694222688674927 | 1.0000000018626451 | 5.480178888461751e-8 |
| . | . | . | . |
| . | . | . | . |
| . | . | . | . |
| $h = 52$ | -0.5 | 1.0000000000000002 | 0.6169422816885382 |
| $h = 53$ | 0.0 | 1.0 | 0.11694228168853815 |
| $h = 54$ | 0.0 | 1.0 | 0.11694228168853815 |

Możemy zauważyć, że wartość $\widetilde{f'}(x_0)$ jest najbliższa wartości rzeczywistej, tzn. błąd jest najmniejszy dla $h = 28$, dla wartości $h > 28$ błąd bezwzględny znów zaczyna wzrastać. Możemy zauważyć również, że wartości $h + 1$ dążą do 1.

### 7.4 Wnioski

Od pewnego momentu zmniejszanie się wartości h nie sprawia, że zaczynamy się zbliżać do poprawnego wyniku, tylko od niego oddalać, dzieje się tak dlatego, że h zaczyna robic się na tyle małe, że jest "pochłaniane" przez 1.0. Z tego samego powodu wartość przybliżonej pochodnej dla $h = 53$ i $h = 54$ wynosi 0.