Sprawozdanie z pierwszej listy zadań na laboratorium *Obliczenia Naukowe*

Paweł Rubin

Październik 2019

1 Zadanie 1

1.1 Opis zadania

Zadanie polegało na wyznaczeniu **iteracyjnie** następujących stałych dla arytmetyki 16, 32 i 64 bitowej oraz porównanie wyników z wywołaniami funkcji w języku **Julia**.

- *epsilon maszynowy* (ang, machine epsilon) - najmniejsza liczba $macheps > 0$ taka, że $fl(1.0 + macheps) > 1.0$
- *eta* - najmniejsza dodatnia liczba w danej arytmetyce
- *MAX* - największa dodatnia liczba w danej arytmetyce

1.2 Rozwiązanie

Wszystkie powyższe stałe zostały wyznaczone **iteracyjnie** poprzez proste funkcje napisane w języku Julia.

epsilon maszynowy został wyznaczony poprzez **dzielenie** przez dwa w pętli, zaczynając od jedynki, dopóki wynik dodany do jedynki zwiększał jej wartość (zgodnie z definicją *macheps*).

```
function get_machine_epsilon(type::Type)
    epsilon = one(type)
    while one(type) + epsilon / 2 > one(type)
        epsilon /= 2
    end
    epsilon
end
```

eta została również wyznaczona poprzez **dzielenie** przez dwa w pętli, zaczynając od jedynki, dopóki wyniki był większy od zera.

```
function get_eta(type::Type)
    eta = one(type)
    while eta / 2 > 0
        eta /= 2
    end
    eta
end
```

MAX został wyznaczony poprzez **mnożenie** przez dwa w pętli, zaczynając od `prevfloat(one(type))`, aby osiągnąć maksymalną mantysę (same jedynki), dzięki czemu poprzez mnożenie przez dwa osiągniemy maksymalną eksponentę, która wraz z maksymalną mantysą da liczbę *MAX*.

```
function get_max(type::Type)
    max = prevfloat(one(type))
    while !isinf(max * 2)
        max *= 2
    end
    max
end
```

1.3 Wyniki

Poniższe tabele przedstawiają wyniki wykonań funkcji napisanych przeze mnie oraz funkcji bibliotecznych.

type	get_machine_epsilon(type)	eps(type)
Float16	0.000977	0.000977
Float32	1.1920929e-7	1.1920929e-7
Float64	2.220446049250313e-16	2.220446049250313e-16

Tabela 1: *epsilon maszynowy*

type	get_eta(type)	next_float(type(0.0))
Float16	6.0e-8	6.0e-8
Float32	1.0e-45	1.0e-45
Float64	5.0e-324	5.0e-324

Tabela 2: *eta*

type	get_max(type)	floatmax(type)	float.h
Float16	6.55e4	6.55e4	-
Float32	3.4028235e38	3.4028235e38	3.4028235e38
Float64	1.7976931348623157e308	1.7976931348623157e308	1.7976931348623157e308

Tabela 3: *MAX*

1.4 Wnioski

Zaimplementowane przeze mnie funkcje zwracają poprawne wyniki, zgodne z funkcjami bibliotecznymi. Liczby w standardzie mają skończoną dokładność, o której należy pamiętać wykonując obliczenia.

2 Zadanie 2

2.1 Opis zadania

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

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

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

2.2 Rozwiązanie

Sprawdziłem eksperymentalnie, poprzez obliczenie owego wyrażenia funkcją napisaną w języku Julia. Następnie wyniki zostały porównane z wartościami `eps(type)` dla kolejnych typów.

```
function kahan_eps(type)
    type(3) * (type(4) / type(3) - one(type)) - one(type)
end
```

2.3 Wyniki

Poniższa tabela przedstawia wyniki wywołań funkcji `kahan_eps(type)` wraz z wartościami `eps(type)` dla kolejnych typów zmiennopozycyjnych.

type	kahan_eps(type)	eps(type)
Float16	-0.000977	0.000977
Float32	1.1920929e-7	1.1920929e-7
Float64	-2.220446049250313e-16	2.220446049250313e-16

Tabela 4: *Kahan epsilon*

2.4 Wnioski

Kahan **prawie** miał rację, ponieważ obliczając owe wyrażenie rzeczywiście dostaniemy *epsilon maszynowy*, ale w arytmetykach `Float16` oraz `Float64` wynik jest ujemny.

3 Zadanie 3

3.1 Opis zadania

Zadanie polegało na sprawdzeniu rozmieszczenia liczb zmiennoprzecinkowych w arytmetyce IEEE 754 podwójnej precyzji w danych przedziałach liczbowych

- $[0.5, 1.0)$
- $[1.0, 2.0)$
- $[2.0, 4.0)$

3.2 Rozwiązanie

Jeżeli eksponenty pierwszej i ostatniej liczby w przedziale są różne, wyklucza to równomierne ich rozmieszczenie.

Jeśli w danym przedziale liczby są równomiernie rozmieszczone to odstęp między nimi możemy obliczyć ze wzoru

$$2^{eksponenta-1023} \cdot 2^{-52}$$

Poniższa funkcja w języku **Julia** implementuje to podejście

```
function get_spread(a :: Float64, b :: Float64)
    a_exponent = SubString(bitstring(a), 2:12)
    prev_b_exponent = SubString(bitstring(prevfloat(b)), 2:12)

    if a_exponent != prev_b_exponent
        return 0.0
    end

    exponent = parse(Int, a_exponent, base = 2)

    return 2.0 ^ (exponent - 1023) * (2.0 ^ (-52))
end
```

3.3 Wyniki

Poniższa tabela przedstawia wykonania funkcji `get_spread(a, b)` dla danych kolejnych przedziałów.

przedział	`get_spread(a, b)`
$[0.5, 1.0)$	1.1102230246251565e-16
$[1.0, 2.0)$	2.220446049250313e-16
$[2.0, 4.0)$	4.440892098500626e-16

Tabela 5: *krok rozmieszczenia*

3.4 Wnioski

W arytmetyce IEEE 754 dokładność reprezentacji różni się między przedziałami. Stały odstęp wynika ze stałej liczby bitów mantysy.

4 Zadanie 4

4.1 Opis zadania

Należało znaleźć eksperymentalnie w arytmetyce `Float64` liczbę zmiennopozycyjną x, taką że:

$$x\left(\frac{1}{x}\right) \neq 1; \text{ tj. } fl\left(xfl\left(\frac{1}{x}\right)\right) \neq 1$$

- w przedziale $(1, 2)$
- najmniejszą taką liczbę

4.2 Rozwiązanie

Liczba x została wyznaczona poprzez użycie funkcji napisanej w języku **Julia**, odpowiednio dla 1 i 0.

```
function find_frantic_number(start)
    num = start(Float64)
    while nextfloat(num) * (one(Float64) / nextfloat(num)) == one(Float64)
        num = nextfloat(num)
    end
    nextfloat(num)
end
```

4.3 Wyniki

Poniższa tabela przedstawia wartości otrzymanych po wywołaniu funkcji `find_frantic_number` odpowiednio dla 1 oraz 0.

`start`	`find_frantic_number(start)`
`one`	1.000000057228997
`zero`	5.0e-324

Tabela 6: *frantic number*

4.4 Wnioski

Brak precyzji w reprezentacji liczb prowadzi do błędnych wyników.

5 Zadanie 5

5.1 Opis zadania

Zadanie polegało na obliczeniu iloczynu skalarnego dwóch wektorów

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

według czterech następujących algorytmów.

- a) "W przód", dodając kolejne iloczyny
- b) "W tył", dodając kolejne iloczyny od końca
- c) dodając iloczyny w kolejności od największego (dodaj dodatnie liczby w porządku od największegodo najmniejszego, dodaj ujemne liczby w porządku od najmniejszego do największego,a następnie daj do siebie obliczone sumy częściow)
- d) dodając iloczyny w kolejności od najmniejszego (przeciwnie do metody (c))

5.2 Rozwiązanie

Powyższe algorytmy zostały zaimplementowane jako proste funkcje w języku **Julia**.

```
function a(x::Vector, y::Vector)
    S = 0.0
    for i = 1:length(x)
        S += x[i] * y[i]
    end
    S
end

function b(x::Vector, y::Vector)
    S = 0
    for i = length(x):-1:1
        S += x[i] * y[i]
    end
    S
end

function c(x::Vector, y::Vector)
    products = map(x -> x[1] * x[2], zip(x, y))
    sumPositives = foldl(+, sort(filter(x -> x >= 0, products), rev=true))
    sumNegatives = foldl(+, sort(filter(x -> x < 0, products)))
    sumNegatives + sumPositives
end

function d(x::Vector, y::Vector)
    products = map(x -> x[1] * x[2], zip(x, y))
    sumPositives = foldl(+, sort(filter(x -> x >= 0, products)))
    sumNegatives = foldl(+, sort(filter(x -> x < 0, products), rev=true))
    sumNegatives + sumPositives
end
```

5.3 Wyniki

Tabele 7 oraz 8 przedstawiają wyniki dla arytmetyki `Float64` oraz `Float32`

algorytm	wynik
a	1.0251881368296672e-10
b	-1.5643308870494366e-10
c	0.0
d	0.0

Tabela 7: Wyniki dla `Float64`

algorytm	wynik
a	-0.3472038161853561
b	-0.4543457
c	-0.5
d	-0.5

Tabela 8: Wyniki dla `Float32`

5.4 Wnioski

Tylko pierwsza metoda w arytmetyce `Float64` osiągnęła poprawny wynik. Zarówno precyzja jak i kolejność operacji ma wpływ na dokładność obliczeń

6 Zadanie 6

6.1 Opis zadania

Zadanie polegało na policzeniu w języku Julia w arytmetyce `Float64` następujących funkcji

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

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

dla kolejnych wartości argumentu x ze zbioru $\{8^{-1}, 8^{-2}, 8^{-3}, \dots\}$.

6.2 Rozwiązanie

Wartości funkcji f oraz g zostały obliczone przez funkcje napisane w języku **Julia**.

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

function g(x)
    x^2 / (sqrt(x^2 + 1) + 1)
end
```

6.3 Wyniki

Poniższa tabela przedstawia wartości funkcji f oraz g dla kolejnych wartości x.

x	$f(8^{-x})$	$g(8^{-x})$
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

Tabela 9: f vs g

6.4 Wnioski

Odejmowanie bliskich sobie liczb powoduje spadek dokładności obliczeń.

7 Zadanie 7

7.1 Opis zadania

Zadanie polegało na obliczeniu przybliżonej wartości pochodnej $f(x)$ w punkcie x za pomocą następującego wzoru

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

dla następującej funkcji f

$$f(x) = \sin x + \cos 3x,$$

której dokładna wartość pochodnej w punkcie $x_0 = 1$ wynosi

$$f'(1) \approx 0.11694228168853805$$

7.2 Rozwiązanie

Zadanie zostało rozwiązane za pomocą funkcji napisanej w języku **Julia**, która dosłownie realizuje powyższy wzór.

```
function derivative(f, 1, h)
    return (f(x0 + h) - f(x0)) / h
end
```

7.3 Wyniki

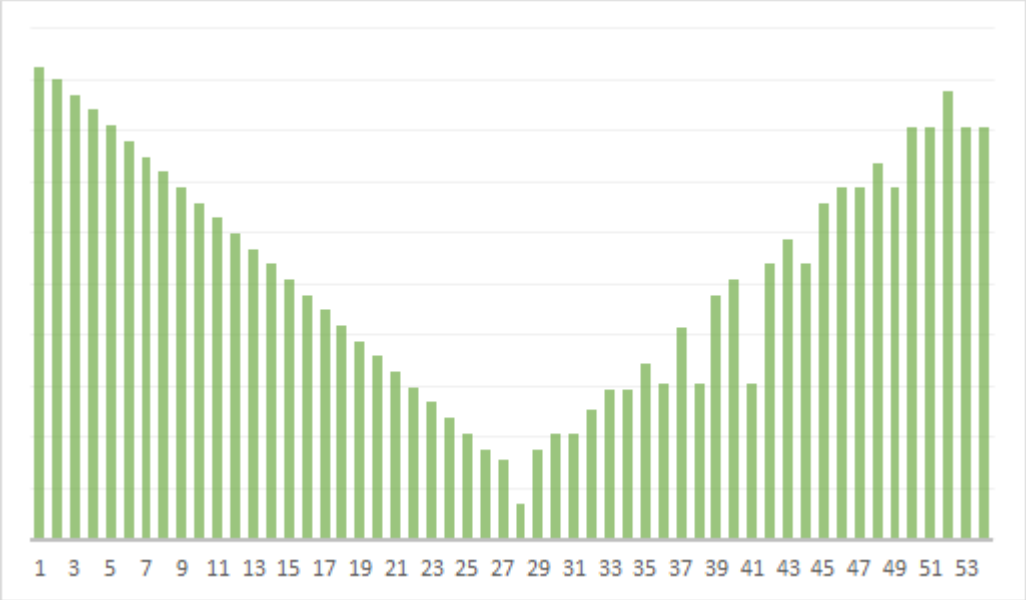
Poniższa tabela przedstawia wartości wywołań funkcji `derivative(f, x0, h)`, gdzie `f = sin(x) + cos(3*x)`, dla kolejnych wartości h ze zbioru $\{2^{-1}, 2^{-2}, 2^{-3}, \ldots, 2^{-54}\}$ wraz z wartościami $1 + h$ oraz błędem bezwzględnym. Ponadto wykres poniżej przedstawia wartości błedu bezwzględnego w zależnosci od wartości i we wzorze $h = 2^{-i}$.

Warto zauważyć, że najlepsze przybliżenie osiągamy dla $h = 2^{-28}$, kolejne wartości h dają coraz większy błąd.

h	$1 + h$	$\tilde{f}'(x)$	błąd
2^{-1}	1.5	1.8704413979316472	1.753499116243109
2^{-2}	1.25	1.1077870952342974	0.9908448135457594
2^{-3}	1.125	0.6232412792975817	0.5062989976090436
2^{-4}	1.0625	0.3704000662035192	0.25345778451498113

2^{-5}	1.03125	0.24344307439754687	0.12650079270900882
2^{-6}	1.015625	0.18009756330732785	0.06315528161878979
2^{-7}	1.0078125	0.1484913953710958	0.031549113682557736
2^{-8}	1.00390625	0.1327091142805159	0.01576683259197785
2^{-9}	1.001953125	0.1248236929407085	0.007881411252170442
2^{-10}	1.0009765625	0.12088247681106168	0.003940195122523624
2^{-11}	1.00048828125	0.11891225046883847	0.00196996878030041
2^{-12}	1.000244140625	0.11792723373901026	0.000984952050472207
2^{-13}	1.0001220703125	0.11743474961076572	0.0004924679222276657
2^{-14}	1.00006103515625	0.11718851362093119	0.00024623193239313446
2^{-15}	1.000030517578125	0.11706539714577957	0.0001231154572415155
2^{-16}	1.0000152587890625	0.11700383928837255	6.155759983449138e-5
2^{-17}	1.0000076293945312	0.11697306045971345	3.077877117539651e-5
2^{-18}	1.0000038146972656	0.11695767106721178	1.538937867372192e-5
2^{-19}	1.0000019073486328	0.11694997636368498	7.69467514692701e-6
2^{-20}	1.0000009536743164	0.11694612901192158	3.847323383529555e-6
2^{-21}	1.0000004768371582	0.1169442052487284	1.9235601903394572e-6
2^{-22}	1.000000238418579	0.11694324295967817	9.61271140118014e-7
2^{-23}	1.0000001192092896	0.11694276239722967	4.807086916164272e-7
2^{-24}	1.0000000596046448	0.11694252118468285	2.394961447910182e-7
2^{-25}	1.0000000298023224	0.116942398250103	1.1656156494177505e-7
2^{-26}	1.0000000149011612	0.11694233864545822	5.695692016638443e-8
2^{-27}	1.0000000074505806	0.11694231629371643	3.4605178375612944e-8
2^{-28}	1.0000000037252903	0.11694228649139404	4.802855987917631e-9
2^{-29}	1.0000000018626451	0.11694222688674927	5.4801788787472994e-8
2^{-30}	1.0000000009313226	0.11694216728210449	1.1440643356286362e-7
2^{-31}	1.0000000004656613	0.11694216728210449	1.1440643356286362e-7
2^{-32}	1.0000000002328306	0.11694192886352539	3.528250126644261e-7
2^{-33}	1.0000000001164153	0.11694145202636719	8.296621708675511e-7
2^{-34}	1.0000000000582077	0.11694145202636719	8.296621708675511e-7
2^{-35}	1.0000000000291038	0.11693954467773438	2.737010803680051e-6
2^{-36}	1.000000000014552	0.116943359375	1.0776864619449489e-6
2^{-37}	1.000000000007276	0.1169281005859375	1.4181102600555051e-5
2^{-38}	1.000000000003638	0.116943359375	1.0776864619449489e-6
2^{-39}	1.000000000001819	0.11688232421875	5.995746978805505e-5
2^{-40}	1.0000000000009095	0.1168212890625	0.00012099262603805505
2^{-41}	1.0000000000004547	0.116943359375	1.0776864619449489e-6
2^{-42}	1.0000000000002274	0.11669921875	0.00024306293853805505
2^{-43}	1.0000000000001137	0.1162109375	0.000731344188538055
2^{-44}	1.0000000000000568	0.1171875	0.00024521831146194495
2^{-45}	1.0000000000000284	0.11328125	0.003661031688538055
2^{-46}	1.0000000000000142	0.109375	0.007567281688538055
2^{-47}	1.000000000000007	0.109375	0.007567281688538055
2^{-48}	1.0000000000000036	0.09375	0.023192281688538055
2^{-49}	1.0000000000000018	0.125	0.008057718311461945
2^{-50}	1.0000000000000009	0.0	0.11694228168853806
2^{-51}	1.0000000000000004	0.0	0.11694228168853806
2^{-52}	1.0000000000000002	-0.5	0.616942281688538
2^{-53}	1.0	0.0	0.11694228168853806
2^{-54}	1.0	0.0	0.11694228168853806

Tabela 10: Wyniki obliczonej pochodnej wraz z błędem bezwzględnym



Rysunek 1: Wartości błędu pochodnej w zależności od i we wzorze $h = 2^{-i}$ w skali logarytmicznej

7.4 Wnioski

Należy unikać operacji na liczbach bardzo bliskich zeru.