Obliczenia naukowe - lista 1

Jacek Zub, 268 490

1 Rozpoznanie arytmetyki

1.1 Epsilon maszynowy

Liczbą ϵ (epsilon) nazywamy najmniejszą liczbą w danej arytmetyce t. że $1.0 + \epsilon > 1.0$. Maszynowym epsilonem (macheps) nazwiemy najmniejszą liczbę zmiennoprzecinkową spełniającą to równanie.
Aby go znaleźć możemy dzielić liczbę 1.0 przez 2 dopóki spełniony jest warunek:

```
function epsilon(type)
    one       = type(1.0)
    macheps   = one

    while one + (macheps / 2) > one
        macheps = macheps / 2
    end

    return macheps
end
```

Otrzymane wartości porównane z wywołaniem eps() oraz wartościami wziętymi z pliku float.h wyglądają w następujący sposób:

Typ liczbowy	Obliczony epsilon	Epsilon z funkcji eps()	Epsilon z float.h
Float16	0.000977	0.000977	brak
Float32	1.1920929e-7	1.1920929e-7	1.1920929e-7
Float64	2.220446049250313e-16	2.220446049250313e-16	2.220446049250313e-16

Kompilator C dostępny na mojej maszynie nie obsługuje liczb zmienno przecinkowych o długości 16 bitów więc nie mogłem wpisać dla nich epsilonu z float.h.
Poza tym, wszystkie wartości jak można się było spodziewać są takie same.

Jaki związek ma liczba macheps z precyzją arytmetyki?
Epsilon maszynowy jest równy jej precyzji arytmetyki.
Precyzja arytmetyki jest zdefiniowana jako $\epsilon = \frac{1}{2}\beta^{1-t}$, gdzie β to baza rozwinięcia (w naszym przypadku 2), t to liczba bitów mantysy (odpowiednio 10, 23 i 52).
Po obliczeniu wychodzą nam wartości:

Float16: $\epsilon = \frac{1}{2}2^{1-10} = 0.000977$
Float32: $\epsilon = \frac{1}{2}2^{1-23} = 1.1920929 * 10^{-7}$
Float64: $\epsilon = \frac{1}{2}2^{1-52} = 2.220446049250313 * 10^{-16}$

Zgadzające się z epsilonem maszynowym.

1.2 Eta maszynowa

Liczbę eta definiujemy jako najmniejszą liczbę, która spełnia nierówność $\eta > 0.0$.
Maszynową etą będzie najmniejsza liczba zmiennoprzecinkowa spełniająca ten warunek.
W celu jej znalezienia wykorzystamy funkcję:

```
function eta(type)
    zero = type(0.0)
    eta  = type(1.0)

    while eta / 2 > zero
        eta = eta / 2
    end

    return eta
end
```

Która dzieli jedynkę przez 2 dopóki spełniony jest warunek.
Otrzymane wartości porównane są z etą otrzymaną przy użyciu wywołania nextfloat(0.0).

Typ liczbowy	Obliczona eta	Eta z nextfloat
Float16	6.0e-8	6.0e-8
Float32	1.0e-45	1.0e-45
Float64	5.0e-324	5.0e-324

Jak widać otrzymane liczby pokrywają się z tymi oczekiwanymi.

Jaki związek ma liczba eta z liczbą MIN_{sub}***?***
Liczba $MIN_{sub} = 2^{1-t} * 2^{c_{min}}$, gdzie t - 1 to liczba bitów użyta do zapisania mantysy a c_{min} to minimalna cecha, $c_{min} = -2^{d-1} + 2$ gdzie d jest liczbą bitów użytych do zapisania cechy.
Dla naszych typów MIN_{sub} będzie równa:

Float16: $MIN_{sub} = 2^{-10} * 2^{-2^{5-1}+2} = 6.0 * 10^{-8}$
Float32: $MIN_{sub} = 2^{-23} * 2^{-2^{8-1}+2} = 1.0 * 10^{-45}$
Float64: $MIN_{sub} = 2^{-51} * 2^{-2^{11-1}+2} = 5.0 * 10^{-324}$

Jak widać $MIN_{sub} = \eta$.

1.3 floatmin

Co zwracają funkcje floatmin(Float32) i floatmin(Float64) i jaki jest związek zwracanych wartości z liczbą MIN_{nor}***?***
Użyjmy tych komend w interpreterze języka julia:

```
julia> floatmin(Float32)
1.1754944f-38

julia> floatmin(Float64)
2.2250738585072014e-308
```

MIN_{nor} jest określone jako $2^{c_{min}}$, co dla Float32 i Float64 daje odpowiednio: $1.1754944 * 10^{-38}$ i $2.2250738585072014 * 10^{-308}$.
Czyli MIN_{nor} jest równy wartością zwracanym przez floatmin().

1.4 Liczba MAX

Liczba MAX jest maksymalną liczbą możliwą do zapisania w danej arytmetyce.
Można ją uzyskać zwiększając jedynkę tak długo jak nie zostanie ona oznaczona jako nieskończoność.

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

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

    add = max / 2
    while !isinf(max + add)
        max = max + add
        add = add / 2
    end

    return max
end
```

Wyniki powyższej funkcji:

Typ liczbowy	Obliczony MAX	MAX z maxfloat	MAX z float.h
Float16	6.55e4	6.55e4	brak
Float32	3.4028235e38	3.4028235e38	3.4028235e38
Float64	1.7976931348623157e308	1.7976931348623157e308	1.7976931348623157e308

Jak widać wyniki pokrywają się, tak samo jak w poprzednim zadaniu brakuje wartości z nagłówka float.h dla typu Float16.

1.5 Wnioski

Korzystając z typu w standardzie IEEE-754 o większej liczbie bitów np. Double/Float64 zamiast Float/Float32 możemy liczyć na większą precyzję, większą liczbę maksymalną możliwą do zaprezentowania tak jak i mniejszą liczbę minimalną.

2 Wzór Kahana

Wzór Kahana wyrażany jest jako: $= 3(\frac{4}{3} - 1)$, funkcja go realizująca jest bardzo prosta:

```
function eps(type)
    return type(3) * (type(4) / type(3) - type(1)) - type(1)
end
```

Wyniki działania tej funkcji wyglądają następująco:

Typ liczbowy	Wynik funkcji	Rzeczywisty epsilon
Float16	-0.000977	0.000977
Float32	1.1920929e-7	1.1920929e-7
Float64	-2.220446049250313e-16	2.220446049250313e-16

Jak widać wartość bezwzględna epislonu pokrywa się z wartością rzeczywistą epsilonu.

3 Rozmieszczenie liczb w arytmetyce Float64

3.1 Implementacja

Funkcja sprawdzająca czy liczby w danym przedziale są rozłożone jednakowo z danym krokiem może być zaimplementowana poprzez sprawdzenie czy każda kolejna liczba jest oddalona od poprzedniej o podany krok. Jednak sprawdzenie wszystkich możliwych liczb było by czasochłonne ograniczyłem liczbę sprawdzanych liczb do 3333, są one wybrane z początku, środka oraz końca przedziału.

```
function check_interval(start::Float64, stop::Float64, step::Float64, iter::Int = 3333)
    if start > stop
        start, stop = stop, start
    end
    middle = (start + stop) / 2

    # Sprawdzenie na początku przedziału
    iter_start::Int = 1
    iter_end::Int   = floor(iter / 3)

    curr = start

    for _ in iter_start:iter_end
        next = curr + step

        if nextfloat(curr) != next
            return false
        end

        curr = next
    end

    # Sprawdzenie w środku przedziału
    iter_start = iter_end + 1
    iter_end   = floor(2 * iter / 3)

    curr = middle

    for _ in iter_start:iter_end
        next = curr + step

        if nextfloat(curr) != next
```

```
            return false
        end

        curr = next
    end

    # Sprawdzenie na końcu przedziału
    iter_start = iter_end + 1
    iter_end   = iter

    curr = stop

    for _ in iter_start:iter_end
        prev = curr - step

        if prevfloat(curr) != prev
            return false
        end

        curr = prev
    end

    return true
end
```

Funkcja ta zwraca false jeśli znajdzie w podanym przedziale liczbę oddaloną od innej liczby o wartość inną niż podany krok.

3.2 Przedział [1.0, 2.0]

Dla przedziału [1.0, 2.0] krok jest równy $\delta = 2^{-52}$ co potwierdziłem włączając powyższą funkcję dla podanych danych.

```
interval::Pair{Float64, Float64} = Pair(1.0, 2.0)
delta::Float64 = 2.0^(-52)

r::Bool = check_interval(interval.first, interval.second, delta)
println("check_interval([1.0, 2.0], 2^-52): ", r ? "OK" : "FAIL")
# result: check_interval([1.0, 2.0], 2^-52): OK
```

Deltę przedziału [A, B] (gdzie A i B ma tą samą cechę) możemy zapisać jako $\delta = \frac{|A-B|}{k}$ gdzie k to liczba bitów mantysy.

3.3 Przedział [0.5, 1.0]

Dla przedziału [0.5, 1.0] delta będzie równa $\delta = 2^{-53}$ co również potwierdziłem moją funkcją.

```
interval = Pair(0.5, 1.0)
 = 2.0^(-53)
```

```
r::Bool = check_interval(interval.first, interval.second, delta)
println("check_interval([0.5, 1.0], 2^-53): ", r ? "OK" : "FAIL")
# result: check_interval([0.5, 1.0], 2^-53): OK
```

3.4 Przedział [2.0, 4.0]

A dla przedziału [2.0, 4.0] delta jest równa $\delta = 2^{-51}$

```
interval = Pair(2.0, 4.0)
delta = (2.0^(-51))

r = check_interval(interval.first, interval.second, delta)
println("check_interval([2.0, 4.0], 2^-51): ", r ? "OK" : "FAIL")
# result: check_interval([0.5, 1.0], 2^-51): OK
```

3.5 Wnioski

Korzystając z liczb zmiennoprzecinkowych wraz ze wzrostem wartości spada ich precyzja, co każdą potęgę 2 nasza dokładność spadnie o połowę, może to być niezamierzony efekt np. przy grach komputerowych z dużym otwartym światem, np. w grze Minecraft przy przekraczaniu koordynatów będących kolejnymi potęgami 2 dochodziło do coraz większych problemów gdzie nasza postać np. teleportowała się zamiast płynnie chodzić.

4 Nierówność x * (1/x) != 1

Należy znaleźć liczbę zmiennoprzecinkową Float64 która spełnia równanie $x * \frac{1}{X}! = 1$ dla przedziału (1.0, 2.0).
Aby znaleźć taką liczbę wystarczy sprawdzać następne liczby zmiennoprzecinkowe aż nie spełnimy naszego warunku.

```
function search(start::Float64, stop::Float64)::Float64
    one::Float64 = 1.0

    if start > stop
        start, stop = stop, start
    end

    curr = start

    while curr < stop
        curr = nextfloat(curr)

        inv = one / curr

        if curr * inv != one
            return curr
        end
    end
```

```
        return 0.0
    end
```

Przy wywołaniu $search(1.0, 2.0)$ otrzymujemy 1.000000057228997 która jest najmniejszą taką liczbą w przedziale (1.0, 2.0).

A więc należy uważać przy korzystaniu z liczb zmiennoprzecinkowych gdyż mogą prowadzić do złych wyników i niedokładności, szczególnie w przypadku dzielenia.

5 Iloczyn skalarny

5.1 Implementacja

Należy zaimplementować cztery różne algorytmy obliczania iloczynu skalarnego dwóch wektorów

Algorytm "w przód"

```
function dotp_a(x::Vector{T}, y::Vector{T}) where T
    S::T = 0.0

    for i in 1:length(x)
        S += x[i] * y[i]
    end

    return S
end
```

Algorytm "w tył"

```
function dotp_b(x::Vector{T}, y::Vector{T}) where T
    S::T = 0.0

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

    return S
end
```

Algorytm "od największego do najmniejszego"

```
function dotp_c(x::Vector{T}, y::Vector{T}) where T
    Ss = T[]

    # Obliczenie sum częściowych.
    for i in 1:length(x)
        push!(Ss, x[i] * y[i])
    end

    # Posortowane sumy dodatnie.
```

```
    Ss_pos = sort(
        filter(
            x -> x > 0,
            Ss
        ),
        rev = true)

    # Posortowane sumy ujemne.
    Ss_neg = sort(
        filter(
            x -> x < 0,
            Ss
        ),
        rev = false)

    return sum(Ss_pos) + sum(Ss_neg)
end
```

Algorytm "od najmniejszego do największego"

```
function dotp_d(x::Vector{T}, y::Vector{T}) where T
    Ss = T[]

    # Obliczenie sum częściowych.
    for i in 1:length(x)
        push!(Ss, x[i] * y[i])
    end

    # Posortowane sumy dodatnie.
    Ss_pos = sort(
        filter(
            x -> x > 0,
            Ss
        ),
        rev = false)

    # Posortowane sumy ujemne.
    Ss_neg = sort(
        filter(
            x -> x < 0,
            Ss
        ),
        rev = true)

    return sum(Ss_pos) + sum(Ss_neg)
end
```

5.2 Wyniki

Na podanych algorytmach przeprowadziłem testy dla liczb typu Float32 i Float64 dla wektorów:
$x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$
$y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$
Wyniki wyglądają następująco:

Algorytm	wynik dla Float32	wynik dla Float64
Wartość prawdziwa	-1.006571e-11	-1.00657107e-11
W przód	-0.4999443	1.0251881368296672e-10
W tył	-0.4543457	-1.5643308870494366e-10
Najw. do najm.	-0.5	0.0
Najm. do najw.	-0.5	0.0

Jak widać różnica jest spora, szczególnie w przypadku 3 i 4 algorytmu oraz liczb o mniejszej precyzji.

5.3 Wnioski

Nawet przy w miarę prostych algorytmach mogą wkraść się spore problemy przez niedokładność liczb zmiennoprzecinkowych oraz kolejność też ma znaczenie przy korzystaniu z nich.

6 Wartości równych funkcji

6.1 Implementacja

W zadaniu tym należy zaimplementować w arytmetyce Float64 dwie równe sobie funkcje:

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

oraz policzyć je dla argumentu $x = 8^{-1}, 8^{-2}, 8^{-3}, ...$
Implemenntacja ich jest oczywista:

```
function f(x::Float64)
    one::Float64 = 1.0

    return sqrt(x^2 + one) - one
end

function g(x::Float64)
    one::Float64 = 1.0

    return x^2 / (sqrt(x^2 + one) + one)
end
```

A program je uruchamiający:

```
base::Float64 = 8.0

for exp in -1:-1:-10
```

```
    x::Float64 = base^exp

    println("x = $x")
    println(" f(x) = $(f(x))")
    println(" g(x) = $(g(x))")
    println()
end
```

6.2 Wyniki

Wyniki uruchomienia programu:

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

Początkowe wartości są do siebie zbliżone, ale pomimo tego że funkcje matematycznie są sobie równe to dla wystarczająco małego argumentu (od 8^{-9}) funkcja f zwraca jedynie 0. Wartości zwracane przez g są bardziej wiarygodne.

6.3 Wnioski

Przekształcenie funkcji do matematycznie równej postaci, ale różnej w arytmetyce liczb zmiennoprzecinkowych może poprawić ich precyzję.

7 Przybliżona wartość pochodnej

7.1 Implementacja

Możemy przybliżyć wartość pochodnej funkcji $f(x)$ w punkcie x_0 za pomocą wzoru

$$f`(x_0) \approx \widetilde{f}`(x_0) = \frac{f(x_0 + h) - f(x_0)}{h}$$

Implementacja tego wzoru wygląda w następujący sposób:

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

Musimy obliczyć przybliżoną wartość pochodnej funkcji $f(x) = sinx + cos3x$ w punkcie x_0 oraz błędy $|f`(x_0) - \widetilde{f}`(x_0)|$ dla różnicy $h = 2^{-n} (n = 0, 1, 2, ..., 54)$.
Do obliczenia wartości $f`(x_0)$ posłuży nam pochodna $f`(x) = cosx - 3sinx$

```
f       = x::Float64 -> sin(x) + cos(3 * x)
f_prime = x::Float64 -> cos(x) - 3 * sin(3 * x)
```

7.2 Wyniki

Po uruchomieniu programu wyniki wyglądają w następujący sposób:

n	$\widetilde{f}`(x_0)$	$\lvert f`(x_0) - \widetilde{f}`(x_0)\rvert$
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	5.6956920069239914e-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
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

Jak wytłumaczyć, że od pewnego momentu zmniejszanie wartości h nie poprawia przybliżenia wartości pochodnej?

Najmniejszy błąd uzyskaliśmy przy $n = 28$. Zmniejszając tracimy na dokładności ponieważ operujemy na wartościach o bardzo różnym wykładniku do momentu gdy $f(1) = f(1 + h)$ i h przestaje mieć zupełnie znaczenie.

Jak zachowują się wartości 1 + h?

Wyniki uruchomienia funkcji f dla 1+h:

n	1 + h	f(x)
0	2.0	1.8694677134760478
1	1.5	0.7866991871732747
2	1.25	0.12842526201602544
3	1.125	-0.0706163518803512
4	1.0625	-0.12537150765482896
5	1.03125	-0.14091391571762557
6	1.015625	-0.1457074873658719
7	1.0078125	-0.14736142276621222
8	1.00390625	-0.14800311681489065
9	1.001953125	-0.1482777155172741
10	1.0009765625	-0.1484034624987881
11	1.00048828125	-0.14846344917024967
12	1.000244140625	-0.14849272096399935
13	1.0001220703125	-0.14850717649596556
14	1.00006103515625	-0.14851435917330935
15	1.000030517578125	-0.14851793924014578
16	1.0000152587890625	-0.1485197264556457
17	1.0000076293945312	-0.14852061935892114
18	1.0000038146972656	-0.1485210656344409
19	1.0000019073486328	-0.14852128872817139
20	1.0000009536743164	-0.14852140026402927
21	1.0000004768371582	-0.1485214560292064
22	1.000000238418579	-0.1485214839111071
23	1.0000001192092896	-0.1485214978518853
24	1.0000000596046448	-0.14852150482223148
25	1.0000000298023224	-0.14852150830739386
26	1.0000000149011612	-0.14852151004997227
27	1.0000000074505806	-0.14852151092126076
28	1.0000000037252903	-0.14852151135690494
29	1.0000000018626451	-0.14852151157472704
30	1.0000000009313226	-0.14852151168363803
31	1.0000000004656613	-0.14852151173809347
32	1.0000000002328306	-0.14852151176532125
33	1.0000000001164153	-0.14852151177893513
34	1.0000000000582077	-0.14852151178574202
35	1.0000000000291038	-0.14852151178914552
36	1.000000000014552	-0.14852151179084716
37	1.000000000007276	-0.14852151179169815
38	1.000000000003638	-0.14852151179212347
39	1.000000000001819	-0.1485215117923363
40	1.0000000000009095	-0.14852151179244266
41	1.0000000000004547	-0.14852151179249573
42	1.0000000000002274	-0.14852151179252238
43	1.0000000000001137	-0.1485215117925357
44	1.0000000000000568	-0.14852151179254225
45	1.0000000000000284	-0.1485215117925457

46	1.0000000000000142	-0.14852151179254736
47	1.000000000000007	-0.14852151179254813
48	1.0000000000000036	-0.14852151179254858
49	1.0000000000000018	-0.1485215117925487
50	1.0000000000000009	-0.1485215117925489
51	1.0000000000000004	-0.1485215117925489
52	1.0000000000000002	-0.14852151179254902
53	1.0	-0.1485215117925489
54	1.0	-0.1485215117925489

Jak widać od pewnego momentu h + 1 = 1 więc nasze przybliżenie zawsze zwróci 0.

7.3 Wnioski

Należy uważać przy korzystaniu z bardzo małych liczb zmiennoprzecinkowych oraz gdy operujemy na liczbach o bardzo różnych cechach.