Obliczenia naukowe
Sprawozdanie lista 1

Wojciech Wróblewski 250349

October 2020

Zad1

Opis problemu oraz terminologia

Napisać program w języku Julia wyznaczający iteracyjnie epsilony maszynowe, maszynową liczbę eta oraz liczbę MAX, dla wszystkich dostępnych typów zmiennopozycyjnych Float16, Float32, Float64, zgodnych ze standardem IEEE 754. Wyniki należało porównać z wartościami bibliotecznymi.

- Epsilon maszynowy - Jest to największa liczba nieujemna, której dodanie do jedności daje wynik równy 1.
- Liczba eta to taka liczba, że eta > 0.0 oraz jest to pierwsza liczba większa od 0.0 w zadanej arytmetyce.
- Max jest to największa dodatnia liczba w danej arytmetyce.

Rozwiązanie

Wyznaczenie liczb umożliwią nam funkcje napisaje w języku Julia.

```
function get_machine_epsilon(_type)

    eps = _type(1.0)
    while _type(1.0) + eps / _type(2.0) > _type(1.0)
        eps /= 2
    end

    return eps
end


function get_eta(_type)

    eta = _type(1.0)
```

```
    while eta / _type(2.0) > _type(0.0)
        eta /= _type(2.0)
    end

    return eta
end


function get_max(_type)

    max = prevfloat(_type(1.0))
    while !isinf(max * 2)
        max *= 2
    end

    return max
end
```

Wyniki

Wyniki doświadczenia zostały zaprezentowane w tabelach poniżej.

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

_type	get_eta()	*eta*()
Float16	6.0e-8	6.0e-8
Float32	1.0e-45	1.0e-45
Float64	5.0e-324	5.0e-324

_type	get_max()	floatmax()
Float16	6.55e4	6.55e4
Float32	3.4028235e38	3.4028235e38
Float64	1.7976931348623157e308	1.7976931348623157e308

_type	eps(float.h)	floatmax(float.h)
Float16	-	-
Float32	1.1920929e-7	3.4028235e38
Float64	2.220446049250313e-16	1.7976931348623157e308

Wnioski

Zastosowane algorytmy w poprawny sposób generują wskazane stałe, co jesteśmy w stanie udowodnić porównując uzyskane wyniki do wyników otrzymywanych z funkcji bibliotecznych.
Licza eta to MIN_{sub} z wykładu czyli najmniejsza liczba nieznormalizowana. Funkcje wbudowane floatmin(Float32) oraz floatmin(Float64) zwracają najmniejszą znormalizowaną liczbę zadanych arytmetyk, czyli liczby MIN_{nor} . Epsilon maszynowy (macheps) jest dwa razy większy od wartości precyzji arytmetyki.

Zad2

Opis problemu

W zadaniu należy zweryfikować eksperymentalnie twierdzenie Kahana mówięgo o tym, że epsilon maszynowy (macheps) można otrzymać obliczając wyrażenie $3\left(\frac{4}{3}-1\right)-1$ w arytmetyce zmiennopozycyjnej. Twierdzenie należy zweryfikować dla typów zmiennopozycyjnych Float16, Float32,Float64.

Rozwiązanie

Weryfikację metody Kahana możemy przeprowadzić implementując prosty program w języku Julia.

```
function get_eps(_type)
     _type(3.0) * (_type(4.0)/_type(3.0) -_type(1.0)) -_type(1.0)
end
```

Wyniki

Wyniki doświadczenia zostały zaprezentowane w tabelach poniżej.

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

Wnioski

Obserwujemy, że dla trypu Float32 wyrażenie Kahana wylicza poprawną wartość epsilonu maszynowego, jednak w reszcie przypadków zauważamy rozbieżno-

ści i liczby zgadzają się do co wartości bezwzględnej. Powodem tych rozbieżności jest liczba $\frac{4}{3}$, która nie posiada skończonyego rozwinięcia w IEEE754, co w zależności od stosowanej arytmetyki może zwracać niepoprawne wyniki.

Zad3

Opis problemu

Sprawdź eksperymentalnie w języku Julia, że w arytmetyce Float64(arytmetyce double w standarcie IEEE 754) liczby zmiennopozycyjne są równomiernie rozmieszczone w przedziale [1,2] z krokiem $\delta = 2^{-52}$.

Rozwiązanie

```
# touple przechowujace zadane przedzialy
bounds = [(Float64(0.5),Float64(1.0))
          (Float64(1.0),Float64(2.0))
          (Float64(2.0),Float64(4.0))]


#zwraca delte na podstawie wartosci dolnej
#granicy przedzialu

function get_delta(a)
    return nextfloat(a) - a
end

# zwraca t kolejnych bitstringow zaczynajac
# od dolnej granicy przedzialu a

function show_bitstrings_lowerbound(a,b, t)
    delta = get_delta(a)
    for k in 1:t
        x = Float64(a) + k * delta
        println(bitstring(x))
    end
end

# zwraca t kolejnych bitstringow zaczynajac
# od gornej granicy przedzialu b

function show_bitstrings_upperbound(a, b, t)
    delta = get_delta(a)
```

```
    for k in 1:t
        x = Float64(b) - k * delta
        println(bitstring(x))
    end
end
```

Wyniki

Przedział [0.5, 1.0]

0011111111100000000000000000000000000000000000000000000000000001
0011111111100000000000000000000000000000000000000000000000000010
0011111111100000000000000000000000000000000000000000000000000011
0011111111100000000000000000000000000000000000000000000000000100
0011111111100000000000000000000000000000000000000000000000000101
0011111111100000000000000000000000000000000000000000000000000110
.
.
.
0011111111101111111111111111111111111111111111111111111111111011
0011111111101111111111111111111111111111111111111111111111111010
0011111111101111111111111111111111111111111111111111111111111001
0011111111101111111111111111111111111111111111111111111111111000
0011111111101111111111111111111111111111111111111111111111110111
0011111111101111111111111111111111111111111111111111111111110110

Przedział [1.0, 2.0]

0011111111110000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000011
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000101
0011111111110000000000000000000000000000000000000000000000000110
.
.
.
0011111111111111111111111111111111111111111111111111111111111011
0011111111111111111111111111111111111111111111111111111111111010
0011111111111111111111111111111111111111111111111111111111111001
0011111111111111111111111111111111111111111111111111111111111000
0011111111111111111111111111111111111111111111111111111111110111
0011111111111111111111111111111111111111111111111111111111110110

Przedział [2.0, 4.0]

0100000000000000000000000000000000000000000000000000000000000001
0100000000000000000000000000000000000000000000000000000000000010
0100000000000000000000000000000000000000000000000000000000000011
0100000000000000000000000000000000000000000000000000000000000100
0100000000000000000000000000000000000000000000000000000000000101
0100000000000000000000000000000000000000000000000000000000000110
.
.
.
0100000000001111111111111111111111111111111111111111111111111011
0100000000001111111111111111111111111111111111111111111111111010
0100000000001111111111111111111111111111111111111111111111111001
0100000000001111111111111111111111111111111111111111111111111000
0100000000001111111111111111111111111111111111111111111111110111
0100000000001111111111111111111111111111111111111111111111110110

Wartości δ dla zadanych różnych 3 przedziałów.

przedział	δ
[0.5, 1.0]	1.1102230246251565e-16
[1.0, 2.0]	2.220446049250313e-16
[2.0, 4.0]	4.440892098500626e-16

Wnioski

Dla różnych przedziałów liczbowych mamy inną wartość kroku δ między kolejnymi liczbami. Widzimy, że liczby są równomiernie rozłożone, gdyż końce przedziałów mają taki sam bit znaku jak i eksponenty a przesuwanie ich o wartość δ zwiększa je o 1. Wiemy również, że w każdym z przedziałów mamy tyle samo liczb ze względu na stałą liczbę znaków w mantysie danej reprezentacji. Zauważamy jednak, że wraz ze wzrostem liczb na jakich pracujemy odstępy między kolejnymi liczbami reprezentacji zwiększają się co widzimy po różnych wartościach kroku δ dla róznych przedziałów.

Zad4

Opis problemu

(a) Znajdź eksperymentalnie w arytmetyce Float64 zgodnej ze standardem IEEE 754(double) liczbę zmiennopozycyjną w przedziale $1 < x < 2$, taką, że $x(1/x) \neq 1$ tj. $fl(xfl(1/x)) \neq 1$ (napisz program w języku Julia znajdujący tę liczbę).

(b) Znajdź najmniejszą taką liczbę.

Rozwiązanie

Implementacja krótkich algorytmów wyliczających te liczby. W celu znalezienia liczby z podpunktu a) wykorzystujemy algorytm get_number(). Aby odszukać najmniejszą globalnie liczbę, która jest rozwiązaniem równania jako wartość początkową w pętli podstawiamy zdefiniowaną w języku Julia wartość -Inf.

```
function get_number()

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

    return a
end

function get_smallest_number()

    a = nextfloat(-Inf)
    while nextfloat(a) * (Float64(1.0) / nextfloat(a)) == Float64(1.0)
        a = nextfloat(a)
    end
    return a
end
```

Wyniki

a) 1.0000000572289969 (zarazem najmniejsza w przedziale (1,2))
b) -1.7976931348623157e308 (najmniejsza globalnie)

Wnioski

Obserwujemy, że mimo elementarnego problemu jakim jest zadane równanie, komputer znajduje rozwiązania, które są błędne. Obserwujemy tutaj problem reprezentacji oraz zaokrągleń niektórych liczb w pamięci komputera. Niekiedy powoduje to duże utraty dokładności, które możemy zaobserwować na powyższym przykładzie.

Zad5

Opis problemu

Napisz program w języku Julia realizujący następujący eksperyment obliczania iloczynu skalarnego dwóch wektorów x i y .Zaimplementuj poniższe algorytmy i policz sumę na cztery sposoby dla $n = 5$:

Wykorzystywane cztery sposoby na obliczenie sumy.

(a) "w przód" - dodając iloczyny odpowiednich współrzędnych od początku

(b) "w tył " - dodając iloczyny odpowiednich współrzędnych od końca

(c) Od największego do najmniejszego (dodaj dodatnie liczby w porządku od największegodo najmniejszego, dodaj ujemne liczby w porządku od najmniejszego do największego,a następnie daj do siebie obliczone sumy częściowe)

(d) Od najmniejszego do największego (przeciwnie do metody (c)).

Rozwiązanie

Wyznaczenie liczb umożliwią nam funkcje napisaje w języku Julia.

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

#liczy iloczyn skalarny z podpunktu a

function dot_product_forward(x,y,_type)
  result =_type(0)

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

#liczy iloczyn skalarny z podpunktu c

function dot_product_reverse(x,y,_type)

  result =_type(0)
```

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

  return result
end

#liczy iloczyn skalarny z podpunktu c

function dot_product_c(x,y,_type)
    pos = zeros(_type, 0)
    neg = zeros(_type, 0)

    for i in 1:length(x)
        temp = _type(x[i] * y[i])
        if temp > 0
            push!(pos,temp)
        else
            push!(neg,temp)
        end
    end
    pos = sort(pos, rev = true)
    neg = sort(neg)

    return sum(pos) + sum(neg)

end

#liczy iloczyn skalarny z podpunktu d

function dot_product_d(x,y,_type)
    pos = zeros(_type, 0)
    neg = zeros(_type, 0)

    for i in 1:length(x)
        temp = _type(x[i] * y[i])
        if temp > 0
            push!(pos,temp)
        else
            push!(neg,temp)
        end
    end

    pos = sort(pos)
    neg = sort(neg, rev = true)
```

```
    return sum(pos) + sum(neg)
end
```

Wyniki

Prawidłowa wartość iloczynu skalarnego (dokładność do 15 cyfr):

- $-1.00657107000000 \cdot 10 - 11$

Wyniki doświadczenia zostały zaprezentowane w tabelach poniżej.

podpunkt	Float32	Float64
a)	-0.2499443	1.0251881368296672e-10
b)	-0.2043457	-1.5643308870494366e-10
c)	-0.25	0.0
d)	-0.25	0.0

Wnioski

Zauważamy, że w żadnej z metod sumowania nie otrzymaliśmy poprawnego wyniku. Obserwujemy, że metody sumowania w przód i odpowiednia metoda w tył nie gwarantuje tego samego wyniku przy stałej arytmetyce, co pokazuje, brak przemienności dodawania w testowanych arytmetykach. Metody sumowania z podpunktów c) oraz d) pokazują, że sortowanie a potem dodawanie liczb o znacząco różnych rzędach wprowadza bardzo dużą utratę dokładności obliczeń.

Zad6

Opis problemu

Policz w języku Julia w arytmetyce Float64 wartości następujących funkcji :

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

dla kolejnych wartości argumentu $x = 8^{-1}, 8^{-2}, 8^{-3}, 8^{-4}, ...$ Chociaż f=g komputer daje różne wyniki. Wskaż, które z nich są wiarygodne, a które nie?

Rozwiązanie

Wyznaczenie liczb umożliwią nam funkcje napisaje w języku Julia.

```
function get_x(x,_type)
    return _type(1.0/8^x)
end

function func_f(x,_type)
    return sqrt(x^2+1.0)-1.0
end

function func_g(x,_type)
    return x^2/(sqrt(x^2+1.0)+1.0)
end
```

Wyniki

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
$8^{-}11$	0.0	6.776263578034403e-21
$8^{-}12$	0.0	1.0587911840678754e-22
$8^{-}13$	0.0	1.6543612251060553e-24
$8^{-}14$	0.0	2.5849394142282115e-26
$8^{-}15$	0.0	4.0389678347315804e-28

Wnioski

Mimo, że funkcje są sobie równe na podstawie wyników z doświadczenia otrzymujemy różne wyniki. Już od argumentu $x = 8^{-1}$ wyniki stają się rozbieżne i wraz ze spadkiem wartości argumentu x, różnice te stają się bardziej zauważalne. Funkcja f(x) okazała się być mniej precyzyjna, gdyż dla 9-tego z kolei argumentu zaczęła zaokrąglać wynik do 0.0. Wynika to z tego ,że w funkcji g(x) nie występuje odejmowanie, które powoduje utratę cyfr znaczących i zarazem mniej dokładne rezultaty.

Zad7

Opis problemu

Obliczyć przybliżoną wartość pochodnej f(x) w punkcie x za pomocą następującegowzoru .

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

Zadanie przeprowadzamy dla funkcji. $f(x) = sin(x) + cos(3x)$

Rozwiązanie

W rozwiązaniu implementujemy funkcje obliczające nam wartość funkcji h(n) oraz f(x) w punkcie. Implementujemy również funkcję derivitive_approx() jak i derivitive obliczające odpowiednio przybliżoną jak i dokładną wartość pochodnej funkcji w punkcie.

```
function func_f(x)
    return Float64(sin(x) + cos(3x))
end

function func_h(n)
    return Float64(1/2^n)
end

function derivitive_aprox(x_0,h)
    return Float64(Float64(func_f(x_0 + h)-func_f(x_0))/Float64(h))
end

function derivitive(x_0)
    return Float64(cos(x_0) - 3*sin(3*x_0))
end

function error(x_0,h,derivitive_x0)
    return abs(derivitive_aprox(x_0,h)-derivitive_x0)
end
```

Wyniki

Dokładna wartość pochodnej funkcji f(x) w punkcie $x_0 = 1$ wynosi 0.11694228168853815

n	h(n) =2^-n	f('x)	h + 1	error
1	2^-1	1.87044139793165	1.5	1.75349911624311
2	2^-2	1.1077870952343	1.25	0.990844813545759
3	2^-3	0.623241279297582	1.125	0.506298997609044
4	2^-4	0.370400066203519	1.0625	0.253457784514981
5	2^-5	0.243443074397547	1.03125	0.126500792709009
6	2^-6	0.180097563307328	1.015625	0.06315528161879
7	2^-7	0.148491395371096	1.0078125	0.031549113682558
8	2^-8	0.132709114280516	1.00390625	0.015766832591978
9	2^-9	0.124823692940709	1.001953125	0.00788141125217
10	2^-10	0.120882476811062	1.0009765625	0.003940195122524
11	2^-11	0.118912250468838	1.00048828125	0.0019699687803
12	2^-12	0.11792723373901	1.000244140625	0.000984952050472
13	2^-13	0.117434749610766	1.0001220703125	0.000492467922228
14	2^-14	0.117188513620931	1.00006103515625	0.000246231932393
15	2^-15	0.11706539714578	1.00003051757813	0.000123115457241
16	2^-16	0.117003839288373	1.00001525878906	6.15575998343942E-05
17	2^-17	0.116973060459713	1.00000762939453	3.07787711752994E-05
18	2^-18	0.116957671067212	1.00000381469727	1.53893786736248E-05
19	2^-19	0.116949976363685	1.00000190734863	7.69467514682987E-06
20	2^-20	0.116946129011922	1.00000095367432	3.84732338343241E-06
21	2^-21	0.116944205248728	1.00000047683716	1.92356019024231E-06
22	2^-22	0.116943242959678	1.00000023841858	9.6127114002087E-07
23	2^-23	0.11694276239723	1.00000011920929	4.80708691519283E-07
24	2^-24	0.116942521184683	1.00000005960464	2.39496144693874E-07
25	2^-25	0.116942398250103	1.00000002980232	1.16561564844631E-07
26	2^-26	0.116942338645458	1.00000001490116	5.69569200692399E-08
27	2^-27	0.116942316293716	1.00000000745058	3.46051782784684E-08
28	2^-28	0.116942286491394	1.00000000372529	4.80285589077312E-09
29	2^-29	0.116942226886749	1.00000000186265	5.48017888846175E-08
30	2^-30	0.116942167282105	1.00000000093132	1.14406433660008E-07
31	2^-31	0.116942167282105	1.00000000046566	1.14406433660008E-07
32	2^-32	0.116941928863525	1.00000000023283	3.52825012761571E-07
33	2^-33	0.116941452026367	1.00000000011642	8.29662170964696E-07
34	2^-34	0.116941452026367	1.00000000005821	8.29662170964696E-07
35	2^-35	0.116939544677734	1.0000000000291	2.7370108037772E-06
36	2^-36	0.116943359375	1.00000000001455	1.0776864618478E-06
37	2^-37	0.116928100585938	1.00000000000728	1.41811026006522E-05
38	2^-38	0.116943359375	1.00000000000364	1.0776864618478E-06
39	2^-39	0.11688232421875	1.00000000000182	5.99574697881522E-05
40	2^-40	0.1168212890625	1.00000000000091	0.000120992626038
41	2^-41	0.116943359375	1.00000000000045	1.0776864618478E-06
42	2^-42	0.11669921875	1.00000000000023	0.000243062938538
43	2^-43	0.1162109375	1.00000000000011	0.000731344188538
44	2^-44	0.1171875	1.00000000000006	0.000245218311462
45	2^-45	0.11328125	1.00000000000003	0.003661031688538
46	2^-46	0.109375	1.00000000000001	0.007567281688538
47	2^-47	0.109375	1.00000000000001	0.007567281688538
48	2^-48	0.09375	1	0.023192281688538
49	2^-49	0.125	1	0.008057718311462
50	2^-50	0	1	0.116942281688538
51	2^-51	0	1	0.116942281688538
52	2^-52	-0.5	1	0.616942281688538
53	2^-53	0	1	0.116942281688538
54	2^-54	0	1	0.116942281688538

Wnioski

Można zaobserwować, że błąd obliczany w tablicy wyników jest najmniejszy w okolicy połowy rozpatrywanych argumentów. Dokładnie jest to przypadek, gdy wartość gunkcji $h(n) = 2^{-28}$ .Widzimy, że od pewnego momentu wartość h+1 rozpoczęła zaokrąglać się do 1 przez to, że różnica rzędów pomiędzy 1 a wartością funkcji h jest bardzo duża. Dla kolejnych argumentów błąd względem dokładnej wartości pochodnej zaczyna rosnąć. Na zaburzenie dokładności wyływa dodawanie liczb różniących się rzędem jak i odejmowanie bliskich sobie wartości funkcji f dla coraz mniejszych wartości funkcji $h(n)$.