Obliczenia Naukowe

Sprawozdanie 1

Jakub Czyszczonik

1 Zadanie 1

1.1 Opis Problemu

Wyznaczyć iteracyjnie zero maszynowe (eps), precyzję arytmetyki (eta) oraz max dla typów zmiennoprzecinkowych zgodnych ze standardem IEEE 754.

1.2 Rozwiązanie

```
function eps(Type)
    eps = Type(1.0)
    while 1 + eps / 2 > 1
        eps = eps / 2
    end
    return eps
end
```

Kod. 1: Zero maszynowe

```
function eta(Type)
    eta = Type(1.0)
    while eta / 2 > 0
        eta = eta / 2
    end
    return eta
end
```

Kod. 2: Precyzja Arytmetyki

```
function max(Type)
    max = Type(2.0)
    while isinf(max * 2) == false
        max = max * 2
    end

    completment = Type(max / 2)
    while isinf(max + completment) == false
        max = max + completment
        completment = completment / 2
    end
    return max
end
```

Kod. 3: Maksimum

1.3 Wyniki

Typ Zmiennopozycyjny	Eps Iteracyjny	Eps Lib.	Eta Iteracyjna	Eta Lib.	Max Iteracyjny	Max Lib.
Float16	0.000977	0.000977	6.0e-8	6.0e-8	6.55e4	6.55e4
Float32	1.1920929e-7	1.1920929e-7	1.0e-45	1.0e-45	3.4028235e38	3.4028235e38
Float64	2.220446049 250313e-16	2.220446049 250313e-16	5.0e-324	5.0e-324	1.79769313 48623157e308	1.79769313 48623157e308

Wartości znalezione w pliku float.h są identyczne z otrzymanymi w zadaniu.

1.4 Wnioski

Implementacja liczb zmiennoprzecionkowych jest zgodna ze standardem IEEE 754, jak również wyniki nie odbiegają od tych otrzymanych za pomocą metod analitycznych na ćwiczeniach.

2 Zadanie 2

2.1 Opis Problemu

Sprawdzić eksperymentalnie w języku Julia poprawność twierdzenia Kahana, które mówi, że epsilon maszynowy można wyznaczyć za pomocą wzoru:

$$macheps = 3(\frac{4}{3} - 1) - 1 \tag{1}$$

2.2 Rozwiązanie

```
function kahanEps(Type)
    return 3 * (Type(4/3) - 1) - 1
end
```

Kod. 4: Twierdzenie Kahana

2.3 Wyniki

Typ Zmiennopozycyjny	Eps z Tw. Kahana	Eps Lib.
Float16	-0.000977	0.000977
Float32	1.1920929e-7	1.1920929e-7
Float64	-2.220446049250313e-16	2.220446049250313e-16

2.4 Wnioski

Komputer przechowuje liczby w systemie binarnym, więc wynik obliczenia wyrażenia 4/3 musi zostać zaokrąglony co powoduje utratę dokładności poprzez przybliżenia, co przekład się bezpośrednio na wynik obliczeń.

3 Zadanie 3

3.1 Opis Problemu

Sprawdzić w języku Julia, że w arytmetyce Float64 liczby zmiennopozycyjne są równomiernie rozmieszczone w danych zakresach.

3.2 Rozwiązanie

```
function numberDistributionChecker(min,max)
    delta = Float64(nextfloat(min) - min)
    println("Checking schedule for interval: (",min, ",",max,")")
    println("Delta: 2^",log2(delta))
    #Power of number 2; Representation of Mantissa
    k = Float64(1.0)
    currentFloat = min
    while currentFloat <= max
        #Computing current Float
        currentFloat = Float64(currentFloat + k * delta)
        #Measuring previous step size
        prevStep = currentFloat - prevfloat(currentFloat)
        # If the previous step is not equal to the delta, the precision has
 been changed.
        if prevStep != delta
            println("Delta Changed! 2^", log2(prevStep), " for x = ",
currentFloat)
        end
        println(bits(min + k * delta), " ", min + k * delta)
        k = Float64(k + 1.0)
    end
    println("Delta: 2^",log2(delta)," for interval: (",min, ",",max,")")
end
```

Kod. 5: Sprawdzanie gęstości rozsawienia liczb

3.3 Wyniki

Zakres	Delta
[1;2]	2^{-52}
[0.5;1]	2^{-53}
[2,4]	2^{-51}

Binarnie	Liczba
0011111111110000000000000000000000000000000000000000000000000001	1.0000000000000002
0011111111110000000000000000000000000000000000000000000000000010	1.0000000000000004
0011111111110000000000000000000000000000000000000000000000000011	1.0000000000000007

W lewej tabeli możemy zobaczyć deltę dla rozkładu liczb w danych zakresach, natomiast po prawej stronie znajdują się 3 kolejne liczby z jego reprezentacją binarną, można zauważyć, że kolejne kroki powodują zwiększenie mantysy o 1 przy każdym kroku.

3.4 Wnioski

Wieksze liczby są zapisywane z mniejszą precyzją w arytmetyce Float64, o czym może świadczyć większa delta (odległość między kolejnymi liczbami). Co również bezpośrednio wynika z tego, że przedziały te są potęgami liczby dwa, dlatego z każdym przedziałem delta maleje dwókrotnie.

4 Zadanie 4

4.1 Opis Problemu

Znaleźć liczbę zmiennopozycyjną (w arytmetyce Float64) x w przedziale 1 < x < 2 taką, że

$$x(\frac{1}{x}) \neq 1 : fl(x * fl(\frac{1}{x})) \neq 1 \tag{2}$$

oraz znaleźć najmniejszą taką liczbę spełniającą powyższe równania.

4.2 Rozwiązanie

```
function number(numba)
    x = nextfloat(numba)
    while Float64(x * Float64(1/x)) == 1
        #Searching step by step.
        x = x + Float64(nextfloat(x) - x)
    end
    println("x = ", x)
end
```

Kod. 6: Zero maszynowe

4.3 Wyniki

Liczba z przedziału (1,2)	1.000000057228997
Najmniejsza liczba	-1.7976931348623157e308

4.4 Wnioski

Liczby takie istnieją, ponieważ przy zaokrąglaniu liczby powoduje utracenie precyzji, w skutek czego zachodzi powyższa własność.

5 Zadanie 5

5.1 Opis Problemu

Napisać program obliczający iloczyn skalarny dwóch zadanych wektorów:
x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]
y = 1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]

5.2 Rozwiązanie

```
function Forward(Type)
    n = 5
    x = Type[2.718281828, -3.141592654, 1.414213562, 0.5772156649,
0.3010299957]
    y = Type[1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]
    Sum = Type(0.0)
    for iteration = 1 : n
        Sum = Sum + Type(x[iteration]*y[iteration])
    end
    println("Forward Sum = ", Sum," for ",Type)
 end
```

Kod. 7: Sumowanie od pierwszego do ostatniego

```
 function Backward(Type)
    n = 5
    x = Type[2.718281828, -3.141592654, 1.414213562, 0.5772156649,
0.3010299957]
    y = Type[1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]
    Sum = Type(0.0)
    iteration = n
    while iteration != 0
        Sum = Sum + Type(x[iteration]*y[iteration])
        iteration = iteration - 1
    end
    println("Backward Sum = ", Sum," for ",Type)
 end
```

Kod. 8: Sumowanie od ostatniego do pierwszego

```
1  function Ascending(Type)
2      n = 5
3      x = Type[2.718281828, -3.141592654, 1.414213562, 0.5772156649,
   0.3010299957]
4      y = Type[1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]
5      Array = zeros(Type,n)
6
7      for index = 1 : n
8        Array[index] = x[index]*y[index]
9      end
10
11     sort!(Array)
12     Positive = Type(0.0)
13     for index = 1 : n
14       if Array[index] > 0
15         Positive = Positive + Array[index]
16       end
17     end
18
19     sort!(Array, rev=true)
20     Negative = Type(0.0)
21     for index = 1 : n
22       if Array[index] < 0
23         Negative = Negative + Array[index]
24       end
25     end
26
27     Total = Positive + Negative
28     println("Ascending = ", Total," for ",Type)
29 end
```

Kod. 9: Od największego do najmniejszego

Obliczyć sumy częściowe – zsumować liczby dodatnie od największej do najmniejszej oraz liczby ujemne od najmniejszej do największej, po czym dodać je do siebie. Sumowanie od najmniejszego do największego możemy stworzyć poprzez zamianę ze Sobą lini 11 i 19.

5.3 Wyniki

Typ	do przodu	do tyłu	od największego	od najmniejszego
Float32	-0.4999443	-0.4543457	-0.5	-0.5
Float64	1.0251881368296672e-10	-1.5643308870494366e-10	0.0	0.0

5.4 Wnioski

Najdokładniejszy jest sposób do tyłu dla Float64 dający najbliższy wynik w stosunku do rzeczywistej wartości czyli -1.00657107000000e-11. Przy dodawaniu posortowanych liczb widoczna była strata precyzji przez pochłonięcia mniejszej liczby przez większą.

6 Zadanie 6

6.1 Opis Problemu

Policzyć w arytmetyce Float64 wartości następujących funkcji:

$$f(x) = \sqrt{x^2 + 1} - 1 \tag{3}$$

$$g(x) = \frac{x^2}{\sqrt{x^2 + 1} + 1} \tag{4}$$

dla x $= 8^{-1}, 8^{-2}, 8^{-3}, \ldots$

6.2 Rozwiązanie

```
function f(x)
    return sqrt(x^2.0 + 1.0) - 1.0
end
function g(x)
    return x^2.0 / (sqrt(x^2.0 + 1.0) + 1.0)
end
for i = 1 : 180
    x = f(Float64(8.0^-i))
    y = g(Float64(8.0^-i))
    println("Iteration = ", i, "  f(x) = ", x, "  g(x) = ", y)
end
```

Kod. 10: Obliczenie wartości funkcji dla zadanych x

6.3 Wyniki

```
Iteration = 6  f(x) = 7.275957614183426e-12  g(x) = 7.275957614156956e-12
Iteration = 7  f(x) = 1.1368683772161603e-13  g(x) = 1.1368683772160957e-13
Iteration = 8  f(x) = 1.7763568394002505e-15  g(x) = 1.7763568394002489e-15
Iteration = 9  f(x) = 0.0  g(x) = 2.7755575615628914e-17
Iteration = 10  f(x) = 0.0  g(x) = 4.336808689942018e-19
..........................................................................
Iteration = 176  f(x) = 0.0  g(x) = 6.4758e-319
Iteration = 177  f(x) = 0.0  g(x) = 1.012e-320
Iteration = 178  f(x) = 0.0  g(x) = 1.6e-322
Iteration = 179  f(x) = 0.0  g(x) = 0.0
Iteration = 180  f(x) = 0.0  g(x) = 0.0
```

I

Output. 11: Output z konsoli

6.4 Wnioski

Można zauważyć, że funkcja g jest o wiele dokładniejsza od f, ponieważ dla wykładnika -9 funkcja f zeruje się, natomiast funkcja g przy wykładniku -179. Pomimo, że f = g

7 Zadanie 7

7.1 Opis Problemu

Obliczyć przybliżoną wartość pochodnej $f(x) = \sin x + \cos 3x$ w punkcie $x = 1$ oraz obliczyć błędy między otrzymanymi wartościami, a rzeczywistym wynikiem pochodnej w punkcie $x = 1$

7.2 Rozwiązanie

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

n = 0
while n <= 54
  h = Float64(2.0^-n)
  approximation = ddx(sin, 1, h) + ddx(cos, 3, h)
  println("Approximation = ", approximation, " for h = ", h,"\nh h summed with big number: ",h+1,"\napprox. error = ", abs(0 - approximation))
  n = n + 1
end
```

Kod. 12: Liczenie przybliżenia pochodnej i błędu przybliżenia.

7.3 Wyniki

```
Approximation = 0.390625 for h = 7.105427357601002e-15
h summed with big number: 1.000000000000007
approx. error = 0.390625
Approximation = 0.375 for h = 3.552713678800501e-15
h summed with big number: 1.0000000000000036
approx. error = 0.375
Approximation = 0.375 for h = 1.7763568394002505e-15
h summed with big number: 1.0000000000000018
approx. error = 0.375
Approximation = 0.25 for h = 8.881784197001252e-16
h summed with big number: 1.0000000000000009
approx. error = 0.25
Approximation = 0.25 for h = 4.440892098500626e-16
h summed with big number: 1.0000000000000004
approx. error = 0.25
Approximation = 0.5 for h = 2.220446049250313e-16
h summed with big number: 1.0000000000000002
approx. error = 0.5
Approximation = 0.0 for h = 1.1102230246251565e-16
h summed with big number: 1.0
approx. error = 0.0
```

Output. 13: Liczenie przybliżenia pochodnej i błędu przybliżenia.

7.4 Wnioski

Od h = 2.220446049250313e-16 przybliżenie nie ulega zmianie, ponieważ h jest już tak małe, że zostaje wchłonięte przez większe liczby.