Michał Budnik – Sprawozdanie 1

# Wstęp

Lista pierwsza przedstawia zadania, które są podstawą obliczeń naukowych. Skupia się ona na sprawdzaniu podstawowych limitów komputacji na maszynach cyfrowych obsługujących standard IEEE754.

# Zadania

## Zadanie 1

### 1.1 Opis zagadnienia

Zadanie polega na sprawdzeniu podstawowych precyzji standardów Float16, Float32, oraz Float64. W zadaniu należy wyznaczyć iteracyjnie epsilon maszynowy, liczbę eta, oraz (MAX)dla powyższych typów Float.

### 1.2 Rozwiązanie

Zaimplementowane rozwiązania polegają na iteracyjnym wyznaczaniu odpowiednich wartości.

function machepsCheck()

fl = 1

fl\_prev = 1

while 1 + FloatXX(fl) > 1

fl\_prev = fl

fl /= FloatXX(2)

end

end

function etaCheck()

fl = 1

fl\_prev = 1

while FloatXX(fl) > 0

fl\_prev = fl

fl /= FloatXX(2)

end

end

function maxCheck()

fl = 1

fl\_prev = 1

while !isinf(FloatXX(fl))

fl\_prev = fl

fl += FloatXX(fl\_prev)

end

fl = fl\_prev

fl\_add = fl\_prev/2

while fl+fl\_add != fl

fl\_prev = fl

fl = FloatXX(fl+fl\_add)

fl\_add = FloatXX(fl\_add/2)

end

Ostatnia metoda znajduje poprawną wartość, ponieważ w dowolnym momencie dodanie podwojonej wartości **MAX\_add** skutkowałoby otrzymaniem inf (warunek z pierwszej pętli while).

### 1.3 Wyniki

Float16: macheps is 0.000977

Float16: eps() is 0.000977

Float32: macheps is 1.1920929e-7

Float32: eps() is 1.1920929e-7

Float64: macheps is 2.220446049250313e-16

Float64: eps() is 2.220446049250313e-16

Float16: eta is 6.0e-8

Float16: nextfloat() is 6.0e-8

Float32: eta is 1.0e-45

Float32: nextfloat() is 1.0e-45

Float64: eta is 5.0e-324

Float64: nextfloat() is 5.0e-324

Float16: max is 6.55e4

Float16: realmax is 6.55e4

Float32: max is 3.4028235e38

Float32: realmax is 3.4028235e38

Float64: max is 1.7976931348623157e308

Float64: realmax is 1.7976931348623157e308

### 1.4 Wnioski

Wartości te są w pełni zależne od możliwej reprezentacji liczb w standardzie IEEE754. Wartości w tym standardzie są wyznaczane za pomocą wzoru: , gdzie S oznacza bit znaku, M wartość mantysy, E to wartość cechy, a X to stała zależna od standardu. Łatwo zauważyć dlaczego liczba eta jest o wiele mniejsza niż epsilon maszynowy. Zbliżając się do 1 dysponujemy tylko bitami mantysy którymi możemy manipulować, ponieważ 20 = 1, a następna wartość to 21 = 2. Jednakże zbliżając się do 0 mamy do dyspozycji w pierwszej kolejności bity cechy, których pomiędzy wartością 0 a 1 jest 5 dla Float16, 7 dla Float32, oraz 11 dla Float64 (11 bitów oznacza precyzję rzędu 2-1023). Warto zauważyć, że domyślnie zmienia się wygląd mantysy – przy obliczaniu liczb bliskich 0 zamienia się poprzedzającą jedynkę mantysy na zero. Daje to wyższą precyzję wartości bliskich zeru, ponieważ precyzja jest powiększona o kolejne |M| bitów, i jest to nazywane wartością subnormalną.

Wartość MAX jest zależna w większości od wykładnika, jednakże bity mantysy też mają tutaj wpływ na maksymalną wartość.

## Zadanie 2

### 2.1 Opis zagadnienia

Experymentalne sprawdzenie czy obliczenie wyrażenia (tw. Kahana) dla różnych standardów typu Float zwraca epsilon maszynowy danego standardu.

### 2.2 Rozwiązanie

print("Float16: macheps is $((Float16(3)\*  
 ((Float16(4)/Float16(3))-Float16(1)))-Float16(1))")

### 2.3 Wyniki

Float16: macheps is -0.000977

Float16: eps() is 0.000977

Float32: macheps is 1.1920929e-7

Float32: eps() is 1.1920929e-7

Float64: macheps is -2.220446049250313e-16

Float64: eps() is 2.220446049250313e-16

### 2.4 Wnioski

Wynik wyrażenia (z dokładnością co do znaku) pokazuje epsilon maszynowy w danym standardzie. Analizując to wyrażenie krok po kroku:

> 4/3 = 1 + 1/3:   
 0011111111110101010101010101010101010101010101010101010101010101 (1.3...25931846502499)  
> 4/3 – 1 = 1/3:  
 0011111111010101010101010101010101010101010101010101010101010100 (0.3…25931846502499)  
> 1/3 \* 3 = 1:  
 0011111111101111111111111111111111111111111111111111111111111110 (0. 9...777955395074969)  
> 1 - 1 = 0:  
 1011110010110000000000000000000000000000000000000000000000000000 (-2.22044...E-16)

Jak można zauważyć z otrzymanej reprezentacji bitowej chodzi tutaj o błędne reprezentowanie liczby 1/3 w systemie binarnym.

## Zadanie 3

### 3.1 Opis zagadnienia

Udowodnienie, że dla przedziału [1, 2] liczby w arytmetyce Float64 standardu IEEE754 są równomiernie rozmieszczone z dokładnością:

### 3.2 Rozwiązanie

Żeby udowodnić ten fakt wykorzystana jest wiedza a priori – iteracyjnnie dodawana jest δ i pokazywane jest, że zmienia to kolejno liczby po jednym bicie:

x=1

while x!=2

x += 2.0^-53

print("$(bits(Float64(x)))\n")

end

### 3.3 Wyniki

...

0011111111110000000000000000000000000000000000000001001110000000

0011111111110000000000000000000000000000000000000001001110000001

0011111111110000000000000000000000000000000000000001001110000010

0011111111110000000000000000000000000000000000000001001110000011

0011111111110000000000000000000000000000000000000001001110000100

0011111111110000000000000000000000000000000000000001001110000101

0011111111110000000000000000000000000000000000000001001110000110

0011111111110000000000000000000000000000000000000001001110000111

...

### 3.4 Wnioski

Fakt, że jest najmniejszą reprezentowalną wartością w przedziale [1, 2] w standardzie Float64 wynika bezpośrednia z definicji w IEEE754:

Gdzie (dla przedziału [1, 2]) |M| = 52, E = 1023, S = 0. Najmniejszy możliwy przyrost to dodanie wartości równej wartości ostatniego bitu mantysy, w tym przypadku: Dla przedziału [0.5, 1] , natomiast dla przedziału [2, 4] .

## Zadanie 4

### 4.1 Opis zagadnienia

Experymentalne wyznaczenie najmniejszej liczby z przedziału (1, 2) w standardzie Float64 IEEE754 takiej, że .

### 4.2 Rozwiązanie

x = 1

while Float64(x\*Float64(1/(x))) == 1

x += Float64(2.0^-52)

end

### 4.3 Wyniki

1.000000057228997

0011111111110000000000000000000000001111010111001011111100101010

### 4.4 Wnioski

W wyniku operacji dzielenia, a następnie mnożenia, zachodzi dla tej cyfry zaokrąglenie które wykracza poza precyzje standardu, stąd wynik nie jest idealnie reprezentowany.

## Zadanie 5

### 5.1 Opis zagadnienia

Napisać programy obliczające iloczyn dwóch 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

x = [2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]

y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]

n = 5

function a()

SUM\_32 = 0

SUM\_64 = 0

for i = 1:n

SUM\_32 += Float32(x[i]) \* Float32(y[i])

SUM\_64 += Float64(x[i]) \* Float64(y[i])

end

end

function b()

SUM\_32 = 0

SUM\_64 = 0

for i = 1:n

SUM\_32 += Float32(x[n+1-i]) \* Float32(y[n+1-i])

SUM\_64 += Float64(x[n+1-i]) \* Float64(y[n+1-i])

end

end

function c()

results\_32 = []

results\_64 = []

for i = 1:n

push!(results\_32, Float32(x[i])\*Float32(y[i]))

push!(results\_64, Float64(x[i])\*Float64(y[i]))

end

sort!(results\_32)

sort!(results\_64)

SUM\_NEG\_32 = 0

SUM\_POS\_32 = 0

SUM\_NEG\_64 = 0

SUM\_POS\_64 = 0

counter = 1

while results\_32[counter] < 0

SUM\_NEG\_32 += results\_32[counter]

counter += 1

end

counter = n

while results\_32[counter] > 0

SUM\_POS\_32 += results\_32[counter]

counter -= 1

end

counter = 1

while results\_64[counter] < 0

SUM\_NEG\_64 += results\_64[counter]

counter += 1

end

counter = n

while results\_64[counter] > 0

SUM\_POS\_64 += results\_64[counter]

counter -= 1

end

end

function d()

results\_32 = []

results\_64 = []

for i = 1:n

push!(results\_32, Float32(x[i])\*Float32(y[i]))

push!(results\_64, Float64(x[i])\*Float64(y[i]))

end

sort!(results\_32)

sort!(results\_64)

SUM\_NEG\_32 = 0

SUM\_POS\_32 = 0

SUM\_NEG\_64 = 0

SUM\_POS\_64 = 0

middle\_32 = 1

middle\_64 = 1

while results\_32[middle\_32]<0

middle\_32 +=1

end

while results\_64[middle\_64]<0

middle\_64 +=1

end

counter = middle\_32-1

while counter > 0

SUM\_NEG\_32 += results\_32[counter]

counter -= 1

end

counter = middle\_32

while counter <=n

SUM\_POS\_32 += results\_32[counter]

counter += 1

end

counter = middle\_64-1

while counter > 0

SUM\_NEG\_64 += results\_64[counter]

counter -= 1

end

counter = middle\_64

while counter <=n

SUM\_POS\_64 += results\_64[counter]

counter += 1

end

end

### 5.3 Wyniki

Method A, Float32: -0.4999443

Method A, Float64: 1.0251881368296672e-10

Method B, Float32: -0.4543457

Method B, Float64: -1.5643308870494366e-10

Method C, Float32: -0.5

Method C, Float64: 0.0

Method D, Float32: -0.5

Method D, Float64: 0.0

### 5.4 Wnioski

Najdokładniejszą metodą okazała się metoda B. Przewaga nad metodami C oraz D jest widoczna, i polega na operacjach bliżej wartości 0, gdzie precyzja jest większa. Przewaga nad metodą A prawdopodobnie wynika wyłącznie z kolejności liczb w zadanych macierzach.

## Zadanie 6

### 6.1 Opis zagadnienia

Policzyć w standardzie Float64 wartości tożsamych funkcji:

dla

### 6.2 Rozwiązanie

function f(x)

return sqrt(x^2+1)-1

end

function g(x)

return x^2/(sqrt(x^2+1)+1)

end

for i = 1:10

print("f(8^-$i) = $(f(8.0^-i)), || g(8^-$i) = $(g(8.0^-i))\n\n")

end

### 6.3 Wyniki

f(8^-1) = 0.0077822185373186414, || g(8^-1) = 0.0077822185373187065

f(8^-2) = 0.00012206286282867573, || g(8^-2) = 0.00012206286282875901

f(8^-3) = 1.9073468138230965e-6, || g(8^-3) = 1.907346813826566e-6

f(8^-4) = 2.9802321943606103e-8, || g(8^-4) = 2.9802321943606116e-8

f(8^-5) = 4.656612873077393e-10, || g(8^-5) = 4.6566128719931904e-10

f(8^-6) = 7.275957614183426e-12, || g(8^-6) = 7.275957614156956e-12

f(8^-7) = 1.1368683772161603e-13, || g(8^-7) = 1.1368683772160957e-13

f(8^-8) = 1.7763568394002505e-15, || g(8^-8) = 1.7763568394002489e-15

f(8^-9) = 0.0, || g(8^-9) = 2.7755575615628914e-17

f(8^-10) = 0.0, || g(8^-10) = 4.336808689942018e-19

### 6.4 Wnioski

Ta sama funkcja zdefiniowana na różne sposoby może zwracać różne wartości w zależności od operacji które muszą zostać wykonane. W podanym zagadnieniu funkcja cechuje się wyższą dokładnością, ponieważ odejmowanie bardzo zbliżonych (co do wartości) liczb ma niską precyzję.

## Zadanie 7

### 7.1 Opis zagadnienia

Obliczyć i porównać wartości funkcji pochodnej ze wzoru na przybliżoną wartość , gdzie , w miejscu

### 7.2 Rozwiązanie

function f(x)

return Float64(sin(x)) + Float64(cos(3\*x))

end

function fDer(x)

return Float64(cos(x)) + Float64(-3\*sin(3\*x))

end

function approxDerivative(any\_func::Function, x0, h)

return (Float64(f(x0+h)) - Float64(f(x0)))/Float64(h)

end

x0 = 1

for i = 0:54

result = approxDerivative(f, x0, 2.0^-i)

print("Val for h = 2 ^ -$i: $result.\n Error: $(fDer(x0) - result)\n\n")

end

### 7.3 Wyniki

Val for h = 2 ^ -0: 2.0179892252685967.

Error: -1.9010469435800585

Val for h = 2 ^ -1: 1.8704413979316472.

Error: -1.753499116243109

Val for h = 2 ^ -2: 1.1077870952342974.

Error: -0.9908448135457593

Val for h = 2 ^ -3: 0.6232412792975817.

Error: -0.5062989976090435

...

Val for h = 2 ^ -27: 0.11694231629371643.

Error: -3.460517827846843e-8

Val for h = 2 ^ -28: 0.11694228649139404.

Error: -4.802855890773117e-9

Val for h = 2 ^ -29: 0.11694222688674927.

Error: 5.480178888461751e-8

...

Val for h = 2 ^ -51: 0.0.

Error: 0.11694228168853815

Val for h = 2 ^ -52: -0.5.

Error: 0.6169422816885382

Val for h = 2 ^ -53: 0.0.

Error: 0.11694228168853815

Val for h = 2 ^ -54: 0.0.

Error: 0.11694228168853815

### 7.4 Wnioski

Od pewnego momentu dalsze zmniejszanie wartości h nie zwiększa precyzji wyników, a nawet ją zmniejsza, ponieważ działania na bardzo małych liczbach w precyzji fl może przynieść ze sobą większy błąd – np. dzielenie przez bardzo małą liczbę.